UNIVERSITY OF LIVERPOOL

ELEC373 ASSIGNMENT 2

Designing Electronic Safe

Assessor:
Prof. Jeremy Smith

Abstract

This is the report of the ELEC373 assignment 2. ASM charts, block diagrams, Verilog code, and simulation results of modules are presented with explanations. The full system were tested on both board and simulator, and the simulation result are explained.

Declaration of academic integrity

I confirm that I have read and understood the University's definitions of plagiarism and collusion from the Code of Practice on Assessment. I confirm that I have neither committed plagiarism in the completion of this work nor have I colluded with any other party in the preparation and production of this work. The work presented here is my own and in my own words except where I have clearly indicated and acknowledged that I have quoted or used figures from published or unpublished sources (including the web). I understand the consequences of engaging in plagiarism and collusion as described in the Code of Practice on Assessment (Appendix L).

11th February 2023

Contents

List of Figures

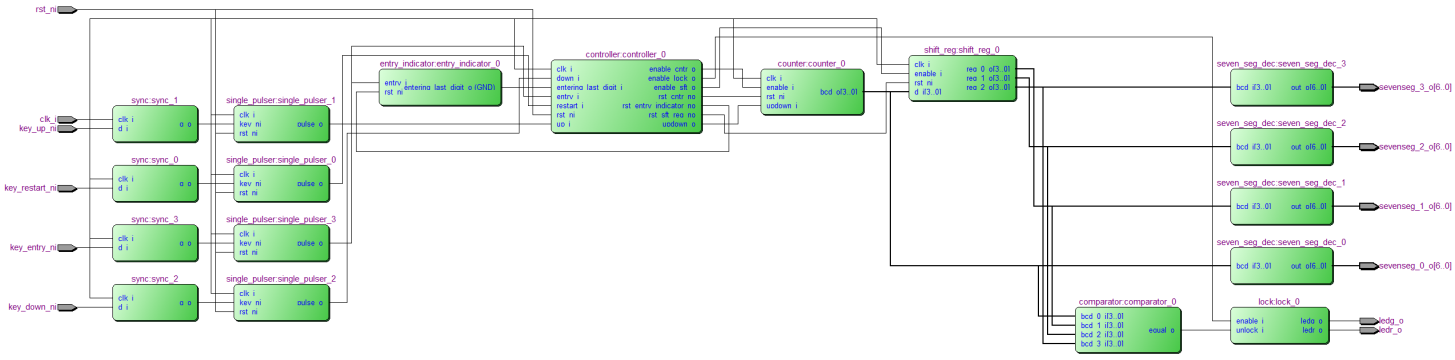
1 Introduction

This assignment is to design an electronic safe and implement it on a field-programmable gate array (FPGA) board by descripting its logic with a hardware description language (HDL) and synthesising the circuit with an electronic design automation (EDA) tool.

During the design and implementation process, students will learn two visualisation techniques and one methodology to help with design: algorithmic state machine (ASM) chart, block diagram, and top-down design, and will also become familiar with three development tools for FPGA implementation: Verilog 2005 which is a HDL whose descendant is SystemVerilog, Quartus II which is the last EDA tool supports Altera DE2 board, and a HDL simulator, ModelSim HDL simulator, which is a standalone software but can be used with other EDA tools includes Quartus II.

The remainder of the report is organised as follows: Section 2 demostrates the architecture of the electronic safe. Section 3 lists all aforementioned modules each with detailed description includes ASM chart, Verilog code, and simulation results with annotations.

2 Architecture



Bit small to read, best to do your own block diagram at the start.

Figure 1: Architectural block diagram for electronic safe. Please zoom in to make it clear.

Fig. 1, generated by the RTL viewer of Quartus II, is a block diagram visualisation of the architecture of the electronic safe. The left side gathers the inputs of the system and the right side gathers the outputs. The system accepts five input signals: up, down, entry, restart, and reset. All inputs go through the synchroniser and single pulser and then into the controller, except reset. The reset only connects the modules in the left half of the controller; the reset of the modules in the right half is controlled by the controller. The electronic safe uses four seven segment displays (SSD) to display the currently entered sequence of digits. The first SSD is connected directly to the output of the counter, the

remaining three are connected to the three register array of the shift register. The binary coded decimal (BCD) output of the counter is also connected to the data input of the shift register. When the shift register is triggered, the input of the counter is deposited into the first register array and the data originally in each register array is moved to the next register array. The counter and shift registerr in conjunction with the controller implements the functionality of sequential inputting a digital sequence. The outputs of the counter and shift registers are fed into the comparator. The comparator outputs a matching signal to the lock. If the lock receives the matching signal, it lights up a green light-emitting diode (LED) to indicate a correct digit sequence has been given, otherwise it lights up a red LED. If the lock is not enabled, none of the LEDs are lit. The entry indicator records the number of times the entry is pressed. When the user is ready to enter the last digit, it gives the controller a flag signal. The controller uses this signal to make a state transition, which preventing the shift register from moving an extra digit.

3 Modules

3.1 Synchroniser

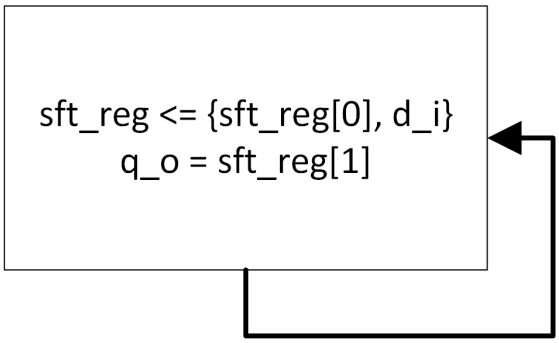


Figure 2: ASM chart of the synchroniser module.

```
1  module sync (
2     input clk_i,
3     input d_i,
4     output q_o
5  );
6
7  reg [1:0] sft_reg;
8
9  always@(posedge clk_i)
10    sft_reg <= {sft_reg[0], d_i};
11
12 assign q_o = sft_reg[1];
13
14 endmodule
```

Technically with 2 D types there are 4 states, this could have been drawn a

```
1  module sync_tb;
2
3  // Inputs
4  reg clk;
5  reg d_i;
6
7  // Outputs
8  wire q_o;
9
10 sync DUT (
11    .clk_i(clk),
12    .d_i(d_i),
13    .q_o(q_o)
14 );
15
16 // Create a 50Mhz clock
17 always #10 clk = !clk;  // every ten nanoseconds invert
18
19 initial begin
20    clk = 1'b0;
21    d_i = 1'b1;
22 end
23
```

```
24  initial begin
25      #20;
26
27      // Long press
28      #16;
29      d_i = 1'b0;
30      #43;
31      d_i = 1'b1;
32
33      // Brief contact
34      #46;
35      d_i = 1'b0;
36      #3;
37      d_i = 1'b1;
38      #20;
39
40      // Long press after brief contact
41      #16;
42      d_i = 1'b0;
43      #43;
44      d_i = 1'b1;
45      #100;
46
47  // Finish the Simulation
48      #100;
49      $finish;
50  end
51
52  endmodule
```

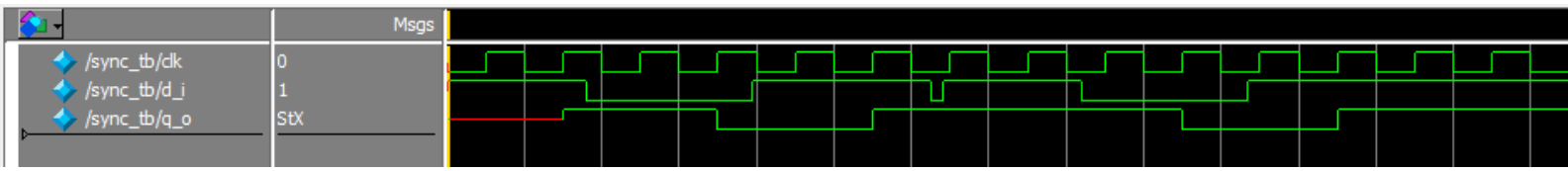


Figure 3: Testbench simulation of the synchroniser module.

Fig. 3 displays the simulation test results of the synchroniser module. The synchroniser delayed input signal by two clock cycle and aligned the edge of the input signal with the clock edges. Short signals may be ignored by the synchroniser.

3.2 Single pulser Three states for a single pulsar? Normally just down with 2 states.

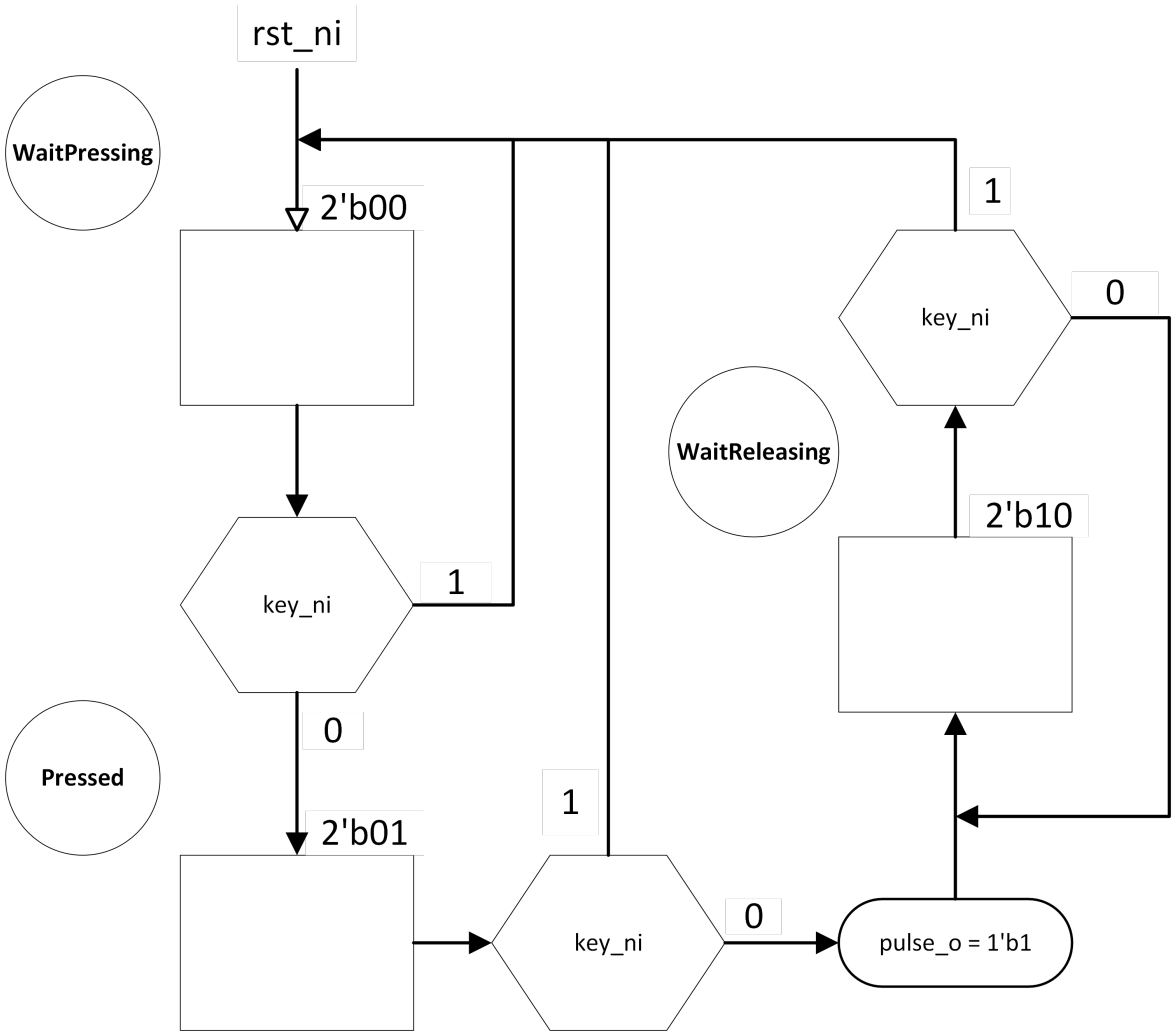


Figure 4: ASM chart of the single pulser module.

```
1  module single_pulser (
2     input clk_i,
3     input rst_ni,
4     input key_ni,
5     output reg pulse_o
6  );
7
8  reg [1:0] state_d, state_q;
9  parameter WaitPressing = 2'b00;
10 parameter Pressed = 2'b01;
11 parameter WaitReleasing = 2'b10;
12
13 always @(posedge clk_i, negedge rst_ni)
14    if (!rst_ni)
15       state_q <= WaitPressing;
16    else
17       state_q <= state_d;
18
```

```
always @(state_q, key_ni) begin
   pulse_o = 1'b0;
   state_d = state_q;

   case (state_d)
      WaitPressing:
         if (!key_ni)
            state_d = Pressed;

      Pressed:
         if (!key_ni) begin
            pulse_o = 1'b1;
            state_d = WaitReleasing;
         end else
            state_d = WaitPressing;

      WaitReleasing:
         if (key_ni)
            state_d = WaitPressing;
   endcase
end

endmodule
```

```
module single_pulser_tb;

// Inputs
reg clk;
reg rst_n;
reg key_ni;

// Outputs
wire pulse_o;

single_pulser DUT (
   .clk_i(clk),
   .rst_ni(rst_n),
   .key_ni(key_ni),
   .pulse_o(pulse_o)
);

// Create a 50Mhz clock
always #10 clk = !clk;  // every ten nanoseconds invert

initial begin
   clk = 1'b0;
   rst_n = 1'b0;
   key_ni = 1'b1;
end

initial begin
   #20 rst_n = 1'b1;  // release reset

   // Long press
   #16;
   key_ni = 1'b0;
   #43;
```

```
34      key_ni = 1'b1;
35
36      // Brief contact
37      #26;
38      key_ni = 1'b0;
39      #3;
40      key_ni = 1'b1;
41      #20;
42
43      // Long press after brief contact
44      #16;
45      key_ni = 1'b0;
46      #43;
47      key_ni = 1'b1;
48
49  // Finish the Simulation
50      #100;
51      $finish;
52  end
53
54  endmodule
```

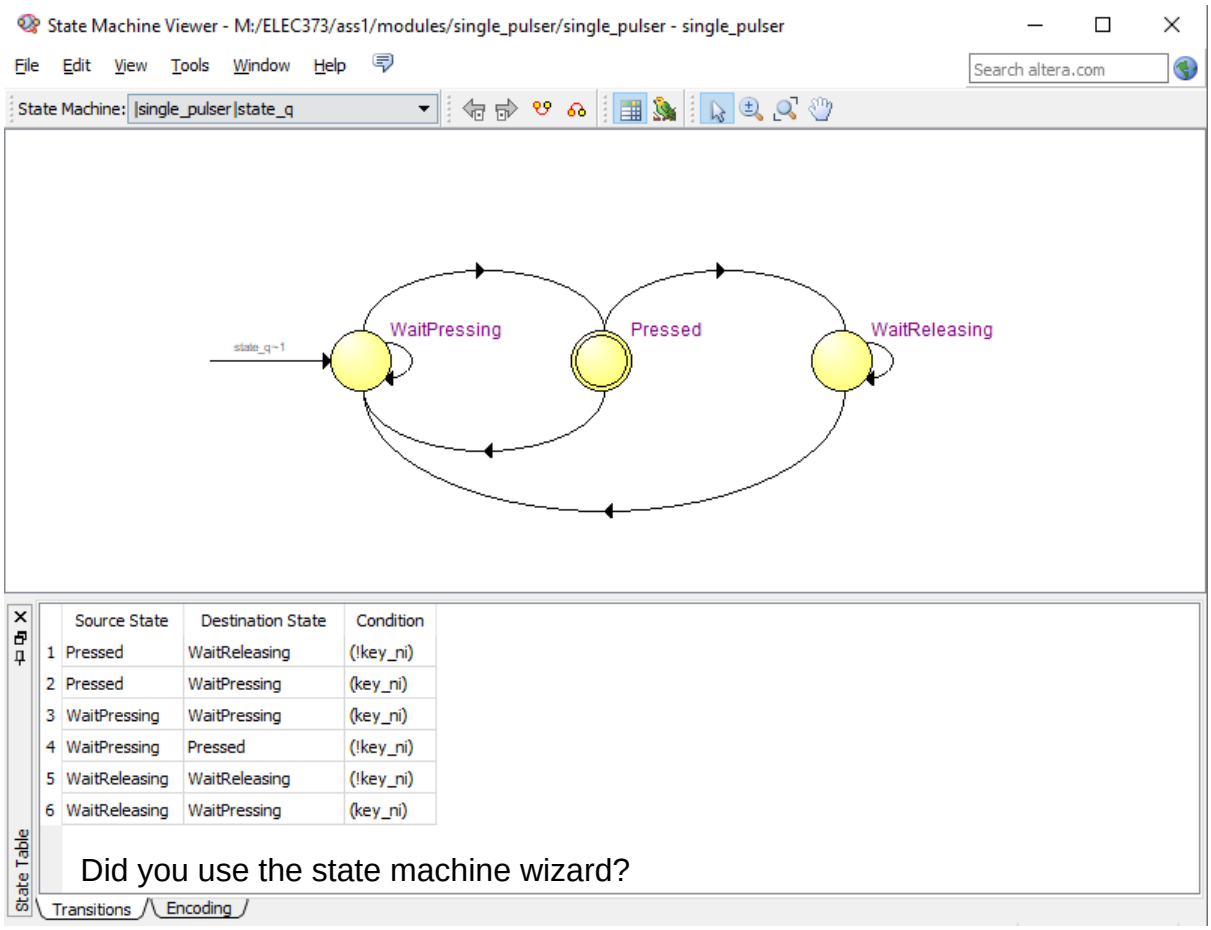


Figure 5: Quartus II state machine viewer shows the expected state machine of the single pulser after synthesis.

Fig. 6 illustrates the simulation test results of the single pulser module. Long lasting

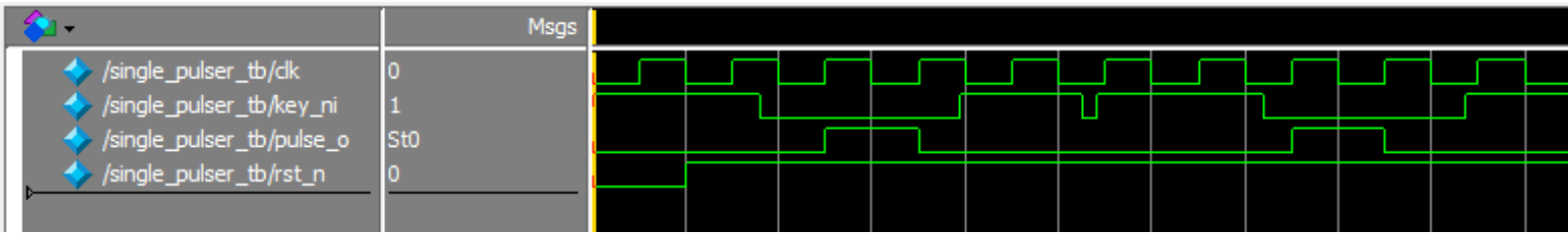


Figure 6: Testbench simulation of the single pulser module.

active-low signal was shortened into one pulse. Short signal was ignored. Short signal didn't affect the functionality of the signle pulser.

3.3 Controller

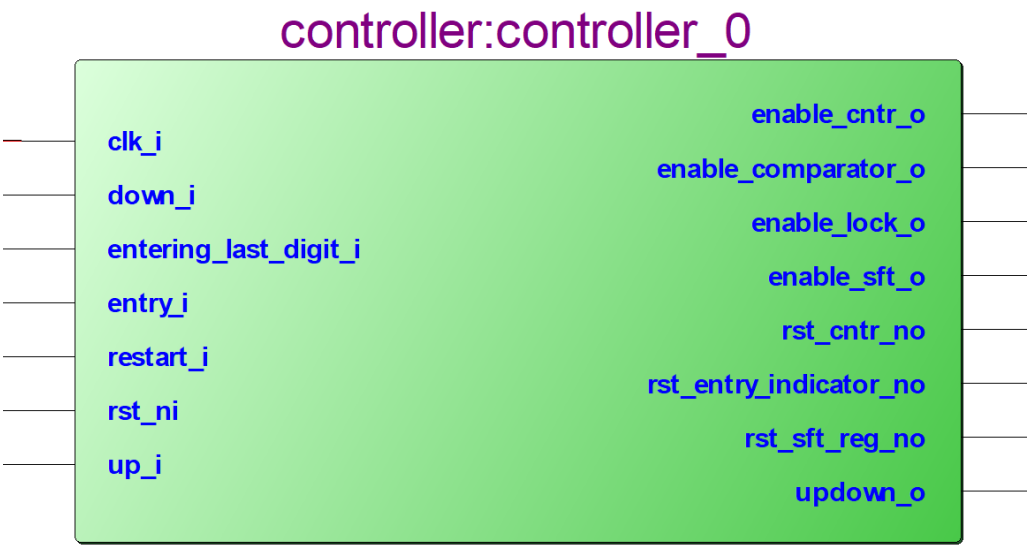
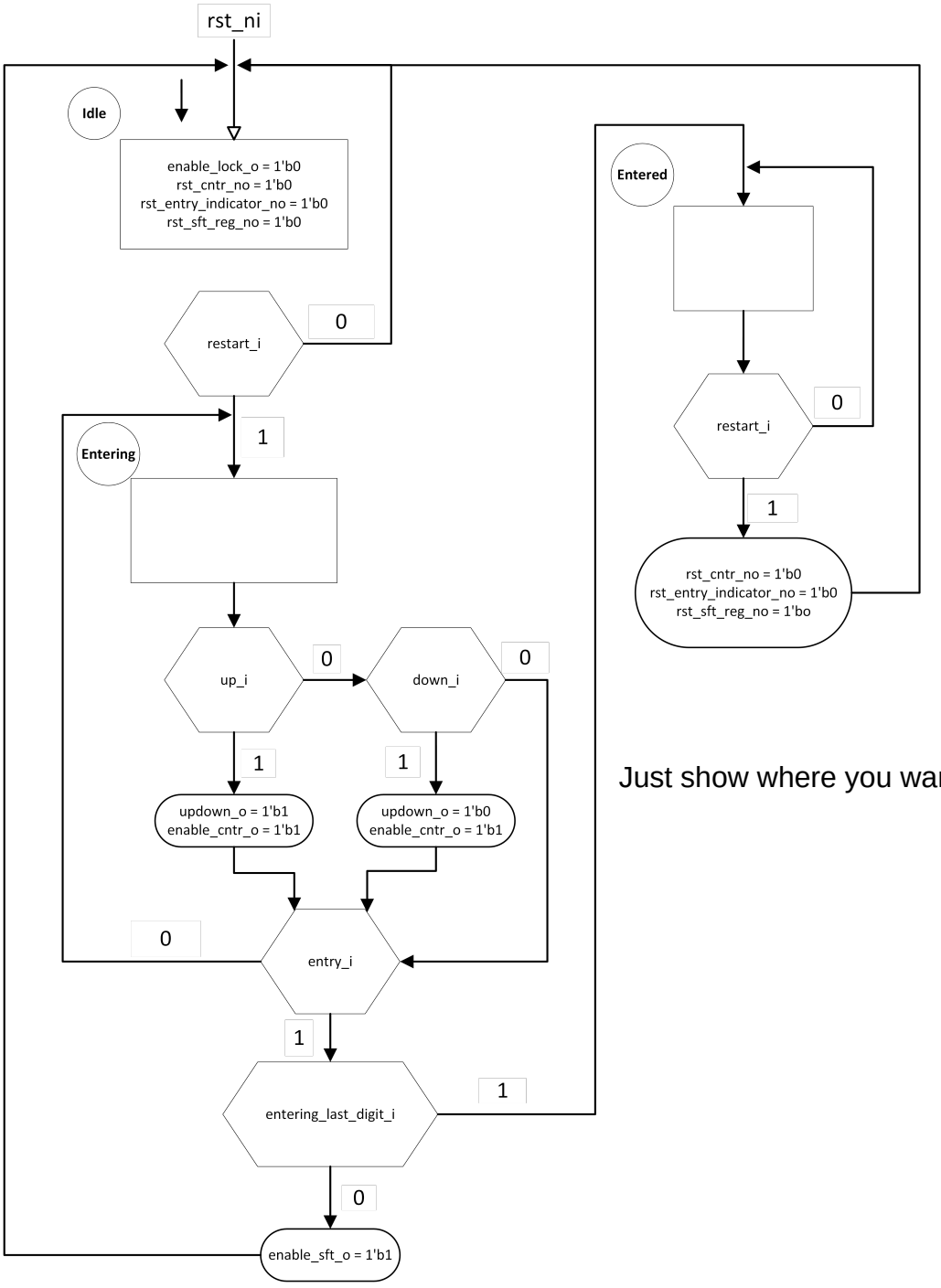


Figure 7: Block diagram of the controller module.

The controller is a finite state machine which takes key presses and a reset switch as inputs, and outputs enable signals and flag signals to the downstream modules. Fig. 7 is the block diagram of the controller, and Fig. 8 is the controller's ASM chart. The controller has three states: Idle, Entering, and Entered. In Idle state, the system will not respond to any input except restart signal. In Entering state, the controller will output signals and flags to manipulate the counter and shift register based on the inputs. In Entered state, the controller will disable counter and shift register and enable comparator and lock. When controller is reset, its state goes back to Idle. When controller is restarted, its state goes to Entering.

```
1  module controller (  // glue logic
2     input clk_i,
3     input rst_ni,
4     input up_i,
5     input down_i,
6     input entry_i,
7     input restart_i,
8     input entering_last_digit_i,
9     output reg enable_cntr_o,
10    output reg updown_o,
11    output reg enable_sft_o,
12    output reg enable_lock_o,
13    output reg rst_cntr_no,
14    output reg rst_entry_indicator_no,
15    output reg rst_sft_reg_no
16 );
17
18 reg [1:0] state_d, state_q;
19 parameter Idle = 2'b00;
20 parameter Entering = 2'b01;
21 parameter Entered  = 2'b10;
22
```



Just show where you want outputs True, th

Figure 8: ASM chart of the controller module.

```
23  always @(posedge clk_i, negedge rst_ni)
24     if (~rst_ni)
25        state_q <= Idle;
26     else
27        state_q <= state_d;
28
29  always @(state_q, up_i, down_i, entry_i, restart_i) begin
30     state_d = state_q;
31     enable_cntr_o = 1'b0;
32     updown_o = 1'b1;
33     enable_sft_o = 1'b0;
34     enable_lock_o = 1'b1;
35     rst_cntr_no = 1'b1;
36     rst_entry_indicator_no = 1'b1;
37     rst_sft_reg_no = 1'b1;
38
39     case (state_d)
40        Idle: begin
41           enable_lock_o = 1'b0;
42           rst_cntr_no = 1'b0;
43           rst_entry_indicator_no = 1'b0;
44           rst_sft_reg_no = 1'b0;
45           if (restart_i)
46              state_d = Entering;
47        end
48
49        Entering: begin
50           if (up_i) begin
51              updown_o = 1'b1;
52              enable_cntr_o = 1'b1;
53           end else if (down_i) begin
54              updown_o = 1'b0;
55              enable_cntr_o = 1'b1;
56           end
57
58           if (entry_i)
59              if (entering_last_digit_i)
60                 state_d = Entered;
61              else
62                 enable_sft_o = 1'b1;
63        end
64
65        Entered:
66           if (restart_i) begin
67              state_d = Entering;
68              rst_cntr_no = 1'b0;
69              rst_entry_indicator_no = 1'b0;
70              rst_sft_reg_no = 1'b0;
71           end
72     endcase
73  end
74
75  endmodule
```

```
1   module controller_tb;
2
3   // Inputs
```

```
4  reg clk;
5  reg rst_n;
6  reg up_i;
7  reg down_i;
8  reg entry_i;
9  reg restart_i;
10 reg entering_last_digit_i;
11
12 // Outputs
13 wire enable_cntr_o;
14 wire updown_o;
15 wire enable_sft_o;
16 wire enable_lock_o;
17 wire rst_cntr_no;
18 wire rst_entry_indicator_no;
19 wire rst_sft_reg_no;
20
21 controller DUT (
22    .clk_i(clk),
23    .rst_ni(rst_n),
24    .up_i(up_i),
25    .down_i(down_i),
26    .entry_i(entry_i),
27    .restart_i(restart_i),
28    .entering_last_digit_i(entering_last_digit_i),
29    .enable_cntr_o(enable_cntr_o),
30    .updown_o(updown_o),
31    .enable_sft_o(enable_sft_o),
32    .enable_lock_o(enable_lock_o),
33    .rst_cntr_no(rst_cntr_no),
34    .rst_entry_indicator_no(rst_entry_indicator_no),
35    .rst_sft_reg_no(rst_sft_reg_no)
36 );
37
38 // Create a 50Mhz clock
39 always #10 clk = !clk;  // every ten nanoseconds invert
40
41 initial begin
42    clk = 1'b0;  // at time 0
43    rst_n = 1'b0;  // reset is active
44    up_i = 1'b0;
45    down_i = 1'b0;
46    entry_i = 1'b0;
47    restart_i = 1'b0;
48    entering_last_digit_i = 1'b0;
49 end
50
51 initial begin
52    #20 rst_n = 1'b1;  // release reset
53
54 // State: Idle
55
56    // Move to State Entering
57    @(posedge clk);
58    restart_i = 1'b1;
59    @(posedge clk);
60    restart_i = 1'b0;
61
```

```
62  // State: Entering
63
64      @(posedge clk);
65      up_i = 1'b1;
66      @(posedge clk);
67      up_i = 1'b0;
68
69      @(posedge clk);
70      down_i = 1'b1;
71      @(posedge clk);
72      down_i = 1'b0;
73
74      @(posedge clk);
75      entry_i = 1'b1;
76      @(posedge clk);
77      entry_i = 1'b0;
78
79      // Move to State Entered
80      entering_last_digit_i = 1'b1;
81      @(posedge clk);
82      entry_i = 1'b1;
83      @(posedge clk);
84      entry_i = 1'b0;
85
86  // State: Entered
87
88      @(posedge clk);
89      up_i = 1'b1;
90      @(posedge clk);
91      up_i = 1'b0;
92
93      @(posedge clk);
94      down_i = 1'b1;
95      @(posedge clk);
96      down_i = 1'b0;
97
98      @(posedge clk);
99      entry_i = 1'b1;
100     @(posedge clk);
101     entry_i = 1'b0;
102
103     // Move to State Entering
104     @(posedge clk);
105     restart_i = 1'b1;
106     @(posedge clk);
107     restart_i = 1'b0;
108     entering_last_digit_i = 1'b0;
109
110 // State: Entering
111
112     @(posedge clk);
113     up_i = 1'b1;
114     @(posedge clk);
115     up_i = 1'b0;
116
117     @(posedge clk);
118     down_i = 1'b1;
119     @(posedge clk);
```

```
120     down_i = 1'b0;
121
122     @(posedge clk);
123     entry_i = 1'b1;
124     @(posedge clk);
125     entry_i = 1'b0;
126
127 // Reset
128
129     #20 rst_n = 1'b0;
130
131 // Finish the Simulation
132     #100;
133     $finish;
134 end
135
136 endmodule
```

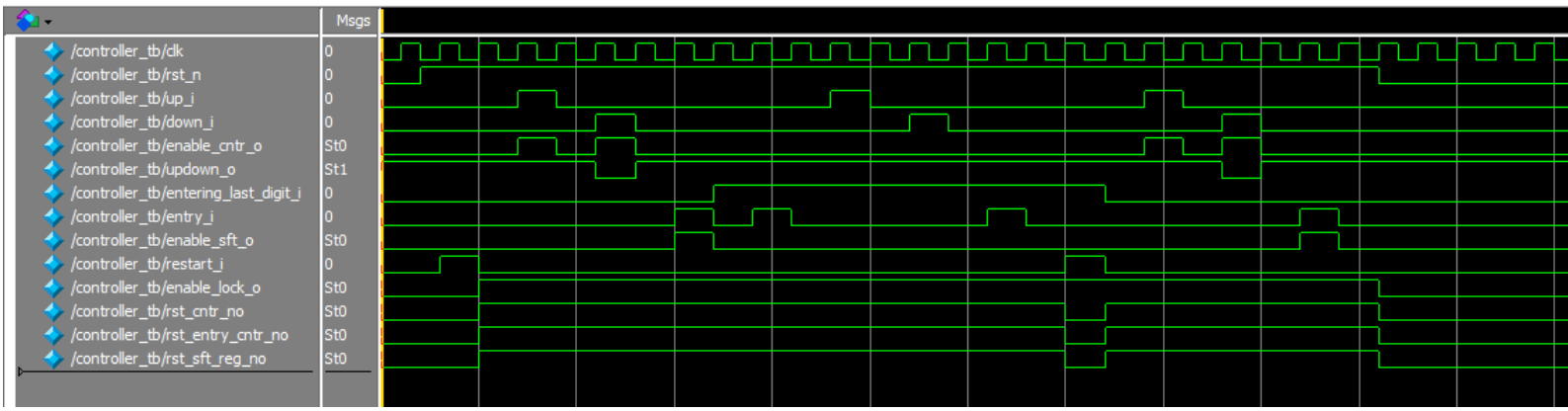


Figure 9: Testbench simulation of the controller module.

Fig. 9 shows the simulation test results of the counter module. After reset was released, the controller was in the Idle state. Pressing the restart key under Idle state relased the reset of the counter, entry indicator, and shift register, enabled the lock, and entered the Entering state. In Entering state, up and down pulse triggered enable counter signal. The down pulse also put the updown signal in low voltage level for one clock cycle. The entry pulse enabled the shift register so that the digits stored in it were shifted once. Inputting entry when flag signal, entering last digit, was raised brought the controller into the Entered state. In Entered state, up, down, and entry had no effect, except the restart reset counter, entry indicator, and shift register and brought the controller into the Entering state.

3.4 Counter

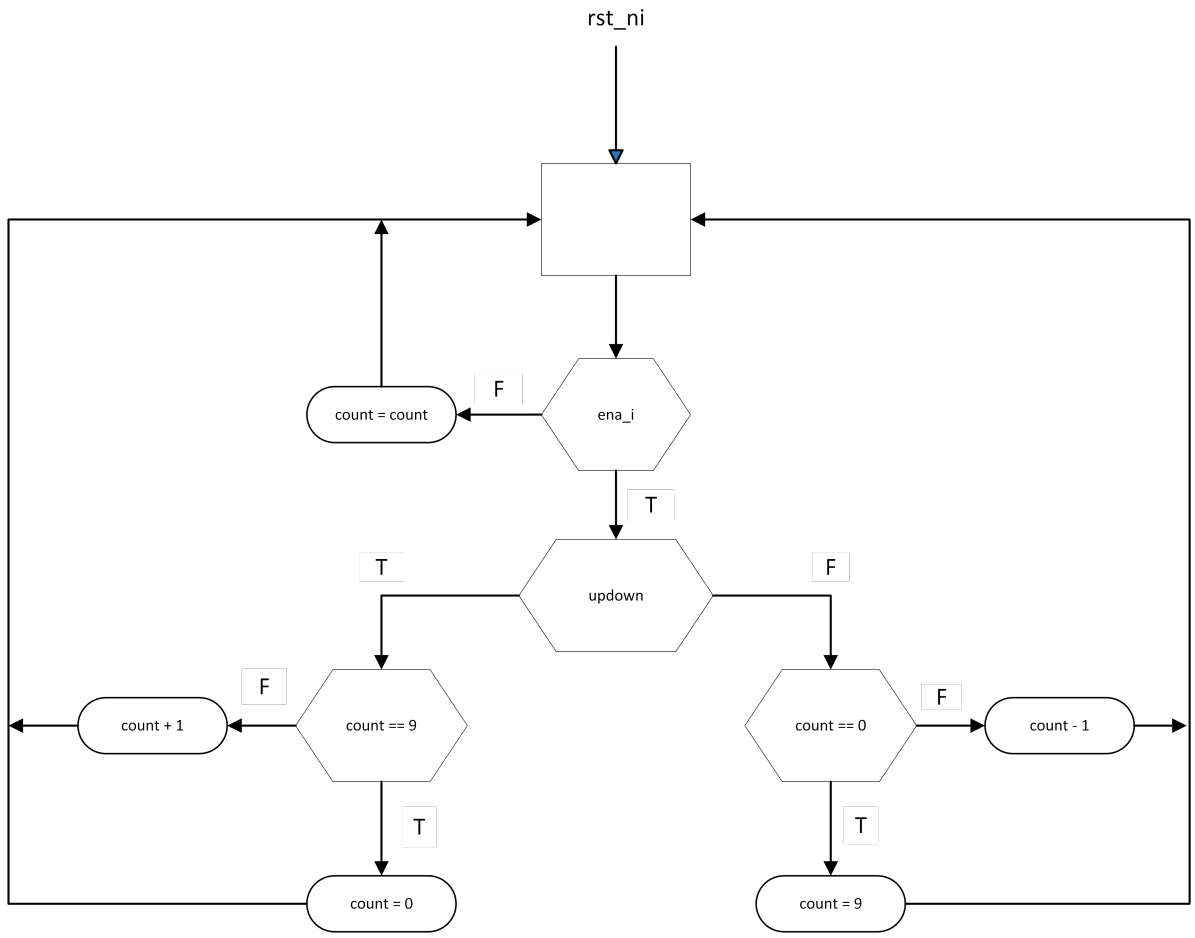


Figure 10: ASM chart of the counter module.

```
1  module counter (
2     input clk_i,
3     input rst_ni,
4     input ena_i,
5     input updown_i,
6     output reg [3:0] cnt_o
7  );
8
9  always @(posedge clk_i, negedge rst_ni)
10    if (!rst_ni)
11       cnt_o <= 4'b0;
12    else if (!ena_i)
13       cnt_o <= cnt_o;
14    else
15       if (updown_i)
16          if (cnt_o == 4'd9)
17             cnt_o <= 4'b0;
18          else
19             cnt_o <= cnt_o + 1'b1;
20       else
21          if (cnt_o == 4'd0)
```

```
            cnt_o <= 4'd9;
         else
            cnt_o <= cnt_o - 1'b1;

endmodule
```

```
module counter_tb;

// Inputs
reg clk;
reg rst_n;
reg ena_i;
reg updown_i;

// Outputs
wire [3:0] cnt_o;

counter DUT (
   .clk_i(clk),
   .rst_ni(rst_n),
   .ena_i(ena_i),
   .updown_i(updown_i),
   .cnt_o(cnt_o)
);

// Create a 50Mhz clock
always #10 clk = !clk;  // every ten nanoseconds invert

initial begin
   clk = 1'b0;
   rst_n = 1'b0;
   ena_i = 1'b0;  // disabled
   updown_i = 1'b1;  // count up
end

initial begin
   #20 rst_n = 1'b1;  // release reset

   // Test 0 -> 9
   @(posedge clk);
   ena_i = 1'b1;
   @(posedge clk);
   ena_i = 1'b0;
   updown_i = 1'b0;
   @(posedge clk);
   ena_i = 1'b1;
   @(posedge clk);
   ena_i = 1'b0;
   @(posedge clk);
   ena_i = 1'b1;
   @(posedge clk);
   ena_i = 1'b0;

   // Test 9 -> 0
   updown_i = 1'b1;
   @(posedge clk);
   ena_i = 1'b1;
```

```
52      @(posedge clk);
53      ena_i = 1'b0;
54      @(posedge clk);
55      ena_i = 1'b1;
56      @(posedge clk);
57      ena_i = 1'b0;
58
59      // Test reset
60      @(posedge clk);
61      ena_i = 1'b1;
62      @(posedge clk);
63      ena_i = 1'b0;
64      @(posedge clk);
65      ena_i = 1'b1;
66      @(posedge clk);
67      ena_i = 1'b0;
68      @(posedge clk);
69      rst_n = 1'b0;
70      @(posedge clk);
71      rst_n = 1'b1;
72
73  // Finish the Simulation
74      #100;
75      $finish;
76  end
77
78  endmodule
```

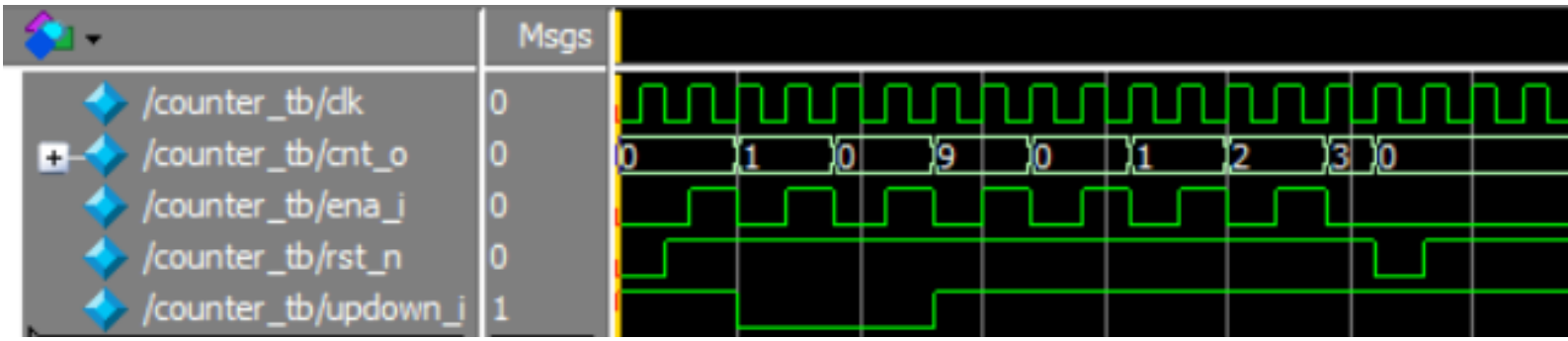


Figure 11: Testbench simulation of the counter module.

Fig. 11 shows the simulation test results of the counter module. The counter counts up when updown is high potential and counts down when it is low potential. The enable signal triggers the counter to count once.

The first test case was to count from nine to zero. Nine set to the biggest digit this counter can count to. The second test case was to count beyond nine, which returned the count to zero. Next, Reset singal reset the count to zero.

Where is entry_cnt initialised?

3.5 Entry indicator

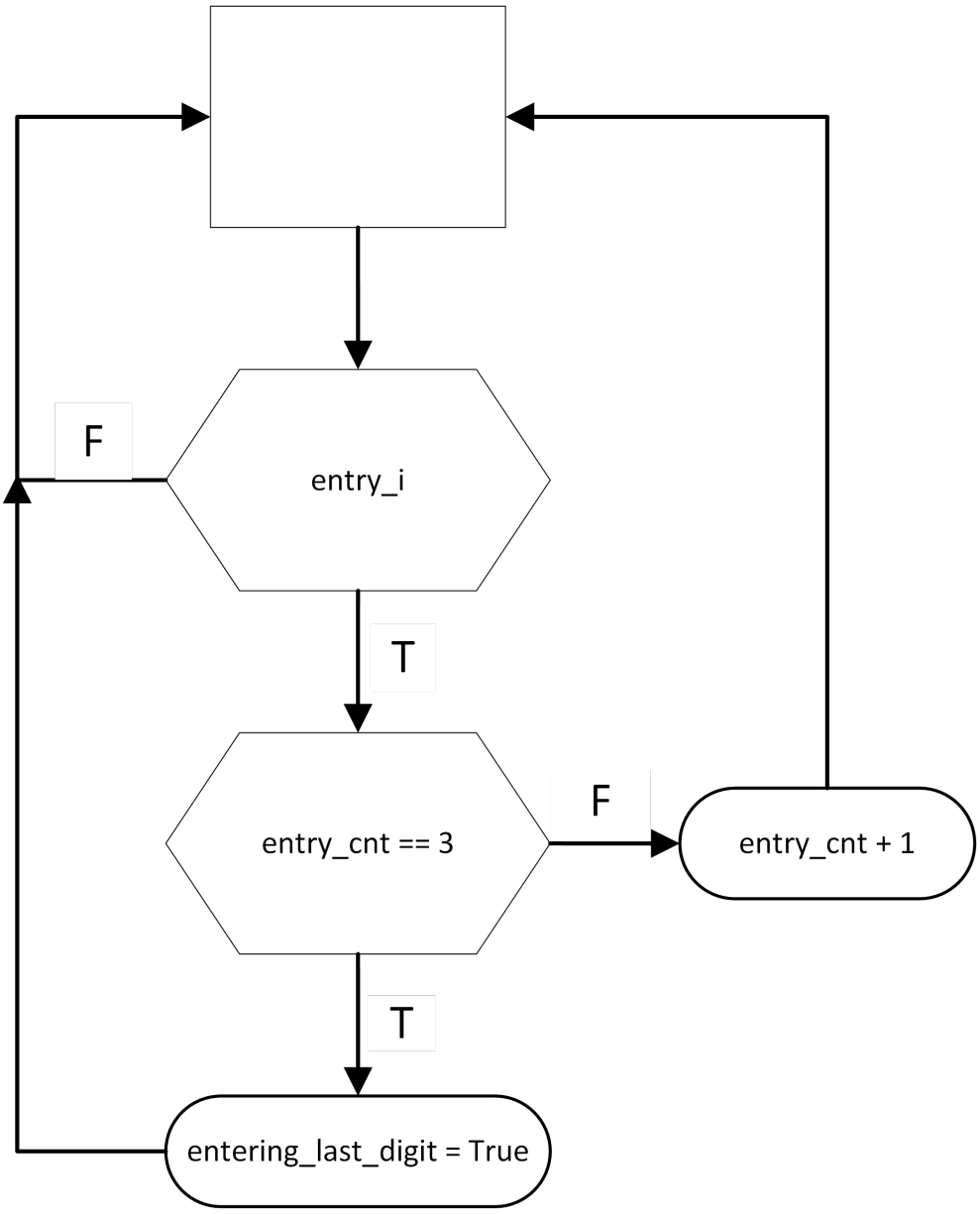


Figure 12: ASM chart of the entry indicator module.

```
1  module entry_indicator (
2     input clk_i,
3     input rst_ni,
4     input entry_i,
5     output entering_last_digit_o
6  );
7
8  reg [2:0] state_d, state_q;
9  parameter EnteringLastDigit = 3'd3;
10
11 assign entering_last_digit_o = state_q == EnteringLastDigit;
12
```

Not clear what enetering_last_digit is? Is it a Boolean or a co

```
always @(posedge clk_i, negedge rst_ni)
   if (~rst_ni)
      state_q <= 3'd0;
   else
      state_q <= state_d;

always @(state_q, entry_i) begin
   state_d = state_q;  // default assignment next state is present state
   if (entry_i)
      if (state_d == EnteringLastDigit)
         state_d = EnteringLastDigit;
      else
         state_d = state_d + 1'b1;
end
                    best to use state_q?

endmodule
```

```
module entry_indicator_tb;

// Inputs
reg clk;
reg rst_n;
reg entry_i;

// Outputs
wire entering_last_digit_o;

entry_indicator DUT (
   .clk_i(clk),
   .rst_ni(rst_n),
   .entry_i(entry_i),
   .entering_last_digit_o(entering_last_digit_o)
);

// Create a 50Mhz clock
always #10 clk = !clk;  // every ten nanoseconds invert

initial begin
   clk = 1'b0;
   rst_n = 1'b0;
   entry_i = 1'b0;
end

initial begin
   #20 rst_n = 1'b1;  // release reset

   repeat (6) begin
      @(posedge clk);
      entry_i = 1'b1;
      @(posedge clk);
      entry_i = 1'b0;
   end

   #40;

   $finish;
end
```

```
41
42 endmodule
```

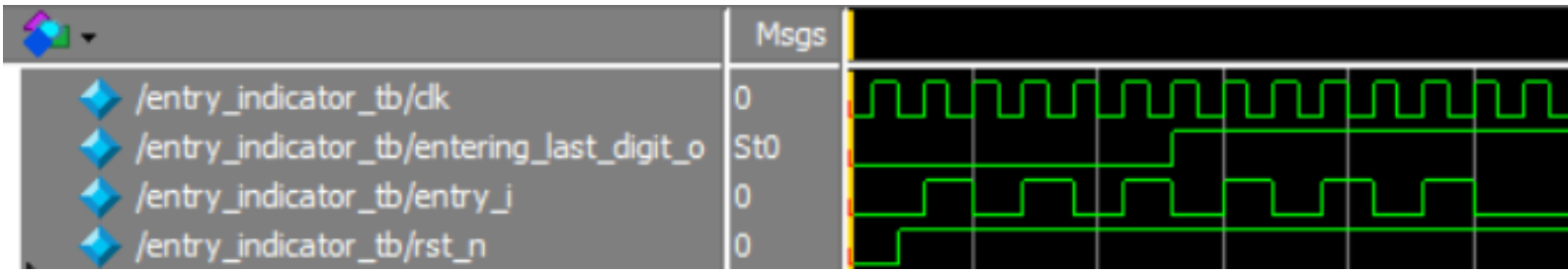


Figure 13: Testbench simulation of the entry indicator module.

Fig. 13 shows that the indicator raises entering-last-digit flag signal after entering the third digit (forth digit is the last digit as per the assignment brief).

Does it get reset?

3.6 Shift register

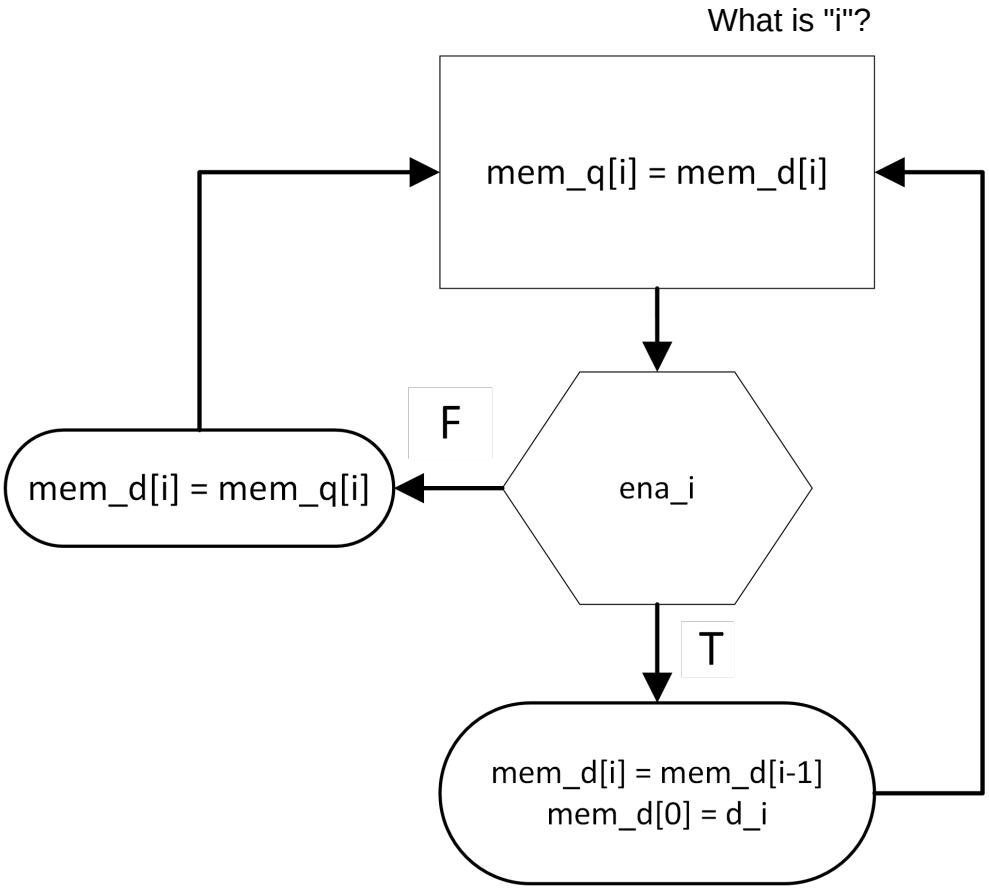


Figure 14: ASM chart of the shift register module.

```
1  module shift_reg #(
2     parameter DataWidth = 4,
3     parameter Depth = 3  // FIXME: Depth is actullay hard-coded
4  )(
5     input clk_i,
6     input rst_ni,
7     input enable_i,
8     input  [DataWidth-1:0] d_i,
9     output [DataWidth-1:0] reg_0_o,
10    output [DataWidth-1:0] reg_1_o,
11    output [DataWidth-1:0] reg_2_o
12 );
13
14 reg [DataWidth-1:0] mem_d [0:Depth-1];
15 reg [DataWidth-1:0] mem_q [0:Depth-1];
16
17 // TODO: more flexible
18 assign reg_0_o = mem_q[0];
19 assign reg_1_o = mem_q[1];
20 assign reg_2_o = mem_q[2];
21
22 integer i;
23
24 always @(posedge clk_i, negedge rst_ni)
```

```
   if (!rst_ni)
      for(i = 0; i < Depth; i = i + 1)
         mem_q[i] <= 0;
      // For SystemVerilog, use array assignment pattern with the default keyword:
      // mem_q <= '{default: '0};
   else
      for(i = 0; i < Depth; i = i + 1)
         mem_q[i] <= mem_d[i];

always @(enable_i, mem_q) begin

   // default assignment next state is present state
   for(i = 0; i < Depth; i = i + 1)
      mem_d[i] = mem_q[i];

   if (enable_i) begin
       for(i = Depth - 1; i > 0; i = i - 1)
          mem_d[i] = mem_d[i-1];
       mem_d[0] = d_i;
   end

end

endmodule
```

```
module shift_reg_tb;

parameter DataWidth = 5;

// Inputs
reg clk;
reg rst_n;
reg enable_i;
reg [DataWidth-1:0] d_i;

// Outputs
wire [DataWidth-1:0] reg_0_o;
wire [DataWidth-1:0] reg_1_o;
wire [DataWidth-1:0] reg_2_o;

shift_reg #(
   .DataWidth(DataWidth)
) DUT (
   .clk_i(clk),
   .rst_ni(rst_n),
   .enable_i(enable_i),
   .d_i(d_i),
   .reg_0_o(reg_0_o),
   .reg_1_o(reg_1_o),
   .reg_2_o(reg_2_o)
);

// Create a 50Mhz clock
always #10 clk = !clk;  // every ten nanoseconds invert

initial begin
   clk = 1'b0;
```

```
33      rst_n = 1'b0;
34      enable_i = 1'b0;
35      d_i = 5'd4;
36  end
37
38  initial begin
39      #20 rst_n = 1'b1;  // release reset
40
41      // Shift a same value multiple times
42      repeat (4) begin
43          @(posedge clk);
44          enable_i = 1'b1;
45          @(posedge clk);
46          enable_i = 1'b0;
47      end
48
49      // Reset
50      @(negedge clk);
51      rst_n = 1'b0;
52      @(negedge clk);
53      rst_n = 1'b1;
54
55      // Shift
56      d_i = 5'd7;
57      @(posedge clk);
58      enable_i = 1'b1;
59      @(posedge clk);
60      enable_i = 1'b0;
61
62  // Finish the Simulation
63      #100;
64      $finish;
65  end
66
67  endmodule
```

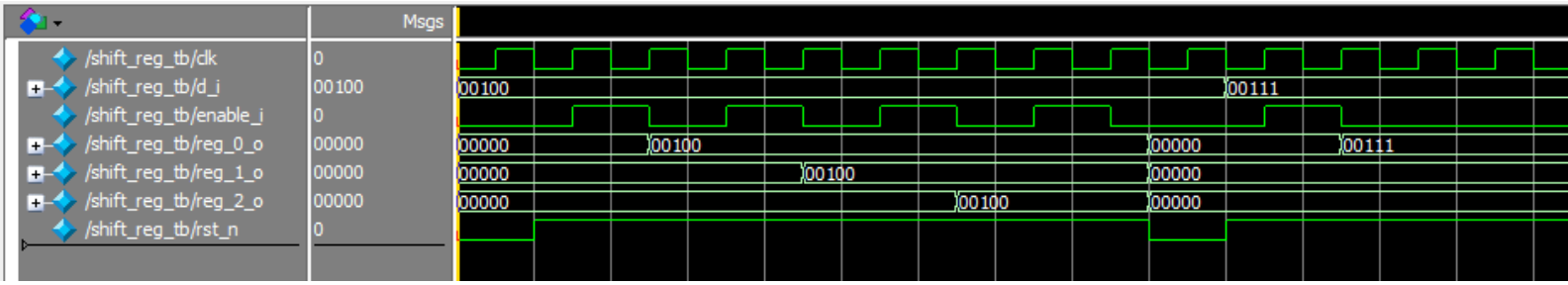


Figure 15: Testbench simulation of the shift register module.

Fig. 15 illustrates the behaviour of the shift register. The shift register put the new data in its first register array and shifted other stored data to their next adjacent register array.

3.7 Comparator

Show the comparison in the ASM with a test condition.

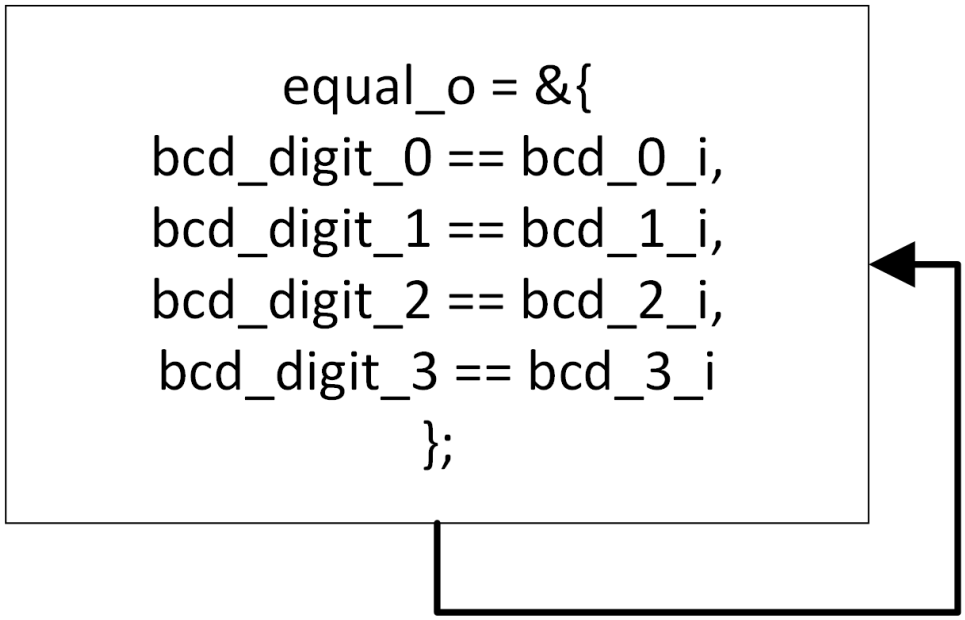


Figure 16: ASM chart of the comparator module.

```
module comparator #(
   parameter bcd_digit_0 = 4'd2,
   parameter bcd_digit_1 = 4'd8,
   parameter bcd_digit_2 = 4'd0,
   parameter bcd_digit_3 = 4'd1
)(
   input [3:0] bcd_0_i,
   input [3:0] bcd_1_i,
   input [3:0] bcd_2_i,
   input [3:0] bcd_3_i,
   output equal_o
);

assign equal_o = &{
   bcd_digit_0 == bcd_0_i,
   bcd_digit_1 == bcd_1_i,
   bcd_digit_2 == bcd_2_i,
   bcd_digit_3 == bcd_3_i
};

endmodule
```

```
module comparator_tb;

reg [3:0] bcd_0_i;
reg [3:0] bcd_1_i;
reg [3:0] bcd_2_i;
reg [3:0] bcd_3_i;

```

```
8   wire equal_o;
9
10  comparator DUT (
11      .bcd_0_i(bcd_0_i),
12      .bcd_1_i(bcd_1_i),
13      .bcd_2_i(bcd_2_i),
14      .bcd_3_i(bcd_3_i),
15      .equal_o(equal_o)
16  );
17
18  initial begin
19      bcd_0_i = 4'd0;
20      bcd_1_i = 4'd0;
21      bcd_2_i = 4'd0;
22      bcd_3_i = 4'd0;
23  end
24
25  initial begin
26      #100;
27
28      bcd_0_i = 4'd2;
29      bcd_1_i = 4'd8;
30      bcd_2_i = 4'd0;
31      bcd_3_i = 4'd1;
32      #100;
33
34      bcd_0_i = 4'd4;
35      bcd_1_i = 4'd4;
36      bcd_2_i = 4'd4;
37      bcd_3_i = 4'd4;
38      #100;
39
40  // Finish the Simulation
41      #100;
42      $finish;
43  end
44
45  endmodule
```

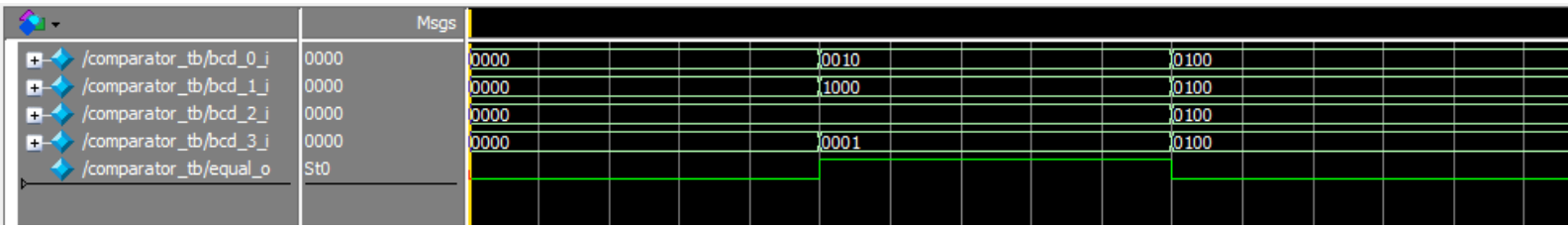


Figure 17: Testbench simulation of the comparator module.

Fig. 17 is the simulation result of the lock module. The module raised equal signal when inputs matched with the stored parameters.

Probably best drawn as a 2 state ASM with the states being locked and unlocked.

3.8 Lock

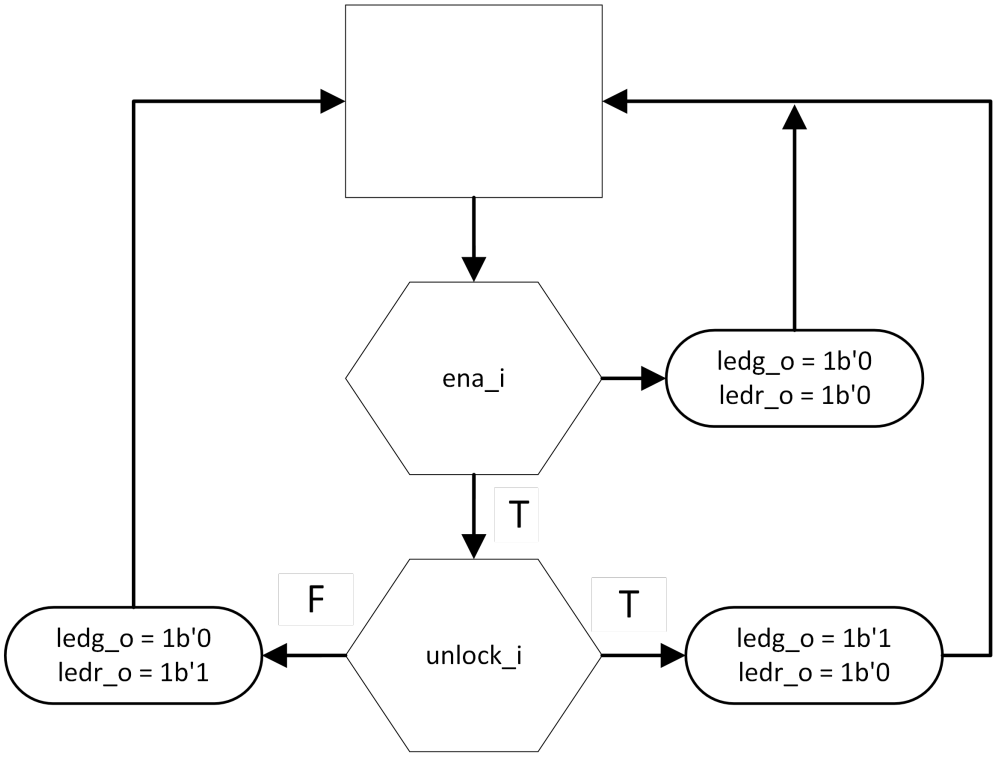


Figure 18: ASM chart of the lock module.

```
module lock (
   input enable_i,
   input unlock_i,
   output reg ledr_o,
   output reg ledg_o
);

always @(unlock_i) begin
   ledr_o = 1'b0;
   ledg_o = 1'b0;
   if (enable_i)
      if (unlock_i)
         ledg_o = 1'b1;
      else
         ledr_o = 1'b1;
end

endmodule
```

```
module lock_tb;

// Inputs
reg enable_i;
reg unlock_i;

// Outputs
```

```
8   wire ledr_o;
9   wire ledg_o;
10
11  lock DUT (
12      .enable_i(enable_i),
13      .unlock_i(unlock_i),
14      .ledr_o(ledr_o),
15      .ledg_o(ledg_o)
16  );
17
18  initial begin
19      enable_i = 1'b0;  // disabled
20      unlock_i = 1'b0;  // locked
21  end
22
23  initial begin
24      #20;
25
26      unlock_i = 1'b1;
27      #20;
28      unlock_i = 1'b0;
29      #20;
30
31      enable_i = 1'b1;  // enabled
32      unlock_i = 1'b1;
33      #20;
34      unlock_i = 1'b0;
35      #40;
36
37  // Finish the Simulation
38      #100;
39      $finish;
40  end
41
42  endmodule
```

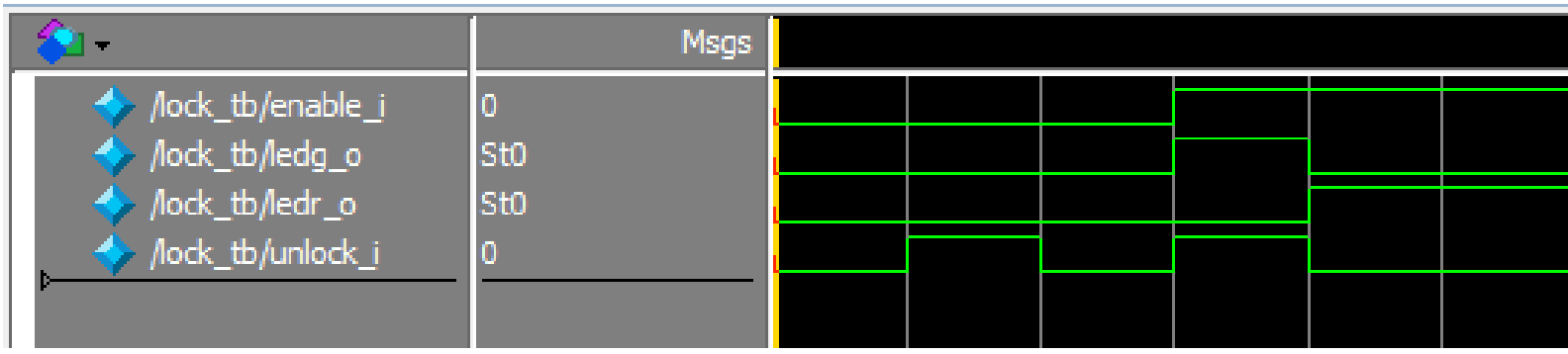


Figure 19: Testbench simulation of the lock module.

Fig. 19 is the simulation result of the lock module. The lock turned off both green and red LED when enable signal wa low, turned on the red LED and turned off the green LED when the enable signal is high and unlock signal is low, turned on the green LED and turned off the red LED when the enable signal is high and the unlock signal is high.

3.9 Seven segment decoder

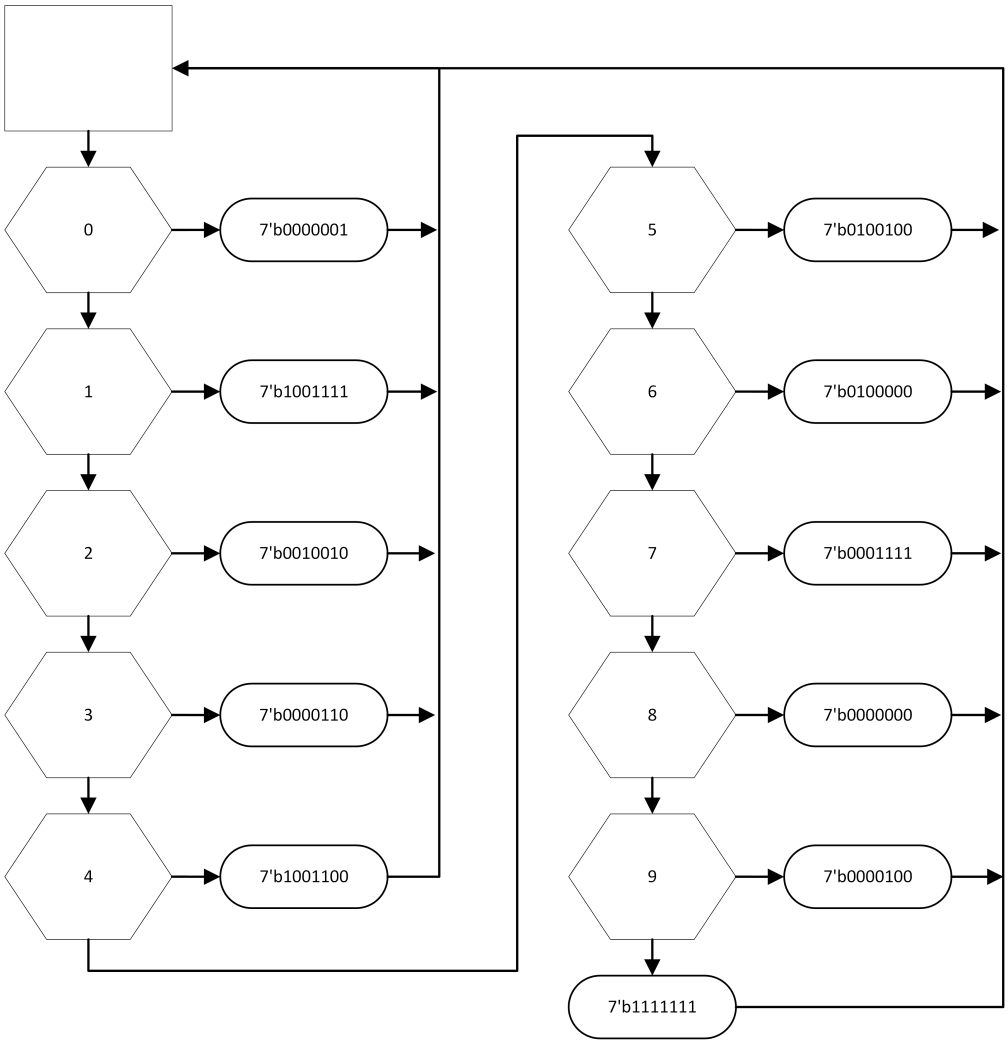


Figure 20: ASM chart of seven segment decoder module.

```
1  // decoder: convert coded input into conded output
2  /**
3  For Altera DE2,
4
5    -a-
6  f|   |b
7    -g-
8  e|   |c
9    -d-
10
11 displays 0123456789ABCDEF
12 */
13 module seven_seg_dec (
14         input [3:0] bcd_i,  // binary coded decimal input
15         output reg [6:0] out_o
16 );
17
18 always @(bcd_i)
```

```
19          case (bcd_i)
20                  //               abcdefg
21                  4'h0: out_o = 7'b0000001;
22                  4'h1: out_o = 7'b1001111;
23                  4'h2: out_o = 7'b0010010;
24                  4'h3: out_o = 7'b0000110;
25                  4'h4: out_o = 7'b1001100;
26                  4'h5: out_o = 7'b0100100;
27                  4'h6: out_o = 7'b0100000;
28                  4'h7: out_o = 7'b0001111;
29                  4'h8: out_o = 7'b0000000;
30                  4'h9: out_o = 7'b0000100;
31                  default: out_o = 7'b1111111;
32          endcase
33
34  endmodule
```

3.10 Full system

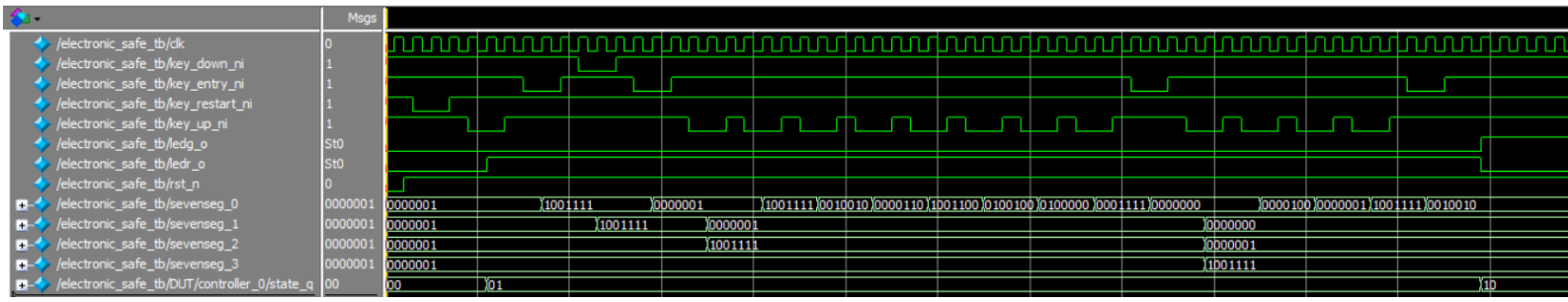


Figure 21: Full system simulation in ModelSim.

Fig. 21 is the full system simulation result in ModelSim. The system was started by released reset and pressed restart key, which was indicated by the controller's state transitived from 00 (Idle) to 01 (Entering) at the bottom line. The red led was turned on after the system started. By entering up, down, and entry keys, the user input digits were stored in the shift register. The seven segment decoders read digits from the shift register and displayed them. After user input every digits, the system went into 10 (Entered) state. In the Entered state, the user input digits were compared with the pin stored in the device. The user input digits matched with the pin, and the green led turned on and red led turned off. The system can be restarted by pressing the restart key.

3.11 Conclusion

In this assignment, a electronic safe system was designed with top-down methodology. Detailed module representations include block diagrams, ASM charts, verilog code, and simulated waveforms are presented. Modules were tested by exhausting paths of their ASM chart, and the simulation test results were explained. The full system was tested by the sepecified use case in both the board and the simulator, and the corresponding simulation test results were also explained.

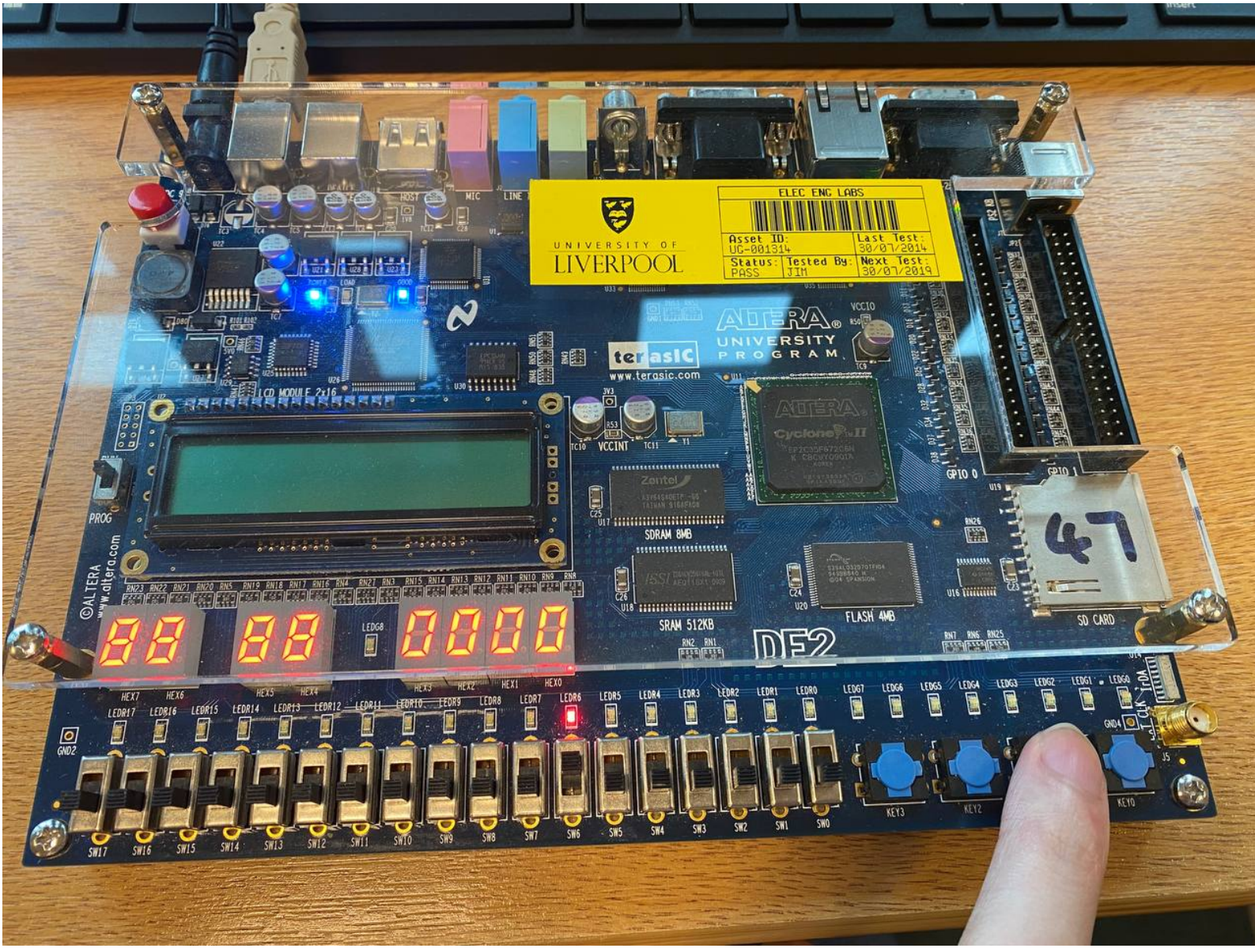


Figure 22: Reset switch was released and restart key was pressed.

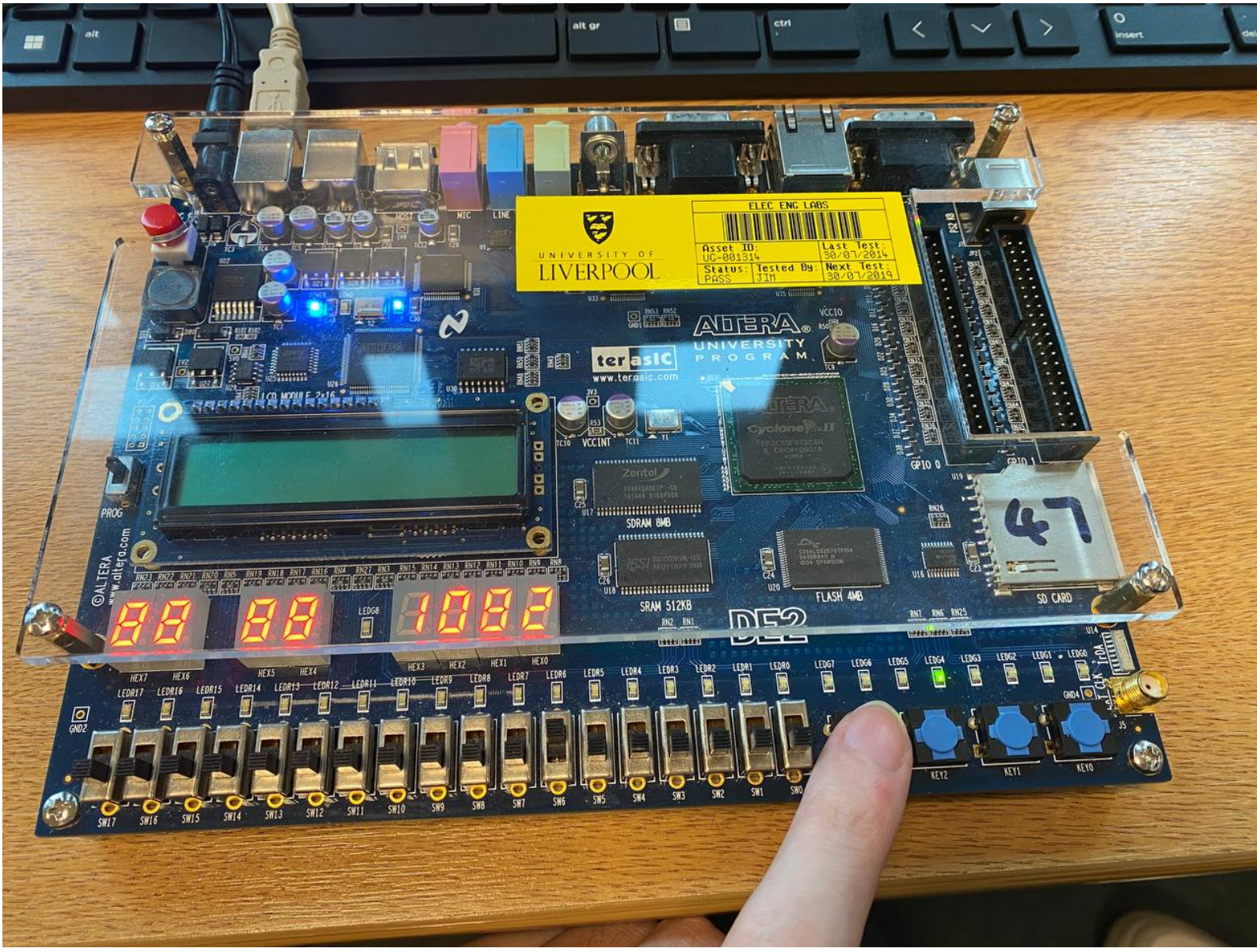


Figure 23: Pin was entered correctly.

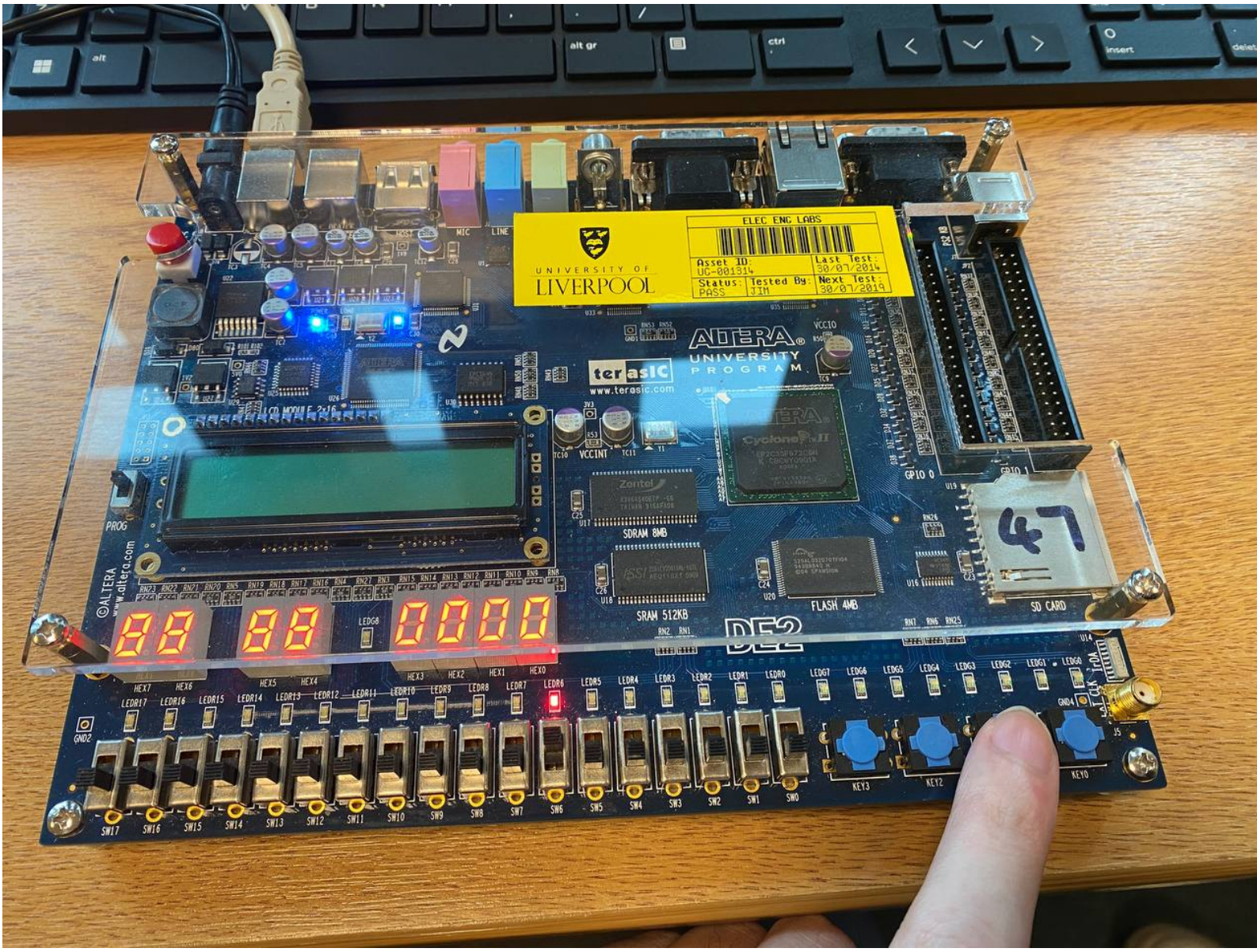


Figure 24: Restart key was pressed and the system was locked and went back to the entering state.