Introduction to VHDL

ECE 5146/4146
Spring 2018
Instructor: Fareena Saqib (fsaqib@uncc.edu)

Project-1

Microcontroller design

Group-1

Swathi Thirunarayanan(801028736)

Dhivya Sivalingam(801045016)

Ryan Swaim(800589926)

Andrew Szoke(800793642)

Abstract

Microcontroller is defines as a computer which is present in a single integrated circuit. It is used to execute one specific application and also perform one task. It contains memory, programmable input/output peripherals as well a processor. Microcontrollers are mostly designed for embedded applications and are heavily used in automatically controlled electronic devices such as mobile phones, cameras, microwave ovens, washing machines, etc. Microcontroller usually uses four bit words and is designed for low power consumption. They operate at a low clock rate frequency. They are used in situations where limited computing functions are needed. Since the size and cost involved in designing the microcontroller is comparatively less than other methods, they are more economical to control electronic devices and processes. The processors used are said to be programmable state machines which are used in executing the programs, which provide a computer or other machine with coded instructions for the automatic performance of a particular task. In this design the microcontroller is designed which is completely embedded in the 8-bit RISC microcontroller. Registers, multiplexers, ALU and other required buses are designed in order to make the microcontroller compact and cost effective.

Introduction

Microcontroller (or microcontroller unit) is a small computer on a single integrated circuit. In modern terminology we can state it as a system on a chip which may include a microcontroller as one of its components. A microcontroller contains one or more CPUs (processor cores) along with memory and programmable input/output peripherals. In this project we have designed, simulated, and synthesized a behavioral embedded microcontroller. A microcontroller that is fully embedded 8-bit RISC microcontroller core is designed which is also made compact, capable, and cost-effective. It can be optimized for many FPGA families. In this design, we have focused to bring the most significant advantages that a processor can offer to a design environment at minimum cost. For this reason, these small processors have been considered to be "Programmable State Machines" and are referred to as "PSM". A PSM, like any other processor, will execute a program. A program is formed by a set of instructions that are defined by the user and held in a memory. Each instruction is encoded into a machine code.

Data bus and address bus, which can support 256 bytes of memory to hold both instructions and data, are included in the design with four registers such as R0 to R3, an Instruction Register IR, the Program Counter PC, and an 8-bit immediate register that holds immediate values. As the ALU only reads from and writes back to the registers data must be brought into registers for manipulation. Source register and both source and destination register are used as operands from ALU and the jump operations included in the design perform absolute jumps. Along with these components multiplexors which are placed in front of the registers determines where a register write comes from the ALU, the immediate register, another register or the data bus. Moreover other multiplexors like address bus multiplexor and PC multiplexor are also used in our design.

Components

The components used here are

- ALU
- Memory
- Stage Counter
- Instruction Register
- Immediate Register
- Program Counter
- 2x1 MUX(8 bit)
- 4x1 MUX(8 bit)
- Register
- Register File
- Decode Logic

In addition to these components we have designed few more components

- 1x4 DEMUX – used in Register File
- 4x1(1 bit) – used in Decode Logic
- 4x1(2 bit) – used in Decode Logic
- 16x1(1 bit) - used in Decode Logic
- 16x1(2 bit) - used in Decode Logic
- Jump Logic – used in Decode Logic

1) ALU

Arithmetic Logic Unit (ALU) is a critical component in a central processing unit CPU. ALU is a combinational circuit that can perform basic arithmetic and logic operations on a set of operands. Microprocessors have a dedicated module ALU for the arithmetic and logical operations. ALU is built first and its inputs are mapped to registers, stack or memory depending on the instruction set architecture (ISA).

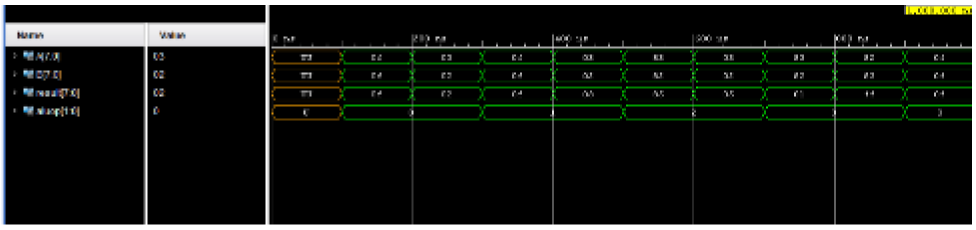
CODE:

```
*************************************************************************   //Entity    : ALU

//Description    :   ALU performs the four operations AND, OR, ADD, SUB, on two 8-bit values, and supports signed ADD  and SUB

//Inputs :   A,B(8 bit operands)

                         aluop(2 bit operator)

//Output          :   result(8 bit result)

*************************************************************************

--Library initialisation

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;


--entity ALU is declared

entity ALU is

port(

      A: in std_logic_vector(7 downto 0);        --8 bit operand 1 as in port

      B: in std_logic_vector(7 downto 0);        --8 bit operand 2 as in port

      aluop: in std_logic_vector(1 downto 0);           -- 2 bit operator as in port

      result: out std_logic_vector(7 downto 0));           --8 bit result as out port

end ALU;


--begin architecture

architecture Behavioral of ALU is

begin
```

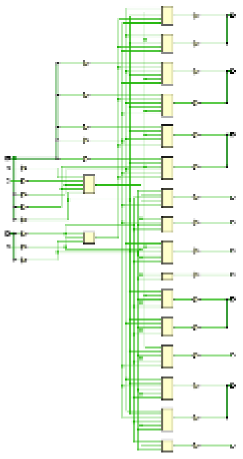
```vhdl
--operand1,operand2,operator are in sensitive list of process
process(A,B,aluop)

--variable temp of integer type is declared
variable temp1:signed(A'length downto 0);
variable temp2:signed(B'length downto 0);
variable tempr:signed(result'length downto 0);
begin

--switch case starts with operator as selection parameter
case aluop is
when "00" =>    --bitwise AND
result <= A and B;
when "01" =>   --bitwise OR
result <= A or B;
when "10" =>
temp1 := resize(signed(A),temp1'length);   --resize signed (A) to 9 bits
temp2 := resize(signed(B),temp2'length);   --resize signed (B) to 9 bits
tempr := temp1 + temp2;    --addition of signed(A)and signed(B)
result <= std_logic_vector(resize(signed(tempr),result'length)); --displaying the output bits after resizing it to 8 bits
when "11" =>
temp1 := resize(signed(A),temp1'length);  --resize signed (A) to 9 bits
temp2 := resize(signed(B),temp2'length);  --resize signed (B) to 9 bits
tempr := temp1 - temp2;   --subtraction of signed(A)and signed(B)
result <= std_logic_vector(resize(signed(tempr),result'length)); --displaying the output bits after resizing it to 8 bits
when others =>
result <= A;
end case;
end process;
```

end Behavioral;

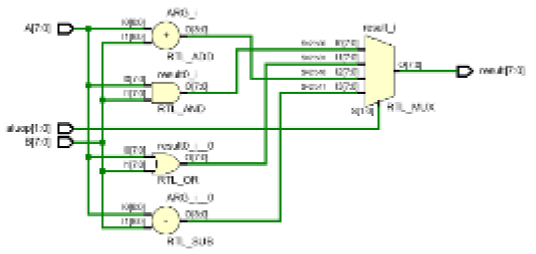
Waveform:



Synthesized Design:



Elaborated Design:



2) Memory

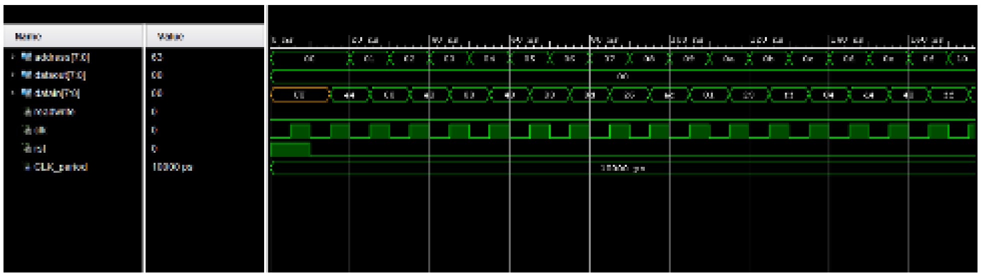
Memory refers to the computer hardware integrated circuits that store information for immediate use in a computer**.** Computer memory operates at a high speed and in our design we have used random-access memory (RAM). It is a semiconductor-based memory, where the CPU or the other hardware devices can read and write data. Though it is slow-to-access information it offers higher capacities. It temporality stores the data and it is a volatile memory. Once the system turns off, it loses the data .As a result RAM is used as a temporary data storage area.

CODE:

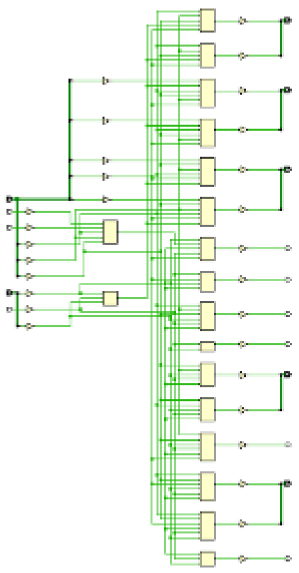
```
***************************************************************************  //Entity    : Memory

//Description    :   The RAM receives the address of the memory location from addressbus.  The address is stored in the form of binary numbers to enable the dataout bus to access memory storage. Based on the output of the readwrite the value in the particular address is sent out.

//Inputs :   address,dataout(8 bit)

           readwrite,clk,rst

//Output          :   datain(8 bit )

***************************************************************************

--Library initialisation

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;




--entity Memory is declared

entity Memory is

--  Port ( );

   port(

       address: in std_logic_vector(7 downto 0);    --8 bit output from mux

       dataout: in std_logic_vector(7 downto 0);   --8 bit output from dbus

       readwrite: in std_logic;                    --output from readwrite mux

       clk: in std_logic;

       rst: in std_logic;
```

```
        datain: out std_logic_vector(7 downto 0));  --8 bit datain is sent as output
end Memory;


--begin architecture
architecture RAM of Memory is
   type ram_type is array (0 to (2**address'length)-1) of std_logic_vector(datain'range);
   signal ramArray : ram_type;           --RAM is an array with the data ranging from 0 to 2^length of
                                           the address



begin


process(clk,rst) is
            begin
               if Rst = '1' then
               -- Clear Memory on Reset
        ramArray <= (others => (others => '0'));
               else
                         --as per the provided example in the manual
               ramArray(0) <= x"e4";
               ramArray(1) <= x"00";
               ramArray(2) <= x"e0";
               ramArray(3) <= x"80";
               ramArray(4) <= x"48";
               ramArray(5) <= x"88";
               ramArray(6) <= x"0d";
               ramArray(7) <= x"26";
               ramArray(8) <= x"ec";
               ramArray(9) <= x"01";
               ramArray(10) <= x"23";

               ramArray(11) <= x"ff";

               ramArray(12) <= x"04";

               ramArray(13) <= x"d4";

               ramArray(14) <= x"40";

               ramArray(15) <= x"ff";

               ramArray(16) <= x"0f";

               ramArray(128) <= x"02";

               ramArray(129) <= x"01";

               ramArray(130) <= x"00";

               if rising_edge(Clk) then

                           if readwrite = '0' then   --during the rising edge of the clock and when readwrite is 0 read the
data in the address and send it out

                           Datain <= ramArray(to_integer(unsigned(Address)));

                           else

                           ramArray(to_integer(unsigned(Address))) <= Dataout; --send the address out

                           end if;

                   end if;

                   end if;


     end process;

end RAM;
```

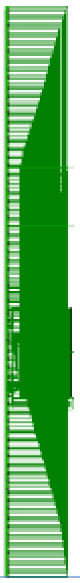
Waveform:



Synthesized Design:



Elaborated Design:



3) Stage Counter

The stage counter is used to output the current stage of execution. This is achieved with a simple 2- bit counter which is controlled by the clock cycle, and which outputs a 2-bit stage line.

CODE:

** //Entity : Phase Counter*

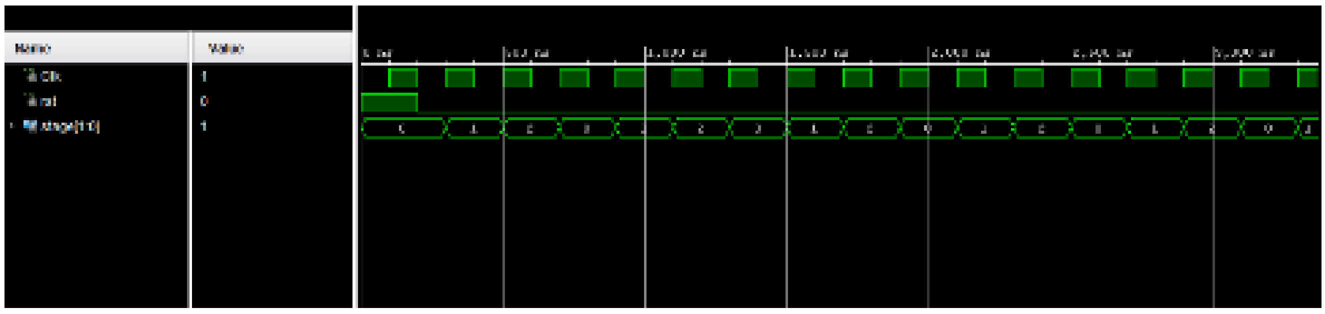
//Description : The stages are sent as output by incrementing the counter and the stage goes from 0 to 2. When the stage is 3 the counter is set back to zero

//Inputs : clk,rst

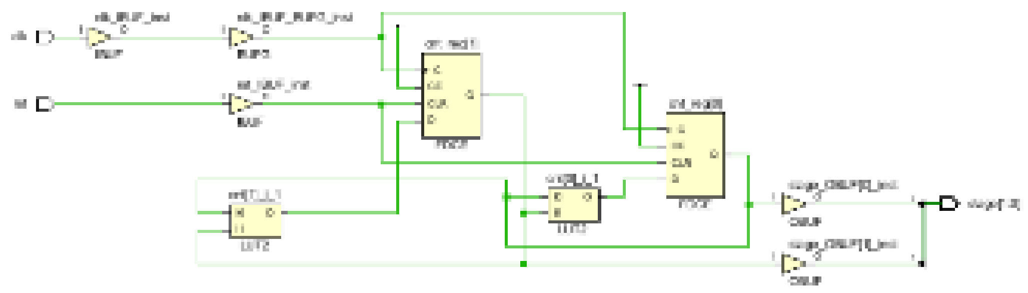
```
//Output: stage(2bit)
*************************************************************************


--Library initialisation
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library UNISIM;
use UNISIM.VComponents.all;


--entity phase_counter  is declared
entity phase_counter is
--  Port ();
port (
      clk: in std_logic;
      rst: in std_logic;
      stage: out std_logic_vector(1 downto 0));    --2 bit stage output


end phase_counter;
--begin architecture
architecture Behavioral of phase_counter is
signal cnt : std_logic_vector(1 downto 0);  --2 bit counter set as signal
signal resetsig : std_logic;
begin
  process(clk,rst)
            begin
                if(rst ='1')then
                            cnt <= "11"; --initializing the counter value to "11" when the reset is 1
```

```
                    else if(rising_edge(clk)) then   --check the counter value at each rising edge of clk
                    cnt <= cnt + 1;
          if(resetsig = '1') then
          cnt <= "00";  display cnt=”00”
          end if;
                 end if;
          end if;
          end process;
          resetsig <= cnt(1) and cnt(0); --AND the counter values
           stage <= cnt;
end Behavioral;
```

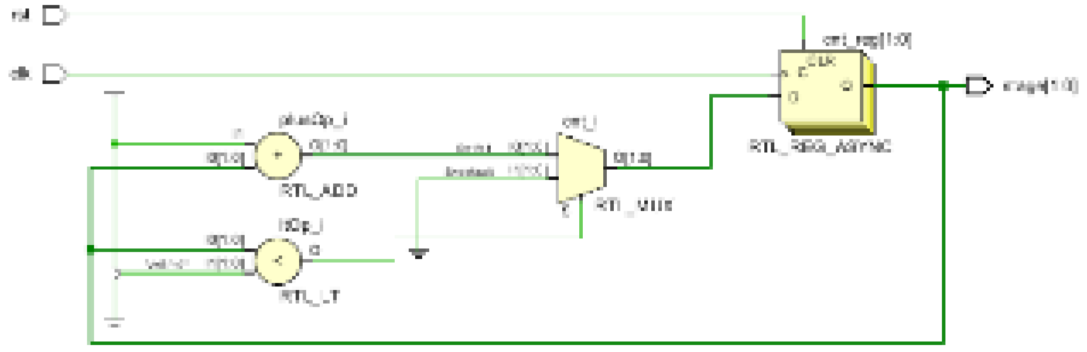
Waveform:



Synthesized Design:



Elaborated Design:



4) Instruction Register (IR)

An instruction register (IR) is the part of a CPU's control unit that holds the instruction currently being executed or decoded. In simple processors each instruction to be executed is loaded into the instruction register which holds it while it is decoded, prepared and ultimately executed.

CODE:

*** *//Entity : IR*

//Description : Irin,which is a 8 bit output from memory(datain), and the output from the irload mux is given as input. The IR should be loaded at stage 0,so the irload mux returns true(1) during the 0 input and false during other inputs. Irin(datain)is sent as output during rising clk and irload sends out true.

//Inputs : Irin (8 bit)

Irload

//Output: Irout (8bit)

****

--Library initialisation

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;


--entity IR is declared

entity IR is

port(

     Irin: in std_logic_vector(7 downto 0);  --8 bit output from memory(datain)

                                                    -- clk

     irload: in std_logic;                         --output from irload multiplexor

     Irout: out std_logic_vector(7 downto 0)); --8 bit output

end IR;


--begin architecture

architecture Behavioral of IR is


begin

process(irload)

begin

                                        --if (rising_edge(clk))

 if (irload = '1') then

irout <= irin;       --output from memory(datain) is sent out as irout when the

                                              condition is satisfied

end if;

end process;

end Behavioral;
```

5) Immediate Register

Immediate Register holds the immediate values. Immediate value is a piece of data that is stored as part of the instruction itself instead of being in a memory location or a register. Immediate values are typically used in instructions that load a value or performs an arithmetic or a logical operation on a constant.

CODE:

```
*************************************************************************** //Entity : Immediate_Register

//Description : Imin,which is a 8 bit output from memory(datain), and the output from the imload mux is given as input. The Immediate Register should be loaded with value from memory at stage 1,if irbit7(from IR)is true.So the imload line is made to be 1 by passing the irbit7 in input 1 of the mux and the Immediate Register is loaded from the datain bus.

//Inputs :   Imin (8 bit)

             Imload

//Output: Imout (8bit)

***************************************************************************

--Library initialisation

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;


--entity Immediate_Register is declared

entity Immediate_Register is

port(

   Imin: in std_logic_vector(7 downto 0);       --8 bit output from memory(datain)

                                                 -- clk

   imload: in std_logic;                          --output from imload multiplexor

   Imout: out std_logic_vector(7 downto 0));    ----8 bit output

end Immediate_Register;


--begin architecture

architecture Behavioral of Immediate_Register is

           --signal imout1 : std_logic_vector(7 downto 0);
```

```
begin
            --imout <= imin when rising_edge(clk) and imload = '1';
process(imload)
begin
            --if (rising_edge(clk))
 if (imload = '1') then
imout <= imin;   --output from memory(datain) is sent out as irout when the
                                                          condition is satisfied
end if;
end process;
             --imout <= imout1;

end Behavioral;
```

6) Program Counter (PC)

Program counter is a register in a computer processor that contains the address (location) of the instruction being executed at the current time. As each instruction gets fetched, the program counter increases its stored value by 1. After each instruction is fetched, the program counter points to the next instruction in the sequence. When the computer restarts or is reset, the program counter normally reverts to 0.

CODE:

```
*************************************************************************   //Entity    : PC
//Description    :    PC is incremented in order to fetch an immediate value in stage 1. PC is always incremented at stage 0. It is incremented at stage 1 if irbit7 is set. PC is loaded with an immediate value in stage 2 if we are performing a jump instruction and the jump test is true.
//Inputs :   pcin (8 bit)
              pcload,clk,rst
//Output: pcout (8bit)
*************************************************************************
```

```
--Library initialisation

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;


--entity PC is declared

entity PC is

port(

     pcin: in std_logic_vector(7 downto 0);  --8 bit output from 2x1pc mux with imout and PC adder as
                                                                      inputs

    clk : in std_logic;

     rst : in std_logic;

     pcload: in std_logic;                              --output from pcload mux

     pcout: out std_logic_vector(7 downto 0));  --8 bit output from PC

end PC;


--begin architecture

architecture Behavioral of PC is


begin

process(clk,rst)

begin

if(rst = '1') then

pcout <= x"00";                          --sends out the hexadecimal"00" when the reset is 1

elsif (rising_edge(clk) and pcload = '1') then

pcout <= pcin;                            -- sends the output from 2x1 pc mux during rising edge of clock and
                                                       when pcload is 1

end if;

end process;

end Behavioral;
```

7) 2x1 MUX

Multiplexer is a combinational logic circuit which is designed to send one of several input lines to a single common output line by the application of control logic. The 2x1 mux designed here is a generalized design which is used in many parts for the microcontroller design. It has 8 bit input lines and 8 bit output line with a control again provided by a multiplexor.

One of the 2x1MUX here is designed in such a way that it takes in the 8 bit outputs of Immediate Register(Imout) and PC Adder as two input pins. The control line is again obtained from 2x1 pcsel mux. The output obtained here is also 8 bits length. When the control line is 0 the first input from immediate register(Imout -8 bit output) is sent as output and when the control line changes to 1 the other other input from PC Adder is sent as output. This mux is placed in front of the Program Controller and the 8 bit output from the mux is given as one of the inputs(pcin) to the program controller.

CODE:

```
****************************************************************************   //Entity    :
s8_2x1_mux

//Description    :   Two 8 bit input lines are attached to the mux which produces an 8 bit output

                     based on the changes in the control line(which is again a output from a multiplexor)

//Inputs :   muxin2x1,muxin2x2 (8 bit)

             control_line2x1

//Output: muxout2x1 (8bit)

****************************************************************************

--Library initialisation

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;



--entity s8_2x1_mux is declared

entity s8_2x1_mux is

port(

    muxin2x1,muxin2x2: in std_logic_vector(7 downto 0);  --8 bit input

    control_line2x1: in std_logic;                       --output from a 2x1 mux

    muxout2x1: out std_logic_vector(7 downto 0));        -- 8 bit output

end s8_2x1_mux;

--begin architecture
```

```
architecture Behavioral of s8_2x1_mux is

begin

with control_line2x1 select muxout2x1 <= muxin2x1 when '0',  --when the control line is zero

                                                                             muxin2x1 is sent as output

                                muxin2x2 when others;                    --when control line is one it sends out

                                                                       muxin2x2

end Behavioral;
```

8)4X1 MUX

The 4x1 mux designed here is a generalized layout which is used in many parts for the microcontroller design. It has 8 bit input lines and an 8 bit output line with a control line that is again an output from a multiplexer. Muxen is provided in the design which helps in enabling the mux.

The address bus multiplexer along with the multiplexer in front of the registers uses the 4x1 logic.

(i)The address bus multiplexer has inputs from Program Counter(Pcload-8 bit),Immediate Register(Imout-8 bit),source register(4x1 mux which sends out sbus-8 bits) and destination register4x1 mux which sends out dbus-8 bits).The control line is from address select(which is of 2 bits) that selects the address from PC, Immediate Register and source register. The 8 bit output from the multiplexor is placed in the address bus.

(ii) The mux infront of the registers takes input from the Immediate register(Imout-8 bit), source register(4x1 mux which sends out sbus-8 bits),memory(datain-8 bit) and ALU(result-8 bit). Register select(2 bit control line) provides the control line to the multiplexer. Regsel select a value to write to the register from Immediate Register, memory and source register. The 8 bit output from the multiplexor is sent to the register.

CODE:

```
**************************************************************************  //Entity    :
s8_4x1_mux

//Description    :  Two 8 bit input lines are attached to the mux which produces an 8 bit output

                    based on the changes in the control line (which is again a output from multiplexor)

//Inputs :  muxin1,muxin2,muxin3,muxin4(8 bit input)

           control_line(2 bit)

//Output: muxout (8bit)

**************************************************************************

--Library initialisation

library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;

--entity s8_2x1_mux is declared

entity s8_4x1_mux is

port(

    muxin1,muxin2,muxin3,muxin4: in std_logic_vector(7 downto 0);  -- 8 bit input

    control_line: in std_logic_vector(1 downto 0);                --2 bit control line from a 4x1 mux

    muxout: out std_logic_vector(7 downto 0));                    --8 bit output

end s8_4x1_mux;

--begin architecture

architecture Behavioral of s8_4x1_mux is

begin

with control_line select muxout <= muxin4 when "11",  --depending on the values sent by the control

                        muxin2 when "01",             the corresponding inputs are sent out

                        muxin3 when "10",

                        muxin1 when others;


end Behavioral;
```

9) 4X1 DEMUX

CODE:

```
*************************************************************************  //Entity    : demux1x4

//Description    :   demuxin from the destination register write is given as input and depending on the drs control line
the output is passed to the registers

//Inputs :   demuxin , demuxctrl(2 bit)

//Output: demuxoutput1,demuxoutput2,demuxoutput3,demuxoutput4

*************************************************************************

--Library initialisation

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;
```

```vhdl
--entity demux1x4 is declared

entity demux1x4 is


port(

    demuxin: in std_logic;        --output from destination register write

    demuxctrl: in std_logic_vector(1 downto 0);     --2bit control line from dregsel

    demuxoutput1,demuxoutput2,demuxoutput3,demuxoutput4: out std_logic);

end demux1x4;


--begin architecture

architecture Behavioral of demux1x4 is

begin

process(demuxin, demuxctrl)

begin

--if(rising_edge(c)) then

case demuxctrl is

when "00" => demuxoutput1 <= demuxin; demuxoutput2 <= '0'; demuxoutput3 <= '0'; demuxoutput4 <='0';

when "01" => demuxoutput1 <='0'; demuxoutput2 <= demuxin; demuxoutput3 <='0'; demuxoutput4 <='0';

when "10" => demuxoutput1 <='0'; demuxoutput2 <='0'; demuxoutput3 <= demuxin; demuxoutput4 <='0';

when others => demuxoutput1 <='0'; demuxoutput2 <='0'; demuxoutput3 <='0'; demuxoutput4 <= demuxin;

end case;

--end if;

end process;

end Behavioral;
```

10) Register

CODE:

```
**************************************************************************    //Entity     : registerfs

//Description     :   During the rising edge of the clock and based  on the enable value from the demux, the 8 bit output is sent out from the register

//Inputs :   din (8 bit)

             clk,enable

//Output: dout (8bit)

            regsig

**************************************************************************

--Library initialisation

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;


--entity registerfsis declared

entity registerfs is

--  Port ( );

port(

    clk : in std_logic;

    enable: in std_logic;

    din: in std_logic_vector(7 downto 0); --8 bit output from dval

    dout : out std_logic_vector(7 downto 0)); --8 bit output from register);

    end registerfs;

--begin architecture

architecture Behavioral of registerfs is


begin

process(clk)
```

```
begin
if(rising_edge(clk)) then
if(enable = '1') then
dout <= din;  --8 bit output from the registers are obtained when the above condition is met
end if;
end if;
end process;
end Behavioral;
```

11) Register File

There are four registers and each register's value is sent to two multiplexors . One multiplexor selects one of the register values based on the dregsel control line (drs), and outputs one value on the dbus line to the ALU and the other multiplexor selects one of the register values based on the sregsel control line, and outputs one value on the sbus line to the ALU. The eight bits on the dbus are split out, and the most significant bit is the Negative output. In addition to this all eight bits of the dbus are ORed together and then negated.

CODE:

```
************************************************************************* //Entity : Register_file

//Description : registerfs,two 4x1 mux and 4x1 dmux are used as components along with dwrite,dval,dregsel,sregsel and clock. These are used as inputs for the register file design to produce 8 bit output in dbus and sbus. The eight bits in the dbus are split out to find the most significant bit(negative output)and the bits are ORed and negated.if the output is zero,the output on the zero line is true.

//Inputs :   dval (8 bit)

             dregsel,sregsel(2 bit)

             clock,dwrite

//Output: dbus,sbus(8bit)

             zero,negative

*************************************************************************
```

```
--Library initialisation
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--entity registerfs  is declared
entity Register_file is
port(
    clock  : in std_logic;
    dwrite: in std_logic;
    dval: in std_logic_vector(7 downto 0);
    dregsel,sregsel: in std_logic_vector(1 downto 0);
    dbus,sbus : out std_logic_vector(7 downto 0);
    zero,negative : out std_logic);
end Register_file;
--begin architecture
architecture Behavioral of Register_file is

component registerfs
port(
    clk : in std_logic;
    enable: in std_logic;
    din: in std_logic_vector(7 downto 0);
    dout : out std_logic_vector(7 downto 0);
    regsig : out std_logic);
end component;

component s8_4x1_mux
```

```
port(
      muxen : in std_logic;
      muxin1,muxin2,muxin3,muxin4: in std_logic_vector(7 downto 0);
      control_line: in std_logic_vector(1 downto 0);
      muxout: out std_logic_vector(7 downto 0));
end component;

component demux1x4 is
port(
      demuxin: in std_logic;
      demuxctrl: in std_logic_vector(1 downto 0);
      demuxoutput1,demuxoutput2,demuxoutput3,demuxoutput4: out std_logic);
end component;

signal rwrite0,rwrite1,rwrite2,rwrite3 : std_logic;
signal regsig : std_logic:='0';
signal dval0,dval1, dval2,dval3,dmux1,dmux2,dmux3,dmux4,dbussig,sbussig: std_logic_vector(7 downto 0);

begin

d: demux1x4 port map(demuxin => dwrite, demuxctrl => dregsel, demuxoutput1 => rwrite0, demuxoutput2 =>
rwrite1, demuxoutput3 => rwrite2, demuxoutput4 => rwrite3);
--the registers are selected based on the value from dval
dval0 <= dval when dregsel ="00" else x"00";
dval1 <= dval when dregsel = "01" else x"00";
dval2 <= dval when dregsel = "10" else x"00";
dval3 <= dval when dregsel = "11" else x"00";
```

r0: registerfs port map(clk => clock, enable => rwrite0, din => dval0, dout => dmux1,regsig => regsig);

r1: registerfs port map(clk => clock, enable => rwrite1, din => dval1, dout => dmux2, regsig => regsig);

r2: registerfs port map(clk => clock, enable => rwrite2, din => dval2, dout => dmux3, regsig => regsig);

r3: registerfs port map(clk => clock, enable => rwrite3, din => dval3, dout => dmux4, regsig => regsig);

--*first mux selects the registers based on the control line dregsel and sends out the value to dbus*

m0: s8_4x1_mux port map(muxen =>regsig,muxin1 => dmux1,muxin2 => dmux2, muxin3 => dmux3, muxin4 => dmux4, control_line=>dregsel,muxout =>dbussig);

--*second mux selects the registers based on the control line sregsel and sends out the value to sbus*

m1: s8_4x1_mux port map(muxen =>regsig,muxin1 => dmux1,muxin2 => dmux2, muxin3 => dmux3, muxin4 => dmux4, control_line=>sregsel,muxout =>sbussig);

--*all the 8 bits placed on the dbus is ORed and negated to check if it is zero*

zero <= not (dbussig(0) or dbussig(1) or dbussig(2) or dbussig(3) or dbussig(4) or dbussig(5) or dbussig(6) or dbussig(7));

--*the most significant bit(7) is the negative output*

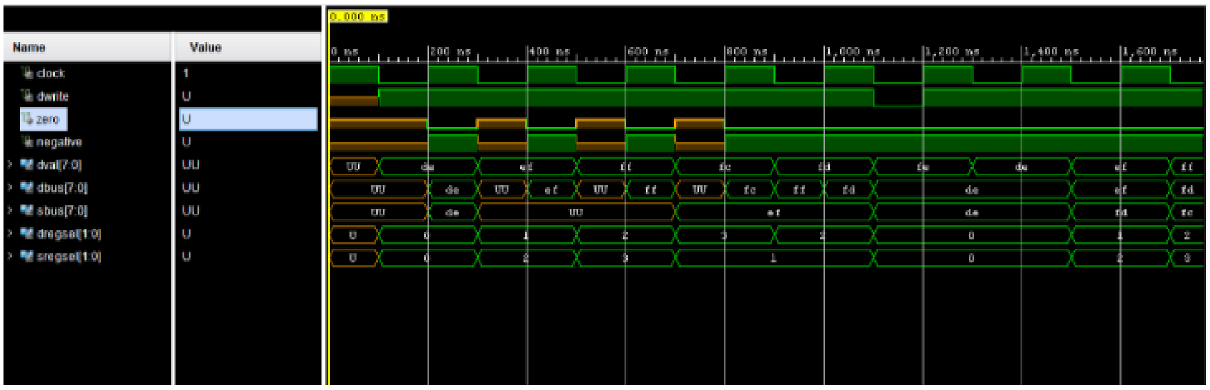
negative <= dbussig(7);

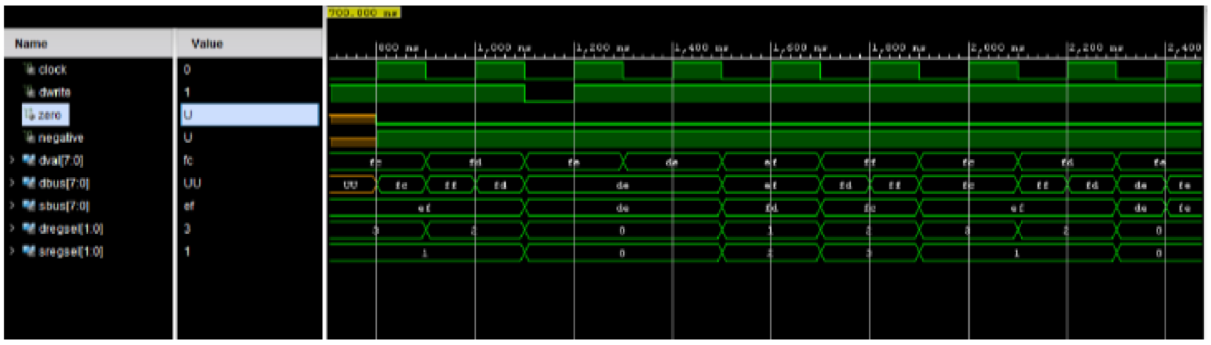
dbus <= dbussig;

sbus <= sbussig;

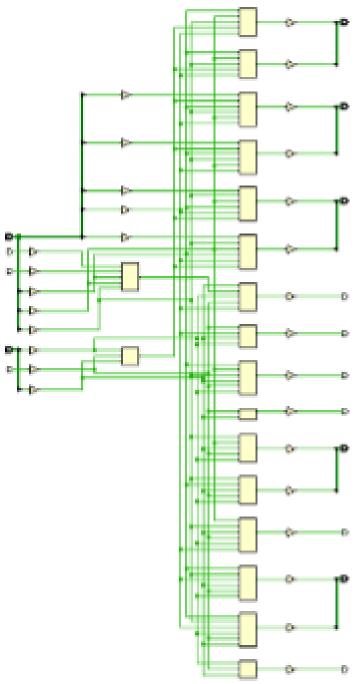
end Behavioral;

Waveform:

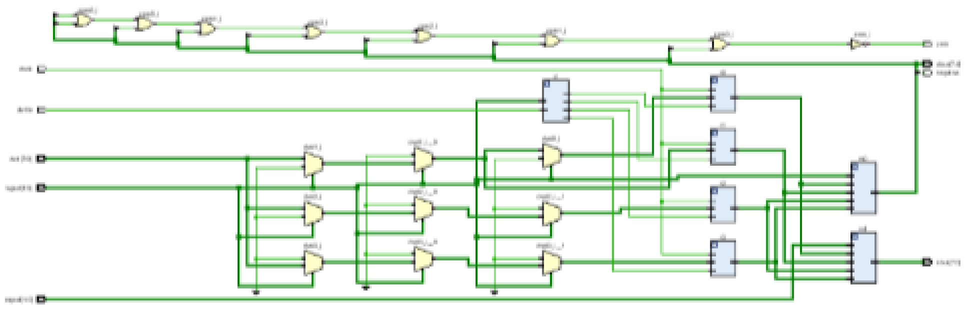




Synthesized Design:



Elaborated Design:



12) Decode Logic

In the Decode Logic block, the value from the Instruction Register is split into individual lines irbit4, irbit5, irbit6 and irbit7. The operands op1 and op2 are split out and op2 exported as aluop. The 4 opcode bits from the instruction are split out as the op1op2 line.

CODE:

```
************************************************************************* //Entity : decode_logic

//Description : Decode Logic has input values from the Instruction Register, and the zero & negative lines of the destination register. The values for control lines are generated by using these inputs

//Inputs :   instruction(8 bit)

           stage(2 bit)

           zero,negative,rst1

//Output:   addrsel , regsel  , aluop , sregsel  , dregsel(2 bit)

Irload,imload    , dwrite  , readwrite  , pcsel ,pcload

*************************************************************************

--Library initialisation

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;


--entity decode_logic is declared

entity decode_logic is

   Port (

         instruction    :     in  STD_LOGIC_VECTOR (7 downto 0); --output from IR register

         zero          :     in  STD_LOGIC; --output from register file

         negative       :     in  STD_LOGIC;  --output from register file

         rst1          :     in  STD_LOGIC;

         stage         :     in  STD_LOGIC_VECTOR (1 downto 0); --stages from stage counter
```

```
        addrsel      :     out STD_LOGIC_VECTOR (1 downto 0);  --2 bit output from addrsel
        irload       :     out STD_LOGIC;
        imload       :     out STD_LOGIC;
        regsel       :     out STD_LOGIC_VECTOR (1 downto 0);  ----2 bit output from regsel
        dwrite       :     out STD_LOGIC;
        aluop        :     out STD_LOGIC_VECTOR (1 downto 0);
        readwrite    :     out STD_LOGIC;
        pcsel        :     out STD_LOGIC;
        pcload       :     out STD_LOGIC;
        sregsel      :     out STD_LOGIC_VECTOR (1 downto 0); --2 bit output
        dregsel      :     out STD_LOGIC_VECTOR (1 downto 0));  --2 bit output
end decode_logic;
--begin architecture
architecture Behavioral of decode_logic is


component s1_4x1_mux
port(
     rstm1 : in std_logic;
     muxins11,muxins12,muxins13,muxins14: in std_logic;
    control_lines1: in std_logic_vector(1 downto 0); --2 bit input control line
    muxouts1: out std_logic);
end component;


component s2_4x1_mux
port(
    rstm2 : in std_logic;
    muxins21,muxins22,muxins23,muxins24: in std_logic_vector(1 downto 0);
    control_lines2: in std_logic_vector(1 downto 0);
    muxouts2: out std_logic_vector(1 downto 0));
```

```
end component;


--this is the jump logic for the pcload

component jump_logic

Port (

            z,rst               :       in      std_logic;

            neg             :       in      std_logic;

            ibit4           :       in      std_logic;

            ibit5           :       in      std_logic;

            ibit6           :       in      std_logic;

            ibit7           :       in      std_logic;

            op              :       in      std_logic_vector (1 downto 0);

            jmp_logic_out    :       out     std_logic

         );

end component;


component s2_16x1_mux

port(

      rstm4 : in std_logic;


mux16ins21,mux16ins22,mux16ins23,mux16ins24,mux16ins25,mux16ins26,mux16ins27,mux16ins28,mux16ins29,m
ux16ins210,mux16ins211,mux16ins212,mux16ins213,mux16ins214,mux16ins215,mux16ins216: in
std_logic_vector(1 downto 0);

      control_line16s2: in std_logic_vector(3 downto 0);

      mux16outs2: out std_logic_vector(1 downto 0));

end component;


component s1_16x1_mux

--  Port ( );

port(
      rstm3 : in std_logic;

mux16ins11,mux16ins12,mux16ins13,mux16ins14,mux16ins15,mux16ins16,mux16ins17,mux16ins18,mux16ins19,m
ux16ins110,mux16ins111,mux16ins112,mux16ins113,mux16ins114,mux16ins115,mux16ins116: in std_logic;

      control_line16s1: in std_logic_vector(3 downto 0);

      mux16outs1: out std_logic);

end component;


signal jmp_logic   :   std_logic;

signal Rd, Rs, op1, op2, reg_big_mux_out , addr_big_mux_out,aluop_out,sregsel_out,dregsel_out :
std_logic_vector(1 downto 0);

signal rw_and, dwrite_big_mux_out   :   std_logic;

signal op1op2 : std_logic_vector (3 downto 0);




begin

op1     <=      instruction(7 downto 6);

op2     <=      instruction(5 downto 4);

Rd<= instruction (3 downto 2);

Rs <= instruction( 1 downto 0);


rw_and  <=      instruction(4)  and (not instruction(5)) and instruction(6);

op1op2(3 downto 2) <= op1;

op1op2(1 downto 0) <= op2;

--mapping and assigning the values to the components

--irload mux

ir_load_logic  : s1_4x1_mux port map(rstm1 => rst1,muxins11 => '1', muxins12 => '0', muxins13 => '0', muxins14 =>
'0', control_lines1 => stage, muxouts1 => irload);


--imload mux
```

```
immed_load_logic : s1_4x1_mux port map (rstm1 => rst1,muxins11 => '0', muxins12 => instruction(7), muxins13 =>
'0', muxins14 => '0', control_lines1 => stage, muxouts1 => imload);


--pcload mux with jump logic

pc_load     : s1_4x1_mux port map (rstm1 => rst1,muxins11 => '1', muxins12 => instruction(7), muxins13 =>
jmp_logic, muxins14 => '0', control_lines1 => stage, muxouts1 => pcload);


--jump logic for pc load

jmp_logic1   :   jump_logic port map (rst => rst1,z => zero, neg => negative, ibit4 => instruction(4), ibit5 =>
instruction(5)

                , ibit6 => instruction(6), ibit7 => instruction(7), op => op2, jmp_logic_out => jmp_logic);

--readwrite mux

read_write_logic  :   s1_4x1_mux port map(rstm1 => rst1,muxins11 => '0', muxins12 => '0', muxins13 => rw_and,
muxins14 => '0', control_lines1 => stage, muxouts1 => readwrite);


--regsel mux

reg_sel      :   s2_4x1_mux port map      (rstm2 => rst1,muxins21 => "00", muxins22 => "00", muxins23 =>
reg_big_mux_out, muxins24 => "00", control_lines2 => stage, muxouts2 => regsel);


-- regsel_big_mux

regsel_big_mux  :   s2_16x1_mux port map (rstm4 => rst1,mux16ins21 => "11", mux16ins22 => "11", mux16ins23 =>
"11", mux16ins24 => "11", mux16ins25 => "10", mux16ins26 => "00", mux16ins27 => "01", mux16ins28 => "00",
mux16ins29 => "00",

                                    mux16ins210 => "00", mux16ins211 => "00", mux16ins212 => "00", mux16ins213 => "10",
mux16ins214 => "00", mux16ins215 => "00", mux16ins216 => "00", control_line16s2 => op1op2 , mux16outs2 =>
reg_big_mux_out);


--addrsel mux

address_sel     :   s2_4x1_mux port map(rstm2 => rst1,muxins21 => "00", muxins22 => "00", muxins23 =>
addr_big_mux_out, muxins24 => "00", control_lines2 => stage, muxouts2 => addrsel);

--addr big mux

addr_big_mux    :   s2_16x1_mux port map  (rstm4 => rst1,mux16ins21 => "00", mux16ins22 => "00", mux16ins23
=> "00", mux16ins24 => "00", mux16ins25 => "10", mux16ins26 => "10", mux16ins27 => "00", mux16ins28 =>
"00", mux16ins29 => "00",
                                    mux16ins210 => "00", mux16ins211 => "00", mux16ins212 => "00", mux16ins213 => "01",
mux16ins214 => "01", mux16ins215 => "00", mux16ins216 => "00", control_line16s2 => op1op2 , mux16outs2 =>
addr_big_mux_out);



--dwrite

dwrite_logic    :   s1_4x1_mux port map     (rstm1 => rst1,muxins11 => '0', muxins12 => '0', muxins13 =>
dwrite_big_mux_out, muxins14 => '0', control_lines1 => stage, muxouts1 => dwrite);



--dwrite_big_mux

dwrite_big_mux  :   s1_16x1_mux port map  (rstm3 => rst1,mux16ins11 => '1', mux16ins12 => '1', mux16ins13 => '1',
mux16ins14 => '1', mux16ins15 => '1', mux16ins16 => '0', mux16ins17 => '1', mux16ins18 => '0', mux16ins19 => '0',

                                    mux16ins110 => '0', mux16ins111 => '0', mux16ins112 => '0', mux16ins113 => '1',
mux16ins114 => '0', mux16ins115 => '1', mux16ins116 => '0', control_line16s1 => op1op2 , mux16outs1 =>
dwrite_big_mux_out);

--aluop mux

aluop_logic     :  s2_4x1_mux port map(rstm2 => rst1,muxins21 => "00", muxins22 => "00", muxins23 => aluop_out,
muxins24 => "00", control_lines2 => stage, muxouts2 => aluop);

--aluop big mux

aluop_bigmux    : s2_16x1_mux port map  (rstm4 => rst1,mux16ins21 => op2, mux16ins22 => op2, mux16ins23 =>
op2, mux16ins24 => op2, mux16ins25 => "00", mux16ins26 => "00", mux16ins27 => "00", mux16ins28 => "00",
mux16ins29 => op2,

                                    mux16ins210 => op2, mux16ins211 => op2, mux16ins212 => op2, mux16ins213 => "00",
mux16ins214 => "00", mux16ins215 => "00", mux16ins216 => "00", control_line16s2 => op1op2 , mux16outs2 =>
aluop_out);

--sregsel mux

sregsel_logic     :  s2_4x1_mux port map(rstm2 => rst1,muxins21 => "00", muxins22 => "00", muxins23 =>
sregsel_out, muxins24 => "00", control_lines2 => stage, muxouts2 => sregsel);

--sregsel big mux

sregsel_bigmux    : s2_16x1_mux port map  (rstm4 => rst1,mux16ins21 => Rs, mux16ins22 => Rs, mux16ins23 =>
Rs, mux16ins24 => Rs, mux16ins25 => Rs, mux16ins26 => Rs, mux16ins27 => Rs, mux16ins28 => "00",
mux16ins29 => "00",

                                    mux16ins210 => "00", mux16ins211 => "00", mux16ins212 => "00", mux16ins213 => "00",
mux16ins214 => "00", mux16ins215 => "00", mux16ins216 => "00", control_line16s2 => op1op2 , mux16outs2 =>
sregsel_out);

--dregsel mux
```

```
dregsel_logic     :  s2_4x1_mux port map(rstm2 => rst1,muxins21 => "00", muxins22 => "00", muxins23 => dregsel_out, muxins24 => "00", control_lines2 => stage, muxouts2 => dregsel);



--dregsel mux

dregsel_bigmux    : s2_16x1_mux port map  (rstm4 => rst1,mux16ins21 => Rd, mux16ins22 => Rd, mux16ins23 => Rd, mux16ins24 => Rd, mux16ins25 => Rd, mux16ins26 => Rd, mux16ins27 => Rd, mux16ins28 => "00", mux16ins29 => Rd, mux16ins210 => Rd, mux16ins211 => Rd, mux16ins212 => Rd, mux16ins213 => Rd, mux16ins214 => Rd, mux16ins215 => Rd, mux16ins216 => "00", control_line16s2 => op1op2 , mux16outs2 => dregsel_out);

--pcsel mux

pcsel_logic       : s1_4x1_mux port map     (rstm1 => rst1, muxins11 => '1', muxins12 => '1', muxins13 => '0', muxins14 => '0', control_lines1 => stage, muxouts1 => pcsel);



end Behavioral;
```

The components used in this decode logic are

(i) s1_4x1_mux: rstm1(reset) ,muxins11,muxins12,muxins13,muxins14,control_lines1 are the inputs and muxouts1 is the output. If the rstml is set to ‘1’ and depending on the values of the control line the outputs are generated.

(ii) s2_4x1_mux: rstm2(reset) , 2 bits - muxins21,muxins22,muxins23,muxins24, control_lines2 are the inputs and 2 bit muxouts2 is the output. If the rstm2 is set to ‘1’ and depending on the values of the control line the outputs are generated.

(iii) jump_logic: z,rst , neg ,ibit4, ibit5,ibit6,ibit7,op (2 bit) are the inputs and jmp_logic_out is the output . The jump logic is used for the pc load.

CODE

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;

entity jump_logic is

Port (  rst             :   in      std_logic;

        z               :   in      std_logic;

        neg             :   in      std_logic;
        ibit4           :   in      std_logic;

        ibit5           :   in      std_logic;

        ibit6           :   in      std_logic;

        ibit7           :   in      std_logic;

        op              :   in      std_logic_vector (1 downto 0);

        jmp_logic_out   :   out     std_logic

         );

end jump_logic;

architecture Behavioral of jump_logic is

component s1_4x1_mux is

port(

    rstm1 : in std_logic;

    muxins11,muxins12,muxins13,muxins14: in std_logic;

      control_lines1: in std_logic_vector(1 downto 0);

      muxouts1: out std_logic);

end component;

signal not_z, z_and_n, jmp_mux_out, and1_out, and2_out : std_logic;

begin

--mapping and assigning values t the component

jmp_mux : s1_4x1_mux  port map (rstm1 => rst, muxins11 => z, muxins12 => not_z, muxins13 => z_and_n,
muxins14 => neg, control_lines1 => op, muxouts1 => jmp_mux_out);

--jump conditions

not_z   <= not z;

z_and_n <= (not z) and (not neg);

and1_out <= (not ibit6) and ibit7 and jmp_mux_out;

and2_out <= ibit4 and ibit5 and ibit6 and ibit7;

jmp_logic_out <= and1_out or and2_out;

end Behavioral;
```

(iv) s2_16x1_mux: rstm4(reset), 2 bit-mux16ins21,mux16ins22,mux16ins23,mux16ins24,mux16ins25,mux16ins26,mux16ins27,mux16ins28,mux16ins29,mux16ins210,mux16ins211,mux16ins212,mux16ins213,mux16ins214,mux16ins215,mux16ins216,3bit-control_line16s2 are the inputs and 2 bit- mux16outs2 is the output. If the rstm4 is set to ‘1’ and depending on the values of the control line the outputs are generated.

(v) s1_16x1_mux: rstm3(reset,)mux16ins11,mux16ins12,mux16ins13,mux16ins14,mux16ins15,mux16ins16,mux16ins17,mux16ins18,mux16ins19,mux16ins110,mux16ins111,mux16ins112,mux16ins113,mux16ins114,mux16ins115,mux16ins116,3bit-control_line16s1 are the inputs and mux16outs1is the output. If the rstm3 is set to ‘1’ and depending on the values of the control line the outputs are generated.

Working

CPU internally h as three stages for the execution of each instruction and the stage counter outputs the current stage of execution. CPU has several 1-bit ,2-bit control lines and a 3-bit control line, which help in the working of the multiplexors.

STAGE 0:

In this stage the instruction is taken from the memory and stored in the Instruction.Register. Initially PC value is set to zero. The address select line is kept as 0 in order to loads the value of the PC in address bus. This address is sent to RAM. The value in the address is placed in datain. The irload line is set as 1 so that IR is loaded from datain. After this PC is incremented.

STAGE 1:

The incremented PC value is sent to RAM. At this stage we need to load the Immediate Register with the value from memory when irbit7 from IR is made true. Also the Immediate Register is loaded with the value in the datain bus by making the imload line to be 1The Decode Logic is used to generate the values to the control lines. It gets its input from Instruction Register,zero and negative lines from the destination register.

STAGE 2:

Depending on the instructions we have specified , the control lines function accordingly.The imload and irload are always zero in this stage.dregsel is connected to the destination register while sregsel is connected to source register and aluop to op2.

Microcontroller Design

CODE:

```
************************************************************************** //Entity : Microcontroller

//Description : The components designed earlier are mapped to their respective signals to produce the desired output

//Inputs : main_clock,main_reset

//Output: alu_out(8 bit)

**************************************************************************

--Library initialisation

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;

use IEEE.NUMERIC_STD.ALL;


entity Microcontroller is

-- Port ( );

port(

    main_clock : in std_logic;

    main_reset : in std_logic;

    alu_out    : out std_logic_vector(7 downto 0));

end Microcontroller;


architecture Behavioral of Microcontroller is

component phase_counter is

-- Port ();

port (

    clk: in std_logic;

    rst: in std_logic;
```

```
      stage: out std_logic_vector(1 downto 0));
end component;

component s8_2x1_mux is
--  Port ( );
port(
      muxin2x1,muxin2x2: in std_logic_vector(7 downto 0);
      control_line2x1: in std_logic;
      muxout2x1: out std_logic_vector(7 downto 0));
end component;

component PC is
--  Port ( );
port(
     pcin: in std_logic_vector(7 downto 0);
     clk : in std_logic;
     rst : in std_logic;
     pcload: in std_logic;
     pcout: out std_logic_vector(7 downto 0));
end component;

component ALU is
port(
        A: in std_logic_vector(7 downto 0);
        B: in std_logic_vector(7 downto 0);
        aluop: in std_logic_vector(1 downto 0);
        result: out std_logic_vector(7 downto 0));
end component;
```

```
component s8_4x1_mux is
--  Port ( );
port(
      muxin1,muxin2,muxin3,muxin4: in std_logic_vector(7 downto 0);
      control_line: in std_logic_vector(1 downto 0);
      muxout: out std_logic_vector(7 downto 0));
end component;

component decode_logic is
    Port (
            instruction     :       in  STD_LOGIC_VECTOR (7 downto 0); --output from IR register
            zero            :       in  STD_LOGIC; --output from registefiler
            negative        :       in  STD_LOGIC;  --output from register file
            rst1            :       in  STD_LOGIC;
            stage           :       in  STD_LOGIC_VECTOR (1 downto 0); --stage output
            addrsel         :       out STD_LOGIC_VECTOR (1 downto 0);
            irload          :       out STD_LOGIC;
            imload          :       out STD_LOGIC;
            regsel          :       out STD_LOGIC_VECTOR (1 downto 0);
            dwrite          :       out STD_LOGIC;
            aluop           :       out STD_LOGIC_VECTOR (1 downto 0);
            readwrite       :       out STD_LOGIC;
            pcsel           :       out STD_LOGIC;
            pcload          :       out STD_LOGIC;
            sregsel         :       out STD_LOGIC_VECTOR (1 downto 0);
            dregsel         :       out STD_LOGIC_VECTOR (1 downto 0));
end component;

component Register_file is
```

```
--  Port ( );
port(
     clock  : in std_logic;
     dwrite: in std_logic;
     dval: in std_logic_vector(7 downto 0);
     dregsel,sregsel: in std_logic_vector(1 downto 0);
     dbus,sbus : out std_logic_vector(7 downto 0);
     zero,negative : out std_logic);
end component;

component IR is
--  Port ( );
port(
     Irin: in std_logic_vector(7 downto 0);
                          --clk
     irload: in std_logic;
     Irout: out std_logic_vector(7 downto 0));
end component;

component Immediate_Register is
--  Port ( );
port(
     Imin: in std_logic_vector(7 downto 0);
                          -- clk
     imload: in std_logic;
     Imout: out std_logic_vector(7 downto 0));
end component;

component Memory is
```

```
--  Port ( );
    port(
        address: in std_logic_vector(7 downto 0);
        dataout: in std_logic_vector(7 downto 0);
        readwrite: in std_logic;
        clk: in std_logic;
        rst: in std_logic;
        datain: out std_logic_vector(7 downto 0));
end component;


--signals for the microcontroller are defined
signal pcinsig,pcoutsig,immedsig, pccounter,sbussig,dbussig,addroutsig,datainsig,iroutsig,regoutsig,aluoutsig:
std_logic_vector(7 downto 0);
signal pcloadsig, pcselsig,readwritesig,irloadsig,imloadsig,dwritesig,zerosig,negativesig : std_logic;
signal addrselsig,regselsig,dregselsig,sregselsig,aluopsig,stage_outsig : std_logic_vector(1 downto 0);


begin
--each component the microcontroller are mapped to their respective signals
pccounter <= std_logic_vector(unsigned(pcoutsig) + 1);
mainpc: PC port map(clk => main_clock, rst=> main_reset , pcin => pcinsig  , pcload => pcloadsig , pcout =>
pcoutsig);
pc_mux: s8_2x1_mux port map(muxin2x1 => immedsig, muxin2x2 => pccounter , control_line2x1 => pcselsig,
muxout2x1 => pcinsig);
address_mux : s8_4x1_mux port map(muxin1 => pcoutsig, muxin2 => immedsig, muxin3 => sbussig , muxin4 =>
dbussig ,control_line => addrselsig, muxout => addroutsig);
mainmemory : memory port map(address => addroutsig, dataout => dbussig, readwrite => readwritesig , clk =>
main_clock, rst => main_reset, datain => datainsig);
mainir : IR port map(irin => datainsig, irout => iroutsig, irload => irloadsig);
mainim : Immediate_Register port map(imin => datainsig, imload => imloadsig, imout => immedsig);
register_mux : s8_4x1_mux port map(muxin1 => immedsig, muxin2 => sbussig, muxin3 => datainsig , muxin4 =>
aluoutsig,control_line => regselsig, muxout => regoutsig);
```

mainregister: Register_file port map(clock => main_clock, dwrite => dwritesig, dval => regoutsig, dregsel => dregselsig , sregsel => sregselsig, dbus => dbussig, sbus => sbussig, zero => zerosig , negative => negativesig);

mainalu : ALU port map(a => dbussig, b => sbussig , aluop => aluopsig ,result => aluoutsig);

decode : decode_logic port map(instruction => iroutsig, zero => zerosig , negative => negativesig , rst1 => main_reset, stage => stage_outsig, addrsel => addrselsig, irload => irloadsig, imload => imloadsig, regsel => regselsig, dwrite => dwritesig, aluop => aluopsig, readwrite => readwritesig, pcsel => pcselsig, pcload => pcloadsig, sregsel => sregselsig, dregsel => dregselsig);

stage : phase_counter port map(clk => main_clock, rst => main_reset, stage => stage_outsig);

alu_out <= aluoutsig;

end Behavioral;

Analysis and Output:

1) Data is loaded into the memory

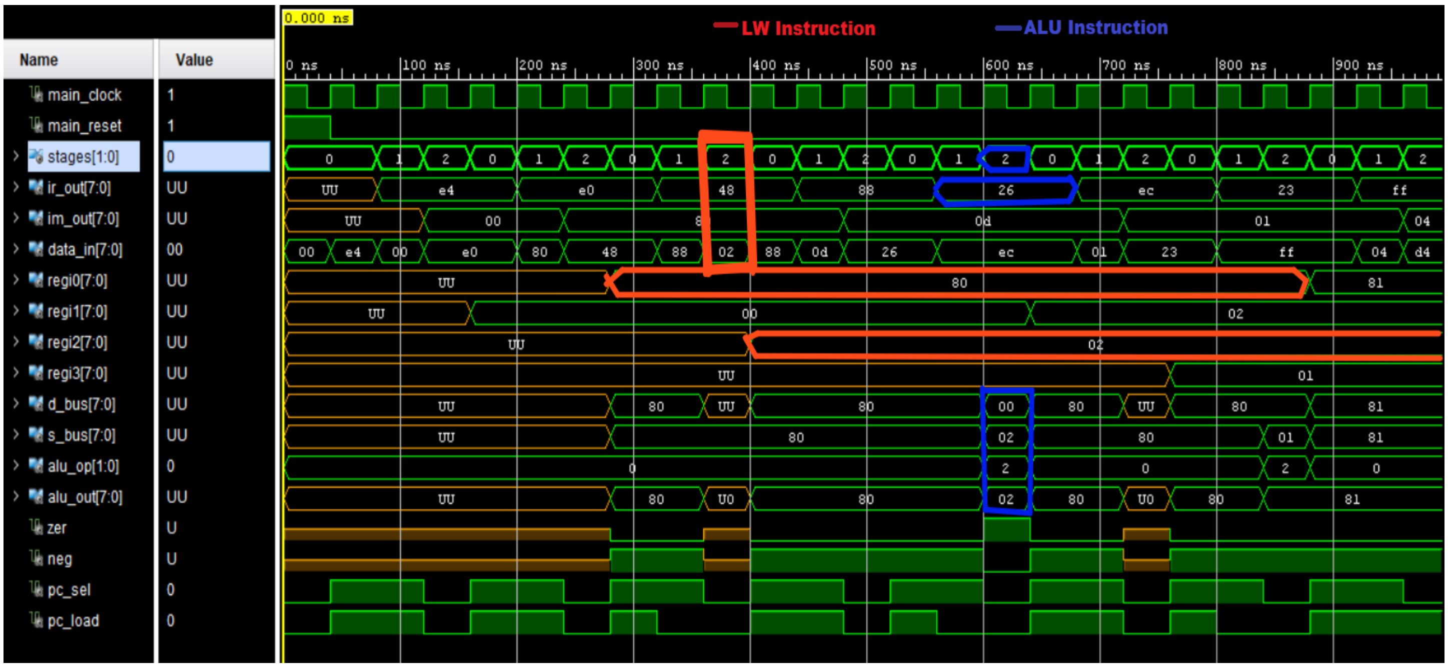
2) IR is loaded with instruction and IM is loaded with a value

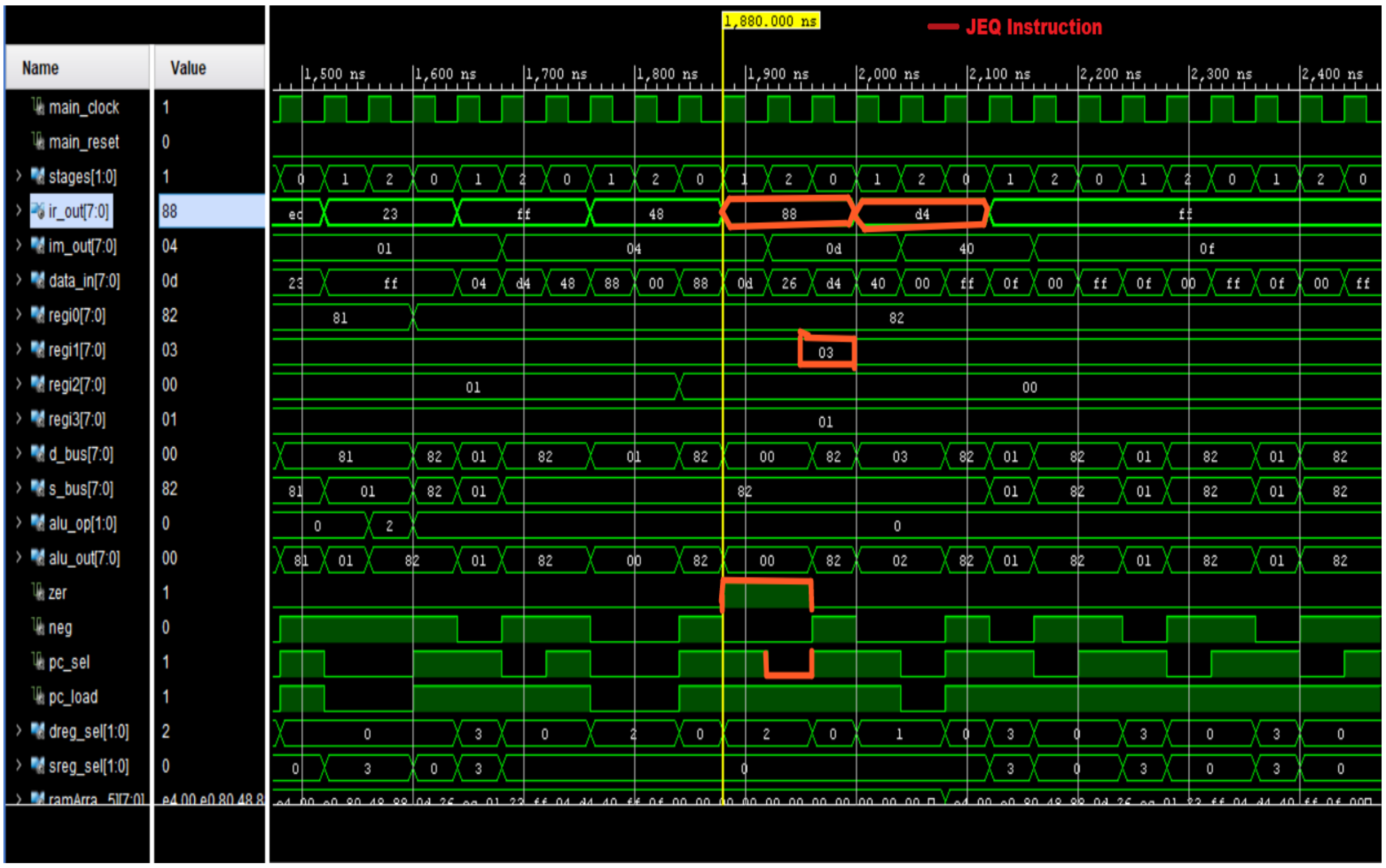
3) sbus is placed on address bus and datain on destination register.

4)ALU instructions are performed

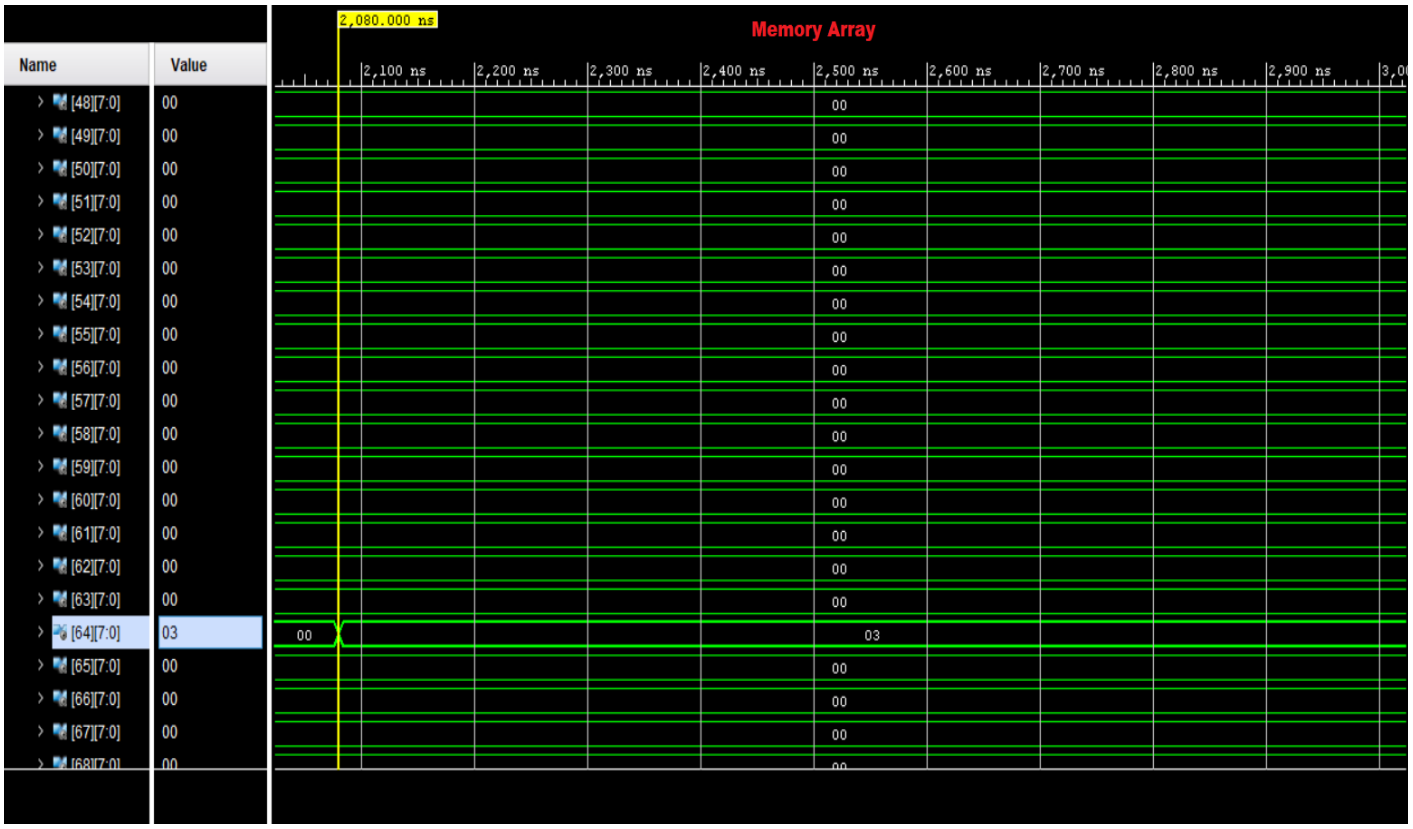
5)N and Z value is showed in decode logic.

Waveform:



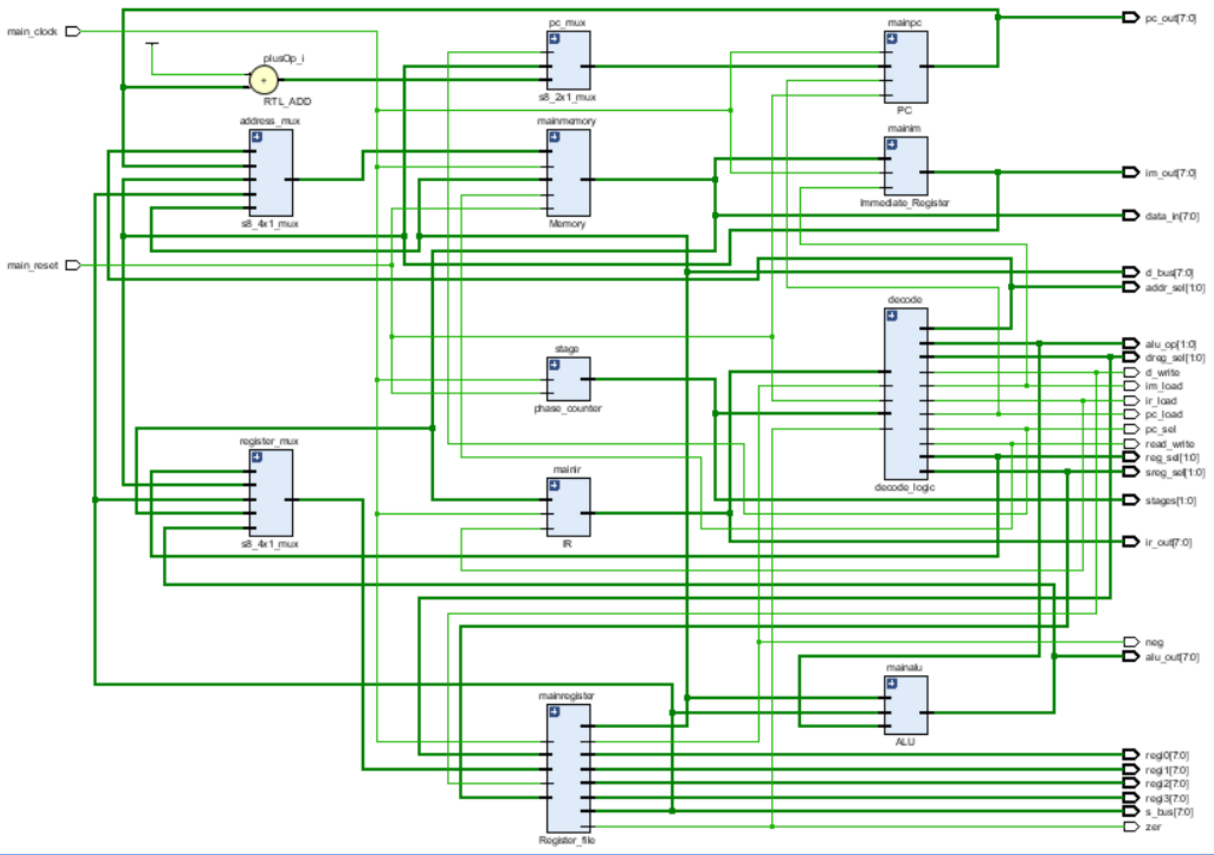


Name	Value
main_clock	1
main_reset	0
stages[1:0]	1
ir_out[7:0]	88
im_out[7:0]	04
data_in[7:0]	0d
regi0[7:0]	82
regi1[7:0]	03
regi2[7:0]	00
regi3[7:0]	01
d_bus[7:0]	00
s_bus[7:0]	82
alu_op[1:0]	0
alu_out[7:0]	00
zer	1
neg	0
pc_sel	1
pc_load	1
dreg_sel[1:0]	2
sreg_sel[1:0]	0
ramArra_5][7:0]	e4 00 e0 80 48 8

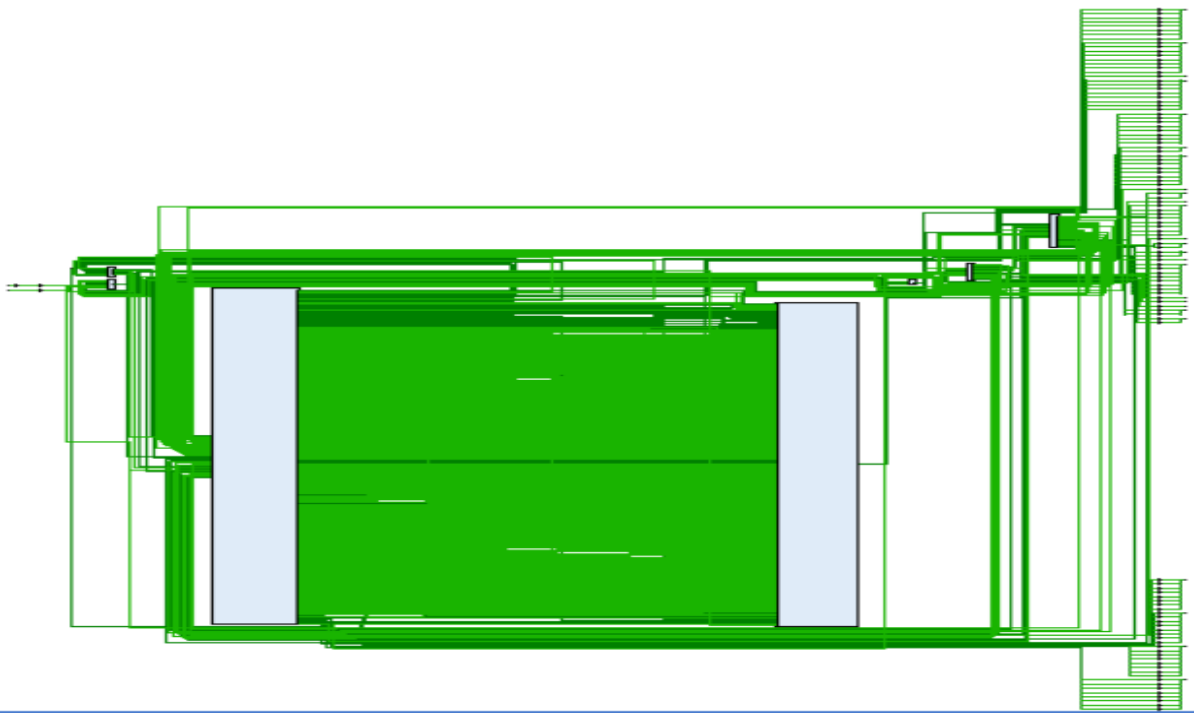


Name	Value
[48][7:0]	00
[49][7:0]	00
[50][7:0]	00
[51][7:0]	00
[52][7:0]	00
[53][7:0]	00
[54][7:0]	00
[55][7:0]	00
[56][7:0]	00
[57][7:0]	00
[58][7:0]	00
[59][7:0]	00
[60][7:0]	00
[61][7:0]	00
[62][7:0]	00
[63][7:0]	00
[64][7:0]	03
[65][7:0]	00
[66][7:0]	00
[67][7:0]	00
[68][7:0]	00

Elaborated Design:



Synthesized Design



Design Issues:

- Initially we faced a problem with the output from stage counter. During reset, the counter was set to zero but the value at the rising edge of clock should be zero, but the output generated showed the count value starting from one as the counter counts from reset zero. Expected output is to start counter from zero even after reset zero.
- Immediate Register and Instruction Register were getting their value after delay of one cycle.
- We had problem in JEQ instruction execution, that is our code didn't stop during JEQ instruction. Since the ir, im register outputs are delayed one cycle on contrary to sbus, dbus, zero, negative which are read asynchronously (i.e. output is produced in same cycle) we had problem in understanding the setting up of zero and negative signal (which is based on register values that were previously stored by the former stage.)