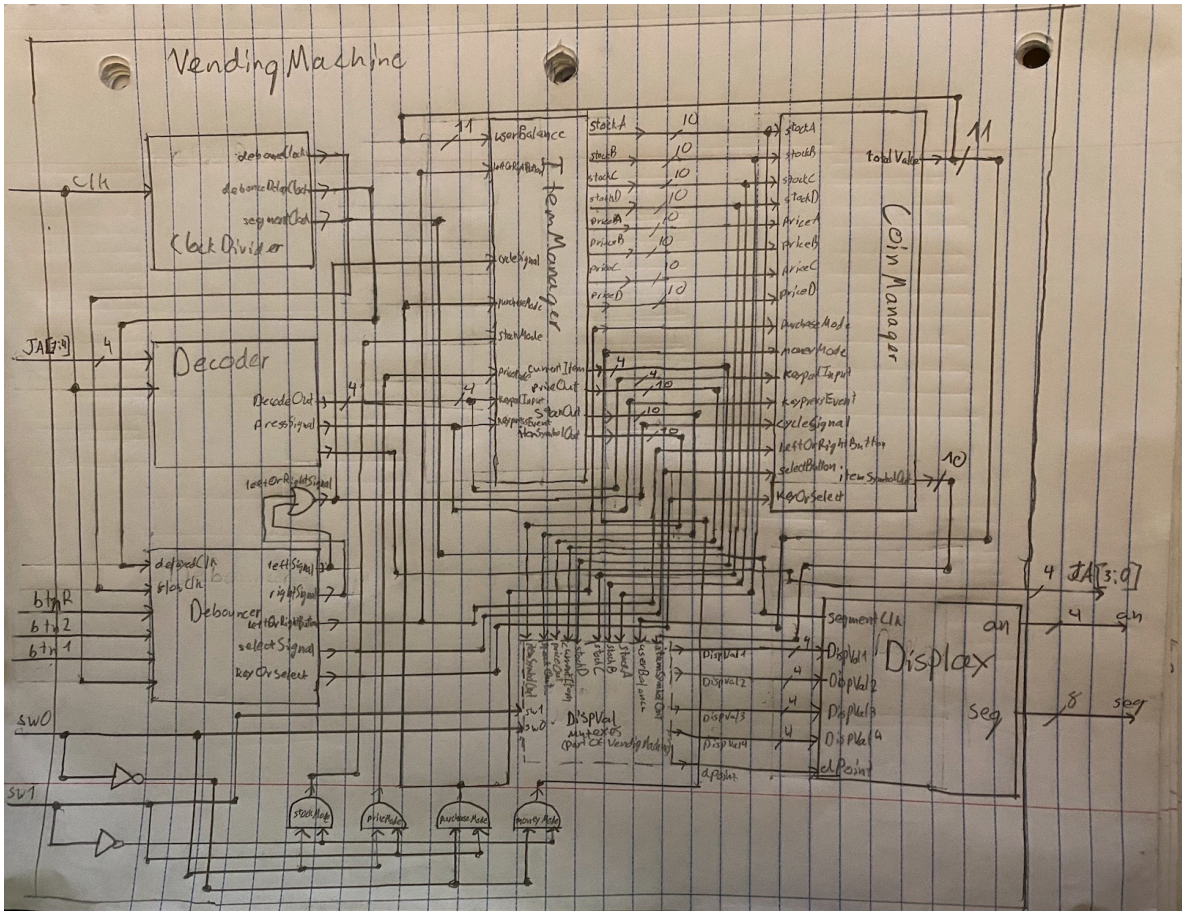
Sam Hopkins
Victor Chinnapan
Group 4
Final Lab (Lab 4)

Introduction and Requirement

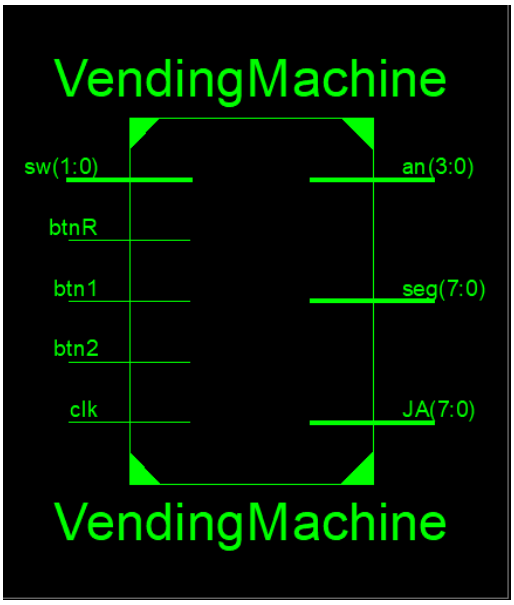
In this lab we used the Xilinx ISE software to design and implement a vending machine controller on the Nexys 3 Spartan-6 FPGA Board. Our design uses the board's seven-segment display to display product information and the user's current monetary balance. Switches can be used to switch between the various modes of the design. We have implemented a purchase mode, price mode, stock mode, and money mode. In price and stock mode, the user can use the buttons to cycle between products, and view their price and current stock. In money mode, the user can see their current balance, and use an attached keypad to add nickel, dime, quarter, or dollar amounts to their total. In purchase mode, they can choose which item they wish to purchase. If they have enough money and the item is in stock, the price of the item will be deducted from their balance and the stock of the item will decrement.

Design Description

Overall Design Diagram



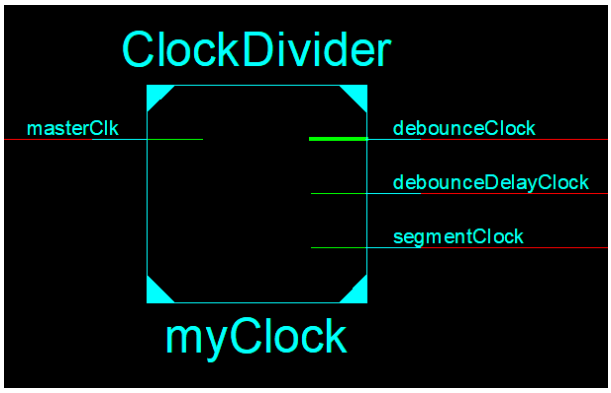
Full Circuit/VendingMachine Module

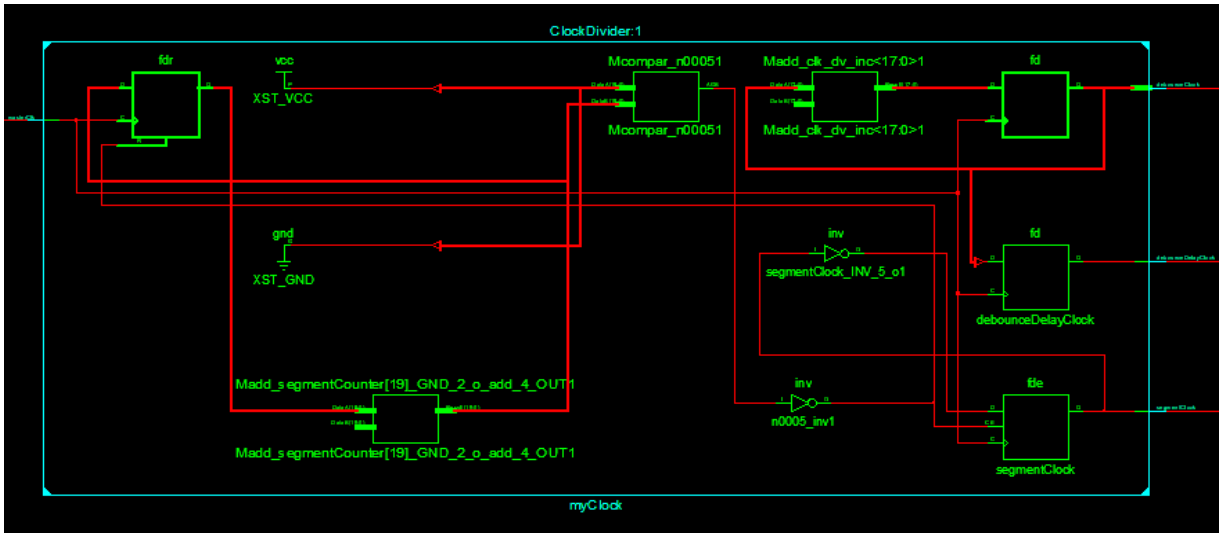


This module is responsible for connecting all the other modules together and choosing which current value to display. The inputs include two switches (sw0 and sw1), btnR and btn2 (cycle buttons), and lastly btn1 which is the select button. There is also an inout JA port (for the keypad). The last input is the 100 MHz clock (clk). There are two outputs (apart from the inout) which are the an and seg outputs needed for the seven segment display. Clk is passed to the clock module which outputs three other clocks. Two of the output clocks are routed to the Debounce module, and the last output tec clock is passed to the Display module. The buttons are passed to the Debounce module which produces signals for all of them as well as two one bit outputs. One stored a bit describing whether the last pressed cycle button was the left or right button and this value was passed to the ItemManager and CoinManager modules, including the left and right signals. The other one bit value is stored whether or not the selectButton was pressed (or a keypad button was pressed) and is also passed to the CoinManager and ItemManager modules with the selectSignal as well. The JA inout and 100 MHz clk are passed to the Decoder module to be used in order to decode keypad inputs. The outputs are Col (the column output to FPGA to be used in order to select the current checked column on keypad), DecodeOut which stored a 4 bit value of which button was pressed, and pressSignal which was set true whenever a button key was pressed. DecodeOut and pressSignal were passed to the CoinManager and ItemManager modules. The VendingMachine module maintains 4 wires each of which are used to determine the different modes (sw0 false and sw1 false for money mode, sw0 false and sw1 true for purchase mode, sw0 true and sw1 false for stock mode, and sw0 true and sw1 true for price mode). These wires are passed in to the ItemManager and CoinManager modules depending on need. There is also a leftOrRightsignal which is true whenever one of the cycle buttons is pressed and is also passed to the CoinManager and ItemManager modules. The ItemManager module takes in the inputs mentioned earlier, as well as the userBalance which is passed from the CoinManager, and outputs the 4 stock values, 4 price values, the currently selected items stock, price, and symbol,

as well as the index of the current time. The price and stock values are passed to the CoinMananger. The CoinManager takes in the values mentioned above and outputs just two values. The current input coins symbol and the user's balance. The last module that the VendingMachine module connects is the Display module which takes in 4 DisplayVals (1 for each seven segment display), dPoint (a value which stores whether or not the decimal point should be displayed) and the segmentClk. The outputs are an and seg which are passed to the FPGA. The 4 DisplayVals are stored in registers in the VendingMachine module and are updated every time a switch is flipped or a displayed value changed. Based on the current mode the digits of the selected value are stored in the DisplayVals registers. The digits are obtained by using the % operator for getting the ones, subtracting the previous value and % 100 for the tens (divide by 10), and the hundreds by subtracting the previous two values and dividing by 100. Overall, most of the logic are handled in the individual modules with only the DisplayVal logic done in the VendingMachine since the values used come from all the other modules.

ClockDivider Module

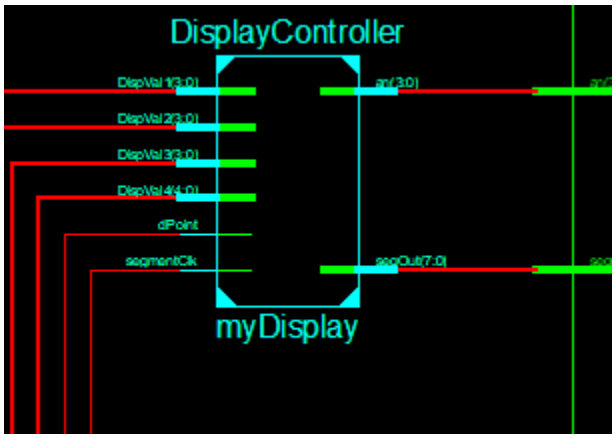


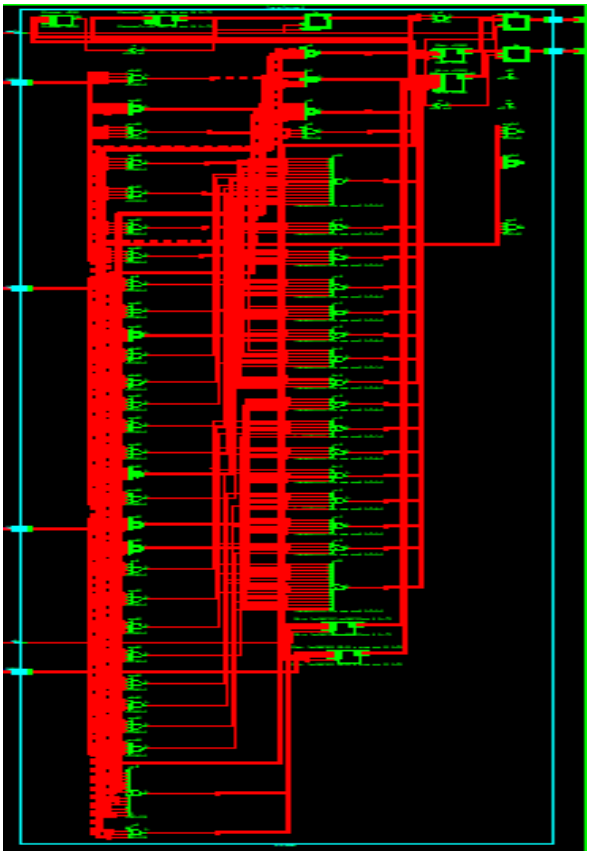


The clock module is responsible for converting the 100 Mhz master clock into three different clocks and outputting them. The input masterClock is the 100 Mhz clock and the

outputs are the debounceClock, debounceDelaylock, and segmentClock. These clocks are used for the debouncer and display modules. The segmentClock is created using a counter that flips the bit every 200000, which generates a 500 Hz clock. The debounceClock and debounceDelayClock are built the same way as the clocks provided in Lab 1, however, an extra bit was added to slow down the clock for better button signal filtering (originally 763 Hz now roughly half that). Overall, this project was more dependent on button presses and switches rather than clock signal, and the ClockDivider who converts the 100MHz clock is used to generate clocks for the Debouncer and Display modules.

Display Module

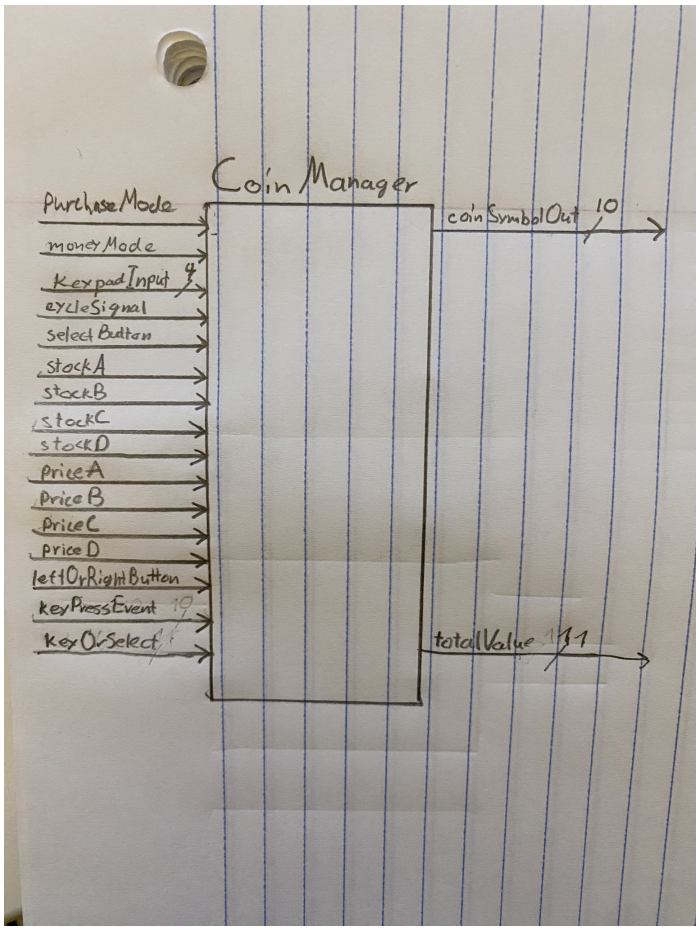




The Display module is responsible for displaying the inputted values on the Seven Segment Displays. The inputs are the 4 DispVals (values to be displayed), the segmentClk (a 500 Hz clock), and dPoint, which is 1 when the decimal point has to be shown and 0 if not. A 4 bit register segTurn stores which display is to be updated. Every clock one of the displays is updated depending on segTurn and segTurn is incremented (once it goes over 3 it loops back to 0). Every display is in charge of displaying a separate DispVal, and they do so using case statements. For the 3rd Display, there are two versions of the case statements: one without the decimal point, and one with the decimal point. If dPoint is enabled the version with the decimal point is used (and vice versa). The outputs are a 4 bit and which are selected based on segTurn

and 8 bit seg, which is assigned in the case statement based on the relative DispVal. Overall the Display module is a relatively simple module in charge of converting the DispVal inputs into seg outputs and looping through the 4 Displays to update them.

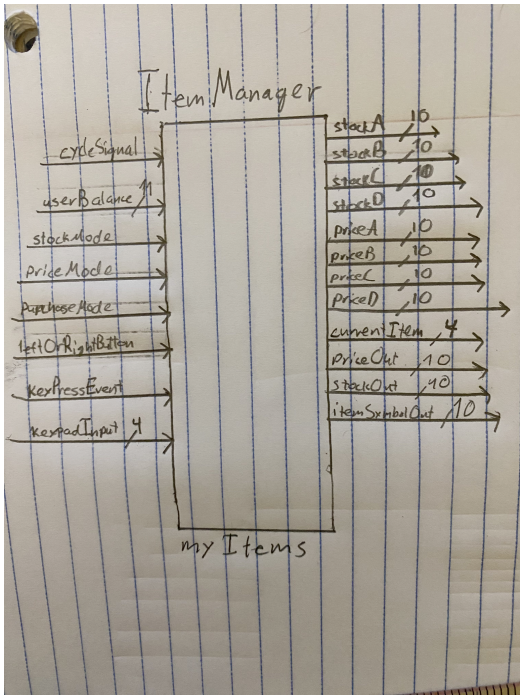
CoinManager Module



The CoinManager module is responsible for managing the user's balance. The inputs are the different modes, 4 stocks, 4 prices, button and keypad signals, keypad input, and two status values that determine whether a button or key was pressed last. Whenever cycleSignal is triggered (left or right button pressed, which we determine using the leftOrRightButton input), we either decrement or increment currentCoin, which is a variable that holds the index of the currently inputted coin type, and if currentCoin would surpass the max (3) we reset it 0 and if it would go below 0 then we set it to 3, in order to give a looping effect. To store the coin types and the values of those coins, a 2D array was created called coinData. The first entry of every subarray is the coin symbol name and the second entry stores the value of that coin (5 for nickel, 10 for dime, etc). There are 4 types of currencies (nickel, dime, quarter, dollar). The current coin value and current coin symbol are assigned currentCoinVal and coinSymbolOut (coinSymbolOut is an output of the module). Whenever we switch in or out of purchaseMode, we set 4 registers (changeA, changeB, changeC, changeD) to their appropriate dispensed change values based on whether the user's balance is >= than the price of that item and the

stock of that item is greater than 0. The reason the change is determined when getting into or out of purchaseMode is because if we were to do it during the key press in purchase mode, there would be synchronization problems since stocks and balance would change at the same time. This way we can avoid those issues. Item B in our case is a dummy item that we use to initialize the machine, hence the change value is hardcoded for it. Lastly whenever the selectSignal or keyPressEvent are fired (we combine these signals into one known as posEvent), we determine whether we are in moneyMode or purchaseMode and using the keyOrSelect input we can determine whether or not the event was a select button press or a key press (1 for select button, 0 for key). If the user is in moneyMode we first check if a variable rst is 1. If it is, we set it back to 0 and we reset the user's balance to 0. Next if the event was a keypress we set coinAddAmount register to the keypad input and the additionAmount register to coinAddAmount multiplied by currentCoinVal (the value of coin). This can continue until the user is satisfied with the amount they want to add. Next if the user presses the selectButton, we simply add the amount to the totalValue (an output of user's balance) unless it would cause the value to go over 999 (the max we can display on 3 seven segment displays). We also reset coinAddAmount and additionAmount to 0, to prevent subsequent select signals from adding more to the user's balance. The user can continue adding different types of currencies until they are satisfied with their balance. When the user switches to purchaseMode, the changes for each items are determines (as mentioned above), and if the user presses a key in this mode, additionAmount and coinAddAmount are set to 0 and totalValue is assigned the change value of that item (determined by what key was pressed). If an invalid key was pressed simply let totalValue remain unchanged. Rst is also set to 1, so the next time the user tries to do something in moneyMode, the user's balance is reset to 0. This lets the change remain on the screen until the user wants to continue purchasing items (or resets it via the selectButton). Overall the CoinManager does all the heavy lifting for managing the user's balance and simply outputs the user's balance and the current coins symbol out to the VendingMachine module in order for them to be displayed and used in other modules when needed.

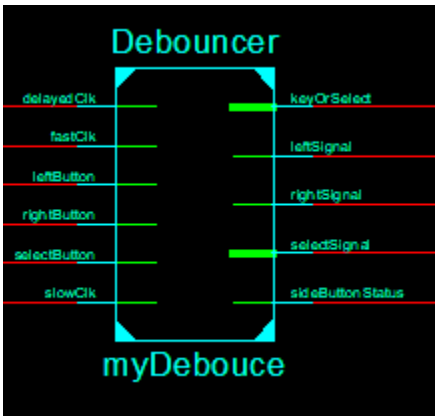
ItemManager Module

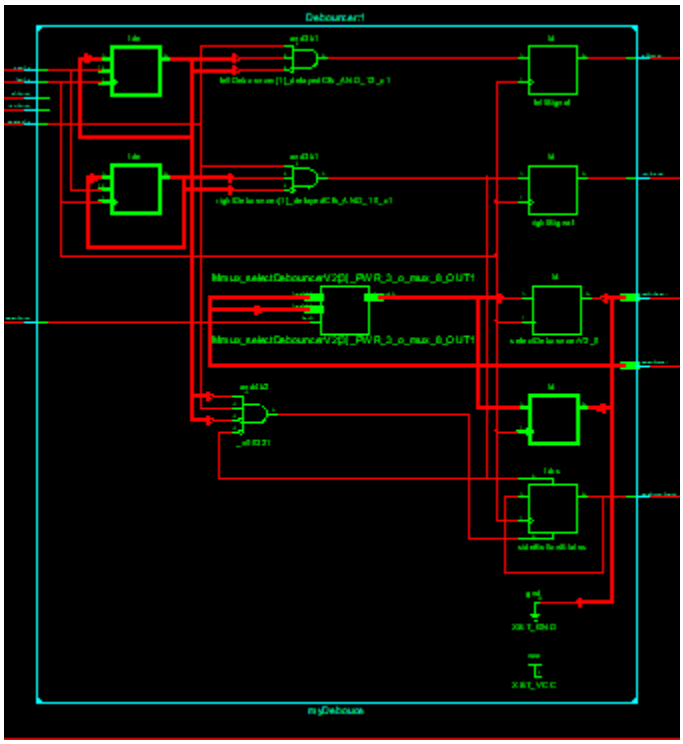


The ItemManager module is responsible for managing the stocks of the items in the machine. As mentioned earlier, item B is a dummy item used for initializing the machine. The inputs are the different modes (priceMode, stockMode, and purchaseMode), userBalance, which is passed from CoinManager, the different events (cycleSignal (left and right button) and keyPressEvent), as well as, keypadInput and the input value that determines whether left or right button was pressed (leftOrRightButton). When cycleSignal is triggered, we check whether we are in price or stock mode and if we are, we increment or decrement currentItem (using leftOrRightButton), and make it loop if it goes above the max or below 0. The data for each item is stored in a 2D array known as itemData. The first entry of each subarray is the symbol of each item, the second entry is the price of that item, and the third entry is the (starting) stock of that item. StockA, stockB, stockC, and stockD are output registers that store the current stocks of each item. PriceA, priceB, priceC, and priceD are outputs that are assigned the prices from itemData. Anytime any of the stocks change or currentItem changes, we assign the needed stock to stockOut, based on currentItem. Whenever we switch in or out of purchaseMode, we assign the new stock values to NewStockA, NewStockB, NewStockC, and NewStockD for each item, if the price of the item is less than or equal to the user's balance and the stock is greater than 0. Once again, this is done during the mode switch to avoid synchronization issues. When keyPressEvent is fired, check if the user is in purchaseMode. If the user is in purchaseMode then simply assign the NewStock of the selected item to the stock of the selected item (if A was pressed assign stockA with NewStockA). No further changes are needed. stockA, stockB, stockC, and stockD as well as the prices are outputted from the module and passed to the

CoinManager module. Furthermore, stockOut, priceOut, and itemSymbolOut are assigned to the values of the current item and used in VendingMachine to display them when necessary. Overall, the ItemManager is in charge of outputting the current items data, managing the stock, and outputting all the stocks and price values of all the items.

Debouncer Module

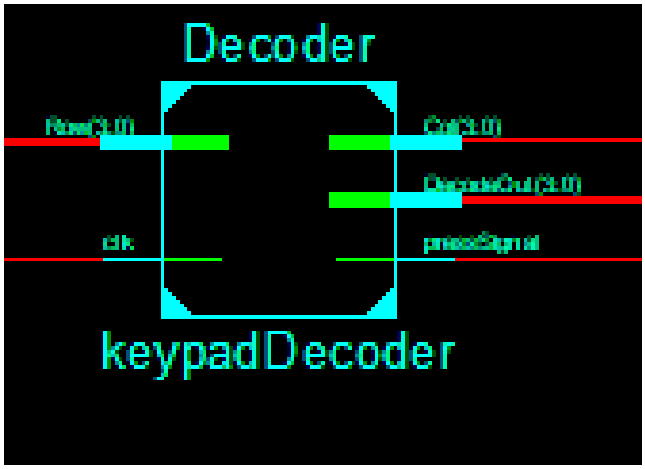


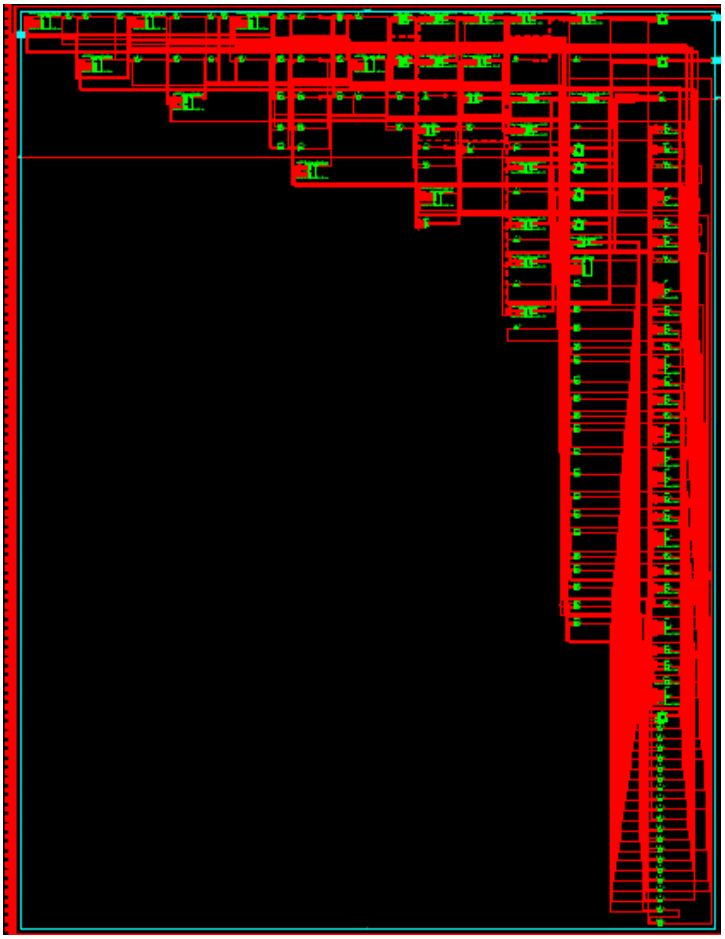


The Debouncer module is responsible for debouncing the button presses and generating one signal when a button is pressed. The inputs are fastClk (100 MHz clk), slowClk (the debounce clock from ClockDivider) and delayedClk (debounceDelayClk from ClockDivider). The debouncers for the left and right button function like the ones in Lab0. In short we have two 3 bit debouncers (1 per button), every fastClk, check for slowClk and if slowClk is true, push the current status of the button into their respective debouncer. Every time the LSB of the

debouncer is 0 and the middle bit is 1, and if the delayedClk is true, the signal is fired. Before the signal is fired, set sideButtonStatus to 1 if it's the right button pressed and 0 if the left button is pressed. The select button debounce works in a bit of a different way. A 5 bit debouncer is used. Whenever the select button is down, 1 is pushed into the MSB of the debouncer, however, if button up was detected, a 0 is pushed onto the LSB of the debouncer. The LSB of the debouncer determines the status of the signal. Before the signal is fired keyOrSelect is also set to the LSB of the debouncer. If keyOrSelect is 1 that means the signal fired is the select button, else it's a key press (or some other button). Overall, the debounce module is responsible for detecting button presses and eliminating noise from their presses, as well as, setting the value for determining which button was pressed.

Decoder Module





The Decoder module is responsible for detecting keypad key presses and decoding them to BCD. The keypad has 4 columns, each with 4 keys. The decoder has a 100MHz clk input and 4 bit row input. There are 3 outputs: a 4 bit Col output (to select which current column to check), a 4 bit DecodeOut output (the decoded BCD), and lastly the pressSignal which is fired anytime the key is pressed. The module cycles goes through columns every millisecond (first Col1, after 1 ms - Col2, etc). For every column the Row value is checked to see if any of the pins is at a logic low. If it is, a key press was detected and DecodeOut is set to the relevant

BCD the debounce for that column is set to 1 (col#Debouncer). Each of the 4 columns have their own 5 bit debouncer. Whenever no press is detected for that column, 0 is pushed into the MSB of the debouncer. The LSBs of all 4 debouncers are ORed together to create the pressSignal, thus whenever a key is pressed in any of the columns, pressSignal is fired. Overall, the Decoder module is in charge of detecting and decoding key presses and generating a pressSignal. DecodeOut is set before press signal fires, so the user is able to check the value and be sure its accurate any time pressSignal is fired.

Testing

This lab was mostly dependent on testing directly on the FPGA instead of simulations. Our team has used a simple simulation to try to simulate debouncing signals, however, it was far easier to set up a dummy variable which is to be displayed on the seven segment display and test debouncing using it instead (which proved to be successful). For example: increment the variable every button press. If the variable is consistently incremented the accurate amount the debouncing is working sufficiently well, however, if there are jumps, then that would mean that some adjustments/fixes are needed. Ultimately, the overwhelming majority was performed using live demos with a connected FPGA, because the design is reliant on the seven segment display, multiple switches, and button inputs.

Conclusion

In summary, our design is a vending machine controller that displays product information and allows the user to insert money and purchase products. Product names, prices, and current stocks are displayed on the seven segment display, as well as the user's current balance. Two switches are used to cycle through the various modes of the vending machine: stock, price, money, and purchase. Buttons are used to cycle through the available items and a keypad is used to insert variable amounts of money in multiple possible denominations.

One of the biggest difficulties we faced in implementing this design was debouncing. It wasn't an issue on the keypad, but our design made heavy use of the FPGA's buttons, which were extremely bouncy. We iterated through many debouncing methods, and finally found one that locked the button input for a small moment after a press. We had trouble at first making the array storing product data such as price and stock, but we were quickly able to figure that out. After some tinkering, we were able to solve updating the array's values as well. The final hurdle was that the user's balance would occasionally take on strange values after a purchase. We were able to solve this by inserting a dummy value into the array, which worked to stabilize the output after being cycled through. Overall, this was a complex design, but we were able to successfully implement it.