Lab 4 Report

By *Automagically*

Sean Murphy
U49850246
(33%)

Jesse Walton
U89823440
(33%)

Austin Matthew
U89904116
(33%)

Introduction:

For this lab we will be constructing a circuit to emulate the logic of a soda machine. As such, the soda machine is capable of counting "coins" placed into it and keeping track of the amount of inserted coins at all times. When the correct value of $0.45 is reached by any combination of denominations, the user is prompted to select a soda. If the user applied coins in a fashion that they went beyond the 45 cent cost of the soda, the soda machine will output "change" in the form of a 3 bit binary number.

We will also apply a debounce function to all of our buttons, however we would like to note that some of the buttons do not require a debounce, but it is added to them for thoroughness. Specifically our coin inputs will be debounced to avoid confusion within the system, but to reduce component cost we would consider not debouncing our Soda Select or Diet Select inputs, as they can not be hazardously affected by a bouncing switch. We will be using a Moore statement machine for this, as our outputs are based upon current state.

Design:

In implementing a robust, error-free finite state machine on the Atlys Spartan-6 FPGA, we were required to address many issues not previously considered. As a template for the finite state machine, the 16 state, stable Moore State Machine was used from within the Language Templates, found in the Xilinx Project Navigator.

The planning stage forced us to take into consideration all possible states and all possible transitions. An initial state diagram map was hand-drawn to include all value amounts between $0.00 and $0.65 at 5 cent increments. Additional states were added as needed, including a transitional wait state, a select soda state, a giving change state and finally, give regular and diet soda states. The total number of states used in the final design reached 20, however, earlier state diagrams had less.[1] Sequential encoding was selected over one-hot encoding merely as a means to save screen real estate. Using the Xilinx parameter keyword, relevant state names (e.g. state_30cents) were assigned to their corresponding 5-bit values, to improve readability.

Input encodings were constructed by considering the number of all possible inputs and assigning each a unique value. The total number of inputs was four, consisting of

[1] See Figure 4.1

quarter, dime, nickel and No Op and were encoded in two bits as 00, 01, 10 and 11 respectively.

Next, the final transition table[2] was constructed using the 20 states and each of the four possible inputs. As a result, each state had 4 transitional states, depending on the input. However, some states have identical transition states no matter the input.

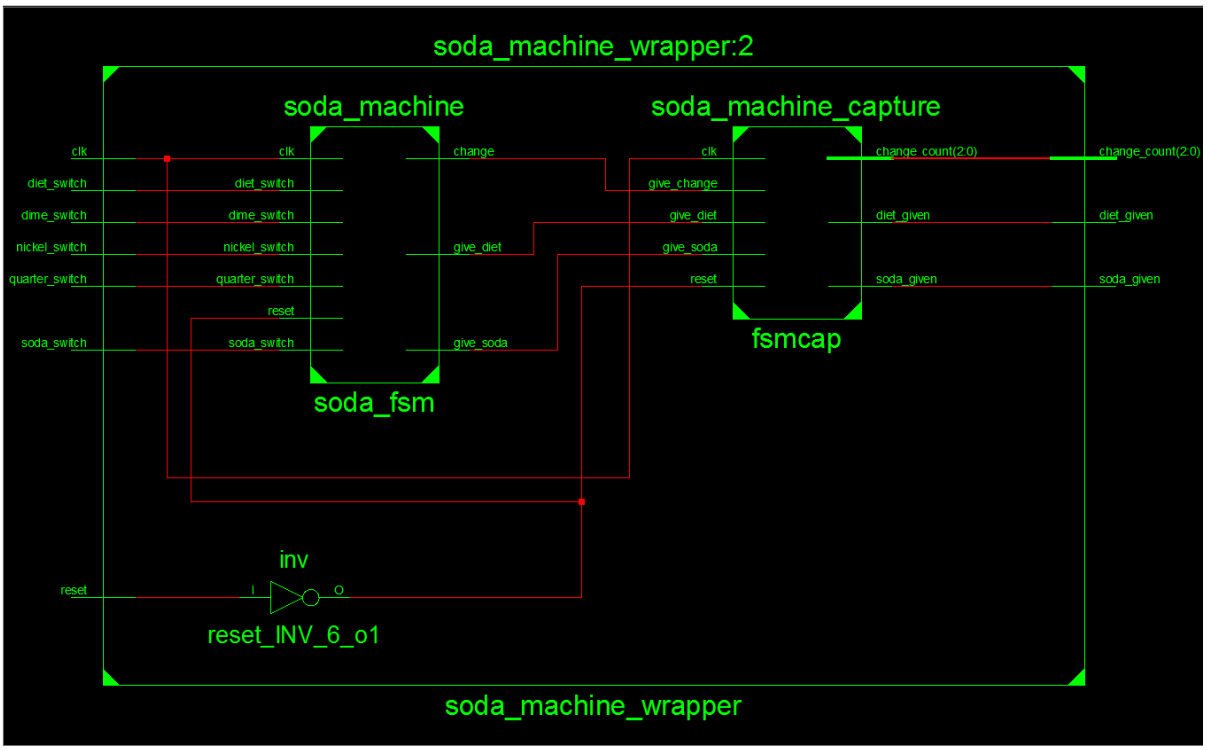


figure 4.0 - external schematic view

[2] See figure 4.3

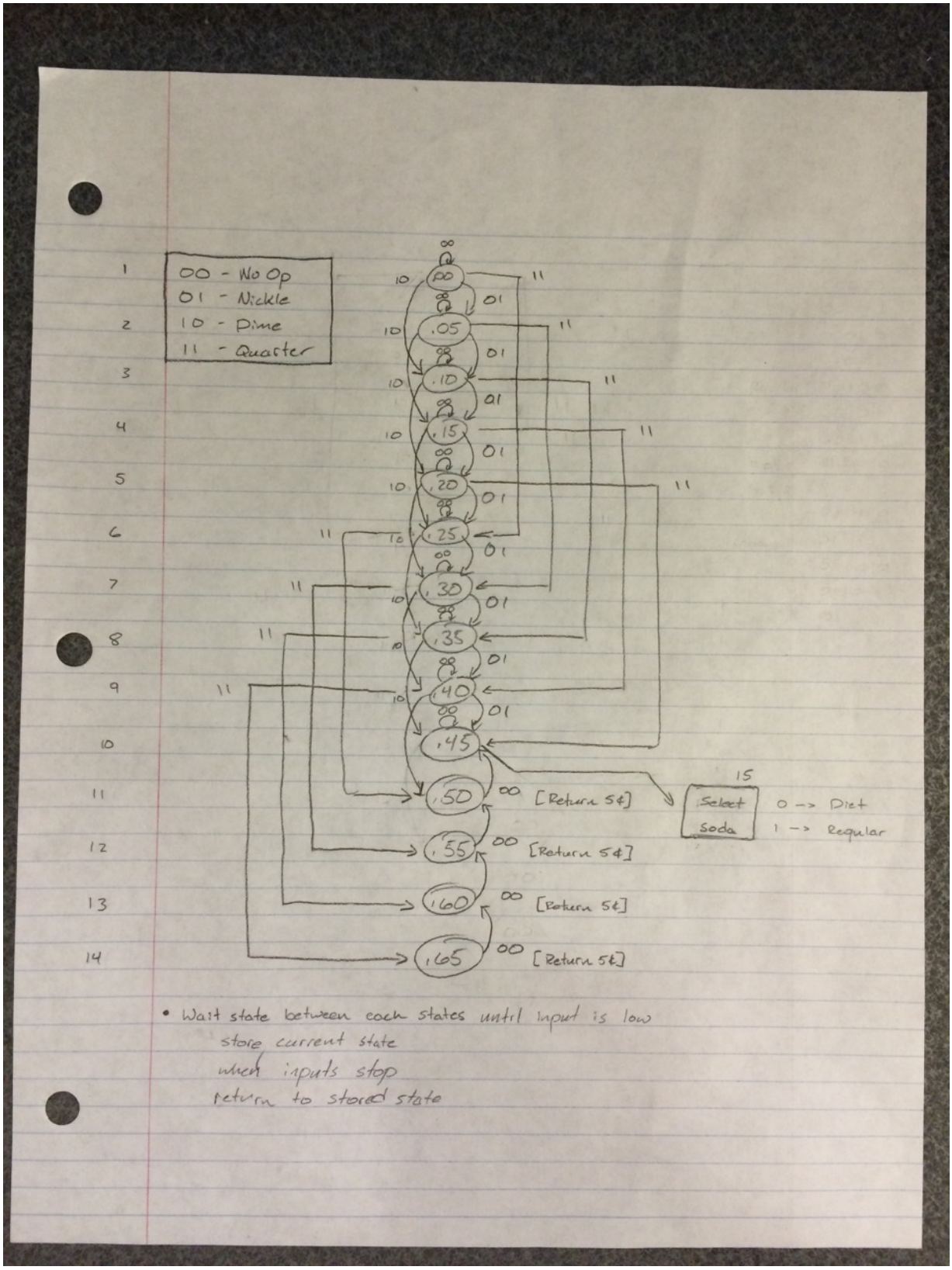
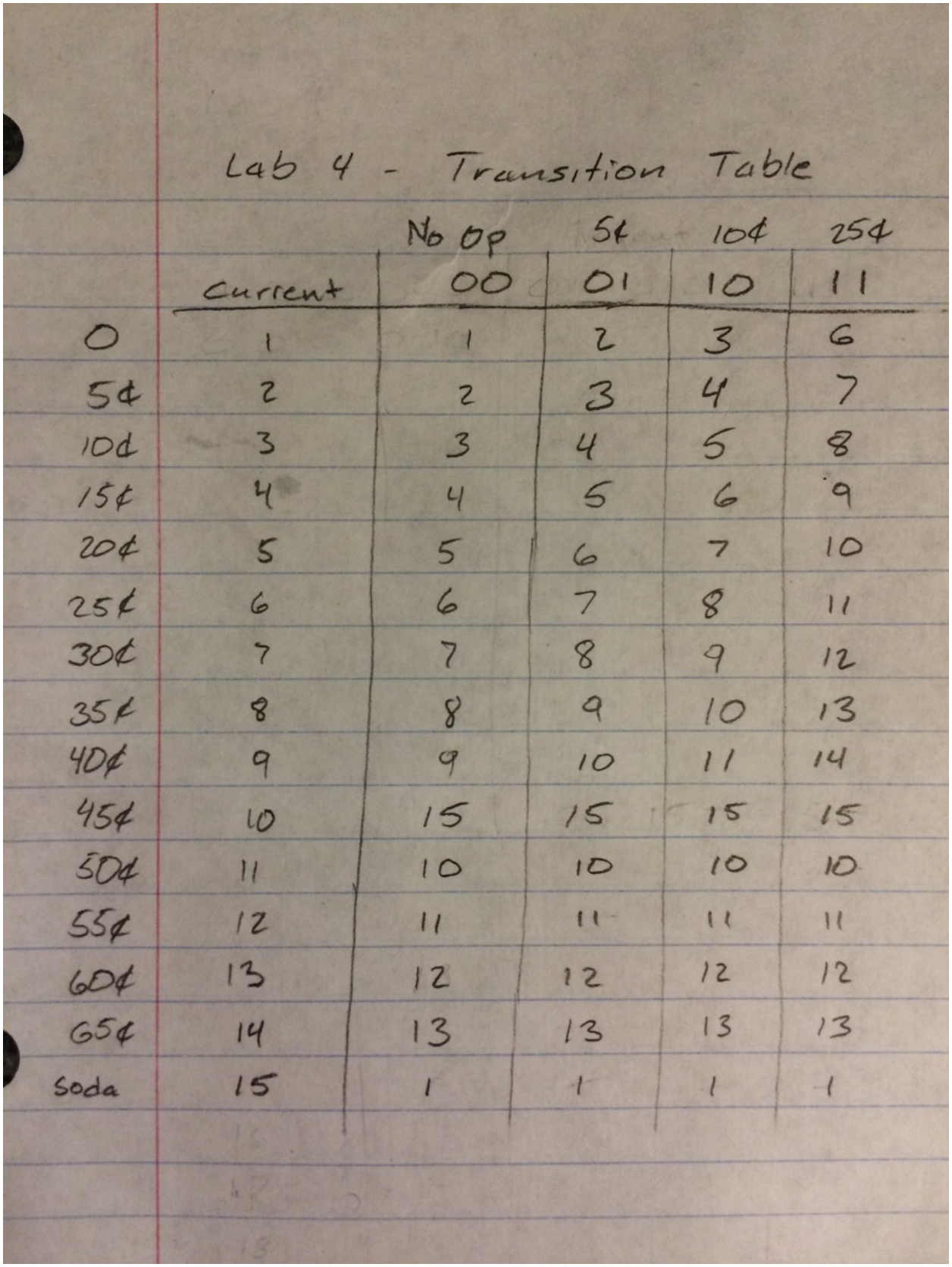


figure 4.1 - Initial State Diagram



Lab 4 - Transition Table

		No Op	5¢	10¢	25¢
	current	00	01	10	11
0	1	1	2	3	6
5¢	2	2	3	4	7
10¢	3	3	4	5	8
15¢	4	4	5	6	9
20¢	5	5	6	7	10
25¢	6	6	7	8	11
30¢	7	7	8	9	12
35¢	8	8	9	10	13
40¢	9	9	10	11	14
45¢	10	15	15	15	15
50¢	11	10	10	10	10
55¢	12	11	11	11	11
60¢	13	12	12	12	12
65¢	14	13	13	13	13
Soda	15	1	1	1	1

figure 4.2 - Initial Transition Table

	No Op (00)	$0.05 (01)	$0.10 (10)	$0.25 (11)
state_00cents	state_00cents	state_05cents	state_10cents	state_25cents
state_05cents	state_05cents	state_10cents	state_15cents	state_30cents
state_10cents	state_10cents	state_15cents	state_20cents	state_35cents
state_15cents	state_15cents	state_20cents	state_25cents	state_40cents
state_20cents	state_20cents	state_25cents	state_30cents	state_45cents
state_25cents	state_25cents	state_30cents	state_35cents	state_50cents
state_30cents	state_30cents	state_35cents	state_40cents	state_55cents
state_35cents	state_35cents	state_40cents	state_45cents	state_60cents
state_40cents	state_40cents	state_45cents	state_50cents	state_65cents
state_45cents	state_selectSoda			
state_50cents	state_45cents			
state_55cents	state_50cents			
state_60cents	state_55cents			
state_65cents	state_60cents			
wait_state	return_state			

state_selectSoda	*not based on standard inputs
	if (diet), state_giving_diet_pulse
	if (regular), state_giving_regular_pulse			

state_giving_diet_pulse	state_00cents			
state_giving_diet_pulse	state_00cents			
state_giving_regular_pulse	state_00cents			

[figure 4.3 - final Transition Table]

* Simplified transition table showing the return state for each cell.
Transition to the wait state is implied in between all transitions.

In planning, we also evaluated when and how we would accept inputs and process outputs.

Another issue to consider was how to handle a user holding down a change input button. In real life, this could be analogous to a quarter on a string that is held over the sensor. This issue was resolved by using an intermediate wait state between each transition. The state machine would then stay at the transition state until the input in question went low.

Button debouncing became a concern as the mechanical button inputs were used to simulate change. This issue was resolved programmatically by registering the rising edge of an input as high, and using a timer to ignore all reverberations within 65 milliseconds of the initial button press. This solution was implemented using the MIT button debounce module for Verilog. As a result, button presses only register one time per press.

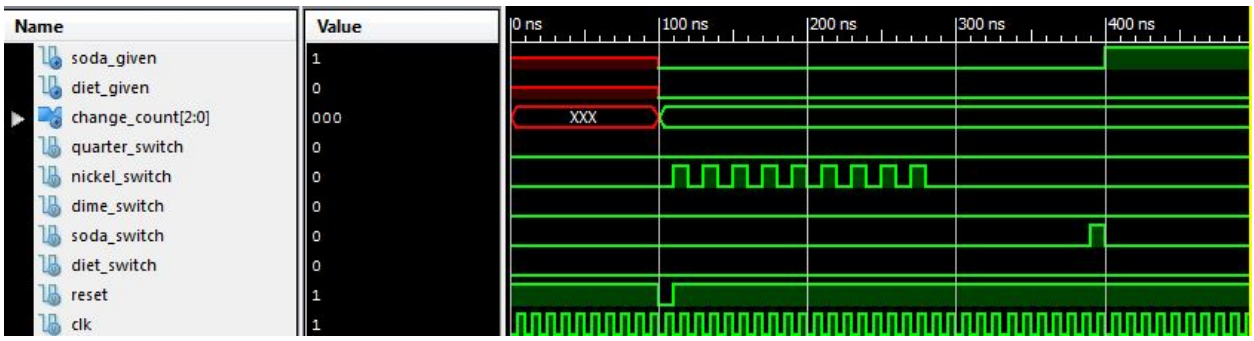
It is also worthwhile to note that multiple denominations of coins being 'inserted' simultaneously (represented as the simultaneous button inputs) is a condition that was not compensated for. It would be impossible to press two buttons at exactly the same time, and as such, whichever one was pressed first will be the input that is used.

Testing:

We took the following scenarios and tested them in our simulator and then again on our board. In the physical board tests, button presses were executed at about a one-per-second pace.

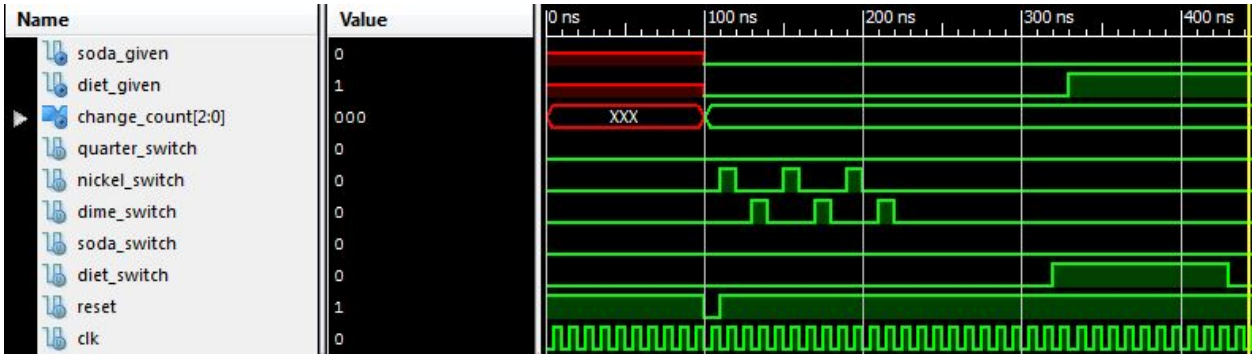
Ex. 1 - Exact Change

110ns: insert 9 x nickels
390ns: select regular soda
400ns: regular soda given



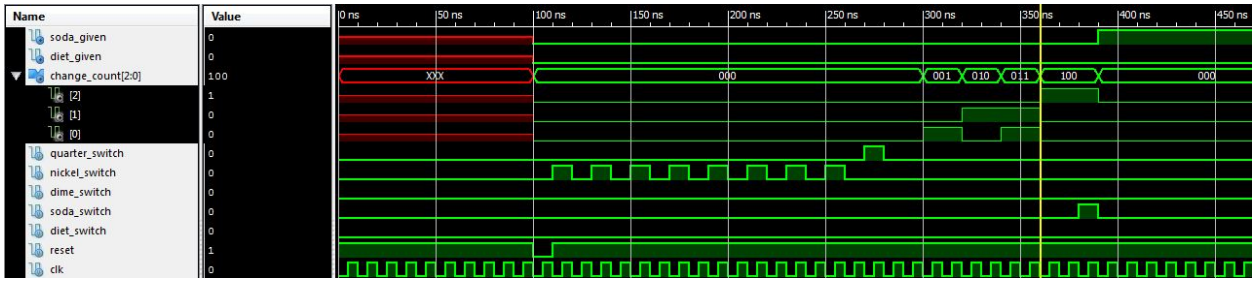
Ex. 2 - Diet

110ns: insert nickel and dime (repeat x3)
320ns: select diet soda
330ns: diet soda given



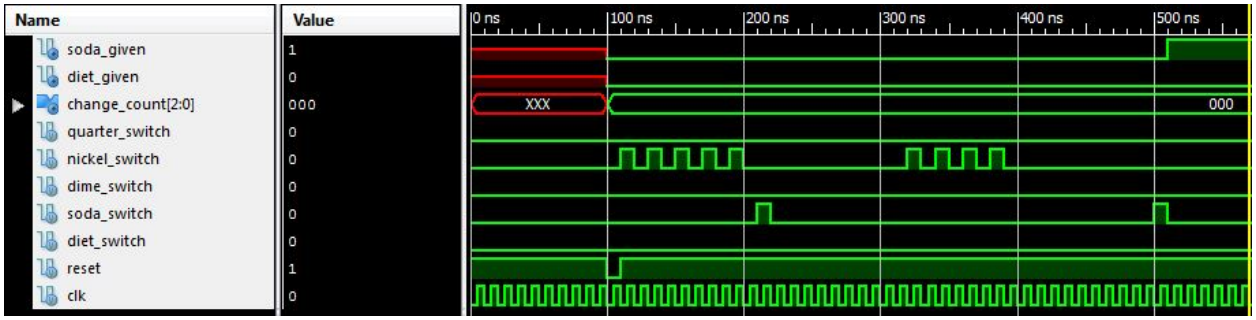
Ex. 3 - 20 cents over

100ns: insert nickel (repeat x8)
270ns: insert quarter
300ns: give change (repeat x4)
380ns: regular select soda
390ns: regular soda given



Ex. 4 - Early Soda Selection

100ns: insert nickel (repeat x5)
210ns: select regular soda
320ns: insert nickel (repeat x4)
500ns: select regular soda
510ns: regular soda given

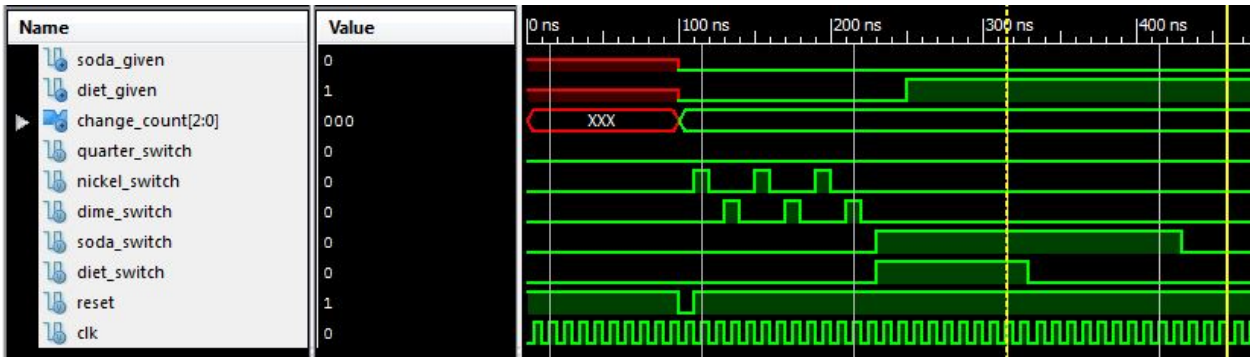


Ex. 5 - Attempts to gain more than one soda

100ns: insert nickel and dime (repeat x3)
230ns: select diet soda
250ns: diet soda given
330ns: select regular soda

Ex. 6 - Press both soda buttons at the same time

100ns: insert nickel and dime (repeat x3)
230ns: select diet soda and regular soda
250ns: give diet soda
300ns: stop being a fatty



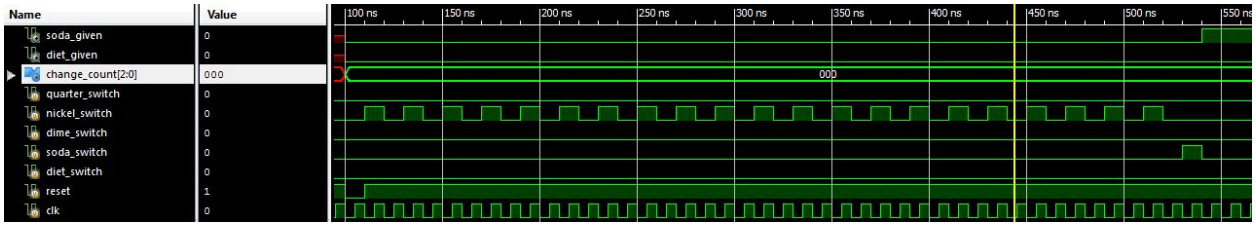
Ex. 7 - Cases of improper input by the user

100ns: insert nickel (repeat x21)

530ns: select regular soda

550ns: give regular soda

* excess money is **not** returned

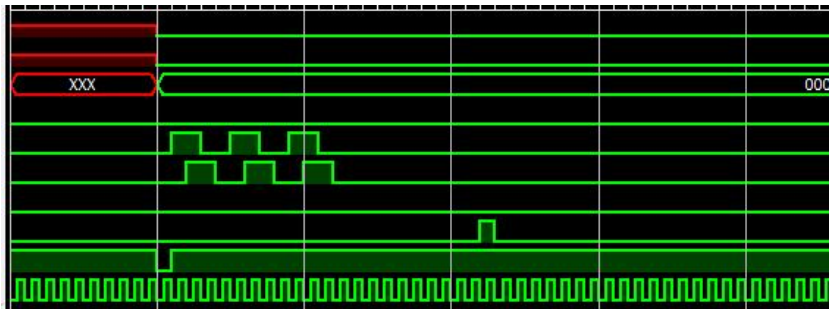


Ex. 8 - Ultra fast coin insert

100ns: insert nickel

110ns: insert dime

(repeat x3)



Results:

All test we ran on the simulator were ran again on the board. Each case worked exactly the same as it had in simulation, except for the 'Ultrafast coin insert' example. In this case, the inputs were triggered within 10 nanoseconds of the previous input's falling edge. As this is a situation we don't expect any user to ever be able to encounter, we chose to not address the problem and left it as is.

Conclusion:

The state machine implemented on an FPGA is an interest concept. Each state is a subroutine in a sense, and within each subroutine/state its given a way to "jump" to another subroutine depending on input. There is a lot of power and flexibility provided by being able to maintain your current state. Also, this turned out to be one of the easier systems to design as you're required to lay out all necessary states, and consider the effect that all possible inputs would have on them. As a result, the finite state machine is, if hardware level glitches are not considered, very robust in their execution. There is no way to get lost going from point A to point B, or any other endpoint as *all* possible situations are pre-programed.

This project in particular has really given us a better understanding of how advanced and helpful the automated design tools actually are. The schematic diagram viewing tool within Xilinx is an incredible view to the internals of the machines we are creating and how it is constructed on a gate-level. With consideration of how the FPGA works and how functions are implemented with LUTs, it would be impossible to plan out and execute a project like this in the time span of a week. However, with the Xilinx Project Navigator, it is possible to create such devices that are able to function at incredible speeds and be updated or reprogrammed again for something else when finished. As a result of these design tools and the flexibility of the FPGA board, many projects that would otherwise be daunting, now seem trivial by comparison.