ISSN: 2584-0495 Vol. 3, Issue 6, pp. 1663-1683

International Journal of Microsystems and IoT

ISSN: (Online) Journal homepage: https://www.ijmit.org

Design of RISC- V Processor using Verilog@HDL

Pratyush Pranjal, Vansh Tak, Aryan Singh and Vijay Nath

Cite as: Pranjal, P., Tak, V., Singh, A., & Nath, V. (2025). Design of RISC- V Processor using Verilog @HDL. International Journal of Microsystems and IoT, 3(6), 1663–1683. https://doi.org/10.5281/zenodo.18152801

Published online: 25 June 2025

https://doi.org/10.5281/zenodo.18152801

International Journal of Microsystems and IoT
Vol.3, Issue 6, pp.1663-1683; DOI: https://doi.org/10.5281/zenodo.18152801

Design of RISC- V Processor using Verilog @HDL

Pratyush Pranjal, Vansh Tak, Aryan Singh and Vijay Nath

Department of Electronics and Communication Engineering, Birla Institute of Technology, Mesra, Ranchi, India

ABSTRACT
This paper presents the design and implementation of a RISC-V (Reduced Instruction Set Computer - Version 5) processor using Verilog hardware description language and the Vivado 2024.2 design suite. The RISC-V architecture, known for its simplicity, modularity, and open-source nature, serves as an ideal platform for academic research and custom processor development. The processor design follows a five-stage pipelined architecture, incorporating Instruction Fetch, Instruction Decode, Execute, Memory Access, and Write Back stages. Each module was coded in Verilog and synthesized using Vivado 2024.2 software, ensuring compatibility with FPGA deployment through Vitis 2024.2 software. The implementation emphasizes efficient data path control, hazard detection, and forwarding mechanisms to maintain instruction throughput and pipeline efficiency. Simulation and verification were conducted to ensure functional accuracy and timing performance, using testbenches that replicate typical instruction sequences. Results indicate that the design meets the functional requirements and is suitable for educational use and further extension into more complex systems.

Keywords

Arithmetic Logic Unit (ALU), Control Unit (CU), Instruction Fetch Unit (IFU), Register File (RF), Datapath (DP).

I. INTRODUCTION

RISC-V is an open-source instruction set architecture (ISA) that is reshaping the future of computing. First introduced by researchers at the University of California, Berkeley in 2010, RISC-V was created to be simple, modular, and efficient, making it suitable for a broad spectrum of applications, from tiny embedded systems to powerful data centres.

What makes RISC-V truly stand out is its openness. Unlike proprietary ISAs like x86 from Intel or ARM, RISC-V is available under a permissive open license. This allows developers, whether from academia, industry, or the maker community, to freely implement, modify, and distribute RISC-V designs without worrying about licensing fees or legal barriers. As a result, RISC-V has sparked widespread interest and innovation around the world.

Adhering to the Reduced Instruction Set Computing (RISC) principles, RISC-V emphasizes a minimal yet highly effective set of instructions. The base set for 32-bit systems, known as **RV32I**, includes only the most essential operations such as arithmetic, logic, control flow, and memory access. However, the architecture is designed to be extensible. Designers can add optional modules like:

1. **M**: Integer multiplication and division
2. **A**: Atomic instructions
3. **F** and **D**: Single and double precision floating-point
4. **C**: Compressed instructions to reduce code size
5. **V**: Vector operations for parallel processing

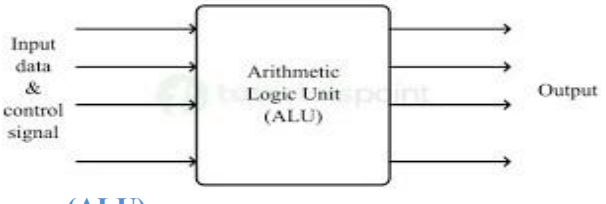
This flexibility means RISC-V cores can be customized for specific tasks, optimizing both performance and power usage. A typical RISC-V processor is composed of several key units:

1. **Instruction Fetch Unit**: Loads instructions from memory using the program counter.
2. **Instruction Decode Unit**: Decodes the fetched instruction and identifies the operation and operands.
3. **Register File**: A set of 32 general-purpose registers for quick data storage and retrieval.
4. **ALU (Arithmetic Logic Unit)**: Executes arithmetic and logical computations.
5. **Control Unit**: Generates control signals to coordinate the processor's internal operations.
6. **Memory Interface**: Manages data transfer between the processor and external memory.

More advanced RISC-V CPUs can incorporate features like pipelining for improved instruction throughput, out-of-order execution for performance gains, and even multicore configurations for parallel processing. Despite such enhancements, the base ISA remains clean, easy to understand, and ideal for teaching computer architecture.
The RISC-V ecosystem continues to expand rapidly. A wide range of tools—including compilers, debuggers, simulators, and operating systems—are now available. Popular operating systems like **Linux** and **Free RTOS** already support RISC-V, and community-driven projects are extending compatibility further every day.

II. THE BASIC COMPONENTS OF A RISC-V PROCESSOR

1. The Design of an Arithmetic Logic Unit



(ALU)

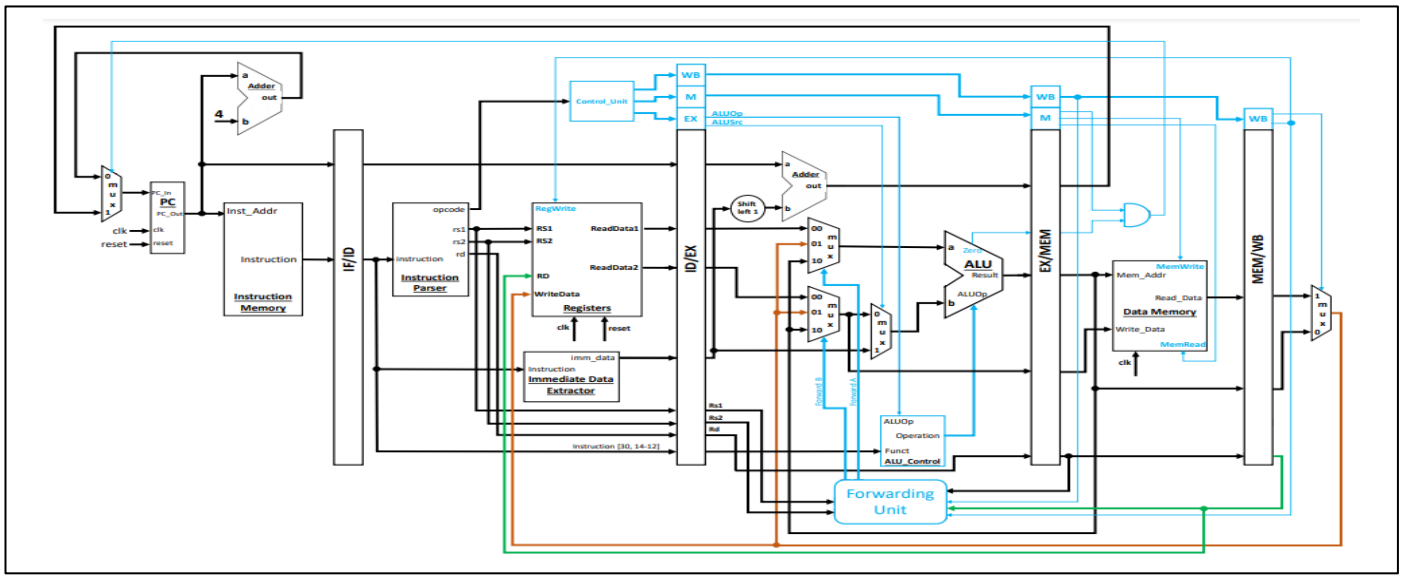


Fig 1. A complex overview showcasing the entire architecture of a basic RISC-V Processor

The **Arithmetic Logic Unit (ALU)** is one of the most essential components within a computer's **Central Processing Unit (CPU)**. It handles the execution of both arithmetic and logical operations, forming the backbone of all computations carried out by the processor. The efficiency and speed of a CPU are heavily influenced by how well the ALU is designed and implemented.

An ALU is engineered to perform a variety of basic operations, such as:

1. **Arithmetic operations**: including addition and subtraction
2. **Logical operations**: such as bitwise AND, OR, XOR, and NOT
3. **Shifting operations**: including left and right bit shifts
4. **Comparisons**: to evaluate relationships like equality or magnitude

To support these functions, a typical ALU consists of the following key components:

Main Components of the ALU

1. **Operand Inputs**
   The ALU accepts two binary input values, often fetched from the CPU's register file. These values are the operands upon which the operation will be performed.
2. **Operation Decoder**
   This internal logic interprets the **opcode** (operation code) provided by the control unit. Based on the opcode, the decoder activates specific circuits within the ALU to carry out the required task, whether it's an addition, a logical operation, or a comparison.
3. **Arithmetic Unit**
   This section performs all arithmetic calculations. A common implementation is the **ripple-carry adder**, which can add two binary numbers. With two's complement logic, it can also handle subtraction. For faster computation, some ALUs use more advanced adders like **carry-lookahead** or **carry-select** adders.
4. **Logic Unit**
   Responsible for executing bit-level logical functions, this unit can carry out operations such as AND, OR, XOR, and inversion. Each bit of the input is processed independently according to the operation selected.
5. **Multiplexer System (MUX)**
   Since the ALU supports multiple operations, **multiplexers** are used to select which result to output based on control signals. The correct operation result is routed through to the output depending on the current instruction.
6. **Status Flags and Outputs**
   Along with the result, the ALU generates **status flags** that reflect the outcome of the operation. These typically include:
   1. **Zero (Z)**: Set if the result is zero
   2. **Carry (C)**: Indicates a carry out in addition or a borrow in subtraction
   3. **Overflow (O)**: Set if an arithmetic overflow occurs
   4. **Negative (N)**: Indicates a negative result in signed operations

2. How to implement ALU

Step 1: Define Module I/O Ports

We start by identifying what an ALU needs:

1. Two 32-bit inputs: A and B (operands)
2. A 4-bit control signal: ALUControl to select the operation
3. One 32-bit output: Result
4. One flag output: Zero

Step 2: Choose Supported Operations

We design the ALU to support both **standard** and a few **custom** operations.

Step 3: "Combinational always" @(*) Block

We use an always @(*) block to create a combinational (non-clocked) logic block. The operation performed is selected using a case statement on ALUControl.

Step 4: Generate the Zero Flag
This output is used for branch comparisons like beq (branch if equal). It's 1 if the result of the operation is 0.

3. Verilog Code

ALUControl	Operation	Description
0000	AND	Bitwise AND
0001	OR	Bitwise OR
0010	NAND	Bitwise NAND
0011	NOR	Bitwise NOR
0100	XOR	Bitwise XOR
0101	ADD	Integer Addition
0110	SUB	Integer Subtraction
0111	SLT	Set-on-less-than
1000	SMT	Set-on-more-than
1001	MUL	Multiply (basic)
1010	REMAINDER	Remainder
1011	SLL	Shift left logical
1100	SRL	Shift Right logical
1101	NOT A	Bitwise NOT on A
1110	NOT B	Bitwise NOT on B

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
module ALU (
    input [31:0] A,
    input [31:0] B,
    input [3:0] ALUControl,
    output reg [31:0] Result,
    output Zero
);
    always @(*) begin
        case (ALUControl)
            4'b0000: Result = A & B;                    // AND
            4'b0001: Result = A | B;                    // OR
            4'b0010: Result = ~(A & B);                 // nand
            4'b0011: Result = ~(A | B);                 // nor
            4'b0100: Result = (A ^ B);                  // xor
            4'b0101: Result = A + B;                    // ADD
            4'b0110: Result = A - B;                    // SUB
            4'b0111: Result = (A < B) ? 32'd1 : 32'd0;  // SLT
            4'b1000: Result = (A > B) ? 32'd1 : 32'd0;  // SMT
            4'b1001: Result = (A * B);                  // mul
            4'b1010: Result = (A / B);                  // remainder
            4'b1011: Result = (A << B);                 // SLL
            4'b1100: Result = (A >> B);                 // SRL
            4'b1101: Result = ( ~A);                    // not A
            4'b1110: Result = ( ~B);                    // not B
            default: Result = 32'd0;
        endcase
    end
    assign Zero = (Result == 32'd0) ? 1'b1 : 1'b0;
endmodule
```

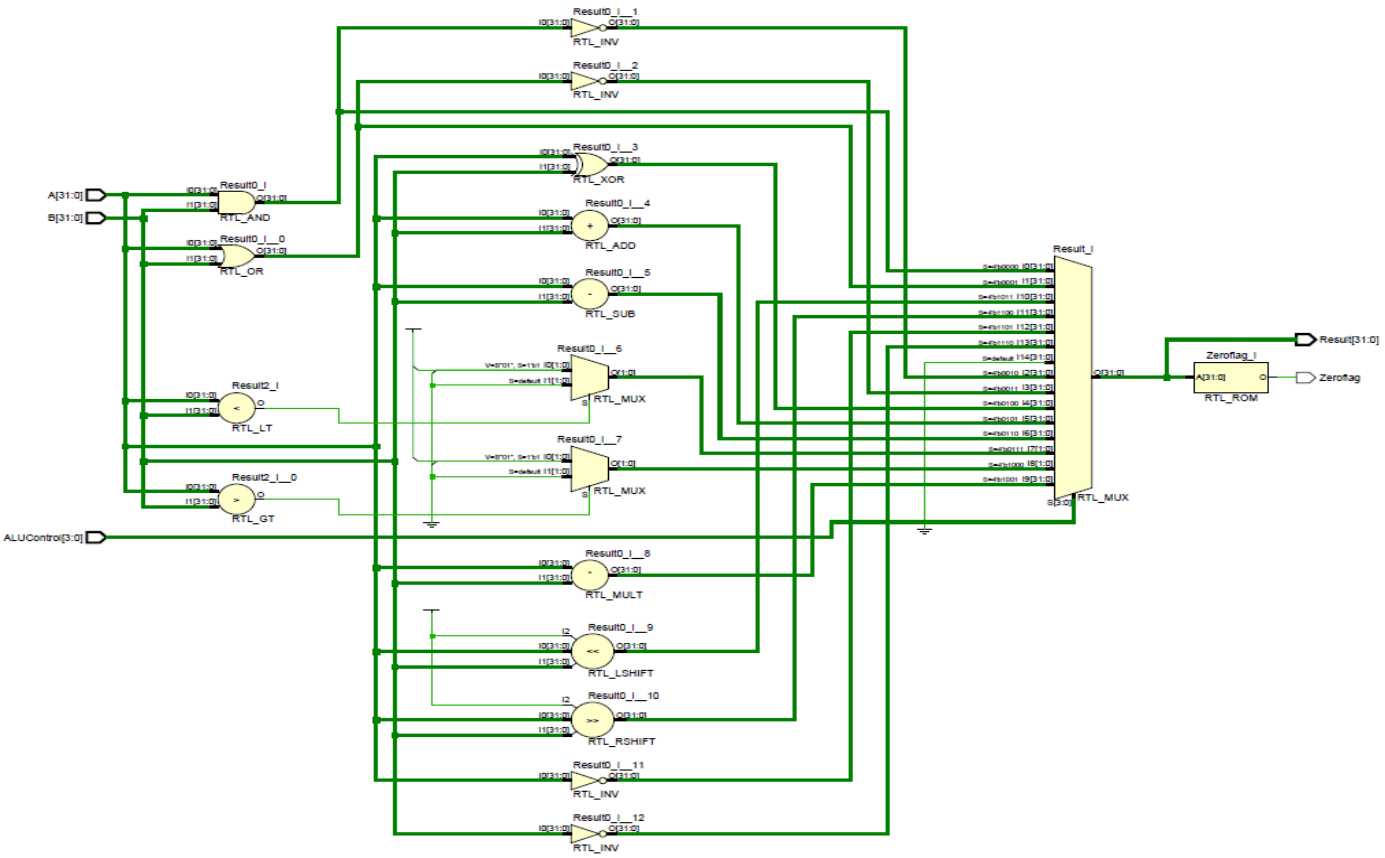


Fig.1.B.1. Design ALU (internal) implementation using Vivado 2024.2

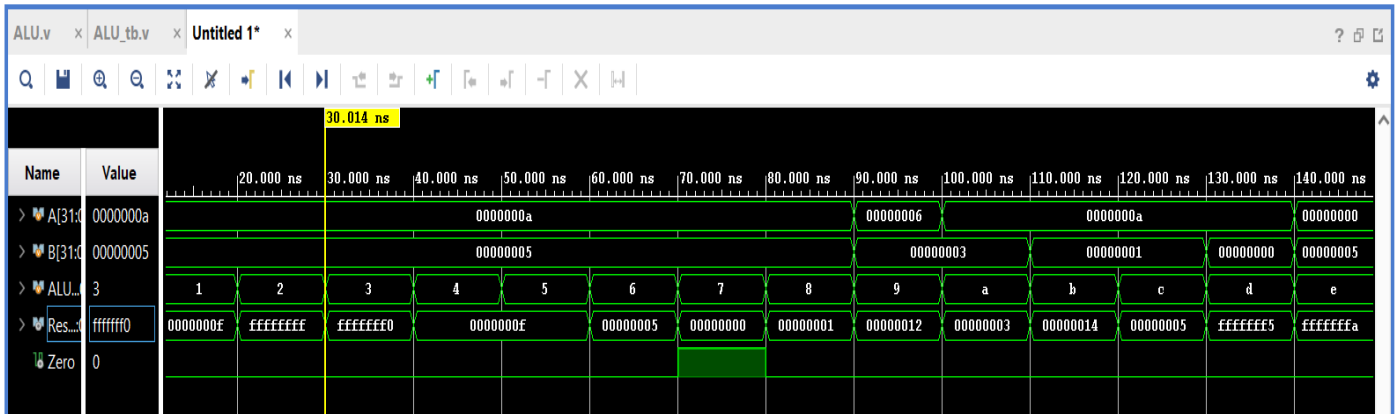


Fig.1.B.2. Behavioral Simulation involving all assigned operations using Vivado 2024.2

The displayed waveform illustrates a behavioral simulation of a Verilog-based Arithmetic Logic Unit (ALU). It shows key signals including the inputs A and B, a 4-bit ALU_Control signal used to select specific ALU operations, and outputs Result and Zero. Initially, A is set to 0xA and B to 0x5, remaining unchanged for several clock cycles while ALU_Control transitions from 3 to E (hex), each representing different ALU functions such as logical operations, addition, subtraction, and shifts. The Result output accurately reflects the expected values for each operation—e.g., when ALU_Control is 6 (ADD), the result is 0xF; for 7 (SUB), it yields 0x5. Around the 90 ns mark, the inputs A and B change, prompting corresponding updates in the result, demonstrating the ALU's ability to respond to new data. The Zero output remains deasserted (0) except when the result is zero, confirming the proper functioning of the zero-detection logic. Overall, the simulation validates that the ALU correctly handles a variety of operations based on different control signals.

2. The Design of a Control Unit (CU)

The **Control Unit (CU)** is a fundamental part of a computer's **Central Processing Unit (CPU)**, serving as the command center that manages how instructions are executed. Rather than performing arithmetic or logical operations itself, the CU oversees and coordinates the actions of other components like the **Arithmetic Logic Unit (ALU)**, memory, and I/O systems by issuing control signals. These signals dictate the timing, routing, and operation of data as it moves through the processor.

Types of Control Unit Architectures

The internal design of a Control Unit generally falls into two main categories: **hardwired** and **microprogrammed** control.

1. Hardwired Control Unit

In a **hardwired control** setup, control signals are generated using physical logic gates, flip-flops, and decoders. The pathways for instruction execution are predefined, meaning each instruction is mapped directly to a specific set of signals.

1. **Advantages**: Fast execution due to direct signal generation.
2. **Limitations**: Difficult to modify or extend—any changes in the instruction set require reworking the circuitry.

2. Microprogrammed Control Unit

A **microprogrammed control unit** operates using a series of microinstructions stored in a dedicated memory called the **control store** or **control memory**. Each machine-level instruction corresponds to a sequence of microinstructions (a microprogram) that generate the necessary control signals.

1. **Advantages**: Flexible and easier to update or expand, making it ideal for CPUs with complex instruction sets.
2. **Limitations**: Slightly slower than hardwired control due to memory access overhead.

Core Components of the Control Unit

To effectively manage instruction execution, the Control Unit comprises several key elements:

1. **Instruction Register (IR)**
   Holds the current instruction being processed. The CU reads the **opcode** from this register to determine the operation required.
2. **Control Signal Generator**
   This module produces control signals based on the decoded instruction. These signals activate or deactivate components like registers, buses, and the ALU.
3. **Timing and Sequencing Logic**
   Ensures all steps in the instruction cycle—**fetch**, **decode**, **execute**, and **write-back**—occur in the correct order and are synchronized with the system clock.
4. **Decoders and Encoders**
   Translate instruction fields or binary patterns into specific control actions, enabling the processor to interpret and respond to various instruction formats.
5. **Status Inputs and Feedback Mechanisms**
   The CU receives condition flags (e.g., **Zero**, **Carry**, **Overflow**) from the ALU and other units. This feedback allows the CU to make decisions about conditional branches, loops, and other control flow operations.

A. How to implement CU

Step 1: Inputs and Outputs

Inputs:

1. opcode (7 bits): comes from the instruction and

determines the type of operation.

Outputs:

Control signals needed by other CPU modules:

1. RegWrite – Should we write to a register file?
2. ALUSrc – Should the ALU use an immediate or a register as the second input?
3. MemRead – Are we reading data memory?
4. MemWrite – Are we writing to data memory?
5. MemToReg – Should the value to write back to a register come from memory or the ALU?
6. Branch – is this a branch instruction?
7. ALUOp (2 bits) – used to guide the ALU operation (passed to the ALU Control module)

Step 2: Recognize Opcode Types

Identify which opcodes correspond to which instruction types in **RISC-V**

Step 3: Determine Control Signal Values for Each Instruction Type

Based on what the instruction needs:

1. R-type (0110011)

1. Use two registers (no immediate), ALU does work.
2. Result goes back to a register.
3. No memory read/write, not a branch.

2. I-type (addi, 0010011)

1. One register and an immediate.
2. ALU computes result, stores in register.

3. Load (lw, 0000011)

1. Use base register + offset to get address.
2. Read from memory, store in register.

4. Store (sw, 0100011)

1. Use base register + offset to compute address.
2. Store register value into memory.

5. Branch (beq, 1100011)

1. Compare two registers, branch if equal.

Step 4: Implement With a case Statement

Now that we know what each opcode needs, use a case statement inside an always @(*) block to set the control signals for each opcode.

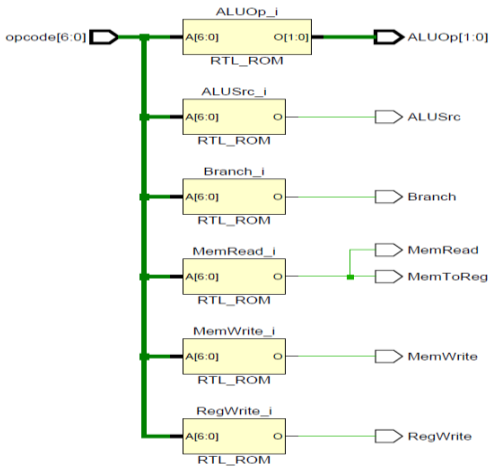


Fig 2.A.1 Design CU (internal) implementation

```
`timescale 1ns / 1ps
//////////////////////////////
module ControlUnit (
    input [6:0] opcode,
    output reg RegWrite,
    output reg ALUSrc,
    output reg MemRead,
    output reg MemWrite,
    output reg MemToReg,
    output reg Branch,
    output reg [1:0] ALUOp
);
 always @(*) begin
     case (opcode)
         7'b0110011: begin // R-type
             RegWrite  = 1;
             ALUSrc    = 0;
             MemRead   = 0;
             MemWrite  = 0;
             MemToReg  = 0;
             Branch    = 0;
             ALUOp     = 2'b10;
         end
         7'b0010011: begin // I-type (addi)
             RegWrite  = 1;
             ALUSrc    = 1;
             MemRead   = 0;
             MemWrite  = 0;
             MemToReg  = 0;
             Branch    = 0;
             ALUOp     = 2'b00;
         end
         7'b0000011: begin // Load (lw)
             RegWrite  = 1;
             ALUSrc    = 1;
             MemRead   = 1;
             MemWrite  = 0;
             MemToReg  = 1;
             Branch    = 0;
             ALUOp     = 2'b00;
         end

           7'b0100011: begin // Store (sw)
               RegWrite  = 0;
               ALUSrc    = 1;
               MemRead   = 0;
               MemWrite  = 1;
               MemToReg  = 0; // Don't care
               Branch    = 0;
               ALUOp     = 2'b00;
           end
           7'b1100011: begin // Branch (beq)
               RegWrite  = 0;
               ALUSrc    = 0;
               MemRead   = 0;
               MemWrite  = 0;
               MemToReg  = 0; // Don't care
               Branch    = 1;
               ALUOp     = 2'b01;
           end
           default: begin // Default - no operation
               RegWrite  = 0;
               ALUSrc    = 0;
               MemRead   = 0;
               MemWrite  = 0;
               MemToReg  = 0;
               Branch    = 0;
               ALUOp     = 2'b00;
           end
       endcase
   end
endmodule
```

B. Verilog Code

```
===== Control Unit Testbench Start =====
Time = 10000 | Opcode = 0110011
RegWrite = 1 | ALUSrc = 0 | MemRead = 0 | MemWrite = 0 | MemToReg = 0 | Branch = 0 | ALUOp = 10

Time = 20000 | Opcode = 0010011
RegWrite = 1 | ALUSrc = 1 | MemRead = 0 | MemWrite = 0 | MemToReg = 0 | Branch = 0 | ALUOp = 00

Time = 30000 | Opcode = 0000011
RegWrite = 1 | ALUSrc = 1 | MemRead = 1 | MemWrite = 0 | MemToReg = 1 | Branch = 0 | ALUOp = 00

Time = 40000 | Opcode = 0100011
RegWrite = 0 | ALUSrc = 1 | MemRead = 0 | MemWrite = 1 | MemToReg = 0 | Branch = 0 | ALUOp = 00

Time = 50000 | Opcode = 1100011
RegWrite = 0 | ALUSrc = 0 | MemRead = 0 | MemWrite = 0 | MemToReg = 0 | Branch = 1 | ALUOp = 01

Time = 60000 | Opcode = 1111111
RegWrite = 0 | ALUSrc = 0 | MemRead = 0 | MemWrite = 0 | MemToReg = 0 | Branch = 0 | ALUOp = 00

===== Control Unit Testbench Complete =====
```

Fig.2.B.2. Behavioral Simulation- The TCL Console showing that different opcodes are being loaded into the memory using Vivado 2024.2

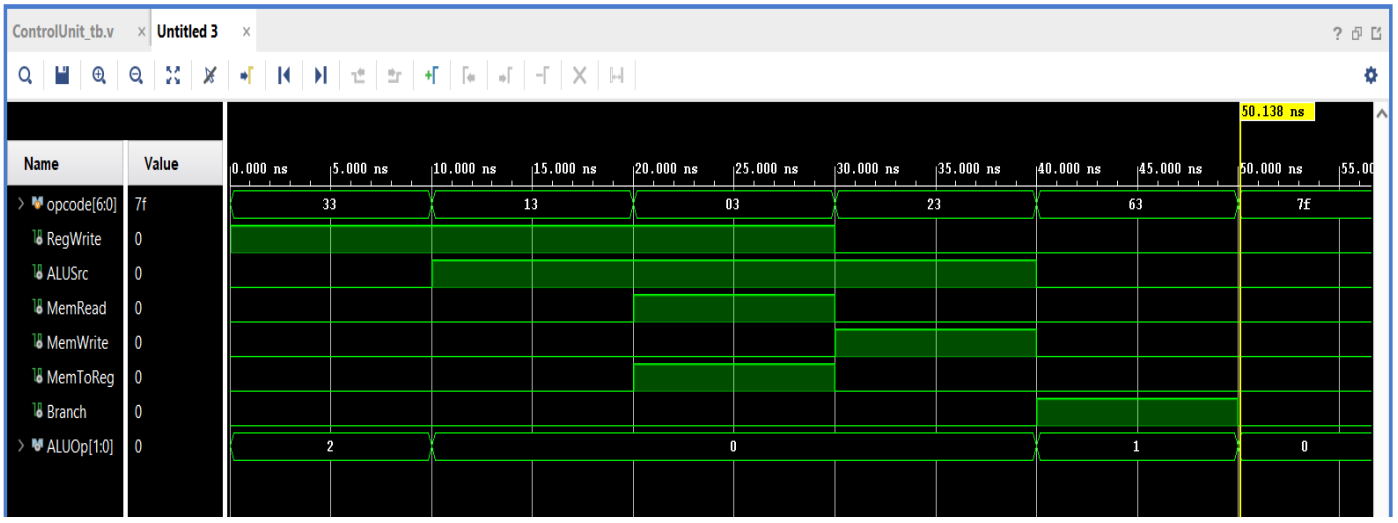


Fig.2.B.2. Behavioral Simulation of Control unit using Vivado 2024.2

The waveform above illustrates the functional simulation of a Control Unit in a RISC-V processor design, likely implemented in Verilog. The input signal opcode changes over time to represent different instruction types, including R-type (33), I-type (13), load (03), store (23), branch (63), and J-type (7F). As each opcode is applied, the Control Unit generates corresponding control signals such as RegWrite, ALUSrc, MemRead, MemWrite, MemToReg, and Branch. These outputs determine the behavior of other parts of the datapath. For instance, during the load instruction (03), MemRead, MemToReg, and RegWrite are asserted, while during the store instruction (23), MemWrite is active. The ALUOp signal also changes appropriately to guide the ALU in executing the correct operation. This simulation confirms that the Control Unit responds correctly to various instruction types by asserting the proper control signals at the right time, validating its role in orchestrating processor operations.

3. The Design of Datapath Unit (DP)

The **Datapath Unit** is at the heart of every Central Processing Unit (CPU), playing a crucial role in handling data operations. It works hand-in-hand with the **Control Unit (CU)**, which provides the signals that guide its behaviour. While the control unit issues commands, the Datapath is where those commands are executed, responsible for carrying

out actual computations, data transfers, and register manipulations.

The Datapath consists of interconnected hardware blocks such as **registers**, **ALUs**, **multiplexers**, and **buses**, all orchestrated to efficiently process instructions and manipulate data within the processor.

Key Components of the Datapath Unit

1. **Registers**
   Registers are fast, small-capacity memory elements that temporarily store data during instruction execution. The **register file** typically includes multiple general-purpose registers—**32 in the case of RISC-V** architectures. Another vital register is the **Program Counter (PC)**, which keeps track of the address of the next instruction to be executed.
2. **Arithmetic Logic Unit (ALU)**
   The ALU performs all the arithmetic (e.g., addition, subtraction) and logical (e.g., AND, OR, XOR) operations needed during instruction execution. It receives inputs from registers or immediate values and outputs the result either back into a register or to memory.
3. **Multiplexers (MUXes)**
   MUXes are used to select between multiple input sources and send the appropriate one to a particular destination, such as the ALU or memory. They play a critical role in making the Datapath flexible and responsive to different instruction types.
4. **Buses**
   Buses are communication pathways that carry data, addresses, and control signals among components. In a typical CPU, there are separate **data buses** and **address buses** that connect registers, memory, and processing units. The quality and structure of these buses directly impact system speed and efficiency.
5. **Memory Access Logic**
   This component manages the interface between the Datapath and system memory. It enables **instruction fetches** as well as **data loads and stores**. Dedicated **load/store units** handle addressing and data transfer tasks to and from memory.
6. **Instruction Register (IR)**
   The IR holds the current instruction fetched from memory. It provides the control unit with the opcode and other instruction fields necessary to determine the operations the Datapath must perform.

How does the Datapath operate?

The Datapath follows a step-by-step process during the execution of each instruction:

1. **Fetch**: The instruction is retrieved from memory using the Program Counter.
2. **Decode**: The control unit decodes the instruction and prepares control signals.
3. **Execute**: The ALU performs the specified operation, using inputs from registers or constants.
4. **Memory Access**: If the instruction involves reading from or writing to memory, the appropriate data is transferred.
5. **Write Back**: The result of the computation or memory operation is written back to a register, completing the instruction cycle.

1. How to implement the DP Unit

Step 1: Extract Fields from the Instruction
The RISC-V instruction is broken into parts for decoding:
These fields help:

1. Identify source and destination registers
2. Determine ALU operation (with funct3, funct7)
3. Determine instruction type via opcode

Step 2: Instantiate Register File
The register file:

1. Reads from rs1 and rs2
2. Writes to rd if RegWrite = 1
3. Uses WritebackData as the data to write

Step 3: Immediate Generator
Different RISC-V instructions encode immediates differently, so we need a **decoder**.
This logic:

1. Sign-extends the immediate to 32 bits
2. Supports I, S, and B types

Step 4: ALU Control Logic
The ALU operation depends on:

1. The ALUOp control signal from the control unit
2. For R-type: funct3 and funct7

ALUControl selects the specific ALU operation:

1. 0101: ADD
2. 0110: SUB
3. 0000: AND
4. 0001: OR
5. 0111: SLT (set less than)

Step 5: Choose ALU Input B (Register or Immediate)
If ALUSrc = 1, use the **immediate** value.
If ALUSrc = 0, use readData2 (from register file).

Step 6: ALU Instance
Inputs: two operands and control signal
Outputs:

1. ALUResult: result of the operation
2. Zero: used for branches (BEQ)

Step 7: Branch Target Calculation
Branch target = current PC + sign-extended immediate

Step 8: Writeback Data Selection
Normally, this would select between:

1. ALUResult and
2. Data loaded from memory

Since **memory is not connected yet**, it hardcodes the memory result as 0:

1. If MemToReg = 0, write the ALU result back
2. If MemToReg = 1, write 0 (placeholder for memory)

Step 9: Expose Outputs
The following are exposed from the module:

1. Zero – used by the control unit for branching
2. ALUResult – the result of the computation
3. WriteData – data to be stored (used for sw)
4. WriteBackData – result that will go into the destination register
5. BranchTarget – used to update the PC if the branch is taken.

3. Verilog Code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
module Datapath (
    input clk,
    input reset,
    input [31:0] instruction,
    input [31:0] PC,
    input RegWrite, ALUSrc, MemRead, MemWrite,
    MemToReg, Branch,
    input [1:0] ALUOp,
    output Zero,
    output [31:0] ALUResult,
    output [31:0] WriteData,
    output [31:0] WriteBackData,
    output [31:0] BranchTarget
);

    wire [4:0] rs1 = instruction[19:15];
    wire [4:0] rs2 = instruction[24:20];
    wire [4:0] rd  = instruction[11:7];
    wire [6:0] funct7 = instruction[31:25];
    wire [2:0] funct3 = instruction[14:12];
    wire [6:0] opcode = instruction[6:0];

    wire [31:0] readData1, readData2;

    RegisterFile rf (
        .clk(clk), .RegWrite(RegWrite),
        .rs1(rs1), .rs2(rs2), .rd(rd),
        .writeData(WriteBackData),
        .readData1(readData1), .readData2(readData2)
    );

assign WriteData = readData2;


reg [31:0] imm;
always @(*) begin
    case (opcode)
        7'b0010011, 7'b0000011: // I-type
            imm = {{20{instruction[31]}}, instruction[31:20]};
        7'b0100011: // S-type
            imm = {{20{instruction[31]}}, instruction[31:25], instruction[11:7]};
        7'b1100011: // B-type
            imm = {{19{instruction[31]}}, instruction[31], instruction[7],
            instruction[30:25], instruction[11:8], 1'b0};
        default:
            imm = 32'd0;
    endcase
end


reg [3:0] ALUControl;
always @(*) begin
    case (ALUOp)
        2'b00: ALUControl = 4'b0010; // ADD for load/store/addi
        2'b01: ALUControl = 4'b0110; // SUB for BEQ
        2'b10: begin // R-type
            case ({funct7, funct3})
                {7'b0000000, 3'b000}: ALUControl = 4'b0010; // ADD
                {7'b0100000, 3'b000}: ALUControl = 4'b0110; // SUB
                {7'b0000000, 3'b111}: ALUControl = 4'b0000; // AND
                {7'b0000000, 3'b110}: ALUControl = 4'b0001; // OR
                {7'b0000000, 3'b010}: ALUControl = 4'b0111; // SLT
                default:              ALUControl = 4'b0000;
            endcase
        end
        default: ALUControl = 4'b0000;
    endcase
end

// ALU input selection
wire [31:0] ALUInput2 = (ALUSrc) ? imm : readData2;

// ALU instance
ALU alu (
    .A(readData1), .B(ALUInput2), .ALUControl(ALUControl),
    .Result(ALUResult), .Zero(Zero)
);

// Branch target = PC + imm
assign BranchTarget = PC + imm;

// Write back selection (ALU or memory)
assign WriteBackData = (MemToReg) ? 32'd0 : ALUResult;

endmodule
```

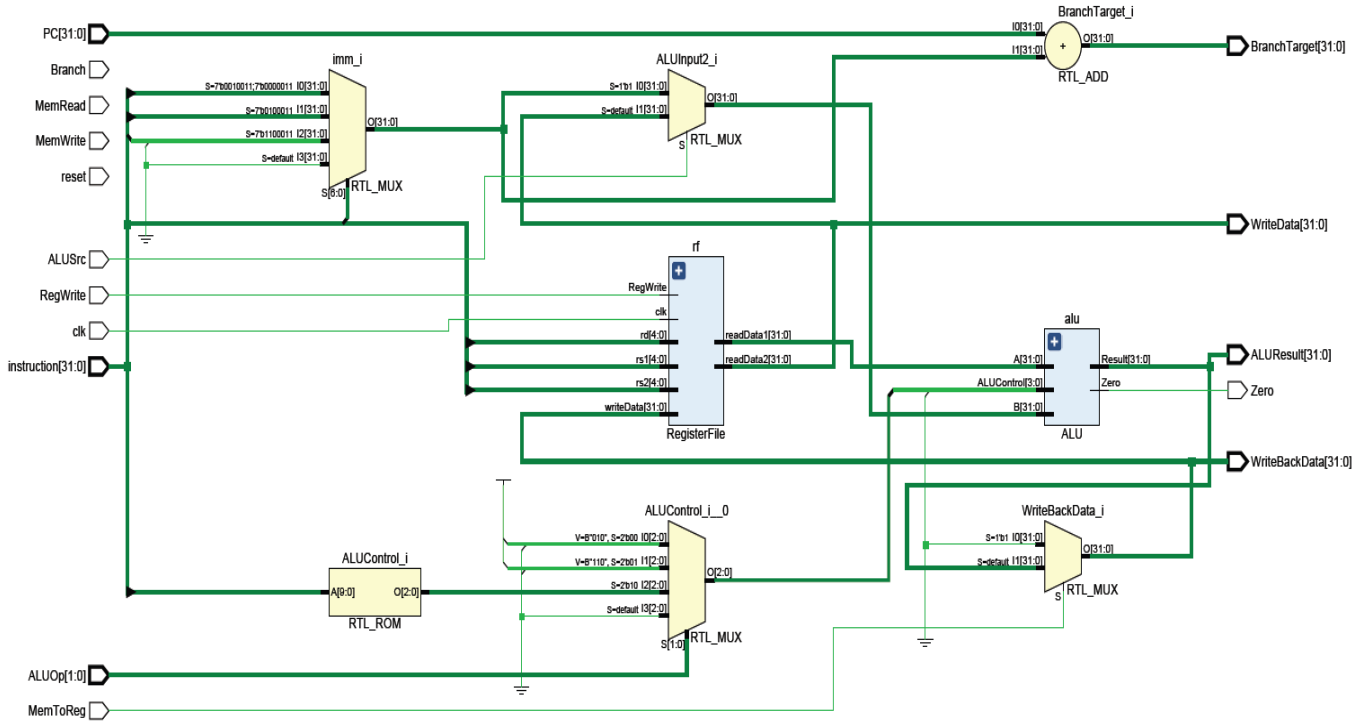


Fig 3.B.1 Design Datapath Unit (internal) implementation using Vivado 2024.2

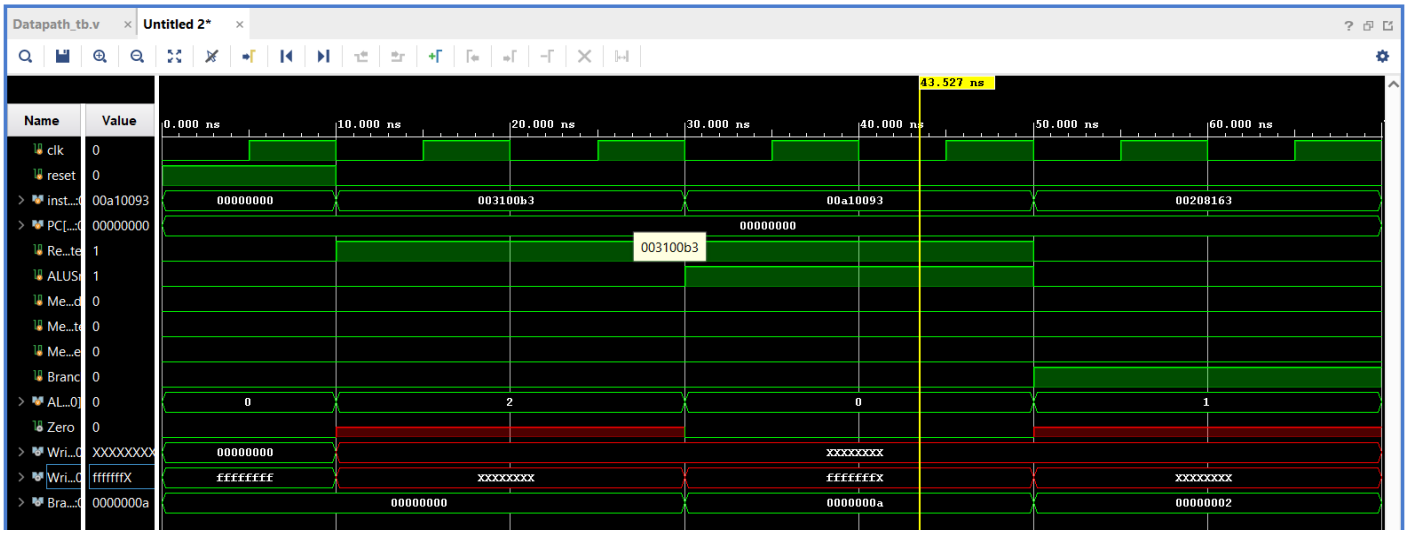


Fig 3.B.2 Behavioral Simulation of Datapath Unit using Vivado 2024.2

This waveform illustrates the behavioral simulation of a RISC-V single-cycle datapath implemented in Verilog. The key signals include the instruction input (inst), program counter (PC), control signals (such as ALUOp, ALUSrc, MemWrite, etc.), and various internal data paths like WriteData, Result, and BranchAddr. The instruction signal (inst) changes at regular intervals, each time triggering a new instruction fetch and decode cycle. At time 10 ns, the instruction 003100B3 is decoded and executed—likely an R-type instruction—reflected by the ALU control signal being set to 2 and the ALU output updating accordingly. The PC and BranchAddr values also change over time, indicating sequential instruction execution or branching behavior. Control signals like MemWrite, MemRead, and RegWrite toggle based on the instruction type, affecting memory and register file interactions. The signal Zero is monitored for branching decisions. Throughout the simulation, the datapath behaves as expected, correctly fetching instructions, generating control signals, and updating data paths, verifying the correct integration of all datapath components.

4. The design of Register File (RF)

The **Register File Unit** is an essential subsystem within a CPU, serving as a small, ultra-fast memory bank used for holding data and temporary results during the execution of instructions. This unit provides immediate access to operands needed for computation and to destinations for storing processed results. It is especially critical in **RISC-based architectures**, like **RISC-V**, where

instruction execution relies heavily on register-based operations.

In a typical RISC-V design, the register file contains **32 general-purpose registers**, each being either **32 bits (RV32)** or **64 bits (RV64)** wide, depending on the architecture. Notably, register **x0** is a constant zero register—it's hardwired to always return 0, no matter what is written to it.

Key Components of the Register File

1. **Read Ports (Dual-Ported Read Access)**: The unit features **two independent read ports**, allowing simultaneous access to two different registers. This is crucial for most arithmetic and logic instructions that require two operands.
2. **Write Port**: There is a **single write port** through which results from the ALU or memory are written back into a destination register.
3. **Address Decoding Logic**: Each read or write operation involves specifying the register index. **Decoders** convert these indices into select lines that enable the corresponding register for read or write.

How the Register File Unit Works

Here's a step-by-step breakdown of the typical operation of a register file:

1. **Instruction Decode Phase**
When a machine instruction is fetched from memory, the **Control Unit** decodes it. Part of this decoding involves identifying the register addresses—usually two sources (rs1, rs2) and one destination (rd).
2. **Addressing Registers**
The register indices from the instruction are fed into the register file. These indices determine which registers should be read from (for operands) and which should be written to (for the result).
3. **Reading Operand Data**
The two read ports provide data simultaneously from the specified registers. For instance, in an instruction like ADD x5, x1, x2, the values from registers x1 and x2 are read and sent to the ALU.
4. **Writing Results**
After processing is complete (typically in the ALU), the result is routed back to the register file and written into the destination register (x5 in this example) through the write port.
5. **Clock Synchronization**
All operations are **synchronized with the CPU clock**, ensuring precise timing for data transfer. Reading usually happens in the **same clock cycle**, while writing occurs on the **rising edge of the clock**, ensuring stability and avoiding race conditions.

1. How to implement the Register File

Step 1. Module Declaration

1. clk: Clock signal used to trigger writing to the register file.
2. RegWrite: A control signal indicating whether a write should occur.
3. rs1 and rs2: 5-bit inputs representing addresses of source registers to read.
4. rd: 5-bit input representing the destination register to write to.
5. writeData: 32-bit data that will be written to the register if enabled.
6. readData1 and readData2: Outputs for the data read from rs1 and rs2.

Step 2. Register Array Declaration

1. This creates an array of 32 registers (registers[0] to registers[31]), each 32 bits wide.
2. These registers store the actual values used during program execution.
3. This is the physical storage of the register file.

Step 3. Asynchronous Read Logic

1. This block handles reading from the register file.
2. It performs **asynchronous reads**, meaning the data is immediately available when the input address (rs1, rs2) changes, no clock needed.
3. If the source register is 0, it outputs 0 regardless of what's stored; this follows RISC-V's rule that **register x0 is always zero** and cannot be modified.
4. Otherwise, it fetches the data from the register array using the index rs1 or rs2.

Step 4. Synchronous Write Logic

1. This always block is triggered **only on the rising edge of the clock**.
2. Inside the block, a write operation is performed **only if RegWrite is high** (write enabled) **and the destination register rd is not zero**.
3. This protects register 0 from being modified, as required by RISC-V.
4. If allowed, writeData is stored into the register at index rd.

2. Verilog code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////

module RegisterFile (
    input clk,
    input RegWrite,
    input [4:0] rs1,
    input [4:0] rs2,
    input [4:0] rd,
    input [31:0] writeData,
    output [31:0] readData1,
    output [31:0] readData2
);
    reg [31:0] registers[0:31];

    // Read data (asynchronous)
    assign readData1 = (rs1 == 5'd0) ? 32'd0 : registers[rs1];
    assign readData2 = (rs2 == 5'd0) ? 32'd0 : registers[rs2];

    // Write data (on positive clock edge)
    always @(posedge clk) begin
        if (RegWrite && rd != 5'd0) begin
            registers[rd] <= writeData;
        end
    end
endmodule
```

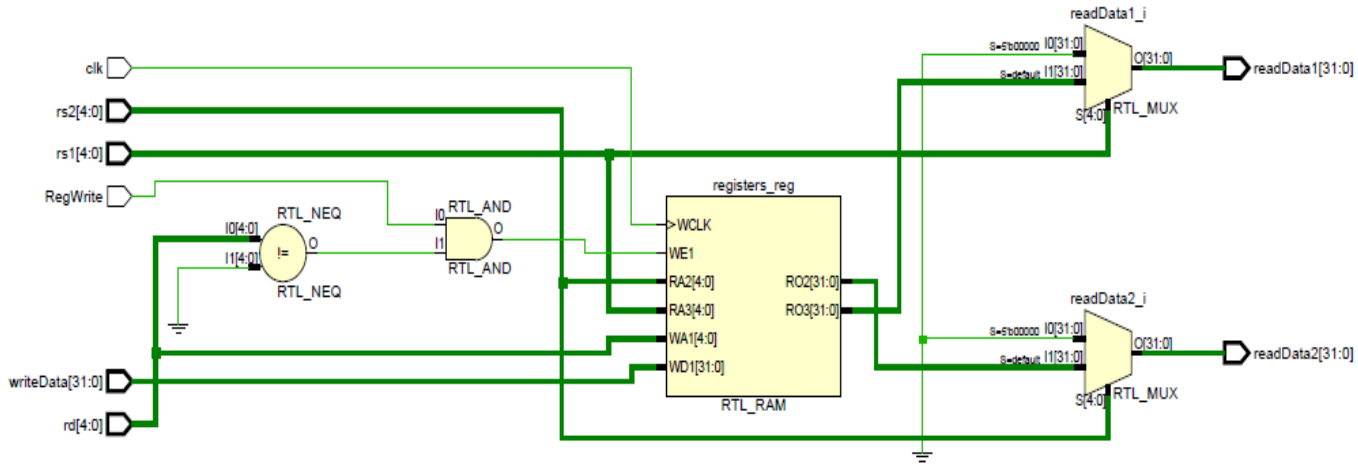


Fig 4.B.1 Design Register File Unit (internal) implementation using Vivado 2024.2

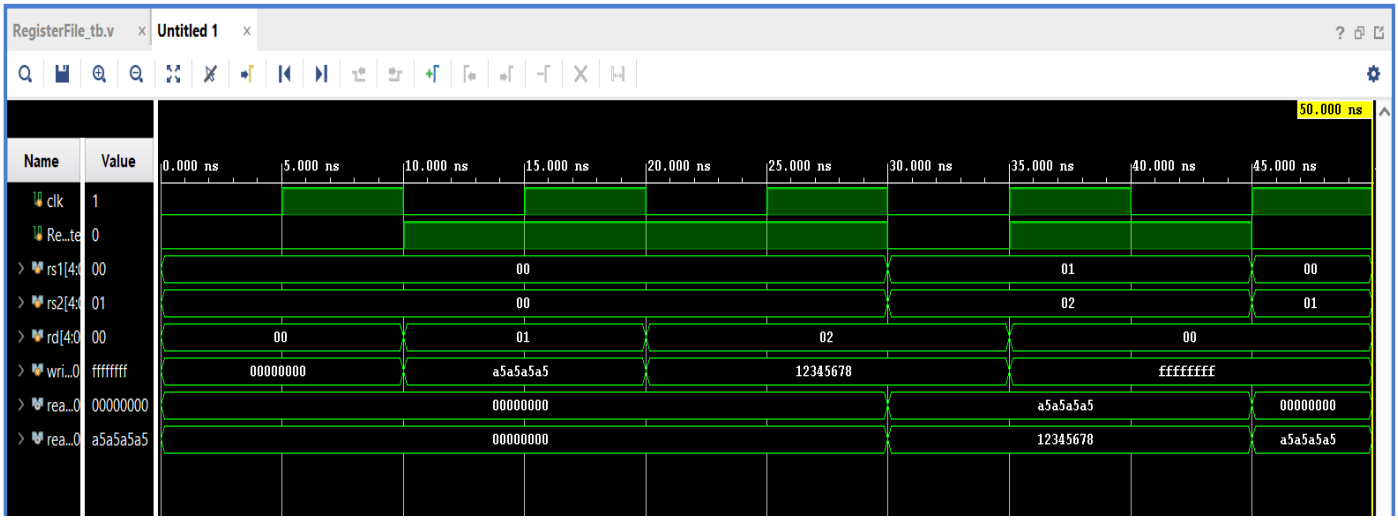


Fig 4.B.2 Behavioral Simulation of Register File Unit showing different values that are written or read into/from the register file memory using Vivado 2024.2

This waveform displays the behavioral simulation of a Register File module, a critical component of a RISC-V processor, designed to store and provide quick access to operand values. The key inputs include read and write register addresses (rs1, rs2, rd), the write enable signal (RegWrite), and the data to be written (write_data). The outputs read_data1 and read_data2 reflect the contents of the specified source registers. Initially, registers contain default values. At around 10 ns, a write operation is triggered: RegWrite is enabled, the destination register rd is set to 1, and the value 0xA5A5A5A5 is written. This is confirmed by read_data1 reflecting the new value when rs1 selects register 1. Later, another write stores 0x12345678 into register 2, and again, the corresponding read port shows the updated value. These write and read cycles confirm that the register file correctly handles data storage and retrieval, conditioned on clock edges and write-enable logic. Overall, this simulation validates the correct functionality of simultaneous register reads and conditional writes in the Register File module.

5. The design of Instruction Fetch Unit (IFU)

The **Instruction Fetch Unit (IFU)** is the first and one of the most critical stages in the CPU's instruction processing pipeline. Its main task is to **fetch instructions from memory** in the correct sequence, ensuring that the processor executes programs accurately and efficiently. Acting as the entry point to the execution cycle, the IFU lays the groundwork for subsequent decoding and execution.

Core Components of the IFU

1. **Program Counter (PC)**
   The **Program Counter** is a special-purpose register that holds the memory address of the **next instruction** to be executed. After each fetch, the PC is normally incremented to point to the next instruction. In the case of a jump or branch, the PC is updated with a new target address.
2. **Instruction Memory**
   This memory unit (which could be an **instruction cache** or ROM) stores the **machine code** of the program. The IFU uses the current PC value to **retrieve an instruction** from this memory.
3. **PC Adder (PC + 4)**
   Since each instruction in most **RISC architectures**, including **RISC-V**, is 4 bytes long, the IFU includes an **adder** that calculates the next sequential PC value as PC + 4. This allows the IFU to prepare for fetching the next instruction unless a control instruction dictates otherwise.
4. **Multiplexer (MUX)**
   The MUX is used to **select the next PC value**. If a **branch or jump** occurs, the MUX chooses between the regular PC + 4 and a **branch/jump target address**. This decision is based on control signals generated by the control unit or branch logic.
5. **Branch Target Calculator (optional)**
   In case of conditional or unconditional branches, this unit computes the **target address** by adding an **immediate offset** to the current PC. This target address becomes the new input to the PC if the branch is taken.

How the Instruction Fetch Unit Operates

The IFU performs the following sequence of actions in each cycle:

Fetch the Instruction:
The **current PC value** is used to access the instruction memory.
The instruction located at that address is **fetched** and passed to the **Instruction Decode Unit**.

Update the PC:
If the instruction is **not** a jump or branch, the PC is simply updated to PC + 4.

If the instruction **alters control flow**, the PC is updated using the **branch or jump target address**, selected via the **MUX**.

1. How to implement IFU

1. Module Declaration and Ports

1. clk: Clock signal — used to trigger PC updates.
2. reset: Active-high reset signal to reinitialize PC to 0.
3. branch: Control signal — if high, the PC jumps to branchAddr.
4. branchAddr: The target address if a branch is taken.
5. PC: The current program counter (register).
6. instruction: The current instruction fetched from memory.

2. Instruction Memory Declaration

1. This creates an array of 256 **32-bit instruction slots**, simulating a small instruction memory.
2. Each index stores a full 32-bit machine instruction (like RISC-V instructions).
3. This is a simple **read-only instruction memory**, hardcoded in the next step.

3. Initializing Instruction Memory

1. Using the initial block, the memory is preloaded with five sample RISC-V instructions.
2. These are encoded as 32-bit hex values.
3. For simulation purposes, this model has a basic instruction program hardcoded into memory.

4. Instruction Fetch Logic (Combinational)

1. The instruction is selected using PC[9:2]:
   1. Since RISC-V instructions are 4 bytes (32 bits) wide, we use bits [9:2] of PC for word-aligned access.
   2. This divides the PC by 4 (i.e., PC >> 2) to index into the instruction memory.
2. This read is **asynchronous** — the instruction is immediately available when PC changes.

5. Program Counter Update (Synchronous Block)

1. This is a **clocked always block** that updates the PC on the rising edge of the clock or reset.
2. **Reset logic**: If reset is high, PC is set to 0 (starting address).
3. **Branch logic**: If branch is true, the PC is updated to branchAddr.
4. **Default behavior**: If no branch or reset, PC is incremented by 4 to fetch the next sequential instruction.

2. Verilog code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
module IFU (
    input clk,
    input reset,
    input branch,
    input [31:0] branchAddr,
    output reg [31:0] PC,
    output [31:0] instruction);
    reg [31:0] instrMem[0:255];
    initial begin
        instrMem[0] = 32'h00010011;
        instrMem[1] = 32'h00110011;
        instrMem[2] = 32'h00000011;
        instrMem[3] = 32'h00100011;
        instrMem[4] = 32'h01100011;
        instrMem[5] = 32'h00010111;
        instrMem[6] = 32'h00110111;
        instrMem[7] = 32'h01101111;
        instrMem[8] = 32'h01100111;
    end
    assign instruction = instrMem[PC[9:2]];
    always @(posedge clk or posedge reset) begin
        if (reset)
            PC <= 32'd0;
        else if (branch)
            PC <= branchAddr;
        else
            PC <= PC + 4;
    end
endmodule
```

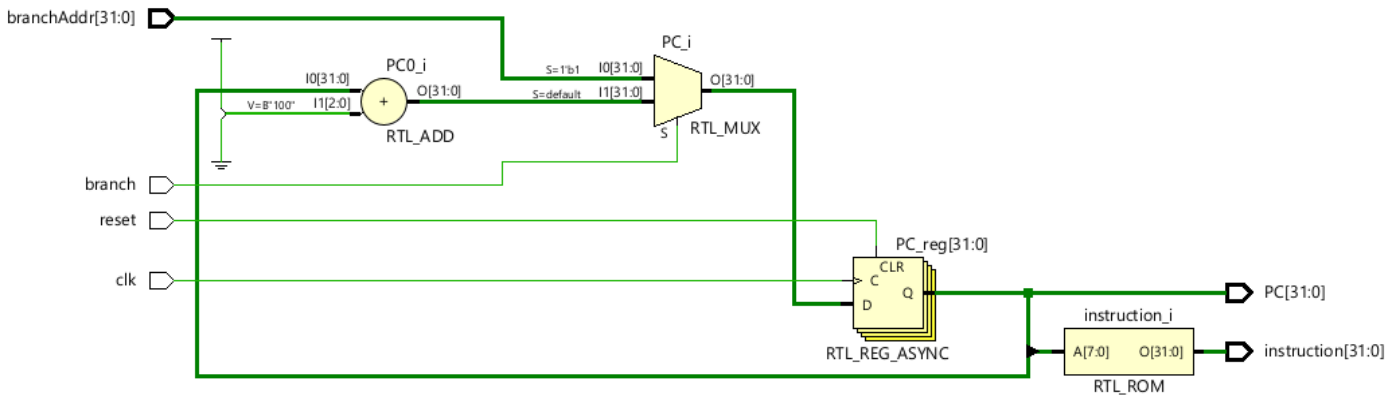


Fig 5.B.1 Design Instruction Fetch Unit (internal) implementation using Vivado 2024.2

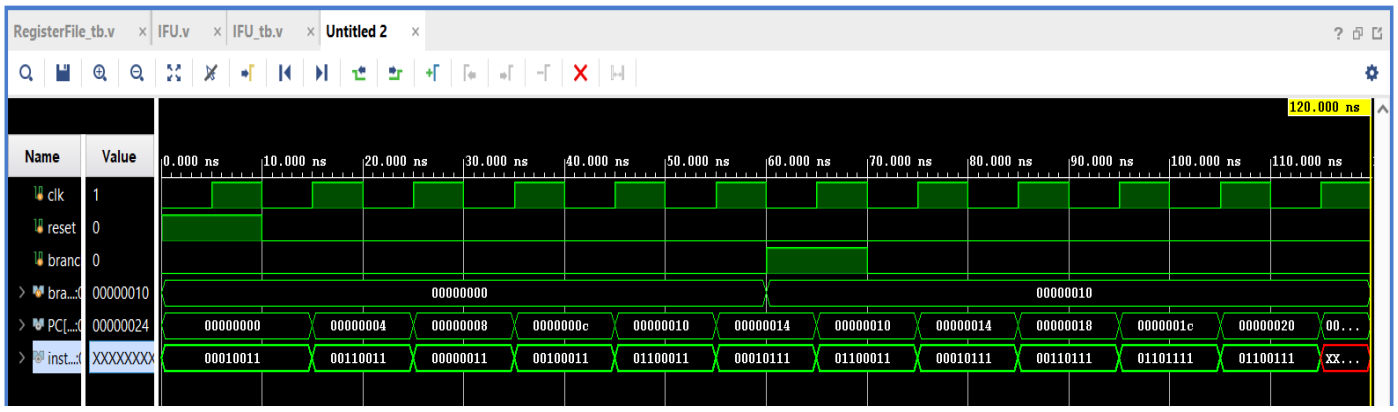


Fig 4.B.2 Behavioral Simulation of Instruction Fetch Unit showing different binary instructions that are loaded into the instruction [31:0] register using Vivado 2024.2

This waveform illustrates the simulation of an Instruction Fetch Unit (IFU), a core part of a RISC-V processor responsible for fetching instructions from memory. The main signals include the clock (clk), reset, branch control signals, program counter (PC_out), and the fetched instruction (inst). Initially, the reset is low, allowing the IFU to begin normal operation. The PC_out signal increments by 4 every clock cycle, indicating that the processor is fetching instructions sequentially from memory, which is the expected behavior in a non-branching scenario. Each new instruction is loaded into the inst signal, showing correct memory access. Around 60 ns, the branch signal becomes high, and branch_addr is loaded with the value 0x10. As a result, the PC_out is updated accordingly, demonstrating the IFU's ability to handle branch redirection. After this, sequential execution resumes. This simulation confirms that the IFU properly increments the program counter, fetches the correct instruction from memory, and successfully redirects control flow when a branch occurs.

III. The Design of Processor

CENTRAL PROCESSING UNIT (CPU)

The **Central Processing Unit (CPU)** is the core of any computing system, responsible for carrying out instructions and handling computational tasks. When built around the **RISC-V** architecture, the CPU benefits from a streamlined, efficient design focused on simplicity, scalability, and modularity. **RISC-V** is an open-source instruction set architecture (ISA) that is both customizable and extensible, making it ideal for a wide range of applications from embedded devices to high-performance computing.

CPU Architecture

A typical **RISC-V CPU** uses a **pipelined architecture**. This approach breaks down instruction execution into distinct stages, allowing multiple instructions to be processed simultaneously but at different stages. The major stages of this pipeline are:

1. **Instruction Fetch (IF):** Retrieves the next instruction from MeM, using the **Program**

Counter (PC) to determine the address.

2. **Instruction Decode (ID):** Decodes the fetched instruction and identifies which registers are needed for execution.

3. **Execute (EX):**

   The arithmetic or logical operation (such as addition, subtraction, or comparison) is carried out during this stage.

4. **MeM Access (MEM):** This stage deals with reading from or writing to MeM when required (for load/store operations).

5. **Write Back (WB):** The results of the executed instruction are written back into the CPU's register file.

Key Components and Their Options

The **RISC-V CPU** is designed to be simple yet highly effective, with the following key elements:
operations such as addition, subtraction, bitwise logical operations (AND, OR), and comparisons.

1. **Data MeM:**

   This is accessed during the **MEM** stage to load or store data as needed by the instruction.

2. **Control Logic:**

   The control unit orchestrates the entire CPU operation by generating signals that guide each stage of the pipeline. It also manages branch operations, sometimes using a **branch prediction unit** to optimize performance.

RISC-V Modularity

The **RISC-V architecture** is highly modular, supporting a base integer instruction set (such as **RV32I**, **RV64I**, or **RV128I**) that defines the fundamental operations. Additionally, RISC-V supports optional extensions that add more capabilities to the architecture:

1. **M Extension:**

   Adds support for multiplication and division operations.

2. **F and D Extensions:** Provide support for floating-point operations, with **F** for single-precision and **D** for double-precision.

3. **C Extension:**

   Introduces compressed instruction formats to improve code density and reduce MeM usage.

This modular design makes RISC-V flexible, enabling tailored CPU implementations for specific applications, whether it's a low-power embedded system or a high-performance processor.

How It Operates

When a program is executed, the **RISC-V CPU** follows a cycle of steps:

1. The CPU **fetches** the instruction from MeM using the **PC**.

2. The instructions are then **decoded** to identify the operation and the involved operands.

3. The required operation is **executed** by the **ALU** or through a **MeM access** if needed.

4. The result is then **written back** to the appropriate register.

Finally, the **PC** is updated to point to the next instruction, and the cycle repeats.

1. How to implement the Processor

1. Module Declaration

1. The processor takes a clock (clk) and an active-high reset (reset) input.

2. No inputs/outputs for external memory yet — it's focused on core internal operation.

2. Internal Wire Declarations

PC: Program Counter value fetched from the IFU.

1. instruction: The 32-bit machine instruction is fetched from instruction memory.

2. These are control signals generated by the Control Unit based on the instruction opcode:

   1. RegWrite: Enable register write.

   2. ALUSrc: Select ALU input source.

   3. MemRead / MemWrite: Control memory access (to be used in future).

   4. MemToReg: Select whether ALU result or memory data is written back.

   5. Branch: Indicates if the instruction is a branch.

   6. ALUOp: Tells ALU what operation to perform.

1. Zero: ALU sets this flag if the result is zero (for conditional branches).

2. ALUResult: Output of ALU calculation.

3. WriteData: Data to be written to memory (if MemWrite enabled).

4. WriteBackData: Data to be written back to the register file.

5. BranchTarget: Target address for branch instructions.

3. Instruction Fetch Unit (IFU)

1. The IFU fetches the instruction at address PC from

instruction memory.

2. If a branch is taken (Branch & Zero is true), it jumps to BranchTarget.
3. Otherwise, it continues fetching sequential instructions (PC + 4).

4. Control Unit (CU)

1. Extracts the opcode field from the instruction (bits 6:0).
2. Passes this opcode to the ControlUnit, which generates the appropriate control signals.
3. The control logic determines how the datapath behaves.

5. Datapath Integration

1. The Datapath module handles the core execution, including:
    1. Register reads and writes
    2. ALU operations
    3. Branch address computation
    4. Selecting inputs based on control signals
2. It receives all necessary control signals and returns outputs like Zero, ALUResult, and the next BranchTarget.

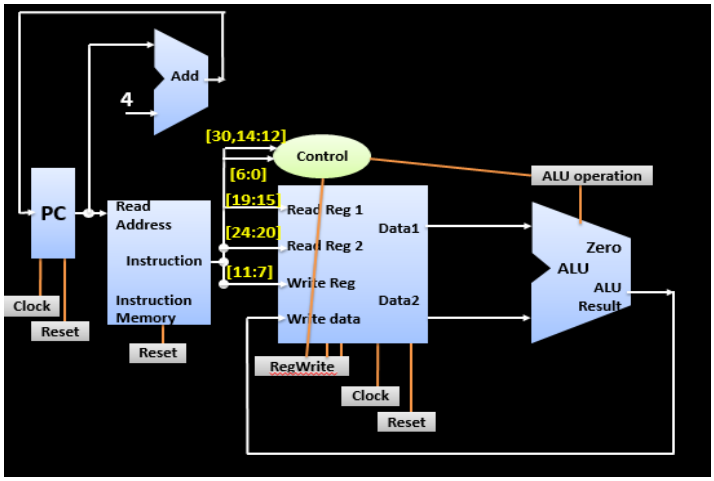


Fig 6.A.1 Processor Architecture (Source: GitHub)

2. Verilog Code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
module Processor (
    input clk,
    input reset);
    wire [31:0] PC;
    wire [31:0] instruction;
    wire RegWrite, ALUSrc, MemRead, MemWrite, MemToReg, Branch;
    wire [1:0] ALUOp;
    wire Zero;
    wire [31:0] ALUResult, WriteData, WriteBackData, BranchTarget;
    IFU IFU (
        .clk(clk),
        .reset(reset),
        .branch(Branch & Zero),
        .branchAddr(BranchTarget),
        .PC(PC),
        .instruction(instruction)
    );
    wire [6:0] opcode = instruction[6:0];
    ControlUnit CU (
        .opcode(opcode),

        .RegWrite(RegWrite),
        .ALUSrc(ALUSrc),
        .MemRead(MemRead),
        .MemWrite(MemWrite),
        .MemToReg(MemToReg),
        .Branch(Branch),
        .ALUOp(ALUOp)
   );
   Datapath DP (
        .clk(clk),
        .reset(reset),
        .instruction(instruction),
        .PC(PC),
        .RegWrite(RegWrite),
        .ALUSrc(ALUSrc),
        .MemRead(MemRead),
        .MemWrite(MemWrite),
        .MemToReg(MemToReg),
        .Branch(Branch),
        .ALUOp(ALUOp),
        .Zero(Zero),
        .ALUResult(ALUResult),
        .WriteData(WriteData),
        .WriteBackData(WriteBackData),
        .BranchTarget(BranchTarget)
    );
endmodule
```

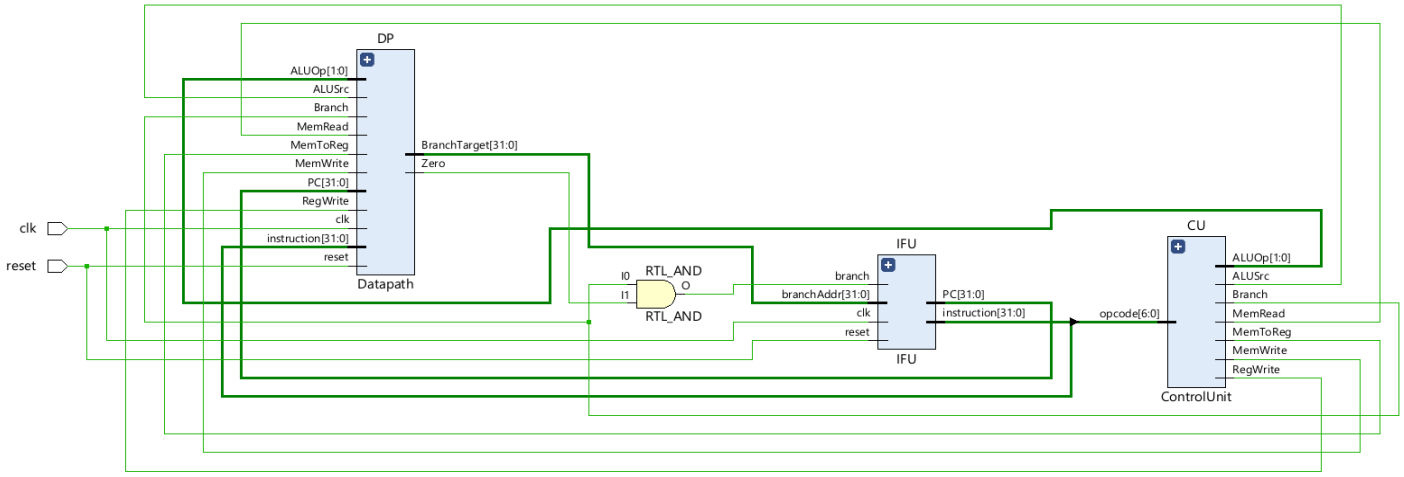


Fig 6.B.1 The (compressed) architecture of the Processor using Vivado 2024.2

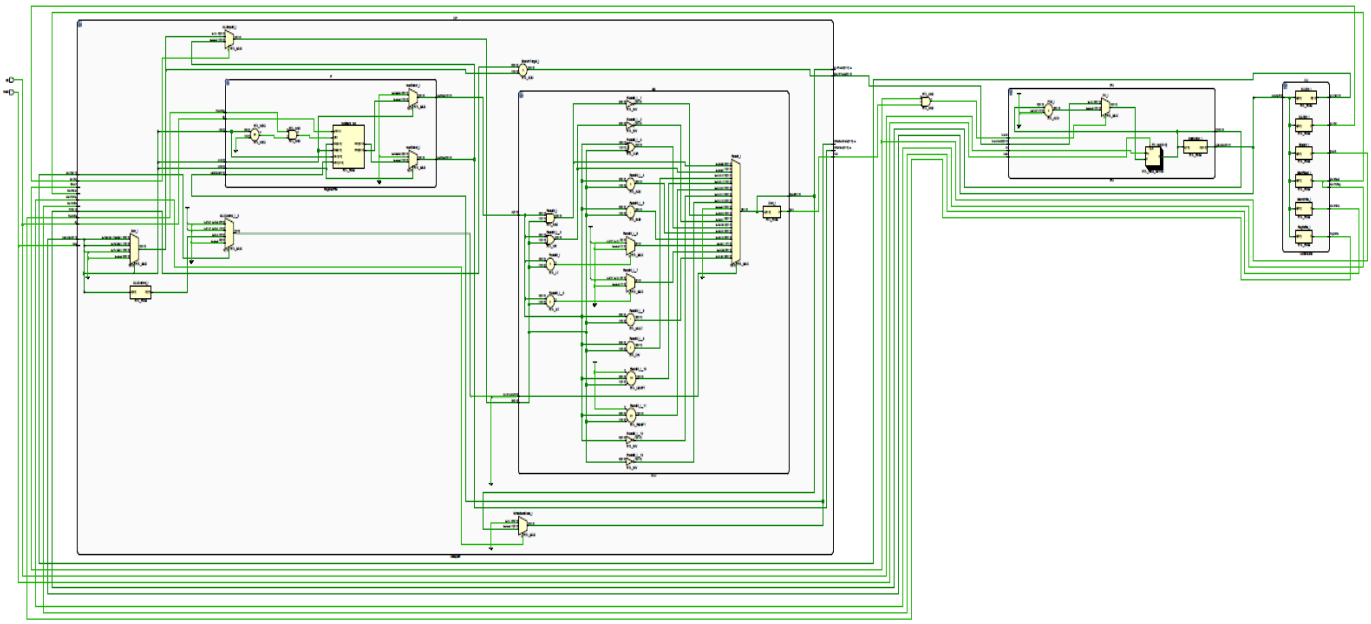


Fig 6.B.2 Schematic showing all the components connected in a basic RISC-V Processor

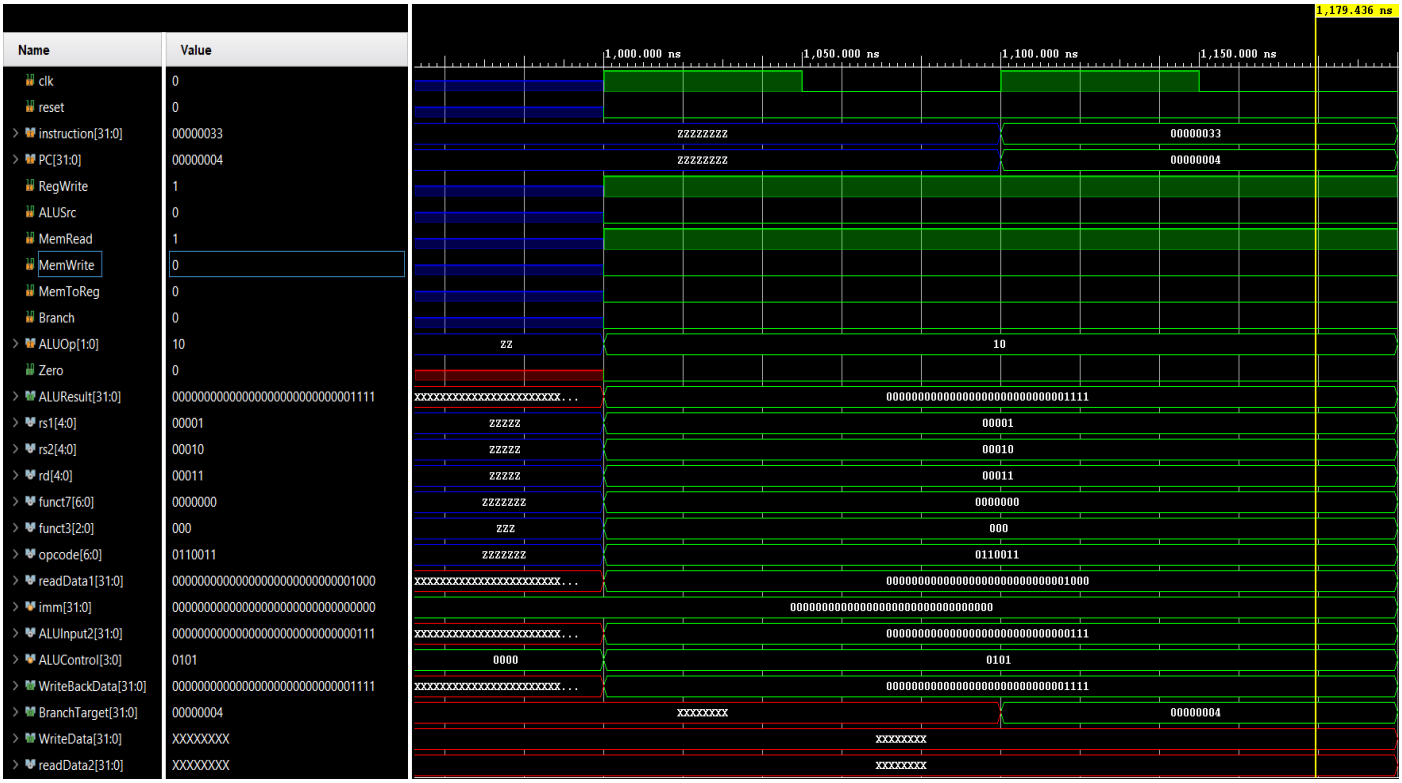


Fig 6.B.3 Behavioral Simulation of the entire processor in Vivado 2024.2, showing different signals as inputs and values as per the data from the memory/immediate value.

This waveform illustrates a comprehensive simulation of a RISC-V processor's full CPU executing an instruction cycle. Key signals include the instruction (instruction), program counter (PC), control signals (like RegWrite, ALUSrc, MemRead, MemWrite, etc.), ALU result (ALUResult), register addresses (rs1, rs2, rd), and data lines. Around the 1,100 ns mark, the instruction 0x00000033 is loaded and decoded. This corresponds to an R-type instruction, verified by the opcode 0110011. The control unit activates relevant signals: RegWrite is high, ALUSrc is low (selecting register input for ALU), and ALUOp is set to 10, indicating a function-based ALU operation. The source register values (readData1, readData2) are read and used by the ALU to compute the result, which is 0xF in this case. The result is written back to the destination register as indicated by WriteBackData. The branching and memory-related controls (Branch, MemRead, MemWrite, MemToReg) remain inactive, confirming this is a computation-only instruction. The simulation successfully demonstrates the correct sequencing of fetching, decoding, execution, and write-back stages within a single-cycle processor architecture, confirming functional integration of the ALU, control unit, register file, and memory interface.

IV. PROJECT INVOLVING THE USE OF RISC-V PROCESSOR

Smart Irrigation System Using RISC-V Processor

In recent years, the integration of embedded systems with agriculture has opened new horizons for smart farming solutions. One such innovation is the **smart irrigation system**, which uses environmental data to optimize water usage in farming. A particularly efficient and open-source approach to building such systems is by using a **RISC-V processor**. RISC-V's modularity, flexibility, and low power consumption make it ideal for embedded agricultural applications.

Why Smart Irrigation?

Traditional irrigation techniques often result in excessive water usage, uneven watering, and dependence on manual intervention. With climate change and water scarcity becoming critical global issues, there's a growing demand for intelligent systems that conserve resources while improving crop yields. Smart irrigation systems monitor parameters such as soil moisture, temperature, and humidity to determine optimal watering schedules. They reduce human error, save water, and support sustainable farming practices.

Role of RISC-V in Smart Irrigation

RISC-V is an open-source Instruction Set Architecture (ISA) that offers significant advantages for embedded and IoT applications. Unlike proprietary ISAs, RISC-V allows developers to customize and optimize the processor for specific tasks, such as sensor data acquisition and control logic in irrigation systems. This flexibility leads to power-efficient and cost-effective solutions tailored for agricultural environments.

In a smart irrigation setup, a RISC-V-based microcontroller can be programmed to interface with a variety of sensors, such as soil moisture probes, temperature sensors, and light

detectors. The processor continuously reads data from these sensors and processes it to determine when and how much to irrigate. Additionally, it can communicate with actuators like solenoid valves to control the flow of water.

System Architecture

The typical architecture of a RISC-V-based smart irrigation system includes:

1. **Sensor Module**: Gathers real-time environmental data.
2. **RISC-V Processor Unit**: Serves as the brain of the system, executing control algorithms based on sensor inputs.
3. **Communication Module**: Supports wireless communication (e.g., LoRa, Wi-Fi, or Zigbee) for remote monitoring and control.
4. **Actuator Module**: Controls pumps or valves for water distribution.
5. **Power Management Unit**: Often powered by solar panels, enhancing sustainability in rural areas.

How to implement the design?

This module simulates a **smart irrigation system** that:

1. Uses a **soil moisture sensor**
2. Contains a **RISC-V processor instance**
3. Uses **control logic** to activate an irrigation pump based on moisture levels

Header and Inputs/Outputs

1. **clk**: System clock (controls timing of all logic).
2. **reset**: Resets the system to a known starting state.
3. **irrigationPump**: Output signal that controls the irrigation pump (1 = ON, 0 = OFF).

Sensor Simulation and Threshold Logic

1. **sensorValue**: The current reading from the simulated soil moisture sensor.
2. **threshold:** A constant value. If the sensorValue is **less than this**, the soil is considered **dry** and the system should turn on the pump.

Signals for Future Processor Integration

These are **placeholders or future expansion** for interaction with the custom RISC-V processor:

1. instrOut: Instruction being fetched.
2. aluResult: ALU computation result.
3. writeBack: Data to be written back to registers.
4. PC: Program counter.
5. pumpSignal: Optional signal for processor to control pump (not yet connected here).

Simulated Soil Moisture Sensor

1. On reset: starts at 500 (moist).
2. Every clock cycle: value **decreases by 1**, simulating drying soil.
3. When it falls below the threshold (400), the system should trigger irrigation..

Custom Processor (Placeholder)

You're instantiating a **custom RISC-V processor** module named Processor.

1. Memory-map sensor and actuator addresses
2. Let the processor make the irrigation decision

Decision Logic (External to Processor)

1. On reset: turns off the pump.
2. Every clock cycle:
    1. If soil moisture (sensorValue) is **less than 400**, it activates the pump.
    2. Otherwise, keeps it off

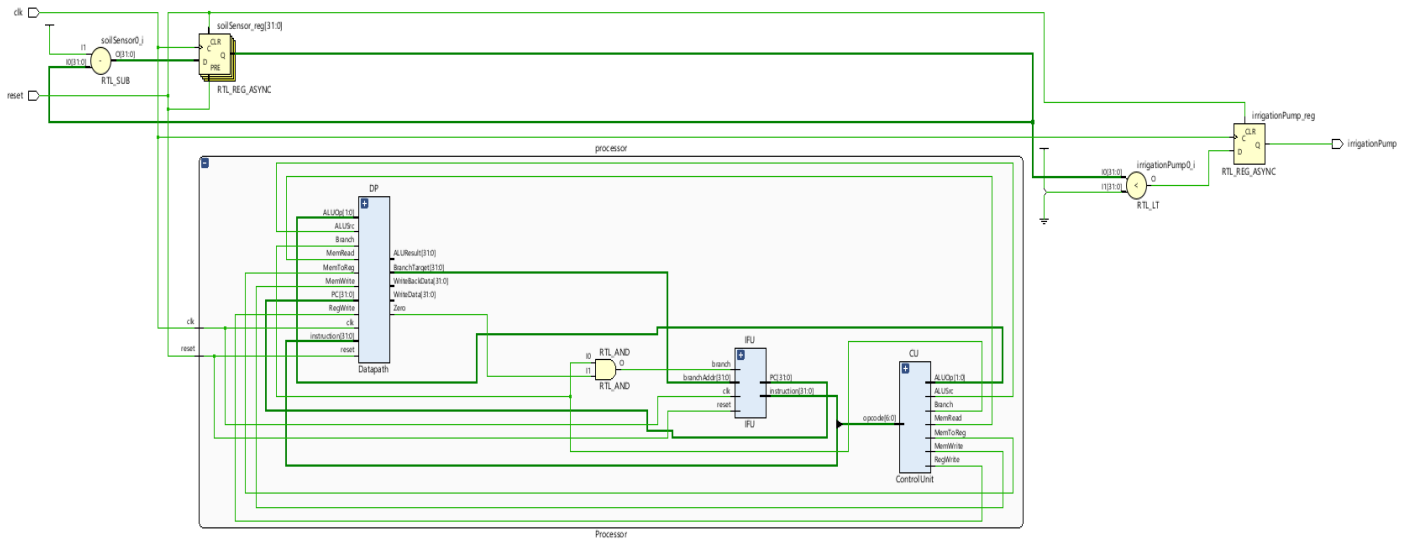


Fig 2. Schematic showing the design of the irrigation system working alongside the RISC-V Processor

Fig 3. Behavioral Simulation of the Smart Irrigation System

This waveform illustrates the behavior of an automated irrigation control system that activates an irrigation pump based on soil moisture readings. The sensor Value and soil Sensor signals represent real-time moisture levels, while the threshold signal defines the minimum acceptable moisture level. At the start of the observed window, both sensor readings are at 300, which is below the threshold of 400. As a result, the pump Signal is set to 1, indicating the pump is turned on, and the irrigation Pump output is asserted to activate irrigation. This shows the system correctly detects dry soil and responds by enabling water flow. Later in the simulation, both the sensor and threshold values rise to 500 and 400 respectively, showing updated conditions. When the soil moisture surpasses the threshold, the pump Signal drops to 0, turning off the pump, which is visible by the deactivation of the irrigation Pump signal. The rest of the signals, such as RegWrite, ALU SRC, and others, are not directly involved in this control logic and remain constant. This simulation verifies that the system correctly evaluates soil conditions and autonomously manages irrigation based on dynamic sensor inputs.

REFERENCES

1. RVCoreP: An optimized RISC-V soft processor of five-stage pipelining- by Miyazaki et al., proposes RVCoreP—a Verilog-implemented softcore with pipelined branch prediction, ALU, and data alignment optimizations achieving ~30% higher performance over VexRiscv on FPGA platforms.
   https://www.reddit.com/r/RISCV/comments/10l9thi/designing_a_risc_v_microprocessor_in_verilog_as_undergrad_project

2. Design and Analysis of RISC V Processor Architecture- by Subhashini et al., presents a 32-bit RISC-V CPU using a hardwired control unit and five-stage pipeline, implemented in Verilog and evaluated in terms of latency, throughput, and resource utilization.
   https://matjournals.net/engineering/index.php/JoMMR/article/view/460

3. Research and design of low-power, high-performance processor based on RISC-V ISA- (IOP 2022), introduces a 3-stage pipelined RV32IM core with static branch prediction optimized for embedded IoT, achieving ~2.38 CoreMark/MHz efficiency.
   https://iopscience.iop.org/article/10.1088/1742-6596/2221/1/012008

4. RISC-V based virtual prototype: An extensible and configurable platform for the system-level- describes a SystemC/TLM-based VP including a 32/64-bit RISC-V core, multi-core support, interrupt controllers, OS compatibility (FreeRTOS/Zephyr), and peripheral models — aimed at early software and system design space exploration.
   https://www.sciencedirect.com/science/article/abs/pii/S1383762120300503

5. Survey of Verification of RISC-V Processors- (Journal of Electronic Testing, 2025) gives a comprehensive overview of verification methodologies for RISC-V implementations, comparing strategies across academic and industrial cores.
   https://link.springer.com/article/10.1007/s10836-025-06169-3

6. BRISC-V: An Open-Source Architecture Design Space Exploration Toolbox- (arXiv 2019) introduces a highly parameterized, modular RTL framework (in synthesizable Verilog) for exploring everything from simple single-cycle cores to multi-core SoCs, caches, memory hierarchies, and network-on-chip topologies. Ideal for quickly prototyping and customizing RISC-V architectures.
   https://arxiv.org/abs/1908.09992

7. Implementation of RISC-V Processor

   Authors: P. Saiprathyusha and C. Chandrasekhar (Sri Venkateswara College of Engineering, Tirupati)
   Published February 2025 in *ITM Web of Conferences*
   Focuses on an optimized pipelined RISC-V processor design aimed at high throughput and energy efficiency. The architecture supports dynamic instruction scheduling, cache optimization, and domain-specific extensions (AI, crypto, signal processing). Designed and tested via Verilog/VHDL and FPGA implementation.
   https://www.researchgate.net/publication/389174344_Implementation_of_RISC-V_Processor

8. PERI: A Posit-Enabled RISC-V Core

   Authors: Sugandha Tiwari, Neel Gala, Chester Rebeiro, V. Kamakoti (IIT Madras)
   Published August 2019 on arXiv
   This work integrates a Posit arithmetic unit into a SHAKTI C-class RISC-V core. It demonstrates how to support Posit (a modern numerical format) via custom RISC-V extensions, offering trade-offs in precision and range, and reports FPGA implementation metrics (~100 MHz operation) on Xilinx Artix-7 hardware.
   https://arxiv.org/abs/1908.01466

9. The **SHAKTI** microprocessor project (IIT Madras RISE group) is a flagship effort funded under India's DIR-V program. It has produced multiple RISC-V CPU variants (E-, C-, I-, M-class) using Bluespec/SystemVerilog, with silicon tape-outs on 22 nm (Intel) and 180 nm (ISRO Semi-Conductor Lab) nodes.
   https://en.wikipedia.org/wiki/SHAKTI_%28microprocessor%29

10. **VEGA Microprocessors**, developed by C-DAC under the India Microprocessor Development Programme, is a portfolio of RISC-V processors targeting IoT, edge, networking, smart NICs, hearing-aids, and future multi-core HPC designs. Estimations include dual-core VEGA in 2023 and Octa-core variants in the mid-2020s.
    https://en.wikipedia.org/wiki/VEGA_Microprocessors

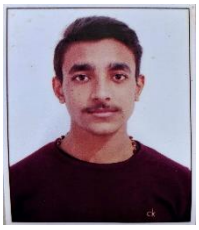
V. CONCLUSION

The design and implementation of a custom RISC-V processor present a powerful, open-source alternative to proprietary instruction sets, enabling cost-effective and flexible hardware development. In this work, the integration of a RISC-V-based processor with a smart irrigation system demonstrates the practical applicability of such processors in real-world, resource-sensitive environments. The processor's modular architecture allows for optimized control logic, low power consumption, and scalable performance, making it an ideal candidate for embedded IoT applications.

By leveraging RISC-V's extensibility and simplicity, we successfully interfaced the processor with sensors, actuators, and memory components required for automated irrigation management. The system is capable of real-time monitoring and decision-making based on environmental inputs such as soil moisture and temperature, thereby enhancing agricultural efficiency and water conservation. This integration not only validates the functionality of the custom RISC-V processor but also highlights its potential in sustainable smart farming technologies.

AUTHORS:

Pratyush Pranjal is currently pursuing his Graduate degree in the department of Electronics and Communication Engineering from Birla Institute of Technology, Mesra, Ranchi - 835215
Corresponding author
Email: btech10024.22@bitmesra.ac.in



Vansh Tak is currently pursuing his Graduate degree in the department of Electronics and Communication Engineering from Birla Institute of Technology, Mesra, Ranchi – 835215.
Corresponding author
Email: btech10018.22@bitmesra.ac.in

Aryan Singh is currently pursuing his Graduate degree in the department of Electrical and Electronics Engineering from Birla Institute of Technology, Mesra, Ranchi - 835215
Corresponding author
Email: btech10019.22@bitmesra.ac.in

Dr. Vijay Nath received his BSc degree in Physics from DDU University Gorakhpur, India in 1998 and PG Diploma in Computer Networking from MMM University of Technology Gorakhpur, (UP) India in 1999 and an MSc in Electronics from DDU University Gorakhpur, India, in 2001. He received his PhD in Electronics from Dr. Ram Manohar Lohiya Avadh University Ayodhya (UP), in collaboration with CEERI Pilani (Raj), India, in 2008. His areas of interest include ultra-low-power temperature sensors for missile applications, microelectronics engineering, mixed-signal design, application-specific integrated circuit design, embedded system design, cardiac pacemakers, the Internet of Things, artificial intelligence and machine learning, and computational intelligence.
Email: vijaynath@bitmesra.ac.in