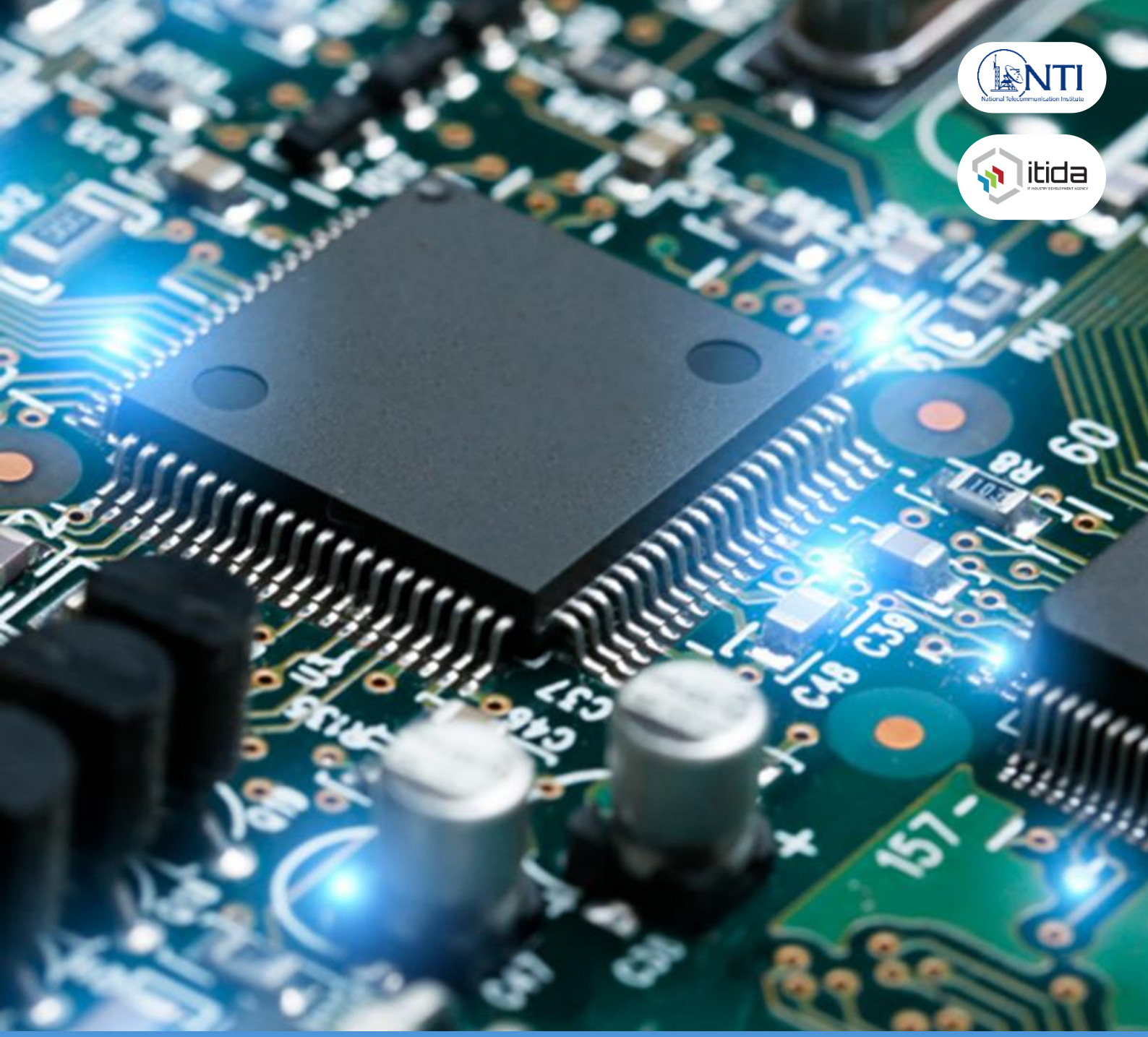


8-Bit Microprocessor

Digital Design using FPGA – Final Project

Under Supervision of : Prof. Mohamed Youssef

Mohaned Waled Hosny Ibrahim

Muhammad Sameer Abdelhamid

15th August 2025

Abstract

This report documents the complete design, implementation, and verification of a custom 8-bit, single-cycle microprocessor. The primary objective was to apply computer architecture principles to create a functional CPU model from the ground up using the VHDL hardware description language. A modular, bottom-up methodology was employed, beginning with the development of individual components such as a 16-function Arithmetic Logic Unit (ALU), an 8x8 register file, and a control unit. The final architecture is based on a custom 16-bit, register-to-register Instruction Set Architecture (ISA). A critical design challenge, the Read-After-Write (RAW) data hazard inherent in single-cycle designs, was identified and resolved through the implementation of a data forwarding path. The fully integrated microprocessor was rigorously tested in simulation, where it successfully executed a multi-line test program, with final waveform results precisely matching the expected execution trace. The project concludes with a functional and verifiable VHDL model of an 8-bit microprocessor, providing a robust foundation for future enhancements such as data memory, conditional branching, and pipelining.

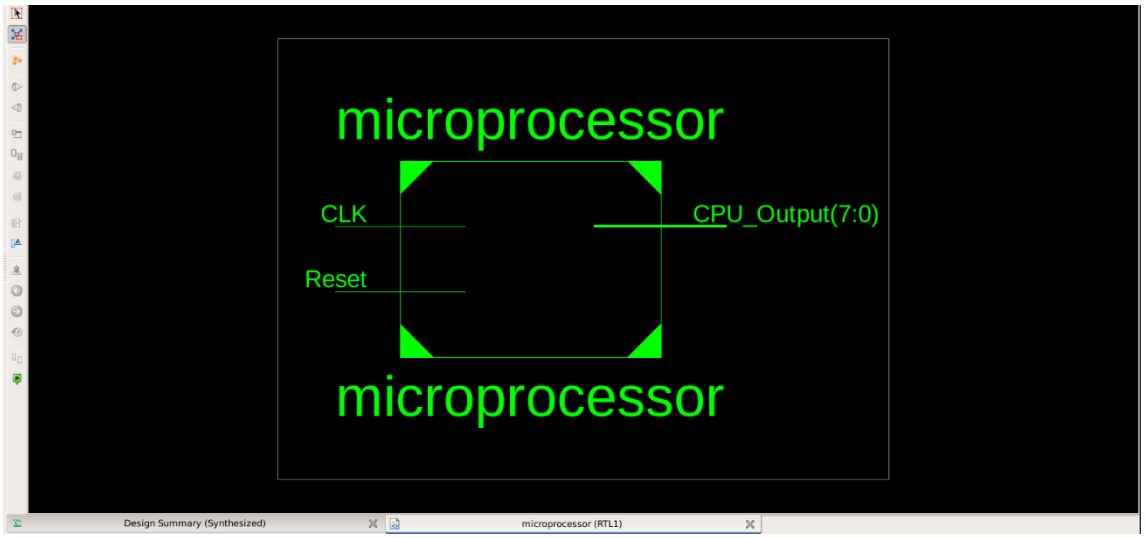


Table of Contents

Abstract 1
1.0 Introduction 3
1.1 Project Overview 3
1.2 Objectives 3
1.3 Scope and Limitations 3
1.4 Report Structure 4
2.0 Theoretical Background 5
2.1 CPU Architecture Fundamentals 5
2.2 Datapath and Control Path 5
2.3 Hardware Description Language (HDL) 6
3.0 Microprocessor Architecture and Design 6
3.1 High-Level Architecture 6
3.2 Instruction Set Architecture (ISA) 7
3.3 Opcode Mapping 7
3.4 Data Hazard and Forwarding 8
4.0 Modular Implementation (VHDL) 9
4.1 Arithmetic Logic Unit (ALU) 9
4.2 Register File 14
4.3 Program Counter (PC) 15
4.4 Instruction Memory (ROM) 17
4.5 Status Register 18
4.6 Control Unit 20
4.7 Top-Level Integration 22
5.0 Verification and Testing 25
5.1 Simulation Strategy 25
5.2 Component-Level Simulation 25
5.2.1 Arithmetic Logic Unit (ALU) 26
5.2.2 Register File 27
5.2.3 Instruction Memory (ROM) 28
5.2.4 Program Counter (PC) 29
5.2.5 Status Register 30
5.2.6 Control Unit 31
5.3 Top-Level System Simulation 32
6.0 Discussion and Challenges 33
6.1 Analysis of Results 33
6.2 Key Challenges and Solutions 33
6.3 Design Trade-offs 33
7.0 Conclusion and Future Work 35
7.1 Conclusion 35
7.2 Future Work and Enhancements 35
References 36

1.0 Introduction

1.1 Project Overview

The study of computer architecture provides the theoretical foundation for how modern processors work, yet a comprehensive understanding is best achieved through practical application. This project bridges the gap between theory and practice by documenting the design, implementation, and verification of a custom 8-bit microprocessor. Using the VHDL hardware description language, a complete central processing unit (CPU) was constructed from the ground up, following a modular, bottom-up design methodology. The final result is a functional, single-cycle microprocessor model capable of executing a defined set of arithmetic and logical instructions, serving as a tangible demonstration of core computer engineering principles.

1.2 Objectives

The primary goal of this project was divided into the following specific objectives:

- **Design a custom 8-bit Instruction Set Architecture (ISA):** Define a fixed-length, 16-bit instruction format to support 16 unique register-to-register operations, including arithmetic, logical, and shift/rotate functions.
- **Implement Core Hardware Modules in VHDL:** Develop and individually verify all essential hardware components, including a multi-function Arithmetic Logic Unit (ALU), an 8x8 register file, a program counter, status register, instruction memory, and a combinational control unit.
- **Integrate Modules into a Cohesive System:** Assemble all verified modules into a top-level, single-cycle datapath architecture, complete with the necessary control logic and internal buses to ensure correct data flow and operation.
- **Verify the Complete Microprocessor Design:** Perform a comprehensive system-level simulation to validate the microprocessor's ability to correctly fetch, decode, and execute a multi-line test program, verifying the final output against a predicted execution trace.

1.3 Scope and Limitations

To ensure the project remained focused and achievable, the following scope was defined:

- **In Scope:**
    - **Architecture:** A single-cycle processor where each instruction completes in one clock cycle.
    - **Instruction Set:** A set of 16 register-to-register instructions.
    - **Data Path:** An 8-bit data path width.
    - **Hazard Handling:** Implementation of a data forwarding path to resolve Read-After-Write (RAW) data hazards.
    - **Verification:** The design is verified exclusively through simulation.

- **Out of Scope:**
    - **Memory Interface:** The design does not include a separate data memory (RAM) or the corresponding LOAD/STORE instructions.
    - **Control Flow:** Advanced control flow instructions such as conditional branches, jumps, and subroutines (requiring a stack) are not implemented.
    - **Advanced Features:** The architecture does not include pipelining, interrupt handling, or external I/O interfaces.

1.4 Report Structure

This report is structured to logically present the design process from conception to verification. Section 2, the next section provides the theoretical background on CPU architecture and VHDL.

Section 3 details the high-level design of the microprocessor, including its custom ISA. Section 4 presents the VHDL implementation of each individual hardware module. Section 5 documents the verification strategy and presents the final simulation results. Finally, Sections 6 and 7 discuss the challenges encountered during the project and offer a conclusion on the work accomplished, along with potential directions for future enhancements.

2.0 Theoretical Background

2.1 CPU Architecture Fundamentals

A Central Processing Unit (CPU) is the primary component of a computer that executes instructions. Its design is governed by a set of architectural principles that dictate how it operates. Most processor designs are based on one of two fundamental models: **Von Neumann** or **Harvard**. The Von Neumann architecture uses a single, shared memory and a common bus for both instructions and data. In contrast, the Harvard architecture uses physically separate memories and buses for instructions and data, allowing simultaneous access to both and potentially higher performance. This project's design more closely resembles a **Harvard** architecture, as it utilizes a dedicated Instruction Memory.

Regardless of the model, all processors operate on a fundamental loop known as the **Fetch-Decode-Execute cycle**.

This three-step process is the basis for all program execution:

1. **Fetch:** The processor retrieves the next instruction from memory. The address of this instruction is supplied by a special register called the Program Counter (PC).

2. **Decode:** The fetched instruction, represented as a binary number, is interpreted by the Control Unit to determine which operation is to be performed and which data to use.

3. **Execute:** The operation is carried out by the appropriate components. This may involve a calculation in the Arithmetic Logic Unit (ALU), a data movement between registers, or a change in the program flow. Once complete, the cycle repeats.

2.2 Datapath and Control Path

A microprocessor's internal architecture can be logically divided into two distinct parts: the datapath and the control path.

- The **Datapath** can be thought of as the "muscles" of the CPU. It comprises all the hardware components that hold, process, and move data. In this project, the datapath includes the **Arithmetic Logic Unit (ALU)**, the **Register File**, the **Program Counter (PC)**, and the internal buses that connect them. Its primary function is to perform the operations specified by the instructions.

- The **Control Path** is the "brain" of the CPU. Its core is the **Control Unit**, which is responsible for directing the datapath's activities. By decoding the current instruction, the control path generates a series of command signals that tell the datapath components what to do in a given cycle—for example, which operation the ALU should perform, which registers to read from, and whether to write a result back.

2.3 Hardware Description Language (HDL)

This microprocessor was designed using VHDL (VHSIC Hardware Description Language). Unlike traditional software programming languages that describe a sequence of steps, an HDL is used to describe the structure and behavior of electronic hardware. The most critical distinction is **concurrency**; in VHDL, most operations are modeled as occurring simultaneously, just as they would in a physical circuit with millions of transistors operating in parallel.

VHDL serves three primary purposes in the digital design workflow:

1. **Design:** It provides a precise, text-based method for defining the logic of each hardware module.
2. **Simulation:** It allows for the creation of testbenches to rigorously verify the correctness of the design before it is physically created.
3. **Synthesis:** VHDL code can be compiled by a synthesis tool, which automatically translates the hardware description into a netlist of logic gates and interconnections that can be implemented on a physical device, such as a Field-Programmable Gate Array (FPGA). This project focuses on the design and simulation stages.

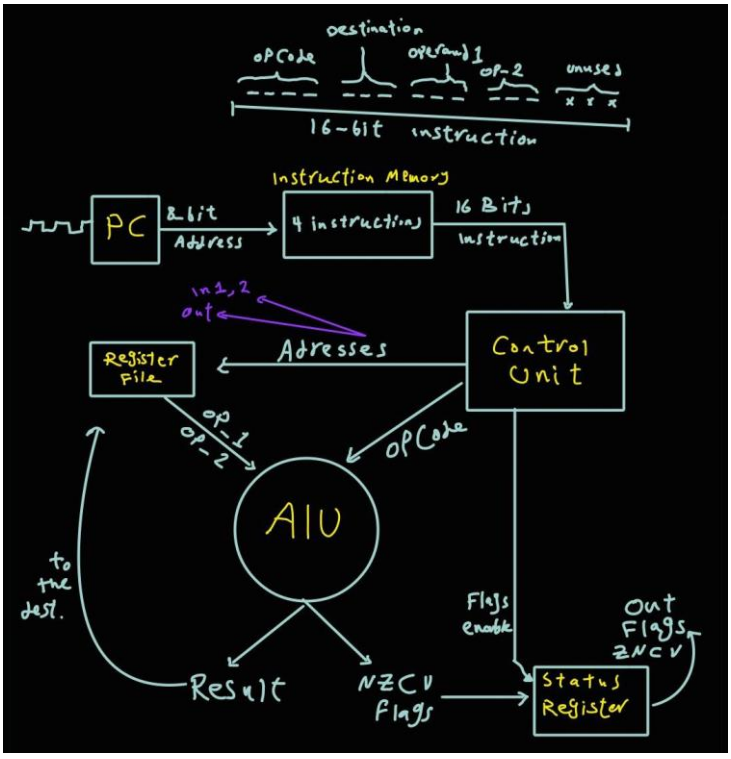
3.0 Microprocessor Architecture and Design

3.1 High-Level Architecture

The microprocessor is designed as a **single-cycle** machine, where each instruction is fully executed in one clock cycle. This approach simplifies the control logic significantly. The architecture consists of five primary, interconnected functional blocks: the Program Counter (PC), Instruction Memory, Register File, Arithmetic Logic Unit (ALU), and Control Unit.

The data flow for a single instruction is as follows:

1. The **Program Counter** provides an 8-bit address to the **Instruction Memory**.
2. The **Instruction Memory** outputs the corresponding 16-bit instruction.
3. The instruction is sent to the **Control Unit** for decoding and to the **Register File** to select source operands.
4. The **Register File** outputs the two 8-bit source operands to the **ALU**.
5. The **Control Unit** sends a 4-bit opcode to the **ALU**, which performs the specified operation.
6. The 8-bit result from the ALU is written back to the **Register File**.



3.2 Instruction Set Architecture (ISA)

The Instruction Set Architecture (ISA) defines the interface between the software (the program) and the hardware. A custom 16-bit, fixed-length ISA was developed for this project, based on a simple **Register-to-Register (R-Type)** format. This means that ALU operations are performed on data held in registers, and the result is stored in a register.

The 16-bit instruction word is partitioned into the following fields:

Bits	15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
Purpose	**Opcode**	**Dest. Reg (Rd)**	**Source Reg 1 (Rs1)**	**Source Reg 2 (Rs2)**	Unused
Size	4 bits	3 bits	3 bits	3 bits	3 bits

- **Opcode:** Specifies the operation to be performed by the ALU.
- **Rd (Destination Register):** The 3-bit address of the register where the result will be stored.
- **Rs1 (Source Register 1):** The 3-bit address of the first source operand.
- **Rs2 (Source Register 2):** The 3-bit address of the second source operand.

3.3 Opcode Mapping

The 4-bit opcode field allows for 24=16 unique instructions. The mapping from the binary opcode to the instruction mnemonic is defined below.

Opcode	Mnemonic	Description
0000	ADD	Add Rs1 and Rs2, store in Rd
0001	SUB	Subtract Rs2 from Rs1, store in Rd
0010	MUL	Multiply Rs1 and Rs2, store in Rd
0011	GT	If Rs1 > Rs2, store 1 in Rd, else 0
0100	LT	If Rs1 < Rs2, store 1 in Rd, else 0
0101	EQ	If Rs1 = Rs2, store 1 in Rd, else 0
0110	AND	Bitwise AND Rs1 and Rs2, store in Rd
0111	OR	Bitwise OR Rs1 and Rs2, store in Rd
1000	XOR	Bitwise XOR Rs1 and Rs2, store in Rd
1001	NOT	Bitwise NOT Rs1, store in Rd
1010	INC	Increment Rs1, store in Rd

1011	DEC	Decrement Rs1, store in Rd
1100	LSL	Logical Shift Left Rs1, store in Rd
1101	LSR	Logical Shift Right Rs1, store in Rd
1110	ROL	Rotate Left Rs1, store in Rd
1111	ROR	Rotate Right Rs1, store in Rd

3.4 Data Hazard and Forwarding

A critical challenge in any processor design is the **data hazard**, specifically the Read-After-Write (RAW) hazard. This occurs when an instruction attempts to read a register's value before a preceding instruction has finished writing its new value back to that register.

For example, in the sequence:

1. ADD R3, R1, R2
2. SUB R4, R3, R1

The SUB instruction needs the new value of R3 calculated by ADD. In our clocked design, the ADD result is only written to R3 on the rising clock edge that begins the *next* cycle. The SUB instruction, however, needs to *read* from R3 at the start of that same cycle. This timing conflict would cause the SUB to read the old, stale value of R3, leading to an incorrect result.

The architectural solution implemented to solve this is **Data Forwarding** (also known as bypassing). Instead of waiting for the data to be written to and then read back from the Register File, a bypass path is created. This path forwards the ALU's result from the end of one instruction's execution directly back to the ALU's input for the very next instruction, ensuring the most current data is always used. This was implemented in the top-level VHDL using combinational logic that functions as a multiplexer ahead of the ALU's inputs.

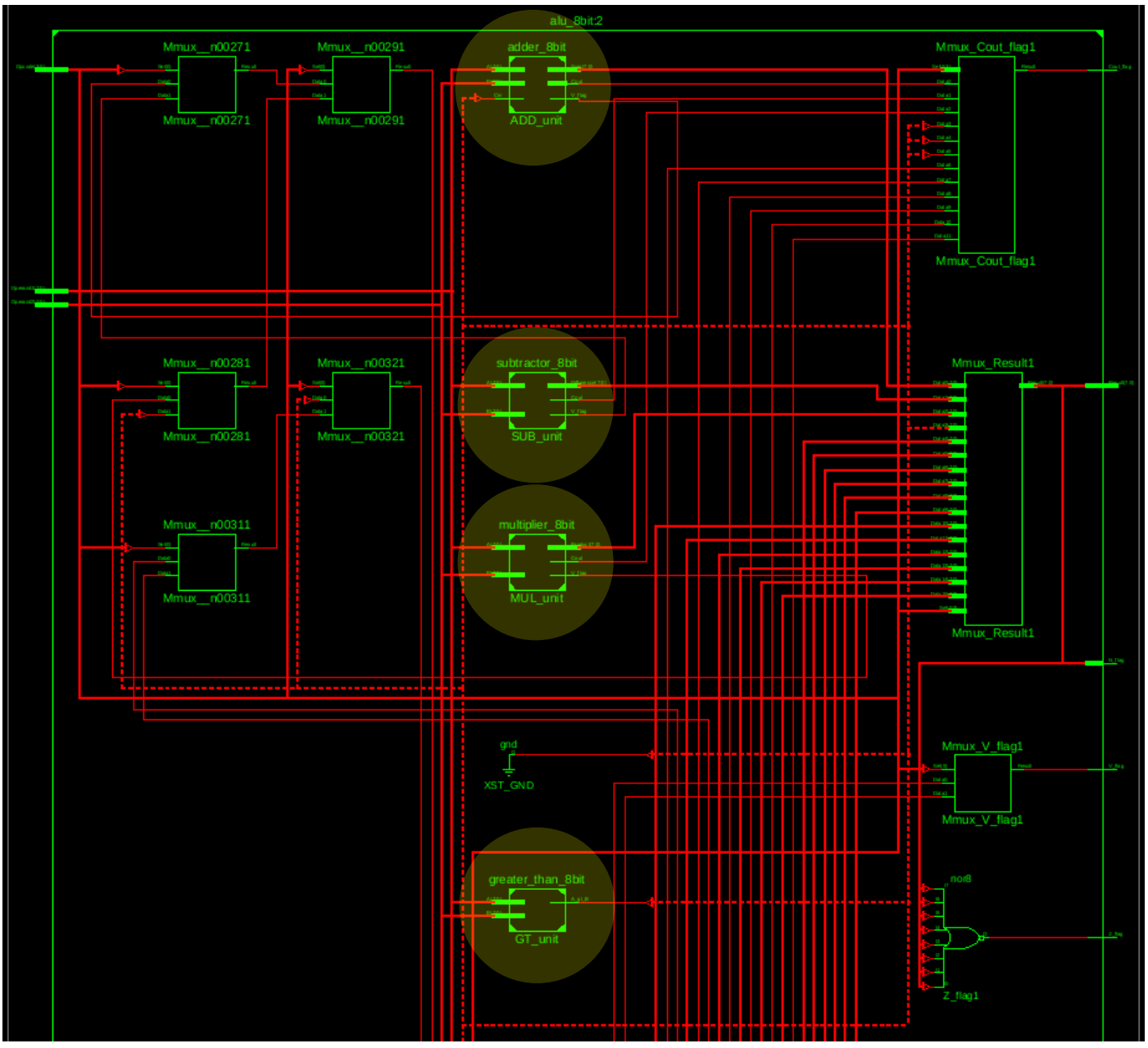
Of course. Here is the complete Section 4.0, with all the subsections combined and placeholders added for your VHDL code snapshots.

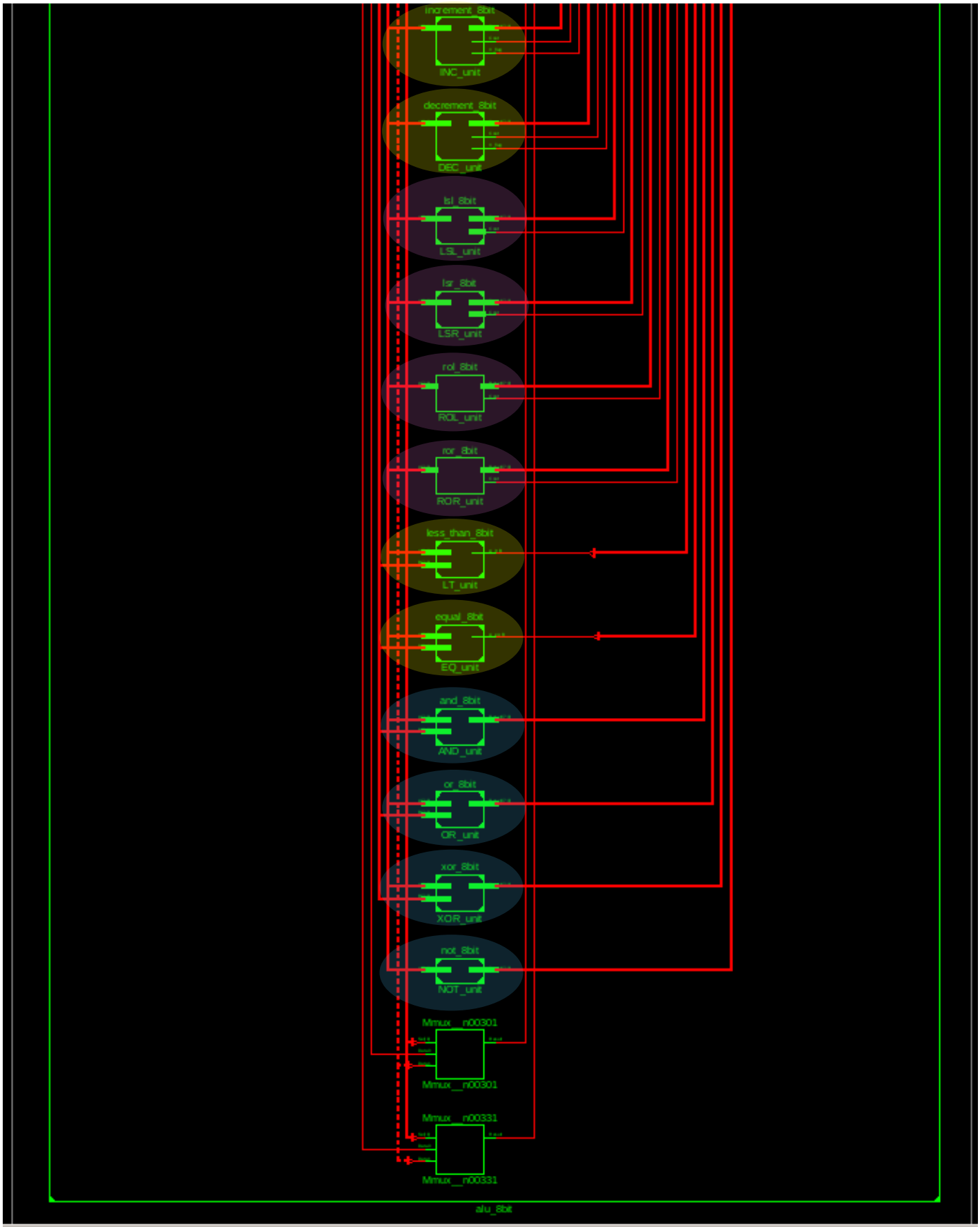
4.0 Modular Implementation (VHDL)

This section details the VHDL implementation of each major hardware module of the microprocessor. A consistent bottom-up design and verification strategy was employed, ensuring each component was functional before its integration into the top-level design.

4.1 Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) is the computational core of the microprocessor. It is a **combinational circuit** designed to perform the 16 distinct arithmetic and logical operations defined by the Instruction Set Architecture. The specific operation to be executed is determined by the **4-bit ALU Opcode** signal from the Control Unit. The ALU takes two 8-bit operands as input and produces an 8-bit result, along with four status flags (Negative, Zero, Carry, Overflow) that describe the outcome of the operation.

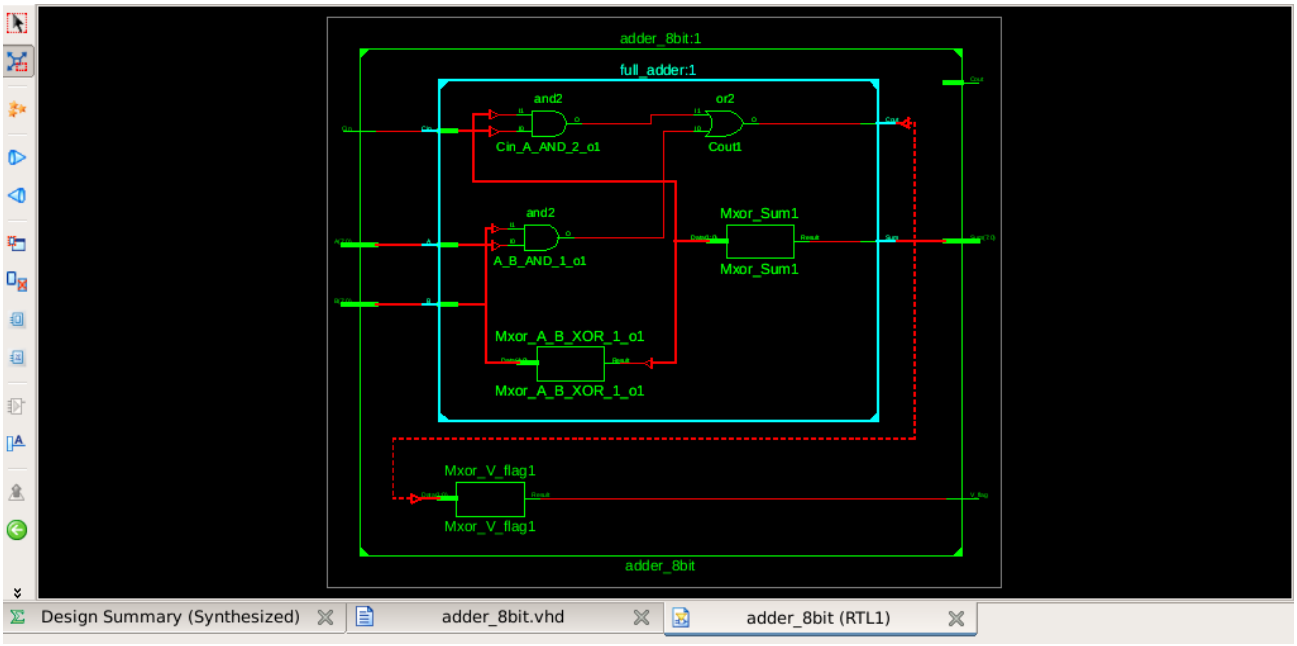



increment_8bit
INC_unit
decrement_8bit
DEC_unit
lsl_8bit
LSL_unit
lsr_8bit
LSR_unit
rol_8bit
ROL_unit
ror_8bit
ROR_unit
less_than_8bit
LT_unit
equal_8bit
EQ_unit
and_8bit
AND_unit
or_8bit
OR_unit
xor_8bit
XOR_unit
not_8bit
NOT_unit
Mmux__n00301
Mmux__n00301
Mmux__n00331
Mmux__n00331
alu_8bit

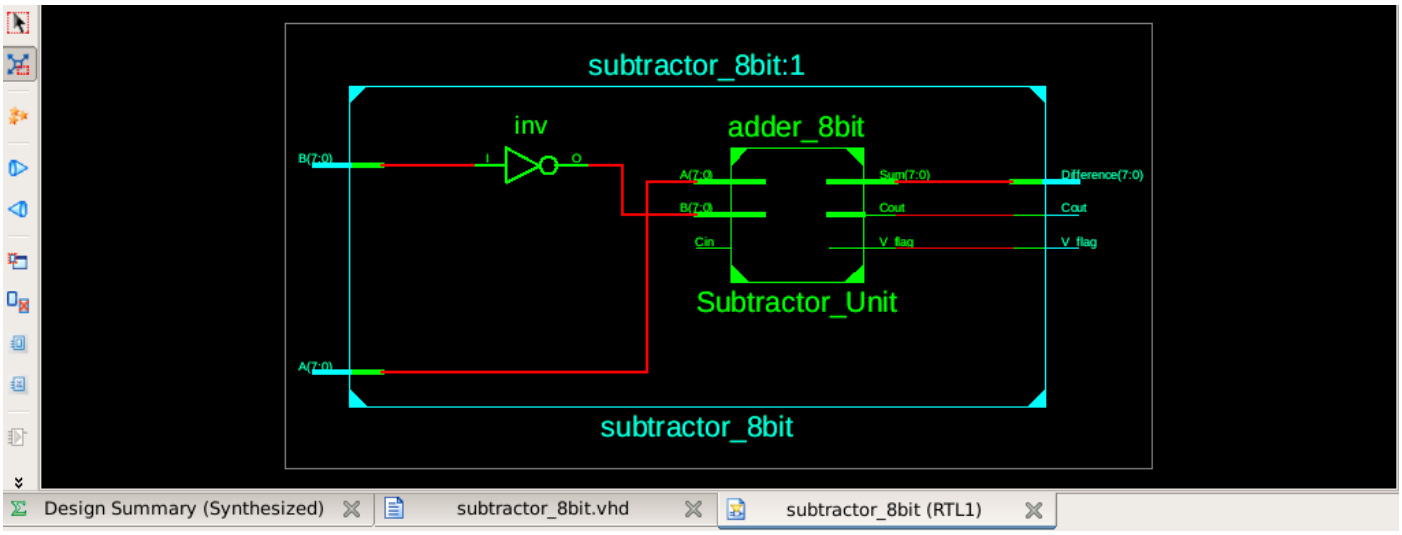
To manage complexity and facilitate testing, the ALU was not designed as a single monolithic block. Instead, it was constructed using a **modular design** approach, where individual functional units were developed and verified independently before being integrated into the final ALU entity.

The key sub-components of the ALU are described below:

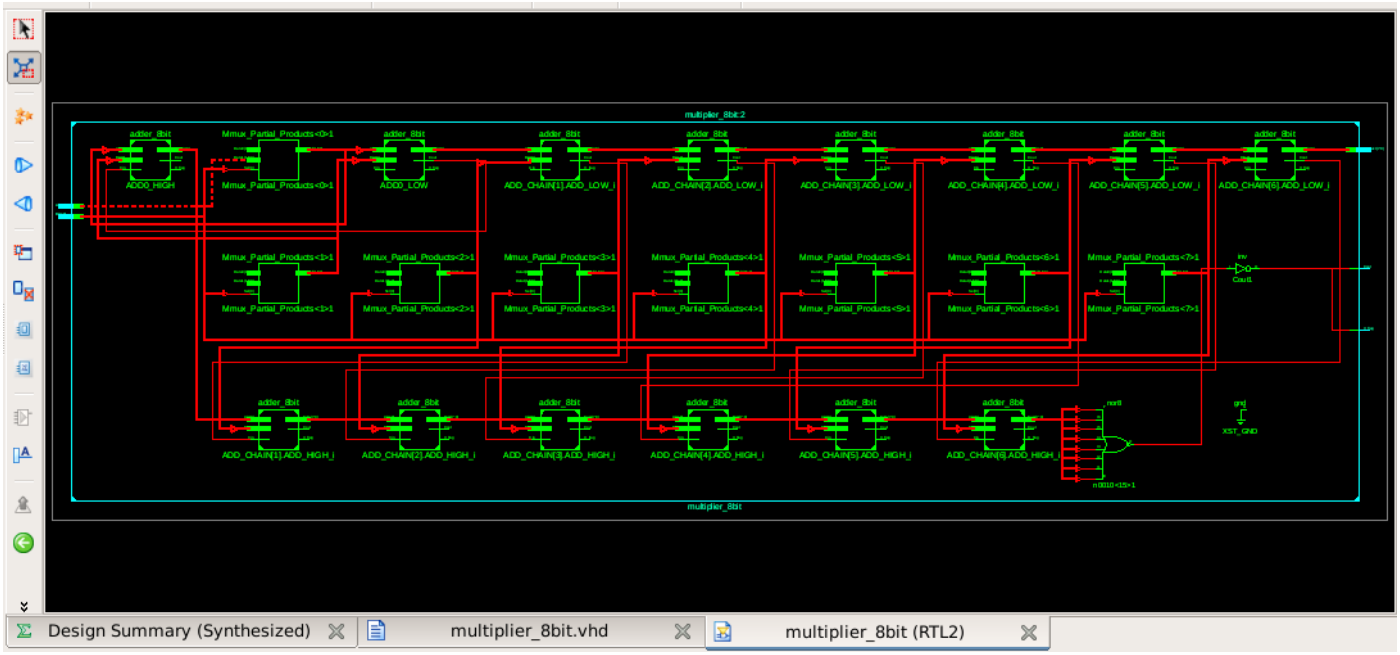
- **Adder (adder_8bit):** This unit performs 8-bit binary addition. It was implemented structurally as a **ripple-carry adder**, composed of eight instances of a fundamental full_adder component. This block is responsible for generating the primary Carry (Cout) and Overflow (V_flag) flags for addition-based operations.



- **Subtractor (subtractor_8bit):** To promote hardware reuse, a dedicated subtractor was not created. Instead, subtraction is performed using the adder via the **two's complement** method. The operation A - B is implemented as A + not(B) + 1, which involves inverting the second operand and setting the adder's carry-in bit to '1'.

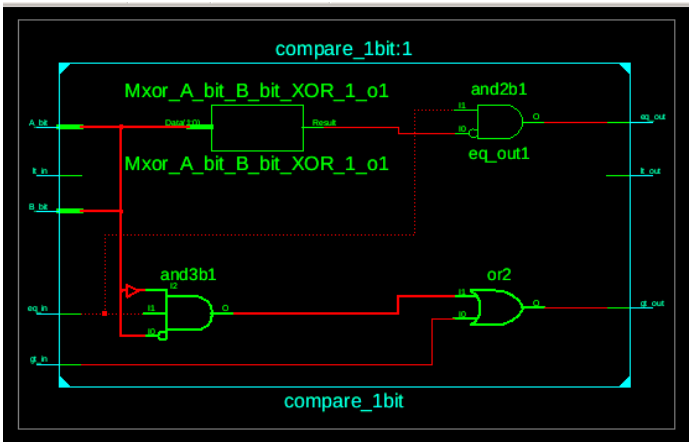


- **Multiplier (multiplier_8bit):** This is the most complex arithmetic unit. It was implemented structurally using a chain of adders to sum an array of shifted partial products. The full 16-bit mathematical result is truncated to the lower 8 bits for the final output, with the upper 8 bits used to set the Carry and Overflow flags if the result exceeds 255.



- **Comparators (greater_than_8bit, less_than_8bit, equal_8bit):**

  These units were built structurally using a cascading design. A single, reusable compare_1bit slice was designed and then instantiated eight times to create a full 8-bit comparator for each condition.



- **Logical and Shift Units:** Simpler operations such as AND, OR, XOR, NOT, and all shift/rotate functions were implemented as simple behavioral VHDL entities, as their logic does not require complex structural decomposition.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity alu_8bit is
    port (
        Operand1  : in  std_logic_vector(7 downto 0);
        Operand2  : in  std_logic_vector(7 downto 0);
        Opcode    : in  std_logic_vector(3 downto 0);
        Result    : out std_logic_vector(7 downto 0);
        Cout_flag : out std_logic;
        V_flag    : out std_logic;
        Z_flag    : out std_logic;
        N_flag    : out std_logic
    );
end entity alu_8bit;

architecture Structural of alu_8bit is

    -- Component Declarations
    component adder_8bit is port (A:in std_logic_vector(7 downto 0); B:in std_logic_vector(7 downto 0); Cin:in std_logic; Sum:out std_logic_vector(7 downto 0); Cout:out std_logic; V_flag:out std_logic); end component;
    component subtractor_8bit is port (A:in std_logic_vector(7 downto 0); B:in std_logic_vector(7 downto 0); Difference:out std_logic_vector(7 downto 0); Cout:out std_logic; V_flag:out std_logic); end component;
    component multiplier_8bit is port (A:in std_logic_vector(7 downto 0); B:in std_logic_vector(7 downto 0); Product:out std_logic_vector(7 downto 0); Cout:out std_logic; V_flag:out std_logic); end component;
    component greater_than_8bit is port (A:in std_logic_vector(7 downto 0); B:in std_logic_vector(7 downto 0); A_gt_B:out std_logic); end component;
    component less_than_8bit is port (A:in std_logic_vector(7 downto 0); B:in std_logic_vector(7 downto 0); A_lt_B:out std_logic); end component;
    component equal_8bit is port (A:in std_logic_vector(7 downto 0); B:in std_logic_vector(7 downto 0); A_eq_B:out std_logic); end component;
    component and_8bit is port (A:in std_logic_vector(7 downto 0); B:in std_logic_vector(7 downto 0); Result:out std_logic_vector(7 downto 0)); end component;
    component or_8bit is port (A:in std_logic_vector(7 downto 0); B:in std_logic_vector(7 downto 0); Result:out std_logic_vector(7 downto 0)); end component;
    component xor_8bit is port (A:in std_logic_vector(7 downto 0); B:in std_logic_vector(7 downto 0); Result:out std_logic_vector(7 downto 0)); end component;
    component not_8bit is port (A:in std_logic_vector(7 downto 0); Result:out std_logic_vector(7 downto 0)); end component;
    component increment_8bit is port (A:in std_logic_vector(7 downto 0); Result:out std_logic_vector(7 downto 0); Cout:out std_logic; V_flag:out std_logic); end component;
    component decrement_8bit is port (A:in std_logic_vector(7 downto 0); Result:out std_logic_vector(7 downto 0); Cout:out std_logic; V_flag:out std_logic); end component;
    component lsl_8bit is port (A:in std_logic_vector(7 downto 0); Result:out std_logic_vector(7 downto 0); Cout:out std_logic); end component;
    component lsr_8bit is port (A:in std_logic_vector(7 downto 0); Result:out std_logic_vector(7 downto 0); Cout:out std_logic); end component;
    component rol_8bit is port (A:in std_logic_vector(7 downto 0); Result:out std_logic_vector(7 downto 0); Cout:out std_logic); end component;
    component ror_8bit is port (A:in std_logic_vector(7 downto 0); Result:out std_logic_vector(7 downto 0); Cout:out std_logic); end component;

    -- Internal signals to hold the outputs of each operation block
    signal s_add_res, s_sub_res, s_mul_res, s_and_res, s_or_res, s_xor_res, s_not_res : std_logic_vector(7 downto 0);
    signal s_inc_res, s_dec_res, s_lsl_res, s_lsr_res, s_rol_res, s_ror_res : std_logic_vector(7 downto 0);
    signal s_add_c, s_add_v, s_sub_c, s_sub_v, s_mul_c, s_mul_v : std_logic;
    signal s_gt_res, s_lt_res, s_eq_res : std_logic;
    signal s_inc_c, s_inc_v, s_dec_c, s_dec_v : std_logic;
    signal s_lsl_c, s_lsr_c, s_rol_c, s_ror_c : std_logic;

    -- Internal signal to hold the multiplexer result before assigning to the output port
    signal s_result_internal : std_logic_vector(7 downto 0);

begin

    -- Instantiate all 16 operation blocks
    ADD_unit: adder_8bit port map(A=>Operand1, B=>Operand2, Cin=>'0', Sum=>s_add_res, Cout=>s_add_c, V_flag=>s_add_v);
    SUB_unit: subtractor_8bit port map(A=>Operand1, B=>Operand2, Difference=>s_sub_res, Cout=>s_sub_c, V_flag=>s_sub_v);
    MUL_unit: multiplier_8bit port map(A=>Operand1, B=>Operand2, Product=>s_mul_res, Cout=>s_mul_c, V_flag=>s_mul_v);
    GT_unit: greater_than_8bit port map(A=>Operand1, B=>Operand2, A_gt_B=>s_gt_res);
    LT_unit: less_than_8bit port map(A=>Operand1, B=>Operand2, A_lt_B=>s_lt_res);
    EQ_unit: equal_8bit port map(A=>Operand1, B=>Operand2, A_eq_B=>s_eq_res);
    AND_unit: and_8bit port map(A=>Operand1, B=>Operand2, Result=>s_and_res);
    OR_unit: or_8bit port map(A=>Operand1, B=>Operand2, Result=>s_or_res);
    XOR_unit: xor_8bit port map(A=>Operand1, B=>Operand2, Result=>s_xor_res);
    NOT_unit: not_8bit port map(A=>Operand1, Result=>s_not_res);
    INC_unit: increment_8bit port map(A=>Operand1, Result=>s_inc_res, Cout=>s_inc_c, V_flag=>s_inc_v);
    DEC_unit: decrement_8bit port map(A=>Operand1, Result=>s_dec_res, Cout=>s_dec_c, V_flag=>s_dec_v);
    LSL_unit: lsl_8bit port map(A=>Operand1, Result=>s_lsl_res, Cout=>s_lsl_c);
    LSR_unit: lsr_8bit port map(A=>Operand1, Result=>s_lsr_res, Cout=>s_lsr_c);
    ROL_unit: rol_8bit port map(A=>Operand1, Result=>s_rol_res, Cout=>s_rol_c);
    ROR_unit: ror_8bit port map(A=>Operand1, Result=>s_ror_res, Cout=>s_ror_c);

    -- Main multiplexer to select the final result and flags based on the Opcode
    with Opcode select
        s_result_internal <= s_add_res when "0000", -- ADD
                             s_sub_res when "0001", -- SUB
                             s_mul_res when "0010", -- MUL
                             ("0000000" & s_gt_res) when "0011", -- GT
                             ("0000000" & s_lt_res) when "0100", -- LT
                             ("0000000" & s_eq_res) when "0101", -- EQ
                             s_and_res when "0110", -- AND
                             s_or_res  when "0111", -- OR
                             s_xor_res when "1000", -- XOR
                             s_not_res when "1001", -- NOT
                             s_inc_res when "1010", -- INC A
                             s_dec_res when "1011", -- DEC A
                             s_lsl_res when "1100", -- LSL A
                             s_lsr_res when "1101", -- LSR A
                             s_rol_res when "1110", -- ROL A
                             s_ror_res when "1111", -- ROR A
                             (others => 'X') when others;

    with Opcode select
        Cout_flag <= s_add_c when "0000",
                     s_sub_c when "0001",
                     s_mul_c when "0010",
                     '0'     when "0011",
                     '0'     when "0100",
                     '0'     when "0101",
                     '0'     when "0110",
                     '0'     when "0111",
                     '0'     when "1000",
                     '0'     when "1001",
                     s_inc_c when "1010",
                     s_dec_c when "1011",
                     s_lsl_c when "1100",
                     s_lsr_c when "1101",
                     s_rol_c when "1110",
                     s_ror_c when "1111",
                     'X'     when others;

    with Opcode select
        V_flag    <= s_add_v when "0000",
                     s_sub_v when "0001",
                     s_mul_v when "0010",
                     '0'     when "0011",
                     '0'     when "0100",
                     '0'     when "0101",
                     '0'     when "0110",
                     '0'     when "0111",
                     '0'     when "1000",
                     '0'     when "1001",
                     s_inc_v when "1010",
                     s_dec_v when "1011",
                     '0'     when "1100",
                     '0'     when "1101",
                     '0'     when "1110",
                     '0'     when "1111",
                     'X'     when others;

    -- Assign internal signal to final output port
    Result <= s_result_internal;

    -- Generate Zero and Negative flags based on the internal result signal
    Z_flag <= '1' when s_result_internal = x"00" else '0';
    N_flag <= s_result_internal(7);

end architecture Structural;
```

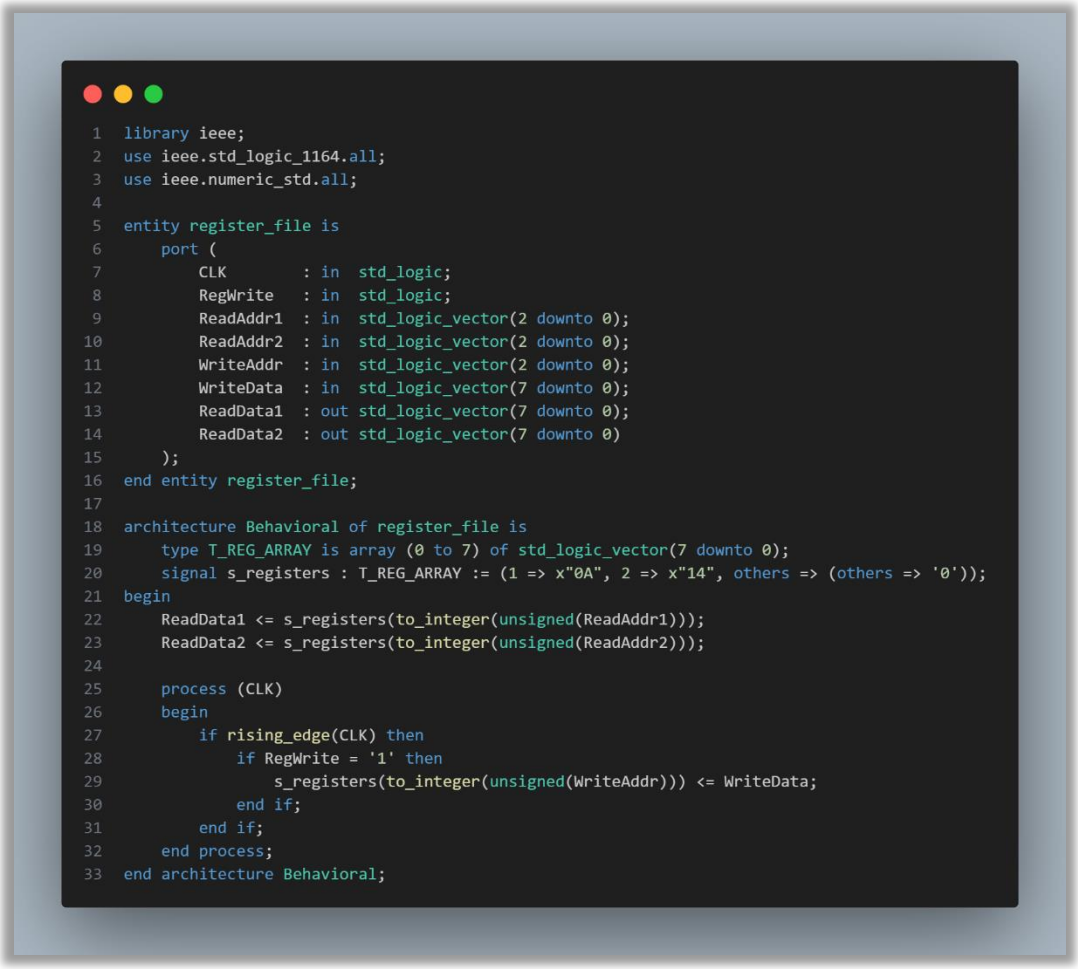
4.2 Register File

The Register File is the primary high-speed storage unit within the microprocessor's datapath. It serves as a small, on-chip scratchpad memory used to hold the operands required by the ALU and to store the results of computations. The designed file consists of **8 registers**, with each register being **8 bits** wide.

To facilitate the single-cycle execution of instructions, the Register File was designed with a **dual-port read, single-port write** architecture. This configuration is critical for performance and consists of:

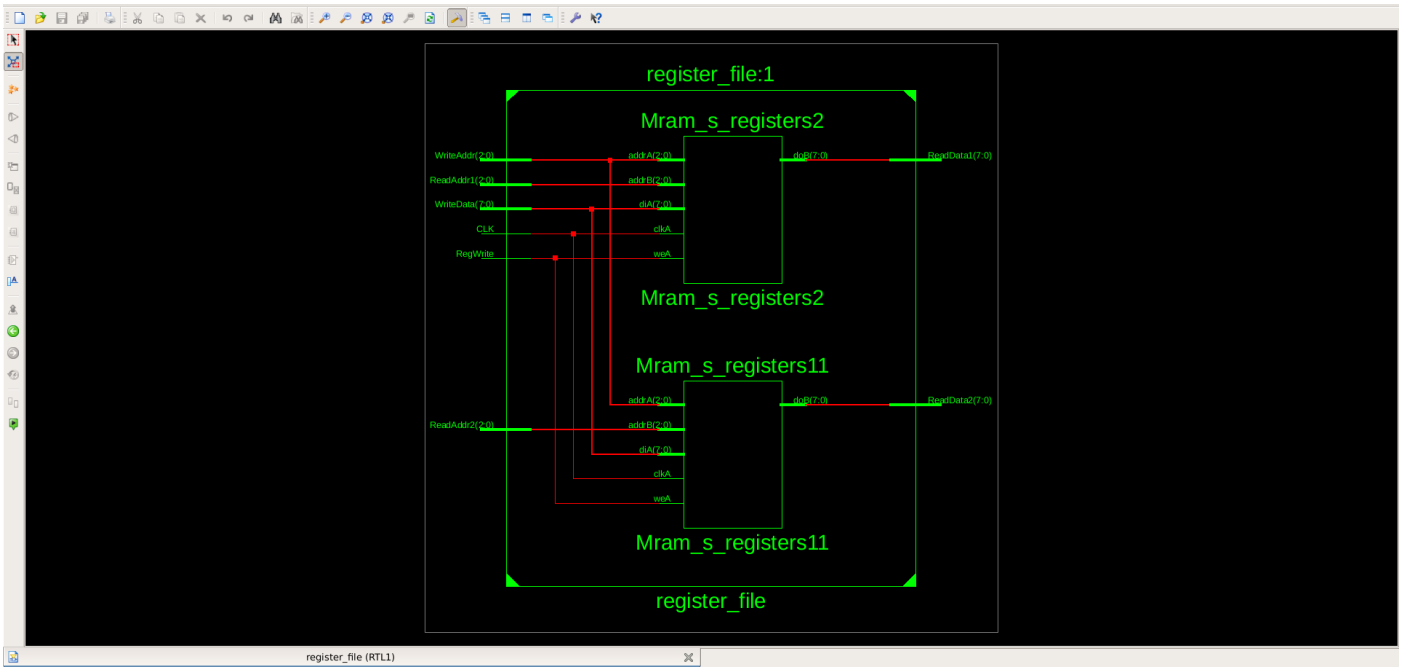
- Two independent **asynchronous read ports** (ReadData1, ReadData2). These ports allow the contents of any two registers, specified by ReadAddr1 and ReadAddr2, to be read simultaneously and combinationally. This enables the CPU to fetch both source operands for the ALU in the same clock cycle.

- One **synchronous write port**. A result from the ALU is written into the register specified by WriteAddr only on the rising edge of the CLK and only if the RegWrite control signal from the Control Unit is asserted. This synchronous behavior ensures that the state of the registers only changes at predictable intervals, which is essential for stable processor operation.



```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity register_file is
    port (
        CLK        : in  std_logic;
        RegWrite   : in  std_logic;
        ReadAddr1  : in  std_logic_vector(2 downto 0);
        ReadAddr2  : in  std_logic_vector(2 downto 0);
        WriteAddr  : in  std_logic_vector(2 downto 0);
        WriteData  : in  std_logic_vector(7 downto 0);
        ReadData1  : out std_logic_vector(7 downto 0);
        ReadData2  : out std_logic_vector(7 downto 0)
    );
end entity register_file;

architecture Behavioral of register_file is
    type T_REG_ARRAY is array (0 to 7) of std_logic_vector(7 downto 0);
    signal s_registers : T_REG_ARRAY := (1 => x"0A", 2 => x"14", others => (others => '0'));
begin
    ReadData1 <= s_registers(to_integer(unsigned(ReadAddr1)));
    ReadData2 <= s_registers(to_integer(unsigned(ReadAddr2)));

    process (CLK)
    begin
        if rising_edge(CLK) then
            if RegWrite = '1' then
                s_registers(to_integer(unsigned(WriteAddr))) <= WriteData;
            end if;
        end if;
    end process;
end architecture Behavioral;
```

In the VHDL implementation, the physical registers were modeled using an array of std_logic_vector. The asynchronous read logic was implemented using concurrent signal assignments, while the synchronous write logic was implemented within a clocked process sensitive to the rising edge of the CLK and the RegWrite enable signal.



4.3 Program Counter (PC)

The Program Counter (PC) is the component responsible for controlling the flow of program execution. It is a dedicated 8-bit register whose sole function is to hold and output the memory address of the next instruction to be fetched from the Instruction ROM. Its 8-bit width allows it to address a memory space of up to 256 instructions.

The PC's operation is entirely **synchronous**, with its value updating on the rising edge of the CLK based on control signals from the Control Unit. The VHDL implementation is centered around a clocked process that prioritizes control signals in the following order:

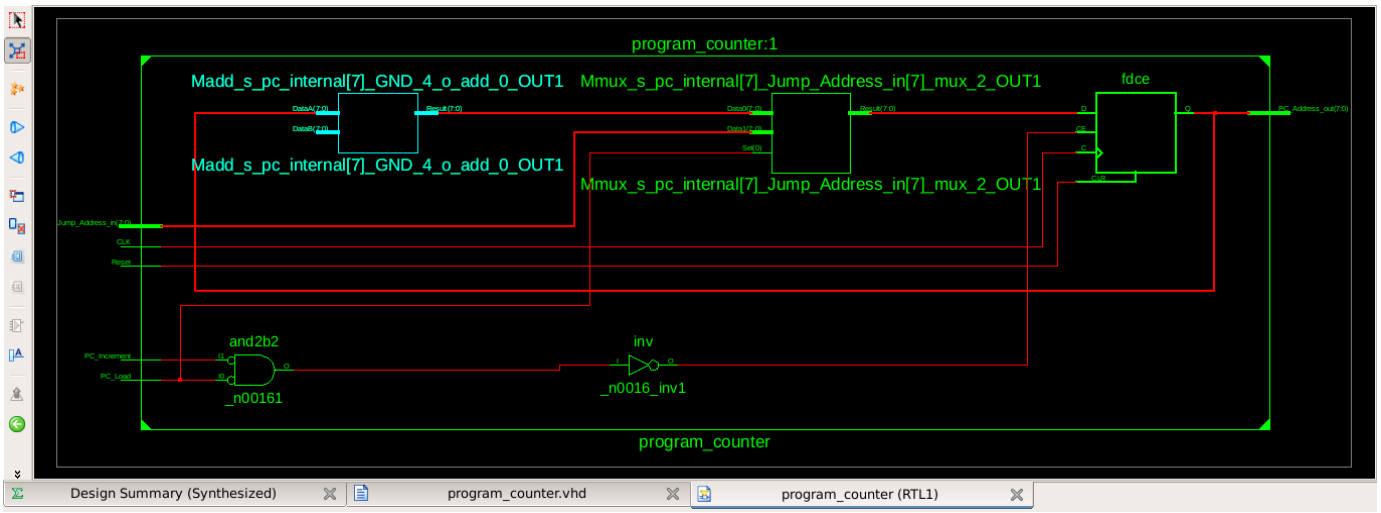
1. **Asynchronous Reset:** A high-priority Reset signal immediately and asynchronously forces the PC's value to x"00". This ensures that the microprocessor always begins execution from the first instruction in memory upon startup.

2. **Synchronous Load:** When the PC_Load signal is asserted, the PC is overwritten with an address supplied on the Jump_Address_in port. While not fully utilized in the current design, this functionality is the essential hardware support for future jump and branch instructions.

3. **Synchronous Increment:** If PC_Load is not active but PC_Increment is, the PC's value is incremented by one. This is the default behavior for sequential instruction execution.

If none of these control signals are active, the PC holds its current value. The logic was implemented using an internal unsigned signal to facilitate the addition operation, providing a robust and predictable mechanism for program sequencing.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity program_counter is
    port (
        CLK             : in  std_logic;
        Reset           : in  std_logic; -- Asynchronous reset to 0x00
        PC_Load         : in  std_logic; -- Enable to load a new address
        PC_Increment    : in  std_logic; -- Enable to increment the PC
        Jump_Address_in : in  std_logic_vector(7 downto 0);
        PC_Address_out  : out std_logic_vector(7 downto 0)
    );
end entity program_counter;

architecture Behavioral of program_counter is
    signal s_pc_internal : unsigned(7 downto 0) := (others => '0');
begin

    process (CLK, Reset)
    begin
        if Reset = '1' then
            s_pc_internal <= (others => '0');
        elsif rising_edge(CLK) then
            if PC_Load = '1' then
                s_pc_internal <= unsigned(Jump_Address_in);
            elsif PC_Increment = '1' then
                s_pc_internal <= s_pc_internal + 1;
            end if;
            -- If neither Load nor Increment is '1', the PC holds its value
        end if;
    end process;

    -- Continuously output the internal PC value
    PC_Address_out <= std_logic_vector(s_pc_internal);

end architecture Behavioral;
```



4.4 Instruction Memory (ROM)

The Instruction Memory is the component that stores the program to be executed by the microprocessor. For this project, it was designed as a **Read-Only Memory (ROM)** with a capacity for 256 instructions, where each instruction is 16 bits wide. This configuration corresponds to the 8-bit address bus from the Program Counter and the 16-bit instruction width defined by the ISA.

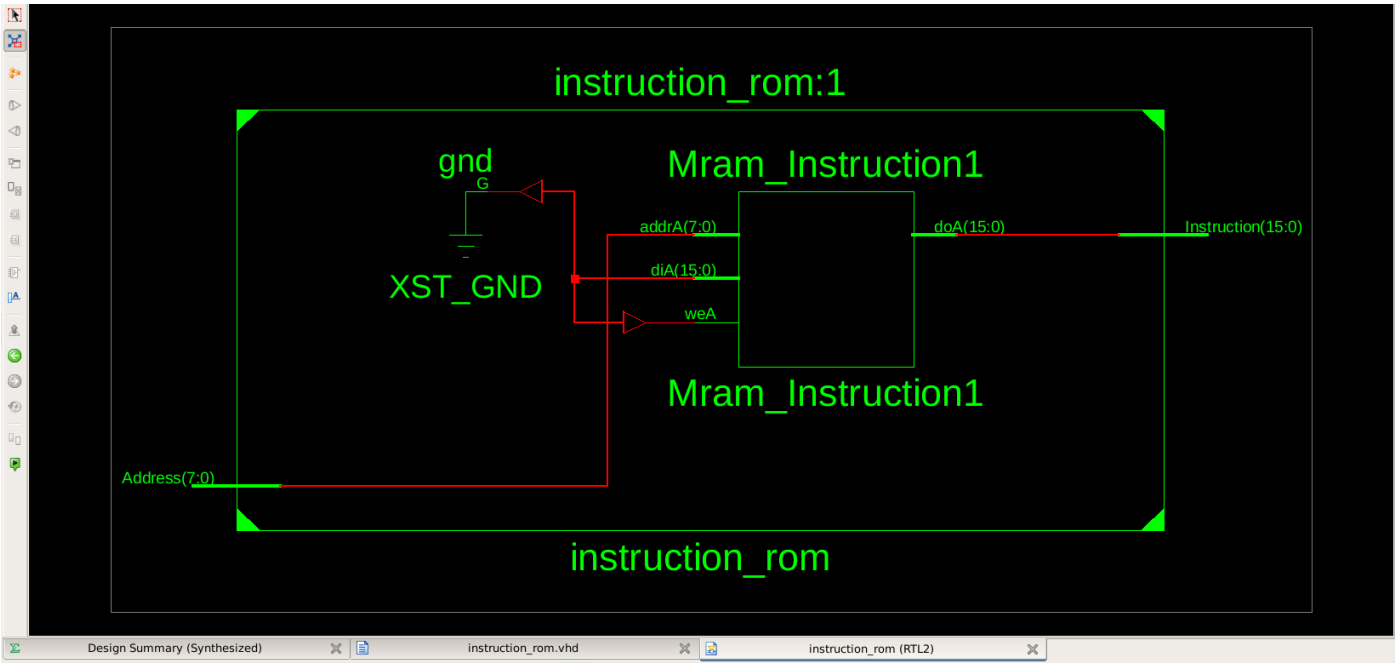
As a ROM, its contents are fixed at design time and cannot be modified by the microprocessor during execution. Architecturally, the Instruction Memory is a purely **combinational** block; it takes an 8-bit address as an input and, after a short propagation delay, presents the corresponding 16-bit instruction on its output.

The VHDL implementation uses a **constant array** of std_logic_vector to model the memory. This is a standard, synthesizable approach for creating a ROM. The read operation is a simple, concurrent array-indexing statement. For verification, a 4-line test program was encoded according to the ISA and stored in the memory, as detailed in the memory map below.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity instruction_rom is
    port (
        Address     : in  std_logic_vector(7 downto 0);
        Instruction : out std_logic_vector(15 downto 0)
    );
end entity instruction_rom;

architecture Behavioral of instruction_rom is

    type T_ROM_ARRAY is array (0 to 255) of std_logic_vector(15 downto 0);

    constant C_INSTRUCTION_ROM : T_ROM_ARRAY := (
        -- 0: ADD R3, R1, R2   (Opcode=0000, Rd=3, Rs1=1, Rs2=2)
        0 => x"0650",
        -- 1: SUB R4, R3, R1   (Opcode=0001, Rd=4, Rs1=3, Rs2=1)
        1 => x"18C8",
        -- 2: AND R5, R3, R4   (Opcode=0110, Rd=5, Rs1=3, Rs2=4)
        2 => x"6AE0",
        -- 3: NOT R6, R5       (Opcode=1001, Rd=6, Rs1=5, Rs2=ignored)
        3 => x"9D40",

        -- Fill the rest of the memory with NOPs
        others => x"0000"
    );

begin
    Instruction <= C_INSTRUCTION_ROM(to_integer(unsigned(Address)));
end architecture Behavioral;
```

Program Memory Map

Address (Hex)	Machine Code (Hex)	Instruction (Assembly)
0x00	0650	ADD R3, R1, R2
0x01	18C8	SUB R4, R3, R1
0x02	6AE0	AND R5, R3, R4
0x03	9D40	NOT R6, R5
0x04..0xFF	0000	NOP



4.5 Status Register

The **Status Register**, often called the flags register, is a 4-bit register that captures the outcome of arithmetic and logical operations performed by the ALU. Its primary purpose is to store this state information so that it can be used by future instructions, which is the fundamental mechanism for enabling **conditional branching** and intelligent program flow.

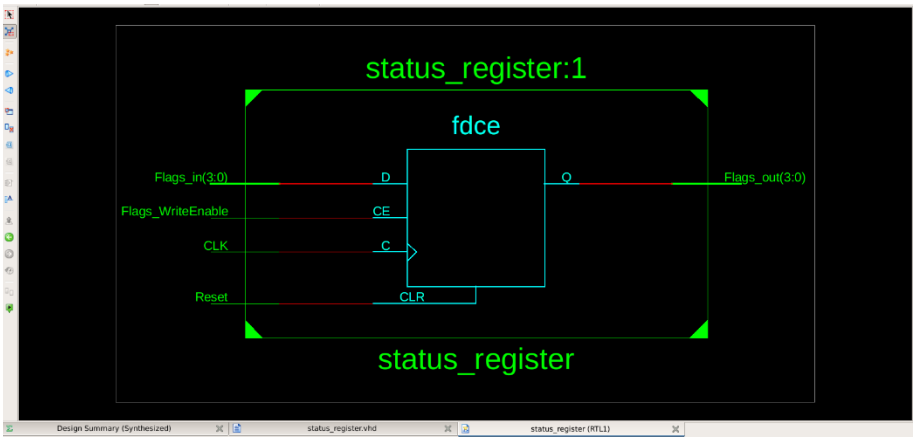
The register stores the four primary status flags generated by the ALU, with the following bit assignments:

- **Bit 3: N** (Negative flag)
- **Bit 2: Z** (Zero flag)
- **Bit 1: C** (Carry flag)
- **Bit 0: V** (Overflow flag)

The operation of the Status Register is fully **synchronous**. It updates its value only on the rising edge of the CLK and only when explicitly enabled by the Flags_WriteEnable signal from the Control Unit. This selective writing is critical, as it ensures that only instructions intended to affect the flags (like ADD or LSL) will modify them, while other instructions (like the comparators) will leave them unchanged. The register also includes an **asynchronous reset** to clear all flags to a known '0' state upon system startup.

The VHDL implementation consists of a clocked process that latches the 4-bit Flags_in vector into an internal signal when Flags_WriteEnable is active. This internal signal is then concurrently assigned to the Flags_out port.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity status_register is
    port (
        CLK              : in  std_logic;
        Reset            : in  std_logic; -- Asynchronous reset
        Flags_WriteEnable : in  std_logic; -- Enable signal from Control Unit
        Flags_in         : in  std_logic_vector(3 downto 0); -- (N,Z,C,V) from ALU
        Flags_out        : out std_logic_vector(3 downto 0)  -- (N,Z,C,V) to Control Unit
    );
end entity status_register;

architecture Behavioral of status_register is
    signal s_flags_internal : std_logic_vector(3 downto 0) := (others => '0');
begin

    process(CLK, Reset)
    begin
        if Reset = '1' then
            s_flags_internal <= (others => '0');
        elsif rising_edge(CLK) then
            if Flags_WriteEnable = '1' then
                s_flags_internal <= Flags_in;
            end if;
            -- If WriteEnable is '0', the register holds its value
        end if;
    end process;

    -- Continuously output the internal register's state
    Flags_out <= s_flags_internal;

end architecture Behavioral;
```



4.6 Control Unit

The **Control Unit** is the central nervous system of the microprocessor. It serves as the primary **combinational** logic block that directs the operation of the entire datapath. Its fundamental purpose is to decode the instruction fetched from memory and, based on that instruction, generate all the control signals required to orchestrate the actions of the Program Counter, Register File, and ALU for a single clock cycle.

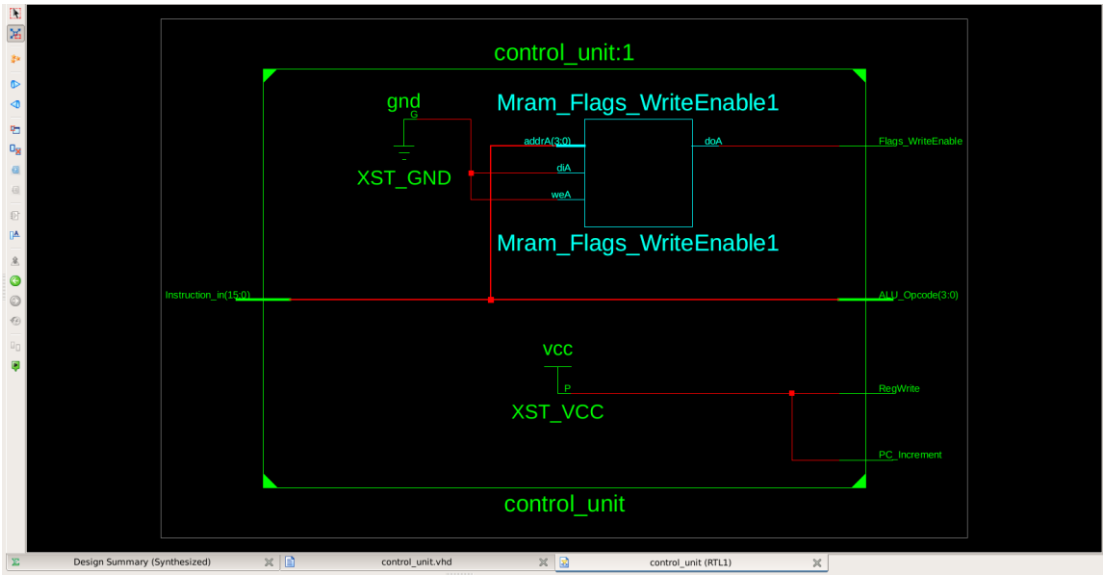
The decoding process is straightforward:

1. The Control Unit receives the 16-bit instruction word from the Instruction Memory.
2. It extracts the 4-bit **opcode** field ([15:12]) from this word.
3. This opcode is used as the input to the core decoding logic, which then asserts the appropriate output control signals.

The VHDL implementation consists of a single combinational process containing a case statement. This statement maps each of the 16 possible opcodes to a specific, hardwired set of values for the output control lines. A when others clause is included to define a safe, default state—effectively a No-Operation (NOP)—for any unused or illegal opcodes, ensuring predictable behavior. The table below illustrates the control signals generated for a few representative instruction types.

Control Signal Mapping

Instruction (Assembly)	Opcode	ALU_Opcode	RegWrite	Flags_ WriteEnable	PC_ Increment
ADD R3, R1, R2	0000	0000	1	1	1
GT R1, R2, R3	0011	0011	1	0	1
AND R5, R3, R4	0110	0110	1	1	1
NOT R6, R5	1001	1001	1	1	1

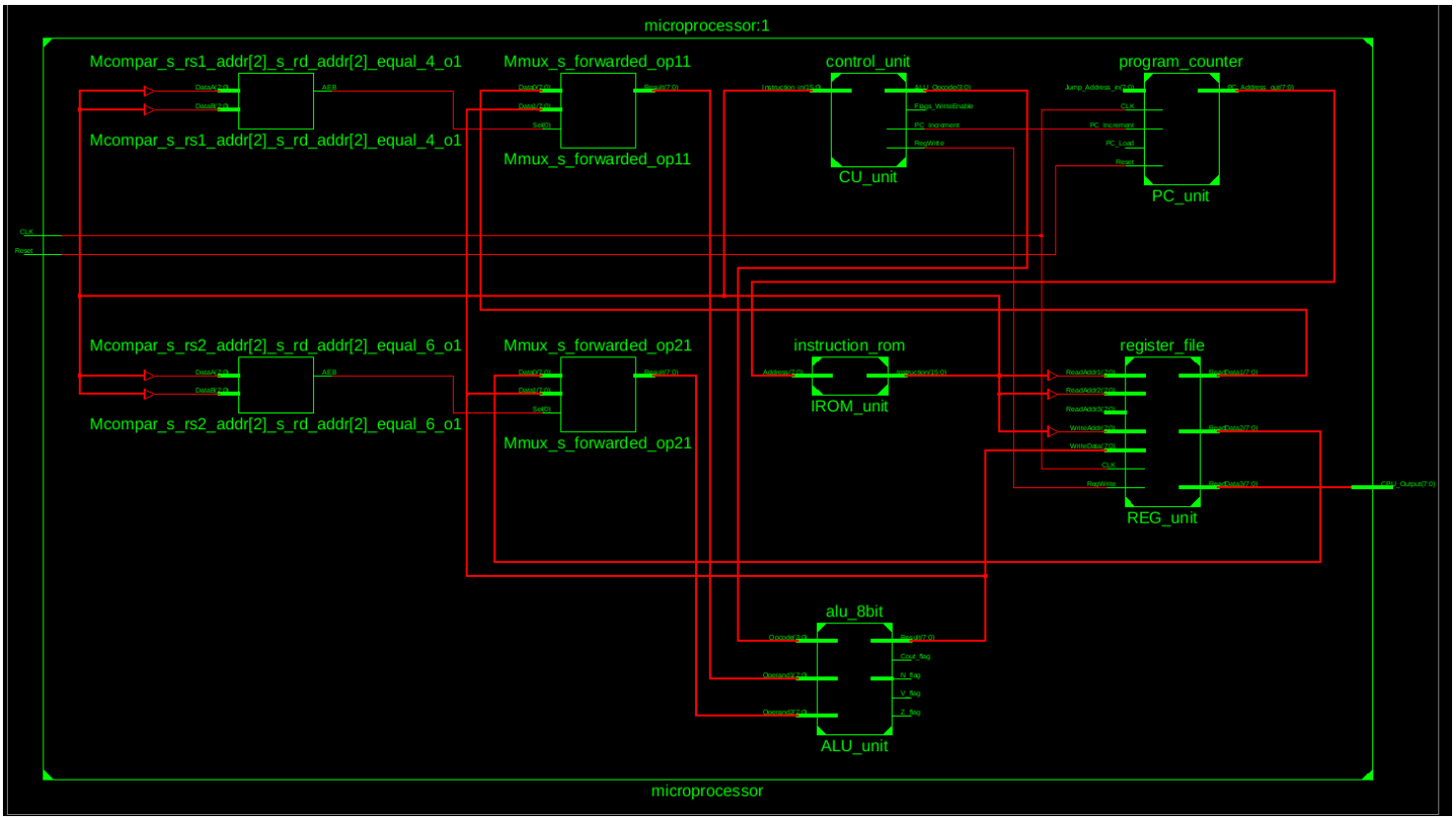


```vhdl

library ieee;
use ieee.std_logic_1164.all;

entity control_unit is
    port (
        -- Input from Instruction Memory
        Instruction_in    : in  std_logic_vector(15 downto 0);

        -- Outputs to control the entire datapath
        ALU_Opcode        : out std_logic_vector(3 downto 0);
        RegWrite          : out std_logic; -- Enable writing to the Register File
        Flags_WriteEnable : out std_logic; -- Enable writing to the Status Register
        PC_Increment      : out std_logic  -- Enable the PC to increment
    );
end entity control_unit;

architecture Behavioral of control_unit is
    signal s_opcode : std_logic_vector(3 downto 0);
begin
    -- Extract the 4-bit opcode from the full 16-bit instruction
    s_opcode <= Instruction_in(15 downto 12);

    process(s_opcode)
    begin
        -- By default, all control signals are inactive (a safe state)
        RegWrite          <= '0';
        Flags_WriteEnable <= '0';
        PC_Increment      <= '0';
        ALU_Opcode        <= "0000"; -- Default to ADD

        case s_opcode is
            -- Opcodes that write to registers and update flags
            when "0000" | "0001" | "0010" | "0110" | "0111" | "1000" | "1001" | "1010" | "1011" | "1100" | "1101" | "1110" | "1111" =>
                ALU_Opcode        <= s_opcode;
                RegWrite          <= '1';
                Flags_WriteEnable <= '1';
                PC_Increment      <= '1';

            -- Opcodes for comparators (GT, LT, EQ)
            when "0011" | "0100" | "0101" =>
                ALU_Opcode        <= s_opcode;
                RegWrite          <= '1';
                Flags_WriteEnable <= '0'; -- Comparators do not set NZCV flags
                PC_Increment      <= '1';

            -- Default case for unused/invalid opcodes
            when others =>
                RegWrite          <= '0';
                Flags_WriteEnable <= '0';
                PC_Increment      <= '0';
                ALU_Opcode        <= "0000"; -- NOP (can be defined as ADD R0, R0, R0)
        end case;
    end process;

end architecture Behavioral;
```

4.7 Top-Level Integration

The final step in the implementation process is the top-level integration, where all the previously designed and verified hardware modules are instantiated and interconnected to form the complete microprocessor. The architecture of this top-level entity is purely structural, meaning it contains no procedural logic of its own. Instead, it consists of component instantiations and the internal signals that function as the system's data and control buses.

This module is also where the crucial **Data Forwarding** logic is implemented. By using concurrent signal assignments to create multiplexers before the ALU's inputs, this module resolves the Read-After-Write data hazard, ensuring the integrity of the datapath. The VHDL code below represents the complete and final wiring of the microprocessor.



```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity microprocessor is
    port (
        CLK        : in  std_logic;
        Reset      : in  std_logic;
        CPU_Output : out std_logic_vector(7 downto 0)
    );
end entity microprocessor;

architecture Structural of microprocessor is

    -- Component Declarations (register_file is now the 2-port version)
    component alu_8bit is
        port (
            Operand1   : in  std_logic_vector(7 downto 0);
            Operand2   : in  std_logic_vector(7 downto 0);
            Opcode     : in  std_logic_vector(3 downto 0);
            Result     : out std_logic_vector(7 downto 0);
            Cout_flag  : out std_logic;
            V_flag     : out std_logic;
            Z_flag     : out std_logic;
            N_flag     : out std_logic
        );
    end component;

    component register_file is
        port (
            CLK        : in  std_logic;
            RegWrite   : in  std_logic;
            ReadAddr1  : in  std_logic_vector(2 downto 0);
            ReadAddr2  : in  std_logic_vector(2 downto 0);
            WriteAddr  : in  std_logic_vector(2 downto 0);
            WriteData  : in  std_logic_vector(7 downto 0);
            ReadData1  : out std_logic_vector(7 downto 0);
            ReadData2  : out std_logic_vector(7 downto 0)
        );
    end component;

    component instruction_rom is
        port (
            Address     : in  std_logic_vector(7 downto 0);
            Instruction : out std_logic_vector(15 downto 0)
        );
    end component;

    component program_counter is
        port (
            CLK             : in  std_logic;
            Reset           : in  std_logic;
            PC_Load         : in  std_logic;
            PC_Increment    : in  std_logic;
            Jump_Address_in : in  std_logic_vector(7 downto 0);
            PC_Address_out  : out std_logic_vector(7 downto 0)
        );
    end component;

    component status_register is
        port (
            CLK               : in  std_logic;
            Reset             : in  std_logic;
            Flags_WriteEnable : in  std_logic;
            Flags_in          : in  std_logic_vector(3 downto 0);
            Flags_out         : out std_logic_vector(3 downto 0)
        );
    end component;

    component control_unit is
        port (
            Instruction_in     : in  std_logic_vector(15 downto 0);
            ALU_Opcode         : out std_logic_vector(3 downto 0);
            RegWrite           : out std_logic;
            Flags_WriteEnable  : out std_logic;
            PC_Increment       : out std_logic
        );
    end component;

    -- Internal Signals (Buses)
    signal s_pc_address     : std_logic_vector(7 downto 0);
    signal s_instruction    : std_logic_vector(15 downto 0);
    signal s_rd_addr, s_rs1_addr, s_rs2_addr : std_logic_vector(2 downto 0);
    signal s_reg_read_data1, s_reg_read_data2 : std_logic_vector(7 downto 0);
    signal s_alu_result     : std_logic_vector(7 downto 0);
    signal s_alu_flags      : std_logic_vector(3 downto 0);
    signal s_status_reg_out : std_logic_vector(3 downto 0);
    signal s_alu_opcode     : std_logic_vector(3 downto 0);
    signal s_reg_write      : std_logic;
    signal s_flags_write    : std_logic;
    signal s_pc_increment   : std_logic;
    signal s_forwarded_op1, s_forwarded_op2  : std_logic_vector(7 downto 0);
```

```
92
93  begin
94      -- Decode register addresses
95      s_rd_addr  <= s_instruction(11 downto 9);
96      s_rs1_addr <= s_instruction(8 downto 6);
97      s_rs2_addr <= s_instruction(5 downto 3);
98
99      -- Data Forwarding Logic
100     s_forwarded_op1 <= s_alu_result when (s_reg_write = '1' and s_rd_addr = s_rs1_addr) else s_reg_read_data1;
101     s_forwarded_op2 <= s_alu_result when (s_reg_write = '1' and s_rd_addr = s_rs2_addr) else s_reg_read_data2;
102
103     -- Connect CPU_Output directly to the ALU's internal result bus
104     CPU_Output <= s_alu_result;
105
106     -- Instantiate all CPU components
107     PC_unit: program_counter
108         port map (
109             CLK             => CLK,
110             Reset           => Reset,
111             PC_Load         => '0',
112             PC_Increment    => s_pc_increment,
113             Jump_Address_in => (others => '0'),
114             PC_Address_out  => s_pc_address
115         );
116
117     IROM_unit: instruction_rom
118         port map (
119             Address     => s_pc_address,
120             Instruction => s_instruction
121         );
122
123     CU_unit: control_unit
124         port map (
125             Instruction_in     => s_instruction,
126             ALU_Opcode         => s_alu_opcode,
127             RegWrite           => s_reg_write,
128             Flags_WriteEnable  => s_flags_write,
129             PC_Increment       => s_pc_increment
130         );
131
132     FLAG_unit: status_register
133         port map (
134             CLK               => CLK,
135             Reset             => Reset,
136             Flags_WriteEnable => s_flags_write,
137             Flags_in          => s_alu_flags,
138             Flags_out         => s_status_reg_out
139         );
140
141     REG_unit: register_file
142         port map (
143             CLK        => CLK,
144             RegWrite   => s_reg_write,
145             ReadAddr1  => s_rs1_addr,
146             ReadAddr2  => s_rs2_addr,
147             WriteAddr  => s_rd_addr,
148             WriteData  => s_alu_result,
149             ReadData1  => s_reg_read_data1,
150             ReadData2  => s_reg_read_data2
151         );
152
153     ALU_unit: alu_8bit
154         port map (
155             Operand1  => s_forwarded_op1,
156             Operand2  => s_forwarded_op2,
157             Opcode    => s_alu_opcode,
158             Result    => s_alu_result,
159             Cout_flag => s_alu_flags(1),
160             V_flag    => s_alu_flags(0),
161             Z_flag    => s_alu_flags(2),
162             N_flag    => s_alu_flags(3)
163         );
164
165 end architecture Structural;
```

5.0 Verification and Testing

Verification is a critical phase in the hardware design lifecycle, ensuring that the VHDL implementation of the microprocessor is logically correct and behaves as intended by the architectural specification. This section outlines the strategy used for simulation, the tools employed, and the results obtained from both component-level and system-level testing.

5.1 Simulation Strategy

To manage the complexity of verifying a multi-component system, a systematic, **bottom-up testing** methodology was adopted. This approach involves verifying each hardware module in isolation before testing them as an integrated system, allowing for more targeted debugging and ensuring that the fundamental building blocks are robust. The entire simulation process was conducted using the ModelSim HDL simulation environment.

The strategy was divided into two primary stages:

1. **Component-Level Testing:** Each VHDL module, including the ALU, Register File, Program Counter, and Control Unit, was individually tested using a dedicated VHDL testbench. Each testbench was designed to apply a comprehensive set of input vectors that covered typical operations, boundary conditions, and edge cases specific to that module's function. The primary goal of this stage was to confirm that each component behaved correctly according to its own specification.

2. **System-Level Testing:** Once all individual components were successfully verified, the top-level microprocessor entity was tested. This integration test involved simulating the execution of a complete, multi-line program stored in the Instruction ROM. The testbench for this stage was simple, providing only the master clock and reset signals. This allowed the processor to run freely, fetching, decoding, and executing the program instructions sequentially.

For both stages, verification was performed through **manual waveform inspection**. The output signals generated by the simulation were visually compared against pre-defined verification tables to confirm that the design's behavior matched the expected outcome at each clock cycle.

5.2 Component-Level Simulation

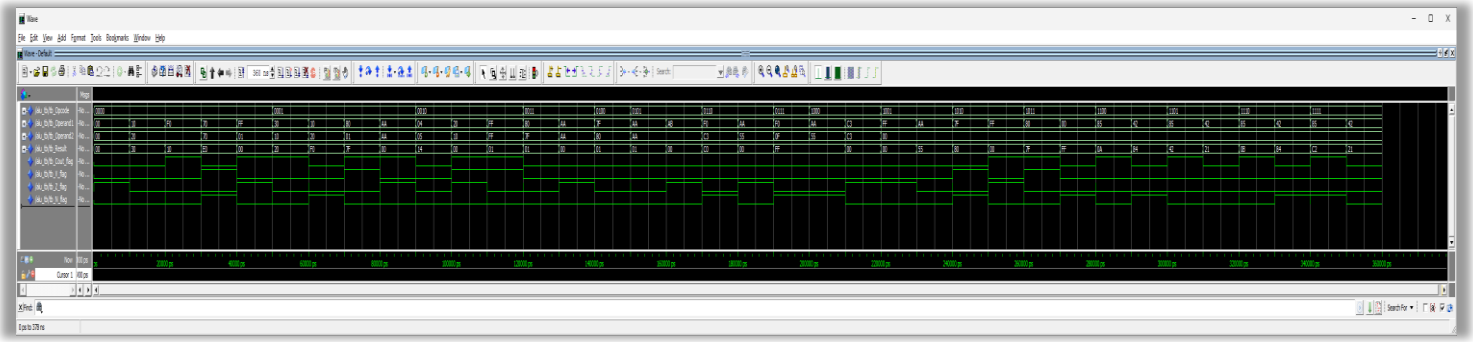
Following the bottom-up strategy, each module was tested in isolation to verify its logical correctness. For each component listed below, a dedicated VHDL testbench was developed to apply a specific set of input stimuli. The resulting output was captured in a simulation waveform and validated against a pre-defined verification table. This process ensured that each building block was fully functional before its inclusion in the top-level microprocessor design.

5.2.1 Arithmetic Logic Unit (ALU)

The ALU was tested to confirm the correct execution of all 16 specified arithmetic and logical operations. The testbench applied a series of operand pairs designed to validate not only the 8-bit Result but also the behavior of all four status flags (N, Z, C, V). Special attention was given to edge cases, including carry generation, signed overflow conditions, and multiplication that exceeded the 8-bit result capacity.

8-bit ALU Verification Table

Opcode	Operation	Operand1 (Hex)	Operand2 (Hex)	Expected Result (Hex)	C	V	Z	N	Notes
0000	ADD	10	20	30	0	0	0	0	Simple addition
		F0	20	10	1	0	0	0	Carry out
		70	70	E0	0	1	0	1	Signed overflow (Pos+Pos=Neg)
		FF	01	00	1	0	1	0	Add to make zero
0001	SUB	30	10	20	1	0	0	0	Simple subtraction
		10	20	F0	0	0	0	1	Borrow needed
		80	01	7F	1	1	0	0	Signed overflow (Neg-Pos=Pos)
		AA	AA	00	1	0	1	0	Subtract to make zero
0010	MUL	04	05	14	0	0	0	0	Simple multiplication
		20	10	00	1	1	1	0	Result x"0200", overflow
		FF	FF	01	1	1	0	0	Max value, result x"FE01"
0011	GT (>)	80	7F	01	0	0	0	0	A is greater
		AA	AA	00	0	0	1	0	A is equal
		7F	80	00	0	0	1	0	A is less
0100	LT (<)	7F	80	01	0	0	0	0	A is less
		AA	AA	00	0	0	1	0	A is equal
		80	7F	00	0	0	1	0	A is greater
0101	EQ (=)	AA	AA	01	0	0	0	0	A is equal
		AB	AA	00	0	0	1	0	A is not equal
0110	AND	F0	C3	C0	0	0	0	1	
		AA	55	00	0	0	1	0	
0111	OR	F0	0F	FF	0	0	0	1	
		00	00	00	0	0	1	0	
1000	XOR	AA	55	FF	0	0	0	1	
		C3	C3	00	0	0	1	0	
1001	NOT	FF	--	00	0	0	1	0	Operand2 is ignored
		AA	--	55	0	0	0	0	Operand2 is ignored
1010	INC	7F	--	80	0	1	0	1	Signed overflow
		FF	--	00	1	0	1	0	Carry out
1011	DEC	80	--	7F	1	1	0	0	Signed overflow
		00	--	FF	0	0	0	1	Borrow needed
1100	LSL	85	--	0A	1	0	0	0	Bit 1 shifted to carry
		42	--	84	0	0	0	1	Bit 0 shifted to carry
1101	LSR	85	--	42	1	0	0	0	Bit 1 shifted to carry
		42	--	21	0	0	0	0	Bit 0 shifted to carry
1110	ROL	85	--	0B	1	0	0	0	Bit 1 rotated
		42	--	84	0	0	0	1	Bit 0 rotated
1111	ROR	85	--	C2	1	0	0	1	Bit 1 rotated
		42	--	21	0	0	0	0	Bit 0 rotated



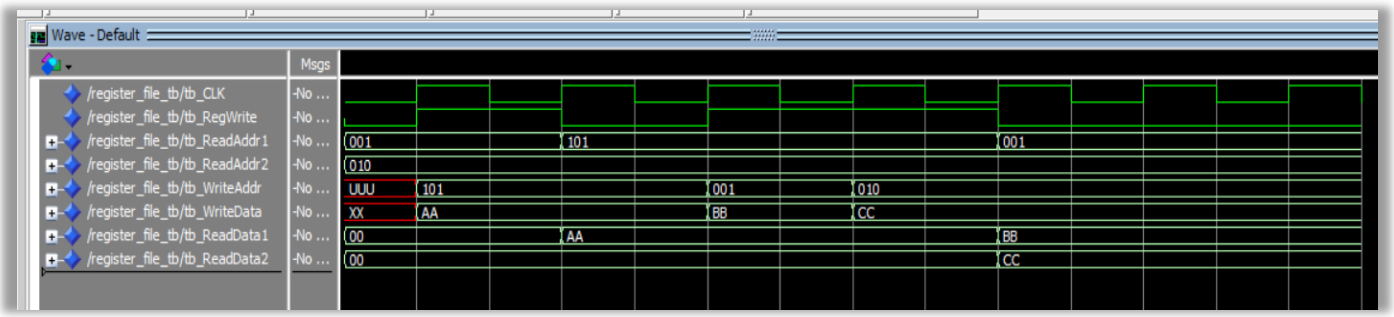
5.2.2 Register File

The Register File was tested to verify its dual-port read and single-port write capabilities. The simulation sequence confirmed the **synchronous** nature of the write operation, data was only latched on a rising clock edge when the RegWrite signal was asserted. The test also validated the **asynchronous** behavior of the two read ports, which provided data instantaneously upon a change in the read address inputs. The test sequence involved writing distinct values to several registers and then reading them back simultaneously to ensure data integrity.

Verification Table for register_file

Clock Cycle	Action During Cycle	RegWrite	Write Addr	Write Data	Read Addr1	Read Addr2	Expected ReadData1	Expected ReadData2	Notes
1	Read R1, R2	0	---	---	001	010	x"00"	x"00"	Initial state, all registers are zero.
2	Write x"AA" to R5	1	101	x"AA"	001	010	x"00"	x"00"	Write is armed. Read ports are unchanged.
3	Read R5	0	---	---	101	010	x"AA"	x"00"	Write to R5 is now complete. Reading it back on ReadData1.
4	Write x"BB" to R1	1	001	x"BB"	101	010	x"AA"	x"00"	Write to R1 is armed. Read ports are unchanged.
5	Write x"CC" to R2	1	010	x"CC"	101	010	x"AA"	x"00"	Write to R1 is complete. Write to R2 is armed.
6	Read R1, R2	0	---	---	001	010	x"BB"	x"CC"	Write to R2 is complete. Reading both new values.
7	Hold and Observe	0	---	---	001	010	x"BB"	x"CC"	Values are stable.

```
library ieee;
use ieee.std_logic_1164.all;

entity register_file_tb is
end entity register_file_tb;

architecture behavioral of register_file_tb is

    component register_file is
        port (
            CLK       : in  std_logic;
            RegWrite  : in  std_logic;
            ReadAddr1 : in  std_logic_vector(2 downto 0);
            ReadAddr2 : in  std_logic_vector(2 downto 0);
            WriteAddr : in  std_logic_vector(2 downto 0);
            WriteData : in  std_logic_vector(7 downto 0);
            ReadData1 : out std_logic_vector(7 downto 0);
            ReadData2 : out std_logic_vector(7 downto 0)
        );
    end component register_file;

    -- Testbench signals
    signal tb_CLK       : std_logic := '0';
    signal tb_RegWrite  : std_logic;
    signal tb_ReadAddr1 : std_logic_vector(2 downto 0);
    signal tb_ReadAddr2 : std_logic_vector(2 downto 0);
    signal tb_WriteAddr : std_logic_vector(2 downto 0);
    signal tb_WriteData : std_logic_vector(7 downto 0);
    signal tb_ReadData1 : std_logic_vector(7 downto 0);
    signal tb_ReadData2 : std_logic_vector(7 downto 0);

    constant clk_period : time := 10 ns;

begin
    -- Instantiate the Device Under Test
    DUT: register_file
        port map (
            CLK       => tb_CLK,      RegWrite  => tb_RegWrite,  ReadAddr1 => tb_ReadAddr1,
            ReadAddr2 => tb_ReadAddr2, WriteAddr => tb_WriteAddr, WriteData => tb_WriteData,
            ReadData1 => tb_ReadData1, ReadData2 => tb_ReadData2
        );

    -- Clock generation process
    tb_CLK <= not tb_CLK after clk_period/2;

    -- Stimulus process with corrected timing
    stimulus_proc: process
    begin
        -- Step 1: Initial state, reading R1 and R2
        tb_RegWrite <= '0';
        tb_ReadAddr1 <= "001";
        tb_ReadAddr2 <= "010";
        wait until rising_edge(tb_CLK);

        -- Step 2: Set up to write x"AA" to R5
        tb_RegWrite <= '1';
        tb_WriteAddr <= "101";
        tb_WriteData <= x"AA";
        wait until rising_edge(tb_CLK);   -- The write happens here

        -- Step 3: Verify the write. Disable write, set read address to R5.
        tb_RegWrite <= '0';
        tb_ReadAddr1 <= "101";
        tb_ReadAddr2 <= "010"; -- Keep reading R2
        wait until rising_edge(tb_CLK); -- ReadData1 now shows x"AA"

        -- Step 4: Set up to write x"BB" to R1
        tb_RegWrite <= '1';
        tb_WriteAddr <= "001";
        tb_WriteData <= x"BB";
        wait until rising_edge(tb_CLK); -- The write to R1 happens here

        -- Step 5: Set up to write x"CC" to R2
        tb_WriteAddr <= "010";
        tb_WriteData <= x"CC";
        wait until rising_edge(tb_CLK); -- The write to R2 happens here

        -- Step 6: Verify final state. Disable write, read R1 and R2.
        tb_RegWrite <= '0';
        tb_ReadAddr1 <= "001";
        tb_ReadAddr2 <= "010";
        wait for clk_period * 2; -- Wait a couple cycles to observe the final stable state

        -- Halt the simulation
        wait;
    end process stimulus_proc;
end architecture behavioral;
```



5.2.3 Instruction Memory (ROM)

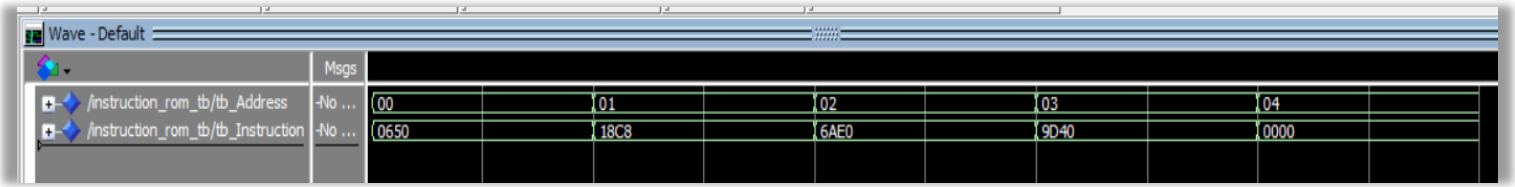
The Instruction Memory test was designed to confirm that the correct 16-bit instruction word was output for a given 8-bit address. The testbench sequentially provided the memory addresses corresponding to the pre-loaded 4-line program. The resulting Instruction output on the waveform was compared against the program's memory map to verify that the machine code was stored and read back correctly.

3. Verification Table (Program Memory Map)

This table documents the simple program loaded into our ROM, showing how the assembly-level instructions translate into the 16-bit machine code based on our defined format.

Address (Hex)	Machine Code (Hex)	Instruction (Assembly)	Breakdown (Op Rd Rs1 Rs2)
0x00	0650	ADD R3, R1, R2	0000 011 001 010
0x01	18C8	SUB R4, R3, R1	0001 100 011 001
0x02	6AE0	AND R5, R3, R4	0110 101 011 100
0x03	9D40	NOT R6, R5	1001 110 101 ---
0x04..0xFF	0000	NOP	0000 000 000 000

```
library ieee;
use ieee.std_logic_1164.all;

entity instruction_rom_tb is
end entity instruction_rom_tb;

architecture behavioral of instruction_rom_tb is

    component instruction_rom is
        port (
            Address     : in  std_logic_vector(7 downto 0);
            Instruction : out std_logic_vector(15 downto 0)
        );
    end component instruction_rom;

    -- Testbench signals
    signal tb_Address     : std_logic_vector(7 downto 0);
    signal tb_Instruction : std_logic_vector(15 downto 0);

    constant stimulus_delay : time := 10 ns;

begin
    -- Instantiate the Device Under Test
    DUT: instruction_rom port map (Address => tb_Address, Instruction => tb_Instruction);

    -- Stimulus process to read from the ROM
    stimulus_proc: process
    begin
        -- Read from Address 0x00
        tb_Address <= x"00";
        wait for stimulus_delay;

        -- Read from Address 0x01
        tb_Address <= x"01";
        wait for stimulus_delay;

        -- Read from Address 0x02
        tb_Address <= x"02";
        wait for stimulus_delay;

        -- Read from Address 0x03
        tb_Address <= x"03";
        wait for stimulus_delay;

        -- Read from an empty Address (should be 0000)
        tb_Address <= x"04";
        wait for stimulus_delay;

        -- Halt the simulation
        wait;
    end process stimulus_proc;
end architecture behavioral;
```



5.2.4 Program Counter (PC)

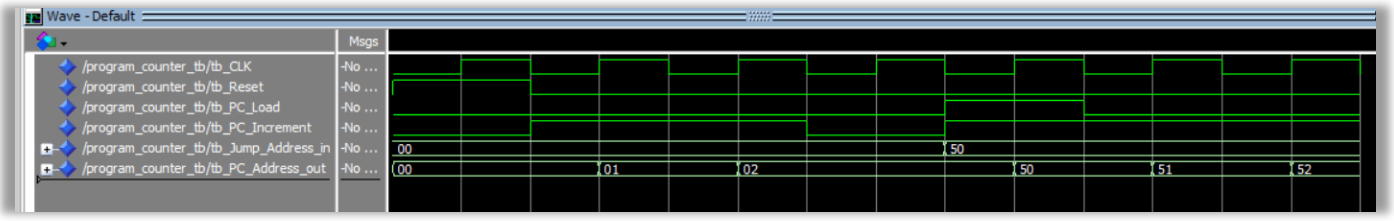
The Program Counter was tested to validate its four primary functions: asynchronous reset, synchronous increment, synchronous load, and hold. The simulation sequence verified that the PC correctly resets to x"00", increments sequentially when enabled, holds its value when disabled, and that the PC_Load signal correctly overrides the PC_Increment signal to facilitate jumps.

Verification Table

This table shows how the PC's output changes based on the control signals at each clock cycle.

Clock Cycle	Reset	PC_Load	PC_Increment	Jump_Address_in	Expected PC_Address_out	Notes
0 (Initial)	1	0	0	x"00"	x"00"	Asynchronous reset is active.
1	0	0	1	x"00"	x"00"	Setup to increment.
2	0	0	1	x"00"	x"01"	PC increments on clock edge.
3	0	0	0	x"00"	x"02"	Setup to hold value.
4	0	0	0	x"00"	x"02"	PC holds its value.
5	0	1	1	x"50"	x"02"	Setup to load x"50". Load has priority.
6	0	0	1	x"50"	x"50"	PC loaded x"50". Now setup to increment.
7	0	0	1	x"50"	x"51"	PC increments from the new address.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity program_counter_tb is
end entity program_counter_tb;

architecture behavioral of program_counter_tb is

    component program_counter is
        port (
            CLK             : in  std_logic;
            Reset           : in  std_logic;
            PC_Load         : in  std_logic;
            PC_Increment    : in  std_logic;
            Jump_Address_in : in  std_logic_vector(7 downto 0);
            PC_Address_out  : out std_logic_vector(7 downto 0)
        );
    end component program_counter;

    -- testbench signals
    signal tb_CLK             : std_logic := '0';
    signal tb_Reset           : std_logic := '0';
    signal tb_PC_Load         : std_logic := '0';
    signal tb_PC_Increment    : std_logic := '0';
    signal tb_Jump_Address_in : std_logic_vector(7 downto 0) := (others => '0');
    signal tb_PC_Address_out  : std_logic_vector(7 downto 0);

    constant clk_period : time := 10 ns;

begin
    -- Instantiate the Device Under Test
    DUT: program_counter port map (
        CLK => tb_CLK, Reset => tb_Reset,
        PC_Increment => tb_PC_Increment, PC_Load => tb_PC_Load,
        Jump_Address_in => tb_Jump_Address_in, PC_Address_out => tb_PC_Address_out
    );

    -- Clock generation
    tb_CLK <= not tb_CLK after clk_period/2;

    -- Stimulus process
    stimulus_proc: process
    begin
        -- Assert Reset
        tb_Reset <= '1';
        wait for clk_period;

        -- De-assert Reset and start incrementing
        tb_Reset <= '0';
        tb_PC_Load <= '0';
        tb_PC_Increment <= '1';
        wait for clk_period * 2; -- Increment twice (to 0x01, then 0x02)

        -- Hold the value
        tb_PC_Increment <= '0';
        wait for clk_period;

        -- Load a new address (Load should override Increment)
        tb_PC_Load <= '1';
        tb_PC_Increment <= '1';
        tb_Jump_Address_in <= x"50";
        wait for clk_period;

        -- Disable load and continue incrementing
        tb_PC_Load <= '0';
        tb_PC_Increment <= '1';
        wait for clk_period * 2; -- Increment twice (to 0x51, then 0x52)

        -- Halt simulation
        wait;
    end process stimulus_proc;
end architecture behavioral;
```



5.2.5 Status Register

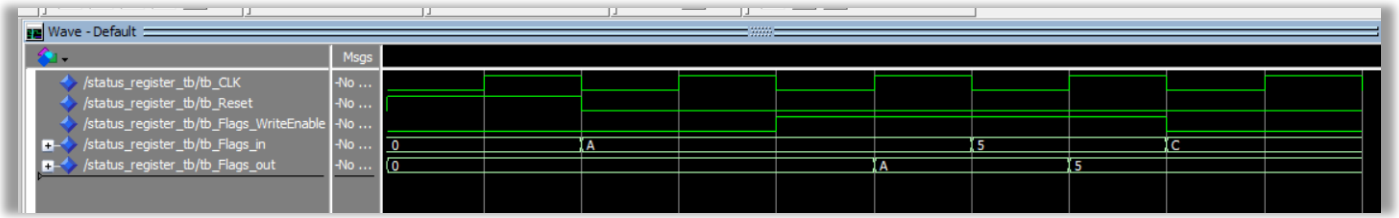
The 4-bit Status Register was tested to confirm its ability to correctly load and hold the NZCV flag vector. The test sequence verified that the register's contents updated on the rising clock edge only when the Flags_WriteEnable signal was active, held its previous state when the enable was inactive, and was cleared to x"0" by the asynchronous reset.

Verification Table

This table shows how the register's output behaves based on the Reset and Flags_WriteEnable control signals over several clock cycles.

Clock Cycle	Reset	Flags_ WriteEnable	Flags_in	Expected Flags_out	Notes
0 (Initial)	1	0	x"0"	x"0"	Asynchronous reset is active.
1	0	0	x"A"	x"0"	Write is disabled; register holds x"0".
2	0	1	x"A"	x"0"	Setup to load x"A" (1010).
3	0	1	x"5"	x"A"	Load x"A" completes. Setup to load x"5".
4	0	0	x"5"	x"5"	Load x"5" completes. Disable write.
5	0	0	x"C"	x"5"	Write is disabled; register holds x"5".

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity status_register_tb is
end entity status_register_tb;

architecture behavioral of status_register_tb is

    component status_register is
        port (
            CLK              : in  std_logic;
            Reset            : in  std_logic;
            Flags_WriteEnable : in  std_logic;
            Flags_in         : in  std_logic_vector(3 downto 0);
            Flags_out        : out std_logic_vector(3 downto 0)
        );
    end component status_register;

    -- Testbench signals initialized to '0'
    signal tb_CLK              : std_logic := '0';
    signal tb_Reset            : std_logic := '0';
    signal tb_Flags_WriteEnable : std_logic := '0';
    signal tb_Flags_in         : std_logic_vector(3 downto 0) := (others => '0');
    signal tb_Flags_out        : std_logic_vector(3 downto 0);

    constant clk_period : time := 10 ns;

begin
    -- Instantiate the Device Under Test
    DUT: status_register
        port map (
            CLK              => tb_CLK,
            Reset            => tb_Reset,
            Flags_WriteEnable => tb_Flags_WriteEnable,
            Flags_in         => tb_Flags_in,
            Flags_out        => tb_Flags_out
        );

    -- Clock generation
    tb_CLK <= not tb_CLK after clk_period/2;

    -- Stimulus process
    stimulus_proc: process
    begin
        -- Assert Reset
        tb_Reset <= '1';
        wait for clk_period;
        tb_Reset <= '0';

        -- Hold state (WriteEnable is '0')
        tb_Flags_in <= x"A";
        wait for clk_period;

        -- Enable write to load x"A"
        tb_Flags_WriteEnable <= '1';
        wait for clk_period;

        -- Set up next value (x"5") while previous value (x"A") is being observed
        tb_Flags_in <= x"5";
        wait for clk_period;

        -- Disable write to hold the new value (x"5")
        tb_Flags_WriteEnable <= '0';
        tb_Flags_in <= x"C"; -- This input should be ignored
        wait for clk_period;

        -- Halt simulation
        wait;
    end process stimulus_proc;
end architecture behavioral;
```



5.2.6 Control Unit

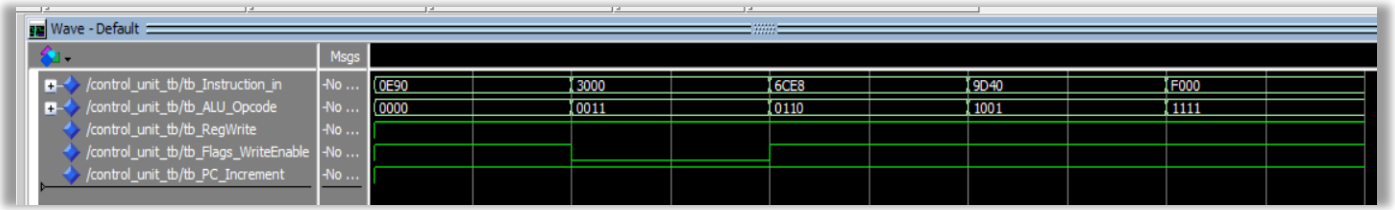
The Control Unit's combinational decoding logic was tested by providing a series of sample 16-bit instructions to its input. The simulation waveform was then inspected to verify that the unit produced the correct pattern of output control signals (ALU_Opcode, RegWrite, Flags_WriteEnable, etc.) for each instruction type. The test also included an invalid opcode to confirm that the unit reverted to a safe, default state.

Functional Table (Control Signal Mapping)

This table shows the expected control signals for a few representative instruction types.

Instruction (Assembly)	Opcode	ALU_Opcode	RegWrite	Flags_ WriteEnable	PC_ Increment
ADD R3, R1, R2	0	0	1	1	1
GT R1, R2, R3	11	11	1	0	1
AND R5, R3, R4	110	110	1	1	1
NOT R6, R5	1001	1001	1	1	1

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity control_unit_tb is
end entity control_unit_tb;

architecture behavioral of control_unit_tb is

    component control_unit is
        port (
            Instruction_in    : in  std_logic_vector(15 downto 0);
            ALU_Opcode        : out std_logic_vector(3 downto 0);
            RegWrite          : out std_logic;
            Flags_WriteEnable : out std_logic;
            PC_Increment      : out std_logic
        );
    end component control_unit;

    -- Testbench signals initialized to '0'
    signal tb_Instruction_in    : std_logic_vector(15 downto 0) := (others => '0');
    signal tb_ALU_Opcode        : std_logic_vector(3 downto 0);
    signal tb_RegWrite          : std_logic;
    signal tb_Flags_WriteEnable : std_logic;
    signal tb_PC_Increment      : std_logic;

    constant stimulus_delay : time := 10 ns;

begin
    -- Instantiate the Device Under Test
    DUT: control_unit
        port map (
            Instruction_in    => tb_Instruction_in,
            ALU_Opcode        => tb_ALU_Opcode,
            RegWrite          => tb_RegWrite,
            Flags_WriteEnable => tb_Flags_WriteEnable,
            PC_Increment      => tb_PC_Increment
        );

    -- Stimulus process
    stimulus_proc: process
    begin
        -- Test ADD instruction (Opcode 0000)
        tb_Instruction_in <= x"0E90"; -- Represents ADD R3, R1, R2
        wait for stimulus_delay;

        -- Test GT instruction (Opcode 0011)
        tb_Instruction_in <= x"3000"; -- Represents GT R0, R0, R0
        wait for stimulus_delay;

        -- Test AND instruction (Opcode 0110)
        tb_Instruction_in <= x"6CE8"; -- Represents AND R5, R3, R4
        wait for stimulus_delay;

        -- Test NOT instruction (Opcode 1001)
        tb_Instruction_in <= x"9D40"; -- Represents NOT R6, R5
        wait for stimulus_delay;

        -- Test an invalid Opcode (should result in safe default values)
        tb_Instruction_in <= x"F000";
        wait for stimulus_delay;

        -- Halt simulation
        wait;
    end process stimulus_proc;
end architecture behavioral;
```



5.3 Top-Level System Simulation

The final stage of verification is the **system-level integration** test. The purpose of this test is to confirm that all the individually verified modules—the ALU, Register File, PC, Instruction Memory, Status Register, and Control Unit—operate together correctly as a cohesive microprocessor. This test validates the entire datapath, the control signal distribution, and the critical timing of the integrated design.

The test procedure involved simulating the complete, top-level microprocessor entity. To provide initial data for the program, the Register File was pre-initialized with R1 = x"0A" and R2 = x"14". The system-level testbench then supplied a master clock and an initial reset pulse, allowing the processor to begin fetching and executing the 4-line **test program** stored in its Instruction ROM.

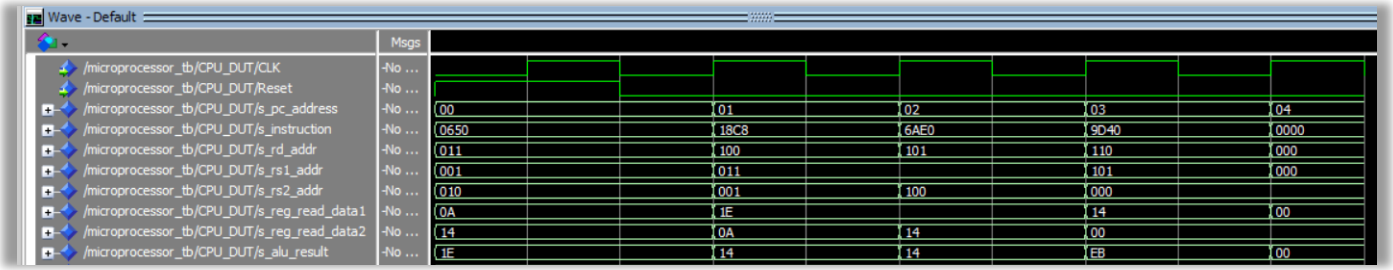
Verification was performed by observing the state of the key internal buses and control signals in the ModelSim waveform viewer. The correctness of the program's execution was confirmed by comparing the simulation output, cycle by cycle, against the detailed **System Verification Table** presented below. The simulation results precisely matched the expected values, thus successfully validating the final microprocessor design, including the data forwarding mechanism that resolved the data hazard.

Expected Outcome

(Initial State: R1 = x"0A", R2 = x"14")

Cycle	PC Addr	Instruction Fetched	Assembly	ALU Input 1	ALU Input 2	ALU Result	Final Register State
1	00	0650	ADD R3, R1, R2	x"0A"	x"14"	x"1E"	R3 write pending.
2	01	18C8	SUB R4, R3, R1	x"1E"	x"0A"	x"14"	R3 is now x"1E".
3	02	6AE0	AND R5, R3, R4	x"1E"	x"14"	x"14"	R4 is now x"14".
4	03	9D40	NOT R6, R5	x"14"	---	x"EB"	R5 is now x"14".
5	04	0000	NOP	x"00"	x"00"	x"00"	R6 is now x"EB".

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity microprocessor_tb is
end entity microprocessor_tb;

architecture behavioral of microprocessor_tb is

    component microprocessor is
        port (
            CLK   : in std_logic;
            Reset : in std_logic
        );
    end component microprocessor;

    -- Testbench signals
    signal tb_CLK   : std_logic := '0';
    signal tb_Reset : std_logic := '0';

    constant clk_period : time := 10 ns;
begin
    -- Instantiate the CPU
    CPU_DUT: microprocessor port map (CLK => tb_CLK, Reset => tb_Reset);

    -- Clock generation
    tb_CLK <= not tb_CLK after clk_period/2;

    -- Stimulus: Reset the CPU, then let it run
    stimulus_proc: process
    begin
        -- Assert Reset for one cycle to start the PC at 0
        tb_Reset <= '1';
        wait for clk_period;

        -- De-assert Reset and let the program execute
        tb_Reset <= '0';
        wait for clk_period * 10; -- Let it run for 10 cycles

        -- Halt the simulation
        wait;
    end process stimulus_proc;
end architecture behavioral;
```



6.0 Discussion and Challenges

6.1 Analysis of Results

The simulation results obtained from the top-level system test successfully validated the logical and architectural correctness of the 8-bit microprocessor design. The cycle-by-cycle trace of the internal buses and control signals, as observed in the final waveform, precisely matched the predicted values in the system verification table. This successful test confirms several key aspects of the design: the datapath components (ALU, Register File, PC) interact correctly; the Control Unit generates the proper control signals for each instruction in the test program; and the overall program flow is managed as intended.

Crucially, the successful execution of back-to-back dependent instructions (such as the SUB instruction using the result of the preceding ADD) validates the implementation of the data forwarding mechanism. The results demonstrate that the processor can correctly handle Read-After-Write data hazards, ensuring program integrity.

6.2 Key Challenges and Solutions

Two significant challenges were encountered during the integration and testing phase, both of which are common in processor design and provided valuable learning experiences.

- **Read-After-Write (RAW) Data Hazard:** The most critical architectural challenge was a RAW data hazard. The initial integrated simulation revealed that an instruction was attempting to read an operand from the Register File before the result of the previous instruction had been written back. This timing conflict caused the ALU to receive stale data, leading to a cascade of incorrect calculations. The solution was to implement a **Data Forwarding** path. This hardware modification creates a "shortcut" that bypasses the Register File, sending the ALU's result from the end of one cycle directly back to its inputs for the start of the next, ensuring the most current data is always available.

- **Machine Code Encoding Errors:** A persistent issue during debugging was the manual translation of assembly instructions into their 16-bit hexadecimal machine code equivalents for the Instruction ROM. Initial simulation failures were traced not to hardware logic errors, but to faulty "software." The processor was correctly executing the flawed instructions it was given. This was resolved through a meticulous, bit-by-bit re-verification of all machine code values against the defined ISA. This challenge highlighted the critical importance of verifying the test program itself with the same rigor as the hardware design.

6.3 Design Trade-offs

The primary architectural decision for this project was the implementation of a **single-cycle architecture**. This choice carries significant trade-offs between design simplicity and performance.

- **Advantages:** The main benefit of the single-cycle approach is its simplicity. The control logic is purely combinational, which makes it significantly easier to design, implement, and debug. Each instruction has a clear and self-contained execution path within one clock cycle, making the overall data flow easy to trace and understand.

- **Disadvantages:** The primary drawback is performance inefficiency. The **clock period** for the entire processor must be long enough to accommodate the single slowest instruction (e.g., multiplication). This means that faster instructions (such as a simple AND or ADD) complete their work early but must wait for the full clock cycle to finish, wasting valuable processing time. This design is therefore not suitable for high-performance applications.

For the educational goals of this project, the single-cycle architecture was the optimal choice, prioritizing clarity of design and fundamental principles over performance optimization.

7.0 Conclusion and Future Work

7.1 Conclusion

This project successfully achieved its primary objective: the complete design, implementation, and verification of a custom 8-bit, single-cycle microprocessor. Through a modular, bottom-up methodology using VHDL, all core components—including a 16-function ALU, an 8x8 register file, a program counter, and a combinational control unit—were developed and individually tested. These components were then successfully integrated into a top-level design based on a custom 16-bit ISA.

The final system-level simulation confirmed the processor's ability to correctly fetch, decode, and execute a sequence of instructions. Key architectural challenges, such as the Read-After-Write data hazard, were identified and successfully resolved with a data forwarding mechanism, demonstrating a practical application of fundamental computer architecture principles. The project concludes with a fully functional and verifiable VHDL model of a microprocessor, which serves as a robust and well-documented foundation for further study and enhancement.

7.2 Future Work and Enhancements

While the current design is a complete single-cycle processor, its capabilities can be significantly expanded. The modular nature of the design provides a clear path for the following future enhancements:

- **Data Memory and I/O:** The most critical next step is the implementation of a dedicated data memory (RAM). This would involve adding LOAD and STORE instructions to the ISA and expanding the control unit to manage memory read/write operations, allowing the processor to work with larger data sets.
- **Advanced Control Flow:** The functionality could be greatly improved by adding control flow instructions. This includes:
    - **Conditional Branches** (BEQ, BNE, etc.) that use the Status Register flags to make decisions.
    - **Unconditional Jumps** (JMP) to alter the program sequence.
    - A **Stack Pointer** and CALL/RETURN instructions to enable subroutines.
- **Architectural Improvements:** To overcome the performance limitations of the single-cycle design, the architecture could be evolved:
    - **Multi-Cycle Architecture:** Redesigning the processor to allow different instructions to take a variable number of clock cycles, improving overall efficiency.
    - **Pipelining:** Implementing a multi-stage pipeline (e.g., Fetch, Decode, Execute, Memory, Write-back) to overlap instruction execution and dramatically increase throughput.
- **FPGA Implementation:** The VHDL code for this processor is synthesizable. A logical next step would be to synthesize the design and implement it on a physical FPGA board to test its operation in real hardware.

References

- YouTube Tutorials.
- Scribd: https://www.scribd.com/document/470219727/A-complete-8-bit-Microcontroller-in-VHDL#:~:text=%EF%82%B7%20Condition%20Code%20Register%20,C
- Wikipedia.org: https://en.wikipedia.org/wiki/Arithmetic_logic_unit#:~:text=In%20computing%20%2C%20an%20arithmetic,3
- Wikipedia.org: https://en.wikipedia.org/wiki/Microprocessor
- Wikipedia.org: https://en.wikipedia.org/wiki/Harvard_architecture#:~:text=The%20Harvard%20architecture%20is%20a,3
- Instructables: https://www.instructables.com/Making-an-8-Bit-Computer/?utm_source
- Electronicsforu : https://www.electronicsforu.com/electronics-projects/simple-8-bit-computer-learning?utm_source