```
1   `timescale 1ns / 1ns // `timescale time_unit/time_precision
2
3   /* top-level entity for the morse code generator */
4   module lab5Part3 (input [2:0] SW, input [1:0] KEY, input CLOCK_50, output [9:0] LEDR);
5
6               assign LEDR[9:1] = 9'd0;
7
8                morseCode_Generator M1 (.alphabet(SW[1:0]), .reset(KEY[0]), .clock(CLOCK_50),
9                                        .msg_Send(KEY[1]), .morseQueue_Top(LEDR[0]));
10
11  endmodule // lab5Part3
12
13  /*  */
14  module morseCode_Generator (input [2:0] alphabet, input reset, clock, msg_Send,
15                              output morseQueue_Top);
16
17              wire [12:0] morse_Binary;
18              wire dwnClk_Enable;
19              wire [12:0] queueMorse;
20              assign morseQueue_Top = queueMorse[0];
21              morseCode_Mux M1 (.sel(alphabet), .morseCode(morse_Binary));
22              rateDivider R1 (.clock(clock), .reset(reset), .downClock(dwnClk_Enable));
23              universal_ShiftReg U1 (.DATA_IN(morse_Binary), .ParallelLoad_n(msg_Send),
24                                     .clock(dwnClk_Enable), .reset(reset), .Q_out(queueMorse));
25
26  endmodule // morseCode_Generator
27
28  /* binary-coded morse selection */
29  module morseCode_Mux (input sel, output reg [12:0] morseCode);
30
31              always @(*)
32              begin
33
34                 case (sel)
35                    3'b000: morseCode = 13'b1010100000000; // letter S
36                    3'b001: morseCode = 13'b1110000000000; // letter T
37                    3'b010: morseCode = 13'b1010111000000; // letter U
38                    3'b011: morseCode = 13'b1010101110000; // letter V
39                    3'b100: morseCode = 13'b1011101110000; // letter W
40                    3'b101: morseCode = 13'b1110101011100; // letter X
41                    3'b110: morseCode = 13'b1110101110111; // letter Y
42                    3'b001: morseCode = 13'b1110111010100; // letter Z
43                 endcase
44
45              end
46
47  endmodule // morseCode_Mux
48
49  /* downclocks input 50 MHz clock to 2Hz */
50  module rateDivider(input clock, reset, output downClock);
51
52              reg [25:0] cycleCount;
53
54              always @(posedge clock) // triggered on edges of clock
55              begin
56
57                 if (reset == 1'b0) // synchronous active -low
58                    cycleCount <= 26'd0;
59                 else if (cycleCount == 26'd0)
60                    cycleCount <= 26'd25000000; // reset counter to 25M
61                 else
62                    cycleCount <= cycleCount - 1'b1; // decrement state
63
64              end
65
66              assign downClock = (cycleCount == 26'd0) ? (1'b1):(1'b0);
67
68  endmodule // rateDivider
69
70  /* universal shift register, being used to store (reversed) morse code */
71  module universal_ShiftReg (input [12:0] DATA_IN, ParallelLoad_n, clock, reset,
72                             output [12:0] Q_out);
73
74              wire [12:0] Q; // carries the output of the flip-flops and subsequent connections
75              assign Q_out = Q; // assigning outputs of flip-flops to the register output
76
```

```
77                // instantiation of all 13 flip-flops for the register
78                flipFlop F12 (.clock(clock), .reset(reset), .load_n(ParallelLoad_n),
79                            .loadLeft(1'b1), .left(Q[0]), .right(Q[11]), .data(DATA_IN[0]),
80                            .LSRight(1'b1), .Q(Q[12])); // flip-flop for bit 12
81
82                flipFlop F11 (.clock(clock), .reset(reset), .load_n(ParallelLoad_n),
83                            .loadLeft(1'b1), .left(Q[12]), .right(Q[10]), .data(DATA_IN[1]),
84                            .LSRight(1'b0), .Q(Q[11])); // flip-flop for bit 11
85
86                flipFlop F10 (.clock(clock), .reset(reset), .load_n(ParallelLoad_n),
87                            .loadLeft(1'b1), .left(Q[11]), .right(Q[9]), .data(DATA_IN[2]),
88                            .LSRight(1'b0), .Q(Q[10])); // flip-flop for bit 10
89
90                flipFlop F9 (.clock(clock), .reset(reset), .load_n(ParallelLoad_n),
91                            .loadLeft(1'b1), .left(Q[10]), .right(Q[8]), .data(DATA_IN[3]),
92                            .LSRight(1'b0), .Q(Q[9])); // flip-flop for bit 9
93
94                flipFlop F8 (.clock(clock), .reset(reset), .load_n(ParallelLoad_n),
95                            .loadLeft(1'b1), .left(Q[9]), .right(Q[7]), .data(DATA_IN[4]),
96                            .LSRight(1'b0), .Q(Q[8])); // flip-flop for bit 8
97
98                flipFlop F7 (.clock(clock), .reset(reset), .load_n(ParallelLoad_n),
99                            .loadLeft(1'b1), .left(Q[8]), .right(Q[6]), .data(DATA_IN[5]),
100               .LSRight(1'b0), .Q(Q[7])); // flip-flop for bit 7
101
102               flipFlop F6 (.clock(clock), .reset(reset), .load_n(ParallelLoad_n),
103                           .loadLeft(1'b1), .left(Q[7]), .right(Q[5]), .data(DATA_IN[6]),
104                           .LSRight(1'b0), .Q(Q[6])); // flip-flop for bit 6
105
106               flipFlop F5 (.clock(clock), .reset(reset), .load_n(ParallelLoad_n),
107                           .loadLeft(1'b1), .left(Q[6]), .right(Q[4]), .data(DATA_IN[7]),
108                           .LSRight(1'b0), .Q(Q[5])); // flip-flop for bit 5
109
110               flipFlop F4 (.clock(clock), .reset(reset), .load_n(ParallelLoad_n),
111                           .loadLeft(1'b1), .left(Q[5]), .right(Q[3]), .data(DATA_IN[8]),
112                           .LSRight(1'b0), .Q(Q[4])); // flip-flop for bit 4
113
114               flipFlop F3 (.clock(clock), .reset(reset), .load_n(ParallelLoad_n),
115                           .loadLeft(1'b1), .left(Q[4]), .right(Q[2]), .data(DATA_IN[9]),
116                           .LSRight(1'b0), .Q(Q[3])); // flip-flop for bit 3
117
118               flipFlop F2 (.clock(clock), .reset(reset), .load_n(ParallelLoad_n),
119                           .loadLeft(1'b1), .left(Q[3]), .right(Q[1]), .data(DATA_IN[10]),
120                           .LSRight(1'b0), .Q(Q[2])); // flip-flop for bit 2
121
122               flipFlop F1 (.clock(clock), .reset(reset), .load_n(ParallelLoad_n),
123                           .loadLeft(1'b1), .left(Q[2]), .right(Q[0]), .data(DATA_IN[11]),
124                           .LSRight(1'b0), .Q(Q[1])); // flip-flop for bit 1
125
126               flipFlop F0 (.clock(clock), .reset(reset), .load_n(ParallelLoad_n),
127                           .loadLeft(1'b1), .left(Q[1]), .right(Q[12]), .data(DATA_IN[12]),
128                           .LSRight(1'b0), .Q(Q[0])); // flip-flop for bit 0 (LSB)
129
130   endmodule // universal_ShiftReg
131
132   /* flip-flop with multiplexers to select input */
133   module flipFlop (input clock, reset, load_n, data, loadLeft, right, left,
134                  LSRight, output reg Q);
135
136               wire R, D;
137
138               /*
139                * loadleft = 1 will select left, and right otherwise
140                * load_n = 0 will select data, and output of rotation select otherwise
141               */
142
143               mux2to1 M1 (.x(right), .y(left), .sel(loadLeft), .f(R)); // left-right rotate
      option select
144               mux2to1 M2 (.x(data), .y(R), .sel(load_n), .f(D)); // parallel load and rotate
      select
145
146               // triggered on rising edge of the clock signal and falling edge of clear
147               always @(posedge clock, negedge reset)
148               begin
149
150                  if (reset == 1'b0) // active-low, asynchronous reset to 0
```

```
151                     Q <= 1'b0;
152                 else if (LSRight == 1'b1 && loadLeft == 1'b1) // override D during logical
    shift right
153                     Q <= 1'b0;
154                 else
155                     Q <= D; // if reset and LSRight are 0, flip-flop tracks D
156
157             end
158
159 endmodule // flipFlop
160
161 /* 2 to 1 multiplexer */
162 module mux2to1(input x, y, sel, output f);
163
164             assign f = sel ? y : x; // f = y when sel = 1, x otherwise
165
166 endmodule // mux2to1
```