ECE 501

Contemporary Digital Systems

Prof. Eric J. Balster

Homework #2

Aditya Sankaramanchi

101266059

**Introduction**

An Arithmetic and Logic unit (ALU) is a digital circuit that performs arithmetic operations like add, subtract, multiply, divide and logical operations such as AND, OR, NOT, etc. It is a fundamental building block of the central processing unit of a computer. In modern CPUs and GPUs a single ALU component may contain a number of ALUs. This gives very powerful and fast CPUs.

The data to be operated (also called operands) is given as an input to the ALU. The control unit gives a code to the ALU to specify which operation is to be performed. This code is called as operation code or opcode. The output is the result of these inputs.

To perform these operations in high speed, the operands are brought from the main memory to the storage elements of the processor called as registers. The registers are high speed storage elements typically with an access time 5 to 10 times of the main memory. A single register is capable of storing more than one word of data and frequently used operands.

In modern digital circuits, an ALU is specifically designed to perform integer arithmetic operations like two’s compliment and BCD. Complex arithmetic calculations which include floating point, complex numbers, etc. are designed in circuits called as floating point unit.

**Design**

The Logic circuitry in the arithmetic and logic unit is entirely combinational, which implements arithmetic operations such as ADD, SUBTRACT and logical operations such as AND, OR.

The truth table considered for the design of ALU is shown below

|  |  |
| --- | --- |
| OpCode | Output |
| 0 | A |
| 1 | B |
| 2 | A+B |
| 3 | A-B |
| 4 | Not A |
| 5 | Not B |
| 6 | Shift A left by 2 bits (mult. by 4) |
| 7 | Sign shift A right by 2 bits (div. by 4) |
| 8 | Shift B left by 2 bits (mult. by 4) |
| 9 | Sign shift N right by 2 bits (div. by 4) |
| 10 | A and B (bitwise AND) |
| 11 | A or B (bitwise OR) |
| Other | ERROR |

Table 1. Truth Table for the combinational logic circuit

The ALU contains inputs A, B and an Operation code. The Opcode is the instruction code that specifies the operation to be performed.

Bitwise Operations:

The bitwise operations are performed on one or more sequence of bits (zeros and ones) rather than decimal, hexadecimal or octal numbers. For example, consider the decimal number 9 with its binary representation as 1001. The bitwise operators perform their operations on the individual bits. This helps in a fast and primitive action directly supported by the processor.

The bitwise operators are of two types:

1. Bitwise logical operators
2. Bitwise shift operators

* Bitwise logical operators: In this type, each bit of the first operand is paired with its equivalent bit in the second operand and so on. The required operator is applied to the pair of bits and the result is obtained.

The bitwise logical operators are of 4 types:

1. Bitwise AND: The AND operation is performed on each pair of bits. The truth table for the AND operation is shown below:

|  |  |  |
| --- | --- | --- |
| A | B | A and B |
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

Table 2. Truth table for AND operation

Example: Consider two 4 bit operands A = 0011 (3); B= 0100 (4).

A and B = 0111 (7).

1. Bitwise OR: The OR operation is performed on each pair of bits. The truth table is shown below:

|  |  |  |
| --- | --- | --- |
| A | B | A or B |
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

Table 3. Truth table for OR operation

Example: Consider two 4 bit operands A= 1000 (8) and B= 0111 (7)

A or B = 1111 (15)

1. Bitwise NOT: The bitwise NOT performs a logical negation on each bit. This results in the inverted value. It is also called as one’s compliment. The truth table is shown below :

|  |  |
| --- | --- |
| A | Not A |
| 0 | 1 |
| 1 | 0 |

Table 4. Truth table for NOT operation

Example: Consider A= 1000 (8). Not A = 0111 (7)

1. Bitwise XOR: The bitwise XOR is the logical exclusive OR operation performed on each pair of corresponding bits. The output of the XOR is 1 if both the inputs are different. The truth table is shown below:

|  |  |  |
| --- | --- | --- |
| A | B | A XOR B |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Table 5: Truth table for XOR operation

Example: Consider two operands A= 1000 (8) and B= 0111 (7)

A XOR B = 1111 (15)

* Bitwise shift operators: With the shift operators, the bits are shifted to the left or right. The shift operators have two operands. The first is the quantity that is to be shifted and second denotes the number of bit positions the first operand is to be shifted. The operand on the left is converted to a 32 bit integer in big-endian order and then the shift operation is performed. The result is of the same type of the left operand.

Example 1: Left shit: The operand bits are shifted to the left starting from the most significant bit (MSB) and following to the LSB.

Syntax: A sll 2; where the operand A bits are shifted left by 2 bits.

Consider A= 0011 (3); A sll 2 = 1100 (12)

The equivalent is multiplying decimal A by 2^n, where n is the number of bits to be shifted.

Example 2: Right shift: The operand bits are shifted to the right starting from the LSB to the MSB.

Syntax: A srl 2; where the operand A bits are shifted right by 2 bits.

Consider A= 1000 (8); A srl 2 = 0010 (2).

The equivalent is dividing the decimal A by 2^n, where n is the number of bits to be shifted.

Design used in VHDL code: The design file for the ALU is written in behavioral design. The architecture contains the case and when statements.

Portion of the VHDL design code is shown below:

-- for output A\_in (0)

case Opcode is

when "0000" =>

result <= std\_logic\_vector(A\_in);

-- for output B\_in (1)

when "0001" =>

result <= std\_logic\_vector(B\_in);

The architecture can also be written using if statements as:

-- for output A\_in (0)

if(Opcode = “0000’”) then

result <= std\_logic\_vector(A\_in);

-- for output B\_in (1)

elsif(Opcode = "0001") then

result <= std\_logic\_vector(B\_in);

**Result:** The VHDL design file and the test bench is compiled and simulated in the ModelSim.

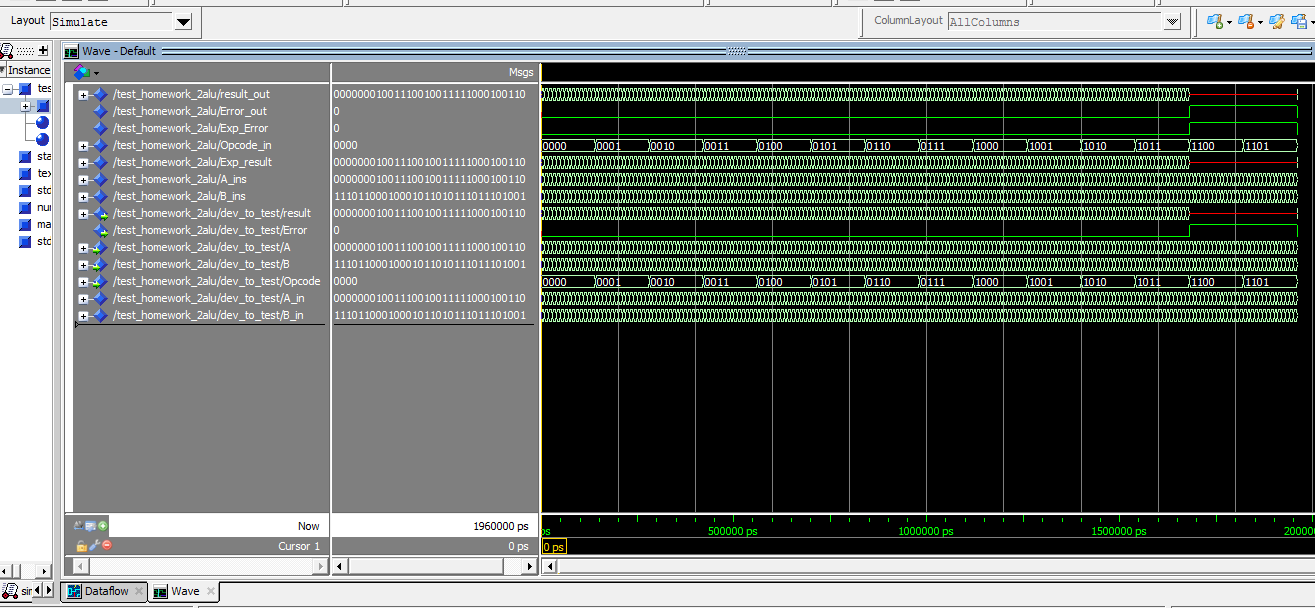


Figure 1: Simulation result for the ALU design.

The Transcript window is shown below

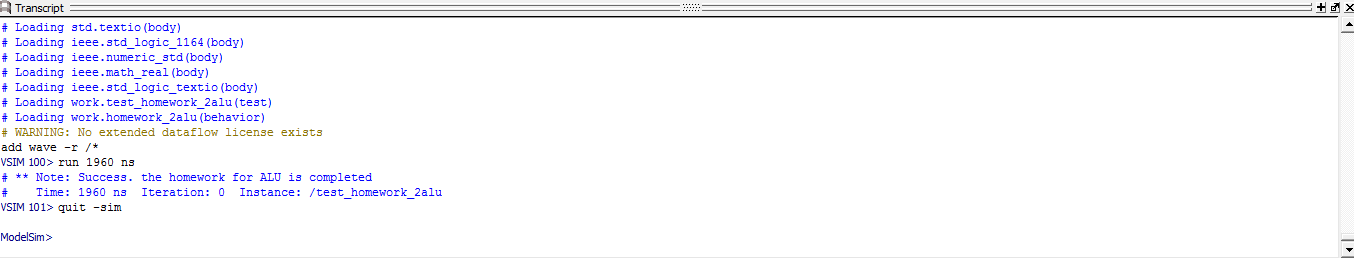


Figure 2: Transcript window of the ModelSim simulation

Explanation:

First the simulation is performed with a time period of 2500 ns. The result is obtained below:

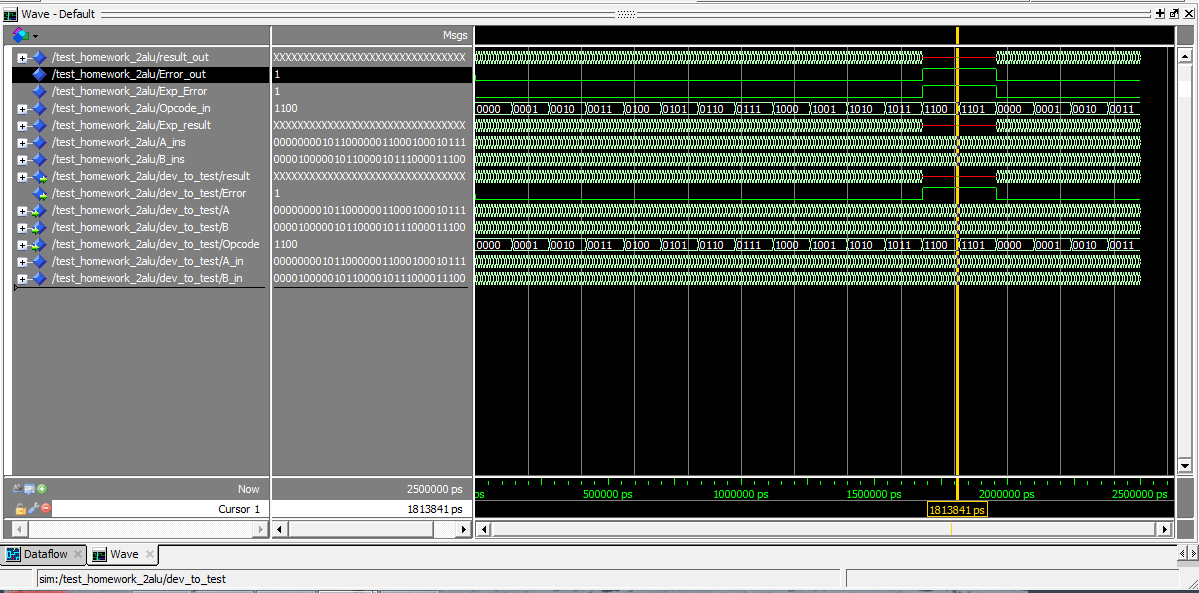


Fig 3: Simulation result for ALU design with t= 2500 ns

So the Opcode entries end at 1960 ns and the loop starts again. So the second simulation shown in Fig 1 is obtained with 1960 ns.

So the result is verified by choosing the required Opcode entry and checking the result\_out and Exp\_result.

Quartus output:

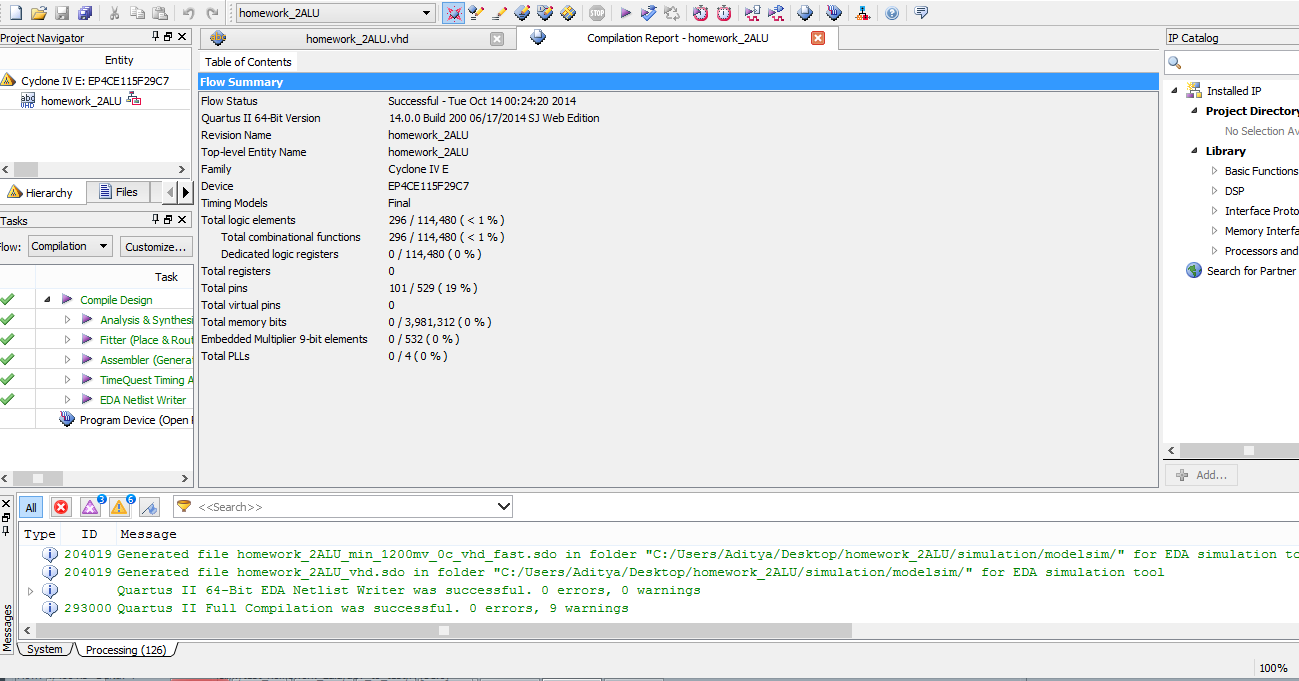


Fig 4:Quartus result for the ALU

Total logic elements = 296

As the ALU is a combinational logic circuit,

Total combinational functions = 296

Total pins = 101

RTL view of the ALU :

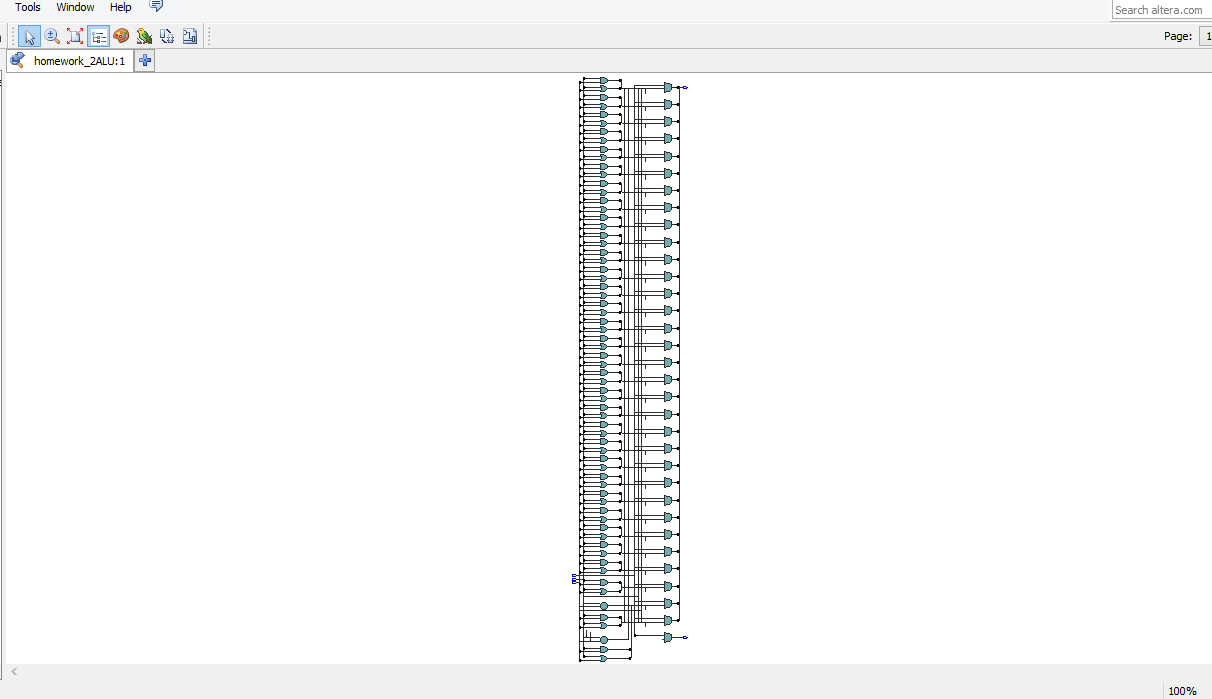


Fig 5. RTL view of the ALU

Now if the ALU design file is written using if statements, then the following results are obtained:

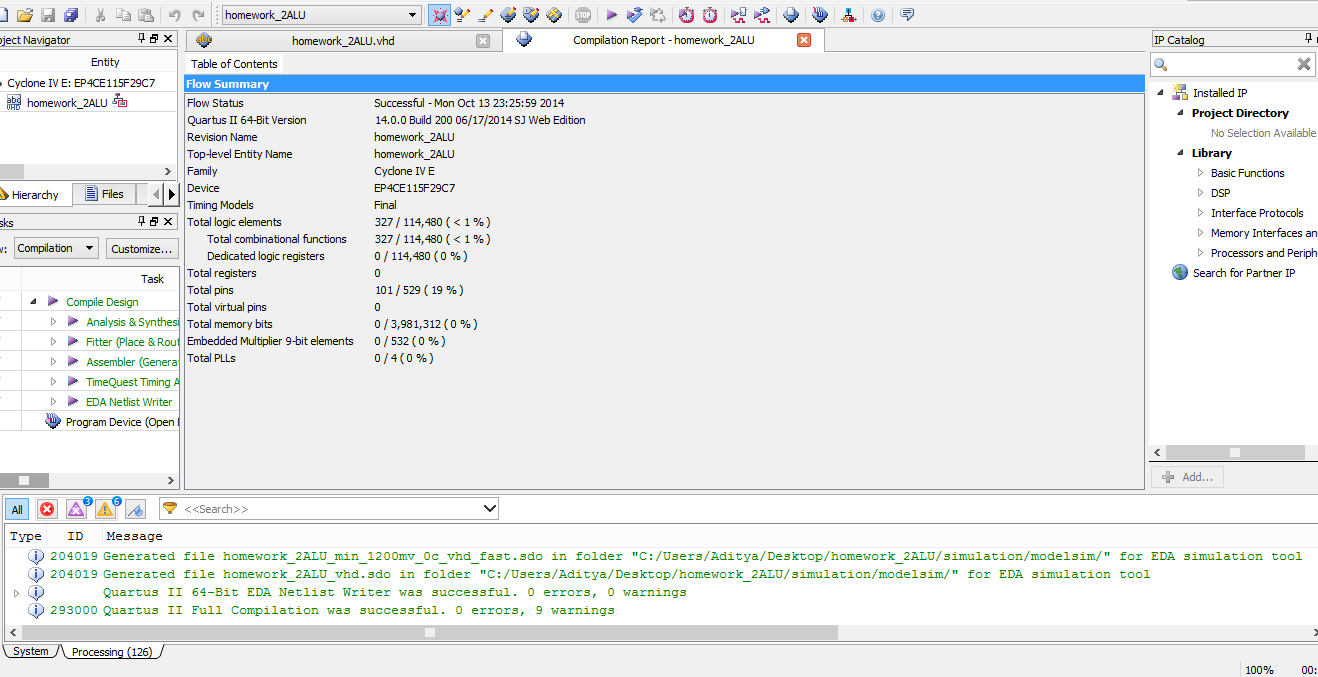


Fig 6. Quartus output when if statements are considered.

Total logic elements: 327

Total combinational functions = 327

Total pins = 101

So we can say that the logic elements increase because of the if statements used in the design.

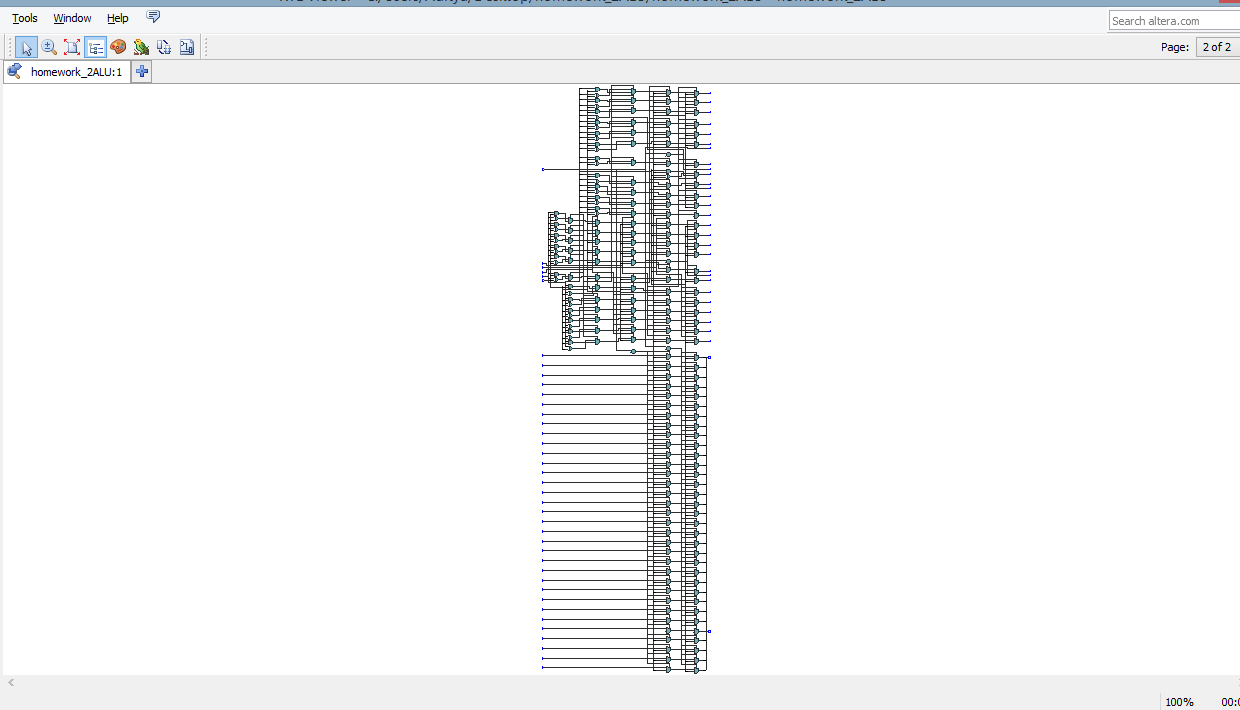


Fig 7. RTL view of the ALU when if statements are used in the design

**Conclusion:**

For the given truth table of the ALU, the VHDL code is written in the behavioral design. The inputs are A, B and the operation code is considered in the range of 0 to 11. The bitwise logical operations like AND, OR and bitwise shift operations like left shift and right shift operations are performed on the operands. If the opcode crosses 11 (1011), an ERROR signal is obtained in the output. A test bench is used for every opcode to check for its correctness. The simulation is done using ModelSim and the output waveforms are obtained. The output satisfied the required functionality. Further, Quartus is used in developing logic design for the truth table and the RTL views are obtained.

**Appendix- VHDL code:**

homework\_2ALU.vhd

-- Homework #2

--ALU

-- entity declaration

library IEEE;

use IEEE.std\_logic\_1164.all;

use IEEE.numeric\_std.all;

entity homework\_2ALU is -- given entity

generic (

bit\_depth : integer := 32 );

port (

result : out std\_logic\_vector(bit\_depth-1 downto 0);

Error : out std\_logic;

A : in std\_logic\_vector(bit\_depth-1 downto 0);

B : in std\_logic\_vector(bit\_depth-1 downto 0);

Opcode : in std\_logic\_vector(3 downto 0)

);

end homework\_2ALU;

architecture behavior of homework\_2ALU is

signal A\_in : signed(bit\_depth-1 downto 0); -- new signed signals

signal B\_in : signed(bit\_depth-1 downto 0);

begin

A\_in <= signed(A);

B\_in <= signed(B);

state\_op: process(Opcode, A\_in, B\_in)

begin

error <= '0';

-- for output A\_in (0)

case Opcode is

when "0000" =>

result <= std\_logic\_vector(A\_in);

-- for output B\_in (1)

when "0001" =>

result <= std\_logic\_vector(B\_in);

-- for output A\_in+B\_in (2)

when "0010" =>

result <= std\_logic\_vector(A\_in + B\_in);

-- for output A\_in-B\_in (3)

when "0011" =>

result <= std\_logic\_vector(A\_in - B\_in);

-- for output Not A\_in (4)

when "0100" =>

result <= std\_logic\_vector(not A\_in);

-- for output Not B\_in (5)

when "0101" =>

result <= std\_logic\_vector(not B\_in);

-- for output Shift A\_in left by 2 bits(mult. by 4) (6)

-- sll for shift left in the form A L sll R where L is the left operand A

-- and R is the number of times the shift is to be done

when "0110" =>

result <= std\_logic\_vector(A\_in sll 2);

-- now right shift A\_in by 2 bits(7)

when "0111" =>

result <= std\_logic\_vector(A\_in srl 2);

-- now left shift of B\_in (8)

when "1000" =>

result <= std\_logic\_vector(B\_in sll 2);

-- now right shift B\_in by 2 (9)

when "1001" =>

result <= std\_logic\_vector(B\_in srl 2);

-- now bitwise A\_in and B\_in 10

when "1010" =>

result <= std\_logic\_vector(A\_in) and std\_logic\_vector(B\_in);

-- now bitwise A\_in OR B\_in 11

when "1011" =>

result <= std\_logic\_vector(A\_in) or std\_logic\_vector(B\_in);

-- now for others

when others =>

result <= (others => 'X');

error <= '1';

end case;

end process;

end behavior;

test\_homework\_2ALU.vhd

-- testbench for ALU

library IEEE;

use IEEE.std\_logic\_1164.all;

use IEEE.numeric\_std.all;

use ieee.math\_real.all;

use std.textio.all;

use IEEE.std\_logic\_textio.all;

entity test\_homework\_2ALU is

end;

architecture test of test\_homework\_2ALU is

component homework\_2ALU

generic (

bit\_depth : integer := 32 );

port (

result : out std\_logic\_vector(bit\_depth-1 downto 0);

Error : out std\_logic;

A : in std\_logic\_vector(bit\_depth-1 downto 0);

B : in std\_logic\_vector(bit\_depth-1 downto 0);

Opcode : in std\_logic\_vector(3 downto 0)

);

end component;

constant bit\_depth : integer := 32;

signal result\_out : std\_logic\_vector(bit\_depth-1 downto 0);

signal Error\_out : std\_logic := '0';

signal Exp\_Error : std\_logic := '0';

signal Opcode\_in : std\_logic\_vector(3 downto 0) := (others => '0');

signal Exp\_result : std\_logic\_vector(bit\_depth - 1 downto 0);

signal A\_ins : signed(bit\_depth-1 downto 0):= (others => '0');

signal B\_ins : signed(bit\_depth-1 downto 0):= (others => '0');

constant max\_input\_value : integer := 2\*\*(bit\_depth-2); -- for random number

constant mid\_point : integer := max\_input\_value / 2;

begin

dev\_to\_test: homework\_2ALU

generic map(bit\_depth => bit\_depth)

port map(

result => result\_out, Error => Error\_out,

A => std\_logic\_vector(A\_ins), B => std\_logic\_vector(B\_ins), Opcode => Opcode\_in);

expected\_proc : process(A\_ins, B\_ins, Opcode\_in)

begin

Exp\_Error <= '0';

case Opcode\_in is

when "0000" => Exp\_result <= std\_logic\_vector(A\_ins);

when "0001" => Exp\_result <= std\_logic\_vector(B\_ins);

when "0010" => Exp\_result <= std\_logic\_vector(A\_ins+B\_ins);

when "0011" => Exp\_result <= std\_logic\_vector(A\_ins-B\_ins);

when "0100" => Exp\_result <= std\_logic\_vector(Not A\_ins);

when "0101" => Exp\_result <= std\_logic\_vector(Not B\_ins);

when "0110" => Exp\_result <= std\_logic\_vector(A\_ins sll 2);

when "0111" => Exp\_result <= std\_logic\_vector(A\_ins srl 2);

when "1000" => Exp\_result <= std\_logic\_vector(B\_ins sll 2);

when "1001" => Exp\_result <= std\_logic\_vector(B\_ins srl 2);

when "1010" => Exp\_result <= std\_logic\_vector(A\_ins and B\_ins);

when "1011" => Exp\_result <= std\_logic\_vector(A\_ins or B\_ins);

when others => Exp\_result <= (others => 'X');

Exp\_Error <= '1';

end case;

end process expected\_proc;

stimulus : process

variable ErrCnt : integer := 0;

variable WriteBuf : line ;

variable seed1, seed2 : positive;

variable rand, rval : real;

begin

for i in 0 to 13 loop

Opcode\_in <= std\_logic\_vector(to\_unsigned(i,4));

for j in 0 to 13 loop

UNIFORM(seed1, seed2, rand); -- for a random number

rval := trunc(rand\*real(max\_input\_value));

A\_ins <= to\_signed(integer(rval)-mid\_point,bit\_depth);

UNIFORM(seed1, seed2, rand); -- for a random number

rval := trunc(rand\*real(max\_input\_value));

B\_ins <= to\_signed(integer(rval)-mid\_point,bit\_depth);

wait for 10 ns;

if(Exp\_result /= result\_out) then

write(WriteBuf, string'("Error ALU test failed at result\_out : "));

write(WriteBuf, std\_logic\_vector(A\_ins));

write(WriteBuf, string'(", B ="));

write(WriteBuf, std\_logic\_vector(B\_ins));

write(WriteBuf, string'(", Opcode ="));

write(WriteBuf, Opcode\_in);

writeline(Output, WriteBuf);

ErrCnt := ErrCnt+1;

end if;

if(Exp\_Error /= Error\_out) then

write(WriteBuf, string'(" Error: Opcodes > 1011 giving output"));

writeline(Output, WriteBuf);

end if;

end loop;

end loop;

if (ErrCnt = 0) then

report "Success. the homework for ALU is completed";

else

report " the ALU is broken. " severity warning;

end if;

end process stimulus;

end test;