John Arena

Professor Gertner

CSC 342/343

Lab 2

Due 3/17/19

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 create a static random-access memory (also known as **SRAM)** chip using D-Latches. We will be able to store multiple bits at various addresses. Then we can output the values in a 7-segment display. So the following objectives must be completed

- Designing a 16x4 SRAM connected to a 4 to 7 decoder to output to a 7-segment display. Input will be using 4 signals and a key signal to write to memory.
- Designing a 16x32 SRAM. An appropriate decoder must be attached. Must figure out how to input a 32bit number with a limited number of switches available. Then the decoder will output to a 7-segment display.
- Design a 16x4 SIGNED SRAM
- Verifying their correctness using waveform simulations
- Programming pin assignments for the board

Section 2) Description and Specifications

SR Latch

The first circuit I will be designing is a **SR Latch**. "A latch is a binary storage device, composed of two or more gates, with feedback, that is, for the simplest two-gate latch, the output of each gate is connected to the input of the other gate"**[1]**. This simplest type of latch is called an SR Latch. On the next page in **Figure 1** shows this latch.

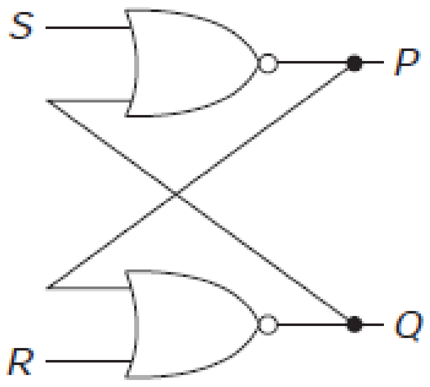


Figure 1: SR Latch

The SR Latch works as follows. If both the *Set* and *Reset* inputs are both 0, there is no change. If *Reset* is on, obviously there's a reset on output *Q*, and if *Set* is on, obviously *Q* is set. Both *Reset* and *Set* produces undefined behavior.

Table **1** below shows the truth table of a **SR Latch**. I will denote the two inputs as S and R and the output as Q and Q'.

S	**R**	**Q**	**Q'**
0	**0**	**Q**	**Q'**
0	**1**	**0**	**1**
1	**0**	**1**	**0**
1	**1**	**Undefined**	**Undefined**

Table 1: SR Latch Truth Table

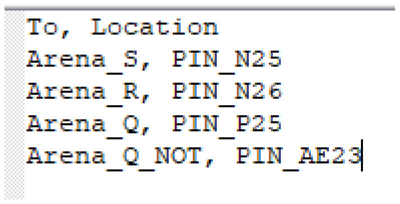
The equations representing them are shown below in Equation **1**

$$Q = R \downarrow \hat{Q}_p$$
$$\hat{Q} = S \downarrow Q_p$$

Equation 1: Equations for SR Latch

The upside-down arrow denotes NOR. So this design will be composed of two NOR gates. The design of NOR 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_S, PIN_N25
Arena_R, PIN_N26
Arena_Q, PIN_P25
Arena_Q_NOT, PIN_AE23
```

Figure 2: Pin Assignment for SR Latch

The format is as follows. To, Location. To is the input/outputs from the object 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 SR Latch.

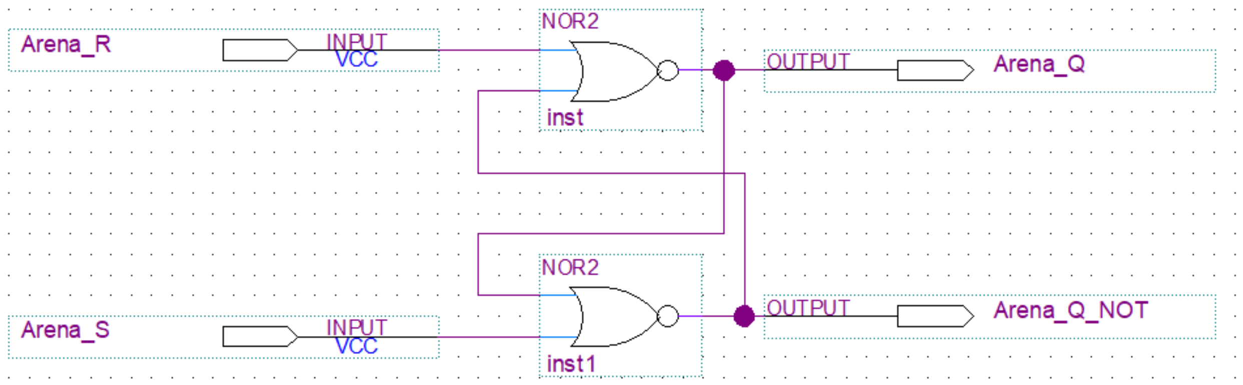


Figure 3: SR Latch at the Logic Level

As can be seen in the figure, the output consists of two 2-input NOR gates connected to the appropriate inputs, described in Equation 1.

Control SR Latch

The second circuit I will be designing is a **Control SR Latch.** A Control SR Latch is similar to a SR Latch, except we now have a control bit that determines the state changes. When the control bit is 0, there is no change and the previous state remains the same. When the control bit is 1, then the circuit acts like a normal SR Latch. On the next page in **num here** is a diagram of one.

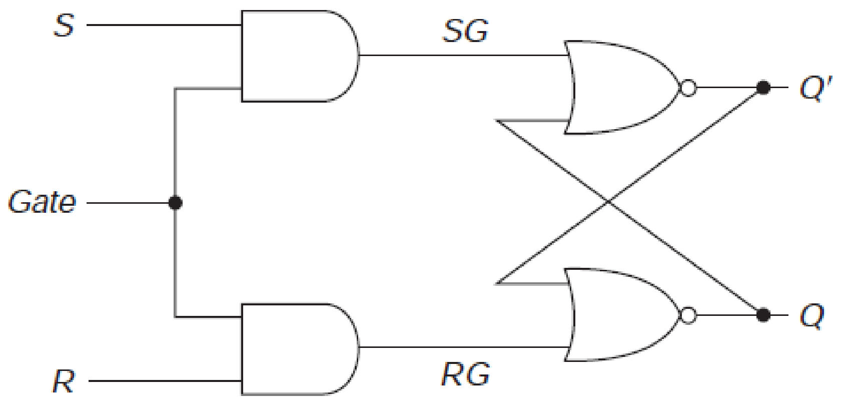


Figure 4 : Control SR Latch

Knowing these rules, below in Table **2** describes the functionality of a Control SR Latch, with C denoting Control.

C	**S**	**R**	**Q**	**Q'**
0	**0**	**0**	**Q**	**Q'**
0	**0**	**1**	**Q**	**Q'**
0	**1**	**0**	**Q**	**Q'**
0	**1**	**1**	**Q**	**Q'**
1	**0**	**0**	**Q**	**Q'**
1	**0**	**1**	**0**	**1**
1	**1**	**0**	**1**	**0**
1	**1**	**1**	**Undefined**	**Undefined**

Table 2: Control SR Latch 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.

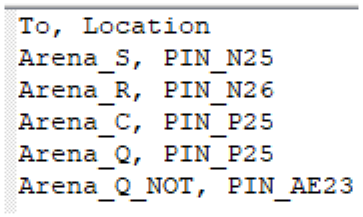
$$Q = (S \uparrow C) \uparrow Q'$$

$$Q' = (R \uparrow C) \uparrow Q'$$

Equation 2: Output of Control SR Latch

Looking at these functions, the arrow pointing up denotes NAND. So, these will be created using NAND gates, specifically 4 NAND gates by looking at the equation.

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



```
To, Location
Arena_S, PIN_N25
Arena_R, PIN_N26
Arena_C, PIN_P25
Arena_Q, PIN_P25
Arena_Q_NOT, PIN_AE23
```

Figure 5: Pin Assignment for 2to1 Mux

There is 3 input switches used and 2 output LEDs. Below in Figure **6** is the design I made in Quartus for the Control SR Latch.

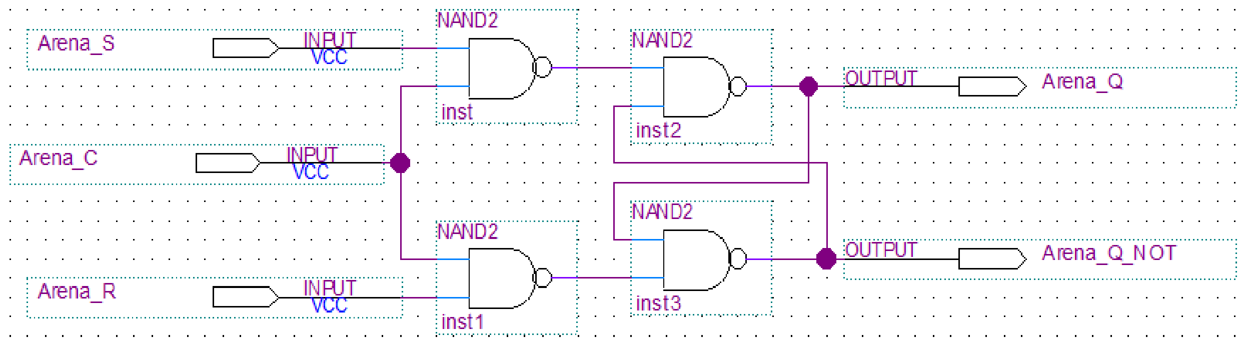


Figure 6: Control SR Latch at Gate Level

As seen in the figure, there are three inputs, S and R in their own respective NAND gates but both sharing the control bit, and the outputs going into a structure similar to the SR Latch designed previously, except that here the SR Latch uses NAND gates. This is okay! SR Latches can be designed using NAND or NOR gates, the truth table will change a bit but the functionality is the same.

D Latch

The third circuit I will be designing is a **D Latch.** A D Latch is the same as a Control SR Latch **EXCEPT** S and R have been replaced with D and D’, respectively. This will assure that the circuit will never run into the case where S and R are both 1. This eliminates the undefined state.

Knowing these rules, below in Table **4** describes the functionality of a D Latch.

C	**D**	**Q**	**Q'**	**State**
0	**X**	**Q**	**Q'**	**No Change**
1	**0**	**0**	**1**	**Reset**
1	**1**	**1**	**0**	**Set**

Table 3: D Latch Truth Table

We can derive the Boolean algebra expression of a D Latch looking at table **4**. Looking at the table, we get the following in equation **3** below.

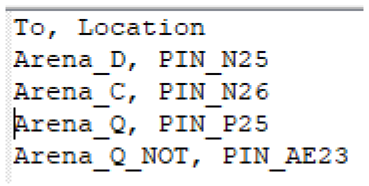
$$Q = (D \uparrow C) \uparrow Q'$$

$$Q' = (D' \uparrow C) \uparrow Q$$

Equation 3: Output of D Latch

Looking at these equations, again, the up arrow denotes a NAND gate. So there are three inputs and 4 NAND gates.

The inputs and outputs will be assigned as follows on our board, seen in Figure **7** below. It comes from the pin assignment text file for this circuit.



```
To, Location
Arena_D, PIN_N25
Arena_C, PIN_N26
Arena_Q, PIN_P25
Arena_Q_NOT, PIN_AE23
```

Figure 7: Pin Assignment D Latch

There is 3 input switches used and 2 output LEDs. On the next page in Figure **8** is the D Latch

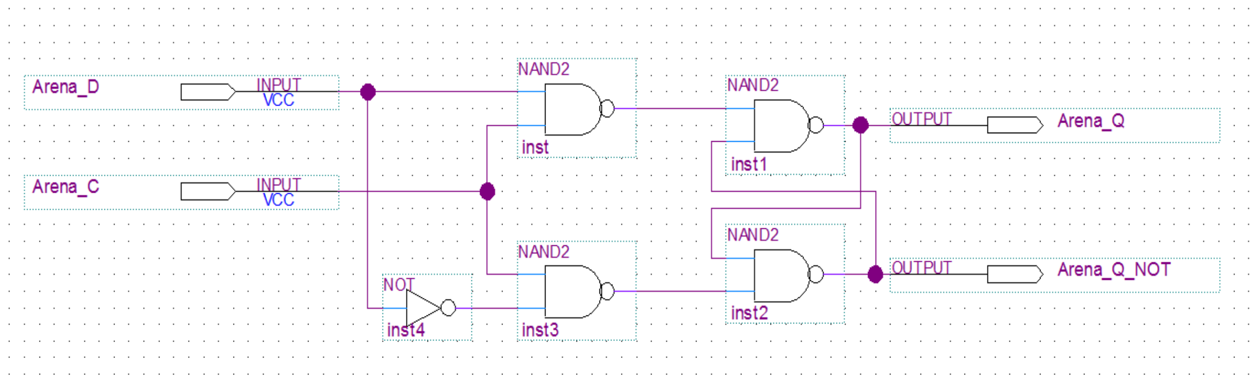


Figure 8: Control SR Latch

As seen in the figure, it's the same as the Control SR Latch except with S and R replaced with D and D', respectively.

Static-RAM Cell with Master Slave D Flip Flops

The fourth circuit I will be designing is a **Static Ram Cell** using **Master-Slave D Flip Flop.** First, a flip flop is a "clocked binary storage device, that stores either a 0 or 1. Under normal operation, that value will only change on the appropriate transition of the clock. The state of the system (that is, what is in memory) changes on the transition of the clock".**[2]** A Master-Slave D Flip Flop is two D Latches connected in series. The outputs of the first D Latch serve as the inputs of the second D Latch. So Q1 goes to D2 and D2'. But one thing to note is the control bit is now considered the clock bit, and it's inverted into the first D Latch, not to the second. This is a negative edge-triggered circuit, rather than a positive/pulse-triggered circuit. So the output Q1 will equal input D only on a clock falling edge in this case. So Q only changes once the clock goes from 1 to 0.

The difference between this and the latch is this flip flop depends on the transition of the clock signal from 1 to 0, while the latch depends on the current state of the clock signal. Below is a truth table denoting the behavior, with m and s denoting master and slave, respectively.

Clock	**Dm**	**Cm**	**Ds**	**Cs**	**Q**
0	**D1**	**1**	**Q1**	**0**	**Q0**
1	**D2**	**0**	**Q1**	**1**	**Q1**
0	**D3**	**1**	**Q3**	**0**	**Q1**
0	**D4**	**1**	**Q4**	**0**	**Q1**
1	**D5**	**0**	**Q4**	**1**	**Q4**
1	**D6**	**0**	**Q4**	**1**	**Q4**

Table 4: Master-Slave D Flip Flop Truth Table

As said, the design will be a Static-RAM Cell. It will be designed using the master-slave d flip flops. An SRAM cell has 3 inputs.

- IN: This is the Latch Data Input.
- Select Chip: This input activates the SRAM Cell when it is high. If it is low, then the cell will **not** store the data, and there will be no output.
- Write Enable: This input allows the latch to store the data from the IN data input.

The SRAM cell will also utilize another gate called a **Tristate Logic Buffer.** It has two inputs, A and B and an output C. It works as follows. When B = 1, C = A. Otherwise, C = Z (which denotes nothing). The truth table is shown below in table **5**.

A	**B**	**C**
0	**0**	**Z**
0	**1**	**0**
1	**0**	**Z**
1	**1**	**1**

Table 5: Tristate Logic Buffer Truth Table

It’s symbol is below in **figure 9** here.

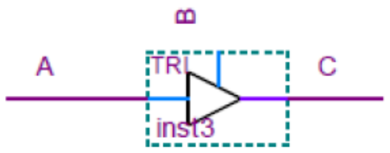
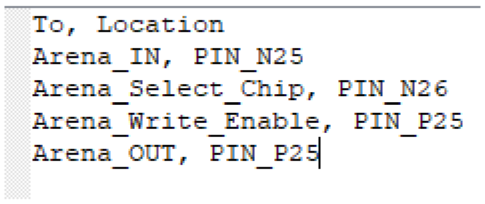


Figure 9: Tristate Logic Buffer

In the SRAM cell, A of the tristate buffer is **Qs** (the output of the slave), B of the tristate buffer is the **Select Chip** , and the output of the SRAM cell is **C** of the tristate buffer. 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 **10.**



```
To, Location
Arena_IN, PIN_N25
Arena_Select_Chip, PIN_N26
Arena_Write_Enable, PIN_P25
Arena_OUT, PIN_P25
```

Figure 10: Pin Assignment for SRAM Cell

There is 3 input switches used and 1 output LEDs. On the next page in figure **11** is the design of the circuit.

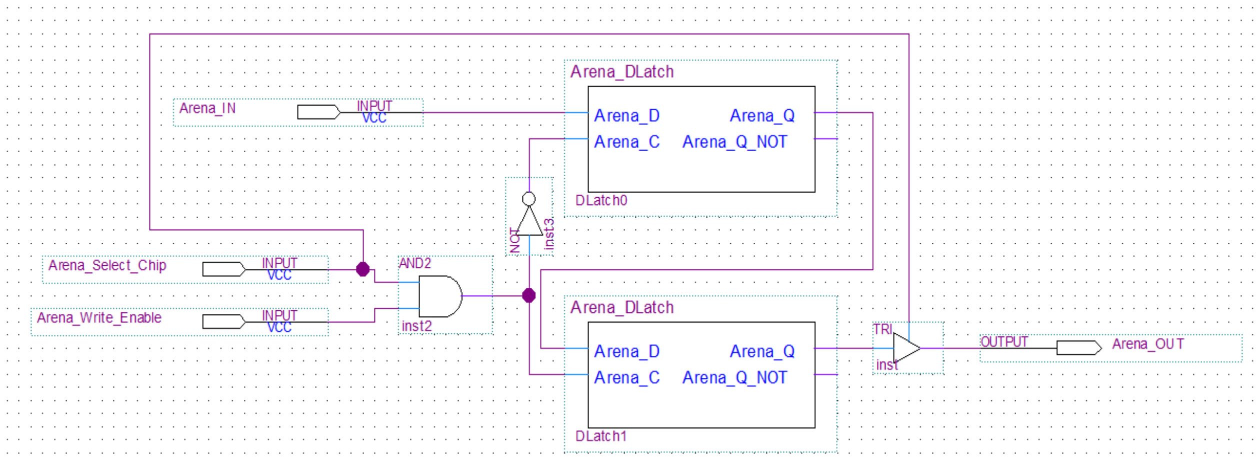


Figure 11: SRAM Cell Design

TRUTH TABLE NEEDED

<u>16x4 Static RAM</u>

The fifth circuit I will be designing is a **16x4 Static RAM** using the SRAM Cell I designed. **16x4** means 16 storage locations (so 16 cells) and it is 4 bits wide. An example of a 4x4 SRAM is shown below in Figure **12**.

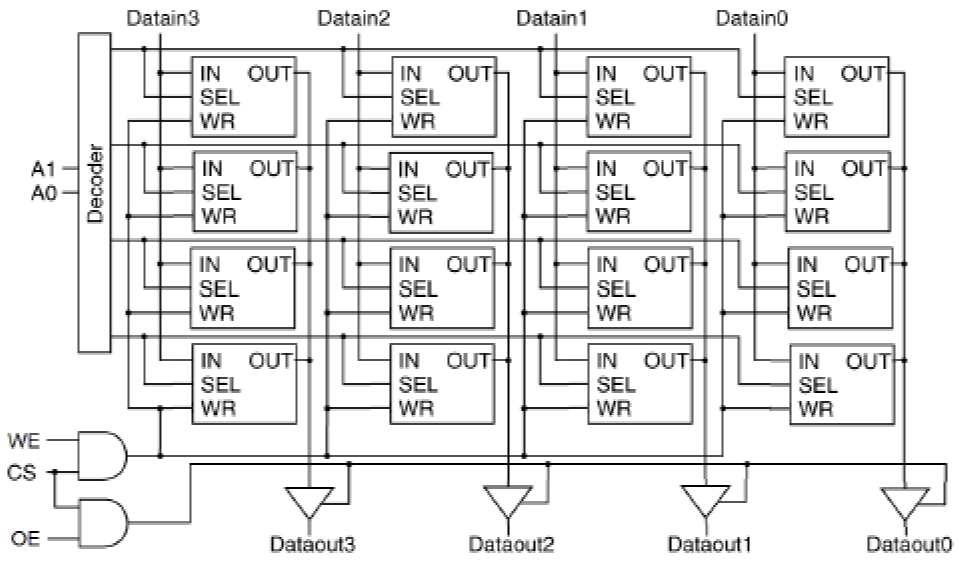


Figure 12: 4x4 SRAM

As can be seen, it is 4 bits wide (Dataout3, 2…Dataout0) and the strength of the tristate buffer is used here. If one of the cells in each bit has an output, the buffer will detect it. This removes the need to have a multiplexer. For the 16x4 SRAM, each bit will have 16 SRAM Cells.

The 16x4 SRAM will have several inputs and outputs:

- DataIn[15..0]: The input that will be written into the SRAM at a certain address.
- A[3..0]: The address input. It tells us which row of SRAM cells will be turned on to read or write data. I will be designing a 4 to 16 Decoder to decode the address input and control the SRAM cells.

- WE: Write Enable. Will tell SRAM to write to the cells when it is high. If low, the chip will not allow any data to be written to any cell.
- OE: Output Enable. When high, SRAM is allowed to output data at whatever address. Otherwise, all tristate buffers will cut output from the SRAM.
- CS: Chip Select. This input when low prevents output from the SRAM and prevents any writing, so it basically shuts the SRAM off. Make it high to turn on the SRAM.

Before getting to the 16x4 SRAM, lets look at the decoder needed for a moment. As the name states, this is a 4 to 16 Decoder, meaning an 4-bit input to a 16-bit output.. We know a 4-bit number has 2^4=16 numbers, 0-15, so we need 16 states, so 16 possible outputs. With that said, I came up with the truth table in table **6** below. The 4 inputs shall be denoted **A,B,C,D** and the outputs **F0, F1…F15.**

A	B	C	D	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Table 6: 4to16 Decoder Truth Table

On the next page in figure **13** is the VHDL code I created for the 4to16 Decoder

```
-- (First, Last) John Arena - CSC 342/343 - Lab 2 - Spring 2019 Due: 3/6/19
-- Arena_4to16Decoder.vhd
library ieee;
use ieee.std_logic_1164.all;

entity Arena_4to16Decoder is
      port(
            Arena_In: in std_logic_vector(3 downto 0); -- 4 inputs
            Arena_Dec: out std_logic_vector(15 downto 0) -- 16 outputs
            );
end Arena_4to16Decoder;

architecture Arena_Arch_4to16Decoder of Arena_4to16Decoder is --
Architecture to describe functionalitty

begin

with Arena_In select
      Arena_Dec <= "0000000000000001" when "0000", -- F0
                               "0000000000000010" when "0001", -- F1...
                               "0000000000000100" when "0010",
                               "0000000000001000" when "0011",
                               "0000000000010000" when "0100",
                               "0000000000100000" when "0101",
                               "0000000001000000" when "0110",
                               "0000000010000000" when "0111",
                               "0000000100000000" when "1000",
                               "0000001000000000" when "1001",
                               "0000010000000000" when "1010",
                               "0000100000000000" when "1011",
                               "0001000000000000" when "1100",
                               "0010000000000000" when "1101",
                               "0100000000000000" when "1110",
                               "1000000000000000" when "1111", --...F15
                               "0000000000000000" when others;
end Arena Arch 4to16Decoder;
```

Figure 13: 4to16 Decoder VHDL code

From this I created a symbol for the decoder, as seen below in figure **14**.

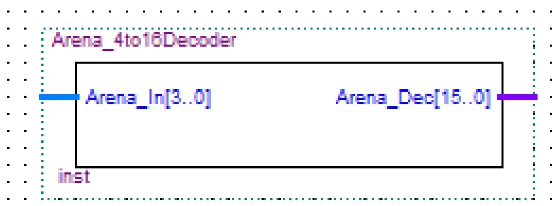


Figure 14: 4to16 Decoder Symbol

Back to the 16x4 SRAM, so we need 16 cells for each bit. I decided to create a 16x1 SRAM so the 16x4 SRAM block symbol file can be reduced in size to make it easier to see. Below in Figure **15** is the 16x1 SRAM.

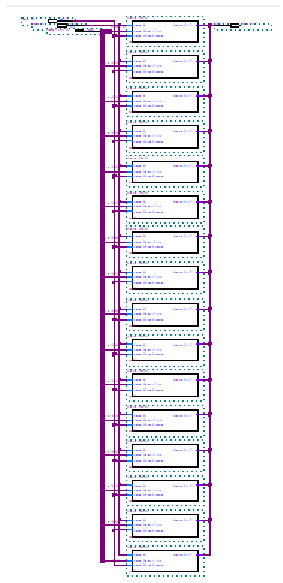


Figure 15: 16x1 SRAM

It's very small which is why it will be better to make it into it's own symbol as I said. Below in **16** is the top zoomed in for a better view

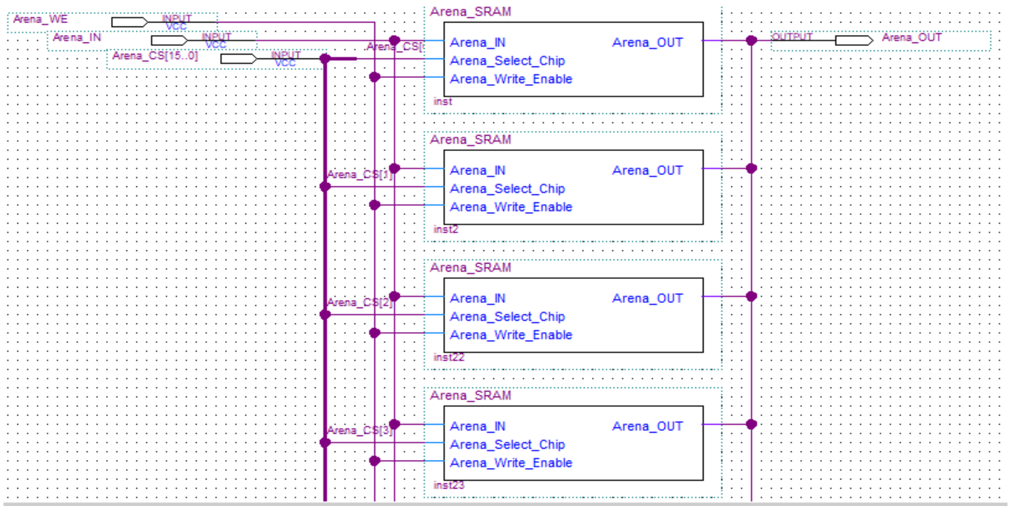


Figure 16: 16x1 SRAM – Zoomed

Below in Figure **17** is the symbol for the 16x1 SRAM cell.

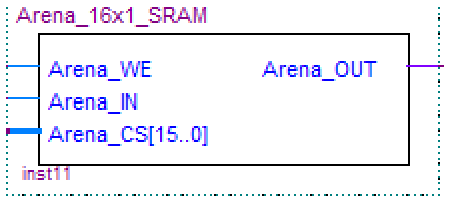


Figure 17: 16x1 SRAM Symbol

Finally connecting all these together, I made a 16x4 SRAM shown below in Figure **18**.

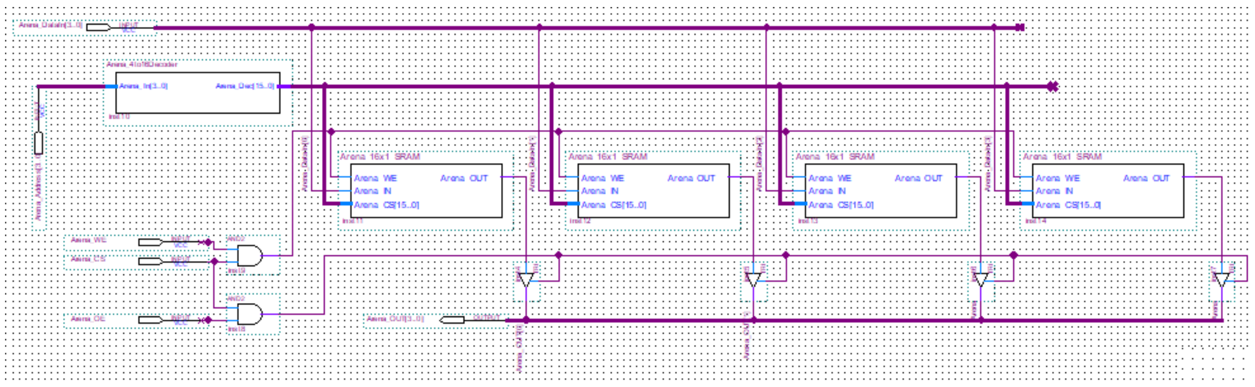


Figure 18: 16x4 SRAM

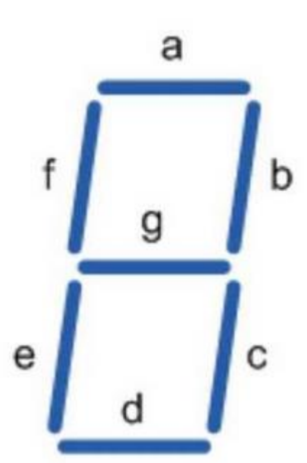
Another component is needed. The component needed is the 4to7 Decoder for our seven segment display. We have 4 outputs, for a total of 2^4 = 16 possibly outputs of 0-15. Using Hex representation, we can display 0-9,A-F on a seven segment display. So 4 inputs to 7 outputs, since there are 7 segments on the display. Below in figure **19** is the VHDL code for it.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 2 - Spring 2019 Due: 3/13/19
-- Arena_Dec_To_Hex.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
-- Hexadecimal to 7 segment decoder for LED display

entity Arena_Dec_To_Hex is
      port(
            Arena_hexDigit : in std_logic_vector(3 downto 0); -- inputs
            Arena_segment_A, Arena_segment_B, Arena_segment_C, --outputs of 
segments
            Arena_segment_D, Arena_segment_E, Arena_segment_F,
            Arena_segment_G : out std_logic
            );
end Arena_Dec_To_Hex;

architecture Arena_Arch_Dec_To_Hex of Arena_Dec_To_Hex is
      signal Arena_segment_data : std_logic_vector(6 downto 0);
begin
      process (Arena_hexDigit)
      -- Hex to 7 segment decoder
      begin
            case Arena_hexDigit is
                  when "0000" =>
                  Arena_segment_data <= "1111110";
                  when "0001" =>
                  Arena_segment_data <= "0110000";
                  when "0010" =>
                  Arena_segment_data <= "1101101";
                  when "0011" =>
                  Arena_segment_data <= "1111001";
                  when "0100" =>
                  Arena_segment_data <= "0110011";
                  when "0101" =>
                  Arena_segment_data <= "1011011";
                  when "0110" =>
                  Arena_segment_data <= "1011111";
                  when "0111" =>
                  Arena_segment_data <= "1110000";
                  when "1000" =>
                  Arena_segment_data <= "1111111";
                  when "1001" =>
                  Arena_segment_data <= "1110011";
                  when "1010" =>
                  Arena_segment_data <= "1110111";
                  when "1011" =>
                  Arena_segment_data <= "0011111";
                  when "1100" =>
                  Arena_segment_data <= "1001110";
                  when "1101" =>
                  Arena_segment_data <= "0111101";
                  when "1110" =>
                  Arena_segment_data <= "1001111";
                  when "1111" =>
                  Arena_segment_data <= "1000111";
            end case;
      end process;
--extract segment data bits and invert since operates on low
-- led driver circuit is inverted
Arena_segment_A <= NOT Arena_segment_data(6); --NOT gate cause the segments 
respond to active low
Arena_segment_B <= NOT Arena_segment_data(5);
Arena_segment_C <= NOT Arena_segment_data(4);
Arena_segment_D <= NOT Arena_segment_data(3);
Arena_segment_E <= NOT Arena_segment_data(2);
```

```
Arena_segment_F <= NOT Arena_segment_data(1);
Arena_segment_G <= NOT Arena_segment_data(0);
end Arena_Arch_Dec_To_Hex;
```

Figure 19: 16x1 SRAM – Zoomed

To explain this, lets look at an example. Below is the seven-segment display



The format for the seven-segments corresponds as follows : 000000-abcdefg. So if the input is ‘0000’, in hex, that is 0. We would have to display a,b,c,d,e,f. No g. So the corresponding segment data would be 1111110. But since as we said the display is active low, it goes through a not gate to get 0000001. This is the idea behind the 4to7 decoder. I called it Dec_To_Hex. Below in figure **20** is the symbol.

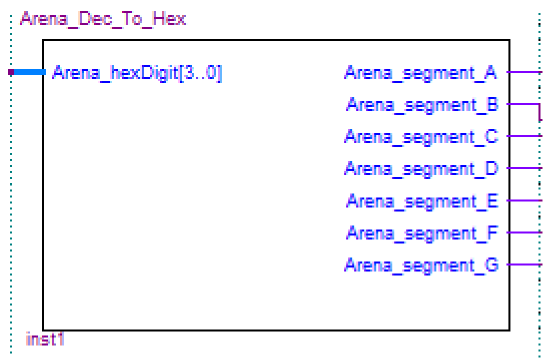


Figure 20: Decimal to Hex Display Symbol

Finally, in figure **21** below is the 16x4 SRAM connected to the decoder. The 16x4 SRAM has been turned into a symbol as well.

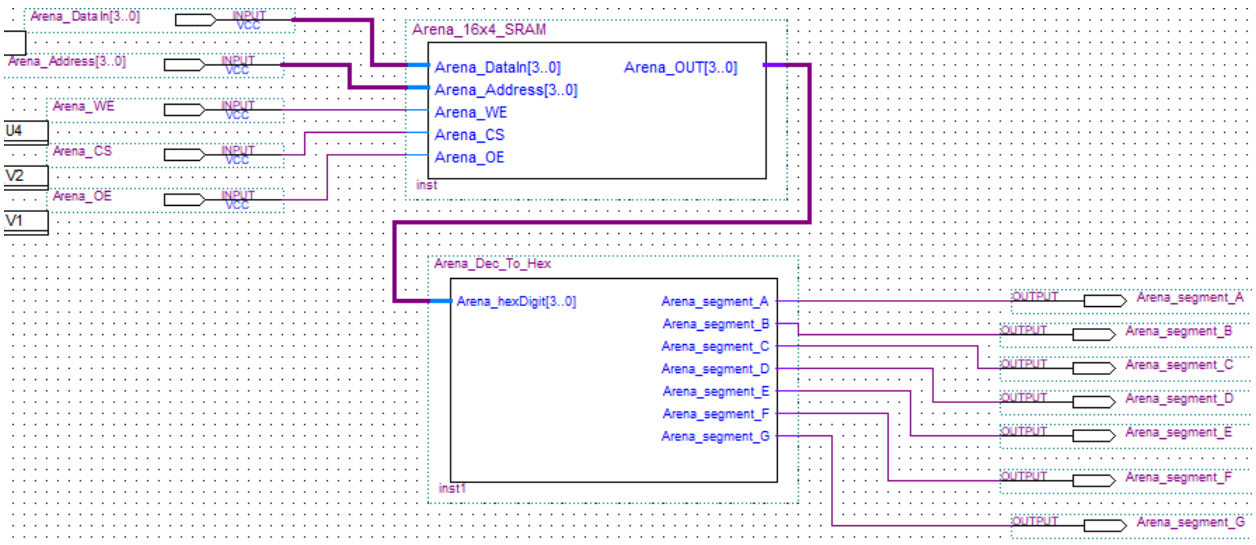
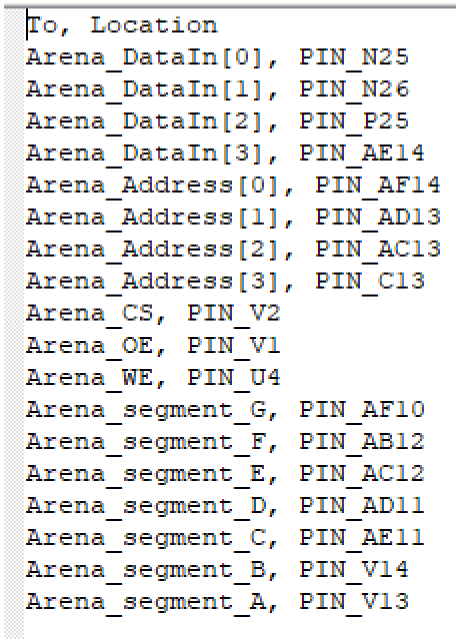


Figure 21: 16x4 SRAM with Seven Segment Display

The pin assignments for the circuit are shown below in figure **22**



```
To, Location
Arena_DataIn[0], PIN_N25
Arena_DataIn[1], PIN_N26
Arena_DataIn[2], PIN_P25
Arena_DataIn[3], PIN_AE14
Arena_Address[0], PIN_AF14
Arena_Address[1], PIN_AD13
Arena_Address[2], PIN_AC13
Arena_Address[3], PIN_C13
Arena_CS, PIN_V2
Arena_OE, PIN_V1
Arena_WE, PIN_U4
Arena_segment_G, PIN_AF10
Arena_segment_F, PIN_AB12
Arena_segment_E, PIN_AC12
Arena_segment_D, PIN_AD11
Arena_segment_C, PIN_AE11
Arena_segment_B, PIN_V14
Arena_segment_A, PIN_V13
```

Figure 22: 16x4 SRAM with Seven Segment Display PIN Assignments

16x32 Static RAM

The sixth circuit I will be designing is a **16x32 Static RAM** using the SRAM Cell I designed. **16x32** means 16 storage locations (so 16 cells) and it is 32 bits wide. So essentially the design for the 16x4 SRAM has to be extended. The inputs and outputs are still the same, the design is still the same, there are just more bits, again, 32 now. Below in Figure **23** is the design. It's very hard to see since its so wide, which can be seen below in Figure **23**.

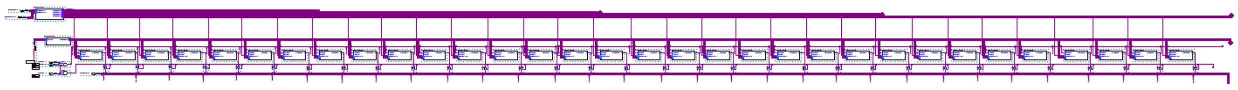


Figure 23: 16x32 SRAM

I created a symbol for it below in figure **24** here.

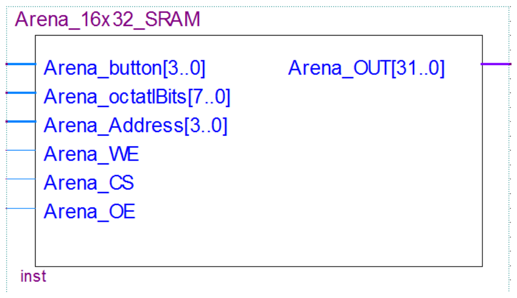
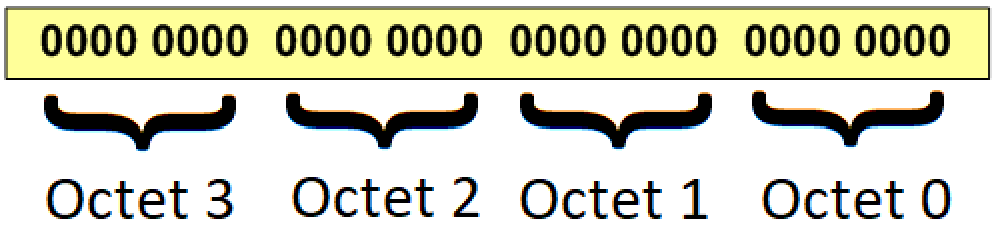


Figure 24: 16x32 SRAM Symbol

Now comes an issue. The FGPA we use has a total of 16 switches. 8 of them are in use from the inputs already needed to control the functionality, which means 8 are remaining. How can a 32 bit be number inputted then? My idea is as follows. Along with the 8 switches, there are 4 buttons available. 32bits/8 switches = 4. I can write some VHDL code to contain octet variables which gets an inputs from the 8 switches, and which octet variable the switch values are assigned to depends on the button pressed. Below is an example of the idea.



So when Button 0 is pressed, the 8 switch values can be stored in octet 0, then when Button 1 is pressed the values are stored in octet 1, and so on. Using this, we can then input these values to the corresponding 16x1 SRAM's. As you can see in the symbol, I have inputs for the buttons and octalbit inputs.

The next issue is how to get the number onto the 7-segment displays. There are only 8 of them. We can do this by converting the 32 bit binary number into an 8 bit hex number. Every 4 digits of the 32 bit number can be represented by hex. So I can just attach the 4 to 7 decoder for every 4 output bits. 8 decoders will be needed for a total of 32 bits. Using the same 4 to 7 decoder used for the 16x4 SRAM, just with appropriate pin assignments, the final circuit diagram is shown below in figure **25**.

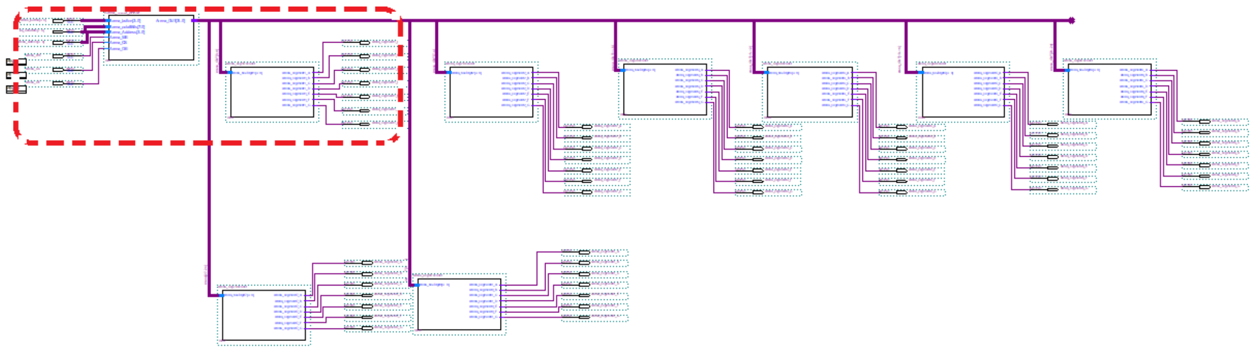


Figure 25: 16x32 SRAM with Seven Segment Displays

The section in red is the figure on the next page in **Figure 26**, it highlights the zoomed up red area.

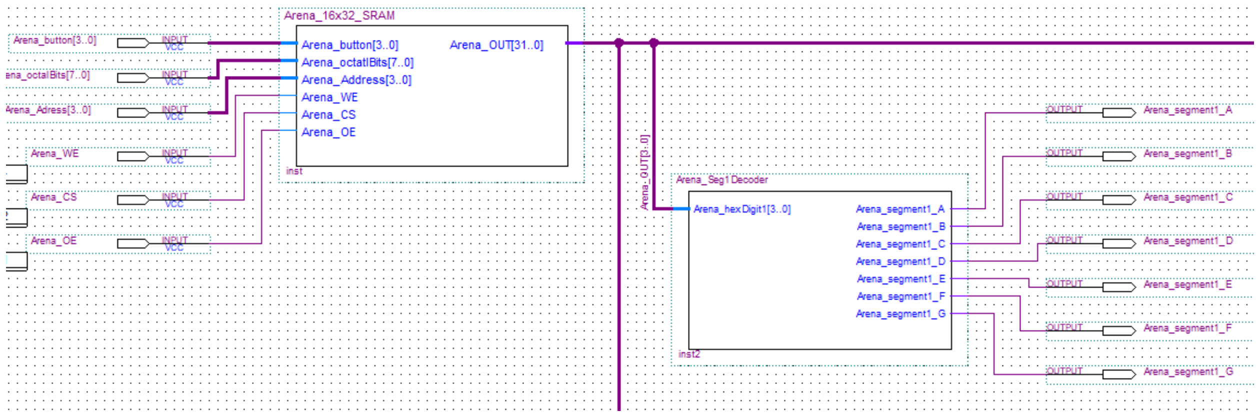


Figure 26: 16x32 SRAM Zoomed In

Below in Figure 27 is the Pin Assignments

```
To, Location
Arena_octalBits[0], PIN_N25
Arena_octalBits[1], PIN_N26
Arena_octalBits[2], PIN_P25
Arena_octalBits[3], PIN_AE14
Arena_octalBits[4], PIN_AF14
Arena_octalBits[5], PIN_AD13
Arena_octalBits[6], PIN_AC13
Arena_octalBits[7], PIN_C13
Arena_Address[0], PIN_B13
Arena_Address[1], PIN_A13
Arena_Address[2], PIN_N1
Arena_Address[3], PIN_P1
Arena_CS, PIN_V2
Arena_OE, PIN_V1
Arena_WE, PIN_U4
Arena_button[0], PIN_G26
Arena_button[1], PIN_N23
Arena_button[2], PIN_P23
Arena_button[3], PIN_W26

Arena_segment1_G, PIN_V13
Arena_segment1_F, PIN_V14
Arena_segment1_E, PIN_AE11
Arena_segment1_D, PIN_AD11
Arena_segment1_C, PIN_AC12
Arena_segment1_B, PIN_AB12
Arena_segment1_A, PIN_AF10

Arena_segment2_A, PIN_V20
Arena_segment2_B, PIN_V21
Arena_segment2_C, PIN_W21
Arena_segment2_D, PIN_Y22
Arena_segment2_E, PIN_AA24
Arena_segment2_F, PIN_AA23
```

```
Arena_segment3_A, PIN_AB23
Arena_segment3_B, PIN_V22
Arena_segment3_C, PIN_AC25
Arena_segment3_D, PIN_AC26
Arena_segment3_E, PIN_AB26
Arena_segment3_F, PIN_AB25
Arena_segment3_G, PIN_Y24

Arena_segment4_A, PIN_Y23
Arena_segment4_B, PIN_AA25
Arena_segment4_C, PIN_AA26
Arena_segment4_D, PIN_Y26
Arena_segment4_E, PIN_Y25
Arena_segment4_F, PIN_U22
Arena_segment4_G, PIN_W24

Arena_segment5_A, PIN_U9
Arena_segment5_B, PIN_U1
Arena_segment5_C, PIN_U2
Arena_segment5_D, PIN_T4
Arena_segment5_E, PIN_R7
Arena_segment5_F, PIN_R6
Arena_segment5_G, PIN_T3

Arena_segment6_A, PIN_T2
Arena_segment6_B, PIN_P6
Arena_segment6_C, PIN_P7
Arena_segment6_D, PIN_T9
Arena_segment6_E, PIN_R5
Arena_segment6_F, PIN_R4
Arena_segment6_G, PIN_R3

Arena_segment7_A, PIN_R2
Arena_segment7_B, PIN_P4
Arena_segment7_C, PIN_P3
Arena_segment7_D, PIN_M2
Arena_segment7_E, PIN_M3
Arena_segment7_F, PIN_M5
Arena_segment7_G, PIN_M4

Arena_segment8_A, PIN_L3
Arena_segment8_B, PIN_L2
Arena_segment8_C, PIN_L9
Arena_segment8_D, PIN_L6
Arena_segment8_E, PIN_L7
Arena_segment8_F, PIN_P9
Arena_segment8_G, PIN_N9
```

Figure 27: 16x32 SRAM Pin Assignments

16x4 SIGNED Static RAM

The seventh and final circuit I will be designing is a **16x4 SIGNED Static RAM** using the SRAM Cell I designed. This is the exact same design as the 16x4 SRAM except with one change. The decoder now has a switch to indicate whether we want unsigned or signed values. So for unsigned, 2^4 = 16, 0,1,2..15. For signed, 2^4=16, 0,1..8 and -8,-7..-1. It's design can be seen below in Figure **28** and the VHDL code in Figure **29.**

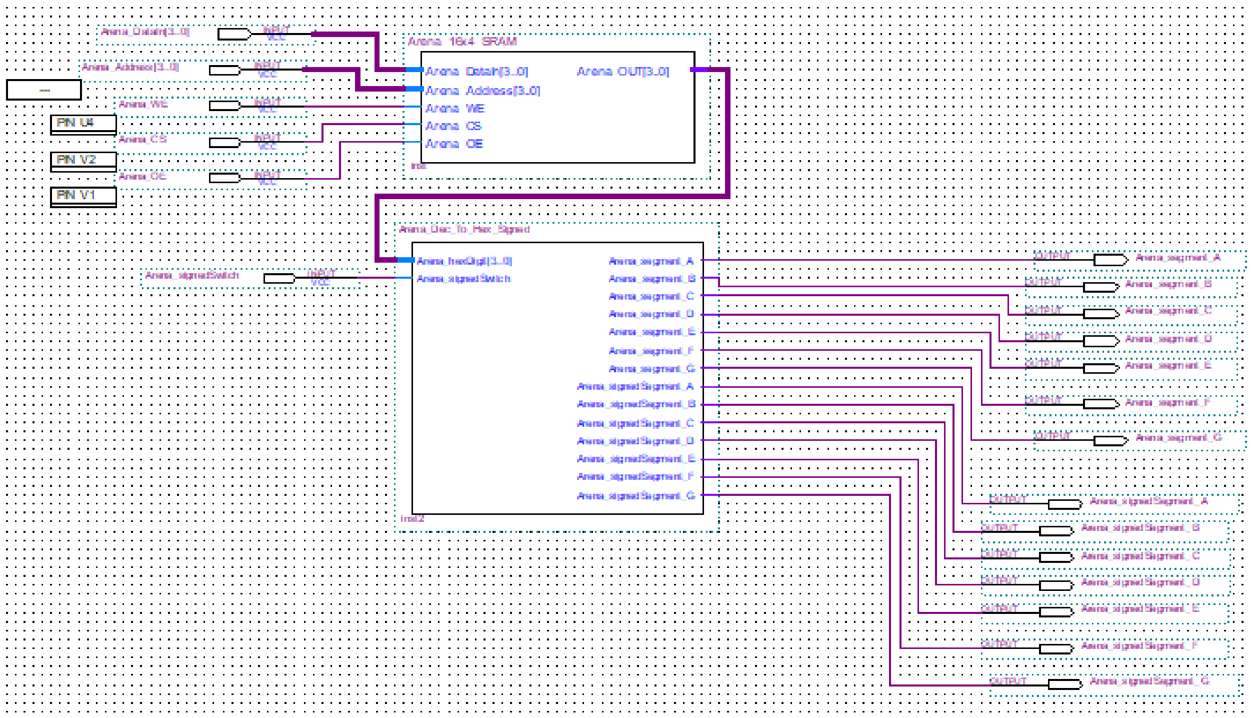


Figure 28: 16x4 Signed SRAM with Seven Segment Display

```
-- (First, Last) John Arena - CSC 342/343 - Lab 2 - Spring 2019 Due: 3/13/19
-- Arena_Dec_To_Hex.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
-- Hexadecimal to 7 segment decoder for LED display
entity Arena_Dec_To_Hex_Signed is
    port(
        Arena_hexDigit : in std_logic_vector(3 downto 0); -- inputs
        Arena_signedSwitch : in std_logic;
        Arena_segment_A, Arena_segment_B, Arena_segment_C, --outputs of
segments
```

```
        Arena_segment_D, Arena_segment_E, Arena_segment_F,
        Arena_segment_G : out std_logic;
              Arena_signedSegment_A, Arena_signedSegment_B,
Arena_signedSegment_C, -- outputs of segments for sign + or -
              Arena_signedSegment_D, Arena_signedSegment_E,
Arena_signedSegment_F,
              Arena_signedSegment_G : out std_logic
        );
end Arena_Dec_To_Hex_Signed;
architecture Arena_Arch_Dec_To_Hex_Signed of Arena_Dec_To_Hex_Signed is
    signal Arena_segment_data : std_logic_vector(6 downto 0);
    signal Arena_hexDigit_converted : signed(3 downto 0);
    signal Arena_signedSegment_data : std_logic_vector(6 downto 0);
begin
Arena_hexDigit_converted <= signed(Arena_hexDigit);
process(Arena_signedSwitch)
begin
case Arena_signedSwitch is
    when '1' =>
            case Arena_hexDigit_converted is
                when "0000" => --0
                Arena_segment_data <= "1111110";
                when "0001" => --1
                Arena_segment_data <= "0110000";
                when "0010" => --2
                Arena_segment_data <= "1101101";
                when "0011" => --3
                Arena_segment_data <= "1111001";
                when "0100" => --4
                Arena_segment_data <= "0110011";
                when "0101" => --5
                Arena_segment_data <= "1011011";
                when "0110" => --6
                Arena_segment_data <= "1011111";
                when "0111" => --7
                Arena_segment_data <= "1110000";
                when "1000" => --8
                Arena_segment_data <= "1111111";
                when "1001" => -- -7
                Arena_segment_data <= "1110000";
                when "1010" => -- -6
                Arena_segment_data <= "1011111";
                when "1011" => -- -5
                Arena_segment_data <= "1011011";
                when "1100" => -- -4
                Arena_segment_data <= "0110011";
                when "1101" => -- -3
                Arena_segment_data <= "1111001";
                when "1110" => -- -2
                Arena_segment_data <= "1101101";
                when "1111" => -- -1
                Arena_segment_data <= "0110000";
            end case;
            Arena_signedSegment_data <= "0000001";
    when '0' =>
            case Arena_hexDigit is
                when "0000" =>
                Arena_segment_data <= "1111110";
                when "0001" =>
                Arena_segment_data <= "0110000";
                when "0010" =>
                Arena_segment_data <= "1101101";
                when "0011" =>
                Arena_segment_data <= "1111001";
                when "0100" =>
                Arena_segment_data <= "0110011";
                when "0101" =>
                Arena_segment_data <= "1011011";
                when "0110" =>
                Arena_segment_data <= "1011111";
                when "0111" =>
                Arena_segment_data <= "1110000";
                when "1000" =>
                Arena_segment_data <= "1111111";
                when "1001" =>
                Arena_segment_data <= "1110011";
                when "1010" =>
                Arena_segment_data <= "1110111";
                when "1011" =>
                Arena_segment_data <= "0011111";
                when "1100" =>
                Arena_segment_data <= "1001110";
                when "1101" =>
                Arena_segment_data <= "0111101";
                when "1110" =>
                Arena_segment_data <= "1001111";
                when "1111" =>
                Arena_segment_data <= "1000111";
            end case;
            Arena_signedSegment_data <= "0110001";
    when others =>
        null;
end case;
end process;
--extract segment data bits and invert since operates on low
-- led driver circuit is inverted
Arena_segment_A <= NOT Arena_segment_data(6); --NOT gate cause the segments
respond to active low
Arena_segment_B <= NOT Arena_segment_data(5);
Arena_segment_C <= NOT Arena_segment_data(4);
Arena_segment_D <= NOT Arena_segment_data(3);
Arena_segment_E <= NOT Arena_segment_data(2);
Arena_segment_F <= NOT Arena_segment_data(1);
Arena_segment_G <= NOT Arena_segment_data(0);
```

```
Arena_signedSegment_A <= NOT Arena_signedSegment_data(6);
Arena_signedSegment_B <= NOT Arena_signedSegment_data(5);
Arena_signedSegment_C <= NOT Arena_signedSegment_data(4);
Arena_signedSegment_D <= NOT Arena_signedSegment_data(3);
Arena_signedSegment_E <= NOT Arena_signedSegment_data(2);
Arena_signedSegment_F <= NOT Arena_signedSegment_data(1);
Arena_signedSegment_G <= NOT Arena_signedSegment_data(0);
end Arena_Arch_Dec_To_Hex_Signed;
```

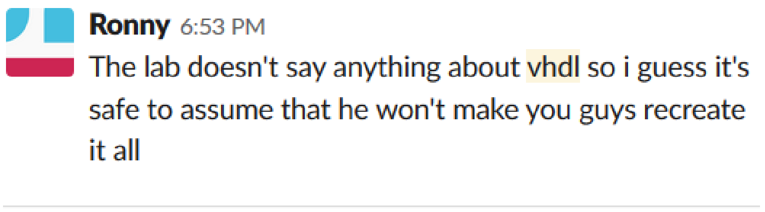
Figure 29: Signed Seven Segment Display VHDL Code

Section 3) Simulations

SR Latch

The first simulation will be done for the SR Latch.

THE LAB PDF SAID NOTHING OF VHDL FILES OR BENCHMARKS, SO THEY ARE NOT INCLUDED. TA'S ALSO CONFIRMED ON SLACK.



Ronny 6:53 PM
The lab doesn't say anything about vhdl so i guess it's safe to assume that he won't make you guys recreate it all

Figure **30** below is shows results of the SR Latch waveform.. Our results should correspond with the truth table in Table **1**.

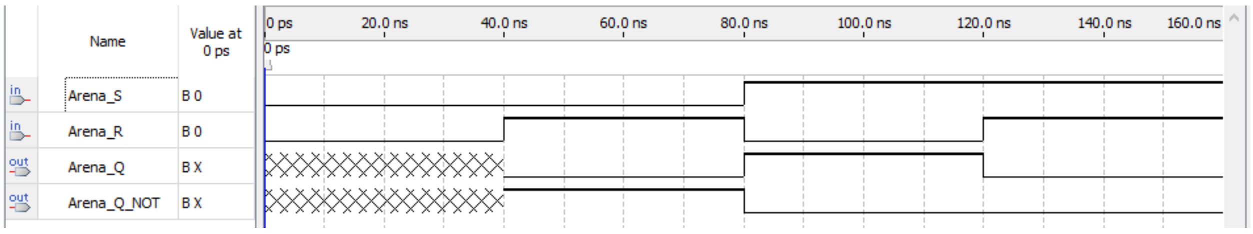


Figure 30: SR Latch Waveform

Looking at the figures above, we can see our design for the SR Latch is correct. We see comparing the waveform to the truth table. We know whenever Arena_S = 0, the output

Arena_M = X. Looking at 00 for example, we see no change. 11 gives an unknown change, here it made it low, and the other states set Q appropriately.

Control SR Latch

The second simulation will be done for the Control SR Latch. Our results should correspond with the truth table in Table **2**. Below in Figure **31** is the results from the Waveform file.

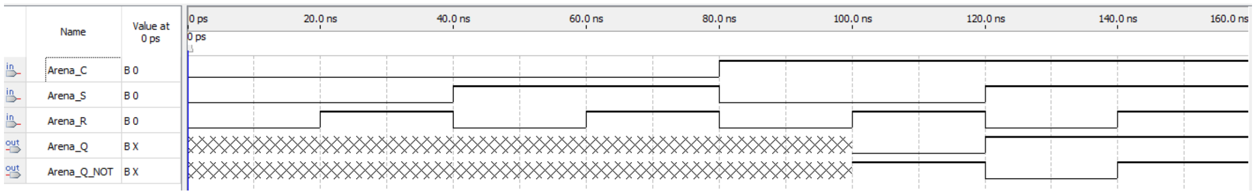


Figure 31: Control SR Latch Waveform

We can see it's correct is comparing the Waveform to the truth table. We can see in the figures whenever Control is 0, the circuit does not change Q. When its high, it behaves like an SR Latch.

D Latch

The third simulation will be done for the D Latch. Our results should correspond with the truth table in Table **3**. Below in Figure **32** is the results from the Waveform file.

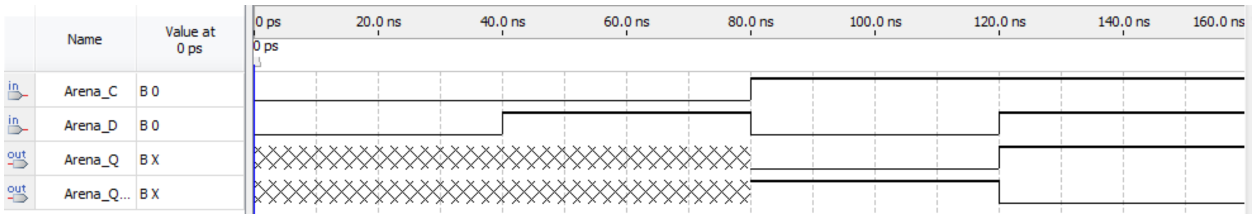


Figure 32: D Latch Waveform

We see it's correct is comparing the Waveform to the truth table. We can see in the figures that when control is low, the circuit does not change Q. When its high, Q is set appropriately, without the problem of having an input of '11'.

SRAM Cell

The fourth simulation will be done for the SRAM Cell. The results should correspond with the truth table in Table **4**. Below in Figure **33** is the results from the Waveform file.

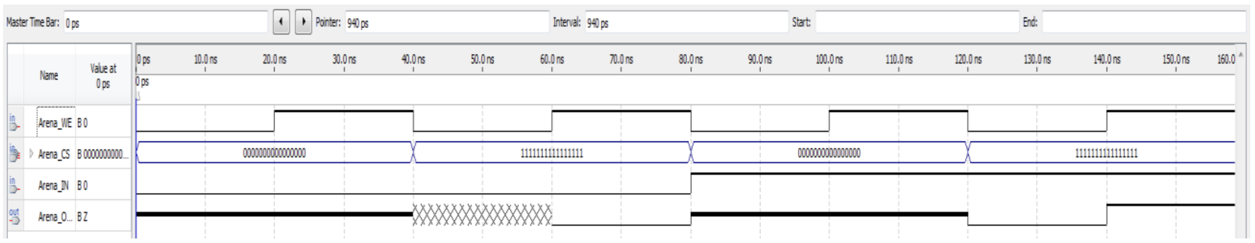


Figure 33: SRAM Cell Waveform

We see it's correct when comparing the Waveform to the truth table. Whenever Select chip and Write Enable is set to high, output is whatever the input was. Other stands produce a nothing value or no change appropriately, as shown in the truth table.

16x4 SRAM

The fifth simulation will be done for the 16x4 SRAM. he results should correspond with the truth table in Table **5**. Below in Figure **34** is the results from the Waveform file.

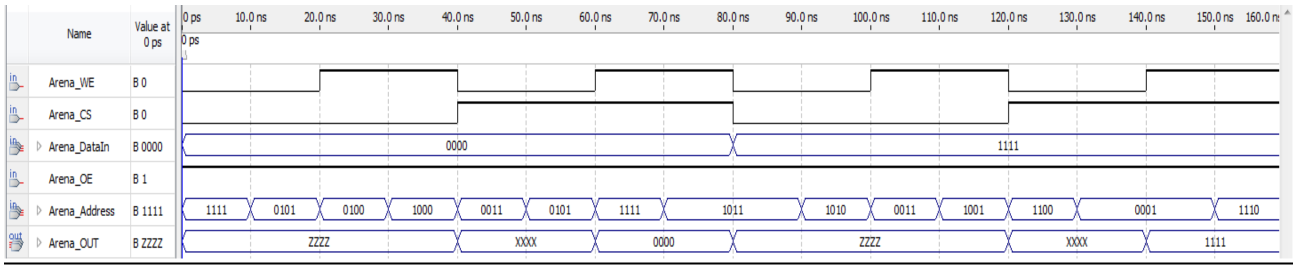


Figure 34: 16x4 SRAM Waveform

We can see it’s correct is comparing the Waveform to what we know about SRAM We can see when Chip Select is disabled, nothing is outputted. When its high, something can be outputted. Only when write enable is high is data written to the output when CS is high, otherwise no change happens at the output. Below in figure **35** shows what happens when output enable is disabled.

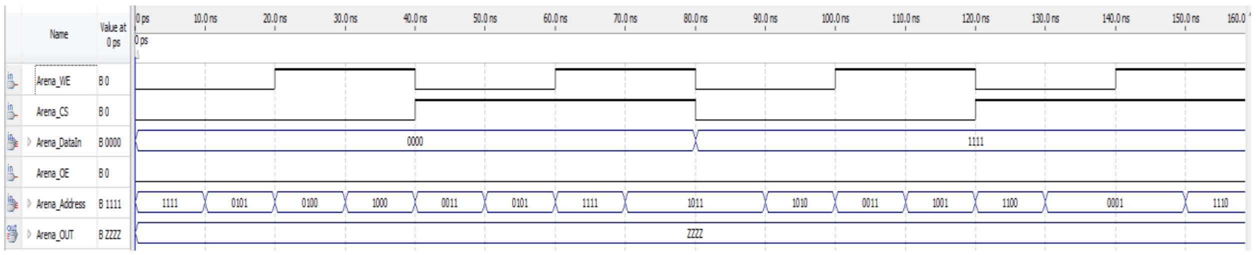
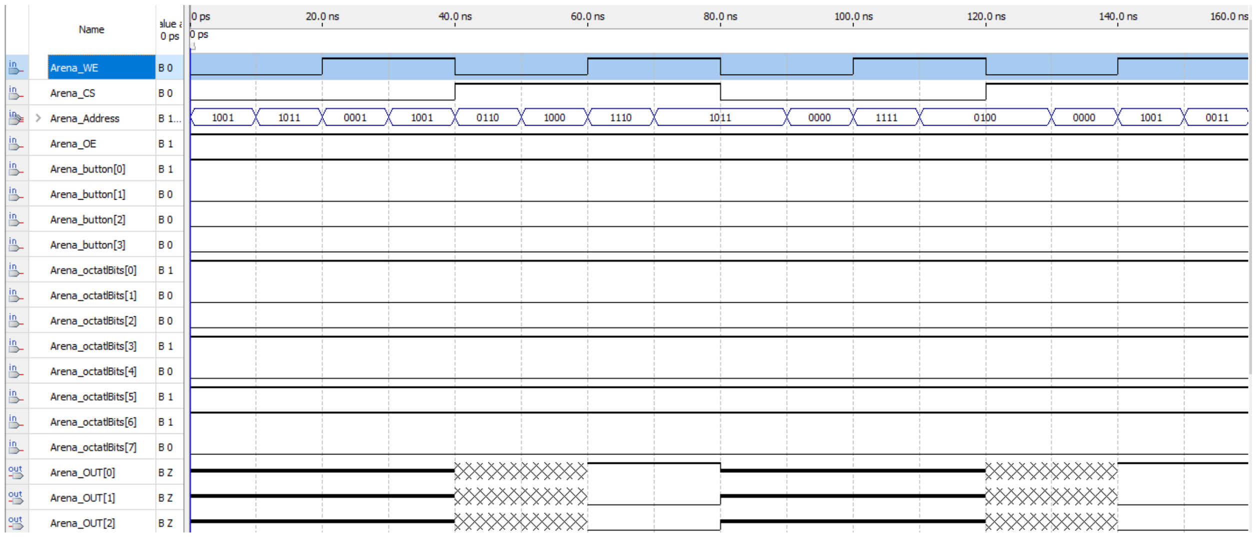


Figure 35: 16x4 SRAM Waveform

As expected, there is nothing at the output since output is disabled.

<u>16x32 SRAM</u>

The fifth and final simulation will be done for the 16x32 SRAM. The results should correspond with the truth table in Table **5**. Below in Figure **36** is the results from the Waveform file. Its split into two pictures since it cannot all be squeezed into one photo



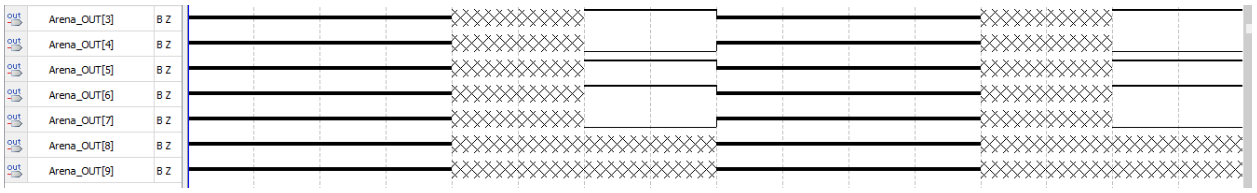


Figure 36: 16x4 SRAM Waveform

In this case, there are 8 switches set (the octal bits) with random inputs (0s or 1s). Button 0 has been set to 1 (indicating its being pressed) and button 1 has been set to 0 (indicating its not being pressed). Now remember this is using the same functionality as the 16x4 since its built out of the same SRAM cells. So, output enable is high, so the output will give us something rather than nothing. Then whenever write enable is high and chip select is high, we can see the Arena_OUT bits are written to. Now notice bits Arena_OUT[0].. Arena_OUT[7]. They get a value written to them. Arena_OUT[8] and Arena_OUT[9] is not written to. The reason is because remember, button0 corresponds to bits 0-7, button1 corresponds to 8-15. Button0 is “pressed”, so the values are assigned to the first octet. Since button1 is not “pressed”, the values are **NOT** assigned to the second octet. So that confirms my design works as intended.

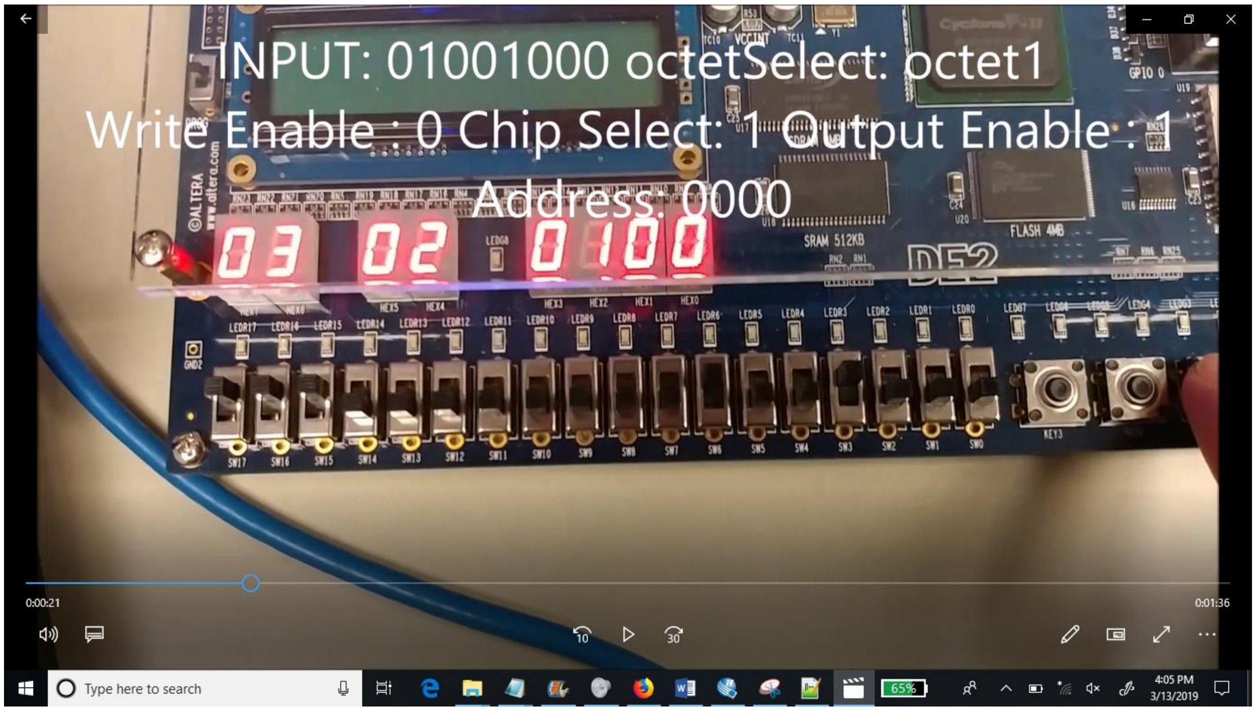
16x4 SIGNED SRAM

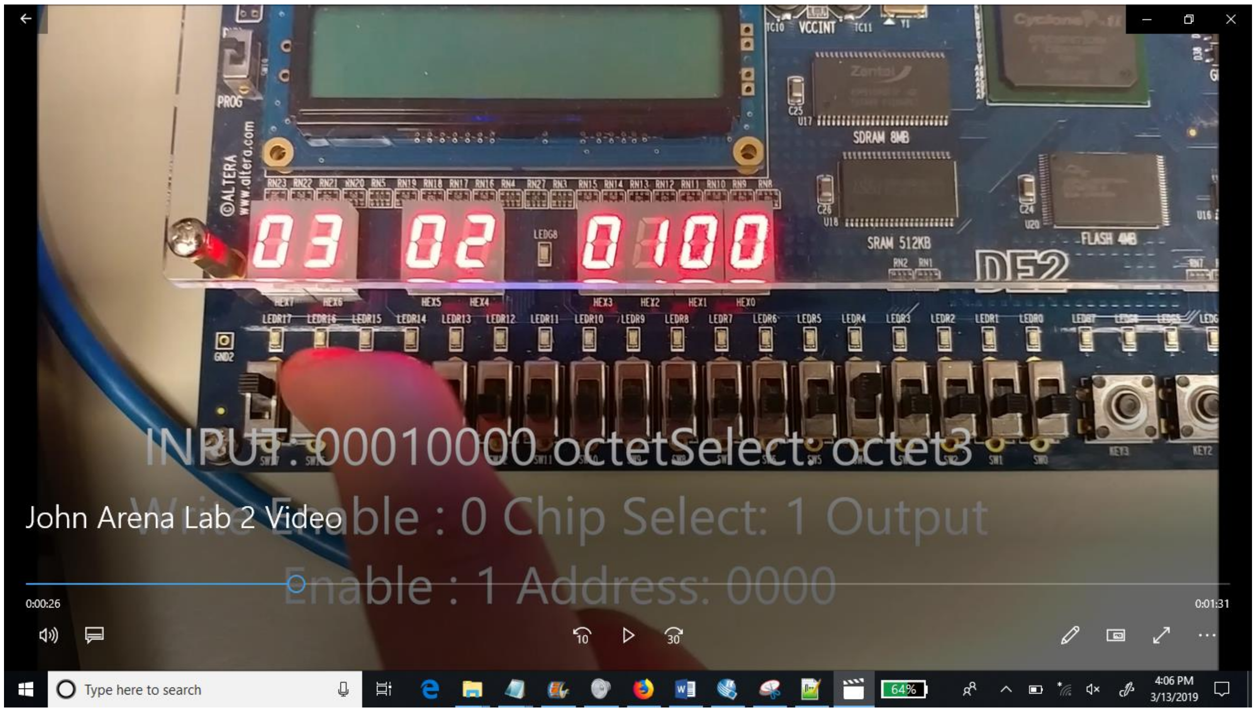
I did not include a simulation for this, as again this is the same as the 16x4 SRAM, with just some added functions to the decoder.

Section 4) Demonstration Pictures

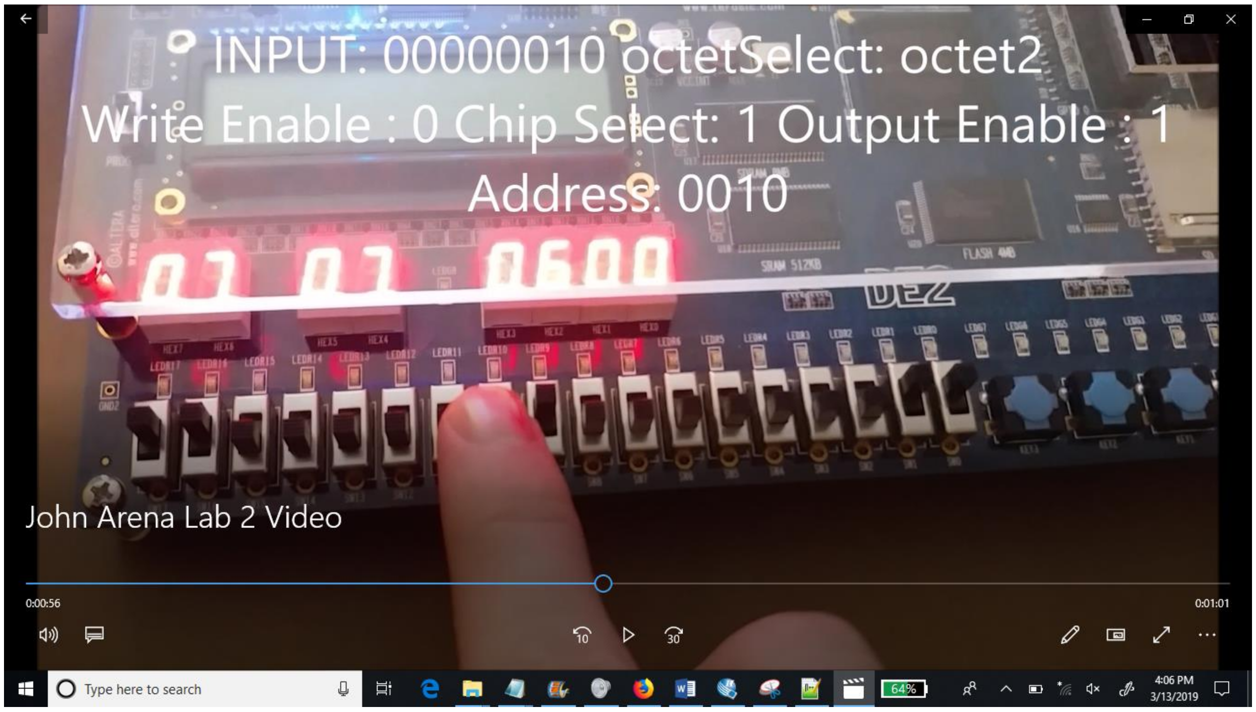
(Not every case is shown in the pictures, all cases shown in video)

16x32 SRAM

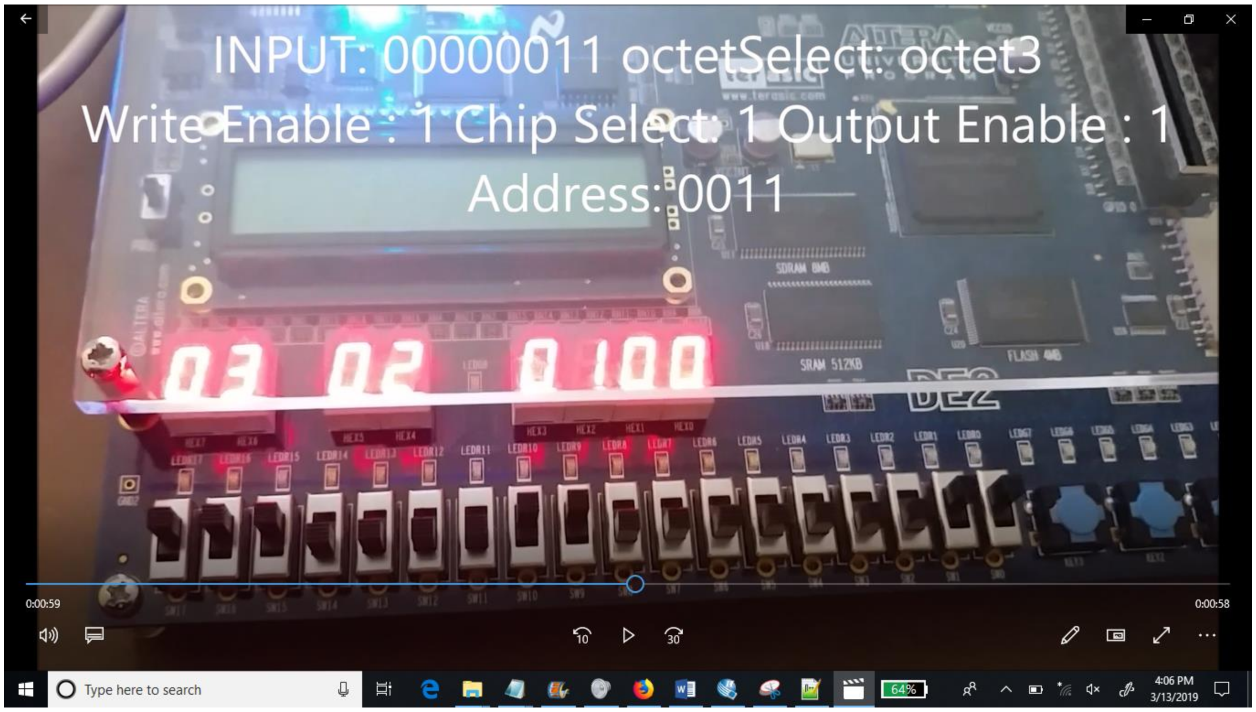

INPUT: 01001000 octetSelect: octet1
Write Enable : 0 Chip Select: 1 Output Enable : 1
Address: 0000
0:00:21
0:01:36
Type here to search
4:05 PM
3/13/2019



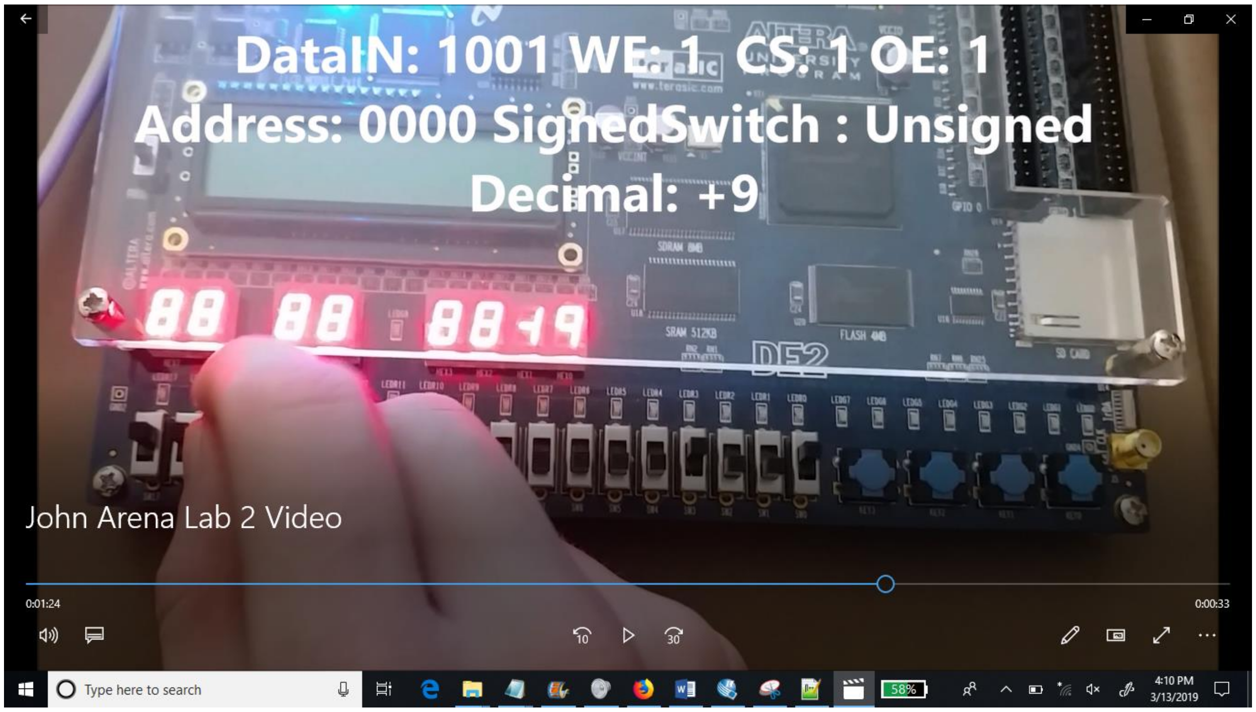
INPUT: 00010000 octetSelect: octet3
Write Enable : 0 Chip Select: 1 Output Enable : 1 Address: 0000

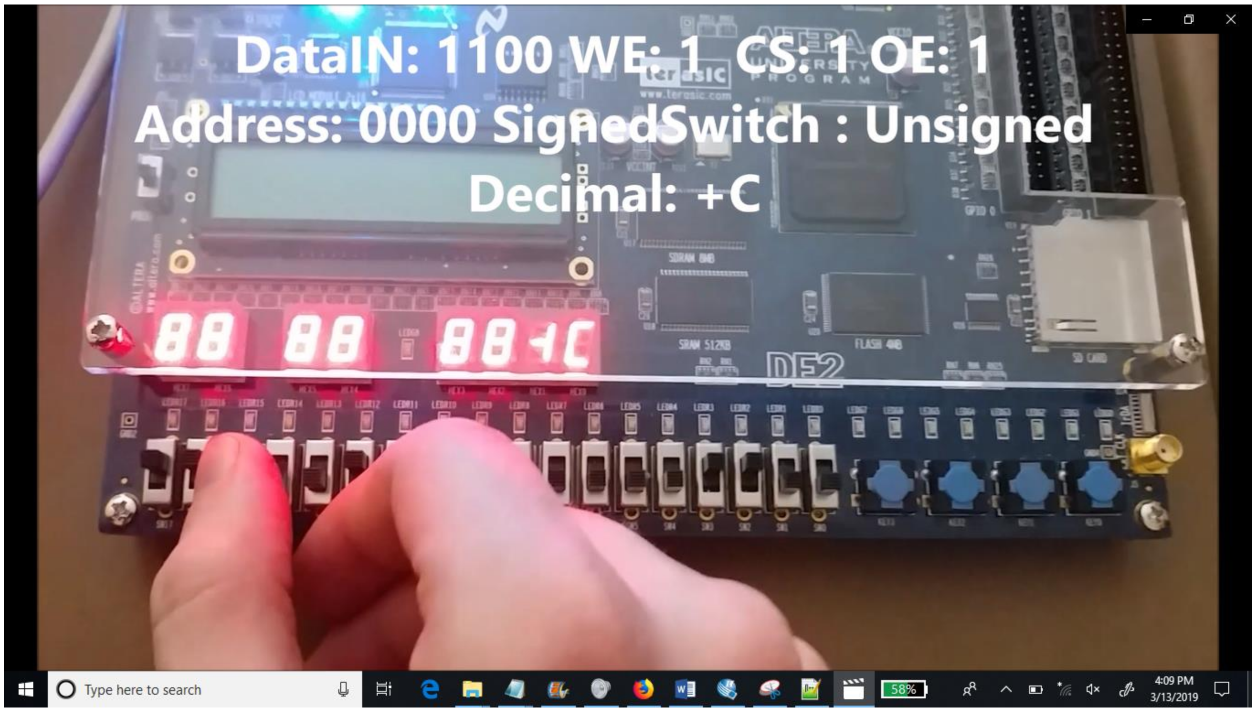


INPUT: 00000010 octetSelect: octet2
Write Enable : 0 Chip Select: 1 Output Enable : 1
Address: 0010

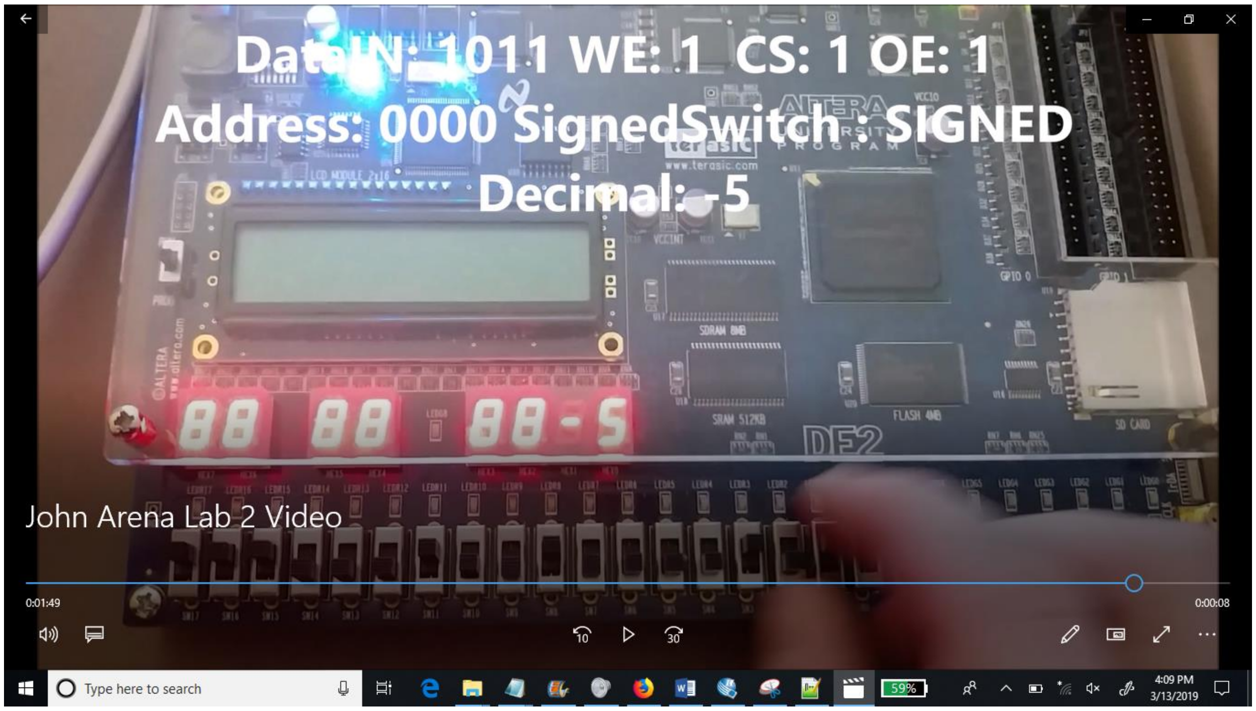

INPUT: 00000011 octetSelect: octet3
Write Enable : 1 Chip Select: 1 Output Enable : 1
Address: 0011
03 02 0 100
0:00:59
0:00:58
Type here to search
64%
4:06 PM
3/13/2019

16x4 SIGNED SRAM

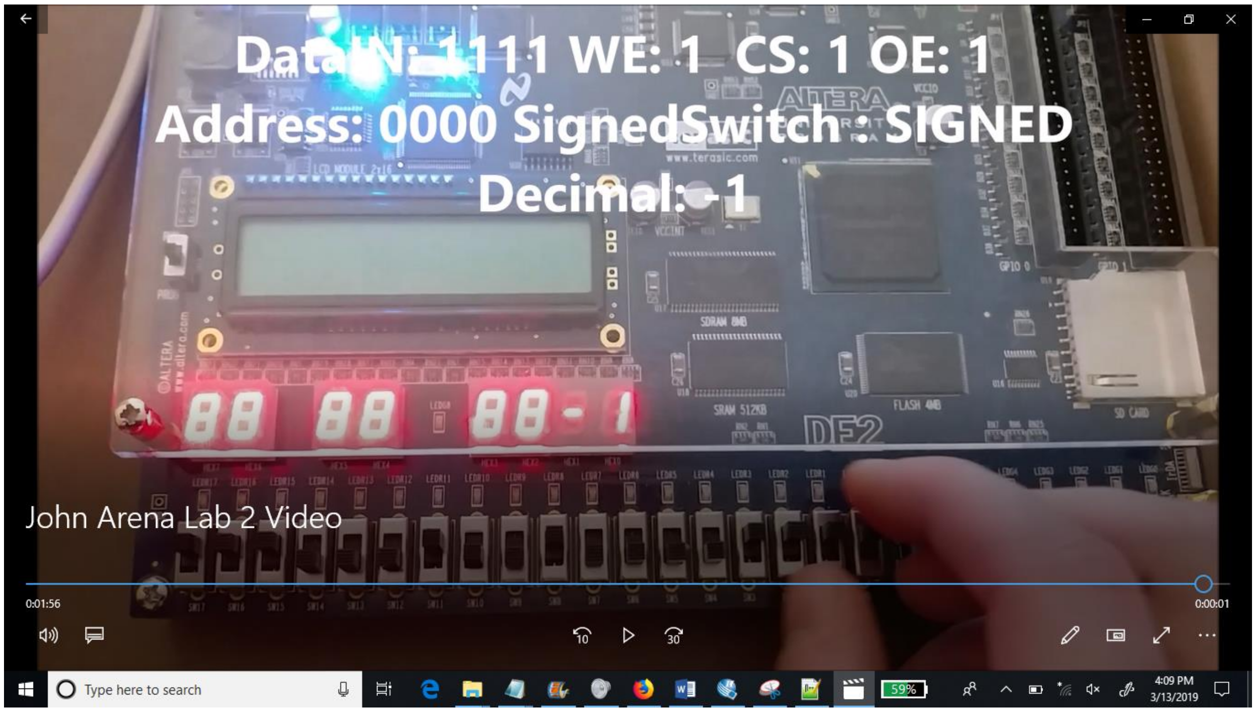




DataIN: 1100 WE: 1 CS: 1 OE: 1
Address: 0000 SignedSwitch : Unsigned
Decimal: +C
Type here to search
58%
4:09 PM
3/13/2019



DataIN: 1011 WE: 1 CS: 1 OE: 1
Address: 0000 SignedSwitch : SIGNED
Decimal: -5
John Arena Lab 2 Video
0:01:49
0:00:08
Type here to search
59%
4:09 PM
3/13/2019



Section 5) Conclusion

In this lab I designed various circuits, the main ones being a 16x4 SRAM, a 16x32 SRAM and a 16x4 Signed and Unsigned SRAM. I observed some things such as the differences between latches and flip flops. I learned how the data is stored in these, and how we can use these to build various things. I then proceeded to build an SRAM cell and observed its capabilities and how they can be extended to a 4 bit cell, all the way to 32 bit. For the 16x32, I would say I figured out a creative way to enter the 32bit number using only 4 switches utilizing the 4 buttons provided on the board, buttons 0-3. I also learned how to write VHDL code to use segment seven displays by using a decoder, which I didn't think would be a component to use and saw how actually it made it really easy to do. After learning that, the signed 16x4 SRAM was pretty straight forward after realizing the strength of the 4 to 7 segment display decoder. I would say one thing I learned is to pay more attention to the board, as at one point I was debugging an issue with the 16x32 SRAM because I thought button 0 was button 3, button 1 being button 2, and so far and so forth.