EE 231

Homework 10

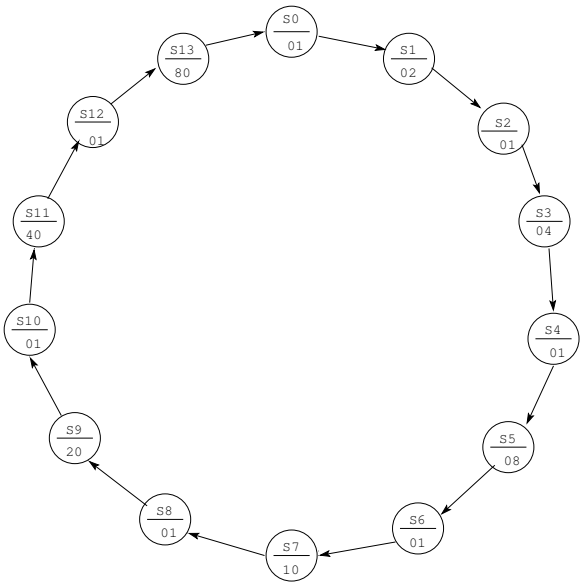
Due November 5, 2010

1. Design a synchronous sequential circuit which generates the following sequence. (The sequence should repeat itself.)

```
00000001
00000010
00000001
00000100
00000001
00001000
00000001
00010000
00000001
00100000
00000001
01000000
00000001
10000000
00000001
00000010
```

(a) Draw a state transition diagram for the circuit.

This is a system with 14 states, which repeats itself:



(b) Write a Verilog program to implement the circuit.

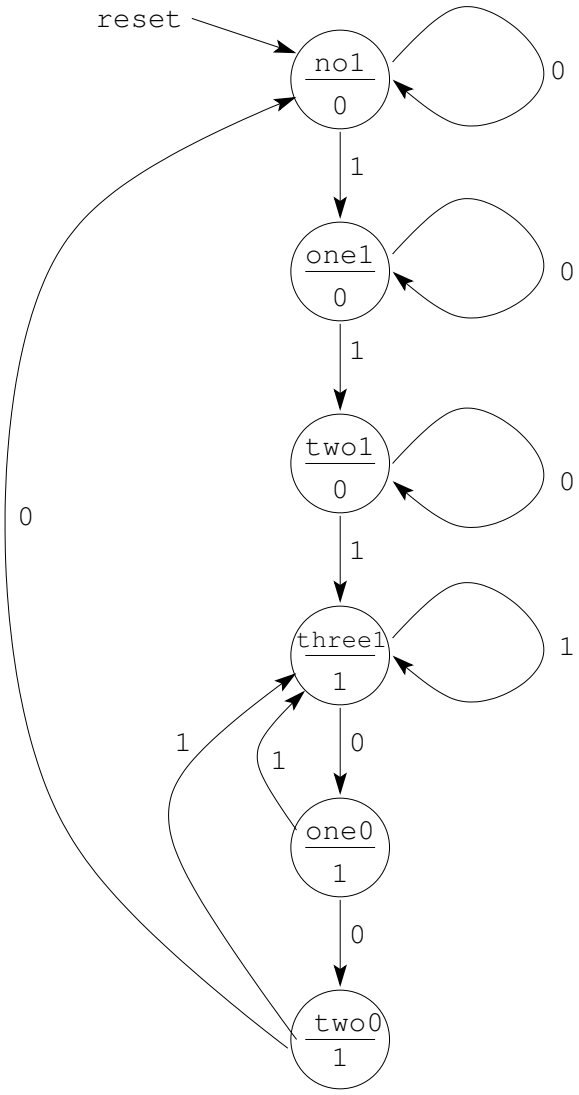
```
module hw10_p1(input clk, reset, output reg [7:0] count);

reg [3:0] state;

always @(posedge clk negedge reset)
    if (~reset) state <= 4'h0;
    else case (state)
        4'd0:    state <= 4'd1;
        4'd1:    state <= 4'd2;
        4'd2:    state <= 4'd3;
        4'd3:    state <= 4'd4;
        4'd4:    state <= 4'd5;
        4'd5:    state <= 4'd6;
        4'd6:    state <= 4'd7;
        4'd7:    state <= 4'd8;
        4'd8:    state <= 4'd9;
        4'd9:    state <= 4'd10;
        4'd10:   state <= 4'd11;
        4'd11:   state <= 4'd12;
        4'd12:   state <= 4'd13;
        4'd13:   state <= 4'd0;
        default: state <= 4'd0;
    endcase

always @(state)
    case (state)
        4'd0, 4'd2, 4'd4, 4'd6, 4'd8, 4'd10, 4'd12: count = 8'h00;
        4'd1:  count = 8'h02;
        4'd3:  count = 8'h04;
        4'd5:  count = 8'h08;
        4'd7:  count = 8'h10;
        4'd9:  count = 8'h20;
        4'd11: count = 8'h40;
        4'd13: count = 8'h80;
    endcase

endmodule
```

2. Design a synchronous sequential circuit which detects the occurrence of at least three 1's arriving at the input. The 1's do not need to arrive in consecutive clock periods. The output will go high after it sees three 1's at the input. The output will stay high until the system sees three **consecutive** 0's at the input. When it sees three consecutive 0's, the circuit should return to the reset state and start looking for three 1's.

   Here is what the output should look like for typical input:

Input	0	0	1	1	0	0	0	1	0	1	0	0	1	1	1	0	1	0	0	0	0	1	0	1	0	1	1	0
Output	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	

   (a) Draw a state transition diagram for the circuit.



(b) Write down a state transition table for the circuit.

Current State	Input	Next State	Output
no1	0	no1	0
no1	1	one1	0
one1	0	one1	0
one1	1	two1	0
two1	0	two1	0
two1	1	three1	0
three1	0	one0	1
three1	1	three1	1
one0	0	two0	1
one0	1	three1	1
two0	0	no1	1
two0	1	three1	1

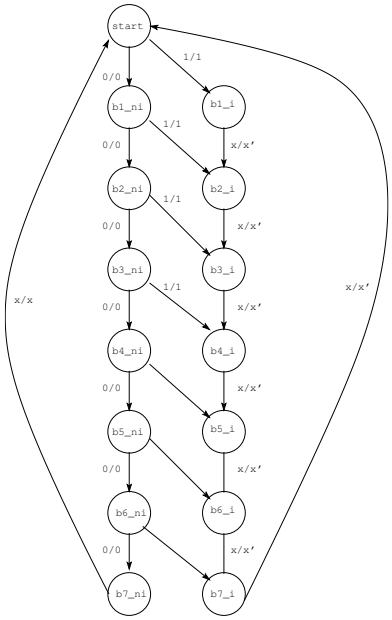
(c) Write a Verilog program to implement the circuit.

```
module hw10_p2(input clk, reset, x, output reg z);

parameter no1    = 3'h0,
          one1   = 3'h1,
          two1   = 3'h2,
          three1 = 3'h3,
          one0   = 3'h4,
          two0   = 3'h5;

reg [2:0] state, next_state;

always @(posedge clk, negedge reset)
    if (reset == 1'b0) state <= no1;
    else state <= next_state;

///* Define next state combinational block */
always @(state, x)
    case (state)
        no1 :    if (x == 1'b0) next_state = no1;
                 else next_state = one1;
        one1:    if (x == 1'b0) next_state = one1;
                 else next_state = two1;
        two1:    if (x == 1'b0) next_state = two1;
                 else next_state = three1;
        three1:  if (x == 1'b0) next_state = one0;
                 else next_state = three1;
        one0:    if (x == 1'b0) next_state = two0;
                 else next_state = three1;
        two0:    if (x == 1'b0) next_state = no1;
                 else next_state = three1;
        default: state = no1;
    endcase
//
/* Define output combinational block */
always @(state)
    case (state)
        no1,one1,two1:    z = 1'b0;
        three1,one0,two0: z = 1'b1;
        default:          z = 1'b0;
    endcase

endmodule
```

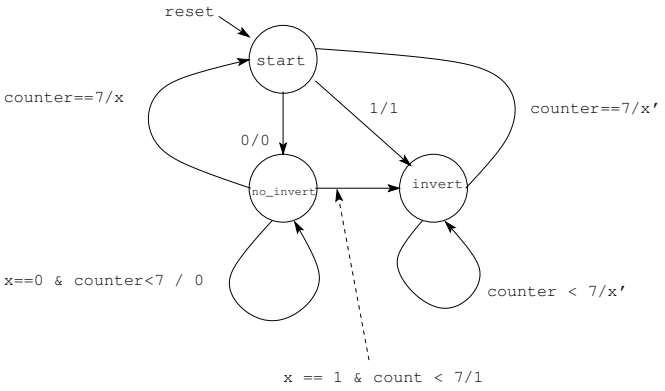
3. Design a serial 8-bit two's complementer. Eight bits are fed serially into the circuit, least significant bit first. The serial output should be the 8-bit two's complement of the input. The circuit will transition back to the reset state after each eight-bit packet is processed. Page 11 of the text discusses how you can find the 2's complement of a number by looking at the bits, starting with the least significant bit.

   (a) Draw a state transition diagram for the circuit.

   From the description in the text, you can take a 2's complement by leaving all the least significant 0's and the first 1 alone, then inverting all subsequent bits. The state machine should have the output equal to the input until the first 1. After that, the output should be in the inverse of the input. You need to keep track of the number of events, and whether or not you've gotten the first 1:



   You could also implement this with just three states and a 3-bit counter. Have the counter keep track of the number of bits. When the number of bits is seven, go back to the START state:



(b) Write a Verilog program to implement the circuit.

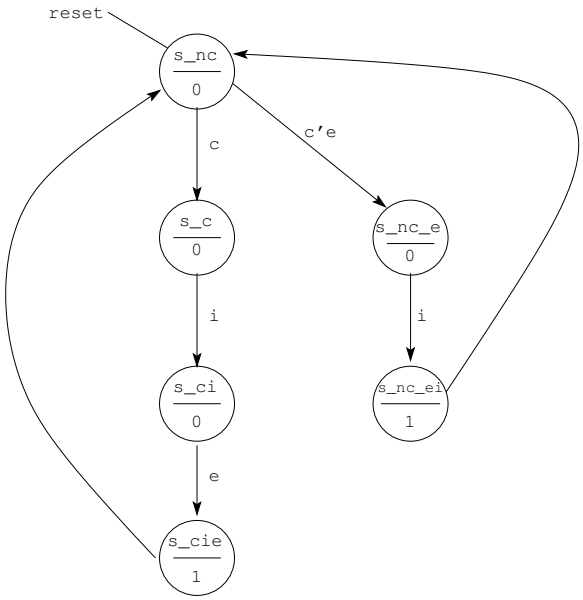
Here is a verilog program to implement the second state diagram:

```
module hw10_p3(input clk, reset, x, output reg z);

parameter START     = 2'b0,
          NO_INVERT = 2'b1,
          INVERT    = 2'h1;
reg [1:0] state;
reg [2:0] count;

always @(posedge clock, negedge reset)
    if (~reset) begin
                    state <= START;
                    count <= 3'b0;
                end
    else case (state)
        START:      begin
                        count <= 3'h1;
                        if (x == 1'b0) state <= NO_INVERT;
                        else state <= INVERT;
                    end
        NO_INVERT:  begin
                        count <= count + 1;
                        if (count == 7) state <= START;
                        else if (x == 0) state <= NO_INVERT;
                        else state <= INVERT;
                    end
        INVERT:     begin
                        count <= count + 1;
                        if (count == 7) state <= START;
                        else state <= INVERT;
                    end
        default:    begin
                        state <= START;
                        count <= 3'h0;
                    end
    endcase

always @(*)
    case (state)
        START:        z = x;
        NO_INVERT:  z = x;
        INVERT:        z = x';
        default:    z = 1'bx;
    endcase

endmodule
```

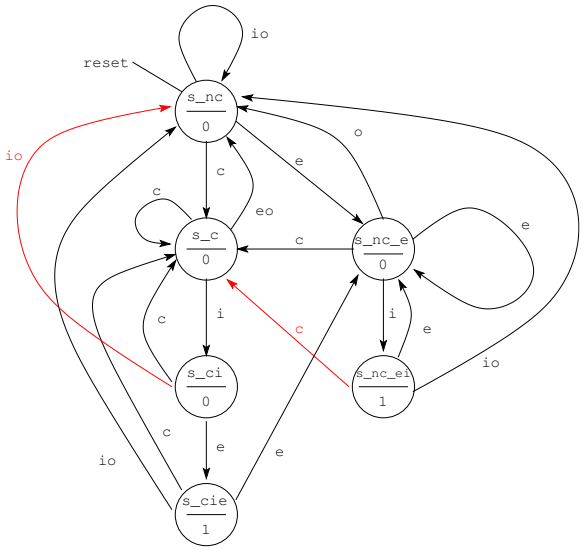
4. Design a synchronous sequential circuit with two inputs x_1 and x_0 and a single output z. The circuit detects any violation of the rule i *before* e *except after* c. The letter i is represented by $x = 01$, the letter e is represented by $x = 10$, the letter c is represented by $x = 11$, and all other letters are represented by $x = 00$. The output z will go high for one clock cycle if the circuit sees either an e followed by an i which was not preceded by a c, or if the circuit sees $c\ i\ e$.

   (a) Draw a state transition diagram for the circuit.

   The rule is violated if we see $c\ i\ e$ or $c'\ e\ i$. We have the states `start`, `s_c` (last character a c), `s_ce` (last two characters $c\ e$), etc. In the state diagram, `c` means the input character is a c, `e` means the input character is an e, `i` means the input character is an i, and `o` means the input character is anything other than a c, e or i. Here is a start to the state diagram:



The above state diagram shows only one path to get to the state `s_c` (the state where the last letter was c). However, if you are in any of the other states (e.g., state `s_nc_e`) and a c comes along, you need to go the state `s_c`. Similarly, if you are in any state except `s_c` and an e comes along, you need to go to the state $s_n c_e$. This gives a lot more paths to get to the states `s_c` and `s_nc_e`:



(b) Write a Verilog program to implement the circuit.

```
module hw10_p4(input clk, reset, input [1:0] x, output reg err);

parameter lo = 2'b00,       // Letter other than c, e, i
          li = 2'b01,       // Letter i
          le = 2'b10,       // Letter e
          lc = 2'b11;       // Letter c

parameter s_nc    = 3'h0,   // Last character was not one in sequence
          s_c     = 3'h1,   // Last character was a c
          s_ci    = 3'h2,   // Last two chars were c i
          s_cie   = 3'h3,   // Last three chars were c i e
          s_nc_e  = 3'h4,   // Last two chars were o e
          s_nc_ei = 3'h5;   // Last three chars were o e i

reg [2:0] state;

always @(posedge clock, negedge reset)
    if (~reset) state <= s_nc;
    else case (state)
        start:    if x == lc state <= s_c;
                  else if x == le state <= s_nc_e;
                  else state <= start;
        s_c:      if x == lc state <= s_c;
                  else if x == li state <= s_ci;
                  else state <= start;
        s_ci:     if x == lc state <= s_c;
                  else if x == le state <= s_cie;
                  else state <= start;
        s_nc_e:   if x == lc state <= s_c;
                  else if x == li state <= s_nc_ei;
                  else if x == le state <= s_nc_e;
                  else state <= start;
        s_nc_ei:  if x == lc state <= s_c;
                  else if x == le state <= s_nc_e;
                  else state <= start;
        default:  state <= start;
    endcase

always @(state)
    case (state)
        start, s_c, s_ci, s_nc_e: z = 1'b0;
        s_cie, s_nc_ei:           z = 1'b1;
        default:                  z = 1'b0;
    endcase
endmodule
```