CSE 112: Computer Organization and Architecture

Major Task Report

Made By:

Ahmed Wael Samir Abdelmegied 20P7271 (Group Leader)

Ahmed Hossam Moussa Sakr 20P1009

Tamer Ihab Mohamed Abdelwahab 20P5567

Omar Ahmed Mohamed Gamaleldin Swelam 21P0405

Mohamed Tarek Mohamed Abdalla Ahmed Khafagy 20P6211

Phase 1

The MIPS Register Description

In order to implement the MIPS Register File, we needed to implement a normal register module, a 5-32 decoder and a 32x1 multiplexer, all to be used as components to implement the main module required “RegisterFile”. Firstly, we implemented a normal register in a module called “RegDef”.

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

The “RegDef” register is the regular implementation of a register that takes the following inputs: The data input to the register, a reset, a clock, load and increment. It produces only one output. In case the reset = ‘1’, the output is filled with zeros. However, when the reset is equal to ‘0’ and there is a clock pulse received, at the falling edge of the clock, if the load = ‘1’, the input value to the register is loaded and transferred to the output, however, if the increment = ‘1’, the output is incremented by 1.

Next, we implemented the 5-32 decoder called “The Decoder”. Its entity contains two input ports which are called “input” and “enable”, and one output port called “output”. The decoder job will be taking 5-bit input and converting it to 32-bit output, this will be used in writing data to a register, where the 5-bits will be the address of the register for the data to be written in, and the 32-output will be connected to 32 registers of the MIPS Register File as enable lines, whichever one is activated during the writing process, the equivalent register will be selected and the data will be written to it.

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

entity TheDecoder is
    Port (
          input : in STD_LOGIC_VECTOR (4 downto 0);
          enable : in STD_LOGIC;
          output : out STD_LOGIC_VECTOR (31 downto 0));
end TheDecoder;

architecture Behavioral of TheDecoder is

begin

output <= (others => 'Z') when enable = '0' else
          "00000000000000000000000000000001" when input = "00000" else
          "00000000000000000000000000000010" when input = "00001" else
          "00000000000000000000000000000100" when input = "00010" else
          "00000000000000000000000000001000" when input = "00011" else
          "00000000000000000000000000010000" when input = "00100" else
          "00000000000000000000000000100000" when input = "00101" else
          "00000000000000000000000001000000" when input = "00110" else
          "00000000000000000000000010000000" when input = "00111" else
          "00000000000000000000000100000000" when input = "01000" else
          "00000000000000000000001000000000" when input = "01001" else
          "00000000000000000000010000000000" when input = "01010" else
          "00000000000000000000100000000000" when input = "01011" else
          "00000000000000000001000000000000" when input = "01100" else
          "00000000000000000010000000000000" when input = "01101" else
          "00000000000000000100000000000000" when input = "01110" else
          "00000000000000001000000000000000" when input = "01111" else
          "00000000000000010000000000000000" when input = "10000" else
          "00000000000000100000000000000000" when input = "10001" else
          "00000000000001000000000000000000" when input = "10010" else
          "00000000000010000000000000000000" when input = "10011" else
          "00000000000100000000000000000000" when input = "10100" else
          "00000000001000000000000000000000" when input = "10101" else
          "00000000010000000000000000000000" when input = "10110" else
          "00000000100000000000000000000000" when input = "10111" else
          "00000001000000000000000000000000" when input = "11000" else
          "00000010000000000000000000000000" when input = "11001" else
          "00000100000000000000000000000000" when input = "11010" else
          "00001000000000000000000000000000" when input = "11011" else
          "00010000000000000000000000000000" when input = "11100" else
          "00100000000000000000000000000000" when input = "11101" else
          "01000000000000000000000000000000" when input = "11110" else
          "10000000000000000000000000000000" when input = "11111" else
          (others => 'X');

end Behavioral;
```

Last but not least, 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.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity Mux32x1 is
5      Port ( S : in  STD_LOGIC_VECTOR (4 downto 0);
6             i0 : in  STD_LOGIC_VECTOR (31 downto 0);
7             i1 : in  STD_LOGIC_VECTOR (31 downto 0);
8             i2 : in  STD_LOGIC_VECTOR (31 downto 0);
9             i3 : in  STD_LOGIC_VECTOR (31 downto 0);
10            i4 : in  STD_LOGIC_VECTOR (31 downto 0);
11            i5 : in  STD_LOGIC_VECTOR (31 downto 0);
12            i6 : in  STD_LOGIC_VECTOR (31 downto 0);
13            i7 : in  STD_LOGIC_VECTOR (31 downto 0);
14            i8 : in  STD_LOGIC_VECTOR (31 downto 0);
15            i9 : in  STD_LOGIC_VECTOR (31 downto 0);
16            i10: in  STD_LOGIC_VECTOR (31 downto 0);
17            i11: in  STD_LOGIC_VECTOR (31 downto 0);
18            i12: in  STD_LOGIC_VECTOR (31 downto 0);
19            i13: in  STD_LOGIC_VECTOR (31 downto 0);
20            i14: in  STD_LOGIC_VECTOR (31 downto 0);
21            i15: in  STD_LOGIC_VECTOR (31 downto 0);
22            i16: in  STD_LOGIC_VECTOR (31 downto 0);
23            i17: in  STD_LOGIC_VECTOR (31 downto 0);
24            i18: in  STD_LOGIC_VECTOR (31 downto 0);
25            i19: in  STD_LOGIC_VECTOR (31 downto 0);
26            i20: in  STD_LOGIC_VECTOR (31 downto 0);
27            i21: in  STD_LOGIC_VECTOR (31 downto 0);
28            i22: in  STD_LOGIC_VECTOR (31 downto 0);
29            i23: in  STD_LOGIC_VECTOR (31 downto 0);
30            i24: in  STD_LOGIC_VECTOR (31 downto 0);
31            i25: in  STD_LOGIC_VECTOR (31 downto 0);
32            i26: in  STD_LOGIC_VECTOR (31 downto 0);
33            i27: in  STD_LOGIC_VECTOR (31 downto 0);
34            i28: in  STD_LOGIC_VECTOR (31 downto 0);
35            i29: in  STD_LOGIC_VECTOR (31 downto 0);
36            i30: in  STD_LOGIC_VECTOR (31 downto 0);
37            i31: in  STD_LOGIC_VECTOR (31 downto 0);
38            output : out  STD_LOGIC_VECTOR (31 downto 0));
39 end Mux32x1;
```

```
40
41  architecture Behavioral of Mux32x1 is
42
43  begin
44
45    output <= i0  when S = "00000" else
46              i1  when S = "00001" else
47              i2  when S = "00010" else
48              i3  when S = "00011" else
49              i4  when S = "00100" else
50              i5  when S = "00101" else
51              i6  when S = "00110" else
52              i7  when S = "00111" else
53              i8  when S = "01000" else
54              i9  when S = "01001" else
55              i10 when S = "01010" else
56              i11 when S = "01011" else
57              i12 when S = "01100" else
58              i13 when S = "01101" else
59              i14 when S = "01110" else
60              i15 when S = "01111" else
61              i16 when S = "10000" else
62              i17 when S = "10001" else
63              i18 when S = "10010" else
64              i19 when S = "10011" else
65              i20 when S = "10100" else
66              i21 when S = "10101" else
67              i22 when S = "10110" else
68              i23 when S = "10111" else
69              i24 when S = "11000" else
70              i25 when S = "11001" else
71              i26 when S = "11010" else
72              i27 when S = "11011" else
73              i28 when S = "11100" else
74              i29 when S = "11101" else
75              i30 when S = "11110" else
76              i31 when S = "11111" else
77              (others => 'Z');
78
79  end Behavioral;
80
```

Consequently, we placed all the previous modules in a package called "RegPackage" to be used in implementing the main module "RegisterFile".

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3   use work.RegPackage.all;
4
5   entity RegisterFile is
6       Port ( read_sel1 : in  STD_LOGIC_VECTOR (4 downto 0);
7              read_sel2 : in  STD_LOGIC_VECTOR (4 downto 0);
8              write_sel : in  STD_LOGIC_VECTOR (4 downto 0);
9              write_ena : in  STD_LOGIC;
10             clk       : in  STD_LOGIC;
11             write_data: in  STD_LOGIC_VECTOR (31 downto 0);
12             data1     : out STD_LOGIC_VECTOR (31 downto 0);
13             data2     : out STD_LOGIC_VECTOR (31 downto 0));
14  end RegisterFile;
```

At the beginning, we included the previously created package "RegPackage" and declared the ports of the "RegisterFile" as described in the major task pdf.

Secondly, we declared a signal called 'L' with size of 32 bits to be used as the load enables of each register starting from L(0) to L(31) to select the required register in the writing process. Also, we declared 32 signals starting from "out0" till "out31" in order to hold the output of each register of the 32 registers to be used later on as the inputs of the 32x1 multiplexer to choose the required output in the reading process.

```
17
18  signal L,out0,out1,out2,out3,out4,out5,out6,out7,out8,out9,out10,

,out10,out11,out12,out13,out14,out15,out16,out17,out18,out19,out20,out21,out22,

,out22,out23,out24,out25,out26,out27,out28,out29,out30,out31: STD_LOGIC_VECTOR (31 downto 0);
```

Moreover, then we created onedecoder calling “TheDecoder” previous implementation, mapping the write_sel to the input, the write_ena to the enable and Signal ‘L’ to the output of the decoder.

```
22    onedecoder : TheDecoder port map (write_sel,write_ena,L);
```

After that, we created 32 Registers using the “RegDef” module and mapped the write_data to the input of the registers except for “Reg0” to have 32 zeros to the input because it will represent $zero,then, ‘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.

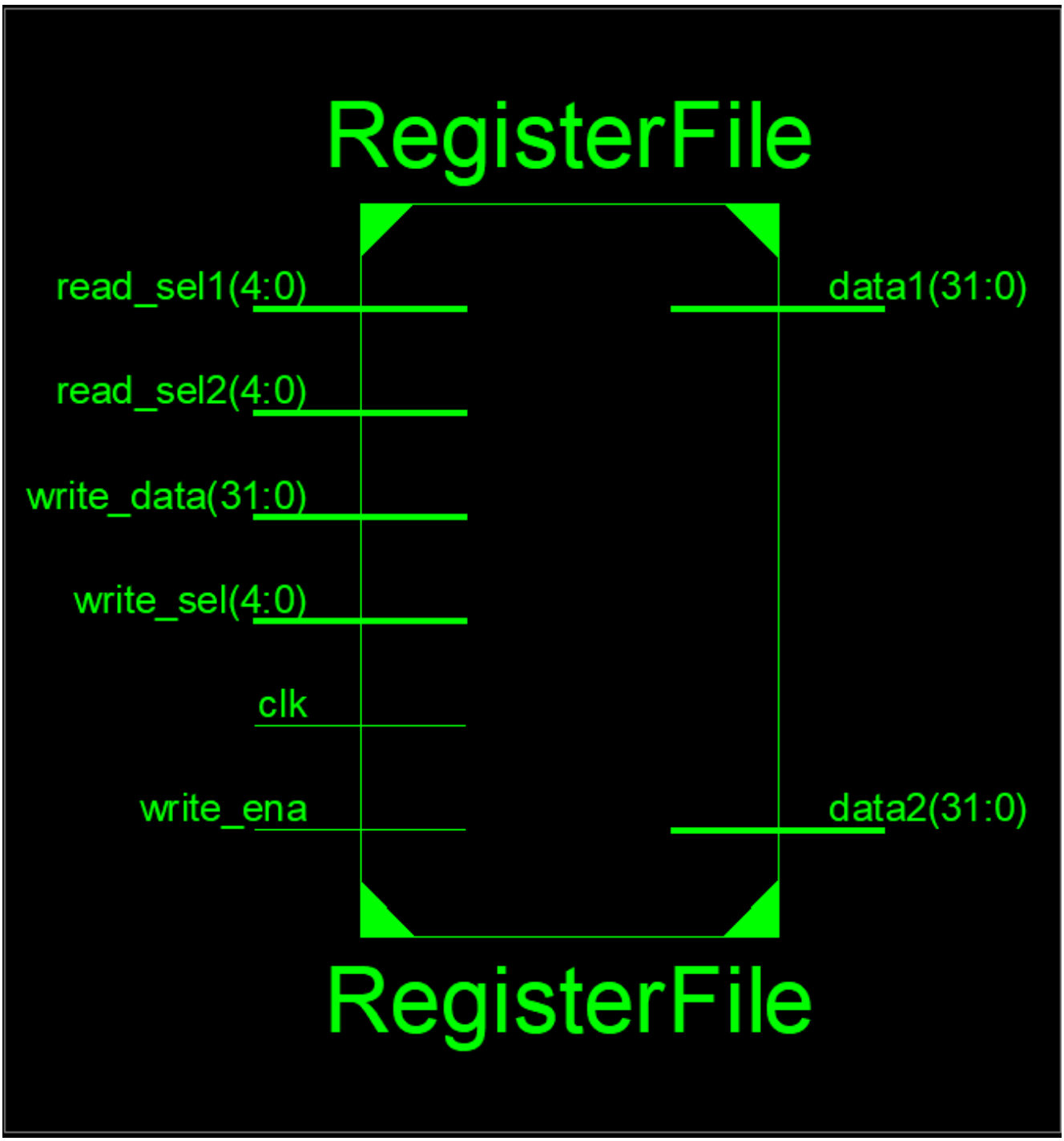
```
23
24    Reg0    : RegDef    port map (x"00000000",'0',clk,L(0),'0',out0 ); --$zero
25    Reg1    : RegDef    port map (write_data,'0',clk,L(1),'0', out1  );  --$at
26    Reg2    : RegDef    port map (write_data,'0',clk,L(2),'0', out2  ); --$v0
27    Reg3    : RegDef    port map (write_data,'0',clk,L(3),'0', out3  );--$v1
28    Reg4    : RegDef    port map (write_data,'0',clk,L(4),'0', out4  );--$a0
29    Reg5    : RegDef    port map (write_data,'0',clk,L(5),'0', out5  );--$a1
30    Reg6    : RegDef    port map (write_data,'0',clk,L(6),'0', out6  );--$a2
31    Reg7    : RegDef    port map (write_data,'0',clk,L(7),'0', out7  );--$a3
32    Reg8    : RegDef    port map (write_data,'0',clk,L(8),'0', out8  );--$t0
33    Reg9    : RegDef    port map (write_data,'0',clk,L(9),'0', out9  );--$t1
34    Reg10   : RegDef    port map (write_data,'0',clk,L(10),'0',out10 );--$t2
35    Reg11   : RegDef    port map (write_data,'0',clk,L(11),'0',out11 );--$t3
36    Reg12   : RegDef    port map (write_data,'0',clk,L(12),'0',out12 );--$t4
37    Reg13   : RegDef    port map (write_data,'0',clk,L(13),'0',out13 );--$t5
38    Reg14   : RegDef    port map (write_data,'0',clk,L(14),'0',out14 );--$t6
39    Reg15   : RegDef    port map (write_data,'0',clk,L(15),'0',out15 );--$t7
40    Reg16   : RegDef    port map (write_data,'0',clk,L(16),'0',out16 );--$s0
41    Reg17   : RegDef    port map (write_data,'0',clk,L(17),'0',out17 );--$s1
42    Reg18   : RegDef    port map (write_data,'0',clk,L(18),'0',out18 );--$s2
43    Reg19   : RegDef    port map (write_data,'0',clk,L(19),'0',out19 );--$s3
44    Reg20   : RegDef    port map (write_data,'0',clk,L(20),'0',out20 );--$s4
45    Reg21   : RegDef    port map (write_data,'0',clk,L(21),'0',out21 );--$s5
46    Reg22   : RegDef    port map (write_data,'0',clk,L(22),'0',out22 );--$s6
47    Reg23   : RegDef    port map (write_data,'0',clk,L(23),'0',out23 );--$s7
48    Reg24   : RegDef    port map (write_data,'0',clk,L(24),'0',out24 );--$t8
49    Reg25   : RegDef    port map (write_data,'0',clk,L(25),'0',out25 );--$t9
50    Reg26   : RegDef    port map (write_data,'0',clk,L(26),'0',out26 );--$k0
51    Reg27   : RegDef    port map (write_data,'0',clk,L(27),'0',out27 );--$k1
52    Reg28   : RegDef    port map (write_data,'0',clk,L(28),'0',out28 );--$gp
53    Reg29   : RegDef    port map (write_data,'0',clk,L(29),'0',out29 );--$sp
54    Reg30   : RegDef    port map (write_data,'0',clk,L(30),'0',out30 );--$fp
55    Reg31   : RegDef    port map (write_data,'0',clk,L(31),'0',out31 );--$ra
```

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, the selector 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.

```
56
57      FirstMux  : Mux32x1 port map (read_sel1,out0,out1,out2,out3,out4,out5,out6,
58
59      SecondMux : Mux32x1 port map (read_sel2,out0,out1,out2,out3,out4,out5,out6,
```

```
,out7,out8,out9,out10,out11,out12,out13,out14,out15,out16,out17,out18,out19,
,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,data1);
out20,out21,out22,out23,out24,out25,out26,out27,out28,out29,out30,out31,data2);
```

The MIPS Register RTL Schematic



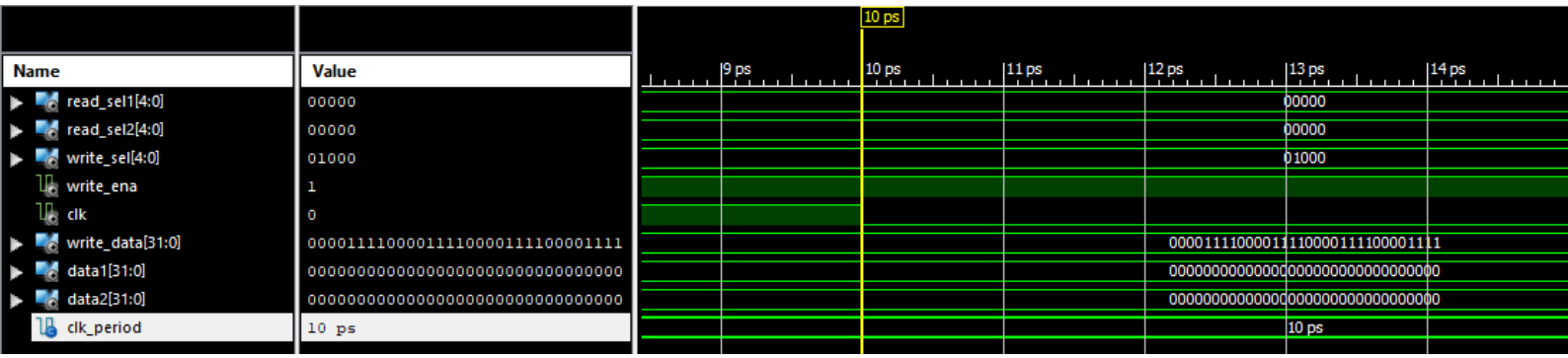
The MIPS Register Testbench Code

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY RegisterFileTest IS
END RegisterFileTest;

ARCHITECTURE behavior OF RegisterFileTest IS

    COMPONENT RegisterFile
    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 COMPONENT;


   signal read_sel1 : std_logic_vector(4 downto 0) := (others => '0');
   signal read_sel2 : std_logic_vector(4 downto 0) := (others => '0');
   signal write_sel : std_logic_vector(4 downto 0) := (others => '0');
   signal write_ena : std_logic := '0';
   signal clk : std_logic := '0';
   signal write_data : std_logic_vector(31 downto 0) := (others => '0');

   signal data1 : std_logic_vector(31 downto 0);
   signal data2 : std_logic_vector(31 downto 0);

   constant clk_period : time := 10 ps;

BEGIN

   uut: RegisterFile PORT MAP (
          read_sel1 => read_sel1,
          read_sel2 => read_sel2,
          write_sel => write_sel,
          write_ena => write_ena,
          clk => clk,
          write_data => write_data,
          data1 => data1,
          data2 => data2
        );

```

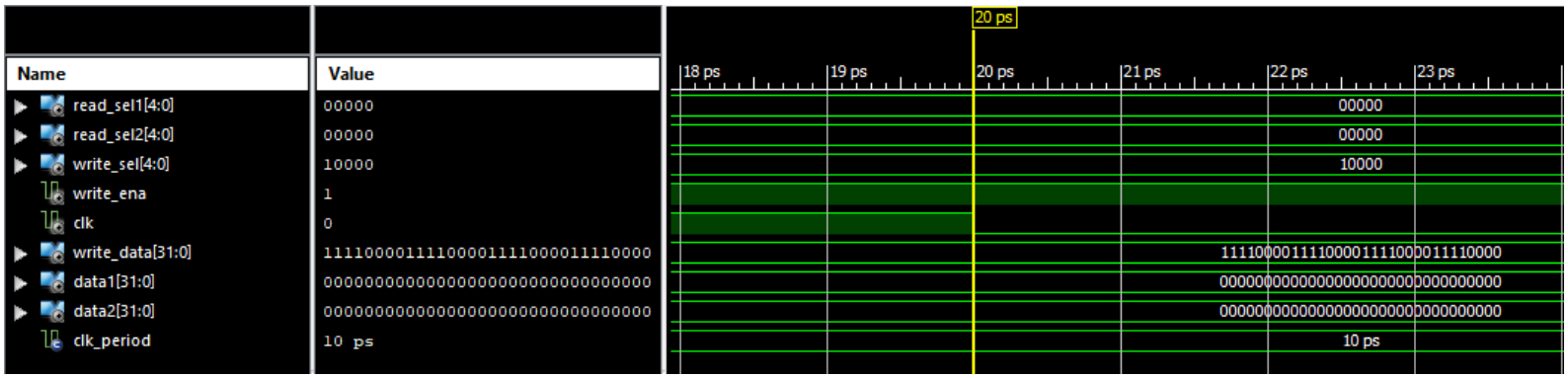
```
47
48      clk_process :process
49      begin
50          clk <= '0';
51          wait for clk_period/2;
52          clk <= '1';
53          wait for clk_period/2;
54      end process;
55
56
57      -- Stimulus process
58      stim_proc: process
59      begin
60          wait for clk_period - 3 ps;
61
62          --Write value in $t0
63              write_sel <= "01000" ; --$t0
64              write_data <= "00001111000011110000111100001111" ;
65              write_ena <= '1' ;
66              wait for clk_period * 1;
67
68          --Write value in $s0
69              write_sel <= "10000" ; --$s0
70              write_data <= "11110000111100001111000011110000" ;
71              write_ena <= '1' ;
72              wait for clk_period * 1;
73
74          --Read data from $t0 and $s0
75              read_sel1 <= "01000" ; --$t0
76              read_sel2 <= "10000" ; --$s0
77              write_ena <= '0' ;
78              wait for clk_period * 2;
79
80              report "Test1";
81              assert(data1 = "00001111000011110000111100001111") report "1:Fail" severity error;
82              report "Test2";
83              assert(data2 = "11110000111100001111000011110000") report "2:Fail" severity error;
84
85              wait for clk_period * 1 ;
86
87          --Read data from $t0 and $s0 and write new data in $t0
88              read_sel1 <= "01000" ; --$t0
89              read_sel2 <= "10000" ; --$s0
90              write_sel <= "01000" ; --$t0
91              write_data <= "00000000000000000000000000000000" ;
92              write_ena <= '1' ;
93
```

```
93                  _
94              wait for clk_period * 2;
95
96              report "Test3";
97              assert(data1 = "00000000000000000000000000000000") report "3:Fail" severity error;
98              report "Test4";
99              assert(data2 = "11110000111100001111000011110000") report "4:Fail" severity error;
100         report "Test Complete";
101
102         wait;
103     end process;
104
105 END;
106
```

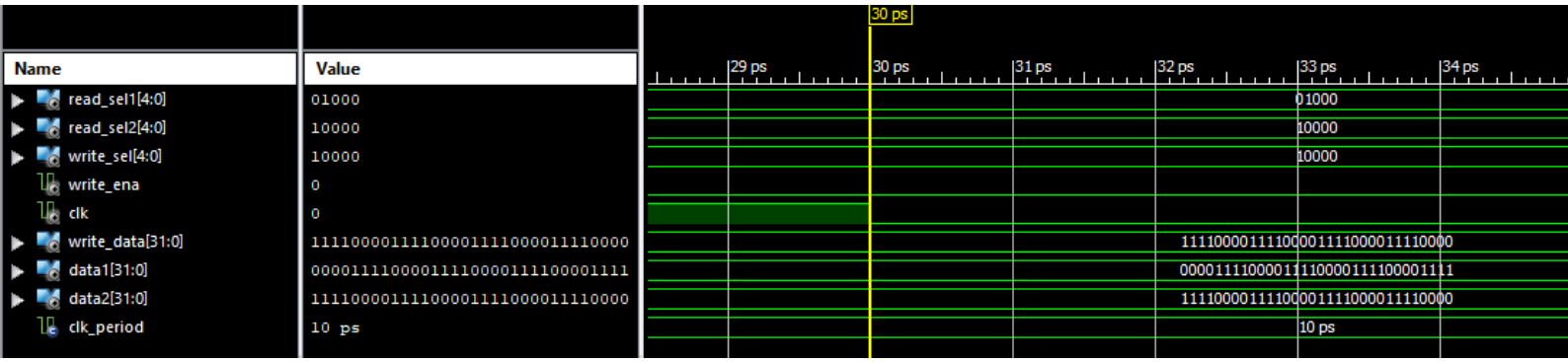
The MIPS Register Sample Output



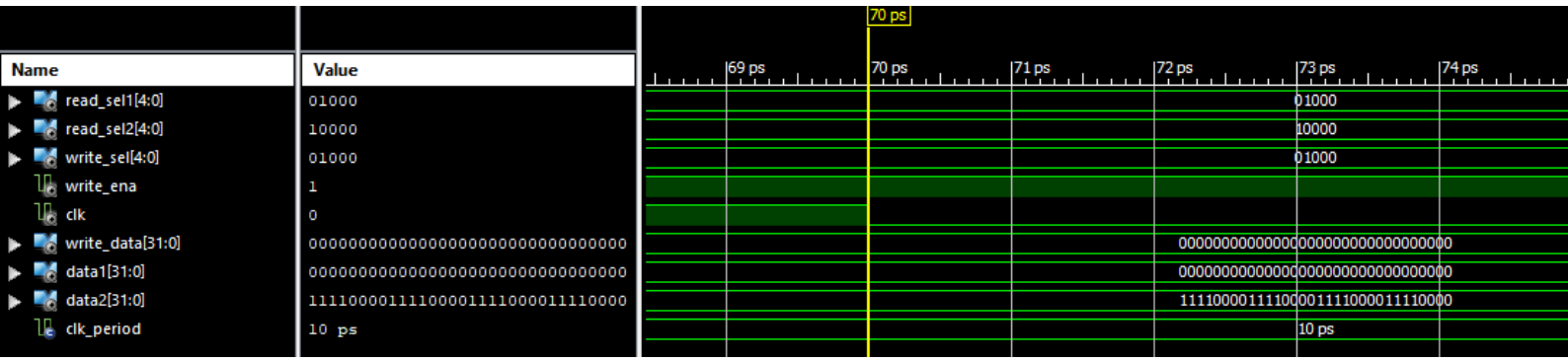
At 10 ps, the write_ena was equal to '1' and the write_sel was equal to "01000" and the write_data was equal to "00001111000011110000111100001111". This means that the 32-bits placed in write_data should be written to the register of the address equal to "01000".



At 20 ps, the write_ena was equal to '1' and the write_sel was equal to "10000" and the write_data was equal to "11110000111100001111000011110000". This means that the 32-bits placed in write_data should be written to the register of the address equal to "01000".



At 30 ps, the write_ena changed to be equal to '0' which means no writing will occur at this step. Also, at the same time the read_sel1 changed to be equal to "01000" and read_sel2 changed to be equal to "10000". This means that reading process will occur and the previously written two values in the previous page will now be read from the equivalent registers and produced as outputs in data1 and data2 respectively. Consequently, data1 is now equal to "00001111000011110000111100001111" and data2 is equal to "11110000111100001111000011110000".



At 70 ps, the write_ena changed to be equal to ‘1’ again which means a writing process will take place. However, read_sel1 was equal to “01000” and read_sel2 was equal to “10000”. Accordingly, writing and reading processes will occur simultaneously. The writing will occur to the register corresponding to the write_sel equal to “01000” with write_data equal to “00000000000000000000000000000000”. While for the reading process, it will happen to the following two registers with addresses read_sel1 “01000” and read_sel2 “10000” respectively. Consequently, data1 is equal to “00000000000000000000000000000000” which was just written and data2 was equal to “11110000111100001111000011110000” which was previously written in previous steps.

The ALU Register Description

For the second part of phase 1, it was implementing a 32-Bit Full ALU. Firstly, we declared the entity as requested in the project description document.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_signed.all;
4
5  entity ALU is
6
7      Port ( data1   : in  STD_LOGIC_VECTOR (31 downto 0);
8             data2   : in  STD_LOGIC_VECTOR (31 downto 0);
9             aluop   : in  STD_LOGIC_VECTOR (3 downto 0);
10            cin     : in  STD_LOGIC;
11            dataout : out STD_LOGIC_VECTOR (31 downto 0);
12            cflag   : out STD_LOGIC;
13            zflag   : out STD_LOGIC;
14            oflag   : out STD_LOGIC);
15
16
17 end ALU;
```

Secondly, we began in writing the architecture of the ALU. We decided to implement the ALU using processes and variables, and we will use the case and for loop. At the beginning, we wrote the process sensitivity list of (data1,data2,aluop,cin) to make the results reassessed whenever one of these four inputs experiences a change. We declared the following variables : temp (with size of n downto 0 – 32 downto 0) which will be used in the addition and subtraction processes, outvariable (with size of n-1 downto 0 – 31 downto 0) which will be used to store the final results of the functions of the ALU, cintemp which will be used to store the carry in of the addition and subtraction processes, cvariable for the carry out of the addition and subtraction processes, and finally zvariable 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, temp, zvariable all with zeros.

```
18
19  architecture Behavioral of ALU is
20
21  begin
22
23     process(data1,data2,aluop,cin)
24     variable temp : STD_LOGIC_VECTOR (32 downto 0);
25     variable outvariable : STD_LOGIC_VECTOR (31 downto 0);
26
27     variable cvariable, zvariable, cintemp : STD_LOGIC;
28        BEGIN
29        cflag <= '0';
30        oflag <= '0';
31        temp := "00000000000000000000000000000000";
32        zvariable := '0';
```

Then, we started in writing the case process with "case aluop" for different actions to be taken based on the value of the aluop. The first case is the addition when the aluop = "0010". Then, the cintemp will be equal to the value of the cin. The temp will be equal to the result of the sum of data1 and data2 and cintemp (carry input if there is carry input by the user), however, data1 and data2 are not summed directly, both data1 and data2 are concatenated with '0' at the most significant bit to increase the size from 32 bit to 33 bits temporarily to provide an extra bit empty for holding any carry out. Moreover, the outvariable will be equal to the temp taking only the first 32 bits and the 33th bit will be placed in the cvariable. For the oflag (overflow flag), we wrote the equation found in the screenshot to detect the overflow which happens if two large numbers are added greater than the value that the software could process resulting in an unexpected answer with an unexpected sign, overflow happens for example when adding two large positive numbers and the result turns out to be negative or adding two negative numbers and the results turns out to be positive. Finally, the cflag is assigned to hold the value of the cvariable. 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 aluop is equal to "0110" along with the cintemp equal to '1' and with a subtraction operand instead.

```
case aluop is

   ------addition-----------
   when "0010" =>
   cintemp := cin;
   temp := ('0' & data1) + ('0' & data2) + cintemp;
   outvariable := temp (31 downto 0);
   cvariable := temp(32);
   oflag <= (data1(31) AND data2(31) AND NOT(outvariable(31))) OR (NOT(data1(31)) AND NOT(data2(31)) AND outvariable(31));
   cflag <= cvariable;
   ---------------------

   ------subtraction-----------
   when "0110" =>
   cintemp := '1';
   temp := ('0' & data1) + ('0' & NOT(data2)) + cintemp;
   outvariable := temp (31 downto 0);
   cvariable := NOT(temp(32));
   oflag <= (data1(31) AND NOT(data2(31)) AND NOT(outvariable(31))) OR (NOT(data1(31)) AND data2(31) AND outvariable(31));
   cflag <= cvariable;
   ---------------------
```

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 outvariable as well. In the case, no one of the aluop stated is entered by the user, the outvariable will be filled with 'Z'.

```
      ------AND-----------
      when "0000" =>
      outvariable := data1 and data2;
      ---------------------

      ------OR-----------
      when "0001" =>
      outvariable := data1 or data2;
      ---------------------

      ------NOR-----------
      when "1100" =>
      outvariable := data1 nor data2;
      ---------------------

      when others =>
      outvariable := "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";

   end case;
```

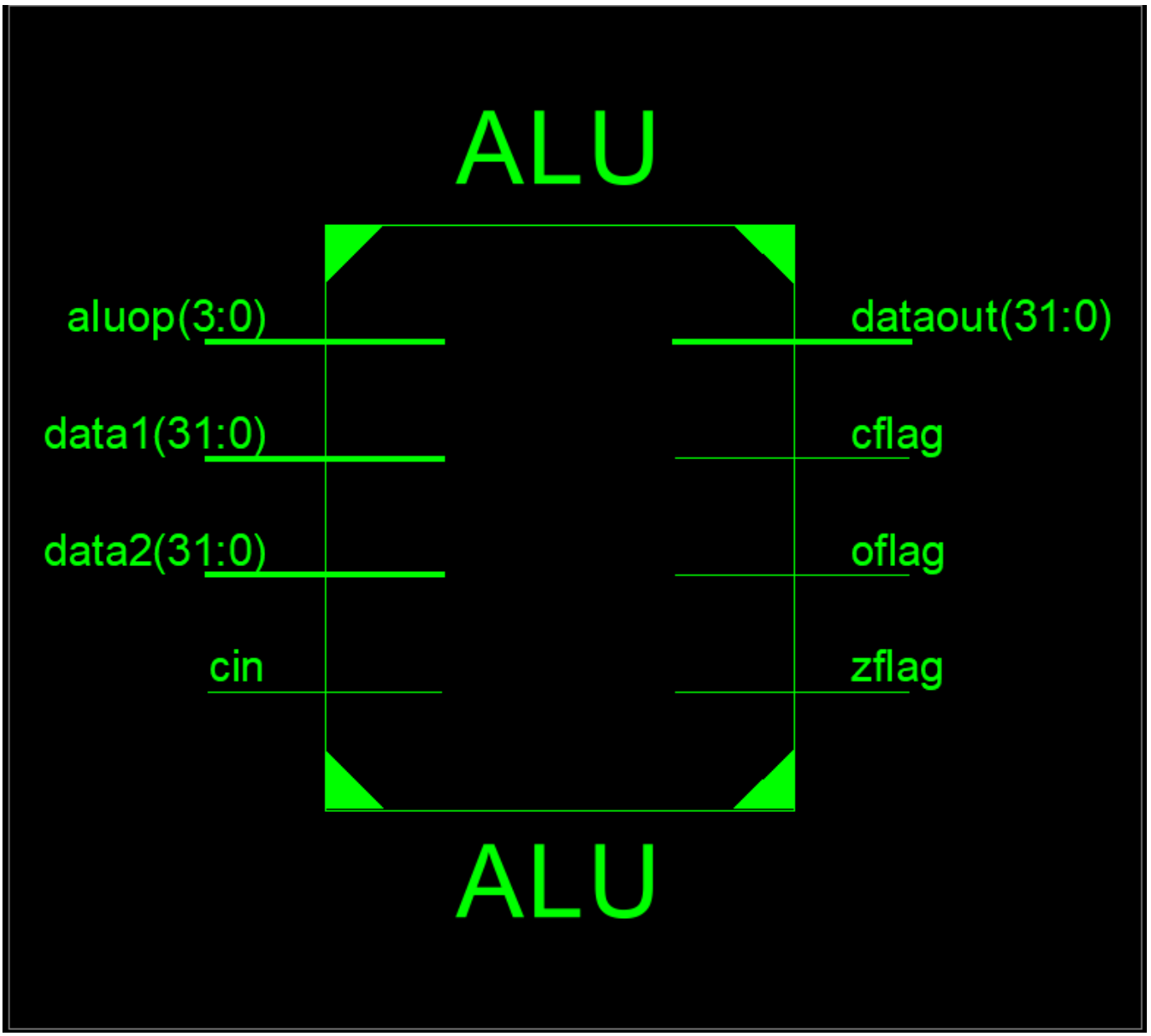
Next, we constructed a for loop that basically checks for all the bits of the outvariable starting for the bit no. 0 to the bit no. 31 if they are all equal to zero by making an "OR" operation with the zvariable which was initialized previously by zero. If all the bits are zeros, the zvariable will remain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

```
--zero output check--
for i in 0 to 31 loop
   zvariable := zvariable or outvariable(i);
end loop;
--------------------
```

At the end, dataout is assigned to hold the contents of the outvariable and the zflag is assigned to hold the opposite of the zvariable to be equal to '1' when all the bits are equal to '0' and vice versa.

```
        dataout <= outvariable;
        zflag <= not zvariable;

END PROCESS;
```

The ALU RTL Schematic



The ALU Testbench Code

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4
5  ENTITY ALUTest IS
6  END ALUTest;
7
8  ARCHITECTURE behavior OF ALUTest IS
9
10
11     COMPONENT ALU
12     PORT(
13          data1 : IN  std_logic_vector(31 downto 0);
14          data2 : IN  std_logic_vector(31 downto 0);
15          aluop : IN  std_logic_vector(3 downto 0);
16          cin : IN  std_logic;
17          dataout : OUT  std_logic_vector(31 downto 0);
18          cflag : OUT  std_logic;
19          zflag : OUT  std_logic;
20          oflag : OUT  std_logic
21         );
22     END COMPONENT;
23
24
25    signal data1 : std_logic_vector(31 downto 0) := (others => '0');
26    signal data2 : std_logic_vector(31 downto 0) := (others => '0');
27    signal aluop : std_logic_vector(3 downto 0) := (others => '0');
28    signal cin : std_logic := '0';
29
30    signal dataout : std_logic_vector(31 downto 0);
31    signal cflag : std_logic;
32    signal zflag : std_logic;
33    signal oflag : std_logic;
34
35
36 BEGIN
37
38    uut: ALU PORT MAP (
39           data1 => data1,
40           data2 => data2,
41           aluop => aluop,
42           cin => cin,
43           dataout => dataout,
44           cflag => cflag,
45           zflag => zflag,
46           oflag => oflag
47         );
```

```
48
49
50
51  -- Stimulus process
52        stim_proc: process
53  begin
54           cin <= '0';
55           wait for 10 ns;
56        --AND testcase1
57           data1 <= "11000000000000000000000000000000" ;
58           data2 <= "10100000000000000000000000000000" ;
59           aluop <= "0000" ;
60           wait for 10 ns;
61           report "Test1";
62           assert(dataout = "10000000000000000000000000000000" and zflag = '0') report "1:Fail" severity error;
63
64           wait for 1 ns;
65
66        --AND testcase2
67           data1 <= x"0F0F0F0F";
68           data2 <= x"F0F0F0FF";
69           aluop <= "0000" ;
70           wait for 10ns;
71           report "Test2";
72           assert(dataout = x"0000000F" and zflag = '0') report "2:Fail" severity error;
73
74           wait for 1 ns;
75
76        --OR testcase1
77           data1 <= "11000000000000000000000000000000" ;
78           data2 <= "10100000000000000000000000000000" ;
79           aluop <= "0001" ;
80           wait for 10 ns;
81           report "Test3";
82           assert(dataout = "11100000000000000000000000000000" and zflag = '0') report "3:Fail" severity error;
83
84           wait for 1 ns;
85
```

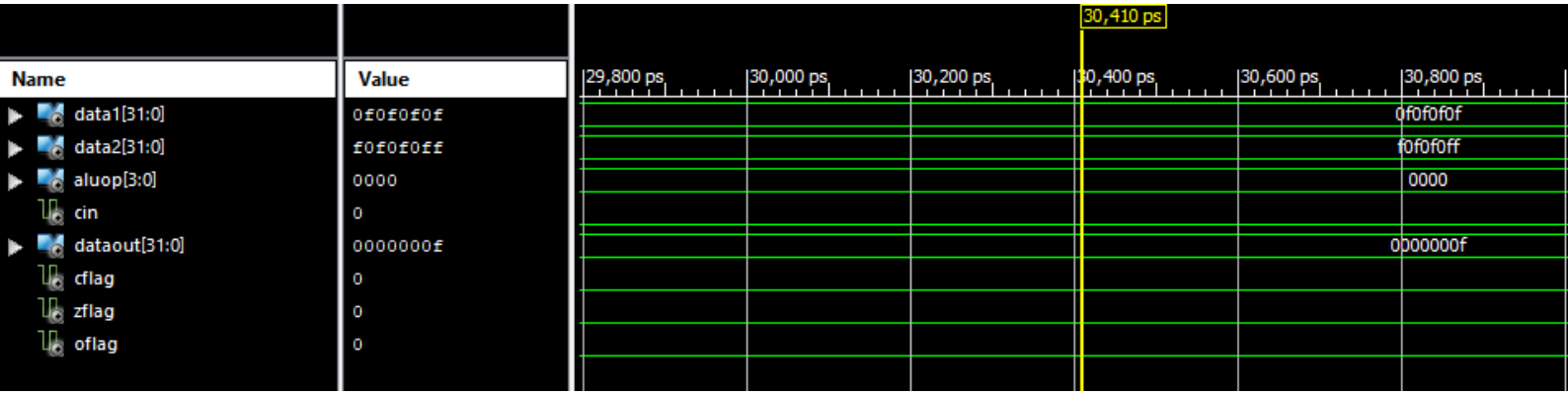
```
85
86     --OR testcase2
87        data1 <= x"0F0F0F0F";
88        data2 <= x"F0F0F0FF";
89        aluop <= "0001" ;
90        wait for 10ns;
91        report "Test4";
92        assert(dataout = x"FFFFFFFF" and zflag = '0') report "5:Fail" severity error;
93
94        wait for 1 ns;
95
96     --NOR testcase1
97        data1 <= "11000000000000000000000000000000" ;
98        data2 <= "10100000000000000000000000000000" ;
99        aluop <= "1100" ;
100       wait for 10 ns;
101       report "Test6";
102       assert(dataout = "00011111111111111111111111111111" and zflag = '0') report "6:Fail" severity error;
103
104       wait for 1 ns;
105
106    --NOR testcase2
107       data1 <= x"0F0F0F0F";
108       data2 <= x"F0F0F0FF";
109       aluop <= "1100" ;
110       wait for 10ns;
111       report "Test7";
112       assert(dataout = x"00000000" and zflag = '1') report "7:Fail" severity error;
113
114       wait for 1 ns;
115
116    --ADD testcase1 (overflow = 1, cout = 0)
117       data1 <= "01110000000000000000000000000000" ;
118       data2 <= "01100000000000000000000000000000" ;
119       aluop <= "0010" ;
120       wait for 10 ns;
121       report "Test8";
122       assert(dataout = "11010000000000000000000000000000" and oflag = '1' and cflag = '0' and zflag = '0') report "8:Fail" severity error;
123
124       wait for 1 ns;
125
```

```
125
126     --ADD testcase2 (zero = 1, cout = 1)
127        data1 <= "11110000000000000000000000000000" ;
128        data2 <= "00010000000000000000000000000000" ;
129        aluop <= "0010" ;
130        wait for 10 ns;
131        report "Test9";
132        assert(dataout = "00000000000000000000000000000000" and oflag = '0' and cflag = '1' and zflag = '1') report "9:Fail" severity error;
133
134        wait for 1 ns;
135
136     --ADD testcase3 (cout = 0)
137        data1 <= x"00000009";
138        data2 <= x"0000000A";
139        aluop <= "0010" ;
140        wait for 10ns;
141        report "Test10";
142        assert(dataout = "00000000000000000000000000010011" and oflag = '0' and cflag = '0' and zflag = '0') report "10:Fail" severity error;
143
144        wait for 1 ns;
145
146     --ADD testcase4 (cout = 1)
147        data1 <= x"FFFFFFF9";
148        data2 <= x"FFFFFFFA";
149        aluop <= "0010" ;
150        wait for 10ns;
151        report "Test11";
152        assert(dataout = "11111111111111111111111111110011" and oflag = '0' and cflag = '1' and zflag = '0') report "11:Fail" severity error;
153
154        wait for 1 ns;
155
156     --ADD testcase5 (overflow =1 cout = 0)
157        data1 <= x"7FFFFFFF";
158        data2 <= x"7FFFFFFF";
159        aluop <= "0010" ;
160        wait for 10ns;
161        report "Test12";
162        assert(dataout = "11111111111111111111111111111110" and oflag = '1' and cflag = '0' and zflag = '0') report "12:Fail" severity error;
163
164        wait for 1 ns;
```

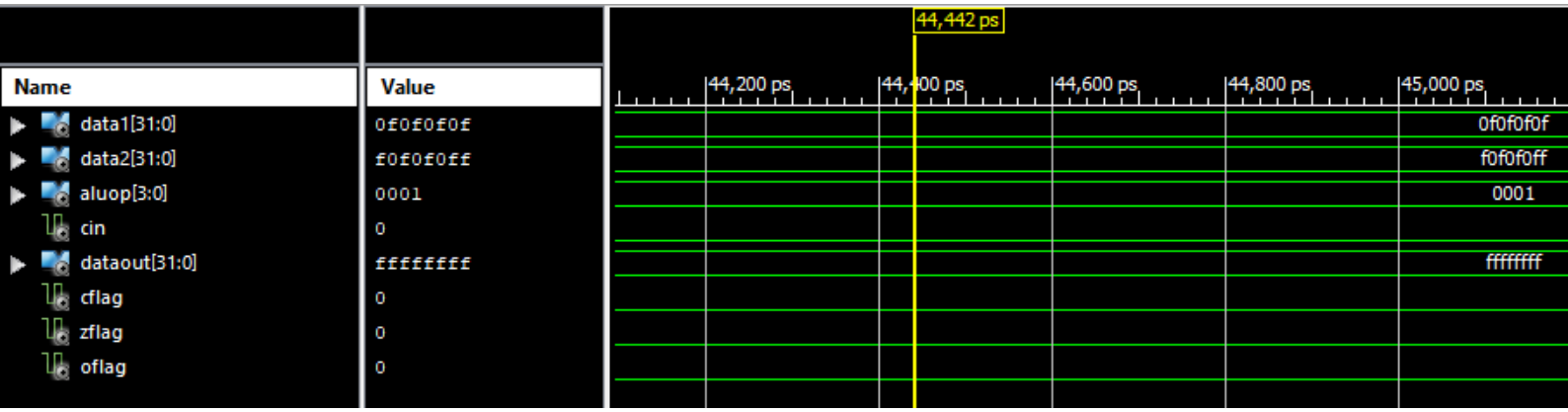
```
165
166     --SUB testcase1 (cout = 0)
167        data1 <= "00000000000000000000000000000111" ; --a = 7
168        data2 <= "00000000000000000000000000000110" ; --b = 6
169        cin <= '1';
170        aluop <= "0110" ;
171        wait for 10 ns;
172        report "Test13";
173        assert(dataout = "00000000000000000000000000000001" and oflag = '0' and cflag = '0' and zflag = '0') report "13:Fail" severity error;
174
175        wait for 1 ns;
176
177     --SUB testcase2 (cout = 1)
178        data1 <= "00000000000000000000000000000110" ; --a = 6
179        data2 <= "00000000000000000000000000000111" ; --b = 7
180        aluop <= "0110" ;
181        wait for 10 ns;
182        report "Test14";
183        assert(dataout = "11111111111111111111111111111111" and oflag = '0' and cflag = '1' and zflag = '0') report "14:Fail" severity error;
184
185        wait for 1 ns;
186
187     --SUB testcase3 (zero = 1, cout = 0)
188        data1 <= "00000000000000000000000000000110" ; --a = 6
189        data2 <= "00000000000000000000000000000110" ; --b = 6
190        aluop <= "0110" ;
191        wait for 10ns;
192        report "Test15";
193        assert(dataout = "00000000000000000000000000000000" and oflag = '0' and cflag = '0' and zflag = '1') report "15:Fail" severity error;
194
195        wait for 1 ns;
196
197     --SUB testcase4 (cout = 1)
198        data1 <= "00000000000000000000000000000110" ; --a = 6
199        data2 <= "11111111111111111111111111111010" ; --b = -6
200        aluop <= "0110" ;
201        wait for 10ns;
202        report "Test16";
203        assert(dataout = "00000000000000000000000000001100" and oflag = '0' and cflag = '1' and zflag = '0') report "16:Fail" severity error;
204
205        wait for 1 ns;
```

```
205         wait for 1 ns;
206
207      --SUB testcase5 (cout = 0)
208         data1 <= x"0000000C";
209         data2 <= x"0000000A";
210         aluop <= "0110" ;
211         wait for 10ns;
212         report "Test17";
213         assert(dataout = "00000000000000000000000000000010" and oflag = '0' and cflag = '0' and zflag = '0') report "17:Fail" severity error;
214
215         wait for 1 ns;
216
217      --SUB testcase6 (cout = 1)
218         data1 <= x"FFFFFFF9";
219         data2 <= x"FFFFFFFA";
220         aluop <= "0110" ;
221         wait for 10ns;
222         report "Test18";
223         assert(dataout = "11111111111111111111111111111111" and oflag = '0' and cflag = '1' and zflag = '0') report "18:Fail" severity error;
224
225         wait for 1 ns;
226
227      --SUB testcase7 (zero = 1, cout = 0)
228         data1 <= x"00000001";
229         data2 <= x"00000001";
230         aluop <= "0110" ;
231         wait for 10ns;
232         report "Test19";
233         assert(dataout = "00000000000000000000000000000000" and oflag = '0' and cflag = '0' and zflag = '1') report "19:Fail" severity error;
234
235         wait for 1 ns;
236
237      report "Test Complete";
238
239      wait;
240   end process;
241
242 END;
243
```

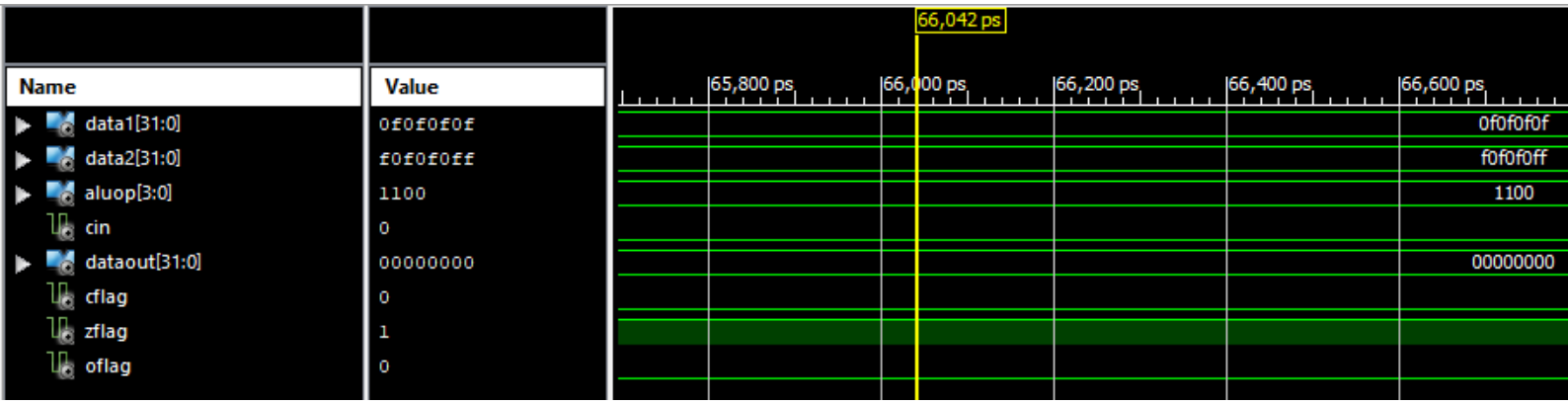
The ALU Sample Output



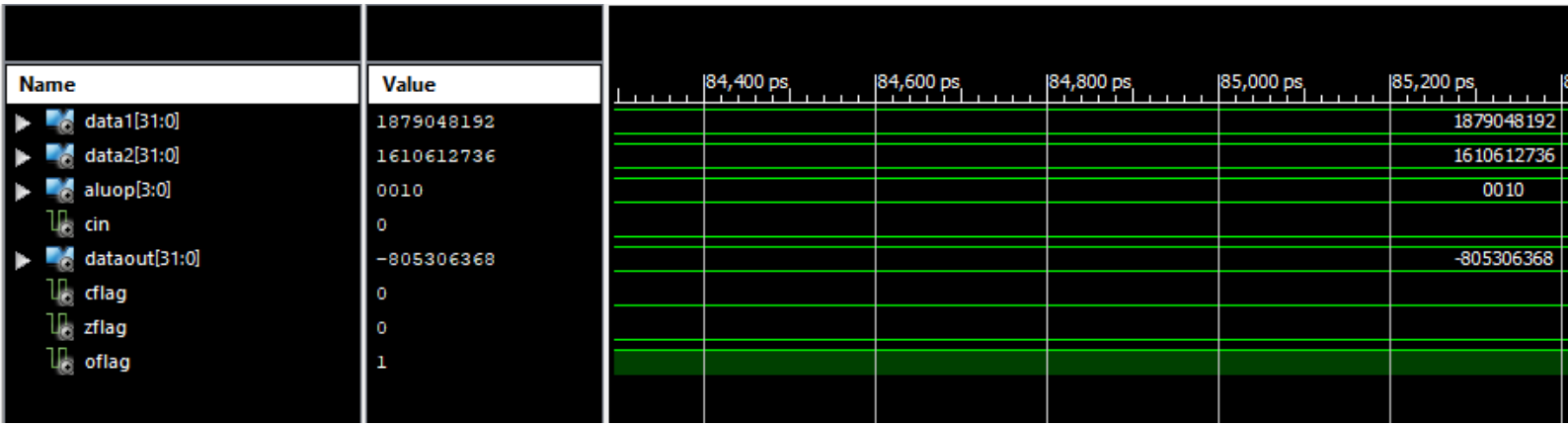
At 30,410 ps, the aluop was equal to "0000", accordingly, "AND" operation should be performed. Data1 was equal to "0f0f0f0f" and data 2 was equal to "f0f0f0f0" both in hexadecimal. Dataout was equal to "0000000f" which was expected and is correct.



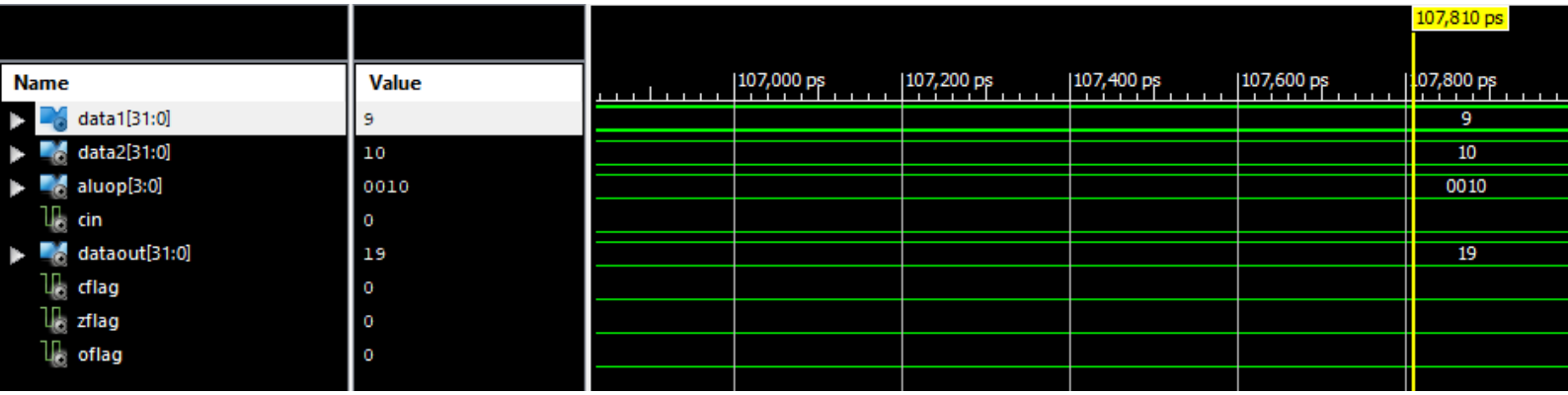
At 44,442 ps, the aluop was equal to "0001", accordingly, "OR" operation should be performed. Data1 was equal to "0f0f0f0f" and data 2 was equal to "f0f0f0f0" both in hexadecimal. Dataout was equal to "ffffffff" which is expected and correct.



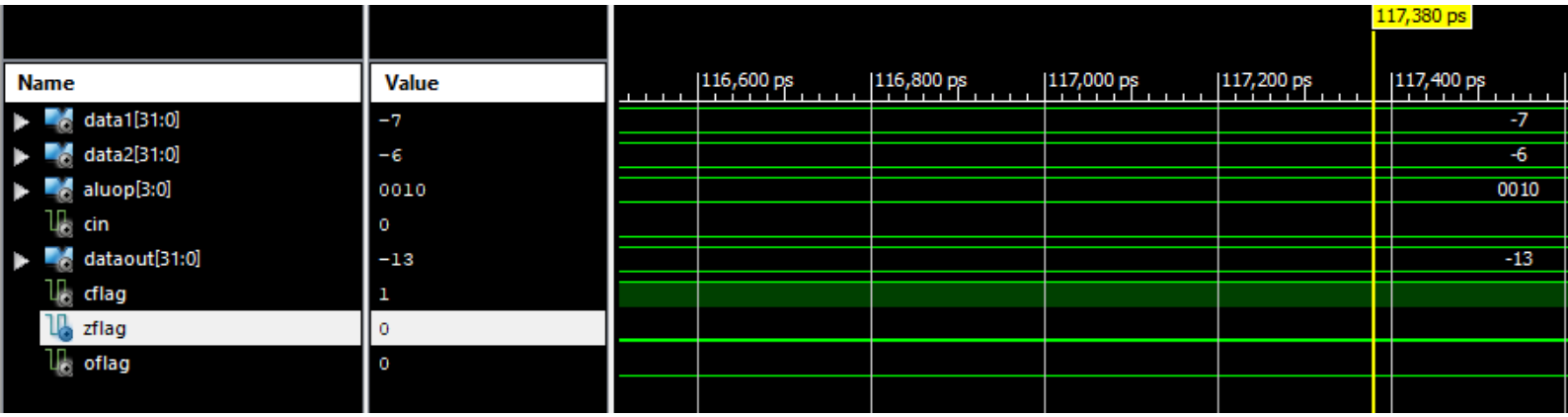
At 66,042 ps, the aluop was equal to “1100”, accordingly, “NOR” operation should be performed. Data1 was equal to “0f0f0f0f” and data 2 was equal to “f0f0f0f0” both in hexadecimal. Dataout was equal to “00000000” which is expected and correct.



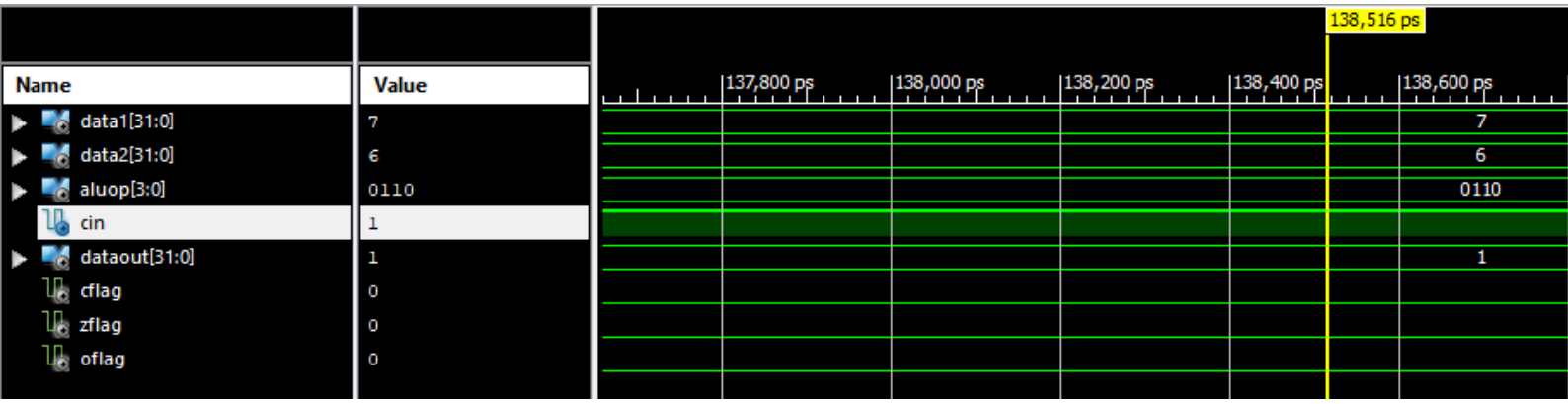
At around 85,200 ps, the aluop was equal to “0010”, accordingly, “Addition” operation should be performed. Data1 was equal to a very large positive number and data2 was equal to a very large positive number. The dataout was equal to a very large negative number which is an error and the case of overflowing, accordingly the oflag was equal to 1.



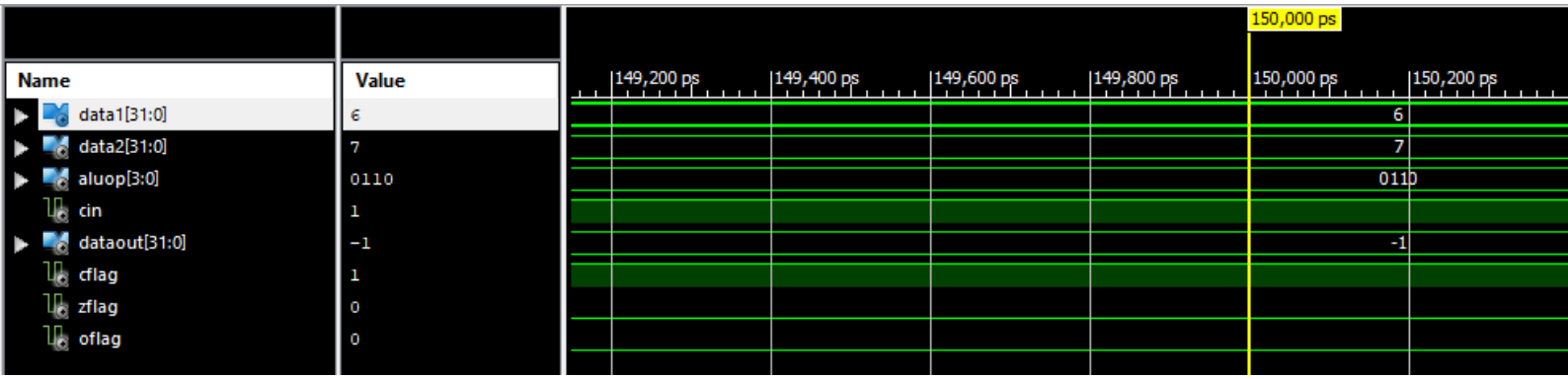
At 107,810 ps, the aluop was equal to “0010”, accordingly, “Addition” operation should be performed. Data1 was equal to “9” and data2 was equal to “10” both in signed decimals. Dataout was equal to “19” and all the three flags were equal to zero which is expected and correct.



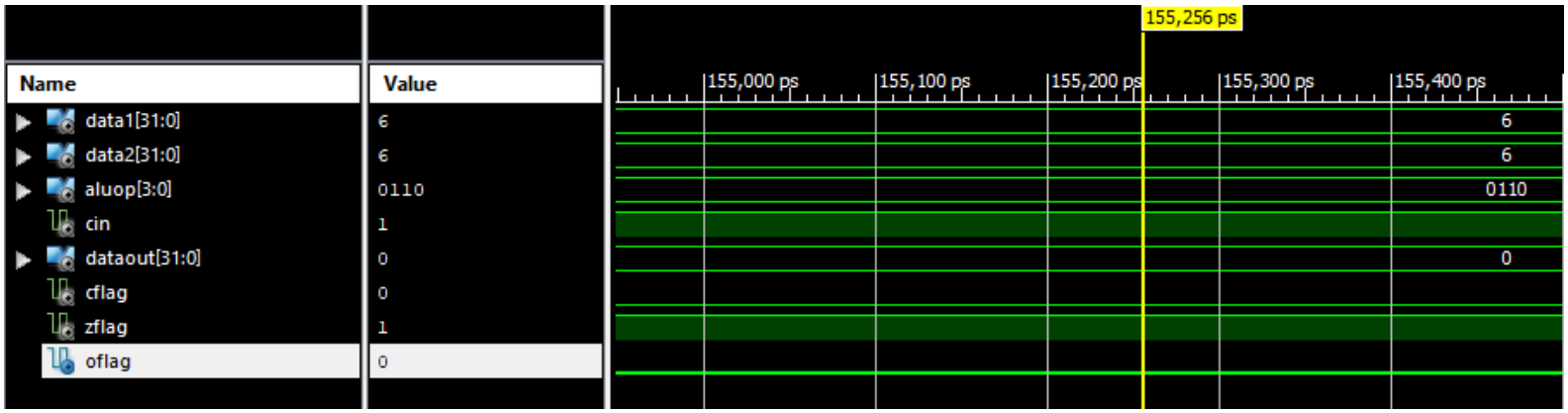
At 117,380 ps, the aluop was equal to “0010”, accordingly, “Addition” operation should be performed. Data1 was equal to “-7” and data2 was equal to “-6” both in signed decimals. Dataout was equal to “-13” with the cflag is equal to ‘1’ indicating that there has been a carry out while the rest two flags were equal to zero which is expected and correct.



At 138,516 ps, the aluop was equal to “0110”, accordingly, “Subtraction” operation should be performed. Data1 was equal to “7” and data2 was equal to “6” both in signed decimals. Dataout was equal to “1” with all the three flags equal to zero which is expected and correct.



At 150 ps, the aluop was equal to “0110”, accordingly, “Subtraction” operation should be performed. Data1 was equal to “6” and data2 was equal to “7” both in signed decimals. Dataout was equal to “-1” with the cflag is equal to ‘1’ indicating that there has been a carry out while the rest two flags were equal to zero which is expected and correct.

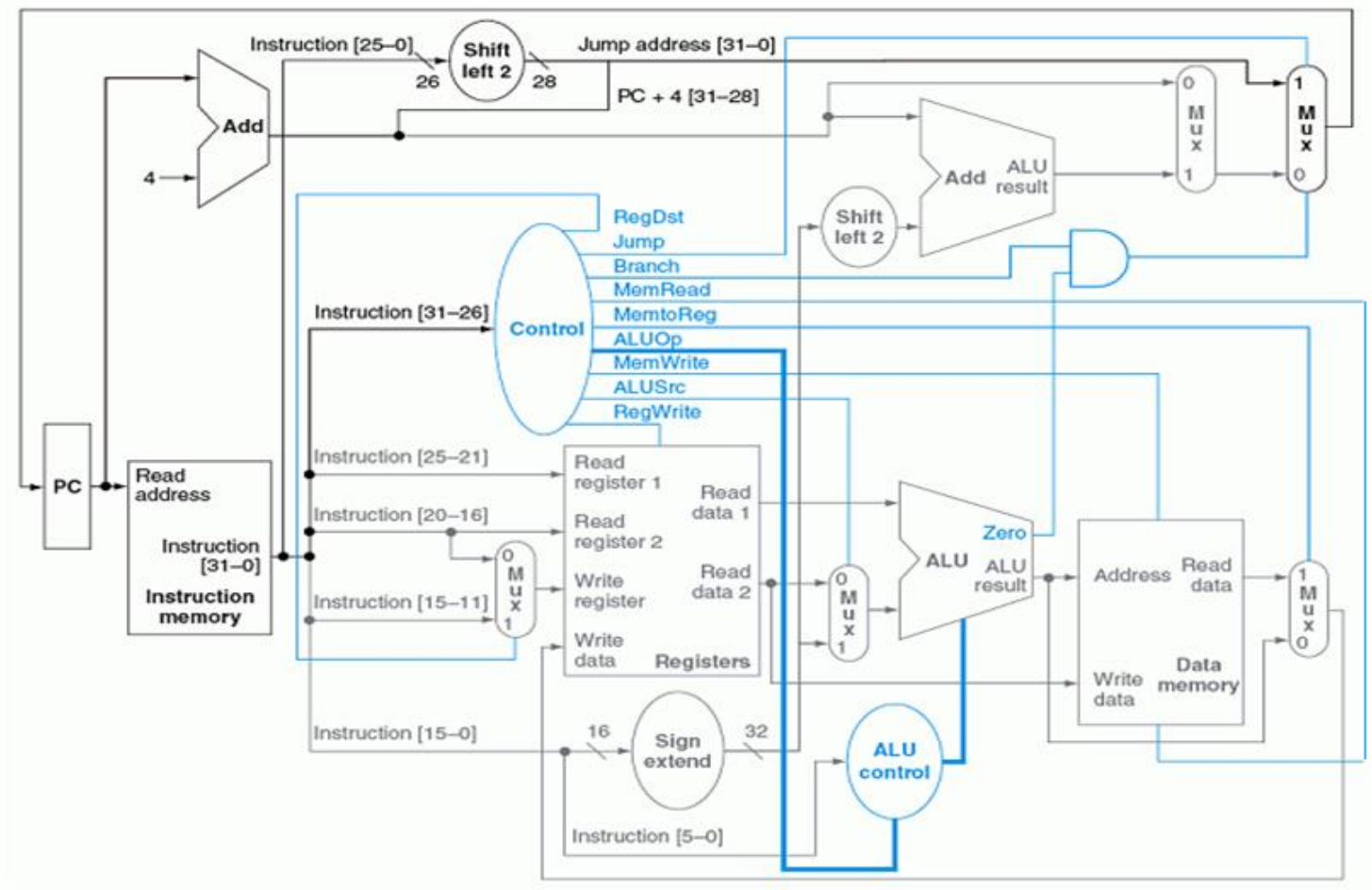


At 155,256 ps, the aluop was equal to “0110”, accordingly, “Subtraction” operation should be performed. Data1 was equal to “6” and data2 was equal to “6” both in signed decimals. Dataout was equal to “0” with the zflag is equal to ‘1’ indicating that the output is equal to zero while the rest two flags were equal to zero which is expected and correct.

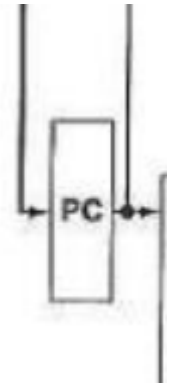
Phase 2

The MIPS CPU Description

In phase 2, it was required to design a simple MIPS CPU. The proposed 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. To make this cpu, we will need to design every single module found in this diagram as an alone module and then use all these modules to make the main module required of the MIPS CPU.



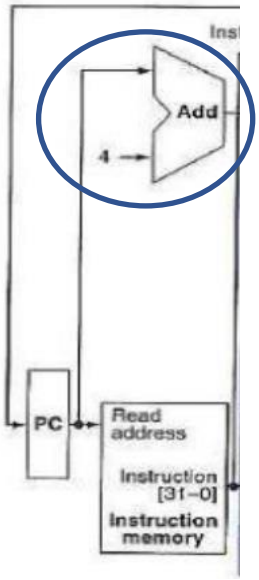
Firstly, we designed the “PC” module which takes three inputs and produces one output. The three inputs are the following: “CLK”, “RESET”, “Address” and the only output is “PC”. The “Address” and the “PC” are both buses with 32 bits, while the “CLK” and the “RESET” are only one bit. Followingly, we created a signal named “instructionAddress” to be used as a buffer inside the process instead of directly using the output and then the signal will be assigned to the output at the end. The architecture of the PC is simple as it is a process which we named “PC_Process” with “CLK” and “RESET” in the sensitivity list. It functions by placing the input “Address” into the signal “instructionAddress” (substitute of the output) at the rising edge of the clock, but if the “RESET” happens to be equal to ‘1’, then the “instructionAddress” will be reset and the whole 32 bits will be zeros.




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

Secondly, we designed the "Adder4" module that increments the output coming from the PC module by 4, it takes one input and produces one output where both are of size 32 bits. The architecture is simple where "Output" will be equal to "Input + 4" and we included the "Numeric_STD" library in order to use the "signed" function needed for the addition.

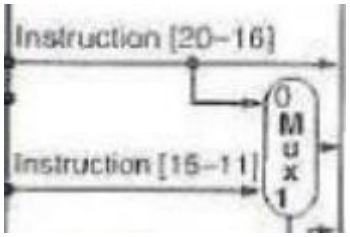
```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity Adder4 is
6      Port ( Input  : in   STD_LOGIC_VECTOR (31 downto 0);
7             Output : out  STD_LOGIC_VECTOR (31 downto 0));
8  end Adder4;
9
10 architecture Behavioral of Adder4 is
11
12 begin
13
14    Output <= STD_LOGIC_VECTOR(signed(Input) + 4);
15
16 end Behavioral;
17
18
```



Thirdly, we designed the "ShiftLeft2" module that shifts the input received by 2 bits from the left side but with preserving the sign-bit. It takes one input and produces one output where both are of size 32 bits. The architecture is simple where the 32th bit of the "Output" (Output (31)) will be equal to the 32th bit of the "Input" (Input (31)) which is the step of preserving the sign-bit. Then, three bits are skipped from the input to do the shifting operation by skipping the already transferred sign-bit (the 32th bit) and the 2 bits of the shifting which will be the 31st and the 30th bits, followingly, starting from the 29th bit till the 0th bit of the input (Input (28 downto 0)) will be transferred to the output to represent the 31st bit to the 3rd bit (Output (30 downto 2)). Accordingly, the output is produced but with the 1st and 0th bit unassigned (Output (1 downto 0)) which will be filled with zeros according to the rules of shifting.

```
 1  library IEEE;
 2  use IEEE.STD_LOGIC_1164.ALL;
 3
 4  entity ShiftLeft2 is
 5      Port ( Input  : in   STD_LOGIC_VECTOR (31 downto 0);
 6             Output : out  STD_LOGIC_VECTOR (31 downto 0));
 7  end ShiftLeft2;
 8
 9  architecture Behavioral of ShiftLeft2 is
10  begin
11
12     Output(31)          <= Input(31);
13     Output(30 downto 2) <= Input(28 downto 0);
14     Output(1 downto 0)  <= (Others => '0');
15
16  end Behavioral;
```

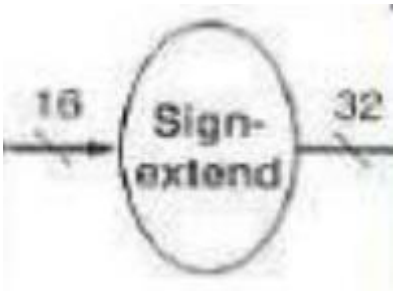
Fourthly, we designed two multiplexers, both multiplexers are 2x1 Mux receiving two inputs, one-bit selection line and producing one output. The two multiplexers are the same architecture and function, but one was designed to receive 32-bits inputs producing one 32-bits output and the other receives 5-bits inputs producing one 5-bits output instead. This is because all multiplexers used in the CPU are working with 32-bits basis but the multiplexer whose output goes into the "WriteRegister" which receives the address of the register for the data to be written into is 5-bits. The two multiplexers are named "CPUMux" and "5x1Mux" (5x1 was used just to indicate that it receives 5-bits and produces 1 output) respectively. The architecture is easy where the output will be determined based on the selection line which is named "Selector" in "CPUMux" and named 'S' in "5x1Mux", if the selection line is equal to '0', the output will be equal to the input "i0", and if the selection line is equal to '1', the output will be equal to the input "i1" while all other cases will produce 32 Z's.



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

entity CPUMux is
    Port ( Selector : in  STD_LOGIC;
           i0       : in  STD_LOGIC_VECTOR (31 downto 0);
           i1       : in  STD_LOGIC_VECTOR (31 downto 0);
           Output   : out STD_LOGIC_VECTOR (31 downto 0));
end CPUMux;

architecture Behavioral of CPUMux is

begin

   Output <= i0 when Selector = '0' else
             i1 when Selector = '1' else
             (others => 'Z');

end Behavioral;
```

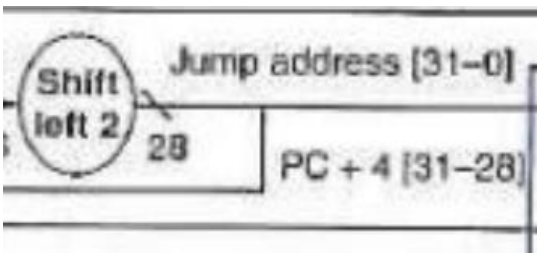
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Mux5x1 is
    Port ( S      : in   STD_LOGIC;
           i0     : in  STD_LOGIC_VECTOR (4 downto 0);
           i1     : in  STD_LOGIC_VECTOR (4 downto 0);
           output : out  STD_LOGIC_VECTOR (4 downto 0));
end Mux5x1;

architecture Behavioral of Mux5x1 is

begin

  output <= i0  when S = '0' else
            i1  when S = '1' else
            (others => 'Z');

end Behavioral;
```

Fifthly, we designed the “SignExtend” module that does what it actually says by receiving a 16-bit input and extend its sign to produce a 32-bit output where the 16 bits difference is a repetition of the sign-bit. How it works is as the two lines written in the architecture where it checks the sign-bit of the input which the 16th bit (Input (15)), if the sign-bit is ‘1’, then the output will be equal to input but concatenated with 16 ones to the left side of the input, however, if the sign-bit is ‘0’, the output will be equal to the input concatenated with 16 zeros the left side of the input and any other case will produce Zs.



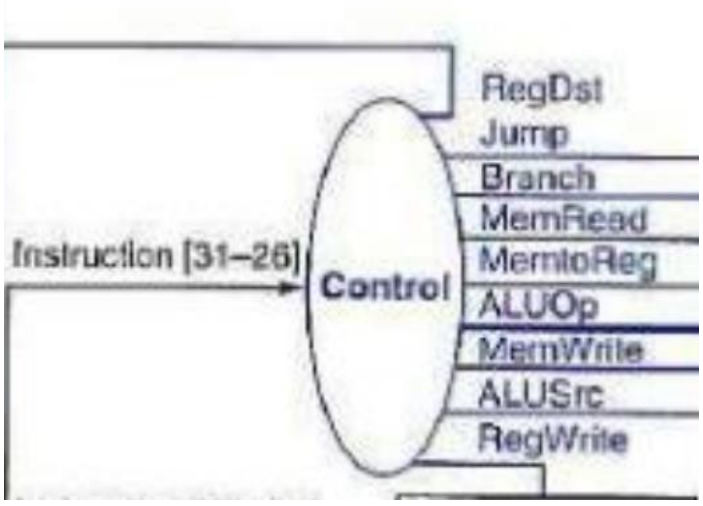

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity SignExtend is
5      Port ( Input  : in  STD_LOGIC_VECTOR (15 downto 0);
6             Output : out STD_LOGIC_VECTOR (31 downto 0));
7  end SignExtend;
8
9  architecture Behavioral of SignExtend is
10 begin
11
12    Output <= "0000000000000000" & Input WHEN Input(15) = '0' ELSE
13              "1111111111111111" & Input WHEN Input(15) = '1' ELSE
14              (Others => 'Z');
15
16
17 end Behavioral;
```

Moving on to the 6th module, we made a module named “JumpAddressConcat” which will be responsible for creating the jump address by the concatenation of the last 4 bits from the output of the “Adder4” module (PC + 4) (InputLeft (31 downto 28)) to the first 28 bits of the output from shifting left 2 of the instruction (InputRight (27 downto 0)).



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

entity JumpAddressConcat is
    Port ( InputLeft  : in   STD_LOGIC_VECTOR (31 downto 0);
           InputRight  : in   STD_LOGIC_VECTOR (31 downto 0);
           Output : out  STD_LOGIC_VECTOR (31 downto 0));
end JumpAddressConcat;

architecture Behavioral of JumpAddressConcat is
begin

   Output <= InputLeft(31 downto 28) & InputRight(27 downto 0);

end Behavioral;
```

For the 7th module, it will be the "MainControlUnit" which is responsible for controlling the CPU, it takes only one 6-bit input and produces 9 outputs where all outputs are a single bit except the "ALUOp" output will be 2 bits. The input is the OpCode which the last 6 bits of the instruction loaded (instruction [31-26]). The eight single bit outputs are the following: MemRead, MemWrite, RegDst, Branch, MemtoReg, ALUSrc, Jump and RegWrite. The "MainControlUnit" was using the case process format where cases are based on the following truth table which could be found in Lecture 10.



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

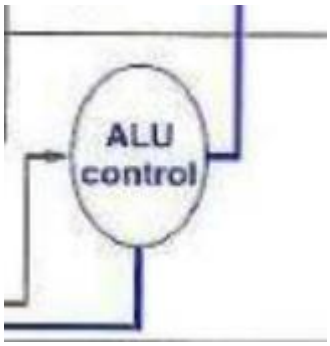
However, the jump process and the bne were not in this table. In the case of the bne, the outputs are exactly like the beq process, but the opcode is different as the bne opcode is “000101”. For the jump, the opcode will be equal to “000010” all the outputs are zeros except for the RegDst and the MemtoReg will be don’t cares (‘X’) and obviously the jump output will be equal to ‘1’. For any other cases, all the outputs will be reset and equal to zeros.

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

entity MainControlUnit is

    Port ( OpCode   : in  STD_LOGIC_VECTOR (5 downto 0);
           MemRead  : out  STD_LOGIC;
           MemWrite : out  STD_LOGIC;
           RegDst   : out  STD_LOGIC;
           Branch   : out  STD_LOGIC;
           MemtoReg : out  STD_LOGIC;
           ALUSrc   : out  STD_LOGIC;
           ALUOp    : out  STD_LOGIC_VECTOR (1 downto 0);
           Jump     : out STD_LOGIC;
           RegWrite : out  STD_LOGIC);

end MainControlUnit;

architecture Behavioral of MainControlUnit is
begin

   Process (OpCode)
   BEGIN

      RegWrite <= '0';

         CASE OpCode IS

            ---------RType---------
            WHEN "000000" =>
               MemRead   <= '0'  ;
               MemWrite  <= '0'  ;
               RegDst    <= '1'  ;
               Branch    <= '0'  ;
               MemtoReg  <= '0'  ;
               ALUSrc    <= '0'  ;
               ALUOp     <= "10" ;
               Jump      <= '0'  ;
               RegWrite  <= '1'  ;
            ----------------------

```

```
41
42              ---------LW---------
43              WHEN "100011" =>
44                  MemRead   <= '1'  ;
45                  MemWrite  <= '0'  ;
46                  RegDst    <= '0'  ;
47                  Branch    <= '0'  ;
48                  MemtoReg  <= '1'  ;
49                  ALUSrc    <= '1'  ;
50                  ALUOp     <= "00" ;
51                  Jump      <= '0'  ;
52                  RegWrite  <= '1'  ;
53              ----------------------
54
55              ---------SW---------
56              WHEN "101011" =>
57                  MemRead   <= '0'  ;
58                  MemWrite  <= '1'  ;
59                  RegDst    <= 'X'  ;
60                  Branch    <= '0'  ;
61                  MemtoReg  <= 'X'  ;
62                  ALUSrc    <= '1'  ;
63                  ALUOp     <= "00" ;
64                  Jump      <= '0'  ;
65                  RegWrite  <= '0'  ;
66              ----------------------
67
68              ---------beq---------
69              WHEN "000100" =>
70                  MemRead   <= '0'  ;
71                  MemWrite  <= '0'  ;
72                  RegDst    <= 'X'  ;
73                  Branch    <= '1'  ;
74                  MemtoReg  <= 'X'  ;
75                  ALUSrc    <= '0'  ;
76                  ALUOp     <= "01" ;
77                  Jump      <= '0'  ;
78                  RegWrite  <= '0'  ;
79              ----------------------
```

```
80
81              ---------bne---------
82              WHEN "000101" =>
83                 MemRead   <= '0'  ;
84                 MemWrite  <= '0'  ;
85                 RegDst    <= 'X'  ;
86                 Branch    <= '1'  ;
87                 MemtoReg  <= 'X'  ;
88                 ALUSrc    <= '0'  ;
89                 ALUOp     <= "01" ;
90                 Jump      <= '0'  ;
91                 RegWrite  <= '0'  ;
92              ----------------------
93
94
95              ---------Jump---------
96              WHEN "000010" =>
97                 MemRead   <= '0'  ;
98                 MemWrite  <= '0'  ;
99                 RegDst    <= 'X'  ;
100                Branch    <= '0'  ;
101                MemtoReg  <= 'X'  ;
102                ALUSrc    <= '0'  ;
103                ALUOp     <= "00" ;
104                Jump      <= '1'  ;
105                RegWrite  <= '0'  ;
106             ----------------------
107
108             WHEN OTHERS => NULL;
109                MemRead   <= '0'  ;
110                MemWrite  <= '0'  ;
111                RegDst    <= '0'  ;
112                Branch    <= '0'  ;
113                MemtoReg  <= '0'  ;
114                ALUSrc    <= '0'  ;
115                ALUOp     <= "00" ;
116                Jump      <= '0'  ;
117                RegWrite  <= '0'  ;
118
119
120          END CASE;
121
122    END PROCESS;
123
124
125 end Behavioral;
```

Next was the "ALUControlUnit". This module takes in two inputs, the FunctionCode which is 6 bits (it is the function field in the instruction which is the first 6 bits – instruction [5-0]) and the ALUOp which is 2 bits which is the same ALUOp coming out as an output from the "MainControlUnit". The output produced is the "ALUFunct" of a size of 4 bits which is the ALU Control Lines which will be the input of the ALU to decide which operation to make. The "ALUControlUnit" was designed based on the upcoming truth table.

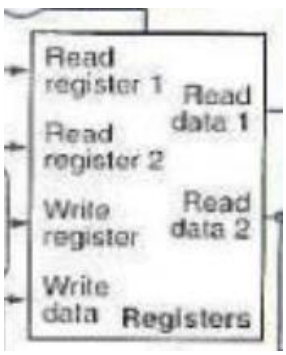


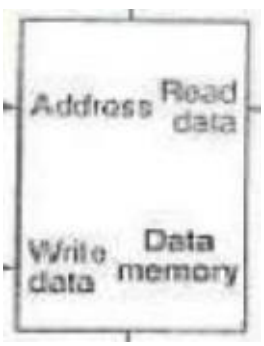
Instruction OpCode	ALUOp	Instruction Operation	Funct Field	Desired ALU Action	ALU Control Input
LW	00	Load Word	xxxxxx	Add	0010
SW	00	Store Word	xxxxxx	Add	0010
BEQ	01	Branch Equal	xxxxxx	Subtract	0110
BNE	01	Branch Not Equal	xxxxxx	Subtract	0110
R-Type	10	Add	100000	Add	0010
R-Type	10	Subtract	100010	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
R-Type	10	Nor	100111	Nor	1100

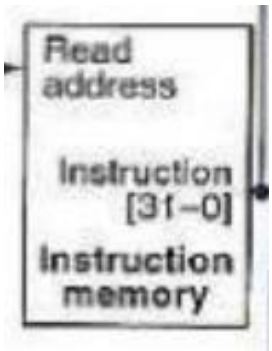
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ALUControlUnit is
    Port ( FunctionCode : in  STD_LOGIC_VECTOR (5 downto 0);
           ALUOp        : in  STD_LOGIC_VECTOR (1 downto 0);
           ALUFunct     : out STD_LOGIC_VECTOR (3 downto 0));
end ALUControlUnit;

architecture Behavioral of ALUControlUnit is
begin


ALUFunct(3) <= (ALUOp(1) AND FunctionCode(0) AND FunctionCode(1) AND FunctionCode(5));

ALUFunct(2) <= ALUOp(0) OR (ALUOp(1) AND FunctionCode(1));

ALUFunct(1) <= NOT ALUOp(1) OR ((NOT FunctionCode(2)) AND NOT(FunctionCode(0)));

ALUFunct(0) <= (FunctionCode(3) OR FunctionCode(0)) AND (NOT (FunctionCode(1) AND FunctionCode(0))) AND ALUOp(1);


end Behavioral;
```

That leaves us with four components or modules for the CPU which are the Registers, ALU, Data Memory, Instruction Memory. For the Registers and the ALU, we used the previously built modules of "RegisterFile" and "ALU" of Phase 1 but with a small modification to the ALU which is adding the SLT function (Set On Less Than), accordingly, you could refer back to the description of both modules in the "Phase 1" section.

```
70
71              ------slt-----------
72              when "0111" =>
73              if (data1<data2) THEN
74              outvariable := x"00000001";
75              else
76              outvariable := x"00000000";
77              end if;
78              ---------------------
```







Regarding the data and instruction memories, they were already provided by the doctor, so we used them with only minor edits. For both modules, we removed the generic part and replaced the sizes with direct numbers instead as the whole modules were non-generic, so we decided to complete it that way. However, in the instruction memory, there were some errors regarding the instructions supplied, so we made some more edits. Firstly, we deactivated (by making it in the format of a comment) the last instruction which was originally placed in MEMORY(32) which was a jumping instruction and instead we made MEMORY(32) to hold the binary value of this instruction "add $s0, $s0, $t8" which will is responsible for writing the last value of the sequence which is '-1' in the register $s0. Also, we added an extra instruction to be MEMORY(33) which is the same instruction in the previous line of MEMORY(32) but this time the goal was not performing the exact instruction itself but was to add any extra instruction with no concern about the instruction contents itself but only an instruction the involves using register $s0 in order for us to read the new value of $s0 which was written by the previous instruction.

```
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
6  use IEEE.std_logic_textio.all;
7  library STD;
8  use STD.textio.all;
9
10 ----Only removed the generic variables and replaced them with equivalent numbers as the whole cod
11
12 entity INSTRMEMORY is
13
14   port(
15     LoadIt  : in Std_logic ;
16     DATA    : out STD_LOGIC_VECTOR(31 downto 0);
17     ADDRESS : in STD_LOGIC_VECTOR(31 downto 0);
18     CLK     : in STD_LOGIC
19     );
20 end INSTRMEMORY;
21
22
23 architecture BEHAVIORAL of INSTRMEMORY is
24    signal ADDRover4: STD_LOGIC_VECTOR(29 downto 0);
25 begin
26
27 ROM_PROCESS: process(CLK, ADDRESS) is
28                                    type MEM is array(0 to 63) of STD_LOGIC_VECTOR(31 downto 0);
29                                    variable MEMORY: MEM := (others => X"00000000");
30                                    variable IADR: INTEGER;
31
32 begin
33
```

For the cut off comments, it is briefly described in the previous paragraphs, however if you want to see the full comments, please check the VHDL file itself.

```
34      if LoadIt = '1' then
35  -------------------------
36  --Project1 test
37  -------------------------
38        MEMORY(0)  := "00000000000000000100000000100101" ;
39        MEMORY(1)  := "10001100000011010000000000110000" ;
40        MEMORY(2)  := "10001100000011000000000000110100" ;
41        MEMORY(3)  := "10001100000011001000000000111000" ;
42        MEMORY(4)  := "00000000000011000010100000100000" ;
43        MEMORY(5)  := "10101101000010100000000000000000" ;
44        MEMORY(6)  := "10101101000010100000000000000100" ;
45        MEMORY(7)  := "00000001101110000100100000100010" ; ---
46        MEMORY(8)  := "00000001001110000100100000100010" ;
47        MEMORY(9)  := "10001101000010110000000000000000" ; -- loop
48        MEMORY(10) := "10001101000011000000000000000100" ;
49        MEMORY(11) := "00000001011011000101000000100000" ;
50        MEMORY(12) := "10101101000010100000000000001000" ;
51        MEMORY(13) := "00000001000110010100000000100000" ;
52        MEMORY(14) := "00000001001110000100100000100010" ;
53        MEMORY(15) := "00000000000010010000100000101010" ;
54        MEMORY(16) := "00010000001000000000000000000001" ; -- beq   $at, $zero, exit
55        MEMORY(17) := "00001000000000000000000000001001" ;
56        MEMORY(18) := "00000000000001000010000000100100" ;
57        MEMORY(19) := "00000000000011010010100000100000" ;
58        MEMORY(20) := "00001000000000000000000000010110" ;
59        MEMORY(21) := "00000000000011000100000000100010" ; -- sub $s0, $zero, $t8
60        MEMORY(22) := "00000001000000000100000000100100" ;
61        MEMORY(23) := "00000001000010000100100000100000" ;
62        MEMORY(24) := "00000001001010010100100000100000" ;
63        MEMORY(25) := "00000000100010010101000000100000" ;
64        MEMORY(26) := "10001101010100000000000000000000" ;
65        MEMORY(27) := "00000001000110000100000000100000" ;
66        MEMORY(28) := "00000001000001010000100000101010" ;
67        MEMORY(29) := "00010100001000001111111111111001" ; -- BNE $at $zero 0xFFF9 -- 0x1420FFF9
68        MEMORY(30) := "00000000000000001100000000100111" ;
69        MEMORY(31) := "00000010000110001000000000100111" ;
70  --    MEMORY(32) := "00001000000000000000000000010101" ; -- removed this instruction of jumping
71        MEMORY(32) := "00000010000110001000000000100000" ; -- added this instruction of add $s0, $
72        MEMORY(33) := "00000010000110001000000000100000" ; -- added an extra instruction same as t
73
74
75    end if;
76
77     if FALLING_EDGE(CLK) then
78       IADR:= CONV_INTEGER(ADDRover4);
79       DATA <= MEMORY(IADR);
80     end if;
```

```
76
77       if FALLING_EDGE(CLK) then
78         IADR:= CONV_INTEGER(ADDRover4);
79         DATA <= MEMORY(IADR);
80       end if;
81     end process;
82
83     ADDRover4 <= ADDRESS(31 downto 2) ;
84
85  end BEHAVIORAL;
```

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

entity DATAMEMORY is
----Only removed the generic variables and replaced them with equiva
  port ( LoadIt    : in STD_LOGIC;
         INPUT     : in STD_LOGIC_VECTOR (31 downto 0);
         OUTPUT    : out STD_LOGIC_VECTOR (31 downto 0);
         MEM_READ  : in STD_LOGIC;
         MEM_WRITE : in STD_LOGIC;
         ADDRESS   : in STD_LOGIC_VECTOR (31 downto 0);
         CLK       : in STD_LOGIC
         );

end DATAMEMORY;

architecture BEHAVIORAL of DATAMEMORY 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;
begin

  process ( MEM_READ, MEM_WRITE, CLK, ADDRESS, INPUT ) is
  begin
  if LoadIt = '1' then
-----------------------
--Project1 test
-----------------------
      memory(0)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
      memory(1)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
      memory(2)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
      memory(3)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
      memory(4)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
      memory(5)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
      memory(6)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
      memory(7)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
      memory(8)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
      memory(9)  <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
      memory(10) <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
      memory(11) <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" ;
      memory(12) <= "00000000000000000000000000001100" ;
      memory(13) <= "00000000000000000000000000000001" ;
      memory(14) <= "00000000000000000000000000000100" ;

  else
    if FALLING_EDGE(CLK) then
      if MEM_WRITE = '1' then
         MEMORY(ADDR_int) <= INPUT;
      end if;
    end if;
   end if;

end process;

   ADDRover4 <= ADDRESS(31 downto 2) ;
   ADDR_int <= CONV_INTEGER(ADDRover4);
   OUTS <= MEMORY(ADDR_int) when MEM_READ = '1' and (ADDR_int < 64) else
           (others => 'Z') when MEM_READ = '0' ;

   OUTPUT <= OUTS;

end BEHAVIORAL;
```

Moving on to the next step, we created a package by the name of "MIPSPackage" where we placed all the previous components and also added several signals that will be used later on in the main module called "MIPSCPU".

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

package MIPSPackage is

    component ALUControlUnit is
        Port ( FunctionCode : in  STD_LOGIC_VECTOR (5 downto 0);
               ALUOp        : in  STD_LOGIC_VECTOR (1 downto 0);
               ALUFunct     : out STD_LOGIC_VECTOR (3 downto 0));
    end component;



    component ALU is

        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;
               zflag   : out STD_LOGIC;
               oflag   : out STD_LOGIC);


    end component;



    component CPUMux is
        Port ( Selector : in  STD_LOGIC;
               i0       : in  STD_LOGIC_VECTOR (31 downto 0);
               i1       : in  STD_LOGIC_VECTOR (31 downto 0);
               Output   : out STD_LOGIC_VECTOR (31 downto 0));
    end component;

```

```
38
39      component DATAMEMORY is
40
41        port ( LoadIt     : in STD_LOGIC;
42               INPUT      : in STD_LOGIC_VECTOR (31 downto 0);
43               OUTPUT     : out STD_LOGIC_VECTOR (31 downto 0);
44               MEM_READ   : in STD_LOGIC;
45               MEM_WRITE  : in STD_LOGIC;
46               ADDRESS    : in STD_LOGIC_VECTOR (31 downto 0);
47               CLK        : in STD_LOGIC
48               );
49
50      end component;
51
52
53
54      component INSTRMEMORY is
55
56        port(
57          LoadIt  : in Std_logic ;
58          DATA    : out STD_LOGIC_VECTOR(31 downto 0);
59          ADDRESS : in STD_LOGIC_VECTOR(31 downto 0);
60          CLK     : in STD_LOGIC
61          );
62      end component;
63
64
65      component MainControlUnit is
66
67          Port ( OpCode   : in  STD_LOGIC_VECTOR (5 downto 0);
68                 MemRead  : out  STD_LOGIC;
69                 MemWrite : out  STD_LOGIC;
70                 RegDst   : out  STD_LOGIC;
71                 Branch   : out  STD_LOGIC;
72                 MemtoReg : out  STD_LOGIC;
73                 ALUSrc   : out  STD_LOGIC;
74                 ALUOp    : out  STD_LOGIC_VECTOR (1 downto 0);
75                 Jump     : out STD_LOGIC;
76                 RegWrite : out  STD_LOGIC);
77
78      end component;
79
80
```

```
80
81
82      component RegisterFile is
83          Port ( read_sel1  : in  STD_LOGIC_VECTOR (4 downto 0);
84                 read_sel2  : in  STD_LOGIC_VECTOR (4 downto 0);
85                 write_sel  : in  STD_LOGIC_VECTOR (4 downto 0);
86                 write_ena  : in  STD_LOGIC;
87                 clk        : in  STD_LOGIC;
88                 write_data : in  STD_LOGIC_VECTOR (31 downto 0);
89                 data1      : out STD_LOGIC_VECTOR (31 downto 0);
90                 data2      : out STD_LOGIC_VECTOR (31 downto 0));
91      end component;
92
93
94
95      component ShiftLeft2 is
96          Port ( Input  : in   STD_LOGIC_VECTOR (31 downto 0);
97                 Output : out  STD_LOGIC_VECTOR (31 downto 0));
98      end component;
99
100
101
102     component SignExtend is
103         Port ( Input  : in  STD_LOGIC_VECTOR (15 downto 0);
104                Output : out STD_LOGIC_VECTOR (31 downto 0));
105     end component;
106
107     component PC is
108         Port(
109             signal CLK, RESET : in std_logic;
110             signal Address : in std_logic_vector(31 downto 0);
111             signal PC : out std_logic_vector(31 downto 0)
112         );
113     end component;
114
115     component Adder4 is
116        Port ( Input  : in   STD_LOGIC_VECTOR (31 downto 0);
117               Output : out  STD_LOGIC_VECTOR (31 downto 0));
118     end component;
119
120     component JumpAddressConcat is
121         Port ( InputLeft  : in   STD_LOGIC_VECTOR (31 downto 0);
122                InputRight  : in   STD_LOGIC_VECTOR (31 downto 0);
123                Output : out  STD_LOGIC_VECTOR (31 downto 0));
124     end component;
```

```
125
126     component Mux5x1 is
127         Port ( S      : in   STD_LOGIC;
128                i0     : in  STD_LOGIC_VECTOR (4 downto 0);
129                i1     : in  STD_LOGIC_VECTOR (4 downto 0);
130                output : out  STD_LOGIC_VECTOR (4 downto 0));
131     end component;
132 ----------------------------------------------------------start signals----
133
134 --previous instruction address in PC
135 signal instructionAddress  : std_logic_vector(31 downto 0) := X"00000000";
136 signal InstrAddressAdd4    : std_logic_vector(31 downto 0) := X"00000000";
137
138 --related to jump process
139 signal instructionShifted  : std_logic_vector(31 downto 0) := X"00000000";
140 signal jumpInstrAddress    : std_logic_vector(31 downto 0) := X"00000000";
141
142 --related to branch process
143 signal branchInstAddress   : std_logic_vector(31 downto 0) := X"00000000";
144 signal immedValue16        : std_logic_vector(31 downto 0) := X"00000000";
145 signal branchValueShifted  : std_logic_vector(31 downto 0) := X"00000000";
146 signal intermediateAddress : std_logic_vector(31 downto 0) := X"00000000";
147
148 --final new instruction address result
149 signal newInstrAddress     : std_logic_vector(31 downto 0) := X"00000000";
150
151 --instruction from INSTMEMORY
152 signal instruction         : std_logic_vector(31 downto 0) := X"00000000";
153
154 -------------------Main control unit signals---------------------
155 signal MemRead             : std_logic := '0';
156 signal MemWrite            : std_logic := '0';
157 signal RegDst              : std_logic := '0';
158 signal Branch              : std_logic := '0';
159 signal MemtoReg            : std_logic := '0';
160 signal ALUSrc              : std_logic := '0';
161 signal ALUOp               : std_logic_vector(1 downto 0) := "00";
162 signal Jump                : std_logic := '0';
163 signal RegWrite            : std_logic := '0';
164
165 -------------------Register unit signals---------------------
166 signal WriteRegister       : std_logic_vector(4 downto 0) := "00000";
167 signal WriteData           : std_logic_vector(31 downto 0) := X"00000000";
168 signal ReadData1           : std_logic_vector(31 downto 0) := X"00000000";
169 signal ReadData2           : std_logic_vector(31 downto 0) := X"00000000";
170
```

```
165  --------------------Register unit signals----------------------
166  signal WriteRegister       : std_logic_vector(4 downto 0) := "00000";
167  signal WriteData           : std_logic_vector(31 downto 0) := X"00000000";
168  signal ReadData1           : std_logic_vector(31 downto 0) := X"00000000";
169  signal ReadData2           : std_logic_vector(31 downto 0) := X"00000000";
170
171
172  -------------------ALU unit & control signals----------------
173  signal ALUFunct            : std_logic_vector(3 downto 0)  := "0000";
174  signal ALUZero             : std_logic  := '0';
175  signal ALUMuxInput         : std_logic_vector(31 downto 0) := X"00000000";
176  signal ALUOutTemp          : std_logic_vector(31 downto 0) := X"00000000";
177
178  -------------------Data Memory signals-----------------
179  signal DataMemOutTemp            : std_logic_vector(31 downto 0) := X"00000000";
180
181
182
183  --open signal for later adding
184
185  signal BranchingSignal : std_logic :='0';
186
187
188  ------------------------------------------------------------end signals-------------
189
190  end MIPSPackage;
191
192  package body MIPSPackage is
193  end MIPSPackage;
194
```

Coming to the final stage, we created a main module named “MIPSCPU” and included the “MIPSPackage” where the main module will be completed by making all the connections between all the previous modules according to the connections drawn in diagram of the CPU provided in the major task pdf.

At the beginning, we declared all the inputs and outputs exactly as mentioned in the major task pdf and as can be seen in the upcoming screenshot, however, we decided to add two extra outputs “ReadReg1O” and “ReadReg2O” to display the address of the registers that are used in the reading process in the current instruction in order to ease the output tracing at the end in the simulation.

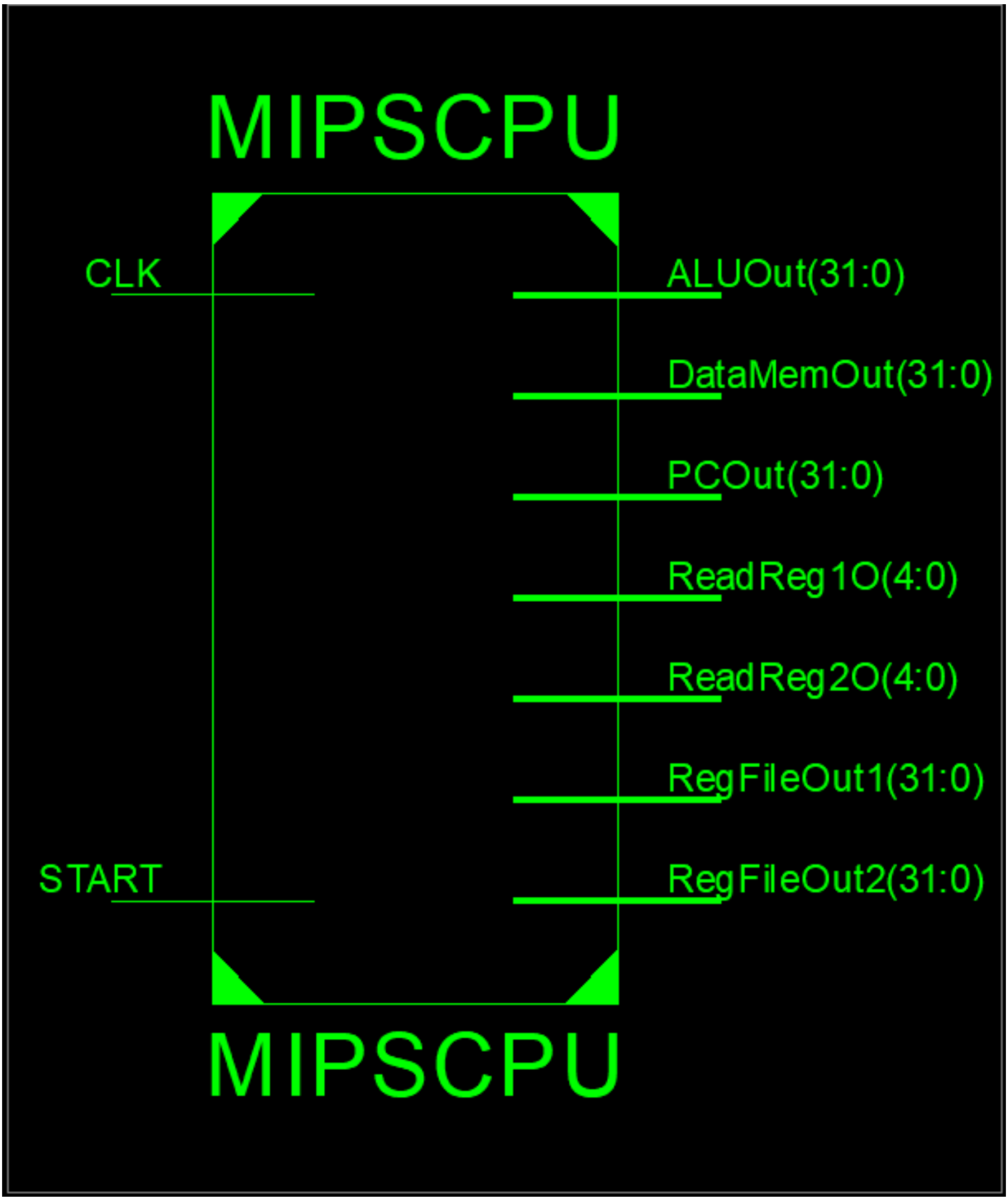
```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use work.MIPSPackage.all;
4
5  entity MIPSCPU is
6      Port ( START       : in  STD_LOGIC;
7             CLK         : in  STD_LOGIC;
8             RegFileOut1 : out STD_LOGIC_VECTOR (31 downto 0);
9             RegFileOut2 : out STD_LOGIC_VECTOR (31 downto 0);
10            ALUOut      : out STD_LOGIC_VECTOR (31 downto 0);
11            PCOut       : out STD_LOGIC_VECTOR (31 downto 0);
12            DataMemOut  : out STD_LOGIC_VECTOR (31 downto 0);
13
14 ---------------------------The previous outputs are the ones requ
15 --The next outputs are added as extras by us to verify that the f
16
17            ReadReg1O   : out STD_LOGIC_VECTOR (4 downto 0);
18            ReadReg2O   : out STD_LOGIC_VECTOR (4 downto 0)
19            );
20 end MIPSCPU;
21
```

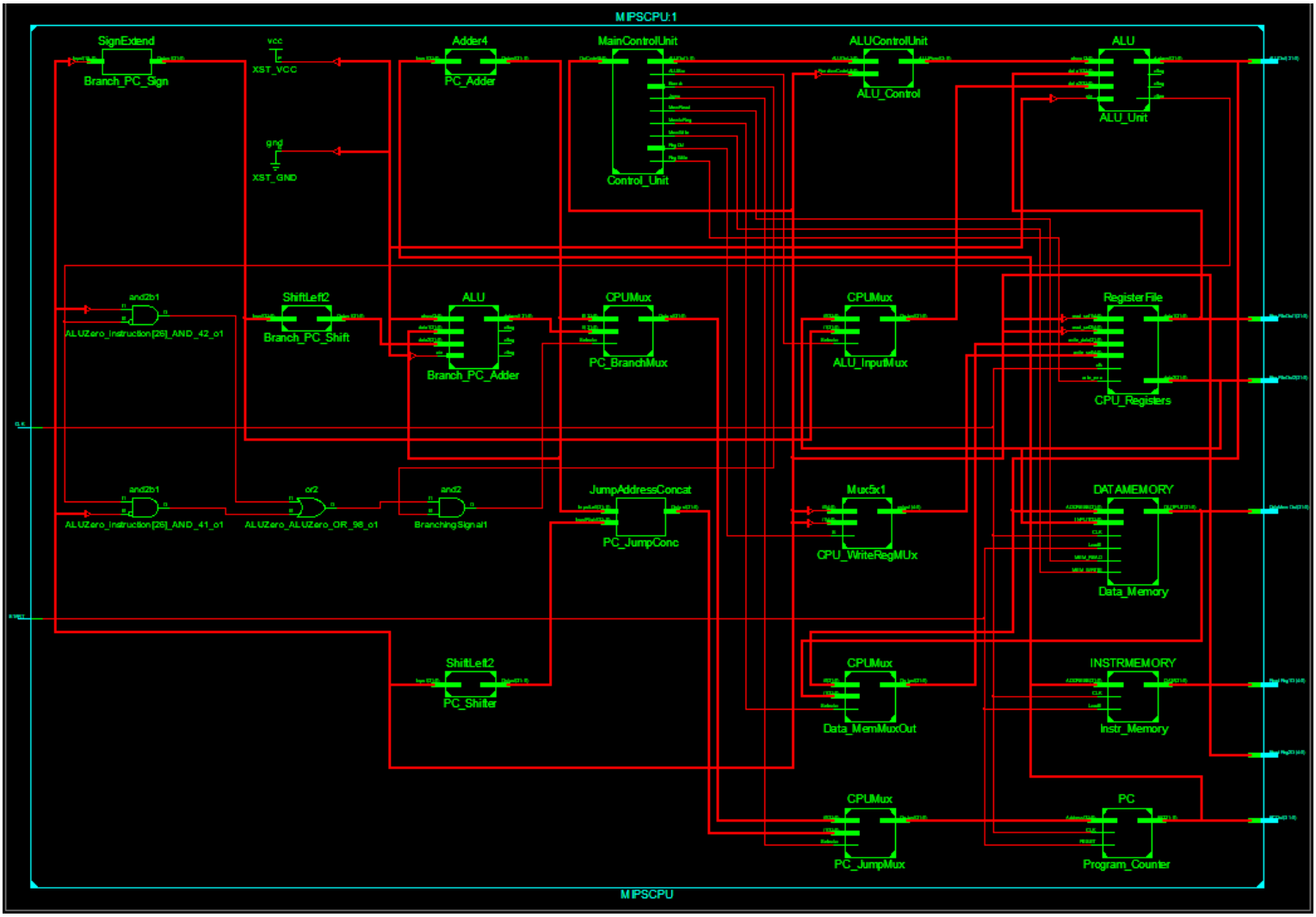
Next, we used the previously created signals and components found in "MIPSPackage" to make the connections between the modules like the diagram of the CPU given in the major task pdf with just a small modification for adding the "bne" instruction. So we wrote the port maps using the created signals when needed and any output that is found in one of the components but wont be used in the main module was assigned to "open". At the end, we assigned the outputs of the main module called "MIPSCPU" to their equivalent signals.

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

entity MIPSCPU is
    Port ( START       : in  STD_LOGIC;
           CLK         : in  STD_LOGIC;
           RegFileOut1 : out STD_LOGIC_VECTOR (31 downto 0);
           RegFileOut2 : out STD_LOGIC_VECTOR (31 downto 0);
           ALUOut      : out STD_LOGIC_VECTOR (31 downto 0);
           PCOut       : out STD_LOGIC_VECTOR (31 downto 0);
           DataMemOut  : out STD_LOGIC_VECTOR (31 downto 0);

----------------------------The previous outputs are the ones requested in the Major Task PDF---------------------------------
--The next outputs are added as extras by us to verify that the fibonnaci sequence appears in $s0 whose decimal value is 16--

           ReadReg1O   : out STD_LOGIC_VECTOR (4 downto 0);
           ReadReg2O   : out STD_LOGIC_VECTOR (4 downto 0)
           );
end MIPSCPU;

architecture Behavioral of MIPSCPU is
----------------------Signals used in this module is found in the package----------------------
begin



-----------------PC related operations---------------------------------
PC_Adder        : Adder4            port map (Input => instructionAddress, Output => InstrAddressAdd4);

--Branch related
Branch_PC_Sign  : SignExtend        port map (Input => instruction(15 downto 0), Output => immedValue16);
Branch_PC_Shift : ShiftLeft2        port map (Input => immedValue16, Output => 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    : CPUMux            port map (Selector => BranchingSignal, i0 => InstrAddressAdd4, i1 => branchInstAddress
                                             , Output => intermediateAddress);
--Jump related
PC_Shifter      : ShiftLeft2        port map (Input => instruction, Output => instructionShifted);
PC_JumpConc     : JumpAddressConcat port map (InputLeft => InstrAddressAdd4, InputRight => instructionShifted
                                             , Output => jumpInstrAddress);
PC_JumpMux      : CPUMux            port map (Selector => Jump, i0 => intermediateAddress, i1 => jumpInstrAddress
                                             , Output => newInstrAddress);

--PC final impact
Program_Counter : PC                port map (CLK => CLK, RESET => START , Address => newInstrAddress, PC => instructionAddress);
```

```
48
49
50  --------------Instruction memory unit----------------------------------------
51
52  Instr_Memory    : INSTRMEMORY       port map (LoadIt => START,  DATA => instruction,  ADDRESS => instructionAddress, CLK => CLK);
53
54
55  --------------Main Control Unit-----------------------------------------
56
57  Control_Unit    : MainControlUnit   port map (OpCode => instruction(31 downto 26), MemRead => MemRead, MemWrite => MemWrite
58                                              , RegDst =>RegDst, Branch => Branch, MemtoReg => MemtoReg, ALUSrc => ALUSrc
59                                              , ALUOp => ALUOp,  Jump => Jump, RegWrite => RegWrite);
60
61  --------------Registers unit--------------------------------------------
62  CPU_WriteRegMUx : Mux5x1            port map (S => RegDst, i0 => instruction(20 downto 16), i1 => instruction(15 downto 11)
63                                              , output => WriteRegister);
64
65  CPU_Registers   : RegisterFile      port map (read_sel1 => instruction(25 downto 21), read_sel2 => instruction(20 downto 16)
66                                              , write_sel => WriteRegister, write_ena => RegWrite, clk => CLK
67                                              , write_data => WriteData, data1 => ReadData1, data2 =>  ReadData2);
68
69  --------------ALU & ALU Control unit-----------------------------------------
70
71  ALU_Control     : ALUControlUnit    port map (FunctionCode => instruction(5 downto 0), ALUOp => ALUOp, ALUFunct => ALUFunct);
72
73  ALU_InputMux    : CPUMux            port map (Selector => ALUSrc, i0 => ReadData2, i1 => immedValue16
74                                              , Output => ALUMuxInput);
75
76  ALU_Unit        : ALU               port map (data1 => ReadData1, data2 => ALUMuxInput,  aluop => ALUFunct,  cin => '0'
77                                              , dataout => ALUOutTemp, cflag => open, zflag => ALUZero, oflag =>open );
78  --------------Data memory unit-----------------------------------------
79
80  Data_Memory    : DATAMEMORY         port map (LoadIt => START, INPUT => ReadData2,   OUTPUT => DataMemOutTemp
81                                              , MEM_READ => MemRead, MEM_WRITE => MemWrite, ADDRESS => ALUOutTemp,  CLK => CLK);
82  Data_MemMuxOut : CPUMux             port map (Selector => MemtoReg, i0 => ALUOutTemp, i1 => DataMemOutTemp
83                                              , Output => WriteData);
84
85  --------------Outputs----------------------------------------
86   RegFileOut1 <= ReadData1;
87   RegFileOut2 <= ReadData2;
88   ALUOut      <= ALUOutTemp;
89   PCOut       <= instructionAddress;
90   DataMemOut  <= DataMemOutTemp;
91   ReadReg1O <= instruction(25 downto 21);
92   ReadReg2O <= instruction(20 downto 16);
93
94  end Behavioral;
```

The MIPS CPU RTL Schematic




MIPSCPU:1
SignExtend
Branch_PC_Sign
vcc
XST_VCC
gnd
XST_GND
Adder4
PC_Adder
MainControlUnit
Control_Unit
ALUControlUnit
ALU_Control
ALU
ALU_Unit
and2b1
ALUZero_Instruction[26]_AND_42_o1
ShiftLeft2
Branch_PC_Shift
ALU
Branch_PC_Adder
CPUMux
PC_BranchMux
CPUMux
ALU_InputMux
RegisterFile
CPU_Registers
and2b1
ALUZero_Instruction[26]_AND_41_o1
or2
ALUZero_ALUZero_OR_98_o1
and2
BranchingSignal1
JumpAddressConcat
PC_JumpConc
Mux5x1
CPU_WriteRegMUx
DATAMEMORY
Data_Memory
ShiftLeft2
PC_Shifter
CPUMux
Data_MemMuxOut
INSTRMEMORY
Instr_Memory
CPUMux
PC_JumpMux
PC
Program_Counter
MIPSCPU

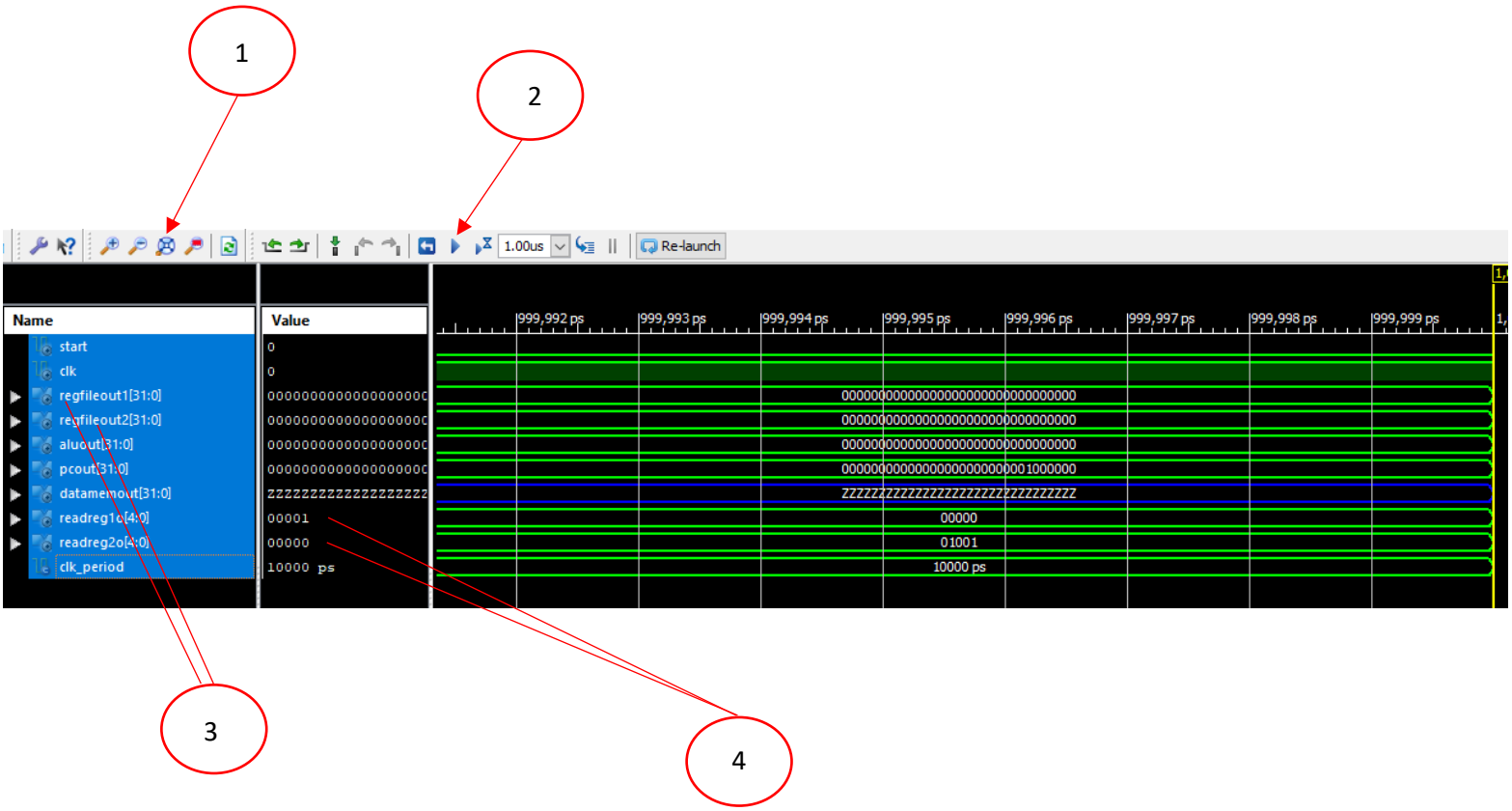
The MIPS CPU Testbench Code

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;


ENTITY test_instr IS
END test_instr;

ARCHITECTURE behavior OF test_instr IS

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

    COMPONENT MIPSCPU
    PORT(
         START : IN  std_logic;
         CLK : IN  std_logic;
         RegFileOut1 : OUT  std_logic_vector(31 downto 0);
         RegFileOut2 : OUT  std_logic_vector(31 downto 0);
         ALUOut : OUT  std_logic_vector(31 downto 0);
         PCOut : OUT  std_logic_vector(31 downto 0);
         DataMemOut : OUT  std_logic_vector(31 downto 0);
         ReadReg1O     : out STD_LOGIC_VECTOR (4 downto 0);
         ReadReg2O     : out STD_LOGIC_VECTOR (4 downto 0)

        );
    END COMPONENT;


   --Inputs
   signal START : std_logic := '0';
   signal CLK : std_logic := '0';

   --Outputs
   signal RegFileOut1 : std_logic_vector(31 downto 0);
   signal RegFileOut2 : std_logic_vector(31 downto 0);
   signal ALUOut : std_logic_vector(31 downto 0);
   signal PCOut : std_logic_vector(31 downto 0);
   signal DataMemOut : std_logic_vector(31 downto 0);
   signal ReadReg1O : std_logic_vector(4 downto 0);
   signal ReadReg2O : std_logic_vector(4 downto 0);


   -- Clock period definitions
   constant CLK_period : time := 10 ns;
```

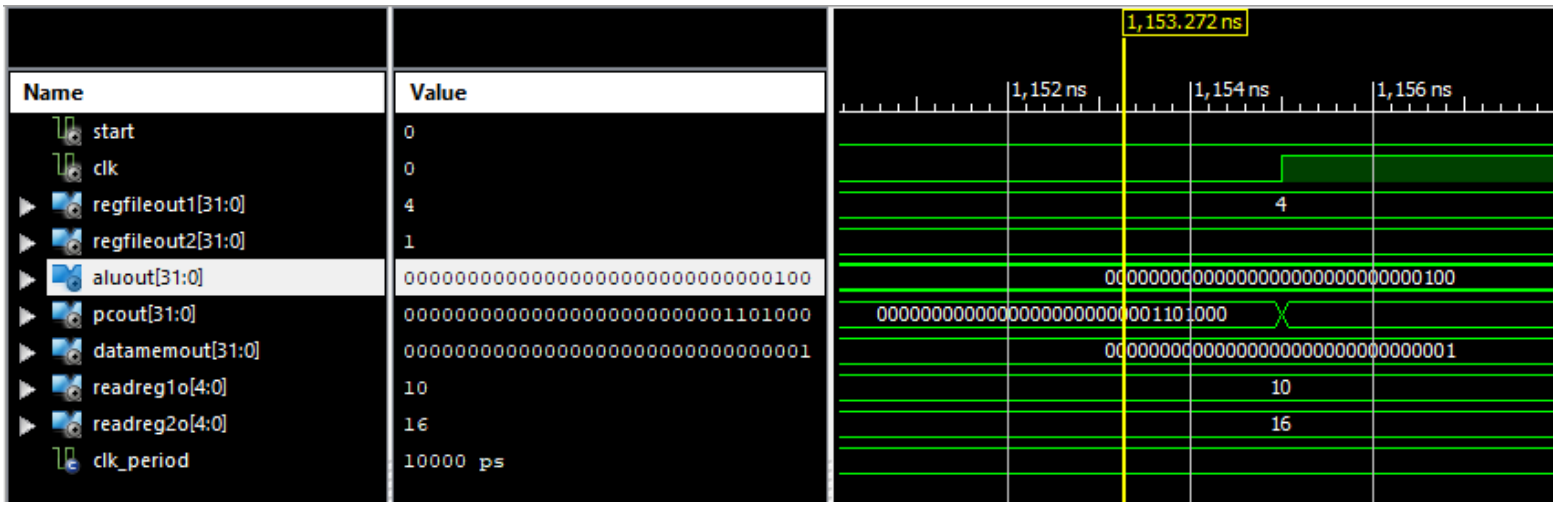
```
44
45  BEGIN
46
47      -- Instantiate the Unit Under Test (UUT)
48      uut: MIPSCPU PORT MAP (
49              START => START,
50              CLK => CLK,
51              RegFileOut1 => RegFileOut1,
52              RegFileOut2 => RegFileOut2,
53              ALUOut => ALUOut,
54              PCOut => PCOut,
55              DataMemOut => DataMemOut,
56              ReadReg1O => ReadReg1O,
57              ReadReg2O => ReadReg2O
58
59
60
61          );
62
63      -- Clock process definitions
64      CLK_process :process
65      begin
66          CLK <= '0';
67          wait for CLK_period/2;
68          CLK <= '1';
69          wait for CLK_period/2;
70      end process;
71
72
73      -- Stimulus process
74      stim_proc: process
75      begin
76
77          START <= '1';
78          WAIT for 30 ns;
79          START <= '0';
80          WAIT for 30ns;
81
82
83          wait;
84      end process;
85
86  END;
87
```

The MIPS CPU Sample Output

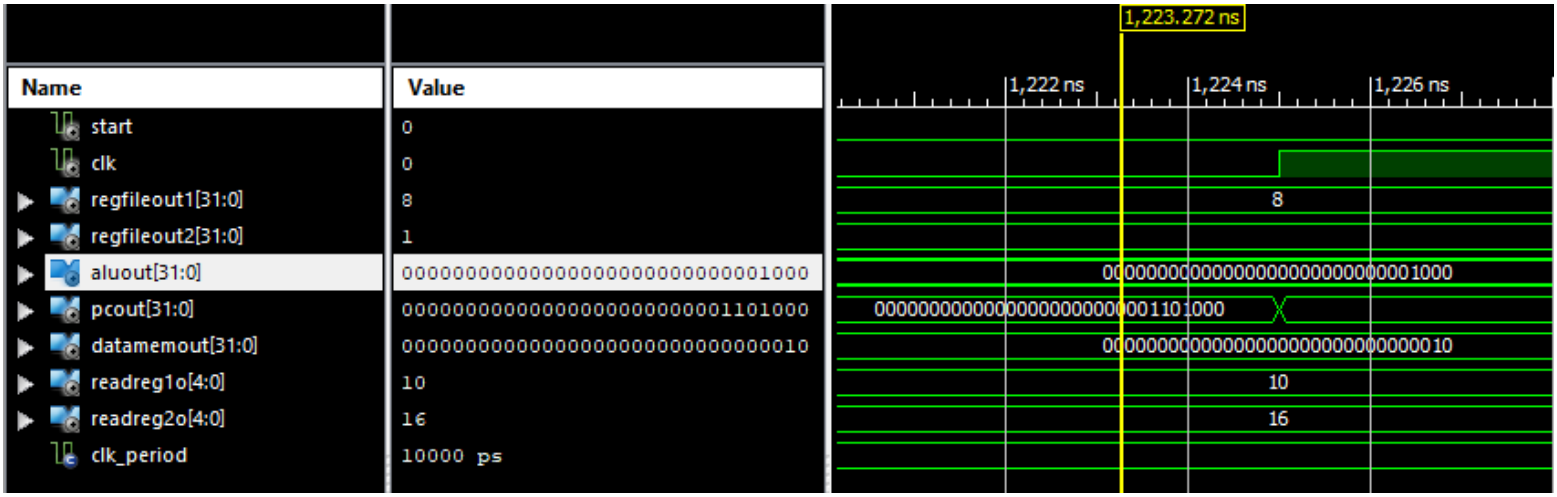
At the beginning of the simulation, a few steps should be done in order to see the output and making the tracing process easier. Step 1 is to fit the page to the screen. Step 2 is to press the "Run All" button in order to display all the program including the output because by default it stops at 1000 ns which is not sufficient. Step 3 is to change the radix of "regfileout1" and "regfileout2" to signed decimal in order to read the Fibonacci sequence in decimals which is easier. Step 4 is to change the radix of the two extra outputs of "readreg1o" and "readreg2o" to unsigned decimal to display the address of the registers in use (the address of the registers does not use the sign) to trace the outputs of the required register $s0 by checking the values in the registers output when either of the "readreg1o" or "readreg2o" is equal to "16" which is the decimal number of "10000" which is the address of $s0 in MIPS.



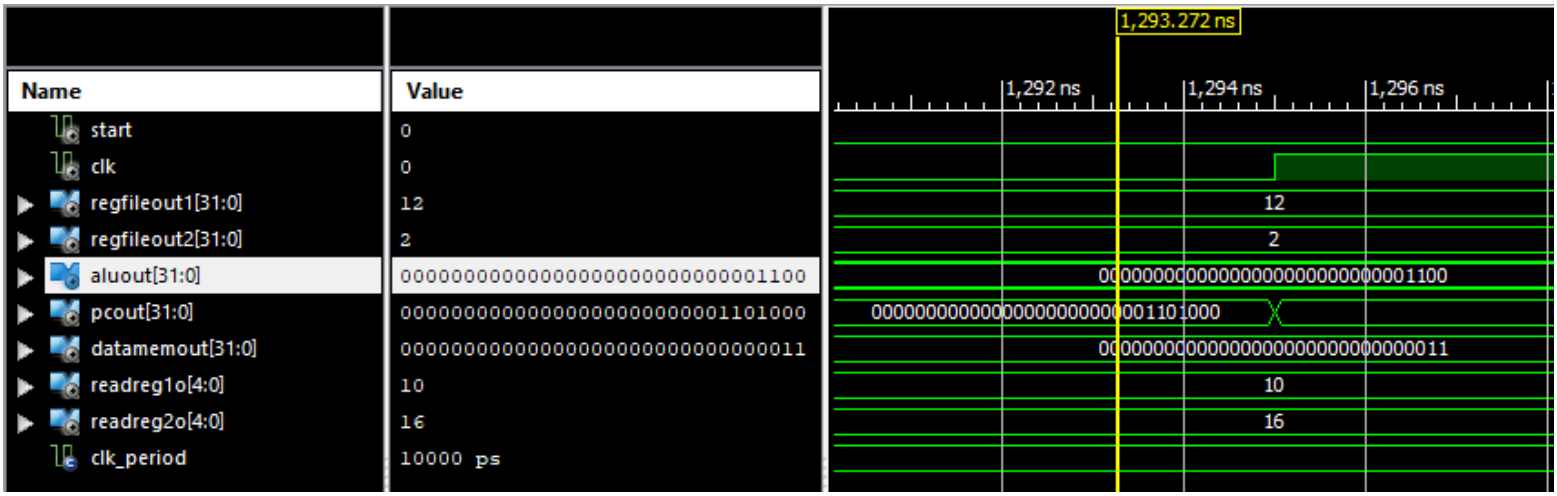
For approximately the first 1100 – 1150 ns, this was the time related to the first instructions of calculating actually the values and the other needed instructions, but from nearly the 1150 ns till the end, this was the time of the loop outputting the sequence of Fibonacci. Accordingly, the output of the sequence could be seen starting from nearly 1150 ns.



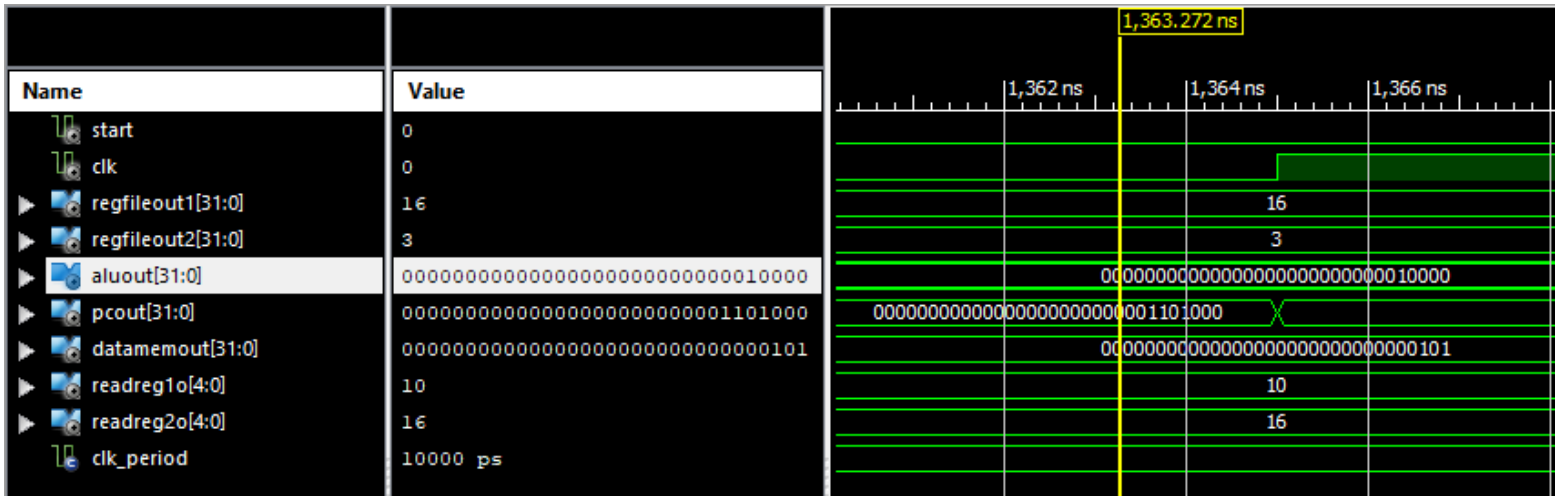
At 1153.272 ns, the program started in executing the loop related to outputting the Fibonacci sequence. As it can be seen, when “readreg2o” is equal to “16” which is the address of register $s0, “regfileout2” had ‘1’ as its output which is the first number in the Fibonacci sequence.



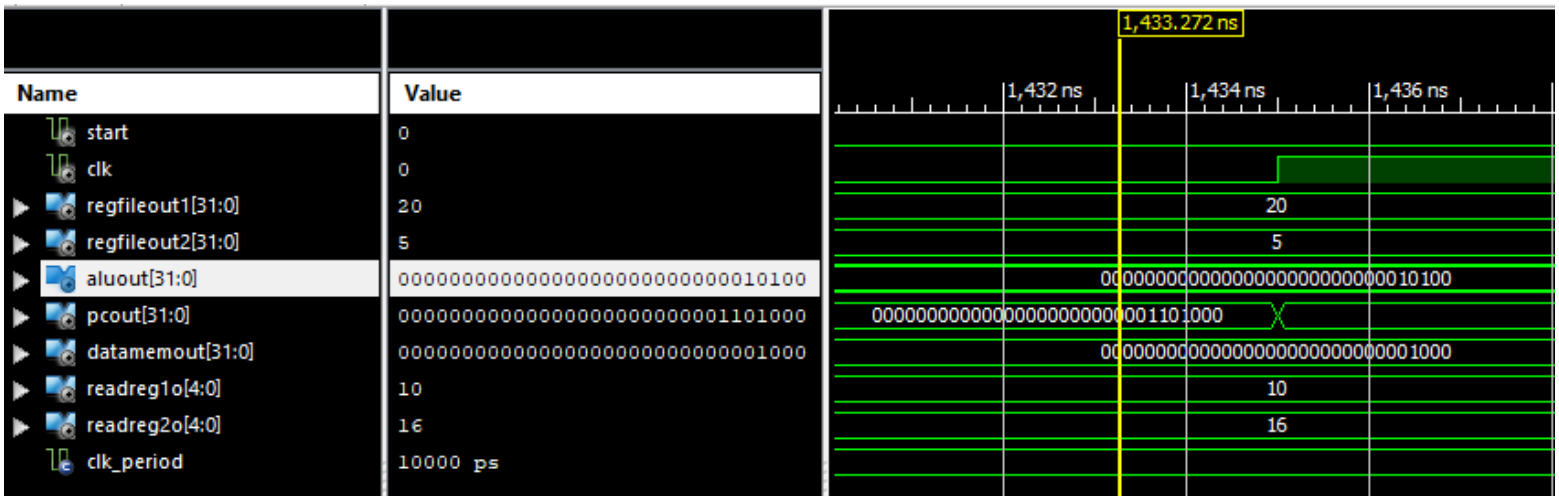
Proceeding forward with 70 more nanoseconds, at 1223.272 ns and as seen, when “readreg2o” is equal to “16” which is the address of register $s0, “regfileout2” had ‘1’ as its output which is the second number in the Fibonacci sequence.



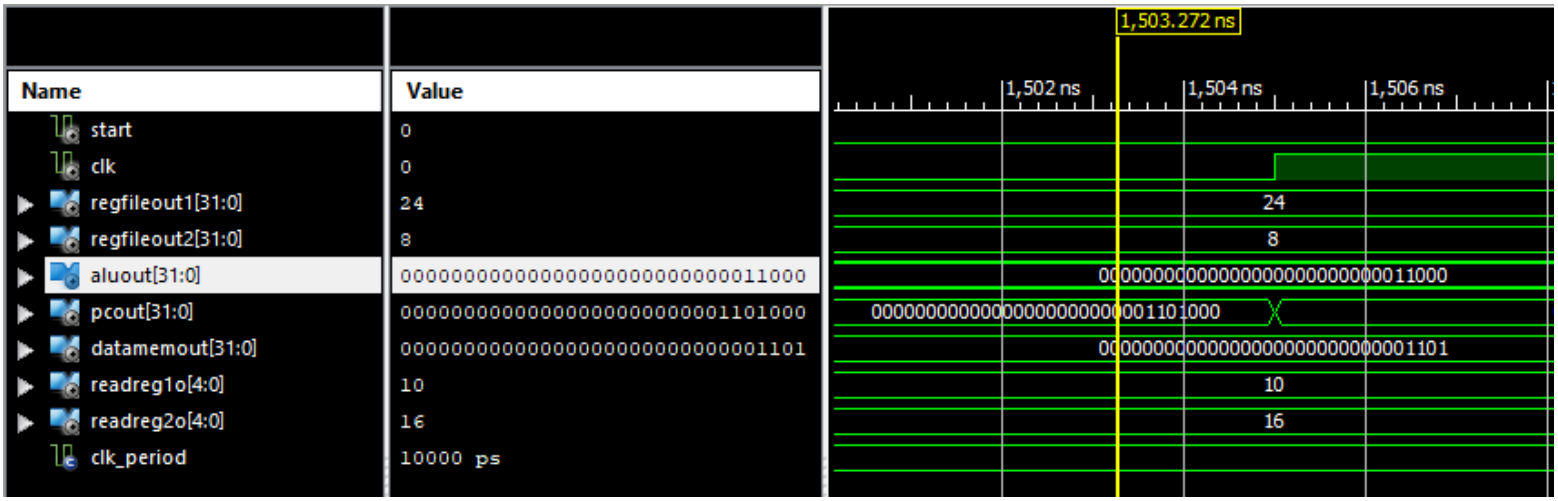
Proceeding forward with 70 more nanoseconds, at 1293.272 ns and as seen, when “readreg2o” is equal to “16” which is the address of register $s0, “regfileout2” had ‘2’ as its output which is the third number in the Fibonacci sequence.



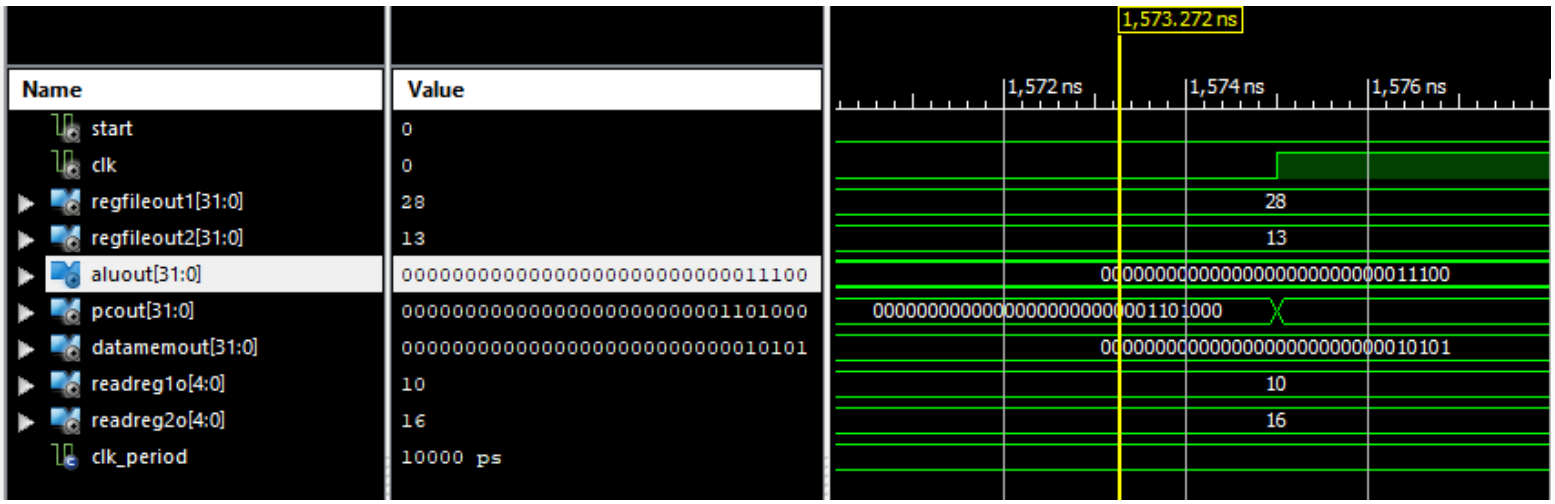
Proceeding forward with 70 more nanoseconds, at 1363.272 ns and as seen, when "readreg2o" is equal to "16" which is the address of register $s0, "regfileout2" had '3' as its output which is the fourth number in the Fibonacci sequence.



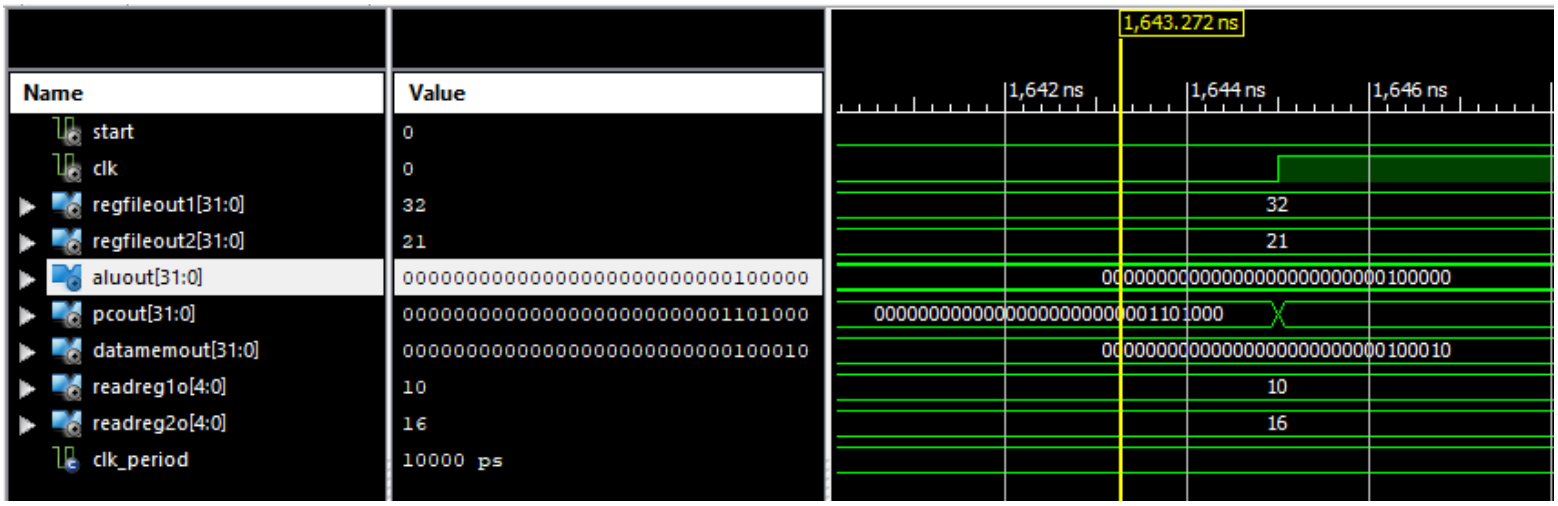
Proceeding forward with 70 more nanoseconds, at 1433.272 ns and as seen, when "readreg2o" is equal to "16" which is the address of register $s0, "regfileout2" had '5' as its output which is the fifth number in the Fibonacci sequence.



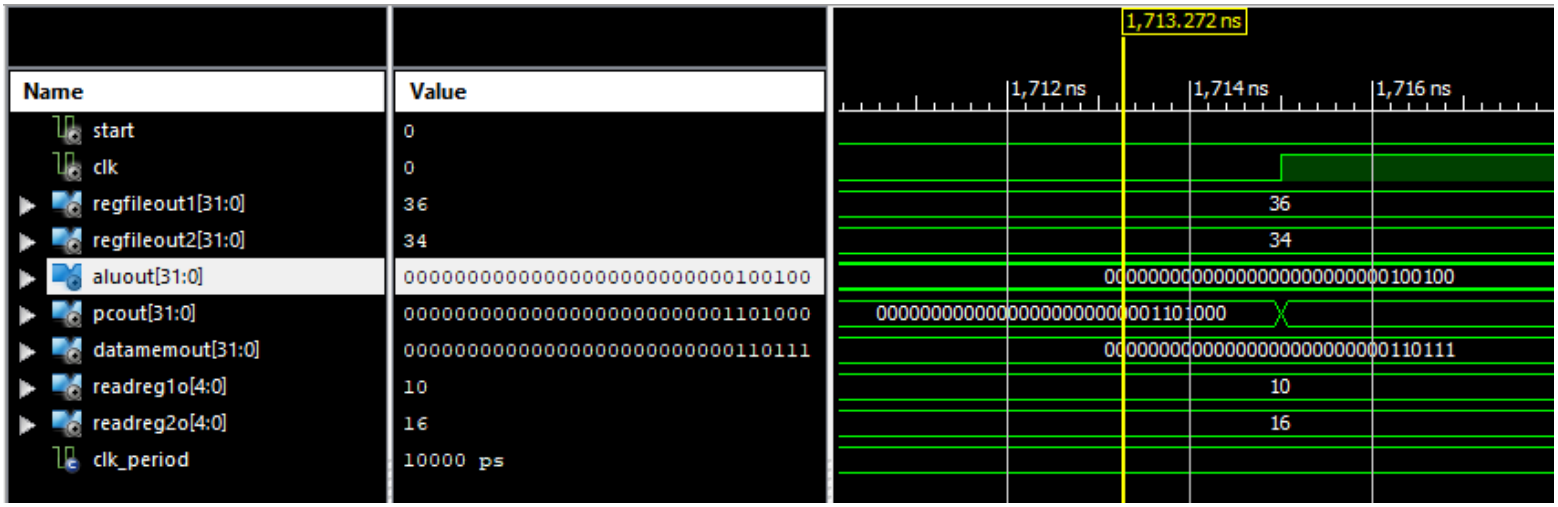
Proceeding forward with 70 more nanoseconds, at 1503.272 ns and as seen, when "readreg2o" is equal to "16" which is the address of register $s0, "regfileout2" had '8' as its output which is the sixth number in the Fibonacci sequence.



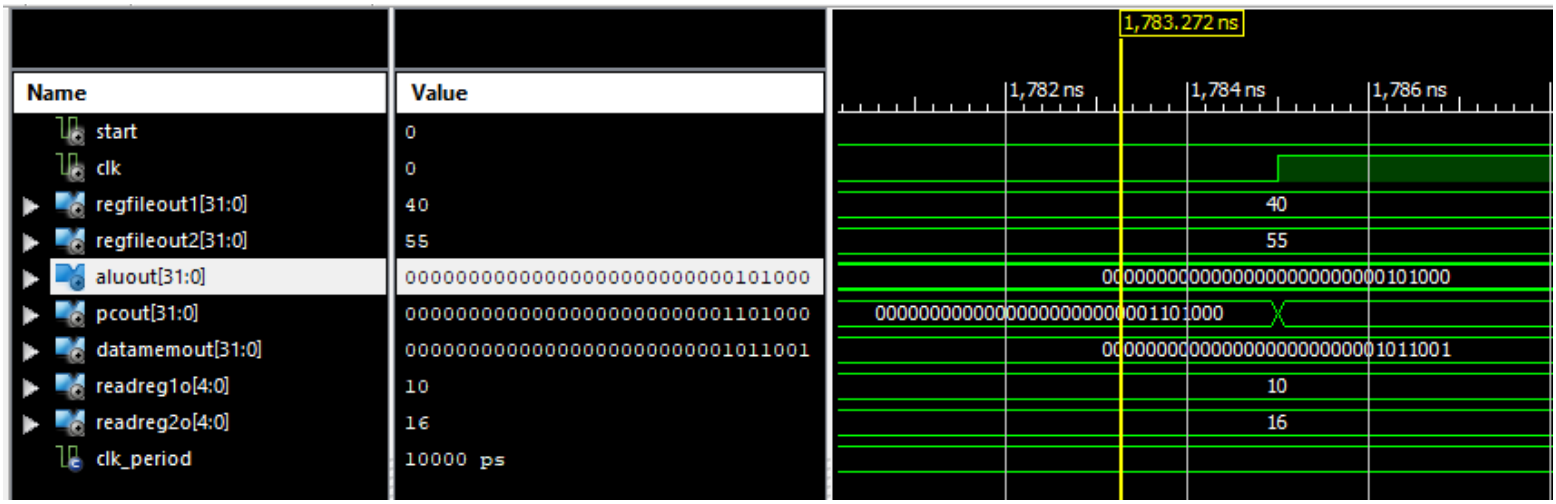
Proceeding forward with 70 more nanoseconds, at 1573.272 ns and as seen, when "readreg2o" is equal to "16" which is the address of register $s0, "regfileout2" had "13" as its output which is the seventh number in the Fibonacci sequence.



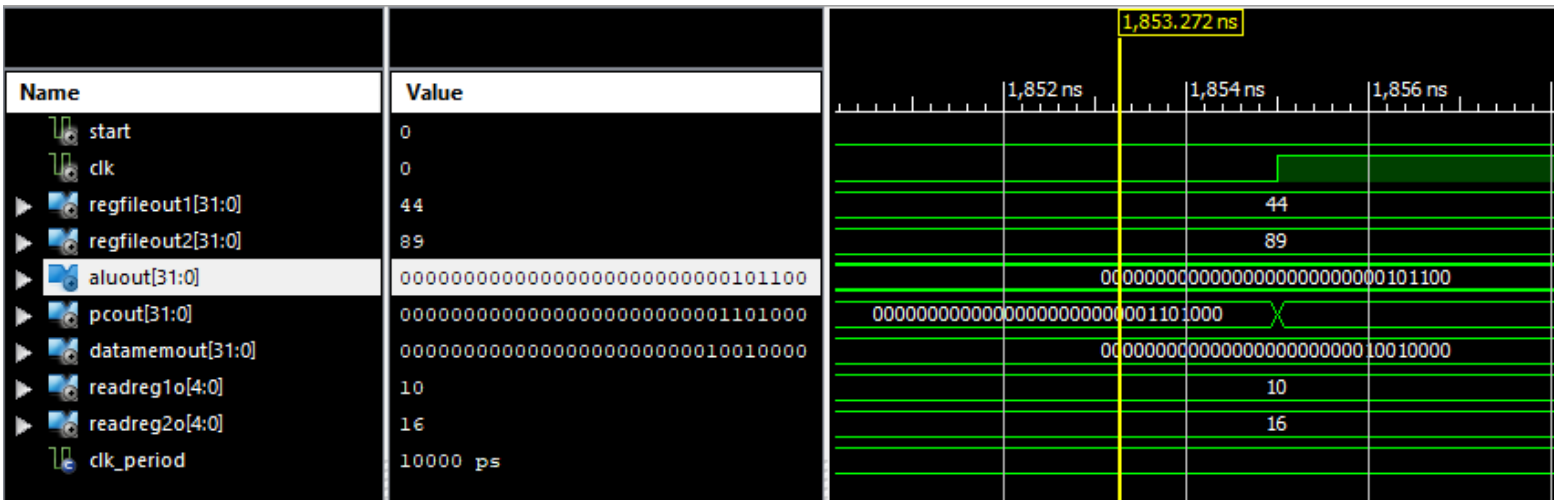
Proceeding forward with 70 more nanoseconds, at 1643.272 ns and as seen, when “readreg2o” is equal to “16” which is the address of register $s0, “regfileout2” had “21” as its output which is the eighth number in the Fibonacci sequence.



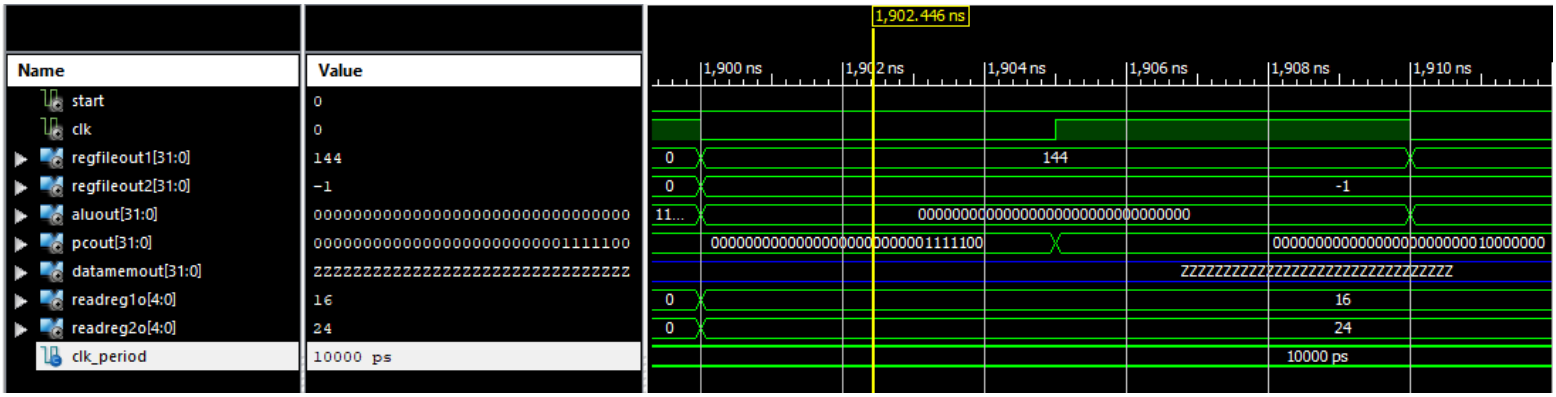
Proceeding forward with 70 more nanoseconds, at 1713.272 ns and as seen, when “readreg2o” is equal to “16” which is the address of register $s0, “regfileout2” had “34” as its output which is the ninth number in the Fibonacci sequence.



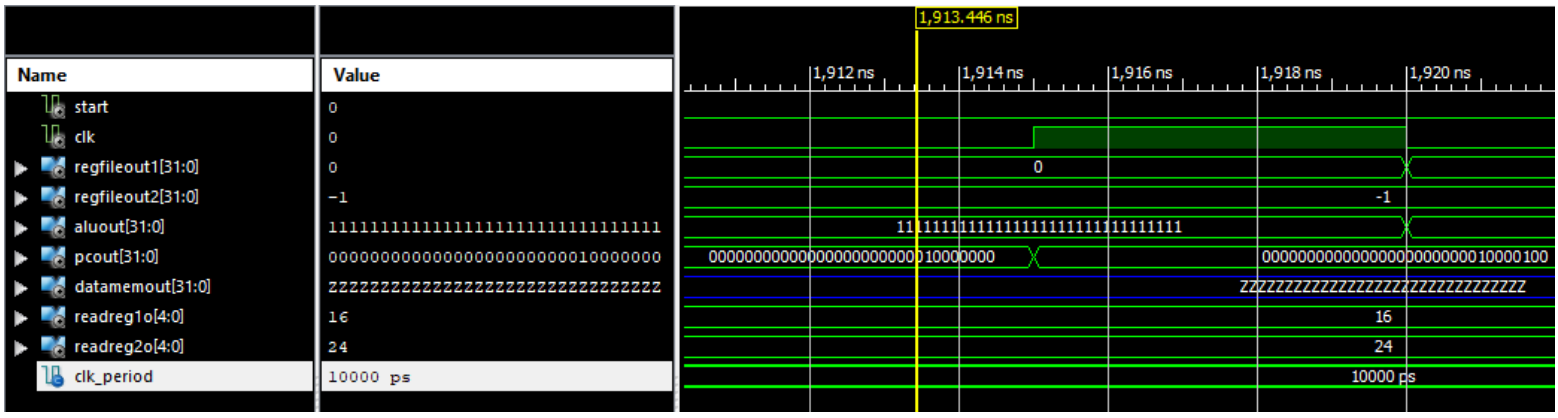
Proceeding forward with 70 more nanoseconds, at 1783.272 ns and as seen, when “readreg2o” is equal to “16” which is the address of register $s0, “regfileout2” had “55” as its output which is the tenth number in the Fibonacci sequence.



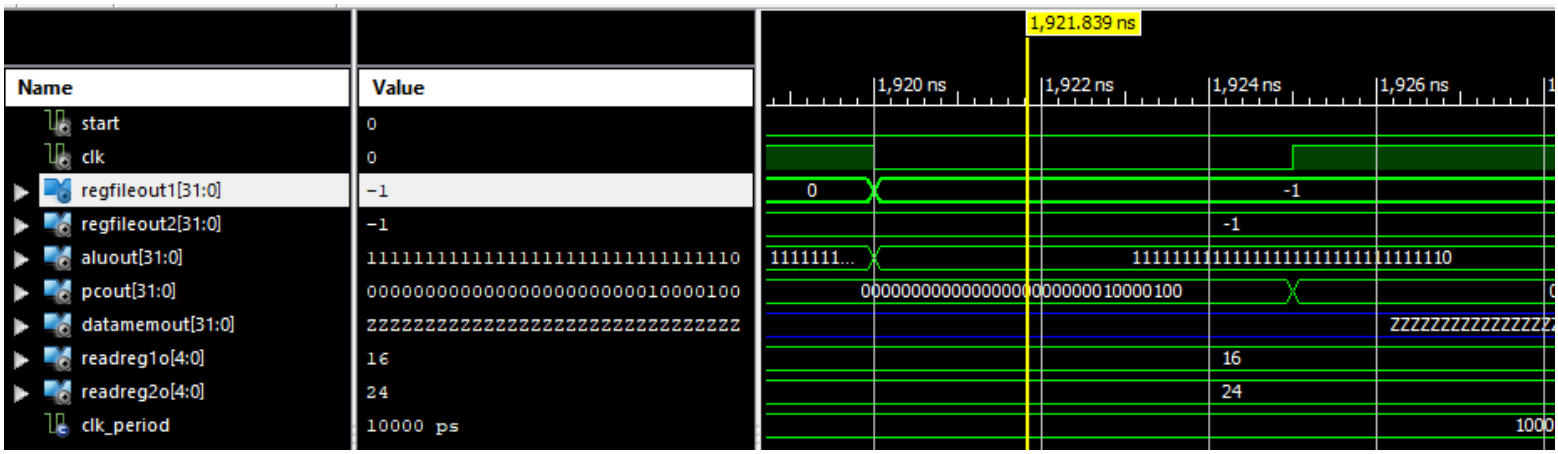
Proceeding forward with 70 more nanoseconds, at 1853.272 ns and as seen, when “readreg2o” is equal to “16” which is the address of register $s0, “regfileout2” had “89” as its output which is the eleventh number in the Fibonacci sequence.



Then at 1902.446 ns and as seen, when “readreg1o” is equal to “16” which is the address of register $s0, “regfileout1” had “144” as its output which is the twelfth number in the Fibonacci sequence.



Then at 1913.446 ns and as seen, when “readreg1o” is equal to “16” which is the address of register $s0, “regfileout1” had ‘0’ as its output which is the required number to appear after displaying the first twelve Fibonacci numbers.



Then at 1921.839 ns and as seen, when “readreg1o” is equal to “16” which is the address of register $s0, “regfileout1” had ‘-1’ as its output which is the final number required to appear in the output.

Team Members'

Equivalent Contribution

- Ahmed Hossam Moussa Sakr (20P1009) – Wrote the codes of: "ALUControlUnit", "CPUMux" and half the code of the "MainControlUnit" modules with overall contribution of 20% of the total effort.

- Mohamed Tarek Mohamed Abdalla Ahmed Khafagy (20P6211) – Wrote the codes of: "ShiftLeft2", "RegisterFile" and "TheDecoder" modules with overall contribution of 20% of the total effort.

- Ahmed Wael Samir Abdelmegied (20P7271) – Wrote the codes of: "ALU", "SignExtend" and "JumpAddressConcat" with overall contribution of 20% of the total effort.

- Omar Ahmed Mohamed Gamaleldin Swelam (21P0405) – Wrote the codes of: "MIPSCPU", "PC" and half the code of the "MainControlUnit" modules with overall contribution of 20% of the total effort.

- Tamer Ihab Mohamed Abdelwahab 20P5567 – Wrote the codes of: “RegDef”, “Mux32x1”, “Mux5x1” and “Adder4” modules with overall contribution of 20% of the total effort.

- All the members participated in the editing of the “INSTRMEMORY”, test benching the “ALU”, “RegisterFile” and “MIPSCPU”.