Lab 5 INTEGRATION PROJECT

CSC 343 Fall 2017

October 23, 2017

Jeter Gutierrez

TABLE OF CONTENTS

1. Objective

In this lab we will be implementing several bit wise operations using VHDL as well as an adder and subtractor, we will also be using LPM modules to implement addition and subtraction for n-bits. We will design a 32 bit adder and test a 6 bit adder on DE2-115 board. The purpose of this lab is for us to gain an understanding in how bitwise operations work and how opt code works in our integration project as well in combination with an adder and subtractor. We need to understand how opt code works in order to gain an understanding in processors like Mips, Intel, AMD, CORTEX, and SNAP DRAGON. Once we understand how processors use opt code to perform several operations efficiently and effectively by doing them quickly and without wasting memory, then we can begin to step in the direction of designing our own processor. We will design each component individually in order to test the structure of the design and then we will design a larger component which functions using each of the smaller components and takes opt codes as inputs in order to perform each different task. We will be designing these circuits in VHDL using Quartus Prime Software. We will be testing the circuits using modelsim waveform simulation. Another method that will be used to test the circuits is testbench with predefined values for each of the n-bit words. The results of our design will be tested using Modelsim, Testbench code and on DE2-115 board, we will use 7-segment displays in hexadecimal formats

for outputting our data. After we have completed designing the circuits we will run the circuits in simulation to assert that they are working as expected. Then we will demonstrate how each circuit works by programming our design into a programmable DE2-115 FPGA board.

The circuits we will be designing in this lab are:

1. 32-bit adder
2. 32-bit subtractor
3. Bitwise and
4. Bitwise or
5. Bitwise xor
6. Bitwise not
7. Shift left
8. Shift right
9. Rotate left
10. Rotate right
11. Set less than.
12. Opcode component.

2. 32 BIT ADDER

2.1 Functionality and specifications for 32 bit adder.

The purpose of the 32 bit adder is to add 2 32 bit words and keep track of the overflow value, the carry output and total sum. The input values will be stored in X and Y which are both 32 bit vectors in vhdl, the sum of the two bits will be stored in sum_prime, the overflow will be stored in Overflow and the final carry value will be stored in Cout. If a value for the sum of the 2 32 bit

words is out of range from the 32 bit word capacity the value of overflow will be 1 or true, this means that the value in sum_prime will be an incorrect value and not useful information. This circuit will be capable of adding 2 32 bit words in signed, unsigned formats correctly keeping track of overflow.

```
library ieee;--Jeter Gutierrez September 24 ,2017
use ieee.std_logic_1164.all;--Jeter Gutierrez September 24 ,2017
use ieee.std_logic_arith.all;--Jeter Gutierrez September 24 ,2017
use ieee.std_logic_signed.all;--Jeter Gutierrez September 24 ,2017
use work.GUTIERREZ_ADDER_PACKAGE_COMPARE.all;--Jeter Gutierrez September 24 ,2017
entity GUTIERREZ_ADDER_32_BITS is--Jeter Gutierrez September 24 ,2017
    port ( cin: in std_logic;--Jeter Gutierrez September 24 ,2017
             X, Y: in std_logic_vector(31 downto 0);--Jeter Gutierrez September 24 ,2017
             sum_prime: out std_logic_vector (31 downto 0);--Jeter Gutierrez September 24 ,2017
             Cout, Overflow: out std_logic);--Jeter Gutierrez September 24 ,2017
end entity;--Jeter Gutierrez September 24 ,2017

architecture DESIGN_32 of GUTIERREZ_ADDER_32_BITS is --Jeter Gutierrez September 24 ,2017
    component GUTIERREZ_FULL_ADDER_USING_GATES--Jeter Gutierrez September 24 ,2017
        Port ( a,b,cin_prime : in  STD_LOGIC;--Jeter Gutierrez September 24 ,2017
             sum,cout_prime : out  STD_LOGIC);--Jeter Gutierrez September 24 ,2017
    end component;--Jeter Gutierrez September 24 ,2017

    signal C: std_logic_vector(31 downto 0);--Jeter Gutierrez September 24 ,2017
    signal sum_temp: std_logic_vector(31 downto 0);--Jeter Gutierrez September 24 ,2017
    begin--Jeter Gutierrez September 24 ,2017
        FIRST: GUTIERREZ_FULL_ADDER_USING_GATES port map  (cin, X(0),Y(0),sum_temp(0),C(0));--Jeter C
        SECOND: Cout<= C(31);--Jeter Gutierrez September 24 ,2017
        --Jeter Gutierrez September 24 ,2017
    THIRD:--Jeter Gutierrez September 24 ,2017
        for i in 1 to (31) generate--Jeter Gutierrez September 24 ,2017
            FOURTH: GUTIERREZ_FULL_ADDER_USING_GATES port map  (C(i-1), X(i),Y(i),sum_temp(i),C(i));-
        end generate;--Jeter Gutierrez September 24 ,2017
        FIFTH: Overflow<= '1' WHEN ((X(31)=Y(31) AND sum_temp(31)/=X(31))) ELSE '0';--Jeter Gutierrez
        FINAL: sum_prime<=sum_temp;--Jeter Gutierrez September 24 ,2017
end DESIGN_32;--Jeter Gutierrez September 24 ,2017
```

Figure 1: VHDL code for 32 bit adder.

2.2 Simulation for 32 bit adder

In this simulation X and Y will be given different values, in order to add them together, their result will be sent to sum_prime, we will also keep track of overflow and carry out values, then we will observe the simulation and consider its correctness.

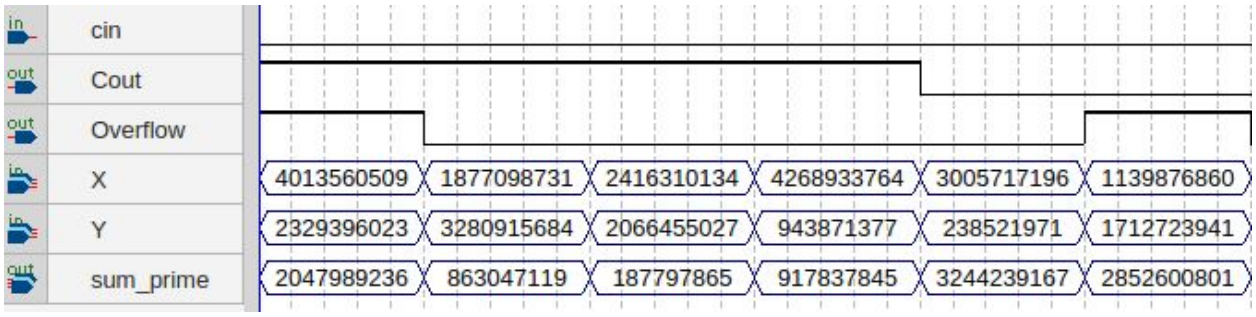

Figure 2: Vector waveform simulation for 32 bit adder. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design a 32 bit adder.

3. 32 BIT ADDER TEST

3.1 Functionality and specifications for 32 bit adder test.

The purpose of this circuit is to create a test bench in order to simulate our 32 bit adder. We already simulated our 32 bit adder in Vector waveform but it is more professional and exact to use a test bench to test the correctness of our circuits because we have more control over the simulation this way. The test bench imitates a physical lab bench making our simulation cleaner and more organized.

```
library ieee;
use ieee.std_logic_1164.all;--Jeter Gutierrez September 24 ,2017
use ieee.numeric_std.all;--Jeter Gutierrez September 24 ,2017
use ieee.std_logic_unsigned.all;--Jeter Gutierrez September 24 ,2017
use work.GUTIERREZ_ADDER_PACKAGE_COMPARE.all;--Jeter Gutierrez September 24 ,2017
--USE work.fulladd_package.all ;--Jeter Gutierrez September 24 ,2017
entity GUTIERREZ_TEST_32_BIT_ADDER is--Jeter Gutierrez September 24 ,2017
end GUTIERREZ_TEST_32_BIT_ADDER;--Jeter Gutierrez September 24 ,2017
architecture arch_test_32 of GUTIERREZ_TEST_32_BIT_ADDER is--Jeter Gutierrez September 24 ,2017
--componengt declaration for the Unit Under Test--Jeter Gutierrez September 24 ,2017
component GUTIERREZ_ADDER_32_BITS--Jeter Gutierrez September 24 ,2017
PORT (Cin: IN STD_LOGIC;--Jeter Gutierrez September 24 ,2017
X,Y: IN STD_LOGIC_VECTOR(31 DOWNTO 0);--Jeter Gutierrez September 24 ,2017
sum_prime: OUT STD_LOGIC_VECTOR(31 DOWNTO 0);--Jeter Gutierrez September 24 ,2017
Cout,Overflow: OUT STD_LOGIC );--Jeter Gutierrez September 24 ,2017
end component;--Jeter Gutierrez September 24 ,2017
signal A,B,S :STD_LOGIC_VECTOR(31 DOWNTO 0);--Jeter Gutierrez September 24 ,2017
signal Ci,Co,Overflow2 :STD_LOGIC;--Jeter Gutierrez September 24 ,2017
begin--Jeter Gutierrez September 24 ,2017
-----Instantiate the Unit Under Test (UUT--Jeter Gutierrez September 24 ,2017)
uut: GUTIERREZ_ADDER_32_BITS port map (--Jeter Gutierrez September 24 ,2017
Cin => Ci,--Jeter Gutierrez September 24 ,2017
X => A,--Jeter Gutierrez September 24 ,2017
Y => B,--Jeter Gutierrez September 24 ,2017
sum_prime => S,--Jeter Gutierrez September 24 ,2017
Cout =>Co,--Jeter Gutierrez September 24 ,2017
Overflow => Overflow2--Jeter Gutierrez September 24 ,2017
);--Jeter Gutierrez September 24 ,2017
---- Test Bench ---User Defined Proces--Jeter Gutierrez September 24 ,2017s
tb : process--Jeter Gutierrez September 24 ,2017
begin--Jeter Gutierrez September 24 ,2017
--Hold reset state for 100 ns--Jeter Gutierrez September 24 ,2017
wait for 100 ns;--Jeter Gutierrez September 24 ,2017
report "TESTING 32 BIT ADDER";--Jeter Gutierrez September 24 ,2017
A<="00000000000000000000000000000000";--Jeter Gutierrez September 24 ,2017
B<="00000000000000000000000000000000";--Jeter Gutierrez September 24 ,2017
Ci<='0';--Jeter Gutierrez September 24 ,2017
--Loop over all values of A--Jeter Gutierrez September 24 ,2017
for I in 0 to 256 loop--Jeter Gutierrez September 24 ,2017
--Loop over all values of B--Jeter Gutierrez September 24 ,2017
for J in 0 to 256 loop--Jeter Gutierrez September 24 ,2017
--Wait for outputto update--Jeter Gutierrez September 24 ,2017
wait for 10 ns;--Jeter Gutierrez September 24 ,2017
---report " the A+B = " & integer'image(to_integer(unsigned((A+B))));--Jeter Gutierrez September 24 ,2017
---The statement below checks for ALL possible input values if the ouput is correct.--Jeter Gutierrez Sep
assert (S = A+B) report "The sum from 32 bit adder is S= " & integer'image(to_integer(unsigned((S)))) &--
" while the expected A+B = " & integer'image(to_integer(unsigned((A+B)))) severity ERROR;--Jeter Gutierre
assert(overflow2=overflow2) report "Overflow is wrong" severity ERROR;--Jeter Gutierrez September 24 ,201
--Increment to the next value of B--Jeter Gutierrez September 24 ,2017
B<=B+"00000000000000000000000000000001";--Jeter Gutierrez September 24 ,2017
end loop;--Jeter Gutierrez September 24 ,2017
--Increment to the next value of A--Jeter Gutierrez September 24 ,2017
A<=A+"00000000000000000000000000000001";--Jeter Gutierrez September 24 ,2017
--Echo to users test is finished--Jeter Gutierrez September 24 ,2017
end loop;--Jeter Gutierrez September 24 ,2017
report "Test completed";--Jeter Gutierrez September 24 ,2017
wait; -- will wait for ever--Jeter Gutierrez September 24 ,2017
end process;--Jeter Gutierrez September 24 ,2017
---END User Defined Process--Jeter Gutierrez September 24 ,2017
end arch_test_32;--Jeter Gutierrez September 24 ,2017
```

Figure 3: VHDL code for the test bench for testing 32 bit adder. In this design modelsim will be used in order to simulate what is written in the testing code. It will be used to test the values for 2

32 bit words by increasing each one by 1 bit and determine whether for every possible case of adding 2 32 bit words after a certain period of time or after testing another state it will continue to be correct. The reason we do this is to have more control over our testing for correctness of our 32 bit adder.

3.2 Simulation for 32 bit adder Test.

In this simulation we will be using modelsim to run our test bench file for the 32 bit adder, if we get no errors we know we have designed a 32 bit adder correctly. The difference in this state is that we already wrote what values we want to test for and modelsim will create those values for us instead of us having to use the cursor to select the values using waveform. Using a test bench is more efficient than manually inputting values to test, this way we test every possible value.

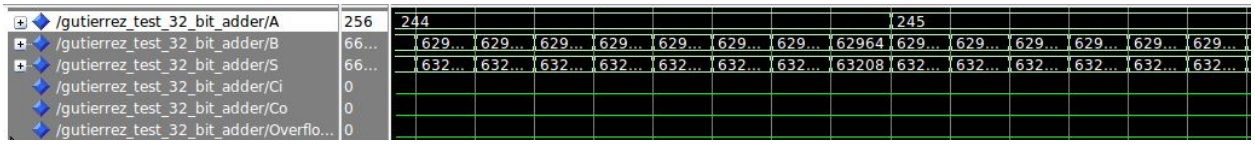

Figure 4: Vector waveform simulation for for test bench of 32 bit adder test. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design a 32 bit adder test. The test bench here was used to test every possible combination of 32-Bit words.

4. 32 BIT ADDER SUBTRACTOR

4.1 Functionality and specifications for 32 bit adder subtractor.

The purpose of the 32 bit adder subtractor is to add or subtract 2 32 bit words and keep track of the overflow value, the carry output and total sum or difference. The input values will be stored in X and Y which are both 32 bit vectors in vhdl, the sum of the two bits will be stored in S, the overflow will be stored in Overflow and the final carry value will be stored in Cout. If a value for

the sum or difference of the 2 32 bit words is out of range from the 32 bit word capacity the value of overflow will be 1 or true, this means that the value in S will be an incorrect value and not useful information. This circuit will be capable of adding 2 32 bit words in signed, unsigned formats correctly keeping track of overflow. It will also be capable of subtracting 2 32 bit words when the signal for SUBTRACT is 1, when it is 0 the circuit will add the 2 32 bit words. The idea behind keeping track of overflow is considering that adding 2 positive numbers should ideally result in another positive number, and that subtracting two negative numbers should result in a negative number as well, that is how we keep track of overflow in the design of this 32 bit adder subtractor just like we did with the 4 bit adder subtractor and the 16 bit adder subtractor.

```
library IEEE;--Jeter Gutierrez September 24 ,2017
use IEEE.std_logic_1164.all;--Jeter Gutierrez September 24 ,2017
use IEEE.std_logic_arith.all;--Jeter Gutierrez September 24 ,2017
use IEEE.std_logic_signed.all;--Jeter Gutierrez September 24 ,2017
use work.GUTIERREZ_ADDER_PACKAGE_COMPARE.all;--Jeter Gutierrez September 24 ,2017
entity GUTIERREZ_ADD_SUB_32_BITS is--Jeter Gutierrez September 24 ,2017
    port ( SUBTRACT: in std_logic;--Jeter Gutierrez September 24 ,2017
             X, Y: in std_logic_vector(31 downto 0);--Jeter Gutierrez September 24 ,2017
             S: out std_logic_vector (31 downto 0);--Jeter Gutierrez September 24 ,2017
             Cout, Overflow: out std_logic);--Jeter Gutierrez September 24 ,2017
end entity;--Jeter Gutierrez September 24 ,2017
architecture STRUCTURE_32 of GUTIERREZ_ADD_SUB_32_BITS is --Jeter Gutierrez September 24 ,2017
component GUTIERREZ_ADDER_32_BITS is --Jeter Gutierrez September 24 ,2017
    port ( cin: in std_logic;--Jeter Gutierrez September 24 ,2017
             X, Y: in std_logic_vector(31 downto 0);--Jeter Gutierrez September 24 ,2017
             sum_prime: out std_logic_vector (31 downto 0);--Jeter Gutierrez September 24 ,2017
             Cout, Overflow: out std_logic);--Jeter Gutierrez September 24 ,2017
end component;--Jeter Gutierrez September 24 ,2017
    signal COMPLEMENT: std_logic_vector(31 downto 0);--Jeter Gutierrez September 24 ,2017
    signal overflow2: std_logic;--Jeter Gutierrez September 24 ,2017
    signal SUM_TEMP: std_logic_vector(31 downto 0);--Jeter Gutierrez September 24 ,2017
    begin--Jeter Gutierrez September 24 ,2017
    FIRST:--Jeter Gutierrez September 24 ,2017
        for i in 0 to (31) generate--Jeter Gutierrez September 24 ,2017
            INVERSE: COMPLEMENT(i) <= Y(i) xor SUBTRACT;--Jeter Gutierrez September 24 ,2017
        end generate;--Jeter Gutierrez September 24 ,2017
    SECOND : GUTIERREZ_ADDER_32_BITS port map(SUBTRACT, X, COMPLEMENT, SUM_TEMP, Cout, Overflow2);--Jet
    Overflow<= '1' WHEN (SUBTRACT='0' AND X(31)=COMPLEMENT(31) AND SUM_TEMP(31)/=X(31)) or--Jeter Gutie
                        (SUBTRACT='1' AND X(31)/=COMPLEMENT(31) AND SUM_TEMP(31)/=X(31)) ELSE '0';-
    S<=SUM_TEMP;--Jeter Gutierrez September 24 ,2017
end STRUCTURE_32;--Jeter Gutierrez September 24 ,2017
```

Figure 5: VHDL code for 32 bit adder subtractor.

4.2 Simulation for 32 bit adder Subtractor

In this simulation X and Y will be given different values, in order to add them together or subtract them from each other based on the value of SUBTRACT, their result will be sent to S, we will also keep track of overflow and carry out values, then we will observe the simulation and consider its correctness.

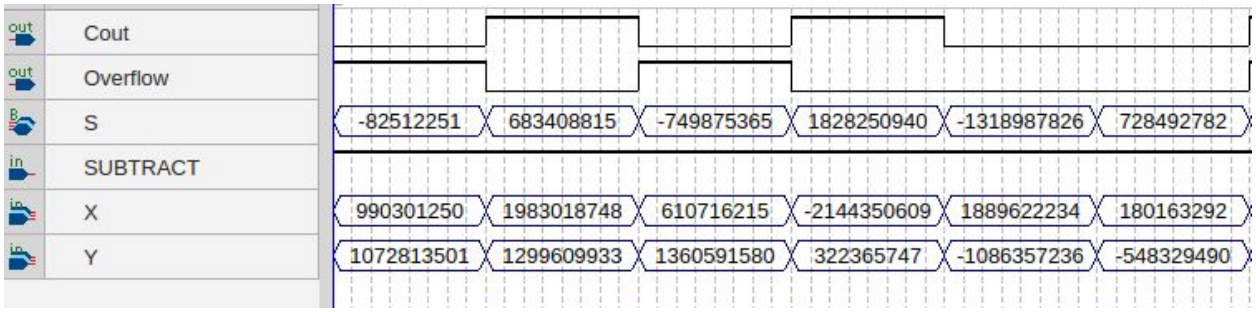

Figure 6: Vector waveform simulation for 32 bit adder subtractor. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design a 32 bit adder subtractor.

5. 32 BIT ADDER SUBTRACTOR TEST

5.1 Functionality and specifications for 32 bit adder subtractor test.

The purpose of this circuit is to create a test bench in order to simulate our 32 bit adder subtractor. We already simulated our 32 bit adder subtractor in Vector waveform but it is more professional and exact to use a test bench to test the correctness of our circuits because we have more control over the simulation this way. The test bench imitates a physical lab bench making our simulation cleaner and more organized.

```
library ieee;--Jeter Gutierrez September 24 ,2017
use ieee.std_logic_1164.all;--Jeter Gutierrez September 24 ,2017
use ieee.numeric_std.all;--Jeter Gutierrez September 24 ,2017
use ieee.std_logic_unsigned.all;--Jeter Gutierrez September 24 ,2017
use work.GUTIERREZ_ADDER_PACKAGE_COMPARE.all;--Jeter Gutierrez September 24 ,2017
--USE work.fulladd_package.all ;--Jeter Gutierrez September 24 ,2017
entity GUTIERREZ_TEST_32_BIT_ADD_SUB is--Jeter Gutierrez September 24 ,2017
end GUTIERREZ_TEST_32_BIT_ADD_SUB;--Jeter Gutierrez September 24 ,2017
architecture arch_test of GUTIERREZ_TEST_32_BIT_ADD_SUB is--Jeter Gutierrez September 24 ,2017
--componengt declaration for the Unit Under Test--Jeter Gutierrez September 24 ,2017
component GUTIERREZ_ADD_SUB_32_BITS--Jeter Gutierrez September 24 ,2017
PORT (SUBTRACT: IN STD_LOGIC;--Jeter Gutierrez September 24 ,2017
X,Y: IN STD_LOGIC_VECTOR(31 DOWNTO 0);--Jeter Gutierrez September 24 ,2017
S: OUT STD_LOGIC_VECTOR(31 DOWNTO 0);--Jeter Gutierrez September 24 ,2017
Cout,Overflow: OUT STD_LOGIC );--Jeter Gutierrez September 24 ,2017
end component;--Jeter Gutierrez September 24 ,2017
signal A,B,Sum_temp :STD_LOGIC_VECTOR(31 DOWNTO 0);--Jeter Gutierrez September 24 ,2017
signal Ci,Co,Overflow2 :STD_LOGIC;--Jeter Gutierrez September 24 ,2017
begin--Jeter Gutierrez September 24 ,2017
-----Instantiate the Unit Under Test (UUT--Jeter Gutierrez September 24 ,2017)
uut: GUTIERREZ_ADD_SUB_32_BITS port map (--Jeter Gutierrez September 24 ,2017
SUBTRACT => Ci,--Jeter Gutierrez September 24 ,2017
X => A,--Jeter Gutierrez September 24 ,2017
Y => B,--Jeter Gutierrez September 24 ,2017
S => Sum_temp,--Jeter Gutierrez September 24 ,2017
Cout =>Co,--Jeter Gutierrez September 24 ,2017
Overflow => Overflow2--Jeter Gutierrez September 24 ,2017
);--Jeter Gutierrez September 24 ,2017
---- Test Bench ---User Defined Process--Jeter Gutierrez September 24 ,2017
tb : process--Jeter Gutierrez September 24 ,2017
begin--Jeter Gutierrez September 24 ,2017
--Hold reset state for 100 ns--Jeter Gutierrez September 24 ,2017
wait for 100 ns;--Jeter Gutierrez September 24 ,2017
report "TESTING 32 BIT SUBTRACTOR";--Jeter Gutierrez September 24 ,2017
A<="00000000000000000000000000000000";--Jeter Gutierrez September 24 ,2017
B<="00000000000000000000000000000000";--Jeter Gutierrez September 24 ,2017
Ci<='1';--Jeter Gutierrez September 24 ,2017
--Loop over all values of A--Jeter Gutierrez September 24 ,2017
for I in 0 to 256 loop--Jeter Gutierrez September 24 ,2017
--Loop over all values of B--Jeter Gutierrez September 24 ,2017
for J in 0 to 256 loop--Jeter Gutierrez September 24 ,2017
--Wait for outputto update--Jeter Gutierrez September 24 ,2017
wait for 10 ns;--Jeter Gutierrez September 24 ,2017
---report " the A+B = " & integer'image(to_integer(unsigned((A+B))));--Jeter Gutierrez September 24 ,2017
---The statement below checks for ALL possible input values if the ouput is correct.--Jeter Gutierrez September 24 ,2017
assert (Sum_temp = A-B) report "The DIFFERENCE FROM THE 32 BIT SUBTRACTOR IS S= " & integer'image(to_integer(signed((Sum_temp)))) &-
" while the expected A-B = " & integer'image(to_integer(signed((A-B)))) severity ERROR;--Jeter Gutierrez September 24 ,2017
assert (overfow2=overflow2) report "Overflow is wrong" severity ERROR;--Jeter Gutierrez September 24 ,2017
--Increment to the next value of B--Jeter Gutierrez September 24 ,2017
B<=B+"00000000000000000000000000000001";--Jeter Gutierrez September 24 ,2017
end loop;--Jeter Gutierrez September 24 ,2017
--Increment to the next value of A--Jeter Gutierrez September 24 ,2017
A<=A+"00000000000000000000000000000001";--Jeter Gutierrez September 24 ,2017
--Echo to users test is finished--Jeter Gutierrez September 24 ,2017
end loop;--Jeter Gutierrez September 24 ,2017
report "Test completed";--Jeter Gutierrez September 24 ,2017
wait; -- will wait for ever--Jeter Gutierrez September 24 ,2017
end process;--Jeter Gutierrez September 24 ,2017
---END User Defined Process--Jeter Gutierrez September 24 ,2017
end arch_test;--Jeter Gutierrez September 24 ,2017
```

Figure 7: VHDL code for the test bench for testing 32 bit adder subtractor. In this design modelsim will be used in order to simulate what is written in the testing code. It will be used to test the values for 2 32 bit words by increasing each one by 1 bit and determine whether for every possible case of adding or subtracting 2 32 bit words after a certain period of time or after testing another state it will continue to be correct. The reason we do this is to have more control over our testing for correctness of our 32 bit adder subtractor.

5.2 Simulation for 32 bit adder subtractor Test.

In this simulation we will be using modelsim to run our test bench file for the 32 bit adder subtractor, if we get no errors we know we have designed a 32 bit adder subtractor correctly. The difference in this state is that we already wrote what values we want to test for and modelsim will create those values for us instead of us having to use the cursor to select the values using waveform. Using a test bench is more efficient than manually inputting values to test, this way we test every possible value.

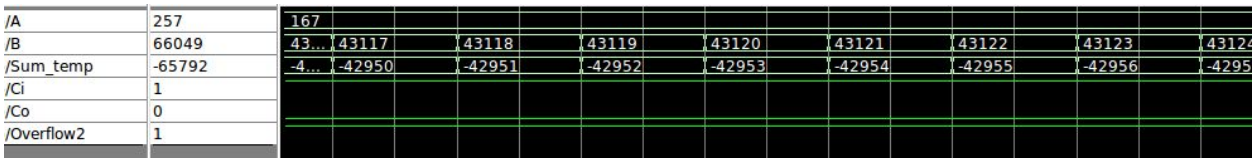

Figure 8: Vector waveform simulation for for test bench of 32 bit adder subtractor test. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design a 32 bit adder subtractor test. The test bench here was used to test every possible combination of 32-Bit words.

6. Bitwise and.

6.1 Functionality and specifications for bitwise and.

The purpose of this circuit is to take as an input two n- bit words, in this case we will be taking 6 bit words for each input. The result of this component is to perform the bitwise and operation on each of the bits in pairs. The only time we should have a value of 1 in a bit of a result is when it is 1 in both X input and Y input. If in a given position we do not have a value of 1 on both X and Y then that position on the result will be 0 instead of 1.

```
library ieee;--Jeter Gutierrez: October, 4, 2017|
use ieee.std_logic_1164.all;--Jeter Gutierrez: October, 4, 2017|
use work.GUTIERREZ_OPT_CODE_PACKAGE.all;--Jeter Gutierrez: October, 4, 2017|
entity GUTIERREZ_BITWISE_AND is--Jeter Gutierrez: October, 4, 2017|
port( X: in std_logic_vector(5 downto 0);--Jeter Gutierrez: October, 4, 2017|
        Y:  in std_logic_vector (5 downto 0);--Jeter Gutierrez: October, 4, 2017|
        RESULT: out std_logic_vector (5 downto 0));--Jeter Gutierrez: October, 4, 2017|
end GUTIERREZ_BITWISE_AND;--Jeter Gutierrez: October, 4, 2017|
architecture design_and of GUTIERREZ_BITWISE_AND is--Jeter Gutierrez: October, 4, 2017|
begin--Jeter Gutierrez: October, 4, 2017|
RESULT<=X and Y;--Jeter Gutierrez: October, 4, 2017|
end design_and;--Jeter Gutierrez: October, 4, 2017|
```

Figure 9: Vhdl Code for bitwise and.

6.2 Simulation for bitwise and.

In this simulation X and Y will be given different values, in order to perform bitwise and, their result will be sent to RESULT then we will observe the simulation and consider its correctness.

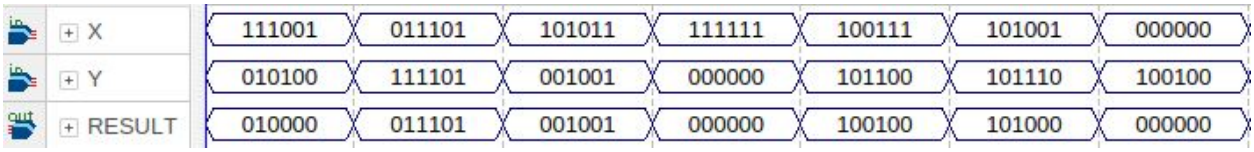

Figure 10: Vector waveform simulation for bitwise and. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design a bitwise and component. For every position in an instance of X and Y where they share a value of 1 it is also 1 in the result which means that it is correct.

7. Bitwise Or.

7.1 Functionality and specifications for bitwise or.

The purpose of this circuit is to take as an input two n- bit words, in this case we will be taking 6 bit words for each input. The result of this component is to perform the bitwise or operation on each of the bits in pairs. The only time we should have a value of 1 in a bit of a result is when it

is 1 in either X input or Y input or both. If in a given position we do not have a value of 1 on either X or Y then that position on the result will be 0 instead of 1.

```
library ieee;--Jeter Gutierrez: October, 4, 2017|
use ieee.std_logic_1164.all;--Jeter Gutierrez: Octob
use work.GUTIERREZ_OPT_CODE_PACKAGE.all;--Jeter Gut:
entity GUTIERREZ_BITWISE_OR is--Jeter Gutierrez: Oct
port( X: in std_logic_vector(5 downto 0);--Jeter Gut
      Y: in std_logic_vector (5 downto 0);--Jeter Gu
      RESULT: out std_logic_vector (5 downto 0));--
end GUTIERREZ_BITWISE_OR;--Jeter Gutierrez: October,
architecture design_OR of GUTIERREZ_BITWISE_OR is--
begin--Jeter Gutierrez: October, 4, 2017|
RESULT<=X or Y;--Jeter Gutierrez: October, 4, 2017|
end design_OR;--Jeter Gutierrez: October, 4, 2017|
```

Figure 11: VHDL code for bitwise or.

7.2 Simulation for bitwise or.

In this simulation X and Y will be given different values, in order to perform bitwise or, their result will be sent to RESULT then we will observe the simulation and consider its correctness.

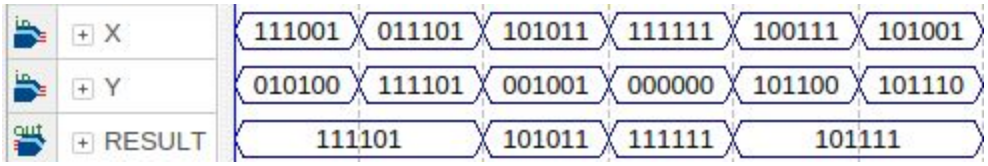

Figure 12: Vector waveform simulation for bitwise or. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design a bitwise or component. For every position in an instance of X and Y where either has a value of 1 it is also 1 in the result which means that it is correct.

8. Bitwise XOR.

8.1 Functionality and specifications for bitwise XOR.

The purpose of this circuit is to take as an input two n- bit words, in this case we will be taking 6 bit words for each input. The result of this component is to perform the bitwise xor operation on each of the bits in pairs. The only time we should have a value of 1 in a bit of a result is when it is 1 in either X input or Y input but not both. If in a given position we do not have a value of 1 on either X or Y then that position on the result will be 0 instead of 1.

```
library ieee;--Jeter Gutierrez: October, 4, 2017|
use ieee.std_logic_1164.all;--Jeter Gutierrez: October, 4, 2017|
use work.GUTIERREZ_OPT_CODE_PACKAGE.all;--Jeter Gutierrez: October, 4, 2017|
entity GUTIERREZ_BITWISE_XOR is--Jeter Gutierrez: October, 4, 2017|
port( X: in std_logic_vector(5 downto 0);--Jeter Gutierrez: October, 4, 2017|
      Y: in std_logic_vector (5 downto 0);--Jeter Gutierrez: October, 4, 2017|
      RESULT: out std_logic_vector (5 downto 0));--Jeter Gutierrez: October, 4, 2017|
end GUTIERREZ_BITWISE_XOR;--Jeter Gutierrez: October, 4, 2017|
architecture design_XOR of GUTIERREZ_BITWISE_XOR is--Jeter Gutierrez: October, 4, 2017|
begin--Jeter Gutierrez: October, 4, 2017|
RESULT<=X xor Y;--Jeter Gutierrez: October, 4, 2017|
end design_XOR;--Jeter Gutierrez: October, 4, 2017|
```

Figure 13: VHDL code for Bitwise XOR.

8.2 Simulation for bitwise XOR.

In this simulation X and Y will be given different values, in order to perform bitwise xor, their result will be sent to RESULT then we will observe the simulation and consider its correctness.

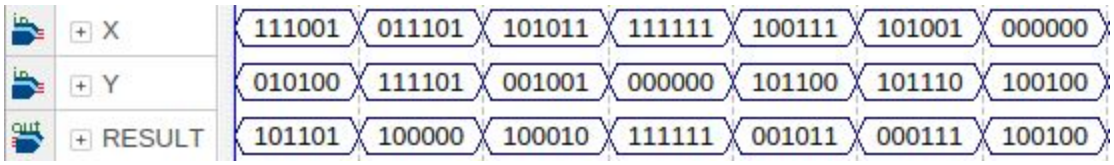

Figure 14: Vector waveform simulation for bitwise xor. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design a bitwise xor component. For every position in an instance of X and Y where either has a value of 1 but not both it is also 1 in the result which means that it is correct.

9. Bitwise NOT.

9.1 Functionality and specifications for bitwise NOT.

The purpose of this circuit is to take as an input one n- bit words, in this case we will be taking a 6 bit word as an input. The result of this component is to perform the bitwise not operation on each bit in the word. The only time we should have a value of 1 in a bit of a result is when it is 0 in X and we should have a 0 in the RESULT when there is a 1 in X.

```
library ieee;--Jeter Gutierrez: October, 4, 2017|
use ieee.std_logic_1164.all;--Jeter Gutierrez: Octob
use work.GUTIERREZ_OPT_CODE_PACKAGE.all;--Jeter Guti
entity GUTIERREZ_BITWISE_NOT is--Jeter Gutierrez: Oc
port( X: in std_logic_vector(5 downto 0);--Jeter Gut
    RESULT: out std_logic_vector (5 downto 0));--J
end GUTIERREZ_BITWISE_NOT;--Jeter Gutierrez: October,
architecture design_NOT of GUTIERREZ_BITWISE_NOT is-
begin--Jeter Gutierrez: October, 4, 2017|
RESULT<=not X;--Jeter Gutierrez: October, 4, 2017|
end design_NOT;--Jeter Gutierrez: October, 4, 2017|
```

Figure 15: VHDL code for bitwise not.

9.2 Simulation for bitwise NOT.

In this simulation X will be given different values, in order to perform bitwise not, the result will be sent to RESULT then we will observe the simulation and consider its correctness.

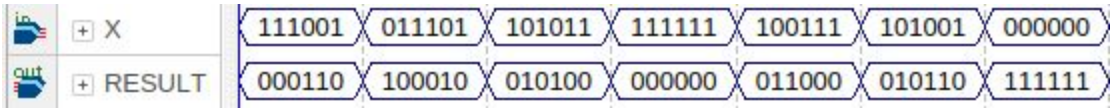

Figure 16: Vector waveform simulation for bitwise not. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design a bitwise not component. For every instance in X where a bit is equal to 1 in the same position in the RESULT we get a value of 0, and for every instance in X where a bit is equal to 0 we get a value of 1 in RESULT.

10. Shift Left.

10.1 Functionality and specifications for shift left.

The purpose of this circuit is to take as an input one n- bit word, in this case we will be taking a 6 bit word as an input. The result of this component is to perform the shift left operation on each bit in the word. After all the bits are shifted to the left then any empty bits will be automatically set to 0.

```
library ieee;--Jeter Gutierrez: October, 4, 2017|
use ieee.std_logic_1164.all;--Jeter Gutierrez: October, 4, 2017|
use work.GUTIERREZ_OPT_CODE_PACKAGE.all;--Jeter Gutierrez: October
entity GUTIERREZ_BITWISE_SHIFT_LEFT is--Jeter Gutierrez: October,
port( X: in std_logic_vector(5 downto 0);--Jeter Gutierrez: Octobe
    RESULT: out std_logic_vector (5 downto 0));--Jeter Gutierrez
end GUTIERREZ_BITWISE_SHIFT_LEFT;--Jeter Gutierrez: October, 4, 20
architecture design_SHIFT_LEFT of GUTIERREZ_BITWISE_SHIFT_LEFT is
begin--Jeter Gutierrez: October, 4, 2017|
RESULT<=to_stdlogicvector(to_bitvector(X)sll 1);--Jeter Gutierrez:
end design_SHIFT_LEFT;--Jeter Gutierrez: October, 4, 2017|
```

Figure 17: VHDL code for shift left.

10.2 Simulation for shift left.

In this simulation X will be given different values, in order to perform shift left, the result will be sent to RESULT then we will observe the simulation and consider its correctness.

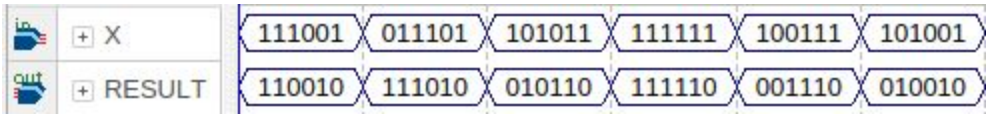

Figure 18: Vector waveform simulation for shift left. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design a bitwise shift left component. For every instance in X where a bit is equal to 1 in a position to the left in the RESULT we get a value of 1, and for every instance in X where a bit is equal to 0 we get a value of 1 in RESULT a position to the left, while the rightmost bit is set to 0 each time because we don't have any bits to shift in its position.

11. Shift Right.

11.1 Functionality and specifications for shift right.

The purpose of this circuit is to take as an input one n- bit word, in this case we will be taking a 6 bit word as an input. The result of this component is to perform the shift right operation on each bit in the word. After all the bits are shifted to the left then any empty bits will be automatically set to 0.

```
library ieee;--Jeter Gutierrez: October, 4, 2017|
use ieee.std_logic_1164.all;--Jeter Gutierrez: October, 4, 2017|
use work.GUTIERREZ_OPT_CODE_PACKAGE.all;--Jeter Gutierrez: October,
entity GUTIERREZ_BITWISE_SHIFT_RIGHT is--Jeter Gutierrez: October, 4,
port( X: in std_logic_vector(5 downto 0);--Jeter Gutierrez: October,
    RESULT: out std_logic_vector (5 downto 0));--Jeter Gutierrez:
end GUTIERREZ_BITWISE_SHIFT_RIGHT;--Jeter Gutierrez: October, 4, 201
architecture design_SHIFT_RIGHT of GUTIERREZ_BITWISE_SHIFT_RIGHT is
begin--Jeter Gutierrez: October, 4, 2017|
RESULT<=to_stdlogicvector(to_bitvector(X)srl 1);--Jeter Gutierrez: O
end design_SHIFT_RIGHT;--Jeter Gutierrez: October, 4, 2017|
```

Figure 19: VHDL code for Shift Right.

11.2 Simulation for shift right.

In this simulation X will be given different values, in order to perform shift right, the result will be sent to RESULT then we will observe the simulation and consider its correctness.

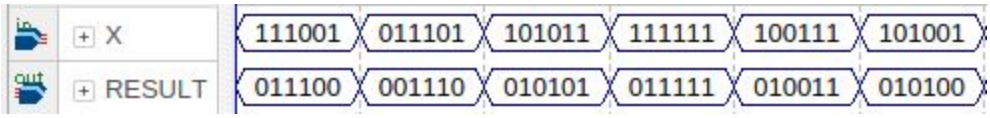

Figure 20: Vector waveform simulation for shift right. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design a bitwise shift right component. For every instance in X where a bit is equal to 1 in a position to the right in the RESULT we get a value of 1, and for every instance in X where a bit is equal to 0 we get a value of 1 in RESULT a position to the

right, while the leftmost bit is set to 0 each time because we don't have any bits to shift in its position.

12. Rotate Left.

12.1 Functionality and specifications for Rotate left.

The purpose of this circuit is to take as an input one n- bit word, in this case we will be taking a 6 bit word as an input which will be stored in X. The result will be stored in RESULT. The result of this component is to perform the rotate left operation on each bit in the word. After all the bits are rotated to the left in a closed loop, that means that the most significant bit is stored in the least significant bit after the operation instead of setting the least significant bit to 0 by default like we do in shift left.

```
library ieee;--Jeter Gutierrez: October, 4, 2017|
use ieee.std_logic_1164.all;--Jeter Gutierrez: October, 4, 2017|
use work.GUTIERREZ_OPT_CODE_PACKAGE.all;--Jeter Gutierrez: October,
entity GUTIERREZ_BITWISE_ROTATE_LEFT is--Jeter Gutierrez: October, 4
port( X: in std_logic_vector(5 downto 0);--Jeter Gutierrez: October,
    RESULT: out std_logic_vector (5 downto 0));--Jeter Gutierrez:
end GUTIERREZ_BITWISE_ROTATE_LEFT;--Jeter Gutierrez: October, 4, 201
architecture design_ROTATE_LEFT of GUTIERREZ_BITWISE_ROTATE_LEFT is
begin--Jeter Gutierrez: October, 4, 2017|
RESULT<=to_stdlogicvector(to_bitvector(X)rol 1);--Jeter Gutierrez: O
end design_ROTATE_LEFT;--Jeter Gutierrez: October, 4, 2017|
```

Figure 21: VHDL code for rotate left.

12.2 Simulation for rotate left.

In this simulation X will be given different values, in order to perform rotate left, the result will be sent to RESULT then we will observe the simulation and consider its correctness.

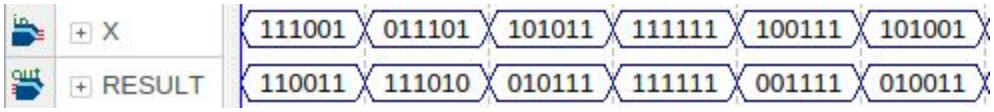

Figure 22: Vector waveform simulation for rotate left. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design a bitwise rotate left component. For every instance in X where a bit is equal to 1 in a position to the left in the RESULT we get a value of 1, and for every instance in X where a bit is equal to 0 we get a value of 1 in RESULT a position to the left, while the least significant bit becomes the value that was originally stored in the most significant bit instead of being set to 0 by default.

13. Rotate Right.

13.1 Functionality and specifications for Rotate Right.

The purpose of this circuit is to take as an input one n- bit word, in this case we will be taking a 6 bit word as an input which will be stored in X. The result will be stored in RESULT. The result of this component is to perform the rotate right operation on each bit in the word. After all the bits are rotated to the right in a closed loop, that means that the least significant bit is stored in the most significant bit after the operation instead of setting the most significant bit to 0 by default like we do in shift right.

```
library ieee;--Jeter Gutierrez: October, 4, 2017|
use ieee.std_logic_1164.all;--Jeter Gutierrez: October, 4, 2017|
use work.GUTIERREZ_OPT_CODE_PACKAGE.all;--Jeter Gutierrez: October, 4,
entity GUTIERREZ_BITWISE_ROTATE_RIGHT is--Jeter Gutierrez: October, 4,
port( X: in std_logic_vector(5 downto 0);--Jeter Gutierrez: October, 4
    RESULT: out std_logic_vector (5 downto 0));--Jeter Gutierrez: Oc
end GUTIERREZ_BITWISE_ROTATE_RIGHT;--Jeter Gutierrez: October, 4, 2017
architecture design_ROTATE_RIGHT of GUTIERREZ_BITWISE_ROTATE_RIGHT is
begin--Jeter Gutierrez: October, 4, 2017|
RESULT<=to_stdlogicvector(to_bitvector(X)ror 1);--Jeter Gutierrez: Oct
end design_ROTATE_RIGHT;--Jeter Gutierrez: October, 4, 2017|
```

Figure 23: VHDL code for Rotate Right.

13.2 Simulation for rotate right.

In this simulation X will be given different values, in order to perform rotate right, the result will be sent to RESULT then we will observe the simulation and consider its correctness.

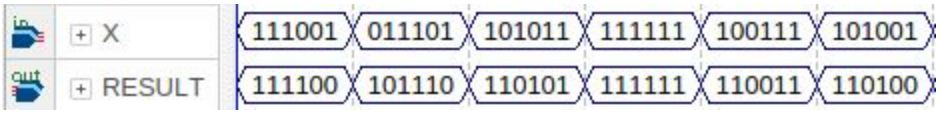

Figure 24: Vector waveform simulation for rotate right. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design a bitwise rotate right component. For every instance in X where a bit is equal to 1 in a position to the right in the RESULT we get a value of 1, and for every instance in X where a bit is equal to 0 we get a value of 1 in RESULT a position to the right, while the most significant bit becomes the value that was originally stored in the least significant bit instead of being set to 0 by default.

14. Integration project

14.1 Functionality and specifications for Integration Project.

The purpose of this component is to implement the above 13 bitwise operations into a single vhdl file that is capable of performing each of the 13 operations given a certain opcode. The inputs of this circuit are to have X and Y both of 6 bits as inputs and then to perform any of the following operations on either X or Y and store the result in RESULT. For the set less than operation the result will be sent to an led light that will be 1 if X is less than Y and 0 if otherwise. It will also need to perform addition and subtraction using my designed implementation of addition and subtraction, it will also perform addition and subtraction using LPM modules.

Bitwise and

Bitwise or

Bitwise xor

Bitwise not

Shift left

Shift right

Rotate left

Rotate right

Set less than

Addition --My design

Subtraction --My design

Addition --LPM

Subtraction --LPM

Are the operations that the opcode component will be able to perform. In order to make the design more efficient, instead of using any of the previous 13 designs as components we will be redefining their functions within this single vhdl file considering that they aren't super complicated tasks.

```
library ieee;--Jeter Gutierrez: October, 20, 2017|
use ieee.std_logic_1164.all;--Jeter Gutierrez: October, 20, 2017|
use ieee.numeric_std.all;--Jeter Gutierrez: October, 20, 2017|
use ieee.std_logic_unsigned.all;--Jeter Gutierrez: October, 20, 2017|
use IEEE.std_logic_arith.all;
use work.GUTIERREZ_OPT_CODE_PACKAGE.all;--Jeter Gutierrez: October, 20, 2017|
entity GUTIERREZ_OPCODE is --Jeter Gutierrez: October, 20, 2017|
port( CLOCK: in std_logic;--Jeter Gutierrez: October, 20, 2017|
                OPCODE: in std_logic_vector(3 downto 0);--Jeter Gutierrez: October, 20, 2017|
                X, Y: in std_logic_vector(5 downto 0);--Jeter Gutierrez: October, 20, 2017|
                X_IS_LESS_THAN_Y,NEGATIVE,Z,OVERFLOW_FINAL: out std_logic;
                RESULT: out std_logic_vector(5 downto 0));--Jeter Gutierrez: October, 20, 2017|
end GUTIERREZ_OPCODE;--Jeter Gutierrez: October, 20, 2017|
architecture DESIGN_OPCODE of GUTIERREZ_OPCODE is --Jeter Gutierrez: October, 20, 2017|
component GUTIERREZ_ADD_SUB_6_BITS is
port ( SUBTRACT: in std_logic;--Jeter Gutierrez September 24 ,2017
                        X, Y: in std_logic_vector(5 downto 0);--Jeter Gutierrez September 24 ,2017
                        S: out std_logic_vector (5 downto 0);--Jeter Gutierrez September 24 ,2017
                        Cout, Overflow: out std_logic);--Jeter Gutierrez September 24 ,2017
end component;
component GUTIERREZ_ADD_SUB_6_BITS_LPM is
port ( add_sub          : IN STD_LOGIC ;
                dataa           : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
                datab           : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
                overflow                : OUT STD_LOGIC ;
                result          : OUT STD_LOGIC_VECTOR (5 DOWNTO 0));
end component;

signal S_TEMP: std_logic_vector(5 downto 0);
signal Cout_temp,Overflow_temp,Cout_temp1,Overflow_temp1,Overflow_temp2,Overflow_temp3: std_logic;
signal RESULT_temp,RESULT_temp1,RESULT_temp2,RESULT_temp3:std_logic_vector (5 downto 0);
        begin--Jeter Gutierrez: October, 20, 2017|
        ADDITION:GUTIERREZ_ADD_SUB_6_BITS port map ('0',X,Y,RESULT_temp,Cout_temp,Overflow_temp);
        SUBTRACTION:GUTIERREZ_ADD_SUB_6_BITS port map ('1',X,Y,RESULT_temp1,Cout_temp1,Overflow_temp1);
        ADDITION_LPM:GUTIERREZ_ADD_SUB_6_BITS_LPM port map('1',X,Y,Overflow_temp2,RESULT_temp2);
        SUBTRACTION_LPM:GUTIERREZ_ADD_SUB_6_BITS_LPM port map('0',X,Y,Overflow_temp3,RESULT_temp3);
        process(CLOCK)--Jeter Gutierrez: October, 20, 2017|
                begin--Jeter Gutierrez: October, 20, 2017|
                        if(CLOCK = '1') then--Jeter Gutierrez: October, 20, 2017|
                                case OPCODE is --Jeter Gutierrez: October, 20, 2017|
                                when "0000" =>--Jeter Gutierrez: October, 20, 2017|
                                        RESULT <= X and Y;--Jeter Gutierrez: October, 20, 2017|
                                when "0001" =>--Jeter Gutierrez: October, 20, 2017|
                                        RESULT <= X or Y;--Jeter Gutierrez: October, 20, 2017|
                                when "0010" =>--Jeter Gutierrez: October, 20, 2017|
                                        RESULT <= X xor Y;--Jeter Gutierrez: October, 20, 2017|
                                when "0011" =>--Jeter Gutierrez: October, 20, 2017|
                                        RESULT <= not X;--Jeter Gutierrez: October, 20, 2017|
                                when "0100" =>--Jeter Gutierrez: October, 20, 2017|
                                        RESULT <= to_stdlogicvector(to_bitvector(X)sll 1);--Jeter Gutierrez: Oc
                                when "0101" =>--Jeter Gutierrez: October, 20, 2017|
                                        RESULT <= to_stdlogicvector(to_bitvector(X)srl 1);--Jeter Gutierrez: Oc
                                when "0110" =>--Jeter Gutierrez: October, 20, 2017|
                                        RESULT <= to_stdlogicvector(to_bitvector(X)rol 1);--Jeter Gutierrez: Oc
                                when "0111" =>--Jeter Gutierrez: October, 20, 2017|
                                        RESULT <= to_stdlogicvector(to_bitvector(X)ror 1);--Jeter Gutierrez: Oc
                                when "1000"=> IF (X<Y) THEN X_IS_LESS_THAN_Y<='1'; END IF;
                                                                IF (X>Y) THEN X_IS_LESS_THAN_Y<='0'; END IF;
```

```
when "1001"=> RESULT<= RESULT_temp;
                                NEGATIVE<=RESULT_temp(5);
                                Z<=((RESULT_temp(0) nor RESULT_temp(1)) and (RESULT_temp(2) nor RESULT_temp(3))and(RESULT_temp(4) nor RESULT_temp(5)));
                                OVERFLOW_FINAL<=Overflow_temp;
when "1010"=> RESULT<=RESULT_temp1;
                                NEGATIVE<=RESULT_temp1(5);
                                Z<=((RESULT_temp1(0) nor RESULT_temp1(1)) and (RESULT_temp1(2) nor RESULT_temp1(3))and(RESULT_temp1(4) nor RESULT_temp1(5)));
                                OVERFLOW_FINAL<=Overflow_temp1;
when "1011"=> RESULT<=RESULT_temp2;
                                NEGATIVE<=RESULT_temp2(5);
                                Z<=((RESULT_temp2(0) nor RESULT_temp2(1)) and (RESULT_temp2(2) nor RESULT_temp2(3))and(RESULT_temp2(4) nor RESULT_temp2(5)));
                                OVERFLOW_FINAL<=Overflow_temp2;
WHEN "1100"=> RESULT<=RESULT_temp3;
                                NEGATIVE<=RESULT_temp3(5);
                                Z<=((RESULT_temp3(0) nor RESULT_temp3(1)) and (RESULT_temp(2) nor RESULT_temp(3))and(RESULT_temp(4) nor RESULT_temp(5)));
                                OVERFLOW_FINAL<=Overflow_temp3;
when others =>--Jeter Gutierrez: October, 20, 2017|
NULL;--Jeter Gutierrez: October, 20, 2017|
end case;--Jeter Gutierrez: October, 20, 2017|
```

```
                end process;--Jeter Gutierrez: October, 20, 2017|
end DESIGN_OPCODE;--Jeter Gutierrez: October, 20, 2017|
```

Figure 25: VHDL code for Integration project Opcode component.

14.2 Simulation for Integration Project.

In this simulation we will be giving the clock, opcode, X and Y different values in order to perform the different tasks that we have implemented into our opcode component. The result after each operation will be sent to RESULT then we will observe the simulation and consider its correctness.

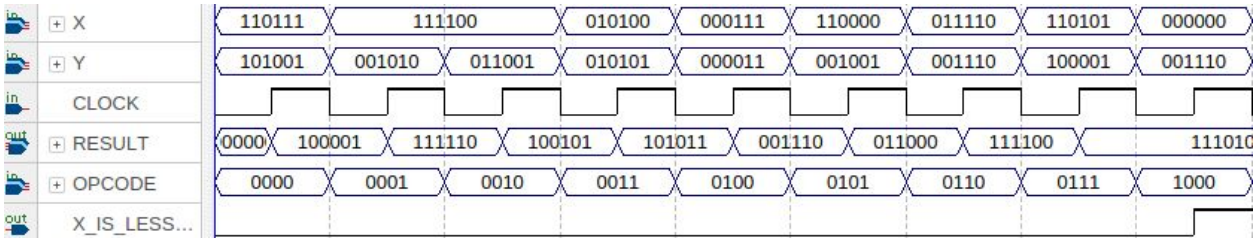

Figure 26: Vector waveform simulation for opcode component. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design an opcode component that performs 8 operations. When there was a rising edge signal the component would take into account what the opcode is and perform the given operations, the binary operands are performed on both X and Y while the unary operations were performed only on X. We were able to correctly design the opcode component and did everything correct in VHDL and even in simulation. In the end we can see that our set less than is also working correctly as it was implemented properly and can detect that 0 is less than 38 which is correct. That signal is sent to X_IS_LESS_THAN_Y.

15. OPCODE TEST

15.1 Functionality and specifications for OPCODE TEST

The purpose of this circuit is to create a test bench in order to simulate our opcode component. We already simulated our opcode component in Vector waveform but it is more professional and exact to use a test bench to test the correctness of our circuits because we have more control over the simulation this way. The test bench imitates a physical lab bench making our simulation cleaner and more organized. We also are capable of testing every possible combination using a test bench code instead of just using vector waveform simulation.

```
library ieee;--Jeter Gutierrez: October, 4, 2017
use ieee.std_logic_1164.all;--Jeter Gutierrez: October, 4, 2017
use ieee.numeric_std.all;--Jeter Gutierrez: October, 4, 2017
use ieee.std_logic_unsigned.all;--Jeter Gutierrez: October, 4, 2017
use work.GUTIERREZ_OPT_CODE_PACKAGE.all;--Jeter Gutierrez: October, 4, 2017
entity GUTIERREZ_TEST_OPCODE is--Jeter Gutierrez: October, 4, 2017
end GUTIERREZ_TEST_OPCODE;--Jeter Gutierrez: October, 4, 2017
architecture arch_test_opcode of GUTIERREZ_TEST_OPCODE is--Jeter Gutierrez: October, 4, 2017
component GUTIERREZ_OPCODE is --Jeter Gutierrez: October, 4, 2017
port( CLOCK: in std_logic;--Jeter Gutierrez: October, 4, 2017
        OPCODE: in std_logic_vector(3 downto 0);--Jeter Gutierrez: October, 4, 2017
        X, Y: in std_logic_vector(5 downto 0);--Jeter Gutierrez: October, 4, 2017
        RESULT: out std_logic_vector(5 downto 0));--Jeter Gutierrez: October, 4, 2017
end component;--Jeter Gutierrez: October, 4, 2017
signal X_PRIME,Y_PRIME,RESULT_PRIME :STD_LOGIC_VECTOR(5 DOWNTO 0);--Jeter Gutierrez: October, 4, 2017
signal CLOCK_PRIME: std_logic := '0';--Jeter Gutierrez: October, 4, 2017
signal OPCODE_PRIME: std_logic_vector(3 downto 0);--Jeter Gutierrez: October, 4, 2017
begin--Jeter Gutierrez: October, 4, 2017
uut: GUTIERREZ_OPCODE port map (--Jeter Gutierrez: October, 4, 2017
CLOCK=> CLOCK_PRIME,--Jeter Gutierrez: October, 4, 2017
OPCODE=> OPCODE_PRIME,--Jeter Gutierrez: October, 4, 2017
X => X_PRIME,--Jeter Gutierrez: October, 4, 2017
Y => Y_PRIME,--Jeter Gutierrez: October, 4, 2017
RESULT => RESULT_PRIME--Jeter Gutierrez: October, 4, 2017
);--Jeter Gutierrez: October, 4, 2017
tb : process--Jeter Gutierrez: October, 4, 2017
begin--Jeter Gutierrez: October, 4, 2017
wait for 5 ns;--Jeter Gutierrez: October, 4, 2017
report "TESTING OPCODE";--Jeter Gutierrez: October, 4, 2017
X_PRIME<="000000";--Jeter Gutierrez: October, 4, 2017
Y_PRIME<="000000";--Jeter Gutierrez: October, 4, 2017
OPCODE_PRIME<="0000";--Jeter Gutierrez: October, 4, 2017
for XS in 0 to 2710 loop--Jeter Gutierrez: October, 4, 2017
for YS in 0 to 2710 loop--Jeter Gutierrez: October, 4, 2017
for OPCODES in 0 to 16 loop--Jeter Gutierrez: October, 4, 2017
wait for 5 ns;--Jeter Gutierrez: October, 4, 2017
CLOCK_PRIME<='1';--Jeter Gutierrez: October, 4, 2017
wait for 5 ns;--Jeter Gutierrez: October, 4, 2017
CLOCK_PRIME<='0';--Jeter Gutierrez: October, 4, 2017
if (OPCODE_PRIME="0000") then--Jeter Gutierrez: October, 4, 2017
assert( RESULT_PRIME =(X_PRIME and Y_PRIME)) report "0000 ERROR IN AND" severity ERROR;--Jeter Gutierrez: October, 4, 20
end if;--Jeter Gutierrez: October, 4, 2017
if (OPCODE_PRIME="0001") then--Jeter Gutierrez: October, 4, 2017
assert(RESULT_PRIME =(X_PRIME OR Y_PRIME)) report "0001 ERROR IN OR" severity ERROR;--Jeter Gutierrez: October, 4, 2017
end if;--Jeter Gutierrez: October, 4, 2017
if (OPCODE_PRIME="0010") then--Jeter Gutierrez: October, 4, 2017
assert(RESULT_PRIME =(X_PRIME XOR Y_PRIME)) report "0010 ERROR IN XOR" severity ERROR;--Jeter Gutierrez: October, 4, 201
end if;--Jeter Gutierrez: October, 4, 2017
if (OPCODE_PRIME="0011") then--Jeter Gutierrez: October, 4, 2017
assert(RESULT_PRIME =(NOT X_PRIME)) report "0011 ERROR IN NOT" severity ERROR;--Jeter Gutierrez: October, 4, 2017
end if;--Jeter Gutierrez: October, 4, 2017
if (OPCODE_PRIME="0100") then--Jeter Gutierrez: October, 4, 2017
assert(RESULT_PRIME = to_stdlogicvector(to_bitvector(X_PRIME)sll 1)) report "0100 ERROR IN SHIFT LEFT" severity ERROR;--
end if;--Jeter Gutierrez: October, 4, 2017
if (OPCODE_PRIME="0101") then--Jeter Gutierrez: October, 4, 2017
assert(RESULT_PRIME = to_stdlogicvector(to_bitvector(X_PRIME)srl 1)) report "0101 ERROR IN SHIFT RIGHT" severity ERROR;-
end if;--Jeter Gutierrez: October, 4, 2017
if (OPCODE_PRIME="0110") then--Jeter Gutierrez: October, 4, 2017
assert(RESULT_PRIME = to_stdlogicvector(to_bitvector(X_PRIME)rol 1)) report "0110 ERROR IN ROTATE LEFT" severity ERROR;-
end if;--Jeter Gutierrez: October, 4, 2017
if (OPCODE_PRIME="0111") then--Jeter Gutierrez: October, 4, 2017
assert(RESULT_PRIME = to_stdlogicvector(to_bitvector(X_PRIME)ror 1)) report "0111 ERROR IN ROTATE RIGHT" severity ERROR;
end if;--Jeter Gutierrez: October, 4, 2017
X_PRIME<=X_PRIME+"000001";--Jeter Gutierrez: October, 4, 2017
end loop;--Jeter Gutierrez: October, 4, 2017
Y_PRIME<=Y_PRIME+"000001";--Jeter Gutierrez: October, 4, 2017
end loop;--Jeter Gutierrez: October, 4, 2017
wait for 5 ns;--Jeter Gutierrez: October, 4, 2017
OPCODE_PRIME<=OPCODE_PRIME+"0001";--Jeter Gutierrez: October, 4, 2017
end loop;--Jeter Gutierrez: October, 4, 2017
report "Test completed";--Jeter Gutierrez: October, 4, 2017
wait; -- will wait for ever--Jeter Gutierrez: October, 4, 2017
end process;--Jeter Gutierrez: October, 4, 2017
end arch_test_opcode;--Jeter Gutierrez: October, 4, 2017
```

Figure 27: VHDL code for the test bench for testing opcode component. In this design modelsim will be used in order to simulate what is written in the testing code. It will be used to test the

values for 2 6 bit words by increasing each one by 1 bit and determine whether for every possible case of performing each of the 9 operations on 2 6 bit words after a certain period of time or after testing another state it will continue to be correct. The reason we do this is to have more control over our testing for correctness of our opcode component. Including the set less than.

15.2 Simulation for opcode component test.

In this simulation we will be using modelsim to run our test bench file for the opcode component, if we get no errors we know we have designed a opcode component correctly. The difference in this state is that we already wrote what values we want to test for and modelsim will create those values for us instead of us having to use the cursor to select the values using waveform. Using a test bench is more efficient than manually inputting values to test, this way we test every possible value.

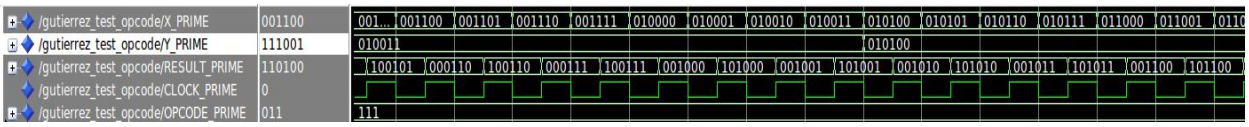

Figure 28: Vector waveform simulation for for test opcode component. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design an opcode component. The test bench here was used to test every possible combination of 6-Bit words and every combination of the 3 bit opcode operations as well.

16 Demonstration of Integration Project on DE2-115 Board.

Final design combined with 7-segment hexadecimal format and the integration project is as follows.

```
library ieee;--Jeter Gutierrez: October, 4, 2017|
use ieee.std_logic_1164.all;--Jeter Gutierrez: October, 4, 2017|
use ieee.numeric_std.all;--Jeter Gutierrez: October, 4, 2017|
use ieee.std_logic_unsigned.all;--Jeter Gutierrez: October, 4, 2017|
use IEEE.std_logic_arith.all;
use work.GUTIERREZ_OPT_CODE_PACKAGE.all;--Jeter Gutierrez: October, 4, 2017|
entity GUTIERREZ_INTEGRATION_PROJECT is --Jeter Gutierrez: October, 4, 2017|
port( CLOCK: in std_logic;--Jeter Gutierrez: October, 4, 2017|
      OPCODE: in std_logic_vector(3 downto 0);--Jeter Gutierrez: October, 4, 2017|
      X, Y: in std_logic_vector(5 downto 0);--Jeter Gutierrez: October, 4, 2017|
      X_IS_LESS_THAN_Y,NEGATIVE,Z,OVERFLOW: out std_logic;
      RESULT: out std_logic_vector(5 downto 0);--Jeter Gutierrez: October, 4, 2017|
      DISPLAY:out std_logic_vector(13 downto 0));
end GUTIERREZ_INTEGRATION_PROJECT;--Jeter Gutierrez: October, 4, 2017|
architecture DESIGN_INTEGRATION of GUTIERREZ_INTEGRATION_PROJECT is --Jeter Gutierrez: October, 4, 2017|
component GUTIERREZ_OPCODE is --Jeter Gutierrez: October, 4, 2017|
port( CLOCK: in std_logic;--Jeter Gutierrez: October, 4, 2017|
      OPCODE: in std_logic_vector(3 downto 0);--Jeter Gutierrez: October, 4, 2017|
      X, Y: in std_logic_vector(5 downto 0);--Jeter Gutierrez: October, 4, 2017|
      X_IS_LESS_THAN_Y,NEGATIVE,Z,OVERFLOW_FINAL: out std_logic;
      RESULT: out std_logic_vector(5 downto 0));--Jeter Gutierrez: October, 4, 2017|
end component;--Jeter Gutierrez: October, 4, 2017|
component GUTIERREZ_DEC_TO_HEX is
port(hex_digit: in std_logic_vector(3 downto 0);
a,b,c,d,e,f,g : out std_logic);
end component;
signal WORD_A: std_logic_vector (3 downto 0);
signal WORD_B: std_logic_vector (3 downto 0);
signal RESULT_temp:std_logic_vector (5 downto 0);
signal NEGATIVE_TEMP:std_logic;
begin
OPTCODE_RUNNING:GUTIERREZ_OPCODE port map(CLOCK,OPCODE,X,Y,X_IS_LESS_THAN_Y,NEGATIVE_TEMP,Z,OVERFLOW,RESULT=>RESULT_temp);
WORD_A(0)<=RESULT_temp(0);
WORD_A(1)<=RESULT_temp(1);
WORD_A(2)<=RESULT_temp(2);
WORD_A(3)<=RESULT_temp(3);
WORD_B(0)<=RESULT_temp(4);
WORD_B(1)<=RESULT_temp(5);
WORD_B(2)<='0';
WORD_B(3)<='0';
HEX_1:GUTIERREZ_DEC_TO_HEX port map(WORD_A,DISPLAY(0),DISPLAY(1),DISPLAY(2),DISPLAY(3),DISPLAY(4),DISPLAY(5),DISPLAY(6));
HEX_2:GUTIERREZ_DEC_TO_HEX port map(WORD_B,DISPLAY(7),DISPLAY(8),DISPLAY(9),DISPLAY(10),DISPLAY(11),DISPLAY(12),DISPLAY(13));
RESULT<=RESULT_temp;
NEGATIVE<=NOT NEGATIVE_TEMP;
end DESIGN_INTEGRATION;--Jeter Gutierrez: October, 4, 2017|
```

Figure 29: VHDL code for final product of combined design, this connects the opcode component with the display in order to program the fpga board correctly.

The inputs and outputs assigned to the DE2-115 board are:

CLOCK is assigned to PIN_Y23

OPCODE[0] is assigned to PIN_AA24

OPCODE[1] is assigned to PIN_AA23

OPCODE[2] is assigned to PIN_AA22

OPCODE[3] is assigned to PIN_Y24

X[0] is assigned to PIN_AB28

X[1] is assigned to PIN_AC28

X[2] is assigned to PIN_AC27

X[3] is assigned to PIN_AD27

X[4] is assigned to PIN_AB27

X[5] is assigned to PIN_AC26

Y[0] is assigned to PIN_AD26

Y[1] is assigned to PIN_AB26

Y[2] is assigned to PIN_AC25

Y[3] is assigned to PIN_AB25

Y[4] is assigned to PIN_AC24

Y[5] is assigned to PIN_AB24

X_IS_LESS_THAN_Y is assigned to PIN_E21

NEGATIVE is assigned to PIN_W28

Z is assigned to PIN_E24

OVERFLOW is assigned to PIN_G22

RESULT[0] is assigned to PIN_G19

RESULT[1] is assigned to PIN_F19

RESULT[2] is assigned to PIN_E19

RESULT[3] is assigned to PIN_F21

RESULT[4] is assigned to PIN_F18

RESULT[5] is assigned to PIN_E18

DISPLAY[0] is assigned to PIN_G18

DISPLAY[1] is assigned to PIN_F22

DISPLAY[2] is assigned to PIN_E17

DISPLAY[3] is assigned to PIN_L26

DISPLAY[4] is assigned to PIN_L25

DISPLAY[5] is assigned to PIN_J22

DISPLAY[6] is assigned to PIN_H22

DISPLAY[7] is assigned to PIN_M24

DISPLAY[8] is assigned to PIN_Y22

DISPLAY[9] is assigned to PIN_W21

DISPLAY[10] is assigned to PIN_W22

DISPLAY[11] is assigned to PIN_W25

DISPLAY[12] is assigned to PIN_U23

DISPLAY[13] is assigned to PIN_U24

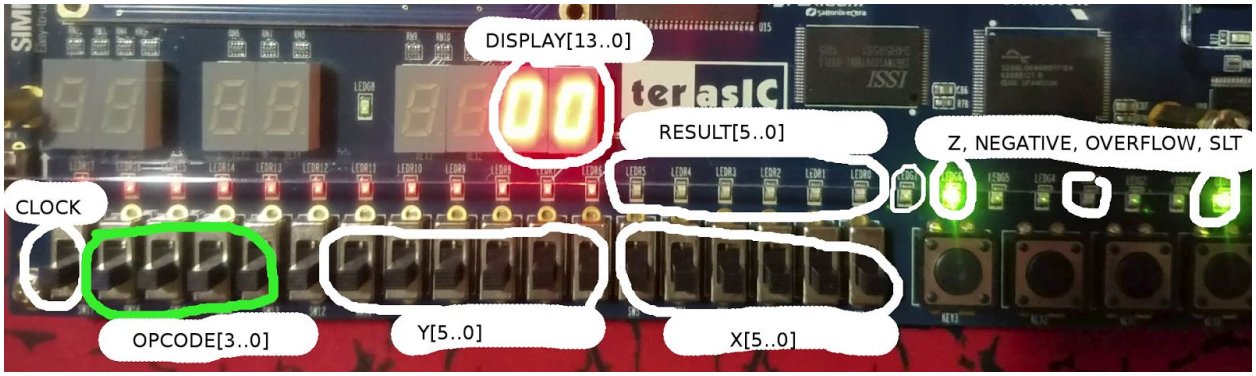

Figure 30: Digital Circuit of Integration Project. We have set switches for the X 6 bit word and the Y 6 bit word as well as for the 4 bit opcode, the clock signal and led lights for the 6 bit result. The 6 bit result is also being sent to the 7-segment hexadecimal display.

17. Conclusion.

As it turns out designing these circuits wasn't hard. It was however very useful to design a component that uses opcode, it saves time to be capable of performing various different operations all using one circuit and 7-segment displays to display our results. It saves time and it also is impressive and very useful, that is why processors like intel, amd, cortex and mips etc. are capable of performing such tasks. If we were to implement multiplication for example it could be seen as difficult but it also isn't difficult, it is just shifted addition using something like a linear shift register after each bit which is very useful. Higher level languages like Java, C++ and Ruby have similar operations like << or >> for shifting or | for performing or operations, they are very useful as well and very important to performing higher level tasks. Shifting right or left can be used to perform multiplications on n-bit words by simply using an adder and a shifter in order to add and multiply or divide accordingly. Overall designing this lab was easy after I realized I could simply implement all the components into the same file. What I learned is that VHDL is actually a smart language and that using opcode is very effective and efficient on performing higher level operations. The set less than is also a useful bit of information that determines which n-bit word is greater or less than between two or more n-bit words. We were able to add addition and subtraction to the project as well which was very useful to show that we can use this design to perform any operation that we want to perform without hesitation.

18. Appendix.

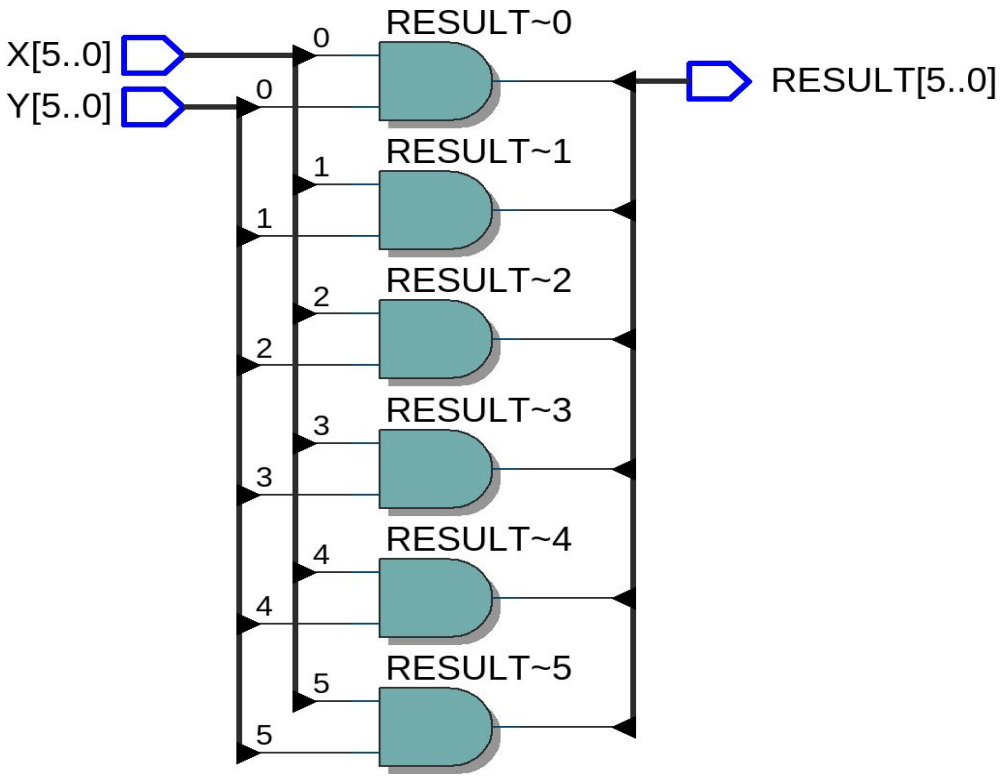

Figure 31: Block Diagram for bitwise and.

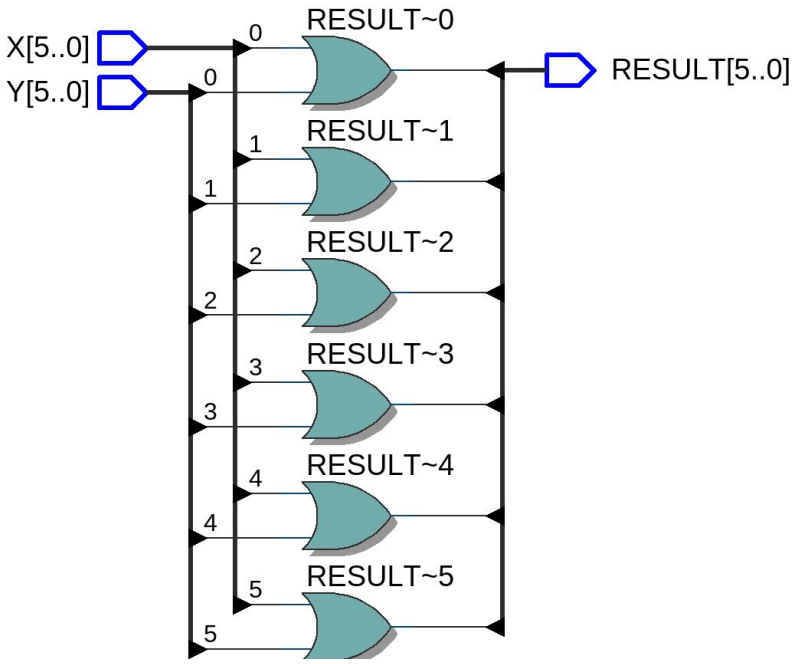

Figure 32: Block diagram for bitwise or.

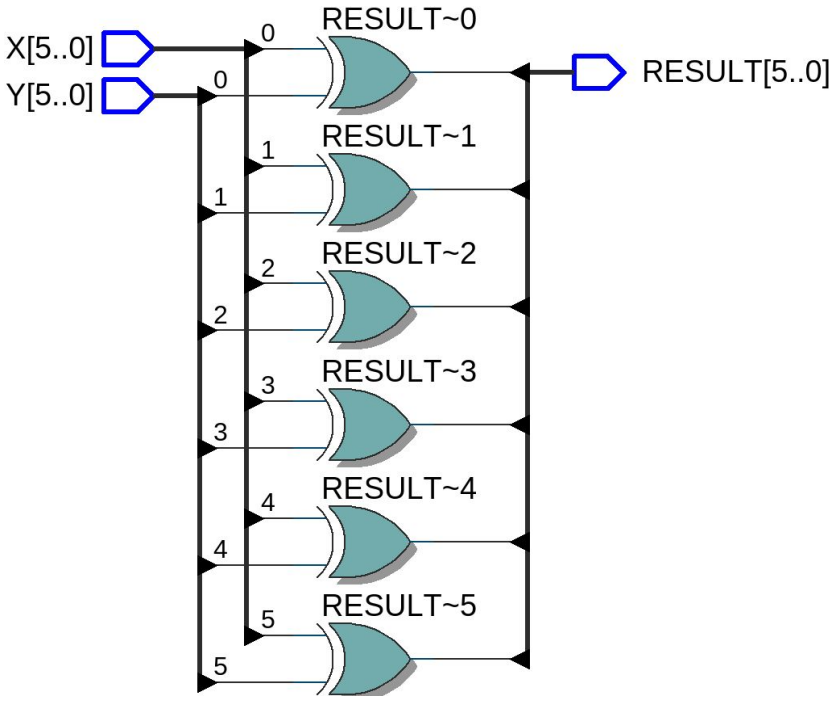

Figure 33: Block diagram for bitwise xor.

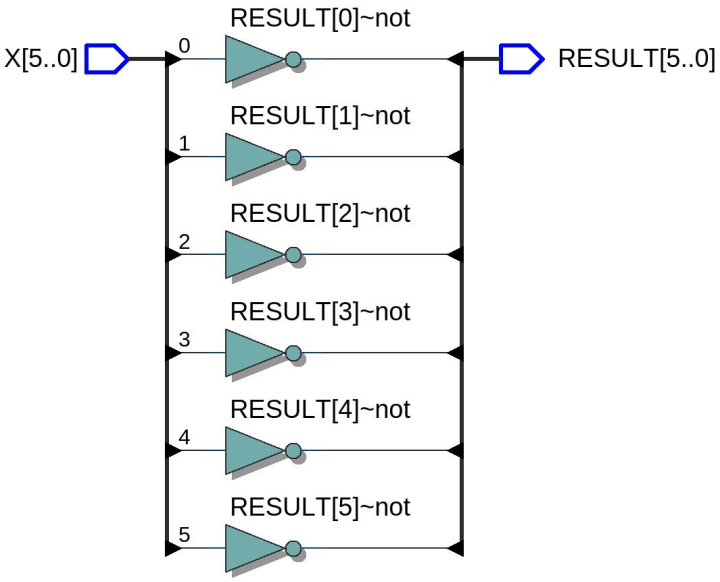

Figure 34: Block diagram for bitwise not.

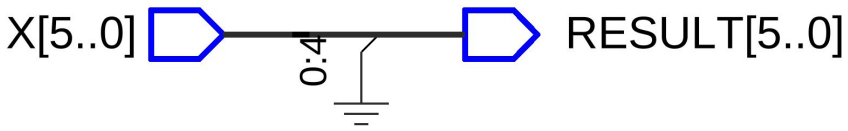

Figure 35: Block diagram for shift left.

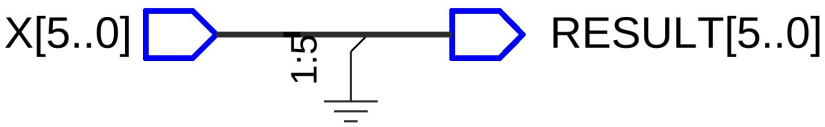

Figure 36: Block diagram for shift right.

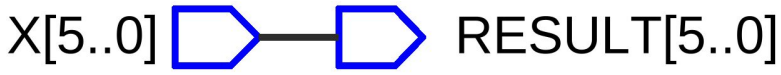

Figure 37: Block diagram for rotate left.

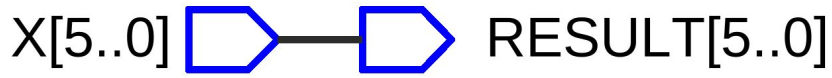

Figure 38: Block diagram for rotate right.

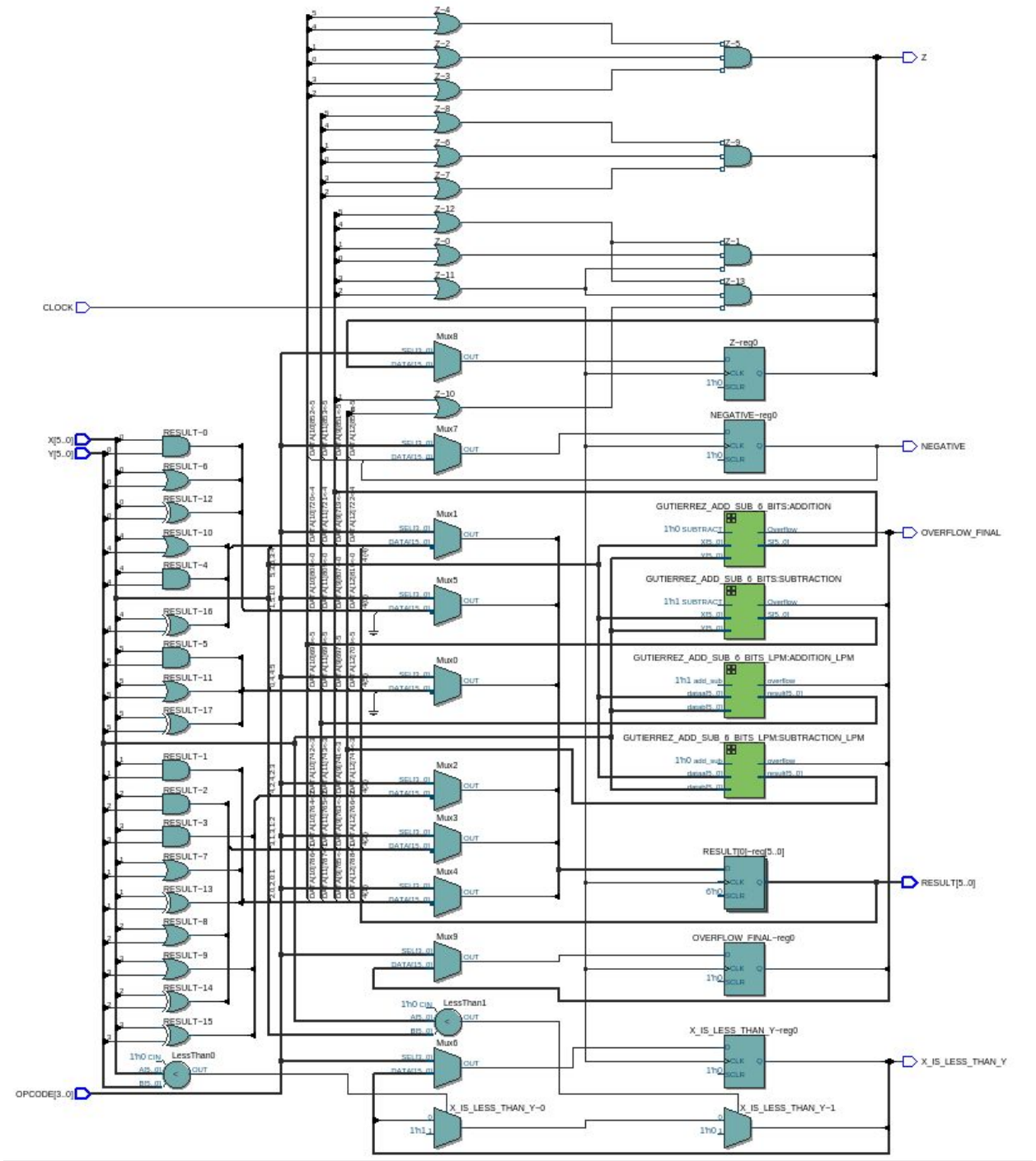

Figure 39: Block diagram for opcode component.

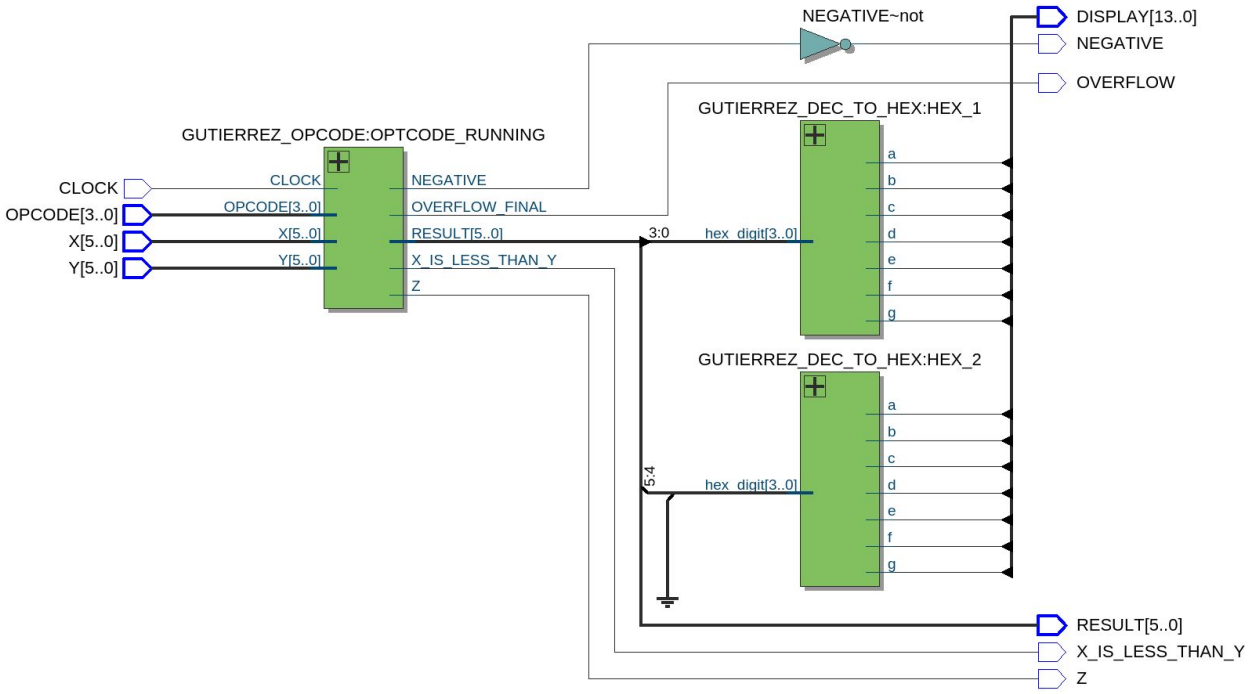

Figure 40: Block diagram for Integrated project component.

```
To, Location
CLOCK,PIN_Y23
OPCODE[0],PIN_AA24
OPCODE[1],PIN_AA23
OPCODE[2],PIN_AA22
OPCODE[3],PIN_Y24
X[0],PIN_AB28
X[1],PIN_AC28
X[2],PIN_AC27
X[3],PIN_AD27
X[4],PIN_AB27
X[5],PIN_AC26
Y[0],PIN_AD26
Y[1],PIN_AB26
Y[2],PIN_AC25
Y[3],PIN_AB25
Y[4],PIN_AC24
Y[5],PIN_AB24
X_IS_LESS_THAN_Y,PIN_E21
NEGATIVE,PIN_W28
Z,PIN_E24
OVERFLOW,PIN_G22
RESULT[0],PIN_G19
RESULT[1],PIN_F19
RESULT[2],PIN_E19
RESULT[3],PIN_F21
RESULT[4],PIN_F18
RESULT[5],PIN_E18
DISPLAY[0],PIN_G18
DISPLAY[1],PIN_F22
DISPLAY[2],PIN_E17
DISPLAY[3],PIN_L26
DISPLAY[4],PIN_L25
DISPLAY[5],PIN_J22
DISPLAY[6],PIN_H22
DISPLAY[7],PIN_M24
DISPLAY[8],PIN_Y22
DISPLAY[9],PIN_W21
DISPLAY[10],PIN_W22
DISPLAY[11],PIN_W25
DISPLAY[12],PIN_U23
DISPLAY[13],PIN_U24
```

Figure 40: Pin assignments for Integration project on DE2-115 board.