FSM System Design

Lab Report 2 for ECE327

Digital Systems Design

Submitted by

Julian Coy

Undergraduate of Electrical & Computer Engineering

Clemson University

March 12, 2014

Abstract

Finite state machines (FSMs) are the basic building blocks of complex computing. They allow for logical implementation of algorithms, which is critical to the engineering of computing systems. In this lab, a finite state machine is constructed to flag "stop codons" in a sample of DNA code. The implementation of this finite machine is done though VHDL using behavioral modeling techniques. Testbench results are provided later in this report to show a working simulation of the FSM when connected to other logical components. The other logical components built for this lab are a Parallel-to-Serial Output (PISO) register and a counter register. The inputs to the FSM design will be threefold. There will be one array of 18 switches which are used to simulate the DNA input. There will also be a "reset" button and a "load" button. The reset button will asynchronously reset the counter block, while the load button will be used to push new data into the PISO register. This design showcases the power of finite state machine logic. It is critical for engineers to grasp the power of logical computation and to see the scalability of such power. By combining simple FSM logical blocks, one can create a vast and complicated algorithm with ease.

Note: All clocks generated for simulation were built using Morten Zilmers clock gen package [1].

INTRODUCTION

The purpose of this lab is to sequence DNA bases consisting of "T", "A", "C", and "G" and look for stop codons. Stop codons are series of three bases that trigger the end of a DNA sequence. The stop codons we look for in this lab are "TAA", "TGA", and "TAG". In order to differentiate the bases, we use a binary representation for the bases (Figure 1.1). The bases are then fed into the FSM, the number of stop codons are counted, and then the count is displayed on the Altera board.

Nitrogen base	Binary code
A	00
T	01
C	10
G	11

Figure 1.1: DNA Base Representation

Lab two consists of three major logical components (the FSM logic, the counter logic, and the PISO logic) and one minor component (the LED logic). The counter and PISO components are standalone and can be simulated through ModelSim for testing or used for implementation with other systems with minimal modifications. The FSM logic is designed to model state transitions of the incoming DNA data from the PISO. And the minor LED logic simply displays a number of LEDs correspondent to the output of the counter logic.

1.1 FSM SYSTEM DESIGN

1.1.1 PISO LOGIC DESIGN

The PISO design for this lab was built specifically to handle 18 bits of input data at a time. This was a design choice made for the layout of the Altera board, as it only allowed for 18 bits of modifiable input at a time. The PISO logic parses the 18 bits of data by 2 bit intervals. It then outputs the first two bits of data on the rising edge of the clock. This output is connected to the FSM input. The PISO then shifts the data left by two bits and adds a "01" to the right end. The "01" code was chosen for the shift packing to make sure that when the data is fully read, no extra stop codons are accidentally detected. Since the code for the "T" base is "01" and no stop codons end with "T" or consist of all "T" bases, this eliminates that possibility.

1.1.2 Testing of the PISO

To test the PISO, a testbench was created that would send in a string of data to the PISO and monitor the output on the rising edge of the following clock cycles (Figure 1.2). Notice that the output data is not started until the rising edge of the clock cycle following the release of the load button (low active). In the example shown, the data input is "01001100010101001" which corresponds to "TAGATTTAT". The output is exactly what we expect. Not shown in the figure is the loading of "T" ("01") values on the preceeding end because it shows no change to the final value in the simulation.

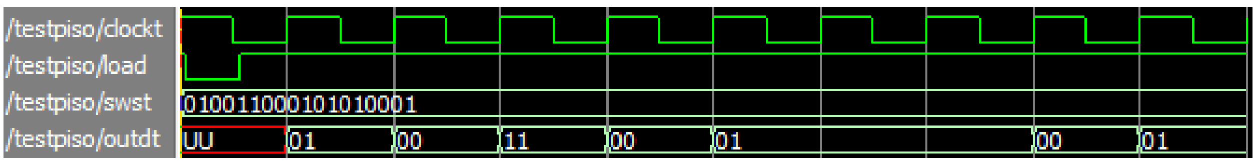

Figure 1.2: PISO Simulation Results

1.1.3 Counter Logic Design

The counter is modeled behaviorally and simply counts the number of stop flags from the FSM it encounters. The maximum count used for this project is three, but can be modified to much larger ranges with only a few lines of code. The counter outputs its value on an integer signal that is sent to the LED logic.

1.1.4 Testing of the Counter

The counter was tested by sending stop flags to the input and periodically sending a reset signal (Figure 1.3). The counter performed without issue even when using large integer values. The maximum count size tested was 255, but since the maximum number of stop codons possible from the board were three, the counter integer limit was set to 3.

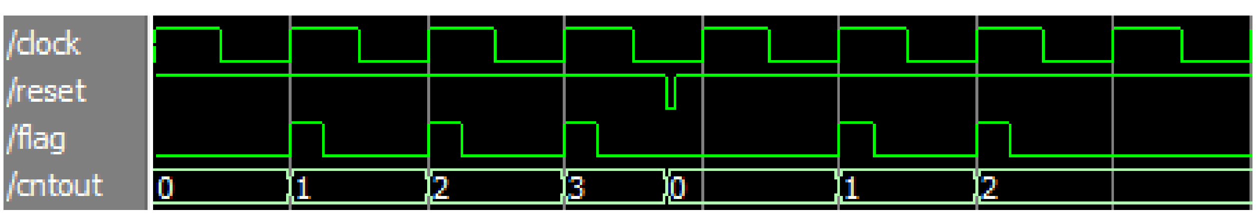

Figure 1.3: Counter Simulation Results [1]

1.1.5 FSM LOGIC DESIGN

The FSM was designed with 4 logical states: A, B, C, and D. The first state, A, was the reset state. This state represents no stop codons and no detection of the beginning of any codons (which always start with a "T" base). Once a "T" is detected the state will shift to B. From B there are four options which are shown in Figure 1.4. In the figure, the input to the FSM "w" determines the next state. This FSM uses a Mealy design principle. This allows for faster outputs and less state management. In fact, to implement this in a Moore model, at least 5 states would need to be used.

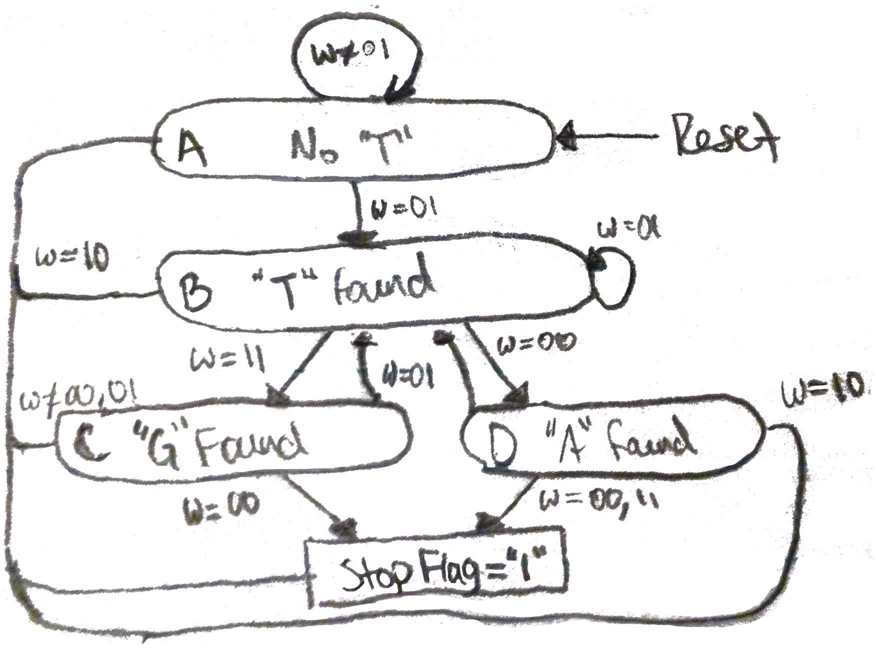

Figure 1.4: FSM State Diagram

Once the FSM is in state C or D it looks for specific input signals. If the signals match a certain condition set (w=00 for C or w=00,11 for D) then the stop flag is triggered and the state resets to A. The only danger to this model is when the asynchronous reset is called between the C and D input conditional and the stop flag assignment. If this were to occur the counter could be prematurely incremented on the next cycle. However, this is incredibly rare as for this to occur requires that the reset must toggle within the fraction of between a comparison and signal assignment.

1.1.6 Testing of the FSM

The FSM was tested by

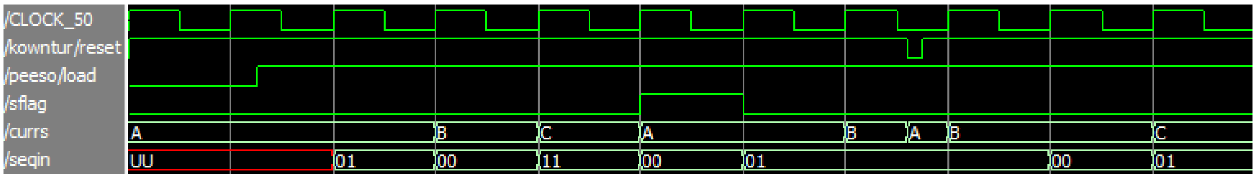

Figure 1.5: FSM Simulation Results

1.2 Conclusions

This lab is a testament to the modularity, reliability, and scope of FPGA programming with VHDL. Simple components or systems can be designed and connected to form incredibly complex systems that can perform a wide range of actions. Originally, the subsystems in the lab seemed unconnected to one another. However, by the end of the lab it was clear that the systems were building upon each other. This lab also clearly demonstrated the value of simulations in attempting to analyze complex FPGA systems.

WORKS CITED

[1] M. Zilmer, “Clock Generator.” Internet: http://stackoverflow.com/questions/17904514/vhdl-how-should-i-create-a-clock-in-a-testbench, 2013.

Appendix A: 5-Seven Segment Decoder

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
USE work.all;

ENTITY LAB1d IS

  PORT (
    SW            : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
    HEX0,HEX1,HEX2,HEX3,HEX4 : OUT STD_LOGIC_VECTOR(6 DOWNTO 0)
  );

END ENTITY LAB1d;

ARCHITECTURE LAB1d_arch OF LAB1d IS

  COMPONENT seven_seg_parse
    PORT (
      DATA  : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
      OUTPUT : OUT STD_LOGIC_VECTOR(6 DOWNTO 0)
    );
  END COMPONENT seven_seg_parse;

  SIGNAL S0,S1,S2,S3,S4 : STD_LOGIC_VECTOR(6 DOWNTO 0);

BEGIN
    behav : PROCESS (S0,S1,S2,S3,S4)
```

```
BEGIN
  CASE SW(17 DOWNTO 15) IS
    WHEN "000" =>
      HEX0 <= S0;
      HEX1 <= S1;
      HEX2 <= S2;
      HEX3 <= S3;
      HEX4 <= S4;
    WHEN "001" =>
      HEX0 <= S1;
      HEX1 <= S2;
      HEX2 <= S3;
      HEX3 <= S4;
      HEX4 <= S0;
    WHEN "010" =>
      HEX0 <= S2;
      HEX1 <= S3;
      HEX2 <= S4;
      HEX3 <= S0;
      HEX4 <= S1;
    WHEN "011" =>
      HEX0 <= S3;
      HEX1 <= S4;
      HEX2 <= S0;
      HEX3 <= S1;
      HEX4 <= S2;
    WHEN "100" =>
      HEX0 <= S4;
      HEX1 <= S0;
      HEX2 <= S1;
      HEX3 <= S2;
      HEX4 <= S3;
    WHEN others =>
      HEX0 <= S0;
      HEX1 <= S1;
      HEX2 <= S2;
```

```
        HEX3 <= S3;
        HEX4 <= S4;
      END CASE;
    END PROCESS behav;
    seven_seg_0 : seven_seg_parse PORT MAP(SW(2 DOWNTO 0),S0);
    seven_seg_1 : seven_seg_parse PORT MAP(SW(5 DOWNTO 3),S1);
    seven_seg_2 : seven_seg_parse PORT MAP(SW(8 DOWNTO 6),S2);
    seven_seg_3 : seven_seg_parse PORT MAP(SW(11 DOWNTO 9),S3);
    seven_seg_4 : seven_seg_parse PORT MAP(SW(14 DOWNTO 12),S4);
END ARCHITECTURE LAB1d_arch;

-------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
USE work.all;

ENTITY seven_seg_parse IS

  PORT (
    DATA  : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
    OUTPUT : OUT STD_LOGIC_VECTOR(6 DOWNTO 0)
  );

END ENTITY seven_seg_parse;

ARCHITECTURE basic OF seven_seg_parse IS

BEGIN
  behav : PROCESS (DATA)
  BEGIN
    CASE DATA IS
      WHEN "000" => OUTPUT <= "0000111";
      WHEN "001" => OUTPUT <= "1111001";
      WHEN "010" => OUTPUT <= "0010000";
```

```
      WHEN "011" => OUTPUT <= "0000110";
      WHEN "100" => OUTPUT <= "1001100";
      WHEN others => OUTPUT <= "0000000";
    END CASE;
  END PROCESS behav;
END ARCHITECTURE basic;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
USE work.all;

ENTITY mux21 IS

  PORT (
    SEL,X,Y : IN STD_LOGIC;
    M   : OUT STD_LOGIC);
END ENTITY mux21;

ARCHITECTURE basic OF mux21 IS
BEGIN
  mux21_behavior : PROCESS (SEL,X,Y)
  BEGIN
      if (SEL = '0') then M <= X; else M <= Y; END if;
  END PROCESS mux21_behavior;
END ARCHITECTURE basic;
```

Appendix B: Test Bench (Part 4)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY TB1d IS
END TB1d;

ARCHITECTURE TB1d_arch OF TB1d IS

  -- test signals going into and out of the mux
  SIGNAL TESTSO1 : STD_LOGIC_VECTOR(6 DOWNTO 0);
  SIGNAL TESTSO2 : STD_LOGIC_VECTOR(6 DOWNTO 0);
  SIGNAL TESTSO3 : STD_LOGIC_VECTOR(6 DOWNTO 0);
  SIGNAL TESTSO4 : STD_LOGIC_VECTOR(6 DOWNTO 0);
  SIGNAL TESTSO5 : STD_LOGIC_VECTOR(6 DOWNTO 0);
  SIGNAL TESTSW : STD_LOGIC_VECTOR(17 DOWNTO 0);

  COMPONENT LAB1d
    PORT (
      SW   : IN  STD_LOGIC_VECTOR(17 DOWNTO 0);
      HEX0 : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
      HEX1 : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
      HEX2 : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
      HEX3 : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
      HEX4 : OUT STD_LOGIC_VECTOR(6 DOWNTO 0)
    );
  END COMPONENT;

  -- map the signals

  BEGIN
```

```
multimux : LAB1d
PORT MAP (
  SW   => TESTSW,
  HEX0 => TESTSO1,
  HEX1 => TESTSO2,
  HEX2 => TESTSO3,
  HEX3 => TESTSO4,
  HEX4 => TESTSO5
);

-- start the test

test : PROCESS
BEGIN

-- set the initial word to be TIGER and initial position
TESTSW(14 DOWNTO 0) <= "100011010001000";
TESTSW(17 DOWNTO 15) <= "000"; WAIT FOR 4 ns;

-- cycle around message
TESTSW(17 DOWNTO 15) <= "001"; WAIT FOR 2 ns;
TESTSW(17 DOWNTO 15) <= "010"; WAIT FOR 2 ns;
TESTSW(17 DOWNTO 15) <= "011"; WAIT FOR 2 ns;
TESTSW(17 DOWNTO 15) <= "100"; WAIT FOR 2 ns;

-- change message
TESTSW(17 DOWNTO 15) <= "000";
TESTSW(14 DOWNTO 0) <= "100001010011000"; WAIT FOR 4 ns;

-- cycle randomly message
TESTSW(17 DOWNTO 15) <= "011"; WAIT FOR 2 ns;
TESTSW(17 DOWNTO 15) <= "010"; WAIT FOR 2 ns;
TESTSW(17 DOWNTO 15) <= "001"; WAIT FOR 2 ns;
TESTSW(17 DOWNTO 15) <= "100"; WAIT FOR 2 ns;
```

```
  WAIT;
  END PROCESS test;

END TB1d_arch;
```

APPENDIX C: WAVEFORM (PART 4)