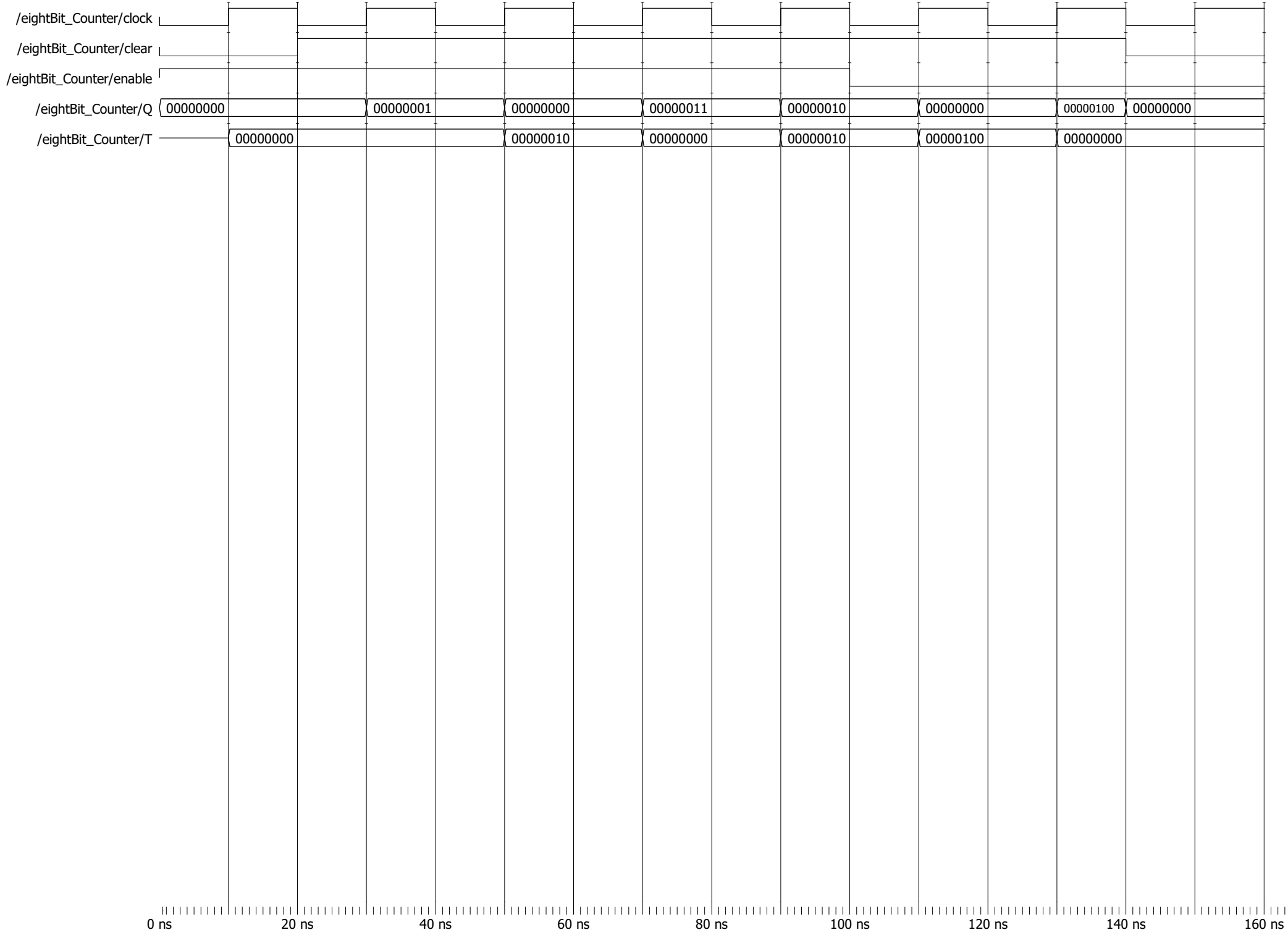
```
1   `timescale 1ns / 1ns // `timescale time_unit/time_precision
2
3   /* top-level entity for the 8-bit counter */
4   module lab5Part1 (input [1:0] SW, input [0:0] KEY, output [9:0] LEDR,
5                     output [6:0] HEX0, HEX1);
6
7               assign LEDR[9:0] = 10'b0000000000; // all LEDs remain off
8               wire [7:0] Q_state; // current state of the counter circuit
9
10              /*
11               * KEY[0] is the clock signal
12               * SW[0] is the clear signal
13               * SW[1] is the enable signal
14               * HEX0, HEX1 are for display of inputs and output
15               * Rest HEX[k] stay off (un-initialized)
16               */
17
18               eightBit_Counter C1 (.clock(~KEY[0]), .clear(SW[0]),
19                                    .enable(SW[1]), .Q(Q_state)); // instantiating 8-bit counter
20
21               BCD_to_HEX_Decoder D1 (.C(Q_state[3:0]), .HEX(HEX0)); // display LSB of Q
22               BCD_to_HEX_Decoder D2 (.C(Q_state[7:4]), .HEX(HEX1)); // display MSB of Q
23
24  endmodule // lab5part1
25
26  /* 8-bit counter using 8 serial t_flip-flops */
27  module eightBit_Counter (input clock, clear, enable, output [7:0] Q);
28
29              wire [7:0] T; // (N-1) = 7-bit bus to track toggle inputs. T[0] not used
30              t_flipFlop T0 (.T(enable), .clock(clock), .clear(clear), .Q(Q[0]), .Q_p1(T[1]));
     // bit 0
31              t_flipFlop T1 (.T(T[1]), .clock(clock), .clear(clear), .Q(Q[1]), .Q_p1(T[2]));
    // bit 1
32              t_flipFlop T2 (.T(T[2]), .clock(clock), .clear(clear), .Q(Q[2]), .Q_p1(T[3]));
    // bit 2
33              t_flipFlop T3 (.T(T[3]), .clock(clock), .clear(clear), .Q(Q[3]), .Q_p1(T[4]));
    // bit 3
34              t_flipFlop T4 (.T(T[4]), .clock(clock), .clear(clear), .Q(Q[4]), .Q_p1(T[5]));
    // bit 4
35              t_flipFlop T5 (.T(T[5]), .clock(clock), .clear(clear), .Q(Q[5]), .Q_p1(T[6]));
    // bit 5
36              t_flipFlop T6 (.T(T[6]), .clock(clock), .clear(clear), .Q(Q[6]), .Q_p1(T[7]));
    // bit 6
37              t_flipFlop T7 (.T(T[7]), .clock(clock), .clear(clear), .Q(Q[7]), .Q_p1(T[0]));
    // bit 7
38
39  endmodule // eightBit_Counter
40
41  /* t-flip flop module with asynchronous clear */
42  module t_flipFlop (input T, clock, clear, output reg Q, Q_p1);
43
44              // triggered on rising edge of the clock signal and falling edge of clear
45              always @(posedge clock, negedge clear)
46              begin
47
48                 if (clear == 1'b0)
49                    Q <= 1'b0; // active-low, asynchronous reset to 0
50                 else if (T == 1'b0)
51                    Q <= Q; // if toggle is 0, t_flip-flop maintains state
52                 else
53                    Q <= ~Q; // if toggle is 1, t_flip-flop changes state to T'
54
55                 Q_p1 = Q & T; // T for the next t_flip-flop
56
57              end
58
59  endmodule // t_flipFlop
60
61
62  /* BCD to common-anode seven-segment display decoder */
63  module BCD_to_HEX_Decoder (input [3:0] C, output [6:0] HEX);
64
65              // maxterms for every segment LEDs with common anode
66              assign HEX[0] = !((C[3]|C[2]|C[1]|!C[0]) & (C[3]|!C[2]|C[1]|C[0]) &
67                           (!C[3]|C[2]|!C[1]|!C[0]) & (!C[3]|!C[2]|C[1]|!C[0]));
68
```

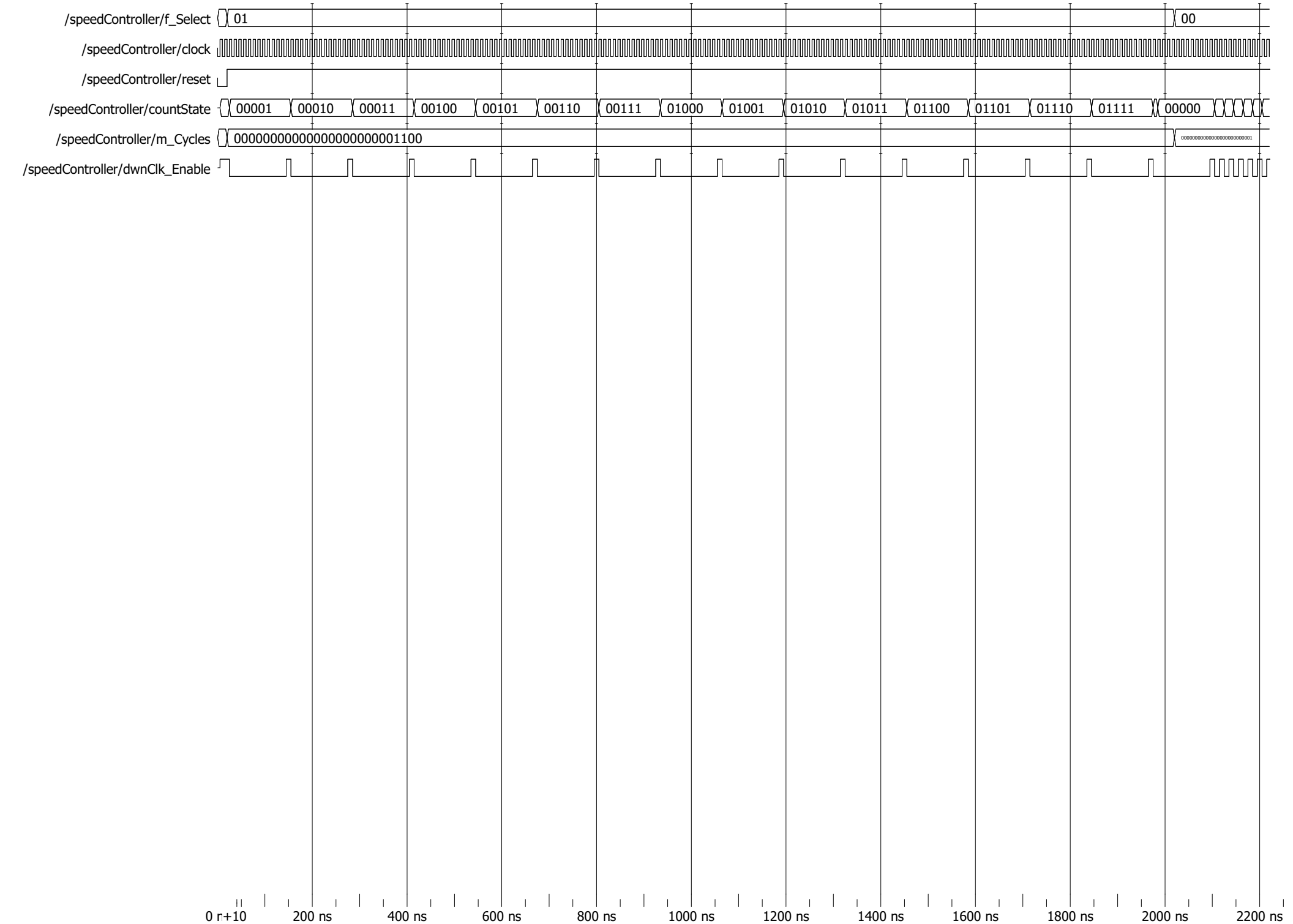
```
69                assign HEX[1] = !((C[3]||!C[2]|C[1]||!C[0]) & (C[3]||!C[2]||C[1]|C[0]) &
70                                  (!C[3]|C[2]||!C[1]||!C[0]) & (!C[3]||C[2]|C[1]|C[0]) &
71                                  (!C[3]||!C[2]||C[1]|C[0]) & (!C[3]||C[2]||!C[1]||!C[0]));
72
73                assign HEX[2] = !((C[3]|C[2]||!C[1]|C[0]) & (!C[3]||!C[2]|C[1]|C[0]) &
74                                  (!C[3]||!C[2]||!C[1]|C[0]) & (!C[3]||C[2]||!C[1]||!C[0]));
75
76                assign HEX[3] = !((C[3]|C[2]|C[1]||!C[0]) & (C[3]||!C[2]|C[1]|C[0]) &
77                                  (C[3]||!C[2]||!C[1]||!C[0]) & (!C[3]|C[2]|C[1]||!C[0]) &
78                                  (!C[3]|C[2]||!C[1]|C[0]) & (!C[3]||!C[2]||!C[1]||!C[0]));
79
80                assign HEX[4] = !((C[3]|C[2]|C[1]||!C[0]) & (C[3]|C[2]||!C[1]||!C[0]) &
81                                  (C[3]||!C[2]|C[1]|C[0]) & (C[3]||!C[2]|C[1]|C[0]) &
82                                  (C[3]||!C[2]|C[1]||!C[0]) & (C[3]||!C[2]||!C[1]||!C[0]) &
83                                  (!C[3]|C[2]|C[1]||!C[0]));
84
85                assign HEX[5] = !((C[3]|C[2]|C[1]||!C[0]) & (C[3]|C[2]||!C[1]|C[0]) &
86                                  (C[3]|C[2]||!C[1]||!C[0]) & (C[3]||!C[2]||!C[1]||!C[0]) &
87                                  (!C[3]||!C[2]|C[1]||!C[0]));
88
89                assign HEX[6] = !((C[3]|C[2]|C[1]|C[0]) & (C[3]|C[2]|C[1]||!C[0]) &
90                                  (C[3]||!C[2]||!C[1]||!C[0]) & (!C[3]||!C[2]|C[1]|C[0]));
91
92    endmodule // BCD_to_HEX_Decoder
```



Entity:eightBit_Counter Architecture: Date: Mon Oct 22 10:19:47 EDT 2018 Row: 1 Page: 1

```
1   `timescale 1ns / 1ns // `timescale time_unit/time_precision
2
3   /* top-level entity for the speed controlled counter */
4   module lab5Part2 (input [2:0] SW, input CLOCK_50, output [9:0] LEDR, output [6:0] HEX0);
5
6               assign LEDR[9:0] = 10'b0000000000; // all LEDs remain off
7               wire [4:0] disp_Count;
8
9               speedController S1 (.f_Select(SW[1:0]), .clock(CLOCK_50),
10                                  .reset(SW[2]), .countState(disp_Count));
11              BCD_to_HEX_Decoder D1 (.C(disp_Count[3:0]), .HEX(HEX0));
12
13  endmodule // lab5Part2
14
15  /*  */
16  module speedController (input [1:0] f_Select, input clock, reset,
17                          output [4:0] countState);
18
19              wire [25:0] m_Cycles; // to connect max cycles.
20              wire dwnClk_Enable; // downclocked, synchronous enable for counter
21
22              speedSelect S1 (.sel(f_Select), .maxCycles(m_Cycles));
23              rateDivider R1 (.maxCycles(m_Cycles), .clock(clock), .reset(reset),
24                              .downClock(dwnClk_Enable));
25              fourBit_Counter C1 (.clock(clock), .reset(reset),
26                                  .enable(dwnClk_Enable), .Q(countState));
27
28  endmodule // speedController
29
30  /**/
31  module speedSelect (input [1:0] sel, output reg [25:0] maxCycles);
32
33              always @(*)
34              begin
35
36                 case(sel)
37                    2'b00: maxCycles = 26'd1; // 50MHz
38                    2'b01: maxCycles = 26'd12500000; // 4Hz
39                    2'b10: maxCycles = 26'd25000000; // 2Hz
40                    2'b11: maxCycles = 26'd50000000; // 1Hz
41                    default: maxCycles = 26'd50000000; // dwef
42                 endcase
43
44              end
45
46  endmodule // speedSelect
47
48  /* downclocks input 50 MHz clock for feeding into other modules */
49  module rateDivider(input [25:0] maxCycles, input clock, reset, output downClock);
50
51              reg [25:0] cycleCount;
52
53              always @(posedge clock) // triggered on edges of clock
54              begin
55
56                 if (reset == 1'b0) // synchronous active -low
57                    cycleCount <= 26'd0;
58                 else if (cycleCount == 26'd0)
59                    cycleCount <= maxCycles; // reset counter to 50M
60                 else
61                    cycleCount <= cycleCount - 1'b1; // decrement state
62
63              end
64
65              assign downClock = (cycleCount == 26'd0) ? (1'b1):(1'b0);
66
67  endmodule // rateDivider
68
69  /* 4-bit counter to which accepts the downclocked enable */
70  module fourBit_Counter (input clock, reset, enable, output reg [4:0] Q);
71
72              always @(posedge clock) // triggered on rising edge of clock
73              begin
74
75                 if (reset == 1'd0) // synch reset active-low
76                    Q <= 5'd0;
```

```
77                  else if (Q == 5'd16) // max vvalll
78                     Q <= 5'd0;
79                  else if (enable == 1'd1) // increment on enable
80                     Q <= Q + 1;
81                  else
82                     Q <= Q;
83
84               end
85
86   endmodule // fourBit_Counter
87
88   /* BCD to common-anode seven-segment display decoder */
89   module BCD_to_HEX_Decoder (input [3:0] C, output [6:0] HEX);
90
91               // maxterms for every segment LEDs with common anode
92               assign HEX[0] = !((C[3]|C[2]|C[1]|!C[0]) & (C[3]|!C[2]|C[1]|C[0]) &
93                           (!C[3]|C[2]|!C[1]|!C[0]) & (!C[3]|!C[2]|C[1]|!C[0]));
94
95               assign HEX[1] = !((C[3]|!C[2]|C[1]|!C[0]) & (C[3]|!C[2]|!C[1]|C[0]) &
96                           (!C[3]|C[2]|!C[1]|!C[0]) & (!C[3]|!C[2]|C[1]|C[0]) &
97                           (!C[3]|!C[2]|!C[1]|C[0]) & (!C[3]|!C[2]|!C[1]|!C[0]));
98
99               assign HEX[2] = !((C[3]|C[2]|!C[1]|C[0]) & (!C[3]|!C[2]|C[1]|C[0]) &
100                          (!C[3]|!C[2]|!C[1]|C[0]) & (!C[3]|!C[2]|!C[1]|!C[0]));
101
102              assign HEX[3] = !((C[3]|C[2]|C[1]|!C[0]) & (C[3]|!C[2]|C[1]|C[0]) &
103                          (C[3]|!C[2]|!C[1]|!C[0]) & (!C[3]|C[2]|C[1]|!C[0]) &
104                          (!C[3]|C[2]|!C[1]|C[0]) & (!C[3]|!C[2]|!C[1]|!C[0]));
105
106              assign HEX[4] = !((C[3]|C[2]|C[1]|!C[0]) & (C[3]|C[2]|!C[1]|!C[0]) &
107                          (C[3]|!C[2]|C[1]|C[0]) & (C[3]|!C[2]|C[1]|C[0]) &
108                          (C[3]|!C[2]|C[1]|!C[0]) & (C[3]|!C[2]|!C[1]|!C[0]) &
109                          (!C[3]|C[2]|C[1]|!C[0]));
110
111              assign HEX[5] = !((C[3]|C[2]|C[1]|!C[0]) & (C[3]|C[2]|!C[1]|C[0]) &
112                          (C[3]|C[2]|!C[1]|!C[0]) & (C[3]|!C[2]|!C[1]|!C[0]) &
113                          (!C[3]|!C[2]|C[1]|!C[0]));
114
115              assign HEX[6] = !((C[3]|C[2]|C[1]|C[0]) & (C[3]|C[2]|C[1]|!C[0]) &
116                          (C[3]|!C[2]|!C[1]|!C[0]) & (!C[3]|!C[2]|C[1]|C[0]));
117
118  endmodule // BCD_to_HEX_Decoder
```

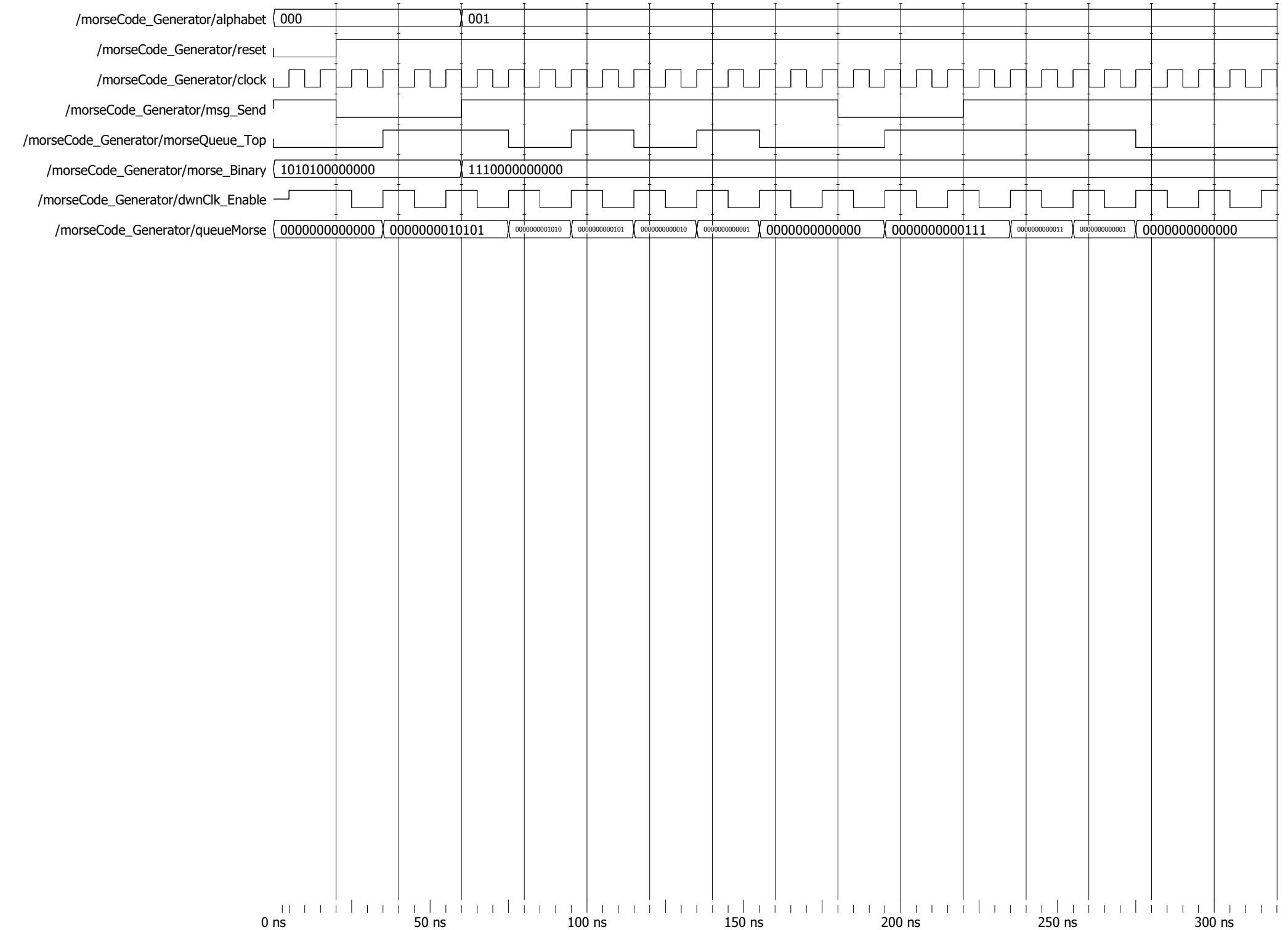


Entity:speedController Architecture: Date: Mon Oct 22 08:30:28 EDT 2018 Row: 1 Page: 1

```
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
```



Entity:morseCode_Generator Architecture: Date: Mon Oct 22 03:05:28 EDT 2018 Row: 1 Page: 1