Faculty of Engineering & Technology

Department of Electrical and Computer Engineering

Summer Semester 2020/2021

ENC3310– Advanced Digital Design

Course Project

Made By: Nour Naji -1190270.

Instructor: Dr. Abdullatif Abuissa.

Section: 2.

TABLE OF CONTENTS

Table of figures

Table of codes

Table of simulations

INTRODUCTION

The aim of this project is to design a simple part of a microprocessor (ALU and the register file), then validate all possible cases by writing a verification-Test bench- code.

VHDL

VHDL (VHSIC Hardware Description Language) is an efficient programming language designed to describe the behavior of digital systems and circuits at multiple levels of abstraction, from system level to logical gate level, for design entry, documentation, and verification goals.

ALU

An Arithmetic Logic Unit (ALU) is a digital circuit used to perform arithmetic and logical operations. It is the most important component of the system and is used in many devices such as calculators, mobile phones, and computers, and is the basic building block of a computer's central processing unit (CPU). The base ALU contains three parallel data buses consisting of two input operators (A and B) and the resultant output (Result), and input (opcode) it's a parallel bus that conveys to the ALU an operation selection code, which is an enumerated value that specifies the desired arithmetic or logic operation to be performed by the ALU. Each data bus is a set of signals that transmit a binary number. The widths of A, B and Y buses (the number of signals that each bus comprises of) are usually identical and match the original word size of the external circuits.

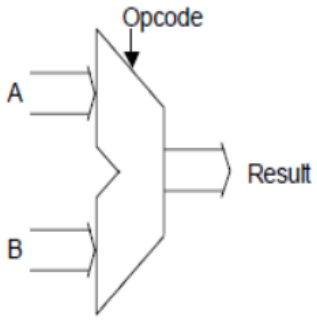


Figure 1(An Arithmetic Logic Unit (ALU))

The register file

Computer random access memory (RAM) is one of the most important components in determining your system's performance. RAM gives applications a place to store and access data on a short-term basis. It stores the information your computer is actively using so that it can be accessed quickly.

Figure 2(Computer Memory RAM-1)

RAM allows your computer to perform many of its everyday tasks, such as loading applications, browsing the internet, editing a spreadsheet, or experiencing the latest game. Memory also allows you to switch quickly among these tasks, remembering where you are in one task when you switch to another task. As a rule, the more memory you have, the better.

This programmable logic RAM is a simple, single port memory component that outputs data from the specified memory address and, if a write enable is asserted, writes input data to this address.

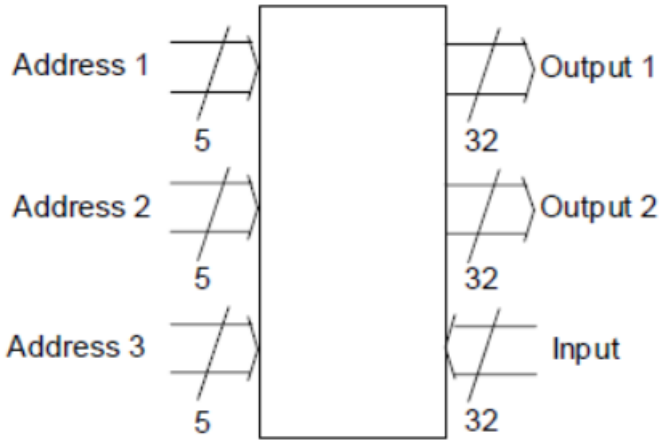


Figure 3(Computer Memory RAM-2)

PHILOSOPHY OF THE DESIGN

To design any piece, we should determine the inputs and outputs. In addition to determining the generic numbers which represent bits size for input and output.

- ***Stage 1***

As shown in *Figure 1*, to implement the ALU, we need two inputs A, B, each one is 32 bits, and 6-bit opcode, and we have to create a few variables.
In case of output Result (32 bits) with Opcode =>

- Add (A+B) when Opcode = 001000
- Add (A- B) when Opcode = 001001
- |A| when Opcode = 000010
- -A when Opcode = 001010
- max (A, B) when Opcode = 001100
- min (A, B) when Opcode = 000001
- avg (A, B) when Opcode = 001101
- not A when Opcode = 000101
- A or B when Opcode = 000100
- A and B when Opcode = 001011
- A or B when Opcode = 001111

variable that were created:

- **A_int =>** I convert A from std_logic_vector to an integer by using conv_integer and store the output in A_int**.**
- **A_int =>** I convert B from std_logic_vector to an integer by using conv_integer and store the output in B_int**.**
- **Avg =>** I calculated the average between a and b and stored it in Avg.

First, I followed the cases method to make the Alu, which is to check the opcode input and compare it with the existing conditions to choose which process it will perform whether adding or subtracting, etc.

We enter two inputs and a specific Opcode value. The Alu checks the Opcode value and compares it with the conditions listed in the code, If the Opcode value is "001000", it does the addition, but if it is "001001", it subtracts, and so on. But if Opcode is "001101", he performs the process of average, where he converts the value of A to an integer and B to an integer by using conv_integer, then the output of the average comes by adding A and B, and dividing them by 2, and then storing the average value in the Result, but Result are vector, then we convert the value of the average from an integer to Vector by using conv_std_logic_vector.

- ***Stage 2***

As shown in *Figure 3*, to implement the register file, we need six inputs, Address1, Address2, Address3, enable, input and clock (to control the verification process), and we have two output output1, output2, and we have to create array type to store the initial values in the register file.

To Design File that implements a 32x32-bit simple dual-port RAM with separate read and write addresses, First, I stored the values in the register file (memory),-(x"00000000",x"00003ABA",x"00002296",x"000000AA",x"00001C3A",x"00001180",x"000022E0",x"00001C86",x"000022DA",x"00000414",x"00001A32",x"00000102",x"00001CBA",x"00000CDE",x"00003994",x"00001984",x"000028C4",x"00002E7C",x"00003966",x"0000227E",x"00002208",x"000011B4",x"0000237C",x"0000360E",x"00002722",x"00000500",x"000016B6",x"0000029E",x"00002280",x"00002B52", x"000011A0",x"00000000")- Then I checked whether the clock is Rising or not. If the condition was verification, I put another conditional sentence to check if the enable was 1 or not, If 1, then, doing the following:

This memory component outputs data from the memory address specified and also writes input data to this address, in another word, it stores the value of the Address1 in the output1, stores the value of the Address2 in the output2, and it writes the value of the input in Address3.

Now, how we can check if the value of the enable is 1 or 0? I wrote a code -It consists of one input and one output- that checks if the entered Opcode is equal to any Opcode It is included in the conditions listed, then makes the value of the enable 1.

- ***Stage 3***

right Now, to design BIT register, we need two input Machine instructions (32 bit) and clock (to control the verification process), and four output Address1, Address2, Address3 (5 bit) and opcode (6 bit).

The principle of operation of the register is:

- 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

Then , The values stored in bit 31 to bit 26 from Machine instructions are put into bit number 5 to 0 from opcode , and The values stored in bit 25 to bit 21 from Machine instructions are put into bit number 4 to 0 from Address1 , and The values stored in bit 20 to bit 16 from Machine instructions are put into bit number 4 to 0 from

Address2,and the values stored in bit 15 to bit 11 from Machine instructions are put into bit number 4 to 0 from Address3 , and final 11 bits are unused.

- ***Stage4***

Finally, to design the microprocessor by using ALU and register file**,** we need two input , instructions 32 bit and clock , and we have 8 output , and they are like this : 5 bit Address1, Address2 ,Address3 ,and 6 bit of opcuode , enable , and 32 bit of ram_out1,ram_out2 ,Result .The circuit is installed according to the *Figure 4 (*microprocessor*)*

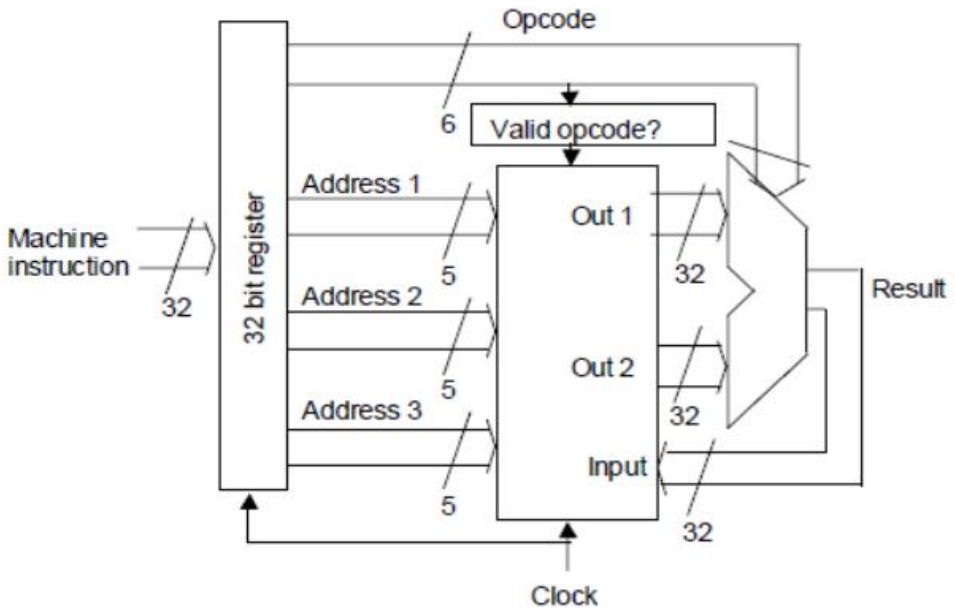


Figure4 (microprocessor)

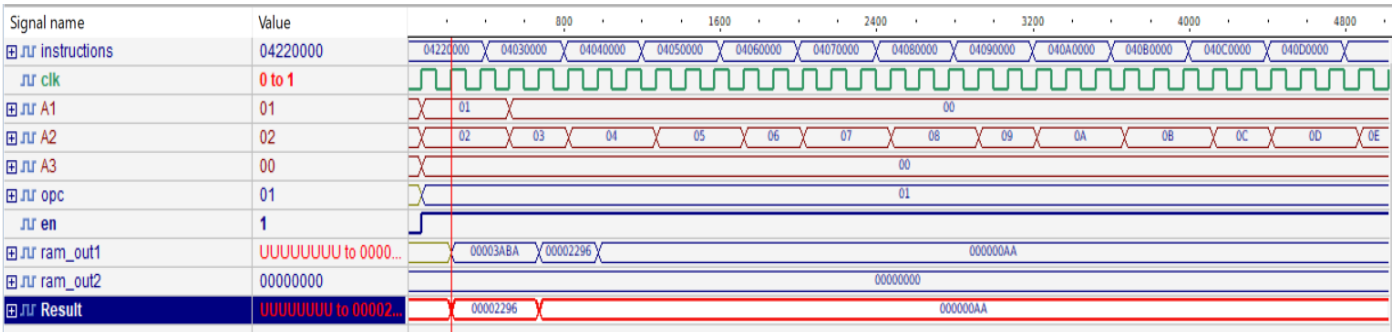
- ***Stage 5***

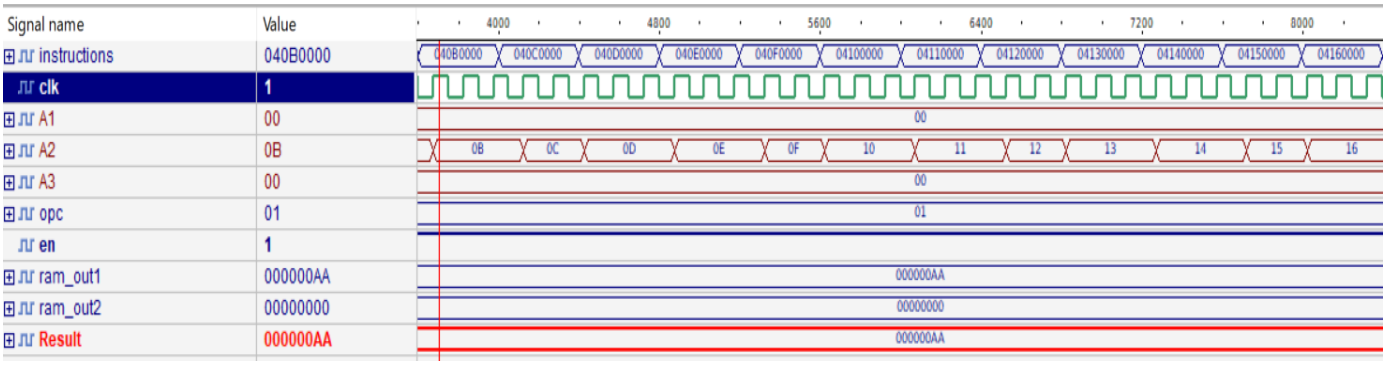
To make sure the codes are working correctly, I designed a clocked test bench called test bench , The aim of this part is to find the smallest number stored in the memory , This is done by comparing the stored value in the Address one with the stored value in the Address two and setting the result in Address zero , And then compare the value stored in the Address zero with the value stored in the third Address and also put the result in Address zero ,Then compare the value stored at address zero with the value stored at address four and also put the result at address zero , And so on until all values are compared with each other, and put the final result in Address zero .

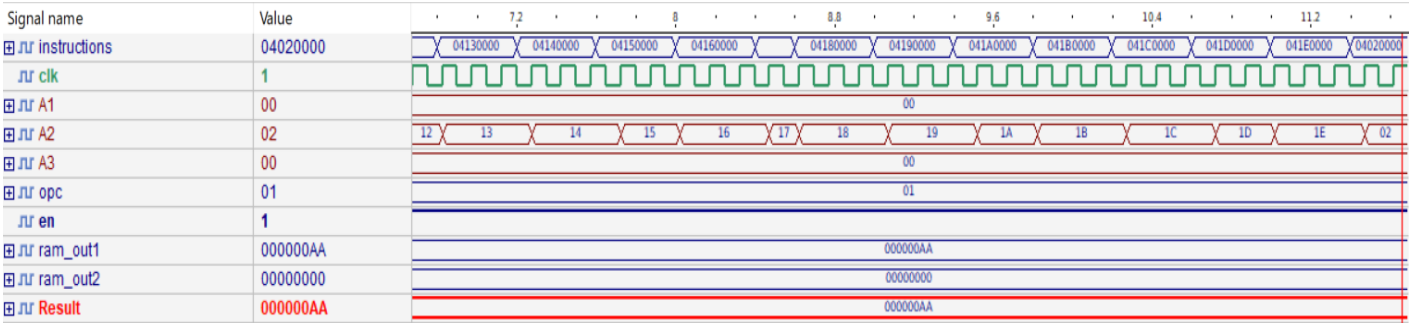
According to the values stored in the memory, we notice that the smallest value is x"000000AA ", but will the same result appear on the Simulation?

SIMULATION RESULTS

TEST BENCH







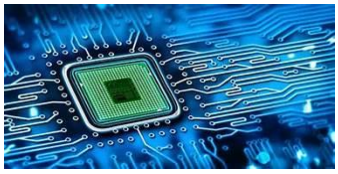
Simulation (Test bench)

We note that the end result is identical to the actual value that should appear, but how did this happen?

When entering the value of the instructions -**00000100001000100000000000000000**-, it is divided as follows:

- Opcode = 000001 ➔ This means finding the minimum value
- Address1 = 00001 → The first Address contains value x"00003ABA".
- Address2 = 00010 → The second Address contains value x"00002296".
- Address3 = 00000 → The second Address contains value x"00000000".
- The opcode is checked if it is present within the conditions or not, and since it is present, the value of the enable becomes 1.
- The value in the Address one is placed on the output1.
- The value in the Address two is placed on the output2.
- The value of the output1and output2 and the opcode is entered on the ALU.
- Since the opcode chooses the minimum value, and the value in the Address two is smaller than the first, then the result of the ALU is x"00002296", This value is entered into the memory and written in Address zero , Then the second instruction, and the result comes out with the same steps, until he enters the last one.

CONCLUSION AND FUTURE WORK



In this project I've learned how to design a microprocessor using the VHDL programming language, also I've learned to test the code by simulating it. Finally, I've learned how to build a clocked test bench to test the code.

I also learned that the microprocessor follows a sequence: fetch, decode, and execute. Initially, the instructions are stored in memory in sequential order. The microprocessor fetches these instructions from memory, then decodes them and executes these instructions until a STOP instruction is reached. Later, it sends the output in binary to the output port. Between these operations, the register caches data and the ALU performs computing functions.

In the end, every project encountered some difficulties and needs improvement. Some of the problems I faced:

1- The lack of resources for VHDL on the Internet, and thus the difficulty of solving the problems I encountered.

2- The statement in the structural style is concurrent, not sequential. Which we aren't used to seeing in other programming languages.

3- I noticed that the results started showing up after a full cycle had passed, as it was getting late, so I put a wait statement to solve this problem.

The proposed improvement is to use the clock in both stages, as I could not use it because of several problems I encountered, including the need for a process, which means that the project will be behavioral and not structural as we must do.

"The great aim of education is not knowledge but action."
— Herbert Spencer

REFERNCES

- https://www.crucial.com/articles/about-memory/support-what-does-computer-memory-do
- https://en.wikipedia.org/wiki/VHDL
- https://www.tutorialspoint.com/microprocessor/microprocessor_overview.htm

APPENDIX

The Screenshots of Code

- **The ALU**

```
--**************************************************************************************************************
--                                                    ALU
--**************************************************************************************************************
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;
use IEEE.std_logic_arith.all;

entity alu is
port ( A, B: in std_logic_vector (31 downto 0);
opcode: in std_logic_vector(5 downto 0);
Result: out std_logic_vector (31 downto 0));
end entity alu;

architecture dataopcode of alu is
begin
     process(A,B,opcode)
     variable A_int , B_int ,avg: integer ;
     begin
          case opcode is
               when "001000" => Result <= A + B;
               when "001001" => Result <= A - B;
               when "000010" => Result <= abs(A);
               when "001010" => Result <= -A;
               when "001100" => if (A > B) then  result <= A;
                   elsif(A < B) then result <= B;
                    else NULL ;
                    end if ;
               when "000001" => if (A > B) then  result <= B;
                   elsif(A < B) then result <= A;
                    else NULL ;
                    end if ;
               when "001101" =>
               A_int := conv_integer(signed(A));
               B_int := conv_integer(signed(B));
               avg   := (A_int + B_int)/2;
               Result <= conv_std_logic_vector(avg, 32);
               when "000101" => Result <= not (A);
               when "000100" => Result <= A or B;
               when "001011" => Result <= A and B;
               when "001111" => Result <= A xor B;
              when others => NULL;
end case;
end process;
end architecture dataopcode;
```

code 1(ALU – stage1)

- **RAM**

```
--**********************************************************************************************************
--                                                      RAM
--**********************************************************************************************************
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity RAM32x32 is
   port ( Address1, Address2 ,Address3: in std_logic_vector (4 downto 0);
   enable : in std_logic;
   input : in  std_logic_vector (31 downto 0);
   clk : in  std_logic ;
   output1, output2 : out std_logic_vector (31 downto 0));
end entity RAM32x32;

architecture dataflow of RAM32x32 is
type rom_array is array (0 to 31) of std_logic_vector (31 downto 0);
signal rom_data: rom_array := (x"00000000", x"00003ABA", x"00002296",x"000000AA",
x"00001C3A",x"00001180", x"000022E0",x"00001C86",x"000022DA",x"00000414",x"00001A32",
x"00000102", x"00001CBA",x"00000CDE", x"00003994",x"00001984", x"000028C4",
x"00002E7C", x"00003966",x"0000227E", x"00002208",x"000011B4", x"0000237C",
x"0000360E",x"00002722", x"00000500", x"000016B6",x"0000029E",x"00002280",
x"00002B52", x"000011A0",x"00000000");
begin
    process(clk)
begin
    if(rising_edge(clk)) then
    if(enable = '1') then
        output1 <= rom_data(conv_integer(Address1));
        output2 <= rom_data(conv_integer(Address2));
        rom_data(conv_integer(Address3)) <= input;
end if;
end if;
end process;
end architecture dataflow;
```

code 2(Ram – Stage2)

- **Enable**

```
--*********************************************************************************************************
--                                                    Enable
--*********************************************************************************************************
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_signed.all;

entity my_Enable is
    port(opcode : in std_logic_vector(5 downto 0);
         enable : out std_logic );
end my_Enable;

architecture arch of my_Enable is
begin
    process(opcode)
    begin
        case opcode is
            when "001000" => enable <= '1';
            when "001001" => enable <= '1';
            when "000010" => enable <= '1';
            when "001010" => enable <= '1';
            when "001100" => enable <= '1';
            when "000001" => enable <= '1';
            when "001101" => enable <= '1';
            when "000101" => enable <= '1';
            when "000100" => enable <= '1';
            when "001011" => enable <= '1';
            when "001111" => enable <= '1';
            when others   => enable <= '0';
        end case;
    end process;
end arch;
```

code 3(Enable)

- **32 BIT register**

```
--*************************************************************************************************************
--                                                  32 BIT register
--*************************************************************************************************************
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity bit_register is
port (Machine_instructions :in std_logic_vector (31 downto 0);
clk : in  std_logic ;
Address1, Address2 ,Address3: out std_logic_vector (4 downto 0);
opcode: out std_logic_vector(5 downto 0));
end entity bit_register;

architecture data of bit_register is
begin
    process(clk)
begin
    if(rising_edge(clk)) then
    opcode  (5 downto 0) <= Machine_instructions(31 downto 26) ;   --6 bit
    Address1(4 downto 0) <= Machine_instructions(25 downto 21) ;   --5 bit
    Address2(4 downto 0) <= Machine_instructions(20 downto 16) ;   --5 bit
    Address3(4 downto 0) <= Machine_instructions(15 downto 11 );   --5 bit
end if;
end process;
end architecture data;
```

code 4(32 BIT register – Stage3)

- **design**

```
--*************************************************************************************************************
--                                                  design
--*************************************************************************************************************
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;


entity project is
port (instructions :in std_logic_vector (31 downto 0);
clk : in  std_logic;
r ,out1 ,out2: out std_logic_vector (31 downto 0);
A11 ,A22 ,A33 : out std_logic_vector (4 downto 0);
opcuode : out std_logic_vector(5 downto 0);
enable :out std_logic );
end entity project;

architecture design of project is
signal A1 ,A2 ,A3 :std_logic_vector (4 downto 0);
signal opc :std_logic_vector(5 downto 0);
signal en :std_logic;
signal ram_out1,ram_out2 ,Result : std_logic_vector (31 downto 0);
begin

    r    <= Result;
    A11  <= A1;
    A22  <= A2;
    A33  <= A3;
    out1 <= ram_out1;
    out2 <= ram_out1;
    opcuode <= opc;
    enable <= en;

    enablee : entity work.my_Enable(arch) port map(opc,en);
    reg : entity work.bit_register(data) port map (instructions,clk,A1,A2,A3,opc);
    alu : entity work.alu(dataopcode)    port map(ram_out1,ram_out2,opc,Result);
    ram : entity work.RAM32x32(dataflow) port map(A1,A2,A3,en,Result,clk,ram_out1,ram_out2);
end architecture design;
```

code 5(design-Stage4)

- **test bench**

```
--**************************************************************************************************************
--                                                test bench
--**************************************************************************************************************
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity alu_test is
end entity alu_test;

architecture test of alu_test is

signal instructions : std_logic_vector (31 downto 0);
signal clk : std_logic := '0';
signal A1 ,A2 ,A3 :std_logic_vector (4 downto 0) := (others => '0');
signal opc :std_logic_vector(5 downto 0) := (others => '0');
signal en :std_logic := '0';
signal ram_out1,ram_out2 ,Result : std_logic_vector (31 downto 0) := (others => '0');
begin
    system : entity work.project(design) port map(instructions,clk,Result,ram_out1,ram_out1,A1,A2,A3,opc,en);
    clk <= not clk after 75 ns;
    instructions <=
    "00000100001000100000000000000000" ,
    "00000100000000110000000000000000"  after 400  ns ,
    "00000100000001000000000000000000"  after 800  ns ,
    "00000100000001010000000000000000"  after 1200 ns ,
    "00000100000001100000000000000000"  after 1600 ns ,
    "00000100000001110000000000000000"  after 2000 ns ,
    "00000100000010000000000000000000"  after 2400 ns ,
    "00000100000010010000000000000000"  after 2800 ns ,
    "00000100000010100000000000000000"  after 3200 ns ,
    "00000100000010110000000000000000"  after 3600 ns ,
    "00000100000011000000000000000000"  after 4000 ns ,
    "00000100000011010000000000000000"  after 4400 ns ,
    "00000100000011100000000000000000"  after 4800 ns ,
    "00000100000011110000000000000000"  after 5200 ns ,
    "00000100000100000000000000000000"  after 5600 ns ,
    "00000100000100010000000000000000"  after 6000 ns ,
    "00000100000100100000000000000000"  after 6400 ns ,
    "00000100000100110000000000000000"  after 6800 ns ,
    "00000100000101000000000000000000"  after 7200 ns ,
    "00000100000101010000000000000000"  after 7600 ns ,
    "00000100000101100000000000000000"  after 8000 ns ,
    "00000100000101110000000000000000"  after 8400 ns ,
    "00000100000110000000000000000000"  after 8600 ns ,
    "00000100000110010000000000000000"  after 9000 ns ,
    "00000100000110100000000000000000"  after 9400 ns ,
    "00000100000110110000000000000000"  after 9800 ns ,
    "00000100000111000000000000000000"  after 10200 ns,
    "00000100000111010000000000000000"  after 10600 ns,
    "00000100000111100000000000000000"  after 11000 ns,
    "00000100000000010000000000000000"  after 11400 ns;

end architecture test;
```

code 6(Test bench – Stage5)

THE WRITTEN CODE

```
--**************************************************************************************************************
--                                                                ALU
--**************************************************************************************************************
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;
use IEEE.std_logic_arith.all;

entity alu is
port ( A, B: in std_logic_vector (31 downto 0);
opcode: in std_logic_vector(5 downto 0);
Result: out std_logic_vector (31 downto 0));
end entity alu;
architecture dataopcode of alu is
begin
process(A,B,opcode)
variable A_int , B_int ,avg: integer ;
begin
case opcode is
when "001000" => Result <= A + B;
when "001001" => Result <= A - B;
when "000010" => Result <= abs(A);
when "001010" => Result <= -A;
when "001100" => if (A > B) then  result <= A;
elsif(A < B) then result <= B;
else NULL ;
end if ;
when "000001" => if (A > B) then  result <= B;
elsif(A < B) then result <= A;
else NULL ;
end if ;
when "001101" =>
A_int := conv_integer(signed(A));
```

```
B_int := conv_integer(signed(B));
avg   := (A_int + B_int)/2;
Result <= conv_std_logic_vector(avg, 32);
when "000101" => Result <= not (A);
when "000100" => Result <= A or B;
when "001011" => Result <= A and B;
when "001111" => Result <= A xor B;
when others => NULL;
end case;
end process;
end architecture dataopcode;
--**************************************************************************************************************
--                                                    RAM
--**************************************************************************************************************
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity RAM32x32 is
port ( Address1, Address2 ,Address3: in std_logic_vector (4 downto 0);
enable : in std_logic;
input : in  std_logic_vector (31 downto 0);
clk : in  std_logic ;
output1, output2 : out std_logic_vector (31 downto 0));
end entity RAM32x32;

architecture dataflow of RAM32x32 is
type rom_array is array (0 to 31) of std_logic_vector (31 downto 0);
signal rom_data: rom_array := (x"00000000", x"00003ABA", x"00002296",x"000000AA",
x"00001C3A",x"00001180", x"000022E0",x"00001C86",x"000022DA",x"00000414",x"00001A32",
x"00000102", x"00001CBA",x"00000CDE", x"00003994",x"00001984", x"000028C4", x"00002E7C",
x"00003966",x"0000227E", x"00002208",x"000011B4", x"0000237C", x"0000360E",x"00002722",
x"00000500", x"000016B6",x"0000029E",x"00002280", x"00002B52", x"000011A0",x"00000000");
begin
process(clk)
```

```
begin
if(rising_edge(clk)) then
if(enable = '1') then
output1 <= rom_data(conv_integer(Address1));
output2 <= rom_data(conv_integer(Address2));
rom_data(conv_integer(Address3)) <= input;
end if;
end if;
end process;
end architecture dataflow;
--**************************************************************************************************************
--                                                    Enable
--**************************************************************************************************************
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_signed.all;

entity my_Enable is
port(opcode : in std_logic_vector(5 downto 0);
enable : out std_logic );
end my_Enable;

architecture arch of my_Enable is
begin
process(opcode)
begin
case opcode is
when "001000" => enable <= '1';
when "001001" => enable <= '1';
when "000010" => enable <= '1';
when "001010" => enable <= '1';
when "001100" => enable <= '1';
when "000001" => enable <= '1';
```

```
when "001101" => enable <= '1';
when "000101" => enable <= '1';
when "000100" => enable <= '1';
when "001011" => enable <= '1';
when "001111" => enable <= '1';
when others   => enable <= '0';
end case;
end process;
end arch;
--**************************************************************************************************************
--                                                32 BIT register
--**************************************************************************************************************
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity bit_register is
port (Machine_instructions :in std_logic_vector (31 downto 0);
clk : in  std_logic ;
Address1, Address2 ,Address3: out std_logic_vector (4 downto 0);
opcode: out std_logic_vector(5 downto 0));
end entity bit_register;

architecture data of bit_register is
begin
process(clk)
begin
if(rising_edge(clk)) then
opcode  (5 downto 0) <= Machine_instructions(31 downto 26) ;   --6 bit
Address1(4 downto 0) <= Machine_instructions(25 downto 21) ;   --5 bit
Address2(4 downto 0) <= Machine_instructions(20 downto 16) ;   --5 bit
Address3(4 downto 0) <= Machine_instructions(15 downto 11 );   --5 bit
end if;
end process;
```

```
end architecture data;
--**************************************************************************************************************
--                                                  design
--**************************************************************************************************************
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity project is
port (instructions :in std_logic_vector (31 downto 0);
clk : in  std_logic;
r ,out1 ,out2: out std_logic_vector (31 downto 0);
A11 ,A22 ,A33 : out std_logic_vector (4 downto 0);
opcuode : out std_logic_vector(5 downto 0);
enable :out std_logic );
end entity project;

architecture design of project is
signal A1 ,A2 ,A3 :std_logic_vector (4 downto 0);
signal opc :std_logic_vector(5 downto 0);
signal en :std_logic;
signal ram_out1,ram_out2 ,Result : std_logic_vector (31 downto 0);
begin
r    <= Result;
A11  <= A1;
A22  <= A2;
A33  <= A3;
out1 <= ram_out1;
out2 <= ram_out1;
opcuode <= opc;
enable <= en;

enablee : entity work.my_Enable(arch) port map(opc,en);
reg : entity work.bit_register(data) port map (instructions,clk,A1,A2,A3,opc);
```

```
alu : entity work.alu(dataopcode)     port map(ram_out1,ram_out2,opc,Result);
ram : entity work.RAM32x32(dataflow) port map(A1,A2,A3,en,Result,clk,ram_out1,ram_out2);
end architecture design;
--**************************************************************************************************************
--                                                    test bench
--**************************************************************************************************************
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity alu_test is
end entity alu_test;

architecture test of alu_test is
signal instructions : std_logic_vector (31 downto 0);
signal clk : std_logic := '0';
signal A1 ,A2 ,A3 :std_logic_vector (4 downto 0) := (others => '0');
signal opc :std_logic_vector(5 downto 0) := (others => '0');
signal en :std_logic := '0';
signal ram_out1,ram_out2 ,Result : std_logic_vector (31 downto 0) := (others => '0');
begin
system : entity work.project(design) port
map(instructions,clk,Result,ram_out1,ram_out1,A1,A2,A3,opc,en);
clk <= not clk after 75 ns;
instructions <=
"00000100001000100000000000000000" ,
"00000100000000110000000000000000"  after 400  ns ,
"00000100000001000000000000000000"  after 800  ns ,
"00000100000001010000000000000000"  after 1200 ns ,
"00000100000001100000000000000000"  after 1600 ns ,
"00000100000001110000000000000000"  after 2000 ns ,
"00000100000010000000000000000000"  after 2400 ns ,
"00000100000010010000000000000000"  after 2800 ns ,
"00000100000010100000000000000000"  after 3200 ns ,
```

"00000100000010110000000000000000" after 3600 ns ,

"00000100000011000000000000000000" after 4000 ns ,

"00000100000011010000000000000000" after 4400 ns ,

"00000100000011100000000000000000" after 4800 ns ,

"00000100000011110000000000000000" after 5200 ns ,

"00000100000100000000000000000000" after 5600 ns ,

"00000100000100010000000000000000" after 6000 ns ,

"00000100000100100000000000000000" after 6400 ns ,

"00000100000100110000000000000000" after 6800 ns ,

"00000100000101000000000000000000" after 7200 ns ,

"00000100000101010000000000000000" after 7600 ns ,

"00000100000101100000000000000000" after 8000 ns ,

"00000100000101110000000000000000" after 8400 ns ,

"00000100000110000000000000000000" after 8600 ns ,

"00000100000110010000000000000000" after 9000 ns ,

"00000100000110100000000000000000" after 9400 ns ,

"00000100000110110000000000000000" after 9800 ns ,

"00000100000111000000000000000000" after 10200 ns,

"00000100000111010000000000000000" after 10600 ns,

"00000100000111100000000000000000" after 11000 ns,

"00000100000000010000000000000000" after 11400 ns;

end architecture test;