University of Michigan Dearborn

Experiment 6 Report

ECE 475

Kevin Zaka
3-29-2020

Objectives

The objective is to design a vending machine and implement it using VHDL and JK flip-flops.

Concepts

It is common practice to design somewhat complex circuits using Finite State Machines (FSM) which produce an output based on the input and/or on the current state (if Mealy). These FSM are implemented with flip-flops which hold the current state and generate the next state. Either type of flip-flop can be used to implement a FSM. States can be encoded using different methods:

- One-hot encoding: one flip-flop used per state
- Binary encoding: the number of flip-flops needed is equal to $\lceil \log_2(\#states) \rceil$
- Gray-code encoding, etc.

Let's recall how a JK flip-flop works. The operation (truth) table for a JK flip-flop is as following:

J	K	Q^+
0	0	Q (no change)
0	1	0 (reset)
1	0	1 (set)
1	1	Q' (toggle)

Table 1: JK flip-flop truth table

Using the table above, it is possible to determine the set of inputs that cause the JK flip-flop to switch from the present state to the next state, called the excitation table:

Q	Q^+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Table 2: JK flip-flop excitation table

The excitation table is used to determine the J & K inputs when the present state and next state are known, once the state diagram has been designed.

Requirements

The vending machine only accepts $0.10, $0.25, and $1.00 and offers four items at $0.45, $0.75, $0.80, and $1.00. Change must be always displayed and only one item can be dispensed at a time. The State Machine controlling the vending machine must be implemented using JK flip-flops.

Design

This kind of circuit has too many possible combinations of inputs to build an exhaustive state diagram and table for all possible inputs and states. Therefore, the state transitioning is implemented as an Algorithmic State Machine (ASM). The ASM is a mixture of a state diagram and a flowchart. However, first, write how the circuit behaves in pseudocode. The more similar the pseudocode is to RTL, the easier it is to implement the circuit. The circuit is designed in two parts: the Control Logic and the Datapath.

The Control Logic sends signals to the datapath components and the datapath sends signals to the control logic upon which the control unit determines the next action. The control unit, in this case, is implemented as a State Machine and the datapath part is a combinational circuit.

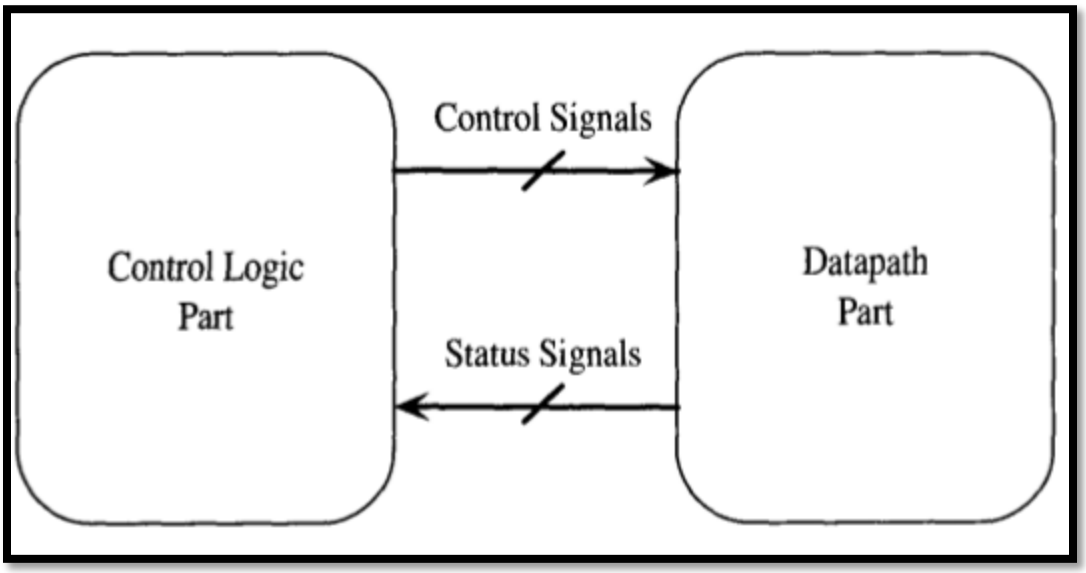

Figure 1: Circuit Design

1. Reset
2. Select Item
    - $ITEM \leftarrow I_1 I_0$
    - $\rightarrow (IS\ ITEM\ SELECTED, IS\ ITEM\ NOT\ SELECTED)|(3,2)$
3. Input Money
    - $INPUT \leftarrow IP_1 IP_0$
    - $IS\ MONEY\ IN \leftarrow TRUE$
4. Add to Credit
    - $TOTAL \leftarrow TOTAL + IP_1 IP_0$
5. Compare Total to Cost
    - $\rightarrow (COST > TOTAL, COST =< TOTAL)|(6,3)$
6. Dispense Change
    - $CHANGE \leftarrow TOTAL - COST$
7. Dispense Item
    - $ITEM \leftarrow I_1 I_0$
    - $\rightarrow (1)$

The next step is to turn the pseudocode into a flowchart for a visual representation. An ASM chart consists of states (and conditions leading to those states) as well conditional outputs.

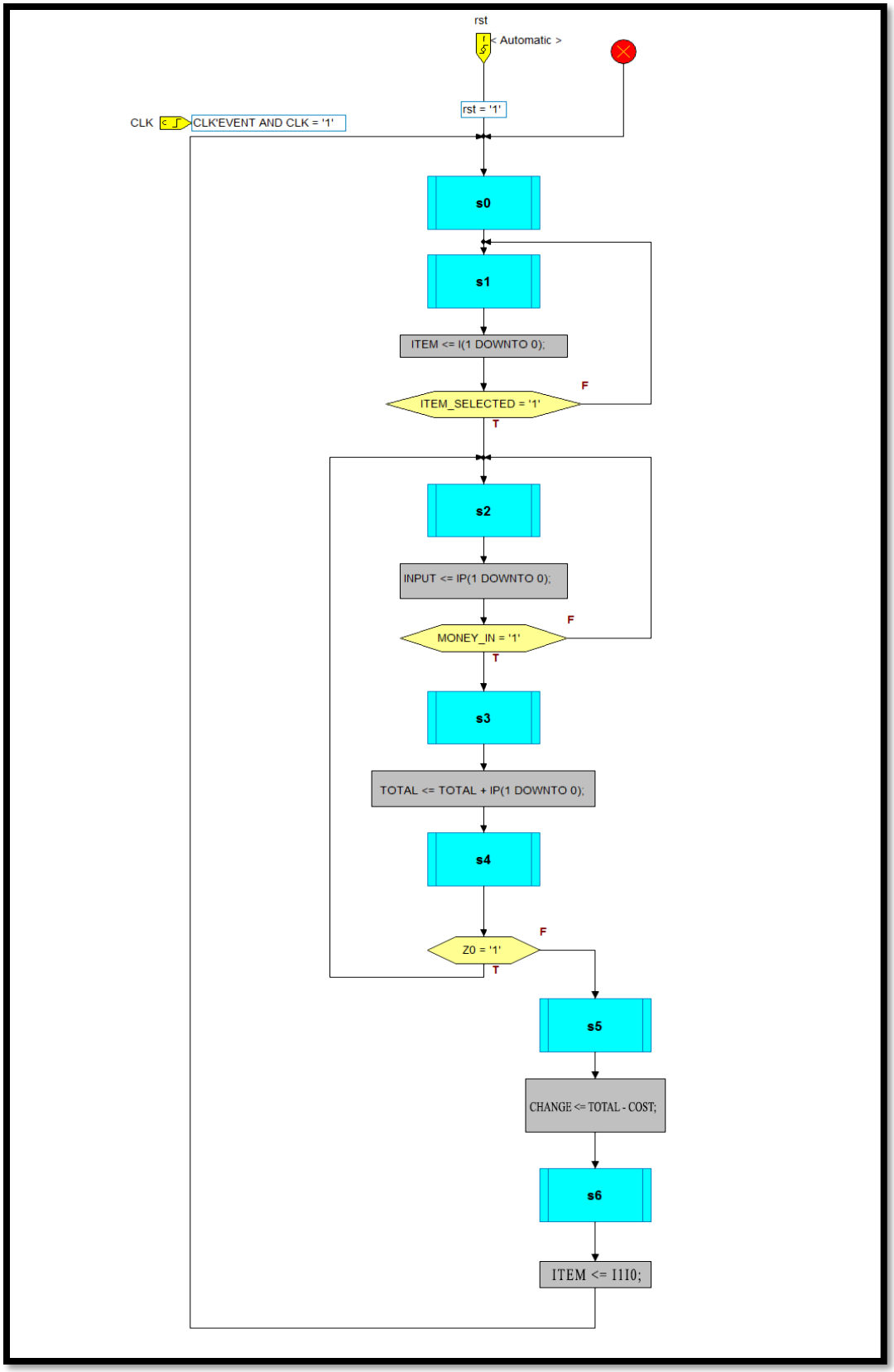

Figure 2: Flowchart

It is now easier to build a datapath circuit. The following circuit was designed in Logisim:

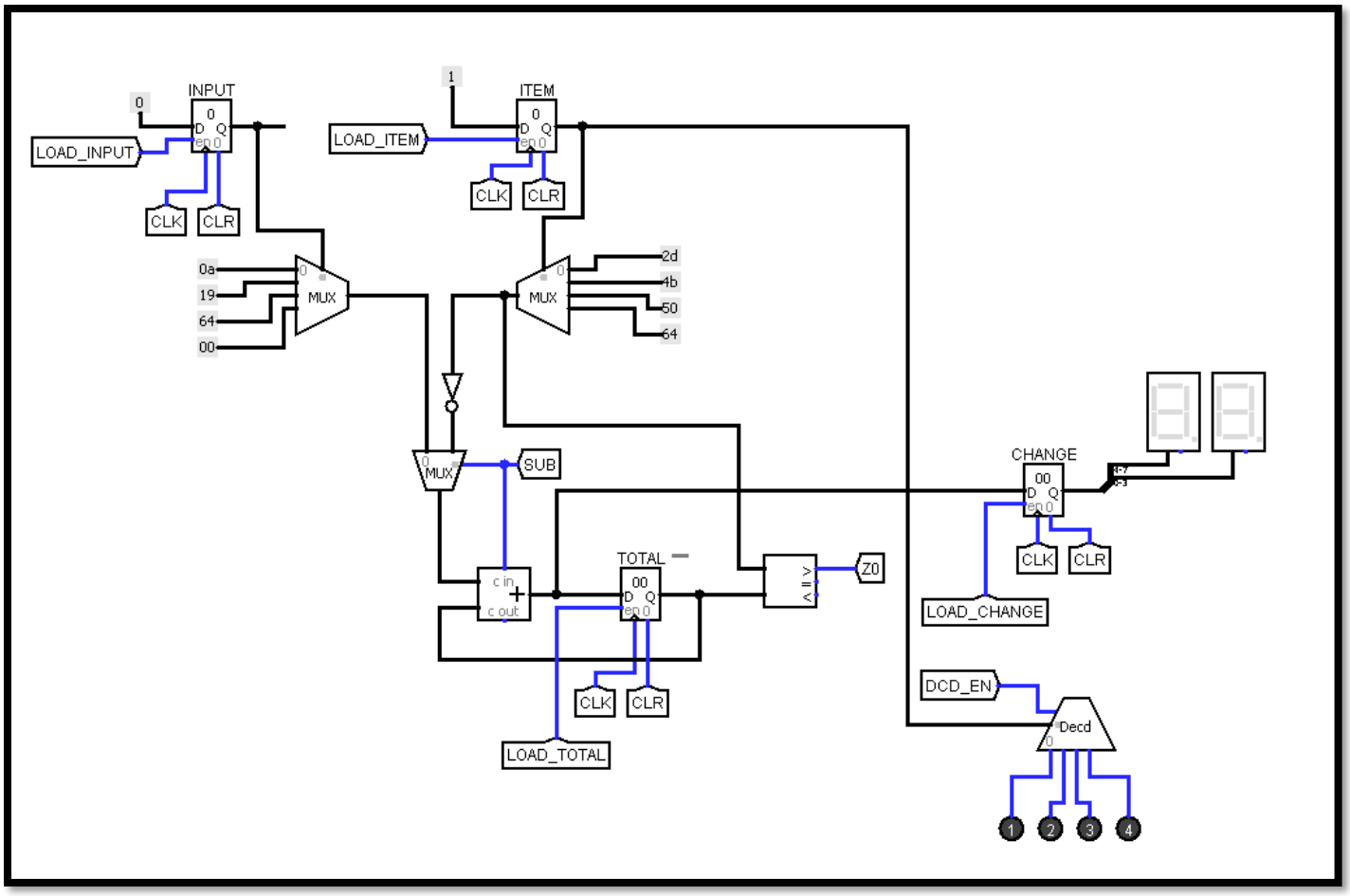

Figure 3: Combinational Datapath

In the circuit above, the user first selects the item he/she wants to buy; this value is stored in the "ITEM" register; it is a 2-bit value which selects one of the options in the MUX below it. Similarly, the user enters the money, which, in the final product, is scanned by a sensor and encoded to a value the circuit understands. In this case, the encoding is a 2-bit value which selects which one of the three allowed money inputs the user put in and selects is in the MUX below the "INPUT" register. The data is loaded onto these registers only when the appropriate signals are asserted ("LOAD_ITEM" and "LOAD_INPUT"). These monetary values are 8 bits in length to accommodate the highest value possible which could happen when the user has selected an item that costs $1.00, has input $0.95, and then the user inputs an additional $1.00, leading to the necessity of enough storage for $1.95 (195 = 11000011b).

The money value is added onto a register which accumulates the credit the user has for the session. Each time money is added onto credit, it is compared with the cost of the item selected through the comparator to the right of the "TOTAL" register. The comparator reports if the cost is greater than the credit amount; if yes, the user is asked to enter more money. If not, change is calculated and displayed, and the item ejected.

The change is calculated by subtracting the credit amount from the item cost. The item cost is first inverted bitwise to get 1C notation, and then the signal “SUB” is asserted to set the adder’s carry in to ‘1’. By adding ‘1’ to the 1C, 2C notation is achieved. The 2C notation is selected when SUB = 1. The “LOAD_CHANGE” signal is asserted so the subtraction result is loaded onto the change register. In the simulation above, the change is displayed using a 7-segment HEX display. The item to be ejected is decoded using the 2-bit value and a 2-to-4 decoder to select one of the items. The lines are connected to LEDs in the simulation above to show how this circuit could be implemented in real life.

The ASM is updated with the necessary signals to assert:

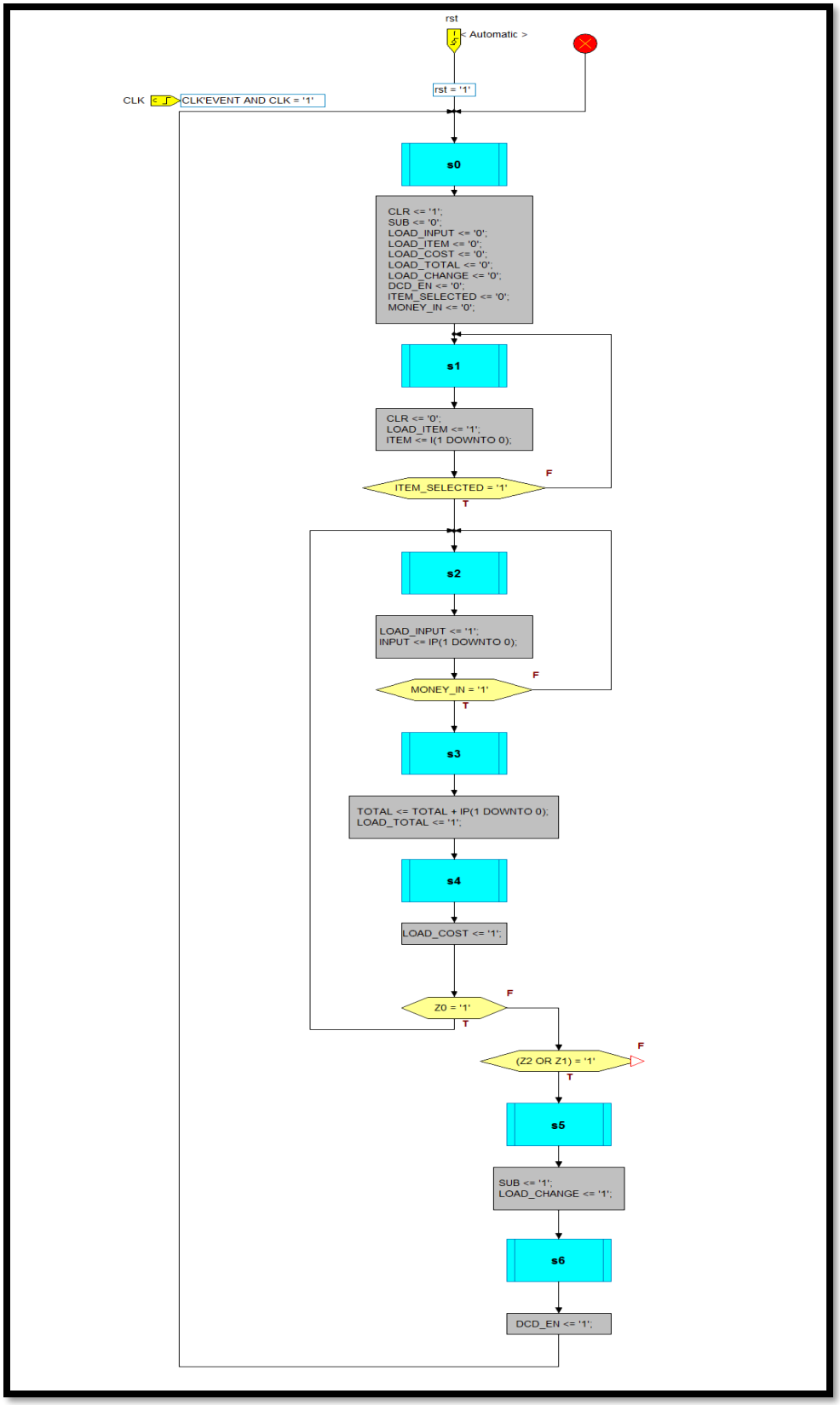

Figure 4: Updated ASM

All the possible paths are taken into consideration and the following table is built:

Link Path	PS	INPUT			NS	OUTPUT						
		ITEM_ SELECTED	MONEY_ IN	GREATER_ THAN		CLR	SUB	LOAD_ INPUT	LOAD_ ITEM	LOAD_ TOTAL	LOAD_ CHANGE	DCD_ EN
1	S0	X	X	X	S1	1	0	0	0	0	0	0
2	S1	1	X	X	S2	0	0	0	1	0	0	0
3	S1	0	X	X	S1	0	0	0	1	0	0	0
4	S2	X	1	X	S3	0	0	1	0	0	0	0
5	S2	X	0	X	S2	0	0	1	0	0	0	0
6	S3	X	X	X	S4	0	0	0	0	1	0	0
7	S4	X	X	1	S2	0	0	0	0	0	0	0
8	S4	X	X	0	S5	0	0	0	0	0	0	0
9	S5	X	X	X	S6	0	1	0	0	0	1	0
10	S6	X	X	X	S0	0	0	0	0	0	0	1

Table 3: State Table

To determine the most efficient encoding for the states, the following K-MAP is built:

		Q2Q3			
		00	01	11	10
Q1	0	S0	S1	S2	S3
	1	S6	S5	S4	

Table 4: State Encoding

The states are encoded as following:

- $S_0 = 000$
- $S_1 = 001$
- $S_2 = 011$
- $S_3 = 010$
- $S_4 = 111$
- $S_5 = 101$
- $S_6 = 100$

Using these encodings, the following table is built with the JK inputs for each flip-flop

PS	NS	J_1K_1	J_2K_2	J_3K_3
000	001	0X	0X	1X
001	011	0X	1X	X0
001	001	0X	0X	X0
011	010	0X	X0	X1
011	011	0X	X0	X0
010	111	1X	X0	1X
111	011	X1	X0	X0
111	101	X0	X1	X0
101	100	X0	0X	X1

100	000	X1	0X	0X

Table 5: JK Inputs

From the table above, the equation for the J & K inputs as well as the ASM outputs are derived:

- $J_1 = Q_2 Q_3'$
- $K_1 = GREATERTHAN(Q_1 + Q_2) + \ Q_3{}'$
- $J_2 = Q_3 Q_1{}'(ITEMSELECTED)$
- $K_2 = Q_1(GREATERTHAN)$
- $J_3 = Q_1' + Q_2$
- $K_3 = Q_1 Q_2' + Q_1' Q_2(MONEYIN)$
- $CLR = Q_1' Q_2' Q_3'$
- $SUB = Q_1 Q_2' Q_3$
- $LOADINPUT = Q_1' Q_2 Q_3$
- $LOADITEM = Q_1' Q_2' Q_3$
- $LOADCHANGE = Q_1 Q_2' Q_3$
- $DCDEN = Q_1 Q_2' Q_3'$
- $LOADTOTAL = Q_1' Q_2 Q_3'$

The updated State Diagram looks like the following:

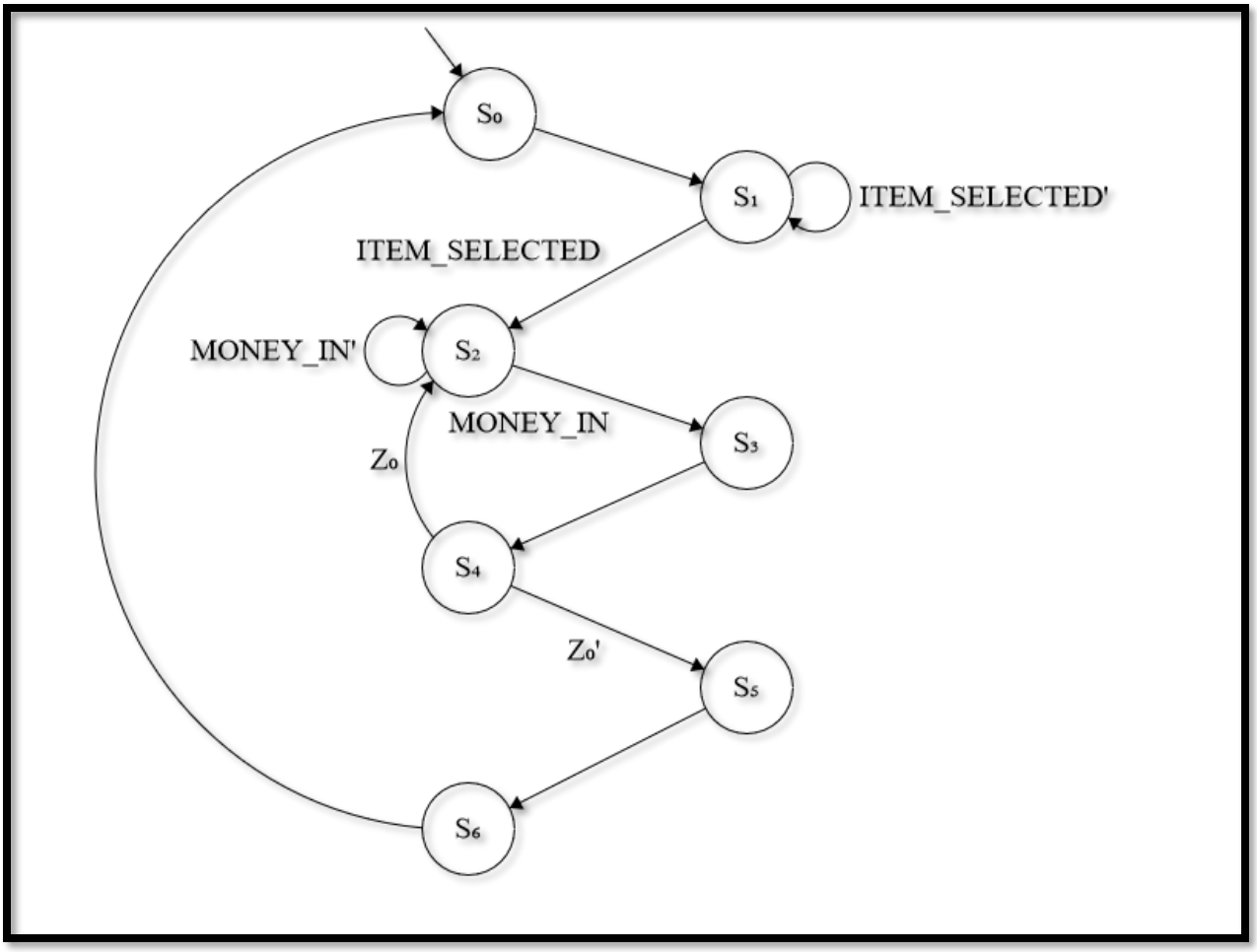

Figure 5: State Diagram

The next state generation circuit looks like:

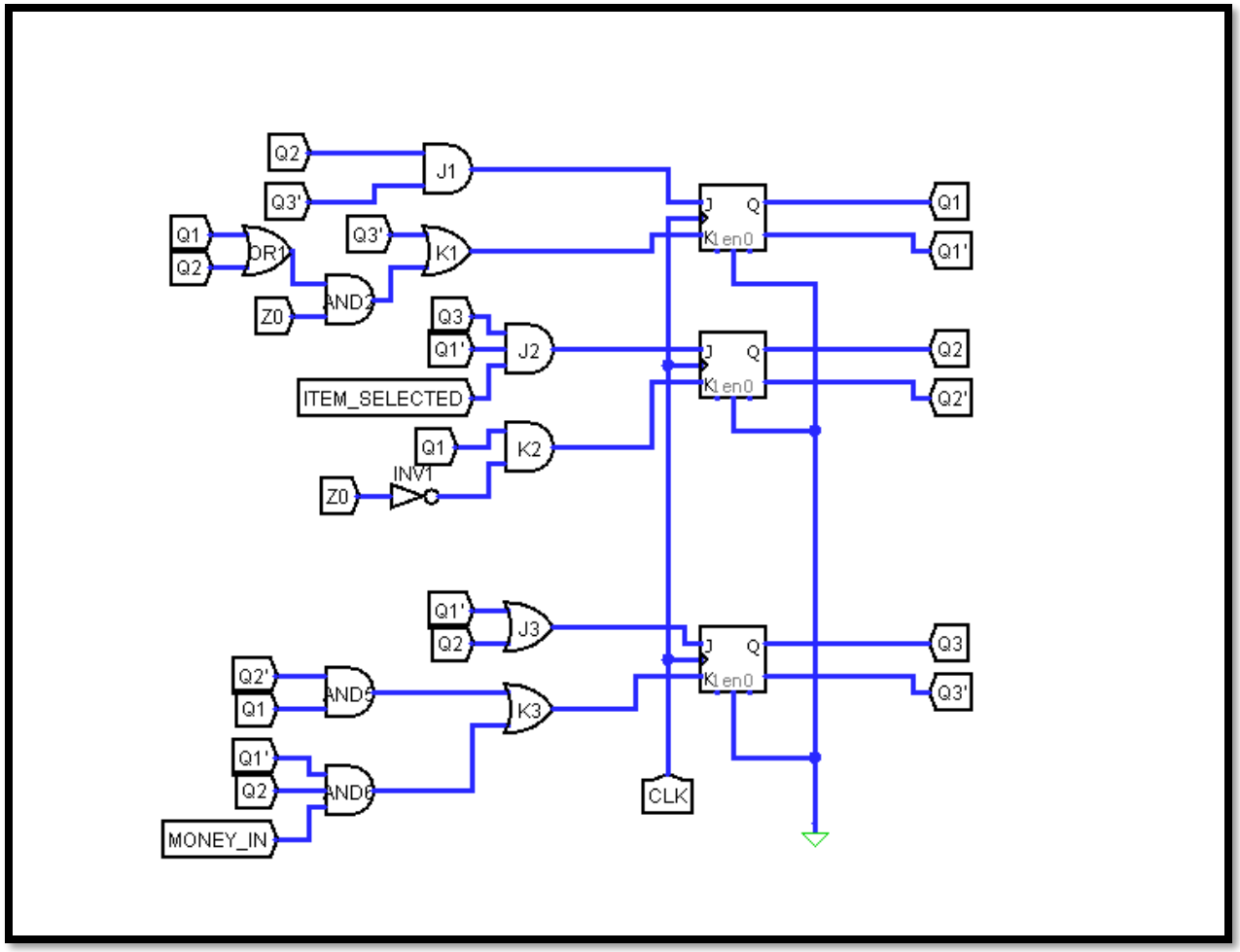

Figure 6: State Generation

And the outputs wiring looks like:

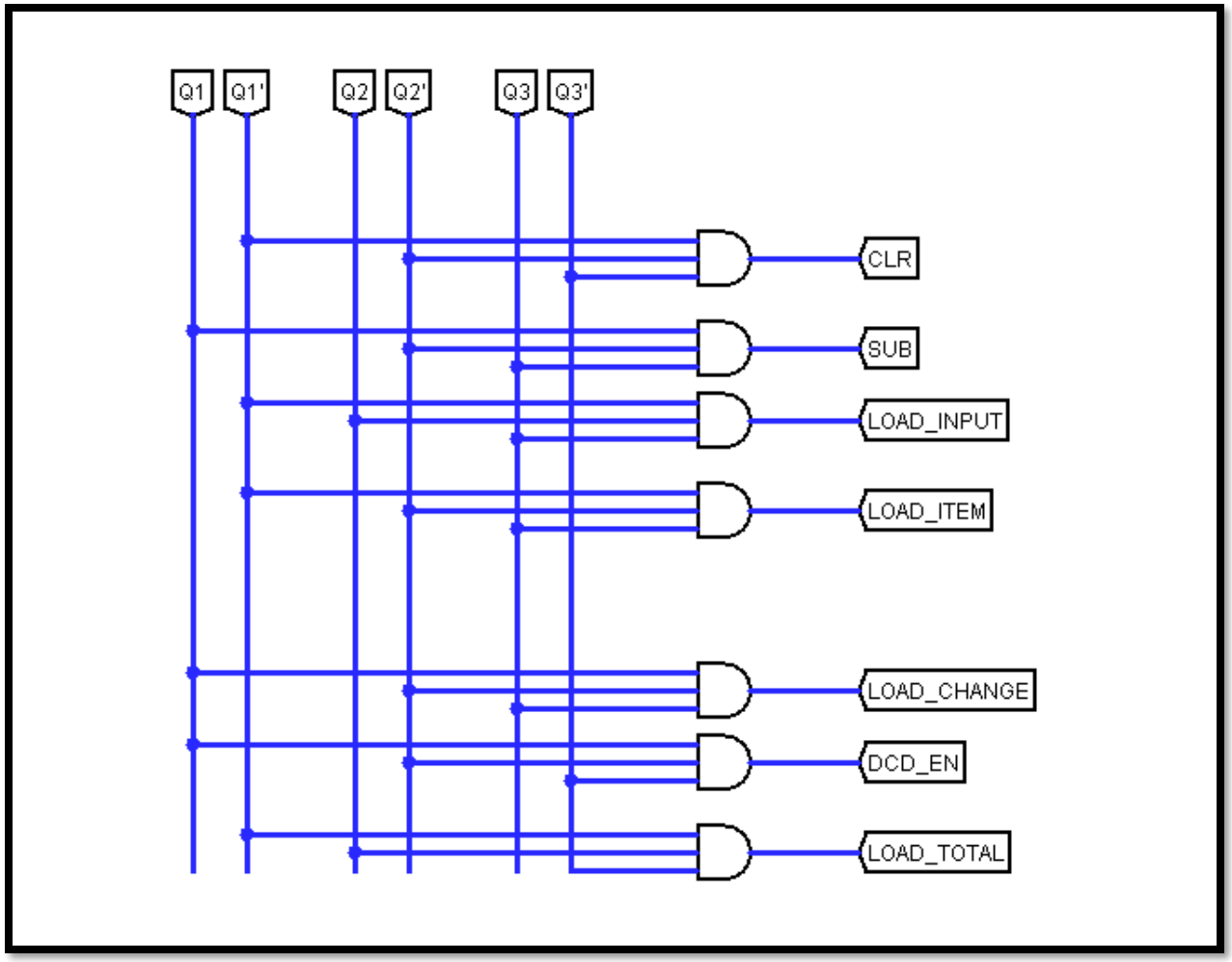

Figure 7: Outputs

Implementation

The circuit was implemented structurally with the lower components implemented either behaviorally or concurrently. The following components were built

- **Registers (2bit & 8bit):**

```
entity REG_2BIT is
    Port ( P_IN : in  STD_LOGIC_VECTOR (1 downto 0);
           CLK : in  STD_LOGIC;
           CLR : in  STD_LOGIC;
           EN : in  STD_LOGIC;
           P_OUT : out  STD_LOGIC_VECTOR (1 downto 0));
end REG_2BIT;

entity REG_8BIT is
    Port ( P_IN : in  STD_LOGIC_VECTOR (7 downto 0);
           CLK : in  STD_LOGIC;
           CLR : in  STD_LOGIC;
           EN : in  STD_LOGIC;
```

```
           P_OUT : out  STD_LOGIC_VECTOR (7 downto 0));
end REG_8BIT;
```

- **Gates**

```
entity ORtwo is
    Port ( OR_IN_A : in  STD_LOGIC;
           OR_IN_B : in  STD_LOGIC;
           OR_OUT : out  STD_LOGIC);
end ORtwo;

entity ANDtwo is
    Port ( AND_IN_A : in  STD_LOGIC;
           AND_IN_B : in  STD_LOGIC;
           AND_OUT : out  STD_LOGIC);
end ANDtwo;

entity ANDthree is
    Port ( AND_IN_A : in  STD_LOGIC;
           AND_IN_B : in  STD_LOGIC;
           AND_IN_C : in  STD_LOGIC;
           AND_OUT : out  STD_LOGIC);
end ANDthree;

entity INVone is
    Port ( INV_IN : in  STD_LOGIC;
           INV_OUT : out  STD_LOGIC);
end INVone;

entity INV_8BIT is
    Port ( INV_IN : in  STD_LOGIC_VECTOR (7 downto 0);
           INV_OUT : out  STD_LOGIC_VECTOR (7 downto 0));
end INV_8BIT;
```

- **MUXs**

```
entity MUX2to1_8BIT is
    Port ( D0 : in  STD_LOGIC_VECTOR (7 downto 0);
           D1 : in  STD_LOGIC_VECTOR (7 downto 0);
           S0 : in  STD_LOGIC;
           M_OUT : out  STD_LOGIC_VECTOR (7 downto 0));
end MUX2to1_8BIT;


entity MUX4to1_8BIT is
    Port ( D0 : in  STD_LOGIC_VECTOR (7 downto 0);
           D1 : in  STD_LOGIC_VECTOR (7 downto 0);
           D2 : in  STD_LOGIC_VECTOR (7 downto 0);
           D3 : in  STD_LOGIC_VECTOR (7 downto 0);
           Sel : in  STD_LOGIC_VECTOR (1 downto 0);
           M_OUT : out  STD_LOGIC_VECTOR (7 downto 0));
end MUX4to1_8BIT;
```

- **JK flip-flop**

```
entity JKFF is
    Port ( J : in  STD_LOGIC;
           K : in  STD_LOGIC;
           CLK : in  STD_LOGIC;
                       RST : in STD_LOGIC;
           Q : out  STD_LOGIC;
           QN : out  STD_LOGIC);
end JKFF;
```

- **2-to-4 Decoder**

```
entity DCD2to4 is
    Port ( DCD_IN : in  STD_LOGIC_VECTOR (1 downto 0);
           DCD_O1 : out  STD_LOGIC;
           DCD_O2 : out  STD_LOGIC;
           DCD_O3 : out  STD_LOGIC;
           DCD_O4 : out  STD_LOGIC;
           EN : in  STD_LOGIC);
end DCD2to4;
```

- **Comparator**

```
entity CMP_8BIT is
    Port ( CMP_IN_A : in  STD_LOGIC_VECTOR (7 downto 0);
           CMP_IN_B : in  STD_LOGIC_VECTOR (7 downto 0);
           CMP_GREATER : out  STD_LOGIC);
end CMP_8BIT;
```

- **Adder/Subtractor**

```
entity Adder_8BIT is
    Port ( ADD_IN_A : in  STD_LOGIC_VECTOR (7 downto 0);
           ADD_IN_B : in  STD_LOGIC_VECTOR (7 downto 0);
           C_IN : in  STD_LOGIC;
           ADD_OUT : out  STD_LOGIC_VECTOR (7 downto 0));
end Adder_8BIT;
```

Also, the datapath and the control unit were separated into two modules which were then implemented under the top-level entity “Vending_Machine”.

```
entity ASMController is
    Port ( CLK : in STD_LOGIC;
       RST : in STD_LOGIC;
       ITEM_SELECTED : in  STD_LOGIC;
        MONEY_IN : in  STD_LOGIC;
        GREATER_THAN : in  STD_LOGIC;
        CLR : out  STD_LOGIC;
        SUB : out  STD_LOGIC;
        LOAD_INPUT : out  STD_LOGIC;
        LOAD_ITEM : out  STD_LOGIC;
        LOAD_CHANGE : out  STD_LOGIC;
        LOAD_TOTAL : out  STD_LOGIC;
        DCD_EN : out  STD_LOGIC);
end ASMController;

entity Datapath is
```

```
    Port ( CLK : in  STD_LOGIC;
           CLR : in  STD_LOGIC;
           MONEY : in  STD_LOGIC_VECTOR (1 downto 0);
           ITEM : in  STD_LOGIC_VECTOR (1 downto 0);
           SUB : in  STD_LOGIC;
           LOAD_INPUT : in  STD_LOGIC;
           LOAD_ITEM : in  STD_LOGIC;
           LOAD_CHANGE : in  STD_LOGIC;
           DCD_EN : in  STD_LOGIC;
           LOAD_TOTAL : in  STD_LOGIC;
           GREATER_THAN : out  STD_LOGIC;
           CHANGE : out  STD_LOGIC_VECTOR (7 downto 0);
           ITEM1_EJECTED : out  STD_LOGIC;
           ITEM2_EJECTED : out  STD_LOGIC;
           ITEM3_EJECTED : out  STD_LOGIC;
           ITEM4_EJECTED : out  STD_LOGIC);
end Datapath;
```

These components were port-mapped and intermediary signals were used. Below are the signals and port-maps for the ASMController and the Datapath entities:

- **Datapath Entity**

```
signal IP : STD_LOGIC_VECTOR (1 DOWNTO 0); --INPUT SELECT
signal IT : STD_LOGIC_VECTOR (1 DOWNTO 0); --ITEM ITEM SELECT
signal MONEY_INPUT : STD_LOGIC_VECTOR (7 DOWNTO 0); --INPUT REGISTER
signal COST_INPUT : STD_LOGIC_VECTOR (7 DOWNTO 0); --ITEM REGISTER
signal COST_INV : STD_LOGIC_VECTOR (7 DOWNTO 0); --1C OF COST
signal MONEY_OR_COST : STD_LOGIC_VECTOR (7 DOWNTO 0); --OUTPUT OF 2X1 MUX
signal ADDER_TOTAL : STD_LOGIC_VECTOR (7 DOWNTO 0); --ADDER OUTPUT
signal TOTAL_OUT : STD_LOGIC_VECTOR (7 DOWNTO 0); --TOTAL REGISTER


begin

INPUT_REG: REG_2BIT port map (P_IN => MONEY, CLK => CLK, CLR => CLR, EN => LOAD_INPUT, P_OUT
=>IP);
ITEM_REG: REG_2BIT port map (P_IN => ITEM, CLK => CLK, CLR => CLR, EN => LOAD_ITEM, P_OUT =>
IT);

MUX_INPUT: MUX4to1_8BIT port map (D0 => "00001010", D1 => "00011001", D2 => "01100100",
       D3 => "00000000", Sel => IP, M_OUT => MONEY_INPUT);
MUX_ITEM: MUX4to1_8BIT port map (D0 => "00101101", D1 => "01001011", D2 => "01010000",
       D3 => "01100100", Sel => IT, M_OUT => COST_INPUT);

INV1C: INV_8BIT port map (INV_IN => COST_INPUT, INV_OUT => COST_INV);
MUX_MONEY_OR_COST: MUX2to1_8BIT port map (D0 => MONEY_INPUT, D1 => COST_INV, S0 => SUB, M_OUT
=> MONEY_OR_COST);

ALU: Adder_8BIT port map (ADD_IN_A => TOTAL_OUT, ADD_IN_B => MONEY_OR_COST, C_IN => SUB,
ADD_OUT => ADDER_TOTAL);

TOTAL_REG: REG_8BIT port map (P_IN => ADDER_TOTAL, CLK => CLK, CLR => CLR, EN => LOAD_TOTAL,
P_OUT => TOTAL_OUT);
```

```
COMPARE: CMP_8BIT port map (CMP_IN_A => TOTAL_OUT, CMP_IN_B => COST_INPUT, CMP_GREATER =>
GREATER_THAN);

SHOW_CHANGE: REG_8BIT port map (P_IN => ADDER_TOTAL, CLK => CLK, CLR => CLR, EN =>
LOAD_CHANGE, P_OUT => CHANGE);

DISPENSE: DCD2to4 port map (DCD_IN => IT, DCD_O1 => ITEM1_EJECTED, DCD_O2 => ITEM2_EJECTED,
       DCD_O3 => ITEM3_EJECTED, DCD_O4 => ITEM4_EJECTED, EN => DCD_EN);
```

- **ASMController Entity**

```
signal tmp1, tmp2, tmp3, tmp4, tmp5 : std_logic; --intermediate gates
signal J1, K1, J2, K2, J3, K3 : std_logic; --FF input
signal Q1, Q1N, Q2, Q2N, Q3, Q3N : std_logic; --FF output

begin

--FF1
AND1: ANDtwo port map (AND_IN_A => Q2, AND_IN_B => Q3N, AND_OUT => J1);
OR1: ORtwo port map (OR_IN_A => Q1, OR_IN_B => Q2, OR_OUT => tmp1);
AND2: ANDtwo port map (AND_IN_A => tmp1, AND_IN_B => GREATER_THAN, AND_OUT => tmp2);
OR2: ORtwo port map (OR_IN_A => Q3N, OR_IN_B => tmp2, OR_OUT => K1);
JKFF1: JKFF port map (J => J1, K => K1, CLK => CLK, RST => RST, Q => Q1, QN => Q1N);

--FF2
AND3: ANDthree port map (AND_IN_A => Q3, AND_IN_B => Q1N, AND_IN_C => ITEM_SELECTED, AND_OUT
=> J2);
INV1: INVone port map (INV_IN => GREATER_THAN, INV_OUT => tmp3);
AND4: ANDtwo port map (AND_IN_A => Q1, AND_IN_B => tmp3, AND_OUT => K2);
JKFF2: JKFF port map (J => J2, K => K2, CLK => CLK, RST => RST, Q => Q2, QN => Q2N);

--FF3
OR3: ORtwo port map (OR_IN_A => Q1N, OR_IN_B => Q2, OR_OUT => J3);
AND5: ANDtwo port map (AND_IN_A => Q2N, AND_IN_B => Q1, AND_OUT => tmp4);
AND6: ANDthree port map (AND_IN_A => Q1N, AND_IN_B => Q2, AND_IN_C => MONEY_IN, AND_OUT =>
tmp5);
OR4: ORtwo port map (OR_IN_A => tmp4, OR_IN_B => tmp5, OR_OUT => K3);
JKFF3: JKFF port map (J => J3, K => K3, CLK => CLK, RST => RST, Q => Q3, QN => Q3N);

--OUTPUT
AND7: ANDthree port map (AND_IN_A => Q1N, AND_IN_B => Q2N, AND_IN_C => Q3N, AND_OUT => CLR); -
-CLR
AND8: ANDthree port map (AND_IN_A => Q1, AND_IN_B => Q2N, AND_IN_C => Q3, AND_OUT => SUB); --
SUB
AND9: ANDthree port map (AND_IN_A => Q1N, AND_IN_B => Q2, AND_IN_C => Q3, AND_OUT =>
LOAD_INPUT); --LOAD_INPUT
ANDA: ANDthree port map (AND_IN_A => Q1N, AND_IN_B => Q2N, AND_IN_C => Q3, AND_OUT =>
LOAD_ITEM); --LOAD_ITEM
ANDB: ANDthree port map (AND_IN_A => Q1, AND_IN_B => Q2N, AND_IN_C => Q3, AND_OUT =>
LOAD_CHANGE); --LOAD_CHANGE
ANDC: ANDthree port map (AND_IN_A => Q1, AND_IN_B => Q2N, AND_IN_C => Q3N, AND_OUT => DCD_EN);
--DCD_EN
ANDD: ANDthree port map (AND_IN_A => Q1N, AND_IN_B => Q2, AND_IN_C => Q3N, AND_OUT =>
LOAD_TOTAL); --LOAD_TOTAL
```

Testing

Each component was tested and simulated to ensure optimal performance. The top-level design was also tested and simulated as if money was being inserted into the machine at different intervals, the change was displayed as well as which item was dispensed.

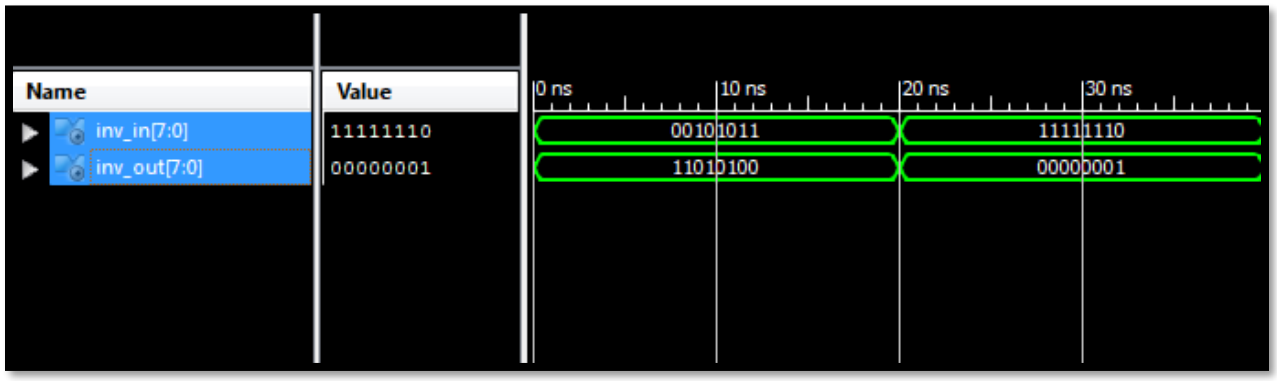

Figure 8: 8bit Inverter Used to Get 1C

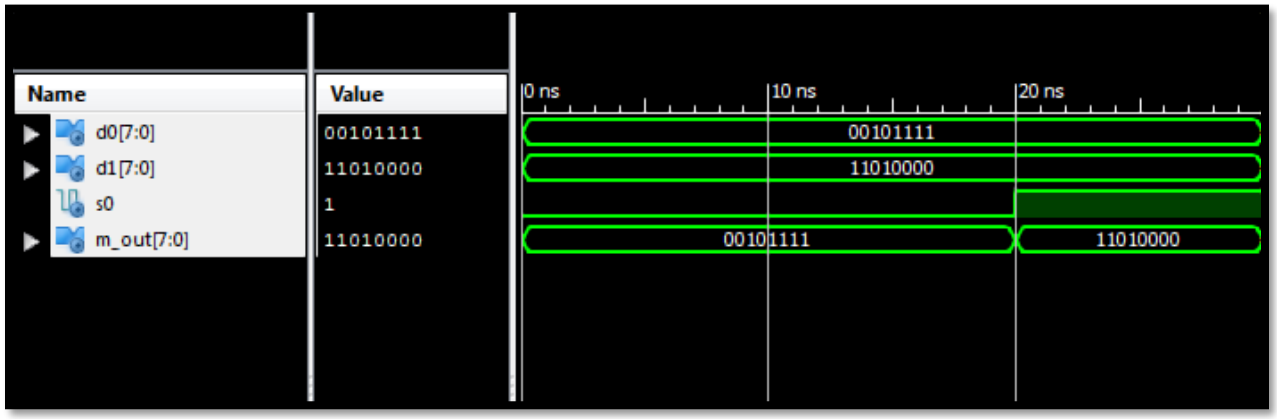

Figure 9: 8bit 2x1 MUX

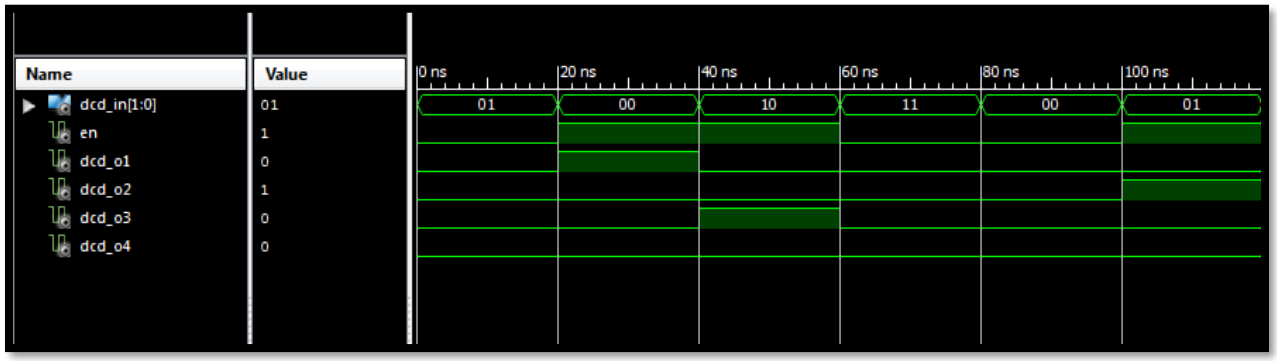

Figure 10: 8bit 2x4 DCD

Figure 11: 8bit Adder/Subtractor

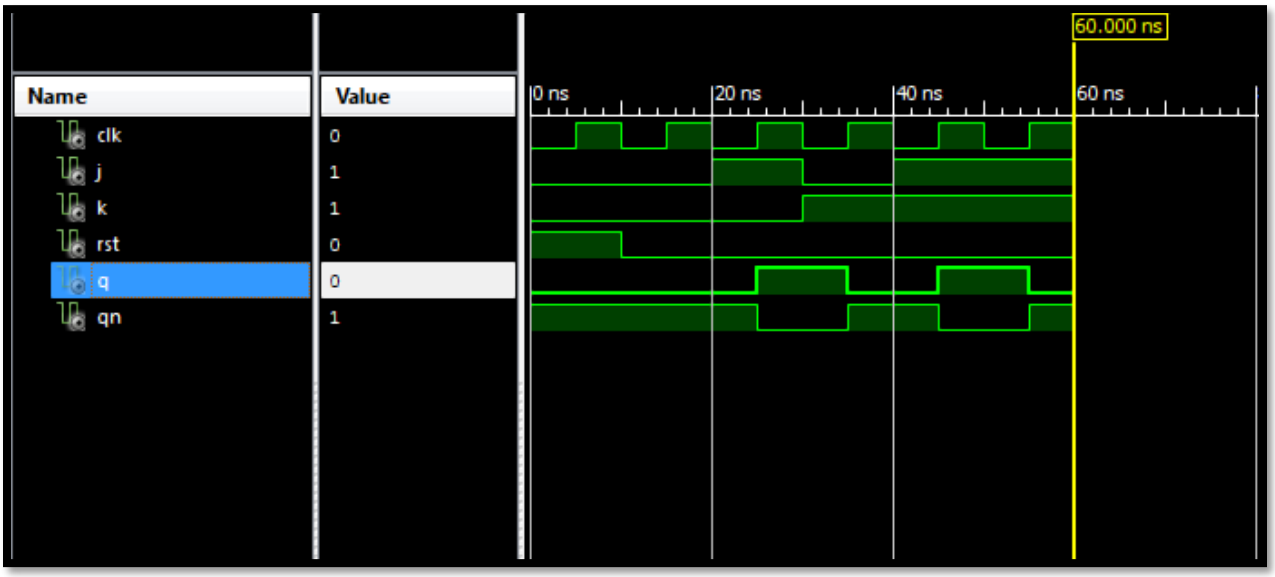

Figure 12: JK flip-flop with asynchronous reset

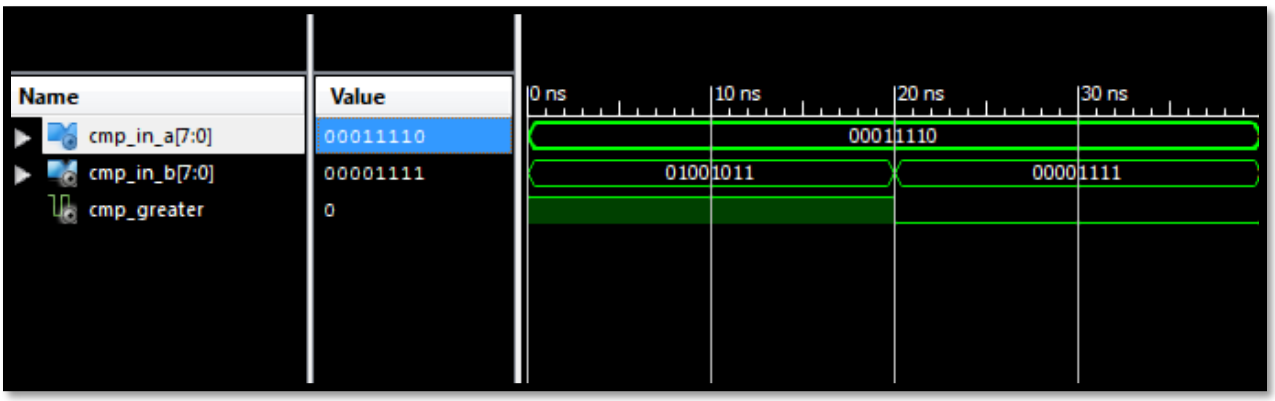

Figure 13: 8bit Comparator

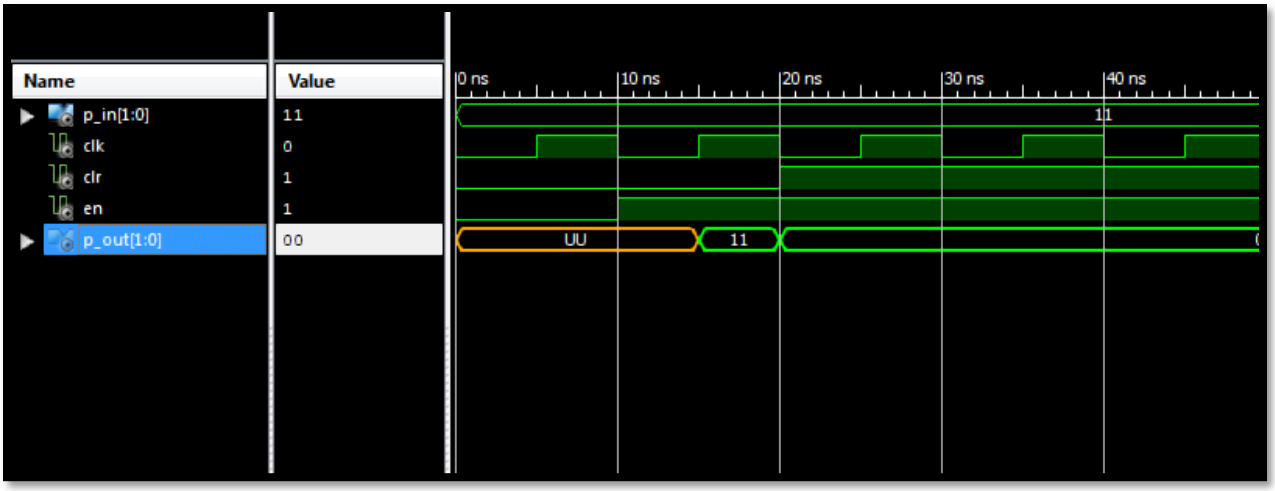

Figure 14: 2bit Parallel Load Register

Figure 15: ASMController with State Output

After synthesis, the following RTL schematic is shown:

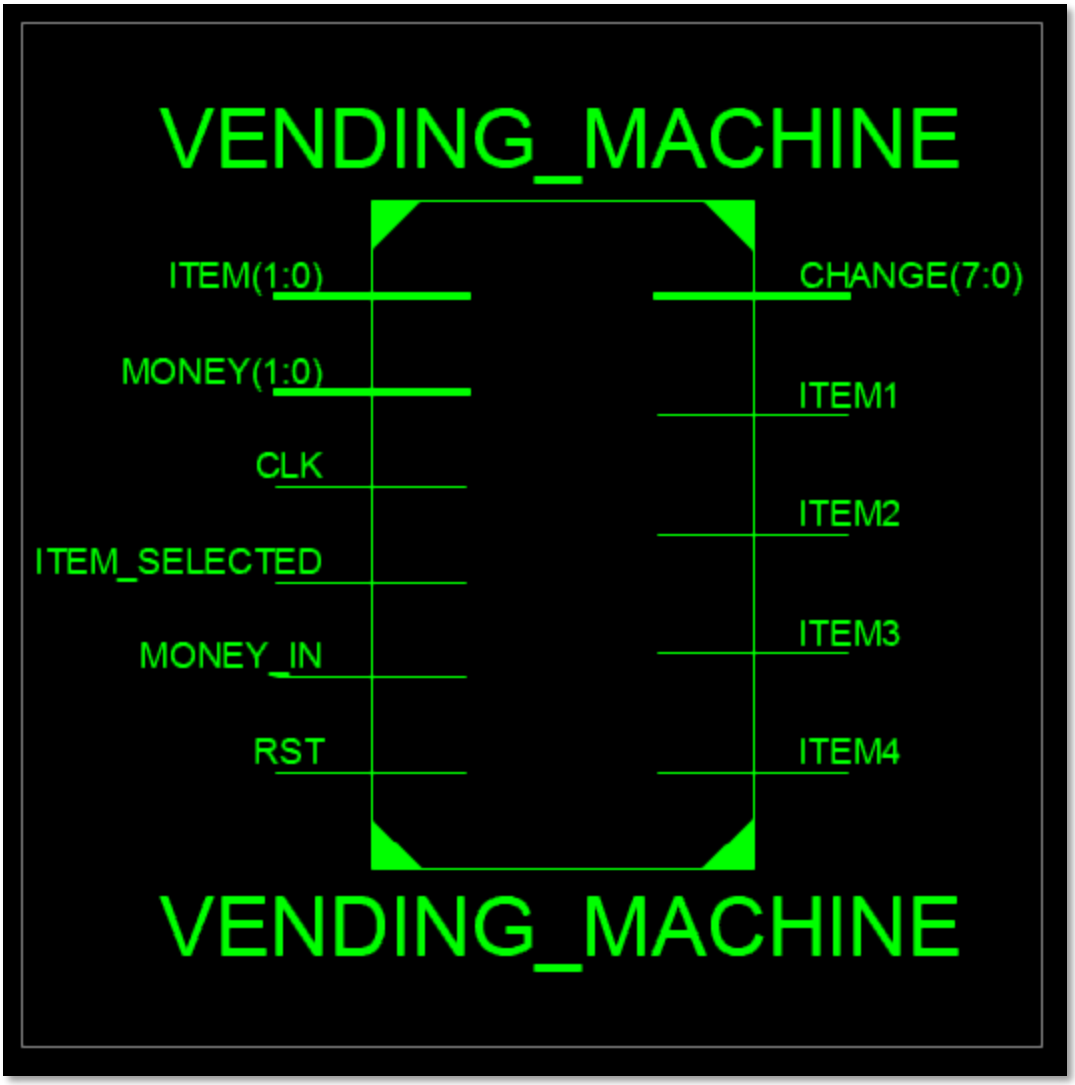

Figure 16: Top-Level Design

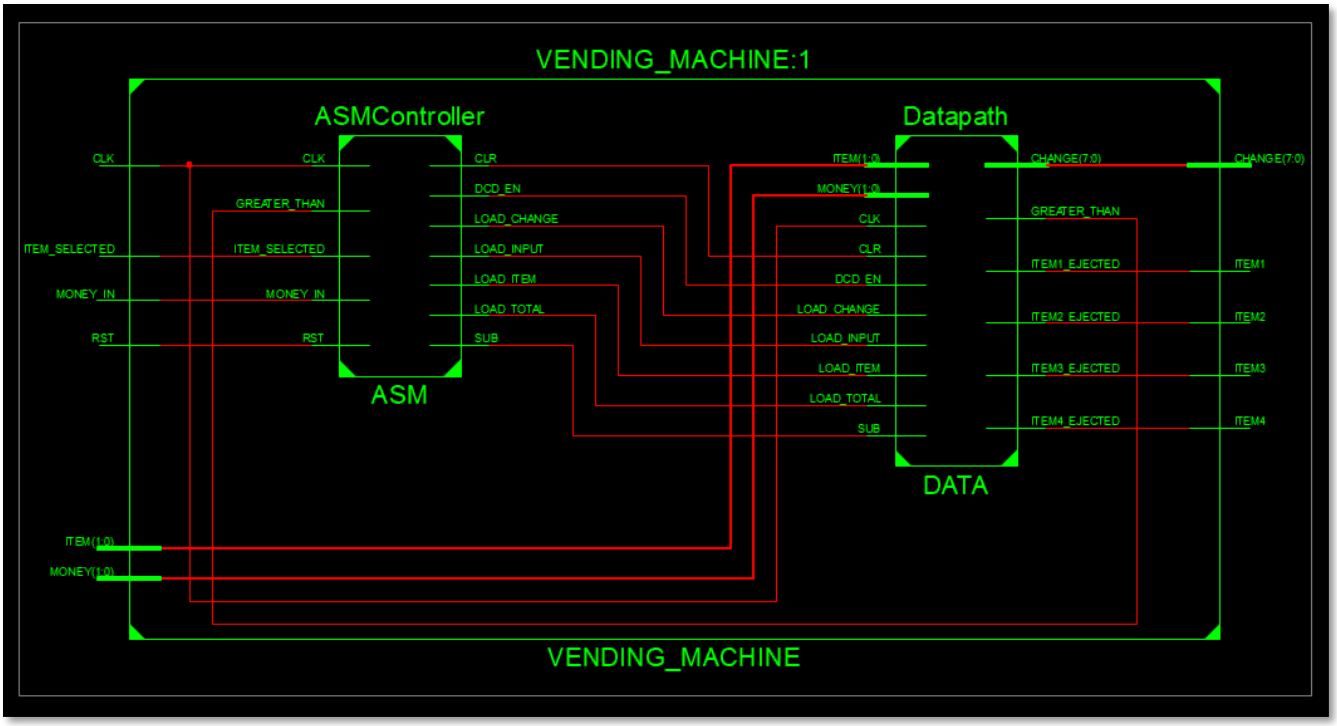

Figure 17: Second Layer Design with ASMController and Datapath blocks

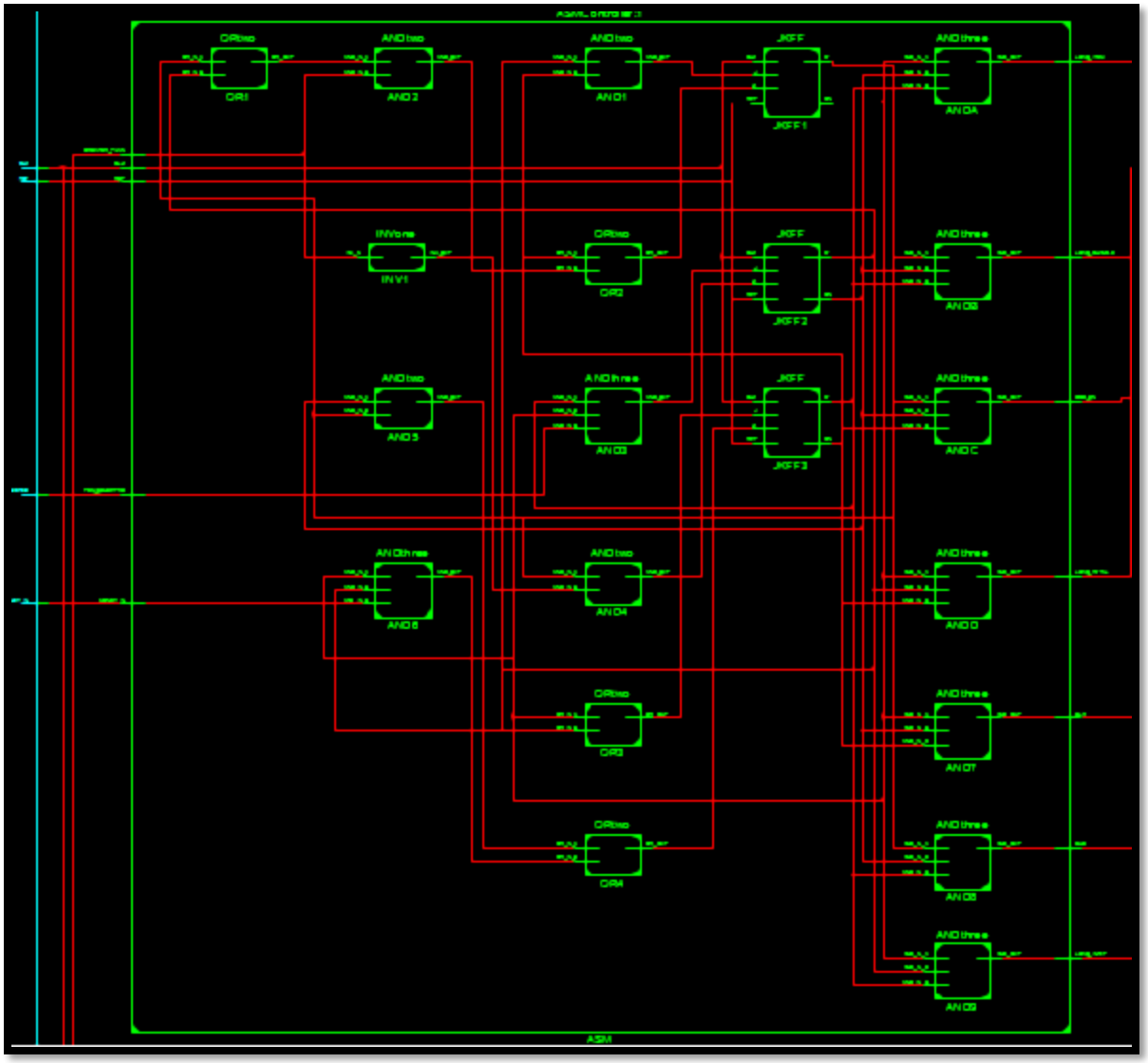

Figure 18: ASMController Circuit

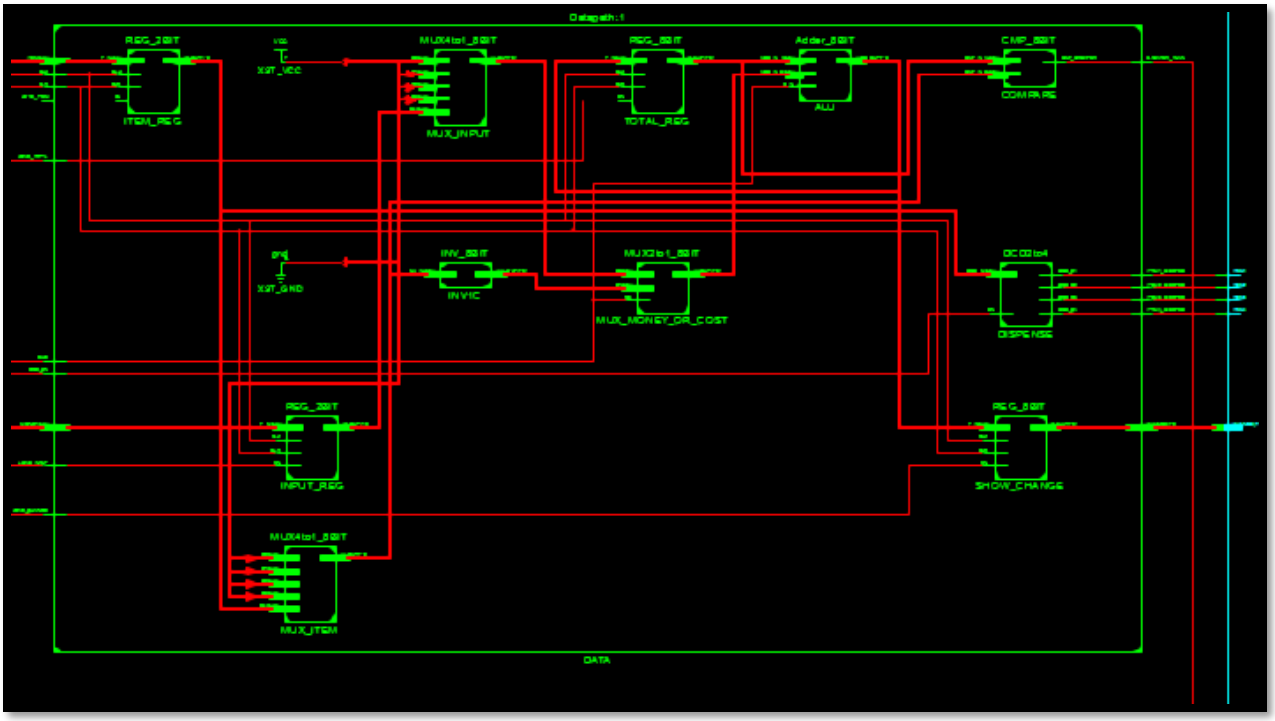

Figure 19: Datapath Circuit

The advantage of the structural design is that the synthesized circuit uses readily-available components such as IC gates, flip-flops, Adders, and more. This circuit can be easily assembled onto an IC chip as-is. When writing behavioral code for a complex circuit, it is not guaranteed to be synthesized into a structural design. It is important to know which VHDL code synthesize to a combinational circuit and which synthesizes to a sequential circuit.

The top-level design (the vending machine) was tested with the following stimulus:

```
RST <= '1';
wait for CLK_period;

RST <= '0';
MONEY <= "01"; --QUARTER
wait for CLK_period * 3;

MONEY_IN <= '1';

ITEM <= "00"; --.45
wait for CLK_period;

MONEY_IN <= '1';
ITEM_SELECTED <= '1';
wait for CLK_period * 5;

MONEY <= "10"; --DOLLAR
wait for CLK_period;

MONEY_IN <= '1';
wait;
```

And the following result:

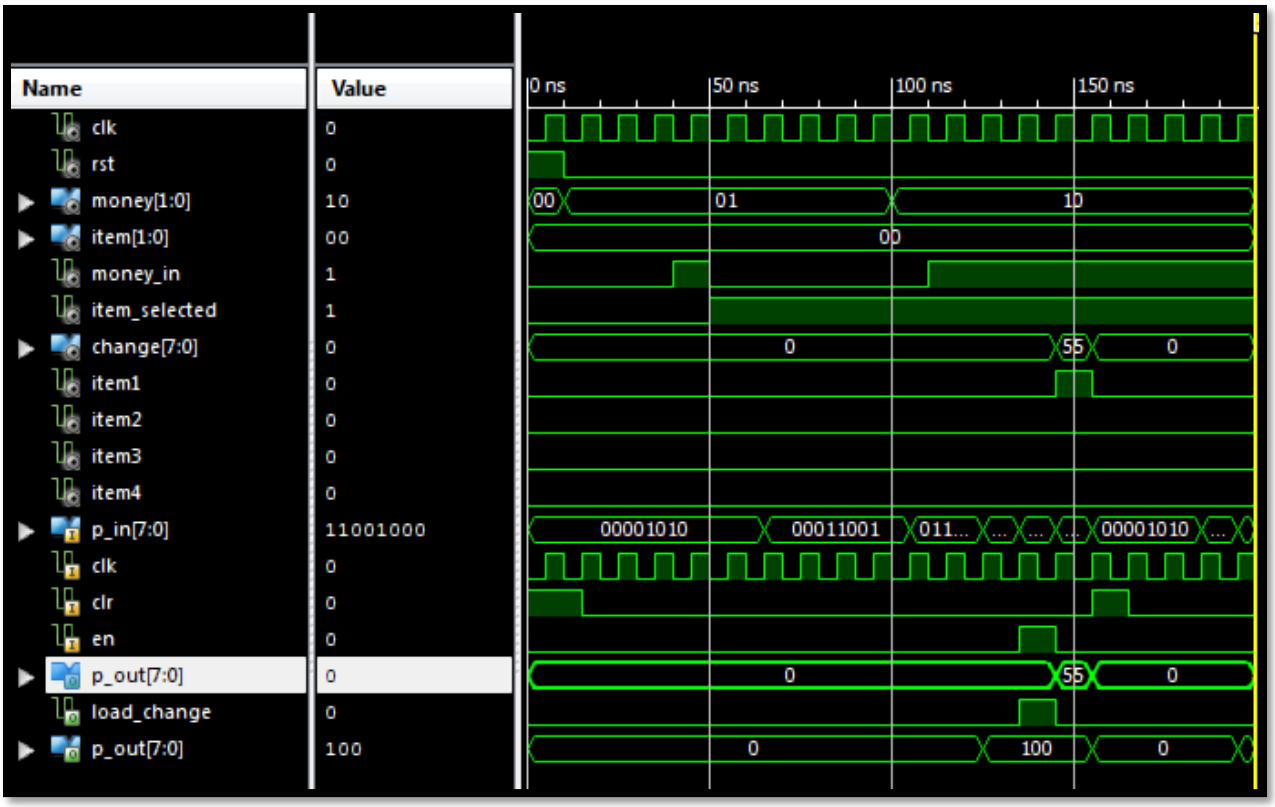

Figure 20: Vending Machine simulation; Item Cost = $0.45; Money Input = $1.00; Change = $0.55

Link to Simulation Recording

Link to Project ZIP file