COA PROJECT
PHASE ONE

TEAM :

NAME	ID
NADER MOHAEMD ELFEEL	22P0285
mohamed magdy Mahmoud basal	22P0127
Mohamed Ibrahim Ghoneem	22P0124
Zakaria Alaa Fouad Youssef Eisa	22P0128
Mahmoud Nashaat	22P0126

name	contribution
NADER MOHAEMD ELFEEL	Register file design and implementation from scratch ” some logic gates and muxs” to coding , also the connection between all functions in the Register file to enable writing and displaying data based on inputs.

mohamed magdy Mahmoud basal	**Decoder and mux , also how these elements can contact with whole program and how they can be helpful in each module.**
Mohamed Ibrahim Ghoneem	**The whole Report , and the basic unit of the register file “Flop Register” , also helped in designing the CPU connections with the other modules.**
Zakaria Alaa Fouad Youssef Eisa	**ALU from basic design on paper to the actual implementation on the IDE**
Mahmoud Nashaat	**The controlling unit “CPU” that contains all of the previous modules and their communication**

ABSTRACT

The MIPS processor stands as an example of Reduced Instruction Set Computing (RISC) architecture, renowned for its simplicity, efficiency, and widespread application across diverse computing domains. This report presents a comprehensive design, implementation, and testing overview of a MIPS processor, focusing on its key components: the register file, arithmetic logic unit (ALU), and central processing unit (CPU). Leveraging VHDL (VHSIC Hardware Description Language), the register file module facilitates efficient data storage and retrieval, while the ALU module performs arithmetic and logic operations on data operands. The CPU module orchestrates instruction execution, seamlessly integrating with the register file and ALU to execute MIPS instructions accurately and efficiently. Through simulation results, RTL implementation photos, and VHDL code snippets, this report offers an in-depth exploration of the MIPS processor design, underscoring its significance in contemporary computing landscapes.

I. MIPS Processor Introduction:

The MIPS processor is a key player in computing, known for its simplicity, efficiency, and versatility. Developed by MIPS Computer Systems, it's widely used in various devices like personal computers and electronic gadgets. This report explores the design, implementation, and testing of a MIPS processor, focusing on its core components: the register file, arithmetic logic unit (ALU), and central processing unit (CPU). Using VHDL (VHSIC Hardware Description Language), we'll dive into each component's workings and how they execute MIPS instructions effectively. By understanding the MIPS processor's design, we gain insight into its importance in modern computing.

II. Register File Design:

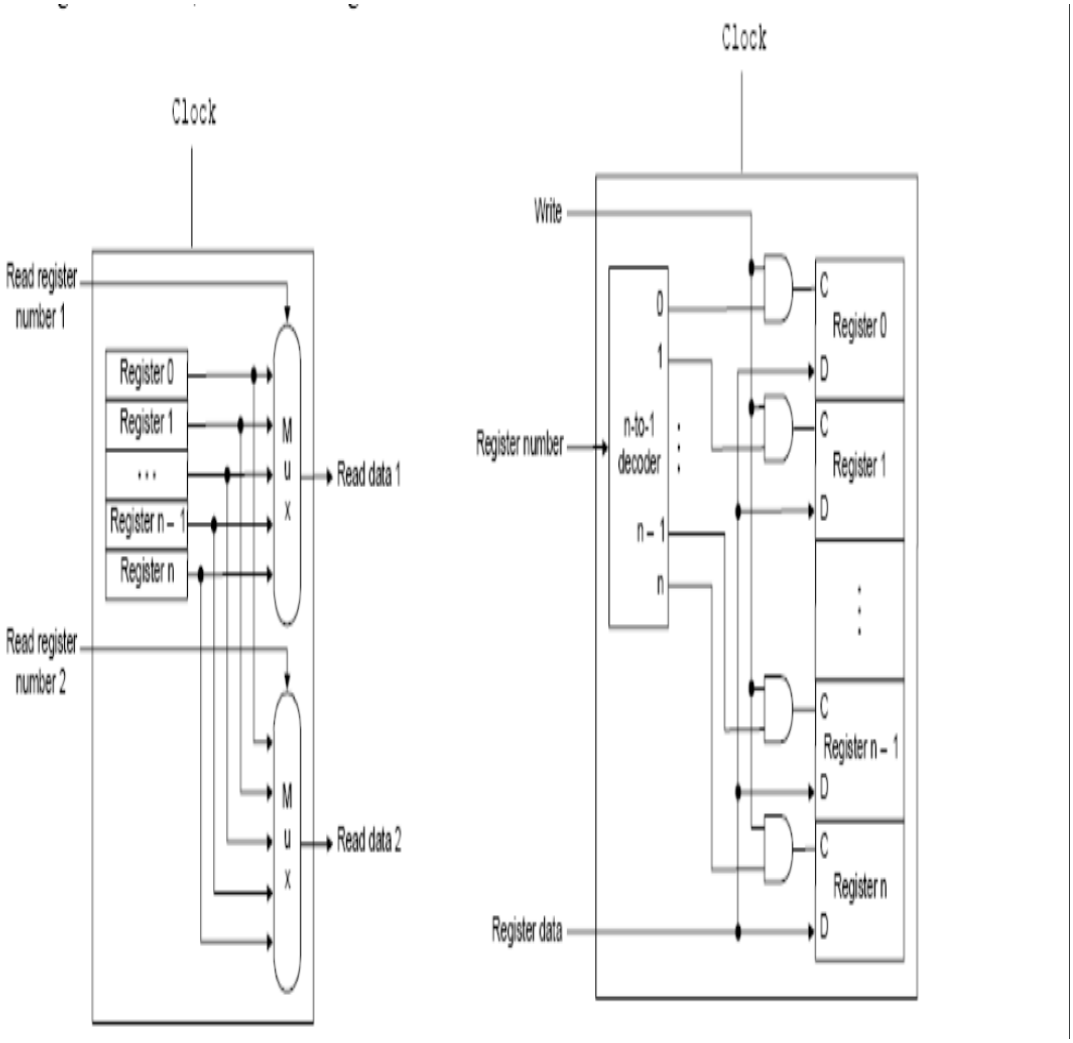
A. Purpose and Functionality:

The register file serves as a high-speed storage unit for data operands in the MIPS processor. It consists of multiple registers, each capable of storing a 32-bit data value. The register file's main purpose is to facilitate efficient access and manipulation of data during instruction execution.

B. VHDL Implementation:

The register file is implemented using VHDL (VHSIC Hardware Description Language), a hardware description language used for digital circuit design. The VHDL code includes modules for read and write operations on individual registers. The design incorporates input ports for selecting read and write addresses, enabling write operations, and providing data input

Register File has been implemented based on this design :



=>The decoder decides which register to get written in , under these conditions (write enable is high and clk is in falling_edge).

=>The MUX is used to get the output from the right register

There are 2 muxs for 2 data selection and 2 data out puts.

Code :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity R_F2 is
    Port ( read_sel1 : in  STD_LOGIC_VECTOR (4 downto 0);
           read_sel2 : in  STD_LOGIC_VECTOR (4 downto 0);
           write_sel : in  STD_LOGIC_VECTOR (4 downto 0);
           write_ena : in  STD_LOGIC;
           clk : in  STD_LOGIC;
           write_data : in  STD_LOGIC_VECTOR (31 downto 0);
           data1 : out  STD_LOGIC_VECTOR (31 downto 0);
           data2 : out  STD_LOGIC_VECTOR (31 downto 0));
end R_F2;

architecture Behavioral of R_F2 is

component decoder is

    Port (
        write_sel1 : in  std_logic_vector(4  downto 0);
        output : out std_logic_vector(31 downto 0)
    );
end component;
```

```
35  component REG22 is
36
37     port(
38        Q : out std_logic_vector(31 downto 0);
39        clk :in std_logic;
40        async_reset: in std_logic;
41        write_ena: in std_logic;
42        D :in  std_logic_vector (31 downto 0)
43     );
44  end component;
45
46
47  component mux1 is
48      Port (
49             in0 : in  STD_LOGIC_VECTOR (31 downto 0);
50             in1 : in  STD_LOGIC_VECTOR (31 downto 0);
51             in2 : in  STD_LOGIC_VECTOR (31 downto 0);
52             in3 : in  STD_LOGIC_VECTOR (31 downto 0);
53             in4 : in  STD_LOGIC_VECTOR (31 downto 0);
54             in5 : in  STD_LOGIC_VECTOR (31 downto 0);
55             in6 : in  STD_LOGIC_VECTOR (31 downto 0);
56             in7 : in  STD_LOGIC_VECTOR (31 downto 0);
57             in8 : in  STD_LOGIC_VECTOR (31 downto 0);
58             in9 : in  STD_LOGIC_VECTOR (31 downto 0);
59             in10 : in  STD_LOGIC_VECTOR (31 downto 0);
60             in11 : in  STD_LOGIC_VECTOR (31 downto 0);
61             in12 : in  STD_LOGIC_VECTOR (31 downto 0);
62             in13 : in  STD_LOGIC_VECTOR (31 downto 0);
63             in14 : in  STD_LOGIC_VECTOR (31 downto 0);
64             in15 : in  STD_LOGIC_VECTOR (31 downto 0);
65             in16 : in  STD_LOGIC_VECTOR (31 downto 0);
66             in17 : in  STD_LOGIC_VECTOR (31 downto 0);
67             in18 : in  STD_LOGIC_VECTOR (31 downto 0);
```

```
            in29 : in  STD_LOGIC_VECTOR (31 downto 0);
            in30 : in  STD_LOGIC_VECTOR (31 downto 0);
            in31 : in  STD_LOGIC_VECTOR (31 downto 0);
             sel : in  STD_LOGIC_VECTOR (4 downto 0);
            outs : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

signal dec_sig : std_logic_vector(31 downto 0);
--signal reg_sig : std_logic_vector(31 downto 0);
signal async_reset1 : std_logic :='0';
Signal reg0out : std_logic_vector (31 downto 0);
Signal reg1out : std_logic_vector (31 downto 0);
Signal reg2out : std_logic_vector (31 downto 0);
Signal reg3out : std_logic_vector (31 downto 0);
Signal reg4out : std_logic_vector (31 downto 0);
Signal reg5out : std_logic_vector (31 downto 0);
Signal reg6out : std_logic_vector (31 downto 0);
Signal reg7out : std_logic_vector (31 downto 0);
Signal reg8out : std_logic_vector (31 downto 0);
Signal reg9out : std_logic_vector (31 downto 0);
Signal reg10out : std_logic_vector (31 downto 0);
Signal reg11out : std_logic_vector (31 downto 0);
Signal reg12out : std_logic_vector (31 downto 0);
Signal reg13out : std_logic_vector (31 downto 0);
Signal reg14out : std_logic_vector (31 downto 0);
Signal reg15out : std_logic_vector (31 downto 0);
Signal reg16out : std_logic_vector (31 downto 0);
Signal reg17out : std_logic_vector (31 downto 0);
Signal reg18out : std_logic_vector (31 downto 0);
Signal reg19out : std_logic_vector (31 downto 0);
Signal reg20out : std_logic_vector (31 downto 0);
Signal reg21out : std_logic_vector (31 downto 0);
Signal reg22out : std_logic_vector (31 downto 0);
Signal reg23out : std_logic_vector (31 downto 0);
```

```
118 Signal reg30out : std_logic_vector (31 downto 0);
119 Signal reg31out : std_logic_vector (31 downto 0);
120
121 signal clk1: std_logic ;
122
123 begin
124
125   --clk1 <='1' when (boolean(dec_sig(0)) and write_ena='1' and (clk' event and clk='0') = '1');
126
127 dec : decoder
128 port map (
129         write_sel1 => write_sel,
130         output     => dec_sig
131         );
132
133 reg0 : REG22 port map (Q => reg0out,clk => clk ,write_ena => dec_sig(0),async_reset => async_reset1, D =>write_data);
134 reg1 : REG22 port map (Q => reg1out,clk => clk ,write_ena => dec_sig(1),async_reset => async_reset1, D =>write_data);
135 reg2 : REG22 port map (Q => reg2out,clk => clk ,write_ena => dec_sig(2),async_reset => async_reset1, D =>write_data);
136 reg3 : REG22 port map (Q => reg3out,clk => clk ,write_ena => dec_sig(3),async_reset => async_reset1, D =>write_data);
137 reg4 : REG22 port map (Q => reg4out,clk => clk ,write_ena => dec_sig(4),async_reset => async_reset1, D =>write_data);
138 reg5 : REG22 port map (Q => reg5out,clk => clk ,write_ena => dec_sig(5),async_reset => async_reset1, D =>write_data);
139 reg6 : REG22 port map (Q => reg6out,clk => clk ,write_ena => dec_sig(6),async_reset => async_reset1, D =>write_data);
140 reg7 : REG22 port map (Q => reg7out,clk => clk ,write_ena => dec_sig(7),async_reset => async_reset1, D =>write_data);
141 reg8 : REG22 port map (Q => reg8out,clk => clk ,write_ena => dec_sig(8),async_reset => async_reset1, D =>write_data);
142 reg9 : REG22 port map (Q => reg9out,clk => clk ,write_ena => dec_sig(9),async_reset => async_reset1, D =>write_data);
143 reg10 : REG22 port map (Q => reg10out,clk => clk ,write_ena => dec_sig(10),async_reset => async_reset1, D =>write_data);
144 reg11 : REG22 port map (Q => reg11out,clk => clk ,write_ena => dec_sig(11),async_reset => async_reset1, D =>write_data);
145 reg12 : REG22 port map (Q => reg12out,clk => clk ,write_ena => dec_sig(12),async_reset => async_reset1, D =>write_data);
146 reg13 : REG22 port map (Q => reg13out,clk => clk ,write_ena => dec_sig(13),async_reset => async_reset1, D =>write_data);
147 reg14 : REG22 port map (Q => reg14out,clk => clk ,write_ena => dec_sig(14),async_reset => async_reset1, D =>write_data);
148 reg15 : REG22 port map (Q => reg15out,clk => clk ,write_ena => dec_sig(15),async_reset => async_reset1, D =>write_data);
149 reg16 : REG22 port map (Q => reg16out,clk => clk ,write_ena => dec_sig(16),async_reset => async_reset1, D =>write_data);
150 reg17 : REG22 port map (Q => reg17out,clk => clk ,write_ena => dec_sig(17),async_reset => async_reset1, D =>write_data);
151 reg18 : REG22 port map (Q => reg18out,clk => clk ,write_ena => dec_sig(18),async_reset => async_reset1, D =>write_data);
```

```
reg29 : REG22 port map (Q => reg29out,clk => clk ,write_ena => dec_sig(29),async_reset => async_reset1, D =>write_dat
reg30 : REG22 port map (Q => reg30out,clk => clk ,write_ena => dec_sig(30),async_reset => async_reset1, D =>write_dat
reg31 : REG22 port map (Q => reg31out,clk => clk ,write_ena => dec_sig(31),async_reset => async_reset1, D =>write_dat



mux11: mux1 port map (
in0 =>reg0out,
in1 =>reg1out,
in2=>reg2out,
in3=>reg3out,
in4=>reg4out,
in5=>reg5out,
in6=>reg6out,
in7=>reg7out,
in8=>reg8out,
in9=>reg9out,
in10=>reg10out,
in11=>reg11out,
in12=>reg12out,
in13=>reg13out,
in14=>reg14out,
in15=>reg15out,
in16=>reg16out,
in17=>reg17out,
in18=>reg18out,
in19=>reg19out,
in20=>reg20out,
in21=>reg21out,
in22=>reg22out,
in23=>reg23out,
in24=>reg24out,
```

```
201   in28=>reg28out,
202   in29=>reg29out,
203   in30=>reg30out,
204   in31=>reg31out,
205   sel=>read_sel1,
206   outs=>data1
207   );
208
209
210   mux2: mux1 port map (
211   in0=>reg0out,
212   in1=>reg1out,
213   in2=>reg2out,
214   in3=>reg3out,
215   in4=>reg4out,
216   in5=>reg5out,
217   in6=>reg6out,
218   in7=>reg7out,
219   in8=>reg8out,
220   in9=>reg9out,
221   in10=>reg10out,
222   in11=>reg11out,
223   in12=>reg12out,
224   in13=>reg13out,
225   in14=>reg14out,
226   in15=>reg15out,
227   in16=>reg16out,
228   in17=>reg17out,
229   in18=>reg18out,
230   in19=>reg19out,
231   in20=>reg20out,
232   in21=>reg21out,
233   in22=>reg22out,
234   in23=>reg23out,
```

C. Register File Operations:

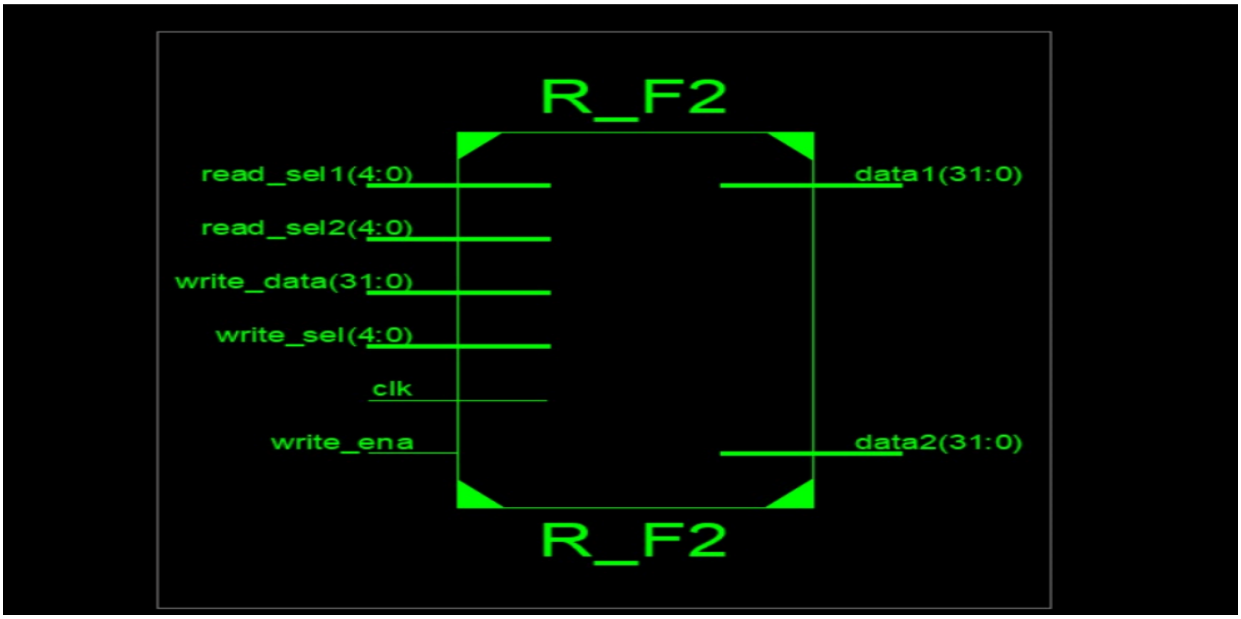
The register file supports the following operations:

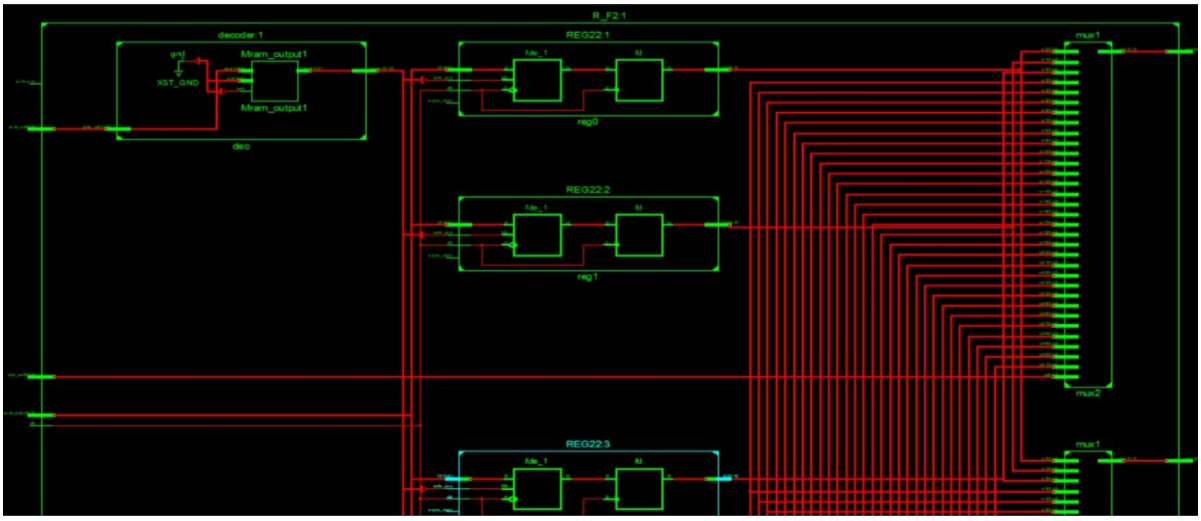
- Read: Retrieves the contents of a specified register.
- Write: Stores data into a specified register.

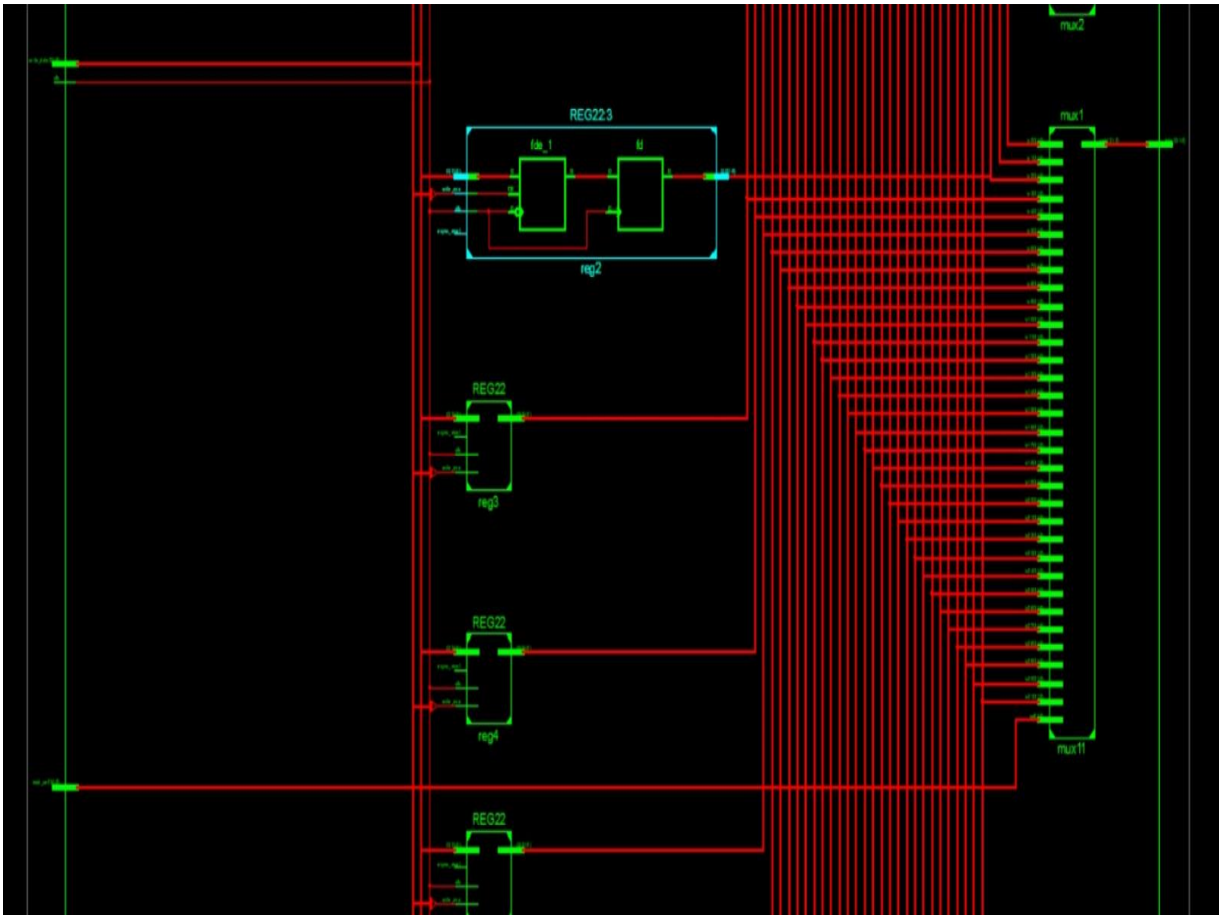
D. Simulation Results:

Simulation results validate the functionality of the register file module, demonstrating successful read and write operations. Test cases are executed to verify the correctness of data retrieval and storage within the register file.

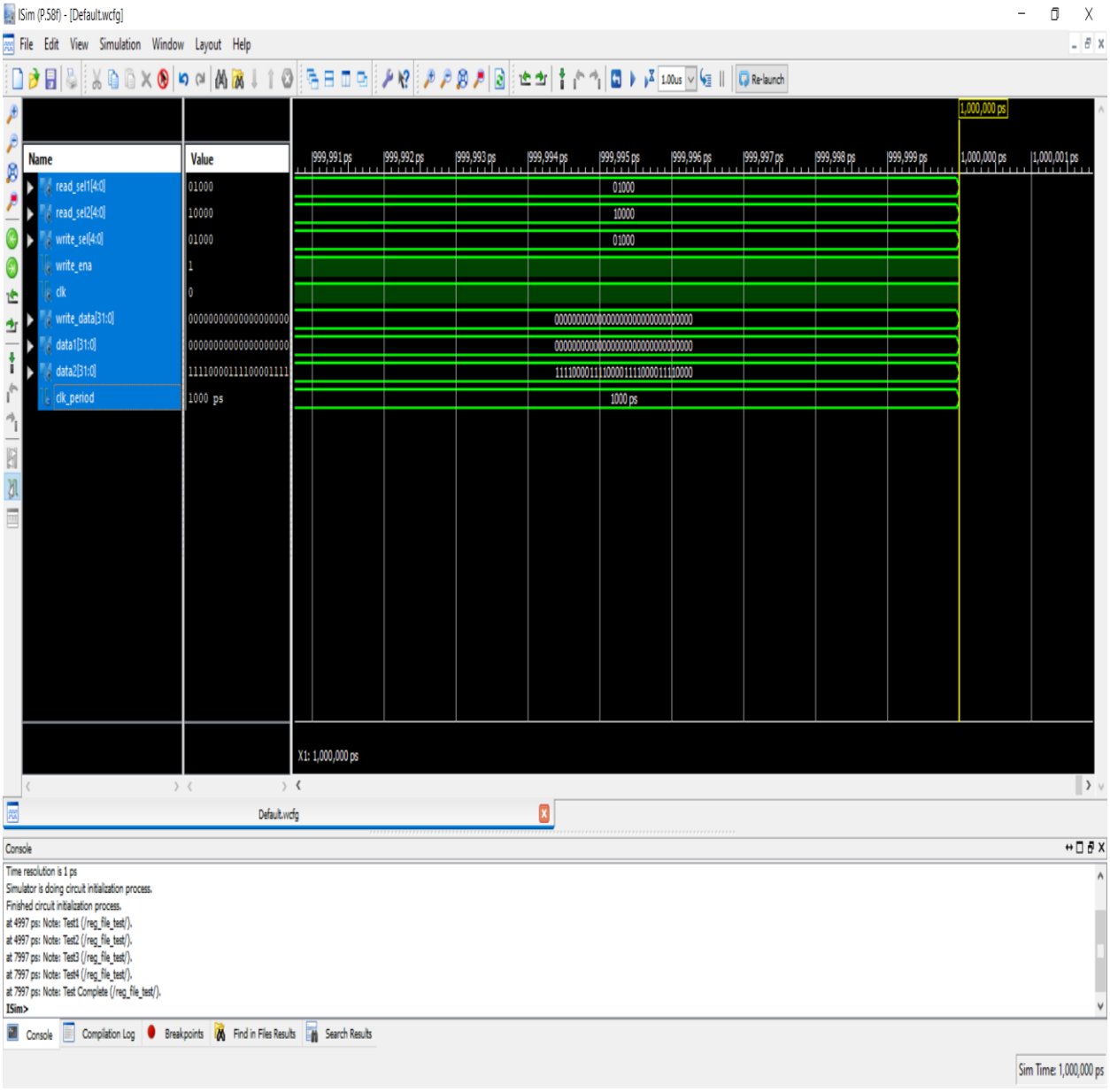
RTL Implementation Photo:




R_F2:1
decoder:1
gnd
XST_GND
Mram_output1
Mram_output1
dec
REG22:1
fde_1
fd
reg0
REG22:2
reg1
mux1
mux2
REG22:3


mux2
REG223
mux1
fde_1
fd
reg2
REG22
reg3
REG22
reg4
mux11
REG22

Test cases :


ISim (P.58f) - [Default.wcfg]
File Edit View Simulation Window Layout Help
Re-launch
Name
Value
read_sel1[4:0]
read_sel2[4:0]
write_sel[4:0]
write_ena
clk
write_data[31:0]
data1[31:0]
data2[31:0]
clk_period
01000
10000
01000
1
0
00000000000000000000000000000000
00000000000000000000000000000000
11110000111100001111000011110000
1000 ps
1,000,000 ps
X1: 1,000,000 ps
Default.wcfg
Console
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
at 4997 ps: Note: Test1 (/reg_file_test/).
at 4997 ps: Note: Test2 (/reg_file_test/).
at 7997 ps: Note: Test3 (/reg_file_test/).
at 7997 ps: Note: Test4 (/reg_file_test/).
at 7997 ps: Note: Test Complete (/reg_file_test/).
ISim>
Console
Compilation Log
Breakpoints
Find in Files Results
Search Results
Sim Time: 1,000,000 ps

III. Arithmetic Logic Unit (ALU) Design:

A. Purpose and Functionality:

The ALU is a critical component of the MIPS processor responsible for performing arithmetic and logic operations on data operands. It supports various operations, including AND, OR, addition, subtraction, set less than (SLT), and NOR.

B. VHDL Implementation:

The ALU is implemented in VHDL, incorporating logic to perform arithmetic and logic operations based on the ALU operation code (aluop). The VHDL code includes modules for each supported operation, enabling flexible functionality within the AL

=>data inputs of ALU are the outputs of the Register File

Code :

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  -- Uncomment the following library declaration if using
4  -- arithmetic functions with Signed or Unsigned values
5  use IEEE.NUMERIC_STD.ALL;
6  --use IEEE.UNSIGNED.ALL;
7
8
9  -- Uncomment the following library declaration if instantiating
10 -- any Xilinx primitives in this code.
11 --library UNISIM;
12 --use UNISIM.VComponents.all;
13
14 entity ALU is
15     Port ( data1 : in  STD_LOGIC_VECTOR (31 downto 0);
16            data2 : in  STD_LOGIC_VECTOR (31 downto 0);
17            aluop : in  STD_LOGIC_VECTOR (3 downto 0);
18            dataout : out  STD_LOGIC_VECTOR (31 downto 0);
19            zflag : out  STD_LOGIC);
20 end ALU;
21
22 architecture Behavioral of ALU is
23 signal result : STD_LOGIC_VECTOR(31 downto 0);
24 begin
25   process(data1,data2,aluop)
26    begin
27    case aluop is
28      when "0000" =>  -- and
29        result <= data1 and data2;
30
31      when "0001" =>  -- or
32        result <= data1 or data2;
33
34      when "0010" =>  -- add
```

```
  when "0001" =>  -- or
    result <= data1 or data2;

  when "0010" =>  -- add

    result <= std_logic_vector( unsigned(data1)  +   unsigned(data2) );

  when "0110" =>  -- sub
    result <= std_logic_vector( unsigned(data1)  -  unsigned(data2)  );

  when "0111" =>  -- SLT

   if(data1 < data2)then

    result <= X"00000001" ;
     else

    result <= X"00000000";

    end if;


  when "1100" =>  -- Nor
    result <= data1 nor data2;
  when others =>  null;-- others
        result <= x"00000000";

end case;
end process;

dataout <= result;
zflag <='1' when result <= X"00000000"else
       '0';
```

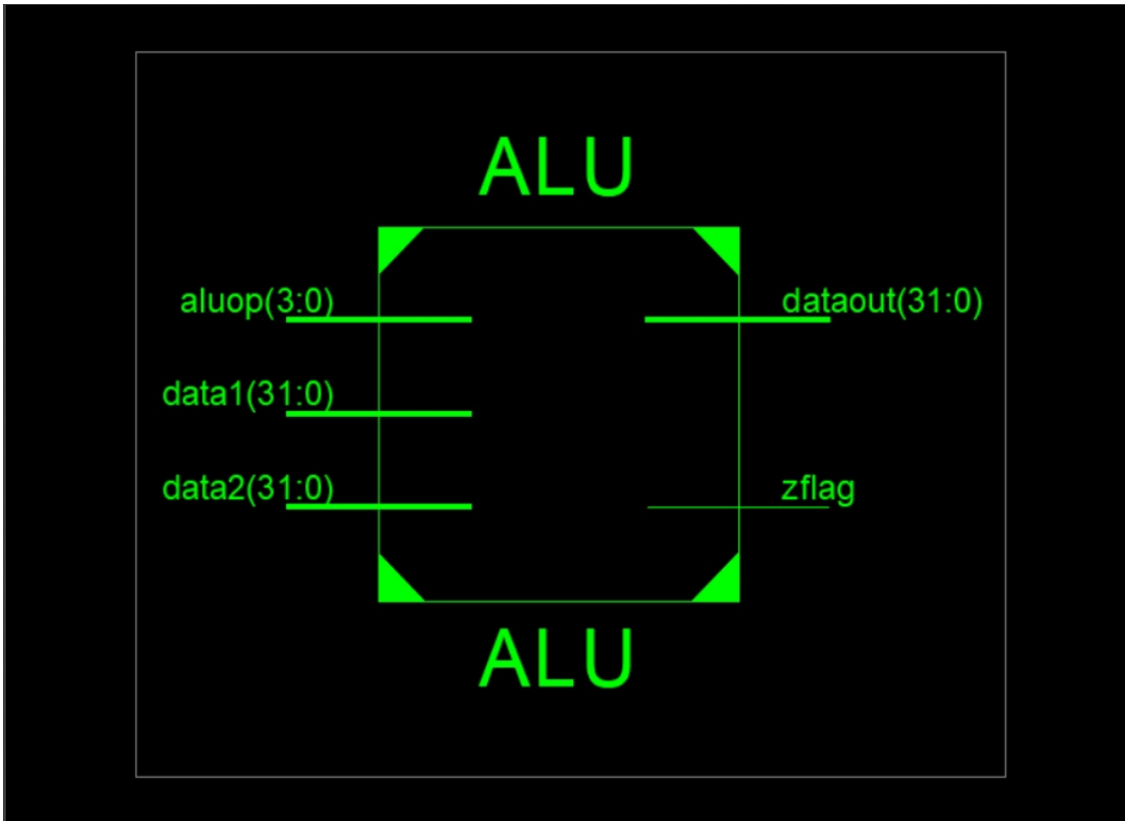
C. ALU Operations:

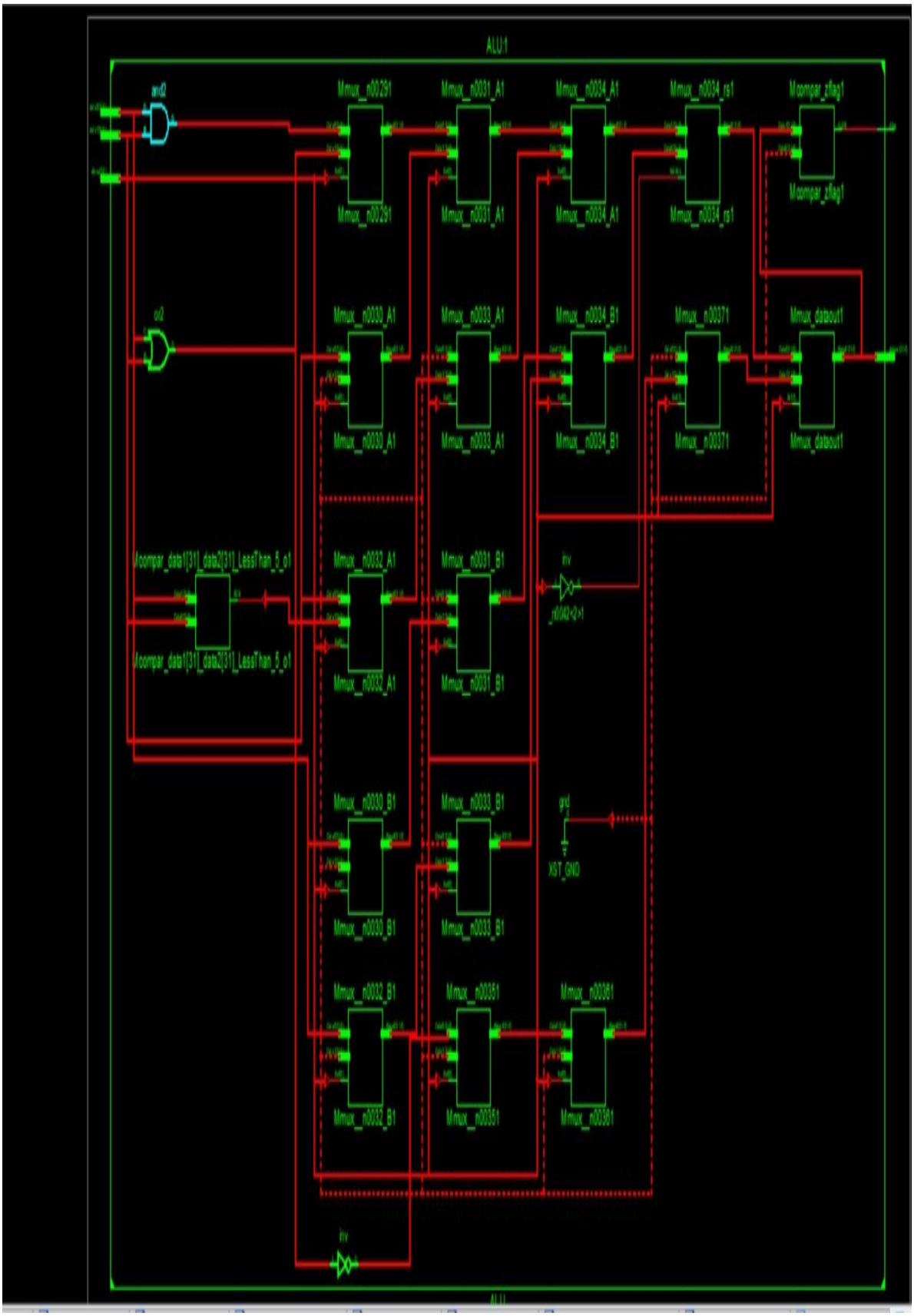
The ALU supports the following operations based on the aluop input:

- AND
- OR
- Addition
- Subtraction
- Set Less Than (SLT)
- NOR
- **D. Simulation Results:**

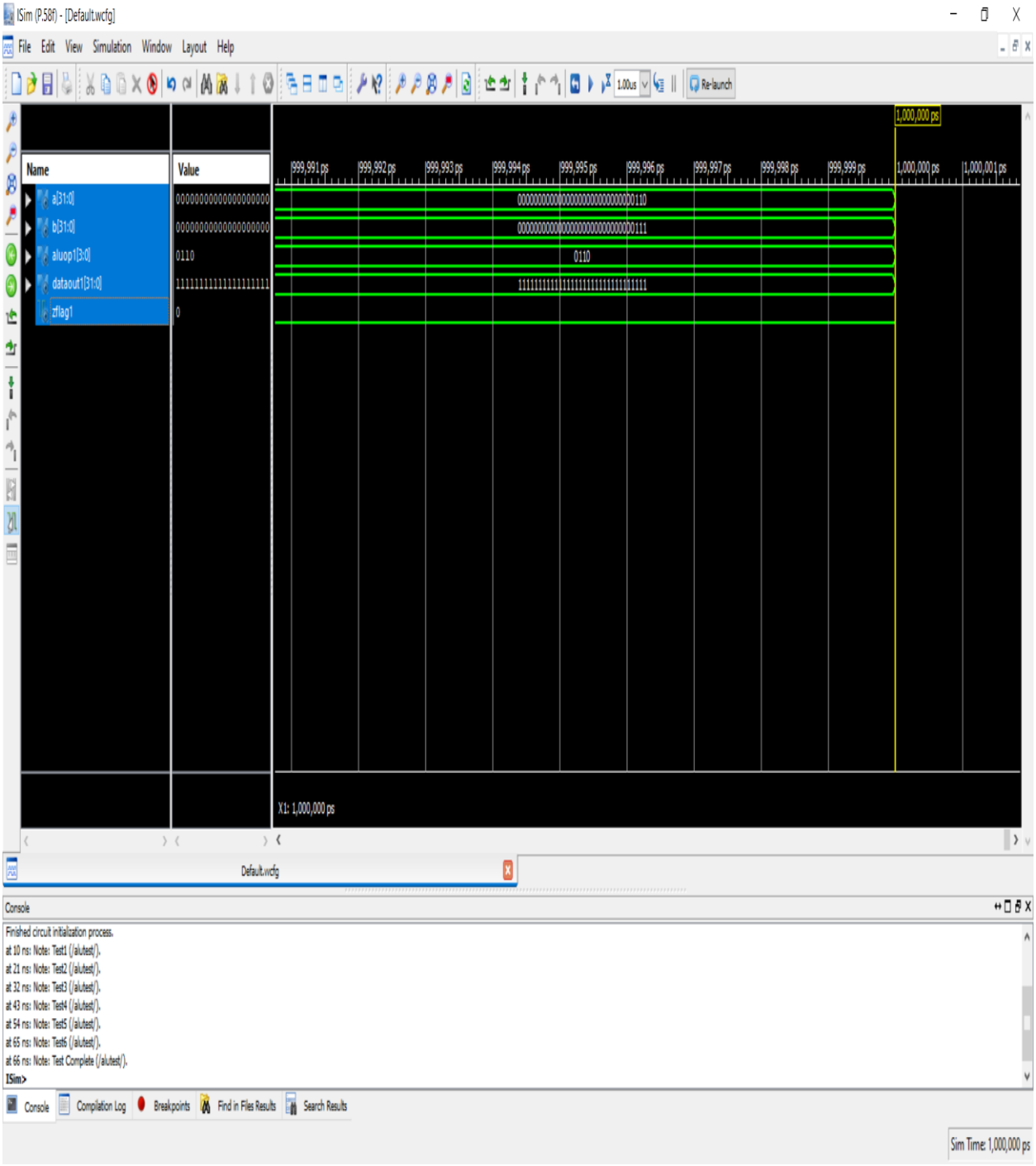
Simulation results demonstrate the correct functionality of the ALU module, producing expected outputs for various input combinations and operation codes. Test cases validate the accuracy of arithmetic and logic operations performed by the ALU.

Rtl implementation photo:




ALU1
Mmux_n0029_1
Mmux_n0031_A1
Mmux_n0034_A1
Mmux_n0034_rs1
Maompar_zflag1
Mmux_n0030_A1
Mmux_n0033_A1
Mmux_n0034_B1
Mmux_n0037_1
Mmux_dataout1
Mmux_n0032_A1
Mmux_n0031_B1
Mmux_n0030_B1
Mmux_n0033_B1
Mmux_n0032_B1
Mmux_n0036_1
Mmux_n0038_1
XST_GND
ALU

Test cases:



IV. Central Processing Unit (CPU) Design:

A. Purpose and Functionality:

The CPU is the core component of the MIPS processor, responsible for executing instructions fetched from memory. It performs instruction fetch, decode, execution, and control operations to execute program instructions effectively.

B. VHDL Implementation:

The CPU is implemented in VHDL, integrating modules for instruction fetch, decode, and execution. It interacts with the register file and ALU to execute instructions, manage data flow, and control signal management effectively.

=> this design enables the previous blocks to communicate with each other .

=> as the output of the ALU after each operation will be the output of the hole block and the input of the writing data in the register file to be stored.

Code :

```
48  component ALU is
49      Port ( data1 : in  STD_LOGIC_VECTOR (31 downto 0);
50             data2 : in  STD_LOGIC_VECTOR (31 downto 0);
51             aluop : in  STD_LOGIC_VECTOR (3 downto 0);
52             dataout : out  STD_LOGIC_VECTOR (31 downto 0);
53             zflag : out  STD_LOGIC);
54  end component;
55
56  signal alu_out : STD_LOGIC_VECTOR (31 downto 0);
57  signal alu_in1 : STD_LOGIC_VECTOR (31 downto 0);
58  signal alu_in2 : STD_LOGIC_VECTOR (31 downto 0);
59  signal zflag2 : STD_LOGIC;
60  signal reset : STD_LOGIC :=rst;
61
62  begin
63
64  reg_file : R_F2
65  port map (
66             read_sel1 => inst(25 downto 21),
67             read_sel2 => inst(20 downto 16),
68             write_sel => inst(15 downto 11),
69             write_ena => regwrite,
70             clk  => clk1,
71             write_data => alu_out, -- alu output
72             --write_data => alu_out, -- alu output
73             data1 => alu_in1, --input of alu
74             data2 => alu_in2  --input of alu
75
76  );
77
78   alu_comp : ALU
79  port map (
80             data1    => alu_in1,
81             data2    => alu_in2,
```

C. CPU Operation:

The CPU performs the following operations:

1. Instruction Fetch
2. Instruction Decode
3. Execution
4. Control

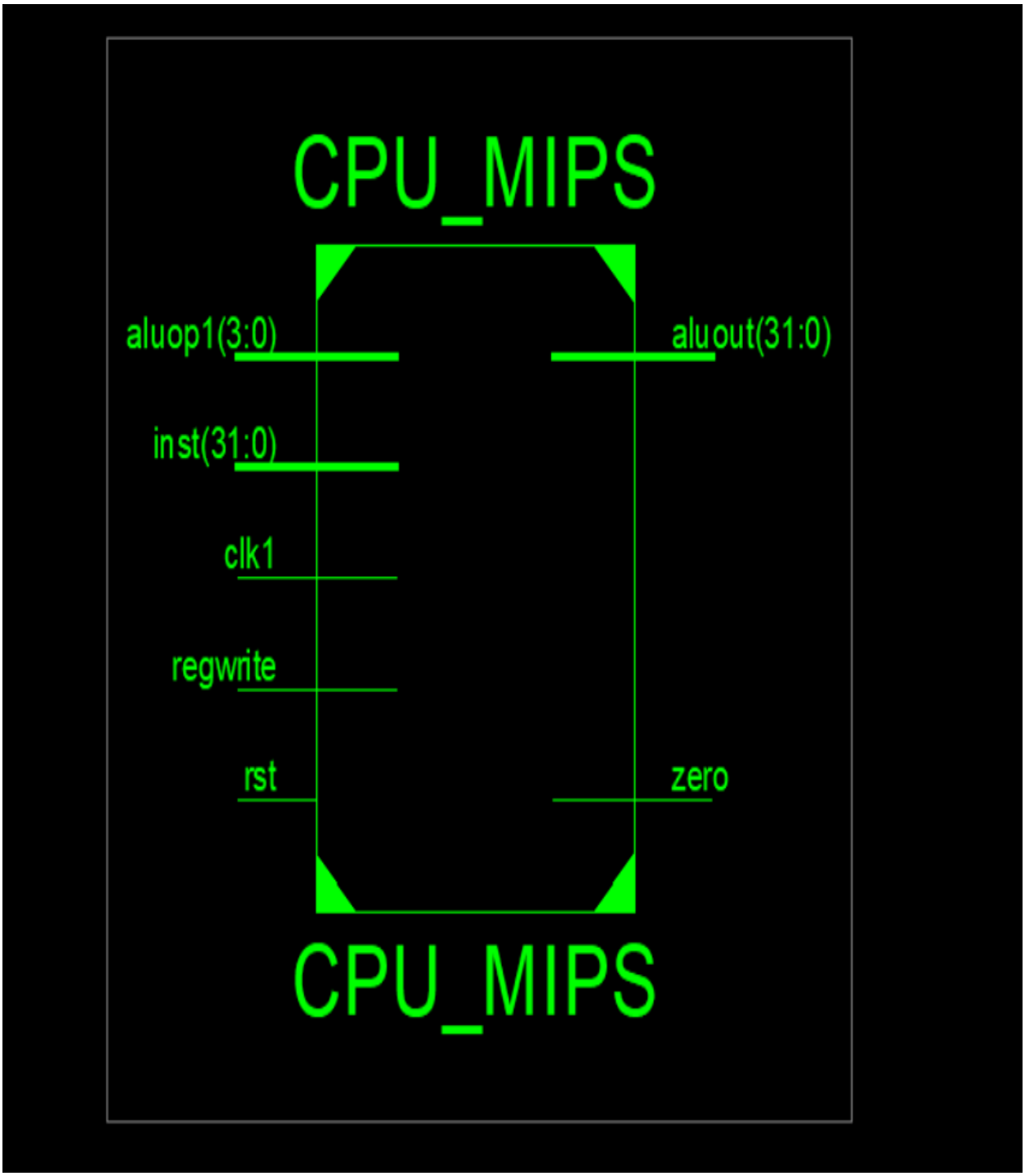
D. Integration with Register File and ALU:

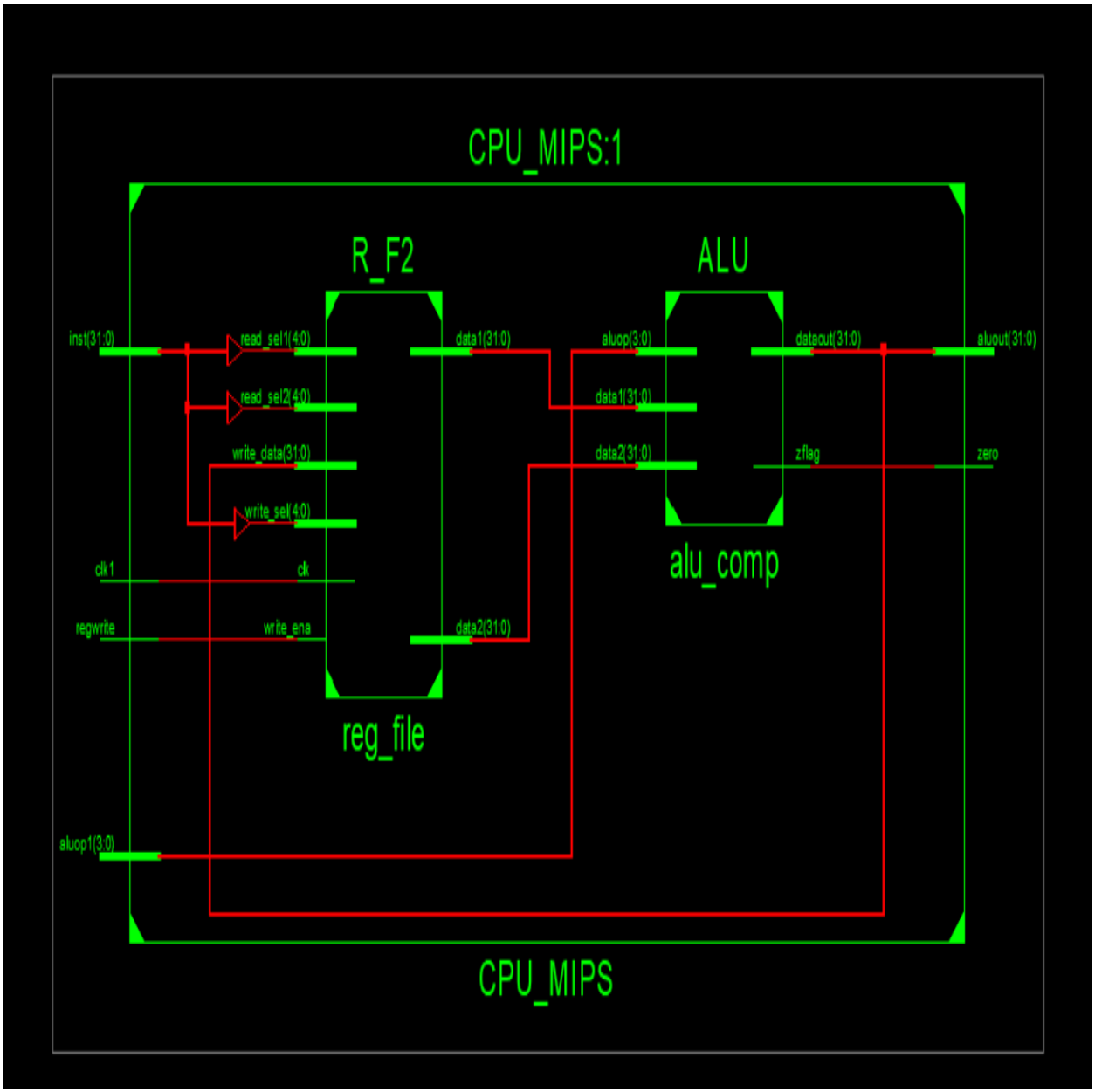
The CPU seamlessly integrates with the register file and ALU to execute instructions efficiently. It manages data flow and control signals, ensuring proper interaction between components during instruction execution.

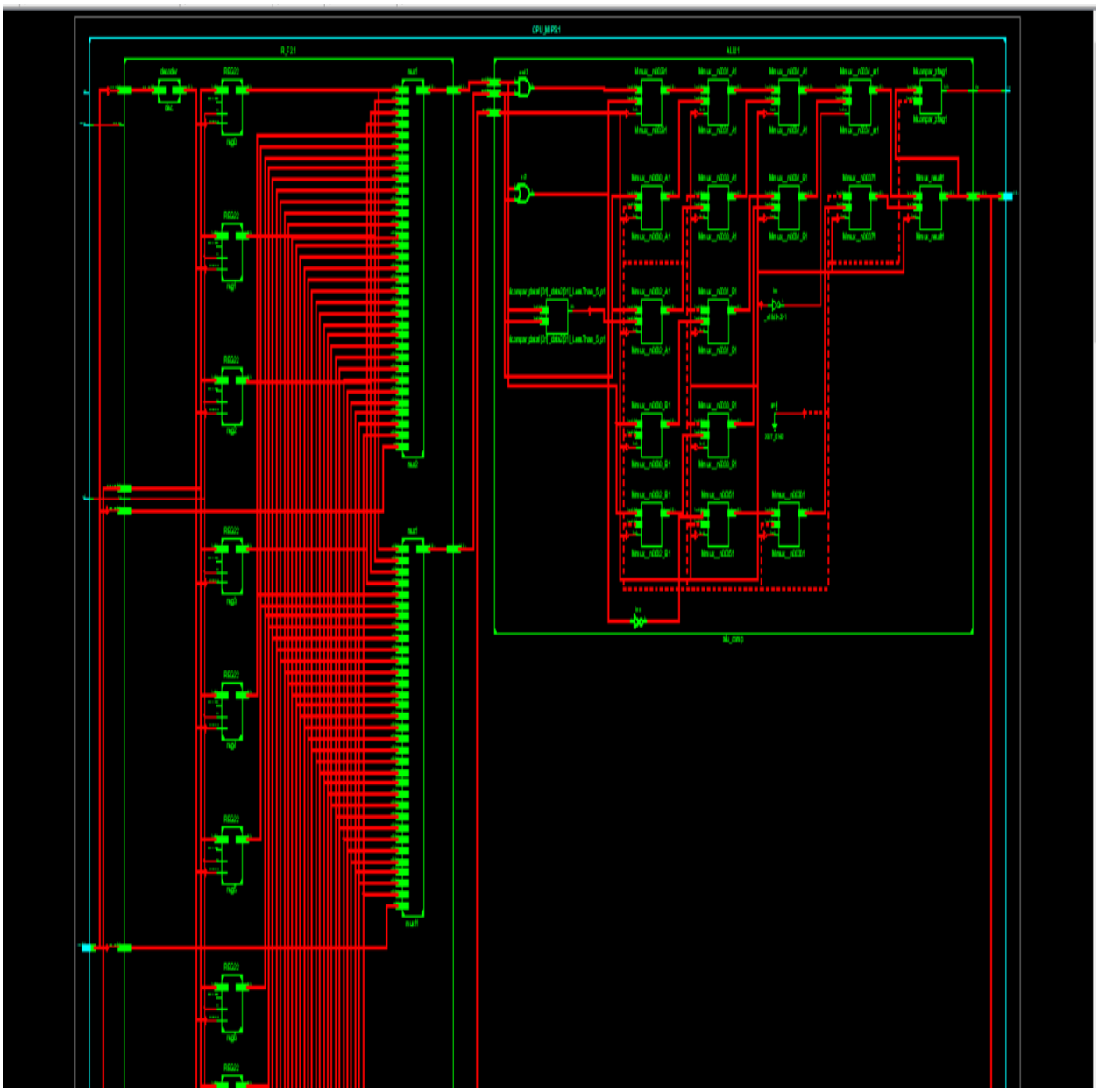
E. Simulation Results:

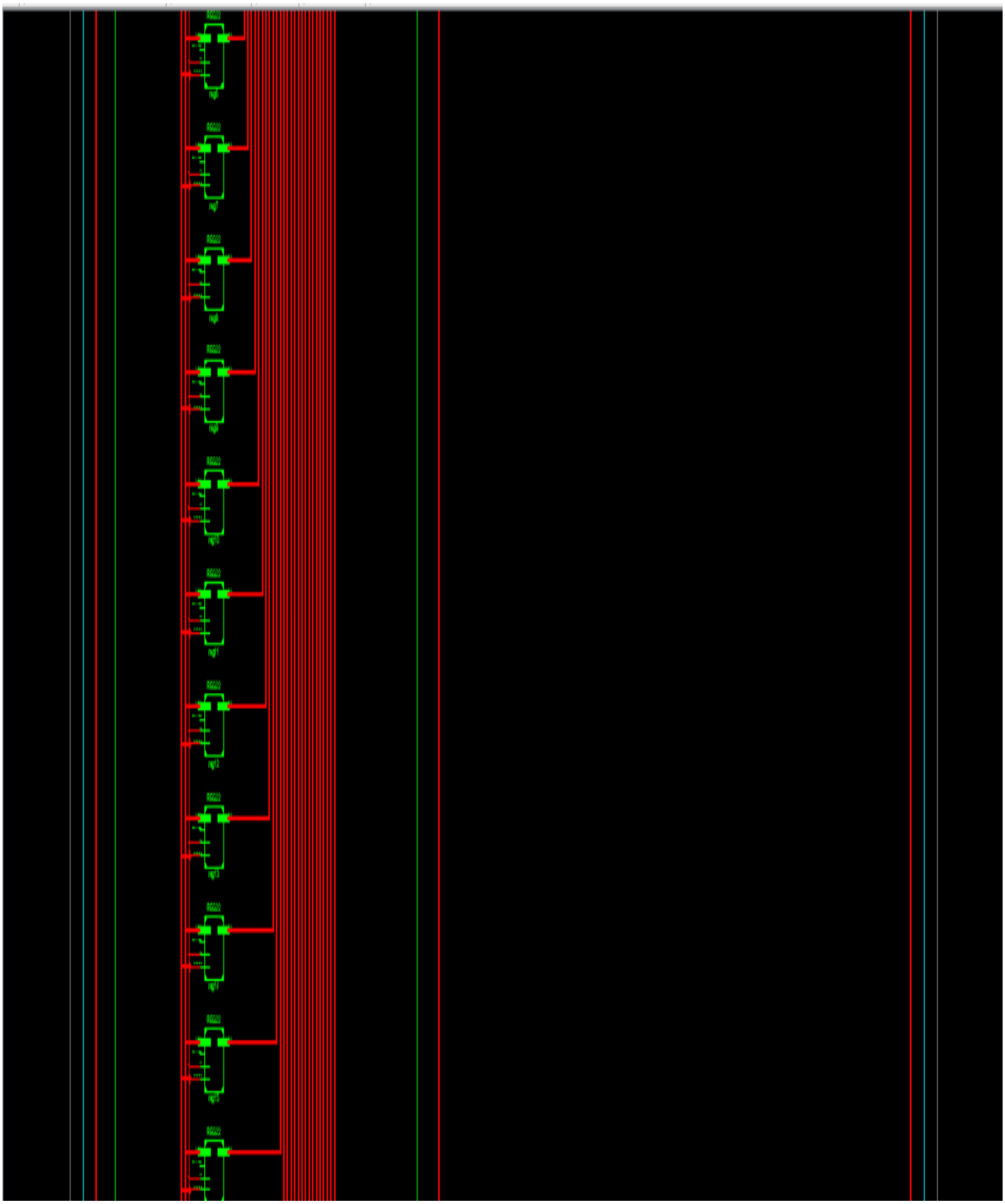
Simulation results validate the functionality of the CPU module, including instruction execution, data manipulation, and control signal management. Test cases verify the correctness of instruction execution and interaction between components within the CPU.

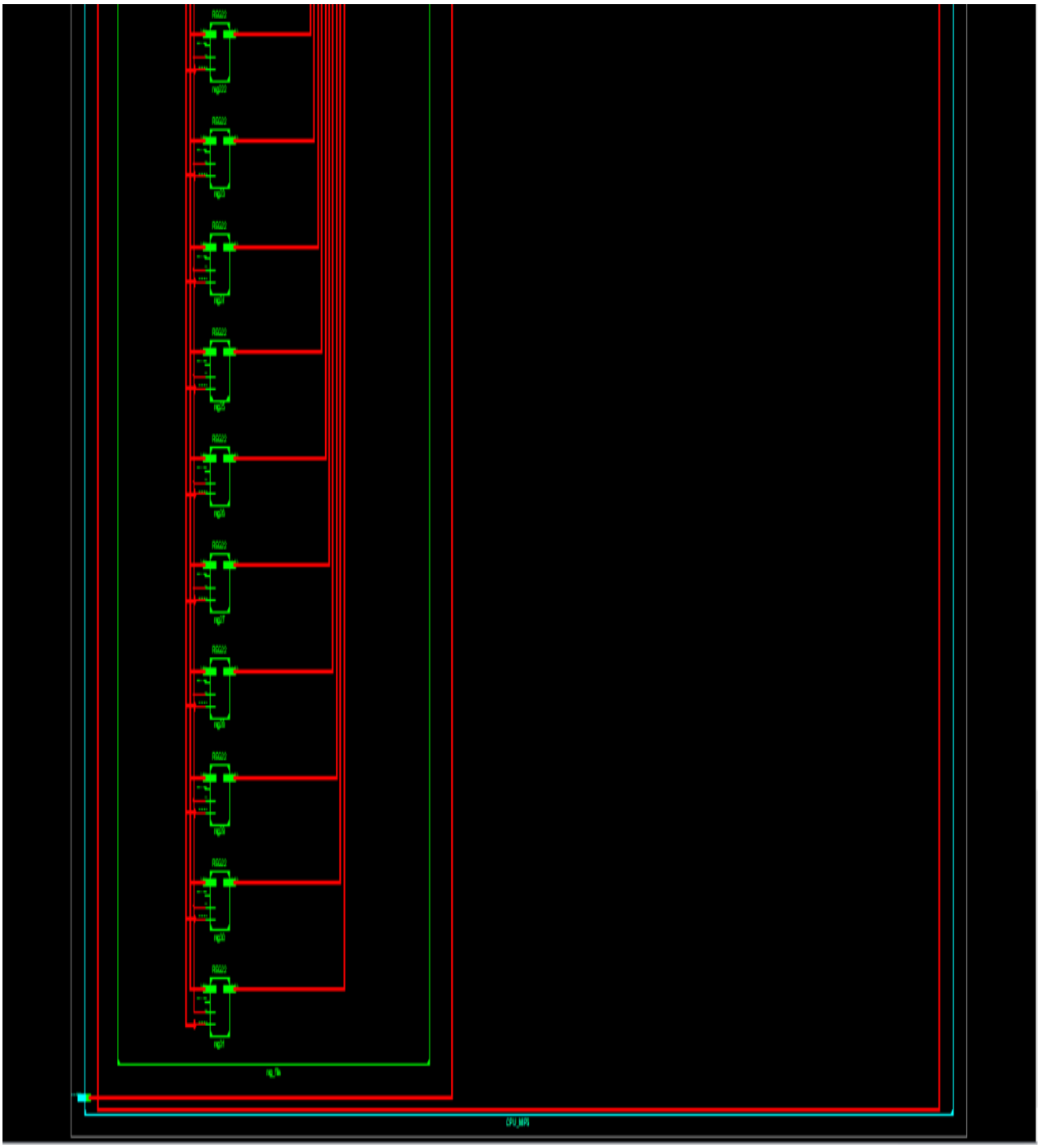
RTL Implementation Photo:


CPU_MIPS
aluop1(3:0)
aluout(31:0)
inst(31:0)
clk1
regwrite
rst
zero
CPU_MIPS


CPU_MIPS:1
R_F2
ALU
inst(31:0)
read_sel1(4:0)
read_sel2(4:0)
write_data(31:0)
write_sel(4:0)
clk1
clk
regwrite
write_ena
data1(31:0)
data2(31:0)
reg_file
aluop(3:0)
data1(31:0)
data2(31:0)
dataout(31:0)
zflag
aluout(31:0)
zero
alu_comp
aluop1(3:0)
CPU_MIPS





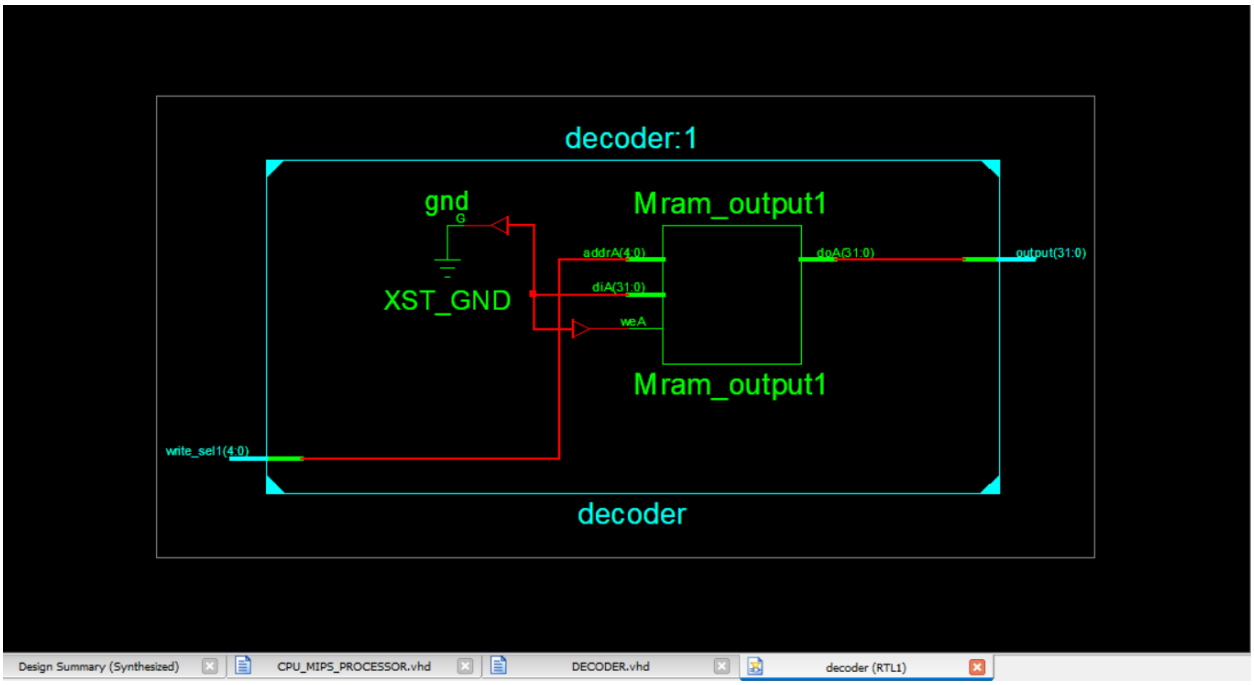


Some helpful components used in the project :

Decoder :

```
1
2   -- Revision:
3   -- Revision 0.01 - File Created
4   -- Additional Comments:
5   --
6   ----------------------------------------------------------------------------------
7   library IEEE;
8   use IEEE.STD_LOGIC_1164.ALL;
9
10  -- Uncomment the following library declaration if using
11  -- arithmetic functions with Signed or Unsigned values
12  --use IEEE.NUMERIC_STD.ALL;
13
14  -- Uncomment the following library declaration if instantiating
15  -- any Xilinx primitives in this code.
16  --library UNISIM;
17  --use UNISIM.VComponents.all;
18
19  entity decoder is
20
21      Port (
22          write_sell : in  std_logic_vector(4  downto 0);
23          output : out std_logic_vector(31  downto 0)
24      );
25  end decoder;
26
27  architecture Behavioral of decoder is
28  begin
29      process(write_sell)
30      begin
31          case write_sell is
32              when "00000" =>
33                  output <= "00000000000000000000000000000001";
34              when "00001" =>
```

```
67                  output <= "00000000000000010000000000000000";
68              when "10010" =>
69                  output <= "00000000000001000000000000000000";
70              when "10011" =>
71                  output <= "00000000000010000000000000000000";
72              when "10100" =>
73                  output <= "00000000000100000000000000000000";
74              when "10101" =>
75                  output <= "00000000001000000000000000000000";
76              when "10110" =>
77                  output <= "00000000010000000000000000000000";
78              when "10111" =>
79                  output <= "00000000100000000000000000000000";
80              when "11000" =>
81                  output <= "00000001000000000000000000000000";
82              when "11001" =>
83                  output <= "00000010000000000000000000000000";
84              when "11010" =>
85                  output <= "00000100000000000000000000000000";
86              when "11011" =>
87                  output <= "00001000000000000000000000000000";
88              when "11100" =>
89                  output <= "00010000000000000000000000000000";
90              when "11101" =>
91                  output <= "00100000000000000000000000000000";
92              when "11110" =>
93                  output <= "01000000000000000000000000000000";
94                  when "11111" =>
95                  output <= "10000000000000000000000000000000";
96                  when others =>
97                  output <= (others => '0');
98          end case;
99      end process;
100 end Behavioral;
```

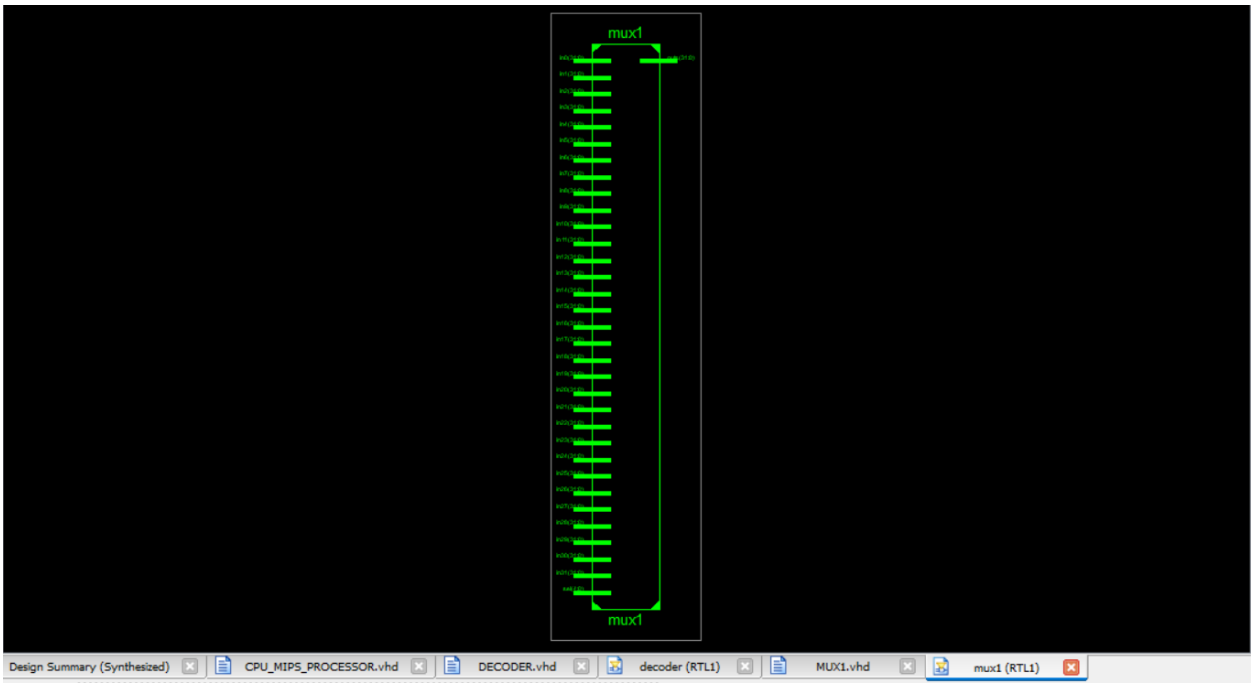

decoder:1
gnd
G
XST_GND
Mram_output1
addrA(4:0)
diA(31:0)
weA
doA(31:0)
output(31:0)
Mram_output1
write_sel1(4:0)
decoder
Design Summary (Synthesized)
CPU_MIPS_PROCESSOR.vhd
DECODER.vhd
decoder (RTL1)

2 / MUX

:

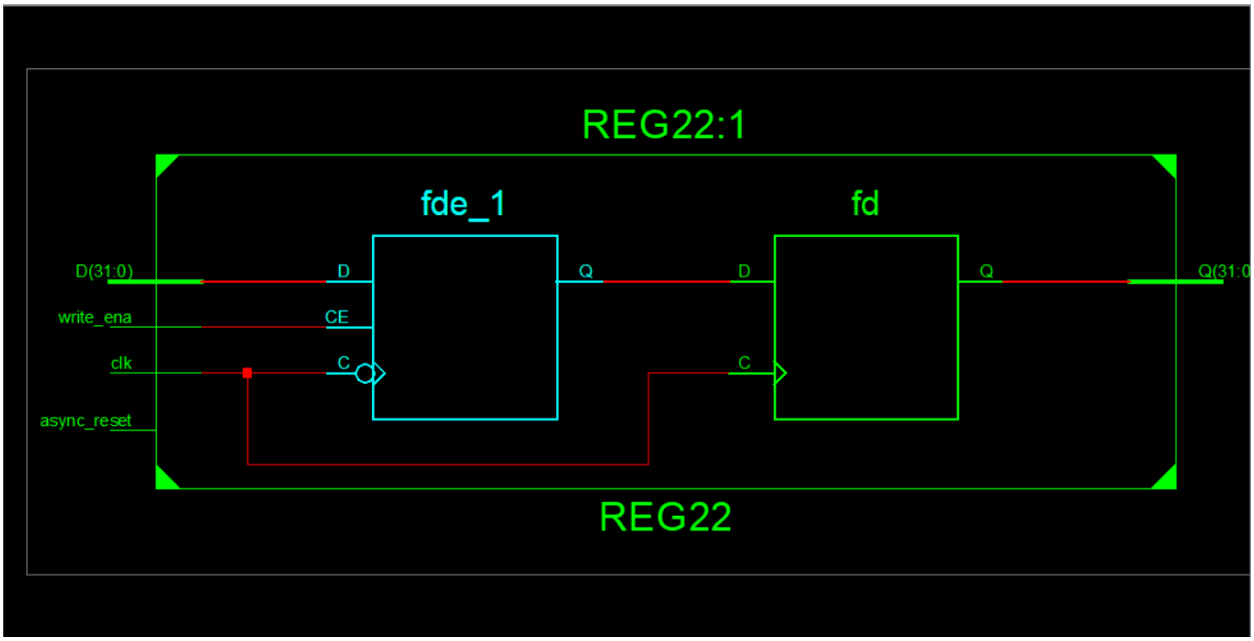
```
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22
23  -- Uncomment the following library declaration if using
24  -- arithmetic functions with Signed or Unsigned values
25  --use IEEE.NUMERIC_STD.ALL;
26
27  -- Uncomment the following library declaration if instantiating
28  -- any Xilinx primitives in this code.
29  --library UNISIM;
30  --use UNISIM.VComponents.all;
31
32  entity mux1 is
33  --generic (n:integer:=32);
34      Port (
35
36            in0 : in  STD_LOGIC_VECTOR (31 downto 0);
37            in1 : in  STD_LOGIC_VECTOR (31 downto 0);
38            in2 : in  STD_LOGIC_VECTOR (31 downto 0);
39            in3 : in  STD_LOGIC_VECTOR (31 downto 0);
40            in4 : in  STD_LOGIC_VECTOR (31 downto 0);
41            in5 : in  STD_LOGIC_VECTOR (31 downto 0);
42            in6 : in  STD_LOGIC_VECTOR (31 downto 0);
43            in7 : in  STD_LOGIC_VECTOR (31 downto 0);
44            in8 : in  STD_LOGIC_VECTOR (31 downto 0);
45            in9 : in  STD_LOGIC_VECTOR (31 downto 0);
46            in10 : in  STD_LOGIC_VECTOR (31 downto 0);
47            in11 : in  STD_LOGIC_VECTOR (31 downto 0);
48            in12 : in  STD_LOGIC_VECTOR (31 downto 0);
49            in13 : in  STD_LOGIC_VECTOR (31 downto 0);
50            in14 : in  STD_LOGIC_VECTOR (31 downto 0);
51            in15 : in  STD_LOGIC_VECTOR (31 downto 0);
52            in16 : in  STD_LOGIC_VECTOR (31 downto 0);
53            in17 : in  STD_LOGIC_VECTOR (31 downto 0);
```

```
71  end mux1;
72
73  architecture Behavioral of mux1 is
74
75  begin
76  outs <=
77       in0 when (sel="00000") else
78       in1 when (sel="00001") else
79       in2 when (sel="00010") else
80       in3 when (sel="00011") else
81       in4 when (sel="00100") else
82       in5 when (sel="00101") else
83       in6 when (sel="00110") else
84       in7 when (sel="00111") else
85       in8 when (sel="01000") else
86       in9 when (sel="01001") else
87       in10 when (sel="01010") else
88       in11 when (sel="01011") else
89       in12 when (sel="01100") else
90       in13 when (sel="01101") else
91       in14 when (sel="01110") else
92       in15 when (sel="01111") else
93       in16 when (sel="10000") else
94       in17 when (sel="10001") else
95       in18 when (sel="10010") else
96       in19 when (sel="10011") else
97       in20 when (sel="10100") else
98       in21 when (sel="10101") else
99       in22 when (sel="10110") else
100      in23 when (sel="10111") else
101      in24 when (sel="11000") else
102      in25 when (sel="11001") else
103      in26 when (sel="11010") else
104      in27 when (sel="11011") else
```


mux1
mux1
Design Summary (Synthesized)
CPU_MIPS_PROCESSOR.vhd
DECODER.vhd
decoder (RTL1)
MUX1.vhd
mux1 (RTL1)

3/Register :

```vhdl
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created-- FPGA projects using VHDL/ VHDL
-- fpga4student.com
-- VHDL code for D Flip FLop
-- VHDL code for Rising edge D flip flop with Synchronous Reset input
Library IEEE;
USE IEEE.Std_logic_1164.all;

entity REG22 is

   port(
      Q : out std_logic_vector(31 downto 0);
      clk :in std_logic;
      async_reset: in std_logic;
      write_ena: in std_logic;
      D :in  std_logic_vector (31 downto 0)
   );
end REG22;
architecture Behavioral of REG22 is

signal ot : std_logic_vector(31 downto 0);

begin

    Q <= ot when clk'event and clk='1';

    --Q <= (others => '0') when  async_reset = '1'  ;


   ot<= D when write_ena='1' and clk'event and clk='0' ;
```



Phase two :

name	contribution
NADER MOHAEMD ELFEEL	ALU CTRL unit . THE connection of all components together to get the full MIPS PROCESSOR. Testing the outputs. 32 bit adder . jumbFunction used to connect the 4 bits in j type instructions , shift lift of 26 bit in j type instructions .

mohamed magdy Mahmoud basal	**Program counter “PC”.** • The report with mohamed Ghoneem
Mohamed Ibrahim Ghoneem	**Next instruction (PC+4)** **Most of the report**
Zakaria Alaa Fouad Youssef Eisa	**MAIN CONTROL.** Also helped in testing and connection
Mahmoud Nashaat	**Shifter** **sign extender**

Introduction:

In the realm of computer architecture and digital design, understanding the inner workings of processors is paramount. The MIPS (Microprocessor without Interlocked Pipeline Stages) architecture stands as a seminal example of Reduced Instruction Set Computing (RISC) principles, renowned for its simplicity, efficiency, and widespread adoption in both academic and industrial settings.

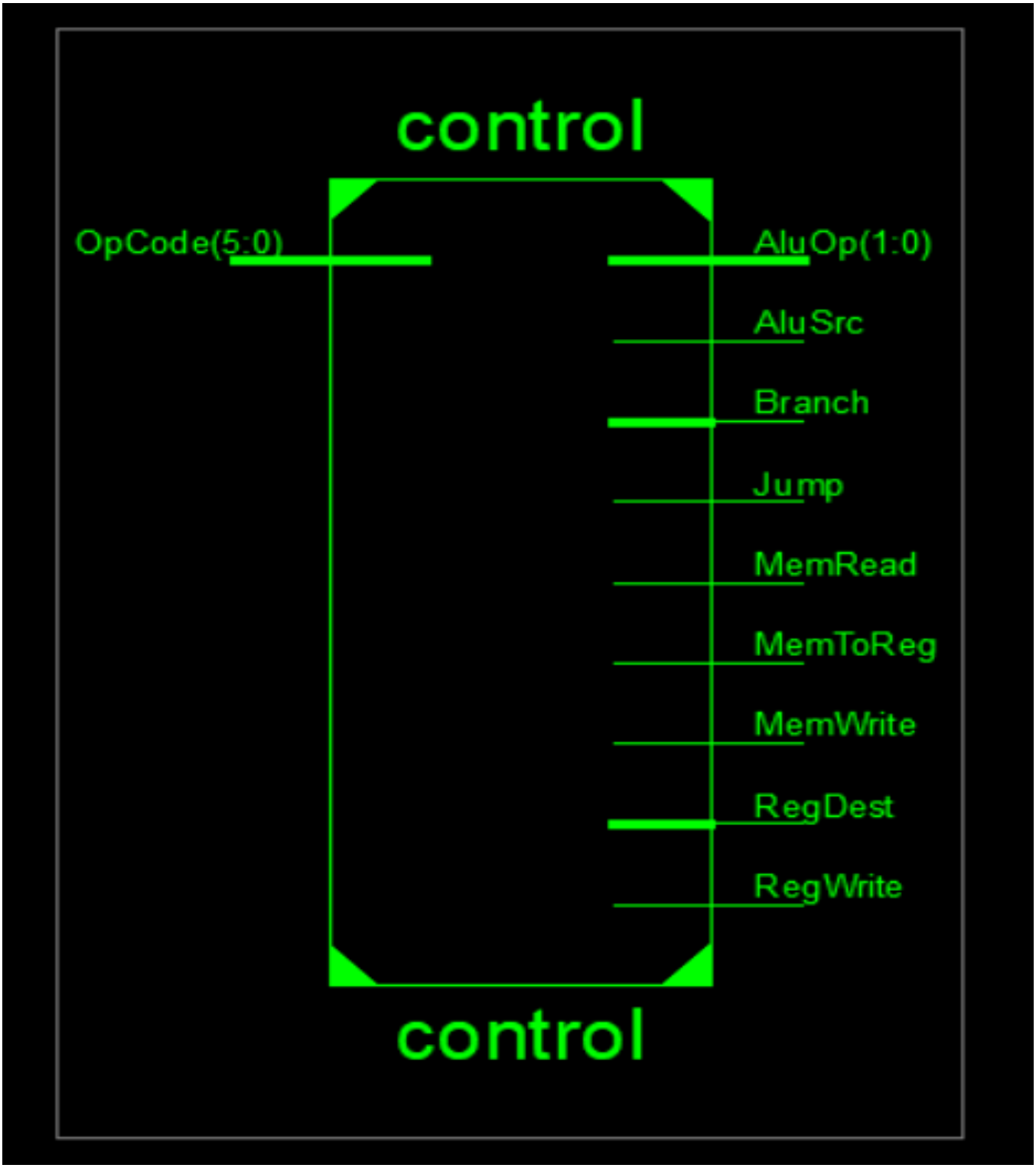
Objectives:

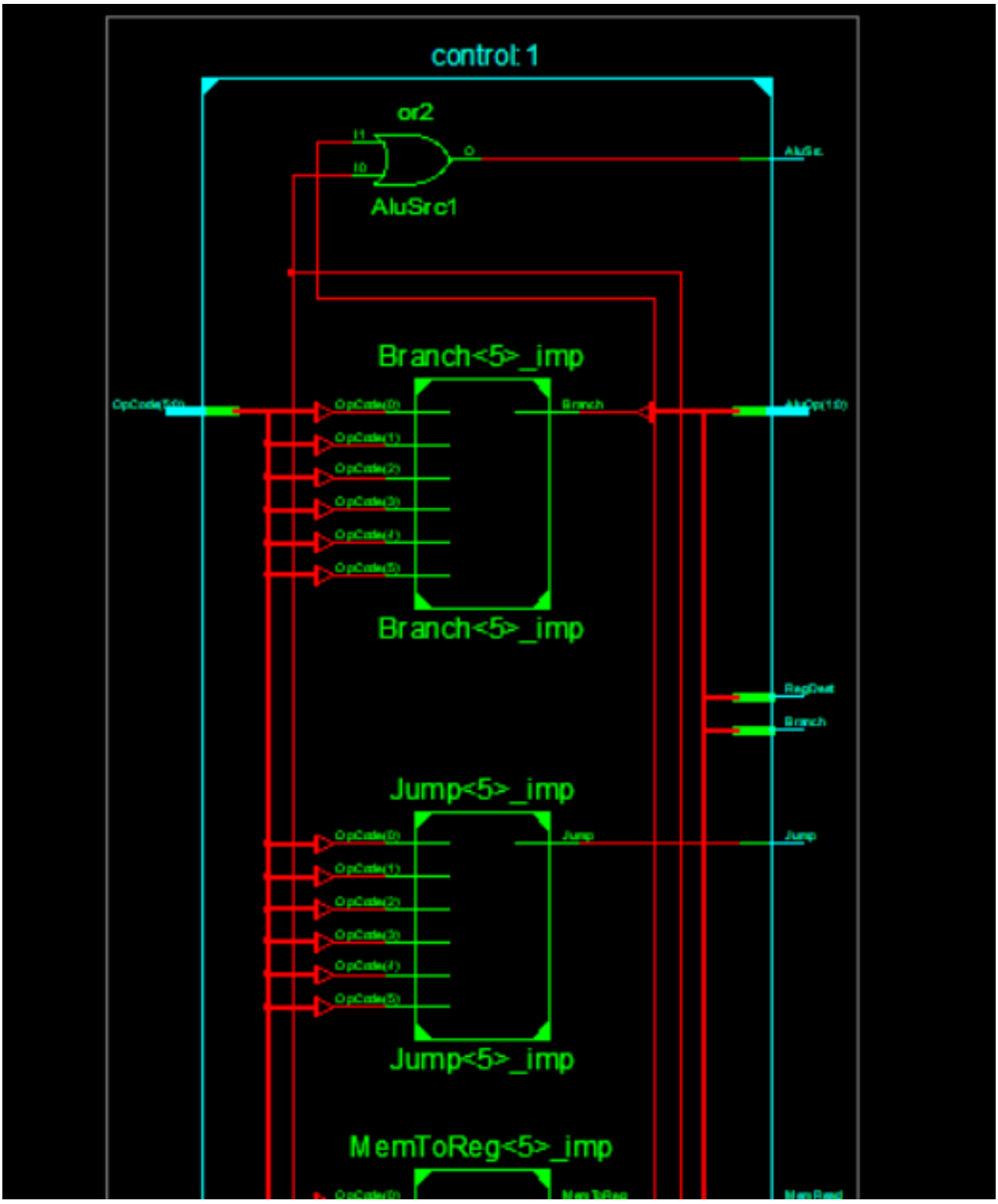
Design Implementation: Constructing the MIPS processor components using VHDL to simulate both the data and control paths.

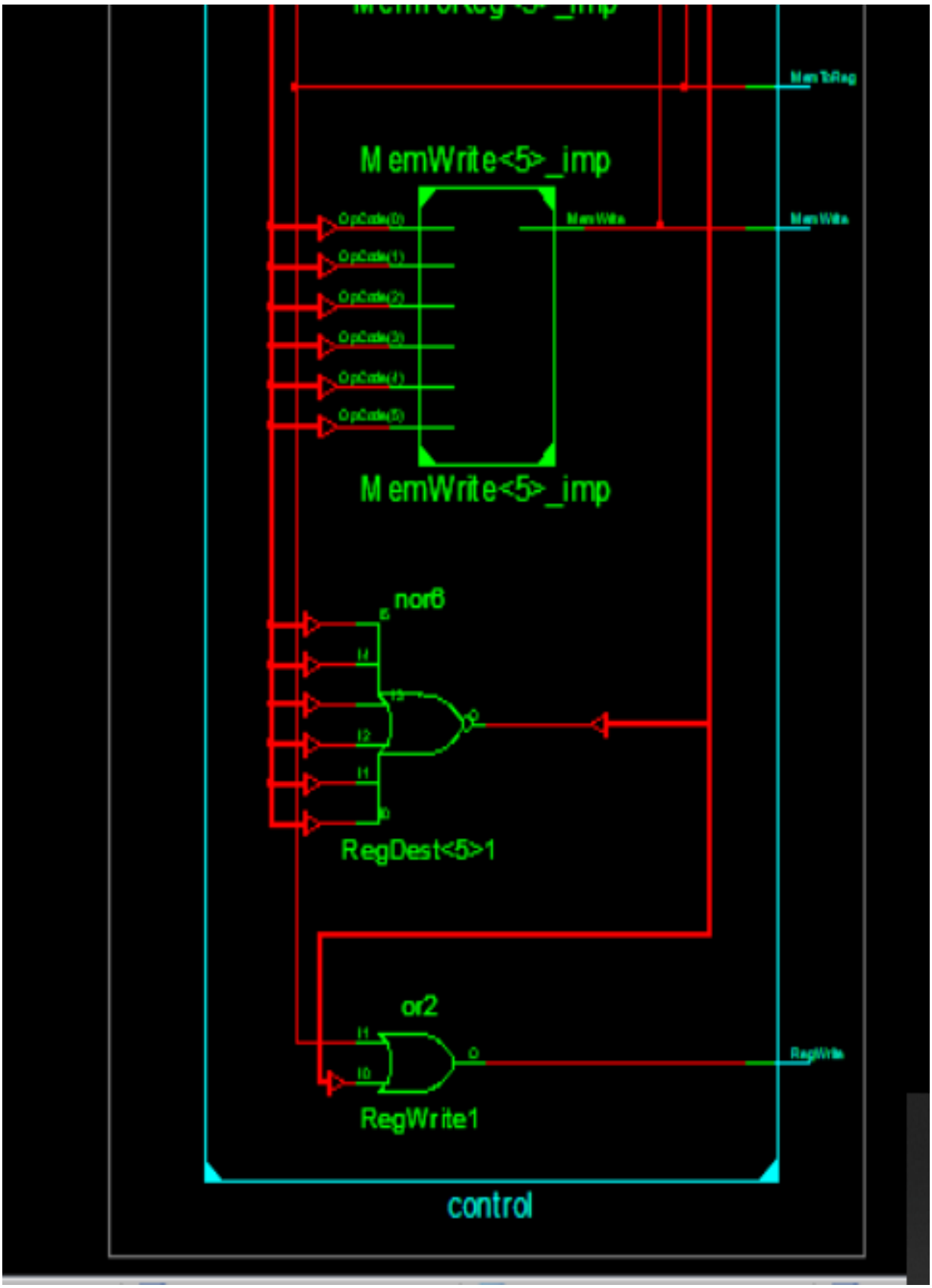
Functional Representation: Demonstrating the flow of data and control signals within the processor during instruction execution.

Verification and Validation: Rigorously testing and verifying the correctness and functionality of the MIPS processor design through comprehensive simulation.

1.control unit:




control:1
or2
I1
I0
O
AluSrc
AluSrc1
Branch<5>_imp
OpCode(0)
OpCode(1)
OpCode(2)
OpCode(3)
OpCode(4)
OpCode(5)
Branch
Branch<5>_imp
RegDest
Branch
Jump<5>_imp
Jump
Jump<5>_imp
MemToReg<5>_imp


MemWrite<5>_imp
OpCode(0)
OpCode(1)
OpCode(2)
OpCode(3)
OpCode(4)
OpCode(5)
MemWrite
MemWrite
MemWrite<5>_imp
nor6
I4
I3
I2
I1
I0
O
RegDest<5>1
or2
I1
I0
O
RegWrite
RegWrite1
control

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity control is
    Port ( OpCode : in  STD_LOGIC_VECTOR (5 downto 0);
           RegDest : out  STD_LOGIC;
           Jump : out  STD_LOGIC;
           Branch : out  STD_LOGIC;
           MemRead : out  STD_LOGIC;
           MemToReg : out  STD_LOGIC;
           AluOp : out  STD_LOGIC_VECTOR (1 downto 0);
           MemWrite : out  STD_LOGIC;
           AluSrc : out  STD_LOGIC;
           RegWrite : out  STD_LOGIC);
end control;

architecture Behavioral of control is

begin
process (OpCode)
begin

RegWrite <='0';

Case OpCode is

when "000000"=> --RType
RegDest<='1';
Jump<='0';
Branch<='0';
```

```vhdl

when "000000"=> --RType
RegDest<='1';
Jump<='0';
Branch<='0';
MemRead<='0';
MemToReg<='0';
AluOp<="10";
MemWrite<='0';
AluSrc<='0';
RegWrite<='1';

when "100011"=> -- LW
RegDest<='0';
Jump<='0';
Branch<='0';
MemRead<='1';
MemToReg<='1';
AluOp<="00";
MemWrite<='0';
AluSrc<='1';
RegWrite<='1';

when "101011"=> -- SW
RegDest<='X';
Jump<='0';
Branch<='0';
MemRead<='0';
MemToReg<='X';
AluOp<="00";
```

```
66  AluOp<="01";
67  MemWrite<='0';
68  AluSrc<='0';
69  RegWrite<='0';
70
71  when "000010"=> -- J
72  RegDest<='X';
73  Jump<='1';
74  Branch<='0';
75  MemRead<='0';
76  MemToReg<='X';
77  AluOp<="00";
78  MemWrite<='0';
79  AluSrc<='0';
80  RegWrite<='0';
81
82  when others => null; -- J
83  RegDest<='0';
84  Jump<='0';
85  Branch<='0';
86  MemRead<='0';
87  MemToReg<='0';
88  AluOp<="00";
89  MemWrite<='0';
90  AluSrc<='0';
91  RegWrite<='0';
92
93  end case;
94  end process;
95  end Behavioral;
```

```
49  when "101011"=> -- SW
50  RegDest<='X';
51  Jump<='0';
52  Branch<='0';
53  MemRead<='0';
54  MemToReg<='X';
55  AluOp<="00";
56  MemWrite<='1';
57  AluSrc<='1';
58  RegWrite<='0';
59
60  when "000100"=> -- Branch
61  RegDest<='X';
62  Jump<='0';
63  Branch<='1';
64  MemRead<='0';
65  MemToReg<='X';
66  AluOp<="01";
67  MemWrite<='0';
68  AluSrc<='0';
69  RegWrite<='0';
70
71  when "000010"=> -- J
72  RegDest<='X';
73  Jump<='1';
74  Branch<='0';
75  MemRead<='0';
76  MemToReg<='X';
77  AluOp<="00";
78  MemWrite<='0';
```

Purpose:

The control unit is responsible for generating control signals based on the opcode of the instruction. These control signals determine the behavior of various components within the MIPS processor during instruction execution.

Entity:

- **Name:** control

- **Ports:**

  - **OpCode (in):** Input representing the opcode of the instruction.

  - **RegDest (out):** Output indicating whether the destination register is written.

  - **Jump (out):** Output indicating a jump instruction.

  - **Branch (out):** Output indicating a branch instruction.

  - **MemRead (out):** Output indicating memory read operation.

  - **MemToReg (out):** Output indicating whether data should be written to the register from memory.

  - **AluOp (out):** Output indicating the ALU operation to be performed.

  - **MemWrite (out):** Output indicating memory write operation.

  - **AluSrc (out):** Output indicating the ALU source.

  - **RegWrite (out):** Output indicating whether the register should be written.

Architecture: Behavioral

- **Description:** Describes the behavior of the control unit based on the opcode of the instruction.

- **Process:** Sensitivity list includes the OpCode input signal.

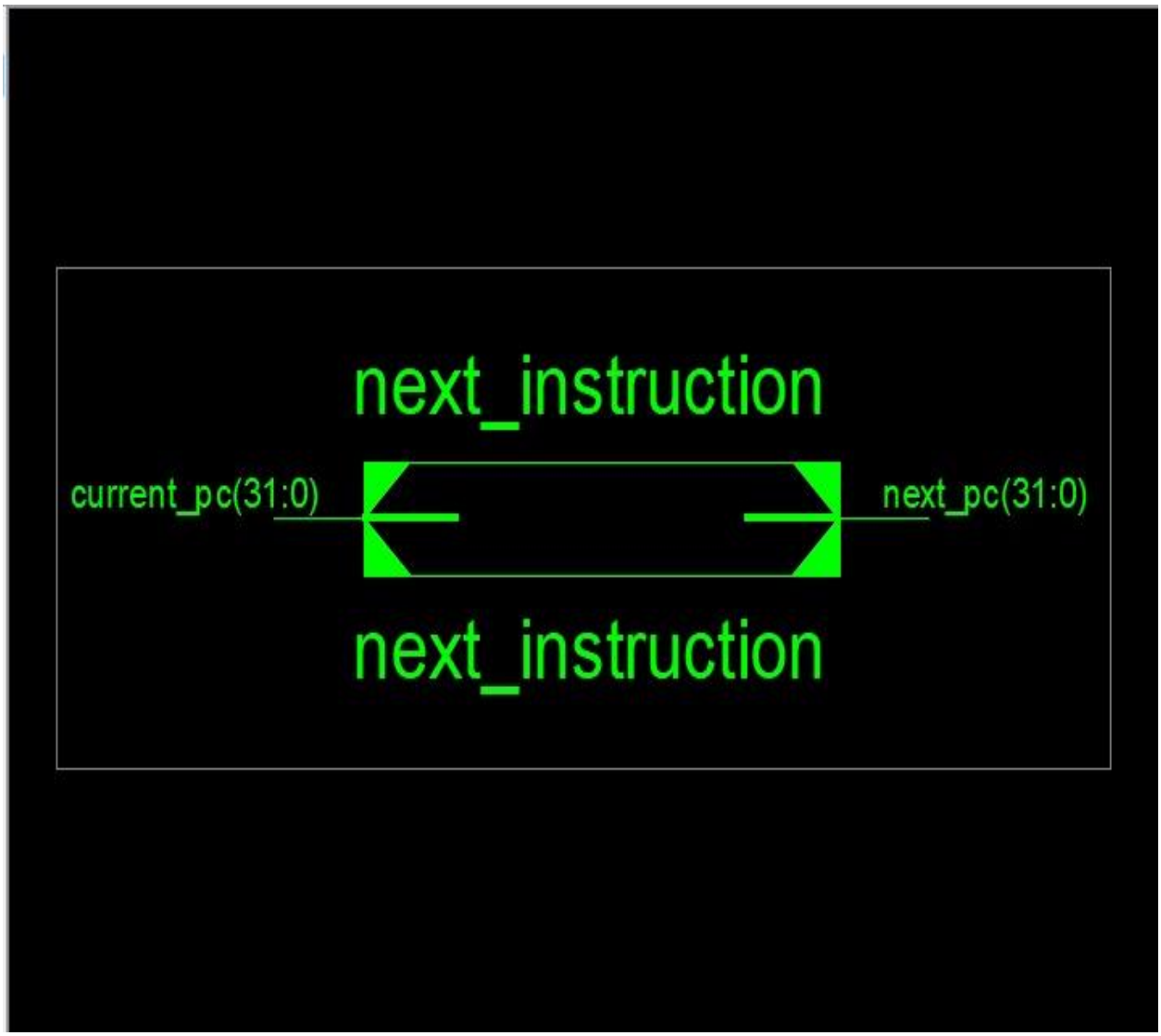
- **Behavior:** Utilizes a case statement to handle different opcode values and assign appropriate control signals.

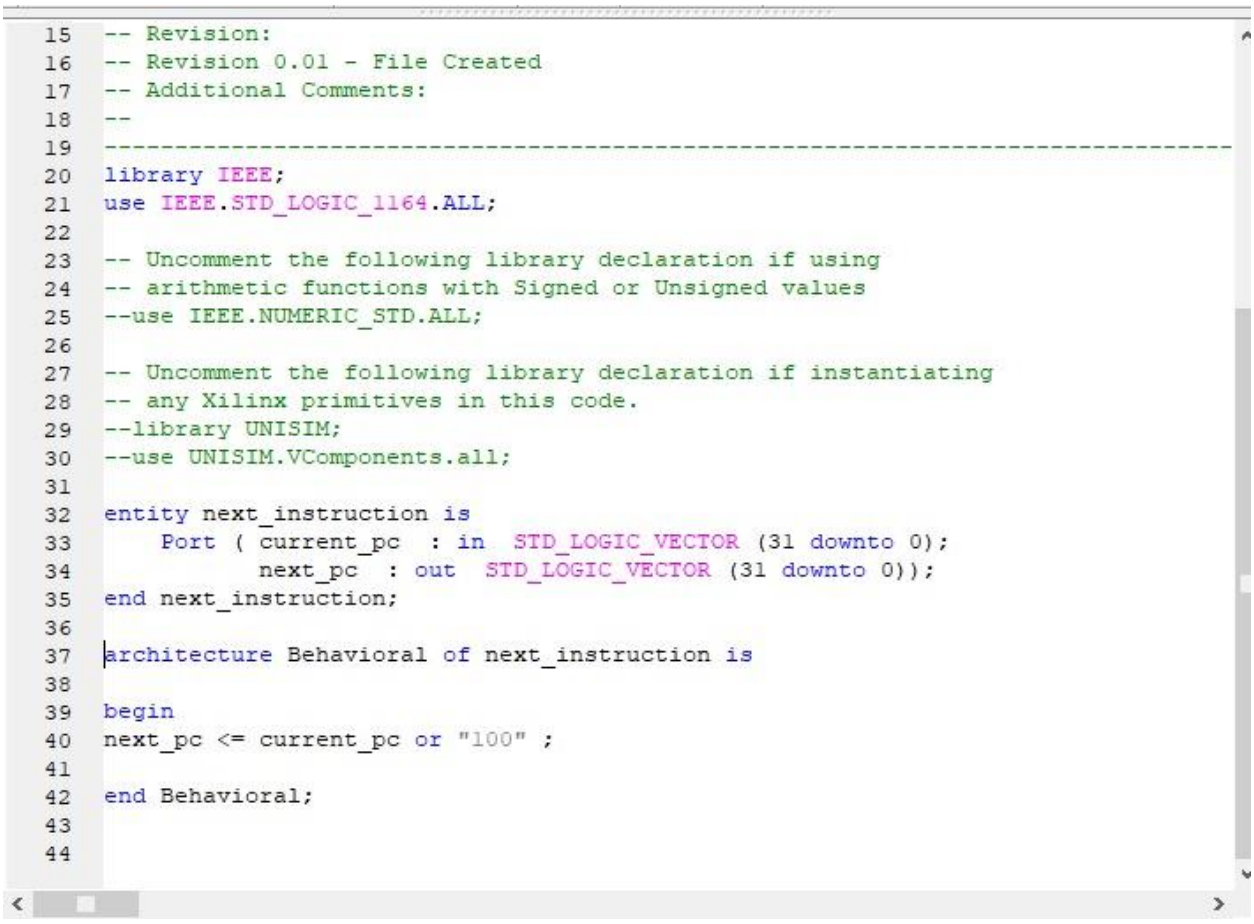
Behavior:

- For each opcode value, the control unit assigns specific values to the output ports based on the MIPS instruction type.

- Control signals are generated to control operations such as register writes, memory accesses, ALU operations, branching, and jumping.

Next instruction:


next_instruction
current_pc(31:0)
next_pc(31:0)
next_instruction



```
15  -- Revision:
16  -- Revision 0.01 - File Created
17  -- Additional Comments:
18  --
19  ----------------------------------------------------------------------------------
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22
23  -- Uncomment the following library declaration if using
24  -- arithmetic functions with Signed or Unsigned values
25  --use IEEE.NUMERIC_STD.ALL;
26
27  -- Uncomment the following library declaration if instantiating
28  -- any Xilinx primitives in this code.
29  --library UNISIM;
30  --use UNISIM.VComponents.all;
31
32  entity next_instruction is
33      Port ( current_pc  : in  STD_LOGIC_VECTOR (31 downto 0);
34             next_pc  : out  STD_LOGIC_VECTOR (31 downto 0));
35  end next_instruction;
36
37  architecture Behavioral of next_instruction is
38
39  begin
40  next_pc <= current_pc or "100" ;
41
42  end Behavioral;
43
44
```

Purpose:

The purpose of the "next_instruction" module is to compute the address of the next instruction to be fetched based on the current program counter (PC) value.

Entity:

- **Name:** next_instruction

- **Ports:**

  - **current_pc (in):** Input representing the current program counter value.

  - **next_pc (out):** Output representing the next program counter value.

Architecture: Behavioral

- **Description:** Describes the behavior of the module.

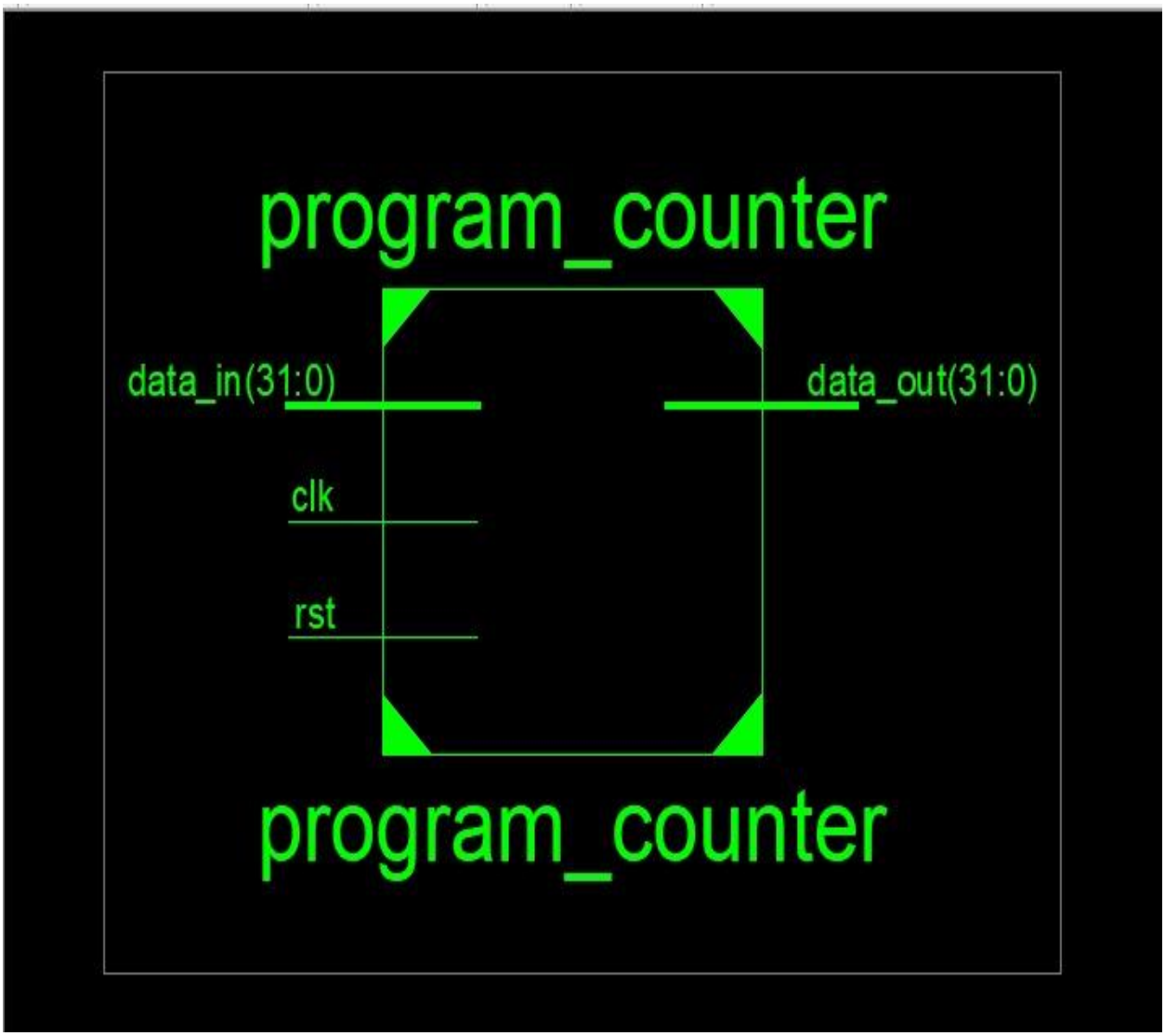
- **Process:** There is no process defined in this architecture; instead, the next program counter value is directly assigned in the concurrent statement.

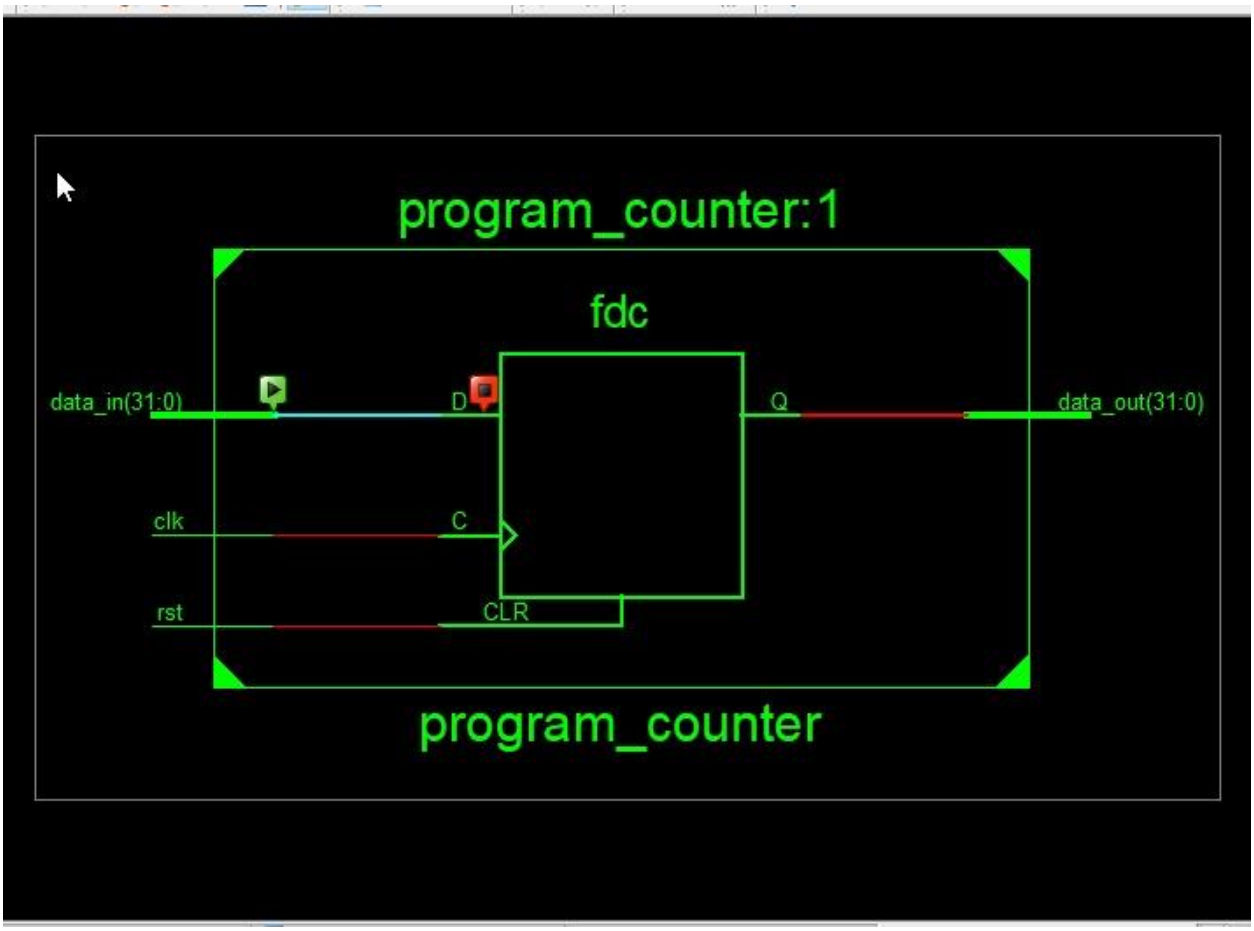
Behavior:

- The next program counter value is computed by logically OR-ing the current PC value with the binary value "100" (equivalent to adding 4 to the PC).

- This operation effectively increments the current PC by 4, assuming each instruction in memory is stored in a word-aligned manner (i.e., each instruction occupies 4 bytes).

Progam counter (pc):




program_counter:1
fdc
data_in(31:0)
D
Q
data_out(31:0)
clk
C
rst
CLR
program_counter

```
19  ----------------------------------------------------------------------------------
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22
23  -- Uncomment the following library declaration if using
24  -- arithmetic functions with Signed or Unsigned values
25  --use IEEE.NUMERIC_STD.ALL;
26
27  -- Uncomment the following library declaration if instantiating
28  -- any Xilinx primitives in this code.
29  --library UNISIM;
30  --use UNISIM.VComponents.all;
31
32  entity program_counter is
33      Port ( clk : in  STD_LOGIC;
34             data_in : in  STD_LOGIC_VECTOR (31 downto 0);
35             data_out  : out  STD_LOGIC_VECTOR (31 downto 0);
36             rst : in  STD_LOGIC);
37  end program_counter;
38
39  architecture Behavioral of program_counter is
40
41  begin
42  process(clk,rst)
43
44  begin
45  if rst='1'  then
46     data_out<=(others=>'0');
47  elsif clk' event and clk='1' then
48     data_out<=data_in;
49     end if ;
50     end process;
51  end Behavioral;
52
53
```

Purpose:

The purpose of the "program_counter" module is to maintain and update the value of the program counter in a digital system. The PC value represents the address of the next instruction to be fetched from memory.

Entity:

- **Name:** program_counter

- **Ports:**

  - **clk (in):** Input clock signal for synchronization.

  - **data_in (in):** Input representing the new PC value to be loaded.

  - **data_out (out):** Output representing the current PC value.

- **rst (in):** Input reset signal for resetting the PC value.

Architecture: Behavioral

- **Description:** Describes the behavior of the module.

- **Process:** A synchronous process is used to update the PC value based on the clock and reset signals.

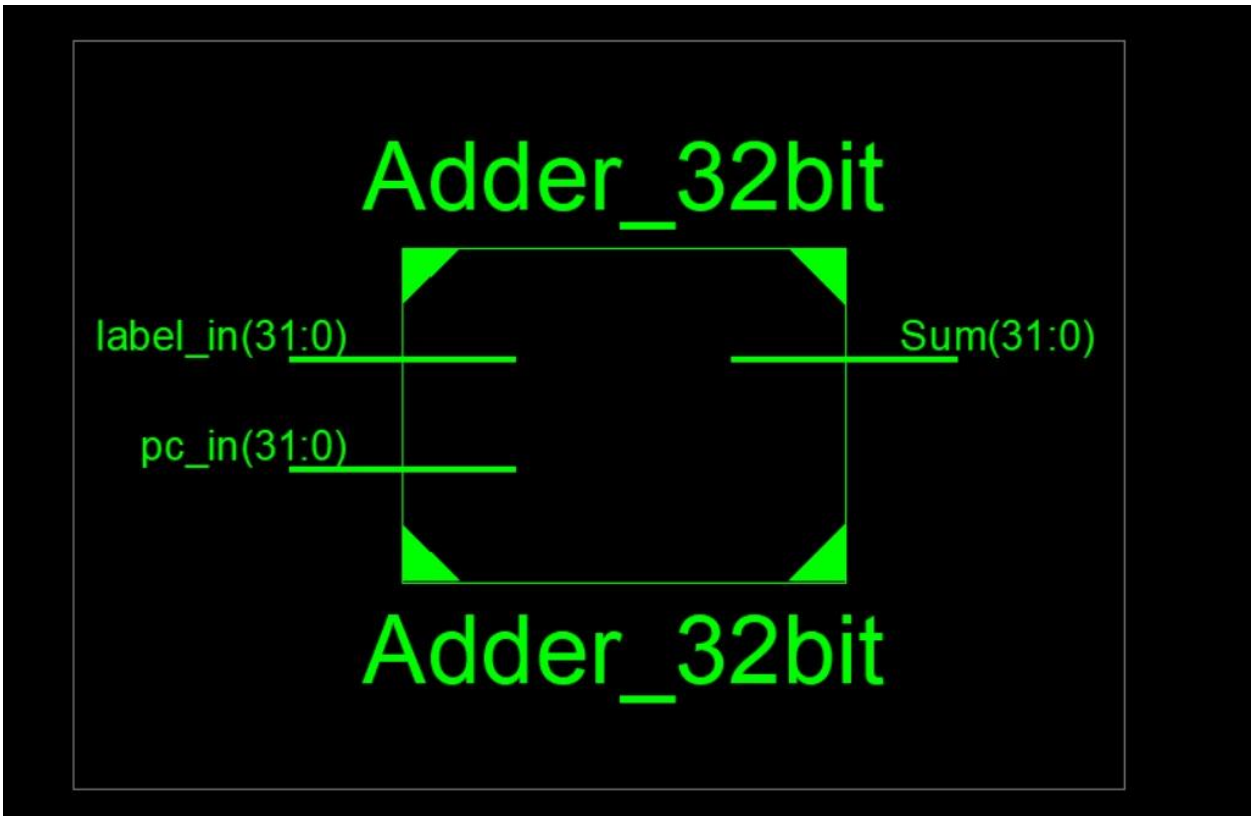
Behavior:

- **Reset Handling:** If the reset signal (`rst`) is asserted (`'1'`), the PC value (`data_out`) is set to zero, indicating the start of program execution.

- **Clock Edge Detection:** On each rising clock edge (`clk'event and clk='1'`), the PC value is updated with the new value provided at the `data_in` input.

- **Synchronization:** The process is sensitive to changes in the clock and reset signals to ensure synchronous operation.

32 - bit adder :




```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Adder_32bit is
    port (
        pc_in : in  std_logic_vector(31 downto 0);
        label_in : in  std_logic_vector(31 downto 0);
        Sum : out std_logic_vector(31 downto 0)
    );
end entity Adder_32bit;

architecture Behavioral of Adder_32bit is
begin
    process(pc_in, label_in)
        variable pc_internal, labl_internal, Sum_internal : signed(31 downto 0);
    begin
        pc_internal := signed(pc_in);
        labl_internal := signed(label_in);
        Sum_internal := pc_internal + labl_internal;
        Sum <= std_logic_vector(Sum_internal);
    end process;
end architecture Behavioral;
```

Purpose:

The purpose of the 32-bit adder module is to perform arithmetic addition on two 32-bit binary numbers. In the context of a MIPS processor, this component is crucial for executing arithmetic instructions such as addition, subtraction, and certain logical operations.

Entity:

- **Name:** adder_32bit

- **Ports:**

  - **A (in):** Input representing the first 32-bit operand.

  - **B (in):** Input representing the second 32-bit operand.

  - **Sum (out):** Output representing the result of the addition operation, also a 32-bit value.

Architecture: Behavioral

- **Description:** Describes the behavior of the module.

- **Process:** Utilizes a combinational process to compute the sum of the input operands.

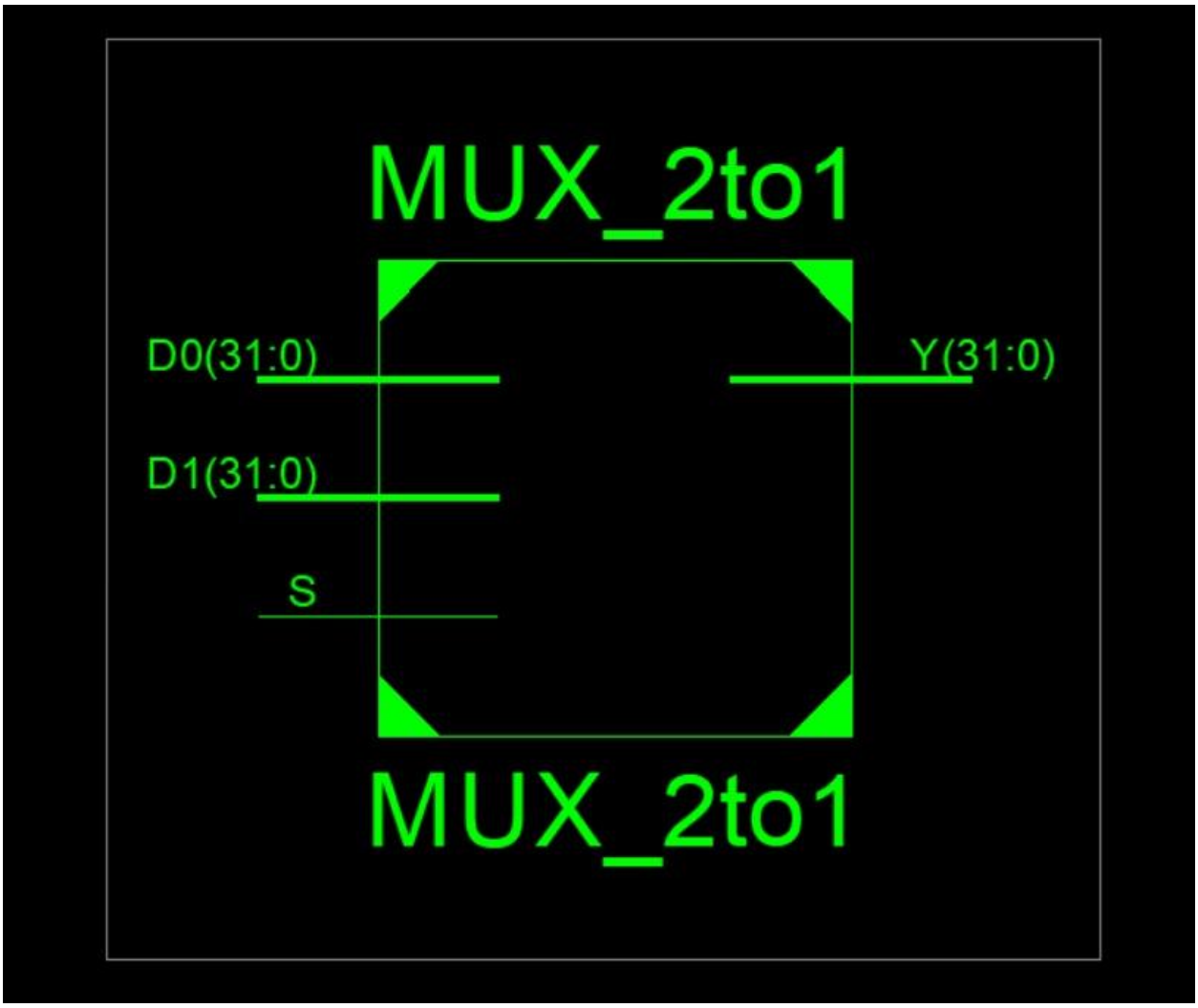
Behavior:

- **Addition Operation:** On each clock cycle or in a purely combinational manner, the module computes the sum of the two input operands (A and B).

- **32-Bit Addition:** Performs addition on each bit of the input operands, taking into account carry bits from lower-order bits to higher-order bits.

- **Overflow Handling:** Detects and handles overflow conditions, ensuring that the result fits within the 32-bit output range.

2x1 MUX:



Purpose:

The 2-to-1 multiplexer (MUX) module is a fundamental component in digital circuit design, allowing the selection of one of two input signals based on a control signal. In the context of a MIPS processor, the MUX is commonly used for selecting between two data sources or control signals.

Entity:

- **Name:** mux_2to1

- **Ports:**

  - **A (in):** Input representing the first data signal.

  - **B (in):** Input representing the second data signal.

  - **Select (in):** Input control signal determining which input signal to pass to the output.

  - **Y (out):** Output representing the selected data signal.

Architecture: Behavioral

- **Description:** Describes the behavior of the module.

- **Process:** Utilizes a combinational process to select the appropriate input signal based on the control signal.

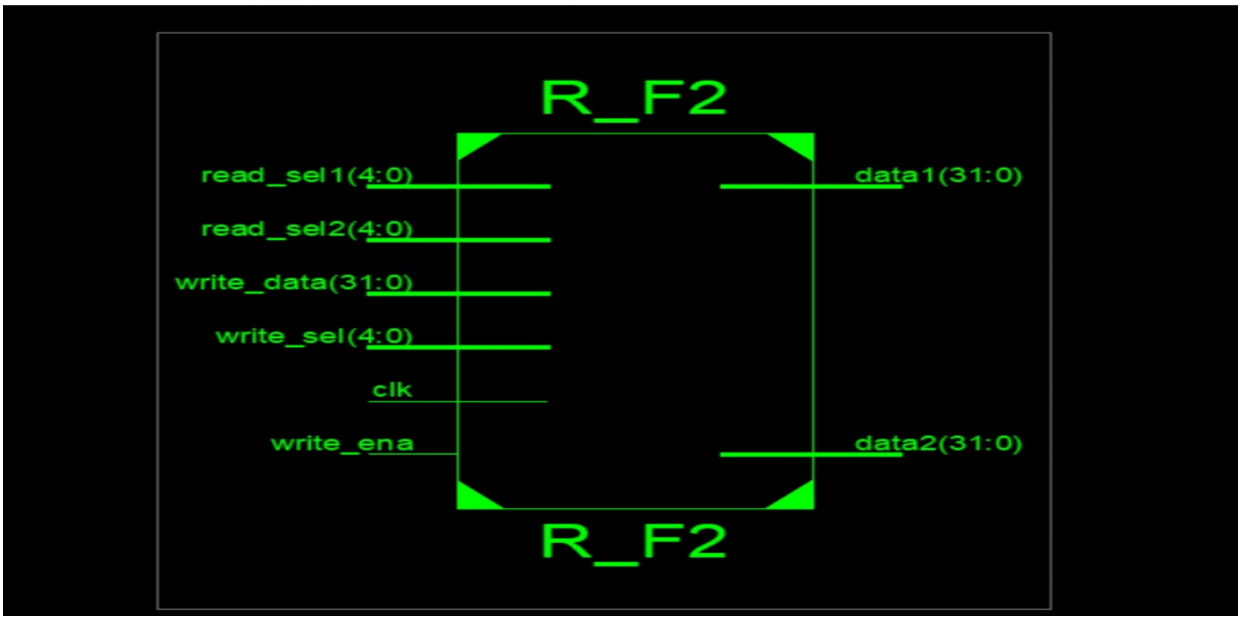
Behavior:

- **Selection Logic:** The module selects either input signal A or B based on the value of the control signal (Select).

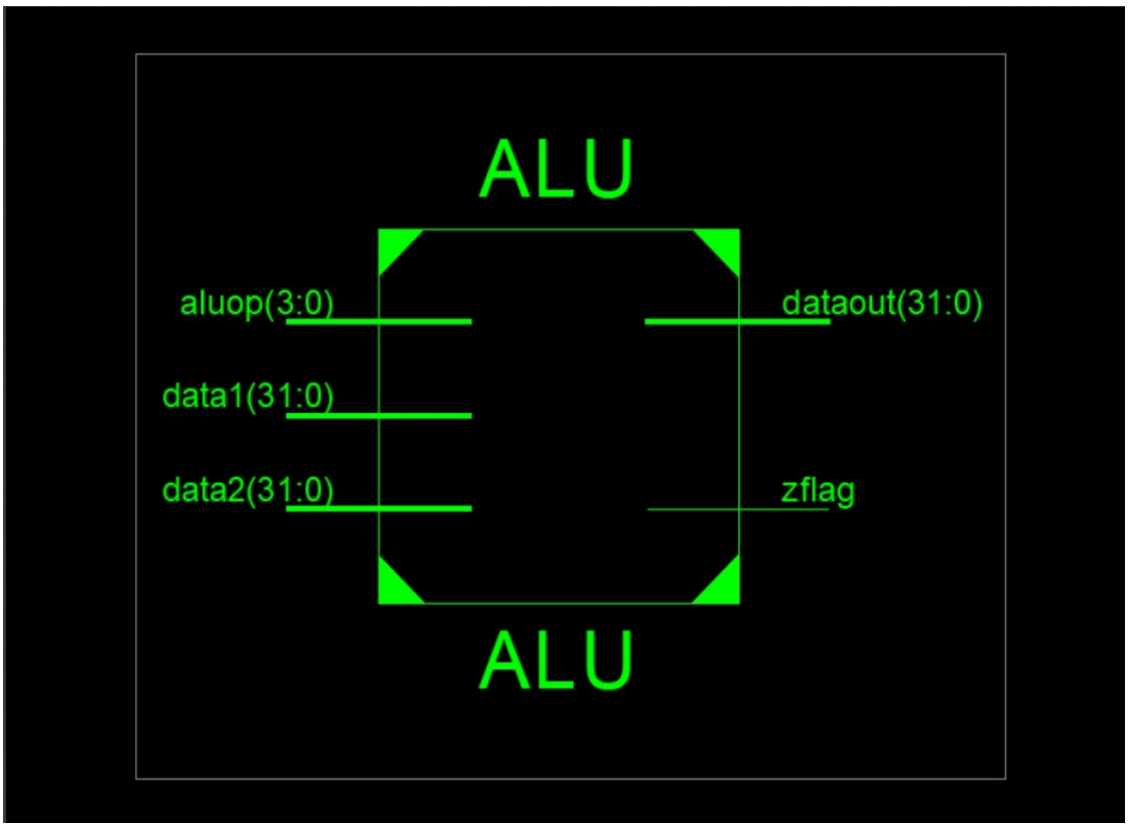
- **Output Assignment:** If the control signal is '0', the output (Y) is assigned the value of input signal A. If the control signal is '1', the output (Y) is assigned the value of input signal B.

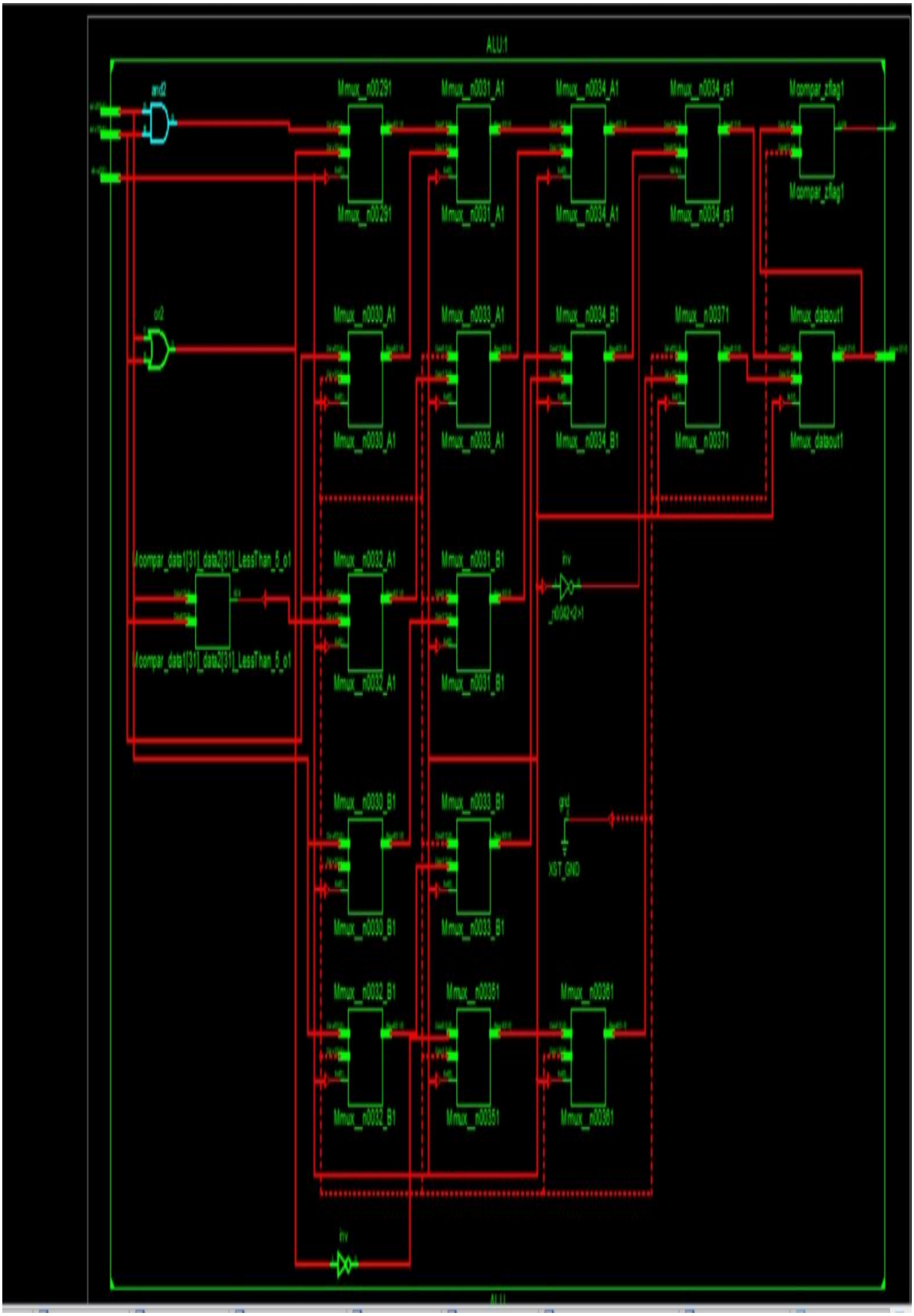
- **Synchronization:** The module's behavior is purely combinational, responding immediately to changes in the input and control signals.

Register file as was mentioned before:

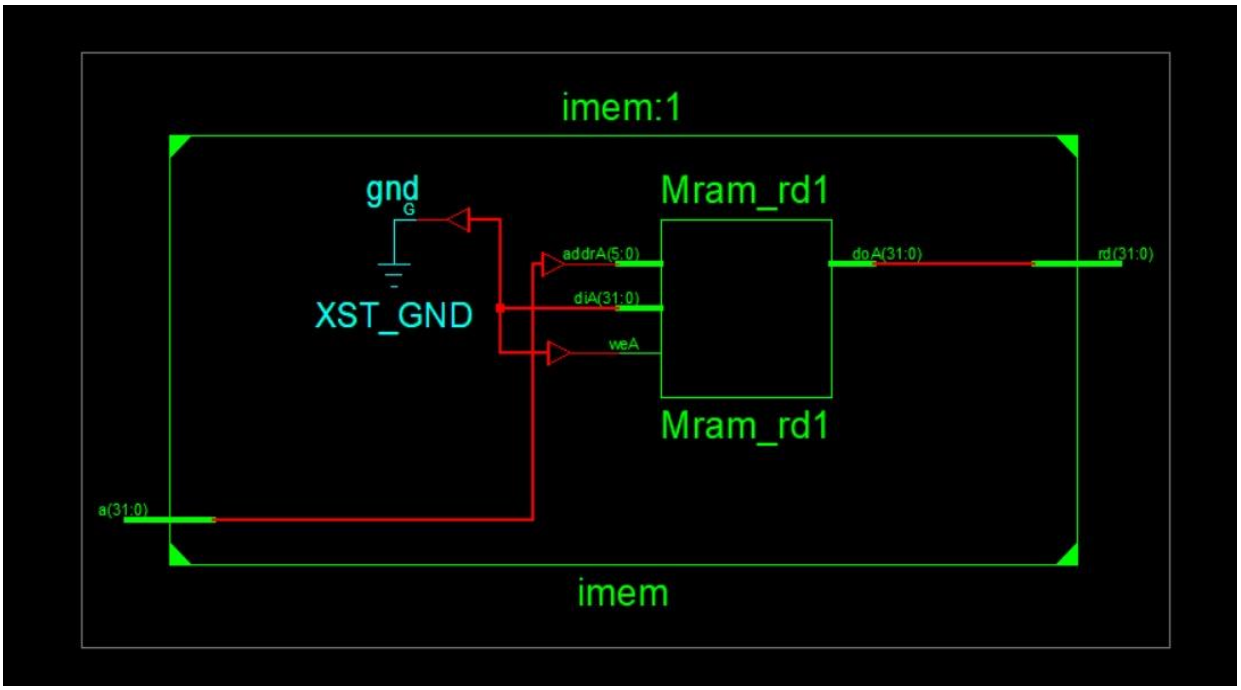


Alu as was mentioned before:




ALU1
Mmux_n0029_1
Mmux_n0031_A1
Mmux_n0034_A1
Mmux_n0034_rs1
Mcompar_zflag1
Mmux_n0030_A1
Mmux_n0033_A1
Mmux_n0034_B1
Mmux_n0037_1
Mmux_dataout1
Mcompar_data1[31]_data2[31]_LessThan_5_o1
Mmux_n0032_A1
Mmux_n0031_B1
inv
_n0042<2>1
gnd
XST_GND
Mmux_n0030_B1
Mmux_n0033_B1
Mmux_n0032_B1
Mmux_n0036_1
Mmux_n0038_1

The (imem):



Purpose:

The Instruction Memory (imem) component is integral to the Processor architecture, responsible for storing and providing instructions to the processor for execution. It serves as the primary source of program instructions during program execution.

Entity:

- **Name:** imem

Ports:

- **a (in):** Input signal representing the memory address from which to read the instruction.

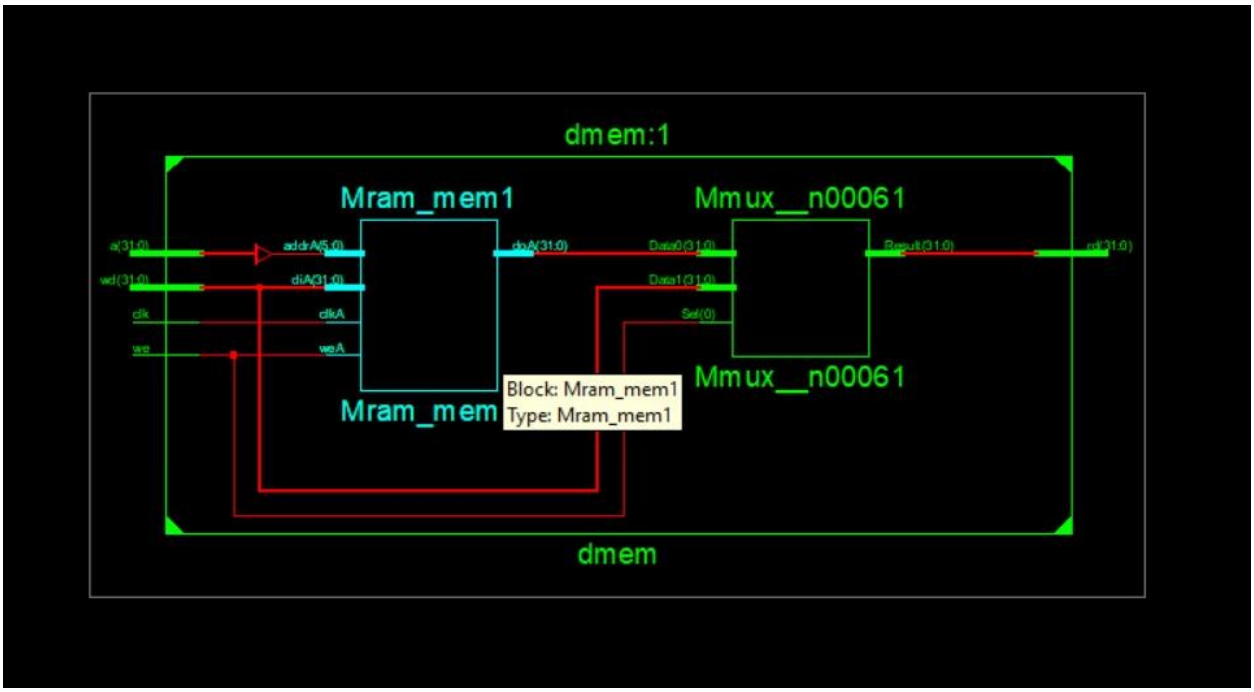
- **rd (out):** Output signal containing the instruction fetched from the memory at the specified address.

Behavior:

- **Instruction Fetch:** When provided with a valid memory address (`a`), the Instruction Memory retrieves the instruction stored at that address and outputs it through the `rd` port.

- **Read-Only:** The Instruction Memory operates as a read-only memory (ROM), meaning it does not support write operations. It only provides instructions stored in memory for fetching.

The (dmem):



Purpose:

The data memory (dmem) component serves as the storage unit for data accessed and manipulated during program execution within the processor. It provides a means for storing and retrieving data values, supporting both read and write operations.

Entity:

- **Name:** dmem

- **Ports:**

  - **clk (in):** Input clock signal for synchronization.

  - **we (in):** Input signal indicating write enable, determining whether data is written to the memory.

  - **a (in):** Input address signal representing the memory address for read or write operations.

  - **wd (in):** Input data signal to be written to the memory when write enable (`we`) is asserted.

- **rd (out):** Output data signal representing the data read from the memory corresponding to the input address (`a`).

Behavior:

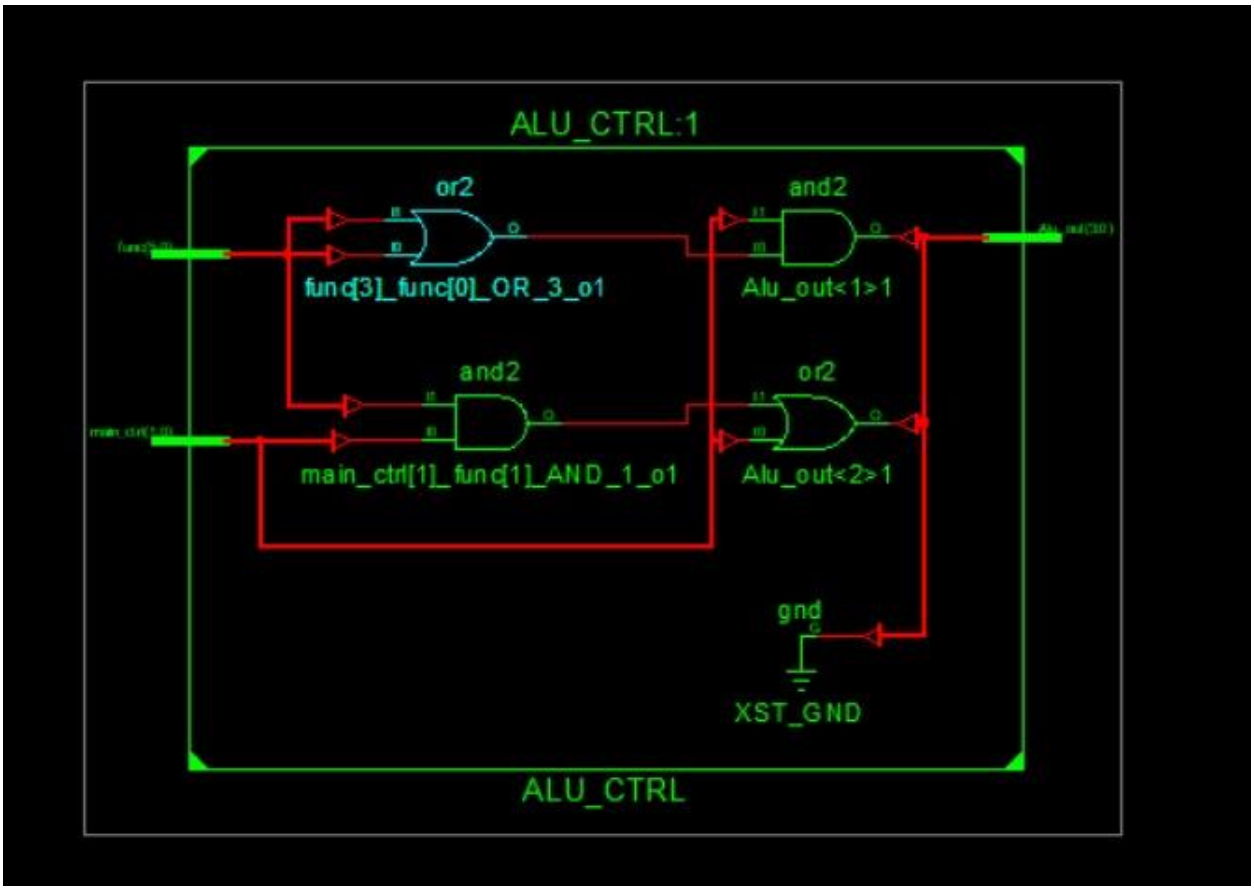
- **Read Operation:** When the write enable (`we`) signal is not asserted, the memory reads the data value stored at the address specified by the input address signal (`a`) and outputs it through the `rd` port.

- **Write Operation:** When the write enable (`we`) signal is asserted, the memory writes the input data signal (`wd`) to the address specified by the input address signal (`a`).

Synchronization:

- The data memory (dmem) component operates synchronously with the system clock (`clk`), ensuring that read and write operations occur reliably and in coordination with other processor components.

ALU CONTROL:




```vhdl
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ALU_CTRL is
    Port ( main_ctrl : in  STD_LOGIC_VECTOR (1 downto 0);
           func : in  STD_LOGIC_VECTOR (5 downto 0);
           Alu_out : out  STD_LOGIC_VECTOR (3 downto 0));
end ALU_CTRL;

architecture Behavioral of ALU_CTRL is
signal result : STD_LOGIC_VECTOR(2 downto 0);
begin

  process(main_ctrl,func)
   begin
    Alu_out(3) <= '0';
    Alu_out(2) <= main_ctrl(0) or (main_ctrl(1) and func(1));
    Alu_out(1) <= not main_ctrl(1) or not func(2);
    Alu_out(1) <= (func(3) or func(0)) and main_ctrl(1);

   end process;
```

Purpose:

The ALU Controller is a vital component within the Processor architecture responsible for generating control signals that determine the operation to be performed by the Arithmetic Logic Unit (ALU). It interprets the instruction opcode and other relevant signals to select the appropriate ALU operation.

Entity:

- **Name:** ALU_CTRL

Ports:

- **main_ctrl (in):** Input signal representing the primary control signal for selecting ALU operation.

- **func (in):** Input signal containing additional control information, such as function code from the instruction.

- **Alu_out (out):** Output signal representing the control signals to be sent to the ALU, specifying the desired operation.

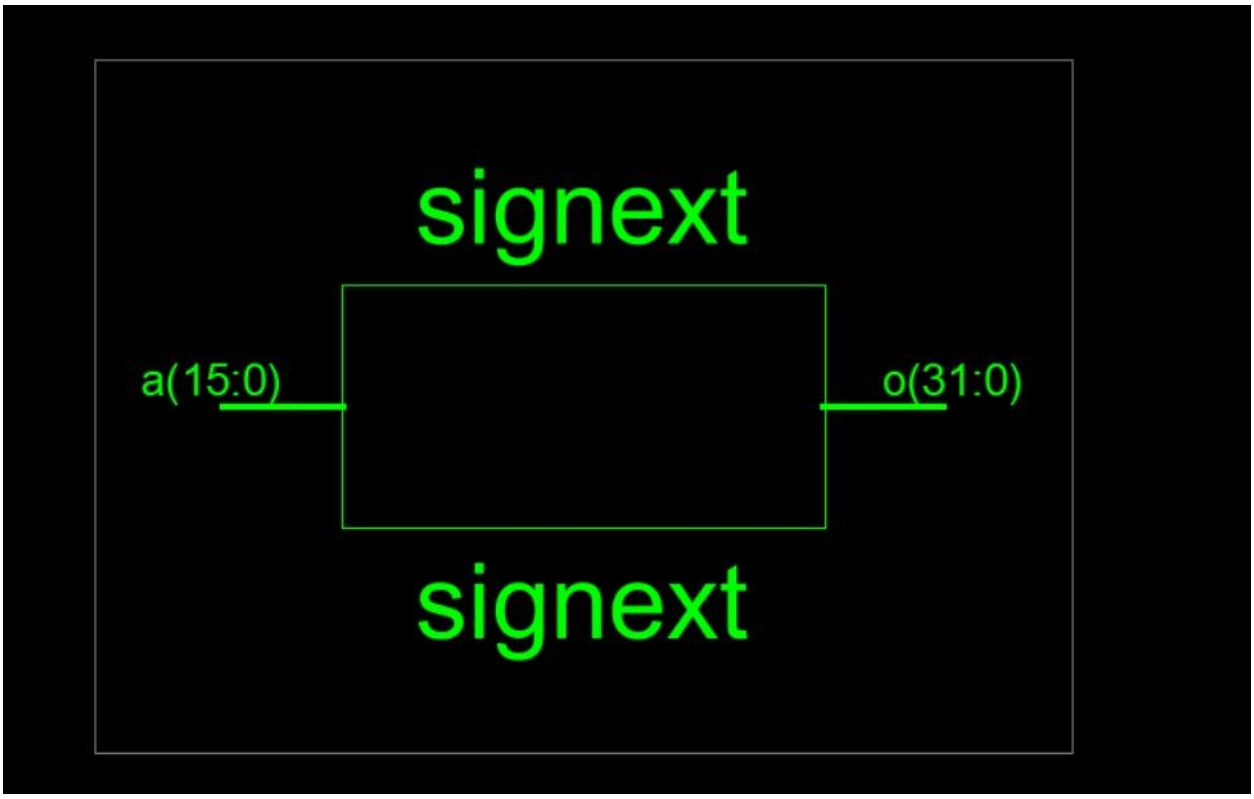
Behavior:

- **Operation Selection:** Based on the primary control signal (`main_ctrl`) and additional control information (`func`), the ALU Controller determines the operation to be performed by the ALU.

- **Control Signal Generation:** Generates control signals (`Alu_out`) encoding the selected ALU operation, enabling the ALU to execute the desired arithmetic or logical operation.

- **ALU Operation Mapping:** Maps specific combinations of input signals (`main_ctrl` and `func`) to corresponding ALU operations, such as addition, subtraction, bitwise AND/OR, or comparison.

SIGN EXTENDER:




```
13  -- Dependencies:
14  --
15  -- Revision:
16  -- Revision 0.01 - File Created
17  -- Additional Comments:
18  --
19  ----------------------------------------------------------------------------------
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22
23  -- Uncomment the following library declaration if using
24  -- arithmetic functions with Signed or Unsigned values
25  --use IEEE.NUMERIC_STD.ALL;
26
27  -- Uncomment the following library declaration if instantiating
28  -- any Xilinx primitives in this code.
29  --library UNISIM;
30  --use UNISIM.VComponents.all;
31
32  entity signext is
33      Port ( a : in  STD_LOGIC_VECTOR (15 downto 0);
34             o : out  STD_LOGIC_VECTOR (31 downto 0));
35  end signext;
36
37  architecture Behavioral of signext is
38
39  begin
40  o<= X"ffff" &a when a(15)='1' else
41  X"0000" &a;
42  end Behavioral;
```

Purpose:

The Sign Extension Unit is a key component within the Processor architecture responsible for extending the sign of a 16-bit immediate value to a 32-bit value. It ensures proper handling of signed values in instructions, particularly those involving immediate values.

Entity:

- **Name:** signext

Ports:

- **a (in):** Input signal containing the 16-bit immediate value to be sign-extended.

- **o (out):** Output signal containing the sign-extended 32-bit value.

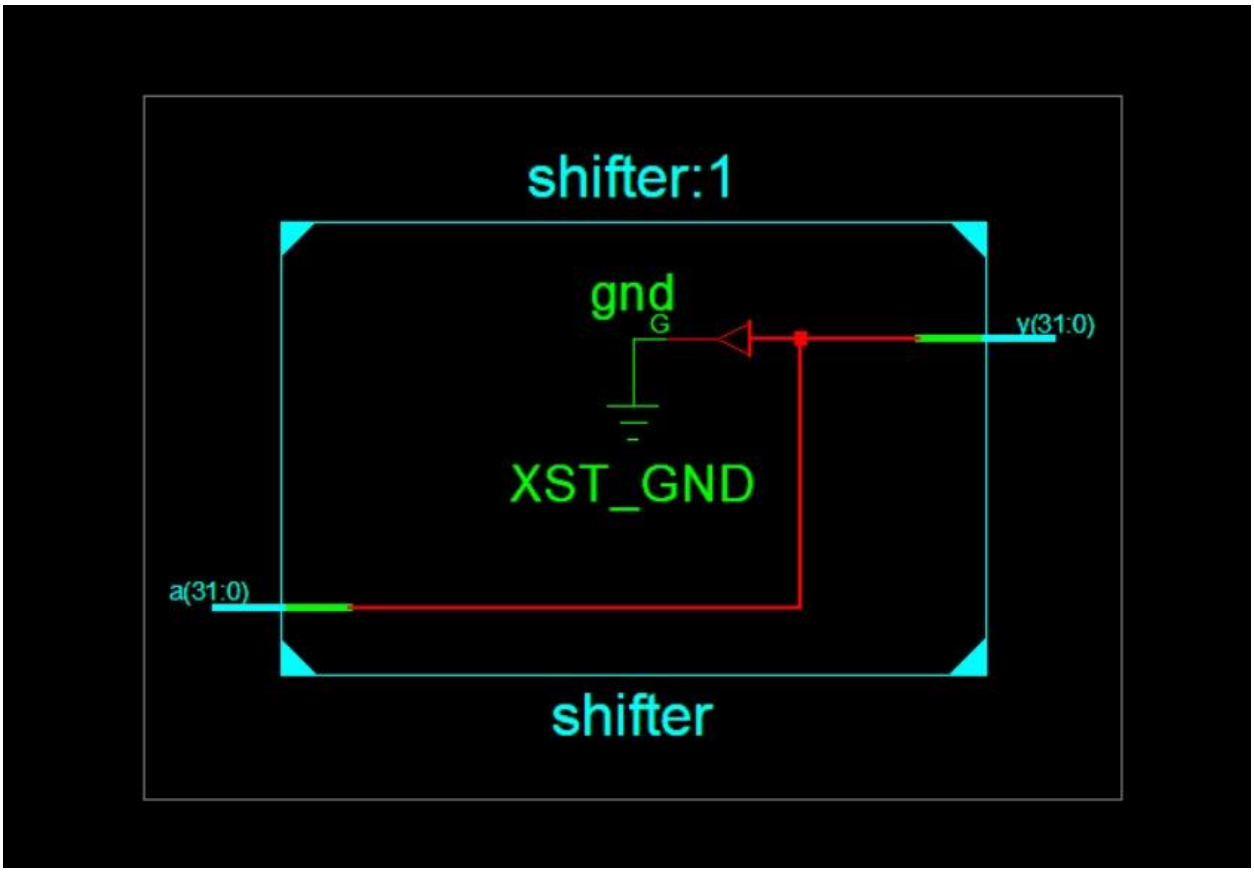
Behavior:

- **Sign Extension:** The Sign Extension Unit takes a 16-bit immediate value as input and extends its sign to create a 32-bit value.

- **Sign Bit Replication:** Replicates the sign bit (the most significant bit) of the input value to fill the additional 16 bits, ensuring that the sign is preserved during extension.

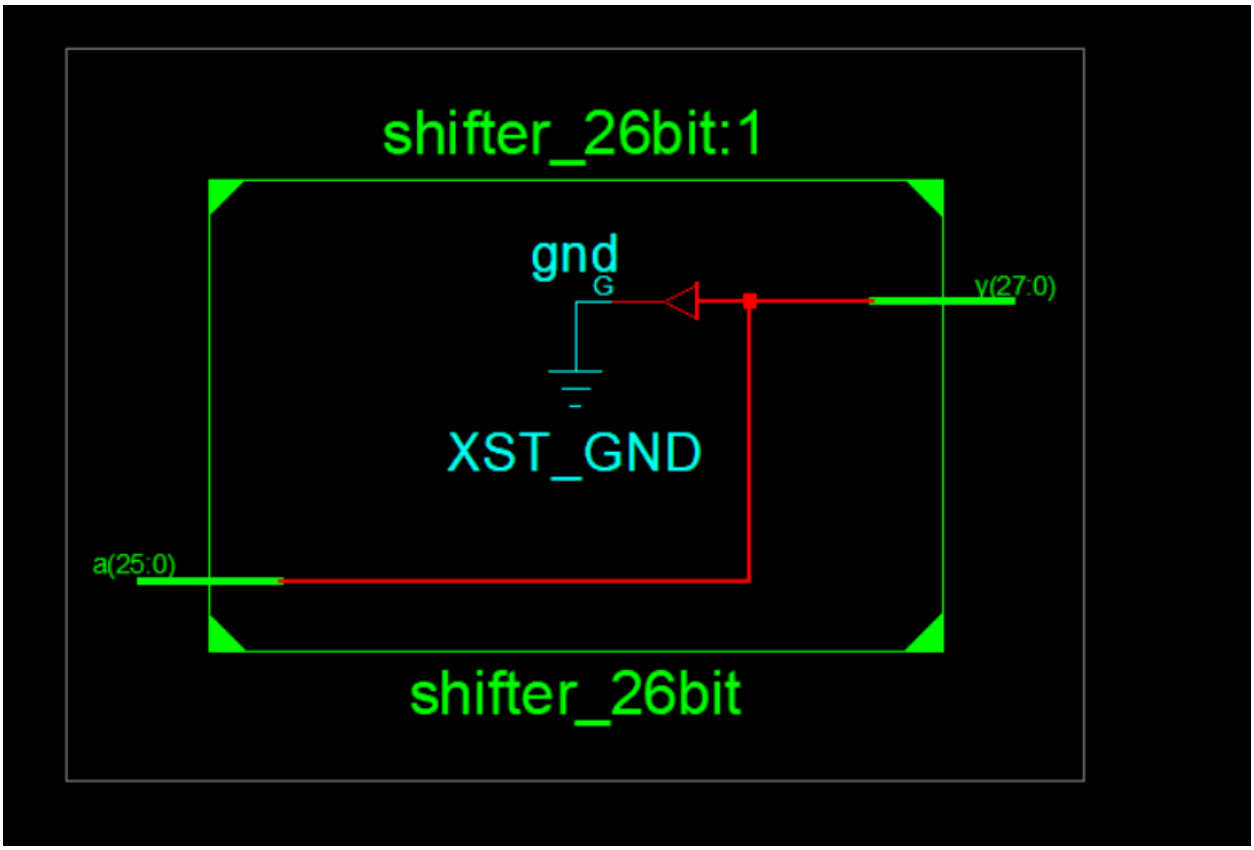
- **Two's Complement Representation:** If the original value is negative, the sign bit is extended with '1's; if positive, it is extended with '0's.

Shifter:




```
14  --
15  -- Revision:
16  -- Revision 0.01 - File Created
17  -- Additional Comments:
18  --
19  ----------------------------------------------------------------------------------
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22
23  -- Uncomment the following library declaration if using
24  -- arithmetic functions with Signed or Unsigned values
25  --use IEEE.NUMERIC_STD.ALL;
26
27  -- Uncomment the following library declaration if instantiating
28  -- any Xilinx primitives in this code.
29  --library UNISIM;
30  --use UNISIM.VComponents.all;
31
32  entity shifter is
33      Port ( a : in  STD_LOGIC_VECTOR (31 downto 0);
34             y : out  STD_LOGIC_VECTOR (31 downto 0));
35  end shifter;
36
37  architecture Behavioral of shifter is
38
39  begin
40
41   y<=a(29 downto 0) & "00";
42
43  end Behavioral;
```

Another shifter (for adding “00” to the j instructions):




```
14  --
15  -- Revision:
16  -- Revision 0.01 - File Created
17  -- Additional Comments:
18  --
19  ----------------------------------------------------------------------------------
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22
23  -- Uncomment the following library declaration if using
24  -- arithmetic functions with Signed or Unsigned values
25  --use IEEE.NUMERIC_STD.ALL;
26
27  -- Uncomment the following library declaration if instantiating
28  -- any Xilinx primitives in this code.
29  --library UNISIM;
30  --use UNISIM.VComponents.all;
31
32  entity shifter_26bit is
33      Port ( a : in  STD_LOGIC_VECTOR (25 downto 0);
34             y : out  STD_LOGIC_VECTOR (27 downto 0));
35  end shifter_26bit;
36
37  architecture Behavioral of shifter_26bit is
38
39  begin
40
41   y<=a(25 downto 0) & "00";
42
43  end Behavioral;
```

Shifter Component Description

Purpose:

The Shifter component is essential within the Processor architecture, responsible for performing shift operations on input data. It allows for the manipulation of data by shifting its bits left or right based on control signals.

Entity:

- **Name:** shifter

Ports:

- **a (in):** Input signal containing the data to be shifted.

- **y (out):** Output signal containing the shifted data.

Behavior:

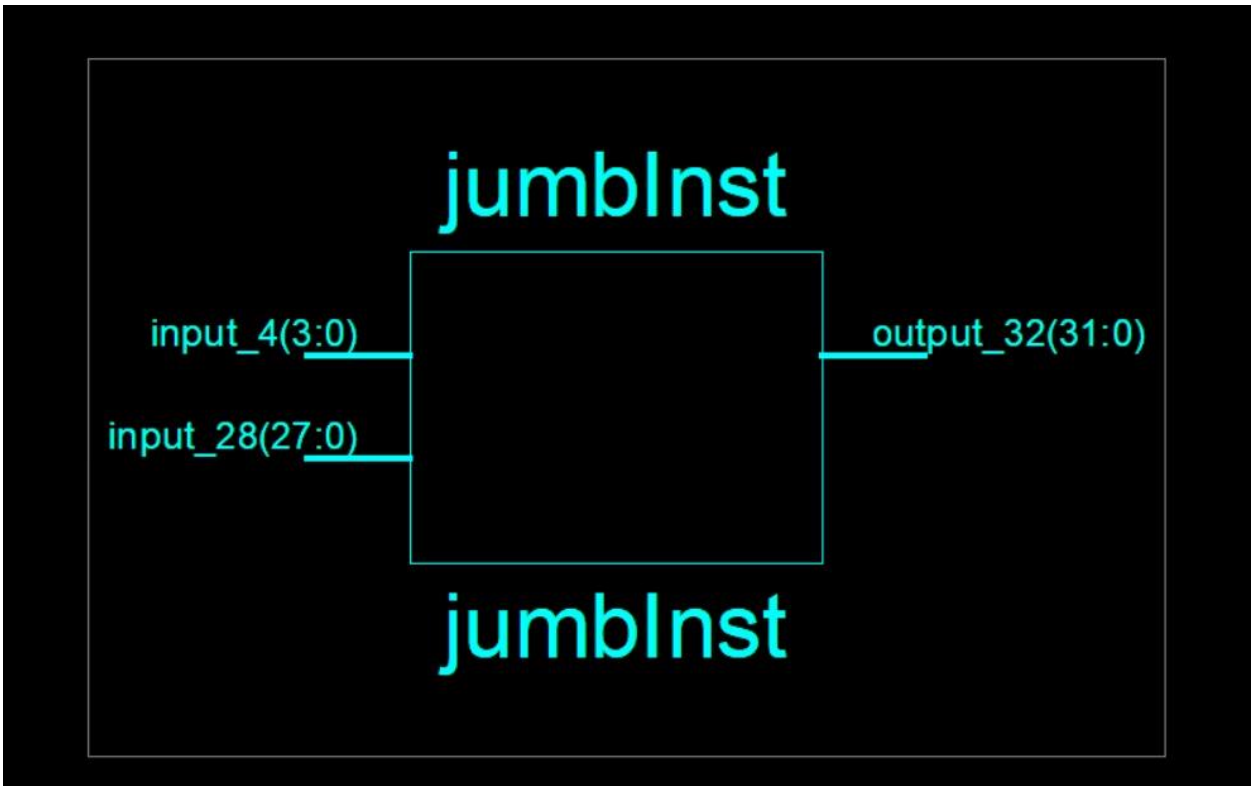
- **Shift Operation:** The Shifter component takes the input data signal (`a`) and performs a shift operation based on the control signals provided.

- **Left Shift:** If the control signals indicate a left shift operation, the Shifter shifts the bits of the input data to the left, effectively multiplying the value by a power of 2.

- **Right Shift:** If the control signals indicate a right shift operation, the Shifter shifts the bits of the input data to the right, effectively dividing the value by a power of 2.

- **Arithmetic vs. Logical Shift:** Depending on the type of shift operation (arithmetic or logical), the Shifter may preserve the sign bit (for arithmetic shifts) or shift in zeros (for logical shifts).

Jump instruction:



Purpose:

The Jump Instruction component is a critical element within the Processor architecture, responsible for handling jump instructions in the instruction set. It facilitates altering the program flow by directing the processor to jump to a new memory address based on a specified condition or target address.

Entity:

- **Name:** jumbInst

Ports:

- **input_28 (in):** Input signal containing the 28-bit target address for the jump instruction.

- **input_4 (in):** Input signal containing the 4-bit jump offset or condition for the jump instruction.

- **output_32 (out):** Output signal containing the 32-bit address resulting from the jump operation.

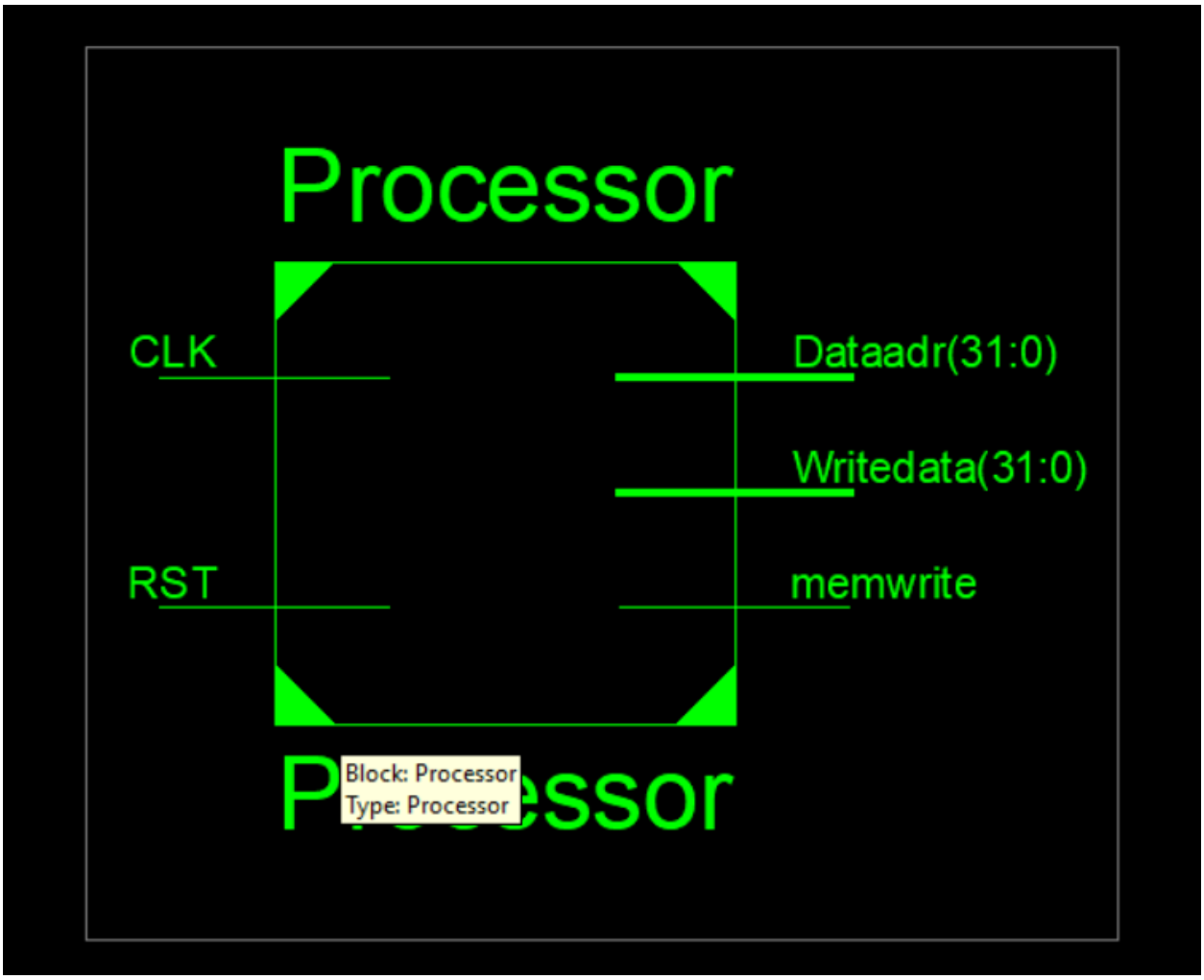
Behavior:

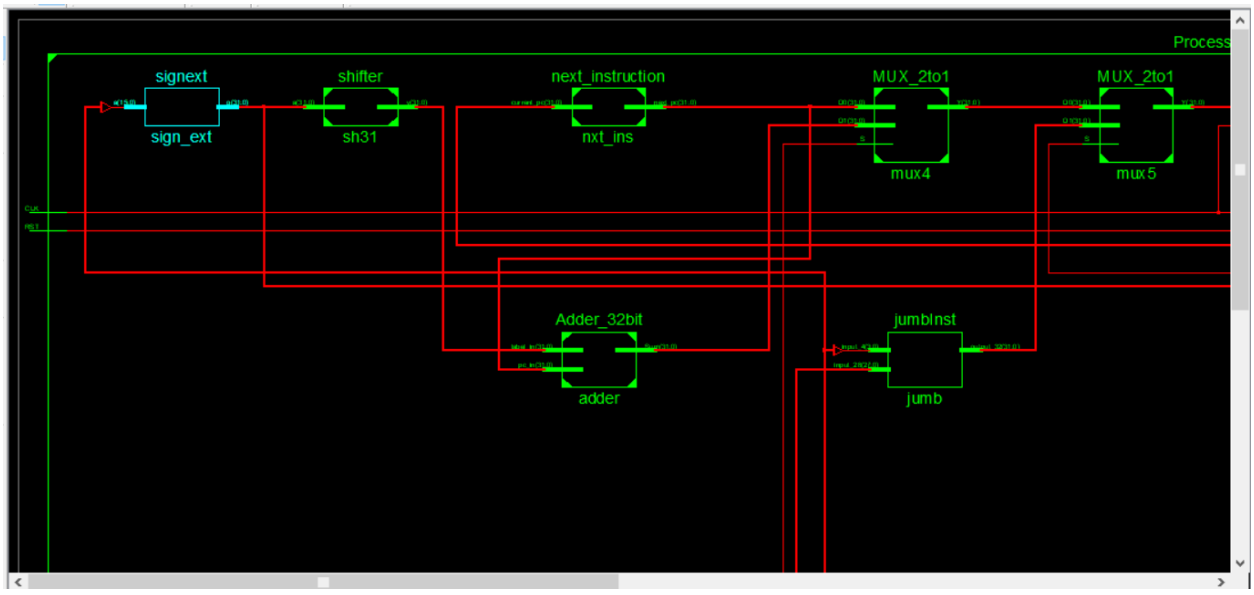
- **Jump Operation:** The Jump Instruction component processes the input signals to determine the target address for the jump operation.

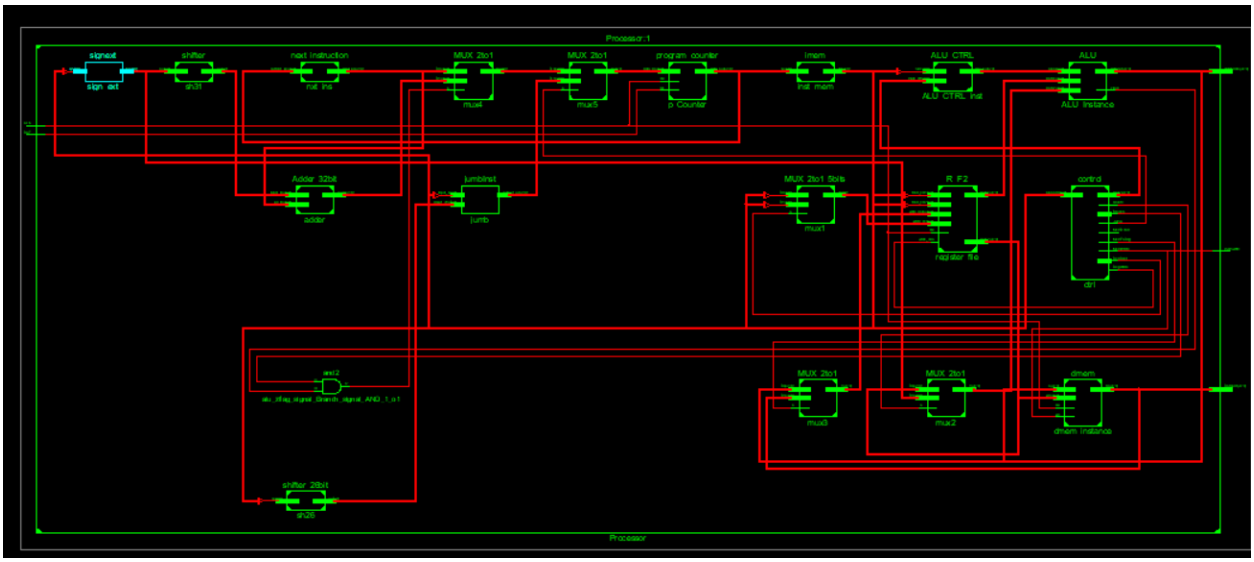
- **Target Address Calculation:** Combines the input signals (`input_28` and `input_4`) to compute the target memory address for the jump operation.

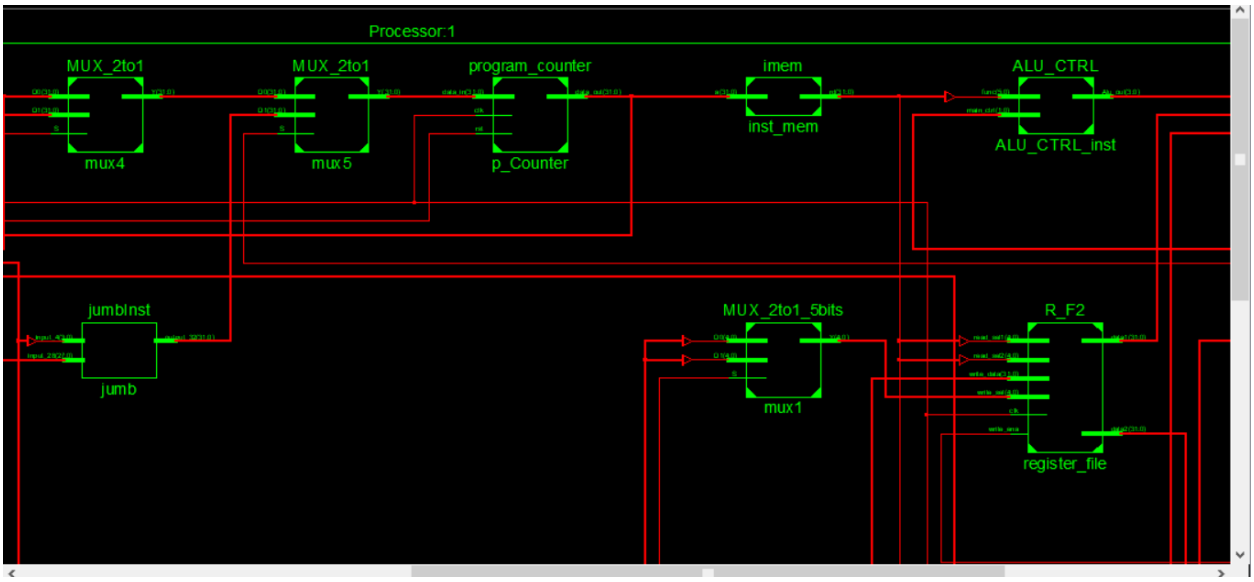
- **Conditional Jump:** If the jump instruction includes a condition, the Jump Instruction component evaluates the condition and determines whether the jump should be executed based on the condition's outcome.

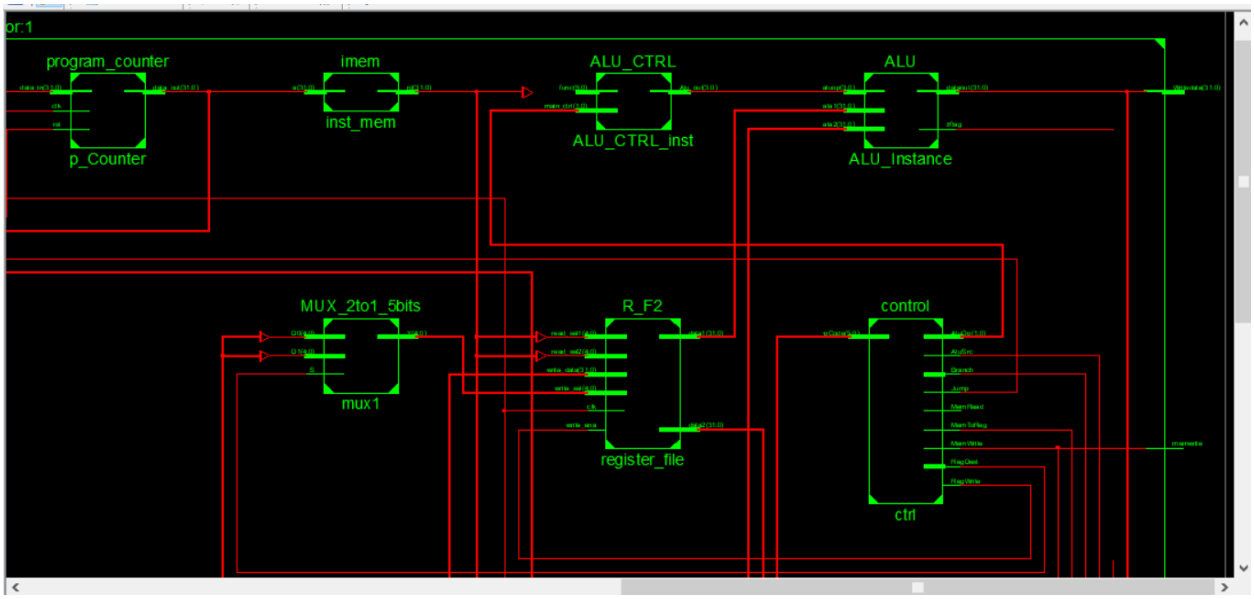
Processor:

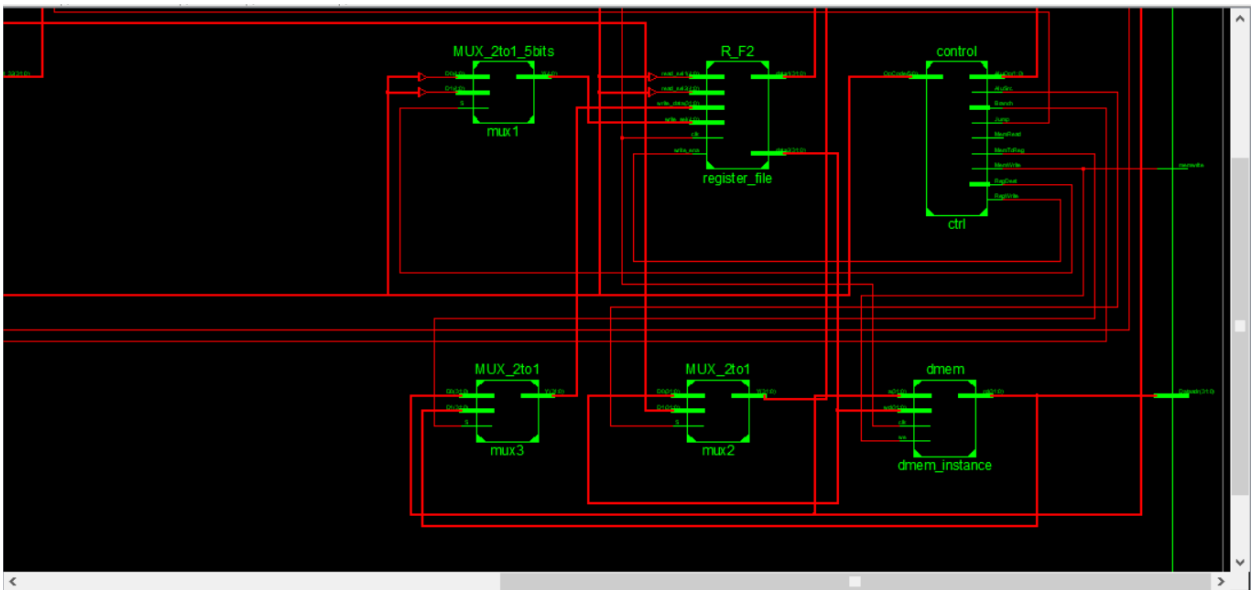

Processor
CLK
RST
Dataadr(31:0)
Writedata(31:0)
memwrite
Block: Processor
Type: Processor

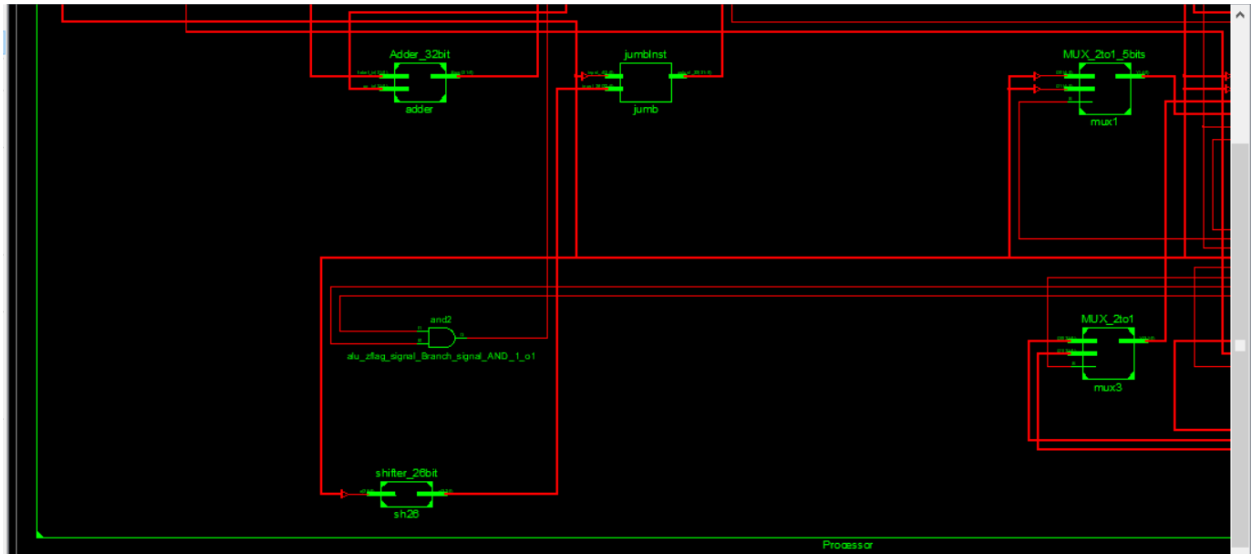

signext
sign_ext
shifter
sh31
next_instruction
nxt_ins
MUX_2to1
mux4
MUX_2to1
mux5
Adder_32bit
adder
jumbInst
jumb


signext
sign_ext
shifter
sh31
next instruction
nxt ins
MUX 2to1
mux4
MUX 2to1
mux5
program counter
p Counter
imem
inst mem
ALU CTRL
ALU CTRL Inst
ALU
ALU Instance
Adder_32bit
adder
jumbInst
jumb
MUX 2to1 5bits
mux1
R F2
register file
control
ctrl
and2
shifter 28bit
sh28
MUX 2to1
mux3
MUX 2to1
mux2
dmem
dmem instance


Processor:1
MUX_2to1
mux4
MUX_2to1
mux5
program_counter
p_Counter
imem
inst_mem
ALU_CTRL
ALU_CTRL_inst
jumbInst
jumb
MUX_2to1_5bits
mux1
R_F2
register_file


or:1
program_counter
p_Counter
imem
inst_mem
ALU_CTRL
ALU_CTRL_inst
ALU
ALU_Instance
MUX_2to1_5bits
mux1
R_F2
register_file
control
ctrl





Purpose:

The Processor architecture represents the internal structure and behavior of a processor in a digital system. It integrates various components such as ALU, register file, control unit, data memory, instruction memory, and multiplexers to execute instructions.

Internal Components:

1. **ALU (Arithmetic Logic Unit):** Performs arithmetic and logical operations based on control signals and input data.

2. **Register File (R_F2):** Stores register values and supports read and write operations.

3. **Control Unit:** Decodes instruction opcodes to generate control signals for other components.

4. **Data Memory (dmem):** Stores data accessed during execution.

5. **Instruction Memory (imem):** Stores program instructions.

6. **Program Counter (program_counter):** Maintains the address of the next instruction to be fetched.

7. **Shifter:** Performs shift operations on input data.

8. **Sign Extension Unit (signext):** Extends the sign of a 16-bit immediate value to 32 bits.

9. **32-Bit Adder (Adder_32bit):** Performs addition operation for branch and jump instructions.

10. **Multiplexers (MUX_2to1, MUX_2to1_5bits):** Selects between multiple input signals based on control signals.

Behavior:

- **Clock and Reset Handling:** The architecture is synchronous, responding to changes in the clock signal (`CLK`) and reset signal (`RST`).

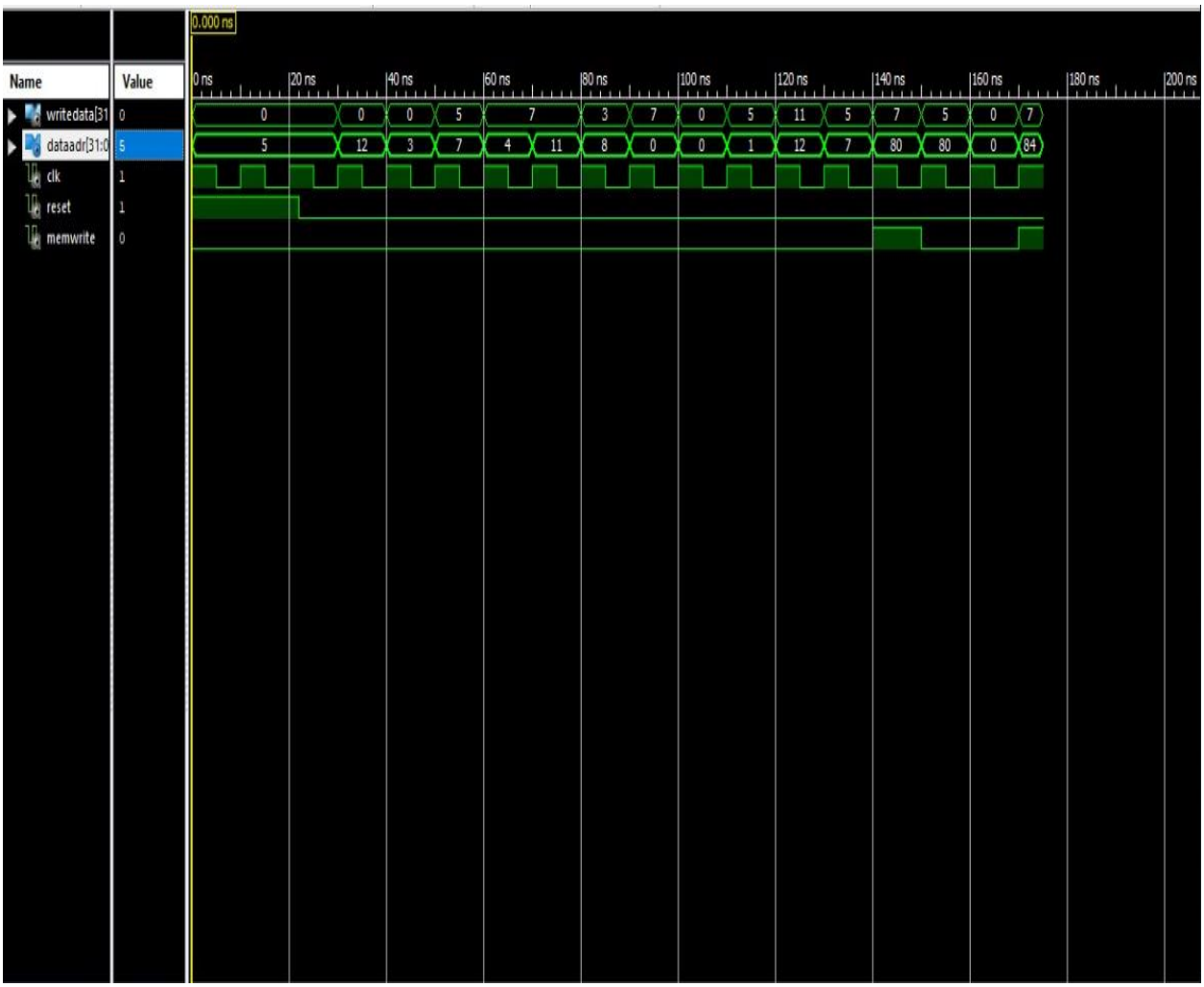
- **Instruction Fetch and Decode:** The program counter increments to fetch the next instruction from the instruction memory. The instruction is decoded by the control unit to generate control signals.

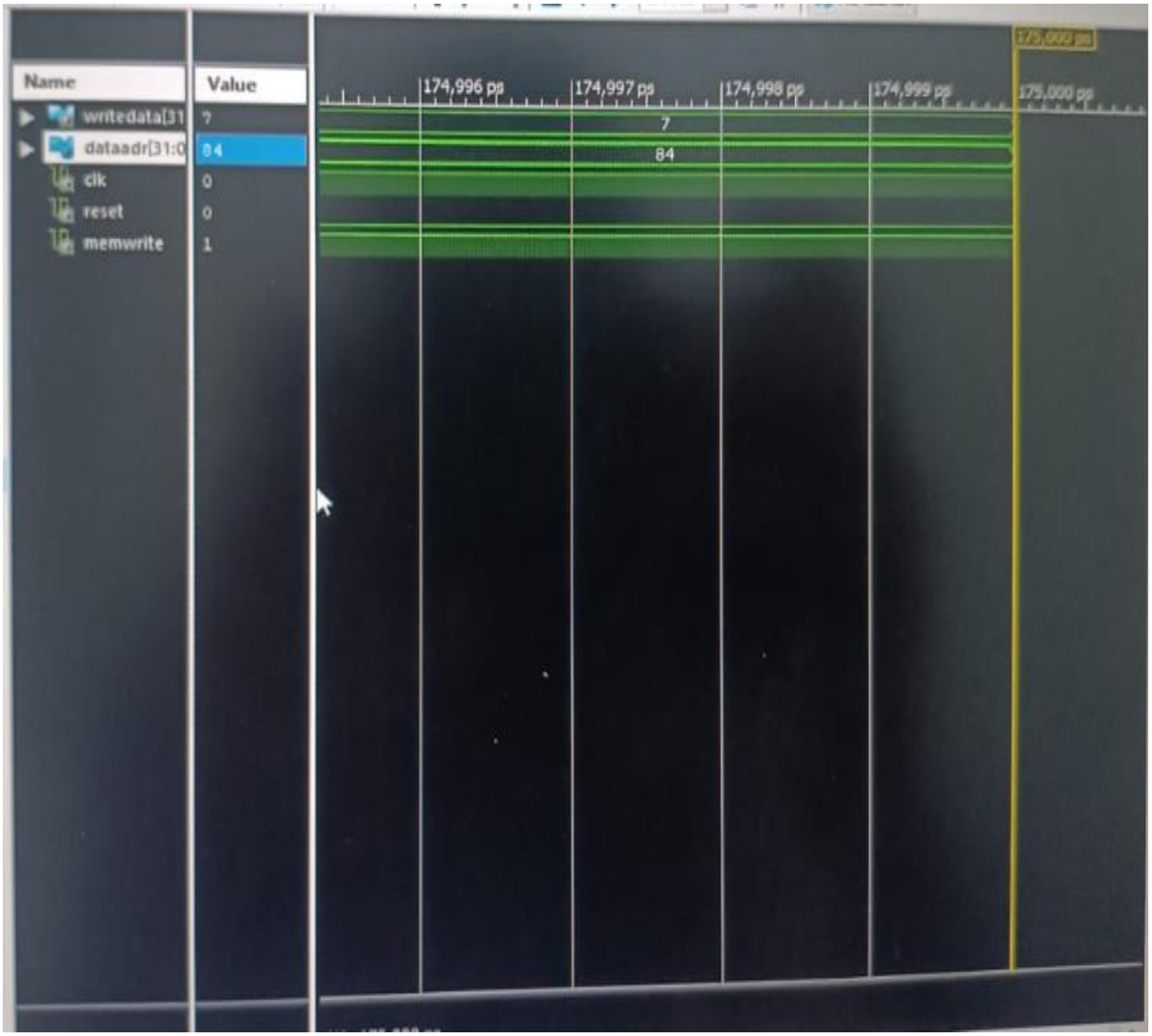
- **Instruction Execution:** Based on the decoded control signals, data is fetched from registers or memory, processed by the ALU, and stored back in registers or memory.

- **Branch and Jump Handling:** Branch and jump instructions are executed by calculating the target address using the ALU and selecting the appropriate next program counter value.

- **Data Path Control:** Multiplexers control the flow of data within the processor, selecting between different sources based on control signals.

TESTING :


0.000 ns
Name
Value
0 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns
160 ns
180 ns
200 ns
writedata[31
0
0 0 0 5 7 3 7 0 5 11 5 7 5 0 7
dataadr[31:0
5
5 12 3 7 4 11 8 0 0 1 12 7 80 80 0 84
clk
1
reset
1
memwrite
0


Name
Value
writedata[31
7
dataadr[31:0
84
clk
0
reset
0
memwrite
1
174,996 ps
174,997 ps
174,998 ps
174,999 ps
175,000 ps
7
84