Khoa Quach

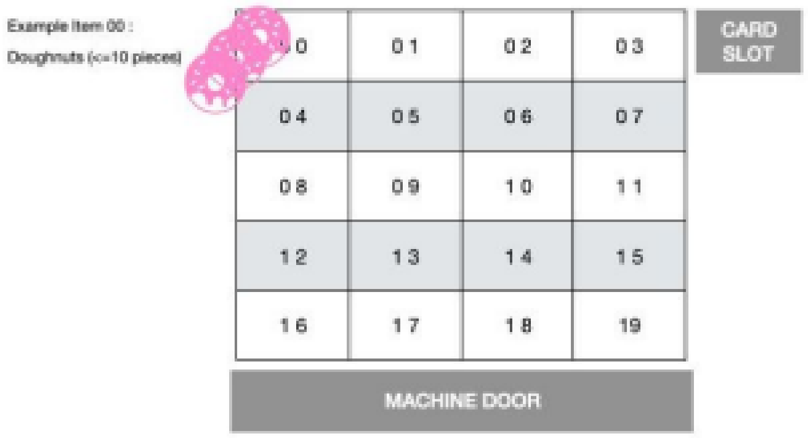
Vending Machine

1. <u>Introduction</u>

a. Background information

The project includes building a finite state machine to implement a vending machine in Verilog. Finite State Machines (FSMs)are used to model sequential circuits and are used in many low level systems. There are two types of FSMs: Mealy and Moore. A Mealy FSM is one where the output depends on the present state and input, while for a Moore FSM, the output only depends on only the present state.

b. Design requirements



A vending_machine.v Verilog module is made to implement this vending machine above.

The inputs of the vending_machine.v module are:

- CARD_IN: Customer inserts credit card in order to buy something
- VALID_TRAN: valid transaction and VEND selected item
- ITEM_CODE<3:0>: Each item in the vending machine is labeled. The labels start from 00 to 19. The codes are read one by one. There are 20 items to select from.
- KEY_PRESS: It must be one in order to read an item code press by the customer
- DOOR_OPEN: It simply indicates opening the door.
- RELOAD: Restock the number of items to ten.
- CLK: clock pulse at 10ns
- RESET: set all outputs and item counters to 0

The outputs of the vending_machine.v module are:

- VEND: Decrement counter of corresponding item by one (the stock of the product) (i.e., the item is dispensed) and set VEND to 1
- INVALID_SE L: Customer selected an invalid item selection number.
- FAILED_TRAN: The transaction failed.
- COST<2:0>: The cost of the selection chosen by the customer.

The descriptions above are just simple descriptions. There are still some edge cases to consider when designing the state machine. The module design is required to cover these edge cases:

INVALID_SEL is set to 1 only when only ITEM_CODE is entered and there is no second digit after 5 clock cycles, The second digit code is invalid, and the counter for one of the selection items is 0 (no more food/drink of that type).

If the door never goes low (the customer picks an item but it unfortunately gets stuck there), the door remains in that state unless a door close or reset happens which will remove it from that state.

VEND: Set to HIGH once the transaction is valid. Set to LOW once DOOR_OPEN goes high and then low , OR if the door does not open in 5 clock cycles.

After VEND, the machine goes to idle state.

In idle state, all outputs are set to zero except item counts. This differs from reset which also resets the item counts to zero.

CARD_IN can go low any time after the transaction begins , and the card's info. is stored by the machine when inserted.

The 'RELOAD' input is valid only when the machine is idle. CARD_IN is ignored if it goes high after reload becomes high. A new transaction can begin only once reload goes low.

FAILED_TRAN: Set to 1 if VALID_TRAN signal does not go high within 5 clock cycles of determining the ITEM_CODE

The prices of the item selections are as shown:

ITEM CODE	COST ($)	ITEM CODE	COST ($)
00, 01,02,03	1	12,13,14,15	4
04,05,06,07	2	16,17	5
08,09,10,11	3	18,19	6

2. Design Description

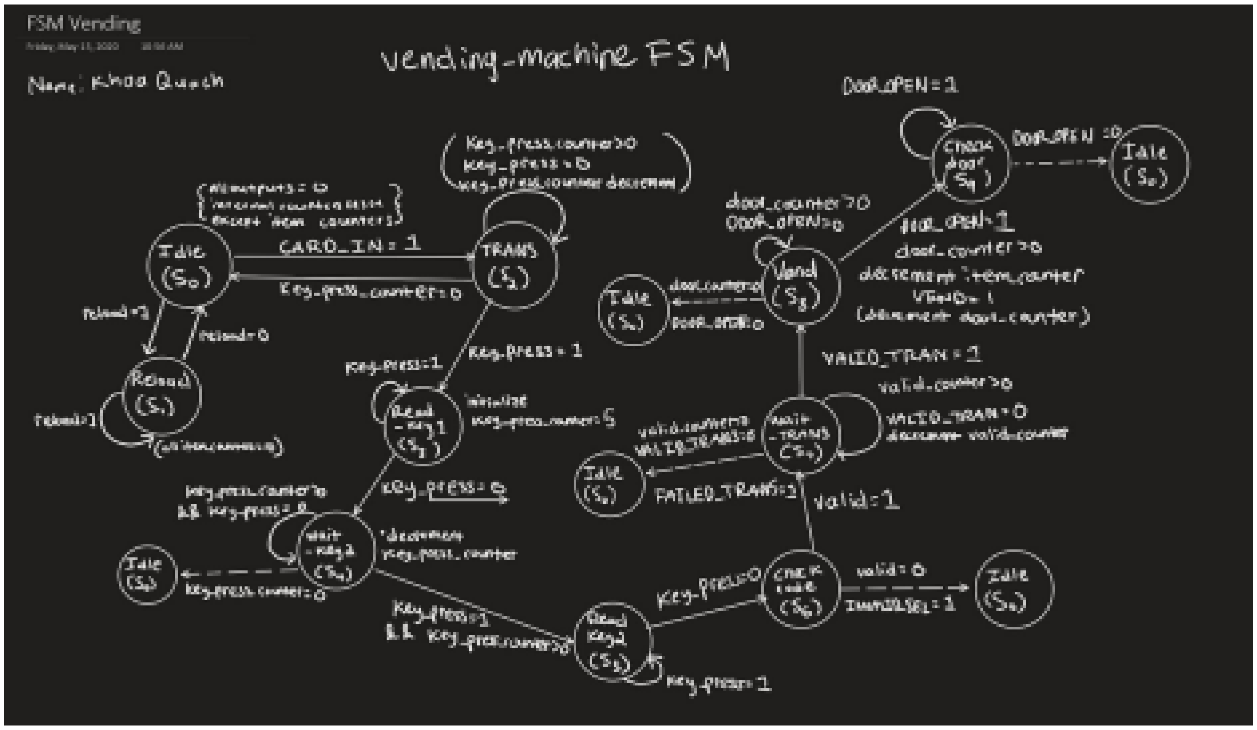


Figure 1: Finite State Machine of vending_machine.v

I have ten states in my design labeled by S with ascending indices:

S0: Idle (all outputs = 0 and internal counters are reset)

SI: Reload

S2: TRANS (Intermediate point where the user starts inputting the item numbers on the number pad)

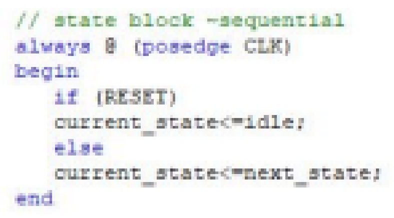
S3: Read key 1 (state of reading the first code inputted by the user)

S4: Wait key 2 (intermediate point between first code and second code input)

S5: Read key 2 (state of reading the second code inputted by the user)
// After this the item code insertion is done
S6: Check Code // (check if the item code is valid or not i.e out of range, out of stock) S7: Wait TRANS (where it considers VALID_TRANS and FAILED_TRANS) S8: VEND/vending (intermediate point of "vending" the item to the user to get S9: Check Door (Opening and closing of the door. User grabs the selection) There are internal counters such as key_press_counter and door_counter to implement the 5 clock cycle details described in the project manuscript.

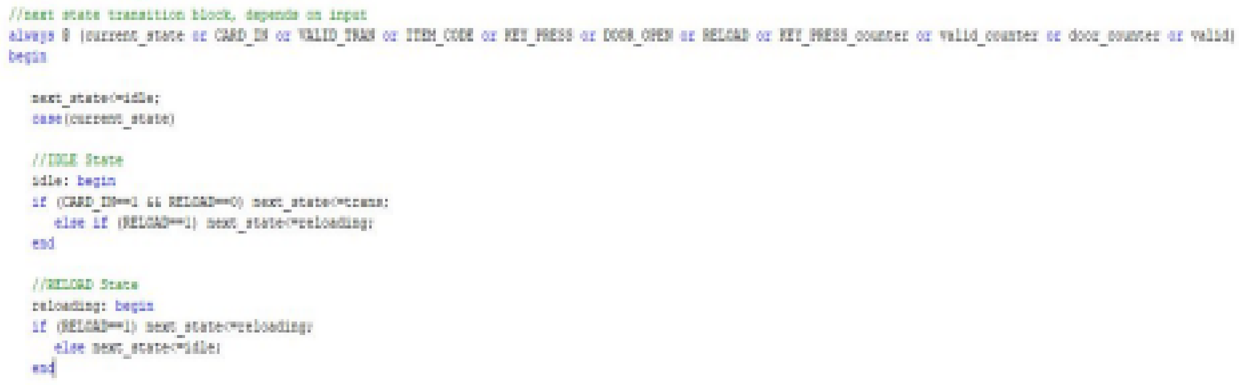
- S0 -> S1 if RELOAD = 1, S1-> SO if RELOAD = 0, It is important to know that in the always state sequential block that if RESET = 1, the current_state is set to idle, otherwise the current_state is set to its appropriate next_state.
- S1->S1 self loop if RELOAD = 1
- S0->S2 if CARD_IN = 1, S2 -> S0 if key_press_counter = 0
- S2->S2 if key_press_counter = 0, key_press = 0, and also implements decrementing of counter in Verilog implementation
- S2->S3 if key_press = 1
- S3->S3 self loop if key_press = 1. At S3, initialize key_press_counter = 5 clock pulses (to deal with validity)
- S3->S4 if key_press = 0
- At S4, decrement the key_press_counter
- S4->S4 self loop if key_press_counter > 0 && key_press = 0
- S4 -> S0 (idle) if key_press_counter = 0
- S4->S5 if key_press_counter > 0 && key_press = 1
- S5->S5 self loop if key_press = 1
- S5->S6 if key_press = 0
- S6->S0(idle) if valid = 0, INVALID_SEL = 1 (see three conditions in table for when INVALID_SEL = 1)
- S6->S7 if valid = 1
- S7->S7 self loop if VALID_TRAN = 0 and remember to decrement the valid_counter ● S7->S0(idle) if valid_counter = 0 setting VALID_TRANS to 0 and FAILED_TRANS to 1
- S7->S8 if VALID_TRAN = 1
- S8->S0 if door_counter = 0 and DOOR_OPEN = 0
- S8->S8 self loop if door_counter > 0 and DOOR_OPEN = 0
- S8->S9 if DOOR_OPEN = 1 and door_counter > 0. VEND = 1. Remember to decrement door_counter during state 8.
- S9->S9 self loop if DOOR_OPEN = 1
- S9->S0 if DOOR_OPEN = 0. This transition means success.

The following describes the code and how it used to implement the state transitions and fulfill the requirements of the system input and output active/low statements mentioned in the project manuscript:



```
// state block -sequential
always @ (posedge CLK)
begin
   if (RESET)
   current_state<=idle;
   else
   current_state<=next_state;
end
```

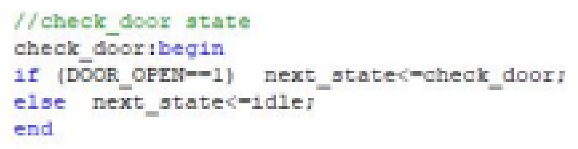
Like in the example code given in the project manuscript, I created an always block that updates the state. If RESET = 1, I set the current_state to idle, otherwise I set it to the next_state.



```
//next state transition block, depends on input
always @ (current_state or CARD_IN or VALID_TRAN or ITEM_CODE or KEY_PRESS or DOOR_OPEN or RELOAD or KEY_PRESS_counter or valid_counter or door_counter or valid)
begin

  next_state<=idle;
  case(current_state)

  //IDLE State
  idle: begin
  if (CARD_IN==1 && RELOAD==0) next_state<=trans;
    else if (RELOAD==1) next_state<=reloading;
  end

  //RELOAD State
  reloading: begin
  if (RELOAD==1) next_state<=reloading;
    else next_state<=idle;
  end
```

As this FSM depends on inputs, I have an always block that deals with the next state transitions. I put those inside a sensitivity list @(sensitivty list) to consider the next state.

I have a case block inside of this always block. Each case refers to a state described in my FSM diagram drawn above. For example, for the case of the current state being “idle”, if CARD_IN = 1 and RELOAD = 0 it goes to the TRANS state in my state diagram (S2).

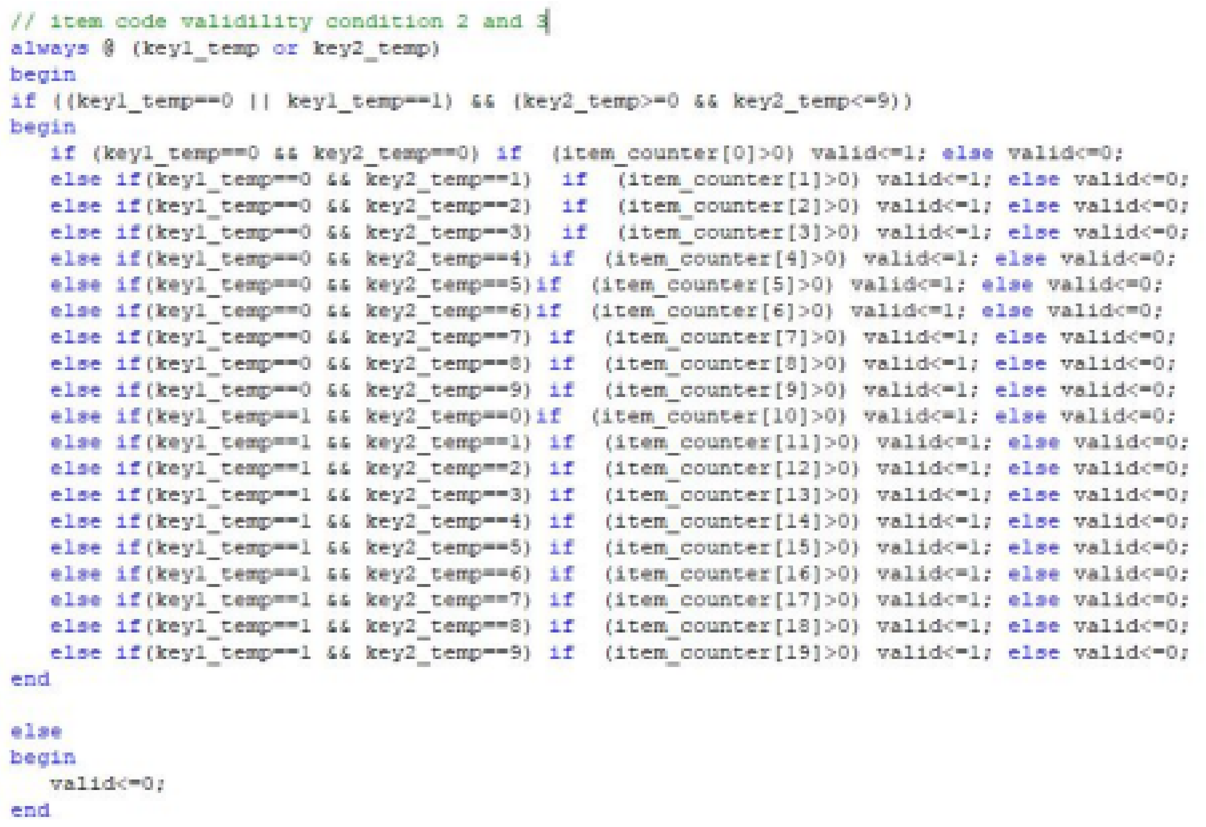


```
//check_door state
check_door:begin
if (DOOR_OPEN==1)  next_state<=check_door;
else  next_state<=idle;
end
```

Another example is in my last state “Check door” (S9). If the door never goes low - the machine should not do anything until the door is closed. It should just remain in that state. The only thing that can remove it from this state is a door close or a reset.

The rest of cases for current_state are built the same way in the case block, but the conditions vary following my FSM diagram with some minor extra details where needed to determine what the next_state should be.

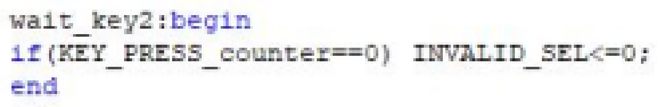
Now for the verification part of the design:



```
// item code validility condition 2 and 3
always @ (key1_temp or key2_temp)
begin
if ((key1_temp==0 || key1_temp==1) && (key2_temp>=0 && key2_temp<=9))
begin
   if (key1_temp==0 && key2_temp==0) if  (item_counter[0]>0) valid<=1; else valid<=0;
   else if(key1_temp==0 && key2_temp==1)  if  (item_counter[1]>0) valid<=1; else valid<=0;
   else if(key1_temp==0 && key2_temp==2)  if  (item_counter[2]>0) valid<=1; else valid<=0;
   else if(key1_temp==0 && key2_temp==3)  if  (item_counter[3]>0) valid<=1; else valid<=0;
   else if(key1_temp==0 && key2_temp==4) if  (item_counter[4]>0) valid<=1; else valid<=0;
   else if(key1_temp==0 && key2_temp==5)if  (item_counter[5]>0) valid<=1; else valid<=0;
   else if(key1_temp==0 && key2_temp==6)if  (item_counter[6]>0) valid<=1; else valid<=0;
   else if(key1_temp==0 && key2_temp==7) if  (item_counter[7]>0) valid<=1; else valid<=0;
   else if(key1_temp==0 && key2_temp==8) if  (item_counter[8]>0) valid<=1; else valid<=0;
   else if(key1_temp==0 && key2_temp==9) if  (item_counter[9]>0) valid<=1; else valid<=0;
   else if(key1_temp==1 && key2_temp==0)if  (item_counter[10]>0) valid<=1; else valid<=0;
   else if(key1_temp==1 && key2_temp==1) if  (item_counter[11]>0) valid<=1; else valid<=0;
   else if(key1_temp==1 && key2_temp==2) if  (item_counter[12]>0) valid<=1; else valid<=0;
   else if(key1_temp==1 && key2_temp==3) if  (item_counter[13]>0) valid<=1; else valid<=0;
   else if(key1_temp==1 && key2_temp==4) if  (item_counter[14]>0) valid<=1; else valid<=0;
   else if(key1_temp==1 && key2_temp==5) if  (item_counter[15]>0) valid<=1; else valid<=0;
   else if(key1_temp==1 && key2_temp==6) if  (item_counter[16]>0) valid<=1; else valid<=0;
   else if(key1_temp==1 && key2_temp==7) if  (item_counter[17]>0) valid<=1; else valid<=0;
   else if(key1_temp==1 && key2_temp==8) if  (item_counter[18]>0) valid<=1; else valid<=0;
   else if(key1_temp==1 && key2_temp==9) if  (item_counter[19]>0) valid<=1; else valid<=0;
end

else
begin
   valid<=0;
end
```

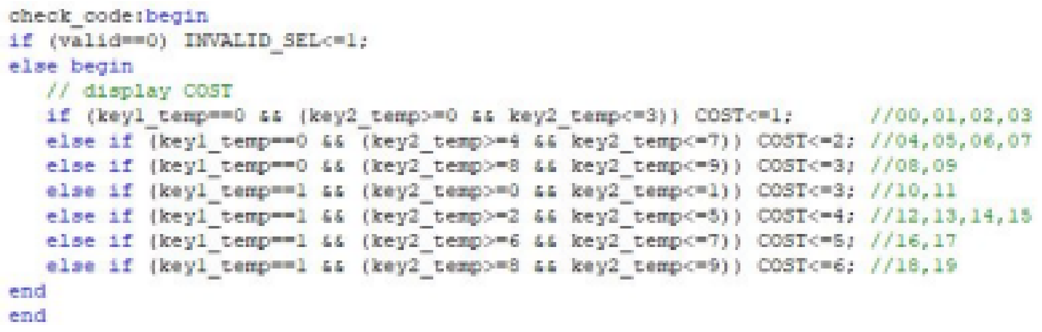
At the check_code state (S6), it checks if the item code input is between 00-19. It also checks if that item is out of stock or still has some left. If it is out of stock, valid is set to 0.

“valid” is also used in the output block of my code:



```
wait_key2:begin
if(KEY_PRESS_counter==0) INVALID_SEL<=0;
end
```

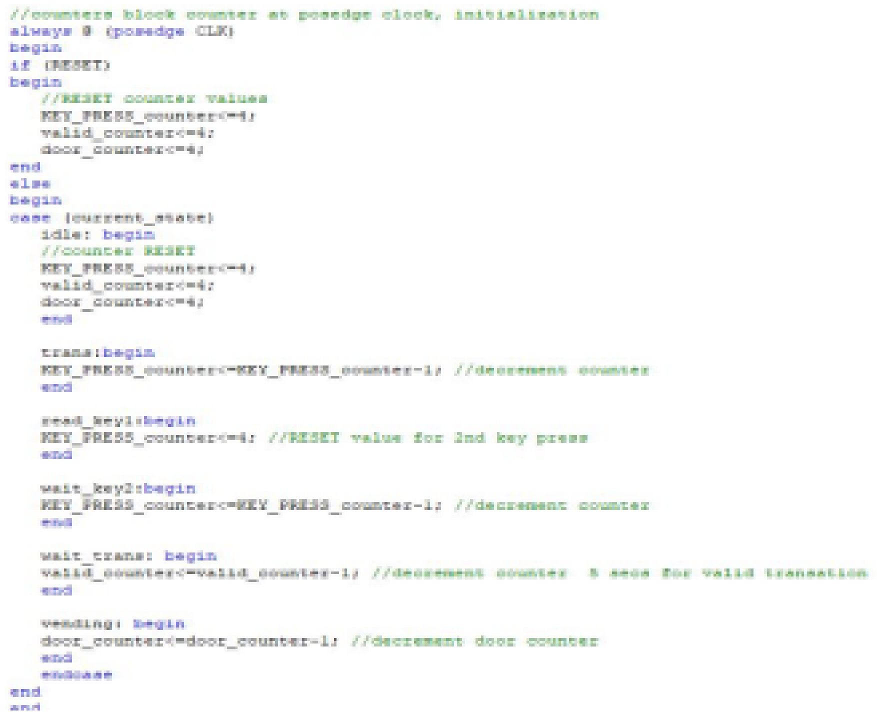
In addition, if the second ITEM_CODE input digit does not come before the 5 clock cycles after from the first ITEM_CODE input digit, then INVALID_SEL is set to high.



```
check_code:begin
if (valid==0) INVALID_SEL<=1;
else begin
   // display COST
   if (key1_temp==0 && (key2_temp>=0 && key2_temp<=3)) COST<=1;      //00,01,02,03
   else if (key1_temp==0 && (key2_temp>=4 && key2_temp<=7)) COST<=2; //04,05,06,07
   else if (key1_temp==0 && (key2_temp>=8 && key2_temp<=9)) COST<=3; //08,09
   else if (key1_temp==1 && (key2_temp>=0 && key2_temp<=1)) COST<=3; //10,11
   else if (key1_temp==1 && (key2_temp>=2 && key2_temp<=5)) COST<=4; //12,13,14,15
   else if (key1_temp==1 && (key2_temp>=6 && key2_temp<=7)) COST<=5; //16,17
   else if (key1_temp==1 && (key2_temp>=8 && key2_temp<=9)) COST<=6; //18,19
end
end
```

If valid is equal to 0, I set INVALID_SEL equal to 1. Otherwise, display the cost.

Another section of my code is an always(posedge CLK) deals with the internal counters I created to implement the 5 clock cycle check requirement for validity:



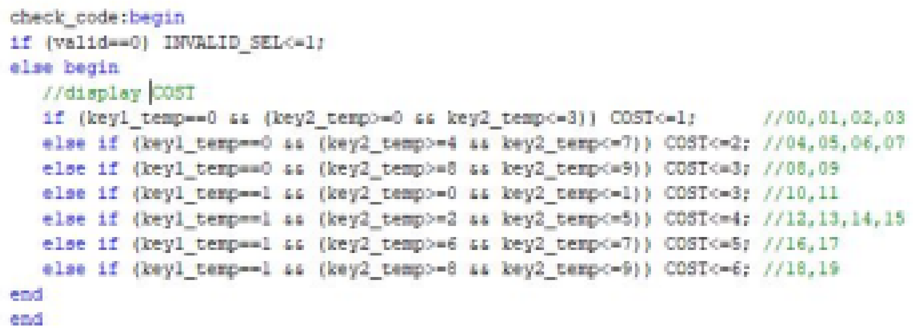
```
//counters block counter at posedge clock, initialization
always @ (posedge CLK)
begin
if (RESET)
begin
   //RESET counter values
   KEY_PRESS_counter<=4;
   valid_counter<=4;
   door_counter<=4;
end
else
begin
case (current_state)
   idle: begin
   //counter RESET
   KEY_PRESS_counter<=4;
   valid_counter<=4;
   door_counter<=4;
   end

   trans:begin
   KEY_PRESS_counter<=KEY_PRESS_counter-1; //decrement counter
   end

   read_key1:begin
   KEY_PRESS_counter<=4; //RESET value for 2nd key press
   end

   wait_key2:begin
   KEY_PRESS_counter<=KEY_PRESS_counter-1; //decrement counter
   end

   wait_trans: begin
   valid_counter<=valid_counter-1; //decrement counter  5 secs for valid transation
   end

   vending: begin
   door_counter<=door_counter-1; //decrement door counter
   end
   endcase
end
end
```

If RESET is high, all internal counters (key_press_counter, door_counter, valid_counter) are reset to 4 (0 -4 = 5 clock cycles). This also applies to when reset is 0, but it is at the idle(S0) state. As you can see in the code, I decrement the counters by 1 at the corresponding states where it should be decremented. I reset the key_press_counter in expectation for the second input item code digit by the customer.

Finally, similar to the turnstile example, I have the output section of my code at the end.

When reset = 1, all outputs and item counters are set to zero. However, while reset = 0 and the system is or goes to idle(S0) state, only the outputs are set to zero.

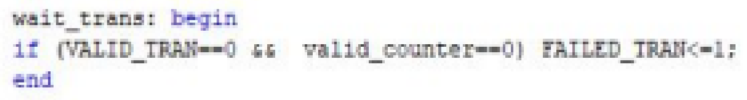
This block also covers the “4. Transact” part in the project manuscript that deals with VALID_TRAN and the price of items



```
check_code:begin
if (valid==0) INVALID_SEL<=1;
else begin
   //display COST
   if (key1_temp==0 && (key2_temp>=0 && key2_temp<=3)) COST<=1;      //00,01,02,03
   else if (key1_temp==0 && (key2_temp>=4 && key2_temp<=7)) COST<=2; //04,05,06,07
   else if (key1_temp==0 && (key2_temp>=8 && key2_temp<=9)) COST<=3; //08,09
   else if (key1_temp==1 && (key2_temp>=0 && key2_temp<=1)) COST<=3; //10,11
   else if (key1_temp==1 && (key2_temp>=2 && key2_temp<=5)) COST<=4; //12,13,14,15
   else if (key1_temp==1 && (key2_temp>=6 && key2_temp<=7)) COST<=5; //16,17
   else if (key1_temp==1 && (key2_temp>=8 && key2_temp<=9)) COST<=6; //18,19
end
end
```

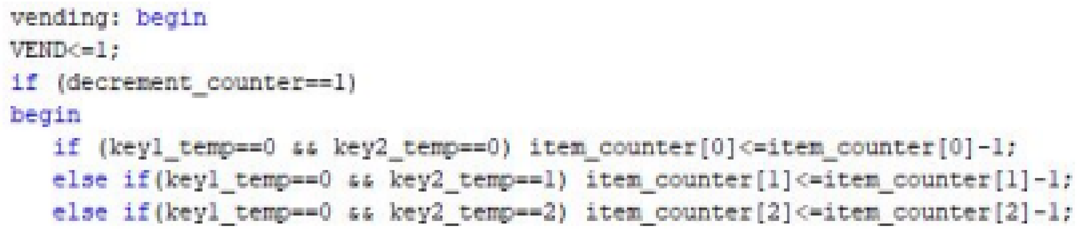
The prices of the specific items are followed by the table given in the project manuscript. The <= and >= operators were used to decrease code lines since sub-ranges of item codes have the same price.

Following the FSM diagram for S7 to S0 described above to deal with FAILED_TRAN:



```
wait_trans: begin
if (VALID_TRAN==0 &&  valid_counter==0) FAILED_TRAN<=1;
end
```

The following code snippet shows I implemented decreasing the item count of the respective items if chosen by the customer:



```
vending: begin
VEND<=1;
if (decrement_counter==1)
begin
   if (key1_temp==0 && key2_temp==0) item_counter[0]<=item_counter[0]-1;
   else if(key1_temp==0 && key2_temp==1) item_counter[1]<=item_counter[1]-1;
   else if(key1_temp==0 && key2_temp==2) item_counter[2]<=item_counter[2]-1;
```

....continued

RTL Schematic:

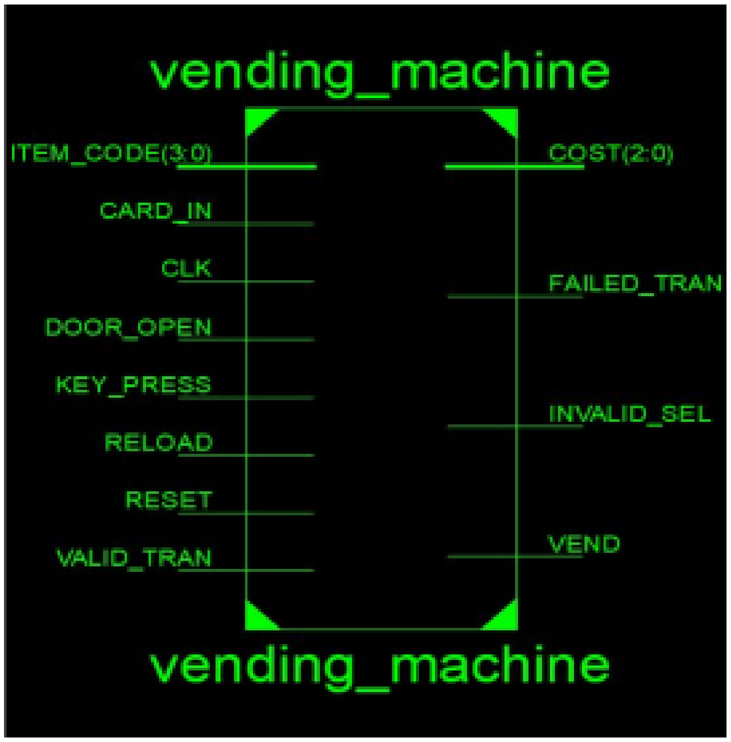


Figure 2: High-level block schematic of vending_machine generated by Xilinx ISE.

As described in the finite state machine analysis, the inputs are shown on the left of the block diagram. The outputs of the module are on the right of the block diagram. The module definition is written as :

module vending_machine(CARD_IN,VALID_TRAN,ITEM_CODE,KEY_PRESS,DOOR_OPEN,RELOAD,CLK,RESET, VEND,INVALID_SEL,FAILED_TRAN,COST);

The deeper level of this schematic is very large and there's not enough space to fit the whole diagram as a screenshot. It consists of lots of D flip flops to implement counters, and lots of comparators and logic gates/multiplexers for the if statements and for the counter manipulation such as decrementing.

3. Simulation Documentation

My testbench is categorized into two sections: a normal case with a successful run and a set of special cases.

The test cases with explanations here are done with the input at the positive edge of the clk. The negative edge input testbench screenshots are shown here:

https://drive.google.com/open?id=1AKcK1MMrLTIs0EWjVDRuokNmPkD2bEB0gUEHODDMpoM

The **normal test case** is with the item selected being 11. This means the cost should be 3 dollars.

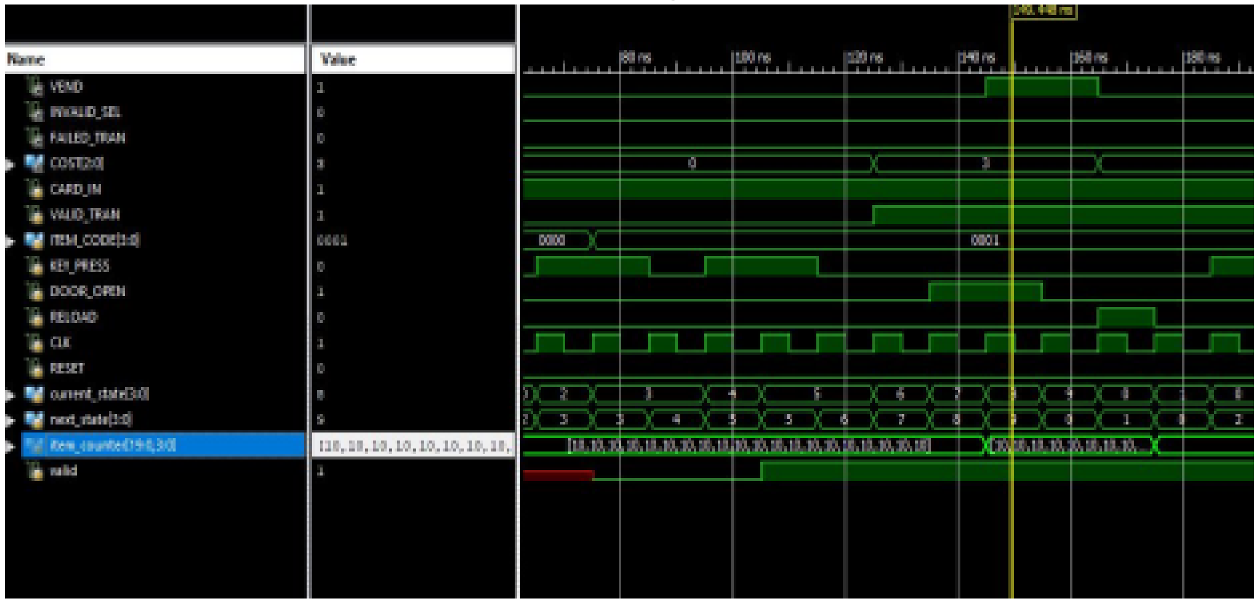


Figure 3: Simulation of normal test case of item 11.
CARD_IN is set to 1 as the customer is trying to purchase an item. You can also observe the KEY_PRESS going high and low depicting the customer's inputs of the first and second ITEM_CODE digit. As you can see, the COST is set to 3 dollars. VEND is set to 1. DOOR_OPEN should be 1 as it is the input. It is indeed a VALID_TRAN, meaning FAILED_TRAN should be low/0. Also looking at the item_counter array, the selection number (11) which initially had a stock amount of 10 has decreased to a stock size of 9. After DOOR_OPEN goes to zero, the system goes from check_door(S9) back to Idle (S0).

Special cases:

Figure 4: case when RELOAD and CARD_IN are both set to 1.

The system should not go to the transaction state covering FAQ question 5. As you can see, the system remains in the reloading state (S1) and goes back to the Idle state (S0) after RELOAD goes low.

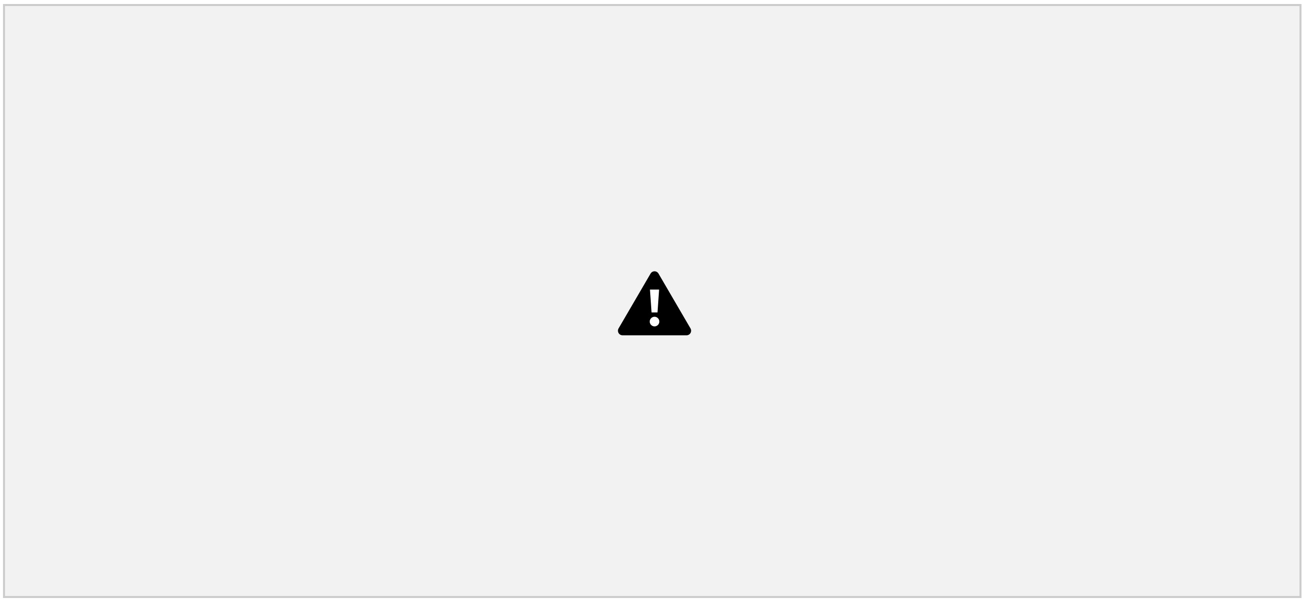


Figure 5: case when VALID_TRANS is not high within 5 clock cycles

In such a case, the system should go back to idle. As you can see in the screenshot, my system stays in wait_trans state (S7) for 5 clock cycles and then goes to Idle(S0). In addition, FAILED_TRAN is set to 1 matching with the requirements in the project manuscript “Transact” section and its condition in the I/O table.

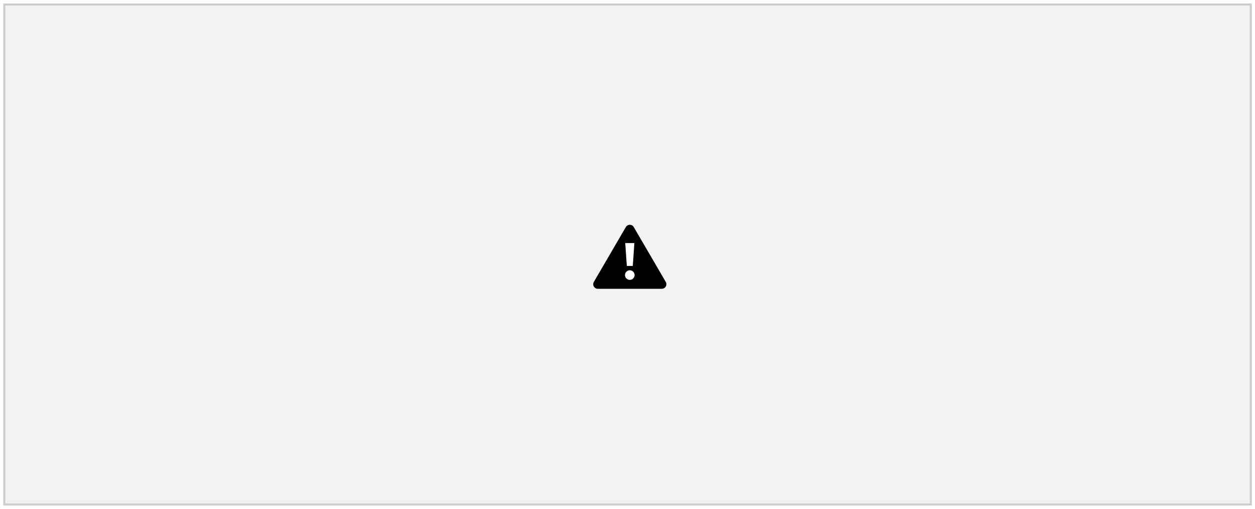


Figure 6: case when there is no first key input after 5 clock cycles.

There is no user input in 5 clock cycles. As you can see the system remains in TRANS state (S2) for 5 clock cycles and returns to Idle state(S0).

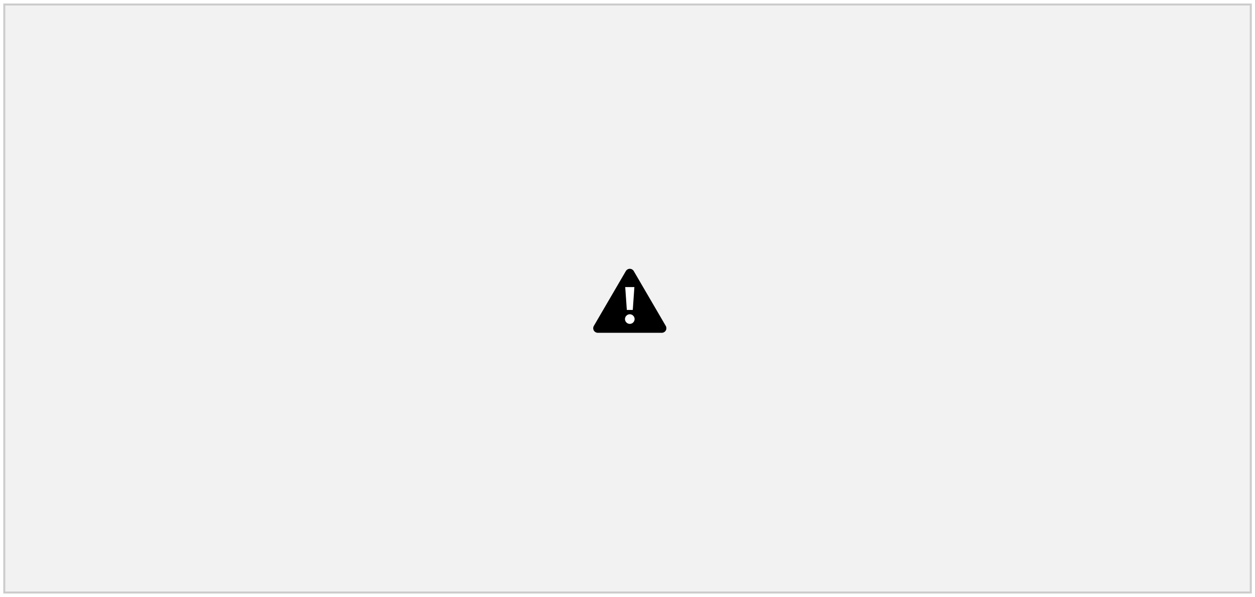


Figure 7: case when there is no 2nd key input after 5 clock cycles.
Like my FSM diagram design, as you can see, the system stays at wait_key2 state (S4) for 5 clock cycles because no second input was pressed by the customer. Afterward, it goes to idle state (S0).

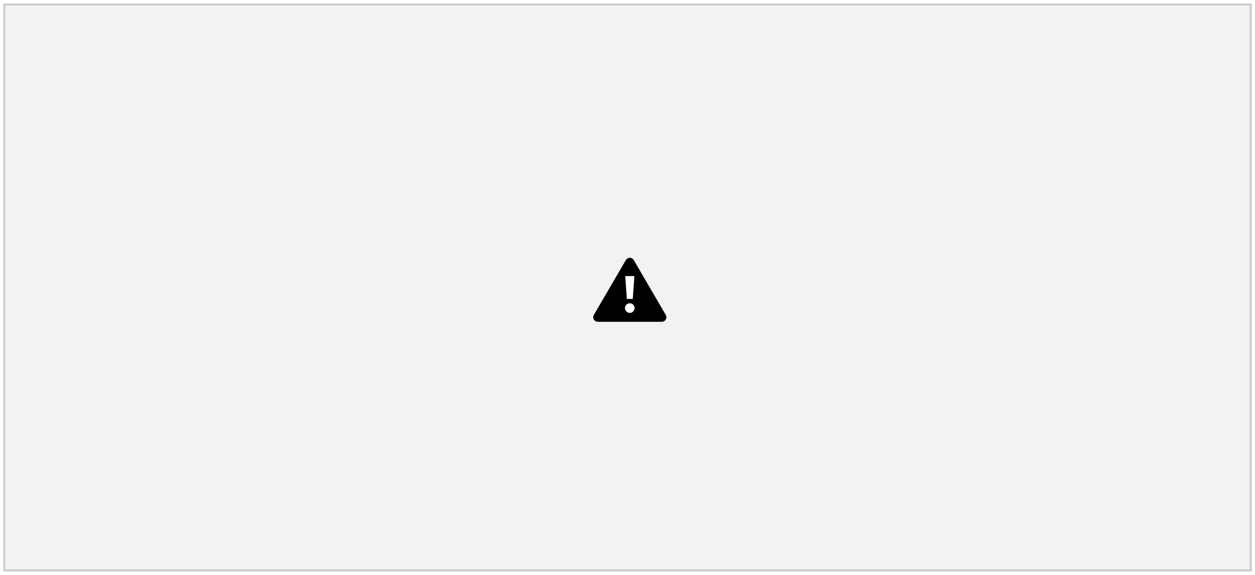


Figure 8: Case where door is not open. Vending state
Please refer to my FSM diagram for the S8 transitions. If door_count > 0 and DOOR_OPEN = 0, S8 self loops to itself S8. If door_counter = 0 and DOOR_OPEN = 0, then S8 goes S0 (Idle). My system stays at the vending state (S8) and goes to Idle(S0) when DOOR_OPEN is equal to 1.

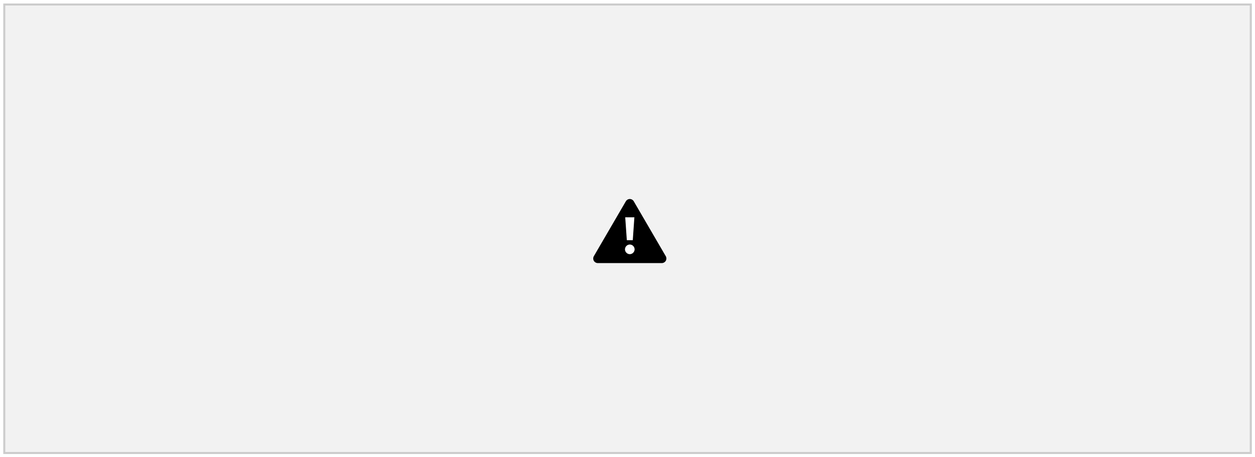


Figure 9: Case when invalid input i.e not in the range 00-19.
In this case, the input was 21. This is an invalid input. This means INVALID_SEL should be set to 1 which it does in the simulation fulfilling condition one for INVALID_SEL in the manuscript table. Also the system goes from state 6 back to idle S0.

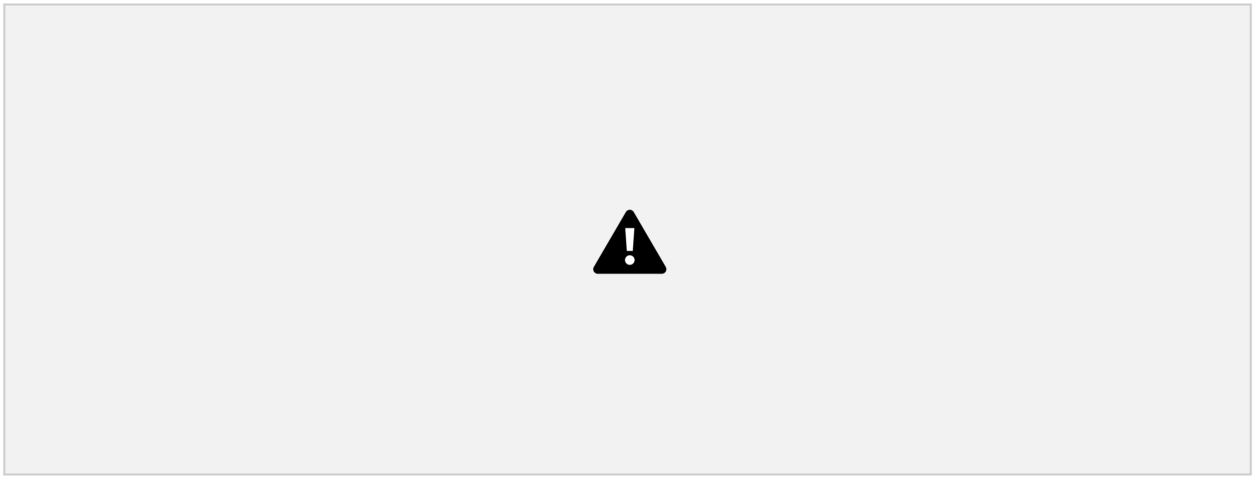


Figure 10: case when no items left in the machine or out of stock for the item selected.

For my testbench, I just set reset to high to set all the item counts to 0. Therefore, regardless of the item code selected, there won't be any left in the vending machine. Because of this, INVALID_SEL should become 1 as shown in the simulation fulfilling condition 3 for INVALID_SEL in the manuscript table. Also the system goes from state 6 back to idle (S0).

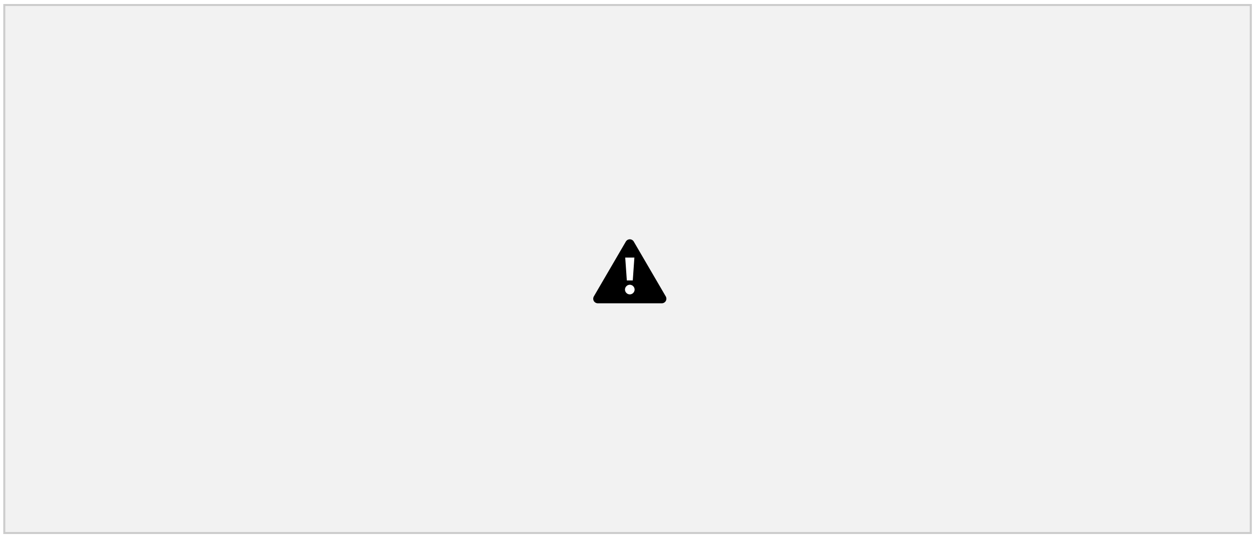


Figure 11: Case when door never goes low. FAQ Question 1

The solution to this FAQ was that if the door never goes low, the machine should just remain in that state. The only thing that can remove it from that state is a door close or a rest. As you can see in the screen shot, DOOR_OPEN is 1 for a while and following the required behavior, the machine remains in that state (S9). The door does not close for like 20 cycles. Afterward, reset becomes 1 and it returns to idle state (S0). In my test bench, I could have made it so it remained in S9 always or put DOOR_OPEN = 0 to change the state instead.

Synthesis Report (Design Summary)

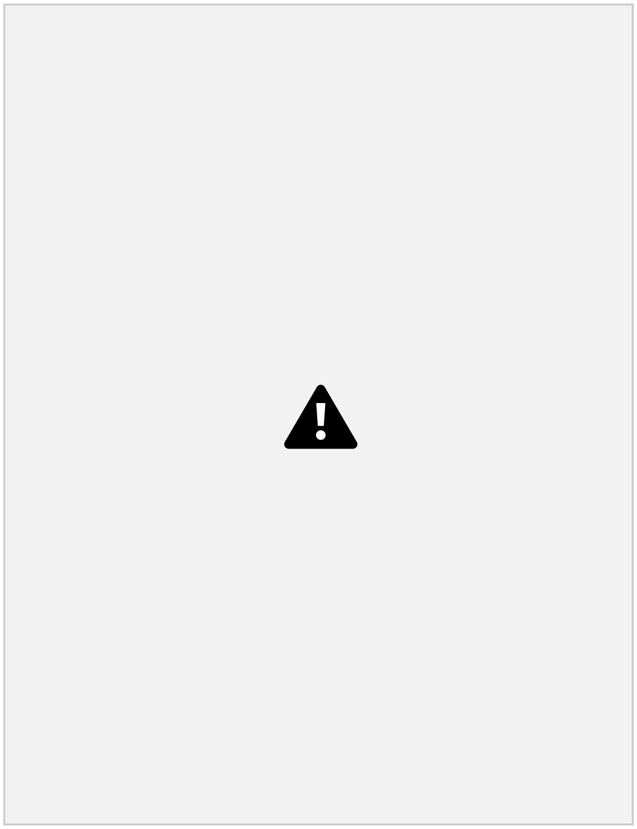


Figure 12: Design Summary section of Synthesis Report and Summary section of Implementation Map. (full files can be found in code files zip folder)

From what I can observe, lots of Flip flops and latches are used to implement this vending machine. I assume it is because of the transition states and counters. I assume the buffers are used to deal with the intermediate points between the states such as the user inputs.

4. <u>Conclusion</u>

The design includes the input and outputs required by the manuscript with additional internal counters to cover the 5 clock cycle detail. The parameter keyword is used to identify states. There is an always block running on the positive edge of the clock to update the state. I have another always block with an embedded case block that determines the next state transitions from one state to another depending on certain conditions coinciding with my finite state machine diagram drawn. I have another always block that runs on the key inputs that checks if the item code is valid (in the range 00-19 and the item selected is in stock). Another always block that runs on the positive edge of the input clock is used to update the internal counters key_press_counter, valid_counter, and door_counter. The rest of the code is for the outputs , displaying the cost, and decrementing the item count of an item when chosen and successfully outputted to the customer. Some difficulties I had were following the testbench I had created. I solved this issue by commenting in my test bench what section checks for what and counting the amount of delay I had used to know where to look for the cases in the waveforms displayed. Another problem I had was organizing my code for a Finite State Machine type problem. I had to look up more complicated examples in addition to the turnstile to get a grasp of organizing the code efficiently (i.e state block and output block).