# John Arena

# Professor Gertner

# CSC 342/343

# Lab 1

# Due 2/20/19

# Spring 2019

# Table of Contents

**Note: Files sent to your email address on 12/3/18**

## Section 1) Objective

For this lab, the objective will be to apply everything we've learned about Quartus II in the tutorials. I will be doing the following:

- Building the following circuits using Object form and VHDL: 2to1 Multiplexer, 1-bit Half Adder, 1-bit Full Adder, 3to8 Decoder and 8to3 Encoder
- Verifying their correctness using waveform simulations
- Writing testbench files in VHDL to test the correct of the designs
- Programming pin assignments for the board

## Section 2) Description and Specifications

### 2to1 Multiplexer

The first circuit I will be designing is a 2to1 **Multiplexer**. A multiplexer, also known as a mux, is "basically a switch that passes one of its data inputs through to the output, as a function of a set of select inputs"[1]. One of their uses is to choose among several multibit input numbers. The typical logic symbol of a 2to1 Multiplexer is shown below in Figure **1**

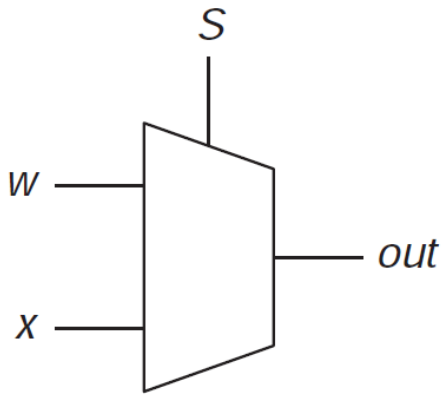



*Figure 1: 2to1 Multiplexer*

The way a 2to1 multiplexer works as follows. If the select input, *S*, is equal to 0, the output, *out*, is equal to the value of X. If the select input is equal to 1, the output is equal to the value of Y. Table **1** below shows the truth table of a 2to1 Mux. I will denote the two inputs as X and Y and the output as M.

| **X** | **Y** | **S** | **M** |
|---|---|---|---|
| **0** | **0** | **0** | **0** |
| **0** | **0** | **1** | **0** |
| **0** | **1** | **0** | **0** |
| **0** | **1** | **1** | **1** |
| **1** | **0** | **0** | **1** |
| **1** | **0** | **1** | **0** |
| **1** | **1** | **0** | **1** |
| **1** | **1** | **1** | **1** |

*Table 1: 2to1 Multiplexer Truth Table*

A simplified table is shown in Table **2** below

| **S** | **M** |
|---|---|
| **0** | **X** |
| **1** | **Y** |

*Table 2: 2to1 Multiplexer - Simplified Truth Table*

We can derive the Boolean algebra expression of a 2to1 multiplexer from table **1** Looking at the table, we get the following

$$M = XS' + YS$$

*Equation 1: Out of 2to1 Mux*

This can be composed of two AND gates, one OR gate and one NOT gate. The design of AND, OR and NOT gates can be designed using transistors, but are not within the scope of this course, so shall not be discussed.

The inputs and outputs will be assigned as follows on our board, seen in Figure **2** below. It comes from the pin assignment text file for this circuit.

```
To, Location
Arena_X, PIN_N25
Arena_Y, PIN_N26
Arena_S, PIN_P25
Arena_M, PIN_AE23
```

*Figure 2: Pin Assignment for 2to1 Mux*

The format is as follows. To, Location. To is the input/outputs from the object/vhdl file. The Location is the appropriate pins used for inputs and outputs. The pins are gotten from the pin assignment file.

On the next page in Figure **3** is the design I made in Quartus for the 2to1 Multiplexer.

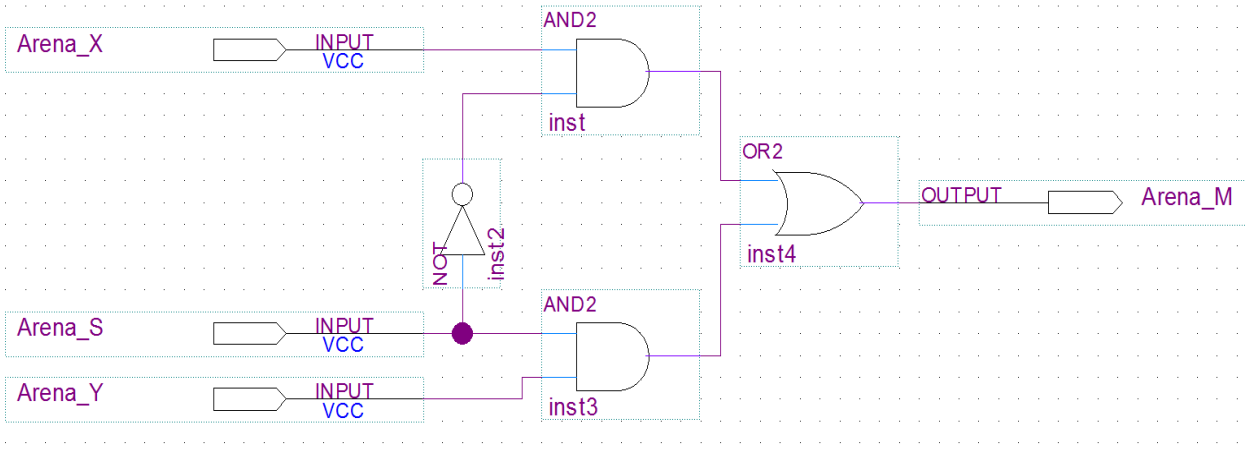



*Figure 3: 2to1 Multiplexer at the Logic Level*

As can be seen in the figure, the output consists of an 2-input OR gate connected to two 2-input AND gates with the appropriate inputs, with one AND gate having it's input connected to a NOT gate just as described in Equation 1.

Below in Figure **4** is the VHDL code for the circuit, generated by the MegaWizard tool.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 1 - Spring 2019 Due: 2/20/19
-- megafunction wizard: %LPM_MUX%
-- GENERATION: STANDARD
-- VERSION: WM1.0
-- MODULE: LPM_MUX

-- ============================================================
-- File Name: Arena_muxLPM.vhd
-- Megafunction Name(s):
--                LPM_MUX
--
-- Simulation Library Files(s):
--                lpm
-- ============================================================
-- ************************************************************
-- THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
--
-- 13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
-- ************************************************************


--Copyright (C) 1991-2013 Altera Corporation
--Your use of Altera Corporation's design tools, logic functions
--and other software and tools, and its AMPP partner logic
--functions, and any output files from any of the foregoing
```

```
--(including device programming or simulation files), and any
--associated documentation or information are expressly subject
--to the terms and conditions of the Altera Program License
--Subscription Agreement, Altera MegaCore Function License
--Agreement, or other applicable license agreement, including,
--without limitation, that your use is for the sole purpose of
--programming logic devices manufactured by Altera and sold by
--Altera or its authorized distributors.  Please refer to the
--applicable agreement for further details.


LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY lpm;
USE lpm.lpm_components.all;

ENTITY Arena_muxLPM IS
      PORT
      (
            Arena_data0       : IN STD_LOGIC ; --Appropriate inputs/ outputs
            Arena_data1       : IN STD_LOGIC ; --for the external interface
            Arena_sel         : IN STD_LOGIC ;
            Arena_result            : OUT STD_LOGIC
      );
END Arena_muxLPM;


ARCHITECTURE SYN OF arena_muxlpm IS

--    type STD_LOGIC_2D is array (NATURAL RANGE <>, NATURAL RANGE <>) of
STD_LOGIC;

      SIGNAL Arena_sub_wire0  : STD_LOGIC_VECTOR (0 DOWNTO 0); Various vars
      SIGNAL Arena_sub_wire1  : STD_LOGIC ;
      SIGNAL Arena_sub_wire2  : STD_LOGIC ;
      SIGNAL Arena_sub_wire3  : STD_LOGIC_2D (1 DOWNTO 0, 0 DOWNTO 0);
      SIGNAL Arena_sub_wire4  : STD_LOGIC ;
      SIGNAL Arena_sub_wire5  : STD_LOGIC ;
      SIGNAL Arena_sub_wire6  : STD_LOGIC_VECTOR (0 DOWNTO 0);

BEGIN
      Arena_sub_wire4    <= Arena_data0; --Appropriate assignments
      Arena_sub_wire1    <= Arena_sub_wire0(0);
      Arena_result    <= Arena_sub_wire1;
      Arena_sub_wire2    <= Arena_data1;
      Arena_sub_wire3(1, 0)     <= Arena_sub_wire2;
      Arena_sub_wire3(0, 0)     <= Arena_sub_wire4;
      Arena_sub_wire5    <= Arena_sel;
      Arena_sub_wire6(0)     <= Arena_sub_wire5;

      LPM_MUX_component : LPM_MUX
      GENERIC MAP ( -- Passing information to an entity
            lpm_size => 2,
            lpm_type => "LPM_MUX",
            lpm_width => 1,
            lpm_widths => 1
```

```
	)
	PORT MAP ( --Port maps
		data => Arena_sub_wire3,
		sel => Arena_sub_wire6,
		result => Arena_sub_wire0
	);


END SYN;

-- ============================================================
-- CNX file retrieval info
-- ============================================================
-- Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone II"
-- Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
-- Retrieval info: PRIVATE: new_diagram STRING "1"
-- Retrieval info: LIBRARY: lpm lpm.lpm_components.all
-- Retrieval info: CONSTANT: LPM_SIZE NUMERIC "2"
-- Retrieval info: CONSTANT: LPM_TYPE STRING "LPM_MUX"
-- Retrieval info: CONSTANT: LPM_WIDTH NUMERIC "1"
-- Retrieval info: CONSTANT: LPM_WIDTHS NUMERIC "1"
-- Retrieval info: USED_PORT: data0 0 0 0 0 INPUT NODEFVAL "data0"
-- Retrieval info: USED_PORT: data1 0 0 0 0 INPUT NODEFVAL "data1"
-- Retrieval info: USED_PORT: result 0 0 0 0 OUTPUT NODEFVAL "result"
-- Retrieval info: USED_PORT: sel 0 0 0 0 INPUT NODEFVAL "sel"
-- Retrieval info: CONNECT: @data 1 0 1 0 data0 0 0 0 0
-- Retrieval info: CONNECT: @data 1 1 1 0 data1 0 0 0 0
-- Retrieval info: CONNECT: @sel 0 0 1 0 sel 0 0 0 0
-- Retrieval info: CONNECT: result 0 0 0 0 @result 0 0 1 0
-- Retrieval info: GEN_FILE: TYPE_NORMAL Arena_muxLPM.vhd TRUE
-- Retrieval info: GEN_FILE: TYPE_NORMAL Arena_muxLPM.inc FALSE
-- Retrieval info: GEN_FILE: TYPE_NORMAL Arena_muxLPM.cmp TRUE
-- Retrieval info: GEN_FILE: TYPE_NORMAL Arena_muxLPM.bsf FALSE
-- Retrieval info: GEN_FILE: TYPE_NORMAL Arena_muxLPM_inst.vhd FALSE
-- Retrieval info: LIB_FILE: lpm
```

*Figure 4 : VHDL Code for 2to1 Mux*

## 1-bit Half Adder

The second circuit I will be designing is a **1-bit Half Adder.** A 1-bit Half Adder is a circuit that can add two 1-bit numbers, and produce a Sum and a possible carry over. We know from basic math for example, if we had 3+3, we will have a sum of 6 with a carry over of 0, but 7+4 will produce a sum off 11 with a carry over of 1. We know from Boolean algebra that a 1-bit

number is base two, since it can only have two digits, either a 0 or a 1. Since that is the case, let's quickly review the rules of binary addition of 1-bit numbers

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 0, Carry\ 1$$

Knowing these rules, below in Table **3** describes the functionality of a 1-bit half adder.

| **X** | **Y** | **Sum** | **Carry Out** |
|---|---|---|---|
| **0** | **0** | **0** | **0** |
| **0** | **1** | **1** | **0** |
| **1** | **0** | **1** | **0** |
| **1** | **1** | **0** | **1** |

*Table 3: 1-bit Half Adder Truth Table*

We can derive the Boolean algebra expression of a 1-bit adder from table **3**. Looking at the table, we get the following in equation **2** below.

$$Sum = X'Y + XY'$$

$$Carry\ Out = XY$$

*Equation 2: Out of 2to1 Mux*

Looking at these functions, we can see Sum is an XOR function of variables X and Y, while Carry Out is an AND function of X and Y. These will be created using an XOR gate and an AND gate.

The inputs and outputs will be assigned as follows on our board, seen in Figure **5** below. It comes from the pin assignment text file for this circuit.

```
To, Location
Arena_X, PIN_N25
Arena_Y, PIN_N26
Arena_Sum, PIN_AE23
Arena_CarryOut, PIN_AF23
```

*Figure 5: Pin Assignment for 2to1 Mux*

There is 2 input switches used and 2 output LEDs. Below in Figure **6** is the design I made in Quartus for the 1-bit Half-Adder.

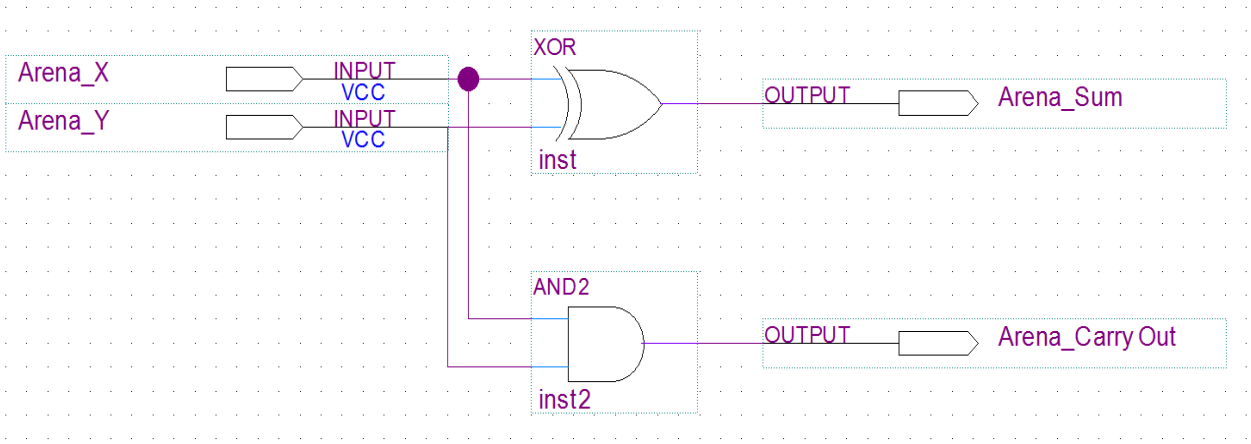



*Figure 6: 1-bit Half Adder at Gate Level*

As seen in the figure, there are two inputs, X and Y going into the XOR gate with the output as the Sum, and X and Y going into an AND gate with the output as CarryOut.

On the next page in Figure **7** is the VHDL code I created for the 1-bit Half Adder.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 1 - Spring 2019 Due: 2/20/19
-- Arena_HalfAdder.vhd

library ieee;
use ieee.std_logic_1164.all;

Entity Arena_HalfAdder is
      Port(
            Arena_X, Arena_Y: in std_logic;      -- Two Inputs for X and Y
            Arena_Sum, Arena_CarryOut: out std_logic -- Two outputs for the
Sum and the CarryOut bit.
      );
end Arena_HalfAdder; -- End of Entity Arena_HalfAdder

Architecture Arena_Arch_HalfAdder of Arena_HalfAdder is -- Architecture of
the Entity (Describes functionality)
begin
      Arena_Sum <= (Arena_X xor Arena_Y); -- Sum = X XOR Y
      Arena_CarryOut <= (Arena_X and Arena_Y); -- Carryout = X*Y
end Arena_Arch_HalfAdder; -- End of Architecture statement
```

*Figure7: 1-bit Half Adder VHDL Code*

## 1-bit Full Adder

The third circuit I will be designing is a **1-bit Full Adder.** A 1-bit Half Adder is a circuit that has three inputs. Two 1-bit numbers and one CarryIn input and produces a sum and a possible carry over. We know from basic math for example, if we have 17+14, we get 31, with a carry over and the tenth's place gets a CarryIn of 1. We know from Boolean algebra that a 1-bit number is base two, since it can only have two digits, either a 0 or a 1. Since that is the case, let's quickly review the rules of binary addition of 1-bit numbers

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 0, Carry\ 1$$

Knowing these rules, below in Table **4** describes the functionality of a 1-bit half adder.

| X | Y | Carry In | Sum | Carry Out |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

*Table 4: 1-bit Half Adder Truth Table*

We can derive the Boolean algebra expression of a 1-bit Full Adder from table **4**. Looking at the table, we get the following in equation **3** below.

$$Sum = X'Y'Ci + X'YCi' + XY'Ci' + XYCi$$

$$=> Ci(X'Y' + XY) + Ci'(X'Y + Y'X)$$

$$=> Ci(X\ xand\ Y) + Ci'(X\ xor\ Y)$$

$$=> Ci((X\ xor\ Y)') + Ci'(X\ xor\ Y)$$

$$=> \boldsymbol{Sum = Ci\ xor\ (X\ xor\ Y)}$$

$$Carry\ Out = X'YCi + XY'Ci + XYCi + XYCi'$$

$$=> Ci(X'Y + Y'X) + XY(Ci + Ci')$$

$$=> Ci(X\ xor\ Y) + XY(1)$$

$$=> \boldsymbol{Carry\ Out = Ci(X\ xor\ Y) + (XY)}$$

*Equation 3: Out of 2to1 Mux*

Looking at these equations, it can be seen it can be derived to much simpler equations using Boolean algebra,. What's interesting is as follows. Taking a look at the Sum equation, if we say for example **(X xor Y)** can be represented by **M,** we can say **Sum = Ci xor M.** This looks like what we had for the 1-bit Half Adder in equation **2**. For the Carry Out, notice we can also say **Ci(M)**, which is also from the 1-bit Half Adder. So I can design this full adder by cascading two half adders.

The inputs and outputs will be assigned as follows on our board, seen in Figure **8** below. It comes from the pin assignment text file for this circuit.

```
To, Location
Arena_X, PIN_N25
Arena_Y, PIN_N26
Arena_CarryIn, PIN_P25
Arena_Sum, PIN_AE23
Arena_CarryOut, PIN_AF23
```

*Figure 8: Pin Assignment for 2to1 Mux*

There is 3 input switches used and 2 output LEDs. Before showing the 1-bit Full Adder, below in Figure **9** is my symbol I created for a 1-bit Half Adder.

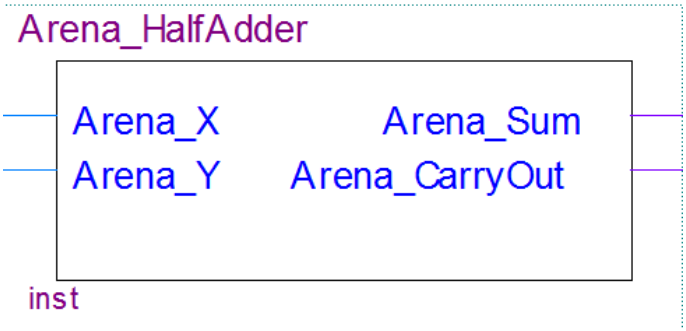



*Figure 9: 1-bit Half Adder Symbol*

Below in Figure **10** is the 1-bit Full Adder using Half Adders.

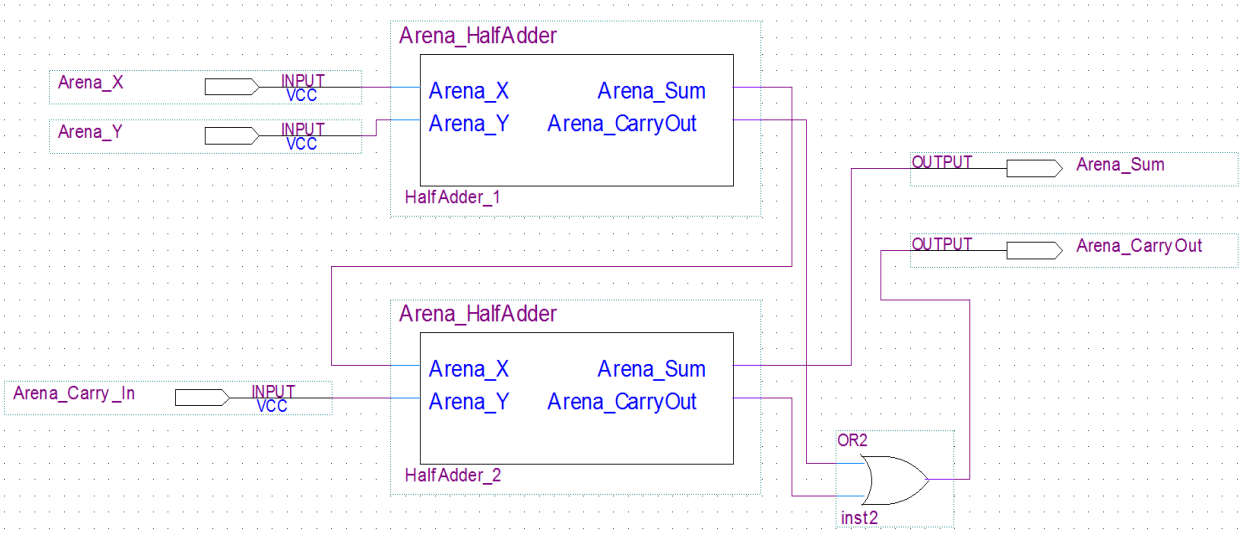



*Figure 10: 1-bit Full Adder*

As seen in the figure, there are two half adder's cascaded together.

Looking at Arena_X for Half Adder 2, it takes in the output of the Sum in Half Adder 1, which is **X xor Y**. Arena_Y for Half Adder 1 takes in the CarryIn input. Knowing the design of the half adders from **Fig 6**, Arena_Sum for Half Adder 2 will be **Sum = Ci xor (X xor Y).**

Since for the CarryOut of Half Adder 2 is $\mathbf{Carry\ Out = Ci(X\ xor\ Y) + (XY)}$**,** we know that the CarryOut of Half Adder 1 is (**XY**), so this can be plugged into an OR gate with the output of the Carryout of Half Adder 2, which gives us the final CarryOut of the Full Adder.

On the next page in figure **11** is the VHDL code I created for the 1-bit Full Adder.

```vhdl
-- (First, Last) John Arena - CSC 342/343 - Lab 1 - Spring 2019 Due: 2/20/19
-- Arena_FullAdder.vhd
library ieee;
use ieee.std_logic_1164.all;

entity Arena_FullAdder is
      port(
            Arena_X, Arena_Y, Arena_CarryIn : in std_logic; -- Three inputs,
X, Y and CarryIn
            Arena_Sum, Arena_CarryOut : out std_logic -- Two outputs, Sum
and CarryOut
      );
      end Arena_FullAdder; -- End of entity

architecture Arena_Arch_FullAdder of Arena_FullAdder is -- Architecture
describing functionality
      signal Arena_Sum1, Arena_CarryOut1, Arena_CarryOut2 : std_logic; --
Variables for mapping

      component Arena_HalfAdder -- Using Half Adder component
      port(
            Arena_X, Arena_Y : in std_logic;
            Arena_Sum, Arena_CarryOut : out std_logic
            );
end component;

begin

      HA1: Arena_HalfAdder port map (Arena_X, Arena_Y, Arena_Sum1,
Arena_CarryOut1);
      -- X into X, Y into Y, Sum1 out of Sum1, Co1 out of Co1
      HA2: Arena_HalfAdder port map(Arena_Sum1, Arena_CarryIn, Arena_Sum,
Arena_CarryOut2);
   -- Sum1 into X, Ci into Y, Sum out as final Sum, CarryOut2 out of
CarryOut2

      Arena_CarryOut <= Arena_CarryOut1 or Arena_CarryOut2; -- Final
CarryOut
      --Sum is already final, don't need a statement
end Arena_Arch_FullAdder; -- end of architecture
```

*Figure 11: 1-bit Full Adder VHDL Code*

## 3to8 Decoder

The fourth circuit I will be designing is a **3 to 8 Decoder.** A decoder, also known as a binary decoder, "is a device that, when activated, selects one of several output lines, based on a coded input signal. Most commonly, the input is an n-bit binary number, and there are up to 2^n

output lines." "The inputs are treated as a binary number, and the output selected is made active". There are two forms of decoders. One is called *active high*, and one is called *active low*. Active high means an active output is 1 and an inactive output is 0. Active low is the opposite, where an active output is 0 and an inactive output is 1. For my design, I will be using an active high.

As the name states, this is a 3to8 decoder, meaning a 3 bit number to 8 corresponding output lines. As the definition even said, with n=3, 2^3 = 8. With that said, I came up with the truth table in table **5** below. The 3 inputs shall be denoted **A, B, C** and the outputs as **F0, F1…F7** for a total of 8.

| A | B | C | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

*Table 5: 3to8 Decoder Truth Table*

We can derive the Boolean algebra expression of each active high. They are as follows

$$F0 = A'B'C', \qquad F1 = A'B'C, \qquad F2 = A'BC', \qquad F3 = A'BC$$

$$F4 = AB'C' \qquad F5 = AB'C \qquad F6 = ABC' \qquad F7 = ABC$$

Looking at these equations, it can be seen these are all 3 input AND gates with NOT gates. So the design will consist of 8 three-input AND gates and 3 NOT gates.

The inputs and outputs will be assigned as follows on our board, seen in Figure **12** below. It comes from the pin assignment text file for this circuit seen below in Figure **12**.

```
To, Location
Arena_A, PIN_N25
Arena_B, PIN_N26
Arena_C, PIN_P25
Arena_F0, PIN_AE23
Arena_F1, PIN_AF23
Arena_F2, PIN_AB21
Arena_F3, PIN_AC22
Arena_F4, PIN_AD22
Arena_F5, PIN_AD23
Arena_F6, PIN_AD21
Arena_F7, PIN_AC21
```

*Figure 12: Pin Assignment for 2to1 Mux*

There is 3 input switches used and 8 output LEDs. On the next page in figure **13** is the design of the circuit.

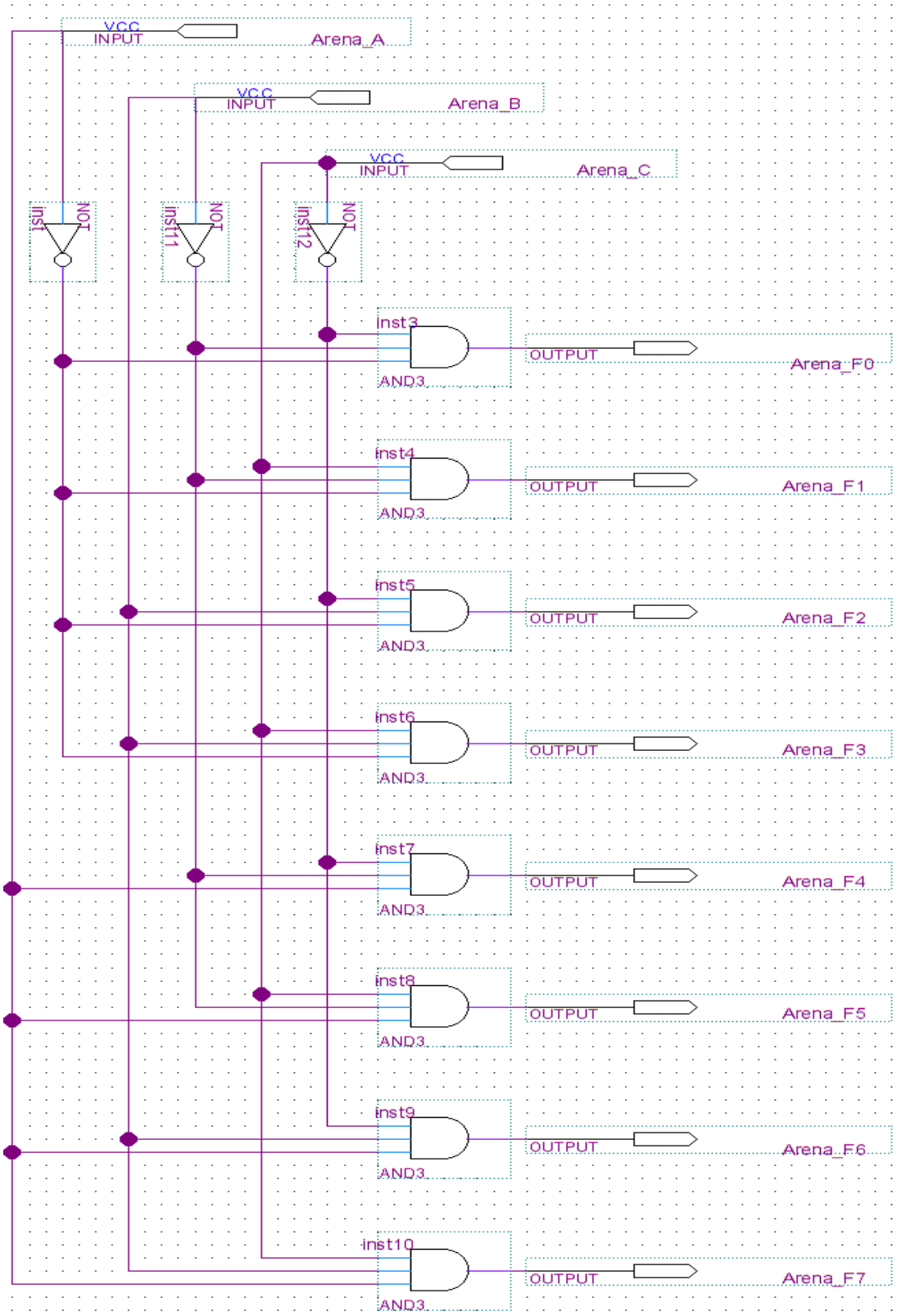



*Figure 23: 3to8 Decoder Gate Level*

Below in figure **14** is the VHDL code I created for the 3to8 Decoder.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 1 - Spring 2019 Due: 2/20/19
-- Arena_3to8Decoder.vhd
library ieee;
use ieee.std_logic_1164.all;

entity Arena_3to8Decoder is -- Decoder entity
      port(
            Arena_A, Arena_B, Arena_C : in std_logic;    -- 3 inputs, A B
and C
            Arena_F0, Arena_F1, Arena_F2 : out std_logic; -- 8 outputs,
F0..F7
            Arena_F3, Arena_F4, Arena_F5 : out std_logic;
            Arena_F6, Arena_F7 : out std_logic -- all the way up to F7
            );
end Arena_3to8Decoder; -- end of entity

architecture Arena_Arch_3to8Decoder of Arena_3to8Decoder is -- Architecture
of Decoder, describing functionality

begin
      Arena_F0 <= '1' when (Arena_A = '0' and Arena_B = '0' and Arena_C =
'0') else '0'; -- Set high when appropriate
      Arena_F1 <= '1' when (Arena_A = '0' and Arena_B = '0' and Arena_C =
'1') else '0'; -- Otherwise 0
      Arena_F2 <= '1' when (Arena_A = '0' and Arena_B = '1' and Arena_C =
'0') else '0';
      Arena_F3 <= '1' when (Arena_A = '0' and Arena_B = '1' and Arena_C =
'1') else '0';
      Arena_F4 <= '1' when (Arena_A = '1' and Arena_B = '0' and Arena_C =
'0') else '0';
      Arena_F5 <= '1' when (Arena_A = '1' and Arena_B = '0' and Arena_C =
'1') else '0';
      Arena_F6 <= '1' when (Arena_A = '1' and Arena_B = '1' and Arena_C =
'0') else '0';
      Arena_F7 <= '1' when (Arena_A = '1' and Arena_B = '1' and Arena_C =
'1') else '0';
end Arena_Arch_3to8Decoder; -- end of architecture
```

*Figure 34: 3to8 Decoder VHDL code*

8to3 Encoder

The fifth and final circuit I will be designing is a **8 to 3 Encoder.** A decoder, also known as a *binary encoder* is essentially the inverse of a binary decoder. So essentially whatever is active high (assuming this is an active high encoder), it will produce a 3-bit binary number output. “It is useful when one of several devices may be signaling a computer (by putting a 1 on a wire from that device); the encoder then produces the device number).” For my design, I will be using an active high.

As the name states, this is a 8to3 Encoder, meaning an 8-bit input to a 3-bit output.. We know a 3-bit number has 2^3=8 numbers, 0-7, so we need 8 states, so 8 possible inputs. With that said, I came up with the truth table in table **6** below. The 8 inputs shall be denoted **Y0, Y1…Y7** and the output as **A, B and C.**

| Y0 | Y1 | Y2 | Y3 | Y4 | Y5 | Y6 | Y7 | F0 | F1 | F2 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

*Table6: 8to3 Encoder Truth Table*

We can derive the Boolean algebra expression of each output. They are as follows

$$\boldsymbol{F0 = Y3 + Y2 + Y1 + Y0}$$

$$\boldsymbol{F1 = Y5 + Y4 + Y1 + Y0}$$

$$\boldsymbol{F2 = Y6 + Y4 + Y2 + Y0}$$

Looking at these equations, it can be seen these are all 4 input OR gates. The design will consist of 3 OR gates.

The inputs and outputs will be assigned as follows on our board, seen in Figure **15** below. It comes from the pin assignment text file for this circuit seen below in Figure **15.**

```
To, Location
Arena_Y0, PIN_N25
Arena_Y1, PIN_N26
Arena_Y2, PIN_P25
Arena_Y3, PIN_AE14
Arena_Y4, PIN_AF14
Arena_Y5, PIN_AD13
Arena_Y6, PIN_AC13
Arena_F0, PIN_AE23
Arena_F1, PIN_AF23
Arena_F2, PIN_AB21
```

*Figure 45: Pin Assignment for 8to3 Encoder*

There is 8 input switches used and 3 output LEDs. On the next page in figure **16** is the design of the circuit.

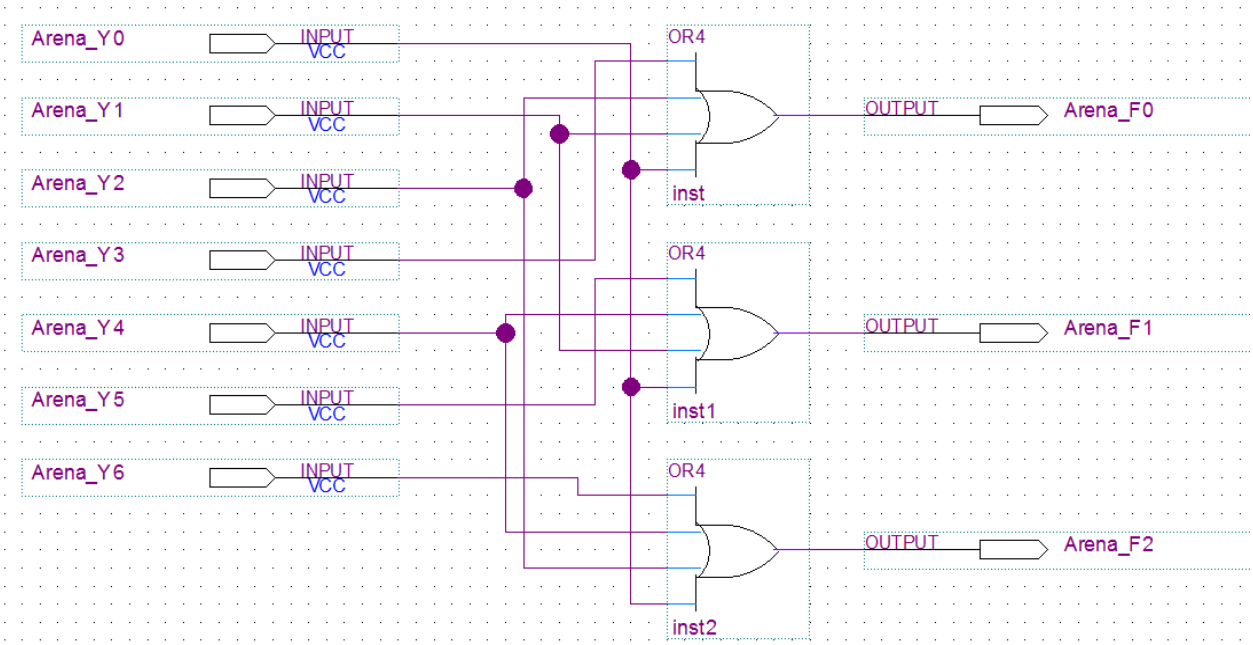



*Figure 16: 8to3 Encoder at Gate Level*

Below in figure **17** is the VHDL code I created for the 8to3 Encoder.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 1 - Spring 2019 Due: 2/20/19
-- Arena_3to8Decoder.vhd
library ieee;
use ieee.std_logic_1164.all;

entity Arena_8to3Encoder is -- Entity for Encoder
      port(
            Arena_Y0, Arena_Y1, Arena_Y2, Arena_Y3 : in std_logic; -- 8
inputs
            Arena_Y4, Arena_Y5, Arena_Y6, Arena_Y7 : in std_logic;
            Arena_F0, Arena_F1, Arena_F2 : out std_logic -- 3 outputs
            );
end Arena_8to3Encoder; -- End of entity

architecture Arena_Arch_8to3Encoder of Arena_8to3Encoder is --Architecture
to describe functionality
begin
      Arena_F0 <= (Arena_Y3 or Arena_Y2 or Arena_Y1 or Arena_Y0); -- F0 =
Y3+Y2+Y1+Y0
      Arena_F1 <= (Arena_Y5 or Arena_Y4 or Arena_Y1 or Arena_Y0); -- F1 =
Y5+Y4+Y1+Y0
      Arena_F2 <= (Arena_Y6 or Arena_Y4 or Arena_Y2 or Arena_Y0); -- F2 =
Y6+Y4+Y2+Y0
end Arena_Arch_8to3Encoder;    -- End of architecture
```

*Figure 57: 8to3 Encoder VHDL code*

## Section 3) Simulations

### 2to1 Multiplexer

The first simulation will be done for the 2to1 Multiplexer. Figure **18** below is shows the testbench code for the multiplexer.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 1 - Spring 2019 Due: 2/20/19
-- Arena_mux2to1_tb.vhd

library ieee;
use ieee.std_logic_1164.all;

Entity Arena_mux2to1_tb is
end Arena_mux2to1_tb;

Architecture Arena_Arch_mux2to1_tb of Arena_mux2to1_tb is
      signal Arena_X, Arena_Y, Arena_S : std_logic;
      signal Arena_M : std_logic;

    -- declare record type
    type test_vector is record
        Arena_X, Arena_Y, Arena_S : std_logic;
        Arena_M : std_logic;
    end record;

    type test_vector_array is array (natural range <>) of test_vector;
    constant test_vectors : test_vector_array := (
        -- Arena_X, Arena_Y, Arena_S, Arena_M   -- positional method is used
below
        ('0', '0', '0', '0'), -- or (X => '0', Y => '0', S => '0', M => '0')
        ('0', '0', '1', '0'),
        ('0', '1', '0', '0'),
        ('0', '1', '1', '1'),
        ('1', '0', '0', '1'),
        ('1', '0', '1', '0'),
        ('1', '1', '0', '1'),
        ('1', '1', '1', '1')

        );

begin
      UUT: entity work.Arena_mux2to1 port map (Arena_X => Arena_X, Arena_Y =>
Arena_Y, Arena_S => Arena_S, Arena_M => Arena_M);

            tb1: process
            begin
                  for i in test_vectors'range loop
            Arena_X <= test_vectors(i).Arena_X;  -- signal a = i^th-row-value
of test_vector's a
            Arena_Y <= test_vectors(i).Arena_Y; -- row left to right
```

```
                    Arena_S <= test_vectors(i).Arena_S;
                    Arena_M <= test_vectors(i).Arena_M;
            wait for 20 ns;

            assert (
                    (Arena_X = test_vectors(i).Arena_X) and
                    (Arena_Y = test_vectors(i).Arena_Y) and
                    (Arena_S = test_vectors(i).Arena_S) and
                    (Arena_M = test_vectors(i).Arena_M)
                )

            -- image is used for string-representation of integer etc.
            report  "test_vector " & integer'image(i) & " failed " &  --
T'image(x) is a string represesntation of x of type T
                    " for input Arena_X = " & std_logic'image(Arena_X) &
                    " and Arena_Y = " & std_logic'image(Arena_Y) &
                    " and Arena_S = " & std_logic'image(Arena_S) &
                    " for output Arena_M = " & std_logic'image(Arena_M)
                    severity error;
        end loop;
        wait;
    end process;
end Arena_Arch_mux2to1_tb;
```

*Figure 68: 2to1 Mux VHDL TB Code*

Our results should correspond with the truth table in Table **1**. Below in Figure **19** is the results from the Waveform file.

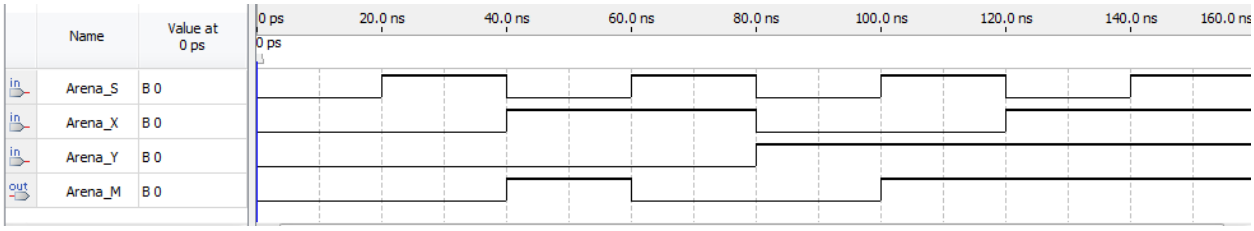



*Figure 19: 2to1 Mux Waveform*

Below in Figure **20** is the results from the testbench file.

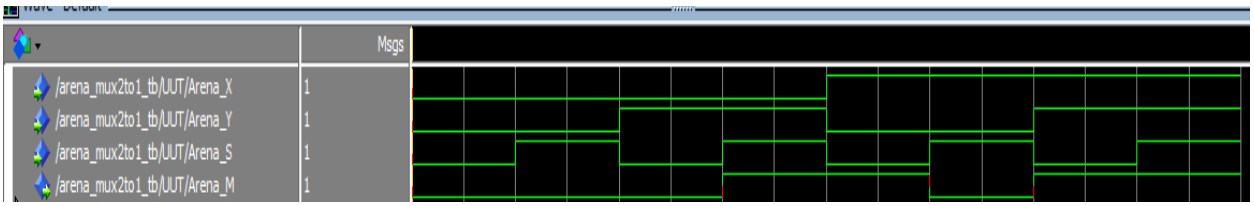



*Figure 20: 2to1 Mux Testbench*

Below in figure **21** is the output from the console for the testbench.

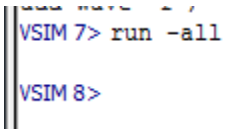



*Figure 7: 2to1 Mux Testbench Console Output*

Looking at the figures above, we can see our design for the 2to1 Multiplexer is correct. The easiest way to see this is from the output from the console. After running the whole testbench, it finished with no output. We know from the assert statement at the top of the page will only print an error message if the equality in the assert statement was found to be false. Another way to see it's correct is comparing the Waveform and the Testbench to the truth table. We know whenever Arena_S = 0, the output Arena_M = X. Looking at 000 for example, we see Arena_X=0, and Arena_S=0, so the output Arena_M = 0. This can be seen for all those cases. For Arena_S = 1, we see the output Arena_M = Y, which is what our truth table states, proving our design is correct.

## 1-bit Half Adder

The second simulation will be done for the 1-bit Half Adder. Figure **22** below is shows the testbench code for the half adder.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 1 - Spring 2019 Due: 2/20/19
-- Arena_HalfAdder_tb.vhd

library ieee;
use ieee.std_logic_1164.all;

Entity Arena_HalfAdder_tb is
end Arena_HalfAdder_tb;

Architecture Arena_Arch_HalfAdder_tb of Arena_HalfAdder_tb is -- Describiing
functionality
```

```
	signal Arena_X, Arena_Y : std_logic;
	signal Arena_CarryOut, Arena_Sum : std_logic;
begin
	-- connecting testbench signals with Arena_HalfAdder.vhd
	UUT : entity work.Arena_HalfAdder port map (Arena_X => Arena_X, Arena_Y
=> Arena_Y, Arena_CarryOut => Arena_CarryOut, Arena_Sum => Arena_Sum);

	tb1: process
	constant period: time := 40ns;
	begin
		Arena_X <= '0';
		Arena_Y <= '0';
		wait for period;
		assert((Arena_Sum ='0') and (Arena_CarryOut = '0')) -- expected
output
		-- error reported below if the sum or carry is not 0
		report "Test failed for input combination 00" severity error;

		Arena_X <= '0';
		Arena_Y <= '1';
		wait for period;
		assert((Arena_Sum ='1') and (Arena_CarryOut = '0')) -- expected
output
		-- error reported below if the sum or carry is not 0
		report "Test failed for input combination 01" severity error;

		Arena_X <= '1';
		Arena_Y <= '0';
		wait for period;
		assert((Arena_Sum ='1') and (Arena_CarryOut = '0')) -- expected
output
		-- error reported below if the sum or carry is not 0
		report "Test failed for input combination 10" severity error;

		Arena_X <= '1';
		Arena_Y <= '1';
		wait for period;
		assert((Arena_Sum ='0') and (Arena_CarryOut = '1')) -- expected
output
		-- error reported below if the sum or carry is not 0
		report "Test failed for input combination 11" severity error;

		wait;
	end process;
end Arena_Arch_HalfAdder_tb;
```

*Figure 228: 1-bit Half Adder VHDL TB Code*

Our results should correspond with the truth table in Table **2**. Below in Figure **23** is the results from the Waveform file.

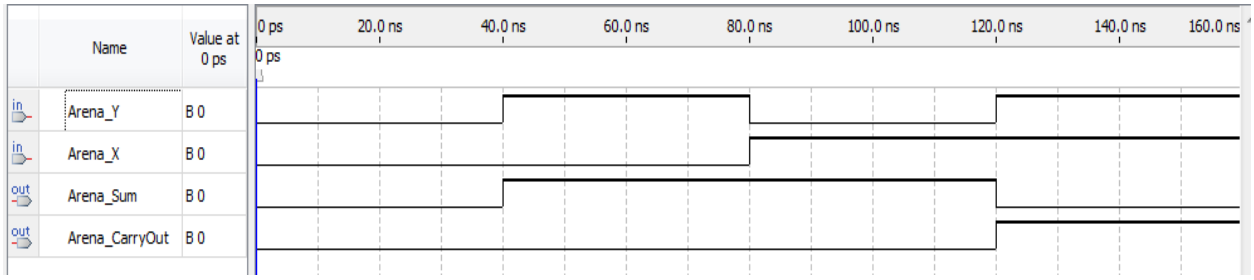



*Figure 23: 1-bit Half Adder Waveform*

Below in Figure **24** is the results from the testbench file. In this testbench, it actually compares the actual half adder VHDL file to the testbench results. The testbench are the last four waves, denoted with a UUT on the left side, which stands for **Unit Under Test.**

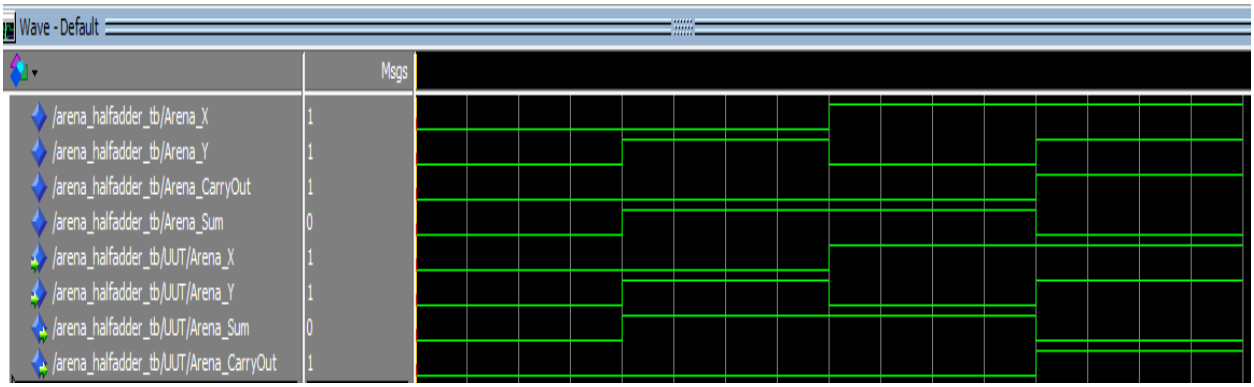



*Figure 24: 1-bit Half Adder Testbench*

Below in figure **25** is the output from the console for the testbench.

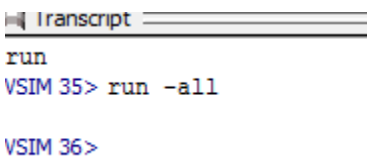



*Figure25: 1-bit Half Adder Console Output*

Looking at the figures above, we can see our design for 1-bit Half Adder is correct. The easiest way to see this is from the output from the console. After running the whole testbench, it finished with no output. We know from the various assert statements in figure **25** will only print an error message if the equality in the assert statement was found to be false. Another way to see it's correct is comparing the Waveform and the Testbench to the truth table. We can see in the figures whenever Arena_X != Arena_Y, then Arena_Sum = 1, otherwise 0, and when Arena_X=Arena_Y, Arena_CarryOut = 1, otherwise 0, which is what our truth table states, proving our design is correct.

## 1-bit Full Adder

The third simulation will be done for the 1-bit Full Adder. Figure **26** below is shows the testbench code for the multiplexer.

```
-- Arena_FullAdder_tb.vhd

library ieee;
use ieee.std_logic_1164.all;

Entity Arena_FullAdder_tb is
end Arena_FullAdder_tb;

Architecture Arena_Arch_FullAdder_tb of Arena_FullAdder_tb is
      signal Arena_X, Arena_Y, Arena_CarryIn : std_logic;
      signal Arena_CarryOut, Arena_Sum : std_logic;

      type test_vector is record
            Arena_X, Arena_Y, Arena_CarryIn : std_logic;
            Arena_CarryOut, Arena_Sum : std_logic;
      end record;

      type test_vector_array is array (natural range <> ) of test_vector;
      constant test_vectors : test_vector_array := (
            -- Arena_X, Arena_Y, Arena_CarryIn, Arena_CarryOut, Arena_Sum
            ('0', '0', '0', '0', '0'), -- X=0,Y=0,Ci=0, Co=0,S=0
            ('0', '0', '1', '0', '1'),
            ('0', '1', '0', '0', '1'),
            ('0', '1', '1', '1', '0'),
            ('1', '0', '0', '0', '1'),
```

```
            ('1', '0', '1', '1', '0'),
            ('1', '1', '0', '1', '0'),
            ('1', '1', '1', '1', '1'),

            ('0', '0', '1', '1', '0') -- fail test
            );

begin
      -- connecting testbench signals with Arena_HalfAdder.vhd
      UUT : entity work.Arena_FullAdder port map (Arena_X1 => Arena_X,
Arena_Y1 => Arena_Y, Arena_CarryIn1 => Arena_CarryIn, Arena_CarryOut =>
Arena_CarryOut, Arena_Sum => Arena_Sum);

      tb1: process
      constant period: time := 40ns;
      begin
            for i in test_vectors'range loop
                  Arena_X <= test_vectors(i).Arena_X; -- signal a = i^th-row-
value of test_vector's a
                  Arena_Y <= test_vectors(i).Arena_Y;
                  Arena_CarryIn <= test_vectors(i).Arena_CarryIn;

                  wait for period;

                  assert(
                        (Arena_CarryOut = test_vectors(i).Arena_CarryOut) and
                        (Arena_Sum = test_vectors(i).Arena_Sum)
                        )

            report"test_vector " & integer'image(i) & " failed " &
            " for input Arena_X = " & std_logic'image(Arena_X) &
                  " and Arena_Y = " & std_logic'image(Arena_Y) &
                  " and Arena_CarryIn = " & std_logic'image(Arena_CarryIn) &
            " For output Arena_CarryOut = " & std_logic'image(Arena_CarryOut)
&                 " and output Arena_Sum = " & std_logic'image(Arena_Sum)
                              severity error;
            end loop;
            wait;
      end process;
end Arena_Arch_FullAdder_tb;
```

*Figure 269: 1-bit Full Adder VHDL TB Code*

The interesting thing about this testbench file is it uses an array of numbers and a for loop to go through them. There are 9 tests in the array for the for loop to go through. If there was no for loop, there would be 9 assert and report statements, which is very messy and confusing.

Our results should correspond with the truth table in Table **3**. Below in Figure **27**is the results from the Waveform file.

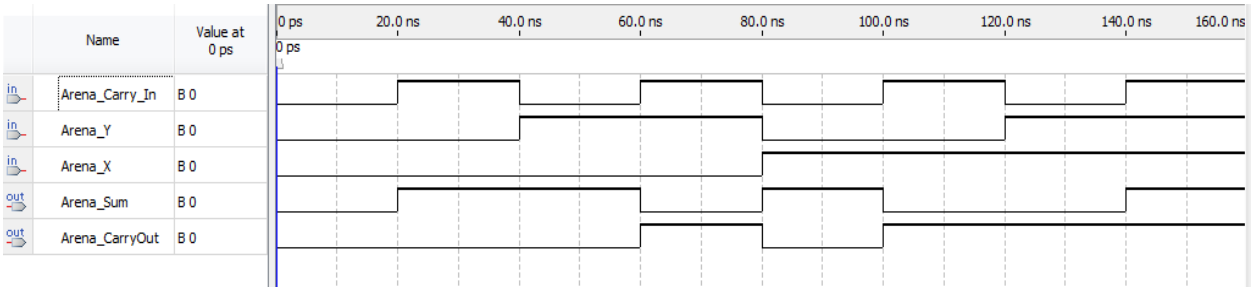



*Figure 27: 1-bit Full Adder Waveform*

Below in Figure **28** is the results from the testbench file. In this testbench, similar to the last one, it compares the actual full adder VHDL file to the testbench results. The testbench are the last five waves, denoted with a UUT on the left side, which stands for **Unit Under Test.**

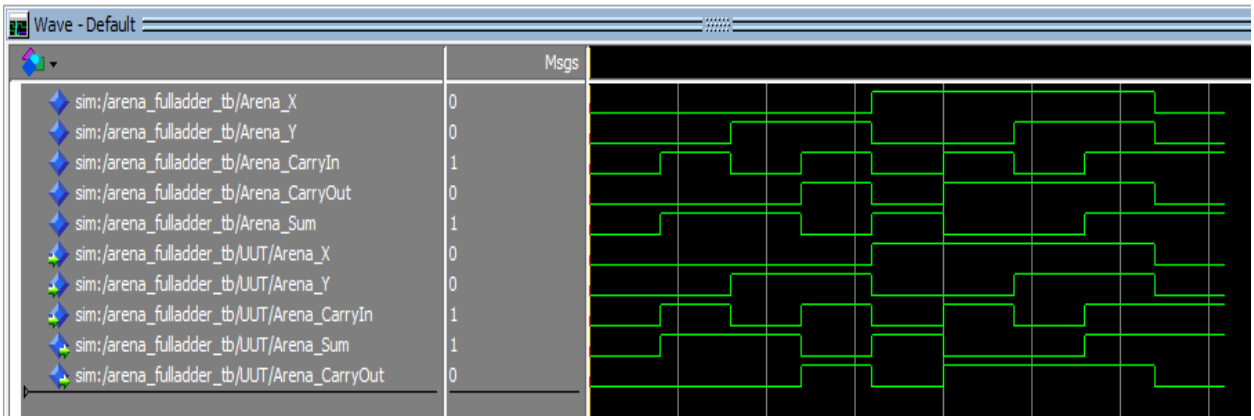



*Figure 2810: 1-bit Half Adder Testbench*

Below in figure **29** is the output from the console for the testbench.

```
VSIM 7> run -all
# ** Error: test_vector 8 failed  for input Arena_X = '0' and Arena_Y = '0' and Arena_CarryIn = '1' For output Arena_CarryOut = '0' and output Arena_Sum = '1'
#    Time: 360 ns  Iteration: 0  Instance: /arena_fulladder_tb

VSIM 8>
```

*Figure 29: 1-bit Full Adder Console Output*

Looking at the figures above, we can see our design for 1-bit Full Adder is correct. The easiest way to see this is from the output from the console. After running the whole testbench, it finished with 1 error message. This is different compared to our last two simulations, but is this a problem? Let's take a look at the message. It says for Inputs **X=0, Y=0, CarryIn=1 and outputs CarryOut=0 and Sum = 1, vector(8) failed.** Looking back at the code, vector(8) is as follows:

```
      ('0', '0', '1', '1', '0') -- fail test for verification
```

This has **X=0, Y=0, Ci=1, Co=1, S=0.** Notice that this actually doesn't make sense. Looking back on page 9, if a 0 and 1 is added, the resulting sum should be 1 with a carry of 0. Here it says a carry of 1 wit ha sum of 0, which is incorrect. So if the assert statement failed on an incorrect test, that means the Full Adder actually behaved as it should! Also, seeing this is the only fail test verification for this testbench, and there was no other output to the console, the testbench passed. Another way to see it's correct is comparing the Waveform and the Testbench to the truth table. We can see in the figures that the results compared to the truth table are correct, thus confirming our design is correct.

## 3to8 Decoder

The fourth simulation will be done for the 3to8Decoder. Figure **30** below is shows the testbench code for the multiplexer.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 1 - Spring 2019 Due: 2/20/19
-- Arena_3to8Decoder_tb.vhd

library ieee;
use ieee.std_logic_1164.all;

Entity Arena_3to8Decoder_tb is
end Arena_3to8Decoder_tb;

Architecture Arena_Arch_3to8Decoder_tb of Arena_3to8Decoder_tb is
      signal Arena_A, Arena_B, Arena_C : std_logic; -- Defining signals
```

```
	signal Arena_F0, Arena_F1, Arena_F2 : std_logic;
	signal Arena_F3, Arena_F4, Arena_F5 : std_logic;
	signal Arena_F6, Arena_F7 : std_logic;

	type test_vector is record -- collection of signals in one object, like
C structures
		Arena_A, Arena_B, Arena_C : std_logic;
		Arena_F0, Arena_F1, Arena_F2 : std_logic;
		Arena_F3, Arena_F4, Arena_F5 : std_logic;
		Arena_F6, Arena_F7 : std_logic;
	end record;

	type test_vector_array is array (natural range <> ) of test_vector; --
Array of test_vector
	constant test_vectors : test_vector_array := (
		-- Arena_A, Arena_B, Arena_C, Arena_F0, Arena_F1, Arena_F2,
Arena_F3, Arena_F4, Arena_F5, Arena_F6, Arena_F7,
		('0', '0', '0', '1', '0', '0', '0', '0', '0', '0', '0'), --
A=0,B=0,C=0,F0=0...F7=0
		('0', '0', '1', '0', '1', '0', '0', '0', '0', '0', '0'),
		('0', '1', '0', '0', '0', '1', '0', '0', '0', '0', '0'),
		('0', '1', '1', '0', '0', '0', '1', '0', '0', '0', '0'),
		('1', '0', '0', '0', '0', '0', '0', '1', '0', '0', '0'),
		('1', '0', '1', '0', '0', '0', '0', '0', '1', '0', '0'),
		('1', '1', '0', '0', '0', '0', '0', '0', '0', '1', '0'),
		('1', '1', '1', '0', '0', '0', '0', '0', '0', '0', '1')

		);

begin
	-- connecting testbench signals with Arena_3to8Decoder.vhd
	UUT : entity work.Arena_3to8Decoder port map (Arena_A => Arena_A,
Arena_B => Arena_B, Arena_C => Arena_C, Arena_F0 => Arena_F0, Arena_F1 =>
Arena_F1,
			Arena_F2 => Arena_F2, Arena_F3 => Arena_F3, Arena_F4 =>
Arena_F4, Arena_F5 => Arena_F5, Arena_F6 => Arena_F6, Arena_F7 => Arena_F7);

	tb1: process
	constant period: time := 40ns;
	begin
		for i in test_vectors'range loop
			Arena_A <= test_vectors(i).Arena_A; -- signal a = i^th-row-
value of test_vector's a
			Arena_B <= test_vectors(i).Arena_B; -- row is left to right
			Arena_C <= test_vectors(i).Arena_C;
			Arena_F0 <= test_vectors(i).Arena_F0;
			Arena_F1 <= test_vectors(i).Arena_F1;
			Arena_F2 <= test_vectors(i).Arena_F2;
			Arena_F3 <= test_vectors(i).Arena_F3;
			Arena_F4 <= test_vectors(i).Arena_F4;
			Arena_F5 <= test_vectors(i).Arena_F5;
			Arena_F6 <= test_vectors(i).Arena_F6;
			Arena_F7 <= test_vectors(i).Arena_F7;

			wait for period;

			assert(
```

```
                                (Arena_A <= test_vectors(i).Arena_A) and
                                (Arena_B <= test_vectors(i).Arena_B) and
                                (Arena_C <= test_vectors(i).Arena_C) and
                                (Arena_F0 <= test_vectors(i).Arena_F0) and
                                (Arena_F1 <= test_vectors(i).Arena_F1) and
                                (Arena_F2 <= test_vectors(i).Arena_F2) and
                                (Arena_F3 <= test_vectors(i).Arena_F3) and
                                (Arena_F4 <= test_vectors(i).Arena_F4) and
                                (Arena_F5 <= test_vectors(i).Arena_F5) and
                                (Arena_F6 <= test_vectors(i).Arena_F6) and
                                (Arena_F7 <= test_vectors(i).Arena_F7)
                                )

            report"test_vector " & integer'image(i) & " failed " &
                  " for input Arena_A = " & std_logic'image(Arena_A) &
                  " and Arena_B = " & std_logic'image(Arena_B) &
                  " and Arena_C = " & std_logic'image(Arena_C) &
                  " For output Arena_F0 = " & std_logic'image(Arena_F0) &
                  " and output Arena_F1 = " & std_logic'image(Arena_F1) &
                  " and output Arena_F2 = " & std_logic'image(Arena_F2) &
                  " and output Arena_F3 = " & std_logic'image(Arena_F3) &
                  " and output Arena_F4 = " & std_logic'image(Arena_F4) &
                  " and output Arena_F5 = " & std_logic'image(Arena_F5) &
                  " and output Arena_F6 = " & std_logic'image(Arena_F6) &
                  " and output Arena_F7 = " & std_logic'image(Arena_F7)
            severity error;
            end loop;
            wait;
      end process;
end Arena_Arch_3to8Decoder_tb;
```

*Figure 11: 3to8 Decoder VHDL TB Code*

As like the testbench file for the full adder, this testbench file uses an array of numbers and a for loop to go through them. There are 8 tests in the array for the for loop to go through. If there was no for loop, there would be 8 assert and report statements, which is very messy and confusing.

The results should correspond with the truth table in Table **4**. Below in Figure **31** is the results from the Waveform file.

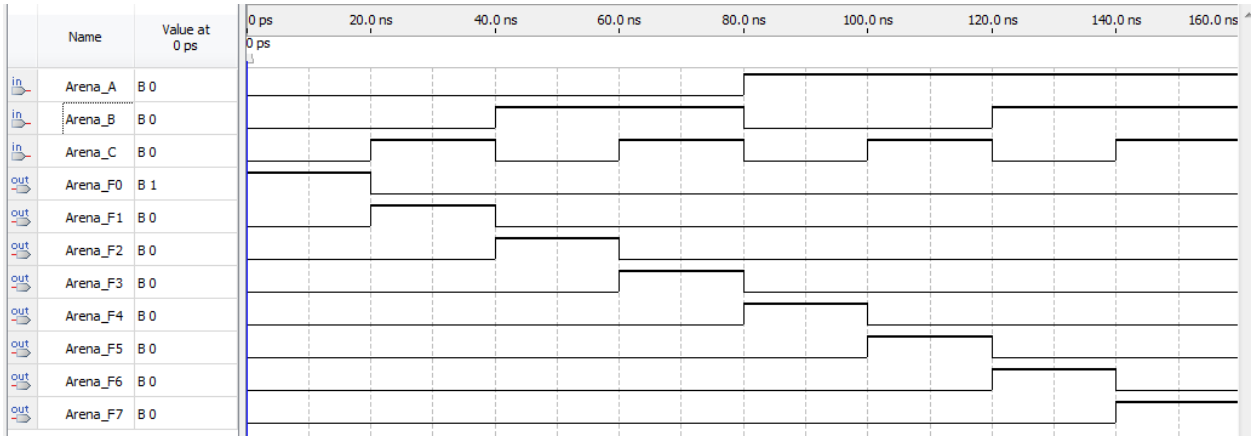



*Figure 31: 3to8 Decoder Waveform*

Below in Figure **32** is the results from the testbench file. In this testbench, unlike the last two, I did not include the comparison to the actual 3to8 decoder file, the reason being is the waveforms were too small to be seen properly. In doing so, I left the ones denoted with a UUT on the left side, which stands for **Unit Under Test,** which is the testbench testing the 3to8 decoder.

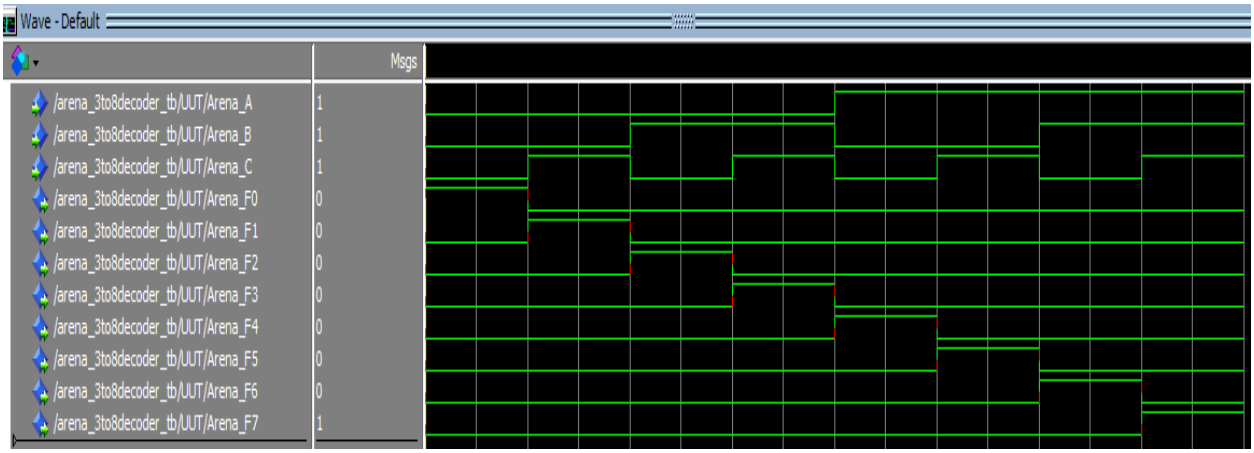



*Figure 32: 1-bit Half Adder Testbench*

Below in figure **33** is the output from the console for the testbench.

```
VSIM 11> run -all

VSIM 12>
```

*Figure 33: 3to8 Decoder Console Output*

Looking at the figures above, we can see the design for 3to8 Decoder is correct. The easiest way to see this is from the output from the console. After running the whole testbench, it finished with no error message. In this test, there was no fail test included, but can be added if wanted. With that said, there was no output to the console, the testbench passed. Another way to see it's correct is comparing the Waveform and the Testbench to the truth table. We can see in the figures that the results compared to the truth table are correct, thus confirming our design is correct.

## 8to3 Encoder

The fifth and final simulation will be done for the 8to3 Encoder. Figure **34** below is shows the testbench code for the 8to3 Encoder.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 1 - Spring 2019 Due: 2/20/19
-- Arena_8to3Encoder_tb.vhd

library ieee;
use ieee.std_logic_1164.all;

Entity Arena_8to3Encoder_tb is
end Arena_8to3Encoder_tb;

Architecture Arena_Arch_8to3Encoder_tb of Arena_8to3Encoder_tb is
      signal Arena_Y0, Arena_Y1, Arena_Y2 : std_logic; -- define signals
      signal Arena_Y3, Arena_Y4, Arena_Y5 : std_logic;
      signal Arena_Y6, Arena_Y7 : std_logic;
      signal Arena_F0, Arena_F1, Arena_F2 : std_logic;


      type test_vector is record -- collection of signals in one object, like
C structures
```

```
            Arena_Y0, Arena_Y1, Arena_Y2 : std_logic;
            Arena_Y3, Arena_Y4, Arena_Y5 : std_logic;
            Arena_Y6, Arena_Y7 : std_logic;
            Arena_F0, Arena_F1, Arena_F2 : std_logic;
      end record;

      type test_vector_array is array (natural range <> ) of test_vector; --
Array of test vector
      constant test_vectors : test_vector_array := (
            -- Arena_Y0, Arena_Y1, Arena_Y2, Arena_Y3, Arena_Y4, Arena_Y5,
Arena_Y6, Arena_Y7, Arena_Y0, Arena_Y1, Arena_Y2,
            ('0', '0', '0', '0', '0', '0', '0', '1', '0', '0', '0'), --
Y0=0...Y7=0,F0=0,F1=0,F2=0
            ('0', '0', '0', '0', '0', '0', '1', '0', '0', '0', '1'),
            ('0', '0', '0', '0', '0', '1', '0', '0', '0', '1', '0'),
            ('0', '0', '0', '0', '1', '0', '0', '0', '0', '1', '1'),
            ('0', '0', '0', '1', '0', '0', '0', '0', '1', '0', '0'),
            ('0', '0', '1', '0', '0', '0', '0', '0', '1', '0', '1'),
            ('0', '1', '0', '0', '0', '0', '0', '0', '1', '1', '0'),
            ('1', '0', '0', '0', '0', '0', '0', '0', '1', '1', '1')

            --('0', '0', '1', '1', '0', '0', '0', '0', '0', '0', '1') -- fail
test
            );

begin
      -- connecting testbench signals with Arena_8to3Encoder.vhd
      UUT : entity work.Arena_8to3Encoder port map (Arena_F0 => Arena_F0,
Arena_F1 => Arena_F1, Arena_F2 => Arena_F2, Arena_Y0 => Arena_Y0, Arena_Y1 =>
Arena_Y1,
                  Arena_Y2 => Arena_Y2, Arena_Y3 => Arena_Y3, Arena_Y4 =>
Arena_Y4, Arena_Y5 => Arena_Y5, Arena_Y6 => Arena_Y6, Arena_Y7 => Arena_Y7);

      tb1: process
      constant period: time := 40ns;
      begin
            for i in test_vectors'range loop
                  Arena_Y0 <= test_vectors(i).Arena_Y0; -- signal a = i^th-
row-value of test_vector's a
                  Arena_Y1 <= test_vectors(i).Arena_Y1; -- row is left to
right
                  Arena_Y2 <= test_vectors(i).Arena_Y2;
                  Arena_Y3 <= test_vectors(i).Arena_Y3;
                  Arena_Y4 <= test_vectors(i).Arena_Y4;
                  Arena_Y5 <= test_vectors(i).Arena_Y5;
                  Arena_Y6 <= test_vectors(i).Arena_Y6;
                  Arena_Y7 <= test_vectors(i).Arena_Y7;
                  Arena_F0 <= test_vectors(i).Arena_F0;
                  Arena_F1 <= test_vectors(i).Arena_F1;
                  Arena_F2 <= test_vectors(i).Arena_F2;
                  wait for period;

                  assert(
                              (Arena_Y0 <= test_vectors(i).Arena_Y0) and
                              (Arena_Y1 <= test_vectors(i).Arena_Y1) and
                              (Arena_Y2 <= test_vectors(i).Arena_Y2) and
                              (Arena_Y3 <= test_vectors(i).Arena_Y3) and
```

```
                                (Arena_Y4 <= test_vectors(i).Arena_Y4) and
                                (Arena_Y5 <= test_vectors(i).Arena_Y5) and
                                (Arena_Y6 <= test_vectors(i).Arena_Y6) and
                                (Arena_Y7 <= test_vectors(i).Arena_Y7) and
                                (Arena_F0 <= test_vectors(i).Arena_F0) and
                                (Arena_F1 <= test_vectors(i).Arena_F1) and
                                (Arena_F2 <= test_vectors(i).Arena_F2)
                                )

                    report"test_vector " & integer'image(i) & " failed " & --
T'image(x) is a string represesntation of x of type T
                    " For input Arena_Y0 = " & std_logic'image(Arena_Y0) &
                    " and input Arena_Y1 = " & std_logic'image(Arena_Y1) &
                    " and input Arena_Y2 = " & std_logic'image(Arena_Y2) &
                    " and input Arena_Y3 = " & std_logic'image(Arena_Y3) &
                    " and input Arena_Y4 = " & std_logic'image(Arena_Y4) &
                    " and input Arena_Y5 = " & std_logic'image(Arena_Y5) &
                    " and input Arena_Y6 = " & std_logic'image(Arena_Y6) &
                    " and input Arena_Y7 = " & std_logic'image(Arena_Y7) &
                    " for output Arena_F0 = " & std_logic'image(Arena_F0) &
                    " and Arena_F1 = " & std_logic'image(Arena_F1) &
                    " and Arena_F2 = " & std_logic'image(Arena_F2)
                    severity error;
                end loop;
                wait;
        end process;
end Arena_Arch_8to3Encoder_tb;
```

*Figure 34: 8to3 Encoder VHDL TB Code*

As like the testbench file for the full adder and the decoder, this testbench file uses an array of numbers and a for loop to go through them. There are 8 tests in the array for the for loop to go through. If there was no for loop, there would be 8 assert and report statements, which is very messy and confusing.

The results should correspond with the truth table in Table **5**. Below in Figure **35** is the results from the Waveform file.

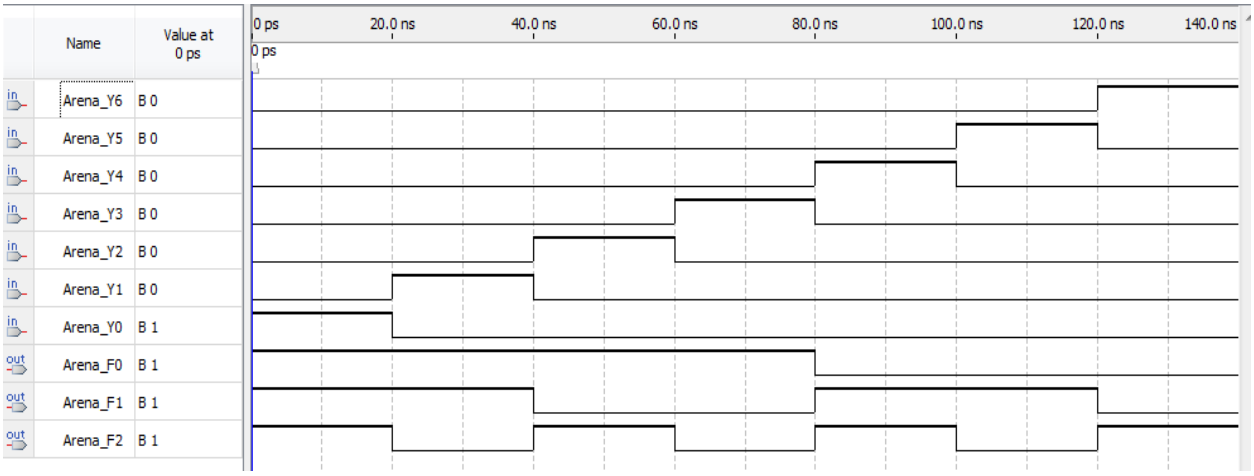



*Figure 3512: 8to3 Encoder Waveform*

Below in Figure **36** is the results from the testbench file. In this testbench, unlike the last three, I did not include the comparison to the actual 8to3 encoder file, the reason being is the waveforms were too small to be seen properly. In doing so, I left the ones denoted with a UUT on the left side, which stands for **Unit Under Test,** which is the testbench testing the 8to3 encoder.

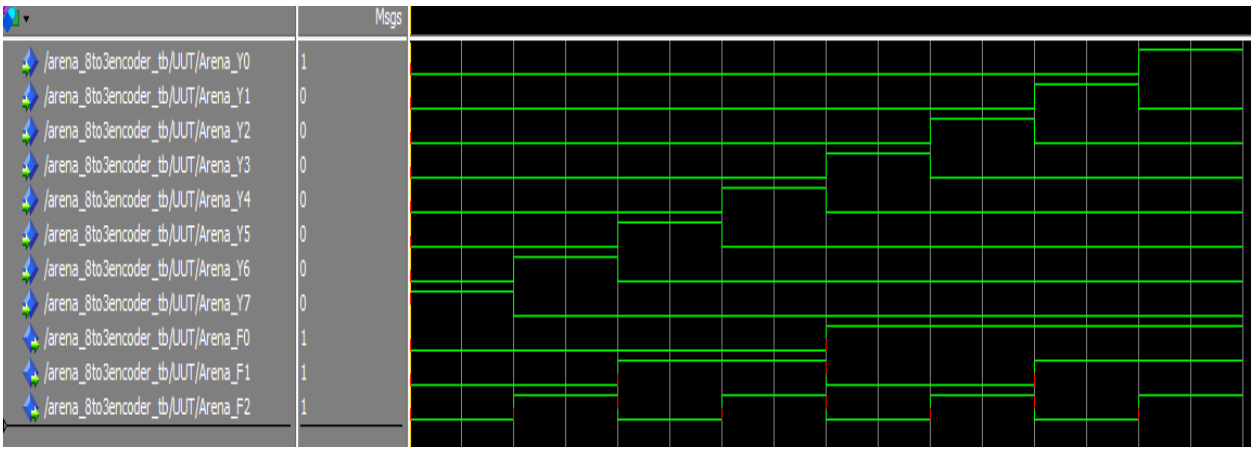



*Figure 13: 1-bit Half Adder Testbench*

Below in figure **37** is the output from the console for the testbench.

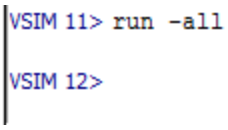



*Figure 14: 8to3 Encoder Console Output*

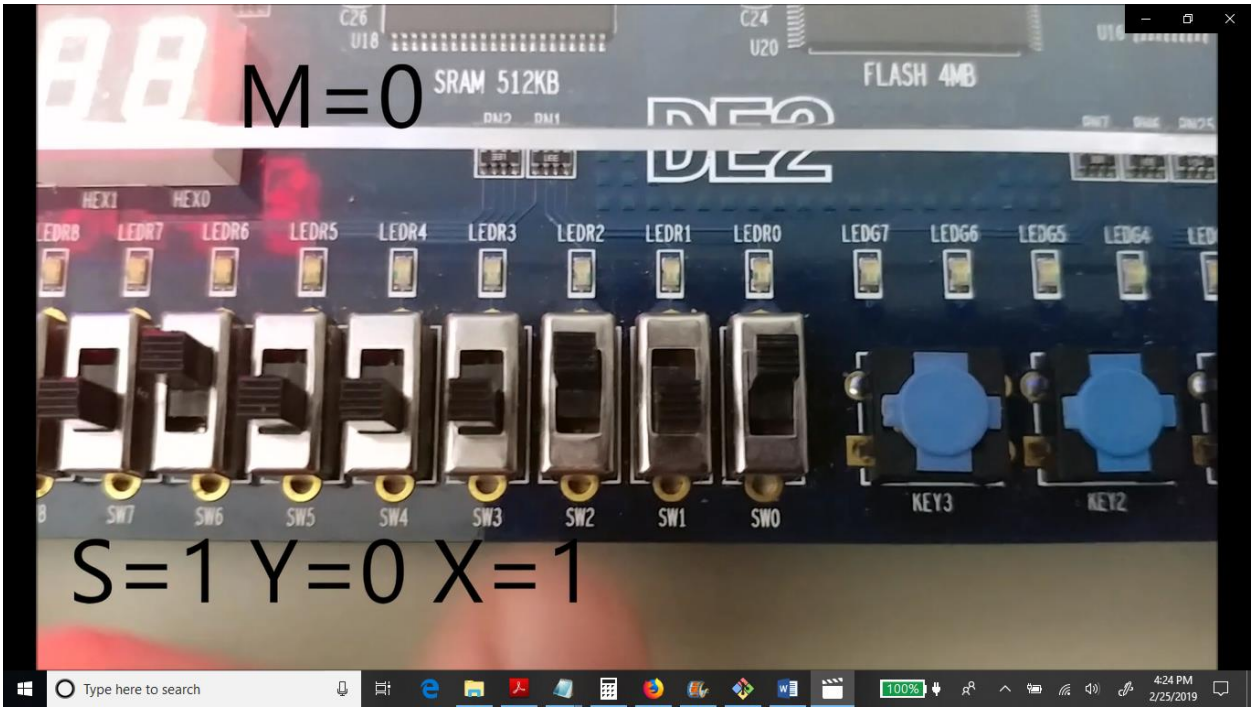
Looking at the figures above, we can see the design for 8to3 Encoder is correct. The easiest way to see this is from the output from the console. After running the whole testbench, it finished with no error message. In this test, there was no fail test included, but can be added if wanted. With that said, there was no output to the console, the testbench passed. Another way to see it's correct is comparing the Waveform and the Testbench to the truth table. We can see in the figures that the results compared to the truth table are correct, thus confirming our design is correct.

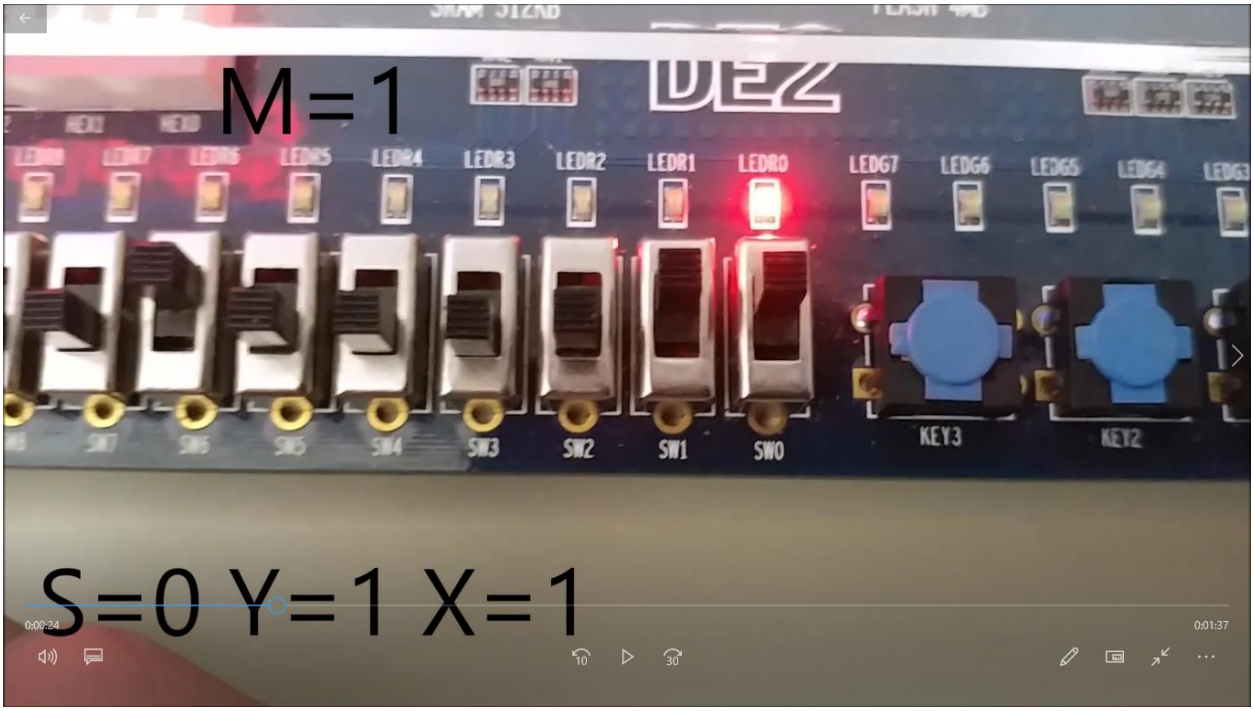
## Section 4) Demonstration Pictures

(Not every case is shown in the pictures, all cases shown in video)

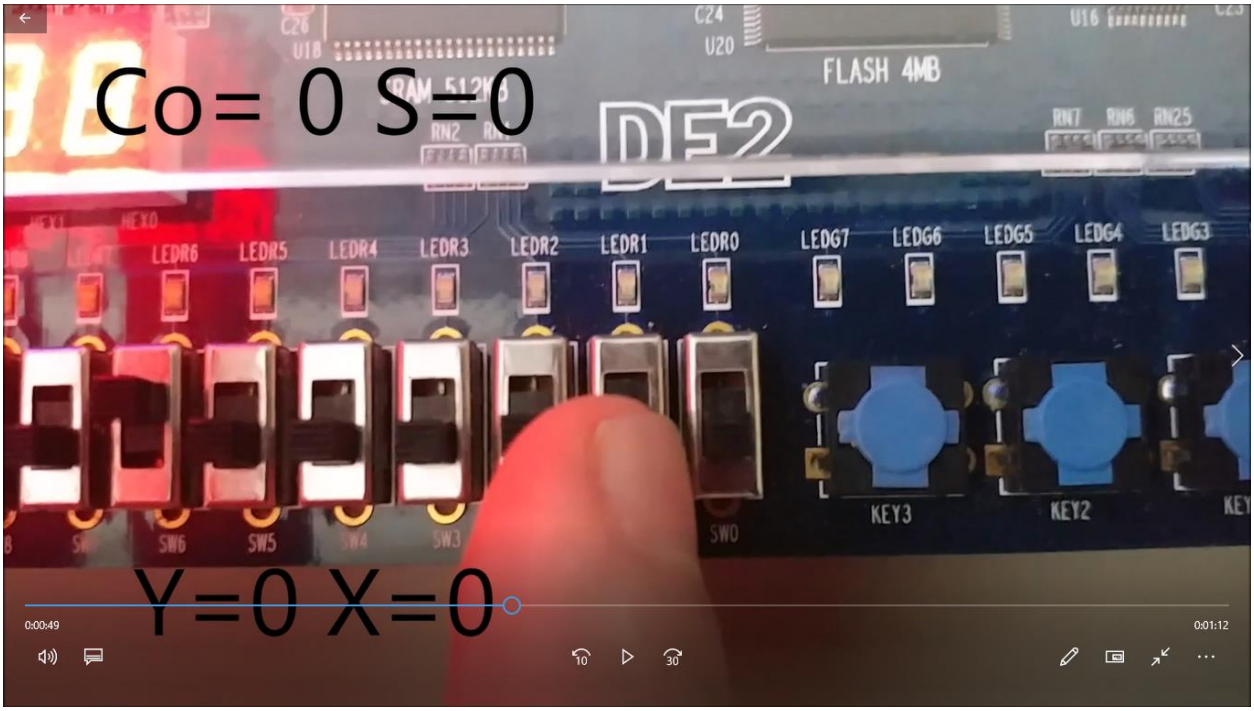
### 2to1Mux

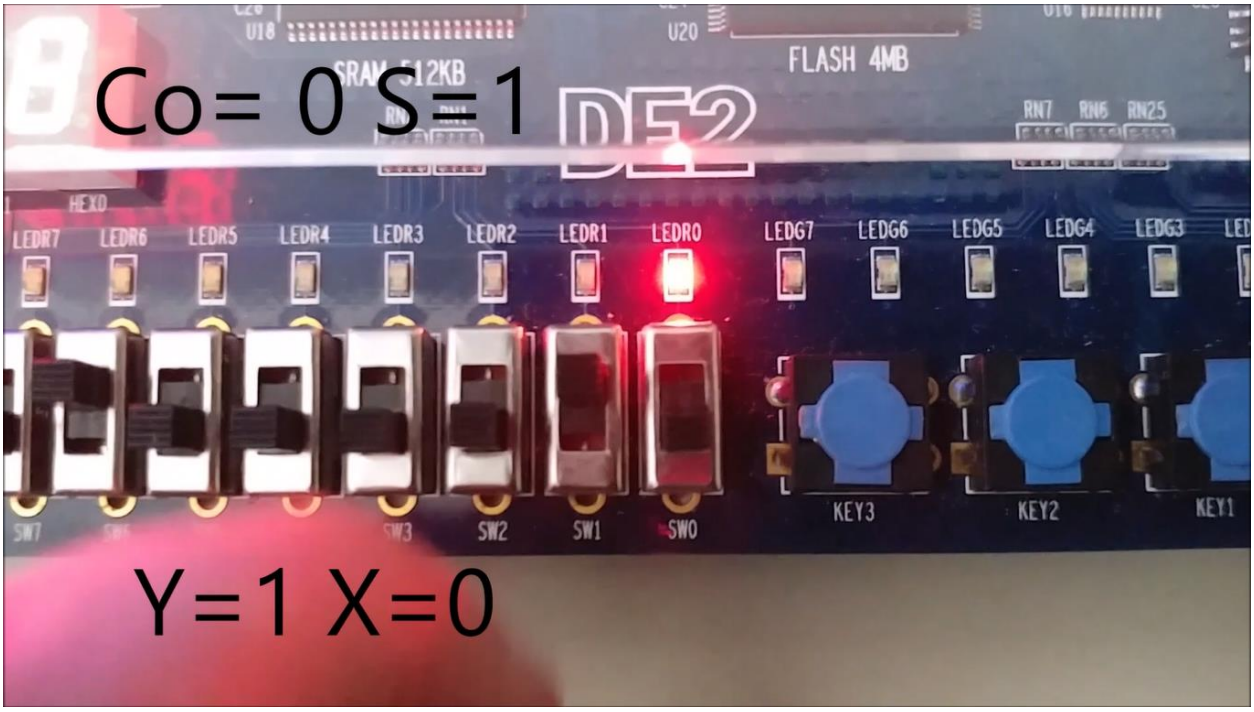

M=0
FLASH 4MB
LEDR6 LEDR5 LEDR4 LEDR3 LEDR2 LEDR1 LEDR0 LEDG7 LEDG6 LEDG5 LEDG4 LEDG3 LEDG2
KEY2
KEY1
S=0 Y=0 X=0

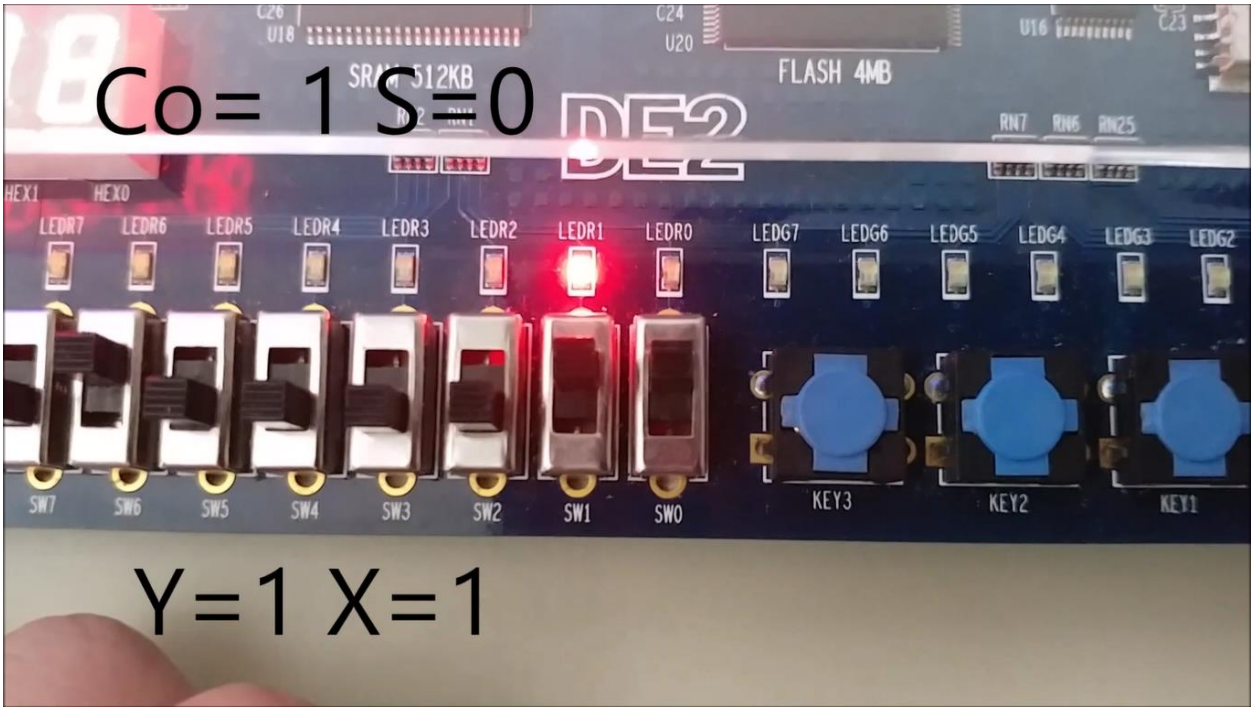

M=0
SRAM 512KB
FLASH 4MB
HEX1 HEX0
LEDR8 LEDR7 LEDR6 LEDR5 LEDR4 LEDR3 LEDR2 LEDR1 LEDR0 LEDG7 LEDG6 LEDG5 LEDG4
SW7 SW6 SW5 SW4 SW3 SW2 SW1 SW0
KEY3 KEY2
S=1 Y=0 X=1
Type here to search
100%
4:24 PM
2/25/2019


M=1
LEDR4
LEDR3
LEDR2
LEDR1
LEDR0
LEDG7
LEDG6
LEDG5
LEDG4
SW3
SW2
SW1
SW0
KEY3
KEY2
S=0 Y=1 X=1
0:00:24
0:01:37

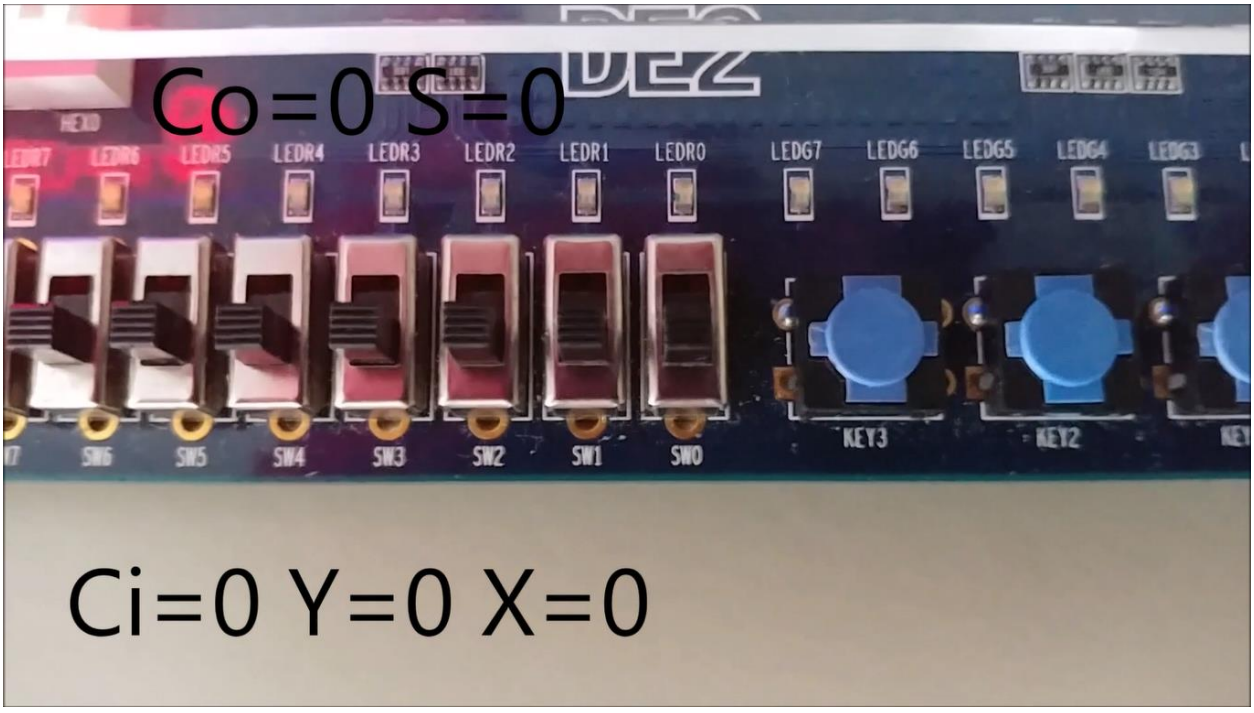
## Half Adder

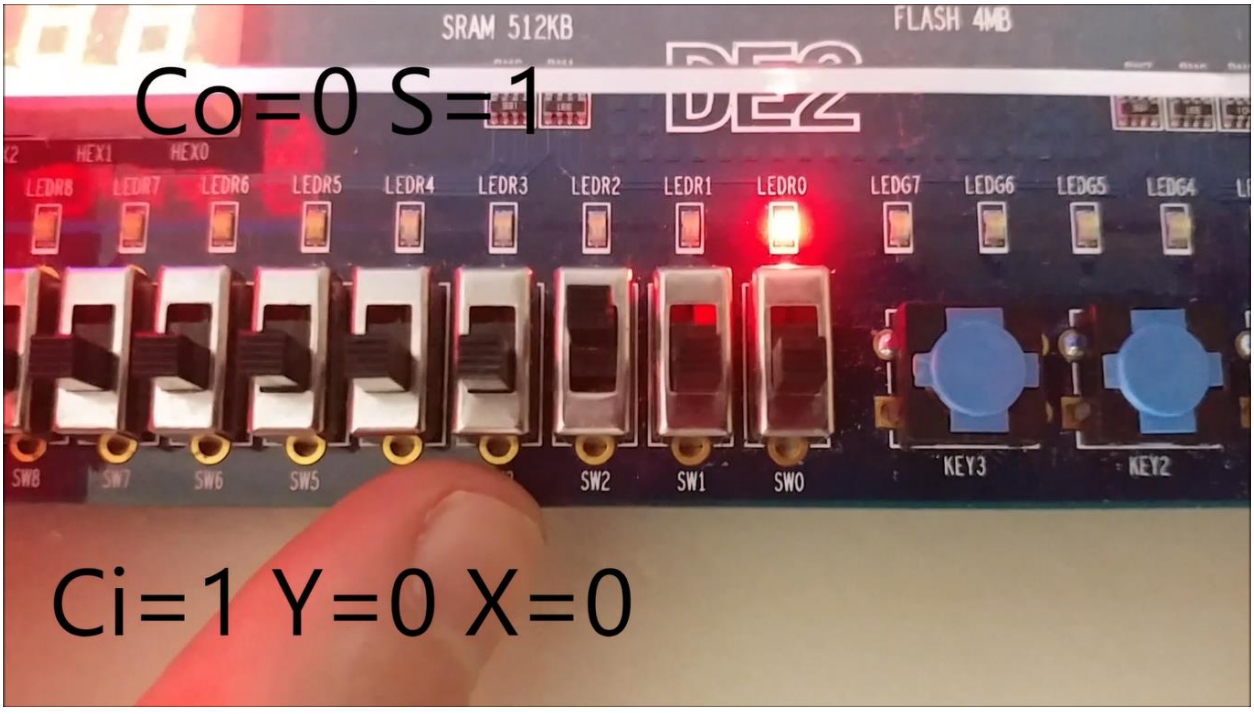

Co= 0 S=0
FLASH 4MB
DE2
LEDR6 LEDR5 LEDR4 LEDR3 LEDR2 LEDR1 LEDR0
LEDG7 LEDG6 LEDG5 LEDG4 LEDG3
KEY3 KEY2
SW6 SW5 SW0
Y=0 X=0
0:00:49
0:01:12

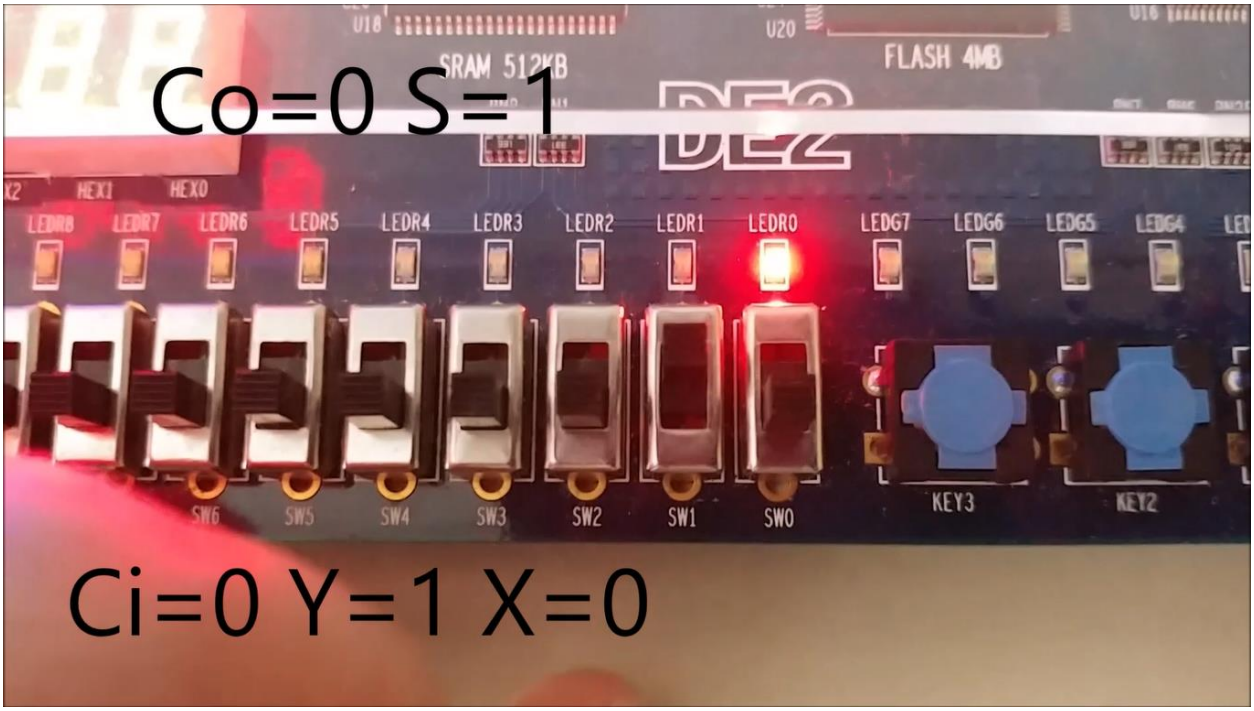

Co= 0 S=1
FLASH 4MB
DE2
HEX0
LEDR7 LEDR6 LEDR5 LEDR4 LEDR3 LEDR2 LEDR1 LEDR0
LEDG7 LEDG6 LEDG5 LEDG4 LEDG3
KEY3 KEY2 KEY1
SW3 SW2 SW1 SW0
Y=1 X=0

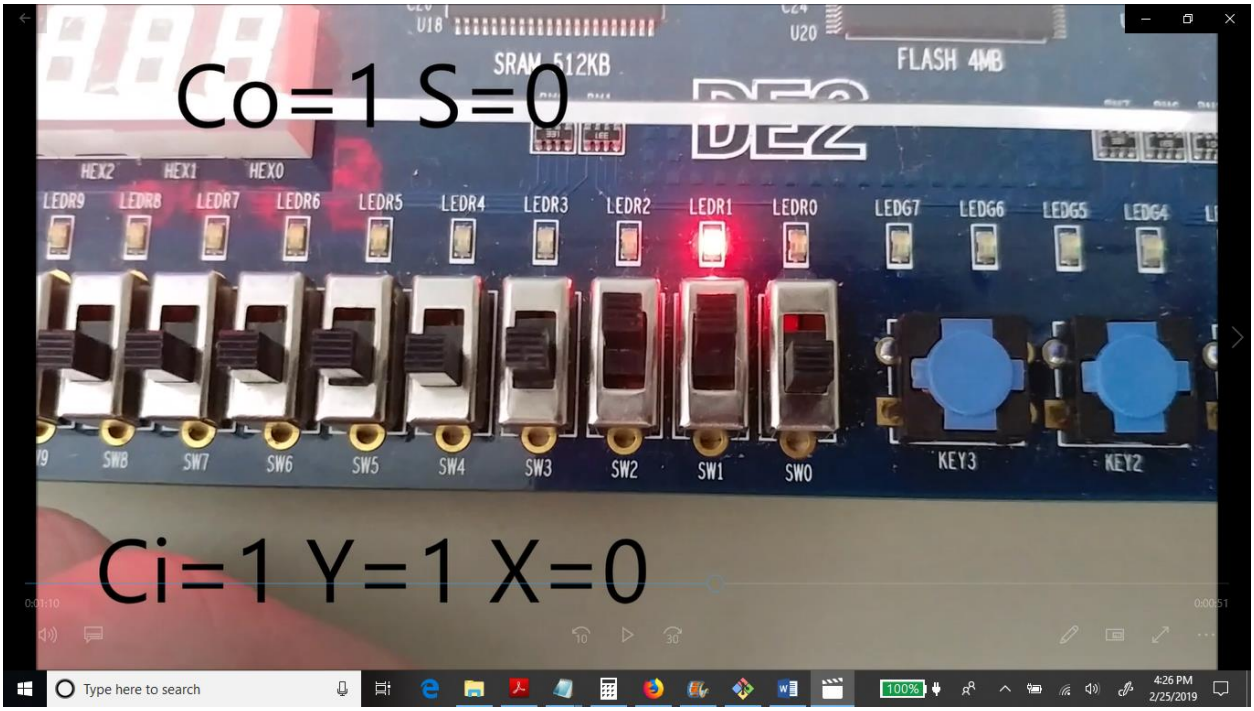

Co= 1 S=0
FLASH 4MB
LEDR7 LEDR6 LEDR5 LEDR4 LEDR3 LEDR2 LEDR1 LEDR0
LEDG7 LEDG6 LEDG5 LEDG4 LEDG3 LEDG2
SW7 SW6 SW5 SW4 SW3 SW2 SW1 SW0
KEY3 KEY2 KEY1
Y=1 X=1

**Full Adder**


Co=0 S=0
LEDR7 LEDR6 LEDR5 LEDR4 LEDR3 LEDR2 LEDR1 LEDR0
LEDG7 LEDG6 LEDG5 LEDG4 LEDG3
SW6 SW5 SW4 SW3 SW2 SW1 SW0
KEY3 KEY2
Ci=0 Y=0 X=0

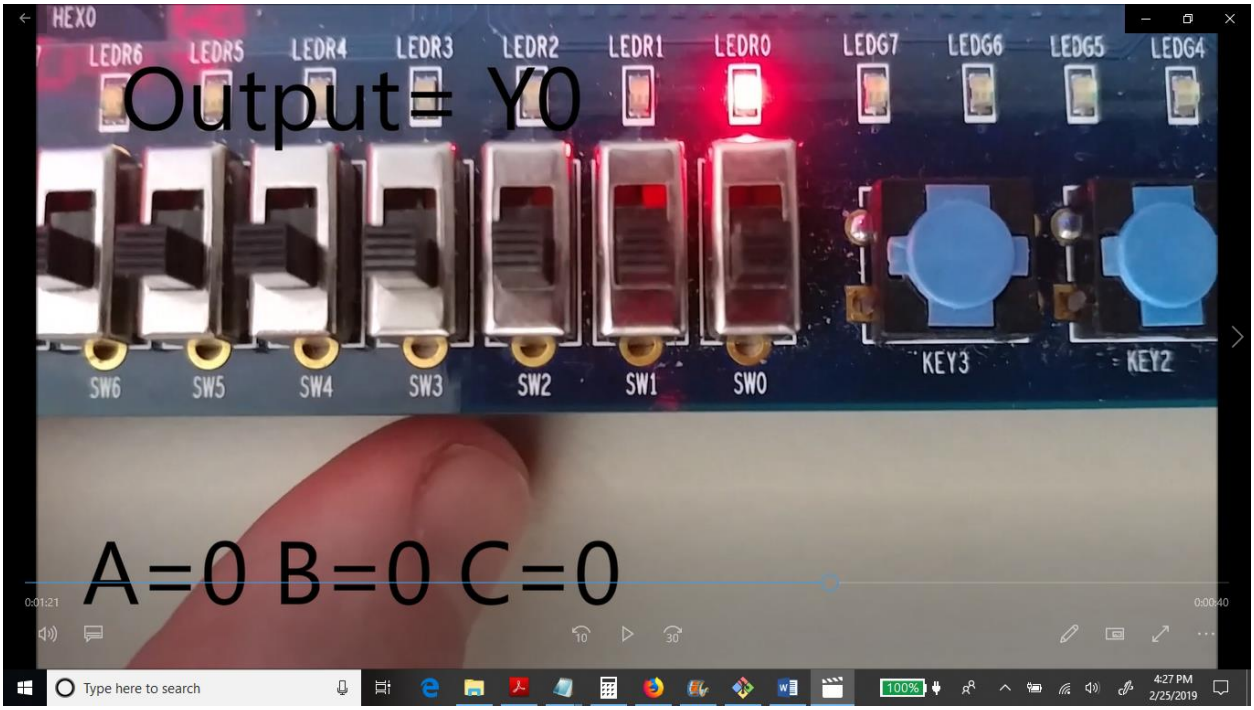

SRAM 512KB
FLASH 4MB
Co=0 S=1
HEX1 HEX0
LEDR8 LEDR7 LEDR6 LEDR5 LEDR4 LEDR3 LEDR2 LEDR1 LEDR0
LEDG7 LEDG6 LEDG5 LEDG4
SW8 SW7 SW6 SW5 SW2 SW1 SW0
KEY3 KEY2
Ci=1 Y=0 X=0


Co=0 S=1
SRAM 512KB
FLASH 4MB
HEX1
HEX0
LEDR8
LEDR7
LEDR6
LEDR5
LEDR4
LEDR3
LEDR2
LEDR1
LEDR0
LEDG7
LEDG6
LEDG5
LEDG4
SW6
SW5
SW4
SW3
SW2
SW1
SW0
KEY3
KEY2
Ci=0 Y=1 X=0

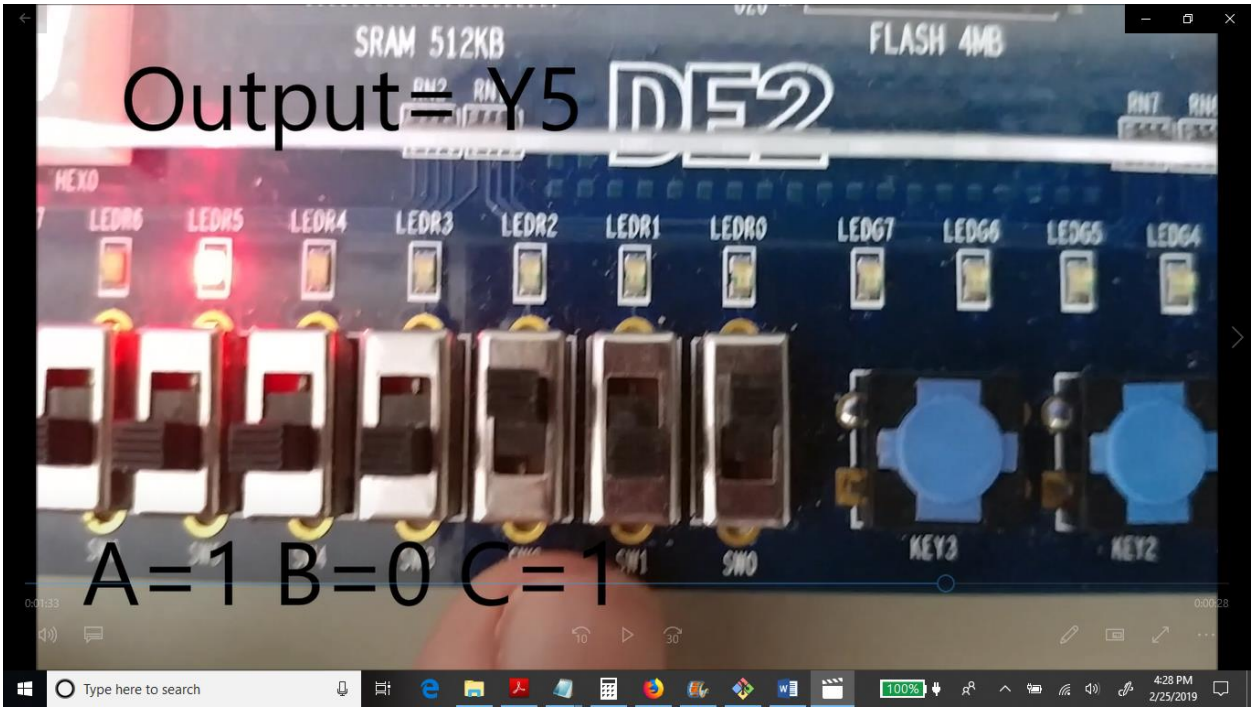

Co=1 S=0
Ci=1 Y=1 X=0
SRAM 512KB
FLASH 4MB
HEX2 HEX1 HEX0
LEDR9 LEDR8 LEDR7 LEDR6 LEDR5 LEDR4 LEDR3 LEDR2 LEDR1 LEDR0
LEDG7 LEDG6 LEDG5 LEDG4
SW8 SW7 SW6 SW5 SW4 SW3 SW2 SW1 SW0
KEY3 KEY2
Type here to search
4:26 PM
2/25/2019


Co=1 S=1
Ci=1 Y=1 X=1
SRAM 512KB
FLASH 4MB
HEX3 HEX2 HEX1 HEX0
LEDR9 LEDR8 LEDR7 LEDR6 LEDR5 LEDR4 LEDR3 LEDR2 LEDR1 LEDR0
LEDG7 LEDG6 LEDG5 LEDG4
SW5 SW4 SW0
KEY3 KEY2
Type here to search
4:27 PM
2/25/2019

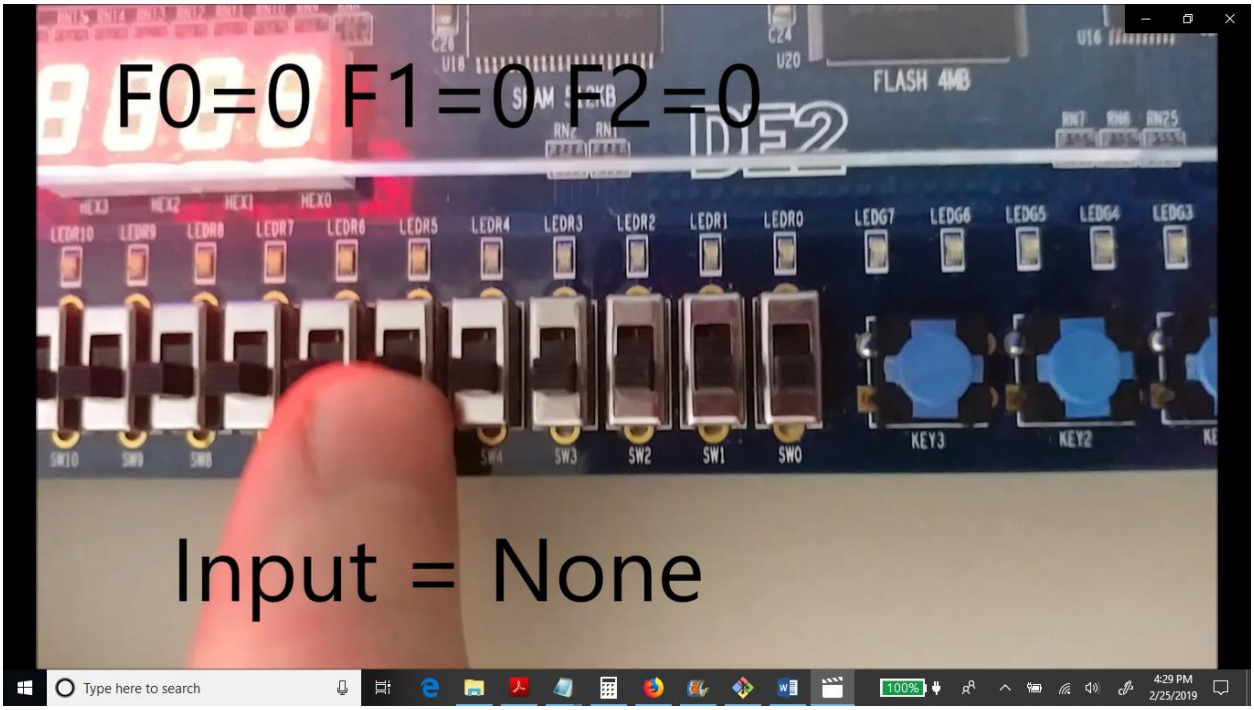
## 3to8 Decoder




Output= Y4
HEX0
LEDR6 LEDR5 LEDR4 LEDR3 LEDR2 LEDR1 LEDR0
LEDG7 LEDG6 LEDG5 LEDG4
SW6 SW5 SW4 SW3 SW2 SW1 SW0
KEY3 KEY2
A=1 B=0 C=0
Type here to search
100%
4:28 PM
2/25/2019

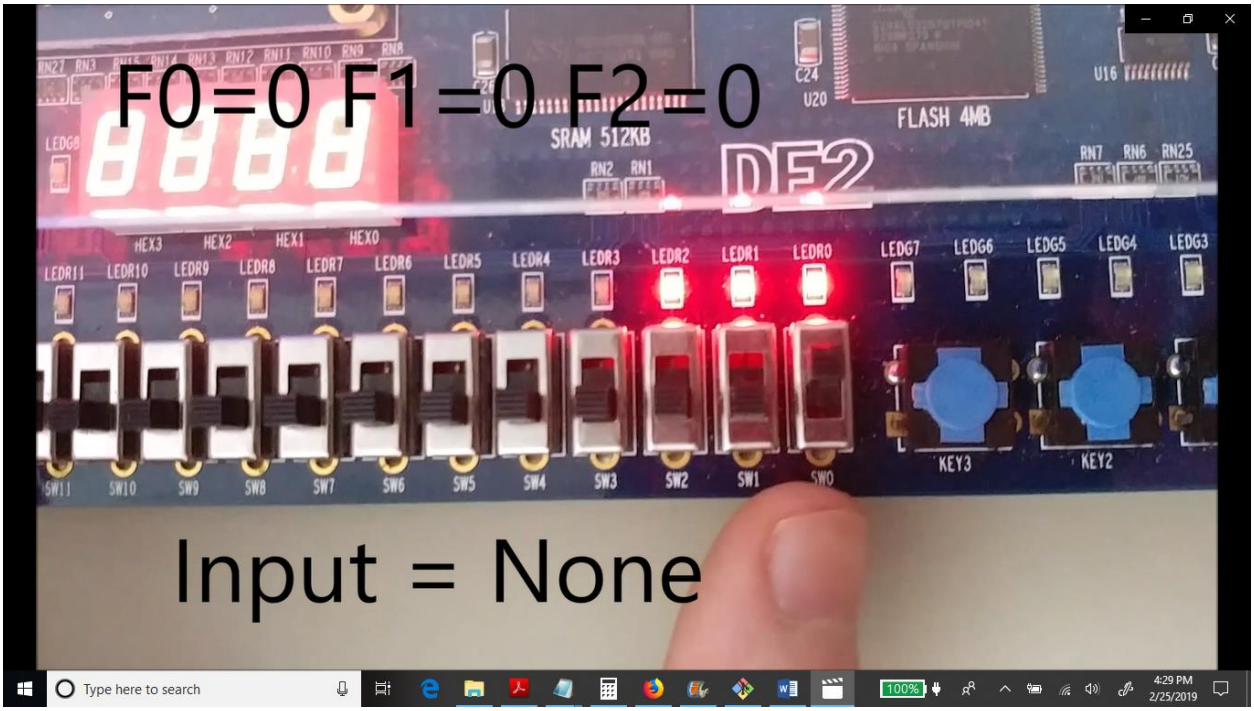

Output= Y5
SRAM 512KB
FLASH 4MB
DE2
HEX0
LEDR6 LEDR5 LEDR4 LEDR3 LEDR2 LEDR1 LEDR0
LEDG7 LEDG6 LEDG5 LEDG4
KEY3 KEY2
A=1 B=0 C=1
Type here to search
100%
4:28 PM
2/25/2019


## 8to3 Encoder




F0=0 F1=1 F2=1
SRAM 512KB
FLASH 4MB
DE2
LEDR10 LEDR9 LEDR8 LEDR7 LEDR6 LEDR5 LEDR4 LEDR3 LEDR2 LEDR1 LEDR0
LEDG7 LEDG6 LEDG5 LEDG4 LEDG3
SW10 SW9 SW8 SW4 SW3 SW2 SW1 SW0
KEY3 KEY2
Input = Y4
Type here to search
4:29 PM
2/25/2019


F0=1 F1=0 F2=1
SRAM 512KB
FLASH 4MB
DE2
LEDR10 LEDR9 LEDR8 LEDR7 LEDR6 LEDR5 LEDR4 LEDR3 LEDR2 LEDR1 LEDR0
LEDG7 LEDG6 LEDG5 LEDG4 LEDG3
SW10 SW9 SW8 SW7 SW6 SW5 SW4 SW3 SW2 SW1 SW0
KEY3 KEY2
Input = Y2
Type here to search
4:29 PM
2/25/2019




## Section 5) Conclusion

In this lab I designed various circuits, which were a 2to1 Mux, 1-bit Half Adder, 1-bit Full Adder, a 3to8 Decoder, and 8to3 Decoder. I learned various things about designing in VHDL. I learned how to design components, simulate them and then create testbenches for all these components. I also learned about improving readability and more efficient programming, for example using for loops with vectors instead of having the same statement repeated many times. I also learned some debugging skills and learned a more efficient way of organizing my projects from now. Another unexpected thing was how intense designing for FPGA boards can be and the amount of time that must be invested into having a working system, which I did not expect for just the first lab.