John Arena

Professor Gertner

CSC 342/343

Lab 4

Due 4/17/19

Spring 2019

NOT COMPLETED YET

Table of Contents

Section 1) Objective

For this "real lab", I will be extending my knowledge of VHDL, ModelSim and Quartus to examine, design, test and implement arithmetic circuits that add, subtract, multiply and divide **32-bit integers.** I will use 32-bit registers to store operands I input with switches on our FPGA board. The results will be displayed on 7-segment display and LEDS, using unsigned arithmetic only. The objectives are as follows.

- Design and test an 8-bit accumulator (along with test-benches in VHDL and test in ModelSIM) with an opcode to switch between subtraction an addition. **Extend to 32-bits.**
- Design and test a 4-bit array multiplier circuit (along with test-benches in VHDL and test in ModelSIM) using 1-bit adders. **Extend to 32-bits.**
- Design and test a 4-bit array multiplier circuit (along with test-benches in VHDL and test in ModelSIM) using **n-bit adders**. **Extend to 16-bits (gives 32-bit output).**
- Design a parallel adder tree circuit
- Design and test a 16-bit array multiplier circuit using the parallel adder tree circuits in place of an n-bit adder.
- Design and test a 32-bit divider (along with test-benches in VHDL and test in ModelSIM). Output on HEX Display and remainder using LED's

Section 2) Description and Specifications

32-bit Accumulator

The first circuit I will be designing is a 32-bit Accumulator. An accumulator is " An accumulator is a register for short-term, intermediate storage of arithmetic and logic data in a

computer's CPU (central processing unit). ... In a modern computers, any register can function as an accumulator. The most elementary use for an accumulator is adding a sequence of numbers.”. [https://whatis.techtarget.com/definition/accumulator]

With that said, again I will be designing a 32-bit accumulator. For this lab, it’s really just a series of 1-bit full adders all the way up to n, for an n-bit accumulator. A full adder has the following truth table

X	**Y**	**Carry In**	**Sum**	**Carry Out**
0	**0**	**0**	**0**	**0**
0	**0**	**1**	**1**	**0**
0	**1**	**0**	**1**	**0**
0	**1**	**1**	**0**	**1**
1	**0**	**0**	**1**	**0**
1	**0**	**1**	**0**	**1**
1	**1**	**0**	**0**	**1**
1	**1**	**1**	**1**	**1**

Table 1: 1-bit Full Adder 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.

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

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

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

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

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

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

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

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

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

Equation 1: Sum and Carry Out Equations of Full Adder

These have been derived to much simpler equations using Boolean algebra simplification. To make a 32-bit ripple adder, I will feed the carry outs into the carry ins into the n+1th FA. The first adder has a carry-in supplied by the user. The last adder outputs the final carry out to the user.

Also needed for this design is subtraction. Below is the truth table and equations.

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

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

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

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

$$=> \boldsymbol{Difference = A\ xor\ B\ xor\ Bin}$$

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

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

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

$$=> \boldsymbol{Borrow\ Out = (A' * Bin) + \big((A! * B)\big) + (Bin * B)}$$

Below in Figure 1 is the VHDL code I created for the accumulator. To extend this design to 32 bit accumulator, I simply created four arrays to hold in Carry Ins and Outs, Borrow Ins and Borrow outs, that way they can be passed into the next ripple adder or subtractor.

```
-- (First, Last) John Arena - CSC 342/343 - Lab 4 - Spring 2019 Due: 4/17/19
-- Arena_32bitAdder.vhd
library ieee;
use ieee.std_logic_1164.all;

entity Arena_32bitAdder is
port(
      Arena_A_32bit, Arena_B_32bit: in std_logic_vector(31 downto 0);
      Arena_sub_add: in std_logic; -- 1 = subtraciton, 0 = addition
      Arena_Cin_32bit : in std_logic := '0'; -- Carry for addition, also used
for borrowing in subtraction
      Arena_Bin_32bit : in std_logic := '0'; -- Borrow in
      Arena_AccumOut_32bit : out std_logic_vector(31 downto 0):=
"00000000000000000000000000000000";
      Arena_Difference_32bit : out std_logic_vector(31 downto 0):=
"00000000000000000000000000000000";
      Arena_Cout_32bit : out std_logic := '0'; -- Carry out
      Arena_Bout_32bit : out std_logic := '0'; -- Borrow out
      Arena_reset : in std_logic -- Active low reset
      );
end Arena_32bitAdder;

architecture Arena_32bitAdder_arch of Arena_32bitAdder is

      begin
            process(Arena_A_32bit, Arena_B_32bit, Arena_Cin_32bit,
Arena_sub_add, Arena_reset)
                  variable Arena_Cin_32bit_vars : std_logic_vector(31 downto
0):= (others => '0');
                  variable Arena_Cout_32bit_vars : std_logic_vector(31 downto
0):= (others => '0');
                  variable Arena_Bin_32bit_vars : std_logic_vector(31 downto
0):= (others => '0');
                  variable Arena_Bout_32bit_vars : std_logic_vector(31 downto
0):= (others => '0');

      begin
      if(Arena_sub_add = '0') then
                                    -- Adder 0
            Arena_AccumOut_32bit(0) <= Arena_A_32bit(0) xor
Arena_B_32bit(0)xor Arena_Cin_32bit;

            Arena_Cout_32bit_vars(0) := (Arena_A_32bit(0) and
Arena_B_32bit(0)) or (Arena_Cin_32bit and Arena_A_32bit(0)) or
(Arena_Cin_32bit and Arena_B_32bit(0));

            Arena_Cin_32bit_vars(1) := Arena_Cout_32bit_vars(0);
                  for i in 1 to 30 loop
```

```
                        Arena_AccumOut_32bit(i) <= Arena_A_32bit(i) xor
Arena_B_32bit(i)xor Arena_Cin_32bit_vars(i);

                        Arena_Cout_32bit_vars(i) := (Arena_A_32bit(i) and
Arena_B_32bit(i)) or (Arena_Cin_32bit_vars(i) and Arena_A_32bit(i)) or
(Arena_Cin_32bit_vars(i) and Arena_B_32bit(i));

                        Arena_Cin_32bit_vars(i+1) :=
Arena_Cout_32bit_vars(i);
                  end loop;

                        -- Adder 31
            Arena_AccumOut_32bit(31) <= Arena_A_32bit(31) xor
Arena_B_32bit(31)xor Arena_Cin_32bit_vars(31);

            Arena_Cout_32bit <= (Arena_A_32bit(31) and Arena_B_32bit(31))
                        or (Arena_Cin_32bit_vars(31) and Arena_A_32bit(31))
or (Arena_Cin_32bit_vars(31) and Arena_B_32bit(31));
      elsif (Arena_sub_add = '1') then
                                          -- Subtractor 0
            Arena_Difference_32bit(0) <= Arena_A_32bit(0) xor
Arena_B_32bit(0)  xor Arena_Bin_32bit;

            Arena_Bout_32bit_vars(0) := ((not Arena_A_32bit(0)) and
Arena_Bin_32bit)or ((not Arena_A_32bit(0)) and Arena_B_32bit(0)) or
(Arena_Bin_32bit and Arena_B_32bit(0));

            Arena_Bin_32bit_vars(1) := Arena_Bout_32bit_vars(0);
                  for i in 1 to 30 loop
                        Arena_Difference_32bit(i) <= Arena_A_32bit(i) xor
Arena_B_32bit(i)xor Arena_Bin_32bit_vars(i);

                        Arena_Bout_32bit_vars(i) := ((not Arena_A_32bit(i))
and Arena_Bin_32bit_vars(i))or ((not Arena_A_32bit(i)) and Arena_B_32bit(i))
or (Arena_Bin_32bit_vars(i) and Arena_B_32bit(i));

                        Arena_Bin_32bit_vars(i+1) :=
Arena_Bout_32bit_vars(i);
                  end loop;

                  -- Subtractor 31
                  Arena_Difference_32bit(31) <= Arena_A_32bit(31) xor
Arena_B_32bit(31) xor Arena_Bin_32bit_vars(31);

                  Arena_Bout_32bit <= ((not Arena_A_32bit(31)) and
Arena_Bin_32bit_vars(31))or ((not Arena_A_32bit(31)) and Arena_B_32bit(31))
or (Arena_Bin_32bit_vars(31) and Arena_B_32bit(31));
                  else
                        null;
                  end if;
            end process;
                        end Arena_32bitAdder_arch;
```

Figure 1: Bitwise Accumulator Code

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

REPORT NOT COMPLETE

FROM HERE

DOWNARDS. VIDEO IS

DONE AND FILES ARE

DONE.

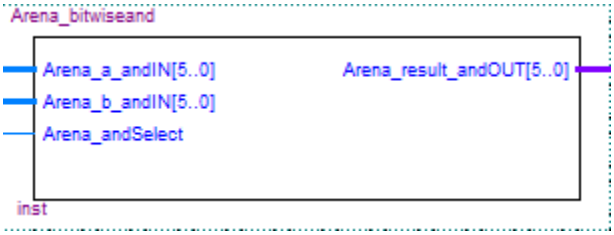


Figure 2: Bitwise AND Symbol

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

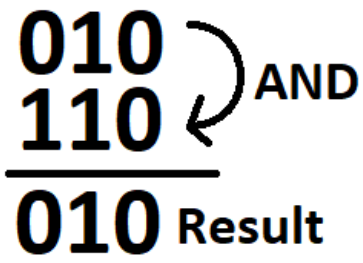


Figure 2: Bitwise AND illustration

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.

```
--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.

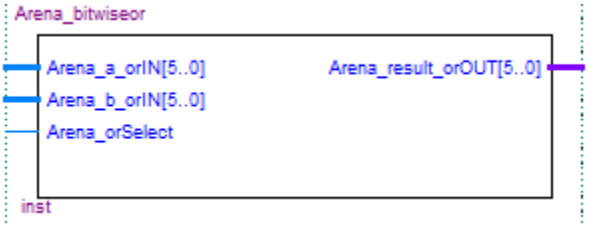


Figure 4: Bitwise OR Symbol

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

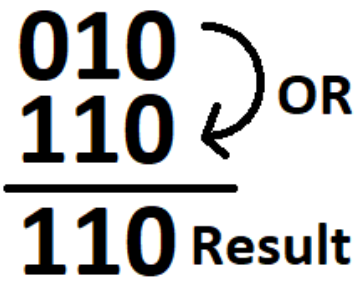


Figure 5: Bitwise OR illustration

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 0^{th} bit of each one will be XOR'd together and become the 0^{th} bit of the output. So X ⊕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 XOR VHDL Code

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

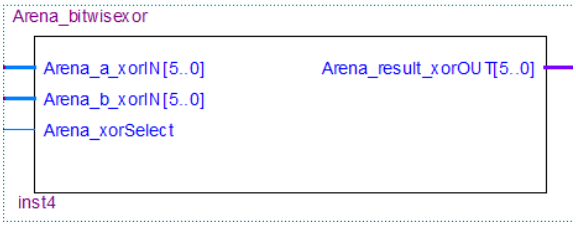


Figure 7: Bitwise XOR Symbol

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

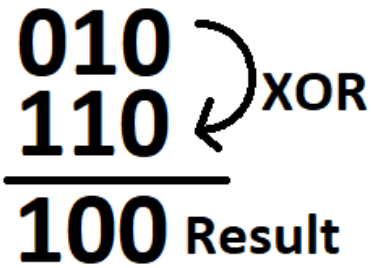


Figure 8: Bitwise XOR illustration

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 0^{th} bit of each one will be NOT'd and become the 0^{th} 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 NOT VHDL Code

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

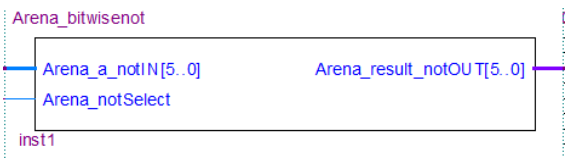


Figure 10: Bitwise NOT Symbol

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

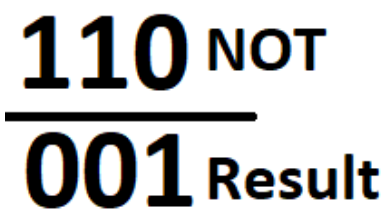


Figure 11: Bitwise NOT illustration

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.

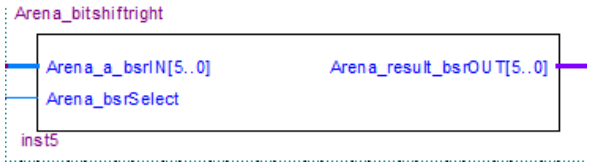


Figure 13: Bitwise BSR Symbol

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 BSL VHDL Code

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

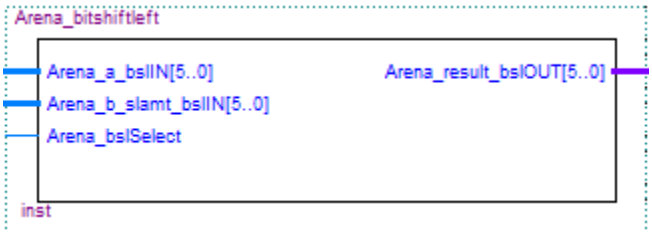


Figure 16: Bitwise BSL Symbol

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

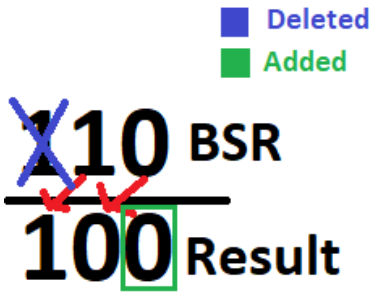


Figure 17: Bitwise BSL illustration

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 BRL.

```
-- (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.

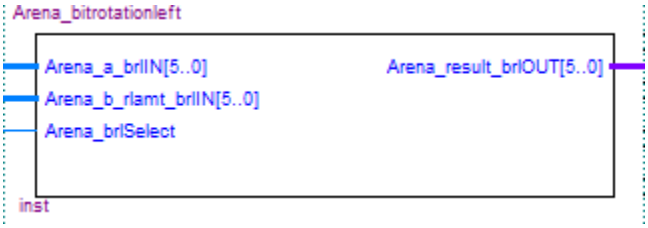


Figure 19: Bitwise BRL Symbol

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

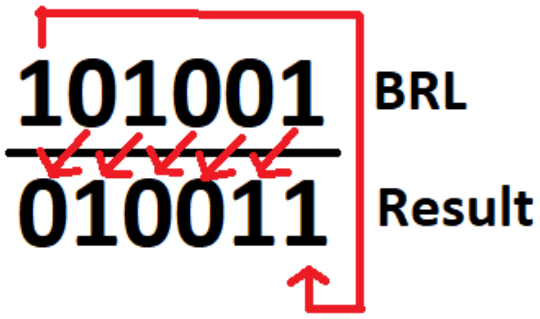


Figure 20: Bitwise BRL illustration

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.

Bitwise Bit Rotate Right

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 BRR VHDL Code

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

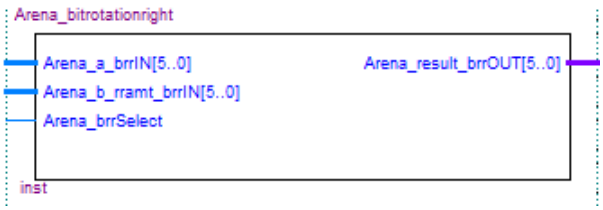


Figure 22: Bitwise BRR Symbol

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

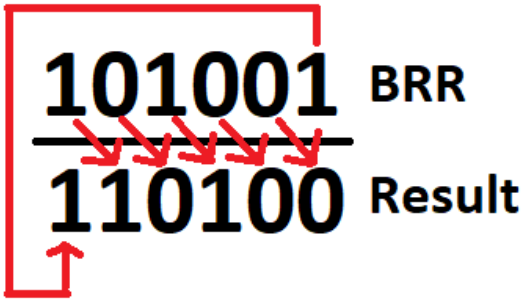


Figure 23: Bitwise BRL illustration

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.

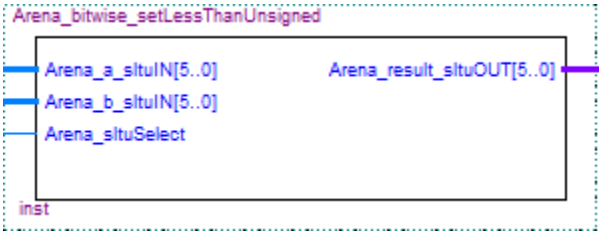


Figure 25: Bitwise SLTU Symbol

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

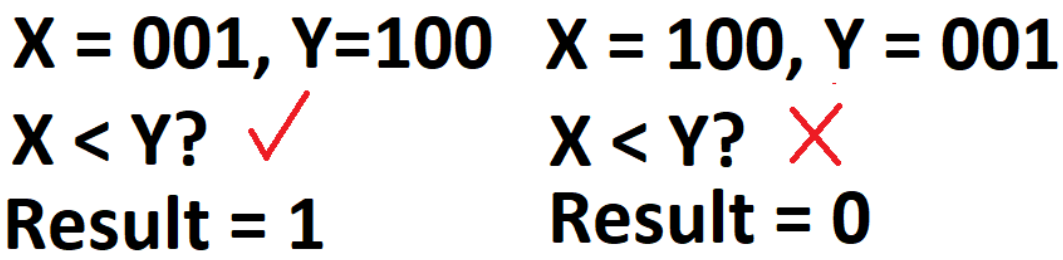


Figure 26: Bitwise SLTU illustration

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
  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 SLT VHDL Code

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

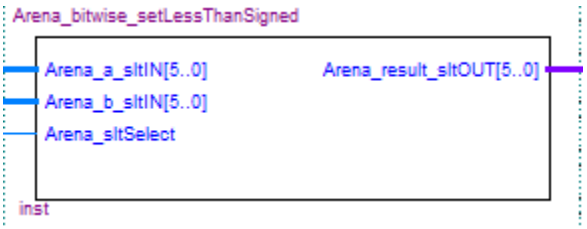


Figure 28: Bitwise SLT Symbol

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

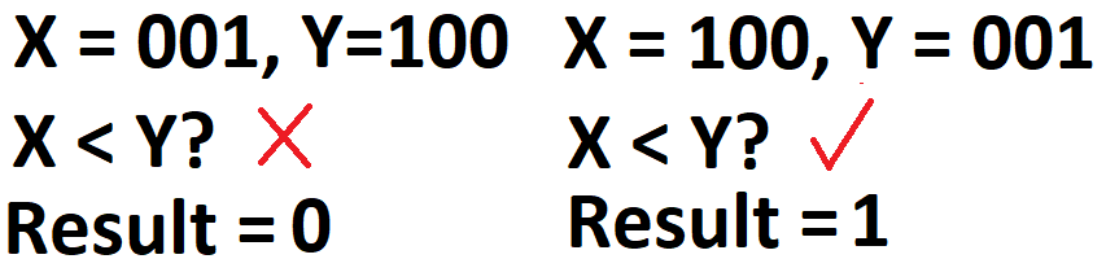


Figure 29: Bitwise SLTU illustration

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.

REDESIGN

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.

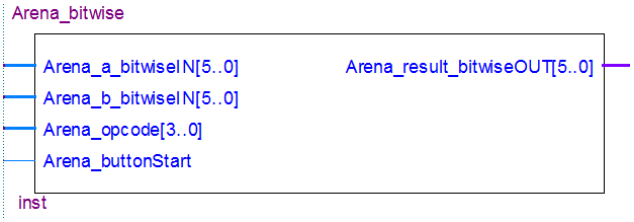


Figure 31: Bitwise Operations Schematic Symbol

Below in figure 32 is the final circuit schematic.

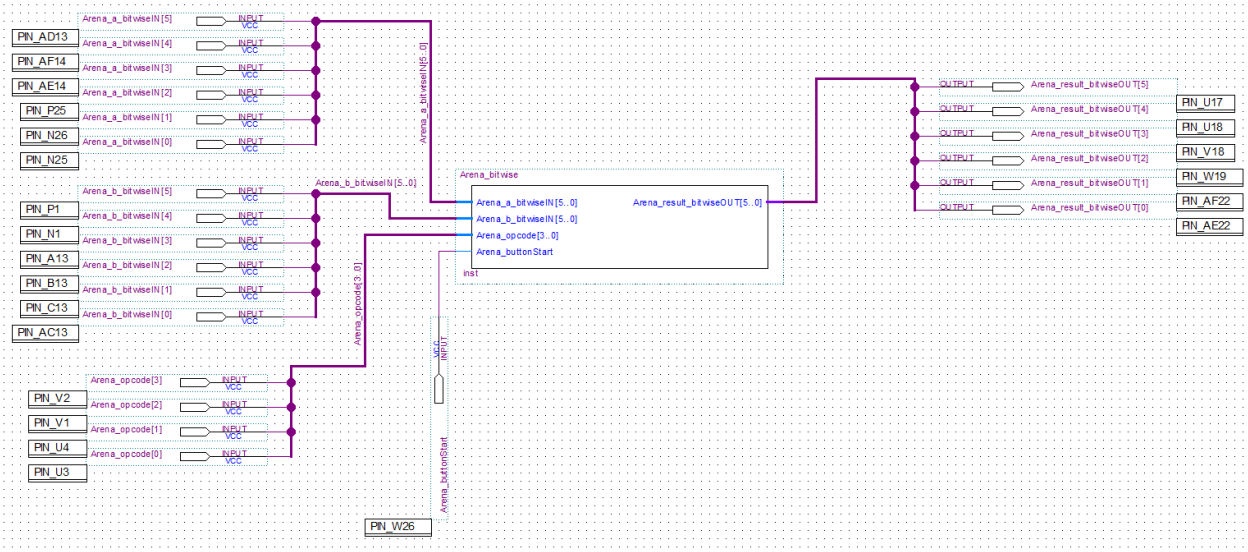


Figure 32: Bitwise Operations Final Schematic

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.

Section 3) Simulations

Bitwise AND

The first simulation will be done for the bitwise AND.

Figure **34** below is shows results of the bitwise AND waveform. Our results should correspond with the truth table in Table **1**.

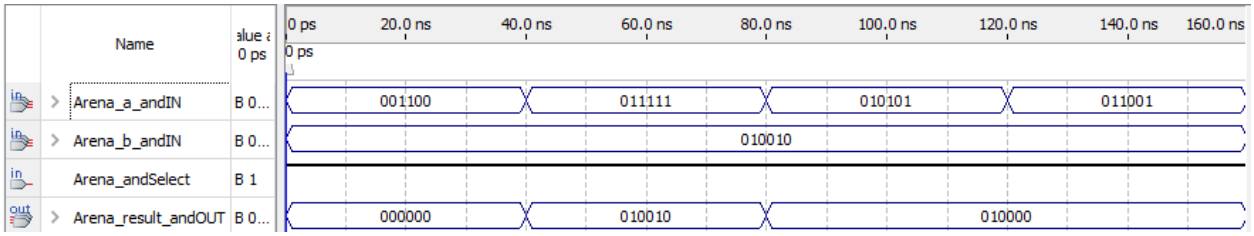


Figure 34: AND Waveform

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.

Bitwise OR

The second simulation will be done for the bitwise OR.

Figure **35** below is shows results of the bitwise OR waveform. Our results should correspond with the truth table in Table **2**.

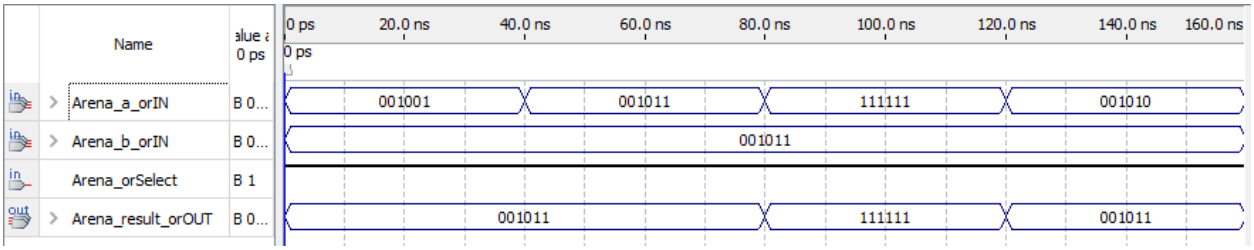


Figure 35: OR Waveform

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.

Bitwise XOR

The third simulation will be done for the bitwise XOR.

Figure **36** on the next page is shows results of the bitwise XOR waveform. Our results should correspond with the truth table in Table **3**.

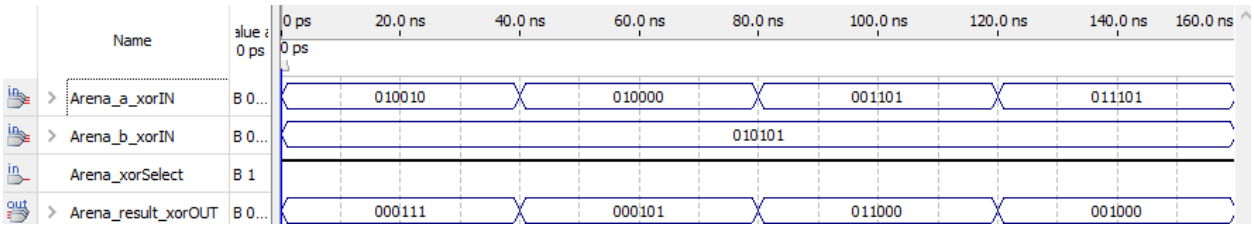


Figure 36: XOR Waveform

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.

Bitwise NOT

The fourth simulation will be done for the bitwise NOT.

Figure **37** below shows results of the bitwise NOT waveform. Our results should correspond with the truth table in Table **4**.

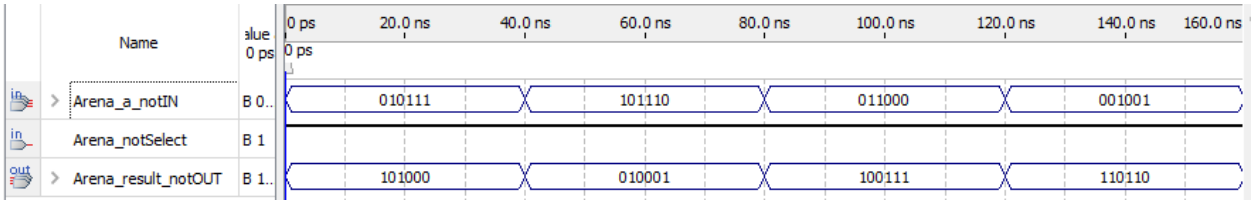


Figure 37: NOT Waveform

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.

Bitwise Bit Shift Right

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

Figure **38** below is shows results of the bitwise bsr waveform. Our results should correspond with Figure 14.

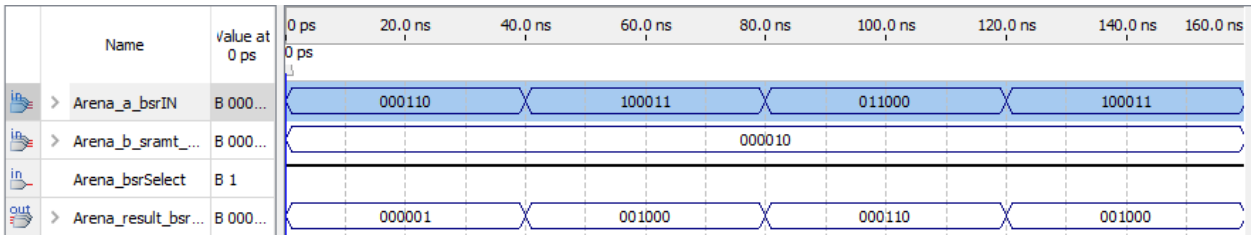


Figure 38: BSR Waveform

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

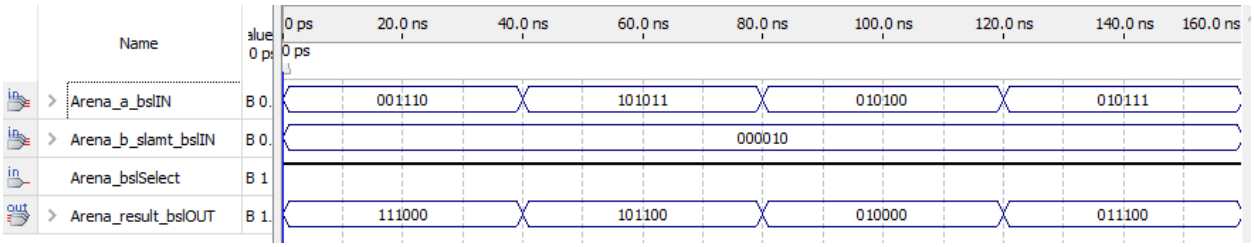


Figure 39: BSL Waveform

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

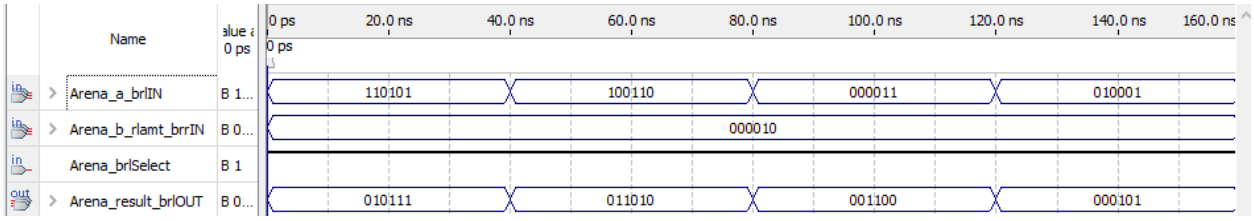


Figure 40: BRL Waveform

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.

Bitwise Bit Rotate Right

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

Figure **41** below is shows results of the bitwise brr waveform. Our results should correspond with Figure 23.

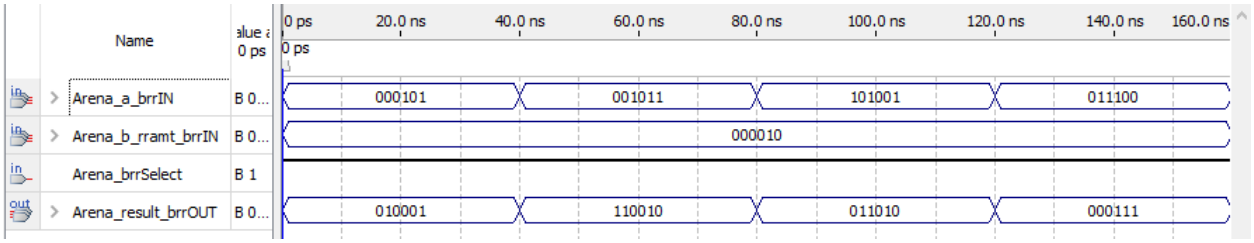


Figure 41: BRR Waveform

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.

Bitwise Set Less Than Unsigned

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

Figure **42a and 42b** below is shows results of the bitwise sltu waveform. Our results should correspond with Figure 26.

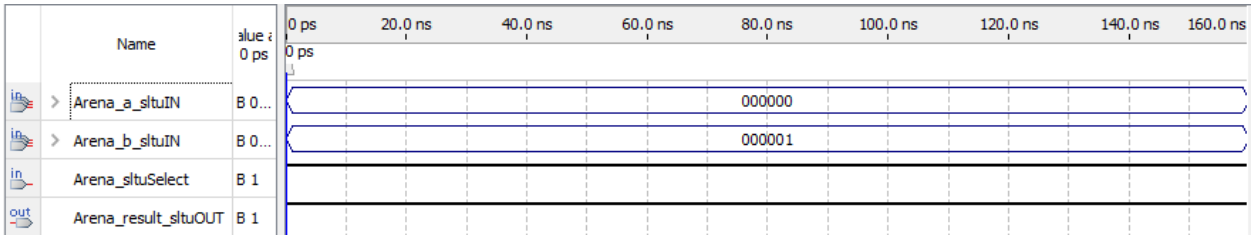


Figure 42a: SLTU Waveform 1

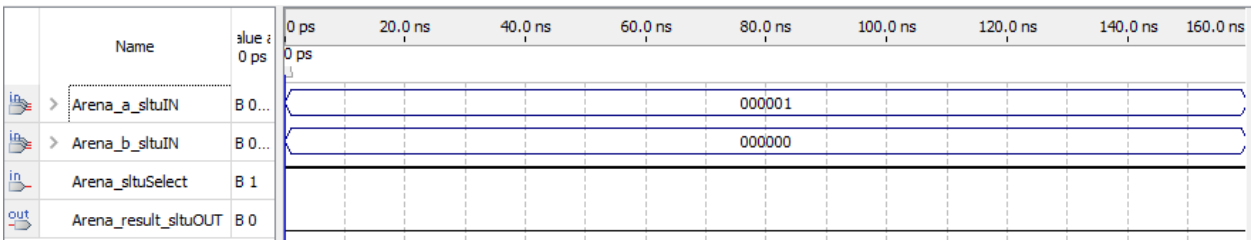


Figure 42b: SLTU Waveform 2

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.

Bitwise Set Less Than Signed

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

Figure **43a and 43b** below is shows results of the bitwise slt waveform. Our results should correspond with Figure 29.

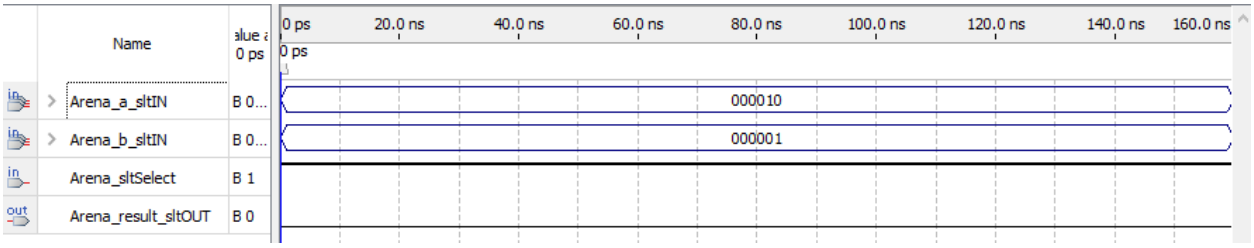


Figure 43a: SLT Waveform 1

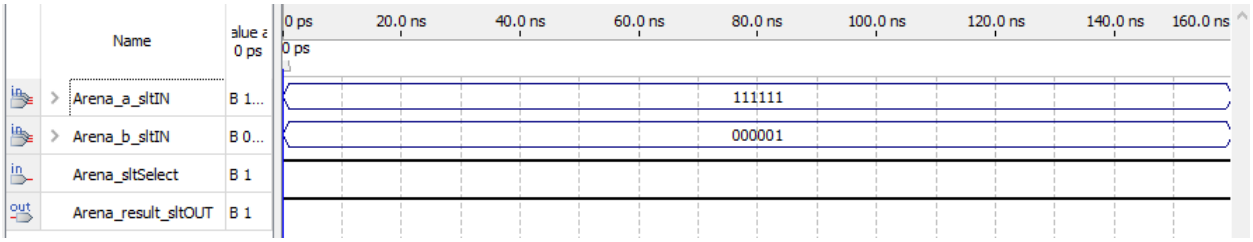


Figure 43b: SLT Waveform 2

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 **44a and 44b** below is shows results of the bitwise operation circuit in VMF. It should correspond with how it was described in the specifications.

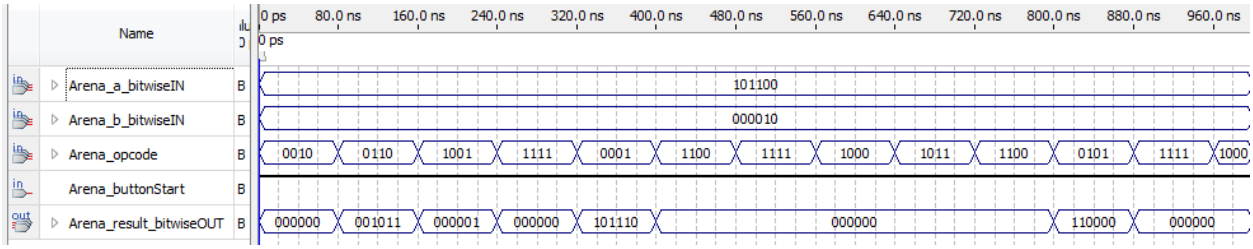


Figure 44a: Redesign/Final Design Waveform 1

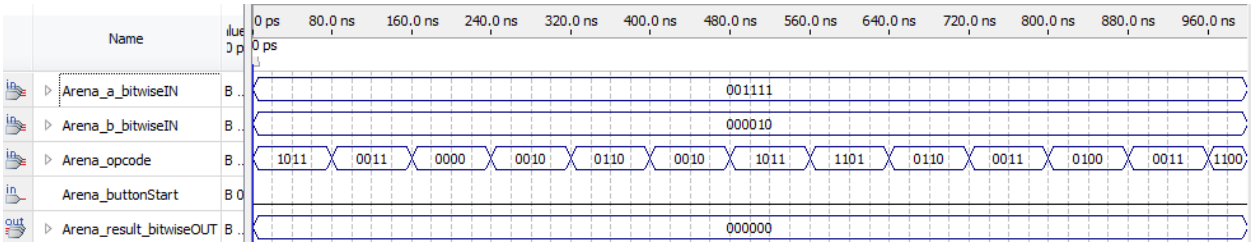


Figure 44b: Redesign/Final Design Waveform 2

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.

Next in Figure **45** on the next page is the VHDL testbench code of the bitwise operation circuit in ModelSim. It should correspond with how it was described in the specifications.

```
-- (First, Last) John Arena -CSC 342/343 - Lab 3 - Spring 2019 Due: 4/1/19
-- Arena_bitwise_tb.vhd

library ieee;
use ieee.std_logic_1164.all;

entity Arena_bitwise_tb is

end Arena_bitwise_tb;
architecture Arena_bitwise_tb_arch of Arena_bitwise_tb is
   signal Arena_a_bitwiseIN_tb,Arena_b_bitwiseIN_tb: std_logic_vector(5
downto 0);
   signal Arena_result_bitwiseOUT_tb: std_logic_vector (5 downto 0);
      signal Arena_opcode_tb: std_logic_vector (3 downto 0);
      signal Arena_buttonStart_tb: std_logic;

begin
      UUT : entity work.Arena_bitwise port map (Arena_a_bitwiseIN =>
Arena_a_bitwiseIN_tb, Arena_b_bitwiseIN => Arena_b_bitwiseIN_tb,
         Arena_result_bitwiseOUT => Arena_result_bitwiseOUT_tb, Arena_opcode
=> Arena_opcode_tb, Arena_buttonStart => Arena_buttonStart_tb);
```

```
	tb1: process
	constant period: time := 40ns;
	begin
		Arena_a_bitwiseIN_tb <= "000000";
		Arena_opcode_tb <="0000"; -- Test for NOT operation
		Arena_buttonStart_tb <= '0';
	   wait for period;
		assert(Arena_result_bitwiseOUT_tb = "111111") -- Expected output
		report "Test failed for input 000000 for NOT" severity error;

		Arena_a_bitwiseIN_tb <= "111111";
		Arena_opcode_tb <="0000"; -- Test for NOT operation
		Arena_buttonStart_tb <= '0';
	   wait for period;
		assert(Arena_result_bitwiseOUT_tb = "000000") -- Expected output
		report "Test failed for input 111111 for NOT" severity error;

		Arena_a_bitwiseIN_tb <= "101010";
		Arena_opcode_tb <="0000"; -- Test for NOT operation
		Arena_buttonStart_tb <= '0';
	   wait for period;
		assert(Arena_result_bitwiseOUT_tb = "010101") -- Expected output
		report "Test failed for input 101010 for NOT" severity error;

		Arena_a_bitwiseIN_tb <= "000001";
		Arena_b_bitwiseIN_tb <= "111111";
		Arena_opcode_tb <="0001"; -- Test for OR operation
		Arena_buttonStart_tb <= '0';
	   wait for period;
		assert(Arena_result_bitwiseOUT_tb = "111111") -- Expected output
		report "Test failed for input A:000001 and B:111111 for OR"
severity error;

		Arena_a_bitwiseIN_tb <= "000000";
		Arena_b_bitwiseIN_tb <= "000000";
		Arena_opcode_tb <="0001"; -- Test for OR operation
		Arena_buttonStart_tb <= '0';
	   wait for period;
		assert(Arena_result_bitwiseOUT_tb = "000000") -- Expected output
		report "Test failed for input A:000000 and B:0000000 for OR"
severity error;

		Arena_a_bitwiseIN_tb <= "000000";
		Arena_b_bitwiseIN_tb <= "000000";
		Arena_opcode_tb <="0010"; -- Test for AND operation
		Arena_buttonStart_tb <= '0';
	   wait for period;
		assert(Arena_result_bitwiseOUT_tb = "000000") -- Expected output
		report "Test failed for inputs A:000000 and B:000000 for AND"
severity error;

		Arena_a_bitwiseIN_tb <= "000011";
		Arena_b_bitwiseIN_tb <= "111100";
		Arena_opcode_tb <="0010"; -- Test for AND operation
		Arena_buttonStart_tb <= '0';
	   wait for period;
		assert(Arena_result_bitwiseOUT_tb = "000000") -- Expected output
```

```
            report "Test failed for inputs A:000011 and B:111100 for AND"
severity error;

            Arena_a_bitwiseIN_tb <= "000001";
            Arena_b_bitwiseIN_tb <= "111111";
            Arena_opcode_tb <="0010"; -- Test for AND operation
            Arena_buttonStart_tb <= '0';
         wait for period;
            assert(Arena_result_bitwiseOUT_tb = "000001") -- Expected output
            report "Test failed for inputs A:000001 and B:111111 for AND"
severity error;

            Arena_a_bitwiseIN_tb <= "000000";
            Arena_b_bitwiseIN_tb <= "000000";
            Arena_opcode_tb <="0011"; -- Test for XOR operation
            Arena_buttonStart_tb <= '0';
         wait for period;
            assert(Arena_result_bitwiseOUT_tb = "000000") -- Expected output
            report "Test failed for inputs A:000000 and B:000000 for XOR"
severity error;

            Arena_a_bitwiseIN_tb <= "101010";
            Arena_b_bitwiseIN_tb <= "010101";
            Arena_opcode_tb <="0011"; -- Test for XOR operation
            Arena_buttonStart_tb <= '0';
         wait for period;
            assert(Arena_result_bitwiseOUT_tb = "111111") -- Expected output
            report "Test failed for inputs A:101010 and B:010101 for XOR"
severity error;

            Arena_a_bitwiseIN_tb <= "111111";
            Arena_b_bitwiseIN_tb <= "111111";
            Arena_opcode_tb <="0011"; -- Test for XOR operation
            Arena_buttonStart_tb <= '0';
         wait for period;
            assert(Arena_result_bitwiseOUT_tb = "000000") -- Expected output
            report "Test failed for inputs A:111111 and B:111111 for XOR"
severity error;

            Arena_a_bitwiseIN_tb <= "100000";
            Arena_b_bitwiseIN_tb <= "000001";
            Arena_opcode_tb <="0100"; -- Test for BSR operation
            Arena_buttonStart_tb <= '0';
         wait for period;
            assert(Arena_result_bitwiseOUT_tb = "010000") -- Expected output
            report "Test failed for inputs A:100000 and B:000001 for BSR"
severity error;

            Arena_a_bitwiseIN_tb <= "100011";
            Arena_b_bitwiseIN_tb <= "000011";
            Arena_opcode_tb <="0100"; -- Test for BSR operation
            Arena_buttonStart_tb <= '0';
         wait for period;
            assert(Arena_result_bitwiseOUT_tb = "000100") -- Expected output
            report "Test failed for inputs A:100011 and B:000011 for BSR"
severity error;
```

```
            Arena_a_bitwiseIN_tb <= "000001";
            Arena_b_bitwiseIN_tb <= "000001";
            Arena_opcode_tb <="0101"; -- Test for BSL operation
            Arena_buttonStart_tb <= '0';
        wait for period;
            assert(Arena_result_bitwiseOUT_tb = "000010") -- Expected output
            report "Test failed for inputs A:000001 and B:000001 for BSL"
severity error;

            Arena_a_bitwiseIN_tb <= "110001";
            Arena_b_bitwiseIN_tb <= "000011";
            Arena_opcode_tb <="0101"; -- Test for BSL operation
            Arena_buttonStart_tb <= '0';
        wait for period;
            assert(Arena_result_bitwiseOUT_tb = "001000") -- Expected output
            report "Test failed for inputs A:110001 and B:000011 for BSL"
severity error;

            Arena_a_bitwiseIN_tb <= "000001";
            Arena_b_bitwiseIN_tb <= "000001";
            Arena_opcode_tb <="0110"; -- Test for BRR operation
            Arena_buttonStart_tb <= '0';
        wait for period;
            assert(Arena_result_bitwiseOUT_tb = "100000") -- Expected output
            report "Test failed for inputs A:000001 and B:000001 for BRR"
severity error;

            Arena_a_bitwiseIN_tb <= "001000";
            Arena_b_bitwiseIN_tb <= "000011";
            Arena_opcode_tb <="0110"; -- Test for BRR operation
            Arena_buttonStart_tb <= '0';
        wait for period;
            assert(Arena_result_bitwiseOUT_tb = "000001") -- Expected output
            report "Test failed for inputs A:001000 and B:000011 for BRR"
severity error;

            Arena_a_bitwiseIN_tb <= "100000";
            Arena_b_bitwiseIN_tb <= "000001";
            Arena_opcode_tb <="0111"; -- Test for BRL operation
            Arena_buttonStart_tb <= '0';
        wait for period;
            assert(Arena_result_bitwiseOUT_tb = "000001") -- Expected output
            report "Test failed for inputs A:100000 and B:000001 for BRL"
severity error;

            Arena_a_bitwiseIN_tb <= "000100";
            Arena_b_bitwiseIN_tb <= "000011";
            Arena_opcode_tb <="0111"; -- Test for BRL operation
            Arena_buttonStart_tb <= '0';
        wait for period;
            assert(Arena_result_bitwiseOUT_tb = "100000") -- Expected output
            report "Test failed for inputs A:000100 and B:000011 for BRL"
severity error;

            Arena_a_bitwiseIN_tb <= "000000";
            Arena_b_bitwiseIN_tb <= "000001";
            Arena_opcode_tb <="1000"; -- Test for SLTU operation
```

```
            Arena_buttonStart_tb <= '0';
         wait for period;
            assert(Arena_result_bitwiseOUT_tb = "000001") -- Expected output
            report "Test failed for inputs A:000000 and B:000001 for SLTU"
severity error;

            Arena_a_bitwiseIN_tb <= "111111";
            Arena_b_bitwiseIN_tb <= "000001";
            Arena_opcode_tb <="1000"; -- Test for SLTU operation
            Arena_buttonStart_tb <= '0';
         wait for period;
            assert(Arena_result_bitwiseOUT_tb = "000000") -- Expected output
            report "Test failed for inputs A:111111 and B:000001 for SLTU"
severity error;

            Arena_a_bitwiseIN_tb <= "000000";
            Arena_b_bitwiseIN_tb <= "000001";
            Arena_opcode_tb <="1001"; -- Test for SLT operation
            Arena_buttonStart_tb <= '0';
         wait for period;
            assert(Arena_result_bitwiseOUT_tb = "000001") -- Expected output
            report "Test failed for inputs A:000000 and B:000001 for SLT"
severity error;

            Arena_a_bitwiseIN_tb <= "111111";
            Arena_b_bitwiseIN_tb <= "000001";
            Arena_opcode_tb <="1001"; -- Test for SLT operation
            Arena_buttonStart_tb <= '0';
         wait for period;
            assert(Arena_result_bitwiseOUT_tb = "000001") -- Expected output
            report "Test failed for inputs A:111111 and B:000001 for SLT"
severity error;

            Arena_a_bitwiseIN_tb <= "111110";
            Arena_b_bitwiseIN_tb <= "111101";
            Arena_opcode_tb <="1001"; -- Test for SLT operation
            Arena_buttonStart_tb <= '0';
         wait for period;
            assert(Arena_result_bitwiseOUT_tb = "000000") -- Expected output
            report "Test failed for inputs A:111110 and B:111101 for SLT"
severity error;

            wait;
      end process;

      end Arena_bitwise_tb_arch;
```

Figure 45: VHDL Code BITWISE TestBench

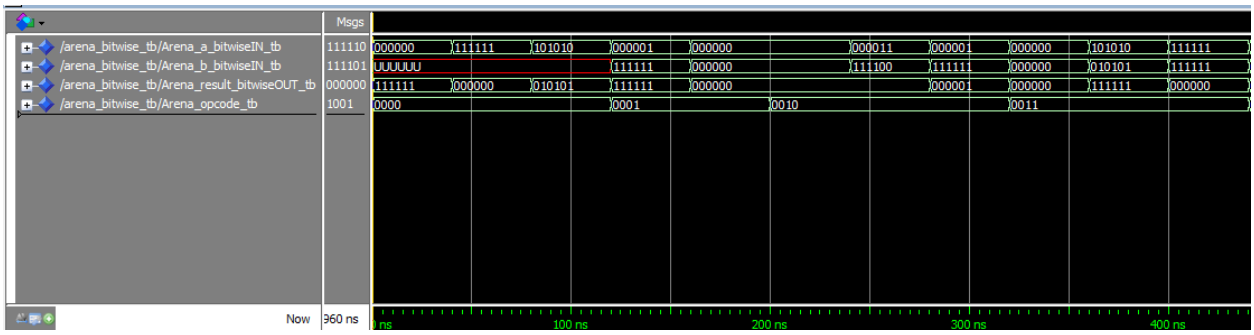


Figure 46a: Redesign/Final Design TB Waveform 1

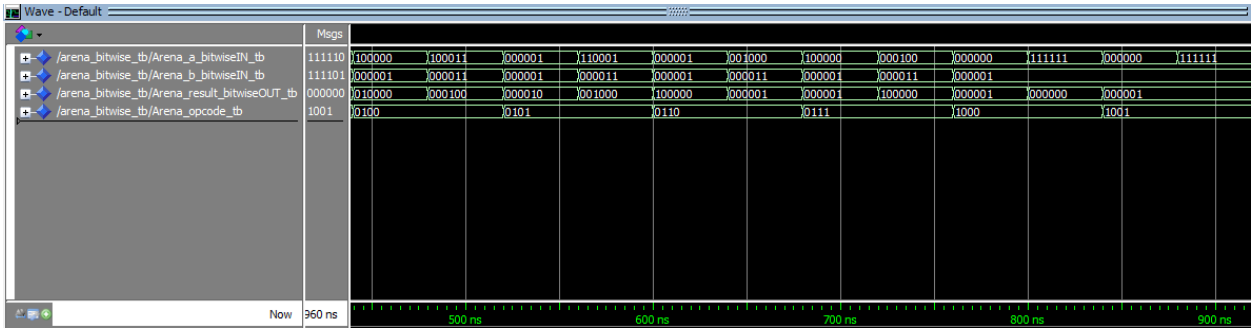


Figure 46b: Redesign/Final Design TB Waveform 2

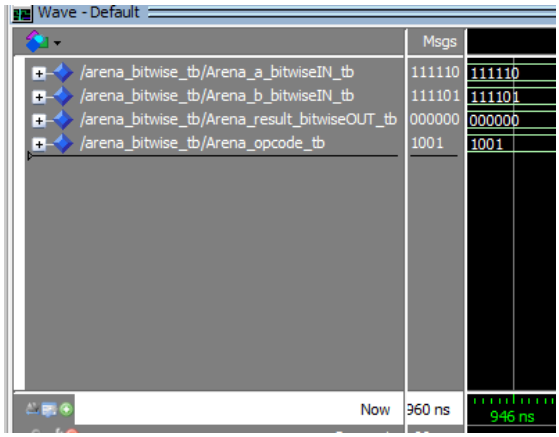


Figure 46c: Redesign/Final Design TB Waveform 2

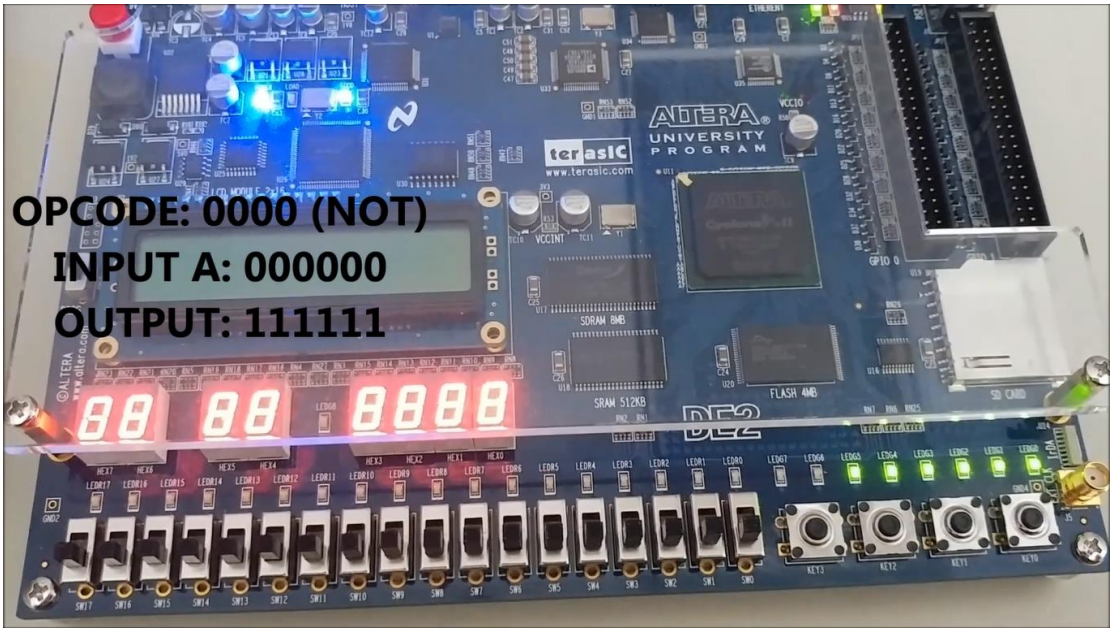
Looking at Figure **46a, 46b and 46c** above shows results of the bitwise testbench in ModelSim. We can see our design is correct. I used inputs for each operation that highlighted the exact functionality of each operation. For example, rotation doesn't lose its values like shift does. Each

opcode as stated already corresponds to a bitwise operation. 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 Figures.

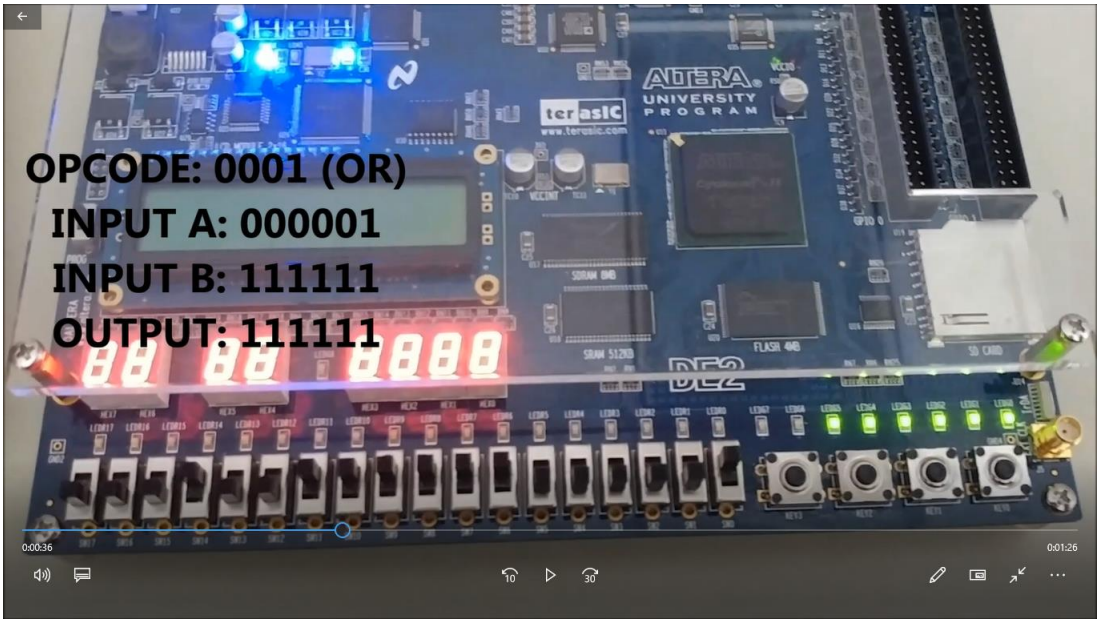
Section 4) Demonstration Pictures

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

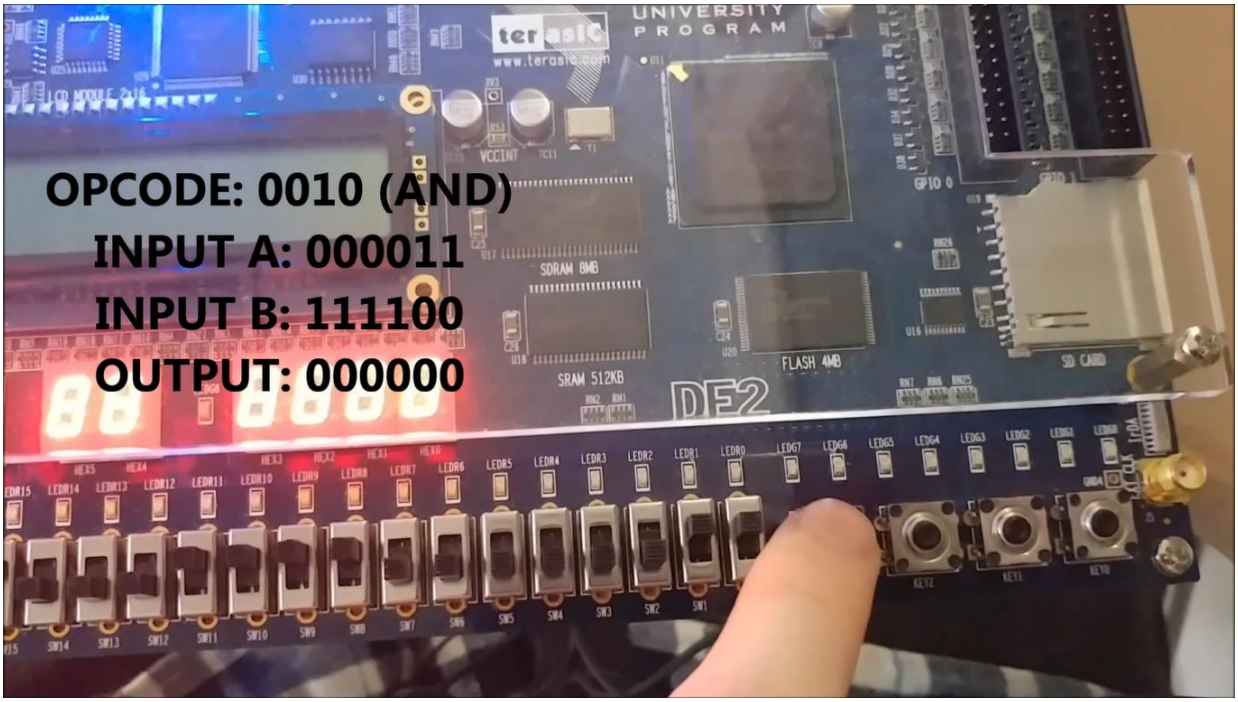
NOT


OPCODE: 0000 (NOT)
INPUT A: 000000
OUTPUT: 111111

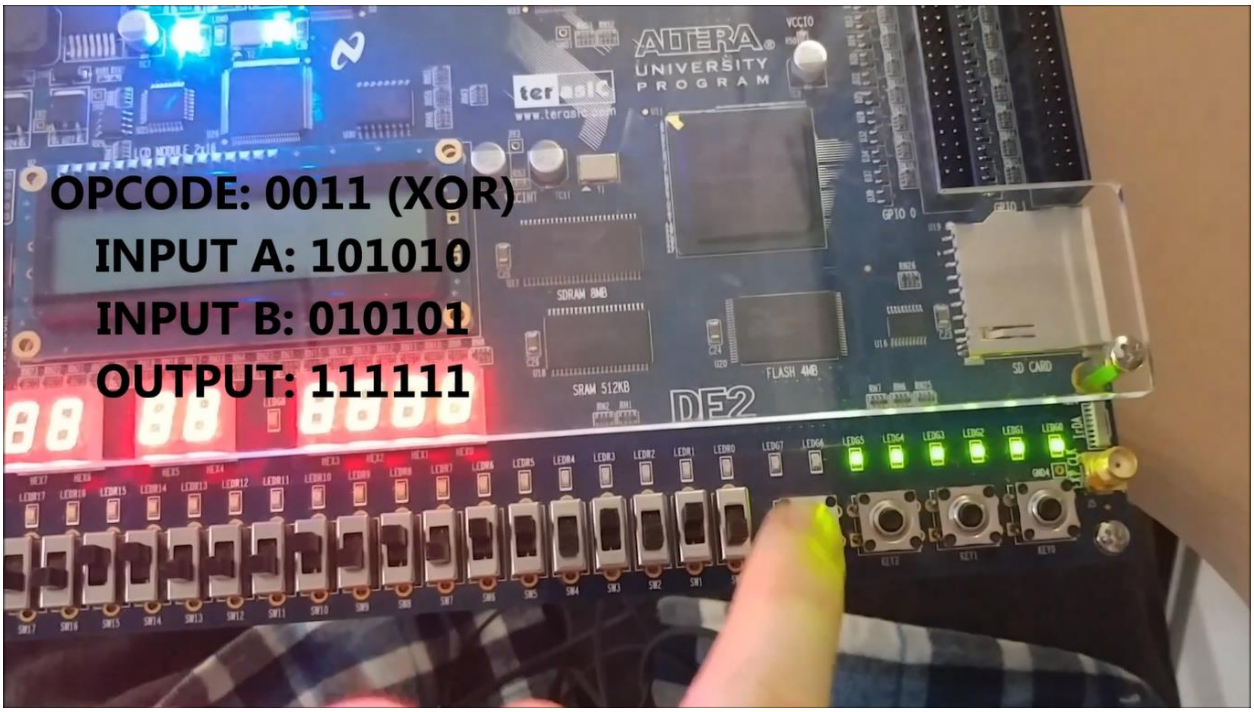
OR


OPCODE: 0001 (OR)
INPUT A: 000001
INPUT B: 111111
OUTPUT: 111111
0:00:36
0:01:26

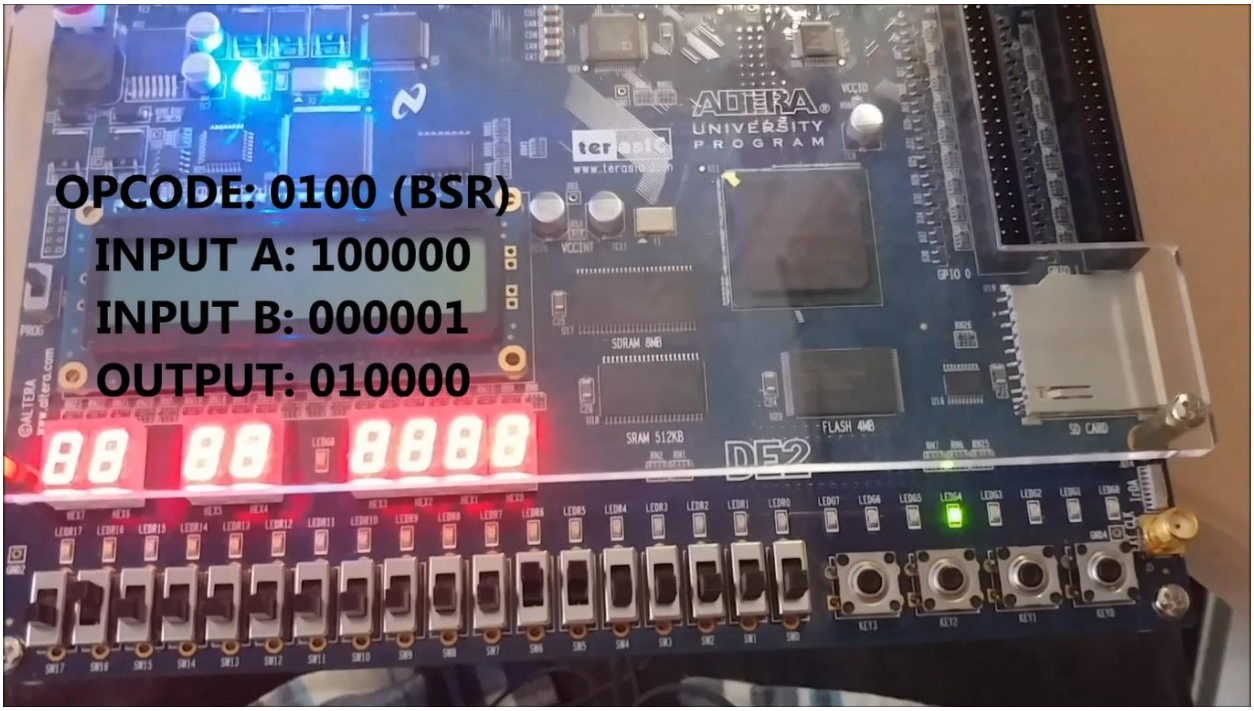
AND



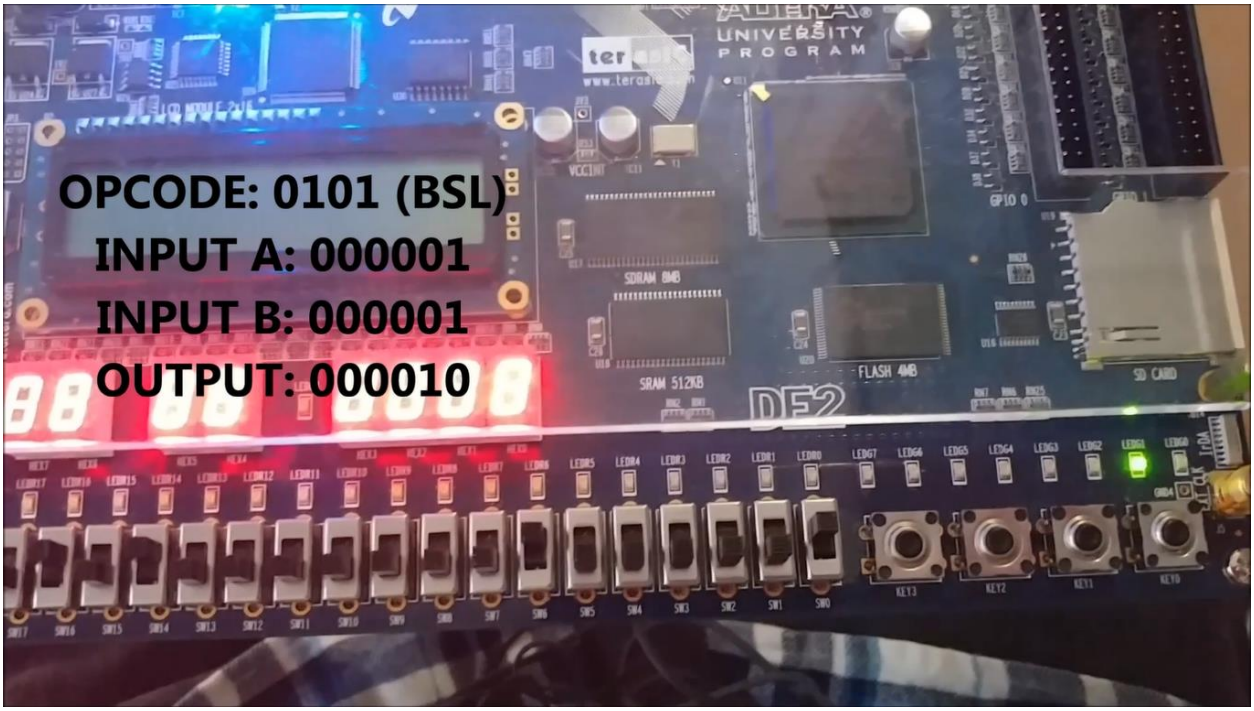
XOR



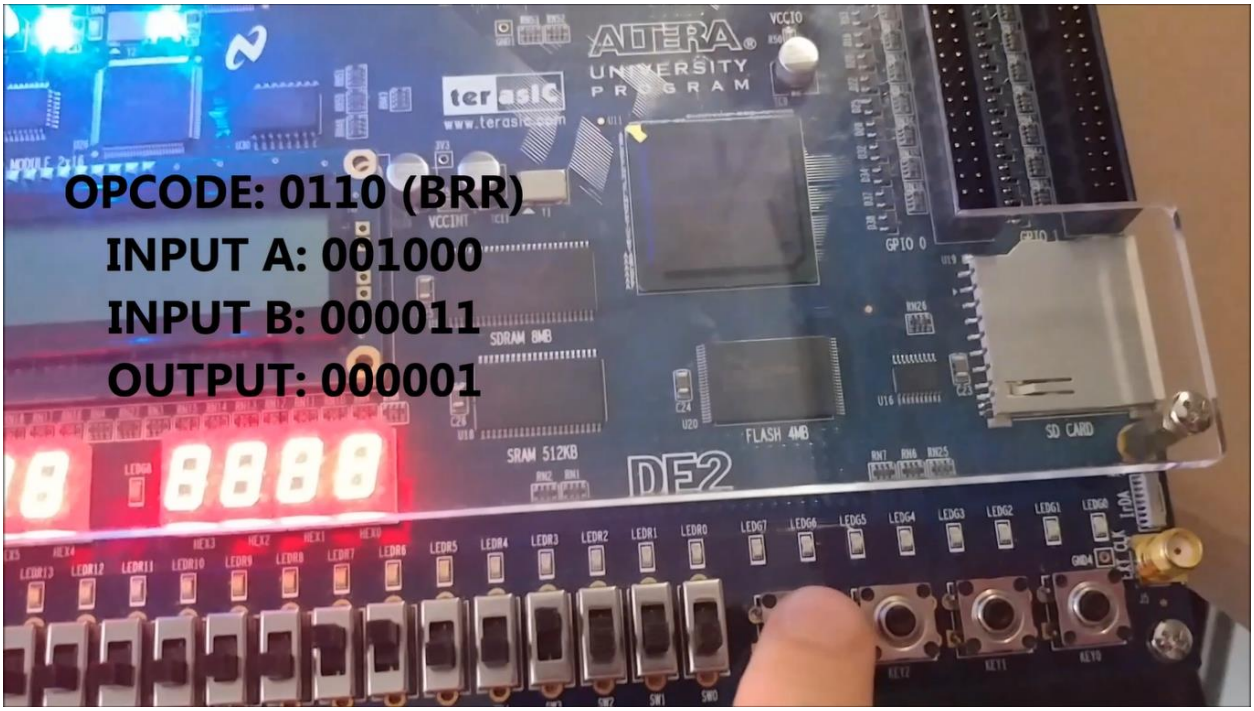
Bit Shift Right



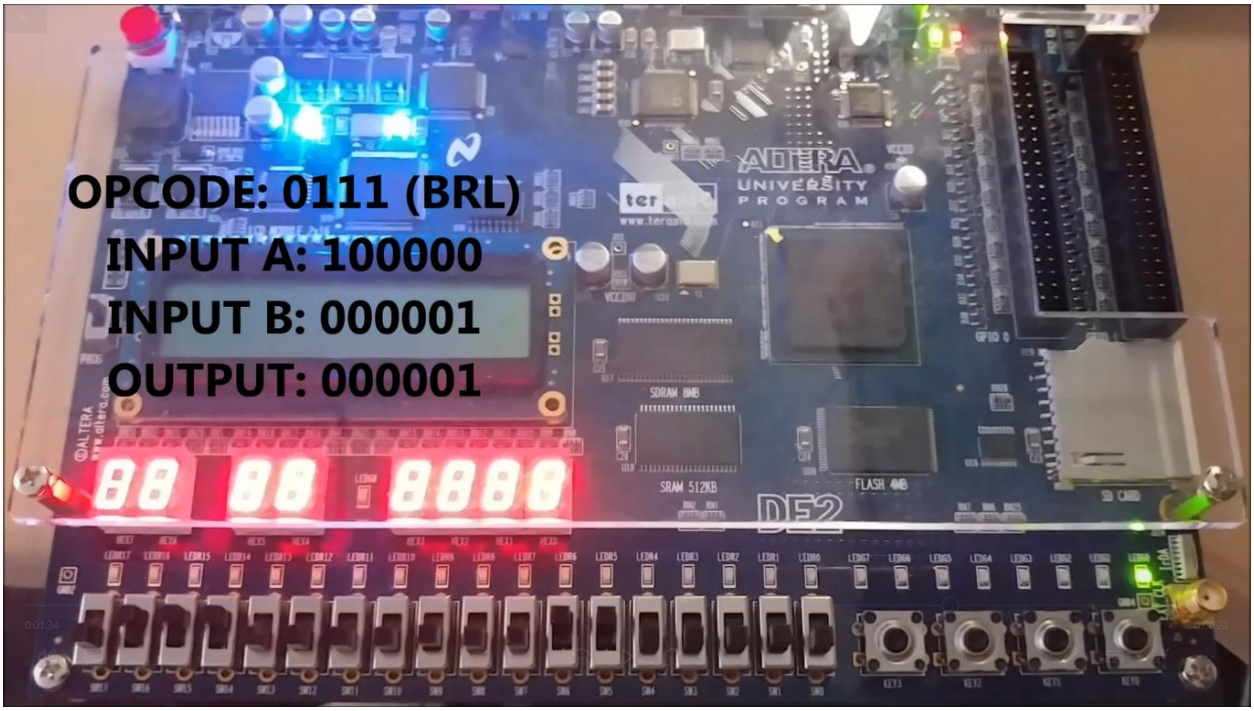
Bit Shift Left



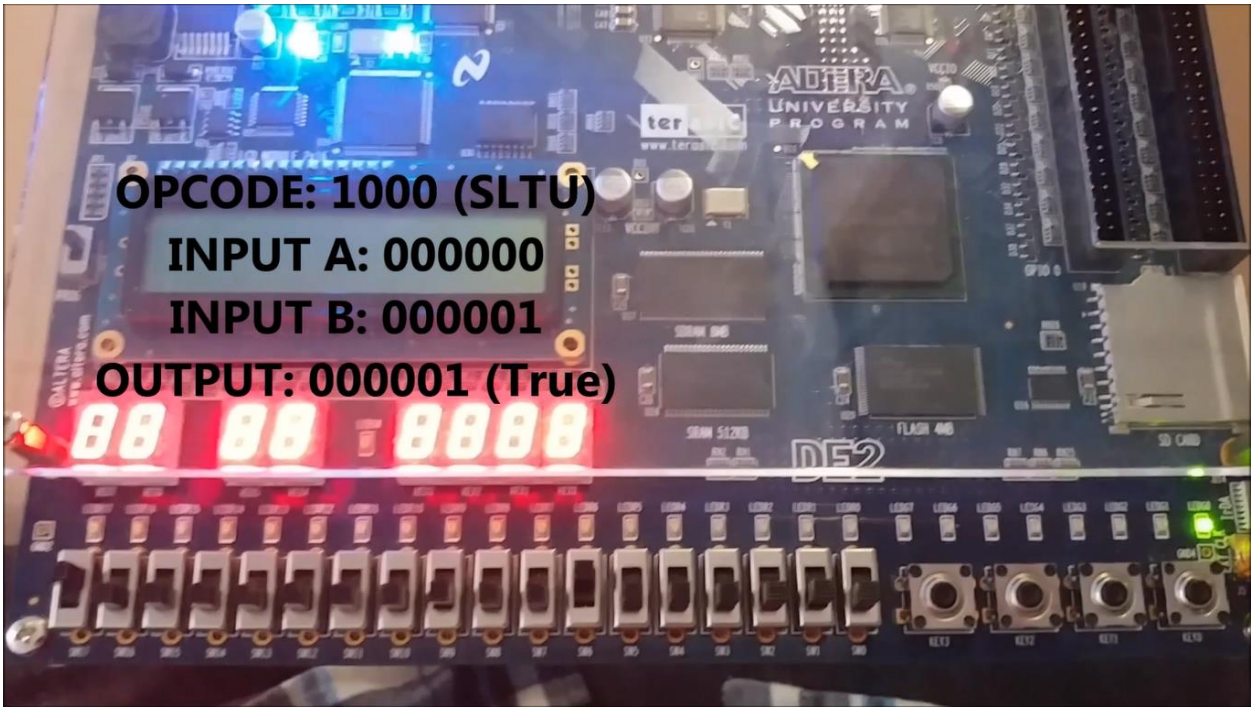
Bit Rotation Right



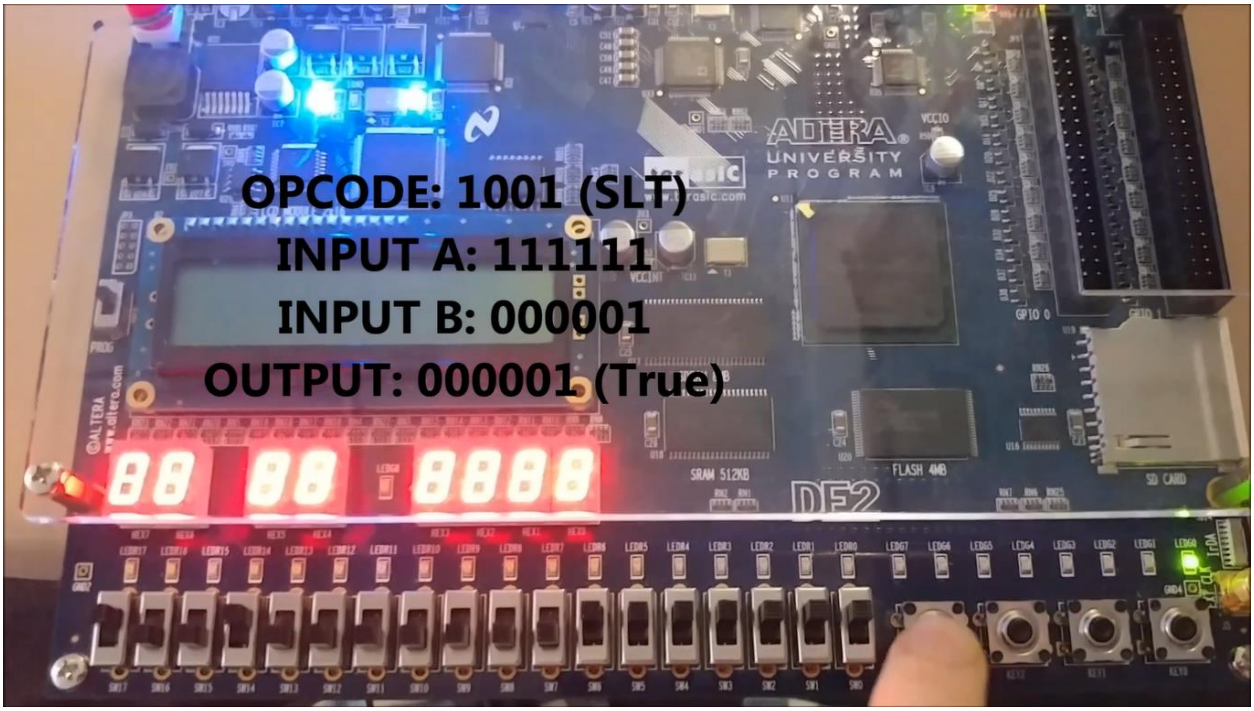
Bit Rotation Left



Set Less Than Unsigned



Set Less Than Signed



Section 5) Conclusion

In this lab I designed various circuits, with designing one overall circuit at the end. I would say it was much easier designing everything as one overall circuit. My original design used a schematic importing all the symbols I created for each function. This became problematic with wiring, pin assignments, etc. Very messy and tedious. I realized later I can just have an opcode select the functions. Due to some unfortunate family issues, my mind was just not in the right place and I wasn't thinking straight, so I believe that led me to start off with a bad design. Other than that, this lab was straight forward. In fact, I would say the easiest out of them all so far. I would say I learned a good amount in the differences between operations and **bitwise** operations.