PSet 8

Part A

```
module fec_pipelined(clk_in, start_in, data_in, done_out, fec_out, fsm_state_out, cycle_count_out);
    input           clk_in;
    input           start_in;
    input           data_in;
    output          done_out;
    output   [95:0] fec_out;
    output  [179:0] fsm_state_out;
    output    [5:0] cycle_count_out;

    localparam CRC_WIDTH = 16;

    wire     [CRC_WIDTH-1:0] crc;

    // output of the crc combined with data which will go to fec
    wire     [47:0] fec_input_intermediate;
    // shift register for data
    reg      [31:0] data_shift_reg = 32'b0;
    // counter which detemines no of clock cycles
    reg       [5:0] cycle_count_out = 0;
    // fun little easter egg
    wire    [179:0] msg_for_the_ta;

    assign msg_for_the_ta = 180'h07303210811111810103212111111703207110510904 6;
    // as I'm doing the fec encoding in one clock cycle, is there a state?
    assign fsm_state_out  = msg_for_the_ta;

    crc crc1(
                .clk_in(clk_in),
                .start_in(start_in),
                .data_in(data_in),
                .done_out(done_out),
                .r_out(crc)
            );

    // concatenate the bits
```

```
    assign fec_input_intermediate = {data_shift_reg, crc};

    fec f1  (
                .data_in(fec_input_intermediate),
                .fec_out(fec_out)
            );

    always @(posedge clk_in or posedge start_in)
    begin
        if (start_in)
        begin
            cycle_count_out <= 6'b0;
            data_shift_reg  <= 32'b0;
        end
        else if (~done_out)
        begin
            cycle_count_out <= cycle_count_out + 1;
            data_shift_reg  <= {data_shift_reg[30:0], data_in};
        end
    end

endmodule

`include "crc.v"
`include "fec.v"
```

crc.v

```
module crc(clk_in, start_in, data_in, done_out, r_out);
    input clk_in, start_in, data_in;
    output done_out;
    output [15:0] r_out;

    reg start_latch          = 1'b0;

    reg  [15:0] r_out        = 16'hFFFF;
    // determines number of clock cycles before done signal is asserted
    reg  [6:0] counter       = 7'd31;
    // the next value of the counter
    wire [6:0] counter_next;
    // done signal is asserted when we get to 64
    // this prevents any potential glitches
    assign done_out = counter_next[6];
    // increment the counter
    assign counter_next = counter + 1;
```

```
    always @ (posedge clk_in)
    begin
    if (start_in)
        begin
            // reset the vals
            start_latch  <= 1'b1;
            r_out        <= 16'hFFFF;
            counter      <= 7'd31;
        end
        else
        begin
            if (start_latch & ~(counter_next[6]))
            begin
                // create the gen polynomial
                r_out <= { r_out[15] ^ r_out[14] ^ data_in,
                           r_out[13:2],
                           r_out[15] ^ data_in ^ r_out[1],
                           r_out[0],
                           r_out[15] ^ data_in
                     };
                counter <= counter_next;
            end
            else
            begin
                start_latch <= 1'b0;
                // If you don't want done_out to stay high, uncomment below
                // counter <= 7'd32;
            end
        end
    end

endmodule
```

fec.v:

```
module fec  (
                data_in,
                fec_out
            );
    // calculate the fec combinationally
    localparam DATA_WIDTH = 48;

    input      [DATA_WIDTH - 1:0]   data_in;
    output     [(2*DATA_WIDTH-1):0] fec_out;
```

```
    wire       [DATA_WIDTH - 1:0]   data_in_rev;
    wire       [(2*DATA_WIDTH-1):0] fec_out_rev;

    // assign the first 6 values of the fec because x[n-3] is not defined
    // do it so p[1] and p[0] are reversed as we will reverse later on
    assign fec_out_rev[1] = data_in_rev[0];
    assign fec_out_rev[0] = data_in_rev[0];

    assign fec_out_rev[3] = data_in_rev[1] ^ data_in_rev[0];
    assign fec_out_rev[2] = data_in_rev[1];

    assign fec_out_rev[5] = data_in_rev[2] ^ data_in_rev[1] ^ data_in_rev[0];
    assign fec_out_rev[4] = data_in_rev[2] ^ data_in_rev[0];

    // generate the rest of the fec
    generate
        genvar i;
        for (i = 3; i < DATA_WIDTH; i = i + 1)
        begin:fec_rev_gen
            assign fec_out_rev[2*i + 1]     = data_in_rev[i] ^ data_in_rev[i-1] ^ data_in_rev[i-2] ^ data_in_rev[i-3];
            assign fec_out_rev[2*i]         = data_in_rev[i] ^ data_in_rev[i-2] ^ data_in_rev[i-3];
        end
    endgenerate

    // reverse the fec so it is the correct endianness
    generate
        genvar j;
        for (j = 0; j < 2*DATA_WIDTH; j = j + 1)
        begin:fec_gen
            assign fec_out[j] = fec_out_rev[(2*DATA_WIDTH-1)-j];
        end
    endgenerate

    // revese the data input endianness
    generate
        genvar k;
        for (k = 0; k < DATA_WIDTH; k = k + 1)
        begin:input_rev
            assign data_in_rev[k] = data_in[(DATA_WIDTH-1)-k];
        end
    endgenerate

endmodule
```

Testbench:

```
module fec_pipelined_tb;
    reg clk = 1'b0;
    reg  [31:0] data = 32'h03010203;
    wire [95:0] correct_fec = 96'h000E8C037C0DF00E828C0E5E;
    wire correct;

    assign correct = (correct_fec == fec);

    reg   start;
    reg   serial;
    wire  done;
    wire  [95:0] fec;
    wire  [179:0] fsm_state;
    wire  [5:0] cycle_count;
    reg         data_clk;

    fec_pipelined fp1(
                        .clk_in(clk),
                        .start_in(start),
                        .data_in(serial),
                        .done_out(done),
                        .fec_out(fec),
                        .fsm_state_out(fsm_state),
                        .cycle_count_out(cycle_count)
                    );

   initial
   begin   // system clock
      forever #5 clk = !clk;
   end

   initial
   begin   // data_clk, ensures setup time met
      #2
      forever #5 data_clk = !data_clk;
   end

    integer i;

    initial
    begin
        $dumpfile("fec_test.vcd");
        $dumpvars(0, fec_pipelined_tb);
        clk = 0;
        data_clk = 0;
        start = 0;
```

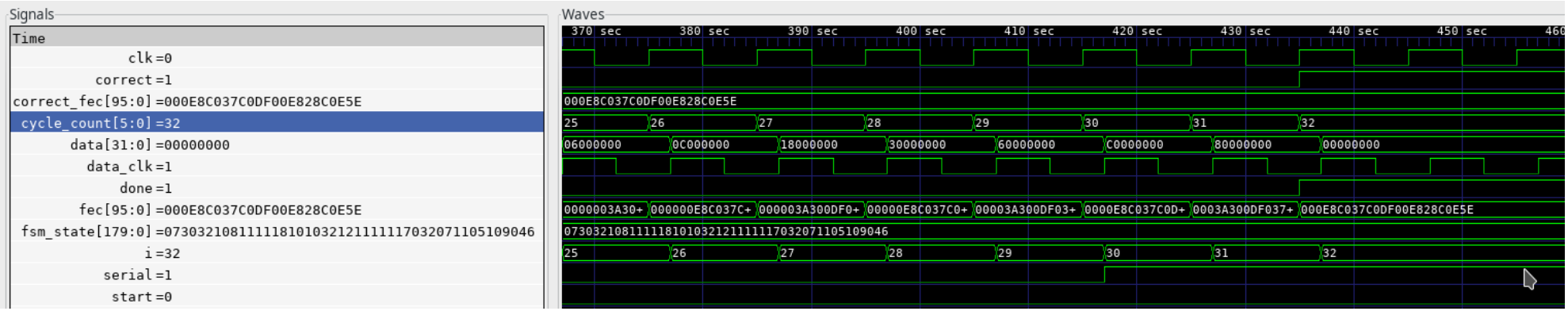
```
        serial = 0;
        // Wait 100 ns for global reset to finish
        #100;

        // start
        start=1;
        #10 start = 0;
        start=1;
        #10
        start = 0;
        #5;

        for (i=0; i<32; i=i+1)
        begin
            serial = data[31];
            @(posedge data_clk) data = {data[30:0],1'b0};
        end

        #100;
        $finish;
    end

endmodule
```



Waveform when done is asserted. The counter never reaches 34 because my solution is done by 32.

Part B

Pipelined code:

```
localparam RADIUS_SQ = RADIUS*RADIUS
```

```
always @(posedge pixel_clk)
begin
    deltax <= (hcount > (x+RADIUS)) ? (hcount - (x + RADIUS)) : ((x + RADIUS) - hcount);
    deltax <= (vcount > (y+RADIUS)) ? (vcount - (x + RADIUS)) : ((y + RADIUS) - vcount);
    if (deltax*deltax + deltay*deltay <= RADIUS_SQ)
    begin
        pixel <= COLOR;
    end
    else
    begin
        pixel <= 1'b0;
    end
end
```

The new pipelined code takes 2 clock cycles to produce an output. Thus in order to ensure the inputs from round_puck and the inputs hcount and vcount to pong_game are in sync, 2 pipeline stages are added to the input of the pong_game, rectangle and color bars module. Each set of registers that will hold previous values of hcount and vcount and be of width 11 and 10 respectively. Assuming the outputs of phsync, pvsync and pblank inside pong_game are set continuously to hsync, vsync and blank respectively, the hsync, vsync and blank inputs to the ADV7125 should also have 2 delay registers, each register storing 1 bit (for each of the 3 signals).