Ain Shams University
Faculty of Engineering
Discipline Programs

Computer Organization and Architecture Major Task Report

Computer Engineering and Software Systems (CESS)

Submitted to:

Dr. Kareem Emara

Eng. Omar Samy

Submitted by:

Habiba Yasser 20P3072

Nadine Hisham 20P9880

Jumana Yasser 20P8421

Salma Nasreldin 20P7105

Hamsa Ahmad 20P1874

PHASE 1

Table of Contents

1.0 Register

1.1 Register Files

A standard register module, a 5-32 decoder, and a 32x1 multiplexer had to be developed in order to implement the MIPS Register File; these modules were to be used as parts of the main "RegisterFile" module. First, in a module titled "Reg" we implemented a standard register. The standard implementation of a register that accepts the following inputs is the "Reg" register. The register's data i, rst, clk, load, and inc. There is only one output from it. If the rst is set to "1," the output is zeroed out. The output is instead increased by 1 if the inc is set to 1, but the input value to the register is loaded and transferred to the output if the load is set to "1" when the rst is equal to "0" and a clock pulse is received.

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4  use ieee.std_logic_signed.all;
5  use IEEE.NUMERIC_STD.ALL;
6
7
8  entity reg is
9      Port ( i : in  STD_LOGIC_VECTOR (31 downto 0);
10            rst : in  STD_LOGIC;
11            clk : in  STD_LOGIC;
12            load : in  STD_LOGIC;
13            inc : in  STD_LOGIC;
14            o : out  STD_LOGIC_VECTOR (31 downto 0));
15 end reg;
16
17 architecture Behavioral of reg is
18 signal tmp: std_logic_vector (31 downto 0) :=x"00000000";
19 begin
20 process (clk, rst)
21 begin
22 if (rst = '1') then
23 tmp <= (others => '0');
24    elsif (Falling_edge(clk) and load = '1') then
25    tmp <= i;
26       elsif (falling_edge(clk) and inc = '1') then
27       tmp <= std_logic_vector(signed(tmp)+1);
28 end if;
29 end process;
30 o <= tmp;
31
32 end Behavioral;
33
```

The 5-32 decoder "dec5x32" was then implemented. Its entity has one output port called "O," two input ports called "I"(Input) and "E"(Enable). The decoder will take a 5-bit input and convert it to a 32-bit output, which will be used to write data to a register. The 5-bit input will be the register's address, and the 32-output will be connected to the 32 registers of the MIPS Register File as enable lines; whichever one is activated during the writing process will select the equivalent register and the data will be written to it.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity dec5x32 is
5      Port ( I : in  STD_LOGIC_VECTOR (4 DOWNTO 0);
6             E : in  STD_LOGIC;
7  O : out STD_LOGIC_VECTOR (31 DOWNTO 0));
8  end dec5x32;
9  architecture Behavioral of dec5x32 is
10 begin
11 O <= (OTHERS => 'Z') WHEN E= '0' ELSE
12        "00000000000000000000000000000001" WHEN I="00000" ELSE
13        "00000000000000000000000000000010" WHEN I="00001" ELSE
14         "00000000000000000000000000000100" WHEN I="00010" ELSE
15         "00000000000000000000000000001000" WHEN I="00011" ELSE
16         "00000000000000000000000000010000" WHEN I="00100" ELSE
17         "00000000000000000000000000100000" WHEN I="00101" ELSE
18         "00000000000000000000000001000000" WHEN I="00110" ELSE
19         "00000000000000000000000010000000" WHEN I="00111" ELSE
20         "00000000000000000000000100000000" WHEN I="01000" ELSE
21         "00000000000000000000001000000000" WHEN I="01001" ELSE
22         "00000000000000000000010000000000" WHEN I="01010" ELSE
23         "00000000000000000000100000000000" WHEN I="01011" ELSE
24         "00000000000000000001000000000000" WHEN I="01100" ELSE
25         "00000000000000000010000000000000" WHEN I="01101" ELSE
26        "00000000000000000100000000000000" WHEN I="01110" ELSE
27         "00000000000000001000000000000000" WHEN I="01111" ELSE
28        "00000000000000010000000000000000" WHEN I="10000" ELSE
29 "00000000000000100000000000000000" WHEN I="10001" ELSE
30 "00000000000001000000000000000000" WHEN I="10010" ELSE
31 "00000000000010000000000000000000" WHEN I="10011" ELSE
32 "00000000000100000000000000000000" WHEN I="10100" ELSE
33 "00000000001000000000000000000000" WHEN I="10101" ELSE
34 "00000000010000000000000000000000" WHEN I="10110" ELSE
35 "00000000100000000000000000000000" WHEN I="10111" ELSE
36 "00000001000000000000000000000000" WHEN I="11000" ELSE
37 "00000010000000000000000000000000" WHEN I="11001" ELSE
38 "00000100000000000000000000000000" WHEN I="11010" ELSE
39 "00001000000000000000000000000000" WHEN I="11011" ELSE
40 "00010000000000000000000000000000" WHEN I="11100" ELSE
41 "00100000000000000000000000000000" WHEN I="11101" ELSE
42 "01000000000000000000000000000000" WHEN I="11110" ELSE
43 "10000000000000000000000000000000" WHEN I="11111" ELSE
44 (others => 'Z');
45 end Behavioral;
```

Then we implemented a 32x1 multiplexer, which takes 32 inputs which will be the data stored in the 32 registers and produces only one output which will be the data required to be read from the selected register.

```
6
7   entity mux32x1 is
8       Port ( s : in  STD_LOGIC_VECTOR (4 downto 0);
9              i0 : in  STD_LOGIC_VECTOR (31 downto 0);
10             i1 : in  STD_LOGIC_VECTOR (31 downto 0);
11             i2 : in  STD_LOGIC_VECTOR (31 downto 0);
12             i3 : in  STD_LOGIC_VECTOR (31 downto 0);
13             i4 : in  STD_LOGIC_VECTOR (31 downto 0);
14             i5 : in  STD_LOGIC_VECTOR (31 downto 0);
15             i6 : in  STD_LOGIC_VECTOR (31 downto 0);
16             i7 : in  STD_LOGIC_VECTOR (31 downto 0);
17             i8 : in  STD_LOGIC_VECTOR (31 downto 0);
18             i9 : in  STD_LOGIC_VECTOR (31 downto 0);
19             i10 : in  STD_LOGIC_VECTOR (31 downto 0);
20             i11 : in  STD_LOGIC_VECTOR (31 downto 0);
21             i12 : in  STD_LOGIC_VECTOR (31 downto 0);
22             i13 : in  STD_LOGIC_VECTOR (31 downto 0);
23             i14 : in  STD_LOGIC_VECTOR (31 downto 0);
24             i15 : in  STD_LOGIC_VECTOR (31 downto 0);
25             i16 : in  STD_LOGIC_VECTOR (31 downto 0);
26             i17 : in  STD_LOGIC_VECTOR (31 downto 0);
27             i18 : in  STD_LOGIC_VECTOR (31 downto 0);
28             i19 : in  STD_LOGIC_VECTOR (31 downto 0);
29             i20 : in  STD_LOGIC_VECTOR (31 downto 0);
30             i21 : in  STD_LOGIC_VECTOR (31 downto 0);
31             i22 : in  STD_LOGIC_VECTOR (31 downto 0);
32             i23 : in  STD_LOGIC_VECTOR (31 downto 0);
33             i24 : in  STD_LOGIC_VECTOR (31 downto 0);
34             i25 : in  STD_LOGIC_VECTOR (31 downto 0);
35             i26 : in  STD_LOGIC_VECTOR (31 downto 0);
36             i27 : in  STD_LOGIC_VECTOR (31 downto 0);
37             i28 : in  STD_LOGIC_VECTOR (31 downto 0);
38             i29 : in  STD_LOGIC_VECTOR (31 downto 0);
39             i30 : in  STD_LOGIC_VECTOR (31 downto 0);
40             i31 : in  STD_LOGIC_VECTOR (31 downto 0);
41             o : out  STD_LOGIC_VECTOR (31 downto 0));
42  end mux32x1;
43
```

```
44  architecture Behavioral of mux32x1 is
45
46  begin
47   o <=           i0  when S = "00000" else
48                  i1  when S = "00001" else
49                  i2  when S = "00010" else
50                  i3  when S = "00011" else
51                  i4  when S = "00100" else
52                  i5  when S = "00101" else
53                  i6  when S = "00110" else
54                  i7  when S = "00111" else
55                  i8  when S = "01000" else
56                  i9  when S = "01001" else
57                  i10 when S = "01010" else
58                  i11 when S = "01011" else
59                  i12 when S = "01100" else
60                  i13 when S = "01101" else
61                  i14 when S = "01110" else
62                  i15 when S = "01111" else
63                  i16 when S = "10000" else
64                  i17 when S = "10001" else
65                  i18 when S = "10010" else
66                  i19 when S = "10011" else
67                  i20 when S = "10100" else
68                  i21 when S = "10101" else
69                  i22 when S = "10110" else
70                  i23 when S = "10111" else
71                  i24 when S = "11000" else
72                  i25 when S = "11001" else
73                  i26 when S = "11010" else
74                  i27 when S = "11011" else
75                  i28 when S = "11100" else
76                  i29 when S = "11101" else
77                  i30 when S = "11110" else
78                  i31 when S = "11111" else
79                  (others => 'Z');
80
81
82  end Behavioral;
```

To be used in the implementation of the primary module, "RegisterFile" we put all the earlier modules into a package called "mypackage".

```
 3  library IEEE;
 4  use IEEE.STD_LOGIC_1164.all;
 5
 6  package mypackage is
 7
 8  component reg is
 9   Port ( i : in  STD_LOGIC_VECTOR (31 downto 0);
10          rst : in  STD_LOGIC;
11          clk : in  STD_LOGIC;
12          load : in  STD_LOGIC;
13          inc : in  STD_LOGIC;
14          o : out  STD_LOGIC_VECTOR (31 downto 0));
15  end component;
16
17  component dec5x32 is
18   Port ( I : in  STD_LOGIC_VECTOR (4 DOWNTO 0);
19          E : in  STD_LOGIC;
20  O : out STD_LOGIC_VECTOR (31 DOWNTO 0));
21  end component;
22
23  component mux32x1 is
24  Port ( s : in  STD_LOGIC_VECTOR (4 downto 0);
25          i0 : in  STD_LOGIC_VECTOR (31 downto 0);
26          i1 : in  STD_LOGIC_VECTOR (31 downto 0);
27          i2 : in  STD_LOGIC_VECTOR (31 downto 0);
28          i3 : in  STD_LOGIC_VECTOR (31 downto 0);
29          i4 : in  STD_LOGIC_VECTOR (31 downto 0);
30          i5 : in  STD_LOGIC_VECTOR (31 downto 0);
31          i6 : in  STD_LOGIC_VECTOR (31 downto 0);
32          i7 : in  STD_LOGIC_VECTOR (31 downto 0);
33          i8 : in  STD_LOGIC_VECTOR (31 downto 0);
34          i9 : in  STD_LOGIC_VECTOR (31 downto 0);
35          i10 : in  STD_LOGIC_VECTOR (31 downto 0);
36          i11 : in  STD_LOGIC_VECTOR (31 downto 0);
```

```
37          i12 : in  STD_LOGIC_VECTOR (31 downto 0);
38          i13 : in  STD_LOGIC_VECTOR (31 downto 0);
39          i14 : in  STD_LOGIC_VECTOR (31 downto 0);
40          i15 : in  STD_LOGIC_VECTOR (31 downto 0);
41          i16 : in  STD_LOGIC_VECTOR (31 downto 0);
42          i17 : in  STD_LOGIC_VECTOR (31 downto 0);
43          i18 : in  STD_LOGIC_VECTOR (31 downto 0);
44          i19 : in  STD_LOGIC_VECTOR (31 downto 0);
45          i20 : in  STD_LOGIC_VECTOR (31 downto 0);
46          i21 : in  STD_LOGIC_VECTOR (31 downto 0);
47          i22 : in  STD_LOGIC_VECTOR (31 downto 0);
48          i23 : in  STD_LOGIC_VECTOR (31 downto 0);
49          i24 : in  STD_LOGIC_VECTOR (31 downto 0);
50          i25 : in  STD_LOGIC_VECTOR (31 downto 0);
51          i26 : in  STD_LOGIC_VECTOR (31 downto 0);
52          i27 : in  STD_LOGIC_VECTOR (31 downto 0);
53          i28 : in  STD_LOGIC_VECTOR (31 downto 0);
54          i29 : in  STD_LOGIC_VECTOR (31 downto 0);
55          i30 : in  STD_LOGIC_VECTOR (31 downto 0);
56          i31 : in  STD_LOGIC_VECTOR (31 downto 0);
57          o : out  STD_LOGIC_VECTOR (31 downto 0));
58  end component;
59
```

Second, we designated a signal "L" of 32-bit length to be used as the load enable of each register, from L(0) to L(31), in order to select the necessary register during the writing process. Additionally, we designated 32 signals, labelled "out0" through "out31," to hold the output of each of the 32 registers so that they could later be used as inputs for the 32x1 multiplexer to select the necessary output during the reading process.

Additionally, we developed one decoder that we called "dec5x32" in the past implementation, mapping write_sel to input, write_ena to enable, and Signal 'L' to decoder output.

Then, using the "Reg" module, we created 32 registers and assigned write data to each of their inputs, with the exception of "Reg0," which had 32 zeros as its input because it would represent $zero. Next, we assigned '0' to the reset, 'clk' to the clk, 'L' signal starting from L(0) to L(31) respectively to the load, '0' to the increment, and from out0 to out31 respectively to the outputs of the registers.

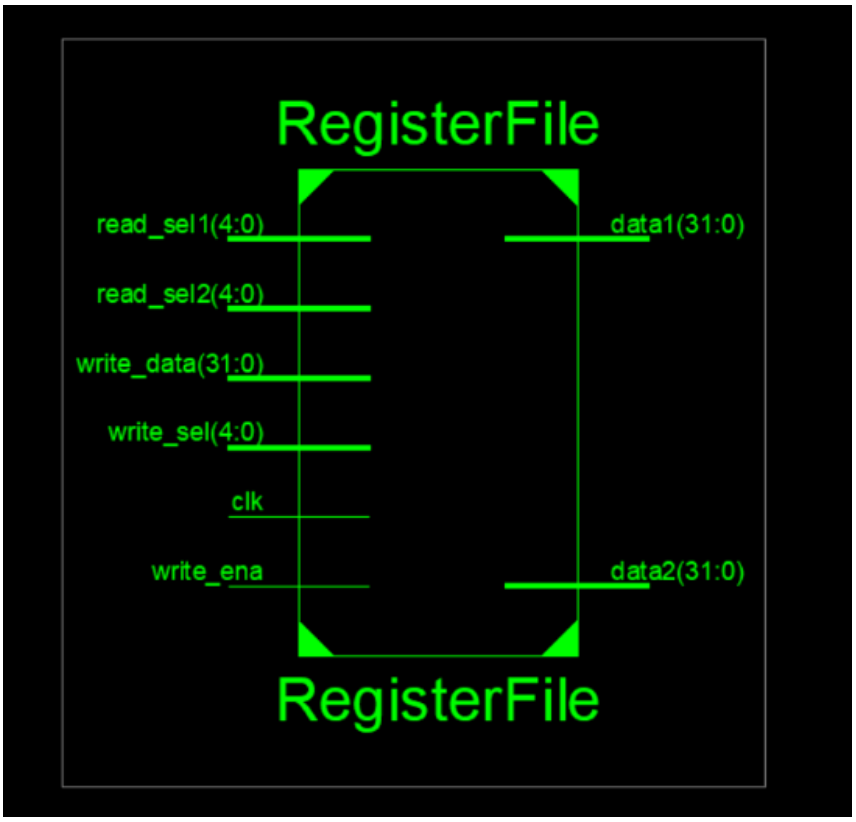
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_signed.all;
use work.mypackage.all;


entity RegisterFile 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 RegisterFile;

architecture Behavioral of RegisterFile is
signal L, out0, out1, out2, out3, out4, out5, out6,
out7, out8, out9, out10, out11, out12, out13, out14, out15,
out16, out17, out18, out19, out20, out21, out22, out23, out24,
out25, out26, out27, out28, out29, out30, out31: std_logic_vector (31 downto 0);
begin

decoo: dec5x32 port map (write_sel, write_ena, L);
reg0:  reg  port map (x"00000000", '0', clk, L(0), '0', out0);
reg1:  reg  port map (write_data, '0', clk, L(1), '0', out1);
reg2:  reg  port map (write_data, '0', clk, L(2), '0', out2);
reg3:  reg  port map (write_data, '0', clk, L(3), '0', out3);
reg4:  reg  port map (write_data, '0', clk, L(4), '0', out4);
reg5:  reg  port map (write_data, '0', clk, L(5), '0', out5);
reg6:  reg  port map (write_data, '0', clk, L(6), '0', out6);
reg7:  reg  port map (write_data, '0', clk, L(7), '0', out7);
reg8:  reg  port map (write_data, '0', clk, L(8), '0', out8);
```

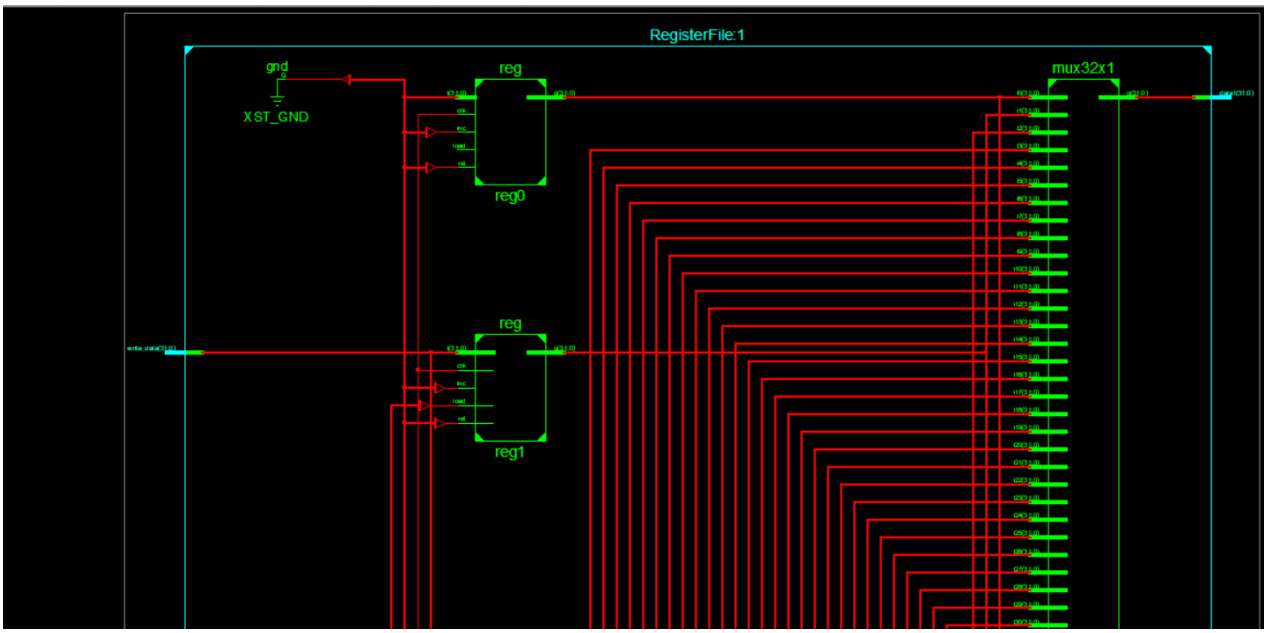
```vhdl
reg9:  reg  port map (write_data, '0', clk, L(9), '0', out9);
reg10:  reg  port map (write_data, '0', clk, L(10), '0', out10);
reg11:  reg  port map (write_data, '0', clk, L(11), '0', out11);
reg12:  reg  port map (write_data, '0', clk, L(12), '0', out12);
reg13:  reg  port map (write_data, '0', clk, L(13), '0', out13);
reg14:  reg  port map (write_data, '0', clk, L(14), '0', out14);
reg15:  reg  port map (write_data, '0', clk, L(15), '0', out15);
reg16:  reg  port map (write_data, '0', clk, L(16), '0', out16);
reg17:  reg  port map (write_data, '0', clk, L(17), '0', out17);
reg18:  reg  port map (write_data, '0', clk, L(18), '0', out18);
reg19:  reg  port map (write_data, '0', clk, L(19), '0', out19);
reg20:  reg  port map (write_data, '0', clk, L(20), '0', out20);
reg21:  reg  port map (write_data, '0', clk, L(21), '0', out21);
reg22:  reg  port map (write_data, '0', clk, L(22), '0', out22);
reg23:  reg  port map (write_data, '0', clk, L(23), '0', out23);
reg24:  reg  port map (write_data, '0', clk, L(24), '0', out24);
reg25:  reg  port map (write_data, '0', clk, L(25), '0', out25);
reg26:  reg  port map (write_data, '0', clk, L(26), '0', out26);
reg27:  reg  port map (write_data, '0', clk, L(27), '0', out27);
reg28:  reg  port map (write_data, '0', clk, L(28), '0', out28);
reg29:  reg  port map (write_data, '0', clk, L(29), '0', out29);
reg30:  reg  port map (write_data, '0', clk, L(30), '0', out30);
reg31:  reg  port map (write_data, '0', clk, L(31), '0', out31);
```

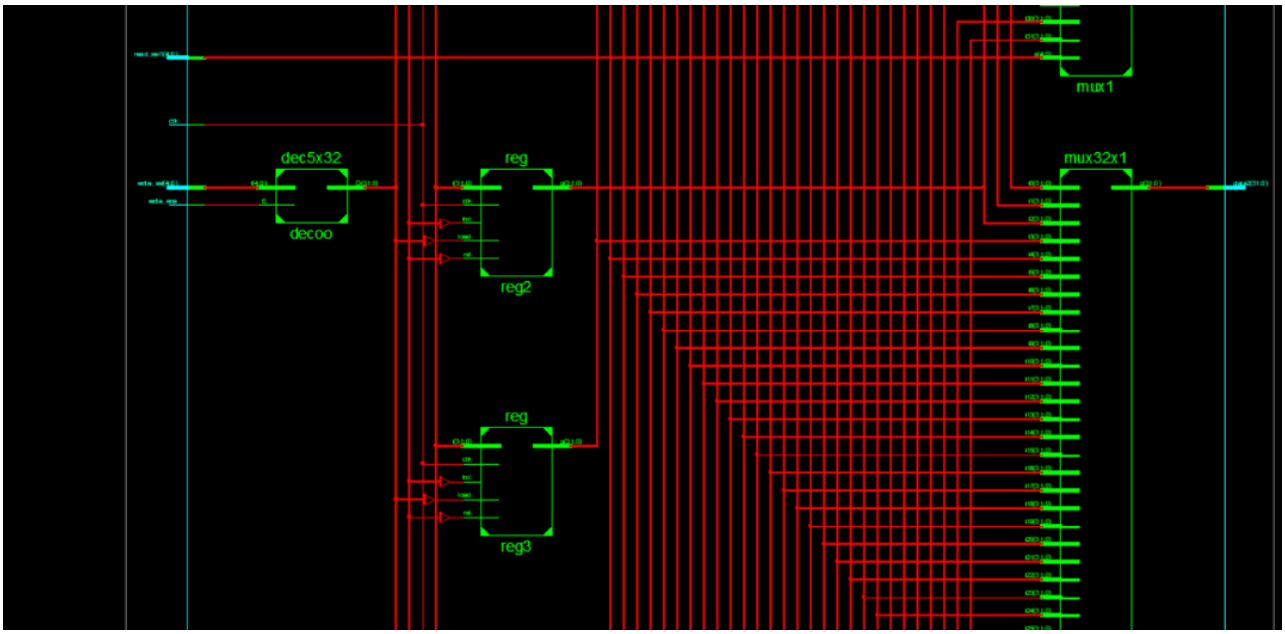
At the end, we created two 32x1 multiplexers, each taking the out0 till the out31 signals as the 32 inputs of the multiplexer, but for the first multiplexer, theselector will be read_sel1 and the output will be data1, and for the second multiplexer, the selector will be read_sel2 and the output will be data2. This will be used in the reading process of data 1 and data 2.

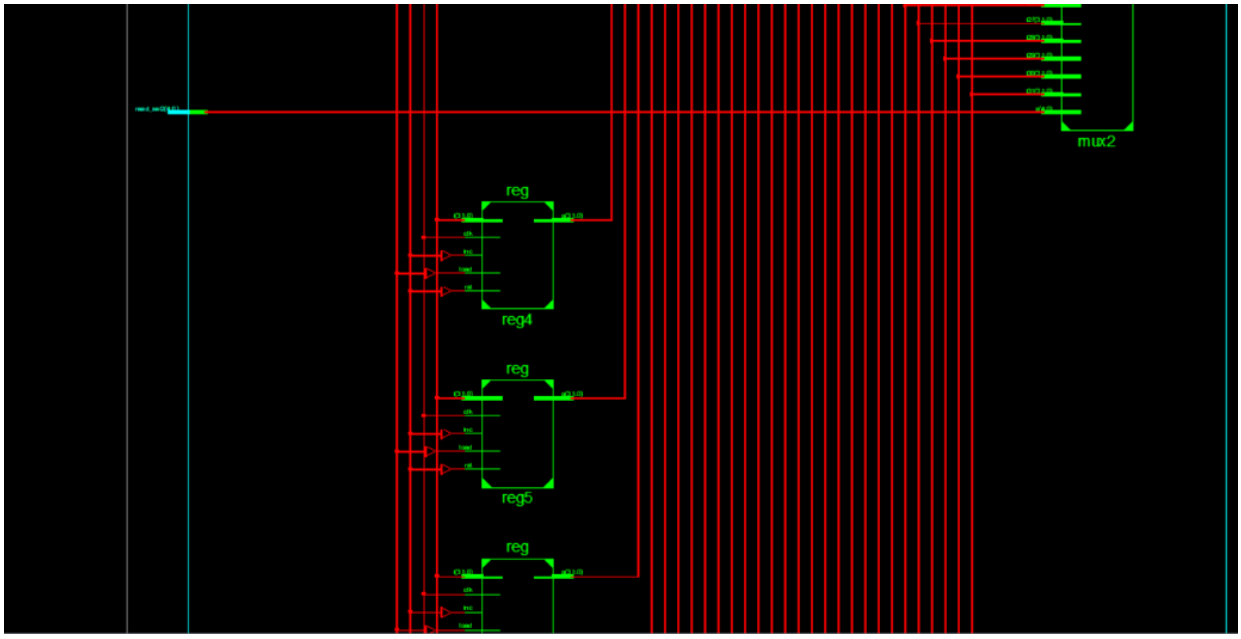
```
60  mux1: mux32x1 port map (read_sel1, out0, out1, out2, out3, out4, out5, out6,
61                          out7, out8, out9, out10, out11, out12, out13, out14, out15,
62                          out16, out17, out18, out19, out20, out21, out22, out23, out24,
63                          out25, out26, out27, out28, out29, out30, out31, data1);
64
65  mux2: mux32x1 port map (read_sel2, out0, out1, out2, out3, out4, out5, out6,
66                          out7, out8, out9, out10, out11, out12, out13, out14, out15,
67                          out16, out17, out18, out19, out20, out21, out22, out23, out24,
68                          out25, out26, out27, out28, out29, out30, out31, data2);
69
70  end Behavioral;
71
```

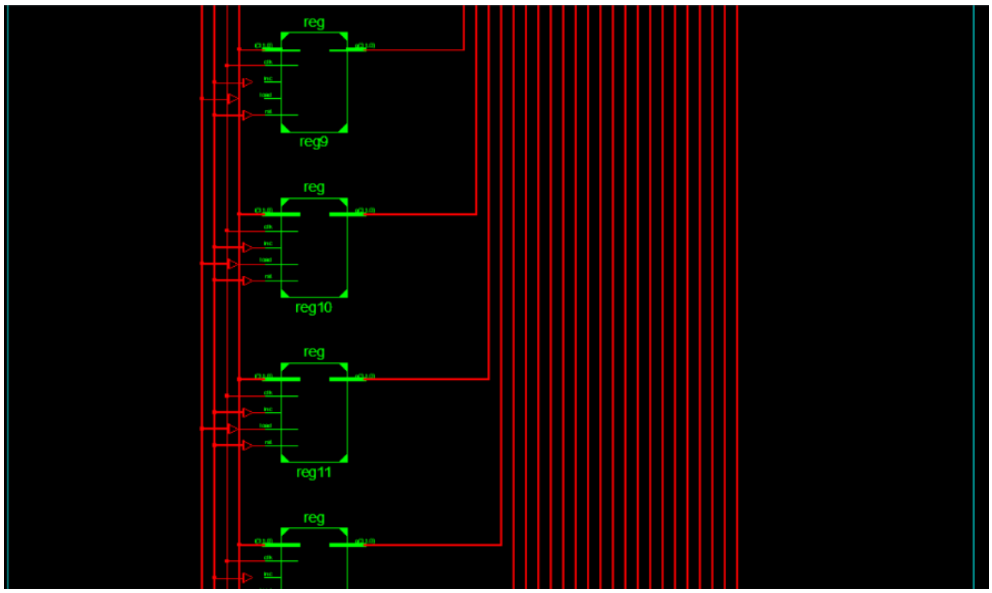
1.2 Register RTL Schematic

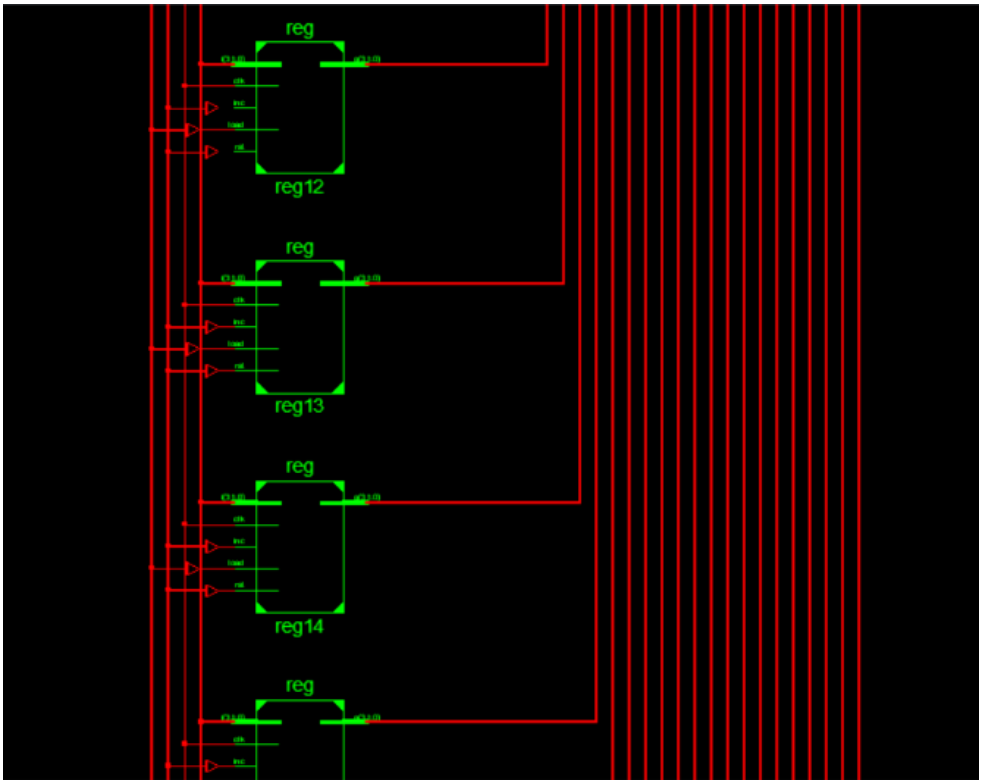


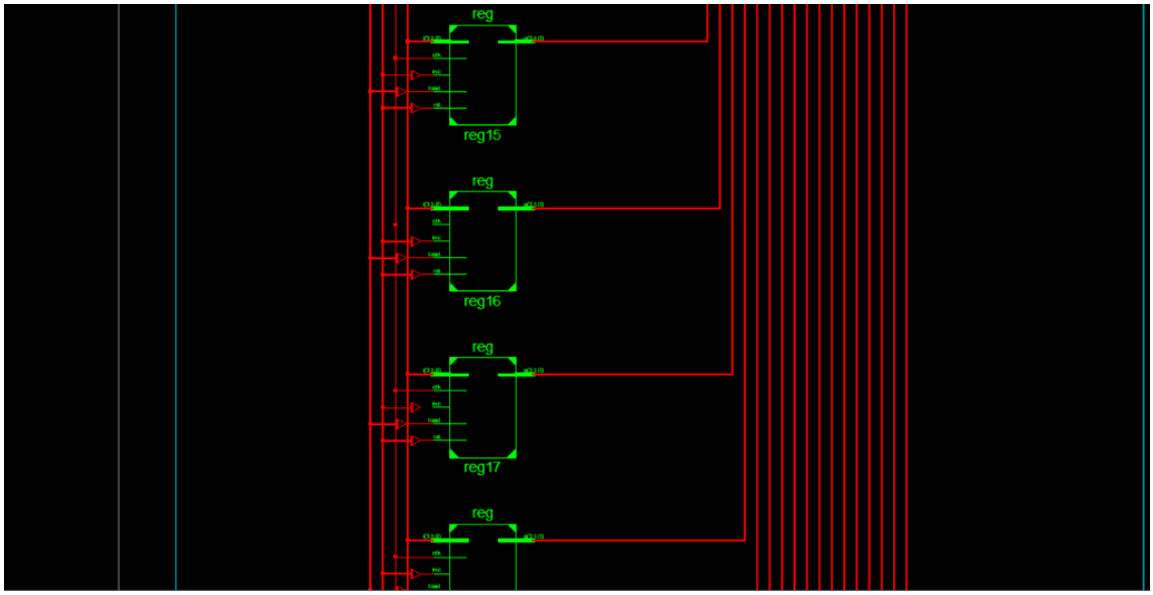

RegisterFile:1
gnd
XST_GND
reg
reg0
reg
reg1
mux32x1

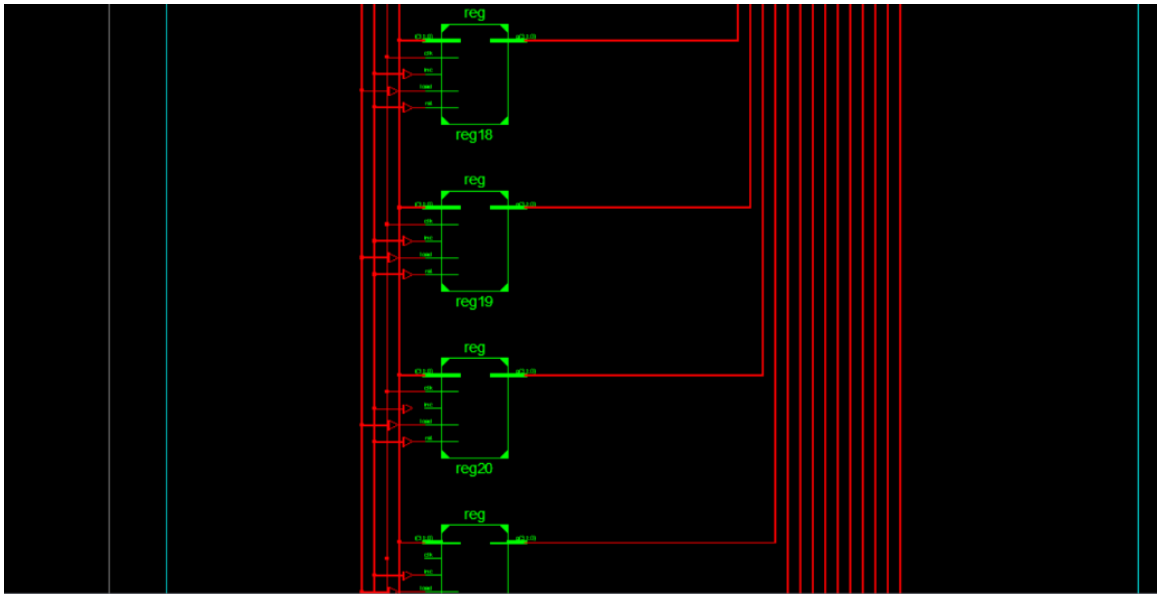

mux1
dec5x32
decoo
reg
reg2
reg
reg3
mux32x1

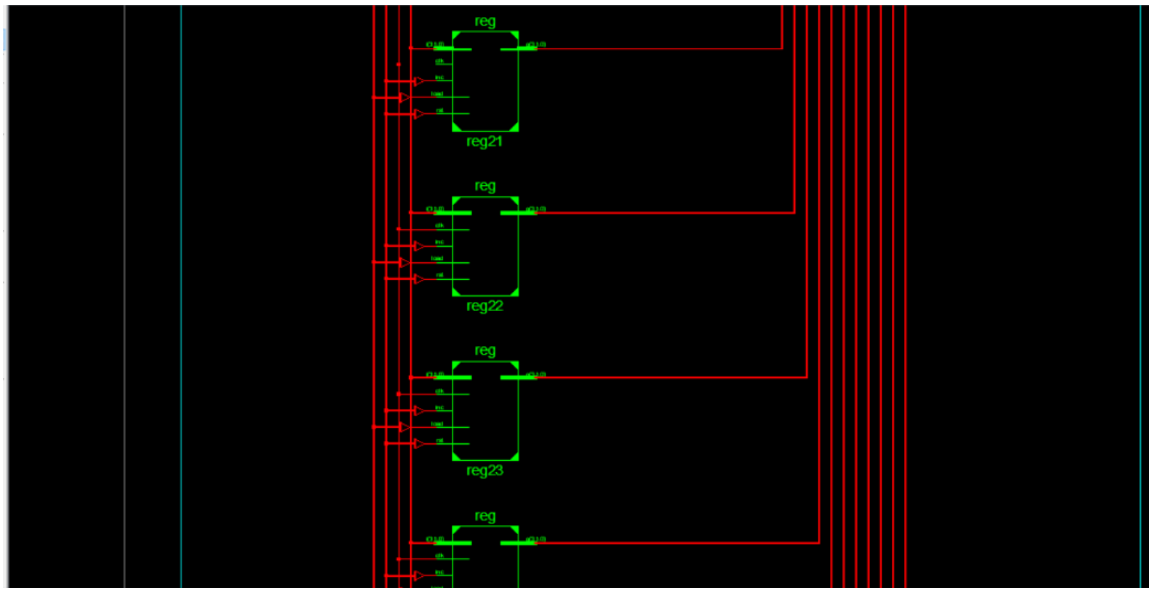

mux2
reg
reg4
reg
reg5
reg
reg
reg6
reg
reg7
reg
reg8
reg

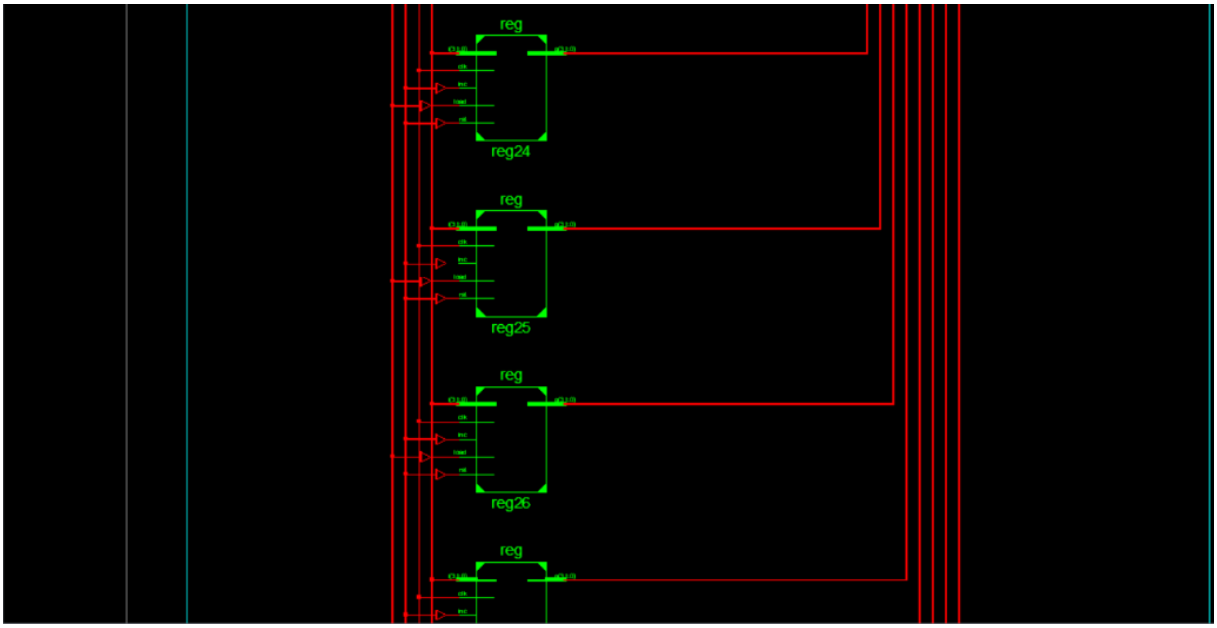

reg
reg9
reg
reg10
reg
reg11
reg

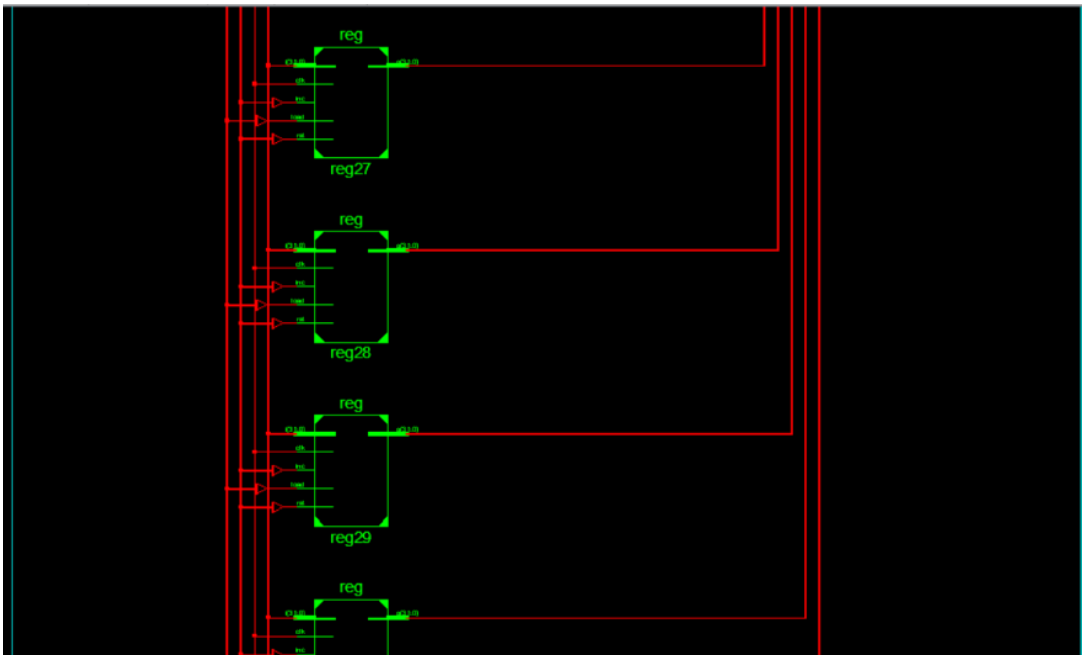

reg
reg12
reg
reg13
reg
reg14
reg

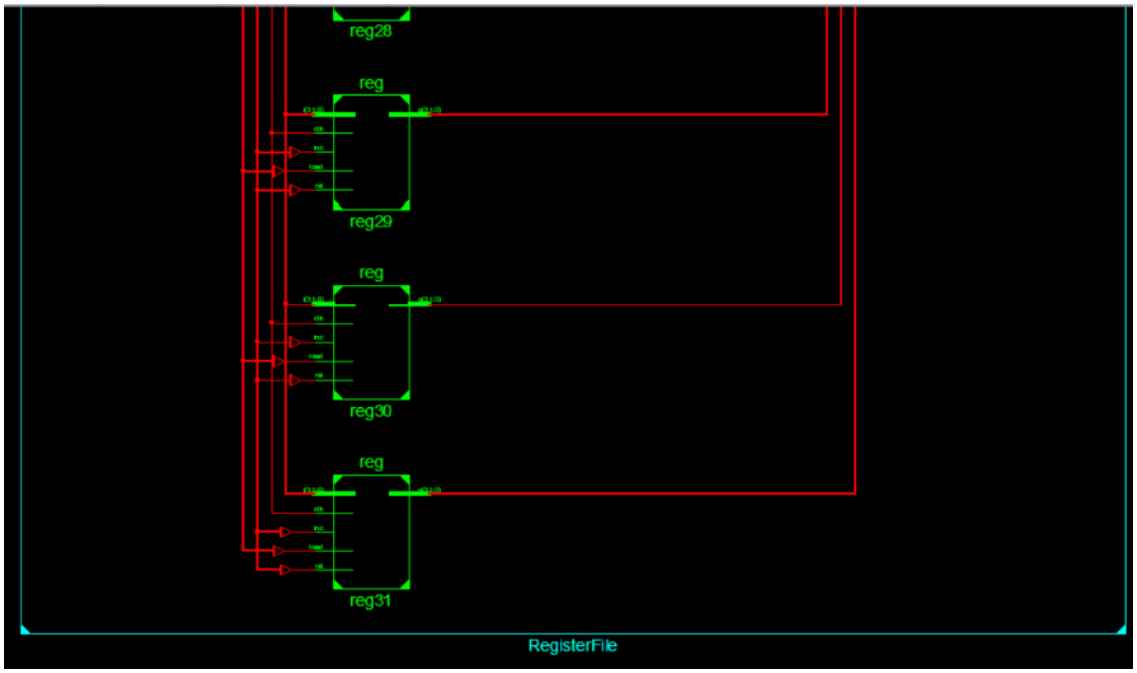

reg
reg15
reg
reg16
reg
reg17
reg


reg
reg18
reg
reg19
reg
reg20
reg


reg
reg21
reg
reg22
reg
reg23
reg


reg
reg24
reg
reg25
reg
reg26
reg


reg
reg27
reg
reg28
reg
reg29
reg


reg28
reg
reg29
reg
reg30
reg
reg31
RegisterFile

1.3 Register Test Bench Code

```
1
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.ALL;
4  USE ieee.std_logic_unsigned.all;
5  USE ieee.numeric_std.ALL;
6
7  ENTITY RegisterFileTest IS
8  END RegisterFileTest;
9
10 ARCHITECTURE behavior OF RegisterFileTest IS
11
12     COMPONENT RegisterFile
13     PORT(
14          read_sel1 : IN  std_logic_vector(4 downto 0);
15          read_sel2 : IN  std_logic_vector(4 downto 0);
16          write_sel : IN  std_logic_vector(4 downto 0);
17          write_ena : IN  std_logic;
18          clk : IN  std_logic;
19          write_data : IN  std_logic_vector(31 downto 0);
20          data1 : OUT  std_logic_vector(31 downto 0);
21          data2 : OUT  std_logic_vector(31 downto 0)
22         );
23     END COMPONENT;
24
25
26    --Inputs
27    signal read_sel1 : std_logic_vector(4 downto 0) := (others => '0');
28    signal read_sel2 : std_logic_vector(4 downto 0) := (others => '0');
29    signal write_sel : std_logic_vector(4 downto 0) := (others => '0');
30    signal write_ena : std_logic := '0';
31    signal clk : std_logic := '0';
32    signal write_data : std_logic_vector(31 downto 0) := (others => '0');
33
```

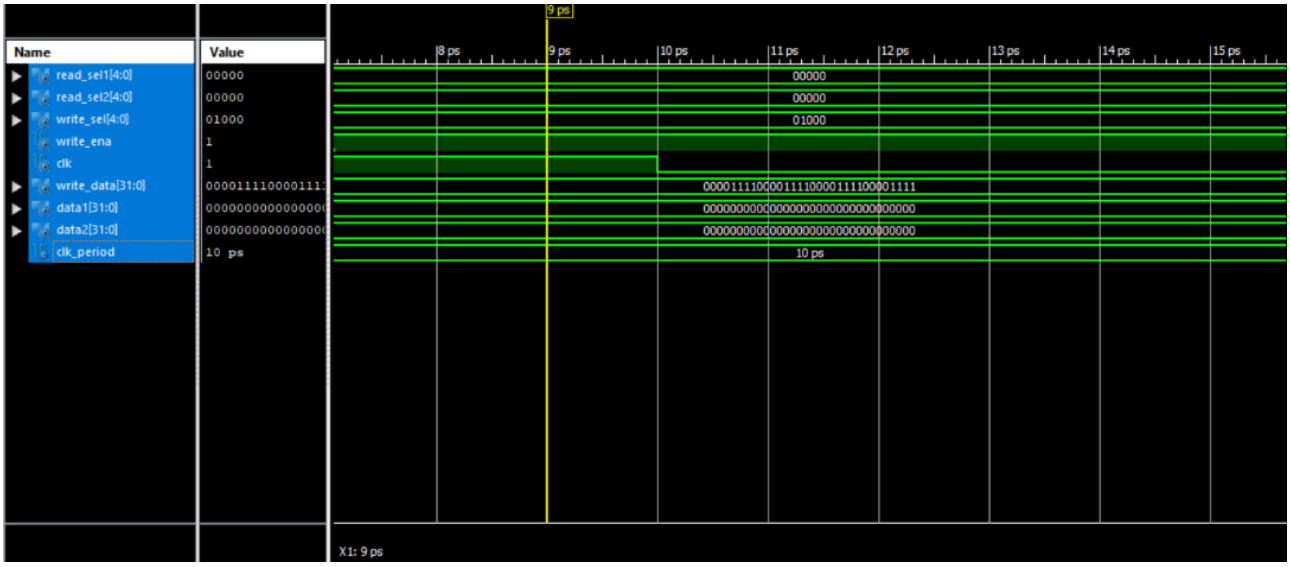
```
34     --Outputs
35     signal data1 : std_logic_vector(31 downto 0);
36     signal data2 : std_logic_vector(31 downto 0);
37
38     -- Clock period definitions
39    constant clk_period : time := 10 ps;
40
41  BEGIN
42
43     -- Instantiate the Unit Under Test (UUT)
44     uut: RegisterFile PORT MAP (
45            read_sel1 => read_sel1,
46            read_sel2 => read_sel2,
47            write_sel => write_sel,
48            write_ena => write_ena,
49            clk => clk,
50            write_data => write_data,
51            data1 => data1,
52            data2 => data2
53          );
54
55     -- Clock process definitions
56    clk_process: process
57    begin
58    clk <= '0';
59    wait for clk_period/2;
60    clk <= '1';
61    wait for clk_period/2;
62    end process;
63
64     -- Stimulus process
65     stim_proc: process
66     begin
67           wait for clk_period - 3ps;
```

```
64    -- Stimulus process
65    stim_proc: process
66    begin
67          wait for clk_period - 3ps;
68
69       --Write value in $t0
70          write_sel <= "01000" ; --$t0
71          write_data <= "00001111000011110000111100001111" ;
72          write_ena <= '1' ;
73          wait for clk_period * 1;
74
75       --Write value in $s0
76          write_sel <= "10000" ; --$s0
77          write_data <= "11110000111100001111000011110000" ;
78          write_ena <= '1' ;
79          wait for clk_period * 1;
80
81       --Read data from $t0 and $s0
82          read_sel1 <= "01000" ; --$t0
83          read_sel2 <= "10000" ; --$s0
84          write_ena <= '0' ;
85          wait for clk_period * 2;
86
87          report "Test1";
88          assert(data1 = "00001111000011110000111100001111") report "1:Fail" severity error;
89          report "Test2";
90          assert(data2 = "11110000111100001111000011110000") report "2:Fail" severity error;
91
92          wait for clk_period * 1 ;
93
94       --Read data from $t0 and $s0 and write new data in $t0
95          read_sel1 <= "01000" ; --$t0
96          read_sel2 <= "10000" ; --$s0
97          write_sel <= "01000" ; --$t0
```

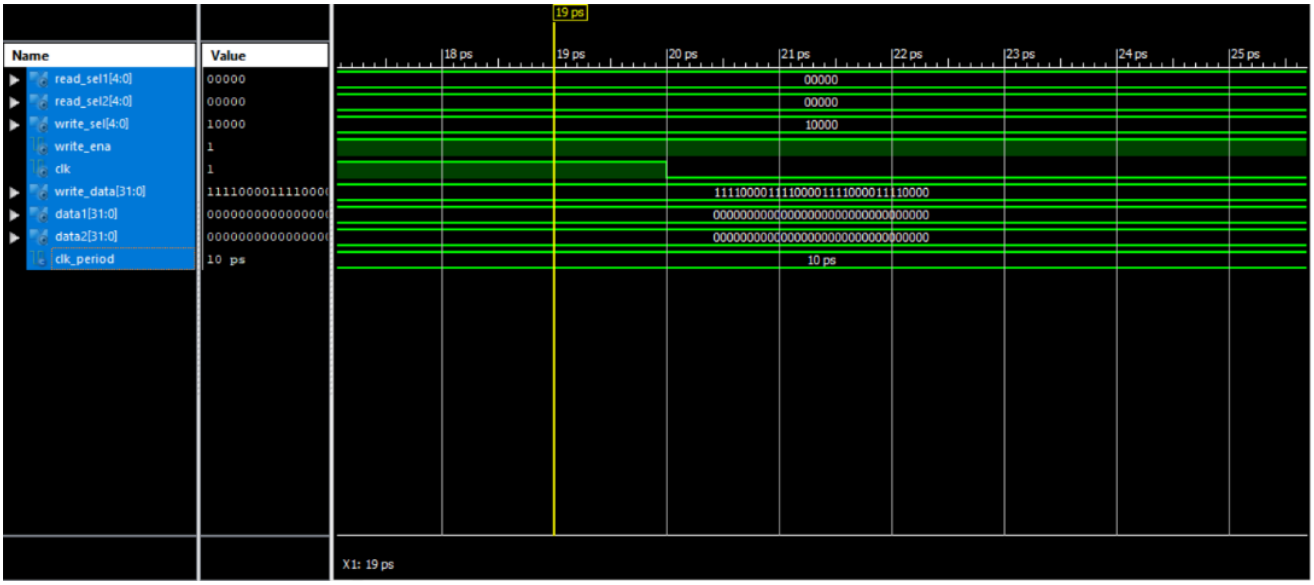
```
84          write_ena <= '0' ;
85          wait for clk_period * 2;
86
87          report "Test1";
88          assert(data1 = "00001111000011110000111100001111") report "1:Fail" severity error;
89          report "Test2";
90          assert(data2 = "11110000111100001111000011110000") report "2:Fail" severity error;
91
92          wait for clk_period * 1 ;
93
94       --Read data from $t0 and $s0 and write new data in $t0
95          read_sel1 <= "01000" ; --$t0
96          read_sel2 <= "10000" ; --$s0
97          write_sel <= "01000" ; --$t0
98          write_data <= "00000000000000000000000000000000" ;
99          write_ena <= '1' ;
100
101         wait for clk_period * 2;
102
103         report "Test3";
104         assert(data1 = "00000000000000000000000000000000") report "3:Fail" severity error;
105         report "Test4";
106         assert(data2 = "11110000111100001111000011110000") report "4:Fail" severity error;
107      -- insert stimulus here
108      report "Test Complete";
109      wait;
110   end process;
111
112
113
114
115
116 END;
117
```

1.4 Register Output

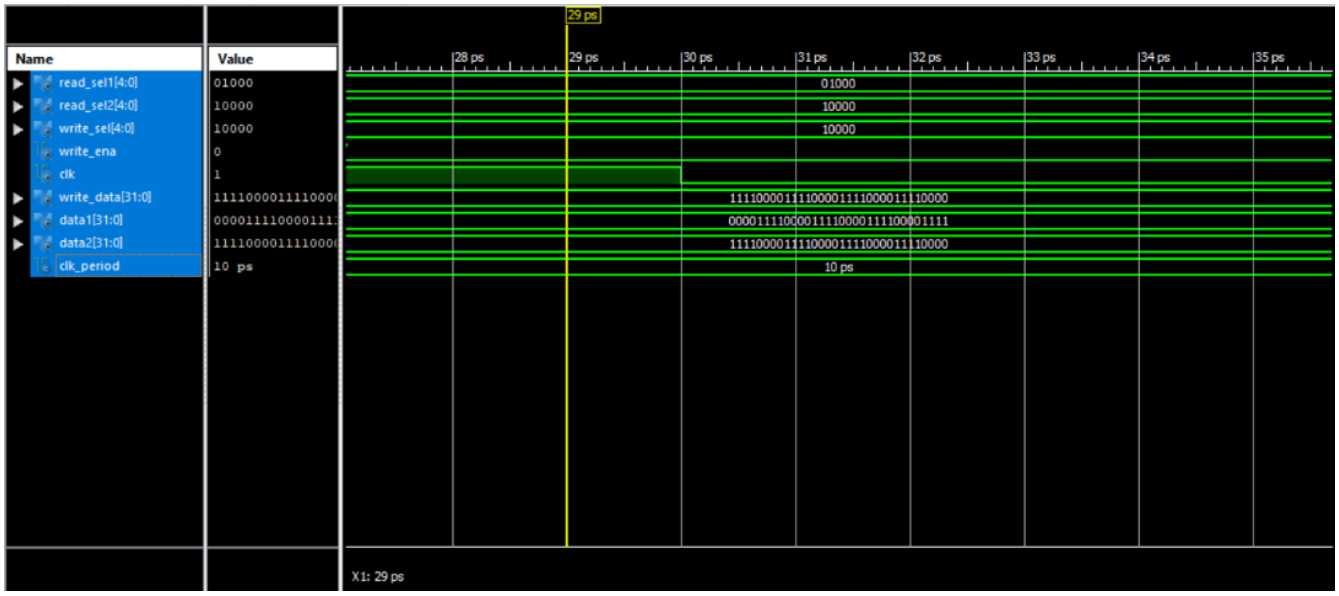
1.4.1 Output 1



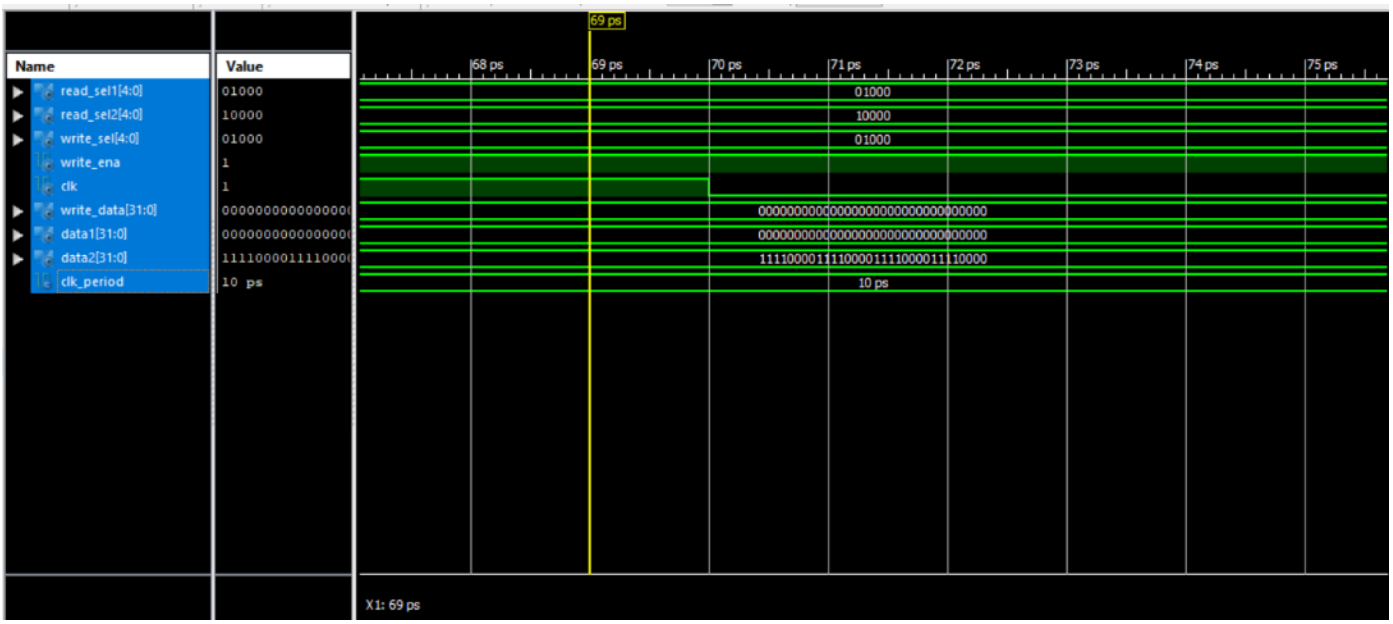
1.4.2 Output 2



1.4.3 Output 3



1.4.4 Output 4



2.0 ALU

2.1 Entity Declaration

The following entity was declared according to project document as requested.

```
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22  use ieee.std_logic_signed.all;
23
24  entity ALU is
25      Port ( data1 : in  STD_LOGIC_VECTOR (31 downto 0);
26             data2 : in  STD_LOGIC_VECTOR (31 downto 0);
27             aluop : in  STD_LOGIC_VECTOR (3 downto 0);
28             cin : in  STD_LOGIC;
29             dataout : out  STD_LOGIC_VECTOR (31 downto 0);
30             cflag : out  STD_LOGIC;
31             zflag : out  STD_LOGIC;
32             oflag : out  STD_LOGIC);
33  end ALU;
```

Then we started out by writing the ALU's architecture. We made the decision to implement the ALU using procedures, variables, the case statement, and a for loop. We initially created the process sensitivity list of (data1,data2,aluop,cin) to force a reevaluation of the results whenever one of these four inputs changes. The following variables were declared: tmp, which will be utilised in the addition and subtraction processes, has a size range of(n down to 0 - 32 down to 0),cintmp will be used to store the carry in of the addition and subtraction processes, and outv (with size of n-1 down to 0 - 31 down to 0) will be used to store the final results of the ALU's functions, cvar for the carry out of the addition and subtraction processes, and finally zvar which will be used to check if the result of the requested function is equal to zero or not. Also, we initialized the cflag, oflag, tmp, zvar all with zeros.

```
34
35  architecture Behavioral of ALU is
36
37  begin
38  process (data1, data2, aluop, cin)
39  variable tmp: std_logic_vector (32 downto 0);
40  variable outv: std_logic_vector (31 downto 0);
41  variable cvar, zvar, cintmp: std_logic;
42  begin
43  cflag <= '0';
44  oflag <= '0';
45  tmp := "00000000000000000000000000000000";
46  zvar := '0';
```

Following that, we began to write the case process with "case aluop" for various actions to be taken based on the value of the aluop. When the aluop equals "0010," the first scenario involves addition. The cintmp will then be equal to the cin's value. Data1 and Data2 are not added directly; instead, they are concatenated with a '0' at the most significant bit to temporarily increase the size from 32 bits to 33 bits in order to provide an extra bit empty for holding any carry out. The result of the addition of data1 and data2 and cintmp (carry input if there is carry input by the user) will be the value of the tmp. The 33rd bit will be put in the cvar, and the outv will be equal to the tmp taking only the first 32 bits. We created the equation for the oflag to detect overflow, which occurs when two large numbers are added that are greater than the value that the software could handle, producing an unexpected answer with an unexpected sign. Overflow occurs, for instance, when adding two large positive numbers and the result turns out to be negative or when adding two large negative numbers and the result turns out to be positive. The value of the cvar is then assigned to the cflag. The subtraction is the same case of the addition as we implemented the subtraction to be carried out using addition of the 2's complement instead of direct subtraction, and it happens when the aluopis equal to "0110" along with the cintmp equal to '1' and with a subtraction operand instead. Moreover, the rest of the

operations is completed where the "AND" operation is done when the aluop is equal to "0000", the "OR" operation is done when the aluop is equal to "0001", and the "NOR" operation is done when the aluop is equal to "1100". Also, the results of the operations is store in the outv as well. In the case, no one of the aluop stated is entered by the user, the outv will be filled with 'Z'.

```
50  when "0010" =>
51  cintmp := cin;
52  tmp := ('0' & data1) + ('0' & data2) + cintmp;
53  outv:= tmp (31 downto 0);
54  cvar := tmp(32);
55  oflag <= (data1(31) and  data2(31) and not (outv(31))) or (not(data1(31)) and not (data2(31)) and outv(31));
56  cflag <= cvar;
57
58
59  when "0110" =>
60  cintmp := '1';
61  tmp := ('0' & data1) + ('0' & not(data2)) + cintmp;
62  outv := tmp (31 downto 0);
63  cvar := not (tmp(32));
64  oflag <= (data1(31) and not (data2(31)) and not (outv(31))) or (not(data1(31)) and data2(31) and outv(31));
65  cflag <= cvar;
66
67  when "0000" =>
68  outv := data1 and data2;
69  when "0001" =>
70  outv := data1 or data2;
71  when "1100" =>
72  outv := data1 nor data2;
73  when others =>
74  outv := "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
75  end case;
```

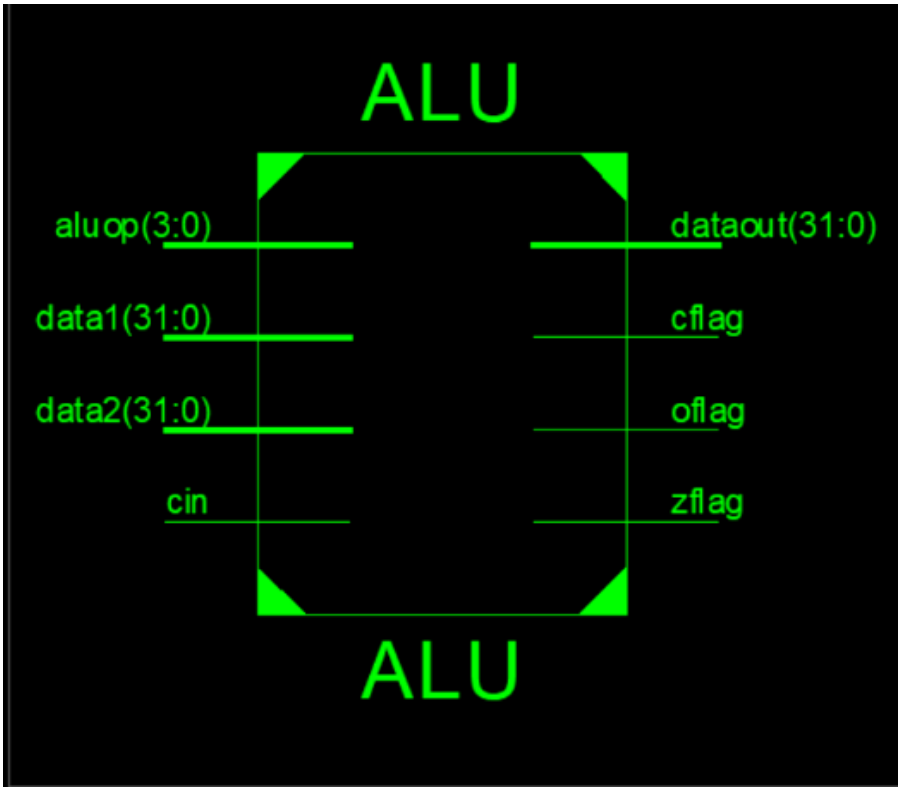
Next, we constructed a for loop that basically checks for all the bits of the outv starting for the bit no. 0 to the bit no. 31 if they are all equal to zero by making an "OR" operation with the zvar which was initialized previouslyby zero. If all the bits are zeros, the zvar will remain equal to 0.

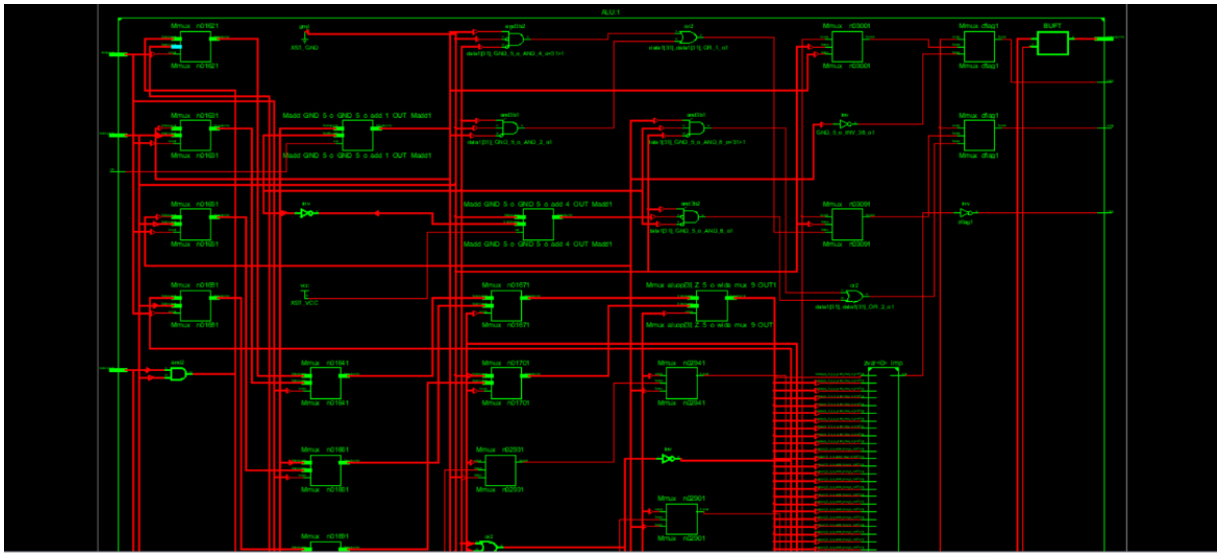
```
77  for i in 0 to 31 loop
78  zvar := zvar or outv(i);
79  end loop;
```

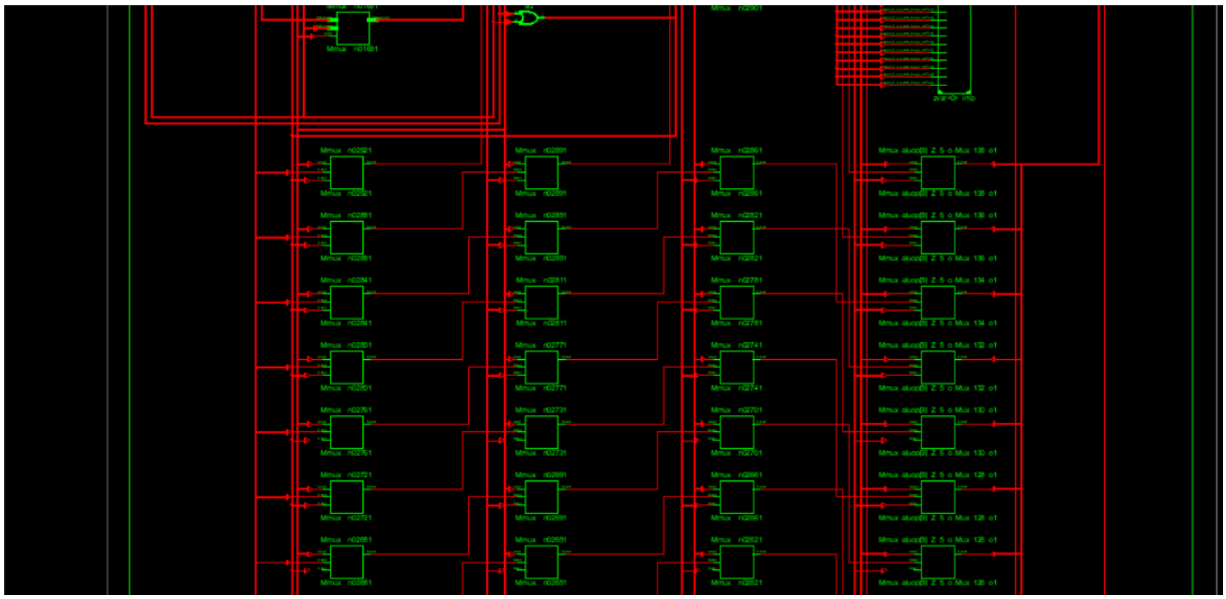
At the end, dataout is assigned to hold the contents of the outv and the zflag is assigned to hold the opposite of the zvar to be equal to ‘1’ when all the bits are equal to ‘0’ and vice versa.

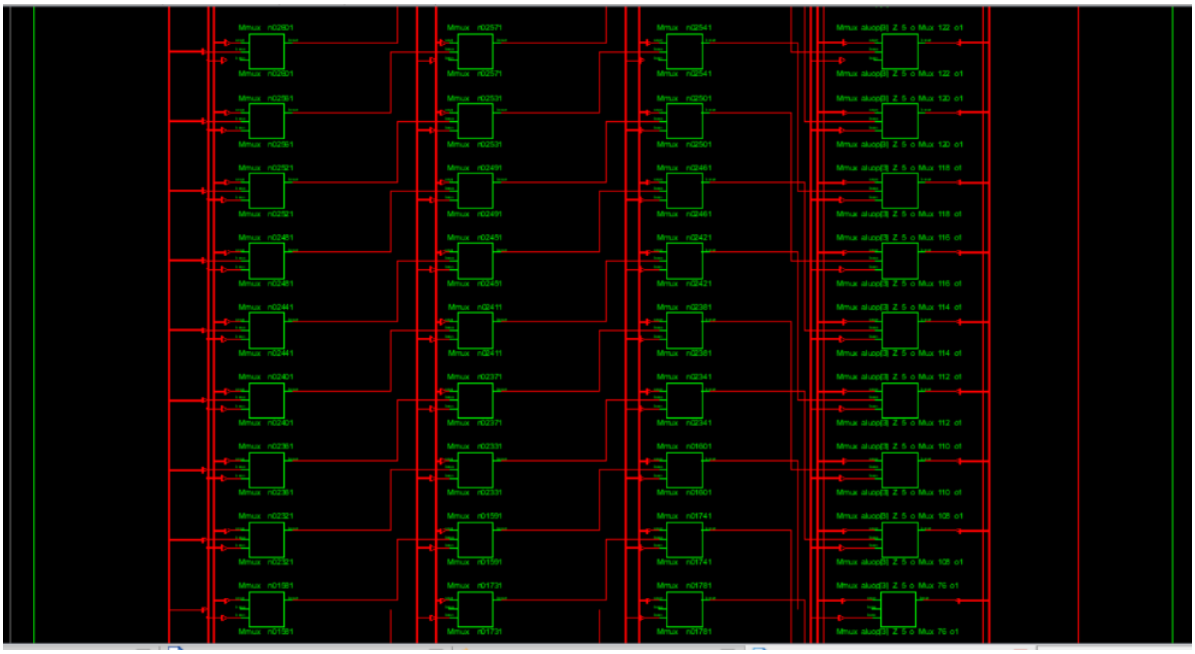
```
81   dataout <= outv;
82   zflag <= not zvar;
83   end process;
```

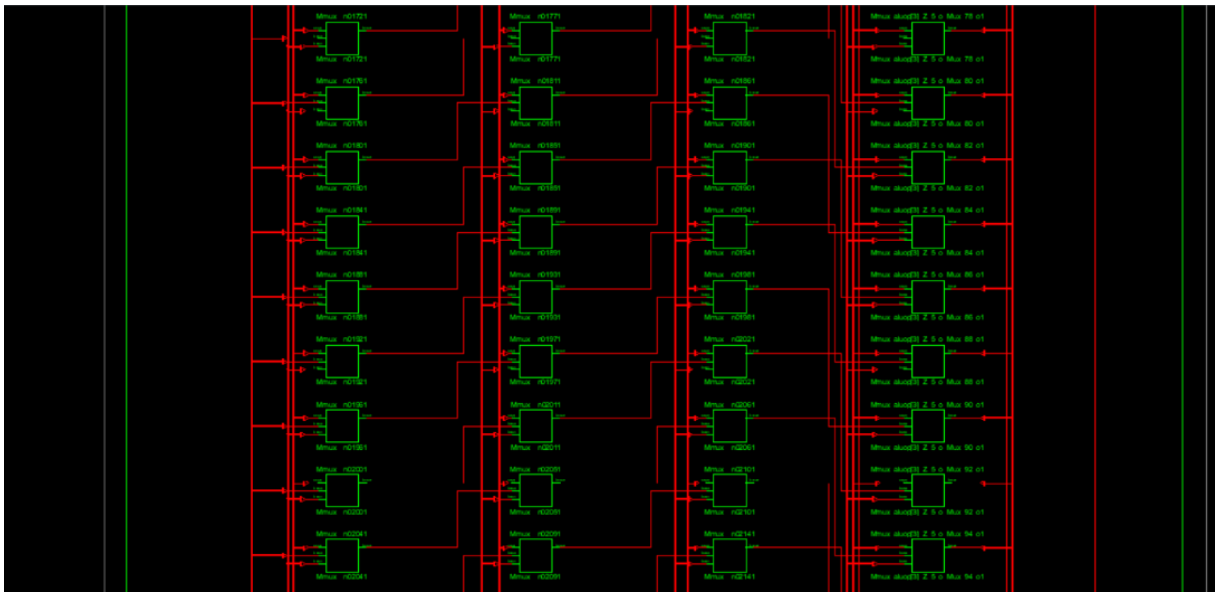
2.2 ALU RTL Schematic

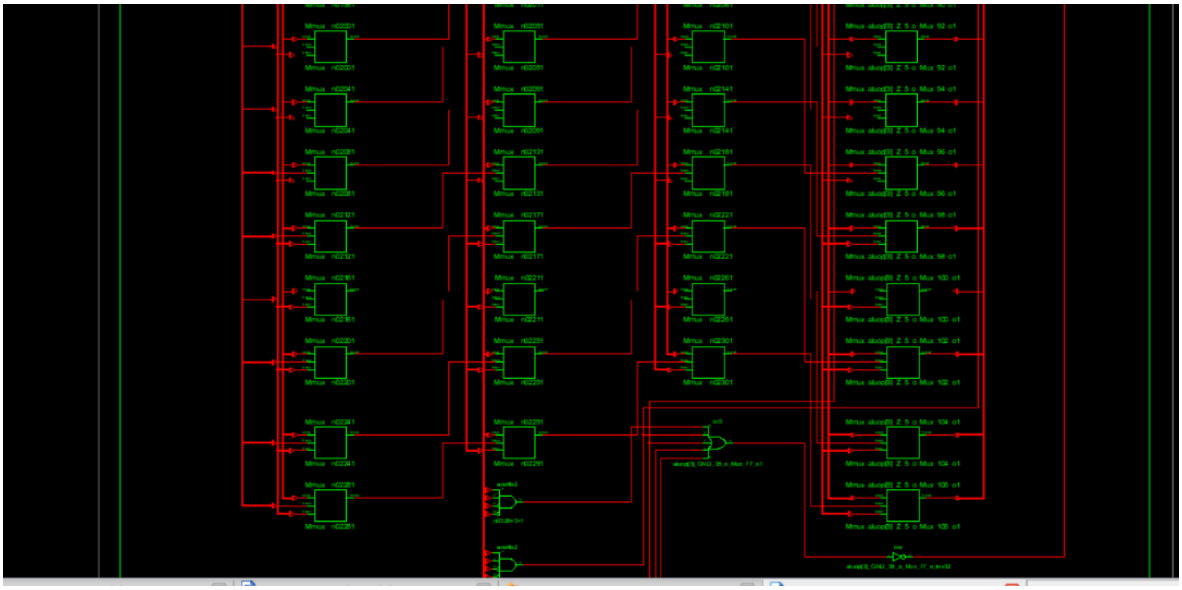


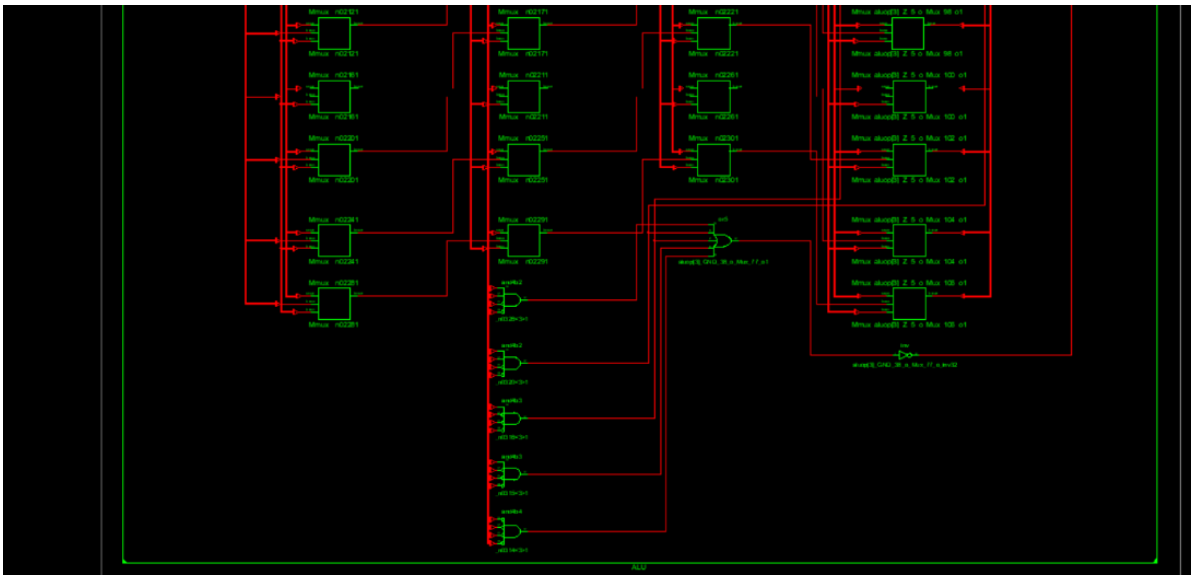












2.3 ALU Test Bench Code

```
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;

ENTITY ALUTest IS
END ALUTest;

ARCHITECTURE behavior OF ALUTest IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT Alu    PORT(
          data1: in  STD_LOGIC_VECTOR(31 downto 0);
           data2: in  STD_LOGIC_VECTOR(31 downto 0);
           aluop: in std_logic_vector (3 downto 0);
           Cin : in  std_logic;

           dataout : out  STD_LOGIC_VECTOR (31 downto 0);
           cflag : out  STD_LOGIC;
           oflag : out  STD_LOGIC;
           zflag : out  STD_LOGIC
         );
    END COMPONENT;


   --Inputs
   signal data1: std_logic_vector(31 downto 0) := (others => '0');
   signal data2: std_logic_vector(31 downto 0) := (others => '0');
   signal aluop : std_logic_vector(3 downto 0) := (others => '0');
   signal Cin : std_logic := '0';
```

```
BEGIN

   -- Instantiate the Unit Under Test (UUT)
   uut: alu PORT MAP (
          data1=> data1,
          data2=> data2,
          aluop => aluop,
          Cin => Cin,
          dataout => dataout,
          cflag => cflag,
          zflag => zflag,
          oflag => oflag
        );


   -- Stimulus process
   stim_proc: process
   begin
   cin <= '0';
      -- hold dataoutet state for 100 ns.
         wait for 10 ns;
      --AND testcase
         data1<= "11000000000000000000000000000000" ;
         data2<= "10100000000000000000000000000000" ;
         aluop <= "0000" ;
         wait for 10ns;
         report "Test1";
         assert(dataout = "10000000000000000000000000000000" and zflag = '0') report "1:Fail" severity error;

         wait for 1ns;

      --OR testcase
         data1<= "11000000000000000000000000000000" ;
         data2<= "10100000000000000000000000000000" ;
```

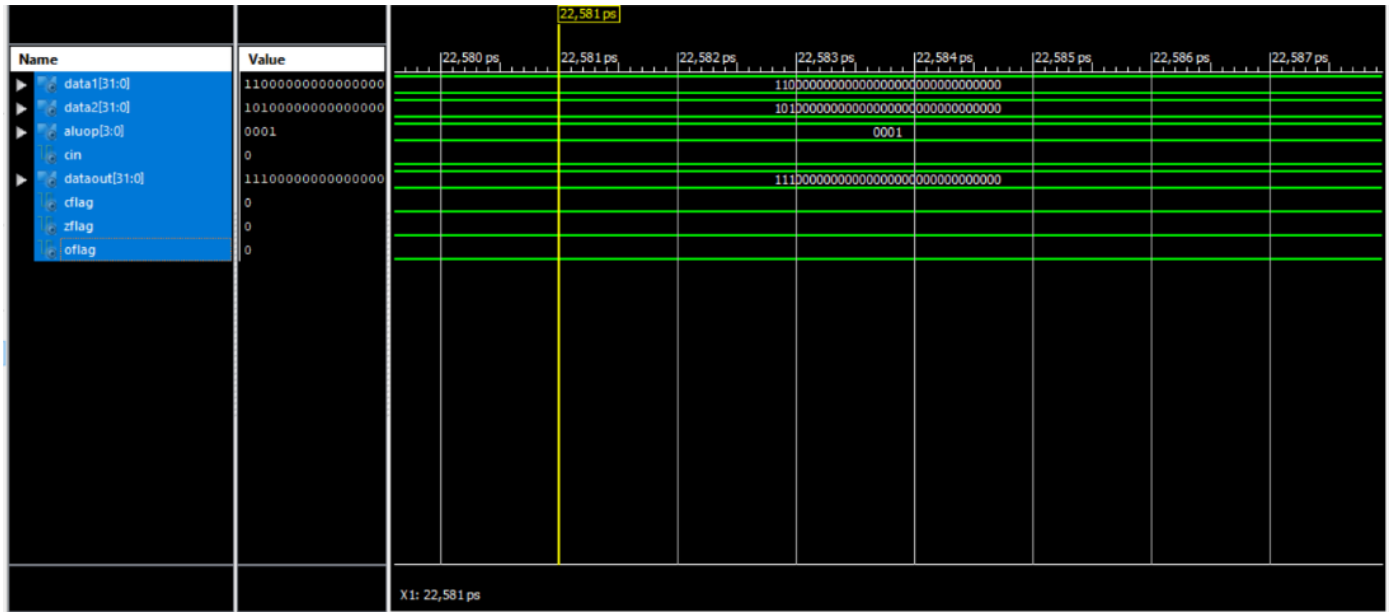
```
74
75          --OR testcase
76             data1<= "11000000000000000000000000000000" ;
77             data2<= "10100000000000000000000000000000" ;
78             aluop <= "0001" ;
79             wait for 10ns;
80             report "Test2";
81             assert(dataout = "11100000000000000000000000000000" and zflag = '0') report "2:Fail" severity error;
82
83             wait for 1ns;
84
85          --ADD testcase1 (overflow = 1, cout = 0)
86             data1<= "01110000000000000000000000000000" ;
87             data2<= "01100000000000000000000000000000" ;
88             aluop <= "0010" ;
89             wait for 10ns;
90             report "Test3";
91             assert(dataout = "11010000000000000000000000000000" and oflag = '1' and cflag = '0' and zflag = '0') report "3:Fail" severity error;
92
93             wait for 1ns;
94
95          --ADD testcase2 (zero = 1, cout = 1)
96             data1<= "11110000000000000000000000000000" ;
97             data2<= "00010000000000000000000000000000" ;
98             aluop <= "0010" ;
99             wait for 10ns;
100            report "Test4";
101            assert(dataout = "00000000000000000000000000000000" and oflag = '0' and cflag = '1' and zflag = '1') report "4:Fail" severity error;
102
103            wait for 1ns;
104
105         --SUdata2testcase1 (cout = 1)
106            data1<= "00000000000000000000000000000111" ; --data1= 7
107            data2<= "00000000000000000000000000000110" ; --data2= 6
```

```
99             wait for 10ns;
100            report "Test4";
101            assert(dataout = "00000000000000000000000000000000" and oflag = '0' and cflag = '1' and zflag = '1') report "4:Fail" severity error;
102
103            wait for 1ns;
104
105         --SUdata2testcase1 (cout = 1)
106            data1<= "00000000000000000000000000000111" ; --data1= 7
107            data2<= "00000000000000000000000000000110" ; --data2= 6
108            Cin <= '1' ;
109            aluop <= "0110" ;
110            wait for 10ns;
111            report "Test5";
112            assert(dataout = "00000000000000000000000000000001" and oflag = '0' and cflag = '1' and zflag = '0') report "5:Fail" severity error;
113
114            wait for 1ns;
115
116         --SUdata2testcase2 (cout = 0)
117            data1<= "00000000000000000000000000000110" ; --data1= 6
118            data2<= "00000000000000000000000000000111" ; --data2= 7
119            Cin <= '1' ;
120            aluop <= "0110" ;
121            wait for 10ns;
122            report "Test6";
123            assert(dataout = "11111111111111111111111111111111" and oflag = '0' and cflag = '0' and zflag = '0') report "6:Fail" severity error;
124
125            wait for 1ns;
126
127         report "Test Complete";
128         wait;
129      end process;
130
131   END;
132
```

2.4 ALU Output

2.4.1 AND Output

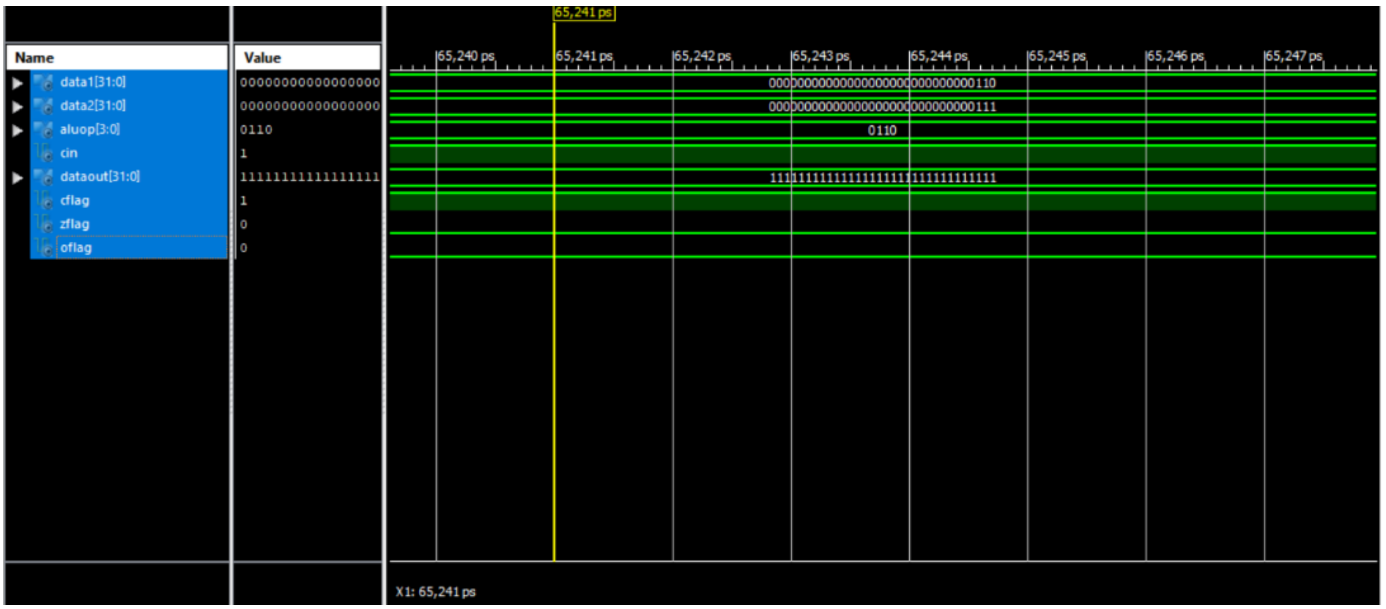
2.4.2 OR Output



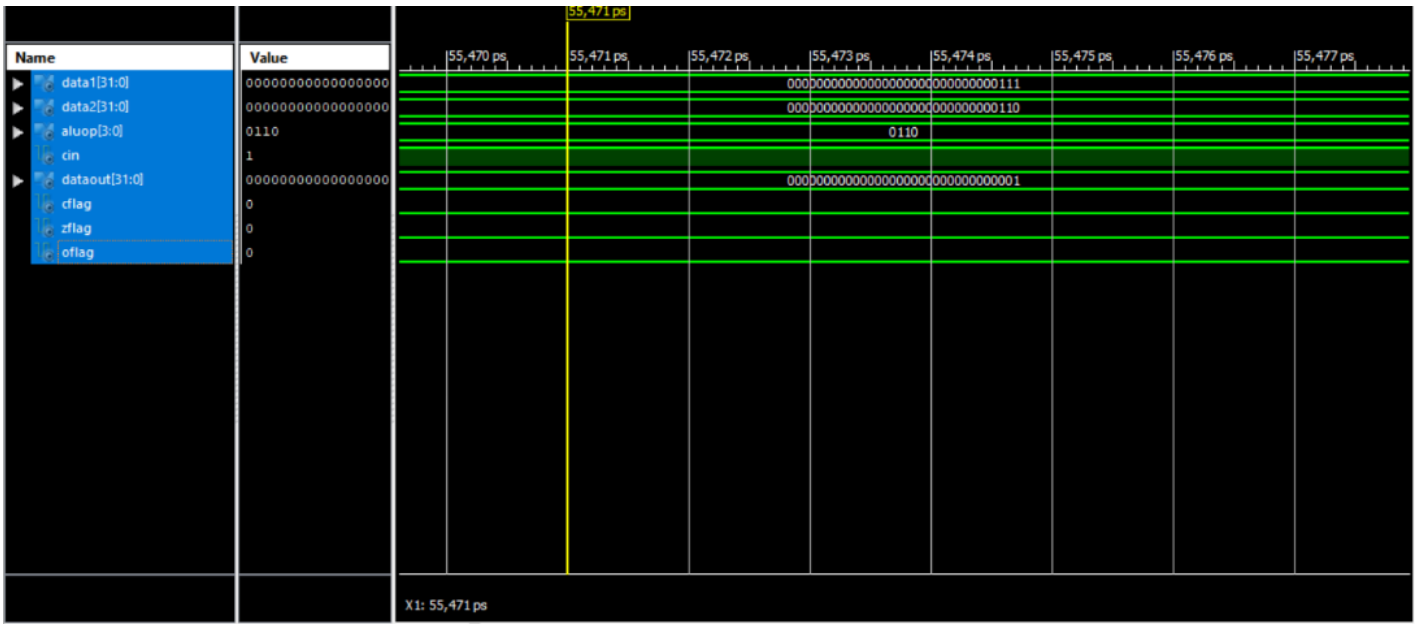
2.4.3 ADD Test Case1 Output

2.4.4 ADD Test Case2 Output

2.4.4 Sub Test Case1 Output



2.4.6 Sub Test Case2 Output



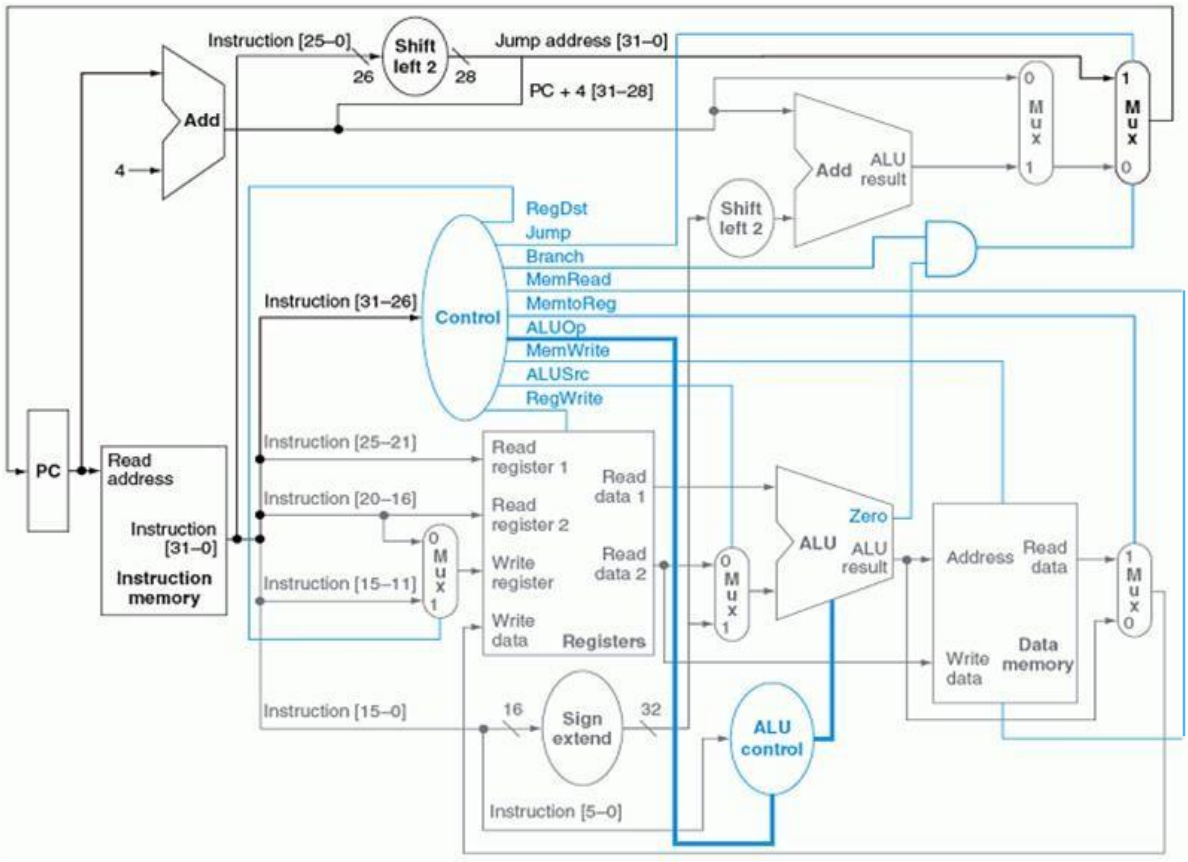
PHASE 2

Table of Contents

1.0 CPU Design

1.1 PC

In phase 2, it was required to design a simple MIPS CPU like the one in the following diagram. The CPU should be able to perform certain instructions: R-type (AND, OR, ADD, SUB, SLT and NOR), I-type (lw, sw, beq, bne) and J instruction.



In the beginning, we created the "PC" module, which has three inputs and one output. "clk," "rst," and "address" are the three inputs, and "PC" is the only output. While the "clk" and the "rst" are only one bit buses, the "address" and the "PC" are both 32 bit buses. Then, rather than using the output directly, we created a signal called "instaddress" to be used as a buffer inside the process, and the signal will then be assigned to the output

at the end. The PC's architecture is straightforward because it consists of a process with the names "PC Process" and "clk" and "rst" in the sensitivity list. It works by placing the input “address” into the signal “instaddress” (substitute of the output) the "instaddress" will be reset and the entire 32 bits will be zeros if the "rst" happens to be equal to "1" at the rising edge of the clock.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  entity PC is
4  port    (
5      clk: in std_logic;
6      rst : in std_logic;
7      address: in std_logic_vector (31 downto 0);
8      PC: out std_logic_vector (31 downto 0));
9  end PC;
10
11 architecture Behavioral of PC is
12
13 signal instAddress: std_logic_vector (31 downto 0) := x"00000000";
14
15 begin
16
17 PC_Process: process (clk, rst)
18
19 begin
20 if rising_edge(clk) then
21    instaddress <= address;
22    end if;
23 if rst= '1' then
24    instaddress <= x"00000000";
25    end if;
26 end process;
27
28 PC <= instaddress;
29 end behavioral;
30
```

1.2 Adder

Then, we created the "plus4" module, which increments the output from the PC module by 4. It has one input and one output, both of which are 32 bits in size. The architecture is straightforward and uses the "signed" function required for addition in the "Numeric STD" library. "o" will equal "i+ 4".

```
1  Library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  USE IEEE.NUMERIC_STD.ALL;
4
5  entity plus4 is
6     port ( i: in std_logic_vector (31 downto 0);
7                 o: out std_logic_vector (31 downto 0));
8  end plus4;
9  architecture Behavioral of plus4 is
10 begin
11 o <= std_logic_vector(signed(i)+4);
12 end behavioral;
13
```

1.3 Shift Left

Then, we created the "SLA2" module, which preserves the sign bit while shifting the input received by 2 bits from the left. It uses one input and generates one output, both of which are 32 bits in size. The architecture is straightforward, preserving the sign-bit by making the 32nd bit of the "o" (Output (31)) equal to the 32nd bit of the "i" (Input (31)) which is the step of preserving the sign-bit. Then, three bits are skipped from the input to perform the shifting operation, including the sign-bit that has already been transferred (the 32nd bit) and the two bits for the shifting that will be the 31st and 30th bits. Then, from the 29th bit until the 0th bit of the input, the shifting operation is performed. In order to represent the 31st bit as the third bit (Output (30 down to 2)), (Input (28 down to 0)) will be transferred to the output. As a result, the output is generated, but the first and zeroth bits

are left unassigned (Output (1 down to 0)), which will be filled with zeros in accordance with the shifting rules.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  entity SLA2 is
4        port ( i: in STD_LOGIC_VECTOR (31 downto 0);
5                  o: out STD_LOGIC_VECTOR (31 downto 0));
6  end sla2;
7  architecture Behavioral of sla2 is
8  begin
9  --to keep the sign:
10 o(31) <=  i(31);
11 o(30 downto 2) <= i(28 downto 0);
12 o(1 downto 0) <= (others => '0');
13 end behavioral;
```

1.4 Multiplexers

Fourthly, we created two multiplexers, each a 2x1 Mux that takes two inputs, a single bit selection line, and produces a single output. The two multiplexers share the same architecture and functionality, but one is built to take 32-bit inputs producing one 32-bit output, while the other takes 5-bit inputs and outputs one 5-bit output. The reason for this is that while all multiplexers used by the CPU operate on a 32-bit basis, the multiplexer whose output enters the "WriteRegister," which receives the register's address for the data to be written into, only operates on a 5-bit basis. The two multiplexers are referred to as "muxx2x1" and "mux5bitx1" respectively (5x1 was only used to denote that it receives 5 bits and outputs 1). The architecture is simple; the selection line, which is referred to as "s" in "muxx2x1" and "S" in "mux5bitx1" determines the output. If the selection line equals "0," the output will equal the input "i0," and if it equals "1," the output will equal the input "i1," while all other cases will result in 32 Zs.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  entity muxx2x1 is
4     port ( s: in std_logic;
5                i0: in std_logic_vector (31 downto 0);
6                 i1: in std_logic_vector (31 downto 0);
7                             o: out std_logic_vector (31 downto 0));
8  end muxx2x1;
9  architecture Behavioral of muxx2x1 is
10 begin
11 o <=  i0 when s = '0' else
12    i1 when s = '1' else
13    (others => 'Z');
14 end Behavioral;
15
```

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  entity mux5bitx1 is
4     Port ( S: in std_logic;
5          i0: in std_logic_vector (4 downto 0);
6          i1: in std_logic_vector (4 downto 0);
7          o: out std_logic_vector (4 downto 0));
8  end mux5bitx1;
9  architecture Behavioral of mux5bitx1 is
10 begin
11 o <= i0 when s= '0' else
12    i1 when s= '1' else
13    (others => 'Z');
14 end behavioral;
```

1.5 Sign Extend

Then, we created a module called "signExtend" that takes a 16-bit input and extends its sign to produce a 32-bit output, where the 16-bit difference is a repetition of the sign-bit. The two lines in the architecture that check the sign-bit of the input, which is the 16th bit (Input (15)), are how it works. If the sign-bit is a 1, the output will be the same as the input, but concatenated with 16 ones to the left side of the input. If the

sign-bit is a 0, the output will be the same as the input, concatenated with 16 zeros to the left of the input. Any other case will result in Zs.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  entity signExtend is
4  port (   i: in std_logic_vector (15 downto 0);
5     o: out std_logic_vector (31 downto 0));
6  end signExtend;
7  architecture Behavioral of signExtend is
8  begin
9  o <= "0000000000000000" & i when i(15) = '0' else
10          "1111111111111111" & i when i(15) = '1' else
11    (others => 'Z');
12 end Behavioral;
13
```

1.6 Jump Address

Then, we created a module called "JA" that will be in charge of producing the jump address by concatenating the first 28 bits of the output from shifting left 2 of the instruction (rightin (27 down to 0)) with the final 4 bits of the output from the "plus4" module (PC + 4) output(leftin (31 downto 28).

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  entity JA is
4  Port (leftin:in STD_LOGIC_VECTOR (31 downto 0) ;
5  rightin:in
6  STD_LOGIC_VECTOR (31 downto 0);
7  Output :out STD_LOGIC_VECTOR (31 downto 0));
8  end JA;
9  architecture Behavioral of JA is
10 begin
11 Output <= leftin (31 downto 28) & rightin (27 downto 0);
12 end Behavioral;
```

1.7 Control Unit

Then we developed the "ControlUnit," which is in charge of managing the CPU. It only requires a single 6-bit input and generates 9 single-bit outputs, with the exception of the "ALUOp" output, which will be 2 bits. The instruction's final six bits are loaded into the input, or OpCode (instruction [31-26]). MemRead, MemWrite, RegDst, Branch, MemtoReg, ALUSrc, Jump, and RegWrite are the eight single bit outputs. The cases were based on the following truth table except for jump and bne instructions The outputs are identical to those of the beq process, but the bne opcode is "000101," which is different from the beq opcode. The opcode for the jump will be "000010." Except for the RegDst and the MemtoReg, which will be don't cares ('X'), and obviously the jump output, which will be equal to '1', all of the outputs are zeros. All outputs will be reset to zero in all other circumstances.

Control	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

```
1  library IEEE;
2  Use IEEE.STD_LOGIC_1164.ALL;
3  entity ControlUnit is
4  Port
5  (     Opcode:in STD_LOGIC_VECTOR (5 downto 0) ;
6        MemRead:out STD_LOGIC;
7        Memwrite : out STD_LOGIC;
8        RegDst:out STD_LOGIC;
9        Branch:out STD_LOGIC;
10       MemtoReg:out STD_LOGIC;
11       ALUSrc:out STD_LOGIC;
12       ALUOp:out STD_LOGIC_VECTOR (1 downto 0) ;
13       Jump: out STD_LOGIC;
14       RegWrite:out STD_LOGIC);
15 end ControlUnit;
16
17
18 architecture Behavioral of ControlUnit is
19 begin
20 Process(Opcode)
21 BEGIN
22 RegWrite<= '0';
23 CASE OpCode Is
24
25
26
27 --R format--
28
29
30
31 WHEN "000000"=>
32 MemRead<= '0' ;
33 MemWrite <= '0' ;
34 RegDst <= '1' ;
```

```
34  RegDst <= '1' ;
35  Branch <= '0' ;
36  MemtoReg <= '0' ;
37  ALUSrc <= '0' ;
38  ALUOp <= "10";
39  Jump <= '0' ;
40  RegWrite <= '1' ;
41
42
43
44  --lw--
45
46
47
48  WHEN "100011"=>
49  MemRead <= '1' ;
50  MemWrite <= '0' ;
51  RegDst <= '0' ;
52  Branch <= '0' ;
53  MemtoReg <= '1' ;
54  ALUSrc<= '1' ;
55  ALUOp <= "00";
56  Jump <= '0';
57  RegWrite <= '1' ;
58
```

```
61  --sw--
62  WHEN "101011" =>
63  MemRead <= '0';
64   MemWrite <= '1' ;
65  RegDst <= 'X';
66  Branch <= '0';
67  MemtoReg <= 'X';
68  ALUSrc <= '1' ;
69  ALUOp <= "00";
70  Jump <= '0' ;
71  RegWrite <= '0' ;
72
73
74
75  --beq--
76  WHEN "000100" =>
77  MemRead <= '0' ;
78  MemWrite <= '0' ;
79  RegDst <= 'X' ;
80  Branch <= '1' ;
81  MemtoReg <= 'X' ;
82  ALUSrc <= '0' ;
83  ALUOp <= "01";
84  Jump <= '0' ;
85  RegWrite <= '0' ;
86
87
88
89  --bne--
90  WHEN "000101" =>
91  MemRead <= '0' ;
92  MemWrite <= '0' ;
93  RegDst <= 'X' ;
94  Branch <= '1' ;
```

```
 95  MemtoReg <= 'X' ;
 96  ALUSrc <= '0' ;
 97  ALUOp <= "01";
 98  Jump <= '0' ;
 99  RegWrite <= '0' ;
100
101
102
103  --beq--
104  WHEN "000010" =>
105  MemRead <= '0' ;
106  MemWrite <= '0' ;
107  RegDst <= 'X' ;
108  Branch <= '0' ;
109  MemtoReg <= 'X' ;
110  ALUSrc <= '0' ;
111  ALUOp <= "00";
112  Jump <= '1' ;
113  RegWrite <= '0' ;
114
115
116
117  WHEN OTHERS => NULL;
118  MemRead <= '0' ;
119  MemWrite <= '0' ;
120  RegDst <= '0' ;
121  Branch <= '0' ;
122  MemtoReg <= '0' ;
```

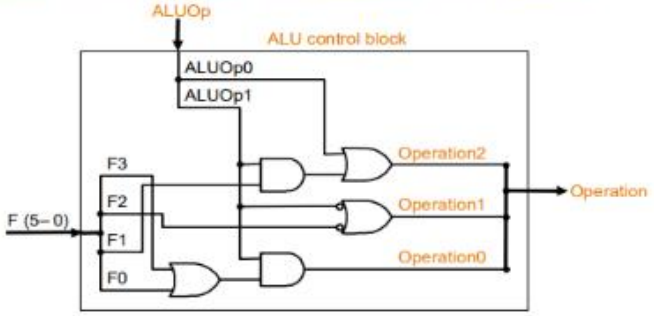
```
123  ALUSrc <= '0' ;
124  ALUOp <= "00";
125  Jump <= '0' ;
126  RegWrite <= '0' ;
127
128
129
130  END CASE;
131
132
133
134  END PROCESS;
135
136
137
138  end Behavioral;
```

1.8 ALU Control Unit

The "ALUControlUnit" came next. This module accepts two inputs: the ALUOp, which is 2 bits and is the same ALUOp coming out of the "ControlUnit," and the FunctCode, which is 6 bits (the function field in the instruction is the first 6 bits, instruction [5-0]). The "ALUFunct" output, which has a 4 bit size, is the ALU Control Lines input, which the ALU will use to determine which operation to perform. The upcoming truth table and diagram served as the basis for designing the "ALUControlUnit."

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

- **The circuit for ALU control unit**
  - Obtained through combinational digital logic design method



The Registers, ALU, Data Memory, and Instruction Memory are the four remaining CPU components or modules. You can refer back to the descriptions of both modules in the "Phase 1" section for more information. For the Registers and the ALU, we used the previously constructed modules of "RegisterFile" and "ALU" of Phase 1. However, we added the SLT function (Set On Less Than) to the ALU.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  entity ALUControlUnit is
4  Port ( Functcode : in STD_LOGIC_VECTOR (5 downto 0);
5  ALUOp:in STD_LOGIC_VECTOR (1 downto 0);
6  ALUFunct:out STD_LOGIC_VECTOR (3 downto 0));
7  end ALUControlUnit;
8  architecture Behavioral of ALUControlUnit is
9  begin
10
11
12
13 ALUFunct(3) <= (ALUOp(1) AND FunctCode (0) AND FunctCode (1) AND FunctCode (5));
14
15
16
17 ALUFunct (2) <= ALUOp (0) OR (ALUOp(1) AND Functcode (1)) ;
18
19
20
21 ALUFunct (1) <= NOT ALUOp(1) OR ((NOT FunctCode (2)) AND NOT (FunctCode (0)));
22
23
24
25 ALUFunct (0) <= (FunctCode (3) OR FunctCode (0)) AND (NOT (FunctCode(1) AND FunctCode (0) )) AND ALUOp(1);
26
27
28
29 end Behavioral;
```

SLT in ALU in RegisterFile is below.

```
75  when "0111" =>
76  if(datal < data2) Then
77  outv := x"00000001";
78  else
79  outv := x"00000000";
80  end if;
81
```

1.9 Instruction and Data Memory

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3  use IEEE.STD_LOGIC_SIGNED.all;
4  use IEEE.STD_LOGIC_ARITH.all;
5  use IEEE.std_logic_textio.all;
6  library STD;
7  use STD.textio.all;
8
9
10 entity instrmemory is
11    port( LoadIt: in STD_LOGIC;
12          DATA: out STD_LOGIC_VECTOR (31 downto 0);
13          ADDRESS: in STD_LOGIc_VECTOR (31 downto 0);
14          CLK : in STD_LOGIC);
15 end instrmemory;
16
17
18 architecture BEHAVIORAL of instrmemory is
19 signal ADDRover4: STD_LOGIC_VECTOR (29 downto 0) ;
20 begin
21
22 ROM_PROCESS: process (CLK, ADDRESS) is
23 type MEM is array (0 to 63) of STD_LOGIC_VECTOR (31 downto 0);
24 variable MEMORY: MEM:= (others => X"00000000");
25 variable IADR: INTEGER;
26 begin
27 if LoadIt = '1' then
28
29
30 MEMORY(0):=
31 "00000000000000000100000000100101";
32 MEMORY (1):=
33 "10001100000011010000000000110000";
34 MEMORY (2):=
```

```
35 "10001100000011000000000000110100";
36 MEMORY (3):=
37 "10001100000011001000000000111000";
38 MEMORY (4):=
39 "00000000000011000010100000100000";
40 MEMORY (5):=
41 "10101101000010100000000000000000";
42 MEMORY (6):=
43 "10101101000010100000000000000100";
44 MEMORY (7):=
45 "00000001101110000100100000100010";
46 MEMORY (8):=
47 "00000001001110000100100000100010";
48 MEMORY (9):=
49 "10001101000010110000000000000000";
50 MEMORY (10):=
51 "10001101000011000000000000000100";
52 MEMORY (11):=
53 "00000001011011000101000000100000";
54 MEMORY (12):=
55 "10101101000010100000000000001000";
56 MEMORY (13):=
57 "00000001000110010100000000100000";
58 MEMORY (14):=
59 "00000001001110000100100000100010";
60 MEMORY (15) :=
61 "00000000000010010000100000101010";
62 MEMORY (16) :=
63 "00010000001000000000000000000001";
```

```
MEMORY (17):=
"00001000000000000000000000001001";
MEMORY (18):=
"00000000000001000010000000100100";
MEMORY (19) :=
"00000000000011010010100000100000";
MEMORY (20):=
"00001000000000000000000000010110";
MEMORY (21):=
"00000000000110001000000000100010";
MEMORY (22):=
"00000001000000000100000000100100";
MEMORY (23):=
"00000001000010000100100000100000";
MEMORY (24):=
"00000001001010010100100000100000";
MEMORY (25) :=
"00000000100010010101000000100000";
MEMORY (26):=
"10001101010100000000000000000000";
MEMORY (27):=
"00000001000110000100000000100000";
MEMORY (28):=
"00000001000001010000100000101010";
MEMORY (29):=
"00010100001000001111111111111001";
MEMORY (30):=
"00000000000000001100000000100111";
MEMORY (31):=
"00000010000110001000000000100111";
MEMORY (32):=
"00000010000110001000000000100000";
MEMORY (33) :=
"00000010000110001000000000100000";
```

```
        memory(34) := "00000000000000001010000000100101" ; -- or $8, $0, $1
        memory(35) := "00000001000000001010010000100000" ; -- add $9, $8, $1
        memory(36) := "00000001000010010101000000101010" ; -- slt $10, $8, $9
end if;
if FALLING_EDGE (CLK) then
IADR:= CONV_INTEGER (ADDRover4) ;
DATA <= MEMORY (IADR);
end if;
end process;
ADDRover4<= ADDRESS (31 downto 2);
end BEHAVIORAL;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity datamem is

port( Loadit : in STD_LOGIC;
      INPUT : in STD_LOGIC_VECTOR (31 downto 0);
      OUTPUT : out STD_LOGIC_VECTOR (31 downto 0);
      memoryread: in STD_LOGIC;
      memorywrite: in STD_LOGIC;
      ADDRESS : in STD_LOGIC_VECTOR (31 downto 0);
      CLK : in STD_LOGIC );



end datamem;


Architecture BEHAVIORAL Of datamem is type MEM is array (0 to 63) of STD_LOGIC_VECTOR (31 downto 0) ;
signal MEMORY : MEM;
signal OUTS: STD_LOGIC_VECTOR (31 downto 0) ;
signal ADDRover4: STD_LOGIC_VECTOR (29 downto 0) ;
signal ADDR_int: integer;

```

```
31
32
33  process( memoryread, memorywrite, CLK, ADDRESS, INPUT) is
34
35
36
37  begin
38
39
40
41  if Loadit = '1' then
42
43
44
45  memory (0)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
46  memory (1)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
47  memory (2)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
48  memory (3)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
49  memory (4)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
50  memory (5)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
51  memory (6)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
52  memory (7)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
53  memory (8)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
54  memory (9)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
55  memory (10) <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
56  memory (11) <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
57  memory (12) <= "00000000000000000000000000001100";
58  memory (13) <= "00000000000000000000000000000001";
59  memory (14) <= "00000000000000000000000000000100";
60
61
```

```
63  else
64
65
66
67
68  if FALLING_EDGE (CLK) then
69  if memorywrite = '1' then
70  MEMORY (ADDR_int) <= INPUT;
71  end if;
72  end if;
73  end if;
74  end process;
75  ADDRover4 <= ADDRESS (31 downto 2);
76  ADDR_int <= CONV_INTEGER(ADDRover4);
77  OUTS <= MEMORY (ADDR_int) when memoryread = '1' and (ADDR_int < 64) else
78  (others => 'Z') when memoryread = '0';
79  OUTPUT <= OUTS;
80  end BEHAVIORAL;
```

2.0 MyPackage

We made a package called "MyPackage" in which we put all the earlier components and added a number of signals that would later be used in the main module, "MIPSCPU."

```
1
2
3  library IEEE;
4  use IEEE.STD_LOGIC_1164.all;
5
6  package MyPackage is
7
8  component reg is
9   Port ( i : in  STD_LOGIC_VECTOR (31 downto 0);
10            rst : in  STD_LOGIC;
11            clk : in  STD_LOGIC;
12            load : in  STD_LOGIC;
13            inc : in  STD_LOGIC;
14            o : out  STD_LOGIC_VECTOR (31 downto 0));
15 end component;
16
17 component dec5x32 is
18  Port ( I : in  STD_LOGIC_VECTOR (4 DOWNTO 0);
19            E : in  STD_LOGIC;
20 O : out STD_LOGIC_VECTOR (31 DOWNTO 0));
21 end component;
22
```

```
22
23 component mux32x1 is
24 Port ( s : in  STD_LOGIC_VECTOR (4 downto 0);
25           i0 : in  STD_LOGIC_VECTOR (31 downto 0);
26           i1 : in  STD_LOGIC_VECTOR (31 downto 0);
27           i2 : in  STD_LOGIC_VECTOR (31 downto 0);
28           i3 : in  STD_LOGIC_VECTOR (31 downto 0);
29           i4 : in  STD_LOGIC_VECTOR (31 downto 0);
30           i5 : in  STD_LOGIC_VECTOR (31 downto 0);
31           i6 : in  STD_LOGIC_VECTOR (31 downto 0);
32           i7 : in  STD_LOGIC_VECTOR (31 downto 0);
33           i8 : in  STD_LOGIC_VECTOR (31 downto 0);
34           i9 : in  STD_LOGIC_VECTOR (31 downto 0);
35           i10 : in  STD_LOGIC_VECTOR (31 downto 0);
36           i11 : in  STD_LOGIC_VECTOR (31 downto 0);
37           i12 : in  STD_LOGIC_VECTOR (31 downto 0);
38           i13 : in  STD_LOGIC_VECTOR (31 downto 0);
39           i14 : in  STD_LOGIC_VECTOR (31 downto 0);
40           i15 : in  STD_LOGIC_VECTOR (31 downto 0);
41           i16 : in  STD_LOGIC_VECTOR (31 downto 0);
42           i17 : in  STD_LOGIC_VECTOR (31 downto 0);
43           i18 : in  STD_LOGIC_VECTOR (31 downto 0);
44           i19 : in  STD_LOGIC_VECTOR (31 downto 0);
45           i20 : in  STD_LOGIC_VECTOR (31 downto 0);
46           i21 : in  STD_LOGIC_VECTOR (31 downto 0);
47           i22 : in  STD_LOGIC_VECTOR (31 downto 0);
48           i23 : in  STD_LOGIC_VECTOR (31 downto 0);
49           i24 : in  STD_LOGIC_VECTOR (31 downto 0);
50           i25 : in  STD_LOGIC_VECTOR (31 downto 0);
51           i26 : in  STD_LOGIC_VECTOR (31 downto 0);
52           i27 : in  STD_LOGIC_VECTOR (31 downto 0);
53           i28 : in  STD_LOGIC_VECTOR (31 downto 0);
54           i29 : in  STD_LOGIC_VECTOR (31 downto 0);
55           i30 : in  STD_LOGIC_VECTOR (31 downto 0);
56           i31 : in  STD_LOGIC_VECTOR (31 downto 0);
57           o : out  STD_LOGIC_VECTOR (31 downto 0));
58 end component;
```

```
60  component ALUControlUnit is
61  Port (FunctCode : in STD_LOGIC_VECTOR (5 downto 0);
62        ALUOp: in STD_LOGIC_VECTOR (1 downto 0);
63        ALUFunct : out STD_LOGIC_VECTOR (3 downto 0));
64  end component;
65
66  component muxx2x1 is
67  port ( s: in std_logic;
68                i0: in std_logic_vector (31 downto 0);
69                i1: in std_logic_vector (31 downto 0);
70                o: out std_logic_vector (31 downto 0));
71  end component;
72
73  component INSTRMEMORY is
74  port(
75  LoadIt: in Std_logic;
76  DATA: out STD_LOGIC_VECTOR (31 downto 0);
77  ADDRESS: in STD_LOGIC_VECTOR (31 downto 0);
78  CLK : in STD_LOGIC);
79
80  end component;
81
82
83
84  component ControlUnit is
85  Port
86  ( Opcode:in STD_LOGIC_VECTOR (5 downto 0) ;
87  MemRead:out STD_LOGIC;
88  Memwrite : out STD_LOGIC;
89  RegDst:out STD_LOGIC;
90  Branch:out STD_LOGIC;
91  MemtoReg:out STD_LOGIC;
92  ALUSrc:out STD_LOGIC;
93  ALUOp:out STD_LOGIC_VECTOR (1 downto 0) ;
94  Jump: out STD_LOGIC;
95  RegWrite:out STD_LOGIC);
96  end component;
```

```
99  component RegisterFile is
100 Port (read_sel1 : in STD_LOGIC_VECTOR (4 downto 0);
101       read_sel2 : in STD_LOGIC_VECTOR (4 downto 0);
102       write_sel: in STD_LOGIC_VECTOR (4 downto 0);
103       write_ena : in STD_LOGIC;
104       clk: in STD_LOGIC;
105       write_data : in STD_LOGIC_VECTOR (31 downto 0);
106       data1: out STD_LOGIC_VECTOR (31 downto 0);
107       data2 : out STD_LOGIC_VECTOR (31 downto 0));
108 end component;
109
110
111 component sla2 is
112 port ( i: in STD_LOGIC_VECTOR (31 downto 0);
113        o: out STD_LOGIC_VECTOR (31 downto 0));
114 end component;
115
116 component SignExtend is
117 port (   i: in std_logic_vector (15 downto 0);
118          o: out std_logic_vector (31 downto 0));
119 end component;
120
121 component PC is
122 port    (
123     clk: in std_logic;
124     rst : in std_logic;
125     address: in std_logic_vector (31 downto 0);
126     PC: out std_logic_vector (31 downto 0));
127 end component;
128
129 component plus4 is
130 port ( i: in std_logic_vector (31 downto 0);
131        o: out std_logic_vector (31 downto 0));
132 end component;
```

```
component JA is
Port (   leftin:in STD_LOGIC_VECTOR (31 downto 0) ;
         rightin: in STD_LOGIC_VECTOR (31 downto 0);
         Output :out STD_LOGIC_VECTOR (31 downto 0));
end component;


component mux5bitx1 is
Port ( S: in std_logic;
       i0: in std_logic_vector (4 downto 0);
       i1: in std_logic_vector (4 downto 0);
       o: out std_logic_vector (4 downto 0));
end component;

component datamem is
port( Loadit : in STD_LOGIC;
      INPUT : in STD_LOGIC_VECTOR (31 downto 0);
      OUTPUT : out STD_LOGIC_VECTOR (31 downto 0);
      memoryread: in STD_LOGIC;
      memorywrite: in STD_LOGIC;
      ADDRESS : in STD_LOGIC_VECTOR (31 downto 0);
      CLK : in STD_LOGIC );
end component;

component alu is
 Port ( datal : in  STD_LOGIC_VECTOR (31 downto 0);
           data2 : in  STD_LOGIC_VECTOR (31 downto 0);
           aluop : in  STD_LOGIC_VECTOR (3 downto 0);
           cin : in  STD_LOGIC;
           dataout : out  STD_LOGIC_VECTOR (31 downto 0);
           cflag : out  STD_LOGIC;
           zflag : out  STD_LOGIC;
           oflag : out  STD_LOGIC);
end component;
```

```
signal instructionAddress : std_logic_vector (31 downto 0) := X"00000000";
signal InstrAddressAdd4 : std_logic_vector (31 downto 0)  := X"00000000";

 signal instructionShifted : std_logic_vector (31 downto 0) := X"00000000";
signal jumpInstrAddress : std_logic_vector (31 downto 0) := X"00000000";

signal branchInstAddress : std_logic_vector (31 downto 0) := X"00000000";
signal immedValue16: std_logic_vector (31 downto 0) := X"00000000";
signal branchvalueShifted : std_logic_vector (31 downto 0) := X"00000000";
signal intermediateAddress : std_logic_vector (31 downto 0) := X"00000000";

signal newInstrAddress : std_logic_vector (31 downto 0) := X"00000000";
signal instruction : std_logic_vector (31 downto 0):= X"00000000";

signal MemRead: std_logic := '0';
signal MemWrite: std_logic := '0';
signal RegDst: std_logic := '0';
signal Branch: std_logic := '0';
signal MemtoReg: std_logic := '0';
signal ALUSrc: std_logic := '0';
signal ALUOp: std_logic_vector (1 downto 0) := "00";
signal Jump: std_logic := '0';
signal RegWrite: std_logic := '0';


signal WriteRegister : std_logic_vector (4 downto 0) := "00000";
signal WriteData: std_logic_vector (31 downto 0) := X"00000000";
signal ReadData1: std_logic_vector (31 downto 0) := X"00000000";
signal ReadData2: std_logic_vector (31 downto 0) := X"00000000";

signal ALUFunct : std_logic_vector (3 downto 0) := "0000";
signal ALUZero: std_logic := '0';
signal ALUMuxInput: std_logic_vector (31 downto 0):= X"00000000";
signal ALUOutTemp: std_logic_vector (31 downto 0):= X"00000000";

signal DataMemoutTemp: std_logic_vector (31 downto 0) := X"00000000";

signal BranchingSignal : std_logic := '0';
```

```
195
196  signal WriteRegister : std_logic_vector (4 downto 0) := "00000";
197  signal WriteData: std_logic_vector (31 downto 0) := X"00000000";
198  signal ReadData1: std_logic_vector (31 downto 0) := X"00000000";
199  signal ReadData2: std_logic_vector (31 downto 0) := X"00000000";
200
201  signal ALUFunct : std_logic_vector (3 downto 0) := "0000";
202  signal ALUZero: std_logic := '0';
203  signal ALUMuxInput: std_logic_vector (31 downto 0):= X"00000000";
204  signal ALUOutTemp: std_logic_vector (31 downto 0):= X"00000000";
205
206  signal DataMemoutTemp: std_logic_vector (31 downto 0) := X"00000000";
207
208  signal BranchingSignal : std_logic := '0';
209
210  end MyPackage;
211
212
213  package body mypackage is
214
215  ---- Example 1
216  --  function <function_name>  (signal <signal_name> : in <type_declaration>  ) return <type_declaration> is
217  --    variable <variable_name>     : <type_declaration>;
218  --  begin
219  --    <variable_name> := <signal_name> xor <signal_name>;
220  --    return <variable_name>;
221  --  end <function_name>;
222
223  ---- Example 2
224  --  function <function_name>  (signal <signal_name> : in <type_declaration>;
225  --                         signal <signal_name>   : in <type_declaration>  ) return <type_declaration> is
226  --  begin
227  --    if (<signal_name> = '1') then
228  --      return <signal_name>;
229  --    else
230  --      return 'Z';
231  --    end if;
232  --  end <function_name>;
233
```

2.1 MIPS CPU

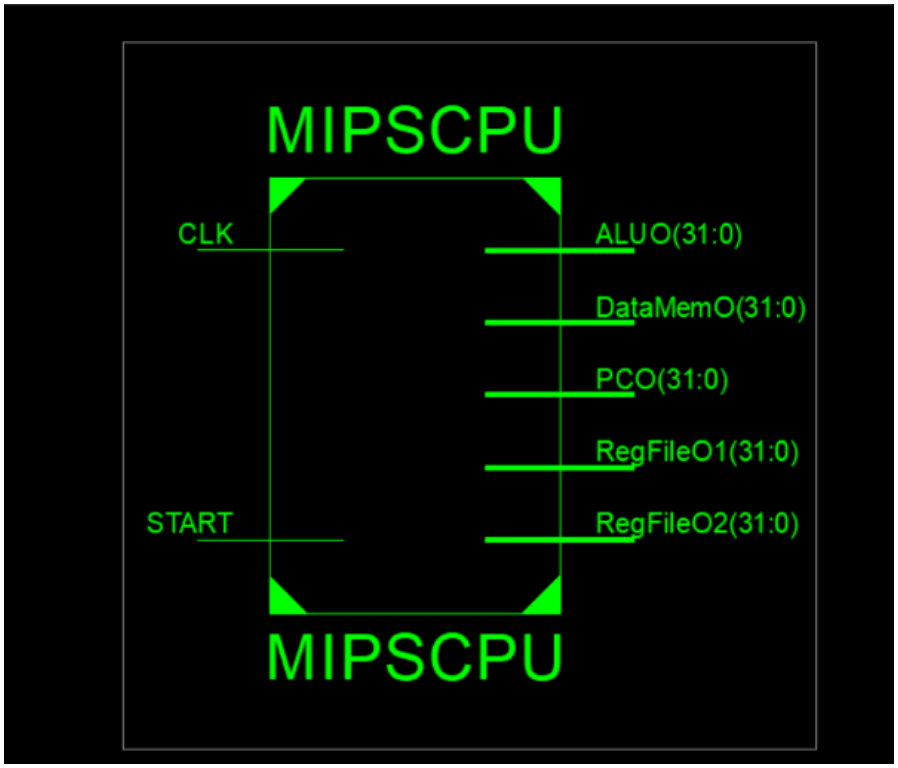
Finally, we created a main module called "MIPSCPU" and included "MIPSPackage," where the main module will be finished by connecting every previous module in accordance with the connections shown in the provided CPU diagram. The connections between the modules were then made using the previously created signals and components found in "MIPSPackage," similar to the CPU diagram. "open" was given to any output that was found in a component but wasn't used in the main module. Finally, we assigned the outputs of the "MIPSCPU" main module to their corresponding signals.

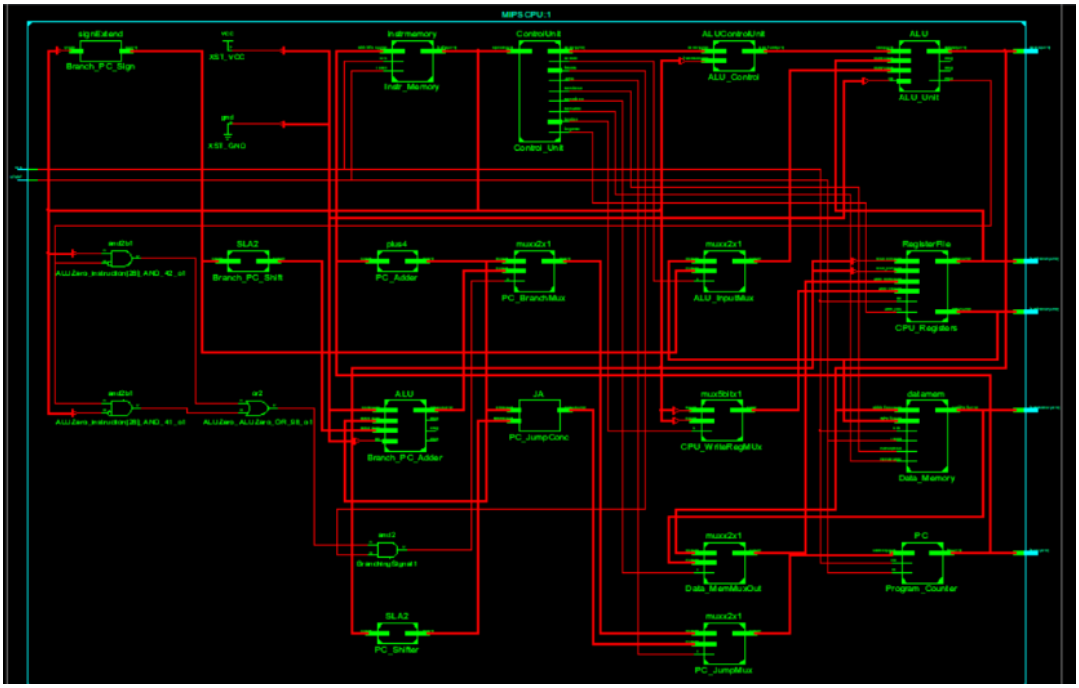
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.MyPackage.all;

entity MIPSCPU is
   Port (START: in STD_LOGIC;
         CLK: in STD_LOGIC;
         RegFileO1 : out STD_LOGIC_VECTOR (31 downto 0);
         RegFileO2 : out STD_LOGIC_VECTOR (31 downto 0);
         ALUO: out STD_LOGIC_VECTOR (31 downto 0);
         PCO : out STD_LOGIC_VECTOR (31 downto 0);
         DataMemO: out STD_LOGIC_VECTOR (31 downto 0)
         );
end MIPSCPU;

architecture Behavioral of MIPSCPU is
begin

-- PC operations

   PC_Adder: plus4              port map (i => instructionAddress, o => InstrAddressAdd4);

-- branches

   Branch_PC_Sign: SignExtend    port map (i => instruction (15 downto 0), o => immedValue16);

   Branch_PC_Shift : sla2        port map (i => immedValue16, o => branchValueShifted);

   Branch_PC_Adder : ALU         port map (data1 => InstrAddressAdd4, data2 => branchvalueShifted, aluop => "0010", cin =>'0'
                                          ,dataout => branchInstAddress, cflag => open, zflag =>open, oflag =>open);

 BranchingSignal <= Branch AND ((ALUZero AND NOT (instruction (26))) OR (NOT (ALUZero) AND instruction (26)));
```

```

   PC_BranchMux : muxx2x1       port map (S => BranchingSignal, i0 => InstrAddressAdd4, i1 => branchInstAddress, O => intermediateAddress);

-- jumps

   PC_Shifter: sla2             port map (I => instruction, O => instructionShifted);

   PC_JumpConc: JA              port map (leftin => InstrAddressAdd4, rightin => instructionShifted,
                                          Output => jumpInstrAddress);

   PC_JumpMux:  muxx2x1         port map (S => Jump, i0 => intermediateAddress, i1 => JumpInstrAddress,
                                          o => newInstrAddress);
-- PC
   Program_Counter : PC         port map (CLK => CLK, RST => START, Address => newInstrAddress, PC => instructionAddress);


--Instruction memory
Instr_Memory: INSTRMEMORY       port map (LoadIt => START, DATA => instruction, ADDRESS => instructionAddress, CLK => CLK);


--Main Control Unit
Control_Unit: ControlUnit       port map(OpCode => instruction (31 downto 26), MemRead => MemRead, MemWrite =>
                                         MemWrite, RegDst =>RegDst, Branch=> Branch, MemtoReg => MemtoReg, ALUSrc => ALUSrc,
                                         ALUOp => ALUOp,Jump => Jump, RegWrite => RegWrite);

--RegisterFile
CPU_WriteRegMUx : Mux5bitx1     port map (S => RegDst, i0 => instruction (20 downto 16), i1 => instruction (15 downto 11),
                                          o => WriteRegister);
```

```
CPU_Registers: RegisterFile     port map (read_sel1 => instruction (25 downto 21), read_sel2 => instruction (20 downto 16),
                                          write_sel => WriteRegister, write_ena => RegWrite, clk => CLK,
                                          write_data => WriteData, data1 => ReadData1, data2 => ReadData2);


--ALU & ALU Control unit
ALU_Control : ALUControlUnit    port map (FunctCode => instruction (5 downto 0), ALUOp => ALUOp, ALUFunct => ALUFunct);

ALU_InputMux: muxx2x1           port map (s => ALUSrc, i0 => ReadData2, i1 => immedValue16,
                                          o => ALUMuxInput);

ALU_Unit: ALU                   port map (data1 => ReadData1, data2 => ALUMuxInput,
                                          aluop => ALUFunct, cin => '0', dataout => ALUOutTemp, cflag => open,
                                          zflag => ALUZero, oflag =>open );

--Data memory

Data_Memory:DATAMEM             port map (LoadIt => START, INPUT => ReadData2,
                                          OUTPUT => DataMemOutTemp, memoryread => MemRead, MEMORYWRITE => MemWrite,
                                          ADDRESS => ALUOutTemp, CLK => CLK);


Data_MemMuxOut : muxx2x1        port map (s => MemtoReg, i0 => ALUOutTemp, i1 => DataMemOutTemp, o => WriteData);


--Outputs--

RegFileO1 <= ReadData1;
RegFileO2 <= ReadData2;
ALUO <= ALUOutTemp;
PCO <= instructionAddress;
DataMemO <= DataMemOutTemp;
```

2.0 CPU RTL Schematic

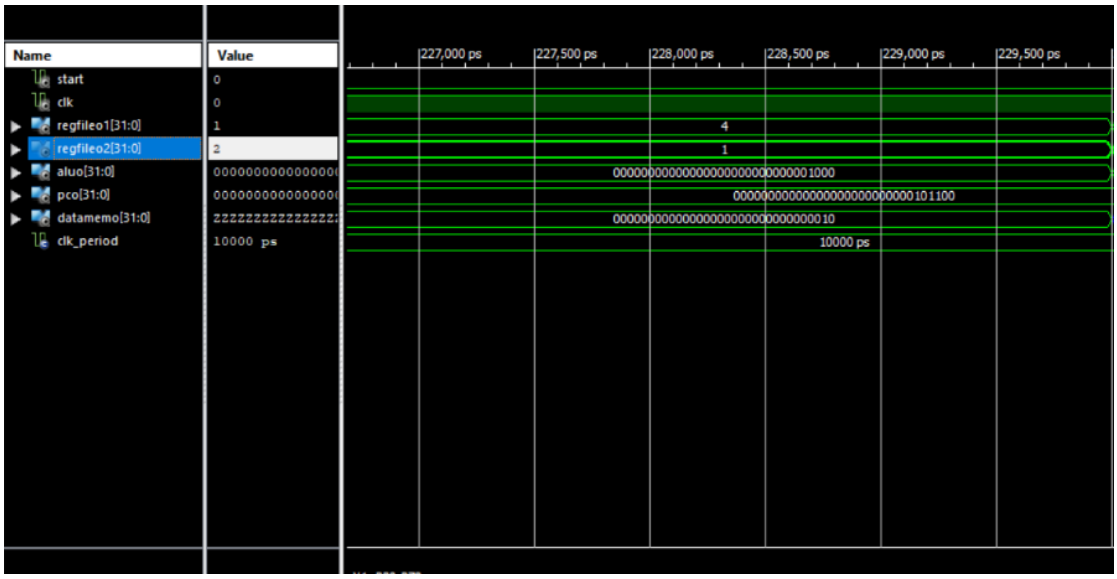


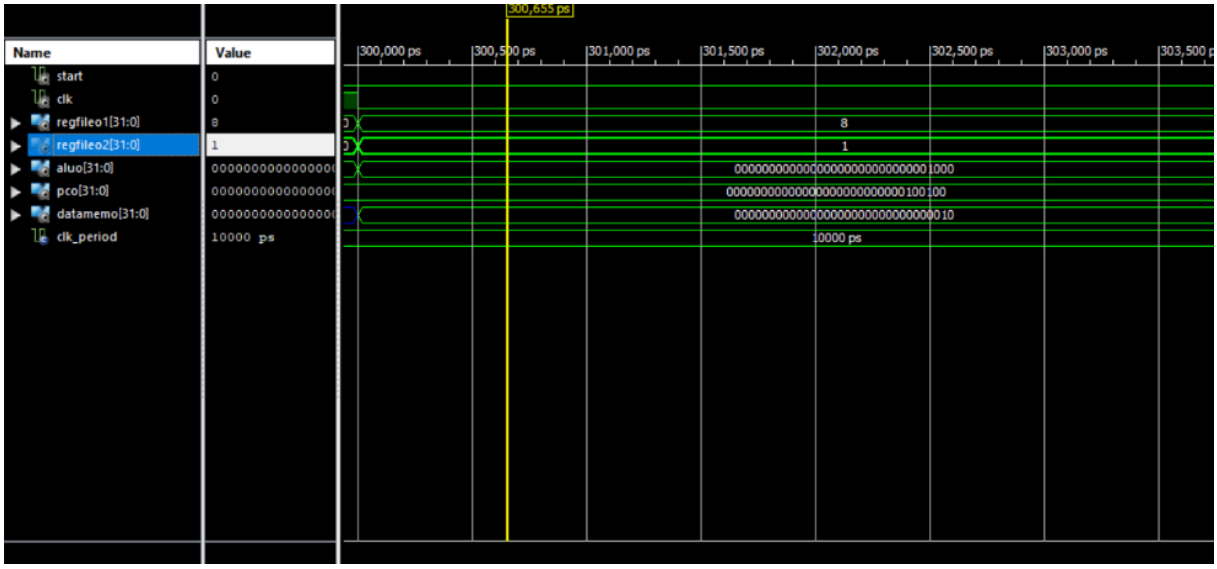


3.0 CPU Test Bench

```
1
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.ALL;
4
5
6  ENTITY mipsCPUtest IS
7  END mipsCPUtest;
8
9  ARCHITECTURE behavior OF mipsCPUtest IS
10
11
12     COMPONENT MIPSCPU
13     PORT(
14          START : IN  std_logic;
15          CLK : IN  std_logic;
16          RegFileO1 : OUT  std_logic_vector(31 downto 0);
17          RegFileO2 : OUT  std_logic_vector(31 downto 0);
18          ALUO : OUT  std_logic_vector(31 downto 0);
19          PCO : OUT  std_logic_vector(31 downto 0);
20          DataMemO : OUT  std_logic_vector(31 downto 0)
21
22         );
23     END COMPONENT;
24
25
26    --Inputs
27    signal START : std_logic := '0';
28    signal CLK : std_logic := '0';
29
30    --Outputs
31    signal RegFileO1 : std_logic_vector(31 downto 0);
32    signal RegFileO2 : std_logic_vector(31 downto 0);
33    signal ALUO : std_logic_vector(31 downto 0);
34    signal PCO : std_logic_vector(31 downto 0);
35    signal DataMemO : std_logic_vector(31 downto 0);
36
37
38    -- Clock period definitions
39    constant CLK_period : time := 10 ns;
40
41 BEGIN
42
43    -- Instantiate the Unit Under Test (UUT)
44    uut: MIPSCPU PORT MAP (
45           START => START,
46           CLK => CLK,
47           RegFileO1 => RegFileO1,
48           RegFileO2 => RegFileO2,
49           ALUO => ALUO,
50           PCO => PCO,
51           DataMemO => DataMemO
52         );
53
54    -- Clock process definitions
55    CLK_process :process
56    begin
57       CLK <= '0';
58       wait for CLK_period/2;
59       CLK <= '1';
60       wait for CLK_period/2;
61    end process;
62
63
64     -- Stimulus process
65     stim_proc: process
66     begin
67        start <= '1';
68        wait for 30 ns;
69        start <='0';
70        wait for 30 ns;
71
72        wait;
73     end process;
74
75 END;
76
```

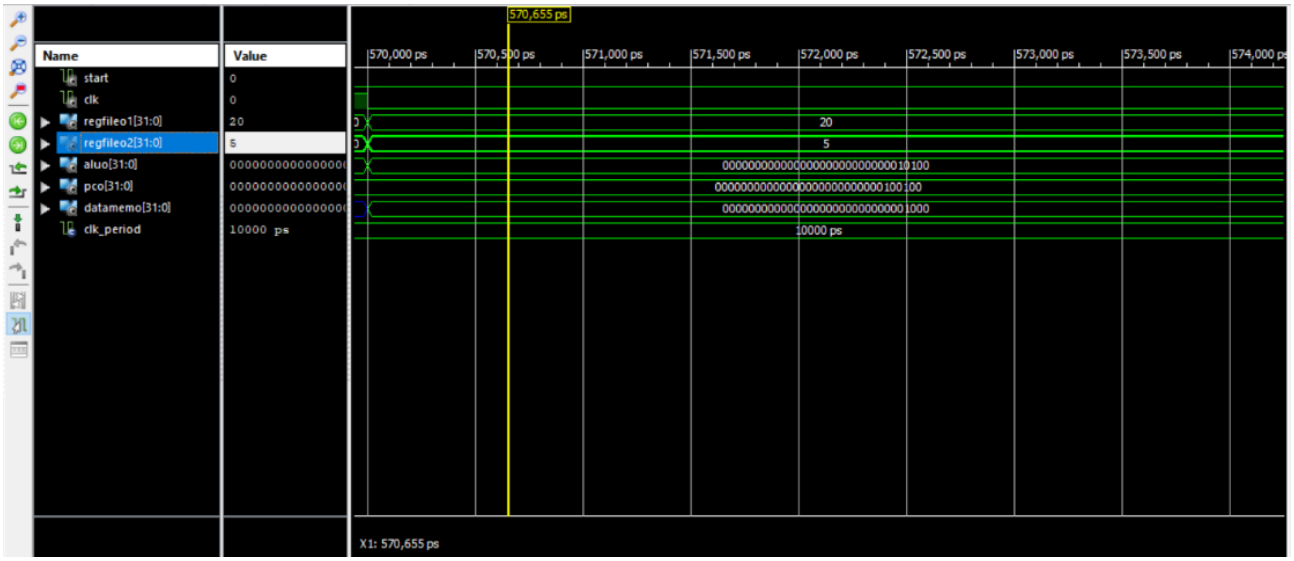
4.0 CPU Simulation

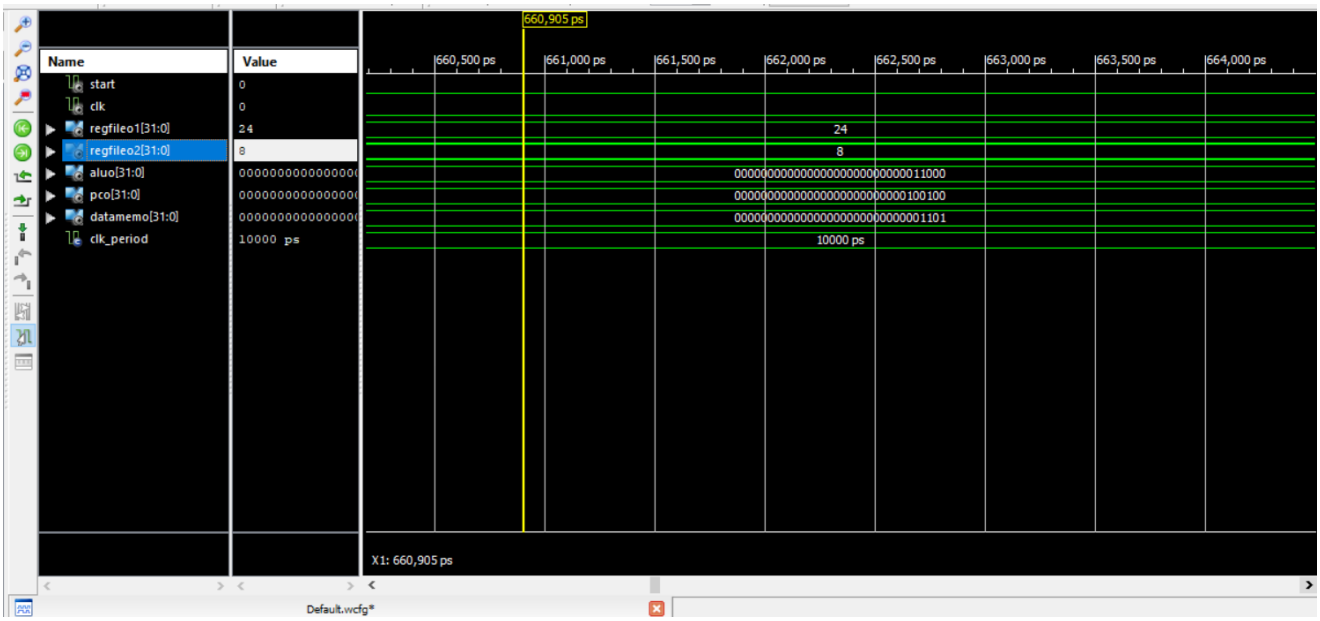




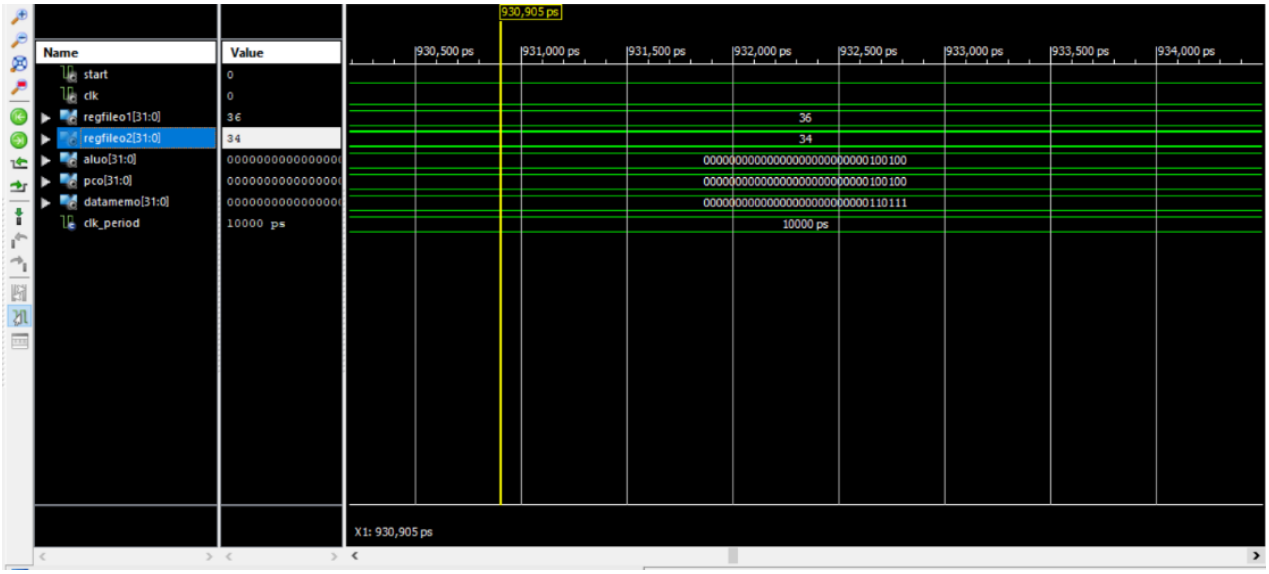

390,905 ps
Name
Value
start
clk
regfileo1[31:0]
regfileo2[31:0]
aluo[31:0]
pco[31:0]
datamemo[31:0]
clk_period
0
0
12
2
10000 ps
390,500 ps
391,000 ps
391,500 ps
392,000 ps
392,500 ps
393,000 ps
393,500 ps
394,000 ps
12
2
10000 ps
X1: 390,905 ps

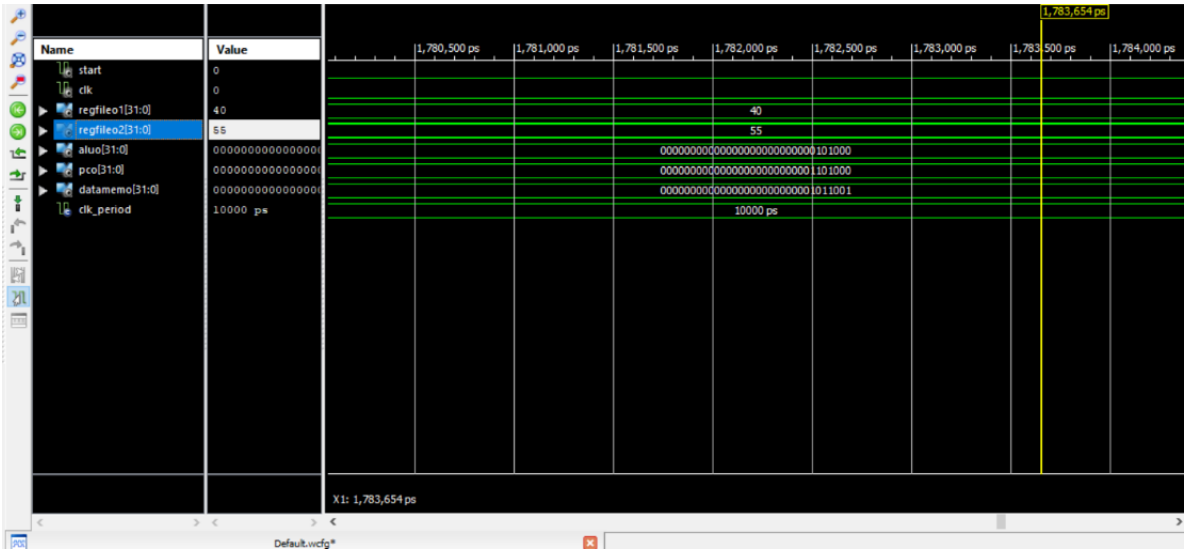

480,655 ps
Name
Value
start
clk
regfileo1[31:0]
regfileo2[31:0]
aluo[31:0]
pco[31:0]
datamemo[31:0]
clk_period
0
0
16
3
10000 ps
480,000 ps
480,500 ps
481,000 ps
481,500 ps
482,000 ps
482,500 ps
483,000 ps
483,500 ps
484,000 ps
16
3
10000 ps
X1: 480,655 ps

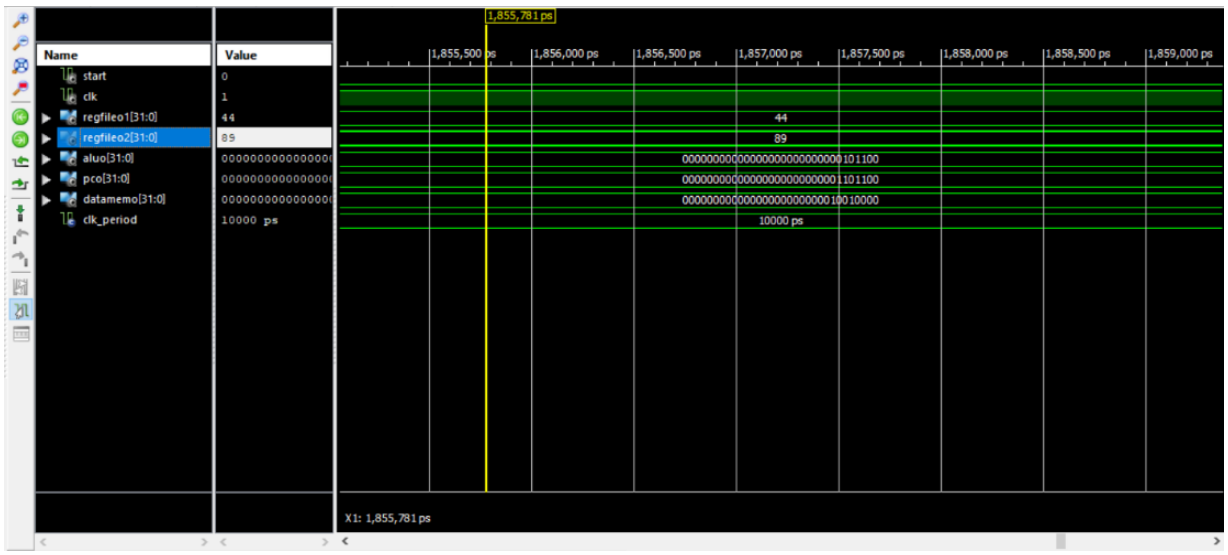

570,655 ps
Name
Value
start
clk
regfileo1[31:0]
regfileo2[31:0]
aluo[31:0]
pco[31:0]
datamemo[31:0]
clk_period
0
0
20
5
10000 ps
570,000 ps
570,500 ps
571,000 ps
571,500 ps
572,000 ps
572,500 ps
573,000 ps
573,500 ps
574,000 ps
20
5
10000 ps
X1: 570,655 ps

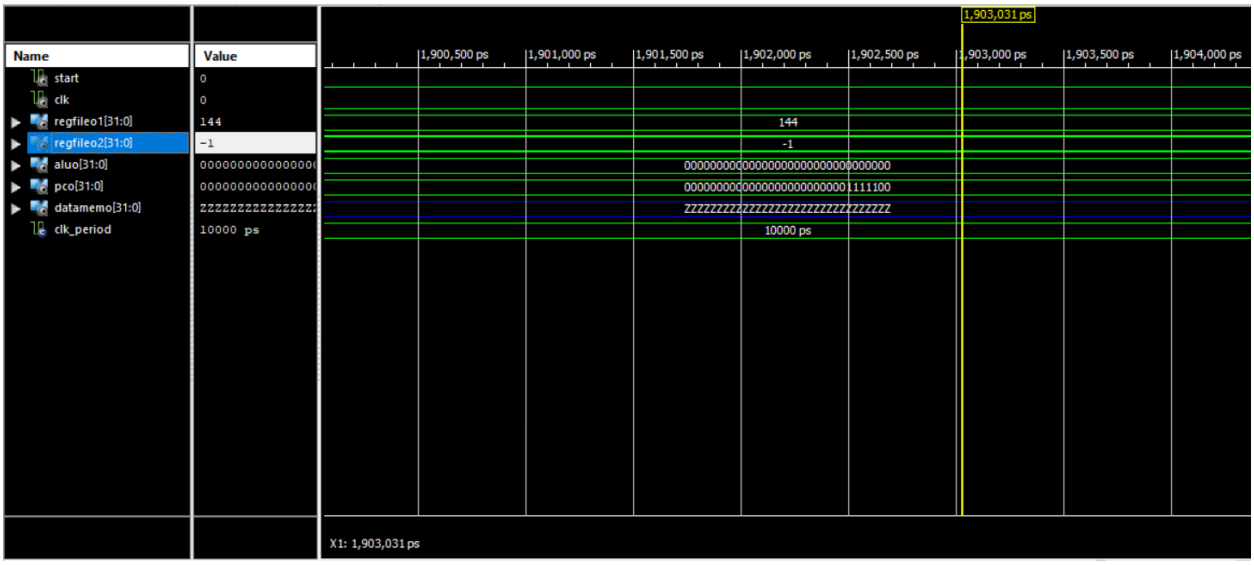


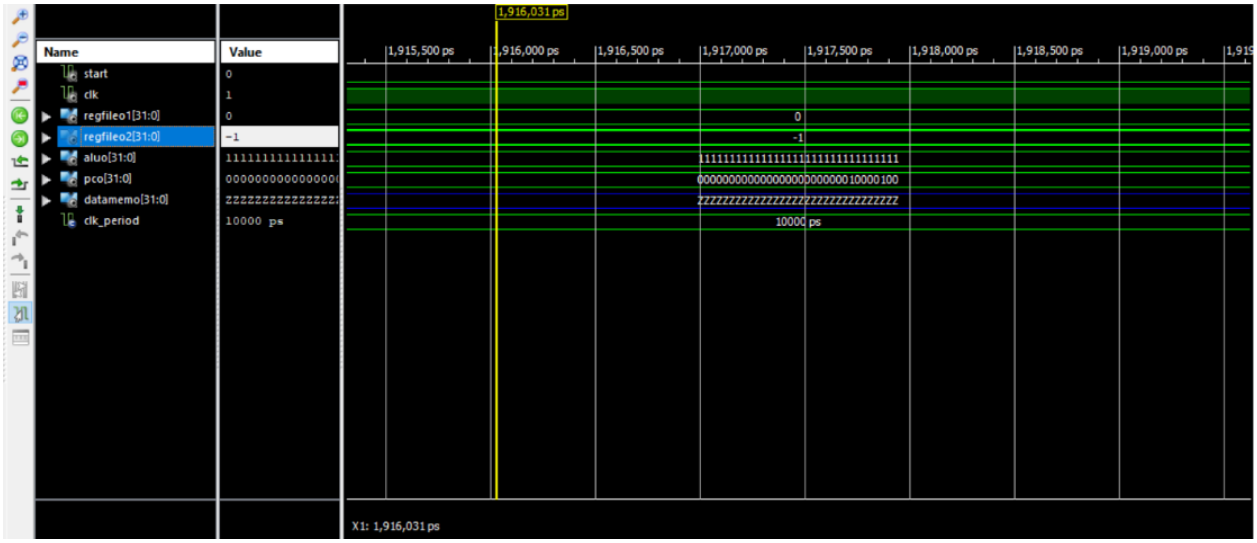
Default.wcfg*

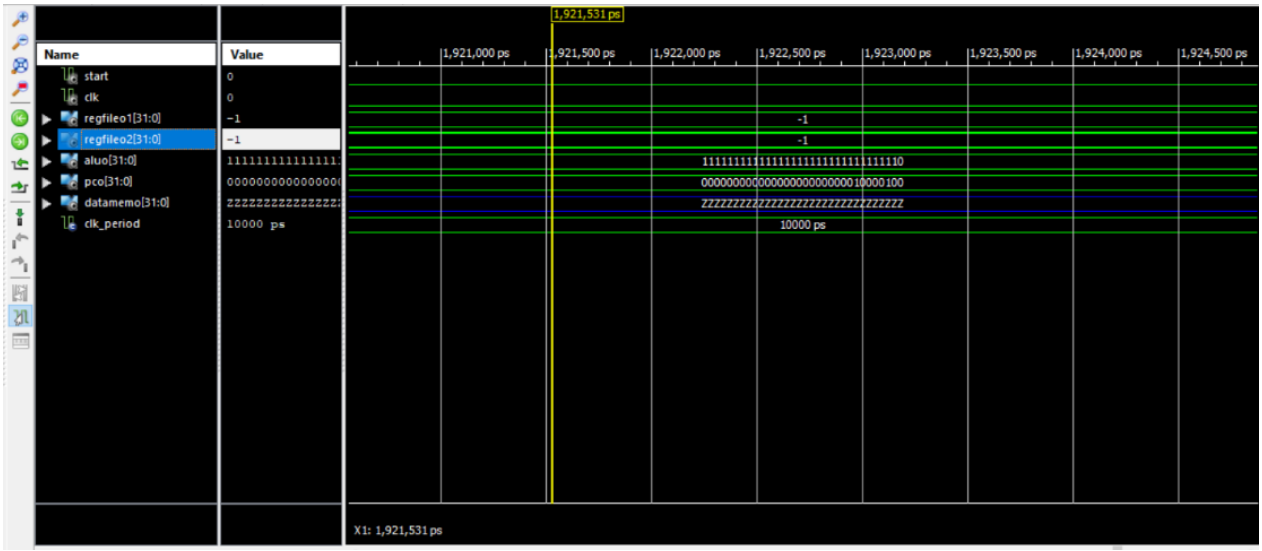

930,905 ps
Name
Value
start
clk
regfileo1[31:0]
regfileo2[31:0]
aluo[31:0]
pco[31:0]
datamemo[31:0]
clk_period
0
0
36
34
10000 ps
930,500 ps
931,000 ps
931,500 ps
932,000 ps
932,500 ps
933,000 ps
933,500 ps
934,000 ps
36
34
10000 ps
X1: 930,905 ps


1,783,654 ps
Name
Value
start
clk
regfileo1[31:0]
regfileo2[31:0]
aluo[31:0]
pco[31:0]
datamemo[31:0]
clk_period
0
0
40
55
10000 ps
1,780,500 ps
1,781,000 ps
1,781,500 ps
1,782,000 ps
1,782,500 ps
1,783,000 ps
1,784,000 ps
40
55
10000 ps
X1: 1,783,654 ps
Default.wcfg*


1,855,781 ps
Name
Value
start
clk
regfileo1[31:0]
regfileo2[31:0]
aluo[31:0]
pco[31:0]
datamemo[31:0]
clk_period
0
1
44
89
10000 ps
1,855,500 ps
1,856,000 ps
1,856,500 ps
1,857,000 ps
1,857,500 ps
1,858,000 ps
1,858,500 ps
1,859,000 ps
44
89
10000 ps
X1: 1,855,781 ps


Name
Value
start
clk
regfileo1[31:0]
regfileo2[31:0]
aluo[31:0]
pco[31:0]
datamemo[31:0]
clk_period
0
0
144
-1
10000 ps
1,903,031 ps
1,900,500 ps
1,901,000 ps
1,901,500 ps
1,902,000 ps
1,902,500 ps
1,903,000 ps
1,903,500 ps
1,904,000 ps
144
-1
10000 ps
X1: 1,903,031 ps


Name
Value
start
clk
regfileo1[31:0]
regfileo2[31:0]
aluo[31:0]
pco[31:0]
datamemo[31:0]
clk_period
0
1
0
-1
10000 ps
1,916,031 ps
1,915,500 ps
1,916,000 ps
1,916,500 ps
1,917,000 ps
1,917,500 ps
1,918,000 ps
1,918,500 ps
1,919,000 ps
0
-1
10000 ps
X1: 1,916,031 ps


Name
Value
start
clk
regfileo1[31:0]
regfileo2[31:0]
aluo[31:0]
pco[31:0]
datamemo[31:0]
clk_period
0
0
-1
-1
10000 ps
1,921,531 ps
1,921,000 ps
1,921,500 ps
1,922,000 ps
1,922,500 ps
1,923,000 ps
1,923,500 ps
1,924,000 ps
1,924,500 ps
-1
-1
10000 ps
X1: 1,921,531 ps

5.0 Our Contribution

All of us contributed and worked equally on each component through several online meetings together. Jumana Yasser 20% , Nadine Hisham 20%, Habiba Yasser 20%, Hamsa Ahmad 20%, Salma Nasreldin 20%.