

# **RISC Processor Documentation**

**By:**

**Alaa ayman mohamed**

**Omnia Ayman Mohamed**

**Rahma mostafa**

## 1.2 Introduction

The primary goal of this project is to design a 32-bit single-cycle RISC processor and simulate its functionality using Logisim. A single-cycle processor architecture is one where each instruction is executed in a single clock cycle, which simplifies the design but limits the overall performance due to the fixed time needed to execute any given instruction.

This project specifically focuses on implementing a custom Instruction Set Architecture (ISA) with three types of instruction formats: R-type, I-type, and SB-type. Each of these formats corresponds to different kinds of operations, ranging from arithmetic and logical operations to memory access and branching operations. The architecture supports 31 general-purpose registers (R1 to R31) and a special register R0 that is always set to zero.

The ISA will allow a range of operations:

1. Arithmetic operations (addition, subtraction, etc.)
2. Logical operations (AND, OR, etc.)
3. Memory operations (load and store)
4. Branch operations (conditional jumps)

In a single-cycle processor design, each instruction must pass through all stages of the execution in one cycle. These stages include fetching the instruction from memory, decoding the instruction to understand what operation to perform, executing the operation (e.g., arithmetic), memory access (if necessary), and writing back the result to the register file. Although this design is simple and easy to implement, it is not as efficient as pipelined designs -will focus on phase2- because every instruction, regardless of its complexity, takes the same amount of time to execute.

The project will achieve the design and simulation of a single-cycle processor, capable of executing a predefined set of instructions in one clock cycle. It will demonstrate a solid understanding of the fundamental operations within a CPU, including instruction fetching, decoding, execution, and memory access. The processor will be tested using assembly code to ensure correct execution of instructions, with a focus on achieving efficient data flow between the processor's components. By completing this project, the team will gain hands-on experience with digital logic design and simulation tools like Logisim.

The technologies used for this project are primarily based on Logisim for the design and simulation of the digital processor. Logisim is an intuitive tool that allows for the construction of complex circuits, providing an easy interface for the design and simulation of processor architectures.

At the end, this project is the result of strong collaboration and coordinated efforts within our team. Each member played a crucial role in ensuring the success of the design and simulation of the processor. From the initial planning stages to the final testing and debugging phases, our team worked together closely to divide tasks efficiently, ensuring that each aspect of the project was completed with precision. Team members with expertise in digital logic design focused on the architecture and circuits, while others contributed by writing and testing the

assembly code to ensure the processor functioned correctly. Regular communication and collaborative problem-solving helped us address challenges and fine-tune the processor design to meet the project objectives that all will be discussed later.

## 2.1 Register File

The Register File is a critical component in the architecture of the single-cycle MIPS processor. It serves as a fast, accessible storage unit for temporary data, supporting the execution of computational and memory operations by providing immediate access to operands and storage for results. In the single-cycle processor, each instruction completes its entire operation—including register reads and writes—within a single clock cycle, necessitating a highly efficient and reliable Register File design.

### 1. Structure and Organization

The Register File contains **32 general-purpose registers**:

- Labeled R0 through R31.
- Each register is 32 bits wide, matching the processor's word size.
- It supports **two read ports** and **one write port**:
  - This configuration enables reading two operands simultaneously while writing back a result in the same clock cycle, which is essential for most instruction types (e.g., ALU operations, memory loads).

The Register File is fully combinational for reads (zero-cycle access latency) and synchronous for writes (writing occurs on the rising edge of the clock).

### 2. Register R0

Register R0 plays a unique role:

- It is hardwired to the constant value 0.
- Any read from R0 always returns a 32-bit zero.
- Any write operation directed at R0 is ignored, and the contents of R0 remain unaffected.

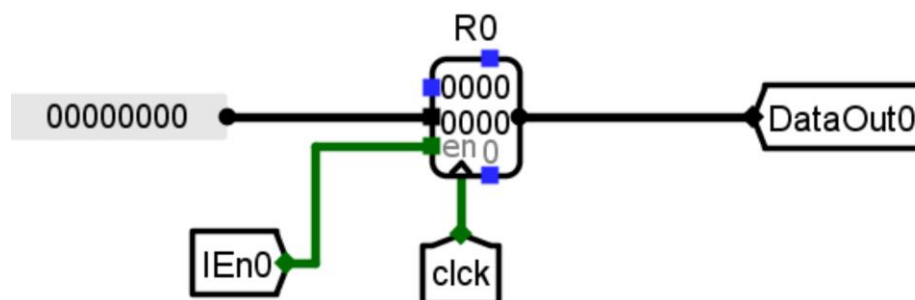


Figure 2.2.1

The presence of R0 simplifies the implementation of operations requiring a constant zero value and aids in programmatic tasks such as clearing registers, conditional branching, and address calculations.

### 3.Functional Operation

Each instruction that interacts with the Register File follows a simple, efficient model:

- **Read Phase:**
  - Two registers are specified as source operands using the instruction fields **S1** and **S2**.
  - The contents of the selected registers are fetched simultaneously through two independent read ports.
- **Write Phase:**
  - If the instruction specifies a destination register and the write-enable signal is asserted, the result of the instruction is written to the specified register (**d field**).
  - Write operations occur synchronously on the rising edge of the clock signal.

### 4. Inputs, Outputs, and Control Signals

The Register File interfaces with the rest of the processor through the following signals:

Name	Width	Direction	Description
Read Register 1 (RA)	5 bits	Input	Address of the first register to be read
Read Register 2 (RB)	5 bits	Input	Address of the second register to be read
Write Register (RW)	5 bits	Input	Address of the register to be written
Write Data (BusW)	32 bits	Input	Data to be written into the destination register
Write Enable (RegWrite)	1 bit	Input	Enables writing when asserted high
Read Data 1 (BusA)	32 bits	Output	Data read from register specified by S1

Read Data 2      32 bits    Output      Data read from register specified by S2 (BusB)

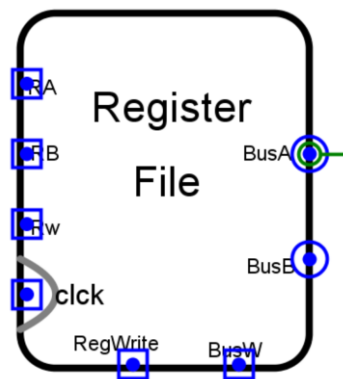


Figure 2.2.2

## 5. Internal Implementation

Internally, the Register File consists of:

- **32x32-bit Register Array**
- **Two Read Multiplexers:**
  - Each read port uses a 32-to-1 multiplexer to select one of the 32 available registers based on the 5-bit input address.

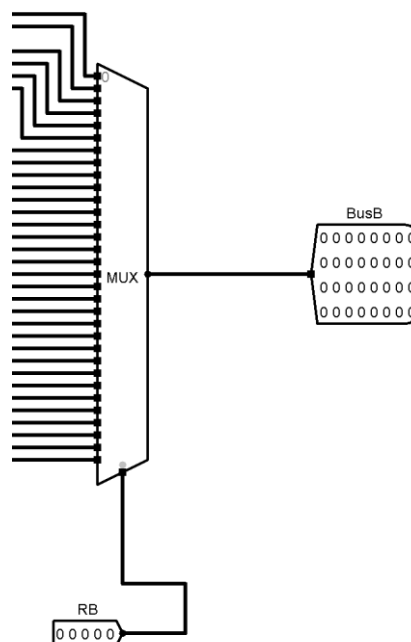


Figure 2.2.3

- **One Write Decoder:**

- A 5-to-32 decoder translates the 5-bit write address into a one encoded signal to enable exactly one register for writing.

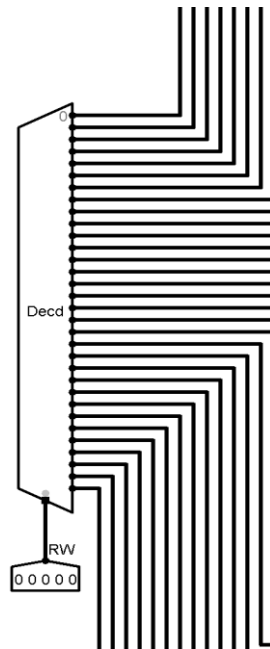


Figure 2.2.4

- **Register Write Enable Mechanism:**

- To guarantee that only the intended register is updated during a write operation, a dedicated control circuit is implemented using a logical AND gate.
- The first input to the AND gate is the **global Write Enable signal** (Write), which is asserted by the control unit when a write-back is required.
- The second input is the **individual register select signal** (InputR1), produced by the **Register Decoder** to activate one specific register based on the destination address.

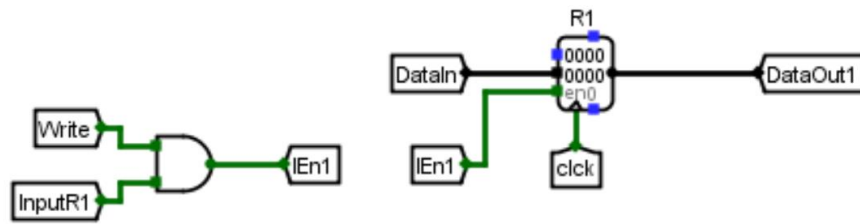


Figure 2.2.5

The output of the AND gate (IEn1) enables the corresponding register for writing. A register will capture the input data only if **both**:

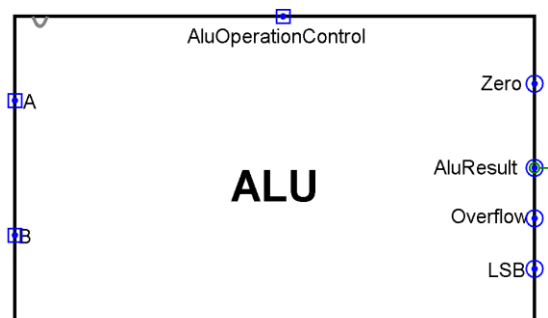
1. A write operation is requested (Write = 1).
2. The register matches the destination register (InputR1 = 1).



## 2.2 Arithmetic and Logic Unit (ALU):-

### 1. Introduction

The Arithmetic and Logic Unit (ALU) is a central element of the 32-bit risc processor. It is responsible for performing all arithmetic, logical, shift, rotation, and comparison operations that are necessary for the successful execution of the instruction set. Operating at the core of the processor's execution stage, the ALU directly influences both the correctness and the performance of instruction execution



### 2. ALU Functional Overview

The ALU processes two 32-bit input operands and produces a 32-bit result according to a specified operation code. It is capable of handling a wide variety of tasks including:

- Integer arithmetic operations (addition, subtraction, multiplication)
- Bitwise logic operations (AND, OR, XOR, NOR)
- Bit shifting and rotation operations (logical shift, arithmetic shift, rotate)
- Comparison operations (less-than, equality)

### 3. ALU Supported Operations

The ALU supports operations corresponding to both R-type and I-type instructions as defined in the project ISA:

#### 3.1 Arithmetic Operations

- ADD / ADDI: Sum of two operands.
- SUB: Subtraction of the second operand from the first.
- MUL: Multiplication of two operands .

### **3.2 Logical Operations**

- AND / ANDI: Bitwise AND operation.
- OR / ORI: Bitwise OR operation.
- XOR / XORI: Bitwise Exclusive OR.
- NOR / NORI: Bitwise NOR (NOT of OR).

### **3.3 Shift and Rotate Operations**

- SLL / SLLI: Logical shift left.
- SRL / SRLI: Logical shift right (zero-fill).
- SRA / SRAI: Arithmetic shift right (sign-extended).
- ROR / RORI: Bit rotation right.

### **3.4 Comparison Operations**

- SLT / SLTI: Set on less than (signed comparison).
- SLTU / SLTIU: Set on less than (unsigned comparison).
- SEQ / SEQI: Set if equal.

### **3.5 Additional Operation**

- SSET: Set value which is concatenation of the (lower 16 bit) first operand and (lower 16 bit) second operand

These operations are performed based on the ALU Control signals determined by instruction decoding logic.

## 4. ALU Inputs and Outputs

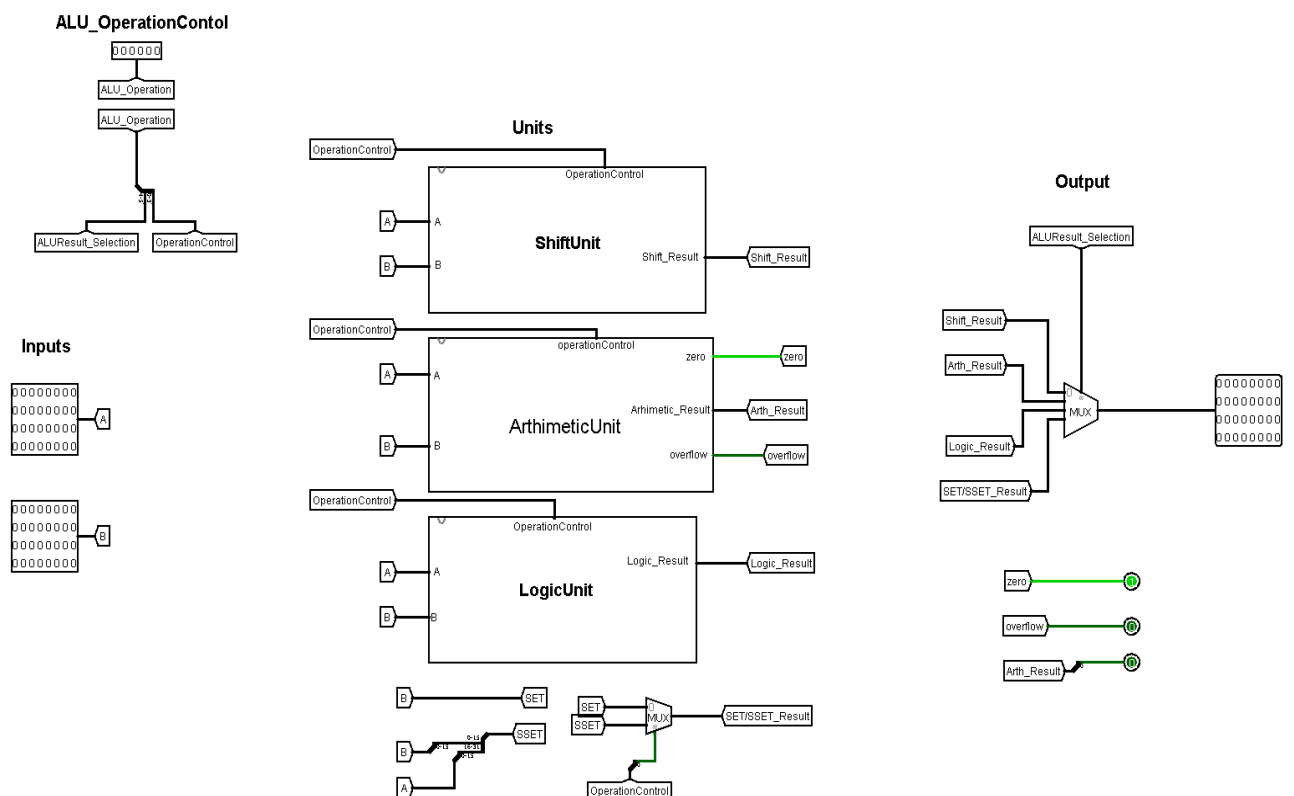
### 4.1 Inputs

- Operand A (32 bits): sourced from register RS1 / Rd.
- Operand B (32 bits): Sourced either from register RS2 or an immediate value.
- ALU Control Signal (ALUOp): Selects the operation to be performed.

### 4.2 Outputs

- Result (32 bits): The final computed value.
- Zero Flag (1 bit): Asserted if the result equals zero, used primarily in branch decisions.
- (Optional) Overflow Flag: Indicating arithmetic overflow in signed operations.

## 5. ALU Hardware Design



The hardware architecture of the ALU is composed of the following primary components:

## 5.1 Arithmetic Unit

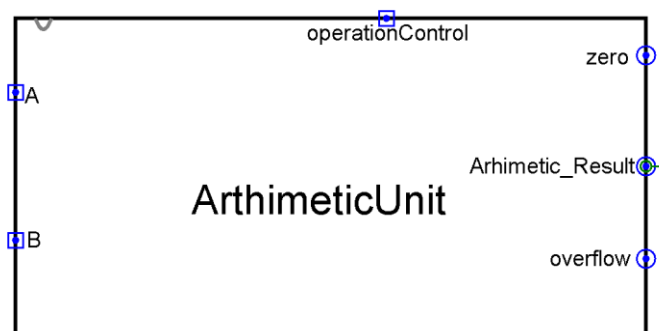
### 5.1.1. Introduction

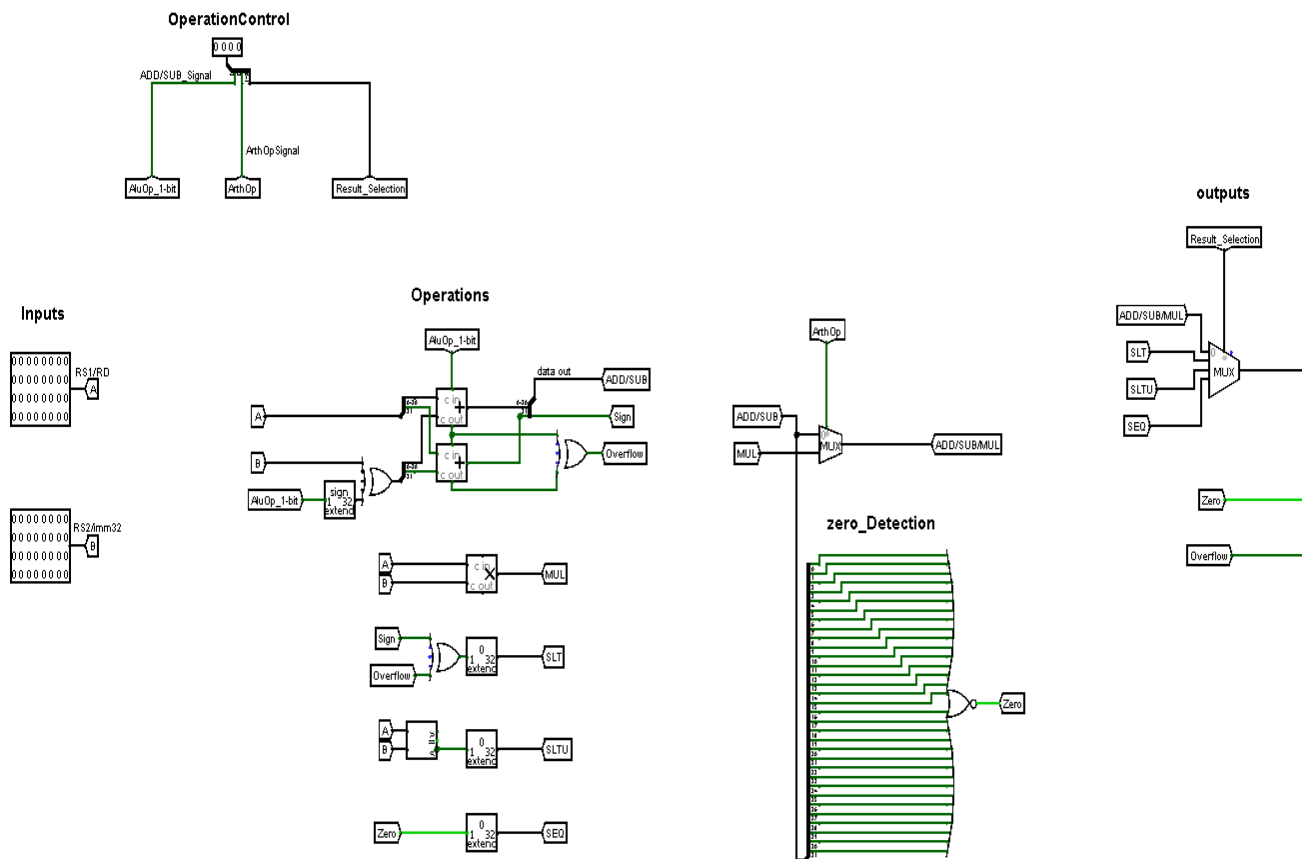
The arithmetic unit must be carefully designed to complete operations within a single clock cycle in a single-cycle processor, and within the execution stage of a pipelined processor.

It is dedicated to executing integer arithmetic operations, such as addition, subtraction, and

Multiplication.

The correct and efficient performance of the Arithmetic Unit is critical for achieving the processor's required functionality and performance, especially because many instructions depend on arithmetic computations (e.g., ADD, SUB, MUL, ADDI).





### 5.1.2. Functional Overview

The Arithmetic Unit performs three main types of operations:

- **Addition** of two 32-bit operands.
- **Subtraction** of two 32-bit operands.
- **Multiplication** of two 32-bit operands (only the lower 32 bits are retained).

Each of these operations is selected based on the ALU Control Signals generated from instruction decoding.

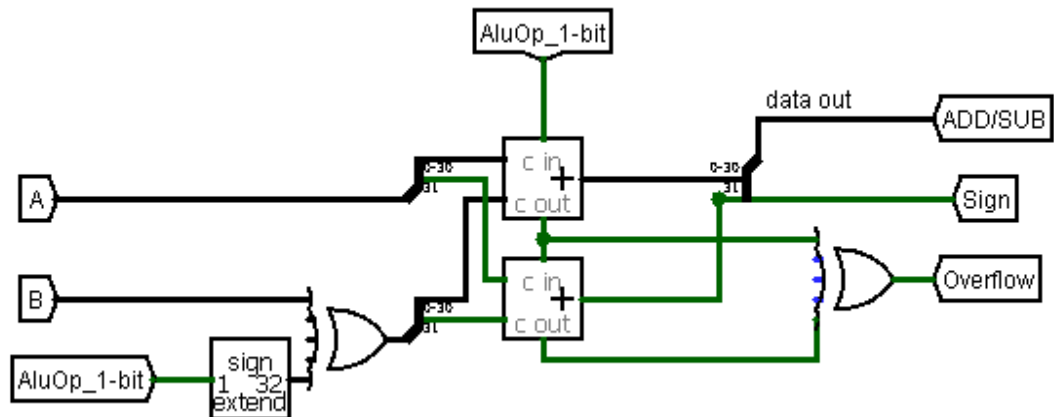
### 5.1.3. Hardware Components

The Arithmetic Unit is composed of the following primary building blocks:

#### 5.1.3.1 32-bit Adder/Subtractor

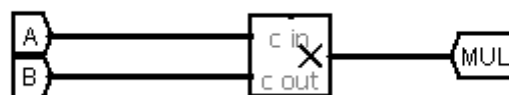
- Addition is performed using a 32-bit adder
- Subtraction is performed using the same adder by taking advantage of two's complement arithmetic:

- To subtract B from A (i.e.,  $A - B$ ), the unit inverts all bits of B and adds one (two's complement) before performing addition.
- A signal received from the Alu Control Unit to select whether to pass B directly (for addition) or its complement (for subtraction) .
- **Carry-in** is set to 1 for subtraction (to complete two's complement) and 0 for addition.



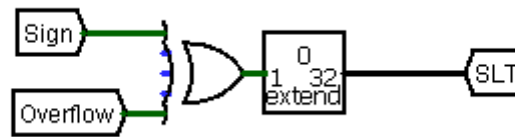
### 5.1.3.2 32-bit Multiplier

- The multiplier performs a signed integer multiplication of two 32-bit operands.
- Only the lower 32 bits of the 64-bit multiplication result are stored and used, consistent with the instruction set's definition (MUL operation).



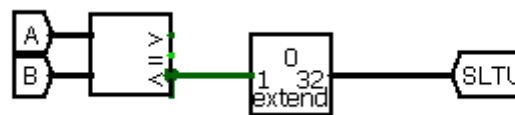
### 5.1.3.3 SetLessThan(SLT)

- The unit sets the output to 1 if operand A is less than operand B; otherwise, it sets the output to 0.
- The evaluation relies on the sign and overflow flags, which are determined from the result of the subtraction operation.
- The condition for  $(A < B)$  is identified when the sign and overflow flags are not equal, a comparison implemented using an XOR gate



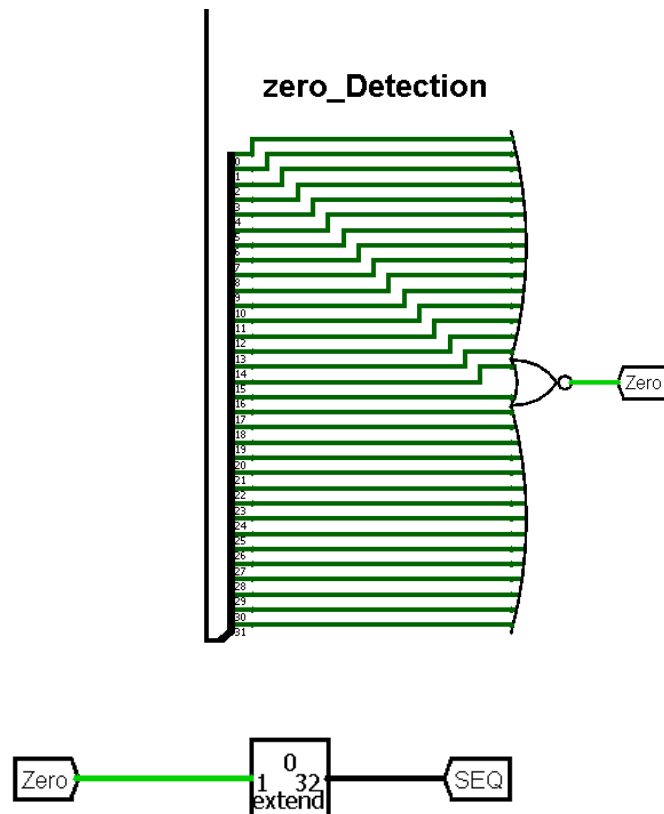
#### 5.1.3.4 SetLessThanUnsigned(SLTU)

- The unit sets the output to 1 if the unsigned value of Operand A is less than the unsigned value of Operand B, utilizing a 32-bit comparator for evaluation.



#### 5.1.3.5 SEQ

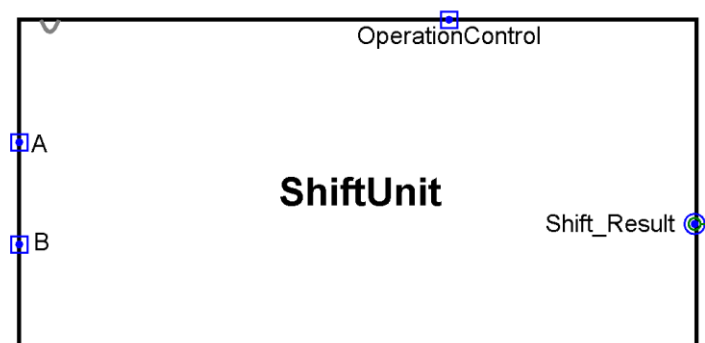
- The unit sets the output to 1 if the value of Operand A is equal to the value of Operand B, determined by checking the Zero flag generated from the subtraction operation.
- Zero detection is performed by evaluating the output of a NOR gate, whose inputs are the result bits of the subtraction operation.



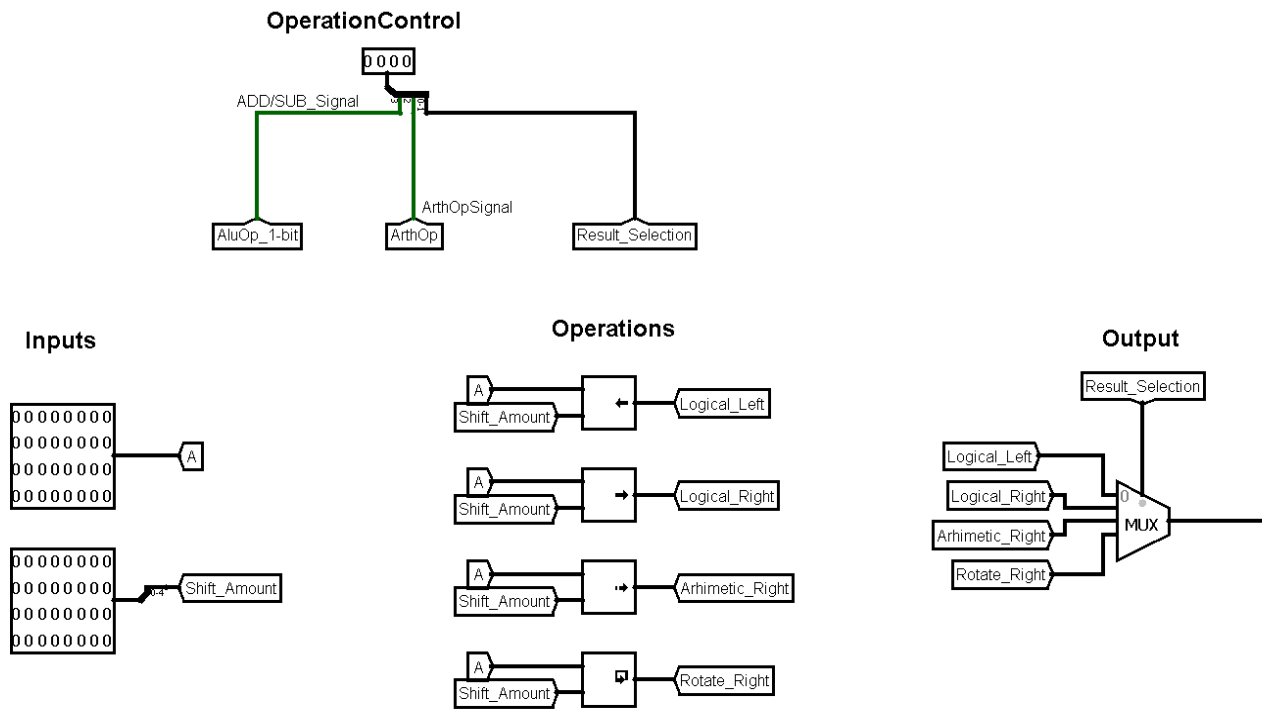
## 5.2 shift Unit

### 5.2.1 introduction

The Shift Unit is a dedicated hardware block within the Arithmetic and Logic Unit (ALU) responsible for performing all bit-shifting and rotation operations required by the processor's instruction set. Shifting is a fundamental operation in computer architectures, widely used in arithmetic computations, logical operations, and efficient data manipulation.







### 5.2.2. Functional Overview

The shift unit supports the following operations:

- **Logical Shift Left (SLL, SLLI):** Shifts bits to the left, filling vacated bits with zeros.
- **Logical Shift Right (SRL, SRLI):** Shifts bits to the right, filling vacated bits with zeros.
- **Arithmetic Shift Right (SRA, SRAI):** Shifts bits to the right while replicating the sign bit to preserve the number's sign (for signed integers).
- **Rotate Right (ROR, RORI):** Rotates bits to the right, with bits shifted out on the right re-entering on the left.

Each of these operations uses a dedicated or shared hardware block and is selected through control signals generated by the ALU control logic.

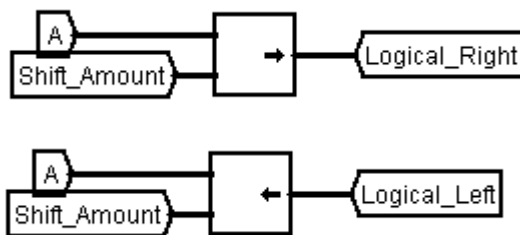
### 5.2.3. Hardware Components

The Shift Unit hardware typically consists of the following subcomponents:

#### 5.2.3.1 Logical Shifting

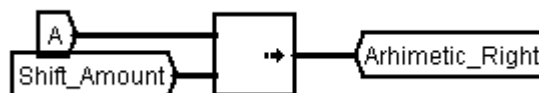
For Logical Shift Left (SLL) and Logical Shift Right (SRL):

- The barrel shifter simply moves bits left or right.
- Vacant bits are filled with zeros.



#### 5.2.3.2 Arithmetic Shifting

- For Arithmetic Shift Right (SRA):
  - Similar to logical right shift but instead of inserting zeros into the most significant bits (MSBs), the sign bit (the original MSB) is replicated.
  - This behavior maintains the correct sign in signed integer operations.
- Implemented using conditional logic that checks the original sign bit and propagates it during shifting.

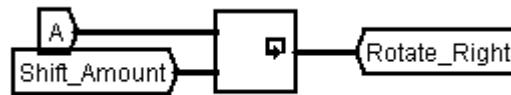


#### 5.2.3.3 Rotate Right

For Rotate Right (ROR):

- Bits shifted out from the least significant positions (LSBs) are wrapped around and reinserted at the most significant positions (MSBs).

- This is typically achieved by concatenating two shifted versions of the input and selecting the appropriate bits.

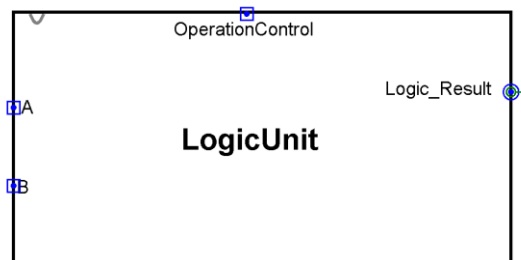


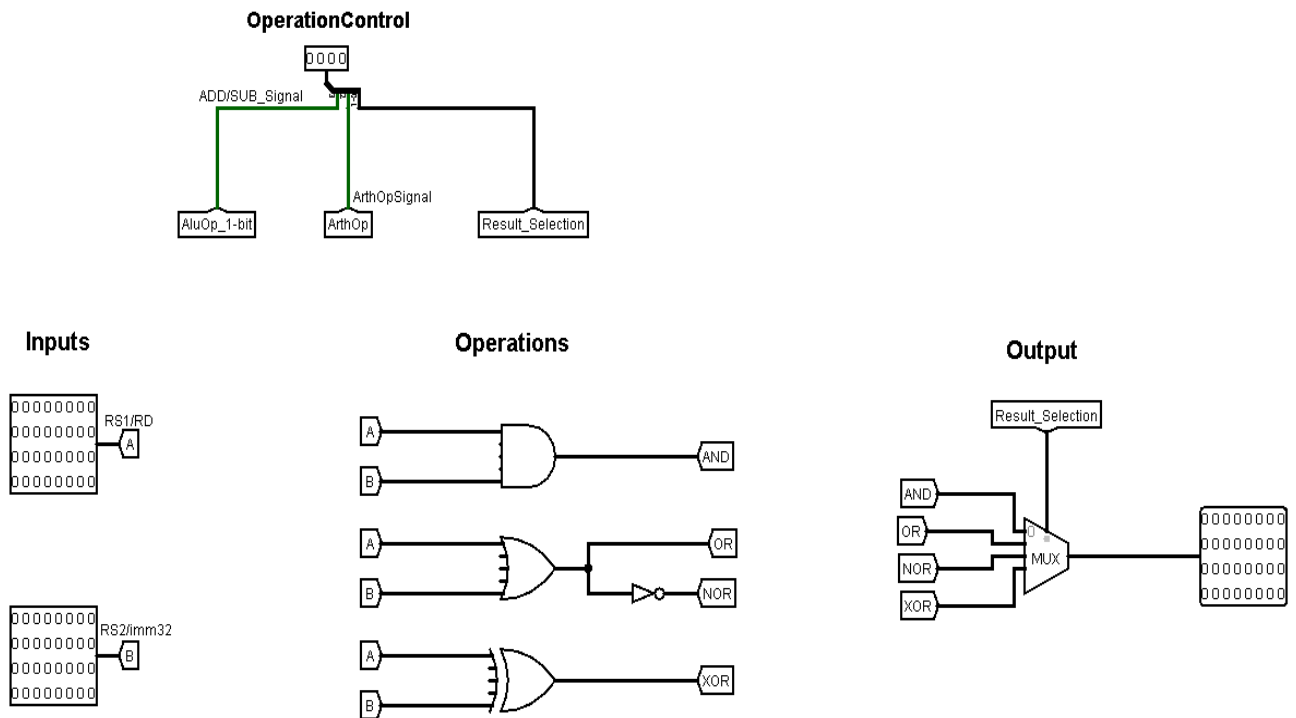
## 5.3 logical Unit

### 5.3.1. Introduction

The Logic Unit is an integral subcomponent of the Arithmetic and Logic Unit (ALU). Its primary role is to perform bitwise logical operations on two 32-bit operands. Logical operations are fundamental to many computational tasks, including condition evaluation, masking, setting, or clearing specific bits, and manipulating control flow.

The Logic Unit is designed for single-cycle operation, ensuring that all logical functions complete within one clock cycle during both single-cycle and pipelined execution modes.





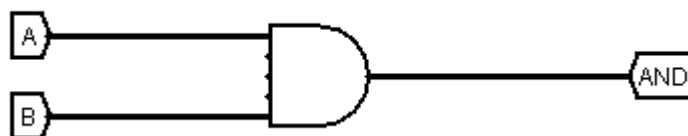
### 5.3.2. Functional Overview

The Logic Unit supports the following bitwise operations:

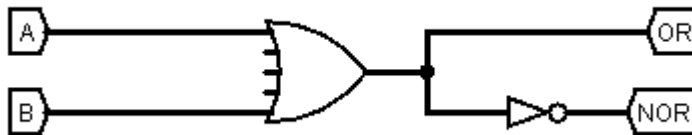
- **AND (AND, ANDI):** Performs a logical AND between each corresponding bit of two operands.
- **OR (OR, ORI):** Performs a logical OR between each corresponding bit.
- **XOR (XOR, XORI):** Performs a logical exclusive OR between corresponding bits.
- **NOR (NOR, NORI):** Performs a logical NOR, equivalent to NOT(Operand A OR Operand B).

### 5.3.3. Hardware Components

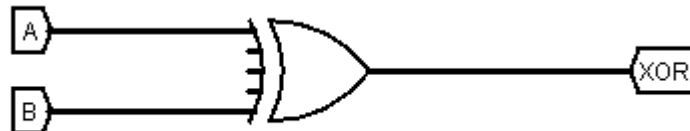
- **5.3.3.1 32-bit AND Gate :** Performs a bitwise AND across all corresponding bits of the two operands.



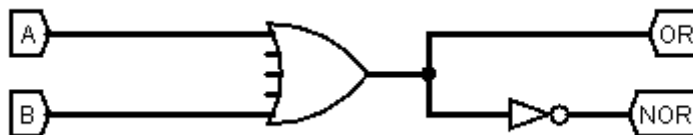
- **5.3.3.2 32-bit OR Gate** : Performs a bitwise OR operation.



- **5.3.3.3 32-bit XOR Gate** : Executes bitwise exclusive OR



- **5.3.3.4 32-bit NOR Gate** : Executes bitwise NOR by first performing an OR operation and then inverting each bit.



Each array consists of 32 individual 2-input gates, operating in parallel for maximum speed.

## 5.4 Additional Instructions(SET/SSET)

### 5.4.1 introduction

The additional instructions are integral subcomponents of the Arithmetic and Logic Unit (ALU). Its primary role is to set a specific value.

### 5.4.2. Functional Overview

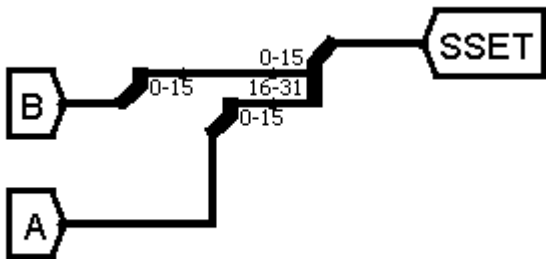
- SetEqual(SET): sets a 32-bit imm which is sign extended of imm16
- (SSET): sets a 32-bit imm which is concatenated of the first 16-bit of the first operand and the first 16-bit of the second operand

5.4.3 Hardware Components

- 5.4.3.1 SET :



- 5.4.3.2 SSET :



5.5 ALU Control Logic

The control unit decodes the opcode and function field (for R-type) to generate a 6-bit ALU control signal. This signal selects one of the arithmetic, logic, shift/rotate, or comparison outputs via a multiplexer.

5.6 Alu Operations Table

the following table represent the operation executed corresponding to AluOpCtrlSignals:

AluOpCtrl(6-bit)				instruction
Alu-Result-Slc(2 bit)	Operation Control			
	AluOp(1 bit)	ArthOp(1 bit)	Result-Slc(2 bit)	
1	0	0	0	ADD
1	1	0	0	SUB
1	X	1	0	MUL
1	1	X	11	SEQ
1	1	X	1	SLT
1	X	X	10	SLTU
10	X	X	0	AND
10	X	X	1	OR
10	X	X	11	XOR
10	X	X	10	NOR
0	X	X	0	SLL
0	X	X	1	SRL
0	X	X	10	SRA
0	X	X	11	ROR
11	X	X	X0	SET
11	X	X	X1	SSET

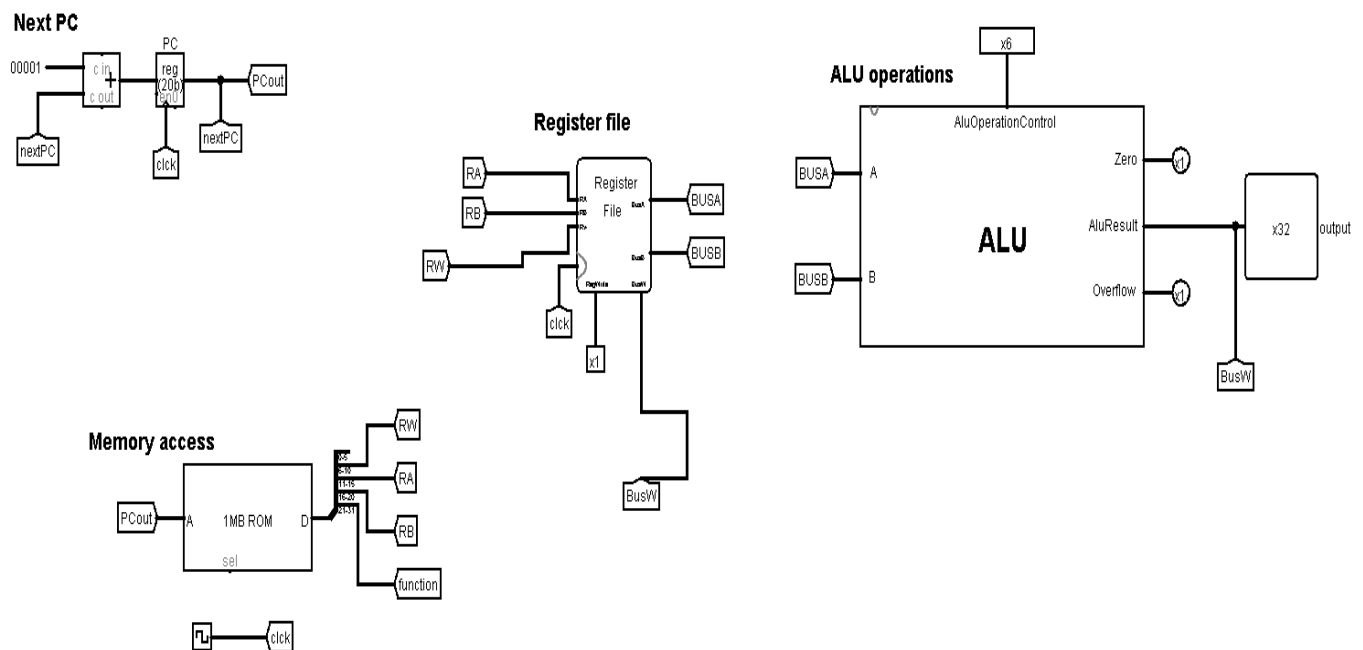
### 3.1 R-Type Data Path

This section describes the structure and components of the hardware design for the single-cycle processor that we are implementing in this project. Each part of the processor plays a crucial role in executing instructions, which include data manipulation and memory operations.

$F^{11}$	$S_2^5$	$S_1^5$	$d^5$	$OP^6$
----------	---------	---------	-------	--------

The main components of the R-type hardware design are the Program Counter (PC), Register File, ALU (Arithmetic Logic Unit), and Memory Access unit(Instruction Memory). These elements work in unison to fetch, decode, execute, and write back results of instructions in one clock cycle.

#### *R\_Type DataPath*



#### 3.1.1 Program Counter (PC)

The Program Counter (PC) shown in *figure 3.1.1* plays a role in managing the flow of execution. The PC stores the address of the next instruction to be fetched from memory. For R-type instructions, which are primarily register-to-register operations, the PC is updated sequentially by adding 1 to its current value, ensuring that the processor fetches the next instruction in memory during the next cycle.



In the single-cycle processor design, the PC is incremented at the beginning of each cycle, pointing to the subsequent instruction unless a branch or jump instruction is encountered.

### Functionality:

**Instruction Fetch:** The PC points to the current instruction in memory. In the single-cycle processor, the instruction is fetched during the first stage of execution.

**Update Mechanism:** The PC is updated after every instruction cycle. The increment is usually by 1, with 20 data bits. If the instruction is a branch or jump, the PC is updated with the target address based on the specific control signals in another way explained in Itype, SBtype.

**Reset:** At the start of the program execution, the PC is initialized to a starting address zero address).

### PC Structure:

- PC Register: Stores the current address of the instruction to be fetched.
- Incrementer: Automatically increments the PC by 1 on each cycle to fetch the next sequential instruction.  $\text{NextPC} = \text{PC} + 1$
- 

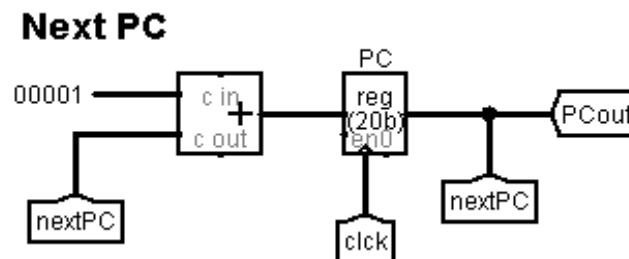


Figure 3.1.1

### 3.1.2 Register File

The register file in an R-type shown in *figure 3.1.2* is a critical component that stores and provides fast access to the operands needed for computation. It consists of a set of registers, typically 32 in number, each capable of holding a 32-bit value. The register file is integral to the execution of R-type instructions, which typically involve arithmetic and logical operations between registers.

### Functionality

The operation of the register file in an R-type architecture can be broken down into the following steps:

1. Reading Operands: The two source registers (s2 and s1) are read from the register file in parallel. The values from these registers are passed to the ALU, which performs the operation specified by the instruction (e.g., addition, subtraction, AND, OR).
2. ALU Execution: The ALU performs the arithmetic or logical operation based on the values fetched from the register file and the operation code (opcode = 0) provided by the instruction. The result is generated and sent to the write port of the register file.
3. Writing the Result: After the ALU completes its operation, the result is written back into the destination register (d), which is part of the instruction. This is typically done at the end of the instruction cycle, ensuring the result of the operation is stored for use in subsequent instructions.

### RegisterFile Structure

The register file supports at least two read ports, allowing the processor to fetch two operands simultaneously. For an R-type instruction, these are usually the source registers (denoted as s2 and s1), which hold the values needed for operations such as addition, subtraction, or logical AND/OR.

The register file also includes a write port, where the result of the operation (computed by the ALU) is stored. In the case of R-type instructions, this is typically the destination register (d), which is specified in the instruction. Only one register is written back during a cycle, as R-type instructions use a single destination register.

1. rs (Read Register 1): Specifies the first register to be read.
2. rt (Read Register 2): Specifies the second register to be read.
3. rd (Write Register): Specifies the register where the ALU result will be stored.
4. RegWrite: A control signal that enables writing to the register file.

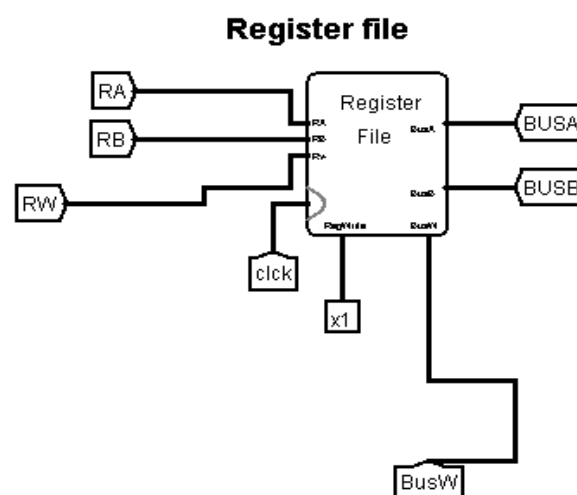


Figure 3.1.2

### 3.1.3 ALU

The Arithmetic Logic Unit (ALU) shown in *figure 3.1.3* is a fundamental building block in the execution of R-type instructions within a single-cycle processor architecture. It is the core computational element responsible for executing a wide range of arithmetic and logical operations that are defined by the processor's instruction set. In the context of R-type instructions, which involve operations between two register operands, the ALU serves as the execution engine that processes the required computations in a single clock cycle.

The ALU performs essential arithmetic operations such as addition and subtraction, which are crucial for mathematical computations, address calculations, and data manipulation tasks. Additionally, it carries out various logical operations like bitwise AND, bitwise OR, and bitwise NOR, which are important for tasks such as masking bits, setting conditions, and implementing logical decision-making at the hardware level.

Moreover, the ALU supports comparison operations like Set on Less Than (SLT), which evaluates whether one operand is smaller than another. This result can then be used in subsequent conditional operations or control flow decisions within a program.

#### **Structure:**

1. **Input Selection:** The ALU receives two inputs from the outputs of the register file (Read Data 1 and Read Data 2).
2. **Operation Selection:** Based on the decoded function (funct) field of the R-type instruction, the ALU control unit selects the appropriate operation (e.g., ADD, SUB, AND, OR, SLT).
3. **Computation:** The ALU performs the required arithmetic or logical computation based on the selected operation.
4. **Result Output:** The computed result is forwarded to be written back into the destination register (d) within the register file.
5. **Zero Flag:** Additionally, the ALU typically generates a Zero output signal, which is set if the result of the operation is zero. This flag can be used in other types of instructions like branches but is also generated during R-type operations.

#### **Inputs and Outputs of the ALU**

- **Inputs:**
  - Operand A (first source register)
  - Operand B (second source register)
  - ALUControl (from the control unit)
- **Outputs:**

- ALUResult (32-bit result of the operation)
- Zero flag (1 if the ALUResult is zero, 0 otherwise)

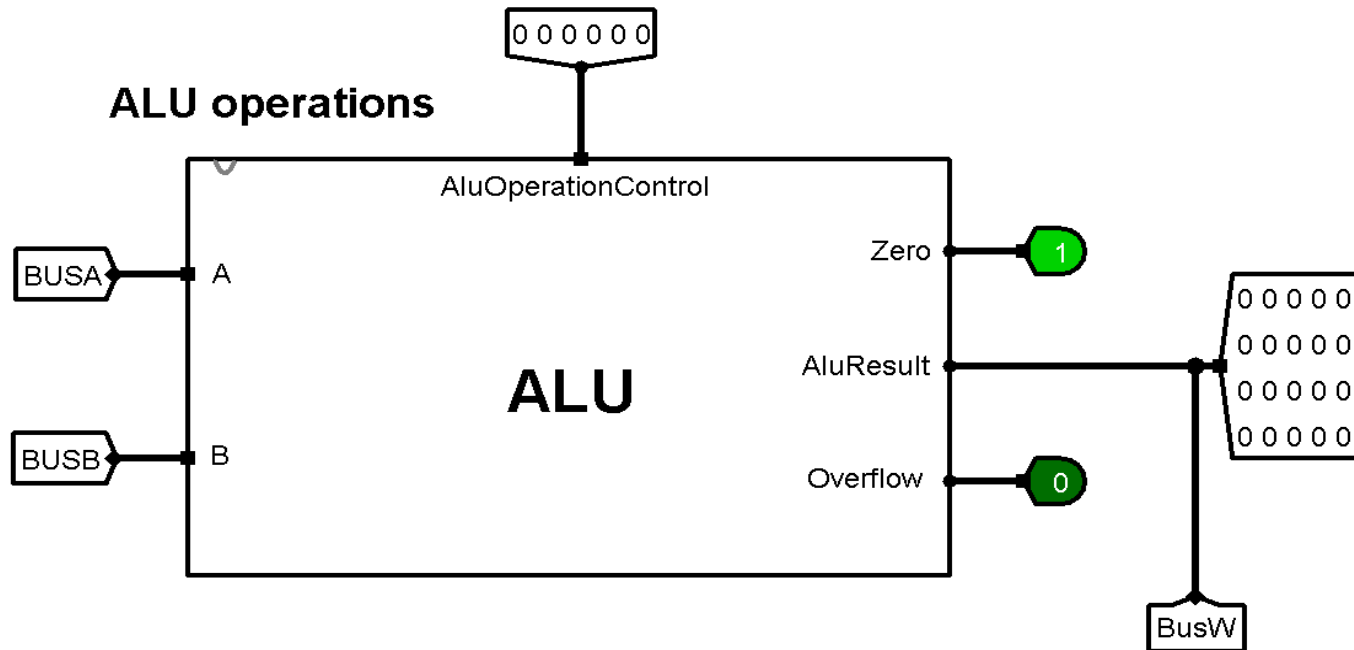


figure 3.1.3

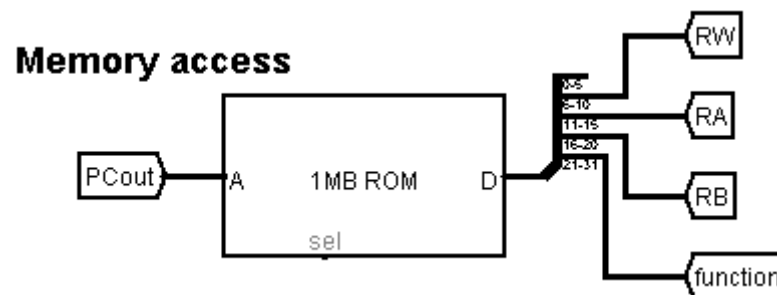
### 3.1.4 Memory Access (Instruction Memory)

The Instruction Memory shown in *Figure 3.1.4* is a crucial component designed to store and provide the program instructions necessary for execution. Structurally, the instruction memory is organized as a read-only memory (ROM) unit that is addressed using the current value of the Program Counter (PC). Upon receiving the address from the PC, the memory deals with 20 address bit width and 32 data bit width. This instruction contains several fields, including the opcode, source registers, destination register, shift amount, and function code.

Functionally, the instruction memory ensures that each instruction is fetched accurately and quickly, enabling the processor to decode and execute it within the same clock cycle. It is important to note that in a single-cycle processor. The instruction memory thus plays a foundational role in maintaining the processor's synchronous operation and ensuring a seamless flow from fetching to decoding and execution.

**Functionality:**

- Receives the address from the PC and outputs the corresponding instruction.
- Supports only read operations during program execution.
- Provides the instruction within a single clock cycle to maintain single-cycle operation timing.
- Critical for feeding instructions into the decode and execution stages without delay.



*Figure 3.1.4*

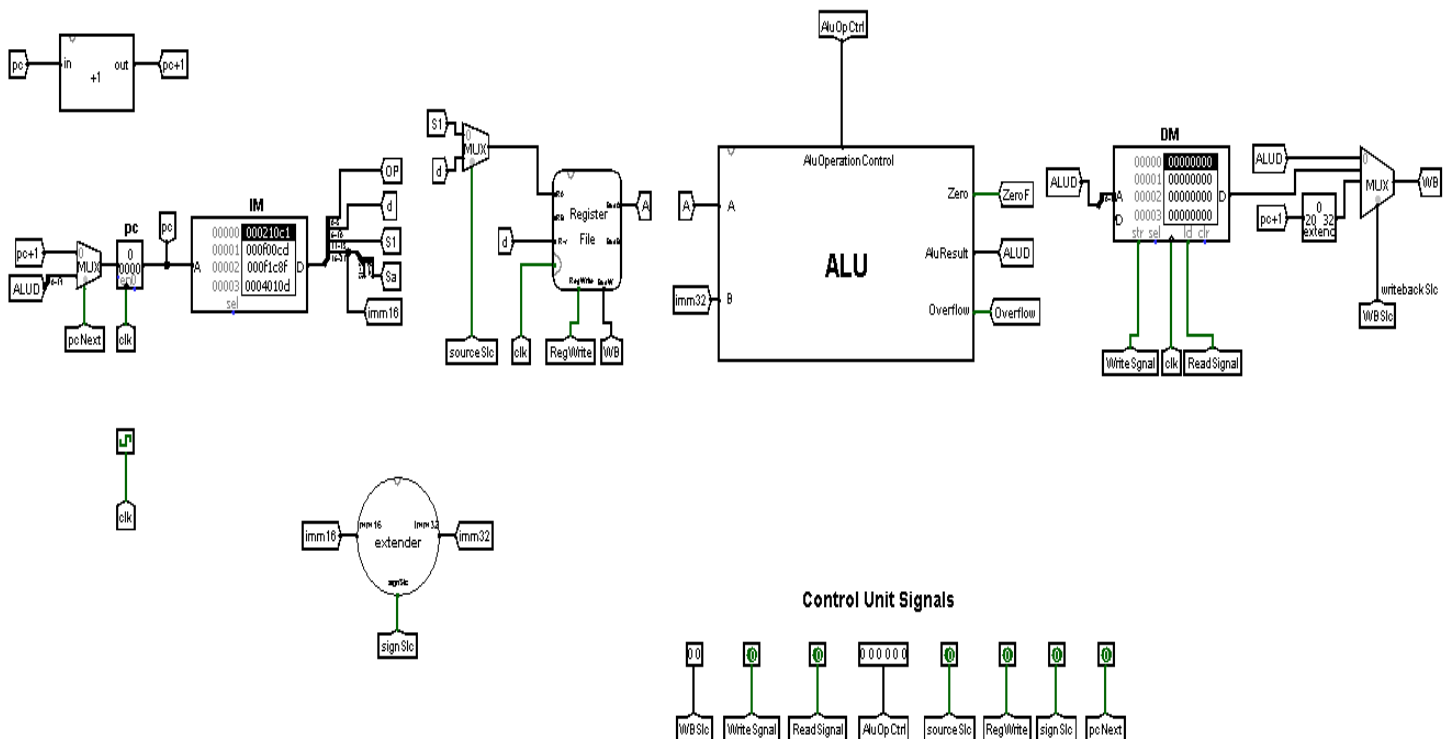
### 3.2 I-Type data path

### 3.2.1. Introduction

In a Risc processor, I-type (Immediate-type) instructions form a crucial part of the instruction set architecture. These instructions operate on a register operand and an immediate value (a constant encoded directly within the instruction). The I-type instruction format enables efficient execution of operations such as arithmetic with immediates, memory access (load) .

Examples of I-type instructions include ADDI, ANDI, ORI, LW and SLTI.

The I-type datapath must support reading register operands, extending immediate values, performing ALU operations, accessing memory (for load).



### 3.2.2. I-Type Instruction Format

The typical 32-bit format of an I-type instruction is:

<b>Imm 16</b>	<b>S<sub>1</sub><sup>5</sup></b>	<b>d<sup>5</sup></b>	<b>OP<sup>6</sup></b>
---------------	----------------------------------	----------------------	-----------------------

- Opcode (6 bits): Identifies the operation type (e.g., ADDI, LW, BEQ).

- s1 (5 bits): Source register number.
- d (5 bits): Destination register number (or source for SSET).
- Immediate (16 bits): Constant value (sign-extended or zero-extended to 32 bits).

### 3.2.3. Functional Flow of the I-Type Datapath

The execution of an I-type instruction involves the following steps:

- **Instruction Fetch:**

The instruction is fetched from instruction memory using the Program Counter (PC).

- **Instruction Decode and Register Read:**

The opcode, source register (**s1**), destination register (**d**), and immediate field are decoded.

The content of the **s1** and **d** registers is read from the register file.

- **Immediate Extension:**

The 16-bit immediate is extended to a 32-bit value:

- **Sign-extended** for operations like **ADDI**, **LW**.
- **Zero-extended** for logical operations like **ANDI**, **ORI**, **XORI**.

- **ALU Operation:**

The ALU performs the required computation between the value from **rs / rd** and the extended immediate:

Arithmetic (e.g., addition for **ADDI**, address calculation for **LW**).

Logical operations (e.g., **ANDI**, **ORI**).

- **Memory Access :**

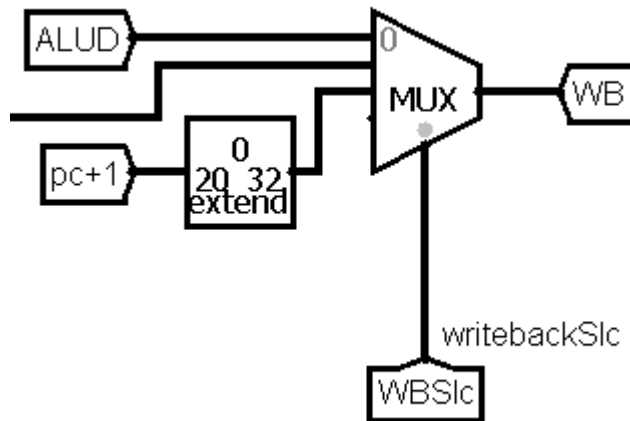
For **LW** (Load Word), the ALU output is used as the memory address to read data from data memory.

- **Write Back:**

For arithmetic/logical instructions, the ALU result is written back to the rd register.

For load instructions (LW), the memory data is written to the rd register.

For jump and link register (JALR), the pc+1 is written to the rd register.

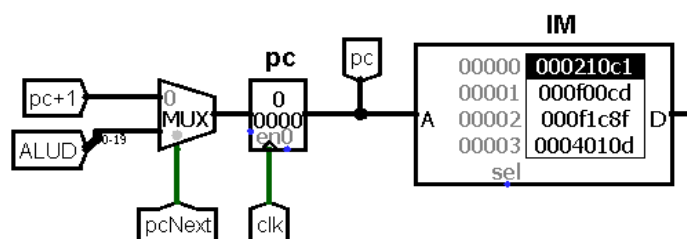
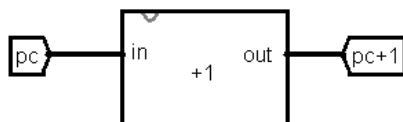


### 3.2.4. Hardware Design of the I-Type Datapath

The I-Type datapath requires the following hardware components:

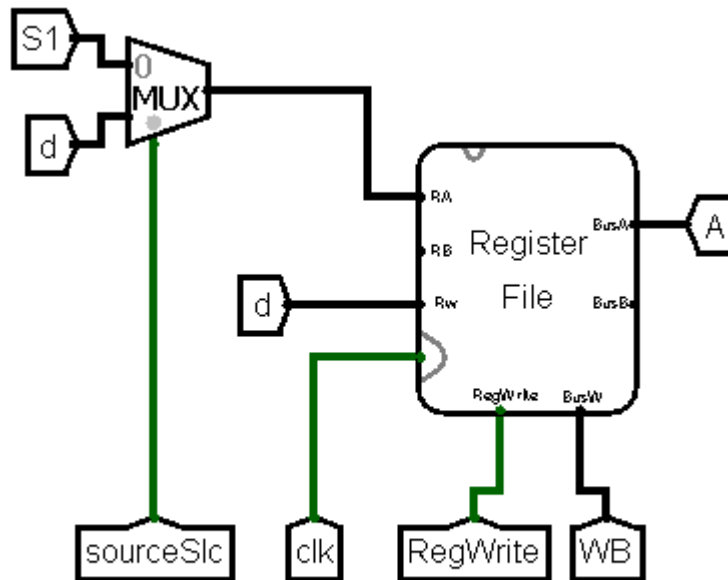
#### 3.2.4.1 Main Components

- **Program Counter (PC):** Holds the address of the current instruction.

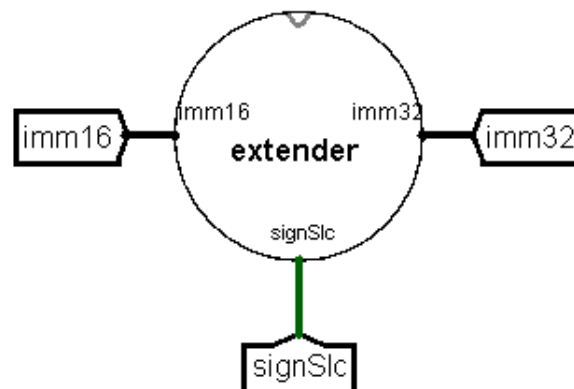


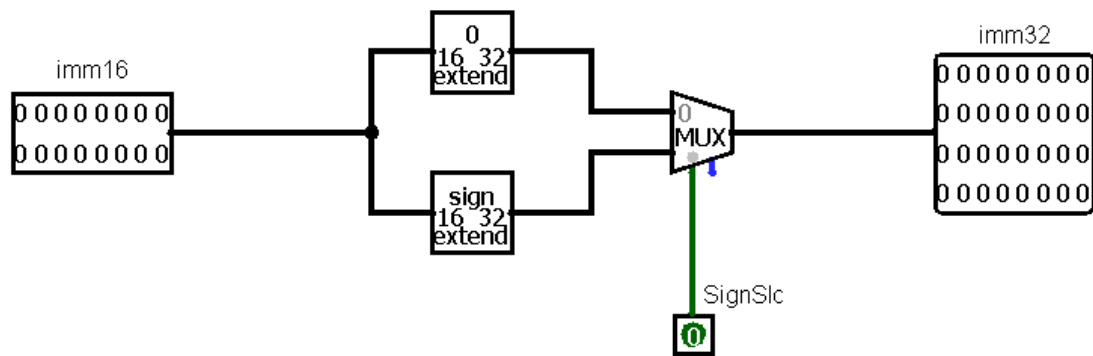


- **Instruction Memory:** Stores the instructions to be fetched.
- **Register File:**
  - Two read ports: for reading the **rs** (always used) and sometimes **rt** (for store/branch).
  - One write port: for writing back results to **rt**.



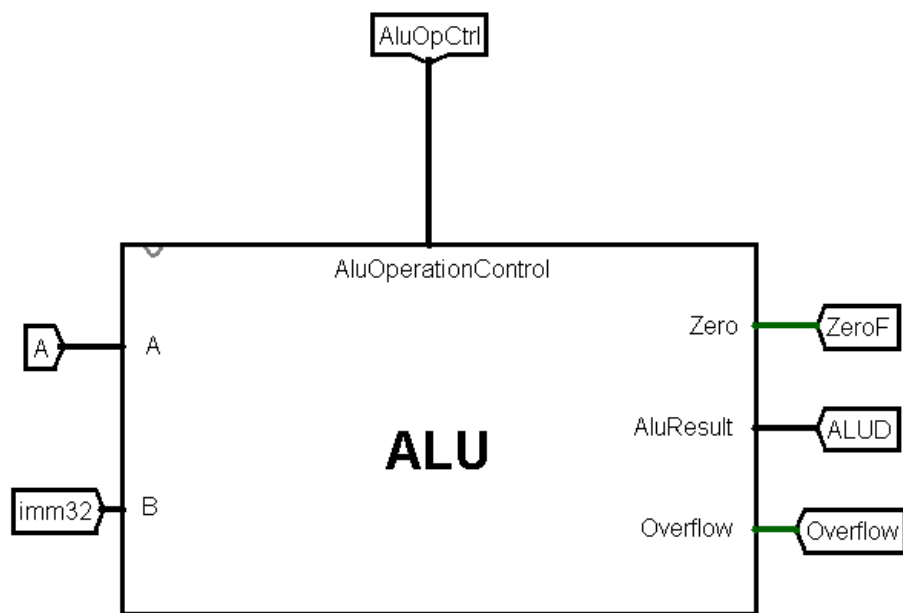
- **Immediate Extender:**
  - Extends the 16-bit immediate field to 32 bits.
  - Supports sign-extension and zero-extension based on instruction type.





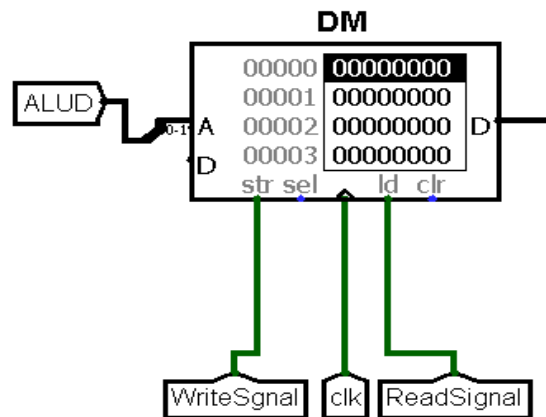
- **ALU:**

- Performs arithmetic or logical operations.
- Inputs: operand from `rs/rd` register and 32-bit extended immediate.

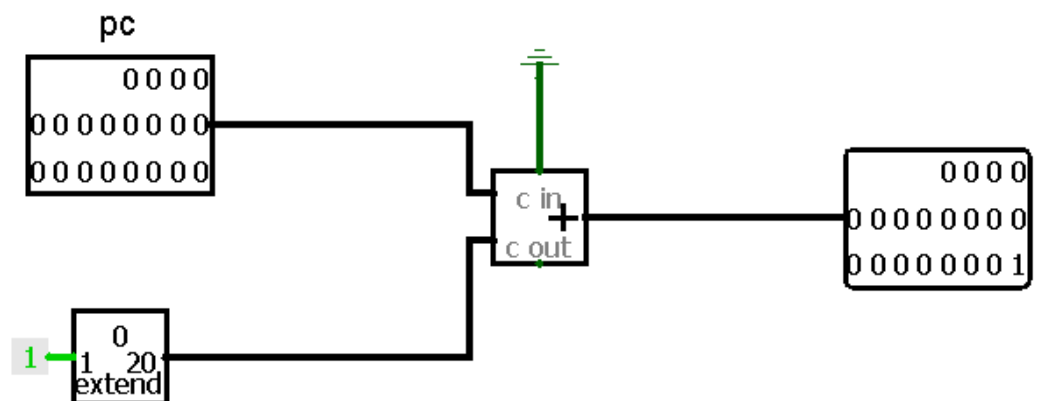
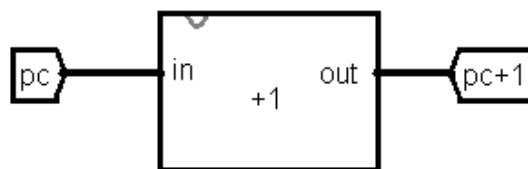


- **Data Memory:**

- Accessed during `LW` instruction.
- ALU result used as memory address.



- **Adder (PC + 1):**
  - Increments PC after instruction fetch for sequential execution.



### 3.2.4.1 Control Signals

- **pcNext:** Signal selects the next address( the pc+1 or the jump address ) that should be updated in pc
- **MemRead:** Signals whether a memory read (**LW**) should occur.

- **MemWrite:** Signals whether a memory write (**SW**) should occur.
- **WB Slc (MemtoReg):** Selects whether to write ALU result or memory data back to the register or pc+1.
- **RegWrite:** Enables writing to the register file.
- **ALUOpCtrl:** Specifies which ALU operation to perform.
- **SignSlc:** Signal whether the 16-bit immediate should be zero extended or sign extended
- **SourceSlc:** Signal whether the RS1 should be decoded or RD in Bus A

### 3.3 SB-Type Instruction Format and Execution in the Processor

The SB-type instruction format datapath shown in *Figure 3.3.1* in the designed single-cycle RISC processor plays a fundamental role in supporting store and conditional branch operations. It enables memory storage functionalities and dynamic changes in control flow based on register comparisons, forming a core part of the processor's capability to execute complex programs.

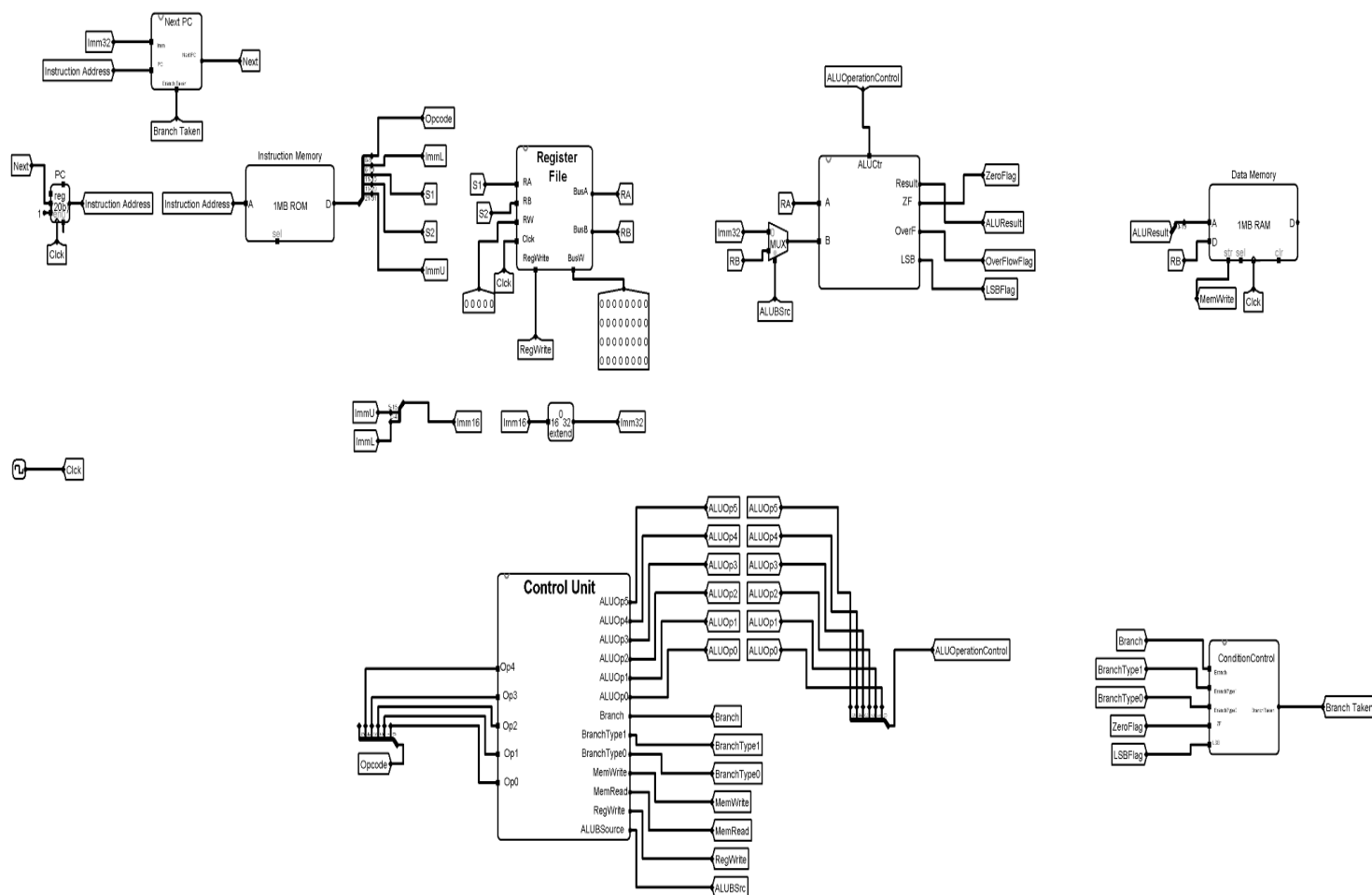


Figure 3.3.1

### 3.3.1. Instruction Encoding Structure

The SB-type instruction is encoded into a 32-bit word according to the following field distribution:

<b>ImmU<sup>11</sup></b>	<b>S<sub>2</sub><sup>5</sup></b>	<b>S<sub>1</sub><sup>5</sup></b>	<b>ImmL<sup>5</sup></b>	<b>OP<sup>6</sup></b>
--------------------------	----------------------------------	----------------------------------	-------------------------	-----------------------

Field Name	Bit Width	Description
Opcode	6 bits	Specifies the operation type.
ImmL	5 bits	Lower bits of the immediate value.
S1	5 bits	First source register address.
S2	5 bits	Second source register address.
ImmU	11 bits	Upper bits of the immediate value.

The immediate value required for the operation is formed by concatenating ImmU and ImmL, resulting in a 16-bit signed immediate, which is sign-extended to 32 bits before being used in memory or PC address calculations.

### 3.3.2. Functional Purpose

SB-type instructions serve two primary purposes:

- Store Operations:**  
 Instructions such as SW (Store Word) allow the processor to write data from a register into memory. The memory address is calculated as  $RS1 + \text{sign\_extend}(\{ImmU, ImmL\})$ , where RS1 holds the base address and the immediate provides an offset.
- Conditional Branch Operations:**  
 Instructions like BEQ, BNE, BLT, BGE, BLTU, and BGEU enable the processor to alter the flow of execution based on the comparison between two register values (RS1 and RS2). If the branch condition is satisfied, the PC is updated to a new address computed using the sign-extended immediate.

These operations enable control structures such as loops, if-else branches, and function calls within programs.

### 3.3.3. Immediate Value Construction

Immediate values in SB-type instructions are constructed as follows:

- **Concatenation:**

The fields ImmU (11 bits) and ImmL (5 bits) are concatenated to form a 16-bit value {ImmU, ImmL}.

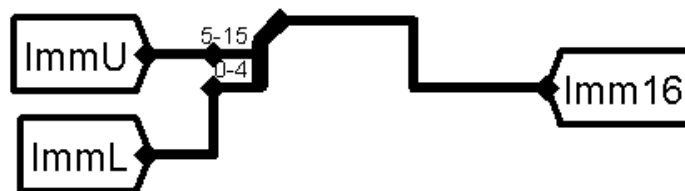


Figure 3.3.2

- **Sign Extension:**

The 16-bit immediate is sign-extended to 32 bits to correctly represent both positive and negative displacements.

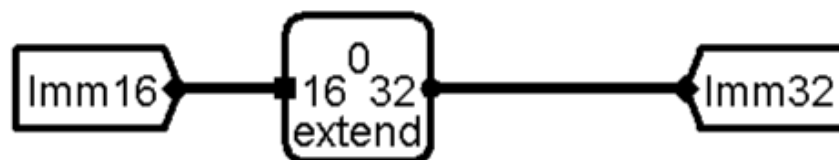


Figure 3.3.3

### 3.3.4. Execution Flow

The execution of SB-type instructions proceeds through the following major stages:

1. **Instruction Fetch:**

The instruction is fetched from Instruction Memory using the current value of the Program Counter (PC).

2. **Instruction Decode:**

The opcode and associated fields (S1, S2, ImmU, ImmL) are decoded.

The Control Unit identifies the instruction type and activates the corresponding control signals.

### 3. Register Operand Fetch:

The Register File is accessed to read the values of RS1 and RS2.

### 4. Immediate Handling:

The immediate fields ImmU and ImmL are combined and sign-extended to 32 bits.

### 5. Execution:

- For SW, the ALU computes the memory address ( $RS1 + Imm32$ ) and data from RS2 is stored into Data Memory.
- For branch instructions, the ALU compares RS1 and RS2, and the ConditionControl block determines whether the branch condition is satisfied.

### 6. Program Counter Update:

Based on the Branch Taken signal from ConditionControl:

- If the branch is taken, the NextPC circuit calculates  $PC + Imm32$ .
- Otherwise, the PC is simply incremented by 1.

### 7. Memory Access (for SW only):

Data is written to the computed memory address.

## 3.3.5. Sub-Components in SB-Type Execution

Several key subcomponents collaborate to implement SB-type instruction functionality:

### 3.3.5.1 Condition Control Unit

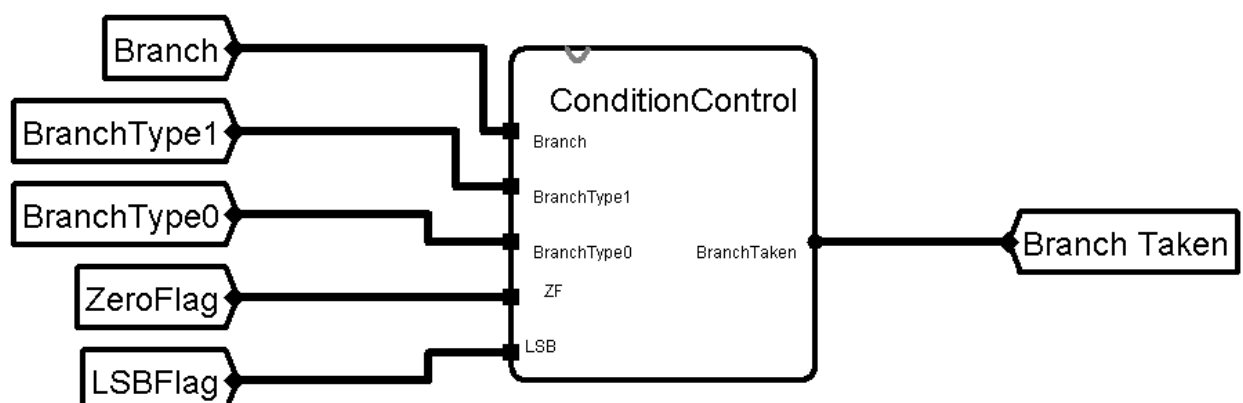


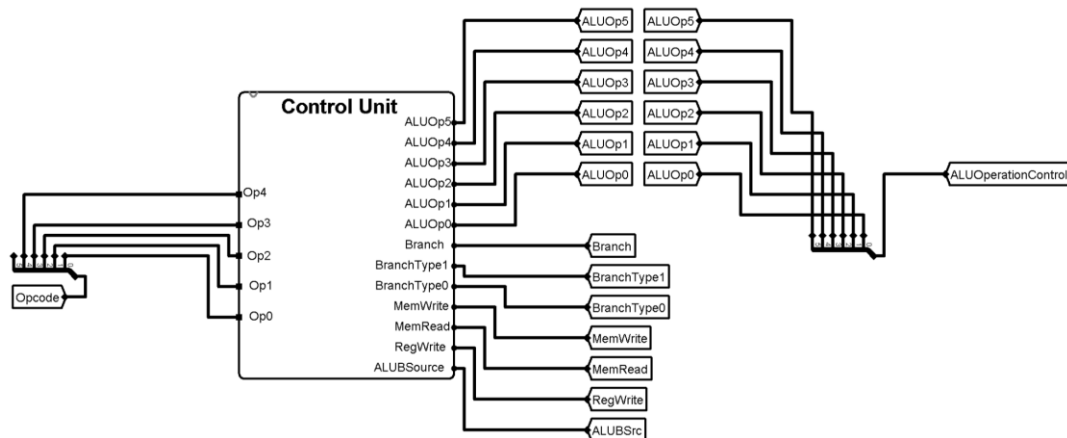
Figure 3.3.4



The Condition Control unit shown in *Figure 3.3.4* is responsible for evaluating branch conditions based on the result of ALU operations.

- **Inputs:**
  - Branch signal from the Control Unit.
  - BranchType bits (2 bits) specifying the type of branch (e.g., BEQ, BLT) from the Control Unit.
  - ZeroFlag and LSBFlag status flags from the ALU.
- **Functionality:**
  - Decides whether the branch condition is met.
  - Outputs Branch Taken, a single-bit signal that controls whether the PC will jump or continue sequentially.

### 3.3.5.2 Control Unit



*Figure 3.3.5*

The Control Unit shown in *Figure 3.3.5* decodes the instruction's opcode and orchestrates the operation of all major datapath components.

- **Inputs:**
  - 6-bit Opcode field.
- **Outputs:**
  - Control signals such as Branch, MemWrite, RegWrite, ALUOp, ALUBSrc, and branch types.
- **Functionality:**
  - Differentiates between store and branch instructions.
  - Activates appropriate signals for memory writing, ALU operation, and branching logic.

### 3.3.5.3 NextPC

The primary purpose of the Next PC block is to select between:

- Sequential instruction execution ( $PC + 1$ ), when no branch is taken.
- Branch target address ( $PC + \text{sign\_extend}(\text{Imm16})$ ), when a branch is taken based on a conditional evaluation.

Thus, the Next PC circuit shown in *Figure 3.3.6* allows the processor to either continue normal execution or to alter the flow of the program dynamically during branch instructions.

<i><b>Name</b></i>	<i><b>Width</b></i>	<i><b>Type</b></i>	<i><b>Description</b></i>
<i>Instruction Address (PC)</i>	<i>20 bits</i>	<i>Input</i>	<i>Current Program Counter value.</i>
<i>Imm32</i>	<i>32 bits</i>	<i>Input</i>	<i>Extended 32-bit immediate value formed from the instruction.</i>
<i>Branch Taken</i>	<i>1 bit</i>	<i>Input</i>	<i>Signal indicating whether a branch condition has been satisfied.</i>
<i>NextPC</i>	<i>20 bits</i>	<i>Output</i>	<i>Computed address of the next instruction to fetch.</i>

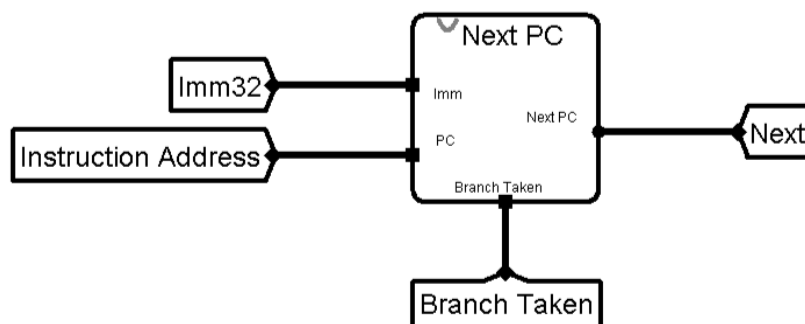


Figure 3.3.6

## Detailed Internal Design

The internal structure of the next PC subcircuit consists of the following elements:

- **Adder for target calculation**

Adds the current PC value to the sign-extended immediate value (Imm32) to compute the branch target address.

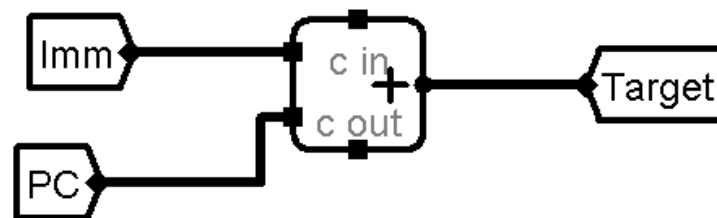


Figure 3.3.7

This operation is necessary for relative addressing used by branch instructions.

- **Adder for sequential execution**

Independently computes  $PC + 1$  for normal sequential instruction fetching.

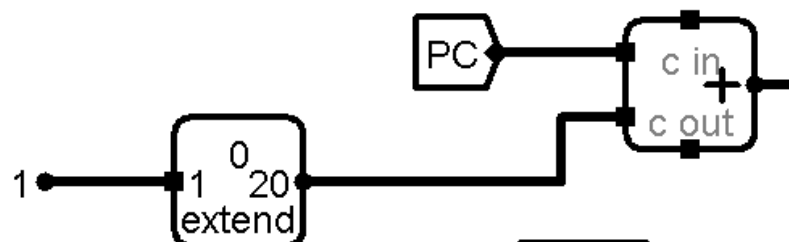


Figure 3.3.8

- **Multiplexer (MUX)**

Selects between the sequential PC ( $PC + 1$ ) and the branch target address ( $PC + Imm32$ ) based on the BranchTaken signal.

- If BranchTaken = 1, the MUX selects the branch target address.
- If BranchTaken = 0, the MUX selects the next sequential address.

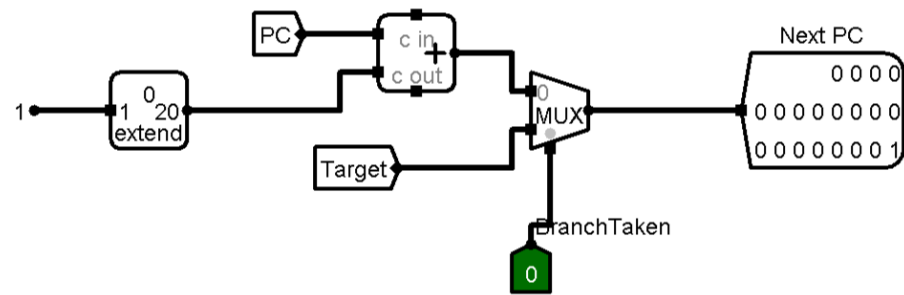


Figure 3.3.8

### 3.3.5.4 ALU (Arithmetic Logic Unit)

The ALU performs essential arithmetic and logic operations necessary for SB-type instructions.

- **Inputs:**
  - Two operand sources (from registers or immediate).
  - ALU operation control signals.
- **Outputs:**
  - Arithmetic or comparison results.
  - Status flags: ZeroFlag, LSBFlag, OverflowFlag.
- **Functionality:**
  - Computes effective memory addresses for store operations.
  - Compares operands to assist in branch condition evaluation.

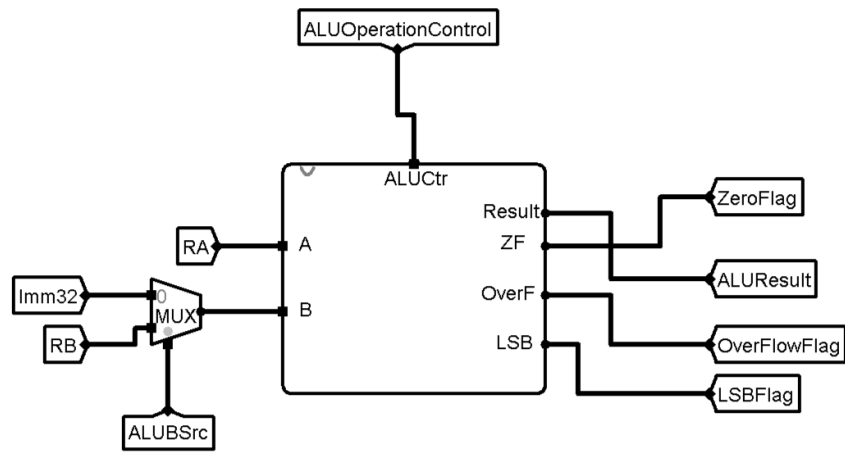


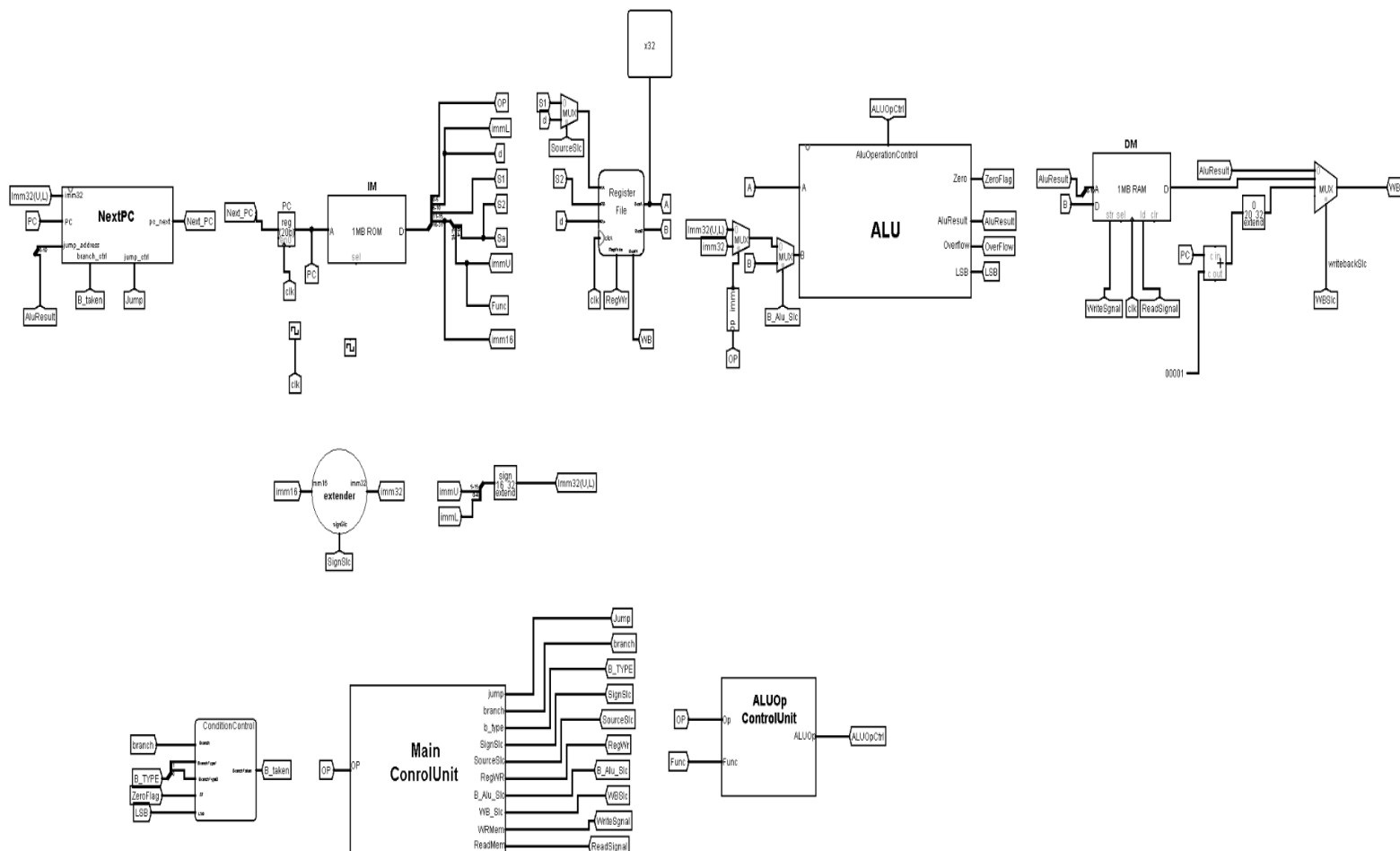
Figure 3.3.9

## 4. General Data Path

After Making all hardware paths it is time for the general datapath shown in *figure 4.1* which integrates all the essential components required for instruction execution in a single-cycle processor, providing a clear flow of data and control signals. It begins with the Program Counter (PC), which holds the address of the next instruction to be fetched from the Instruction Memory (IM). The fetched instruction is decoded to extract source registers, destination registers, immediate values, and function codes. These fields are used to select operands from the Register File and determine the operation to be performed by the Arithmetic Logic Unit (ALU). The ALU executes arithmetic or logical operations based on control signals generated by the Main Control Unit and the ALU Control Unit. For memory-related instructions, the Data Memory (DM) is accessed either to read from or write to memory, with the ALU result often providing the effective address.

Multiplexers (MUXes) throughout the datapath dynamically select between data sources based on control signals, ensuring correct data flow for different instruction types. Control units such as the Main Control Unit, ALUOp Control Unit, Condition Control Unit, and various smaller multiplexing and extending units work in harmony to drive the datapath effectively.

This fully integrated design ensures easy instruction fetching, decoding, execution, memory access, and write-back in a single clock cycle, achieving both efficiency and modular clarity.



## 4.1 Next PC

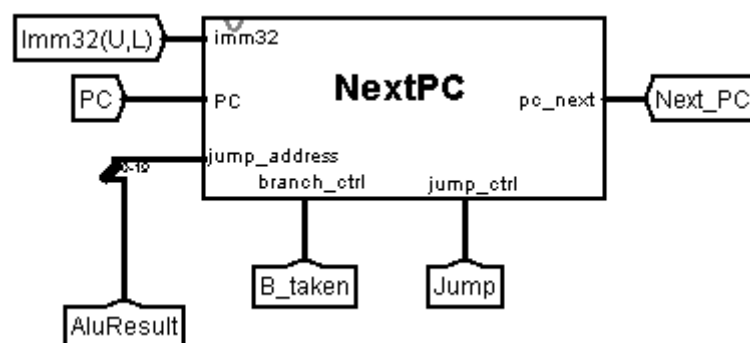
In an R-type instruction, the next Program Counter (PC) shown in *Figure 4.1* value was typically incremented by 1, which allows the processor to fetch the next sequential instruction. In contrast, an I-type instruction may modify the PC differently. For I-type instructions, the PC could either be updated by adding 1 (similar to R-type) or by utilizing an immediate value in conjunction with the base register, specifically in operations such as jump instruction. This means that the result is determined by adding the value in the source register (S1) with the sign-extended immediate field. The ALU performs this addition to determine the target address.

For SB-type (branch) instructions, the PC update behavior varies. If the branch is taken, the PC is updated by adding the sign-extended concatenated immediate value to the current PC, where the immediate value is split into upper and lower parts (often referred to as immU and immL). If the branch is not taken, the PC is simply incremented by 1 to proceed to the next instruction in sequence.

*SO, HOW TO COLLECT ALL THAT IN ONE CIRCUIT?*

To integrate all of this functionality into a single circuit, we need to check whether a jump is present. If a jump is detected, the PC is updated as follows:  $PC = RS1 + \text{sign-extended}(\text{Imm16})$ . For branch instructions, we must evaluate whether the branch is taken. If the branch is taken, the PC is updated by adding the sign-extended concatenation of the upper and lower immediate values:  $PC = PC + \text{sign-extended}(\{\text{immU}, \text{immL}\})$ . If the branch is not taken, the PC is simply incremented by 1.

In the NEXT PC SB-TYPE subcircuit, this logic has already been implemented to handle the second and third operations (branch taken or not, and jump). To combine all these operations, we use a multiplexer (MUX) connected to the circuit, which selects between the jump and branch logic. The MUX takes two control signals: one for branch operations and one for jump operations, allowing the appropriate PC update to be chosen based on the current instruction type and condition.



*Figure 4.1*

**Jump Detection:**

If a jump instruction is detected, the PC is updated as:

$$PC = RS1 + \text{sign-extended}(\text{Imm16}).$$

**Branch Condition Evaluation:**

If the instruction is a branch:

If the branch is taken:

$$PC = PC + \text{sign-extended}(\{\text{immU}, \text{immL}\}).$$

If the branch is not taken:

$$PC = PC + 1.$$

**SB-TYPE Subcircuit:**

The logic for evaluating whether the branch is taken or not (second and third operations) is already implemented in the NEXT PC SB-TYPE subcircuit.

**MUX Integration:**

A multiplexer (MUX) is used to integrate all the operations into the circuit.

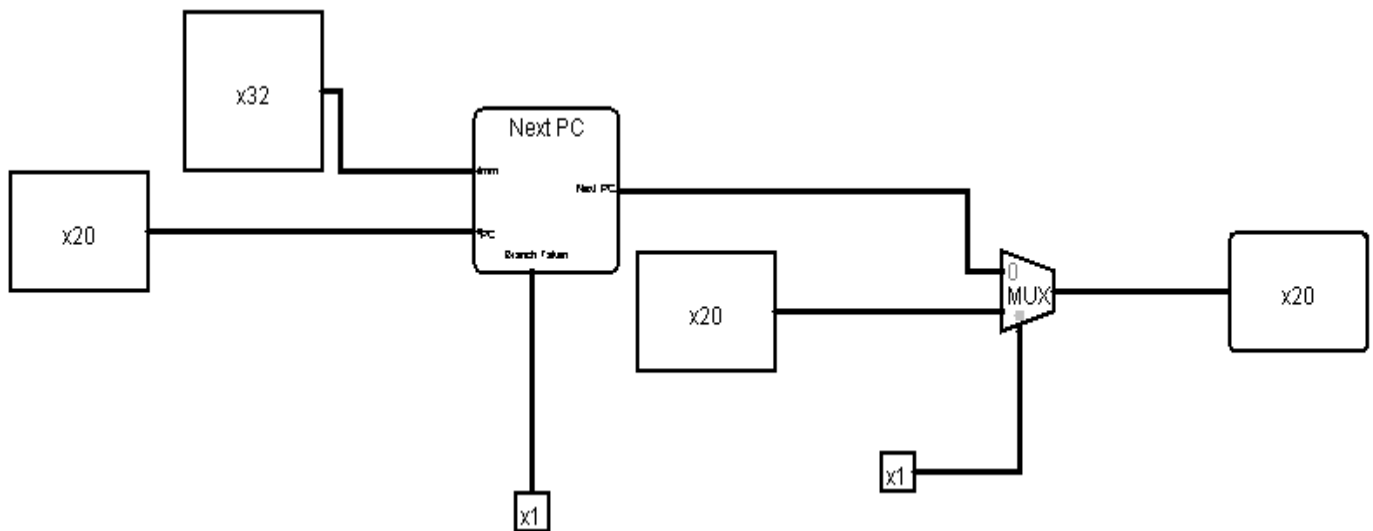
The MUX selects between jump and branch operations based on two control signals:

One signal for branch operations.

One signal for jump operations.

This allows the appropriate PC update to be selected based on the current instruction type and condition.





## 4.2 Instruction Memory

In the instruction memory stage shown in *Figure 4.2*, the processor retrieves the PC and extracts the immediate values required by the instruction. To achieve this, we split the 32-bit instruction into the necessary fields for the three instruction types (R-type, I-type, and SB-type). The instruction is divided as follows:

### 1. R-type Instruction:

- Opcode (bits 0:5): This identifies the operation to be performed.
- s1 (bits 6:10): The first source register.
- s2 (bits 11:15): The second source register.
- d (bits 16:20): The destination register.
- func (bits 21:31): The function code for additional operation details.

### 2. I-type Instruction:

- Opcode (bits 0:5): Identifies the operation.
- s1 (bits 6:10): The source register.
- d (bits 11:15): The destination register.
- Imm16 (bits 16:31): The 16-bit immediate value.

### 3. SB-type (branch-type) Instruction:

- Opcode (bits 0:5): Identifies the branch operation.

- ImmL (bits 6:10): The lower portion of the immediate value.
- s1 (bits 11:15): The first source register.
- s2 (bits 16:20): The second source register.
- ImmU (bits 21:25): The upper portion of the immediate value.
- func (bits 26:31): The function code.

To handle the splitting of the instruction further, we take the Imm16 field (bits 16:31), which contains the remaining necessary variables, and split it as follows:

- s2 and sa (bits 0:4): These represent the second source register and the shift amount.
- ImmU (bits 5:15): The upper part of the immediate value.
- func (bits 16:31): The function code.

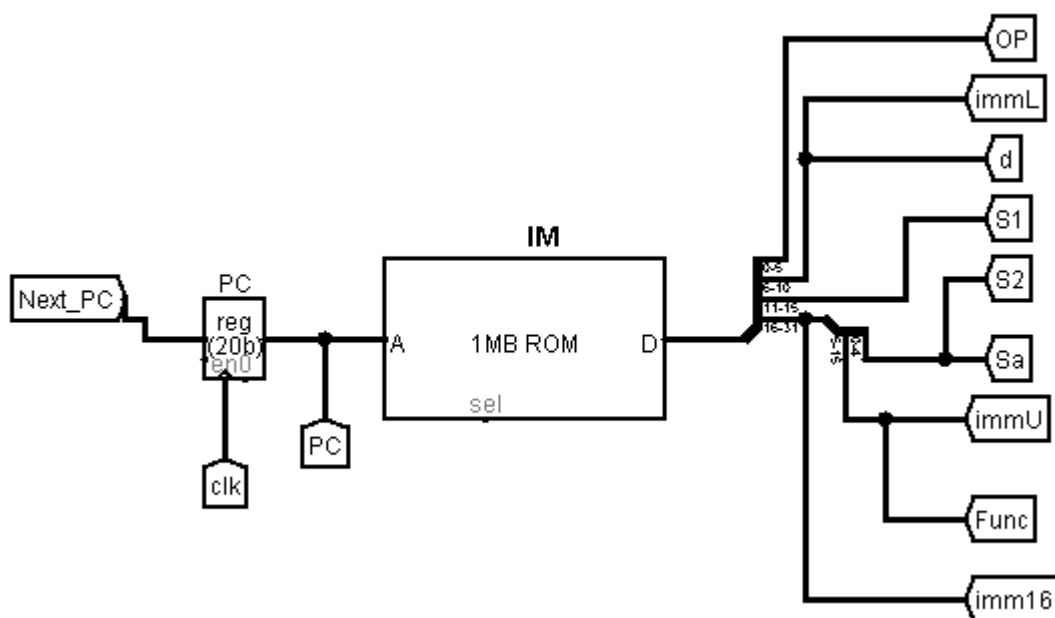


Figure 4.2

### 4.3 Register File

Now, in the general datapath, the register file shown in *Figure 4.3* is shared by the control signals, which will be described in the control unit section. By default, the register file outputs

the values of registers A and B. The input to the register file consists of two registers: s1 and s2 for R-type instructions, s1 for I-type instructions, and either s1 or d (the destination register). For SB-type instructions, both s1 and s2 are used.

To combine all these inputs, we will include s1, s2, and d, with a multiplexer (MUX) that ensures the correct selection for the I-type instruction structure. This MUX will select RA based on the signal from the source selection control (SLC). This configuration ensures that the register file receives the appropriate input depending on the instruction type.

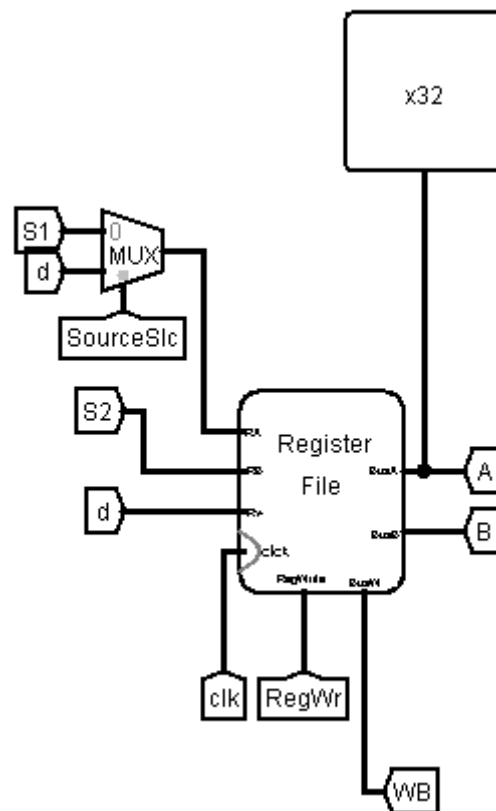


Figure 4.3

## 4.4 ALU

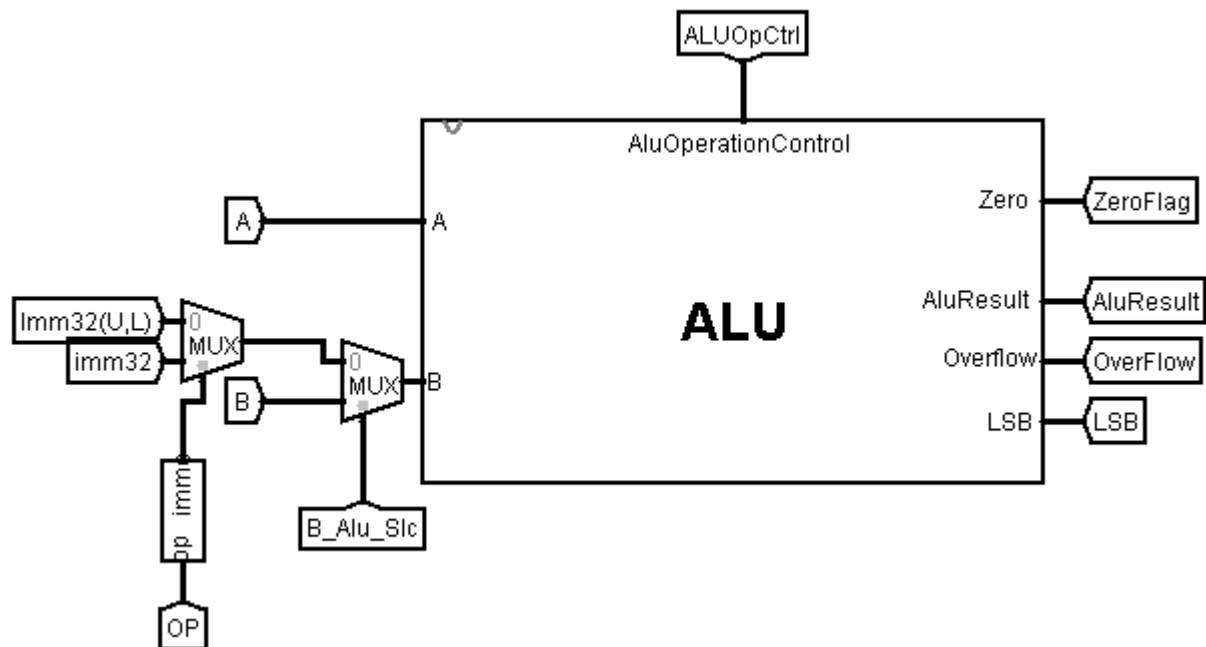
In the case of R-type instructions, the ALU takes the outputs from the register file (RA and RB), performs the execution, and generates the result, which is then passed back to the register file. For I-type instructions, the ALU uses the immediate value and one output from the register (A). In SB-type instructions, the ALU takes the output from register A and selects either the value from register B or an immediate value via a multiplexer (MUX).

*SO, HOW TO COLLECT ALL THAT IN ONE CIRCUIT?*

To combine these operations, we observe that A is shared across all instruction types, so it can be easily routed to the ALU. A multiplexer is used to choose between B or the immediate value. When focusing on the structure of I-type and SB-type instructions, we see that the source of the immediate value differs: for I-type, the immediate value is directly 16 bit sign-extended to 32 bits, while for SB-type, the immediate value is a combination of immU and immL extended to 32 bit.

To handle this difference, we add another multiplexer that selects the immediate value source, either from I-type or SB-type, based on the opcode. This ensures the ALU receives the correct immediate value for each instruction type.

Now the General Data Path ALU is ready and shown in *Figure 4.4*



*Figure 4.4*

## 4.5 Data Memory

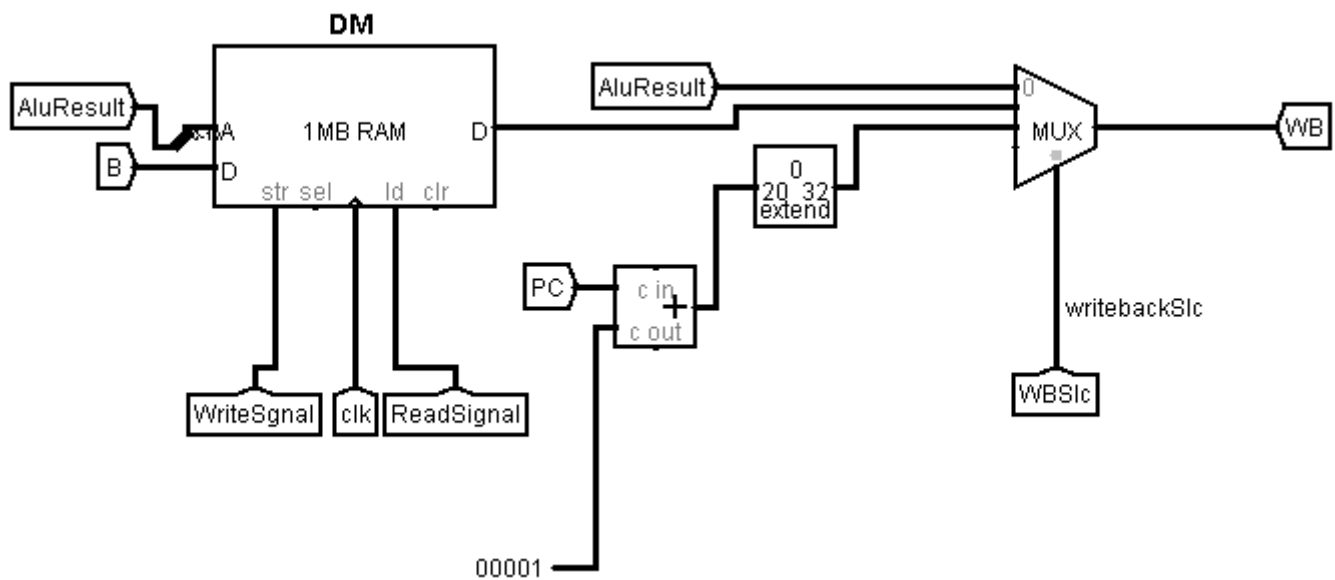
### 4.5.1 Input

In the datapath, data memory is not used in R-type instructions. For I-type instructions, data memory receives the ALU result directly. In SB-type instructions, the ALU result is used as the address for data memory, and may also provide input for storing data at that address. To combine these operations, the ALU result is shared among all instruction types, and an additional input, B, is used to satisfy the requirements of SB-type instructions.

### 4.5.2 Output

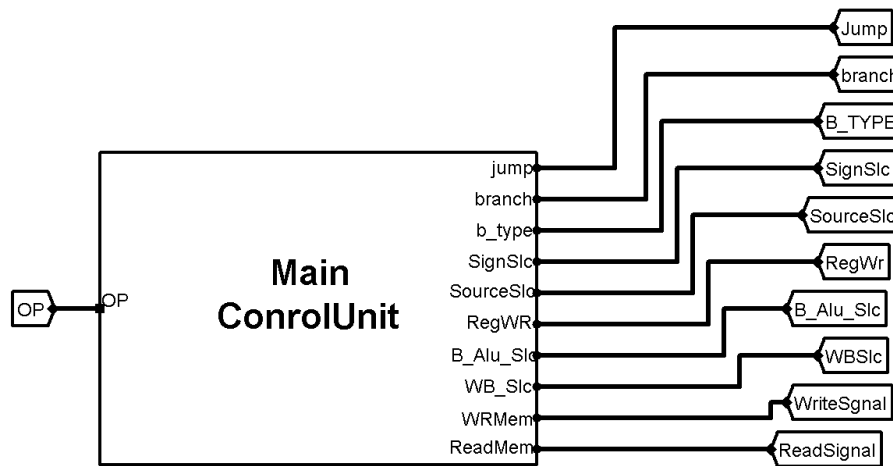
Regarding the output from data memory, R-type instructions do not utilize data memory since the ALU result is passed directly back to the register file. In SB-type instructions, data memory is used for storing data (e.g., with the sw instruction), but does not return any data to the register file. In I-type instructions, the output of data memory may be used based on various conditions: it can either be the result of PC+1 (extended from 20 bits to 32 bits), the ALU result, or the data memory output. A multiplexer (MUX), controlled by the wbslc signal, selects the appropriate source based on the instruction type.

To combine these operations in one cycle, the output from data memory in I-type instructions is handled by the same mechanism, ensuring the correct data is selected.



## 4.1.Main Control Unit

The Main Control Unit shown in *Figure 4.4.1* serves as the central decision-making block in the processor's datapath, generating the control signals that orchestrate the operation of all major functional units.



*Figure 4.4.1*

In this project, a systematic and modular approach was followed to design the Control Unit in a clear, scalable, and maintainable manner.

The design methodology was divided into two major phases:

### 4.1.1. Modular Control Units Based on Instruction Types

Initially, separate control modules were conceptually designed for each major instruction type, based on their specific datapath requirements:

#### 4.4.1.1 R-Type Instructions (Arithmetic/Logic Operations)

The Opcode field is always set to `000000`.

Upon detecting an R-Type instruction, the Main Control Unit performs minimal control actions:

1. It enables writing to the Register File by setting `RegWrite = 1`.
2. It selects both ALU operands from the Register File (setting `ALUSrc = 1`).

3. It disables memory operations by setting  $\text{MemRead} = 0$  and  $\text{MemWrite} = 0$ .
4. It disables control flow changes by setting  $\text{Branch} = 0$  and  $\text{Jump} = 0$ .

However, the Main Control Unit does not determine the specific ALU operation to be performed (such as ADD, SUB, AND, or OR).

Instead, the detailed ALU behavior is controlled by a separate ALU Control Unit, which operates as follows:

- The ALU Control Unit receives the Function field as an input.
- Based on the Function code, the ALU Control Unit generates the specific ALU control signals required to perform the appropriate arithmetic or logic operation.

This modular design allows the processor to handle a wide range of R-Type operations flexibly while keeping the Main Control Unit simple and efficient.

A detailed explanation of the ALU Control Unit design and its operation is provided separately in this documentation.

#### **4.4.1.2 I-Type Instructions (Immediate Operations)**

The Control Unit for I-Type instructions was carefully designed to support a wide range of operations where one operand is a register value and the other is an immediate value or a shift amount, depending on the specific instruction.

The goal of the I-Type control unit is to generate the appropriate control signals to configure the datapath correctly for all I-Type instructions while maintaining simplicity and efficiency.

#### **Design Strategy**

The design approach began by analyzing the common operational structure of I-Type instructions.

Regardless of specific function, all I-Type instructions generally share key characteristics:

- They require access to one source register (RS1).
- They utilize an immediate value (Imm16) or a shift amount (Sa) as the second operand.
- They typically write the result into a destination register (Rd).

- They do not, in general, affect memory (except for load instructions).
- They do not involve conditional branching or direct jump control (with the exception of JALR).

Recognizing these similarities allowed for designing a unified I-Type control module that generates a fixed set of signals for most I-Type instructions, with specific variations handled separately when necessary.

### Control Signal Generation

The I-Type control logic was developed to assert the following primary control signals:

- **RegWrite = 1:**  
Enables writing the ALU (or memory read) result into the destination register Rd, which is essential for all computation-based and load operations.
- **ALUSrc = 0:**  
Selects the immediate value or shift amount as the second operand input to the ALU instead of a second register value (RS2).
- **SignSrc:**  
Determines whether the immediate value is sign-extended (for operations like ADDI, SLTI) or zero-extended (for operations like ANDI, ORI).  
This distinction is crucial for correct data interpretation.
- **Memory Access Signals:**
  - For typical ALU-based I-Type operations:  
MemRead = 0, MemWrite = 0.
  - For memory load instructions (LW):  
MemRead = 1 (to enable reading from Data Memory).
- **Branch = 0, Jump = 0:**  
Set to zero in standard I-Type operations, as they do not affect the control flow, except for JALR, which asserts the Jump signal separately.

Thus, the I-Type control unit ensures that the datapath is configured to read one register, use an immediate value appropriately, compute the result in the ALU, and write the result back to the Register File



### 4.4.1.3 SB-Type Instructions

The Control Unit for SB-Type instructions was designed to support two main categories of operations within the processor:

- **Conditional branch operations**, where the control flow changes based on a comparison between two registers.
- **Store operation**, where data from a register is written into memory.

Thus, the SB-Type Control Unit must generate control signals that configure the datapath correctly for both branching and memory writing scenarios.

#### Design Strategy

The design methodology for the SB-Type Control Unit began by carefully identifying the distinct operational behaviors within this instruction class.

Branch instructions such as BEQ, BNE, BLT, and BGE require conditional evaluation, while store instructions like SW involve memory writing operations without altering control flow.

This classification led to designing two parallel paths within the SB-Type Control Unit:

- One for configuring the datapath for conditional branch evaluation,
- One for configuring the datapath for memory store operations.

The Control Unit differentiates between these paths based on the Opcode.

#### Control Signal Generation

The SB-Type Control Unit generates control signals as follows, depending on the specific instruction:

- **For Branch Instructions (BEQ, BNE, BLT, BGE, etc.):**
  - Branch = 1: Activates the condition evaluation path.
  - B\_type1 and B\_type0: Specify the particular branch condition type.
  - RegWrite = 0: Disables writing to the Register File.
  - ALUSrc = 1: Selects register operand for ALU second input.

- MemRead = 0: No memory read occurs.
- MemWrite = 0: No memory write occurs.
- Jump = 0: Indicates a conditional branch, not an unconditional jump.
- **For Store Instructions (SW):**
  - Branch = 0: No branch condition evaluation is needed.
  - RegWrite = 0: No register write-back occurs.
  - ALUSrc = 0: Selects an immediate value to calculate the memory address (offset addressing).
  - MemRead = 0: No memory read occurs.
  - MemWrite = 1: Activates memory writing to store data.
  - Jump = 0: Store instructions do not involve jump control.

### **Interaction with the Condition Control Unit**

It is important to note that the SB-Type Control Unit does not directly evaluate the branch condition itself. Instead, it prepares all required control signals and operands, while the Condition Control Unit, discussed separately in this documentation, performs the actual evaluation by analyzing ALU output flags (ZeroFlag, LSB) and generating the BranchTaken signal.

This separation of responsibilities ensures modularity, clarity, and easier maintenance of the control path design

## 4.1.2. Merging into a General Control Unit

After designing individual control modules for each instruction type (R-Type, I-Type, and SB-Type), the next phase involved integrating them into a single unified Main Control Unit. This final Control Unit is capable of decoding any instruction and generating the appropriate set of control signals to steer the datapath components correctly.

### Integration Strategy

The integration was performed based on the instruction Opcode, which serves as the primary identifying field for each instruction type.

Upon decoding the Opcode, the Main Control Unit selects the corresponding set of control signals designed for that instruction class:

- If the Opcode corresponds to an R-Type instruction, control signals for register-register ALU operations are activated, and detailed ALU control is delegated to the ALU Control Unit.
- If the Opcode corresponds to an I-Type instruction, control signals are configured to support operations with immediate values, memory loads, or jumps, depending on the subtype.
- If the Opcode corresponds to an SB-Type instruction, control signals are set to enable branch condition evaluation, forwarding the necessary signals to the Condition Control Unit.

### Tools Used

- **Logisim's Combinational Analysis Tool:** was used to automate the generation and minimization of the truth table into a combinational circuit.
- The entire truth table was manually prepared first, matching all instruction behaviors exactly.
- Minimization reduced the number of gates and logic complexity.

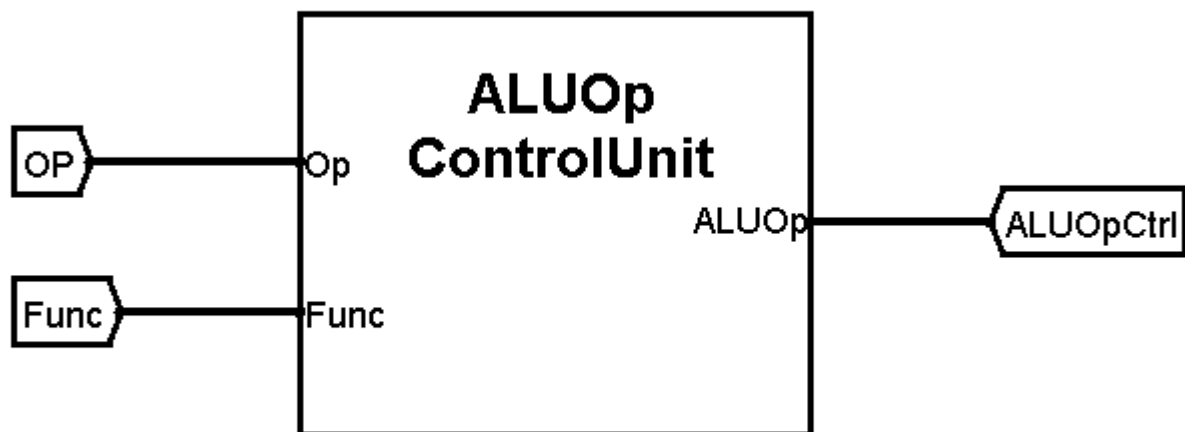
## 4.2Alu Control Unit

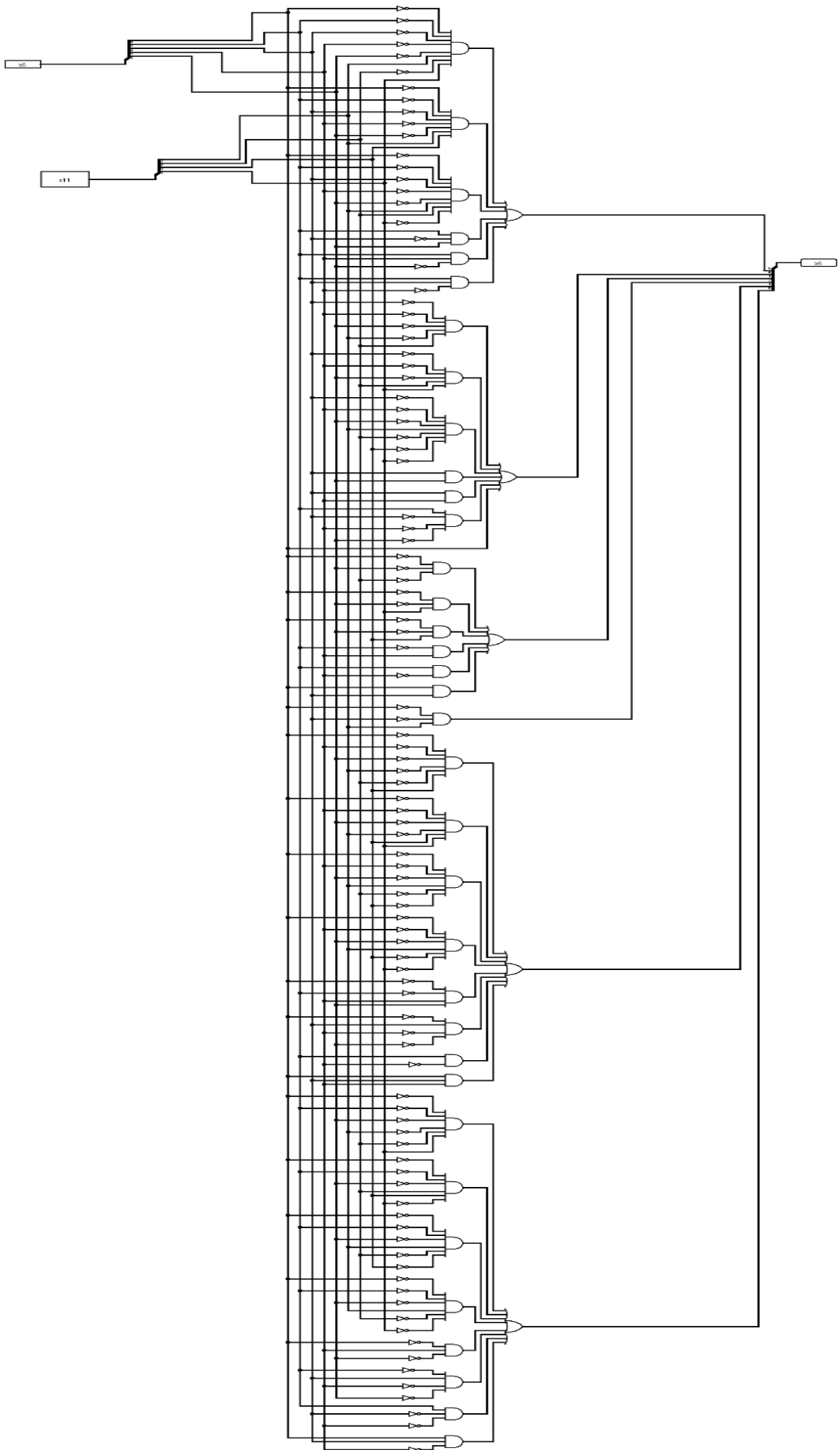
### 4.2.1. Introduction

The ALU Control Unit is a fundamental subcomponent of the processor's control logic in the 32-bit Risc architecture. Its primary purpose is to generate precise control signals that dictate the specific operation to be performed by the Arithmetic and Logic Unit (ALU) during the execution stage.

While the main control unit broadly categorizes instructions (such as whether an instruction is arithmetic, logical, memory-related, or a branch), it is the responsibility of the ALU Control Unit to translate these categories into specific ALU operations. For example, for R-type instructions, it determines whether the ALU should add, subtract, perform logical AND, logical OR, or any other operation based on the function field. For I-type instructions, it directs the ALU based on the opcode and the immediate extension.

Thus, the ALU Control Unit acts as an essential intermediary between the main control unit and the ALU, ensuring the correct execution of arithmetic, logical, shift, and comparison instructions.





### 4.2.2. Functional Overview

The ALU Control Unit receives two main inputs:

- Opcode(op): From instruction (e.g., R-type uses opcode 0, I-type and SB-type have different opcodes)
- Function Field (Func): For R-type instructions, a specific field in the instruction that further refines the operation (e.g., distinguish between ADD, SUB, AND).

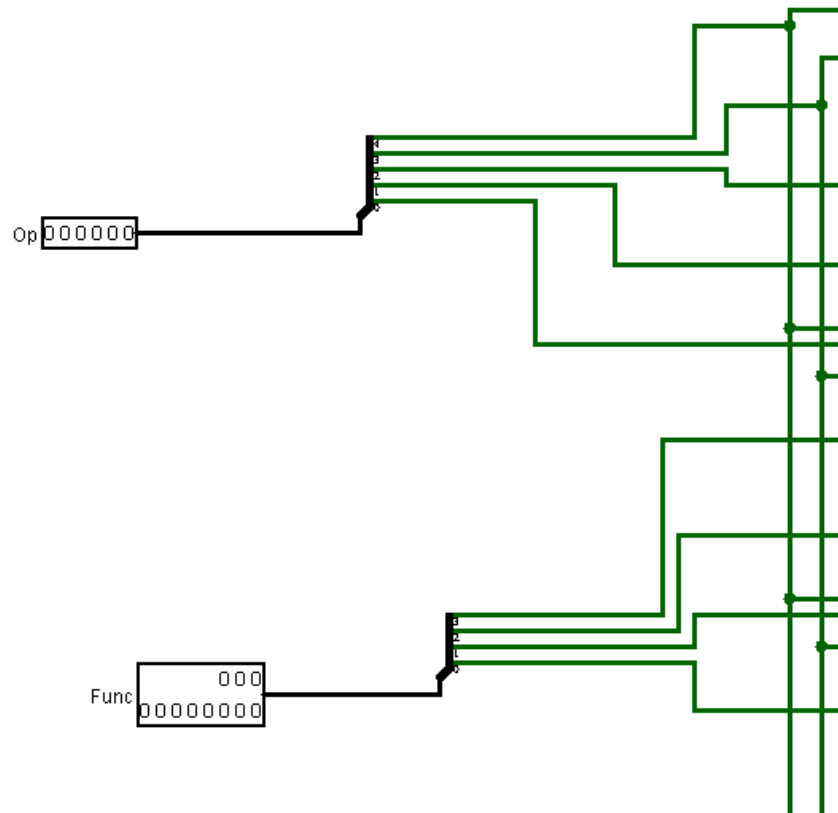
Based on these inputs, the ALU Control Unit produces an output:

- ALUControl Signals(AluOp): A set of binary control lines that precisely define the ALU operation to be performed.

### 4.2.3. Hardware Design of the ALU Control Unit

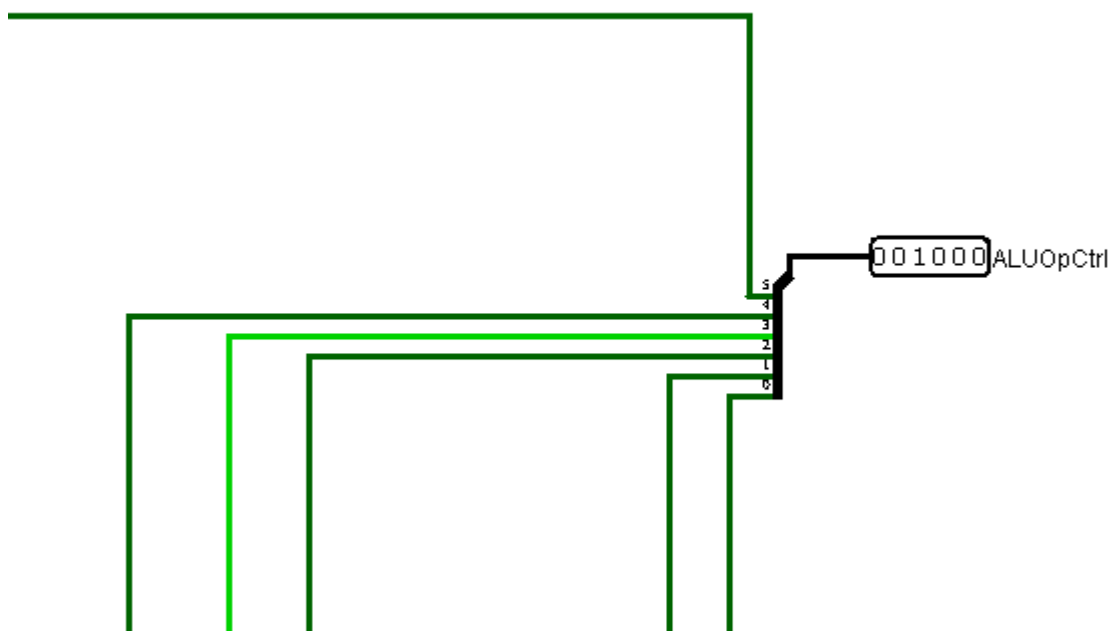
#### 4.2.3.1 Inputs:

- **Op(6 bits):**
  - Provided by the instruction.
- **Function Field (11 bits):**
  - Comes from the instruction itself (for R-type instructions only).
  - Specifies the exact operation (e.g., ADD, SUB, AND, OR, etc.).



#### 4.2.3.2 Output

- **ALUControl (6 bits):**
  - Drives the ALU's internal multiplexers and function selectors to perform the correct operation.

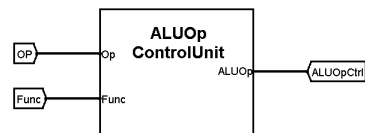


### 4.2.3.3 Hardware Structure

#### 4.2.3.3.1 Control Logic (Combinational Circuit)

- Combinational logic gates (AND, OR, NOT, XOR) are used to map input conditions (ALUOp and Function) to output control signals

#### 4.2.3.4 Conceptual Block Diagram



Internally:

- If ALUOp specifies "R-type", use Function Field to determine the operation.
- If ALUOp specifies "Branch" (e.g., BEQ), set ALU to perform subtraction.
- If ALUOp specifies "Load/Store", set ALU to perform addition (address calculation).

#### 4.2.3.5 ALU Control Signal Mapping

Op(6-bit)	Func(11-bit)	instruction	AluOpCtrl(6-bit)			
			Alu-Result-Stc(2 bit)	AluOp(1 bit)	ArthOp(1 bit)	Result-Stc(2 bit)
0	0	SLL	0	x	x	0
0	1	SRL		x	x	1
0	10	SRA	0	x	x	10
0	11	ROR	0	x	x	11
0	100	ADD	1	0	0	0
0	101	SUB	1	1	0	0
0	110	SLT	1	1	x	1
0	111	SLTU	1	x	x	10
0	1000	SEQ	1	1	x	11
0	1001	XOR	10	x	x	11
0	1010	OR	10	x	x	1
0	1011	AND	10	x	x	0
0	1100	NOR	10	x	x	10
0	1101	MUL	1	x	1	0
1	x	SLLI	0	x	x	0
10	x	SRLI	0	x	x	1
11	x	SRAI	0	x	x	10
100	x	RORI	0	x	x	11
101	x	ADDI	1	0	0	0
110	x	SLTI	1	1	x	1
111	x	SLTU	1	x	x	10
1000	x	SEQI	1	1	x	11
1001	x	XORI	10	x	x	11
1010	x	ORI	10	x	x	1
1011	x	ANDI	10	x	x	0
1100	x	NORI	10	x	x	10
1101	x	SET	11	x	x	x0
1110	x	SSET	11	x	x	x1
1111	x	JALR(add)	1	0	0	0



10000	x	LW(add)	1	0	0	0
10001	x	SW(add)	1	0	0	0
10010	x	BEQ(sub)	1	1	0	0
10011	x	BNE(sub)	1	1	0	0
10100	x	BLT(slt)	1	1	x	1
10101	x	BGE(slt)	1	1	x	1
10110	x	BLTU(sltu)	1	x	x	10
10111	x	BGEU(sltu)	1	x	x	10

## **5.1.Condition Control Unit**

The Condition Control Unit is a dedicated combinational logic block within the processor architecture, designed specifically to evaluate branch conditions for SB-type instructions.

It plays a crucial role in dynamic program control flow by determining whether a conditional branch should be taken based on register comparison results generated by the ALU.

### **5.3.1. Purpose**

The main purpose of the Condition Control Unit is to generate the BranchTaken signal, which indicates whether the Program Counter (PC) should jump to a branch target address or continue sequentially.

Its function is tightly coupled with SB-type conditional branch instructions:

- BEQ (Branch if Equal)
- BNE (Branch if Not Equal)
- BLT (Branch if Less Than)
- BGE (Branch if Greater Than or Equal)
- BLTU (Branch if Less Than Unsigned)
- BGEU (Branch if Greater Than or Equal Unsigned)

The decision to branch depends on comparison results obtained from the ALU and control signals provided by the instruction decoding process.

### **5.3.2. Inputs and Outputs**

Name	Width	Type	Description
Branch	1 bit	Input	Indicates whether the current instruction is a branch.
BranchType1	1 bit	Input	Higher bit of the branch type encoding.
BranchType0	1 bit	Input	Lower bit of the branch type encoding.
ZeroFlag (ZF)	1 bit	Input	Output from ALU indicating whether the two operands are equal (RS1 == RS2).

LSB	1 bit	Input	Output from ALU indicating the most significant bit of the comparison result (sign bit for signed operations).
BranchTaken	1 bit	Output	Output signal asserted if the branch condition is satisfied.

### 5.3.3. Truth Table

The Condition Control Unit operates according to the following truth table:

Branch	BranchType1	BranchType0	ZeroFlag (ZF)	LSB	BranchTaken
0	X	X	X	X	0
1	0	0	0	X	0
1	0	0	1	X	1
1	0	1	0	X	1
1	0	1	1	X	0
1	1	0	X	0	0
1	1	0	X	1	1
1	1	1	X	0	1
1	1	1	X	1	0

X denotes a "don't care" condition.

**Interpretation:**

- When  $\text{Branch} = 0$ , the unit forces  $\text{BranchTaken} = 0$  (the instruction is not a branch).
- For  $\text{Branch} = 1$ :
  - $\text{BranchType} = 00$  (BEQ): Branch if  $\text{ZeroFlag} = 1$ .
  - $\text{BranchType} = 01$  (BNE): Branch if  $\text{ZeroFlag} = 0$ .
  - $\text{BranchType} = 10$  (BLT/BLTU): Branch if  $\text{LSB} = 1$ .
  - $\text{BranchType} = 11$  (BGE/BGEU): Branch if  $\text{LSB} = 0$ .

### 5.3.4. Internal Logic Structure

The Condition Control Unit is implemented as a purely combinational circuit as shown in *Figure 5.1.1* using:

- Inverters (NOT gates) to negate input conditions where necessary.
- AND gates to combine input conditions corresponding to specific branch types.
- OR gate to collect the results from the individual conditions into the final **BranchTaken** output.

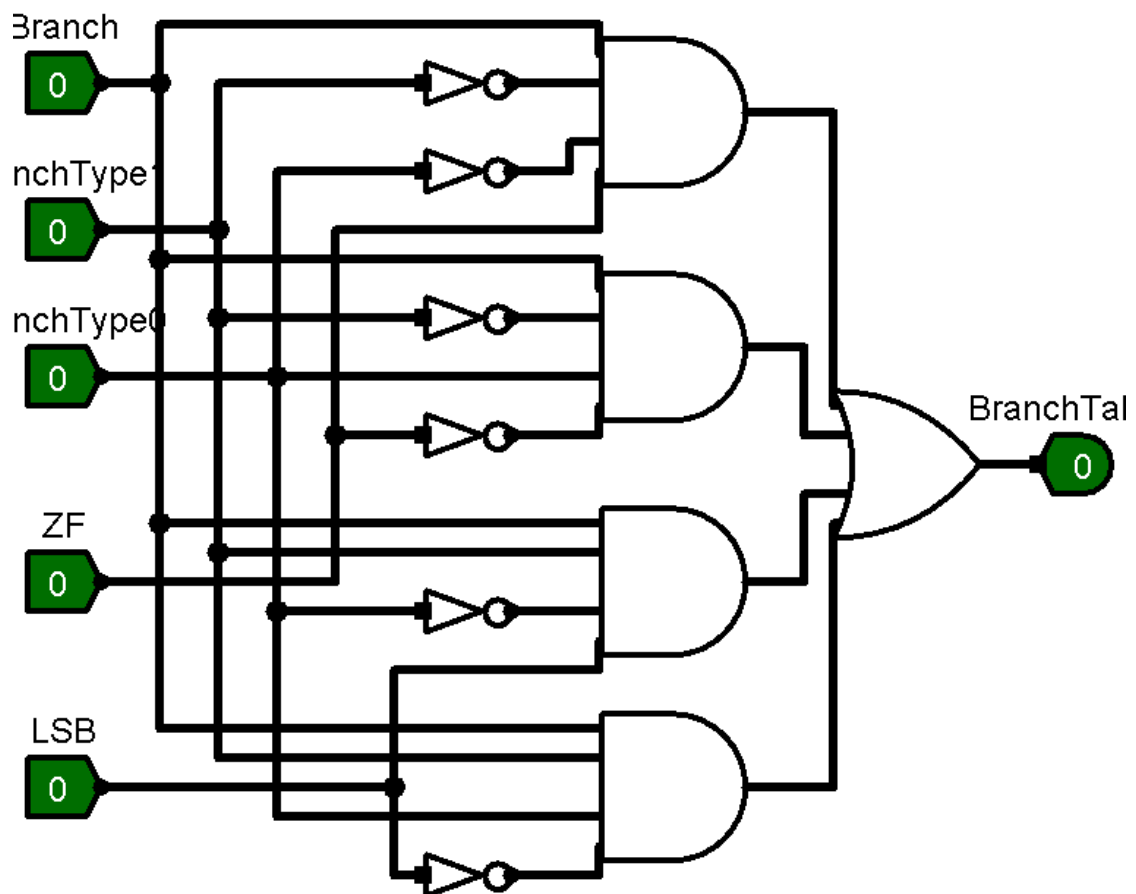


Figure 5.1.1

### 5.3.5. Functional Flow

The working of the Condition Control Unit follows this sequence:

- 1. Instruction Decode:**

The processor decodes the instruction and identifies it as a branch instruction, asserting the Branch signal.

- 2. ALU Comparison:**

The ALU generates the ZeroFlag and LSB based on the comparison of RS1 and RS2.

- 3. Branch Condition Evaluation:**

The Condition Control Unit evaluates the condition using the BranchType, ZeroFlag, and LSB.

**4. Branch Decision:**

If the branch condition is satisfied, BranchTaken is asserted.

**5. Next PC Selection:**

The BranchTaken output is forwarded to the Next PC unit to determine the correct next instruction address.

## 3.1 DataPaths Testing

### 3.1.1 RType Testing

Assuming R1 = 2, R3 = 3

Instruction	Destination	Expected	Actual
SLL R4, R1, R3	R4	8	8
SRL R5, R1, R3	R5	0	0
SRA R6, R1, R3	R6	0	0
ROR R7, R1, R3	R7	2	2
ADD R8, R1, R3	R8	5	5
SUB R9, R1, R3	R9	-1	-1
SLT R10, R1, R3	R10	1	1
SLTU R11, R1, R3	R11	1	1
SEQ R12, R1, R3	R12	0	0
XOR R13, R1, R3	R13	1	1
OR R14, R1, R3	R14	3	3
AND R15, R1, R3	R15	2	2
NOR R16, R1, R3	R16	-4	-4
MUL R17, R1, R3	R17	6	6



Correctness		100%	100%
-------------	--	------	------

### 3.1.2 I Type Testing

Instruction	Destination	Expected	Actual
SET R1, 5	R1	5	5
SLLI R2, R1, 2	R2	20	20
SRLI R3, R1, 2	R3	1	1
SRAI R4, R1, 2	R4	1	1
RORI R5, R1, 5	R5	0x40000001	0x40000001
ADDI R6, R1, 5	R6	10	10
SLTI R7, R1, 6	R7	1	1
SLTIU R8, R1, 7	R8	1	1
SEQI R9, R1, 5	R9	1	1
XORI R10, R1, 0X0011	R10	20	20

ANDI R11, R1, 0X0011	R11	1	1
ORI R12, R1, 0X0011	R12	21	21
NORI R13, R1, 0X0011	R13	-22	-22
LW R6, 0(R2)	R6	Mem[20]	Mem[20]
JALR R0, R7, 0	R0	0	0
Correctness		100%	100%

### 3.1.3 SB Type Testing

#assume R1 = 3 , R2 = 1

Instruction	Destination	Expected	Actual
SW R1, 2(R0)	Memory[2]	3	3
BLT R1, R2, loop	loop	Not taken	Not taken
BGE R1, R2, break	break	taken	taken
BNE R1, R0, target	target	taken	taken
BLTU R0, R2, next	next	taken	taken
BGEU R2, R0, final	Final	taken	taken
Correctness		100%	100%

## 3.3 General Data Path Testing

### 3.3.1 Test Code

We have successfully completed the assigned test code, along with other related files, and they are all functioning as expected. You can observe the correctness of the data flow by reviewing the video linked below. Additionally, we have implemented the necessary modifications and updates in the project files, which are available for review in the repository at the following link:

GitHub Repository: <https://github.com/omnia197/32-bit-RISC-Processor>

Within the repository, you will find:

- Project Files: Comprehensive implementation files detailing the processor design and testing.
- Test Results:  
You can view the output of the test code in the file named TestCodeOutput.pdf located in the project repository. This file documents the results and verifies that the processor behaves as intended.

Hex File:

To inspect the compiled hex code used for testing, you can access the file test.hex located at:

These materials together demonstrate the successful implementation and correctness of the processor design and data handling.

We have also made it as a screenRec Video to show the data flow over the processor you could access it here:

[https://www.canva.com/design/DAGl87iDt40/leUk\\_N5736inpGF4yTq3aw/watch?utm\\_content=DAGl87iDt40&utm\\_campaign=designshare&utm\\_medium=link2&utm\\_source=uniquelinks&utlId=h6f16c5b4e8](https://www.canva.com/design/DAGl87iDt40/leUk_N5736inpGF4yTq3aw/watch?utm_content=DAGl87iDt40&utm_campaign=designshare&utm_medium=link2&utm_source=uniquelinks&utlId=h6f16c5b4e8)

Also to be ensured we have done it correctly we have made a real-time test on the video describing the project, you could access it here:

[https://www.canva.com/design/DAGl7qpneN4/hM22F8vpWSDxCd\\_imp6EEQ/watch?utm\\_content=DAGl7qpneN4&utm\\_campaign=designshare&utm\\_medium=link2&utm\\_source=uniquelinks&utlId=h994518c5a2](https://www.canva.com/design/DAGl7qpneN4/hM22F8vpWSDxCd_imp6EEQ/watch?utm_content=DAGl7qpneN4&utm_campaign=designshare&utm_medium=link2&utm_source=uniquelinks&utlId=h994518c5a2)

## 6. Assembler

An assembler is a crucial component in the software development toolchain, particularly for low-level programming and processor design. Its main role is to translate human-readable assembly language instructions into machine-readable binary code (also known as object code). Assembly language is a low-level programming language that is closely related to machine instructions for a specific architecture, but it uses instructions (like ADD, SUB, BEQ) instead of raw binary. By developing my own assembler, you have built a tool that automates this translation process, allowing programs written in assembly language to be executed by the processor.

### 6.1 Structure

#### 6.1.1 Models

We begin developing the assembler by first identifying all the essential modules and the different types of inputs it may receive. This includes analyzing the full content of the input file, then breaking it down line by line to extract each instruction. For every line, we determine whether it contains a label, and if so, we collect the label and its associated variables.

This structure helps ensure that the assembler correctly interprets and processes all parts of the input

```
models/  
├── __init__.py  
├── instruction.py  
├── label.py  
└── line_assembly.py
```

```
@dataclass  
class AssemblyLine:  
    content: str  
    line_number: int  
    instruction: Optional[Instruction] = None  
    label: Optional[Label] = None
```

#### 6.1.2 Parsers

After identifying the input structure, we introduce parsers that process the input and extract the necessary parameters. In the immediate (IMM) parser, we pass the immediate value while considering that we are working with 16-bit signed values. Therefore, the maximum value is 32767

and the minimum value is -32768. If the immediate value is negative, it may require sign extension, meaning we must add additional '1' bits to maintain its correct signed representation.

In the line parser, the input line is cleaned by removing any unnecessary parameters, such as extra spaces or comments. It also checks if the line contains a label and processes it accordingly.

In the register file, when dealing with register inputs, we ensure the register number is within the valid range of 0 to 31, following our model's 32-register structure.

Finally, we have the instruction parser, which extracts the opcode from the instruction and determines its format. If the instruction is a simple form like `ADD R2, R1, R1`, it directly maps the registers. However, if the instruction is of a more complex form, such as `lw R6, 0(R2)`, we need to further divide the instruction into an offset part and a register part for correct parsing and processing.

parsers/

├── \_\_init\_\_.py

├── imm\_parser.py

├── instruction\_parser.py

├── line\_parser.py

└── register\_parser.py

```
4
5     @staticmethod
6     def parse(reg_str: str) -> int:
7         reg = reg_str.upper()
8         if reg in RegisterParser.REGISTERS:
9             return RegisterParser.REGISTERS[reg]
10
```

```

mem_match = re.match(r'^(-?\d+)\s((\w+)\s)$', op_part)
if mem_match:
    offset = mem_match.group(1)
    register = mem_match.group(2)
    operands.append(offset)
    operands.append(register)
else:
    operands.append(op_part)

return Instruction(opcode=op, operands=operands)

```

### 6.1.3 Encoders

Next, we move on to the encoders, which group all similar methods based on instruction types. The encoder starts by fetching the parameters, including the opcode and function code (func). For R-type instructions, it first checks if the instruction exists in the instruction set. Then, it retrieves the destination register and operand values. These values are processed by shifting and performing bitwise OR operations to assemble the final binary form of the instruction.

It is totally connected with our ISA design format to be built on

encoders/

└─ \_\_init\_\_.py

└─ encoder.py

└─ i\_encoder.py

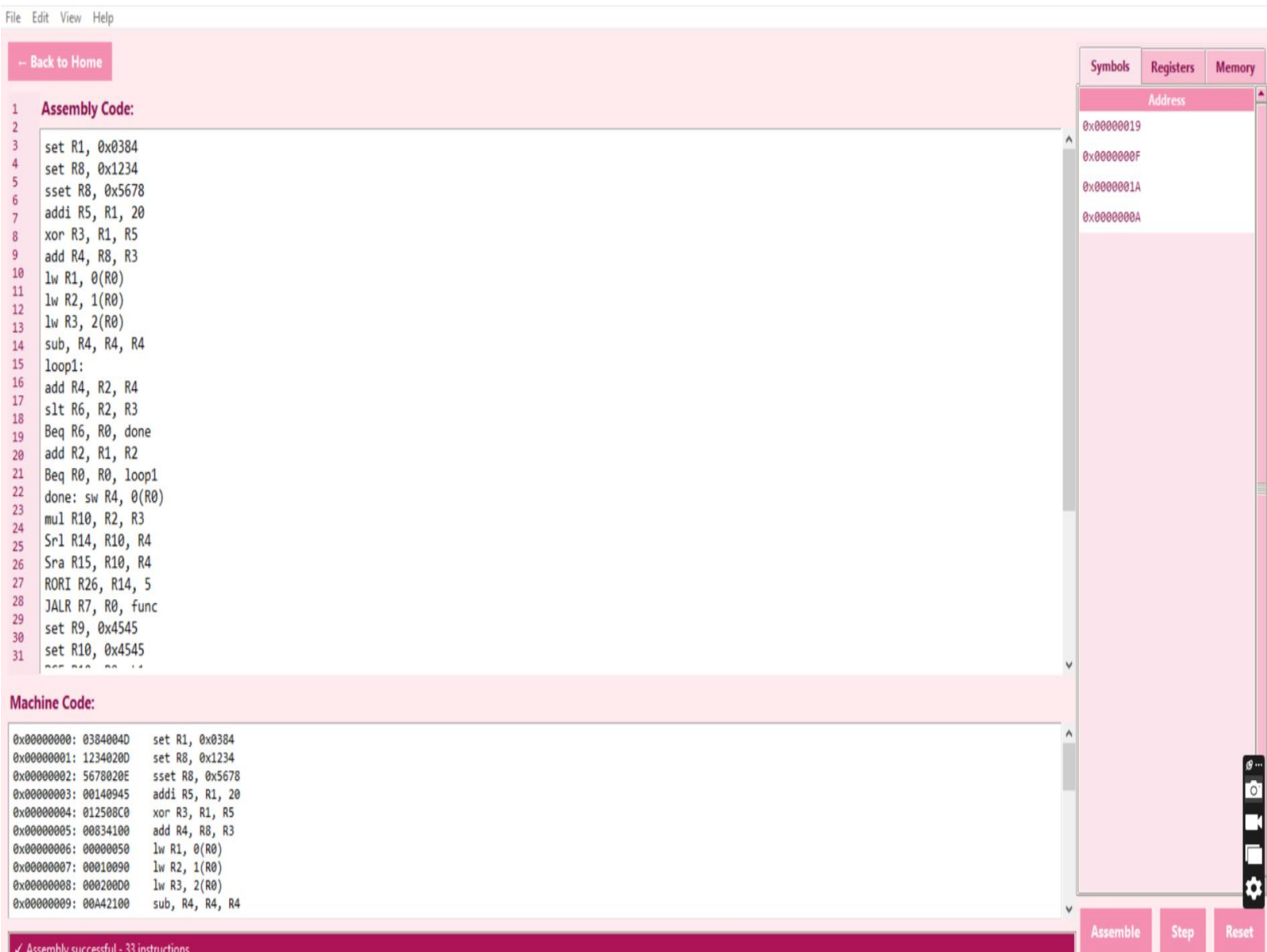
└─ sb\_encoder.py

```
def encode(self, instruction, symbol_table, current_pc):  
    f = self.FUNCTIONS[instruction.opcode]  
    d = RegisterParser.parse(instruction.operands[0])  
    s1 = RegisterParser.parse(instruction.operands[1])  
    s2 = RegisterParser.parse(instruction.operands[2])  
    return (f << 21) | (s2 << 16) | (s1 << 11) | (d << 6) | 0
```

```
if instruction.opcode == 'SW':  
    rs2 = RegisterParser.parse(instruction.operands[0])  
    offset = ImmediateParser.parse(instruction.operands[1])  
    rs1 = RegisterParser.parse(instruction.operands[2])  
    immU = (offset >> 5) & 0x7FF  
    immL = offset & 0x1F  
  
    return (immU << 21) | (rs2 << 16) | (rs1 << 11) | (immL << 6) | opcode
```

## 6.2 GUI

To improve the user experience, we develop a visually appealing and interactive GUI that is both functional and easy to navigate. The interface will feature colorful buttons and a clean layout to make it intuitive for users to interact with the system.



### 6.2.1 File Upload and Saving

The application provides options for users to upload files easily and work with different file types. Users can upload assembly files (ASM), allowing them to process and analyze the instructions within. Once the work is done, the system will allow users to save the file in ASM format for further use or editing. Additionally, users can save the output as a hexadecimal file, ensuring compatibility with various systems and making it easy to export the results.

### 6.2.2 Code Editor with Syntax Highlighting

The built-in code editor will offer advanced features like syntax highlighting to help users distinguish between different components of the code. The editor will highlight keywords, instructions, registers, and labels in various colors, making the coding process more efficient and visually engaging.



```

2
3 #shifting
4     SET R21, 1
5     SET R22, 2
6     SET R1, 1
7     SET R2, 2
8
9
10
11     #r_type
12     SLL R23, R21, R22
13     SLL R3, R1, R2
14     SRL R24, R21, R22
15     SRA R25, R21, R22
16     ROR R26, R21, R22
17
18     #i_type
19     SLLI R27, R21, 3
20     SRLI R28, R21, 1
21     SRAI R29, R21, 1
22     RORI R30, R21, 1

```

### 6.2.3 Memory Registers and Labels

As part of the code editor, we will incorporate functionality that visually separates the code into distinct memory registers and labels. This feature will help users quickly identify and work with different parts of the code. The system will provide a clear and structured view, making it easier to navigate through the instructions, understand the flow, and ensure the correct output is generated.

Symbols	Registers	Memory
Instruction/Data		
0384004D (set R1, 0x0384)		
1234020D (set R8, 0x1234)		
5678020E (sset R8, 0x5678)		
00140945 (addi R5, R1, 20)		
012508C0 (xor R3, R1, R5)		
00834100 (add R4, R8, R3)		
00000050 (lw R1, 0(R0))		
00010090 (lw R2, 1(R0))		
000200D0 (lw R3, 2(R0))		
00A42100 (sub, R4, R4, R4)		

## 6.2.4 Output Generation

Finally, the application will take the processed code and generate the desired output. Whether it's an assembly file, a hexadecimal representation. The output will be presented in an easy-to-understand manner. The combination of a fancy GUI, file management options, and color-coded editor will streamline the development and testing process for users, ensuring a smooth and efficient experience.

**Machine Code:**

0x00000000: 0384004D	set R1, 0x0384
0x00000001: 1234020D	set R8, 0x1234
0x00000002: 5678020E	sset R8, 0x5678
0x00000003: 00140945	addi R5, R1, 20
0x00000004: 012508C0	xor R3, R1, R5
0x00000005: 00834100	add R4, R8, R3
0x00000006: 00000050	lw R1, 0(R0)
0x00000007: 00010090	lw R2, 1(R0)
0x00000008: 000200D0	lw R3, 2(R0)
0x00000009: 00A42100	sub, R4, R4, R4

✓ Assembly successful - 33 instructions

## 6.3 Testing & Verification

In testing the core, we ensured that all instructions were properly implemented and functioned correctly with the appropriate format. These instructions were used to test the general datapath and control unit of our RISC processor, ensuring their seamless integration and functionality

## 6.4 Installation

### 6.4.1 Prerequisites

#### Software Requirements

##### 1. Python:

- This assembler is built using Python, so you will need to have Python installed on your system. It is recommended to use Python 3.6 or above for compatibility with the libraries used in the project.

To verify if Python is installed on your system, open a terminal or command prompt and run:

```
python --version
```

- If Python is not installed, you can download and install it from the official Python website: [Python Downloads](#).

## 2. Git:

- You will need Git to clone the repository from GitHub. If you don't have Git installed, download it from the official site: [Git Downloads](#).

To check if Git is installed, use the following command:

```
git --version
```

## 3. Visual Studio Code (VS Code) (Optional, but recommended)

### Getting the Project

#### 1. Cloning the Repository:

Clone the assembler project from the GitHub repository to your local machine using the following Git command:

```
git clone https://github.com/omnia197/32-bit-RISC-Processor.git
```

### 6.4.2 Usage Options

Once you have the project set up, there are two main ways to use the assembler:

#### 1. Command-Line Interface (CLI):

- The program can be run directly from the terminal within Visual Studio Code using Python. To do this, you will invoke the main.py script from the assembler directory.
- Open the integrated Terminal in Visual Studio Code by selecting Terminal > New Terminal from the top menu, and then run the following command  
`python assembler/main.py input.asm output.asm`
- Here:

`input.asm`: The input assembly file you want to assemble.

`output.asm`: The output file where the result will be saved.

#### Example:

```
python assembler/main.py myprogram.asm myprogram_output.asm
```

#### 2. Graphical User Interface (GUI):

- Alternatively, you can use the Graphical User Interface (GUI), which is built using the UI.py class. This option allows you to run the assembler in a more user-friendly manner with interactive controls.

To use the GUI, simply run the UI.py

This will launch the GUI, where you can load an input file and interact with the program without needing to type commands manually. The GUI interface will provide options to load and process your assembly file.

## 7.1 Challenges

### 7.1.1 Hardware Challenges

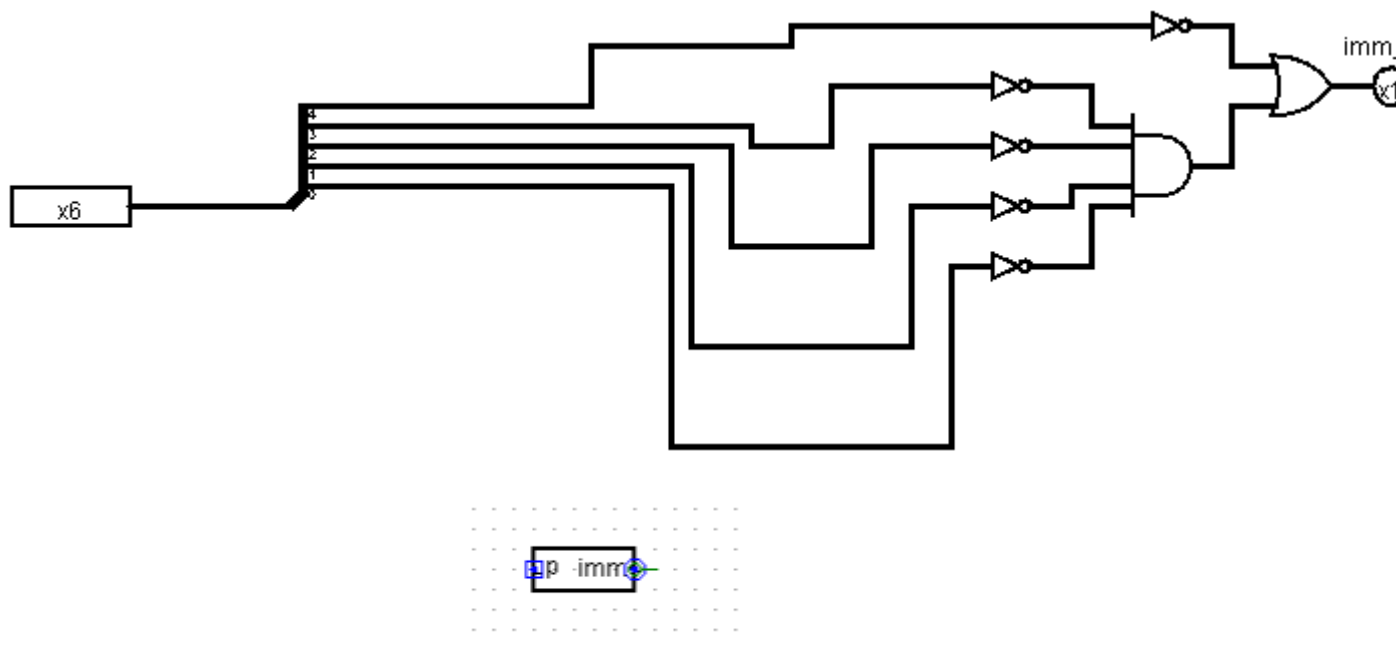
#### 7.1.1.1 Problem

During the design process of the control unit, we faced a challenge with constructing the correct truth table. Specifically, we overlooked an essential signal that needed to be included alongside the input signals. This signal was critical for selecting between the immediate value derived from the I-type or SB-type instructions. For about two minutes, this oversight caused some concern as it was unclear how to integrate this missing signal without disrupting the entire design.

WHAT COULD WE DO NOW?

#### 7.1.1.2 Solution

Fortunately, we were able to address this issue by designing the missing signal independently and then integrating it into the existing structure of the control unit. This solution allowed us to resolve the problem effectively without needing to redesign the entire control unit or modify its overall shape. The addition of the signal ensured that the control unit could select between the immediate values of I-type and SB-type instructions, thus maintaining the integrity of the original design.



## 7.1.2 Software Challenge

### 7.1.2.1 Problem

We encountered a software challenge related to understanding and handling little-endian and big-endian formats. The issue arose because the code was initially developed using little-endian format, which caused confusion when attempting to work with the desired format. The challenge was to correctly manage this disparity between the two formats.

### 7.1.2.2 Solution

We managed to solve this issue by encoding the code in the opposite format to what was originally intended. Instead of changing the entire structure of the code, we handled the encoding in little-endian and then read the results in the opposite format. This approach effectively transformed the system from little-endian to big-endian, as reading little-endian data in reverse gave us the correct big-endian format. Ultimately, this solution allowed us to achieve the correct result without overhauling the entire code, enabling us to maintain the original little-endian structure and still produce the correct output in big-endian format. (Noting that I have a problem in branching the assembler code that I have known at 28/4/2025 11:00PM so I was not have time to solve it after it was solved in the first version of the assembler)

```
with open(output_file, 'wb') as f:
    for instruction in self.output:
        f.write(instruction.to_bytes(4, byteorder='little'))
```

## 7.2 Tasks & Meetings

### 7.2.1 Meetings

Day	Start Time	End Time	Place	Description
18/4/2025	10:20 PM	10:45	Discord	Create a plan outlining our tasks, along with a discussion on the workflow and timeline
20/4/2025	1:30 PM	2:00 PM	Discord	Review and resolve our issues or errors to eliminate them early
23/4/2025	3:00 PM	6:00 PM	Offline	Testing SB Type & Handling Assembler Errors
24/4/2025	1:30PM	10PM	Offline	<ul style="list-style-type: none"><li>- Making The General DataPath</li><li>- Construct the control unit</li><li>- Make ALUOP Control Unit</li><li>- Enhancing Assembler</li></ul>
25/4/2025	1:30 PM	9:00PM	Offline	<ul style="list-style-type: none"><li>- Testing DataPath, controlUnit</li><li>- Test the code required from us.</li></ul>
27/4/2025	4:00 PM	9:00 PM	Offline	Documentation

### 7.2.2 Tasks Needed

Before collaborating on building the general datapath, we started by dividing tasks to prepare all the required circuits and different datapaths.

Task Number	Day	Assigned To	Status	Task
1	18/4/2025	Rahma	Done	RegisterFile
2	18/4/2025	Alaa	Done	ALU
3	18/4/2025	Omnia	Working On	Assembler Design
4	19/4/2025	Rahma	Didn't do	SB_Type DataPath
5	19/4/2025	Alaa	75%Done	I_Type DataPath
6	19/4/2025	Omnia	50% Done	R_Type DataPath
7	20/4/2025	Omnia	Done	Troubleshooting Assembler
8	20/4/2025	Rahma	Done	Troubleshoot Regfile
9	20/4/2025	Alaa	Done	TroubleshootALU
10	21/4/2025	Rahma	WorkingOn	Complete_SB_DataPath
11	21/4/2025	Alaa	Done	Complete_I_DataPath
12	21/4/2025	Omnia	Done	Complete_R_DataPath



### 7.2.3 Contribution

Alaa	Omnia	Rahma
<ul style="list-style-type: none"><li>- ALU</li><li>- I Type DataPath</li><li>- ControlUnit</li><li>- ALUOP Unit</li><li>- Full DataPath</li><li>- Hardware Testing</li><li>- Documentation</li><li>- Video</li></ul>	<ul style="list-style-type: none"><li>- Software Implementation</li><li>- R Type DataPath</li><li>- Full DataPath</li><li>- Hardware Testing</li><li>- Test Codes</li><li>- Documentation</li><li>- Video</li></ul>	<ul style="list-style-type: none"><li>- RegisterFile</li><li>- SB Type DataPath</li><li>- ControlUnit</li><li>- ALUOP Unit</li><li>- Full DataPath</li><li>- Hardware Testing</li><li>- Documentation</li><li>- Video</li></ul>