4-Way Traffic Light Regulator

Beau Tucker, Ben Bonham, Nick Schneider
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: bstucker@oakland.edu, bmbonham@oakland.edu, nschneider@oakland.edu

***Abstract*- Outdated analog control systems have become problematic for aging infrastructure. This work attempts to characterize and simulate a four-way traffic indicator controller by way of FPGA digital design. Traffic controller logic will be achieved using FSM theory in parallel with various combinational digital circuitry. This design will then be implemented and demonstrated via a Nexys A7-100T board.**

I. Introduction

The streets around Oakland county are plagued with lights running off of old electrical architectures. These include systems based on outdated analog circuitry, resulting in increased disturbances to traffic patterns. Often these control devices do not allow for quick upgrades to be made as many physical components would need to be altered. This project seeks to address this issue by implementing the use of a Finite State Machine (FSM) and other various digital asynchronous systems to construct a controllable four-way traffic signal (shown in figure 1 below). This traffic signal will be capable of state changes, allowing for alternative configurations for various output types. To achieve this, the system will be modeled utilizing a Field-Programmable Gate Array (FPGA) board to synthesize the transistor-to-transistor logic.

II. Methodology

A. Basic Intersection Design

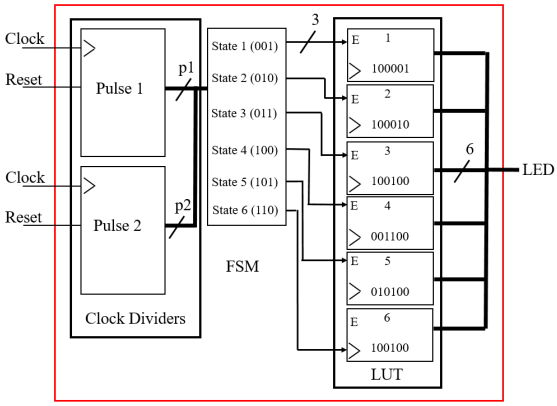

Figure 1: Proposed Block Diagram

To begin the process of designing a 4-way traffic intersection, it is critical to understand the physical layout and structure. Figure 2 represents the proposed intersection and the lights required to control it. The design here is simple in that both the North and South facing lights (blue quadrant) will function in unison while the East and West lights (yellow quadrant) follow a similar scheme. From a design standpoint, this means that six individual lights will be needed, for a total combination of five discrete lighting configurations. These different combinations will be described in more detail when discussing the proposed finite state machine that will control the intersection.

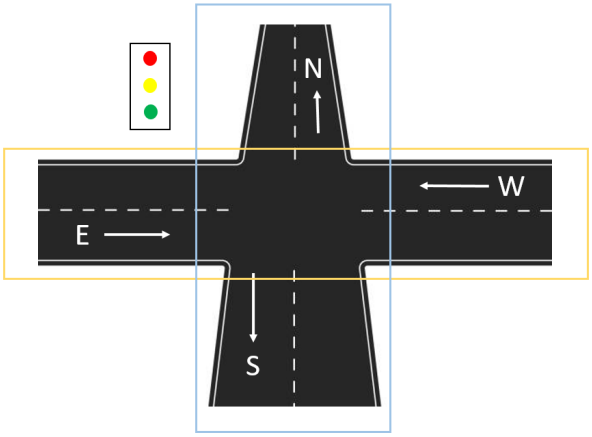

Figure 2: Basic Intersection Layout

Lighting control can be completed in a number of different ways, however the most common method is based solely on timing. Here, a timed approach will be implemented as it can readily be adjusted to account for various traffic conditions. Intersection control logic will be designed using VHSIC Hardware Description Language (VHDL) code, and implemented with a Nexys A7-100T FPGA board. It is important to note that this board provides a continuous 100 Mhz internal clock that will be utilized to provide intersection timing, as well as other button inputs to simulate real-world intersection inputs.

B. *Timing Control - Pulse Clock Dividers*

As shown in the block diagram of Figure 1, timing for the FSM controlled intersection will be achieved using pulse generating clock dividers. Here there are two that will be implemented, known as P1 & P2 (pulse 1, pulse 2). This form of clock divider was chosen because it can provide the FSM with a timed logic high or logic low signal. Referencing Appendix A for the pulse generator code, it is important to note two variables that can be easily changed to provide various intersection timing conditions. The first variable located in Line 10 "signal b" can be changed to alter the starting logic of the pulse timer. This was done to provide two alternating one second pulses, P1 and P2, that work directly inverse of each other. The second variable of interest is located in Line 20. Here the "count" number can be changed to generate different pulse widths. Since the internal clock of the Nexys A7-100T board runs at 100 Mhz, a count number of 100,000,000 provides a 1-second pulse (count = 100000000). Changing this number by a factor of 2 (count = 200000000) will then generate a 2-second pulse. It is clear to see that this bit of code is particularly useful in its potential for quick alterations to the timing scheme.

C. *The Finite State Machine (FSM)*

Of the two types of Finite State Machines (FSM) currently utilized in entry-level computing, the Mealy style FSM has been chosen here. Mealy-type FSM logic is important because the outputs of the device depend on the current state of the FSM, in addition to various external inputs [1]. This means that for the proposed FSM, the individual outputs are decided by the state of the device as well as the input signals (P1 and P2). For the purposes of controlling a traffic intersection containing signal lights, the FSM here is proposed to have six individual states as shown in Figure 3.

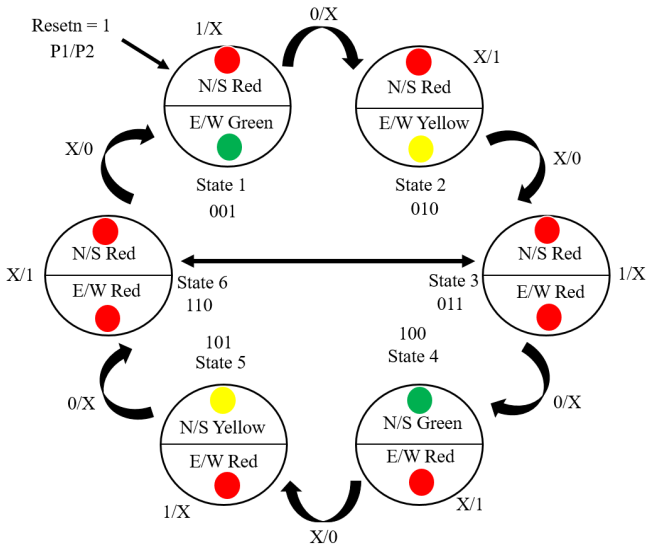

Figure 3: FSM Diagram

Close inspection of the state diagram would indicate that two states, 3 and 6, are interchangeable, however they are treated as unique states with identical outputs. Table 1 shows the excitation table for the designed FSM with the expected inputs from each pulse generator, P1 and P2, as well as the expected outputs from each given state. This design utilizes a logic low signal to transition from state to state, while a logic high indicates that the state should remain the same.

State Assignment		P1	P2	Present State	Next State	FSM Output (State)	LUT Output
S1	001	0	x	S1	S2	010	100010
S2	010	x	0	S2	S3	011	100100
S3/S6	011	0	x	S3	S4	100	001100
S4	100	x	0	S4	S5	101	010100
S5	101	0	x	S5	S6	011	100100
		x	0	S6	S1	001	100001
		1	x	S1	S1	001	100001
		x	1	S2	S2	010	100010
		1	x	S3	S3	011	100100
		x	1	S4	S4	100	001100
		1	x	S5	S5	101	010100
		x	1	S6	S6	011	100100

Table 1: Excitation Table

D. *LUT*

The Look-Up Table (LUT) in this design is to function as a read-only memory (ROM), and read from the FSM and output a 6-bit number for each state (shown in Table 1). As the FSM generates a 3 bit output, the number is sent to the LUT in the form of an enable signal (FSM Output in Table 1). The 6-bit stored number for that state is then output to LED's simulating the traffic indicator lights. Since the LED's are set to illuminate when a logic high is sent, the 6-bit number is interpreted as follows: "RYGRYG". where the three most significant bits indicate the N/S facing lights, and the three least significant bits for E/W facing lights. Lines 11 to 16 of Appendix C shows the various six-digit combinations needed to fulfill the requirements of the intersection lighting.

III. Experimental Setup

All coding was completed using Vivado design suite and VHDL language. To simulate the function of the circuit, a test bench was created that allowed the full system to run for a total of six seconds. Figure 4 shows the final output of the device where the signals (sig[0-5]) indicate the 6-bit digit output that is used to power each individual LED. Indeed these values match up with the expected outputs shown on the excitation table.

With the simulation completed, the final circuit design was then programmed onto the Nexys A7 100T board. Traffic lights were then simulated by wiring each LED input signal to one of the Pmod outputs and connecting that to a breadboard. N/S and E/W lights were then wired together so that a total of four traffic lights would be functioning. To ensure proper functioning of each LED,

220 Ω dropdown resistors were used. This can be seen in Figure 5.

IV. Results

Figure 4 shows the results from the experimental simulation of the traffic control system. Column one shows that within the first second, the generated output is "sig = 100001" which is equivalent to state 1 (S1) of the FSM diagram. Continuing on to later intervals of time such as column four, output is then "sig = 001100", meaning that S4 has been achieved. Further examination of each column indicated that the desired states and outputs have been achieved as intended. To further verify the system is running properly, the values match up with the excitation table shown in Table 1.

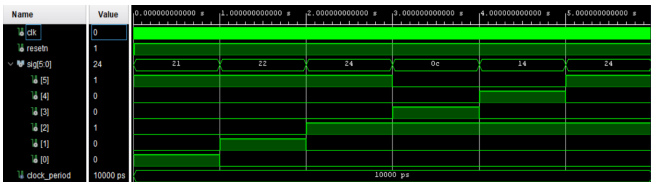

Figure 4: Simulation Results

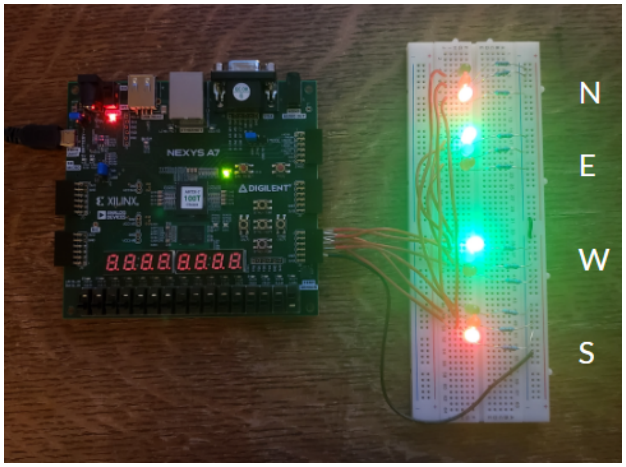

Figure 5: FPGA board with Circuit Connection

Conclusions

Finite state machines can provide a relatively simple means to solve some outwardly difficult problems. In the case of this traffic control system, the FSM model provided the ideal solution for creating a time based intersection. In addition to the benefits the FSM provides, it remains clear that implementing other simple combinatorial circuits around them can provide an even greater tool. The use of VHDL coding language also provides a modular approach to the problem that allows for significant upgradability. Should one have an intersection with more lights, or simply need different timing cycles, only a few lines of code need to be altered to achieve the desired outcome. Additionally, the use of HDL code allows for quick modeling of a circuit without having to undergo tedious circuitry design by hand. Some additional work to follow this study should include the addition of various traffic indication signals, further improving the experience of traffic intersections. Although timed lights can potentially alleviate traffic congestion, the implementation of in-the-road sensors to dynamically vary signal timing could be of greater use. It is also possible that this design could be used in parallel with other signal logic systems to include additional lighting indicators such as crosswalks or turn indicators.

References

[1] Llamoca slides VHDL Coding for FPGAs Unit 6 https://moodle.oakland.edu/pluginfile.php/7846507/mod_resource/content/3/Unit%206.pdf

[2] Radio Electronics, Computer Science, Control - National University. The Scientific Journal

[3] Llamoca "4 to 1 LUT" VHDL coding for FPGAs http://www.secs.oakland.edu/~llamocca/VHDLforFPGAs.html

[4] "Clock Divider In VHDL." Stack Overflow. https://stackoverflow.com/questions/61878127/clock-divider-in-vhdl-from-100mhz-to-1hz-code

[5] "Design of a VHDL LUT Module" Stack Overflow https://stackoverflow.com/questions/21976749/design-of-a-vhdl-lut-module

[6] Nexys A-7 FPGA Trainer Board-Reference Manuel https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual?redirect=1

Appendix

A. *Pulse Generator* [4]

```
1.  entity pulse_1 is
2.  port (clk1 : in std_logic;
3.        clr : in std_logic;
4.         clk : out std_logic);
5.  end pulse_1;
6.
7.  architecture Behavioral of pulse_1 is
8.
9.  signal count : integer :=0;
10. signal b : std_logic :='0';
11. begin
12.
13. process(clk1)
14. begin
15. if clr = '0' then
16. count <= 0;
17. elsif(rising_edge(clk1)) then
18. count <=count+1;
19. --1 SECOND PULSE
20. if(count = 100000000) then
21. b <= not b;
22. count <=0;
23.
24. end if;
25. end if;
```

```
clk<=b;
end process;
end Behavioral;
```

B. FSM [1]

```
entity FSM is
Port (resetn, clk, p1, p2, p3: in std_logic;
      s: out std_logic_vector(2 downto 0));
end FSM;

architecture behavioral of FSM is

type state is (S1, S2, S3, S4, S5, S6);
signal y: state;
begin
   transistions: process (resetn, clk)
   begin
      if resetn = '0' then y <= S1;
      elsif (clk'event and clk = '1') then
         case y is
            when S1 => --R/G NS/EW
               if p2 = '0' then
                  y <= S2;
               else
                  y <= S1;end if;
            when S2 => --R/Y NS/EW
               if p1 = '0' then
                  y <= S3;
               else
                  y <= S2;end if;
            when S3 => --R/R NS/EW
               if p2 = '0' then
                  y <= S4;
               else
                  y <= S3;end if;
            when S4 => --G/R NS/EW
               if p1 = '0' then
                  y <= S5;
               else
                  y <= S4;end if;
            when S5 => --Y/R NS/EW
               if p2 = '0' then
                  y <= S6;
               else
                  y <= S5;end if;
            when S6 => --R/R NS/EW
               if p1 = '0' then
                  y <= S1;
               else
                  y <= S6;end if;
      end case;
   end if;
end process;

output: process (y)
begin
s <= "000";
case y is
   when S1 => s <= "001"; --if p1 = '0' then r1 <= '0'; r2 <= '1'; r3 <= '1'; end if;

   when S2 => s <= "010"; --if p2 = '0' then r1 <= '1'; r2 <= '0'; r3 <= '1'; end if;

   when S3 => s <= "011"; --if p3 = '0' then r1 <= '1'; r2 <= '1'; r3 <= '0'; end if;

   when S4 => s <= "100"; --if p1 = '0' then r1 <= '0'; r2 <= '1'; r3 <= '1'; end if;

   when S5 => s <= "101"; --if p2 = '0' then r1 <= '1'; r2 <= '0'; r3 <= '1'; end if;

   when S6 => s <= "011"; --if p3 = '0' then r1 <= '1'; r2 <= '1'; r3 <= '0'; end if;

end case;
end process;
end behavioral;
```

C. LUT [5]

```
entity LUT is

port ( state: in std_logic_vector (2 downto 0);
       OLUT: out std_logic_vector (5 downto 0));
end LUT;

architecture struct of LUT is
begin
        with state select

 OLUT <= "100001" when "001", --
         "100010" when "010",
         "100100" when "011",
         "001100" when "100",
         "010100" when "101",
         "000000" when others;

        end struct;
```