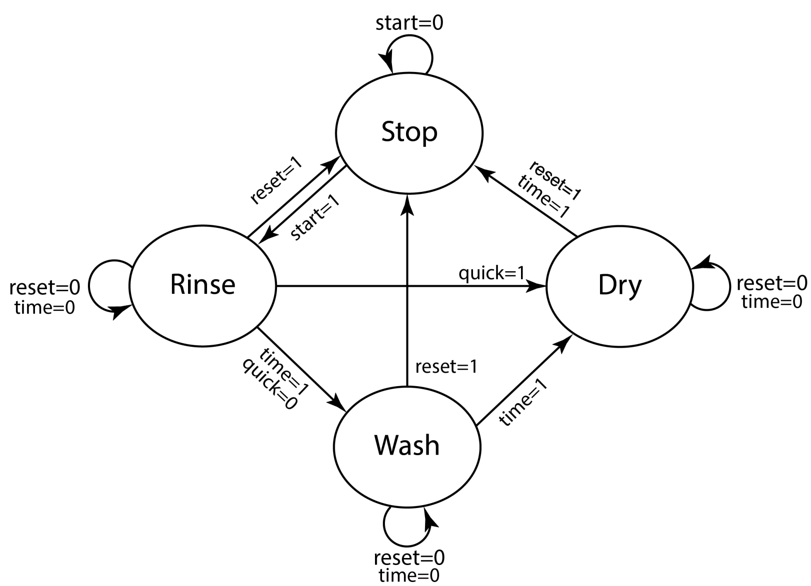
EECS 151/251A Discussion 4

Saturday 17^{th} February, 2024

Problem 1. FSM Design (HW4 Spring 4)

Consider a simple dishwasher that has an option for a quick rinse cycle or a full wash cycle. The dishwasher will first rinse the dishes with hot water, then decide whether to wash with detergent or skip straight to drying depending on the setting. If you open the door of the dishwasher prematurely, a door switch will trigger a `reset` and the dishwasher will stop and go back to the initial state. A timer raises a `time` flag when the dishwasher is ready to move to the next state. The dishwasher has three outputs: `water`, `detergent`, `heat`. During the rinse cycle, it will request water. During the wash cycle, it will request water and detergent. During the dry cycle it will request only heat. Here is a conceptual state transition diagram for the dishwasher:



For each of the following scenarios, please provide:

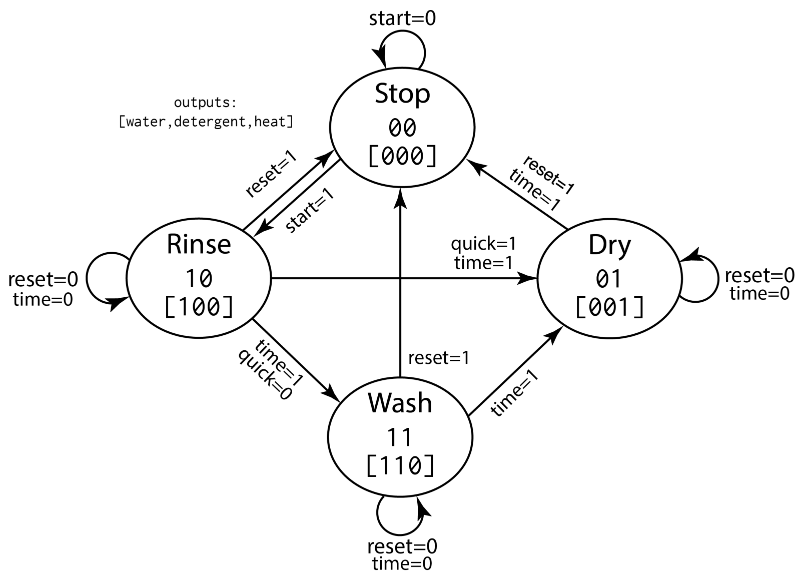
- A state transition diagram with the state names and encoding (e.g. `STOP (00)`), as well as outputs labeled appropriately.
- A circuit diagram of the state machine. The state machine will receive the inputs `reset`, `start`, `quick`, and `time`. The state machine must have the outputs `water`, `detergent`, and `heat`. You may use logic gates of 4 inputs or fewer as well as multiplexers to implement your next state logic.
- A quick (1-2 sentence) summary of your design process and decisions.

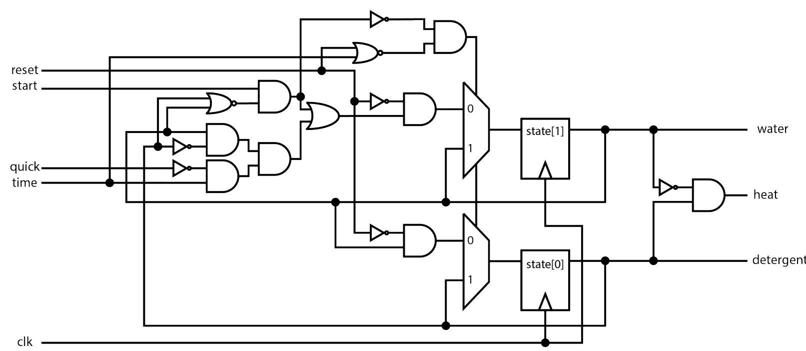
Design the FSM for each of these scenarios:

(a) Binary-encoded Moore Machine

Solution:

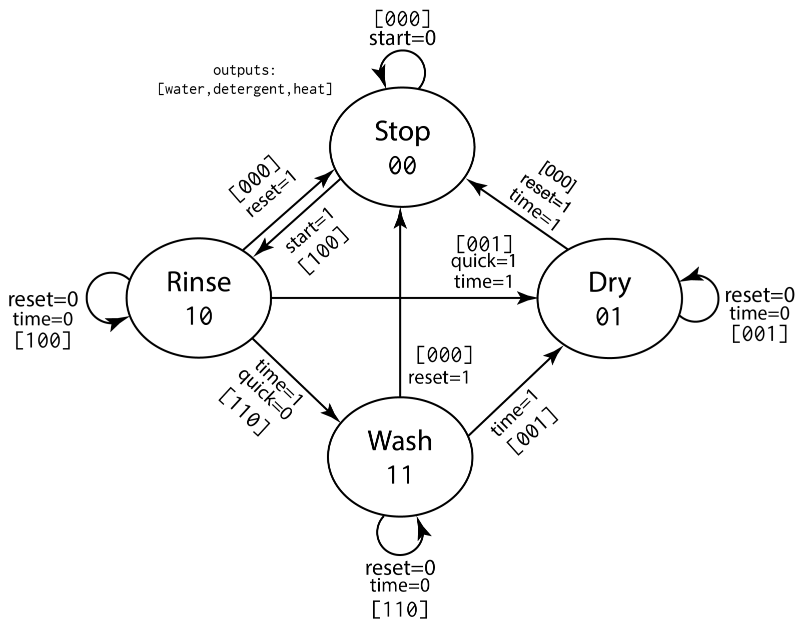
There are many approaches to this problem. In this case, I have purposely encoded my states so that the output logic is relatively simple.

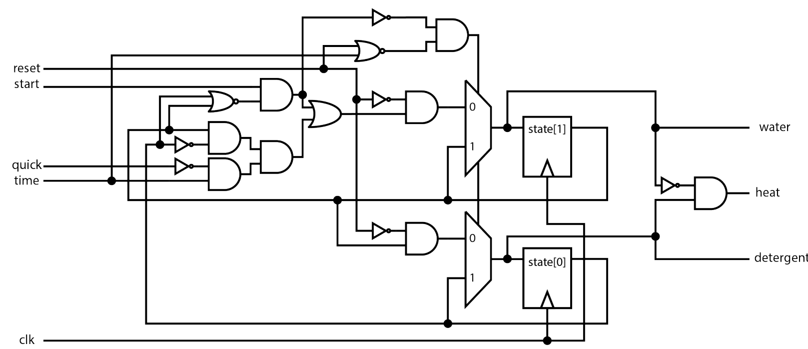




(b) Binary-encoded Mealy Machine

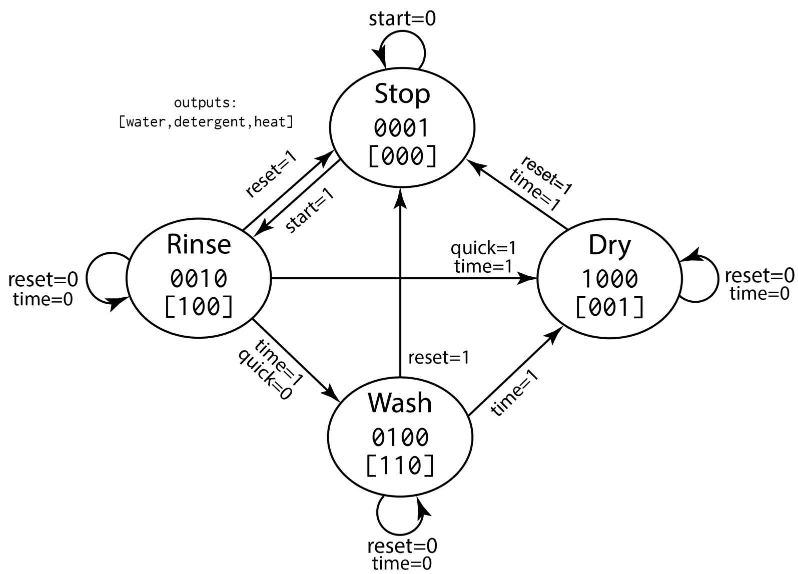
Solution:

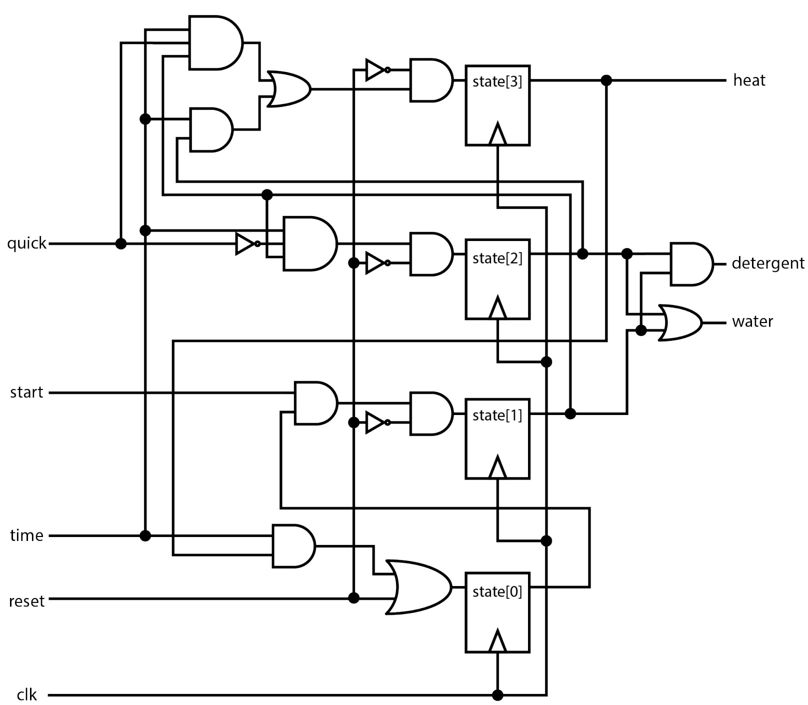




(c) One-hot Moore Machine

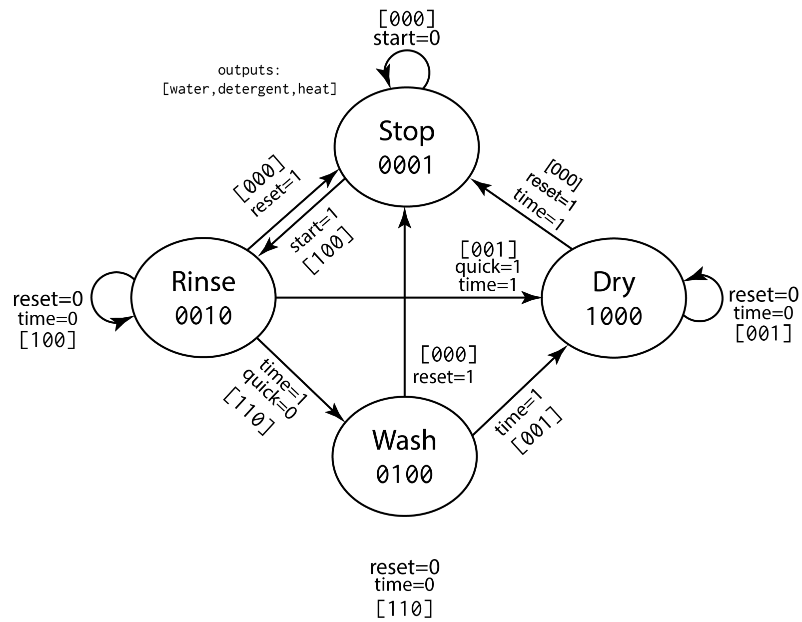
Solution:

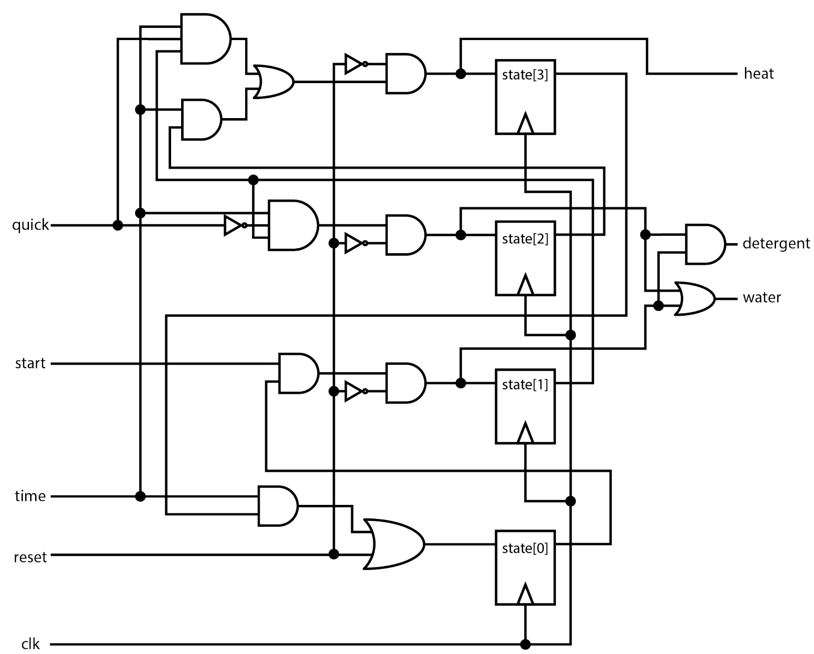




(d) One-hot Mealy Machine

Solution:





Problem 1: Extra: FSMs in Verilog

FSMs are ubiquitous in hardware. Being able to write a FSM in verilog is a necessary skill. For an educational opportunity, we present a real word use of a FSM: power up sequence for a system. Electronic systems often can not instantly power on. In fact, complex have a specific sequence powering up different subsystems in order to ensure correct functionality and prevent damage to the system. An example could be a drone. Below we example the two popular styles of writing an FSM in Verilog. Furthermore, we demonstrate the different between the Moore and Mealy. Our simplified power up sequnce has the following specification:

1. The FSM is idle when the system is off
2. The power up sequence begins when the *pwr_on* signal is asserted
3. Before the rails are brought up (powered up) one must wait a full clock cycle
4. Once up, the FSM will being to power down with the *pwr_off* signal is asserted
5. The system has 3 rails which are brought up when the corresponding output is asserted

Below is the older style of writing an FSM with the combinational and sequential logic separately. Note this is also a Moore state machine.

```
/*
 * This module is a Moore FSM which style where sequential
 * and combinational logic are keep separate. A single register
 * represents the sequential logic. The always block is completely
 * combinational using blocking statements. This was common back in
 * the day for clarity and to reduce ambiguity for the synthesis tool.
 * CAD tools are much improved and there is no advantage to
 * writing FSMs like this, it is purely a stylistic decision.
 */
module power_on_seq_moore(
  input clk,
  input rst,
  //
  input pwr_on,
  input pwr_off,
  //
  output rail_1,
  output rail_2,
  output rail_3);

  // Constants
  // State Variables
  localparam IDLE=0;
  localparam WARM_UP=1;
  localparam VDD_ON=3;
  localparam PWR_DOWN=2;


  // Signals
  reg [1:0] nxt_state;
  wire [1:0] state;

  // Instantiations
  REGISTER_R #(2) state_reg(.clk(clk),
                            .rst(rst),
                            .ce(1'b1),
                            .d(nxt_state),
                            .q(state));

  always @(*) begin
    rail_1 = 1'b0;
    rail_2 = 1'b0;
    rail_3 = 1'b0;
    nxt_state = IDLE;

    case (state)
      // Idle state
      IDLE: begin
              if (pwr_on == 1'b1) begin
                nxt_state = WARM_UP;
              end
```

```
            end

      // One cycle wait state
      WARM_UP: nxt_state = VDD_ON;

      // Bring up Rails
      VDD_ON: begin
                if (pwr_off == 1'b1) begin
                  nxt_state = PWR_DOWN;
                end

              rail_1 = 1'b1;
              rail_2 = 1'b1;
              rail_3 = 1'b1;
            end

      // Bring down rails
      PWR_DOWN: begin
                nxt_state = IDLE
                rail_1 = 1'b0;
                rail_2 = 1'b0;
                rail_3 = 1'b0;
              end

      default : begin
                nxt_state = IDLE;
                rail_1 = 1'b0;
                rail_2 = 1'b0;
                rail_3 = 1'b0;
              end
    endcase
  end
endmodule
```

Below is the more modern style of writing an FSM with a single process with combinational and sequential logic together. Note this is also a Mealy state machine.

```
/*
 * This module is a mealy FSM which style where there is a
 * single clocked process containing both sequential and
 * combinational. Note this style uses inferred registers
 * which is not allowed in the course. This is for
 * educational purposes only.
 */
module power_on_seq_mealy(
  input clk,
  input rst,
  //
  input pwr_on,
  input pwr_off,
  //
  output rail_1,
  output rail_2,
  output rail_3);

  // Constants
  // State Variables
  localparam IDLE=0;
  localparam WARM_UP=1;
  localparam VDD_ON=3;


  // Signals
  reg [1:0] state;


  always @(posedge clk) begin
    if (rst == 1'b1) begin
      state  = IDLE
      rail_1 = 1'b0;
      rail_2 = 1'b0;
      rail_3 = 1'b0;
    end else begin

      case (state)
        // Idle state
        IDLE: begin
                if (pwr_on == 1'b1) begin
                  state = WARM_UP;
                end
              end

        // One cycle wait state
        WARM_UP:  begin
                    state = VDD_ON;
                    rail_1 = 1'b1;
                    rail_2 = 1'b1;
                    rail_3 = 1'b1;
```

```
                  end

          // Bring up Rails
          VDD_ON: begin
                  if (pwr_off == 1'b1) begin
                    state = IDLE;
                  end

                  // Bring down rails
                  rail_1 = 1'b0;
                  rail_2 = 1'b0;
                  rail_3 = 1'b0;
                end


          default : begin
                    state = IDLE;
                    rail_1 = 1'b0;
                    rail_2 = 1'b0;
                    rail_3 = 1'b0;
                  end
        endcase
      end
    end
  endmodule
```