Hotel Safe Box Lock

Eric Wu, 75286914

University of British Columbia

Description

Inspired by my quarantine hotel safe box (Figure 1), I thought that implementing a hotel safe box lock would be a pretty interesting use for FSMs. However, to narrow the scope of the project, the HEX display was not implemented.

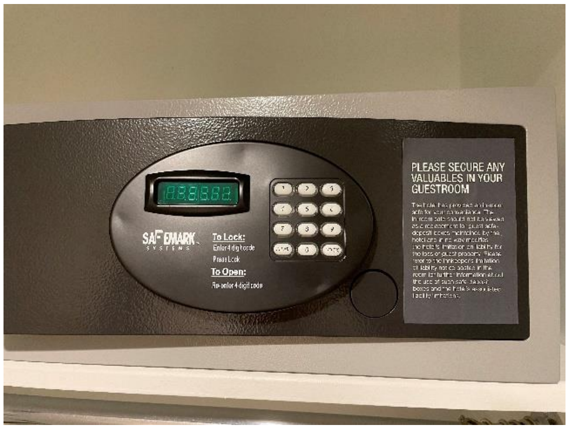


Figure 1.

The design is separated into two modules. The number button capturer and the locking FSM.

The number button capturer assumes number buttons are logic 0 when unpressed and logic 1 when pressed. It not only detects the edge of the button press, but also keeps the value of the number.

The locking FSM can be separated into three parts:

- Setting up a security code (for hotel owners)
- Setting up a customer code (for customer use)
- Unlocking/Resetting

The user interface would look something like this (Figure 2):

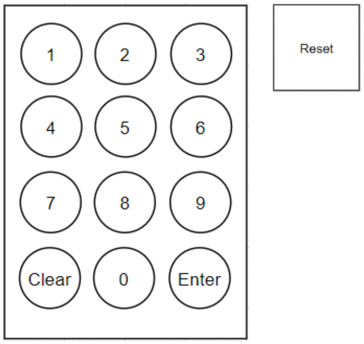


Figure 2.

Block Diagrams

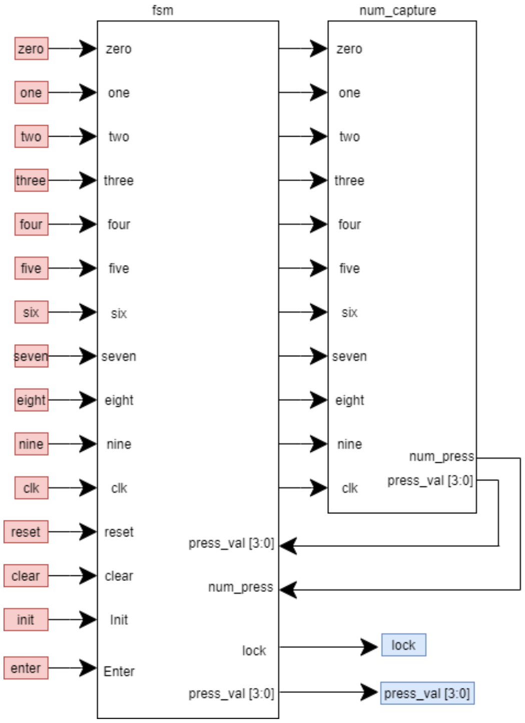


Figure 3. Block diagram of FSM modules (inputs are red, outputs are blue)

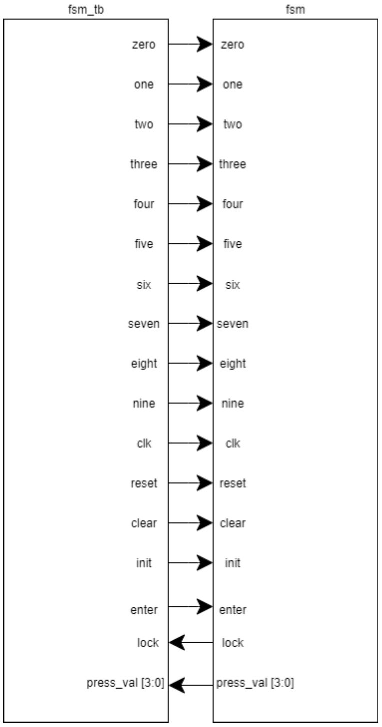


Figure 4. Block diagram of TB (fsm_tb) and DUT (fsm)

State diagram of FSM flow

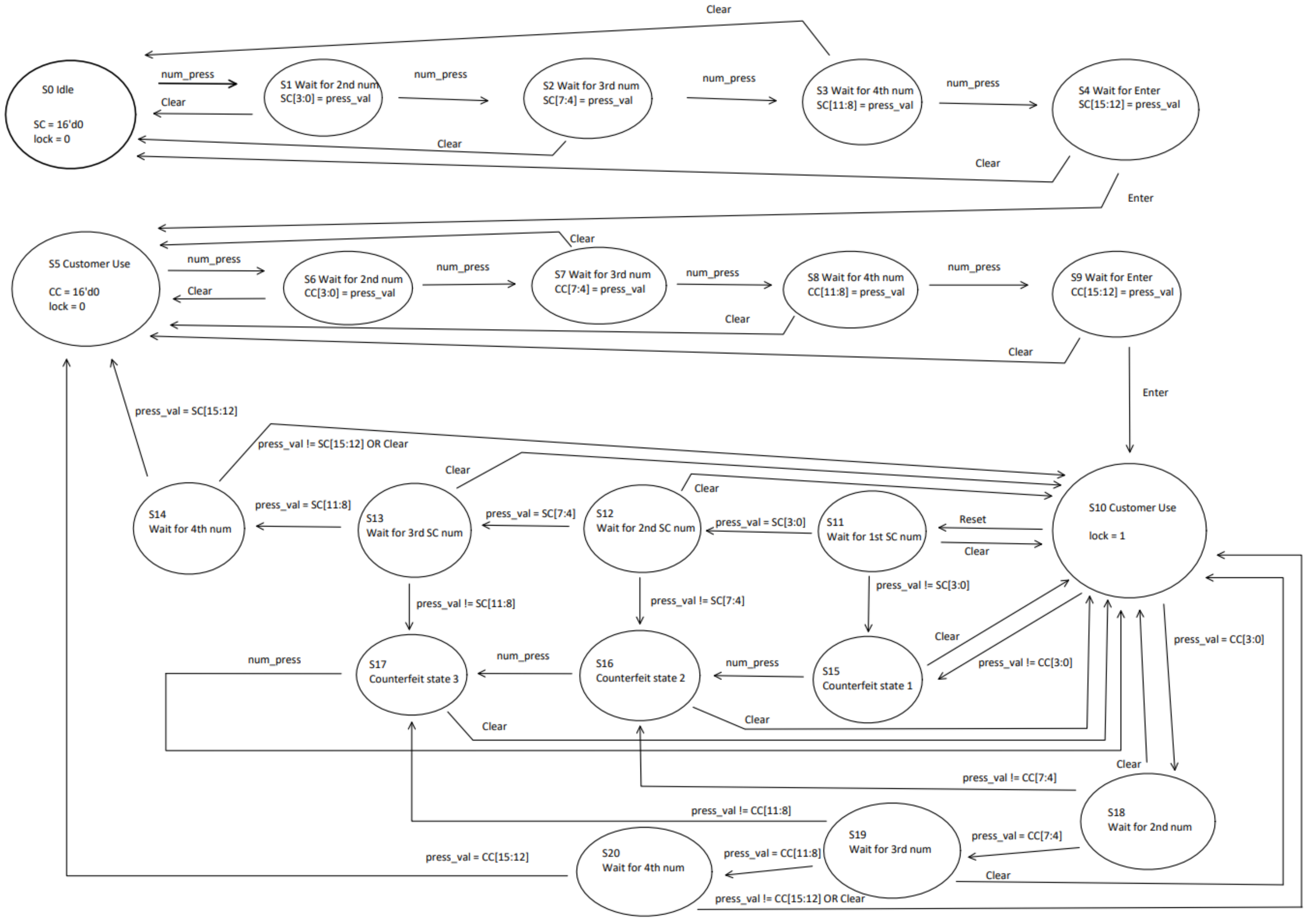


Figure 5. State diagram of FSM

S0-S4 sets up the security code, S5 is the state where it is ready to be used by the customer.

S5-S9 sets up the customer code, S10 is the state where it is locked.

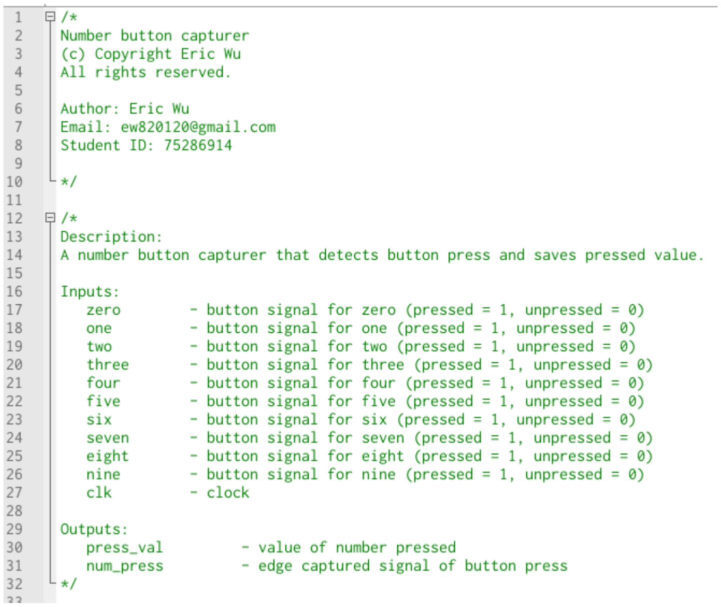
S11-S14 is the reset process. When the customer forgets the customer code, it can be reset by entering the security code.

S15-S17 are counterfeit states. These states prevent logic loopholes when a HEX display is used. If pressing a wrong number goes directly back to S10, the HEX display will tell the person the number is wrong for that digit, meaning that the lock can be easily unlocked with at most 40 tries. With counterfeit states, it needs at most 10000 tries to be unlocked.

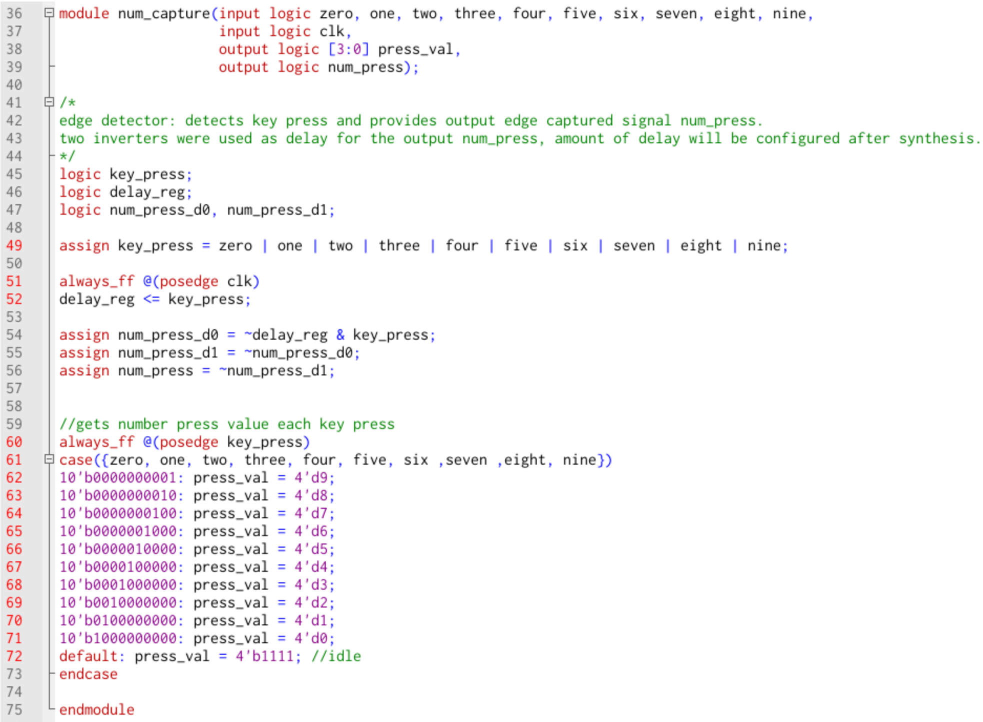
S18-S19 is the normal unlock process.

SystemVerilog Modules

num_capture:

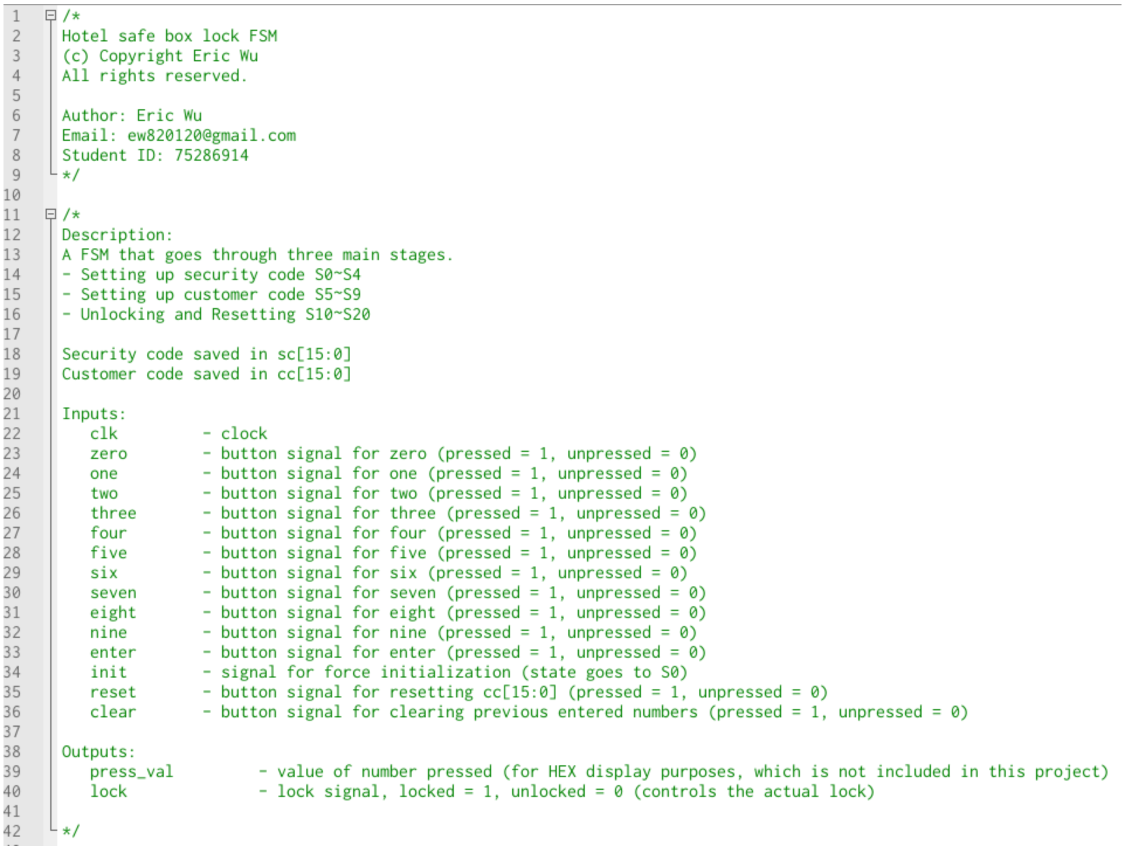


```
/*
Number button capturer
(c) Copyright Eric Wu
All rights reserved.

Author: Eric Wu
Email: ew820120@gmail.com
Student ID: 75286914

*/

/*
Description:
A number button capturer that detects button press and saves pressed value.

Inputs:
   zero        - button signal for zero (pressed = 1, unpressed = 0)
   one         - button signal for one (pressed = 1, unpressed = 0)
   two         - button signal for two (pressed = 1, unpressed = 0)
   three       - button signal for three (pressed = 1, unpressed = 0)
   four        - button signal for four (pressed = 1, unpressed = 0)
   five        - button signal for five (pressed = 1, unpressed = 0)
   six         - button signal for six (pressed = 1, unpressed = 0)
   seven       - button signal for seven (pressed = 1, unpressed = 0)
   eight       - button signal for eight (pressed = 1, unpressed = 0)
   nine        - button signal for nine (pressed = 1, unpressed = 0)
   clk         - clock

Outputs:
   press_val         - value of number pressed
   num_press         - edge captured signal of button press
*/
```

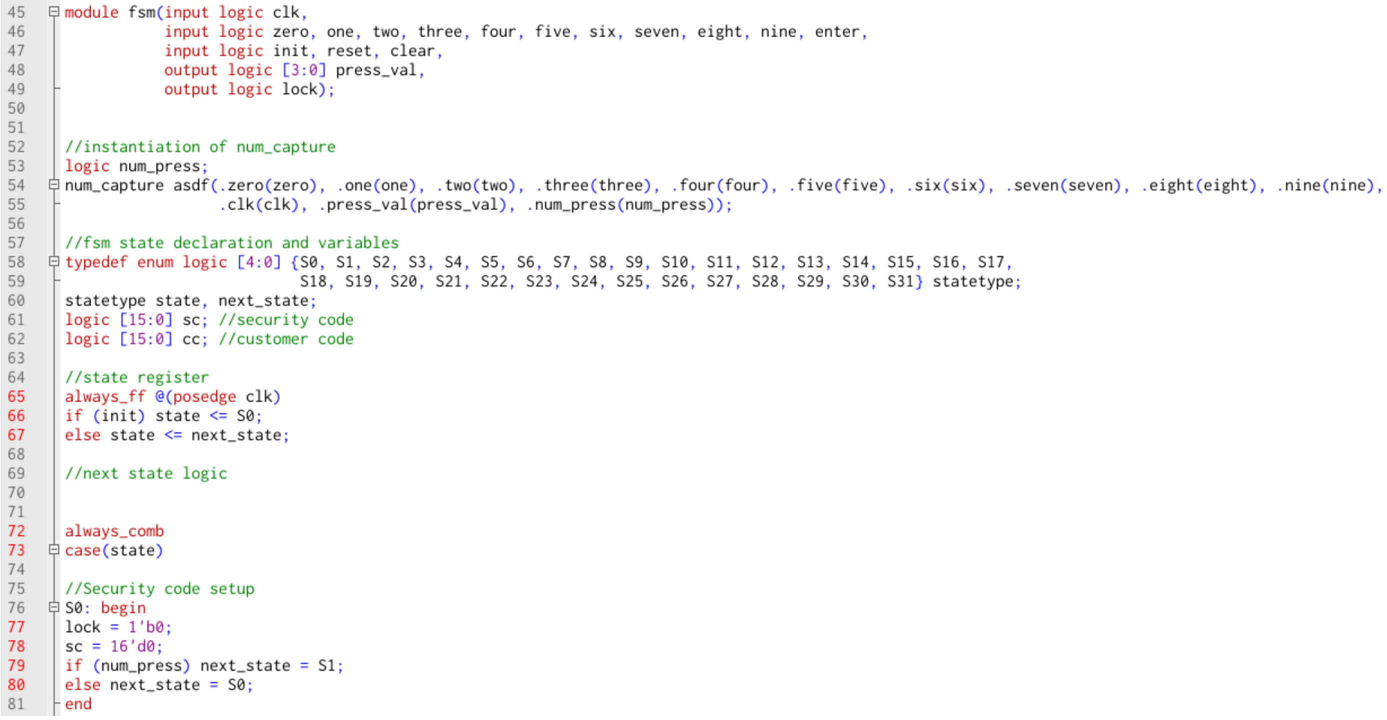


```
module num_capture(input logic zero, one, two, three, four, five, six, seven, eight, nine,
                   input logic clk,
                   output logic [3:0] press_val,
                   output logic num_press);

/*
edge detector: detects key press and provides output edge captured signal num_press.
two inverters were used as delay for the output num_press, amount of delay will be configured after synthesis.
*/
logic key_press;
logic delay_reg;
logic num_press_d0, num_press_d1;

assign key_press = zero | one | two | three | four | five | six | seven | eight | nine;

always_ff @(posedge clk)
delay_reg <= key_press;

assign num_press_d0 = ~delay_reg & key_press;
assign num_press_d1 = ~num_press_d0;
assign num_press = ~num_press_d1;


//gets number press value each key press
always_ff @(posedge key_press)
case({zero, one, two, three, four, five, six ,seven ,eight, nine})
10'b0000000001: press_val = 4'd9;
10'b0000000010: press_val = 4'd8;
10'b0000000100: press_val = 4'd7;
10'b0000001000: press_val = 4'd6;
10'b0000010000: press_val = 4'd5;
10'b0000100000: press_val = 4'd4;
10'b0001000000: press_val = 4'd3;
10'b0010000000: press_val = 4'd2;
10'b0100000000: press_val = 4'd1;
10'b1000000000: press_val = 4'd0;
default: press_val = 4'b1111; //idle
endcase

endmodule
```

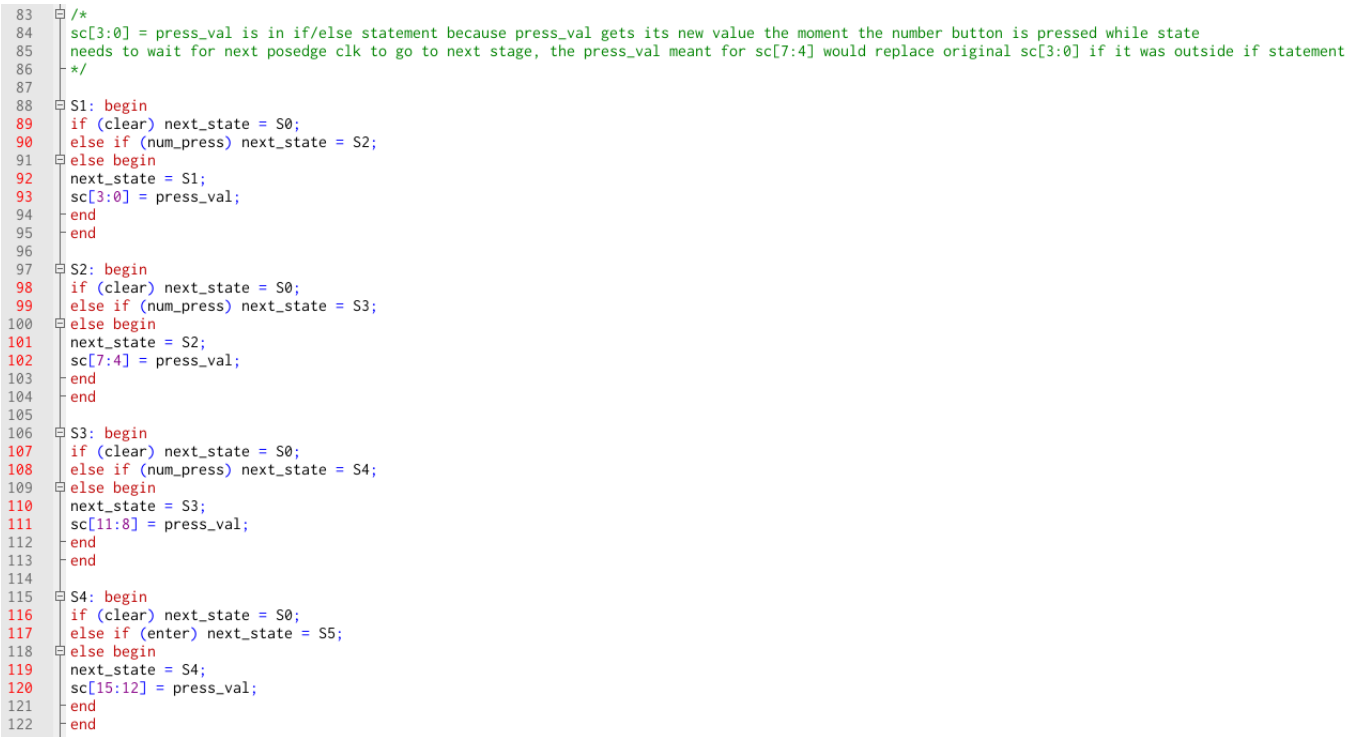
fsm:



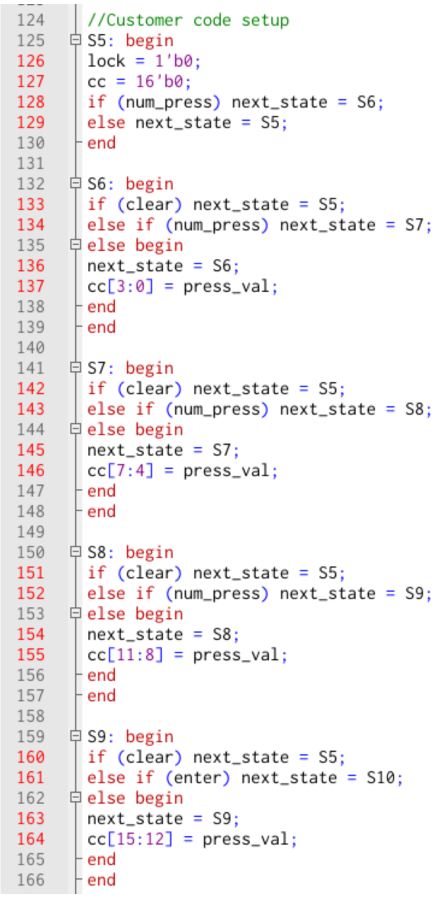
```
/*
Hotel safe box lock FSM
(c) Copyright Eric Wu
All rights reserved.

Author: Eric Wu
Email: ew820120@gmail.com
Student ID: 75286914
*/

/*
Description:
A FSM that goes through three main stages.
- Setting up security code S0~S4
- Setting up customer code S5~S9
- Unlocking and Resetting S10~S20

Security code saved in sc[15:0]
Customer code saved in cc[15:0]

Inputs:
   clk         - clock
   zero        - button signal for zero (pressed = 1, unpressed = 0)
   one         - button signal for one (pressed = 1, unpressed = 0)
   two         - button signal for two (pressed = 1, unpressed = 0)
   three       - button signal for three (pressed = 1, unpressed = 0)
   four        - button signal for four (pressed = 1, unpressed = 0)
   five        - button signal for five (pressed = 1, unpressed = 0)
   six         - button signal for six (pressed = 1, unpressed = 0)
   seven       - button signal for seven (pressed = 1, unpressed = 0)
   eight       - button signal for eight (pressed = 1, unpressed = 0)
   nine        - button signal for nine (pressed = 1, unpressed = 0)
   enter       - button signal for enter (pressed = 1, unpressed = 0)
   init        - signal for force initialization (state goes to S0)
   reset       - button signal for resetting cc[15:0] (pressed = 1, unpressed = 0)
   clear       - button signal for clearing previous entered numbers (pressed = 1, unpressed = 0)

Outputs:
   press_val         - value of number pressed (for HEX display purposes, which is not included in this project)
   lock              - lock signal, locked = 1, unlocked = 0 (controls the actual lock)

*/
```



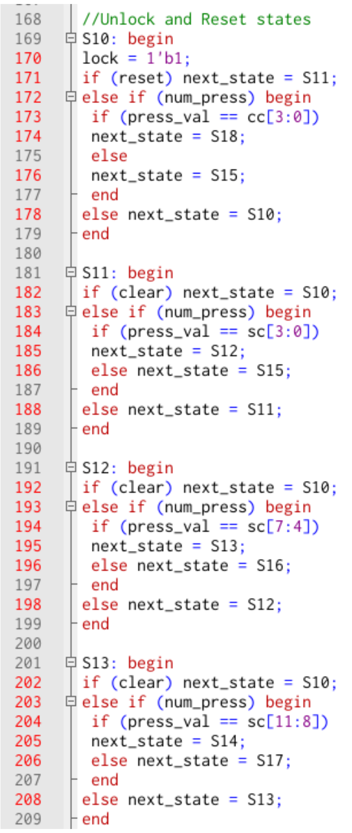
```
module fsm(input logic clk,
           input logic zero, one, two, three, four, five, six, seven, eight, nine, enter,
           input logic init, reset, clear,
           output logic [3:0] press_val,
           output logic lock);


//instantiation of num_capture
logic num_press;
num_capture asdf(.zero(zero), .one(one), .two(two), .three(three), .four(four), .five(five), .six(six), .seven(seven), .eight(eight), .nine(nine),
                 .clk(clk), .press_val(press_val), .num_press(num_press));

//fsm state declaration and variables
typedef enum logic [4:0] {S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17,
                          S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31} statetype;
statetype state, next_state;
logic [15:0] sc; //security code
logic [15:0] cc; //customer code

//state register
always_ff @(posedge clk)
if (init) state <= S0;
else state <= next_state;

//next state logic


always_comb
case(state)

//Security code setup
S0: begin
lock = 1'b0;
sc = 16'd0;
if (num_press) next_state = S1;
else next_state = S0;
end
```



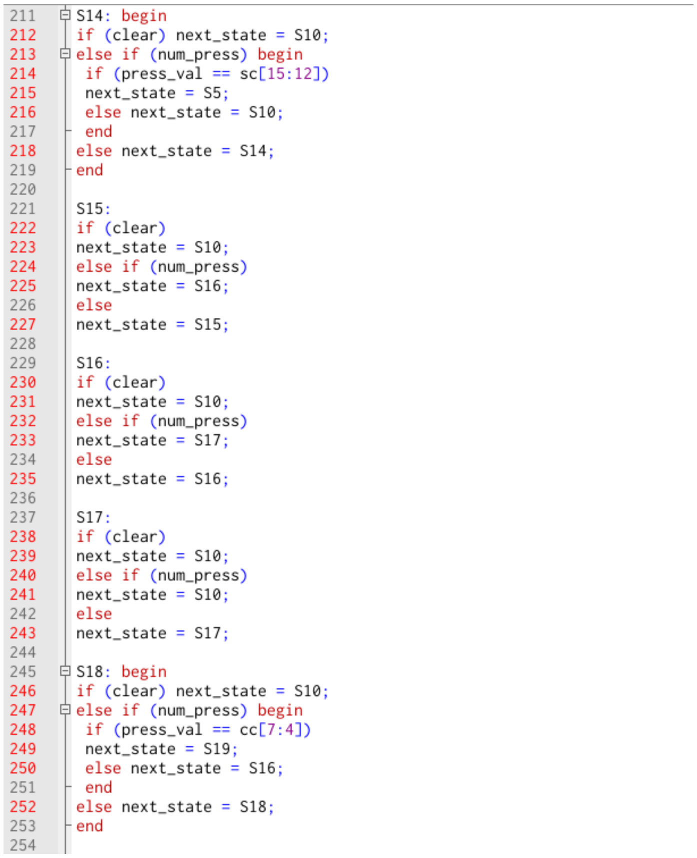
```
/*
sc[3:0] = press_val is in if/else statement because press_val gets its new value the moment the number button is pressed while state
needs to wait for next posedge clk to go to next stage, the press_val meant for sc[7:4] would replace original sc[3:0] if it was outside if statement
*/

S1: begin
if (clear) next_state = S0;
else if (num_press) next_state = S2;
else begin
next_state = S1;
sc[3:0] = press_val;
end
end

S2: begin
if (clear) next_state = S0;
else if (num_press) next_state = S3;
else begin
next_state = S2;
sc[7:4] = press_val;
end
end

S3: begin
if (clear) next_state = S0;
else if (num_press) next_state = S4;
else begin
next_state = S3;
sc[11:8] = press_val;
end
end

S4: begin
if (clear) next_state = S0;
else if (enter) next_state = S5;
else begin
next_state = S4;
sc[15:12] = press_val;
end
end
```



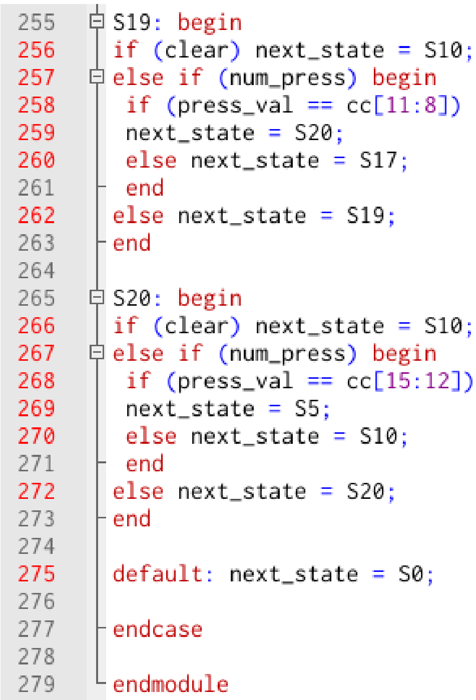
```
//Customer code setup
S5: begin
lock = 1'b0;
cc = 16'b0;
if (num_press) next_state = S6;
else next_state = S5;
end

S6: begin
if (clear) next_state = S5;
else if (num_press) next_state = S7;
else begin
next_state = S6;
cc[3:0] = press_val;
end
end

S7: begin
if (clear) next_state = S5;
else if (num_press) next_state = S8;
else begin
next_state = S7;
cc[7:4] = press_val;
end
end

S8: begin
if (clear) next_state = S5;
else if (num_press) next_state = S9;
else begin
next_state = S8;
cc[11:8] = press_val;
end
end

S9: begin
if (clear) next_state = S5;
else if (enter) next_state = S10;
else begin
next_state = S9;
cc[15:12] = press_val;
end
end
```



```
//Unlock and Reset states
S10: begin
lock = 1'b1;
if (reset) next_state = S11;
else if (num_press) begin
 if (press_val == cc[3:0])
 next_state = S18;
 else
 next_state = S15;
 end
else next_state = S10;
end

S11: begin
if (clear) next_state = S10;
else if (num_press) begin
 if (press_val == sc[3:0])
 next_state = S12;
 else next_state = S15;
 end
else next_state = S11;
end

S12: begin
if (clear) next_state = S10;
else if (num_press) begin
 if (press_val == sc[7:4])
 next_state = S13;
 else next_state = S16;
 end
else next_state = S12;
end

S13: begin
if (clear) next_state = S10;
else if (num_press) begin
 if (press_val == sc[11:8])
 next_state = S14;
 else next_state = S17;
 end
else next_state = S13;
end
```



```
S14: begin
if (clear) next_state = S10;
else if (num_press) begin
 if (press_val == sc[15:12])
 next_state = S5;
 else next_state = S10;
 end
else next_state = S14;
end

S15:
if (clear)
next_state = S10;
else if (num_press)
next_state = S16;
else
next_state = S15;

S16:
if (clear)
next_state = S10;
else if (num_press)
next_state = S17;
else
next_state = S16;

S17:
if (clear)
next_state = S10;
else if (num_press)
next_state = S10;
else
next_state = S17;

S18: begin
if (clear) next_state = S10;
else if (num_press) begin
 if (press_val == cc[7:4])
 next_state = S19;
 else next_state = S16;
 end
else next_state = S18;
end
```

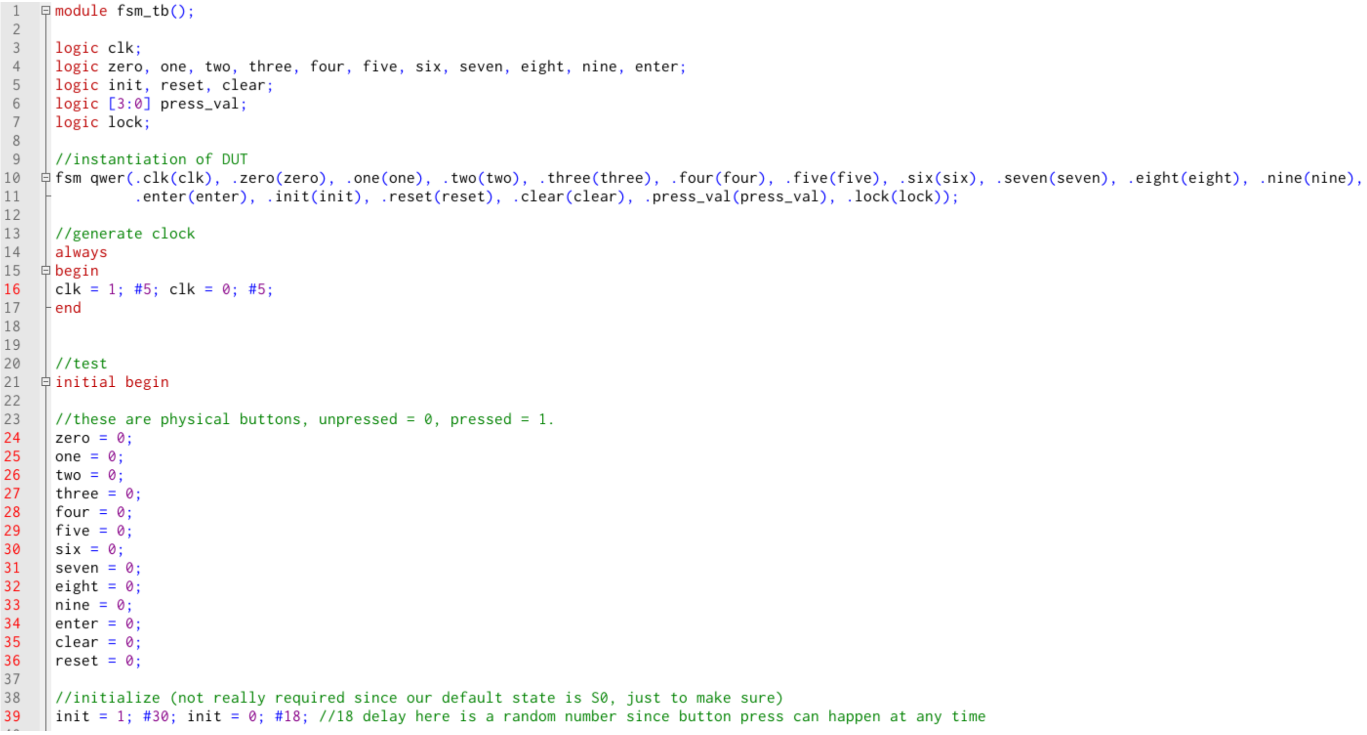


```
S19: begin
if (clear) next_state = S10;
else if (num_press) begin
 if (press_val == cc[11:8])
 next_state = S20;
 else next_state = S17;
 end
else next_state = S19;
end

S20: begin
if (clear) next_state = S10;
else if (num_press) begin
 if (press_val == cc[15:12])
 next_state = S5;
 else next_state = S10;
 end
else next_state = S20;
end

default: next_state = S0;

endcase

endmodule
```

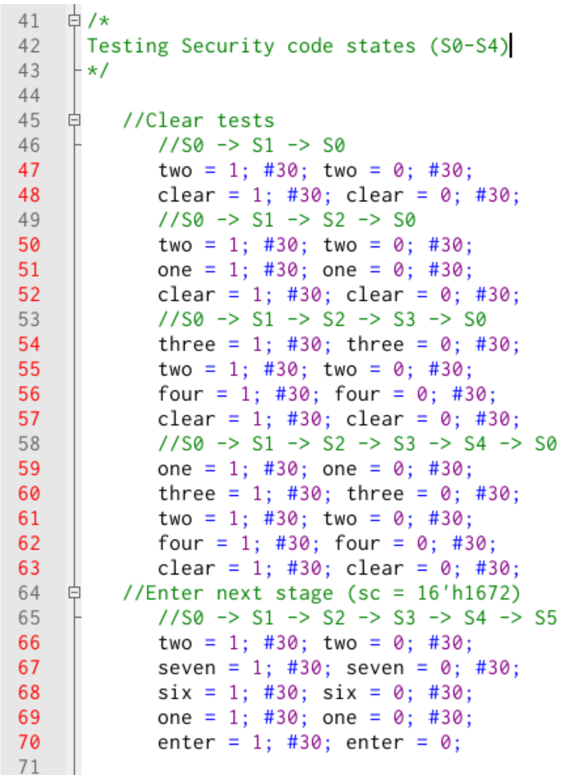
fsm_tb:

My testbench goes through every possible situation.

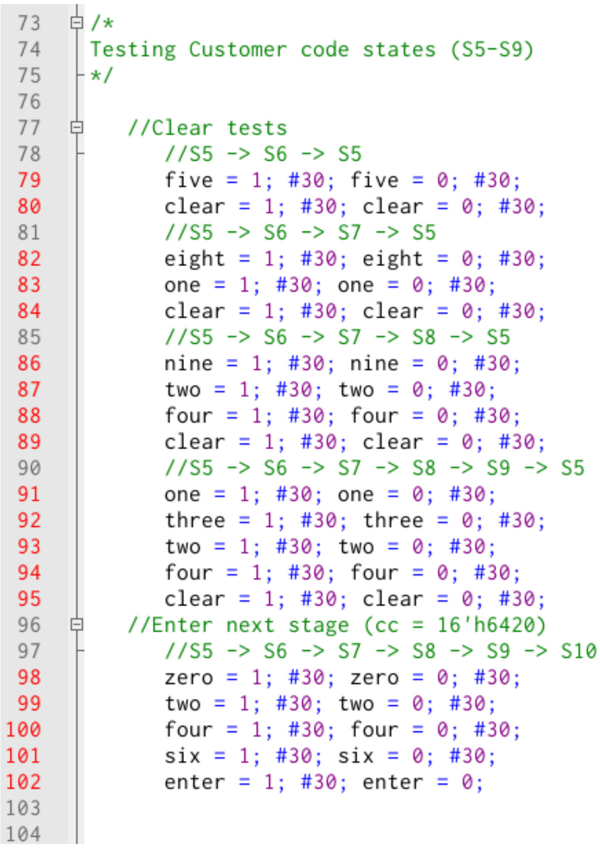
DUT was made sure working by checking “lock” signal, states and sc/cc values.



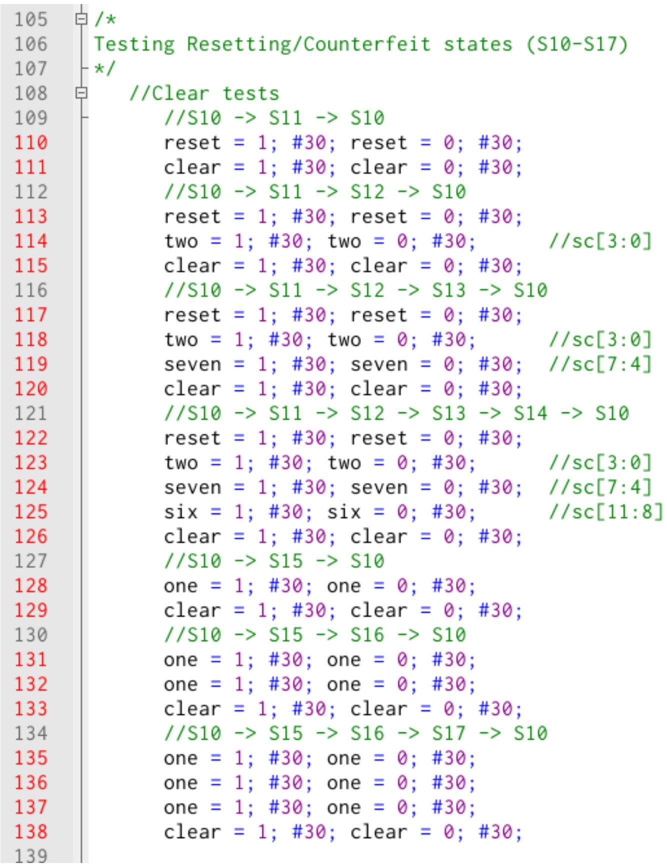
```
module fsm_tb();

logic clk;
logic zero, one, two, three, four, five, six, seven, eight, nine, enter;
logic init, reset, clear;
logic [3:0] press_val;
logic lock;

//instantiation of DUT
fsm qwer(.clk(clk), .zero(zero), .one(one), .two(two), .three(three), .four(four), .five(five), .six(six), .seven(seven), .eight(eight), .nine(nine),
         .enter(enter), .init(init), .reset(reset), .clear(clear), .press_val(press_val), .lock(lock));

//generate clock
always
begin
clk = 1; #5; clk = 0; #5;
end


//test
initial begin

//these are physical buttons, unpressed = 0, pressed = 1.
zero = 0;
one = 0;
two = 0;
three = 0;
four = 0;
five = 0;
six = 0;
seven = 0;
eight = 0;
nine = 0;
enter = 0;
clear = 0;
reset = 0;

//initialize (not really required since our default state is S0, just to make sure)
init = 1; #30; init = 0; #18; //18 delay here is a random number since button press can happen at any time
```



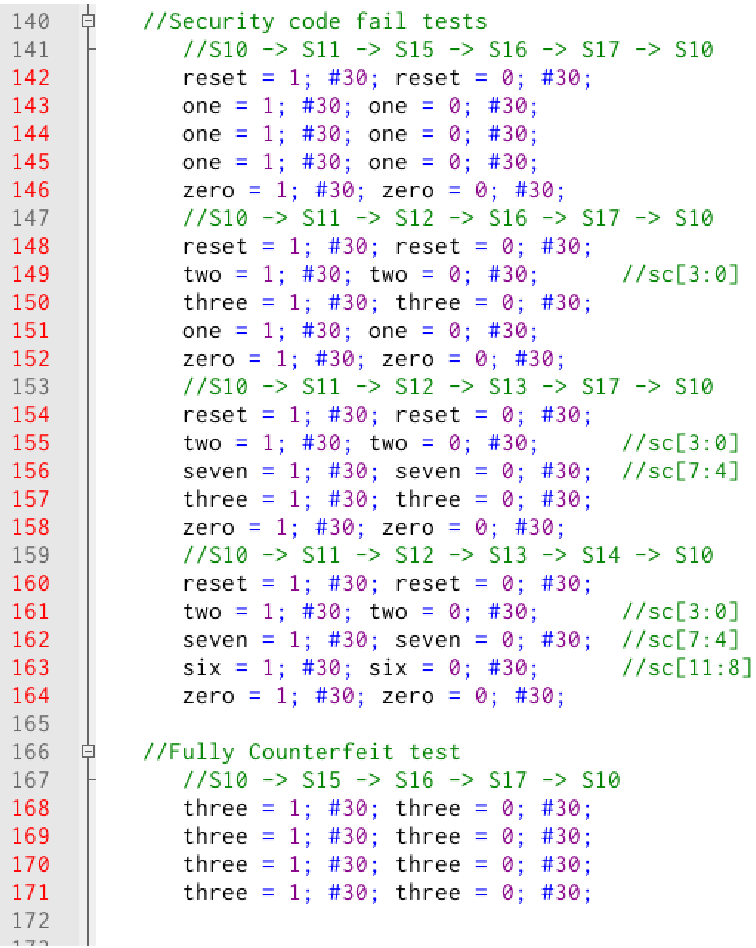
```
/*
Testing Security code states (S0-S4)
*/

    //Clear tests
        //S0 -> S1 -> S0
        two = 1; #30; two = 0; #30;
        clear = 1; #30; clear = 0; #30;
        //S0 -> S1 -> S2 -> S0
        two = 1; #30; two = 0; #30;
        one = 1; #30; one = 0; #30;
        clear = 1; #30; clear = 0; #30;
        //S0 -> S1 -> S2 -> S3 -> S0
        three = 1; #30; three = 0; #30;
        two = 1; #30; two = 0; #30;
        four = 1; #30; four = 0; #30;
        clear = 1; #30; clear = 0; #30;
        //S0 -> S1 -> S2 -> S3 -> S4 -> S0
        one = 1; #30; one = 0; #30;
        three = 1; #30; three = 0; #30;
        two = 1; #30; two = 0; #30;
        four = 1; #30; four = 0; #30;
        clear = 1; #30; clear = 0; #30;
    //Enter next stage (sc = 16'h1672)
        //S0 -> S1 -> S2 -> S3 -> S4 -> S5
        two = 1; #30; two = 0; #30;
        seven = 1; #30; seven = 0; #30;
        six = 1; #30; six = 0; #30;
        one = 1; #30; one = 0; #30;
        enter = 1; #30; enter = 0;

```



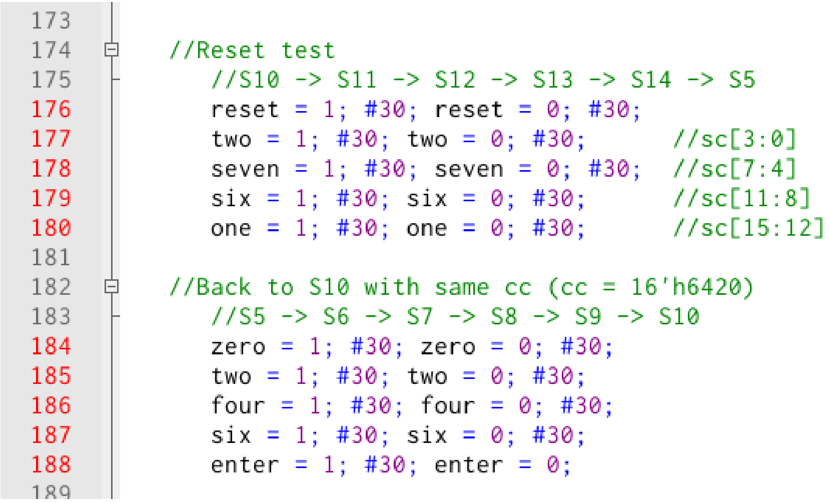
```
/*
Testing Customer code states (S5-S9)
*/

   //Clear tests
      //S5 -> S6 -> S5
      five = 1; #30; five = 0; #30;
      clear = 1; #30; clear = 0; #30;
      //S5 -> S6 -> S7 -> S5
      eight = 1; #30; eight = 0; #30;
      one = 1; #30; one = 0; #30;
      clear = 1; #30; clear = 0; #30;
      //S5 -> S6 -> S7 -> S8 -> S5
      nine = 1; #30; nine = 0; #30;
      two = 1; #30; two = 0; #30;
      four = 1; #30; four = 0; #30;
      clear = 1; #30; clear = 0; #30;
      //S5 -> S6 -> S7 -> S8 -> S9 -> S5
      one = 1; #30; one = 0; #30;
      three = 1; #30; three = 0; #30;
      two = 1; #30; two = 0; #30;
      four = 1; #30; four = 0; #30;
      clear = 1; #30; clear = 0; #30;
   //Enter next stage (cc = 16'h6420)
      //S5 -> S6 -> S7 -> S8 -> S9 -> S10
      zero = 1; #30; zero = 0; #30;
      two = 1; #30; two = 0; #30;
      four = 1; #30; four = 0; #30;
      six = 1; #30; six = 0; #30;
      enter = 1; #30; enter = 0;

```



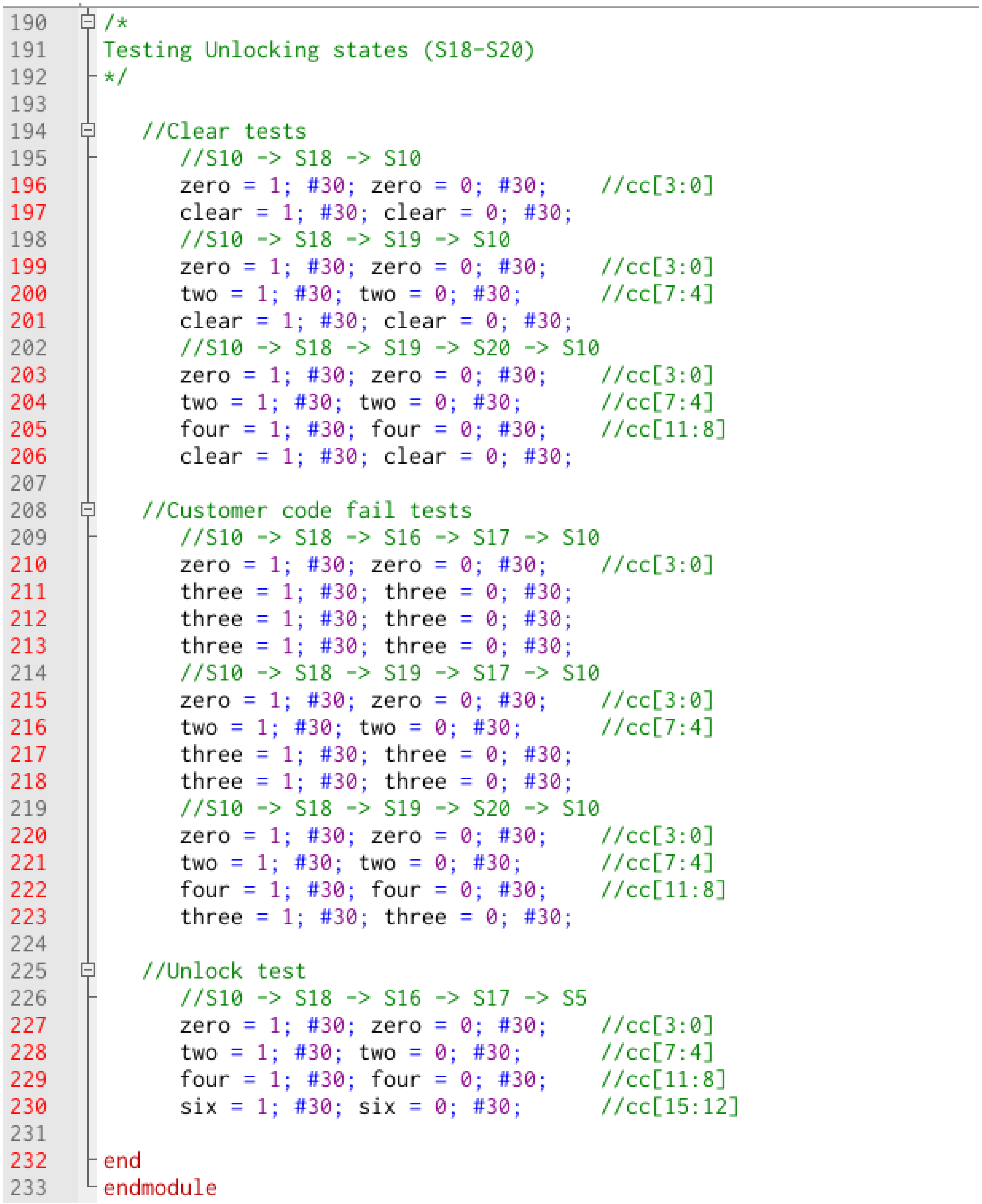
```
/*
Testing Resetting/Counterfeit states (S10-S17)
*/
   //Clear tests
      //S10 -> S11 -> S10
      reset = 1; #30; reset = 0; #30;
      clear = 1; #30; clear = 0; #30;
      //S10 -> S11 -> S12 -> S10
      reset = 1; #30; reset = 0; #30;
      two = 1; #30; two = 0; #30;     //sc[3:0]
      clear = 1; #30; clear = 0; #30;
      //S10 -> S11 -> S12 -> S13 -> S10
      reset = 1; #30; reset = 0; #30;
      two = 1; #30; two = 0; #30;     //sc[3:0]
      seven = 1; #30; seven = 0; #30; //sc[7:4]
      clear = 1; #30; clear = 0; #30;
      //S10 -> S11 -> S12 -> S13 -> S14 -> S10
      reset = 1; #30; reset = 0; #30;
      two = 1; #30; two = 0; #30;     //sc[3:0]
      seven = 1; #30; seven = 0; #30; //sc[7:4]
      six = 1; #30; six = 0; #30;     //sc[11:8]
      clear = 1; #30; clear = 0; #30;
      //S10 -> S15 -> S10
      one = 1; #30; one = 0; #30;
      clear = 1; #30; clear = 0; #30;
      //S10 -> S15 -> S16 -> S10
      one = 1; #30; one = 0; #30;
      one = 1; #30; one = 0; #30;
      clear = 1; #30; clear = 0; #30;
      //S10 -> S15 -> S16 -> S17 -> S10
      one = 1; #30; one = 0; #30;
      one = 1; #30; one = 0; #30;
      one = 1; #30; one = 0; #30;
      clear = 1; #30; clear = 0; #30;

```



```
//Security code fail tests
   //S10 -> S11 -> S15 -> S16 -> S17 -> S10
   reset = 1; #30; reset = 0; #30;
   one = 1; #30; one = 0; #30;
   one = 1; #30; one = 0; #30;
   one = 1; #30; one = 0; #30;
   zero = 1; #30; zero = 0; #30;
   //S10 -> S11 -> S12 -> S16 -> S17 -> S10
   reset = 1; #30; reset = 0; #30;
   two = 1; #30; two = 0; #30;      //sc[3:0]
   three = 1; #30; three = 0; #30;
   one = 1; #30; one = 0; #30;
   zero = 1; #30; zero = 0; #30;
   //S10 -> S11 -> S12 -> S13 -> S17 -> S10
   reset = 1; #30; reset = 0; #30;
   two = 1; #30; two = 0; #30;      //sc[3:0]
   seven = 1; #30; seven = 0; #30;  //sc[7:4]
   three = 1; #30; three = 0; #30;
   zero = 1; #30; zero = 0; #30;
   //S10 -> S11 -> S12 -> S13 -> S14 -> S10
   reset = 1; #30; reset = 0; #30;
   two = 1; #30; two = 0; #30;      //sc[3:0]
   seven = 1; #30; seven = 0; #30;  //sc[7:4]
   six = 1; #30; six = 0; #30;      //sc[11:8]
   zero = 1; #30; zero = 0; #30;

//Fully Counterfeit test
   //S10 -> S15 -> S16 -> S17 -> S10
   three = 1; #30; three = 0; #30;
   three = 1; #30; three = 0; #30;
   three = 1; #30; three = 0; #30;
   three = 1; #30; three = 0; #30;

```

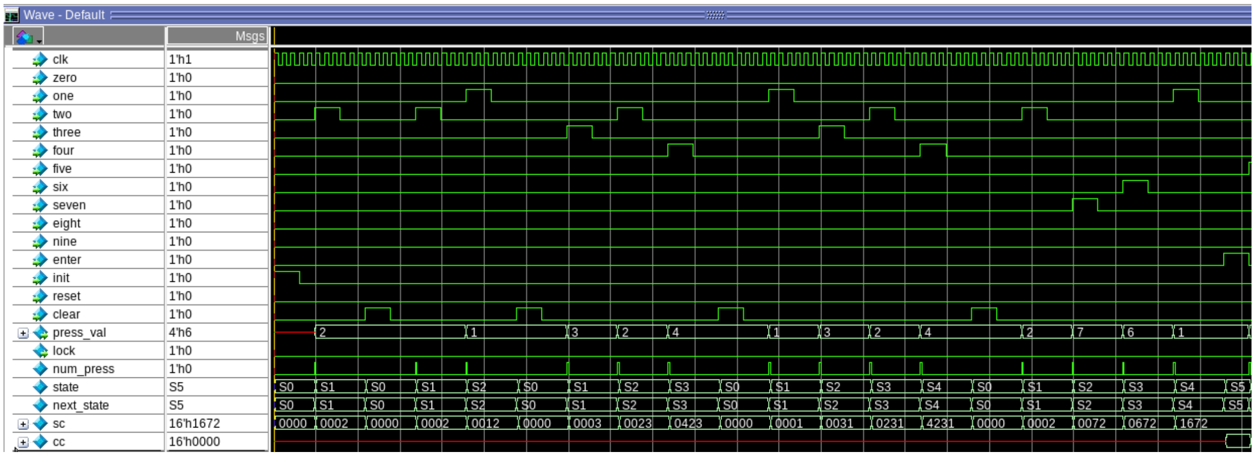


```

//Reset test
   //S10 -> S11 -> S12 -> S13 -> S14 -> S5
   reset = 1; #30; reset = 0; #30;
   two = 1; #30; two = 0; #30;      //sc[3:0]
   seven = 1; #30; seven = 0; #30;  //sc[7:4]
   six = 1; #30; six = 0; #30;      //sc[11:8]
   one = 1; #30; one = 0; #30;      //sc[15:12]

//Back to S10 with same cc (cc = 16'h6420)
   //S5 -> S6 -> S7 -> S8 -> S9 -> S10
   zero = 1; #30; zero = 0; #30;
   two = 1; #30; two = 0; #30;
   four = 1; #30; four = 0; #30;
   six = 1; #30; six = 0; #30;
   enter = 1; #30; enter = 0;
```



```
/*
Testing Unlocking states (S18-S20)
*/

   //Clear tests
      //S10 -> S18 -> S10
      zero = 1; #30; zero = 0; #30;    //cc[3:0]
      clear = 1; #30; clear = 0; #30;
      //S10 -> S18 -> S19 -> S10
      zero = 1; #30; zero = 0; #30;    //cc[3:0]
      two = 1; #30; two = 0; #30;      //cc[7:4]
      clear = 1; #30; clear = 0; #30;
      //S10 -> S18 -> S19 -> S20 -> S10
      zero = 1; #30; zero = 0; #30;    //cc[3:0]
      two = 1; #30; two = 0; #30;      //cc[7:4]
      four = 1; #30; four = 0; #30;    //cc[11:8]
      clear = 1; #30; clear = 0; #30;

   //Customer code fail tests
      //S10 -> S18 -> S16 -> S17 -> S10
      zero = 1; #30; zero = 0; #30;    //cc[3:0]
      three = 1; #30; three = 0; #30;
      three = 1; #30; three = 0; #30;
      three = 1; #30; three = 0; #30;
      //S10 -> S18 -> S19 -> S17 -> S10
      zero = 1; #30; zero = 0; #30;    //cc[3:0]
      two = 1; #30; two = 0; #30;      //cc[7:4]
      three = 1; #30; three = 0; #30;
      three = 1; #30; three = 0; #30;
      //S10 -> S18 -> S19 -> S20 -> S10
      zero = 1; #30; zero = 0; #30;    //cc[3:0]
      two = 1; #30; two = 0; #30;      //cc[7:4]
      four = 1; #30; four = 0; #30;    //cc[11:8]
      three = 1; #30; three = 0; #30;

   //Unlock test
      //S10 -> S18 -> S16 -> S17 -> S5
      zero = 1; #30; zero = 0; #30;    //cc[3:0]
      two = 1; #30; two = 0; #30;      //cc[7:4]
      four = 1; #30; four = 0; #30;    //cc[11:8]
      six = 1; #30; six = 0; #30;      //cc[15:12]

end
endmodule
```

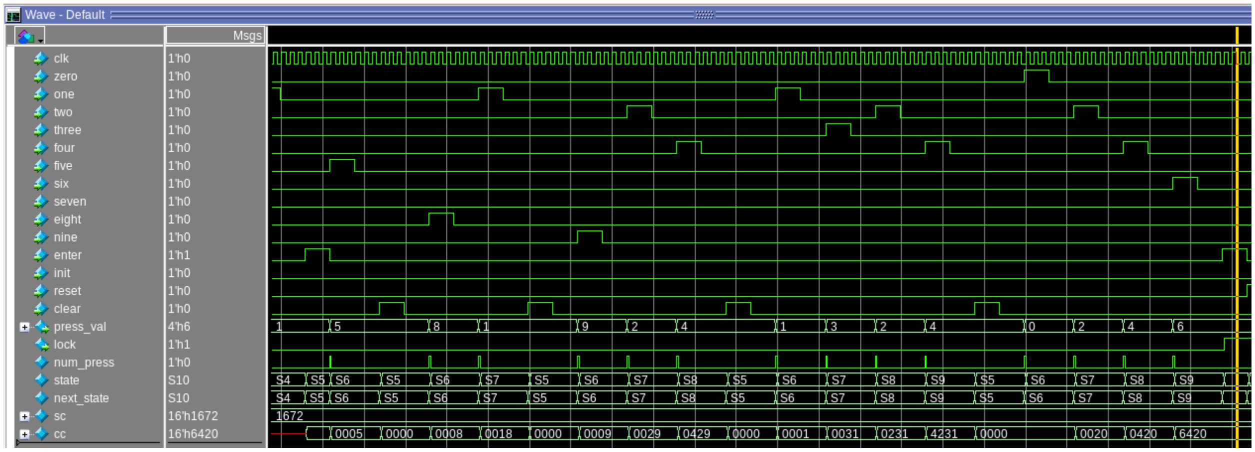
Simulation



Wave - Default

	Msgs
clk	1'h1
zero	1'h0
one	1'h0
two	1'h0
three	1'h0
four	1'h0
five	1'h0
six	1'h0
seven	1'h0
eight	1'h0
nine	1'h0
enter	1'h0
init	1'h0
reset	1'h0
clear	1'h0
press_val	4'h6
lock	1'h0
num_press	1'h0
state	S5
next_state	S5
sc	16'h1672
cc	16'h0000

Figure 6. Waveform of “Testing security code states (S0-S4)” in TB



Wave - Default

	Msgs
clk	1'h0
zero	1'h0
one	1'h0
two	1'h0
three	1'h0
four	1'h0
five	1'h0
six	1'h0
seven	1'h0
eight	1'h0
nine	1'h0
enter	1'h1
init	1'h0
reset	1'h0
clear	1'h0
press_val	4'h6
lock	1'h1
num_press	1'h0
state	S10
next_state	S10
sc	16'h1672
cc	16'h6420

1 5 8 1 9 2 4 1 3 2 4 0 2 4 6
S4 S5 S6 S5 S6 S7 S5 S6 S7 S8 S5 S6 S7 S8 S9 S5 S6 S7 S8 S9
S4 S5 S6 S5 S6 S7 S5 S6 S7 S8 S5 S6 S7 S8 S9 S5 S6 S7 S8 S9
1672
0005 0000 0008 0018 0000 0009 0029 0429 0000 0001 0031 0231 4231 0000 0020 0420 6420

Figure 7. Waveform of “Testing customer code states (S5-S9)” in TB

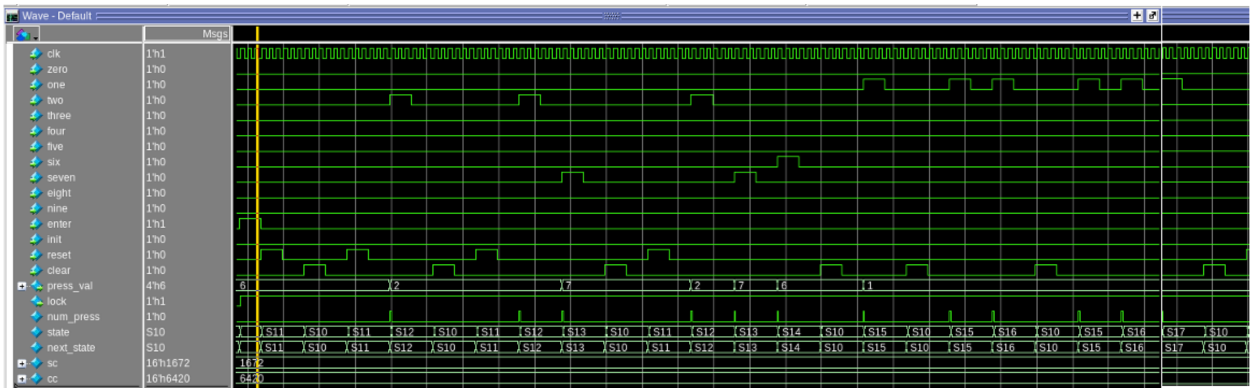


Figure 8. Waveform of "Testing resetting/counterfeit states (S10-S17)", "Clear tests" in TB

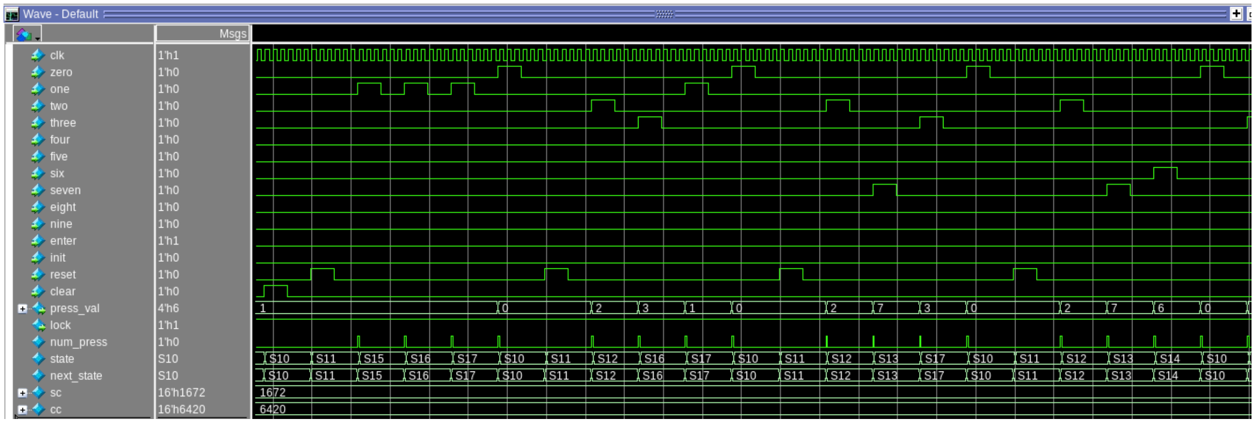


Figure 9. Waveform of "Testing resetting/counterfeit states (S10-S17)", "Security code fail tests" in TB

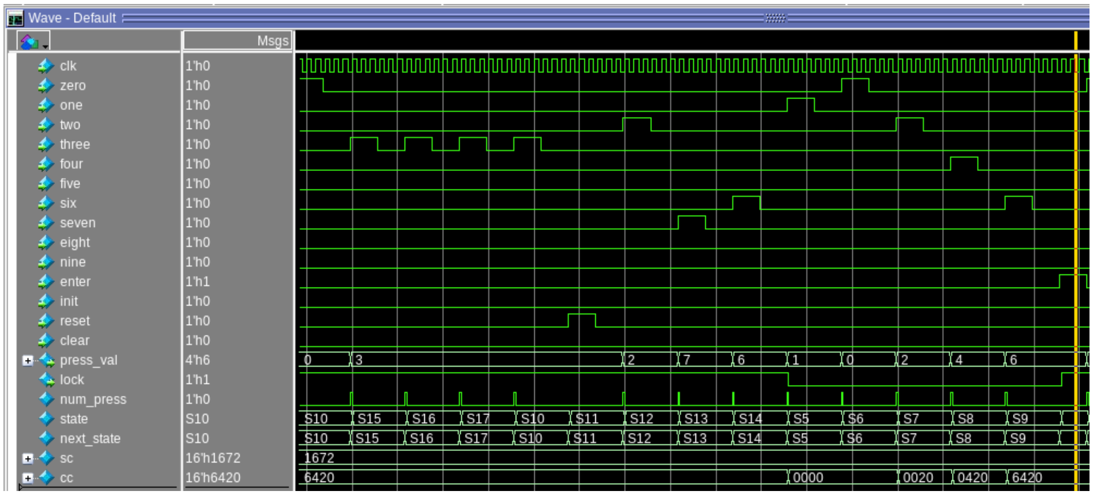
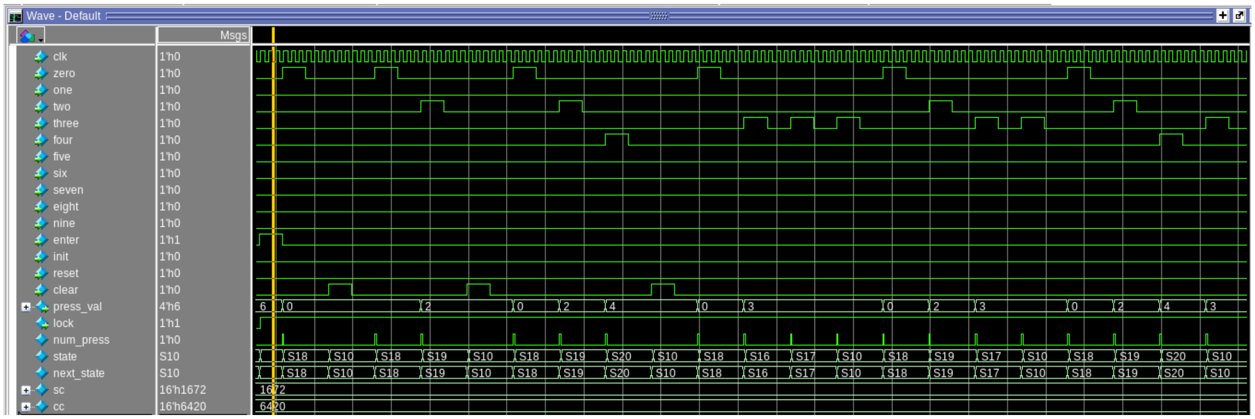


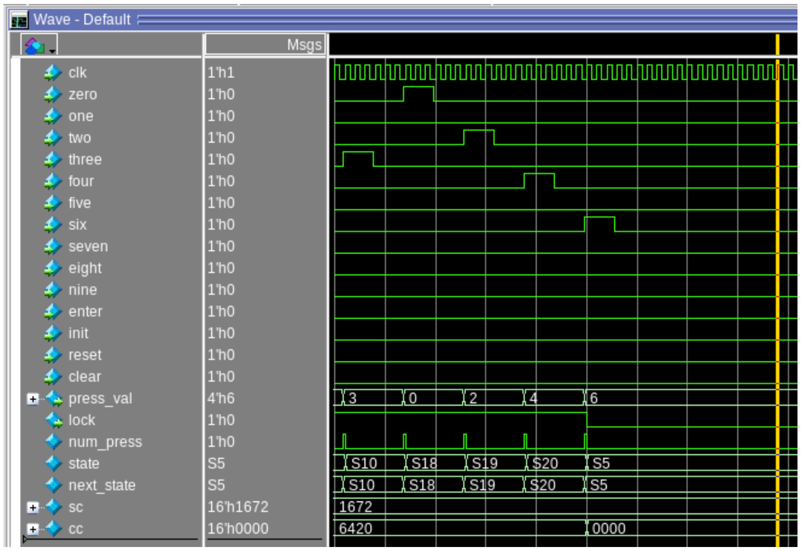
Figure 10. Waveform of "Testing resetting/counterfeit states (S10-S17)", "Fully counterfeit test" + "Reset test" + "Back to S10" with same cc" in TB



Wave - Default

	Msgs
clk	1'h0
zero	1'h0
one	1'h0
two	1'h0
three	1'h0
four	1'h0
five	1'h0
six	1'h0
seven	1'h0
eight	1'h0
nine	1'h0
enter	1'h1
init	1'h0
reset	1'h0
clear	1'h0
press_val	4'h6
lock	1'h1
num_press	1'h0
state	S10
next_state	S10
sc	16'h1672
cc	16'h6420

Figure 11. Waveform of “Testing unlocking states”, “Clear tests” + “Customer code fail tests” in TB



Wave - Default

	Msgs
clk	1'h1
zero	1'h0
one	1'h0
two	1'h0
three	1'h0
four	1'h0
five	1'h0
six	1'h0
seven	1'h0
eight	1'h0
nine	1'h0
enter	1'h0
init	1'h0
reset	1'h0
clear	1'h0
press_val	4'h6
lock	1'h0
num_press	1'h0
state	S5
next_state	S5
sc	16'h1672
cc	16'h0000

Figure 12. Waveform of “Testing unlocking states”, “Unlock test” in TB