In the name of God

Instructor:

Ms Parnian Kamran

Team:

Mohammadreza Shamshirgarha 9331067

Ali Janalizadeh 9331007

The purpose of this project was the implementation of a MIPS Processor.

The data registers of this processor are 16 bits long and the program counter is incremented with every clock cycle.

We used the Modelsim IDE and the VHDL language for the implementation of the project.

RAM implementation:

```
entity Ram is
        port (
          WriteData :in std_logic_vector(15 downto 0);
                Address : in std_logic_vector(4 downto 0);
                Read : in std_logic;
                Write : in std_logic;
                MemData :out std_logic_vector(15 downto 0)
        );
end Ram;
```

as you can see we have a 16bit input that contains the data to be written into the RAM, a 5 bit address since we have 2^5 or 32 lines of data,and a 16bit output that is used for reading from the RAM

we also have 2 control signals read and write for the corresponding instructions.

RegisterFile implementation:

```
entity REGFIL is
        port (
           Read_Register_1 :in std_logic_vector(3 downto 0);
                         Read_Register_2 :in std_logic_vector(3 downto 0);
                         Write_Register :in std_logic_vector(3 downto 0);
                         Write_data : in std_logic_vector(15 downto 0);
                         Write: in std_logic;
                         Read_Data_1 :out std_logic_vector(15 downto 0);
                         Read_Data_2 :out std_logic_vector(15 downto 0)
                         );
end REGFIL;
```

Read_Register_1 and Read_Register_2 are the addresses for the corresponding outputs Read_Data_1 and Read_Data_2.

We might want to write into the register file,for that we'll need to activate the control signal Write and give the 16bit data that's going to be written into the registerfile with the Write_data signal and give the address of the register that we're going to write the data to with the Write_Register input signal.

Instruction_Register implementation:

```
entity INS is
        port (
                        Data_in : in std_logic_vector(15 downto 0);
                        Source1 :out std_logic_vector(3 downto 0);
                        Source2 :out std_logic_vector(3 downto 0);
                        Destination :out std_logic_vector(3 downto 0);
                        OP_Code :out std_logic_vector(3 downto 0)
                        );
end INS;
```

in the instruction register module, the input is given via the 16bit Data_in input signal and the 4 outputs of the instruction_register or IR are the different parts of our instruction format.

ALU implementation:

```
entity ALU is
 port (
  opcode : in std_logic_vector ( 3 downto 0 ) ;
      register_1 : in std_logic_vector ( 15 downto 0 );
      register_2 : in std_logic_vector ( 15 downto 0 );
      carry_flag_sum : out std_logic ;
      carry_flag_sub : out std_logic ;
      alu_out : out std_logic_vector ( 15 downto 0 )
     );
end ALU;
```

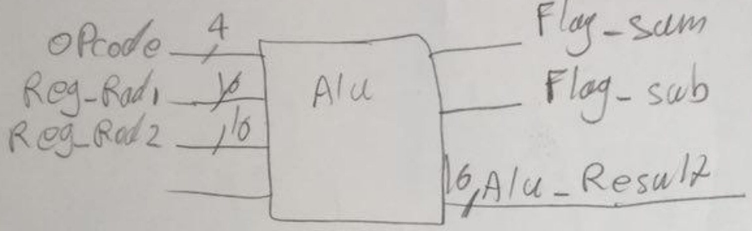
the opcode input signal is used for specifying which operand is to be used. register_1 and register_2 are the 2 input data signals and alu_out

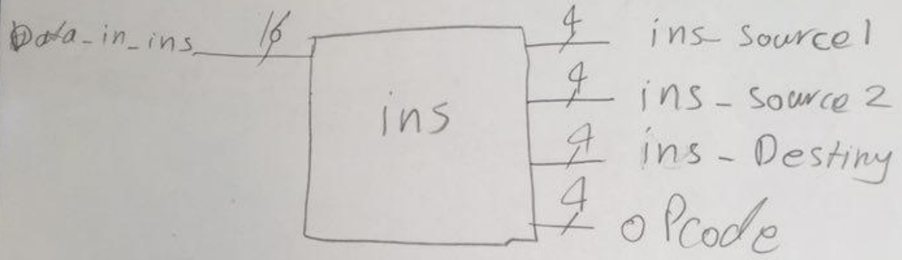
is the output data signal. We also show the two carry_flag_sum and carry_flag_sub flags as the output of this module

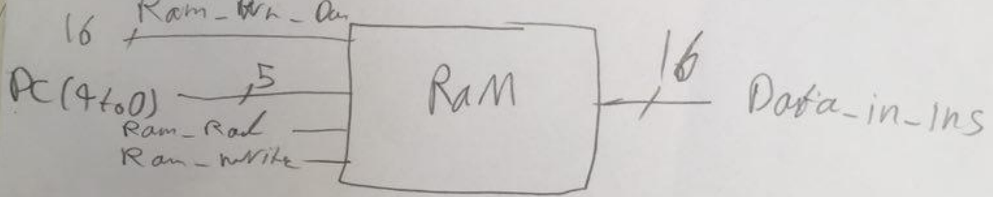
TopModule Implementation:

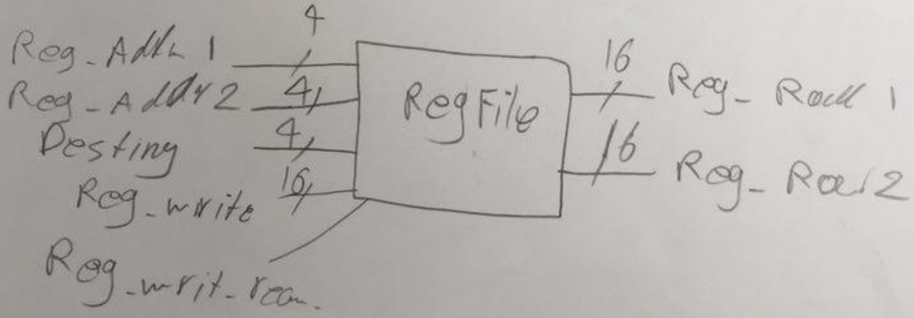
```
entity TopModule is
 port(
          CLK : in std_logic
 );
end TopModule;
```

this is the main control unit of the entire CPU. This module has the responsibility of connecting all the submodules and different components which have been implemented and controlling the different states the CPU goes to while controlling these different components so that the final computer takes shape.

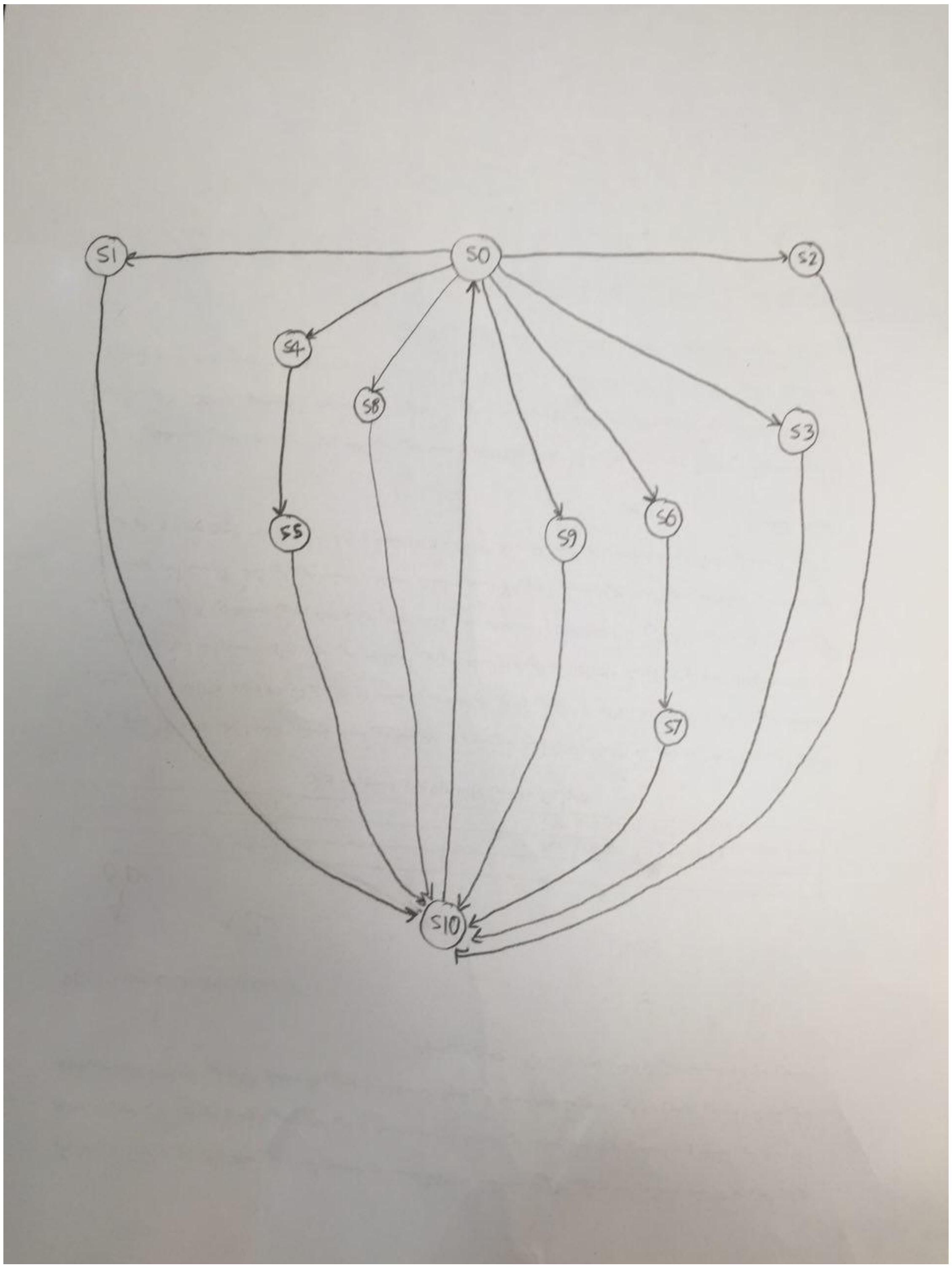

OPcode
4
Reg_Rad1
16
Reg_Rad2
16
Alu
Flag_sum
Flag_sub
16
Alu_Result


Data_in_ins
16
ins
4
ins_source1
4
ins_source2
4
ins_Destiny
4
oPcode


16
PC(4to0)
5
Ram_Rad
Ram_write
RaM
16
Data_in_Ins


Reg_Addr1
4
Reg_Addr2
4
Destiny
4
Reg_write
16
Reg_writ_Rem.
RegFile
16
Reg_Rad1
16
Reg_Rad2

The different states of our CPU:



State0:

This is our initial state, in this state we look at the opcode to see whats to be done, then according to that op code we initialize the appropriate registers to the correct values and then pass on to the corresponding state appropriate to the opcode.

Also if the OPCODE is saying to RESET, we reset right here.

State1:

In this state we wanna work with the ALU so basically, in this state we give the ALU the opcode with the inputs taken from the registerfile and we then write the result of the alu back to the registerfile with the address given to us by the instruction register component.

State 2:

This state is used for handling the Addi and Subi instructions. We sign extend the immediate part of the IR to 16 bits and then look at the instruction, if its Addi, we add the value stored in register1 to it and store it back in register1, if its Subi, we calculate the difference between the sign extended immediate and register1 and store the result back into register1.

State3:

This state is used for the jump instruction and does just as asked in the project definition i.e. changes the PC's value to the said value.

State 4 and State 5:

This state is used for the load instruction.in this state we take the bits 8-11 as the first registers address and the bits 4-7 as the second register address. The bits 0-3 are then sign extended to 16 bits, then we add the value of the result with the value stored in the first register to get the address.then we take the 5 least significant bits of that 16bit address as the address input to the RAM module,then we load the output of the RAM onto the second register.State 4 is for reading from the RAM and State 5 is used for writing onto the RegisterFile

State 6 and State 7:

This state is used for the store instruction.state 6 does exactly what state 4 does and calculates the RAM's address then state 7 goes on and stores the value that's in the second register into the said address In the RAM.

State 8:

This state is used for the branch if the sources are equal instruction.

We store the said value inside the PC

State 9:

This state is used for the branch if the first source is greater instruction.

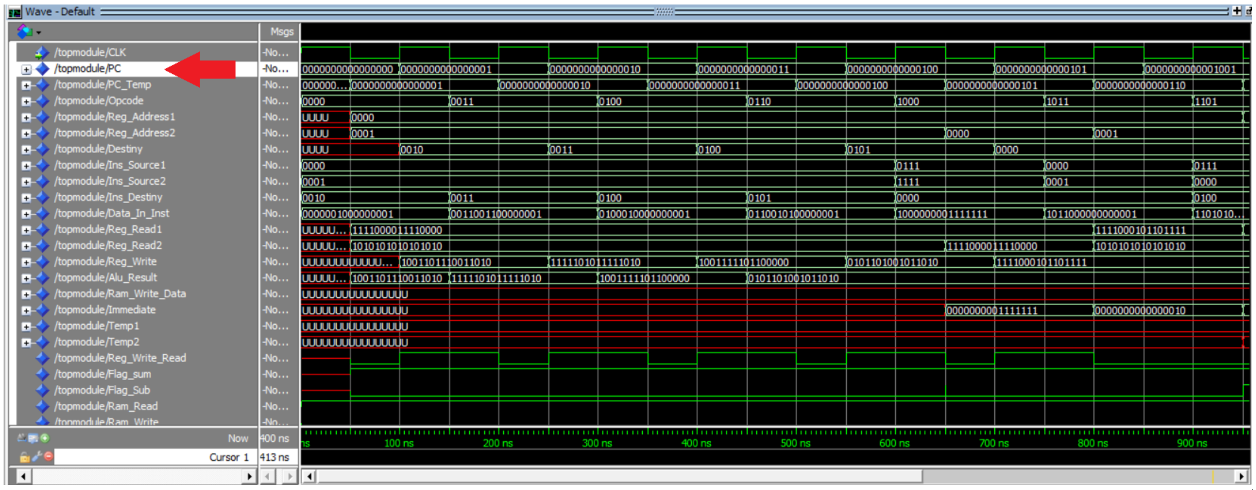
We store the said value inside the PC

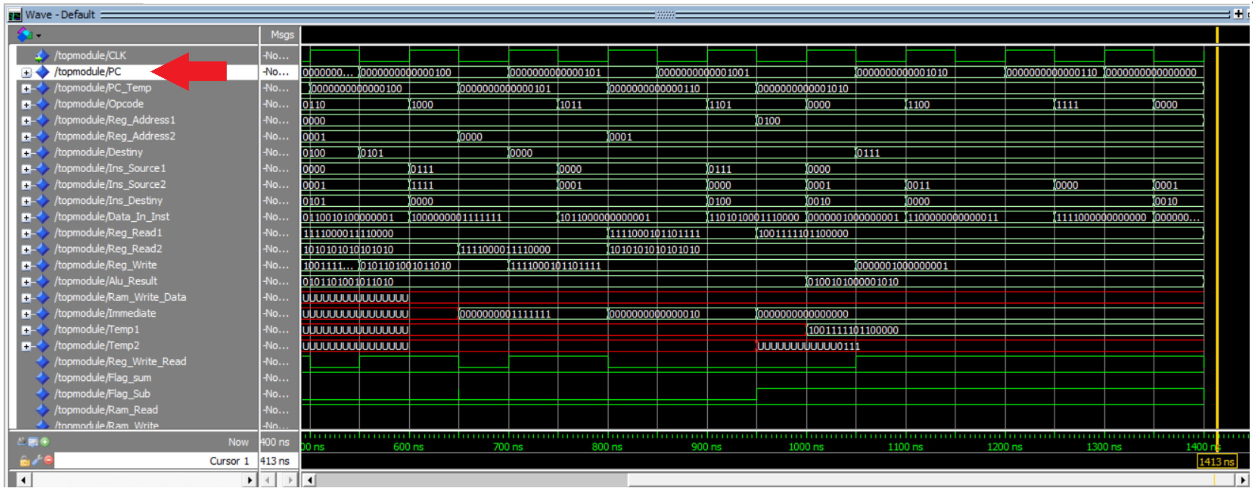
NOTE: we use the bits 4-7 to specify the address of source 1 and the bits 0-3 to specify the address of source2.

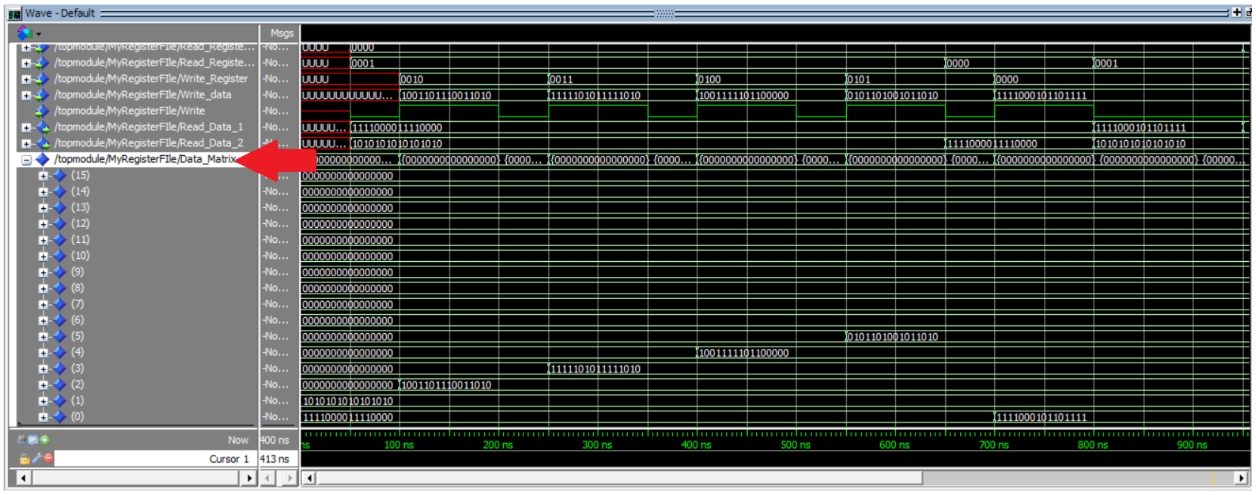
State 10:

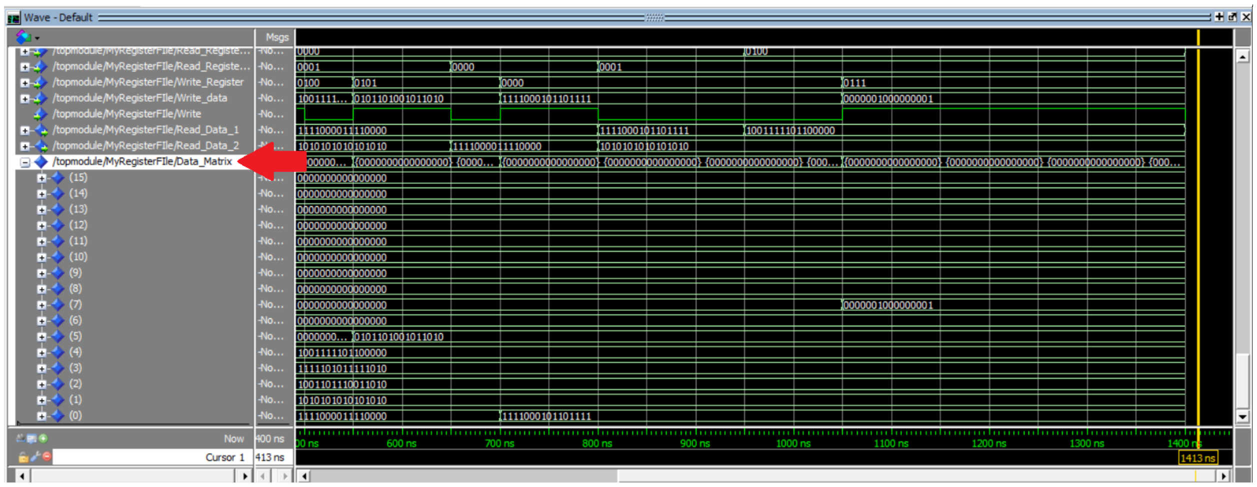
In this state we set the Ram components address to the PC value and read the next instruction from the Ram and return to the initial state(State0).

تست پروژه:


Wave - Default
/topmodule/CLK
/topmodule/PC
/topmodule/PC_Temp
/topmodule/Opcode
/topmodule/Reg_Address1
/topmodule/Reg_Address2
/topmodule/Destiny
/topmodule/Ins_Source1
/topmodule/Ins_Source2
/topmodule/Ins_Destiny
/topmodule/Data_In_Inst
/topmodule/Reg_Read1
/topmodule/Reg_Read2
/topmodule/Reg_Write
/topmodule/Alu_Result
/topmodule/Ram_Write_Data
/topmodule/Immediate
/topmodule/Temp1
/topmodule/Temp2
/topmodule/Reg_Write_Read
/topmodule/Flag_sum
/topmodule/Flag_Sub
/topmodule/Ram_Read
/topmodule/Ram_Write
Now 400 ns
Cursor 1 413 ns


Wave - Default
/topmodule/CLK
/topmodule/PC
/topmodule/PC_Temp
/topmodule/Opcode
/topmodule/Reg_Address1
/topmodule/Reg_Address2
/topmodule/Destiny
/topmodule/Ins_Source1
/topmodule/Ins_Source2
/topmodule/Ins_Destiny
/topmodule/Data_In_Inst
/topmodule/Reg_Read1
/topmodule/Reg_Read2
/topmodule/Reg_Write
/topmodule/Alu_Result
/topmodule/Ram_Write_Data
/topmodule/Immediate
/topmodule/Temp1
/topmodule/Temp2
/topmodule/Reg_Write_Read
/topmodule/Flag_sum
/topmodule/Flag_Sub
/topmodule/Ram_Read
/topmodule/Ram_Write
Now 400 ns
Cursor 1 413 ns
1413 ns


Wave - Default
/topmodule/MyRegisterFile/Read_Registe...
/topmodule/MyRegisterFile/Read_Registe...
/topmodule/MyRegisterFile/Write_Register
/topmodule/MyRegisterFile/Write_data
/topmodule/MyRegisterFile/Write
/topmodule/MyRegisterFile/Read_Data_1
/topmodule/MyRegisterFile/Read_Data_2
/topmodule/MyRegisterFile/Data_Matrix
(15)
(14)
(13)
(12)
(11)
(10)
(9)
(8)
(7)
(6)
(5)
(4)
(3)
(2)
(1)
(0)
Now 400 ns
Cursor 1 413 ns


Wave - Default
Msgs
/topmodule/MyRegisterFIle/Read_Registe...
/topmodule/MyRegisterFIle/Read_Registe...
/topmodule/MyRegisterFIle/Write_Register
/topmodule/MyRegisterFIle/Write_data
/topmodule/MyRegisterFIle/Write
/topmodule/MyRegisterFIle/Read_Data_1
/topmodule/MyRegisterFIle/Read_Data_2
/topmodule/MyRegisterFIle/Data_Matrix
Now 1400 ns
Cursor 1 1413 ns

خانه های رم را به صورت زیر مقدار دهی کردیم:

0 <= "0000001000000001" ,

1 <= "0011001100000001" ,

2 <= "0100010000000001" ,

3 <= "0110010100000001" ,

4 <= "1000000001111111" ,

5 <= "1011000000000001" ,

9 <= "1101010001110000" ,

10 <= "1100000000000011" ,

6 <= "1111000000000000" ,

others => "0000000000000000") ;

pc مقدار 0 دارد. پس خانه صفرم وارد instruction register میشود.

0000001000000001 یعنی عمل جمع (opcode 0000) انجام شود ، با مقادیر داخل رجیستر اول به ادرس 0000 و رجیستر دوم به ادرس 0001 که از ابتدا به ترتیب مقادیر

1111000011110000 و 1010101010101010 را به آن ها داده ایم و نتیجه جمع در رجیستر مقصد به ادرس 0010 ذخیره میشود.

پس دومین خانه register file به ادرس 0010 مقداری برابر 1001101110011010 خواهد داشت.

سپس pc به علاوه 1 میشود و برابر 1 میشود و خانه اول رم وارد instruction register میشود.

0011001100000001 یعنی دستور or بین رجیستر اول 0000 و رجیستر دوم 0001 و پاسخ در رجیستر 0011 ذخیره میشود یعنی مقدار این رجیستر برابر 1111101011111010 خواهد شد.

Pc به علاوه 1 خواهد شد . پس برابر 2 خواهد بود و خانه دوم رم وارد instruction register میشود.

دستور بعدی 0100010000000001 است که 0100 به معنی ضرب محتوای رجیستر 0000 و 0001 است و ذخیره نتیجه در 0100 است. نتیجه 1001111101100000 است.

Pc به علاوه 1 میشود و برابر 3 خواهد شد . پس خانه سوم رم وارد instruction register میشود. 0110010100000001

این دستور یعنی xor بین رجیستر 0000 و 0001 و ذخیره در 0101

مقدار 0101101001011010 در رجیستر مقصد ذخیره میشود.

Pc به علاوه 1 میشود و برابر 4 خواهد شد . پس خانه چهارم رم وارد instruction register میشود. 1000000001111111

این دستور addi است که محتوای ثبات با ادرس 0000 (که برابر 1111000011110000) است را با مقدار immediate بعد از sign extend کردن که برابر 0000000001111111 است جمع میکند در در همان ثبات ذخیره میکند.

(تغییر مقدار ثبات 0000 را به 1111000101101111 شاهد هستیم)

Pc به علاوه 1 میشود و برابر 5 خواهد شد . پس خانه پنجم رم وارد instruction register میشود. 1011000000000001

کد 1011 دستور bgt است که مقدار ثبات 0000 را با 0001 مقایسه میکند در صورتی که شرط برقرار شد (که در اینجا برقرار است چون ثبات اول دارای مقدار 1111000101101111 و ثبات دوم مقدار 1010101010101010 را دارد) دستور branch اجرا میشود که طبق صورت پروژه

Pc <= pc + 2 + (sign-extend(immediate)*2)

است که مقدار immediate بیت 7 تا 0 دستور است یعنی 00000001

Pc مقدار 10 را میگیرد و خانه دهم رم وارد instruction register میشود.

1101010001110000

1101 دستور load است و قرار است مقدار ثبات 0100 را گرفته (1111000101101111) با sign extend شده ی 0000 جمع میکند و این مقدار برابر ادرس خانه رم خواهد بود (چون رم 32 خانه دارد 5 بیت کم ارزش را به عنوان ادرس میگیریم) مقدار خانه مورد نظر از رم خوانده و داخل رجیستر به ادرس 0111 ذخیره میکند.

Pc به علاوه 1 میشود و برابر 11 خواهد شد . پس خانه یازدهم رم وارد instruction register میشود. 1100000000000011

1100 دستور jump است . طبق صورت پروژه

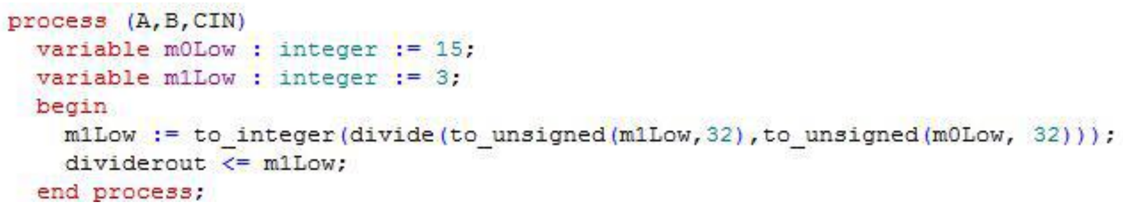
Pc <= pc (15 dwnto 13) && instruction (11 downto 0) && ‘0’

Pc مقدار 6 را میگیرد . خانه ششم رم را وارد instruction register میکند.

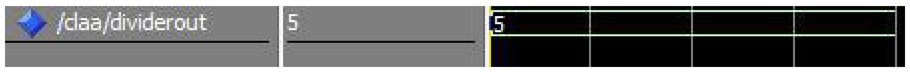
1111000000000000

دستور reset است و مقدار pc را صفر میکند.

دستور تقسیم هم به صورت جداگانه امتحان شد:



```
process (A,B,CIN)
  variable m0Low : integer := 15;
  variable m1Low : integer := 3;
  begin
    m1Low := to_integer(divide(to_unsigned(m1Low,32),to_unsigned(m0Low, 32)));
    dividerout <= m1Low;
  end process;
```



مقدار 15 تقسیم بر 3 نتیجه 5 را داده است.

کد نوشته شده در gitup آپلود شده و اگر شباهتی بین کد ما و بقیه مشاهده شد به ما ارتباطی ندارد.