Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

COMPUTER ARCHITECTURE

Project 2

Prepared by:

Rahaf Abushamma 1180010

Rahaf Baker 1180588

Saja Aljamal 1181005

Instructor:

Dr. Ayman Hroub

Abstract:

The objective of this project is to successfully design, model and simulate a simple Multi-Cycle Processor using Verilog HDL. The button-up design approach was used where each sub-module of the processor was first designed, coded, and tested. Once all sub-modules were designed and determined to be fully functional, they were instantiated into a structural module to form the MIPS processor. The processor was then tested by executing a set of MIPS instructions while verifying proper functionality and timing. Objective The objective of this project is to successfully design, model and simulate a MIPS Multi-Cycle Processor using Verilog HDL. The button-up design approach was used where each sub-module of the processor was first designed, coded, and tested. Once all sub-modules were designed and determined to be fully functional, they were instantiated into a structural module to form the MIPS processor. The processor was then tested by executing a set of MIPS instructions while verifying proper functionality and timing.

Contents

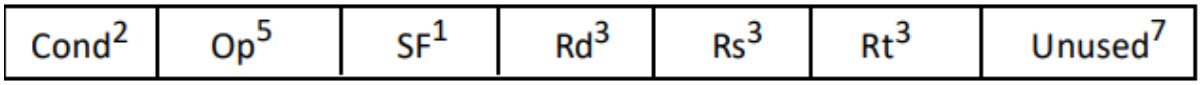
Table of figure:

Theory

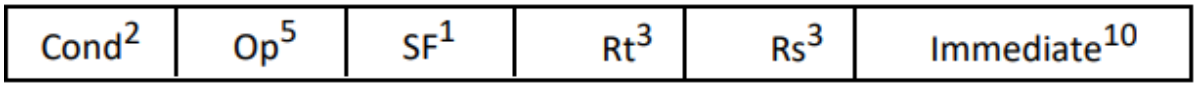
MIPS INSTRUCTIONS:

The project's processor design is based on the MIPS Reduced Instruction Set Computer (RISC) architecture and includes a subset of the MIPS Instruction set. MIPS Instructions are 24- bits wide and use one of the three following instruction types:

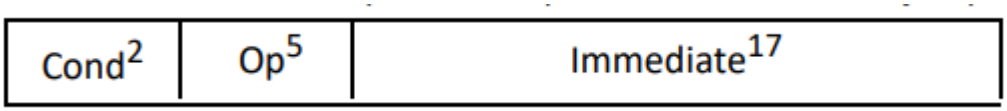
R-Type



I-Type



J-Type



J-Type Instructions:

A. jump – j instruction

00	01100	$Immediate^{17}$

Operation: PC <- PC[23:20] || Inst[16:0] || 00
Number of Cycles: 3

B. jump and link – jal instruction

00	01101	$Immediate^{17}$

Operation: $R7<- PC + 1
PC<- PC[23:20] || Inst[16:0] || 00
Number of Cycles: 3

C. Load upper immediate – lui instruction

00	01110	$Immediate^{17}$

Operation: $R1<- 0000000|| Inst[16:0]
PC<- PC+1
Number of Cycles: 5

R-Type Instructions:

A. Addition – add instruction:

1. where condition:00 so the current instruction will execute

00	00011	0	Rd^3	Rs^3	Rt^3	$UNUSED^7$

Operation: Rd<- Rs + Rt
PC <- PC + 1
Number of Cycles: 4

2. where condition:01 Execute if equal, in other words, if the zero-flag bit is set. Otherwise, the current instruction can be treated as a NOP (no operation).

01	00011	X	Rd^3	Rs^3	Rt^3	$UNUSED^7$

Operation: if(ZF==1){
Rd<- Rs + Rt
PC <- PC + 1
}
Number of Cycles: 4

3. where condition:10 Execute if not equal, in other words, if the zero-flag bit is cleared. Otherwise, the current instruction can be treated as a NOP (no operation).

10	00011	X	Rd^3	Rs^3	Rt^3	$UNUSED^7$

Operation: if(ZF!=1){
Rd<- Rs + Rt
PC <- PC + 1
}
Number of Cycles: 4

B. Subtraction – sub instruction:

1. where condition:00 so the current instruction will execute

00	00100	1	Rd^3	Rs^3	Rt^3	$UNUSED^7$

Operation: Rd<- Rs - Rt
PC <- PC + 1
Number of Cycles: 4

2. where condition:01 Execute if equal, in other words, if the zero-flag bit is set. Otherwise, the current instruction can be treated as a NOP (no operation).

01	00100	1	Rd^3	Rs^3	Rt^3	$UNUSED^7$

Operation: if(ZF==1){
Rd<- Rs - Rt
PC <- PC + 1
}
Number of Cycles: 4

3. where condition:10 Execute if not equal, in other words, if the zero-flag bit is cleared. Otherwise, the current instruction can be treated as a NOP (no operation).

10	00100	1	Rd^3	Rs^3	Rt^3	$UNUSED^7$

Operation: if(ZF!=1){
Rd<- Rs - Rt
PC <- PC + 1
}
Number of Cycles: 4

C. Logical AND – AND instruction:

00	00000	X	Rd^3	Rs^3	Rt^3	$UNUSED^7$

Operation: Rd<- Rs AND Rt
PC <- PC + 1
Number of Cycles: 4

D. Jump register – jr instruction:

00	00110	X	Rd^3	Rs^3	000	$UNUSED^7$

Operation: PC<- Rs
Number of Cycles: 3

E. comparison - cmp instruction:

00	00101	X	Rd^3	Rs^3	Rt^3	$UNUSED^7$

Operation:if((Rs) <(Rt)){
PC <- PC + 1
}
Number of Cycles: 4

D.

I-Type Instructions:

A. Immediate addition: addi instruction

1. where condition:00 so the current instruction will execute

00	01000	X	Rt^3	Rs^3	$Immediate^{10}$

Operation: Rt<- Rs + (sign extended I[9:0])
PC<- PC +1
Number of Cycles: 4

2. where condition:01 Execute if equal, in other words, if the zero-flag bit is set. Otherwise, the current instruction can be treated as a NOP (no operation).

01	01000	X	Rt^3	Rs^3	$Immediate^{10}$

Operation: if(ZF==0){
Rt<- Rs + (sign extended I[9:0])
PC<- PC +1
}
Number of Cycles: 4

3. where condition:10 Execute if equal, in other words, if the zero-flag bit is cleared. Otherwise, the current instruction can be treated as a NOP (no operation).

10	01000	X	Rt^3	Rs^3	$Immediate^{10}$

Operation: if(ZF!=0){
Rt<- Rs + (sign extended I[9:0])
PC<- PC +1
}
Number of Cycles: 4

B. Immediate logic AND: andi instruction

00	00111	X	Rt^3	Rs^3	$Immediate^{10}$

Operation: Rt <- Rs AND (sign extended I[9:0])
PC<- PC + 1
Number of Cycles: 4

C. Load Word: lw instruction

00	01001	X	Rt^3	Rs^3	$Immediate^{10}$

Operation: Rt<- M[Rs + (sign extended I[9:0])]
PC <- PC + 1
Number of Cycles: 5
D. Store Word: sw instruction

00	01010	X	Rt^3	Rs^3	$Immediate^{10}$

Operation: M[Rs + (sign extended I[9:0])]<- Rt
PC<- PC + 1
Number of Cycles: 4

E.Branch on equal: beq instruction

00	01011	X	Rt^3	Rs^3	$Immediate^{10}$

Operation: if(Rs = Rt)then
PC<- PC + 1 + ((sign extended I[9:0]) || 00)
else PC<- PC + 1
Number of Cycles: 3

COMPONENTS AND OPERATION:

The MIPS processor as shown in figure 1bellow design uses a total of 10 different major components. Some components appear in more than one instance and in different variations like the multiplexor.

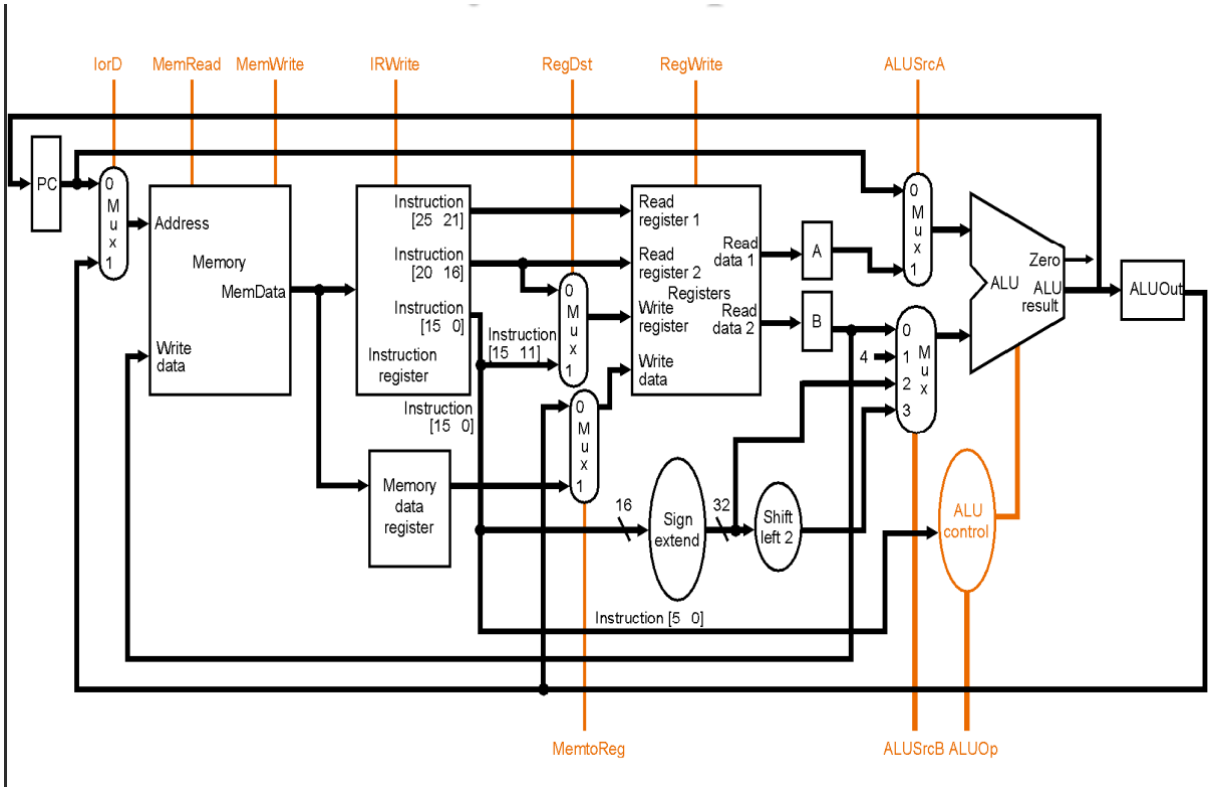


Figure 1:MIPS processor

The major components and quantities used are as follows:

- Register
- Multiplexor
- Random Access Memory
- Sign Extend Module
- Register File
- Shift Module
- Concatenate Module
- Arithmetic Logic Unit
- Arithmetic Logic Unit Controller

Multiplexor: The Mux, is used to enable certain parts of the processor data path at a given time and is controlled by the Sequence Controller.

ALU: The Arithmetic Logic Unit (ALU) performs all of the major arithmetic and logical operations in the processor. Additionally, the ALU performs all of the shift operation with exception to a few 2-bit shift registers outside the ALU.

Program Counter with Control Circuitry: The Program Counter (PC) used in the project's processor design is actually a 24-bit register used to hold the address of the next instruction to be executed by the processor.
the PC_Load signal can be set true in several different scenarios. The most obvious scenario is when the PC_EN signal is set true by the Sequence Controller. This occurs when the Program Counter is loaded with the next instruction address,all other scenarios in which the PC_LOAD signal is set true, enabling input loading of the Program Counter on the rising edge of clock.

Data Memory: This processor design uses the Data Memory as main storage devices. The Memory module is used to store the MIPS Instruction programs that get processed. The RAM module, shown in Figure 1, has 4 inputs and 1 output. All data and address inputs are 24-bits wide in order to support the rest of the processor architecture. The RAM depth can be any arbitrary size up to locations as the Address bus is 24-bits wide. There are also two single bit control signals, namely, MemRd and MemWt. Both control signals are controlled and generated by the Sequence Controller. On the rising edge of MemRd, the Register File content at the location specified by the Address (Addr) bus is placed on the output bus. The RAM module writes the content of the Data input into the Register File at the location specified by the Address bus on the rising edge of MemWt.

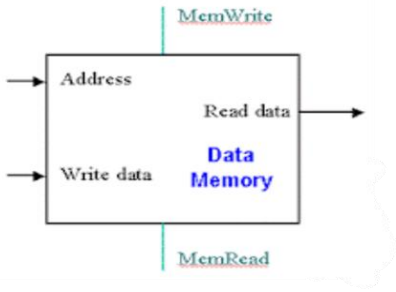


Figure 2:Data Memory

Register File: is similar to the RAM module, only it has dual ports while the RAM module only has a single port. Additionally, in reality the Register File is made from flip flops while RAM is made from DRAM or SRAM cells. This means that the Register File consumes more

power and will take up more area on a real chip but will also be much faster than RAM. Using registers for faster operation is one of the key features of a MIPS processor. The Register File, shown in Figure 2, has a total of 6 inputs and 2 outputs. All data inputs and outputs are 24-bits wide while the Address inputs are 3-bits wide. There are also two single bit control signals, namely RegWr. At the rising edge of Clock, the content of the Register File at the location specified by the Reg1 and Reg2 inputs are placed on the Reg1_Data and Reg2_data outputs, respectively. The Reg1 and Reg2 inputs are derived directly from the Rs (Inst[10-12]) and Rt (Inst[15-13]) fields of the instruction being executed. At the rising edge of RegWr, the content of the Write_Data input is written to the Register File at the location specified by the Write_Reg input.

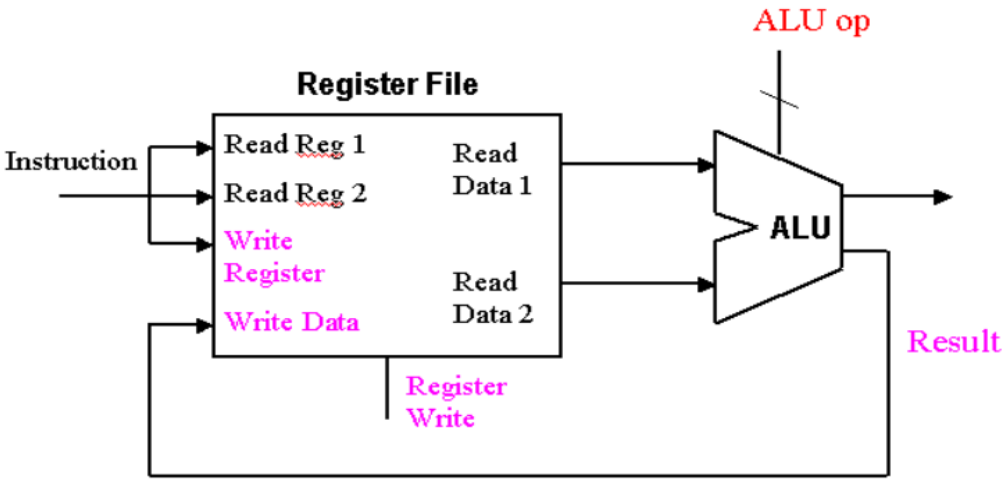


Figure 3: Register File

Sign Extend Module:there two types of extensions Zero-extension for unsigned constants Sign-extension for signed constants , Control signal ExtOp indicates type of extension.

Instruction memory: This is the memory part that contains the instruction needed to be fetched so we can start executing it, this component will take the pc as an input which will be the address of the wanted instruction to be executed. This component can be modified in running time (dynamically) that's why we need to pre define the instructions to the memory so when it is running it will only be used to fetch the instructions from it.

State Diagram:

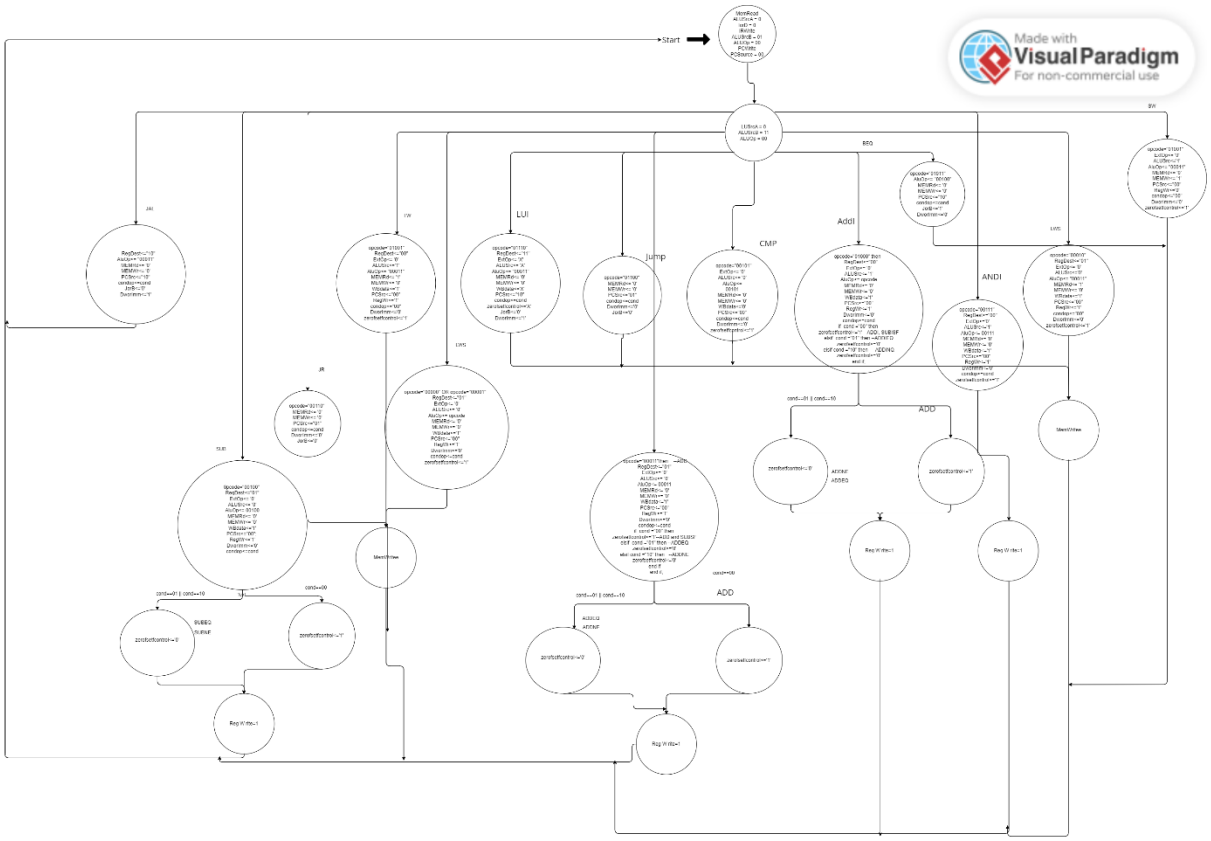


Figure 4:State Diagram

Implementation Design and Test:

To implement the design, Verilog HDL was used.

Instruction Memory:

The instruction Memory Verilog design code is shown bellow

InstructionMem...

```
1
2
3   library IEEE;
4   use IEEE.std_logic_1164.all;
5   use ieee.numeric_std.all;
6
7   entity InstructionMemory is
8       port(
9           ReadAddress : in STD_LOGIC_VECTOR(23 downto 0);
10          Instruction : out STD_LOGIC_VECTOR(23 downto 0)
11          );
12  end InstructionMemory;
13
14  --}} End of automatically maintained section
15
16  architecture Behavioral of InstructionMemory is
17      type RAM is array (0 to 7) of std_logic_vector(23 downto 0);
18      signal IM : RAM;
19  begin
20      IM(0) <=x"012850";
21      IM(1) <=x"012851";
22      IM(2) <=x"012852";
23      IM(3) <=x"012853";
24      IM(4) <=x"012854";
25      IM(5) <=x"012855";
26      IM(6) <=x"012856";
27      IM(7) <=x"012857";
28      Instruction <= IM((to_integer(unsigned(ReadAddress)))) ;
29
30  end Behavioral;
31
```

Test Results: Simulation results for the instruction Memory component is shown in Figure 5 bellow:

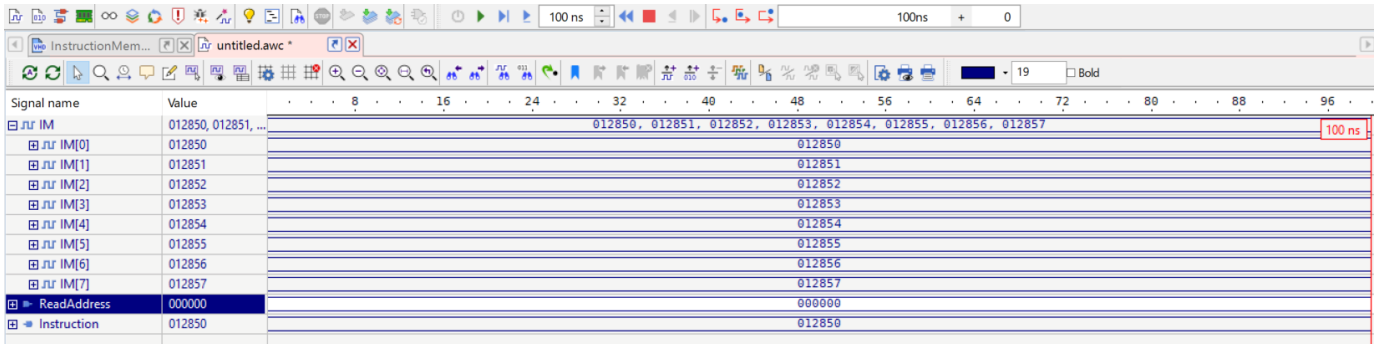


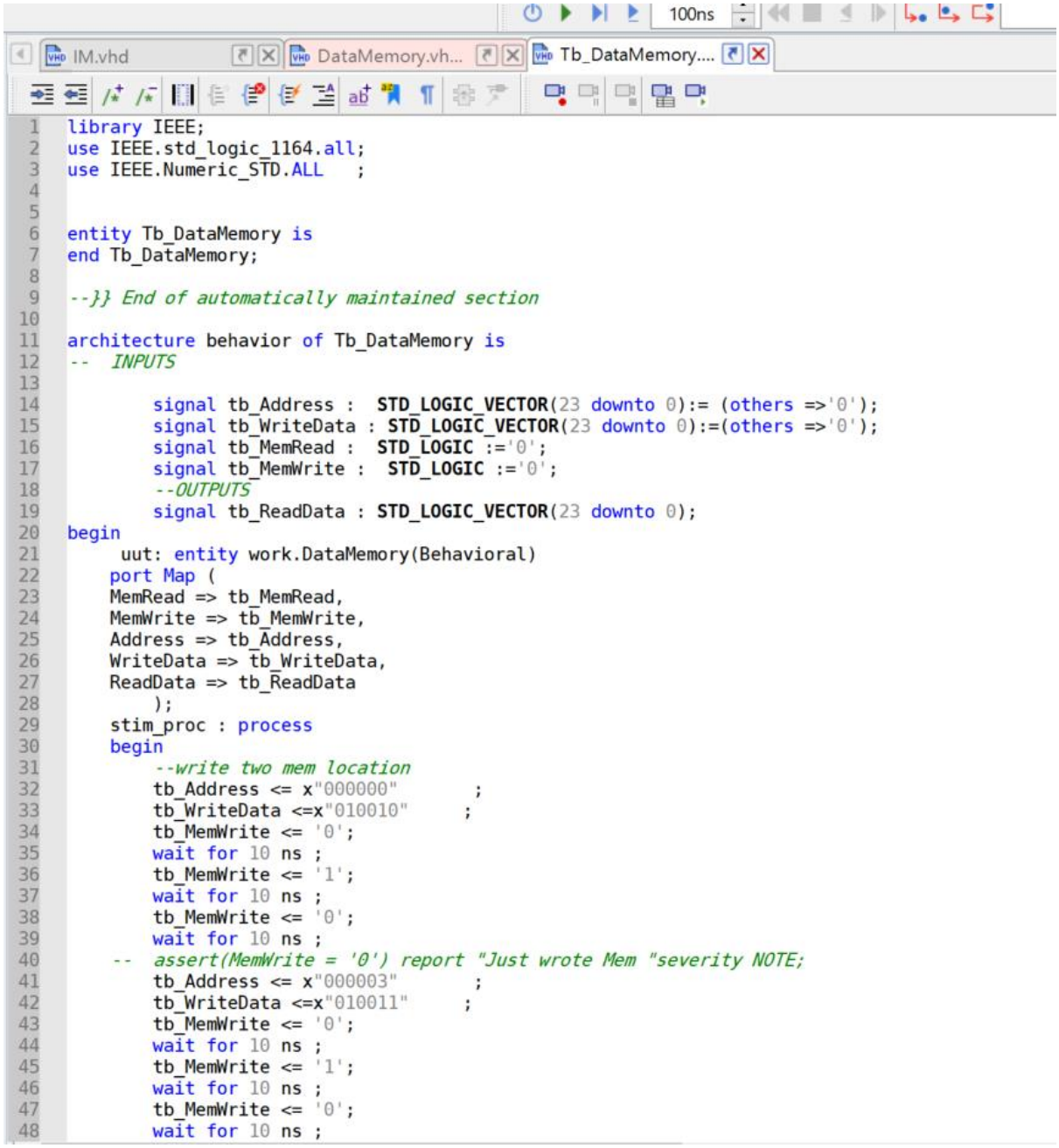
Figure 5:instruction Memory Test

Data Memory:

The Data Memory Verilog design code is shown bellow.

```
entity DataMemory is
     port(
         MemRead : in STD_LOGIC;
         MemWrite : in STD_LOGIC;
         Address : in STD_LOGIC_VECTOR(23 downto 0);
         WriteData : in STD_LOGIC_VECTOR(23 downto 0);
         ReadData : out STD_LOGIC_VECTOR(23 downto 0)
         );
end DataMemory;
--}} End of automatically maintained section
architecture Behavioral of DataMemory is
    type RAM_16_X_24 is array (0 to 15) of std_logic_vector(23 downto 0);
    signal DM : RAM_16_X_24 :=( x"000000",  --Assume starts at 0x100010000
                                x"000000",
                                x"000000",
                                x"000000",
                                x"000000",
                                x"000000",
                                x"000000",
                                x"000000",
                                x"000000",
                                x"000000",
                                x"000000",
                                x"000000",
                                x"000000",
                                x"000000",
                                x"000000"
                        );
begin
    process( MemWrite,MemRead )  --only execute if one of them is 1
    begin
        if (MemWrite = '1')then
            DM((to_integer(unsigned(Address)))/3) <= WriteData; --3 byte per word
        end if;
        if (MemRead = '1')then
             ReadData<= DM((to_integer(unsigned(Address)))/3) ;--3 byte per word
        end if;

    end process;
     -- enter your statements here --
end Behavioral;

```

Test Bench: The Test Bench program for the Data Memory design is shown bellow




```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.Numeric_STD.ALL   ;


entity Tb_DataMemory is
end Tb_DataMemory;

--}} End of automatically maintained section

architecture behavior of Tb_DataMemory is
--  INPUTS

        signal tb_Address :  STD_LOGIC_VECTOR(23 downto 0):= (others =>'0');
        signal tb_WriteData : STD_LOGIC_VECTOR(23 downto 0):=(others =>'0');
        signal tb_MemRead :  STD_LOGIC :='0';
        signal tb_MemWrite :  STD_LOGIC :='0';
        --OUTPUTS
        signal tb_ReadData : STD_LOGIC_VECTOR(23 downto 0);
begin
     uut: entity work.DataMemory(Behavioral)
    port Map (
    MemRead => tb_MemRead,
    MemWrite => tb_MemWrite,
    Address => tb_Address,
    WriteData => tb_WriteData,
    ReadData => tb_ReadData
        );
    stim_proc : process
    begin
        --write two mem location
        tb_Address <= x"000000"       ;
        tb_WriteData <=x"010010"     ;
        tb_MemWrite <= '0';
        wait for 10 ns ;
        tb_MemWrite <= '1';
        wait for 10 ns ;
        tb_MemWrite <= '0';
        wait for 10 ns ;
    --  assert(MemWrite = '0') report "Just wrote Mem "severity NOTE;
        tb_Address <= x"000003"       ;
        tb_WriteData <=x"010011"     ;
        tb_MemWrite <= '0';
        wait for 10 ns ;
        tb_MemWrite <= '1';
        wait for 10 ns ;
        tb_MemWrite <= '0';
        wait for 10 ns ;
```

```
        wait for 10 ns ;


        --Read Code
        tb_Address <= x"000000";
        tb_MemRead <= '0';
        wait for 10 ns ;
        tb_MemRead <= '1';
        wait for 10 ns ;
        tb_MemRead <= '0';
        wait for 10 ns ;
    --  assert(MemWrite = '0') report "Just wrote Mem "severity NOTE;
        tb_Address <= x"000003";
        tb_MemRead <= '0';
        wait for 10 ns ;
        tb_MemRead <= '1';
        wait for 10 ns ;
        tb_MemRead <= '0';
        wait for 10 ns ;
        assert false
    report "End"
    severity failure;
    end  process;
     -- enter your statements here --

end behavior;
```

Test Results: Simulation results for the Data Memory component is shown in Figure 6 bellow:

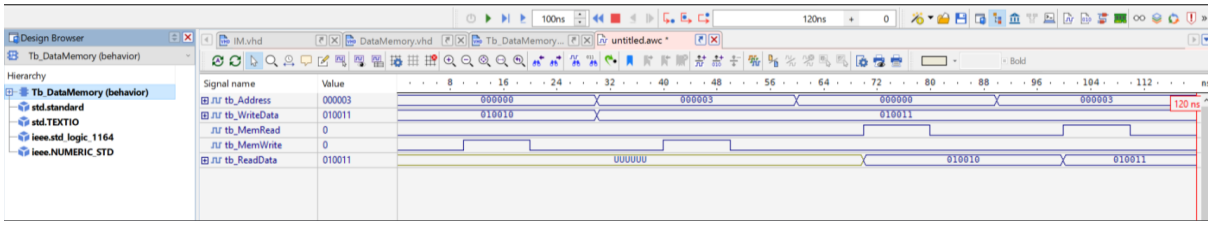


Figure 6:Data Memory Test

MUX 2x1:

the mux were building to select between the zero flag and set flag , because all the instruction need set flag except the instruction in condition 01 and 10, so we need to use mux ,The MUX Verilog design code is shown in Figures 5

```vhdl
-- the entity of Mux2x1 to select between zero flag or set flag
entity mux2to1 is
  port (zero_flag, set_flag, control : in std_logic;
     Zero_flag_orSetFlag_out : out std_logic);
end mux2to1;

architecture behaviour of mux2to1 is
begin
  process (zero_flag, set_flag, control)
  begin
    if control = '0' then
      Zero_flag_orSetFlag_out <= zero_flag;
    else
      Zero_flag_orSetFlag_out <= set_flag;
    end if;
  end process;
end behaviour;
```

Register File:

The Register File unit is a digital circuit that is used to design computer processors. It is a memory unit, it provide quick access to data stored in the registers by stores the values of a number of registers in our case the number of registers are 8 (0-7) that can be represented in 3 bits.
Each register in the register file can store one data value that in our cas can be 24bit , which the processor can generate as an operand for logical and mathematical operations (ALU input). The computer's main memory (data memory) can also receive and send values from the registers.

To build the register file, the first step is to define the input and output port of the component as shown in the figure below, first by using the generic type parameter (that allow us to specify a value of parameters to use them to determine the properties of array for example) we have define two components that are the number of registers and the length of each register (how many bits can store) to use them to define the length of some port, then to define the input and output port we use the port that is interface to a VHDL component or entity, for the inputs we have clk that represent the clock, writeEnable that represent the enable bit that decide if we want to write into a register or not, writeAddress that represent the address of the register that we want to write the data on it usually it is Rd but in some cases it can be Rt , dataIn represent the data that we want to write it inot the register that came from the ALU or data memory, RA,RB that represent the address of the registers that we want to read the data from it, usually they are the registers Rs, Rt. For the output ports, we have BUSA that represent the value that stored in the address of RA, BUSB represent the value that stored in the address of RB.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity registerfile is
  generic (
    numberOfRegister : integer := 8;
    registerLength : integer := 24);
  port (
    clk      : in std_logic;
    writeEnable       : in std_logic;
    writeAddress    : in unsigned(2 downto 0);
    dataIn  : in std_logic_vector(registerLength-1 downto 0);
    RA  : in unsigned(2 downto 0);
    RB  : in unsigned(2 downto 0);
    BUSA: out std_logic_vector(registerLength-1 downto 0);
    BUSB: out std_logic_vector(registerLength-1 downto 0)
  );
end entity;
```

And the way that the register file work is shown in the figure below, first we define an array that represent the data that stored in every register address (filled with random data) for example if RA was equal 101 that is 5 then BUSA will be equal the address on the indes number 4 in array that is 010001, and if writeAddress was equal 011 then the data that came from ALU or data memory will be stored in the address number 3 in the register file, and to do this we make the write process be in the rising edge of the clock, so whenever this happen it will take the value of the dataIn and store it on the address that equal the value of writeAddress such as registers(to_integer(writeAddress)) <= dataIn, and if we want to take the data from the register file we use this command BUSA <= registers(to_integer(RA)), that here take the value of RA and convert it to integer that represent the address of where the data is store, and put this data into the output port that is BUSA

```
architecture regfile of registerfile is
  type arrayOfRegisters is array(0 to numberOfRegister-1) of std_logic_vector(registerLength-1 downto 0);
  signal registers : arrayOfRegisters:=(X"000000",X"000001",  X"000011", X"001001",X"010001", X"101001",X"110001",X"011101");
begin

  process(clk)
  begin
    if rising_edge(clk) then
      if writeEnable = '1' then
        registers(to_integer(writeAddress)) <= dataIn;
      end if;
    end if;
  end process;

  BUSA <= registers(to_integer(RA));
  BUSB <= registers(to_integer(RB));
end architecture;
```

Testing:

The figures below show a simulation of the register file, we have the writeenable=1, means that we will write into the register file on the address 5 that is saved in writeAddress, and we have the value of RA=4, and RB =6, first we can see in the second image that the value of the address 5 is still have the old value when clk =0, but when clk became 1 as shown in the first image the value on the address 5is changed and become equal to dataIn, and then the value of BUSA and BUSB become equal to the value on the address 4 and 6 that the address on RA,RB

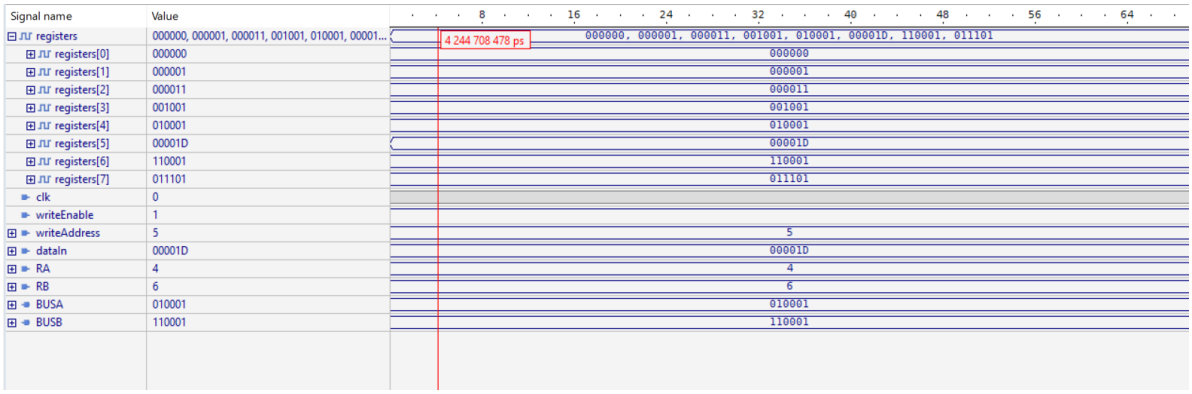


Figure 7:Register File Test1

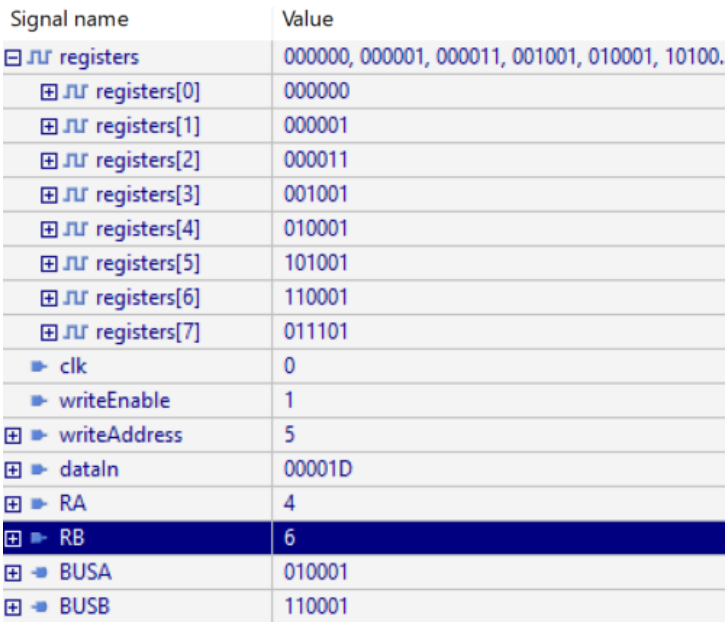


Figure 8: Register File Test2

ALU (Arithmetic and logical unit):

start building an ALU , we define an two input register ,define an input ZeroFlagorSetFlag ,ALU condition ALU opcode as input and ALU out , Zero flag as output , every register size is 24 bit ,condition 2 bit , Opcode 5 bit and ALU out 24 bit.

```
----------------------------------------------------------------------------------------------------
-- the entity of ALU of 24- bit
entity ALU is
    port(
    Reg1, Reg2 : in  std_logic_vector(23 downto 0);  -- 2 inputs 24-bit
    ALU_Out    : out  std_logic_vector(23 downto 0); -- 1 output 24-bit of ALU_out
    Zero_flag_orSetFlag  :  in  std_logic;            -- one bit for zero flag that could be input and output
    ALU_Condition  : in  std_logic_vector(1 downto 0);  -- 2 bit for the condition
    ALU_Opcode :    in  std_logic_vector(4 downto 0); -- 5 bit for the opcode
    zeroFlag : out std_logic        -- zero flag
    );

end;
```

start build the Behavioral of ALU, start process on the input of ALU
in ALU we have 2 input for register , 3 input control signal that they exist from control unit.

RType:

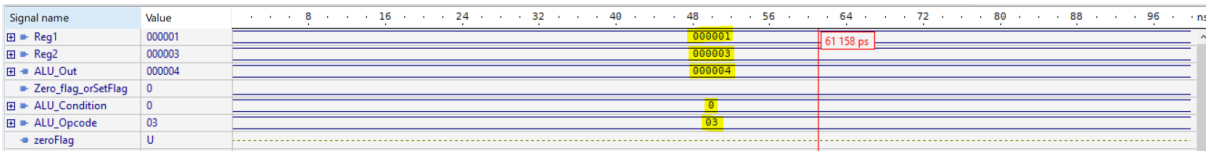
ADD: we check on the opcode ,ALU condition and ZeroflagOrSetFlag (output from mux) if all values are true the ALU will add the two register together then save the result on ALU Out (output of ALU), Add not change on the zero flag value so the zero flag will be the same value from previous instruction .
ADDEQ: we check on the opcode ,ALU condition , then we must check on the result of zero flag ,is executed if and only if the zero flag bit has the value of 1 , so after the output of mux will pass the result of zero flag then check it , depend on it will executed or not .
if the zero flag zero is 0, the instruction not executed and will be a NOP operation that will express it with null .

```
architecture Behavioral of ALU is

begin

    process(Reg1,Reg2,ALU_Opcode,ALU_Condition,Zero_flag_orSetFlag)
    begin
           --RType

        if (ALU_Opcode = "00011" and ALU_Condition = "00" and Zero_flag_orSetFlag='0') then     -- ADD
            ALU_Out <= Reg1 + Reg2;



            elsif (ALU_Opcode = "00011" and ALU_Condition = "01") then  -- ADDEQ
                if (Zero_flag_orSetFlag='1') then  -- the result of mux it will pass a zero flag
                    ALU_Out <= Reg1 + Reg2;
                else
                    null;    -- NOP
                end if;
```

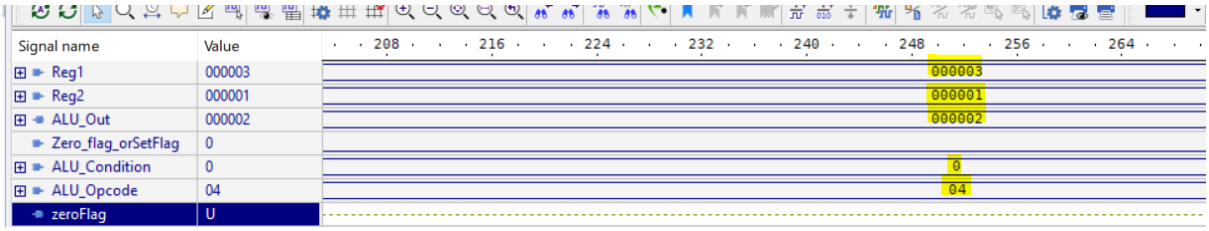
ADDNE:we check on the opcode ,ALU condition , then we must check on the result of zero flag ,is executed if and only if the zero flag bit has the value of 0, so after the output of mux will pass the result of zero flag then check it , depend on it will executed or not .
if the zero flag zero is 1 , the instruction not executed and will be a NOP operation that will express it with null .

```
elsif (ALU_Opcode = "00011" and ALU_Condition = "10") then  -- ADDNE
    if (Zero_flag_orSetFlag='0') then   -- the result of mux it will pass a zero flag
        ALU_Out <= Reg1 + Reg2;
    else
        null;    -- NOP
    end if;


elsif (ALU_Opcode = "00100" and ALU_Condition = "00" and Zero_flag_orSetFlag='0') then  -- SUB  -
    ALU_Out <= Reg1 - Reg2;
```

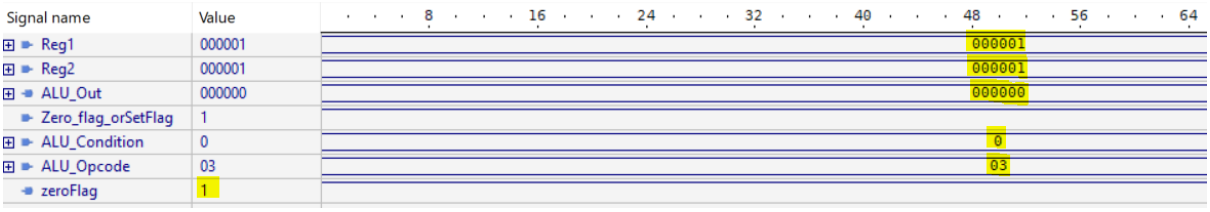
Testing:
All the instruction has a set flag to zero , except ADDEQ, ADDNE
ADD: for the test of ADD , the value of registers (Reg1) is 1 and (Reg2) is 3 , the result will be in ALU out is 4 , but the instruction will executed if the opcode is 3 ⇒ 00011 , the condition is 00 , the zero flag not set because the ADD not set the value of zero flag , the ZeroFlag and SetFlag will be 0 ⇒ the output of the mux will access the setflag and it's will be zero.



SUB: for the test of SUB, the value of registers (Reg1) is 3 and (Reg2) is 1 , the result will be in ALU out is 2 , but the instruction will executed if the opcode is 4 ⇒ 00100, the condition is 00 , the zero flag not set because the SUB not set the value of zero flag , the ZeroFlag and SetFlag will be 0 ⇒ the output of the mux will access the setflag and it's will be zero.



SUBSF: for the test of SUBSF, the value of registers (Reg1) is 1 and (Reg2) is 1 , the result will be in ALU out is 0 , but the instruction will executed if the opcode is 3 ⇒ 00011 , the condition is 00 , the SUBSF can set the value of zero flag that if the result of ALU out is zero , so the zero flag will set 1 as shown in the figure below , and the zeroflagOrsetflag will be 1 , that the output of mux that will pass the value of set value that will be 1 of SUBSF .



AND: for the test of AND, the value of registers (Reg1) is 3 and (Reg2) is 1 , the result will be in ALU out is 1 , but the instruction will executed if the opcode is 0 ⇒ 00000, the condition is 00 , the zero flag not set because the AND not set the value of zero flag , the ZeroFlag and SetFlag will be 0 ⇒ the output of the mux will access the setflag and it's will be zero.

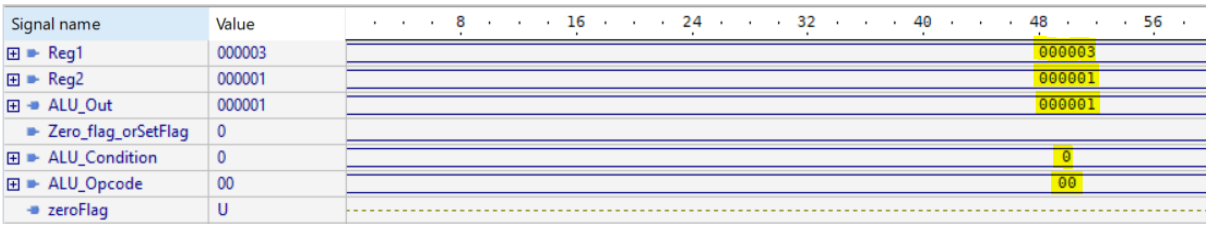
AND:

Reg1:00000000000000000000000011

Reg2:00000000000000000000000001

ALU Out ⇒ Reg1 and Reg2 ⇒ 00000000000000000000000001

as we see in the figure the result show as we calculated .



CMP: for the test of CMP, the value of registers (Reg1) is 3 and (Reg2) is 1, the instruction will executed if the opcode is 5 ⇒ 00101, the condition is 00 , the zero flag will set if Reg1<Reg2 it set 1, the

Sign or Zero Extend.
we use in this project a zero extend so in the code below the data in will be 10 bit and the data out will be 24 bit , the extend on bit will be zero.

Shift By four:
in the shift entity start define a one input that 17 bit and one output 17 bit , then the output will be shift by 4 zeros.

```
-- the library that needed
library ieee;

use ieee.std_logic_1164.all;

use ieee.std_logic_signed.all;

entity shift  is
    port(
    datain : in std_logic_vector(16 DOWNTO 0);
    dataout: out std_logic_vector(16 DOWNTO 0)
    );
end;

-- the behavioral of shift entity


architecture shiftbyfour of shift is

begin
    -- shift by 4
    dataout<= datain(12 downto 0) & "0000";

end;
```

test:
in the figure below , we see that the input has been shifted by 4.

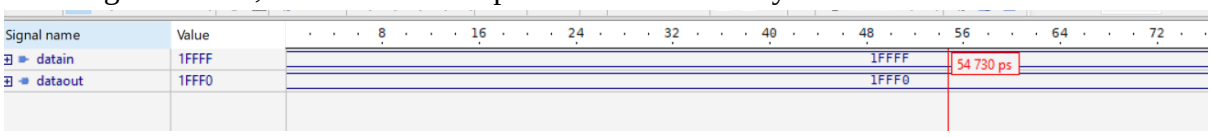


Figure 9:Shift Test

Control Unit:

There are two control units that are the main control unit, and PC control unit, the main control unit of a multi-cycle processor is in charge of producing the control signals that direct the flow of data through the processor's many stages. Each clock cycle, the control unit decodes the instructions from that came from the fetch stage and creates the necessary signals to turn on the appropriate functional units, such as the ALU, memory, and input/output units. For one instruction to be executed in a multi-cycle processor.

In the main control unit we generate the signals (RegDest, ExtOp, ALUSrc, AluOp, MEMRd, MEMWr, WBdata, PCSrc, RegWr, condop, zerofsetfcontrol, JorB, DworImm), the RegDist signal is to decide which register should be the destination register, ExtOp it is used to make the extender to choose to extend the number by sign or not, the ALUSrc it is the selection bit of a mux that choose from the register value or the immediate value to be the input of the ALU, AluOp decide the operation of the instruction that will be running inside the ALU, MEMRd to define if there is a need to read from the memory or not, and MEMWr is to check if the instruction need to write to memory or not, WBdata it is the selection bit of the mux that choose from the ALU result ans the memory output to write it on the register, PCSrc it is used as a selection line to choose which pc address to use, RegWr to enable the writing on register in the register file, condop to send the cond bits to ALU, zerofsetfcontrol is the selection bit of the mux that decide between the zero flag bit or the set flag bit that will be the input of the ALU, JorB decide if the instruction is jump or branch, DworImm to decide wich to write on the register the output from the write back or the value of the PC.

The truth table for the control unit:

INST	RegDest	ExtOp	ALUSrc	AluOp	MEMRd	MEMWr	WBdata	PCSrc	RegWr	condop	Zerofsetf control	JorB
ADD	01	0	0	00011	0	0	00	00	1	00	1	x
ADDEQ	01	0	0	00011	0	0	00	00	1	01	0	x
ADDNE	01	0	0	00011	0	0	00	00	1	10	0	x
ADDI	00	0	1	01000	0	0	00	00	1	00	1	x
ADDIEQ	00	0	1	01000	0	0	00	00	1	01	0	x
ADDINE	00	0	1	01000	0	0	00	00	1	10	0	x
SUBSF	01	0	0	00011	0	0	00	00	1	00	1	x
SUBISF	00	0	1	01000	0	0	00	00	1	00	1	x
AND	01	0	0	00000	0	0	00	00	1	00	1	x
CAS	01	0	0	00001	0	0	00	00	1	00	1	x
LWS	01	0	0	00011	1	0	01	00	1	00	1	x
SUB	01	0	0	00100	0	0	00	00	1	00	1	x
SUBEQ	01	0	0	00100	0	0	00	00	1	01	0	x
SUBNE	01	0	0	00100	0	0	00	00	1	10	0	x
CMP	xx	0	0	00101	0	0	00	00	x	00	1	x
JR	00	0	1	00011	1	0	01	00	1	00	1	x
ANDI	00	0	1	00111	0	0	00	00	1	00	1	x
LW	00	0	1	00011	1	0	01	00	1	00	1	x
SW	xx	0	1	00011	0	1	xx	00	0	00	1	x
BEQ	xx	x	x	00100	0	0	xx	10	x	00	x	1
J	xx	x	x	01100	0	0	xx	01	x	00	x	0
JAL	10	x	x	00011	0	0	10	10	00	00	x	1
LUI	11	x	x	00011	0	0	xx	10	00	00	x	x

Data Path

The component in yellow it represent the entity component (Instruction Memory, Register File, ALU, MUXs , Shift, Extent, Memory data, WB mux), the component in green is the process code that have if statement the program generate a component for them.

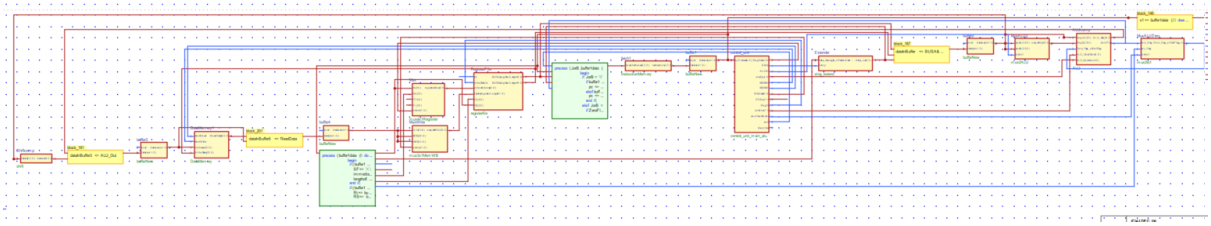


Figure10:Data Path

Result:

The below figures show the test for the whole system when the opcode is for ADD and the cond=00

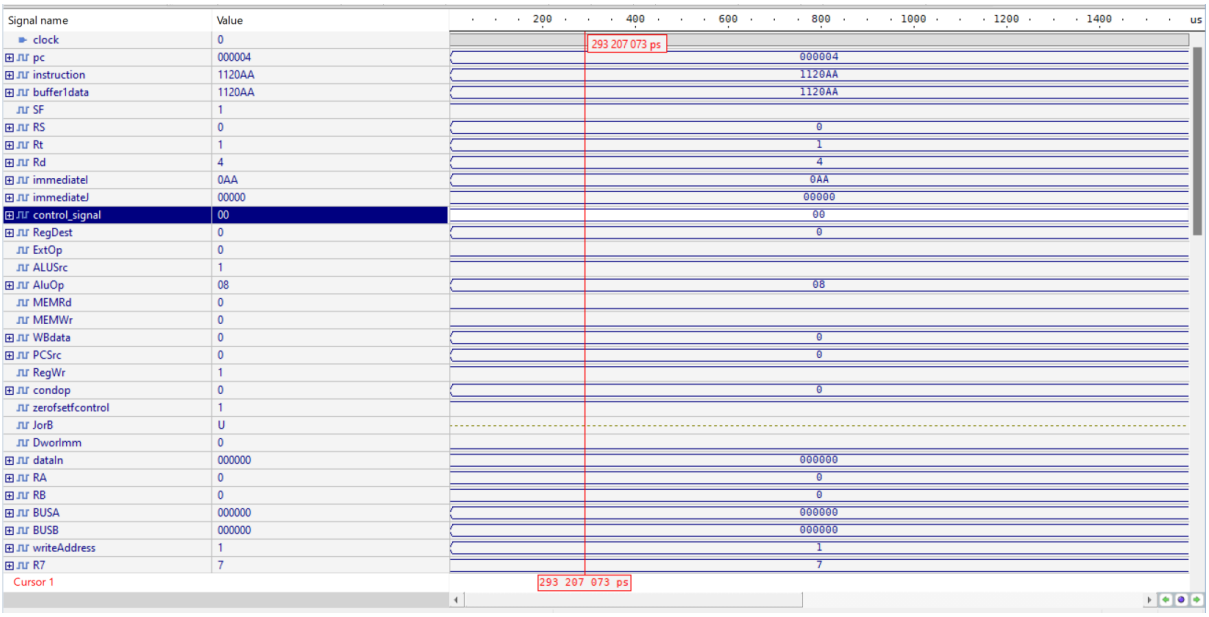


Figure 10:System Test1

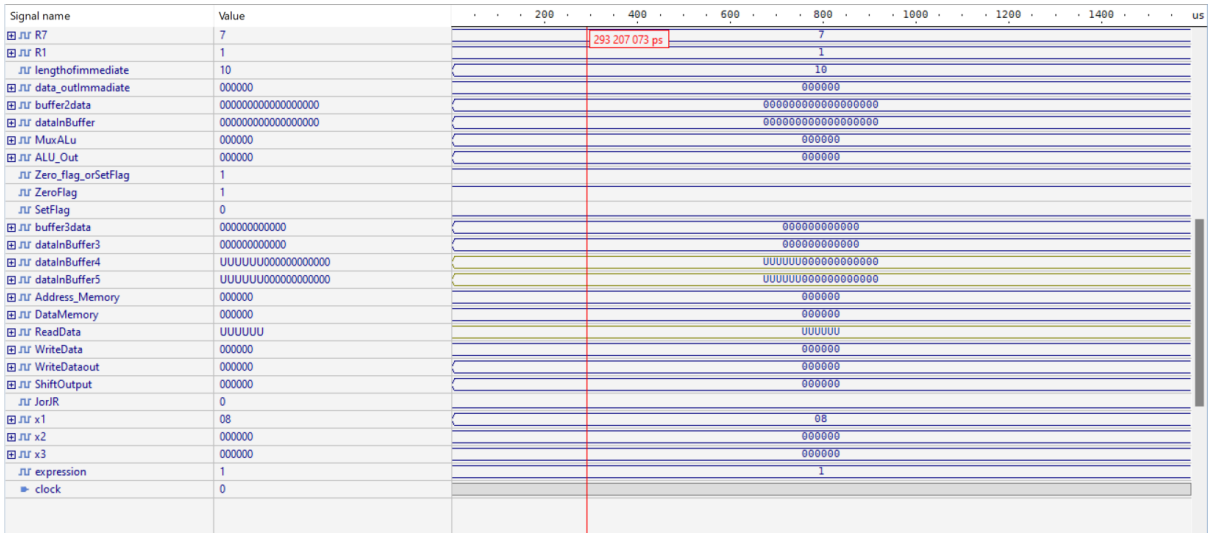
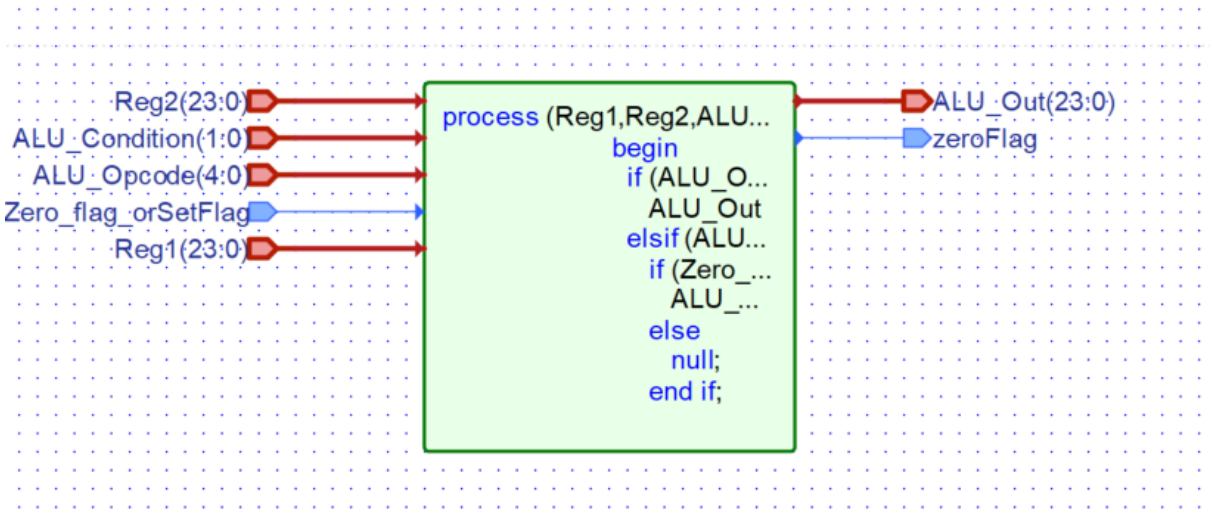


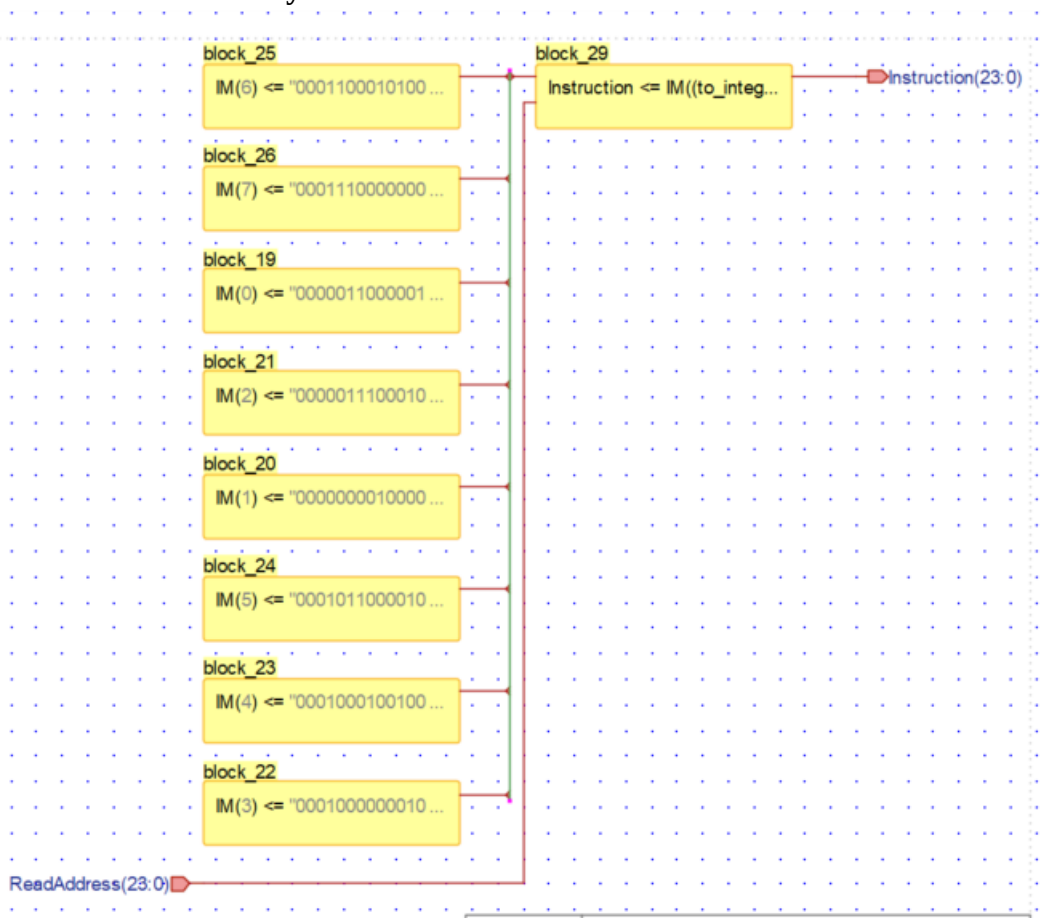
Figure 11:System Test2

The Design For each component:

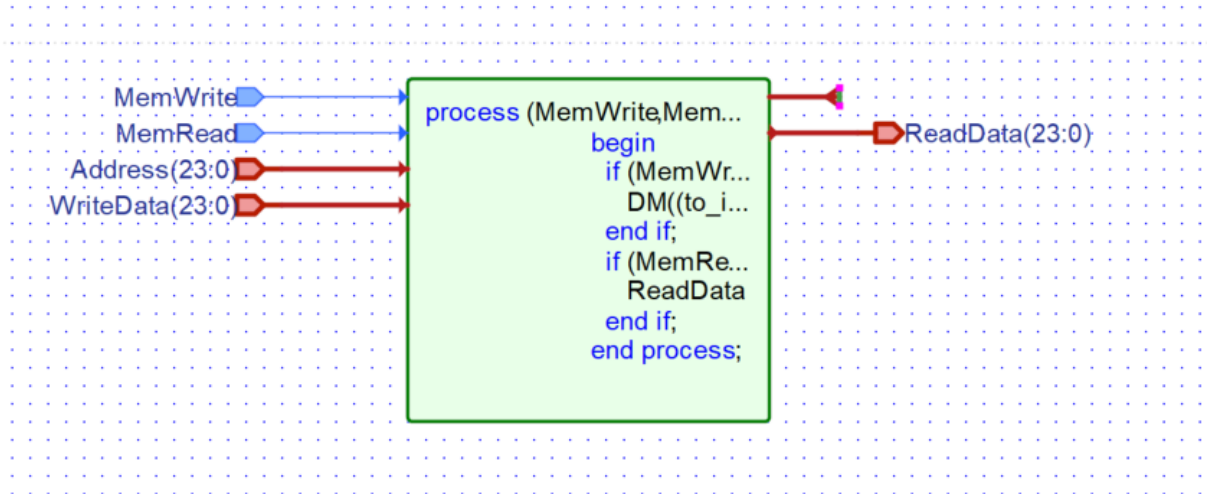
1. ALU



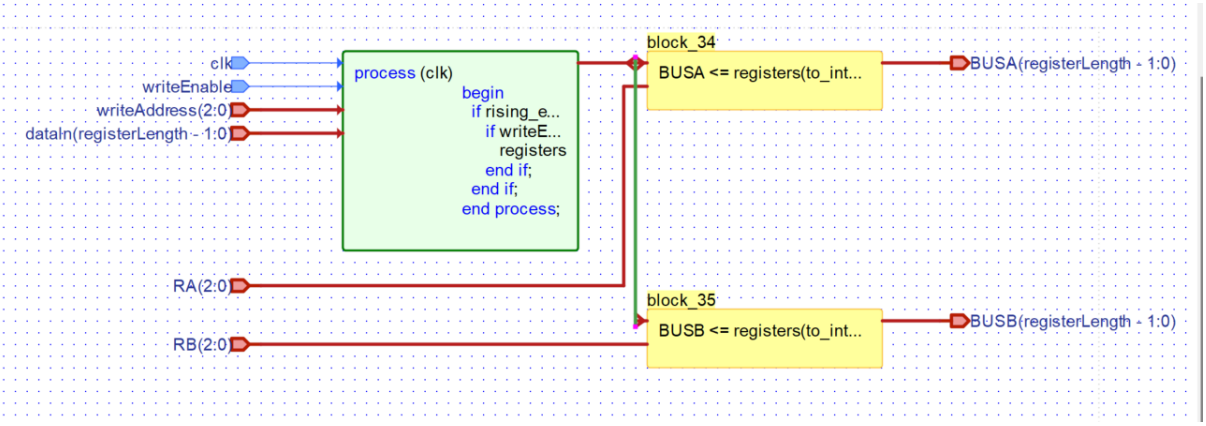
2. Instruction Memory



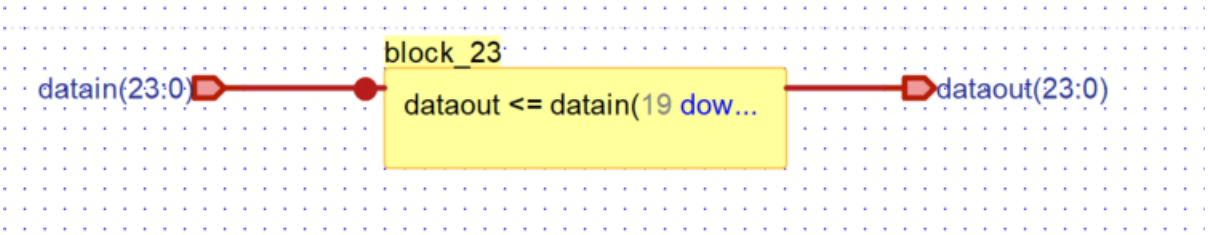
3.Data Memory



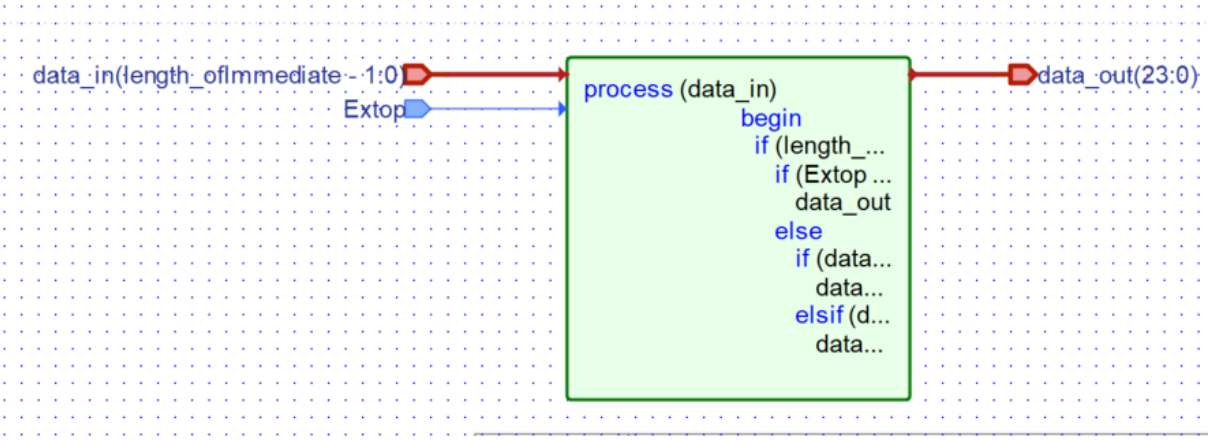
4.Register file



4. Shift



5. Extender



Conclusion:

This project was very helpful since it helped us to increase our understanding and comprehension of MIPs architecture, and getting more familiar with it and since we implemented and simulated using an HDL, and also it helped us getting more knowledge to use VHDL language, and (active-HDL tool) hardware design tools in general. The only hardship that we faced when we were implementing this project was connecting all the components into a single data–path which will be the whole processor