Fayoum University
Faculty of Engineering
Computer Engineering Course

RISC CPU

Project Report

Single-Cycle Processor

Group Members:
Youssef Ibrahim Mohamed (CSE)
Mohamed Wageh Mahmoud (CSE)
Mohamed Ahmed Kassem (ECE)

Spring 2025 Project

Table of Contents

1. Introduction

In this project, we were tasked with designing and simulating a **32-bit RISC (Reduced Instruction Set Computer) processor** using Logisim. The processor is based on a simplified architecture commonly found in educational implementations of RISC designs.

While the overall project's final goal is to construct a **pipelined processor**, this phase focuses on building a **single-cycle processor**.
A processor in which each instruction completes all its operations in a single clock cycle. This means that fetching, decoding, executing, memory access, and write-back occur during one clock tick.

Objectives

- Design a functional 32-bit single-cycle RISC processor.
- Support basic arithmetic, logical, memory, and branching instructions.
- Implement essential datapath components: register file, ALU, control unit, memory, and program counter.
- Simulate instruction execution with Logisim.

Processor Overview

The designed processor includes the following features:

- **32-bit Architecture:** All data paths, operations, and registers are 32 bits wide.

- **32 Registers:**
  - `R0` is **hardwired to zero;** any attempt to write to it is ignored.
  - `R1` to `R31` are **general-purpose registers** available for arithmetic and data storage

- **20-bit Program Counter (PC):**
  - This allows the processor to address up to 2^{20} **= 1,048,576** instruction memory locations.
  - The PC is incremented (or conditionally modified) each cycle to point to the next instruction.

Supported Instruction Formats

The processor recognizes **three instruction formats**, each corresponding to a different category of operation:

R-Type Format (Register-Register Operations)

Used for operations that involve two source registers and one destination register (e.g., `add`, `sub`).

Field	Size(bits)	Description
F	11	Function-specific bits (e.g., used to select an operation)
S2	5	Source Register 2
S1	5	Source Register 1
D	5	Destination Register

OP	6	Opcode (operation code)

I-Type Format (Immediate Operations & Memory Access)

Used for operations that involve a constant (immediate) value, such as `addi`, `lw`, or `sw`.

Field	Size(bits)	Description
Imm	16	Immediate Value
S1	5	Source Register
D	5	Destination Register
OP	6	Op code

SB-Type Format (Branching Instructions)

Used for conditional branch instructions like `beq`, which compare registers and possibly modify the PC.

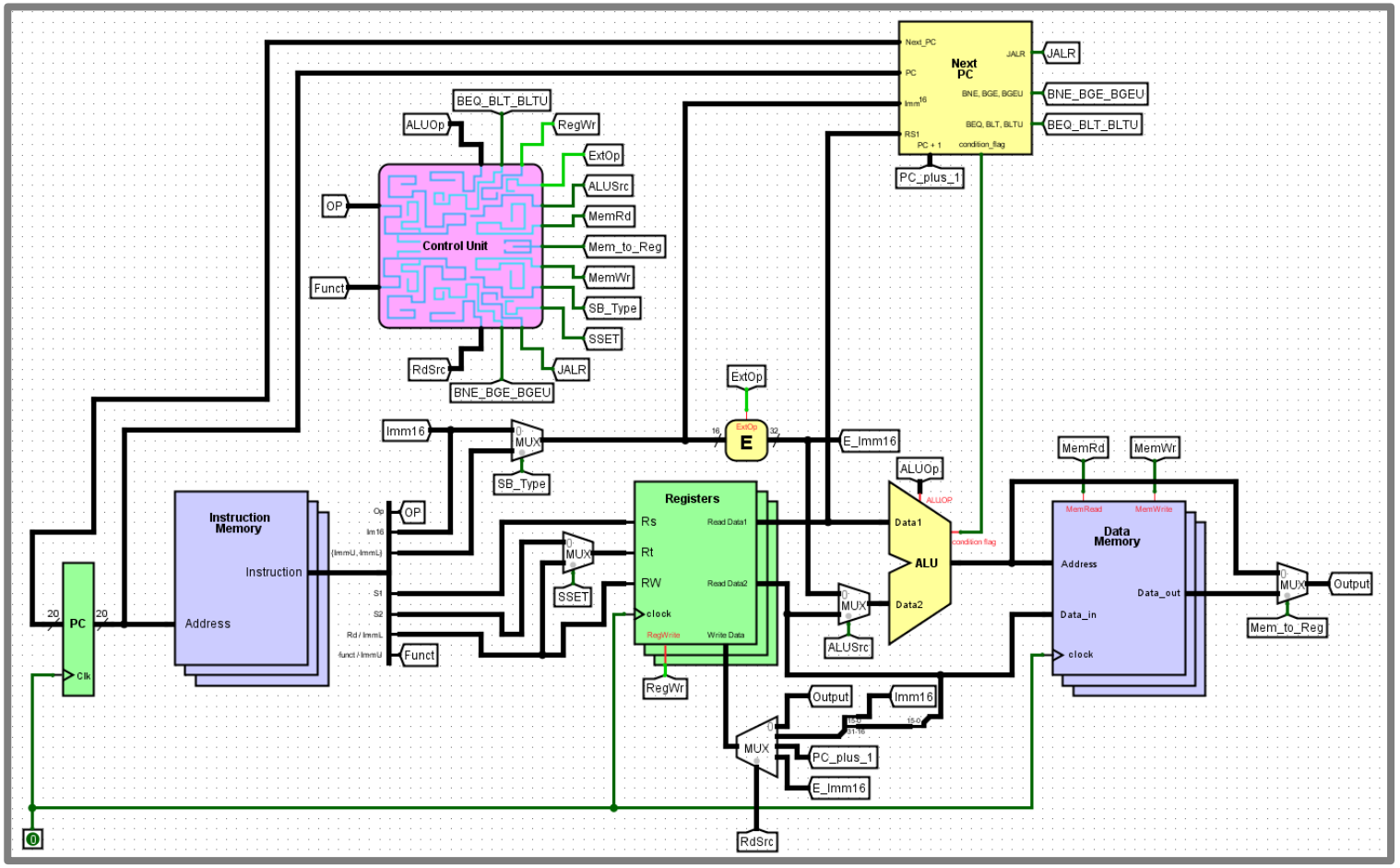
Field	Size(bits)	Description
ImmU	11	Upper part of the branch offset
S2	5	Source Register 2
S1	5	Source Register 1
ImmL	5	Lower part of the branch offset
OP	6	Opcode (operation code)

2. Full Overview of Data Path

This section provides a complete overview of the processor's data path, the core structure that dictates how data flows through the processor to execute instructions

Below is a high-level snapshot of the processor architecture as implemented in **Logisim**:

This diagram illustrates how different components are connected to carry out instruction execution in a single cycle. Each black or green line represents a signal or bus (group of wires), transferring data or control signals between components.



Main Components Overview

The processor is composed of the following main functional units:

1. Program Counter (PC)

- A 20-bit register that holds the address of the current instruction.
- Automatically updated each cycle to fetch the next instruction, unless altered by a branch instruction.

2. Instruction Memory

- Stores the set of instructions the processor can execute.
- Uses the address from the PC to output the current instruction.

3. Control Unit

- Decodes the instruction's opcode and generates control signals.
- These signals control all other components (e.g., whether to write to registers, select ALU operation, or read/write memory).

4. Next PC Logic

- Determines the next value of the PC depending on the type of instruction.
- Supports sequential execution (`PC+4`) and conditional branching (e.g., `beq`, `bne`).

5. Register File

- Consists of 32 general-purpose registers (R0–R31).
- Can read two source registers and write to one destination register per cycle.

6. ALU (Arithmetic Logic Unit)

- Performs arithmetic and logical operations (e.g., add, subtract, AND, OR).
- Receives control signals from the ALU Control logic to determine the operation.

7. Data Memory

- Used for `load` and `store` instructions.
- Allows reading or writing 32-bit values to/from specific memory addresses.

8. Multiplexers (MUXes)

- Allow dynamic selection between different inputs.
- Used in several places, such as choosing between register and immediate values, or memory vs ALU results.

9. Sign Extenders

- Extend immediate values (e.g., 16-bit or 5-bit) to 32-bit to be compatible with the rest of the datapath.

How It All Works

1. **Instruction Fetch**:

   The PC provides an address to the Instruction Memory, which outputs a 32-bit instruction.

2. **Instruction Decode:**

   The Control Unit interprets the opcode and sets control signals. Register

numbers are used to read data from the Register File.

3. **Execution:**

   Data from registers and immediate values are sent to the ALU for computation.

4. **Memory Access (if applicable):**

   For `lw` or `sw`, the ALU result (address) is used to read from or write to the Data Memory.

5. **Write Back:**

   Results are written back to the Register File, if required.

6. **PC Update:**

   The Next PC logic calculates the next instruction address, possibly adjusting for branches.

3. Instruction Splitter

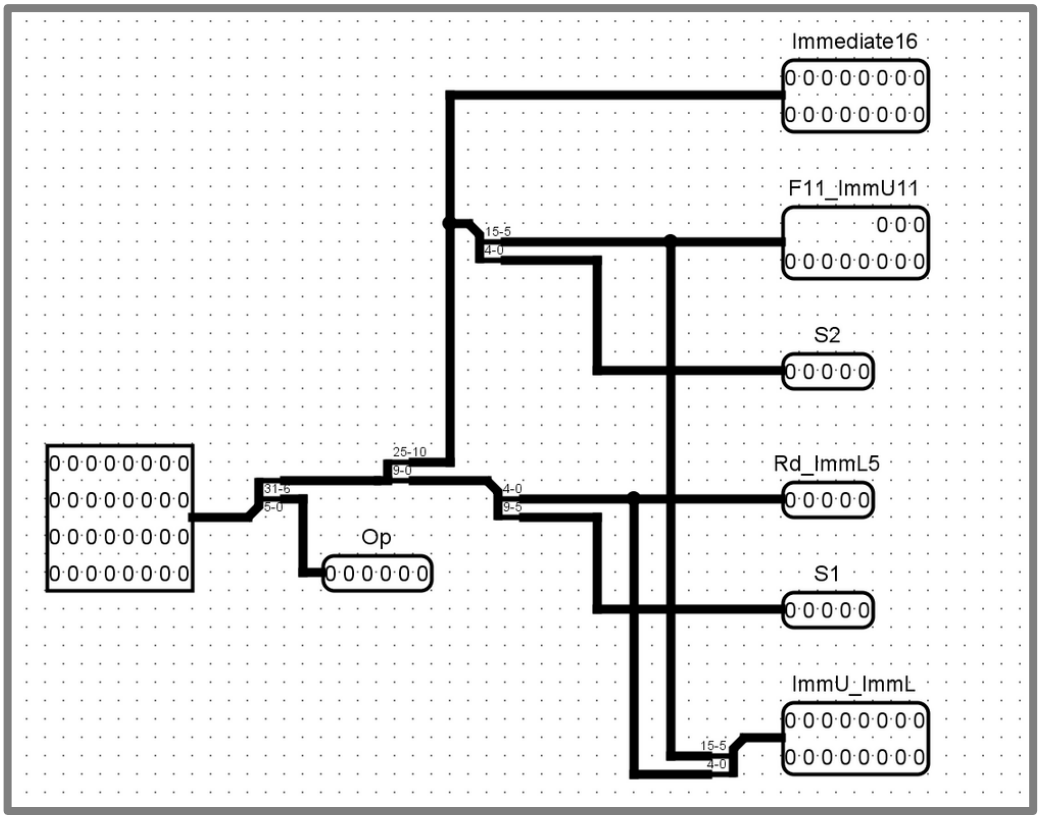
The **Instruction Splitter** is responsible for breaking down a single 32-bit instruction into smaller fields that different components of the processor can understand and use. Each instruction type (R-Type, I-Type, SB-Type) has its own format, but all follow the same general structure: **operation code (opcode)** and **operands or immediate values**.

What It Does

When an instruction is fetched from the Instruction Memory, it enters the Instruction Splitter, which divides it into specific parts based on their bit

positions. This decomposition allows other components, like the Control Unit, ALU, or Register File, to interpret and execute the instruction correctly.

Circuit Snapshot :



Bit Breakdown

Let's break down the 32-bit instruction field:

1. Opcode (OP): 6 bits

- Located at the least significant bits (`[5:0]`)
- **Used in all instruction formats**
- Tells the Control Unit what operation the instruction is performing.

2. **Remaining 26 Bits:** Split depending on instruction type:

- **For I-Type:**

- Immediate (Imm^{16}) = 16 bits (`[31:16]`)
- Source Register (S1) = 5 bits (`[15:11]`)
- Destination Register (Rd) = 5 bits (`[10:6]`)

- **For R-Type**:
  - Function code (F^{11}) = 11 bits (`[31:21]`)
  - Source Register 2 (S2) = 5 bits (`[20:16]`)
  - Source Register 1 (S1) = 5 bits (`[15:11]`)
  - Destination Register (Rd) = 5 bits (`[10:6]`)

- **For SB-Type (Branch)**:
  - Upper Immediate ($ImmU^{11}$) = 11 bits (`[31:21]`)
  - Source Register 2 (S2) = 5 bits (`[20:16]`)
  - Source Register 1 (S1) = 5 bits (`[15:11]`)
  - Lower Immediate ($ImmL^{5}$) = 5 bits (`[10:6]`)

3. **Combined Immediate (ImmU_ImmL)**: 16 bits total
   - Concatenation of `ImmU`^{11} and `ImmL`^{5}
   - Used for computing branch offsets in SB-Type instructions.

Summary

The Instruction Splitter simplifies decoding by ensuring each piece of an instruction is routed to the correct processor module. By organizing the 32-bit instruction into standardized fields, the CPU can flexibly support multiple instruction formats without confusion.

4. Instruction Memory

Instruction Memory is a **read-only memory (ROM)** that stores the list of instructions a processor should execute. These instructions are written in **machine language** (binary format) and are executed one at a time by the processor.

In this project, the Instruction Memory is implemented as a **1M × 32-bit ROM**, meaning it can hold up to **1 million 32-bit instructions**. However, for simplicity and memory efficiency, we use only the first **2^{20} = 1,048,576 address locations**, since the Program Counter (PC) is a 20-bit register.

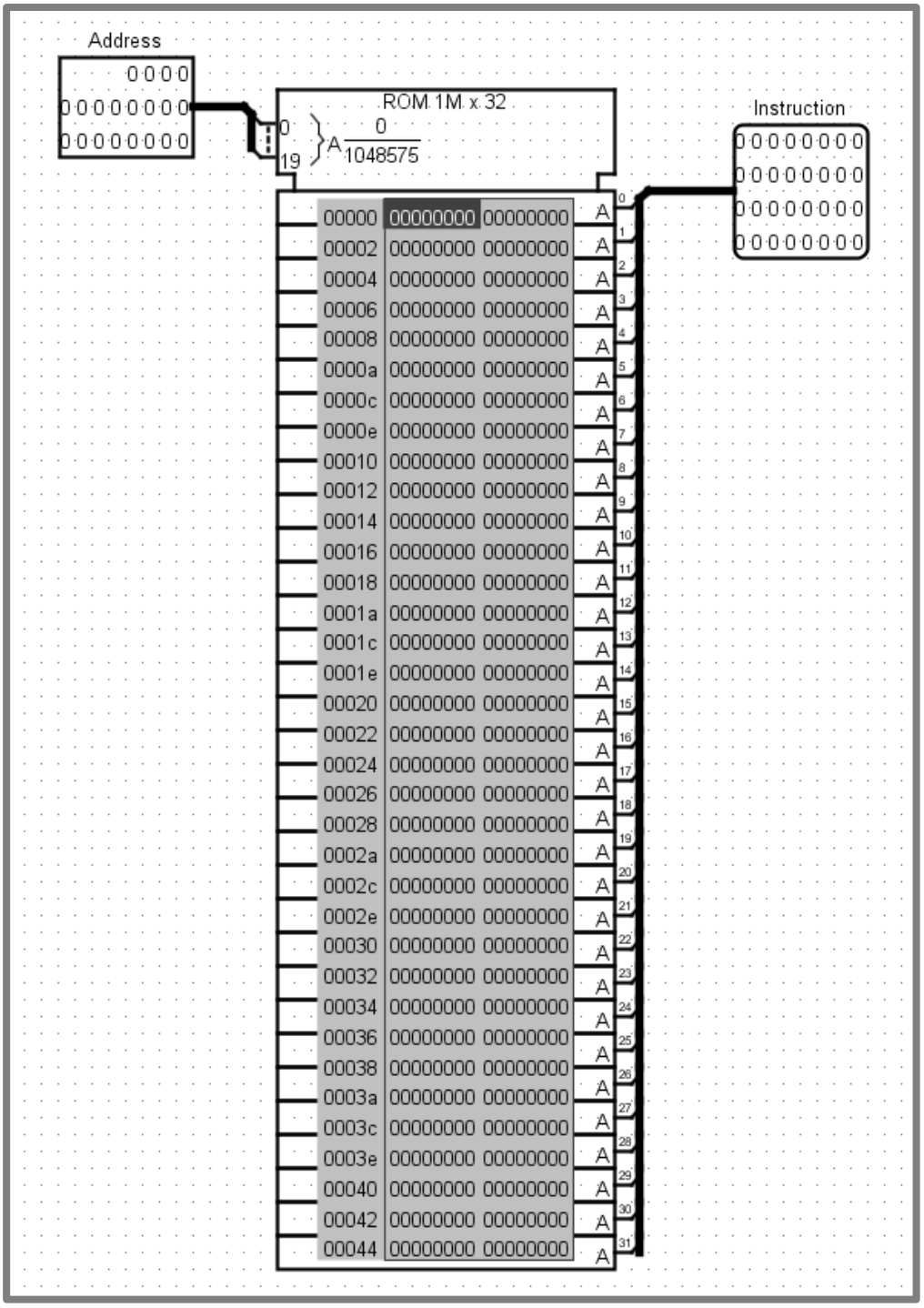
Purpose of the Processor

The Instruction Memory performs **only one job**:

> It takes the **address from the PC (Program Counter)** and returns the **32-bit instruction** stored at that address.

This instruction is then sent to the **Instruction Splitter**, which breaks it down for decoding and execution.

Visual Representation:



Key Components in the Circuit

Address (Input):

- Comes from the **Program Counter** (PC).
- It's a 20-bit input, allowing addressing up to `2^20` locations.

ROM 1M × 32 (Component):

- A built-in ROM block in Logisim.
- Each address points to a 32-bit instruction.
- The ROM is **preloaded** with binary machine instructions before simulation.

Instruction (Output):

- A 32-bit wide output line.
- The full instruction is passed to the next stage (Instruction Splitter).

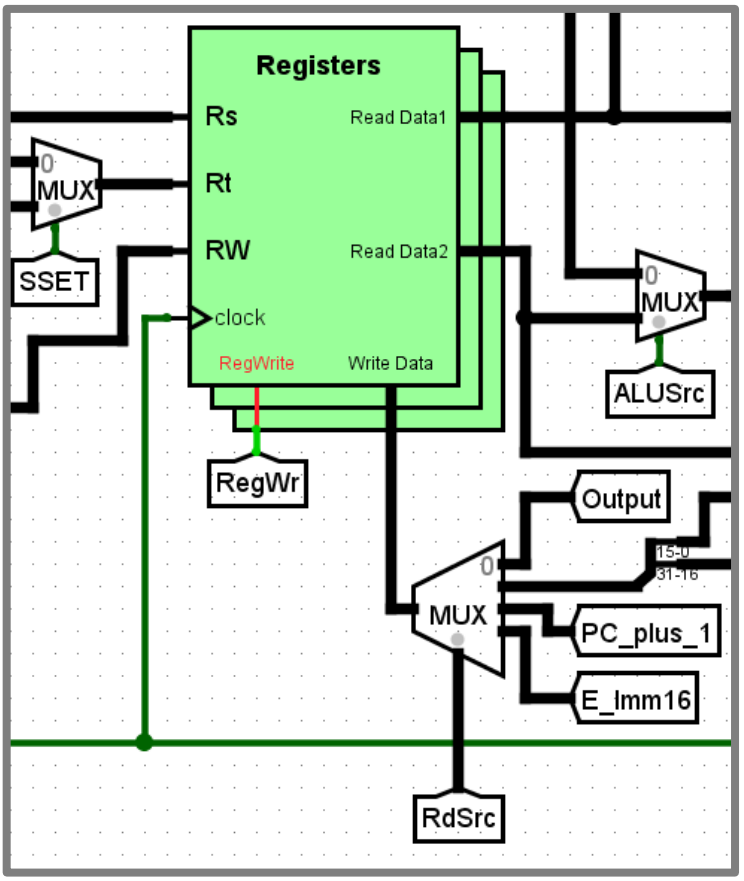
Why ROM?

- ROM is used instead of RAM because **instructions do not change at runtime**.
- It ensures that the processor always reads the same set of instructions unless manually updated before simulation.

Summary

Feature	Description
Type	ROM (Read-Only Memory)
Size	1M x 32 bits
Input	20-bit address from the PC
Output	32-bit instruction
Role	Stores the program instructions
Modifiability	Set before simulation; not changeable at runtime

5. Register File



This section explains the **Register File** component, both as part of the **main data path** and in terms of its **implementation**.

Overview

The **Register File** contains **32 general-purpose 32-bit registers**, labeled R0 to R31.

- R0 is **hardwired to 0** and cannot be modified.
- The remaining registers (R1–R31) are used for storing and retrieving data during instruction execution.

We interface with the Register File using **three register addresses** and a **write-enable signal**, as well as separate data lines for input and output.

A. Inputs

Input	Purpose
Rs	Register source 1, used to read data into **BusA** (always corresponds to S1)
Rt	Register source 2, used to read data into **BusB** (can be S2 or Rd, depending on the instruction type)

Rw	Register destination, the register to write data into (always corresponds to `Rd`)
Write Data	The actual 32-bit value to be stored in `RW`, if enabled

B. Outputs

Output	Purpose
Read Data 1	Data from register `Rs`, sent to **BusA**
Read Data 2	Data from register `Rt`, sent to **BusB**

These outputs are continuously available once the register numbers are selected through the MUXes.

C. Control Signals

Signal	Role
RegWr	Write Enable, if high, allows writing data into `RW`
SSET	Specific to the `SSET` instruction — used when we need data from `Rd` as a source
RdSrc	A 3-bit control line that determines **which data** to write into `RW`: • ALU result • Data from memory • Concatenated value of `Imm16` and `ReadData2` • `PC+1` extended to 32 bits • Extended `Imm16`

Note: A detailed explanation of `RdSrc` control and its encoding is covered in the Control Unit section.

D- Implementation Details

Registers: The core is built using **32 individual 32-bit registers**.

Hardwiring R0: The `R0` register output is fixed to zero, and write operations to it are ignored.

Reading Mechanism:

- Two **5-bit MUXes** are used to select `Rs` and `Rt`.
- The selected registers drive the **Read Data1** and **Read Data2** outputs.

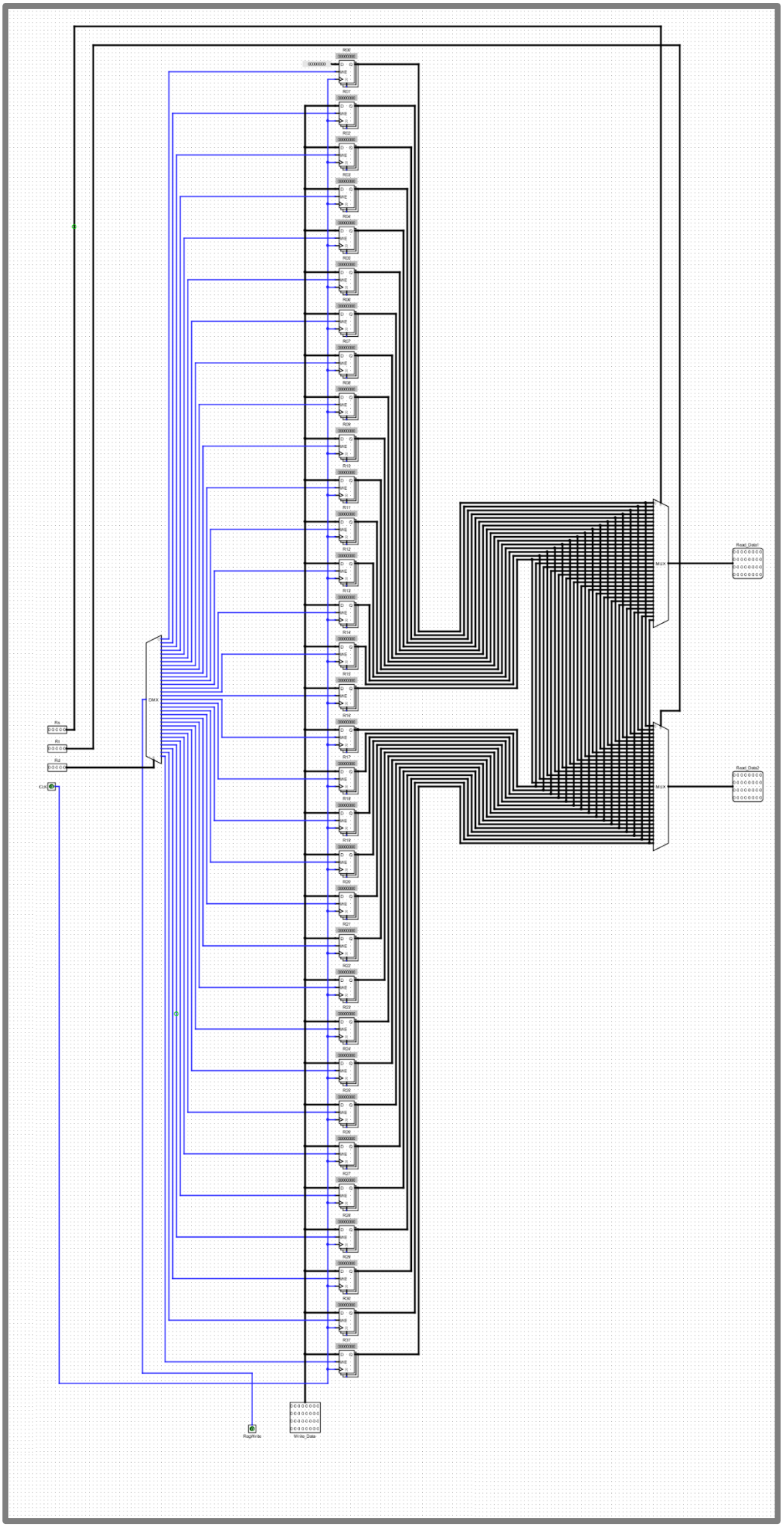
Writing Mechanism:

- A **DEMUX** controlled by `RW` and enabled by `RegWr` directs the write value to the appropriate register.
- Only one register can be written at a time, and only if `RegWr` is active.

Summary

Component	Description
Register Count	32
Bit Width	32 bits per register
Special Behavior	R0 always outputs 0; cannot be overwritten
Write Logic	Controlled by `RegWr`, `RW`, and `RdSrc`
Read Logic	MUX-based selection for Rs and Rt

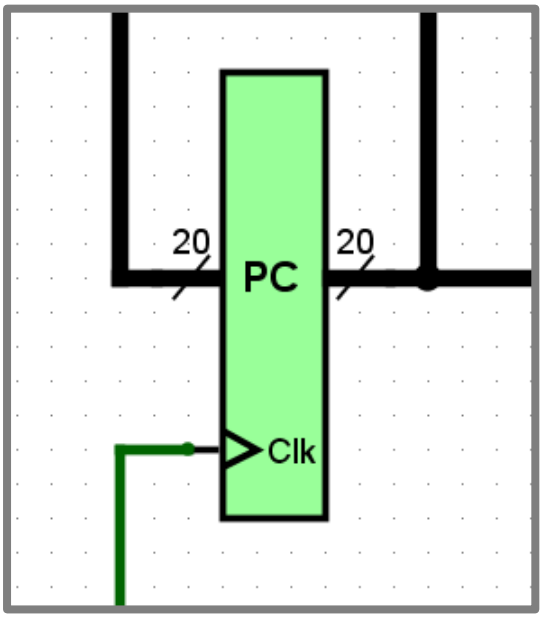
On the next page, an image of the implementation in logism.

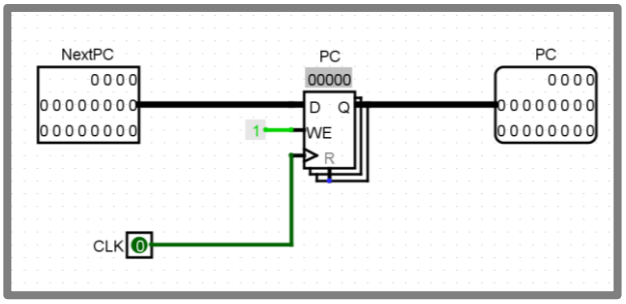

R00
R01
R02
R03
R04
R05
R06
R07
R08
R09
R10
R11
R12
R13
R14
R15
R16
R17
R18
R19
R20
R21
R22
R23
R24
R25
R26
R27
R28
R29
R30
R31
DMX
MUX
MUX
Read_Data1
Read_Data2
Ra
Rb
Rd
CLK
RegWrite
Write_Data

6. PC & NextPC

This section explains the **Program Counter (PC)** and the **NextPC** logic responsible for instruction sequencing, branching, and jumping within the processor.

Program Counter (PC):



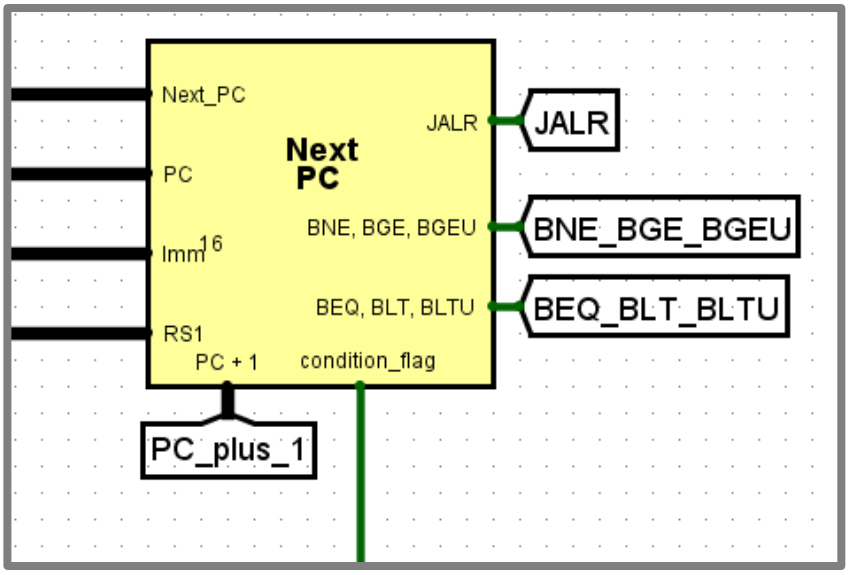


The **Program Counter (PC)** is a **20-bit register** that holds the **address of the current instruction** being executed. It is updated at the end of each clock cycle, based on the value of **NextPC**.

Behavior:

- In the absence of branching or jumping, the PC is incremented normally: `PC = PC + 1`.
- If the instruction is a jump or branch, the PC is updated with a new target address computed by the NextPC logic.

NextPC (as seen in the Data Path):



The **NextPC unit** determines the value of the next instruction address. Its role is to calculate:

- The **sequential address** (PC + 1),
- A **branch target address**, or
- A **jump address**, depending on the instruction type and condition flags.

Inputs:

- **PC** (20-bit): The current program counter.
- **Immediate16** (16-bit): Used in branch or jump offset calculation.
- **RS1**: A source register used during JALR (jump and link register) instructions.

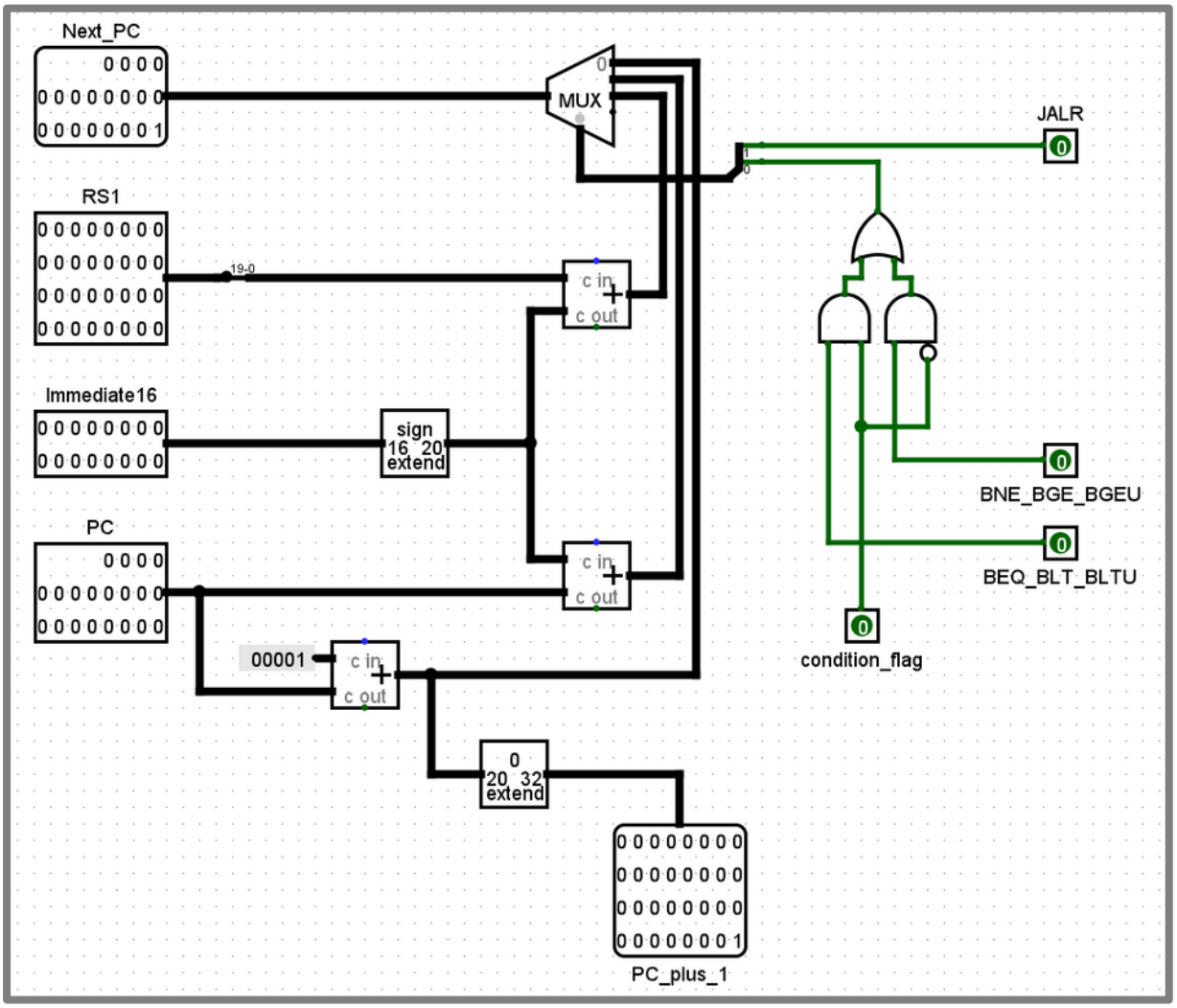
Control Signals:

Signal	Description
JALR	Activates the jump logic for **JALR** instructions.
BNE_BGE_BGEU	Enables branching for **BNE**, **BGE**, and **BGEU** if the

	condition is **false**.
`BEQ_BLT_BLTU`	Enables branching for **BEQ**, **BLT**, and **BLTU** if the condition is **true**.
`Condition_flag`	Evaluates whether the branch condition is satisfied (used alongside the above).

Note: The correct use of `condition_flag` along with the branch signals ensures only the correct instruction type triggers a PC update.

NextPC Implementation Details



Inputs Recap:

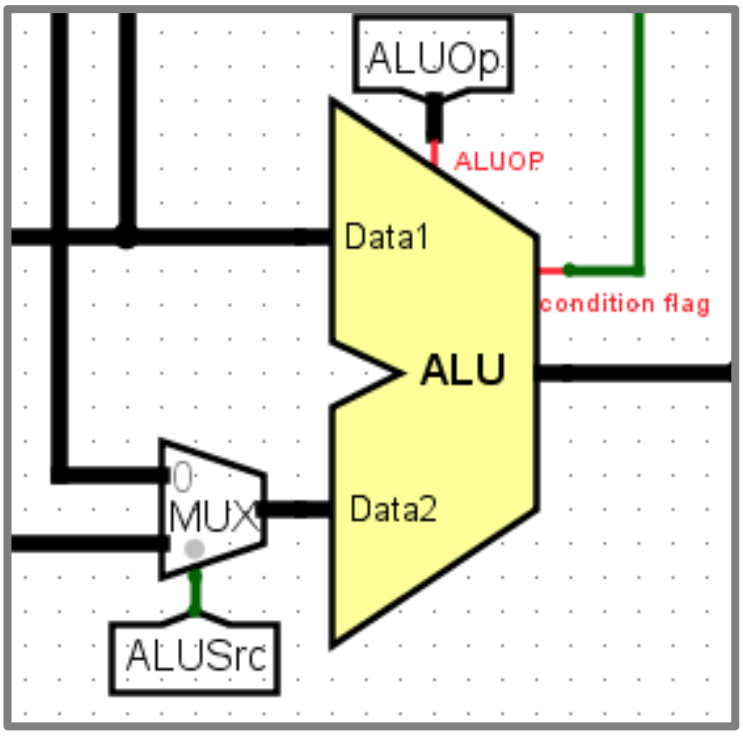
- `PC` → current program counter

- `RS1` → register for base jump address (used in JALR)
- `Immediate16` → offset used in branch or jump

Outputs:

- `NextPC` → the computed next program counter value
- `PC_plus_1` → the sequential address (i.e., `PC + 1`) used for:
  - The default next instruction
  - Storing into `Rd` in JALR cases

7. Arithmetic & Logic Unit (ALU)



This component is to perform all arithmetic and logical operations on our RISC single-cycle processor. It takes the data from two registers or a register with an immediate value and produces results based on the selected operation. The ALU is one of the most crucial components in our processor design as it handles all computational tasks.

Component Structure

To be clear, our ALU has two inputs, one control signal, and two outputs.

Inputs

- **Data1**: is the data from the source register (Rs). This 32-bit value comes directly from the register file and represents the first operand for any operation the ALU needs to perform.
- **Data2**: can be the data from the target register (Rt) or the extended 16-bit immediate value (Imm16). This flexibility allows us to perform operations between two registers or between a register and an immediate value, which is essential for supporting various instruction formats in our RISC architecture.

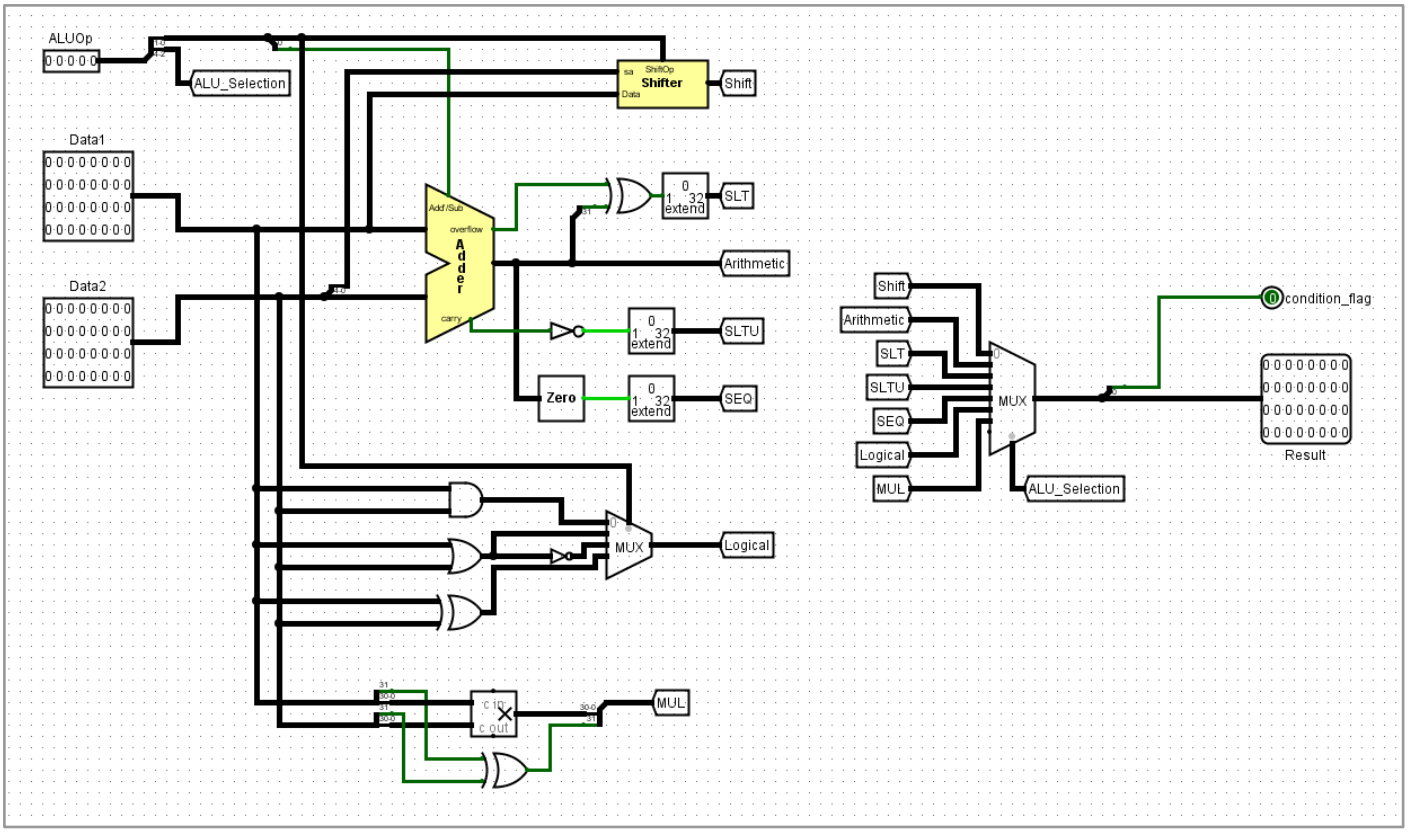
Outputs

- **The main output of the ALU**: the 32-bit result of any operation the ALU can perform. This output is connected to various parts of the processor depending on the instruction type – it might be written back to the register file, used as a memory address, or employed for branch condition evaluation.
- **Condition flag**: is the first bit from the result of the SLT (Set Less Than) or the SEQ (Set Equal) operations. This flag is particularly important for conditional branches and comparison instructions. There are some other details we discussed earlier in the NextPC component documentation. The condition flag indicates whether certain conditions (like equality or less-than comparison) have been met during execution.

Control Signal

- **ALUOp**: This 5-bit signal selects which operation we want its result as the output of the ALU. The complete 5-bit signal gives us the ability to support up to 32 different operations, though we don't use all combinations in our current implementation.

ALU Implementation Details



Here we have two sub-components to make the ALU fully functional, the Adder and the Shifter, and they are explained in detail in separate sections. Our ALU implementation incorporates specialized hardware for each type of operation category, with multiplexers to select the appropriate result based on the control signals.

I will take the main cases of the ALU and explain them and how they would work:

Shift Operations

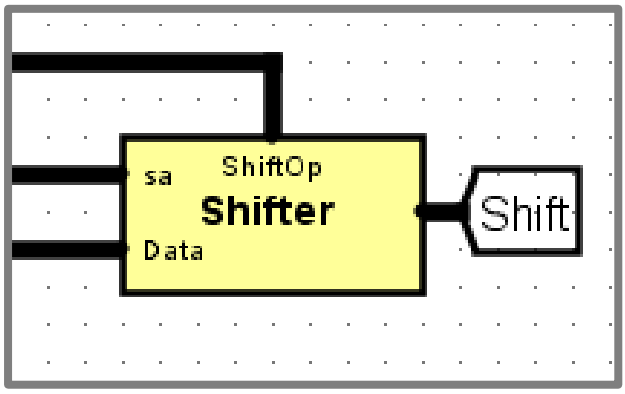
In the case of **shift**, we take the first two bits of the **ALUOp** to select the type of shifting, and the remaining three bits are used to select the result of the ALU as a whole. The shift operation types are:

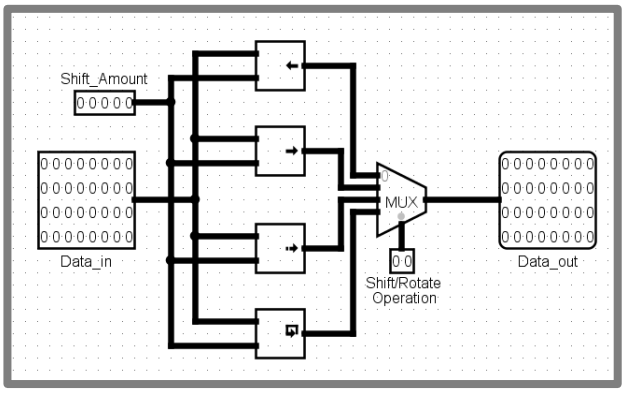
- **00**: Shift Left Logical (SLL) - shifts the bits left and fills with zeros
- **01**: Shift Right Logical (SRL) - shifts the bits right and fills with zeros

- **10**: Shift Right Arithmetic (SRA) - preserves the sign bit while shifting right
- **11**: Rotate Right (ROR) - wraps around the bits that are shifted out

Data1 is the data to be shifted, and Data2's first five bits are the shift amount (0-31 positions). The Shifter component handles all these operations and passes the shifted result to the ALU's output selection multiplexer.

And here is the implementation of the shifter block:





Arithmetic Operations

In case of addition or subtraction, here comes the Adder component and the first bit of the **ALUOp**, which selects whether to add or subtract:

- When the first bit is **0**: Addition operation (Data1 + Data2)
- When the first bit is **1**: Subtraction operation (Data1 - Data2)

The Adder not only produces the sum/difference result but also generates important status signals:

- **Overflow**: indicates when the result cannot be correctly represented in the 32-bit signed format
- **Carry**: shows when there's a carry-out during addition or a borrow during subtraction
- **Zero**: set when all bits of the result are zero (this is useful for the SEQ instruction)

Further explanation of the outputs of the Adder is provided in the Adder section of this documentation.

Logical Operations

In case of **Logical operations**, they are straightforward gates to make every logical operation needed, and the first two bits of the **ALUOp** select which logic operation is taken:

- **00**: AND - performs bitwise AND between Data1 and Data2
- **01**: OR - performs bitwise OR between Data1 and Data2
- **10**: NOR - performs bitwise NOR between Data1 and Data2
- **11**: XOR - performs bitwise XOR between Data1 and Data2

These operations are implemented using simple logic gates that operate on each bit position independently. The outputs of these gates are then fed to the ALU's output selection multiplexer.

Multiplication Operation

In the multiplication operation, the most significant bit (sign bit) of each operand is handled separately to determine the sign of the result. Specifically, the lower 31 bits of both input operands are multiplied to produce the magnitude of the result. The sign bit of the output is then computed by

performing a bitwise XOR on the sign bits of the two operands, yielding `0` for a positive result and `1` for a negative result. Finally, the calculated sign bit is concatenated with the 31-bit product to form the complete 32-bit signed output.

Result Selection

Finally, **ALU Selection** is the signal that chooses the final result from all the operation units (Adder, Shifter, Logic Unit, Multiplier). This selection is controlled by the last three bits of the **ALUOp**. The specific mapping is:

- **000**: Select Shifter result
- **001**: Select Adder result (addition/subtraction)
- **010**: Select SLT result (Set Less Than)
- **011**: Select SLTU result (Set Less Than Unsigned)
- **100**: Select SEQ result (Set Equal)
- **101**: Select Logic Unit result
- **110**: Select Multiplier result
- **111**: Reserved for future expansion

This multiplexer-based selection mechanism allows us to implement a rich set of operations while maintaining a clean, modular design.

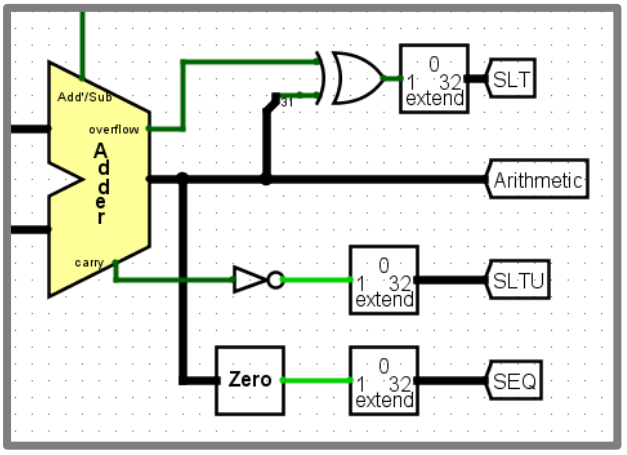
Integration with Other Components

The ALU is central to the processor's data path and interacts with several other components:

1. **Register File**: Provides Data1 and potentially Data2 inputs
2. **Immediate Extension Unit**: May provide the extended immediate value for Data2

3. **Control Unit**: Provides the ALUOp signal based on instruction decoding
4. **Memory Unit**: Uses the ALU result for address calculation
5. **NextPC Unit**: Uses the condition flag for branch decisions

8. Adder Block



Overview

The Adder block is a critical subcomponent of our ALU that handles all addition and subtraction operations in our RISC single-cycle processor. It performs the fundamental arithmetic operations that underpin many of the processor's capabilities, from basic math to address calculations and comparisons.

Component Structure

As shown in the circuit diagram, the Adder has a compact but sophisticated design that enables both addition and subtraction while generating crucial status flags.

Inputs

- **Data1**: The first 32-bit operand, typically coming from the source register (Rs)
- **Data2**: The second 32-bit operand, coming from either the target register (Rt) or an immediate value
- **Add_Sub**: Control signal that determines the operation mode
    - When **0**: Addition operation (Data1 + Data2)
    - When **1**: Subtraction operation (Data1 - Data2)

Outputs

- **Data_out**: The 32-bit result of the addition or subtraction operation
- **carry_flag**: A single-bit output indicating carry status
- **overflow_flag**: A single-bit output indicating arithmetic overflow

Functional Description

The Adder performs two core operations based on the Add_Sub control signal:

Addition Mode (Add_Sub = 0)

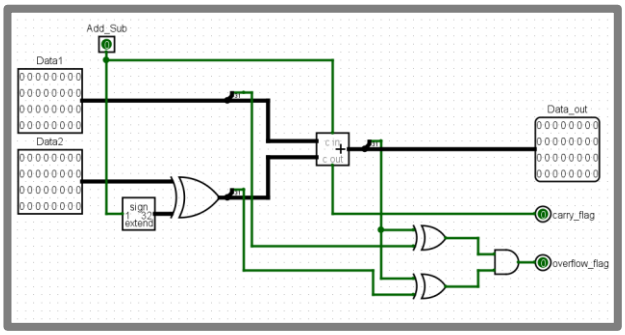
When the Add_Sub control signal is 0, the Adder directly adds Data1 and Data2:

```
Data_out = Data1 + Data2
```

Subtraction Mode (Add_Sub = 1)

When the Add_Sub control signal is 1, the Adder performs subtraction by adding Data1 to the two's complement of Data2:

```
Data_out = Data1 + (~Data2 + 1) = Data1 - Data2
```



As visible in the implementation diagram, this is accomplished by:

1. Inverting all bits of Data2 (using XOR gates with Add_Sub)
2. Setting the carry-in of the adder to the Add_Sub value (effectively adding 1 when in subtraction mode)

Status Flag Generation

For our design, we have some other outputs, two to be specific: the Carry and Overflow flags. These flags are crucial for detecting special conditions and supporting various instructions.

Carry Flag

The carry_flag is set to 1 when there is a carry-out generated from the most significant bit position during the addition/subtraction operation, and 0 otherwise.

- During **addition**: Indicates when the result exceeds the maximum representable unsigned value
- During **subtraction**: Works as a "borrow" indicator - when set to 0, it indicates that a borrow occurred

As shown in the diagram, the carry_flag is directly connected to the carry-out signal from the adder circuit.

Overflow Flag

The overflow_flag is set to 1 when the operation produces a result that cannot be correctly represented in two's complement signed format. This occurs when:

- Adding two positive numbers yields a negative result
- Adding two negative numbers yields a positive result
- Subtracting a negative number from a positive number yields a negative result
- Subtracting a positive number from a negative number yields a positive result

As visible in the implementation diagram, we detect overflow by:

1. XORing the sign bit (most significant bit) of Data1 with the sign bit of the result
2. XORing the sign bit of Data2 (after potential inversion for subtraction) with the sign bit of the result
3. ANDing these two XOR results together

This implementation precisely captures the overflow condition: the sign bit of the result differs from the expected sign based on the operands' signs.

Applications in Comparison Operations

These flags are not merely status indicators but are actively used to implement several crucial comparison operations:

1. Set Less Than (SLT)

The SLT operation determines if Data1 < Data2 for signed numbers. To properly detect this condition, we need to account for both normal comparison and overflow cases:

For SLT to be true (Data1 < Data2), one of two conditions must be met:

- **Case A**: sign bit = 0 and overflow = 1
- **Case B**: sign bit = 1 and overflow = 0

This can be expressed as: sign bit ≠ overflow

Therefore, we implement SLT by XORing the sign bit of the result (the MSB of Data_out) with the overflow flag. When this XOR produces 1, it indicates that Data1 is less than Data2 in signed comparison.

2. Set Less Than Unsigned (SLTU)

For unsigned comparison, the logic is simpler. We use the carry_flag from the subtraction operation:

- When performing Data1 - Data2 (with Add_Sub = 1):
    - If carry_flag = 1: No borrow occurred, meaning Data1 ≥ Data2
    - If carry_flag = 0: A borrow occurred, meaning Data1 < Data2

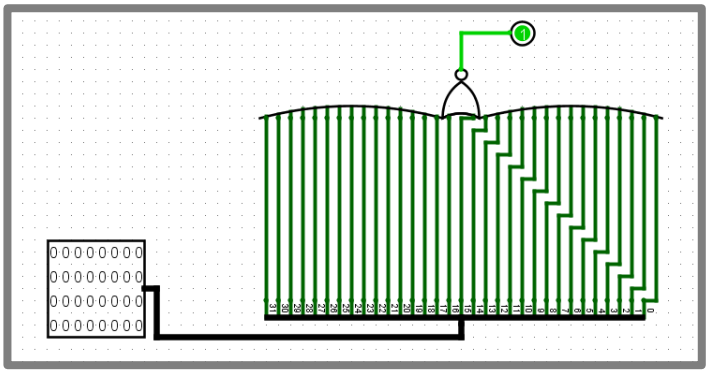
Therefore, for SLTU (unsigned less than), we simply use the inverted carry_flag after a subtraction operation. When the inverted carry_flag is 1, it indicates that Data1 is less than Data2 in an unsigned comparison.

3. Set Equal (SEQ)

The SEQ operation determines if Data1 = Data2. This is implemented by checking if the result of subtraction (Data1 - Data2) is zero:

- If all bits of Data_out are 0 after subtraction, then Data1 = Data2
- Any non-zero bit indicates inequality

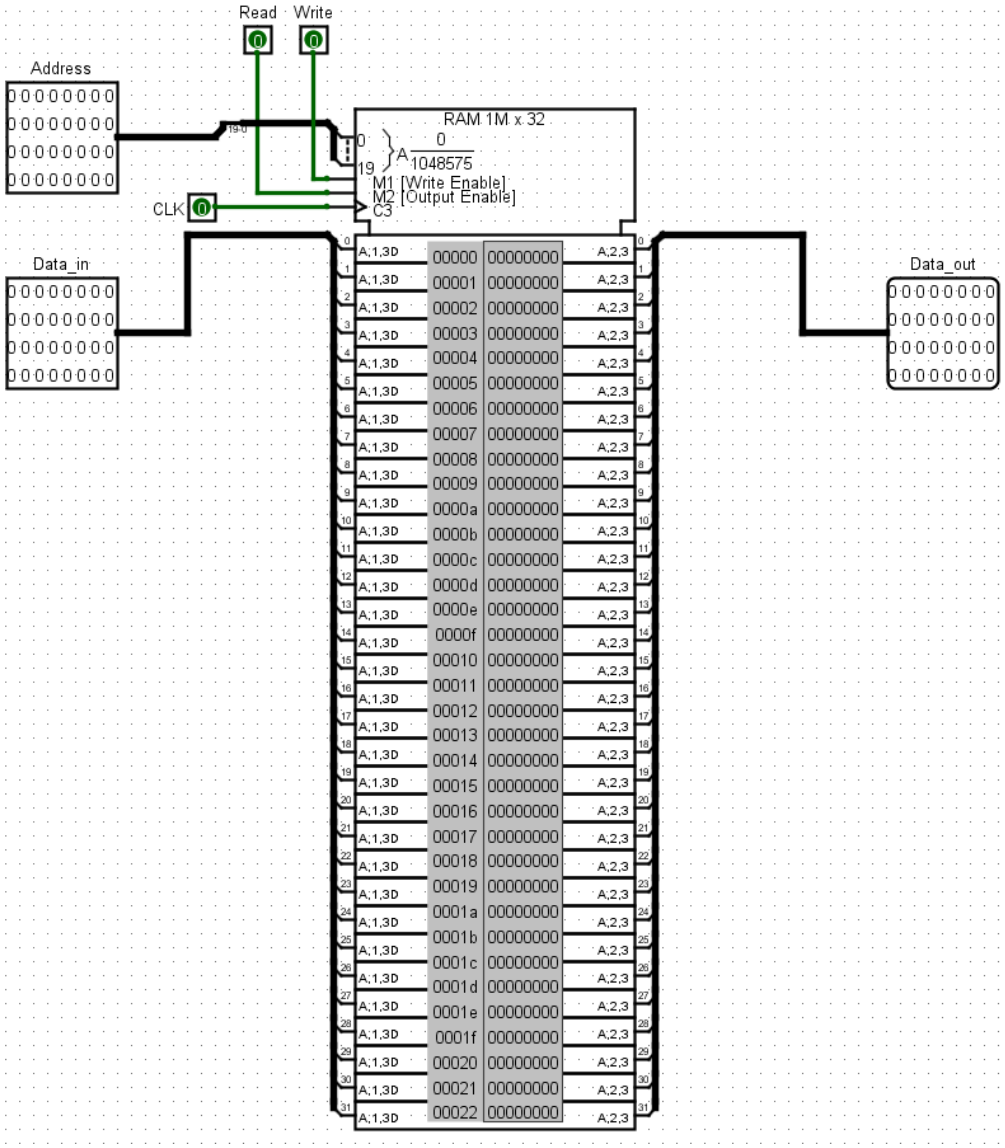
The zero detection is performed by NORing all bits of the Data_out together. When this NOR produces 1, it indicates that Data1 equals Data2.



9. Data Memory

Overview

The Data Memory is a fundamental component of our RISC single-cycle processor that serves as the primary storage for data during program execution. It is implemented as a 1Mx32 RAM (Random Access Memory), which means it contains 1 million (2^20) addressable locations, each capable of storing a 32-bit word. This memory provides temporary storage for variables, arrays, and other data structures that programs need to access during runtime.



Interface Specifications

Inputs

1. **Address Input:**
    - **Width**: 32 bits (though only the lower 20 bits are used)
    - **Function**: Specifies the memory location to read from or write to
    - **Valid Range**: 0x00000 to 0xFFFFF (0 to 1,048,575 in decimal)

Note: We only utilize the first (least significant) 20 bits of the address because our memory contains 2^20 addressable words. The upper 12 bits are ignored by the memory hardware.

2. **Data_in**:

- **Width**: 32 bits

- **Function**: Contains the data to be written to memory during store operations
- **Valid Range**: Any 32-bit value (0x00000000 to 0xFFFFFFFF)
- **Usage**: This input is only used when the MemWr signal is active

Output

Data_out:

- **Width**: 32 bits
- **Function**: Delivers the data read from the specified memory address
- **Timing**: Data becomes available shortly after the Address is provided and MemRd is activated
- **Default State**: When no read operation is being performed (MemRd = 0), the output maintains its previous value

Control Signals

1. **MemRd (Memory Read)**:
   - **Width**: 1 bit
   - **Function**: Activates the read operation from memory
   - **Active State**: When set to 1, the memory outputs the data from the specified address onto Data_out
   - **Inactive State**: When set to 0, no read operation occurs, and the Data_out maintains its previous state
   - **Usage**: Typically activated during load instructions (LW, LH, LB, etc.)
2. **MemWr (Memory Write)**:
   - **Width**: 1 bit
   - **Function**: Activates the write operation to memory

- **Active State**: When set to 1, the data present on Data_in is written to the specified address
- **Inactive State**: When set to 0, no write operation occurs, and memory content remains unchanged
- **Usage**: Typically activated during store instructions (SW, SH, SB, etc.)

Operational Behavior

The Data Memory operates in three primary modes:

1. Read Mode (MemRd = 1, MemWr = 0)

When the processor needs to retrieve data from memory (e.g., during a load instruction):

1. The memory address is placed on the Address input
2. The MemRd signal is asserted (set to 1)
3. The memory retrieves the 32-bit word at the specified address
4. The retrieved data is placed on the Data_out line
5. The processor or destination register captures this data

2. Write Mode (MemRd = 0, MemWr = 1)

When the processor needs to store data to memory (e.g., during a store instruction):

1. The memory address is placed on the Address input
2. The data to be stored is placed on the Data_in input
3. The MemWr signal is asserted (set to 1)
4. The memory writes the 32-bit word from Data_in to the specified address
5. The memory content at that address is updated accordingly

3. Idle Mode (MemRd = 0, MemWr = 0)

When no memory operation is being performed:

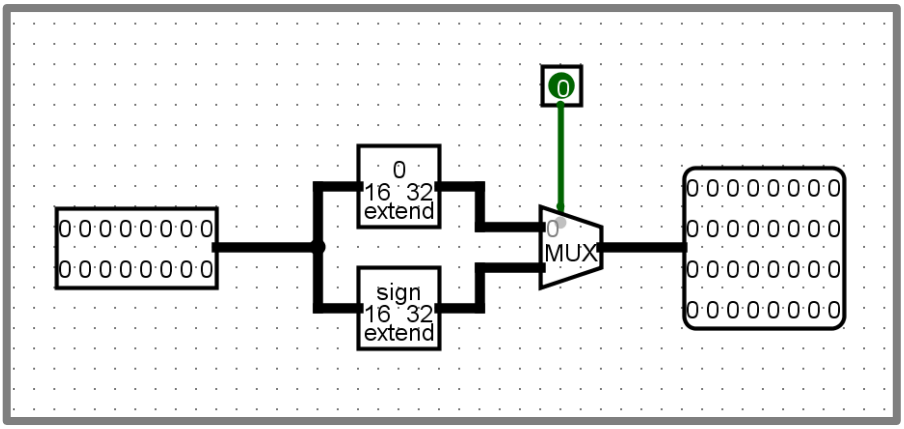
1. Both control signals are deasserted (set to 0)
2. No read or write operation occurs
3. Memory content remains unchanged
4. Data_out maintains its previous value

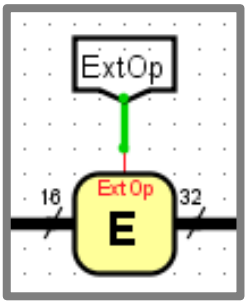
Integration with Other Components

The Data Memory connects to several other processor components:

1. **ALU:**
    - Provides the memory address (typically the result of a base+offset calculation)
    - This address is fed to the Address input of the memory
2. **Register File**:
    - During store operations (SW, SH, SB): Provides the data to be written to memory via Data_in
    - During load operations (LW, LH, LB): Receives the data read from memory via Data_out
3. **Control Unit**:
    - Generates the MemRd and MemWr signals based on the current instruction
    - Ensures proper timing of memory operations

10. Extender





The Extender module is a key component in the datapath that performs bit-width conversion, transforming 16-bit immediate values into 32-bit values suitable for arithmetic and logical operations.

Functionality

The module takes a 16-bit immediate value as input and expands it to 32 bits using one of two extension methods, selected by the ExtOp control signal:

- **Zero Extension (ExtOp = 0)**: Fills the upper 16 bits with zeros
- **Sign Extension (ExtOp = 1)**: Replicates the most significant bit (sign bit) of the input across the upper 16 bits

Implementation

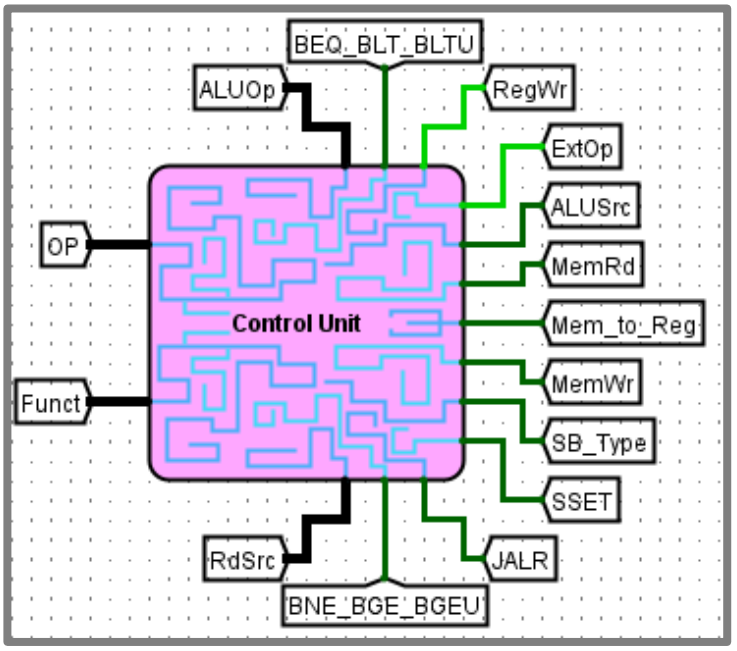
As shown in the diagram, the Extender consists of:

- Parallel zero and sign extension circuits
- A multiplexer (MUX) that selects between these two results based on the ExtOp signal
- Input and output buses for data transfer

The module is represented in the schematic by the yellow "E" block, with the control signal clearly marked as "ExtOp".

11. Control Unit

The control unit is a critical component in the design of the MIPS processor, responsible for directing the operation of the processor. It interprets instructions fetched from memory and generates the necessary control signals to execute those instructions. The control unit ensures that data flows correctly between the processor's components, such as the ALU, registers, and memory.

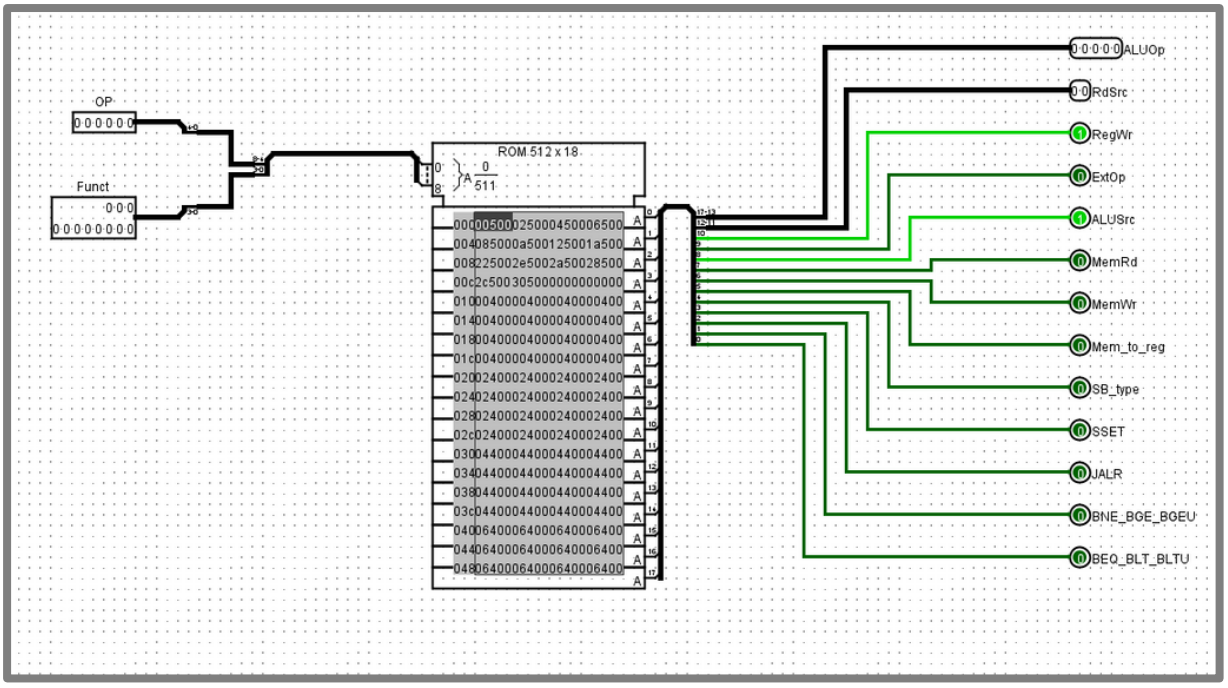


Using ROM for Signal Control:

In the design of the control unit for our MIPS-based processor, a ROM is used to store the control signals corresponding to each instruction. The ROM is indexed using a 9-bit address, which is typically composed of:

- **5 bits** for the **Opcode** (bits [8:4])
- **4 bits** for the **function code** (bits [3:0])

The control unit utilizes Read-Only Memory (ROM) to generate control signals. The ROM is pre-programmed with a set of instructions that correspond to various opcodes and function codes. When an instruction is fetched, the processor uses the opcode and function code as inputs to the ROM. The ROM outputs specific control signals based on this input, dictating the operation of the processor for that particular instruction.



By leveraging ROM in this manner, the design simplifies the control unit's logic, making it more efficient and easier to implement. This approach ensures that each instruction is executed with precise control signals, contributing to the overall efficiency of the processor.

Control Signal Table

	Instruction	OP	F	ALUOp	RdSrc	RegWr	ExtOp	ALUSrc	MemRd	MemWr	Mem to Reg	SB-Type	SSET	JALR	BNE_BGE_BGEU	BEQ_BLT_BLTU	Address (Hex)	Control Signals (Binary)	Control Signals (Hex)
R-Type	SLL	0	0	00000	00	1	X	1	0	0	0	0	0	0	0	0	000	00000 00 10100000000	00500
	SRL	0	1	00001	00	1	X	1	0	0	0	0	0	0	0	0	001	00001 00 10100000000	02500
	SRA	0	2	00010	00	1	X	1	0	0	0	0	0	0	0	0	002	00010 00 10100000000	04500
	ROR	0	3	00011	00	1	X	1	0	0	0	0	0	0	0	0	003	00011 00 10100000000	06500
	ADD	0	4	001X0	00	1	X	1	0	0	0	0	0	0	0	0	004	00100 00 10100000000	08500
	SUB	0	5	001X1	00	1	X	1	0	0	0	0	0	0	0	0	005	00101 00 10100000000	0A500
	SLT	0	6	010X1	00	1	X	1	0	0	0	0	0	0	0	0	006	01001 00 10100000000	12500
	SLTU	0	7	011X1	00	1	X	1	0	0	0	0	0	0	0	0	007	01101 00 10100000000	1A500
	SEQ	0	8	100X1	00	1	X	1	0	0	0	0	0	0	0	0	008	10001 00 10100000000	22500
	XOR	0	9	10111	00	1	X	1	0	0	0	0	0	0	0	0	009	10111 00 10100000000	2E500
	OR	0	10	10101	00	1	X	1	0	0	0	0	0	0	0	0	00A	10101 00 10100000000	2A500
	AND	0	11	10100	00	1	X	1	0	0	0	0	0	0	0	0	00B	10100 00 10100000000	28500
	NOR	0	12	10110	00	1	X	1	0	0	0	0	0	0	0	0	00C	10110 00 10100000000	2C500
	MUL	0	13	110XX	00	1	X	1	0	0	0	0	0	0	0	0	00D	11000 00 10100000000	30500
I-Type	SLLI	1	-	00000	00	1	X	0	0	0	0	0	0	0	0	0	010 -> 01F	00000 00 10000000000	00400
	SRLI	2	-	00001	00	1	X	0	0	0	0	0	0	0	0	0	020 -> 02F	00001 00 10000000000	02400
	SRAI	3	-	00010	00	1	X	0	0	0	0	0	0	0	0	0	030 -> 03F	00010 00 10000000000	04400
	RORI	4	-	00011	00	1	X	0	0	0	0	0	0	0	0	0	040 -> 04F	00011 00 10000000000	06400
	ADDI	5	-	001X0	00	1	1	0	0	0	0	0	0	0	0	0	050 -> 05F	00100 00 11000000000	08600
	SLTI	6	-	010X1	00	1	1	0	0	0	0	0	0	0	0	0	060 -> 06F	01001 00 11000000000	12600
	SLTIU	7	-	011X1	00	1	0	0	0	0	0	0	0	0	0	0	070 -> 07F	01101 00 10000000000	1A400
	SEQI	8	-	100X1	00	1	1	0	0	0	0	0	0	0	0	0	080 -> 08F	10001 00 11000000000	22600
	XORI	9	-	10111	00	1	0	0	0	0	0	0	0	0	0	0	090 -> 09F	10111 00 10000000000	2E400
	ORI	10	-	10101	00	1	0	0	0	0	0	0	0	0	0	0	0A0 -> 0AF	10101 00 10000000000	2A400
	ANDI	11	-	10100	00	1	0	0	0	0	0	0	0	0	0	0	0B0 -> 0BF	10100 00 10000000000	28400
	NORI	12	-	10110	00	1	0	0	0	0	0	0	0	0	0	0	0C0 -> 0CF	10110 00 10000000000	2C400
	SET	13	-	XXXXX	11	1	1	X	0	0	X	0	0	0	0	0	0D0 -> 0DF	00000 11 11000000000	01E00
	SSET	14	-	XXXXX	01	1	X	X	0	0	X	0	1	0	0	0	0E0 -> 0EF	00000 01 10000001000	00C08
	JALR	15	-	XXXXX	10	1	X	X	0	0	X	0	0	1	0	0	0F0 -> 0FF	00000 10 10000000100	01404
	LW	16	-	001X0	00	1	1	0	1	0	1	0	0	0	0	0	100 -> 10F	00100 00 11010100000	086A0
SB-Type	SW	17	-	001X0	00	0	1	0	0	1	X	1	0	0	0	0	110 -> 11F	00100 00 01001010000	08250
	BEQ	18	-	100X1	00	0	X	1	0	0	X	1	0	0	0	1	120 -> 12F	10001 00 00100010001	22111
	BNE	19	-	100X1	00	0	X	1	0	0	X	1	0	0	1	0	130 -> 13F	10001 00 00100010010	22112
	BLT	20	-	010X1	00	0	X	1	0	0	X	1	0	0	0	1	140 -> 14F	01001 00 00100010001	12111
	BGE	21	-	010X1	00	0	X	1	0	0	X	1	0	0	1	0	150 -> 15F	01001 00 00100010010	12112
	BLTU	22	-	011X1	00	0	X	1	0	0	X	1	0	0	0	1	160 -> 16F	01101 00 00100010001	1A111
	BGEU	23	-	011X1	00	0	X	1	0	0	X	1	0	0	1	0	170 -> 17F	01101 00 00100010010	1A112

How the Control Unit Produces the Control Signals:

1. Instruction Fetch and Decode

When an instruction is fetched from memory, it's decoded to extract:

The opcode field (OP) identifies the general instruction type (e.g., R-Type, I-Type). For R-Type instructions,

the function field (F) is also extracted — this specifies the exact operation (e.g., ADD, SUB, SLL, etc.).

2. Address Calculation for ROM

The combination of OP and F forms an address used to index the ROM. This address uniquely identifies the instruction. For instance:

OP = 0, F = 4 → Address = 004 (Hex) → R-Type ADD

OP = 0, F = 10 → Address = 00A (Hex) → R-Type OR

This address is essentially used as the ROM lookup key.

3. ROM Output (Control Word)

Each address in the ROM contains a control word, which is a

binary string representing the control signals for that

instruction.

For example:

ADD: 00100 00 10100000000 (binary) → 08500 (hex)

SUB: 00101 00 10100000000 (binary) → 0A500 (hex)

This control word encodes signals like:

ALUOp: Selects the ALU operation (5 bits)

RdSrc, RegWr: Register destination and write enable

ExtOp, ALUSrc: ALU operand source control

MemRd, MemWr, MemtoReg: Memory access and data routing

Branch and jump signals: SB-Type, SSET, JALR, etc.

4. Signal Distribution

The decoded control signals are then distributed to various processor components:

ALU: Receives ALUOp, ALUSrc

Register File: Receives RegWr, RdSrc

Memory Unit: Receives MemRd, MemWr, MemtoReg

Control Flow: Receives JALR, branch type signals, etc.

12. Testing

The following table contains a sample test program we ran on our CPU.

For each instruction, we list:

- The expected register value after execution
- The actual value obtained during the simulation

PC (Hex)	Instruction	Ins. Code (Hex)	Expected Value	Actual Value	Notes
0x00000	SET R1, 0x0384	0384004D	R1 = 0x00000384	R1 = 0x00000384	
0x00001	SET R8, 0x1234	1234020D	R8 = 0x00001234	R8 = 0x00001234	
0x00002	SSET R8, 0x5678	5678020E	R8 = 0x12345678	R8 = 0x12345678	
0x00003	ADDI R5, R1, 20	00140945	R5 = 0x398	R5 = 0x398	
0x00004	XOR R3, R1, R5	012508C0	R3 = 0x1C	R3 = 0x1C	
0x00005	ADD R4, R8, R3	00834100	R4 = 0x12345694	R4 = 0x12345694	
0x00006	LW R1, 0(R0)	00000050	R1 = 0x00000001	R1 = 0x00000001	
0x00007	LW R2, 1(R0)	00010090	R2 = 0x00000001	R2 = 0x00000001	
0x00008	LW R3, 2(R0)	000200D0	R3 = 0x0000000A	R3 = 0x0000000A	
0x00009	SUB R4, R4, R4	00A42100	R4 = 0	R4 = 0	
	Loop1:				
0x0000A	ADD R4, R2, R4	00841100			R4 += R2
0x0000B	SLT R6, R2, R3	00C31180			R6 = (R2 < R3) ? 1 : 0

0x0000C	BEQ R6, R0, done	000030D2			if R6 == 0 → done
0x0000D	ADD R2, R1, R2	00820880			R2 +=R1
0x0000E	BEQ R0, R0, Loop1	FFE00712			Unconditional jump to loop1
	done:				
0x0000F	SW R4, 0(R0)	00040011	Mem [0] = 0x37	Mem [0] = 0x37	
0x00010	MUL R10, R2, R3	01A31280	R10 = 0x64	R10 = 0x64	
0x00011	SRL R14, R10, R4	00245380	R14 = 0	R14 = 0	
0x00012	SRA R15, R10, R4	004453C0	R15 = 0	R15 = 0	
0x00013	RORI R26, R14, 5	00057684	R26 = 0	R26 = 0	
0x00014	JALR R7, R0, func	001A01CF	R7 = 0x15	R7 = 0x15	Jump to func
0x00015	SET R9, 0x4545	4545024D	R9 = 0x00004545	R9 = 0x00004545	
0x00016	SET R10, 0x4545	4545028D	R10 = 0x00004545	R10 = 0x00004545	
0x00017	BGE R10, R9, L1	00095095			Branch if R10 >= R9 (taken)
0x00018	ANDI R23, R1, 0xFFFF	FFFF0DCB			R23 = R1 & 0x0000FFFF (skipped)
	L1:				
0x00019	BEQ R0, R0, L1	00000012			Infinite Loop to terminate
	func:				
0x0001A	OR R5, R2, R3	01431140	R5 = 0xA	R5 = 0xA	
0x0001B	LW R1, 0(R0)	00000050	R1 = 0x37	R1 = 0x37	
0x0001C	LW R2, 5(R1)	00050890	R2 = 0x128945AC	R2 = 0x128945AC	
0x0001D	LW R3, 6(R1)	000608D0	R3 = 0x05007342	R3 = 0x05007342	
0x0001E	AND R4, R2, R3	01631100	R4 = 0x4100	R4 = 0x4100	
0x0001F	SW R4, 0(R0)	00040011	Mem [0] = 0x4100	Mem [0] = 0x4100	
0x00020	JALR R0, R7, 0	0000380F			Return to caller (JR R7)

Project Phase 2

Pipelined Processor Design

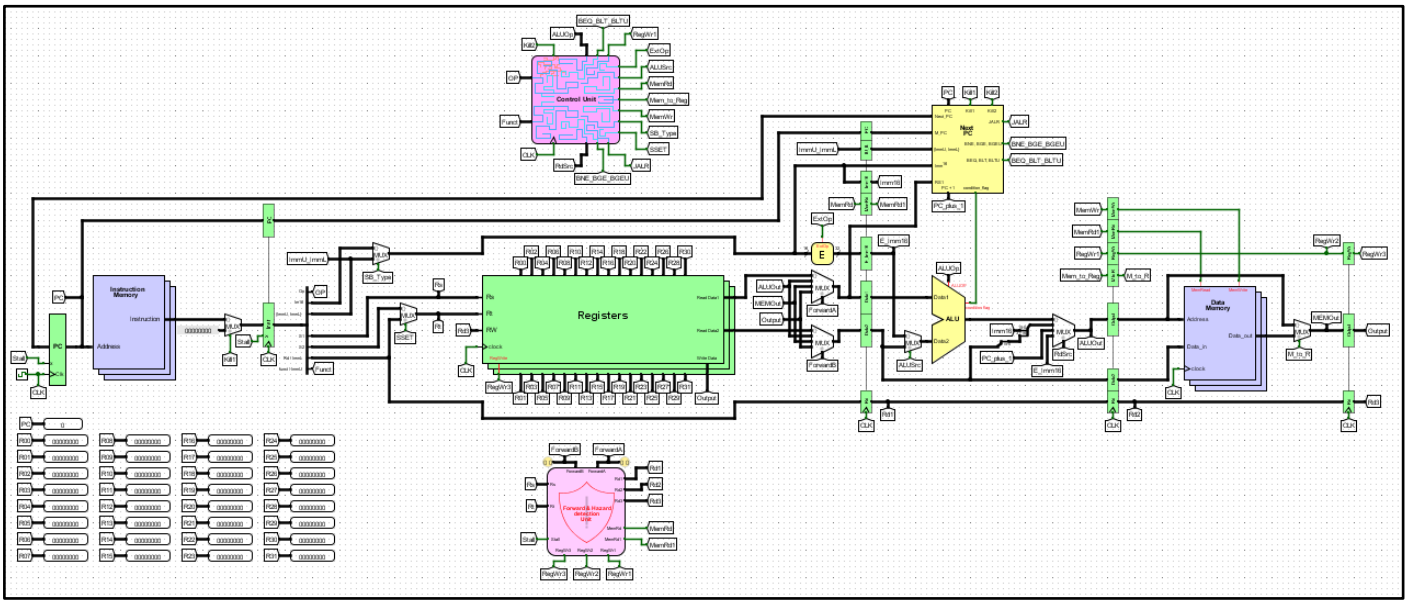
13. Introduction

In the second phase of our project, we extended the initial single-cycle processor to implement a 5-stage pipelined architecture. The goal of pipelining is to improve overall instruction throughput by overlapping the execution of multiple instructions. This design divides instruction execution into five sequential stages: **Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB)**.

Each stage operates concurrently with the others, allowing a new instruction to enter the pipeline at every clock cycle. To support this behavior, we introduced **pipeline registers** between each stage to hold intermediate data and control signals. Additionally, mechanisms for **hazard detection**, **forwarding**, and **branch handling** were implemented to ensure correct execution and minimize performance penalties caused by data and control dependencies.

This section of the project highlights how each pipeline stage was constructed, how data and control flow through the pipeline, and the key components added to support pipelined execution.

14. Data Path



Overview

The pipelined processor datapath divides instruction execution into five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). This design improves performance by allowing multiple instructions to be processed concurrently.

Stage-by-Stage Explanation

1. Instruction Fetch (IF)

- **Process:** Fetches the next instruction from Instruction Memory using the Program Counter (PC) address.
- **Output:** The instruction ("Instr") is sent to the ID stage, and the PC is incremented for the next cycle.

2. Instruction Decode (ID)

- **Process:** Decodes the instruction using the Control Unit to generate signals. Reads operands from the Register File using rs and rt.
- **Output:** Operands and sign-extended immediate values are sent to the EX stage.

3. Execute (EX)

- **Process:** The ALU performs the operation on operands. Computes branch target addresses and handles forwarding for hazard resolution.
- **Output:** ALU result is sent to the MEM stage.

4. Memory Access (MEM)

- **Process:** Uses the ALU result to access Data Memory for load/store operations. Reads data for loads or writes data for stores.
- **Output:** Memory data (for loads) or ALU result (for others) goes to the WB stage.

5. Write Back (WB)

- **Process:** Writes the result (from ALU or Memory) back to the Register File using the rd address.
- **Output:** Completes the instruction execution.

Key Components

- **Registers:** Stores data for processing (green block).
- **Control Unit:** Generates control signals (pink block).
- **Forwarding Units:** Resolve data hazards by providing updated values.
- **Hazard Detection:** Manages stalls to ensure correct pipeline flow.

Advantages

- Overlaps instruction execution, increasing throughput compared to the single-cycle design.

Challenges

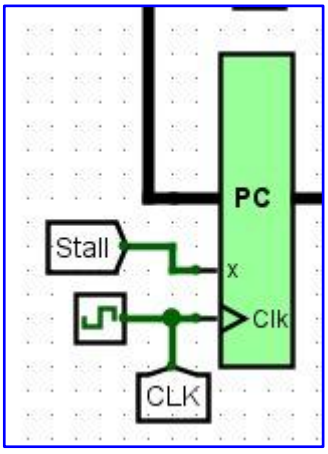
- Requires hazard handling (e.g., data forwarding, stalling) to manage pipeline conflicts.

15. Modifications of Single Cycle CPU Datapath:

1. Stall Control

Modification Description

The original single-cycle processor design had the enable signal for the Program Counter (PC) and Instruction (Inst) registers set to a constant value of 1, ensuring continuous updates with each clock cycle. In the transition to a pipelined architecture, this enable signal has been modified to use the inverted Stall signal (~Stall) as the control input.

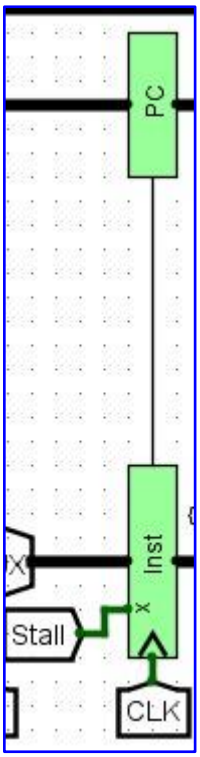


Detailed Explanation

- **Original Behavior:** The enable signal being 1 allowed the PC to increment and the Instruction register to fetch a new instruction on every clock cycle.
- **New Behavior:** The enable input is now controlled by the logical NOT of the Stall signal. The registers are enabled (PC increments, instruction is fetched) only

when Stall is 0. When Stall is 1, indicating a pipeline hazard, the inverted Stall signal becomes 0, disabling the registers and pausing the pipeline.

- **Purpose:** This change supports pipeline synchronization by halting the Instruction Fetch (IF) stage during hazards (e.g., data dependencies or control hazards), preventing incorrect instruction execution until the pipeline is clear.



Impact

- Enhances pipeline control by introducing stall functionality, ensuring data integrity, and proper instruction sequencing in the pipelined processor.

2. Rd Forwarding to WB Stage

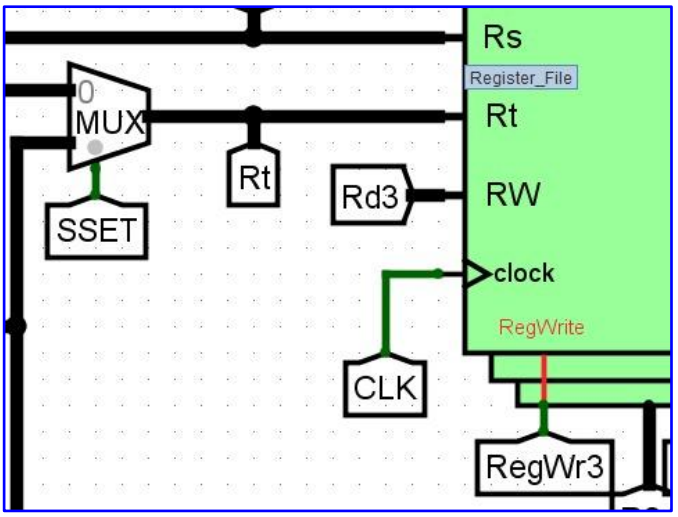
Modification Description

In the original single-cycle processor, the destination register address (Rd) was directly routed to the Register File's write address input (RW) in the Write Back (WB) stage. For the pipelined design, this direct connection has been removed, and Rd is now forwarded through the pipeline stages before reaching the WB stage.

Detailed Explanation

- **Original Behavior:** The Rd signal was directly connected to the RW input of the Register File, allowing immediate writing of the result to the specified register in the same cycle.
- **New Behavior:** The Rd signal (labeled Rd3) is now passed through the pipeline registers, traveling from the Instruction Decode (ID) stage to the Execute (EX) stage, then to the Memory Access (MEM) stage, and finally to

the Write Back (WB) stage. This ensures Rd is available at the WB stage for writing the result to the correct register.



- **Purpose:** This modification aligns with the pipelined architecture, where instructions are processed in stages over multiple cycles. Forwarding Rd ensures the destination register address is preserved and correctly associated with the instruction as it moves through the pipeline.

Impact

- Maintains correct instruction execution by ensuring the destination register address (Rd) reaches the WB stage at the appropriate time.
- Supports hazard handling by making Rd available for dependency checks across pipeline stages, preventing data conflicts.

3. Control Unit in ID-EX Stage

Modification Description

In the single-cycle processor, the Control Unit directly generated control signals for immediate use. In the pipelined design, the Control Unit now includes registers and is integrated into the Instruction Decode to Execute (ID-EX) pipeline stage, with an additional "kill2" signal for branch handling.

Detailed Explanation

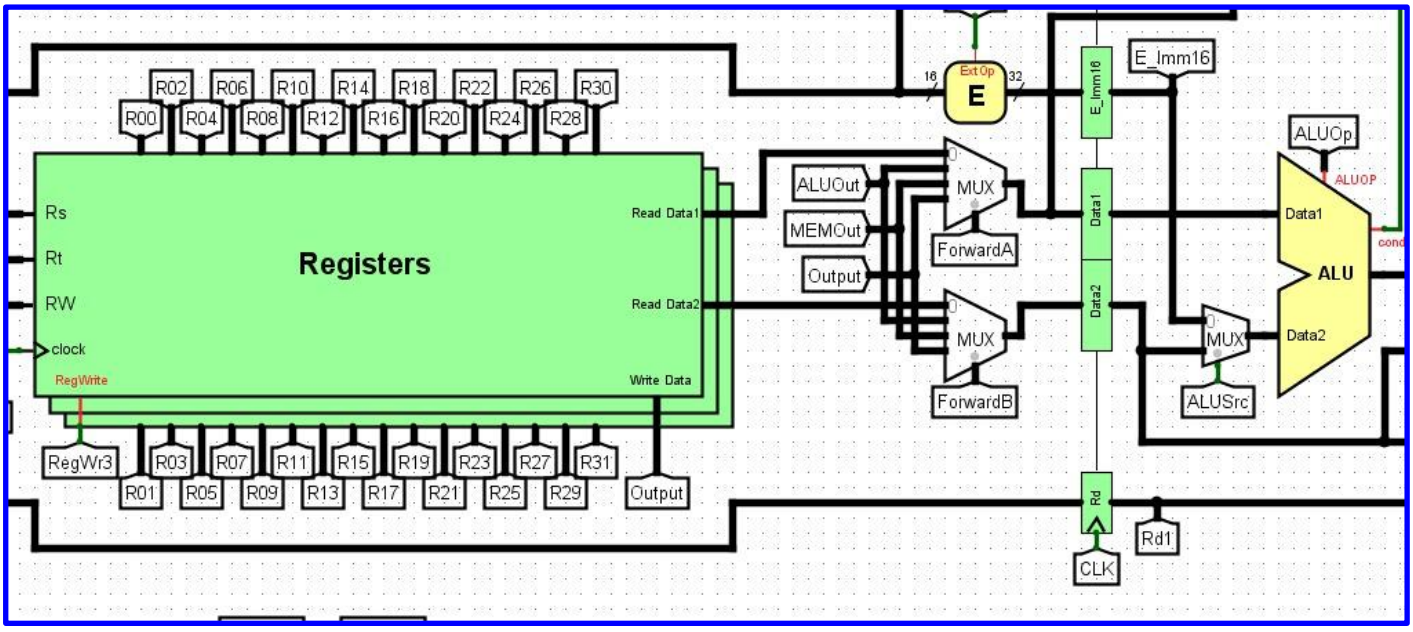
- Original Behavior: The Control Unit generated control signals (e.g., RegDst, ALUOp, RegWr) based on the opcode (OP) and funct fields, and these signals were used within the same cycle.

- New Behavior: The Control Unit is now part of the ID-EX pipeline stage due to space constraints in the design layout. Control signals are generated in the ID stage, stored in the ID-EX registers, and passed to the EX stage for use in subsequent cycles.
- Kill2 Signal: A "kill2" signal has been added to the Control Unit. When a branch instruction is detected, "kill2" sets all control signals to zero, flushing the pipeline for instructions in the ID-EX stage to handle control hazards.
- Purpose: Integrating the Control Unit into the ID-EX stage ensures that control signals are synchronized with the instruction through the pipeline. The "kill2" signal prevents incorrect instruction execution during branches, maintaining proper program flow.

Impact

- Enhances pipeline synchronization by embedding the Control Unit in the ID-EX stage, ensuring control signals are available at the right time.
- Improves control hazard handling through the "kill2" signal, which flushes the pipeline during branches, ensuring correct instruction execution.

4. Register File



Modification Description

In the single-cycle processor, the Register File provided ReadData1 and ReadData2 directly to the ALU. In the pipelined design, two multiplexers (MUXes) have been added after the Register File to implement forwarding.

Detailed Explanation

- **Original Behavior:** The Register File supplied read data (ReadData1 and ReadData2) based on Rs and Rt directly to the ALU inputs.
- **New Behavior:** Two MUXes, ForwardA and ForwardB, are now placed after the Register File outputs. These MUXes select between the Register File data (ReadData1, ReadData2), ALU output (ALUOut), or Memory output (MEMOut) based on forwarding logic. The ALUSrc MUX also selects the sign-extended immediate value (E_Imm16) when needed.
- **Purpose:** The ForwardA and ForwardB MUXes resolve data hazards by forwarding the most recent data from later pipeline stages (e.g., Execute or Memory Access) if the source registers match the destination registers of pending instructions, ensuring the ALU uses updated values.

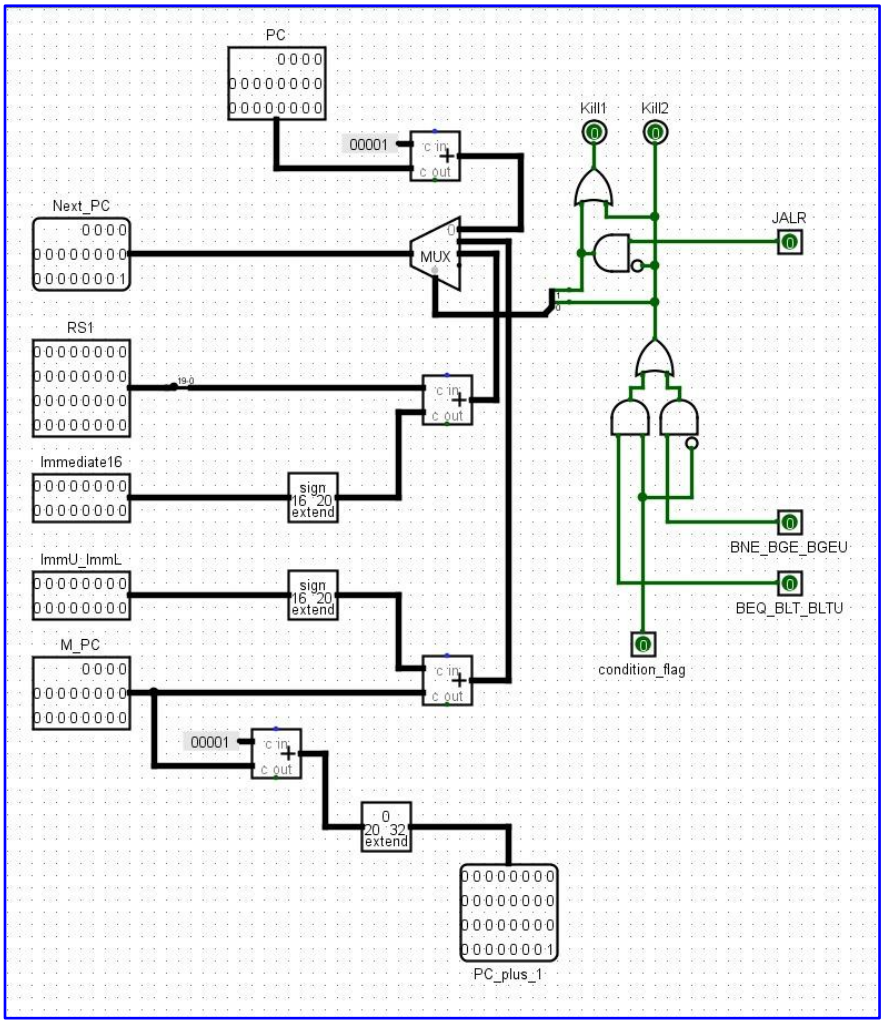
Impact

- Improves pipeline efficiency by reducing stalls caused by data hazards through effective forwarding.
- Maintains data integrity by ensuring the ALU operates with the latest register values, synchronized with the CLK signal.

5. Next PC

Modification Description

In the single-cycle processor, the Next PC was calculated by incrementing the current PC by 1 or using a single immediate value for jumps and branches within the same cycle. In the pipelined design, several changes have been made to address control hazards, including the addition of kill signals, the separation of immediate values, and stage-specific PC inputs.



Detailed Explanation

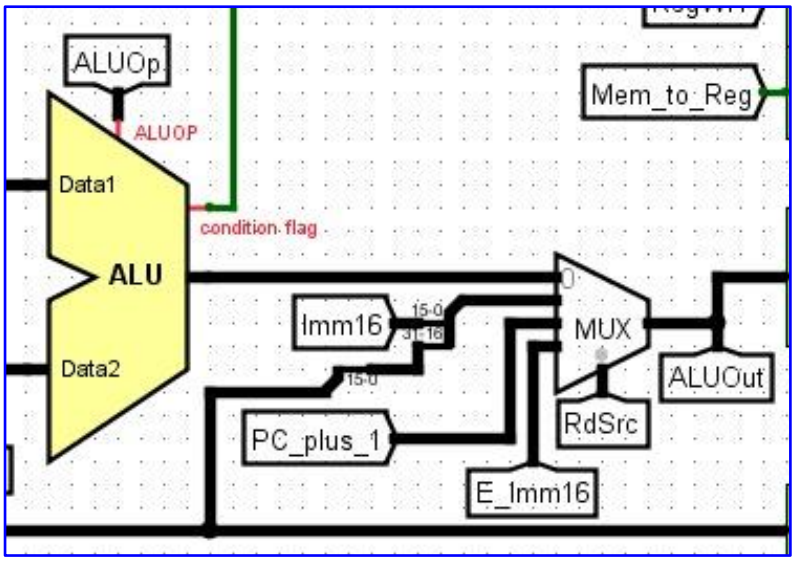
- **Original Behavior:** The Next PC was determined by PC+1 or a single Immediate16 value for both jumps and branches, executed in one cycle.
- **New Behavior:**
  - **Kill Signals (Kill1, Kill2):** Introduced to reset or disable the Next PC calculation when a branch or jump is detected, preventing the fetch of incorrect instructions due to control hazards.
  - **Immediate Value Separation:** The single Immediate16 input is now split into Immediate16 (for branches) and ImmU_ImmL (for jumps), processed independently to handle these operations in different pipeline stages (EX for branches, ID or later for jumps).
  - **PC+1 Input:** The PC+1 input to the MUX now uses the PC from the Instruction Fetch (IF) stage, ensuring consistency with IF's role in

incrementing the PC. Branch calculations, however, use the PC from the Execute (EX) stage.
    - **MUX Logic:** The MUX selects between PC+1 (IF stage), the branch target (Immediate16 + EX stage PC), or the jump target (ImmU_ImmL or JALR), based on the condition flag set by branch evaluations (e.g., BNE, BGE, BEQ) in the EX stage.
- **Purpose:** These changes mitigate control hazards by synchronizing the Next PC with the pipeline stage where control decisions are resolved, separating jump and branch logic, and using kill signals to flush incorrect instructions.

Impact

- Enhances control hazard management by ensuring the Next PC reflects the correct target address based on stage-specific PC values and resolved branch conditions.
- Improves pipeline efficiency by distinguishing jump and branch handling, reducing the risk of fetching incorrect instructions.

6. ALU Output MUX



Modification Description

In the single-cycle processor, the MUX, which selected between the natural ALU result, SET/SSET results, or PC+1, was placed before the ALU. In the pipelined design, this MUX has been moved after the ALU to enable forwarding in the Execute (EX) stage.

Detailed Explanation

- **Original Behavior:** The MUX, positioned before the ALU, chose between the natural ALU result, SET/SSET outputs, or PC+1, and passed the selected value directly to the next stage or Register File in one cycle.
- **New Behavior:** The MUX is now located after the ALU output (ALUOut), selecting between ALUOut, the sign-extended immediate value (E_Imm16), and PC+1. The RdSrc signal controls the selection, while the Mem_to_Reg signal routes the output to the Write Back (WB) stage or memory. This placement allows forwarding of these values from the EX stage to resolve data hazards.
- **Purpose:** Moving the MUX after the ALU enables immediate forwarding of ALUOut, E_Imm16, or PC+1 to earlier stages (e.g., EX) if data dependencies exist. The condition_flag ensures the correct value is selected based on the instruction context, optimizing pipeline flow.

Impact

- Enhances pipeline performance by reducing stalls through efficient forwarding of ALU results or alternative values in the EX stage.
- Ensures data integrity by allowing the latest computed values to be used, synchronized with the Mem_to_Reg and RdSrc controls.

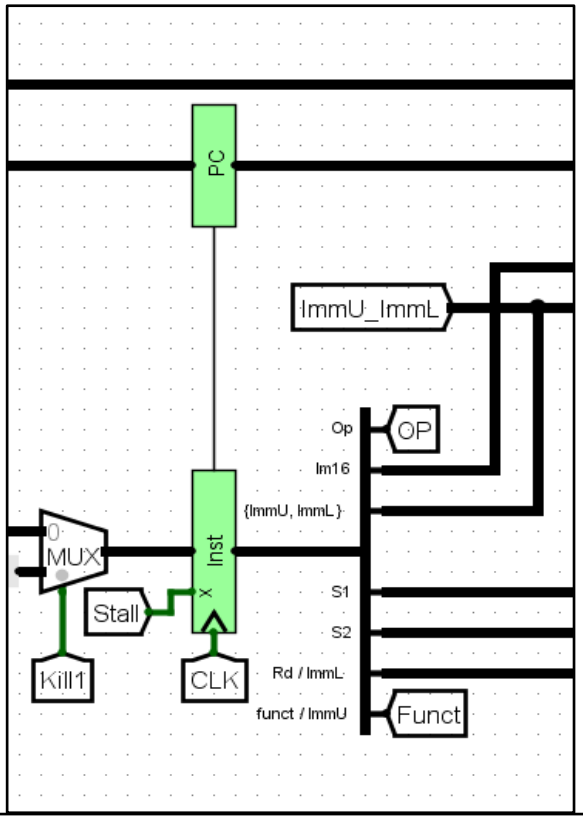
16. Stage Registers

Introduction

The transition from a single-cycle processor to a pipelined architecture requires dividing the Central Processing Unit (CPU) into distinct stages to enable concurrent instruction execution, thereby improving performance. To achieve this division, the various signals within the CPU must be distributed across a series of stage registers. Each time a signal is input into a stage register, it is effectively transferred to the next pipeline stage, ensuring that data and control information progress in a synchronized manner. The process begins with the Instruction Fetch (IF) stage, which outputs the Program Counter (PC) and the fetched Instruction. These outputs are then utilized in the Instruction Decode (ID) stage to derive additional signals. By inputting the Instruction and PC into the IF-ID registers, the data is transferred from the IF stage to the ID stage. This mechanism extends progressively through the pipeline, covering all stages up to the Write Back (WB) stage, facilitating the structured flow of instructions across the pipelined processor.

We will take each register in details in this section.

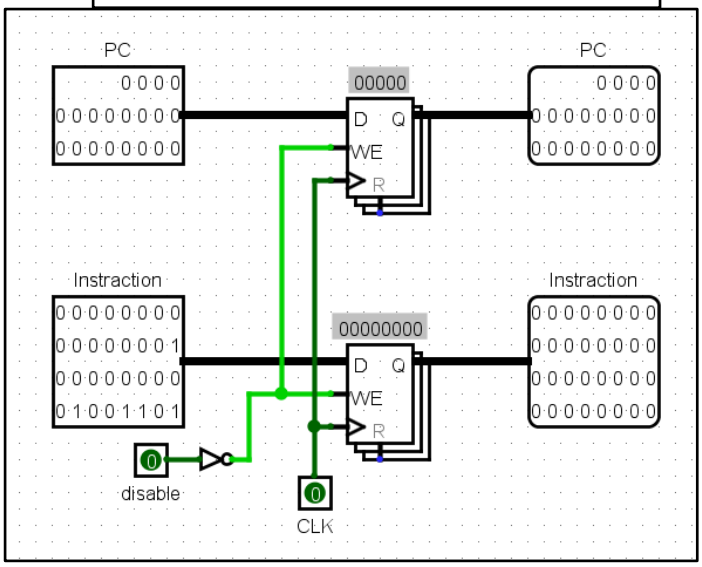
1.IF/ID Register



Overview

The IF/ID register acts as the pipeline register between the Instruction Fetch (IF) and Instruction Decode (ID) stages. It captures and transfers the Program Counter (PC) and the fetched Instruction from the IF stage to the ID stage, enabling concurrent operation of pipeline stages.

Components and Functionality



- **PC Register:**
  - **Purpose:** Stores the current Program Counter value, which is the memory address of the fetched instruction.
  - **Usage:** The PC value is passed to the ID stage, where it is used for calculating branch target addresses or supporting jump instructions.
- **Instruction Register:**
  - **Purpose:** Holds the 32-bit instruction fetched from memory during the IF stage.
  - **Usage:** Transfers the instruction to the ID stage, where it is decoded to extract fields such as the opcode, source registers (rs, rt), destination register (rd), and immediate values.
- **Write Enable (WE) Control:**

- **Mechanism:** Controlled by a "disable" signal through an OR gate (with a constant input of 0 in this case). When disable is 0, WE is enabled, allowing the register to update on each clock cycle.
- **Purpose:** Provides a mechanism to stall the pipeline by preventing updates to the register when the disable signal is asserted (e.g., during a pipeline stall due to control hazards).

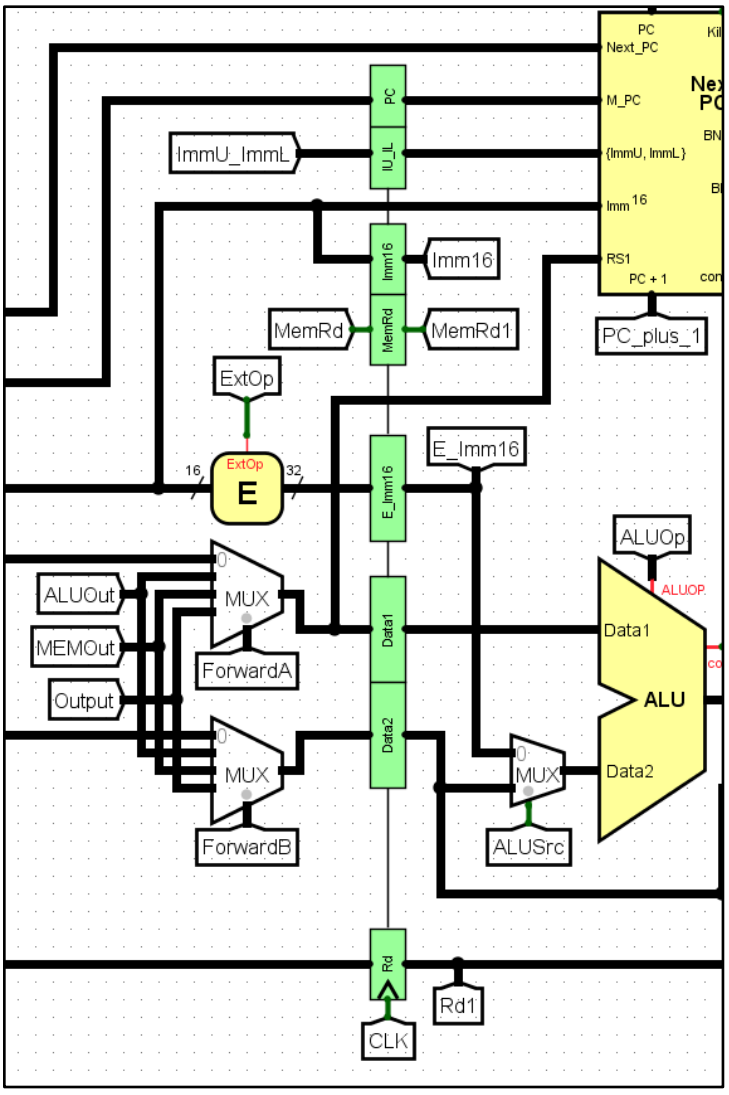
Operation

- **Input Phase:** At the rising edge of the CLK, the IF/ID register captures the PC and Instruction from the IF stage outputs.
- **Output Phase:** These values are held stable and made available to the ID stage for the next clock cycle, allowing the IF stage to fetch the next instruction in parallel.
- **Stall Handling:** The disable signal can deassert WE, halting updates to the register and pausing the pipeline at the IF/ID boundary, which is critical for managing control hazards like branch delays.

Impact

- Facilitates pipelined execution by ensuring the PC and Instruction are reliably transferred from IF to ID, enabling concurrent instruction processing.
- Supports hazard management by providing a stall mechanism, ensuring correct instruction flow during pipeline interruptions.

2. ID/EX Register



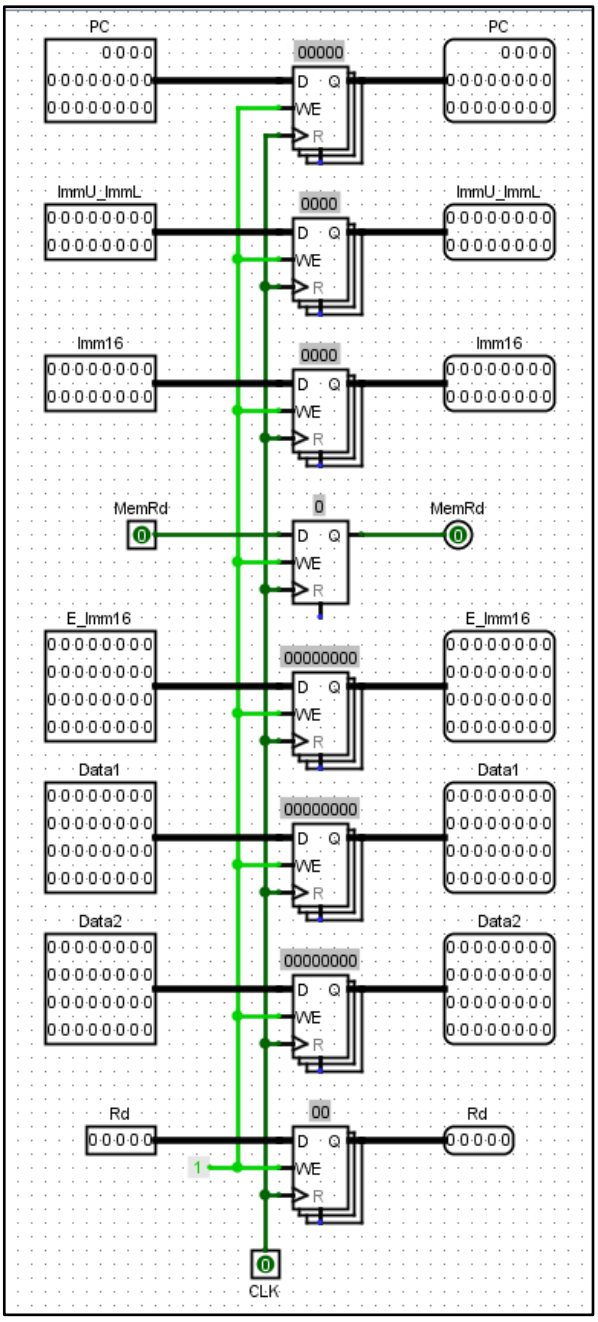
Overview

The ID/EX register bridges the Instruction Decode (ID) and Execute (EX) stages in a pipelined processor, capturing and transferring control signals, data values, and intermediate results from ID to EX. This enables concurrent instruction processing while maintaining synchronization.

Components and Functionality

- **PC Register:**
  - **Purpose:** Stores the Program Counter (PC) for branch or jump target calculations in the EX stage.
  - **Datapath Role:** Transfers the PC from IF/ID to EX for consistent address computation.
  - **Implementation:** A 32-bit D flip-flop, updated with CLK and WE.

- **ImmU_ImmL Register:**
  - **Purpose:** Holds the upper and lower immediate pair for jump instructions.
  - **Datapath Role:** Used in EX for jump target generation.
  - **Implementation:** Two 32-bit D flip-flops, synchronized with CLK and WE.
- **Imm16 Register:**
  - **Purpose:** Stores the 16-bit immediate value for branch offsets or immediate operands.
  - **Datapath Role:** Passed to the ExtOp unit in EX to generate E_Imm16.
  - **Implementation:** A 16-bit D flip-flop, updated with CLK and WE.
- **MemRd Register:**
  - **Purpose:** Holds the Memory Read (MemRd) control signal for EX stage memory operations.
  - **Datapath Role:** Prepares for memory access in the MEM stage.
  - **Implementation:** A single-bit D flip-flop, synchronized with CLK and WE.



- **MemRd1 Register:**
  - **Purpose:** Auxiliary register for MemRd, supporting hazard detection or synchronization.
  - **Datapath Role:** Preserves MemRd for additional checks.
  - **Implementation:** A single-bit D flip-flop, updated with CLK and WE.
- **E_Imm16 Register:**

    - **Purpose:** Stores the sign-extended 32-bit immediate value (E_Imm16).
    - **Datapath Role:** Provides the second ALU operand via ALUSrc MUX.
    - **Implementation:** A 32-bit D flip-flop, synchronized with CLK and WE.
- **Data1 Register:**
    - **Purpose:** Holds the first ALU operand (from rs).
    - **Datapath Role:** Selected by ForwardA MUX for ALU input, handling data hazards.
    - **Implementation:** A 32-bit D flip-flop, updated with CLK and WE.
- **Data2 Register:**
    - **Purpose:** Holds the second ALU operand (from rt).
    - **Datapath Role:** Selected by ForwardB MUX, supports forwarding.
    - **Implementation:** A 32-bit D flip-flop, synchronized with CLK and WE.
- **Rd Register:**
    - **Purpose:** Stores the destination register address (Rd).
    - **Datapath Role:** Used in EX to determine the WB stage write-back register.
    - **Implementation:** A 5-bit D flip-flop (for a 32-register file), updated with CLK and WE.
- **CLK and WE Control:**
    - **Purpose:** CLK synchronizes all flip-flops; WE controls data updates.
    - **Datapath Role:** Ensures timely capture and transfer, with WE enabling stalls.
    - **Implementation:** Common CLK drives all flip-flops; WE is applied to each.

Operation

- **Input Phase:** On CLK rising edge, with WE high, the register captures PC, ImmU_ImmL, Imm16, MemRd, MemRd1, E_Imm16, Data1, Data2, and Rd from the ID stage.
- **Output Phase:** Holds and provides these values to the EX stage for the next cycle, allowing ID to decode the next instruction.

- **Forwarding:** ForwardA and ForwardB MUXes use Data1 and Data2, forwarding ALUOut or MEMOut for hazard resolution.
- **Control Flow:** Next PC logic (with ImmU_ImmL, Imm16, PC+1) and ALUOp/ALUSrc support branch/jump execution.

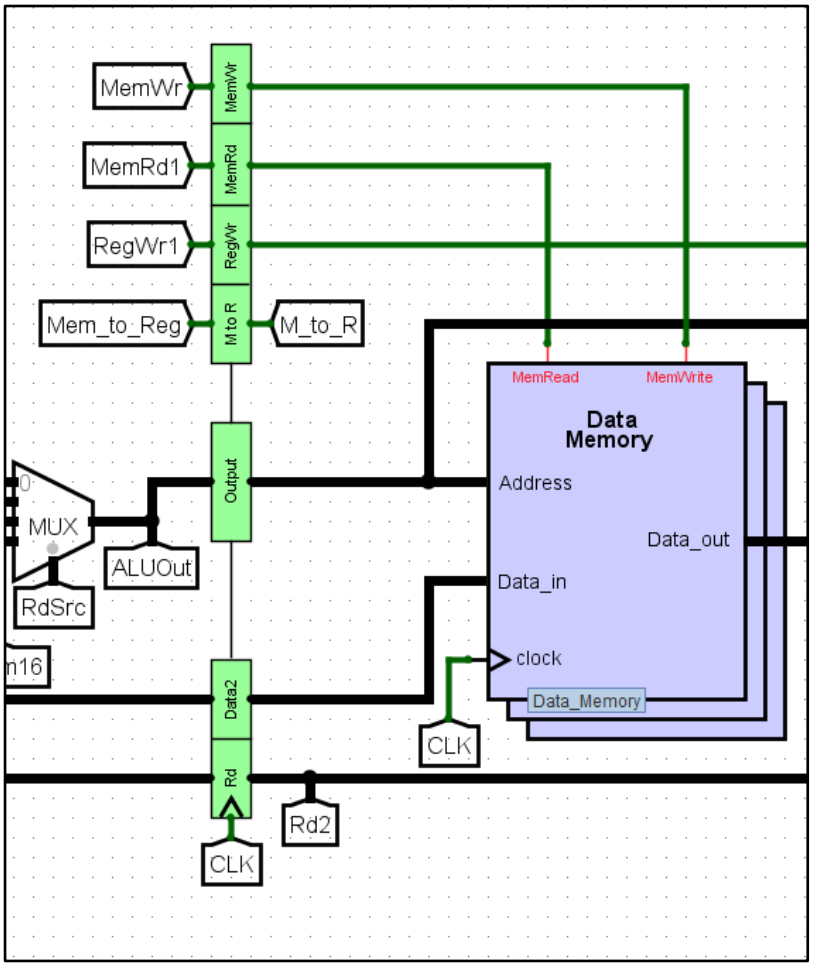
Impact

- **Synchronization:** Ensures all ID stage outputs are transferred to EX, enabling pipelined execution.
- **Hazard Management:** Supports data hazard resolution via forwarding and control hazard handling via Next PC logic.
- **Performance:** Increases throughput by facilitating concurrent stage operation compared to a single-cycle design.

3. EX/MEM Register

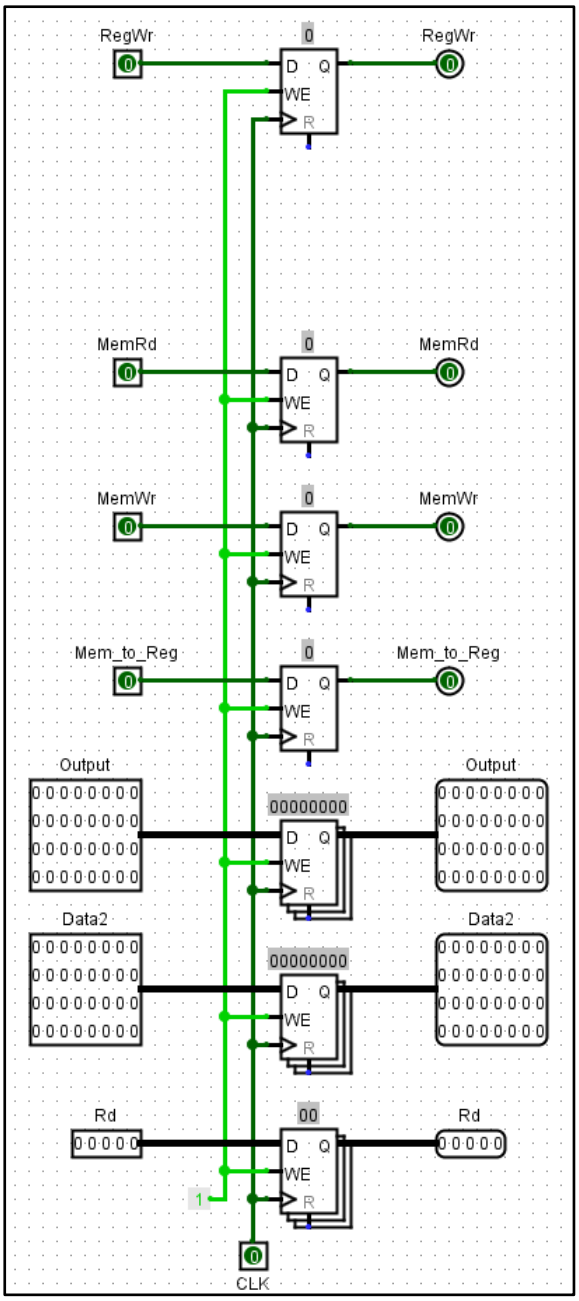
Overview

The EX/MEM register bridges the Execute (EX) and Memory Access (MEM) stages in a pipelined processor, capturing and transferring control signals, ALU results, and data from EX to MEM. This enables concurrent execution and supports memory operations.



Components and Functionality

- **MemWr Register:**
  - **Purpose:** Stores the Memory Write (MemWr) control signal for MEM stage write operations.
  - **Datapath Role:** Enables writing Data2 to Data Memory.
  - **Implementation:** A single-bit D flip-flop, updated with CLK and WE.
- **MemRd Register:**
  - **Purpose:** Holds the Memory Read (MemRd) control signal for MEM stage read operations.
  - **Datapath Role:** Prepares for reading Data_out from memory.
  - **Implementation:** A single-bit D flip-flop, synchronized with CLK and WE.
- **MemRd1 Register:**
  - **Purpose:** Auxiliary MemRd register for synchronization or hazard detection.
  - **Datapath Role:** Ensures consistent MemRd availability.
  - **Implementation:** A single-bit D flip-flop, updated with CLK and WE.



- **RegWr1 Register:**
  - **Purpose:** Stores the Register Write (RegWr1) control signal for WB stage write-back.
  - **Datapath Role:** Controls Register File updates in WB.
  - **Implementation:** A single-bit D flip-flop, synchronized with CLK and WE.
- **Mem_to_Reg Register:**
  - **Purpose:** Holds the Mem_to_Reg control signal to select data source for WB.

    - **Datapath Role:** Controls MUX to choose between ALUOut and Data_out.
    - **Implementation:** A single-bit D flip-flop, updated with CLK and WE.
- **Output Register:**
    - **Purpose:** Stores the ALU output (ALUOut) from the EX stage.
    - **Datapath Role:** Serves as the Data Memory address or WB data.
    - **Implementation:** A 32-bit D flip-flop, synchronized with CLK and WE.
- **Data2 Register:**
    - **Purpose:** Holds the second operand (Data2) for memory write operations.
    - **Datapath Role:** Provides Data_in for store instructions.
    - **Implementation:** A 32-bit D flip-flop, updated with CLK and WE.
- **Rd Register:**
    - **Purpose:** Stores the destination register address (Rd) for WB stage.
    - **Datapath Role:** Specifies the register to be updated in WB.
    - **Implementation:** A 5-bit D flip-flop (for a 32-register file), synchronized with CLK and WE.
- **CLK and WE Control:**
    - **Purpose:** CLK synchronizes all flip-flops; WE controls data updates.
    - **Datapath Role:** Ensures timely transfer, with WE enabling stalls.
    - **Implementation:** Common CLK drives all flip-flops; WE is applied to each.

Operation

- **Input Phase:** On CLK rising edge, with WE high, captures MemWr, MemRd, MemRd1, RegWr1, Mem_to_Reg, Output (ALUOut), Data2, and Rd from EX.
- **Output Phase:** Holds and provides these values to MEM, allowing EX to process the next instruction.
- **Memory Access:** ALUOut is the Address; Data2 is Data_in for stores (MemWr); Data_out is read for loads (MemRd), selected by Mem_to_Reg MUX.
- **Write-Back Prep:** Rd and RegWr1 are passed to WB for register update.

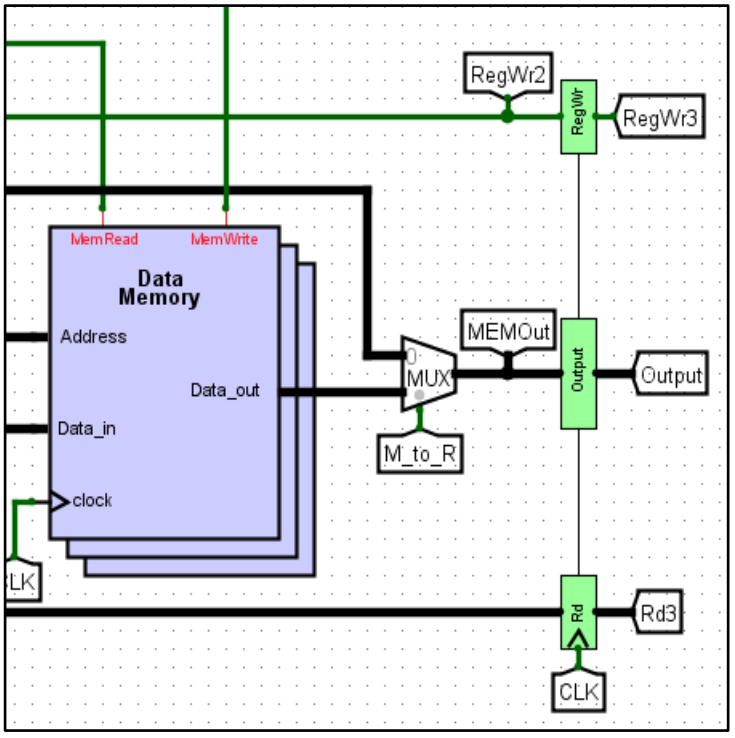
Impact

- **Synchronization:** Transfers EX outputs to MEM, enabling pipelined execution.
- **Memory Support:** Facilitates read/write operations with appropriate control and data signals.
- **Hazard Management:** Ensures correct Rd and control signals for WB, minimizing hazards.
- **Performance:** Increases throughput by allowing concurrent EX and MEM operations.

4. MEM/WB Register

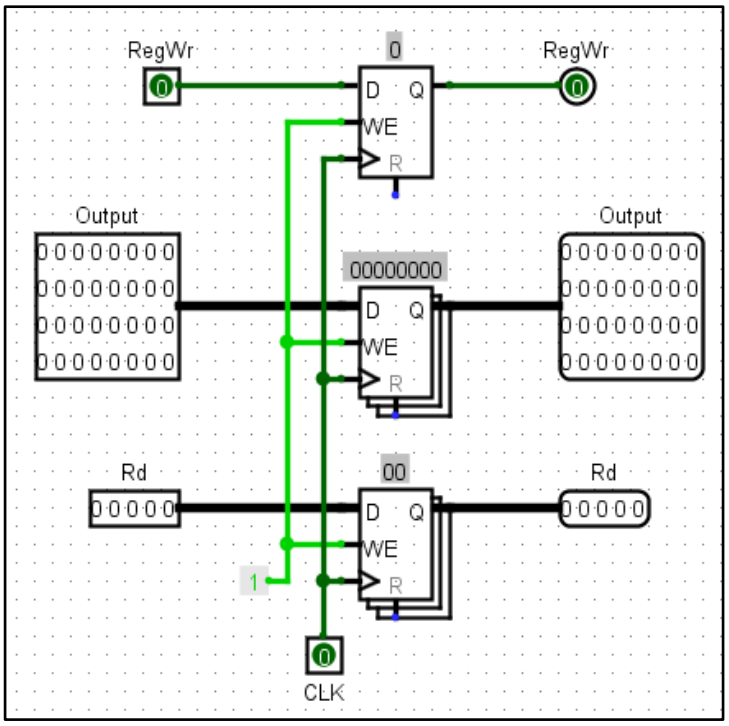
Overview

The MEM/WB register bridges the Memory Access (MEM) and Write Back (WB) stages in a pipelined processor, capturing and transferring control signals and data from MEM to WB. This enables the final write-back of results to the Register File.



Components and Functionality

- **RegWr Register:**
  - **Purpose:** Stores the Register Write (RegWr) control signal for WB stage register updates.
  - **Datapath Role:** Enables writing the Output to the Register File at the Rd address.
  - **Implementation:** A single-bit D flip-flop, updated with CLK and WE.



- **Output Register:**
  - **Purpose:** Holds the final data to be written back (ALUOut or Data_out).
  - **Datapath Role:** Selected by the Mem_to_Reg MUX, passed to WB for Register File update.
  - **Implementation:** A 32-bit D flip-flop, synchronized with CLK and WE.
- **Rd Register:**
  - **Purpose:** Stores the destination register address (Rd) for WB stage.
  - **Datapath Role:** Specifies the target register for the write-back operation.
  - **Implementation:** A 5-bit D flip-flop (for a 32-register file), updated with CLK and WE.
- **CLK and WE Control:**
  - **Purpose:** CLK synchronizes all flip-flops; WE controls data updates.
  - **Datapath Role:** Ensures timely transfer from MEM to WB, with WE enabling stalls.
  - **Implementation:** Common CLK drives all flip-flops; WE is applied to each.

Operation

- **Input Phase:** On CLK rising edge, with WE high, captures RegWr, Output (from Mem_to_Reg MUX), and Rd from MEM.
- **Output Phase:** Holds and provides these values to WB, allowing MEM to process the next instruction.
- **Write-Back Process:** Output is written to the Register File at Rd when RegWr is high, with Mem_to_Reg selecting between ALUOut and Data_out.
- **Memory Interaction:** Data Memory provides Data_out (for loads) or accepts Data_in (for stores) based on MemRd and MemWr from the EX/MEM stage.

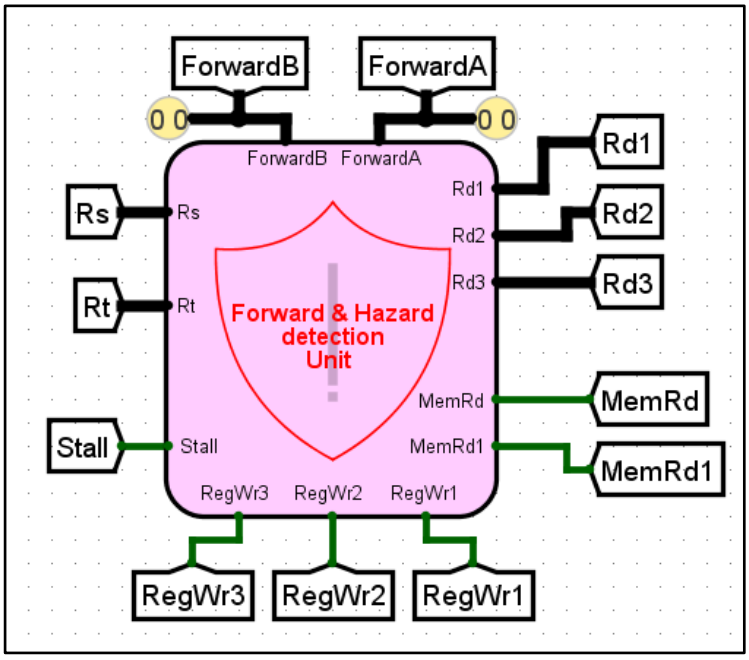
Impact

- **Synchronization:** Transfers MEM outputs to WB, completing the pipelined execution cycle.
- **Write-Back Support:** Ensures accurate register updates with the correct data and address.
- **Hazard Management:** Minimizes hazards by consistent Rd and RegWr transfer.
- **Performance:** Increases throughput by enabling concurrent MEM and WB operations.

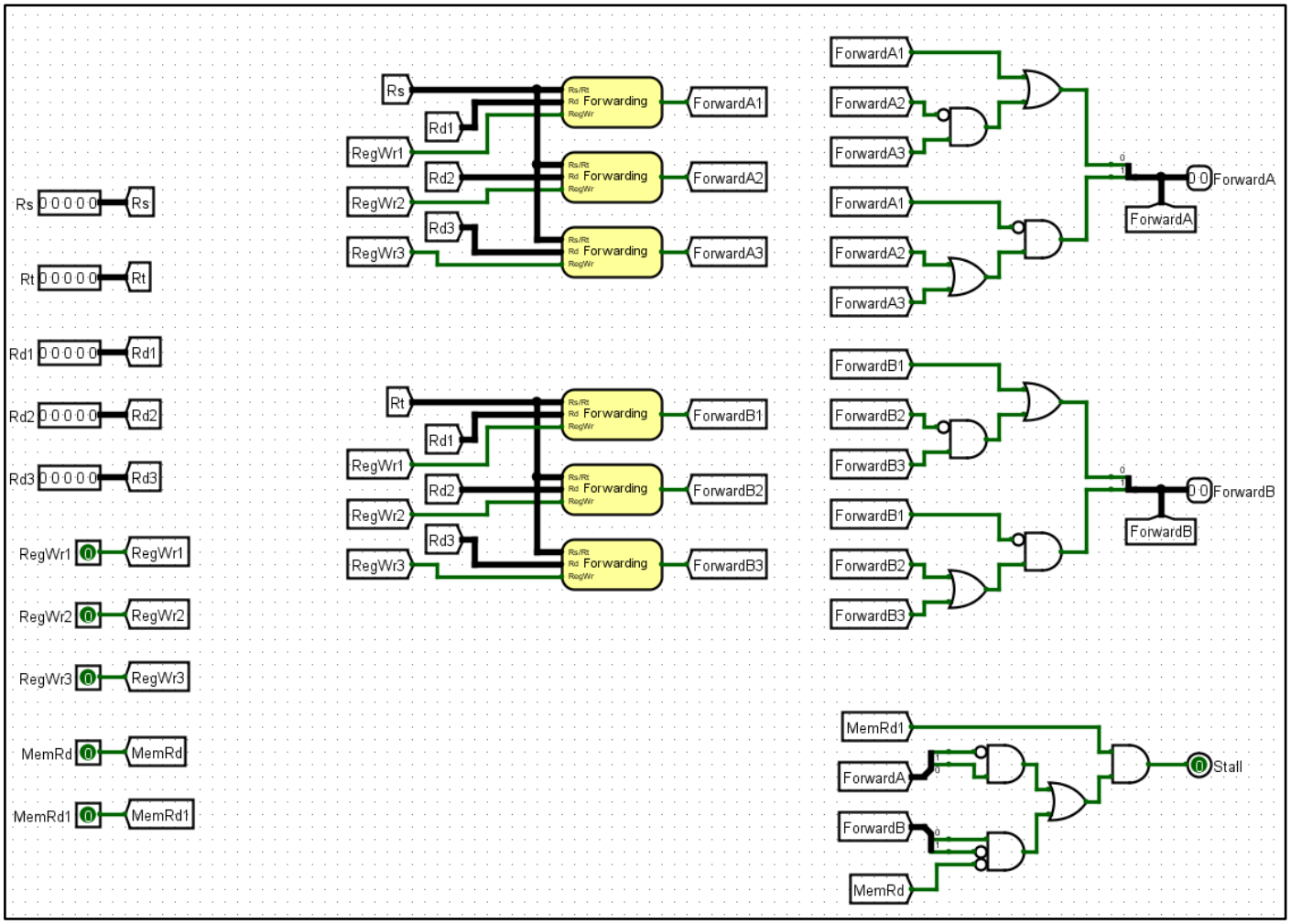
17. Forward & Hazard Detection Unit

Overview

The Forwarding and Hazard Detection Unit addresses Read-After-Write (RAW) hazards in a pipelined processor, enhancing efficiency with forwarding mechanisms and stall detection. It generates ForwardA and ForwardB signals for operand



selection and a Stall signal for load-use hazards.



Components and Functionality

Forwarding Block

- **Input Registers (Rs, Rt, Rd1, Rd2, Rd3):**
  - **Purpose:** Rs and Rt are ID stage source registers; Rd1-Rd4 are destination registers from EX, MEM, and WB stages.
  - **Datapath Role:** Compared to detect data hazards.
  - **Implementation:** 5-bit registers for a 32-register file.
- **RegWr Signals (RegWr1, RegWr2, RegWr3):**
  - **Purpose:** Indicate write-back in EX, MEM, and WB stages.
  - **Datapath Role:** Triggers forwarding with register matches.
  - **Implementation:** Single-bit inputs from pipeline registers.
- **Comparison Logic (Rs/Rt != 0 and Rs/Rt = Rd and RegWr):**

    - **Purpose:** Identifies matches between source and destination registers with active write-back.
    - **Datapath Role:** Initiates forwarding when hazards are detected.
    - **Implementation:** AND gates for (Rs/Rt != 0) AND (Rs/Rt = Rd) AND RegWr.
- **ForwardA and ForwardB Logic Blocks:**
    - **Purpose:** Generate 2-bit ForwardA (for Rs) and ForwardB (for Rt) signals.
    - **Datapath Role:** Control EX stage MUXes to select operands.
    - **Implementation:** Three AND gates per block:
        - ForwardA1/ForwardB1: (Rs/Rt != 0) AND (Rs/Rt = Rd2) AND (EX.RegWr) → 1 (EX data).
        - ForwardA2/ForwardB2: (Rs/Rt != 0) AND (Rs/Rt = Rd3) AND (MEM.RegWr) → 2 (MEM data).
        - ForwardA3/ForwardB3: (Rs/Rt != 0) AND (Rs/Rt = Rd4) AND (WB.RegWr) → 3 (WB data).
        - Else: 0 (Register File data).
    - Optimized with K-maps.
- **Output (ForwardA, ForwardB):**
    - **Purpose:** 2-bit signals (00: Register File, 01: EX, 10: MEM, 11: WB).
    - **Datapath Role:** Drive ForwardA and ForwardB MUXes.
    - **Implementation:** 2-bit outputs from OR gates.

Hazard Detection and Stall Logic

- **MemRd Inputs (MemRd_ID, MemRd_Ex):**
    - **Purpose:** MemRd_ID and MemRd_Ex detect memory reads in ID and EX stages (e.g., LW).
    - **Datapath Role:** Identify load-use hazards.
    - **Implementation:** Single-bit inputs from ID/EX and EX/MEM.
- **ForwardA and ForwardB Inputs:**
    - **Purpose:** Check dependencies on EX stage data (ForwardA = 01 or ForwardB = 01).
    - **Datapath Role:** Indicate RAW hazards with LW instructions.

    - **Implementation:** 2-bit inputs from Forwarding Block.
- **Stall Logic:**
    - **Purpose:** Generates Stall signal for pipeline pause.
    - **Datapath Role:** Prevents ID stage progression during EX load hazards.
    - **Implementation:** AND gate with condition:
        - If ((MemRd_Ex == 1) AND (ForwardA == 01 OR ForwardB == 01) AND (MemRd_ID == 0)) → Stall = 1.
        - Else → Stall = 0.
    - Addresses LW hazards where immediate matches a prior Rd.

Operation

- **Forwarding Process:** Compares Rs/Rt with Rd1-Rd4; sets ForwardA/ForwardB (1, 2, 3) for EX, MEM, WB data, or 0 for Register File data.
- **Hazard Detection:** Monitors MemRd_Ex and ForwardA/ForwardB; asserts Stall for LW in EX with ID dependency.
- **Integration:** ForwardA/ForwardB control EX MUXes; Stall pauses IF/ID and ID/EX registers.

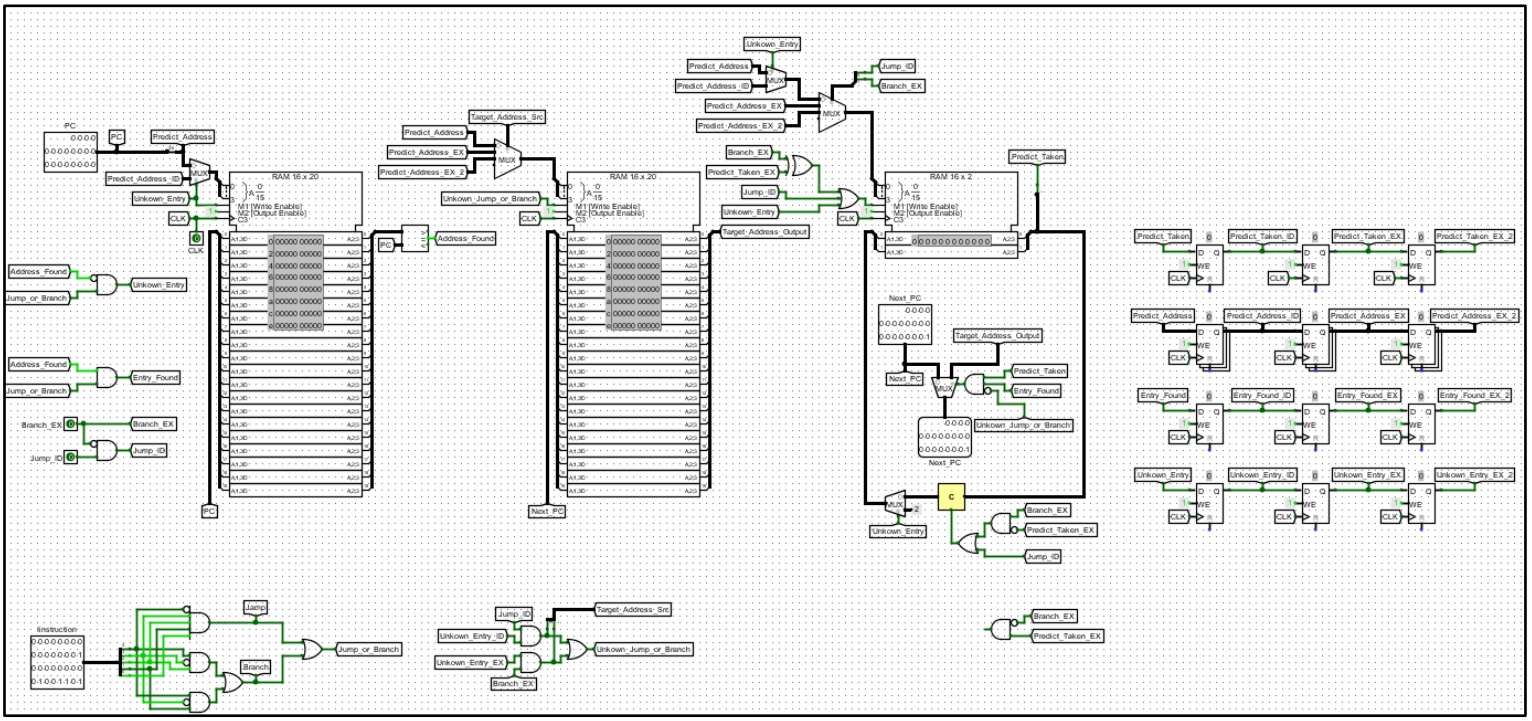
Impact

- **Hazard Resolution:** Eliminates most RAW hazards via forwarding.
- **Performance Optimization:** Reduces stalls with efficient data forwarding.
- **Load Hazard Management:** Handles LW-induced RAW hazards with Stall.
- **Design Simplicity:** Three-block structure and K-map optimization streamline logic.

18. 2-Bit Branch Predictor

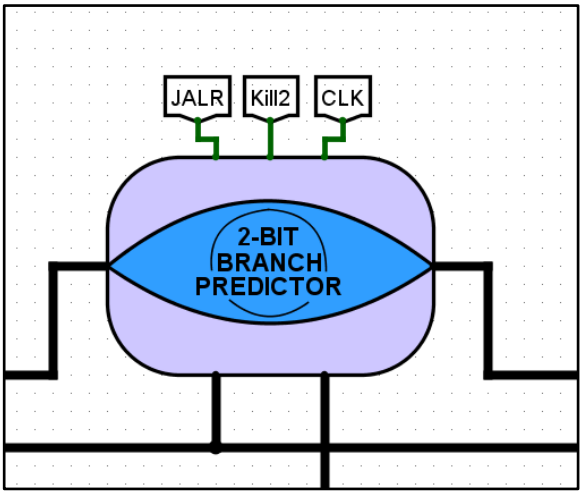
To implement a 2-bit dynamic branch predictor, we used a table-based prediction mechanism supported by three small RAM modules. Each RAM stores a specific piece of prediction-related information:

1. Current PC Table (RAM 1): Holds the recent program counters (PCs) for branch or jump instructions.
2. Target Address Table (RAM 2): Stores the predicted target address corresponding to each PC.
3. Prediction Bit Table (RAM 3): Maintains a 2-bit saturating counter for each entry to track the history of taken/not-taken outcomes.



Addressing and Indexing

All three RAMs are indexed using the lowest 4 bits of the current PC (i.e., PC[3:0]), effectively allowing 16 prediction entries. This acts as a simple hash to quickly access prediction entries based on the instruction address.



Inputs to the Predictor

Signal	Width	Description
PC_IF	32 bits	Program counter in IF stage
Instruction_IF	32 bits	Instruction word to detect if it's a branch or jump
Target_Address_IF	32 bits	Target address of the branch calculated in IF
Branch_Taken_Actual_EX	1 bit	Actual branch decision in EX stage
PC_EX	32 bits	PC value of the instruction in EX stage
Update_Enable	1 bit	Enables update of the predictor from EX stage

Outputs from the Predictor

Signal	Width	Description
Predict_Taken_Out	1 bit	1 if predictor suggests branch is taken
Predicted_Target_Out	32 bits	Target address to jump to if branch is predicted taken
Entry_Found_Out	1 bit	1 if matching PC entry is found

Signal	Width	Description
Mispredict_Flush	1 bit	1 if prediction was incorrect and flush is needed

---

Prediction Phase (IF Stage)

- If the instruction is a branch or jump, the predictor uses the PC[3:0] to index the Current PC RAM.
- It compares the current PC with the stored PC at that index.
- If they match, and the instruction is a branch/jump, an Entry_Found signal is asserted.
- If the 2-bit prediction value is 10 or 11, the predictor sets Predict_Taken to 1 and outputs the target address from the Target RAM to be used as Next_PC.

Update Phase (EX Stage)

- At EX stage, if update is enabled:
    - If the entry was not previously present, write:
        - PC to Current PC RAM
        - Target address to Target Address RAM
        - 01 to Predict-Bit RAM
    - If entry exists:
        - Increment the counter (max 11) if actual branch taken
        - Decrement the counter (min 00) if not taken

Predictor Behavior Summary

Condition	Action Taken
Branch/Jump instruction, no entry	Create entry with PC, Target, and 01 bits

Condition	Action Taken
Entry found & prediction is taken	Use predicted target as Next_PC
Prediction mismatches actual outcome	Trigger Mispredict_Flush and update counter

This design helps minimize pipeline flushes by using historical behavior of branches, reducing delay and improving instruction flow.

19. Teamwork

The project was completed by the following group members:

- **Mohamed Wageh Mahmoud** (CSE)
- **Youssef Ibrahim Mohamed** (CSE)
- **Mohamed Ahmed Kassem** (ECE)

All group members actively participated in the design, implementation, simulation, and testing of the CPU.

Tasks were divided among members to ensure balanced contribution across all project phases.

Members coordinated through meetings and online collaboration to ensure project milestones were achieved on time.

The work was divided logically among all group members to ensure the complete design and simulation of the processor, as shown in the following table:

Task	Contributor(s)
CPU Datapath Design	Youssef Ibrahim, Mohamed Wageh
Register File Design	Youssef Ibrahim Mohamed
ALU Design and Testing	Mohamed Wageh Mahmoud
Control Unit Design	Mohamed Kassem, Mohamed Wageh
Data and Instruction Memory Design	Mohamed Ahmed Kassem
Next PC Logic	Mohamed Wageh, Youssef Ibrahim
Instruction Splitter Design	Youssef Ibrahim Mohamed
Writing an Assembler Script	Mohamed Wageh Mahmoud
Testing and Debugging	All Members
Documentation and Report	All Members
Stage Registers	Youssef Ibrahim, Mohamed Kassem
Modification on Data Path	All Members
Modification on Next PC	All Members
Forward Unit	Mohamed Wageh
Hazard Detection Unit	Mohamed Wageh