# Lab 3 Report

CS M152A, Spring 2021

Caleb Lee

---

## Introduction

Lab 3 was about designing a finite state machine (FSM) that behaves like a vending machine. Verilog was used to code and no FPGA was used. FSM has multiple states that it can transition from one another depending on its current state and input. The vending machine that was requested does 6 actions: reset, idle, reload, transact, vend, and get code.

1. Reset: it sets all the output to 0 when the signal RESET is 1.
2. Idle: machine waits for the transaction to begin. It begins transaction when CARD_IN becomes 1 or reloads the machine when RELOAD becomes 1. This is the initial state of the FSM.
3. Reload: The machine has 10 total items, and there are 10 units of the items stored in 1 slot. Reloading would store all items back to 10.
4. Get code: when the machine receives CARD_IN = 1 during the idle state, it waits for the selection of the item. IEM_CODE<2:0> is used to store the 2 digit item code sequentially. The machine will consider the input to be invalid if there are more than 5 clock cycles passed while waiting for the input for the first and second digit. Once it receives the input, it will continue onto the transaction after checking if the input was valid.
5. Transact: Once it receives VALID_TRAN = 1 within the 5 clock cycle, it vends the item. If not, the machine goes back to idle while FAILED_TRAN = 1.
6. Vend: it decrements the unit of the selected item by 1 in the stack then returns to idle upon completion

The range of the item code is 10 to 14 and 20 to 24; the cost of item 10 to 14 is 2 while the cost of 20 to 24 is 5. In the module, there will be inputs (CLK, RESET, RELOAD, CARD_IN, ITEM_CODE<2:0>, KEY_PRESS, VALID_TRAN) and outputs (VEND, INVALID_SEL, COST<2:0>, FAILED_TRAN)

## Design

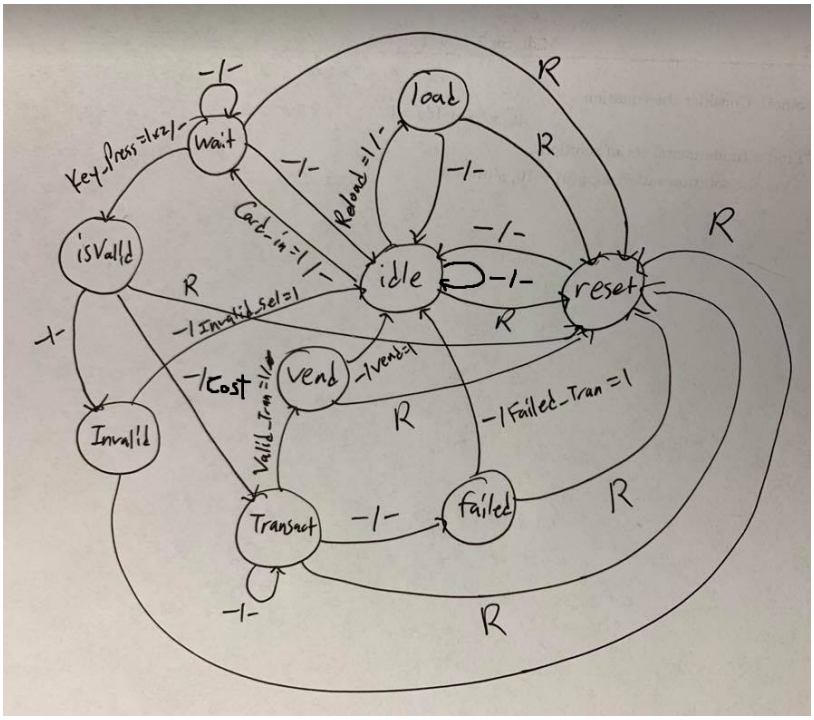


**Figure 1:** free state diagram

The above is the free diagram that was used for my code. The following states are:

```
//define states
parameter reset = 4'b0000;
parameter idle = 4'b0001;
parameter load = 4'b0010;
parameter transact = 4'b0011;
parameter invalid = 4'b0100;
parameter failed = 4'b0101;
parameter vend = 4'b0110;
parameter wait_ = 4'b0111;
parameter isValid = 4'b1000;
```

**Figure 2:** List of states

The following format of x/y is x (input) and y (output). If the input is -, that means there is no input required, but if the output is -, that means the outputs are default (all output is 0). On the transition from isValid state to Transact state, the output is COST without any value assigned. This is due to the fact that the cost changes depending on the user's choice of selection. The free state diagram will be explained throughout the explanation of my code and my design. The below picture will be the variables used for conditions in transitioning to different states. All of the registers that were count followed by a number (i.e. coun10) is the storage of a unit of items in a stack. The module will be the same as defined in the instruction.

```
module vending_machine(input CLK, input RESET, input RELOAD, input CARD_IN, input [2:0] ITEM_CODE, input KEY_PRESS, input VALID_TRAN,
                       output reg VEND, output reg INVALID_SEL, output reg [2:0] COST, output reg FAILED_TRAN);
```

**Figure 3:** Module definition

```
reg [3:0] current;
reg [3:0] next;

reg[2:0] tens;
reg ten_input;
reg[2:0] ones;
reg [2:0] count = 3'b000;
reg count_start = 0;
reg [2:0] second_count = 3'b000;
reg second_count_start = 0;
reg [2:0] trans_count = 3'b000;
reg trans_count_start = 0;


reg [3:0] count10;
reg [3:0] count11;
reg [3:0] count12;
reg [3:0] count13;
reg [3:0] count14;
reg [3:0] count20;
reg [3:0] count21;
reg [3:0] count22;
reg [3:0] count23;
reg [3:0] count24;
```

**Figure 4:** List of extra variables

Transitioning in states depends on the code below. Every state queue up one of the states to the 'next' variable, and that becomes current in the next positive edge clock.

```
always@(posedge CLK)
begin
   if(RESET)
      current <= reset;
   else
      current <= next;
end
```

**Figure 5:** state update

The definition and input/output of the states will be explained below with bullet points followed by diagrams to support the explanation.

- Reset: State enters reset once RESET = 1. At reset, all the outputs will be set to 0 and all the count registers will be 0. Reset goes to the idle state once RESET = 0.
-

```
if(current == reset) begin
   count10 <= 4'b0000;
   count11 <= 4'b0000;
   count12 <= 4'b0000;
   count13 <= 4'b0000;
   count14 <= 4'b0000;
   count20 <= 4'b0000;
   count21 <= 4'b0000;
   count22 <= 4'b0000;
   count23 <= 4'b0000;
   count24 <= 4'b0000;
end
```

```
reset: begin next = idle;  end
```

```
always @(*) begin
   case(current)
      reset,
      idle,
      load,
      wait_,
      isValid: begin
         VEND = 0;
         INVALID_SEL = 0;
         COST = 3'b000;
         FAILED_TRAN = 0;
      end
```

**Figure 6:** Code related to reset

- Idle: Idle is a rest state. Idle differs from reset because it does not set the count registers to be 0. However, it does set all outputs to be 0. As shown in the free state diagram, without any inputs, idle stays as idle. If the input is RELOAD = 1, then it queues up the next state to be load. If the input is CARD_IN = 1, then the next state will be wait_.

```
idle: begin
   count_start = 0;
   second_count_start = 0;
   ten_input = 0;
   trans_count_start = 0;
   if(RELOAD)
      next = load;
   else if(CARD_IN)
      next = wait_;
   else
      next = idle;
   end
```

```
always @(*) begin
   case(current)
      reset,
      idle,
      load,
      wait_,
      isValid: begin
         VEND = 0;
         INVALID_SEL = 0;
         COST = 3'b000;
         FAILED_TRAN = 0;
      end
```

**Figure 7:** Code related to idle

- Load: It loads up the items with a quantity of 10. After one clock cycle, it will return back to idle. The load state can only be activated if the previous state was idle.

```
else if(current == load) begin
   count10 <= 4'b1010;
   count11 <= 4'b1010;
   count12 <= 4'b1010;
   count13 <= 4'b1010;
   count14 <= 4'b1010;
   count20 <= 4'b1010;
   count21 <= 4'b1010;
   count22 <= 4'b1010;
   count23 <= 4'b1010;
   count24 <= 4'b1010;
end
```

```
load: begin next = idle; end
```

```
always @(*) begin
   case(current)
      reset,
      idle,
      load,
      wait_,
      isValid: begin
         VEND = 0;
         INVALID_SEL = 0;
         COST = 3'b000;
         FAILED_TRAN = 0;
      end
```

**Figure 8:** Code related to load

- Wait_: This state waits for the user input. Thus, it waits for KEY_PRESS = 1 then it receives the ITEM_CODE. For the user to receive full input, it needs to wait for KEY_PRESS = 1 for the second time. If the user takes too long (5 clock cycles), then the machine will return to idle. If the user successfully inputs both numbers in time, the next

state will be isValid. In order to count the clock pulse, I use the variable counter and count start to notify when to start counting.

```
wait_: begin
   count_start = 1;
   if(count >= 3'b100 || second_count >= 3'b100)
   begin
      next = idle;
   end
   else if(KEY_PRESS && ~ten_input) begin
      tens = ITEM_CODE;
      ten_input = 1;
      next = wait_;
      count_start = 0;
      second_count_start = 1;
   end
   else if(KEY_PRESS && ten_input) begin
      ones = ITEM_CODE;
      next = isValid;
   end
   else
      next = wait_;
end
```

```
always @(*) begin
   case(current)
      reset,
      idle,
      load,
      wait_,
      isValid: begin
         VEND = 0;
         INVALID_SEL = 0;
         COST = 3'b000;
         FAILED_TRAN = 0;
      end
```

```
 always@(posedge CLK) begin
    if(count_start)
       count <= count + 1'b1;
    else
       count <= 0;

    if(trans_count_start)
       trans_count <= trans_count + 1'b1;
    else
       trans_count <= 0;

    if(second_count_start)
       second_count <= second_count + 1'b1;
    else
       second_count <= 0;
 end
```

**Figure 9:** Code related to wait_ and clock pulse counter

- isValid: this state will check if the input received from the user is in the valid range (10-14 and 20-24). If it is, then the next state will be transact. If not, then it will go to an invalid state.

```
isValid: begin
     if((tens == 3'b001 && ones == 3'b000 && count10 > 4'b0000) ||
           (tens == 3'b001 && ones == 3'b001 && count11 > 4'b0000) ||
           (tens == 3'b001 && ones == 3'b010 && count12 > 4'b0000) ||
           (tens == 3'b001 && ones == 3'b011 && count13 > 4'b0000) ||
           (tens == 3'b001 && ones == 3'b100 && count14 > 4'b0000) ||
           (tens == 3'b010 && ones == 3'b000 && count20 > 4'b0000) ||
           (tens == 3'b010 && ones == 3'b001 && count21 > 4'b0000) ||
           (tens == 3'b010 && ones == 3'b010 && count22 > 4'b0000) ||
           (tens == 3'b010 && ones == 3'b011 && count23 > 4'b0000) ||
           (tens == 3'b010 && ones == 3'b100 && count24 > 4'b0000))
           begin
              next = transact;
           end
        else
           next = invalid;
end
```

```
always @(*) begin
   case(current)
      reset,
      idle,
      load,
      wait_,
      isValid: begin
         VEND = 0;
         INVALID_SEL = 0;
         COST = 3'b000;
         FAILED_TRAN = 0;
      end
```

**Figure 10:** Code related to isValid

- Invalid: this state will output INVALID_SEL = 1 for one clock pulse and then return to idle.

```
invalid: begin
   next = idle;
end
```

```
invalid: begin
   INVALID_SEL = 1;
end
```

**Figure 11:** Code related to invalid

- Transact: this state will wait for VALID_TRAN input to be 1 for 5 clock pulses. If VALID_TRAN does not go high within the 5 clock pulses, the next state becomes failed. If it does, then the next state will be vend. Transact uses trans_count and trans_count_start to notify when to begin to start counting the clock pulses. When the current state is transact, it will display the cost of the user-selected item until the transaction ends. Items from 10 to 14 cost 2 while 20 to 24 cost 5.

```
transact: begin
   trans_count_start = 1;
   if(trans_count >= 3'b100) begin
      next = failed;
      end
   else if(VALID_TRAN) begin
      next = vend;
      end
   else
      next = transact;
end
```

```
transact: begin
   if(tens == 3'b001)
      COST = 3'b010;
   else
      COST = 3'b101;
end
```

```
always@(posedge CLK) begin
   if(count_start)
      count <= count + 1'b1;
   else
      count <= 0;

   if(trans_count_start)
      trans_count <= trans_count + 1'b1;
   else
      trans_count <= 0;

   if(second_count_start)
      second_count <= second_count + 1'b1;
   else
      second_count <= 0;
end
```

**Figure 12:** Code related to transact

- Failed: This state will output FAILED_TRAN = 1 for 1 clock pulse and then return to idle.

```
failed: begin next = idle; end
```

```
failed: begin
   FAILED_TRAN = 1;
end
```

**Figure 13:** Code related to failed

- Vend: This state decrements the count of the selected item by 1, outputs VEND = 1 for one clock pulse, and then returns to idle.

```
vend: begin next = idle; end
```

```
vend: begin
   VEND = 1;
end
```

**Figure 14:** Code related to vend

The following **Figure 15** is the RTL circuit generated by ISE.

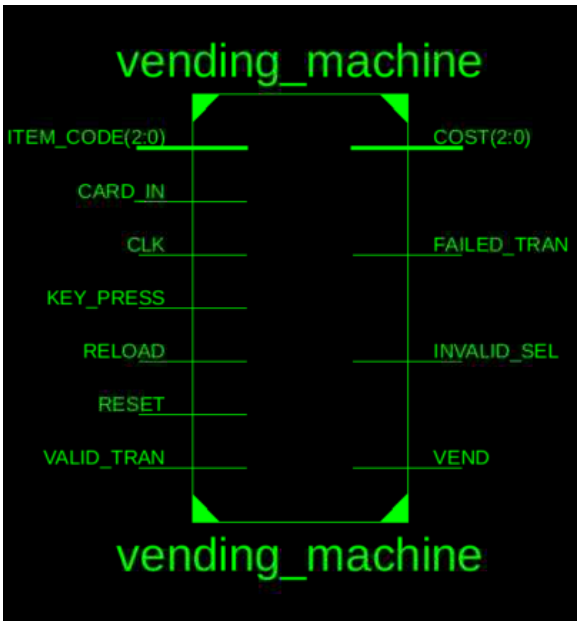


**Figure 15:** RTL circuit of lab 3

This is a black box diagram of my code that takes the variables on the left side as input and outputs variables on the right side.

## Simulation

The testbench code I created tests 1 successful transaction and 5 special cases of errors. (Following code snippets will be long and will be explained in the video too). **Figure 16** is the testbench code for the successful transaction. All the codes in the figures are read from top to bottom first then left to right.

```
//first successful case
CLK = 1;
RESET = 1;
#5
CLK = 0;
#5 //IDLE
RESET = 0;
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
RELOAD = 1; // LOAD
#5
CLK = 1;
#5
RELOAD = 0;
CLK = 0;// IDLE
#5
CLK = 1;
#5
CARD_IN = 1;
CLK = 0; // WAIT FOR INPUT
#5
CLK = 1;
#5
CARD_IN = 0;
ITEM_CODE = 3'b010;
KEY_PRESS = 1; // first input: 2
CLK = 0;
#5
CLK = 1;
#5
KEY_PRESS = 0;
CLK = 0;
#5
CLK = 1;
#5
ITEM_CODE = 3'b010;
KEY_PRESS = 1; //second input: 2
CLK = 0;
#5 //total input : 22
CLK = 1;
#5
KEY_PRESS = 0; //no more input
CLK = 0;
#5
CLK = 1;
#5
VALID_TRAN = 1; //validate to vend
CLK = 0;
#5
CLK = 1;
#5
VALID_TRAN = 0;
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1; // idle
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
```

**Figure 16:** Code for test case 1

This is a normal transaction with an input of 22. So the cost should be 5. **Figure 17** is the waveform of the code, and the VEND becomes high, which means the transaction was successful. Moreover, the cost is 101 = 5.

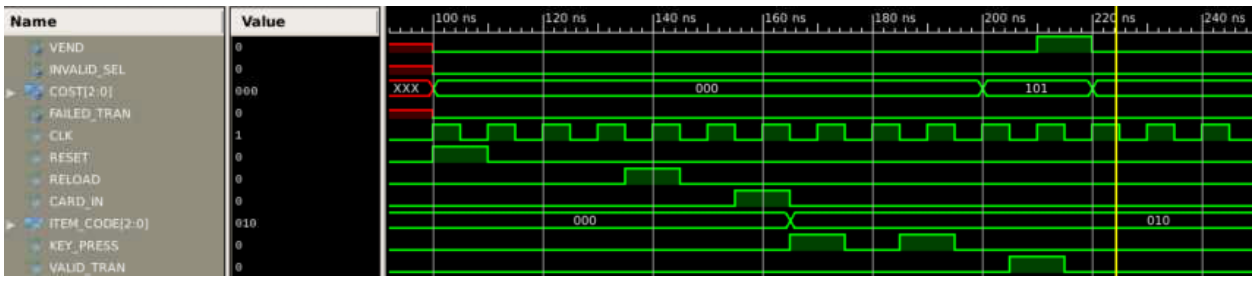


**Figure 17:** Waveform for test case 1

Test case 2 is the first error test case. This tests the situation where we try to vend without reloading the items. It should output INVALID_SEL = 1 because there are not enough items to vend.

```
//first special case. try to
#5
CLK = 1;
RESET = 1;
#5
CLK = 0;
#5 //IDLE
RESET = 0;
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
RELOAD = 0; // Don't LOAD TH
#5
CLK = 1;
#5
RELOAD = 0;
CLK = 0;// IDLE
#5
CLK = 1;
#5
CARD_IN = 1;
CLK = 0; // WAIT FOR INPUT
#5
CLK = 1;
#5
CARD_IN = 0;
ITEM_CODE = 3'b010;
KEY_PRESS = 1; // first
CLK = 0;
#5
CLK = 1;
#5
KEY_PRESS = 0;
CLK = 0;
#5
CLK = 1;
#5
ITEM_CODE = 3'b010;
KEY_PRESS = 1; //second
CLK = 0;
#5 //total input : 22
CLK = 1;
#5
KEY_PRESS = 0; //no mor
CLK = 0;
#5
CLK = 1;
#5
VALID_TRAN = 1; //valid
CLK = 0;
#5
CLK = 1;
#5
VALID_TRAN = 0;
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1; // idle
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
```

**Figure 18:** Code for test case 2

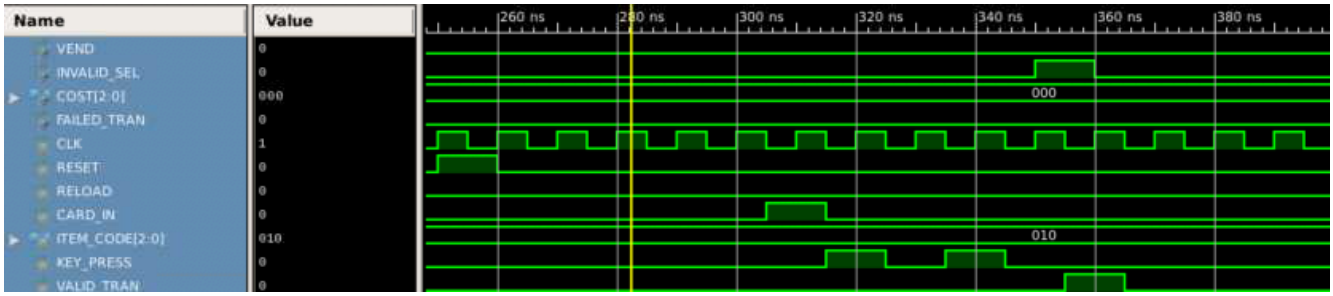


**Figure 19:** Waveform for test case 2

As shown on the waveform in **Figure 19**, the machine does not vend; instead, it sets INVALID_SEL to be high. So, this error is safe.

Test case 3 is the second special case. As shown in **Figure 20**, I never set VALID_TRAN to be 1 and try to run the machine. Correct output should be FAILED_TRAN to be 1.

```
//second case Valid_tran is
#5
CLK = 1;
RESET = 1;
#5
CLK = 0;
#5 //IDLE
RESET = 0;
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
RELOAD = 1; // LOAD
#5
CLK = 1;
#5
RELOAD = 0;
CLK = 0;// IDLE
#5
CLK = 1;
#5
CARD_IN = 1;
CLK = 0; // WAIT FOR INPUT
#5
CLK = 1;
#5
CARD_IN = 0;
ITEM_CODE = 3'b010;
KEY_PRESS = 1; // 1
CLK = 0;
#5
CLK = 1;
#5
KEY_PRESS = 0;
CLK = 0;
#5
CLK = 1;
#5
ITEM_CODE = 3'b010;
KEY_PRESS = 1; //se
CLK = 0;
#5 //total input :
CLK = 1;
#5
KEY_PRESS = 0; //nc
CLK = 0;
#5
CLK = 1;
#5
VALID_TRAN = 0; //I
CLK = 0;
#5
CLK = 1;
#5
VALID_TRAN = 0; //waiting
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1; //IDLE
#5
CLK = 0;
```

**Figure 20:** Code for test case 3

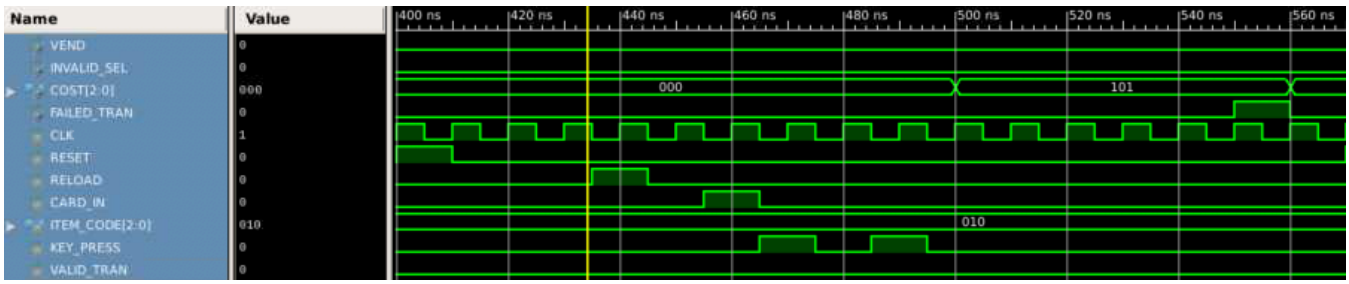


**Figure 21:** Waveform for test case 3

As shown in **Figure 21**, the VALID_TRAN never rises, so vend also never rises. Moreover, FAILED_TRAN becomes high, signifying the success in covering this error.

Test case 4 is the third special case where the user's input/selection is invalid because it is out of range. In **Figure 22**, I input ITEM_CODE to be first 010 then 111, which is 27. Since 27 is not between 20 and 24, it is an invalid selection. Thus, it should not vend and set INVALID_SEL to be high.

```
//third case: selection is in
#5
CLK = 1;
RESET = 1;
#5
CLK = 0;
#5 //IDLE
RESET = 0;
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
RELOAD = 1; // LOAD
#5
CLK = 1;
#5
RELOAD = 0;
CLK = 0;// IDLE
#5
CLK = 1;
#5
CARD_IN = 1;
CLK = 0; // WAIT FOR INPUT
#5
CLK = 1;
#5
CARD_IN = 0;
ITEM_CODE = 3'b010;
KEY_PRESS = 1; // fi
CLK = 0;
#5
CLK = 1;
#5
KEY_PRESS = 0;
CLK = 0;
#5
CLK = 1;
#5
ITEM_CODE = 3'b111;
KEY_PRESS = 1; //sec
CLK = 0;
#5 //total input : 2
CLK = 1;
#5
KEY_PRESS = 0; //no
CLK = 0;
#5
CLK = 1;
#5
VALID_TRAN = 1; //va
CLK = 0;
#5
CLK = 1;
#5
VALID_TRAN = 0;
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1; // idle
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
```

**Figure 22:** Code for test case 4

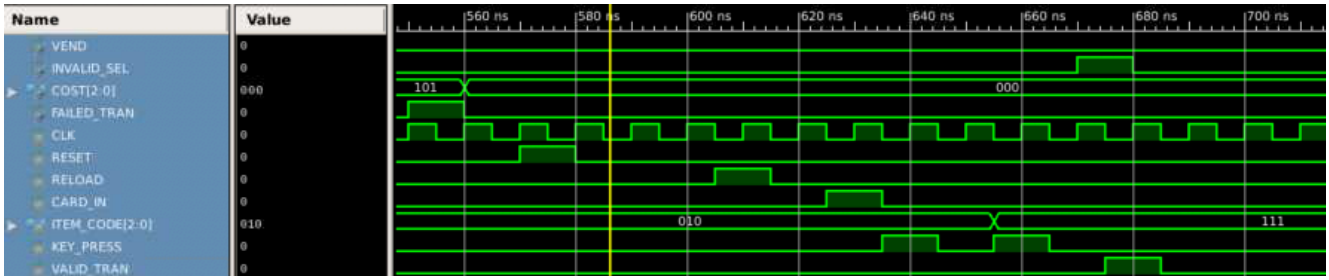


**Figure 23:** Waveform for test case 4

**Figure 23** shows how as soon as ITEM_CODE input is 111, the machine recognizes that 27 is not in range and sets INVALID_SEL to be high. Most importantly, it does not vend anything too.

Test case 5 is the fourth special case where the second input of ITEM_CODE is not inserted within 5 clock cycles. Since there were no complete inputs inserted, the machine should be idle. In **Figure 24**, there are streaks of just clock switching instead of inputting the second digit.

```
//fourth case: second digit
CLK = 1;
RESET = 1;
#5
CLK = 0;
#5 //IDLE
RESET = 0;
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
RELOAD = 1; // LOAD
#5
CLK = 1;
#5
RELOAD = 0;
CLK = 0;// IDLE
#5
CLK = 1;
#5
CARD_IN = 1;
CLK = 0; // WAIT FOR INPUT
#5
CLK = 1;
#5
CARD_IN = 0;
ITEM_CODE = 3'b010;
KEY_PRESS = 1; // fi
CLK = 0;
#5
CLK = 1;
#5
KEY_PRESS = 0; // 5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
ITEM_CODE = 3'b010;
KEY_PRESS = 1; //secor
CLK = 0;
#5 //total input : 22
CLK = 1;
#5
KEY_PRESS = 0; //no mc
CLK = 0;
#5
CLK = 1;
#5
VALID_TRAN = 1; //vali
CLK = 0;
#5
CLK = 1;
#5
VALID_TRAN = 0;
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1; // idle
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
```

**Figure 24:** Code for test case 6

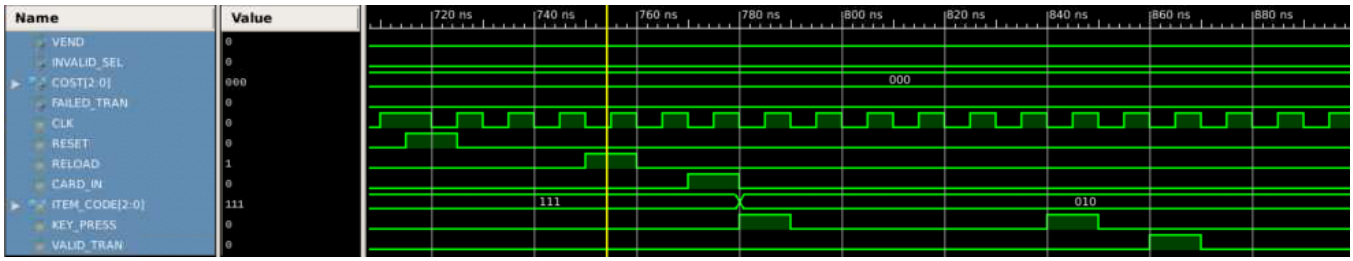


**Figure 25:** Code for test case 6

The waveform in **Figure 25**, shows that the machine successfully went back to idle instead of raising INVALID_SEL to 1 or vending the item. After the 5 clock pulses (positive edge), the machine went back to idle.

Lastly, test case 6 is the fifth special case where we vend the same item 11 times. Since there is a maximum of 10 items in the stack, without reloading the stack, the item will run out before it can vend the 11th one. Thus, the output should be regular transactions until the last transaction. On the last transaction, it should output INVALID_SEL = 1 since there are no items to vend. I did not snip every single part of test case 6's code and waveform because the middle

part where its constant vending is all repetitive. Thus, I included the last transactions and a couple of beginning ones.

```
//fifth case: CARD_IN
CLK = 1;
RESET = 1;
#5
CLK = 0;
#5 //IDLE
RESET = 0;
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
RELOAD = 1; // LOAD
#5
CLK = 1;
#5
RELOAD = 0;
CLK = 0;// IDLE
#5
CLK = 1;
#5
CARD_IN = 1; //turn th
CLK = 0; // WAIT FOR I
#5
--
CLK = 1;
#5
ITEM_CODE = 3'b010;
KEY_PRESS = 1; // first input: 2
CLK = 0;
#5
CLK = 1;
#5
KEY_PRESS = 0;
CLK = 0;
#5
CLK = 1;
#5
ITEM_CODE = 3'b010;
KEY_PRESS = 1; //second input: 2
CLK = 0;
#5 //total input : 22
CLK = 1;
#5
KEY_PRESS = 0; //no more input
CLK = 0;
#5
CLK = 1;
#5
VALID_TRAN = 1; //validate to ve
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1; // idle
#5
CLK = 0;
//second transactio
#5
CLK = 1;
#5
KEY_PRESS = 1; // f
CLK = 0;
#5
CLK = 1;
#5
KEY_PRESS = 0;
CLK = 0;
#5
CLK = 1;
#5
KEY_PRESS = 1; //se
CLK = 0;
#5 //total input :
CLK = 1;
#5
KEY_PRESS = 0; //no
CLK = 0;
```

**Figure 26:** Code for test case 6 pt.1

```
//eleventh transaction = should
#5
CLK = 1;
#5
KEY_PRESS = 1; // first input: 2
CLK = 0;
#5
CLK = 1;
#5
KEY_PRESS = 0;
CLK = 0;
#5
CLK = 1;
#5
KEY_PRESS = 1; //second input: 2
CLK = 0;
#5 //total input : 22
CLK = 1;
#5
KEY_PRESS = 0; //no more input
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1;
#5
CLK = 0;
#5
CLK = 1; // idle
#5
CLK = 0;
```

**Figure 27:** Code for test case 6 pt.2

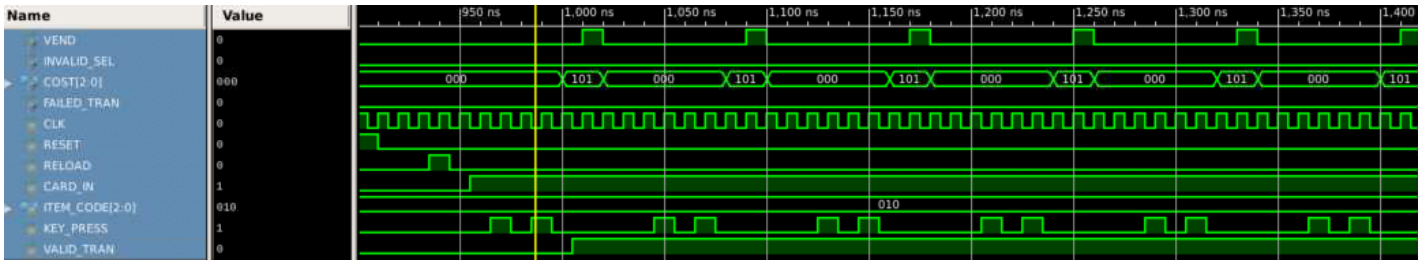


**Figure 28:** Waveform for test case 6 pt.1

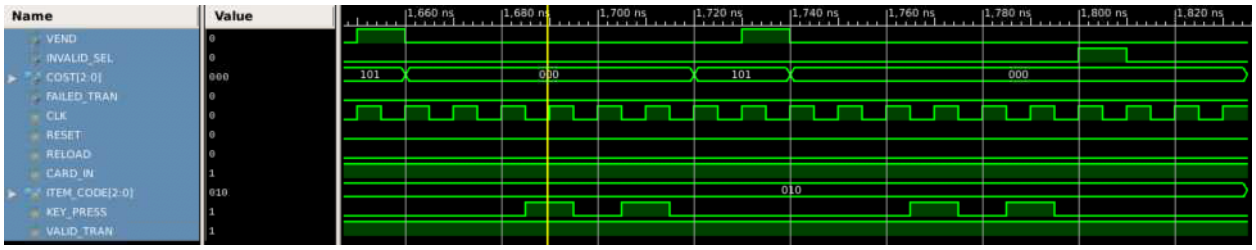


**Figure 29:** Waveform for test case 6 pt.2

As shown in the above waveforms, normal transactions happen until the last transaction. We can conclude that it's a normal transaction due to VEND going high and the cost being 5. On the eleventh transaction, the machine does not vend, and it sets INVALID_SEL high as predicted.

The below diagram is the Design Summary and the implementation report of the code. However, the implementation report for the map report occurs error not due to my code, but due to the error of "No ISE nor WebPack feature version 2013.10 was available" with the addition of licensing issue.

| | | | |
|---|---|---|---|
| **Module Name:** | vending_machine | **Implementation State:** | Synthesized |
| **Target Device:** | xc6slx16-3csg324 | **• Errors:** | No Errors |
| **Product Version:** | ISE 14.7 | **• Warnings:** | 18 Warnings (0 new) |
| **Design Goal:** | Balanced | **• Routing Results:** | |
| **Design Strategy:** | Xilinx Default (unlocked) | **• Timing Constraints:** | |
| **Environment:** | System Settings | **• Final Timing Score:** | |

| **Device Utilization Summary (estimated values)** | | | [-] |
|---|---|---|---|
| **Logic Utilization** | **Used** | **Available** | **Utilization** |
| Number of Slice Registers | 70 | 18224 | 0% |
| Number of Slice LUTs | 121 | 9112 | 1% |
| Number of fully used LUT-FF pairs | 59 | 132 | 44% |
| Number of bonded IOBs | 15 | 232 | 6% |
| Number of BUFG/BUFGCTRLs | 1 | 16 | 6% |

| **Detailed Reports** | | | | | [-] |
|---|---|---|---|---|---|
| **Report Name** | **Status** | **Generated** | **Errors** | **Warnings** | **Infos** |
| Synthesis Report | Current | Fri May 21 12:45:17 2021 | 0 | 18 Warnings (0 new) | 2 Infos (0 new) |
| Translation Report | Out of Date | Fri May 21 12:43:36 2021 | 0 | 0 | 0 |
| Map Report | Out of Date | Fri May 21 12:43:42 2021 | X 1 Error (0 new) | 0 | 0 |
| Place and Route Report | | | | | |
| Power Report | | | | | |
| Post-PAR Static Timing Report | | | | | |
| Bitgen Report | | | | | |

**Figure 30:** Design Summary

| vending_machine Project Status (05/21/2021 - 13:38:57) | | | |
|---|---|---|---|
| **Project File:** | Proj3.xise | **Parser Errors:** | No Errors |
| **Module Name:** | vending_machine | **Implementation State:** | Mapped (Failed) |
| **Target Device:** | xc6slx16-3csg324 | **• Errors:** | X 1 Error (0 new) |
| **Product Version:** | ISE 14.7 | **• Warnings:** | 18 Warnings (0 new) |
| **Design Goal:** | Balanced | **• Routing Results:** | |
| **Design Strategy:** | Xilinx Default (unlocked) | **• Timing Constraints:** | |
| **Environment:** | System Settings | **• Final Timing Score:** | |

| Device Utilization Summary | [-] |
|---|---|

| Detailed Reports | | | | | [-] |
|---|---|---|---|---|---|
| **Report Name** | **Status** | **Generated** | **Errors** | **Warnings** | **Infos** |
| Synthesis Report | Current | Fri May 21 12:45:17 2021 | 0 | 18 Warnings (0 new) | 2 Infos (0 new) |
| Translation Report | Current | Fri May 21 13:38:56 2021 | 0 | 0 | 0 |
| Map Report | Current | Fri May 21 13:39:01 2021 | X 1 Error (0 new) | 0 | 0 |
| Place and Route Report | | | | | |
| Power Report | | | | | |
| Post-PAR Static Timing Report | | | | | |
| Bitgen Report | | | | | |

**Figure 31:** Map Report

## Conclusion

I was not aware that I can fail to synthesize the code even if the simulation works perfectly. I was having the problem of having a unit connected to multiple drivers in always block. Moreover, the map report did not give an error due to an error in my code, but the software. This lab was fair but I do feel like the number of test cases should be lessened.