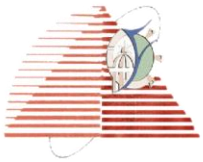


FACULTY OF ENGINEERING – FAYOUM UNIVERSITY
Electrical Engineering Department

Pipelined 32-bit RISC Processor Implementation

Course Title:
CEE210 / CEE216 – Computer Engineering

Academic Term:
Spring 2025

Project Supervisors:
Asst. Prof. Dr. Gihan Naguib
Eng. Jihad Awad

Date of Submission:
May 9, 2025

Phase1 Video **GitHub** **Phase2 Video**

- **Abdullah Mostafa Gomaa**
- **Abdelrahman Zaki Ibrahim**
- **Moataz Mohamed Ali**

Table of Contents

Phase 1: Single-Cycle Processor

1. Introduction

This document presents the design and verification of a 32-bit single-cycle RISC processor based on a MIPS-like architecture.

Implemented in Logisim, the processor features 31 general-purpose registers (R1–R31), a hardwired zero register (R0), and supports R-type, I-type, and SB-type instruction formats.

Each instruction completes execution within a one clock cycle.

Objectives

- Build a working single-cycle processor with full ISA support.
- Validate correct functionality through test programs.

2. Instruction Set Architecture (ISA)

2.1 Registers

- 32 Registers (R0–R31):
    - **R0**: Hardwired to 0 (read-only).
    - **R1–R31**: General-purpose registers.

2.2 Instruction Formats

In the designed processor, each instruction is 32 bits wide and follows one of three formats: **R-type**, **I-type**, or **SB-type**.
The bit numbering starts from 0 (least significant bit) to 31 (most significant bit), and the fields are allocated as described below:

R-type Format

Bits	Field	Description
0–5	Opcode (6 bits)	Specifies the main operation category
6–10	Destination Register (Rd, 5 bits)	Destination register number for the result
11–15	Source Register 1 (RS1, 5 bits)	First source register operand
16–20	Source Register 2 (RS2, 5 bits)	Second source register operand
21–31	Function Code (Func, 11 bits)	Specifies the exact operation to be performed

Usage:
Used for arithmetic, logical, and comparison operations between two registers.

I-type Format (Immediate Type)

Bits	Field	Description
0–5	Opcode (6 bits)	Specifies the instruction type
6–10	Destination Register (Rd, 5 bits)	Destination register number
11–15	Source Register 1 (RS1, 5 bits)	Source register operand
16–31	Immediate Value (Imm16, 16 bits)	16-bit immediate constant value

Usage:
Used for operations involving a register and an immediate constant, such as ADDI, ANDI, LW, etc.

SB-type Format (Store/Branch Type)

Bits	Field	Description
0–5	Opcode (6 bits)	Specifies the branch or store operation
6–10	Source Register 1 (RS1, 5 bits)	First source register operand
11–15	Source Register 2 (RS2, 5 bits)	Second source register operand
16–26	Immediate Upper (ImmU, 11 bits)	Upper part of the branch/store offset
27–31	Immediate Lower (ImmL, 5 bits)	Lower part of the branch/store offset

Usage:
Used for memory store instructions (e.g., SW) and conditional branch instructions (e.g., BEQ, BNE, BLT, BGE), where a signed offset is constructed from the immediate fields.

2.3 Memory

- **Instruction Memory**: 2^{20} word-addressable locations.
- **Data Memory**: Word-addressable, accessed by LW and SW instructions.

2.4 Supported Instructions

- **Arithmetic Instructions**: ADD, SUB, MUL
- **Logical Instructions**: AND, OR, XOR, NOR
- **Shift and Rotate Instructions**: SLL, SRL, SRA, ROR
- **Comparison Instructions**: SLT, SLTU, SEQ
- **Immediate Instructions**: ADDI, SLTI, XORI, ORI, ANDI, SET, SSET
- **Memory Access Instructions**: LW, SW
- **Branch Instructions**: BEQ, BNE, BLT, BGE, BLTU, BGEU

- **Jump Instruction**: JALR

3. Digital Circuit Design

3.1 Register File

Overview

The Register File is a critical component of the Datapath in a 32-bit RISC processor. It contains 32 general-purpose registers, each 32 bits wide, labeled from R0 to R31. This unit supports two read ports and one write port, enabling simultaneous read and write operations in a single clock cycle.

Ports and Signals

- **RA:** Read address for BusA.
- **RB:** Read address for BusB.
- **RW:** Write address.
- **BusA:** Data output from RA.
- **BusB:** Data output from RB.
- **BusW:** Data input for writing to RW.
- **RegWr:** Register write enable.
- **CLK:** Clock input.

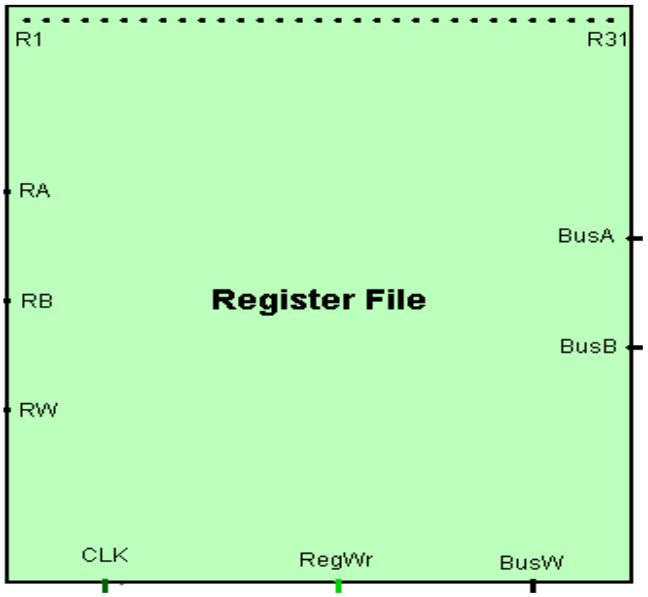


Figure 1: Register File

3.1.1 Functionality

- **Read Operation**:
  On every cycle, the values of the registers specified by RA and RB are output through BusA and BusB, respectively.

- **Write Operation**:
  If RegWr is asserted (1) on the rising edge of the clock, the value on BusW is written into the register specified by RW.

3.1.2 Design Considerations

- Writes occur only on the rising edge of the clock and only if RegWr is enabled.

- Multiple simultaneous read operations increase instruction throughput and are essential for R-type instructions that read two registers.

3.1.3 Key Components:

Decoder: Selects the destination register during write operations.

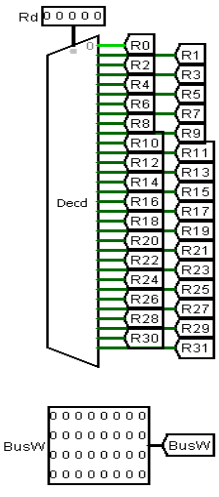


Figure 2: Register File Decoder 1

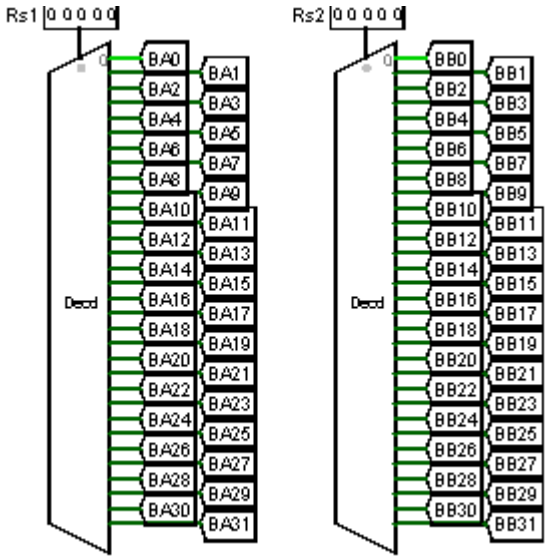


Figure3: Register File Decoder 2

Tri-State Buffers: Enable concurrent read operations from two registers.

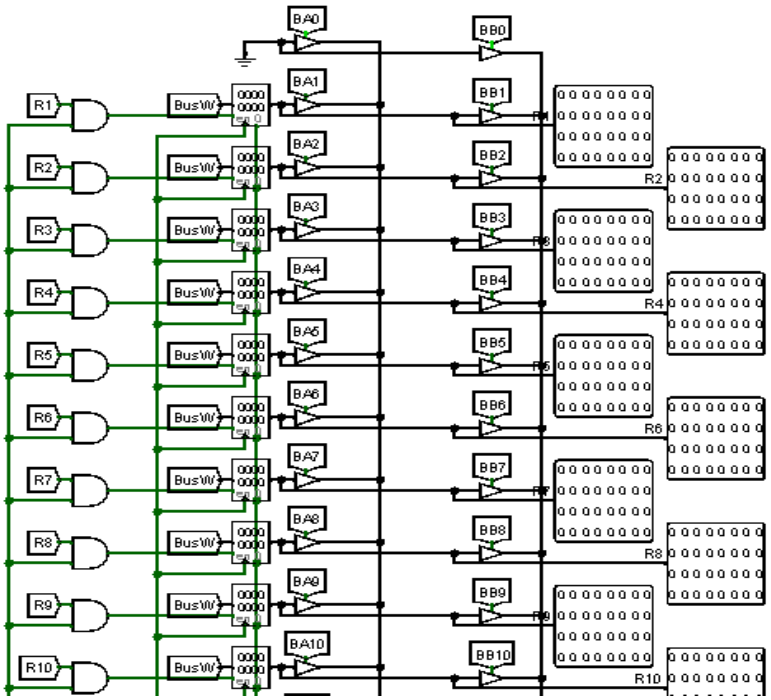


Figure 4: Register File Tri-State Buffers

Debugging Outputs: pins to monitor register values.

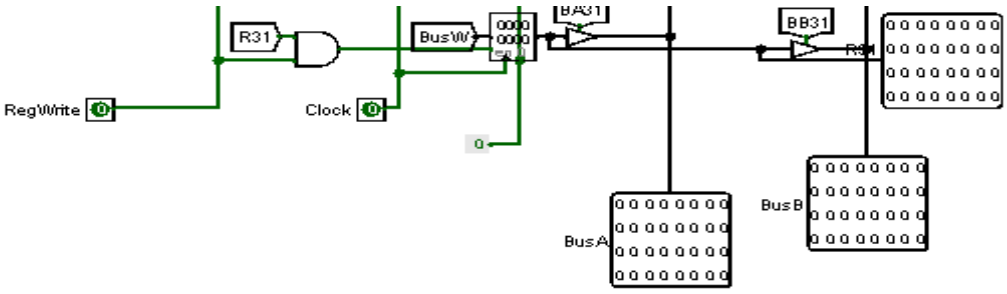


Figure 5: Register File Outputs

3.2 Program Counter Unit

The Program Counter (PC) Unit is a critical component that manages the flow of instruction execution by tracking and updating the address of the next instruction to be fetched. It ensures sequential execution or redirects program flow during branches and jumps. This unit integrates the following components:

20-bit Program Counter Register

- Holds the 20-bit address of the current instruction in the instruction memory.
- Default behavior: Increments by 1 to point to the next sequential instruction (PC = PC + 1).
- Updated during branches, jumps, or exceptions to redirect program flow.

Instruction Memory (ROM)

- Stores the program instructions as 32-bit words.
- The PC value serves as the address input to fetch the next instruction.
- Organized as a word-addressable memory with 2^{20} entries.

PC Adder

- A dedicated adder that increments the PC by 1 (PC + 1) to generate the address of the next sequential instruction.
- Ensures smooth progression through the instruction sequence when no branches or jumps occur.

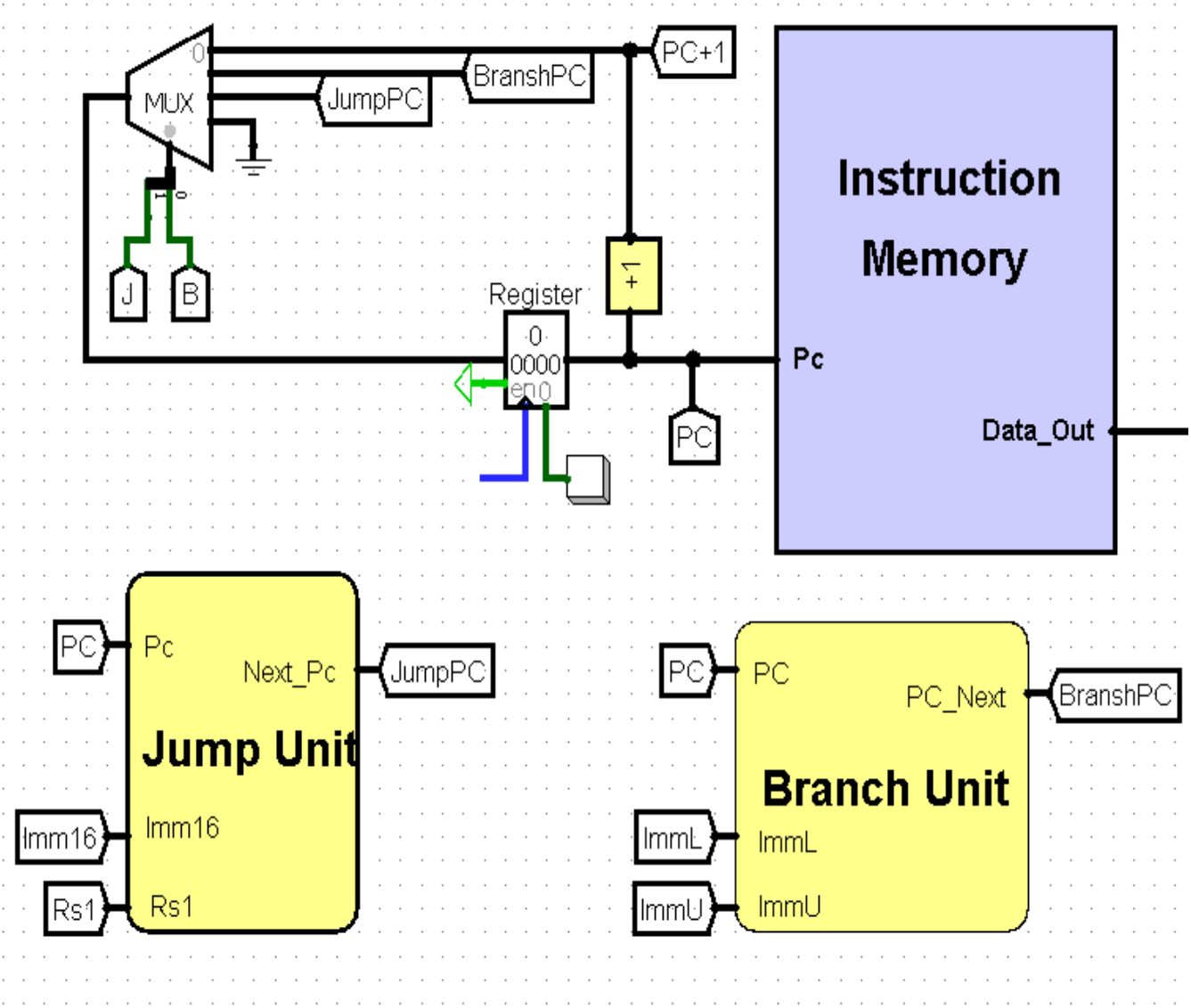


Figure 6: Program Counter Unit

3.2.1 Jump Address Calculation

Manages indirect jump operations (JALR).

- Jump Target Calculation:
  Target Address = RS1 + sign-extend (Imm16)
- Return Address Handling:
  Stores PC + 1 in the destination register (Rd) to support subroutine linking.
- Control Signal:
  A JumpOp (1-bit) signal activates the jump logic when Opcode = 15 (JALR).

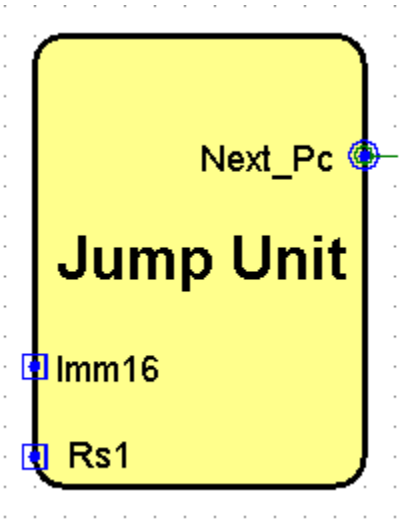


Figure 7: Jump Unit

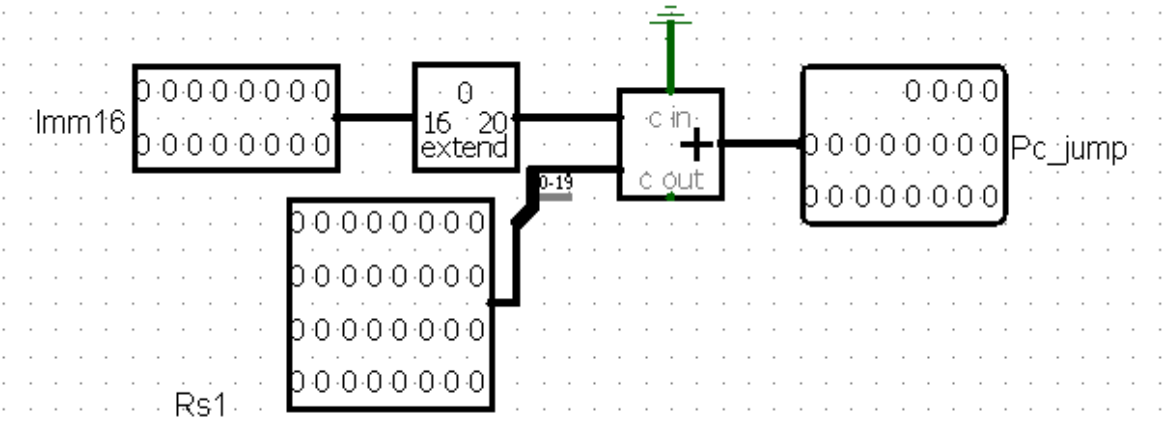


Figure 8: Jump Unit Internal Design

3.2.2 Branch Address Handling

Executes conditional branches such as BEQ, BNE, BLT, etc.

- Condition Evaluation:
  Compares RS1 and RS2 using internal flags (e.g., Zero, SLT, SLTU).
- Branch Target Calculation:
  Target Address = PC + sign-extend { ImmU , ImmL }
- Branch Decision Logic:
  Controlled by a BranchOp (2-bit) signal that selects the branch condition type (e.g., 00 for BEQ, 01 for BNE).

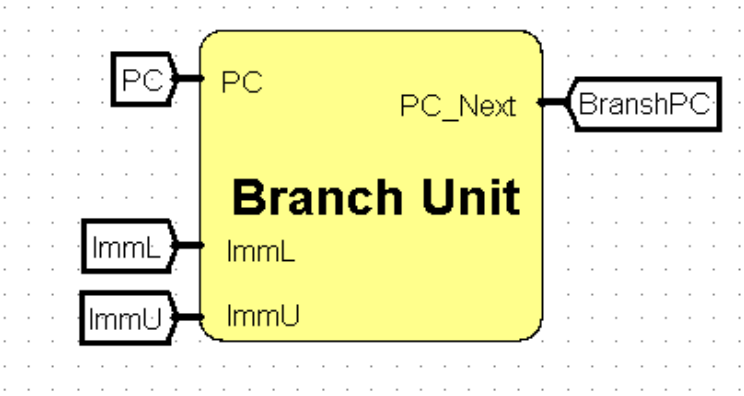


Figure 9: Branch Unit

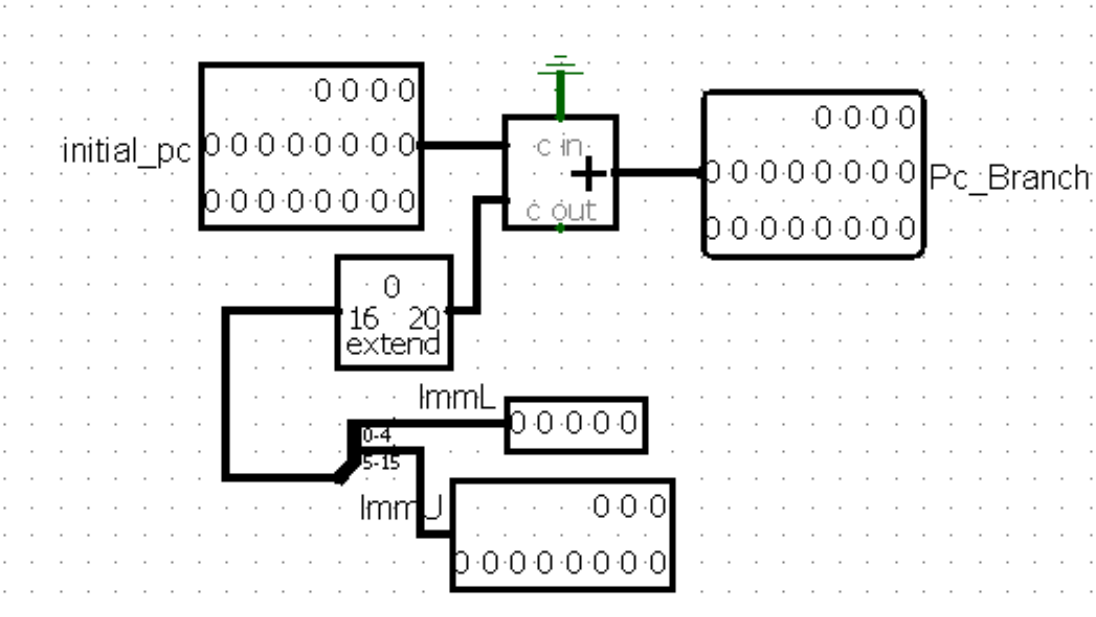


Figure 10: Branch Unit Internal Design

3.2.3 Sequential Addressing

the absence of branch or jump operations, the PC is automatically incremented:

- PC = PC + 1

This mechanism supports linear execution of instructions without interruption.

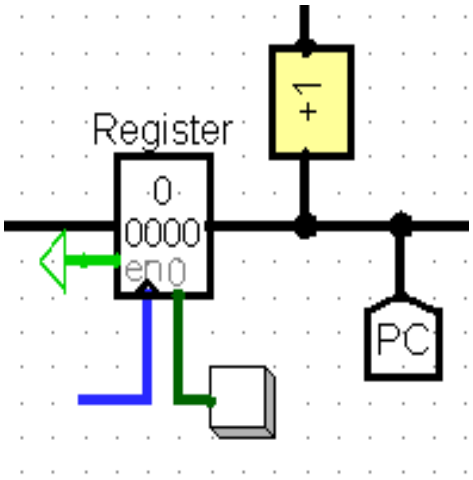


Figure 2: Sequential Addressing

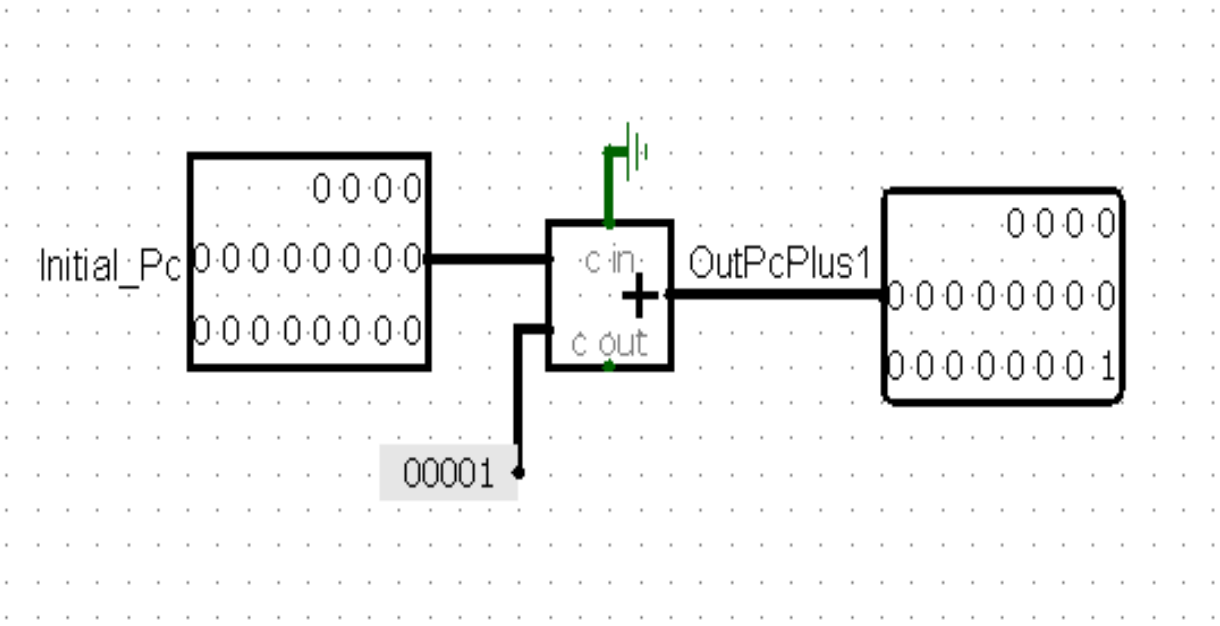


Figure 3: Sequential Addressing Internal Design

3.3 Instruction Memory (ROM)

3.3.1 Overview

The Instruction Memory is a word-addressable ROM that stores the processor's instructions. Each address contains a 32-bit instruction. The ROM outputs the instruction at the address provided by the Program Counter (PC). Essential for instruction fetching and program control, it is preloaded with instructions.

3.3.2 Ports and Signals

- PC (Input): Instruction address from the Program Counter.
- Data_Out (Output): 32-bit instruction from memory.

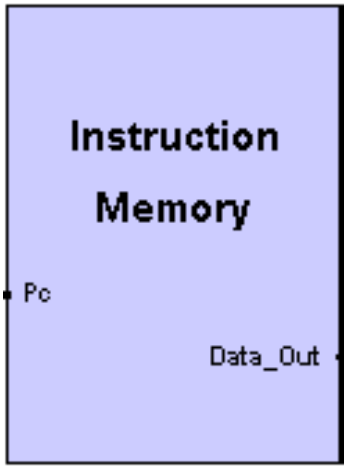


Figure 4: Instruction Memory

3.3.3 Functionality

Fetch: The PC provides the address; the ROM outputs the 32-bit instruction.
Word Addressing: The PC increments by 1 to fetch the next instruction.
Initialization: Instructions are preloaded into the ROM in Logisim.

3.3.4 Design Considerations

- Word Alignment: PC increments by 1 for each 32-bit word.
- Start: Execution begins at address 0.
- Size: Up to 2^{20} instructions for a 20-bit PC.

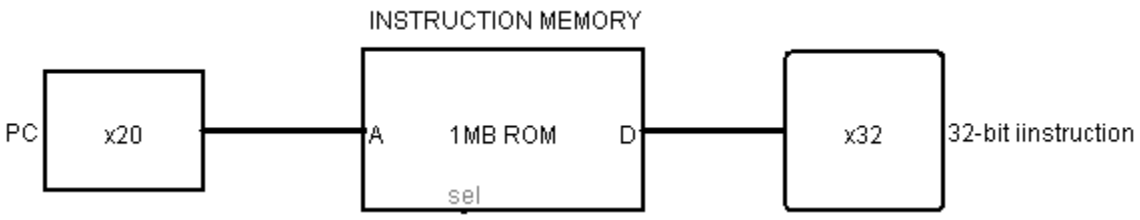


Figure 5: Instruction Memory Internal Design

3.4 Instruction Splitter

3.4.1 Overview

The **Instruction Splitter** is a combinational module that decodes a 32-bit instruction into its key fields. It supports **R-type**, **I-type**, and **SB-type** formats, providing outputs that drive the Control Unit, Register File, and ALU.

3.4.2 Ports and Signals

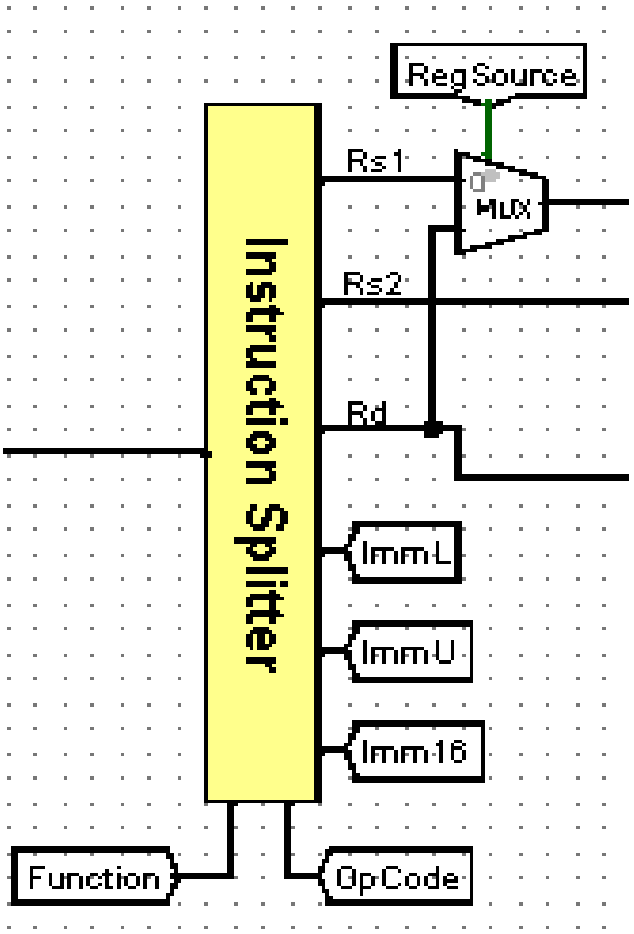


Figure 6: Instruction Splitter

- **RS1 (Output, 5 bits)**: Bits 11–15; first source register.
- **RS2 (Output, 5 bits)**: Bits 16–20; second source register (R/SB-type).
- **Rd (Output, 5 bits)**: Bits 6–10; destination register (R/I-type).
- **ImmL (Output, 5 bits)**: Bits 27–31; lower part of SB-type immediate.
- **ImmU (Output, 11 bits)**: Bits 16–26; upper part of SB-type immediate.
- **Imm16 (Output, 16 bits)**: Bits 16–31; immediate for I-type.
- **Opcode (Output, 6 bits)**: Bits 0–5; operation type.
- **Function (Output, 11 bits)**: Bits 21–31; ALU operation (R-type).

3.4.3 Functionality

The splitter extracts instruction fields based on the format:

- **R-Type**: Opcode (0–5), Rd (6–10), RS1 (11–15), RS2 (16–20), Function (21–31).
- **I-Type**: Opcode (0–5), Rd (6–10), RS1 (11–15), Imm16 (16–31).
- **SB-Type**: Opcode (0–5), RS1 (6–10), RS2 (11–15), ImmU (16–26), ImmL (27–31).

It enables precise control signal generation and correct operand routing during instruction execution.

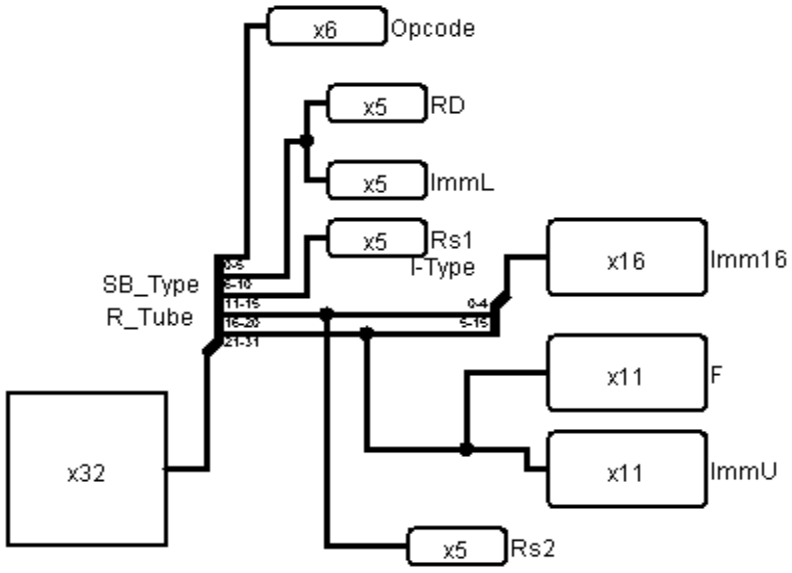


Figure 7: Instruction Splitter Internal Design

3.5 Control Unit

The Control Unit acts as the processor's central decision-making element. It interprets the instruction Opcode and generates the necessary control signals to manage data flow, control ALU input selection, coordinate memory access, and control program sequencing.

3.5.1 Input Signals

- **Opcode (6 bits):**
  Specifies the high-level type of operation, such as arithmetic, branch, or memory access.

Note: The Function field, used to select specific ALU operations for R-type instructions, is handled separately by the ALU Control Unit and is discussed in Section 5.

3.5.2 Output Signals

Register and Memory Control (Storage Control Unit):

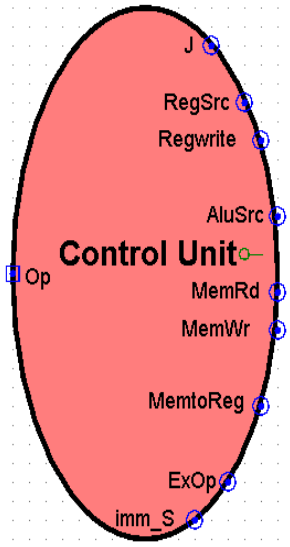


Figure 8: Control Unit

- **RegSrc:** Selects the register file source for writing.
  `0`: Write from Rs1.
  `1`: Write from Rd.
- **RegWrite:** Enables or disables writing to the register file.
  `1`: Enable writing.
  `0`: Disable writing.
- **MemoryRead (MemRd):** Enables reading from the data memory.
  `1`: Read operation enabled.
  `0`: No read.
- **MemoryWrite (MemWr):** Enables writing to the data memory.
  `1`: Write operation enabled.
  `0`: No write.
- **MemtoReg:** Selects the data source for the write-back to the register file:
  `00`: ALU output.
  `01`: Data memory output.
  `10`: PC + 1 (for JALR instruction).
  `11`: No value written back.

ALU Input Control:

- **ALUSrc:** Selects the source for the second ALU operand.
  `0`: Register Rs2 value.
  `1`: Extended immediate value.
- **Extender_Op:** Controls how the immediate value is extended to 32 bits.
  `0`: Zero-extension (filling upper bits with zeros).
  `1`: Sign-extension (filling upper bits with the sign bit).

Immediate Selection (Imm_Select):

- **Imm16:** Selected for I-type instructions such as ADDI, LW, and JALR.
- **{ImmU, ImmL}:** Selected for SB-type instructions such as BEQ, BNE, BLT, and SW.

3.5.3 Internal Design

The Control Unit is organized into four specialized sub-blocks, each responsible for a specific aspect of control signal generation:

- **Storage Control Unit:**
  Generates register and memory-related control signals based on the decoded Opcode.
- **ALU Input Control Unit:**
  Selects the appropriate operand sources for the ALU and determines the extension type for immediate values.
- **Immediate Selection Unit (Imm_Select):**
  Chooses the correct immediate value format based on the instruction type.
- **Jump Control Unit:**
  Generates control signals related to branching, jumping, and program counter updates.

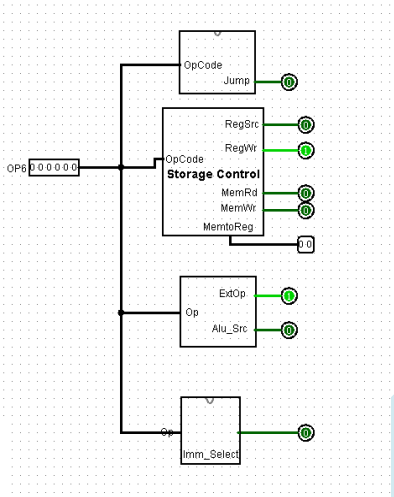


Figure 9: Control Unit Internal Design Blocks

3.5.4 Internal Blocks

Each internal block of the Control Unit is

implemented as a separate module, as detailed below:

Storage Control Unit

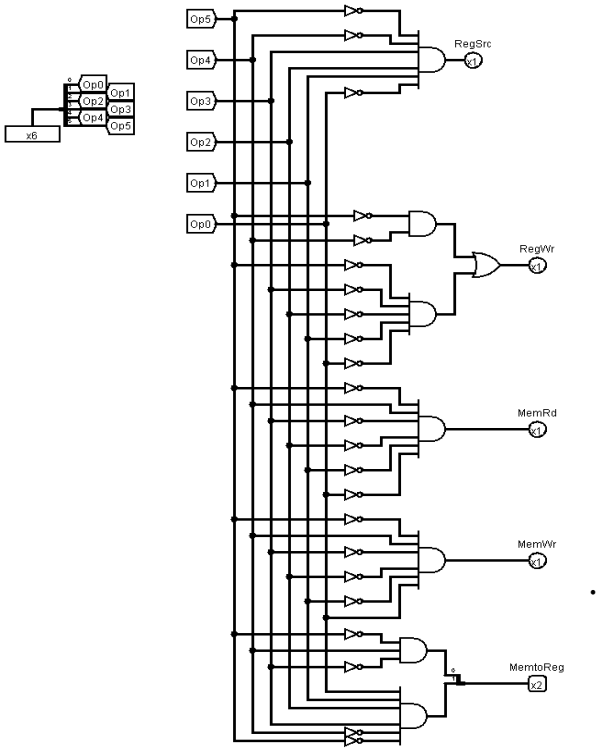


Figure 11: Storage Control Internal Design

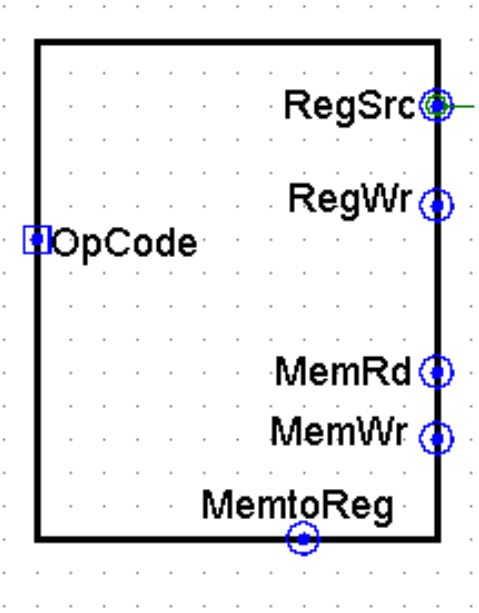


Figure 10: Storage Control Block

ALU Input Control Unit

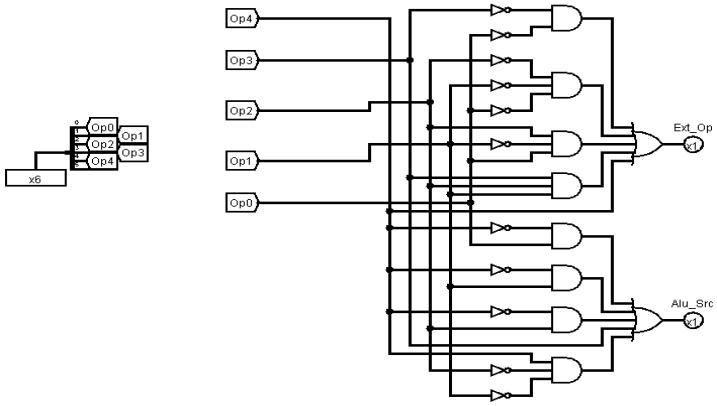


Figure 13: ALU Input Control Internal Design

Figure 12: ALU Input Control Block

Immediate Selection Unit

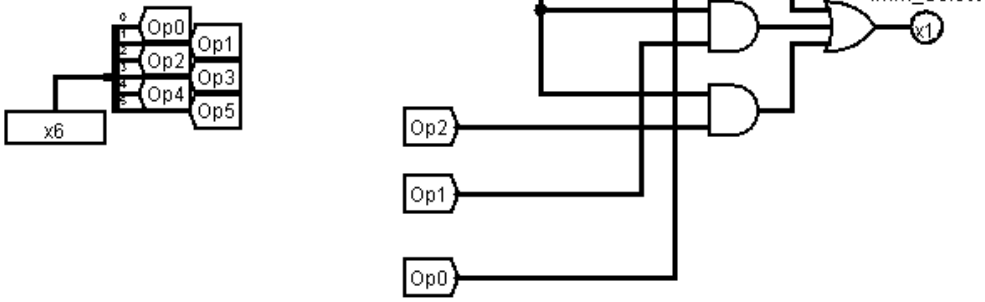


Figure 15: Immediate Selection Internal Design

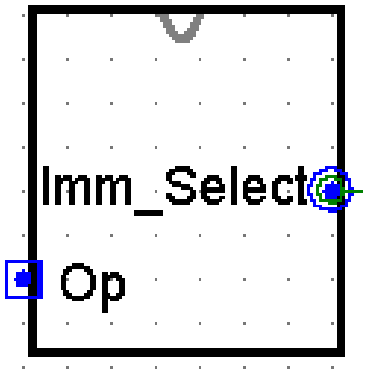


Figure 14: Immediate Selection Block

Jump Control Unit

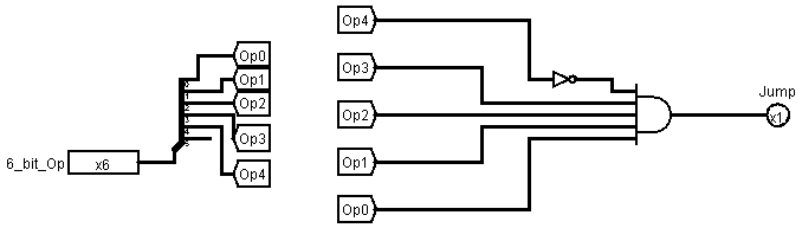


Figure 17: Jump Control Internal Design

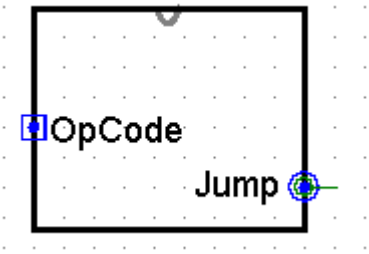


Figure 16: Jump Control Block

Figure 24: Jump Control Unit responsible for generating jump control signal.

3.5.5 Control Signal Table

The control signals generated by the Control Unit vary depending on the instruction type. A detailed truth table is used to define the exact signal values for each instruction. Key highlights include:

- **R-type instructions:** Perform register-register ALU operations, writing the ALU result back to Rd.
- **I-type instructions:** Use an immediate value as the second ALU operand, with appropriate extension.
- **Memory instructions (LW, SW):** Use address calculations for reading or writing data.
- **Branch instructions:** Evaluate register comparisons and update the program counter conditionally.
- **Jump instructions (JALR):** Perform indirect jumps with optional return address saving.

Instruction	RegSource	RegWr	Extender_Op	ALU_Src	Mem_Rd	Mem Wr	Mem_to_Reg	PC_Src	Branch	Jump	imm_select
R_Type	0	1	x	0	0	0	00	11	0	0	x
SLLI	0	1	x	1	0	0	00	11	0	0	x
SRLI	0	1	x	1	0	0	00	11	0	0	x
SRAI	0	1	x	1	0	0	00	11	0	0	x
RORI	0	1	x	1	0	0	00	11	0	0	x
ADDI	0	1	1	1	0	0	00	11	0	0	0
SLTI	0	1	1	1	0	0	00	11	0	0	0
SLTIU	0	1	0	1	0	0	00	11	0	0	0
SEQI	0	1	1	1	0	0	00	11	0	0	0
XORI	0	1	0	1	0	0	00	11	0	0	0
ORI	0	1	0	1	0	0	00	11	0	0	0
ANDI	0	1	0	1	0	0	00	11	0	0	0
NORI	0	1	0	1	0	0	00	11	0	0	0
SET	x	1	1	1	0	0	00	11	0	0	0
SSET	x	1	x	1	0	0	00	11	0	0	0
JALR	0	1	1	1	0	0	10	10	0	1	0
LW	0	1	1	1	1	0	01	11	0	0	0
SW	0	0	1	1	0	1	x	11	0	0	1
BEQ	0	0	1	0	0	0	x	01	1	0	1
BNE	0	0	1	0	0	0	x	01	1	0	1
BLT	0	0	1	0	0	0	x	01	1	0	1
BGE	0	0	1	0	0	0	x	01	1	0	1
BLTU	0	0	1	0	0	0	x	01	1	0	1
BGEU	0	0	1	0	0	0	x	01	1	0	1

Table 1: Control Signals Table

3.5.6 ALU Control Signal

During the design of the control circuits for the Logic Unit, Shift Unit, and Set Unit, it was observed that certain operations such as **XOR** (in Logic), **SLL** (in Shift), and **SET** (in Set) were not automatically included in the internal control signal calculations.

These instructions did not naturally fit into the simple selection logic designed for other operations.

Therefore, it was necessary to **manually generate separate control signals** for these specific instructions.

The individual signals for **XOR**, **SLL**, and **SET** were extracted based on the ALUControl input, and then **combined using OR gates** with the main control signal inside each respective unit.

This ensures that ALU correctly recognizes and executes these operations alongside the others.

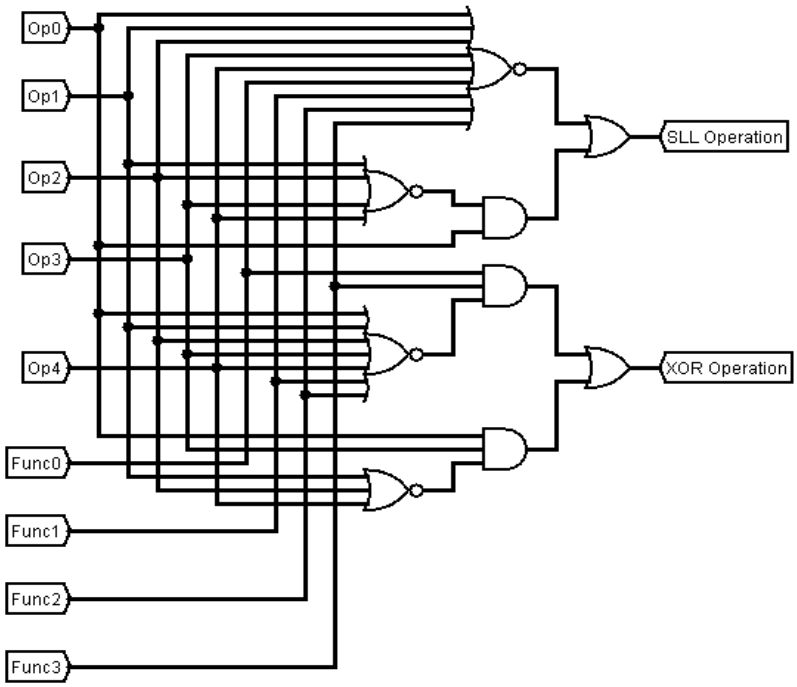


Figure 18: SLL and XOR Operations signal circuit

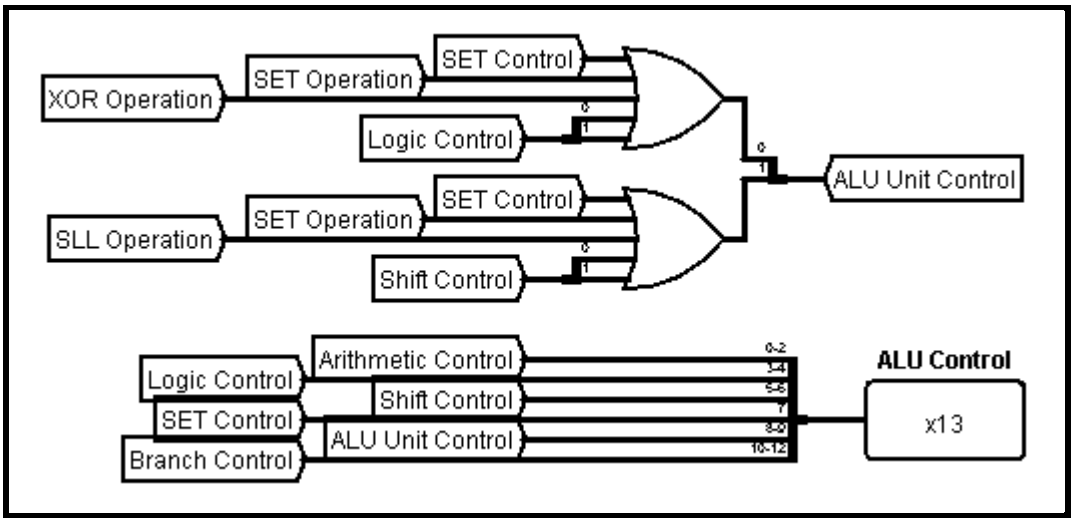


Figure 19: ALU Control signal circuit

The corresponding mapping table is provided below:

Instruction	Function (0-3 bits)	Opcode (0-4 bits)	ALU unit	ALU Control
ADD	0100	00000	Arithmetic	00
SUB	0101	00000	Arithmetic	00
SLT	0110	00000	Arithmetic	00
SLTU	0111	00000	Arithmetic	00
SEQ	1000	00000	Arithmetic	00
MUL	1101	00000	Arithmetic	00
ADDI	-	00101	Arithmetic	00
SLTI	-	00110	Arithmetic	00
SLTIU	-	00111	Arithmetic	00
SEQI	-	01000	Arithmetic	00
JALR	-	01111	Arithmetic	00
LW	-	10000	Arithmetic	00
SW	-	10001	Arithmetic	00
XOR	1001	0000	Logic	01
OR	1010	0000	Logic	01
AND	1011	0000	Logic	01
NOR	1100	0000	Logic	01
XORI	-	1001	Logic	01

ORI	-	1010	Logic	01
ANDI	-	1011	Logic	01
NORI	-	1100	Logic	01
SLL	00	000	Shift	10
SRL	01	000	Shift	10
SRA	10	000	Shift	10
ROR	11	000	Shift	10
SLLI	-	001	Shift	10
SRLI	-	010	Shift	10
SRAI	-	011	Shift	10
RORI	-	100	Shift	10
SET	-	01101	SET	11
SSET	-	01110	SET	11

Table 2: ALU Control Signals Table

3.5.7 ALU Control Circuit

ALU Block

Inputs:

- **Opcode** (6-bit)
- **Function** (11-bit)

Outputs:

- **ALU Control Signal** (13-bit)

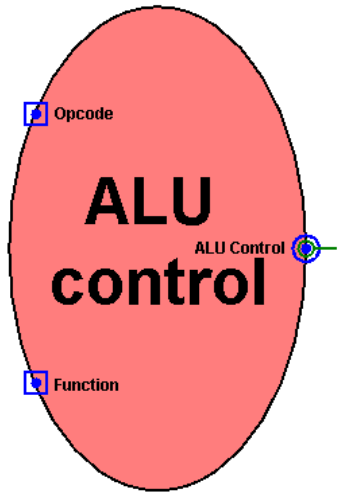


Figure 20: ALU Control Block

The **ALU Control Circuit** is responsible for generating the 13-bit **ALUControl** signal based on the instruction type and function fields provided by the main control unit.
This circuit decodes the **Opcode** and **Function** fields from the instruction and determines the required ALU operation by setting the appropriate control code.
The generated **ALUControl** signal is then used inside the ALU to select between the outputs of the Arithmetic, Logic, Shift, and Set Units, and to handle branch operations correctly.

The complete ALU Control Circuit is shown below:

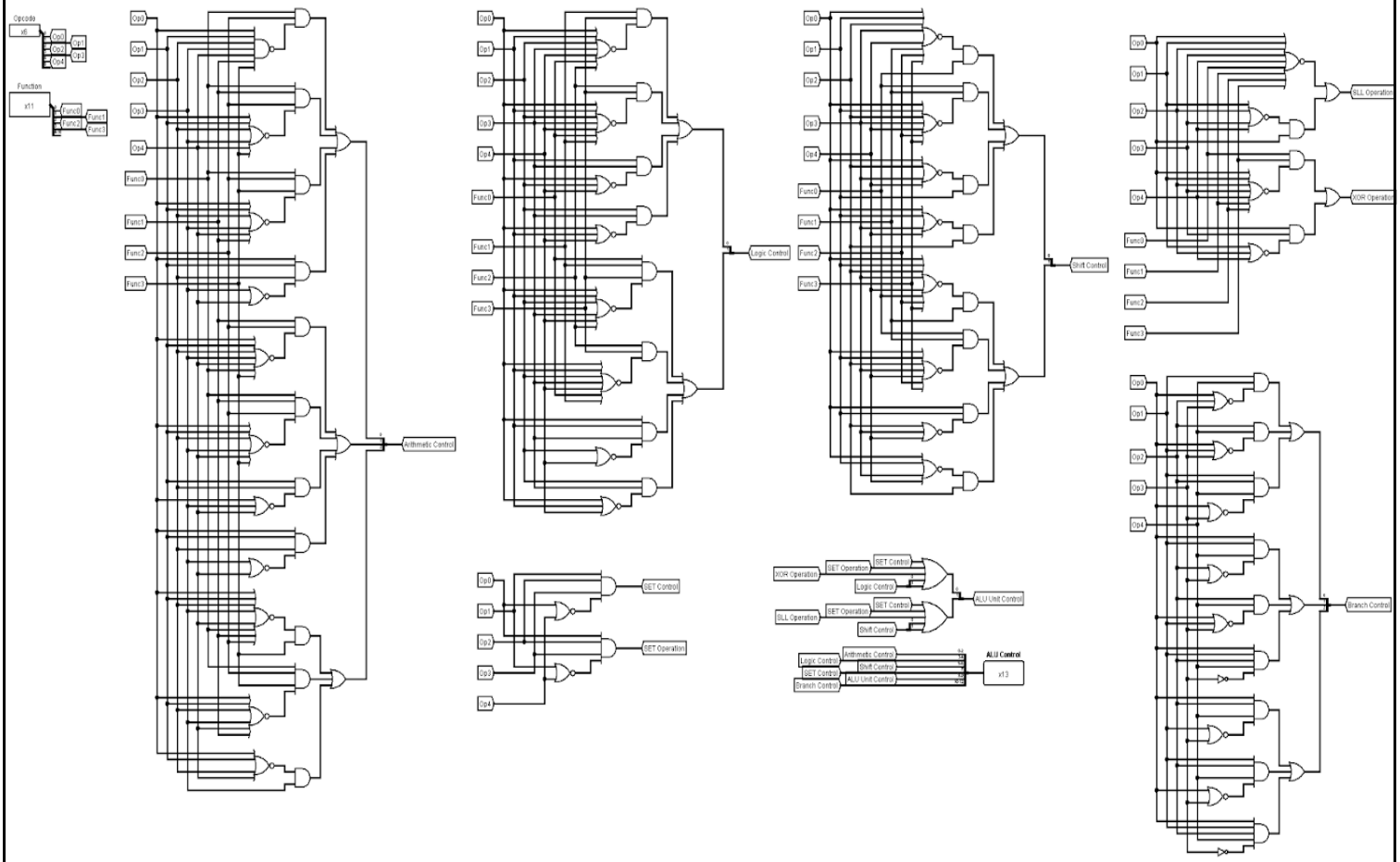


Figure 21: ALU Control circuit

3.6 Bit Extender

The Bit Extender plays a critical role in the processor's datapath by enabling correct manipulation of immediate values from various instruction formats. It ensures that smaller immediate fields are accurately extended to 32-bit operands required for arithmetic, memory addressing, and control operations. The extension mode—signed or unsigned—is dynamically selected based on the instruction type through dedicated control signals.

The Bit Extender accepts the following input sources:

- **Imm16:** A 16-bit immediate field used in I-type instructions.
- **ImmU || ImmL:** A concatenated 16-bit immediate used in SB-type branch instructions.
- **PC + 1:** A 20-bit program counter value, extended to 32 bits for sequential addressing.

Internally, the extender uses two multiplexers:

- The first multiplexer selects between the immediate sources (Imm16 or {ImmU || ImmL}) depending on instruction format.
- The second multiplexer determines the extension type (sign or zero extension), ensuring correct operand preparation for downstream units.

3.6.1 Signed Extension

When operating in signed extension mode, the Bit Extender replicates the most significant bit (MSB) of the immediate value across the higher-order bits to fill 32 bits. This preserves the signed magnitude of values, allowing correct interpretation during operations like addition, subtraction, comparisons, and memory address calculations.

Signed extension is applied in arithmetic instructions (ADDI, SLTI, SET, LW, SW) and all conditional branch instructions (BEQ, BNE, BLT, etc.).

3.6.2 Unsigned Extension

In unsigned extension mode, the higher-order bits are filled with zeros, treating the immediate as a purely positive quantity. This is particularly important for logical operations where sign information is irrelevant, such as in XORI, ORI, ANDI, and NORI.

By providing both signed and unsigned extension capabilities, the Bit Extender allows the processor to efficiently support a versatile and compact instruction set architecture.

Figure 3.6.1: Bit Extender Circuit Diagram
This figure illustrates the internal design of the Bit Extender, showing immediate selection, extension operations, and 32-bit output generation for the Datapath.

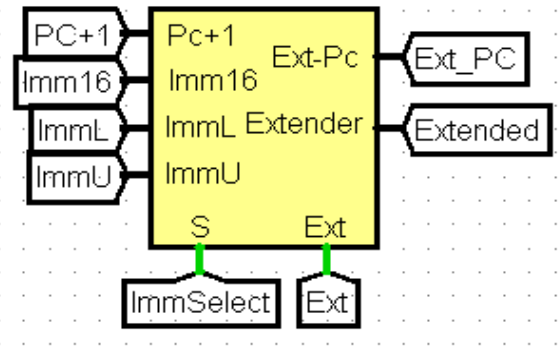


Figure 22: Bit Extender

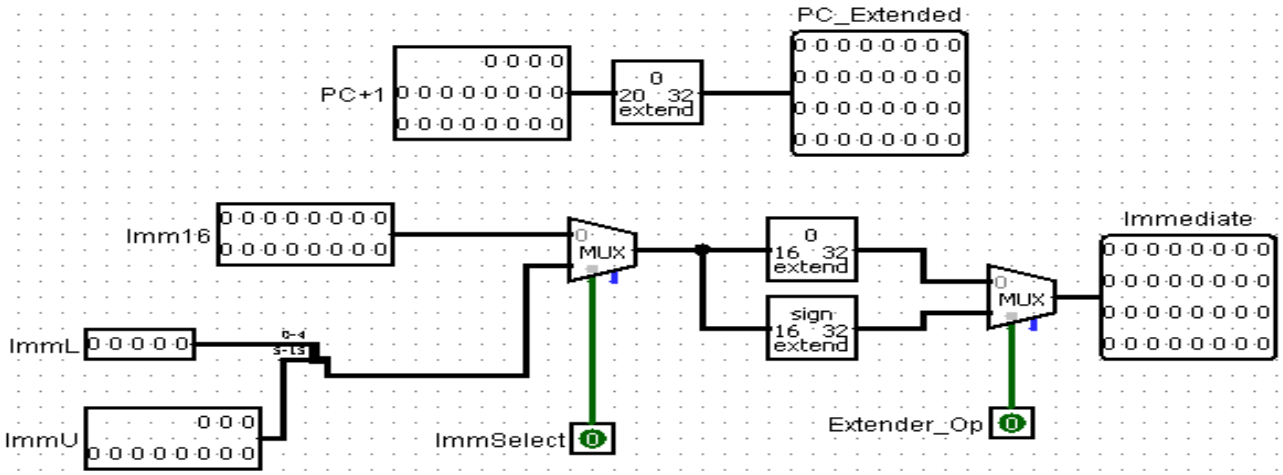


Figure 23: Bit Extender Internal Design

3.7 ALU

The Arithmetic Logic Unit (**ALU**) is a central component of the processor responsible for executing all arithmetic, logical, and branch condition operations. In this project, the ALU was designed to support **R-type**, **I-type**, and **SB-type** instructions based on a custom instruction set architecture (ISA). These include operations such as addition, subtraction, multiplication, logic gates, shifts, comparisons, and set instructions.

The ALU receives two **32-bit** inputs (commonly referred to as Rs and Rt), along with a 20-bit **ALU Control** signal that determines the specific operation. The output includes a 32-bit result, a **Branch** signal used for branching instructions, and an **Overflow** flag.

ALU Block have:

- inputs:
    1. Rs (first source 32-bit)
    2. Rt (second source 32-bit)
    3. ALU Control Signal (13-bit)

- outputs:
    1. Output (32-bit)
    2. Branch Signal (1-bit) according to branch instruction
    3. overflow Signal (1-bit)

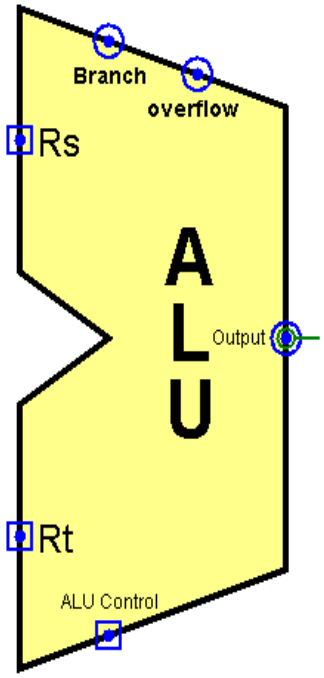


Figure 24: ALU inputs and outputs Block

By separating ALU functions into 4 types (Arithmetic, Logic, Shift, SET) and Branch signal, our complex problem **Building ALU** became more separable and can focus in each part separately.

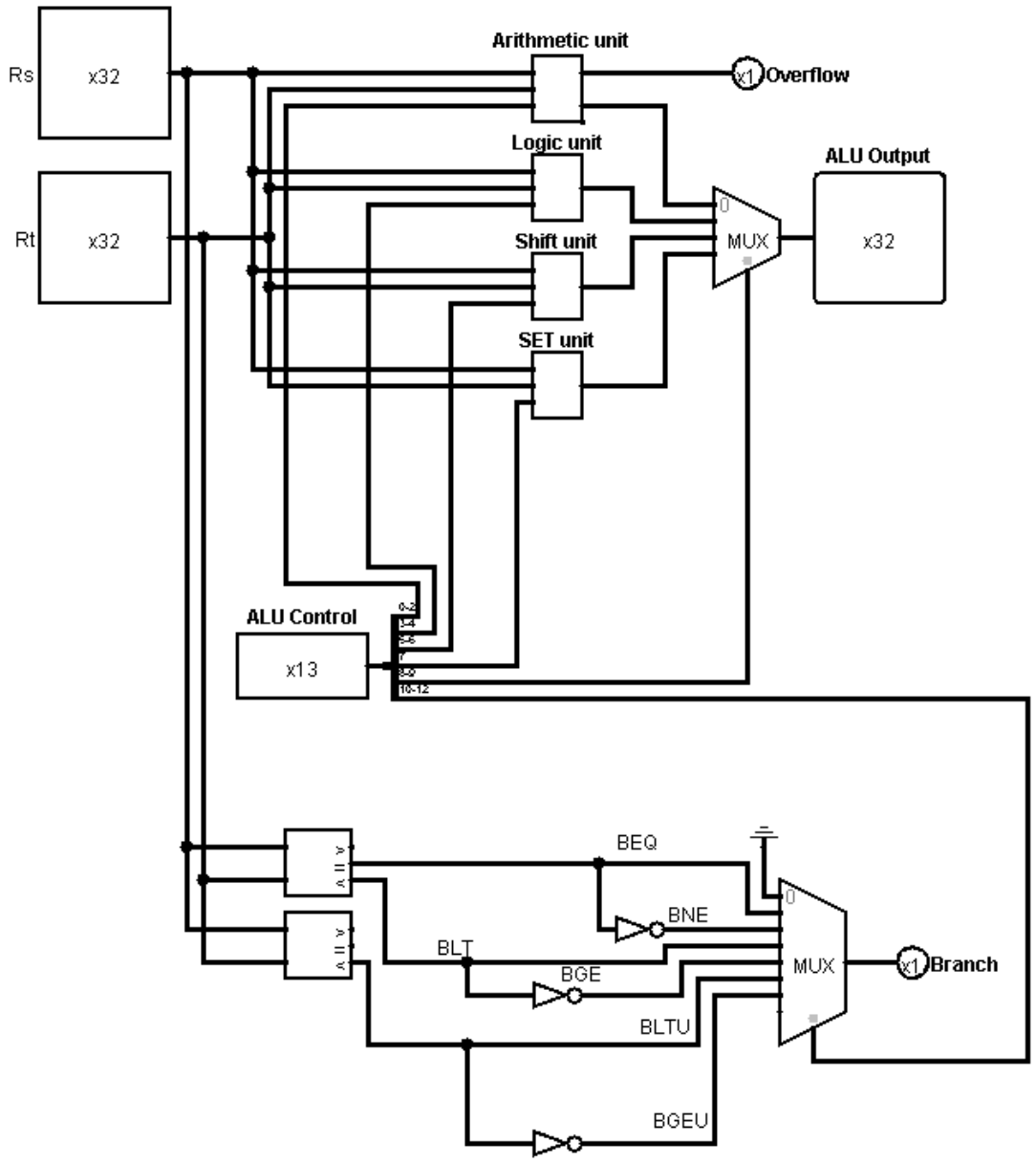


Figure 25: ALU design in Logisim showing inputs (Rs, Rt, ALU Control), outputs (ALU Output, Overflow Signal, Branch Signal) and internal circuit.

All internal ALU units – Arithmetic, Logic, Shift, and SET – operate **in parallel**. Each unit continuously receives the two 32-bit inputs and independently computes its corresponding result.

However, only one output is selected based on the 2-bit **ALUMuxControl** signal (8-9 bits in ALU Control signal), which is generated by the **ALU Control Circuit**.

A multiplexer inside the ALU uses ALUMuxControl value to choose the appropriate result among the different unit outputs, ensuring the correct operation is performed without unnecessary delay.

3.7.1 ALU Features

The ALU supports the following instruction type:

- **Arithmetic**: ADD, SUB, MUL, SLT, SLTU
- **Logical**: AND, OR, XOR, NOR
- **Shift and Rotate**: SLL, SRL, SRA, ROR
- **Set operations**: SEQ, SLT ,SLTU
- **Assignment processes**: SET, SSET
- **Memory processes**: LW, SW
- **Branch condition support**: Equal, Less Than (signed & unsigned)

The same ALU structure supports both R-type and I-type instructions, minimizing hardware duplication.

The ALU is designed in a modular way using Logisim subcircuits, each handling a specific group of operations:

- **Arithmetic Unit**: performs ADD, SUB, MUL, SEQ, SLT, SLTU
- **Logic Unit**: Handles AND, OR, NOR, XOR
- **Shift Unit**: makes SLL, SRL, SRA, ROR
- **SET Unit**: sets the register using SET, SSET
- **Branch Condition Logic**: Determines branch decisions based on instruction type and internal flag Branch.

Each unit receives the same 32-bit inputs, and a multiplexer inside the ALU selects the final output based on the **ALU Control** code.

3.7.2 Arithmetic Unit

The Arithmetic Unit receives two 32-bit inputs and performs arithmetic operations including addition, subtraction, multiplication, and comparison. It also provides **overflow flag** used for adder and subtractor overflow. The unit output is selected according to **Arithmetic Control** comes from (ALU Control Circuit -> ALU Circuit).

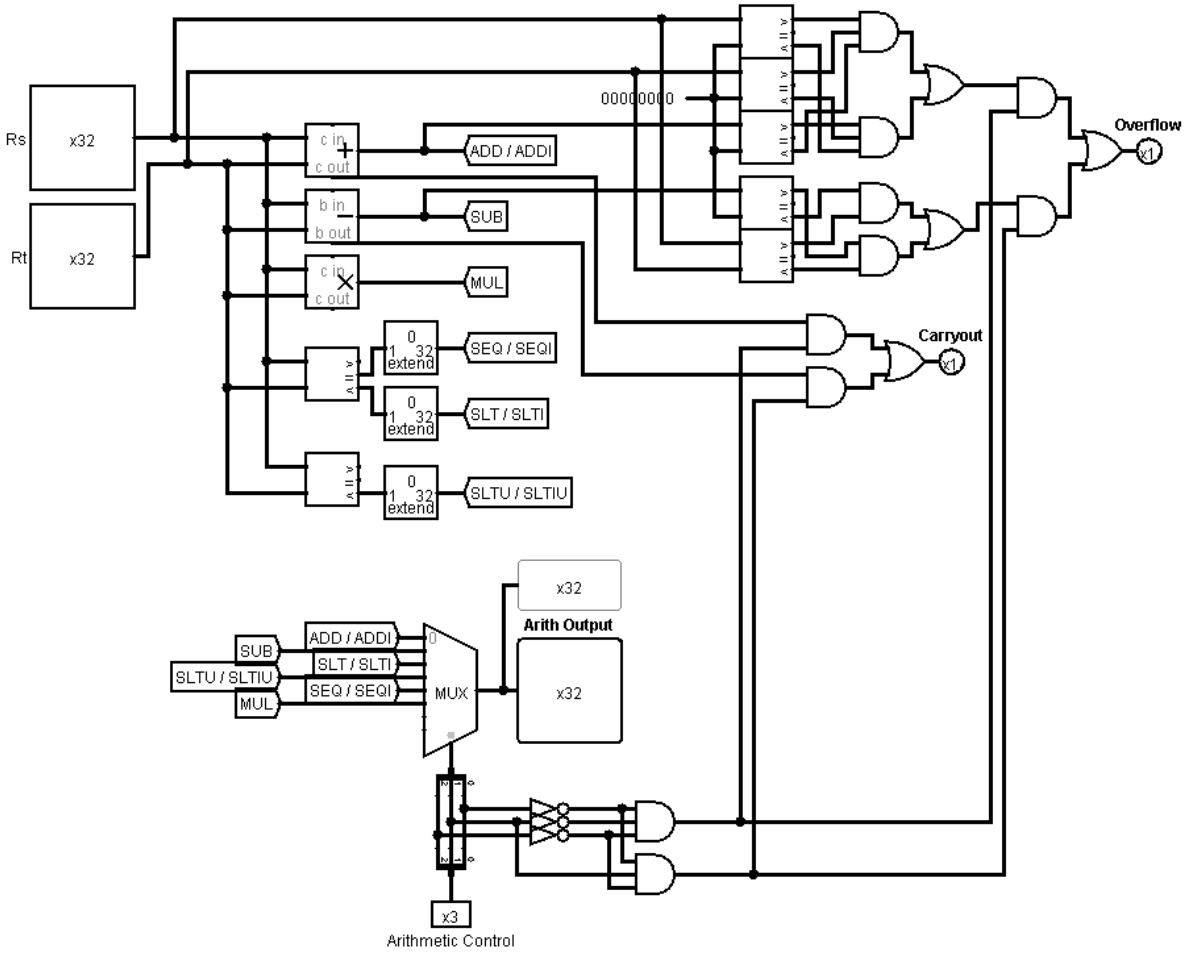


Figure 26: Arithmetic Unit performs (ADD, SUB, MUL, SEQ, SLT, SLTU).

This specific control signal, called **Arithmetic Control**, is generated inside the ALU Control Circuit based on the instruction type and function code.

The Arithmetic Control signal then drives the internal multiplexer of the Arithmetic Unit to choose the correct arithmetic operation output.

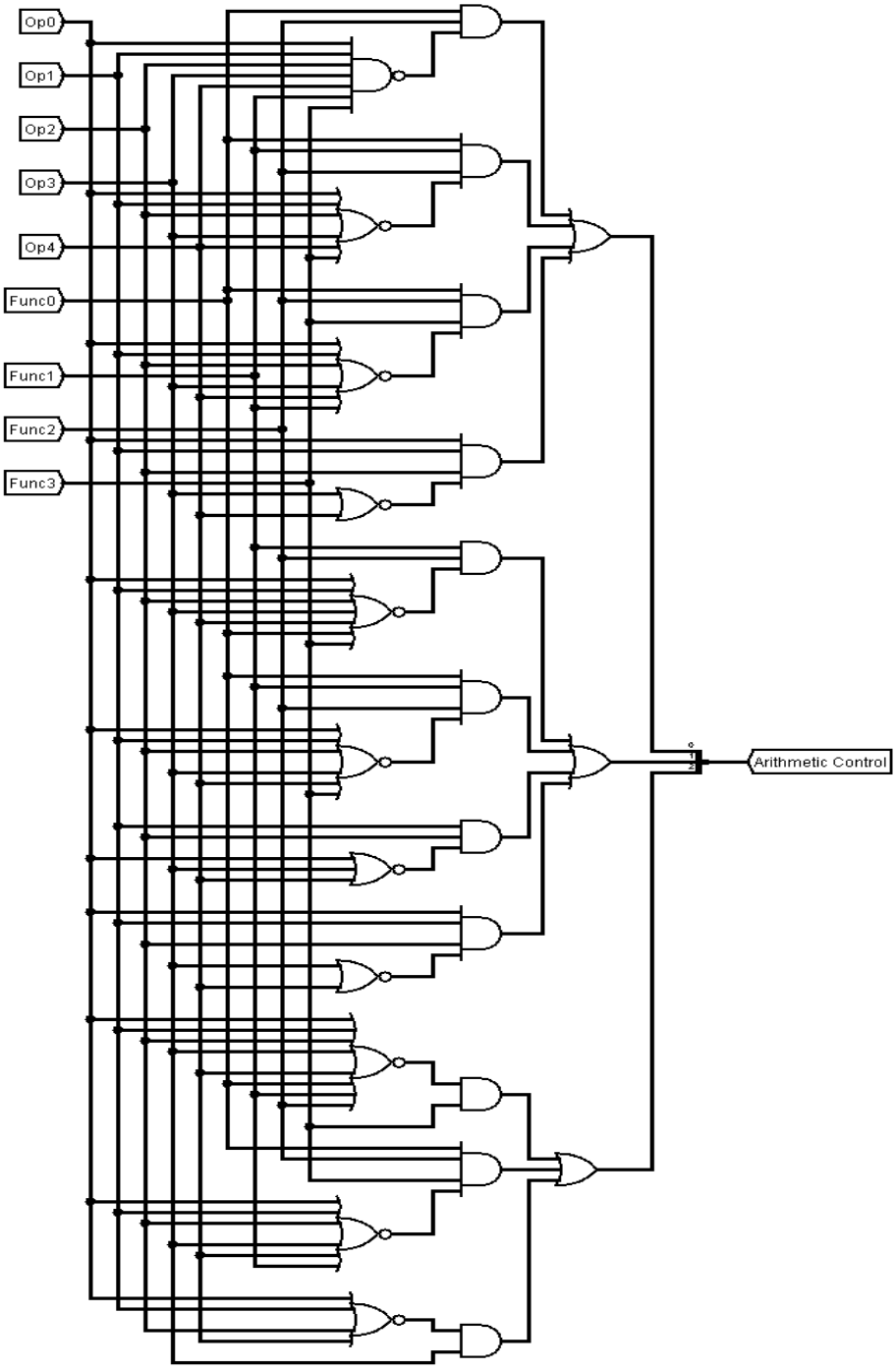


Figure 27: Arithmetic Control signal circuit

This Arithmetic Control Circuit was designed based on the following table, which maps ALU operations to their corresponding control signals.

The control signal determines the specific arithmetic operation (such as ADD, SUB, MUL, SLT, or SEQ) to be performed by the Arithmetic Unit.

Instruction	Function (0-3 bits)	Opcode (0-4 bits)	Operation	Arithmetic Control
ADD	0100	00000	Adder	000
SUB	0101	00000	Subtractor	001
SLT	0110	00000	SLT	010
SLTU	0111	00000	SLTU	011
SEQ	1000	00000	SEQ	100
MUL	1101	00000	Multiplier	101
ADDI	-	00101	Adder	000
SLTI	-	00110	SLT	010
SLTIU	-	00111	SLTU	011
SEQI	-	01000	SEQ	100
JALR	-	01111	Adder	000
LW	-	10000	Adder	000
SW	-	10001	Adder	000

Table 3: Arithmetic Instructions and its control signal

3.7.3 Logic Unit

This subcircuit performs logic operations by combining the two 32-bit input operands using the specified bitwise gate. It is activated when the ALU receives logic-based control codes (e.g., for AND, OR, XOR, etc) It ensures fast and simple evaluation of logical conditions for both R-type and I-type instructions.

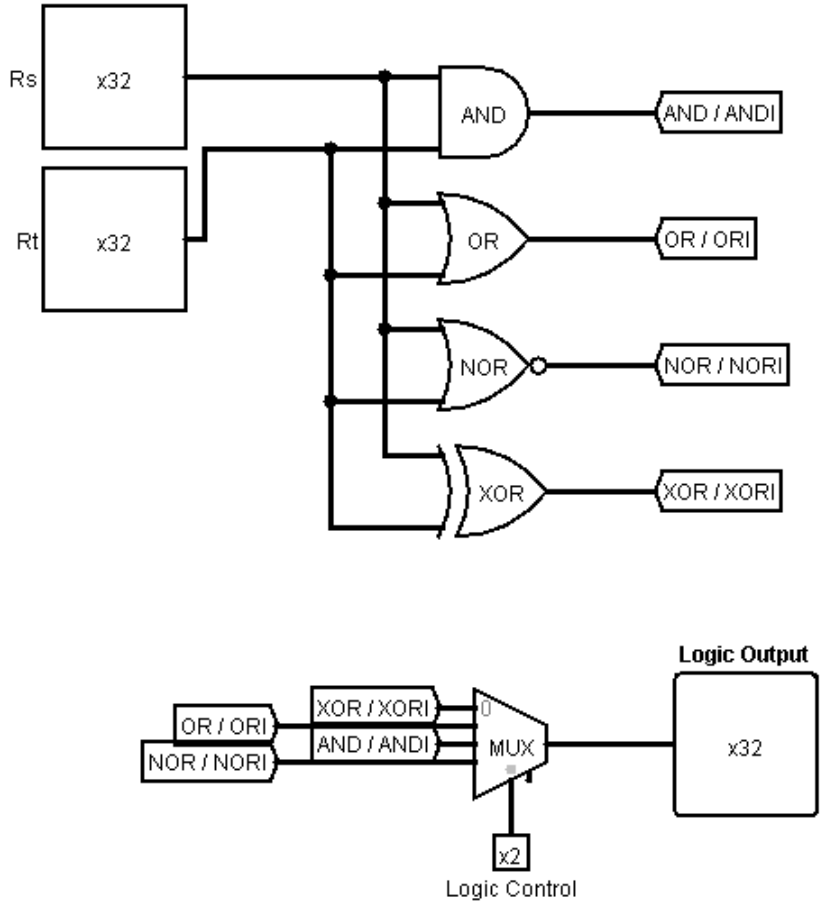


Figure 28: Logic Unit handles (AND, OR, NOR, XOR)

The specific logical operation to be executed is selected by a dedicated **Logic Control signal**, generated from the ALU Control Circuit.

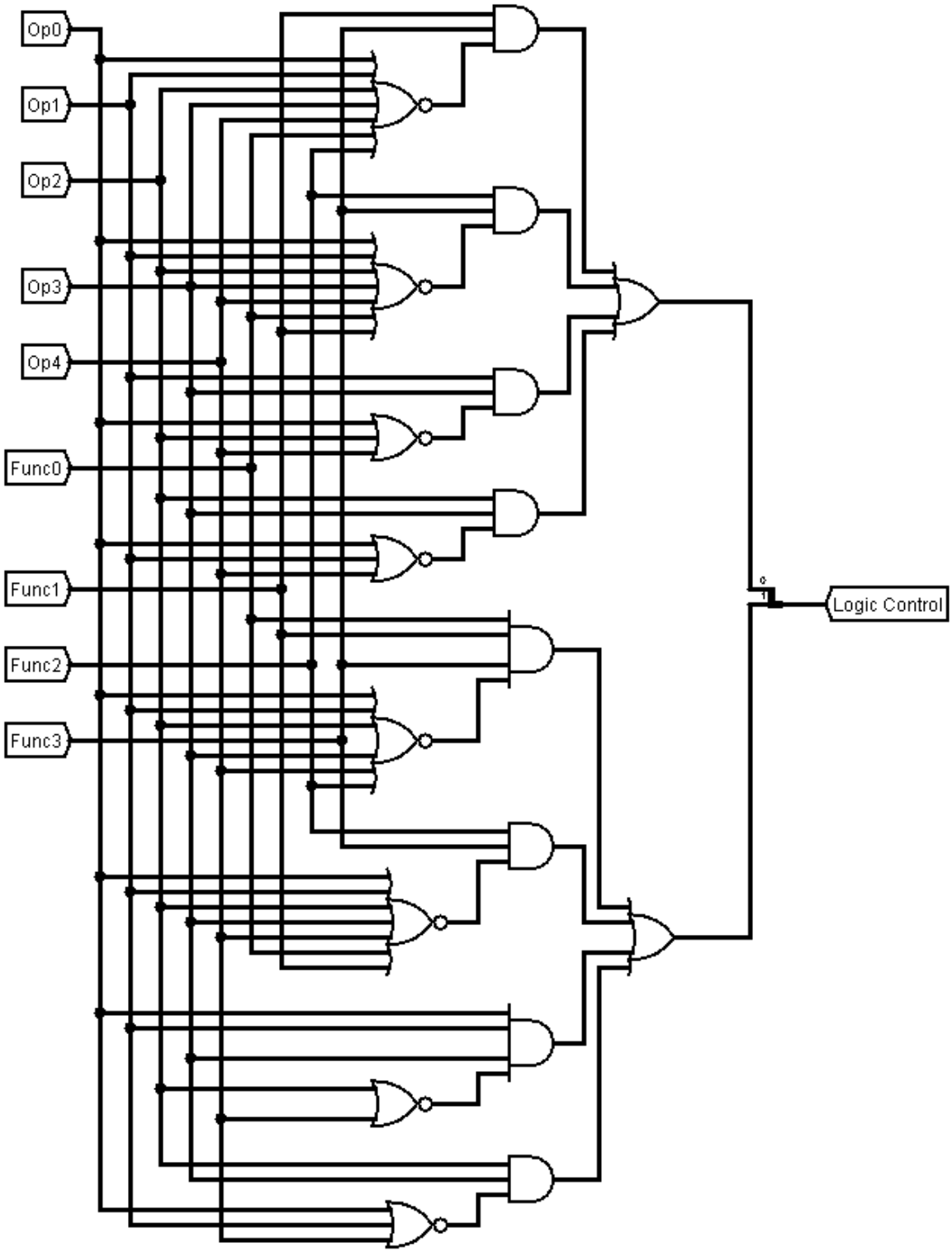


Figure 29: Logic Control signal circuit

This Logic Control Circuit was designed based on the following table, which maps ALU operations to their corresponding logical control signals.

The Logic Control signal selects between the available logic operations inside the Logic Unit (AND, OR, XOR, NOR).

Instruction	Function (0-3 bits)	Opcode (0-4 bits)	Operation	Logic Control
XOR	1001	00000	XOR	00
OR	1010	00000	OR	01
AND	1011	00000	AND	10
NOR	1100	00000	NOR	11
XORI	-	01001	XOR	00
ORI	-	01010	OR	01
ANDI	-	01011	AND	10
NORI	-	01100	NOR	11

Table 4: Logic Instructions and its control signal

3.7.4 Shift Unit

This unit shifts the bits of the input either logically or arithmetically, left or right, or rotates them based on the instruction. The shift amount is determined either from the register input (R-type) or immediate field (I-type). It is used in various bit-manipulation operations and is selected by corresponding **ALUControl** codes.

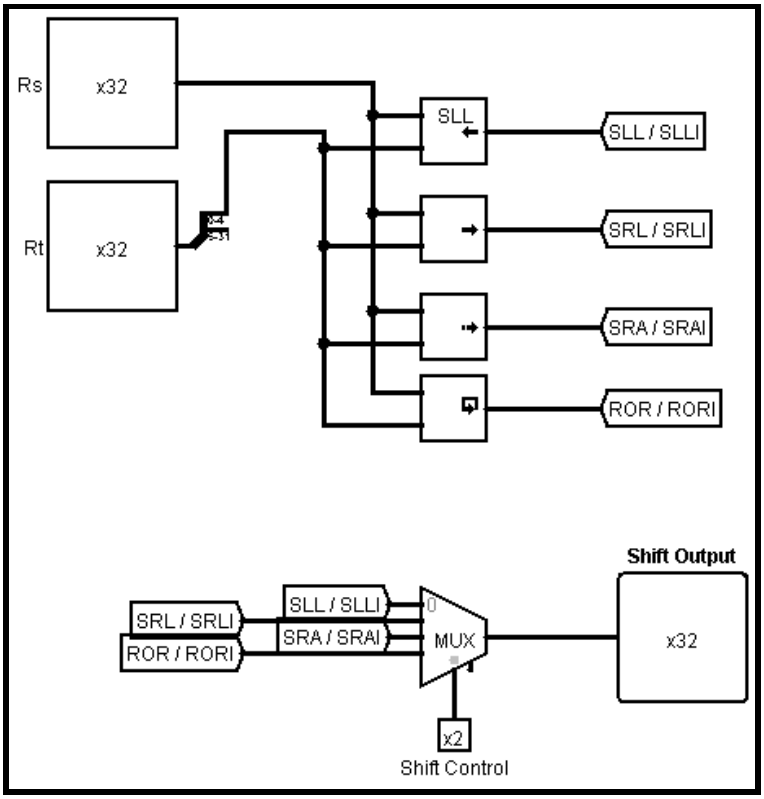


Figure 30: Shift Unit makes (SLL, SRL, SRA, ROR)

Similar to other ALU units, the Shift Unit processes the two 32-bit input operands in parallel and produces a shifted result based on a control signal.

The specific shift operation to perform is selected using a dedicated **Shift Control signal**, which is generated from the ALU Control Circuit.

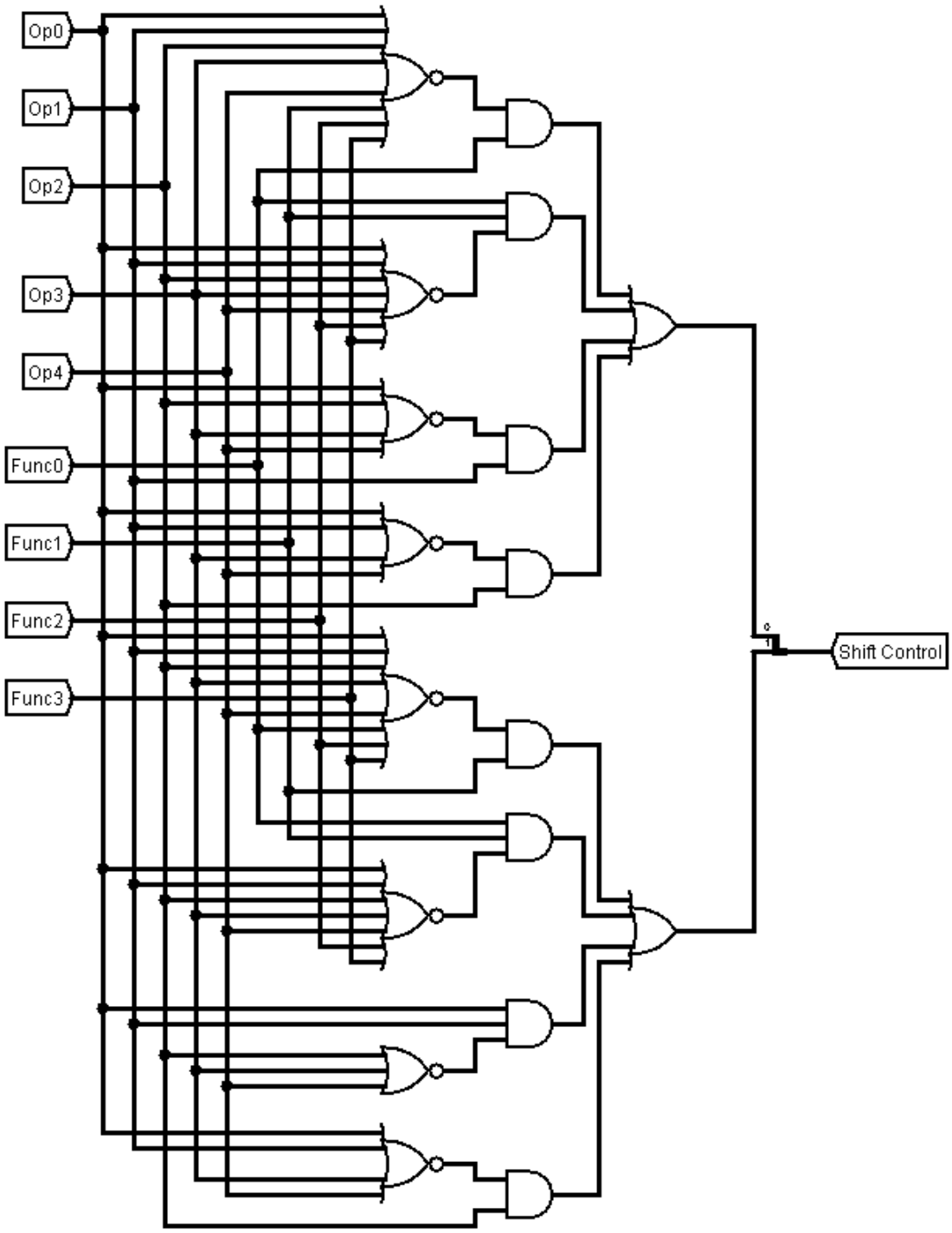


Figure 31: Shift Control signal circuit

The Shift Control Circuit was designed based on the following table, which associates each shift instruction with its corresponding control code.

The Shift Control signal selects the appropriate shift or rotate operation to be executed by the Shift Unit.

Instruction	Function (0-3 bits)	Opcode (0-4 bits)	Operation	Shift Control
SLL	0000	00000	SLL	00
SRL	0001	00000	SRL	01
SRA	0010	00000	SRA	10
ROR	0011	00000	ROR	11
SLLI	-	00001	SLL	00
SRLI	-	00010	SRL	01
SRAI	-	00011	SRA	10
RORI	-	00100	ROR	11

Table 5: Shift Instructions and its control signal

3.7.5 SET Unit

The Set Unit handles special instructions that either directly load immediate values into a register (**SET**) or concatenate register and immediate parts (**SSET**). This unit simplifies data initialization and control setups in programs. Its outputs are routed through the main ALU multiplexer.

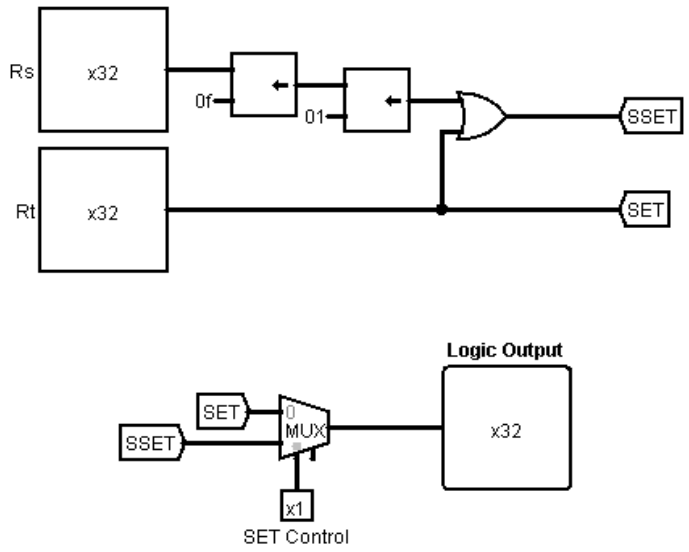


Figure 32: SET Unit sets register with (SET, SSET)

A simple internal logic differentiates between **SET** and **SSET** operations based on the **ALUControl** signal provided by the ALU Control Circuit.

The following circuit shows the generation of the selection signal used inside the SET Unit.

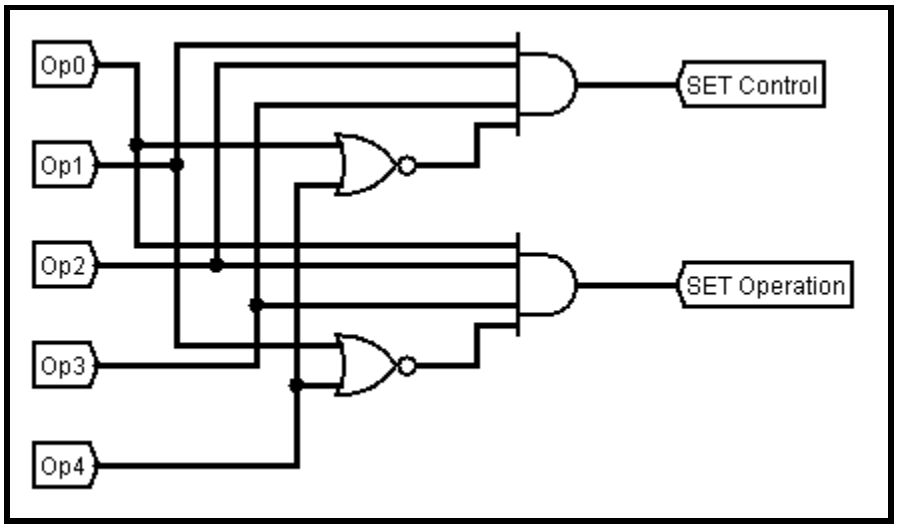


Figure 33: SET Control signal circuit

The corresponding mapping table is provided below:

Instruction	Opcode (0-4 bits)	Operation	SET Control
SET	01101	SET	0
SSET	01110	SSET	1

Table 6: SET and SSET Instructions and its control signal

3.7.6 Branch Unit

The ALU includes internal logic for supporting **branch instructions**, specifically **BEQ**, **BNE**, **BLT**, **BGE**, **BLTU**, and **BGEU**.

Although the branch logic is not separated into an independent subcircuit, it is an essential part of the ALU's functionality.

The Branch Logic uses flags for every instruction and collected in Mux to evaluate branch conditions depending on the instruction type.

Based on these evaluations, the ALU produces a 1-bit **Branch** signal that is used by the processor's control logic to decide whether the branch is taken.

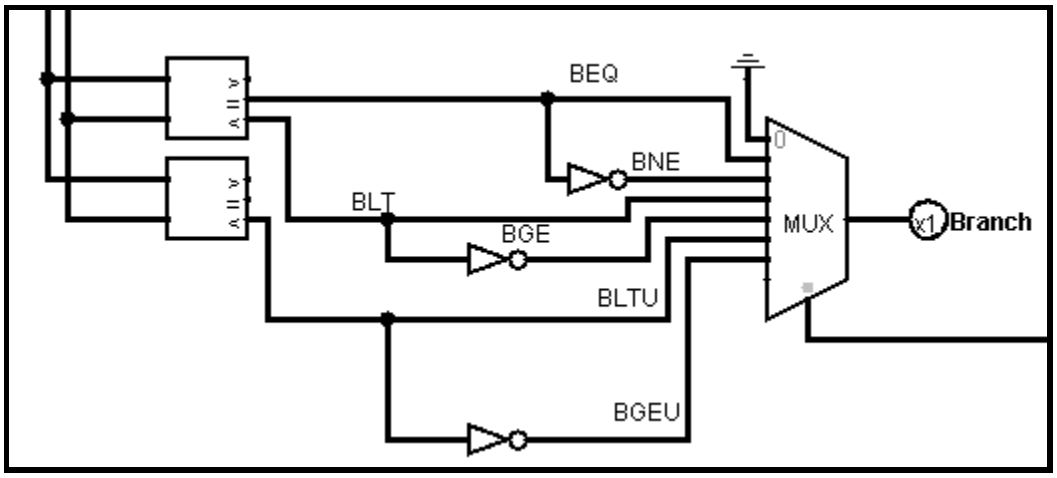


Figure 34: Branch signal part of ALU

The Branch Control Logic inside the ALU selects the correct comparison flag based on the branch instruction type.

This selection is made using a dedicated control signal generated from the ALUControl input.

The following circuit shows how the ALU selects between instructions to produce the correct Branch output.

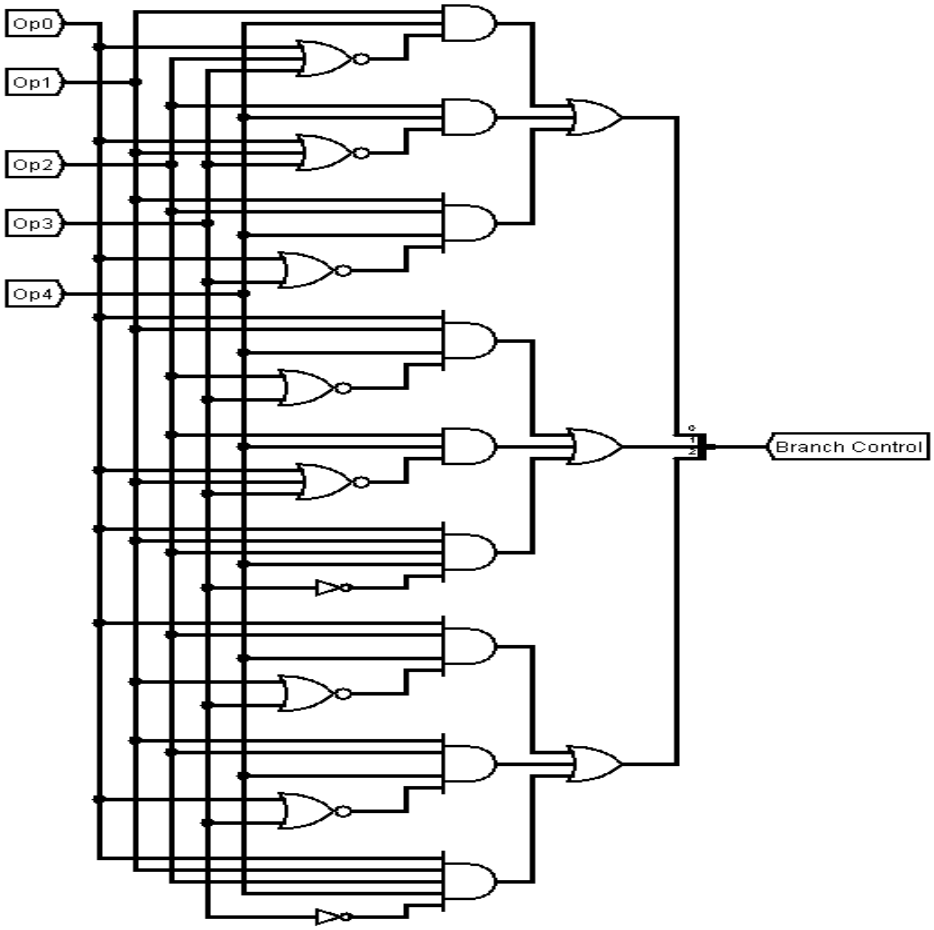


Figure 35: Branch Control signal circuit

The corresponding mapping table is provided below:

Instruction	Opcode (0-4 bits)	Operation	Branch Control
BEQ	10010	BEQ	001
BNE	10011	BNE	010
BLT	10100	BLT	011
BGE	10101	BGE	100
BLTU	10110	BLTU	101
BGEU	10111	BGEU	110

Table 7: Branch Instructions and its control signal

3.8 Data Memory (RAM)

3.8.1 Overview

The Data Memory is a word-addressable RAM used for data storage and retrieval during program execution, accessed by LW and SW instructions. It supports both read and write operations.

3.8.2 Ports and Signals

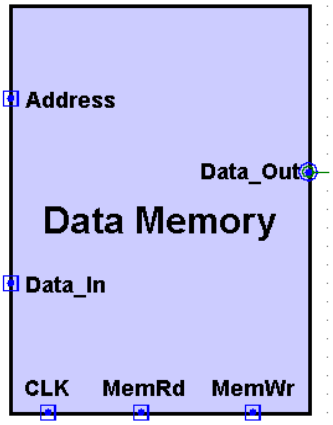


Figure 36: Data Memory Block

- Address (Input): Memory location for read/write.
- Data_In (Input): Data to be written.
- Data_Out (Output): Data read from memory.
- CLK (Input): Clock signal for write synchronization.
- MemRd (Input): Read enable (1 = read).
- MemWr (Input): Write enable (1 = write).

3.8.3 Functionality

- Read: If MemRd is asserted, data at Address is output on Data_Out.
- Write: If MemWr is asserted on the rising edge of CLK, Data_In is stored at Address.
- Read and write are mutually exclusive.

3.8.4 Design Considerations

- Word Addressing: Each address refers to a 32-bit word.
- Initialization: Memory contents are preloaded or initialized via SET and SW.
- Latency: Assumes one-cycle read/write.
- Alignment: Data is word-aligned (32 bits). Misaligned accesses are unsupported.

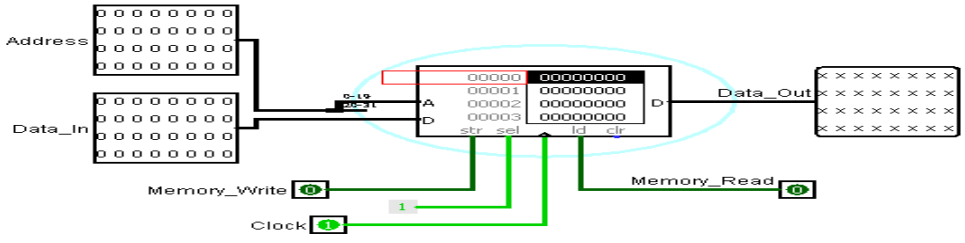


Figure 37: Data Memory Internal Design

4.Single-Cycle Implementation

The Single-Cycle Processor integrates all major components into a unified datapath, enabling the execution of each instruction within one clock cycle. This architecture prioritizes simplicity and clarity, making it ideal for understanding the fundamental operations of a processor. Each instruction progresses through the stages of fetch, decode, execute, memory access, and write-back sequentially, completing all stages before the next instruction begins.

4.1 Datapath Overview

The single-cycle datapath comprises the following main components:

- **Program Counter (PC):** Fetches the address of the next instruction.
- **Instruction Memory:** Provides the instruction corresponding to the PC value.
- **Instruction Splitter:** Divides the instruction into opcode, register fields, and immediate values.
- **Register File:** Supplies source operands and stores destination results.
- **Bit Extender:** Extends immediate values to 32 bits when required.
- **Arithmetic Logic Unit (ALU):** Executes arithmetic and logical operations.
- **Data Memory:** Supports load and store instructions for memory operations.
- **Control Unit:** Generates the control signals based on the opcode and function fields.

Each instruction completes its execution through all stages within a single clock cycle without overlapping or pipelining.

4.2 Component Interconnections

The datapath connects the components as follows:

- The **PC** supplies the instruction address to **Instruction Memory**.
- The fetched instruction is processed by the **Instruction Splitter** into fields like opcode, source registers, destination register, and immediate values.
- **Register File** reads the operands based on the instruction fields.
- **Bit Extender** processes and extends immediate values when necessary.
- **ALU** executes computations with inputs from the **Register File** or extended immediates.

- **Data Memory** is accessed if the instruction involves memory operations (load/store).
- The result is selected either from the **ALU** output or **Data Memory**, and written back to the **Register File**.
- **Control signals** coordinate the operations across all components.

4.3 Instruction Flow

Each instruction progresses through the following stages within one clock cycle:

- **Fetch:**
  - PC sends address to Instruction Memory.
  - Instruction is retrieved and passed into the datapath.
- **Decode:**
  - Instruction fields are split.
  - Register operands are read from the Register File.
- **Execute:**
  - ALU performs the required operation.
  - Branch conditions are evaluated if applicable.
- **Memory Access:**
  - Load instructions read from Data Memory.
  - Store instructions write to Data Memory.
- **Write-Back:**
  - ALU results or loaded data are written back to the Register File.

4.4 Key Control Signals

The Control Unit generates the following key control signals:

Control Signal	Purpose
RegDst	Selects the destination register
ALUSrc	Chooses between register or immediate value for ALU
MemtoReg	Selects ALU result or memory output for write-back
RegWrite	Enables writing to the Register File
MemRead	Enables reading from Data Memory
MemWrite	Enables writing to Data Memory
Branch	Activates branching operations

Control Signal	Purpose
ALUOp	Specifies the ALU operation
Extender_Op	Chooses sign or zero extension for immediates
ImmSelect	Selects between Imm16 and ImmU formats

Table 8: Control Signals

These control signals are derived from the opcode and function fields of the instruction.

4.5 Single-Cycle Datapath Diagram

The following figure illustrates the complete single-cycle processor datapath, integrating all major components described:

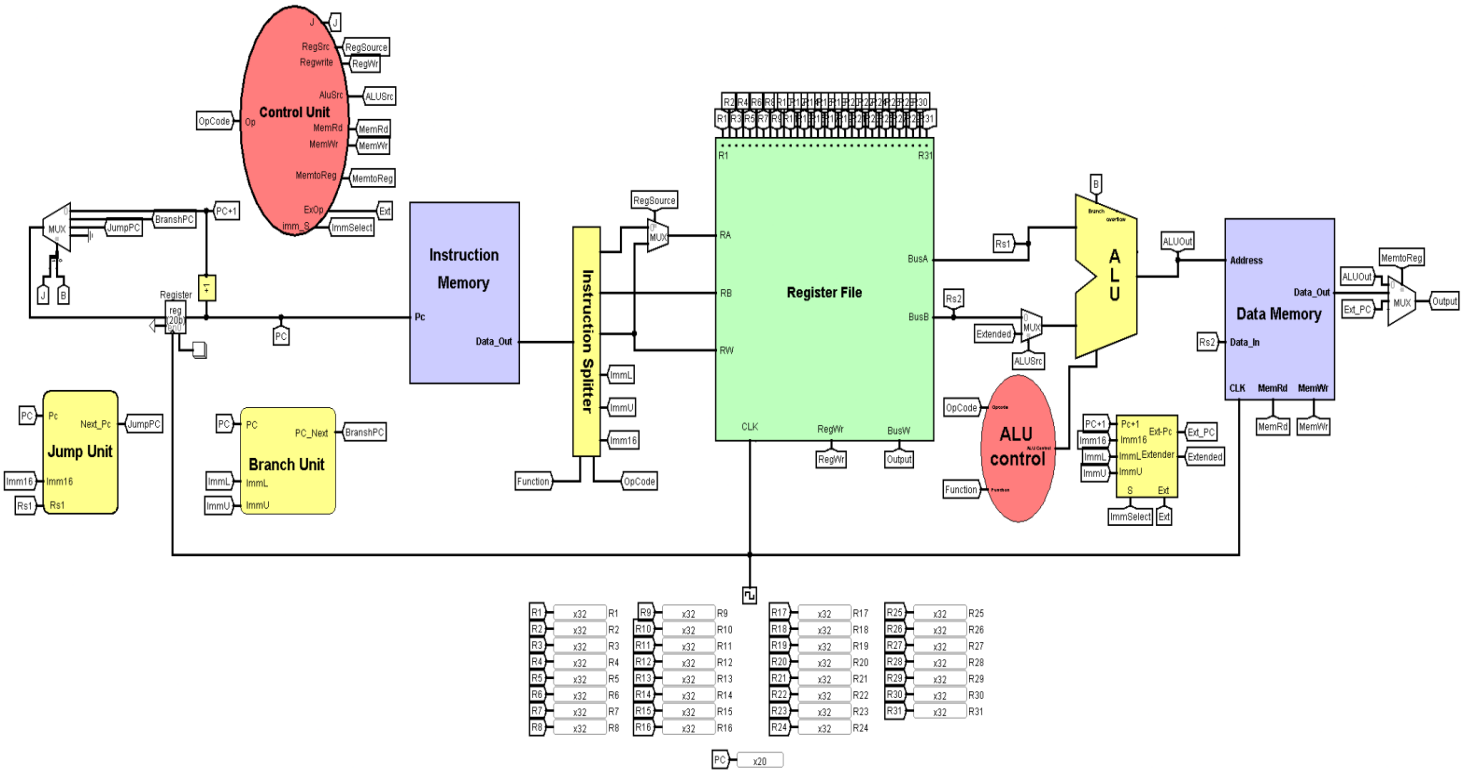


Figure 38: : Single-Cycle Processor Datapath

5. Simulation and Testing

5.1 Unit Testing

Each component (ALU, Register File, Control Unit) was tested independently to ensure correctness.

5.2 Instruction Testing

A test program was written including all instruction types. Instructions were verified by observing changes in register and memory values.

5.3 Test Program Results

- **The following table shows the results of executing the provided test code:**

Instruction	address	Expected value	Actual value
SET R1, 0x0384	0x0000B	R1 = 0x00000384	R1 = 0x00000384
SET R8, 0x1234	0x0000C	R8 = 0x00001234	R8 = 0x00001234
SSET R8, 0x5678	0x0000D	R8 = 0x12345678	R8 = 0x12345678
ADDI R5, R1, 20	0x0000E	R5 = 0x00000398	R5 = 0x00000398
XOR R3, R1, R5	0x0000F	R3 = 0x0000001C	R3 = 0x0000001C
ADD R4, R8, R3	0x00010	R4 = 0x12345694	R4 = 0x12345694
LW R1, 0(R0)	0x00011	R1 = Mem[0] = 0x00000001	R1 = Mem[0] = 0x00000001
LW R2, 1(R0)	0x00012	R2 = Mem[1] = 0x00000001	R2 = Mem[1] = 0x00000001
LW R3, 2(R0)	0x00013	R3 = Mem[2] = 0x0000000A	R3 = Mem[2] = 0x0000000A
SUB R4, R4, R4	0x00014	R4 = 0x00000000	R4 = 0x00000000
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x00000001	R4 = 0x00000001
SLT R6, R2, R3	0x00016	R6 = 0x00000001	R6 = 0x00000001
BEQ R6, R0, done	0x00017	Branch = 0	Branch = 0
ADD R2, R1, R2	0x00018	R2 = 0x00000002	R2 = 0x00000002
BEQ R0, R0, Loop1	0x00019	Branch = 1	Branch = 1

Loop1: ADD R4, R2, R4	0x00015	R4 = 0x00000003	R4 = 0x00000003
SLT R6, R2, R3	0x00016	R6 = 0x00000001	R6 = 0x00000001
BEQ R6, R0, done	0x00017	Branch = 0	Branch = 0
ADD R2, R1, R2	0x00018	R2 = 0x00000003	R2 = 0x00000003
BEQ R0, R0, Loop1	0x00019	Branch = 1	Branch = 1
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x00000006	R4 = 0x00000006
SLT R6, R2, R3	0x00016	R6 = 0x00000001	R6 = 0x00000001
BEQ R6, R0, done	0x00017	Branch = 0	Branch = 0
ADD R2, R1, R2	0x00018	R2 = 0x00000004	R2 = 0x00000004
BEQ R0, R0, Loop1	0x00019	Branch = 1	Branch = 1
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x0000000A	R4 = 0x0000000A
SLT R6, R2, R3	0x00016	R6 = 0x00000001	R6 = 0x00000001
BEQ R6, R0, done	0x00017	Branch = 0	Branch = 0
ADD R2, R1, R2	0x00018	R2 = 0x00000005	R2 = 0x00000005
BEQ R0, R0, Loop1	0x00019	Branch = 1	Branch = 1
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x0000000f	R4 = 0x0000000f
SLT R6, R2, R3	0x00016	R6 = 0x00000001	R6 = 0x00000001
BEQ R6, R0, done	0x00017	Branch = 0	Branch = 0
ADD R2, R1, R2	0x00018	R2 = 0x00000006	R2 = 0x00000006
BEQ R0, R0, Loop1	0x00019	Branch = 1	Branch = 1
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x00000015	R4 = 0x00000015
SLT R6, R2, R3	0x00016	R6 = 0x00000001	R6 = 0x00000001
BEQ R6, R0, done	0x00017	Branch = 0	Branch = 0
ADD R2, R1, R2	0x00018	R2 = 0x00000007	R2 = 0x00000007

BEQ R0, R0, Loop1	0x00019	Branch = 1	Branch = 1
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x0000001C	R4 = 0x0000001C
SLT R6, R2, R3	0x00016	R6 = 0x00000001	R6 = 0x00000001
BEQ R6, R0, done	0x00017	Branch = 0	Branch = 0
ADD R2, R1, R2	0x00018	R2 = 0x00000008	R2 = 0x00000008
BEQ R0, R0, Loop1	0x00019	Branch = 1	Branch = 1
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x00000024	R4 = 0x00000024
SLT R6, R2, R3	0x00016	R6 = 0x00000001	R6 = 0x00000001
BEQ R6, R0, done	0x00017	Branch = 0	Branch = 0
ADD R2, R1, R2	0x00018	R2 = 0x00000009	R2 = 0x00000009
BEQ R0, R0, Loop1	0x00019	Branch = 1	Branch = 1
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x0000002D	R4 = 0x0000002D
SLT R6, R2, R3	0x00016	R6 = 0x00000001	R6 = 0x00000001
BEQ R6, R0, done	0x00017	Branch = 0	Branch = 0
ADD R2, R1, R2	0x00018	R2 = 0x0000000A	R2 = 0x0000000A
BEQ R0, R0, Loop1	0x00019	Branch = 1	Branch = 1
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x00000037	R4 = 0x00000037
SLT R6, R2, R3	0x00016	R6 = 0x00000000	R6 = 0x00000000
BEQ R6, R0, done	0x00017	Branch = 1	Branch = 1
done: SW R4, 0(R0)	0x0001A	Mem[0] = 0x00000037, R0 = 0x00000000	Mem[0] = 0x00000037, R0 = 0x00000000
MUL R10, R2, R3	0x0001B	R10 = 0x00000064	R10 = 0x00000064
SRL R14, R10, R4	0x0001C	R14 = 0x00000000	R14 = 0x00000000
SRA R15, R10, R4	0x0001D	R15 = 0x00000000	R15 = 0x00000000
RORI R26, R14, 5	0x0001E	R26 = 0x00000000	R26 = 0x00000000

JALR R7, R0, func	0x0001F	PC = func, R7 = 0x00000020	PC = func, R7 = 0x00000020
func: OR R5, R2, R3	0x00025	R5 = 0x0000000A	R5 = 0x0000000A
LW R1, 0(R0)	0x00026	R1 = 0x00000037	R1 = 0x00000037
LW R2, 5(R1)	0x00027	R2 = 0x128945AC	R2 = 0x128945AC
LW R3, 6(R1)	0x00028	R3 = 0x05007342	R3 = 0x05007342
AND R4, R2, R3	0x00029	R4 = 0x00004100	R4 = 0x00004100
SW R4, 0(R0)	0x0002A	Mem[0] = 0x00004100	Mem[0] = 0x00004100
JALR R0, R7, 0	0x0002B	PC = 0x00020, R0 = 0x00000000	PC = 0x00020, R0 = 0x00000000
SET R9, 0x4545	0x00020	R9 = 0x00004545	R9 = 0x00004545
SET R10, 0x4545	0x00021	R10 = 0x00004545	R10 = 0x00004545
BGE R10, R9, L1	0x00022	Branch = 1	Branch = 1
ANDI R23, R1, 0xFFFF	Skipped		
L1: BEQ R0, R0, L1	0x00024	infinite loop: End	infinite loop: End

Table 9: Test Code Results

All instructions were executed successfully without errors.

- **Registers and Data Memory**: All register updates and memory accesses matched the expected outputs at every instruction step.
- **Program Counter (PC)**: The PC advanced properly for each sequential instruction and correctly handled branches and jumps.
- **Control Signals**: Control signals generated by the Control Unit matched the expected behavior for every instruction type, including arithmetic, memory, and branch operations.
- **Loops and Branches**: All loop iterations and conditional branches were evaluated correctly, demonstrating proper branching logic.
- **Special Operations**: Instructions such as shifts, rotates, and JALR (jump and link register) were verified to behave as designed.

Thus, the processor passed all functional tests according to the given test program.

5.4 Array Test Code: Temperature Monitoring System

5.4.1 Program Objective

This test program simulates a **Temperature Monitoring System** using our custom RISC instruction set. The goal is to:

- Initialize an array with temperature values.
- Calculate the **average temperature**.
- Compare the average to a threshold.
- Store both the average and an **alert flag** in memory.

This program involves procedures that interact with arrays and control logic, as required by the project specification.

5.4.2 Program Structure

A. Initialization

```
SET   R3, 8            # Number of readings
SET   R4, 20           # Starting temperature
SET   R5, 5            # Step increment
SET   R6, 4            # Word size (bytes)
SET   R1, 0x2000       # Base address for array
SET   R2, 0            # Loop counter
```

Initial values are configured for storing 8 readings starting from 20°C, with a 5°C increment, into memory.

B. Filling the Temperature Array

```
Loop_fill:
SW    R4, 0(R1)
ADD   R1, R1, R6
ADD   R4, R4, R5
ADDI  R2, R2, 1
SLTI  R7, R2, 8
BNE   R7, R0, Loop_fill
```

This loop fills memory addresses from `0x2000` to `0x201C` with the values:
20, 25, 30, 35, 40, 45, 50, 55

C. Summing the Temperature Values

```
SET   R1, 0x2000        # Reset base address
SET   R2, 0             # Reset loop counter
SET   R7, 0             # Sum accumulator

loop_sum:
LW    R4, 0(R1)
ADD   R7, R7, R4
ADD   R1, R1, R6
ADDI  R2, R2, 1
SLTI  R8, R2, 8
BNE   R8, R0, loop_sum
```

This section calculates the sum of the temperature values:
Sum = 20 + 25 + ... + 55 = 300

D. Average Calculation and Storage

```
SRLI  R8, R7, 3         # Divide by 8 using shift
SET   R9, 0x3000
SW    R8, 0(R9)
```

The sum is divided by 8 using logical right shift (`>> 3`), resulting in the average value `37`, which is stored at memory address `0x3000`.

E. Threshold Comparison and Alert Flag

```
SET   R10, 50
SLT   R11, R10, R8
SET   R12, 0x3004
SW    R11, 0(R12)
```

This block compares the average temperature to a threshold value of 50°C.
If the average exceeds the threshold, an **alert flag** is raised (`R11 = 1`), otherwise `0`.
In this case:
37 < 50 ⇒ Alert = 0 → Stored at `0x3004`.

5.4.3 Memory Output Summary

Address	Description	Value Stored
0x2000–0x201C	Temperature readings	20, 25, ..., 55
0x3000	Average temperature	37
0x3004	Alert flag	0

5.4.4 Machine Code Representation

The program was assembled into binary using a custom Python-based assembler. Below is the machine code loaded into instruction memory:

```
000800CD 0014010D 0005014D 0004018D 2000004D 0000008D
00040811
00860840 00852100 00011085 000811C6 FFE03ED3 2000004D
0000008D
000001CD 00000910 008439C0 00860840 00011085 00081206
FFE046D3
00033A02 3000024D 00084811 0032028D 00C852C0 3004030D
000B6011
```

Each line represents a 32-bit instruction in hexadecimal, following the project's instruction format.

5.4.5 Outcome and Analysis

- The program successfully initializes an array and calculates the average temperature.
- Memory values match expected results.
- The average (37°C) is below the threshold (50°C), and the alert flag is correctly set to 0.
- The program demonstrates key features of our pipelined processor, including:
    - Memory operations (LW, SW)
    - Arithmetic and logic (ADD, SLT, SRLI)
    - Control flow (BNE)
    - Structured procedure-like behavior

This test program satisfies the project's requirement for a functional example involving array manipulation, parameter passing, arithmetic computation, and conditional logic.

Phase 2: Pipelined Processor

6. Pipelined Processor Design and Implementation

6.1 Introduction

In the first part of our project, we built a Single-Cycle RISC processor to learn about how processors work. This design could run one instruction at a time, and while it worked well, it was slow because it had to finish one instruction before starting the next.

To fix this and make the processor faster, we improved our design in Phase 2 by using **pipelining**. Pipelining lets the processor work on several instructions at once by breaking each instruction into smaller steps and handling those steps in different parts of the processor.

But pipelining comes with new problems—like when one instruction needs data from another, or when a branch instruction changes the flow of the program. To handle these problems, we added special circuits to:

- Detect and handle instruction conflicts (Hazard Detection Unit)
- Send data where it's needed without waiting (Forwarding Logic)
- Pause the pipeline when necessary (Stalling Mechanism)

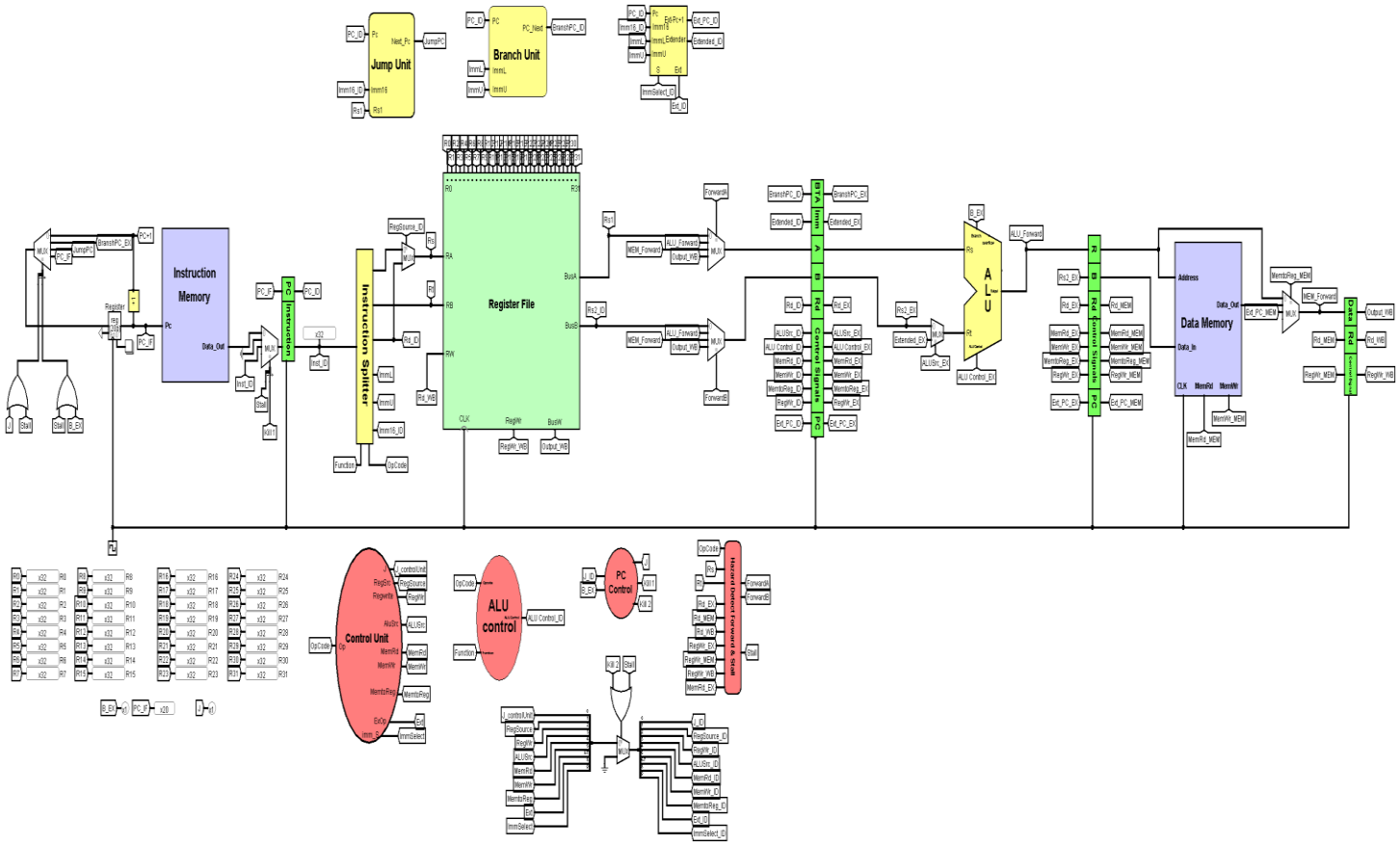
These changes help the processor run faster while still giving the correct results.

6.2 Pipeline Architecture

The pipelined processor consists of **five sequential stages**, with **pipeline registers** separating each stage:

1. **Instruction Fetch (IF)**
2. **Instruction Decode (ID)**
3. **Execute (EX)**
4. **Memory Access (MEM)**
5. **Write Back (WB)**

Each stage processes part of the instruction, and every pipeline register holds all necessary information (data and control signals) to continue execution into the next stage



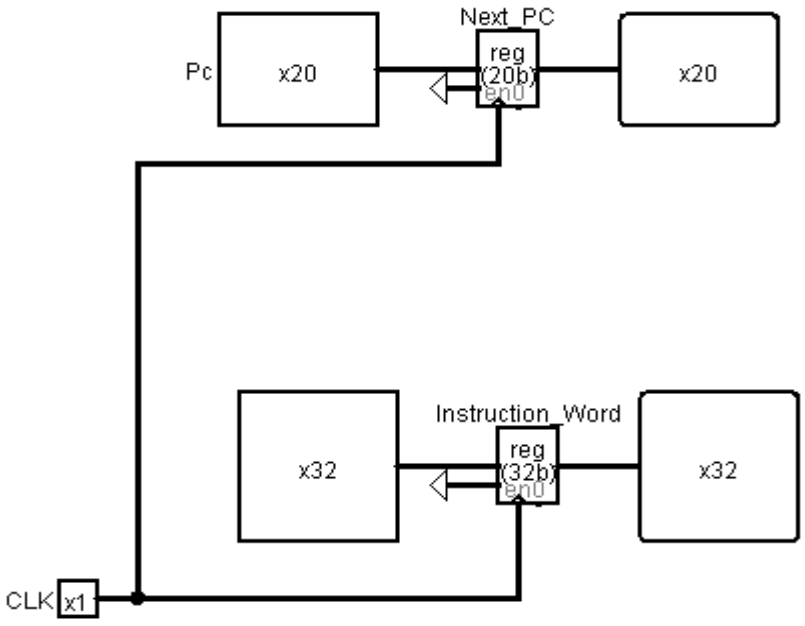
6.3 Pipeline Registers

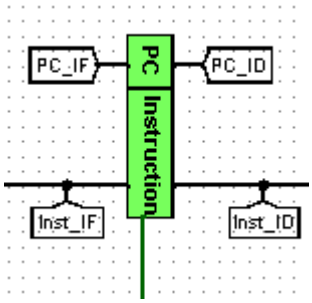
To support pipelining, we introduced four main sets of registers between the stages. These registers store data, addresses, and control signals needed for the next stage of instruction processing.

6.3.1 IF/ID Stage Pipeline Registers

- **Stored Values**:
    - Instruction fetched from memory
    - PC value
- **Purpose**:

These stored values are essential for the Decode (ID) stage, allowing it to extract instruction fields and perform PC-relative operations as needed.



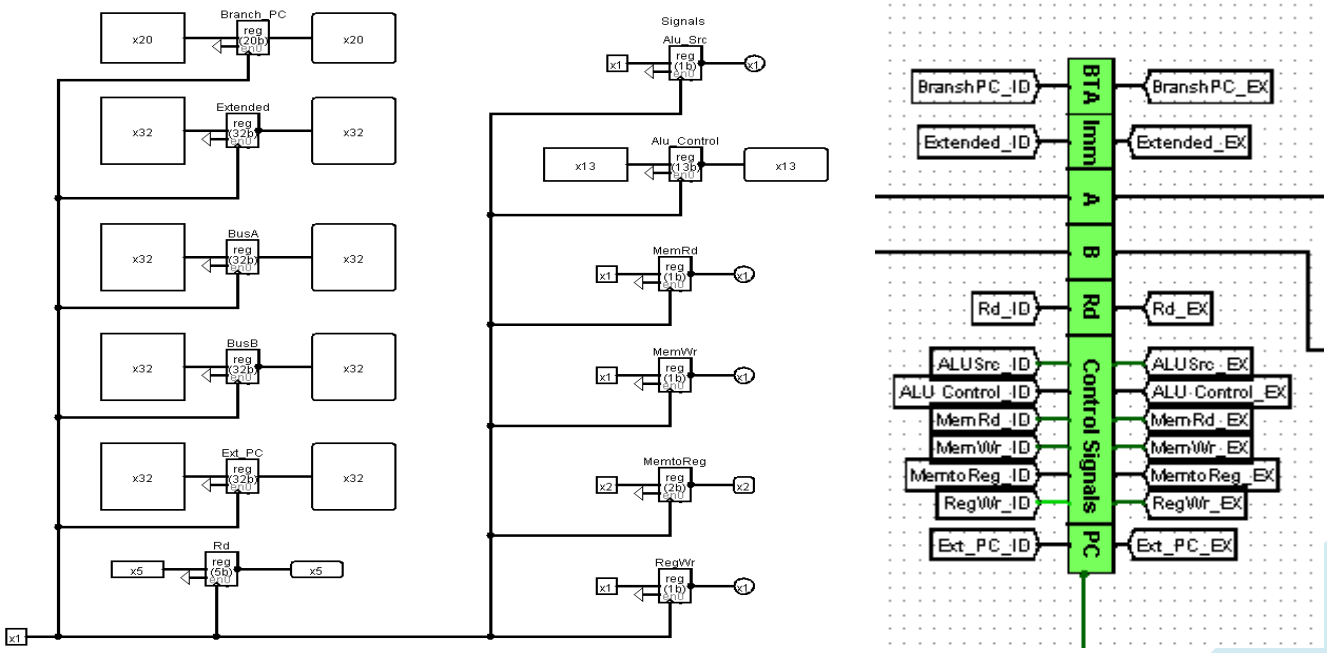


6.3.2 ID/EX Stage Registers

Stored Values:

- **Branch Target Address (Branch_PC):** Calculated address for potential branch instructions.
- **Extended Immediate (Extended):** The sign-extended immediate value extracted from the instruction.
- **Register Data (BusA, BusB):** Values read from the source registers (RS1 and RS2) during the decode stage.
- **Extended PC (Ext_PC):** The next program counter value, used for instructions such as `JALR`.
- **Destination Register (Rd):** The register number where the result should be written back in later stages.
- **Control Signals:**
  - `ALUSrc`, `ALU_Control`, `MemRd`, `MemWr`, `MemtoReg`, `RegWr`

Purpose:
The ID/EX pipeline register holds and transfers all required operand values and control signals from the Instruction Decode (ID) stage to the Execution (EX) stage. This allows the ALU and branching logic to receive the correct inputs and configuration signals needed to carry out the instruction properly in the next clock cycle.



6.3.3 EX/MEM Stage Register

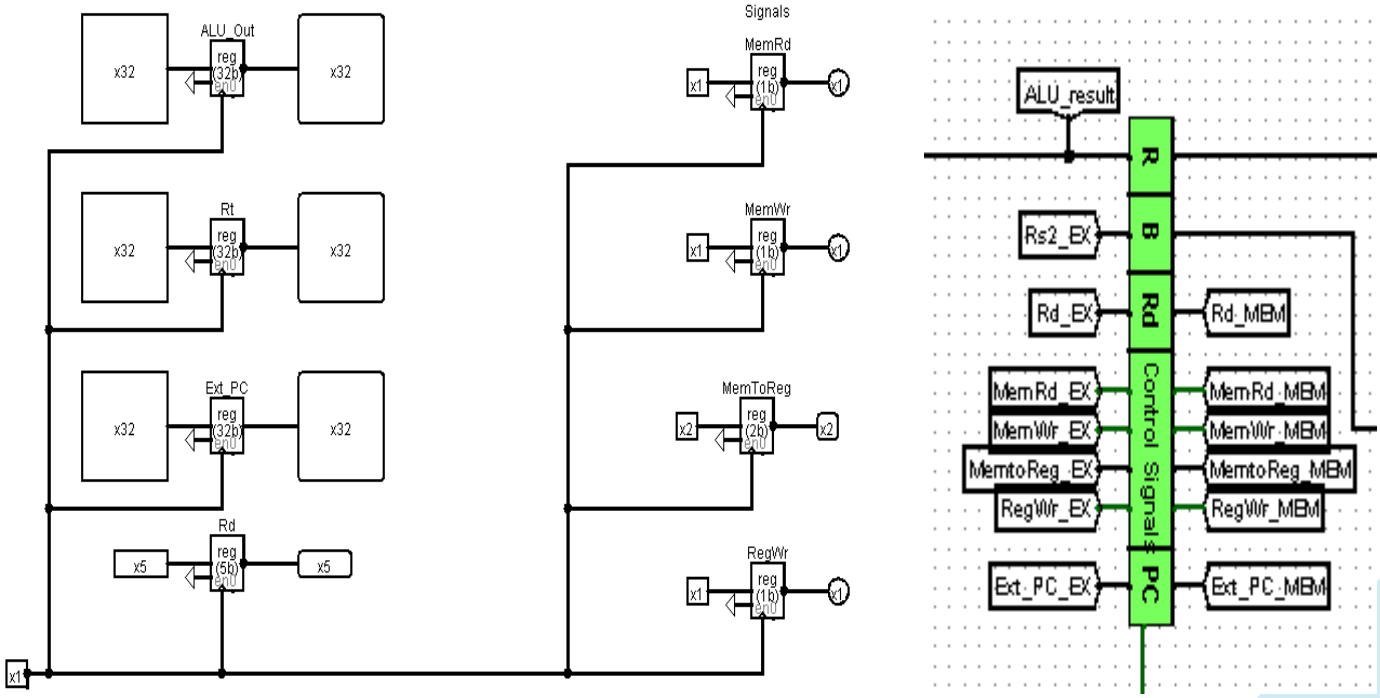
Stored Values:

- **ALU_Out:** The result produced by the ALU during the execution stage.
- **Rs2_EX:** The value of the second source register (RS2), required for store instructions (`SW`).
- **Ext_PC:** Extended program counter value used for `JALR` instructions.
- **Rd:** Destination register identifier for use in the Write-Back stage.

Control Signals:

- **MemRd:** Enables reading from data memory.
- **MemWr:** Enables writing to data memory.
- **MemtoReg:** Determines whether the value to write back comes from memory or the ALU.
- **RegWr:** Enables writing the result to the destination register.

Purpose:
The EX/MEM pipeline register holds the output of the Execution (EX) stage along with all necessary control signals required by the Memory (MEM) stage. It ensures proper data transfer for memory access and prepares all write-back-related information for the next pipeline phase.



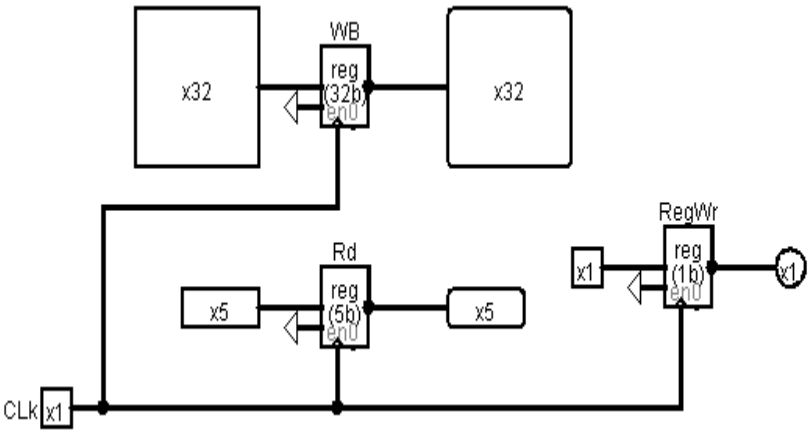
6.3.4 MEM/WB Stage Registers

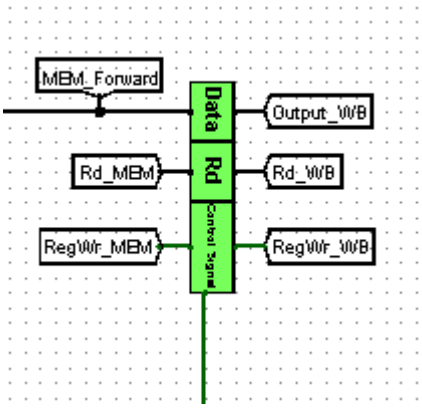
Stored Values:

- **WB_Data:** The final data to be written back to the register file. This data may originate from memory (in load instructions), the ALU (in arithmetic or logic instructions), or the PC (for `JALR` instructions).
- **Rd_MEM:** The destination register address where the result will be written.
- **RegWr_MEM:** Control signal that enables or disables writing to the register file.

Purpose:

The MEM/WB pipeline register holds the output data and corresponding control signals from the Memory (MEM) stage. It ensures that the correct result is delivered to the Write-Back (WB) stage and reliably updates the appropriate register in the processor's register file at the end of the instruction cycle.





6.4 Detailed Pipeline Stages

This section provides a comprehensive explanation of each pipeline stage in our 5-stage pipelined RISC processor. Each stage is responsible for a specific part of instruction execution and passes results forward through dedicated pipeline registers. Our design ensures correct instruction flow, hazard management, and data integrity using control signals, hazard detection, and forwarding logic.

6.4.1 Instruction Fetch (IF)

The **Instruction Fetch (IF)** stage is the entry point of the pipeline. Its primary function is to retrieve the next instruction from memory and determine the next Program Counter (PC) value based on control logic and program flow.

Program Counter (PC) and PC_IF Register

- The **PC_IF** register holds the address of the current instruction.
- This 20-bit register updates every cycle unless a hazard stalls the pipeline or a jump/branch alters the flow.
- The PC is incremented by 1 under normal conditions to fetch the next sequential instruction.

Next PC Selection Logic

A **4-input multiplexer (MUX)** determines the value of the next PC based on current execution context. The four possible sources are:

- **PC + 1**: Default next instruction address for normal, sequential execution.
- **JumpPC**: Target address for jump instructions (e.g., `JALR`), calculated as `RS1 + Imm16` in the **Jump Unit** during the ID stage.
- **BranchPC_EX**: Computed in the **Execute (EX)** stage when a conditional branch is taken. It is derived from the base PC and a sign-extended immediate offset.
- **PC_IF (Feedback):** The current PC is retained during stalls. When a hazard is detected, selecting this value freezes the PC and effectively reuses the same instruction until the hazard is resolved.

- The **PC Control Unit** manages this selection using control signals generated from:
  - The **Hazard Detection Unit**
  - Branch condition outcomes from the EX-stage
  - Jump signals from the ID stage

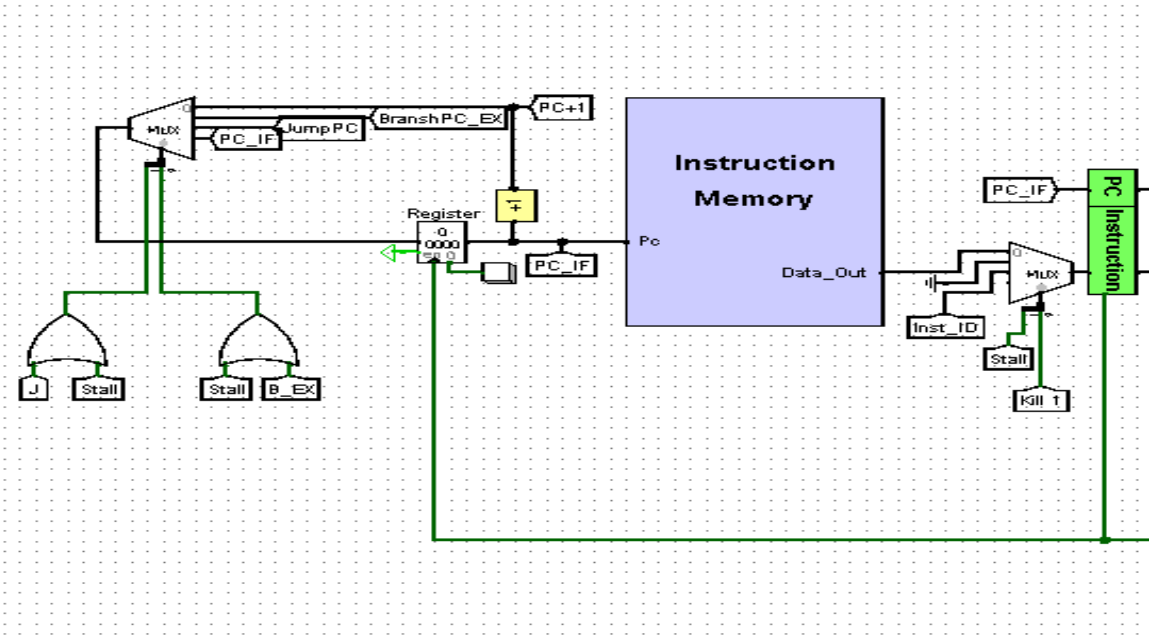
This system ensures the correct instruction address is selected for the next cycle while handling hazards and control flow transitions.

Instruction Memory Access

- Once the PC is selected, it is used to fetch a 32-bit instruction from **Instruction Memory**.
- This instruction is passed to the **Instruction Splitter** for decoding, and also stored in the **IF/ID pipeline register** along with `PC + 1` for use in subsequent stages.

Stall and Kill Logic

- **Stall**: If a load-use hazard is detected (e.g., a `LW` followed by a dependent instruction), the **Stall** signal disables updates to `PC_IF` and the IF/ID register, effectively pausing the pipeline.
- **Kill1**: When a jump or branch is taken, the **Kill1** signal flushes the incorrect instruction from the IF/ID register by replacing it with a NOP, ensuring no wrong-path instruction progresses.



6.4.2 Instruction Decode (ID)

The **Instruction Decode (ID)** stage is where the instruction's binary fields are interpreted, control signals are generated, register operands are read, and immediate values are processed.

Instruction Splitting

- The 32-bit instruction is routed to an **Instruction Splitter**, which extracts:
  - `Opcode`: Indicates instruction type
  - `Rd`: Destination register
  - `RS1`, `RS2`: Source registers
  - `Function`: Specific ALU operation
  - `Imm16`, `ImmU`, `ImmL`: Immediate fields for various instruction types

These fields are forwarded to multiple modules including the **Control Unit**, **Register File**, and **Immediate Extender**.

Register File Access

- `RS1` and `RS2` are used as addresses to read operand values from the **Register File**, producing `BusA` and `BusB`.
- These buses hold the input operands for the ALU and memory stages.
- The **RW** (write address) port is preloaded with `Rd` for future write-back in the WB stage.
- If a `RegWrite` signal is active in the WB stage, the value from `Output_WB` is written to the destination register.

Immediate Extension

- The **Immediate Extender** module receives the appropriate immediate field (selected via `ImmSelect` signal).
  - For I-type: `Imm16`
  - For SB-type: Concatenated `ImmU || ImmL`
- The **Ext_ID** signal determines whether the extension is signed or zero-extended.

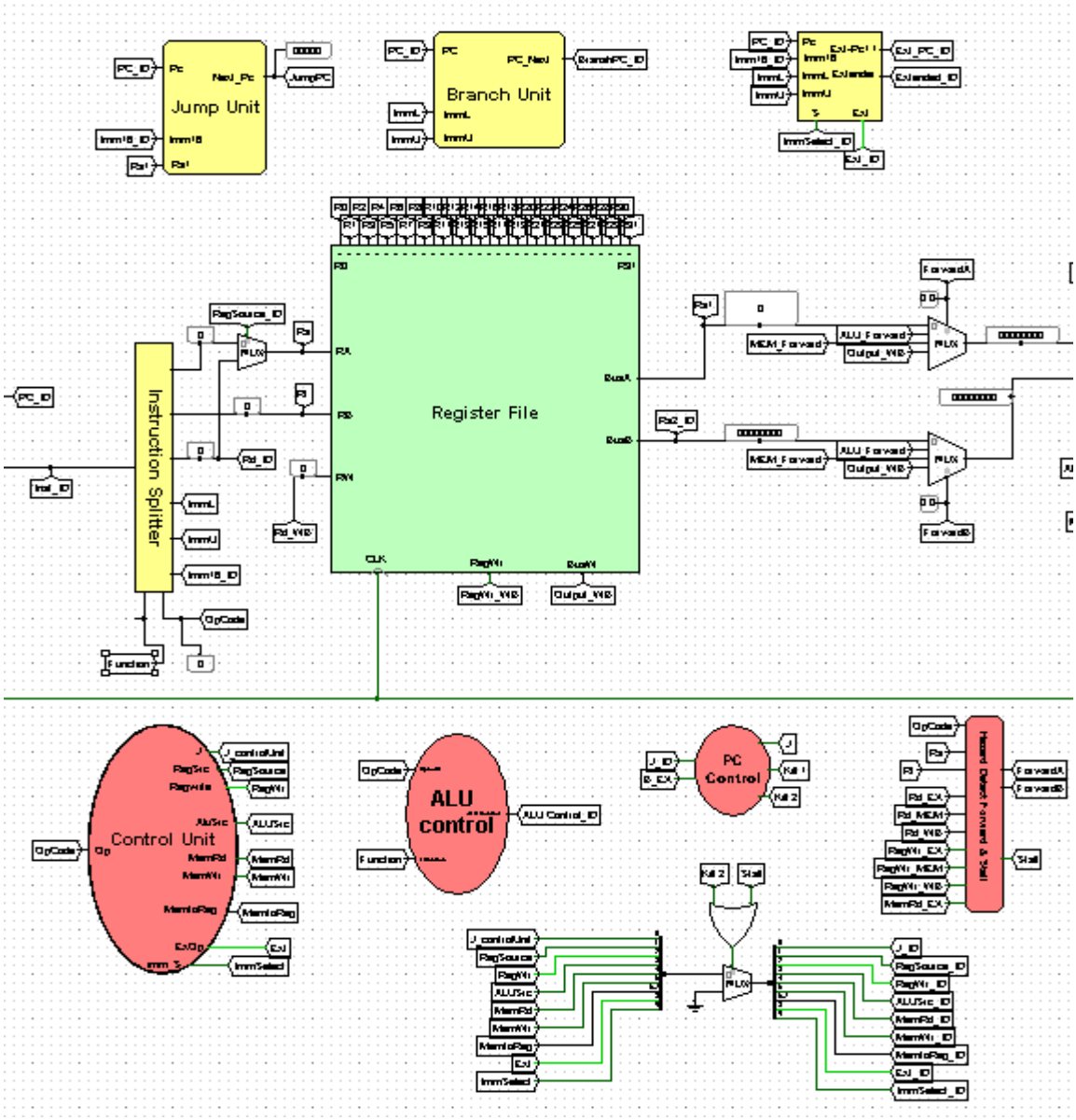
Control Unit and Signal Generation

- The **Control Unit** receives `Opcode`and `Function` and generates essential control signals:
  - `ALUSrc`: Select ALU operand (register or immediate)
  - `MemRead`, `MemWrite`: Enable memory operations
  - `RegWrite`: Enable write-back to register
  - `MemtoReg`: Choose result source (ALU or memory)
  - `Branch`, `Jump`: Enable control flow instructions
  - `Extender_Op`, `ImmSelect`, `RegSource`: Control immediate and destination register behaviors

These signals are passed to the next stage via the **ID/EX pipeline register**.

Hazard Detection and Stall Logic

- The **Hazard Unit** checks if the next instruction depends on a value not yet written back.
- If a **load-use hazard** is detected (e.g., instruction uses a register that a prior `LW` will write), the unit:
  - Asserts **Stall**: freezes PC and IF/ID register
  - Insert a bubble (NOP) into the ID/EX pipeline register by zeroing out all control signals.
- If a branch or jump decision invalidates earlier instructions, the **Kill1** signal flushes the IF/ID register.


Jump Unit
Branch Unit
Register File
Instruction Splitter
Control Unit
ALU control
PC Control

6.4.3 Execution (EX)

The **Execution (EX)** stage performs arithmetic, logical, and branch computations. It is the core of instruction processing.

ALU Operation

- The **ALU** receives two operands:
  - Operand 1: Always `BusA_EX`, sourced from the register file or forwarded data
  - Operand 2: Selected via MUX:
    - From `BusB_EX` if `ALUSrc_EX = 0`
    - From `Extended_EX` if `ALUSrc_EX = 1`
- The **ALU Control Unit** interprets the `Function` and `Opcode` to configure the ALU for the correct operation (ADD, SUB, AND, SLT, etc.).

Branch Evaluation

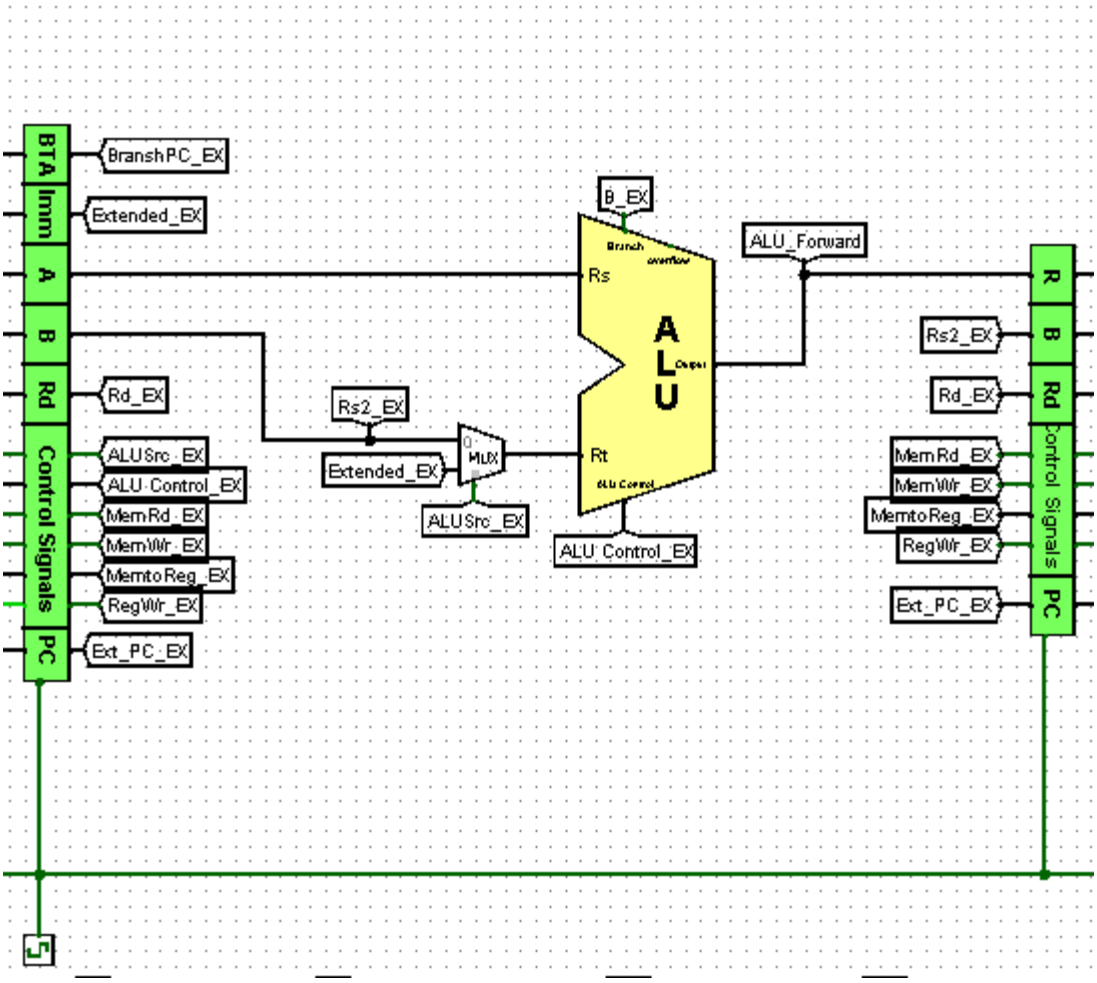
- For branch instructions (e.g., `BEQ`, `BNE`), the **Branch Unit** uses operands to evaluate conditions and compute the target address
- (`BranchPC_EX = PC_ID + Imm_Ext`).
- If the condition is met, `Branch` signal is asserted, and `BranchPC_EX` is passed to the PC Control to update the PC.

Forwarding Logic

- The **Forwarding Unit** resolves data hazards without stalling by checking if operands are available in later stages (MEM or WB).
- If needed, `ForwardA` and `ForwardB` MUXes select forwarded values from:
  - ALU result in EX/MEM
  - Write-back result in MEM/WB
- This ensures correct operand values even if instructions overlap in execution.

Control Signal Propagation

- The following are passed to the MEM stage via the **EX/MEM** register:
  - `ALU result`
  - `MemRead_EX`, `MemWrite_EX`, `MemtoReg_EX`, `RegWrite_EX`
  - Destination register `Rd_EX`
  - `BusB_EX` (used as write data for `SW`)



6.4.4 Memory Access (MEM)

The **Memory Access (MEM)** stage interacts with **Data Memory** to read or write data based on the instruction type.

Data Memory Operations

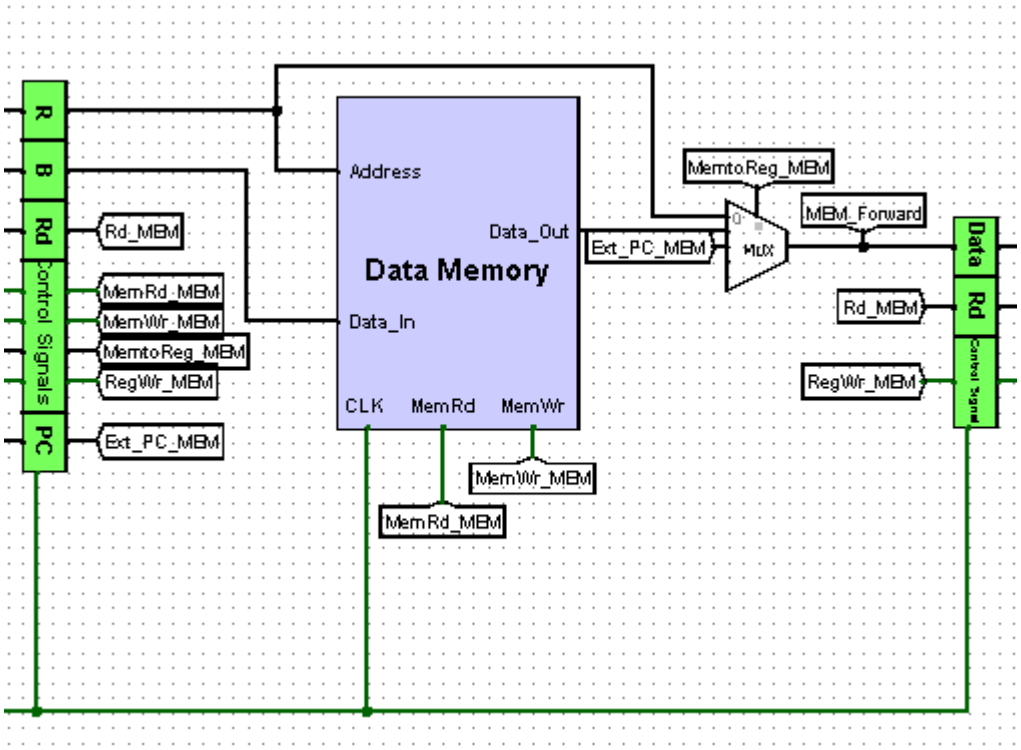
- **Address** input is sourced from `ALU result` computed in EX stage.
- **Data Input** (for stores) comes from `BusB_MEM`, previously read from the register file.
- Controlled by:
  - `MemRead_MEM`: Enables memory reads
  - `MemWrite_MEM`: Enables memory writes
- **Data Output** (for loads) is stored in `Data_Out`

Result Selection for WB

- A **MUX** selects the correct result to write back:
  - If `MemtoReg_MEM = 0`: Choose `ALU result`
  - If `MemtoReg_MEM = 1`: Choose `Data_Out` from memory (LW)
  - If `MemtoReg_MEM = 2`: Choose `Ext_PC_MEM` For JALR Instruction
- Output of this MUX becomes `WriteBack_Data`.

Control Signal Forwarding

- The MEM stage passes control and register information to the WB stage through the **MEM/WB register**:
  - `WriteBack_Data`
  - `Rd_MEM` (destination register)
  - `RegWrite_MEM`, `MemtoReg_MEM` (controls WB behavior)



6.4.5 Write Back (WB)

The Write Back (WB) stage is the final step in the instruction execution pipeline, responsible for updating the register file with the result of the instruction.

Result Selection

A multiplexer, controlled by the `MemtoReg_WB` signal, selects the value to be written back to the register file:

- If `MemtoReg_WB = 0`: The value is `ALU_Out_WB`, the output from the ALU.
- If `MemtoReg_WB = 1`: The value is `Data_Out_WB`, the data retrieved from memory (used in load instructions).
- If `MemtoReg_WB = 2`: The value is `Ext_PC_WB`, the incremented program counter used for `JALR` or `JAL` instructions.

Register File Update

- The selected value is sent through the `BusW` line.
- It is written to the destination register specified by `Rd_WB`.
- The write occurs only if the `RegWrite_WB` signal is asserted (i.e., set to 1), enabling the write operation in the register file.

6.5 Hazard Detect Forward & Stall

The Hazard **Detect Forward & Stall Unit** is responsible for resolving data hazards in the pipelined processor, especially those involving dependencies between instructions in different pipeline stages.

This unit compares the **source registers (Rs & Rt) of the instruction currently in the ID stage** with the **destination registers (Rd) of instructions in the EX, MEM, and WB stages**.It generates the following control signals:

- **ForwardA and ForwardB (2-bit each):**
  Used to control multiplexers after the Register File in the ID stage. These select the correct operand source:
  - **00** → use value from register file (normal)
  - **10** → forward from **EX stage**
  - **10** → forward from **MEM stage**
  - **01** → forward from WB stage
- **Stall:**
  Activated when a **load-use hazard** is detected — that is, when the instruction in the EX stage is a load (**MemRead_EX = 1**), and its destination register matches a source register of the instruction in the ID stage

Optimizations:

1. **Register Zero Ignored:**
   If any of the detected registers (**Rs, Rt**) is register zero (R0), forwarding and stalling are skipped since R0 always contains zero and does not change.

2. **Immediate Instructions:**
   For I-type (immediate) instructions, the **Rt** field in the ID stage does not represent a real register operand. It is part of the 16-bit immediate value, so **hazard detection should only compare Rs**, and ignore **Rt**.

These optimizations reduce unnecessary forwarding and prevent false hazard detection, improving performance and keeping the hazard logic simple.

When the **Stall** signal is asserted:

- The **PC** and **IF/ID** pipeline registers are **frozen** (i.e., they do not update on the next clock cycle).
- The control signals passed into the **ID/EX** pipeline register are **cleared to zero**, effectively injecting a **NOP (bubble)** into the pipeline.

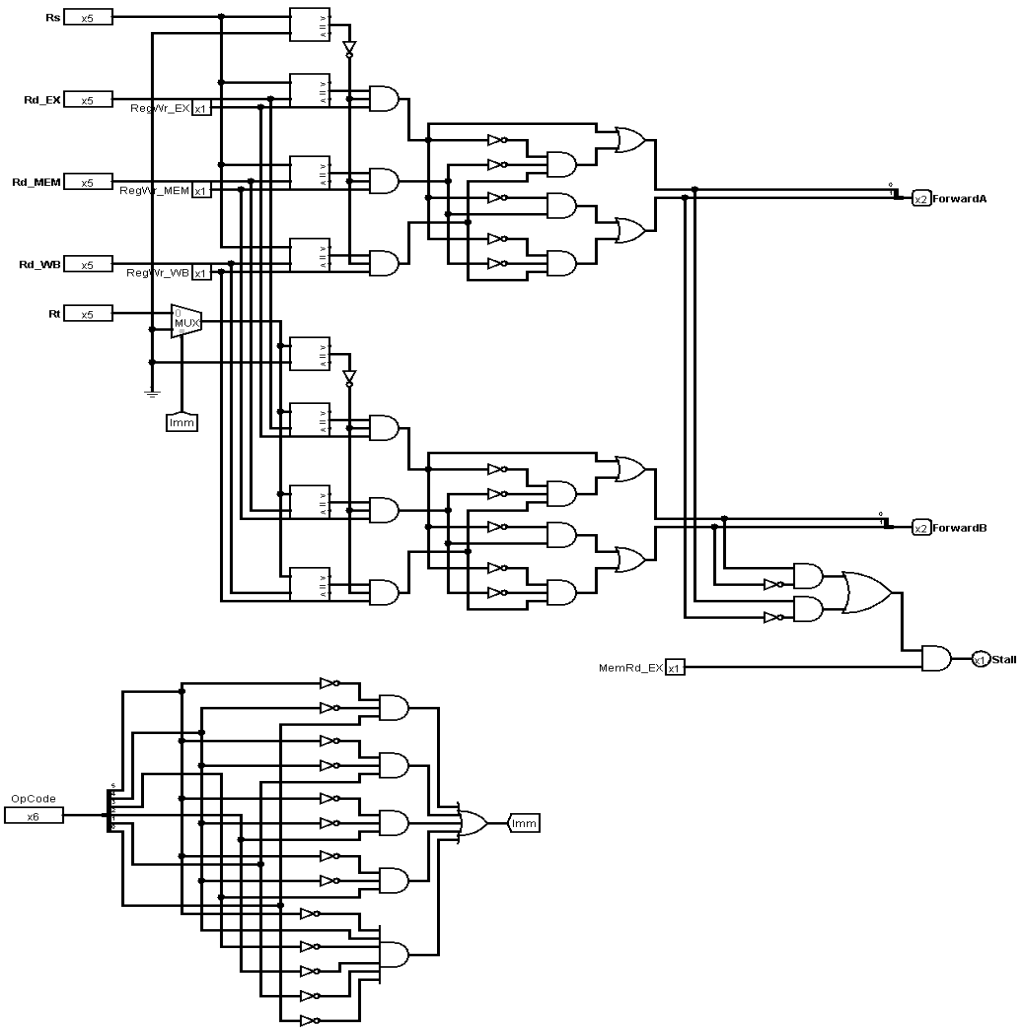


Figure: Hazard Detect Forward & stall

6.6 Hazard Pc Control

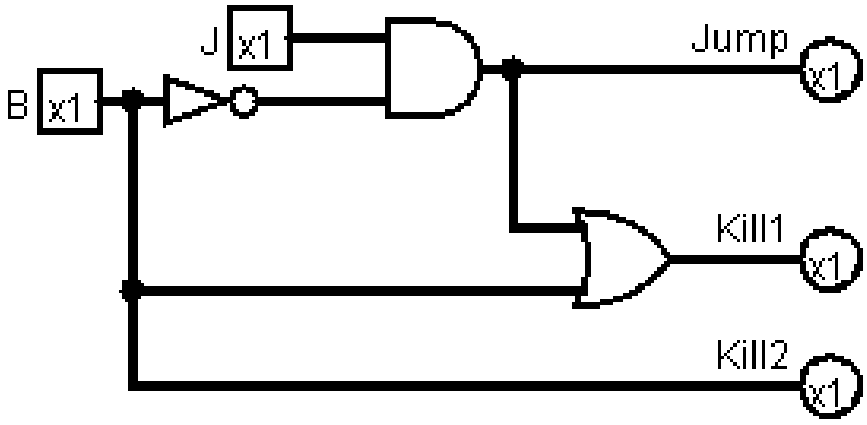


Figure: Hazard Pc Control

The Hazard PC Control Unit is responsible for resolving control hazards caused by branch (B) and jump (J) instructions in the pipelined processor.

Control hazards occur when the processor is unsure of which instruction to fetch next due to a possible change in control flow — typically from branches or jumps.

This unit has two input signals:

- B: Indicates a branch instruction is currently in the pipeline.
- J: Indicates a jump instruction is currently in the pipeline.

It performs simple logic to determine the correct control flow action, and whether to flush instructions that have already been fetched.

It generates the following control signals:

Jump

Asserted when there is a jump (`J = 1`) but no branch (`B = 0`). This triggers an unconditional change to the Program Counter (PC).

- **Logic:** `Jump = J AND (NOT B)`
- **Behavior:** Only true for standalone jump instructions.

Kill1 and Kill2

Both signals are used to flush instructions in earlier pipeline stages (e.g., IF/ID and ID/EX) when a control hazard is detected.

- **Logic:** `Kill1 = Kill2 = B OR Jump`

Behavior:
Flushing occurs either for branches (`B = 1`) or jumps (`Jump = 1`).
Ensures that incorrect instructions following a control flow change are discarded.

Optimizations:

This hazard unit uses minimal logic for fast decision-making, allowing early detection and flushing of incorrect instructions.

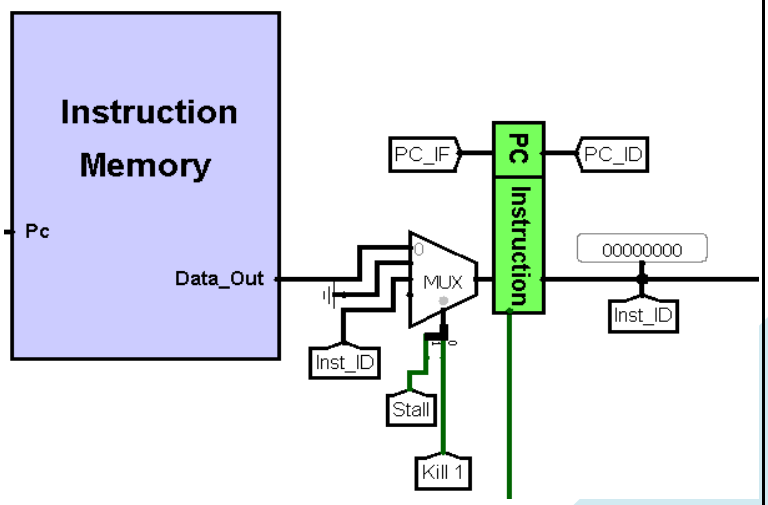
- **Unified Kill Logic:** Instead of separate conditions for each pipeline stage, both `Kill1` and `Kill2` share the same logic, reducing hardware complexity.
- **Jump vs Branch Distinction:** Ensures jumps and branches are treated differently, avoiding misprediction-based hazards for jumps (since they are unconditional).

When Kill signals are asserted:

- The instructions in the IF and ID pipeline stages are flushed.
- This is typically done by zeroing out control signals

  or injecting NOPs (bubbles).
- Prevents execution of wrong-path instructions, maintaining

  correct program behavior.



7.Test Program Results

A comprehensive test program was executed to verify the functionality of the pipelined processor designed in Phase 2. The program covers arithmetic, logical, memory, and control flow instructions, while also testing hazard resolution, forwarding, stalling, and pipeline flushing.

The table below presents the instruction trace, showing the **expected vs. actual outcomes** for each executed instruction. Observations confirm that the datapath stages, control logic, and hazard handling units operate correctly across various instruction types.

Instruction	address	Expected value	Actual value
SET R1, 0x0384	0x0000B	R1 = 0x00000384	R1 = 0x00000384
SET R8, 0x1234	0x0000C	R8 = 0x00001234	R8 = 0x00001234
SSET R8, 0x5678	0x0000D	R8 = 0x12345678	R8 = 0x12345678
ADDI R5, R1, 20	0x0000E	R5 = 0x00000398	R5 = 0x00000398
XOR R3, R1, R5	0x0000F	R3 = 0x0000001C	R3 = 0x0000001C
ADD R4, R8, R3	0x00010	R4 = 0x12345694	R4 = 0x12345694
LW R1, 0(R0)	0x00011	R1 = Mem[0] = 0x00000001	R1 = Mem[0] = 0x00000001
LW R2, 1(R0)	0x00012	R2 = Mem[1] = 0x00000001	R2 = Mem[1] = 0x00000001
LW R3, 2(R0)	0x00013	R3 = Mem[2] = 0x0000000A	R3 = Mem[2] = 0x0000000A
SUB R4, R4, R4	0x00014	R4 = 0x00000000	R4 = 0x00000000
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x00000001	R4 = 0x00000001
SLT R6, R2, R3	0x00016	R6 = 0x00000001	R6 = 0x00000001
BEQ R6, R0, done	0x00017	Branch = 0	Branch = 0
ADD R2, R1, R2	0x00018	R2 = 0x00000002	R2 = 0x00000002
BEQ R0, R0, Loop1	0x00019	Branch = 1	Branch = 1
done: SW R4, 0(R0)	0x0001a	killed	killed
MUL R10, R2, R3	0x0001b	killed	killed
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x00000003	R4 = 0x00000003

SLT R6, R2, R3	0x00016	R6 = 0x00000001	R6 = 0x00000001
BEQ R6, R0, done	0x00017	Branch = 0	Branch = 0
ADD R2, R1, R2	0x00018	R2 = 0x00000003	R2 = 0x00000003
BEQ R0, R0, Loop1	0x00019	Branch = 1	Branch = 1
done: SW R4, 0(R0)	0x0001a	killed	killed
MUL R10, R2, R3	0x0001b	killed	killed
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x00000006	R4 = 0x00000006
SLT R6, R2, R3	0x00016	R6 = 0x00000001	R6 = 0x00000001
BEQ R6, R0, done	0x00017	Branch = 0	Branch = 0
ADD R2, R1, R2	0x00018	R2 = 0x00000004	R2 = 0x00000004
BEQ R0, R0, Loop1	0x00019	Branch = 1	Branch = 1
done: SW R4, 0(R0)	0x0001a	killed	killed
MUL R10, R2, R3	0x0001b	killed	killed
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x0000000A	R4 = 0x0000000A
SLT R6, R2, R3	0x00016	R6 = 0x00000001	R6 = 0x00000001
BEQ R6, R0, done	0x00017	Branch = 0	Branch = 0
ADD R2, R1, R2	0x00018	R2 = 0x00000005	R2 = 0x00000005
BEQ R0, R0, Loop1	0x00019	Branch = 1	Branch = 1
done: SW R4, 0(R0)	0x0001a	killed	killed
MUL R10, R2, R3	0x0001b	killed	killed
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x0000000f	R4 = 0x0000000f
SLT R6, R2, R3	0x00016	R6 = 0x00000001	R6 = 0x00000001
BEQ R6, R0, done	0x00017	Branch = 0	Branch = 0
ADD R2, R1, R2	0x00018	R2 = 0x00000006	R2 = 0x00000006
BEQ R0, R0, Loop1	0x00019	Branch = 1	Branch = 1

done: SW R4, 0(R0)	0x0001a	killed	killed
MUL R10, R2, R3	0x0001b	killed	killed
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x00000015	R4 = 0x00000015
SLT R6, R2, R3	0x00016	R6 = 0x00000001	R6 = 0x00000001
BEQ R6, R0, done	0x00017	Branch = 0	Branch = 0
ADD R2, R1, R2	0x00018	R2 = 0x00000007	R2 = 0x00000007
BEQ R0, R0, Loop1	0x00019	Branch = 1	Branch = 1
done: SW R4, 0(R0)	0x0001a	killed	killed
MUL R10, R2, R3	0x0001b	killed	killed
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x0000001C	R4 = 0x0000001C
SLT R6, R2, R3	0x00016	R6 = 0x00000001	R6 = 0x00000001
BEQ R6, R0, done	0x00017	Branch = 0	Branch = 0
ADD R2, R1, R2	0x00018	R2 = 0x00000008	R2 = 0x00000008
BEQ R0, R0, Loop1	0x00019	Branch = 1	Branch = 1
done: SW R4, 0(R0)	0x0001a	killed	killed
MUL R10, R2, R3	0x0001b	killed	killed
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x00000024	R4 = 0x00000024
SLT R6, R2, R3	0x00016	R6 = 0x00000001	R6 = 0x00000001
BEQ R6, R0, done	0x00017	Branch = 0	Branch = 0
ADD R2, R1, R2	0x00018	R2 = 0x00000009	R2 = 0x00000009
BEQ R0, R0, Loop1	0x00019	Branch = 1	Branch = 1
done: SW R4, 0(R0)	0x0001a	killed	killed
MUL R10, R2, R3	0x0001b	killed	killed
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x0000002D	R4 = 0x0000002D
SLT R6, R2, R3	0x00016	R6 = 0x00000001	R6 = 0x00000001

BEQ R6, R0, done	0x00017	Branch = 0	Branch = 0
ADD R2, R1, R2	0x00018	R2 = 0x0000000A	R2 = 0x0000000A
BEQ R0, R0, Loop1	0x00019	Branch = 1	Branch = 1
done: SW R4, 0(R0)	0x0001a	killed	killed
MUL R10, R2, R3	0x0001b	killed	killed
Loop1: ADD R4, R2, R4	0x00015	R4 = 0x00000037	R4 = 0x00000037
SLT R6, R2, R3	0x00016	R6 = 0x00000000	R6 = 0x00000000
BEQ R6, R0, done	0x00017	Branch = 1	Branch = 1
ADD R2, R1, R2	0x00018	killed	killed
BEQ R0, R0, Loop1	0x00019	killed	killed
done: SW R4, 0(R0)	0x0001A	Mem[0] = 0x00000037, R0 = 0x00000000	Mem[0] = 0x00000037, R0 = 0x00000000
MUL R10, R2, R3	0x0001B	R10 = 0x00000064	R10 = 0x00000064
SRL R14, R10, R4	0x0001C	R14 = 0x00000000	R14 = 0x00000000
SRA R15, R10, R4	0x0001D	R15 = 0x00000000	R15 = 0x00000000
RORI R26, R14, 5	0x0001E	R26 = 0x00000000	R26 = 0x00000000
JALR R7, R0, func	0x0001F	PC = func, R7 = 0x00000020	PC = func, R7 = 0x00000020
SET R9, 0x4545	0x00020	killed	killed
func: OR R5, R2, R3	0x00025	R5 = 0x0000000A	R5 = 0x0000000A
LW R1, 0(R0)	0x00026	R1 = 0x00000037	R1 = 0x00000037
LW R2, 5(R1)	0x00027	R2 = 0x128945AC	R2 = 0x128945AC
LW R3, 6(R1)	0x00028	stall	stall
LW R3, 6(R1)	0x00028	R3 = 0x05007342	R3 = 0x05007342
AND R4, R2, R3	0x00029	R4 = 0x00004100	R4 = 0x00004100

SW R4, 0(R0)	0x0002A	stall	stall
SW R4, 0(R0)	0x0002A	Mem[0] = 0x00004100	Mem[0] = 0x00004100
JALR R0, R7, 0	0x0002B	PC = 0x00020, R0 = 0x00000000	PC = 0x00020, R0 = 0x00000000
-	0x0002C	killed	killed
SET R9, 0x4545	0x00020	R9 = 0x00004545	R9 = 0x00004545
SET R10, 0x4545	0x00021	R10 = 0x00004545	R10 = 0x00004545
BGE R10, R9, L1	0x00022	Branch = 1	Branch = 1
ANDI R23, R1, 0xFFFF	0x00023	killed	killed
L1: BEQ R0, R0, L1	0x00024	killed	killed
L1: BEQ R0, R0, L1	0x00024	infinite loop: End	infinite loop: End
func: OR R5, R2, R3	0x00025	killed	killed
LW R1, 0(R0)	0x00026	killed	killed

Highlights:

- **Arithmetic and logic operations** executed correctly across registers and immediate values.
- **Memory instructions (`LW`, `SW`)** performed as expected with proper address calculation and data access.
- **Looping and branching** were evaluated accurately with correct PC updates.
- **Forwarding and stalling mechanisms** resolved register hazards, ensuring uninterrupted execution.
- **Killed instructions** reflect proper pipeline flushing on control hazards.
- **Jump-and-link (`JALR`)** functionality worked correctly, confirming multi-stage control flow.

The observed results matched expectations in all cases, confirming the correctness and robustness of the pipelined design.

8. Work Details

8.1 Tasks Per Team Member

Abdullah Mostafa Gomaa

- **ALU Design and Implementation**:
  - Designed and implemented the Arithmetic Logic Unit (ALU).
- **ALU Control Modules Development**:
  - Developed Arithmetic Control.
  - Developed Logic Control.
  - Developed Shift Control.
  - Developed Set-on-Less-Than (SET) Control.
  - Developed Branch Control.
- **Processor Enhancement**:
  - Enhanced processor blocks and finalized frontend design.
- **Pipeline Hazard Detection**:
  - Implemented pipeline hazard detection, forwarding, and stalling.
- **Main Testing**:
  - Conducted testing of processor functionality (Single Cycle and Pipeline).
- **Video Editing and Voice-over**:
  - Edited project video.
  - Provided voice-over for project video.
- **Documentation**:
  - Contributed to writing project documentation collaboratively.

Abdelrahman Zaki Ibrahim

- **Program Counter (PC) Design**:
  - Designed PC increment logic.
  - Implemented Jump Unit and Branch Unit.
  - Developed PC control logic for jump, branch, and stall handling.
- **Instruction Memory**
- **Instruction Splitter Design:** to decode and distribute instruction fields
- **Control Unit Components Design**:
  - Designed ALU input selection and extension handling.
  - Developed immediate selection logic.
- **Pipeline Architecture**:
  - Implemented pipeline architecture and all register stages (IF/ID, ID/EX, EX/MEM, MEM/WB).
- **Project Documentation**:
  - Led the writing and editing of the project report, covering system design, module functionality, and implementation details. Fully formatted and organized the documentation for a clear, professional, and cohesive final submission.
- **Testing**:
  - Performed testing of processor functionality (Single Cycle and Pipeline).
- **Video Voice-over**:
  - Provided voice-over for project video.

Moataz Mohamed Ali

- **Register and Data Memory Design**:
  - Designed and tested the register file.
  - Developed and integrated data memory.
- **Immediate Extender Module**:
  - Implemented the immediate extender module.
- **Control Unit Component Design**:
  - Designed storage control logic.
- **Processor Integration**:
  - Integrated the full processor and completed wiring.
- **Pipeline Hazard Control**:
  - Implemented pipeline hazard control.
- **Final System Testing**:
  - Conducted final system testing.
- **Video Voice-over**:
  - Provided voice-over for project video.
- **Documentation**:
  - Contributed to writing project documentation collaboratively.
- **Testing**:
  - Performed testing of processor functionality (Single Cycle and Pipeline).

8.2 Workflow and Tools

To ensure seamless collaboration and efficient progress, the team utilized several tools and platforms throughout the project:

- **Logisim**
    - Used for designing and simulating processor components and the complete Datapath.
    - Allowed isolated testing of individual modules before full system integration.
- **GitHub**
    - Used for version control, file sharing, and collaboration across modules.
- **Discord**
    - Enabled voice/video meetings, real-time updates, and collaborative debugging during remote work sessions

8.3 Meetings and Collaboration Timeline

Date	Time	Purpose
16/04/2025	10:00 PM – 12:30 AM	Initial meeting: Reviewed project document and distributed tasks.
19/04/2025	8:45 PM – 9:15 PM	Discord meeting: Discussed PC counter design and implementation.
20/04/2025	4:30 PM – 2:00 AM	Work session: Developed subsets of the processor (ALU, Branch Control, Control Signals).
22/04/2025	4:00 PM – 3:00 AM	Finalized module designs and integrated wiring of the Single-Cycle Processor.
23/04/2025	9:15 PM – 9:45 PM	the first full system test.
24/04/2025	2:00 PM – 6:00 PM and 8:30 PM – 5:00 AM	Testing, finalized integration, and started processor documentation.
26/04/2025	12:00 AM – 4:00 PM	Discord meeting: Test code verification and documentation updates.
27/04/2025	4:00 PM – 12:00 AM	Continue documentation and project video recording.
28/04/2025	9:00 PM – 12:00 PM	Final documentation preparation and project video editing
2/05/2025	5:00 PM – 11:30 PM	Starting The Pipeline
6/05/2025	5:00 PM – 2:00 AM	Finishing and Testing
2/05/2025	12:30 AM – 2:00 AM	
8/05/2025	11:40 PM – 1:30 AM	