Fayoum University

Faculty of Engineering

Electronics & Electrical
Communications Department

Project of:

32-Bit Single Cycle MIPS Processor

For

Prof. Dr. Gihan Naguib

Associate Professor, Department of Electrical Engineering.

Eng. Jihad Awad

Teaching Assistant, Department of Electrical Engineering.

Team Members :-

- **Abdelrahman Abdelnasser Abdelrahman**
- **Yusuf Mohamed Saleh Boriek**
- **Raghad Soliman Mohamed**

Students at ***Department of Electronics and Electrical Communications.***

Table of Content

Phase 1: Single-Cycle Processor

Introduction

- ## Overview

In the field of computer architecture, the design and implementation of a MIPS (Microprocessor without Interlocked Pipeline Stages) processor serves as a foundational milestone for understanding modern processor behavior. This project focuses on the development of a Single-Cycle MIPS Processor, designed and simulated using Logisim.

The primary objective of the project was to construct a fully functional 32-bit single-cycle RISC processor capable of executing a predefined instruction set in a single clock cycle. The processor includes a complete datapath and control unit implementation that supports arithmetic, logical, memory access, and control flow instructions.

This documentation outlines the systematic process followed throughout the project—from architectural planning and component design to testing and verification. Key modules such as the Register File, Arithmetic Logic Unit (ALU), Instruction and Data Memory, Program Counter (PC), and Control Units are discussed in detail. Moreover, simulation results and test cases are presented to demonstrate the correctness and efficiency of the implemented processor.

The project not only enhanced our understanding of CPU internals and digital design principles but also fostered collaboration, problem-solving, and critical

thinking—skills essential for future endeavors in computer engineering and embedded systems.

- # Instruction Set Architecture

The Instruction Set Architecture (ISA) defines the set of operations supported by the processor and serves as the interface between software and hardware. In this project, we implemented a custom 32-bit RISC-based ISA designed to support a comprehensive set of operations, including arithmetic, logical, memory access, and control flow instructions.

The ISA includes a total of 31 general-purpose 32-bit registers (R1 to R31), while R0 is hardwired to zero and cannot be modified. The processor also supports a 20-bit Program Counter (PC), enabling it to address up to 2202^{20}220 instructions.

Each instruction is 32 bits wide and word-aligned in memory. Immediate values are either sign-extended or zero-extended based on the instruction type. This design ensures simplicity, consistency, and efficient execution of instructions within a single clock cycle.

The ISA is structured around three main instruction formats:

- **R-Type**: Used for register-to-register operations such as ADD, SUB, AND, and SLT. These instructions utilize two source registers and one destination register.
- **I-Type**: Designed for operations involving immediate values, memory access (e.g., LW, SW), and branch instructions. This format includes a source register, a destination register, and a 16-bit immediate value.

- **SB-Type**: Used specifically for branch instructions like BEQ, BNE, BLT, and BGE, which rely on two registers and a split immediate offset for calculating the branch target.

R-type format

6-bit opcode (Op), 5-bit destination register number d, and two 5-bit source registers numbers S1 & S2 and 11-bit function field F.

F[11]	**S2[5]**	**S1[5]**	**d[5]**	**Op[6]**

I-type format

6-bit opcode (Op), 5-bit destination register number d, and 5-bit source registers number S1 and16-bit immediate (Imm16)

Imm[16]	**S1[5]**	**d[5]**	**Op[6]**

SB-type format

6-bit opcode (Op), 5-bit register numbers (S1, and S2) and 16-bit immediate split into ({ImmU (11-bit) and ImmL(5-bit)})

ImmU[11]	**S2[5]**	**S1[5]**	**ImmL**[16]	**Op[6]**

Register Use

The processor features 32 general-purpose 32-bit registers, labeled from R0 to R31, providing fast-access storage for instruction execution. These registers serve as operands and result holders for various instructions executed by the processor.

General-Purpose Registers

- R1 to R31: Usable for all instruction operations including arithmetic, logic, memory access, and control flow.
- R0: Hardwired to zero. It always reads as 0, and any write operation to this register is ignored. This behavior simplifies instruction design and enables common operations like MOV or CLEAR.

Special Register Use Cases

- R30 (Stack Pointer - optional): In programs involving procedures or nested function calls, register R30 can be used as a stack pointer, pointing to the top of the stack in memory.
- R31 (Return Address - optional): For control flow instructions like JALR (Jump and Link Register), R31 can be used to temporarily store the return address (i.e., PC + 1).

Usage Conventions

Although all registers (except R0) are general-purpose, specific conventions can be followed to organize code and debugging:

- Temporary registers (e.g., R1–R10)
- Saved registers (e.g., R11–R20)
- Argument passing (e.g., R21–R24)
- Return values (e.g., R25–R26)

These conventions are not enforced by hardware but are helpful for code structure, clarity, and maintainability in complex programs.

Instruction Description

The instruction set for the Single-Cycle MIPS Processor is designed to cover a wide range of operations using a simplified and uniform encoding. Each instruction is 32 bits wide and categorized under three main formats: R-type, I-type, and SB-type. Below is an overview of the instructions and their functionality:

R-Type Instructions

R-Type instructions are used for arithmetic and logical operations that involve **two source registers** and produce a result in a **destination register**. These instructions do not involve immediate values or memory addresses.

Format:

- **Opcode (6 bits)**: Specifies the instruction category (typically 000000 for R-type).
- **RS1 (5 bits)**: First source register.
- **RS2 (5 bits)**: Second source register.
- **RD (5 bits)**: Destination register.
- **Unused (function) bits (11 bits)**: Determine the specific operation (e.g., ADD, SUB, AND).

Example Instructions:

- ADD R1, R2, R3 → Adds contents of R2 and R3, stores the result in R1.
- SLT R5, R6, R7 → Sets R5 to 1 if R6 is less than R7, else sets R5 to 0.

I-Type Instructions (Immediate-Type)

I-Type instructions are used for operations involving a **constant (immediate)** value. These include arithmetic with constants, logical operations, and memory access (e.g., LW, SW).

Format:

- **Opcode (6 bits)**: Specifies the operation (unique for each instruction).
- **RS1 (5 bits)**: Source register.
- **RD (5 bits)**: Destination register.
- **Imm16 (16 bits)**: A 16-bit immediate value (can be sign-extended or zero-extended based on the instruction).

Example Instructions:

- ADDI R1, R2, 5 → Adds 5 to the contents of R2 and stores the result in R1.
- LW R3, 8(R4) → Loads a word from memory address R4 + 8 into R3.

- ANDI R6, R6, 0xFF → Performs bitwise AND between R6 and the constant 0xFF.

Immediate Handling:

- Sign-extended for arithmetic/branching.
- Zero-extended for logical operations.

SB-Type Instructions (Split-Immediate Branch-Type)

SB-Type instructions are used for **conditional branching** and **store word (SW)** instructions. They differ in how the immediate value is encoded—split into two fields (ImmU and ImmL) to form a 16-bit signed offset.

Format:

- **Opcode (6 bits)**: Specifies the operation (e.g., BEQ, BNE, SW, etc.).
- **RS1 (5 bits)**: First source register (used in condition checking or address base).
- **RS2 (5 bits)**: Second source register (used in condition checking or data source).
- **ImmU (11 bits)**: Upper part of the immediate.
- **ImmL (5 bits)**: Lower part of the immediate.

The two immediate parts (ImmU, ImmL) are concatenated and sign-extended to calculate the **offset for branch instructions** or the **effective memory address for SW**.

Example Instructions:

- BEQ R1, R2, Label → Branches to Label if the contents of R1 and R2 are equal.
- SW R3, offset(R4) → Stores the value in R3 to memory at the address calculated by R4 + offset.

Processor Architecture

The architecture of the single-cycle MIPS processor is designed to execute every instruction within a single clock cycle. This design emphasizes simplicity and clarity, making it highly suitable for educational and foundational processor development. The processor is composed of several interconnected components, each responsible for a distinct part of instruction execution.

Core Components

The processor is built around the following fundamental components:

- **Register File**: A set of 32 general-purpose 32-bit registers, including a hardwired zero register (R0). It provides two read ports and one write port, enabling simultaneous access to operands and storing results efficiently.
- **Arithmetic Logic Unit (ALU)**: A 32-bit combinational unit that performs all arithmetic and logical operations required by the instruction set, such as addition, subtraction, AND, OR, comparisons, and shifts.
- **Instruction Memory**: A word-addressable, read-only memory that stores the binary instructions to be executed. Instructions are fetched based on the current value of the Program Counter (PC).
- **Data Memory**: A word-addressable memory block used to read from and write to data values during program execution. Access is performed using the LW and SW instructions.
- **Program Counter (PC)**: A 20-bit register that holds the address of the current instruction. It increments by 1 on each cycle unless modified by a branch or jump instruction.
- **Control Unit**: Decodes the instruction and generates control signals that coordinate the behavior of all other components. It includes sub-units:
    - **Main Control Unit** – generates signals for register selection, memory access, and ALU input control.
    - **ALU Control Unit** – determines the specific ALU operation based on the instruction opcode and function field.
    - **PC-control Unit**– manages branch and JALR decisions using comparison outputs from the ALU (such as Zero or Set flags) to Set

the next PC.

Datapath Design

All components are interconnected through a carefully constructed datapath that includes:

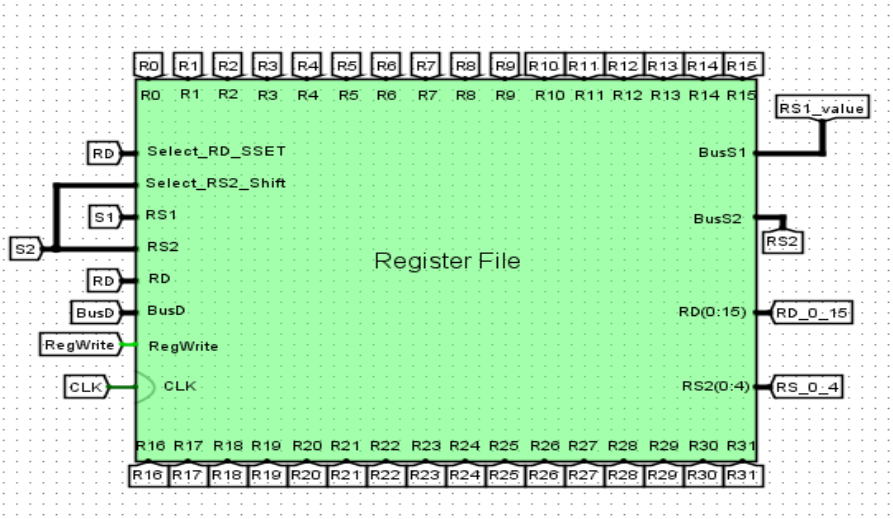
- **Multiplexers**: Used for selecting between multiple sources of data based on control signals.
- **Sign and Zero Extenders**: Extend 16-bit immediate values to 32 bits for ALU operations or address calculation.
- **Splitters**: Extract specific fields (like opcode, register addresses, immediate values) from the 32-bit instruction.
- **Pipeline-unaware sequential flow**: Since this is a single-cycle processor, each instruction completes all stages of execution (fetch, decode, execute, memory, and write-back) in one clock cycle without overlapping other instructions.

Instruction Flow

1. **Fetch**: The instruction is fetched from the instruction memory using the address in the PC.
2. **Decode**: The instruction is decoded to determine the required operation, source registers, and control signals.
3. **Execute**: The ALU performs the arithmetic or logical operation, or computes the effective address.
4. **Memory Access**: For load or store instructions, the data memory is accessed.
5. **Write-Back**: The result of the operation or memory read is written back into the register file.

Single-Cycle Architecture

- **Register File**



Register File Overview

The **Register File** is a central component in the processor architecture, responsible for providing fast, temporary storage and retrieval of data required during instruction execution. It consists of **32 general-purpose 32-bit registers** labeled from **R0 to R31**, with **R0** being hardwired to zero.

This Register File is designed to support:

- **Two read operations**
- **One write operation** per clock cycle, all managed through dedicated input and control signals.

Inputs

- **S1**: Selects the first source register (RS1).
- **S2**: Selects the second source register (RS2).
- **RD**: Selects the destination register to be written to.
- **BusD**: Carries the 32-bit data to be written into the register specified by RD.
- **Select_RD_SSET**: A control signal used when writing to the destination register in special instructions like SSET.

- **Select_RS2_Shift**: A control signal used for shift operations to extract specific bits from RS2.
- **RegWrite**: Enables the write operation. When high and triggered by the clock, the data on BusD is written into the selected destination register.
- **CLK**: The system clock signal that synchronizes the write operation.

Outputs

- **R0 to R31**: These represent the values of all 32 registers, typically used for display, debugging, or testing purposes in the Logisim simulation.
- **RS2**: The full 32-bit value read from the register selected by S2.
- **RS2(0–4)**: The lower 5 bits of RS2, often used in shift instructions where only a portion of the register value is needed.
- **RD(0–15)**: The lower 16 bits of the register selected by RD, which can be used in immediate-based operations or reduced-width ALU operations.

Design Considerations

- The use of **multiplexers** enables dynamic selection of source registers.
- The **write mechanism** is controlled by the RegWrite signal and synchronized with the rising edge of the clock.
- The presence of control signals like Select_RD_SSET and Select_RS2_Shift allows the register file to support **extended instruction functionalities** such as loading partial values or controlling shift operations.
- **R0 is fixed to zero** and cannot be written to, simplifying instruction logic such as clearing a register or implementing certain arithmetic comparisons.

Register File Structure

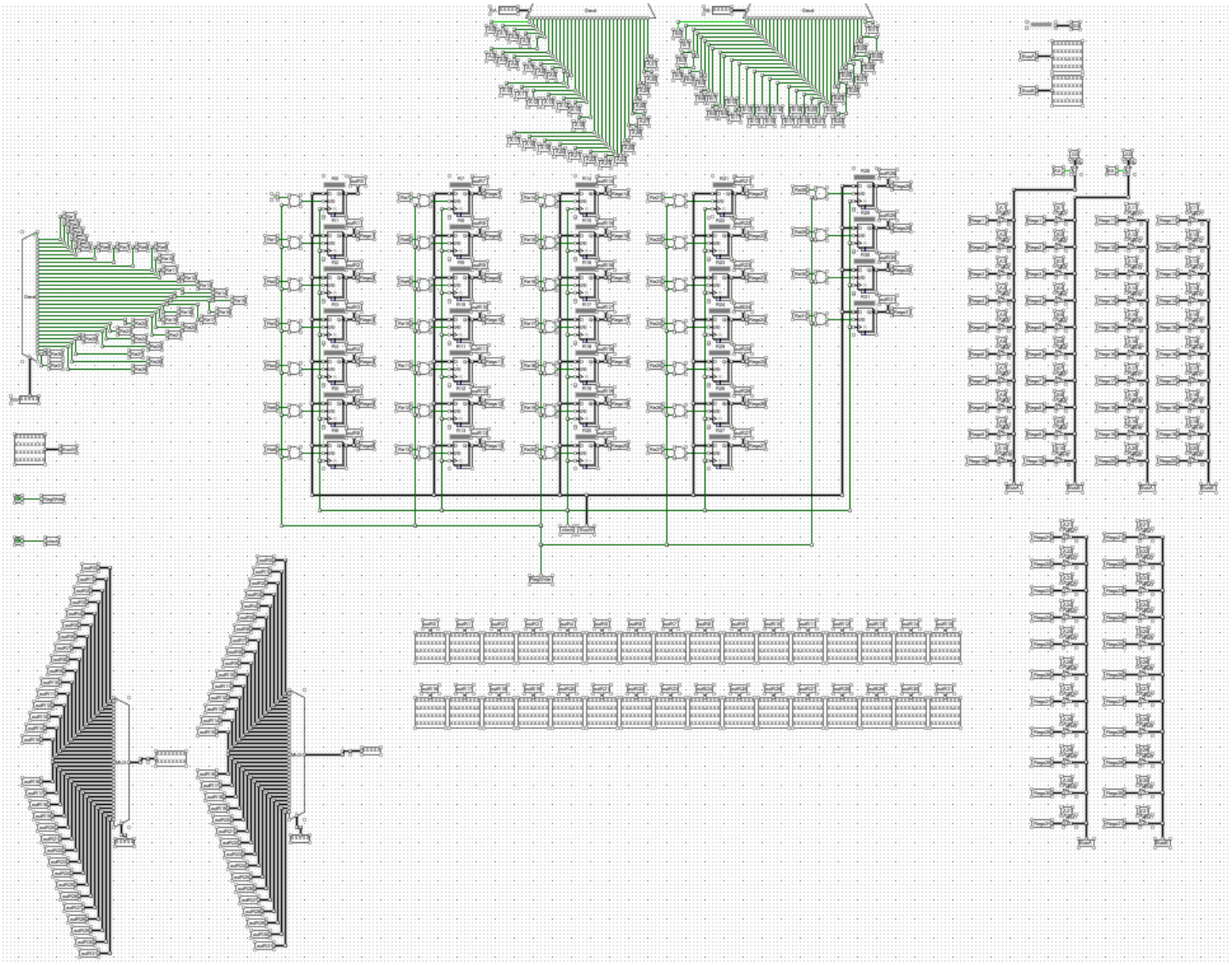
The **Register File structure** is designed to support fast and reliable access to register data, enabling the processor to read from two registers and write to one register within the same clock cycle. The architecture is modular, scalable, and constructed with clarity in mind, as reflected in the Logisim simulation.

Internal Organization

At the heart of the Register File are **32 individual 32-bit registers** labeled R0 to R31. These registers are organized in a grid-like layout to facilitate decoding, multiplexing, and data routing efficiently.

Key characteristics of the structure include:

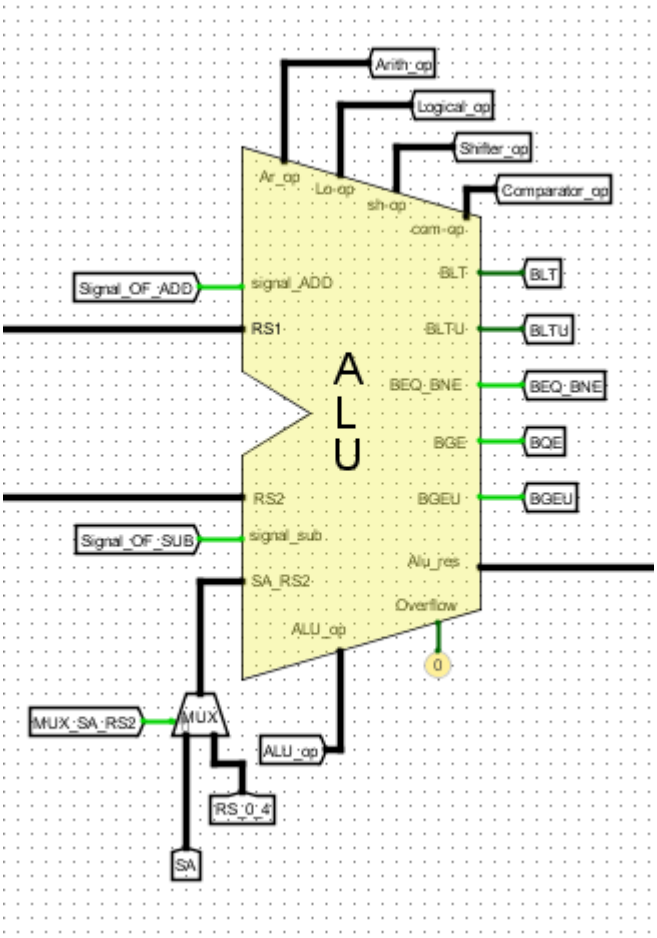
- **Dual Read Ports**: Two source registers are read in parallel using select inputs S1 and S2.
- **Single Write Port**: One destination register is written using the RD select input, BusD data line, and RegWrite signal.



• Arithmetic and Logic Unit (ALU)

Overview

The Arithmetic Logic Unit (ALU) is a critical component of a computer's central processing unit (CPU), serving as the computational core responsible for executing a wide range of operations. These operations include arithmetic calculations (e.g., addition and subtraction), logical operations (e.g., AND, OR), shift operations (e.g., left shift, right shift), and comparisons (e.g., equality, greater than). The ALU is a combinational logic circuit that processes data inputs based on control signals, producing results that are either stored in registers or used to guide the processor's control flow, such as in conditional branching



ALU Structure

The ALU is composed of several specialized functional units, each designed to handle a specific type of operation. These units work together, with their outputs integrated through a multiplexer (MUX) to produce the final ALU result. Below is a breakdown of the key components:

1. Arithmetic Unit:
   - Purpose: Performs arithmetic operations like addition and subtraction.
   - Inputs:
     -Data inputs: RS0 and RS1 (register sources).
     -Control signals: Signal of ADD, Signal of SUB, and Overflow.
   - Outputs:

-Value: The result of the arithmetic operation.
-signal_select_arith_op: A control signal for the MUX.

- Functionality: The Arithmetic Unit executes operations based on control signals. For example, an active Signal of ADD triggers the addition of RS0 and RS1, while Signal of SUB results in subtracting RS1 from RS0. The Overflow signal indicates if the result exceeds the representable range.

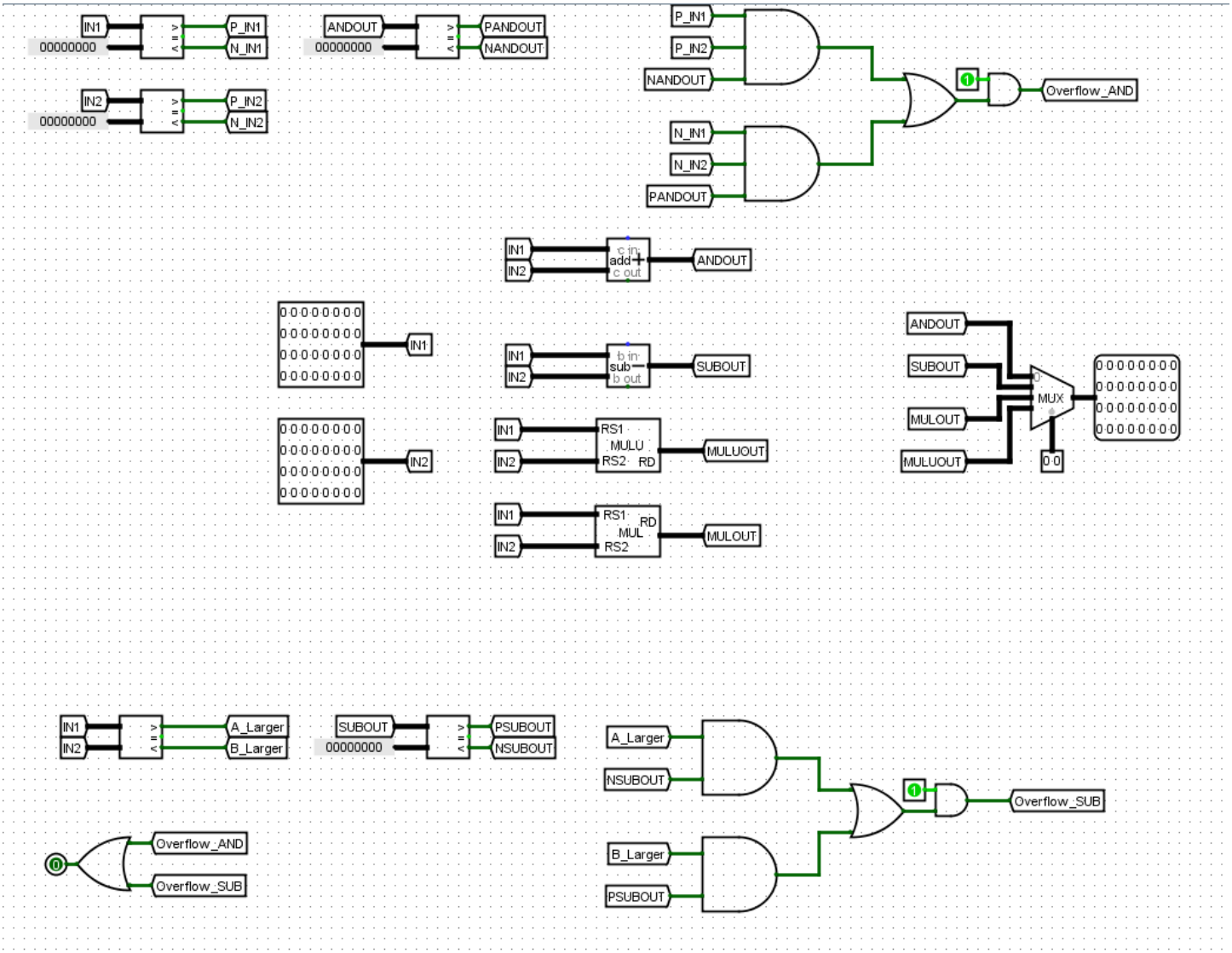
MULU: we use to multiplication unsigned number (32bit*32bit) and remove multiplication of higher bits because want the output 32bit not 64bit, we use equation:

Let IN1 = A , IN2 = B

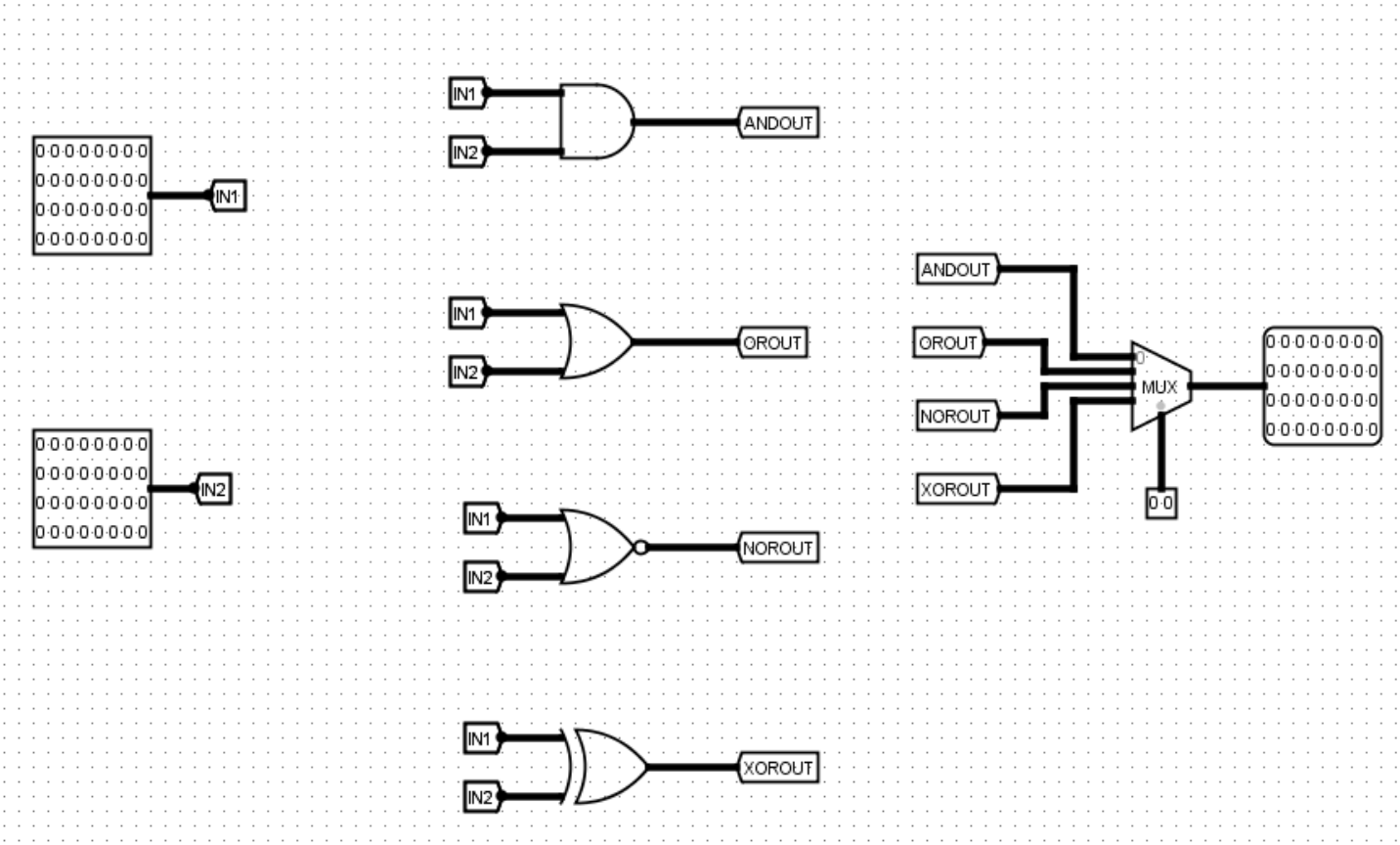
$A = A_{lo} + A_{hi} * 2^{16}$
$B = B_{lo} + B_{hi} * 2^{16}$
$A * B = (A_{lo} * B_{lo}) + (A_{lo} * B_{hi} * 2^{16}) + (B_{lo} * A_{hi} * 2^{16}) + (A_{hi} * 2^{16} * B_{hi} * 2^{16})$
$= (A_{lo} * B_{lo}) + (A_{lo} * B_{hi} * 2^{16}) + (B_{lo} * A_{hi} * 2^{16}) + (A_{hi} * B_{hi} * 2^{32})$

MUL: we used block (31bit*31bit) and generate sign bit by XOR gate

Overflow:
-Add: when inputs the same sign and outputs the opposite sign and vice versa
-SUB: when the output sign is not logical

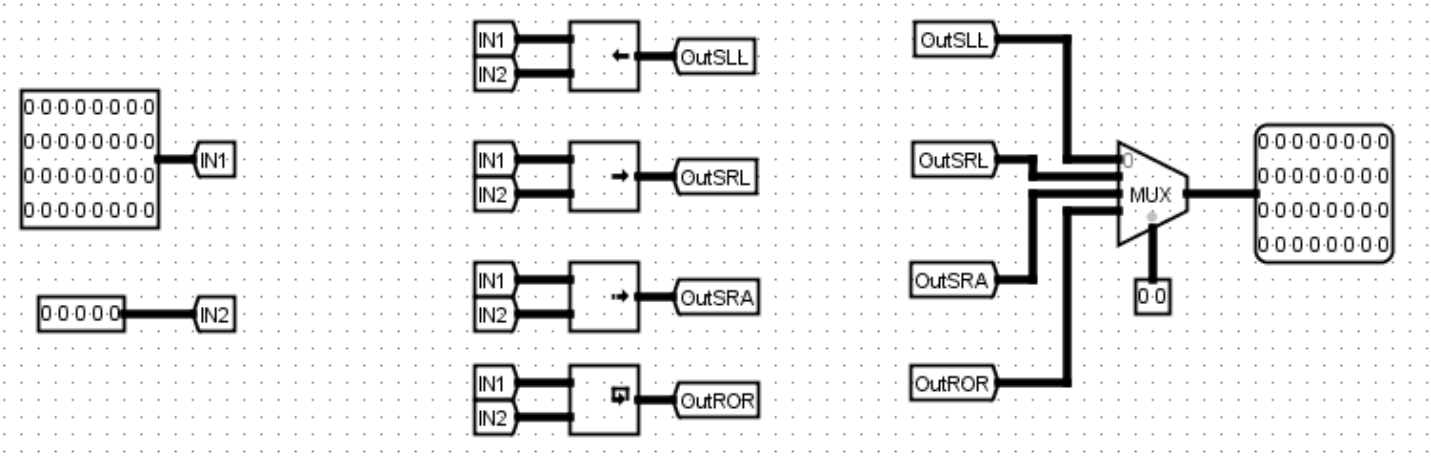


2. Logical Unit:
   - Purpose: Executes logical operations such as AND, OR, and XOR.
   - Inputs:
     -Data inputs: RS0 and RS1.
     -Control signals: Implied signals like Signal of AND (e.g., via Lo_op).
   - Outputs:
     -Value: The result of the logical operation.
     -signal_select_logic_op: A control signal for the MUX.
   - Functionality: This unit performs bitwise logical operations on RS0 and RS1, with the specific operation determined by control signals.



3. Shifter:

- Purpose: Handles shift operations, such as left shift or right shift.
- Inputs:
  - -Data inputs: RS0 and RS1.
  - -Shift amount: SAR0 and SAR1 (shift amount registers).
- Outputs:
  - -Value: The result of the shift operation.
  - -signal_select_shift_op: A control signal for the MUX.
- Functionality: The Shifter adjusts the bit positions of the input data by an amount specified by SAR0 and SAR1, with the type of shift (e.g., logical or arithmetic) dictated by signals like sh-op.

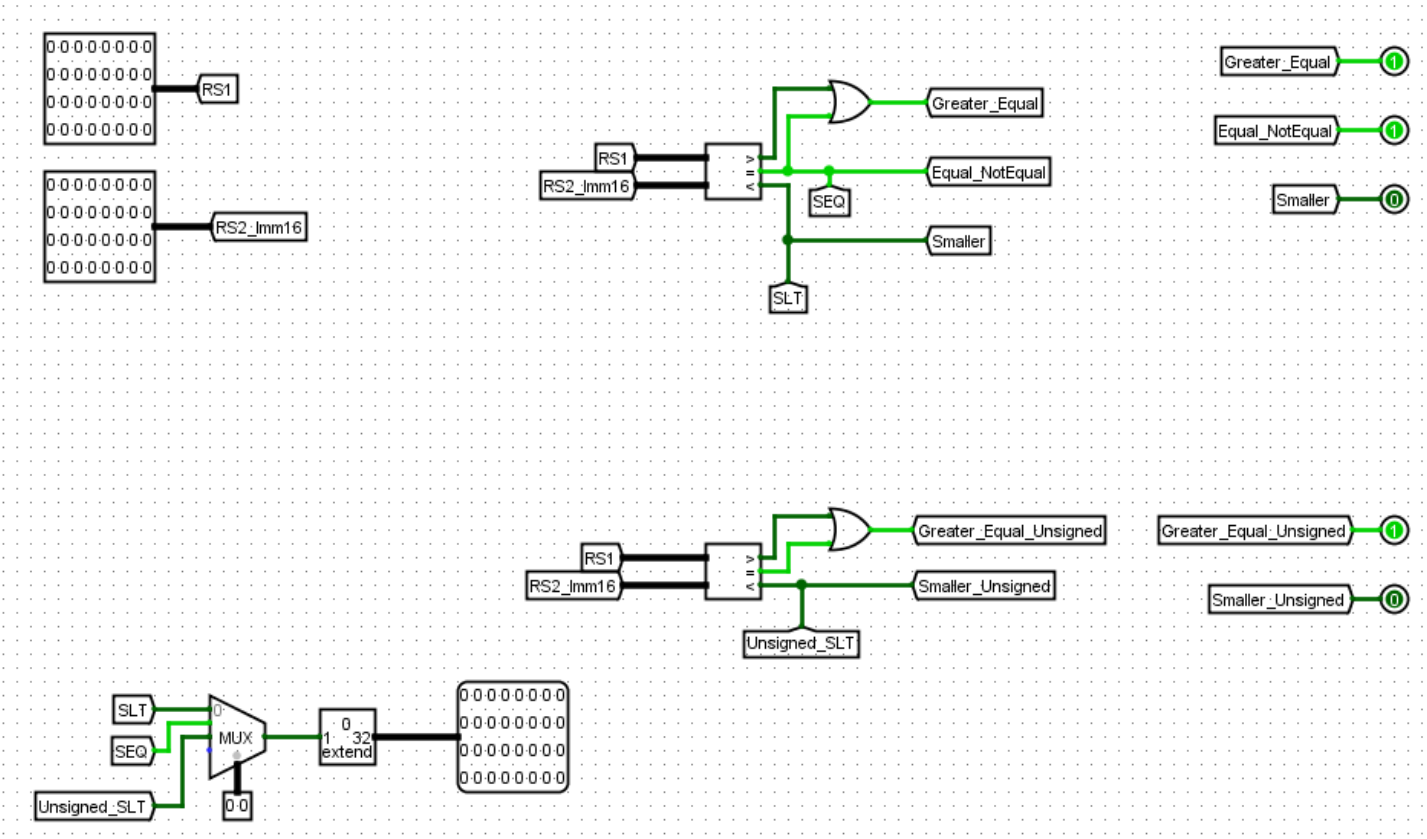


4. Comparator:
   - Purpose: Performs comparison operations to evaluate relationships between inputs.
   - Inputs:

     -Data inputs: RS0 and RS1.
- Outputs:

  -Comparison flags: Smaller, Smaller_Unsigned, Equal_NotEqual, Greater_Equal, Greater_Equal_Unsigned, and RD0_or_1.

  -signal_select_comp_op: A control signal for the MUX.
- Functionality: The Comparator generates flags indicating whether RS0 is less than, equal to, or greater than RS1, for both signed and unsigned interpretations. These flags, such as BLT (Branch if Less Than) or BEQ_BNE (Branch if Equal or Not Equal), are vital for conditional branching.



Multiplexer (MUX):

- Purpose: Selects the final ALU output from the various units.
- Inputs: Value outputs from the Arithmetic Unit, Logical Unit, Shifter, and Comparator.
- Control Signal: select_operation, derived from unit-specific control signals (e.g., signal_select_arith_op).
- Output: Alu_res, the final ALU result.

The MUX, labeled ALU_result, ensures that the appropriate unit's output is selected as the ALU's result based on the operation being performed.

Data Flow:

The ALU's data flow is primarily directional:

- Inputs: Enter from the left as RS0 and RS1 (data) and various control signals (e.g., signal_ADD, ALU_op).
- Processing: Each unit processes the inputs according to the active control signals.
- Output: The MUX combines the results, outputting Alu_res on the right,

while comparison flags are generated separately for control flow decisions.

Control Signals:

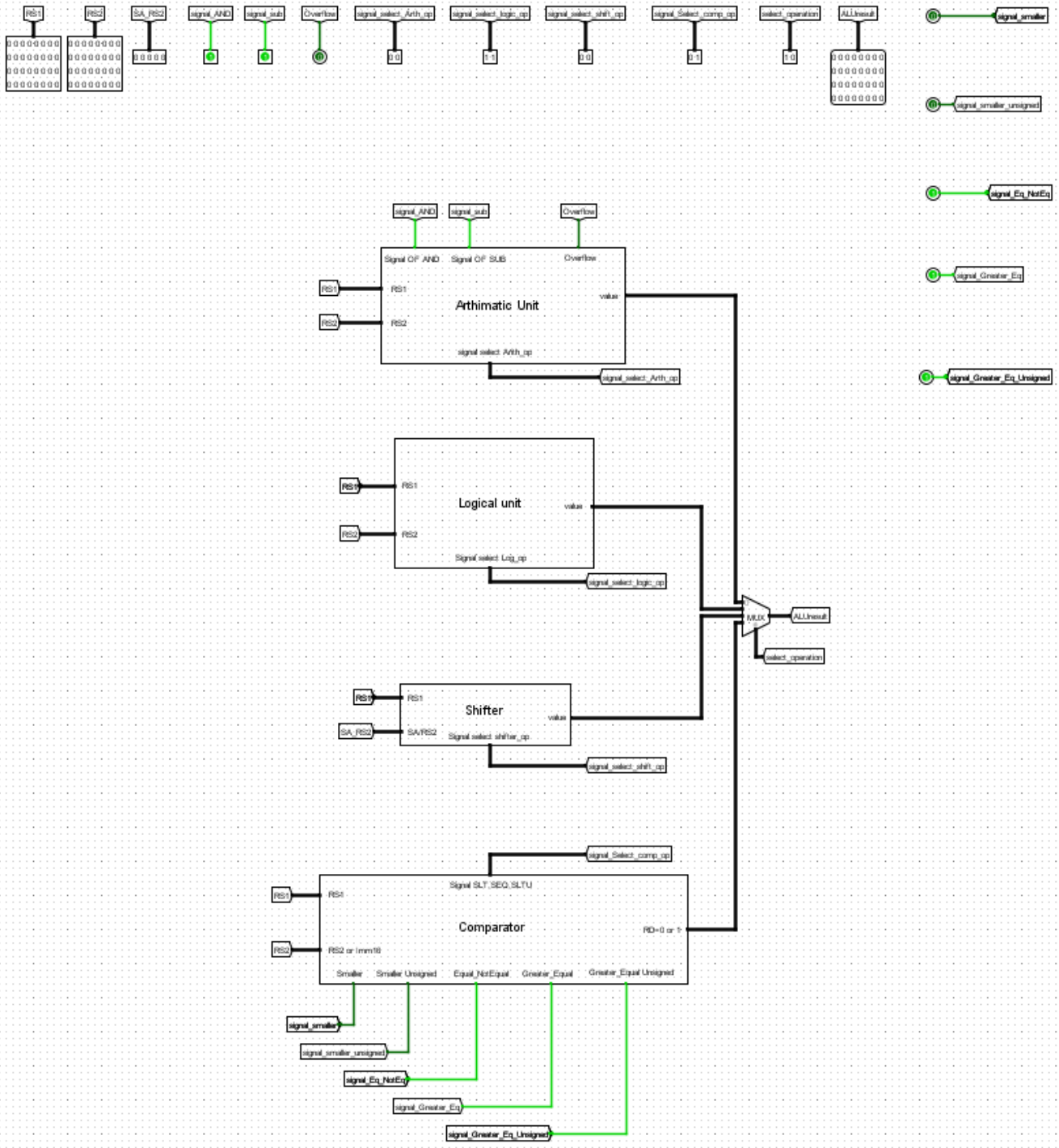
The ALU's operation is governed by control signals from the CPU's control unit, including:

- Operation Selectors: Ar_op (arithmetic), Lo_op (logical), sh-op (shift), com-op (comparison).
- Specific Signals: signal_ADD, signal_sub, etc.
- Primary Control: ALU_op, which orchestrates the overall operation.

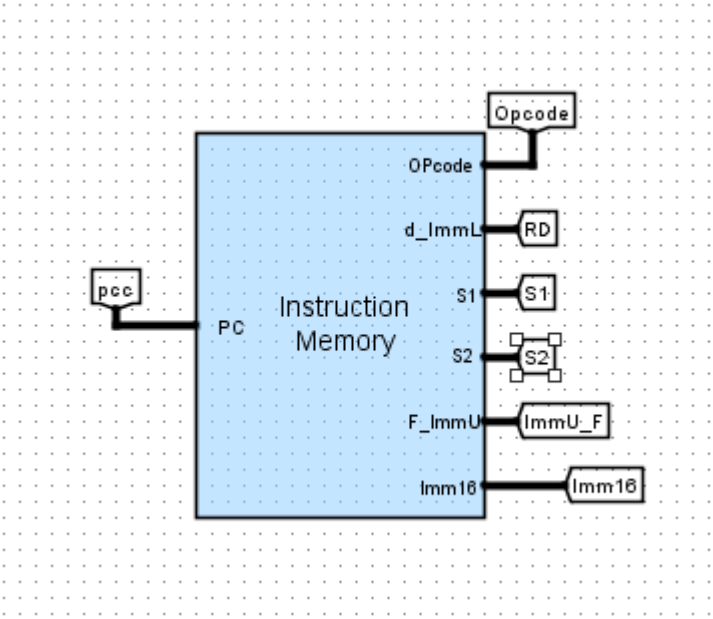
These signals configure the ALU for the required task, ensuring precise execution of instructions.

Inputs and Outputs:

- Inputs:
  - -Data: RS0, RS1, and shift amounts SAR0, SAR1.
  - -Control: Signals like signal_ADD, signal_sub, and ALU_op.
- Outputs:
  - -Result: Alu_res, the primary computational output.
  - -Flags: Comparison outputs like BLT, BGE, and Overflow, used for branching and status checking.


RS1
RS2
SA_RS2
signal_AND
signal_sub
Overflow
signal_select_Arth_op
signal_select_logic_op
signal_select_shift_op
signal_Select_comp_op
select_operation
ALUresult
signal_smaller
signal_smaller_unsigned
signal_Eq_NotEq
signal_Greater_Eq
signal_Greater_Eq_Unsigned
Signal OF AND
Signal OF SUB
Arthimatic Unit
value
signal select Arth_op
Logical unit
Signal select Lcg_op
Shifter
SA/RS2
Signal select shifter_op
MUX
Signal SLT SEQ SLTU
Comparator
RS2 or imm16
Smaller
Smaller Unsigned
Equal_NotEqual
Greater_Equal
Greater_Equal Unsigned

• Struction Memory



• Overview

The **Instruction Memory** is a vital component of the single-cycle processor that stores the program instructions and provides the necessary fields to the processor during execution. It is addressed using the **Program Counter (PC)** and outputs the decoded fields needed to control the datapath and execute operations properly.
Instruction Memory is structured to deliver all essential parts of an instruction simultaneously, ensuring that each cycle can execute one instruction efficiently.

Inputs

- **PC (Program Counter)**:
  A 32-bit address input that specifies which instruction to fetch from memory. The PC is updated every clock cycle to point to the next instruction unless modified by jump, branch, or other control signals.
- **pcc**:
  Additional input (likely representing a pipelined or next cycle PC value) feeding the memory module.

Outputs

- **Opcode**:
  A 6-bit field representing the type of instruction to execute (e.g., arithmetic, load/store, branch).
- **RD**:
  A 5-bit destination register field, specifying the register that will be written after instruction execution.
- **S1**:
  A 5-bit source register field for the first operand.
- **S2**:
  A 5-bit source register field for the second operand.
- **d_ImmL**:
  Immediate data loaded from the instruction, representing lower bits used in immediate-based operations.
- **F_ImmU**:
  An unsigned immediate field for specific types of operations needing zero-extension.
- **Imm16**:
  A 16-bit immediate value, often used in load/store operations, branches, and some arithmetic instructions.

Instruction Format

Depending on the type of instruction (R-type, I-type, or J-type), different combinations of these fields are used.
For example:

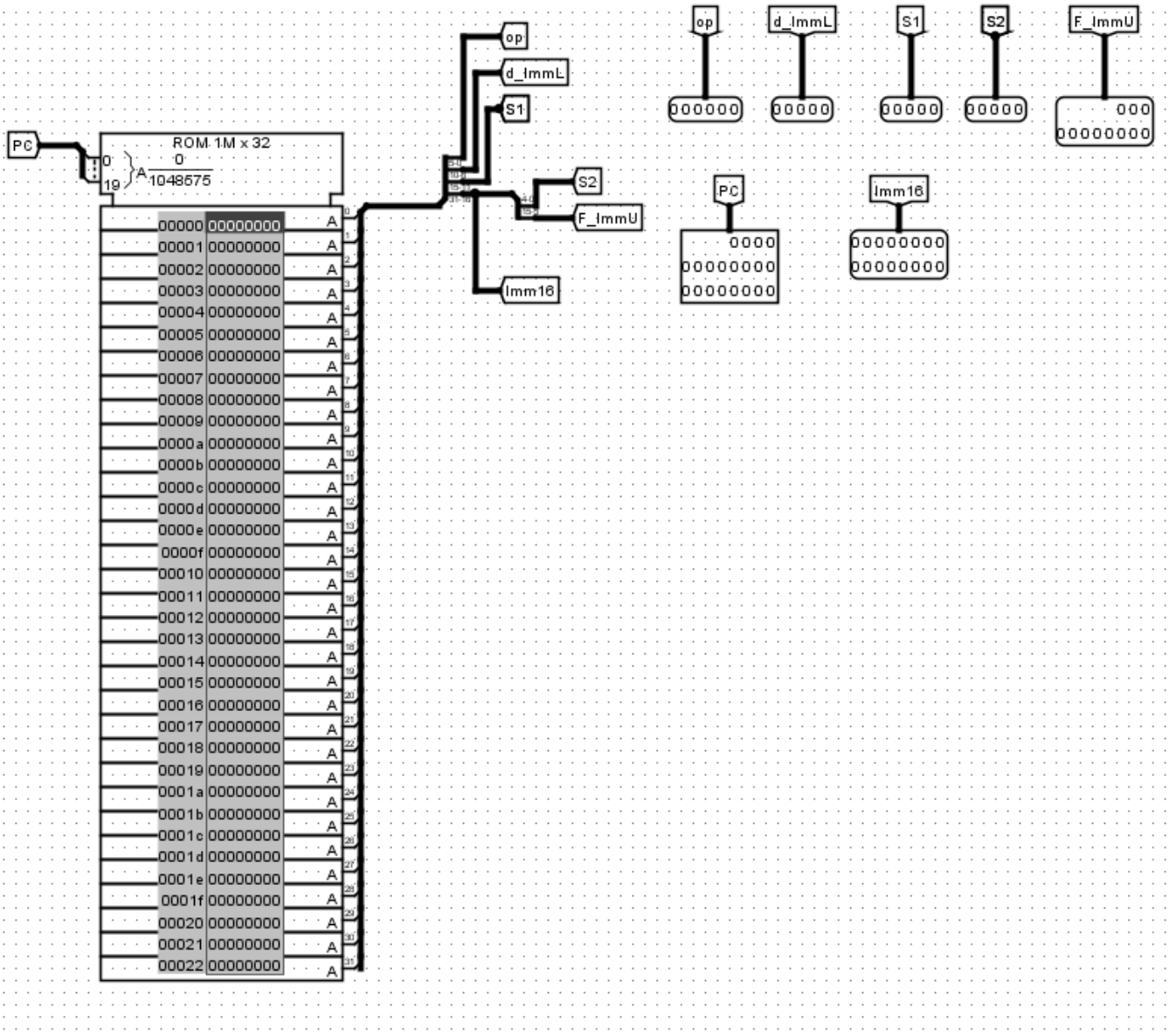
- R-Type instructions use Opcode, S1, S2, and RD.
- I-Type instructions use Opcode, S1, RD, and Immediate fields like Imm16.
- Specialized fields like F_ImmU and d_ImmL are used for fine-grained immediate control or extended operations.

Design Considerations

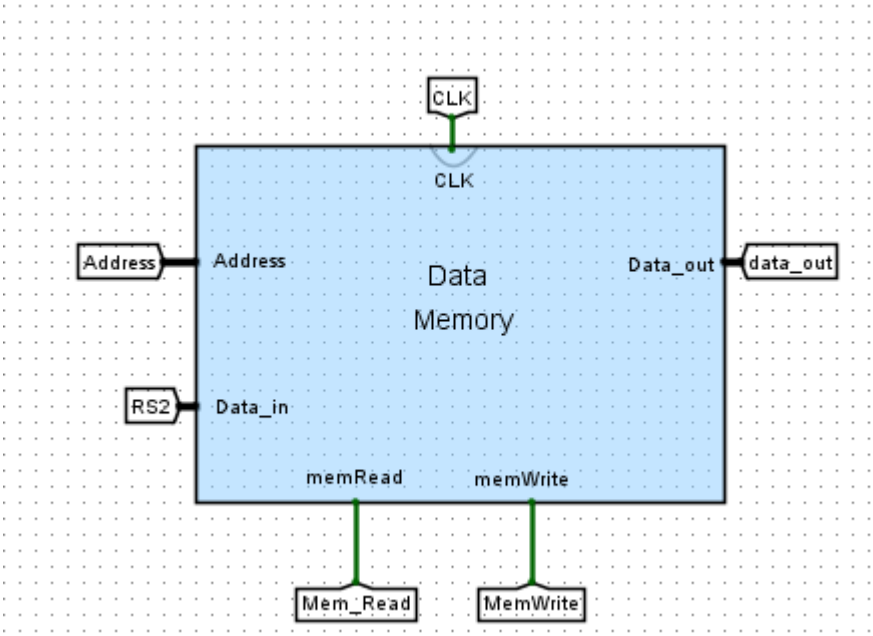
- **Parallel Output**:
  All necessary fields are decoded and made available in parallel to minimize latency and streamline datapath control.
- **Flexibility**:
  Fields like Imm16, d_ImmL, and F_ImmU provide flexibility for

supporting multiple instruction types without complicating control logic.

- **Alignment**:
  The PC input is word-aligned, ensuring correct instruction fetching and avoiding misaligned memory accesses.



• Data Memory



Overview

The **Data Memory** module is responsible for storing and retrieving data during program execution. It interacts with load (LW) and store (SW) instructions, enabling the processor to work with external data beyond the register file.

Data Memory is **word-addressable** and operates synchronously with the system clock (CLK) to ensure reliable and coordinated memory access.

Inputs

- **Address**:
  A 32-bit address input that specifies the memory location for the read or write operation. The address is typically calculated by the ALU based on the base register and offset.

- **RS2 (Data_in)**:

The 32-bit data input that carries the value to be written into memory during a store (SW) operation.

- **MemRead**:
  A control signal that enables a memory read operation. When MemRead is asserted, the data at the specified Address is read and output to the processor.

- **MemWrite**:
  A control signal that enables a memory write operation. When MemWrite is asserted, the data on RS2 (Data_in) is written into the memory at the specified Address.

- **CLK**:
  The system clock that synchronizes memory write operations. Writes occur on the rising edge of the clock to ensure timing correctness.

Outputs

- **Data_out**:
  A 32-bit output that provides the data read from the memory when a load (LW) instruction is executed.

Functional Behavior

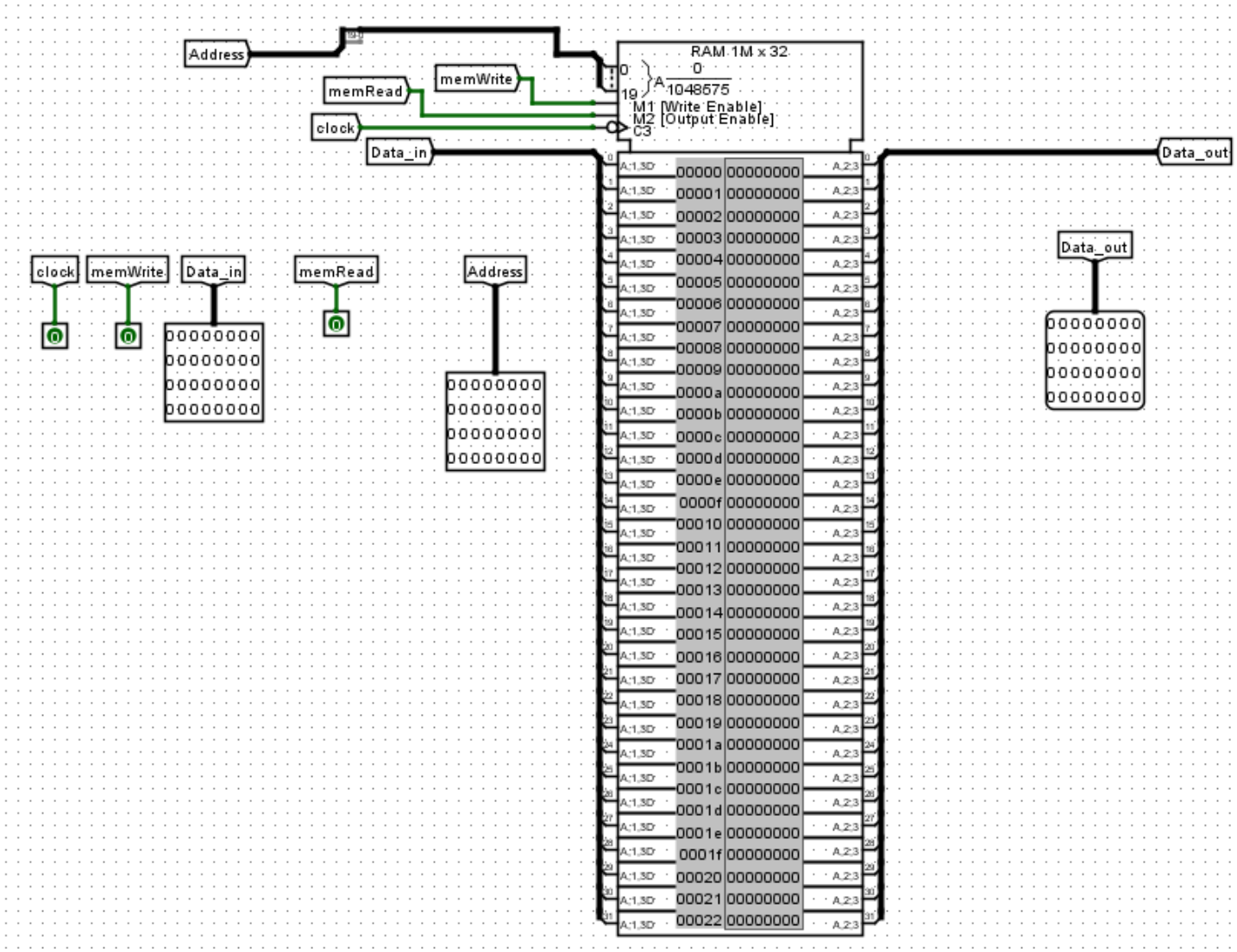
- **Load (Read Operation)**:
  - If MemRead = 1 and MemWrite = 0, the memory returns the data located at the specified Address onto the Data_out bus.

- **Store (Write Operation)**:
  - If MemWrite = 1 and MemRead = 0, the value from RS2 (Data_in) is written into the memory location specified by Address, synchronized with the clock edge.

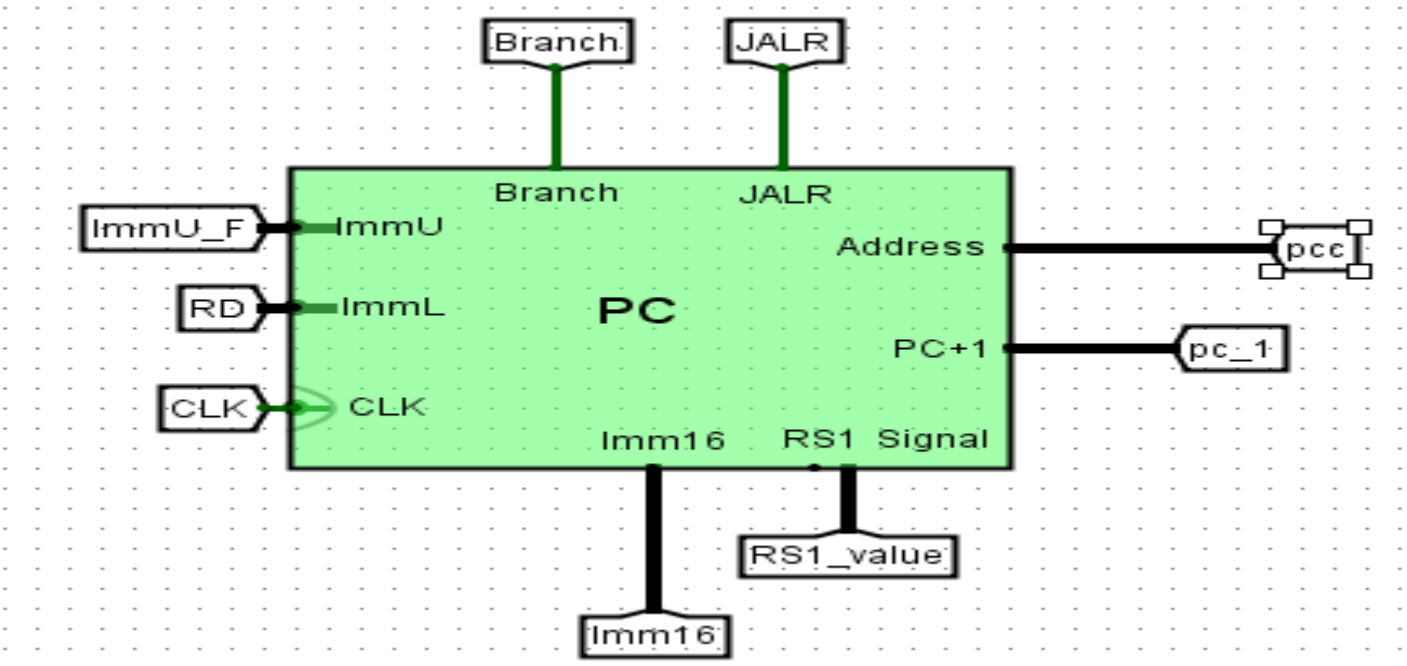
- **Idle (No Operation)**:
    - If both MemRead = 0 and MemWrite = 0, no memory operation occurs, preserving the current state.

Design Considerations

- **Single-Ported Memory**:
  This design uses a single port for both read and write operations, controlled via MemRead and MemWrite signals.
- **Synchronization**:
    - **Writes** are synchronized with CLK to avoid race conditions.
    - **Reads** are typically asynchronous (appearing after address is stabilized), but final latching happens in subsequent stages if needed.
- **Word Alignment**:
  Addresses are expected to be word-aligned to ensure correct access to full 32-bit words.


Address
memRead
memWrite
clock
Data_in
RAM 1M x 32
0
19
A
0
1048575
M1 [Write Enable]
M2 [Output Enable]
C3
Data_out
clock
memWrite
Data_in
memRead
Address
Data_out

- **Program Counter**



PC Overview

The **Program Counter (PC)** is a fundamental component of the processor that keeps track of the current instruction's address. It is responsible for updating the instruction address every clock cycle and correctly handling jumps, branches, and procedure calls.

In this design, the PC is flexible and supports:

- Normal sequential instruction execution (PC + 1),
- Conditional branching,
- Register-based jumping (JALR instructions).

Inputs

- ImmU_F:
  Upper part of the immediate value used in branch address calculations.
- ImmU:
  Full unsigned immediate value used for calculating branch targets.
- RD:
  Destination register address, primarily used during jump and link operations (JALR).
- ImmL:
  Lower part of the immediate value for building complete branch offsets.
- CLK:
  Clock signal controlling PC updates. The PC changes at the rising edge of the clock.
- Branch:
  A control signal that, when asserted, enables branching to a new address calculated using immediate values.
- JALR:
  A control signal for performing a jump and link to a register address plus an immediate offset.
- RS1 Signal:
  Carries the value of the RS1 register, necessary for dynamic branch or jump address calculations.
- RS1_value:
  The 32-bit value read from the RS1 register, used specifically for JALR jumps.
- Imm16:
  A 16-bit immediate value that is used in address calculations during branching or jumping.

Outputs

- Address:
  The 32-bit next instruction address output, used to fetch the next instruction.
- PC+1:
  The incremented program counter value, typically used in normal sequential execution when no branch or jump is taken.

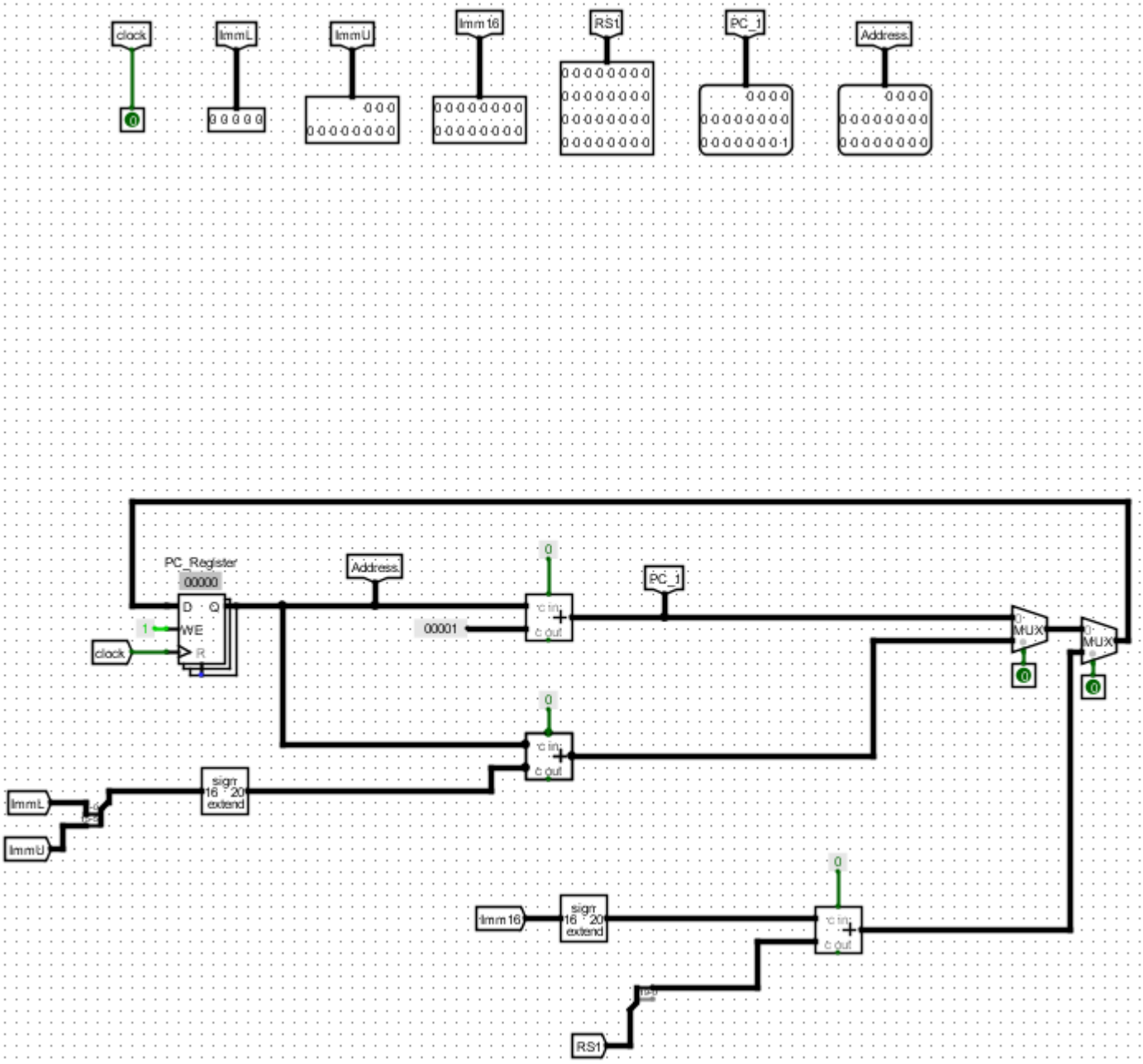
- pcc:
  Represents the current program counter value (before incrementing or jumping), often used for monitoring or pipeline forwarding.
- pc-1:
  Represents the decremented program counter value if needed (used mainly in error checking or specialized addressing).

PC structure & datapath

The **Program Counter (PC)** module is carefully structured to support flexible instruction flow within a single clock cycle. Its design enables normal sequential execution, conditional branching, and register-based jumps (e.g., JALR) while maintaining synchronization with the system clock.

Internal Structure

- **Register Unit**:
  The core of the PC module is a 32-bit register that stores the current instruction address. It updates either sequentially (PC + 1) or with a branch/jump target.
- **Address Calculation Unit**:
  - For normal execution, the PC simply increments by 1.
  - For branching, the immediate fields (ImmU, ImmL) are combined and sign-extended to generate the branch offset.
  - For JALR jumps, the PC is set to the sum of RS1_value and Imm16.
- **Multiplexers (MUXs)**:
  Multiplexers are used to select the next PC value:
  - Between PC + 1, branch target address, or Branch address.
  - Controlled based on Branch and JALR signals.
- **Control Logic**:
  - Evaluates whether the PC should simply increment, branch, or jump based on the active control signals.
  - Ensures smooth flow and accurate redirection of the instruction fetch address.



The **PC Datapath** integrates seamlessly into the overall processor datapath, providing the address for instruction fetch and supporting conditional control flow.

Normal Instruction Flow

1. The PC holds the address of the current instruction.

2. Each clock cycle (unless otherwise controlled), the PC value increments by 1

3. The incremented PC (PC+1) is fed back into the PC register for the next instruction fetch.

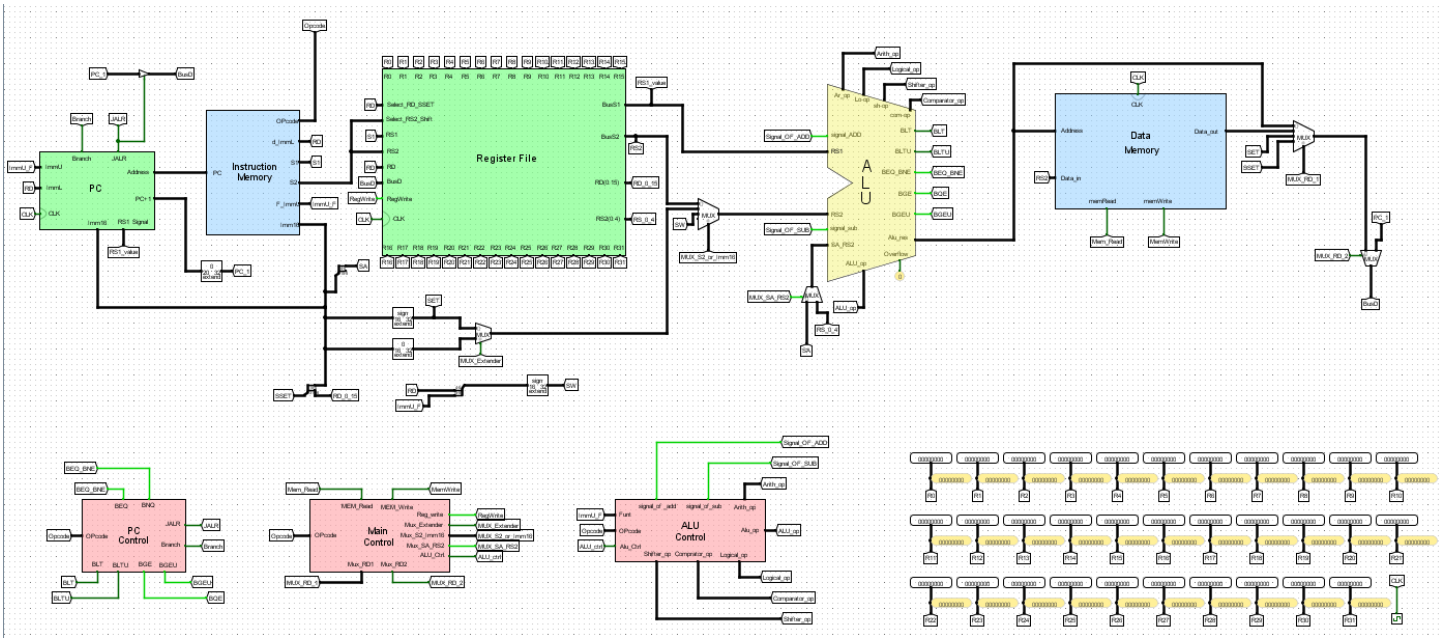
Branch Flow

1. If a branch condition is satisfied (evaluated by the ALU and Branch Control Unit), the Branch control signal is asserted.
2. The immediate fields (ImmU, ImmL) are sign-extended and added to the current PC value.
3. The resulting branch address replaces the current PC value on the next clock cycle, causing a jump to a new instruction.

JALR (Jump and Link Register) Flow

1. When a JALR instruction is executed, the JALR signal is asserted.
2. The branch target address is calculated by adding the value of RS1_value and the Imm16 immediate field.
3. This address is loaded into the PC, effectively performing a jump to a computed location (often used for function calls).

- **Data Path**



Understanding the Datapath of a Processor:

The data path of a processor is a critical component that outlines how data flows through various functional units during the execution of instructions. Based on the provided image attachment descriptions, this response presents a comprehensive overview of the data path, focusing on its structure, key components, and operational flow. This data path likely represents a simplified RISC (Reduced Instruction Set Computer) architecture, such as MIPS or RISC-V, and includes all essential elements for instruction processing.

Overview of the Datapath:

The data path is the hardware framework that facilitates the execution of instructions in a processor. It consists of interconnected components that handle data movement and computation across five primary stages:

1. Instruction Fetch: Retrieving the instruction from memory.

2. Instruction Decode: Interpreting the instruction and preparing operands.
3. Execution: Performing the required computation or operation.
4. Memory Access: Reading from or writing to memory if needed.
5. Write-Back: Storing the result back into the register file.

These stages are executed in a synchronized manner, typically governed by a clock signal (CLK), with control signals directing the flow of data through the data path.

Key Components of the Datapath:

1. Program Counter (PC):
   - Function: Holds the memory address of the next instruction to be fetched.
   - Inputs: Updated by sequential increments, branch targets, or jump addresses (e.g., "Branch," "JALR," "ALU Result").
   - Outputs: Sends the address to the Instruction Memory.
   - Role: Initiates the fetch stage and manages control flow for branches and jumps (e.g., "BLT" for Branch if Less Than, "BEQ" for Branch if Equal).
2. Instruction Memory:
   - Function: Stores the program instructions and retrieves the instruction at the address provided by the PC.
   - Inputs: Address from the PC.
   - Outputs: Instruction sent to the Register File and Control Unit for decoding.
   - Role: Provides the raw instruction data for processing.
3. Register File:

- Function: A set of 32 registers (R0–R31) that store operands and results.
- Structure:

  -Two read ports (RS1, RS2) for fetching operand values.

  -One write port (RD) for storing results.
- Control Signals:

  -"RegWrite": Enables writing to the register file.

  -"ReadReg1," "ReadReg2," "WriteReg": Specify register addresses.
- Role: Supplies operands to the ALU and receives results from computations or memory.

4. Arithmetic Logic Unit (ALU):

- Function: Performs arithmetic, logical, shift, and comparison operations.
- Structure:
  - Comprises specialized units:
    - Arithmetic Unit: Handles operations like addition ("Signal of ADD") and subtraction ("Signal of SUB").
    - Logical Unit: Performs operations like AND ("Signal of AND") and OR.
    - Shifter: Manages bit-shifting operations (e.g., left shift, right shift).
    - Comparator: Evaluates conditions (e.g., "Smaller," "Equal_NotEqual") for branches.
- A multiplexer (MUX) selects the output from these units based on the operation.

- Inputs:
    - Operand A: From RS1.
    - Operand B: From RS2 or an immediate value (via MUX).
- Outputs:
    - "ALU Result": The computed result.
    - Flags: "Overflow," "Zero," etc., for control flow decisions.
- Control Signals: "ALUop" from the ALU Control unit dictates the operation type.

5. ALU Control Unit:

- Function: Decodes the instruction's opcode and function codes to generate ALU control signals.
- Inputs: Opcode and function fields from the instruction.
- Outputs: Signals like "Signal of ADD," "Signal of SUB," and "signal_select_arith_op" to control the ALU.
- Role: Ensures the ALU performs the correct operation for the instruction.

6. Data Memory:

- Function: Manages load and store operations for data access.
- Inputs:
    - Address: From the ALU result (e.g., for address calculations).
    - Write Data: From the Register File (for stores).
- Outputs: Read Data to the Register File (for loads).
- Control Signals:

-"MemRead": Enables reading from memory.

-"MemWrite": Enables writing to memory.

- Role: Facilitates data storage and retrieval during memory-related instructions.

7. Multiplexers (MUX):

- Function: Direct data flow between multiple sources based on control signals.
- Key Instances:

  -ALU Input MUX: Selects between RS2 and an immediate value for the ALU's second operand ("ALUSrc").

  -Write-Back MUX: Chooses between the ALU result and Data Memory output for writing to the Register File.

  -PC MUX: Determines the next PC value (sequential, branch target, or jump target).

- Role: Provides flexibility in routing data through the datapath.

8. Control Unit:

- Function: Decodes the instruction and generates control signals to orchestrate the data path.
- Inputs: Opcode from the Instruction Memory.
- Outputs: "RegWrite," "MemRead," "MemWrite," "ALUSrc," "Branch," "ALUop," etc.
- Role: Acts as the conductor, ensuring each component performs its task at the right time.

9. Immediate Generation:

- Function: Extracts and sign-extends immediate values from the instruction (e.g., for "ADDI" or load/store offsets).
- Outputs: Immediate value ("Imm") sent to the ALU via a MUX.
- Role: Supports instructions requiring constant values.

Data Flow Through the Datapath:

The data path operates in a structured sequence to execute instructions:

1. Instruction Fetch:
   - The PC sends an address to the Instruction Memory.
   - The fetched instruction is sent to the Control Unit and Register File.

2. Instruction Decode:
   - The Control Unit decodes the instruction and generates control signals.
   - The Register File reads operands from RS1 and RS2 based on the instruction's register fields.

3. Execution:
   - The ALU performs the operation (e.g., addition, comparison) using operands from the Register File or an immediate value.
   - For branch instructions, the comparator evaluates conditions (e.g., "BLT," "BEQ") and updates the PC if necessary.

4. Memory Access:
   - For load/store instructions, the ALU result serves as the memory address.

- Data Memory reads ("MemRead") or writes ("MemWrite") data as directed.

5. Write-Back:

- The result (from the ALU or Data Memory) is written back to the Register File via a MUX, controlled by "RegWrite."

Control Signals and Their Roles:

Control signals synchronize and direct the datapath's operations:

- RegWrite: Enables writing to the Register File.
- ALUSrc: Selects the ALU's second operand (RS2 or immediate).
- MemRead/MemWrite: Control Data Memory access.
- Branch: Triggers PC updates for conditional branches.
- ALUop: Specifies the ALU operation (e.g., ADD, SUB, AND).

The clock signal (CLK) ensures these operations occur in a timed sequence, particularly for sequential elements like the PC and Register File.

Branch and Jump Handling:

The data path supports control flow instructions:

- Branches: Instructions like "BLT" (Branch if Less Than), "BEQ" (Branch if Equal), and "BNE" (Branch if Not Equal) use the ALU's comparator outputs ("Smaller," "Equal_NotEqual") to decide whether to update the PC with a branch target address.
- Jumps: Instructions like "JALR" (Jump and Link Register) set the PC to a computed address, often from the ALU result.

- # Control Units

Control units generate required signals which determine suitable data path to execute each instruction in the processor.

We have 3 control units each one generates special signals as follow:

1- Main Control Unit

Signals: (RegWrite, MemRead, MemWrite, ALUctrl, Mux_Extender, Mux_S2_Imm16, Mux_SA_RS2, Mux_RD1, Mux_RD2)

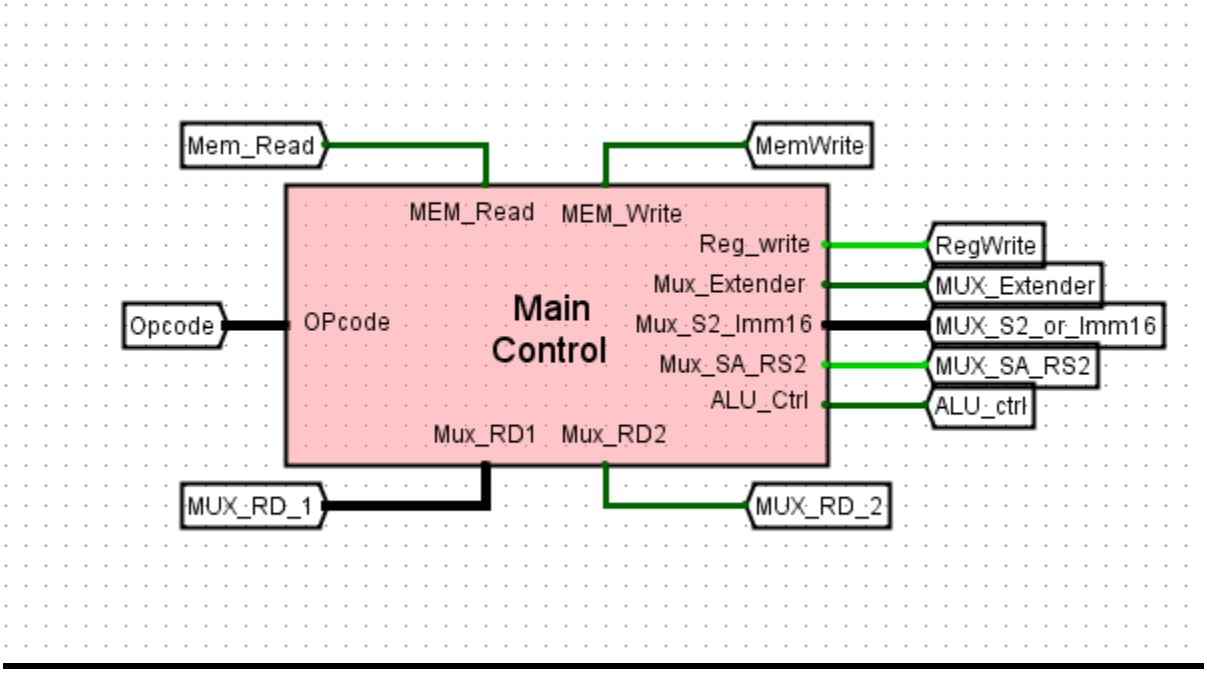
2- ALU Control Unit

Signals: (Signal_OF_ADD, Signal_OF_SUB, , Arthi_op, Logical_op, Shifter_op, Comparator_op, ALU_op)

3- PC Control Unit

Signals: (Branch, JALR)

• Main Control Unit



Overview

The main control unit is a fundamental component of a processor or microcontroller architecture, often considered the "brain" of the system. It is responsible for decoding instructions and generating the control signals necessary to execute those instructions by coordinating the processor's data path components, such as the register file, memory, and Arithmetic Logic Unit (ALU). Below is a detailed overview of its structure, functionality, and role based on the provided descriptions.

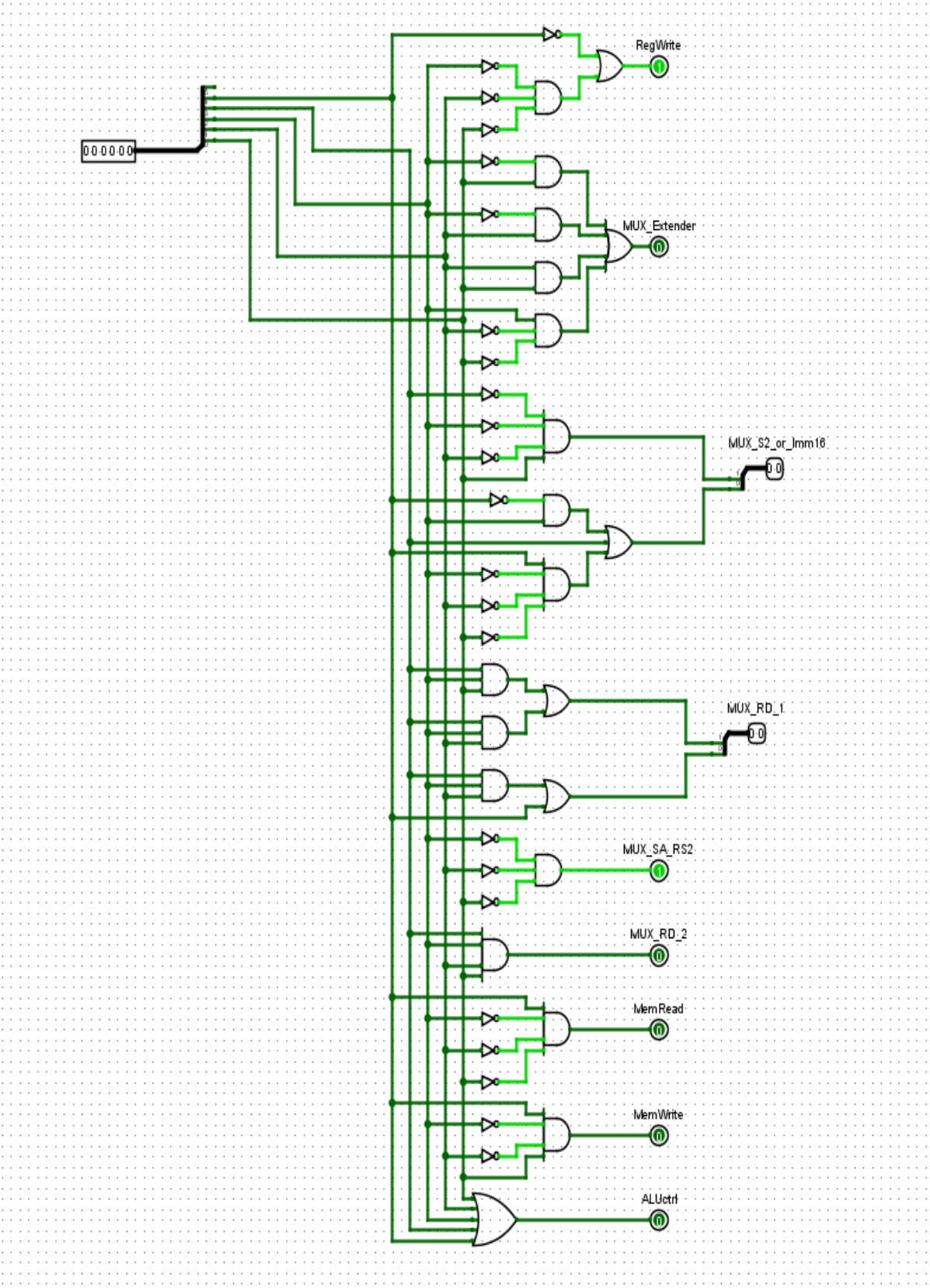
Datapath Interaction

The main control unit orchestrates the processor's datapath by sending control signals to:

Register File: RegWrite enables writes, while Mux_RD1 and Mux_RD2 select source registers.
Memory: MEM_Read and MEM_Write control data access.
ALU: ALU_Ctrl dictates the operation, ensuring correct computation or comparison.
For instance, in a load instruction (LW):

MEM_Read fetches data from memory.
RegWrite stores it in the register file.
Mux_S2_Imm16 selects the immediate offset for the memory address.

Structure


RegWrite
00000
MUX_Extender
MUX_S2_or_Imm16
0
MUX_RD_1
0
MUX_SA_RS2
MUX_RD_2
MemRead
MemWrite
ALUctrl

Key Components and Functionality

1. Input: OpCode
   - The main control unit receives the OpCode (operation code) as its primary input, typically extracted from the instruction register. The OpCode specifies the operation to be performed, such as arithmetic (e.g., ADD, SUB), memory access (e.g., LW, SW), or branching (e.g., BEQ, BNE).
   - This input drives the control unit's logic to determine the appropriate control signals for the current instruction.

2. Outputs: Control Signals
   - The control unit generates several critical control signals to manage the processor's operations:
   - RegWrite: Enables writing data to a register, used after operations like arithmetic calculations or loading data from memory (e.g., set to 1 for LW, ADD).
   - MEM_Read: Allows the processor to read data from memory, essential for load instructions (e.g., set to 1 for LW).
   - MEM_Write: Permits writing data to memory, used in store instructions (e.g., set to 1 for SW).
   - ALU_Ctrl: Specifies the operation the ALU should perform (e.g., addition, subtraction, logical AND), tailored to the instruction type.
   - Multiplexer Signals (Mux): Control various multiplexers to select data paths or sources, such as:
   - Mux_Extender: Manages sign or zero extension of immediate values (e.g., for ADDI, SLTI).
   - Mux_S2_Imm16: Chooses between a register value and a 16-bit immediate value (e.g., set to 1 for immediate-based instructions like ADDI).
   - Mux_SA_RS2: Selects between a shift amount (SA) or a register source (RS2) (e.g., set to 1 for SLL, SRL).
   - Mux_RD1 and Mux_RD2: Determine register data inputs for operations requiring multiple registers.

3. Internal Logic

- The control unit employs combinational logic, such as AND gates, to decode the OpCode into specific control signal patterns. For example:
- A unique OpCode bit pattern triggers specific AND gate outputs, activating signals like RegWrite or MEM_Read.
- Multiplexers (Mux) are integrated to route data or control signals dynamically, enhancing flexibility for different instruction types (e.g., R-type, I-type).

Truth Table

	RF	Data memory		prossesor							
	Reg Write	MEM Read	MEM Write	Mux Extender	Mux RD 1(1)	Mux RD 1(0)	Mux RD 2	Mux S2 Imm16(1)	Mux S2 Imm16(0)	Mux SA RS2	ALU crtl
SLL	1	0	0	*	0	0	0	*	*	1	0
SRL	1	0	0	*	0	0	0	*	*	1	0
SRA	1	0	0	*	0	0	0	*	*	1	0
ROR	1	0	0	*	0	0	0	*	*	1	0
ADD	1	0	0	*	0	0	0	0	0	*	0
SUB	1	0	0	*	0	0	0	0	0	*	0
SLT	1	0	0	*	0	0	0	0	0	*	0
SLTU	1	0	0	*	0	0	0	0	0	*	0
SEQ	1	0	0	*	0	0	0	0	0	*	0
XOR	1	0	0	*	0	0	0	0	0	*	0
OR	1	0	0	*	0	0	0	0	0	*	0
AND	1	0	0	*	0	0	0	0	0	*	0
NOR	1	0	0	*	0	0	0	0	0	*	0
MUL	1	0	0	*	0	0	0	0	0	*	0
MULU	1	0	0	*	0	0	0	0	0	*	0
SLLI	1	0	0	*	0	0	0	*	*	0	1
SRLI	1	0	0	*	0	0	0	*	*	0	1
SRAI	1	0	0	*	0	0	0	*	*	0	1
RORI	1	0	0	*	0	0	0	*	*	0	1
ADDI	1	0	0	0	0	0	0	0	1	*	1
SLTI	1	0	0	0	0	0	0	0	1	*	1
SLTIU	1	0	0	1	0	0	0	0	1	*	1
SEQI	1	0	0	0	0	0	0	0	1	*	1
XORI	1	0	0	1	0	0	0	0	1	*	1
ORI	1	0	0	1	0	0	0	0	1	*	1
ANDI	1	0	0	1	0	0	0	0	1	*	1
NORI	1	0	0	1	0	0	0	0	1	*	1
SET	1	0	0	0	1	0	0	*	*	*	1
SSET	1	0	0	*	1	1	0	*	*	*	1
JALR	1	0	0	*	*	*	1	*	*	*	1
LW	1	1	0	0	0	1	0	0	1	*	1
SW	0	0	1	*	*	*	*	1	0	*	1
BEQ	0	0	0	*	*	*	*	0	0	*	1
BNQ	0	0	0	*	*	*	*	0	0	*	1
BLT	0	0	0	*	*	*	*	0	0	*	1
BGE	0	0	0	*	*	*	*	0	0	*	1
BLTU	0	0	0	*	*	*	*	0	0	*	1
BGEU	0	0	0	*	*	*	*	0	0	*	1

Instruction-Specific Behavior

The control unit tailors its outputs based on the instruction being executed. Here are examples from the provided table:

1. Arithmetic Instructions (e.g., ADD, SUB, SLL):
   - ADD: ALU_Ctrl set to perform addition, RegWrite = 1 to store the result, Mux_S2_Imm16 = 0 (uses register value).
   - SLL: ALU_Ctrl for shift left, Mux_SA = 1 (shift amount selected), RegWrite typically enabled.
   - Common traits: No memory access (MEM_Read = 0, MEM_Write = 0).

2. Memory Instructions:
   - LW (Load Word): MEM_Read = 1 to fetch data from memory, RegWrite = 1 to store it in a register, Mux_S2_Imm16 = 1 (immediate offset used).
   - SW (Store Word): MEM_Write = 1 to write to memory, Mux_S2_Imm16 = 1, no register write (RegWrite = 0).

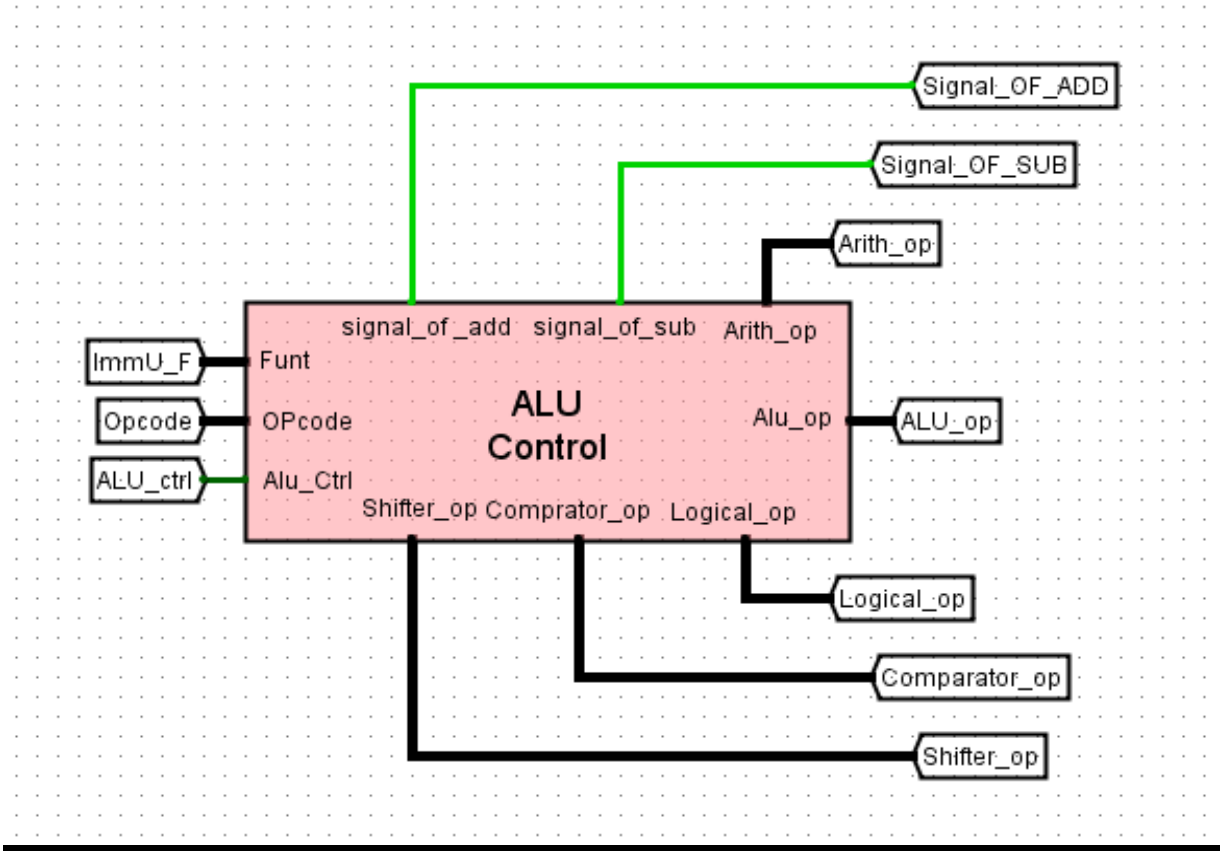
3. Branch Instructions (e.g., BEQ, BNE):
   - ALU_Ctrl performs a comparison (e.g., subtraction to check equality), no register or memory writes (RegWrite = 0, MEM_Read = 0, MEM_Write = 0).
   - The control unit manages branch logic to update the program counter if conditions are met.

4. Immediate Instructions (e.g., ADDI, SLTI):
   - ALU_Ctrl set for the operation, Mux_S2_Imm16 = 1 to use the immediate value, RegWrite = 1 to store the result.

Observations from the Table

- The table splits instructions into two groups based on ALU_Ctrl (0 or 1), possibly indicating different instruction sets or modes:
- ALU_Ctrl = 0: Basic operations (e.g., ADD, SLL) with register-based operands.
- ALU_Ctrl = 1: Immediate-based or memory/branch instructions (e.g., ADDI, LW, BEQ).

- **ALU Control Unit**



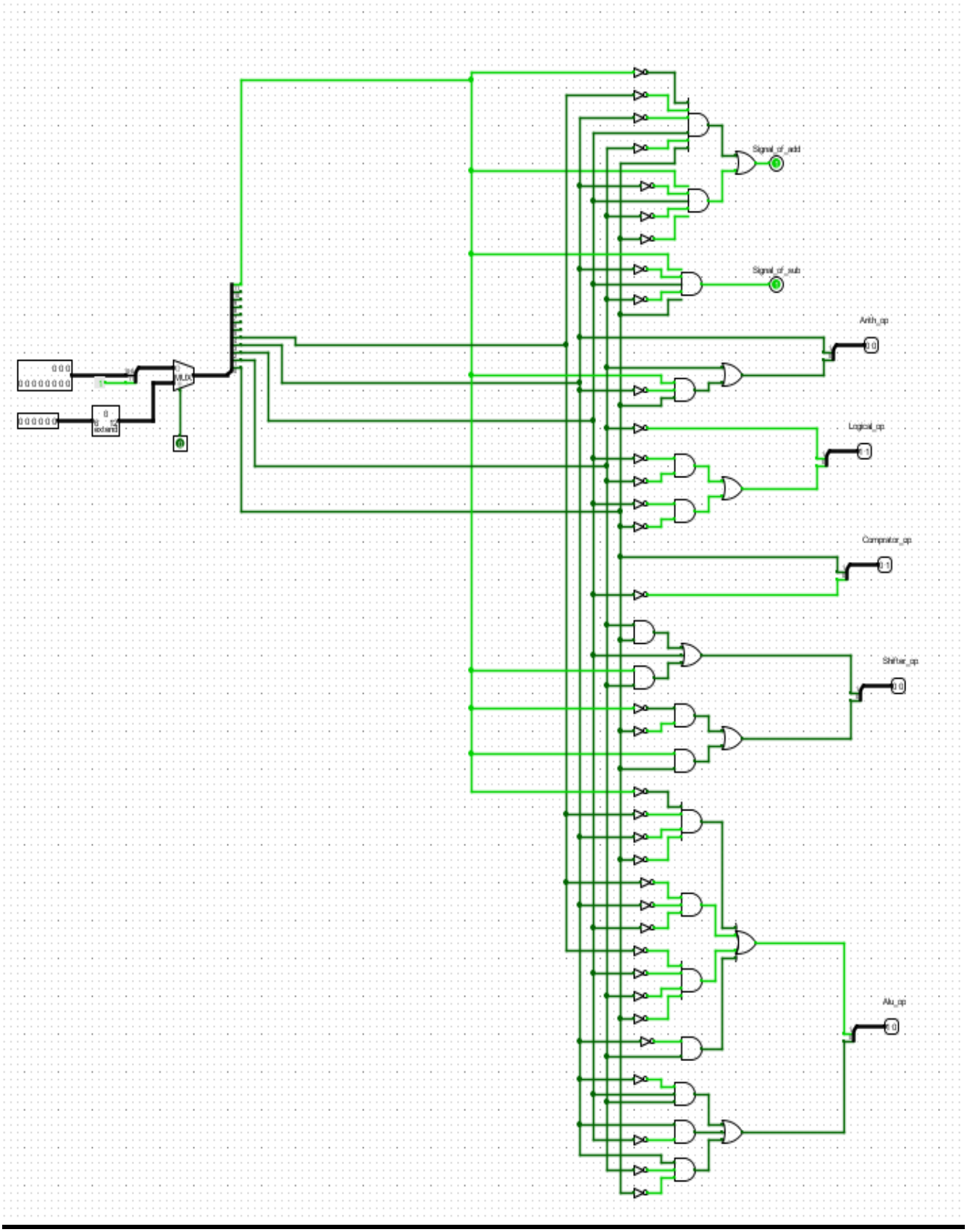
Overview

The ALU control unit serves as the "conductor" of the ALU, determining what operation it should perform based on the instruction being executed. It receives inputs from the instruction—typically the opcode and, in some cases, additional function codes—and generates a set of control signals that configure the ALU accordingly. These operations can include:

- Arithmetic: Addition (ADD), subtraction (SUB), etc.
- Logical: AND, OR, XOR, NOR, etc.
- Shifts: Shift left logical (SLL), shift right logical (SRL), etc.
- Comparisons: Set less than (SLT), set less than unsigned (SLTU), etc.

The control unit ensures that the ALU executes the correct operation by producing signals like "Signal_of_ADD" or "Signal_of_SUB" for arithmetic tasks or broader signals like "ALU_Ctrl" to specify the operation type. This process is essential for enabling the processor to handle a wide variety of instructions efficiently.

Structure


Signal_of_add
Signal_of_sub
Arith_op
Logical_op
Comparator_op
Shifter_op
Alu_op

The ALU control unit is typically implemented as a combinational logic circuit, meaning its outputs depend solely on its current inputs, with no memory or clock dependency. Its structure can be broken down into the following key elements:

1. Inputs:
   - Opcode: Derived from the instruction, this field identifies the type of operation (e.g., arithmetic, logical, memory access).
   - Function Code: For certain instructions (e.g., R-type in MIPS or RISC-V), this provides additional specificity about the operation (e.g., ADD vs. SUB within arithmetic instructions).
   - These inputs are often multi-bit fields, such as a 6-bit opcode or a 6-bit funct field, depending on the architecture.

2. Logic Circuitry:
   - The unit uses a network of logic gates—AND, OR, NOT, etc.—to decode the inputs and produce the appropriate control signals.
   - For example, an AND gate might combine specific opcode bits to activate the "Signal_of_ADD" output, while an OR gate could enable multiple logical operations based on a broader instruction type.

3. Outputs:
   - ALU Operation Code: A multi-bit signal (e.g., 2-bit or 3-bit, such as "ALU operation [1:0]") that specifies the exact operation for the ALU (e.g., 00 for ADD, 01 for SUB, 10 for AND).
   - Specific Control Signals: Outputs like "Signal_of_ADD," "Signal_of_SUB," "Logical_op," "Shifter_op," and "Comparator_op" directly instruct the ALU's functional units.
   - Multiplexer Control Signals: Signals to control multiplexers in the datapath (e.g., "Mux_S2_Imm16," "Mux_RD1") that select operands or data sources.

The structure is designed to map each instruction type to a unique combination of control signals, ensuring that the ALU performs the intended operation. For instance, an ADD instruction might set "Signal_of_ADD" to 1 and "Signal_of_SUB" to 0, while an AND instruction activates "Logical_op" with a

specific configuration.

Truth Table

	Arith				Logical		comprator		Shifter		ALU_operation	
	signal of add	signal of sub	signal select Arith_op(1)	signal select Arith_op(0)	signal select logical_op(1)	signal select logical_op(0)	Signal select SLT,SEQ,SLTU(1)	Signal select SLT,SEQ,SLTU(0)	Signal select shifter_op(1)	Signal select shifter_op(0)	ALU_operation (1)	ALU_operation (0)
SLL	0	0	*	*	*	*	*	*	0	0	1	0
SRL	0	0	*	*	*	*	*	*	0	1	1	0
SRA	0	0	*	*	*	*	*	*	1	0	1	0
ROR	0	0	*	*	*	*	*	*	1	1	1	0
ADD	1	0	0	0	*	*	*	*	*	*	0	0
SUB	0	1	0	1	*	*	*	*	*	*	0	0
SLT	0	0	*	*	*	*	0	0	*	*	1	1
SLTU	0	0	*	*	*	*	1	0	*	*	1	1
SEQ	0	0	*	*	*	*	0	1	*	*	1	1
XOR	0	0	*	*	1	1	*	*	*	*	0	1
OR	0	0	*	*	0	1	*	*	*	*	0	1
AND	0	0	*	*	0	0	*	*	*	*	0	1
NOR	0	0	*	*	1	0	*	*	*	*	0	1
MUL	0	0	1	0	*	*	*	*	*	*	0	0
MULU	0	0	1	1	*	*	*	*	*	*	0	0
SLLI	0	0	*	*	*	*	*	*	0	0	1	0
SRLI	0	0	*	*	*	*	*	*	0	1	1	0
SRAI	0	0	*	*	*	*	*	*	1	0	1	0
RORI	0	0	*	*	*	*	*	*	1	1	1	0
ADDI	1	0	0	0	*	*	*	*	*	*	0	0
SLTI	0	0	*	*	*	*	0	0	*	*	1	1
SLTIU	0	0	*	*	*	*	1	0	*	*	1	1
SEQI	0	0	*	*	*	*	0	1	*	*	1	1
XORI	0	0	*	*	1	1	*	*	*	*	0	1
ORI	0	0	*	*	0	1	*	*	*	*	0	1
ANDI	0	0	*	*	0	0	*	*	*	*	0	1
NORI	0	0	*	*	1	0	*	*	*	*	0	1
SET	0	0	*	*	*	*	*	*	*	*	*	*
SSET	0	0	*	*	*	*	*	*	*	*	*	*
JALR	0	0	*	*	*	*	*	*	*	*	*	*
LW	0	0	0	0	*	*	*	*	*	*	0	0
SW	0	0	0	0	*	*	*	*	*	*	0	0
BEQ	0	0	*	*	*	*	*	*	*	*	*	*
BNQ	0	0	*	*	*	*	*	*	*	*	*	*
BLT	0	0	*	*	*	*	*	*	*	*	*	*
BGE	0	0	*	*	*	*	*	*	*	*	*	*
BLTU	0	0	*	*	*	*	*	*	*	*	*	*
BGEU	0	0	*	*	*	*	*	*	*	*	*	*

Operation-Specific Behavior

The ALU control unit's behavior varies depending on the instruction type:

1. Arithmetic Instructions (e.g., ADD, SUB, ADDI):
   - Sets the ALU to perform addition or subtraction.

- Example: For ADD, "Signal_of_ADD = 1," "Signal_of_SUB = 0," and "ALU operation = 00."

2. Logical Instructions (e.g., AND, OR, XOR):

- Configures the ALU for bitwise operations.
- Example: For AND, "signal select logical op(1:0) = 00," directing the ALU to perform a bitwise AND.

3. Shift Instructions (e.g., SLL, SRL, SRA):

- Activates the shifter logic with signals like "signal select shifter op(1:0)."
- Example: For SLL, "signal select shifter op(1:0) = 01."

4. Comparison Instructions (e.g., SLT, SLTI):

- Enables the comparator logic to evaluate conditions.
- Example: For SLT, "signal select SLT_EQ(1:0) = 10."

5. Memory Access Instructions (e.g., LW, SW):

- Uses the ALU for address calculation (base + offset).
- Example: For LW, "ALU_Ctrl = 1" (addition), with "MEM_Read = 1" and "RegWrite = 1."

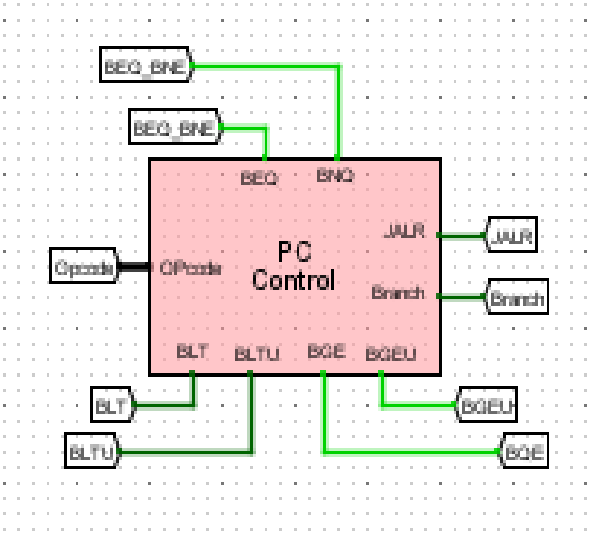
6. Branch Instructions (e.g., BEQ, BNE):

- The ALU compares register values, with the control unit setting it for subtraction to check equality or other conditions.

• PC Control Unit

Overview

The PC Control Unit is a specialized component within the processor's control unit, dedicated to managing the Program Counter (PC). Its primary responsibility is to determine the next PC value, ensuring the correct sequence of instruction execution. This involves handling different execution flows:

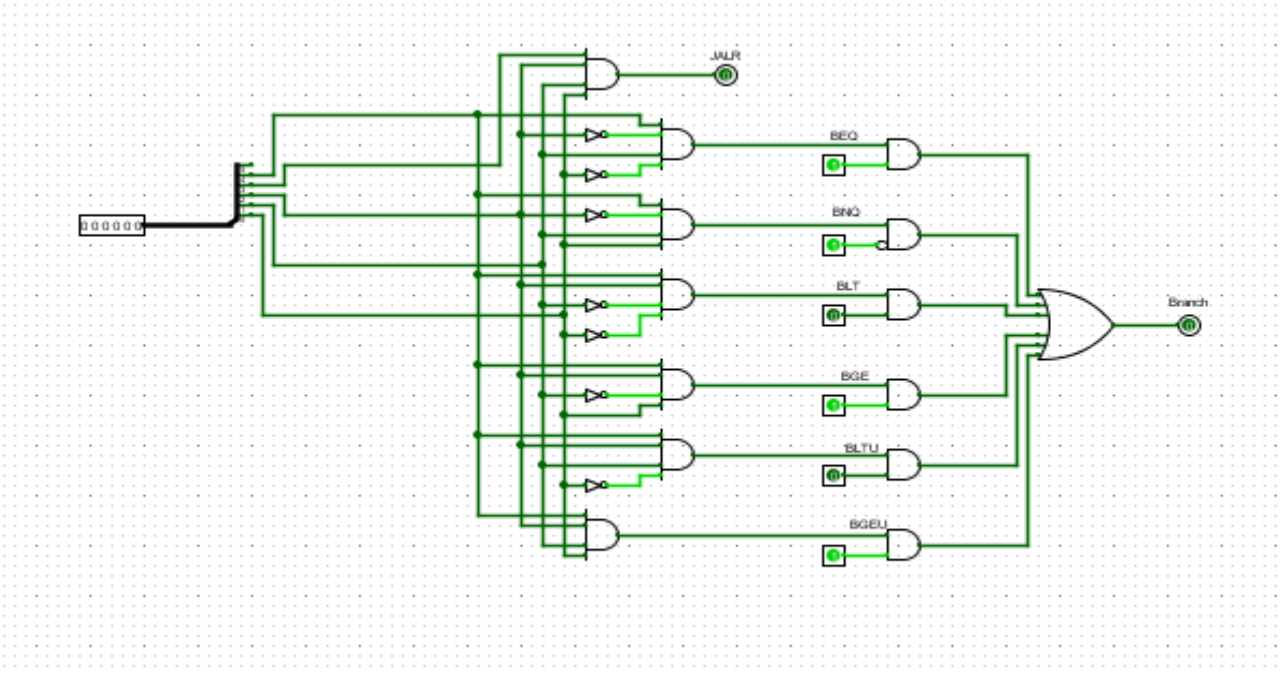


- Sequential Execution: The PC is incremented by 4 (assuming 32-bit instructions) to fetch the next instruction in sequence.
- Conditional Branches: The PC is updated to a target address if a condition (e.g., equality or comparison) is met, such as in instructions like BEQ (Branch if Equal) or BLT (Branch if Less Than).
- Unconditional Jumps: The PC is set to a specific target address, as in JALR (Jump and Link Register), which also saves a return address for subroutine calls.
- Exceptions: Though less detailed here, the PC may be redirected to an exception handler address in some cases.

The PC Control Unit interprets the instruction opcode and evaluates

conditions to generate control signals that dictate these updates, making it a critical driver of program flow.

Structure



The PC Control Unit comprises combinational logic and selection mechanisms that process inputs and produce outputs to update the PC. Its structure can be broken down into the following key elements:

1. Instruction Decoding Logic:
   - Inputs: Receives the opcode and possibly function codes from the instruction fetched via the PC (e.g., from the Instruction Register, IR[31:0]).
   - Function: Decodes the instruction to identify its type—sequential, branch, or jump. For example, instructions like BEQ or JALR trigger specific PC update paths.
   - Evidence: Tables and diagrams indicate mappings of instructions (e.g., ADD, BEQ) to control signals, showing how opcodes are interpreted.

2. Condition Evaluation Logic:
   - Inputs: For conditional branches, it uses results from the ALU or status flags (e.g., zero flag for equality, sign flag for comparisons).

- Function: Evaluates branch conditions such as equality (BEQ), inequality (BNE), or comparisons (BLT, BGE). For instance, BEQ checks if two registers are equal, often via an ALU subtraction yielding zero.
- Evidence: Outputs like BEQ, BNE, BLT, and BGEU are generated based on instruction bits and ALU results, as seen in branch condition logic diagrams.

3. Multiplexer Control:
   - Components: Controls a multiplexer that selects the next PC value from multiple sources:
     - PC + 4: Default for sequential execution.
     - Branch Target Address: Calculated as PC + offset for taken branches.
     - Jump Target Address: A register value plus offset (e.g., for JALR) or a direct address.
   - Function: Generates select signals for the multiplexer based on the instruction type and condition outcomes.
   - Evidence: Multiplexers (e.g., Mux Extender, Mux RD) are shown selecting data paths, with control signals like "Branch" and "JALR" directing the choice.

4. Control Signal Outputs:
   - Outputs: Signals such as "Branch" (for conditional branches) and "JALR" (for jumps) dictate the PC update mechanism.
   - Function: These signals interface with the datapath to enact the decided PC value.
   - Evidence: Diagrams show outputs like "Branch" and "JALR" emanating from the PC Control Unit, influencing downstream components.

Truth Table

Example: Processing a Branch Instruction (BEQ):

To illustrate, consider the execution of a BEQ (Branch if Equal) instruction:

1. Instruction Fetch: The PC addresses the BEQ instruction in memory, loading it into the IR.

2. Decoding: The PC Control Unit identifies the BEQ opcode, signaling a conditional branch.

3. Condition Evaluation: The ALU subtracts two register values; if the result is zero (equal), the zero flag is set.

4. Control Signal: If the condition is true, the PC Control Unit sets the "Branch" signal to 1 and calculates the branch target (PC + offset).

5. Multiplexer Selection: The multiplexer selects the branch target; otherwise, it defaults to PC + 4.

6. PC Update: The chosen value updates the PC, fetching either the branch target or the next sequential instruction.

This process exemplifies the PC Control Unit's role in managing control flow within the datapath.

	PC_counter		Branch					
	JALR	Branch	BLT	BLTU	BNE	BEQ	BGTE	BGTEU
SLL	0	0	0	0	0	0	0	0
SRL	0	0	0	0	0	0	0	0
SRA	0	0	0	0	0	0	0	0
ROR	0	0	0	0	0	0	0	0
ADD	0	0	0	0	0	0	0	0
SUB	0	0	0	0	0	0	0	0
SLT	0	0	0	0	0	0	0	0
SLTU	0	0	0	0	0	0	0	0
SEQ	0	0	0	0	0	0	0	0
XOR	0	0	0	0	0	0	0	0
OR	0	0	0	0	0	0	0	0
AND	0	0	0	0	0	0	0	0
NOR	0	0	0	0	0	0	0	0
MUL	0	0	0	0	0	0	0	0
MULU	0	0	0	0	0	0	0	0
SLLI	0	0	0	0	0	0	0	0
SRLI	0	0	0	0	0	0	0	0
SRAI	0	0	0	0	0	0	0	0
RORI	0	0	0	0	0	0	0	0
ADDI	0	0	0	0	0	0	0	0
SLTI	0	0	0	0	0	0	0	0
SLTIU	0	0	0	0	0	0	0	0
SEQI	0	0	0	0	0	0	0	0
XORI	0	0	0	0	0	0	0	0
ORI	0	0	0	0	0	0	0	0
ANDI	0	0	0	0	0	0	0	0
NORI	0	0	0	0	0	0	0	0
SET	0	0	0	0	0	0	0	0
SSET	0	0	0	0	0	0	0	0
JALR	1	0	0	0	0	0	0	0
LW	0	0	0	0	0	0	0	0
SW	0	0	0	0	0	0	0	0
BEQ	0	1	0	0	0	1	0	0
BNQ	0	1	0	0	1	0	0	0
BLT	0	1	1	0	0	0	0	0
BGE	0	1	0	0	0	0	1	0
BLTU	0	1	0	1	0	0	0	0
BGEU	0	1	0	0	0	0	0	1

Simulation & Testing

- **Instructor Test Code.**

NO	Instruction	Hexa	Expected Value
1	SET R1, 0x0001	0001004D	R1 = 0x00000001
2	SW R1, 0(R0)	00010011	Mem[0] = 0x00000001
3	SW R1, 1(R0)	00010051	Mem[1] = 0x00000001
4	SET R2, 0x000A	000A008D	R2 = 0x0000000A
5	SW R2, 2(R0)	00020091	Mem[2] = 0x0000000A
6	SET R3, 0x1289	128900CD	R3 = 0x00001289
7	SSET R3, 0x45AC	45AC00CE	R3 = 0x128945AC
8	SW R3, 60(R0)	00230711	Mem[60] = 0x128945AC
9	SET R4, 0x0500	0500010D	R4 = 0x00000500
10	SSET R4, 0x7342	7342010E	R4 = 0x05007342
11	SW R4, 61(R0)	00240751	Mem[61] = 0x05007342
12	SET R1, 0x0384	0384004D	R1 = 0x00000384
13	SET R8, 0x1234	1234020D	R8 = 0x00001234
14	SSET R8, 0x5678	5678020E	R8 = 0x12345678
15	ADDI R5, R1, 20	00140945	R5 = R1 + 20 = 0x398
16	XOR R3, R1, R5	012508C0	R3 = R1 ^ R5 = 0x1c

17	ADD R4, R8, R3	00834100	R4 = R8 + R3 = 0x12345694
18	LW R1, 0(R0)	00000050	R1 = Mem[0] = 0x00000001
19	LW R2, 1(R0)	00010090	R2 = Mem[1] = 0x00000001
20	LW R3, 2(R0)	000200D0	R3 = Mem[2] = 0x0000000A
21	SUB R4, R4, R4	00A42100	R4 = 0
22	Loop1:ADD R4, R2, R4	00841100	R4 += R2
23	SLT R6, R2, R3	00C31180	R6 = (R2 < R3) ? 1 : 0
24	BEQ R6, R0, done	000030D2	if R6 == 0 → done
25	ADD R2, R1, R2	00820880	R2 += R1
26	BEQ R0, R0, Loop1	FFE00712	unconditional jump to Loop1
27	done: SW R4, 0(R0)	00040011	Mem[0] = R4 = 0x37
28	MUL R10, R2, R3	01A31280	R10 = R2 * R3=0x64
29	SRL R14, R10, R4	00245380	R14 = R10 >> R4 (logical) = 0
30	SRA R15, R10, R4	004453C0	R15 = R10 >> R4 (arith) = 0
31	RORI R26, R14, 5	00057684	R26 = ROR(R14, 5) = 0
32	JALR R7, R0, func	002501CF	Jump to func, save PC+1 in R7
33	SET R9, 0x4545	4545024D	R9 = 0x00004545
34	SET R10, 0x4545	4545028D	R10 = 0x00004545
35	BGE R10, R9, L1	00095095	Branch if R10 >= R9 (taken)
36	ANDI R23, R1, 0xFFFF	FFFF0DCB	R23 = R1 & 0x0000FFFF (skiped)

37	L1: BEQ R0, R0, L1	00000012	Infinite loop (halt)
38	func: OR R5, R2, R3	01431140	R5 = R2 | R3 = 0xa
39	LW R1, 0(R0)	00000050	R1 = Mem[0] = 0x37
40	LW R2, 5(R1)	00050890	R2 = 0x128945AC
41	LW R3, 6(R1)	000608D0	R3 = Mem[R1 + 6] = 0x05007342
42	AND R4, R2, R3	01631100	R4 = R2 & R3 = 0x4100
43	SW R4, 0(R0)	00040011	Mem[0] = R4
44	JALR R0, R7, 0	0000380F	Return to caller (JR R7)

We decode each program according to the format of each instruction type (R- Type, I-Type and SB-Type), after that we decode it to binary number

Instructor Code Description

SET R1, 0x0001

Sets register R1 to the value 0x00000001 (1 in decimal). The SET instruction assigns a 16-bit immediate value, zero-extended to 32 bits, to the specified register.

SW R1, 0(R0)

Stores the value in R1 (0x00000001) into memory at address 0 + R0. Since R0 is 0, this writes 0x00000001 to memory address 0.

SW R1, 1(R0)

Stores the value in R1 (0x00000001) into memory at address 1 + R0 (memory address 1).

SET R2, 0x000A

Sets R2 to 0x0000000A (10 in decimal).

SW R2, 2(R0)

Stores the value in R2 (0x0000000A) into memory at address 2 + R0 (memory address 2).

SET R3, 0x1289

Sets R3 to 0x00001289 (4745 in decimal).

SSET R3, 0x45AC

Modifies R3 by shifting its current lower 16 bits (0x1289) to the upper 16 bits and setting the lower 16 bits to 0x45AC. Result: R3 = 0x128945AC.

SW R3, 60(R0)

Stores the value in R3 (0x128945AC) into memory at address 60 + R0 (memory address 60).

SET R4, 0x0500

Sets R4 to 0x00000500 (1280 in decimal).

SSET R4, 0x7342

Shifts R4's lower 16 bits (0x0500) to the upper 16 bits and sets the lower 16 bits to 0x7342. Result: R4 = 0x05007342.

SW R4, 61(R0)

Stores the value in R4 (0x05007342) into memory at address 61 + R0 (memory address 61).

SET R1, 0x0384

Sets R1 to 0x00000384 (900 in decimal).

SET R8, 0x1234

Sets R8 to 0x00001234 (4660 in decimal).

SSET R8, 0x5678

Shifts R8's lower 16 bits (0x1234) to the upper 16 bits and sets the lower 16 bits to 0x5678. Result: R8 = 0x12345678.

ADDI R5, R1, 20

Adds the immediate value 20 (0x14) to R1 (0x00000384). Result: R5 = 0x00000384 + 0x14 = 0x00000398 (920 in decimal).

XOR R3, R1, R5

Performs a bitwise XOR between R1 (0x00000384) and R5 (0x00000398). Result: R3 = 0x0000001C (28 in decimal).

ADD R4, R8, R3

Adds R8 (0x12345678) and R3 (0x0000001C). Result: R4 = 0x12345694.

LW R1, 0(R0)

Loads the value from memory address 0 + R0 (memory[0], which is 0x00000001) into R1. R1 = 0x00000001.

LW R2, 1(R0)

Loads the value from memory address 1 + R0 (memory[1], which is

0x00000001) into R2. R2 = 0x00000001.

LW R3, 2(R0)

Loads the value from memory address 2 + R0 (memory[2], which is 0x0000000A) into R3. R3 = 0x0000000A.

SUB R4, R4, R4

Subtracts R4 from itself. Result: R4 = 0x00000000 (0 in decimal).

Loop1: ADD R4, R2, R4

Adds R2 to R4 and stores the result in R4. This is the start of a loop (see below). Initially, R4 = 0x00000001 (R2 = 1, R4 = 0).

SLT R6, R2, R3

Sets R6 to 1 if R2 < R3, otherwise 0. Initially, R2 (0x00000001) < R3 (0x0000000A), so R6 = 1.

BEQ R6, R0, done

Branches to label "done" if R6 equals R0 (0). If R6 = 1, no branch occurs, and execution continues.

ADD R2, R1, R2

Adds R1 (0x00000001) to R2 and stores the result in R2. Initially, R2 becomes 0x00000002 (2 in decimal).

BEQ R0, R0, Loop1

Unconditional branch back to "Loop1" (instruction 22) since R0 always equals R0.

done: SW R4, 0(R0)

Stores R4 (0x00000037) into memory at address 0 + R0 (memory[0]).

MUL R10, R2, R3

Multiplies R2 (0x0000000B) and R3 (0x0000000A). Result: R10 = 0x0000006E (110 in decimal). Note: Thinking trace assumes R2 = 0x0A, but after the loop, R2 = 0x0B.

SRL R14, R10, R4

Shifts R10 (0x0000006E) right logically by R4 (0x00000037) bits. Since 0x37 = 55 > 32, the result is 0x00000000 in R14.

SRA R15, R10, R4

Shifts R10 (0x0000006E) right arithmetically by R4 (0x00000037) bits. Since the shift exceeds 32 bits, the result is 0x00000000 in R15 (positive number, no sign extension effect).

RORI R26, R14, 5

Rotates R14 (0x00000000) right by 5 bits. Result: R26 = 0x00000000 (no change, as it's all zeros).

JALR R7, R0, func

Jumps to label "func" (instruction 33) and stores the address of the next instruction (33 + 1 = 34 in sequence) in R7. R0 = 0 specifies no offset.

func: OR R5, R2, R3

Performs a bitwise OR between R2 (0x0000000B) and R3 (0x0000000A). Result: R5 = 0x0000000B.

LW R1, 0(R0)

Loads the value from memory[0] (0x00000037) into R1. R1 = 0x00000037 (55 in decimal).

LW R2, 5(R1)

Loads the value from memory[R1 + 5] = memory[0x37 + 5] = memory[60] (0x128945AC) into R2. R2 = 0x128945AC.

LW R3, 6(R1)

Loads the value from memory[R1 + 6] = memory[0x37 + 6] = memory[61] (0x05007342) into R3. R3 = 0x05007342.

AND R4, R2, R3

Performs a bitwise AND between R2 (0x128945AC) and R3 (0x05007342). Result: R4 = 0x00004100.

SW R4, 0(R0)

Stores R4 (0x00004100) into memory at address 0 + R0 (memory[0]).

JALR R0, R7, 0

Jumps to the address stored in R7 (instruction 40), effectively returning from the function. R0 as the link register means no new return address is saved.

SET R9, 0x4545

Sets R9 to 0x00004545 (17733 in decimal).

SET R10, 0x4545

Sets R10 to 0x00004545 (17733 in decimal).

BGE R10, R9, L1

Branches to label "L1" (instruction 44) if R10 (0x00004545) >= R9 (0x00004545). Since they are equal, the branch is taken, skipping instruction 43.

ANDI R23, R1, 0xFFFF

(Skipped due to branch) Would perform a bitwise AND between R1 (0x00000037) and 0x0000FFFF, storing the result in R23.

L1: BEQ R0, R0, L1

Unconditional branch to itself (R0 always equals R0), creating an infinite loop. Execution halts here.

Execution Flow Summary:

1. **Initialization (1–20)**: Sets up registers and memory with initial values.
2. **Loop (22–26)**: Sums numbers 1 to 10 into R4 (result: 0x00000037), stored in memory[0] at "done".
3. **Post-Loop Operations (28–31)**: Performs arithmetic and shift operations, mostly resulting in zeros due to large shift amounts.
4. **Function Call (32–39)**: Jumps to "func", performs logical

operations, and updates memory[0] to 0x00004100.

5. **Final Branch (42–44)**: Compares equal values, branches to L1, and enters an infinite loop.

Team Work

Abdelrahman Abdelnasser Abdelrahman:

Design program counter, Data memory, Instruction memory

Test the processor

Design datapath with Yusuf and Raghad

Write documentation with Yusuf

Raghad Soliman Mohamed:

Design Arthimetic unit, Logical unit, MULU

Combine ALU

Write presentation

Design MUL, Main control unit, PC control unit, ALU control unit with Yusuf

Design datapath with Yusuf and Abdelrahman

Yusuf Mohamed Saleh Boriek:

Design Register file, Comparator unit, Shifter unit, Overflow

Design datapath with Raghad and Abdelrahman

Design MUL, Main control unit, PC control unit, ALU control unit with Raghad

Write Documentation with Abdelrahman

-We have about 8 meetings online and 1 meeting offline.

-Total time of working 36 Hours.