COMPUTER SYSTEM ARCHITECTURE

LAB REPORT #02

"Custom Processor"

Prepared by:
SHAZIL 532263

Presented To:
LE USAMA SHAUKAT

NUST CEME
DEPT. COMPUTER ENGINEERING

TABLE OF CONTENT

Lab#2 Custom Processor Design (Part 1)

Objective

The purpose of this lab is to sequentially design and implement a custom microprocessor.

Tools Required

- System Verilog
- Xilinx Vivado

Lab Task Guidelines

- The implementation must follow a **modular approach**. Each task should be written as a **separate module** and instantiated in the top-level file.
- The design must be **parameterized**, with all parameters defined and passed from the top-level module (`top.sv`). The required parameters are listed below:

Parameter Name (SystemVerilog)	Default Value	Description
`IMEM_DEPTH`	4 words	Depth of Instruction Memory
`REGF_WIDTH`	16 bits	Width of Register File
`ALU_WIDTH`	16 bits	Width of ALU
`PROG_VALUE`	3	Maximum Program Counter Value

Introduction to Processor Design

To design a simple microprocessor, the most essential foundation is the **Instruction Set Architecture (ISA)**. An ISA defines the collection of instructions that a processor can interpret and execute. In simple terms, it serves as the agreement between **software** (programs) and **hardware** (the processor).

Core Components of a Processor

- **Register File**
  Registers provide very fast data access but can only store a limited amount of information. They act as temporary storage locations, holding values the processor needs immediately for ongoing operations.
- **Control Unit (CU)**
  The control unit acts like the **orchestrator** of the processor. It fetches instructions from memory, decodes them to determine the required action, and then coordinates other units (such as the ALU) to carry out the operation.
- **Arithmetic Logic Unit (ALU)**
  The ALU is the computational powerhouse of the processor. It performs arithmetic tasks (like addition or subtraction) and logical operations (such as AND, OR, NOT, and comparisons). All calculations directed by the control unit are executed here.

- **Memory**
  While technically external to the processor, memory (e.g., RAM or storage) is essential for its functioning. Instructions and data are fetched from memory, processed within the registers and ALU, and the results may be written back to memory afterward.

Softwares used:

- System Verilog
- Xilinx Vivado

Lab Task 1: Design an Arithmetic Logic Unit

Based on the given instruction set architecture, design a parameterized n-bit Arithmetic Logic Unit (ALU) that supports the operations defined in the ISA. The functionality of the ALU must be thoroughly verified using a testbench.

.

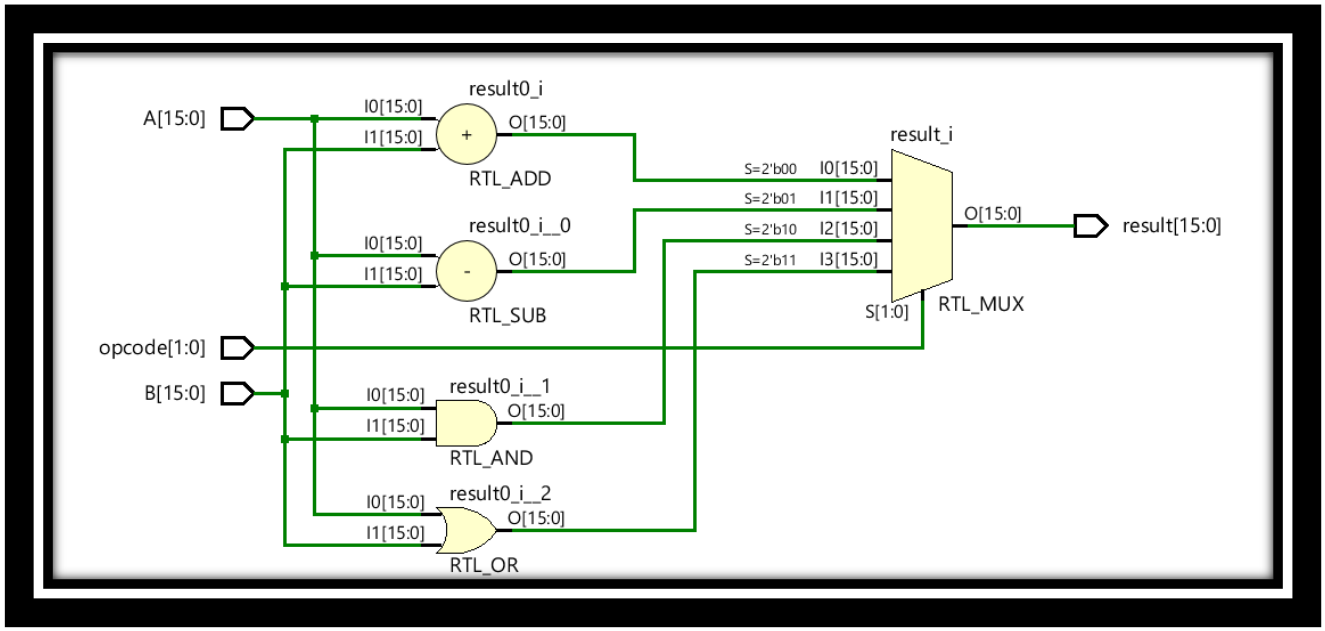
CODE:

```
module ALU #(
    parameter N = 16
)(
    input  logic [N-1:0] A,
    input  logic [N-1:0] B,
    input  logic [1:0]   opcode,
    output logic [N-1:0] result
);

    always_comb
    begin
        case (opcode)
            2'b00: result = A + B;
            2'b01: result = A - B;
            2'b10: result = A & B;
            2'b11: result = A | B;
            default: result = '0;
        endcase
    end

endmodule
```

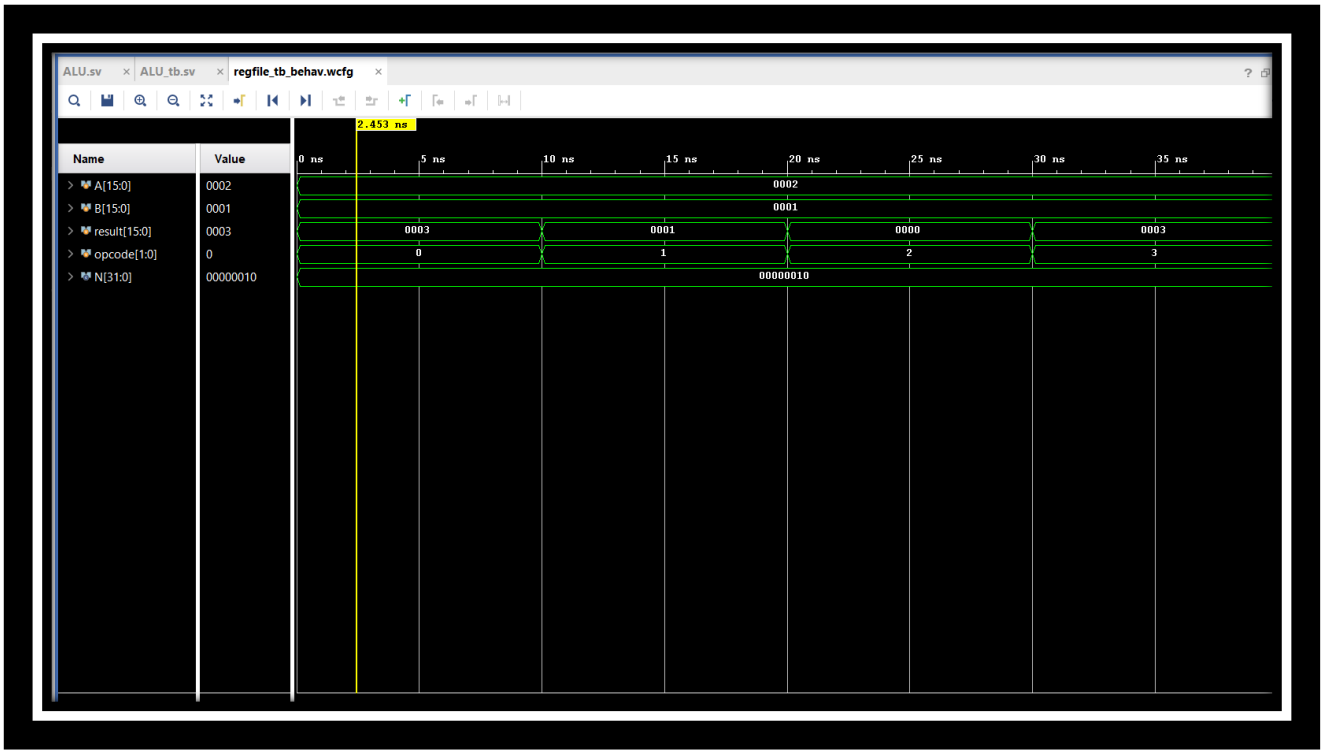
TESTBENCH CODE:

```
21
22
23  module ALU_tb;
24      parameter N =16;
25      logic [N-1:0] A, B;
26      logic [N-1:0] result;
27      logic [1:0] opcode;
28
29      ALU #(.N(N)) dut (.A(A) , .B(B) , .opcode(opcode), .result(result));
30
31      initial
32      begin
33          A = 16'b0000000000000010;
34          B = 16'b0000000000000001;
35          opcode = 2'b00;
36          #10;
37          A = 16'b0000000000000010;
38          B = 16'b0000000000000001;
39          opcode = 2'b01;
40          #10;
41
42          A = 16'b0000000000000010;
43          B = 16'b0000000000000001;
44          opcode = 2'b10;
45          #10;
46          A = 16'b0000000000000010;
47          B = 16'b0000000000000001;
48          opcode = 2'b11;
49          #10;
50
51       $finish;
52       end
53  endmodule
```

SCHEMATICS:



SIMULATIONS:



.....

Lab Task 2: Register File Design

Based on the given instruction set architecture, design a parameterized register file capable of providing two operands while allowing one result to be written back, each of width *n* bits, at the same time. The write-back operation should occur on the positive edge of the clock.

To load initial values into the register file, one of the available commands may be used. For better insight into microprocessor functionality, it is strongly encouraged to initialize the register file using a binary file.

CODE:

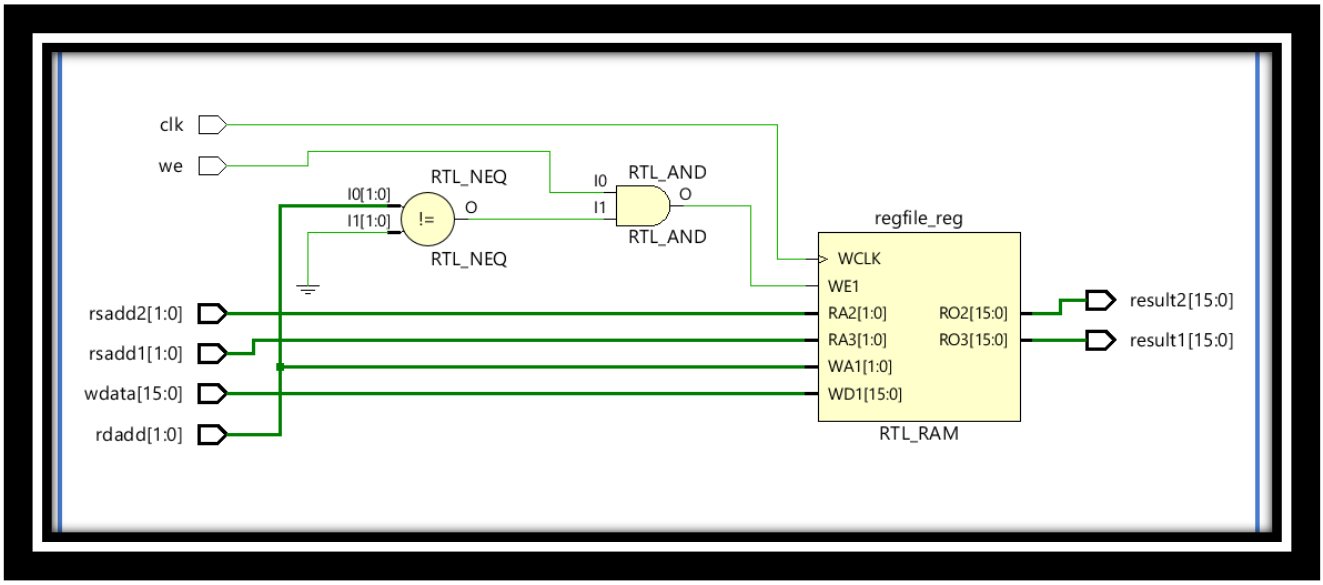
```
22
23  module Reg_file#(
24      parameter DATA_WIDTH = 16,
25      parameter NUM_REGS   = 4,
26      parameter e =2
27  )(
28      input logic clk,
29      input logic we,
30      input logic [e-1:0]rsadd1,
31      input logic [e-1:0]rsadd2,
32      input logic [e-1:0]rdadd,
33      input logic [DATA_WIDTH-1:0]wdata,
34      output logic [DATA_WIDTH-1:0]result1,
35      output logic [DATA_WIDTH-1:0]result2
36  );
37      logic [DATA_WIDTH-1:0] regfile [0:NUM_REGS-1];
38     initial
39     begin
40     $readmemb("rfdata.mem" , regfile);
41     end
42
43     always_ff@(posedge clk)
44     begin
45     if(we && rdadd !=0)
46       begin
47          regfile[rdadd] <= wdata;
48       end
49     end
50
51     assign result1 = regfile[rsadd1];
52     assign result2 = regfile[rsadd2];
53
54  endmodule
55
```

TESTBENCH CODE:

```
21
22
23  module regfile_tb;
24      parameter DATA_WIDTH = 16;
25      parameter NUM_REGS   = 4;
26      parameter e =2;
27      logic clk;
28      logic we;
29      logic [e-1:0]rsadd1;
30      logic [e-1:0]rsadd2;
31      logic [e-1:0]rdadd;
32      logic [DATA_WIDTH-1:0]wdata;
33      logic [DATA_WIDTH-1:0]result1;
34      logic [DATA_WIDTH-1:0]result2;
35
36      Reg_file #(
37      .DATA_WIDTH(DATA_WIDTH),
38      .NUM_REGS(NUM_REGS),
39       .e(e)
40       )
41       dut (
42       .clk(clk),
43       .we(we) ,
44       .rsadd1(rsadd1),
45       .rsadd2(rsadd2) ,
46       .rdadd(rdadd) ,
47       .wdata(wdata) ,
48       .result1(result1) ,
49       .result2(result2)
50       );
51
52       always
53       begin
```

```
52      always
53      begin
54      #5
55      clk = ~clk;
56      end
57
58      initial
59      begin
60         clk = 0;
61         we = 0;
62         rdadd = 0;
63         wdata = 0;
64         rsadd1 = 0;
65         rsadd2 = 0;
66         #10;
67
68         rsadd1 = 0;
69         rsadd2 = 1;
70         #10;
71
72         we = 1;
73         rdadd = 2;
74         wdata = 16'b0000000000000001;
75         #10;
76         we = 0;
77
78         rsadd1 = 2;
79         rsadd2 = 3;
80         #10;
81
82         $stop;
83         end
84  endmodule
```

SCHEMATICS:



SIMULATIONS:

……

Lab Task 3: Instruction Memory Design

To execute user-defined operations, the processor requires a memory unit to store instructions alongside the data being processed. Design an addressable memory module with an 8-bit word size and a parameterized depth (to define the number of instructions). Make sure the instruction memory supports initialization using a binary file.

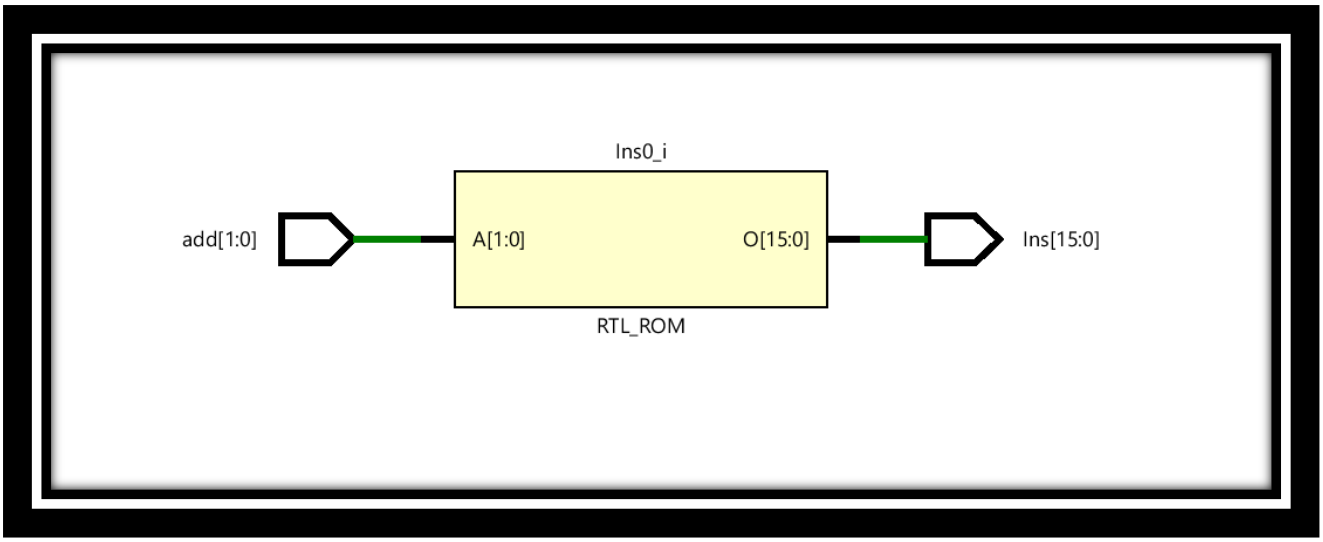
CODE:

```
module InsMemory #(parameter length = 4 , width = 16 , N = 2)
    (
    input logic [N-1:0] add,
    output logic [width-1:0] Ins
     );

     logic [width-1:0] InsMem [0:length-1];


     initial
     begin
     $readmemb("IMdata.mem" , InsMem);
     end

     always_comb
     begin
        Ins = InsMem[add];
     end
endmodule
```

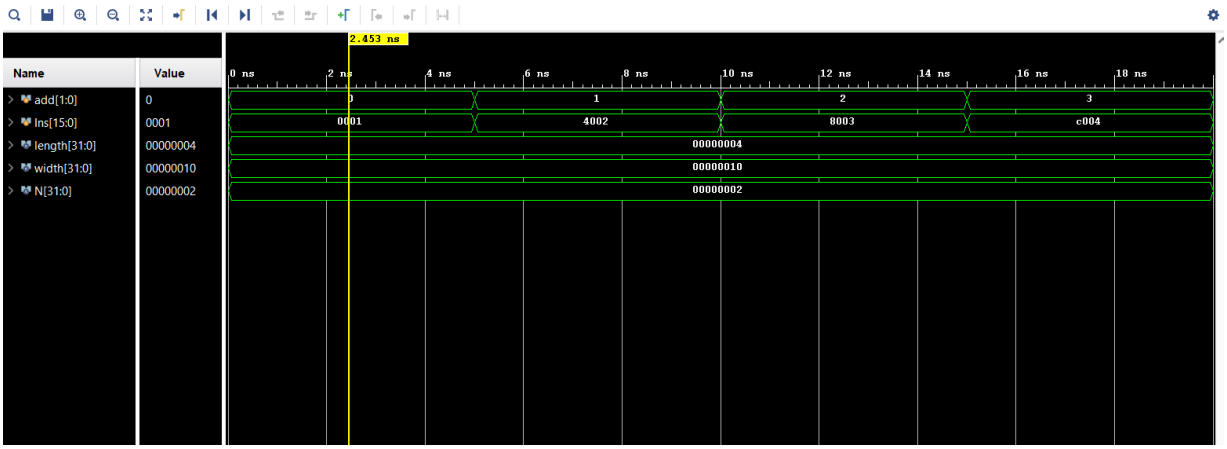
TESTBENCH CODE:

```
18      // Additional Comments:
19      //
20      //////////////////////////////////////////////////////////////////////////////////
21
22      module Insmem_tb;
23          parameter length = 4;
24          parameter width  = 16;
25          parameter N      = 2;
26
27          logic [N-1:0] add;
28          logic [width-1:0] Ins;
29
30          InsMemory #(
31              .length(length),
32              .width(width),
33              .N(N)
34          ) dut (
35              .add(add),
36              .Ins(Ins)
37          );
38
39          initial begin
40              add = 0;
41              #5;
42              add = 1;
43              #5;
44              add = 2;
45              #5;
46              add = 3;
47              #5;
48              $stop;
49          end
50      endmodule
51
```

SCHEMATICS:



SIMULATIONS:



……

Lab Task 4: Program Counter Design

The program counter is a fundamental part of every microprocessor. Its role is to generate addresses for the instruction memory, enabling the processor to fetch instructions from the correct locations. Essentially, the program counter functions as a simple accumulator made up of a register and an adder. It must be capable of sequentially addressing all instructions stored in the instruction memory.

Create a program counter to address instructions from the instruction memory and verify its functionality with a testbench. The program counter shall be initialized with zero and increment on each lock cycle.

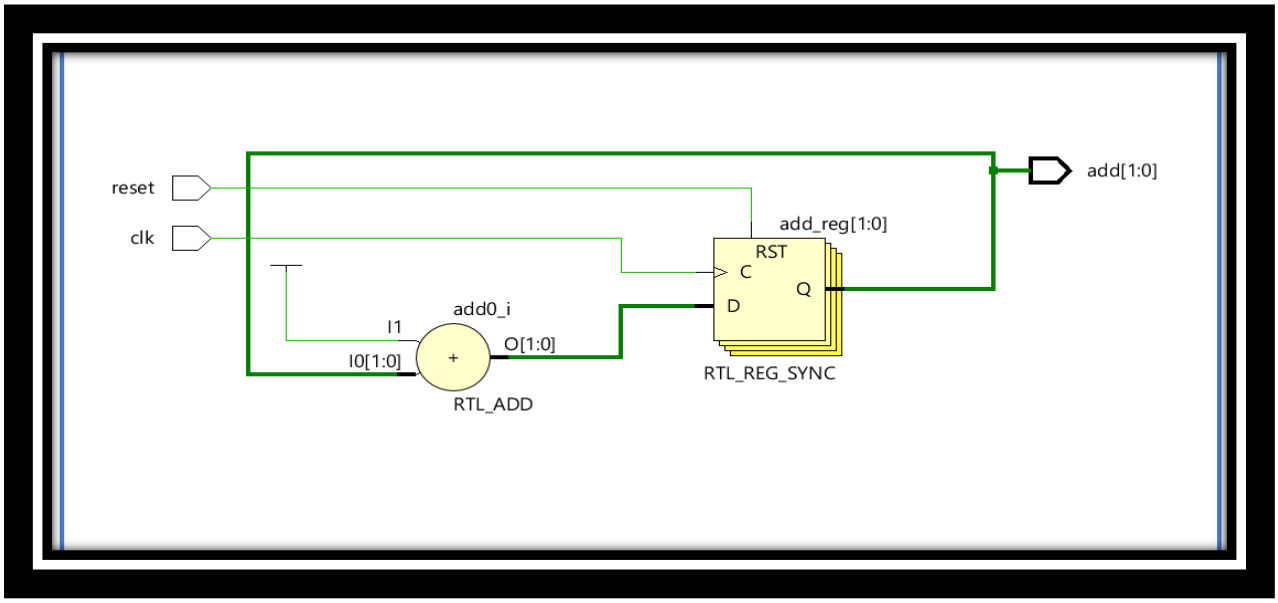
CODE:

```
module PC #(parameter N = 2)
(
    input logic clk,
    input logic reset,
    output logic [N-1:0] add
);
    always_ff@(posedge clk)
    begin
    if(reset == 1'b1)
        add <= 0;
    else
        add <= add+1 ;
    end
endmodule
```

TESTBENCH CODE:

```
23  module PC_tb;
24      parameter N =2;
25      logic clk;
26      logic reset;
27      logic [N-1:0] add;
28
29      PC #(
30      .N(N)
31      )
32      dut(
33      .clk(clk),
34      .reset(reset),
35      .add(add)
36      );
37
38       always
39       begin
40       #5
41       clk = ~clk;
42       end
43
44       initial
45       begin
46          clk   = 0;
47          reset = 1;
48          #10;
49          reset = 0;
50          #50;
51       $stop;
52       end
53  endmodule
54
```

SCHEMATICS:



SIMULATIONS:

CONCLUSION:

The custom processor was successfully implemented on Xilinx Vivado using separate modules for ALU, Register Memory, Instruction Memory, and Program Counter. Each module was designed, simulated, and verified independently, then integrated into the top-level processor architecture. The modular approach improved design clarity, debugging, and reusability. Behavioral simulations confirmed correct execution of arithmetic, logical, memory, and control operations. The processor followed a stepwise execution cycle where the Program Counter fetched instructions from Instruction Memory, data was handled through Register Memory, and computations were processed in the ALU. The design validated the concept of modular digital systems and highlighted the importance of simulation for functional verification. This project provided a strong foundation for understanding processor design and can be extended with more complex instructions in the future.