John Arena

Professor Gertner

CSC 342/343

Lab 3

Due 4/1/19

Spring 2019

Table of Contents

Lab report 90% complete

Section 1) Objective

For this lab, I will be extending my knowledge of VHDL, ModelSim and Quartus by developing a circuit that performs bitwise operations, bit shifts, bit rotations, and a few other operations from the MIPS processor. These functions will be determined by an opcode, a selector which determines which operation to perform. So, the objectives are below:

- Design bitwise operation such as NOT, OR, AND, etc.
- Design an opcode selector (aka a decoder) that will choose which operation to perform. Operation will be confirmed by a button on the FPGA board.
- Design operations to use 6-bit inputs. Output will be shown by using LED's on the FPGA board.
- Verifying the designs correctness using waveform simulations
- Programming pin assignments for the board

Section 2) Description and Specifications

Bitwise AND

The first circuit I will be designing is a bitwise AND. Before doing that, let's define a bitwise operation first. A bitwise operator "is an operator used to perform bitwise operations on bit patterns or binary numerlas that involve the manipulation of individual bits". They are typically used in "communication stacks where the individual bits in the header attached to the data signify important information."[*https://www.techopedia.com/definition/3467/bitwise-operator*]

With that said, again I will be designing bitwise AND. The AND gate functions as follows: The output is only 1 when all inputs are 1. Otherwise, it's 0. Below in Table 1 describes its functionality.

X	**Y**	**Out**
0	**0**	**0**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

Table 1: AND Gate Truth Table

From this, we can derive the Boolean expression of an and gate from table 1. We get the following equation in equation **1** below.

$$AND_{Out} = X * Y$$

Equation 1: Out of AND Gate.

So the idea as follows. Let's say we have two inputs as follows: X = 010. Y = 110. These through an AND bitwise operation would AND each corresponding bit with each other. For example, the 0th bit of each one will be AND'd together and become the 0th bit of the output. So X*Y in this will be 010, following the truth table in Table 1.

This is a straightforward implementation in VHDL. Below in Figure 1 is the VHDL code I created for the bitwise AND.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 3 - Spring 2019 - Due 3/30/19
-- Arena_bitwiseand.vhd
Library ieee;
use ieee.std_logic_1164.all;

entity Arena_bitwiseand is
  port(
    Arena_a_andIN,Arena_b_andIN: in std_logic_vector(5 downto 0);
    Arena_result_andOUT: out std_logic_vector (5 downto 0);
       Arena_andSelect: in std_logic
);
end Arena_bitwiseand;

architecture Arena_bitwiseand_arch of Arena_bitwiseand is
begin
      process(Arena_andSelect)
            begin
                  if(Arena_andSelect = '1') then
                        Arena_result_andOUT <= Arena_a_andIN and
Arena_b_andIN;
                  else
                        null;
                  end if;
      end process;
end Arena_bitwiseand_arch;
```

Figure 1: Bitwise AND VHDL Code

Below in Figure 2 is it's symbol I created for it.

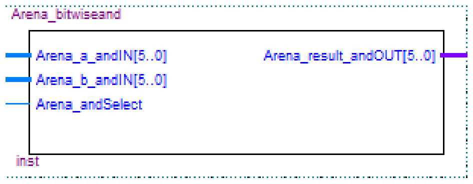


Figure 2: Bitwise AND Symbol

One thing to notice is the andSelect input. This input will be coming from our opcode selector (decoder). When the appropriate opcode is selected for bitwise AND, it will output and the input will go to the input in this, andSelect. That is why I check if andSelect is equal to 1 here, since that means we are choosing the and operation. Otherwise, it will do nothing. On the next page in Figure 2 is an illustration of how it should work

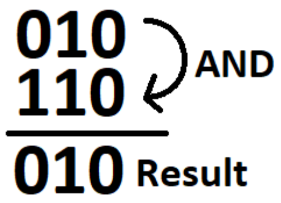


Figure 2: Bitwise AND illustration

As you can see, it each bit gets and'd with each other. Bit 0 and Bit 0 of each one gets and'd, bit 1 and bit 2. As you can see, the result follows the truth table.

Bitwise OR

The second circuit I will be designing is a bitwise OR. The OR gate functions as follows: The output is only 0 when all inputs are 0. Otherwise, it's 1. Below in Table 2 describes its functionality.

X	**Y**	**Out**
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**1**

Table 2: OR Gate Truth Table

From this, we can derive the Boolean expression of an and gate from table 1. We get the following equation in equation **2** below.

$$OR_{Out} = X + Y$$

Equation 2: Out of OR Gate.

So the idea as follows. Let's say we have two inputs as follows: X = 010. Y = 110. These through an OR bitwise operation would OR each corresponding bit with each other. For example, the 0^{th} bit of each one will be OR'd together and become the 0^{th} bit of the output. So X+Y in this will be 110, following the truth table in Table 2.

This is a straightforward implementation in VHDL. Below in Figure 3 is the VHDL code I created for the bitwise OR.

```vhdl
--John Arena
Library ieee;
use ieee.std_logic_1164.all;

entity Arena_bitwiseor is
  port(
    Arena_a_orIN,Arena_b_orIN: in std_logic_vector(5 downto 0);
    Arena_result_orOUT: out std_logic_vector (5 downto 0);
      Arena_orSelect: in std_logic
);
end Arena_bitwiseor;

architecture Arena_bitwiseor_arch of Arena_bitwiseor is
begin
process(Arena_orSelect)
      begin
            if (Arena_orSelect = '1') then
                  Arena_result_orOUT <= Arena_a_orIN or Arena_b_orIN;
            else
                  null;
            end if;
end process;
end Arena_bitwiseor_arch;
```

Figure 3: Bitwise OR VHDL Code

Below in Figure 2 is it's symbol I created for it.

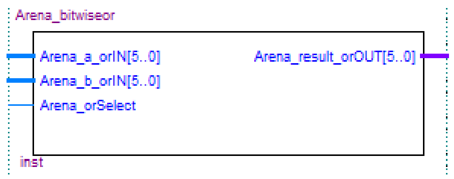


Figure 4: Bitwise OR Symbol

Similar to the and bitwise, notice this circuit has an orSelect.. This input will be coming from our opcode selector (decoder). When the appropriate opcode is selected for bitwise OR, it will output and the input will go to the input in this, orSelect. That is why I check if orSelect is equal to 1 here, since that means we are choosing the or operation. Otherwise, it will do nothing. Below in Figure 5 is an illustration of how it should work

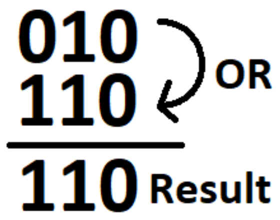


Figure 5: Bitwise OR illustration

As you can see, it each bit gets or'd with each other. Bit 0 and Bit 0 of each one gets or'd, bit 1 and bit 2. As you can see, the result follows the truth table.

Bitwise XOR

The third circuit I will be designing is a bitwise XOR. The XOR gate functions as follows: The output is only 0 when all inputs are equal to each other. Otherwise, it's 1. Below in Table 2 describes its functionality.

X	**Y**	**Out**
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**

1	**1**	**0**

Table 3: XOR Gate Truth Table

From this, we can derive the Boolean expression of an and gate from table 3. We get the following equation in equation **3** below.

$$XOR_{Out} = X \oplus Y$$

Equation 3: Out of XOR Gate.

So the idea as follows. Let’s say we have two inputs as follows: X = 010. Y = 110. These through an XOR bitwise operation would OR each corresponding bit with each other. For example, the 0th bit of each one will be XOR’d together and become the 0th bit of the output. So $X \oplus Y$ in this will be 100, following the truth table in Table 3.

This is a straightforward implementation in VHDL. Below in Figure 6 is the VHDL code I created for the bitwise XOR.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 3 - Spring 2019 - Due 3/30/19
-- Arena_bitwisexor.vhd
Library ieee;
use ieee.std_logic_1164.all;

entity Arena_bitwisexor is
  port(
    Arena_a_xorIN,Arena_b_xorIN: in std_logic_vector(5 downto 0);
    Arena_result_xorOUT: out std_logic_vector (5 downto 0);
       Arena_xorSelect: in std_logic
       );
end Arena_bitwisexor;

architecture Arena_bitwisexor_arch of Arena_bitwisexor is
begin
      process(Arena_xorSelect)
            begin
                  if (Arena_xorSelect = '1') then
                        Arena_result_xorOUT <= Arena_a_xorIN xor
Arena_b_xorIN;
                  else
                        null;
                  end if;
      end process;
end Arena_bitwisexor_arch;
```

Figure 6: Bitwise OR VHDL Code

Below in Figure 7 is it's symbol I created for it.

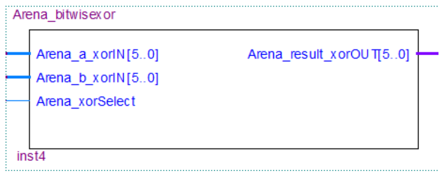


Figure 7: Bitwise XOR Symbol

Similar to the previous bitwise operations, notice this circuit has an xorSelect.. This input will be coming from our opcode selector (decoder). When the appropriate opcode is selected for bitwise XOR, it will output and the input will go to the input in this, xorSelect. That is why I check if xorSelect is equal to 1 here, since that means we are choosing the xor operation. Otherwise, it will do nothing. Below in Figure 8 is an illustration of how it should work

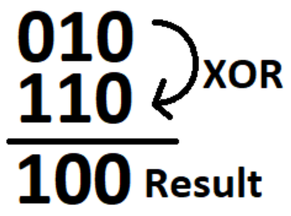


Figure 8: Bitwise XOR illustration

As you can see, it each bit gets xor'd with each other. Bit 0 and Bit 0 of each one gets or'd, bit 1 and bit 2. As you can see, the result follows the truth table.

Bitwise NOT

The fourth circuit I will be designing is a bitwise NOT. The NOT gate functions as follows: The output inverts the input. So if the input is 0, the output is 1, and if the input is 1, the output is 0. Below in Table 4 is the truth table.

X	**Out**
0	**1**
1	**0**

Table 4: NOT Gate Truth Table

From this, we can derive the Boolean expression of an and gate from table 4. We get the following equation in equation **4** below.

$$NOT_{Out} = X'$$

Equation 4: Out of NOT Gate.

So the idea as follows. Let’s say we have one inputs as follows: X = 010. These through an NOT bitwise operation would NOT each corresponding bit. For example, the 0th bit of each one will be NOT’d and become the 0th bit of the output. So X’ in this will be 101, following the truth table in Table 4.

This is a straightforward implementation in VHDL. Below in Figure 9 is the VHDL code I created for the bitwise NOT.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 3 - Spring 2019 - Due 3/30/19
-- Arena_bitwisenot.vhd
Library ieee;
use ieee.std_logic_1164.all;

entity Arena_bitwisenot is
  port(
    Arena_a_notIN: in std_logic_vector(5 downto 0);
    Arena_result_notOUT: out std_logic_vector (5 downto 0);
```

```
       Arena_notSelect: in std_logic
);
end Arena_bitwisenot;

architecture Arena_bitwisenot_arch of Arena_bitwisenot is
begin
      process(Arena_notSelect)
            begin
                  if (Arena_notSelect = '1') then
                        Arena_result_notOUT <= NOT Arena_a_notIN;
                  else
                        null;
                  end if;
      end process;
end Arena_bitwisenot_arch;
```

Figure 9: Bitwise OR VHDL Code

Below in Figure 10 is it's symbol I created for it.

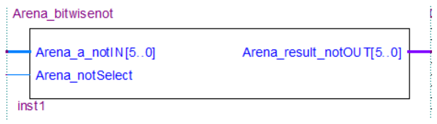


Figure 10: Bitwise NOT Symbol

Similar to the previous bitwise operations, notice this circuit has an notSelect.. This input will be coming from our opcode selector (decoder). When the appropriate opcode is selected for bitwise NOT, it will output and the input will go to the input in this, notSelect. That is why I check if notSelect is equal to 1 here, since that means we are choosing the not operation. Otherwise, it will do nothing. Below in Figure 11 is an illustration of how it should work

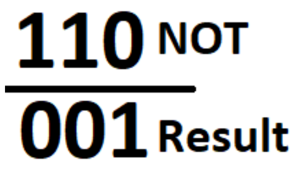


Figure 11: Bitwise NOT illustration

As you can see, it each bit gets not'd with each other. Bit 0 gets not'd, bit 1 and bit 2. As you can see, the result follows the truth table.

Bitwise Bit Shift Right

The fifth circuit I will be designing is a bitwise bit shift right. The bit shift right operator functions as follows: The output takes the input and shifts it to the right by one bit. The right most bit is deleted, and the new bit on the most left side is set to 0.

So, the idea as follows. Let's say we have one inputs as follows: X = 010. These through an BSR(acronym of bit shift right) bitwise operation would shift right each corresponding bit. So the output will become 001.

This is a straightforward implementation in VHDL. Below in Figure 9 is the VHDL code I created for the bitwise NOT.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 3 - Spring 2019 - Due 3/30/19
-- Arena_bitshiftright.vhd
Library ieee;
use ieee.std_logic_1164.all;

entity Arena_bitshiftright is
  port(
    Arena_a_bsrIN: in std_logic_vector(5 downto 0); -- Bit shift right in
    Arena_result_bsrOUT: out std_logic_vector(5 downto 0); -- Bit shift right
out
       Arena_bsrSelect: in std_logic
  );
end Arena_bitshiftright;

architecture Arena_bitshiftright_arch of Arena_bitshiftright is
begin
process(Arena_bsrSelect)
      begin
            if(Arena_bsrSelect = '1') then
                  Arena_result_bsrOUT <=
to_stdlogicvector(to_bitvector(Arena_a_bsrIN) srl 1);
            else
                  null;
            end if;
end process;
end Arena_bitshiftright_arch;
```

Figure 12: Bitwise BSR VHDL Code

Below in Figure 13 is it's symbol I created for it.

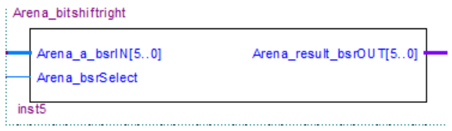


Figure 13: Bitwise BSR Symbol

Similar to the previous bitwise operations, notice this circuit has an bsrSelect.. This input will be coming from our opcode selector (decoder). When the appropriate opcode is selected for bitwise NOT, it will output and the input will go to the input in this, bsrSelect. That is why I check if bsrSelect is equal to 1 here, since that means we are choosing the not operation. Otherwise, it will do nothing. Below in Figure 14 is an illustration of how it should work

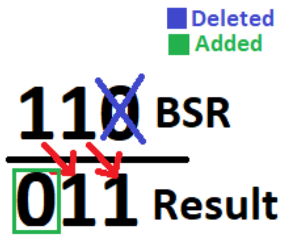


Figure 14: Bitwise BSR illustration

As you can see, it each bit gets shifted to the right with each other. The original bit 1 gets deleted. The others get shifted to the right, and bit 3 is now added a new value of 0.

Bitwise Bit Shift Left

The sixth circuit I will be designing is a bitwise bit shift left. The bit shift left operator functions as follows: The output takes the input and shifts it to the left by one bit. The right most bit is added, 0, and the most left bit is deleted.

So, the idea as follows. Let's say we have one inputs as follows: X = 010. These through an BSL(acronym of bit shift left) bitwise operation would shift left each corresponding bit. So the output will become 100.

This is a straightforward implementation in VHDL. Below in Figure 15 is the VHDL code I created for the bitwise BSL.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 3 - Spring 2019 - Due 3/30/19
-- Arena_bitshiftleft.vhd
Library ieee;
use ieee.std_logic_1164.all;

entity Arena_bitshiftleft is
  port(
    Arena_a_bslIN: in std_logic_vector(5 downto 0); -- Bit shift left
    Arena_result_bslOUT: out std_logic_vector(5 downto 0); -- Bit shift left
out
       Arena_bslSelect: in std_logic
  );
end Arena_bitshiftleft;

architecture Arena_bitshiftleft_arch of Arena_bitshiftleft is
begin
      process(Arena_bslSelect)
            begin
                  if (Arena_bslSelect = '1') then
                        Arena_result_bslOUT <=
to_stdlogicvector(to_bitvector(Arena_a_bslIN) sll 1);
                  else
                        null;
                  end if;
      end process;
end Arena_bitshiftleft_arch;
```

Figure 15: Bitwise BSR VHDL Code

Below in Figure 16 is it’s symbol I created for it.

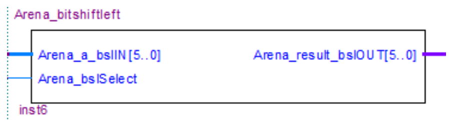


Figure 16: Bitwise BSL Symbol

Similar to the previous bitwise operations, notice this circuit has an bslSelect.. This input will be coming from our opcode selector (decoder). When the appropriate opcode is selected for bitwise BSL, it will output and the input will go to the input in this, bslSelect. That is why I check if bslSelect is equal to 1 here, since that means we are choosing the not operation. Otherwise, it will do nothing. Below in Figure 17 is an illustration of how it should work

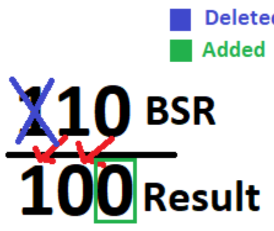


Figure 17: Bitwise BSL illustration

As you can see, it each bit gets shifted to the left with each other. The original bit 2 gets deleted. The others get shifted to the right, and bit 0 is now added a new value of 0.

Bitwise Bit Rotate left

The seventh circuit I will be designing is a bitwise bit rotate left. The bit rotate left operator functions as follows: The output takes the input and rotates it to the left by one bit. The left most bit becomes the right most bit after the rotation.

So, the idea as follows. Let's say we have one inputs as follows: X = 100. These through an BRL(acronym of bit rotate left) bitwise operation would rotate left each corresponding bit. So the output will become 001. The left most bit has become the right most bit

This is a straightforward implementation in VHDL. Below in Figure 18 is the VHDL code I created for the bitwise BSL.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 3 - Spring 2019 Due: 4/1/19
-- Arena_bitrotationleft.vhd
Library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Arena_bitrotationleft is
  port(
    Arena_a_brlIN: in std_logic_vector(5 downto 0); -- Bit rotation left in
       Arena_b_rlamt_brrIN: in std_logic_vector(5 downto 0); -- Bit rotation
left amount IN
    Arena_result_brlOUT: out std_logic_vector(5 downto 0); -- Bit rotation
out
       Arena_brlSelect: in std_logic
  );
end Arena_bitrotationleft;

architecture Arena_bitrotationleft_arch of Arena_bitrotationleft is
signal Arena_b_rlamt_integer : integer; -- Rotate Left integer value
declaration

begin
Arena_b_rlamt_integer <= to_integer(unsigned(Arena_b_rlamt_brrIN)); -- Assign
value
      process(Arena_brlSelect)
            begin
                  if(Arena_brlSelect = '1') then
                        Arena_result_brlOUT <=
to_stdlogicvector(to_bitvector(Arena_a_brlIN) rol Arena_b_rlamt_integer); --
Rotate left by amount
                  else
                        null;
```

```
                    end if;
            end process;
end Arena_bitrotationleft_arch;
```

Figure 18: Bitwise BRL VHDL Code

Below in Figure 19 is it's symbol I created for it.

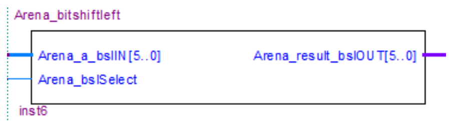


Figure 19: Bitwise BSR Symbol

Commented [j1]: Fix symbol

Similar to the previous bitwise operations, notice this circuit has an brlSelect.. This input will be coming from our opcode selector (decoder). When the appropriate opcode is selected for bitwise BRL, it will output and the input will go to the input in this, brlSelect. That is why I check if brlSelect is equal to 1 here in the VHDL code, since that means we are choosing the not operation. Otherwise, it will do nothing. Below in Figure 20 is an illustration of how it should work

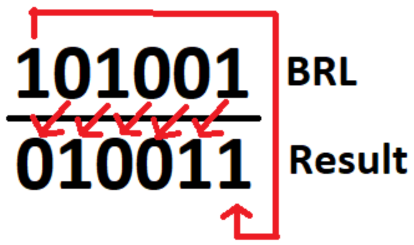


Figure 20: Bitwise BRL illustration

As you can see, it each bit gets rotated to the left with each other. Notice how the most left bit in the original binary number becomes the most right bit in the result.

<u>Bitwise Bit Rotate Right</u>

The eight circuit I will be designing is a bitwise bit rotate right. The bit rotate right operator functions as follows: The output takes the input and rotates it to the right by one bit. The right most bit becomes the left most bit.

So, the idea as follows. Let’s say we have one inputs as follows: X = 001. These through an BRR(acronym of bit rotate right) bitwise operation would rotate right each corresponding bit. So the output will become 100. The right most bit has become the left most bit.

This is a straightforward implementation in VHDL. Below in Figure 21 is the VHDL code I created for the bitwise BRR.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 3 - Spring 2019 Due: 4/1/19
-- Arena_bitrotationright.vhd
Library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Arena_bitrotationright is
  port(
    Arena_a_brrIN: in std_logic_vector(5 downto 0); -- Bit rotation right in
       Arena_b_rramt_brrIN: in std_logic_vector(5 downto 0); -- Bit rotation
right amount IN
    Arena_result_brrOUT: out std_logic_vector(5 downto 0); -- Bit rotation
out
       --Arena_b_shamt_integer: buffer integer;
       Arena_brrSelect: in std_logic
  );
end Arena_bitrotationright;

architecture Arena_bitrotationright_arch of Arena_bitrotationright is
signal Arena_b_rramt_integer : integer; -- Rotate Right integer value
declaration
begin
Arena_b_rramt_integer <= to_integer(unsigned(Arena_b_rramt_brrIN)); -- Assign
value
      process(Arena_brrSelect)
            begin
                  if(Arena_brrSelect = '1') then
                        Arena_result_brrOUT <=
to_stdlogicvector(to_bitvector(Arena_a_brrIN) ror Arena_b_rramt_integer); --
Rotate by amount
                  else
                        null;
                  end if;
      end process;
end Arena_bitrotationright_arch;
```

Figure 21: Bitwise BRL VHDL Code

Below in Figure 22 is it’s symbol I created for it.

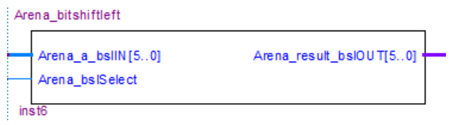


Figure 22: Bitwise BRR Symbol

Commented [j2]: Fix symbol

Similar to the previous bitwise operations, notice this circuit has an brrSelect.. This input will be coming from our opcode selector (decoder). When the appropriate opcode is selected for bitwise BRR, it will output and the input will go to the input in this, brrSelect. That is why I check if brrSelect is equal to 1 here in the VHDL code, since that means we are choosing the not operation. Otherwise, it will do nothing. Below in Figure 23 is an illustration of how it should work

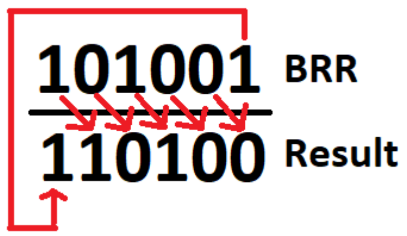


Figure 23: Bitwise BRL illustration

As you can see, it each bit gets rotated to the right with each other. Notice how the most right bit in the original binary number becomes the most left bit in the result.

Bitwise Set Less Than Unsigned

The ninth circuit I will be designing is a bitwise set less than unsigned. It operates as follows: It takes two binary inputs that represented an **UNSIGNED** value. Meaning 0->(2^N-1). If the first binary number is less than the second, it will return an output of 1, representing true. Otherwise 0 is returned, representing false.

So, the idea as follows. Let's say we have twos inputs as follows: X = 001, and Y = 100. These through an SLTU(acronym of set less than unsigned) bitwise operation would be compared, so X < Y. Is X < Y? Yes! X in decimal is 1, and Y in decimal is 4. (If these were signed numbers, the result may be different since the most significant bit is a sign bit in signed numbers). So the output would be set to 1.

This is a straightforward implementation in VHDL. Below in Figure 24 is the VHDL code I created for the bitwise SLTU.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 3 - Spring 2019 Due: 4/1/19
-- Arena_bitwise_setLessThanUnsigned.vhd
Library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Arena_bitwise_setLessThanUnsigned is
  port(
    Arena_a_sltuIN: in std_logic_vector(5 downto 0); -- Bit rotation left in
       Arena_b_sltuIN: in std_logic_vector(5 downto 0); -- Bit rotation left
amount IN
    Arena_result_sltuOUT: out std_logic; -- Bit rotation out
       Arena_sltuSelect: in std_logic
  );
end Arena_bitwise_setLessThanUnsigned;

architecture Arena_bitwise_setLessThanUnsigned_arch of
Arena_bitwise_setLessThanUnsigned is
signal Arena_a_sltuIN_integer : integer; -- Declaring variables for binary to
integer conversion
signal Arena_b_sltuIN_integer : integer; -- Declaring variables for binary to
integer conversion

begin
```

```
Arena_a_sltuIN_integer <= to_integer(unsigned(Arena_a_sltuIN)); -- Conversion
to integer
Arena_b_sltuIN_integer <= to_integer(unsigned(Arena_b_sltuIN)); -- Conversion
to integer
   process(Arena_sltuSelect, Arena_a_sltuIN_integer,
Arena_b_sltuIN_integer)
      begin
         if(Arena_sltuSelect = '1') then
            if(Arena_a_sltuIN_integer < Arena_b_sltuIN_integer)
then -- Check if less than, if true
               Arena_result_sltuOUT <= '1'; -- Set output to 1
            else
               Arena_result_sltuOUT <= '0'; -- Otherwise set
output to 0 if false
            end if;
         else
            null;
         end if;
   end process;
end Arena_bitwise_setLessThanUnsigned_arch;
```

Figure 24: Bitwise SLTU VHDL Code

Below in Figure 25 is it's symbol I created for it.

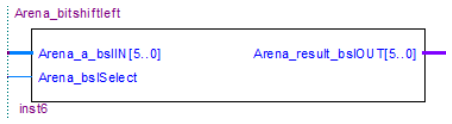


Figure 25: Bitwise SLTU Symbol

Commented [j3]: Fix symbol

Similar to the previous bitwise operations, notice this circuit has a sltuSelect. This input will be coming from our opcode selector (decoder). When the appropriate opcode is selected for bitwise SLTU, it will output and the input will go to the input in this, sltuSelect. That is why I check if sltuSelect is equal to 1 here in the VHDL code, since that means we are choosing the sltu operation. Otherwise, it will do nothing. On the next page in Figure 26 is an illustration of how it should work

Commented [j4]: Fix these in the others

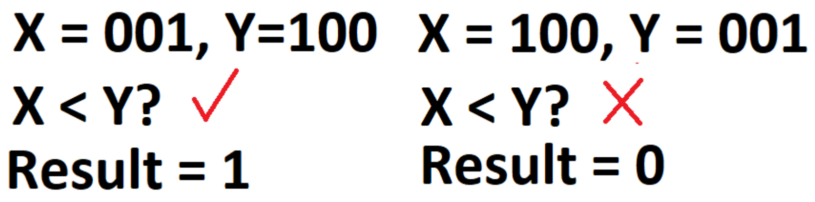


Figure 26: Bitwise SLTU illustration

Again, 001 is 1 and 100 is 4. 1 is less than 4 so that is true, where as 4 is not less than 1, so that is false.

Bitwise Set Less Than Signed

The tenth circuit I will be designing is a bitwise set less than signed. It operates as follows: It takes two binary inputs that represented as a **SIGNED** value. Meaning $-(2^{N-1} - 1)$ to $(2^{N-1} - 1)$. If the first binary number is less than the second, it will return an output of 1, representing true. Otherwise 0 is returned, representing false.

So, the idea as follows. Let's say we have twos inputs as follows: X = 001, and Y = 100. These through an SLT(acronym of set less than signed) bitwise operation would be compared, so X < Y. Is X < Y? **NO**! X in decimal is 1, and Y in decimal is -4. (If these were unsigned numbers, the result may be different since the most significant bit is not used as a signed bit). So the output would be set to 0.

This is a straightforward implementation in VHDL. Below in Figure 27 is the VHDL code I created for the bitwise SLT.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 3 - Spring 2019 Due: 4/1/19
-- Arena_bitwise_setLessThanSigned.vhd
Library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Arena_bitwise_setLessThanSigned is
```

```vhdl
  port(
    Arena_a_sltIN: in std_logic_vector(5 downto 0); -- Bit rotation left in
       Arena_b_sltIN: in std_logic_vector(5 downto 0); -- Bit rotation left
amount IN
    Arena_result_sltOUT: out std_logic; -- Bit rotation out
       Arena_sltSelect: in std_logic
  );
end Arena_bitwise_setLessThanSigned;

architecture Arena_bitwise_setLessThanSigned_arch of
Arena_bitwise_setLessThanSigned is
signal Arena_a_sltIN_integer : integer; -- Declaring variables for binary to
integer conversion
signal Arena_b_sltIN_integer : integer; -- Declaring variables for binary to
integer conversion

begin
Arena_a_sltIN_integer <= to_integer(signed(Arena_a_sltIN)); -- Conversion to
integer
Arena_b_sltIN_integer <= to_integer(signed(Arena_b_sltIN)); -- Conversion to
integer
      process(Arena_sltSelect, Arena_a_sltIN_integer, Arena_b_sltIN_integer)
            begin
                  if(Arena_sltSelect = '1') then
                        if(Arena_a_sltIN_integer < Arena_b_sltIN_integer)
then -- Check if less than, if true
                              Arena_result_sltOUT <= '1'; -- Set output to 1
                        else
                              Arena_result_sltOUT <= '0'; -- Otherwise set
output to 0 if false
                        end if;
                  else
                        null;
                  end if;
      end process;
end Arena_bitwise_setLessThanSigned_arch;
```

Figure 27: Bitwise SLTU VHDL Code

Below in Figure 28 is it's symbol I created for it.

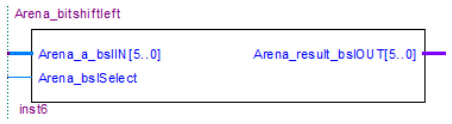


Figure 28: Bitwise SLT Symbol

Commented [j5]: Fix symbol

Similar to the previous bitwise operations, notice this circuit has a sltSelect. This input will be coming from our opcode selector (decoder). When the appropriate opcode is selected for bitwise SLT, it will output and the input will go to the input in this, sltSelect. That is why I check if sltSelect is equal to 1 here in the VHDL code, since that means we are choosing the slt operation. Otherwise, it will do nothing. Below in Figure 29 is an illustration of how it should work

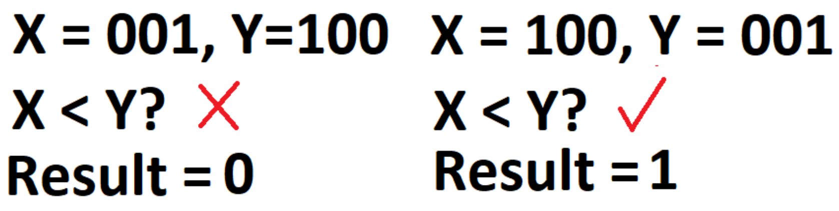


Figure 29: Bitwise SLTU illustration

Again, 001 is 1 and 100 is -4. 1 is not less than 4 so that is false, whereas -4 is less than 1, so that is true.

<u>REDESIGN</u>

Now this is good if you're designing these individually, but very bad if it's suppose to be a group of functions. So, my new design will combine a selector with all the operations. Each bit operation will be chosen depending on an opcode. Below in Figure 30 is my VHDL code of this design.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 3 - Spring 2019 - Due 3/30/19
-- Arena_bitwise.vhd
Library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Arena_bitwise is
  port(
    Arena_a_bitwiseIN,Arena_b_bitwiseIN: in std_logic_vector(5 downto 0);
    Arena_result_bitwiseOUT: out std_logic_vector (5 downto 0);
       Arena_opcode: in std_logic_vector (3 downto 0);
       Arena_buttonStart: in std_logic
);
end Arena_bitwise;

architecture Arena_bitwise_arch of Arena_bitwise is
signal Arena_b_sramt_integer : integer;--Declare and assign value
signal Arena_b_slamt_integer : integer; -- Shift Left integer value
declaration
signal Arena_b_rramt_integer : integer; -- Rotate Right integer value
declaration
signal Arena_b_rlamt_integer : integer; -- Rotate Left integer value
declaration
signal Arena_a_sltuIN_integer : integer; -- Declaring variables for binary to
integer conversion
signal Arena_b_sltuIN_integer : integer; -- Declaring variables for binary to
integer conversion

signal Arena_a_sltIN_integer : integer; -- Declaring variables for binary to
integer conversion
signal Arena_b_sltIN_integer : integer; -- Declaring variables for binary to
integer conversion

begin
Arena_b_sramt_integer <= to_integer(unsigned(Arena_b_bitwiseIN)); -- Assign
value
Arena_b_slamt_integer <= to_integer(unsigned(Arena_b_bitwiseIN)); -- Assign
value
Arena_b_rramt_integer <= to_integer(unsigned(Arena_b_bitwiseIN)); -- Assign
value
Arena_b_rlamt_integer <= to_integer(unsigned(Arena_b_bitwiseIN)); -- Assign
value
```

```
Arena_a_sltuIN_integer <= to_integer(unsigned(Arena_a_bitwiseIN)); --
Conversion to integer
Arena_b_sltuIN_integer <= to_integer(unsigned(Arena_b_bitwiseIN)); --
Conversion to integer

Arena_a_sltIN_integer <= to_integer(signed(Arena_a_bitwiseIN)); -- Conversion
to integer
Arena_b_sltIN_integer <= to_integer(signed(Arena_b_bitwiseIN)); -- Conversion
to integer
process(Arena_buttonStart, Arena_opcode, Arena_a_sltuIN_integer,
Arena_b_sltuIN_integer, Arena_a_sltIN_integer, Arena_b_sltIN_integer)
      begin
            if(Arena_buttonStart = '0') then
                        case Arena_opcode is
                              when "0000" =>
                              Arena_result_bitwiseOUT <= not
Arena_a_bitwiseIN;
                              when "0001" =>
                              Arena_result_bitwiseOUT <= Arena_a_bitwiseIN or
Arena_b_bitwiseIN;
                              when "0010" =>
                              Arena_result_bitwiseOUT <= Arena_a_bitwiseIN
and Arena_b_bitwiseIN;
                              when "0011" =>
                              Arena_result_bitwiseOUT <= Arena_a_bitwiseIN
xor Arena_b_bitwiseIN;
                              when "0100" =>
                              Arena_result_bitwiseOUT <=
to_stdlogicvector(to_bitvector(Arena_a_bitwiseIN) srl Arena_b_sramt_integer);
                              when "0101" =>
                              Arena_result_bitwiseOUT <=
to_stdlogicvector(to_bitvector(Arena_a_bitwiseIN) sll Arena_b_slamt_integer);
                              when "0110" =>
                              Arena_result_bitwiseOUT <=
to_stdlogicvector(to_bitvector(Arena_a_bitwiseIN) ror Arena_b_rramt_integer);
-- Rotate by amount
                              when "0111" =>
                              Arena_result_bitwiseOUT <=
to_stdlogicvector(to_bitvector(Arena_a_bitwiseIN) rol Arena_b_rlamt_integer);
-- Rotate left by amount
                              when "1000" =>
                                    if(Arena_a_sltuIN_integer <
Arena_b_sltuIN_integer) then -- Check if less than, if true
                                          Arena_result_bitwiseOUT <=
"000001"; -- Set output to 1
                                    else
                                          Arena_result_bitwiseOUT <=
"000000"; -- Otherwise set output to 0 if false
                                    end if;
                              when "1001" =>
                                    if(Arena_a_sltIN_integer <
Arena_b_sltIN_integer) then -- Check if less than, if true
                                          Arena_result_bitwiseOUT <=
"000001"; -- Set output to 1
                                    else
                                          Arena_result_bitwiseOUT <=
"000000"; -- -- Otherwise set output to 0 if false
```

```
                                                end if;
                                        When "1111" =>
                                        Arena_result_bitwiseOUT <= "000000";
                                        when others =>
                                        null;
                                end case;
                else
                        null;
                end if;
        end process;
end Arena_bitwise_arch;
```

Figure 30: Bitwise Operation Design in VHDL

As can be seen in this design, there are four inputs and one output. The first two inputs, A and B are 6 bit vectors (6 bit inputs) for the bit operations. The next input is the opcode input. It’s a 4 bit vector (4 bit input), for a total of 2^4 = 16 possible operations. This can be increased if needed, for example, 2^5 = 32 possible operations. The fourth input is the button start input, this is used in the code to tell us whether we should perform an operation or not and the specific operation depends on its opcode. If it’s not pressed, it will not perform. The output is a 6 bit vector (6 outputs), which will go to their own respective LEDs on the FPGA board.

Finally notice the case statement. Each does an operation depending on the opcode. Opcode case ‘1111’, is a special case. This is just used to reset the LED’s so they can be used again after an operation, otherwise the LED’s will not be turned off. A much **better** implementation can be used to solve this. For each operation, we can just have a wait statement after it performs a bitwise operation, followed by the LED clearing. This was not asked for, so I did not implement it this way. But if I were to, it could be for example wait 6 seconds then clear.

Below in figure 31 is the circuit symbol.

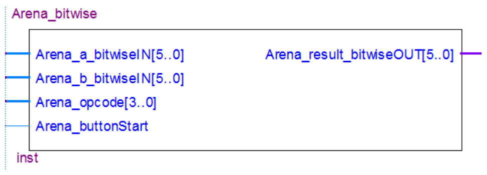


Figure 31: Bitwise Operations Schematic Symbol

Below in figure 32 is the final circuit schematic.

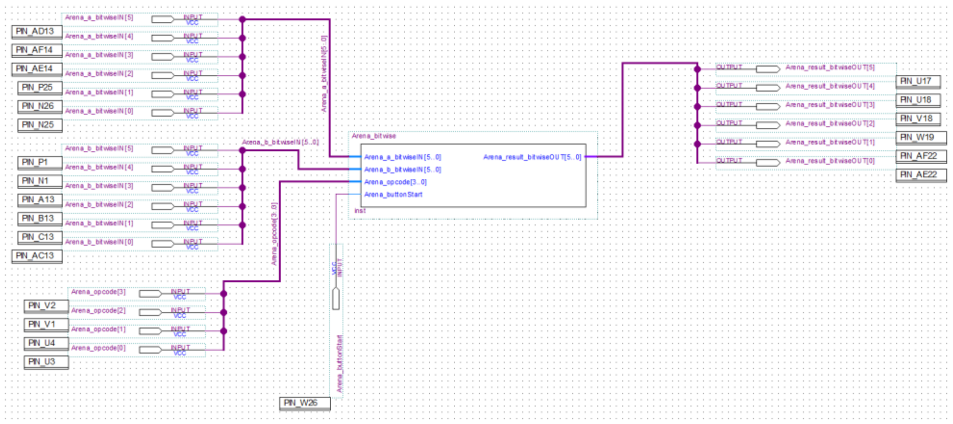
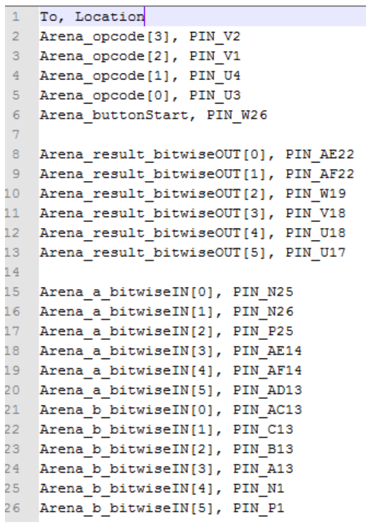


Figure 32: Bitwise Operations Final Schematic

The inputs and outputs have buslines connected appropriately (along with a single wire to a pin for the single input button). They are assigned to the appropriate pins (pin assignments are shown on the next page in Figure 33). Another good thing about this design is the fact the pin assignments are much easier. Assigning pins to the previous design would be problematic, as inputs can be assigned the same pins within the same file.



```
To, Location
Arena_opcode[3], PIN_V2
Arena_opcode[2], PIN_V1
Arena_opcode[1], PIN_U4
Arena_opcode[0], PIN_U3
Arena_buttonStart, PIN_W26

Arena_result_bitwiseOUT[0], PIN_AE22
Arena_result_bitwiseOUT[1], PIN_AF22
Arena_result_bitwiseOUT[2], PIN_W19
Arena_result_bitwiseOUT[3], PIN_V18
Arena_result_bitwiseOUT[4], PIN_U18
Arena_result_bitwiseOUT[5], PIN_U17

Arena_a_bitwiseIN[0], PIN_N25
Arena_a_bitwiseIN[1], PIN_N26
Arena_a_bitwiseIN[2], PIN_P25
Arena_a_bitwiseIN[3], PIN_AE14
Arena_a_bitwiseIN[4], PIN_AF14
Arena_a_bitwiseIN[5], PIN_AD13
Arena_b_bitwiseIN[0], PIN_AC13
Arena_b_bitwiseIN[1], PIN_C13
Arena_b_bitwiseIN[2], PIN_B13
Arena_b_bitwiseIN[3], PIN_A13
Arena_b_bitwiseIN[4], PIN_N1
Arena_b_bitwiseIN[5], PIN_P1
```

Figure 33: Bitwise Operations Pin Assignments.

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.

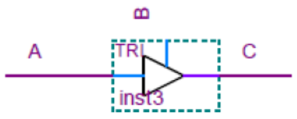
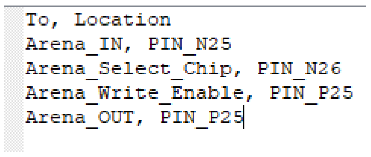


Figure 9: Tristate Logic Buffer

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.

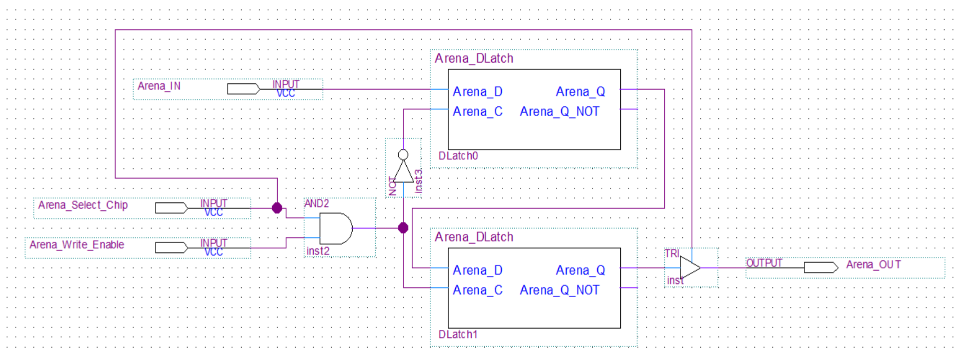


Figure 11: SRAM Cell Design

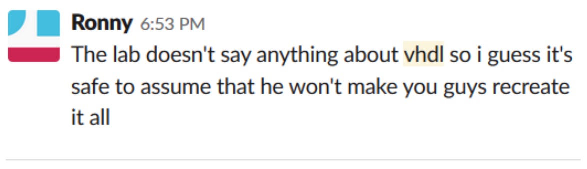
TRUTH TABLE NEEDED

Section 3) Simulations

Bitwise AND

The first simulation will be done for the bitwise AND.

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 bitwise AND waveform. Our results should correspond with the truth table in Table **1**.

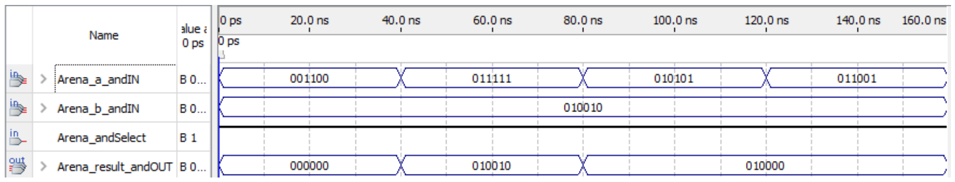


Figure 30: AND Waveform

Looking at the figures above, we can see our design for the AND is correct. We see comparing the waveform to the truth table. We know whenever both bits are not equal to 1, the output will be 0. If they are both equal to 1, the output will be 1. Looking at the first example, A = 001100. B is 010010. There is no situation where each bit put through bitwise AND will produce 1, so the output will be 000000. As we can see, that is the case in Figure 30. The rest of the examples can be seen in the waveform.

<u>Bitwise OR</u>

The second simulation will be done for the bitwise OR.

Figure **31** below is shows results of the bitwise OR waveform. Our results should correspond with the truth table in Table **2**.

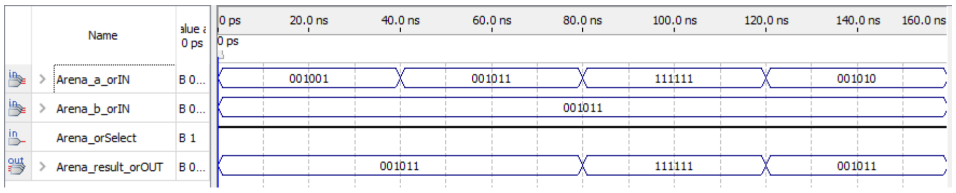


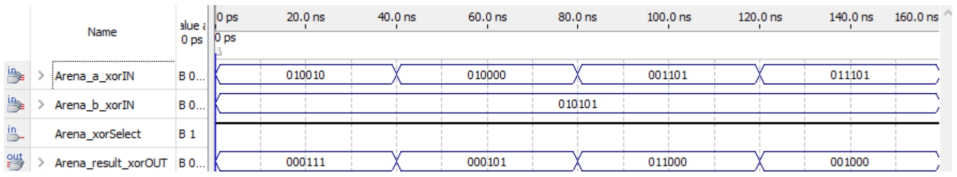
Figure 31: OR Waveform

Looking at the figures above, we can see our design for the OR is correct. We see comparing the waveform to the truth table. We know the output will only be 0 when the inputs are all 0. Looking at the first example, A = 001001. B is 001011. That output is 001011. As we can see, that is the case in Figure 31. The rest of the examples can be seen in the waveform.

<u>**Bitwise XOR**</u>

The third simulation will be done for the bitwise XOR.

Figure **32** on the next page is shows results of the bitwise XOR waveform. Our results should correspond with the truth table in Table **3**.



Name	Value at 0 ps	0 ps – 40.0 ns	40.0 ns – 80.0 ns	80.0 ns – 120.0 ns	120.0 ns – 160.0 ns
Arena_a_xorIN	B 0...	010010	010000	001101	011101
Arena_b_xorIN	B 0...	010101	010101	010101	010101
Arena_xorSelect	B 1				
Arena_result_xorOUT	B 0...	000111	000101	011000	001000

Figure 32: XOR Waveform

Looking at the figures above, we can see our design for the XOR is correct. We see comparing the waveform to the truth table. We know the output will only be 0 when the inputs are all equal. Looking at the first example, A = 010010. B is 010101. That output is 000111. As we can see, that is the case in Figure 33. The rest of the examples can be seen in the waveform.

<u>**Bitwise NOT**</u>

The fourth simulation will be done for the bitwise NOT.

Figure **33** below shows results of the bitwise NOT waveform. Our results should correspond with the truth table in Table **4**.

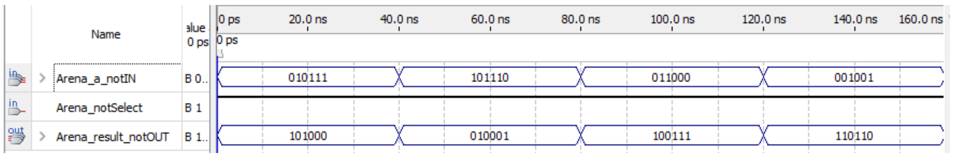


Figure 33: NOT Waveform

Looking at the figures above, we can see our design for the NOT is correct. We see comparing the waveform to the truth table. We know the output will only be 0 when the input is 1, and vice versa. Looking at the first example, A = 010110. The output is 101000. As we can see, that is the case in Figure 33. The rest of the examples can be seen in the waveform.

<u>Bitwise Bit Shift Right</u>

The fifth simulation will be done for the bitwise bit shift right.

Figure **34** below is shows results of the bitwise bsr waveform. Our results should correspond with Figure 14.

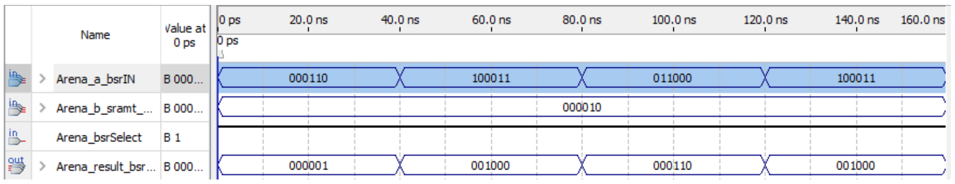


Figure 34: BSR Waveform

Looking at the figures above, we can see our design for the bsr is correct. We see comparing the waveform to the truth table. Looking at the first example, A = 000110. B is 000010. B is the shift amount. In decimal, that value is 2. So, this should shift A 2 bits to the right. As we can see, that is the case in Figure 34. The rest of the examples can be seen in the waveform.

Bitwise Bit Shift Left

The sixth simulation will be done for the bitwise bit shift left.

Figure **35** on the next page is shows results of the bitwise bsl waveform. Our results should correspond with Figure 17.

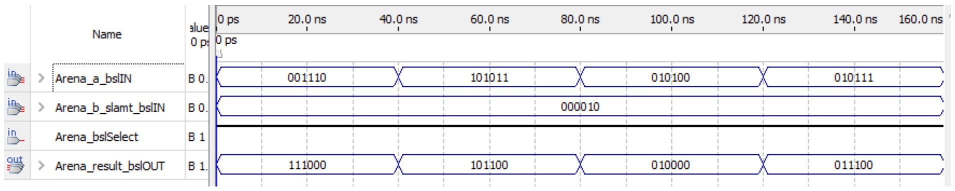


Figure 35: BSL Waveform

Looking at the figures above, we can see our design for the bsl is correct. We see comparing the waveform to the truth table. Looking at the first example, A = 001110. B is 000010. B is the shift amount. In decimal, that value is 2. So, this should shift A 2 bits to the left. As we can see, that is the case in Figure 35. The rest of the examples can be seen in the waveform.

Bitwise Bit Rotate Left

The seventh simulation will be done for the bitwise bit rotate left.

Figure **36** below is shows results of the bitwise brl waveform. Our results should correspond with Figure 20.

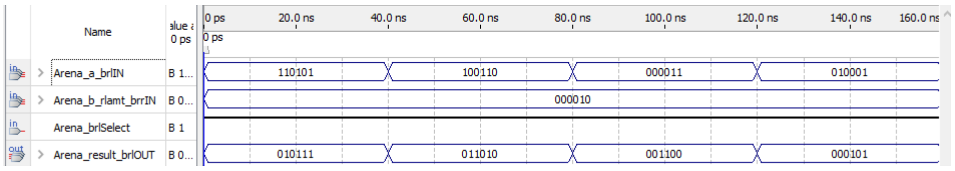


Figure 36: BRL Waveform

Looking at the figures above, we can see our design for the brl is correct. We see comparing the waveform to the truth table. Looking at the first example, A = 110101 B is 000010. B is the shift amount. In decimal, that value is 2. So, this should rotate A to the left twice. As we can see, that is the case in Figure 36. The rest of the examples can be seen in the waveform.

<u>Bitwise Bit Rotate Right</u>

The eigth simulation will be done for the bitwise bit rotate right.

Figure **37** below is shows results of the bitwise brr waveform. Our results should correspond with Figure 23.

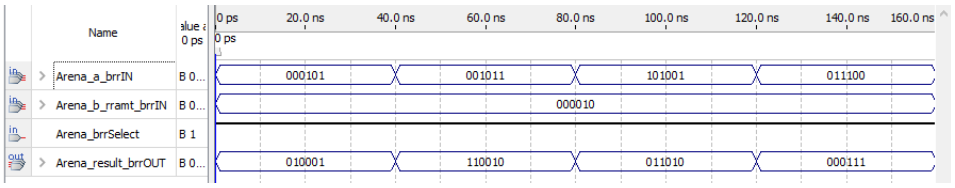


Figure 37: BRR Waveform

Looking at the figures above, we can see our design for the brr is correct. We see comparing the waveform to the truth table. Looking at the first example, A = 000101 B is 000010. B is the shift amount. In decimal, that value is 2. So, this should rotate A to the right twice. As we can see, that is the case in Figure 37. The rest of the examples can be seen in the waveform.

<u>Bitwise Set Less Than Unsigned</u>

The ninth simulation will be done for the bitwise set less than unsigned.

Figure **38a and 38b** below is shows results of the bitwise sltu waveform. Our results should correspond with Figure 26.

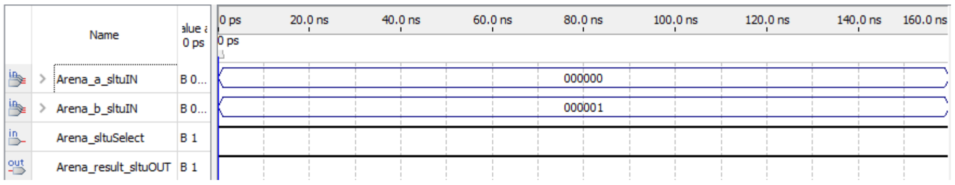


Figure 38a: SLTU Waveform 1

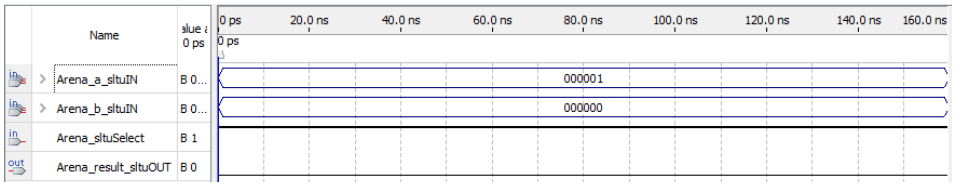


Figure 38b: SLTU Waveform 2

Looking at the figures above, we can see our design for the sltu is correct. We see comparing the waveform to the truth table. Looking at the first example in 38a, A = 000000 B is 000001. This is an unsigned comparison operation, so the values being compared are 0 and 1, so the output should be high for true. As we can see, that is the case in Figure 38a. Looking at the first example in 38b, A = 000001 B is 000000. This is an unsigned comparison operation, so the values being compared are 1 and 0, so the output should be low for false. As we can see, that is the case in Figure 38b. The rest of the examples can be seen in the waveforms.

<u>**Bitwise Set Less Than Signed**</u>

The tenth simulation will be done for the bitwise set less than signed.

Figure **39a and 39b** below is shows results of the bitwise slt waveform. Our results should correspond with Figure 29.

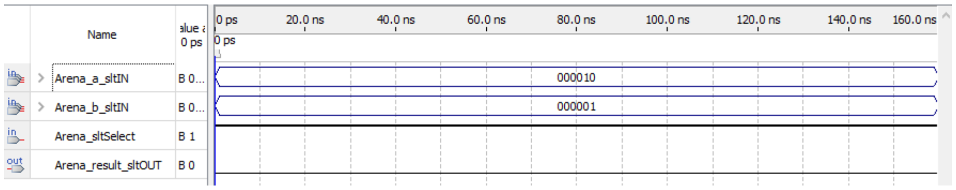


Figure 39a: SLT Waveform 1

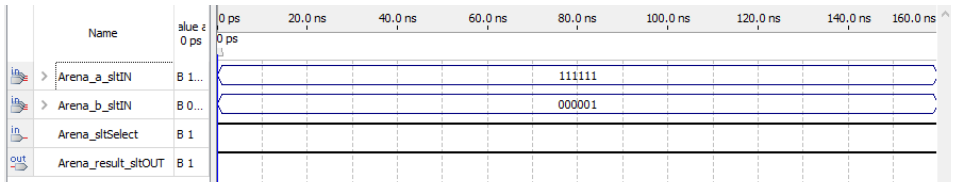


Figure 39b: SLT Waveform 2

Looking at the figures above, we can see our design for the slt is correct. We see comparing the waveform to the truth table. Looking at the first example in 39a, A = 000010 B is 000001. This is an signed comparison operation, so the values being compared are 2 and 1, so the output should be low for false. As we can see, that is the case in Figure 39b. Looking at the first example in 39b, A = 111111 B is 000001. This is an signed comparison operation, so the values being compared are -1 and 1, so the output should be high for true. As we can see, that is the case in Figure 39b. The rest of the examples can be seen in the waveforms.

Bitwise Operation Redesign

The eleventh simulation will be done for the bitwise redesign (aka the final design) Figure **40a and 40b** below is shows results of the bitwise operation circuit. It should correspond with how it was described in the specifications.

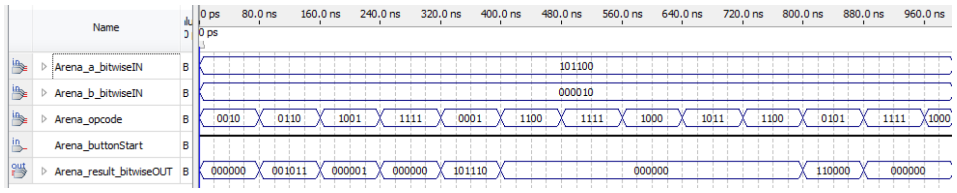


Figure 40a: Redesign/Final Design Waveform 1

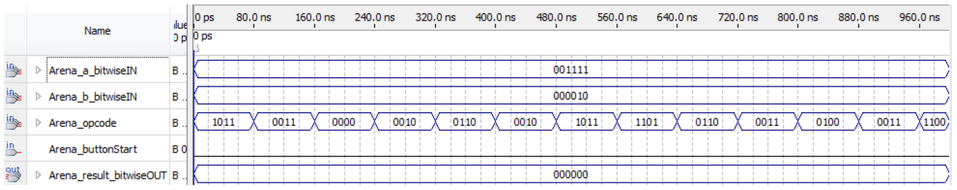


Figure 40b: Redesign/Final Design Waveform 2

Looking at the figures above, we can see our design is correct. I kept the same inputs for all different opcodes just to make It clearer, and a different opcode for each case. The outputs follow accordingly depending on the input. For example, for opcode 0001, we have the **or** operation, and the output shown is 101110, which is correct when 101100 OR 000010. All the other outputs can be seen in the Figure. For Figure 40b, this confirms the design of buttonStart, showing there nothing is outputted when the button isn't used.

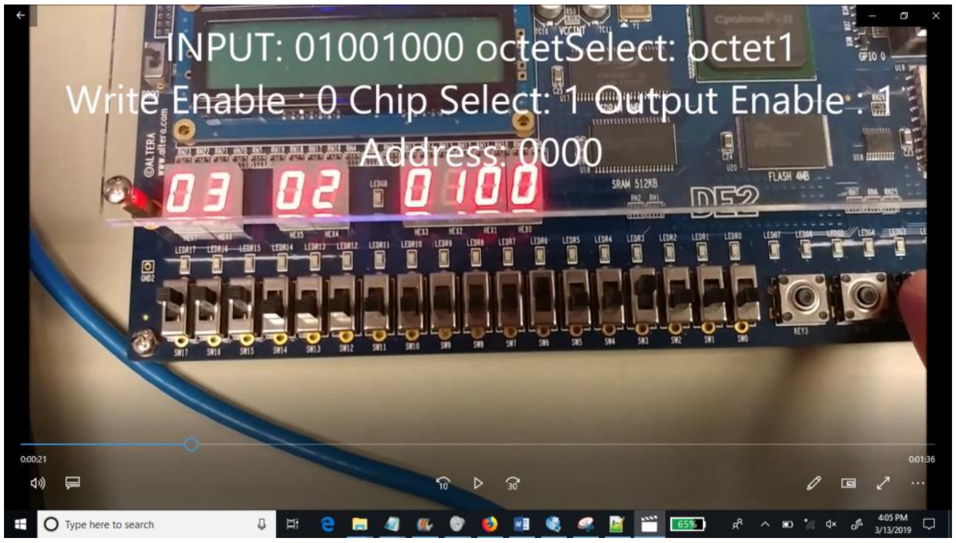
16x4 SIGNED SRAM

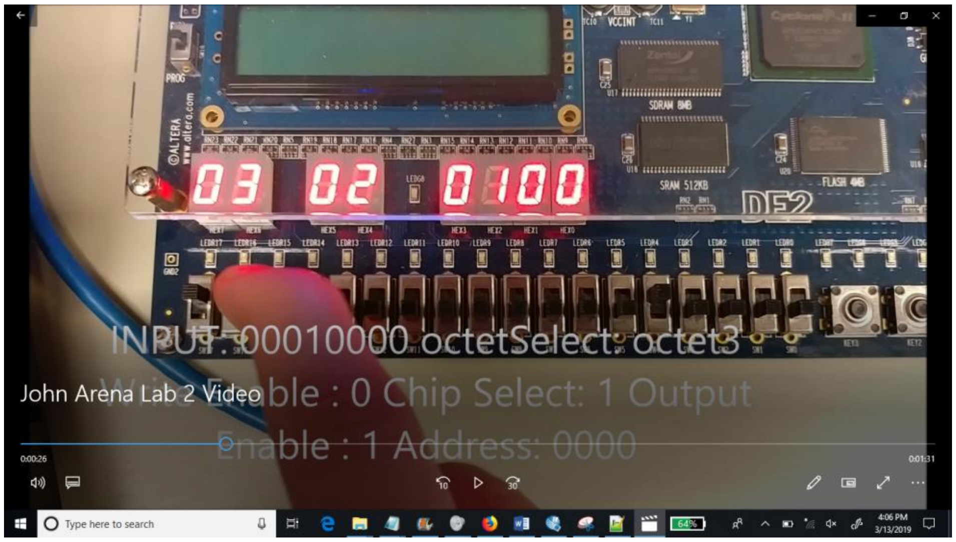
I did not include a simulation for this, as again this is the same as the 16x4 SRAM, with just some added functions to the decoder.

Section 4) Demonstration Pictures

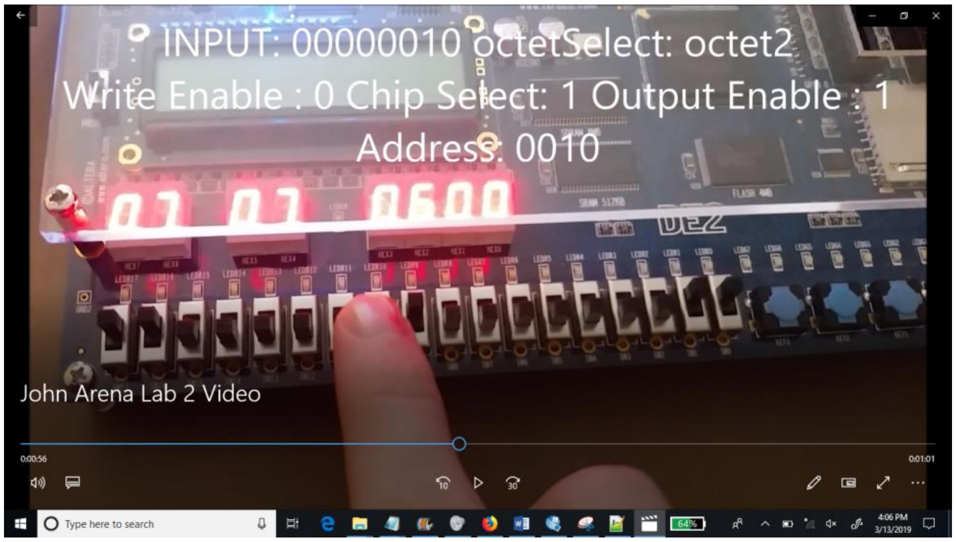
(Not every case is shown in the pictures, all cases shown in video)

16x32 SRAM

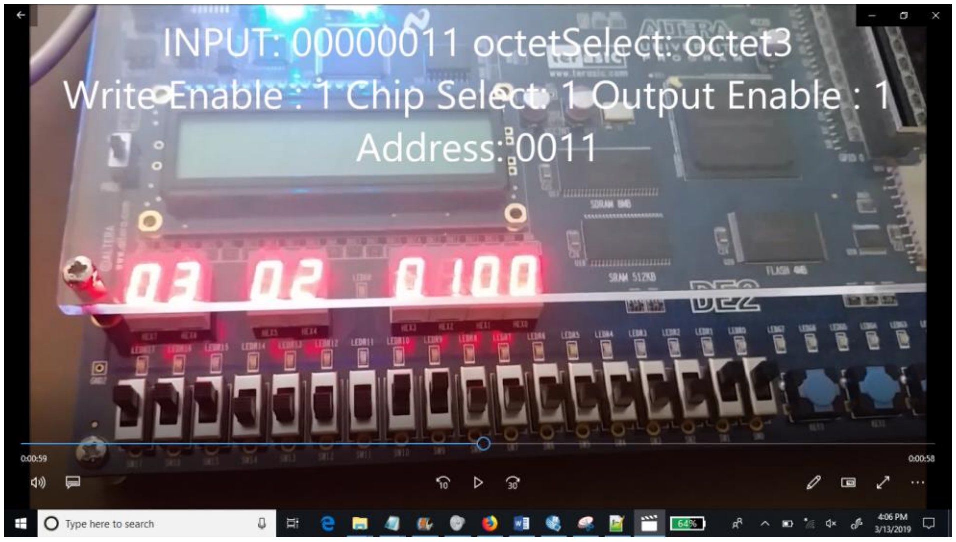

INPUT: 01001000 octetSelect: octet1
Write Enable : 0 Chip Select: 1 Output Enable : 1
Address: 0000
03 02 0100
0:00:21
0:01:36
Type here to search
4:05 PM
3/13/2019



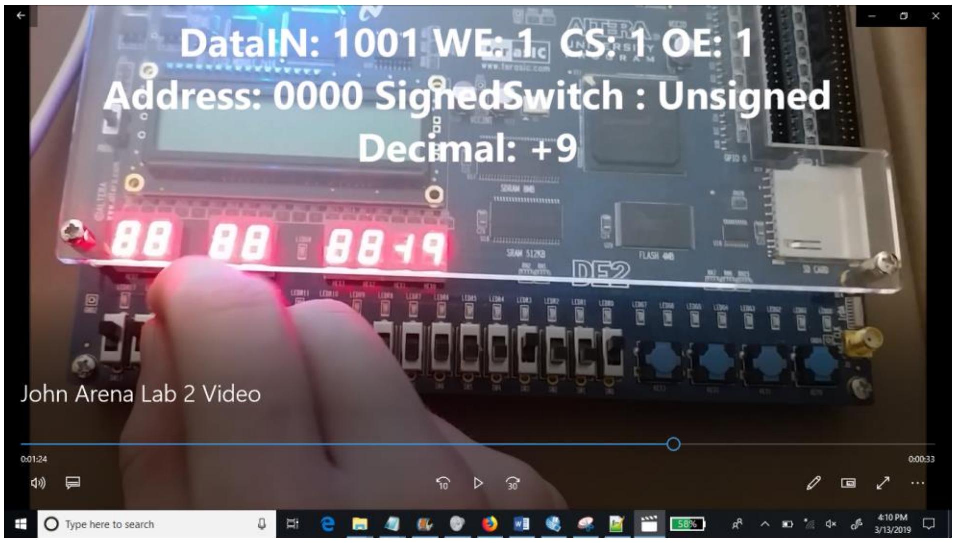
INPUT: 00010000 octetSelect: octet3
Write Enable : 0 Chip Select: 1 Output Enable : 1 Address: 0000

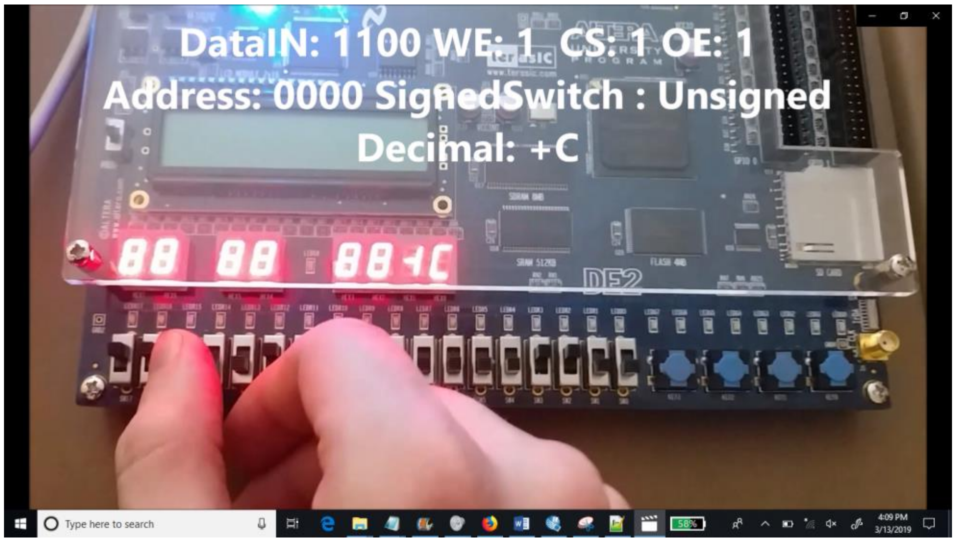


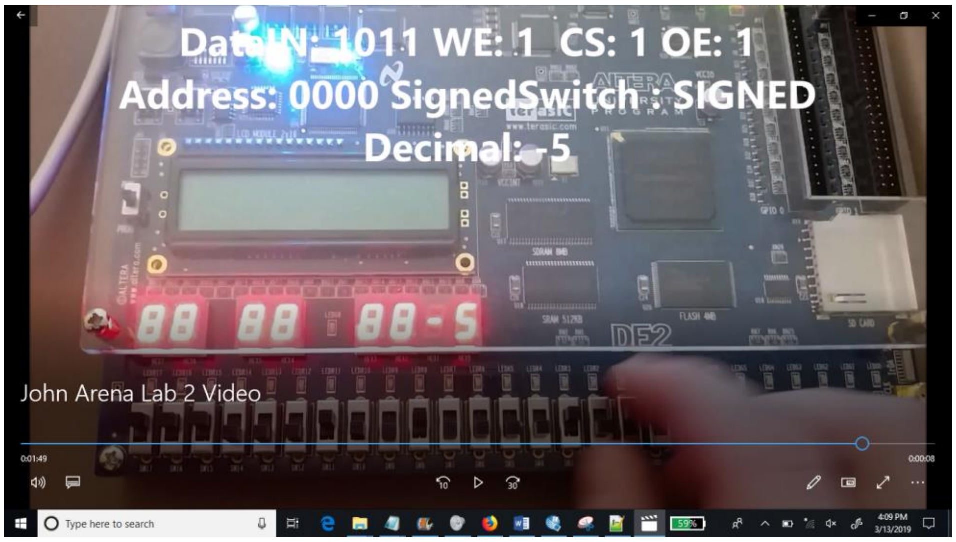
INPUT: 00000010 octetSelect: octet2
Write Enable : 0 Chip Select: 1 Output Enable : 1
Address: 0010

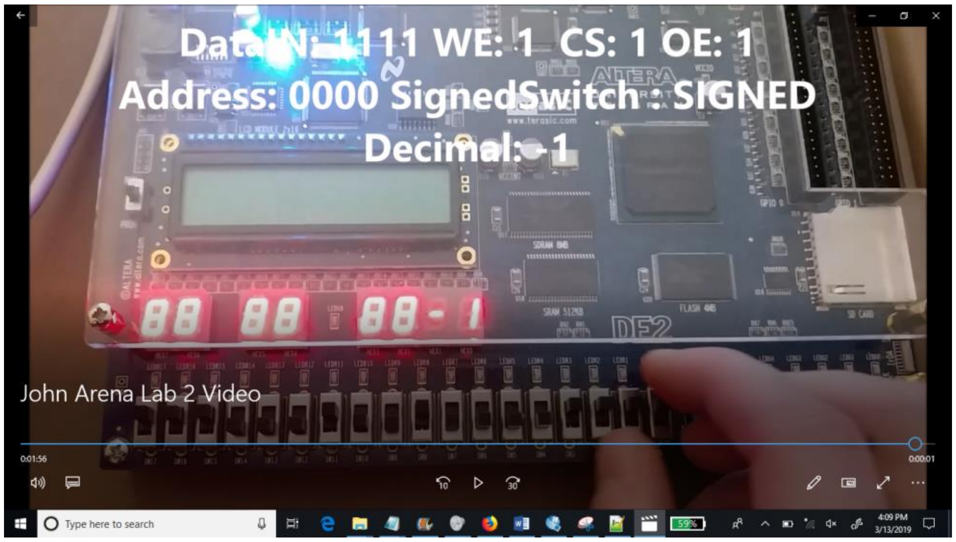

INPUT: 00000011 octetSelect: octet3
Write Enable : 1 Chip Select: 1 Output Enable : 1
Address: 0011
03 02 0 100

<u>16x4 SIGNED SRAM</u>


DataIN: 1001 WE: 1 CS: 1 OE: 1
Address: 0000 SignedSwitch : Unsigned
Decimal: +9
John Arena Lab 2 Video


DataIN: 1100 WE: 1 CS: 1 OE: 1
Address: 0000 SignedSwitch : Unsigned
Decimal: +C


DataIN: 1011 WE: 1 CS: 1 OE: 1
Address: 0000 SignedSwitch : SIGNED
Decimal: -5
John Arena Lab 2 Video


Data IN: 1111 WE: 1 CS: 1 OE: 1
Address: 0000 SignedSwitch : SIGNED
Decimal: -1
John Arena Lab 2 Video

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.