ECE 2123L: Labs 7 and 8
Finite State Machine Design and Implementation

11/10/25

Prepared by:

Sophie Jones

Lab Section 2

Submitted 11/11/25

Note: I tried my best with formatting here; the code made it difficult, and I'm not super well-versed in Word. Apologies!

Pre-Lab

There was no pre-lab for lab 7 or lab 8.

Objectives and Design Requirements

The objective of this laboratory exercise was to design and implement a Finite State Machine that implements the logic of a vending machine which accepts coins and dispenses a soda once the total value reaches 25 cents. The machine accepts nickels, dimes, and quarters as valid inputs and uses LEDs and seven-segment displays (two: tens and ones digits) to indicate the current balance, whether it is dispensing a soda, and the change returned. The main goals were to first design the FSM on paper (Lab 7) and then implement and verify the working circuit on the DE-10 Lite FPGA board (Lab 8). The design used one-hot encoding, requiring ten states to represent the accumulated coin amounts from 0¢ to 45¢ in 5¢ increments. Each state corresponded to the total value currently stored in the machine, and transitions between states occurred whenever a new coin input was detected and the clock button was pressed.

For the design portion, we were required to create a state diagram and state transition table showing all possible coin input combinations and resulting states. We also had to derive input flip-flop equations and output logic equations for Dispense, ReturnNickel, ReturnDime, and ReturnTwoDimes signals. Lastly, for lab 7, we had to develop logic for converting the various state outputs into binary-encoded tens and ones digits that we could dispaly separately onto the seven-segment display.

For the implementation portion, we had to write the Verilog code for the FSM using our state diagram and derived equations from Lab 7. Along with that, we had to convert the code into block-diagram symbols in Quartus in order to build the necessary schematics. In the schematic we connected the FSM to the LED indicators, seven-segment display, coin-input switches, and reset/clock controls on the FPGA board.

Procedure

Instruments used:

- Lab machines
- Quartus Software Suite
- An FPGA board: DE-10 Standard Development and Education Kit by Terasic
- ModelSim Software
- Various cords used for connection including USB

First, we began lab 7 designing a finite state machine. We defined ten distinct states representing total coin values of 0¢, 5¢, 10¢, 15¢, 20¢, 25¢, 30¢, 35¢, 40¢, and 45¢. These are all of the possible addition combinations of adding just one coin of each value. We then drew the state-transition diagram showing all possible transitions for Nickel, Dime, and Quarter inputs, including transitions back to state 0 after vending. This resets it and helps visualize and allows for easier completion of future steps. We then had to assign one-hot encoding to each state, using ten flip-flops where exactly one bit is high for the current state.

Next, we constructed the state-transition table listing the current state, the coin input, next state, and outputs for the Dispense and Change signals. We then derived input equations for each D flip-flop (one per state) and output equations for the LEDs and seven-segment display. We then created state-to-binary mapping tables to convert one-hot state values to BCD digits for tens and ones, ensuring compatibility with the display driver (the second module).

Lastly, we had to implement our FSM in Quartus using Verilog. To do this, we opened the FSM.v template given in Brightspace. We then had to wire our FSM to the seven segment displays by creating a symbol file for the FSM module as well as our the BCD module and integrate it into the schematic. We then programmed the pins in the FPGA, and tested.

Results and Discussion

Shown below are the values on the FPGA board that we routed each input/output pin to.

- → **SW0–SW2** → Nickel, Dime, Quarter inputs
- → **KEY0** → Clock pulse
- → **KEY1** → Reset
- → **LEDR0–LEDR2** → Change outputs
- → **LEDR9** → Dispense output
- → **HEX0–HEX1** → Seven-segment display digits

The output shown in Figure I matched exactly what we expected from the FSM design. When we first pressed the reset (the light blue switch), the circuit went back to the starting state of 0¢, which makes sense because that's how we set it up in the code. After that, we started adding coins, for which each one has its own switch: the green switch for a nickel (5¢), the blue for a dime (10¢), and the red for a quarter (25¢).

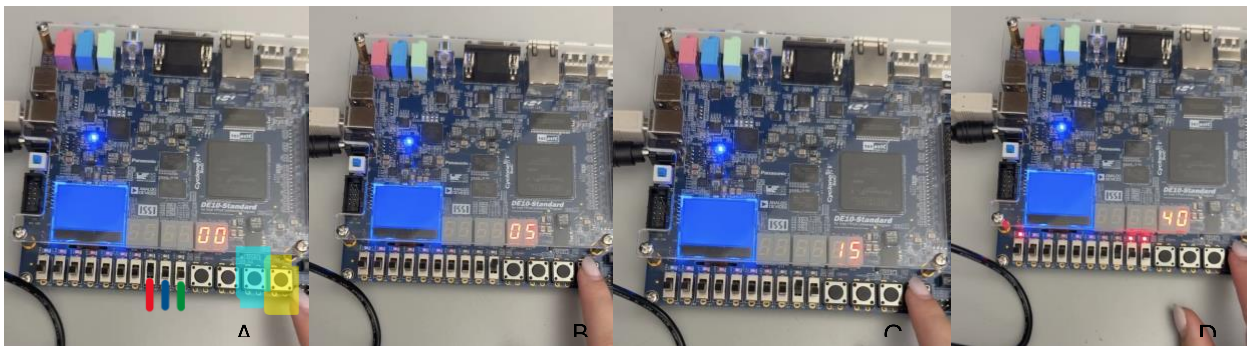


Figure I. *Example output of the FSM*

Every time we flipped one of those switches and then hit the clock button (Key 0), the current state updated on the next clock pulse. That's when the seven-segment display changed to show the new total. So, for example, when we pressed the green switch and then the clock, it moved from 0¢ to 5¢. Doing the same with the blue and red switches added the correct amounts each time, and the LEDs turned on for "Dispense" or "Return Change". The dispense LED was all the way on the left and lit up when the total hit 25¢ or more. The return change LEDs were above their respective switches.

Essentially, what we can see is the FSM cycling through its states exactly as we designed it to. The clock button controlled when the state actually changed, and the reset made sure we could always start over clean. Everything was in sync, and it proved that the Verilog design and wiring worked the way we planned. Shown below is the two files of Verilog code used in this lab.

Lab07FSM.sv

```
module Lab07FSM(
  clk,
  N,
  D,
  Q,
  rst,
  Dispense,
  ReturnNickle,
  ReturnDime,
  ReturnTwoDimes,
  S0,
  S1,
  S2,
  S3,
  S4,
  S5,
  S6,
  S7,
   S8,
   S9
);

input  wire clk;
input  wire N;
input  wire D;
input  wire Q;
input  wire rst;
output wire Dispense;
output wire ReturnNickle;
output wire ReturnDime;
output wire ReturnTwoDimes;

wire Db;
wire Nb;
wire NbDbQb;
wire Qb;
output reg S0;
output reg S1;
output reg S2;
output reg S3;
output reg S4;
output reg S5;
output reg S6;
output reg S7;
output reg S8;
output reg S9;
wire VCC;

assign Nb     = ~N;
assign Db     = ~D;
assign Qb     = ~Q;
assign NbDbQb  = Nb & Db & Qb;   // no coin selected
assign VCC     = 1'b1;

// S0: async active-LOW reset to 1; next = (vend states) -> S0, or hold in S0 if no coin
always @(posedge clk or negedge rst) begin
   if (!rst) begin
     S0 <= 1'b1;
   end else begin
     S0 <= (S5 | S6 | S7 | S8 | S9) | (S0 & NbDbQb); // def   REF
   end
end

// S1: next = S0·N
```

```
always @(posedge clk or negedge rst) begin
   if (!rst) begin
     S1 <= 1'b0;
   end else begin
     S1 <= (S0 & N) | (S1 & NbDbQb); // shows how the state is maintained for no inputs selected
   end
end

// S2: next = S0·D + S1·N
always @(posedge clk or negedge rst) begin
   if (!rst) begin
     S2 <= 1'b0;
   end else begin
     S2 <= (S0 & D) | (S1 & N) | (S2 & NbDbQb);
   end
end

// S3: next = S1·D + S2·N
always @(posedge clk or negedge rst) begin
   if (!rst) begin
     S3 <= 1'b0;
   end else begin
     S3 <= (S1 & D) | (S2 & N) | (S3 & NbDbQb);
   end
end

// S4: next = S2·D + S3·N
always @(posedge clk or negedge rst) begin
   if (!rst) begin
     S4 <= 1'b0;
   end else begin
     S4 <= (S2 & D) | (S3 & N) | (S4 & NbDbQb);
   end
end

// S5: next = S0·Q + S3·D + S4·N
always @(posedge clk or negedge rst) begin
   if (!rst) begin
     S5 <= 1'b0;
   end else begin
     S5 <= (S0 & Q) | (S3 & D) | (S4 & N);
   end
end

// S6: next = S1·Q + S4·D
always @(posedge clk or negedge rst) begin
```

```
  if (!rst) begin
    S6 <= 1'b0;
  end else begin
    S6 <= (S1 & Q) | (S4 & D);
  end
end

// S7: next = S2·Q
always @(posedge clk or negedge rst) begin
  if (!rst) begin
    S7 <= 1'b0;
  end else begin
    S7 <= (S2 & Q);
  end
end

// S8: next = S3·Q
always @(posedge clk or negedge rst) begin
  if (!rst) begin
    S8 <= 1'b0;
  end else begin
    S8 <= (S3 & Q);
  end
end

// S9: next = S4·Q
always @(posedge clk or negedge rst) begin
  if (!rst) begin
    S9 <= 1'b0;
  end else begin
    S9 <= (S4 & Q);
  end
end

// Moore outputs (>=25¢)
assign ReturnNickle   = S6 | S8;              // 30¢ or 40¢ → 5¢ back
assign ReturnDime     = S7 | S8;              // 35¢ or 40¢ → 10¢ back
assign Dispense      = S5 | S6 | S7 | S8 | S9;     // vend in any >=25¢ state
assign ReturnTwoDimes = S9;                   // 45¢ → 20¢ back

endmodule
```

END OF FILE 1

Shown below is the second file of code.

State_to_bcd.sv

```
module state_to_bcd(
  input  wire S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,
  output wire [3:0] tens,
  output wire [3:0] ones
);
  // Ones is 0 or 5: 0=0000, 5=0101
  assign ones[0] = S1 | S3 | S5 | S7 | S9;
  assign ones[1] = 1'b0;
  assign ones[2] = S1 | S3 | S5 | S7 | S9;
  assign ones[3] = 1'b0;

  // Tens across S0..S9: 0,0,1,1,2,2,3,3,4,4
  assign tens[0] = S2 | S3 | S6 | S7;
  assign tens[1] = S4 | S5 | S6 | S7;
  assign tens[2] = S8 | S9;
  assign tens[3] = 1'b0;
endmodule
```

END OF FILE 2.

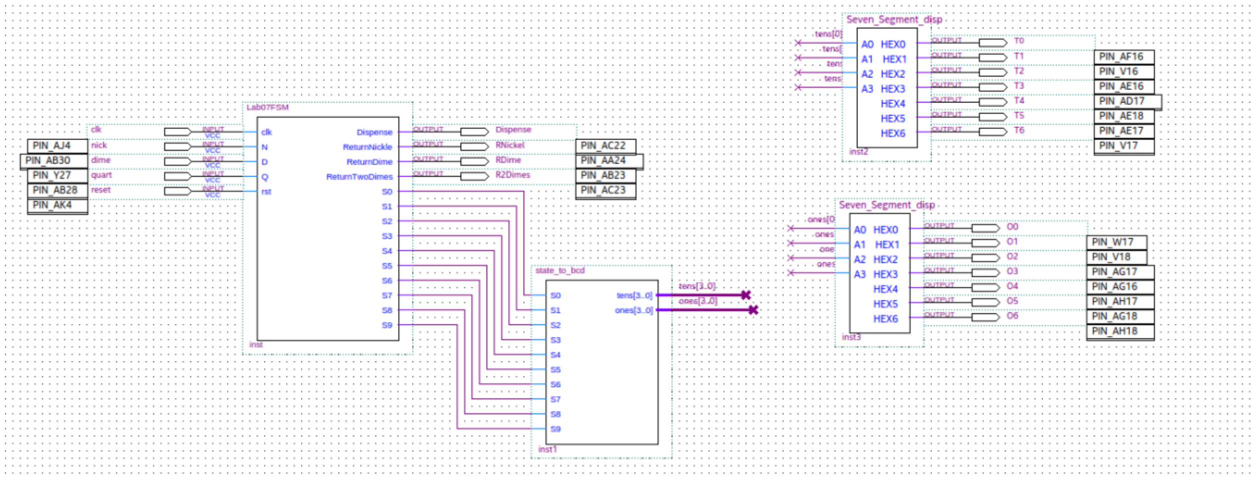


Figure II. *Schematic using the two .sv files as blocks and routing them to the seven-segment displays.*

We also used one-hot encoding, which made the state transitions clear and reliable. Since each state has its own flip-flop, only one is active at a time, which reduces confusion and makes it easier to see which state the machine is currently in. It also helped make debugging simpler because the LEDs clearly showed which state was active, and we didn't have to worry about overlapping states or weird binary transitions.

Post-Lab Results and Answers

1. Drawings & Design:

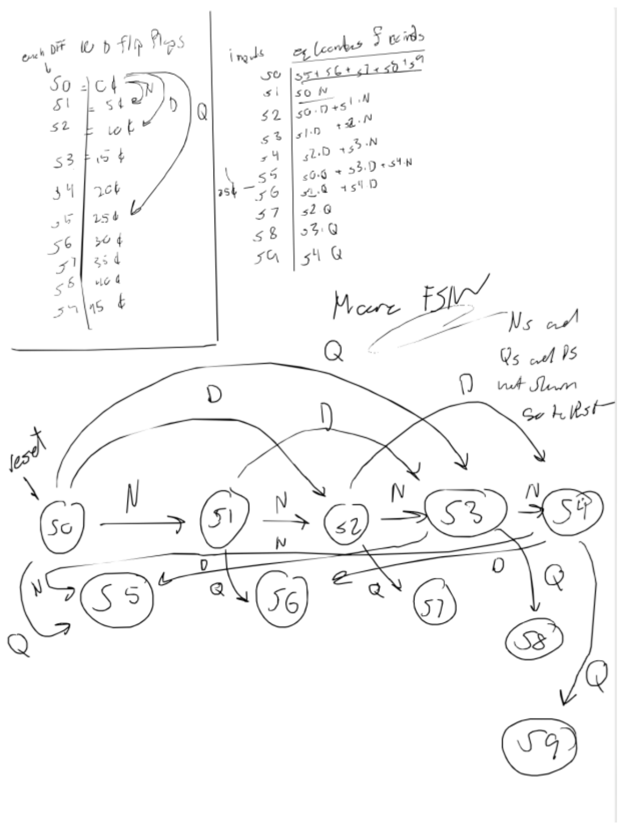
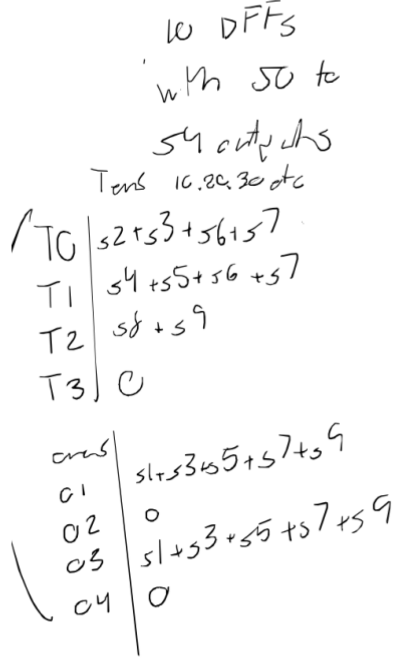


Figure III. *FSM design, transition table, flip flop logic.*



10 DFFs
with S0 to
S9 outputs

Tens 10.20.30 etc

T0	s2 + s3 + s6 + s7
T1	s4 + s5 + s6 + s7
T2	s8 + s9
T3	0

ones	s1 + s3 + s5 + s7 + s9
O1	
O2	0
O3	s1 + s3 + s5 + s7 + s9
O4	0

Figure IV. *Figure III continued, BCD.*

2. How many flip-flops would be required if the states were encoded using binary encoding rather than one-hot encoding?

With 10 states, we can use the equation $\textit{number of bits required} = \log_2(N)$. Using this we can do $\log_2(10) \ = \ 4$. Thus, we would need 4 flip flops.

3. In what ways would the design be simpler with binary encoding of the states? In what ways would the design be more difficult?

It could be simpler with binary encoding because we are using fewer flip-flops. Also, we could use more bits (up to 16) and still only use 4 flip flops. It would be a harder design in that the Boolean equations for each state would be larger and more complicated. Debugging would also be more difficult. With one-hot, an LED per state tells you exactly where you are. With binary, you need a decoder/7-segment block to interpret the state.

Conclusions

This lab emphasized the importance of FSMs in real-world systems. We were able to have hands-on experience combining combinational logic with sequential logic. It tied together the content we've been learning about in class and how these systems have memory—essentially how they remember what happened before and respond differently depending on the current state. This lab also made it clearer why certain design choices like one-hot encoding matter. Even though it uses more flip-flops, it made debugging and understanding the circuit easier because each state was represented directly. Overall, this lab showed how FSMs are a key concept in digital systems and how combining logic design with timing and memory lets us build hardware that behaves predictably in real-world scenarios.

References

Amat, Ashwaq. *ECE 2123 – Digital Systems Lecture Slides.* Vanderbilt University, 2025.

Amat, Ashwaq. *ECE 2123L – Lab 07: Finite State Machine (Design)*, Digital Systems Laboratory, 2025.

Amat, Ashwaq. *ECE 2123L – Lab 08: Finite State Machine (Implementation)*, Digital Systems Laboratory, 2025.

Kaznacheev, Ilya. "Practical Use of Finite-State Machines." *DEV Community*, 29 Mar. 2021, https://dev.to/ilyakaznacheev/practical-use-of-finite-state-machines-3gck.

"One Hot Encoding in Machine Learning." *GeeksforGeeks*, 11 July 2025, https://www.geeksforgeeks.org/machine-learning/ml-one-hot-encoding/.