MIPS Processor Design in VHDL

Phase 1

Technical Project Report

Logic Design and Computer Organization – CSE221

1 Introduction

This report documents the design and implementation of a simplified MIPS processor using VHDL and Xilinx ISE 14.5. The processor implements a subset of the MIPS instruction set, focusing on R-type instructions (AND, OR, ADD, SUB, NOR). This report aims to provide a practical understanding of fundamental computer architecture concepts through the design and simulation of key processor components:

- **Register File:** A 32-bit register file with read and write capabilities.
- **ALU:** A 32-bit arithmetic logic unit supporting the specified R-type instructions.
- **Control Unit:** A control unit responsible for instruction decoding and control signal generation.

The report details the design process, including the analysis of the VHDL code for each component, simulation results demonstrating correct instruction execution, and a discussion of design choices and challenges encountered.

This work describes the design of a simplified MIPS processor and some guidelines for its implementation in VHDL. The outcome will be an implementation of the simplified MIPS processor, which will be tested through simulation.

2 MIPS Architecture

The MIPS (Microprocessor without Interlocked Pipeline Stages) architecture is a Reduced Instruction Set Computing (RISC) architecture widely used. This architecture defines 32 general purpose registers. The first register $r0 always contains value zero. It is characterized by its simple instruction set, load-store architecture (data must be loaded into registers before operations), and a fixed-length 32-bit instruction format. There are 3 instruction types: I-type (immediate), R-type (register), J-type (jump).

This phase of the project focuses on the implementation of R-type instructions, which perform arithmetic and logical operations on data within registers. The general format of an R-type instruction is as follows:

opcode	rs	rt	rd	Shift	Function
6	5	5	5	5	6

R-Type instruction

The `opcode` field identifies the instruction as an R-type, while the `funct` field specifies the specific operation to be performed (e.g., add, subtract, AND). The `rs` and `rt` fields indicate the source registers, and the `rd` field specifies the destination register. The `shamt` field is used for shift operations.

3 Design and Implementation

This section details the design and implementation of the simplified MIPS processor. It outlines the top-level architecture and then delves into the individual components: the register file, the ALU, the control unit, and the data path. VHDL code snippets are provided for key modules to illustrate the implementation.

3.1 Top-Level Design

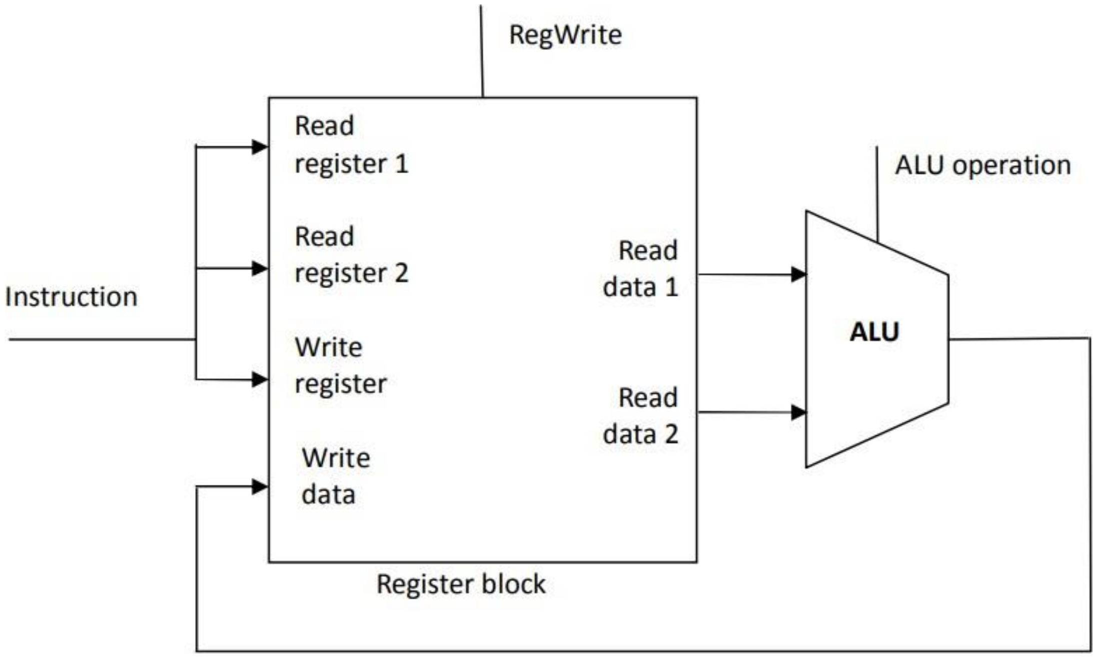


Figure 1 R-type Datapath

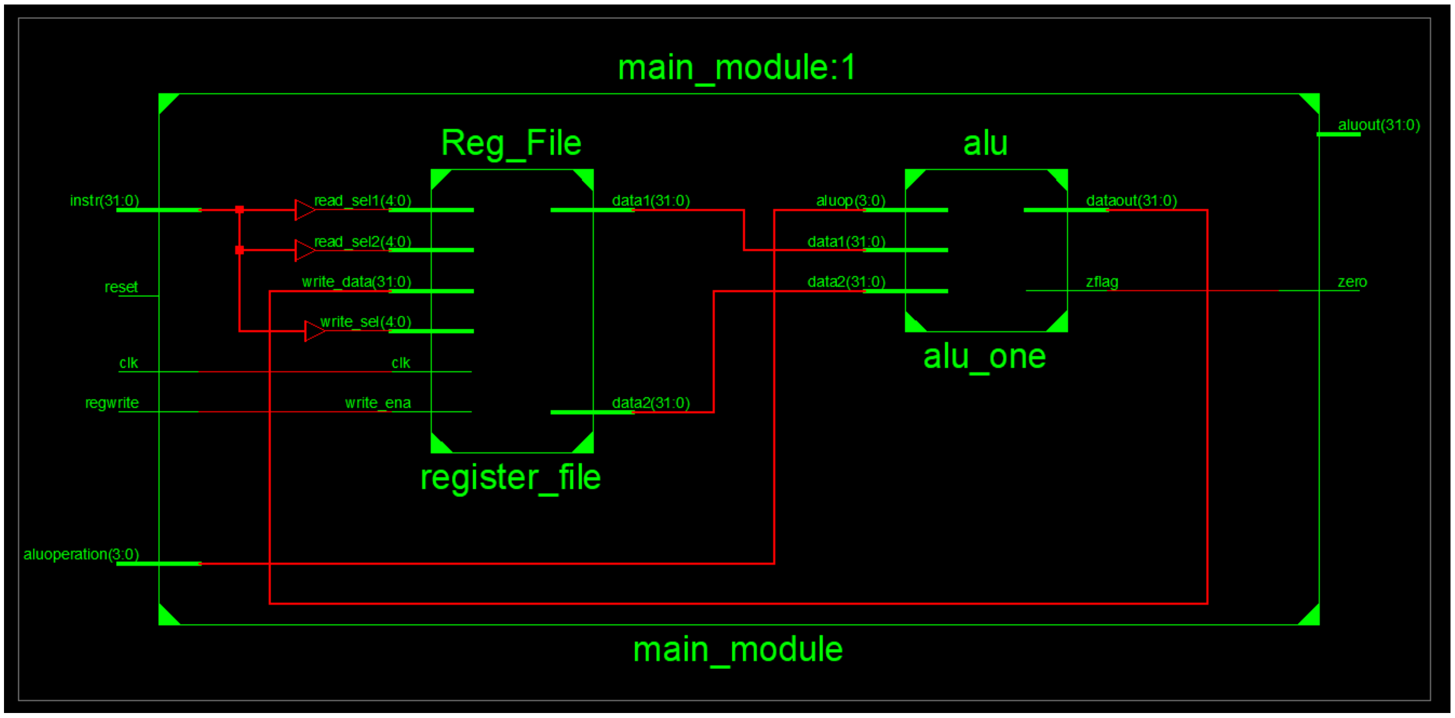
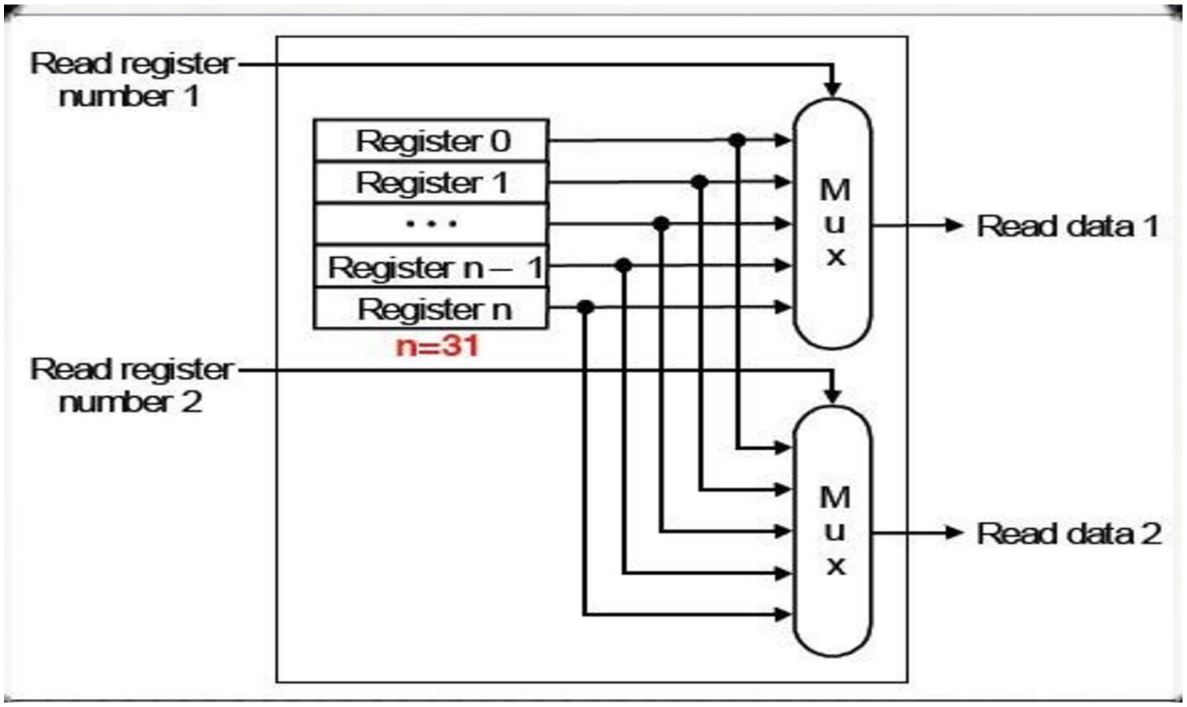
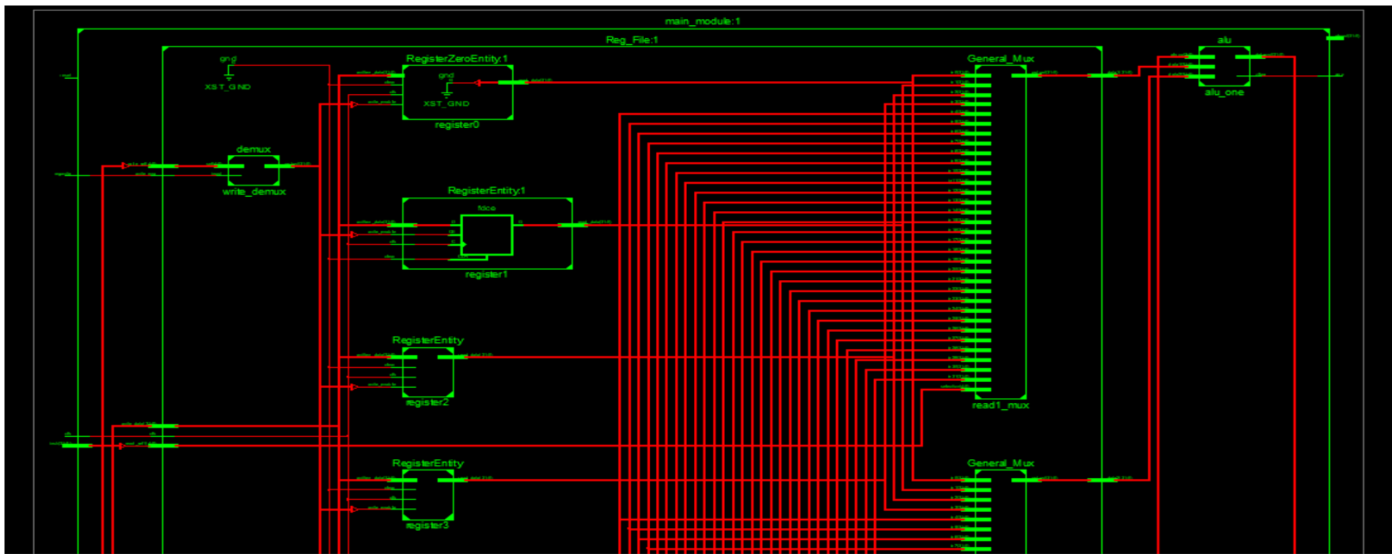
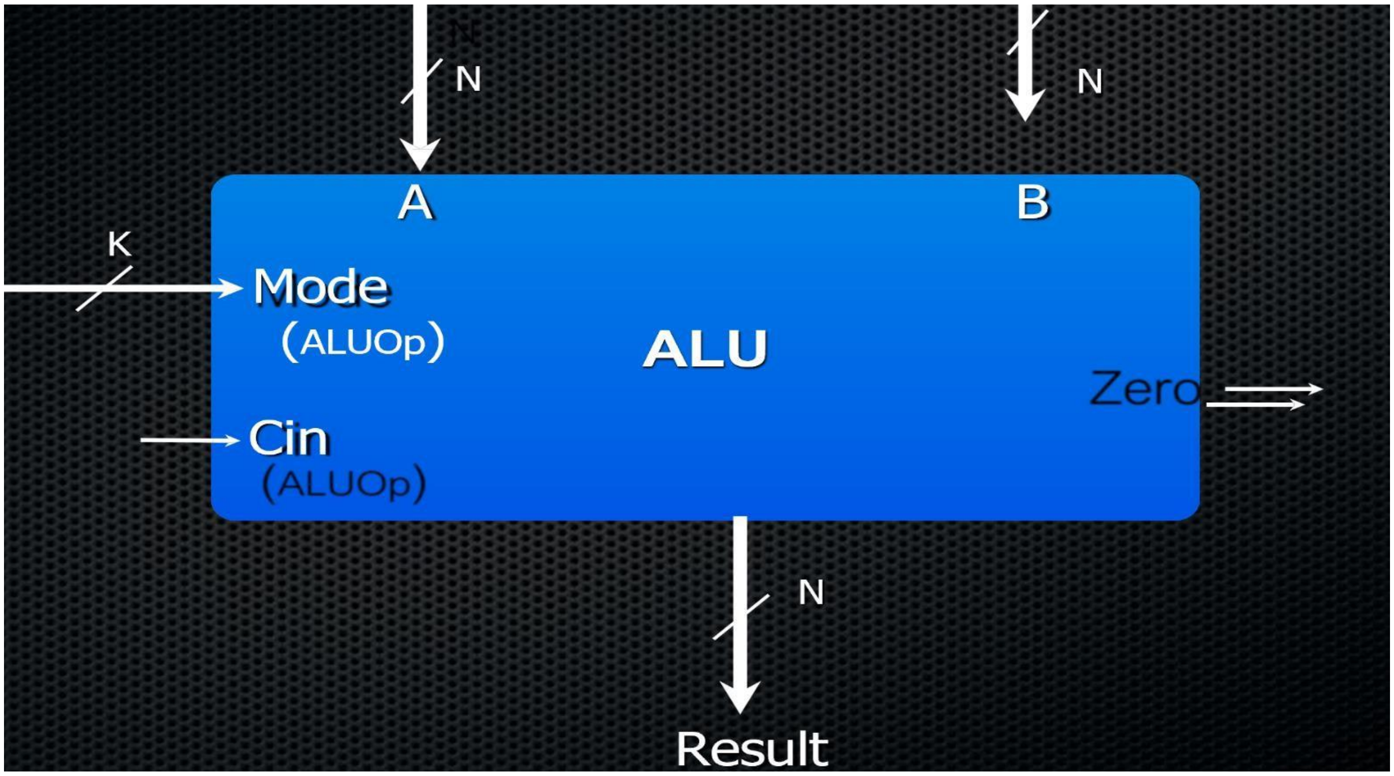
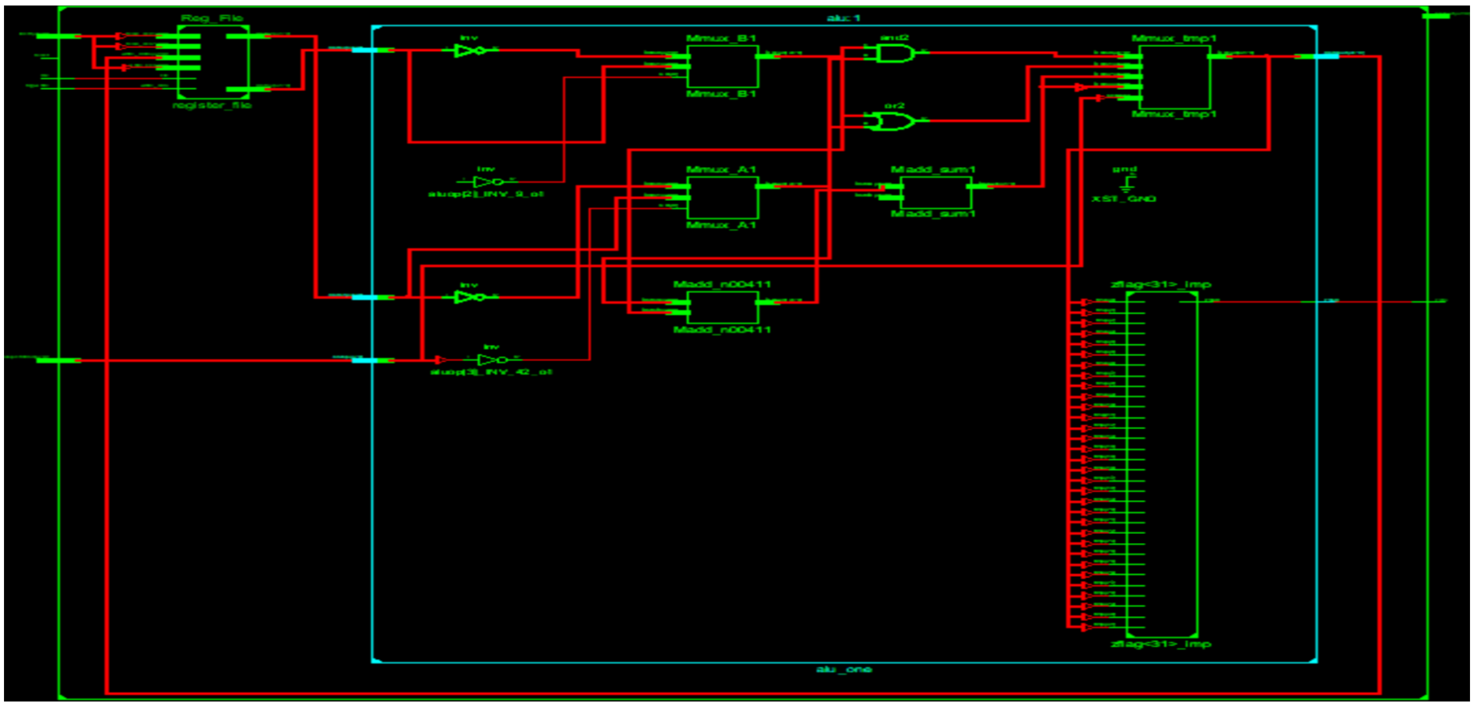


Figure 2 RTL Schematic


Read register number 1
Register 0
Register 1
. . .
Register n – 1
Register n
n=31
Read register number 2
Mux
Read data 1
Mux
Read data 2


main_module:1
Reg_File:1
RegisterZeroEntity:1
gnd
XST_GND
register0
demux
write_demux
RegisterEntity:1
fdce
register1
RegisterEntity
register2
RegisterEntity
register3
General_Mux
read1_mux
General_Mux
alu
alu_one


N
N
A
B
K
Mode
(ALUOp)
ALU
Zero
Cin
(ALUOp)
N
Result


Reg_File
register_file
alu:1
inv
Mmux_B1
and2
or2
Mmux_tmp1
Mmux_A1
Madd_sum1
gnd
XST_GND
aluop2_INV_9_o1
Madd_n00411
zflag<31>_imp
aluop3_INV_42_o1
alu_one

The top-level design of the MIPS processor comprises two main components to execute R-type Instructions:

1. **Register File:** Stores 32 general-purpose registers, each 32 bits wide. It supports simultaneous read and write operations.
2. **ALU:** Performs arithmetic and logical operations on data fetched from the register file.

The block diagram above illustrates the interconnection of these components.

3.2 Register File

The VHDL code for the register file entity and architecture is shown below:

3.2.1 Key Functionality:

- **Read Operation**: Two 5-bit read select signals (read_sel1, read_sel2) specify the addresses of the registers to be read. The data from these registers is then output on data1 and data2 respectively. This allows the CPU to fetch two operands concurrently, which is crucial for efficient instruction execution.

- **Write Operation**: A 5-bit write select signal (write_sel) indicates the register to be written to. The write_ena signal enables the write operation. When write_ena is '1', the data on write_data is written to the register specified by write_sel on the rising edge of the clock signal (clk).

3.2.2 Implementation Details:

The register file utilizes three key components:

- **demux**: A demultiplexer decodes the write_sel signal to enable the write operation for the selected register.
- **RegisterEntity**: This represents a single 32-bit register with write enable, clock, and clear inputs.
- **General_Mux**: A 32-to-1 multiplexer selects the appropriate register output for each read port based on the read_sel signals.

Internal Signals: The writeind signal holds the output of the demux, indicating which register to write to. Signals reg0d to reg31d store the output data of each individual register.

Instantiation and Port Mapping: The architecture body instantiates the demux, 32 RegisterEntity components, and two General_Mux components. Port mapping connects the component ports to the appropriate signals in the design.

3.3 ALU

3.3.1 Key Functionality:

- **Operands:** The ALU takes two 32-bit operands (data1 and data2) as input.
- **Operation Selection:** The aluop input is a 4-bit control signal that selects the operation to be performed.
- **Output:** The dataout output provides the 32-bit result of the operation.
- **Zero Flag:** The zflag output is a single-bit signal that is set to '1' if the result of the operation is zero, otherwise it is '0'.

3.3.2 Implementation Details:

- **Internal Signals:** Signals A and B are used to hold the potentially inverted versions of data1 and data2, respectively, based on the aluop control signals. The sum signal stores the result of the addition/subtraction operation. The tmp signal holds the final result after the selected operation is performed.

3.3.3 Operation Logic:

- **Inversion:** The code first checks aluop(2) and aluop(3) to determine whether to invert data2 (for subtraction) or data1 (for NOR), respectively.
- **Addition/Subtraction:** The sum signal is calculated by adding A, B, and aluop(2). aluop(2) acts as the carry-in for subtraction, effectively performing a two's complement operation on data2 when subtraction is selected.
- **Operation Selection:** A case statement uses aluop(1 downto 0) to select the appropriate operation (AND, OR, ADD/SUB, or SLT). Note that the SLT (set less than) operation is not fully implemented in this simplified version.
- **Output and Zero Flag:** The final result (tmp) is assigned to dataout, and the zflag is set accordingly.

4 Testing and Verification

The testbench simulates the behavior of the register file by applying a sequence of inputs and verifying the outputs.

4.1 Key Parts of the Code:

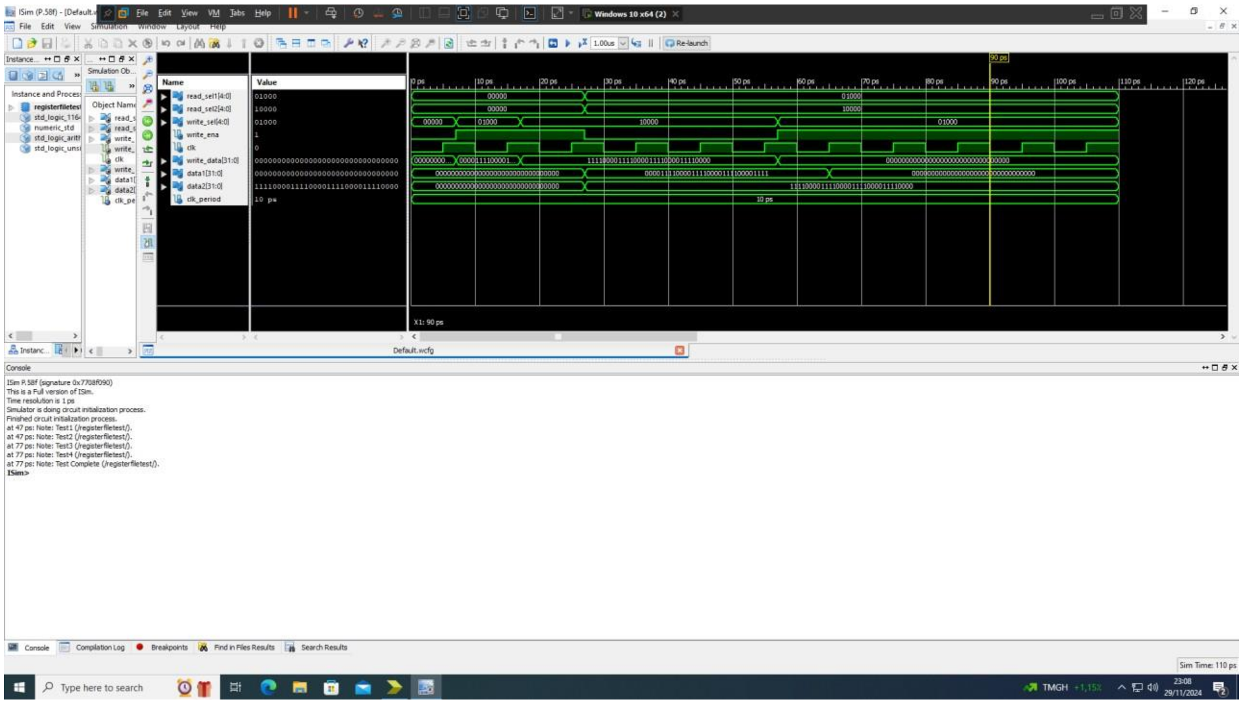
1. **Component Declaration:** The code declares a component called RegisterFile, which represents the entity under test. This declaration specifies the input and output ports of the register file.
2. **Signal Declarations:** Signals are declared to represent the inputs and outputs of the register file. These signals will be used to interact with the instantiated component.
3. **Clock Process:** A process clk_process is defined to generate a clock signal (clk) with a period of 10 ps. This clock signal will be used to drive the register file.

4. **Stimulus Process:** The stim_proc process applies a sequence of stimuli to the register file:
   - **Write Operations:** It writes specific data values to registers $t0 (register 5) and $s0 (register 16).
   - **Read Operations:** It reads data from $t0 and $s0 to verify that the written values are correctly stored.
   - **Simultaneous Read/Write:** It tests the simultaneous read and write capability by reading from $t0 and $s0 while writing a new value to $t0.
5. **Assertions:** After each read operation, assert statements are used to check if the output data (data1, data2) matches the expected values. If an assertion fails, it will generate an error message and stop the simulation.
6. **Report Statements:** report statements are used to print messages to the console, providing information about the progress of the testbench.

4.2 Expected Results:

- **Successful Simulation:** The simulation should complete without any errors. This indicates that all assertions passed, and the register file behaved as expected.
- **Console Output:** The console should display the messages "Test1", "Test2", "Test3", "Test4", and "Test Complete", indicating the successful execution of the test cases.

5 Results



```
Name                Value
read_sel1[4:0]      01000
read_sel2[4:0]      10000
write_sel[4:0]      01000
write_ena           1
clk                 0
write_data[31:0]    00000000000000000000000000000000
data1[31:0]         00000000000000000000000000000000
data2[31:0]         11110000111100001111000011110000
clk_period          10 ps
```

```
ISim P.58f (signature 0x7708f090)
This is a Full version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
at 47 ps: Note: Test1 (/registerfiletest/).
at 47 ps: Note: Test2 (/registerfiletest/).
at 77 ps: Note: Test3 (/registerfiletest/).
at 77 ps: Note: Test4 (/registerfiletest/).
at 77 ps: Note: Test Complete (/registerfiletest/).
ISim>
```

Figure 3Screenshot of the results using Isim

6 References

Elkateeb, A. (2011). *A Processor Design Course Project: Creating Soft-Core MIPS Processor Using Step-By-Step Components' Integration Approach.* International Journal of Information and Education Technology, Vol. 1, No. 5.

(n.d.). *Implementation of a MIPS processor in VHDL.*