



# Project Report

CESS3005: Computer Architecture  
Major Task: MIPS Design using VHDL



## Prepared by :

Abdallah Belal Momen	22P0036
Ahmed Khaled Hamdy	22P0024
Malak Ossama Zaiter	22P0052
Mohamed Khaled Mohamed	22P0041
Yousef Sameh Shoman	22P0010

---

## Prepared to :

Dr. Karim Ahmed Emara  
Eng. Aya

## Table of Contents

Phase 1 .....	1
1. Executive Summary .....	1
1.1. Objectives .....	1
1.2. Implementation Highlights .....	2
2. Implementation Details: .....	3
2.1. Create a Decoder Module .....	3
2.2. Create a Register Module .....	4
2.3. Create a MUX32x1 Module .....	5
2.4. Gather In One Package: .....	6
2.5. Write Main Module RegisterFile: .....	7
2.6. TestBench for RegisterFile Module: .....	10
2.7. Create ALU Module .....	12
2.8. TestBench for ALU Module .....	14
2.9. Create Datapath Module .....	16
3. Contribution .....	17
4. Conclusion .....	17
Phase 2 .....	18
1. Executive Summary .....	18
1.1. Objectives .....	18
2. Implementation Details: .....	19
2.1. Create a Control Module .....	19
2.2. Create a Data Memory Module .....	22
2.3. Create an Instruction Memory Module .....	23
2.4. Create a Sign Extender Module .....	25
2.5. Create a Shift Left x2 Module .....	25
2.6. Create a MUX 2x1 Module .....	26
2.7. Create a Datapath Module .....	27
2.8. Create a MIPS Module .....	29
2.9. Create the Main Module .....	31
3. Contribution .....	33
4. Conclusion .....	33
5. Appendix .....	34

## Table of Figures

Figure 1 RTL Schematic for decoder .....	3
Figure 2 RTL Schematic for decoder .....	3
Figure 3 VHDL code of decoder .....	3
Figure 4 VHDL code of decoder .....	3
Figure 5 RTL Schematic for flopR .....	4
Figure 6 RTL Schematic for flopR .....	4
Figure 7 VHDL code of flopR .....	4
Figure 8 VHDL code of flopR .....	4
Figure 9 RTL Schematic for MUX32x1 .....	5
Figure 10 RTL Schematic for MUX32x1 .....	5
Figure 11 VHDL code of MUX32x1 .....	5
Figure 12 VHDL code of MUX32x1 .....	5
Figure 13 VHDL code of the package .....	6
Figure 14 VHDL code of the package .....	6
Figure 15 RegisterFile as a whole .....	7
Figure 16 RegisterFile as a whole .....	7
Figure 17 Internal Components of the RegisterFile .....	7
Figure 18 Internal Components of the RegisterFile .....	7
Figure 19 RTL Schematic for RegisterFile .....	8
Figure 20 RTL Schematic for RegisterFile .....	8
Figure 21 VHDL code of RegisterFile .....	9
Figure 22 RegisterFile Illustration .....	9
Figure 23 TestBench RegisterFile .....	10
Figure 24 RTL Schematic for ALU .....	12
Figure 25 RTL Schematic for ALU bonus .....	12
Figure 26 VHDL code of ALU bonus .....	13
Figure 27 VHDL code of ALU .....	13
Figure 28 TestBench ALU .....	14
Figure 29 Test Cases Outcome .....	15
Figure 30 VHDL code of Datapath .....	16
Figure 31 RTL Schematic of ALU Control .....	19
Figure 32 Alu Decoder Function Table .....	19
Figure 33 RTL Schematic of Main Control .....	20
Figure 34 Main Decoder Function Table .....	20
Figure 35 RTL Schematic of Controller .....	21
Figure 36 ALU Control Code .....	21
Figure 37 Main Control Code .....	21
Figure 38 Controller Code .....	21
Figure 39 RTL Schematic of DMEM .....	22
Figure 40 DMEM Code .....	22
Figure 41 IMEM Code .....	24
Figure 42 RTL Schematic of SignExtend .....	25
Figure 43 RTL Schematic of ShiftLeft2 .....	25
Figure 44 RTL Schematic of MUX2 .....	26
Figure 45 RTL Schematic of Datapath .....	27
Figure 46 Datapath Code .....	28
Figure 47 RTL Schematic of MIPS .....	29
Figure 48 MIPS Code .....	30
Figure 49 Main Code .....	32

# Phase 1

## 1.Executive Summary

This report details the design, implementation, and simulation of a simplified MIPS processor using VHDL. The project encompasses the creation of a functional MIPS CPU capable of executing basic R-type instructions, with key components including a Register File and a functional 32-bit Arithmetic Logic Unit (ALU). The implementation demonstrates the application of VHDL in modeling and simulating complex digital circuits and systems, with an emphasis on educational use and a foundational understanding of microprocessor architecture.

### 1.1. Objectives

#### 1. Implement a MIPS Register File

- **Module Name:** RegisterFile
- **Functionality:** The register file should allow simultaneous read operations from two registers and a write operation into another register.
- **Inputs and Outputs:**

read_sel1	: Input, 5-bit (std_logic_vector(4 downto 0))
read_sel2	: Input, 5-bit (std_logic_vector(4 downto 0))
write_sel	: Input, 5-bit (std_logic_vector(4 downto 0))
write_ena	: Input, 1-bit (std_logic)
clk	: Input, Clock signal (std_logic)
write_data	: Input, 32-bit (std_logic_vector(31 downto 0))
data1	: Output, 32-bit (std_logic_vector(31 downto 0))
data2	: Output, 32-bit (std_logic_vector(31 downto 0))

#### 2. Modify the 32-bit Full ALU

- **ALU Operations:**
- **Inputs and Outputs:**

0000	: AND	data1	: Input, 32-bit(std_logic_vector(31 downto 0))
0001	: OR	data2	: Input, 32-bit(std_logic_vector(31 downto 0))
0010	: ADD	aluop	: Input, 4-bit(std_logic_vector(3 downto 0))
0110	: SUB	dataout	: Output, 32-bit(std_logic_vector(31 downto 0))
1100	: NOR	zflag	: Output, 1-bit (std_logic)
0111	: SLT		

## Design a Simple MIPS CPU

- **Functionality:** The CPU should handle basic R-type instructions such as AND, OR, ADD, SUB, SLT, and NOR.
- **Components:** Integrate the register file and ALU with additional necessary modules to create the MIPS CPU.
- **Datapath Specifications:**

```
clk, reset    : Inputs (std_logic)

instr         : Input, 32-bit instruction (std_logic_vector(31 downto 0))

aluoperation  : Input, 3-bit operation code (std_logic_vector(2 downto 0))

zero          : Output, zero flag (std_logic)

regwrite      : Input, control signal for a register write operation (std_logic)

aluout        : Buffer, 32-bit ALU output (std_logic_vector(31 downto 0))
```

### 1.2. Implementation Highlights

- **Register File:** Implemented with inputs for two read selectors and one write selector, along with a write enable and clock/reset signal, ensuring controlled data handling and concurrency.
- **ALU:** Modified to perform six essential operations (AND, OR, ADD, SUB, NOR, and SLT) controlled by a 4-bit operation code, producing both result output and a zero flag for conditional operations.
- **CPU Integration:** Combined the Register File and ALU into a cohesive MIPS CPU architecture with the ability to execute, decode, and fetch instructions within a single cycle, adhering to the MIPS instruction set architecture.

## 2.1. Create a Decoder Module

## 2.1. Create a Decoder Module

The decoder will receive the address of the register (5 bits) then it will decode it to act as an enable for the internal 32 registers.

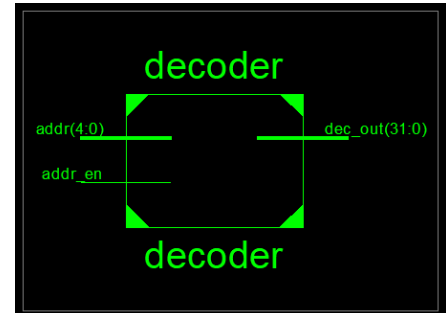


Figure 1 RTL Schematic for decoder


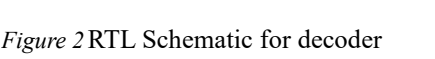
- **addr(4:0):** This is a 5-bit input line to the decoder. Each bit (from addr(0) to addr(4)) represents one bit of the binary address input.  *Figure 1 RTL Schematic for decoder*
- **addr\_en:** This stands for "address enable" and is likely a control signal. When this signal is active (typically high), it enables the decoder to perform its function. When low, it may disable the decoder's output, possibly forcing all outputs to a high-impedance state ('Z').  *Figure 2 RTL Schematic for decoder*
- **dec\_out(31:0):** These are the 32 output lines of the decoder. Only one of these lines will be active (typically high) at a time, corresponding to the binary value of the 5-bit address input. For example, if the addr is "00010", then dec\_out(2) would be high which will then activate register 1 only.

Figure 2 RTL Schematic for decoder

[illegible]

Figure 3 VHDL code of decoder

## 2.2. Create a Register Module

After receiving an output from the decoder, it will be used to enable the appropriate register out of 32 registers, to have 32 registers inside the register file, we will create a flip-flop/register component as follows:

- **datain(31:0)** - This is the data input of the register which is a 32-bit wide bus. This input is where the data will be stored in the register on the rising edge of the clock if the enable (en) signal is active.
- **clk** - The clock input to the register. This signal is used to synchronize the data capturing of the register with the rest of the system. The register typically captures the input data on either the rising or falling edge of this clock signal.
- **write\_en** - Enable input. This is a control signal that determines when the register is allowed to capture and store the input data. If write\_en is high (logic level '1'), the register captures the data present on datain lines at the next active clock edge.
- **rst** - Reset input. This signal is used to reset the register, usually setting all its bits to zero. It is generally active-high, meaning when rst is high, the register's output will be cleared or reset to a predefined value (usually all '0's).
- **dataout(31:0)** - This is the data output of the register which is also a 32-bit wide bus. The data stored in the register is presented on this output.

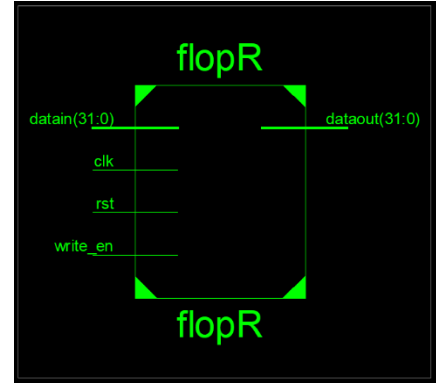


Figure 5 RTL Schematic for flopR

Figure 6 RTL Schematic for flopR

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

entity flopR is
    Port ( datain      : in  STD_LOGIC_VECTOR (31 downto 0);
          dataout     : out STD_LOGIC_VECTOR (31 downto 0);
          write_en    : in  STD_LOGIC; --from decoder output to write register address
          rst         : in  STD_LOGIC;
          clk         : in  STD_LOGIC);
end flopR;

architecture Behavioral of flopR is
    signal internal_dataout: STD_LOGIC_VECTOR (31 downto 0);
    begin
        process (clk,rst,datain, internal_dataout)
        begin
            if (rst='1') then
                internal_dataout <= (others => '0');
            elsif (rising_edge(clk)) then -- Perform actions on the rising edge
                if (write_en='1') then
                    internal_dataout <= datain;
                end if;
            end if;
            dataout <= internal_dataout;
        end process;
    end Behavioral;
end
```

Figure 7 VHDL code of flopR

## 2.3. Create a MUX32x1 Module

This multiplexer will be used to take the output of the previous registers (which is the input data1 or data2) by having the address of the registers (from 0 to 31 in binary) as 5 selectors to get our data as an output, which will then go into the ALU.

- **32 Inputs (in0 to in31):** These are the data inputs of the multiplexer. Each input is capable of carrying a 32-bit value.
- **5-bit Address (addr):** This is the control input that determines which of the 32 data inputs is connected to the output. The value of addr ranges from "00000" to "11111" in binary, allowing for 32 different selections corresponding to the 32 inputs.
- **32-bit Output (mux\_out):** This is where the selected input appears. The selection depends on the value present at the addr lines.

In operation, the multiplexer examines the value of addr, and based on this value, it connects the corresponding (inX) input to the mux\_out.

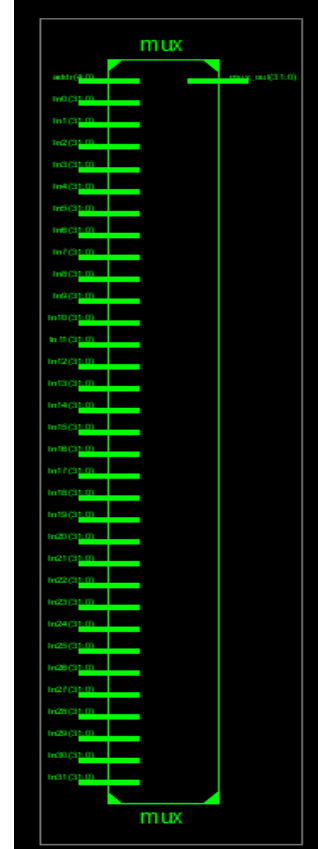


Figure 9 RTL Schematic for MUX32x1

```
entity mux is
  Port (
    In0 : in STD_LOGIC_VECTOR (31 downto 0); --all inputs are from outputs of flip flop
    In1 : in STD_LOGIC_VECTOR (31 downto 0);
    In2 : in STD_LOGIC_VECTOR (31 downto 0);
    In3 : in STD_LOGIC_VECTOR (31 downto 0);
    In4 : in STD_LOGIC_VECTOR (31 downto 0);
    In5 : in STD_LOGIC_VECTOR (31 downto 0);
    In6 : in STD_LOGIC_VECTOR (31 downto 0);
    In7 : in STD_LOGIC_VECTOR (31 downto 0);
    In8 : in STD_LOGIC_VECTOR (31 downto 0);
    In9 : in STD_LOGIC_VECTOR (31 downto 0);
    In10 : in STD_LOGIC_VECTOR (31 downto 0);
    In11 : in STD_LOGIC_VECTOR (31 downto 0);
    In12 : in STD_LOGIC_VECTOR (31 downto 0);
    In13 : in STD_LOGIC_VECTOR (31 downto 0);
    In14 : in STD_LOGIC_VECTOR (31 downto 0);
    In15 : in STD_LOGIC_VECTOR (31 downto 0);
    In16 : in STD_LOGIC_VECTOR (31 downto 0);
    In17 : in STD_LOGIC_VECTOR (31 downto 0);
    In18 : in STD_LOGIC_VECTOR (31 downto 0);
    In19 : in STD_LOGIC_VECTOR (31 downto 0);
    In20 : in STD_LOGIC_VECTOR (31 downto 0);
    In21 : in STD_LOGIC_VECTOR (31 downto 0);
    In22 : in STD_LOGIC_VECTOR (31 downto 0);
    In23 : in STD_LOGIC_VECTOR (31 downto 0);
    In24 : in STD_LOGIC_VECTOR (31 downto 0);
    In25 : in STD_LOGIC_VECTOR (31 downto 0);
    In26 : in STD_LOGIC_VECTOR (31 downto 0);
    In27 : in STD_LOGIC_VECTOR (31 downto 0);
    In28 : in STD_LOGIC_VECTOR (31 downto 0);
    In29 : in STD_LOGIC_VECTOR (31 downto 0);
    In30 : in STD_LOGIC_VECTOR (31 downto 0);
    In31 : in STD_LOGIC_VECTOR (31 downto 0);
    mux_out : out STD_LOGIC_VECTOR (31 downto 0);
    addr : in STD_LOGIC_VECTOR (4 downto 0); --address of register
  );
end mux;
```

Figure 11 VHDL code of MUX32x1

```
architecture Behavioral of mux is
begin
  mux_out <= In0 when addr = "00000" else
    In1 when addr = "00001" else
    In2 when addr = "00010" else
    In3 when addr = "00011" else
    In4 when addr = "00100" else
    In5 when addr = "00101" else
    In6 when addr = "00110" else
    In7 when addr = "00111" else
    In8 when addr = "01000" else
    In9 when addr = "01001" else
    In10 when addr = "01010" else
    In11 when addr = "01011" else
    In12 when addr = "01100" else
    In13 when addr = "01101" else
    In14 when addr = "01110" else
    In15 when addr = "01111" else
    In16 when addr = "10000" else
    In17 when addr = "10001" else
    In18 when addr = "10010" else
    In19 when addr = "10011" else
    In20 when addr = "10100" else
    In21 when addr = "10101" else
    In22 when addr = "10110" else
    In23 when addr = "10111" else
    In24 when addr = "11000" else
    In25 when addr = "11001" else
    In26 when addr = "11010" else
    In27 when addr = "11011" else
    In28 when addr = "11100" else
    In29 when addr = "11101" else
    In30 when addr = "11110" else
    In31 when addr = "11111";
end Behavioral;
```

Figure 12 VHDL code of MUX32x1



## 2.4. Gather In One Package:

A package is a collection of named items like types, subtypes, functions, procedures, constants, attributes, files, components...etc. They are used to centralize the location where any of these things are declared. Generally in a design, if you're gonna use another design as a building block you have to declare it as a component.

The component statement will be a restatement for the entity of each component respectively. The library in which this package will reside once compiled is called "work".

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

package reg_components is

-----DECODER-----
component decoder is
    Port ( addr : in  STD_LOGIC_VECTOR (4 downto 0);
          addr_en : in  STD_LOGIC;
          dec_out : out STD_LOGIC_VECTOR (31 downto 0)); --decode address to enable specific register
end component;

-----FLOPR-----
component flopR is
    Port ( datain : in  STD_LOGIC_VECTOR (31 downto 0);
          dataout : out STD_LOGIC_VECTOR (31 downto 0);
          write_en : in  STD_LOGIC; --from decoder output to write register address
          rst : in  STD_LOGIC;
          clk : in  STD_LOGIC);
end component;

-----MUX32x1-----
component mux is
    Port ( In0 : in  STD_LOGIC_VECTOR (31 downto 0); --all inputs are from outputs of flip flop
          In1 : in  STD_LOGIC_VECTOR (31 downto 0);
          In2 : in  STD_LOGIC_VECTOR (31 downto 0);
          In3 : in  STD_LOGIC_VECTOR (31 downto 0);
          In4 : in  STD_LOGIC_VECTOR (31 downto 0);
          In5 : in  STD_LOGIC_VECTOR (31 downto 0);
          In6 : in  STD_LOGIC_VECTOR (31 downto 0);
          In7 : in  STD_LOGIC_VECTOR (31 downto 0);
          In8 : in  STD_LOGIC_VECTOR (31 downto 0);
          In9 : in  STD_LOGIC_VECTOR (31 downto 0);
          In10 : in  STD_LOGIC_VECTOR (31 downto 0);
          In11 : in  STD_LOGIC_VECTOR (31 downto 0);
          In12 : in  STD_LOGIC_VECTOR (31 downto 0);
          In13 : in  STD_LOGIC_VECTOR (31 downto 0);
          In14 : in  STD_LOGIC_VECTOR (31 downto 0);
          In15 : in  STD_LOGIC_VECTOR (31 downto 0);
          In16 : in  STD_LOGIC_VECTOR (31 downto 0);
          In17 : in  STD_LOGIC_VECTOR (31 downto 0);
          In18 : in  STD_LOGIC_VECTOR (31 downto 0);
          In19 : in  STD_LOGIC_VECTOR (31 downto 0);
          In20 : in  STD_LOGIC_VECTOR (31 downto 0);
          In21 : in  STD_LOGIC_VECTOR (31 downto 0);
          In22 : in  STD_LOGIC_VECTOR (31 downto 0);
          In23 : in  STD_LOGIC_VECTOR (31 downto 0);
          In24 : in  STD_LOGIC_VECTOR (31 downto 0);
          In25 : in  STD_LOGIC_VECTOR (31 downto 0);
          In26 : in  STD_LOGIC_VECTOR (31 downto 0);
          In27 : in  STD_LOGIC_VECTOR (31 downto 0);
          In28 : in  STD_LOGIC_VECTOR (31 downto 0);
          In29 : in  STD_LOGIC_VECTOR (31 downto 0);
          In30 : in  STD_LOGIC_VECTOR (31 downto 0);
          In31 : in  STD_LOGIC_VECTOR (31 downto 0);
          mux_out : out STD_LOGIC_VECTOR (31 downto 0);
          addr : in  STD_LOGIC_VECTOR (4 downto 0)); --address of register
end component;
end reg_components;
```

Figure 13 VHDL code of the package

## 2.5. Write Main Module RegisterFile:

In this module, we will connect every component by associating their port maps logically together. This was achieved by understanding Figure 8 & 9 as follows:

- The Addr signal from the schematic would be used to select which register's data is to be read, corresponding to the readreg1 and readreg2 inputs in the Register File entity.
- The Data IN in the schematic would correspond to the write\_data in the Register File entity.
- The Data OUT lines in the schematic would be the outputs data1 and data2 in the VHDL entity, providing the contents of the selected registers.
- The En and Clr signals in the schematic are control signals that would be managed inside the VHDL entity, possibly by the control unit, to enable the writing to a register and to clear a register respectively.
- The clock signal Clk in the schematic would synchronize the data writing process within the registers in the VHDL code, corresponding to the clk signal in the Register File entity.
- Rst in the schematic would correspond to the clr signal in the VHDL entity, typically used to reset or initialize the registers.

The control logic to manage these connections would be within the VHDL entity and architecture, enabling or disabling each register, handling data input and output, and selecting the appropriate register based on the address input. This logic would ensure the correct data flow through the Register File during each phase of the MIPS instruction cycle (fetch, decode, execute, etc.).

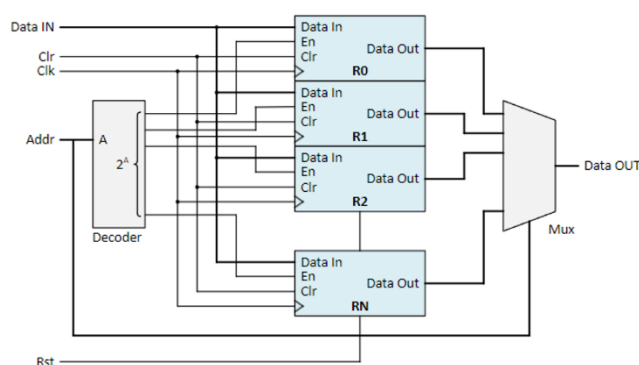


Figure 17 Internal Components of the RegisterFile

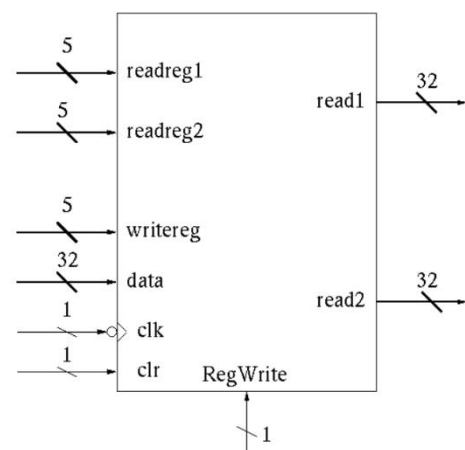


Figure 15 RegisterFile as a whole

- **RegisterFile:** This is a module or a component in the CPU that consists of a set of registers where the CPU stores data temporarily.
- **read\_sel1(4:0) and read\_sel2(4:0):** These are 5-bit input signals used to select which registers to read from. The numbers indicate the bit range, with 4 being the most significant bit (MSB) and 0 the least significant bit (LSB). A 5-bit selector can address 32 different registers ( $2^5$ ).
- **data1(31:0) and data2(31:0):** These are 32-bit output signals corresponding to the data read from the register selected by read\_sel1 and read\_sel2, respectively. The 32-bit width suggests this is a 32-bit architecture, common in MIPS systems.
- **write\_data(31:0):** This is a 32-bit input signal that carries the data to be written into the register selected by write\_sel.
- **write\_sel(4:0):** This is another 5-bit input signal used to select the register into which the write\_data will be written.
- **clk:** This is the clock signal input, which synchronizes the read and write operations of the registers to the rising or falling edge of the clock pulses.
- **Write\_ena:** This is a control signal input. When it is active (usually high), it enables writing the write\_data to the register addressed by write\_sel. When it is not active (usually low), no write operation should occur.
- **rst:** This is the reset signal input. When active, it initializes the registers, typically setting them to a default value like zero.

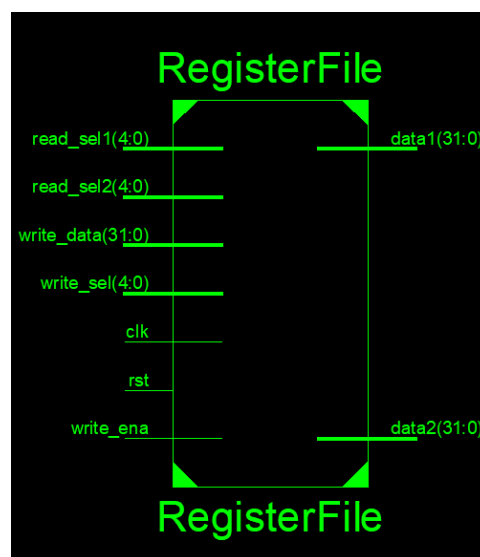


Figure 19 RTL Schematic for RegisterFile

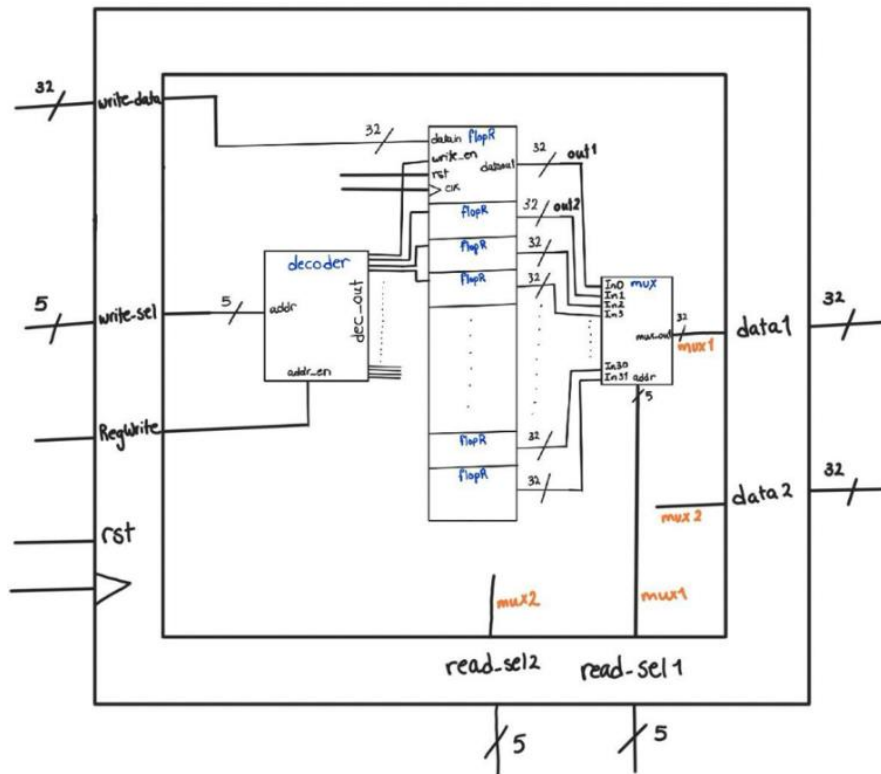


Figure 22 RegisterFile Illustration

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.reg_components.ALL;

entity RegisterFile is
    Port ( read_sel1 : in STD_LOGIC_VECTOR (4 downto 0); --reads address of reg1 to decoder
          read_sel2 : in STD_LOGIC_VECTOR (4 downto 0); --reads address of reg2 to decoder
          write_sel  : in STD_LOGIC_VECTOR (4 downto 0); --write address to mux
          write_data : in STD_LOGIC_VECTOR (31 downto 0); --datain
          clk        : in STD_LOGIC;
          rst        : in STD_LOGIC;
          write_ena  : in STD_LOGIC; --write data into register specified by write_sel (enable)
          data1      : out STD_LOGIC_VECTOR (31 downto 0); --dataout1 from mux
          data2      : out STD_LOGIC_VECTOR (31 downto 0); --dataout2 from mux
    end RegisterFile;

architecture Behavioral of RegisterFile is
    signal out1, out2, out3, out4, out5, out6, out7, out8, out9, out10,
           out11, out12, out13, out14, out15, out16, out17, out18, out19, out20,
           out21, out22, out23, out24, out25, out26, out27, out28, out29, out30, out31, out32 : STD_LOGIC_VECTOR (31 downto 0);
    signal dec_out : STD_LOGIC_VECTOR (31 downto 0);

    begin
        -----DECODER-----
        decoder1: decoder PORT MAP(write_sel, write_ena, dec_out);

        -----FLOPR-----
        flopR1: flopR PORT MAP(write_data, out1, dec_out(0), '0', clk);
        flopR2: flopR PORT MAP(write_data, out2, dec_out(1), '0', clk);
        flopR3: flopR PORT MAP(write_data, out3, dec_out(2), '0', clk);
        flopR4: flopR PORT MAP(write_data, out4, dec_out(3), '0', clk);
        flopR5: flopR PORT MAP(write_data, out5, dec_out(4), '0', clk);
        flopR6: flopR PORT MAP(write_data, out6, dec_out(5), '0', clk);
        flopR7: flopR PORT MAP(write_data, out7, dec_out(6), '0', clk);
        flopR8: flopR PORT MAP(write_data, out8, dec_out(7), '0', clk);
        flopR9: flopR PORT MAP(write_data, out9, dec_out(8), '0', clk);
        flopR10: flopR PORT MAP(write_data, out10, dec_out(9), '0', clk);
        flopR11: flopR PORT MAP(write_data, out11, dec_out(10), '0', clk);
        flopR12: flopR PORT MAP(write_data, out12, dec_out(11), '0', clk);
        flopR13: flopR PORT MAP(write_data, out13, dec_out(12), '0', clk);
        flopR14: flopR PORT MAP(write_data, out14, dec_out(13), '0', clk);
        flopR15: flopR PORT MAP(write_data, out15, dec_out(14), '0', clk);
        flopR16: flopR PORT MAP(write_data, out16, dec_out(15), '0', clk);
        flopR17: flopR PORT MAP(write_data, out17, dec_out(16), '0', clk);
        flopR18: flopR PORT MAP(write_data, out18, dec_out(17), '0', clk);
        flopR19: flopR PORT MAP(write_data, out19, dec_out(18), '0', clk);
        flopR20: flopR PORT MAP(write_data, out20, dec_out(19), '0', clk);
        flopR21: flopR PORT MAP(write_data, out21, dec_out(20), '0', clk);
        flopR22: flopR PORT MAP(write_data, out22, dec_out(21), '0', clk);
        flopR23: flopR PORT MAP(write_data, out23, dec_out(22), '0', clk);
        flopR24: flopR PORT MAP(write_data, out24, dec_out(23), '0', clk);
        flopR25: flopR PORT MAP(write_data, out25, dec_out(24), '0', clk);
        flopR26: flopR PORT MAP(write_data, out26, dec_out(25), '0', clk);
        flopR27: flopR PORT MAP(write_data, out27, dec_out(26), '0', clk);
        flopR28: flopR PORT MAP(write_data, out28, dec_out(27), '0', clk);
        flopR29: flopR PORT MAP(write_data, out29, dec_out(28), '0', clk);
        flopR30: flopR PORT MAP(write_data, out30, dec_out(29), '0', clk);
        flopR31: flopR PORT MAP(write_data, out31, dec_out(30), '0', clk);
        flopR32: flopR PORT MAP(write_data, out32, dec_out(31), '0', clk);

        -----OPERATE MUX1-----
        mux1: mux PORT MAP ( out1, out2, out3, out4, out5, out6, out7, out8, out9, out10,
                           out11, out12, out13, out14, out15, out16, out17, out18, out19, out20,
                           out21, out22, out23, out24, out25, out26, out27, out28, out29, out30, out31, out32, data1, read_sel1);

        -----OPERATE MUX2-----
        mux2: mux PORT MAP ( out1, out2, out3, out4, out5, out6, out7, out8, out9, out10,
                           out11, out12, out13, out14, out15, out16, out17, out18, out19, out20,
                           out21, out22, out23, out24, out25, out26, out27, out28, out29, out30, out31, out32, data2, read_sel2);

    end Behavioral;

```

Figure 21 VHDL code of RegisterFile

## 2.6. TestBench for RegisterFile Module:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.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;
        rst        : IN  std_logic; -- Reset signal
        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;

    --Inputs
    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 rst        : std_logic := '0';
    signal write_data : std_logic_vector(31 downto 0) := (others => '0');

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

    -- Clock period definitions
    constant clk_period : time := 10ps;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: RegisterFile PORT MAP (
        read_sel1 => read_sel1,
        read_sel2 => read_sel2,
        write_sel  => write_sel,
        write_ena  => write_ena,
        clk        => clk,
        rst        => rst,
        write_data => write_data,
        data1      => data1,
        data2      => data2
    );

    -- Clock process definitions
    clk_process: process
    begin
        while now < 1000 ns loop
            clk <= '0';
            wait for clk_period / 2;
            clk <= '1';
            wait for clk_period / 2;
        end loop;
        wait;
    end process;
```

Figure 23 TestBench RegisterFile



## 2.7. Create ALU Module

- **aluop(3:0):** This represents a 4-bit input to the ALU that specifies the operation to be performed. we implemented 6 operations: AND, ADD, SUB, SLT, NOR & OR.
- **data1(31:0) and data2(31:0):** These are two 32-bit inputs to the ALU, representing the operands on which the ALU will perform the specified operation. The numbers (31:0) indicate that these are 32-bit wide inputs, supporting a standard word size for modern processors.

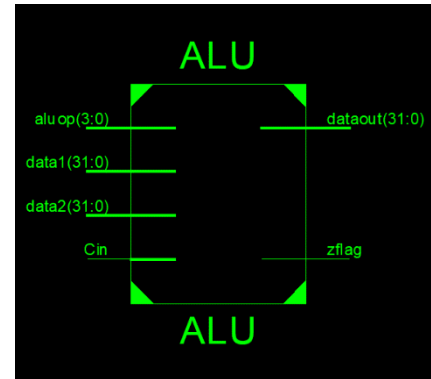


Figure 24 RTL Schematic for ALU

- **Cin:** This is typically the carry-in input for arithmetic operations. It's used in operations like addition, where carry from a previous operation may need to be considered.  
**Note:** Not all ALUs will have a Cin port. It depends on the specific design and requirements of the system that the ALU is a part of.
- **dataout(31:0):** This is the 32-bit output of the ALU, where the result of the operation performed on data1 and data2 is placed.
- **zflag:** This is a flag that is set when the result of the ALU operation is zero. In processor design, zero flags are used to influence conditional branches and other operations depending on whether the result of the previous operation was zero.

### Bonus Ports:

- **cflag:** Short for "Carry Flag". This output indicates whether the last operation resulted in a carry-out, which may be used in subsequent arithmetic calculations, particularly for operations on values that span multiple words.
- **oflag:** Stands for "Overflow Flag". This signal indicates if an arithmetic overflow occurred during the last operation, which is useful for error checking and handling exceptional conditions in calculations.

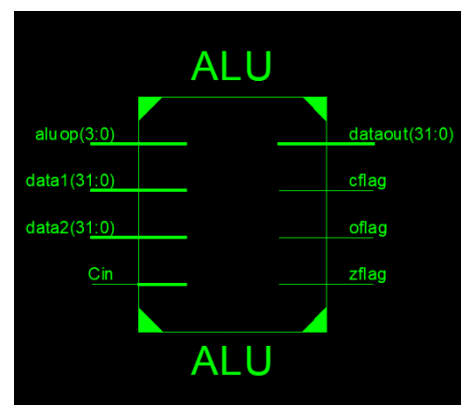


Figure 25 RTL Schematic for ALU bonus

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

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;
    cflag : out STD_LOGIC;
    oflag : out STD_LOGIC;
    Cin : in STD_LOGIC);
end ALU;

architecture Behavioral of ALU is
  signal internal_dataout : STD_LOGIC_VECTOR(31 downto 0);
begin
  process(data1, data2, aluop, internal_dataout, Cin)
  begin
    case aluop is
      when "0000" => internal_dataout <= data1 and data2;
      when "0001" => internal_dataout <= data1 or data2;
      when "0010" => internal_dataout <= data1 + data2 + Cin;
      when "0110" => internal_dataout <= data1 - data2;
      when "1100" => internal_dataout <= data1 nor data2;
      when "0111" =>
        if(data1<data2) then
          internal_dataout <= "00000000000000000000000000000001" ;
        else
          internal_dataout <= "00000000000000000000000000000000" ;
        end if ;
      when others => internal_dataout <= x"00000000";
    end case;

    dataout <= internal_dataout; -- Assign internal_dataout to dataout

    -----ZFLAG-----
    -- Assuming internal_dataout is an unconstrained array
    if internal_dataout = (internal_dataout'range => '0') then
      zflag <= '1';
    else
      zflag <= '0';
    end if;
  end process;
end Behavioral;

```

Figure 27 VHDL code of ALU

```

-----OFLAG-----
if (aluop = "0010") then
  if (data1(31) = '1' and data2(31) = '1' and internal_dataout(31) = '0') or -- Positive overflow
    (data1(31) = '0' and data2(31) = '0' and internal_dataout(31) = '1') then -- Negative overflow
    oflag <= '1'; -- Overflow
  else
    oflag <= '0'; -- No overflow
  end if;
elsif (aluop = "0110") then
  if (data1(31) = '0' and data2(31) = '1' and internal_dataout(31) = '1') then
    oflag <= '1'; -- Overflow
  else
    oflag <= '0'; -- No overflow
  end if;
else
  oflag <= '0'; -- Default assignment to avoid latches
end if;

-----CFLAG-----
if (aluop = "0010") then
  if (data1(31) = '1' or data2(31) = '1') then
    if (internal_dataout(31) = '1') then
      cflag <= '0'; -- No Carry
    else
      cflag <= '1'; -- Carry Exists
    end if;
  else
    cflag <= '0'; -- Ensuring cflag is assigned in all conditions
  end if;
--in subtraction
elsif (aluop = "0110") then
  if (data1(31) = '0' or data2(31) = '0') then
    if (not(internal_dataout(31)) = '1') then
      cflag <= '1'; -- Carry Exits
    else
      cflag <= '0'; -- No Carry
    end if;
  else
    cflag <= '0'; -- Ensuring cflag is assigned in all conditions
  end if;
else
  -- Default assignment to avoid latches
  cflag <= '0';
end if;
end process;
end Behavioral;

```

Figure 26 VHDL code of ALU bonus



## 2.8. TestBench for ALU Module

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY ALUTest IS
END ALUTest;

ARCHITECTURE behavior OF ALUTest IS

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

    COMPONENT ALU
    PORT(
        data1      : in  STD_LOGIC_VECTOR(31 downto 0);
        data2      : in  STD_LOGIC_VECTOR(31 downto 0);
        aluop      : in  std_logic_vector (3 downto 0);
        Cin        : in  std_logic;
        dataout     : out STD_LOGIC_VECTOR (31 downto 0);
        cflag      : out  STD_LOGIC;
        oflag      : out  STD_LOGIC;
        zflag      : out  STD_LOGIC
    );
    END COMPONENT;

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

    --Outputs
    signal dataout : std_logic_vector(31 downto 0);
    signal cflag   : std_logic;
    signal oflag   : std_logic;
    signal zflag   : std_logic;
    -- No clocks detected in port list. Replace <clock> below with
    -- appropriate port name

BEGIN

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

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

        wait for 1ns;

        --OR test case
        data1<= "11000000000000000000000000000000";
        data2<= "10100000000000000000000000000000";
        aluop <= "0001";
        wait for 10ns;
        report "Test2";
        assert(dataout = "11100000000000000000000000000000" and zflag = '0') report "2:Fail" severity error;

        wait for 1ns;

        --ADD test case (overflow = 1, cout = 0)
        data1<= "01110000000000000000000000000000";
        data2<= "01100000000000000000000000000000";
        aluop <= "0010";
        wait for 10ns;
        report "Test3";
        assert(dataout = "11010000000000000000000000000000" and oflag = '1' and cflag = '0' and zflag = '0') report "3:Fail" severity error;

        wait for 1ns;

        --ADD test case2 (zero = 1, cout = 1)
        data1<= "11110000000000000000000000000000";
        data2<= "00010000000000000000000000000000";
        aluop <= "0010";
        wait for 10ns;
        report "Test4";
        assert(dataout = "00000000000000000000000000000000" and oflag = '0' and cflag = '1' and zflag = '1') report "4:Fail" severity error;

        wait for 1ns;

        --SUB data2 test case1 (cout = 1)
        data1<= "00000000000000000000000000000111"; --data1= 7
        data2<= "00000000000000000000000000000110"; --data2= 6
        Cin <= '1'; --get 2s complement
        aluop <= "0110";
        wait for 10ns;
        report "Test5";
        assert(dataout = "00000000000000000000000000000001" and oflag = '0' and cflag = '1' and zflag = '0') report "5:Fail" severity error;

        wait for 1ns;

        --SUB data2 test case2 (cout = 0)
        data1<= "00000000000000000000000000000110"; --data1= 6
        data2<= "00000000000000000000000000000111"; --data2= 7
        Cin <= '1'; --get 2s complement
        aluop <= "0110";
        wait for 10ns;
        report "Test6";
        assert(dataout = "11111111111111111111111111111111" and oflag = '0' and cflag = '0' and zflag = '0') report "6:Fail" severity error;

        data1<= "00000000000000000000000000000111"; --data1= 7
        data2<= "00000000000000000000000000000110"; --data2= 6
        aluop <= "0111";
        wait for 10ns;
        report "Test7";
        assert(dataout = "00000000000000000000000000000000" and zflag = '1') report "7:Fail" severity error;

        data1<= "00000000000000000000000000000110"; --data1= 6
        data2<= "00000000000000000000000000000111"; --data2= 7
        aluop <= "0111";
        wait for 10ns;
        report "Test8";
        assert(dataout = "00000000000000000000000000000001" and zflag = '0') report "8:Fail" severity error;

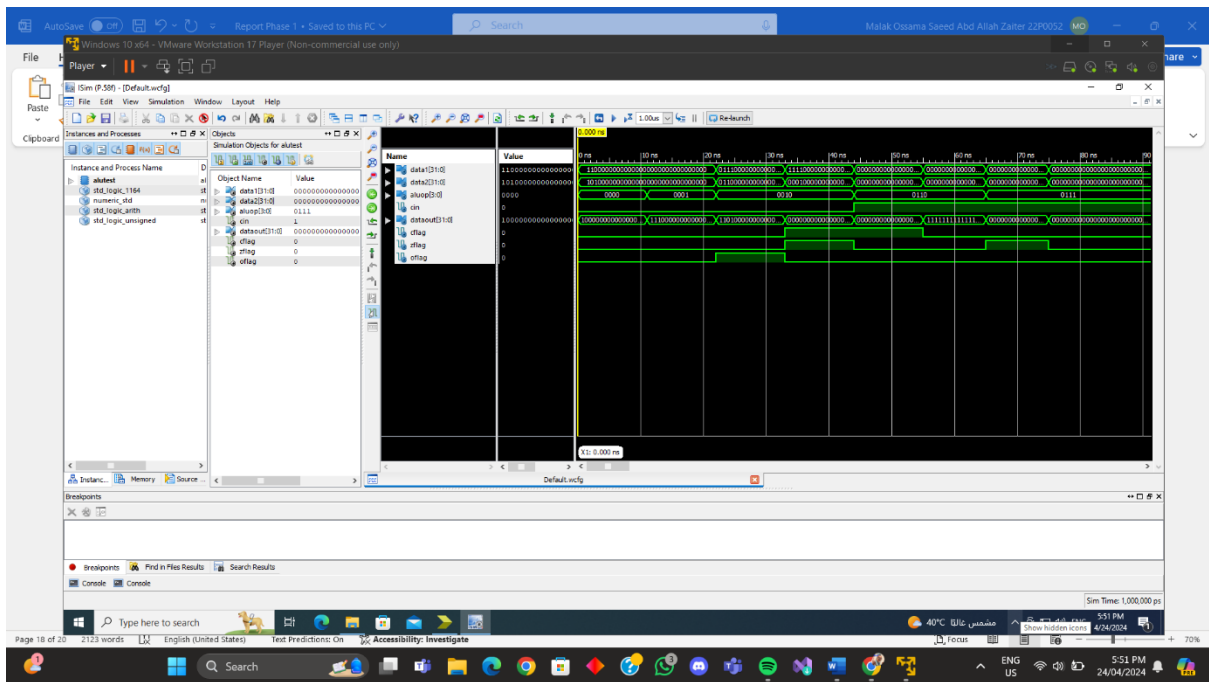
        wait for 1ns;

        report "Test Complete";
        wait;
    end process;

END;

```

Figure 28 TestBench ALU



at 10 ns: Note: Test1 (/alutest/).  
 at 21 ns: Note: Test2 (/alutest/).  
 at 32 ns: Note: Test3 (/alutest/).  
 at 43 ns: Note: Test4 (/alutest/).  
 at 54 ns: Note: Test5 (/alutest/).  
 at 65 ns: Note: Test6 (/alutest/).  
 at 75 ns: Note: Test7 (/alutest/).  
 at 85 ns: Note: Test8 (/alutest/).  
 at 86 ns: Note: Test Complete (/alutest/).

Figure 29 Test Cases Outcome

## 2.9. Create Datapath Module

Lastly, we need to create a datapath module that has both the RegisterFile and ALU components as follows:

- **ALU Instance:** The ALU is instantiated with the data1 and data2 operands and control signals aluop.
- **RegisterFile Instance:** The RegisterFile is instantiated with control signals for reading and writing.

```

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

entity CPU_Datapath is
    port (
        clk, reset: in STD_LOGIC;
        instr: in STD_LOGIC_VECTOR(31 downto 0);
        aluop: in STD_LOGIC_VECTOR(3 downto 0);
        zflag: out STD_LOGIC;
        write_ena: in STD_LOGIC;
        dataout: buffer STD_LOGIC_VECTOR(31 downto 0)
    );
end CPU_Datapath;

architecture Behavioral of CPU_Datapath is

    signal operand_data1: STD_LOGIC_VECTOR(31 downto 0);
    signal operand_data2: STD_LOGIC_VECTOR(31 downto 0);

    -- Declare signals for register file read/write select
    signal read_sel1: std_logic_vector(4 downto 0);
    signal read_sel2: std_logic_vector(4 downto 0);
    signal write_sel : std_logic_vector(4 downto 0);

    -----ALU 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);
        dataout : buffer STD_LOGIC_VECTOR (31 downto 0);
        zflag : out STD_LOGIC;
        cflag : out STD_LOGIC;
        oflag : out STD_LOGIC;
        Cin : in STD_LOGIC);
    end component;

    -----REGISTER_FILE COMPONENT-----
    component RegisterFile is
    Port (
        clk : in std_logic;
        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); --WriteRegister
        write_ena : in std_logic; --from controller write/read
        write_data: in std_logic_vector(31 downto 0);
        data1 : out std_logic_vector(31 downto 0); --ReadData1
        data2 : out std_logic_vector(31 downto 0) --ReadData2
    );
    end component;

    -----MAIN PROGRAM-----
begin
    UNIT1 : ALU Port Map(
        data1 => operand_data1,
        data2 => operand_data2,
        aluop => aluop,
        dataout => dataout,
        zflag => zflag,
        Cin => '0' ); -- Default value for oflag

    UNIT2 : RegisterFile Port Map(
        clk => clk,
        read_sel1 => read_sel1,
        read_sel2 => read_sel2,
        write_sel => write_sel,
        write_ena => write_ena,
        write_data => dataout,
        data1 => operand_data1,
        data2 => operand_data2);
end Behavioral;

```

Figure 30 VHDL code of Datapath

### 3. Contribution

<b>Name</b>	<b>I.D</b>	<b>Part Contributed In</b>	<b>Percentage</b>
Abdallah Belal Momen	22P0036	ALU (operations+flags) & Datapath	20%
Ahmed Khaled Hamdy	22P0024	ALU (operations+flags) & Datapath	20%
Malak Ossama Zaiter	22P0052	Registerfile (including modules & package) & Report	20%
Mohamed Khaled Mohamed	22P0041	Registerfile (including modules & package) & Report	20%
Yousef Sameh Shoman	22P0010	Registerfile (including modules & package) & Report	20%

### 4. Conclusion

The VHDL project successfully achieved the creation of a comprehensive datapath that integrates two critical components: the Register File and the Arithmetic Logic Unit (ALU).

One of the key highlights of the project was the development of the Register File. This module was designed by creating a package that includes a decoder, multiplexer, and flip-flop register (FlopR). These additions have substantially increased the module's efficiency and functionality.

Challenges encountered during the integration of the Register File and ALU were addressed through innovative problem-solving and design optimization, ensuring seamless interaction between these components. This was crucial for validating the functional correctness of the datapath under various simulation scenarios.

In conclusion, this project has demonstrated the effectiveness of VHDL in creating detailed and functional digital designs, providing a strong testament to the language's utility in practical applications in the field of digital engineering.

# Phase 2

## 1.Executive Summary

This report describes the overall development process for the enhancement of a MIPS CPU to execute an extended instruction set, including not only R-type instructions but also I-type (lw, sw, beq) and J instructions. The project involved the design, implementation, and simulation of the modified MIPS processor in VHDL. The main modules developed are the Register File and a 32-bit Arithmetic Logic Unit, which were adapted to handle the extended instruction set. The project shows the ability of VHDL for modeling and simulating advanced digital circuits and systems in general but with the purpose of educational applications and gaining an elementary understanding of microprocessor architecture.

### 1.1. Objectives

1. Implement the control module, which is responsible for all of the control signals.
2. Implement the Mips module by connecting the datapath with the control module .
3. Connect the Mips module with instruction and data memory module together.
4. Fill the memory module by a simple program.
5. The CPU should be able to execute this program.
6. Simulate the results and check the final results

## 2.Implementation Details:

### 2.1. Create a Control Module

The control unit (CU) is a component of a computer's central processing unit (CPU) that directs the operation of the processor. It tells the computer's memory, ALU, others how to respond to the instructions that have been sent to the processor.

The controller module consists of two main parts:

- ALU Decoder/Control:

Decides which operation will be performed by the ALU.

**Aluop(1:0):** ALUOp, which is a 2-bit control signal indicating a 00 (add for loads and stores), a 01 (subtract for branches), and a 10 (use the funct field).

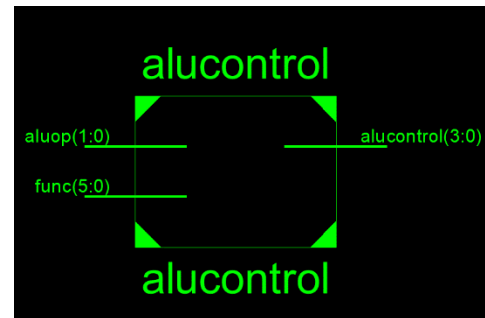


Figure 31 RTL Schematic of ALU Control

**Func(5:0):** funct field, which is a 6-bit signal as follows:

```
"100000" --add      "100101" --or
"100010" --sub      "101010" --slt
"100100" --and      "100111" --nor
```

- \* Remember that for R-type instructions, the opcode field is always zero and the funct field is used to determine the type of operation to perform.

**Output:** a 4-bit field that is fed into the ALU to select the operation to be performed as follows:

```
"0010" --add      "0001" --or
"0110" --sub      "0111" --slt
"0000" --and      "1100" --nor
```

Instruction opcode	ALUOp	Instruction operation	Funct field
LW	00	load word	XXXXXX
SW	00	store word	XXXXXX
Branch equal	01	branch equal	XXXXXX
R-type	10	add	100000
R-type	10	subtract	100010
R-type	10	AND	100100
R-type	10	OR	100101
R-type	10	set on less than	101010

Figure 32 Alu Decoder Function Table

- Main Decoder/Control:

Decodes the opcode—operation code—part of the instruction into control signals that manages the other components of the CPU, such as the Arithmetic Logic Unit, registers, and memory systems.

**op (5:0):** opcode is a 6-bit value that specifies the operation that the instruction will perform. This is the primary input to the decoder.

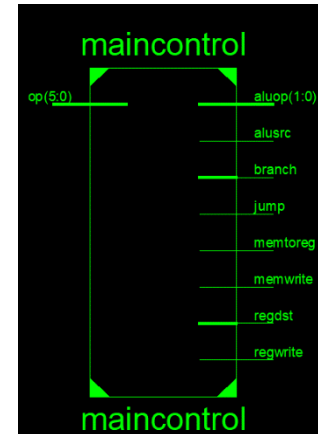


Figure 33 RTL Schematic of Main Control

**aluop (1:0):** These are control signals for the ALU operation. Depending on the opcode and function bits of the MIPS instruction, these two bits help specify which operation the ALU should perform.

**alusrc:** This is a control signal that determines the second operand of the ALU. If set to 0, the ALU takes the second operand from the register file. If set to 1, the ALU takes the second operand from the immediate value or other specific sources.

**branch:** This is a control signal that indicates whether the current instruction is a branch instruction. If set to 1, the processor checks the branch condition and possibly changes the instruction address according to the branch target.

**jump:** This control signal indicates whether the instruction is a jump. If this signal is high, the processor will jump to the address specified by the instruction.

**memtoreg:** This control signal is used to determine the source of data that will be written back into the register. If set to 1, the data to be written into the register comes from memory (used in load instructions). If set to 0, the data comes from the ALU result.

**memwrite:** This is a control signal that enables writing to the memory. If high, data from the register will be written to the address specified by the ALU.

**regdst:** This control signal decides which register will be the destination for the result. In MIPS, this could be either rt or rd, depending on the instruction type.

**regwrite:** This control signal allows writing into the register file. If high, the register specified by the instruction (determined by regdst) will be written with the data determined by memtoreg.

Input or output	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

Figure 34 Main Decoder Function Table

## - Controller Connection:

After creating the ALU decoder and the main decoder, we connect them to create the controller module:

```
entity alucontrol is
    port(
        func: in STD_LOGIC_VECTOR (5 downto 0);
        aluop: in STD_LOGIC_VECTOR(1 downto 0);
        alucontrol: out STD_LOGIC_VECTOR ( 3 downto 0)
    );
end alucontrol;

architecture Behavioral of alucontrol is

begin
    process (aluop,func)
    begin
        case aluop is
            when "00" => alucontrol <= "0010"; --add (for lw/sw/addi)
            when "01" => alucontrol <= "0110"; --sub (for beq)
            when others =>
                case func is --Rtypeinst
                    when "100000" => alucontrol <= "0010" ; --add
                    when "100010" => alucontrol <= "0110" ; --sub
                    when "100100" => alucontrol <= "0000" ; --and
                    when "100101" => alucontrol <= "0001" ; --or
                    when "101010" => alucontrol <= "0111" ; --slt
                    when "100111" => alucontrol <= "1100" ; --nor
                    when others => alucontrol <= "----" ; --illegal
                end case;
            end case;
        end process;
    end Behavioral;
```

**Figure 36 ALU Control Code**

```
entity maincontrol is
    Port (
        op: in STD_LOGIC_VECTOR(5 downto 0);
        memtoreg, memwrite, branch, alusrc, regdst, regwrite, jump : out STD_LOGIC;
        aluop: out STD_LOGIC_VECTOR(1 downto 0)
    );
end maincontrol;

architecture Behavioral of maincontrol is

    signal controls: STD_LOGIC_VECTOR(8 downto 0);

begin
    process(op)
    begin
        case op is
            when "000000" => controls <= "110000010"; --Rtype
            when "100011" => controls <= "101001000"; --lw
            when "101011" => controls <= "001010000"; --sw
            when "000100" => controls <= "000100001"; --beq
            when "001000" => controls <= "101000000"; --addi
            when "000010" => controls <= "000000100"; --j
            when others => controls <= "-----"; --illegal op
        end case;
    end process;

    (regwrite, regdst, alusrc, branch, memwrite, memtoreg, jump, aluop(1),
    aluop(0)) <= controls;

end Behavioral;
```

**Figure 37 Main Control Code**

```
port(
    op, funct: in STD_LOGIC_VECTOR(5 downto 0);
    zero: in STD_LOGIC;
    memtoreg, memwrite: out STD_LOGIC;
    pcsrc, alusrc: out STD_LOGIC;
    regdst, regwrite: out STD_LOGIC;
    jump: out STD_LOGIC;
    alucontrolsignal: out STD_LOGIC_VECTOR(3 downto 0));

end controller;

architecture struct of controller is

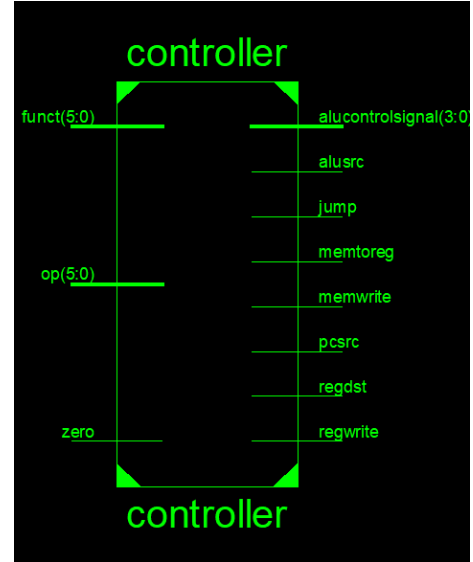
    signal aluop : std_logic_vector ( 1 downto 0);
    signal branch : std_logic;

begin

    maincontrol1 : maincontrol port Map ( op, memtoreg, memwrite, branch, alusrc, regdst, regwrite, jump, aluop );
    alucontrol1 : alucontrol port map ( funct, aluop, alucontrolsignal);

    pcsrc <= branch and zero;
```

**Figure 38 Controller Code**



**Figure 35 RTL Schemtaic of Controller**



## 2.2. Create a Data Memory Module

It is designed to interface to a system's data bus, allowing for both read and write operations to memory locations based on a specified address.

**a(31:0):** This is the address bus. It has 32 lines, indicating that the data memory can address up to  $2^{32}$  different memory locations, which is typical for a 32-bit address space.

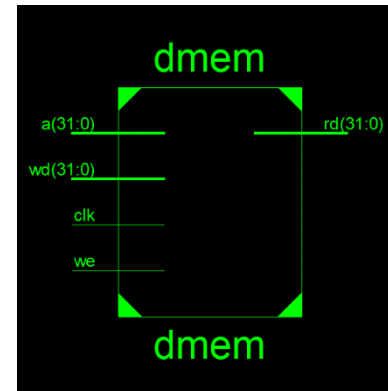


Figure 39 RTL Schematic of DMEM

**wd(31:0):** This stands for "write data". It is a 32-bit bus that carries the data to be written into the memory at the location specified by the address bus.

**rd(31:0):** This stands for "read data". It is a 32-bit bus that carries data read from the memory at the location specified by the address bus back to other parts of the system.

**clk:** This is the clock signal for the memory. Memory operations (read or write) are typically synchronized to the rising or falling edge of the clock signal to ensure data stability and integrity.

**we:** This stands for "write enable". It is a control signal used to indicate when the data should be written to the memory. If we is high (1), the memory will perform a write operation at the next clock edge; if low (0), no write operation occurs, and the memory will be in a read or idle state.

```
entity dmem is
  port(
    clk, we: in STD_LOGIC;
    a, wd: in STD_LOGIC_VECTOR (31 downto 0);
    rd: out STD_LOGIC_VECTOR (31 downto 0)
  );
end;

architecture behave of dmem is
begin
  process is
    type ramtype is array (63 downto 0) of STD_LOGIC_VECTOR(31 downto 0);
    variable mem: ramtype;
    begin

      for i in 1 to 1000 loop
        if rising_edge(clk) then
          if (we='1') then
            mem (CONV_INTEGER('0'&a(7 downto 2))) := wd;
          end if;
        end if;
        rd <= mem (CONV_INTEGER('0'&a (7 downto 2)));
        wait on clk, a;
      end loop;
    end process;
end;
```

Figure 40 DMEM Code

## 2.3. Create an Instruction Memory Module

**mem\_file:** A file variable used to read data from a file (memfile.dat).

**L:** A line variable used for handling lines read from the file.

**ch:** A character variable used to parse individual characters from lines.

**i, index, result:** Integer variables used for looping and indexing.

**ramtype:** A custom data type, which is an array of 64 32-bit std\_logic\_vectors.

**mem:** A variable of type ramtype, representing the memory itself.

Memory Initialization:

- A loop sets each element of mem to 0. This clears the memory initially, ensuring there are no residual values.

File Reading and Memory Population

- Opens a file from a specified path "E:/memfile.dat" for reading.
- Reads the file line by line until the end of the file is reached.
- Each line is expected to contain hexadecimal characters, which are parsed and converted to a binary representation:
- Characters between '0' and '9' are directly converted to their numerical equivalents.
- Characters between 'a' and 'f' are converted to values from 10 to 15, corresponding to hexadecimal digits.
- A nested loop reads 8 characters (representing 32 bits, as each hex digit corresponds to 4 bits) from each line, converting and storing them in the mem array.
- The data is stored into mem using the std\_logic\_vector function and to\_unsigned conversion for correct bit placement.

Memory Reading

- Simulates reading the memory content based on an input signal a, which appears to be an address signal.
- The address from signal a is converted to an integer and used to index into the mem array.
- The value at the indexed location is assigned to an output signal rd.
- This loop continuously responds to changes on the address signal a.

## Error Handling

- The process includes error handling for format errors during file reading, which terminates the simulation/report with an error message if unexpected characters are found.

```
entity imem is -- instruction memory
port(a: in STD_LOGIC_VECTOR(5 downto 0);
rd: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of imem is
begin
process is
file mem_file: TEXT;
variable L: line;
variable ch: character;
variable i, index, result: integer;
type ramtype is array (63 downto 0) of STD_LOGIC_VECTOR(31 downto 0);
variable mem: ramtype;
begin
-- initialize memory
for i in 0 to 63 loop -- set all 0
mem(i) := (others => '0');
end loop;
index := 0;
FILE_OPEN (mem_file, "E:/memfile.dat", READ_MODE);
while not endfile(mem_file) loop
readline(mem_file, L);
result := 0;
for i in 1 to 8 loop
read (L, ch);
if '0' <= ch and ch <= '9' then
result := character'pos(ch) - character'pos('0');
elsif 'a' <= ch and ch <= 'f' then
result := character'pos(ch) - character'pos('a')+10;
else report "Format error on line" & integer'
image(index) severity error;
end if;
mem(index) (35-i*4 downto 32-i*4) := std_logic_vector(to_unsigned(result,4));
end loop;
index := index + 1;
end loop;
-- read memory
for i in 1 to 1000 loop
rd <= mem(CONV_INTEGER(a));
wait on a;
end loop;
end process;
end;
```

*Figure 41 IMEM Code*

## 2.4. Create a Sign Extender Module

This sign extension is crucial in computing architectures, especially when widening smaller bit-width data for operations that require a larger bit-width, ensuring that the sign of the data is correctly maintained.

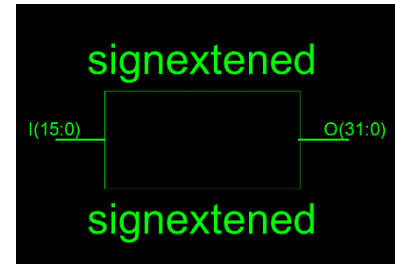


Figure 42 RTL Schematic of SignExtend

**Output O:** assigned based on the most significant bit (MSB) of input a (i.e., I(15)). If I(15) is '1' (indicating the number is negative in two's complement notation), the upper 16 bits of y are set to all ones (X"ffff"). This effectively extends the sign (1) across the upper half of y. If I(15) is '0' (indicating the number is positive), the upper 16 bits of y are set to all zeros (X"0000"), preserving the non-negative value.

```
entity signextended is
    port(
        I: in STD_LOGIC_VECTOR(15 downto 0);
        O: out STD_LOGIC_VECTOR(31 downto 0)
    );
end signextended;
architecture Behavioral of signextended is
begin
    O <= X"ffff" & i WHEN I(15) = '1' ELSE
        X"0000" & i;
end Behavioral;
```

## 2.5. Create a Shift Left x2 Module

In MIPS, the jump and branch instructions use an immediate value that needs to be word-aligned (i.e., the address must be a multiple of 4). The SL2 operation ensures this by shifting the immediate value left by 2 bits, multiplying it by 4.

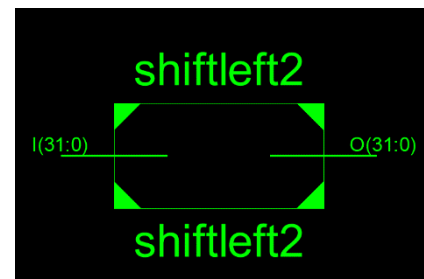


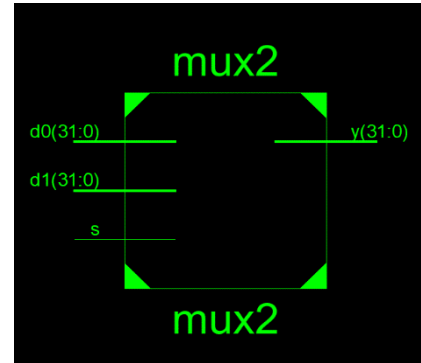
Figure 43 RTL Schematic of ShiftLeft2

**I <= O(29 downto 0) & "00";** This line takes the lower 30 bits of a and concatenates them with two 0s at the least significant bit positions. This effectively shifts all bits in a two positions to the left, with the two least significant bits filled with 0s.

```
entity shiftleft2 is
    Port ( I : in  STD_LOGIC_VECTOR (31 downto 0);
          O : out  STD_LOGIC_VECTOR (31 downto 0));
end shiftleft2;
architecture Behavioral of shiftleft2 is
begin
    o <= i(29 downto 0) & "00";
end Behavioral;
```

## 2.6. Create a MUX 2x1 Module

This multiplexer allows for dynamic selection between two data inputs based on a control signal, making it highly useful in scenarios requiring conditional data routing in digital circuits, such as in ALU operations or control signal management.



*Figure 44 RTL Schematic of MUX2*

**width:** This is a parameter that defines the size of the input and output vectors. The parameter can vary, which allows the multiplexer to be used with different data widths.

**d0, d1:** These inputs represent the two data options from which one will be selected.

**s:** The select line for the multiplexer, it decides which of the inputs d0 or d1 gets passed to the output.

**y:** This is the output port of the multiplexer, it carries the selected input to other parts of the circuit.

```
entity mux2 is generic(width: integer:=32);
    port(d0, d1: in STD_LOGIC_VECTOR(width-1 downto 0);
          s: in STD_LOGIC;
          y: out STD_LOGIC_VECTOR(width-1 downto 0));
END Mux2;
ARCHITECTURE MuxArch OF Mux2 IS
BEGIN
    y <= d1 WHEN s='1' ELSE d0;
END MuxArch;
```

## 2.7. Create a Datapath Module

**Inputs:** clock (clk), reset signal (reset), instruction (instr), and data read from memory (readdata), among others controlling the behavior like memtoreg, jumpsrc, pcsrc, alusrc, regwrite, and regdst.

**Outputs:** program counter (pc), result of the ALU operation (aluout), and data to be written back to memory (writedata). There's also a zero flag output from the ALU indicating a zero result.

Program Counter Logic:

- MYREGISTER is a register that holds the program counter (pc), updated every clock cycle.
- adder instances (pcadd1 and pcadd2) calculate the next value of the program counter (pcplus4 and pcbranch).
- mux2 instances (pcbrmux and pcmux) decide the actual next value of the program counter based on branching or jumping conditions.

Register File:

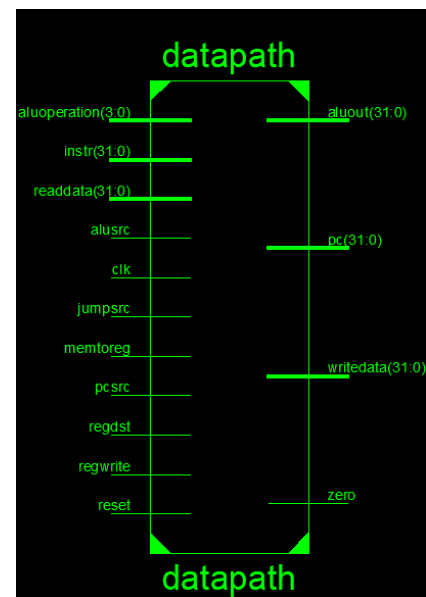
- The registerfile component interfaces directly with the register set, controlled by inputs from the instruction and regwrite signal.
- The mux2 instance wrmux decides which register is the write register based on the regdst signal.

ALU Operations:

- The alu component performs arithmetic and logic operations as dictated by the aluoperation signal, operating on inputs selected by another mux2 (srcbmux).
- The results of the ALU operations are potentially routed back into the register file or elsewhere depending on the memtoreg setting, managed by resmux.

Data and Instruction Handling:

- signextended (likely intended to be signextended) expands immediate values from the instruction.
- shiftright2 shifts the immediate value, typically for address calculations in branching.



*Figure 45 RTL Schematic of Datapath*

```

entity datapath is
  Port (
    clk      : in  STD_LOGIC;
    reset    : in  STD_LOGIC;
    readdata : in  STD_LOGIC_VECTOR(31 downto 0);
    instr    : in  STD_LOGIC_VECTOR (31 downto 0);
    memtoreg : in  STD_LOGIC;
    jumpsrc  : in  STD_LOGIC;
    pcsrc    : in  STD_LOGIC;
    alusrc   : in  STD_LOGIC;
    regwrite : in  STD_LOGIC;
    regdst   : in  STD_LOGIC;
    aluoperation : in STD_LOGIC_VECTOR (3 downto 0);
    zero     : out STD_LOGIC;
    pc       : out STD_LOGIC_VECTOR(31 downto 0);
    aluout   : out STD_LOGIC_VECTOR (31 downto 0);
    writedata : out STD_LOGIC_VECTOR (31 downto 0)
  );
end datapath;

architecture Behavioral of datapath is
  signal writereg: STD_LOGIC_VECTOR(4 downto 0);
  signal pcjump, pcnext, pcnextbtr, pcplus4, pcbranch : STD_LOGIC_VECTOR(31 downto 0);
  signal signexout, slout: STD_LOGIC_VECTOR(31 downto 0);
  signal srca, aluin , datal , data2 , pcsignal , aluoutsignal: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- PC update logic
  pc <= pcsignal;
  aluout <= aluoutsignal;
  writedata <= data2;

  -- Jump address calculation
  pcjump <= pcplus4(31 downto 28) & instr(25 downto 0) & "00";

  -- PC register
  pcreg: MYREGISTER
    port map (pcnext, pcsignal, '1', clk, reset);

  -- Increment PC
  pcadd1: adder
    port map (pcsignal, X"00000004", pcplus4);

  -- Shift left immediate (Branch offset preparation)
  immsh: shiftleft2
    port map (signexout, slout);

  -- Calculate branch target address
  pcadd2: adder
    port map (pcplus4, slout, pcbranch);

  -- Branch decision multiplexer
  pcbrmux: mux2 generic map (32)
    port map (pcplus4, pcbranch, pcsrc, pcnextbtr);

  -- Jump decision multiplexer
  pcmux: mux2 generic map (32)
    port map (pcnextbtr, pcjump, jumpsrc, pcnext);

  -- Register file operations
  rf: registerfile
    port map (clk, instr(25 downto 21), instr(20 downto 16), writereg, regwrite, srca, datal, data2);

  -- Register write multiplexer (Rt or Rd)
  wrmux: mux2 generic map (5)
    port map (instr(20 downto 16), instr(15 downto 11), regdst, writereg);

  -- Result source multiplexer (ALU or Memory)
  resmux: mux2 generic map(32) port map(aluoutsignal , readdata, memtoreg, srca);
  -- Sign extend the immediate value
  se: signextended
    port map (instr(15 downto 0), signexout);

  -- ALU source multiplexer (Rt or Immediate)
  srcbmux: mux2 generic map (32)
    port map (data2, signexout, alusrc, aluin);

  -- ALU operations
  alu32: alu
    port map (datal, aluin, aluoperation, aluoutsignal, zero);
end Behavioral;

```

**Figure 46 Datapath Code**

## 2.8. Create a MIPS Module

The mips entity represents the core of a single-cycle MIPS processor. This entity is defined with the following interface:

### Inputs:

- clk (Clock Signal): Standard logic input that drives the operation of the processor, synchronizing all activities.
- reset (Reset Signal): Standard logic input used to initialize or reset the processor, bringing it to a known state.
- instr (Instruction): A 32-bit input signal carrying the current instruction to be executed.
- readdata (Read Data): A 32-bit input signal that brings data from the data memory into the processor.

### Outputs:

- pc (Program Counter): A 32-bit output that holds the address of the current instruction being executed.
- aluout (ALU Output): A 32-bit output that holds the result of the arithmetic logic unit (ALU) computation.
- writedata (Write Data): A 32-bit output that holds data to be written into the data memory.
- memwrite (Memory Write Signal): A control signal output that indicates when a write operation should occur in data memory.

### Control Signals:

- memtoreg, jumpsrc, pcsrc, alusrc, regwrite, regdst: Control signals generated by the controller to manage the flow of data and operations within the processor.
- zero: A flag signal from the ALU indicating a zero result which can affect branch decisions.
- alucontrolsignal: A 4-bit signal dictating specific operations within the ALU.

### Components:

- ctrl: This is the controller component (controller), which interprets parts of the instruction to produce control signals. It maps portions of the instruction (instr(31 downto 26) and instr(5 downto 0)) along with the zero signal from the ALU to control outputs that orchestrate the processor's operation.
- dpath: The datapath component (datapath) executes the core functions of the processor under the direction of the control signals. It includes everything from managing the program counter, executing instructions in the ALU, handling data memory interactions, and writing back to the register file.

### System Integration

- The controller and datapath are connected in a way that allows the control signals to dictate the operation of the entire processor. The instruction bits and the zero outcome

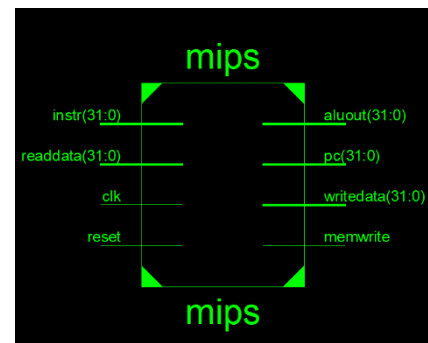


Figure 47 RLT Schematic of MIPS



from the ALU inform the controller, which in turn adjusts the control signals based on the current state and instruction. The datapath utilizes these signals to execute instructions, manipulate data, perform computations, and manage memory operations. All activities are synchronized by the clock signal, and the processor can be reset to an initial state using the reset signal.

- This design facilitates a typical MIPS instruction cycle in a single clock cycle, adhering to the principles of a single-cycle processing architecture. The processor supports basic operations such as arithmetic calculations, logical operations, memory access, and conditional branching.

```
entity mips is -- single cycle
port(clk, reset: in STD_LOGIC;
      instr: in STD_LOGIC_VECTOR(31 downto 0);
      readdata: in STD_LOGIC_VECTOR(31 downto 0);
      pc: out STD_LOGIC_VECTOR(31 downto 0);
      aluout, writedata: out STD_LOGIC_VECTOR(31 downto 0);
      memwrite: out STD_LOGIC
    );
end;

architecture Behavioral of MIPS is

signal memtoreg, jumpsrc, pcsrc, alusrc, regwrite, regdst: std_logic;
signal zero: std_logic;
signal alucontrolsignal: STD_LOGIC_VECTOR(3 downto 0);
begin

ctrl: controller port map ( instr(31 downto 26), instr( 5 downto 0) , zero , memtoreg , memwrite , pcsrc
                          , alusrc , regdst , regwrite , jumpsrc , alucontrolsignal );
dpath: datapath port map ( clk , reset , readdata , instr , memtoreg, jumpsrc , pcsrc, alusrc, regwrite, regdst, alucontrolsignal , zero , pc,
                          aluout, writedata );

end Behavioral;
```

**Figure 48 MIPS Code**

## 2.9. Create the Main Module

The main entity, as described, acts as a container for the MIPS processing system, managing the flow of data and control signals between the processor, instruction memory, and data memory.

### Inputs:

- clk (clock signal): Drives the timing of the entire system, synchronizing operations.
- reset: Used to initialize or reset the system to a known state.

### Outputs:

- writedata, dataadr: Respectively the data to be written to memory and the address at which data should be written. Both are 32-bit wide.
- memwrite: A control signal that indicates when a write operation should occur in the memory.

### Signals:

- memwrite1, dataadr1, writedata1: Intermediate signals used to connect the MIPS core to the data memory and the entity outputs.
- pc (Program Counter), instr (Instruction), readdata: Core signals in a MIPS architecture, where pc represents the address of the current instruction, instr is the current instruction fetched, and readdata is data read from memory.

### Component Instantiations and Interconnections:

- Instruction Memory (imem):
  - Maps a portion of the pc (specifically bits 7 down to 2) to instr. This module is responsible for fetching instructions based on the program counter.
- MIPS Core (mips):
  - The core component of the system, connected to the clock and reset lines. It receives the current instruction (instr), provides outputs for the next program counter (pc), memory address for data operations (dataadr1), data to write (writedata1), and memory write control (memwrite1).
- Data Memory (dmem):
  - Interfaces with the MIPS core to handle data storage and retrieval. It is controlled by the memwrite1 signal, uses dataadr1 for the address, and writedata1 for the data input. The data output is connected to readdata.

### Signal Assignments:

- The signals memwrite, dataadr, and writedata of the main entity are directly assigned from their corresponding internal signals (memwrite1, dataadr1, writedata1). These assignments link the internal processing of the MIPS core and memory components to the external interfaces of the main entity.

```

entity main is
port(clk, reset: in STD_LOGIC;
      writedata, dataadr: out STD_LOGIC_VECTOR(31 downto 0);
      memwrite: out STD_LOGIC
    );
end main;

architecture Behavioral of main is

signal memwritel: STD_LOGIC;
signal pc,instr,readdata,dataadr1,writedata1: STD_LOGIC_VECTOR (31 downto 0);

begin
memwrite<=memwritel;
dataadr<=dataadr1;
writedata<= writedata1;

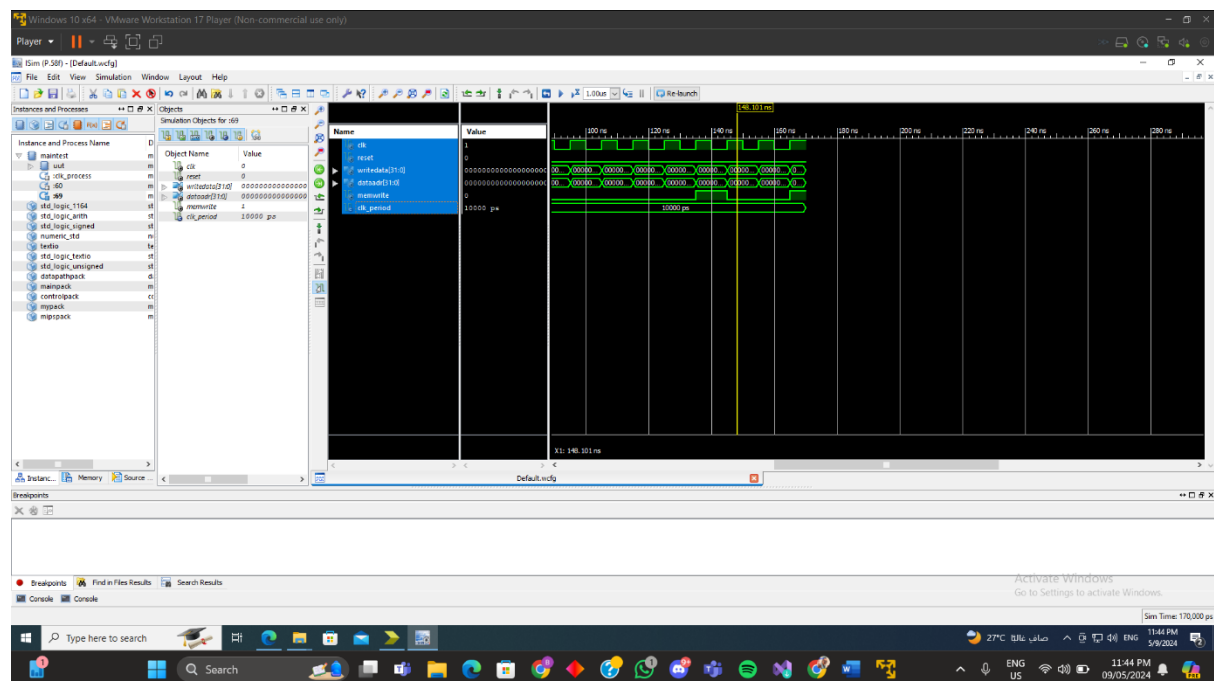
InstructionMemory: imem port map(pc(7 downto 2),instr);
MIPSS: mips port map(clk,reset,instr,readdata,pc,dataadr1,writedata1,memwritel);
DataMemory: dmem port map(clk,memwritel,dataadr1,writedata1,readdata);

end Behavioral;

```

Figure 49 Main Code

## Simulation Results



**\*\* Failure:NO ERRORS: Simulation succeeded**  
**User(VHDL) Code Called Simulation Stop**  
**In process maintest.vhd:69**

INFO: Simulator is stopped.

**ISim> A**

### 3. Contribution

<b>Name</b>	<b>I.D</b>	<b>Part Contributed In</b>	<b>Percentage</b>
Abdallah Belal Momen	22P0036	ALU (operations+flags), Datapath1, Controller & Report	20%
Ahmed Khaled Hamdy	22P0024	ALU (operations+flags), Datapath1, muxs, adders & Report	20%
Malak Ossama Zaiter	22P0052	Registerfile (including modules & package), Datapath2, MIPS, Integration with D-MEM & I-MEM & Report	20%
Mohamed Khaled Mohamed	22P0041	Registerfile (including modules & package), Datapath2, MIPS, Integration with D-MEM and I-MEM & Report	20%
Yousef Sameh Shoman	22P0010	Registerfile (including modules & package), Datapath2, MIPS, Integration with D-MEM & I-MEM & Report	20%

### 4. Conclusion

With the successful completion of Phase 2, the designed MIPS processor became a great deal more competent in its abilities; it now supports a larger set of instructions and is robust in its integration and simulation results. The project not only fulfilled the academic requirements but also provided a practical, hands-on experience in CPU design using VHDL that prepared its team members for more advanced work within the domains of computer architecture and hardware design.

## 5. Appendix

The ALU has been slightly changed from phase 1 to phase 2 as the following figure suggests.

```
entity ALU is
  port(
    data1, 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
  -- Intermediate signals for ALU operations
  signal negated_data2: std_logic_vector(31 downto 0);
  signal negated_data1: std_logic_vector(31 downto 0);
  signal result_add: std_logic_vector(31 downto 0);
  signal result_slt: std_logic_vector(31 downto 0);
  signal alu_result: std_logic_vector(31 downto 0);

begin
  -- Conditional negation based on ALU operation control bits
  negated_data2 <= NOT data2 when aluop(2) = '1' else data2;
  negated_data1 <= NOT data1 when aluop(3) = '1' else data1;

  -- Calculate the addition or subtraction result
  result_add <= data1 + negated_data2 + aluop(2);

  -- Set Less Than operation
  result_slt <= "00000000000000000000000000000000" & result_add(31);

  -- Output selection based on ALU operation control bits
  alu_result <= (negated_data1 AND negated_data2) when aluop(1 downto 0) = "00" else
    (data1 OR negated_data2) when aluop(1 downto 0) = "01" else
    result_add when aluop(1 downto 0) = "10" else
    result_slt when aluop(1 downto 0) = "11";

  -- Assign the computed result to the output
  dataout <= alu_result;

  -- Zero flag determination
  zflag <= '1' when alu_result = x"00000000" else '0';

end Behavioral;
```