RISC Processor Documentation

By:
Alaa ayman mohamed
Omnia Ayman Mohamed
Rahma mostafa

Contents

Single cycle version

Pipelined version

Single Cycle Version

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.

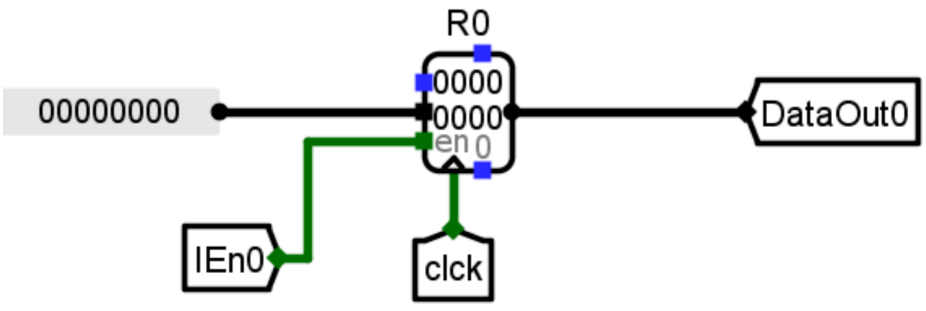


Figure 2.2.1

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`	1 bit	Input	Enables writing when asserted high

(RegWrite)			
Read Data 1 (BusA)	32 bits	Output	Data read from register specified by S1
Read Data 2 (BusB)	32 bits	Output	Data read from register specified by S2

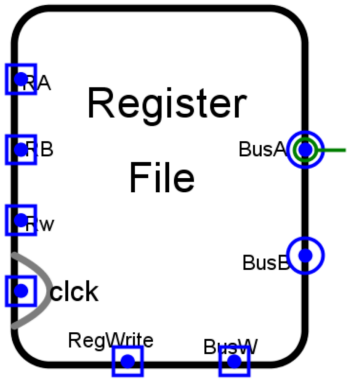


Figure 2.2.2

5. Internal Implementation

Internally, the Register File consists of:

- **32×32-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.

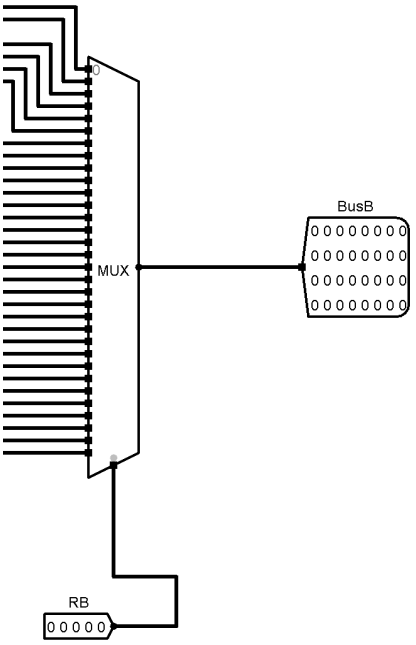


Figure 2.2.3

- **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.

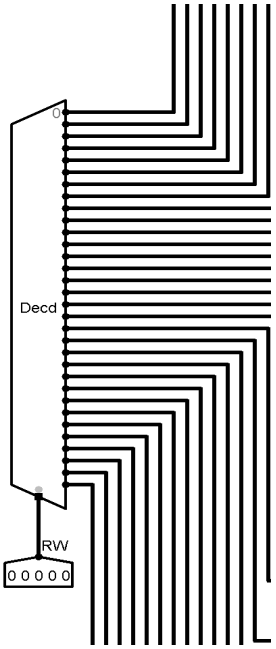


Figure 2.2.4

- **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.

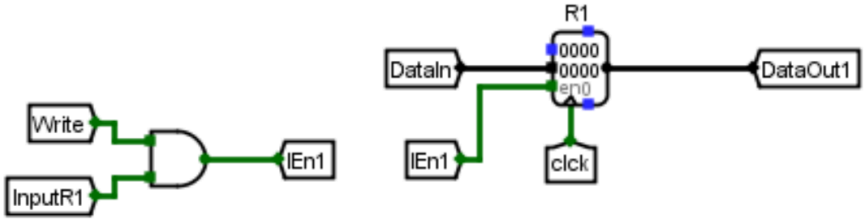


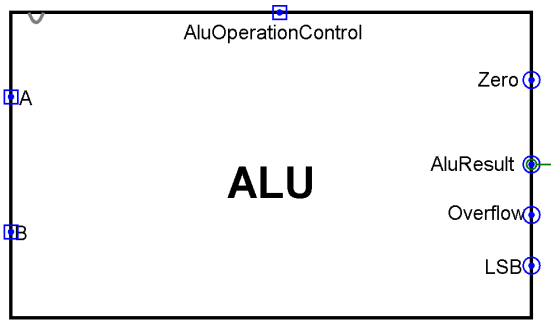
Figure 2.2.5

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

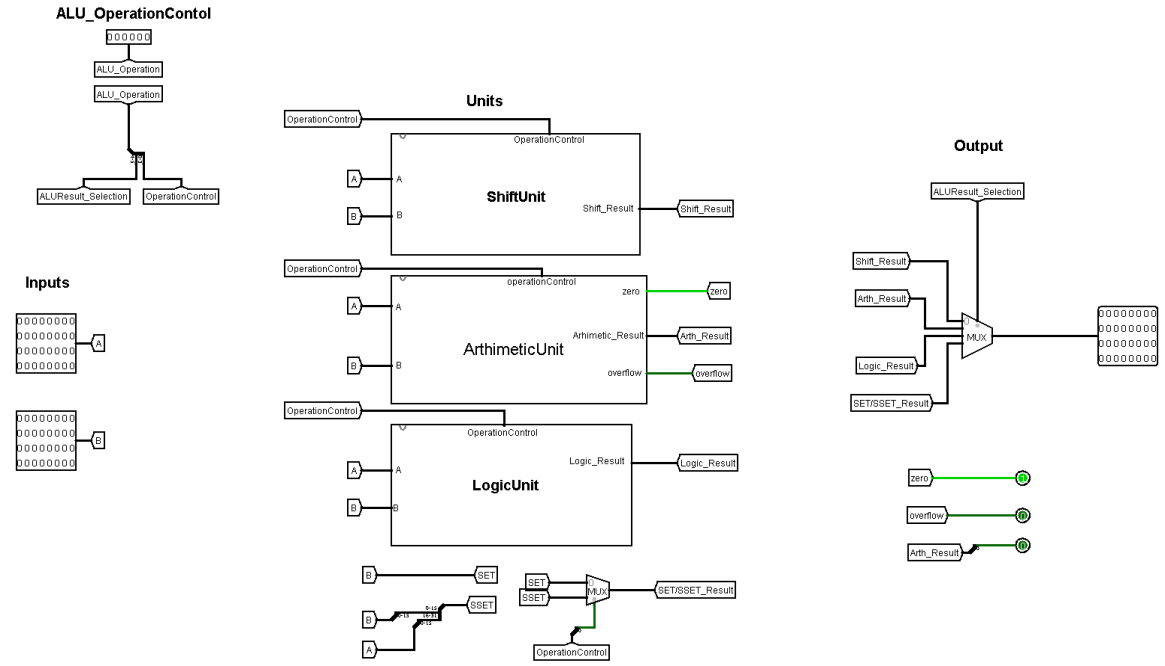
- Operand A (32 bits): sourced from register RS1 / Rd.
- Operand B (32 bits): Sourced either from register RS2 or an immediate value.

- 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:

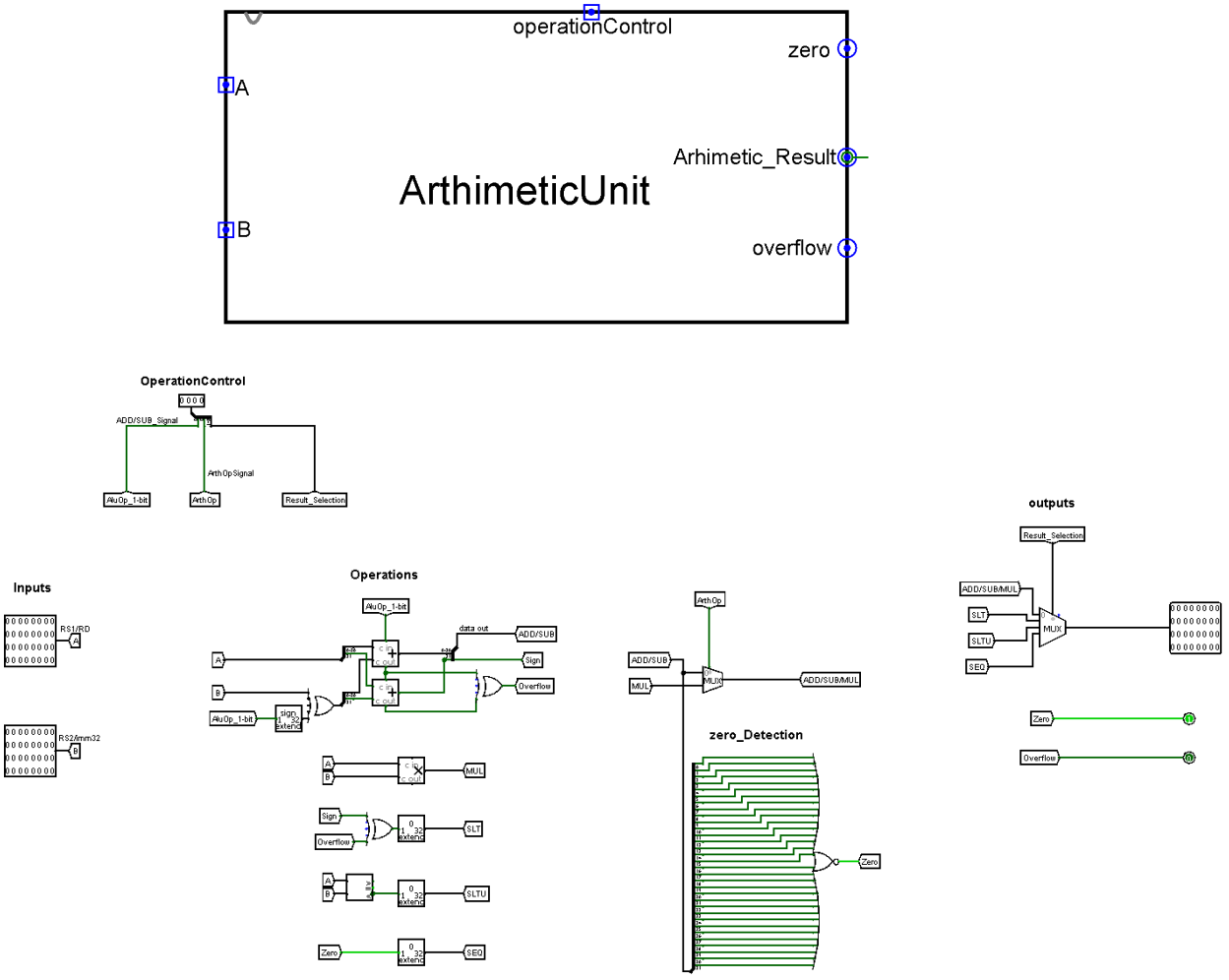
5.1 Arithmetic Unit

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).

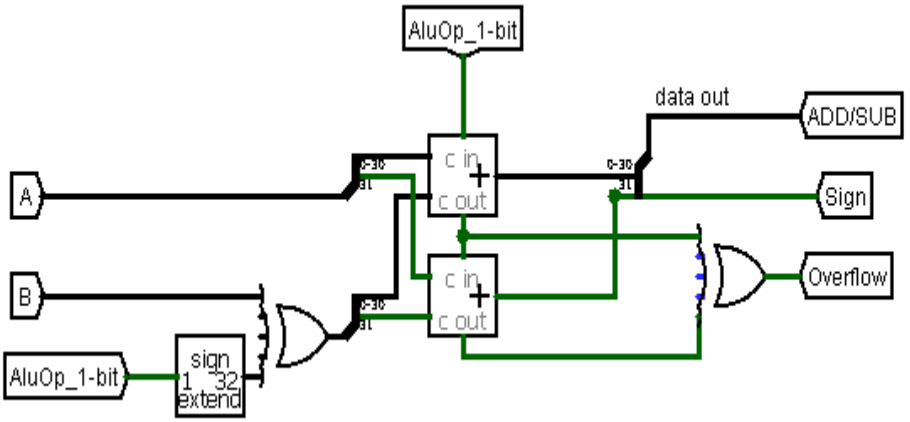
Each of these operations is selected based on the ALU Control Signals generated from instruction decoding.

5.1.3. Hardware Components

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

5.1.3.1 32-bit Adder/Subtractor

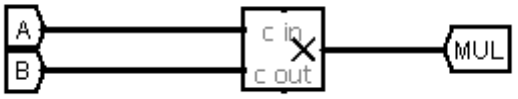
- Addition is performed using a 32-bit adder
- Subtraction is performed using the same adder by taking advantage of two's complement arithmetic:
- To subtract B from A (i.e., A - B), the unit inverts all bits of B and adds one (two's complement) before performing addition.
- A signal received from the 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

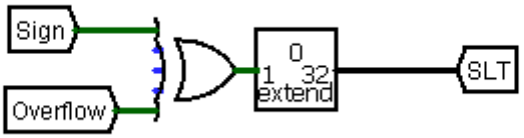
- The multiplier performs a signed integer multiplication of two 32-bit operands.

- Only the lower 32 bits of the 64-bit multiplication result are stored and used, consistent with the instruction set's definition (`MUL` operation).



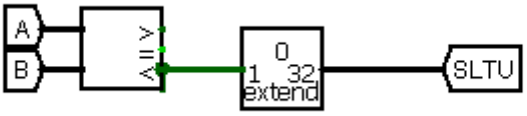
5.1.3.3 SetLessThan(SLT)

- 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



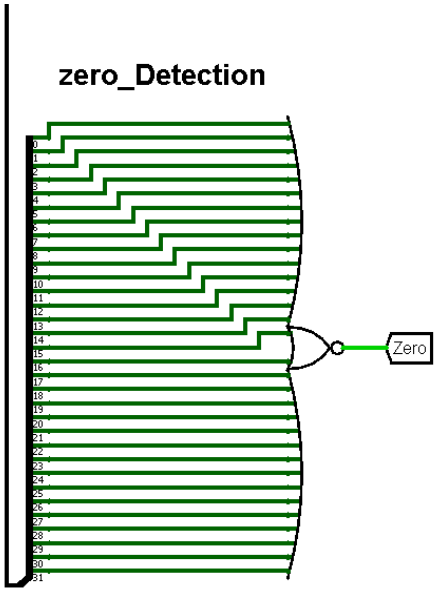
5.1.3.4 SetLessThanUnsigned(SLTU)

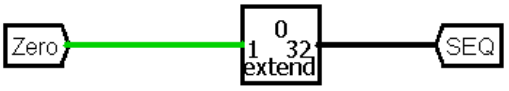
- The unit sets the output to 1 if the unsigned value of Operand A is less than the unsigned value of Operand B, utilizing a 32-bit comparator for evaluation.



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.

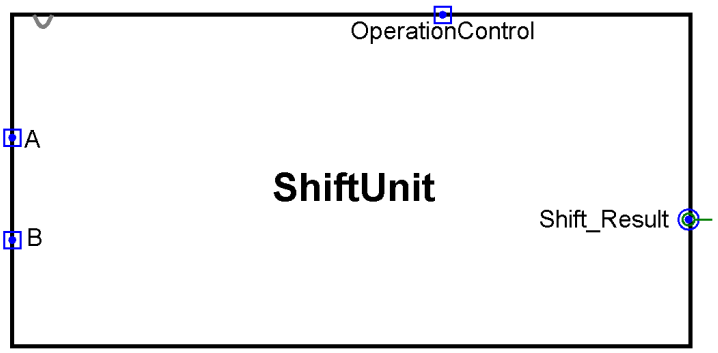


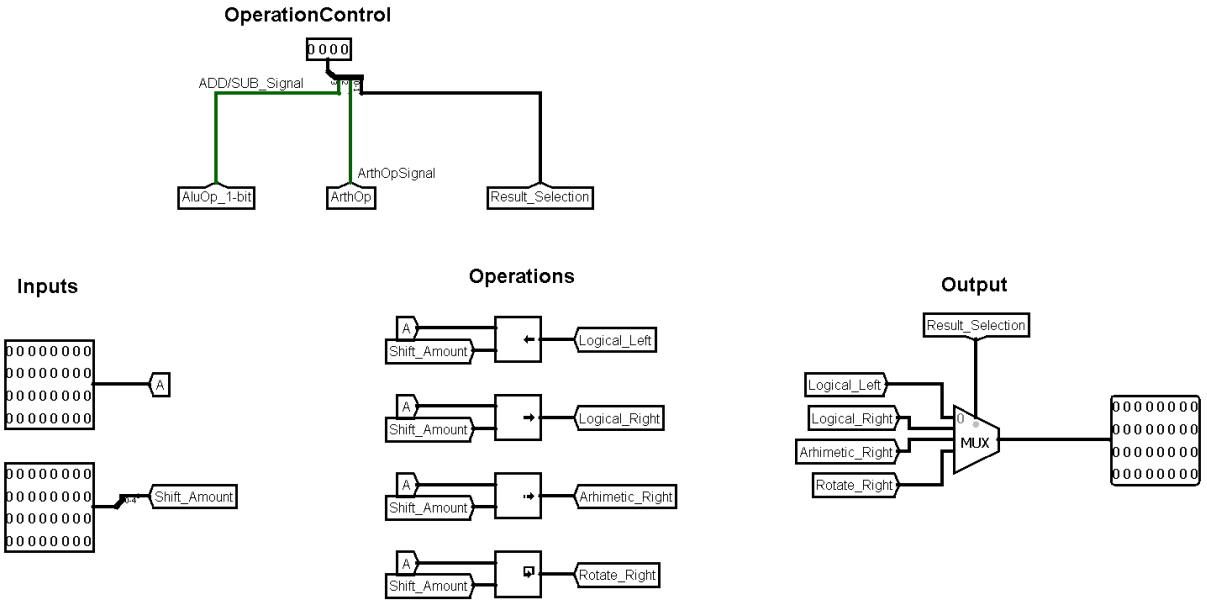


5.2 shift Unit

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

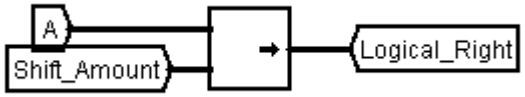
The Shift Unit hardware typically consists of the following subcomponents:

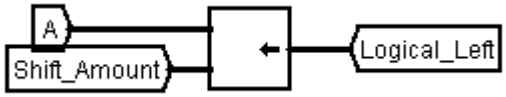
5.2.3.1 Logical Shifting

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.





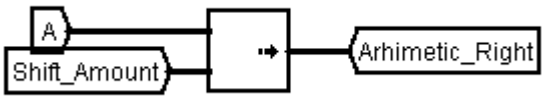
5.2.3.2 Arithmetic Shifting

- 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.

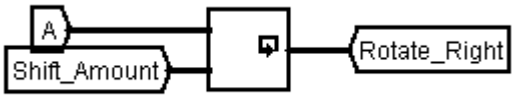
- Implemented using conditional logic that checks the original sign bit and propagates it during shifting.



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.

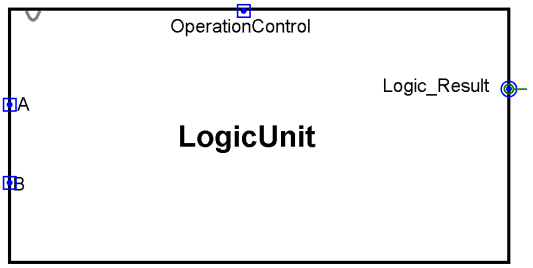


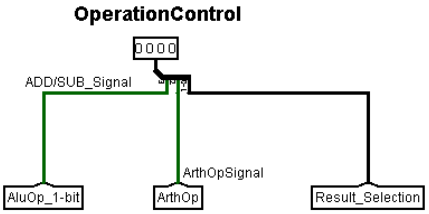
5.3 logical Unit

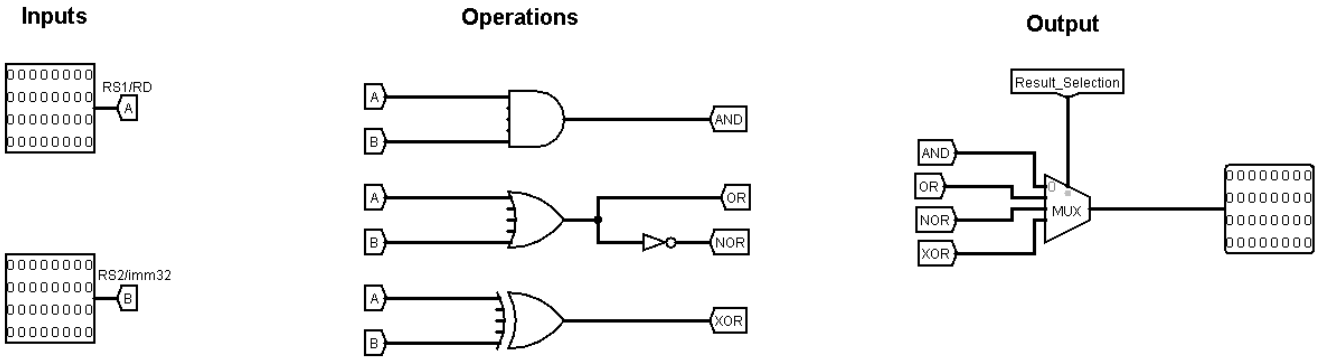
5.3.1. Introduction

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.







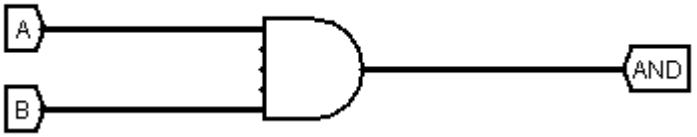
5.3.2. Functional Overview

The Logic Unit supports the following bitwise operations:

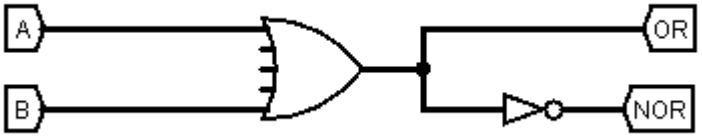
- AND (`AND`, `ANDI`): Performs a logical AND between each corresponding bit of two operands.
- OR (`OR`, `ORI`): Performs a logical OR between each corresponding bit.
- XOR (`XOR`, `XORI`): Performs a logical exclusive OR between corresponding bits.
- NOR (`NOR`, `NORI`): Performs a logical NOR, equivalent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

5.3.3. Hardware Components

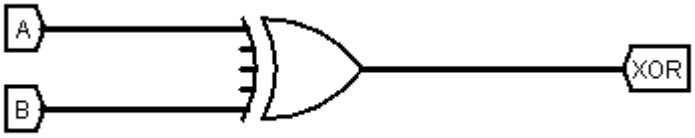
- **5.3.3.1 32-bit AND Gate : Performs a bitwise AND across all corresponding bits of the two operands.**



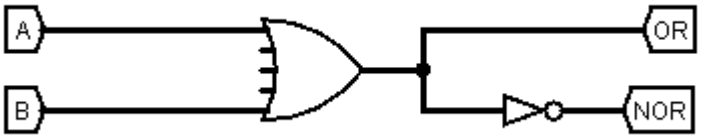
- **5.3.3.2 32-bit OR Gate : Performs a bitwise OR operation.**



- **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.

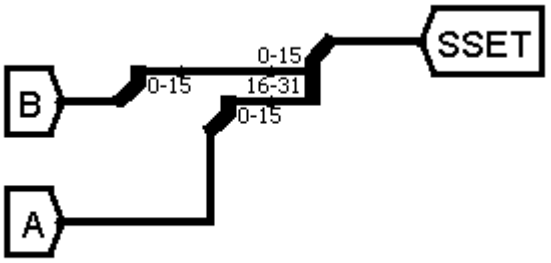
5.4.2. Functional Overview

- 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 AluOpCrtlSignals:

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

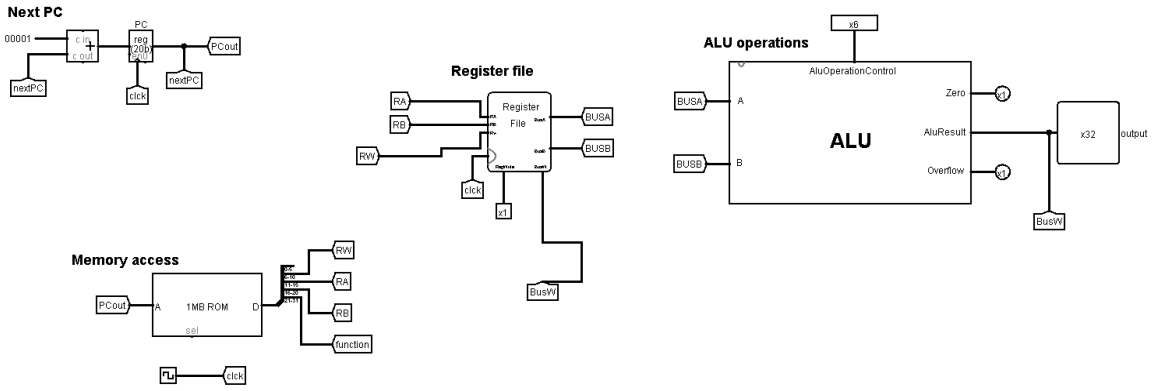
3.1 R-Type Data Path

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. NextPC = PC+1

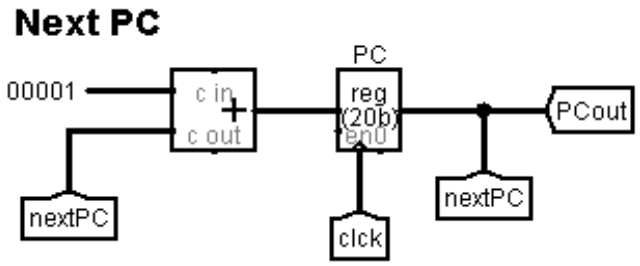


Figure 3.1.1

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

Read Ports: 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.

Write Port: 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.

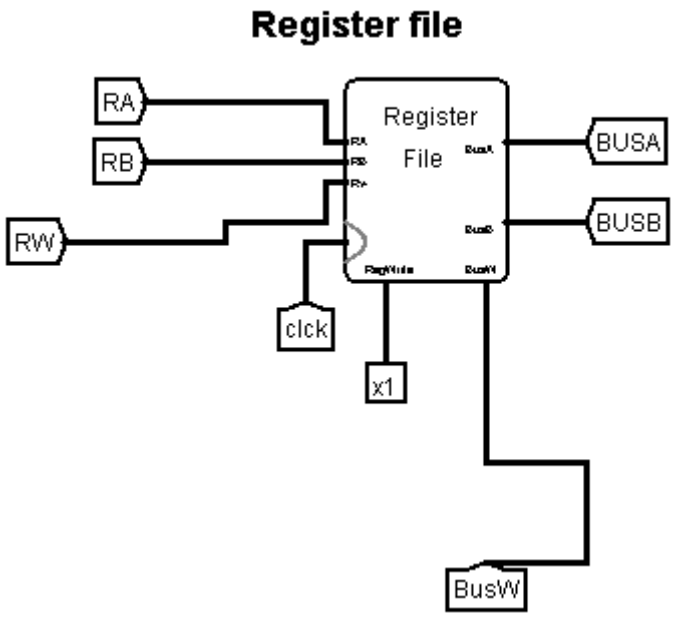


Figure 3.1.2

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)

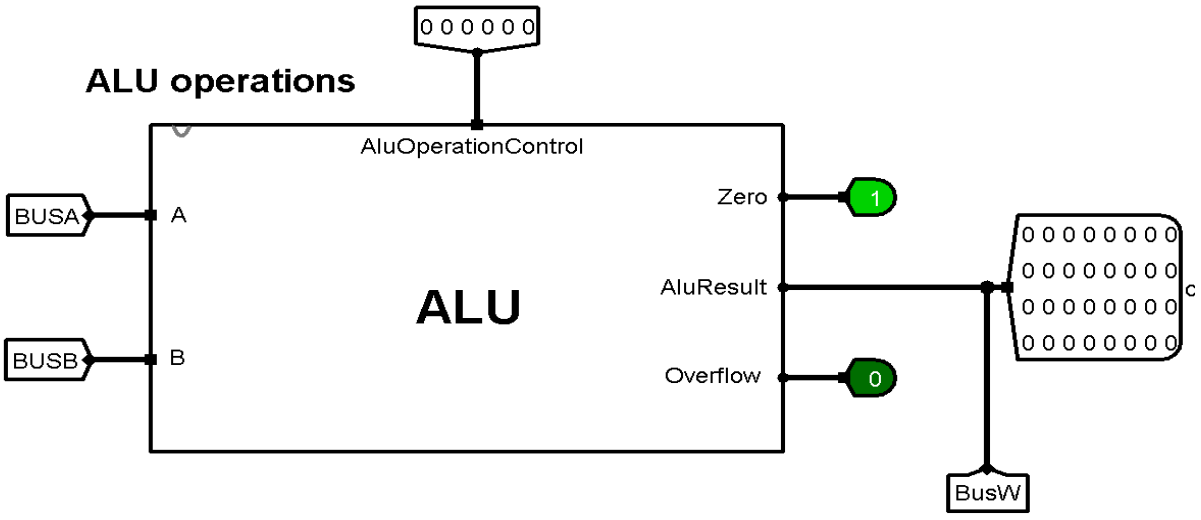


figure 3.1.3

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.

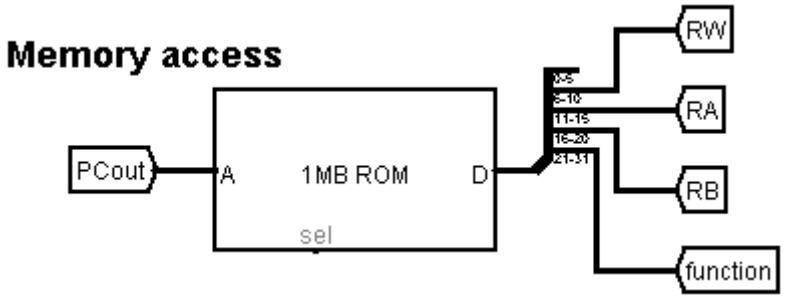


Figure 3.1.4

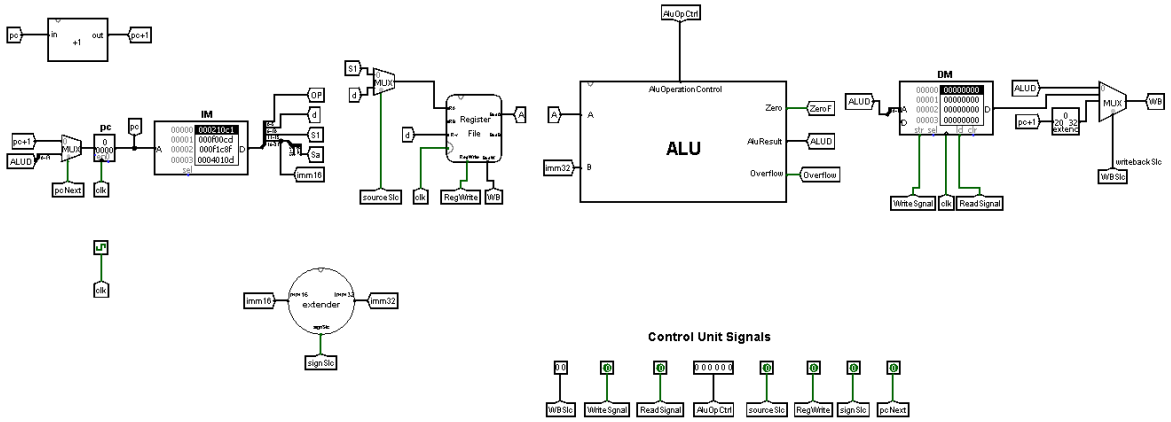
3.2 I-Type data path

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:

Imm 16	S_1^5	d^5	OP^6

- 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** :

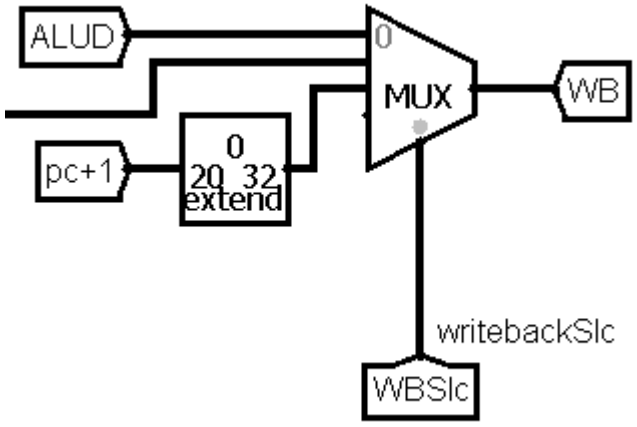
  For `LW` (Load Word), the ALU output is used as the memory address to read data from data memory.

- **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.

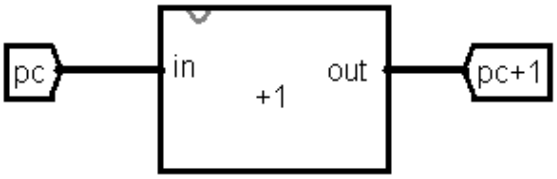


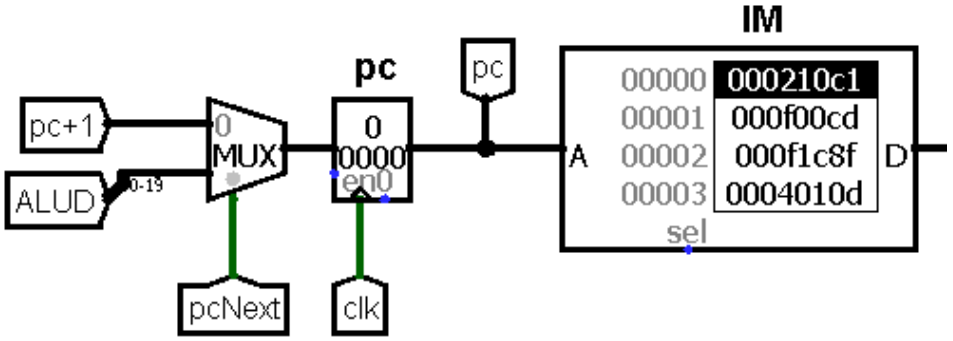
3.2.4. Hardware Design of the I-Type Datapath

The I-Type datapath requires the following hardware components:

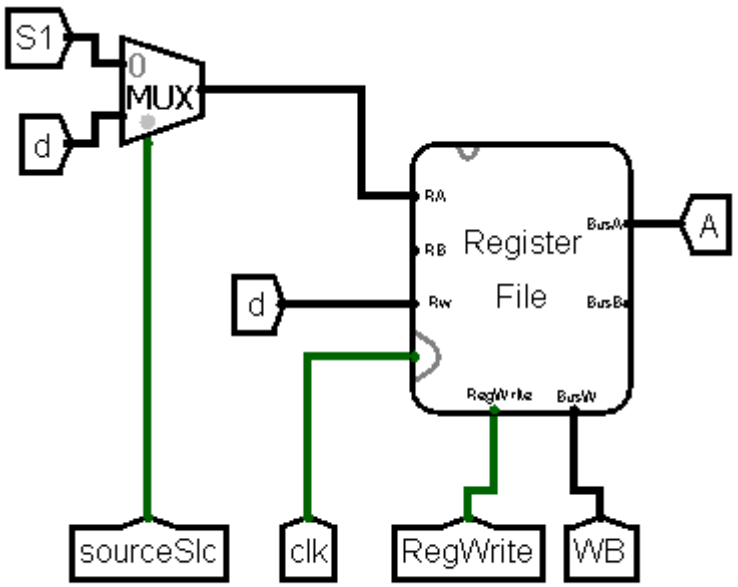
3.2.4.1 Main Components

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

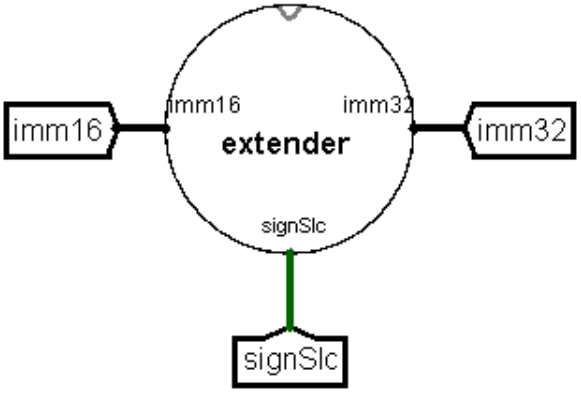


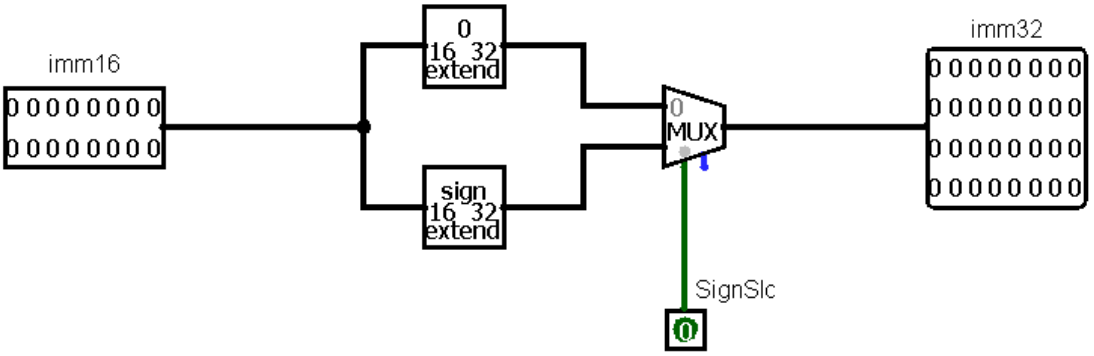


- **Instruction Memory:** Stores the instructions to be fetched.
- **Register File:**
    - Two read ports: for reading the `rs` (always used) and sometimes `rt` (for store/branch).
    - One write port: for writing back results to `rt`.

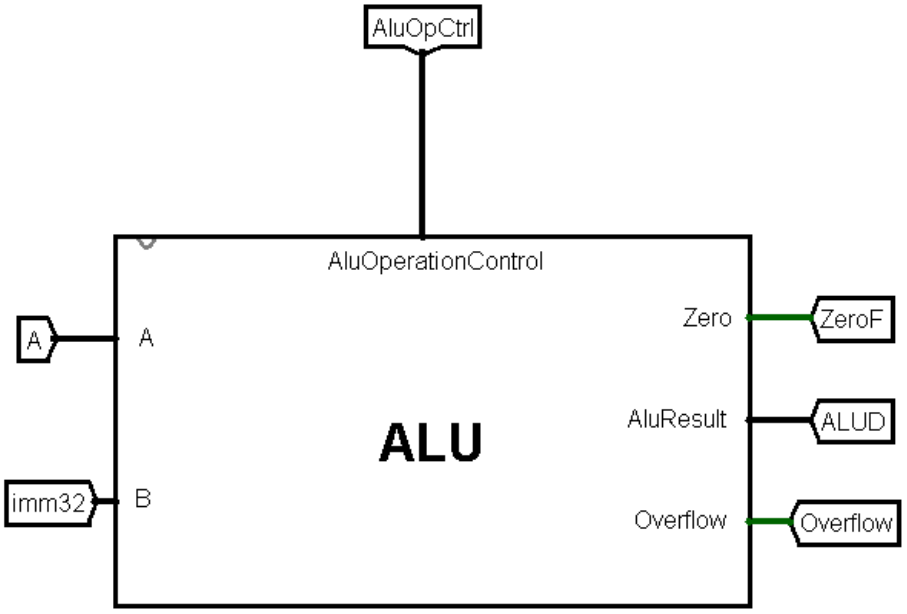


- **Immediate Extender:**
    - Extends the 16-bit immediate field to 32 bits.
    - Supports sign-extension and zero-extension based on instruction type.

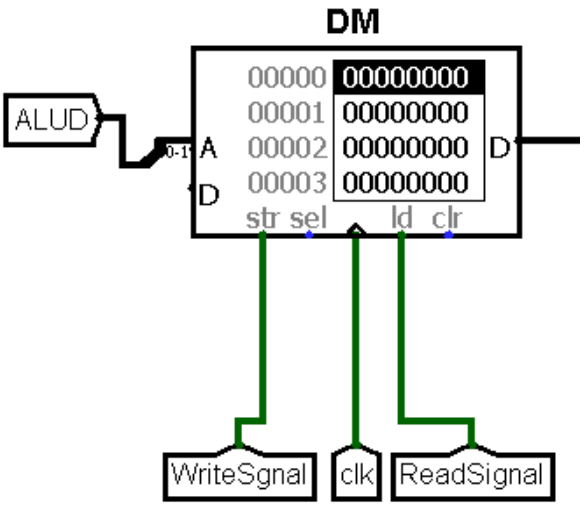




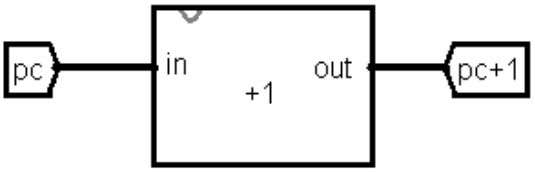
- **ALU:**
    - Performs arithmetic or logical operations.
    - Inputs: operand from `rs/rd` register and 32-bit extended immediate.

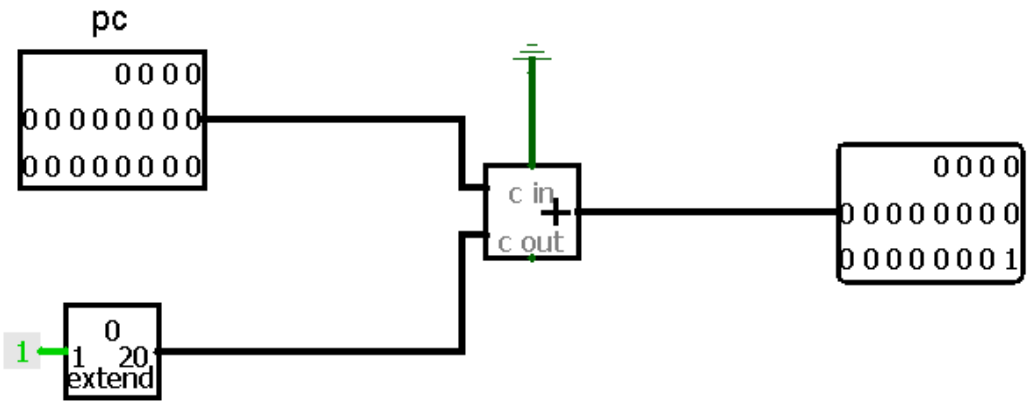


- **Data Memory:**
    - Accessed during `LW` instruction.
    - ALU result used as memory address.



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

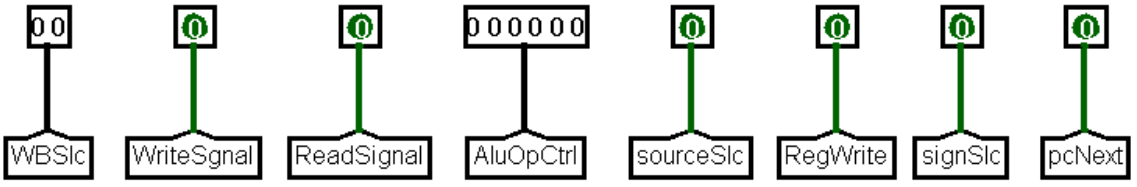




3.2.4.1 Control Signals

- **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 .

Control Unit Signals



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.

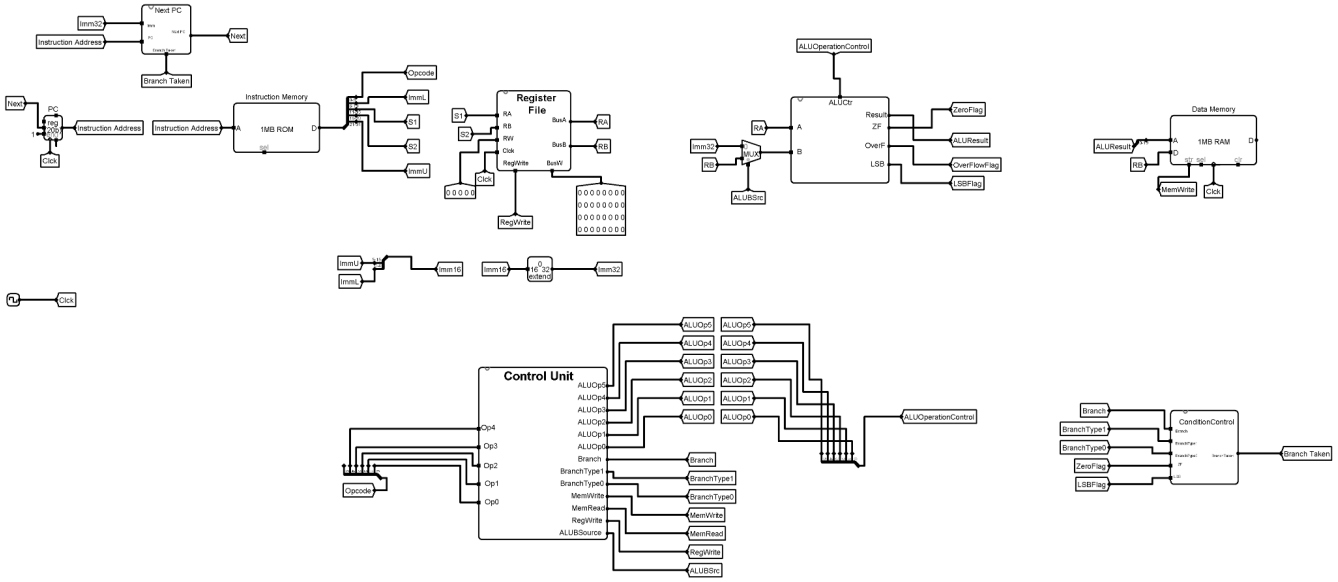


Figure 3.3.1

3.3.1. Instruction Encoding Structure

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

$\mathbf{ImmU^{11}}$	$\mathbf{S_2^5}$	$\mathbf{S_1^5}$	$\mathbf{ImmL^5}$	$\mathbf{OP^6}$

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 + 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.

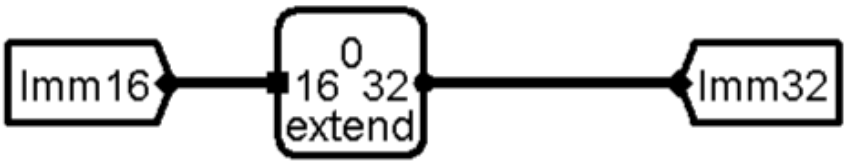


Figure 3.3.3

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

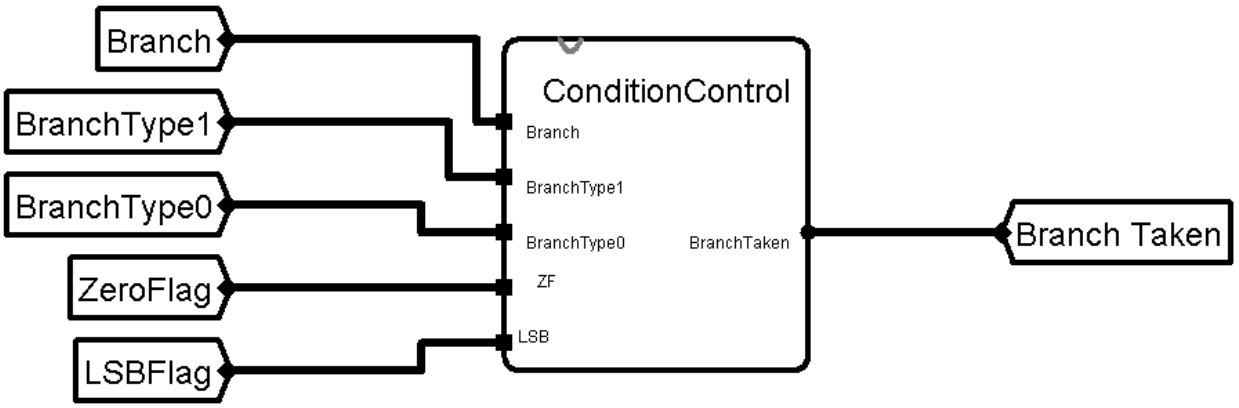


Figure 3.3.4

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

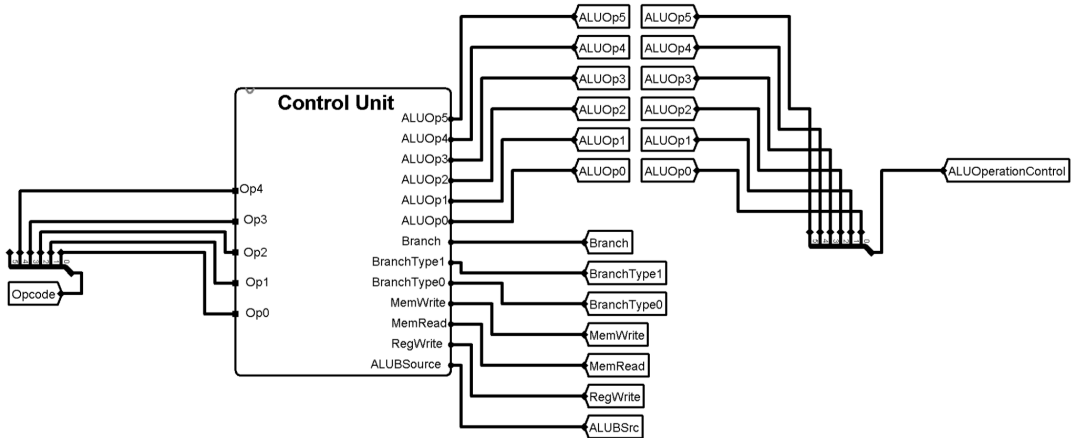


Figure 3.3.5

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 + sign_extend(Imm16)`), when a branch is taken based on a conditional evaluation.

Thus, the Next PC circuitshown 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.

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

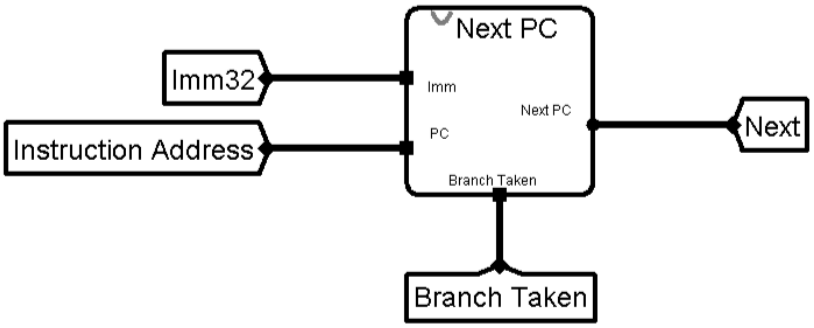


Figure 3.3.6

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.

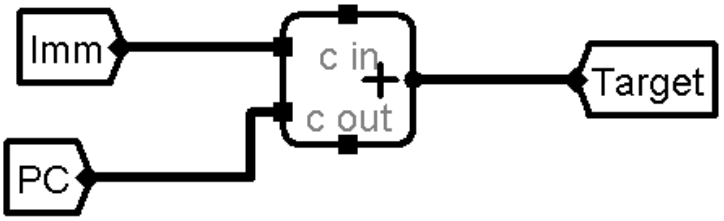


Figure 3.3.7

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

- **Adder for sequential execution**

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

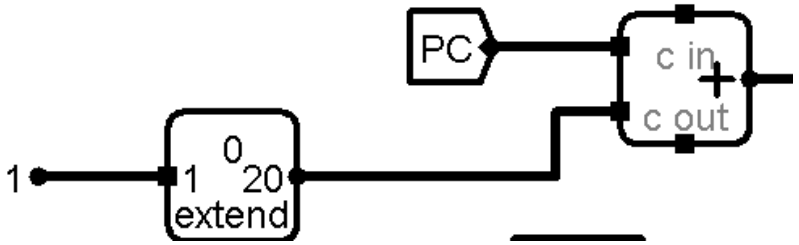


Figure 3.3.8

- **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.

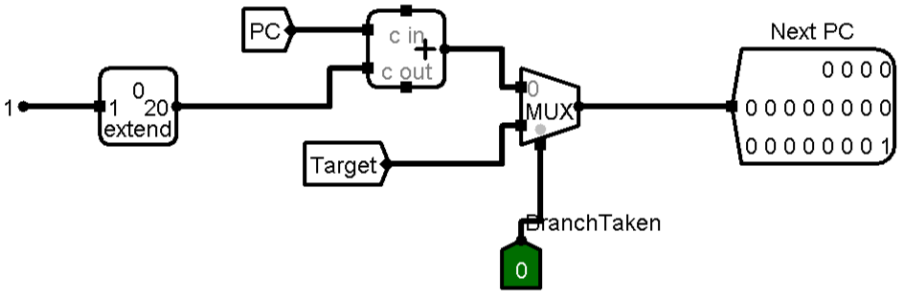


Figure 3.3.8

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.

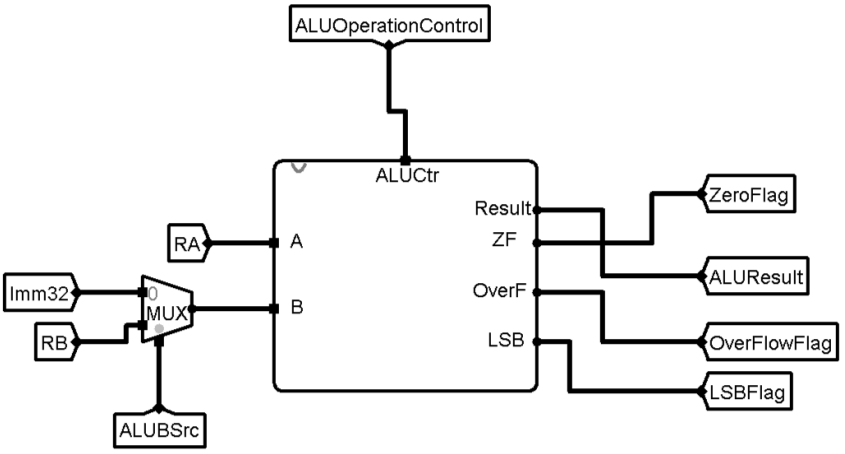


Figure 3.3.9

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.

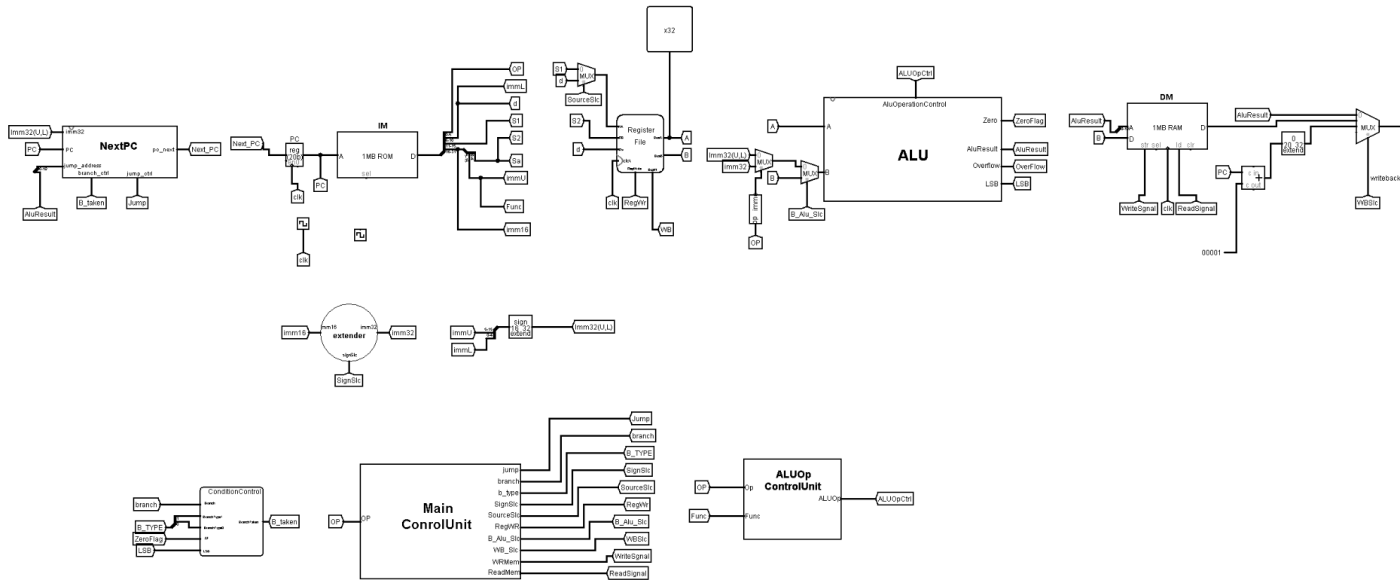


Figure 4.1

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 + sign-extended(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 + sign-extended({immU, 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.

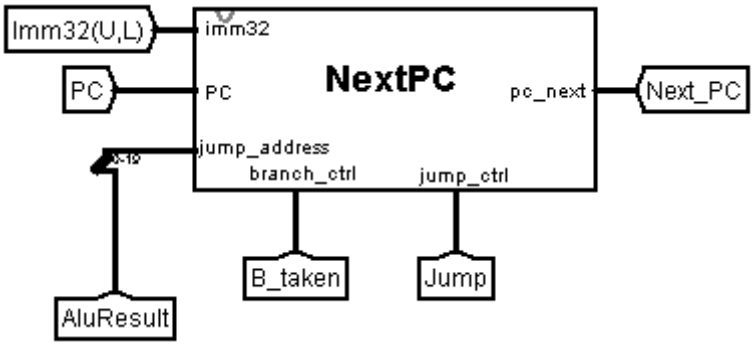


Figure 4.1

Jump Detection:

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

PC = RS1 + sign-extended(Imm16).

Branch Condition Evaluation:

If the instruction is a branch:

If the branch is taken:

PC = PC + sign-extended({immU, 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:

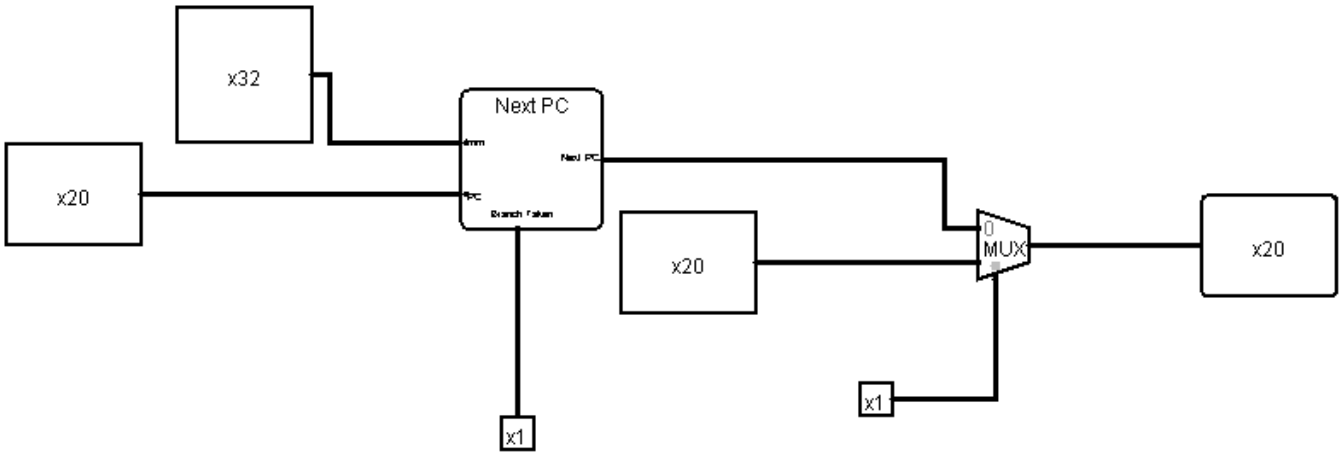
A multiplexer (MUX) is used to integrate all the operations into the circuit.

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.

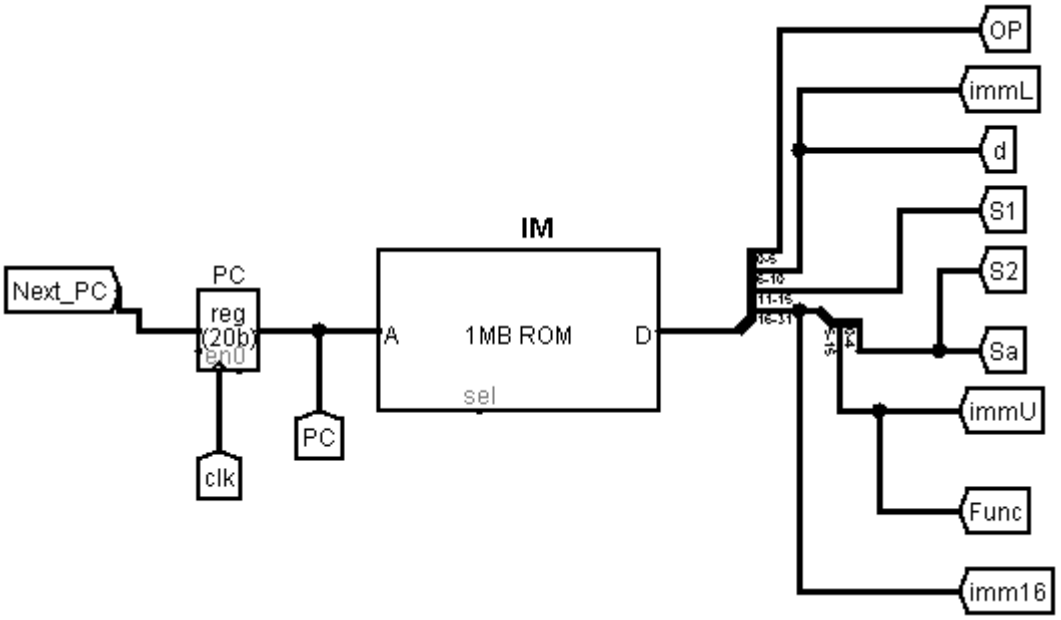


Figure 4.2

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.

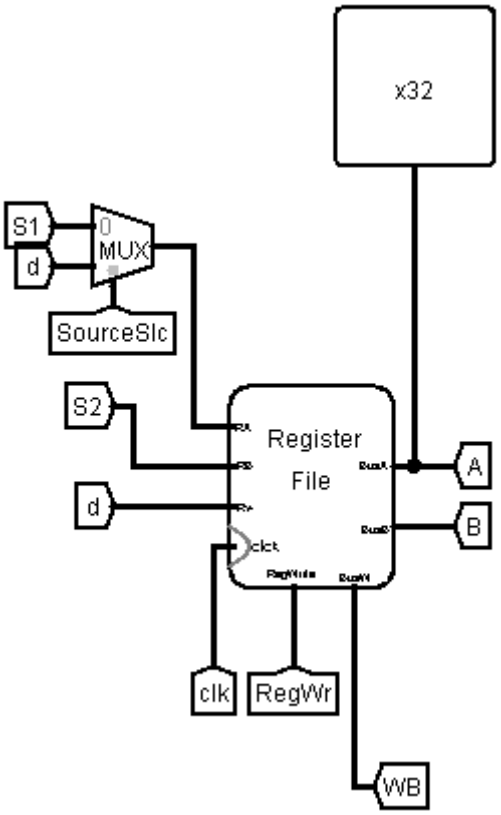


Figure 4.3

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*

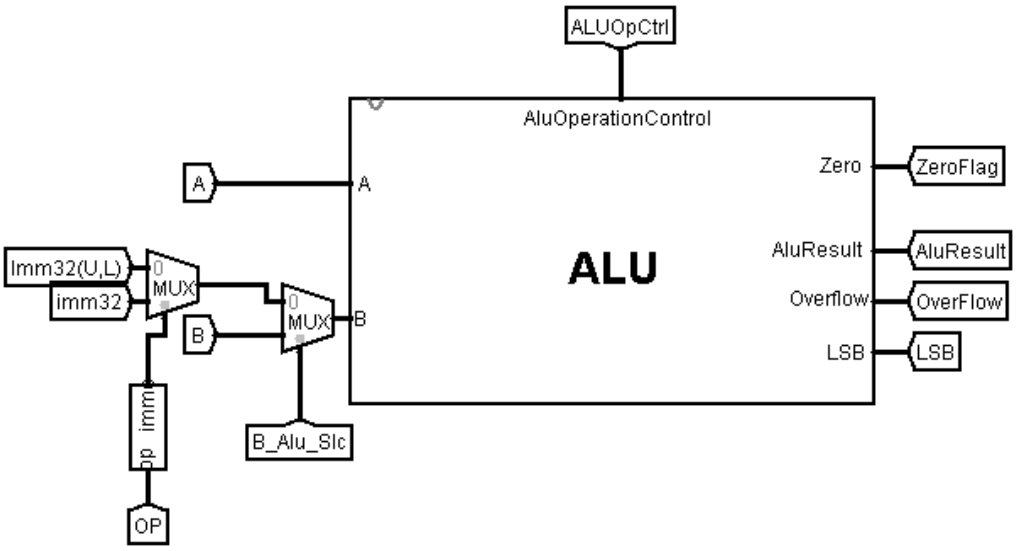


Figure 4.4

4.5 Data Memory

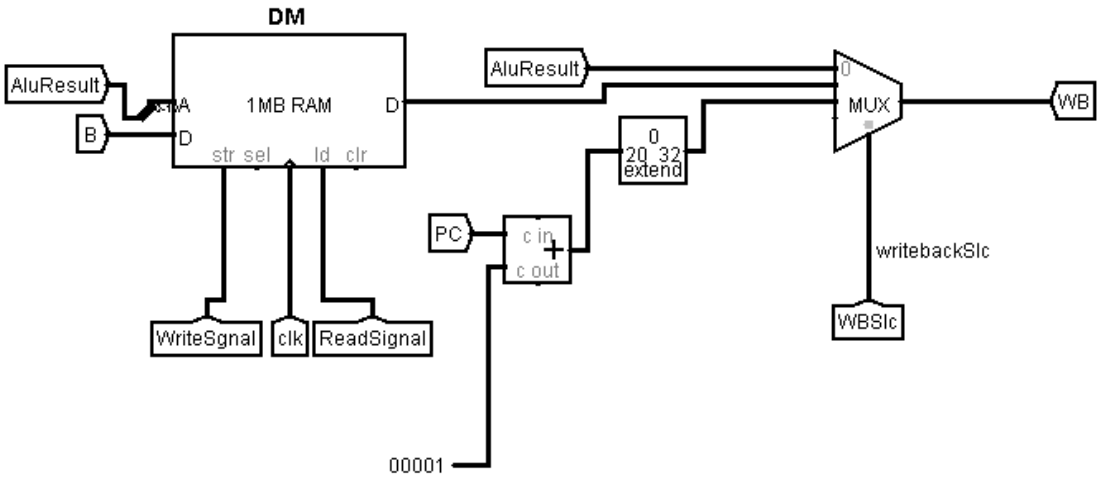
4.5.1 Input

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.


DM
AluResult
B
A
D
1MB RAM
D
str sel ld clr
WriteSgnal
clk
ReadSignal
AluResult
0
MUX
1
WB
0
20 32
extend
PC
c in
c out
writebackSlc
WBSlc
00001

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.

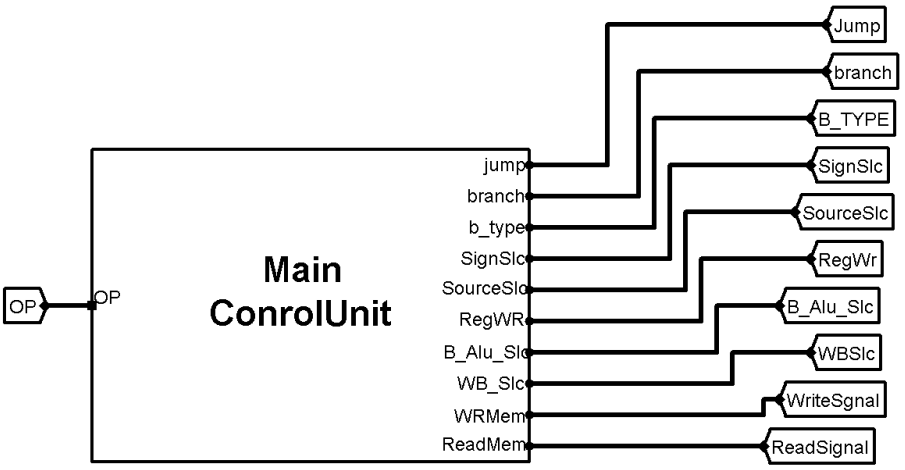


Figure 4.4.1

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 `MemRead = 0` and `MemWrite = 0`.

4. It disables control flow changes by setting `Branch = 0` and `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 Too**l:** was used to automate the generation and minimization of the truth table into a combinational circuit.

- The entire truth table was manually prepared first(Control Unit.xlsx), matching all instruction behaviors exactly.

- Minimization reduced the number of gates and logic complexity.

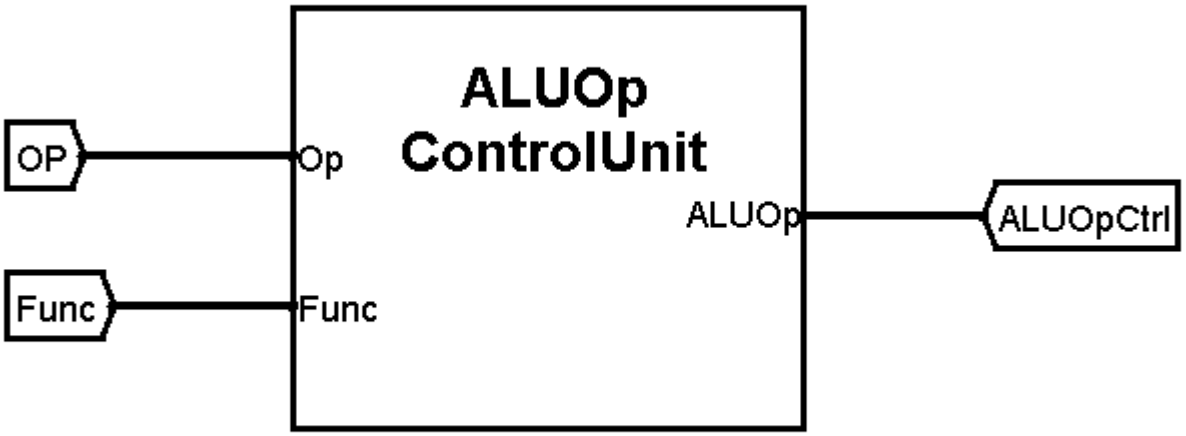
4.2Alu Control Unit

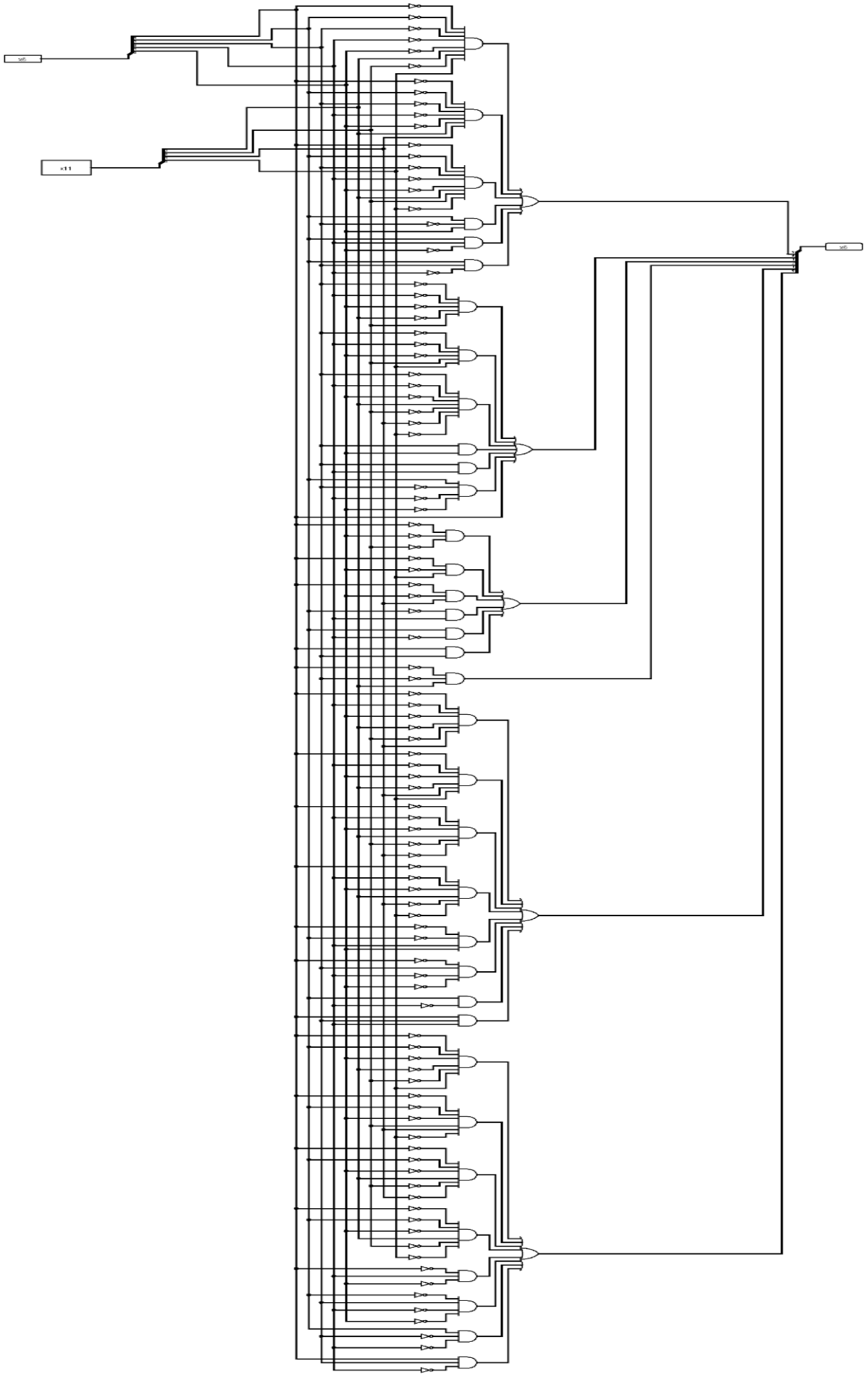
4.2.1. Introduction

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.




x6
x11
x6

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).

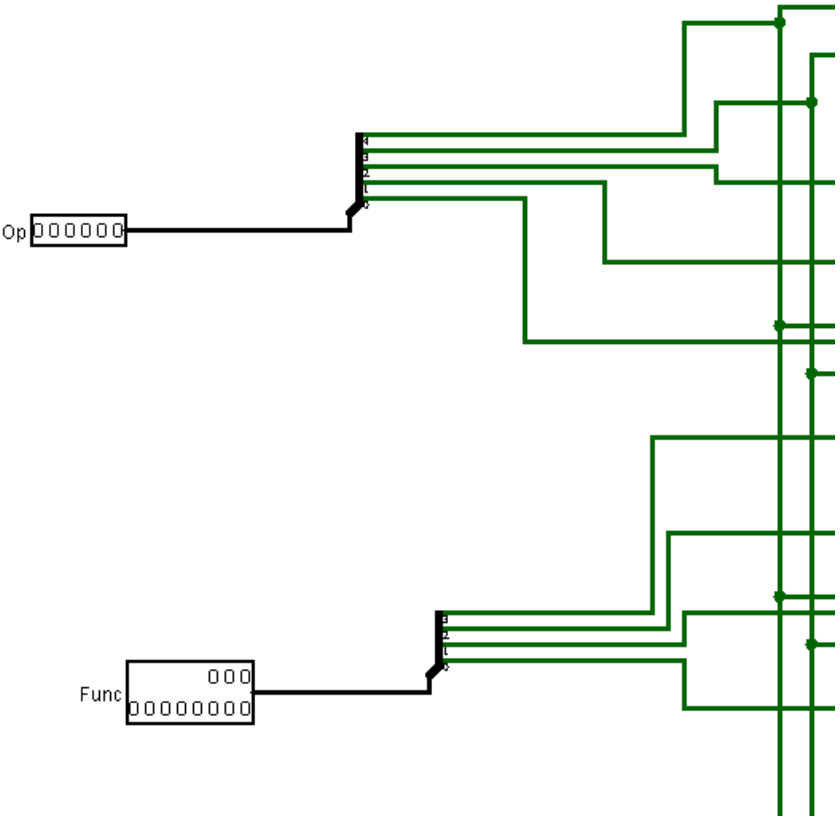
Based on these inputs, the ALU Control Unit produces an output:

- 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

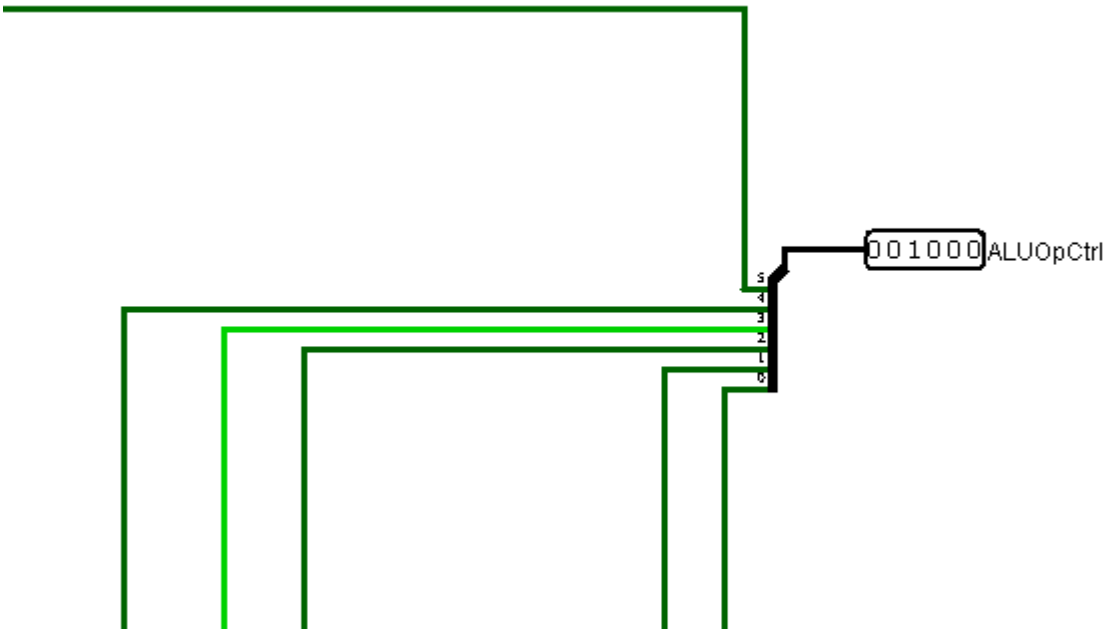
4.2.3.1 Inputs:

- **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.

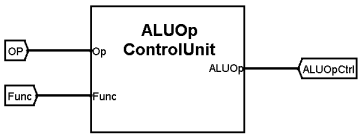


4.2.3.3 Hardware Structure

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-Slc(2 bit)	AluOp(1 bit)	ArthOp(1 bit)	Result-Slc(2 bit)
0	0	SLL	0	x	x	0
0	1	SRL	0	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 `Branch = 0`, the unit forces `BranchTaken = 0` (the instruction is not a branch).
- For `Branch = 1`:
    - BranchType = 00 (BEQ): Branch if `ZeroFlag = 1`.
    - BranchType = 01 (BNE): Branch if `ZeroFlag = 0`.
    - BranchType = 10 (BLT/BLTU): Branch if `LSB = 1`.

- BranchType = 11 (BGE/BGEU): Branch if `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.

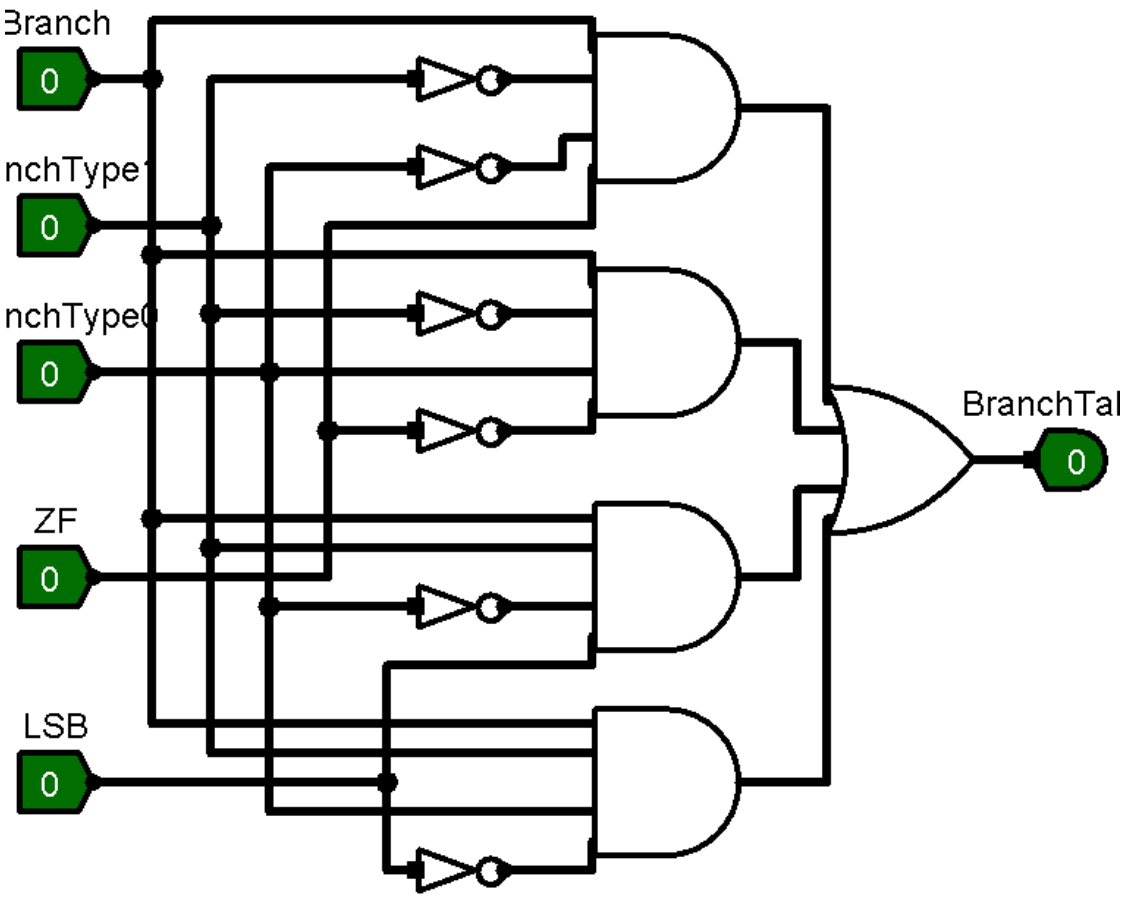


Figure 5.1.1

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/IeUk_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

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

```python
@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/

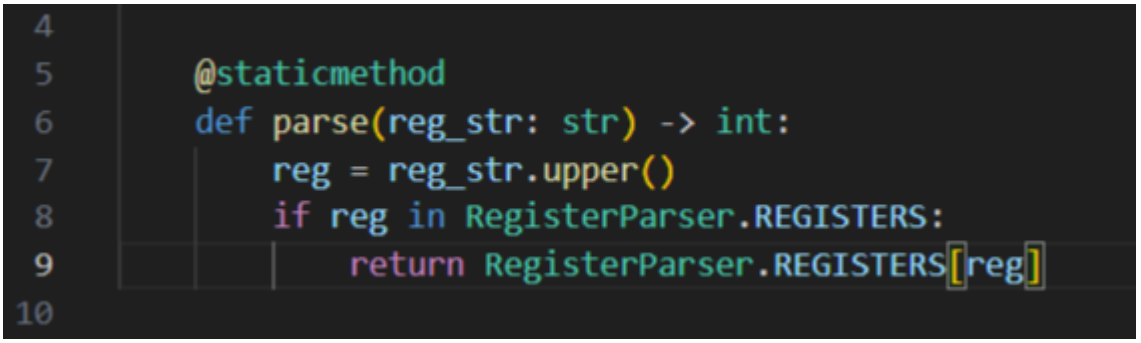
├── __init__.py

├── imm_parser.py

├── instruction_parser.py

├── line_parser.py

└── register_parser.py



```
        mem_match = re.match(r'^(-?\d+)\((\w+)\)$', 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

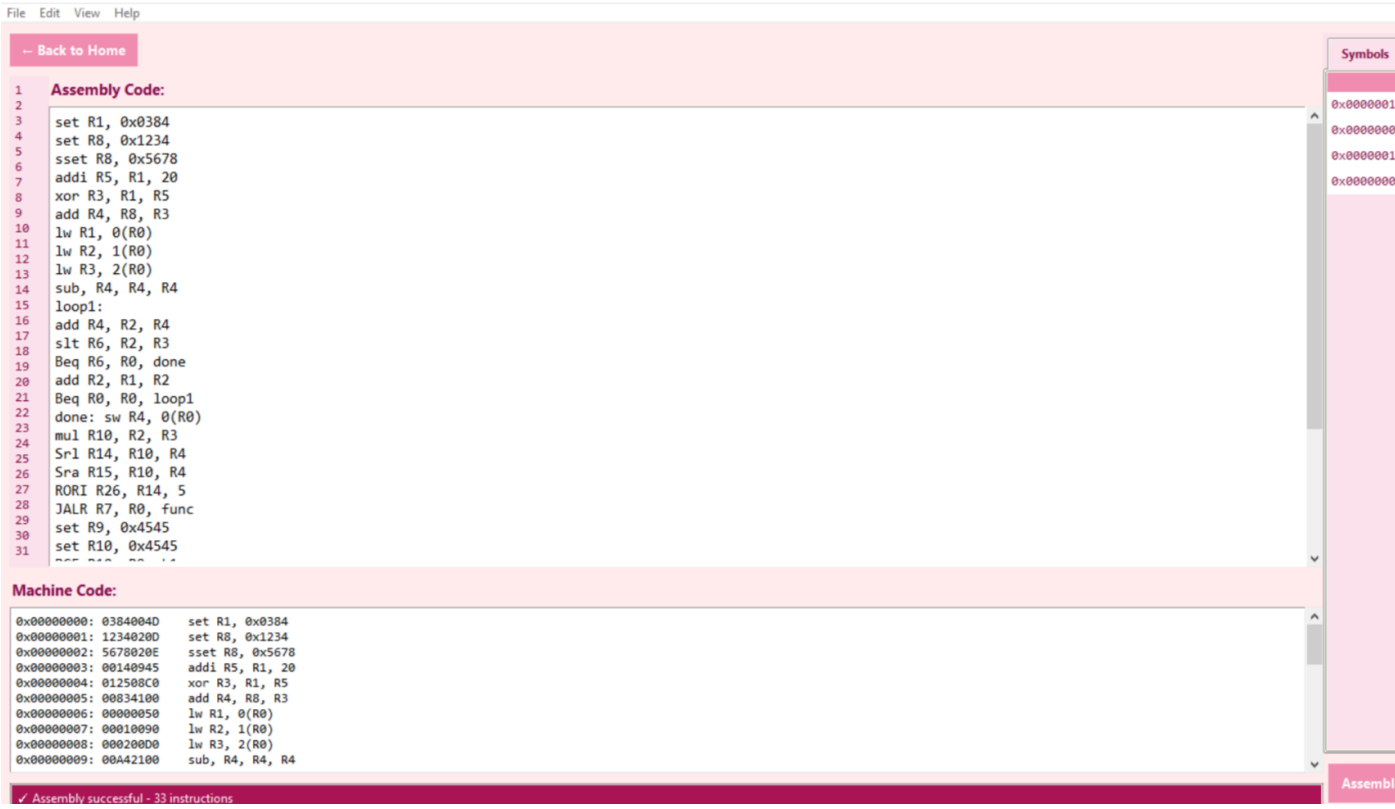
```
def encode(self, instruction, symbol_table, current_pc):
    f = self.FUNCTIONS[instruction.opcode]
    d = RegisterParser.parse(instruction.operands[0])
    s1 = RegisterParser.parse(instruction.operands[1])
    s2 = RegisterParser.parse(instruction.operands[2])
    return (f << 21) | (s2 << 16)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

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.

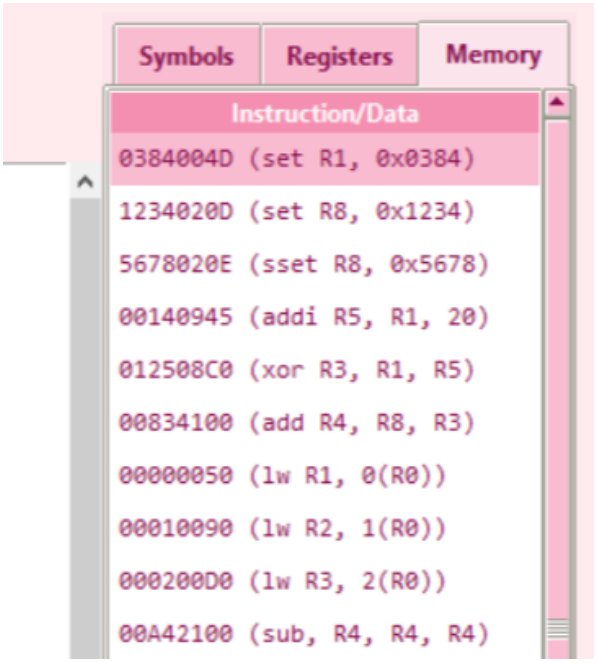
```
#shifting
        SET R21, 1
        SET R22, 2
        SET R1, 1
        SET R2, 2

        #r_type
        SLL R23, R21, R22
        SLL R3, R1, R2
        SRL R24, R21, R22
        SRA R25, R21, R22
        ROR R26, R21, R22

        #i_type
        SLLI R27, R21, 3
        SRLI R28, R21, 1
        SRAI R29, R21, 1
        RORI R30, R21, 1
```

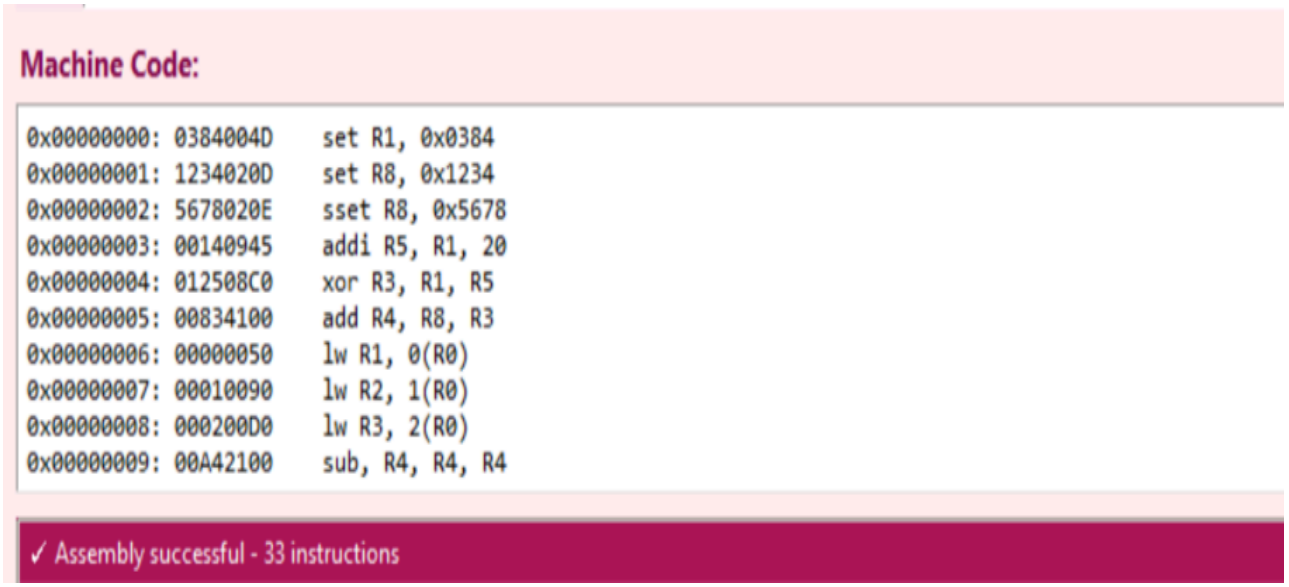
6.2.3 Memory Registers and Labels

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.



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.



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

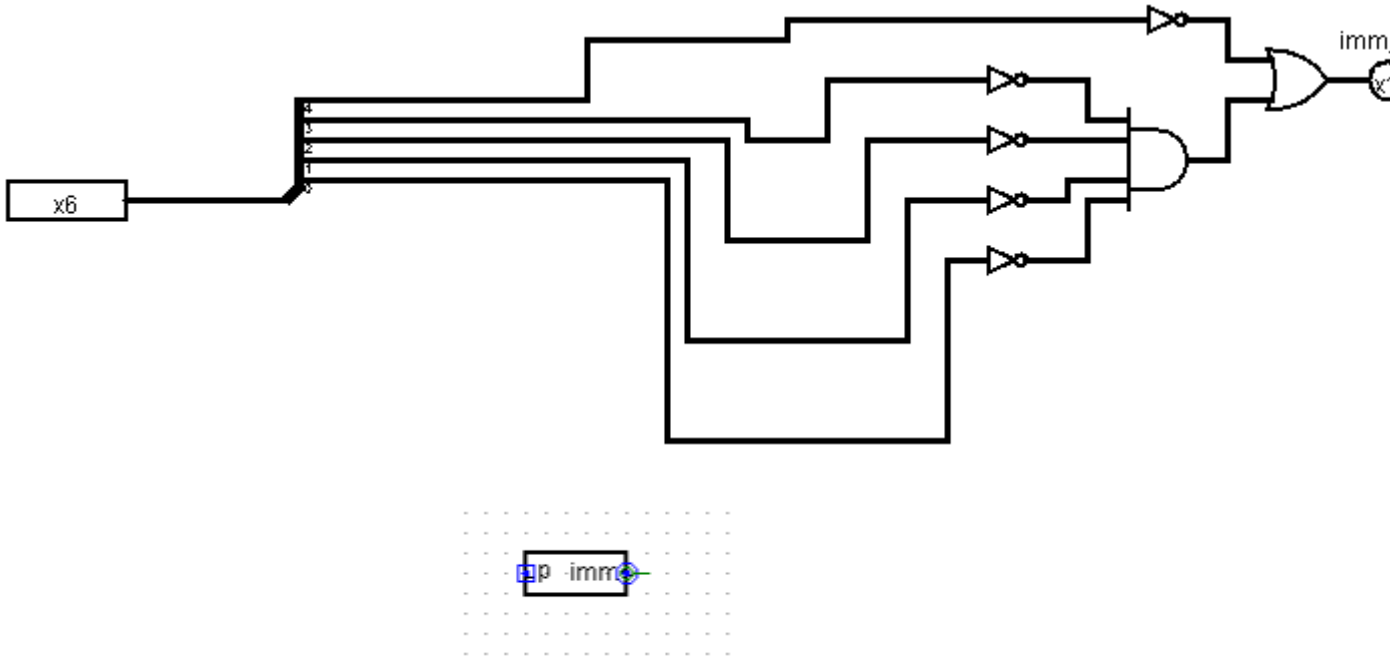
7.1.1.1 Problem

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	- Making The General DataPath - Construct the control unit - Make ALUOP Control Unit

				- Enhancing Assembler
25/4/2025	1:30 PM	9:00PM	Offline	- Testing DataPath, controlUnit - Test the code required from us.
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**
- ALU - I Type DataPath - ControlUnit	- Software Implementation - R Type DataPath	- RegisterFile - SB Type DataPath - ControlUnit

- ALUOP Unit - Full DataPath - Hardware Testing - Documentation - Video	- Full DataPath - Hardware Testing - Test Codes - Documentation - Video	- ALUOP Unit - Full DataPath - Hardware Testing - Documentation - Video

Pipelined Version

1.Introduction

Upon the successful completion of Phase 1 of our single-cycle processor project, we had implemented all the fundamental components necessary for basic instruction execution. These components included the register file, arithmetic logic unit (ALU), and the distinct data paths required to support various instruction formats, specifically R-type, I-type, and SB-type instructions. Following the individual implementation of these elements, we proceeded to construct a comprehensive and unified data path. This unified architecture was further integrated with the main control unit, the ALU operation control module, the PC update (PC-next) circuitry, and other essential control logic needed for seamless instruction execution.

With the single-cycle processor fully functional, we then transitioned to the next major phase of the project: the design and implementation of a pipelined processor. In this phase, we focused on introducing the core features of pipelining to enhance instruction throughput and improve overall performance. A significant part of this stage involved the accurate handling of pipeline hazards. We implemented robust mechanisms for managing both data hazards and control hazards, which included stall and kill units to ensure correct instruction flow and avoid incorrect computations.

As a final enhancement, we developed and integrated a 2-bit branch predictor to improve the efficiency of branch instruction handling. This predictor was carefully tested using the assembler to ensure its correctness and reliability across different branching scenarios. Overall, the project progressed from a basic single-cycle design to a fully pipelined processor with advanced control and prediction capabilities.

The primary advantage of pipelining lies in its ability to reduce the average instruction execution time. Although each instruction still requires multiple cycles to complete, the processor can begin executing a new instruction in every cycle once the pipeline is filled. As a result, pipelining maximizes hardware utilization and improves processor efficiency, making it a critical design choice in high-performance computing systems.

Moreover, pipelining enables better scalability and provides a structured framework for implementing performance enhancements such as hazard detection, forwarding, and branch prediction. These features help maintain correctness while allowing the pipeline to operate at high throughput, thus balancing complexity and efficiency in modern architectures.

Let us now dive into the study of the pipelined processor architecture!

2. Pipeline Implementation

2.1 Pipeline Registers

2.1.1 Pipeline Registers in the Fetch Stage

The first stage of the pipelined processor is the Instruction Fetch (IF) stage, where the processor retrieves the instruction from memory based on the address held in the Program Counter (PC). Once the instruction is fetched, it is stored in the IF/ID pipeline register, which acts as a buffer to transfer the instruction and relevant data to the next stage of the pipeline in the subsequent clock cycle.

At this stage, the PC is incremented by 1 to point to the next sequential instruction, assuming a 32-bit instruction width and byte-addressable memory. This incremented value is written back into the PC to prepare the processor for the next fetch cycle. Additionally, the incremented PC is saved in the IF/ID register, as it may be required later, particularly for instructions such as branch-equal (`BEQ`) that depend on the PC value for calculating the target address.

The IF/ID register also holds the fetched instruction, effectively functioning as the Instruction Register (IR) as shown in figure 2.1.1.1 within the pipelined architecture. Since the instruction type is not yet known during the fetch stage, the processor must assume that any type of instruction could follow. Therefore, it passes all necessary information—specifically, the instruction bits and the updated PC—down the pipeline to be decoded and executed appropriately in the following stages.

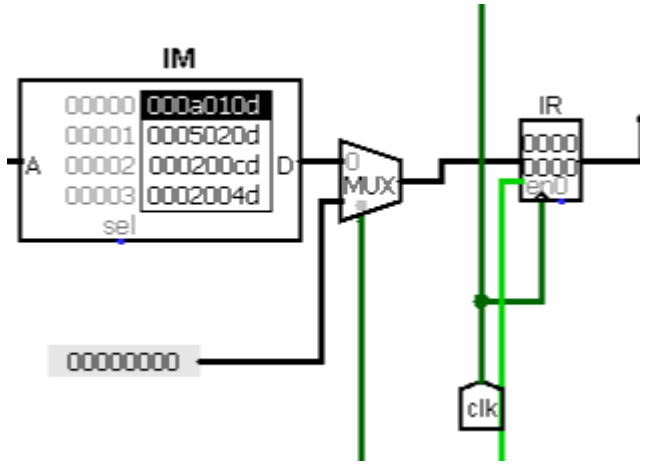


Figure 2.1.1.1 “IR Pipelined Register”

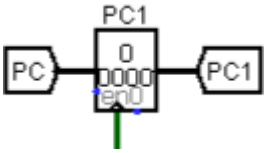


Figure 2.1.1.2 “transforming PC value”

2.1.2 Pipeline Registers in the Decode Stage

After the instruction has been fetched and placed in the IF/ID register, the next step in the pipeline is the Instruction Decode (ID) stage. In this phase, the processor interprets the fetched instruction by extracting relevant fields, such as register operands and immediate values. Specifically, the 16-bit immediate field from the instruction is sign-extended to 32 bits, enabling compatibility with 32-bit operations. Simultaneously, the instruction's register fields are used to identify and read two source operands from the register file.

All the values retrieved during the decode stage —namely, the two read registers (often referred to as A and B before forwarding), the sign-extended immediate value, and the incremented PC _ are stored in the ID/EX pipeline register as shown in figure 2.1.2.1 . This ensures that all potentially required data is preserved for use in the subsequent execution stage, regardless of the instruction type.

At this point, the processor does not yet determine the exact instruction type. Therefore, it extracts and stores all the necessary components that may be needed later, depending on the instruction. This includes the original incremented PC (PC1), the read values from the source registers (A and B), the 32-bit signed or unsigned immediate values derived from I-type and SB-type formats, as well as the destination register number and operation code. This comprehensive transfer of information allows for seamless and efficient instruction execution in the remaining pipeline stages.

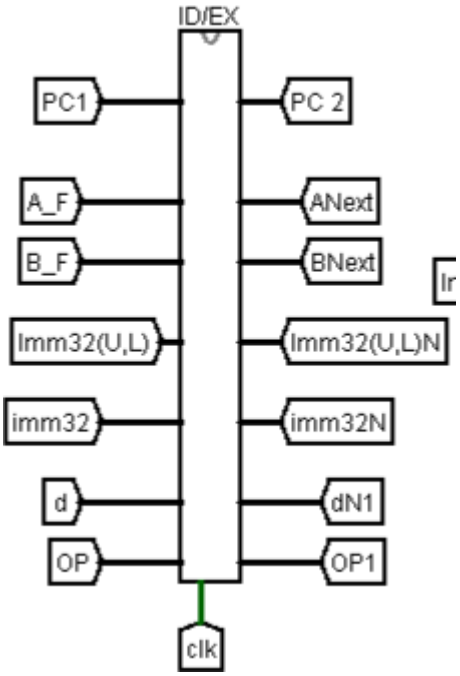


Figure 2.1.2.1 ID/EX pipelined Register

2.1.3 Pipeline Registers in the Execute Stage

In the Execute (EX) stage, the processor performs the necessary arithmetic or logical operations as dictated by the decoded instruction. At this point, the values of the operands, which were passed down from the previous stage, are used in the execution of the instruction. The Arithmetic Logic Unit (ALU) takes the values of the source registers (A and B) or the immediate value depending on the instruction type and performs the required operation, whether it be an addition, subtraction, bitwise operation, or any instruction we have in the ISA.

In addition to the ALU result, the destination register number and the value of operand B are crucial pieces of information that must be passed to the next stage. These values are stored in the EX/MEM pipeline register as shown in Figure 2.1.3.1, ensuring that they are available for the memory access stage or the write-back stage as needed. The ALU result is particularly important, as it will be used in the Memory Access stage for load or store operations, or it may be written back to the register file during the Write-Back stage.

For example, in a store instruction like `sw`, the value of the operand B is used to calculate the address, and the data to be stored is taken from operand B. Similarly, for an arithmetic operation, the ALU result will be transferred to the MEM stage for potential storage in memory or to the WB stage for writing back to the register file. This transfer of critical data ensures that the processor can continue the instruction execution seamlessly through the subsequent stages.

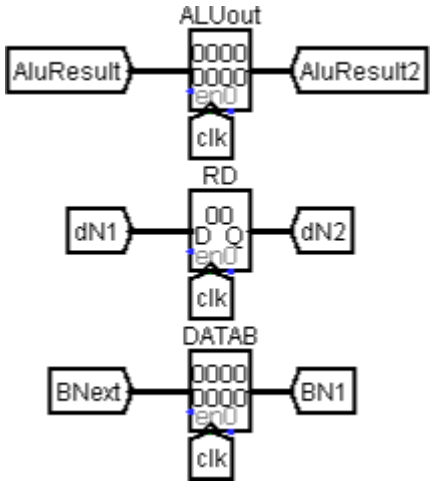


Figure 2.1.3.1 EX/MEM pipeline register

2.1.4 Pipeline Registers in the MEM & WB Stage

In the Memory Access stage, the processor performs data memory operations based on the instruction type. For load instructions (such as lw), the ALU result from the previous stage is used as the memory address from which data is read. For store instructions (such as sw), the same ALU result serves as the target memory address, while the data to be written is taken from operand B. These values—whether the memory read data or the ALU result—are then stored in the MEM/WB pipeline register as shown in Figure 2.1.4.1, along with the destination register number.

In the final stage, Write-Back the processor determines whether to write the memory data in instructions like load or the ALU result back into the destination register in the register file. This completes the instruction's execution, ensuring that the correct value is stored in the appropriate register. The seamless transition of data between the MEM and WB stages, supported by the MEM/WB pipeline register, is essential to maintaining data integrity and enabling the parallelism that the pipelined architecture is designed to exploit. So we save the value of the destination register and the write back data that will be written.

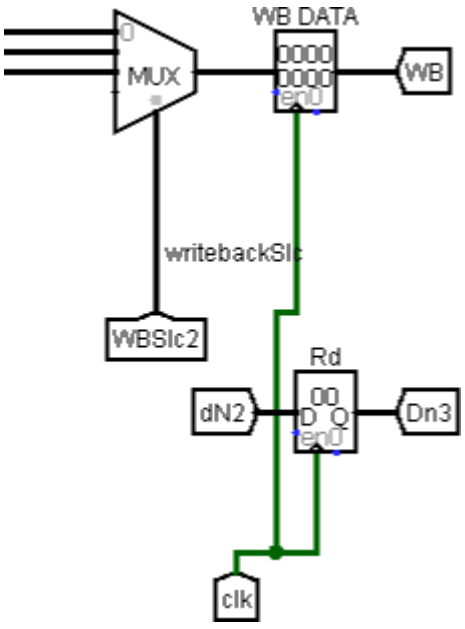


Figure 2.1.4.1 MEM/WB pipeline register

2.2 Pipeline Signals

Just as we implemented pipeline registers to store and forward the necessary data between stages, we also apply the same strategy to the control signals that guide the operation of each stage. These control signals, initially generated by the control unit during the Instruction Decode (ID) stage, must be propagated alongside the instruction data through the pipeline. This is essential to ensure that each instruction carries the correct set of control directives tailored to its type and required operations.

At each stage, the control signals relevant to that stage must reflect the values originally generated during decoding. For instance, during the Execute (EX) stage, the ALU must receive the appropriate control signals (such as ALU operation type and operand selection flags) that were generated for the specific instruction during the ID stage. These signals are stored in the pipeline register (ID/EX) and used during the EX stage to guide the behavior of the ALU accordingly or maybe the read and write signals for the memory access stage.

The concept is consistent across all stages: control signals must travel alongside the instruction data, stage by stage, using the corresponding pipeline registers (ID/EX, EX/MEM, MEM/WB). This design ensures that each instruction has access to the exact control configuration it requires at the moment it is executed, loaded from memory, stored, or written back. Without this mechanism, instructions could behave incorrectly due to outdated or misaligned control signal values, ultimately compromising the reliability of the pipelined architecture. The signals that was transformed could be shown in the next figures.

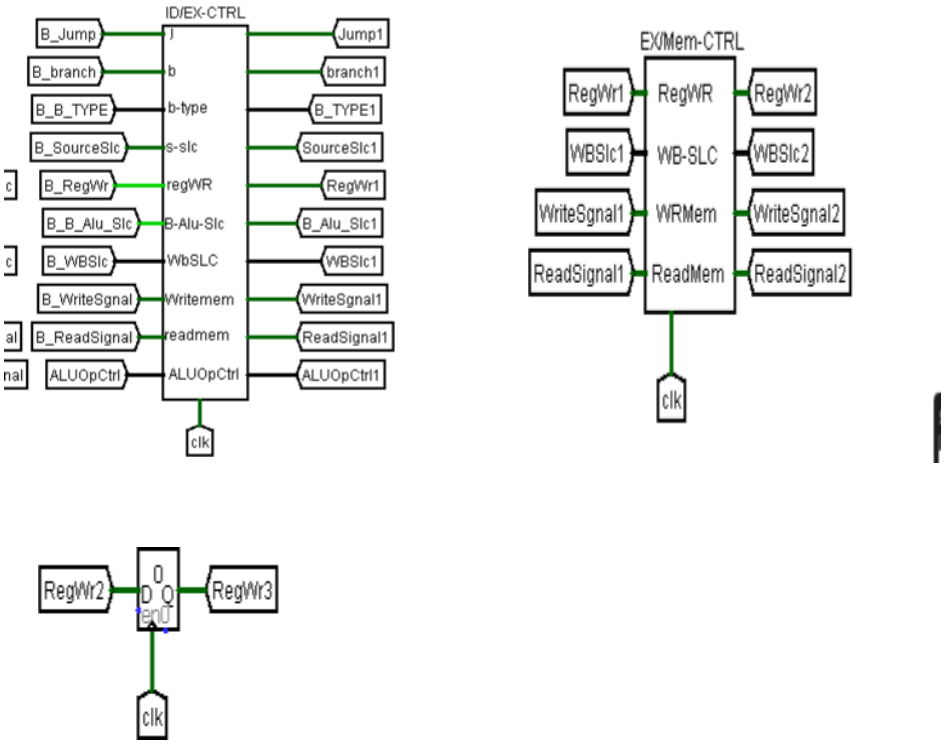


Figure 2.2 Pipelined Signals

The following table summarizes all the values and control signals that are transferred between stages via pipeline registers.

	IF/ID	ID/EX	EX/MEM	MEM/WB
REG_Variables	PC Instruction in IR	PC1 A_F B_F Imm32(I-type) Imm32(SB_type) D OP	ALU_result DN1 B_next	WB _data DN2
Signals	Jump Branch B_type Sign_select Source_select RegWr B-Alu_select WB_select Write_signal read_signal	B-Jump B-Branch B-B_type B-Sign_select B-Source_select B-RegWr B-B-Alu_select B-WB_select B-Write_signal B-read_signal	RegWR1 WB_Select1 Write_seignal1 Read_signal1	RegWR2

Note: In the following table, the letter B in ID/EX stage represents a Bubble, which refers to an inserted empty instruction used to resolve certain types of hazards. The purpose and behavior of bubbles within the pipeline will be discussed in detail in subsequent sections.

2.3 Pipeline Data Path Life Cycle

After having thoroughly explained the components involved in the pipelined processor, we now proceed to explore how data flows through the pipeline during instruction execution. To support this explanation, a figure illustrating the pipelined datapath is provided below. While the diagram includes elements such as stall, kill, and bubble mechanisms, these should not be the main focus at this point. Their inclusion is intended to provide a complete view of the datapath structure; a detailed discussion on how these mechanisms resolve pipeline hazards will follow in the subsequent sections.

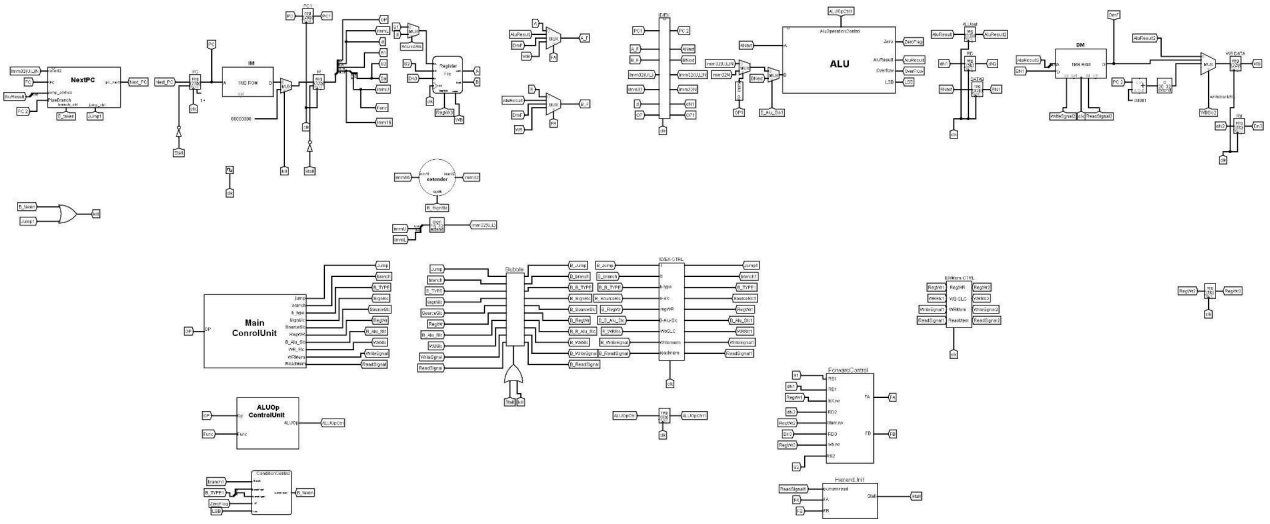
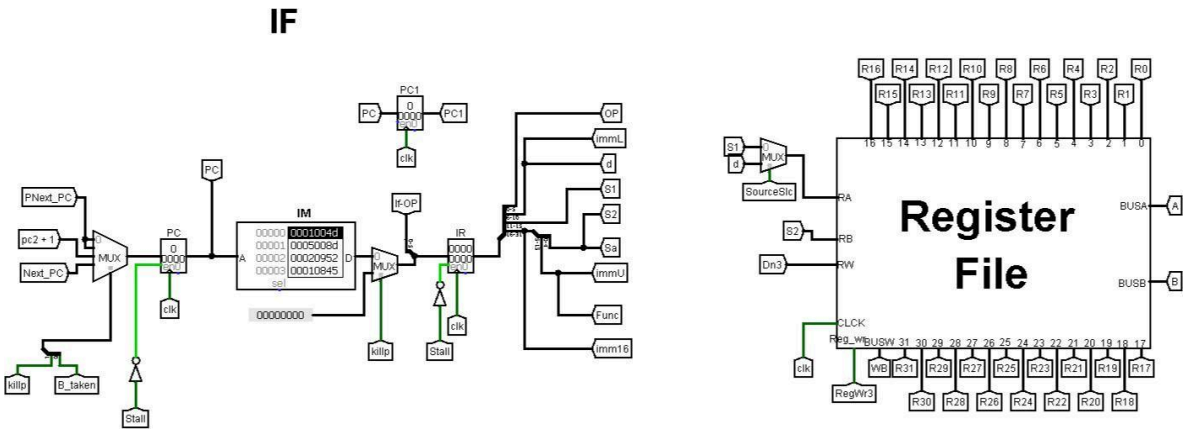
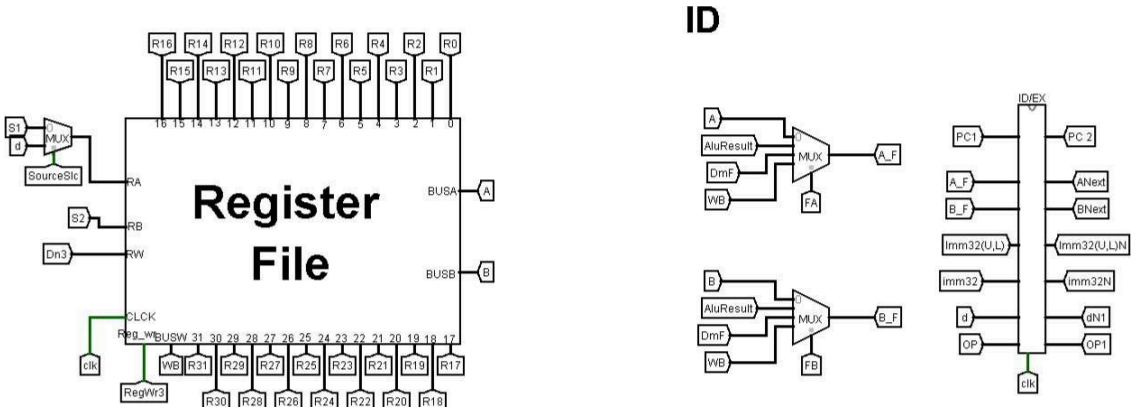


Figure 2.3 Pipelined Data path

In a pipelined processor architecture, instruction execution is divided into multiple stages to allow for simultaneous processing of several instructions, thereby significantly increasing throughput and overall performance. The datapath is organized into five sequential and logically distinct stages: Instruction Fetch (IF), Instruction Decode/Register Fetch (ID), Execute (EX), Memory Access (MEM), and Write-Back (WB). Each stage is separated by dedicated pipeline registers—IF/ID, ID/EX, EX/MEM, and MEM/WB—which serve as temporary storage points, preserving both data and control signals that are needed by subsequent stages. These registers are crucial to maintaining the synchronous flow of operations and ensuring that each stage can operate independently on different instructions in the same clock cycle.

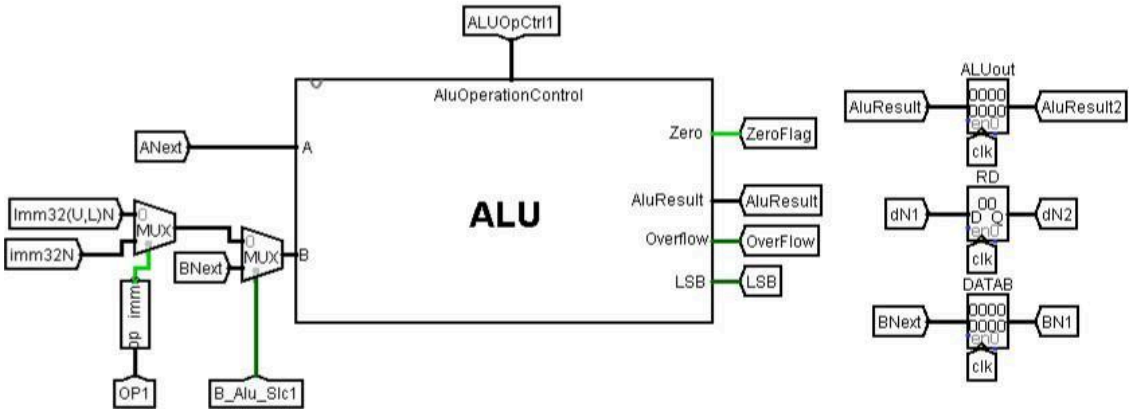


The Instruction Fetch (IF) stage marks the beginning of the instruction's journey through the pipeline. Here, the Program Counter (PC) holds the address of the next instruction to be executed. This address is sent to the instruction memory unit, which retrieves the corresponding machine instruction. Simultaneously, the PC is incremented by 1 and will be stored in the IF/ID pipeline register, allowing the next instruction to be fetched in the following cycle while the current one proceeds to the decode stage. This stage is purely combinational, and its output is stabilized by the IF/ID register to be safely consumed in the next stage.



In the Instruction Decode/Register Fetch (ID) stage, the instruction is broken down into its constituent fields—opcode, source and destination register identifiers, immediate values and function codes for R-type instructions. The register file is accessed to read the values of the source registers identified in the instruction. For instructions that utilize immediate values (e.g., I-type or SB-type instructions), the immediate field is extracted and sign-extended to 32 bits to match the architecture's word size. The control unit is also activated in this stage and generates all necessary control signals (such as RegWrite, MemRead, MemWrite, ALUSrc, Branch, etc.) based on the opcode. All of the decoded information, including the read register values (commonly denoted as A and B), the extended immediate values, the destination register identifiers, the incremented PC, and the generated control signals, are transferred to the next pipeline stage through the ID/EX register. This transfer ensures that the EX stage receives all the necessary inputs in the following cycle without having to re-access earlier stages.

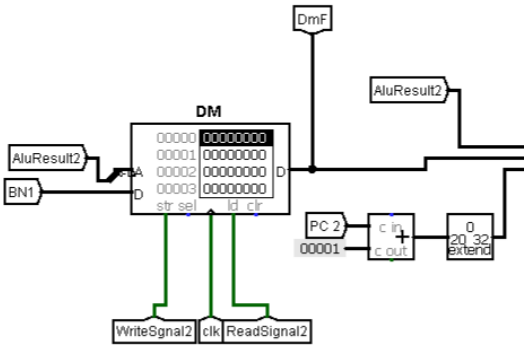
EX



Next, in the Execute (EX) stage, actual computation begins. The Arithmetic Logic Unit (ALU) receives its operands, which may be sourced either directly from the register file (A and B) or from a combination of one register value and an immediate operand, based on the control signal ALUSrc. The ALU Control Unit, which interprets signals from both the main control unit and the instruction's function field, determines the exact operation to perform (e.g., addition, subtraction, logical operations, etc.). The output of the ALU is the primary computational result of the instruction and is passed forward for further use. For certain instructions like store (sw), the second operand (register B value) is also carried forward, as it will be needed for writing into data memory in the next stage. The identity of the destination register (either Rd or Rt depending on the instruction type and RegDst signal) is also forwarded. All of this information—the ALU result, operand B for store instructions, and the target register—is packaged into the EX/MEM pipeline register, along with the relevant control signals such as MemRead, MemWrite, and RegWrite.

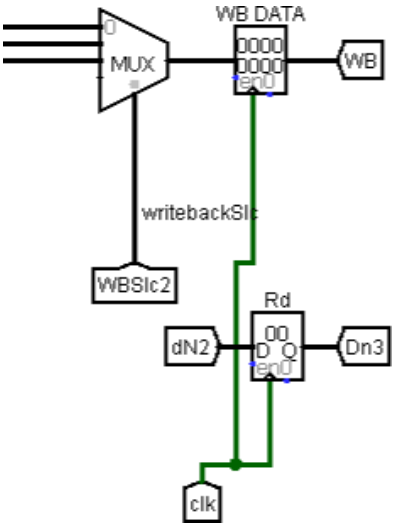
MEM

The Memory Access (MEM) stage interacts with data memory. If the instruction is a load (lw), the memory address (produced by the ALU in the EX stage) is used to fetch a word from memory, which is stored in the MEM/WB register for writing back to the register file. If the instruction is a store (sw), the value from operand B is written to the address computed by the ALU. For R-type instructions, which do not involve memory, the ALU result is simply forwarded unchanged. The control signals are again carried forward to guide the final write-back process. This stage is especially sensitive to memory-related hazards, and its correct execution



depends on prior data being available and no conflicts occurring from instructions ahead or behind in the pipeline.

Finally, the Write-Back (WB) stage is responsible for updating the register file. Based on the MemtoReg control signal, the value to be written is selected either from the memory (for lw instructions) or directly from the ALU output (for R-type or immediate arithmetic instructions). This value is written into the destination register whose address has been passed along through the pipeline stages. This marks the end of the instruction's lifecycle in the pipeline.



Throughout this entire process, control signals—generated during the ID stage—are carefully passed along with the data through each pipeline register to ensure consistent and correct instruction behavior. Without these signals, the subsequent stages would lack the context needed to perform their operations. This separation and passing of control and data across stages not only improves modularity and clarity but also enables instruction-level parallelism, allowing the processor to work on multiple instructions at different stages of execution simultaneously.

The pipelined design, while highly efficient, introduces potential hazards—namely structural hazards (resource conflicts), data hazards (when instructions depend on the results of prior instructions), and control hazards (due to branches). Though this detailed data flow overview assumes an ideal hazard-free environment, actual implementation requires mechanisms such as data forwarding, stalling, pipeline flushing, and bubble insertion to handle these challenges. These mechanisms, which are integral to ensuring correct execution without sacrificing performance, will be explored in depth in subsequent sections.

3. data Hazard Forwarding

In our **Risc pipelined processor**, instructions are executed in a pipelined fashion, meaning multiple instructions are overlapped in execution to improve performance. However, this overlap can introduce **hazards**, which are situations that prevent the next instruction from executing during its designated clock cycle. Two important mechanisms to deal with these hazards are hazard detection and forwarding (also known as bypassing).

3.1. Hazards in Risc

There are three main types of hazards:

1. **Data Hazards:**
   - Occur when an instruction produces a result that is required by a subsequent instruction.
   - The dependent instruction must wait until the preceding instruction completes its data read or write operation.
   - These hazards arise from data dependencies between instructions in the pipeline.
2. **Control Hazards:**
   - Arise from instructions that alter the program's control flow, such as branches and jumps.
   - The decision to change control flow often depends on the outcome of a preceding instruction.
   - These hazards introduce delays in determining the correct next instruction to execute.
3. **Structural Hazards:**
   - Occur when multiple instructions compete for the same hardware resource during the same clock cycle.
   - These hazards result from resource contention due to insufficient hardware to support all concurrent operations in the pipeline.

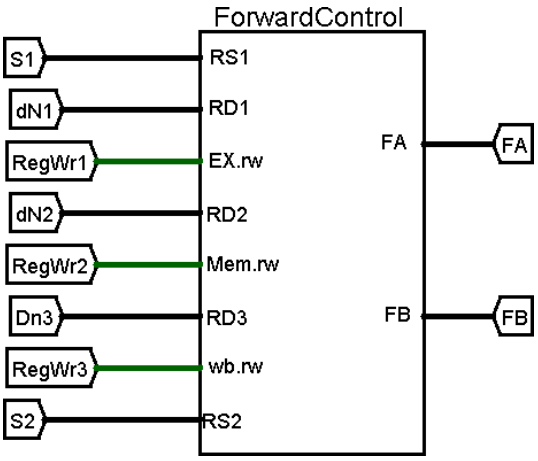
Hazard detection and forwarding specifically deal with **data hazards**.

3.2. Forwarding (Bypassing) Logic

Forwarding logic (also known as bypassing) is a key component in a pipelined Risc processor used to resolve **data hazards**—specifically **Read After Write (RAW)** hazards—without stalling the pipeline. It allows data to be forwarded (bypassed) from later stages of the pipeline to earlier ones, so dependent instructions can proceed without waiting for the register write-back to complete.

3.2.1. Purpose:

To avoid stalling the pipeline when a value needed by an instruction is still in the pipeline but has already been computed (in a later stage).



3.2.2. How It Works:

Forwarding logic detects when an instruction in the EX-stageneeds a value that is currently in the ALU or MEM or WB stages and redirects ("forwards") that value to the EX-stage without waiting for it to be written back to the register file.

3.2.3. Hardware Design:

Here's a simplified breakdown of the forwarding unit:

- **Inputs:**
  - Source register IDs from the ID-stage: RS1, RS2.
  - Destination register IDs from EX and MEM and WB stages: **EX.RD**(RD1), **MEM.RD**(RD2) and **WB.RD3**(RD3).
  - Write-back enable signals: EX.RegWr, MEM.RegWr, WB.RegWr.

- **Logic**:
  - If ((RS1 != 0) and (RS1 == RD1) and (EX.RegWr))ForwardA = 1
  - Else if ((RS1 != 0) and (RS1 == RD2) and (MEM.RegWr)) ForwardA = 2
  - Else if ((RS1 != 0) and (RS1 == RD3) and (WB.RegWr)) ForwardA = 3
  - Else ForwardA = 0

  - If ((RS2 != 0) and (RS2 == RD1) and (EX.RegWr))ForwardB = 1
  - Else if ((RS2 != 0) and (RS2 == RD2) and (MEM.RegWr)) ForwardB = 2
  - Else if ((RS2 != 0) and (RS2 == RD3) and (WB.RegWr)) ForwardB = 3
  - Else ForwardB= 0

- **Multiplexer Design**

  To implement forwarding, use 2 multiplexers in the ID stage:

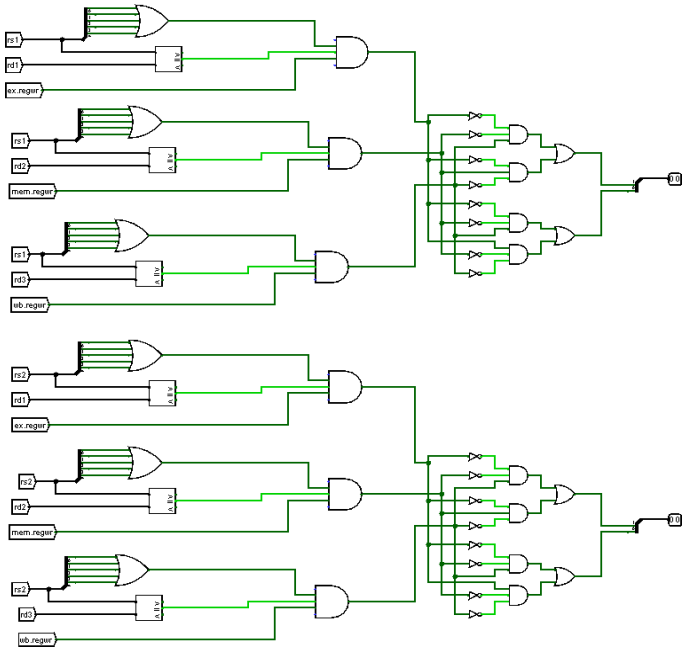
-two before ID/EX Register: one for Bus A and the other for Bus B.

-Select between:

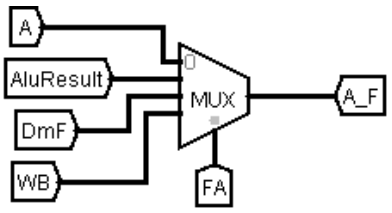
- Register file output (normal path)
- ALU result from EX
- Data from MEM
- Data from WB

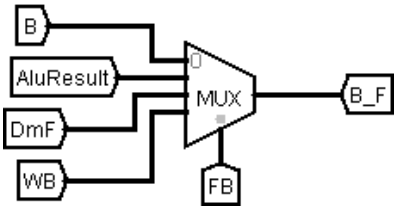
3.2.4. Hardware Block Diagram (Description)

- **Forwarding Unit**:
  - Takes in the `RegWrite` and `rd` from EX, MEM and WB stages.
  - Compare rd with `rs1`, `rs2` from ID stage.
  - Outputs `ForwardA` and `ForwardB` control signals.



- **Multiplexers**:
  - At ID (before ID/EX Register) at BusA and Bus B in ID stage
  - Controlled by `ForwardA` and `ForwardB`
  - Inputs:
    - 00: From register file (normal)
    - 01: From ALU result
    - 10: From MEM
    - 11: From WB





3.2.5. Limitations

Forwarding can't help in **Load-Use hazards**: Forwarding cannot resolve load-use hazards because the load instruction does not produce its result until the MEM stage. Therefore, if a subsequent instruction depends on the loaded data and is positioned immediately after the load instruction, it must be stalled to prevent incorrect execution.

4. Load delay and stall

In pipelined processor architectures, one of the critical challenges to maintaining correct instruction execution is the handling of *data hazards*. A specific and frequent type of data hazard is the load-use hazard, which occurs when an instruction attempts to use data loaded from memory by a preceding **lw** (load word) instruction before the memory access has completed. Due to the nature of pipelining, this data is not yet available for forwarding at the time it is needed by the dependent instruction. To prevent the use of incorrect or uninitialized data, the processor must delay the execution of the dependent instruction. This is achieved by implementing a stalling mechanism.

In this project, we have implemented a dedicated Hazard Detection Unit that detects load-use hazards in the ID stage and inserts a one-cycle stall when necessary. The following sections detail the design, implementation, verification, and integration of this hazard mitigation mechanism.

4.1 Theoretical Background

A *load-use hazard* arises when the destination register of a load instruction (currently in the EX stage) matches either source operand register of the instruction in the ID stage. In such a case, the data will not be available in time for correct operation due to the memory latency, even with forwarding logic in place.

This scenario necessitates a stall to allow the lw instruction to complete the memory access and write-back before the dependent instruction proceeds. The stall ensures data correctness without introducing any structural or control hazards.

4.2 Design Objectives

The design of the hazard detection unit adheres to the following objectives:

1. **Accurate Detection**: Identify only the cases where a true data dependency exists between a load instruction and a subsequent instruction.
2. **Minimal Performance Impact**: Insert a stall only when it is absolutely necessary, i.e., for load-use dependencies.
3. **Pipeline Compatibility**: Seamlessly integrate with the pipeline control signals to freeze the IF/ID register and the Program Counter (PC) during the stall cycle.

4.3 Functional Description

Inputs:

- **ex_memread** *(1 bit)*: A control signal generated in the EX stage indicating that the current instruction is a memory read operation.

- **FA and FB** *(2 bits each)*: Register addresses of the source operands of the instruction in the ID stage.

Output:

- **Stall** *(1 bit)*: A control signal that, when asserted (1), causes a one-cycle stall in the pipeline by freezing the PC and the IF/ID pipeline register.

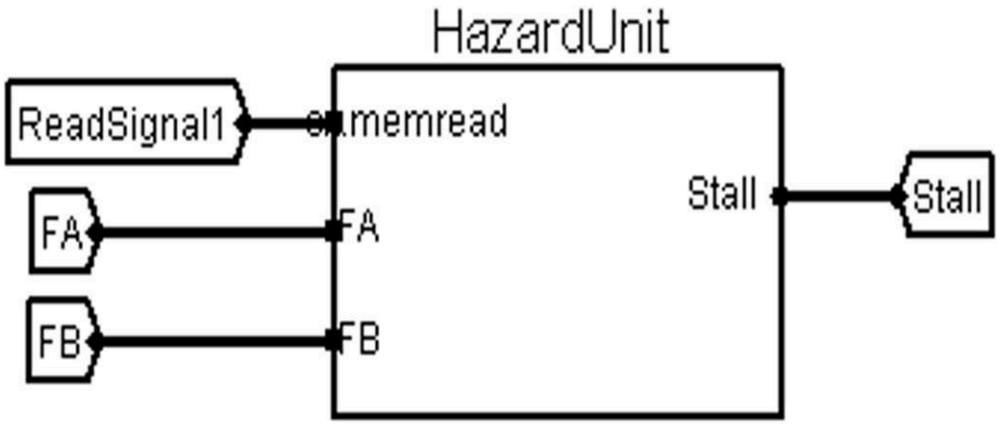


Figure 4.1

4.4 Hardware Implementation

The Hazard Detection Unit is implemented as a combinational logic block that compares register addresses using logic gates. The output is the Stall signal.

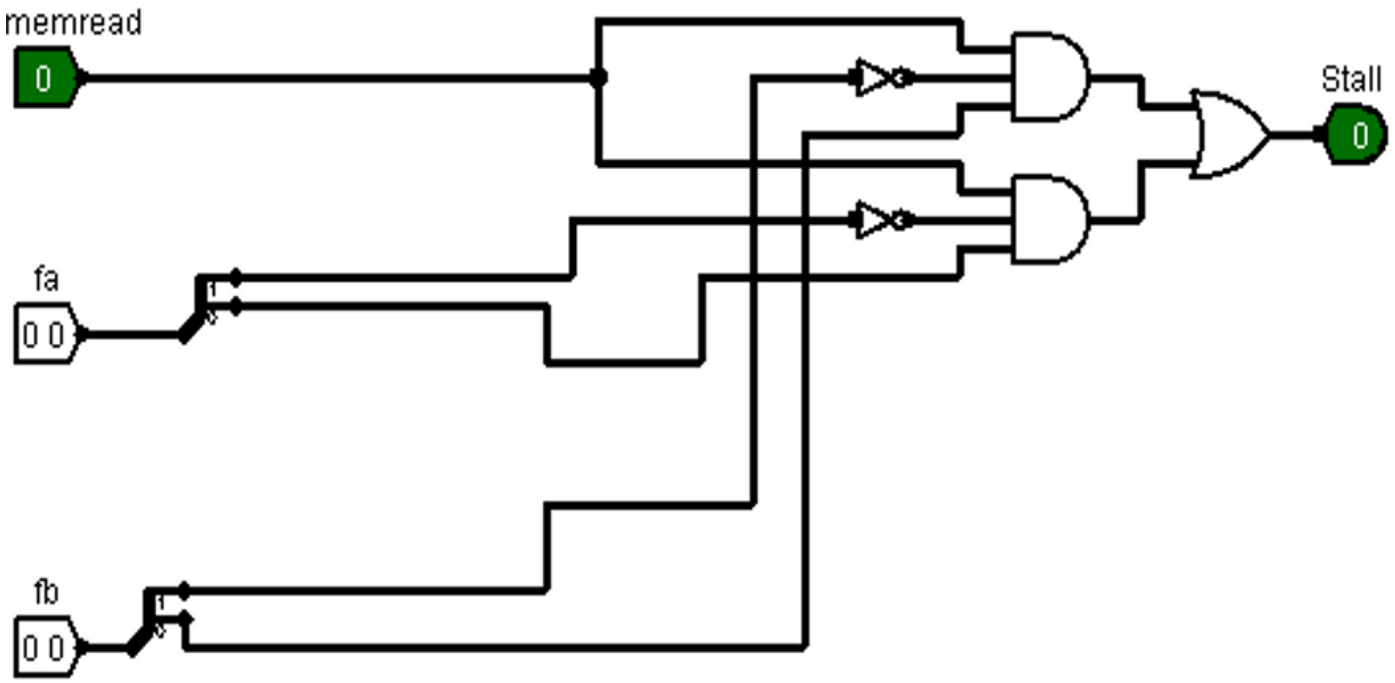


Figure 4.2

Table 4.1: Stall Signal Output for All Input Combinations

ex_memread	FA1	FA0	FB1	FB0	Stall
0	X	X	X	X	0
1	0	1	X	X	1
1	1	0	X	X	1
1	X	X	0	0	0
1	X	X	0	1	1
1	X	X	1	0	1
1	X	X	1	1	0
1	1	1	X	X	0

This table confirms that a stall is only triggered under the precise conditions described above, validating the logic against incorrect or unnecessary stalling.

A full image of the truth table is provided in Figure 4.2.

ex_memread	FA1	FA0	FB1	FB0	Stall
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	0
0	0	1	0	0	0
0	0	1	0	1	0
0	0	1	1	0	0
0	0	1	1	1	0
0	1	0	0	0	0
0	1	0	0	1	0
0	1	0	1	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	0	1	0
0	1	1	1	0	0
0	1	1	1	1	0
1	0	0	0	0	0
1	0	0	0	1	1
1	0	0	1	0	0
1	0	0	1	1	0
1	0	1	0	0	1
1	0	1	0	1	1
1	0	1	1	0	1
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	0	1	1
1	1	0	1	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	0	1	1
1	1	1	1	0	0
1	1	1	1	1	0

Figure 4.3

4.5 Pipeline Integration

The Stall signal is fed back into the Instruction Fetch (IF) stage to:

- Freeze the Program Counter (PC) from incrementing.
- Insert a NOP or bubble in the EX stage to delay execution.

This integration, where the Stall signal connects to PC enable, is shown in Figure 4.4, and insert a NOP is shown in Figure 4.5.

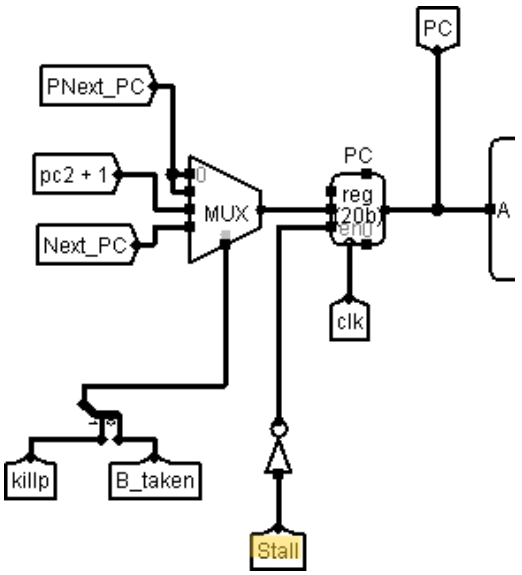


Figure 4.4

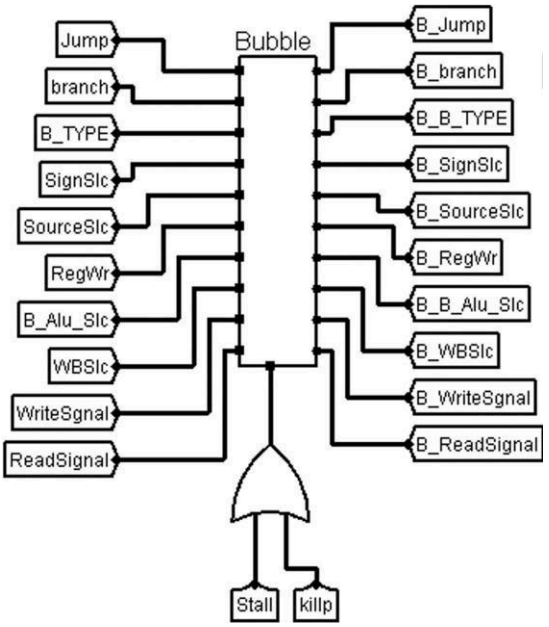


Figure 4.5

BUT HOW TO INSERT A BUBBLE?

To implement a bubble in our pipeline architecture, we introduced a controlled stalling mechanism is shown in Figure 4.6, that effectively inserts a NOP (No Operation) instruction into the pipeline when a load-use hazard is detected. This is achieved by first concatenating all relevant control signals from the Decode (ID) stage into a single composite signal bus. This bus is then passed through a 2-to-1 multiplexer, which selects between the normal signal values and a pre-defined zero vector based on the Stall control signal. When a hazard is detected and Stall is asserted, the multiplexer outputs the zero vector instead of the actual instruction signals. This vector propagates to the Execute (EX) stage, where it is interpreted as a NOP, effectively creating a bubble in the pipeline. The bubble allows the load instruction in the EX stage to complete without interference, thereby resolving the hazard. After the multiplexer, the signal bus is **s**plit back into individual signals, preserving the modular design of the pipeline. This method ensures correct execution flow while maintaining a clean and hardware-efficient implementation of hazard handling.

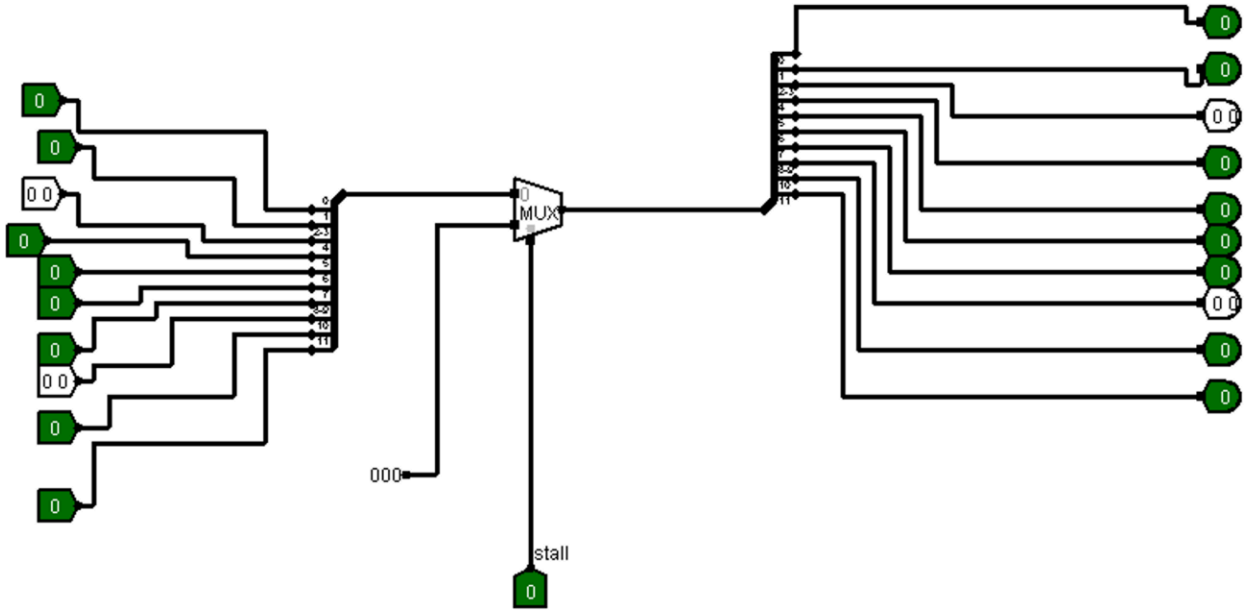


Figure 4.6

5. Control Hazards

In pipelined processor architectures, instructions are executed in overlapping stages to improve performance. However, this parallelism introduces various hazards, including control hazards, which occur when the flow of instruction fetch depends on the outcome of a branch or jump instruction. These hazards arise because the branch target address and decision are typically not known until a later stage in the pipeline.

Failure to handle control hazards correctly can result in incorrect instruction execution and corruption of program state. This chapter discusses the nature of control hazards, the associated performance implications, and the detailed implementation of control hazard resolution in our pipelined processor.

5.1 Nature of Control Hazards

In our 5-stage pipeline (IF → ID → EX → MEM → WB), the Program Counter (PC) is updated every cycle to fetch the next instruction. When a control transfer instruction—such as a conditional branch (BEQ, BNE, BLT, etc.) or an unconditional jump (JALR)—is executed, it may redirect the PC to a new target address. However, the decision to perform this redirection is not resolved until the EX stage, after operand comparison and branch condition evaluation.

By the time the EX stage resolves the control transfer:

- The next instruction has already been fetched (IF stage),
- And its control signals have already been decoded (ID stage).

If the control transfer is taken, the instructions in IF and ID must be discarded. This results in what is known as a control hazard or branch hazard.

5.2 Pipeline Impact and the Two-Cycle Penalty

Because our pipeline resolves branches and jumps in the EX stage till now and does not implement branch prediction or delay slots, it inherently suffers a two-instruction penalty when a control transfer is taken.

The sequence of events is as follows:

Cycle	IF	ID	EX	Outcome
t	BEQ			Branch fetched
t+1	NEXT1	BEQ		Instruction fetched normally
t+2	NEXT2	NEXT1	BEQ	Branch resolved (taken)
t+3	Target	(flushed)	(flushed)	Resume at correct target

- NEXT1 and NEXT2 are invalid instructions fetched along the wrong path.
- They must be killed (flushed) and replaced with NOPs.
- This results in a 2-cycle control hazard stall whenever a branch or jump is taken.

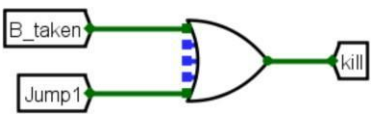
5.3 Flush Logic and Kill Signal Generation

To resolve control hazards, our processor implements a Kill-based flushing mechanism, driven by signals generated in the EX stage. The Kill signal is asserted when a control transfer is confirmed.

5.3.1 Kill Signal Definition

The Kill signal is computed as:

```
Kill = BranchTaken OR Jump
```



Where:

- BranchTaken is output from the Condition Control Unit in the EX stage,
- Jump is a control signal for unconditional jump instructions (JALR), also determined in EX.

This signal captures all control flow transfers that require flushing previous stages.

5.4 Implementation of Instruction Flushing

The Kill signal is used to flush both the IF and ID stages to prevent incorrectly fetched or decoded instructions from executing.

5.4.1 Instruction Register Flushing (IF Stage)

In the IF stage, a multiplexer is inserted between the output of the Instruction Memory (IM) and the Instruction Register (IR) as shown in Figure5.1.

- **MUX Inputs**:
    - Input 0: Normal instruction from Instruction Memory
    - Input 1: Zero constant
- **MUX Select**:
    - Driven by Kill
      → If Kill = 1, the zero constant is passed to IR
      → If Kill = 0, the normal instruction is passed

This ensures that the instruction fetched during a mispredicted control transfer is invalidated.

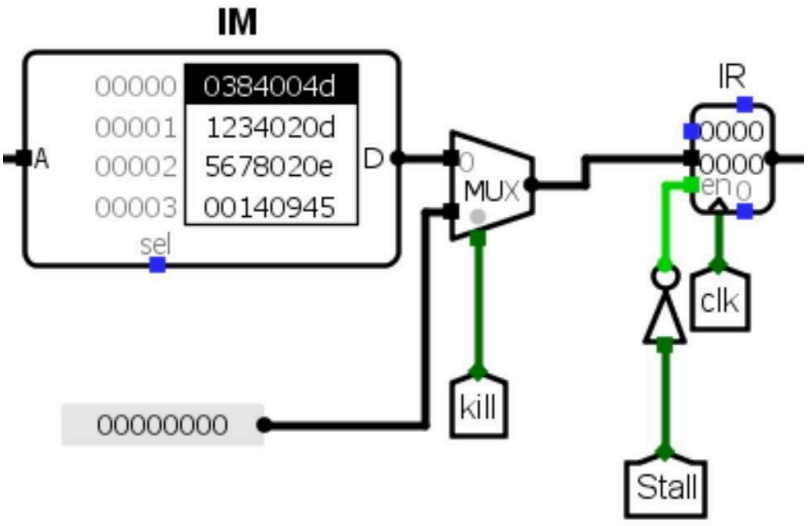


Figure5.1

5.4.2 Bubble Triggering and Instruction Invalidation

For the instruction in the decode stage, bubble insertion is triggered using the same Kill signal. Rather than duplicating bubble logic, the Kill signal is simply combined with the Stall signal (from the load hazard unit), and the result is passed to the bubble control system as shown in Figure5.2.

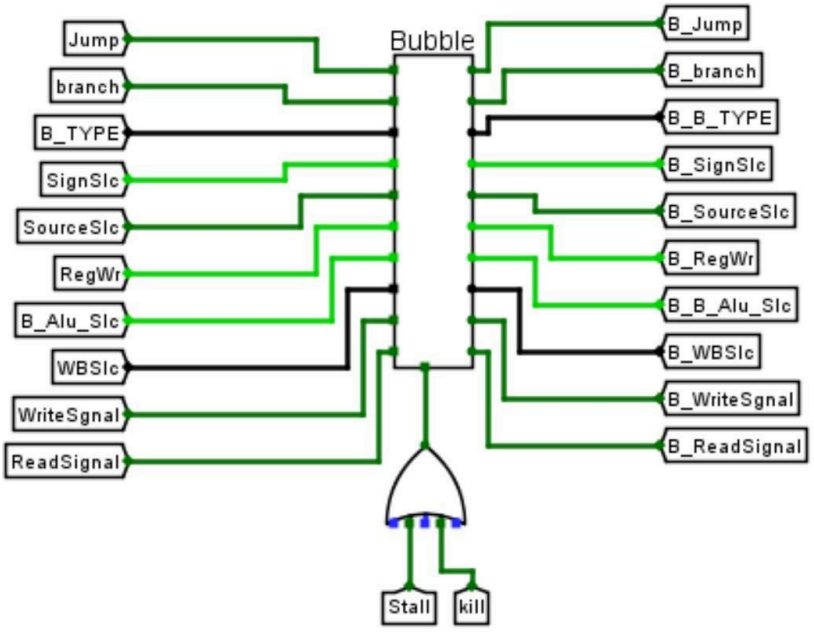


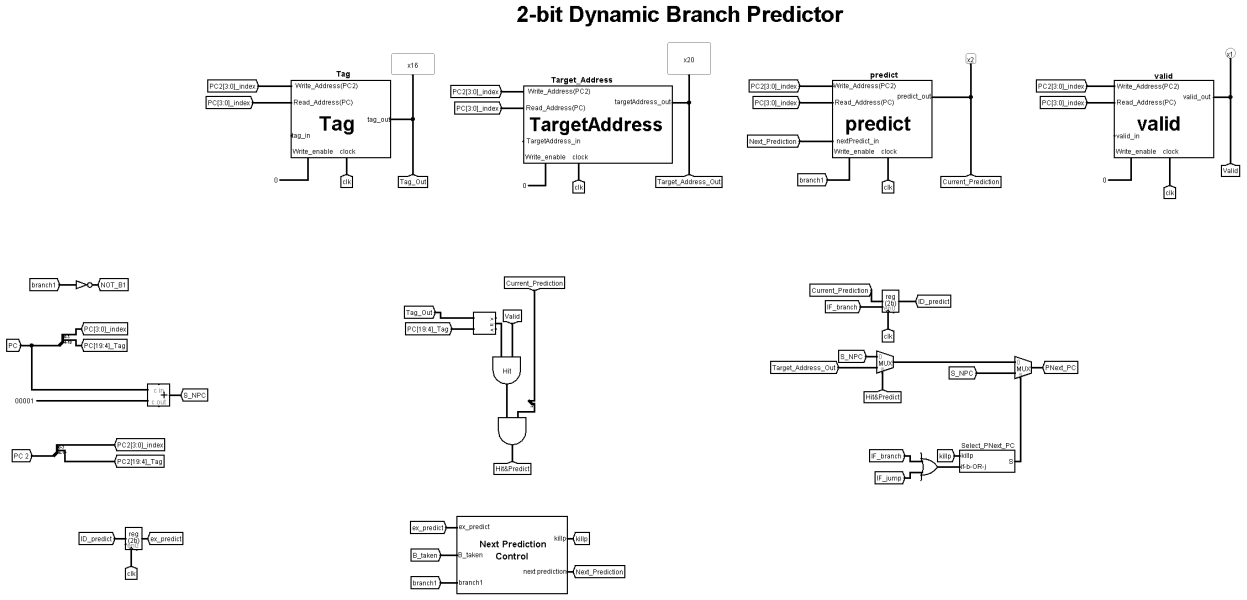
Figure5.2

Note: The unified bubble insertion mechanism and its detailed implementation are fully described in Section ***4.5. This system ensures that control and data hazards share a consistent and efficient flushing interface***

6. 2 Bit Dynamic Branch Prediction in RISC Processor

6.1. Overview

In our pipelined Risc processor, control hazards caused by branch instructions can significantly degrade performance by introducing stalls. To mitigate this, branch prediction techniques are used to guess the outcome of a branch instruction before it is resolved. One effective method is the 2-bit dynamic branch prediction, which improves accuracy over simple static or 1-bit schemes by maintaining a 2-bit saturating counter for each branch instruction.



6.2. Principle of Operation

The 2-bit predictor uses a small **finite state machine (FSM)** to predict whether a branch will be taken or not taken. Unlike 1-bit predictors, which change prediction with a single incorrect outcome, the 2-bit predictor requires two consecutive mispredictions to switch its state, thereby reducing the impact of occasional anomalies in branching behavior.

6.2.1 State Machine

The 2-bit predictor operates in four states:

State	Meaning	Prediction
00	Strongly Not Taken	Not Taken
01	Weakly Not Taken	Not Taken
10	Weakly Taken	Taken
11	Strongly Taken	Taken

Transition Rules:

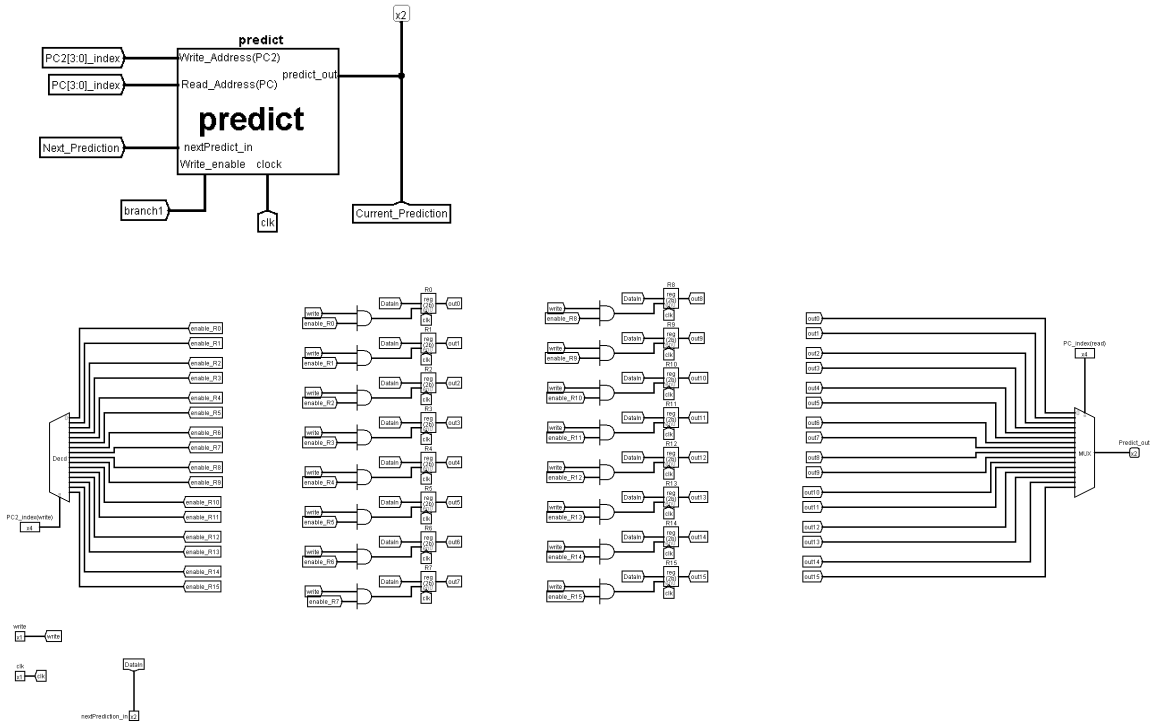
- If the branch is taken, increment the counter (up to 11).

- If the branch is not taken, decrement the counter (down to 00).

6.3. Hardware Design

6.3.1. Components

1. **Branch History Table (BHT)**:
    - A small memory array indexed by the lower bits (the first four bits) of the program counter (PC).
    - Each entry contains a 2-bit counter representing the prediction state.



(the above figure is the design of predict block)

2. **PC Indexing Logic**:
    - Extracts the relevant bits from the PC to access the BHT.
    - Typically, only the least significant bits are used to reduce hardware size.

3. **Prediction Logic**:
    - Reads the BHT entry at fetch stage.
    - If the state is 10 or 11, predict "taken"; otherwise, predict "not taken".

4. **Update Logic**:

- At the branch EX stage, the actual outcome is known.
- The corresponding BHT entry is updated based on the outcome, modifying the 2-bit counter.

6.3.2. Timing and Placement

- **Fetch Stage (IF)**: BHT is read using PC to make the prediction.
- **Execution Stage (EX)**: Actual branch decision is made and BHT is updated with the outcome.

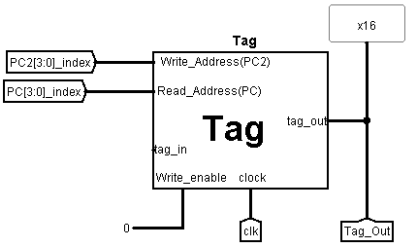
6.4. How does it work

FIRST:

We Create BTB Storage by Using four blocks of registers modules:

- Tag (16 bit)
- Target Address (20 bit)
- Predict (2 bit)
- Valid (1 bit)

All 4 blocks of registers use the same address: PC/PC2 [3:0].

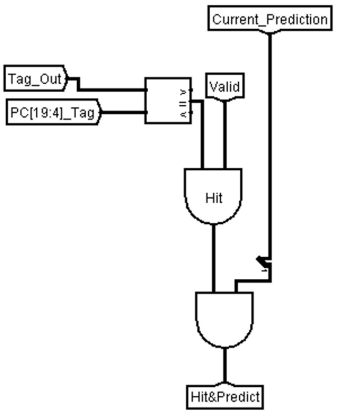


SECOND:

In IF Stage – Predict Branch:

1. **Extract Index and Tag from PC:**
   - PC_index = PC[3:0] → block address
   - PC_tag = PC[19:4] → Compare with stored tag
2. **Read from blocks using PC_index:**
   - stored_tag from Tag block
   - stored_target from Target block
   - stored_counter from Counter block
   - valid_bit from Valid block
3. **Check for BTB Hit:**
   - BTB_Hit = (stored_tag == PC_tag) AND (valid_bit == 1)
4. **Make Prediction:**
   - predicted_taken = stored_counter[1] (MSB)
   - If BTB_Hit AND predicted_taken == 1 : → Set next_PC = stored_target

     Else: → Set next_PC = PC + 1



5. **Pass info to later stages (save PC, stored_counter)**
   - PC (IF stage) **-> IF/ID pipeline register ->** PC1 (ID stage) **-> ID/EX pipeline register ->** PC2 (EX stage)
   - Predict (IF stage) **-> IF/ID register ->** ID_predict (ID stage) **-> ID/EX register ->** EX_predict (EX stage)

THIRD:

In EX Stage – Resolve and Update:

1. **Inputs needed in EX stage:**
   - EX_PC (PC2): original PC of branch instruction
   - EX_taken (B_taken): actual branch outcome (1 or 0)
   - EX_target: actual branch target address
   - EX_is_branch (branch 1): control signal
   - EX_predict: the stored prediction
2. **Extract again:**
   - PC2_index = EX_PC2[3:0]
   - PC2_tag = EX_PC2[19:4]
3. **Write to block at PC2_index:**
   - Predict block -> write next prediction:

   -Read current value (stored_counter)

   -If EX_taken == 1: increment (up to 11)

   -If not: decrement (down to 00)

   - Write updated value at PC2_index according to Counter Update Logic:

     To update the 2-bit predictor:

     - From EX stage:

     -get stored_counter from the second

     Predict register(ID/EX register)

     -get EX_is_branch(branch1)

     -get B_taken

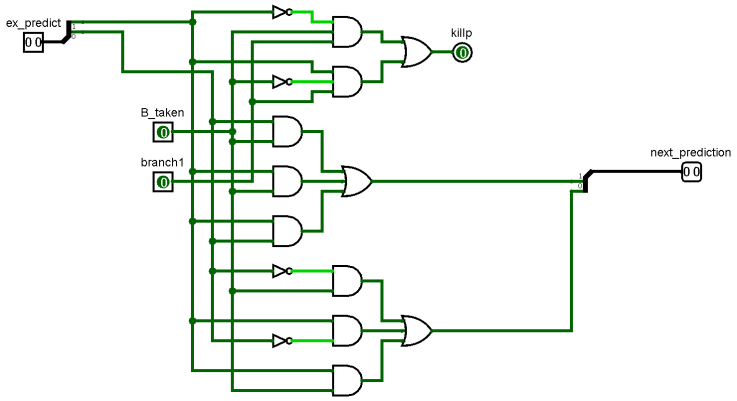
     And according to the next truth table:

-we get killp

-we get nextPrediction

inputs				outputs		
ex_predict_bit 1	ex_predict_bit 0	B_taken	branch1	killp	nextPrediction_bit1	nextPrediction_bit 0
0	0	0	0	0	x	x
0	0	0	1	0	0	0
0	0	1	0	0	x	x
0	0	1	1	1	0	1
0	1	0	0	0	x	x
0	1	0	1	0	0	0
0	1	1	0	0	x	x
0	1	1	1	1	1	0
1	0	0	0	0	x	x
1	0	0	1	1	0	1
1	0	1	0	0	x	x
1	0	1	1	0	1	1
1	1	0	0	0	x	x
1	1	0	1	1	1	0
1	1	1	0	0	x	x
1	1	1	1	0	1	1

(the above table is the Truth Table of next prediction control)



(The combinational circuit of next prediction control)

4. **Write Enable for predict block = EX_is_branch (branch1)**
   Branch1 = write enable for the predict block

FOURTH:

EX Stage - Misprediction Handling:

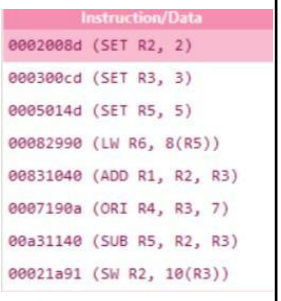
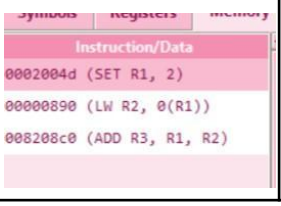
1. **From stored data in EX stage:**
   - predicted_taken(ex_predict) = stored_counter
   - mispredicted = EX_taken(B_taken) XOR predicted_taken(ex_predict)
2. **If mispredicted == 1:**
   - Kill the instructions that were incorrectly fetched :Convert Next1(that is IF stage)and Next2 (ID stage) into bubbles
   - Correct the PC:

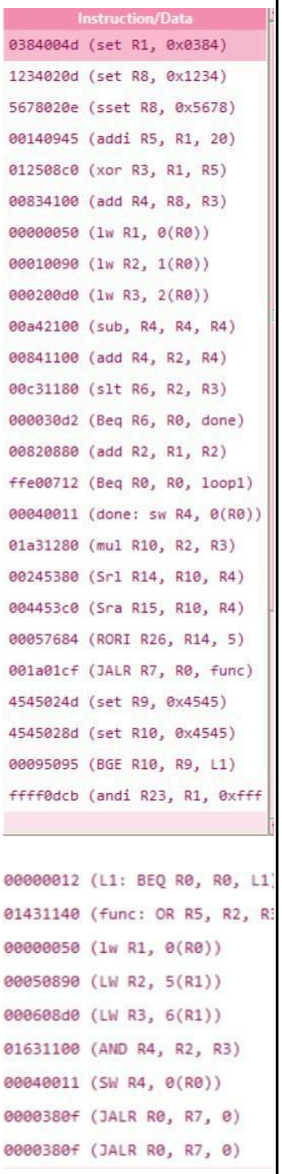
   If EX_taken == 1: set PC = EX_target

Else: set PC = EX_PC + 1

7.Testing and verification

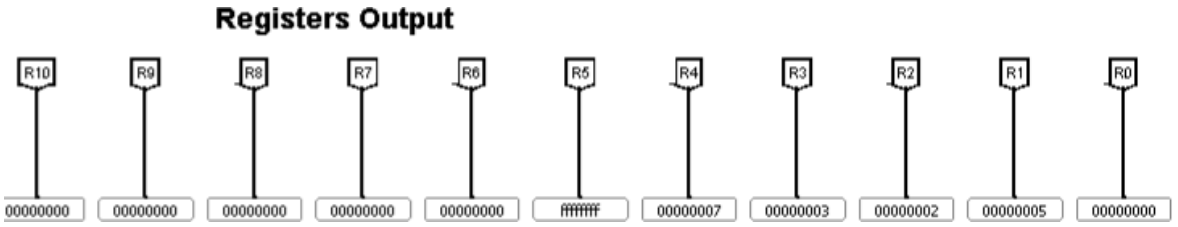
Now that we have completed the processor in all aspects, it's time to begin testing. We will conduct several tests, including general pipeline functionality, data hazards and forwarding, load delays, control hazards, as well as the *testCode* task and array sum operations.

Program Num	Test Subject	Program	Result
1	General Pipeline		Works Well
2	Data hazards		Works Well
3	load delay		Works Well

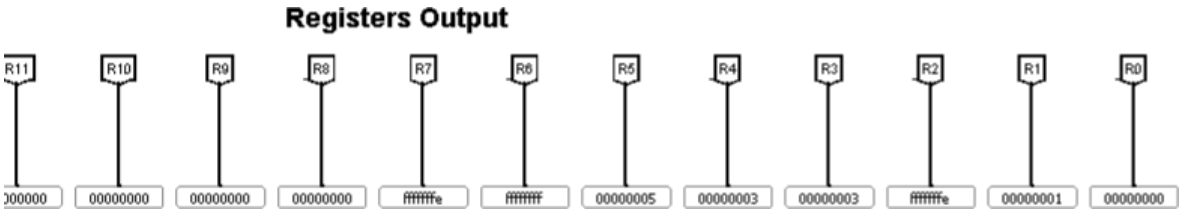
4	testCode		Works Well
5	array sum		Works Well

Processor Outputs

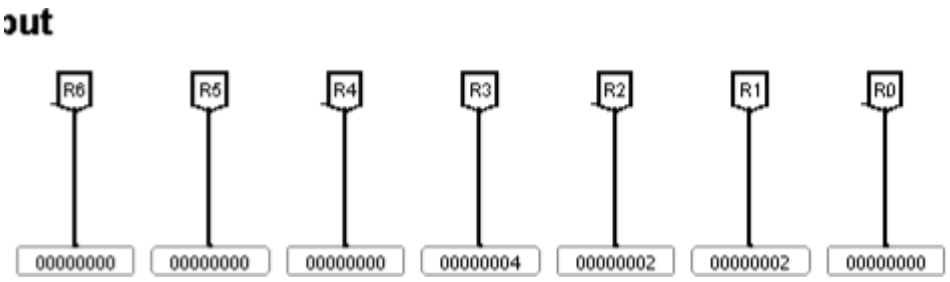
Program1



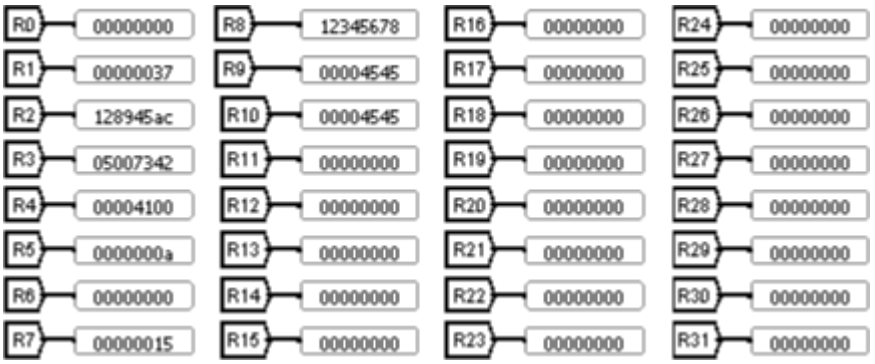
Program2



Program3

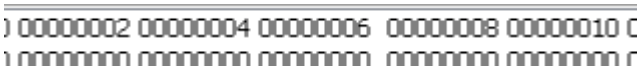


Program4



Program5

Data memory



Processor Output

6. Workflow

6.1 Meetings

Day	Period	Place	What We Have Done
Thursday 1/5/2025	10AM: 9PM	Offline	Phase2: - Pipeline Design - Data Hazards - Load Delay - Control Hazard - Try 2 Bit Branch Predictor
Sunday 4/5/2025	3PM : 9PM	Offline	Try 2 Bit Branch Predictor in more details
Monday 5/5/2025	5PM : 9PM	Offline	Complete 2 Bit Branch Predictor

6.2 Contribution

Omnia Ayman	Alaa Ayman	Rahma Mostafa
- Pipeline design - Documentation - Video	- Pipeline design - Documentation - Video	- Pipeline design - Documentation - Video