RISC Processor

GITHUB

Computer Architecture

Supervised by: Dr. Gihan Naguib

Presented for: Eng. Jihad Awad

By:

Amr Alaa

Zyad Mohammad

Ziad Nasser

Table of Contents:

1. Introduction

For this project, the objective is to create a basic 16-bit RISC processor featuring seven general-purpose registers labeled R1 to R7. R0 is set to zero and cannot be modified, so we have seven usable registers. Additionally, there is a single 16-bit special-purpose register, the program counter (PC). The PC register enables access to a total of 2^{16} instructions. All instructions are limited to 16 bits in length. The processor supports three instruction formats: R-type, I-type, and J-type, each with its own structure and purpose.

R-type format:

The R-type format consists of a 5-bit opcode (Op), a 3-bit destination register (Rd), two 3-bit source registers (Rs and Rt), and a 2-bit function field (F).

$\mathbf{op^5}$	$\mathbf{F^2}$	$\mathbf{rd^3}$	$\mathbf{rs^3}$	$\mathbf{rt^3}$

I-type format:

The I-type format includes a 5-bit opcode (Op), a 3-bit destination register (Rd), a 3-bit source register (Rs), and a 5-bit immediate value.

$\mathbf{op^5}$	$\mathbf{imm^5}$	$\mathbf{rs^3}$	$\mathbf{rt^3}$

J-type format:

The J-type format is comprised of a 5-bit opcode (Op) and an 11-bit immediate value.

$\mathbf{op^5}$	$\mathbf{imm^{11}}$

After analyzing the required instructions, we identified the necessary components for the processor's data path. We then designed each of these components and successfully implemented all the targeted instructions.

2. Phase one

The heart of any computer system is the processor, responsible for executing instructions and manipulating data. Reduced Instruction Set Computing (RISC) processors achieve high performance and efficiency by focusing on simpler instructions that can be executed in a single clock cycle. Building a RISC processor involves creating individual components that work together to achieve this goal.

This guide will introduce you to the essential components of a RISC processor and the concept of the Datapath, which acts as the core pathway for data flow within the processor. We'll delve into the building blocks of the Datapath, including elements like registers, the Arithmetic Logic Unit (ALU), and the control unit. By understanding these components and their interactions, you'll gain a foundational understanding of how a RISC processor operates and executes instructions.

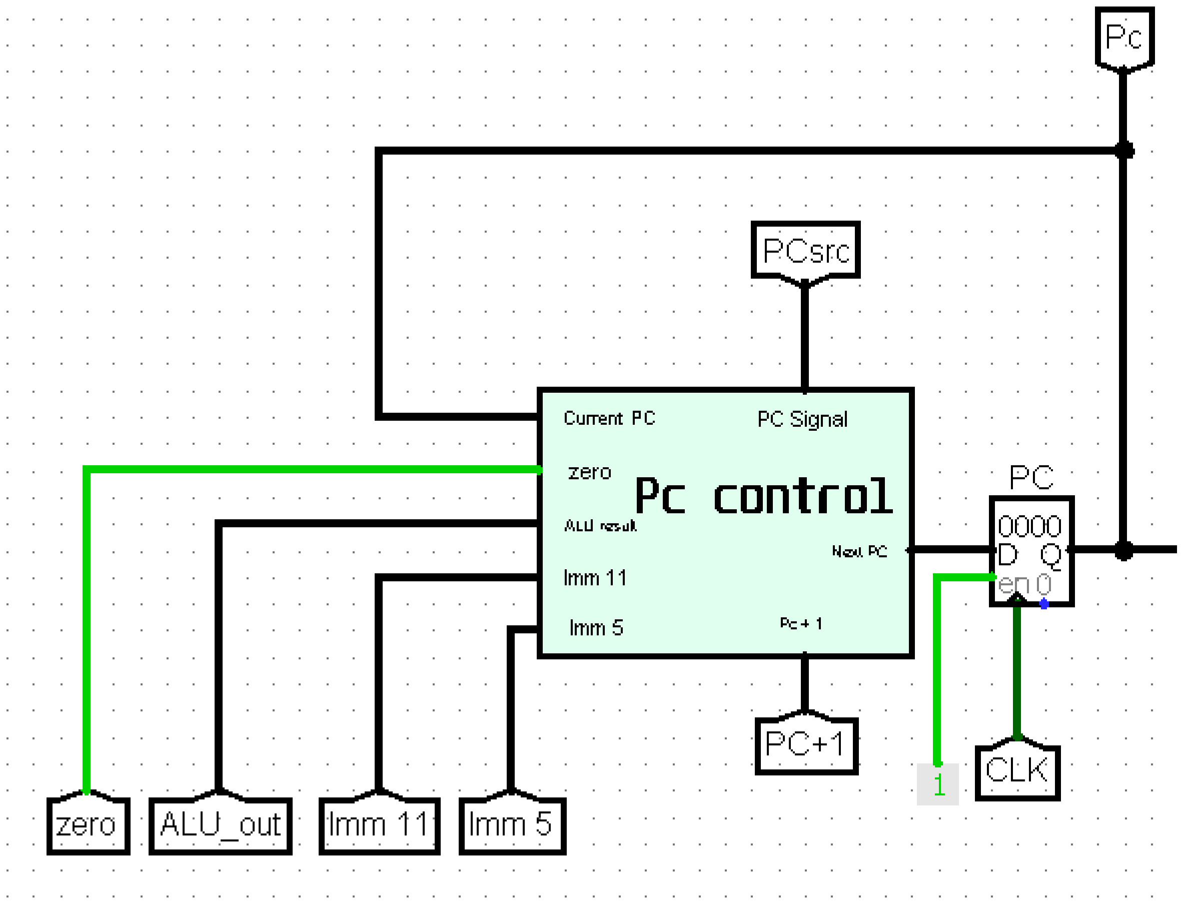
This journey will equip you with the knowledge to explore more advanced topics like instruction pipelining, data hazards, and control unit design, ultimately leading to a comprehensive understanding of RISC processor architecture.

The main components:

1. PC Control
2. Instruction Memory
3. Register File
4. ALU
5. Extender
6. Data Memory
7. Control Unit

2.1. PC Control

The Program Counter Control is the component which is responsible for generating the next PC value.



The PC Control Signal is called PC Src and it is used to decide how the next PC value will be calculated.

2.1.1. The Next PC Values

1- PC + 1:

- This is the main value of the next PC.
- Used for most of the instruction.
- Calculated by adding 1 to the current PC address.
- Its PC Src signal is (xx00).

2- Reg (Rs):

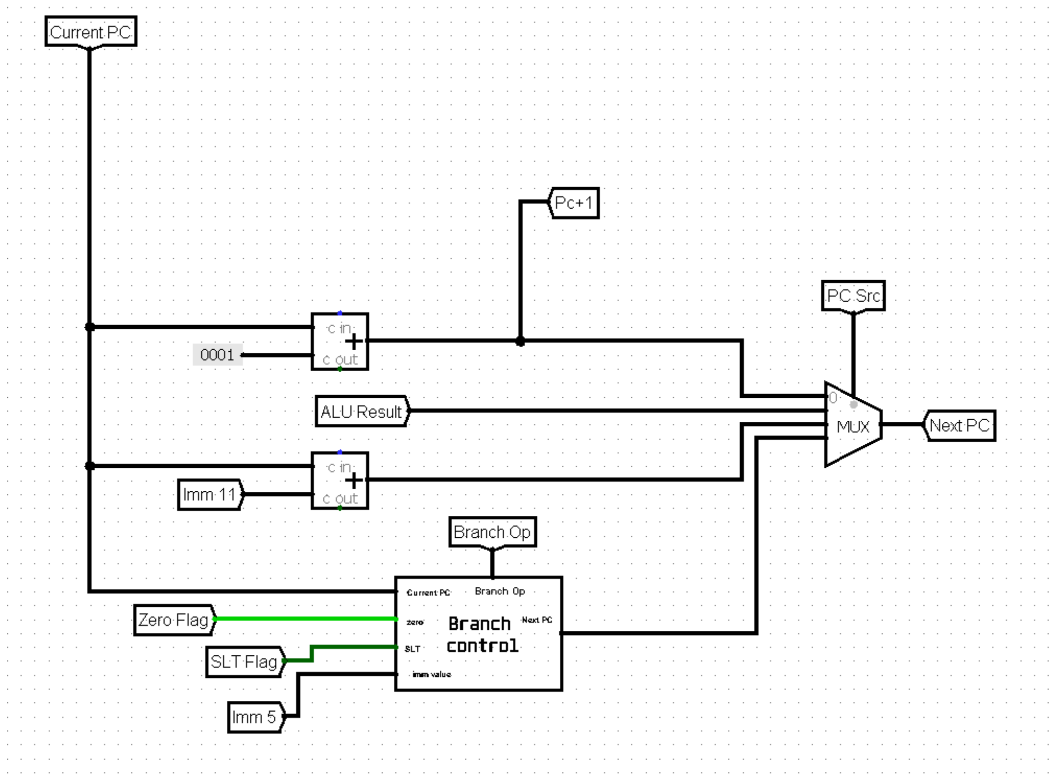
- Used for the Jump Register (JR) instruction.
- In JR instruction Rs is added to 0 and the Pc control takes the result as ALU result and stores that value as the new PC.

3- PC + sign-extend (Imm11):

- Used for the Jump/Jump and Link (J/JAL) instructions.
- The new pc value is the sign extended value of Imm(11) added to the current PC address.

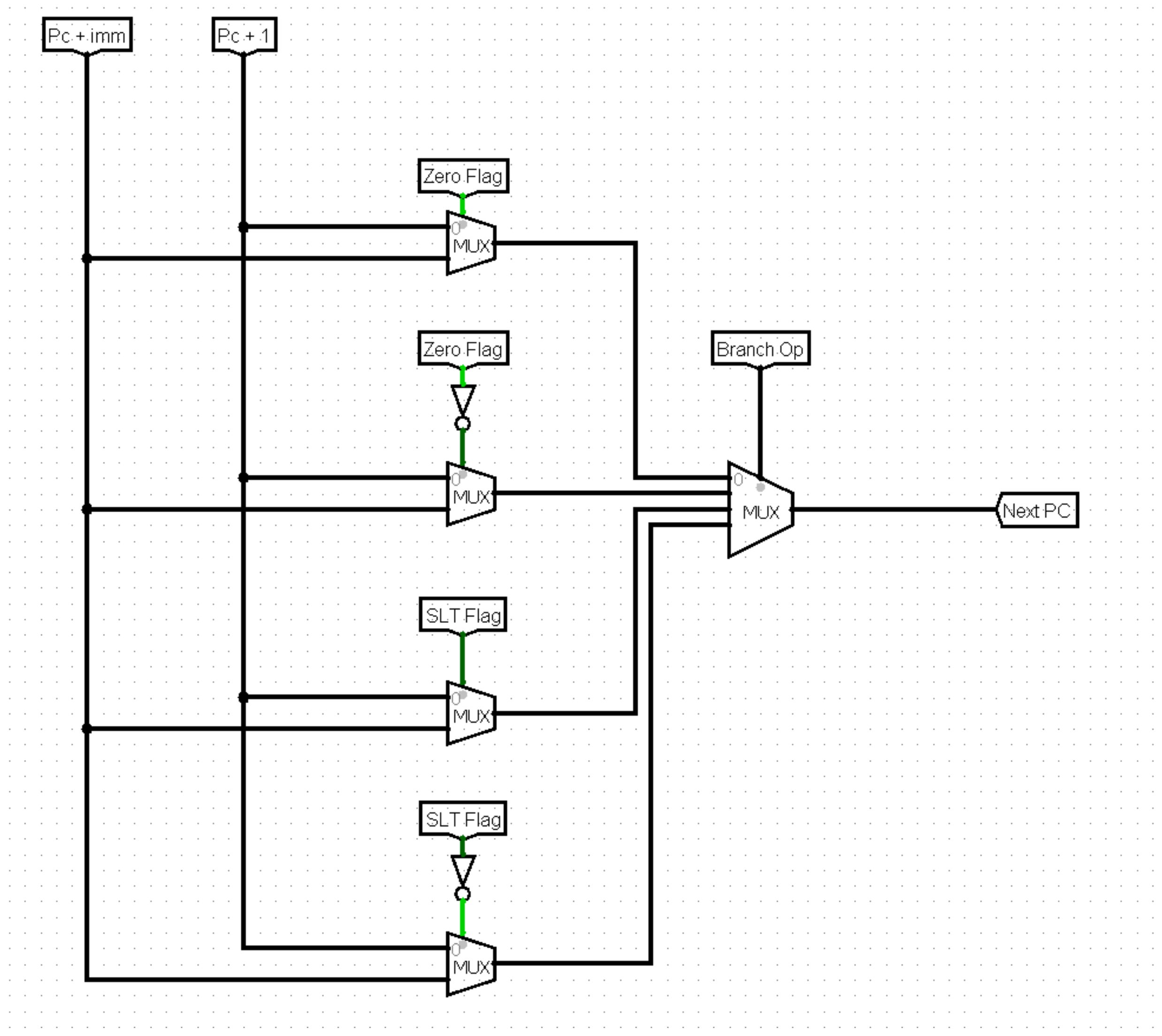
4- PC + sign-extend (Imm5):

- Used for the branch (BEQ/BNE/BLT/BGE) instructions.
- The new pc value is the sign extended value of Imm(5) added to the current PC address.



2.1.2. Branch Control

In the case of a branch instruction the next PC value is either PC + 1 or PC + sign-extend (Imm5). The Branch Control unit uses the zero flag to check for equality and inequality and the SLT flag to check for less than and greater or equal conditions.

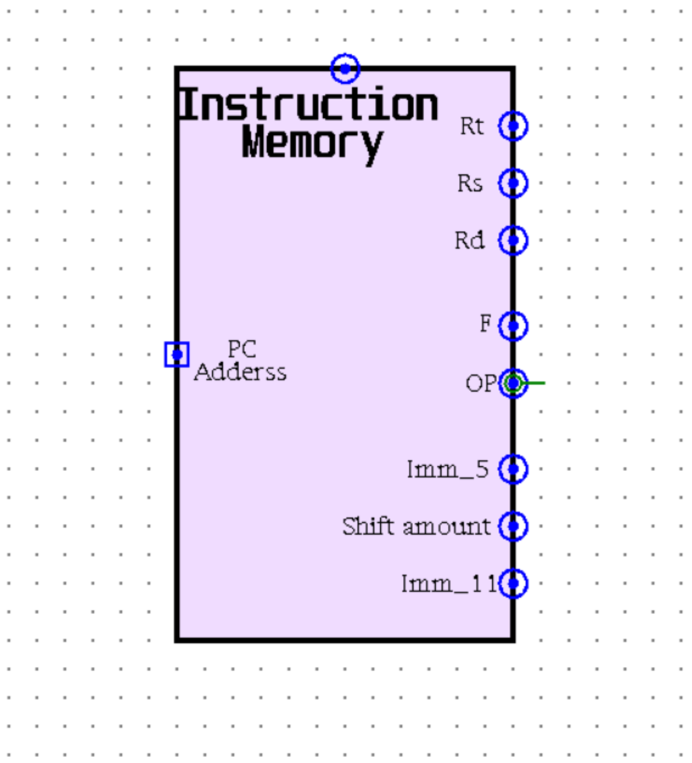


2.2. Instruction Memory

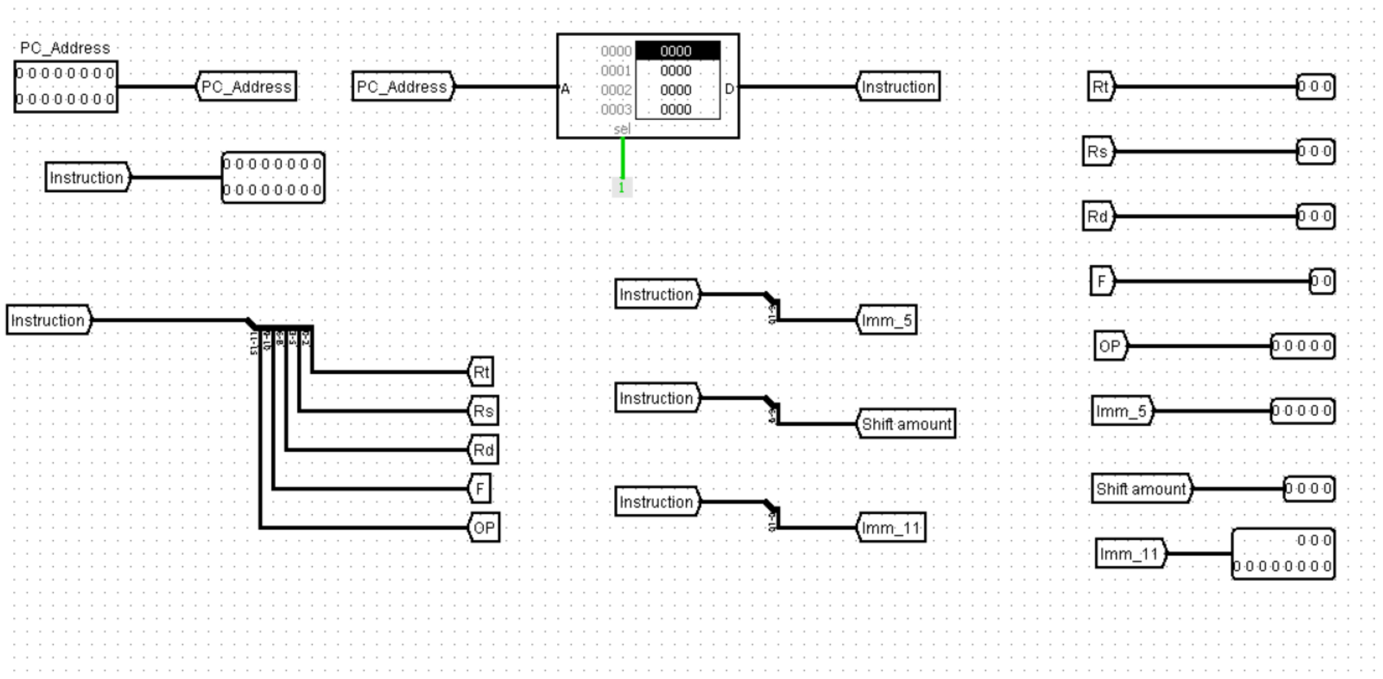
The instruction memory serves as a fundamental component, serving as the starting point for all operations. Operating as a read-only memory (ROM), it is pivotal in program execution. With only one input, it relies on the Program Counter (PC) address to determine the output instruction.

From the 16-bit instruction, we derive eight crucial outputs: bits 0-2 designate Rt, bits 3-5 correspond to Rs, bits 6-8 represent Rd, bits 9-10 signify Function, bits 11-15 denote Opcode, bits 6-10 encode a 5-bit immediate value, bits 0-10 encapsulate an 11-bit immediate value, and bits 6-9 indicate the shift amount.

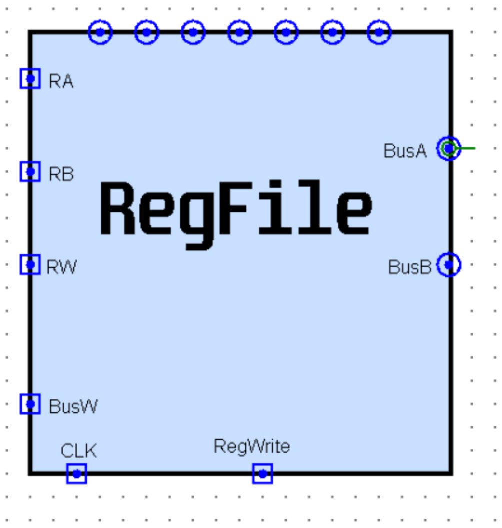
These eight outputs assume a critical role in program execution, as subsequent operations hinge upon their values, guiding the flow of the program.



Internal Architecture



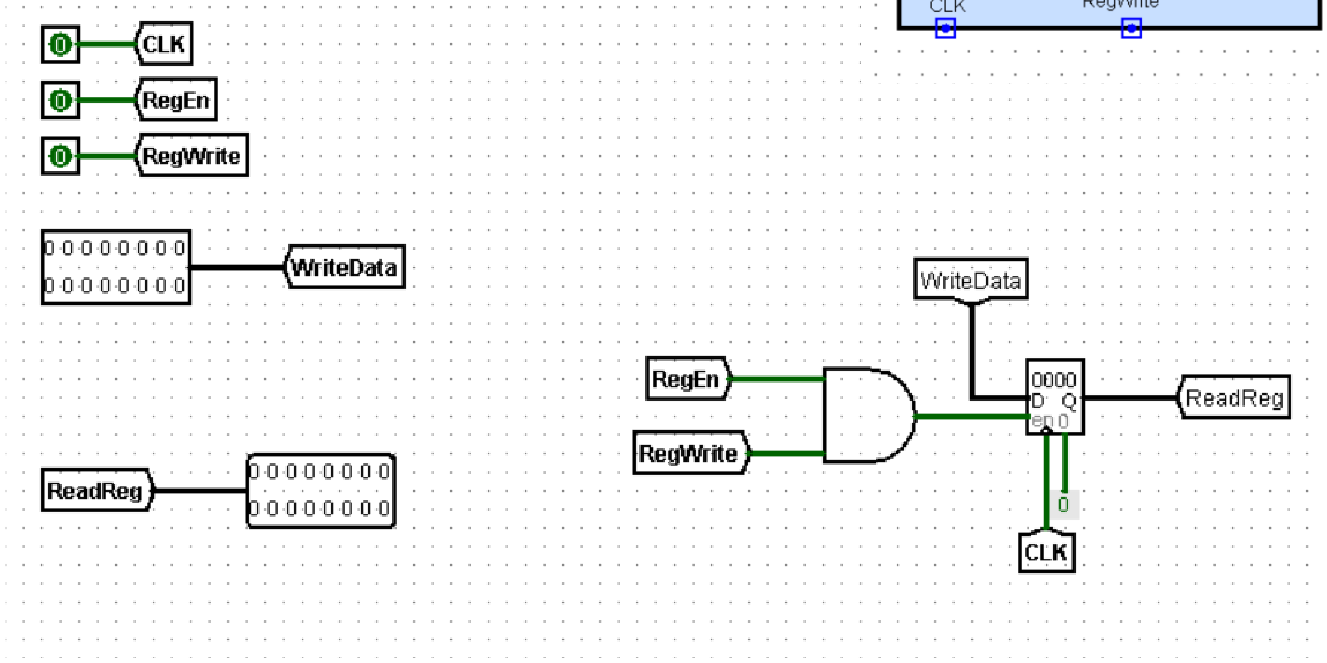
2.3. Register File



For Register file we have 2 sub circuit:

1. Sub Register
2. Register Selector

2.3.1. Sub Register



A sub register is a component that represents an individual register within the processor's data path. It has two data ports: one for input and one for output. It also has an enable signal that controls when the register can participate in data transfers.

To enable a sub register for writing data, two conditions must be met:

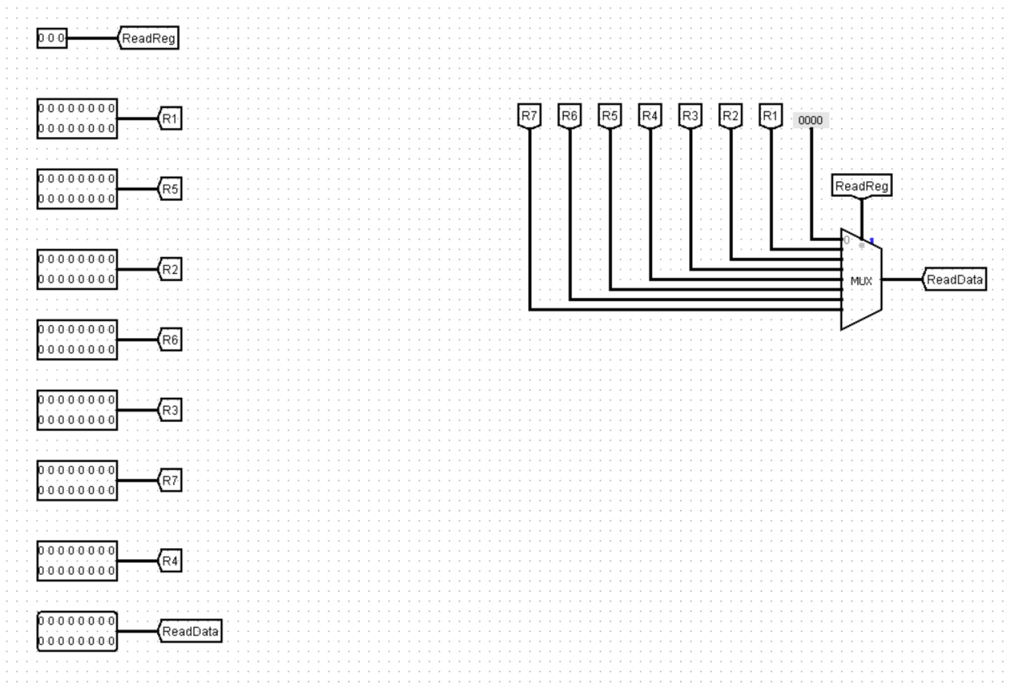
1. The general RegWrite signal must be active (typically logic 1). This signal indicates that a write operation is happening to one of the registers in the data path.
2. The sub register's specific enable signal (often called reg_enable) must also be active (logic 1). This signal selects which specific register within the register file will be written to.

In essence, the RegWrite signal acts as a global write enable for the entire register file, while the Reg_enable signal acts as a specific selector for a particular sub register. Both signals need to be active for a successful write operation to a specific register.

2.3.2. Register Selector

To choose the correct register for reading data during instruction execution, we employ a specialized circuit called the register selector.

The register selector receives a control signal derived from the decoded instruction. This signal typically consists of 3 bits, representing the register number within the register file. For 3 control bits, this allows selection from $2^3 = 8$ register.



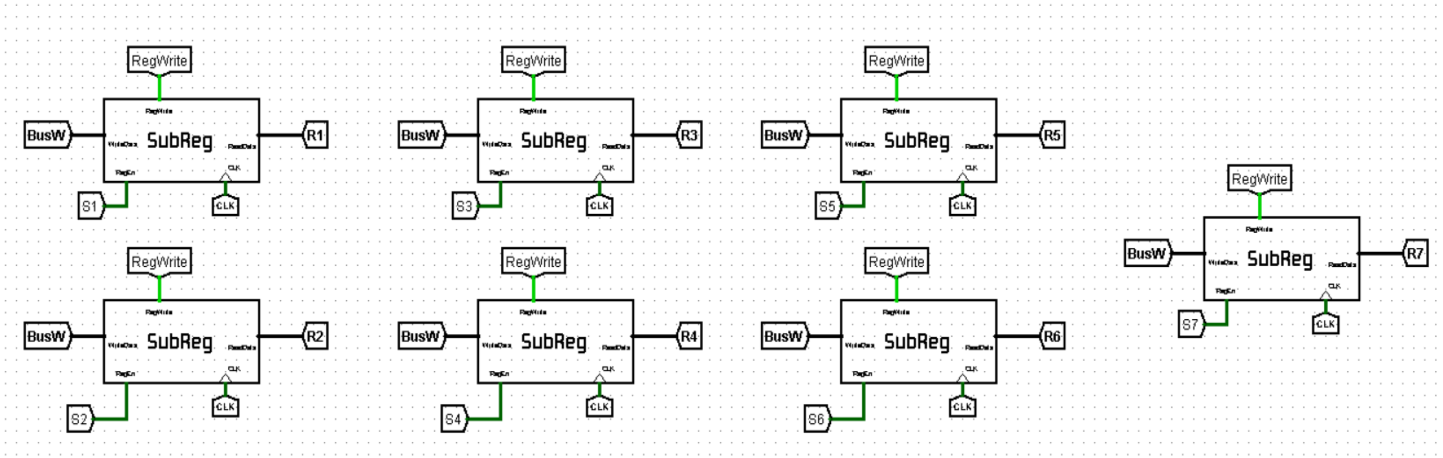
A common implementation for the register selector is a 3*8 multiplexer (MUX). Here's how it works:

- 3 select lines: These receive the control signal bits from the decoded instruction, specifying the desired register to read from.
- 8 data input lines: Each line connects to the output of a different sub register in the register file.
- 1 data output line: This line provides the data from the selected register, which becomes the source data for the instruction.

During instruction execution, when the processor needs to read data from a register, the decoded Instruction provides the control signal to the MUX. The MUX then routes the data from the selected register's output line to the data output line of the register selector. This data becomes the source for further processing within the instruction.

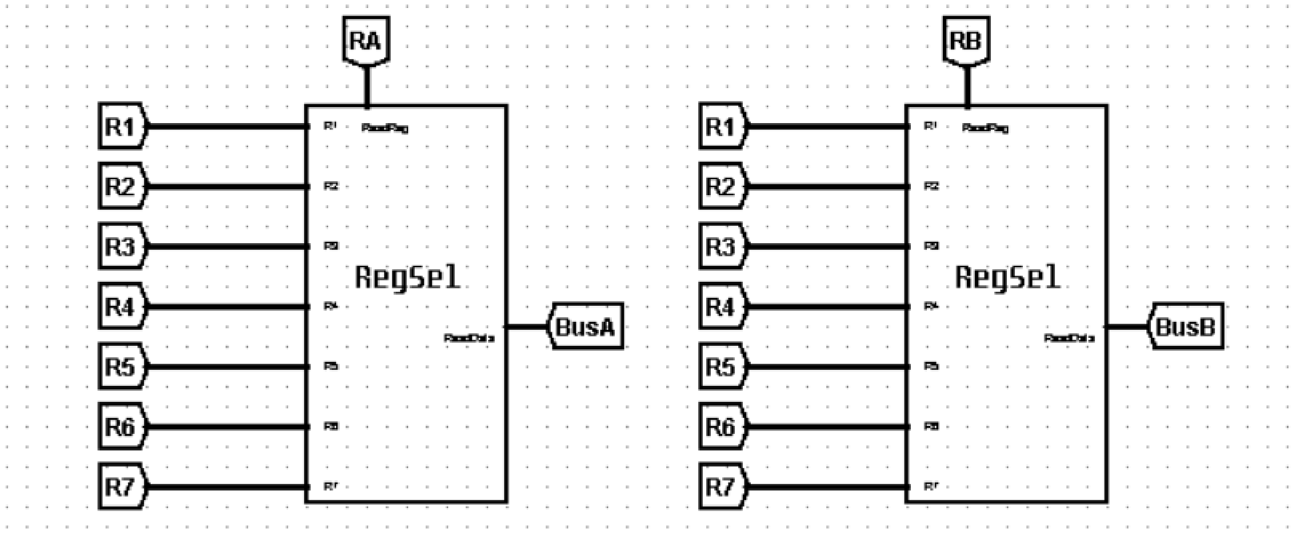
2.3.3. Building the sub circuits

Our foundation lies in the subregister, a fundamental building block discussed earlier. Each subregister represents an individual register in the data path, responsible for data input, output, and enabling signals.
To create a complete register file with eight registers, we'll replicate this subregister seven times. This provides us with the core components for registers r1 through r7.
(register r0 is typically hardwired with zero to ensure consistent behavior and simplify operations within the process)



2.3.4. Reading from the Register File

The register file provides the ability to read data from two registers simultaneously. To achieve this, we employ two register selectors. Each selector receives a control signal (typically 3 bits) from the decoded instruction, allowing it to choose a specific register for reading.

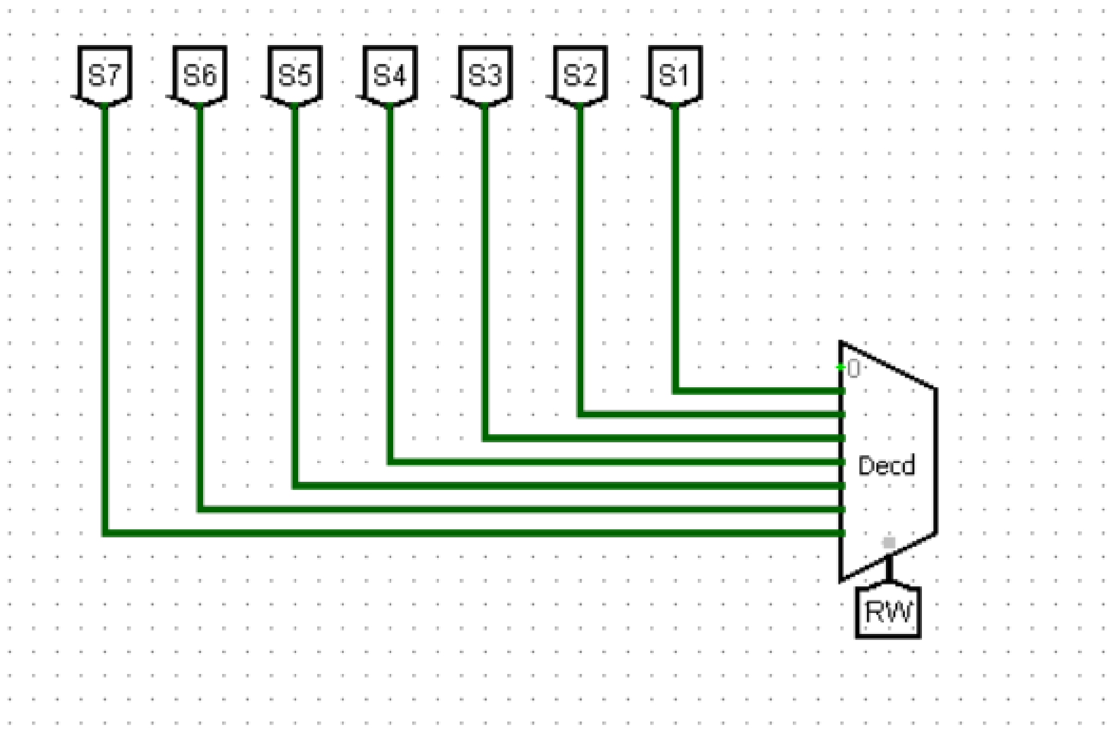


2.3.5. Writing to the Register File

While the register file can receive data through a single input (data.in), only one register can be written to at a time. To ensure this controlled write operation, we utilize two signals:

- RegWrite Signal: This global signal acts as a master switch. When active (typically logic 1), it indicates that a write operation is happening to one of the registers.
- Register Enable Signal: Each register within the file has its own enable signal (often a 3-bit field decoded from the instruction). Only when both the RegWrite signal is active and the correct register's enable signal is activated, will data be written to that specific register.

The register enable signal utilizes a decoder circuit. This decoder receives the 3-bit control signal from the instruction and activates only one of its output lines based on the binary value. This ensures that only the selected register's enable signal goes high, allowing data to be written to that specific register within the register file.

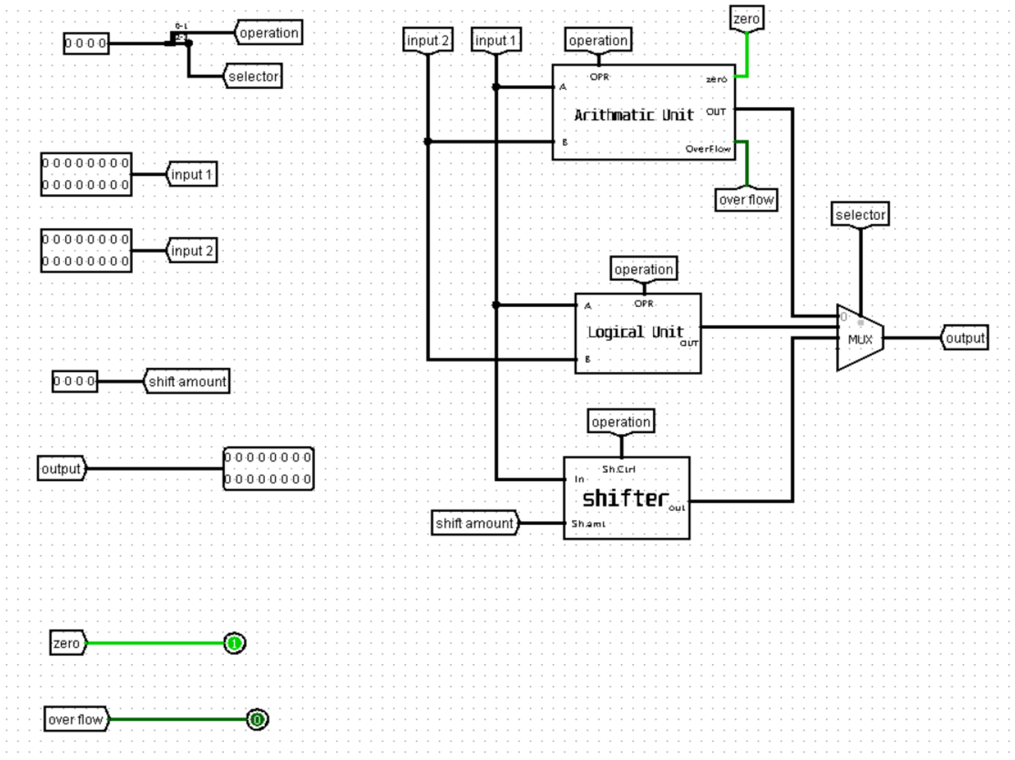


2.4. ALU

Every computer needs a part that can do math and follow instructions. Inside the processor, this critical job belongs to the **ALU**, or Arithmetic Logic Unit. Think of it as a tiny, built-in calculator that can add, subtract, compare things, and even move bits around.

But the ALU isn't just one simple unit. To handle all these tasks efficiently, it's actually made up of three smaller specialists working together:

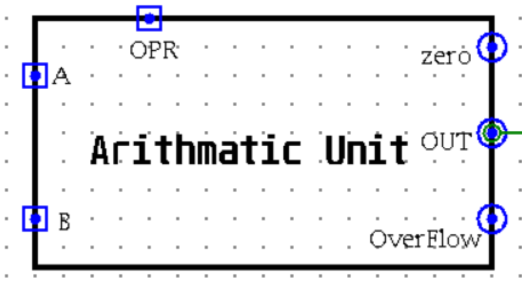
- **The Arithmetic Unit:** This part focuses on math operations, like adding and subtracting numbers.
- **The Logic Unit:** This specialist handles comparisons and logical operations like AND or OR helping the processor make decisions.
- **The Shift Unit:** This unit is a bit shifter, moving data bits left or right as needed for various computing tasks.



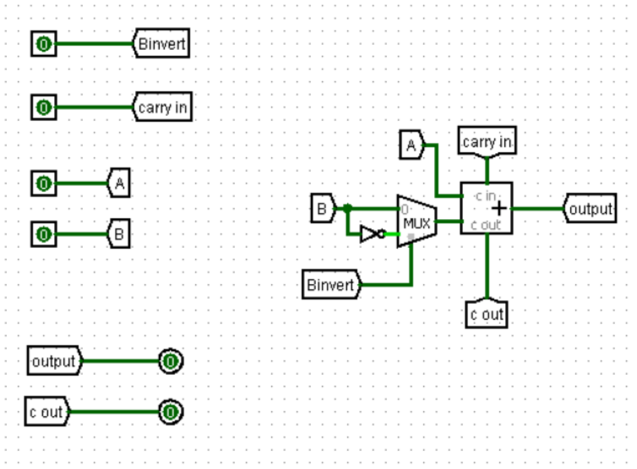
By working together, these three units within the ALU give the processor the power to perform a wide range of calculations and logical operations, making it the heart of the computer's processing power.

In the next sections, we'll explore each of these internal blocks of the ALU in more detail, diving deeper into their specific functionalities.

2.4.1. Arithmetic Unit



The Arithmetic Unit (AU) is the core component within the ALU responsible for all arithmetic operations. It's essentially a high-speed calculator unit specifically designed for binary data manipulation within the processor.



The foundation of the AU lies in the humble 1-bit adder. This fundamental circuit can perform the addition of two single binary digits (bits), generating a sum output and a carry output. By cascading multiple 1-bit adders together, the AU can perform addition on larger binary numbers, like adding the contents of two registers within the processor.

The 1-bit adder itself doesn't inherently know whether to perform addition or subtraction. This decision is made by a control signal provided by the ALU control unit. This signal, often denoted as Cin (Carry In), dictates the adder's behavior:

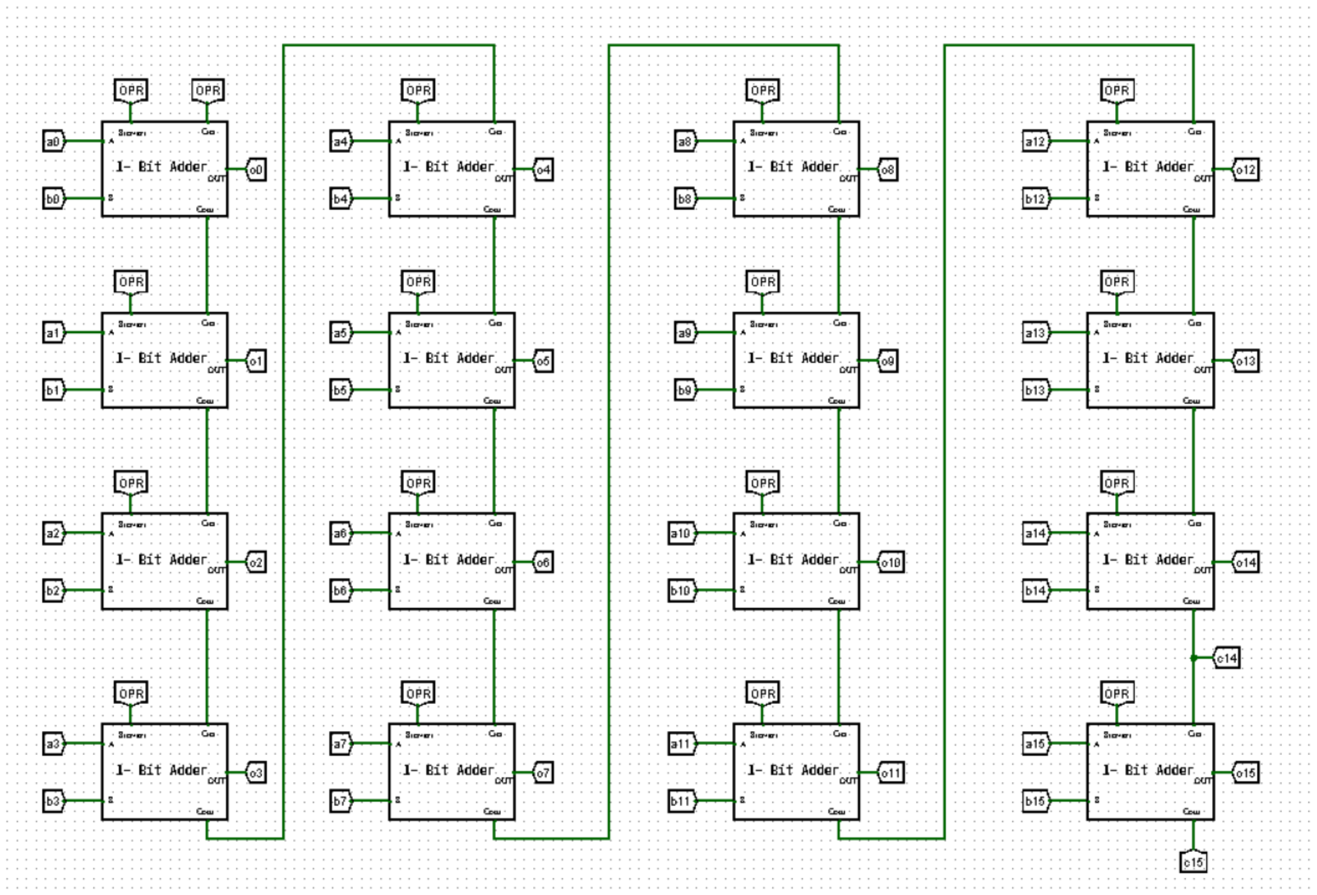
- Cin = 0: When Cin is set to 0, the adder performs a standard addition of the two input bits (A and B).
- Cin = 1: When Cin is set to 1, the adder behaves differently. It takes the complement of the second input bit (B) and adds 1 (effectively performing 2's complement addition). This allows the AU to handle subtraction through clever manipulation.

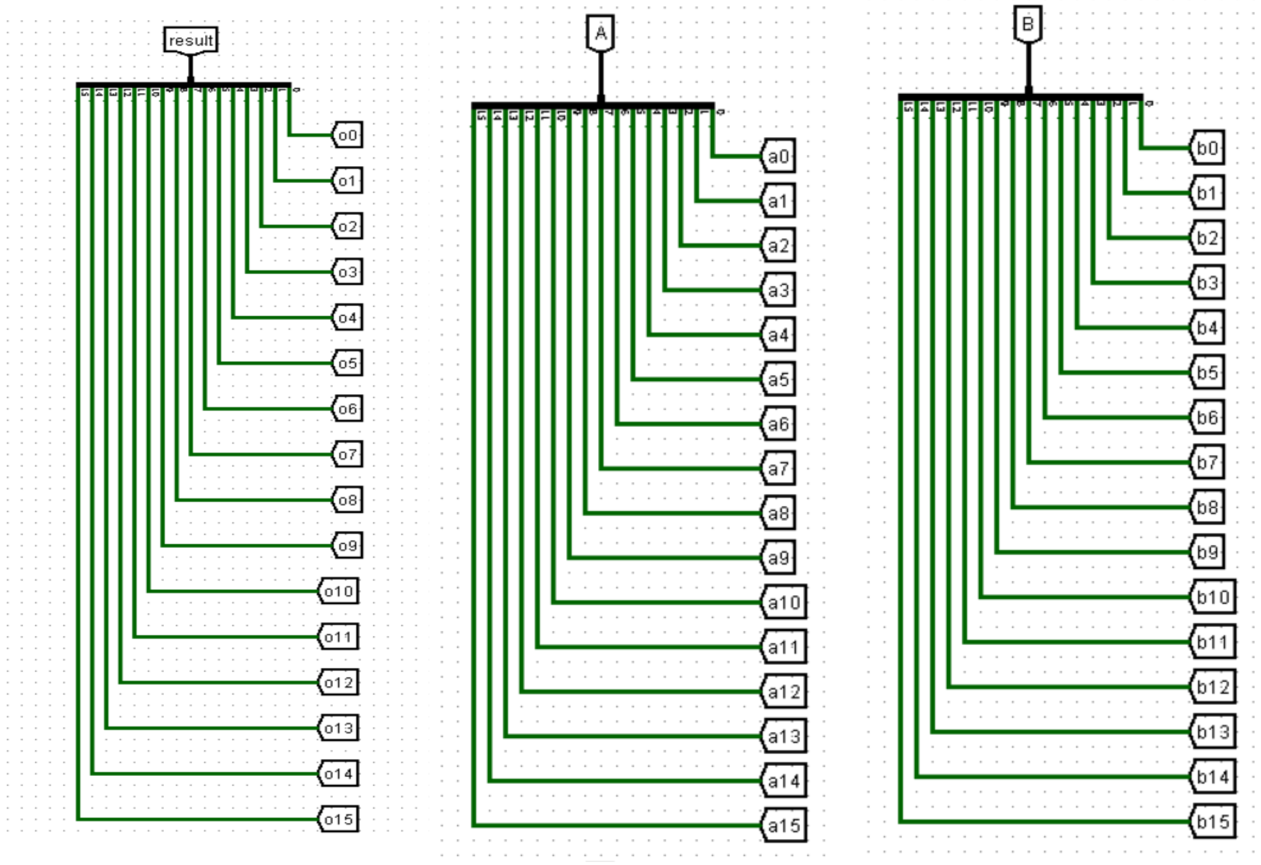
In essence, the AU cleverly transforms subtraction into an addition problem using the power of 2's complement.

We'll need 16 of these 1-bit adders working together. Why 16? Because we want to handle 16-bit numbers, so each adder tackles one bit of the larger numbers.

Each adder has two inputs (A and B) and two outputs (sum and carry). We'll chain these adders together like dominos. The carry output from one adder becomes the carry input (Cin) for the next one, ensuring a smooth calculation across all 16 bits.

Our 16-bit input gets divided into 16 individual bits. Each bit is fed into one of our 16 adders. Each adder produces a sum bit, which contributes to the final answer. We need to combine these 16 sum bits into the final result.

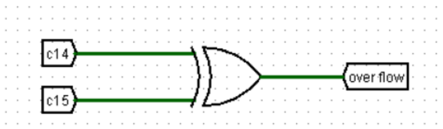

OPR
1- Bit Adder
a0
b0
o0
c14
c15


result
A
B

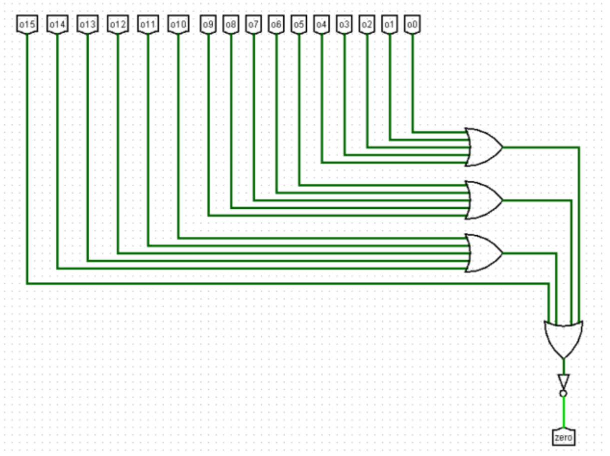
Now, we need a way to choose the final output based on what operation we want to perform (add, subtract, set less than (SLT), or set less than unsigned (SLTU)). Here's where a **multiplexer (MUX)** comes in.

We have a 2-bit control signal that acts like a switch setting. The MUX will connect the appropriate output to the final result line.

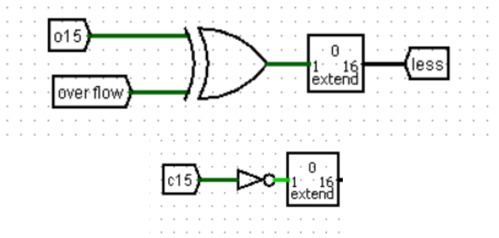
Overflow Flag: To detect overflow (when the result is too large to fit in the designated number of bits), we can use an XOR operation on the last two carry outputs (c14 and c15).



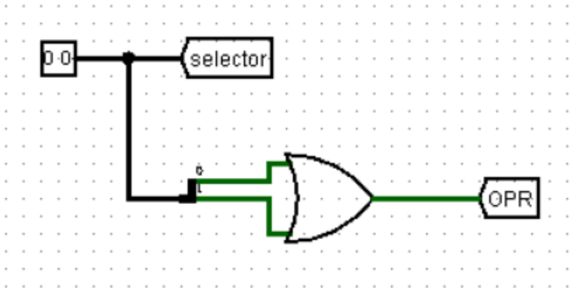
The Zero Flag indicates whether the final result of the operation is zero or not. To determine this, we can perform an **OR operation** on all the 16 output bits from the adders. If the result of the OR operation is 0, it means all individual bits in the final result are 0, indicating a zero value. This zero flag can be a helpful signal for the processor to handle specific conditional statements or branching instructions in your code.



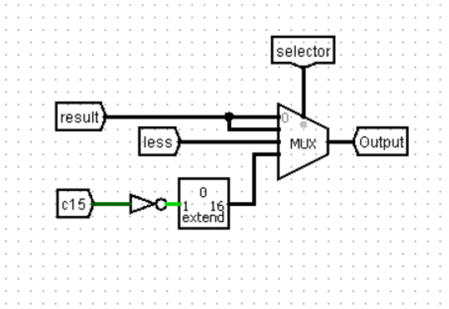
SLT and SLTU: These operations determine if one number is less than another, considering signed or unsigned values. We can use the overflow flag and the last output bit (o15) to calculate SLT, while inverting the last carry (c15) gives us SLTU.



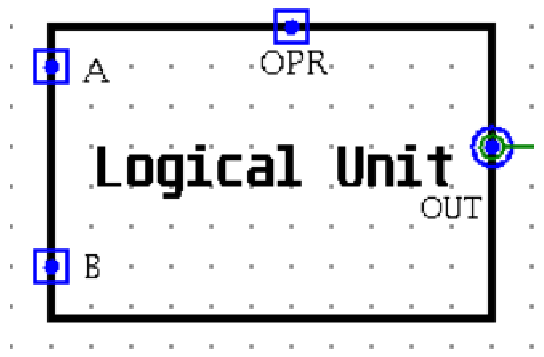
Operation Signal (opSignal): This is a crucial 1-bit signal that tells the first adder (remember, the one with the special Cin control) whether to perform a regular addition (opSignal = 0) or use 2's complement for subtraction. We'll achieve this by cleverly combining the 2-bit control signal using an AND operation.



Connecting the MUX: Finally, the MUX connects the results from the 16 adders (for addition or subtraction) and the calculated SLT and SLTU outputs to the final result line based on the control signal.

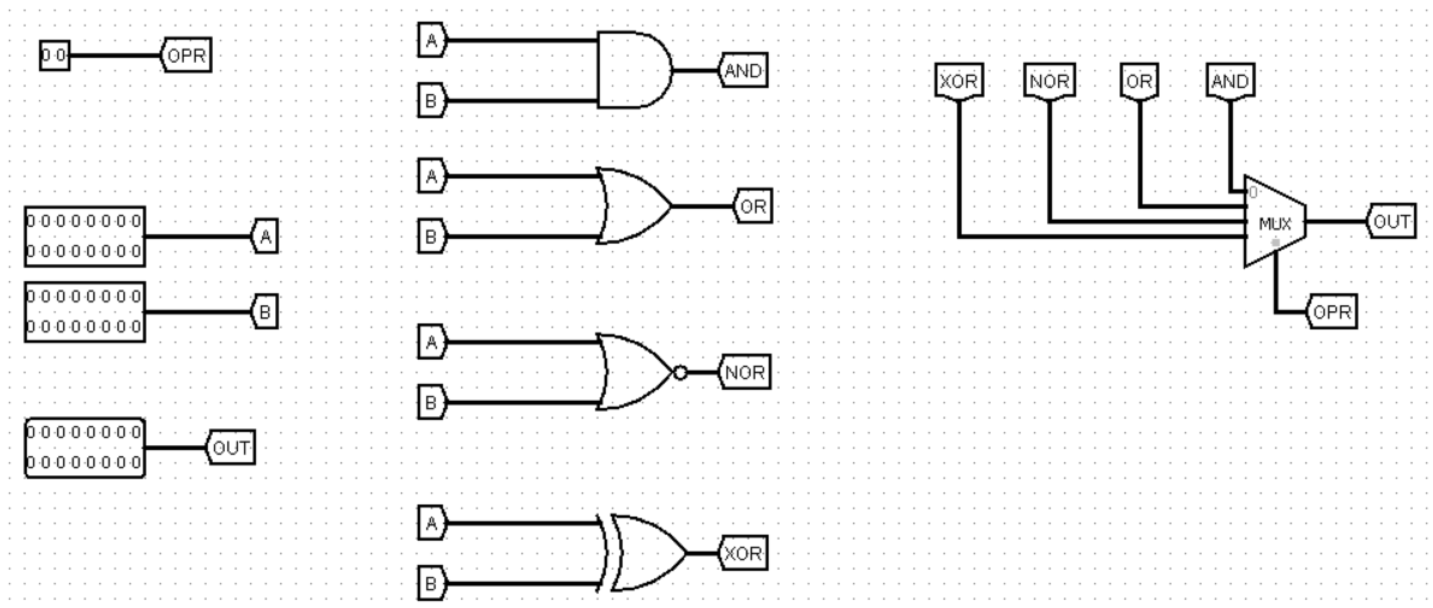


2.4.2. Logic Unit

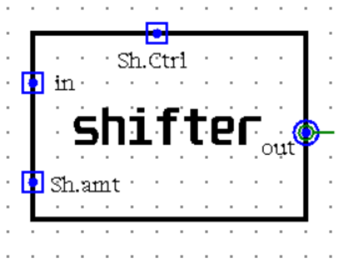


The Logic Unit (LU) keeps things simple. It only needs four basic logic gates (AND, OR, NOR, XOR) to work its magic. A 2-bit control signal acts like a code, telling a handy multiplexer (MUX) which gate to use.

Internal Architecture



2.4.3. Shift Unit



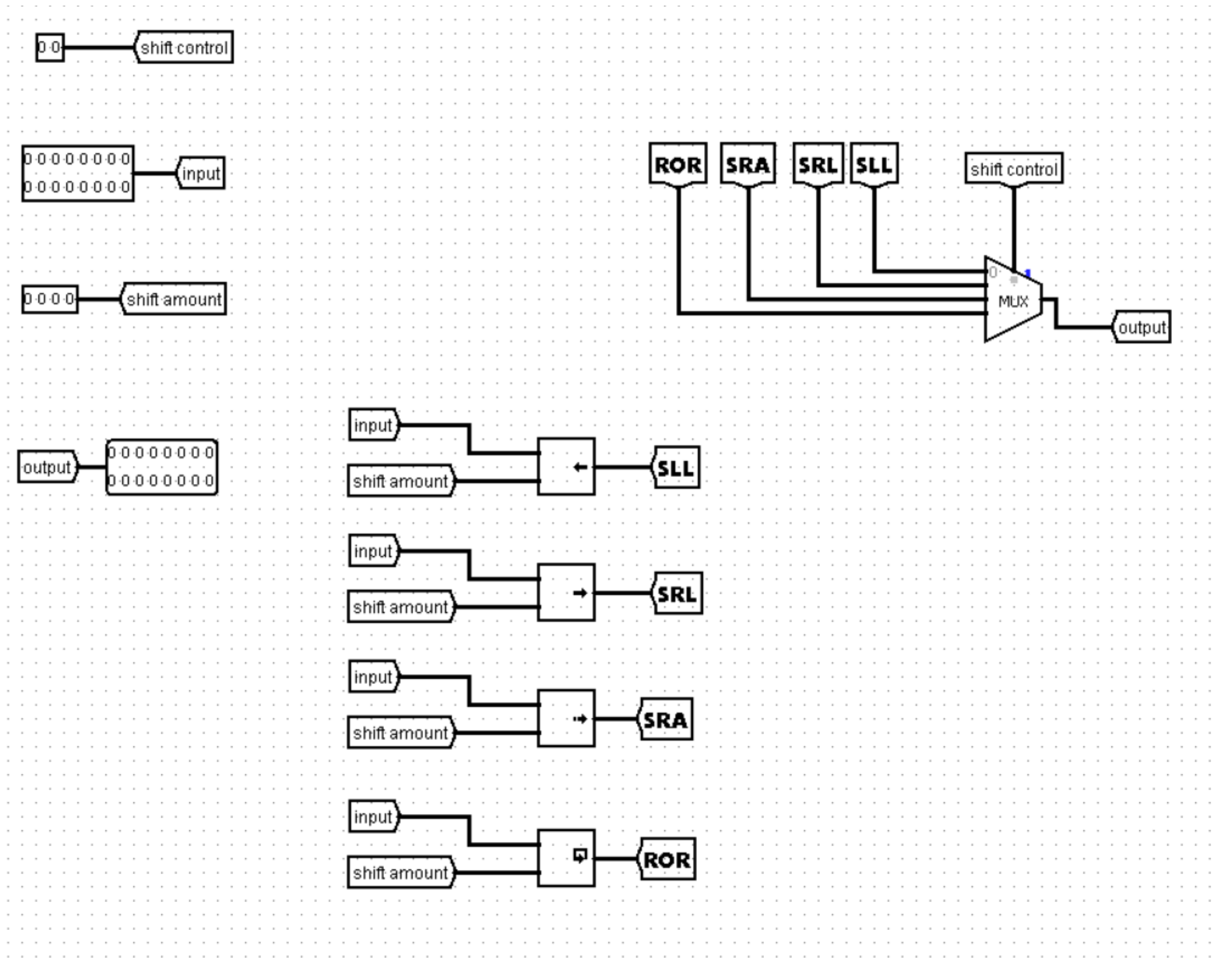
The Shift Unit focuses on one key task: **moving data bits around**.

Just like the LU, the Shift Unit avoids complexity, Four Basic "Shifting Moves": We just need these four built-in functions (SLL, SRL, SRA, ROR) to handle all the different nudge scenarios.

To control how many positions the bits are nudged, the Shift Unit also relies on a special **four-bit input** called the **shift amount**. This input acts like a precise instruction, telling the unit exactly how many marbles (data bits) to move in each operation (left or right shift).

Remember that awesome MUX from before? It makes a comeback here! A control signal tells the MUX which specific shifting move (SLL, SRL, SRA, or ROR) to perform on the data.

Internal Architecture



Now, let's see how they work together!

The Units Get Their Instructions: Each unit (AU, LU, and Shift) receives a special 2-bit control signal. This signal acts like a tiny code, telling the unit what specific task to perform.

The Shifter Gets Specific: In addition to the 2-bit control signal, the Shift Unit has a bonus input – a 4-bit shift amount. This input works like a precise instruction, specifying exactly how many positions to nudge the data bits (left or right) for various shifting operations.

Results Start Flowing: The AU outputs useful flags like zero and overflow to indicate specific conditions during calculations.

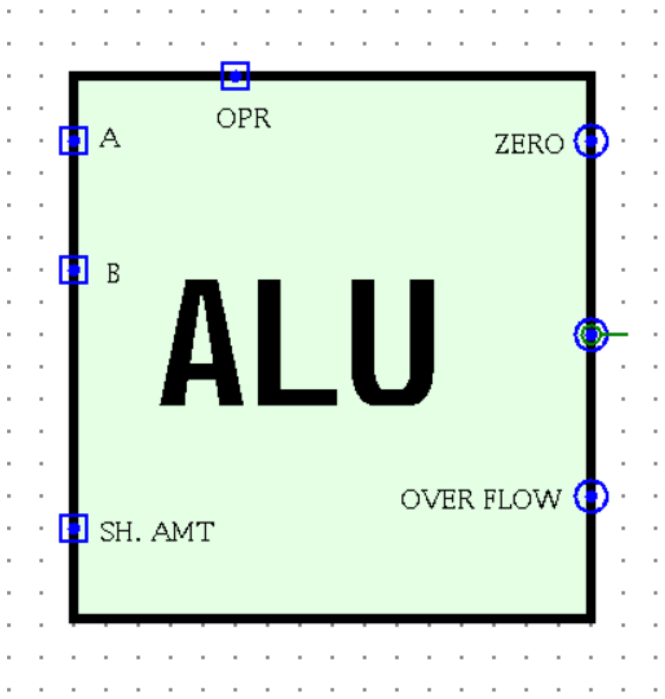
The Multiplexer Steps Up: A critical component in this stage is the multiplexer (MUX). The outputs from all three units (AU, LU, and Shift) are directed to the MUX for selection.

MUX Makes the Final Call: We only need a 2-bit control signal for the MUX (separate from the unit control signals). This 2-bit signal acts like a switch, telling the MUX which specific unit's output to send as the final ALU result.

By combining these control signals, we end up with a 4-bit overall control signal for the ALU. Here's the breakdown:

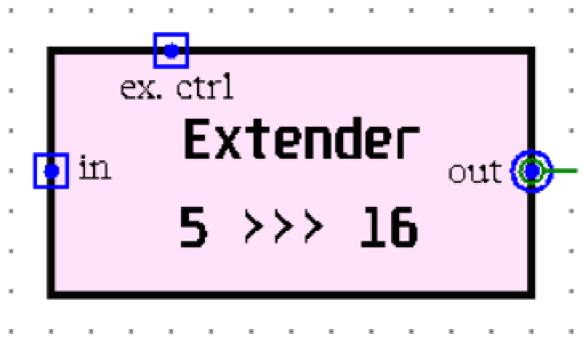
- 2 bits for the unit control signals (AU, LU, or Shift).
- 2 bits for the MUX control signal (which unit's output to use).

With this efficient system, the ALU can perform various tasks based on the chosen operation and data manipulation needs.

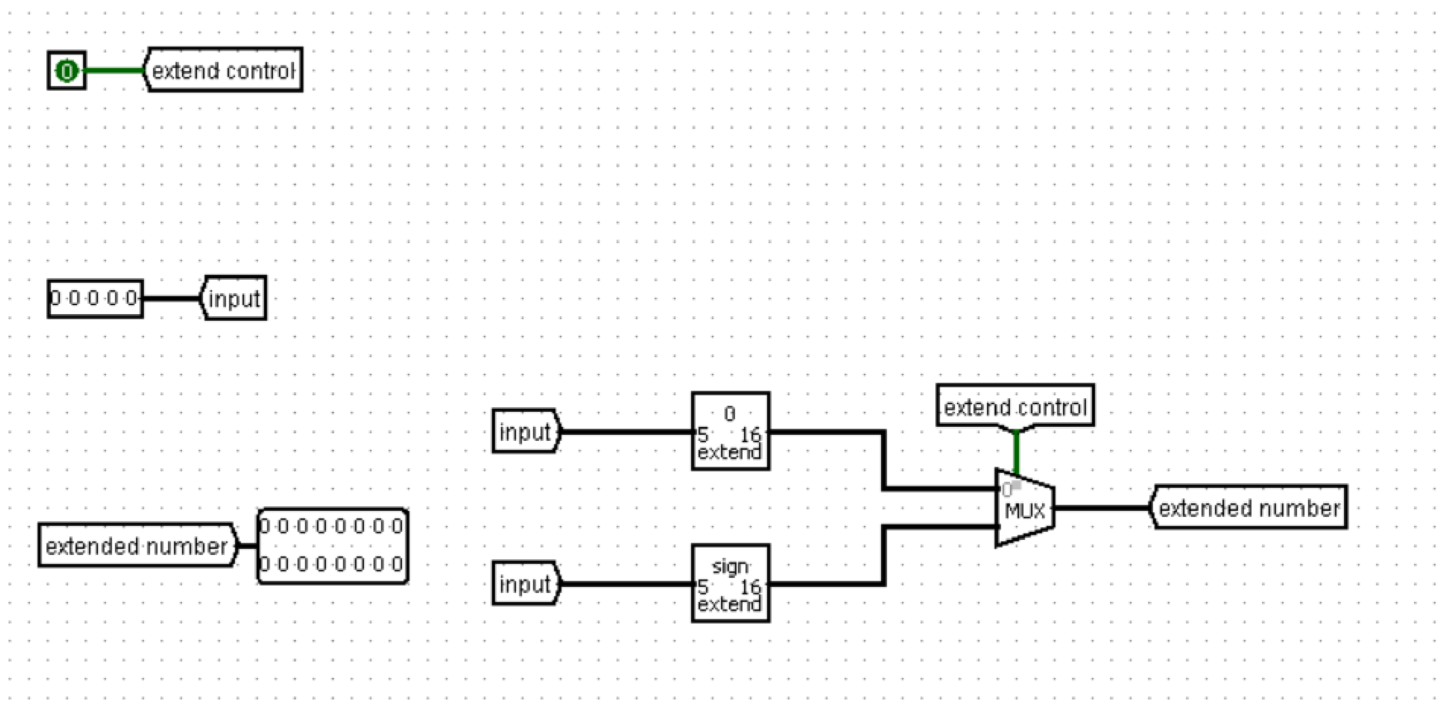


2.5. Extender

The extender is a special circuit that takes this small immediate value (5 bits in our example) and expands it to a full size (maybe 16 bits, depending on the processor). This gives the tiny value the power to contribute to more complex operations!

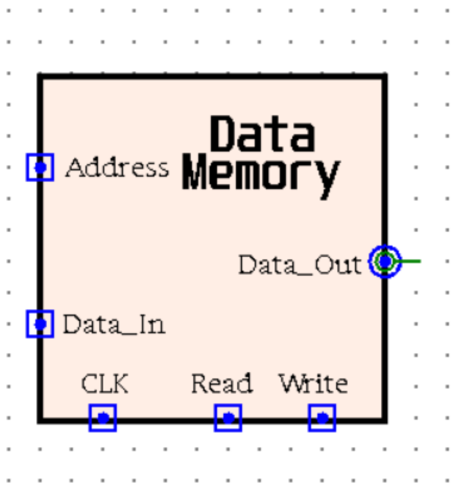


Internal Architecture

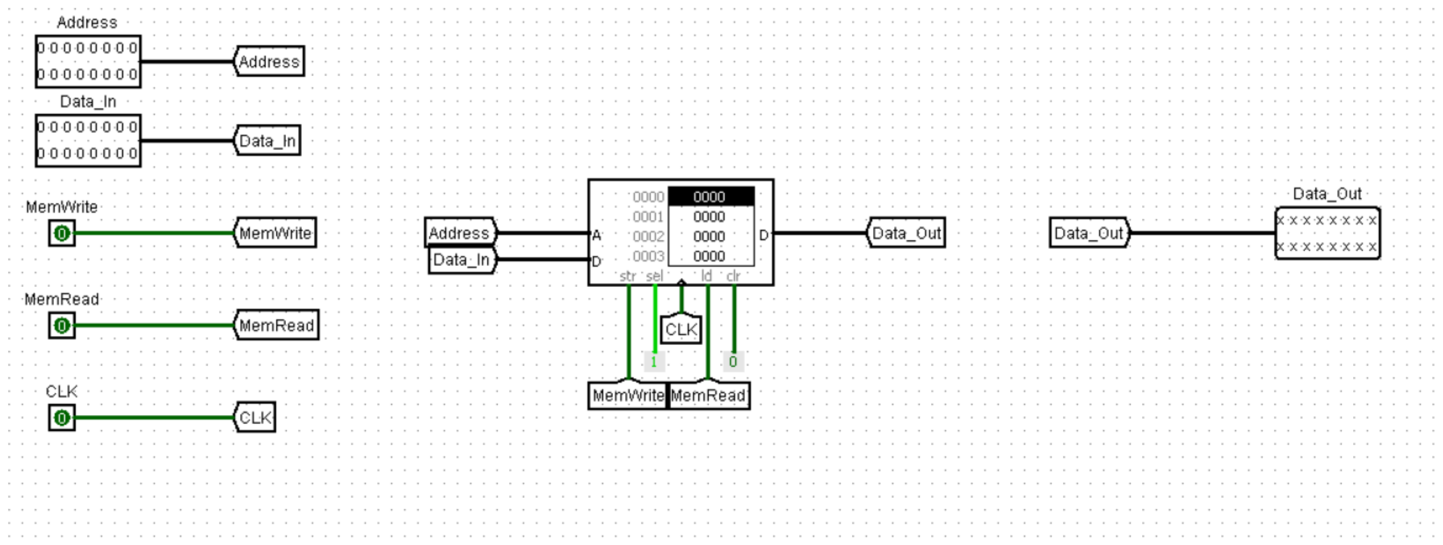


2.6. Data Memory

The data memory operates akin to a Random Access Memory (RAM) system, offering both read and write functionalities. It encompasses five inputs: an address pathway for accessing data, a data pathway for inputting data to be stored, signals for MemWrite (activated when writing) and MemRead (activated when reading), and a clock signal for synchronization. Moreover, it possesses a sole output dedicated to delivering the retrieved data during read operations.

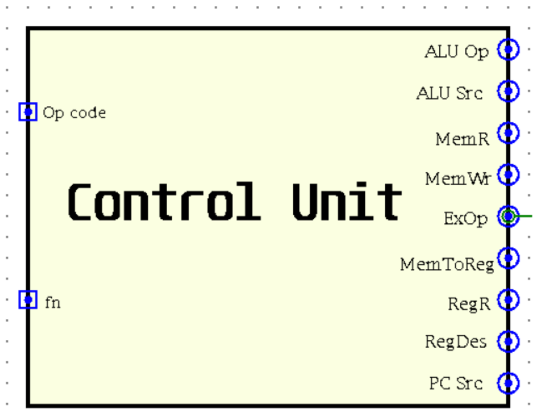


Internal Architecture

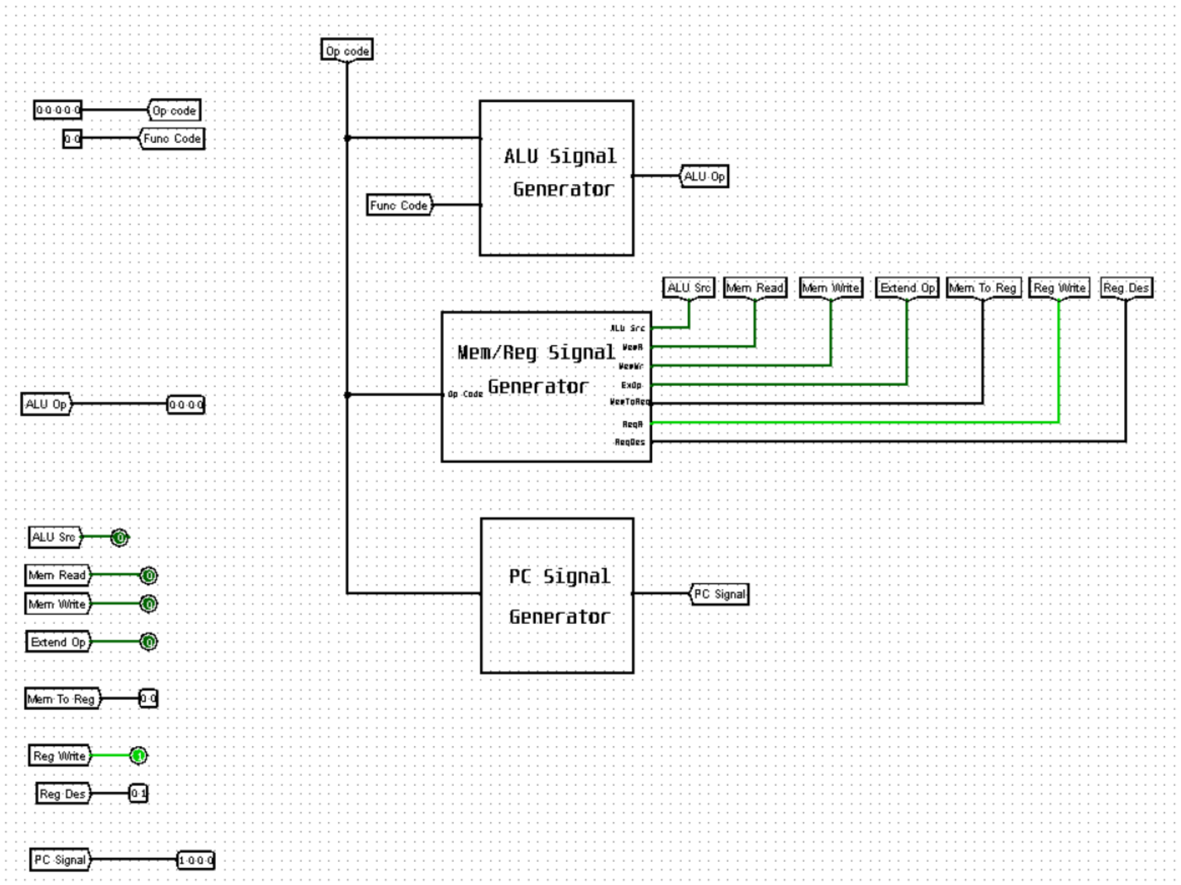


2.7. Control Unit

Utilizing the 5-bit Opcode and 2-bit Function, the control unit orchestrates the execution of various operations based on the decoded instruction.



The internal design



2.7.1. Input signals

7.1.1. The Opcode (5-bit):

The opcode tells the processor what to do (e.g., add, and, load, store, branch, jump…etc)

7.1.2. The Function (2-bit):

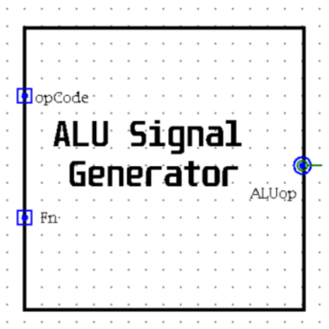
The R-type instruction format utilises the 2-bit function field to precisely control ALU operations.

2.7.2. Output signals

The output signals originate from three distinct subcircuits, each responsible for regulating various facets of the processor's functionality.

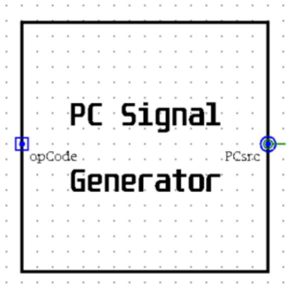
7.2.1 ALU signal generator:

Utilising the Opcode and Function, It generates the **ALUop signal, a 4-bit code** that dictates the ALU operation to be performed. The lower 2 bits determine the operation type (arithmetic, logic, or shift), while the upper 2 bits specify the particular operation (add, sub,etc., and, or, etc., sll, srl, etc.).

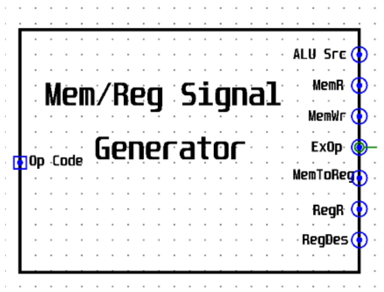


7.2.2 PC signal generator:

It utilises the Opcode to generate a **4-bit PCsrc signal.** This signal determines the next Program Counter (PC) value, selecting between incrementing by 1, jumping to a specified address, or branching based on certain conditions.



7.2.3 Mem/Reg signals generator:



It employs the Opcode to derive values for the remaining control signals necessary for proper operation. These signals encompass various aspects of the CPU operation, such as memory access, register manipulation, and data transfer.

- **ALUsrc (1-bit):** This signal dictates the source of the second operand for the ALU. A logic value of 1 instructs the ALU to utilise the immediate value from the instruction, while a 0 specifies a register value (typically rt in the instruction).

ALUsrc	
Signal	**Source**
0	Register
1	Imm Value

- **Ext_control (1-bit):** This signal controls the extension mode for immediate values. A logic value of 1 keeps the sign of the immediate, while a logic value of 0 sets the empty bits to zero.

Ext_control	
0	Zero Extend
1	Sign Extend

- **RegWrite (1-bit):** This control signal governs write operations to the register file. A logic value of 1 enables writing, allowing data to be stored in the designated register. Conversely, a logic value of 0 disables writing, preventing any changes to the register's content.

- **RegDst (2-bit):** This control signal selects the destination register for write operations based on the instruction type. It has different values depending on the instruction type: 0 for I-type instructions, 1 for R-type instructions, and 2 or 3 for jump instructions.

RegDst	
Signal	**Destination**
00	Rt
01	Rd
10	$1
11	$7

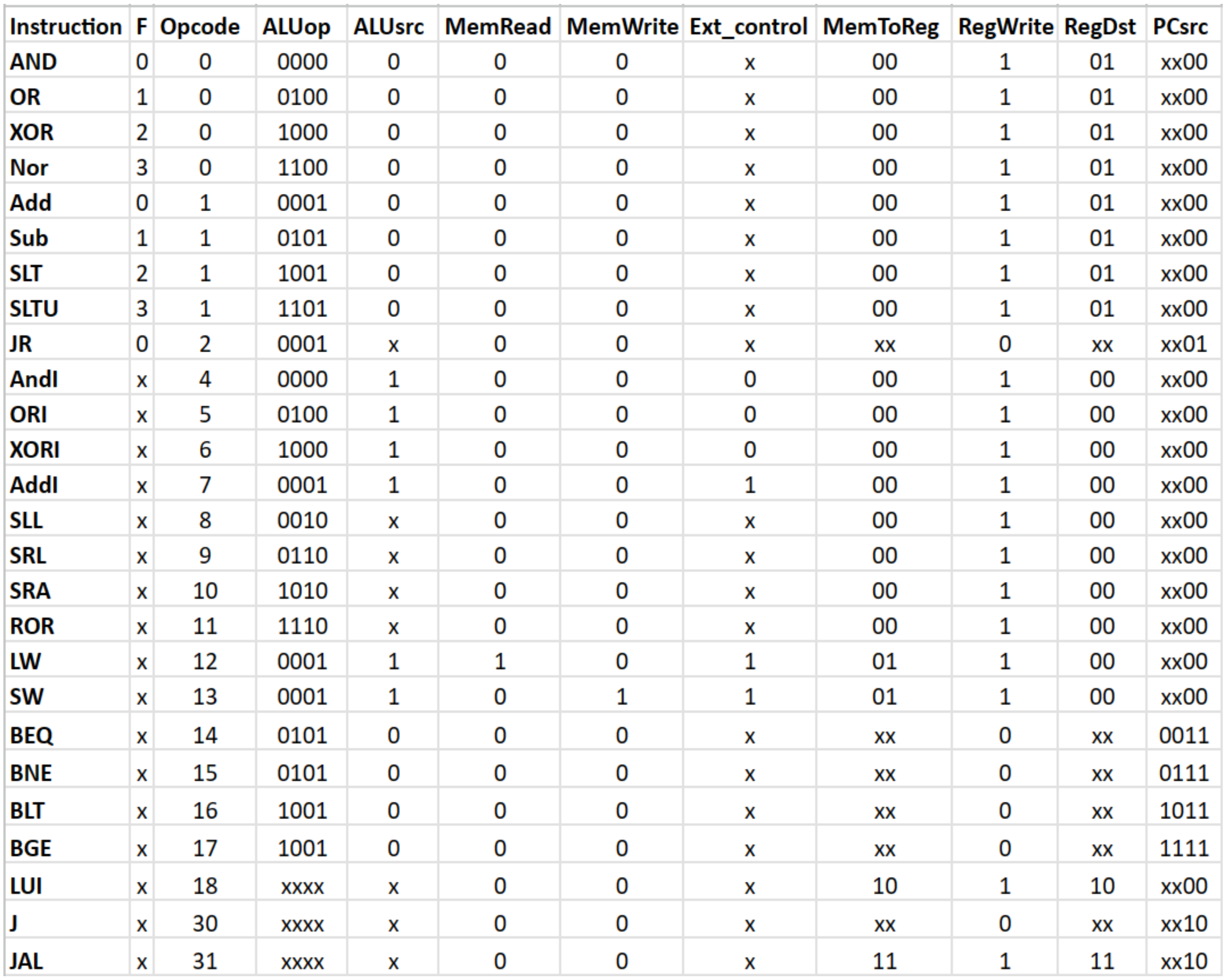
- **MemRead (1-bit):** This signal tells the processor to grab data from memory. The control unit activates this signal whenever an instruction needs something from memory.

- **MemWrite (1-bit):** This signal, on the other hand, tells the processor to write data to memory. The control unit activates it whenever an instruction needs to store something in memory.

- **MemToReg (2-bit):** This control signal selects the data source for writing to the designated register.

MemToReg	
Signal	**Data**
00	ALU Result
01	Memory Out
10	Imm(11)
11	PC + 1

2.7.3. Signals values for each instruction



Instruction	F	Opcode	ALUop	ALUsrc	MemRead	MemWrite	Ext_control	MemToReg	RegWrite	RegDst	PCsrc
AND	0	0	0000	0	0	0	x	00	1	01	xx00
OR	1	0	0100	0	0	0	x	00	1	01	xx00
XOR	2	0	1000	0	0	0	x	00	1	01	xx00
Nor	3	0	1100	0	0	0	x	00	1	01	xx00
Add	0	1	0001	0	0	0	x	00	1	01	xx00
Sub	1	1	0101	0	0	0	x	00	1	01	xx00
SLT	2	1	1001	0	0	0	x	00	1	01	xx00
SLTU	3	1	1101	0	0	0	x	00	1	01	xx00
JR	0	2	0001	x	0	0	x	xx	0	xx	xx01
AndI	x	4	0000	1	0	0	0	00	1	00	xx00
ORI	x	5	0100	1	0	0	0	00	1	00	xx00
XORI	x	6	1000	1	0	0	0	00	1	00	xx00
AddI	x	7	0001	1	0	0	1	00	1	00	xx00
SLL	x	8	0010	x	0	0	x	00	1	00	xx00
SRL	x	9	0110	x	0	0	x	00	1	00	xx00
SRA	x	10	1010	x	0	0	x	00	1	00	xx00
ROR	x	11	1110	x	0	0	x	00	1	00	xx00
LW	x	12	0001	1	1	0	1	01	1	00	xx00
SW	x	13	0001	1	0	1	1	01	1	00	xx00
BEQ	x	14	0101	0	0	0	x	xx	0	xx	0011
BNE	x	15	0101	0	0	0	x	xx	0	xx	0111
BLT	x	16	1001	0	0	0	x	xx	0	xx	1011
BGE	x	17	1001	0	0	0	x	xx	0	xx	1111
LUI	x	18	xxxx	x	0	0	x	10	1	10	xx00
J	x	30	xxxx	x	0	0	x	xx	0	xx	xx10
JAL	x	31	xxxx	x	0	0	x	11	1	11	xx10

2.7.4. Signals expressions

ALUop_0	~OP3 ~OP2 OP0 + ~OP3 ~OP2 OP1 + ~OP3 OP1 OP0 + OP3 OP2 + OP4
ALUop_1	OP3 ~OP2
ALUop_2	~OP4 ~OP3 ~OP2 ~OP1 F0 + ~OP3 OP2 ~OP1 OP0 + OP3 ~OP2 OP0 + OP3 OP2 OP1
ALUop_3	~OP3 ~OP2 ~OP1 F1 + ~OP2 OP1 OP0 + ~OP3 OP2 OP1 ~OP0 + OP3 ~OP2 OP1 + OP4

PCsrc_0	~OP4 ~OP3 ~OP2 OP1 + ~OP4 OP3 OP2 OP1 + OP4 ~OP1
PCsrc_1	OP3 OP2 OP1 + OP4 ~OP1
PCsrc_2	OP0
PCsrc_3	~OP1

RegDst_0	~OP3 ~OP2 ~OP1 + OP4 OP3
RegDst_1	OP4

MemToReg_0	OP3 OP2
MemToReg_1	OP4

ALUsrc	~OP3 OP2 + OP3 ~OP1 + OP4 OP1
Ext_control	OP1 OP0 + OP3
RegWrite	~OP4 ~OP3 ~OP1 + ~OP4 ~OP1 ~OP0 + ~OP4 ~OP3 OP2 + ~OP4 OP3 ~OP2 + OP4 ~OP3 ~OP2 OP1 ~OP0 + OP4 OP3 OP2 OP1 OP0
MemWrite	~OP4 OP3 OP2 ~OP1 OP0
MemRead	~OP4 OP3 OP2 ~OP1 ~OP0

3. Introduction (Phase two)

In the previous phase, we explored the basic single-cycle processor design. While it guarantees correct execution for each instruction, it has a significant drawback: a long clock cycle that limits overall performance. This bottleneck arises because all instruction steps (fetching, decoding, execution, memory access, and writing back) must be completed within a single clock cycle.

Imagine a long assembly line where workers must finish their tasks on a product before it moves on. Similarly, the single-cycle design forces all instructions to wait in line, even if some steps in one instruction could be happening concurrently with another instruction.

To overcome this limitation and achieve higher efficiency, we now turn to a powerful technique called pipelining. Unlike the single-cycle design, pipelining breaks down instruction execution into smaller, specialized stages. We'll be focusing on a *five-stage pipeline* where each stage performs a specific task:

1. **Fetch:** The instruction is retrieved from memory.
2. **Decode:** The instruction is decoded to understand the operation to be performed and the operands involved.
3. **Execute:** The ALU (Arithmetic Logic Unit) performs the operation based on the decoded instruction.
4. **Memory Access**: Data is read from or written to memory (if required by the instruction).
5. **Write Back:** The result of the operation is written back to the register file.

This allows for parallel processing of multiple instructions at different stages, significantly improving the processor's throughput (number of instructions completed per unit time). But there's a catch! How do we hold onto data as it progresses through these stages?

Here's where *pipeline registers* come into play. These are temporary storage elements inserted between pipeline stages. They act like buffers, holding the results from the previous stage until the next stage is ready to receive them. By utilizing pipeline registers, instructions can flow smoothly through the pipeline without waiting for each other, maximizing the processor's efficiency.

The main components:

1. Pipeline Registers
2. Forward Unit
3. Hazard Detection Unit
4. ID Branch Forward Unit
5. Pipeline PC Control
6. Pipeline Instruction Decoder

3.1. Pipeline Registers

In a pipelined processor, one crucial component is the set of pipeline registers. These registers play a vital role by storing the necessary data for each instruction during its execution across different processing stages.

The Stages and the Need for Registers:

A typical pipelined processor can be divided into several stages, such as:

1. **Instruction Fetch (IF):** Retrieving the instruction from memory.
2. **Instruction Decode (ID):** Decoding the instruction and determining the required operations.
3. **Execution (EX):** Performing the operation on the data (often handled by the ALU).
4. **Memory Access (MEM):** Accessing memory to read or write data as needed.
5. **Write Back (WB):** Writing the final result back to the register file.

To ensure smooth and efficient instruction processing, pipeline registers are inserted between each stage. These registers act as buffers, holding the data and control signals required for the next stage while the current stage completes its task.

The Four Main Pipeline Registers:

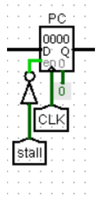
Given the five processing stages mentioned above, we require four main pipeline registers:

1. **IF/ID Register**
2. **ID/EX Register**
3. **EX/MEM Register**
4. **MEM/WB Register**

3.1.1. IF/ID Register

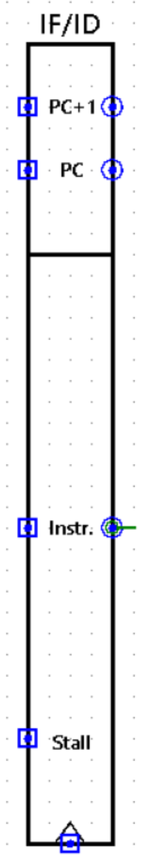
The IF/ID register serves as a critical link between the Instruction Fetch (IF) and Instruction Decode (ID) stages in the pipeline. It acts as a temporary storage for several key inputs:

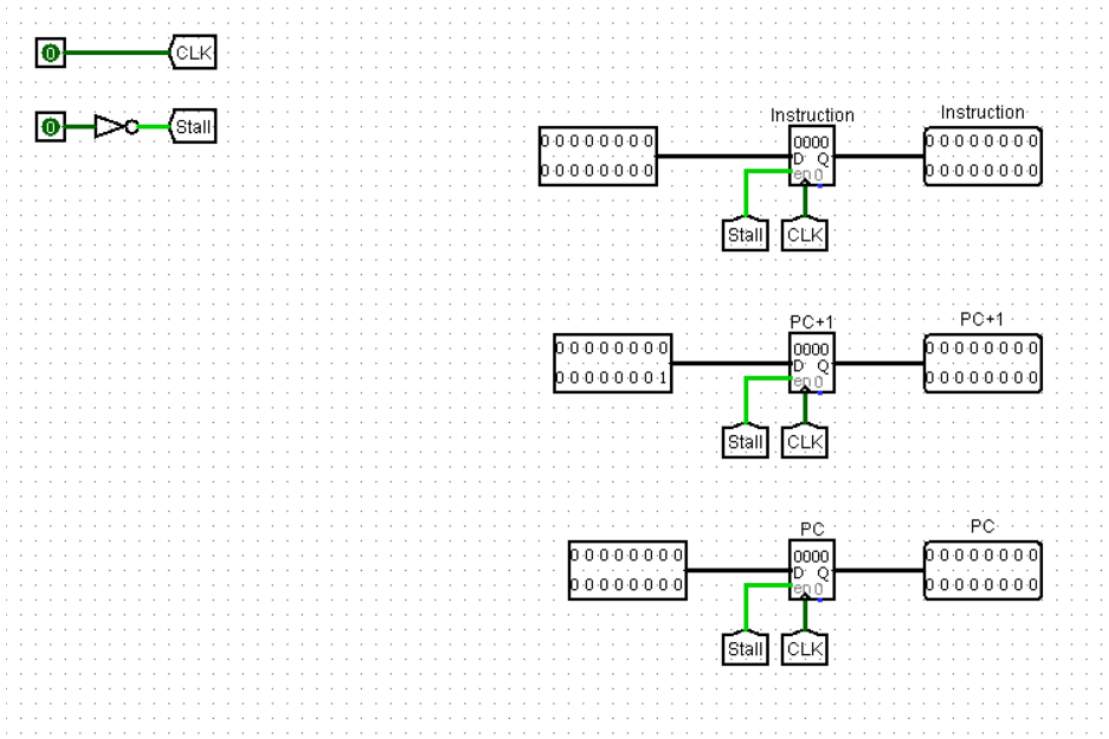
- **Instruction:** This is the actual instruction retrieved from memory during the IF stage.
- **Program Counter (PC):** The PC value points to the address of the fetched instruction.
- **PC+1:** This represents the address of the next instruction in the sequential flow.
- **Stall Signal (Inverted):** The inverted stall signal, generated by the hazard detection unit (explained later), plays a critical role in ensuring smooth pipeline operation during stalls. When the stall signal is **low** (interpreted as **high** due to the inversion):
  - The IF/ID register is **enabled**, allowing it to store the fetched instruction and other data.
  - The PC register can be **updated**, typically incrementing to PC+1 for the next sequential instruction fetch.



However, if the stall signal is **high** (interpreted as **low** due to the inversion):

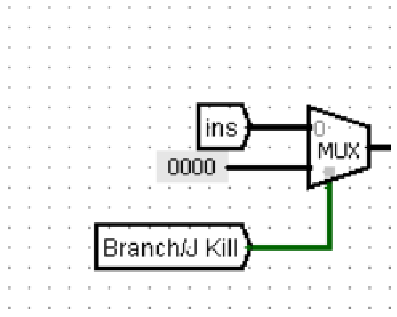
- Both the IF/ID register and the PC register are **disabled**. This prevents any changes to the stored data, effectively pausing the pipeline and maintaining a consistent state during the stall.





Handling Branch and Jump Instructions: The "Zero Instruction" Trick

Beyond the standard inputs, the IF/ID register interacts with a special signal called "Branch/Jump Kill" generated by the PC control unit. This signal plays a crucial role in handling branch and jump instructions:



- **Branch/Jump Kill = 0:** This is the typical scenario for most instructions. The MUX selects the actual instruction received from memory, allowing the ID stage to decode it normally.
- **Branch/Jump Kill = 1:** When a branch or jump instruction is encountered, this signal forces the IF/ID register to store a special "zero instruction." This essentially instructs the processor to perform a no-operation (NOP) for the next cycle.

But why the "zero instruction" trick? Branch and jump instructions modify the PC, directing the processor to fetch an instruction from a different location, not the next one in sequence. By inserting a NOP, the pipeline avoids processing an unintended instruction that would have been fetched sequentially. Once the PC is updated based on the branch or jump, the correct instruction can be fetched in the subsequent cycle.

Dual Role of the Branch/Jump Kill Signal

1. **Selecting the Instruction:** As mentioned previously, it acts as a MUX to choose between the actual instruction and the "zero instruction" for the IF/ID register.
2. **Selecting the Next PC:** This signal also controls another MUX that feeds the PC register. Here's the breakdown:

    - **Branch/Jump Kill = 0 (Normal Sequence):** The MUX selects PC+1 as the input for the PC register, ensuring sequential instruction fetching.
    - **Branch/Jump Kill = 1 (Branch/Jump):** The MUX bypasses PC+1 and selects a different "next PC" value generated by the PC control unit based on the branch or jump instruction. This updated PC value directs the fetch stage to the correct location for the next instruction.

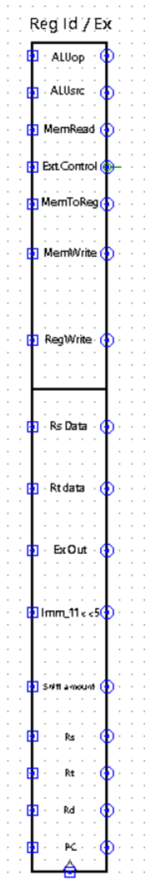
3.1.2. Id/Ex Register

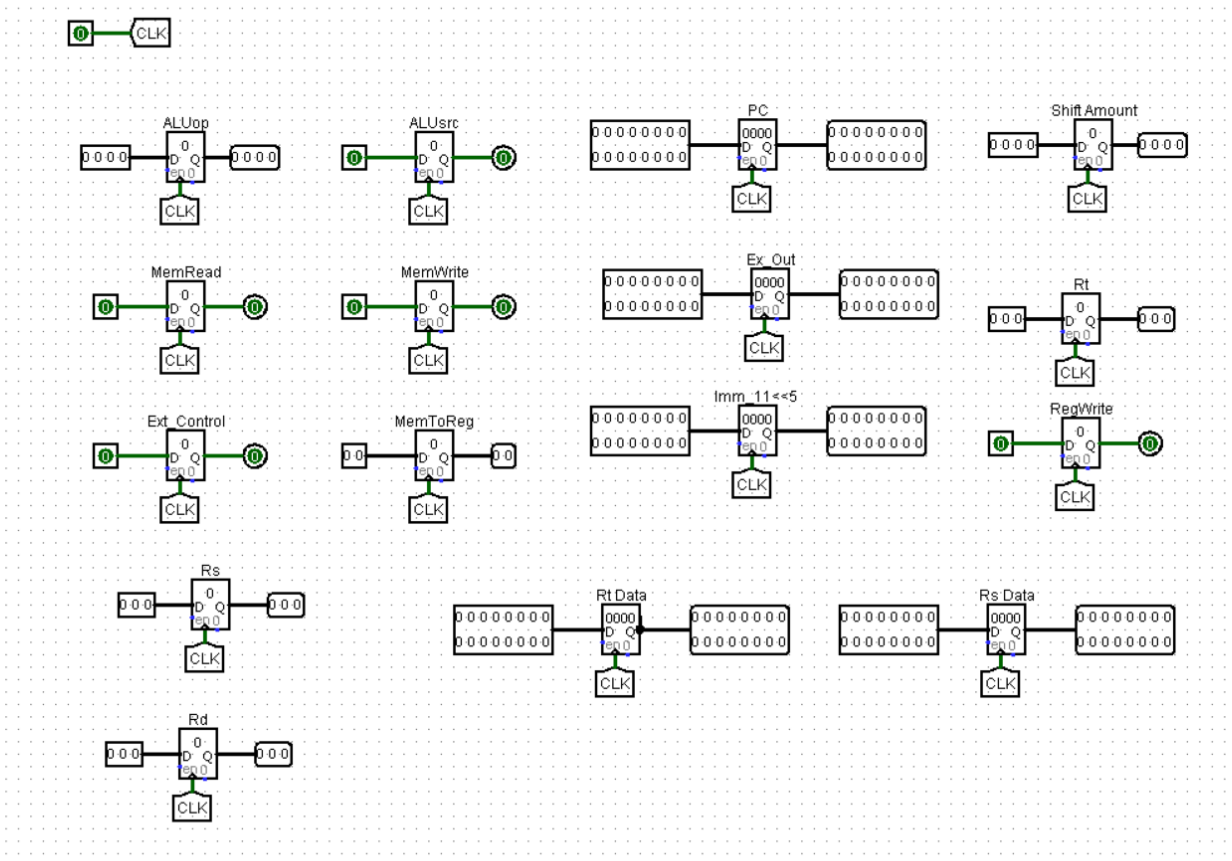
The ID/EX register plays a crucial role in enabling pipelined execution. It acts as a bridge between the Instruction Decode (ID) stage and the Execution (EX) stage, holding onto critical information extracted from instructions during the ID stage.

What's in the ID/EX Register

Here's a breakdown of the key data elements stored in the ID/EX register:

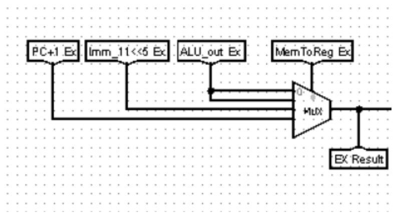
- **Control Signals:** These signals, generated by the control unit based on the decoded instruction, provide specific instructions to the execution unit (often the ALU) on how to process the data. Some important control signals include:
    - **ALUop:** This signal specifies the operation to be performed by the ALU (e.g., addition, subtraction, logical AND, etc.).
    - **ALUsrc:** This signal determines whether the second operand for the ALU comes from a register or an immediate value.
    - **MemRead:** This signal indicates whether the instruction requires reading data from memory.
    - **Ext.Control:** This signal controls how the sign of an immediate value is extended if necessary.
    - **MemToReg:** This signal determines whether the data written back to a register comes from memory or the ALU output.
    - **MemWrite:** This signal indicates whether the instruction requires writing data to memory.
    - **RegWrite:** This signal controls whether the result of the operation needs to be written back to a register file.
- **Data Fields:** These fields hold the actual data required for the execution stage:
    - **Rs Data:** This field stores the data from the source register (identified during decoding).
    - **Rt Data:** This field stores the data from the destination register (identified during decoding).
    - **Extender Out:** This field holds the immediate value after any necessary sign extension.
    - **Imm 11 >> 5 :** This field stores the immediate value shifted by 5 which will be used in (LUI) instruction.
    - **Shift Amount:** This field holds the number of bits for any required shifting operation.
- **Register Identifiers:** The ID/EX register also stores the register numbers involved in the instruction:
    - **Rs:** This field holds the source register number.
    - **Rt:** This field holds the destination register number.
    - **Rd:** This field holds the register number where the result will be written back (if applicable).
    - **PC:** This field holds the Program Counter value at the time the instruction was fetched.


Reg Id / Ex
ALUop
ALUsrc
MemRead
Ext Control
MemToReg
MemWrite
RegWrite
Rs Data
Rt data
Ex Out
Imm_11<<5
Shift amount
Rs
Rt
Rd
PC

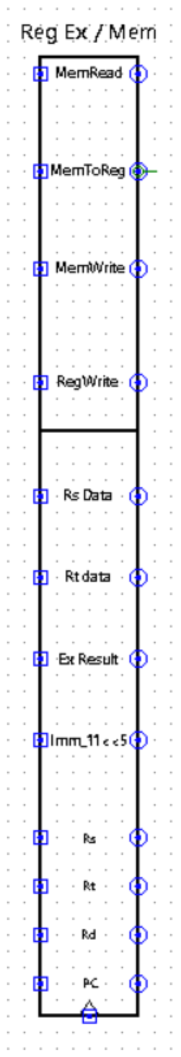

CLK
ALUop
ALUsrc
PC
Shift Amount
MemRead
MemWrite
Ex_Out
Rt
Ext_Control
MemToReg
Imm_11<<5
RegWrite
Rs
Rt Data
Rs Data
Rd

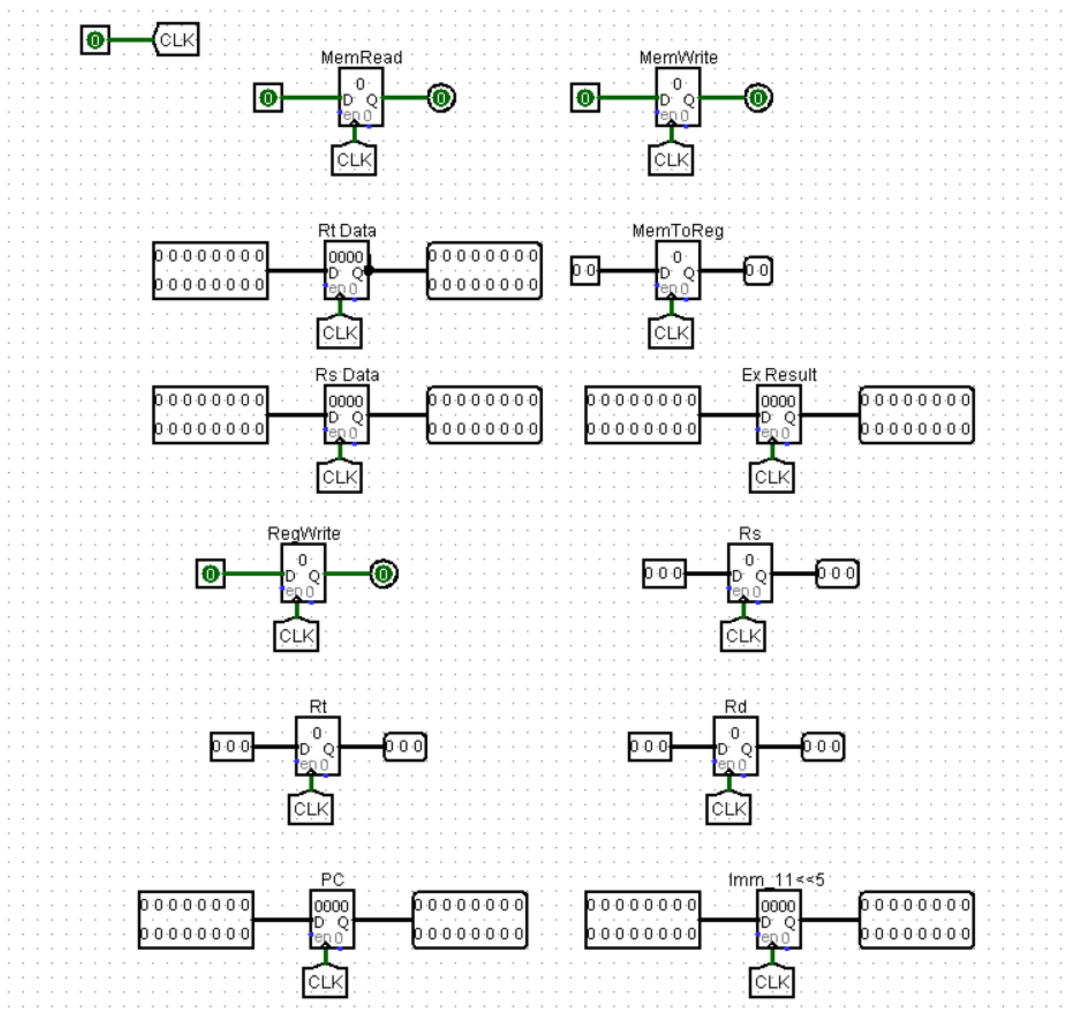
3.1.3. Ex/Mem Register

The Ex/Mem register acts as a crucial bridge between the Execution (EX) and Memory Access (MEM) stages of the processor pipeline. It holds the essential information needed for the MEM stage, including:



- **Execution Result:** This is the main output from the EX stage, such as the result of an ALU operation, a data forwarding operation, or a constant from an LUI instruction (Load Upper Immediate) or PC + 1 (JAL Instruction).
- **Control Signals:** These signals, generated by the control unit, instruct the MEM stage on how to handle the data (e.g., access memory, write to a register).
- **Additional Data (Optional):** Depending on the instruction, the Ex/Mem register might hold extra data like operands needed for memory access or register writes.





3.1.4. Mem/Wb Register

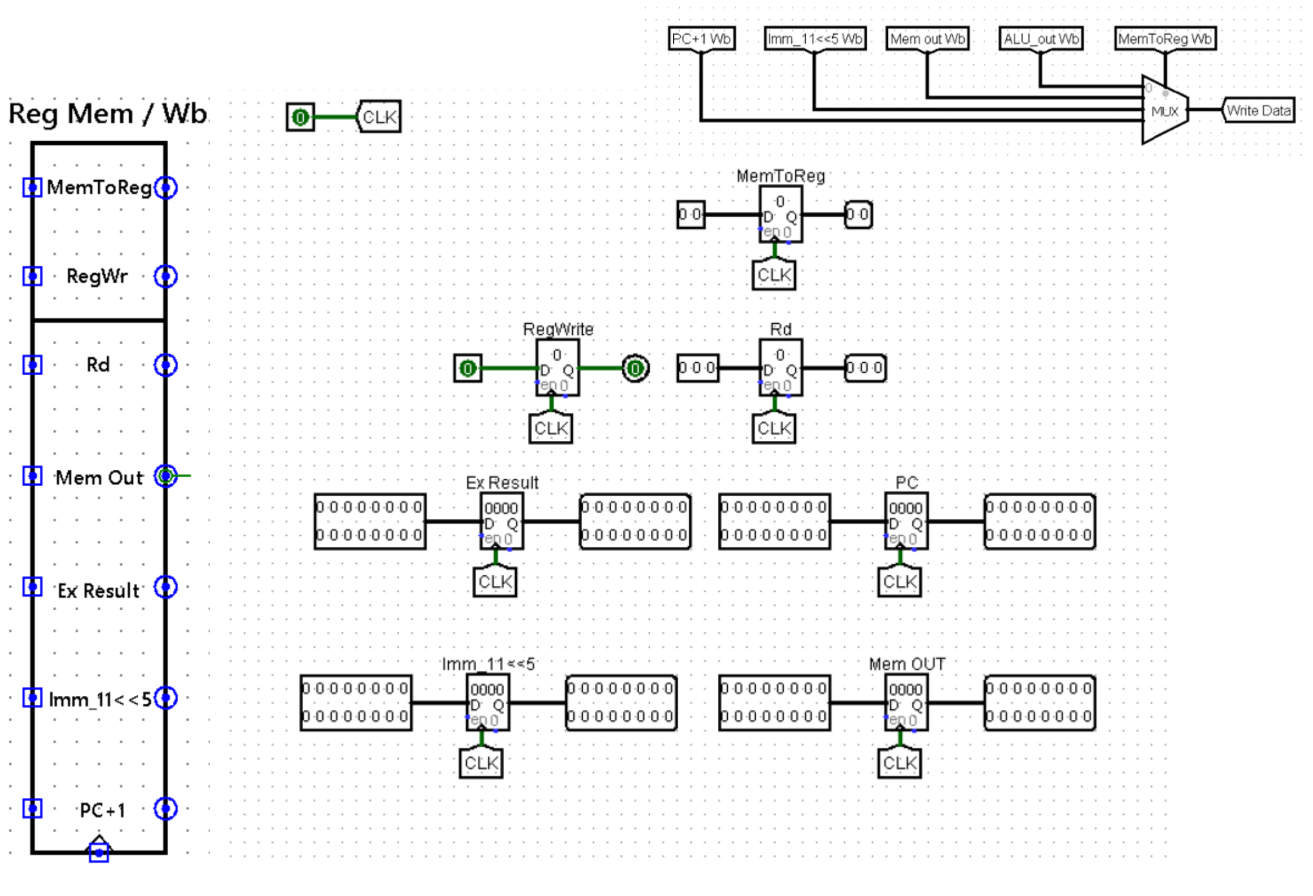
Absolutely, let's revise the Mem/WB register description to remove the "gatekeeper" metaphor:

The Mem/WB Register: Delivering the Final Results

The Mem/WB register plays a crucial role in the final stage of the processor pipeline, acting as a bridge between the Memory Access (MEM) stage and the Write Back (WB) stage. Its primary function is to ensure the correct data gets written back to the processor's registers.

Here's how it accomplishes this:

- **Data Storage:** The Mem/WB register temporarily stores the data that might be written back to registers. This data can come from two sources:
    - **Memory Value:** If the instruction required reading from memory, the retrieved value is stored here.
    - **ALU Output:** For instructions that don't involve memory access, the ALU output from the Execution stage is held in the Mem/WB register.
- **Control Signals for Write Back:** Two critical control signals from the control unit govern the write-back process:
    - **MemToReg:** This signal determines the source of the data written back.
    - **RegWrite:** This signal acts like an on/off switch for writing data back to registers. When set to 1, the data in the Wb stage is written back to the designated register. Otherwise, no write-back occurs.



3.2. Forward Unit

Imagine you're working on a task list. Sometimes, you need information that someone else is about to update. In a processor pipeline, instructions can face a similar problem. An instruction might need a value from a register that another instruction is just about to change.

The Forwarding Unit is here to the rescue! It helps instructions get the correct value, even if it's not yet written to the register file. Here's what it does:

- **Looks for Trouble:** The Forwarding Unit checks instructions as they flow through the pipeline. It keeps an eye out for situations where an instruction (let's call it instruction B) needs a value from a register that another instruction (instruction A) is about to update.
- **Forwarding the Latest Info:** If the Forwarding Unit spots this trouble, it steps in. **Since it operates in the Execute (EX) stage,** it can't directly access data from memory (which is a later stage). However, it can still ensure B gets the correct value by forwarding from a later stage:
    - **The data from later stages (like Mem/WB register):** If instruction A has already finished execution (either in the Memory Access (MEM) or Write Back (WB) stage) but hasn't written back to the register file yet, the Forwarding Unit can grab the value stored in the Ex/Mem register or Mem/WB register and forward it to instruction B in the EX stage.

The Forwarding Unit needs to generate two control signals, one for each register it reads from (source register, Rs, and target register, Rt) in the current instruction (B). These signals will ultimately control a multiplexer (mux) that picks the right data source for each register.

While we can build a single forwarding unit, it's common to have separate circuits for Rs and Rt. This can make things faster, especially in complex processors.

Finding Forwarding Opportunities:

Since the Forwarding Unit works in the EX stage, it can forward data by using info from later stages:

 Memory (MEM) Stage: The Forwarding Unit gets details from the MEM stage, like:

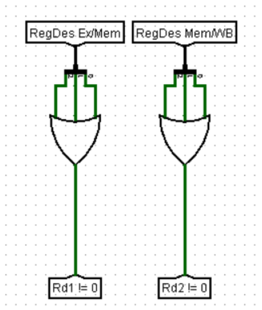
- **Register Destination (RegDest):** This tells the Forwarding Unit which register instruction A wants to write to.
- **Register Write (RegWrite):** This signal tells the Forwarding Unit if instruction A actually wrote to a register (RegWrite = 1) or not (RegWrite = 0).

 Write Back (WB) Stage: Similarly, the Forwarding Unit gets info from the WB stage, including:

- **Register Destination (RegDest):** Same as MEM, this tells us which register a previous instruction wants to write to.
- **Register Write (RegWrite):** Same as before, this signal confirms if the previous instruction wrote to a register.

Conditions for Forwarding:

Before forwarding, the Forwarding Unit checks for specific conditions to ensure it forwards the most relevant value:



1. **Not a Dummy Register:** The register instruction A wants to write to (RegDest from MEM or WB) can't be zero. A zero RegDest usually means an implicit register (like the program counter) and doesn't need forwarding.
2. **Successful Write:** The previous instruction must have successfully written a new value to a register. This is confirmed by a RegWrite signal of 1 (from both MEM and WB stages).

Based on these conditions, the Forwarding Unit generates two flags:

- **Mem Forward Flag:** This flag is set to 1 if forwarding from the MEM stage is a good idea.
- **WB Forward Flag:** This flag is set to 1 if forwarding from the WB stage is a good idea.

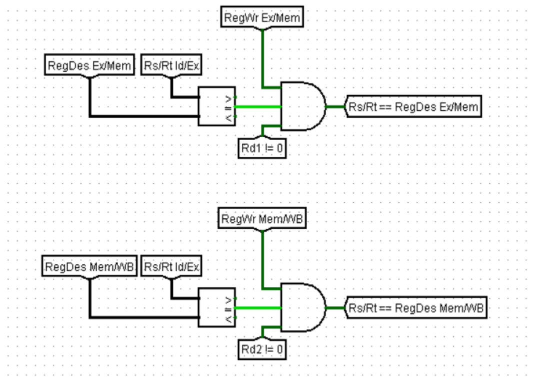
Choosing the Forwarding Source:

Now that we understand the conditions for forwarding, let's see how the Forwarding Unit determines the most suitable source for each operand (Rs and Rt) in the current instruction (B).

Forwarding Flags:

Based on the information received from the MEM and WB stages, the Forwarding Unit generates two control signals, often referred to as flags:

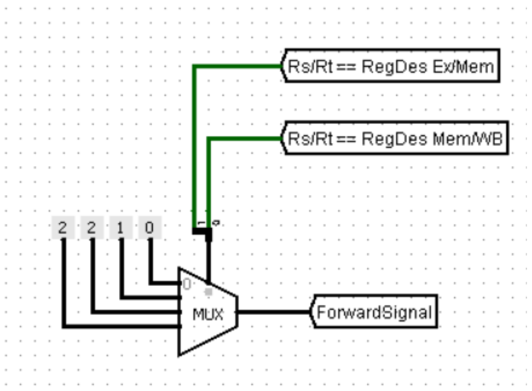
- **Mem Forward Flag:** This flag is set to 1 if a data dependency exists between the current instruction (B) and a previous instruction (A). Specifically, this occurs when:
  - **Register Source Match (Rs/Rt in B):** One of the source registers (Rs or Rt) in instruction B matches the register destination (RegDest) from the MEM stage. This indicates that instruction A is about to write to the same register that instruction B needs to read.
  - **Successful Write (RegWrite = 1):** The RegWrite signal from the MEM stage is 1.



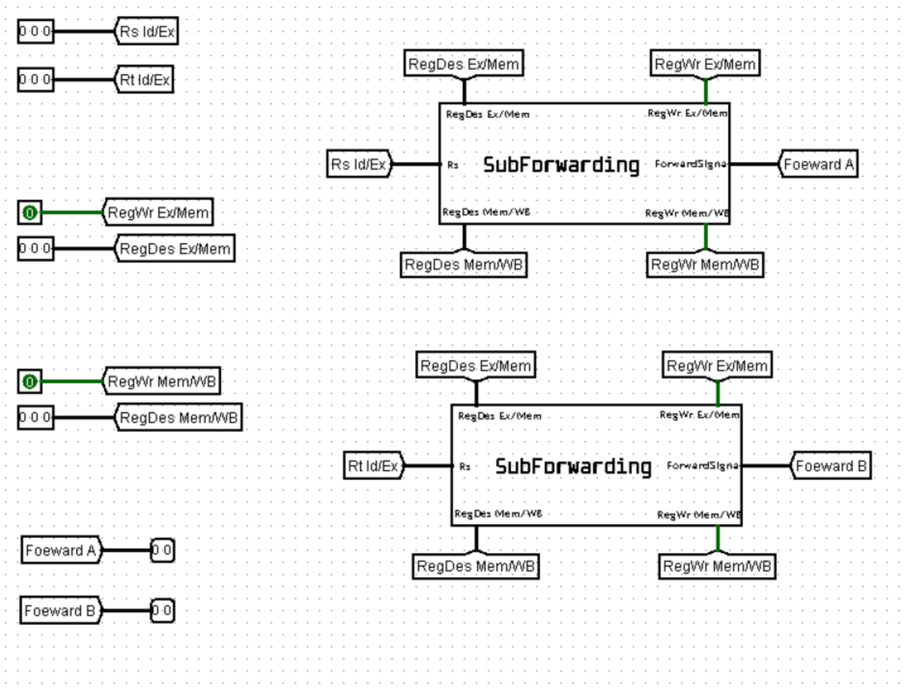
- **WB Forward Flag:** This flag is set to 1 under similar conditions, but using information from the WB stage:
  - **Register Source Match (Rs/Rt in B):** One of the source registers (Rs or Rt) in instruction B matches the register destination (RegDest) from the WB stage.
  - **Successful Write (RegWrite = 1):** The RegWrite signal from the WB stage is 1.

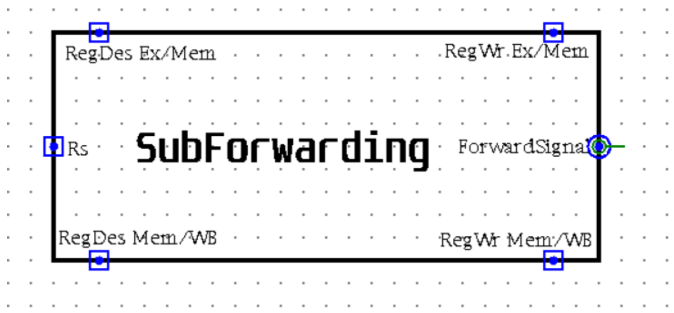
A multiplexer (mux) with constant inputs is used to select the final data source for each operand (Rs and Rt) based on the forwarding flags:

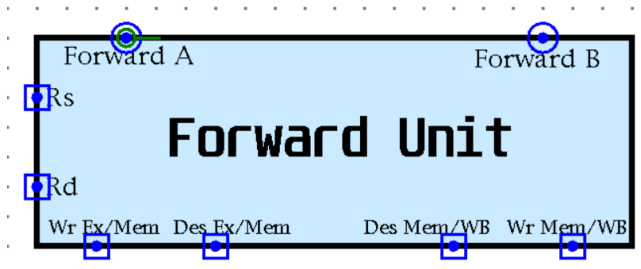
- **Selector Logic:** The mux selector is created by combining the Mem Forward Flag and WB Forward Flag into a 2-bit selection signal.
- **Constant Inputs:** The mux has several constant inputs representing potential data sources:
    - **Forward Signal 0:** This represents the original value from the register file (no forwarding).
    - **Forward Signal 1:** This represents the value forwarded from the WB stage (if the WB Forward Flag is high).
    - **Forward Signal 2:** This represents the value forwarded from the MEM stage (if the Mem Forward Flag is high).



And finally it's our unit:







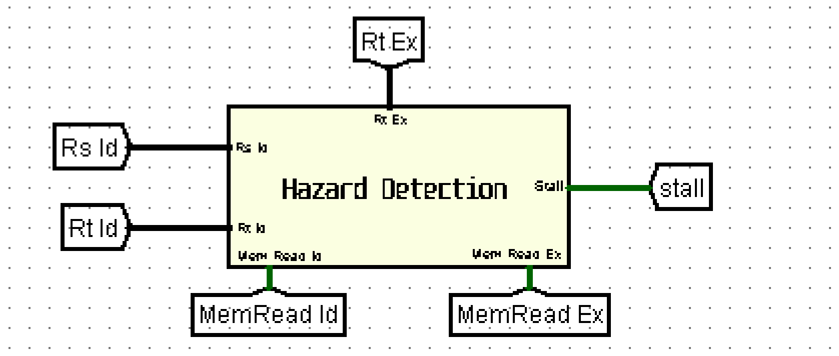
3.3. Hazard Detection Unit

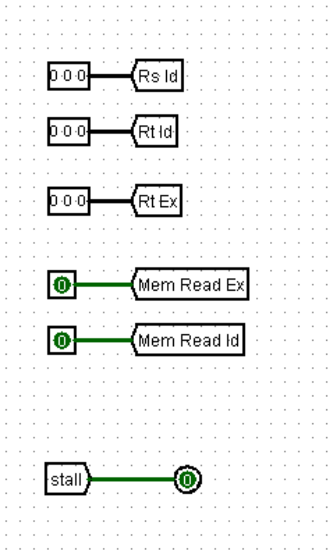
The hazard detection unit plays a critical role in ensuring smooth operation of the pipelined MIPS processor. It analyzes the instruction stream and identifies data hazards (RAW)

Data Hazards: When an instruction attempts to use data (read or write) that hasn't been produced by a previous instruction yet.

The hazard detection unit utilizes the information about instructions in the pipeline stages to detect RAW. Based on the type of hazard identified, the hazard detection unit takes corrective actions such as:

Stalling the pipeline: Pausing the execution of certain instructions until the data is loaded.





The hazard detection unit takes these inputs to generate the stall signal:

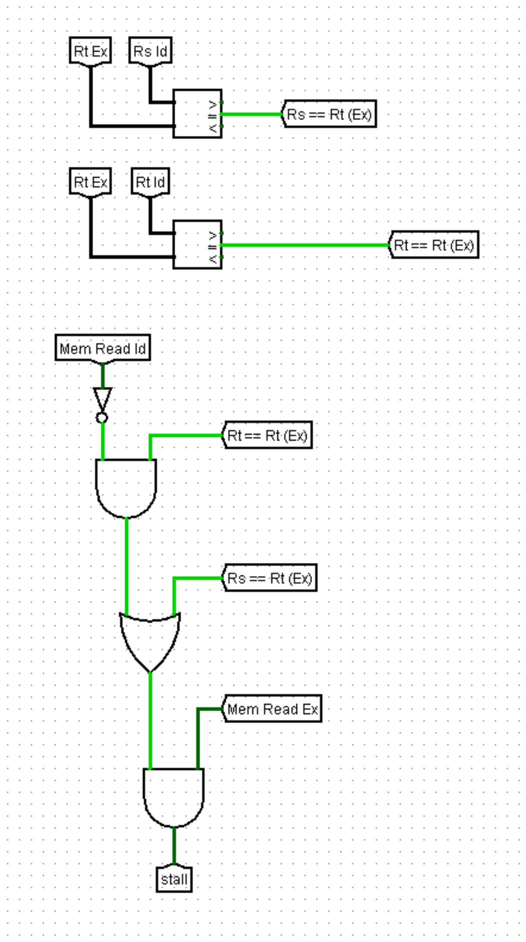
- Rs Id
- Rt Id
- Rt Ex
- Mem Read Ex
- Mem Read Id

Stall Signal:

The stall signal is generated when the instruction in the Ex stage is a load instruction and the data which will be loaded is used in the ID stage.

It is generated by the following circuit which checks for some conditions:

1. Mem Read Ex == 1: The instruction in the Ex stage is a load instruction
2. The data that will be used in the is the data that needs to be loaded first, we do that by comparing:
   2.1. Rs == Rt(Ex)
   2.2. Rt == Rt(Ex)
3. This condition (Rt == Rt(Ex)) only applies if the current instruction is not a load instruction.



3.4. Branch/Jump Forward Unit

To handle potential data dependencies, the branch/jump forward unit resides within the decode stage. This strategic placement allows it to react promptly to forwarding requests from subsequent stages, such as execution, memory, or write back. This is particularly important when a branch instruction is encountered shortly after a register is written to in a preceding instruction.

The branch/jump forward unit is comprised of four subcircuits, each specializing in a specific forwarding task. The first subcircuit determines if forwarding is necessary from the execution stage. Similarly, the second and third subcircuits handle potential forwarding requirements from the memory and writeback stages, respectively. Finally, the fourth subcircuit acts as a central processing unit, analyzing the outputs of the other three and calculating the final forwarding data based on priority (execution < memory < writeback).

3.4.1. Input signals

1. **Branch:** This input identifies if the current instruction is a branch or jump instruction. This is achieved by analyzing the least significant two bits of the **PCsrc** signal (Program Counter source). These bits typically encode different actions for the program counter update, and specific values in the lower two bits indicate a branch or jump. The use of an OR gate suggests that any combination in those two bits triggers a "branch" signal.
2. **ID Rs:** (Source Register in Decode Stage) - This input provides the register number the instruction in the decode stage intends to read from (source operand).
3. **ID Rt:** (Target Register in Decode Stage) - Similarly, this input specifies the register number the decode stage instruction plans to write to (target operand).
4. **EX Rd, MEM Rd, WB Rd (3 separate inputs):** These three inputs represent the destination registers from the execution (EX), memory (MEM), and writeback (WB) stages, respectively. Each input provides the register number that the corresponding stage intends to write to after processing the instruction.
5. **EX RegWrite, MEM RegWrite, WB RegWrite (3 separate inputs):** These three inputs are the register write enable signals from the execution (EX), memory (MEM), and writeback (WB) stages, respectively. Each signal indicates whether the corresponding stage plans to write a value to a register. A value of 1 typically signifies an intended write operation, while 0 indicates no write.

By examining these nine individual inputs, the subcircuits within the branch/jump forward unit can determine if there are any potential data dependencies. Here's how:

1. The unit compares the source and target registers from the decoded instruction (inputs 2 and 3) with the destination registers from the execution, memory, and writeback stages (inputs EX Rd, MEM Rd, WB Rd).
2. Simultaneously, it checks the corresponding RegWrite signals for each stage (inputs EX RegWrite, MEM RegWrite, WB RegWrite) and the branch signal which ensures that the instruction is branch or jump.

If a match is found between a decode stage register (source or target) and a destination register from a later stage (EX, MEM, or WB), and the corresponding RegWrite signal for that later stage is asserted (indicating a write), a data dependency exists. This is because the instruction in the decode stage relies on a value that hasn't been written back to the register file yet.

In such scenarios, the branch/jump forward unit initiates forwarding by providing the necessary data directly from the earlier stage (where it's available) instead of waiting for the writeback to complete. This ensures the instruction in the decode stage has the correct operand value, preventing errors and maintaining pipeline efficiency.

3.4.2. Output signals

Each subcircuit generates two 1-bit outputs:

- **ForwardC:** This signal indicates forwarding to the source register of the decode stage instruction. A value of 1 signifies forwarding from the assigned stage (execution, memory, or writeback), while 0 indicates no forwarding needed from that stage.
- **ForwardD:** Similarly, this signal represents forwarding to the target register of the decode stage instruction. A value of 1 signifies forwarding from the assigned stage, and 0 indicates no forwarding needed from that stage.

The Calc FC/FD subcircuit acts as a central arbiter for forwarding decisions. It receives the three ForwardC outputs (one from each subcircuit) and the three ForwardD outputs (one from each subcircuit) as inputs. These six signals essentially provide a voting scheme for forwarding requirements.

Priority-Based Forwarding Logic:

The Calc FC/FD subcircuit employs a combinational logic circuit that incorporates a priority scheme, typically following execution > memory > writeback. This prioritizes forwarding results from the stage with the most recent data (execution) in case of conflicts.

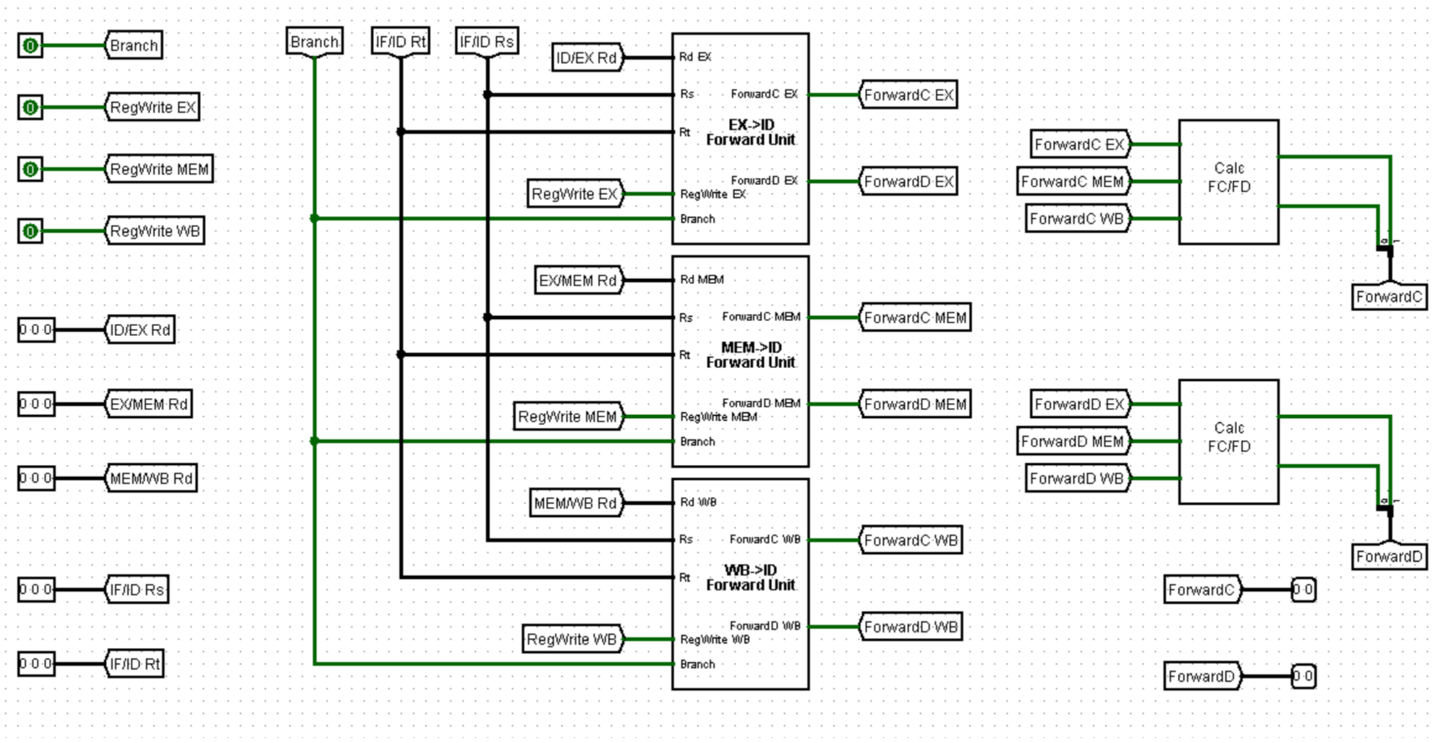
The logic circuit analyzes the six input signals (three ForwardC and three ForwardD) and their corresponding priorities. Based on this analysis, it produces a final 2-bit ForwardC output and a final 2-bit ForwardD output.

These final 4-bit outputs (2-bit ForwardC and 2-bit ForwardD), along with potentially additional control signals, direct the processor on what data to forward and from which stage. This ensures smooth pipeline execution and avoids data hazards.

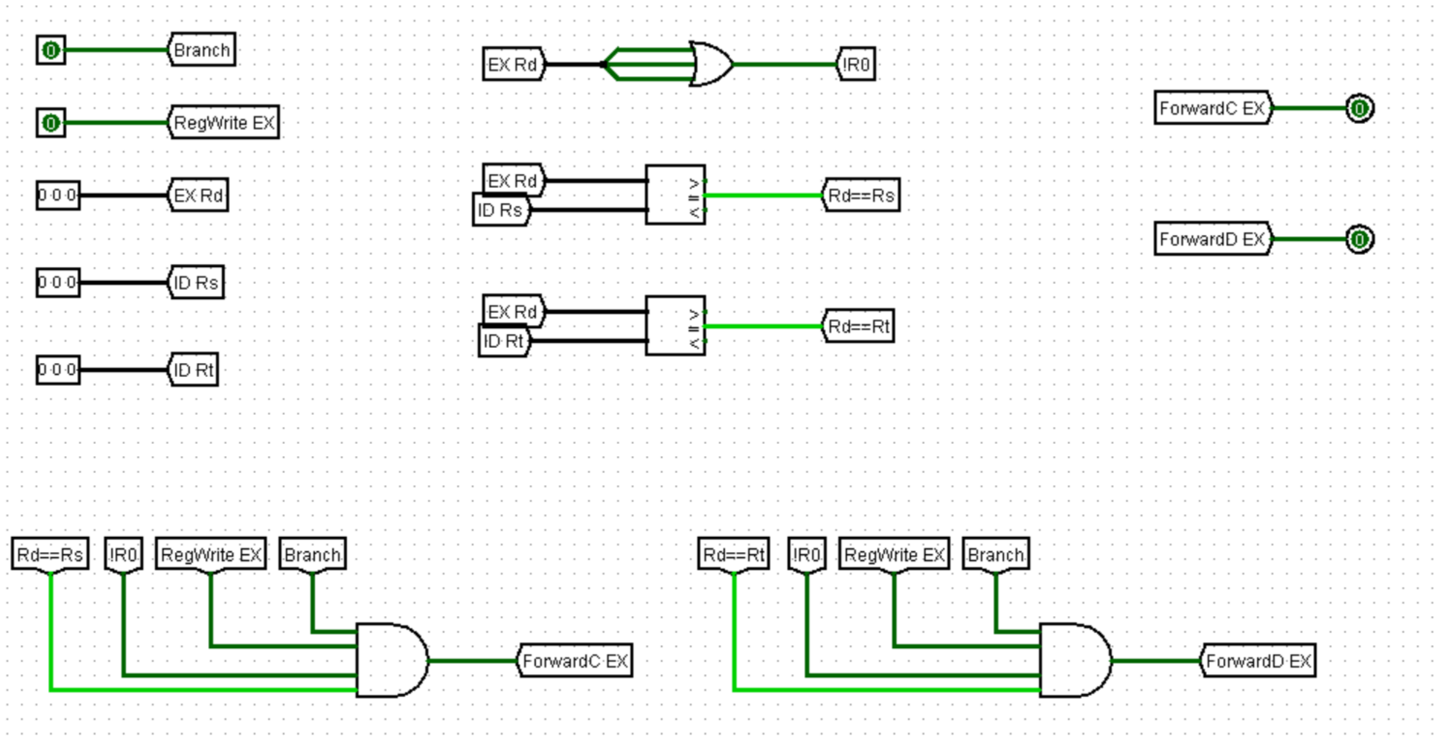
By combining the subcircuit outputs with the priority-based voting and logic within the Calc FC/FD subcircuit, the branch/jump forward unit efficiently manages data dependencies within the processor pipeline.

3.4.3. Internal Architecture

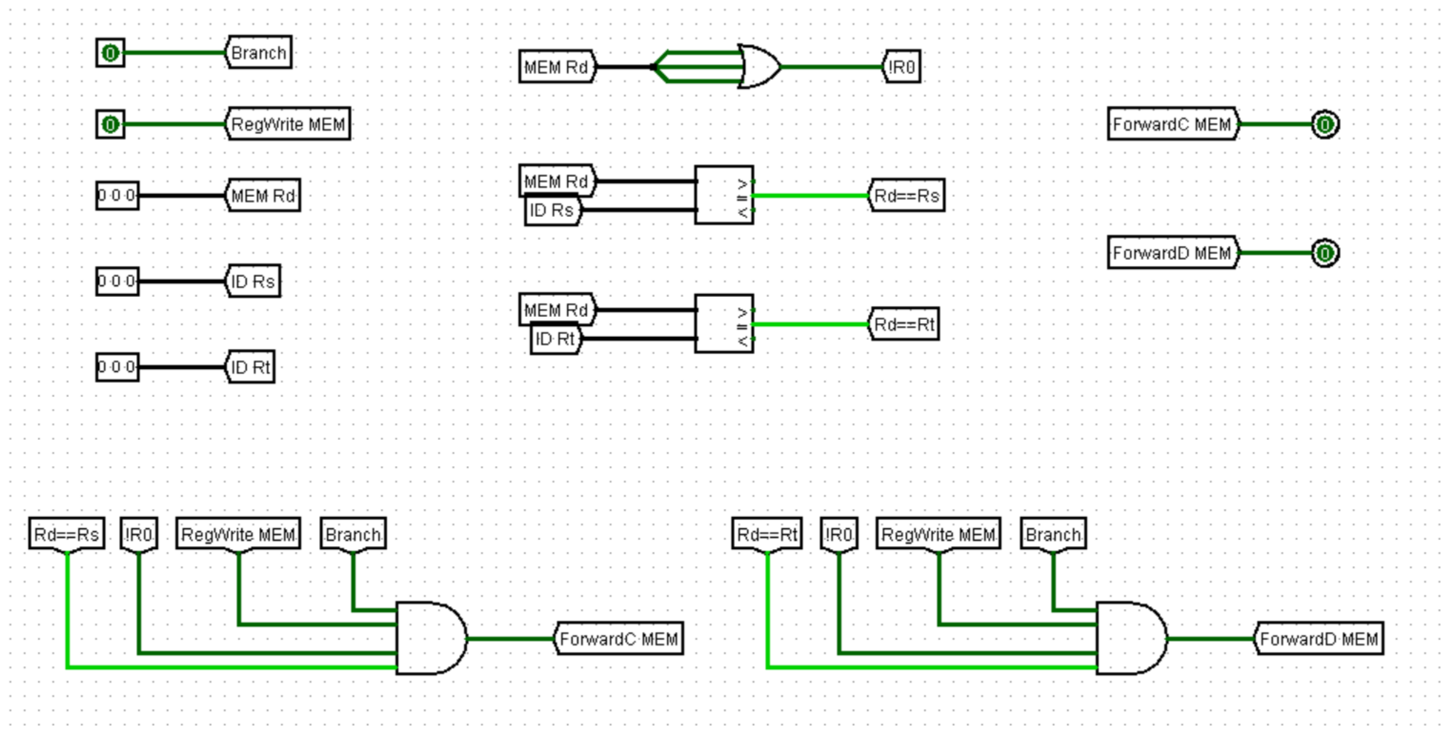
4.3.1. The main circuit



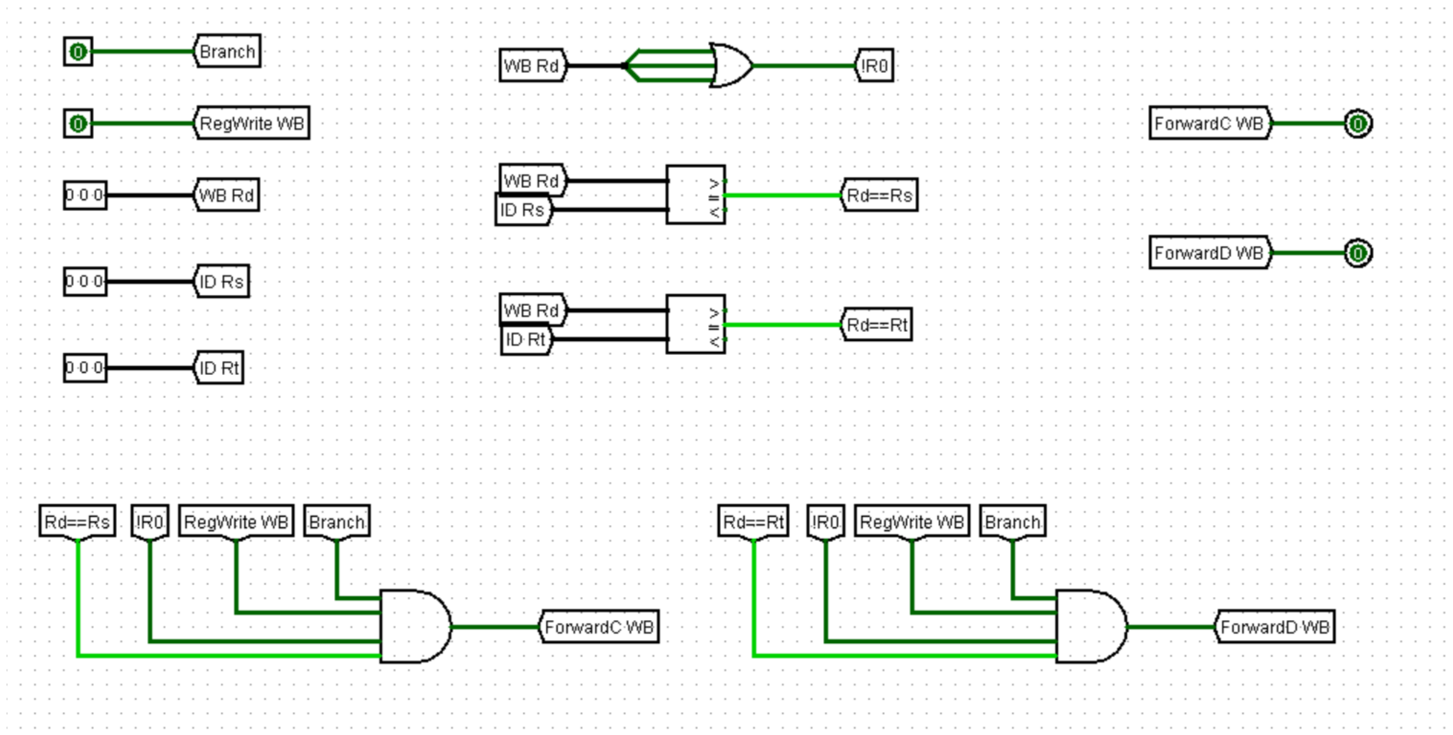
4.3.2. EX->ID Forward Unit



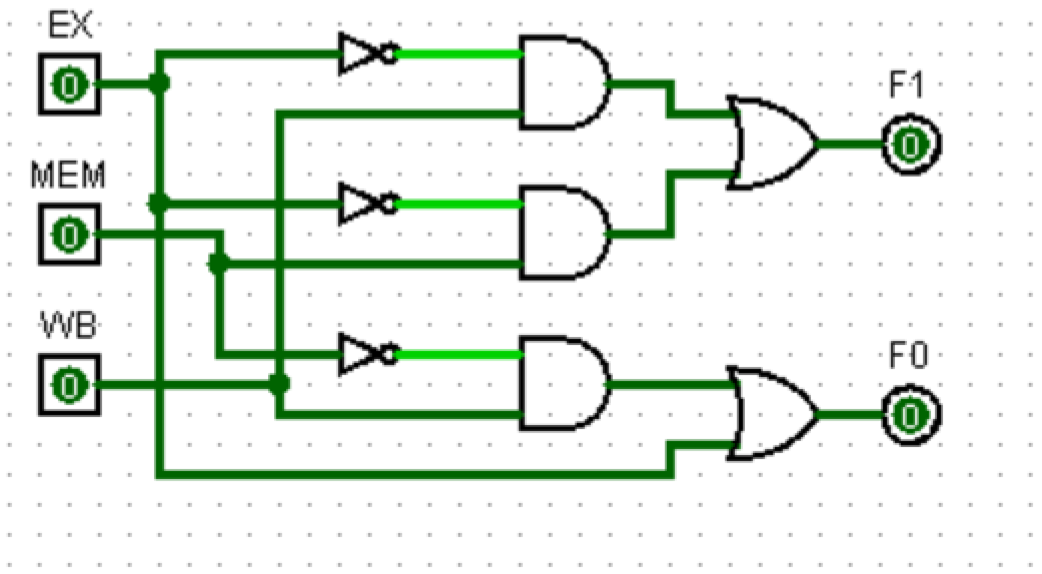
4.3.3. MEM->ID Forward Unit



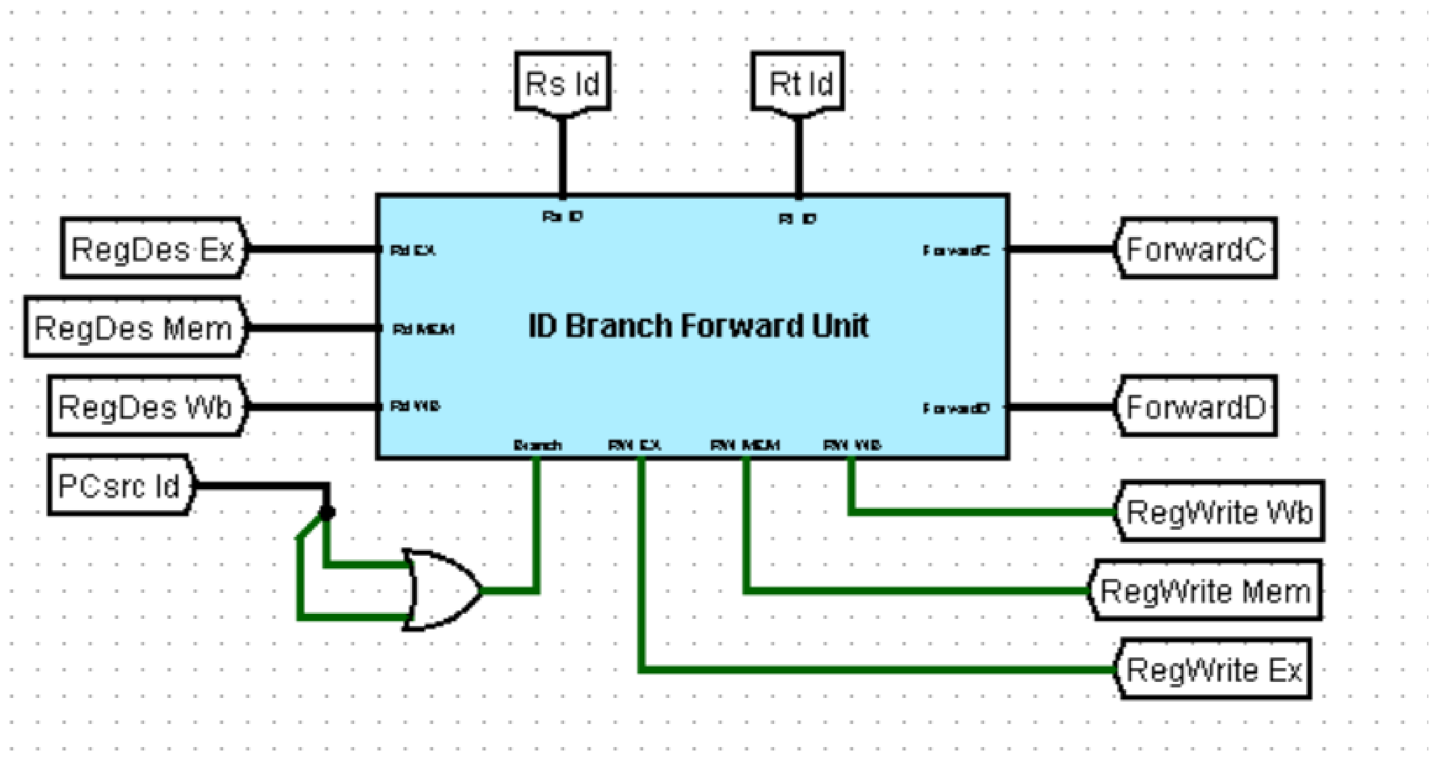
4.3.4. WB->ID Forward Unit



4.3.5. Calc FC/FD



4.3.6. Branch/Jump Forward Unit in Datapath



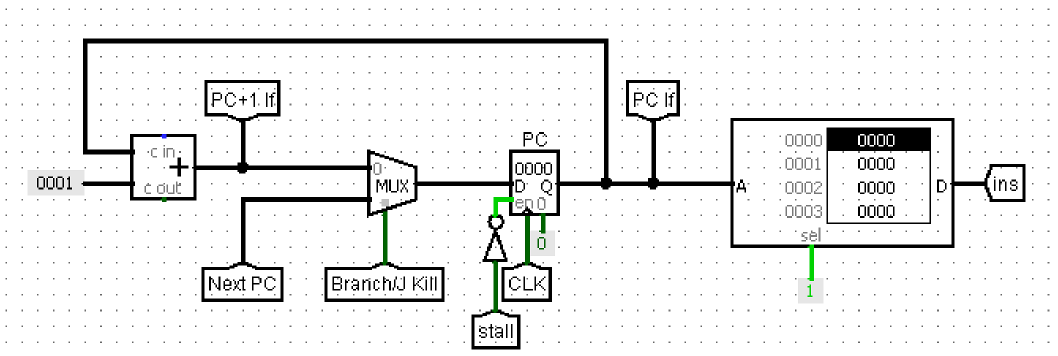
3.5. Pipeline PC Control Unit

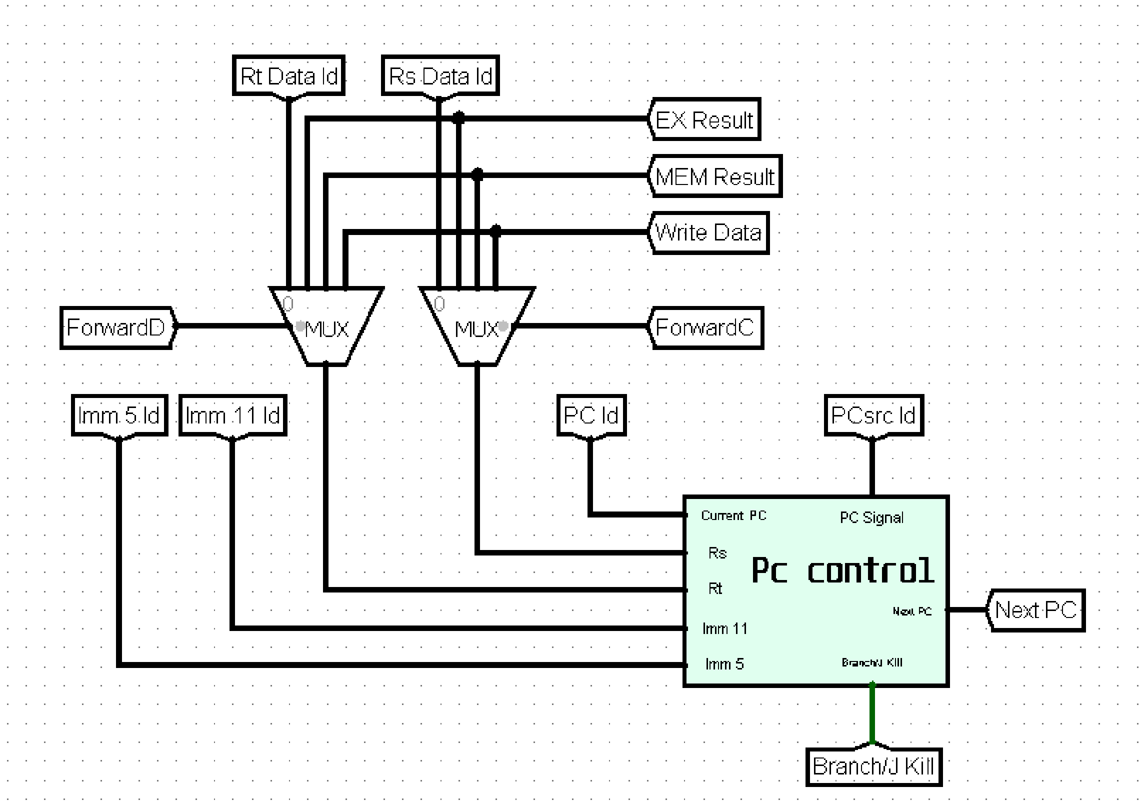
The single cycle PC control will not work in the pipeline circuit as the pc control is located in the ID (Instruction Decoding) stage, so it would take two cycles for the new pc value to be generated.

This is fixed by splitting the responsibility of the pc into two.

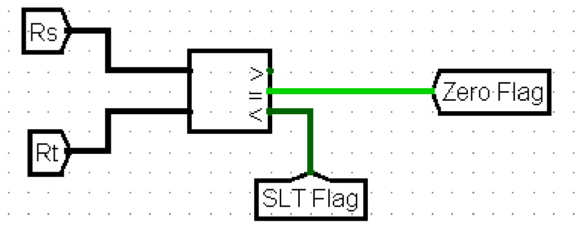
Firstly, the normal instructions which the new pc value is PC + 1 this is done is the IF stage. We added an adder before the pc register to auto increment the pc value by one each cycle.

Secondly, the branch, jump, jump register and jal instruction are handled inside the pc control.



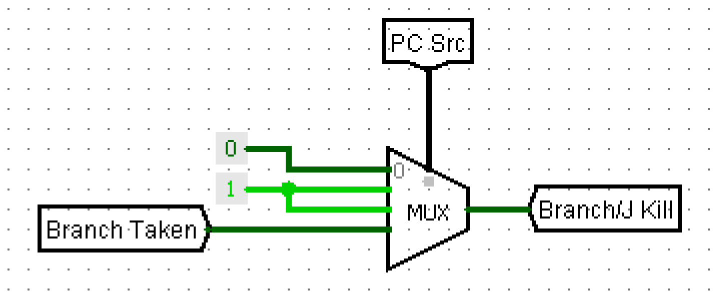


In this version of the pc control we can't take the zero and SLT flags as input, because they are generated by the ALU in the EX stage. So we take the values of Rs and Rt and generate the signals using a comparator.

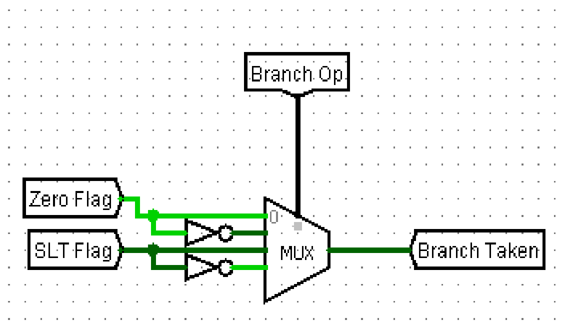


We need to generate a signal that would indicates if the new PC value will be the value incremented by the adder of the value from the pc control which will be (Branch/J Kill)

- **Branch/J Kill Signal:**
  - In the case of J, Jr and Jal instructions the value is always 1.
  - In The branch instructions the value is can be 1/0 based of the Branch Taken signal.



- **Branch Taken Signal In different cases:**
  - **Beq**: value of Zero flag
  - **Bne**: value of Zero flag inverted
  - **Blt**: value of SLT flag
  - **Bge**: value of SLT flag inverted



4. Testing and verification

4.1 Testing (Single Cycle)

In this phase, we implemented sum test programs to functionally validate the various sub-circuits constituting the overall circuit design.

1.1. Test No.1 (Sum Array program)

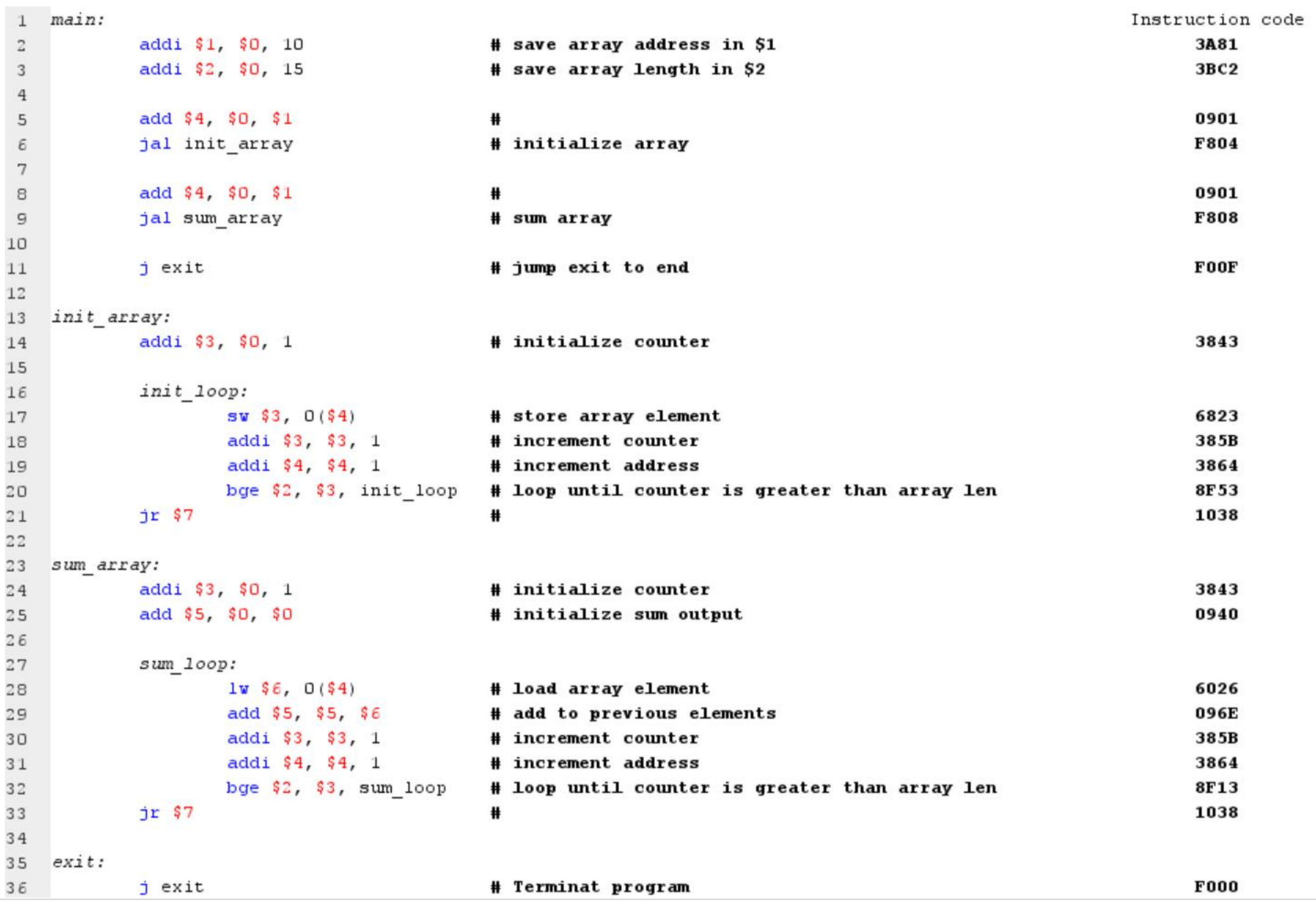
This program initializes an array and then calculates the sum of its elements. It achieves this through two functions:

- init_array: This function takes the array itself and its length as input (likely stored in registers) and fills the array with values from 1 to the length.
- sum_array: This function takes the array and its length as input and calculates the sum of all the elements in the array, returning the final sum.

The program utilizes registers to store temporary values during execution:

- $1: Stores the base address of the array in memory.
- $2: Stores the length of the array.
- $4: Likely used as a general-purpose argument register to pass data to the functions.
- $5: Stores the final sum returned by the sum_array function.

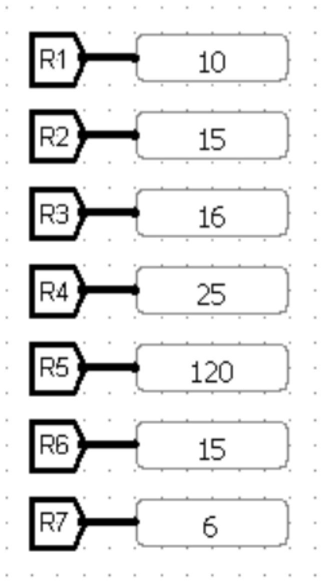
1.1.1. Assembly code and Instruction code



```
main:                                                                                   Instruction code
        addi $1, $0, 10             # save array address in $1                              3A81
        addi $2, $0, 15             # save array length in $2                               3BC2

        add $4, $0, $1              #                                                       0901
        jal init_array              # initialize array                                      F804

        add $4, $0, $1              #                                                       0901
        jal sum_array               # sum array                                             F808

        j exit                      # jump exit to end                                      F00F

init_array:
        addi $3, $0, 1              # initialize counter                                    3843

        init_loop:
                sw $3, 0($4)        # store array element                                   6823
                addi $3, $3, 1      # increment counter                                     385B
                addi $4, $4, 1      # increment address                                     3864
                bge $2, $3, init_loop  # loop until counter is greater than array len       8F53
        jr $7                       #                                                       1038

sum_array:
        addi $3, $0, 1              # initialize counter                                    3843
        add $5, $0, $0              # initialize sum output                                 0940

        sum_loop:
                lw $6, 0($4)        # load array element                                    6026
                add $5, $5, $6      # add to previous elements                              096E
                addi $3, $3, 1      # increment counter                                     385B
                addi $4, $4, 1      # increment address                                     3864
                bge $2, $3, sum_loop   # loop until counter is greater than array len       8F13
        jr $7                       #                                                       1038

exit:
        j exit                      # Terminat program                                      F000
```

1.1.2. Registers Expected vs Actual values

Registers	Expected	Actual
R1	10	10
R2	15	15
R3	16	16
R4	25	25
R5	120	120
R6	15	15
R7	6	6



Test Pass

1.1.3. Memory Expected vs Actual values

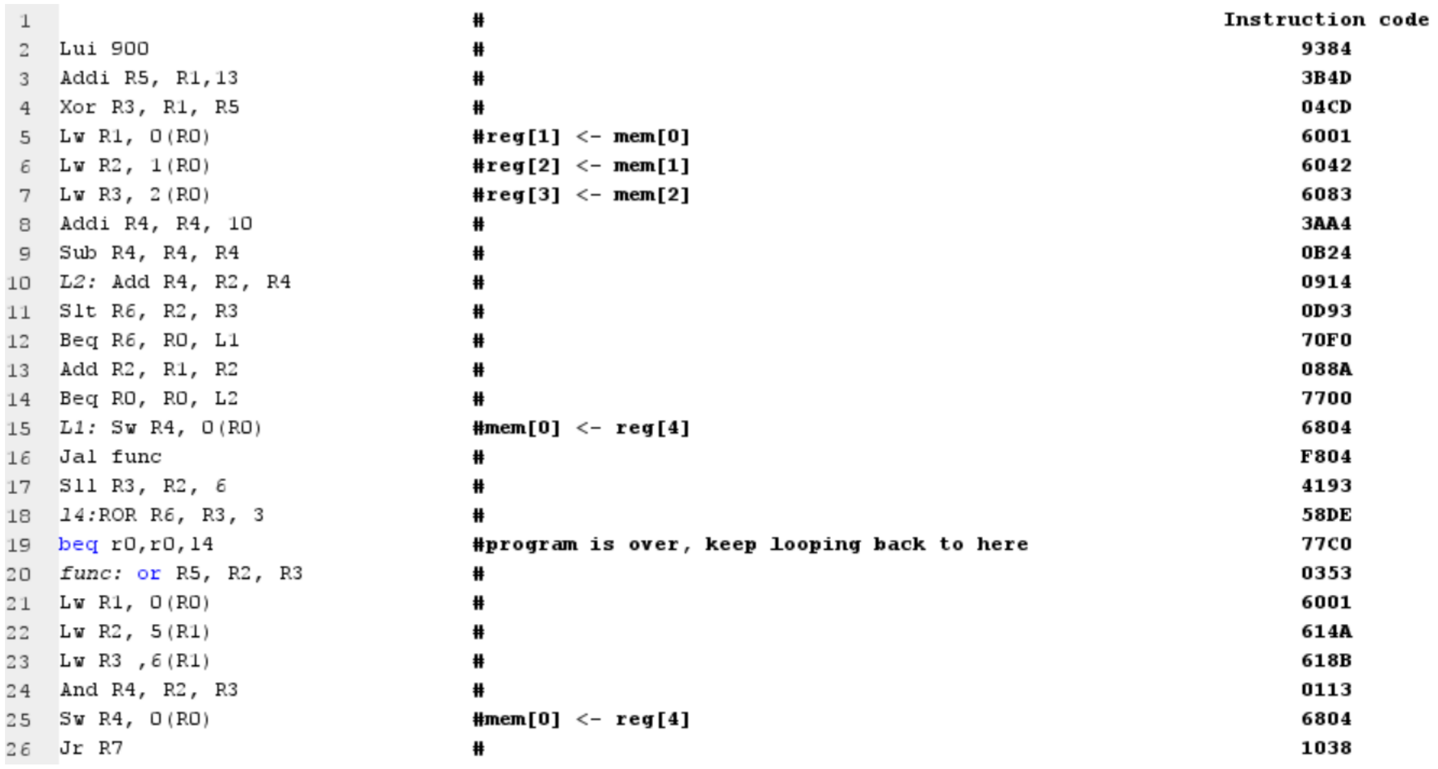
MemAddress	Expected	Actual
a	1	1
b	2	2
c	3	3
d	4	4
e	5	5
f	6	6
10	7	7
11	8	8
12	9	9
13	10	10
14	11	11
15	12	12
16	13	13
17	14	14
18	15	15

Test Pass

4.1.2 Test No.2:

The testing process of the test code sent by Eng. Jihad.

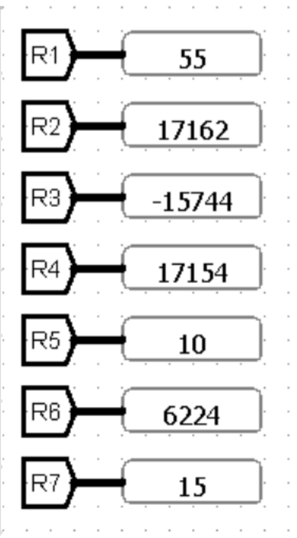
2.2.1. Assembly code and Instruction code



```
                                  #                                             Instruction code
Lui 900                           #                                                 9384
Addi R5, R1,13                    #                                                 3B4D
Xor R3, R1, R5                    #                                                 04CD
Lw R1, 0(R0)                      #reg[1] <- mem[0]                                 6001
Lw R2, 1(R0)                      #reg[2] <- mem[1]                                 6042
Lw R3, 2(R0)                      #reg[3] <- mem[2]                                 6083
Addi R4, R4, 10                   #                                                 3AA4
Sub R4, R4, R4                    #                                                 0B24
L2: Add R4, R2, R4                #                                                 0914
Slt R6, R2, R3                    #                                                 0D93
Beq R6, R0, L1                    #                                                 70F0
Add R2, R1, R2                    #                                                 088A
Beq R0, R0, L2                    #                                                 7700
L1: Sw R4, 0(R0)                  #mem[0] <- reg[4]                                 6804
Jal func                          #                                                 F804
Sll R3, R2, 6                     #                                                 4193
14:ROR R6, R3, 3                  #                                                 58DE
beq r0,r0,14                      #program is over, keep looping back to here       77C0
func: or R5, R2, R3               #                                                 0353
Lw R1, 0(R0)                      #                                                 6001
Lw R2, 5(R1)                      #                                                 614A
Lw R3 ,6(R1)                      #                                                 618B
And R4, R2, R3                    #                                                 0113
Sw R4, 0(R0)                      #mem[0] <- reg[4]                                 6804
Jr R7                             #                                                 1038
```

2.2.2. Registers Expected vs Actual values

Registers	Expected	Actual
R1	55	55
R2	17162	17162
R3	-15744	-15744
R4	17154	17154
R5	10	10
R6	6224	6224
R7	15	15

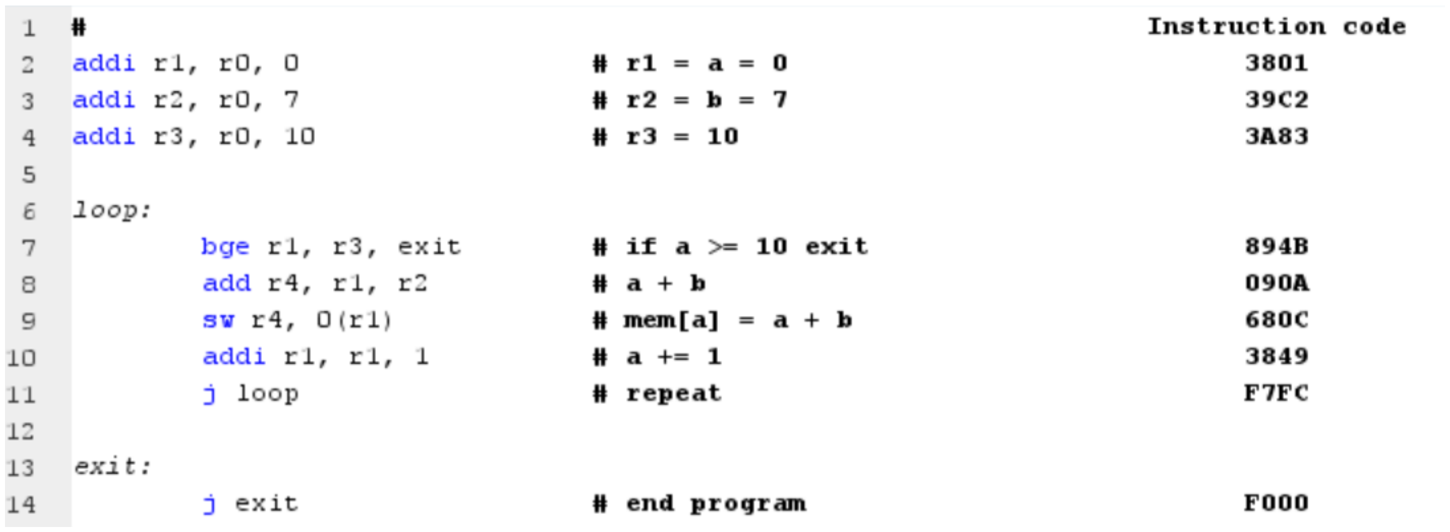


Test Pass

4.1.3. Test No.3 (Loop, Store and Jump program)

This code implements mechanisms to ensure the proper execution of store, jump, and branch instructions.

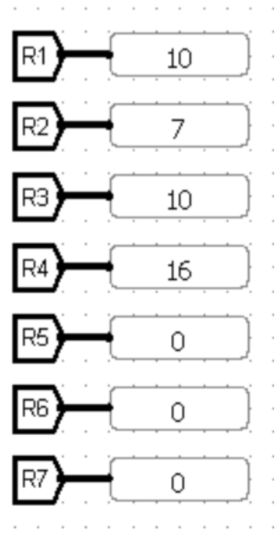
1.3.1. Assembly code and Instruction code



```
#                                                                  Instruction code
addi r1, r0, 0                  # r1 = a = 0                             3801
addi r2, r0, 7                  # r2 = b = 7                             39C2
addi r3, r0, 10                 # r3 = 10                                3A83

loop:
        bge r1, r3, exit        # if a >= 10 exit                        894B
        add r4, r1, r2          # a + b                                  090A
        sw r4, 0(r1)            # mem[a] = a + b                         680C
        addi r1, r1, 1          # a += 1                                 3849
        j loop                  # repeat                                 F7FC

exit:
        j exit                  # end program                            F000
```

1.3.2. Registers Expected vs Actual values

Registers	Expected	Actual
R1	10	10
R2	7	7
R3	10	10
R4	16	16
R5	0	0
R6	0	0
R7	0	0



Test Pass

1.3.3. Memory Expected vs Actual values

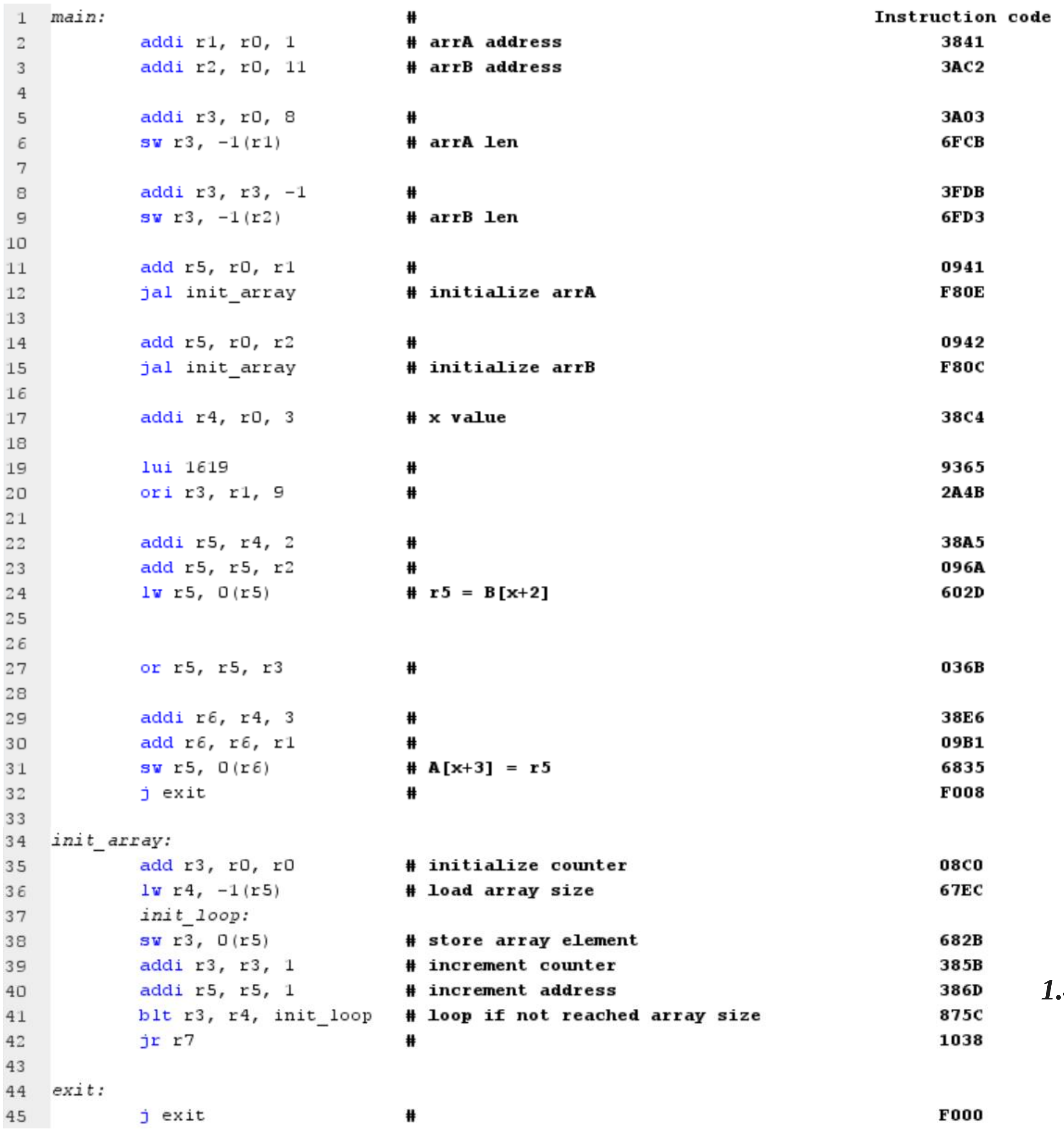
MemAddress	Expected	Actual
0	7	7
1	8	8
2	9	9
3	a	a
4	b	b
5	c	c
6	d	d
7	e	e
8	f	f

Test Pass

4.1.4. Test No.4 (Load and Store program)

This code ensures the proper execution of some essential instructions such as load upper immediately, jump, store, load and branch instructions.

1.4.1. Assembly code and Instruction code

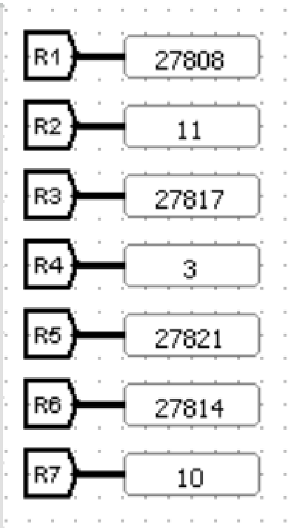


```
main:                               #                                     Instruction code
        addi r1, r0, 1              # arrA address                              3841
        addi r2, r0, 11             # arrB address                              3AC2

        addi r3, r0, 8              #                                           3A03
        sw r3, -1(r1)               # arrA len                                  6FCB

        addi r3, r3, -1             #                                           3FDB
        sw r3, -1(r2)               # arrB len                                  6FD3

        add r5, r0, r1              #                                           0941
        jal init_array              # initialize arrA                           F80E

        add r5, r0, r2              #                                           0942
        jal init_array              # initialize arrB                           F80C

        addi r4, r0, 3              # x value                                   38C4

        lui 1619                    #                                           9365
        ori r3, r1, 9               #                                           2A4B

        addi r5, r4, 2              #                                           38A5
        add r5, r5, r2              #                                           096A
        lw r5, 0(r5)                # r5 = B[x+2]                               602D


        or r5, r5, r3               #                                           036B

        addi r6, r4, 3              #                                           38E6
        add r6, r6, r1              #                                           09B1
        sw r5, 0(r6)                # A[x+3] = r5                               6835
        j exit                      #                                           F008

init_array:
        add r3, r0, r0              # initialize counter                        08C0
        lw r4, -1(r5)               # load array size                           67EC
        init_loop:
        sw r3, 0(r5)                # store array element                       682B
        addi r3, r3, 1              # increment counter                         385B
        addi r5, r5, 1              # increment address                         386D
        blt r3, r4, init_loop       # loop if not reached array size            875C
        jr r7                       #                                           1038

exit:
        j exit                      #                                           F000
```

1.4.2.

Registers Expected vs Actual values

Registers	Expected	Actual
R1	27808	27808
R2	11	11
R3	27817	27817
R4	3	3
R5	27821	27821
R6	27814	27814
R7	10	10



Test Pass

1.4.3. Memory Expected vs Actual values

MemAddress	Expected	Actual
0	8	8
1	0	0
2	1	1
3	2	2
4	3	3
5	4	4
6	5	5
7	6	6
8	7	7
9	0	0
a	7	7
b	0	0
c	1	1
d	2	2
e	3	3
f	4	4
10	5	5
11	6	6

Test Pass

4.1.5 Multiplication and Applications Test:

1.5.1 Multiplication Function:

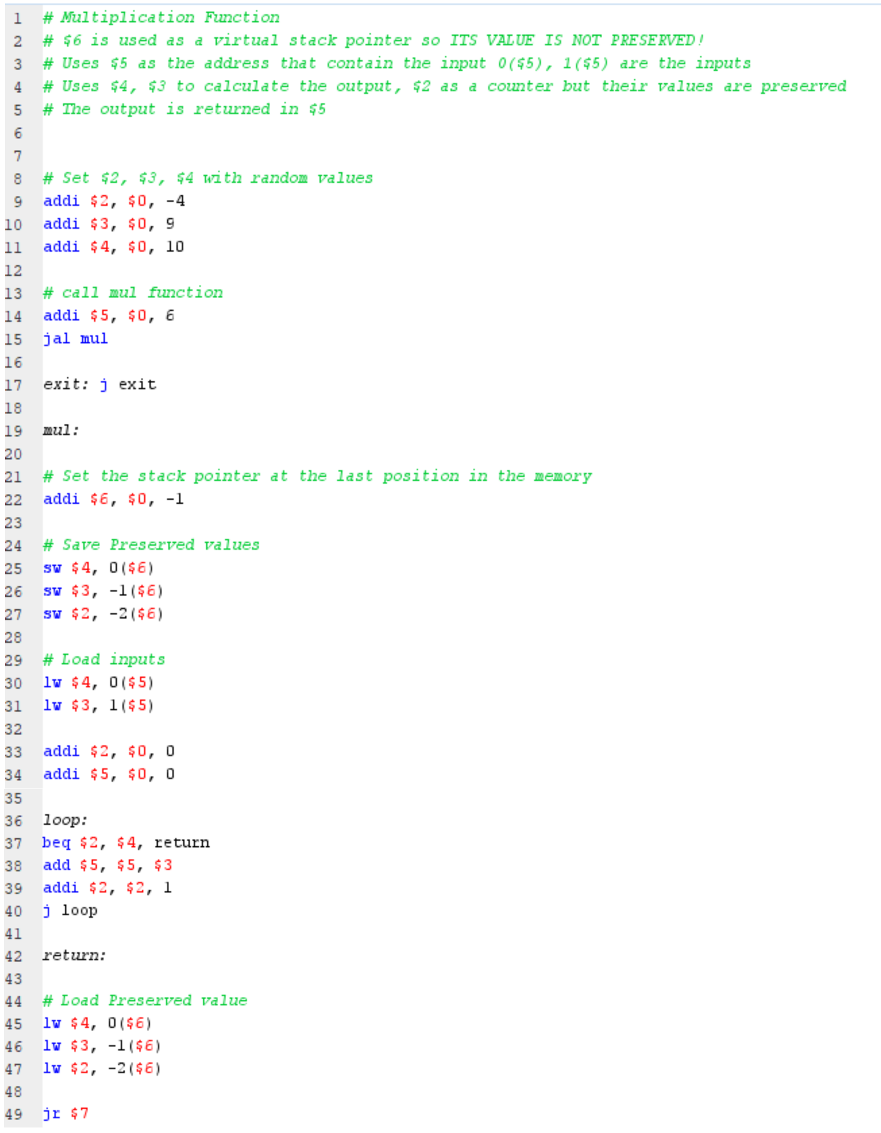
The objective of the implementation of the multiplication function is to be able to use it in in larger applications.

To achieve that we need to use a little of registers as possible. So we will need a virtual Stack.

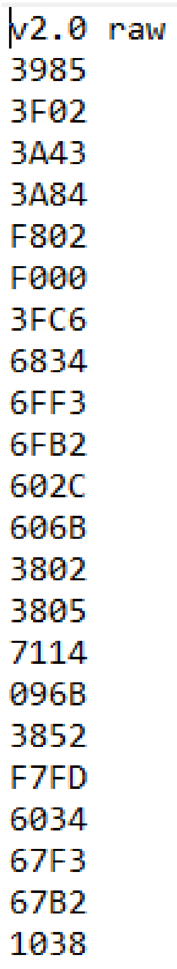
The input of the function is in $5 and it will contain the address that contain the inputs in memory 0($5), 1($5) are the inputs. Uses $4, $3 to calculate the output, $2 as a counter but their values are preserved at the last 3 places in the memory each time you call the function. $6 will be used as the stack pointer only in this function so ITS VALUE IS NOT PRESERVED!

5.1.1 Assembly code and Instruction code

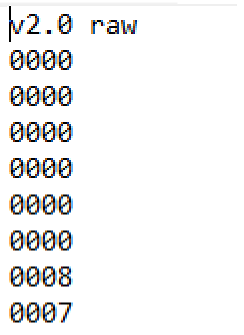
Memory Inputs



```
# Multiplication Function
# $6 is used as a virtual stack pointer so ITS VALUE IS NOT PRESERVED!
# Uses $5 as the address that contain the input 0($5), 1($5) are the inputs
# Uses $4, $3 to calculate the output, $2 as a counter but their values are preserved
# The output is returned in $5


# Set $2, $3, $4 with random values
addi $2, $0, -4
addi $3, $0, 9
addi $4, $0, 10

# call mul function
addi $5, $0, 6
jal mul

exit: j exit

mul:

# Set the stack pointer at the last position in the memory
addi $6, $0, -1

# Save Preserved values
sw $4, 0($6)
sw $3, -1($6)
sw $2, -2($6)

# Load inputs
lw $4, 0($5)
lw $3, 1($5)

addi $2, $0, 0
addi $5, $0, 0

loop:
beq $2, $4, return
add $5, $5, $3
addi $2, $2, 1
j loop

return:

# Load Preserved value
lw $4, 0($6)
lw $3, -1($6)
lw $2, -2($6)

jr $7
```



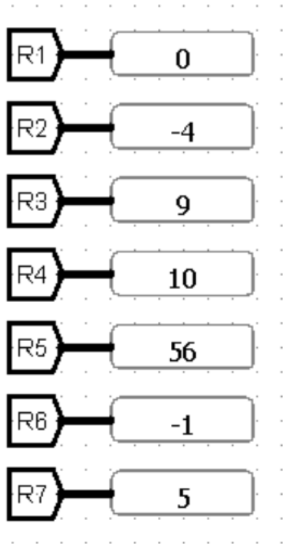
```
v2.0 raw
3985
3F02
3A43
3A84
F802
F000
3FC6
6834
6FF3
6FB2
602C
606B
3802
3805
7114
096B
3852
F7FD
6034
67F3
67B2
1038
```



```
v2.0 raw
0000
0000
0000
0000
0000
0000
0008
0007
```

5.1.2. Registers Expected vs Actual values

Registers	Expected	Actual
R1	0	0
R2	-4	-4
R3	9	9
R4	10	10
R5	56	56
R6	-1	-1
R7	5	5



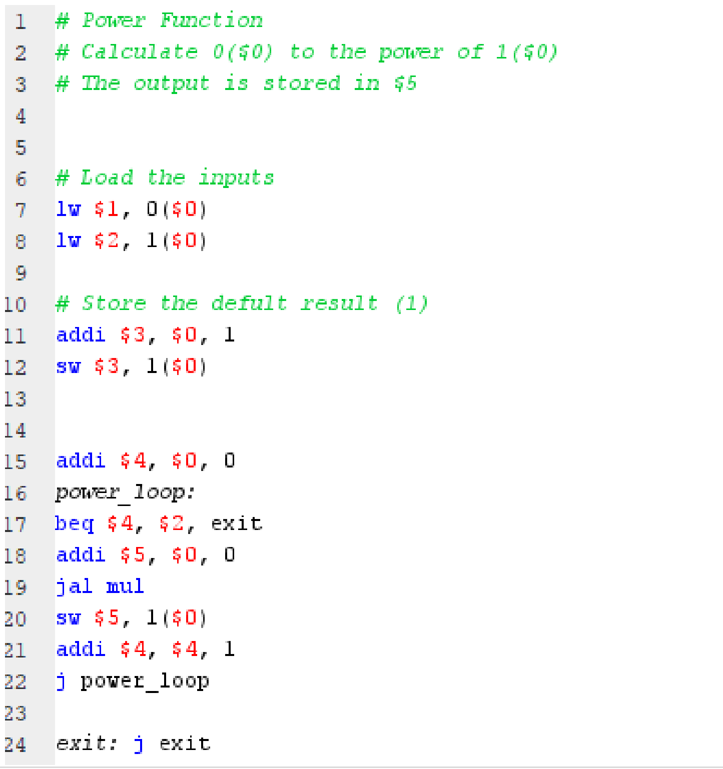
Test Pass

1.5.2 Powers:

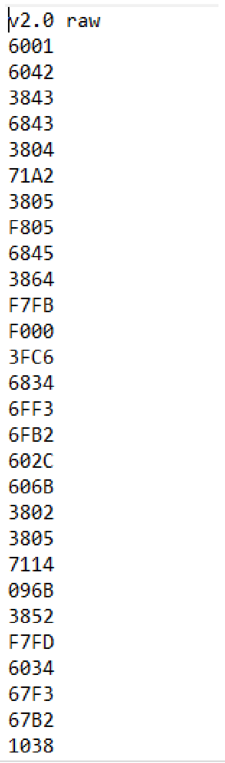
In this test case we use the mul function (3.5.1) to calculate powers. The inputs are the first and second words in memory {0($0), 1($0)}. The output equals the first input raised to the power of the second input. The output is stored in R5

5.2.1 Assembly code and Instruction code

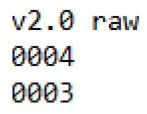
Memory Inputs



```
# Power Function
# Calculate 0($0) to the power of 1($0)
# The output is stored in $5


# Load the inputs
lw $1, 0($0)
lw $2, 1($0)

# Store the defult result (1)
addi $3, $0, 1
sw $3, 1($0)


addi $4, $0, 0
power_loop:
beq $4, $2, exit
addi $5, $0, 0
jal mul
sw $5, 1($0)
addi $4, $4, 1
j power_loop

exit: j exit
```



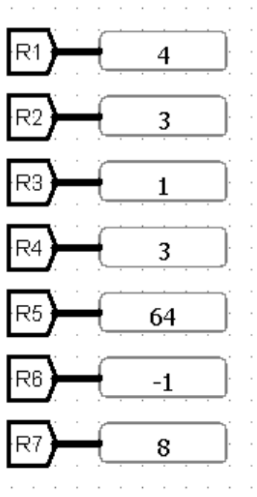
```
v2.0 raw
6001
6042
3843
6843
3804
71A2
3805
F805
6845
3864
F7FB
F000
3FC6
6834
6FF3
6FB2
602C
606B
3802
3805
7114
096B
3852
F7FD
6034
67F3
67B2
1038
```



```
v2.0 raw
0004
0003
```

5.2.2. Registers Expected vs Actual values

Registers	Expected	Actual
R1	4	4
R2	3	3
R3	1	1
R4	3	3
R5	64	64
R6	-1	-1
R7	8	8



Test Pass

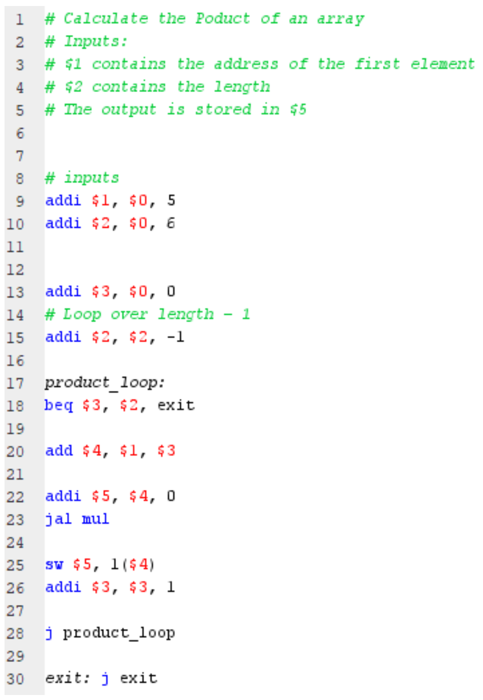
1.5.3 Product of an array:

In this test case we use the mul function (3.5.1) to calculate the product of an array stored in memory.

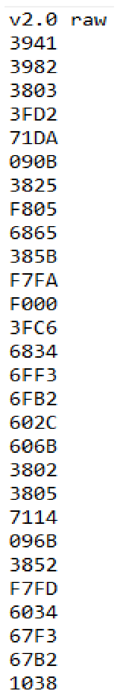
The inputs are R1 contains the address of the first element R2 contains the length of the array.

The output is stored in R5

5.3.1 Assembly code and Instruction code

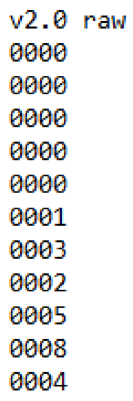


```
# Calculate the Poduct of an array
# Inputs:
# $1 contains the address of the first element
# $2 contains the length
# The output is stored in $5


# inputs
addi $1, $0, 5
addi $2, $0, 6


addi $3, $0, 0
# Loop over length - 1
addi $2, $2, -1

product_loop:
beq $3, $2, exit

add $4, $1, $3

addi $5, $4, 0
jal mul

sw $5, 1($4)
addi $3, $3, 1

j product_loop

exit: j exit
```



```
v2.0 raw
3941
3982
3803
3FD2
71DA
090B
3825
F805
6865
385B
F7FA
F000
3FC6
6834
6FF3
6FB2
602C
606B
3802
3805
7114
096B
3852
F7FD
6034
67F3
67B2
1038
```

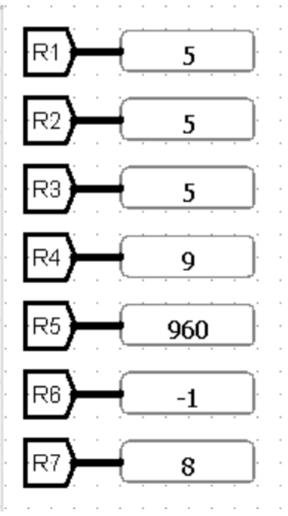
Memory Inputs



```
v2.0 raw
0000
0000
0000
0000
0000
0001
0003
0002
0005
0008
0004
```

5.3.2. Registers Expected vs Actual values

Registers	Expected	Actual
R1	5	4
R2	5	3
R3	5	1
R4	9	3
R5	960	64
R6	-1	-1
R7	8	8



Test Pass

4.2 Testing (Pipeline)

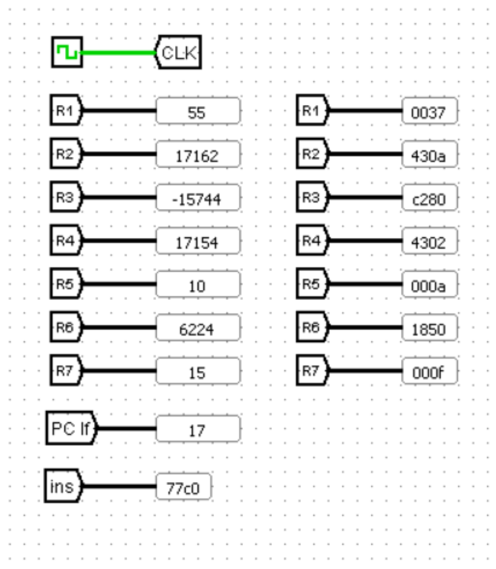
Test No1:

The testing process of the test code sent by Eng. Jihad.

Expected:

Instruction	R1	R2	R3	R4	R5	R6	R7
Lui 900	7080	0	0	0	0	0	0
Addi R5, R1,13	7080	0	0	0	708d	0	0
Xor R3, R1, R5	7080	0	d	0	708d	0	0
Lw R1, 0(R0)	1	0	d	0	708d	0	0
Lw R2, 1(R0)	1	1	d	0	708d	0	0
Lw R3, 2(R0)	1	1	a	0	708d	0	0
Addi R4, R4, 10	1	1	a	a	708d	0	0
Sub R4, R4, R4	1	1	a	0	708d	0	0
L2: Add R4, R2, R4	1	1	a	1	708d	0	0
Slt R6, R2, R3	1	1	a	1	708d	1	0
Beq R6, R0, L1	1	1	a	1	708d	1	0
Add R2, R1, R2	1	2	a	1	708d	1	0
Beq R0, R0, L2	1	2	a	1	708d	1	0
L2: Add R4, R2, R4	1	a	a	37	708d	1	0
Slt R6, R2, R3	1	a	a	37	708d	0	0
Beq R6, R0, L1	1	a	a	37	708d	0	0
L1: Sw R4, 0(R0)	1	a	a	37	708d	0	0
Jal func	1	a	a	37	708d	0	f
func: or R5, R2, R3	1	a	a	37	a	0	f
Lw R1, 0(R0)	37	a	a	37	a	0	f
Lw R2, 5(R1)	37	430a	a	37	a	0	f
Lw R3 ,6(R1)	37	430a	7342	37	a	0	f
And R4, R2, R3	37	430a	7342	4302	a	0	f
Sw R4, 0(R0)	37	430a	7342	4302	a	0	f
Jr R7	37	430a	7342	4302	a	0	f
Sll R3, R2, 6	37	430a	c280	4302	a	0	f
l4:ROR R6, R3, 3	37	430a	c280	4302	a	1850	f
beq r0,r0,l4	37	430a	c280	4302	a	1850	f

Actual :



Test Pass

Load & Store Forward Test:

This test confirms that the forwarding unit works perfectly.

Assembly Code

```
addi $1, $0, 10
sw $1, 0($0)
addi $2, $0, 15
LW $4, 0($0)
add $4,$4,$2
sw $4, 1($0)
```

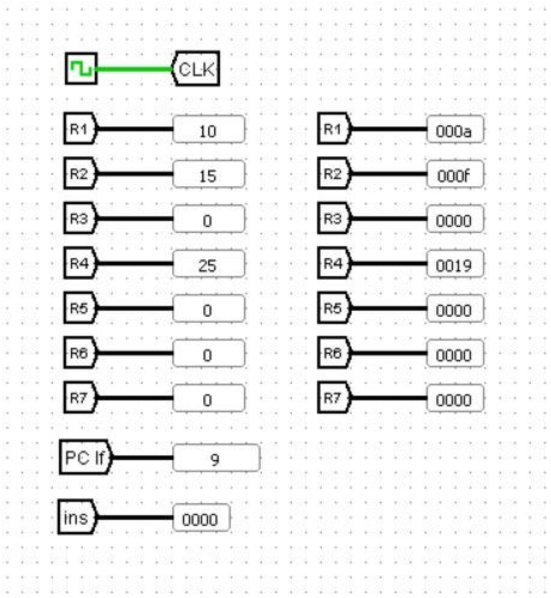
Machine Code

```
v2.0 raw
3A81
6801
3BC2
6004
0922
6844
```

Output:

	R1	R2	R3	R4	R5	R6	R7
Expexcted	10	15	0	25	0	0	0
Actual	10	15	0	25	0	0	0

Register values



Memory

```
v2.0 raw
a 19
```

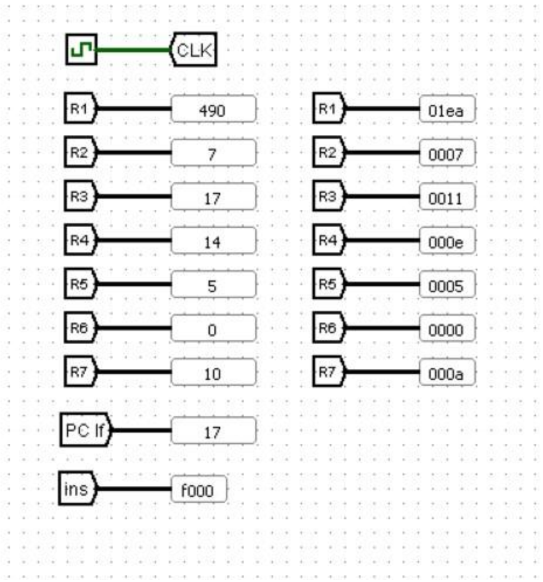
Test Pass

General Pipeline Test:

Assembly Code	Machine Code
`addi $1,$0, 5`	3941
`addi $2,$0, 7`	39C2
`addi $3,$0, 15`	3BC3
`addi $4,$0, 14`	3B84
`sw $1, 0($0)`	6801
`sw $2, 1($0)`	6842
`sw $3, 2($0)`	6883
`sw $4, 3($0)`	68C4
`lw $5, 0($0)`	6005
`jal func`	F806
`l1:lui 15`	900F
`sw $1, 0($0)`	6801
`add $1, $1, $7`	084F
`addi $3, $3, 2`	389B
`jr $3`	1018
`func: add $1, $0, $7`	0847
`bne $1, $0, l1`	7E88
`exit: j exit`	F000

Output:

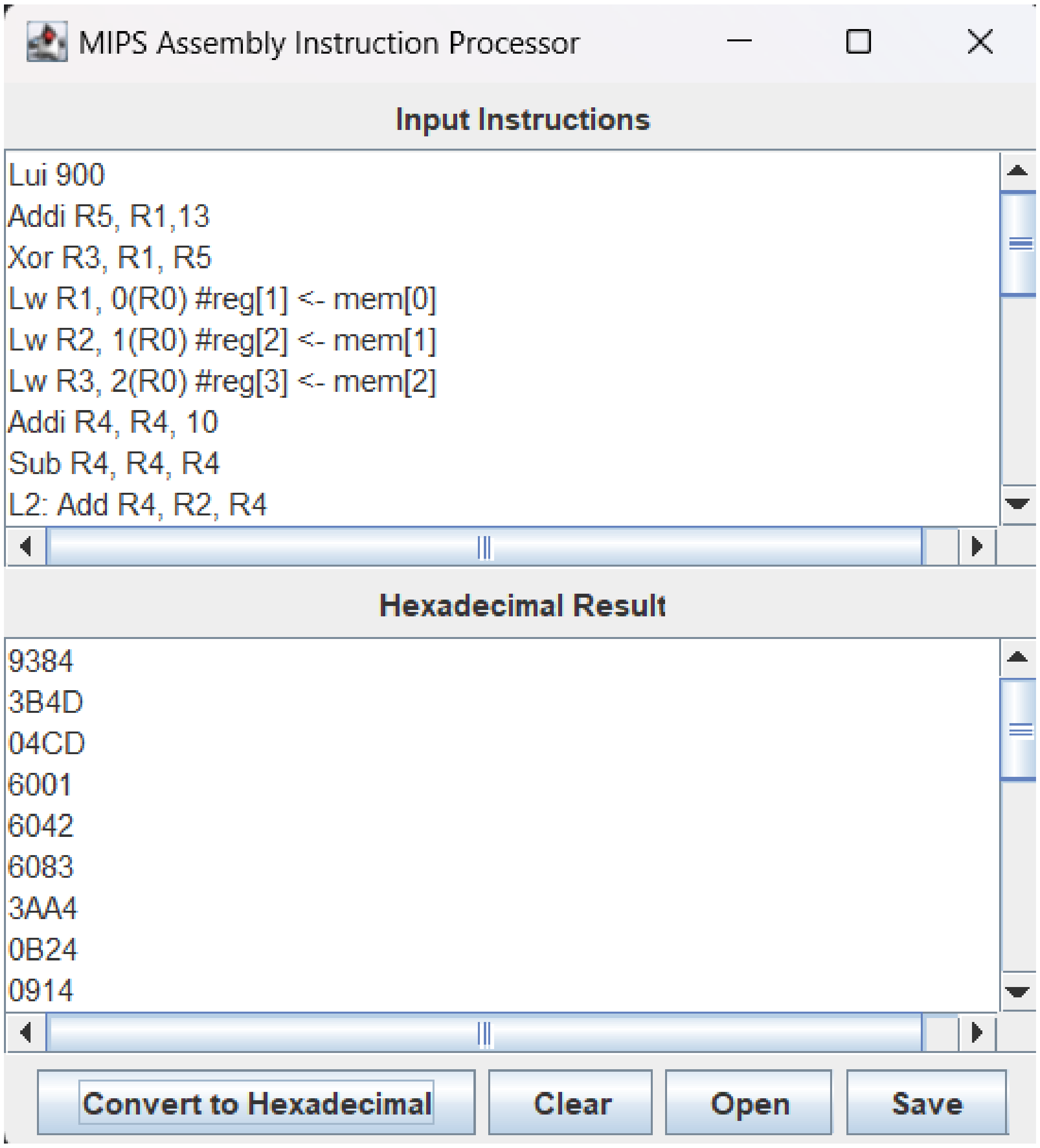
	R1	R2	R3	R4	R5	R6	R7
Expexcted	490	7	17	14	5	0	10
Actual	490	7	17	14	5	0	10



Test Pass

5. Assembler

We need to test our circuit, but it requires instructions in machine code (hexadecimal). To bridge this gap, we've developed a custom assembler program. This program acts as a translator, taking the instructions we provide and converting them into the hexadecimal values the circuit can understand. Now, we can use these machine code instructions to test and verify the circuit's functionality.



```
Lui 900
Addi R5, R1,13
Xor R3, R1, R5
Lw R1, 0(R0) #reg[1] <- mem[0]
Lw R2, 1(R0) #reg[2] <- mem[1]
Lw R3, 2(R0) #reg[3] <- mem[2]
Addi R4, R4, 10
Sub R4, R4, R4
L2: Add R4, R2, R4
```

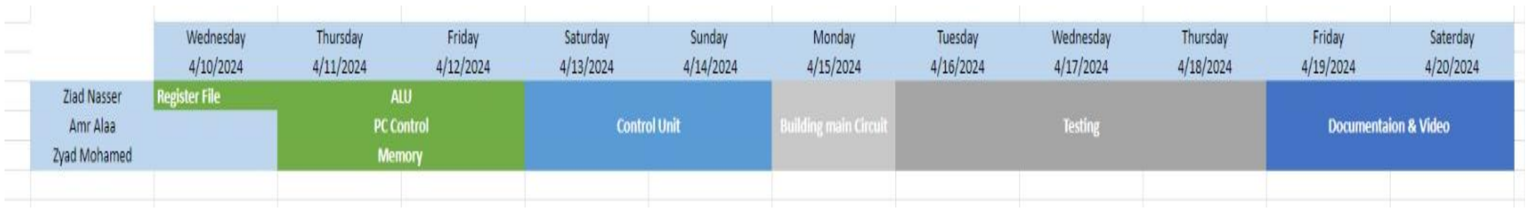
```
9384
3B4D
04CD
6001
6042
6083
3AA4
0B24
0914
```

6. Team work

We implemented this processor using GitHub.

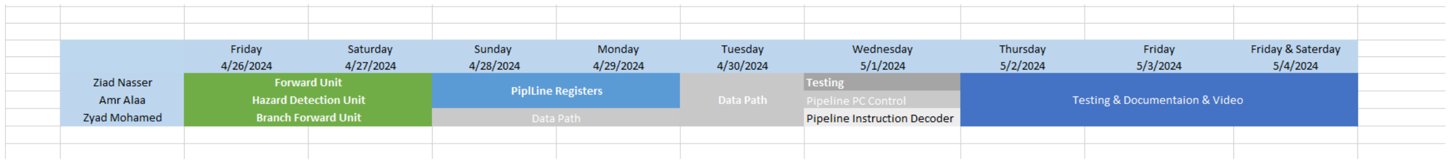
- We have done about 9 meetings since we started on the 1st of this month.

Work distribution chart



	Wednesday 4/10/2024	Thursday 4/11/2024	Friday 4/12/2024	Saturday 4/13/2024	Sunday 4/14/2024	Monday 4/15/2024	Tuesday 4/16/2024	Wednesday 4/17/2024	Thursday 4/18/2024	Friday 4/19/2024	Saterday 4/20/2024
Ziad Nasser	Register File	ALU									
Amr Alaa		PC Control		Control Unit		Building main Circuit	Testing			Documentaion & Video	
Zyad Mohamed		Memory									

- We have done about 9 meetings since we started on the 22nd of this month.



	Friday 4/26/2024	Saturday 4/27/2024	Sunday 4/28/2024	Monday 4/29/2024	Tuesday 4/30/2024	Wednesday 5/1/2024	Thursday 5/2/2024	Friday 5/3/2024	Friday & Saterday 5/4/2024
Ziad Nasser	Forward Unit		PiplLine Registers			Testing			
Amr Alaa	Hazard Detection Unit				Data Path	Pipeline PC Control	Testing & Documentaion & Video		
Zyad Mohamed	Branch Forward Unit		Data Path			Pipeline Instruction Decoder			