University of California at Berkeley
Department of Electrical Engineering and Computer Science

EECS150/EE141/EE241A, Fall 2014

Homework 3 Solutions

Due: Friday Sep 26, 5pm

Total for CS150/EE141: 90

Total for EE241A: 100

Note: For all questions, valid solutions may implement either Mealy or Moore machines unless the question specifies one or the other.

1. *[10 pts]* This solution has an external reset, as well as an internal "reset" to get the circuit ready to receive the next one-hot sequence. A valid answer could also assume, based on the wording of the question, that an external reset would be asserted between arriving one-hot sequences.

```
module onehot_to_bin(
  input clk,
  input rst,
  input serial_in,
  input in_valid,
  output reg bin_out,
  output reg out_valid);

  // synchronous registers
  reg [3:0] one_hot_reg;
  reg [2:0] count;

  // asynchronous regs
  reg [3:0] next_one_hot_reg;
  reg [2:0] next_count;

  always @(posedge clk) begin
    if(rst)
      one_hot_reg <= 4'b0;
      count <= 4'b0;
    else begin
      if(in_valid) begin
        count <= next_count;
        one_hot_reg <= next_one_hot_reg;
      end
```

```
    end
  end

  always @(*) begin
    next_one_hot_reg = {one_hot_reg[2:0], serial_in};
    next_count = count + 1;
    if(count == 4) begin
      next_one_hot_reg = 4'b0;
      next_count = 3'b0;
      case(one_hot_reg)
        4'b0001: begin
          out_valid = 1'b1;
          bin_out = 2'b00;
        end

        4'b0010: begin
          out_valid = 1'b1;
          bin_out = 2'b01;
        end

        4'b0100: begin
          out_valid = 1'b1;
          bin_out = 2'b10;
        end

        4'b1000: begin
          out_valid = 1'b1;
          bin_out = 2'b11;
        end

        default: begin
          out_valid = 1'b0;
          bin_out = 2'b00;
        end
      endcase
    end
  end
endmodule
```

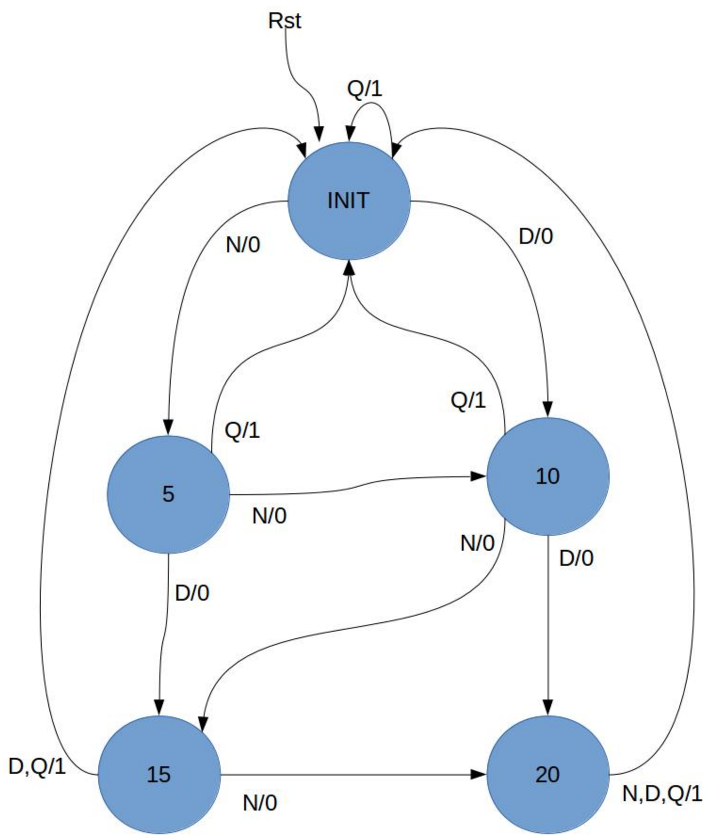
2. *[4 pts]* The case statement will infer a flat mux structure, while nested if-else will infer cascading logic.

3. *[16 pts total]*

   - Problem explicitly states that the machine does not return change. -4 pts overall for a working solution that returns change.

- Problem explicitly states that the solution cannot use a counter. -12 pts overall for any solution that uses a counter to simply keep track of the amount deposited.

(a) *[8 pts]*

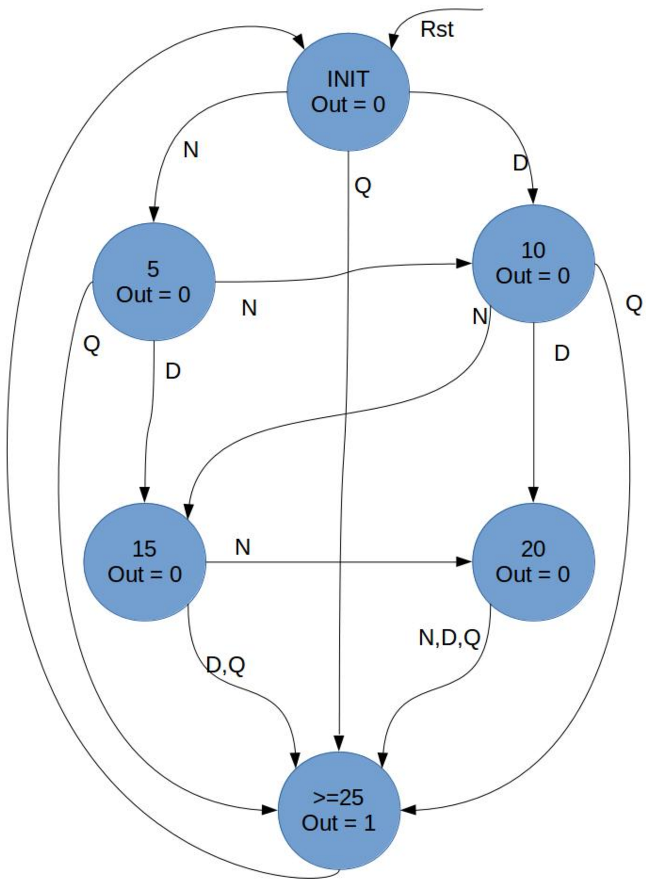



```
module vending_mealy(
  input clk,
  input rst,
  input quarter,
  input dime,
  input nickel,
  output reg dispense);

  localparam INIT = 3'd0;
  localparam FIVE = 3'd1;
  localparam TEN = 3'd2;
  localparam FIFTEEN = 3'd3;
```

```
localparam TWENTY = 3'd4;

reg [2:0] current_state;

always @(posedge clk) begin
  if(rst) begin
    current_state <= INIT;
    dispense <= 0;
  end else begin
    dispense <= 0;
    case(current_state)
      INIT: begin
        if(nickel)
          current_state <= FIVE;
        else if(dime)
          current_state <= TEN;
        else if(quarter) begin
          current_state <= INIT;
          dispense <= 1;
        end
      end

      FIVE: begin
        if(nickel)
          current_state <= TEN;
        else if(dime)
          current_state <= FIFTEEN;
        else if(quarter) begin
          current_state <= INIT;
          dispense <= 1;
        end
      end

      TEN: begin
        if(nickel)
          current_state <= FIFTEEN;
        if(dime)
          current_state <= TWENTY;
        else if(quarter)
          current_state <= INIT;
          dispense <= 1;
      end

      FIFTEEN: begin
        if(nickel)
```

```
            current_state <= TWENTY;
          else if(dime | quarter) begin
            current_state <= INIT;
            dispense <= 1;
          end
        end

        TWENTY: begin
          if(nickel | dime | quarter) begin
            current_state <= INIT;
            dispense <= 1;
          end
        end

        default: current_state <= INIT;
      endcase
    end
  end
endmodule
```

(b) *[8 pts]*




```
module vending_moore(
  input clk,
  input rst,
  input quarter,
  input dime,
  input nickel,
  output dispense);

  localparam INIT = 3'd0;
  localparam FIVE = 3'd1;
  localparam TEN = 3'd2;
  localparam FIFTEEN = 3'd3;
  localparam TWENTY = 3'd4;
  localparam TWENTYFIVE = 3'd5;

  reg [2:0] current_state;
  reg [2:0] next_state;
```

```
always @(posedge clk) begin
  if(rst)
    current_state <= INIT;
  else
    current_state <= next_state;
end

always @(*) begin
  case(current_state)
    INIT: begin
      if(nickel)        next_state = FIVE;
      else if(dime)     next_state = TEN;
      else if(quarter)  next_state = TWENTYFIVE;
      else              next_state = current_state;
    end

    FIVE: begin
      if(nickel)        next_state = TEN;
      else if(dime)     next_state = FIFTEEN;
      else if(quarter)  next_state = TWENTYFIVE;
      else              next_state = current_state;
    end

    TEN: begin
      if(nickel)        next_state = FIFTEEN;
      else if(dime)     next_state = TWENTY;
      else if(quarter)  next_state = TWENTYFIVE;
      else              next_state = current_state;
    end

    FIFTEEN: begin
      if(nickel)                next_state = TWENTY;
      else if(dime | quarter)   next_state = TWENTYFIVE;
      else                      next_state = current_state;
    end

    TWENTY: begin
      if(nickel | dime | quarter) next_state = TWENTY_FIVE;
      else next_state = current_state;
    end

    TWENTYFIVE:
      next_state = INIT;
```

```
        default:
          next_state = INIT;
      endcase
    end

    assign dispense = current_state == TWENTYFIVE;

  endmodule
```

4. *[4 pts]*

   Four major fixes (1 pt each):

   - Need to use always@(posedge clk) instead of always@(*)
   - Need `rst` in the sensitivity list b/c it needs to be asynchronous
   - Need to make use of the `en` input
   - Need to use non-blocking instead of blocking assignments

```
module dff{
  input clk,
  input rst,
  input en,
  input d,
  output q};

  reg q_reg;
  assign q = q_reg;
  always @(posedge clk or posedge rst) begin
    if(rst)
      q_reg <= 1'b0;
    else begin
      q_reg <= en ? d : q_reg;
    end
  end
endmodule
```

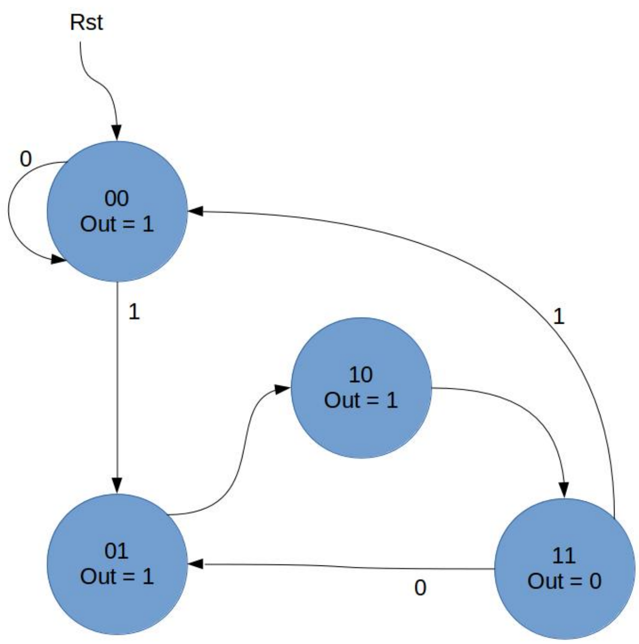
5. *[8 pts total]*

   (a) *[4 pts]*

```
module fsm(
  input clk,
  input in,
  output out);

  reg [1:0] PS;
```

```
    reg [1:0] NS;

    always @(posedge clk) begin
      PS <= NS;
    end

    assign NS[0] = (~in & PS[1]) | (in & ~PS[0]);
    assign NS[1] = PS[0] ^ PS[1];
    assign out = ~PS[0] | ~PS[1];

  endmodule
```

(b) *[4 pts]*



6. *[4 pts]* Half credit if the solution works but doesn't use a generate statement, since the question explicitly asks you to use a generate statement.

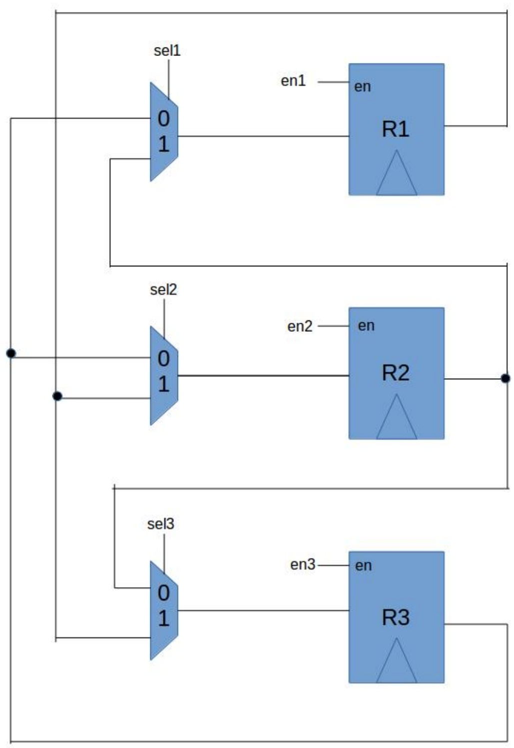
```
module edge_detect #(
  parameter delay = 1;
)
(
  input clk;
  input rst;
  input in;
```

```
  output out;
);

  reg prev_in;
  reg rise;
  reg [delay-1:0] delay_chain;

  genvar i;

  generate
    for(i = 1; i < delay; i = i + 1)
    begin: gen_delay
      always @(posedge clk) begin
        delay_chain[i] <= delay_chain[i-1];
      end
    end
  endgenerate

  always @(posedge clk) begin
    if(rst) begin
      prev_in <= 1'b0;
      rise <= 1'b0;
    end else begin
      delay_chain[0] <= rise;
      prev_in <= in;
      if(~prev_in && in)
        rise <= 1'b1;
      else
        rise <= 1'b0;
    end
  end
endmodule
```

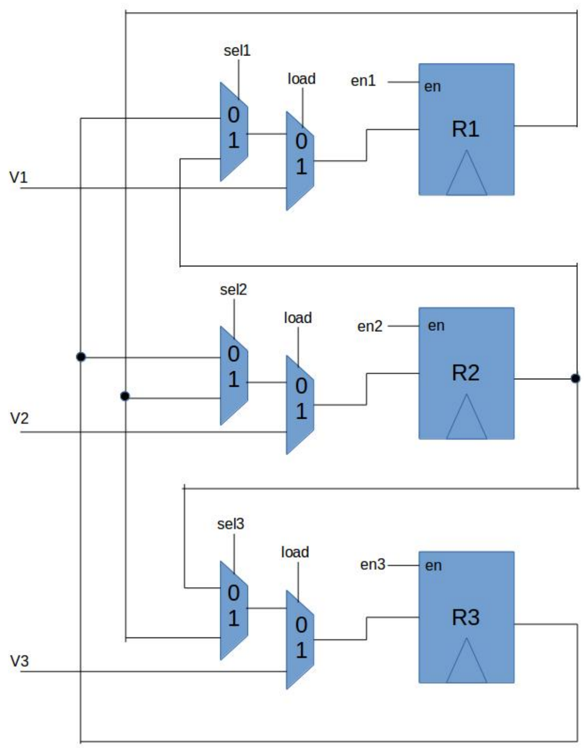
7. *[24 pts total]*

   For each part of this problem, -2 pts for excessively complex solutions.

   (a) *[4 pts]*



(b) *[4 pts]*



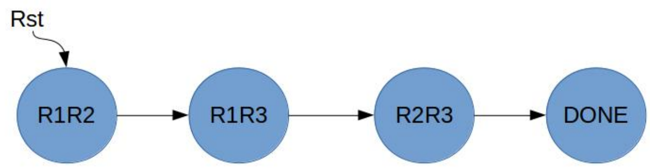
(c) *[4 pts]*

```
module swapper_dpath(
  input clk,
  input v1,
  input v2,
  input v3,
  input sel1,
  input sel2,
  input sel3,
  input en1,
  input en2,
  input en3,
  input load,
  output reg r1,
  output reg r2,
  output reg r3);
```

```
  always @(posedge clk) begin
    if(en1) r1 <= next_r1;
    if(en2) r2 <= next_r2;
    if(en3) r3 <= next_r3;
  end

  always @(*) begin
    next_r1 = sel1 ? r2 : r3;
    next_r1 = load ? v1 : next_r1;
    next_r2 = sel2 ? r1 : r3;
    next_r2 = load ? v2 : next_r2;
    next_r3 = sel3 ? r1 : r2;
    next_r3 = load ? v3 : next_r3;
  end

endmodule
```

(d) *[4 pts]* To swap R1 and R2, you need to make sure "load" is deasserted. Then assert "sel1" and "sel2" and "en1" and "en2" all at the same time. On the next clock edge, the values will be swapped.

(e) *[4 pts]* In first state, use a comparator to determine if R1 is less than R2. If R1 is less, then swap R1 and R2 and trasition to 2nd state. In this state, compare R1 to R3, and swap if R1 is less and transition to 3rd state. Finally, compare R2 and R3, and swap if R2 is less. Then transition to the "DONE" state and stay there until reset.



(f) *[4 pts]*

```
module swapper_control(
  input clk,
  input rst,
  input r1,
  input r2,
  input r3,
  output reg sel1,
  output reg sel2,
  output reg sel3,
  output reg en1,
```

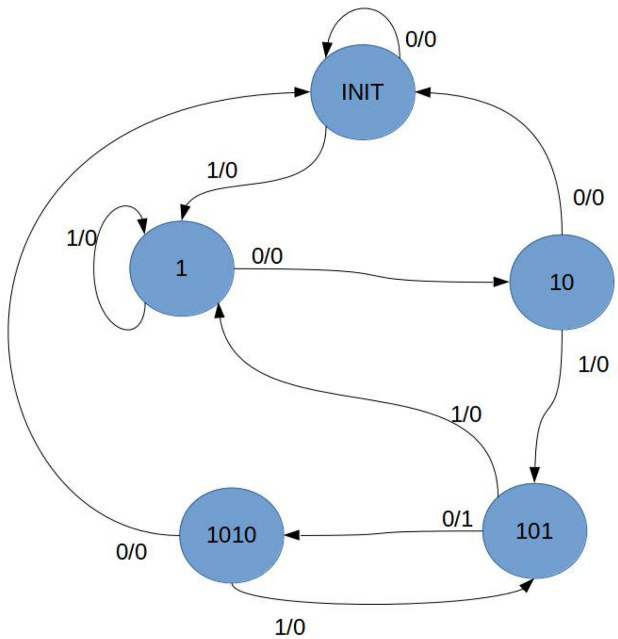
```
output reg en2,
output reg en3,
output done);

localparam R1R2 = 2'd0;
localparam R1R3 = 2'd1;
localparam R2R3 = 2'd2;
localparam DONE = 2'd3;

always @(posedge clk) begin
  if(rst)
    current_state <= R1R2;
  else
    current_state <= next_state;
end

always @(*) begin
  sel1 = 0;
  sel2 = 0;
  sel3 = 0;
  en1 = 0;
  en2 = 0;
  en3 = 0;
  case(current_state)
    R1R2: begin
      next_state = R1R3;
      if(r1 < r2) begin
        sel1 = 1;
        sel2 = 1;
        en1 = 1;
        en2 = 1;
      end
    end

    R1R3: begin
      next_state = R2R3;
      if(r1 < r3) begin
        sel1 = 0;
        sel3 = 1;
        en1 = 1;
        en3 = 1;
      end
    end

    R2R3: begin
```

```
        next_state = DONE;
        if(r2 < r3) begin
          sel2 = 0;
          sel3 = 0;
          en2 = 1;
          en3 = 1;
        end
      end

      DONE:
        next_state = current_state;

    endcase
  end

  assign done = current_state == DONE;

endmodule
```

8. *[20 pts total]*

   (a) *[7 pts]*
   Any valid encoding receives full credit. Here we use one-hot encoding to simplify logic:

```
STATE | ENCODING
----------------
INIT  | 00001
1     | 00010
10    | 00100
101   | 01000
1010  | 10000
```

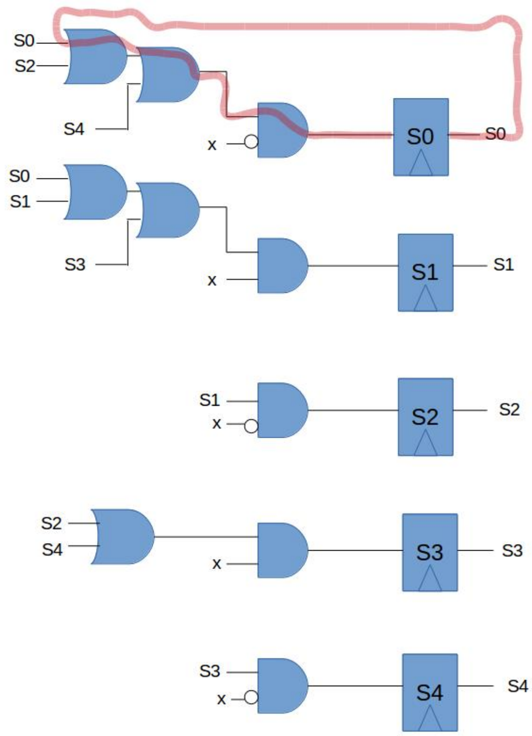


(b) *[4 pts]*

```
 x |  curr_state |  next_state
--------------------------------
 0 |    00001    |    00001
 0 |    00010    |    00100
 0 |    00100    |    00001
 0 |    01000    |    10000
 0 |    10000    |    00001
 1 |    00001    |    00010
 1 |    00010    |    00010
 1 |    00100    |    01000
 1 |    01000    |    00010
 1 |    10000    |    01000
```

(c) *[6 pts]*

For simplicity, not all wires are drawn. Register outputs on the right (S0, S1,..., S4) correspond to the nodes on the left with the same label. The only input is `x`.



(d) *[3 pts]*

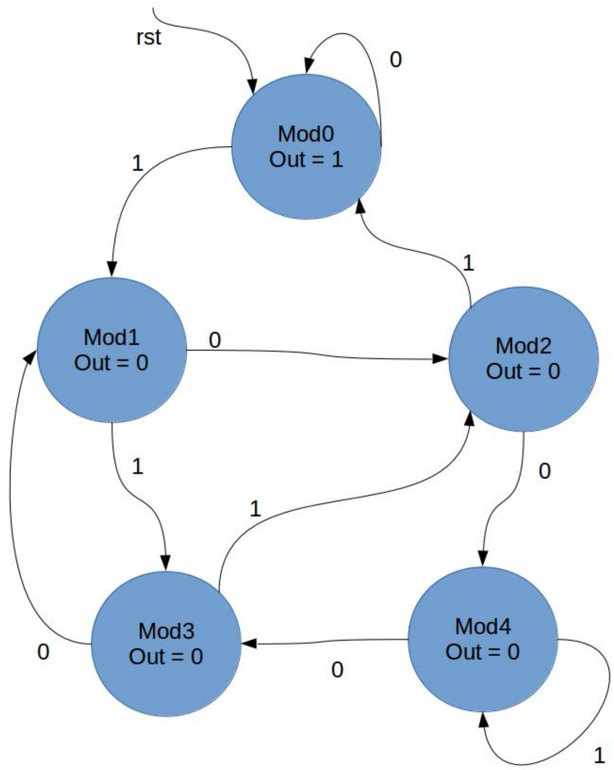
$$T_{clk} > T_{clk-to-q} + T_{CL} + T_{setup}$$

$$T_clk > 1ns + 3ns + 0.8ns$$

$$T_clk > 4.8ns$$

9. *[10 pts]*

   Any solution that recognizes that you only need to keep track of the number mod 5 should receive at least 6 pts for the problem. The other 4 points is for a correct implementation.


rst
0
Mod0
Out = 1
1
1
Mod1
Out = 0
0
Mod2
Out = 0
1
0
1
Mod3
Out = 0
Mod4
Out = 0
0
0
1