Faculty of Information Technology
Computer Systems Engineering Department

ENC3310– Advanced Digital Design
Course Project

Prepared by:
Baraa Fatony-1180566

Instructor: Dr. Abdellatif Abu-Issa

Date:30/8/2021

ABSTRACT:

In this project we will build a simple part of a microprocessor. Firstly we will build two main blocks: the ALU and the register file, then you will connect them together and run a simple machine code program on them, and to verify at the end that your result is correct.
The purpose of this project is to get used to describe hardware with VHDL , Also know how to deal with microprocessors in an effective way.

Table of content:

Theory:

1.1 Microprocessor:

Microprocessor is a controlling unit of a micro-computer, fabricated on a small chip capable of performing ALU (Arithmetic Logical Unit) operations and communicating with the other devices connected to it.
Microprocessor consists of an ALU, register array, and a control unit. ALU performs arithmetical and logical operations on the data received from the memory or an input device. Register array consists of registers identified by letters like B, C, D, E, H, L and accumulator. The control unit controls the flow of data and instructions within the computer [1].

Features of a Microprocessor:
Here is a list of some of the most prominent features of any microprocessor –

- Cost-effective – The microprocessor chips are available at low prices and results its low cost.
- Size – The microprocessor is of small size chip, hence is portable.
- Low Power Consumption – Microprocessors are manufactured by using metaloxide semiconductor technology, which has low power consumption.
- Versatility – The microprocessors are versatile as we can use the same chip in a number of applications by configuring the software program.
- Reliability – The failure rate of an IC in microprocessors is very low, hence it is reliable.

1.2 ALU:

An arithmetic-logic unit is the part of a central processing unit that carries out arithmetic and logic operations on the operands in computer instruction words.
In some processors, the ALU is divided into two units: an arithmetic unit (AU) and a logic unit (LU). Some processors contain more than one AU -- for example, one for fixed-point operations and another for floating-point operations.
In computer systems, floating-point computations are sometimes done by a floating-point unit (FPU) on a separate chip called a numeric coprocessor [2].

How does an arithmetic-logic unit work?
Typically, the ALU has direct input and output access to the processor controller, main memory (random access memory or RAM in a personal computer) and input/output devices. Inputs and outputs flow along an electronic path that is called a bus.
The input consists of an instruction word, sometimes called a machine instruction word, that contains an operation code or "opcode," one or more operands and sometimes a format code. The operation code tells the ALU what operation to perform and the operands are used in the operation.
For example, two operands might be added together or compared logically. The format may be combined with the opcode and tells, for example, whether this is a fixed-point or a floating-point instruction.

The output consists of a result that is placed in a storage register and settings that indicate whether the operation was performed successfully. If it isn't, some sort of status will be stored in a permanent place that is sometimes called the machine status word.
In general, the ALU includes storage places for input operands, operands that are being added, the accumulated result (stored in an accumulator) and shifted results. The flow of bits and the operations performed on them in the subunits of the ALU are controlled by gated circuits.
The gates in these circuits are controlled by a sequence logic unit that uses a particular algorithm or sequence for each operation code. In the arithmetic unit, multiplication and division are done by a series of adding or subtracting and shifting operations [2].

examples of bitwise logical operations and basic arithmetic operations supported by ALUs:

- Addition. Adds A and B with carry-in or carry-out sum at Y.
- Subtraction. Subtracts B from A or vice versa with the difference at Y and carry-in or carry-out.
- Increment. Where A or B is increased by one and Y represents the new value.
- Decrement. Where A or B is decreased by one and Y represents the new value.
- AND. The bitwise logic AND of A and B is represented by Y.
- OR. The bitwise logic OR of A and B is represented by Y.
- Exclusive-OR. The bitwise logic XOR of A and B is represented by Y.

1.3 Register File:

A register file is an array of processor registers in a central processing unit (CPU). Modern integrated circuit-based register files are usually implemented by way of fast static RAMs with multiple ports. Such RAMs are distinguished by having dedicated read and write ports, whereas ordinary multiported SRAMs will usually read and write through the same ports.
The instruction set architecture of a CPU will almost always define a set of registers which are used to stage data between memory and the functional units on the chip. In simpler CPUs, these architectural registers correspond one-for-one to the entries in a physical register file (PRF) within the CPU. More complicated CPUs use register renaming, so that the mapping of which physical entry stores a particular architectural register changes dynamically during execution. The register file is part of the architecture and visible to the programmer, as opposed to the concept of transparent caches [3].

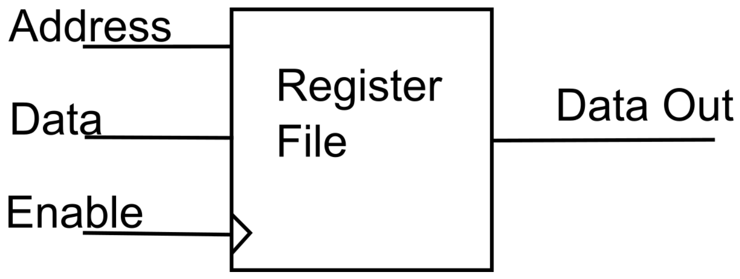


Figure 1.3.1: Register file.

Building microprocessor:

2.1 ALU:

First component of our microprocessor is ALU, so we wrote VHDL description of an ALU with two 32-bit inputs, A and B, and a 32-bit output Result.

The result is derived from one or both of the inputs according to the value of a 6-bit opcode. The operations that the ALU can perform are listed below:

- a + b , opcode = "000100"
- a - b , opcode = "001010"
- |a| , opcode = "000011"
- -a , opcode = "001100"
- max (a, b) , opcode = "001001"
- min (a, b) , opcode = "000010"
- avg(a,b) , (the integer part only) opcode = "000110"
- not a , opcode = "001101"
- a or b , opcode = "001110"
- a and b , opcode = "001011"
- a xor b , opcode = "001000"

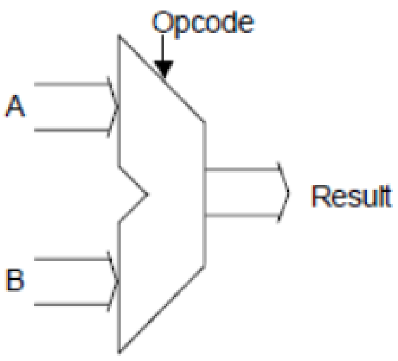
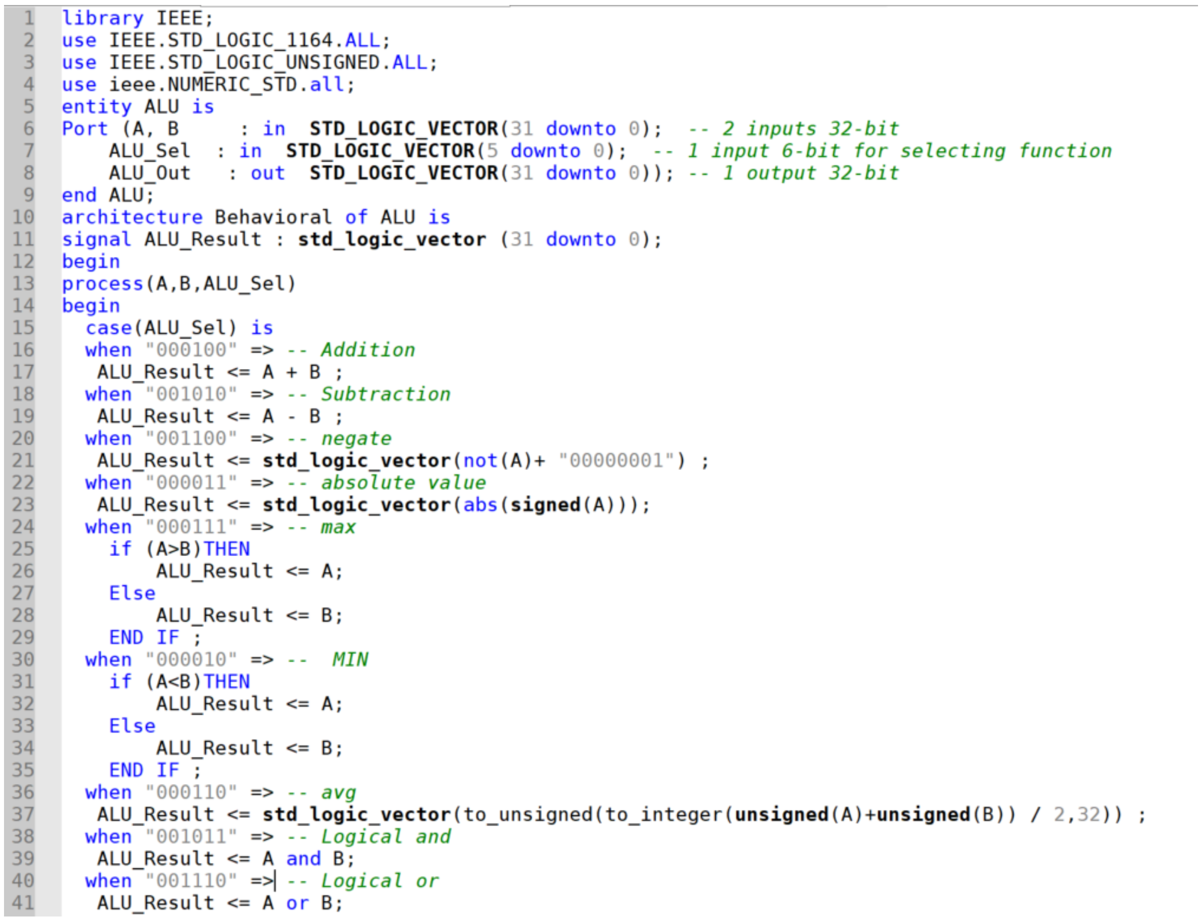


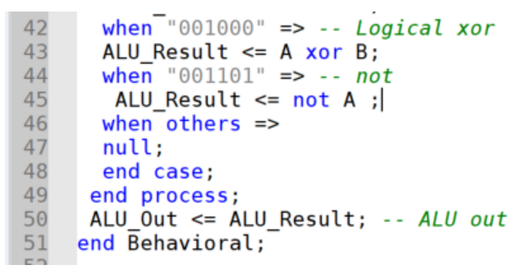
Figure 2.1.1: ALU.

VHDL code for ALU:



```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.NUMERIC_STD.all;
entity ALU is
Port (A, B     : in  STD_LOGIC_VECTOR(31 downto 0);  -- 2 inputs 32-bit
    ALU_Sel  : in  STD_LOGIC_VECTOR(5 downto 0);  -- 1 input 6-bit for selecting function
    ALU_Out   : out  STD_LOGIC_VECTOR(31 downto 0)); -- 1 output 32-bit
end ALU;
architecture Behavioral of ALU is
signal ALU_Result : std_logic_vector (31 downto 0);
begin
process(A,B,ALU_Sel)
begin
  case(ALU_Sel) is
  when "000100" => -- Addition
   ALU_Result <= A + B ;
  when "001010" => -- Subtraction
   ALU_Result <= A - B ;
  when "001100" => -- negate
   ALU_Result <= std_logic_vector(not(A)+ "00000001") ;
  when "000011" => -- absolute value
   ALU_Result <= std_logic_vector(abs(signed(A)));
  when "000111" => -- max
    if (A>B)THEN
        ALU_Result <= A;
    Else
        ALU_Result <= B;
    END IF ;
  when "000010" => --  MIN
    if (A<B)THEN
        ALU_Result <= A;
    Else
        ALU_Result <= B;
    END IF ;
  when "000110" => -- avg
   ALU_Result <= std_logic_vector(to_unsigned(to_integer(unsigned(A)+unsigned(B)) / 2,32)) ;
  when "001011" => -- Logical and
   ALU_Result <= A and B;
  when "001110" => -- Logical or
   ALU_Result <= A or B;
```

Figure 2.1.2: code of ALU.



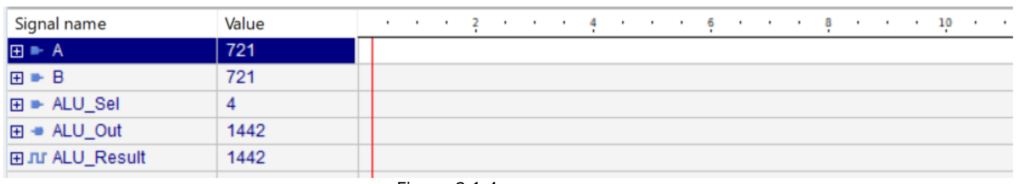
```
    when "001000" => -- Logical xor
    ALU_Result <= A xor B;
    when "001101" => -- not
     ALU_Result <= not A ;
    when others =>
    null;
    end case;
   end process;
   ALU_Out <= ALU_Result; -- ALU out
end Behavioral;
```

Figure 2.1.3: code of ALU.

note: this code will be in Appendix A.

VHDL simulation for ALU:

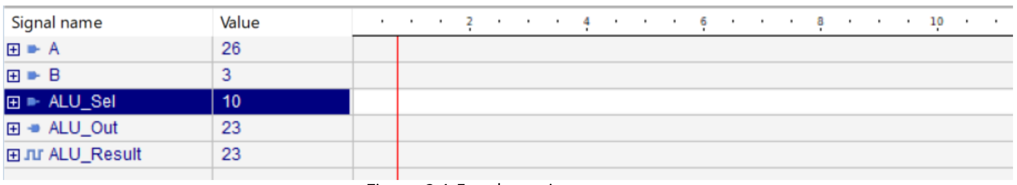
Below some samples of simulation of ALU for different operations .



Signal name	Value
A	721
B	721
ALU_Sel	4
ALU_Out	1442
ALU_Result	1442

Figure 2.1.4: sum.

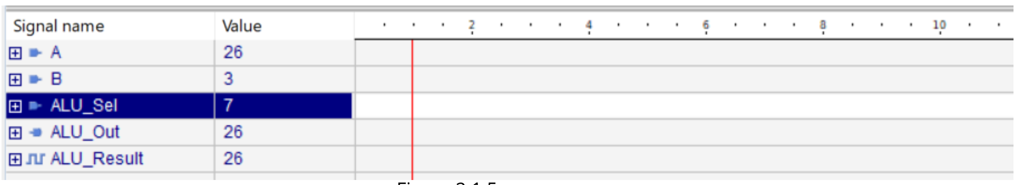
A=721 , B=721 , opcode is 4 , then out= A+B=1442.



Signal name	Value
A	26
B	3
ALU_Sel	10
ALU_Out	23
ALU_Result	23

Figure 2.1.5: subtraction.

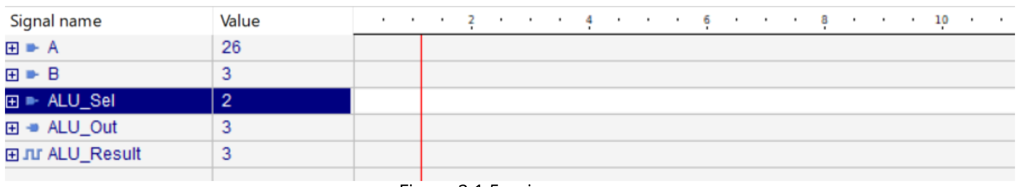
A=26 , B=3 , opcode is 10 , then out= A-B=23.



Signal name	Value
A	26
B	3
ALU_Sel	7
ALU_Out	26
ALU_Result	26

Figure 2.1.5: max.

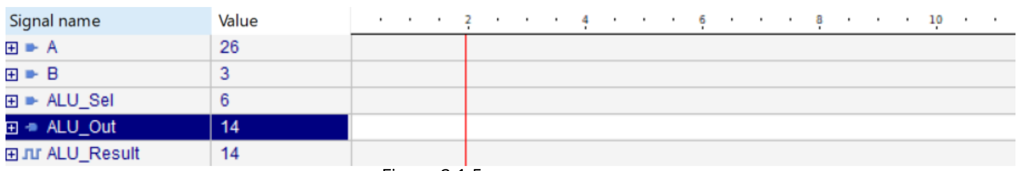
A=26 , B=3 , opcode is 7 , then out= max (A,B)=26.



Signal name	Value
A	26
B	3
ALU_Sel	2
ALU_Out	3
ALU_Result	3

Figure 2.1.5: min.

A=26 , B=3 , opcode is 3 , then out= min (A,B)=3.



Signal name	Value
A	26
B	3
ALU_Sel	6
ALU_Out	14
ALU_Result	14

Figure 2.1.5: average.

A=26 , B=3 , opcode is 3, then out= Avg (A,B)=(26+3)/2=14.5 =14(integer part only).

2.2 Register file:

Inside a modern processor there is a very small amount of memory that is used to hold the operands that it is presently working on. This is called the register file, and normally has the following appearance.
Second component of our microprocessor is register ,in figure 2.2.1 is a very small fast RAM, typically holding 32 x 32-bit words, and therefore requiring a 5-bit address to select out one of the 32-bit words. It is unlike normal RAM in that it can process three addresses at the same time, two of which are always read operations, and one of which is always written to.

Output 1 produces the item within the register file that is address by Address 1. Similarly Output 2 produces the item within the register file that is address by Address 2. Input is used to supply a value that is written into the location addressed by Address 3.

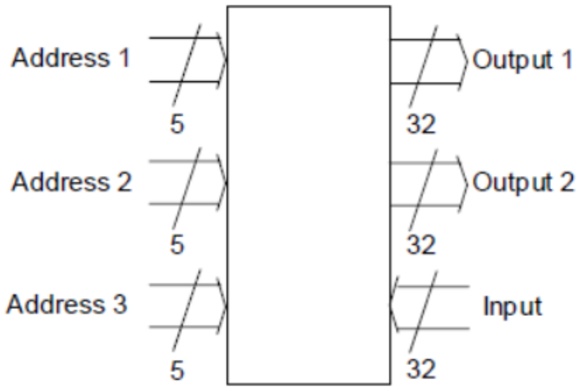
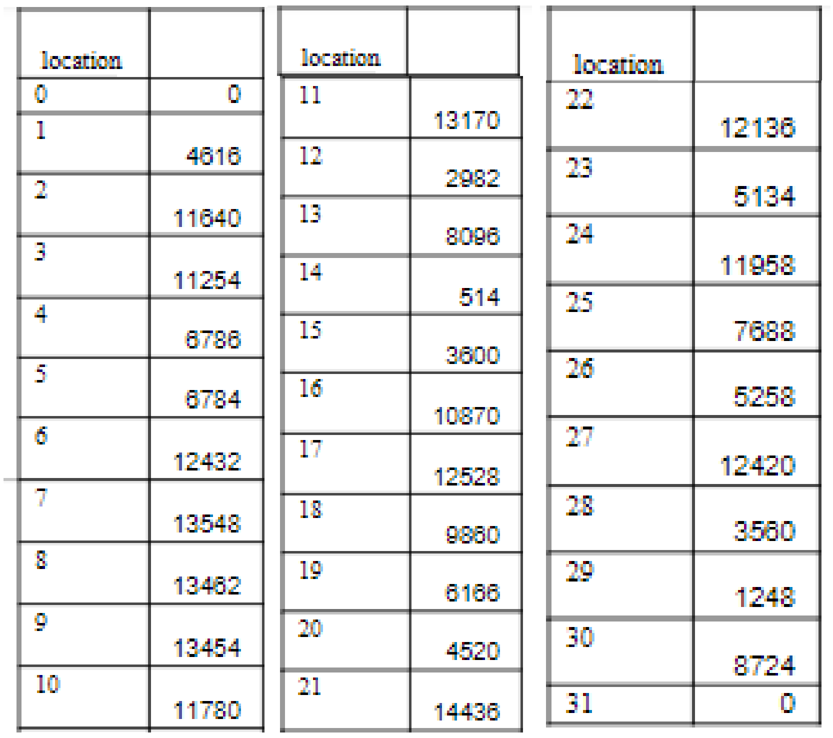


Figure 2.2.1: register file.

The initial values stored in the register file are determined as table below:



location	
0	0
1	4616
2	11640
3	11254
4	6786
5	6784
6	12432
7	13548
8	13462
9	13454
10	11780

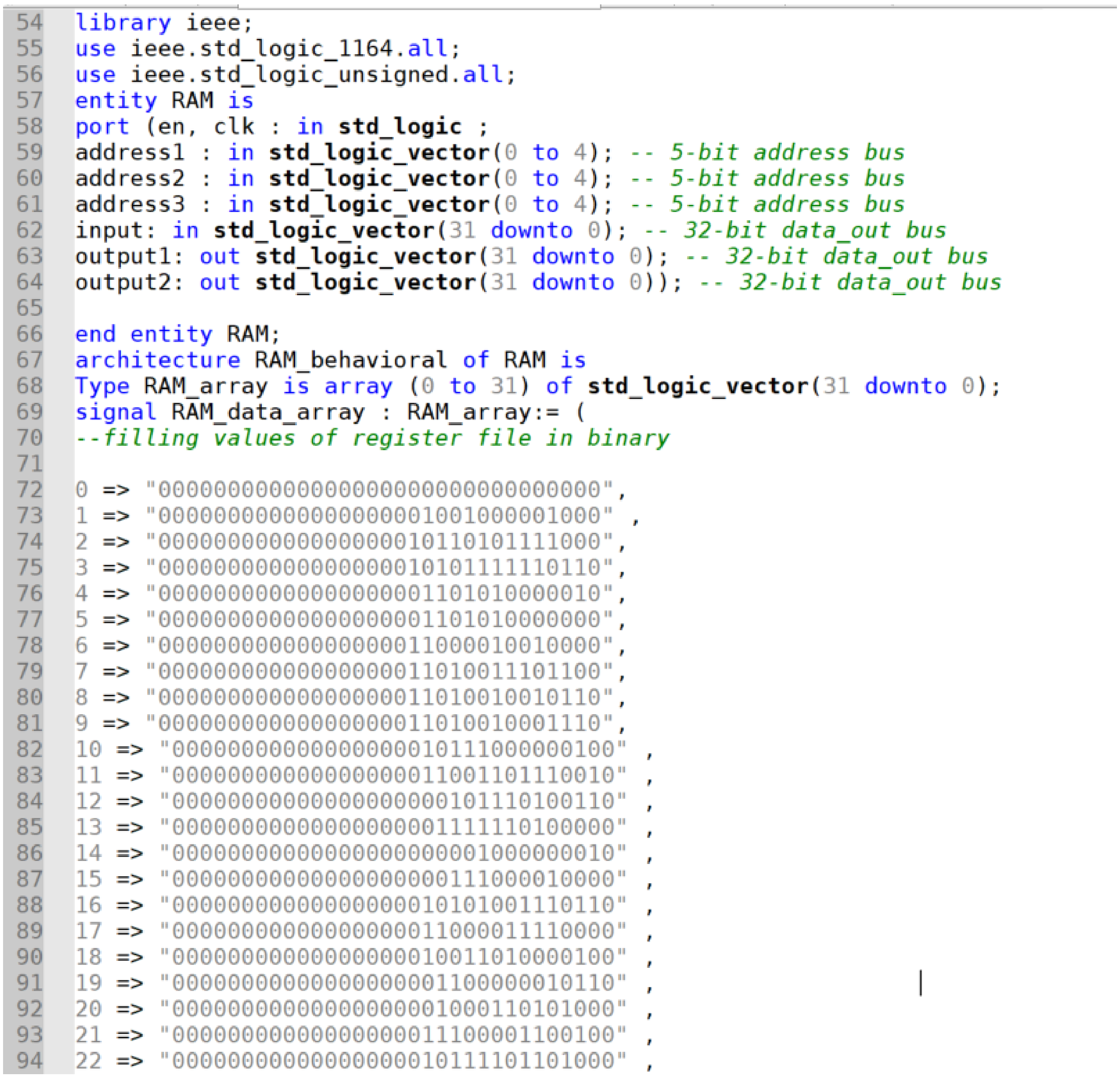
location	
11	13170
12	2982
13	8096
14	514
15	3600
16	10870
17	12528
18	9860
19	6166
20	4520
21	14436

location	
22	12136
23	5134
24	11958
25	7688
26	5258
27	12420
28	3560
29	1248
30	8724
31	0

Figure 2.2.2: values in register file.

VHDL code for Register file:

We wrote VHDL code to describe register file , as figures below .

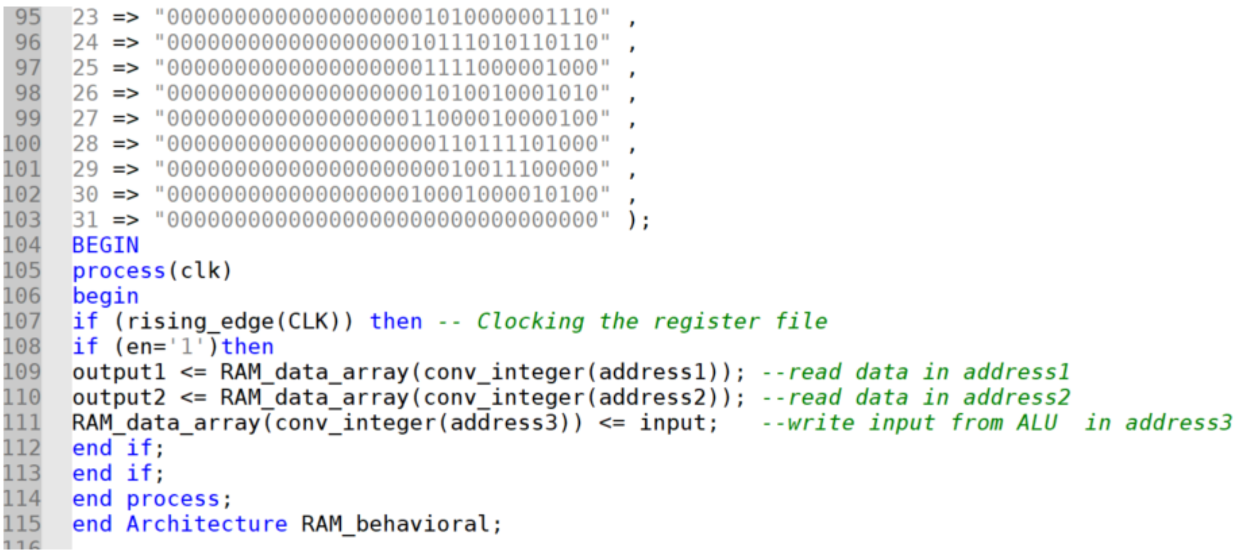


```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity RAM is
port (en, clk : in std_logic ;
address1 : in std_logic_vector(0 to 4); -- 5-bit address bus
address2 : in std_logic_vector(0 to 4); -- 5-bit address bus
address3 : in std_logic_vector(0 to 4); -- 5-bit address bus
input: in std_logic_vector(31 downto 0); -- 32-bit data_out bus
output1: out std_logic_vector(31 downto 0); -- 32-bit data_out bus
output2: out std_logic_vector(31 downto 0)); -- 32-bit data_out bus

end entity RAM;
architecture RAM_behavioral of RAM is
Type RAM_array is array (0 to 31) of std_logic_vector(31 downto 0);
signal RAM_data_array : RAM_array:= (
--filling values of register file in binary

0 => "00000000000000000000000000000000",
1 => "00000000000000000001001000001000" ,
2 => "00000000000000000010110101111000",
3 => "00000000000000000010101111110110",
4 => "00000000000000000001101010000010",
5 => "00000000000000000001101010000000",
6 => "00000000000000000011000010010000",
7 => "00000000000000000011010011101100",
8 => "00000000000000000011010010010110",
9 => "00000000000000000011010010001110",
10 => "00000000000000000010111000000100" ,
11 => "00000000000000000011001101110010" ,
12 => "00000000000000000000101110100110" ,
13 => "00000000000000000001111110100000" ,
14 => "00000000000000000000001000000010" ,
15 => "00000000000000000000111000010000" ,
16 => "00000000000000000010101001110110" ,
17 => "00000000000000000011000011110000" ,
18 => "00000000000000000010011010000100" ,
19 => "00000000000000000001100000010110" ,
20 => "00000000000000000001000110101000" ,
21 => "00000000000000000011100001100100" ,
22 => "00000000000000000010111101101000" ,
```

Figure 2.2.3: code of register file.



```
23 => "00000000000000000001010000001110" ,
24 => "00000000000000000001011101011011 0" ,
25 => "00000000000000000001111000001000" ,
26 => "00000000000000000001010010001010" ,
27 => "00000000000000000011000010000100" ,
28 => "00000000000000000000110111101000" ,
29 => "00000000000000000000010011100000" ,
30 => "00000000000000000010001000010100" ,
31 => "00000000000000000000000000000000" );
BEGIN
process(clk)
begin
if (rising_edge(CLK)) then -- Clocking the register file
if (en='1')then
output1 <= RAM_data_array(conv_integer(address1)); --read data in address1
output2 <= RAM_data_array(conv_integer(address2)); --read data in address2
RAM_data_array(conv_integer(address3)) <= input;   --write input from ALU  in address3
end if;
end if;
end process;
end Architecture RAM_behavioral;
```

Figure 2.2.4: code of register file.

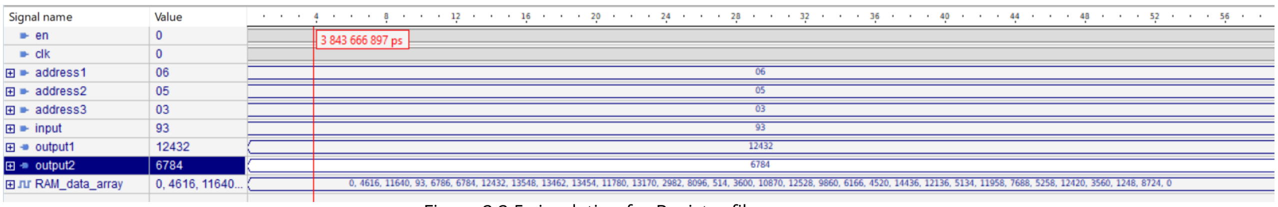
note: this code will be in Appendix B.

We gave the register file an enable input. When the enable input=1 the register file will operate normally, otherwise the register file will ignore its inputs, and will not update its outputs ,that solve the problem become when the simulation initializes (which corresponds to the real hardware being switched on) all the values of the logic signals initializes to some random garbage value (denoted ‘U’ in VHDL, but in real life either a ‘1’ or a ‘0’ chosen at random).

we synchronised the register file to a clock. We added an extra input named clock, and gave the register file the following behaviour:
On the rising edge of the clock:
• Output 1 produces the item within the register file that is address by Address 1.
• Output 2 produces the item within the register file that is address by Address 2.
• Input is used to supply a value that is written into the location addressed by Address 3.
Under all other circumstances:
• the outputs are held constant at the values they assumed during the last rising edge of the clock.
All of that to solve the problem of writing and reading from RAM in the same time

VHDL simulation for Register file:

In Figure 2.2.5 below is sample of simulation of register file.



Signal name	Value
en	0
clk	0
address1	06
address2	05
address3	03
input	93
output1	12432
output2	6784
RAM_data_array	0, 4616, 11640...

```
3 843 666 897 ps
0, 4616, 11640, 93, 6786, 6784, 12432, 13548, 13462, 13454, 11780, 13170, 2982, 8096, 514, 3600, 10870, 12528, 9860, 6166, 4520, 14436, 12136, 5134, 11958, 7688, 5258, 12420, 3560, 1248, 8724, 0
```

Figure 2.2.5:simulation for Register file.

In this sample we put address1=6 , so output1=array(6)=12432, address2=5, so output2=array(5) =6784 , and address3=3 , so write on array(3) to be as input =93.

2.3 Microprocessor:

In this part we connected all components together to make a system like figure 2.3.1 below.

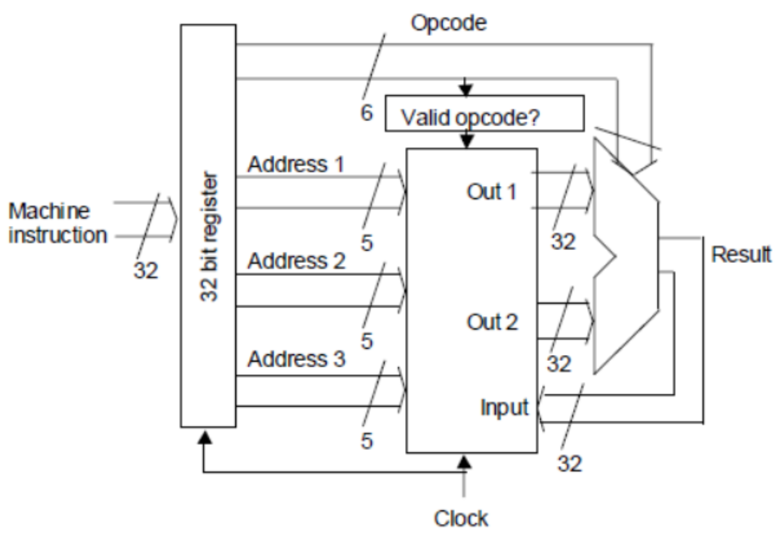


Figure 2.3.1: microprocessor.

Machine instructions are supplied to this arrangement in the form of 32-bit numbers. The format of these instructions is as follows:

- The first 6 bits identify the opcode
- The next 5 bits identify first source register
- The next 5 bits identify second source register
- The next 5 bits identify destination register
- The final 11 bits are unused

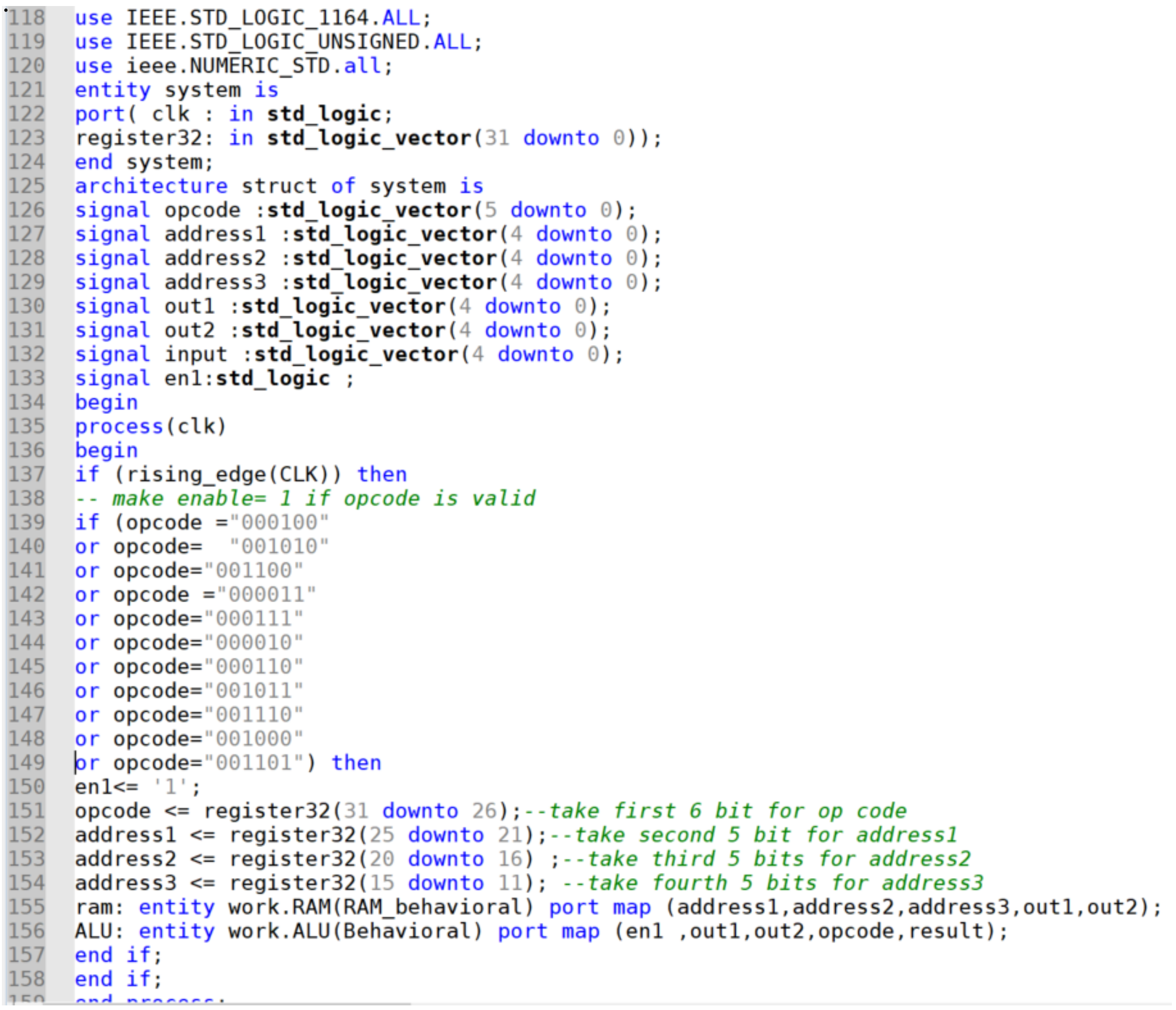
So, for example, if you want to add the contents of register 1 and register 2 and put the result into register 3, then the machine instruction would be as follows:

- The first 6 bits supply the opcode for the add instruction
- The next 5 bits would address register 1, and the next 5 would address register 2
- The next 5 bits address register 3.
- The remaining bits are unused, and should be set to zero.

The enable signal to the register should go high when the opcode contains a valid value, and should be low otherwise.

VHDL code for system:

We wrote VHDL code for system to connecting instance of ALU with clock and instance register file.

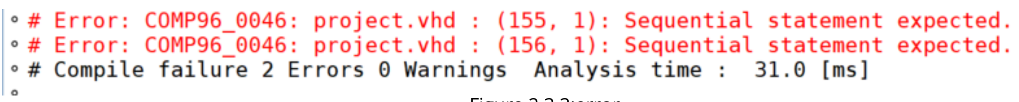


```
118 use IEEE.STD_LOGIC_1164.ALL;
119 use IEEE.STD_LOGIC_UNSIGNED.ALL;
120 use ieee.NUMERIC_STD.all;
121 entity system is
122 port( clk : in std_logic;
123 register32: in std_logic_vector(31 downto 0));
124 end system;
125 architecture struct of system is
126 signal opcode :std_logic_vector(5 downto 0);
127 signal address1 :std_logic_vector(4 downto 0);
128 signal address2 :std_logic_vector(4 downto 0);
129 signal address3 :std_logic_vector(4 downto 0);
130 signal out1 :std_logic_vector(4 downto 0);
131 signal out2 :std_logic_vector(4 downto 0);
132 signal input :std_logic_vector(4 downto 0);
133 signal en1:std_logic ;
134 begin
135 process(clk)
136 begin
137 if (rising_edge(CLK)) then
138 -- make enable= 1 if opcode is valid
139 if (opcode ="000100"
140 or opcode=  "001010"
141 or opcode="001100"
142 or opcode ="000011"
143 or opcode="000111"
144 or opcode="000010"
145 or opcode="000110"
146 or opcode="001011"
147 or opcode="001110"
148 or opcode="001000"
149 or opcode="001101") then
150 en1<= '1';
151 opcode <= register32(31 downto 26);--take first 6 bit for op code
152 address1 <= register32(25 downto 21);--take second 5 bit for address1
153 address2 <= register32(20 downto 16) ;--take third 5 bits for address2
154 address3 <= register32(15 downto 11); --take fourth 5 bits for address3
155 ram: entity work.RAM(RAM_behavioral) port map (address1,address2,address3,out1,out2);
156 ALU: entity work.ALU(Behavioral) port map (en1 ,out1,out2,opcode,result);
157 end if;
158 end if;
159 end process;
```

Figure 2.3.2:code for microprocessor.

note: this code in Appendix C.

when I compiled previous code I have error in figure 2.3.3 , and I tried many methods to solve it but without benefit.

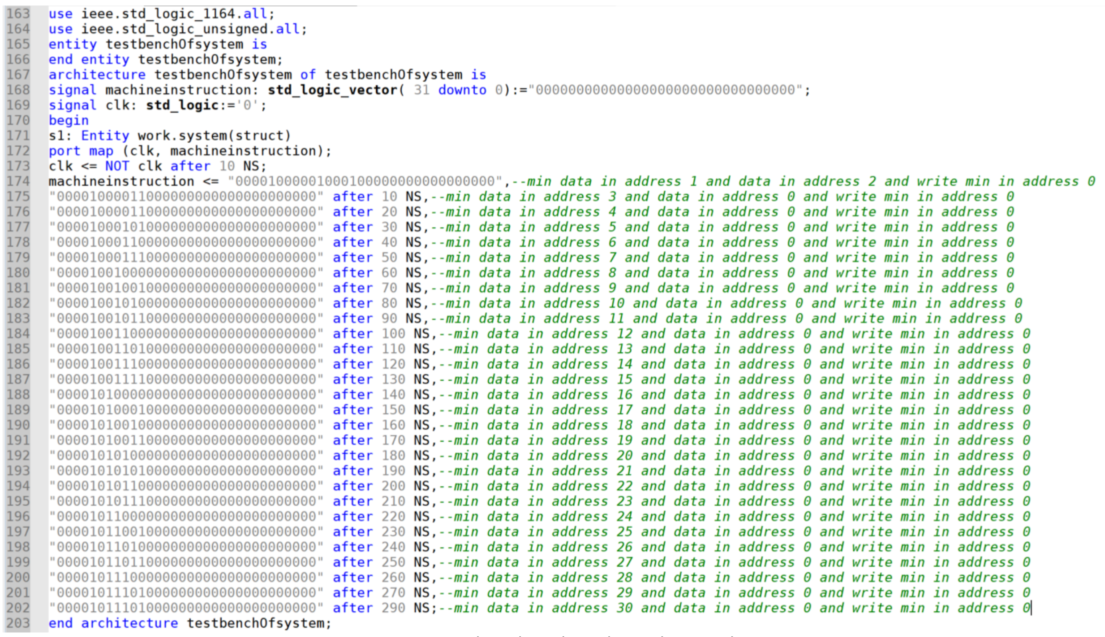


```
# Error: COMP96_0046: project.vhd : (155, 1): Sequential statement expected.
# Error: COMP96_0046: project.vhd : (156, 1): Sequential statement expected.
# Compile failure 2 Errors 0 Warnings  Analysis time :  31.0 [ms]
```

Figure 2.3.3:error.

2.4 testbench:

We wrote VHDL test bench to test the system if work correctly with machine code to find the minimum number stored in addresses from1 to 30 and write it in address 0.



```
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity testbenchOfsystem is
end entity testbenchOfsystem;
architecture testbenchOfsystem of testbenchOfsystem is
signal machineinstruction: std_logic_vector( 31 downto 0):="00000000000000000000000000000000";
signal clk: std_logic:='0';
begin
s1: Entity work.system(struct)
port map (clk, machineinstruction);
clk <= NOT clk after 10 NS;
machineinstruction <= "00001000001000100000000000000000",--min data in address 1 and data in address 2 and write min in address 0
"0000100001100000000000000000000" after 10 NS,--min data in address 3 and data in address 0 and write min in address 0
"0000100001100000000000000000000" after 20 NS,--min data in address 4 and data in address 0 and write min in address 0
"0000100010100000000000000000000" after 30 NS,--min data in address 5 and data in address 0 and write min in address 0
"0000100011000000000000000000000" after 40 NS,--min data in address 6 and data in address 0 and write min in address 0
"0000100011100000000000000000000" after 50 NS,--min data in address 7 and data in address 0 and write min in address 0
"0000100100000000000000000000000" after 60 NS,--min data in address 8 and data in address 0 and write min in address 0
"0000100100100000000000000000000" after 70 NS,--min data in address 9 and data in address 0 and write min in address 0
"0000100101000000000000000000000" after 80 NS,--min data in address 10 and data in address 0 and write min in address 0
"0000100101100000000000000000000" after 90 NS,--min data in address 11 and data in address 0 and write min in address 0
"0000100110000000000000000000000" after 100 NS,--min data in address 12 and data in address 0 and write min in address 0
"0000100110100000000000000000000" after 110 NS,--min data in address 13 and data in address 0 and write min in address 0
"0000100111000000000000000000000" after 120 NS,--min data in address 14 and data in address 0 and write min in address 0
"0000100111100000000000000000000" after 130 NS,--min data in address 15 and data in address 0 and write min in address 0
"0000101000000000000000000000000" after 140 NS,--min data in address 16 and data in address 0 and write min in address 0
"0000101000100000000000000000000" after 150 NS,--min data in address 17 and data in address 0 and write min in address 0
"0000101001000000000000000000000" after 160 NS,--min data in address 18 and data in address 0 and write min in address 0
"0000101001100000000000000000000" after 170 NS,--min data in address 19 and data in address 0 and write min in address 0
"0000101010000000000000000000000" after 180 NS,--min data in address 20 and data in address 0 and write min in address 0
"0000101010100000000000000000000" after 190 NS,--min data in address 21 and data in address 0 and write min in address 0
"0000101011000000000000000000000" after 200 NS,--min data in address 22 and data in address 0 and write min in address 0
"0000101011100000000000000000000" after 210 NS,--min data in address 23 and data in address 0 and write min in address 0
"0000101100000000000000000000000" after 220 NS,--min data in address 24 and data in address 0 and write min in address 0
"0000101100100000000000000000000" after 230 NS,--min data in address 25 and data in address 0 and write min in address 0
"0000101101000000000000000000000" after 240 NS,--min data in address 26 and data in address 0 and write min in address 0
"0000101101100000000000000000000" after 250 NS,--min data in address 27 and data in address 0 and write min in address 0
"0000101110000000000000000000000" after 260 NS,--min data in address 28 and data in address 0 and write min in address 0
"0000101110100000000000000000000" after 270 NS,--min data in address 29 and data in address 0 and write min in address 0
"0000101110100000000000000000000" after 290 NS;--min data in address 30 and data in address 0 and write min in address 0
end architecture testbenchOfsystem;
```

Figure 2.4.1:testbench code with machine code.

note: this code in Appendix D.

But because previous error we can not simulate testbench and see the result of machine code.

Conclusion:

In this project, we became better at writing codes VHDL , we learned about the components of the microprocessor , how it works, and how to organize writing and reading from RAM at the same time.

References:

[1]: Microprocessor - Overview (tutorialspoint.com). [Accessed 29 August 2021 , 16:25]
[2]: What is an arithmetic-logic unit (ALU) and how does it work? (techtarget.com) .
[Accessed 29 August 2021 , 16:30]
[3]: Register file - Wikipedia.[Accessed 29 August 2021 , 18:00]

Appendices:

Appendix A:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.NUMERIC_STD.all;
entity ALU is
Port (A, B     : in  STD_LOGIC_VECTOR(31 downto 0);  -- 2 inputs 32-bit
    ALU_Sel  : in  STD_LOGIC_VECTOR(5 downto 0);  -- 1 input 6-bit for selecting function
    ALU_Out   : out  STD_LOGIC_VECTOR(31 downto 0)); -- 1 output 32-bit
end ALU;
architecture Behavioral of ALU is
signal ALU_Result : std_logic_vector (31 downto 0);
begin
process(A,B,ALU_Sel)
begin
  case(ALU_Sel) is
  when "000100" => -- Addition
   ALU_Result <= A + B ;
  when "001010" => -- Subtraction
   ALU_Result <= A - B ;
  when "001100" => -- negate
   ALU_Result <= std_logic_vector(not(A)+ "00000001") ;
  when "000011" => -- absolute value
   ALU_Result <= std_logic_vector(abs(signed(A)));
  when "000111" => -- max
   if (A>B)THEN
  ALU_Result <= A;
   Else
  ALU_Result <= B;
   END IF ;
  when "000010" => --  MIN
   if (A<B)THEN
  ALU_Result <= A;
   Else
  ALU_Result <= B;
   END IF ;
  when "000110" => -- avg
   ALU_Result <= std_logic_vector(to_unsigned(to_integer(unsigned(A)+unsigned(B)) / 2,32)) ;
  when "001011" => -- Logical and
   ALU_Result <= A and B;
  when "001110" => -- Logical or
   ALU_Result <= A or B;
  when "001000" => -- Logical xor
  ALU_Result <= A xor B;
  when "001101" => -- not
   ALU_Result <= not A ;
  when others =>
  null;
  end case;
 end process;
 ALU_Out <= ALU_Result; -- ALU out
end Behavioral;
```

Appendix B:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity RAM is
port (en, clk : in std_logic ;
address1 : in std_logic_vector(0 to 4); -- 5-bit address bus
address2 : in std_logic_vector(0 to 4); -- 5-bit address bus
address3 : in std_logic_vector(0 to 4); -- 5-bit address bus
input: in std_logic_vector(31 downto 0); -- 32-bit data_out bus
output1: out std_logic_vector(31 downto 0); -- 32-bit data_out bus
output2: out std_logic_vector(31 downto 0)); -- 32-bit data_out bus

end entity RAM;
architecture RAM_behavioral of RAM is
Type RAM_array is array (0 to 31) of std_logic_vector(31 downto 0);
signal RAM_data_array : RAM_array:= (
--filling values of register file in binary

0 => "00000000000000000000000000000000",
1 => "00000000000000000001001000001000" ,
2 => "00000000000000000001011010111000",
3 => "00000000000000000001010111110110",
4 => "00000000000000000001101010000010",
5 => "00000000000000000001101010000000",
6 => "00000000000000000011000010010000",
7 => "00000000000000000011010011101100",
8 => "00000000000000000011010010010110",
9 => "00000000000000000011010010001110",
10 => "00000000000000000010111000000100" ,
11 => "00000000000000000011001101110010" ,
12 => "00000000000000000000101110100110" ,
13 => "00000000000000000001111110100000" ,
14 => "00000000000000000000001000000010" ,
15 => "00000000000000000000111000010000" ,
16 => "00000000000000000010101001110110" ,
17 => "00000000000000000011000011110000" ,
18 => "00000000000000000010011010000100" ,
19 => "00000000000000000001100000010110" ,
20 => "00000000000000000001000110101000" ,
21 => "00000000000000000011100001100100" ,
22 => "00000000000000000010111101101000" ,
23 => "00000000000000000001010000001110" ,
24 => "00000000000000000010111010110110" ,
25 => "00000000000000000001111000001000" ,
26 => "00000000000000000001010010001010" ,
27 => "00000000000000000011000010000100" ,
28 => "00000000000000000000110111101000" ,
29 => "00000000000000000000010011100000" ,
30 => "00000000000000000010001000010100" ,
31 => "00000000000000000000000000000000" );
BEGIN
process(clk)
begin
if (rising_edge(CLK)) then -- Clocking the register file
if (en='1')then
output1 <= RAM_data_array(conv_integer(address1)); --read data in address1
output2 <= RAM_data_array(conv_integer(address2)); --read data in address2
RAM_data_array(conv_integer(address3)) <= input;   --write input from ALU  in address3
end if;
end if;
end process;
end Architecture RAM_behavioral;
```

Appendix C:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.NUMERIC_STD.all;
entity system is
port( clk : in std_logic;
register32: in std_logic_vector(31 downto 0));
end system;
architecture struct of system is
signal opcode :std_logic_vector(5 downto 0);
signal address1 :std_logic_vector(4 downto 0);
signal address2 :std_logic_vector(4 downto 0);
signal address3 :std_logic_vector(4 downto 0);
signal out1 :std_logic_vector(4 downto 0);
signal out2 :std_logic_vector(4 downto 0);
signal input :std_logic_vector(4 downto 0);
signal en1:std_logic ;
begin
process(clk)
begin
if (rising_edge(CLK)) then
-- make enable= 1 if opcode is valid
if (opcode ="000100"
or opcode= "001010"
or opcode="001100"
or opcode ="000011"
or opcode="000111"
or opcode="000010"
or opcode="000110"
or opcode="001011"
or opcode="001110"
or opcode="001000"
or opcode="001101") then
en1<= '1';
opcode <= register32(31 downto 26);--take first 6 bit for op code
address1 <= register32(25 downto 21);--take second 5 bit for address1
address2 <= register32(20 downto 16) ;--take third 5 bits for address2
address3 <= register32(15 downto 11); --take fourth 5 bits for address3
```

```
ram: entity work.RAM(RAM_behavioral) port map (address1,address2,address3,out1,out2);
ALU: entity work.ALU(Behavioral) port map (en1 ,out1,out2,opcode,result);
end if;
end if;
end process;
end;
```

Appendix D:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity testbenchOfsystem is
end entity testbenchOfsystem;
architecture testbenchOfsystem of testbenchOfsystem is
signal machineinstruction: std_logic_vector( 31 downto
0):="00000000000000000000000000000000";
signal clk: std_logic:='0';
begin
s1: Entity work.system(struct)
port map (clk, machineinstruction);
clk <= NOT clk after 10 NS;
machineinstruction <= "00001000001000100000000000000000",--min data in address 1 and data
in address 2 and write min in address 0
"00001000011000000000000000000000" after 10 NS,--min data in address 3 and data in address
0 and write min in address 0
"00001000011000000000000000000000" after 20 NS,--min data in address 4 and data in address
0 and write min in address 0
"00001000101000000000000000000000" after 30 NS,--min data in address 5 and data in address
0 and write min in address 0
"00001000110000000000000000000000" after 40 NS,--min data in address 6 and data in address
0 and write min in address 0
"00001000111000000000000000000000" after 50 NS,--min data in address 7 and data in address
0 and write min in address 0
"00001001000000000000000000000000" after 60 NS,--min data in address 8 and data in address
0 and write min in address 0
"00001001001000000000000000000000" after 70 NS,--min data in address 9 and data in address
0 and write min in address 0
"00001001010000000000000000000000" after 80 NS,--min data in address 10 and data in address
0 and write min in address 0
"00001001011000000000000000000000" after 90 NS,--min data in address 11 and data in address
0 and write min in address 0
"00001001100000000000000000000000" after 100 NS,--min data in address 12 and data in
address 0 and write min in address 0
"00001001101000000000000000000000" after 110 NS,--min data in address 13 and data in
address 0 and write min in address 0
"00001001110000000000000000000000" after 120 NS,--min data in address 14 and data in
address 0 and write min in address 0
```

```
"00001001111000000000000000000000" after 130 NS,--min data in address 15 and data in address 0 and write min in address 0
"00001010000000000000000000000000" after 140 NS,--min data in address 16 and data in address 0 and write min in address 0
"00001010001000000000000000000000" after 150 NS,--min data in address 17 and data in address 0 and write min in address 0
"00001010010000000000000000000000" after 160 NS,--min data in address 18 and data in address 0 and write min in address 0
"00001010011000000000000000000000" after 170 NS,--min data in address 19 and data in address 0 and write min in address 0
"00001010100000000000000000000000" after 180 NS,--min data in address 20 and data in address 0 and write min in address 0
"00001010101000000000000000000000" after 190 NS,--min data in address 21 and data in address 0 and write min in address 0
"00001010110000000000000000000000" after 200 NS,--min data in address 22 and data in address 0 and write min in address 0
"00001010111000000000000000000000" after 210 NS,--min data in address 23 and data in address 0 and write min in address 0
"00001011000000000000000000000000" after 220 NS,--min data in address 24 and data in address 0 and write min in address 0
"00001011001000000000000000000000" after 230 NS,--min data in address 25 and data in address 0 and write min in address 0
"00001011010000000000000000000000" after 240 NS,--min data in address 26 and data in address 0 and write min in address 0
"00001011011000000000000000000000" after 250 NS,--min data in address 27 and data in address 0 and write min in address 0
"00001011100000000000000000000000" after 260 NS,--min data in address 28 and data in address 0 and write min in address 0
"00001011101000000000000000000000" after 270 NS,--min data in address 29 and data in address 0 and write min in address 0
"00001011101000000000000000000000" after 290 NS;--min data in address 30 and data in address 0 and write min in address 0
end architecture testbenchOfsystem;
```