Danning Yu, 305087992
CS M152A, Lab 2
TA: Logan Kuo

Project 4 Report

Introduction

The purpose of this lab is to model a finite state machine (FSM) by designing a vending machine that allows users to select an item, pay for it using a credit card, and then vend it if the transaction is successful. Through this lab, we will learn about finite state machines (FSMs), how to design one using Verilog in the Xilinx ISE, and how to test a FSM in Xilinx Isim with test bench code. Finite state machines are important because they are an effective way of modeling a real life sequential system. Verilog provides a means of implementing one, and then testing it with a test bench will allow its functionality to be verified.

The vending machine should hold 20 items with item codes 00 through 19 and a maximum quantity of 10 for each item. It should have an idle state that serves as its initial and default state, from which a transaction can start if a credit card is inserted, a reset can occur to "restart" the machine with all outputs set to 0 and quantities of all items set to 0, or a reload can occur to reload all items to a quantity of 10. In the idle state, all its outputs should be 0. If a transaction starts, the customer should be able to input a 2 digit code between 00 and 19, 1 digit at a time, to select an item to buy. If this item selection is invalid due to no digit being inputted, the item code being out of range, or there being none of that item left, the machine should output that this was an invalid selection and then return to the idle state. If the item selection was valid, the vending machine should display its cost and wait for the credit card to be authorized. If the credit card is not authorized, the machine should display this failure and return to the idle state. If the credit card is authorized, the machine should vend the item, decrement the quantity left of that item by 1, and wait for the customer to open the door, take the item, and then close the door before returning to the idle state. If the customer does not open the door within a certain time duration, the vending machine should automatically return to the idle state. Only 1 item may be purchased at a time and the timeout duration for all actions (pressing a key, waiting for authorization, and waiting for the door to open) is 5 clock cycles. The system clock runs at 100 MHz, so the timeout duration is 50 ns.

Design

The entire vending machine was implemented in a single module called `vending_machine`. The function of this module is to implement the behavior mentioned in the introduction section in a correct and robust manner, capable of successfully completing a transaction, gracefully handling failures, and being resistant to unexpected combinations or occurrences of input signals. The vending machine is modeled as a FSM and implemented as a sequential circuit driven by a 100 MHz system clock. Its inputs and their functionalities are given in Table 1 and likewise for its outputs in Table 2, both on the next page. Unless otherwise mentioned, signals are 1 bit wide.

Input	Description
`CLK`	System clock signal; frequency of 100 MHz
`RESET`	Signal to reset the vending machine; when high, overrides all other signals, sets all item counters and outputs to 0; causes machine to enter reset state
`RELOAD`	Signal to reload the vending machine; when high, sets all item counters to 10 and causes the machine to enter reload state; only takes effect when machine is in idle state and overrides `CARD_IN` signal
`CARD_IN`	Signal to indicate credit card is currently inserted if this signal is high; starts a transaction if machine is in idle state; can go low any time after transaction starts with no effects
`ITEM_CODE[3:0]`	4 bit signal to indicate the digit of the item being selected; only valid when `KEY_PRESS` is high (ignore the value of this signal otherwise); interpret as unsigned binary number
`KEY_PRESS`	Signal to indicate that the `ITEM_CODE` input should be read in when this signal is high
`VALID_TRAN`	Signal to indicate the credit card is authorized when this signal is high
`DOOR_OPEN`	Signal to indicate the vending machine door is open when this signal is high and closed when this signal is low

Table 1: Inputs to the vending machine and their meanings when high and/or low.

Output	Description
`VEND`	Signal to indicate that an item was vended; goes to high if item selection is valid and the credit card is authorized; goes to low once the door is opened and then closed, or if the door is not opened within 5 clock cycles
`COST[2:0]`	3 bit signal to indicate the cost of the selected item; 3’b0 by default; set to nonzero cost of an item once item selection is found to be valid and remain at this value until idle state is reached
`INVALID_SEL`	Signal to indicate an invalid digit was entered or selection was made; goes to high if no digit is entered within 5 clock cycles for either the first or second digit, or if the 2 digit item code is not valid, or if the quantity of the selected item is 0; goes to low upon entering the idle state
`FAILED_TRAN`	Signal to indicate credit card was not authorized; goes high if `VALID_TRAN` does not go high within 5 clock cycles after the item selection is found to be valid and machine is waiting for credit card authorization; goes to low upon entering the idle state

Table 2: Vending machine outputs, their expected behavior, and meanings when high and/or low.

From this table of inputs and outputs, along with the design specification, a FSM was created to model the behavior of the vending machine. A Moore machine was chosen over a Mealy machine for its output stability and better separation between inputs and outputs, which would make testing and verification easier. The vending machine does not have any speed demands, so the delayed update of outputs with a Moore machine compared to a Mealy machine is acceptable. The state machine diagram is shown on the next page, and it is followed by a table describing the behavior of each state.

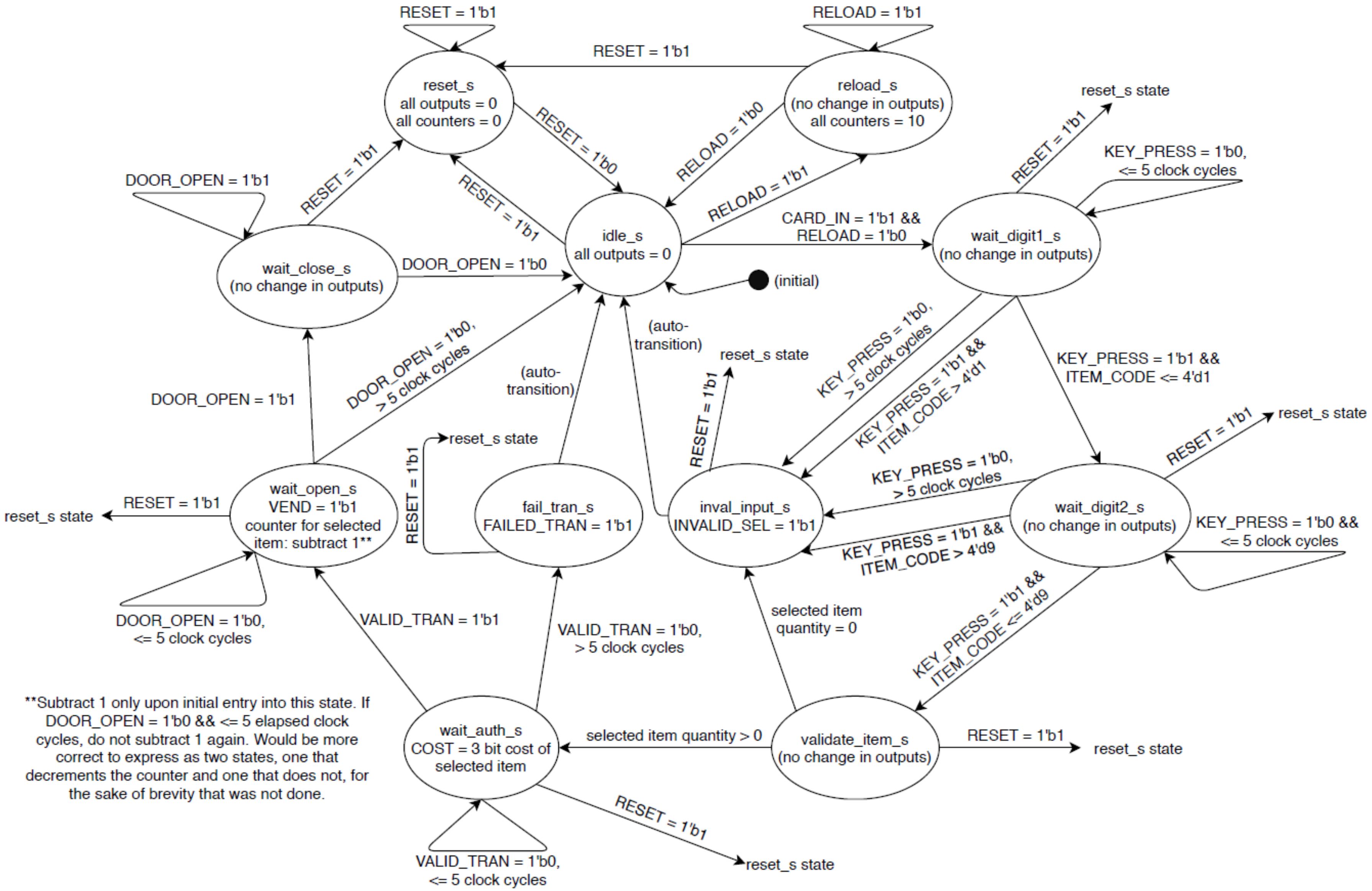


Figure 1: Moore FSM diagram for the vending machine. The machine starts out in the idle state `idle_s`, from which the machine can transition into the `reset_s` state to be reset, `reload_s` state to be reloaded, or the `wait_digit1_s` state to start a new transaction. The `RESET` signal is effective at every state, immediately causing the next state to be the `reset_s` state. There are 2 states for when errors occur: `inval_input_s` for an invalid item selection and `fail_tran_s` for a failed credit card authorization. All other states are for various steps of the transaction process. If an output is not specified for a particular state, assume its value is left unchanged. If an input is not specified for a particular state, assume that input has no effect on that state's transitions.

State #	State Name	Description
0	Reset State `reset_s`	• The machine is in the reset state • Entered when `RESET` = 1'b1 at any other state o Takes precedence over all other inputs • Sets all outputs and item counters to 0 • Exit to `idle_s` state when `RESET` = 1'b0
1	Reload State `reload_s`	• The machine is in the reload state • Entered from `idle_s` state when `RELOAD` = 1'b1 o Takes precedence over `CARD_IN` signal • Sets all item counters to 10 • Exit to `idle_s` state when `RELOAD` = 1'b0
2	Idle State `idle_s`	• The machine is currently idle • The initial state of the machine; all outputs are 0 • Entered when a transaction completes or aborts o from `wait_close_s` when `DOOR_OPEN` = 1'b0 to indicate the door closing an a transaction completing o from `wait_open_s` if `DOOR_OPEN` = 1'b0 for more than 5 clock cycles o from `reset_s` or `reload_s` when resetting or reloading finishes (`RESET` = 1'b0 or `RELOAD` = 1'b0) o from `fail_tran_s` or `inval_input_s` after 1 clock cycle is spent in those states • Exit to `reload_s` if `RELOAD` = 1'b1 • Exit to `wait_digit1_s` if `CARD_IN` = 1'b1 and `RELOAD` = 1'b0 to start a new transaction
3	Wait for Digit 1 State `wait_digit1_s`	• A new transaction has started and the machine is waiting for the first digit of the item selection • Entered from `idle_`s when `CARD_IN` = 1'b1, `RELOAD` = 1'b0 • Wait up to 5 clock cycles for `KEY_PRESS` = 1'b1 to read in a value from `ITEM_CODE` • Exit to `inval_input_s` o if there is no input (`KEY_PRESS` = 1'b0) for more than 5 clock cycles o there is input but the `ITEM_CODE` digit is not 0 or 1 • Exit to `wait_digit2_s` if there is input within 5 clock cycles (`KEY_PRESS` = 1'b1) and `ITEM_CODE` is 4'd0 or 4'd1
4	Wait for Digit 2 State `wait_digit2_s`	• Waiting for the second digit of the item selection • Very similar to `wait_digit1_s` state • Entered from `wait_digit1_s` when first digit entered is valid • Wait for up to 5 clock cycles for a digit to be inputted • Exit to `inval_input_s` o If there is no input for more than 5 clock cycles o There is input but `ITEM_CODE` digit is > 4'd9 • Exit to `validate_item_s` if there is input within 5 clock cycles and `ITEM_CODE` digit is ≤ 4'd9

(table continues on next page)

(table continued from previous page)

State #	State Name	Description
5	Validate Item Selection State `validate_item_s`	• Checking the selected item has a nonzero amount left • Entered from `wait_digit2_s` when the second digit entered is ≤ 4’d9, thus indicating an item code between 00 and 19 • Exit to `wait_auth_s` if there is a nonzero amount of the selected item left • Exit to `inval_input_s` if the selected item is out of stock
6	Wait for Authorization State `wait_auth_s`	• Waiting for credit card authorization • Output cost of the selected item as `COST` according to Table 4 • Entered from `validate_item_s` after item selection is confirmed to have a nonzero amount left • Wait for up to 5 clock cycles for authorization to arrive (`VALID_TRAN` = 1’b1) • Exit to `fail_tran_s` if there is no authorization for more than 5 clock cycles • Exit to `wait_open_s` upon authorization in ≤ 5 clock cycles
7	Wait for Door Open State `wait_open_s`	• Waiting for vending machine door to open • Entered from `wait_auth_s` when credit card is authorized • Item is vended, so set output `VEND` to 1’b1 and decrement the selected item’s counter by 1 o Decrementing the item’s counter is not an reentrant operation; it should only be done upon initial entry into this state and not repeated if the vending machine stays in this state for multiple clock cycles • Wait for up to 5 clock cycles for the door to open (`DOOR_OPEN` = 1’b1) • Exit to `idle_s` if door does not open within 5 clock cycles • Exit to `wait_close_s` if door opens within 5 clock cycles
8	Wait for Door Close State `wait_close_s`	• Waiting for vending machine door to close • Entered from `wait_open_s` when the door opens • Wait until the door closes (`DOOR_OPEN` = 1’b0) o No limit on how long this wait can be • Exit to `idle_s` when door closes
9	Invalid Input State `inval_input_s`	• State to indicate invalid item selection was made • Set `INVALID_SEL` to 1’b1 to indicate this error • Entered from `wait_digit1_s` or `wait_digit2_s` if there is no digit entered for more than 5 clock cycles, or if the digit entered is invalid, or from `validate_item_s` if the item selected is out of stock (quantity is 0) • Exit to `idle_s` after 1 clock cycle
10	Failed Transaction State `fail_tran_s`	• State to indicate credit card authorization failed • Set `FAILED_TRAN` to 1’b1 to indicate this error • Entered from `wait_auth_s` after authorization is not provided for more than 5 clock cycles • Exit to `idle_s` after 1 clock cycle

Table 3: Descriptions of the each state from Figure 1 and its outputs and transitions.

The cost of each item is given by the table below. This table is used to determine the value of `COST` to output for a particular valid item selection.

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

Table 4: Cost of each item according to their item code.

Using Figure 1 and Tables 1 to 4, code was written in Verilog to implement the FSM. In addition to the inputs and outputs given in Tables 1 and 2, within the module, 20 4 bit item counter registers were declared and initialized to 0, each with a name `item<xx>`, where `xx` is an item number in the range 0-19. Also, each state was assigned a number in accordance with Table 3, a universal timeout parameter `timeout` was set to 5, and a 4 bit variable `state` was created to hold the current state and initialized to `idle_s`. Finally, a 5 bit `selected_item` variable was created and initialized to 0 to hold the number of the selected item, a 3 bit `clk_counter` was created and initialized to 1 to count the number of elapsed clock cycles while waiting for a digit input, transaction authorization, or door opening, and a 1 bit `first_digit` variable was created and initialized to 0 to store the value of the first digit entered to use later to calculate the selected item number. 1 bit suffices for the first digit because it must be either 0 or 1.

Then, 2 `always` blocks triggering on the positive edge of the clock `CLK` were used to implement the logic to update the state and create the output. One `always` block updates the state, thus providing the next state logic from the current state using a series of if-else statements on `state` and the machine's inputs. A separate `always` block provides the output from the current state, again by using if-else statements on `state`. The Verilog code extract from the project containing the aforementioned description is given below.

```
-----------------vending_machine Module Code Structure----------------
reg [3:0] item0 = 4'd0;             //initial state: idle
reg [3:0] item1 = 4'd0;             reg [3:0] state = idle_s;
// ...and so on...                  reg [4:0] selected_item = 5'd0;
reg [3:0] item18 = 4'd0;            reg [2:0] clk_counter = 3'd1;
reg [3:0] item19 = 4'd0;            reg first_digit = 1'd0;

parameter reset_s = 4'd0;           always @(posedge CLK) begin
parameter reload_s = 4'd1;            // next state logic goes here
parameter idle_s = 4'd2;              if(state == <state here>) begin
parameter wait_digit1_s = 4'd3;         // update to next state...
parameter wait_digit2_s = 4'd4;       end
parameter validate_item_s = 4'd5;   end
parameter wait_auth_s = 4'd6;
parameter wait_open_s = 4'd7;       always @(posedge CLK) begin
parameter wait_close_s = 4'd8;        // output logic goes here
parameter inval_input_s = 4'd9;       if(state == <state here>) begin
parameter fail_tran_s = 4'd10;          // output something...
                                      end
parameter timeout = 3'd5;           end
----------------------------------------------------------------------
```

The following subsections present Verilog code for each state and show how it implements the state machine given in Figure 1. See Table 3 for details about each state. Only code relevant to that particular state is shown. The internal item counters are considered "outputs" and thus updated in the outputs `always` block.

Reset State (`reset_s`)

The reset state is special in that it can be entered from any other state. Thus, when updating to this state, rather than checking that the current state is a particular state, it is simply placed first in the if-else chain. The only way to exit this state is if `RESET` is 1'b0. In this state, all item counters are zeroed out and all outputs are 0.

```
--------------------------reset_s State Code--------------------------
always @(posedge CLK) begin
  if(RESET) begin
    state = reset_s;
  end
  else if(state == reset_s) begin
    if(!RESET) state = idle_s;
  end
  else if(state ==
    <some other state>) begin
    // update to next state...
  end
end

always @(posedge CLK) begin
  if(state == reset_s) begin
    item0 = 4'd0;
    item1 = 4'd0;
    // ... and so on ...
    item18 = 4'd0;
    item19 = 4'd0;
    VEND = 1'b0;
    INVALID_SEL = 1'b0;
    FAILED_TRAN = 1'b0;
    COST = 3'b0;
  end
  else if(state == <state here>)
  begin
    // output something else...
  end
end
----------------------------------------------------------------------
```

Reload State (`reload_s`)

The reload state can only be entered from the idle state, and we do not exit from this state until `RELOAD` = 1'b0. This state simply serves to set all the item counters to 10.

```
--------------------------reload_s State Code-------------------------
always @(posedge CLK) begin
  if(state == idle_s) begin
    if(RELOAD) state = reload_s;
    //...code for wait_digit1_s...
  end
  else if(state == reload_s) begin
    if(!RELOAD) state = idle_s;
  end
  else if(state ==
    <some other state>) begin
    // update to next state...
  end
end

always @(posedge CLK) begin
  // ... other if-else statements
  else if(state == reload_s) begin
    item0 = 4'd10;
    item1 = 4'd10;
    // ... and so on ...
    item18 = 4'd10;
    item19 = 4'd10;
  end
  else if(state == <state here>)
    // output something else...
end
----------------------------------------------------------------------
```

Idle State (`idle_s`)

The idle state is the default state of the machine. It is entered when `RESET` or `RELOAD` goes to low after going high, or when a transaction is completed with the door closing after it is opened, or when the door is not opened and it times out, or after a failure state due to invalid item selection input or failed credit card authorization. If `RELOAD` goes high, it transitions to the `reload_s` state; if not, then it checks if `CARD_IN` is high, and if so, it transitions into the `wait_digit1_s` state, starting a new transaction. It also resets the clock counter `clk_counter` to 1 to start counting the number of elapsed cycles. The `RELOAD` signal takes priority over the `CARD_IN` signal, so if both become high, the reload state is entered.

```
--------------------------idle_s State Code-------------------------
always @(posedge CLK) begin
  // ... logic for other states
  if(state == reset_s) begin
    if(!RESET) state = idle_s;
  end
  else if(state == reload_s) begin
    if(!RELOAD) state = idle_s;
  end
  else if(state == idle_s) begin
    if(RELOAD) state = reload_s;
    else if(CARD_IN) begin
      state = wait_digit1_s;
      clk_counter = 3'd1; //start counting # of clock cycles
    end
  end
  // ... logic for other states
End

always @(posedge CLK) begin
  // ... logic for other states
  else if(state == idle_s) begin
    VEND = 1'b0;
    INVALID_SEL = 1'b0;
    FAILED_TRAN = 1'b0;
    COST = 3'b0;
  end
  // ... logic for other states
end
----------------------------------------------------------------------
```

(report continues on next page)

Wait for Digit 1 State (`wait_digit1_s`)

To implement this state, which is entered from the idle state when `CARD_IN` is high and `RELOAD` is low, we first check if a timeout has occurred (no input for > 5 clock cycles) by checking the value of `clk_counter`. If this has occurred, the state changes to the invalid input state. Otherwise, we check if a digit is being inputted using `KEY_PRESS`. If so, and the digit inputted is 4'd0 or 4'd1, which are the only possible first digits for the item code, the state saves this digit, resets the clock counter, and moves on to waiting for the second digit. If the digit is invalid, the state becomes the invalid input state. Finally, if no input is provided, then the clock counter is incremented to show that another clock cycle has passed. This state has no outputs.

```
------------------------wait_digit1_s State Code----------------------
always @(posedge CLK) begin
  // ... logic for other states
  else if(state == idle_s) begin
    if(RELOAD) state = reload_s;
    else if(CARD_IN) begin
      state = wait_digit1_s;
      clk_counter = 3'd1;
    end
  end
  else if(state == wait_digit1_s)
  begin
    if(clk_counter == timeout)
      state = inval_input_s;
    else if(KEY_PRESS) begin
      if(ITEM_CODE > 4'd1)
        state = inval_input_s;
      else begin
        state = wait_digit2_s;
        first_digit = ITEM_CODE[0];
        clk_counter = 3'd1;
      end // close ELSE clause
    end // close else if(KEY_PRESS)
    else //KEY_PRESS = 1'b0
      clk_counter = clk_counter + 3'd1;
  end //close if wait_digit1_s
  // ... logic for other states
end //close always block

always @(posedge CLK) begin
  // this state does not change
  // any outputs
end
----------------------------------------------------------------------
```

Wait for Digit 2 State (`wait_digit2_s`)

This state is almost the exact same as the wait for digit 1 state, except that it is entered from the `wait_digit1_s` state, the valid digits are the in the range 0–9, inclusive, and if the inputted digit is valid, it goes to the validate item state. It also calculates the selected item code using the saved value of the previous digit in `first_digit` and stores it in `selected_item`.

```
------------------------wait_digit2_s State Code----------------------
always @(posedge CLK) begin
  // see wait_digit1_s code to see
  // how this state is entered
  else if(state == wait_digit2_s)
  begin
    if(clk_counter == timeout)
      state = inval_input_s;
    else if(KEY_PRESS) begin
      if(ITEM_CODE > 4'd9)
        state = inval_input_s;
      else begin //ITEM_CODE ≤ 9
        selected_item
          = first_digit * 4'd10
            + ITEM_CODE;
        state = validate_item_s;
        clk_counter = 3'd1;
      end //close ELSE clause
    end //close else if(KEY_PRESS)
    else //KEY_PRESS = 1'b0
      clk_counter = clk_counter + 3'd1;
  end //close if for wait_digit2_s
  // ... logic for other states
end //close always block

always @(posedge CLK) begin
  // this state does not change
  // any outputs
end
----------------------------------------------------------------------
```

Validate Item Selection State (`validate_item_s`)

To implement this state, we need to check if the quantity of the item selected is more than 0. Thus, we use the value of `selected_item` to get the selected item code and then check if that item is in stock. If not, it updates state to the invalid input state; otherwise, it resets the clock counter and updates state to `wait_auth_s` to reflect the fact that we are now awaiting credit card authorization. This state does not output anything.

```
-----------------------validate_item_s State Code---------------------
always @(posedge CLK) begin
  // ... logic for other states
  // see wait_digit2_s state code to see how this state is reached
  else if(state == validate_item_s) begin
    if(selected_item == 5'd0 && item0 == 4'd0)
      state = inval_input_s;
    else if(selected_item == 5'd1 && item1 == 4'd0)
      state = inval_input_s;
    else if(selected_item == 5'd2 && item2 == 4'd0)
      state = inval_input_s;
    // ... and so on for all other item codes ...
    else if(selected_item == 5'd18 && item18 == 4'd0)
      state = inval_input_s;
    else if(selected_item == 5'd19 && item19 == 4'd0)
      state = inval_input_s;
    else begin
      state = wait_auth_s;
      clk_counter = 3'd1;
    end
  end
  // ... logic for other states
end

always @(posedge CLK) begin
  // this state does not change any outputs
end
----------------------------------------------------------------------
```

(report continues on next page)

Wait for Authorization State (`wait_auth_s`)

To implement this state, we use logic similar to the waiting for digit states. If 5 clock cycles have passed (counted using `clk_counter`) and the authorization has not yet arrived, the state changes to the failed transaction state. Otherwise, if it arrives, then we update the state to the waiting for the door to open, `wait_open_s.` If it has not yet been 5 clock cycles and the authorization has not yet arrived, we simply increment the clock counter by 1 to reflect that another clock cycle has passed. Since the item selection has been validated, in this state the cost of the item is outputted according to Table 4 with the use of case statement on `selected_item`.

```
-------------------------wait_auth_s State Code-----------------------
always @(posedge CLK) begin
  // ... logic for other states
  // see validate_item_s code to
  // see how this state is entered
  else if(state == wait_auth_s)
  begin
    if(clk_counter == timeout)
      state = fail_tran_s;
    else if(VALID_TRAN) begin
      state = wait_open_s;
      clk_counter = 3'd1;
    end
    else begin
    clk_counter
      = clk_counter + 3'd1;
    end
  end /close if for wait_auth_s
  // ... logic for other states
end

always @(posedge CLK) begin
  // ... logic for other states
  else if(state == wait_auth_s)
  begin
  //case statement for item
    case(selected_item)
      5'd0,
      5'd1,
      5'd2,
      5'd3: COST = 3'd1;
      5'd4,
      5'd5,
      5'd6,
      5'd7: COST = 3'd2;
      5'd8,
      5'd9,
      5'd10,
      5'd11: COST = 3'd3;
      5'd12,
      5'd13,
      5'd14,
      5'd15: COST = 3'd4;
      5'd16,
      5'd17: COST = 3'd5;
      default: COST = 3'd6;
      //guaranteed to be 18 or 19
    endcase
  end // close if for wait_auth_s
  // ... logic for other states
end //close always block
----------------------------------------------------------------------
```

(report continues on next page)

Wait for Door Open State (wait_open_s)

To implement this state, logic similar to the wait for authorization and wait for digit input states was used. However, this time, if the door does not open for 5 clock cycles, there is no intermediate failure state, so the state changes directly to the idle state. Otherwise, if the door has not been opened for less than 5 clock cycles, we stay in this state and increment clk_counter by 1. If the door opens (DOOR_OPEN = 1’b1) within 5 clock cycles, the state changes to waiting for the door to close. Also, since the credit card was authorized from the last state, the output VEND signal is set to high and upon initial entry to this state, the counter for the selected item is decremented by 1 to reflect the item being vended. To prevent the counter from being repeatedly decremented, as this is not a reentrant operation, it is only done when VEND is 1’b0, which is when the state is first entered.

```
-------------------------wait_open_s State Code-----------------------
always @(posedge CLK) begin
  // ... logic for other states
  // see wait_auth_s code to
  // see how this state is entered
  else if(state == wait_auth_s)
  begin
    if(clk_counter == timeout)
      state = fail_tran_s;
    else if(VALID_TRAN) begin
      state = wait_open_s;
      clk_counter = 3'd1;
    end
    else begin
    clk_counter
      = clk_counter + 3'd1;
    end
  end
  // ... logic for other states
end //close always block

always @(posedge CLK) begin
  // ... logic for other states
  else if(state == wait_open_s)
  begin
    if(VEND == 1'b0) begin
      // only decrement once
      // code continued on next
      // column
      case(selected_item)
        5'd0: item0 = item0 - 4'd1;
        5'd1: item1 = item1 - 4'd1;
        5'd2: item2 = item2 - 4'd1;
        5'd3: item3 = item3 - 4'd1;
        5'd4: item4 = item4 - 4'd1;
        5'd5: item5 = item5 - 4'd1;
        5'd6: item6 = item6 - 4'd1;
        5'd7: item7 = item7 - 4'd1;
        5'd8: item8 = item8 - 4'd1;
        5'd9: item9 = item9 - 4'd1;
        5'd10: item10 = item10 - 4'd1;
        5'd11: item11 = item11 - 4'd1;
        5'd12: item12 = item12 - 4'd1;
        5'd13: item13 = item13 - 4'd1;
        5'd14: item14 = item14 - 4'd1;
        5'd15: item15 = item15 - 4'd1;
        5'd16: item16 = item16 - 4'd1;
        5'd17: item17 = item17 - 4'd1;
        5'd18: item18 = item18 - 4'd1;
        default: item19 =
                   item19 - 4'd1;
        //guaranteed to be 19
      endcase
    end //end if(VEND = 1’b0)
    VEND = 1'b1;
  end //close if for wait_open_s
  // ... logic for other states
end //close always block
----------------------------------------------------------------------
```

(report continues on next page)

Wait for Door Close State (wait_close_s)

To implement this state, simply allow it to move on to the idle state if the door is closed (that is, DOOR_OPEN = 1’b0). Otherwise, if the door stays open, we will stay forever in this state unless the RESET input becomes high. This state does not output anything.

```
------------------------wait_close_s State Code-----------------------
always @(posedge CLK) begin        | always @(posedge CLK) begin
  // ... logic for other states    |   // ... logic for other states
  // see wait_open_s code to see   |   // this state does not change
  // how this state is entered     |   // any outputs
  else if(state == wait_close_s)   |   // ... logic for other states
    if(!DOOR_OPEN) state = idle_s; | end
  // ... logic for other states    |
end                                |
----------------------------------------------------------------------
```

Invalid Input State (inval_input_s)

This state is simply an intermediate state that causes the INVALID_SEL output to be high for 1 clock cycle to indicate an invalid selection error. After 1 clock cycle, it automatically moves to the idle state idle_s. Thus, it is very simple, simply consisting of a state update and outputting INVALID_SEL as high. It is entered from one of the wait for digit states if no digit is entered for more than 5 clock cycles or an invalid digit is entered, or from the validate item selection state if there are none left of the item selected.

```
-----------------------inval_input_s State Code-----------------------
always @(posedge CLK) begin        | always @(posedge CLK) begin
  // ... logic for other states    |   // ... logic for other states
  // see other states’ code to see |   else if(state == inval_input_s)
  // how this state is entered     |     INVALID_SEL = 1'b1;
  else if(state == inval_input_s)  |   // ... logic for other states
    state = idle_s;                | end
  // ... logic for other states    |
end                                |
----------------------------------------------------------------------
```

Failed Transaction State (fail_tran_s)

This state is simply an intermediate state that causes the FAILED_TRAN output to be high for 1 clock cycle to indicate an invalid selection error, similar to the invalid input state. After 1 clock cycle, it automatically moves to the idle state idle_s. Thus, it is very simple, simply consisting of a state update and outputting FAILED_TRAN as high. It is entered from the wait for authorization state when the authorization does not come for more than 5 clock cycles.

```
------------------------fail_tran_s State Code------------------------
always @(posedge CLK) begin        | always @(posedge CLK) begin
  // ... logic for other states    |   // ... logic for other states
  // see wait_auth_s code to see   |   else if(state == fail_tran_s)
  // how this state is entered     |     FAILED_TRAN = 1'b1;
  else if(state == fail_tran_s)    | end
    state = idle_s;                |
end                                |
----------------------------------------------------------------------
```

After this module was synthesized, the resulting RTL schematic was extremely complex, most likely due to the 20 registers used to store the quantity of each item and the 10 states used. Most of the states also have multiple branches of logic to decide which state to transition to next, so that most likely increases the complexity as well. The full RTL schematic is included as Appendix A; when shrunk down to the size of 1 page, it is nearly unreadable. However, relevant parts can still be extracted from the generated RTL schematic, and thus are shown below.

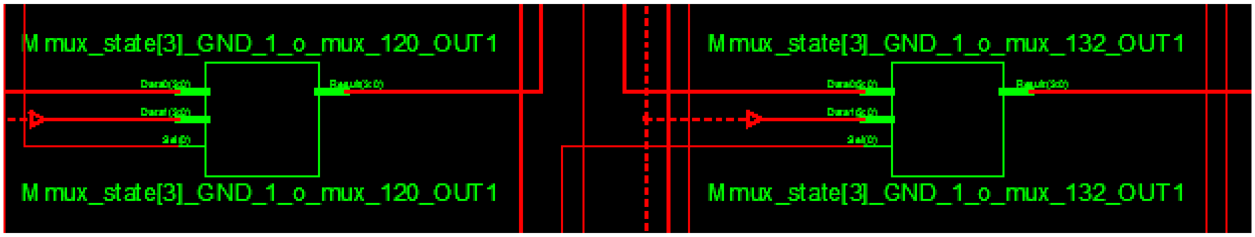

Mmux_state[3]_GND_1_o_mux_120_OUT1
Data0(3:0)
Data1(3:0)
Sel(0)
Result(3:0)
Mmux_state[3]_GND_1_o_mux_120_OUT1
Mmux_state[3]_GND_1_o_mux_132_OUT1
Data0(3:0)
Data1(3:0)
Sel(0)
Result(3:0)
Mmux_state[3]_GND_1_o_mux_132_OUT1

Figure 2: 2 to 1 4 bit multiplexers to select between various states depending on what the inputs are (expressed via the selection bits). These help to determine what the next state should be. There are many of these throughout the RTL schematic.

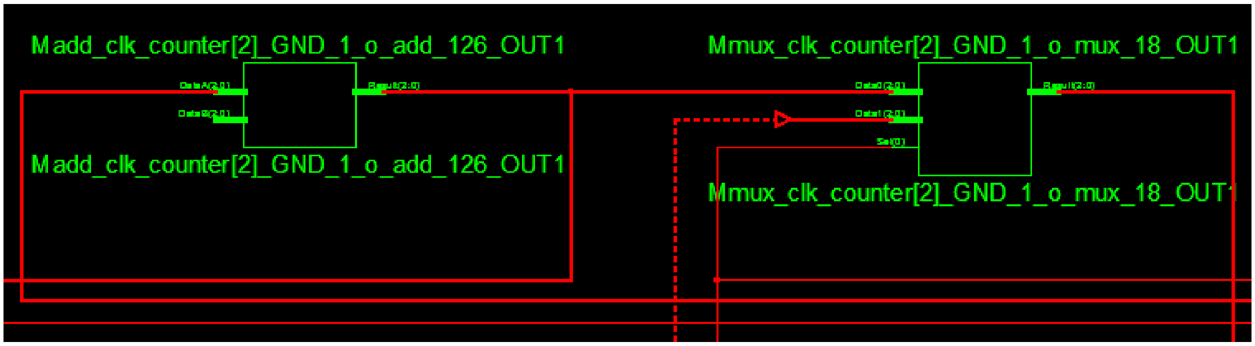

Madd_clk_counter[2]_GND_1_o_add_126_OUT1
DataA(2:0)
DataB(2:0)
Result(2:0)
Madd_clk_counter[2]_GND_1_o_add_126_OUT1
Mmux_clk_counter[2]_GND_1_o_mux_18_OUT1
Data0(2:0)
Data1(2:0)
Sel(0)
Result(2:0)
Mmux_clk_counter[2]_GND_1_o_mux_18_OUT1

Figure 3: An adder for the clock counter signal `clk_counter`, which is then passed on to a 2 to 1 3 bit multiplexer to determine if the counter should be reset or not to 1 (the value in DataB, the dotted red line). This would be used in the wait for digit input states, wait for authorization state, and wait for door to open state—all to track the number of elapsed clock cycles to determine if a timeout has occurred.

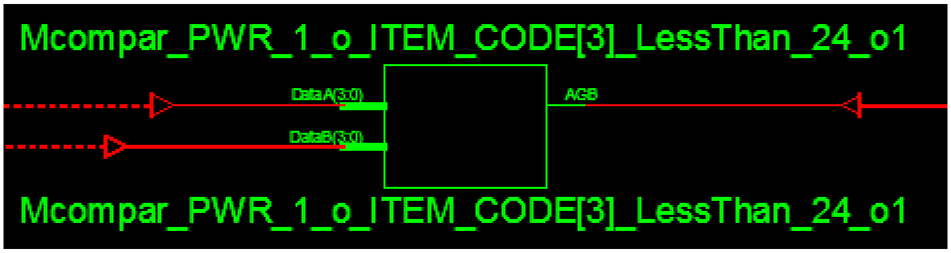

Mcompar_PWR_1_o_ITEM_CODE[3]_LessThan_24_o1
DataA(3:0)
DataB(3:0)
AGB
Mcompar_PWR_1_o_ITEM_CODE[3]_LessThan_24_o1

Figure 4: This shows a comparator block that takes in the item code input `ITEM_CODE` in DataA and a value to compare against in DataB, which for this comparator is 4’b1001 = 4’d9, corresponding to the maximum possible value of the second digit of the item code. Thus, this comparator would be used in the wait for second digit state. Another similar comparator exists that compares the `ITEM_CODE` against 4’d1 for the wait for first digit state.

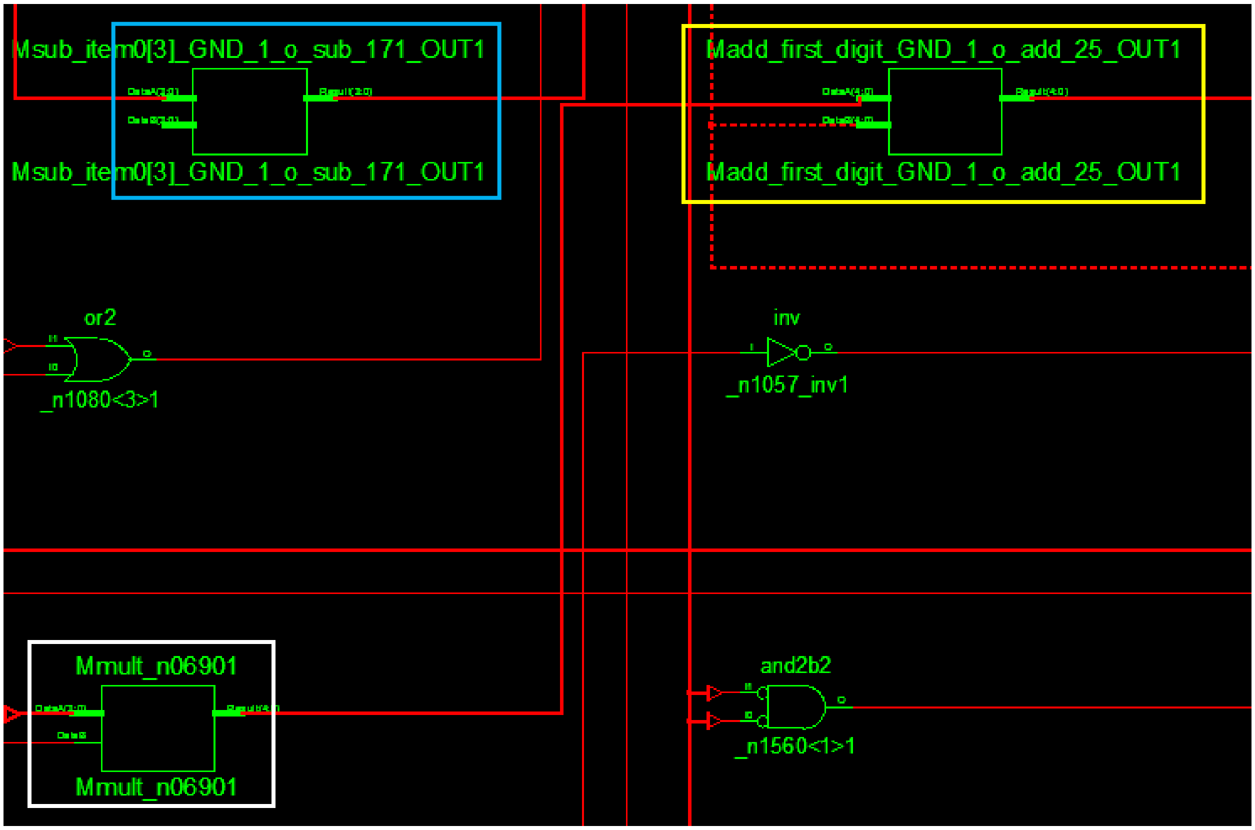


Figure 5: A 4x1 5 bit output multiplier (white) that has its output connected to an adder (yellow). This is present because the selected item number is calculated in the wait for second digit state after it has been confirmed that the inputted digit was valid using the formula `selected_item` = `first_digit` * 10 + `ITEM_CODE`, where `ITEM_CODE` is the second digit. Thus, this necessitates the need for a multiplier. A subtractor for `item0` can also be seen (blue), which decrements the counter for `item0` by 1 if `item0` is vended.

Note: Since the first digit can only be 0 or 1, a case statement or if-else statement on the first digit could probably have been used instead of multiplication to calculate the value of the selected item from the first and second digits. However, it is interesting to see that a multiplier is synthesized.

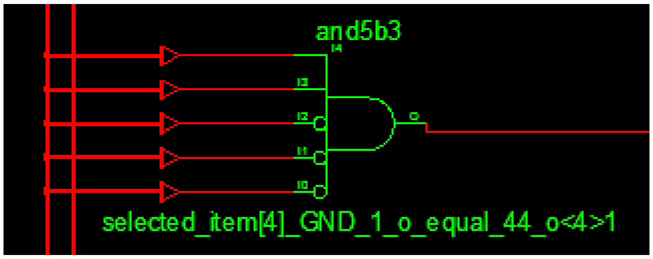


Figure 6: An AND gate to test if the selected item was item 3 (5'b00011). This could be used in the validate item selection state to determine if item 3 should be checked to see if its counter is 0, the wait for authorization state to determine the cost of the selected item, or the wait for door open state to determine which item's counter to decrement.

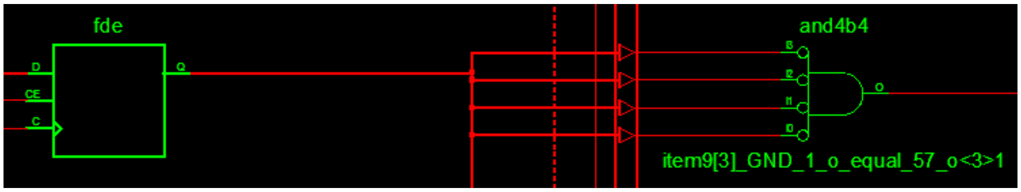


Figure 7: A 4 bit register holding the value of `item9` on the left that has each bit put through an AND gate with inverted inputs to test if the `item9`'s value is 0. This is checking if the `item9` counter is 0, which would indicate an invalid item selection, and is part of the validate item state.

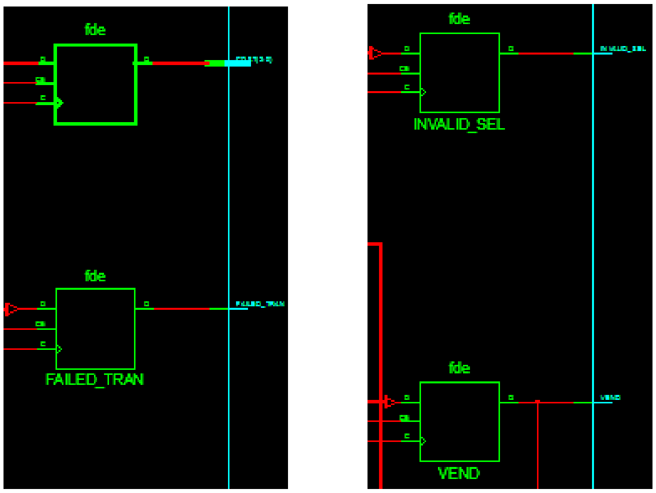


Figure 8: Registers to store the output values so that the outputs stay at a particular value until the value in the register changes. All four outputs (`COST`, `VEND`, `INVALID_SEL`, `FAILED_TRAN`) can be seen here. The top register on the left picture is for storing the 3 bit value of `COST`.

Simulation Documentation

The requirement for the vending machine module is that it can successfully complete a transaction and handle unexpected inputs, adhering to the behavior specified in the introduction. It should take in the appropriate inputs as specified in Table 1 and give the correct outputs according to Tables 2 and 4. 14 tests were performed, which tested the behavior of each state, as well as special cases. They are summarized in the table on the next page. During each test, an assert macro was used to confirm that outputs matched what was expected, and manual inspection was used to confirm that the internal variables, such as `state`, item counters, and `selected_item` matched what was expected. The subsections that follow Table 5 describe each test in detail, as well as the test bench code written for it and the simulation results. Unless otherwise specified, each test begins with the machine in the idle state and all inputs as low. Parameters `LOW` and `HIGH` were also declared, corresponding to 1'b0 and 1'b1, respectively.

```
---------------------------assert Macro Code-------------------------
`define assert(signal, value) \
  if (signal !== value) begin \
    $display($time, " ASSERTION FAILED: \
    Expected %b, got %b", value, signal); \
    $finish; \
  end
----------------------------------------------------------------------
```

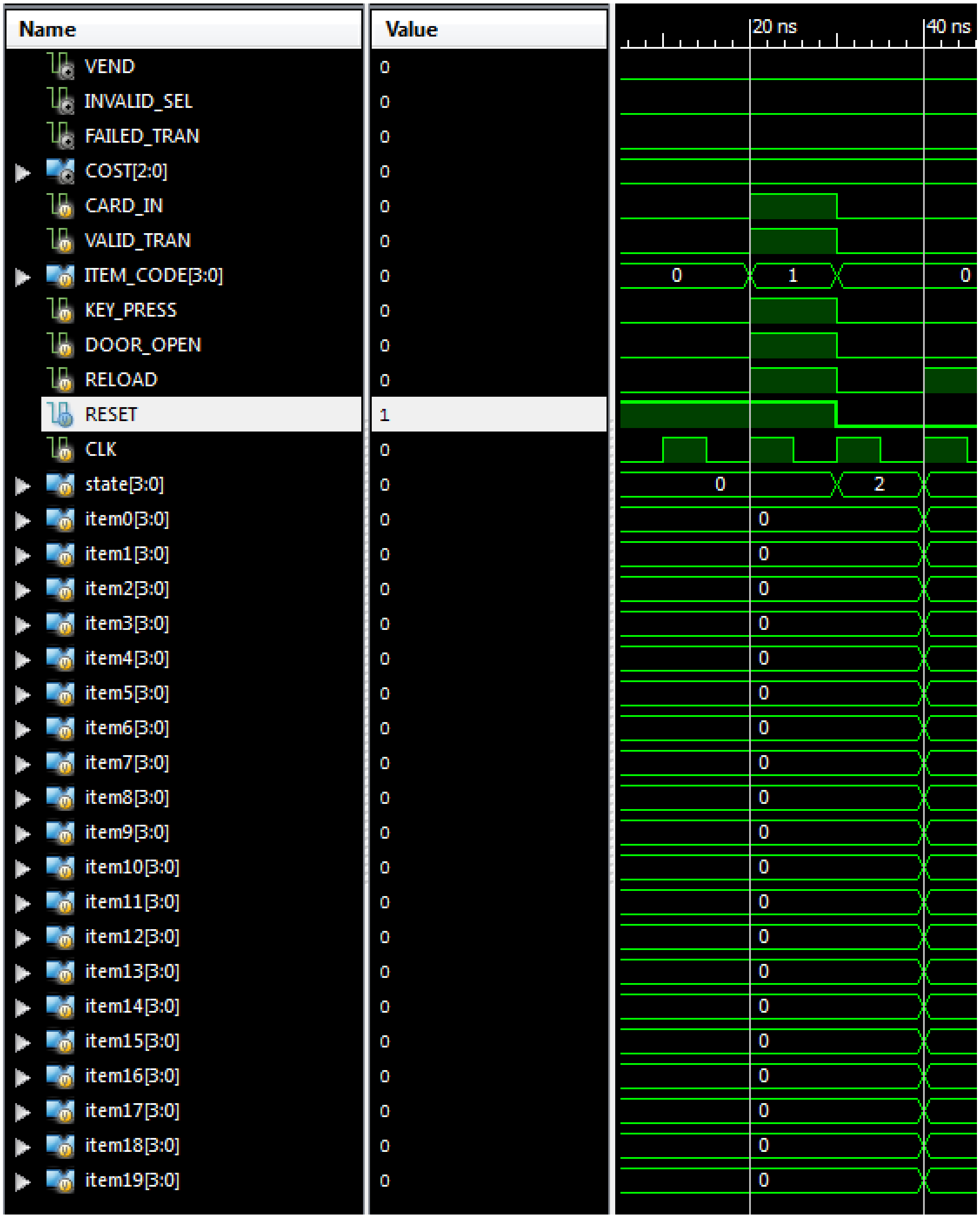
Test #	Test Focus	Description
1	Reset state behavior	Confirm outputs and item counters are 0 and that vending machine stays in reset state until `RESET` goes low
2	Reload state behavior	Confirm that item counters are 10 and that vending machine stays in reload state until `RELOAD` goes low
3	Normal transaction	Provide inputs for a successful transaction involving item 10
4	Wait for digit 1 state behavior	Confirm this state properly handles input, both valid and invalid, for the first digit
5	Wait for digit 2 state behavior	Confirm this state handles input, both valid and invalid, for the second digit. Test `CARD_IN` going low in middle of transaction
6	Validate input state behavior	Confirm this state properly handles cases when the selected item has a nonzero or zero quantity left
7	Waiting for authorization state behavior	Confirm this state properly handles authorization inputs depending on the value of `VALID_TRAN` (including if it does not arrive within 5 clock cycles) and outputs the correct cost
8	Wait for door open state behavior	Confirm this state properly handles waiting for door to open after item is vended, decrements item counter, sets `VEND` to high
9	Wait for door close state behavior	Confirm this state properly handles waiting for the door to close after it is opened
10	Item prices	Confirm the `COST` output is correct for every item
11	Item out of stock	Repeatedly vend a single item, starting from 10 left in the machine, until it runs out to confirm proper behavior
12	Irrelevant inputs at each state	Confirm that each state only responds to the inputs given in the state diagram in Figure 1
13	All inputs on the entire time	Confirm that if all inputs are always on (except for `RESET` and `RELOAD`), the transaction proceeds as normal
14	Reset overrides all other inputs	Confirm that the reset state can be reached from any other state when `RESET` goes high

Table 5: List of tests and their intended purpose. The description is a high level description that will be further elaborated on in each subsection for that test.

Test 1: Reset State Behavior

The purpose of this test is to ensure that when the `RESET` signal becomes high, the reset state (0) is entered, all outputs are 0, all item counters are 0, and that the machine stays in this state until `RESET` becomes low, where it should enter the idle state (2). No other inputs should have an effect. The numbers in parentheses indicate the number corresponding to that state and are taken from Table 3. The test bench code to do this is given below, followed by the result.

```
----------------------Reset State Testbench Code----------------------
RESET = HIGH; //enter reset state   | RELOAD = HIGH;
#20; //confirm it stays in reset_s  | #10; //should still be in reset_s
CARD_IN = HIGH;                     | `assert(VEND, LOW);
VALID_TRAN = HIGH;                  | `assert(COST, 3'b0);
ITEM_CODE = HIGH;                   | `assert(INVALID_SEL, LOW);
KEY_PRESS = HIGH;                   | `assert(FAILED_TRAN, LOW);
DOOR_OPEN = HIGH;                   |
----------------------------------------------------------------------
```



Name	Value
VEND	0
INVALID_SEL	0
FAILED_TRAN	0
COST[2:0]	0
CARD_IN	0
VALID_TRAN	0
ITEM_CODE[3:0]	0
KEY_PRESS	0
DOOR_OPEN	0
RELOAD	0
RESET	1
CLK	0
state[3:0]	0
item0[3:0]	0
item1[3:0]	0
item2[3:0]	0
item3[3:0]	0
item4[3:0]	0
item5[3:0]	0
item6[3:0]	0
item7[3:0]	0
item8[3:0]	0
item9[3:0]	0
item10[3:0]	0
item11[3:0]	0
item12[3:0]	0
item13[3:0]	0
item14[3:0]	0
item15[3:0]	0
item16[3:0]	0
item17[3:0]	0
item18[3:0]	0
item19[3:0]	0

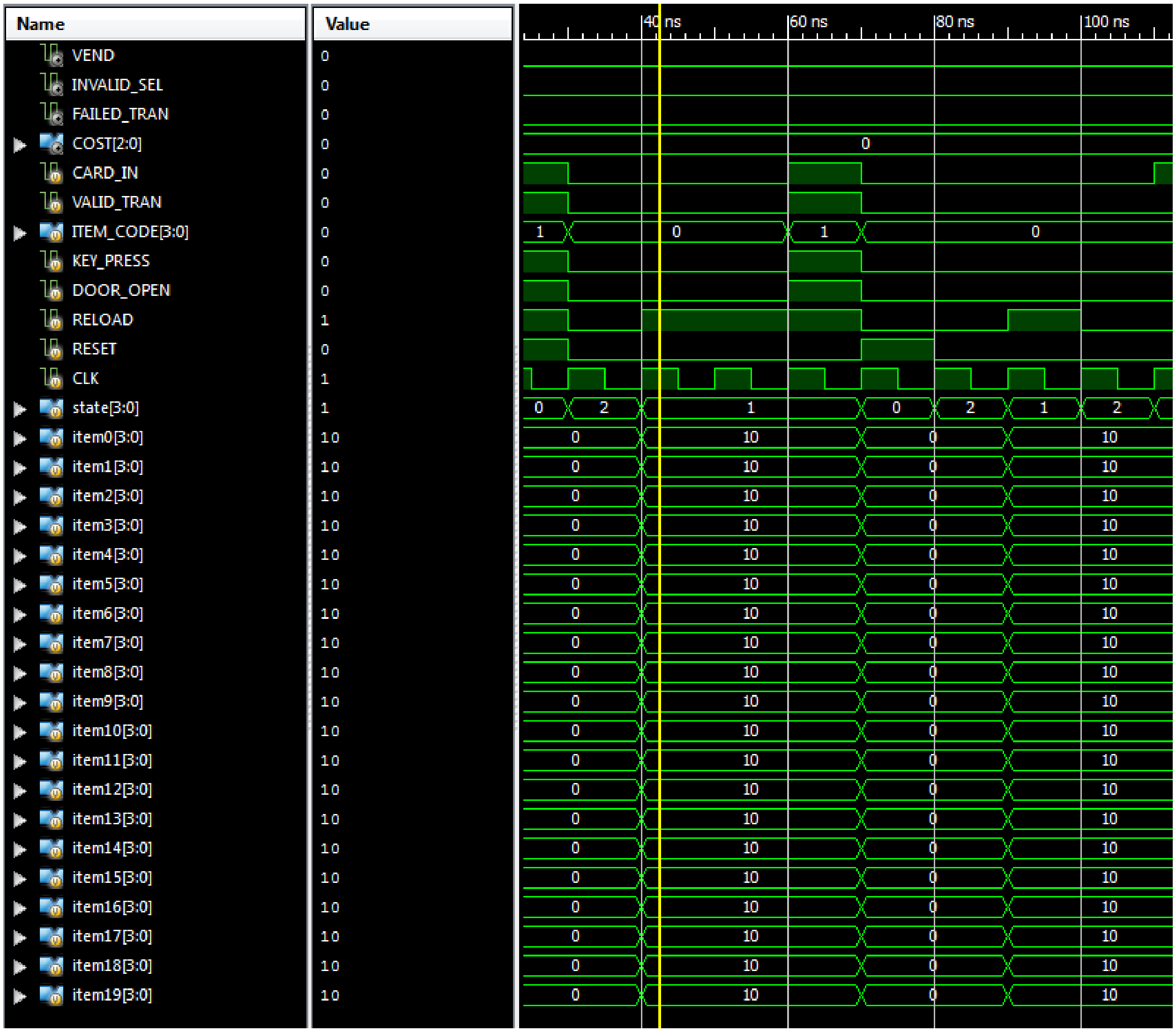
Figure 9: Simulation waveform results for testing the reset state. The state variable is called `state`. When the reset signal `RESET` is on, it enters the reset state (0) and all item counters and outputs are 0, and turning on other inputs has no effect. Once `RESET` goes low, the state goes back to the idle state (2), as expected.

Test 2: Reload State Behavior

The purpose of this test is to ensure that when the RELOAD signal goes high from the idle state (2), the vending machine enters the reload state (1) and stays there until RELOAD goes low, in which case it should return to the idle state. Alternatively, if the RESET signal goes high while in the reload state (1), the state should change to the reset state (0). Also, the item counters should all become 10, and no other inputs other than RESET should have an effect.

```
----------------------Reload State Testbench Code---------------------
RELOAD = HIGH; //enter reload_s
#20; // confirm stays in reload_s
// CHECK: counters are 10

// confirm no other inputs except
// RESET take effect
CARD_IN = HIGH;
VALID_TRAN = HIGH;
ITEM_CODE = HIGH;
KEY_PRESS = HIGH;
DOOR_OPEN = HIGH;
#10;
`assert(VEND, LOW);
`assert(COST, 3'b0);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);

// only signal that takes effect
RESET = HIGH;

// zero out all other outputs
CARD_IN = LOW;
VALID_TRAN = LOW;
ITEM_CODE = LOW;
KEY_PRESS = LOW;
DOOR_OPEN = LOW;
RELOAD = LOW;
#10;
//check counters are 0

RESET = LOW; //enter idle state
#10;

RELOAD = HIGH;
#10; //reload (for real this time)

RELOAD = LOW; //go back to idle_s
#10;
// CHECK: counters are 10
----------------------------------------------------------------------
```

(report continues on next page)



Name	Value
VEND	0
INVALID_SEL	0
FAILED_TRAN	0
COST[2:0]	0
CARD_IN	0
VALID_TRAN	0
ITEM_CODE[3:0]	0
KEY_PRESS	0
DOOR_OPEN	0
RELOAD	1
RESET	0
CLK	1
state[3:0]	1
item0[3:0]	10
item1[3:0]	10
item2[3:0]	10
item3[3:0]	10
item4[3:0]	10
item5[3:0]	10
item6[3:0]	10
item7[3:0]	10
item8[3:0]	10
item9[3:0]	10
item10[3:0]	10
item11[3:0]	10
item12[3:0]	10
item13[3:0]	10
item14[3:0]	10
item15[3:0]	10
item16[3:0]	10
item17[3:0]	10
item18[3:0]	10
item19[3:0]	10

Figure 10: Simulation waveform results for reload state test. The state goes from idle (2) to reload (1), then to reset (0) because `RESET` goes high, and then back to idle (2) and into reload (1) again before returning to idle (2). The item counters get updated to 10 during reload.

Note: From this point forward, not all item counters will be shown so that less space is taken up by the waveform simulation result pictures. Only those relevant to the test will be shown.

Test 3: Normal Transaction

The purpose of this test is to mimic a normal transaction that is successful, thus ensuring that every state behaves as expected when correct ad timely inputs. The transaction starts from idle (2), no error state (9 or 10) should be reached nor should `INVALID_SEL` or `FAILED_TRAN` ever go high, and it should proceed through all states associated with a transaction (3 to 8) before returning to idle. Item 10 was used for this test, so its quantity should decrease from 10 to 9 and the `COST` output should be 4'd3. The test bench code to do this is given in the next page.

```
-------------------Normal Transaction Testbench Code------------------
//go to wait_digit1_s
CARD_IN = HIGH;
#10;

//go to wait_digit2_s
KEY_PRESS = HIGH;
ITEM_CODE = 4'd1;
#10;

ITEM_CODE = 4'd0; //second digit
#10;

//pass through input validation
#10;
`assert(COST, 4'd3);
//check cost is correct
VALID_TRAN = HIGH;
#10;
`assert(VEND, HIGH); //check VEND
// is high; item10 should be 9

DOOR_OPEN = HIGH;
#10;

DOOR_OPEN = LOW;
#10; //ends transaction, back to
// idle state
`assert(VEND, LOW); //all outputs
// should be low in idle state
`assert(COST, 3'b0);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
----------------------------------------------------------------------
```

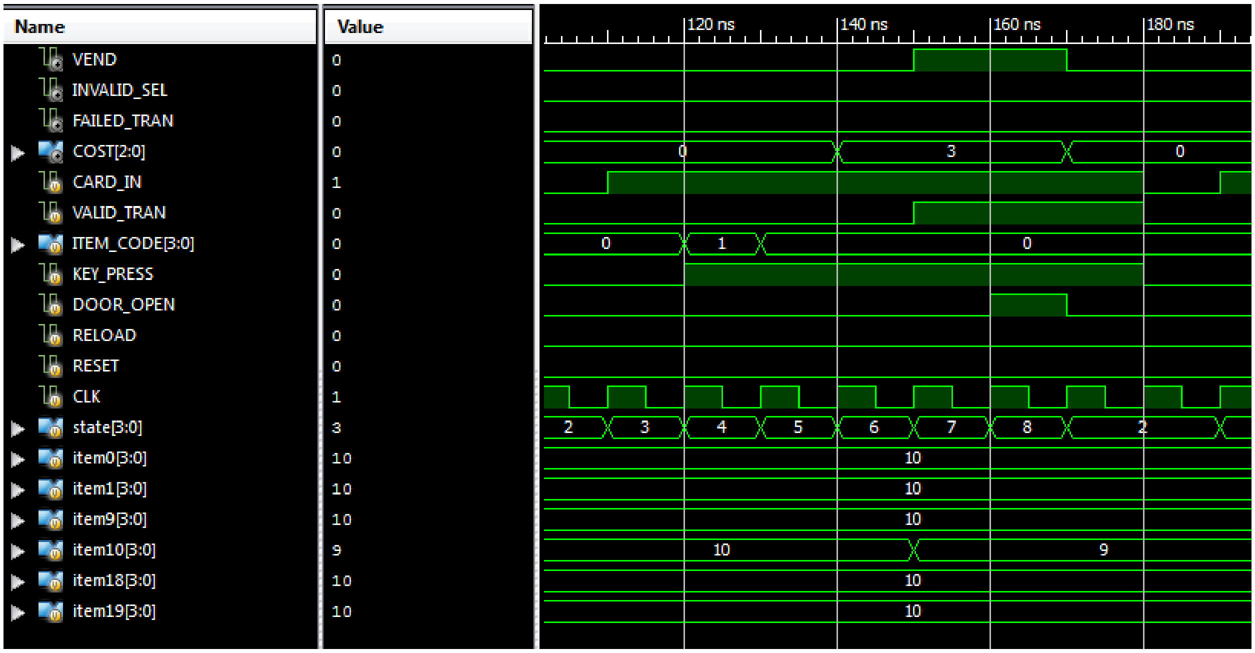
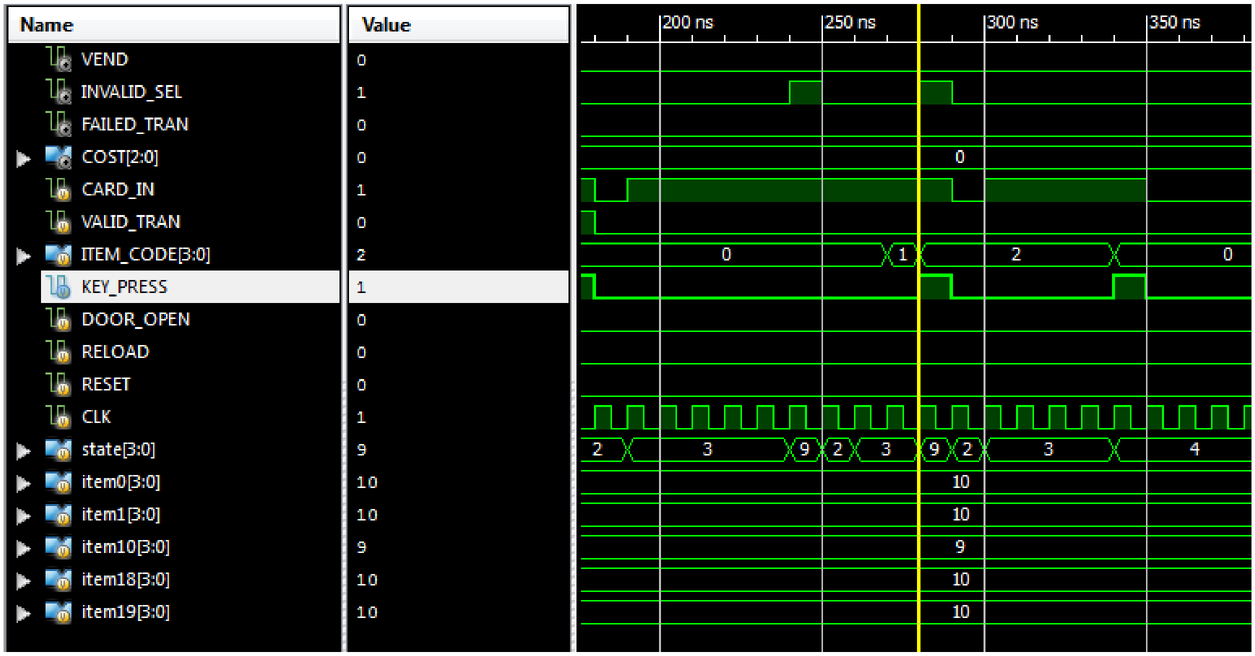


Figure 11: Simulation waveform results for a normal transaction involving item 10. The transaction begins in the idle state (2) and goes through each state that follows in a valid transaction (3 to 8) before returning to the idle state. After authorization is received and it enters the wait for door open state (7), it vends item 10, thus decreasing its quantity from 10 to 9, and also sets `VEND` to high The cost is 3, which matches what is expected.

Test 3: Wait for Digit 1 State

The purpose of this test is to test possible outward transitions that result from the wait for digit 1 state (3). This includes a digit not being entered within 5 clock cycles and an invalid digit being entered (not 4’d0 or 4’d1), which should go to the invalid input state (9). In this case, INVALID_SEL should become high for 1 clock cycle before returning to low because the idle state has been entered. If an ITEM_CODE is provided but KEY_PRESS is not high, the vending machine should treat it as no input and keeping waiting for an input. Finally, if KEY_PRESS is high and ITEM_CODE is valid, it should move on to the wait for digit 2 state (4) and INVALID_SEL should not be high. We should be able to wait for up to 4 clock cycles to give a valid input on the 5th clock cycle.

```
-----------------Wait for Digit 1 State Testbench Code-----------------
CARD_IN = HIGH; //keep this high
  //the entire time
#50; //cause timeout, should go to
  //invalid input then idle

#10; // go to invalid input
`assert(INVALID_SEL, HIGH);

#10; //exit invalid selection
state
`assert(VEND, LOW);
`assert(COST, 3'b0);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);

#10; // CARD_IN still high, so go
// back to wait for digit

ITEM_CODE = 4'd1; //KEY_PRESS not
// high so this shouldn't have an
// effect
#10;
`assert(VEND, LOW); //all outputs
// should be low
`assert(COST, 3'b0);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);

ITEM_CODE = 4'd2;
KEY_PRESS = HIGH; //this should go
// through but is invalid
#10; //go to invalid state
`assert(INVALID_SEL, HIGH);

KEY_PRESS = LOW;
CARD_IN = LOW;
#10; // go to idle state
`assert(INVALID_SEL, LOW);
//invalid selection should go back
//to low

CARD_IN = HIGH;
#10;

#30;
`assert(INVALID_SEL, LOW); //can
// wait up to 4 clock cycles for
// input digit
// 10 ns before + 30 ns now

ITEM_CODE = 4'd0;
KEY_PRESS = HIGH; //provide valid
inputs this time, should move on
to second digit
#10;
`assert(INVALID_SEL, LOW);
----------------------------------------------------------------------
```



Name	Value
VEND	0
INVALID_SEL	1
FAILED_TRAN	0
COST[2:0]	0
CARD_IN	1
VALID_TRAN	0
ITEM_CODE[3:0]	2
KEY_PRESS	1
DOOR_OPEN	0
RELOAD	0
RESET	0
CLK	1
state[3:0]	9
item0[3:0]	10
item1[3:0]	10
item10[3:0]	9
item18[3:0]	10
item19[3:0]	10

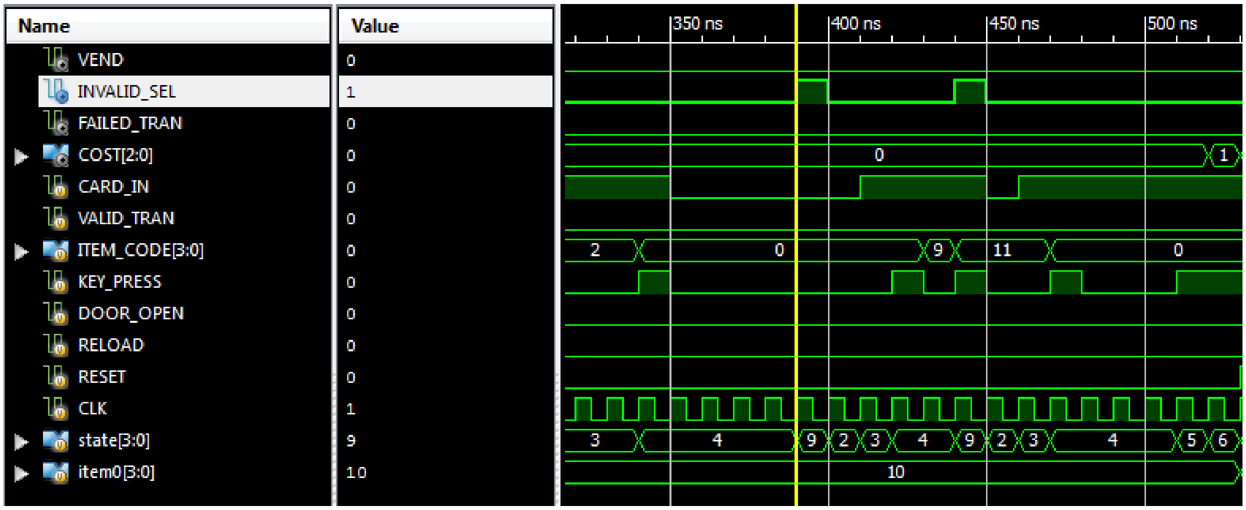
Figure 12: Simulation waveform results for testing the waiting for digit 1 state. We start in the idle state (2), and then time out waiting for a digit (the long 3) after 5 clock cycles, thus proceeding to the invalid selection state (9). Starting over, we enter an `ITEM_CODE`, specifically 4'd1, but `KEY_PRESS` is kept off, so it doesn't have an effect. Then, `ITEM_CODE` becomes 4'd2, which is an invalid input, so it enters the invalid state (9) again. Finally, we restart, and providing `ITEM_CODE` as 4'd0 with `KEY_PRESS` set to high in 4 clock cycles causes a transition into the wait for digit 2 state (4).

Test 5: Wait for Digit 2 State

The purpose of this test is to test possible outward transitions that result from the wait for digit 2 state (4). The tests performed are almost the exact same as that performed for the wait for digit 1 state. This includes a digit not being entered within 5 clock cycles and an invalid digit being entered (> 4'd9), which should go to the invalid input state (9). In this case, `INVALID_SEL` should become high for 1 clock cycle before returning to low because the idle state has been entered. If an `ITEM_CODE` is provided but `KEY_PRESS` is not high, the vending machine should treat it as no input. Finally, if `KEY_PRESS` is high and `ITEM_CODE` is valid, it should move on to the validate input state (5) and `INVALID_SEL` should not be high. The test bench code for this state directly follows from testing the previous state, so we start off in `wait_digit1_s`. This test bench code also sets `CARD_IN` to low to ensure that the transaction continues despite the credit card being taken out.

(report continues on next page)

```
-----------------Wait for Digit 2 State Testbench Code----------------
KEY_PRESS = LOW;
CARD_IN = LOW; // "take out" card
#40; //cause timeout (40 ns here +
// 10 ns earlier)

 #10;
`assert(INVALID_SEL, HIGH);

#10; //back to idle
`assert(INVALID_SEL, LOW);

CARD_IN = HIGH;
#10;

KEY_PRESS = HIGH;
ITEM_CODE = 4'd0; //first digit
#10;
`assert(INVALID_SEL, LOW);
//first digit should be valid
KEY_PRESS = LOW;
ITEM_CODE = 4'd9; // no effect
#10;
`assert(INVALID_SEL, LOW);
//stay in same state

KEY_PRESS = HIGH;
ITEM_CODE = 4'd11; //outside range
#10;
`assert(INVALID_SEL, HIGH);
//second digit invalid
CARD_IN = LOW;
KEY_PRESS = LOW;
#10;
`assert(INVALID_SEL, LOW);
//back to idle state

// start new transaction
CARD_IN = HIGH;
#10;

KEY_PRESS = HIGH;
ITEM_CODE = 4'd0; //first digit
#10;
`assert(INVALID_SEL, LOW);

KEY_PRESS = LOW;
#30; //up to 4 clock cycles (10 ns
// before + 30 ns here) acceptable
`assert(INVALID_SEL, LOW);
//should still be waiting for
//second digit

ITEM_CODE = 4'd0; //second digit
KEY_PRESS = HIGH;
#10;
`assert(INVALID_SEL, LOW);//valid
----------------------------------------------------------------------
```



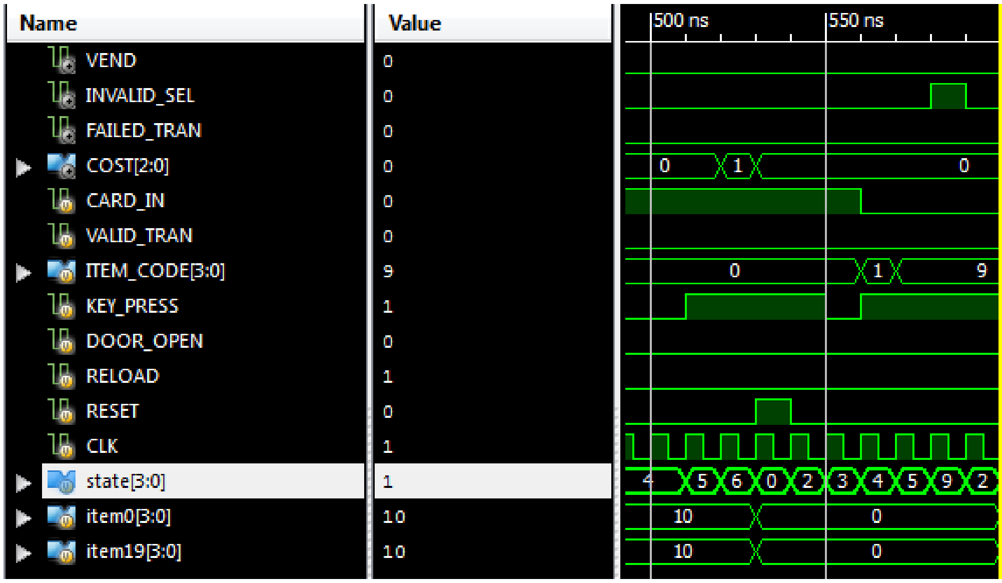
Name	Value
VEND	0
INVALID_SEL	1
FAILED_TRAN	0
COST[2:0]	0
CARD_IN	0
VALID_TRAN	0
ITEM_CODE[3:0]	0
KEY_PRESS	0
DOOR_OPEN	0
RELOAD	0
RESET	0
CLK	1
state[3:0]	9
item0[3:0]	10

Figure 13: Simulation waveform results for testing wait for digit 2 state (4). It goes to the invalid input state (9) after waiting for 50 ns, as well as when an invalid input is given; then it checks that when `KEY_PRESS` is low, the `ITEM_CODE` input has no effect and then sets `KEY_PRESS` to high with a valid digit input to reach the validate item selection input state (5).

Test 6: Validate Item Selection State

The purpose of this test is to confirm the validate item selection state (5) properly handles cases where the selected item (already validated to be between 0 and 19) is in stock or is out of stock. It continues directly from the previous test, where item 0 was selected and currently has a quantity of 10, to confirm that it works for the case of an item being in stock and moves on to the wait for authorization state (6). The `RESET` input is then used to zero the item counters, and the test is rerun to confirm that if the item is out of stock, it sends the vending machine to the invalid item selection state (9) and `INVALID_SEL` is set to high.

```
--------------Validate Item Selection State Testbench Code------------
// from wait_digit2_s, item 00
#10;
`assert(INVALID_SEL, LOW);
//item selection should be valid

RESET = HIGH;
//set all counters to 0
#10;
RESET = LOW; //turn off reset
#10;

CARD_IN = HIGH; //new transaction
VALID_TRAN = LOW;
ITEM_CODE = LOW;
KEY_PRESS = LOW;
DOOR_OPEN = LOW;
#10;
CARD_IN = LOW;
KEY_PRESS = HIGH;
ITEM_CODE = 4'd1; //first digit 1
#10;
`assert(INVALID_SEL, LOW);
ITEM_CODE = 4'd9; //second digit 9
#10;
`assert(INVALID_SEL, LOW);
//selected item 19

#10; //validate input
`assert(INVALID_SEL, HIGH);
//invalid because out of items
----------------------------------------------------------------------
```



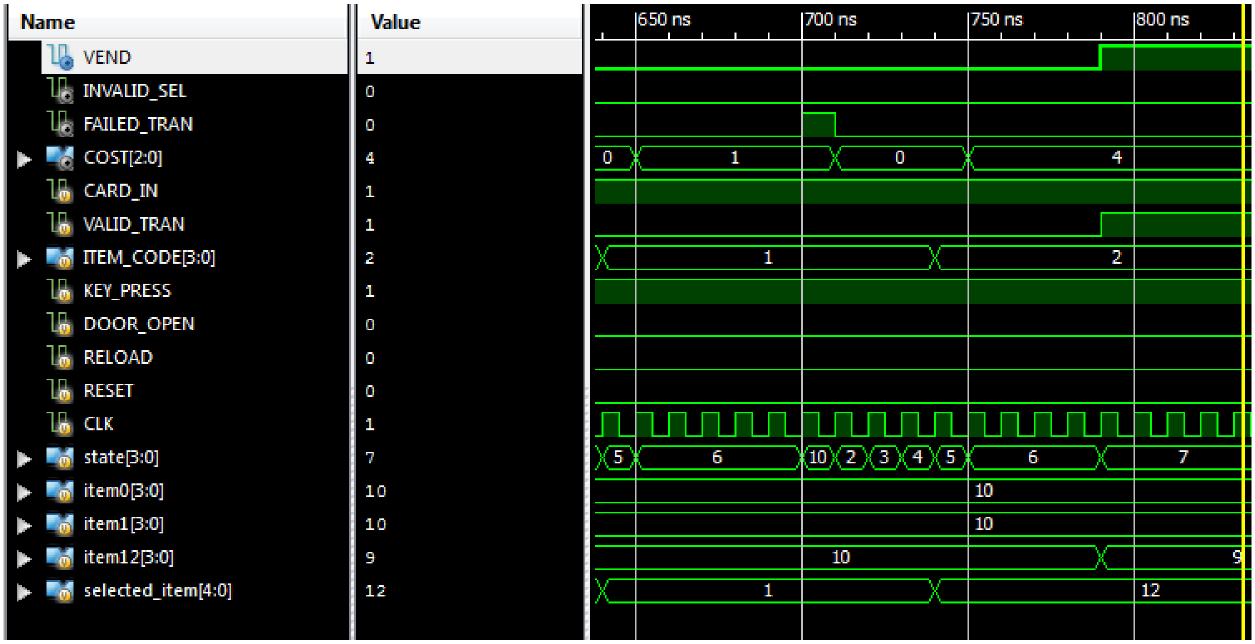
Name	Value
VEND	0
INVALID_SEL	0
FAILED_TRAN	0
COST[2:0]	0
CARD_IN	0
VALID_TRAN	0
ITEM_CODE[3:0]	9
KEY_PRESS	1
DOOR_OPEN	0
RELOAD	1
RESET	0
CLK	1
state[3:0]	1
item0[3:0]	10
item19[3:0]	10

Figure 14: Simulation waveform results for testing validate item selection state (5). Item 0 has a nonzero amount left, so the selection is valid and the vending machine goes to wait for authorization state (6). After resetting the machine, trying to buy item 19 results in discovering that item is out of stock, so state becomes invalid input (9) and `INVALID_SEL` is set to high.

Test 7: Wait for Authorization State

The purpose of this test is to test possible outward transitions that result from the wait for authorization state (6), as well as the `COST` output that is set in this state. The testing of this state is very similar to testing the wait for digit 1 and 2 states, except this time we are waiting for `VALID_TRAN` to go high instead of `KEY_PRESS`. It tests for if authorization does not arrive within 5 clock cycles, in which case the invalid transaction state (10) should be entered and `FAILED_TRAN` should be set to high. This is not an invalid item selection, so `INVALID_SEL` should be low. Otherwise, the vending machine should stay in this state if less than 5 clock cycles have elapsed, and `VALID_TRAN` becomes high, it should move on to the wait for door open state. Also, in this state, the `COST` output should be set according to the item selection and information in Table 4, so this output is checked as well. `COST` should stay high until the idle state is reached, so even if the transaction is not authorized, in the failed transaction state `COST` should still be nonzero.

```
--------------Wait for Authorization State Testbench Code-------------
// from validate_item_s, item 01
#10; //go to wait_auth_s
`assert(INVALID_SEL, LOW);
//valid selection
`assert(COST, 3'd1);
//check cost of item 1 = $1
#40; //timeout waiting for
// authorization
// 10 ns before + 40 ns now

#10;
`assert(FAILED_TRAN, HIGH);
//invalid transaction
`assert(INVALID_SEL, LOW);
//and not invalid selection
`assert(COST, 4'd1);
//cost stays on until idle state

#10;
`assert(FAILED_TRAN, LOW);
// failed_tran goes back to low
// now in idle state

// new transaction, item 12
CARD_IN = HIGH;
#10; // wait for digit 1
KEY_PRESS = HIGH;
ITEM_CODE = 4'd1;
#10; //wait for digit 2
ITEM_CODE = 4'd2;
#10; // wait for verification
#10; // wait for authorization
`assert(INVALID_SEL, LOW);
// valid selection
`assert(COST, 3'd4);
// check cost of item 12

#30;
`assert(FAILED_TRAN, LOW);
// 4 clock cycles is acceptable
// (10 ns + 30 ns)

VALID_TRAN = HIGH;
#10;
`assert(FAILED_TRAN, LOW);
// authorization arrives in time
`assert(VEND, HIGH);
// VEND should stay high
----------------------------------------------------------------------
```



Name	Value
VEND	1
INVALID_SEL	0
FAILED_TRAN	0
COST[2:0]	4
CARD_IN	1
VALID_TRAN	1
ITEM_CODE[3:0]	2
KEY_PRESS	1
DOOR_OPEN	0
RELOAD	0
RESET	0
CLK	1
state[3:0]	7
item0[3:0]	10
item1[3:0]	10
item12[3:0]	9
selected_item[4:0]	12

Figure 15: Simulation waveform results for testing wait for authorization state (6). The selected item is 1, which has a `COST` output of 4’d1 when this state is entered, as expected. `VALID_TRAN` does not go high within 5 clock cycles, so the machine goes to the failed transaction state and `FAILED_TRAN` is set to high (`COST` also stays at 4’d1). Then, item 12 is selected, with a `COST` of 4’d4, as expected, and this time, `VALID_TRAN` goes high before timeout, thus causing state to update to wait for door to open (7).

Test 8: Wait for Open Door State

The purpose of this test is to test possible outward transitions that result from the wait for door to open state (7), as well as the `VEND` output that is set in this state and the one-time decrementing of the item counter. The testing of this state is very similar to testing the wait for digit 1 and 2 and wait for authorization states, except this time we are waiting for `DOOR_OPEN` to go high. It tests if the door does not open within 5 clock cycles, in which case the machine should enter the idle state (2). Otherwise, if less than 5 cycles have elapsed, the machine should stay in this state. If the door opens (`DOOR_OPEN` = 1’b1) in time, the machine should move on to the wait for door to close state (8). Also, since the transaction was authorized in the previous state, `VEND` should be set to high and the counter for the selected item should be decremented by 4’d1 to reflect the item being dispensed. However, if the customer does not open the door immediately and we stay in this state, the item counter should not be decremented again, as no further vending has occurred. `COST` should also still be high. The test bench code for this test immediately follows the test bench code for the previous test, so it starts from an authorized transaction for item 12.

```
----------------Wait for Door Open State Testbench Code---------------
// start from wait_auth_s
// purchase of item 12 authorized
DOOR_OPEN = LOW;
#40; // timeout, all inputs should
// be low (10 ns + 40ns)

#10; //idle to take effect
`assert(VEND, LOW);
`assert(COST, 3'd0);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
// should be in idle state (no
// intermediate failure state this
// time)

// start new transaction, item 02
CARD_IN = HIGH;
VALID_TRAN = LOW;
ITEM_CODE = LOW;
KEY_PRESS = LOW;
DOOR_OPEN = LOW;
#10;
KEY_PRESS = HIGH;
ITEM_CODE = 4'd0; //1st digit
#10;
`assert(INVALID_SEL, LOW);
ITEM_CODE = 4'd2; //2nd digit
#10;
`assert(INVALID_SEL, LOW);
#10; //verification should pass
`assert(INVALID_SEL, LOW);
`assert(COST, 3'd1); //check cost
VALID_TRAN = HIGH;
#10;
`assert(FAILED_TRAN, LOW);
`assert(COST, 3'd1);
//cost stays on until idle state
`assert(VEND, HIGH);

#30;
`assert(COST, 4'd1); //4 clock
// cycles acceptable (30 ns + 10 ns)
`assert(VEND, HIGH);

DOOR_OPEN = HIGH;
#10;
`assert(COST, 3'd1);
`assert(VEND, HIGH);
// both stay high until idle
----------------------------------------------------------------------
```

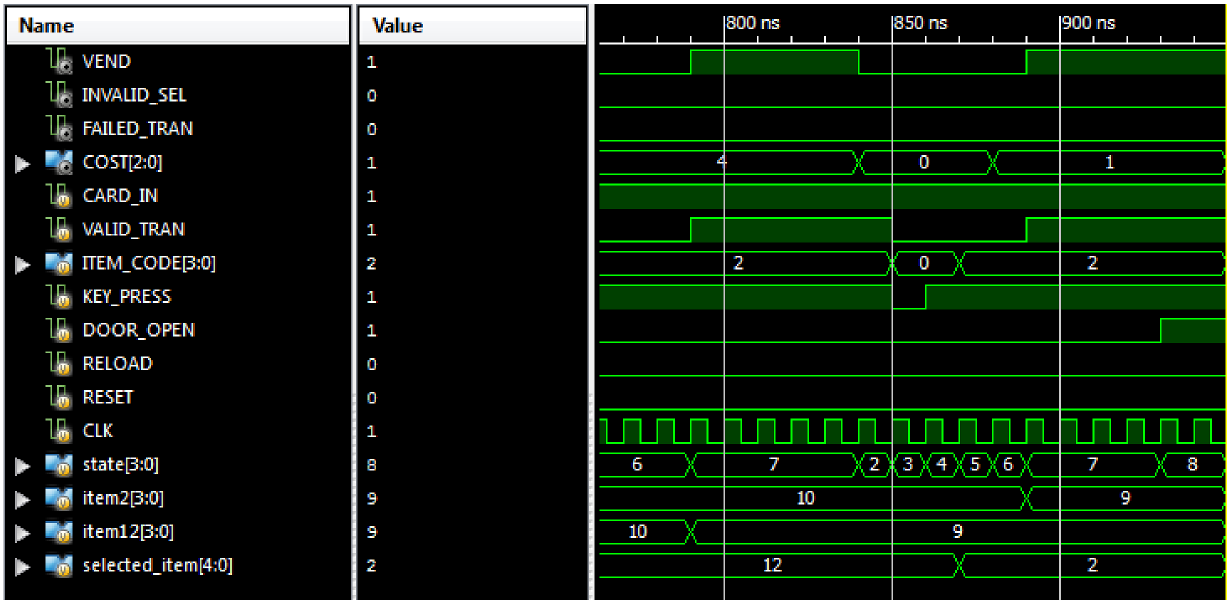


Figure 16: Simulation waveform results for wait for door to open state (7). If `DOOR_OPEN` does not go high within 5 clock cycles, state is updated to the idle state (2). Otherwise, it moves on to the wait for door to close state (8). `VEND` is set to high upon entry into this state, and the item counter is reduced by 1 (from 10 to 9 for items 2 and 12) upon initial entry into this state.

Test 9: Wait for Door Close State

The purpose of this test is to test possible outward transitions that result from the wait for door to close state (8). Once this state is entered, the machine remains in this state until DOOR_OPEN becomes low, which then causes the machine to enter the idle state (2), marking the completion of a transaction. Otherwise, if DOOR_OPEN stays high, the machine stays in this state (unless RESET also goes high). The outputs should remain unchanged, so VEND should still be high and the COST should still be displayed and be a nonzero value.

```
---------------Wait for Door Close State Testbench Code---------------
// start from wait_open_s             | DOOR_OPEN = LOW;
// vended item 2, cost of $1          | #10;
#100;                                 | `assert(VEND, LOW); //all outputs
//do nothing until door closes        | // should be low b/c idle state
`assert(COST, 4'd1);                  | `assert(COST, 3'd0);
`assert(VEND, HIGH);                  | `assert(INVALID_SEL, LOW);
//outputs remain from prev state      | `assert(FAILED_TRAN, LOW);
----------------------------------------------------------------------
```

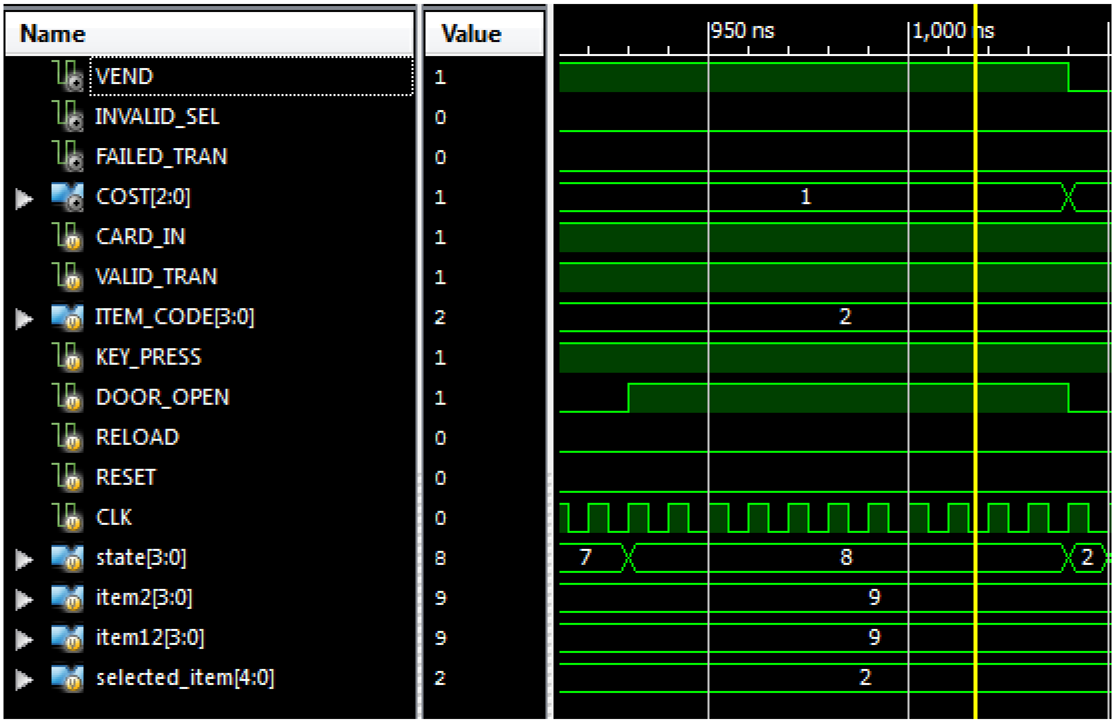
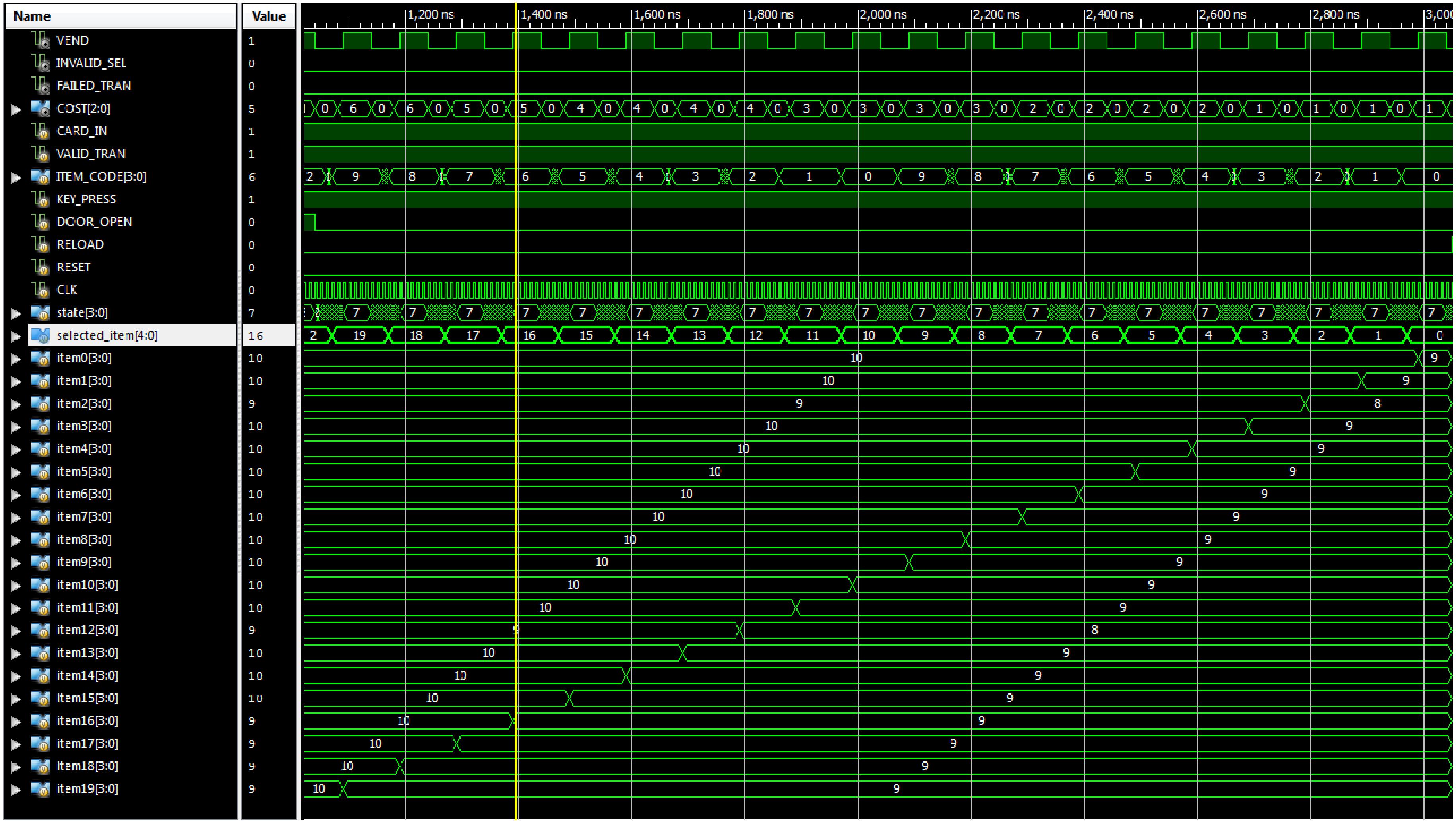


Figure 17: Simulation waveform results for testing the wait for door to close state (8). This state is entered from the wait for door to open state (7) once the DOOR_OPEN goes to high, and it remains in this state until DOOR_OPEN goes to low, where it enters the idle state (2). The outputs COST and VEND also stay at their previous values.

Test 10: Item Prices

The purpose of this test is to check the prices of all items by carrying out a transaction for each item and checking its price when it is displayed upon reaching the wait for authorization state (6). The output `COST` is compared with its expected value in Table 4. Once the cost was displayed, the transaction was authorized and then timeout was allowed to occur so that the machine went from waiting for the door to open state (7) to the idle state (2) to start another transaction. The test bench code starts from the machine being in an idle state (2). `VALID_TRAN`, `KEY_PRESS`, and `CARD_IN` were also kept high throughout the duration of this test to confirm that they only have an effect on the states specified in the state machine, and that Verilog code automatically recognizes them as being high and moves on to the next state at the next clock cycle (this behavior is mentioned is mentioned in the FAQ document). This includes automatically starting a new transaction once the machine is in the idle state.

```
--------------------Check Item Prices Testbench Code------------------
// start from idle_s
// check cost of item 19
CARD_IN = HIGH;
KEY_PRESS = HIGH;
#10;
ITEM_CODE = 4'd1;
#10;
ITEM_CODE = 4'd9;
#10;
#10;
`assert(COST, 4'd6);
VALID_TRAN = HIGH;
#50; //timeout and go to idle
#10; //idle
`assert(COST, 3'd0);

//check item 18 cost
#10; //first digit
ITEM_CODE = 4'd1;
#10; //second digit
ITEM_CODE = 4'd8;
#10; //validate
#10; //wait for auth
`assert(COST, 4'd6);
#50; //timeout and go to idle
#10;
`assert(COST, 3'd0);

// ...and so on for other items...

// check item 1 cost
#10; //first digit
ITEM_CODE = 4'd0;
#10; //second digit
ITEM_CODE = 4'd1;
#10; //validate
#10; //wait for auth
`assert(COST, 4'd1);
#50; //timeout and go to idle
#10;
`assert(COST, 3'd0);
// check item 0 cost
#10; //first digit
ITEM_CODE = 4'd0;
#10; //second digit
ITEM_CODE = 4'd0;
#10; //validate
#10; //wait for auth
`assert(COST, 4'd1);
#50; //timeout and go to idle
#10;
`assert(COST, 3'd0);
----------------------------------------------------------------------
```



Name	Value
VEND	1
INVALID_SEL	0
FAILED_TRAN	0
COST[2:0]	5
CARD_IN	1
VALID_TRAN	1
ITEM_CODE[3:0]	6
KEY_PRESS	1
DOOR_OPEN	0
RELOAD	0
RESET	0
CLK	0
state[3:0]	7
selected_item[4:0]	16
item0[3:0]	10
item1[3:0]	10
item2[3:0]	9
item3[3:0]	10
item4[3:0]	10
item5[3:0]	10
item6[3:0]	10
item7[3:0]	10
item8[3:0]	10
item9[3:0]	10
item10[3:0]	10
item11[3:0]	10
item12[3:0]	9
item13[3:0]	10
item14[3:0]	10
item15[3:0]	10
item16[3:0]	9
item17[3:0]	9
item18[3:0]	9
item19[3:0]	9

Figure 18: Simulation waveform results for testing the cost of every single item. Tracking the `selected_item` variable, we can see that it cycles through every single item, and the `COST` output near the top changes accordingly to each item's cost before being reset to 3'b0 when the idle state is reached. The quantity of each item is also decremented when the transaction is authorized, as well as the `VEND` signal going high. All the costs match the values given in Table 4, thus verifying the displayed item costs.

Test 11: Repeated Vending of an Item

The purpose of this test is to ensure that a single item can be vended multiple times, and that each time the item's quantity decreases by 1. This can continue until the item's quantity reaches 0, and then the machine should go into the invalid transaction state (9) instead and set `INVALID_SEL` to high to indicate this item is out of stock. This was achieved by using a for loop with a parameter `i` (declared at the top of the test bench) to repeatedly vend item 5. Before this test was started, the machine was reloaded to erase the effects of the previous test.

```
---------------Repeatedly Vend Single Item Testbench Code-------------
// start from idle_s
RELOAD = HIGH; //reload machine
#10;
RELOAD = LOW;
#10;
KEY_PRESS = HIGH;
for(i = 0; i < 10; i = i+1) begin
  CARD_IN = HIGH;
  #10; //first digit
  ITEM_CODE = 4'd0;
  #10; //second digit
  ITEM_CODE = 4'd5;
  #10; //validate
  #10; //wait for auth
  `assert(COST, 4'd2);
  VALID_TRAN = HIGH;
  #50; //timeout and go to idle
  #10;
  `assert(COST, 3'd0);
end // end FOR loop

// try again item 5 has none left
CARD_IN = HIGH;
#10; //first digit
ITEM_CODE = 4'd0;
#10; //second digit
ITEM_CODE = 4'd5;
#10; //validate

#10;
`assert(INVALID_SEL, HIGH);
`assert(COST, 3'd0);

#10;
`assert(INVALID_SEL, LOW);
`assert(COST, 3'd0);
----------------------------------------------------------------------
```

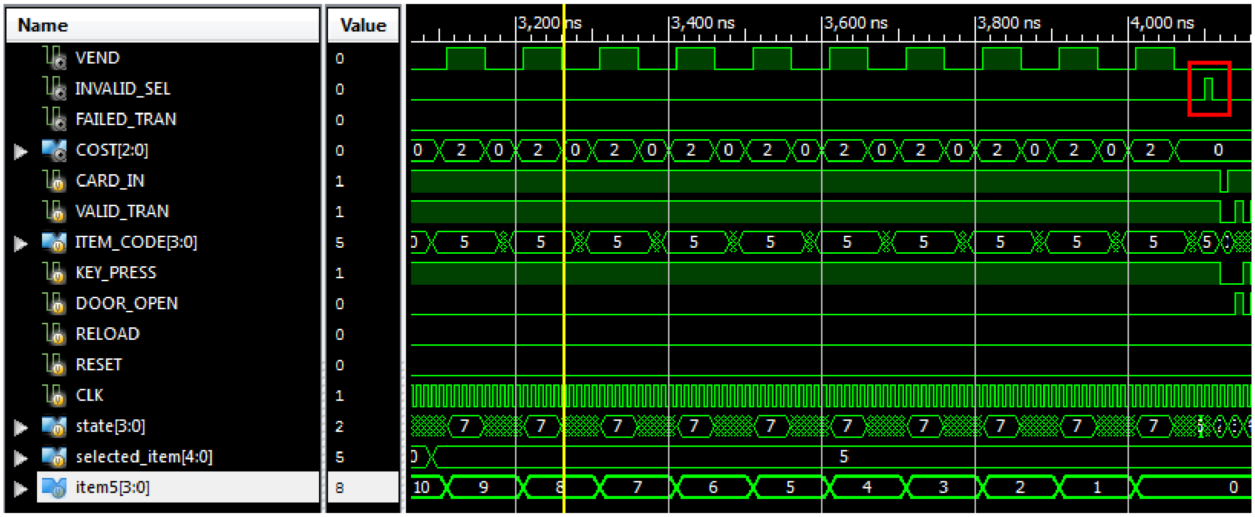
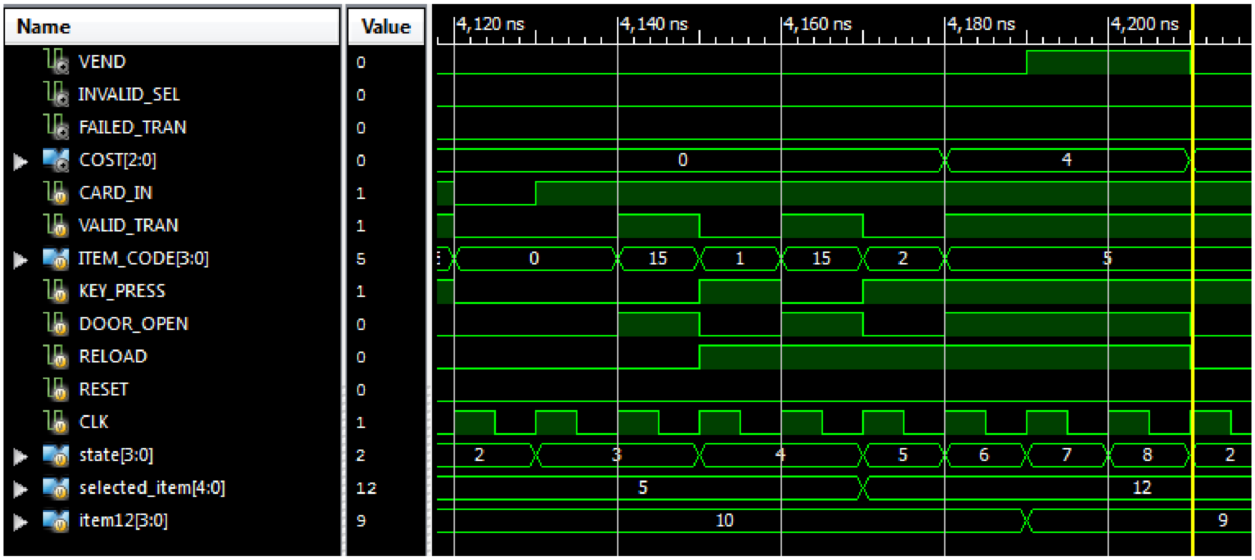


Figure 19: Simulation waveform results of repeatedly vending a single item, in this case item 5. We see that `selected_item` is 5'd5 throughout the duration of this test, and item 5's counter decreases by 1 repeatedly until it reaches 0. Then, when we attempt to buy this item again, the machine detects that there none left and sets `INVALID_SEL` to high to indicate this error (red).

Test 12: Irrelevant Inputs

The purpose of this test is to ensure that at each state, only the inputs specified on the state diagram affect each state. This is done to check the state machine does not exhibit any unexpected behavior and/or freeze up or enter some nonexistent state. Various inputs were turned on at each state (except for `RESET`) and the outputs and internal state was checked to see if they matched what was expected. This test starts in the idle state (2).

```
--------------------Irrelevant Inputs Testbench Code------------------
// start from idle_s
VALID_TRAN = LOW; //zero out
inputs
ITEM_CODE = LOW;
KEY_PRESS = LOW;
DOOR_OPEN = LOW;
CARD_IN = LOW;
#10; //stay in idle_s
// start new trasaction
CARD_IN = HIGH;
#10;
ITEM_CODE = 4'd15; //without
// KEY_PRESS, take no action
VALID_TRAN = HIGH;
DOOR_OPEN = HIGH;
#10;
`assert(VEND, LOW);
`assert(COST, 3'd0);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
VALID_TRAN = LOW;
DOOR_OPEN = LOW;
ITEM_CODE = 4'd1;
KEY_PRESS = HIGH;
RELOAD = HIGH;
#10;
`assert(INVALID_SEL, LOW);
KEY_PRESS = LOW;
ITEM_CODE = 4'd15; //without
// KEY_PRESS, take no action
VALID_TRAN = HIGH; //these inputs
// shouldn't matter
DOOR_OPEN = HIGH;
#10;
`assert(VEND, LOW);
`assert(COST, 3'd0);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
VALID_TRAN = LOW;
DOOR_OPEN = LOW;
ITEM_CODE = 4'd2; //2nd digit: 2
KEY_PRESS = HIGH;
#10; //verification state
`assert(INVALID_SEL, LOW);
//none of these
// should matter in wait_verify_s
VALID_TRAN = HIGH;
KEY_PRESS = HIGH;
ITEM_CODE = 4'd5;
DOOR_OPEN = HIGH;
#10; //wait for auth state
`assert(VEND, LOW);
`assert(COST, 3'd4); cost changes
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
#10;
`assert(VEND, HIGH);
`assert(COST, 3'd4);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
//door already open so
// proceed to next state
#10;
`assert(VEND, HIGH);
`assert(COST, 3'd4);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
DOOR_OPEN = LOW;
RELOAD = LOW;
#10;
`assert(VEND, LOW);
`assert(COST, 3'd0);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
----------------------------------------------------------------------
```



Name	Value
VEND	0
INVALID_SEL	0
FAILED_TRAN	0
COST[2:0]	0
CARD_IN	1
VALID_TRAN	1
ITEM_CODE[3:0]	5
KEY_PRESS	1
DOOR_OPEN	0
RELOAD	0
RESET	0
CLK	1
state[3:0]	2
selected_item[4:0]	12
item12[3:0]	9

Figure 20: Simulation waveform results of turning on random inputs (except `RESET`) at each state for a valid transaction. Tracing the `state` variable shows that the machine correctly progresses from the idle state (2) through every state up to waiting for the door to close (8) before returning to idle (2). All the outputs are set at the correct times and no errors are reported.

(report continues on next page)

Test 13: All Inputs On

The purpose of this test is to test what happens when all inputs except for `RESET` and `RELOAD` are kept on throughout a valid transaction. This was partially performed in test 10 but here, `DOOR_OPEN` is also kept high until the wait for door to close state (8) is reached. The transaction should proceed as normal, as the input signals being high indicates keys are being pressed, authorization is being received, and the door is being kept open. Also, because `CARD_IN` is kept on, a new transaction should start automatically once the previous one ends, so this behavior was checked for as well.

```
----------------------All Inputs On Testbench Code--------------------
// start from idle_s
CARD_IN = HIGH;
VALID_TRAN = HIGH;
DOOR_OPEN = HIGH;
KEY_PRESS = HIGH;
ITEM_CODE = 4'd1;
#10; //enter wait_digit1_s
`assert(INVALID_SEL, LOW);
#10; //enter wait_digit2_s
`assert(INVALID_SEL, LOW);
#10; //enter verification
`assert(INVALID_SEL, LOW);
#10; //enter wait for auth
`assert(INVALID_SEL, LOW);
`assert(COST, 4'd3);
#10; //enter wait for open
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
`assert(COST, 4'd3);
`assert(VEND, HIGH);
#10; //enter wait for close
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
`assert(COST, 4'd3);
`assert(VEND, HIGH);
#10;

DOOR_OPEN = LOW;
#10;
`assert(VEND, LOW); //return to
//idle, all outputs should be low
`assert(COST, 3'd0);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);

// everything is still on so it
// should start new transaction
DOOR_OPEN = HIGH;
#10; //enter wait_digit1_s
`assert(INVALID_SEL, LOW);
#10; //enter wait_digit2_s
`assert(INVALID_SEL, LOW);
#10; //enter verification
`assert(INVALID_SEL, LOW);
#10; //enter wait for auth
`assert(INVALID_SEL, LOW);
`assert(COST, 4'd3);
#10; //enter wait for open
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
`assert(COST, 4'd3);
`assert(VEND, HIGH);
#10; //enter wait for close
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
`assert(COST, 4'd3);
`assert(VEND, HIGH);

CARD_IN = LOW;
VALID_TRAN = LOW;
DOOR_OPEN = LOW;
KEY_PRESS = LOW;
ITEM_CODE = 4'd0;
#10;
`assert(VEND, LOW); //return to
//idle, all outputs should be low
`assert(COST, 3'd0);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
----------------------------------------------------------------------
```

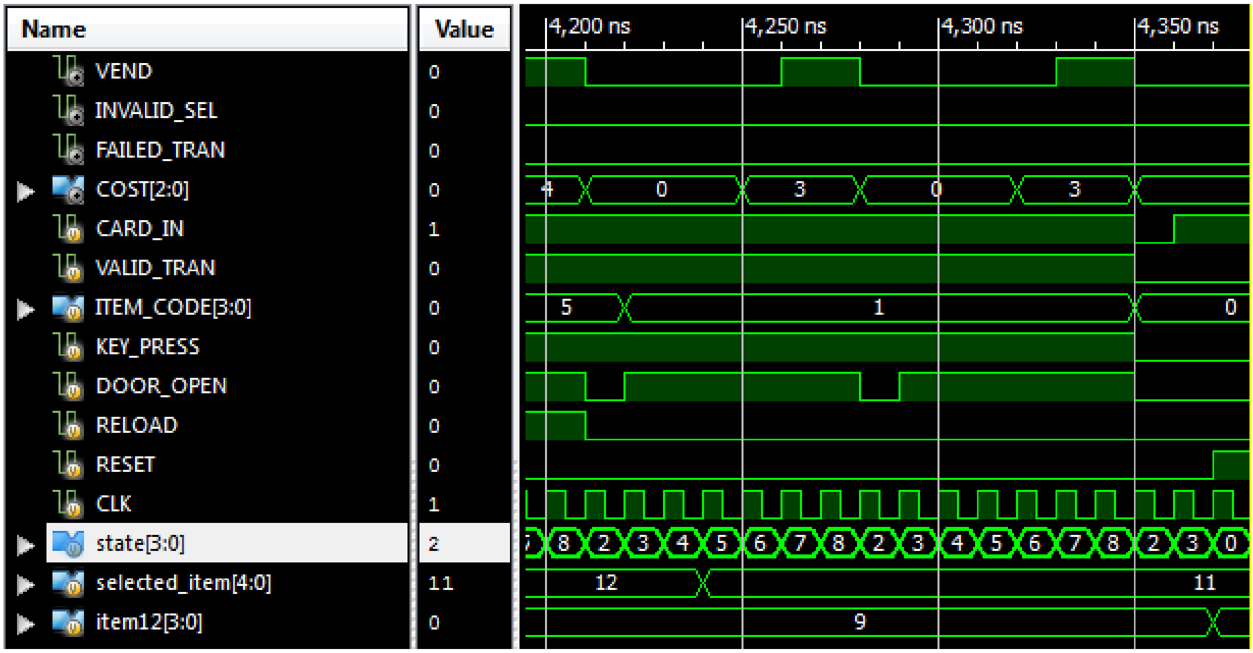


Figure 21: Simulation waveform results of testing the machine with all inputs except for RESET and RELOAD set to high the entire time, except for setting DOOR_OPEN to low for 1 clock cycle to allow for a return to the idle state (2). This shows the transaction proceeding as normal and automatically starting a new transaction from the idle state (2) after the previous one finishes, thus matching the expected behavior. The outputs are also set correctly.

Test 14: Reset From Any State

The purpose of this test is to check that the RESET signal is effective at every single state. No matter what the current state is or what other signals are on, RESET turning high should override these signals and send the machine to the reset state (0). This was done by taking the machine through various states of a transaction and then turning on the RESET signal to confirm that it was effective. This includes the invalid item selection state (9) and failed transaction states (10). To reach the states after validating the item input (5), RELOAD was applied before starting each transaction so that the item counters would be nonzero, which is the result of applying the RESET signal on high. For the sake of brevity, the code to bring the machine to each state has been replaced with a comment so that only the code relevant to RESET being set to high is shown.

(report continues on next page)

```
-------------------Reset from Any State Testbench Code----------------
// start from idle state
CARD_IN = HIGH;
//reset from wait_digit1_s
#10;
RESET = HIGH;
#10;
//confirm in reset state
RESET = LOW;
#10;
`assert(VEND, LOW); //return to
// idle, all outputs should be low
`assert(COST, 3'd0);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);

//...progress to wait_digit2_s...
KEY_PRESS = HIGH;
ITEM_CODE = 4'd1;
#10;
RESET = HIGH;
#10;
RESET = LOW;
#10;
`assert(VEND, LOW);
`assert(COST, 3'd0);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
//..progress to validate_item_s..
RESET = HIGH;
#10;
RESET = LOW;
#10;
`assert(VEND, LOW); //return to
idle, all outputs should be low
`assert(COST, 3'd0);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
//...progress to inval_input_s...
RESET = HIGH;
#10;
RESET = LOW;
#10;
`assert(VEND, LOW);
`assert(COST, 3'd0);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
```

```
//reload items to reach more states
RELOAD = HIGH;
#10;
RELOAD = LOW;
#10;
//...progress to wait_auth_s...
RESET = HIGH;
#10;
RESET = LOW;
#10;
`assert(VEND, LOW);
`assert(COST, 3'd0);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
//...reload items...
//...progress to fail_tran_s...
RESET = HIGH;
#10;
RESET = LOW;
#10;
`assert(VEND, LOW);
`assert(COST, 3'd0);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
//...reload items...
//...progress to wait_open_s...
RESET = HIGH;
#10;
RESET = LOW;
#10;
`assert(VEND, LOW);
`assert(COST, 3'd0);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
//...reload items...
//...progress to wait_close_s...
RESET = HIGH;
#10;
RESET = LOW;
#10;
`assert(VEND, LOW);
`assert(COST, 3'd0);
`assert(INVALID_SEL, LOW);
`assert(FAILED_TRAN, LOW);
----------------------------------------------------------------------
```

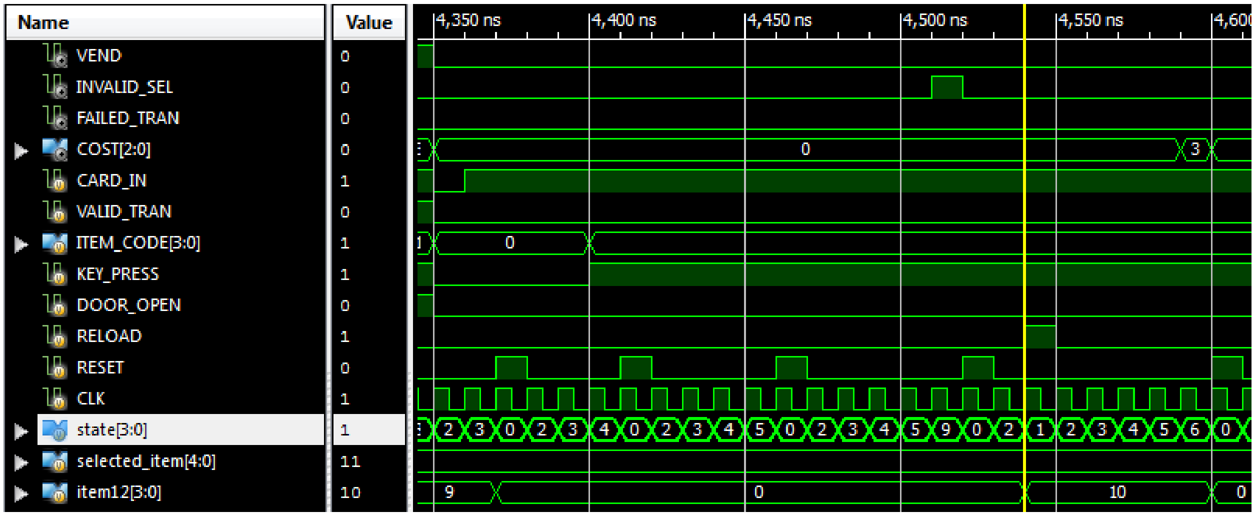


Figure 22: Simulation waveform results of setting `RESET` to high at various states. Examining the state variable shows this is being tested for every state: finding all the reset states (0) shows states 3, 4, 5, 6, and 9 preceding them, showing that `RESET` is effective for those states.

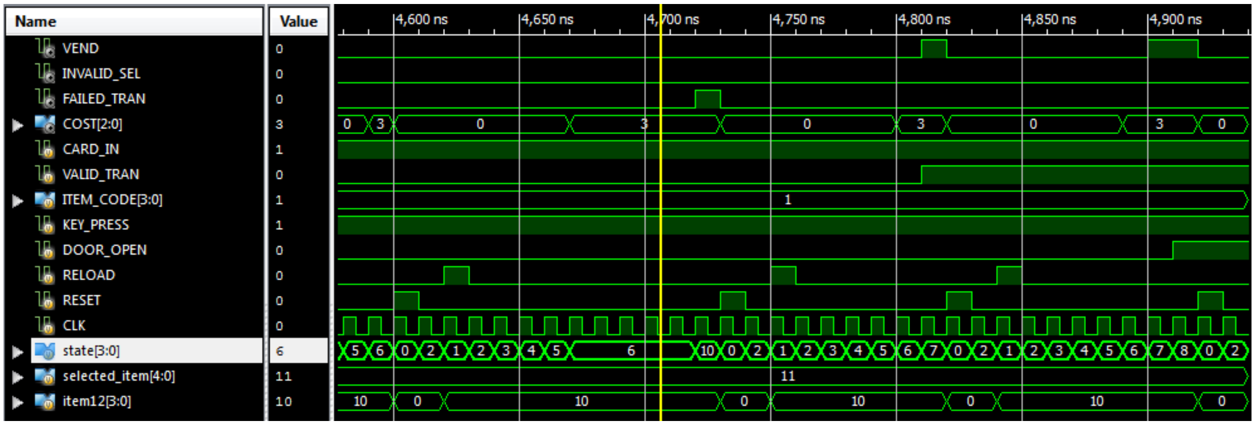


Figure 23: Simulation waveform results of setting `RESET` to high at various states. Examining the state variable shows this is being tested for every state: finding all the reset states (0) shows states 7, 8, and 10 preceding them, showing that `RESET` is effective for those states. This figure, combined with Figure 22, shows that the `RESET` signal is effective for all states (resetting when idle (2) has been tested in other tests).

After synthesis, the design summary section of the text report shows 103 of 18224 slice registers were used, as well as 217 of 9112 slice lookup tables (LUTs). Of the 225 LUT flip flop pairs used, 122 of them had an unused flip flop, 8 had an unused LUT, and 95 used both the LUT and the flip flop. There are 17 bonded IOBs, matching the total number of input and output signal bits. The advanced HDL synthesis report shows the use of, among other macros, 20 4 bit down counters (for the 20 item counters), a 3 bit adder (for the clock counter), 2 4 bit comparator greaters (for comparing if the digit is > 1 and > 9), a 4x1 to 5 bit multiplier accumulator (for calculating the selected item code from the two digits inputted), a 4 bit 13 to 1 multiplexer (for picking next state and output), and other multiplexers of varying sizes. There are only 11 possible states, so I'm not quite sure why a 13 to 1 multiplexer was synthesized.

After implementation, the mapping report showed that 103 of 18224 slice registers were used. The number of slice LUTs was reduced down to 186 of 9112. 60 of 2278 slices are occupied, and of the 187 LUT flip flop pairs used, 107 had an unused flip flop, 1 had an unused LUT, and 79 had both the LUT and flip flop used. The number of bonded IOBs remained at 17.

Overall, the LUT and slice register usage is low for both the synthesis and mapping reports, indicating that this is a feasible design to implement on a real board.

Detailed extracts of the design summary sections of the synthesis and mapping reports are given in Appendices B (synthesis report) and C (mapping report).

Conclusion

In this lab, a vending machine was modeled in Verilog that allowed users to select an item, pay by card, and open/close the door was created, as well as capabilities to handle invalid selection and credit card authorization failures and allow for the machine to be reloaded and reset. It took in various inputs to enable these operations and the transaction process, and its outputs displayed the cost of the selected item and vending status, as well as failure statuses for the 2 failures described above. This vending machine was designed by modeling it as a Moore FSM with an initial and default idle state, states for accepting input, waiting for authorization, checking if the selected item is still in stock, and waiting for the door to open or close to facilitate the transaction process, 2 failure states for an invalid selection and failed authorization, and a reset and reload state for those corresponding operations. Testing this vending machine with a test bench that covered a variety of test cases encompassing successful and failed transactions showed that the vending machine behaved as expected, thus verifying the design. Through this lab, I learned about how to model a FSM in Verilog, writing output and next state logic, as well as how to test the FSM using testbench code and Xilinx Isim software.

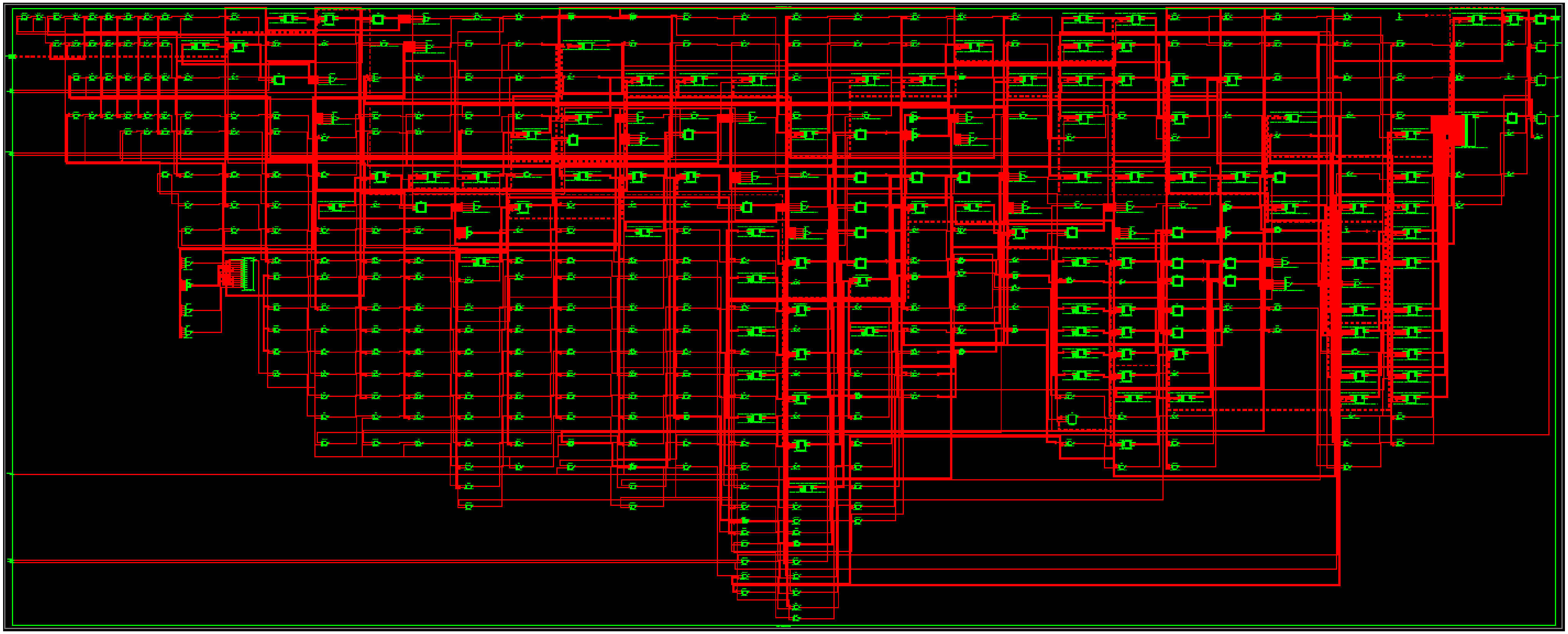
The main difficulty I encountered was figuring out how granular to make my FSM, as if I created too many states, coding and testing it would quickly become unmanageable. However, if too few states were used, I would be trying to do too much in one particular state, making it easy for me to make errors, thus making the machine more susceptible to failure or unintended behavior. I resolved this problem by thinking carefully about the state transitions and how to model them in Verilog, and from this the states naturally fit in.

Overall, the lab was interesting and hard enough to be a challenge, but not overly hard to be not doable or needlessly tedious. Doing this lab helped me reinforce the concepts I learned about state machines. However, when the project specification was initially released, it was unclear and contained a couple contradictions in its wording. This prevented me from starting on the lab early, as I had to wait for my TA to addresses these issues. However, these contradictions and unclear points have been resolved with revisions and the FAQ document, so future students will have a clearer project specification document to work with.

Appendix

Appendix A: Full RTL Schematic

The full RTL schematic shown below is essentially unreadable due to its large size, but I have included it for the sake of completeness to give the reader a sense of the full result when the code is synthesized. The inputs are on the left and the outputs are on the right. The thick red bundle of wires on the right side that converges at a rectangular block is the 13 to 1 4 bit multiplexer that outputs the next state of the vending machine.



Appendix B: Synthesis Report

Note: Only the advanced HDL synthesis and design summary sections are included. Some text has been omitted from the advanced HDL synthesis section.

```
======================================================================
*                       Advanced HDL Synthesis                       *
======================================================================
                       *** Some Text Omitted ***
Advanced HDL Synthesis Report
Macro Statistics
# RAMs                                                 : 1
 32x3-bit single-port distributed Read Only RAM        : 1
# MACs                                                 : 1
 4x1-to-5-bit MAC                                      : 1
# Adders/Subtractors                                   : 1
 3-bit adder                                           : 1
# Counters                                             : 20
 4-bit down counter                                    : 20
# Registers                                            : 14
 Flip-Flops                                            : 14
# Comparators                                          : 2
 4-bit comparator greater                              : 2
# Multiplexers                                         : 44
 3-bit 2-to-1 multiplexer                              : 10
 4-bit 13-to-1 multiplexer                             : 1
 4-bit 2-to-1 multiplexer                              : 33

======================================================================
*                            Design Summary                          *
======================================================================
Top Level Output File Name         : vending_machine.ngc

Primitive and Black Box Usage:
------------------------------
# BELS                             : 223
#      GND                         : 1
#      LUT2                        : 44
#      LUT3                        : 12
#      LUT4                        : 56
#      LUT5                        : 7
#      LUT6                        : 98
#      MUXF7                       : 5
# FlipFlops/Latches                : 103
#      FD                          : 9
#      FDE                         : 94
# Clock Buffers                    : 1
#      BUFGP                       : 1
# IO Buffers                       : 16
#      IBUF                        : 10
#      OBUF                        : 6

Device utilization summary:
---------------------------
Selected Device : 6slx16csg324-3

Slice Logic Utilization:
 Number of Slice Registers:             103  out of  18224     0%
 Number of Slice LUTs:                  217  out of   9112     2%
    Number used as Logic:               217  out of   9112     2%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:    225
   Number with an unused Flip Flop:     122  out of    225    54%
   Number with an unused LUT:             8  out of    225     3%
   Number of fully used LUT-FF pairs:    95  out of    225    42%
   Number of unique control sets:        27

IO Utilization:
 Number of IOs:                          17
 Number of bonded IOBs:                  17  out of    232     7%

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:                1  out of     16     6%

---------------------------
Partition Resource Summary:
---------------------------
  No Partitions were found in this design.
---------------------------
======================================================================
```

Appendix C: Mapping Report

```
Design Summary
--------------
Number of errors:      0
Number of warnings:    0
Slice Logic Utilization:
  Number of Slice Registers:                   103 out of  18,224    1%
    Number used as Flip Flops:                 103
    Number used as Latches:                      0
    Number used as Latch-thrus:                  0
    Number used as AND/OR logics:                0
  Number of Slice LUTs:                        186 out of   9,112    2%
    Number used as logic:                      186 out of   9,112    2%
      Number using O6 output only:             154
      Number using O5 output only:           0
      Number using O5 and O6:               32
      Number used as ROM:                    0
    Number used as Memory:                   0 out of   2,176    0%

Slice Logic Distribution:
  Number of occupied Slices:                60 out of   2,278    2%
  Number of MUXCYs used:                     0 out of   4,556    0%
  Number of LUT Flip Flop pairs used:      187
    Number with an unused Flip Flop:       107 out of     187   57%
    Number with an unused LUT:               1 out of     187    1%
    Number of fully used LUT-FF pairs:      79 out of     187   42%
    Number of unique control sets:          27
    Number of slice register sites lost
      to control set restrictions:         121 out of  18,224    1%

IO Utilization:
  Number of bonded IOBs:                    17 out of     232    7%

Specific Feature Utilization:
  Number of RAMB16BWERs:                     0 out of      32    0%
  Number of RAMB8BWERs:                      0 out of      64    0%
  Number of BUFIO2/BUFIO2_2CLKs:             0 out of      32    0%
  Number of BUFIO2FB/BUFIO2FB_2CLKs:         0 out of      32    0%
  Number of BUFG/BUFGMUXs:                   1 out of      16    6%
    Number used as BUFGs:                    1
    Number used as BUFGMUX:                  0
  Number of DCM/DCM_CLKGENs:                 0 out of       4    0%
  Number of ILOGIC2/ISERDES2s:               0 out of     248    0%
  Number of IODELAY2/IODRP2/IODRP2_MCBs:     0 out of     248    0%
  Number of OLOGIC2/OSERDES2s:               0 out of     248    0%
  Number of BSCANs:                          0 out of       4    0%
  Number of BUFHs:                           0 out of     128    0%
  Number of BUFPLLs:                         0 out of       8    0%
  Number of BUFPLL_MCBs:                     0 out of       4    0%
  Number of DSP48A1s:                        0 out of      32    0%
  Number of ICAPs:                           0 out of       1    0%
  Number of MCBs:                            0 out of       2    0%
  Number of PCILOGICSEs:                     0 out of       2    0%
  Number of PLL_ADVs:                        0 out of       2    0%
  Number of PMVs:                            0 out of       1    0%
  Number of STARTUPs:                        0 out of       1    0%
  Number of SUSPEND_SYNCs:                   0 out of       1    0%
Average Fanout of Non-Clock Nets:            4.21

Peak Memory Usage:  402 MB
Total REAL time to MAP completion:  15 secs
Total CPU time to MAP completion:   11 secs
```