Design and Implementation of a Single-Cycle RISC-V Processor

Table of Contents

List of Figures

List of Tables

1. Abstract

This processor explores the design and implementation of a RISC-V processor using Verilog HDL, emphasizing its benefits for both academic and commercial applications. The RISC-V instruction set architecture (ISA) is an open standard celebrated for its simplicity, scalability, and flexibility. The project utilizes the RV32I base instruction set to develop a 32-bit RISC-V processor with a single-stage pipelined architecture: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). Rigorous testing and simulation verified the functionality and pipeline behavior.

2. Introduction

RISC V is fundamentally designed with a modular approach. It includes 47 base instructions and allows for modular adjustments based on design requirements. Unlike other ISAs, RISC-V does not prescribe how a design must be implemented or which subsets it must contain, providing unparalleled flexibility. Consequently, many RISC-V computers implement compact extensions to reduce power consumption, code size, and memory use. This adaptability makes RISC-V an attractive choice for both academic and commercial use, bridging theoretical concepts with practical hardware applications.

Building a CPU from scratch provides hands-on experience in computer architecture and digital design. Utilizing the RV32I base instruction set, we developed a 32-bit RISC-V processor with a five-stage pipelined architecture: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). Rigorous testing and simulation verified the functionality and pipeline behavior, ensuring that each stage operates correctly and efficiently. This project not only demonstrates the viability of designing a scalable and efficient RISC-V processor but also enables further research into hardware security, power efficiency, and performance.

3. Problem Statement

The challenge is to design and implement a processor that can efficiently execute a wide range of instructions while maintaining high performance and low power consumption.

Table 1 Comparison of Instruction Set Architectures

ISA	Silicon	Architecture	Commercial IP	Instruction	Open-Source
x86	Yes	No	No	No	No
ARM	Yes	Yes	Yes	No	No
MIPS	Yes	Yes	Limited	Limited	No
RISC-V	Yes	Yes	Yes	Yes	Yes

By comparing all the mainstream ISA's in table 1 that are available in the market. The RISC V has all the perks of designing a robust and efficient RISC V processor for different ASIC applications.

4. Objectives

- Gain an understanding of RISC-V architecture fundamentals.
- Implement a single-stage pipelined RISC-V processor using Verilog.
- Verify functionality using test benches and simulation tools.

5. RISC V System Overview

The RISC-V processor employs a single stage to efficiently execute instructions. Each stage-Instruction Fetch, Instruction Decode, Execute, Memory Access, and Write Back-is modularly designed to enable parallelism and enhance performance. The RV32I base instruction set supports arithmetic, logic, load/store, and control flow operations.

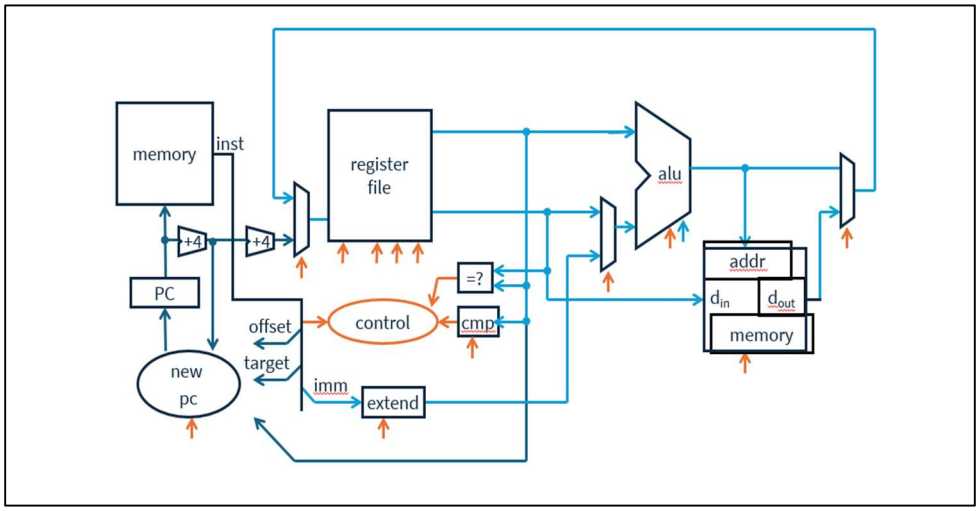

Figure 1 RISC V Single Cycle Processor

5.1.RISC V Execution cycle

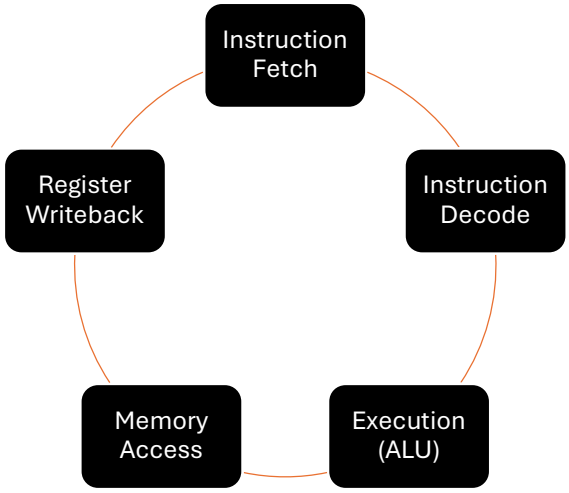

Figure 2 RISC V Execution Cycle

The execution cycle of a RISC-V processor follows a structured flow to fetch, decode, and execute instructions efficiently. For the RV32I base instruction set (32-bit architecture), the execution cycle consists of five primary stages within a pipelined or single-cycle design.

5.2.Pipeline Stages

5.2.1. Instruction Fetch (IF)

The purpose of the Instruction Fetch (IF), Fetches the next instruction from the instruction memory using the Program Counter (PC). The current PC is sent to instruction memory to retrieve the instruction. The PC is updated (incremented by 4 for a 32-bit word) to point to the next instruction.

Key Components:

- Program Counter (PC): Holds the current instruction address.
- Instruction Memory: Provides the instruction corresponding to the PC.
- Adder: Calculates PC + 4 for sequential execution.

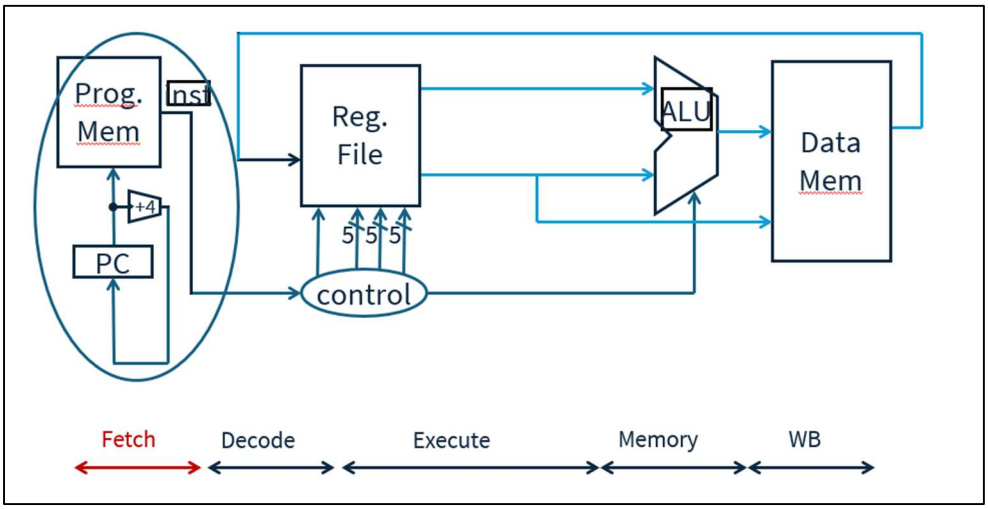

Figure 3 Instruction Fetch Block

5.2.2. Instruction Decode (ID)

The purpose of the Instruction Decode (ID), Decodes the (IF) fetched instruction and generates necessary control signals. Decodes the opcode and identifies the instruction type (R-type, I-type, etc.). Reads register operands using the Register File. Generates control signals for the subsequent pipeline stages.

Key Components:

- Register File: Reads source operands (rs1, rs2) for R-type instructions.
- Immediate Generator: Extracts and extends immediate values from the instruction.
- Control Unit: Generates control signals (ALU_Control, Regwrite_Control).

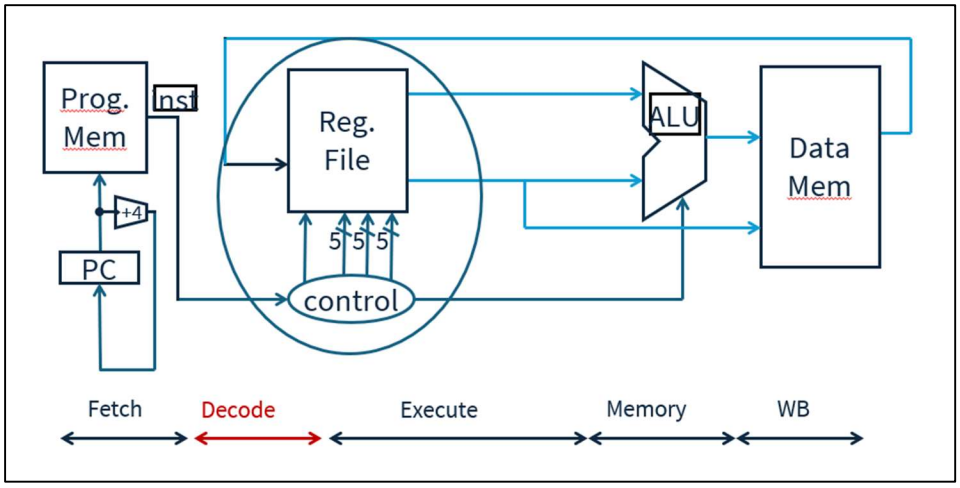

Figure 4 Instruction Decode Block

5.2.3. Execute (EX):

Performs arithmetic/logical operations or computes memory addresses. Uses the ALU to perform operations like ADD, SUB, AND, OR, etc., based on control signals. Computers branch target addresses and evaluates branch conditions.

Key Components:

- ALU (Arithmetic Logic Unit): Executes arithmetic and logical operations.
- ALU Control: Decodes ALU operation type based on instruction.

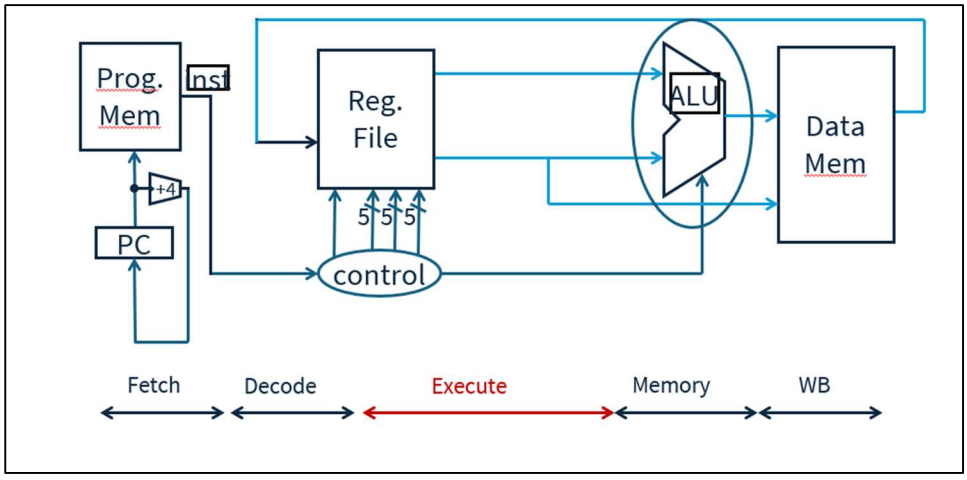

Figure 5 Execute Block

5.2.4. Memory Access (MEM)

Reads from or writes to Data Memory based on the instruction. Reads data from memory using the computed address and writes data from the register to the computed memory address.

Key Components:

- Data Memory: Stores and retrieves data.

- Address Bus: Receives the memory address from the Execute stage.
- Control Signals: Determines if the operation is a read or write.

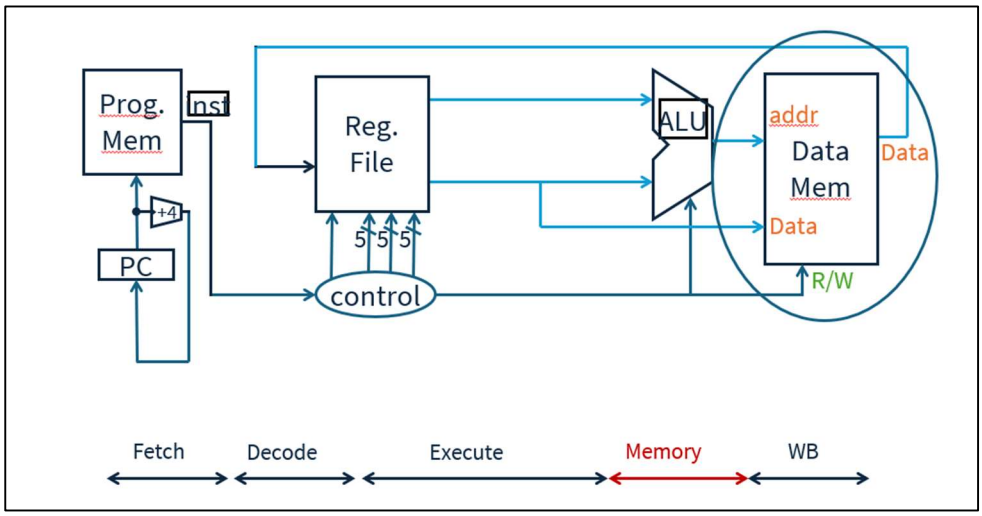

Figure 6 Memory Block

5.2.5. Write Back (WB):

Writes the result of the operation back to the Register File. The load instruction Writes the data fetched from memory back to the register and the ALU instruction writes the ALU result back to the destination register.

Key Components:

- Register File: Updates the destination register with the result.

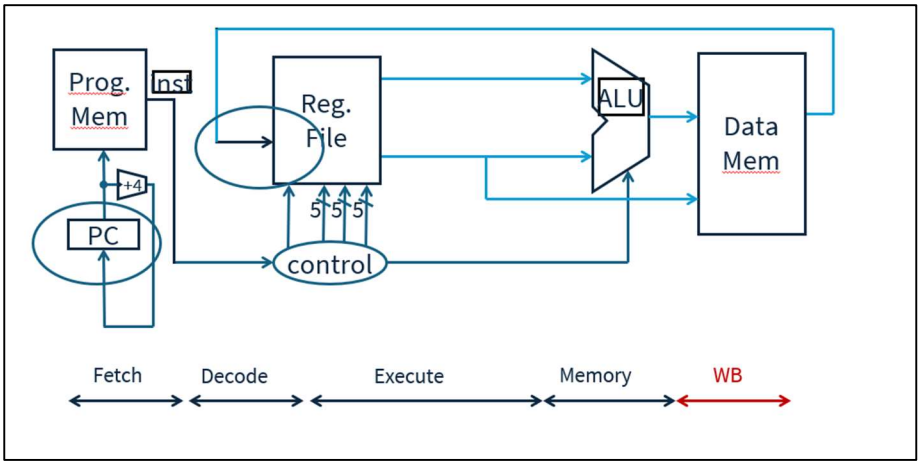

Figure 7 WriteBack Block

6. RISC V ISA

The **RISC-V ISA** (Reduced Instruction Set Computing V) is an open, modular, and extensible instruction set architecture that supports various applications ranging from embedded systems to supercomputers. The ISA is divided into base and optional extensions, ensuring scalability and flexibility. The base ISA for 32-bit systems is **RV32I**, which includes 47 core instructions.

6.1.RV32I Instruction Formats

Instructions in RISC-V are **32 bits** wide and categorized into the following formats in figure 8.

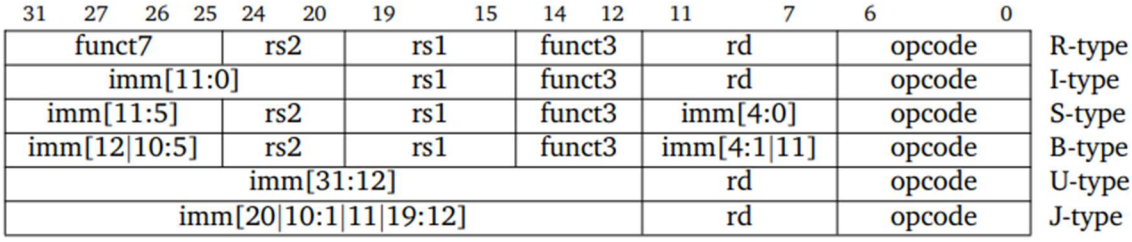

Figure 8 RISC V ISA Format Chat

Here, we are mainly focused on the R type of the ISA. Here is the elaborated view of the R-type in table 2.

Table 2 RISC V R-Type ISA

Instruction	Description	Opcode	Funct3	Funct7
add	Add rs1 and rs2	0110011	000	0000000
sub	Subtract rs2 from rs1	0110011	000	0100000
sll	Shift left logical	0110011	001	0000000
slt	Set less than (signed)	0110011	010	0000000
sltu	Set less than (unsigned)	010011	011	0000000
xor	Bitwise XOR ofrs1 and rs2	0110011	100	0000000
srl	Shift right logical	0110011	101	0000000
sra	Shift right arithmetic	0110011	101	0100000
or	Bitwise OR of rs1 and rs2	0110011	110	0000000
and	Bitwise AND of rs1 and rs2	0110011	111	0000000

The **RISC-V Instruction Set Architecture (ISA)** stands out for its modularity, and openness, making it a versatile choice for a wide range of applications. Unlike proprietary ISAs like **x86** (Intel/AMD) and **ARM**, RISC-V allows free use and customization without licensing costs. Its fixed instruction size and reduced complexity enable better power efficiency, making it ideal for embedded systems, IoT devices, and academic research. In contrast, x86, with its complex instruction set, dominates desktops and servers, focusing on high performance at the cost of power consumption. ARM, known for its energy-efficient RISC design, leads the mobile and IoT markets but remains under proprietary control. RISC-V's modular design, combining a simple base ISA

with optional extensions offers scalability from low-power devices to high-performance computing, positioning it as a growing competitor in modern computing markets. In table 3 the detailed comparison is tabulated.

Table 3 RISC V ISA and Processor Differentiation

Aspect	**RISC-V ISA**	**RISC-V Processor**
Definition	Specification of instructions and behavior.	Physical or simulated implementation of the ISA.
Level of Abstraction	High-level, abstract design.	Low-level, concrete hardware.
Focus	What instructions are supported.	How instructions are executed.
Implementation	No direct physical implementation.	Implemented in hardware or software.
Customization	Extensions define optional features.	Microarchitecture design can vary widely.
Example	Instruction add x1, x2, x3	A processor pipeline that executes add efficiently.

7. Proposed Design Overview

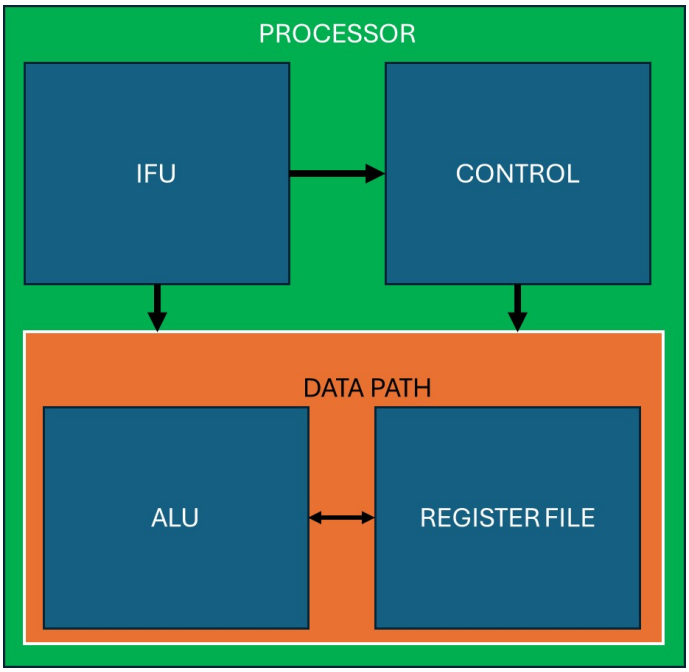

Figure 9 Proposed design overview

The design incorporates several main modules, such as ALU, instruction memory, data memory, register file, and control unit. The functionality of these modules was simulated and verified using AMD VIVADO software.

8. Developer Guide

8.1.Introduction

This guide provides a detailed roadmap for developers to implement a RISC-V processor using Verilog HDL. It covers the design process, key modules, testing, and verification.

Prerequisites

- Basic understanding of digital design and computer architecture.
- Familiarity with Verilog HDL.
- Access to simulation tools like AMD Vivado
- Understanding of the RISC-V ISA.

8.2. Implementation Steps

1. Design Modules: Implement each module individually using Verilog.
2. Simulate and Verify: Use AMD Vivado to simulate and verify the functionality of each module.
3. Integrate Modules: RTL analyses the schematic using the Vivado
4. Testing: Develop test benches to verify the integrated design.

8.3. Testing and Verification

Simulation Tools

- Used VIVADO for functional verification and synthesis of the processor design. Vivado provides a comprehensive suite of tools for simulation, synthesis, and analysis, helping ensure that the Verilog code is correctly implemented and optimized.

Test Benches

- Develop comprehensive test benches in Verilog to verify the correct execution of instructions and the proper operation of the pipeline. Test benches simulate various scenarios to validate the functionality of individual modules as well as the integrated system.

9. Implementation Overview

9.1.Overall system schematic

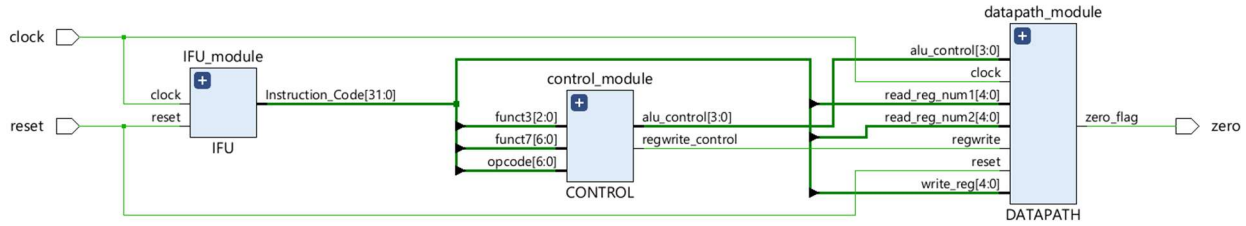

Figure 10 Overall system schematic

9.2.Instruction Fetch Unit (IFU) Module

```
`include "INST_MEM.v"
module IFU(
```

```
    input clock,reset,
    output [31:0] Instruction_Code
);
reg [31:0] PC = 32'b0;  // 32-bit program counter is initialized to zero

    // Initializing the instruction memory block
    INST_MEM instr_mem(PC,reset,Instruction_Code);

    always @(posedge clock, posedge reset)
    begin
        if(reset == 1)  //If reset is one, clear the program counter
        PC <= 0;
        else
        PC <= PC+4;   // Increment program counter on positive clock edge
    end

endmodule
```

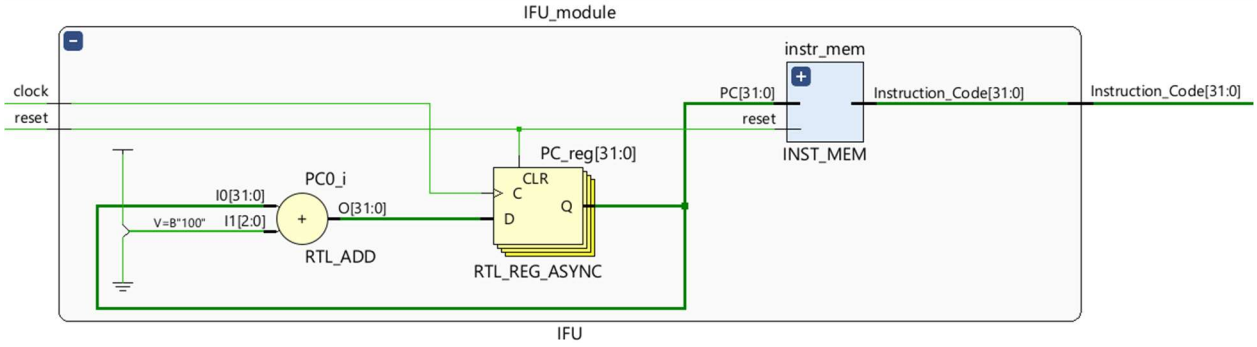

Figure 11 Instruction Fetch Unit module schematic

9.3.Instruction Memory Module

```
module INST_MEM(
    input [31:0] PC,
    input reset,
    output [31:0] Instruction_Code
);
    reg [7:0] Memory [31:0]; // Byte addressable memory with 32 locations

    // Under normal operation (reset = 0), we assign the instr. code, based on PC
    assign Instruction_Code = {Memory[PC+3],Memory[PC+2],Memory[PC+1],Memory[PC]};

    // Initializing memory when reset is one
    always @(reset)
    begin
        if(reset == 1)
        begin
            // Setting 32-bit instruction: add t1, s0,s1 => 0x00940333
            Memory[3] = 8'h00;
            Memory[2] = 8'h94;
            Memory[1] = 8'h03;
            Memory[0] = 8'h33;
            // Setting 32-bit instruction: sub t2, s2, s3 => 0x413903b3
```

```
        Memory[7] = 8'h41;
        Memory[6] = 8'h39;
        Memory[5] = 8'h03;
        Memory[4] = 8'hb3;
        // Setting 32-bit instruction: mul t0, s4, s5 => 0x035a02b3
        Memory[11] = 8'h03;
        Memory[10] = 8'h5a;
        Memory[9] = 8'h02;
        Memory[8] = 8'hb3;
        // Setting 32-bit instruction: xor t3, s6, s7 => 0x017b4e33
        Memory[15] = 8'h01;
        Memory[14] = 8'h7b;
        Memory[13] = 8'h4e;
        Memory[12] = 8'h33;
        // Setting 32-bit instruction: sll t4, s8, s9
        Memory[19] = 8'h01;
        Memory[18] = 8'h9c;
        Memory[17] = 8'h1e;
        Memory[16] = 8'hb3;
        // Setting 32-bit instruction: srl t5, s10, s11
        Memory[23] = 8'h01;
        Memory[22] = 8'hbd;
        Memory[21] = 8'h5f;
        Memory[20] = 8'h33;
        // Setting 32-bit instruction: and t6, a2, a3
        Memory[27] = 8'h00;
        Memory[26] = 8'hd6;
        Memory[25] = 8'h7f;
        Memory[24] = 8'hb3;
        // Setting 32-bit instruction: or a7, a4, a5
        Memory[31] = 8'h00;
        Memory[30] = 8'hf7;
        Memory[29] = 8'h68;
        Memory[28] = 8'hb3;
    end
  end

endmodule
```

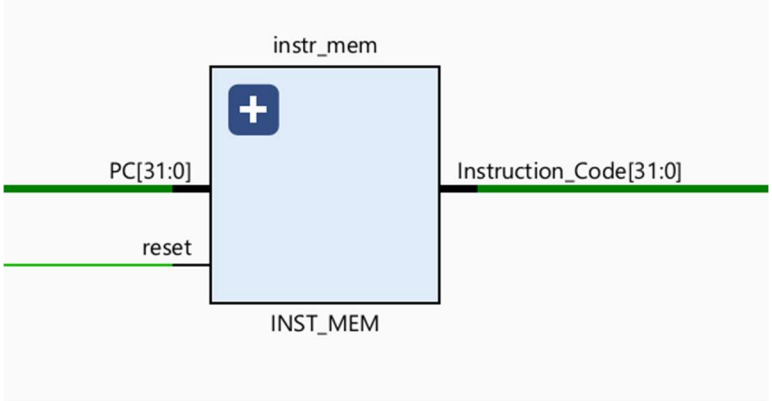

Figure 12 Instruction memory module schematic

9.4.Control Module

```
module CONTROL(
    input [6:0] funct7,
    input [2:0] funct3,
    input [6:0] opcode,
    output reg [3:0] alu_control,
    output reg regwrite_control
);
    always @(funct3 or funct7 or opcode)
    begin
        if (opcode == 7'b0110011) begin // R-type instructions

            regwrite_control = 1;
            // Check the Bianry number before debugging
            case (funct3)
                0: begin
                    if(funct7 == 0)
                    alu_control = 4'b0010; // ADD
                    else if(funct7 == 32)
                    alu_control = 4'b0100; // SUB
                end
                1: alu_control = 4'b0011; // SLL
                2: alu_control = 4'b0110; // MUL
                4: alu_control = 4'b0111; // XOR
                5: alu_control = 4'b0101; // SRL
                6: alu_control = 4'b0001; // OR
                7: alu_control = 4'b0000; // AND
            endcase
        end
    end
endmodule
```

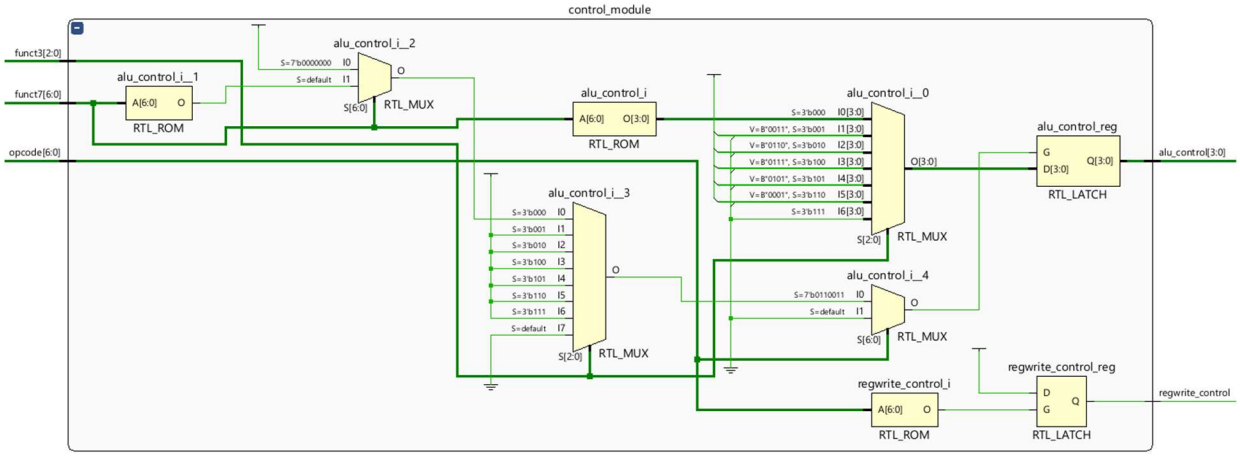

Figure 13 Control module schematic

9.5.Datapath

```
`include "REG_FILE.v"
```

```
`include "ALU.v"
module DATAPATH(
    input [4:0]read_reg_num1,
    input [4:0]read_reg_num2,
    input [4:0]write_reg,
    input [3:0]alu_control,
    input regwrite,
    input clock,
    input reset,
    output zero_flag
);
    // Declaring internal wires that carry data
    wire [31:0]read_data1;
    wire [31:0]read_data2;
    wire [31:0]write_data;
    // Instantiating the register file
    REG_FILE reg_file_module(
    read_reg_num1,
    read_reg_num2,
    write_reg,
    write_data,
    read_data1,
    read_data2,
    regwrite,
    clock,
    reset
    );
    // Instanting ALU
    ALU alu_module(read_data1, read_data2, alu_control, write_data, zero_flag);
endmodule
```

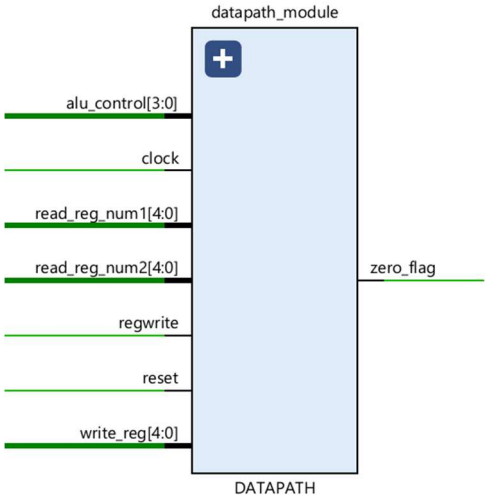

Figure 14 Datapath module schematic

9.6.Register file module

```
module REG_FILE(
    input [4:0] read_reg_num1,
    input [4:0] read_reg_num2,
    input [4:0] write_reg,
    input [31:0] write_data,
    output [31:0] read_data1,
    output [31:0] read_data2,
    input regwrite,
    input clock,
    input reset
);
    reg [31:0] reg_memory [31:0]; // 32 memory locations each 32 bits wide
    integer i=0;
    //  When reset is triggered, we initialize the registers with some values
    always @(posedge reset)
    begin
        reg_memory[0] = 32'h0;
        reg_memory[1] = 32'h1;
        reg_memory[2] = 32'h2;
        reg_memory[3] = 32'h3;
        reg_memory[4] = 32'h4;
        reg_memory[5] = 32'h5;
        reg_memory[6] = 32'h6;
        reg_memory[7] = 32'h7;
        reg_memory[8] = 32'h8;
        reg_memory[9] = 32'h9;
        reg_memory[10] = 32'h10;
        reg_memory[11] = 32'h11;
        reg_memory[12] = 32'h12;
        reg_memory[13] = 32'h13;
        reg_memory[14] = 32'h14;
        reg_memory[15] = 32'h15;
        reg_memory[16] = 32'h16;
        reg_memory[17] = 32'h17;
        reg_memory[18] = 32'h18;
        reg_memory[19] = 32'h19;
        reg_memory[20] = 32'h20;
        reg_memory[21] = 32'h21;
        reg_memory[22] = 32'h22;
        reg_memory[23] = 32'h23;
        reg_memory[24] = 32'h24;
        reg_memory[25] = 32'h25;
                        reg_memory[26] = 32'h26;
        reg_memory[27] = 32'h27;
        reg_memory[28] = 32'h28;
        reg_memory[29] = 32'h29;
        reg_memory[30] = 32'h30;
        reg_memory[31] = 32'h31;

    end
```

```
   // The register file will always output the values corresponding to read register numbers
   // It is independent of any other signal
   assign read_data1 = reg_memory[read_reg_num1];
   assign read_data2 = reg_memory[read_reg_num2];

   // If clock edge is positive and regwrite is 1, we write data to specified register
   always @(posedge clock)
   begin

      if (regwrite) begin
         reg_memory[write_reg] = write_data;
      end
   end
endmodule
```

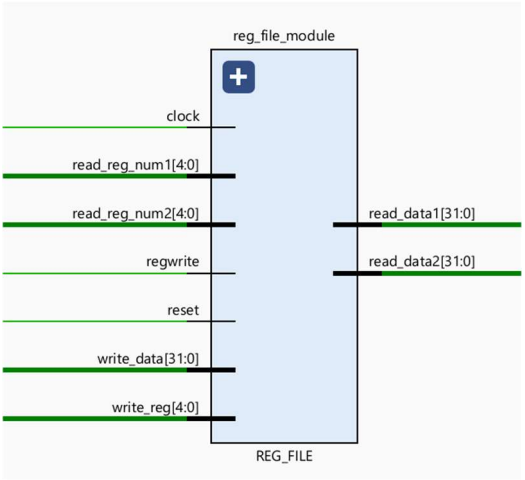

Figure 15 Register file module schematic

9.7.ALU Module

```
module ALU (
   input [31:0] in1,in2,
   input[3:0] alu_control,
   output reg [31:0] alu_result,
   output reg zero_flag
);
   /* ALU Control lines | Function
----------------------------
      0000   Bitwise-AND
      0001   Bitwise-OR
      0010          Add (A+B)
      0100          Subtract (A-B)
      1000          Set on less than
      0011   Shift left logical
      0101   Shift right logical
      0110   Multiply
      0111   Bitwise-XOR
```

```
*/
   always @(*)
   begin
      // Operating based on control input
      case(alu_control)

      4'b0000: alu_result = in1&in2;
      4'b0001: alu_result = in1|in2;
      4'b0010: alu_result = in1+in2;
      4'b0100: alu_result = in1-in2;
      4'b1000: begin
         if(in1<in2)
         alu_result = 1;
         else
         alu_result = 0;
      end
      4'b0011: alu_result = in1<<in2;
      4'b0101: alu_result = in1>>in2;
      4'b0110: alu_result = in1*in2;
      4'b0111: alu_result = in1^in2;

      endcase

      // Setting Zero_flag if ALU_result is zero
      if (alu_result == 0)
         zero_flag = 1'b1;
      else
         zero_flag = 1'b0;

   end
endmodule
```

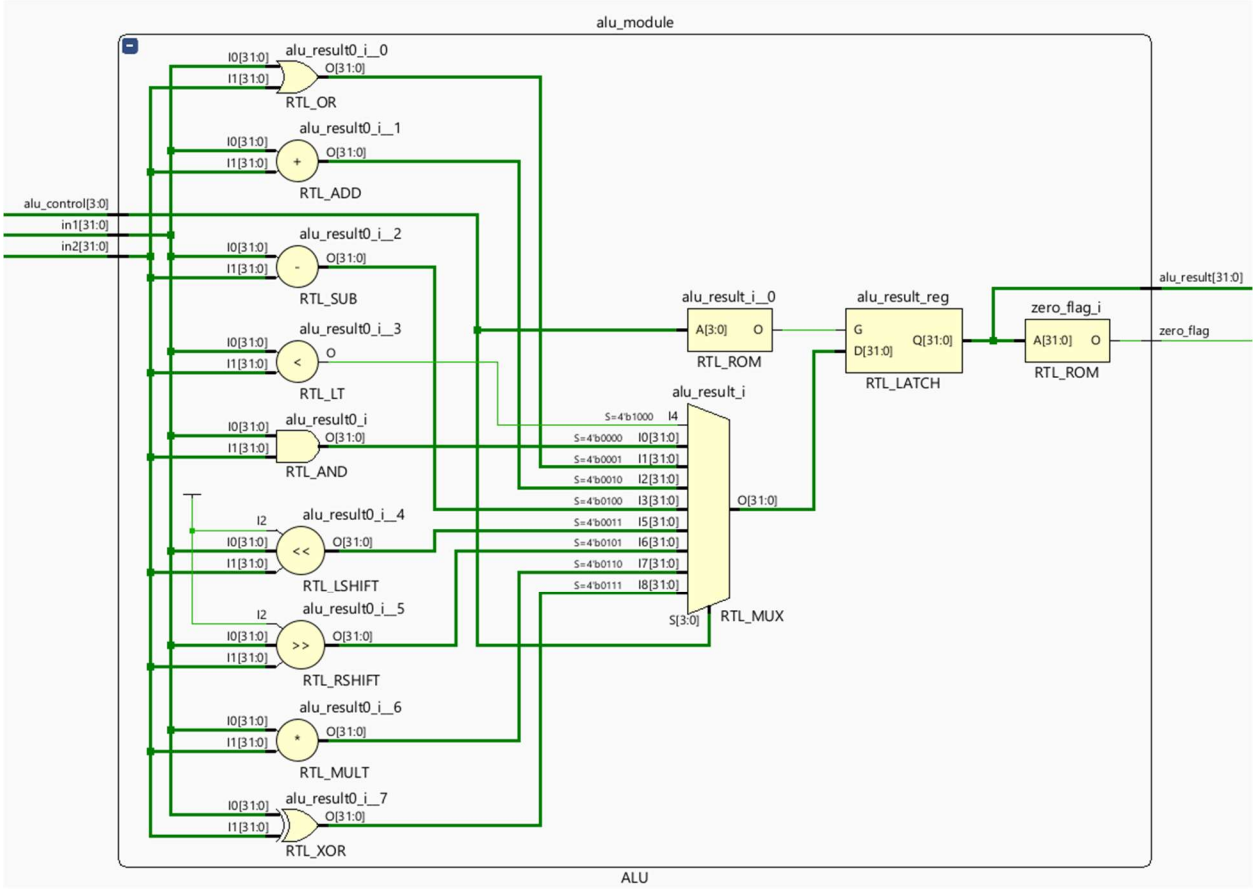

Figure 16 ALU module schematic

10.Conclusion

The RISC-V processor design using Verilog HDL successfully demonstrates the viability of creating a scalable, efficient, and modular CPU core. By leveraging the open and extensible RISC-V ISA, this design not only highlights the flexibility of the instruction set but also underscores the potential for innovation in hardware development. Through robust verification methods and rigorous simulation, the functionality and efficiency of the processor have been validated, ensuring its applicability in real-world scenarios. This work serves as a solid foundation for further advancements in computer architecture, including pipeline optimization, hazard management, and hardware implementation on FPGA or ASIC platforms. Moreover, this guide provides a clear and comprehensive roadmap for developers, enabling them to implement, customize, and expand upon the RISC-V processor design. By following these steps, developers can create CPU cores that are not only efficient and scalable but also adaptable for a wide range of applications from embedded systems to high-performance computing paving the way for future research and innovation in hardware design.

11.Future Work

- Enhanced Testing: Expand test cases to cover edge scenarios.

- Pipeline Optimization: Improve hazard resolution and branch prediction.
- FPGA Implementation: Deploy and test the design on FPGA platforms.

Did you meet your milestone stated in the initial report or the midterm report (if you pivoted)? If not, what made you fail to achieve it?

We have met the milestones outlined in our initial and midterm reports. The primary objectives, such as understanding RISC-V architecture, implementing a single stage pipelined processor using Verilog, and verifying functionality through simulation, were all achieved. Rigorous testing and simulation validated the processor's functionality and pipeline behavior, ensuring that each stage operated correctly and efficiently. However, the processor testbench is not yet completed, as it requires additional time for comprehensive verification and testing.

What are some things that you wish you had known so that you could potentially complete the project more smoothly? If there is none, feel free to say so.

One aspect we wish we had known earlier was the intricacies of handling control hazards and synchronization issues within the pipeline stages. Early knowledge of more advanced techniques for branch prediction and pipeline stalling could have expedited our debugging process and enhanced pipeline performance. Additionally, a deeper understanding of FPGA resource constraints and optimization strategies would have been beneficial as we transitioned our design from simulation to hardware prototyping. Overall, while the project was a success, these insights could have further streamlined our development process and potentially improved performance and reliability.

References

[1] Volokitin, Valentin & Kozinov, Evgeny & Kustikova, Valentina & Liniov, Alexey & Meyerov, Iosif. (2023). Case Study for Running Memory-Bound Kernels on RISC-V CPUs. 10.48550/arXiv.2305.09266.

[2] *Implement 32-bit RISC-V Architecture Processor using Verilog HDL*. (2021, November 16). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/9651130

[3] *Retargeting the MIPS-II CPU core to the RISC-V architecture*. (2019, June 1). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/8787018/

[4] Single cycle RISC-V micro architecture processor and its FPGA prototype. (2017, December 1). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/8303926/similar#similar

[5] https://riscv.org/specifications/ratified/