

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

```
1 module SECEDEDdecoder
2     (input logic [12:0] inCode,
3      output logic [3:0] syndrome,
4      output logic is1BitErr, is2BitErr,
5      output logic [12:0] outCode);
6
7     // declare modules
8     makeSyndrome m1(.cw(inCode), .syndrome(syndrome));
9     makeIs1BitErr m2(.syndrome(syndrome), .is1BitErr(is1BitErr),
10                    .cw(inCode));
11     makeIs2BitErr m4(.syndrome(syndrome), .is2BitErr(is2BitErr),
12                    .cw(inCode));
13     makeCorrect m3(.codeWord(inCode), .syndrome(syndrome),
14                   .is1BitErr(is1BitErr), .correctCodeWord(outCode));
15
16 endmodule : SECEDEDdecoder
17
18 module makeSyndrome
19     (input logic [12:0] cw,
20      output logic [3:0] syndrome);
21     assign syndrome[0] = cw[1]^cw[3]^cw[5]^cw[7]^cw[9]^cw[11];
22     assign syndrome[1] = cw[2]^cw[3]^cw[6]^cw[7]^cw[10]^cw[11];
23     assign syndrome[2] = cw[4]^cw[5]^cw[6]^cw[7]^cw[12];
24     assign syndrome[3] = cw[8]^cw[9]^cw[10]^cw[11]^cw[12];
25 endmodule : makeSyndrome
26
27 module makeCorrect
28     (input logic [12:0] codeWord,
29      input logic [3:0] syndrome,
30      input logic is1BitErr,
31      output logic [12:0] correctCodeWord);
32
33     assign correctCodeWord = (is1BitErr) ?
34         codeWord ^ (13'b1 << syndrome) : codeWord;
35 endmodule : makeCorrect
36
37 module makeGlobalParity(
38     input logic [12:0] cw,
39     output logic globalParity);
40     assign globalParity = cw[0]^cw[1]^cw[2]^cw[3]^cw[4]^cw[5]^cw[6] ^
41                          cw[7]^cw[8]^cw[9]^cw[10]^cw[11]^cw[12];
42
43 endmodule : makeGlobalParity
44
45 module makeIs1BitErr
46     (input logic [3:0] syndrome,
47      input logic [12:0] cw,
48      output logic is1BitErr);
49
50     logic globalParity;
51     makeGlobalParity m(.cw(cw), .globalParity(globalParity));
52
53     assign is1BitErr = (globalParity) ? 1 : 0;
54
55 endmodule : makeIs1BitErr
56
57 module makeIs2BitErr
58     (input logic [3:0] syndrome,
59      input logic [12:0] cw,
60      output logic is2BitErr);
61
62     logic globalParity;
63     makeGlobalParity m(.cw(cw), .globalParity(globalParity));
64
65     always_comb begin
66         is2BitErr = 0;
67         if(globalParity == 0) begin
68             if(syndrome != 0) begin
69                 is2BitErr = 1;
```

```
70             end
71         else
72             is2BitErr = 0;
73         end
74     end
75 endmodule : makeIs2BitErr
76
```

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

```
1 `default_nettype none
2
3 // A PIP0 Shift Register, with controllable shift direction
4 // Load has priority over shifting.
5 module LeftShift8Register
6     (input  logic [7:0] D,
7      input  logic en, clock,
8      output logic [511:0] Q);
9
10    always_ff @(posedge clock)
11        if (en)
12            Q <= {Q[503:0], D};
13
14 endmodule : LeftShift8Register
15
16 module Comparator
17     #(parameter WIDTH = 8)
18     (input  logic [WIDTH-1:0] A, B,
19      output logic          AeqB);
20
21    assign AeqB = (A == B);
22    // Straightforward. Uses compare operator.
23
24 endmodule : Comparator
25 // Magnitude comparator
26
27 /*
28 * A library of components, usable for many future hardware designs.
29 */
30
31 // A Magnitude Comparator does an unsigned comparison of two input values.
32 module MagComp
33     #(parameter WIDTH = 8)
34     (output logic          AltB, AeqB, AgtB,
35      input  logic [WIDTH-1:0] A, B);
36
37    assign AeqB = (A == B);
38    assign AltB = (A < B);
39    assign AgtB = (A > B);
40
41 endmodule: MagComp
42
43 // The Multiplexer chooses one of WIDTH bits
44 module Multiplexer
45     #(parameter WIDTH=8)
46     (input  logic [WIDTH-1:0] I,
47      input  logic [$clog2(WIDTH)-1:0] S,
48      output logic          Y);
49
50    assign Y = I[S];
51
52 endmodule : Multiplexer
53
54 // The 2-to-1 Multiplexer chooses one of two multi-bit inputs.
55 module Mux2to1
56     #(parameter WIDTH = 8)
57     (input  logic [WIDTH-1:0] I0, I1,
58      input  logic          S,
59      output logic [WIDTH-1:0] Y);
60
61    assign Y = (S) ? I1 : I0;
62
63 endmodule : Mux2to1
64
65 // The Decoder converts from binary to one-hot codes.
66 module Decoder
67     #(parameter WIDTH=8)
68     (input  logic [$clog2(WIDTH)-1:0] I,
69      input  logic          en,
```

```
70     output logic [WIDTH-1:0]          D);
71
72     always_comb begin
73         D = '0;
74         if (en)
75             D[I] = 1'b1;
76     end
77
78 endmodule : Decoder
79
80 // A DFlipFlop stores the input bit synchronously with the clock signal.
81 // preset and reset are asynchronous inputs.
82 module DFlipFlop
83     (input logic D,
84     input logic preset_L, reset_L, clock,
85     output logic Q);
86
87     always_ff @(posedge clock, negedge preset_L, negedge reset_L)
88         if (~preset_L & reset_L)
89             Q <= 1'b1;
90         else if (~reset_L & preset_L)
91             Q <= 1'b0;
92         else if (~reset_L & ~preset_L)
93             Q <= 1'bX;
94         else
95             Q <= D;
96
97 endmodule : DFlipFlop
98
99 // A Register stores a multi-bit value.
100 // Enable has priority over Clear
101 module Register
102     #(parameter WIDTH=8)
103     (input logic [WIDTH-1:0] D,
104     input logic en, clear, clock,
105     output logic [WIDTH-1:0] Q);
106
107     always_ff @(posedge clock)
108         if (en)
109             Q <= D;
110         else if (clear)
111             Q <= '0;
112
113 endmodule : Register
114
115 // A binary up-down counter.
116 // Clear has priority over Load, which has priority over Enable
117 module Counter
118     #(parameter WIDTH=8)
119     (input logic [WIDTH-1:0] D,
120     input logic en, clear, load, clock, up,
121     output logic [WIDTH-1:0] Q);
122
123     always_ff @(posedge clock)
124         if (clear)
125             Q <= {WIDTH {1'b0}};
126         else if (load)
127             Q <= D;
128         else if (en)
129             if (up)
130                 Q <= Q + 1'b1;
131             else
132                 Q <= Q - 1'b1;
133
134 endmodule : Counter
135
136 // A Synchronizer takes an asynchronous input and changes it to synchronized
137 module Synchronizer
138     (input logic async, clock,
139     output logic sync);
140
```

```

141 logic metastable;
142
143 DFFlipFlop one(.D(async),
144               .Q(metastable),
145               .clock,
146               .preset_L(1'b1),
147               .reset_L(1'b1)
148               );
149
150 DFFlipFlop two(.D(metastable),
151               .Q(sync),
152               .clock,
153               .preset_L(1'b1),
154               .reset_L(1'b1)
155               );
156
157 endmodule : Synchronizer
158
159 // A PIP0 Shift Register, with controllable shift direction
160 // Load has priority over shifting.
161 module ShiftRegister_PIP0
162     #(parameter WIDTH=8)
163     (input logic [WIDTH-1:0] D,
164     input logic en, left, load, clock,
165     output logic [WIDTH-1:0] Q);
166
167     always_ff @(posedge clock)
168         if (load)
169             Q <= D;
170         else if (en)
171             if (left)
172                 Q <= {Q[WIDTH-2:0], 1'b0};
173             else
174                 Q <= {1'b0, Q[WIDTH-1:1]};
175
176 endmodule : ShiftRegister_PIP0
177
178 // A SIPO Shift Register, with controllable shift direction
179 // Load has priority over shifting.
180 module ShiftRegister_SIPO
181     #(parameter WIDTH=8)
182     (input logic serial,
183     input logic en, left, clock,
184     output logic [WIDTH-1:0] Q);
185
186     always_ff @(posedge clock)
187         if (en)
188             if (left)
189                 Q <= {Q[WIDTH-2:0], serial};
190             else
191                 Q <= {serial, Q[WIDTH-1:1]};
192
193 endmodule : ShiftRegister_SIPO
194
195 // A BSR shifts bits to the left by a variable amount
196 module BarrelShiftRegister
197     #(parameter WIDTH=8)
198     (input logic [WIDTH-1:0] D,
199     input logic en, load, clock,
200     input logic [1:0] by,
201     output logic [WIDTH-1:0] Q);
202
203     logic [WIDTH-1:0] shifted;
204     always_comb
205         case (by)
206             default: shifted = Q;
207             2'b01: shifted = {Q[WIDTH-2:0], 1'b0};
208             2'b10: shifted = {Q[WIDTH-3:0], 2'b0};
209             2'b11: shifted = {Q[WIDTH-4:0], 3'b0};
210         endcase
211

```

```
212     always_ff @(posedge clock)
213         if (load)
214             Q <= D;
215         else if (en)
216             Q <= shifted;
217
218     endmodule : BarrelShiftRegister
219
```

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

```
1 `default_nettype none
2
3 module tb();
4     logic [7:0] D;
5     logic [511:0] Q;
6     logic en, load, clock;
7
8     LeftShift8Register DUT (.*);
9
10    initial begin
11        clock = 1;
12        forever #10 clock = ~clock;
13    end
14
15    initial begin
16        $monitor($time,, "D: %x | Q: %x", D, Q);
17    end
18
19    initial begin
20        D <= 8'h11;
21        en <= 1;
22        @(posedge clock);
23        load <= 0;
24        @(posedge clock);
25        @(posedge clock);
26        @(posedge clock);
27        @(posedge clock);
28        @(posedge clock);
29        D <= 8'hFF;
30        @(posedge clock);
31        @(posedge clock);
32        #1 $finish;
33    end
34 endmodule : tb
```

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

```
1 `default_nettype none
2
3 module Receiver
4     (input logic clock, reset, serialIn,
5      output logic [7:0] messageByte,
6      output logic isNew);
7
8     logic is2bitErr, fs_error, done;
9     // for Shift Register
10    logic S_en;
11    // for Out Register
12    logic R_en, R_clear;
13    // for Cycle Counter
14    logic C_en, C_clear;
15    logic [3:0] C_count;
16    // for Error Counter
17    logic E_en, E_clear;
18    logic [3:0] E_count;
19    logic [7:0] mux_out;
20    logic [12:0] received_message, sm;
21    logic [1:0] state, n_state;
22
23    fsm control(.*);
24    ShiftRegister_SIPO #(13) reg1 (.serial(serialIn),
25                                   .en(S_en),
26                                   .left(1),
27                                   .clock(clock),
28                                   .Q(received_message));
29
30    SECEDEDdecoder dec1 (.inCode(received_message),
31                         .is2BitErr(is2bitErr),
32                         .outCode(sm));
33
34    Counter counter1 (.en(C_en), .clear(C_clear), .up(1),
35                    .clock(clock), .Q(C_count));
36
37    Counter counter2 (.en(E_en), .clear(E_clear), .up(1),
38                    .clock(clock), .Q(E_count));
39
40    Comparator comp1 (.A(C_count), .B(4'd12), .AeqB(done));
41
42    Comparator comp2 (.A(E_count), .B(4'd11), .AeqB(fs_error));
43
44    Mux2to1 m1 (.I0({sm[12], sm[11], sm[10], sm[9],
45                   sm[7], sm[6], sm[5], sm[3]}),
46              .I1(8'h15), .S(is2bitErr | fs_error), .Y(mux_out));
47
48    Register reg2 (.en(R_en), .clear(R_clear), .clock(clock),
49                 .D(mux_out), .Q(messageByte));
50
51 endmodule : Receiver
52
53 module fsm
54     (input logic clock, serialIn, reset, done, fs_error,
55      output logic S_en, R_en, R_clear, C_en, C_clear, E_en, E_clear);
56
57     enum logic [1:0] {
58         idle = 2'b00,
59         running = 2'b01,
60         completed = 2'b10,
61         error = 2'b11
62     } state, n_state;
63
64     always_ff @(posedge clock) begin
65         if (reset)
66             state <= idle;
67         else
68             state <= n_state;
69     end
```



```
70
71     always_comb begin
72         case(state)
73             idle : begin
74                 if(fs_error)
75                     n_state = error;
76                 else if(serialIn)
77                     n_state = running;
78                 else if(~serialIn)
79                     n_state = idle;
80
81                 S_en = 0;
82                 // loop counter signal
83                 C_en = 0;
84                 C_clear = 1;
85                 // error counter signal
86                 E_en = 1;
87                 E_clear = 0;
88                 // register signal
89                 R_en = 0;
90                 R_clear = 0;
91             end
92
93             running : begin
94                 n_state = (done) ? completed : running;
95                 S_en = 1;
96                 C_en = 1;
97                 R_en = 0;
98                 R_clear = 0;
99                 E_en = 0;
100                E_clear = 1;
101                C_clear = 0;
102            end
103
104             completed : begin
105                 n_state = (serialIn) ? error : idle;
106                 S_en = 0;
107                 C_en = 0;
108                 R_en = 1;
109                 R_clear = 0;
110                 E_en = 0;
111                 E_clear = 1;
112                 C_clear = 1;
113            end
114             error : begin
115                 n_state = idle;
116                 S_en = 0;
117                 C_en = 0;
118                 R_en = 1;
119                 R_clear = 0;
120                 E_en = 0;
121                 E_clear = 1;
122                 C_clear = 1;
123            end
124         endcase
125     end
126 endmodule : fsm
```

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

```
1 /*
2  * Lab 3a: Transmitter, Task1
3  *
4  */
5
6 module Sender(
7     input logic clock, reset,
8     output logic serialOut);
9
10 parameter WORDS = 25, WORD_SIZE = 64;
11 logic [WORD_SIZE-1:0] message_rom [WORDS-1:0];
12 logic [12:0] word_counter;
13 logic [5:0] bit_counter;
14
15 initial begin
16     $readmemb("01.vm", message_rom);
17 end
18
19 always_ff @(posedge clock, posedge reset) begin
20     if (reset) begin
21         serialOut <= 0;
22         word_counter <= 0;
23         bit_counter <= WORD_SIZE - 1;
24     end
25     else begin
26         if (word_counter < WORDS)
27             serialOut <= message_rom[word_counter][bit_counter];
28         else
29             serialOut <= 0;
30
31         if (bit_counter == 0) begin
32             bit_counter = WORD_SIZE - 1;
33             word_counter = word_counter + 1;
34         end else
35             bit_counter = bit_counter - 1;
36     end
37 end
38
39 endmodule : Sender
```

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

```
1 `default_nettype none
2
3 module testbench();
4     logic clock, reset, data;
5
6     Sender S (.clock(clock),
7               .reset(reset),
8               .serialOut(data));
9
10    logic [7:0] byteOut;
11    logic isNew;
12    Receiver R (.clock(clock),
13               .reset(reset),
14               .serialIn(data),
15               .messageByte(byteOut),
16               .isNew(isNew));
17
18    initial begin
19        clock = 0;
20        reset = 1;
21        forever #10 clock = ~clock;
22    end
23
24    initial begin
25        $monitor($time,, "%8s %8s Mux: %b Char: %h %s %b %b",
26                R.control.state.name, R.control.n_state.name, R.reg1.Q,
27                byteOut, byteOut, R.fs_error, R.is2bitErr);
28        @(posedge clock);
29        reset <= 0;
30        @(posedge clock);
31        #25000 $finish;
32    end
33 endmodule : testbench
```

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

```
1 `default_nettype none
2
3 module task2
4     (input logic clock, reset, serialIn,
5      output logic [511:0] messageBytes,
6      output logic isNew);
7
8     // 2bitErr, frame errors
9     logic is2bitErr, fs_error, fe_error;
10    // time to sample, time to sample next bit;
11    logic timeToSample, timeNextBit;
12    // a block of 13bits received
13    logic blockReceived;
14
15    // for Shift Register
16    logic S_en; // enable shifting
17
18    // for Out Register
19    logic R_en;
20
21    // for Cycle Counter(13-bit output)
22    logic C_en, C_clear;
23    logic [31:0] C_count;
24
25    // for Error Counter(detect fs_error)
26    logic E_en, E_clear;
27    logic [31:0] E_count;
28
29    // for Sample counter(detect synced timing for sampling)
30    logic A_en, A_clear;
31    logic [31:0] A_count;
32
33    // for Wait counter(detect 16 clock cycles to sample next)
34    logic W_en, W_clear;
35    logic [31:0] W_count;
36
37    // for char counter(count char num received)
38    logic Char_en, Char_clear;
39    logic [31:0] Char_count;
40
41    // if fs_error, fe_error or is2BitErr, output 'h15;
42    logic [7:0] mux_out;
43
44    // uncorrected message, corrected message
45    logic [12:0] uncorrected, corrected;
46
47    // FSM for the module
48    fsm control(.*);
49
50    // shift reg to collect bits
51    ShiftRegister_SIPO #(13) reg1 (.serial(serialIn),
52    .en(S_en), .left(1), .clock(clock), .Q(uncorrected));
53
54    // corrector
55    SECDDEDdecoder dec1 (.inCode(uncorrected), .is2BitErr(is2bitErr),
56    .outCode(corrected));
57
58    // Count whether 13-bit block is received
59    Counter #(32) counter1 (.en(C_en), .clear(C_clear), .up(1),
60    .clock(clock), .Q(C_count));
61
62    // Count whether a beginning frame error occurs
63    Counter #(32) counter2 (.en(E_en), .clear(E_clear), .up(1),
64    .clock(clock), .Q(E_count));
65
66    // Count synced timing for sampling
67    Counter #(32) counter3 (.en(A_en), .clear(A_clear), .up(1),
68    .clock(clock), .Q(A_count));
69
```

```

70 // Count wait cycles
71 Counter #(32) counter4 (.en(W_en), .clear(W_clear), .up(1),
72     .clock(clock), .Q(W_count));
73
74 // Count how many chars received
75 Counter #(32) counter5 (.en(Char_en), .clear(Char_clear), .up(1),
76     .clock(clock), .Q(Char_count));
77
78 // Count timings to generate real edges
79 // Counter counter5 (.en(
80
81 // whether a block is received
82 Comparator #(32) comp1 (.A(C_count), .B(32'd13), .AeqB(blockReceived));
83
84 // whether we get a frame error of 10 consecutive 0s
85 Comparator #(32) comp2 (.A(E_count), .B(32'd47_700), .AeqB(fs_error));
86
87 // whether it's time to sample after syncing 8 * 3975
88 Comparator #(32) comp3 (.A(A_count), .B(32'd1_988), .AeqB(timeToSample));
89
90 // whether it's time to sample without seeing an edge
91 Comparator #(32) comp4 (.A(W_count), .B(32'd3_975), .AeqB(timeNextBit));
92
93 // choose from corrected char or error code
94 Mux2to1 m1 (.I0({corrected[12], corrected[11], corrected[10],
95     corrected[9], corrected[7], corrected[6],
96     corrected[5], corrected[3]}),
97     .I1(8'h15), .S(is2bitErr | fs_error | fe_error),
98     .Y(mux_out));
99
100 // reg to store the result
101 LeftShift8Register reg2 (.en(R_en), .clock(clock),
102     .D(mux_out), .Q(messageBytes));
103
104 endmodule : task2
105
106 module fsm
107     (input logic clock, serialIn, reset, timeToSample,
108         blockReceived, fs_error, timeNextBit, is2bitErr,
109         output logic S_en, R_en, fe_error,
110         output logic C_en, C_clear, E_en, E_clear, A_en, A_clear,
111         W_en, W_clear, Char_en, Char_clear);
112
113     enum logic [2:0] {IDLE = 3'b000, SYNC = 3'b001,
114         SAMPLE = 3'b010, WAIT0 = 3'b011,
115         WAIT1 = 3'b100, COMPLETED = 3'b101,
116         ERROR = 3'b110} state, n_state;
117
118     always_ff @(posedge clock, posedge reset) begin
119         if (reset) begin
120             state <= IDLE;
121             Char_clear <= 1;
122         end
123         else begin
124             state <= n_state;
125             Char_clear <= 0;
126         end
127     end
128
129     always_comb begin
130         case(state)
131             IDLE : begin
132                 fe_error = 0;
133                 if(fs_error)
134                     n_state = ERROR;
135                 else if(serialIn)
136                     n_state = SYNC;
137                 else
138                     n_state = IDLE;
139             end
140             // Do not shift

```

```
141         S_en = 0;
142         // keep clearing loop counter
143         C_en = 0;
144         C_clear = 1;
145         // keep counting 0s in IDLE state
146         E_en = 1;
147         E_clear = 0;
148         // do not count sampling timing after syncing
149         A_en = 0;
150         A_clear = 1;
151         // do not wait for edges or next sampling timing without
152         // syncing
153         W_en = 0;
154         W_clear = 1;
155         // do not increase char count, and do not clear it
156         Char_en = 0;
157         // disable register
158         R_en = 0;
159     end
160
161     SYNC : begin
162         fe_error = 0;
163         if (timeToSample)
164             n_state = SAMPLE;
165         else
166             n_state = SYNC;
167         // Do not shift
168         S_en = 0;
169         // Do not increase loop count and do not clear it
170         C_en = 0;
171         C_clear = 0;
172         // keep clearing frame error counter
173         E_en = 0;
174         E_clear = 1;
175         // start counting sampling timing after syncing
176         A_en = 1;
177         A_clear = 0;
178         // do not wait for edges or next sampling timing without
179         // syncing
180         W_en = 0;
181         W_clear = 1;
182         // do not increase char count, and do not clear it
183         Char_en = 0;
184         // disable register
185         R_en = 0;
186     end
187
188     SAMPLE : begin
189         fe_error = 0;
190         if (serialIn)
191             n_state = WAIT1;
192         else
193             n_state = WAIT0;
194         // Shift once
195         S_en = 1;
196         // Increase num of bit collected by 1
197         C_en = 1;
198         C_clear = 0;
199         if (blockReceived)
200             n_state = COMPLETED;
201         // keep clearing frame error counter
202         E_en = 0;
203         E_clear = 1;
204         // clearing counting sampling timing after syncing
205         A_en = 0;
206         A_clear = 1;
207         // do not wait for next sampling timing without
208         // syncing
209         W_en = 0;
210         W_clear = 1;
211         // do not increase char count, and do not clear it
```

```
212         Char_en = 0;
213         // disable register
214         R_en = 0;
215     end
216
217     WAIT0 : begin
218         fe_error = 0;
219         if (serialIn)
220             n_state = SYNC;
221         else if (timeNextBit)
222             n_state = SAMPLE;
223         else
224             n_state = WAIT0;
225         // Stop shifting
226         S_en = 0;
227         // Stop counting loop counter, do not clear
228         C_en = 0;
229         C_clear = 0;
230         // keep clearing frame error counter
231         E_en = 0;
232         E_clear = 1;
233         // Stop counting sampling timing and clear it
234         A_en = 0;
235         A_clear = 1;
236         // Counting next sampling timing without syncing
237         W_en = 1;
238         W_clear = 0;
239         // do not increase char count, and do not clear it
240         Char_en = 0;
241         // disable register
242         R_en = 0;
243     end
244
245     WAIT1 : begin
246         fe_error = 0;
247         if (~serialIn)
248             n_state = SYNC;
249         else if (timeNextBit)
250             n_state = SAMPLE;
251         else
252             n_state = WAIT1;
253         // Stop shifting
254         S_en = 0;
255         // Stop counting loop counter, do not clear
256         C_en = 0;
257         C_clear = 0;
258         // keep clearing frame error counter
259         E_en = 0;
260         E_clear = 1;
261         // Stop counting sampling timing and clear it
262         A_en = 0;
263         A_clear = 1;
264         // Counting next sampling timing without syncing
265         W_en = 1;
266         W_clear = 0;
267         // do not increase char count, and do not clear it
268         Char_en = 0;
269         // disable register
270         R_en = 0;
271     end
272
273     COMPLETED : begin
274         if (~timeNextBit) begin
275             n_state = COMPLETED;
276             fe_error = 0;
277             R_en = 0;
278             Char_en = 0;
279             W_en = 1;
280             W_clear = 0;
281         end
282         else if (is2bitErr || serialIn) begin
```

```
283         n_state = ERROR;
284         fe_error = 1;
285         R_en = 1;
286         Char_en = 1;
287         W_en = 0;
288         W_clear = 1;
289     end
290     else begin
291         n_state = IDLE;
292         fe_error = 0;
293         R_en = 1;
294         Char_en = 1;
295         W_en = 1;
296         W_clear = 0;
297     end
298     // Stop shifting
299     S_en = 0;
300     // Stop counting loop counter, clear it
301     C_en = 0;
302     C_clear = 1;
303     // keep clearing frame error counter
304     E_en = 0;
305     E_clear = 1;
306     // Stop counting sampling timing and clear it
307     A_en = 0;
308     A_clear = 1;
309     // increase char count, and do not clear it
310     // enable register
311 end
312
313 ERROR : begin
314     fe_error = 0;
315     if (~timeNextBit)
316         n_state = ERROR;
317     else
318         n_state = IDLE;
319         // Stop shifting
320         S_en = 0;
321         // Stop counting loop counter, clear it
322         C_en = 0;
323         C_clear = 1;
324         // keep clearing frame error counter
325         E_en = 0;
326         E_clear = 1;
327         // Stop counting sampling timing and clear it
328         A_en = 0;
329         A_clear = 1;
330         // Stop counting next sampling timing without syncing, clear
331         W_en = 1;
332         W_clear = 0;
333         // do not increase char count, and do not clear it
334         Char_en = 0;
335         // enable register
336         R_en = 0;
337     end
338
339 endcase
340 end
341 endmodule : fsm
```


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

```
1 `default_nettype none
2 module chipInterface(
3     input logic CLOCK_50,
4     input logic [17:0] SW,
5     input logic UART_RXD,
6     input logic [3:0] KEY,
7     output logic [6:0] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0
8 );
9 // here we declare the output
10 logic [511:0] m; // this should be connected to out
11 logic [511:0] mm; // this should be connected to out
12
13 logic [31:0] selected;
14 logic reset;
15 assign reset = ~KEY[0];
16 logic isNew;
17
18 // here we declare our receiver
19 task2 R (.clock(CLOCK_50),
20         .reset(reset),
21         .serialIn(UART_RXD),
22         .messageBytes(mm),
23         .isNew(isNew));
24
25 // here we declare the counter
26 select s (.bits(mm),
27         .addr({SW[7:0]}),
28         .out(selected));
29
30 logic [7:0] blank;
31 assign blank = 8'd0;
32
33 SevenSegmentDisplay s1 (.BCX0(selected[31:28]), .blank(blank), .HEX0(HEX7));
34 SevenSegmentDisplay s2 (.BCX0(selected[27:24]), .blank(blank), .HEX0(HEX6));
35 SevenSegmentDisplay s3 (.BCX0(selected[23:20]), .blank(blank), .HEX0(HEX5));
36 SevenSegmentDisplay s4 (.BCX0(selected[19:16]), .blank(blank), .HEX0(HEX4));
37 SevenSegmentDisplay s5 (.BCX0(selected[15:12]), .blank(blank), .HEX0(HEX3));
38 SevenSegmentDisplay s6 (.BCX0(selected[11:8]), .blank(blank), .HEX0(HEX2));
39 SevenSegmentDisplay s7 (.BCX0(selected[7:4]), .blank(blank), .HEX0(HEX1));
40 SevenSegmentDisplay s8 (.BCX0(selected[3:0]), .blank(blank), .HEX0(HEX0));
41
42
43 endmodule : chipInterface
44
45
46 module select
47 (input logic [511:0] bits,
48  input logic [7:0] addr,
49  output logic [31:0] out);
50
51 logic [511:0] tmp;
52 assign tmp = bits >> (addr * 32);
53 assign out = tmp[31:0];
54
55 endmodule : select
56
57
58 module SevenSegmentDisplay
59 (input logic [3:0] BCX0,
60  input logic [7:0] blank,
61  output logic [6:0] HEX0);
62
63 always_comb begin
64     HEX0 = 7'b0000000;
65     if (~blank[0])
66         case (BCX0)
67             4'h0: HEX0 = 7'b0111111;
68             4'h1: HEX0 = 7'b0000110;
69             4'h2: HEX0 = 7'b1011011;
```

```
70      4'h3: HEX0 = 7'b1001111;
71      4'h4: HEX0 = 7'b1100110;
72      4'h5: HEX0 = 7'b1101101;
73      4'h6: HEX0 = 7'b1111101;
74      4'h7: HEX0 = 7'b0000111;
75      4'h8: HEX0 = 7'b1111111;
76      4'h9: HEX0 = 7'b1100111;
77      4'ha: HEX0 = 7'b1110111;
78      4'hb: HEX0 = 7'b1111100;
79      4'hc: HEX0 = 7'b0111001;
80      4'hd: HEX0 = 7'b1011110;
81      4'he: HEX0 = 7'b1111001;
82      4'hf: HEX0 = 7'b1110001;
83      default: HEX0 = 7'b0000000;
84  endcase
85      HEX0 = ~HEX0;
86  end
87
88
89  endmodule : SevenSegmentDisplay
90
```

Lab Code [10 points]

Filename: 02_clockDriver.sv

AndrewID: xinyew

```
1 module clock_divider
2   #(parameter equiv_cycle = 3975)
3   (input logic en, clock, reset,
4     output logic new_clock);
5
6   logic [$clog2(equiv_cycle)-1:0] Q;
7   logic clock_edge, clear;
8
9   // declare control fsm
10  clock_fsm control(.);
11
12  // Counter to count up to equiv cycle
13  Counter dut1(.en(en), .clear(clear), .clock(clock), .up(1), .Q(Q));
14
15  // Comparator to check whether we have hit equiv cycle
16  Comparator dut2(.A(Q), .B(equiv_cycle), .AeqB(clock_edge));
17
18 endmodule : clock_divider
19
20 module clock_fsm
21   (input logic clock, clock_edge, reset,
22     output logic new_clock, clear);
23
24   // here we declare the possible states
25   enum logic {out0, out1} state, n_state;
26
27   // flip-flop for next states
28   always_ff @(posedge clock) begin
29     if(reset)
30       state <= out0;
31     else
32       state <= n_state;
33   end
34
35   // next state generation
36   always_comb begin
37     case(state)
38       out0 : begin
39         n_state = (clock_edge) ? out1 : out0;
40         clear = (clock_edge) ? 1 : 0;
41         new_clock = 0;
42       end
43       out1 : begin
44         n_state = (clock_edge) ? out0 : out1;
45         clear = (clock_edge) ? 1 : 0;
46         new_clock = 1;
47       end
48     endcase
49   end
50
51 endmodule : clock_fsm
52
53 // this is a test bench for out clock divider
54
55 module clock_divider_test();
56   // declare the variables
57   logic en, reset, clear, clock, new_clock;
58
59   // here we make the clock run at 50 Mhz
60   initial begin
61     clock = 0;
62     forever #2ns clock = ~clock;
63   end
64
65   // here we declare the clock_divider module
66   clock_divider #(5) dut(.);
67
68   // begin test_bench
69   initial begin
```

```
70         reset = 1;
71         en = 1;
72         clear = 1;
73         #1ns;
74         clear = 0;
75         reset = 0;
76         $monitor($time,, "new clock: %b", new_clock);
77         #10ms;
78         $finish;
79     end
80
81 endmodule: clock_divider_test
82
```

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

```
1 `default_nettype none
2
3 module BusDriver
4   #(parameter WIDTH = 8)
5   (input logic en,
6    input logic [WIDTH-1:0] data,
7    output logic [WIDTH-1:0] buff,
8    inout tri [WIDTH-1:0] bus
9   );
10
11   assign bus = (en) ? data : 'bz;
12   assign buff = bus;
13
14 endmodule : BusDriver
15
16 module Memory
17   #(parameter DW = 16,
18           W = 256,
19           AW = $clog2(W))
20   (input logic re, we, clock,
21    input logic [AW-1:0] addr,
22    inout tri [DW-1:0] data);
23
24   logic [DW-1:0] M[W];
25   logic [DW-1:0] rData;
26
27   assign data = (re) ? rData : 'bz;
28
29   always_ff @(posedge clock)
30     if (we)
31       M[addr] <= data;
32
33   always_comb
34     rData = M[addr];
35
36 endmodule : Memory
```

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

```
1 `default_nettype none
2
3 module testbench();
4     logic clock, CLOCK_50, reset, data;
5
6     Sender S (.clock(clock),
7               .reset(reset),
8               .serialOut(data));
9
10    logic [511:0] byteOut;
11    logic isNew;
12    task2 R (.clock(CLOCK_50),
13            .reset(reset),
14            .serialIn(data),
15            .messageBytes(byteOut),
16            .isNew(isNew));
17
18    initial begin
19        clock = 1'b0;
20        reset = 1'b1;
21        //forever #3975 clock = ~clock;
22        forever #3776 clock = ~clock;
23        // forever #4134 clock = ~clock;
24        // forever #4173 clock = ~clock;
25    end
26
27    initial begin
28        CLOCK_50 = 1'b0;
29        forever #1 CLOCK_50 = ~CLOCK_50;
30    end
31
32    initial begin
33        $monitor($time,, "%10s %10s SR: %b Char: %x %b %b %b String: %s",
34                  R.control.state.name, R.control.n_state.name, R.reg1.Q, R.m1.Y,
35                  R.is2bitErr, R.fs_error, R.fe_error, byteOut);
36        @(posedge clock);
37        reset <= 0;
38        @(posedge clock);
39        #15000000 $finish;
40    end
41 endmodule : testbench
```