Lab 4 BITWISE SHIFT ROTATION

CSC 343 Fall 2017

October 4, 2017

Jeter Gutierrez

TABLE OF CONTENTS

1. Objective PG 3-4

2. Bitwise and.

2.1 Functionality and specifications for bitwise and. PG 4-5

2.2 Simulation for bitwise and. PG 5

3. Bitwise Or.

3.1 Functionality and specifications for bitwise or. PG 5-6

3.2 Simulation for bitwise or. PG 6-7

4. Bitwise XOR.

4.1 Functionality and specifications for bitwise XOR. PG 7

4.2 Simulation for bitwise XOR. PG 7-8

5. Bitwise NOT.

5.1 Functionality and specifications for bitwise NOT. PG 8

5.2 Simulation for bitwise NOT. PG 8-9

6. Shift Left.

6.1 Functionality and specifications for shift left. PG 9

6.2 Simulation for shift left. PG 9-10

7. Shift Right.

7.1 Functionality and specifications for shift right. PG 10

7.2 Simulation for shift right. PG 10-11

8. Rotate Left.

8.1 Functionality and specifications for Rotate left. PG 11

8.2 Simulation for rotate left. PG 11-12

9. Rotate Right.

9.1 Functionality and specifications for Rotate Right. PG 12-13

9.2 Simulation for rotate right. PG 13

10. Opcode component

10.1 Functionality and specifications for Opcode component. PG 13-14

10.2 Simulation for opcode component. PG 15-16

11. OPCODE TEST

11.1 Functionality and specifications for OPCODE TEST PG 16-18

11.2 Simulation for opcode component test. PG 18

12 Demonstration of opcode component on DE2-115 Board. PG 18-20

13. Conclusion. PG 20-21

14. Appendix. PG 21-25

1. Objective

In this lab we will be implementing several bit wise operations using VHDL. The purpose of this lab is for us to gain an understanding in how bitwise operations work and how opt code works as well. 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. 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. Bitwise and
2. Bitwise or
3. Bitwise xor
4. Bitwise not
5. Shift left
6. Shift right
7. Rotate left
8. Rotate right
9. Set less than.
10. Opcode component.

2. Bitwise and.

2.1 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 1: Vhdl Code for bitwise and.

2.2 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.

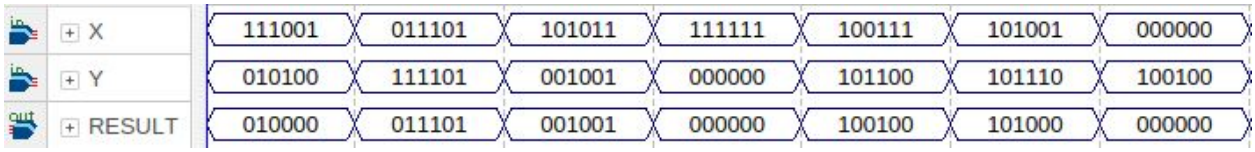

Figure 2: Vector waveform simulation for bitwise and. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design a bitwise and component. For every position in an instance of X and Y where they share a value of 1 it is also 1 in the result which means that it is correct.

3. Bitwise Or.

3.1 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 3: VHDL code for bitwise or.

3.2 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.

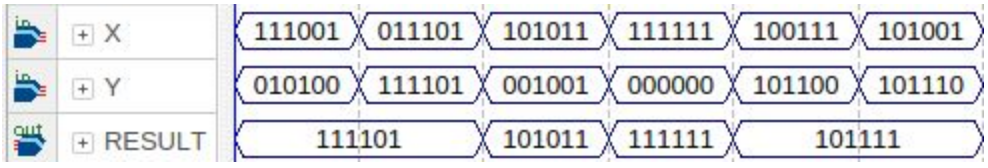

Figure 4: Vector waveform simulation for bitwise or. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design a bitwise or component. For every position in an

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.

4. Bitwise XOR.

4.1 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 5: VHDL code for Bitwise XOR.

4.2 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.

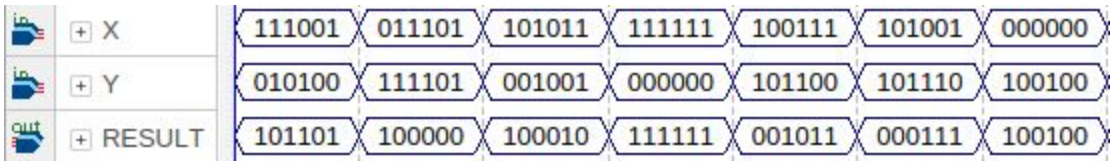

Figure 6: Vector waveform simulation for bitwise xor. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any

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.

5. Bitwise NOT.

5.1 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 7: VHDL code for bitwise not.

5.2 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.

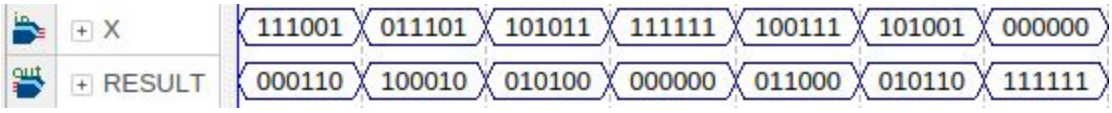

Figure 8: Vector waveform simulation for bitwise not. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any

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.

6. Shift Left.

6.1 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 9: VHDL code for shift left.

6.2 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.

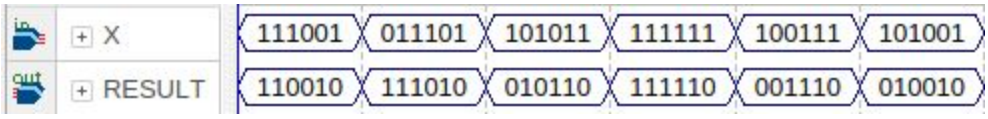

Figure 10: Vector waveform simulation for shift left. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any

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.

7. Shift Right.

7.1 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, 2017
architecture design_SHIFT_RIGHT of GUTIERREZ_BITWISE_SHIFT_RIGHT is
begin--Jeter Gutierrez: October, 4, 2017|
RESULT<=to_stdlogicvector(to_bitvector(X)sr|l 1);--Jeter Gutierrez: O
end design_SHIFT_RIGHT;--Jeter Gutierrez: October, 4, 2017|
```

Figure 11: VHDL code for Shift Right.

7.2 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.

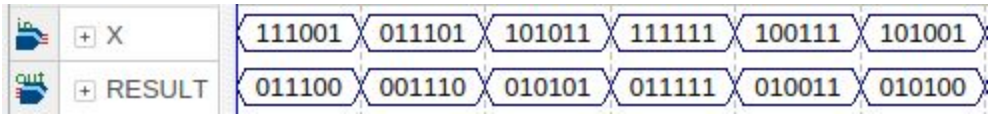

Figure 12: Vector waveform simulation for shift right. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design a bitwise shift right component. For every instance in X where a bit is 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.

8. Rotate Left.

8.1 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 13: VHDL code for rotate left.

8.2 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.

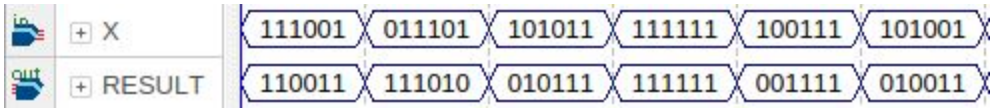

Figure 14: Vector waveform simulation for rotate left. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design a bitwise rotate left component. For every instance in X where a bit is 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.

9. Rotate Right.

9.1 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 15: VHDL code for Rotate Right.

9.2 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.

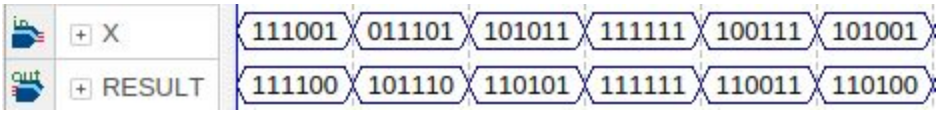

Figure 16: Vector waveform simulation for rotate right. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design a bitwise rotate right component. For every instance in X where a bit 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.

10. Opcode component

10.1 Functionality and specifications for Opcode component.

The purpose of this component is to implement the above 9 bitwise operations into a single vhdl file that is capable of performing each of the 9 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.

Bitwise and

Bitwise or

Bitwise xor

Bitwise not

Shift left

Shift right

Rotate left

Rotate right

Set less than

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 8 designs as components we will be redefining their functions within this single vhdl file considering that they aren't super complicated tasks.

```
1  library ieee;--Jeter Gutierrez: October, 4, 2017|
2  use ieee.std_logic_1164.all;--Jeter Gutierrez: October, 4, 2017|
3  use ieee.numeric_std.all;--Jeter Gutierrez: October, 4, 2017|
4  use ieee.std_logic_unsigned.all;--Jeter Gutierrez: October, 4, 2017|
5  use IEEE.std_logic_arith.all;
6  use work.GUTIERREZ_OPT_CODE_PACKAGE.all;--Jeter Gutierrez: October, 4, 2
7  entity GUTIERREZ_OPCODE is --Jeter Gutierrez: October, 4, 2017|
8  port( CLOCK: in std_logic;--Jeter Gutierrez: October, 4, 2017|
9        OPCODE: in std_logic_vector(3 downto 0);--Jeter Gutierrez: October
10       X, Y: in std_logic_vector(5 downto 0);--Jeter Gutierrez: October,
11       X_IS_LESS_THAN_Y: out std_logic;
12       RESULT: out std_logic_vector(5 downto 0));--Jeter Gutierrez: Octob
13 end GUTIERREZ_OPCODE;--Jeter Gutierrez: October, 4, 2017|
14 architecture DESIGN_OPCODE of GUTIERREZ_OPCODE is --Jeter Gutierrez: Oct
15    begin--Jeter Gutierrez: October, 4, 2017|
16    process(CLOCK)--Jeter Gutierrez: October, 4, 2017|
17       begin--Jeter Gutierrez: October, 4, 2017|
18          if(CLOCK = '1') then--Jeter Gutierrez: October, 4, 2017|
19             case OPCODE is --Jeter Gutierrez: October, 4, 2017|
20             when "0000" =>--Jeter Gutierrez: October, 4, 2017|
21                RESULT <= X and Y;--Jeter Gutierrez: October, 4, 2017|
22             when "0001" =>--Jeter Gutierrez: October, 4, 2017|
23                RESULT <= X or Y;--Jeter Gutierrez: October, 4, 2017|
24             when "0010" =>--Jeter Gutierrez: October, 4, 2017|
25                RESULT <= X xor Y;--Jeter Gutierrez: October, 4, 2017|
26             when "0011" =>--Jeter Gutierrez: October, 4, 2017|
27                RESULT <= not X;--Jeter Gutierrez: October, 4, 2017|
28             when "0100" =>--Jeter Gutierrez: October, 4, 2017|
29                RESULT <= to_stdlogicvector(to_bitvector(X)sll 1);--Jeter
30             when "0101" =>--Jeter Gutierrez: October, 4, 2017|
31                RESULT <= to_stdlogicvector(to_bitvector(X)srl 1);--Jeter
32             when "0110" =>--Jeter Gutierrez: October, 4, 2017|
33                RESULT <= to_stdlogicvector(to_bitvector(X)rol 1);--Jeter
34             when "0111" =>--Jeter Gutierrez: October, 4, 2017|
35                RESULT <= to_stdlogicvector(to_bitvector(X)ror 1);--Jeter
36             when "1000"=> IF (X<Y) THEN X_IS_LESS_THAN_Y<='1'; END IF;
37                           IF (X>Y) THEN X_IS_LESS_THAN_Y<='0'; END IF;
38             when others =>--Jeter Gutierrez: October, 4, 2017|
39             NULL;--Jeter Gutierrez: October, 4, 2017|
40             end case;--Jeter Gutierrez: October, 4, 2017|
41          end if;--Jeter Gutierrez: October, 4, 2017|
42       end process;--Jeter Gutierrez: October, 4, 2017|
43 end DESIGN_OPCODE;--Jeter Gutierrez: October, 4, 2017|
```

Figure 17: VHDL code for opcode component.

10.2 Simulation for opcode component.

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.

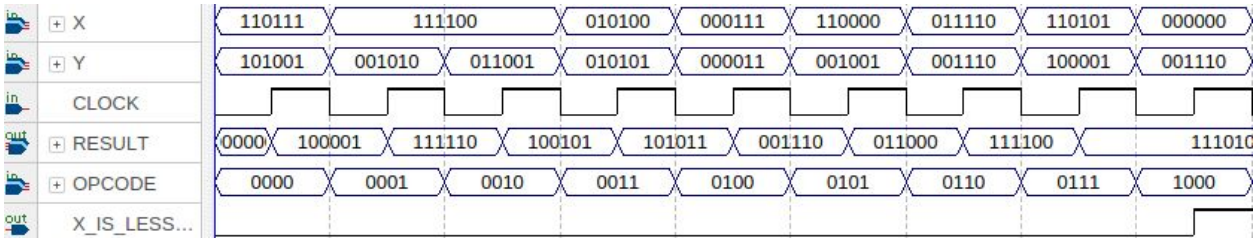

Figure 18: Vector waveform simulation for opcode component. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design an opcode component that performs 8 operations. When there was a rising 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.

11. OPCODE TEST

11.1 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 19: 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.

11.2 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.

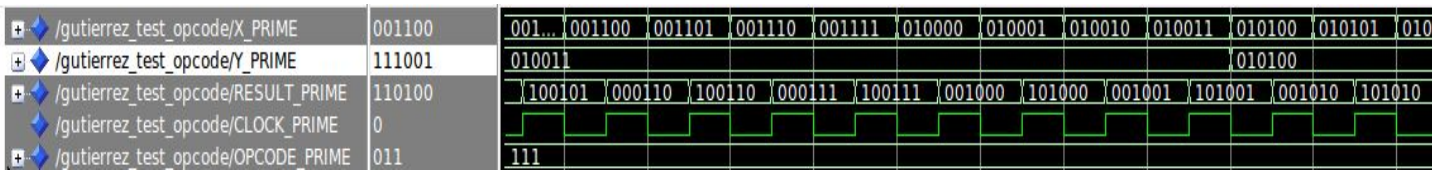

Figure 20: Vector waveform simulation for for test opcode component. As we can see we have no errors in our simulation or design, that means that our design was correct and that we did not make any mistakes, we were able to successfully design an opcode component. The test bench here was used to test every possible combination of 6-Bit words and every combination of the 3 bit opcode operations as well.

12 Demonstration of opcode component on DE2-115 Board.

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

CLOCK is assigned to PIN_Y23

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

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

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

X_IS_LESS_THAN_Y is assigned to PIN_J19

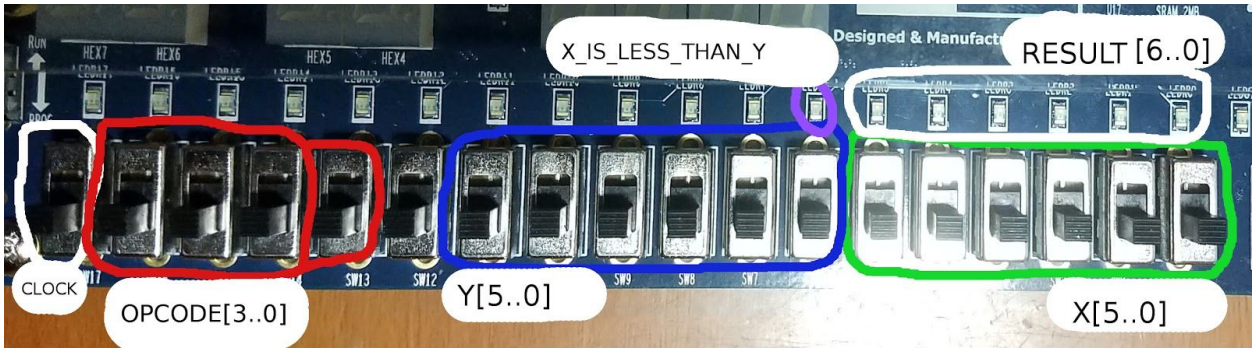

Figure 21: Digital Circuit of OPCODE component. We have set switches for the X 6 bit word and the Y 6 bit word as well as for the 4 bit opcode, the clock signal and led lights for the 6 bit result.

13. 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. 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.

14. Appendix.

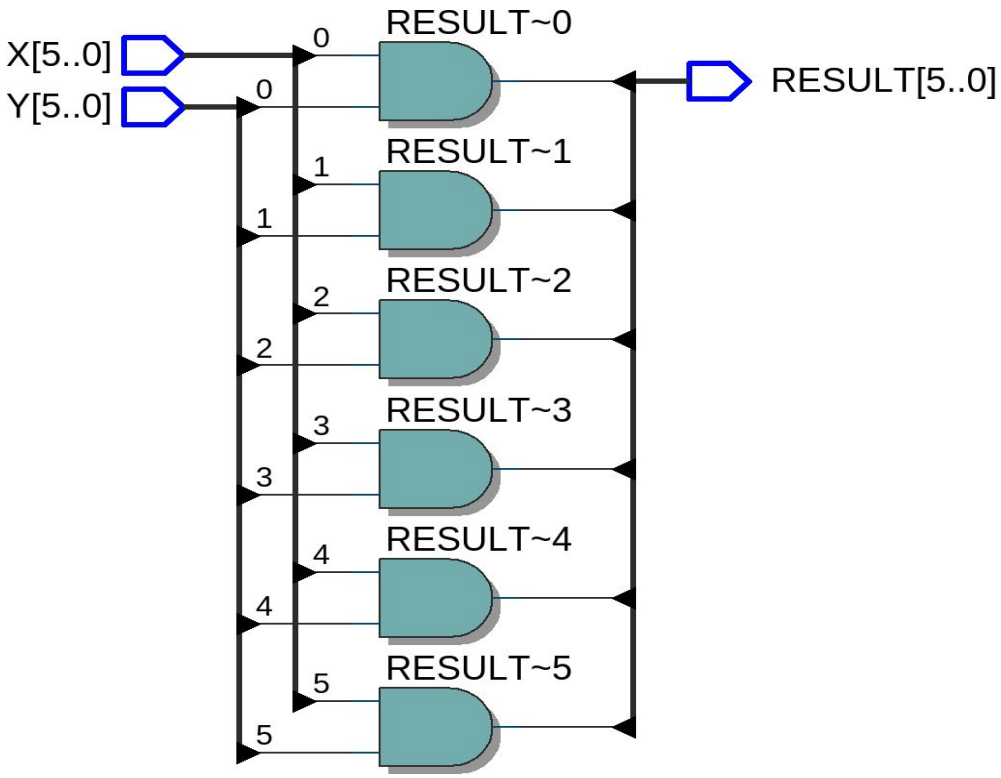

Figure 22: Block Diagram for bitwise and.

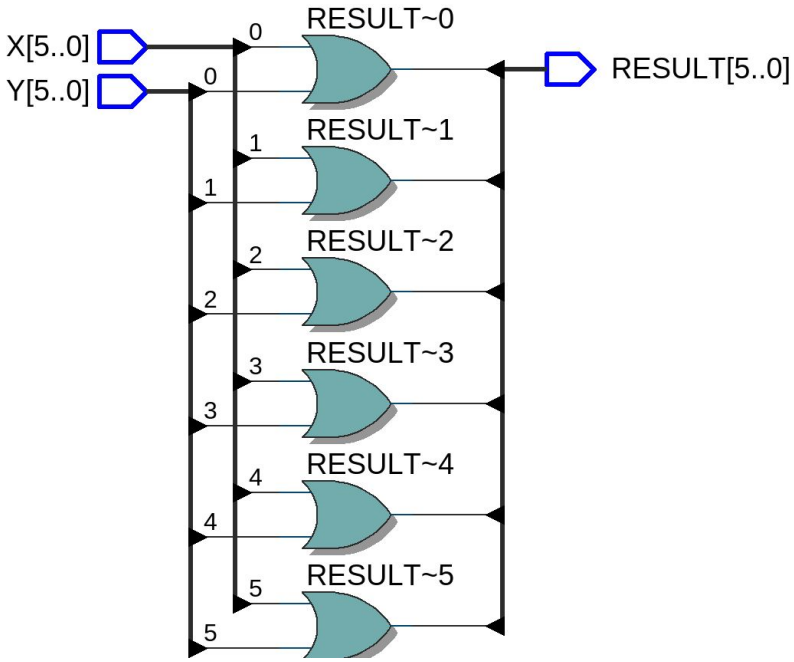

Figure 23: Block diagram for bitwise or.

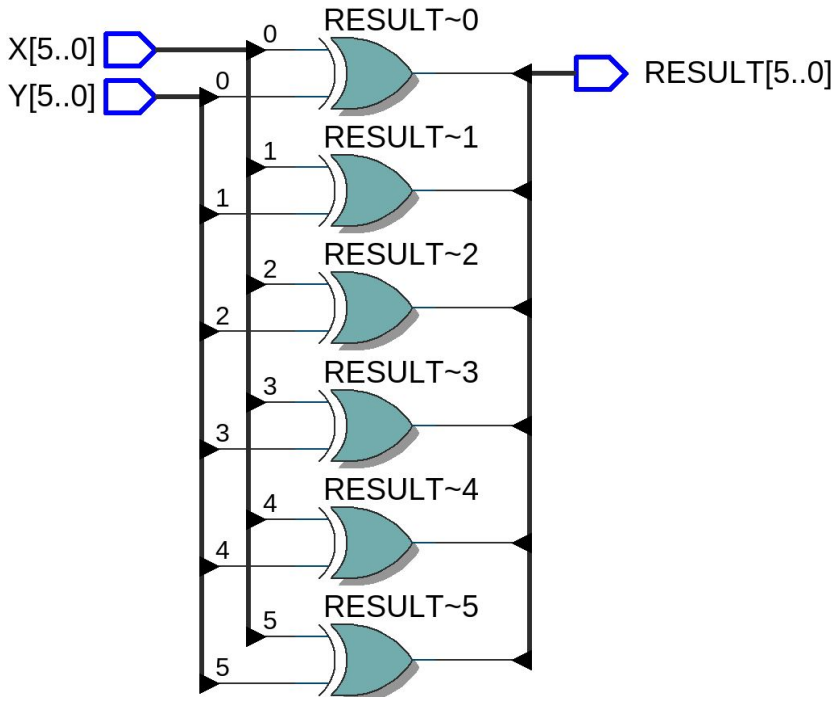

Figure 24: Block diagram for bitwise xor.

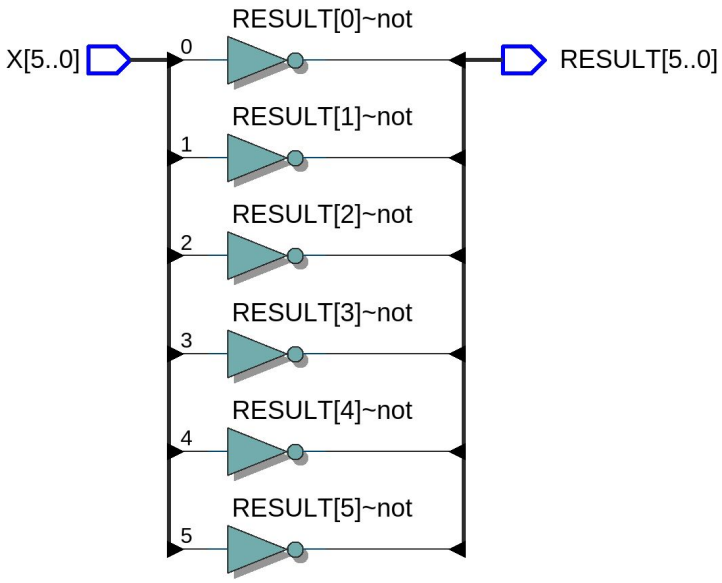

Figure 25: Block diagram for bitwise not.

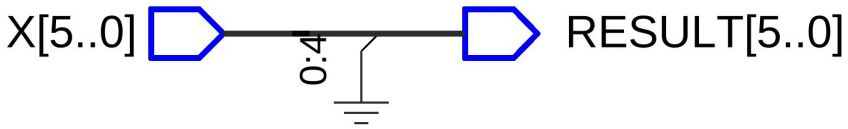

Figure 26: Block diagram for shift left.

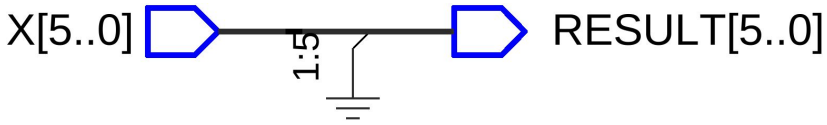

Figure 27: Block diagram for shift right.

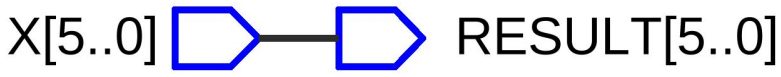

Figure 28: Block diagram for rotate left.

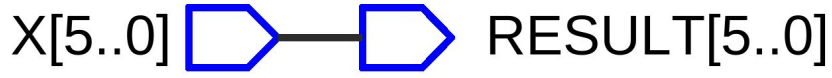

Figure 29: Block diagram for rotate right.

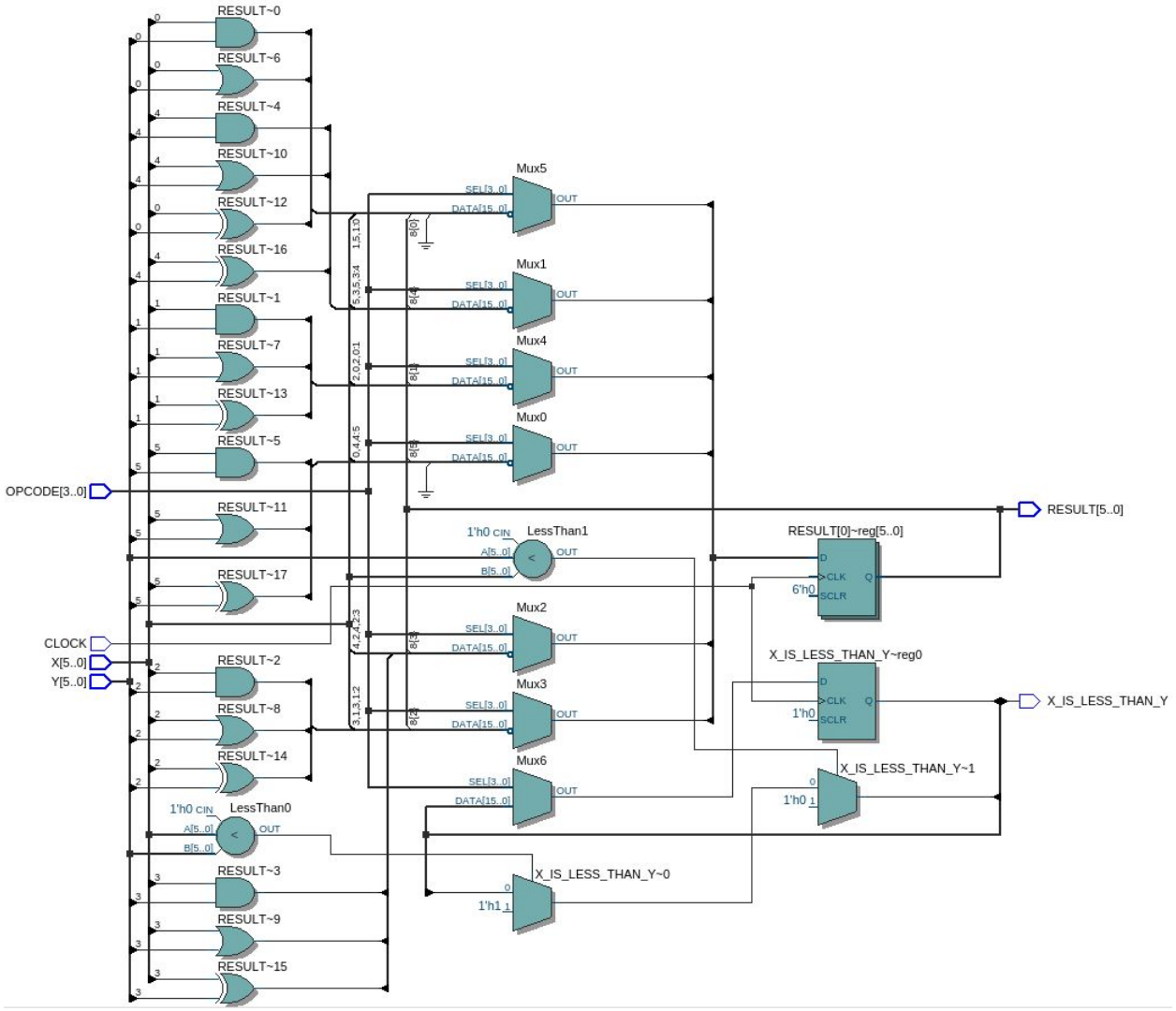

Figure 30: Block diagram for opcode component.

```
To, Location
CLOCK, PIN_Y23
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
OPCODE[0], PIN_AA24
OPCODE[1], PIN_AA23
OPCODE[2], PIN_AA22
OPCODE[3],PIN_Y24
RESULT[0], PIN_G19
RESULT[1], PIN_F19
RESULT[2], PIN_E19
RESULT[3], PIN_F21
RESULT[4], PIN_F18
RESULT[5], PIN_E18
X_IS_LESS_THAN_Y,PIN_J19
```

Figure 31: Pin assignments for opcode component on DE2-115 board.