UNIVERSITY OF LIVERPOOL

ELEC373 ASSIGNMENT 1

Designing Electronic Safe

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own words except where I have clearly indicated and acknowledged that I have quoted or used figures from published or unpublished sources (including the web).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).

27th November 2022

Contents

List of Figures

1 Introduction

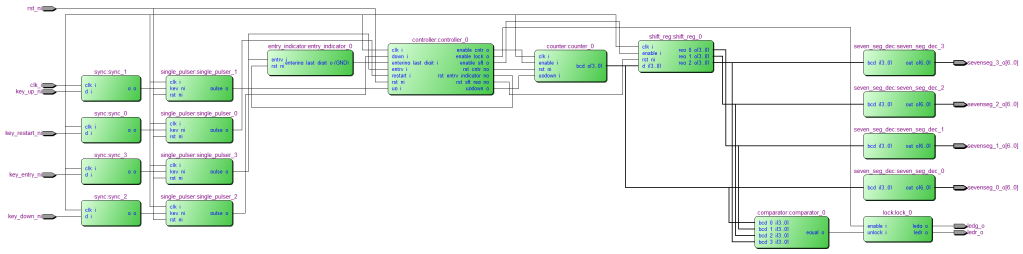
This assignment is to design an electronic safe and implement it on a field-programmable gate array (FPGA) board by descripting its logic with a hardware description language (HDL) and synthesising the circuit with an electronic design automation (EDA) tool.

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

Best to do your own block diagram before you start coding.



Zoomed in but still can't read any text!

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

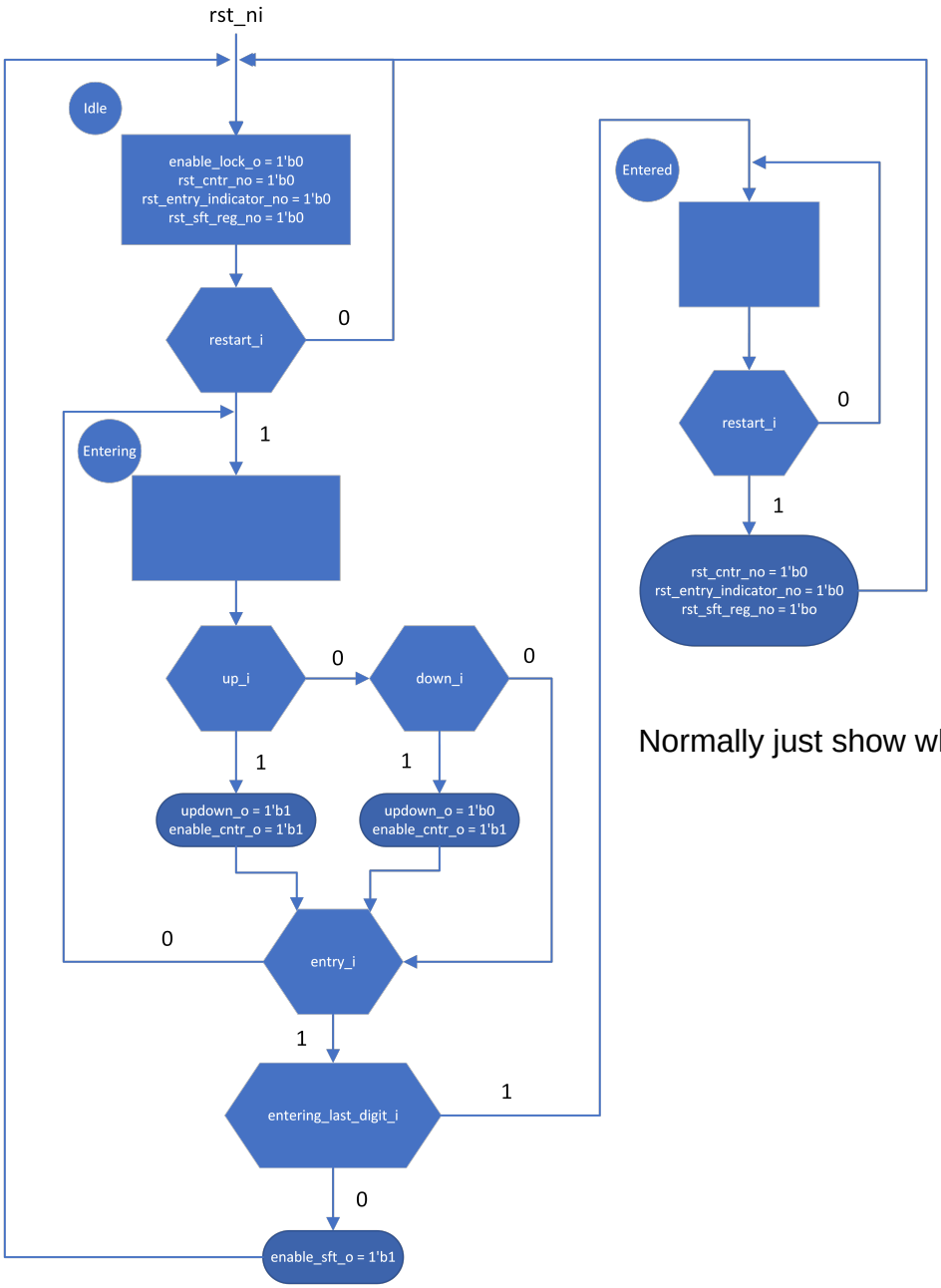
Fig. 1, generated by the RTL viewer of Quartus II, is a block diagram visualisation of the architecture of the electronic safe. The left side gathers the inputs of the system and the right side gathers the outputs. The system accepts five input signals: up, down, entry, restart, and reset. All inputs 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 Controller

Perhaps include a schematic of this block so I now what the inputs and



Normally just show where you want combin

Figure 2: ASM chart of the controller module.

```
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
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
```

```
      Entered:
         if (restart_i) begin
            state_d = Entering;
            rst_cntr_no = 1'b0;
            rst_entry_indicator_no = 1'b0;
            rst_sft_reg_no = 1'b0;
         end
   endcase
end

endmodule
```

```
module controller_tb;

// Inputs
reg clk;
reg rst_n;
reg up_i;
reg down_i;
reg entry_i;
reg restart_i;
reg entering_last_digit_i;

// Outputs
wire enable_cntr_o;
wire updown_o;
wire enable_sft_o;
wire enable_lock_o;
wire rst_cntr_no;
wire rst_entry_indicator_no;
wire rst_sft_reg_no;

controller DUT (
   .clk_i(clk),
   .rst_ni(rst_n),
   .up_i(up_i),
   .down_i(down_i),
   .entry_i(entry_i),
   .restart_i(restart_i),
   .entering_last_digit_i(entering_last_digit_i),
   .enable_cntr_o(enable_cntr_o),
   .updown_o(updown_o),
   .enable_sft_o(enable_sft_o),
   .enable_lock_o(enable_lock_o),
   .rst_cntr_no(rst_cntr_no),
   .rst_entry_indicator_no(rst_entry_indicator_no),
   .rst_sft_reg_no(rst_sft_reg_no)
);

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

initial begin
   clk = 1'b0;  // at time 0
   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
```

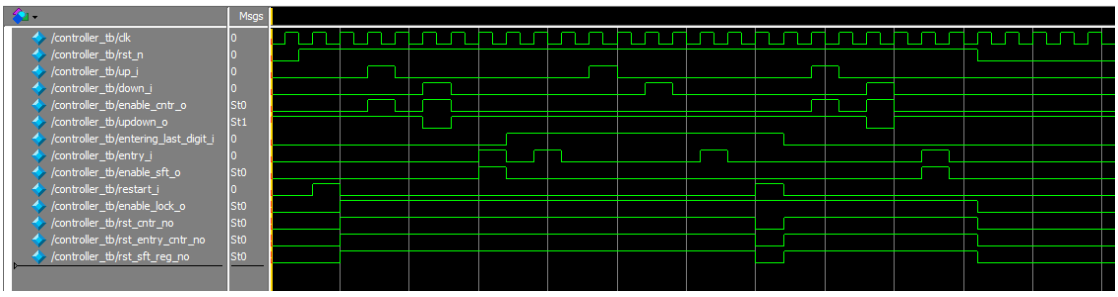


Figure 3: Testbench simulation of the controller module.

Fig. 3 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.

3.2 Single pulser

Single pulser is normally 2 states not 3. Did you synchronise bef

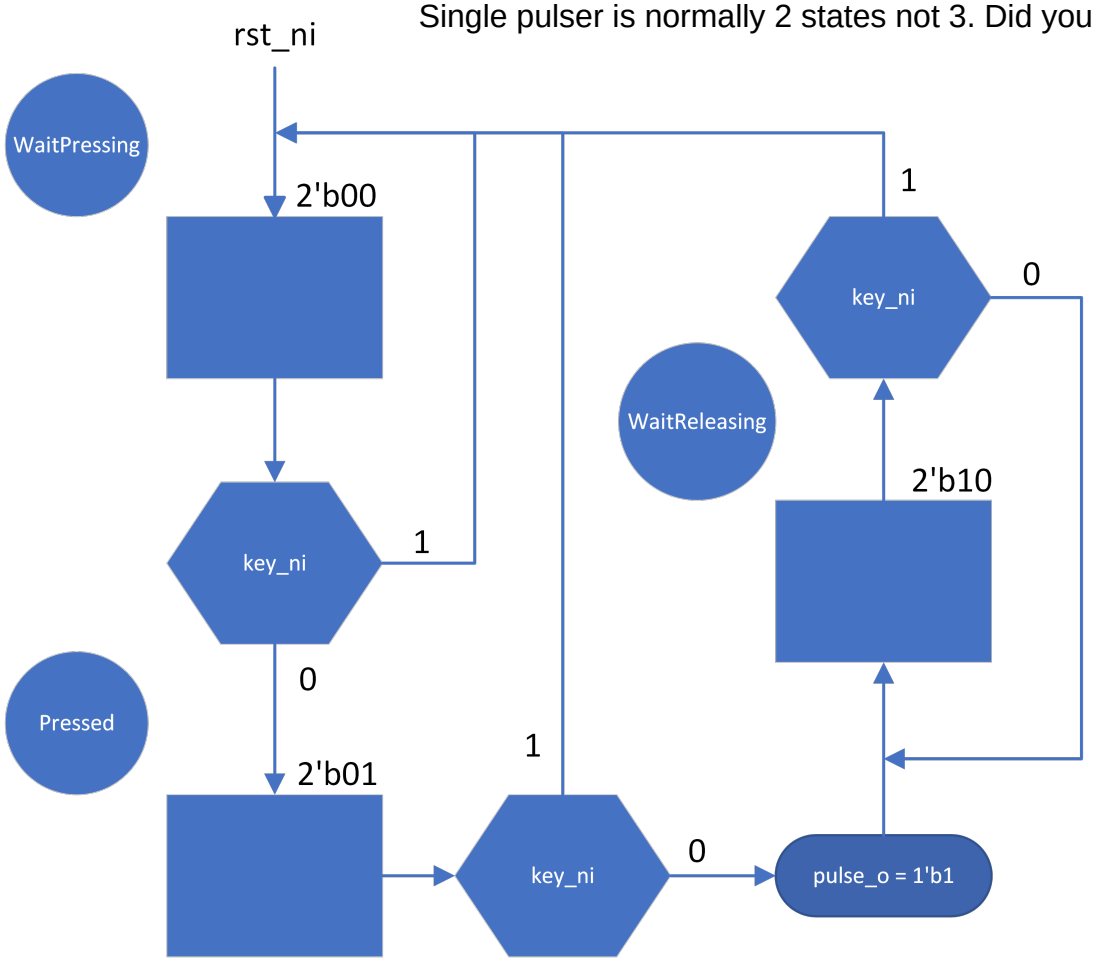


Figure 4: ASM chart of the single pulser module.

```
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;
```

```
32      key_ni = 1'b0;
33      #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
```

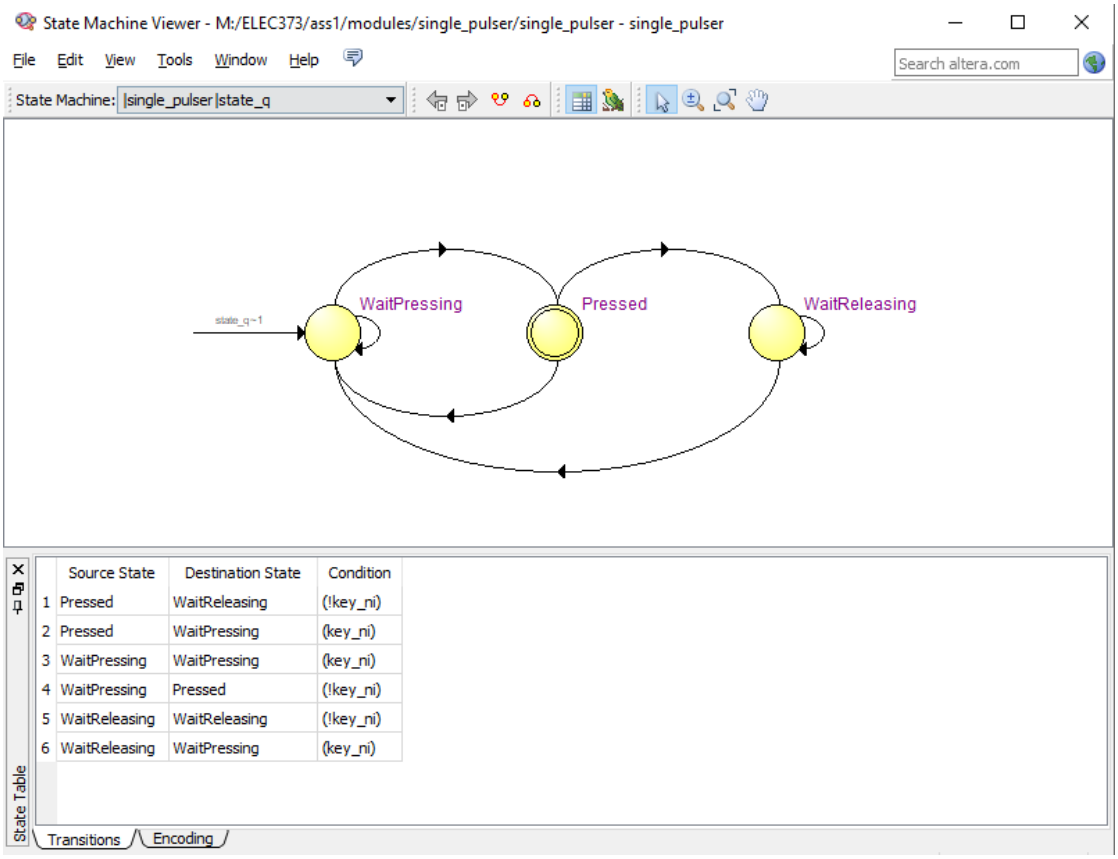


Figure 5: Quartus II state machine viewer shows the expected state machine of the single pulser after synthesis.

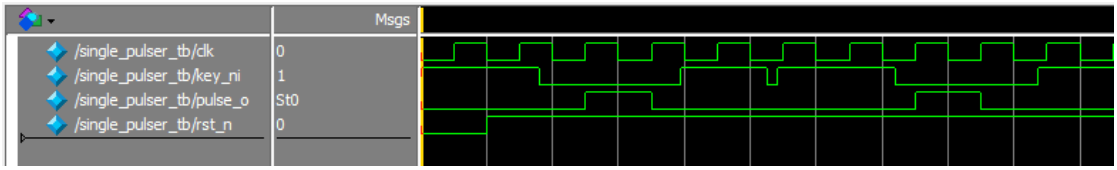


Figure 6: Testbench simulation of the single pulser module.

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

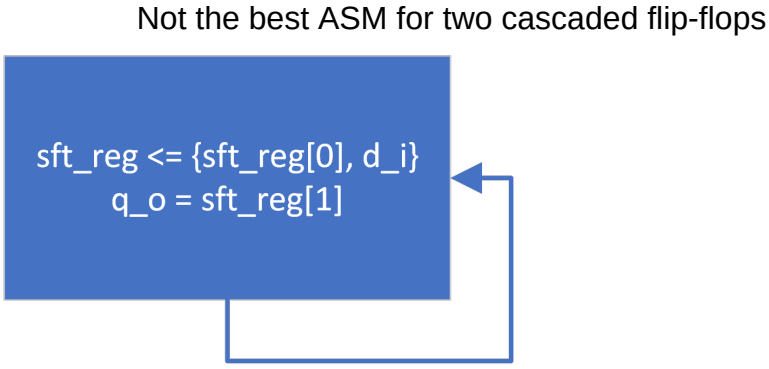


Figure 7: ASM chart of the synchroniser module.

```
module sync (
   input clk_i,
   input d_i,
   output q_o
);

reg [1:0] sft_reg;

always@(posedge clk_i)
   sft_reg <= {sft_reg[0], d_i};

assign q_o = sft_reg[1];

endmodule
```

```
module sync_tb;

// Inputs
reg clk;
reg d_i;

// Outputs
wire q_o;

sync DUT (
   .clk_i(clk),
   .d_i(d_i),
   .q_o(q_o)
);

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

initial begin
   clk = 1'b0;
   d_i = 1'b1;
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
```

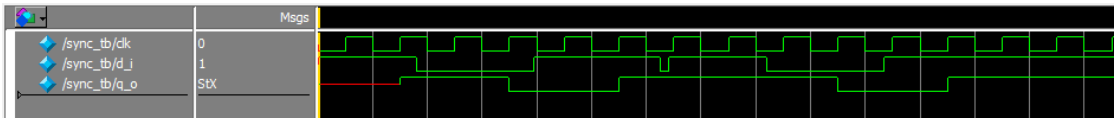


Figure 8: Testbench simulation of the synchroniser module.

Fig. 8 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.

I would have had this block doing the inversion to make the ouptut active high true

3.4 Counter

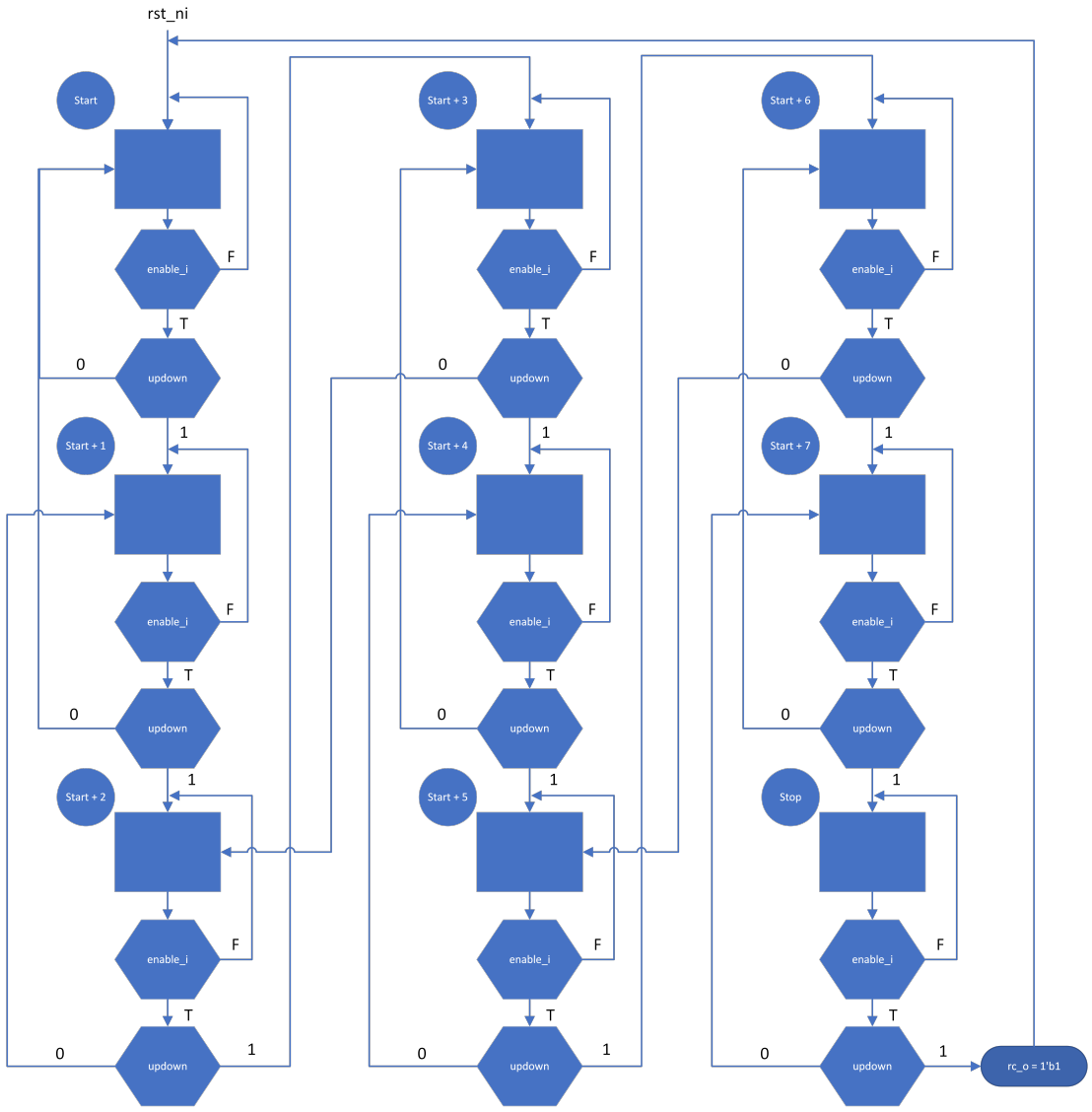


Figure 9: ASM chart of the counter module.

Use behavioural to keep it simple i.e. drop it to a single state with adder and registers.

```
module counter (
    input clk_i,
    input rst_ni,
    input updown_i,
    input enable_i,
    output [3:0] bcd_o,
    output reg rc_o  // ripple carry output
);

reg [3:0] state_d, state_q;
assign bcd_o = state_q;

parameter Start = 4'b0000;
```

```
14  parameter Stop  = 4'b1001;
15
16  always @(posedge clk_i, negedge rst_ni)
17     if (~rst_ni)
18        state_q <= Start;
19     else
20        state_q <= state_d;
21
22  always @(state_q, enable_i, updown_i) begin
23     rc_o = 1'b0;
24     state_d = state_q;  // default assignment next state is present state
25     if (enable_i)
26        if (updown_i)  // count up
27           if (state_d == Stop) begin
28              state_d = Start;
29              rc_o = 1'b1;
30           end else
31              state_d = state_d + 1'b1;
32        else  // count down
33           if (state_d == Start)
34              state_d = Start;
35           else
36              state_d = state_d - 1'b1;
37  end
38
39  endmodule
```

This is not what your ASM shows

```
1   module counter_tb;
2
3   // Inputs
4   reg clk;
5   reg rst_n;
6   reg updown_i;
7   reg enable_i;
8
9   // Outputs
10  wire [3:0] cnt_o;
11  wire ripple_carry_o;
12
13  counter DUT (
14     .clk_i(clk),
15     .rst_ni(rst_n),
16     .updown_i(updown_i),
17     .enable_i(enable_i),
18     .bcd_o(cnt_o),
19     .rc_o(ripple_carry_o)
20  );
21
22  // Create a 50Mhz clock
23  always #10 clk = !clk;  // every ten nanoseconds invert
24
25  initial begin
26     clk = 1'b0;
27     rst_n = 1'b0;
28     enable_i = 1'b0;  // disabled
```

```
29      updown_i = 1'b1;  // count up
30  end
31
32  initial begin
33      #20 rst_n = 1'b1;  // release reset
34
35      // Test count up and ripple carry generation
36      repeat (11) begin
37          @(posedge clk);
38          enable_i = 1'b1;
39          @(posedge clk);
40          enable_i = 1'b0;
41      end
42
43      // Reset
44      @(negedge clk);
45      rst_n = 1'b0;
46      @(negedge clk);
47      rst_n = 1'b1;
48
49      // Test count down
50      repeat (9) begin
51          @(posedge clk);
52          enable_i = 1'b1;
53          @(posedge clk);
54          enable_i = 1'b0;
55      end
56      //updown_i = 1'b0;  // count down
57      repeat (10) begin
58          @(posedge clk);
59          enable_i = 1'b1;
60          updown_i = 1'b0;
61          @(posedge clk);
62          enable_i = 1'b0;
63          updown_i = 1'b1;
64      end
65
66  // Finish the Simulation
67      #100;
68      $finish;
69  end
70
71  endmodule
```

Too small to read

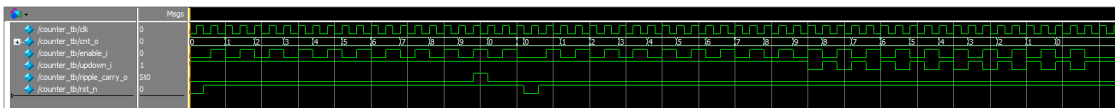


Figure 10: Testbench simulation of the counter module.

Fig. 10 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 zero to nine. Nine set to the biggest digit this counter can count to. Counting once beyond nine returned the count to zero and generated a ripple carry. Next, Reset singal reset the count to zero. Then counting down was tested. Note that if count down to zero and then count down again, the count did not go back to nine but stayed at zero.

3.5 Entry indicator

Not yet designed.

3.6 Shift register

ASM?

```
module shift_reg #(
   parameter DataWidth = 4,
   parameter Depth = 3  // FIXME: Depth is actullay hard-coded
)(
   input clk_i,
   input rst_ni,
   input enable_i,
   input  [DataWidth-1:0] d_i,
   output [DataWidth-1:0] reg_0_o,
   output [DataWidth-1:0] reg_1_o,
   output [DataWidth-1:0] reg_2_o
);

reg [DataWidth-1:0] mem_d [0:Depth-1];
reg [DataWidth-1:0] mem_q [0:Depth-1];

// TODO: more flexible
assign reg_0_o = mem_q[0];
assign reg_1_o = mem_q[1];
assign reg_2_o = mem_q[2];

integer i;

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;
```

```
4
5   // Inputs
6   reg clk;
7   reg rst_n;
8   reg enable_i;
9   reg [DataWidth-1:0] d_i;
10
11  // Outputs
12  wire [DataWidth-1:0] reg_0_o;
13  wire [DataWidth-1:0] reg_1_o;
14  wire [DataWidth-1:0] reg_2_o;
15
16  shift_reg #(
17      .DataWidth(DataWidth)
18  ) DUT (
19      .clk_i(clk),
20      .rst_ni(rst_n),
21      .enable_i(enable_i),
22      .d_i(d_i),
23      .reg_0_o(reg_0_o),
24      .reg_1_o(reg_1_o),
25      .reg_2_o(reg_2_o)
26  );
27
28  // Create a 50Mhz clock
29  always #10 clk = !clk;  // every ten nanoseconds invert
30
31  initial begin
32      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
```

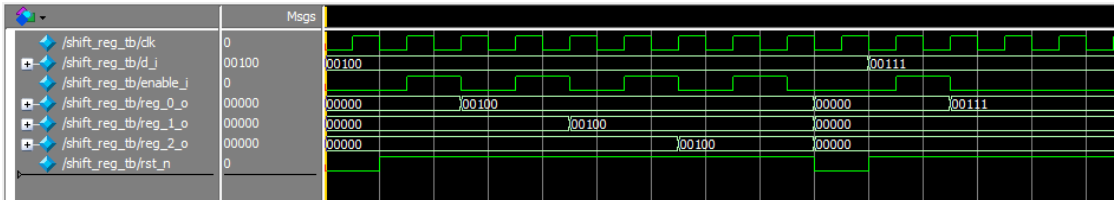


Figure 11: Testbench simulation of the shift register module.

3.7 Comparator

```
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;

wire equal_o;

comparator DUT (
    .bcd_0_i(bcd_0_i),
    .bcd_1_i(bcd_1_i),
    .bcd_2_i(bcd_2_i),
    .bcd_3_i(bcd_3_i),
    .equal_o(equal_o)
);

initial begin
    bcd_0_i = 4'd0;
    bcd_1_i = 4'd0;
    bcd_2_i = 4'd0;
    bcd_3_i = 4'd0;
end

initial begin
    #100;

    bcd_0_i = 4'd2;
    bcd_1_i = 4'd8;
    bcd_2_i = 4'd0;
    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
```

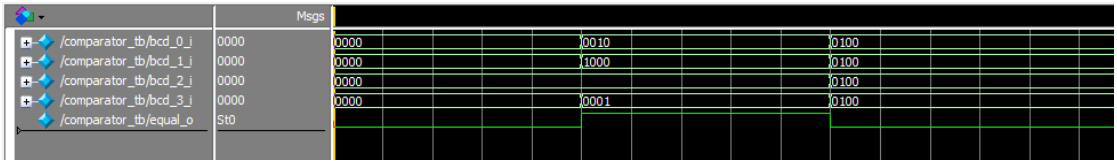


Figure 12: Testbench simulation of the comparator module.

3.8 Lock

```
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
wire ledr_o;
wire ledg_o;

lock DUT (
   .enable_i(enable_i),
   .unlock_i(unlock_i),
   .ledr_o(ledr_o),
   .ledg_o(ledg_o)
);

initial begin
   enable_i = 1'b0;  // disabled
   unlock_i = 1'b0;  // locked
end

initial begin
   #20;

   unlock_i = 1'b1;
   #20;
   unlock_i = 1'b0;
   #20;

   enable_i = 1'b1;  // enabled
   unlock_i = 1'b1;
   #20;
   unlock_i = 1'b0;
```

```
35      #40;
36
37  // Finish the Simulation
38      #100;
39      $finish;
40  end
41
42  endmodule
```

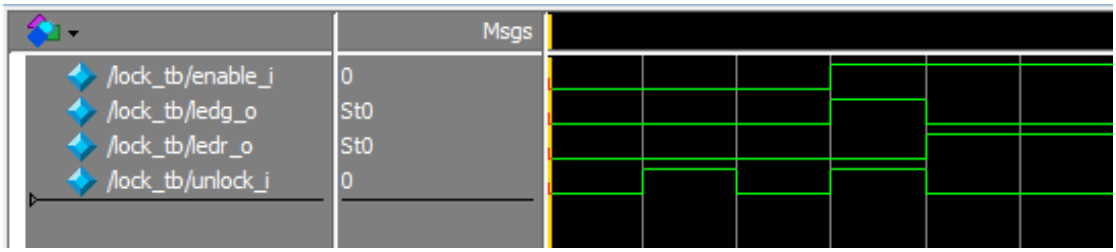
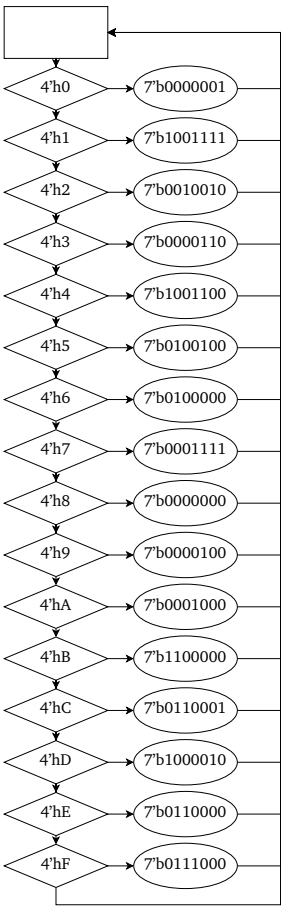


Figure 13: Testbench simulation of the lock module.

3.9 Seven segment decoder



What is the default?

Figure 14: ASM chart of seven segment decoder.

```
/**
For Altera DE2,
  -a-
f|   |b
  -g-
e|   |c
  -d-
*/
module seven_seg_dec (
   input [3:0] bcd_i,  // binary coded decimal input
   output reg [6:0] out_o
);

always @(bcd_i)
   case (bcd_i)
   //                  abcdefg
      4'h0: out_o = 7'b0000001;
      4'h1: out_o = 7'b1001111;
      4'h2: out_o = 7'b0010010;
      4'h3: out_o = 7'b0000110;
      4'h4: out_o = 7'b1001100;
```

```
22          4'h5: out_o = 7'b0100100;
23          4'h6: out_o = 7'b0100000;
24          4'h7: out_o = 7'b0001111;
25          4'h8: out_o = 7'b0000000;
26          4'h9: out_o = 7'b0000100;
27          4'hA: out_o = 7'b0001000;
28          4'hB: out_o = 7'b1100000;
29          4'hC: out_o = 7'b0110001;
30          4'hD: out_o = 7'b1000010;
31          4'hE: out_o = 7'b0110000;
32          4'hF: out_o = 7'b0111000;
33      endcase
34
35  endmodule
```