Lab Code [10 points]
Filename: 00_SECDED.sv
AndrewID: xinyew

```
module SECDEDdecoder
    (input logic [12:0] inCode,
    output logic [3:0] syndrome,
    output logic is1BitErr, is2BitErr,
    output logic [12:0] outCode);

    // declare modules
    makeSyndrome m1(.cw(inCode), .syndrome(syndrome));
    makeIs1BitErr m2(.syndrome(syndrome), .is1BitErr(is1BitErr),
                  .cw(inCode));
    makeIs2BitErr m4(.syndrome(syndrome), .is2BitErr(is2BitErr),
                  .cw(inCode));
    makeCorrect m3(.codeWord(inCode), .syndrome(syndrome),
                 .is1BitErr(is1BitErr), .correctCodeWord(outCode));

endmodule : SECDEDdecoder

module makeSyndrome
    (input logic [12:0] cw,
    output logic [3:0] syndrome);
    assign syndrome[0] = cw[1]^cw[3]^cw[5]^cw[7]^cw[9]^cw[11];
    assign syndrome[1] = cw[2]^cw[3]^cw[6]^cw[7]^cw[10]^cw[11];
    assign syndrome[2] = cw[4]^cw[5]^cw[6]^cw[7]^cw[12];
    assign syndrome[3] = cw[8]^cw[9]^cw[10]^cw[11]^cw[12];
endmodule : makeSyndrome

module makeCorrect
    (input logic [12:0] codeWord,
    input logic [3:0] syndrome,
    input logic is1BitErr,
    output logic [12:0] correctCodeWord);

    assign correctCodeWord = (is1BitErr) ?
        codeWord ^ (13'b1 << syndrome) : codeWord;
endmodule : makeCorrect

module makeGlobalParity(
    input logic [12:0] cw,
    output logic globalParity);
    assign globalParity = cw[0]^cw[1]^cw[2]^cw[3]^cw[4]^cw[5]^cw[6] ^
                          cw[7]^cw[8]^cw[9]^cw[10]^cw[11]^cw[12];

endmodule : makeGlobalParity

module makeIs1BitErr
    (input logic  [3:0] syndrome,
     input logic [12:0] cw,
     output logic is1BitErr);

    logic globalParity;
    makeGlobalParity m(.cw(cw), .globalParity(globalParity));

    assign is1BitErr = (globalParity) ? 1 : 0;

endmodule : makeIs1BitErr

module makeIs2BitErr
    (input logic  [3:0] syndrome,
     input logic [12:0] cw,
     output logic is2BitErr);

    logic globalParity;
    makeGlobalParity m(.cw(cw), .globalParity(globalParity));

    always_comb begin
        is2BitErr = 0;
        if(globalParity == 0) begin
            if(syndrome != 0) begin
                is2BitErr = 1;
            end
            else
                is2BitErr = 0;
        end
    end
endmodule : makeIs2BitErr

```

Lab Code [10 points]
Filename: 00_library.sv
AndrewID: xinyew

```
`default_nettype none

// A PIPO Shift Register, with controllable shift direction
// Load has priority over shifting.
module LeftShift8Register
  (input  logic [7:0] D,
   input  logic en, clock,
   output logic [511:0] Q);

  always_ff @(posedge clock)
    if (en)
      Q <= {Q[503:0], D};

endmodule : LeftShift8Register

module Comparator
   #(parameter   WIDTH = 8)
  (input  logic [WIDTH-1:0] A, B,
   output logic        AeqB);

  assign AeqB = (A == B);
  // Straightforward.  Uses compare operator.

endmodule : Comparator
// Magnitude comparator

/*
 * A library of components, usable for many future hardware designs.
 */

// A Magnitude Comparator does an unsigned comparison of two input values.
module MagComp
  #(parameter   WIDTH = 8)
  (output logic             AltB, AeqB, AgtB,
   input  logic [WIDTH-1:0] A, B);

  assign AeqB = (A == B);
  assign AltB = (A <  B);
  assign AgtB = (A >  B);

endmodule: MagComp

// The Multiplexer chooses one of WIDTH bits
module Multiplexer
  #(parameter WIDTH=8)
  (input  logic [WIDTH-1:0]         I,
   input  logic [$clog2(WIDTH)-1:0] S,
   output logic                     Y);

   assign Y = I[S];

endmodule : Multiplexer

// The 2-to-1 Multiplexer chooses one of two multi-bit inputs.
module Mux2to1
  #(parameter WIDTH = 8)
  (input  logic [WIDTH-1:0] I0, I1,
   input  logic             S,
   output logic [WIDTH-1:0] Y);

  assign Y = (S) ? I1 : I0;

endmodule : Mux2to1

// The Decoder converts from binary to one-hot codes.
module Decoder
  #(parameter WIDTH=8)
  (input  logic [$clog2(WIDTH)-1:0] I,
   input  logic                     en,
   output logic [WIDTH-1:0]         D);

  always_comb begin
    D = '0;
    if (en)
      D[I] = 1'b1;
  end

endmodule : Decoder

// A DFlipFlop stores the input bit synchronously with the clock signal.
// preset and reset are asynchronous inputs.
module DFlipFlop
  (input  logic D,
   input  logic preset_L, reset_L, clock,
   output logic Q);

  always_ff @(posedge clock, negedge preset_L, negedge reset_L)
    if (~preset_L & reset_L)
      Q <= 1'b1;
    else if (~reset_L & preset_L)
      Q <= 1'b0;
    else if (~reset_L & ~preset_L)
      Q <= 1'bX;
    else
      Q <= D;

endmodule : DFlipFlop

// A Register stores a multi-bit value.
// Enable has priority over Clear
module Register
  #(parameter WIDTH=8)
  (input  logic [WIDTH-1:0] D,
   input  logic             en, clear, clock,
   output logic [WIDTH-1:0] Q);

  always_ff @(posedge clock)
    if (en)
      Q <= D;
    else if (clear)
      Q <= '0;

endmodule : Register

// A binary up-down counter.
// Clear has priority over Load, which has priority over Enable
module Counter
  #(parameter WIDTH=8)
  (input  logic [WIDTH-1:0] D,
   input  logic             en, clear, load, clock, up,
   output logic [WIDTH-1:0] Q);

  always_ff @(posedge clock)
    if (clear)
      Q <= {WIDTH {1'b0}};
    else if (load)
      Q <= D;
    else if (en)
      if (up)
        Q <= Q + 1'b1;
      else
        Q <= Q - 1'b1;

endmodule : Counter

// A Synchronizer takes an asynchronous input and changes it to synchronized
module Synchronizer
  (input  logic async, clock,
   output logic sync);

  logic metastable;

  DFlipFlop one(.D(async),
                .Q(metastable),
                .clock,
                .preset_L(1'b1),
                .reset_L(1'b1)
               );

  DFlipFlop two(.D(metastable),
                .Q(sync),
                .clock,
                .preset_L(1'b1),
                .reset_L(1'b1)
               );

endmodule : Synchronizer

// A PIPO Shift Register, with controllable shift direction
// Load has priority over shifting.
module ShiftRegister_PIPO
  #(parameter WIDTH=8)
  (input  logic [WIDTH-1:0] D,
   input  logic             en, left, load, clock,
   output logic [WIDTH-1:0] Q);

  always_ff @(posedge clock)
    if (load)
      Q <= D;
    else if (en)
      if (left)
        Q <= {Q[WIDTH-2:0], 1'b0};
      else
        Q <= {1'b0, Q[WIDTH-1:1]};

endmodule : ShiftRegister_PIPO

// A SIPO Shift Register, with controllable shift direction
// Load has priority over shifting.
module ShiftRegister_SIPO
  #(parameter WIDTH=8)
  (input  logic             serial,
   input  logic             en, left, clock,
   output logic [WIDTH-1:0] Q);

  always_ff @(posedge clock)
    if (en)
      if (left)
        Q <= {Q[WIDTH-2:0], serial};
      else
        Q <= {serial, Q[WIDTH-1:1]};

endmodule : ShiftRegister_SIPO

// A BSR shifts bits to the left by a variable amount
module BarrelShiftRegister
  #(parameter WIDTH=8)
  (input  logic [WIDTH-1:0] D,
   input  logic             en, load, clock,
   input  logic [      1:0] by,
   output logic [WIDTH-1:0] Q);

  logic [WIDTH-1:0] shifted;
  always_comb
    case (by)
      default: shifted = Q;
      2'b01: shifted = {Q[WIDTH-2:0], 1'b0};
      2'b10: shifted = {Q[WIDTH-3:0], 2'b0};
      2'b11: shifted = {Q[WIDTH-4:0], 3'b0};
    endcase

  always_ff @(posedge clock)
    if (load)
        Q <= D;
    else if (en)
        Q <= shifted;

endmodule : BarrelShiftRegister

```

Lab Code [10 points]
Filename: 00_test.sv
AndrewID: xinyew

```
`default_nettype none

module tb();
  logic [7:0] D;
  logic [511:0] Q;
  logic en, load, clock;

  LeftShift8Register DUT (.*);

  initial begin
      clock = 1;
      forever #10 clock = ~clock;
  end

  initial begin
      $monitor($time,, "D: %x | Q: %x", D, Q);
  end

  initial begin
      D <= 8'h11;
      en <= 1;
      @(posedge clock);
      load <= 0;
      @(posedge clock);
      @(posedge clock);
      @(posedge clock);
      @(posedge clock);
      @(posedge clock);
      D <= 8'hFF;
      @(posedge clock);
      @(posedge clock);
      #1 $finish;
  end
endmodule : tb
```

Lab Code [10 points]
Filename: 01.sv
AndrewID: xinyew

```
`default_nettype none

module Receiver
    (input logic clock, reset, serialIn,
     output logic [7:0] messageByte,
     output logic isNew);

    logic is2bitErr, fs_error, done;
    // for Shift Register
    logic S_en;
    // for Out Register
    logic R_en, R_clear;
    // for Cycle Counter
    logic C_en, C_clear;
    logic [3:0] C_count;
    // for Error Counter
    logic E_en, E_clear;
    logic [3:0] E_count;
    logic [7:0] mux_out;
    logic [12:0] received_message, sm;
    logic [1:0] state, n_state;

    fsm control(.*);
    ShiftRegister_SIPO #(13) reg1 (.serial(serialIn),
                                   .en(S_en),
                                   .left(1),
                                   .clock(clock),
                                   .Q(received_message));

    SECDEDdecoder dec1 (.inCode(received_message),
                        .is2BitErr(is2bitErr),
                        .outCode(sm));

    Counter counter1 (.en(C_en), .clear(C_clear), .up(1),
                      .clock(clock), .Q(C_count));

    Counter counter2 (.en(E_en), .clear(E_clear), .up(1),
                      .clock(clock), .Q(E_count));

    Comparator comp1 (.A(C_count), .B(4'd12), .AeqB(done));

    Comparator comp2 (.A(E_count), .B(4'd11), .AeqB(fs_error));

    Mux2to1 m1 (.I0({sm[12], sm[11], sm[10], sm[9],
                     sm[7], sm[6], sm[5], sm[3]}),
                .I1(8'h15), .S(is2bitErr | fs_error), .Y(mux_out));

    Register reg2 (.en(R_en), .clear(R_clear), .clock(clock),
                .D(mux_out), .Q(messageByte));

endmodule : Receiver

module fsm
    (input logic clock, serialIn, reset, done, fs_error,
     output logic S_en, R_en, R_clear, C_en, C_clear, E_en, E_clear);

    enum logic [1:0] {
                        idle = 2'b00,
                        running = 2'b01,
                        completed = 2'b10,
                        error = 2'b11
                        } state, n_state;

    always_ff @(posedge clock) begin
        if (reset)
            state <= idle;
        else
            state <= n_state;
    end

     always_comb begin
        case(state)
            idle : begin
                if(fs_error)
                    n_state = error;
                else if(serialIn)
                    n_state = running;
                else if(~serialIn)
                    n_state = idle;

                S_en = 0;
                // loop counter signal
                C_en = 0;
                C_clear = 1;
                // error counter signal
                E_en = 1;
                E_clear = 0;
                // register signal
                R_en = 0;
                R_clear = 0;
            end

            running : begin
                n_state = (done) ? completed : running;
                S_en = 1;
                C_en = 1;
                R_en = 0;
                R_clear = 0;
                E_en = 0;
                E_clear = 1;
                C_clear = 0;
            end

            completed : begin
                n_state = (serialIn) ? error : idle;
                S_en = 0;
                C_en = 0;
                R_en = 1;
                R_clear = 0;
                E_en = 0;
                E_clear = 1;
                C_clear = 1;
            end
            error : begin
                n_state = idle;
                S_en = 0;
                C_en = 0;
                R_en = 1;
                R_clear = 0;
                E_en = 0;
                E_clear = 1;
                C_clear = 1;
            end
        endcase
     end
endmodule : fsm
```

Lab Code [10 points]
Filename: 01_sender.sv
AndrewID: xinyew

```systemverilog
/*
 * Lab 3a: Transmitter, Task1
 *
 */

module Sender(
  input  logic clock, reset,
  output logic serialOut);

  parameter WORDS = 25, WORD_SIZE = 64;
  logic [WORD_SIZE-1:0] message_rom [WORDS-1:0];
  logic [12:0] word_counter;
  logic [5:0] bit_counter;

  initial begin
    $readmemb("01.vm", message_rom);
  end

  always_ff @(posedge clock, posedge reset) begin
    if (reset) begin
      serialOut <= 0;
      word_counter <= 0;
      bit_counter <= WORD_SIZE - 1;
    end
    else begin
      if (word_counter < WORDS)
        serialOut <= message_rom[word_counter][bit_counter];
      else
        serialOut <= 0;

      if (bit_counter == 0) begin
        bit_counter = WORD_SIZE -1;
        word_counter = word_counter + 1;
      end else
        bit_counter = bit_counter - 1;
    end
  end

endmodule : Sender
```

Lab Code [10 points]
Filename: 01_tb.sv
AndrewID: xinyew

```
`default_nettype none

module testbench();
  logic clock, reset, data;

  Sender S (.clock(clock),
            .reset(reset),
            .serialOut(data));

  logic [7:0] byteOut;
  logic isNew;
  Receiver R (.clock(clock),
              .reset(reset),
              .serialIn(data),
              .messageByte(byteOut),
              .isNew(isNew));

  initial begin
    clock = 0;
    reset = 1;
    forever #10 clock = ~clock;
  end

  initial begin
    $monitor($time,, "%8s %8s Mux: %b Char: %h %s %b %b",
    R.control.state.name, R.control.n_state.name, R.reg1.Q,
    byteOut, byteOut, R.fs_error, R.is2bitErr);
    @(posedge clock);
    reset <= 0;
    @(posedge clock);
  #25000 $finish;
  end
endmodule : testbench
```

Lab Code [10 points]
Filename: 02.sv
AndrewID: xinyew

```
`default_nettype none

module task2
    (input logic clock, reset, serialIn,
     output logic [511:0] messageBytes,
     output logic isNew);

    // 2bitErr, frame errors
    logic is2bitErr, fs_error, fe_error;
    // time to sample, time to sample next bit;
    logic timeToSample, timeNextBit;
    // a block of 13bits received
    logic blockReceived;

    // for Shift Register
    logic S_en; // enale shifting

    // for Out Register
    logic R_en;

    // for Cycle Counter(13-bit output)
    logic C_en, C_clear;
    logic [31:0] C_count;

    // for Error Counter(detect fs_error)
    logic E_en, E_clear;
    logic [31:0] E_count;

    // for Sample counter(detect synced timing for sampling)
    logic A_en, A_clear;
    logic [31:0] A_count;

    // for Wait counter(detect 16 clock cycles to sample next)
    logic W_en, W_clear;
    logic [31:0] W_count;

    // for char counter(count char num received)
    logic Char_en, Char_clear;
    logic [31:0] Char_count;

    // if fs_error, fe_error or is2BitErr, output 'h15;
    logic [7:0] mux_out;

    // uncorrected message, corrected message
    logic [12:0] uncorrected, corrected;

    // FSM for the module
    fsm control(.*);

    // shift reg to collect bits
    ShiftRegister_SIPO #(13) reg1 (.serial(serialIn),
                .en(S_en), .left(1), .clock(clock), .Q(uncorrected));

    // corrector
    SECDEDdecoder dec1 (.inCode(uncorrected), .is2BitErr(is2bitErr),
                        .outCode(corrected));

    // Count whether 13-bit block is received
    Counter #(32) counter1 (.en(C_en), .clear(C_clear), .up(1),
                    .clock(clock), .Q(C_count));

    // Count whether a beginning frame error occurs
    Counter #(32) counter2 (.en(E_en), .clear(E_clear), .up(1),
                    .clock(clock), .Q(E_count));

    // Count synced timing for sampling
    Counter #(32) counter3 (.en(A_en), .clear(A_clear), .up(1),
                    .clock(clock), .Q(A_count));

    // Count wait cycles
    Counter #(32) counter4 (.en(W_en), .clear(W_clear), .up(1),
                    .clock(clock), .Q(W_count));

    // Count how many chars receited
    Counter #(32) counter5 (.en(Char_en), .clear(Char_clear), .up(1),
                    .clock(clock), .Q(Char_count));

    // Count timings to generate real edges
    // Counter counter5 (.en(

    // whether a block is received
    Comparator #(32) comp1 (.A(C_count), .B(32'd13), .AeqB(blockReceived));

    // whether we get a frame error of 10 consecutive 0s
    Comparator #(32) comp2 (.A(E_count), .B(32'd47_700), .AeqB(fs_error));

    // whether it's time to sample after syncing 8 * 3975
    Comparator #(32) comp3 (.A(A_count), .B(32'd1_988), .AeqB(timeToSample));

    // whether it's time to sample without seeing an edge
    Comparator #(32) comp4 (.A(W_count), .B(32'd3_975), .AeqB(timeNextBit));

    // choose from corrected char or error code
    Mux2to1 m1 (.I0({corrected[12], corrected[11], corrected[10],
                     corrected[9], corrected[7], corrected[6],
                     corrected[5], corrected[3]}),
                .I1(8'h15), .S(is2bitErr | fs_error | fe_error),
                .Y(mux_out));

    // reg to store the result
    LeftShift8Register reg2 (.en(R_en), .clock(clock),
                .D(mux_out), .Q(messageBytes));

endmodule : task2

module fsm
    (input logic clock, serialIn, reset, timeToSample,
                 blockReceived, fs_error, timeNextBit, is2bitErr,
     output logic S_en, R_en, fe_error,
     output logic C_en, C_clear, E_en, E_clear, A_en, A_clear,
                  W_en, W_clear, Char_en, Char_clear);

    enum logic [2:0] {IDLE = 3'b000, SYNC = 3'b001,
                      SAMPLE = 3'b010, WAIT0 = 3'b011,
                      WAIT1 = 3'b100, COMPLETED = 3'b101,
                      ERROR = 3'b110} state, n_state;

    always_ff @(posedge clock, posedge reset) begin
        if (reset) begin
            state <= IDLE;
            Char_clear <= 1;
        end
        else begin
            state <= n_state;
            Char_clear <= 0;
        end
    end

    always_comb begin
        case(state)
            IDLE : begin
                fe_error = 0;
                if(fs_error)
                    n_state = ERROR;
                else if(serialIn)
                    n_state = SYNC;
                else
                    n_state = IDLE;

                // Do not shift
```

```
                    S_en = 0;
                    // keep clearing loop counter
                    C_en = 0;
                    C_clear = 1;
                    // keep counting 0s in IDLE state
                    E_en = 1;
                    E_clear = 0;
                    // do not count sampling timing after syncing
                    A_en = 0;
                    A_clear = 1;
                    // do not wait for edges or next sampling timing without
                    // syncing
                    W_en = 0;
                    W_clear = 1;
                    // do not increase char count, and do not clear it
                    Char_en = 0;
                    // disable register
                    R_en = 0;
                end

                SYNC : begin
                    fe_error = 0;
                    if (timeToSample)
                        n_state = SAMPLE;
                    else
                        n_state = SYNC;
                    // Do not shift
                    S_en = 0;
                    // Do not increase loop count and do not clear it
                    C_en = 0;
                    C_clear = 0;
                    // keep clearning frame error counter
                    E_en = 0;
                    E_clear = 1;
                    // start counting sampling timing after syncing
                    A_en = 1;
                    A_clear = 0;
                    // do not wait for edges or next sampling timing without
                    // syncing
                    W_en = 0;
                    W_clear = 1;
                    // do not increase char count, and do not clear it
                    Char_en = 0;
                    // disable register
                    R_en = 0;
                end

                SAMPLE : begin
                    fe_error = 0;
                    if (serialIn)
                      n_state = WAIT1;
                    else
                      n_state = WAIT0;
                    // Shift once
                    S_en = 1;
                    // Increase num of bit collected by 1
                    C_en = 1;
                    C_clear = 0;
                    if (blockReceived)
                      n_state = COMPLETED;
                    // keep clearning frame error counter
                    E_en = 0;
                    E_clear = 1;
                    // clearing counting sampling timing after syncing
                    A_en = 0;
                    A_clear = 1;
                    // do not wait for next sampling timing without
                    // syncing
                    W_en = 0;
                    W_clear = 1;
                    // do not increase char count, and do not clear it
```

```
            Char_en = 0;
            // disable register
            R_en = 0;
        end

        WAIT0 : begin
            fe_error = 0;
            if (serialIn)
              n_state = SYNC;
            else if (timeNextBit)
              n_state = SAMPLE;
            else
              n_state = WAIT0;
            // Stop shifting
            S_en = 0;
            // Stop counting loop counter, do not clear
            C_en = 0;
            C_clear = 0;
            // keep clearing frame error counter
            E_en = 0;
            E_clear = 1;
            // Stop counting sampling timing and clear it
            A_en = 0;
            A_clear = 1;
            // Counting next sampling timing without syncing
            W_en = 1;
            W_clear = 0;
            // do not increase char count, and do not clear it
            Char_en = 0;
            // disable register
            R_en = 0;
        end

        WAIT1 : begin
            fe_error = 0;
            if (~serialIn)
              n_state = SYNC;
            else if (timeNextBit)
              n_state = SAMPLE;
            else
              n_state = WAIT1;
            // Stop shifting
            S_en = 0;
            // Stop counting loop counter, do not clear
            C_en = 0;
            C_clear = 0;
            // keep clearing frame error counter
            E_en = 0;
            E_clear = 1;
            // Stop counting sampling timing and clear it
            A_en = 0;
            A_clear = 1;
            // Counting next sampling timing without syncing
            W_en = 1;
            W_clear = 0;
            // do not increase char count, and do not clear it
            Char_en = 0;
            // disable register
            R_en = 0;
        end

        COMPLETED : begin
            if (~timeNextBit) begin
              n_state = COMPLETED;
              fe_error = 0;
              R_en = 0;
              Char_en = 0;
              W_en = 1;
              W_clear = 0;
            end
            else if (is2bitErr || serialIn) begin
```

```
                    n_state = ERROR;
                    fe_error = 1;
                    R_en = 1;
                    Char_en = 1;
                    W_en = 0;
                    W_clear = 1;
                  end
                  else begin
                    n_state = IDLE;
                    fe_error = 0;
                    R_en = 1;
                    Char_en = 1;
                    W_en = 1;
                    W_clear = 0;
                  end
                  // Stop shifting
                  S_en = 0;
                  // Stop counting loop counter, clear it
                  C_en = 0;
                  C_clear = 1;
                  // keep clearing frame error counter
                  E_en = 0;
                  E_clear = 1;
                  // Stop counting sampling timing and clear it
                  A_en = 0;
                  A_clear = 1;
                  // increase char count, and do not clear it
                  // enable register
              end

              ERROR : begin
                  fe_error = 0;
                  if (~timeNextBit)
                    n_state = ERROR;
                  else
                    n_state = IDLE;
                  // Stop shifting
                  S_en = 0;
                  // Stop counting loop counter, clear it
                  C_en = 0;
                  C_clear = 1;
                  // keep clearing frame error counter
                  E_en = 0;
                  E_clear = 1;
                  // Stop counting sampling timing and clear it
                  A_en = 0;
                  A_clear = 1;
                  // Stop counting next sampling timing without syncing, clear
                  W_en = 1;
                  W_clear = 0;
                  // do not increase char count, and do not clear it
                  Char_en = 0;
                  // enable register
                  R_en = 0;
              end

          endcase
      end
endmodule : fsm
```

Lab Code [10 points]
Filename: 02_chipInterface.sv
AndrewID: xinyew

```systemverilog
`default_nettype none
module chipInterface(
    input logic CLOCK_50,
    input logic [17:0] SW,
    input logic UART_RXD,
    input logic[3:0] KEY,
    output logic [6:0] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0
    );
    // here we declare the output
    logic [511:0] m; // this should be connected to out
    logic [511:0] mm; // this should be connected to out

    logic [31:0] selected;
    logic reset;
    assign reset = ~KEY[0];
    logic isNew;

    // here we declare our receiver
    task2 R (.clock(CLOCK_50),
              .reset(reset),
              .serialIn(UART_RXD),
              .messageBytes(mm),
              .isNew(isNew));

    // here we declare the counter
    select s (.bits(mm),
              .addr({SW[7:0]}),
              .out(selected));

    logic [7:0] blank;
    assign blank = 8'd0;

    SevenSegmentDisplay s1 (.BCX0(selected[31:28]), .blank(blank), .HEX0(HEX7));
    SevenSegmentDisplay s2 (.BCX0(selected[27:24]), .blank(blank), .HEX0(HEX6));
    SevenSegmentDisplay s3 (.BCX0(selected[23:20]), .blank(blank), .HEX0(HEX5));
    SevenSegmentDisplay s4 (.BCX0(selected[19:16]), .blank(blank), .HEX0(HEX4));
    SevenSegmentDisplay s5 (.BCX0(selected[15:12]), .blank(blank), .HEX0(HEX3));
    SevenSegmentDisplay s6 (.BCX0(selected[11:8]), .blank(blank), .HEX0(HEX2));
    SevenSegmentDisplay s7 (.BCX0(selected[7:4]), .blank(blank), .HEX0(HEX1));
    SevenSegmentDisplay s8 (.BCX0(selected[3:0]), .blank(blank), .HEX0(HEX0));


endmodule : chipInterface


module select
 (input logic [511:0] bits,
  input logic [7:0] addr,
  output logic [31:0] out);

  logic [511:0] tmp;
  assign tmp = bits >> (addr * 32);
  assign out = tmp[31:0];

endmodule : select


module SevenSegmentDisplay
  (input  logic [3:0] BCX0,
   input  logic [7:0] blank,
   output logic [6:0] HEX0);

  always_comb begin
    HEX0 = 7'b0000000;
    if (~blank[0])
      case (BCX0)
        4'h0: HEX0 = 7'b0111111;
        4'h1: HEX0 = 7'b0000110;
        4'h2: HEX0 = 7'b1011011;
        4'h3: HEX0 = 7'b1001111;
        4'h4: HEX0 = 7'b1100110;
        4'h5: HEX0 = 7'b1101101;
        4'h6: HEX0 = 7'b1111101;
        4'h7: HEX0 = 7'b0000111;
        4'h8: HEX0 = 7'b1111111;
        4'h9: HEX0 = 7'b1100111;
        4'ha: HEX0 = 7'b1110111;
        4'hb: HEX0 = 7'b1111100;
        4'hc: HEX0 = 7'b0111001;
        4'hd: HEX0 = 7'b1011110;
        4'he: HEX0 = 7'b1111001;
        4'hf: HEX0 = 7'b1110001;
        default: HEX0 = 7'b0000000;
      endcase
    HEX0 = ~HEX0;
  end


endmodule : SevenSegmentDisplay

```

Lab Code [10 points]
Filename: 02_clockDriver.sv
AndrewID: xinyew

```
module clock_divider
  #(parameter equiv_cycle = 3975)
  (input logic en, clock, reset,
   output logic new_clock);

  logic [$clog2(equiv_cycle)-1:0] Q;
  logic clock_edge, clear;

  // declare control fsm
  clock_fsm control(.*);

  // Counter to count up to equiv cycle
  Counter dut1(.en(en), .clear(clear), .clock(clock), .up(1), .Q(Q));

  // Comparator to check whether we have hit equiv cycle
  Comparator dut2(.A(Q), .B(equiv_cycle), .AeqB(clock_edge));

endmodule : clock_divider

module clock_fsm
  (input logic clock, clock_edge, reset,
    output logic new_clock, clear);

  // here we declare the possible states
  enum logic {out0, out1} state, n_state;

  // flip-flop for next states
  always_ff @(posedge clock) begin
    if(reset)
      state <= out0;
    else
      state <= n_state;
  end

  // next state generation
  always_comb begin
    case(state)
      out0 : begin
        n_state = (clock_edge) ? out1 : out0;
        clear = (clock_edge) ? 1 : 0;
        new_clock = 0;
      end
      out1 : begin
        n_state = (clock_edge) ? out0 : out1;
        clear = (clock_edge) ? 1 : 0;
        new_clock = 1;
      end
    endcase
  end

endmodule : clock_fsm

// this is a test bench for out clock divider

module clock_divider_test();
    // declare the variables
    logic en, reset, clear, clock, new_clock;

    // here we make the clock run at 50 Mhz
    initial begin
        clock = 0;
        forever #2ns clock = ~clock;
    end

    // here we declare the clock_divider module
    clock_divider #(5) dut(.*);

    // begin test_bench
    initial begin
```

```
        reset = 1;
        en = 1;
        clear = 1;
        #1ns;
        clear = 0;
        reset = 0;
        $monitor($time,, "new clock: %b", new_clock);
        #10ms;
        $finish;
    end

endmodule: clock_divider_test

```

Lab Code [10 points]
Filename: 02_memory.sv
AndrewID: xinyew

```
`default_nettype none

module BusDriver
 #(parameter WIDTH = 8)
  (input  logic en,
   input  logic [WIDTH-1:0] data,
   output logic [WIDTH-1:0] buff,
   inout  tri   [WIDTH-1:0] bus
   );

   assign bus = (en) ? data : 'bz;
   assign buff = bus;

endmodule : BusDriver

module Memory
 #(parameter DW = 16,
             W  = 256,
             AW = $clog2(W))
  (input  logic re, we, clock,
   input  logic [AW-1:0] addr,
   inout  tri   [DW-1:0] data);

  logic [DW-1:0] M[W];
  logic [DW-1:0] rData;

  assign data = (re) ? rData : 'bz;

  always_ff @(posedge clock)
    if (we)
      M[addr] <= data;

  always_comb
    rData = M[addr];

endmodule : Memory
```

Lab Code [10 points]
Filename: 02_tb.sv
AndrewID: xinyew

```
`default_nettype none

module testbench();
  logic clock, CLOCK_50, reset, data;

  Sender S (.clock(clock),
            .reset(reset),
            .serialOut(data));

  logic [511:0] byteOut;
  logic isNew;
  task2 R (.clock(CLOCK_50),
              .reset(reset),
              .serialIn(data),
              .messageBytes(byteOut),
              .isNew(isNew));

  initial begin
    clock = 1'b0;
    reset = 1'b1;
    //forever #3975 clock = ~clock;
    forever #3776 clock = ~clock;
    // forever #4134 clock = ~clock;
    // forever #4173 clock = ~clock;
  end

  initial begin
    CLOCK_50 = 1'b0;
    forever #1 CLOCK_50 = ~CLOCK_50;
  end

  initial begin
    $monitor($time,, "%10s %10s SR: %b Char: %x %b %b %b String: %s",
             R.control.state.name, R.control.n_state.name,R.reg1.Q, R.m1.Y,
             R.is2bitErr, R.fs_error, R.fe_error,byteOut);
    @(posedge clock);
    reset <= 0;
    @(posedge clock);
  #15000000 $finish;
  end
endmodule : testbench
```