Joshua Aymett

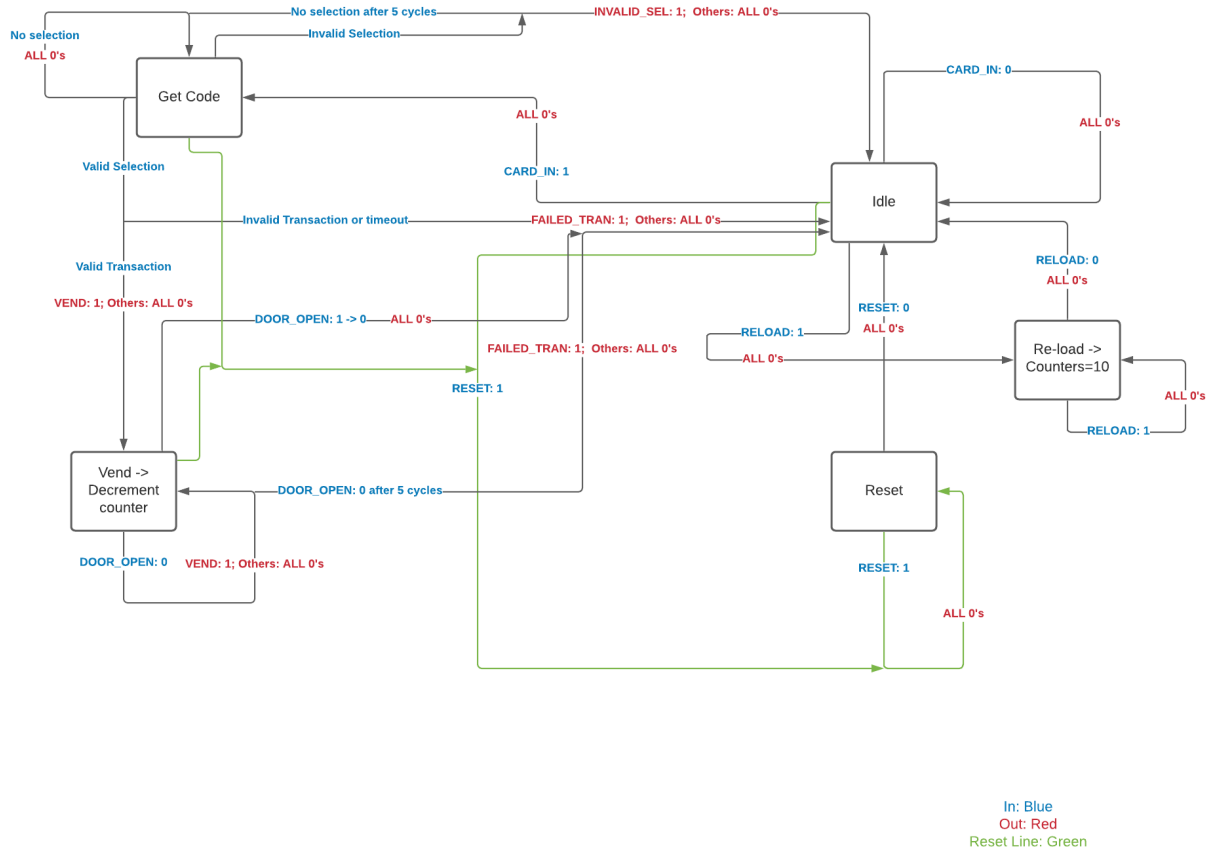
COM SCI M152A-6

02/28/2021

Project 3: Vending Machine

Project Summary

This project involves the design of a vending machine which accepts card payments, detects the validity of a transaction, tracks item inventory, and vends accordingly. It is designed as a mealy finite state machine and its outputs depend on both current state and input as can be seen through the diagram below.



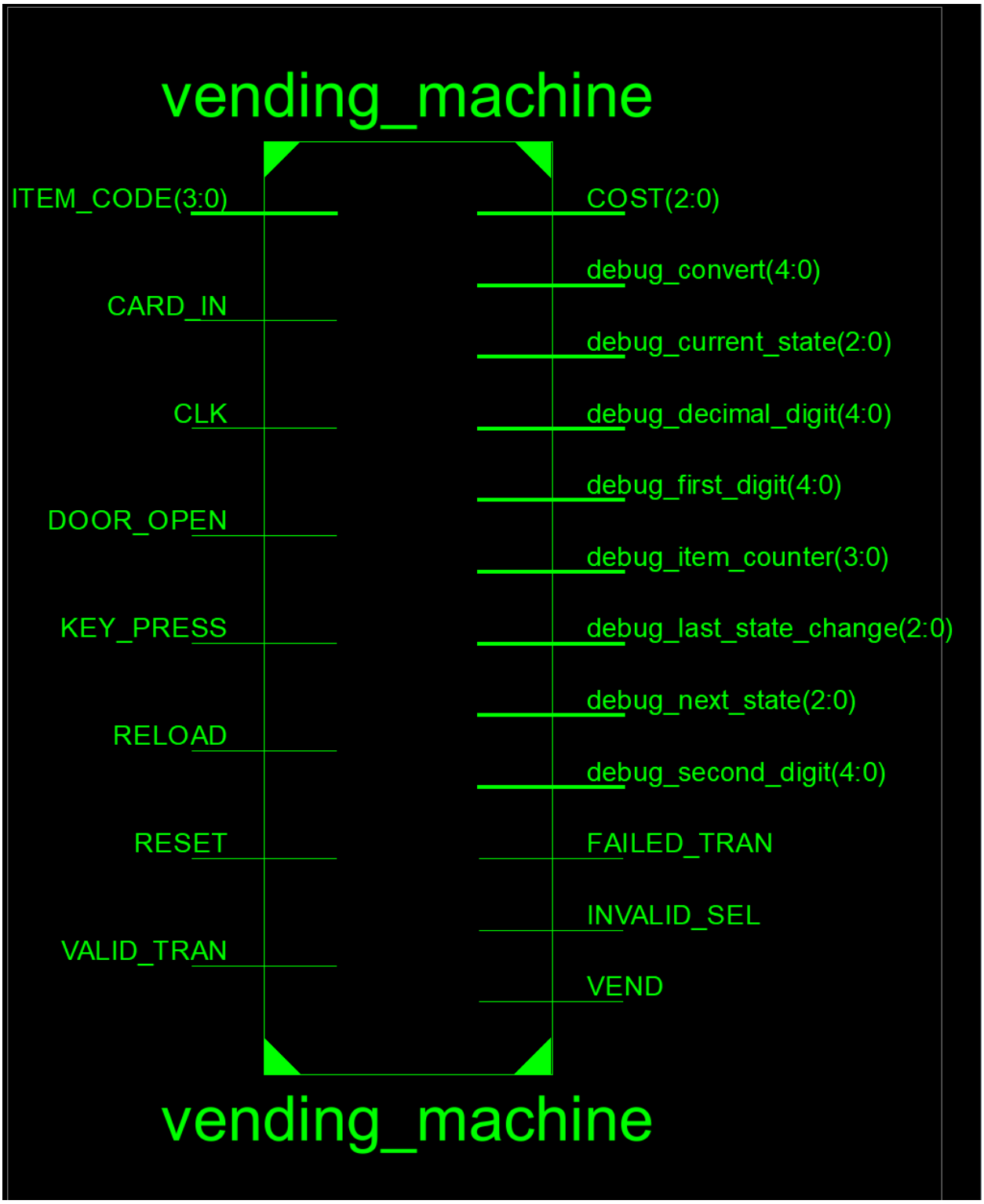
vending_machine Project Status (02/28/2021 - 12:43:33)			
Project File:	Project3.xise	**Parser Errors:**	No Errors
Module Name:	vending_machine	**Implementation State:**	Placed and Routed
Target Device:	xc6slx16-3csg324	**• Errors:**	
Product Version:	ISE 14.7	**• Warnings:**	
Design Goal:	Balanced	**• Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	**• Timing Constraints:**	All Constraints Met
Environment:	System Settings	**• Final Timing Score:**	0 (Timing Report)

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	104	18,224	1%	
Number used as Flip Flops	3			
Number used as Latches	101			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	172	9,112	1%	
Number used as logic	172	9,112	1%	
Number using O6 output only	138			
Number using O5 output only	0			
Number using O5 and O6	34			
Number used as ROM	0			
Number used as Memory	0	2,176	0%	
Number of occupied Slices	61	2,278	2%	
Number of MUXCYs used	0	4,556	0%	
Number of LUT Flip Flop pairs used	187			
Number with an unused Flip Flop	83	187	44%	
Number with an unused LUT	15	187	8%	
Number of fully used LUT-FF pairs	89	187	47%	
Number of unique control sets	29			
Number of slice register sites lost to control set restrictions	128	18,224	1%	
Number of bonded IOBs	50	232	21%	
IOB Flip Flops	3			
IOB Latches	13			
Number of RAMB16BWERs	0	32	0%	
Number of RAMB8BWERs	0	64	0%	
Number of BUFIO2/BUFIO2_2CLKs	0	32	0%	
Number of BUFIO2FB/BUFIO2FB_2CLKs	0	32	0%	
Number of BUFG/BUFGMUXs	1	16	6%	
Number used as BUFGs	1			
Number used as BUFGMUX	0			
Number of DCM/DCM_CLKGENs	0	4	0%	
Number of ILOGIC2/ISERDES2s	0	248	0%	
Number of IODELAY2/IODRP2/IODRP2_MCBs	0	248	0%	
Number of OLOGIC2/OSERDES2s	16	248	6%	
Number used as OLOGIC2s	16			
Number used as OSERDES2s	0			
Number of BSCANs	0	4	0%	
Number of BUFHs	0	128	0%	
Number of BUFPLLs	0	8	0%	
Number of BUFPLL_MCBs	0	4	0%	
Number of DSP48A1s	2	32	6%	
Number of ICAPs	0	1	0%	
Number of MCBs	0	2	0%	
Number of PCILOGICSEs	0	2	0%	
Number of PLL_ADVs	0	2	0%	
Number of PMVs	0	1	0%	
Number of STARTUPs	0	1	0%	
Number of SUSPEND_SYNCs	0	1	0%	
Average Fanout of Non-Clock Nets	3.65			

Performance Summary				[-]
Final Timing Score:	0 (Setup: 0, Hold: 0)	**Pinout Data:**	Pinout Report	
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report	
Timing Constraints:	All Constraints Met			

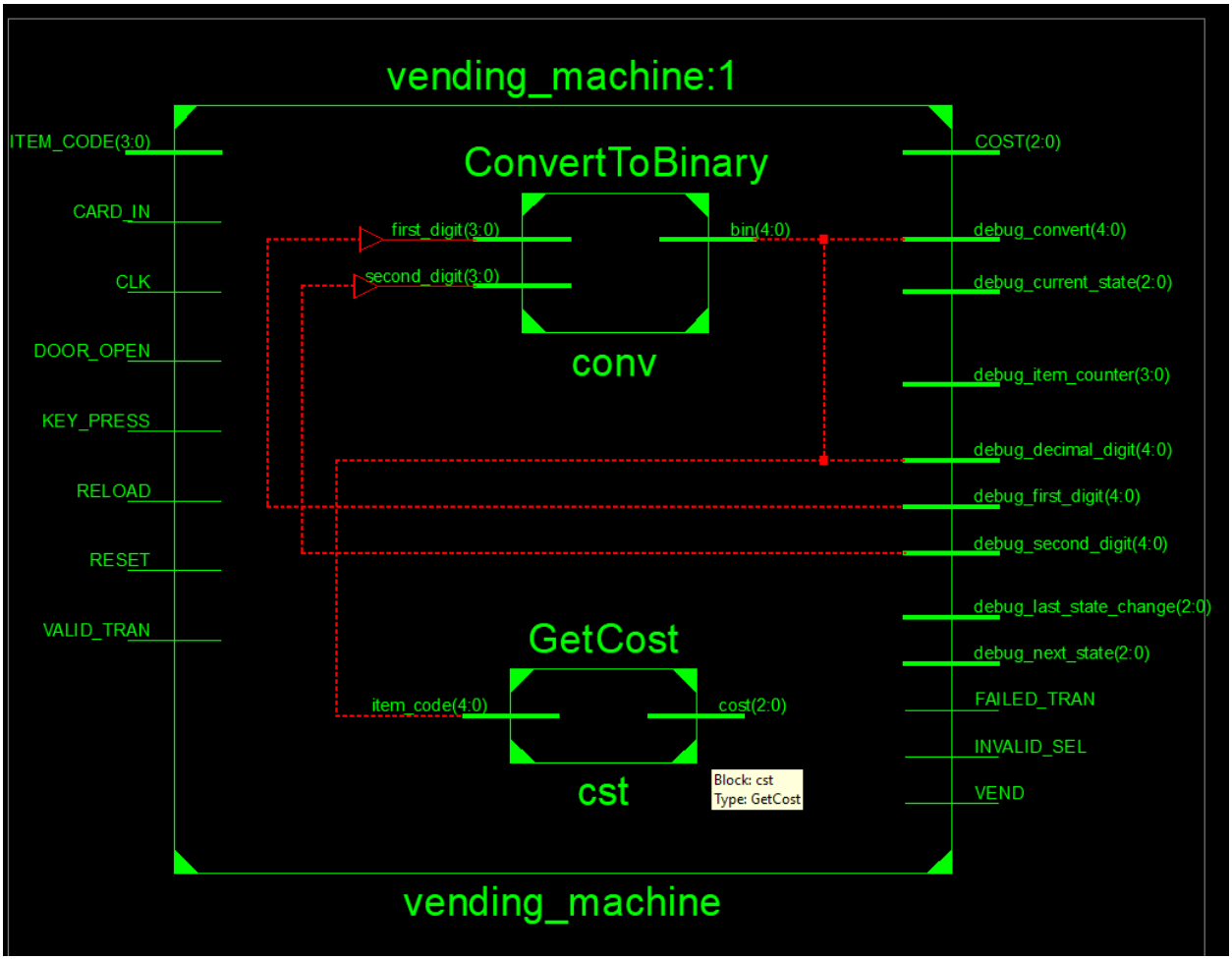
Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Sun Feb 28 22:17:06 2021			
Translation Report	Current	Sun Feb 28 22:18:31 2021	0	0	0
Map Report	Current	Sun Feb 28 22:18:56 2021	0	29 Warnings (29 new)	6 Infos (0 new)
Place and Route Report	Current	Sun Feb 28 22:19:13 2021	0	0	3 Infos (0 new)
Power Report					
Post-PAR Static Timing Report	Current	Sun Feb 28 22:19:25 2021	0	0	4 Infos (0 new)
Bitgen Report					

Design summary of vending_machine module. The presence of slice registers indicates that the module is a sequential circuit.

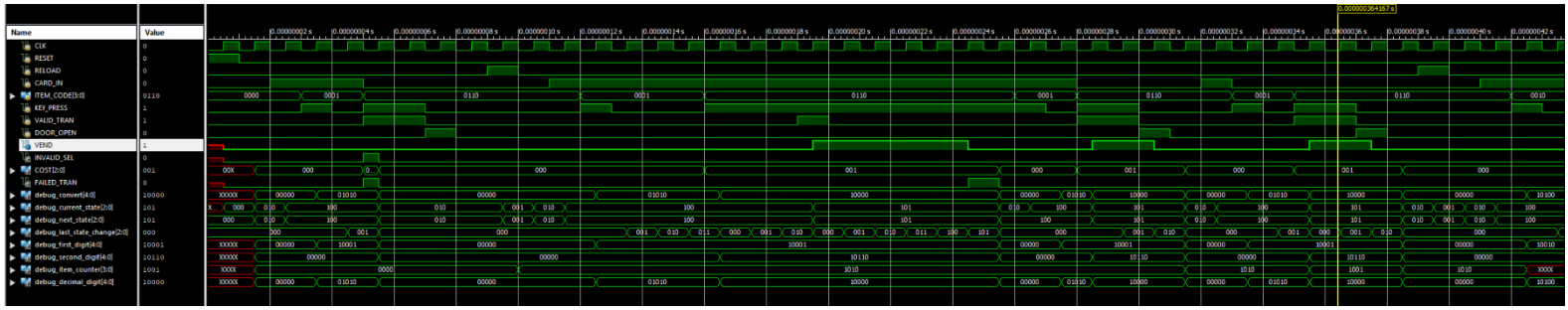


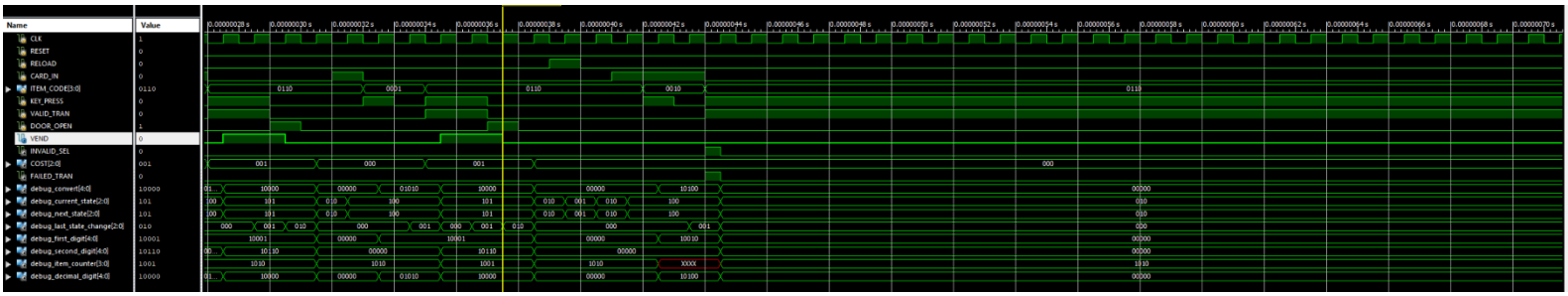
High level design of vending_machine. Displays all inputs and outputs with additional debug outputs.

\



High level design of vending_machine. Displays how each submodel connects to parent module output.





Simulation outputs which display the following test bench:

1. Attempted purchasing of out of stock items. Correct because the transaction fails with invalid selection and failed transaction flags, returning to idle state.

2. Reloading. Correct because the quantity changes from 0 to 10 (can be seen through bonus debug outputs)

3. Timeout by door failing to open. Correct because even though the transaction and selections were valid, the door failing to open resulted in a failed transaction with valid selection and successful return to idle state.

4. Two subsequent successful purchases of the same item. Correct because the machine successfully moves through the entire workflow twice without any errors and the corresponding item's quantity is seen to be decremented through the debug_item_counter output. NOTE: this output points to whatever input digits have been entered, even if invalid. When moving back to idle, it points to 00 which will have a count of 10 (1010 bin).

5. Invalid item selection. Correct because in spite of valid_tran being high, incorrect selection and failed transaction flags are set to high and the state returns to idle because of invalid selection.

These test cases also test for irregularities in timing inputs and residual inputs from previous transactions such as key_press never going to 0. This demonstrates the robustness of the design because even unlikely edge-case inputs do not break the machine.

****Item counter is implemented using an 80 bit register (4 bits/item * 20 items = 80 bits). This is reloaded using a for loop and decremented using the single binary converted value from the ConvertToBinary module.**

Implementation Summary:

The machine is composed of 5 states: *Reset, Reload, Idle, Get Code, and Vend.* The reset state is active when the reset input is high and it sets all outputs to 0, clearing inventory counters. Similarly, the reload state is active when the reload input is high; however, instead of clearing counters, it sets them to 10. When these signals drop, both states move to the idle state. All three of these states output 0's for cost, vend, and error flags. Once in the idle state, the machine waits for a card to be entered. When CARD_IN goes high while the machine is idling, a state change is triggered and the machine moves to the GET_CODE state. In this state, the machine is waiting for two digits to be entered separately and for the transaction to be deemed valid. It will wait 5 clock cycles for the first digit to be entered, 5 clock cycles for the second digit to be entered, and then 5 more clock cycles for the valid transaction input to go high. Once two valid digits are entered, the cost is outputted until the end of the transaction (return to IDLE). If one of these conditions fails, failed transaction and invalid selection outputs will equal one and the machine will return to the idle state. If all of these conditions are satisfied, the machine moves to the VEND state. In this state, the user is able to remove their purchased item by opening a door and taking it. However, if the door does not open within 5 clock cycles, the transaction is assumed to have failed and the FAILED_TRAN flag is set to high while the state is changed back to IDLE. If the door opens and closes successfully, the counter for the selected item is decremented and the machine returns to the idle state with no error flags.

Inputs and outputs with parameters used to represent states for readability:

```
module vending_machine(
  input CLK,
  input RESET,
  input RELOAD,
  input CARD_IN,
  input [3:0] ITEM_CODE,
  input KEY_PRESS,
  input VALID_TRAN,
  input DOOR_OPEN,
  output reg VEND,
  output reg INVALID_SEL,
  output reg [2:0] COST,
  output reg FAILED_TRAN,

            // Debug outputs
            output reg [4:0] debug_convert,
            output reg [2:0] debug_current_state,
            output reg [2:0] debug_last_state_change,
            output reg [2:0] debug_next_state,
            output reg [4:0] debug_first_digit,
            output reg [4:0] debug_second_digit,
            output reg [3:0] debug_item_counter,
            output reg [4:0] debug_decimal_digit
  );

            // State parameters
            parameter RESET_ST = 3'b000;
            parameter RELOAD_ST = 3'b001;
            parameter IDLE_ST = 3'b010;
            parameter GET_CODE_ST = 3'b100;
            parameter VEND_ST = 3'b101;
```

The next state decoder controls state transitions by comparing current state and output and deciding what the next state should be. This is accomplished through a case statement and if statements which check current state and input values. For example, if the machine is in the RESET state, it checks for the reset flag. If the reset flag is high, it keeps resetting. If it is low, it moves to idle. Similarly, the RELOAD state checks for the reload flag and behaves accordingly. Next, the idle state checks for reload, reset, or card in. If reload or reset are high, the next state is the RELOAD or RESET state respectively. If CARD_IN goes high, the machine moves to the GET_CODE state and waits for input. Once in the GET_CODE state, a counter increments each clock cycle to enable timeout functionality. This logic is fairly complex due to the sensitivity of timing. For instance, the valid transaction flag can go high any time after the card is inputted which means that a separate flag must be used to track it as there is no guarantee that the valid transaction input will be high when an item is selected. Next, a series of conditions are checked which include the reset flag, valid input, invalid input, and timeout. One of the most important parts of this code is the code entry portion which checks if the codes have been set and if they are valid. Notice how the fifth bit, bit 4, is used even though the digits themselves are only four bits long. This is because the fifth bit is used as a flag to differentiate unset bits from a 0 digit. Directly below the GET_CODE portion is the vend logic. Similarly to before, the number of clock cycles since the last state change are tracked so that the machine will timeout if the door is not opened within 5 clock cycles. If it has opened and closed, the corresponding portion of the item_ct register is decremented by 1 to reflect a decrease in inventory.

```
// Next State Decoder
	always @(*) begin
		// <DEBUG>
		debug_convert = decimal_digit;
		debug_current_state = current_state;
		debug_next_state = next_state;
		debug_last_state_change = last_state_change;
		debug_first_digit = first_digit;
		debug_second_digit = second_digit;
		debug_item_counter = item_ct [((decimal_digit +1)*4) +:4];
		debug_decimal_digit = decimal_digit;
		// </DEBUG>

		case(current_state)
			RESET_ST : begin
				// Set item count to 0
				item_ct = 80'b0;

				// Check for reset signal
				if (RESET) begin
					next_state = RESET_ST;
				end
				else begin
					next_state = IDLE_ST;
				end
			end
			RELOAD_ST : begin
				for (i = 0; i < 20; i = i + 1) begin
					item_ct[i*4 +: 4] = 4'b1010;
				end
				if (RELOAD) begin
					next_state = RELOAD_ST;
				end
				else begin
					next_state = IDLE_ST;
				end
			end
			IDLE_ST : begin
				// Clear flags
				valid_tran_flag = 0;
				delay_count = 0;
				// Clear last state change (not used for idle)
				if (last_state_change > 3'b000)
					last_state_change <= 3'b000;
				if (CARD_IN) begin
					next_state = GET_CODE_ST;
				end
				else if (RELOAD) begin
					next_state = RELOAD_ST;
				end
				else if (RESET) begin
					next_state = RESET_ST;
				end
			end
			GET_CODE_ST : begin
				#10;
				last_state_change <= last_state_change + 1'b1;
				if (VALID_TRAN)
					valid_tran_flag = 1'b1;
				if (RESET) begin
					next_state = RESET_ST;
					valid_tran_flag = 1'b0;
				end
				// If two digits have been entered
				else if (first_digit[4] & first_digit[3:0] < 4'b0010 & second_digit[4]) begin
					// Check for item in stock
					if (item_ct [((decimal_digit +1)*4) +:4] > 4'b0000) begin
						if (valid_tran_flag) begin
							next_state = VEND_ST;
							last_state_change <= 3'b000;
```

```
                                        end
                                end
                                else begin
                                        next_state = IDLE_ST;
                                end
                        end
                        else if (first_digit[4] & first_digit[3:0] > 4'b0001 & second_digit[4]) begin
                                next_state = IDLE_ST;
                        end
                        // Give 5 extra cycles per key press
                        if (first_digit[4] & delay_count == 0) begin
                                last_state_change <= 0;
                                delay_count = delay_count + 1;
                        end
                        else if (second_digit[4] & delay_count == 1)  begin
                                last_state_change <= 0;
                                delay_count = delay_count + 1;
                        end
                        // Check timeout
                        if (last_state_change > 4) begin
                                next_state = IDLE_ST;
                                last_state_change <= 0;
                                delay_count = 0;
                        end
                end
                VEND_ST : begin
                        #10;
                        last_state_change <= last_state_change + 1'b1;
                        // Door is open
                        if (DOOR_OPEN) begin
                                opened_flag = 1'b1;
                        end
                        // Door opened but is now closed
                        else if (opened_flag) begin
                                item_ct [((decimal_digit +1)*4) +:4] = item_ct [((decimal_digit +1)*4) +:4] - 4'b0001;
                                opened_flag = 1'b0;
                                next_state = IDLE_ST;
                        end
                        else if (last_state_change > 4) begin
                                next_state = IDLE_ST;
                                last_state_change <= 0;
                        end
                end
                default : begin
                        // Reset if no state has been set
                        next_state = RESET_ST;
                end
        endcase
end
```

The output decoder handles outputs based on state and input. It is more simple than the next state decoder with all 0's being outputted for the RESET, RELOAD, and IDLE states. Notice how flags are cleared in the idle state. This is because the idle state is the central state which all other states connect to. Because of this, it is a perfect location to clear flags. Directly below this is the GET_CODE state which handles inputs, valid transaction and selection timeouts, and selection input validation. The outputs can be summed up as follows: if an invalid number is entered or the machine times out, invalid selection and failed transaction flags go high and vend is set to 0 as the machine returns to idle. If none of these conditions are satisfied, no output changes occur as the the machine is already outputting the correct value and the machine moves into the VEND state. In this state, the VEND flag is set to high until the door is opened and then set to 0 after it opens. If the machine times out, the failed transaction flag is set to high as the machine returns to the idle state.

```
// Output Decoder
always @(*) begin
        COST = costw & first_digit[4] & second_digit[4];
        decimal_digit = decimal_digitw;
        case(current_state)
                RESET_ST : begin
                        VEND = 1'b0;
                        INVALID_SEL = 1'b0;
                        FAILED_TRAN = 1'b0;
                end

                RELOAD_ST : begin
                        VEND = 1'b0;
                        INVALID_SEL = 1'b0;
                        FAILED_TRAN = 1'b0;
                end

                IDLE_ST : begin
                        first_digit = 0;
                        second_digit = 0;
                        key_released = 0;
                        VEND = 1'b0;
                        INVALID_SEL = 1'b0;
                        COST = 3'b000;
                        FAILED_TRAN = 1'b0;
                end

                GET_CODE_ST : begin
                        if (RESET) begin
                                VEND = 1'b0;
                                INVALID_SEL = 1'b0;
                                FAILED_TRAN = 1'b0;
                        end
                        else if (KEY_PRESS) begin
                                // If the first digit has not been set, set it to item code
                                if (!first_digit[4]) begin
                                        first_digit[4] = 1;
                                        first_digit[3:0] = ITEM_CODE;
                                end
                                // If the second digit has not been set, set it to item code
                                else if (key_released & !second_digit[4]) begin
                                        second_digit[4] = 1;
                                        second_digit[3:0] = ITEM_CODE;
                                        // Check for invalid input
                                        if(first_digit[3:0] > 4'b0001) begin
                                                VEND = 1'b0;
                                                INVALID_SEL = 1'b1;
```

```
                                                FAILED_TRAN = 1'b1;
                                        end
                                        // Else valid
                                        else begin
                                                // Check item in stock
                                                if (!(item_ct[((decimal_digit +1)*4) +:4] > 4'b0000)) begin
                                                        VEND = 1'b0;
                                                        INVALID_SEL = 1'b1;
                                                        FAILED_TRAN = 1'b1;
                                                end
                                        end
                                end
                        end
                        else if (first_digit[4])
                                key_released = 1'b1;
                        if (last_state_change > 4) begin
                                VEND = 1'b0;
                                INVALID_SEL = 1'b1;
                                FAILED_TRAN = 1'b1;
                        end
                end

                VEND_ST : begin
                        if (!opened_flag) begin
                                VEND = 1'b1;
                                INVALID_SEL = 1'b0;
                                FAILED_TRAN = 1'b0;
                        end
                        else begin
                                VEND = 1'b0;
                                INVALID_SEL = 1'b0;
                                FAILED_TRAN = 1'b0;
                        end
                        if (last_state_change > 4) begin
                                VEND = 1'b0;
                                INVALID_SEL = 1'b0;
                                FAILED_TRAN = 1'b1;
                        end
                end
        endcase
    end
endmodule
```

Lastly, the ConvertToBinary and GetCost modules convert the two independent digits to one and return the item cost respectively.

```
// Converts two separate digits to one binary number
module ConvertToBinary (
	input [3:0] first_digit,
	input [3:0] second_digit,
	output [4:0] bin
	);
	assign bin = (10 * first_digit) + second_digit;
endmodule


// Returns the cost of an item based on its code
module GetCost (
		input [4:0] item_code,
		output reg [2:0] cost
	);

	always @(*) begin
		if (item_code < 4)
			cost <= 3'b001;
		else if (item_code < 8)
			cost <= 3'b010;
		else if (item_code < 12)
			cost <= 3'b011;
		else if (item_code < 16)
			cost <= 3'b100;
		else if (item_code < 18)
			cost <= 3'b101;
		else
			cost <= 3'b110;
	end
endmodule
```

Post-Project Evaluation

My methodology for solving the given task began with the creation of a state diagram. I used this state diagram as a foundation for my code which made the implementation far easier. I used additional debug outputs to verify functionality for use by both myself and the grader. Using this approach allowed me to ensure that my logical design worked before I became too committed to a specific logical design. This approach saved time by catching bugs before other code was based on faulty logic; however, I still ran into many challenges. For example, outputting based on input proved particularly challenging because of the way I had designed my state. Initially, I had included a transact state which moved directly into the GET_CODE state; however, it proved to be fairly useless and should have been substituted for other states which described errors. I only realized this once I was very far along in my design and rather than rewrite the code, I used additional registers at the module level to compensate for my few states. Although not ideal, the code functioned and proved to be readable and maintainable.

I found this project to be difficult and was significantly challenged by it, especially in parts requiring more complex logic. However, I was still able to successfully complete it and appreciated its relevance to finite state machines and hardware logic design. To improve the project, I would add specific example workflows relating to edge cases so that students can understand by tracing through examples and possible pseudocode. In conclusion, I believe that this project was a challenging and relevant assignment that helped develop my Verilog abilities.