CS M152A

Lab 4: Creative Project

Section 3

TA: Farhad Shahmohammadi

Roger Chen, 504043927

Kailin Chang, 503999157

Introduction and Requirement:

The purpose of this lab was to come up with a project that utilized everything we've learned in the class. We decided on implementing a vending machine controller with additional functionalities on the Nexys3 FPGA board. The vending machine supports multiple items and accepts nickels, dimes, and quarters through button presses. Through the use of switches, the user may access the additional modes that the vending machine has and also be able to select different items. The digits will be displayed on the seven-segment display. The user will press a button to confirm their purchase and then the same button again to ready the vending machine for item purchases.

The vending machine has the following functionalities:

- INFO: 0 when showing price of an item, 1 when showing quantity of item
- ADMIN: 0 if user mode, 1 if admin mode

ADMIN	INFO	Selected
0	0	Price of item
0	1	Quantity of item
1	0	Revenue made
1	1	Update stock of item

- RESET: Cancels a transaction and returns change back to user if in user mode, otherwise, reset the information being shown in admin mode to 0.
- ITEM: 2 switches that can be toggled to select an item for a total of 4 possible items.
- BUY: Button to purchase an item or to ready the vending machine again. The user is also prevented from buying an out of stock item; when BUY is pressed, the display will show "NONE" on the display and wait for the user to ready the vending machine again.
- When ADMIN and INFO are both high, the normal buttons used for inputting nickels, dimes, and quarters can be used to increase the stock by 5, 10, and 25 respectively. The display will also blink.
- If there is change to be given out, the display will show the letter "C" and the amount of change to be given out in terms of nickels, dimes, and quarters; otherwise, it will show the word "HERE" on the display if exact amount is given.

The implementation is primarily separated into 5 parts: the vending machine, change machine, clock divider, seven-segment display, and binary to decimal converter. The

vending machine is responsible for implementing the finite state machine that transitions to different states upon given inputs. The change machine is responsible for giving back change to the user in denominations of nickels, dimes, and quarters. The clock divider consists of 3 clocks: a 20 Hz clock for debouncing, a 2 Hz clock for blinking, and a 167 Hz clock for the seven-segment display. Lastly, the binary to decimal converter converts binary numbers into decimal so they can be properly displayed on the seven-segment display.

Design Description

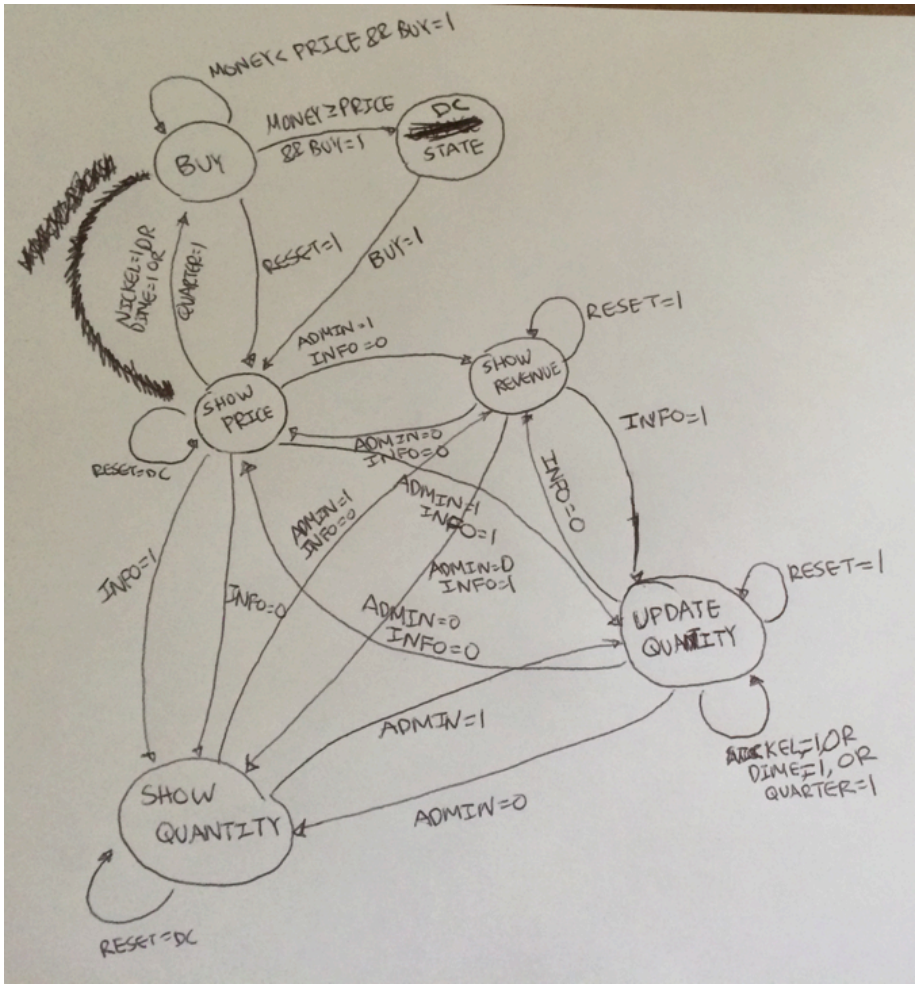

Figure 1: Vending Machine FSM

We designed our stopwatch to be a Mealy Machine with six main states: Buy, DC (for displaying Here and Change), Show Price, Show Quantity, Update Quantity, and Show Revenue. For the most part, the output depends only on the current state. However, we also added an input that indicates whether a reset signal was sent, so the output now depends on the input as well, making it a Mealy Machine.

Below is the overall outline of the modules for our design:

1. A sub-system that divides the clock into a 20 Hz, 2 Hz, 167 Hz to the 3 clocks mentioned in the Introduction.
2. The vending machine that reads the inputs (INFO, ADMIN, RESET, ITEM, BUY, MONEY) and translates those into states.
3. A sub-system that takes in change represented as a 7-bit binary number and converts it to a nickels, dimes, quarters representation.
4. A sub-system that takes in the 16-bit binary number to be displayed and converts it to a representable decimal number for the display.
5. A sub-system that translates a given number to its equivalent cathode value for the seven-segment display

- **Clock Divider**
  - At every positive edge of the input clock we pass into the module, the designated set of increment registers increment by 1 similar to the code in Lab 1.
  - When the increment registers hit the constants associated with clocks at 2 Hz, 167 Hz, and 20 Hz signals, another set of registers called enabling registers are set to 1. The enabling registers are set to 0 at all other times. For example, the constant for 1 Hz is 100000000, because it should take 100000000 clock cycles to go to '1' and then back to '0', assuming that we have a 100 MHz master clock.
  - The outputs of these enabling registers are set as the signals to `clkdisplay`, `clkdebounce`, and `clkblink`.

- **Vending Machine**
  - At the positive edge of `clkdebounce`, we check for inputs and transition the current state according to the inputs.
  - First, we check what item is selected by translating the ITEM switches accordingly: 2'b00 → Item A, 2'b01 → Item B, 2'b10 → Item C, 2'b11 → Item D
  - If the current state is in DC state, then we check if the BUY button has been pressed and transitions the vending machine back to the PRICE state. Our DC state is also the state used when a user tries to buy an out of stock item.
  - If the current state is in PRICE state, we check if the nickel, dime, or quarter input is positive and stable. If they are, then we check these conditions:
    - If the quantity of the item currently selected is 0, then transition to the DC state and set the flag indicating "OOS" to 1
    - Else, Update the current money inputted by the correct amount and transition to the BUY state

- If the current state is in SHOW QUANTITY state, we check for the conditions shown in the FSM: e.g. if ADMIN == 1 && INFO == 1, then transition to the UPDATE QUANTITY state.
- Similarly, if the current state is in the SHOW REVENUE state, we check for the conditions shown in the FSM. If the reset signal is positive and stable, then we reset the revenue to 0.
- If the current state is in UPDATE QUANTITY state:
    - We first check if the reset signal is positive and stable and if it is, we reset the quantity of the item currently selected to 0.
    - If the nickel, dime, or quarter input is positive and stable, then we update the selected item's quantity accordingly: nickel → increase stock by 5 items, dime → increase stock by 10 items, quarter → increase stock by 25 items.
- If the current state is in BUY state:
    - We first check if the reset signal is positive and stable and if it is, we reset the current money deposited and return it as change and transition to the PRICE state
    - If the buy signal is positive and stable, then we check if the current money inputted is able to buy the selected item
        - If the current money is exactly the amount needed to buy the selected item, then transition to the DC state, decrease the selected item's quantity by 1 and update the revenue by the item's price
        - If the current money is more than the item's price, transition to the DC state, decrease the selected item's quantity by 1, update revenue, and update change to be the difference of the money inputted and the price.
        - Otherwise, stay in BUY state
    - If the nickel, dime, or quarter input is positive and stable, then just update the current money inputted.

- **16-bit Binary to Decimal Converter**
    - We first separate the number into 4 registers of 4 bits each. The 4 register represents the D4 input (represents a C for change), the hundreds place, the tens place, and the units place.
    - We iterate through all bits in the 16 bit number. At any time, if any of the 4 registers has a value greater than 5, we add 3 to that register.
    - During each iteration, we shift all the bits to the left by 1.
    - At the end of the iterations, each register has the value corresponding to the decimal digit it should represent.

- **Change Machine**
  - The change machine takes the input money and loops until it has found the correct number of denominations to represent the input money.
  - It first finds the modulo for quarter (25) and dime (10)
  - It compares the modulo, and if either of them are 0, the number of denominations we need to return has been found, so just divide the input money by the denomination to get the number we need
  - If the modulo of 25 is greater than the modulo of 10, then the number of dimes is increased accordingly; otherwise, then the number of quarters is increased accordingly
  - If the remaining money to be converted is less than 10, then simply return the number of nickels needed to convert it.

- **Seven Segment Display Counter**
  - Whenever the input count changes, the output is set to the current count, which is then set as the seven-segment display value.

- **Seven-Segment Display (TODO)**
  - At the positive edge of the 167 Hz clock (`clkdisplay`), the counter called `ss_display_counter` which increments by 1.
  - When the `ss_display_counter` is 00
    - If the `D4_flag` is 01, the `anode_register` and `segment_register` values are set to output the H in HERE.
    - If the `D4_flag` is 10, the `anode_register` and `segment_register` values are set to output C (for Change).
    - If the `OOS_flag` is set to 1, the `anode_register` and `segment_register` values are set to output N (for NONE).
    - Otherwise, depending on the values of admin and info, the `anode_register` and `segment_register` values are set to output nothing or the corresponding digit assigned to `display_D4`.
  - When the `display_counter` is 01
    - If the `D4_flag` is 01, the `anode_register` and `segment_register` values are set to output the E in HERE.
    - If the `D4_flag` is 10, the `anode_register` and `segment_register` values are set to output the value corresponding to how many nickels to dispense according to the Change Machine.
    - If the `OOS_flag` is set to 1, the `anode_register` and `segment_register` values are set to output O (for NONE).

        - Otherwise, depending on the values of admin and info, the `anode_register` and `segment_register` values are set to output the corresponding digit assigned to `display_hundreds`.
- When the `display_counter` is 10
    - If the `D4_flag` is 01, the `anode_register` and `segment_register` values are set to output the R in HERE.
    - If the `D4_flag` is 10, the `anode_register` and `segment_register` values are set to output the value corresponding to how many dimes to dispense according to the Change Machine.
    - If the `OOS_flag` is set to 1, the `anode_register` and `segment_register` values are set to output N (for NONE).
    - Otherwise, depending on the values of admin and info, the `anode_register` and `segment_register` values are set to output the corresponding digit assigned to `display_tens`.
- When the `display_counter` is 11
    - If the `D4_flag` is 01, the `anode_register` and `segment_register` values are set to output the E in HERE.
    - If the `D4_flag` is 10, the `anode_register` and `segment_register` values are set to output the value corresponding to how many quarters to dispense according to the Change Machine.
    - If the `OOS_flag` is set to 1, the `anode_register` and `segment_register` values are set to output E (for NONE).
    - Otherwise, depending on the values of admin and info, the `anode_register` and `segment_register` values are set to output the corresponding digit assigned to `display_ones`.
- To create the blinking effect, when the ADJ bit is 1 and the SEL bit is 0:
    - If the 2 Hz clock (`clkblink`) is 1, the display is blank
    - If the 2 Hz clock (`clkblink`) is 0, the display is shown

Simulation Documentation

Test Bench for Change Machine (change_machine_test.v)

```
module change_machine_test;
	// Inputs
	reg [6:0] change;

	// Outputs
	wire [3:0] nickel;
	wire [3:0] dime;
	wire [3:0] quarter;
```

```
	// Instantiate the Unit Under Test (UUT)
	change_machine uut (
		.change(change),
		.nickel(nickel),
		.dime(dime),
		.quarter(quarter)
	);

	initial begin
		// Initialize Inputs
		change = 0;

		// Wait 100 ns for global reset to finish
		#100;

		// Add stimulus here
		change = 7'b0001111;
		#1000;
	end
endmodule
```

Figure 2: Simulation for Change Machine

Test Bench for 16-Bit Binary to Decimal Converter (bcd_test.v)

```
module sixteen_bit_bcd_test;

	// Inputs
	reg [15:0] binary;

	// Outputs
	wire [3:0] D4;
```

```
	wire [3:0] hundreds;
	wire [3:0] tens;
	wire [3:0] ones;

	// Instantiate the Unit Under Test (UUT)
	sixteen_bit_bcd uut (
		.binary(binary),
		.D4(D4),
		.hundreds(hundreds),
		.tens(tens),
		.ones(ones)
	);

	initial begin
		// Initialize Inputs
		binary = 0;

		// Wait 100 ns for global reset to finish
		#100;

		// Add stimulus here
		binary = 8'b01111101;  // Test for 8 bit input
		#100;
		binary = 16'b0000010011100101;  // Test for 16 bit input
		#100;

	end
endmodule
```

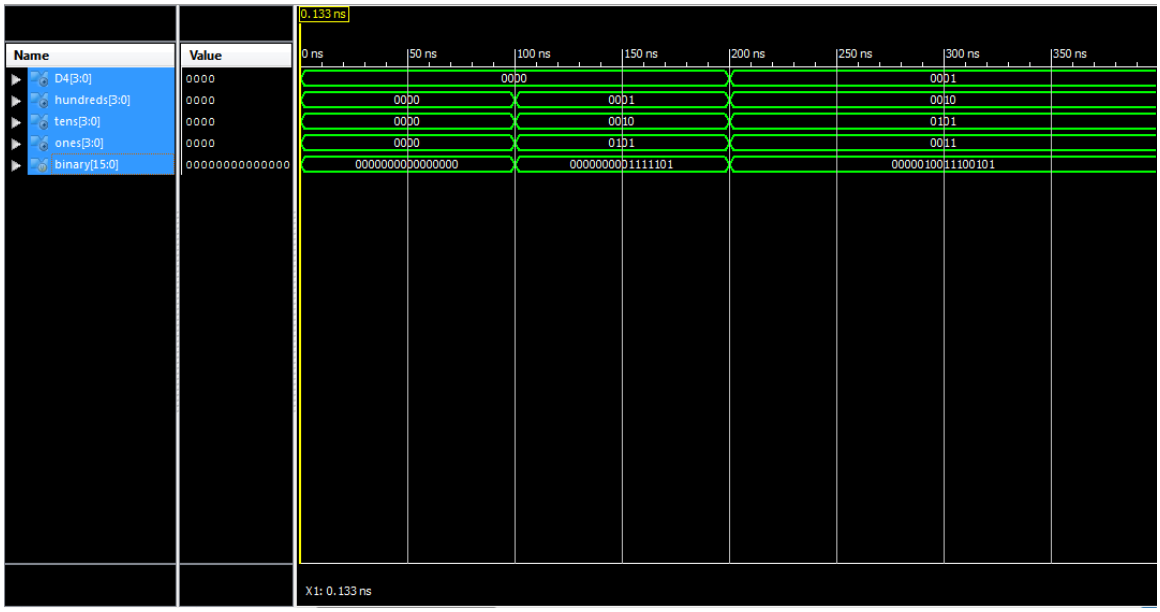

Figure 3: Simulation for 16-bit Binary to Decimal Converter

Test Bench for Vending Machine (vending_machine_test.v)

```
module vending_machine_test;

	// Inputs
	reg clk;
	reg [1:0] item;
	reg nickel;
	reg dime;
	reg quarter;
	reg rst;
	reg admin;
	reg info;
	reg buy;

	// Outputs
	wire [2:0] output_state;
	wire [6:0] output_money;
	wire [6:0] output_change;
	wire [15:0] output_quantity;
	wire [15:0] output_revenue;
	wire [3:0] ones; // rightmost
	wire [3:0] tens;
   wire [3:0] hundreds;
	wire [3:0] D4; // leftmost
	wire [3:0] anode;
	wire [6:0] seg_display;

	// Instantiate the Unit Under Test (UUT)
	vending_machine uut (
		.clk(clk),
		.item(item),
		.nickel(nickel),
		.dime(dime),
		.quarter(quarter),
		.rst(rst),
		.admin(admin),
		.info(info),
		.buy(buy),
		.output_state(output_state),
		.output_money(output_money),
		.output_change(output_change),
		.output_quantity(output_quantity),
		.output_revenue(output_revenue),
		.ones(ones), // rightmost
	   .tens(tens),
	   .hundreds(hundreds),
	   .D4(D4), // leftmost
```

```
   .anode(anode),
   .seg_display(seg_display)
);

initial begin
       // Initialize Inputs
       clk = 0;
       item = 0;
       nickel = 0;
       dime = 0;
       quarter = 0;
       rst = 0;
       admin = 0;
       info = 0;
       buy = 0;
       // Initial State: Price - 000

       // Wait 100 ns for global reset to finish
       #100;

       // Add stimulus here
       // Testing the transitions into different states
       info = 1;
       #1000;              // State: Quantity - 001
       info = 0;
       #1000;              // State: Price - 000
       admin = 1;
       #1000;              // State: Revenue - 010
       info = 1;
       #1000;              // State: Update - 011
       admin = 0;
       #1000;
       buy = 1;
       #1000;
       buy = 0;
       #1000;              // State: Quantity - 001
       admin = 1;
       info = 0;
       #1000;              // State: Revenue - 010
       //nickel = 1;
       #1000;              // State: Revenue - 010
       //nickel = 0;
       admin = 0;
       info = 0;
       #1000;              // State: Price – 000
```

```
// Testing money input
nickel = 1;         // Money: 0000101
#10;                // State: Buy - 100
nickel = 0;
#10;
buy = 1;
#100;
buy = 0;            // Not enough money
#1000;
nickel = 1;         // Money: 0001010
#10;
nickel = 0;
#1000;
dime = 1;           // Money: 0010100
#10;
dime = 0;
#1000;
quarter = 1;        // Money: 0101101
#10;
quarter = 0;
#10;
buy = 1;
#100;

// Testing revenue
buy = 0;            // Change: 0001010
#100;               // State: Price - 000
                    // Quantity: 1001
                    // Revenue: 0000000000100011
admin = 1;
#1000;              // State: Revenue - 010
info = 1;
#1000;              // State: Update - 011

// Testing update quantity
nickel = 1;
#10;
nickel = 0;
#10;                // Quantity: 00001110
dime = 1;
#10;
dime = 0;
#10;                // Quantity: 00011000
quarter = 1;
#10;
quarter = 0;
#10;                // Quantity: 00110001
admin = 0;
```

```
            #1000;

            // Testing reset
            info = 0;
            #1000;              // State: Price - 000
            nickel = 1;         // Money: 0000101
            #10;                // State: Buy - 100
            nickel = 0;
            #10;
            rst = 1;            // State: Price - 000
            #1000;              // Money: 0000000
            rst = 0;            // Change: 0000101
            #1000;
            admin = 1;
            #1000;
            info = 0;           // State: Revenue 010
            #1000;
            rst = 1;
            #1000;              // Revenue: 0000000000000000
            rst = 0;
            #1000;
            info = 1;           // State: Update - 011
            #1000;
            rst = 1;
            #1000;              // Quantity: 00000000
      end
      always
            #5 clk = ~clk;
endmodule
```

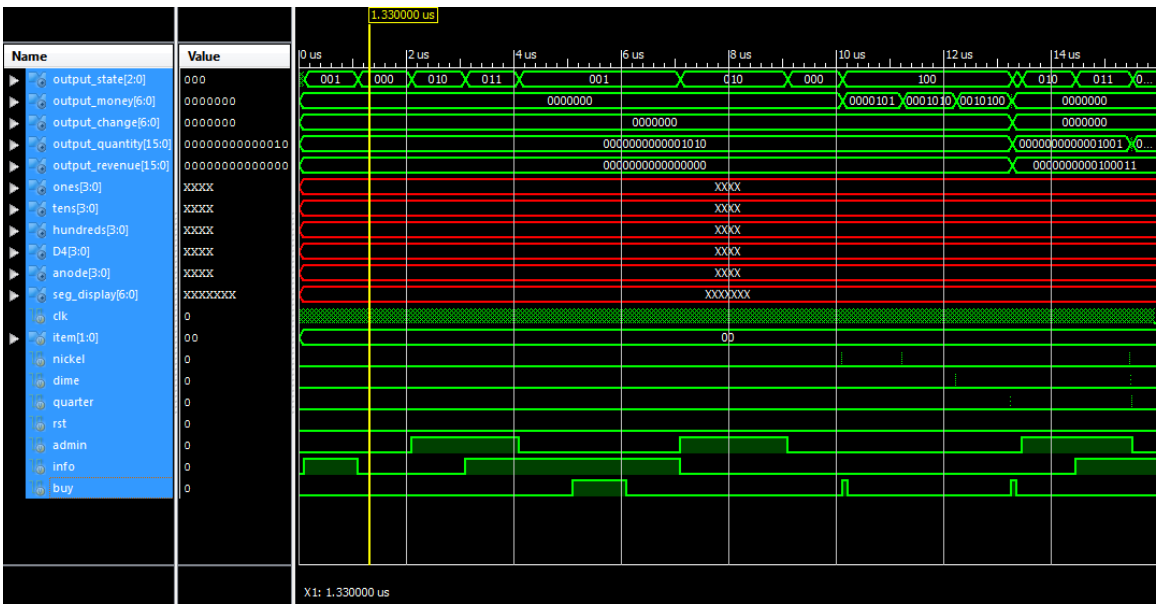

Figure 4: Simulation of Vending Machine

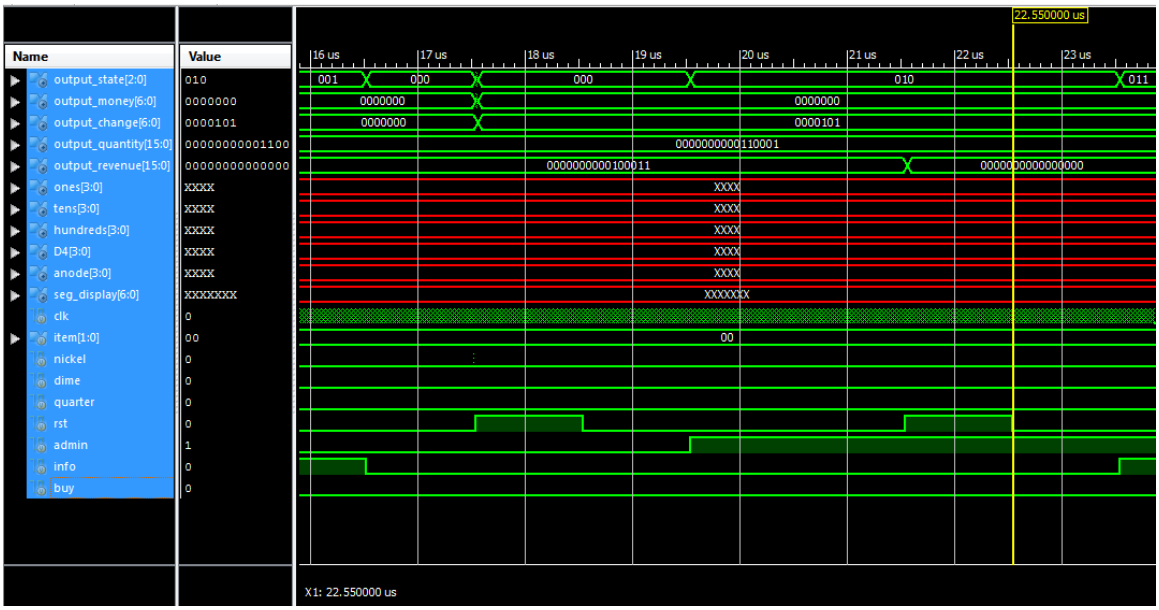

Figure 5: Continued Simulation of Vending Machine

Conclusion

One problem we encountered was implementing the change machine. We wanted to implement the change machine, so it always returns the least number of coins for a given amount. This proved more difficult to implement in Verilog because it required the use of Dynamic Programming and for synthesis to succeed in Verilog, it requires the amount of iterations for a loop to be run to be set. We abandoned this route and opted to make our own coin change algorithm, which usually returns the least number of coins.

Another problem we occurred was that we could not get the VGA display working. We consulted the example code given to us by the TA, but we still could not see what we did wrong. After many attempts of trying to fix our code, we abandoned our attempt to implement VGA, and focused our efforts to having an outstanding seven-segment display that required more complicated logic.

We also ran into issues of our alphabetical displays looking like our numerical displays. For example, our D would look like a 0 and OOS would look like 005, so we had to adjust our proposal and change some of the alphabet letters we wanted to display. We originally planned on displaying D for Dispense, but we changed it to “HERE” instead. For OOS, we changed it to “NONE.”