(CSE112) Computer Organization and Architecture

Sophomore CESS

SPRING 2022

Group ID (13)

Project Phase (1) Report

Name	ID
Ahmed Ehab Mohamed El-Baramony	21P0261
Ahmed Mohamed Mohamed	22P0283
Hesham Mohamed El-Afandy	21P0054

Table Of Figures:

Project Overview:

A design of MIPS processor using VHDL that illustrates a basic computer system by simulating the data and control paths.

Phase (1) Requirements:

1- Implement the MIPS register file that reads simultaneously from two registers and write into another. The implementation should follow the internal logic design of the register file:

- The main module should be called "RegisterFile"
- The entity should look as follows:
  - **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)

2- Modify the 32 bit full ALU.

- ALU functional specifications:

ALU Operation	Function
0000	AND
0001	OR
0010	ADD
0110	SUB
1100	NOR

- Entity should look as follows:
  - **data1** : in std_logic_vector(31 downto 0)
  - **data2** : in std_logic_vector(31 downto 0)
  - **aluop** : in std_logic_vector(3 downto 0)
  - **dataout** : out std_logic_vector(31 downto 0)
  - **zflag** : out std_logic

3- Connect the already-built modules including register file, ALU, to design a simple MIPS CPU Using VHDL. The proposed CPU should be able to perform certain instructions: R-type (**AND**, **OR**, **ADD**, **SUB** & **NOR**).
    - Datapath entity should look as follows:
        - **clk, reset**: in STD_LOGIC;
        - **instr**: in STD_LOGIC_VECTOR(31 downto 0);
        - **aluoperation**: in STD_LOGIC_VECTOR(2 downto 0);
        - **zero**: out STD_LOGIC;
        - **regwrite**: in STD_LOGIC;
        - **aluout** : buffer STD_LOGIC_VECTOR(31 downto 0));

Implementation:

Part (1):

First, implement the necessary components for the Register & ALU design and package them together for repeated use, including:

- Decoder 5x32
- MUX 32x1
- Flopr

Code:

Decoder 5x32

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

entity Decoder_5x32 is
    Port ( Selector : in  STD_LOGIC_VECTOR (4 downto 0);
           Data_Out : out  STD_LOGIC_VECTOR (31 downto 0));
end Decoder_5x32;

architecture Behavioral of Decoder_5x32 is

begin

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

end Behavioral;
```

MUX 32x1

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

entity MUX_32x1 is
    Port (  input_0 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_1 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_2 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_3 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_4 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_5 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_6 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_7 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_8 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_9 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_10 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_11 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_12 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_13 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_14 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_15 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_16 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_17 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_18 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_19 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_20 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_21 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_22 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_23 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_24 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_25 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_26 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_27 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_28 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_29 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_30 : in  STD_LOGIC_VECTOR (31 downto 0);
                input_31 : in  STD_LOGIC_VECTOR (31 downto 0);
                Selector : in  STD_LOGIC_VECTOR (4 downto 0);
                Data_Out : out  STD_LOGIC_VECTOR (31 downto 0));
end MUX_32x1;

architecture Behavioral of MUX_32x1 is

begin


Data_Out <= input_0 when Selector = "00000" else
                input_1 when Selector = "00001" else
                input_2 when Selector = "00010" else
                input_3 when Selector = "00011" else
                input_4 when Selector = "00100" else
                input_5 when Selector = "00101" else
                input_6 when Selector = "00110" else
                input_7 when Selector = "00111" else
                input_8 when Selector = "01000" else
                input_9 when Selector = "01001" else
                input_10 when Selector = "01010" else
                input_11 when Selector = "01011" else
                input_12 when Selector = "01100" else
                input_13 when Selector = "01101" else
                input_14 when Selector = "01110" else
                input_15 when Selector = "01111" else
                input_16 when Selector = "10000" else
                input_17 when Selector = "10001" else
                input_18 when Selector = "10010" else
                input_19 when Selector = "10011" else
                input_20 when Selector = "10100" else
                input_21 when Selector = "10101" else
                input_22 when Selector = "10110" else
                input_23 when Selector = "10111" else
                input_24 when Selector = "11000" else
                input_25 when Selector = "11001" else
                input_26 when Selector = "11010" else
                input_27 when Selector = "11011" else
                input_28 when Selector = "11100" else
                input_29 when Selector = "11101" else
                input_30 when Selector = "11110" else
                input_31 when Selector = "11111";

end Behavioral;
```

Flopr

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

entity Flopr is
    GENERIC (n: NATURAL := 32);
     Port ( CLK : in  STD_LOGIC;
           RST : in  STD_LOGIC;
           Load : in STD_LOGIC;
           D : in  STD_LOGIC_VECTOR (n-1 downto 0);
           Q : out  STD_LOGIC_VECTOR (n-1 downto 0));
end Flopr;

architecture Behavioral of Flopr is

begin

PROCESS (CLK,RST)
BEGIN

IF (RST = '1') THEN
Q <= (others => '0');

ELSIF (CLK' EVENT AND CLK ='1') THEN
  IF (Load = '1') THEN
    Q <= D;
  END IF;
END IF;
END PROCESS;

end Behavioral;
```

Package

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
package Project_Package is
--Flopr Implementation
component Flopr GENERIC(n:integer);
     Port ( CLK : in  STD_LOGIC;
           RST : in  STD_LOGIC;
           Load : in STD_LOGIC;
           D : in  STD_LOGIC_VECTOR (n-1 downto 0);
           Q : out  STD_LOGIC_VECTOR (n-1 downto 0));
end component;

--Decoder Implementation
component Decoder_5x32
    Port ( Selector : in  STD_LOGIC_VECTOR (4 downto 0);
           Data_Out : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

--MUX Implementation
component MUX_32x1
    Port (  input_0 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_1 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_2 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_3 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_4 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_5 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_6 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_7 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_8 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_9 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_10 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_11 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_12 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_13 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_14 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_15 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_16 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_17 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_18 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_19 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_20 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_21 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_22 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_23 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_24 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_25 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_26 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_27 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_28 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_29 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_30 : in  STD_LOGIC_VECTOR (31 downto 0);
            input_31 : in  STD_LOGIC_VECTOR (31 downto 0);
            Selector : in  STD_LOGIC_VECTOR (4 downto 0);
            Data_Out : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

end Project_Package;
```

RTL Diagrams:

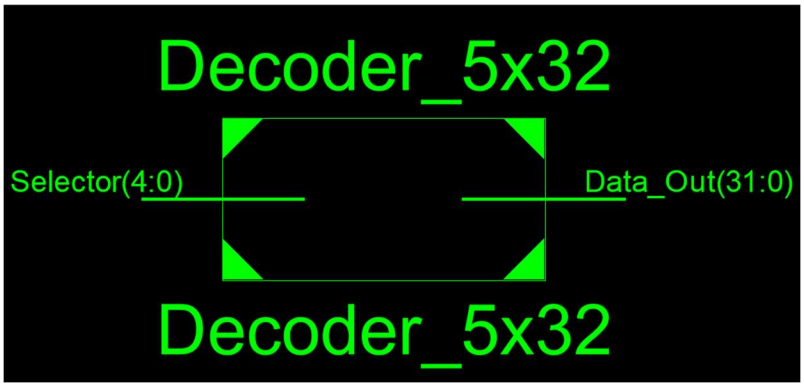

Figure 1: Decoder 5x32 RTL Diagram

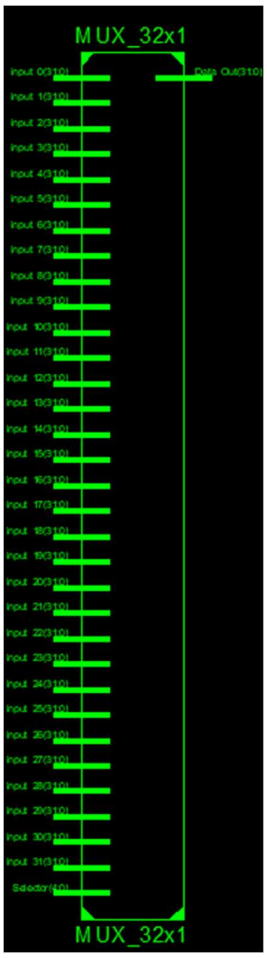

Figure 2: MUX 32x1 RTL Diagram

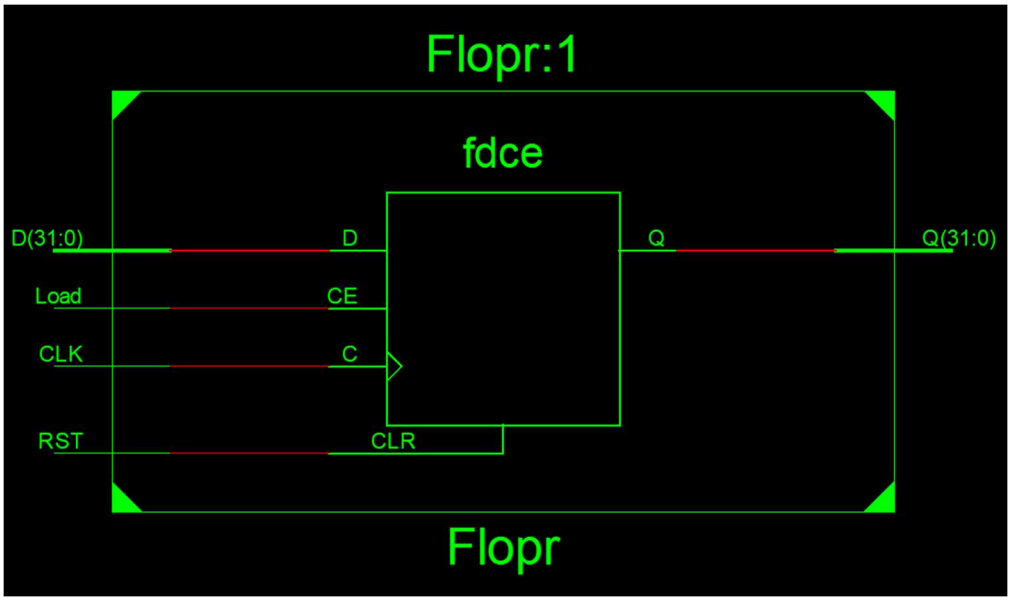

Figure 3: Flopr RTL Diagram

Part (2):

Designing the register file according to the following requirements the register should have both the ability to READ/WRITE in some memory locations which we use Floprs to implement.

Diagram:

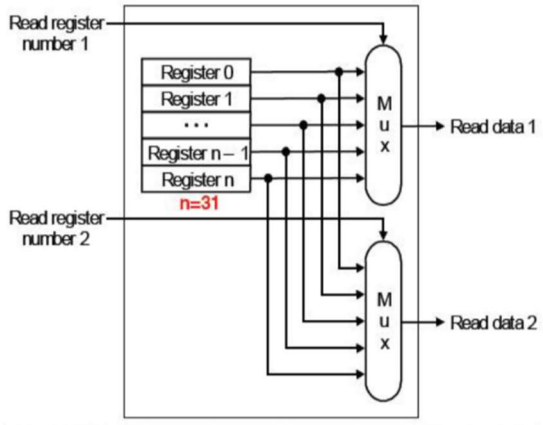

Figure 4: Read Function

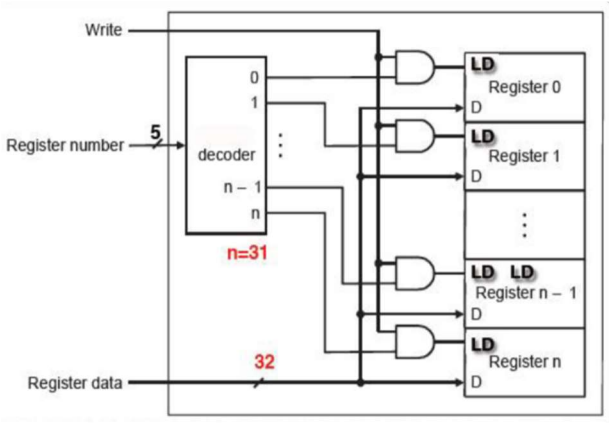

Figure 5: Write Function

Code:

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;
use work.Project_Package.ALL;

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

architecture Behavioral of Reg is

--Signal Declaration
signal decoder_output: STD_LOGIC_VECTOR (31 downto 0);

type reg_array_t is array (0 to 31) of STD_LOGIC_VECTOR(31 downto 0);
signal reg_outputs : reg_array_t;

begin

-------------------------------------------------------------------------------------------
-----------------------------------Write Implementation------------------------------------
-------------------------------------------------------------------------------------------

--Write Decoder
decoder1: Decoder_5x32 port map (write_sel,decoder_output);

--Registers
reg_0 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(0), write_data, reg_outputs(0));
reg_1 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(1), write_data, reg_outputs(1));
reg_2 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(2), write_data, reg_outputs(2));
reg_3 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(3), write_data, reg_outputs(3));
reg_4 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(4), write_data, reg_outputs(4));
reg_5 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(5), write_data, reg_outputs(5));
reg_6 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(6), write_data, reg_outputs(6));
reg_7 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(7), write_data, reg_outputs(7));
reg_8 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(8), write_data, reg_outputs(8));
reg_9 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(9), write_data, reg_outputs(9));
reg_10 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(10), write_data, reg_outputs(10));
reg_11 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(11), write_data, reg_outputs(11));
reg_12 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(12), write_data, reg_outputs(12));
reg_13 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(13), write_data, reg_outputs(13));
reg_14 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(14), write_data, reg_outputs(14));
reg_15 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(15), write_data, reg_outputs(15));
reg_16 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(16), write_data, reg_outputs(16));
reg_17 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(17), write_data, reg_outputs(17));
reg_18 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(18), write_data, reg_outputs(18));
reg_19 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(19), write_data, reg_outputs(19));
reg_20 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(20), write_data, reg_outputs(20));
reg_21 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(21), write_data, reg_outputs(21));
reg_22 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(22), write_data, reg_outputs(22));
reg_23 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(23), write_data, reg_outputs(23));
reg_24 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(24), write_data, reg_outputs(24));
reg_25 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(25), write_data, reg_outputs(25));
reg_26 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(26), write_data, reg_outputs(26));
reg_27 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(27), write_data, reg_outputs(27));
reg_28 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(28), write_data, reg_outputs(28));
reg_29 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(29), write_data, reg_outputs(29));
reg_30 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(30), write_data, reg_outputs(30));
reg_31 : Flopr GENERIC MAP (32) port map (clk, '0', write_ena AND decoder_output(31), write_data, reg_outputs(31));

-------------------------------------------------------------------------------------------
-----------------------------------Read Implementation-------------------------------------
-------------------------------------------------------------------------------------------

--MUX (1)
MUX_1: MUX_32x1 port map(reg_outputs (0) , reg_outputs (1) , reg_outputs (2) , reg_outputs (3) , reg_outputs (4) , reg_outputs
(5) , reg_outputs (6) , reg_outputs (7) , reg_outputs (8) , reg_outputs (9) , reg_outputs (10) , reg_outputs (11) , reg_outputs
(12) , reg_outputs (13) , reg_outputs (14) , reg_outputs (15) , reg_outputs (16) , reg_outputs (17) , reg_outputs (18) ,
reg_outputs (19) , reg_outputs (20) , reg_outputs (21) , reg_outputs (22) , reg_outputs (23) , reg_outputs (24) , reg_outputs
(25) , reg_outputs (26) , reg_outputs (27) , reg_outputs (28) , reg_outputs (29) , reg_outputs (30) , reg_outputs (31) ,
read_sel1 , data1);

--MUX (2)
MUX_2: MUX_32x1 port map(reg_outputs (0) , reg_outputs (1) , reg_outputs (2) , reg_outputs (3) , reg_outputs (4) , reg_outputs
(5) , reg_outputs (6) , reg_outputs (7) , reg_outputs (8) , reg_outputs (9) , reg_outputs (10) , reg_outputs (11) , reg_outputs
(12) , reg_outputs (13) , reg_outputs (14) , reg_outputs (15) , reg_outputs (16) , reg_outputs (17) , reg_outputs (18) ,
reg_outputs (19) , reg_outputs (20) , reg_outputs (21) , reg_outputs (22) , reg_outputs (23) , reg_outputs (24) , reg_outputs
(25) , reg_outputs (26) , reg_outputs (27) , reg_outputs (28) , reg_outputs (29) , reg_outputs (30) , reg_outputs (31) ,
read_sel2 , data2);


end Behavioral;
```

Register Code Comments:

- We declare a decoder for the WRITE operation to choose which register to make sure that one desired register is ON while writing
- We declare 32 Floprs to be used in memory storing
- We declare 2 MUXs to read 2 memory locations “Floprs” @ one time from the Register

RTL Diagram:

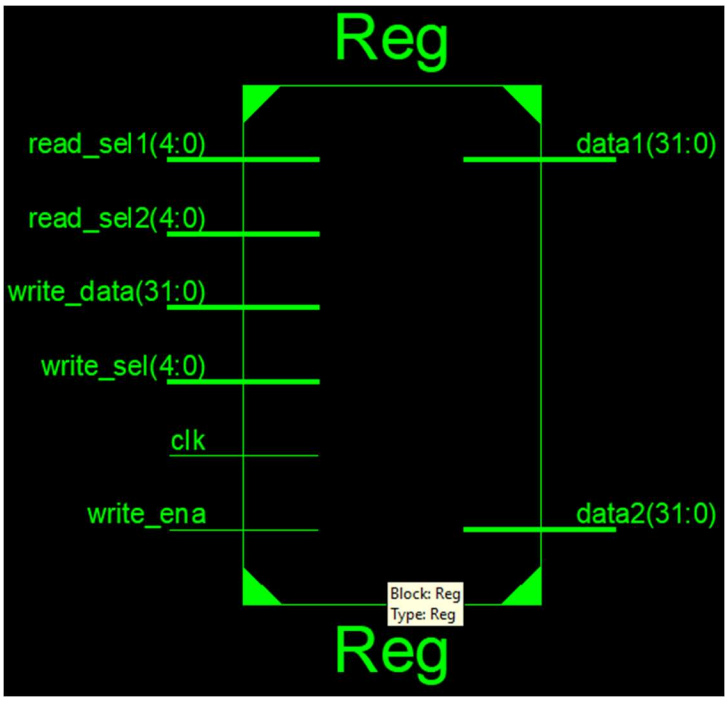

Figure 6: Register File RTL Diagram

Test:

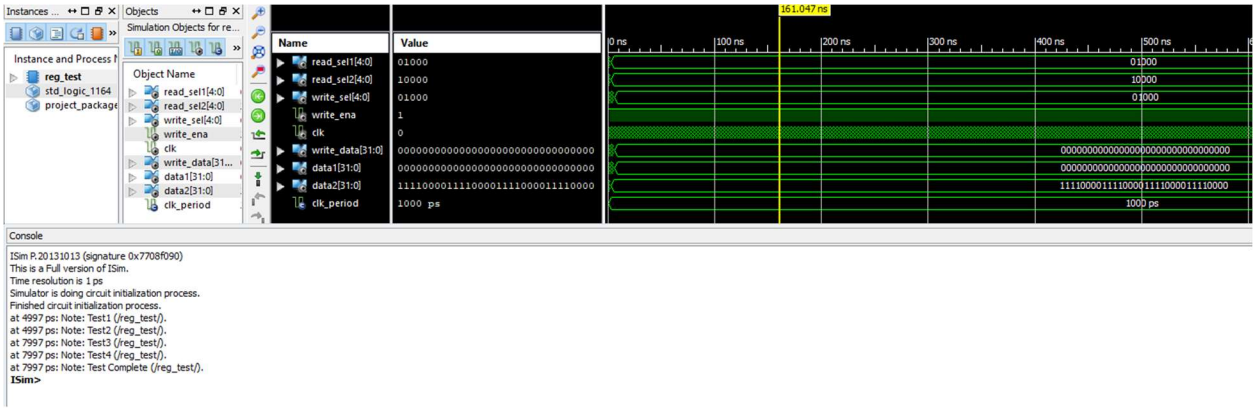

Figure 7: Register Testing

Part (3):

32-Bit ALU main function is to take the data stored in the registers and make operations on it and then store it back on the registers.

Diagram:

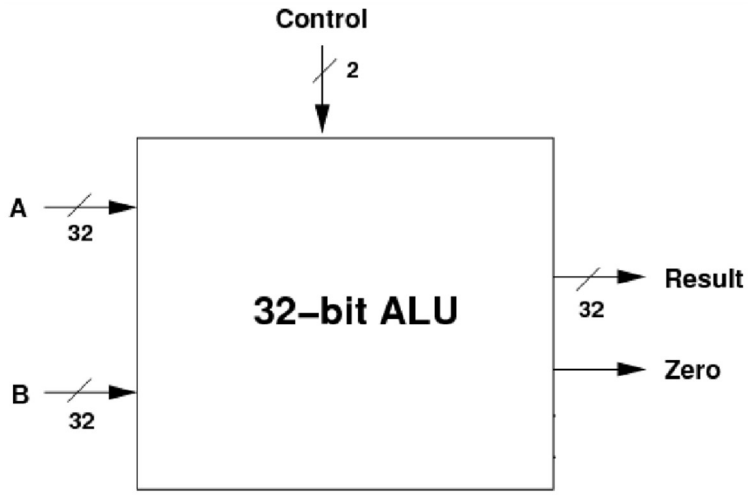

Figure 8: ALU Diagram

Code:

```
library IEEE;

--Library Declaration (Unsigned) -> "+-...."
use IEEE.STD_LOGIC_1164.ALL;

--UNSIGNED library is used to introduce some arithmetic operation such as Addition, Subtraction, etc...
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

--Input/Output Declarations
entity 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);
           dataout : out  STD_LOGIC_VECTOR (31 downto 0);
           zflag : out STD_LOGIC);
end ALU;

architecture Behavioral of ALU is

SIGNAL tmp  : STD_LOGIC_VECTOR (31 downto 0);
SIGNAL BB : STD_LOGIC_VECTOR (31 downto 0);
SIGNAL result : STD_LOGIC_VECTOR (31 downto 0);

begin

BB <= (not data2) when (aluop(3 downto 2) = "01") else data2;
tmp <= data1 + BB + aluop(2);
result <=   tmp when (aluop = "0010" or aluop = "0110") else
                (data1 AND data2) when aluop <= "0000" else
                (data1 OR data2) when aluop <= "0001" else
                (data1 NOR data2) when aluop <= "1100" else
                (Others => 'Z');

zflag <= '1' when result = x"00000000" else '0';

dataout <= result;

end Behavioral;
```

ALU Code Comments:

- We declare a signal BB to either put data1 as it is to be used in the addition operation or its 2's complement to be used in subtraction
- We use **<= …. when condition** to apply conditioning according to the ALU operation
- **zflag** is a signal that indicates whether the result of any of the operations performed by the ALU is equal **ZERO**

RTL Diagram:

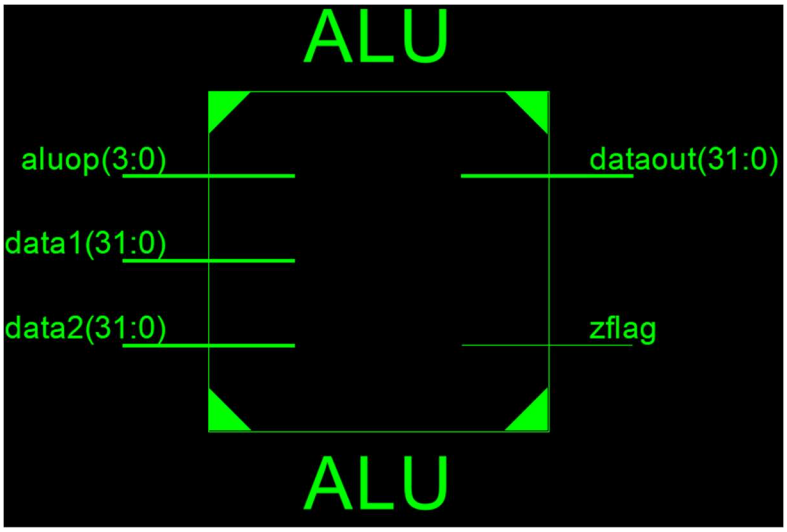

Figure 9: ALU RTL Diagram

Test:

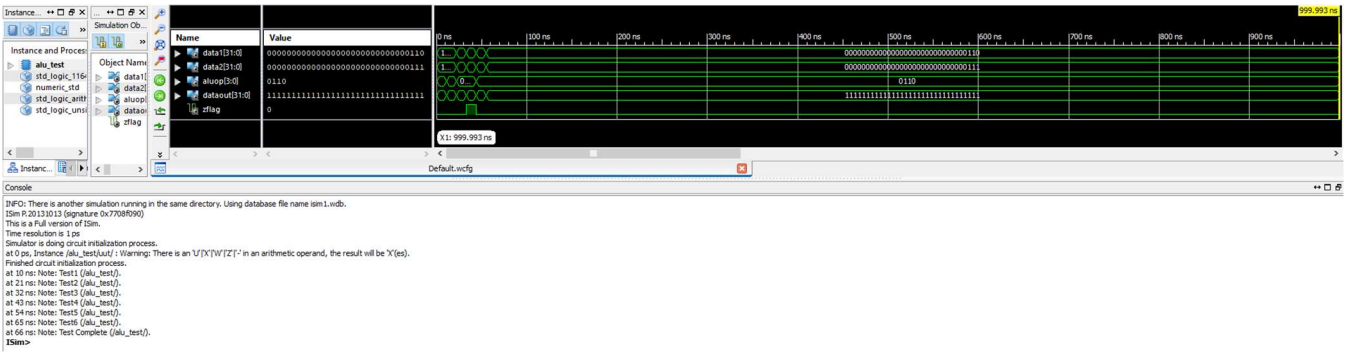

Figure 10: ALU Test

Part (4) “Final Part”:

At the final part we need to connect both the Register & ALU to have our simple R-Type instructions processor. We do that by putting both files in a Package to be used in the Datapath file.

Diagram:

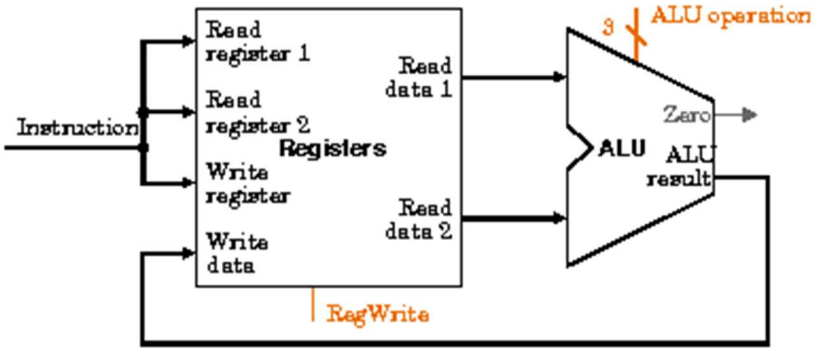

Figure 11: R-Type Simple Processor Diagram

Code:

Package

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

package Datapath_Package is

component Reg
    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;

component ALU
    Port ( data1 : in  STD_LOGIC_VECTOR (31 downto 0);
           data2 : in  STD_LOGIC_VECTOR (31 downto 0);
           aluop : in  STD_LOGIC_VECTOR (3 downto 0);
           dataout : out  STD_LOGIC_VECTOR (31 downto 0);
              zflag : out STD_LOGIC);
end component;

end Datapath_Package;
```

Datapath

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

entity Datapath is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           instr : in  STD_LOGIC_VECTOR (31 downto 0);
           aluoperation : in  STD_LOGIC_VECTOR (3 downto 0);
           zero : out  STD_LOGIC;
           regwrite : in  STD_LOGIC;
           aluout : buffer  STD_LOGIC_VECTOR (31 downto 0));
end Datapath;

architecture Behavioral of Datapath is

SIGNAL regout1 : STD_LOGIC_VECTOR (31 downto 0);
SIGNAL regout2 : STD_LOGIC_VECTOR (31 downto 0);

begin

--Register
Regist : Reg port map (instr(25 downto 21) , instr(20 downto 16) , instr(15 downto 11) , '1', clk , aluout , regout1 , regout2);


--ALU
ALU1 : ALU port map (regout1 , regout2 , aluoperation , aluout , zero);


end Behavioral;
```

Datapath Code Comments:

- **instr** is divided into 3 parts to be used in the **read1**, **read2**, **writesel** where the processor takes the 2 registers to read from and the one to write at.
- The data read from the register is then entered to the ALU along with the **aluoperation** to perform any of the implemented operations
- The data output from the ALU is then written in a memory location inside the register in the desired memory location.
- The **zero** output is the same as the **zflag** output from ALU.

RTL Diagram:

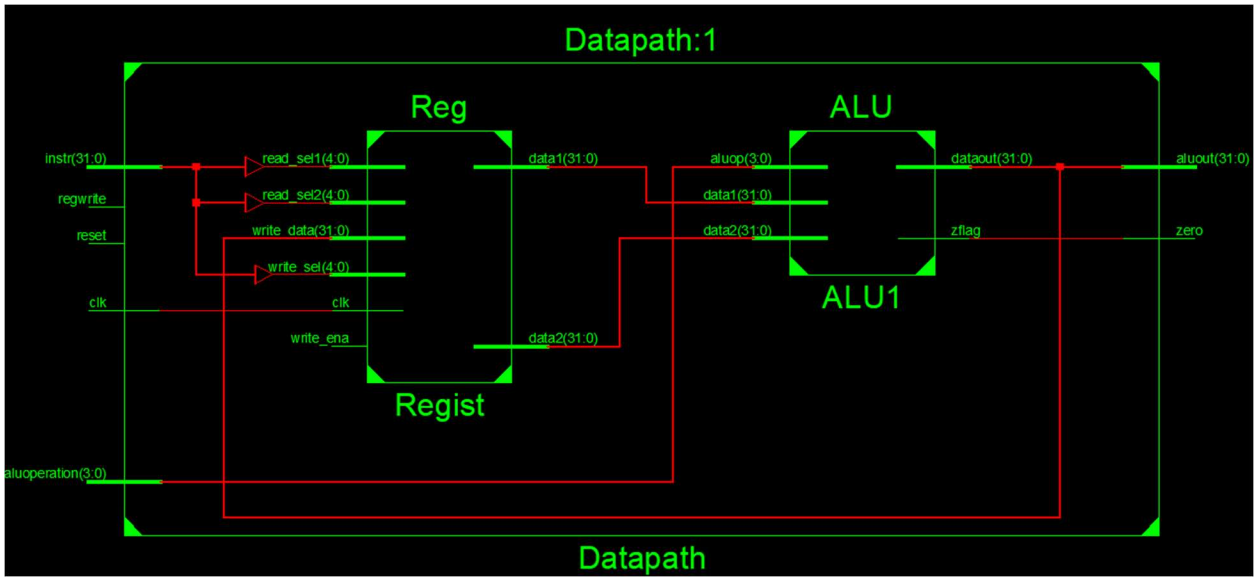

Figure 12: Datapath RTL Diagram