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 FSM 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 are threefold. There is one array of 18 switches which is used to simulate the DNA input. There is also be a "reset" button and a "load" button. The reset button asynchronously resets the counter block, while the load button pushes 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 2 consists of three major logical components: the FSM logic, the counter logic, and the PISO logic. There is also 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. The minor LED logic simply displays a number of LEDs correspondent to the output of the counter logic.

1.1 LAB 2 SYSTEM DESIGN

1.1.1 PISO LOGIC DESIGN

The PISO component 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.

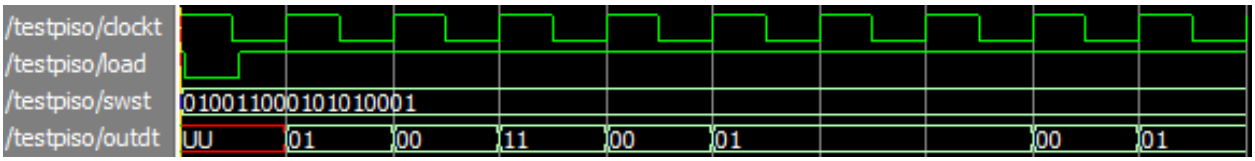

Figure 1.2: PISO Simulation Results

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.

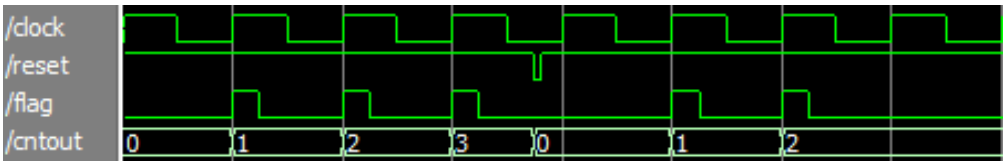

Figure 1.3: Counter Simulation Results [1]

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.

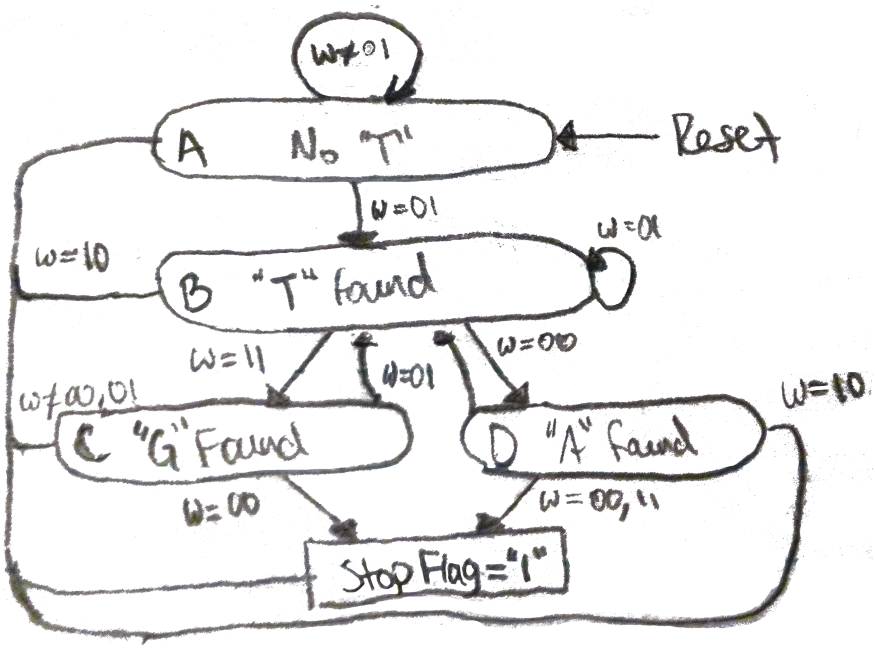

Figure 1.4: FSM State Diagram

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. This can be controlled by pulsing the reset signal for only the duration required. If using a human controller, like is done in this lab, the risk is unavoidable with a Mealy design.

1.1.6 TESTING OF THE FSM

The FSM was tested by inputing the PISO data into the FSM and monitoring the state changes and flag signals that come out. In Figure 1.5 you can see that the state remains in A when the input data is not "01" or a "T". Once a "T" is found the state transitions to B. The figure shows the progression for detection of one stop codon with some input data.

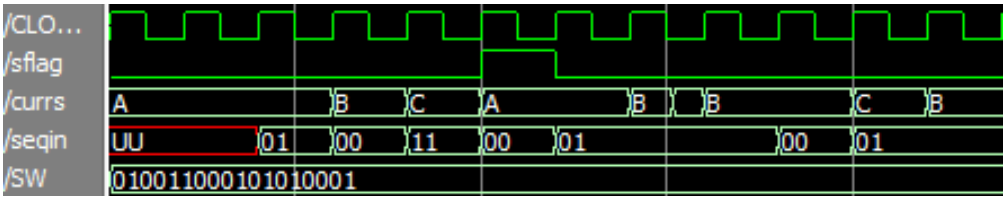

Figure 1.5: FSM Simulation Results

1.2 CONCLUSIONS

There are minor changes that could be done to the lab that would increase its power consuption and modularity for use by other projects. For example, the clock signal could be AND gated with a run signal that would allow complete removal of the clock signal to the logical subsystems. This would allow for cleaner code inside the PISO module that would in turn make error checking in the FSM much easier.

Overall, this lab goes to show that there two very similar, yet distinct ways to design and model finite state machines. By using a Mealy model, this lab is able to show that even though there may be inherant risk for data instability, that risk is either minute enough to ignore or can be controlled by outside systems. This lab also clearly demonstrates the power and versatiliy of FSM logic.

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: LAB 2 SYSTEM

```
library ieee,work;
use ieee.std_logic_1164.all;
use work.all;

entity Lab2 is
  port (
    KEY     : in  std_logic_vector(1 downto 0);
    CLOCK_50 : in std_logic;
    SW      : in  std_logic_vector(17 downto 0);
    LEDR    : out std_logic_vector( 2 downto 0) := "000"
  );
end entity Lab2;

architecture behav of Lab2 is
  -- Thaipsch
  type state is (A,B,C,D);

  -- Schignuls
  signal currs : state;
  signal sflag : std_logic := '0';
  signal seqin : std_logic_vector(1 downto 0);
  signal count : integer range 0 to 3;

  -- Pair-El-El to Cereal Owt-Puht
  component PISO is
    port (
      clk,load : in std_logic;
      pin     : in  std_logic_vector(17 downto 0);
      sout    : out std_logic_vector( 1 downto 0)
    );
  end component PISO;
```

```
  -- Kowntur
  component counter is
    port (
      reset : in std_logic;
      flag  : in std_logic;
      clock : in std_logic;
      cntout : out integer range 0 to 3
    );
  end component counter;

  -- Ark-It-Eck-Sure
begin
  -- Peeso Kumpownent
  peeso : PISO PORT MAP (
    clk             => CLOCK_50,
    load            => KEY(1),
    pin(17 downto 0) => SW(17 downto 0),
    sout            => seqin
  );

  -- Kowntur Kumpownent
  kowntur : counter PORT MAP (
    reset           => KEY(0),
    flag            => sflag,
    clock           => CLOCK_50,
    cntout          => count
  );

  -- Ef-Esh-Ehm Lawjick
  fsm_lawjick : process (CLOCK_50, seqin, KEY(0))
  begin
    if (KEY(0) = '0') then
      currs <= A;
      sflag <= '0';
    elsif (rising_edge(CLOCK_50)) then
```

```
    sflag <= '0';
    case currs is
      when A =>
        if (seqin = "01") then currs <= B; end if;
      when B =>
        if (seqin = "00") then currs <= C;
        elsif (seqin = "11") then currs <= D;
        elsif (seqin = "01") then currs <= B;
        else currs <= A; end if;
      when C =>
        if (seqin = "00" or seqin = "11") then
          sflag <= '1';
          currs <= A;
        elsif (seqin = "01") then currs <= B;
        else currs <= A; end if;
      when D =>
        if (seqin = "00") then
          sflag <= '1';
          currs <= A;
        elsif (seqin = "01") then currs <= B;
        else currs <= A; end if;
      when others =>
        currs <= A;
    end case;
  end if;
end process fsm_lawjick;

-- Ellie-Dee Lawjick
elliedee : process (count)
begin
  case count is
    when 1 =>
      LEDR(2) <= '0';
      LEDR(1) <= '0';
      LEDR(0) <= '1';
    when 2 =>
```

```
        LEDR(2) <= '0';
        LEDR(1) <= '1';
        LEDR(0) <= '1';
      when 3 =>
        LEDR(2) <= '1';
        LEDR(1) <= '1';
        LEDR(0) <= '1';
      when others =>
        LEDR(2) <= '0';
        LEDR(1) <= '0';
        LEDR(0) <= '0';
    end case;
  end process elliedee;

end architecture behav;
```

APPENDIX B: FSM TESTBENCH

```
library ieee,work;
use ieee.std_logic_1164.all;
use work.clk_package.all;
use work.all;

entity test2 is --test-bench
end entity test2;

architecture behav of test2 is
  component lab2 is
    port (
      KEY     : in  std_logic_vector(1 downto 0);
      CLOCK_50 : in std_logic;
      SW      : in  std_logic_vector(17 downto 0);
      LEDR    : out std_logic_vector( 2 downto 0)
    );
  end component;

  signal keyt  : std_logic_vector( 1 downto 0) := "00";
  signal clockt : std_logic;
  signal swst  : std_logic_vector(17 downto 0);
  signal ledrt : std_logic_vector( 2 downto 0);

  signal run   : std_logic := '1';

begin

  labtest : lab2
  port map (keyt, clockt, swst, ledrt);

  clk_gen(clockt, 50.000E6, 0 fs, run);
```

```
  test : process is
  begin
    swst  <= "0100110001010100001";
    keyt(0) <= '1'; wait for 25 ns;
    keyt(1) <= '1'; wait for 127 ns;
    keyt(0) <= '0'; wait for 3 ns;
    keyt(0) <= '1'; wait;

  end process;

end architecture behav;
```

APPENDIX C: PISO TESTBENCH

```
library ieee,work;
use ieee.std_logic_1164.all;
use work.clk_package.all;
use work.all;

entity testpiso is --test-bench
end entity testpiso;

architecture behav of testpiso is
  -- Pair-El-El to Cereal Owt-Puht
  component PISO is
    port (
      clk,load : in std_logic;
      pin      : in  std_logic_vector(17 downto 0);
      sout     : out std_logic_vector( 1 downto 0)
    );
  end component PISO;

  signal clockt : std_logic;
  signal load  : std_logic;
  signal swst  : std_logic_vector(17 downto 0);
  signal outdt : std_logic_vector( 1 downto 0);

  signal run   : std_logic := '1';

begin

  pisotest : piso
  port map (clockt,load,swst,outdt);

  clk_gen(clockt, 50.000E6, 0 fs, run);
```

```
  test : process is
  begin
    swst  <= "0100110001010100001";
    load <= '1'; wait for 1 ns;
    load <= '0'; wait for 10 ns;
    load <= '1'; wait;

  end process;

end architecture behav;
```

APPENDIX D: COUNTER TESTBENCH

```
library ieee,work;
use ieee.std_logic_1164.all;
use work.clk_package.all;
use work.all;

% entity testcounter is --test-bench
end entity testcounter;

architecture behav of testcounter is
  -- Kowntur
  component counter is
    port (
      reset : in std_logic;
      flag  : in std_logic;
      clock : in std_logic;
      cntout : out integer range 0 to 3
    );
  end component counter;

  signal clock : std_logic;
  signal reset : std_logic := '1';
  signal flag  : std_logic := '0';
  signal cntout : integer range 0 to 3;

  signal run   : std_logic := '1';

begin

  countertest : counter
  port map (reset,flag,clock,cntout);
```

```
  clk_gen(clock, 50.000E6, 0 fs, run);

  test : process is
  begin
    wait for 20 ns;
    flag  <= '1'; wait for 5 ns;
    flag  <= '0'; wait for 15 ns;
    flag  <= '1'; wait for 5 ns;
    flag  <= '0'; wait for 15 ns;
    flag  <= '1'; wait for 5 ns;
    flag  <= '0'; wait for 10 ns;
    reset <= '0'; wait for 1 ns;
    reset <= '1'; wait for 24 ns;
    flag  <= '1'; wait for 5 ns;
    flag  <= '0'; wait for 15 ns;
    flag  <= '1'; wait for 5 ns;
    flag  <= '0'; wait;

  end process;

end architecture behav;
```