Design of Basic Gates and I/O

Lab Report 1 for ECE327

Digital Systems Design

Submitted by

Julian Coy

Undergraduate of Electrical & Computer Engineering

Clemson University

February 5, 2013

Abstract

Digital system design is the process of creating a complex digital system by building and integrating various fundamental components. In this lab, multiplexers and decoders are implemented to familiarize students with the operation of an FPGA device and the Altera development environment. Testbenches are used to demonstrate the power and usefulness of simulations. The simple elements designed in this lab can be used as basic components in much larger, more complicated designs. It is critical to grasp the scope and scalability of this technology. This lab also focuses on connecting smaller components to make larger more complicated processes simpler. The final part of this lab, the 5 piece seven-segment decoder is the culmination of these principles.

Introduction

Lab one is broken into four distinct subsystems. Each system is standalone and can be simulated through ModelSim for testing and also implemented on the Altera DE 2 FPGA board. The first system is a 2-to-1 multiplexer with an 8-bit input/output. The second system is a modification of the first. It expands the multiplexer to a 5-to-1 but decreases the input/output channel size to 3 bits. The third system is a seven segment display implementation do display the letters 'T', 'I', 'G', 'E', and 'R'. The fourth and final system is a combination of the previous two. The design uses a 5-to-1 multiplexer with 5 seven segment displays. The seven segment displays are capable of showing a rotating string based upon input values defined by the engineer. Each of these systems is comparable to real-world counterparts. They are an analogy to other systems in which complex designs are built from smaller, simpler components.

1.1 2-to-1 Multiplexer

1.1.1 Design of the 2-to-1 Multiplexer

Building the 2-to-1 multiplexer system in VHDL requires 3 signals: a select signal, an 8-bit input channel, and an 8-bit output channel. Once those are defined the engineer has multiple options on how to implement the functionality. Figure 1.2 shows the digital logic implementation of the first system. By using the powerful non-linearity of VHDL the entire behavior of the multiplexer can be implemented in one simple if-then-else block (Figure 1.1).

```
IF (S = '1') THEN
    M <= Y;
ELSE
    m <= X;
END IF;
```

Figure 1.1: 2-to-1 Multiplexer Code [1]

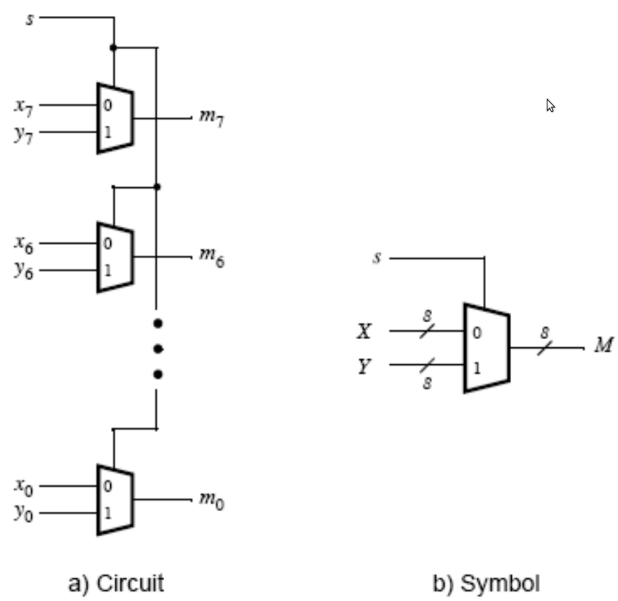

Figure 1.2: 2-to-1 Multiplexer Logic

1.1.2 Testing of the 2-to-1 Multiplexer

Testing of the multiplexer was done with a testbench through ModelSim. Inside the test bench the input signals were initialized and set to different values. Next, the channels were cycled through using the select signal. Once cycled, the signal that was selected was changed and the change was verified through the Wave view in ModelSim. Finally, the signal that was not selected was changed and monitored to make sure the output did not change.

Once the testbench was completed, the code was compiled inside of Quartus II and flashed onto the Cyclone IV FPGA. Once the code was loaded onto the FPGA the switches and LEDS were used to verify the operation of the code. For this multiplexer design, the switches 15 to 8 were used for the Y channel and switched 7 to 0 were used for X. Switch 16 was used as the select signal and the red LEDs were connected to the output channel. By toggling through the earlier mentioned states, the program was verified in hardware.

1.2 5-to-1 Multiplexer

1.2.1 Designing the 5-to-1 Multiplexer

The second system was a 5-to-1 multiplexer design. This design is very similar to the previous system. The addition of three more channels will require a more complex decoding algorithm. This can complicate the design greatly; however, smart implementation techniques can limit the overhead of the computation.

As shown in Figure 1.3 a simple case block was sufficient to perform the select signal decoding for the 3 bit select line. This method is inefficient with larger select lines as they exponentially increase the number of required cases per select channel bit.

```
CASE S(2 DOWNTO 0) IS
    WHEN "001"   => M <= Y;
    WHEN "010"   => M <= X;
    WHEN "011"   => M <= W;
    WHEN "100"   => M <= V;
    WHEN "101"   => M <= U;
    WHEN OTHERS => M <= "111";
END CASE;
```

Figure 1.3: 5-to-1 Decoder

1.2.2 Testing the 5-to-1 Multiplexer

The testing of the 5-to-1 multiplexer was almost exactly the same as the test for the 2-to-1. Figure 1.4 shows the timing diagram of the signals. The diagram shows the transition from the output signal (TESTM) as the select signal (TESTS) is modified. The first 5 changes for TESTS show the value of the output matching the selected input. The second to last change shows that non-selected input does not affect the output. The final change to TESTW when it is selected shows that the output changes immediately with the selected signal.

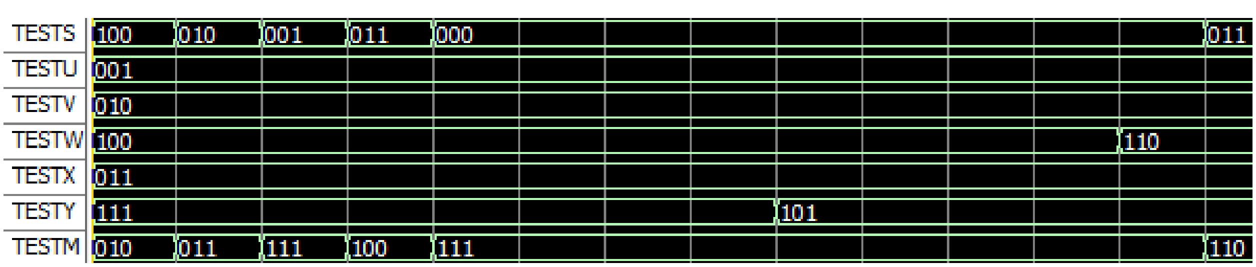

Figure 1.4: 5-to-1 Decoder

1.3 Seven Segment Decoder

1.3.1 Designing the Seven Segment Decoder

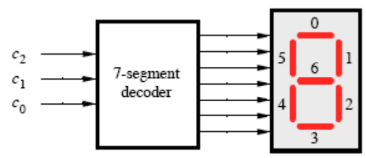

Figure 8. A 7-segment decoder.

Table 1.

$c_2c_1c_0$	Character
000	T
001	I
010	G
011	E
100	R
101	
110	
111	

Figure 1.5: Seven Segment Display Logic

The basic truth table and signal layout for the seven segment decoder (SSD) are shown in Figure 1.5. The SSD is supposed to take a 3-bit input and display a letter corresponding to the values in the truth table. This means that a maximum of 2^3 Each segment in the decoder is set by a bit. So the input channel must be 7 bits wide to accommodate the SSD.

1.3.2 Testing the Seven Segment Decoder

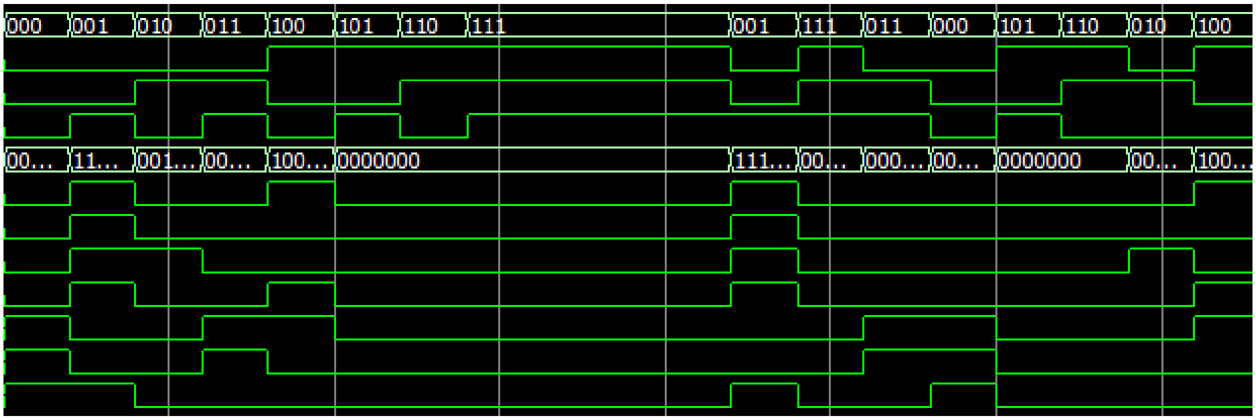

Figure 1.6: Seven Segment Timing Waveform

Testing of the SSD is more efficient on the Cyclone IV due to the nature of the visual display. For a human eye the waveform simulation was unfortunately hard to follow. Although the waveform allowed for proper logical adjustments, it was unknown that the SSD required a low bit to activate a segment and all of the letters were inverted. Figure 1.6 shows the timing waveform.

1.4 5-Seven Segment Display

1.4.1 Designing the 5-Seven Segment Display

The final system of Lab 1 is a 5-count SSD that can display a word. The word must be able to be shifted by a 3-bit select channel and each letter must be modifiable by another 3-bit select channel. Each of the previous subsystems is included for the final design. There are 5 5-to-1 multiplexers which are connected to SSDs.

$SW_{17}SW_{16}SW_{15}$	Character Pattern				
000	H	E	L	L	O
001	O	H	E	L	L
010	L	O	H	E	L
011	L	L	O	H	E
100	E	L	L	O	H

Figure 1.7: 5 Seven Segment Truth Table

Figure 1.7 shows the truth table for the select line input. This table is assuming that the SSD letter codes are not changing when the select line does. Each SSD also has a 3-bit select line that will tell the multiplexer what values to send to the decoder. The diagram for the signals is shown in Figure 1.8.

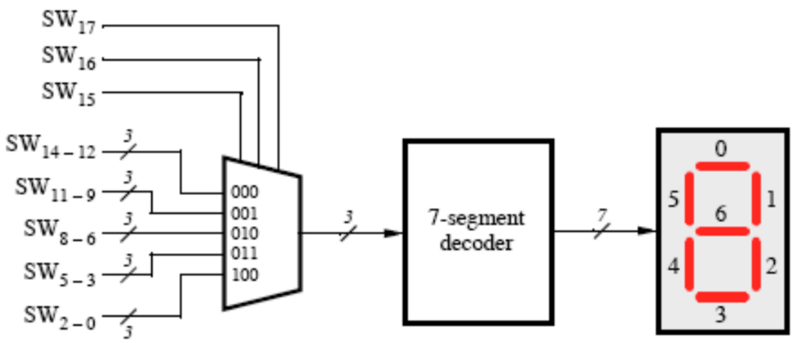

Figure 1.8: 5 Seven Segment Logic

1.4.2 Testing the 5-Seven Segment Display

As the complexity of a system grows, so does the complexity of its verification. The final system in lab 1 required multiple tests between components and tests of the components themselves to verify functionality. For the test, the SSD were initialized to display "TIGER". Then the letters were permuted through the TESTSW signal. Once that was verified, the letters were randomized and then permuted randomly through the SSDs. In the code the "default" value for the SSD was set to display the number "8" for easy debugging.

The code for the testbench can be seen in Appendix B. The waveform simulation can be found in Appendix C.

1.5 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] J. Lewis, "Vhdl quick reference." Internet: `http://www.synthworks.com/downloads/vhdl_quickref.pdf`, 2008.

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 TESTS01 : STD_LOGIC_VECTOR(6 DOWNTO 0);
  SIGNAL TESTS02 : STD_LOGIC_VECTOR(6 DOWNTO 0);
  SIGNAL TESTS03 : STD_LOGIC_VECTOR(6 DOWNTO 0);
  SIGNAL TESTS04 : STD_LOGIC_VECTOR(6 DOWNTO 0);
  SIGNAL TESTS05 : 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
```

```vhdl
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)