

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:

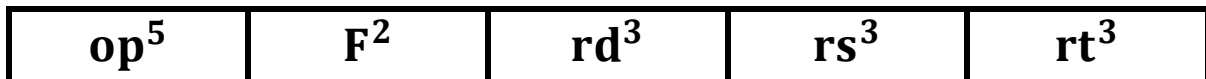
1	Introduction
2	Phase One Components
2.1.	PC Control
2.1.1.	Branch Control
2.2.	Instruction Memory
2.3.	Register File.....
2.4.	ALU
2.5.	Extender
2.6.	Data Memory
2.7.	Control Unit
3	Testing
3.1.	Sum of an Array
3.2.	Test Code 2024_v2
3.3.	Test 0
3.4.	Test 1
3.5.	Multiplication and Applications Test
3.5.1.	Multiplication Function
3.5.2.	Powers
3.5.3.	Product of an Array.....
4	Teamwork

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.

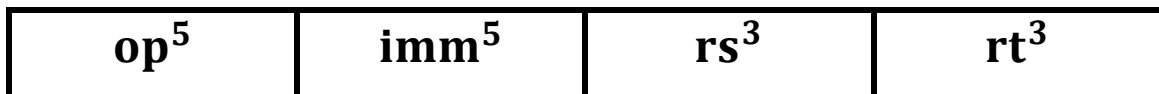
R-type format:

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).



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.



J-type format

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



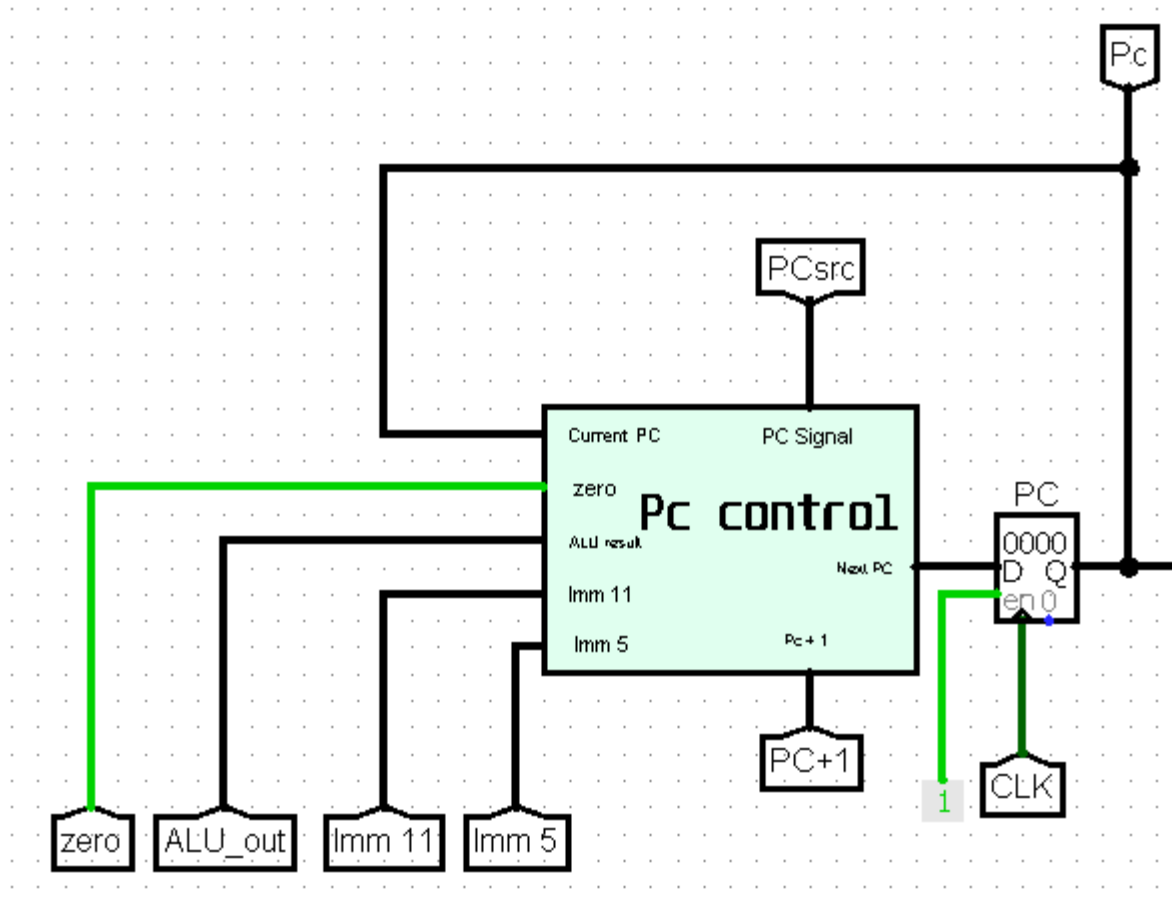
After analyzing the required instructions, we identified the necessary components for the processor's data path. We then designed each of these components and successfully implemented all the targeted instructions.

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):

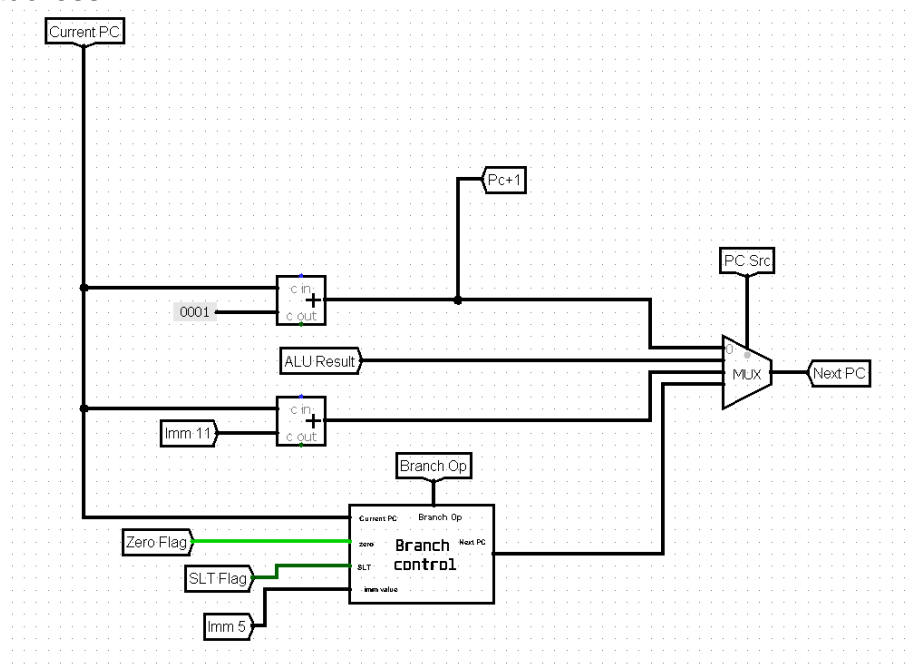
- Used for the Jump Register (JR) instruction.
- In JR instruction Rs is added to 0 and the Pc control takes the result as ALU result and stores that value as the new PC.

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.

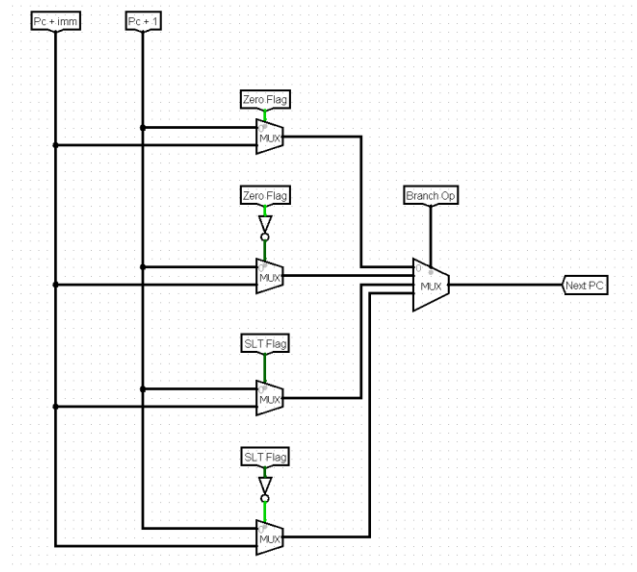
4- PC + sign-extend (Imm5):

- 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.

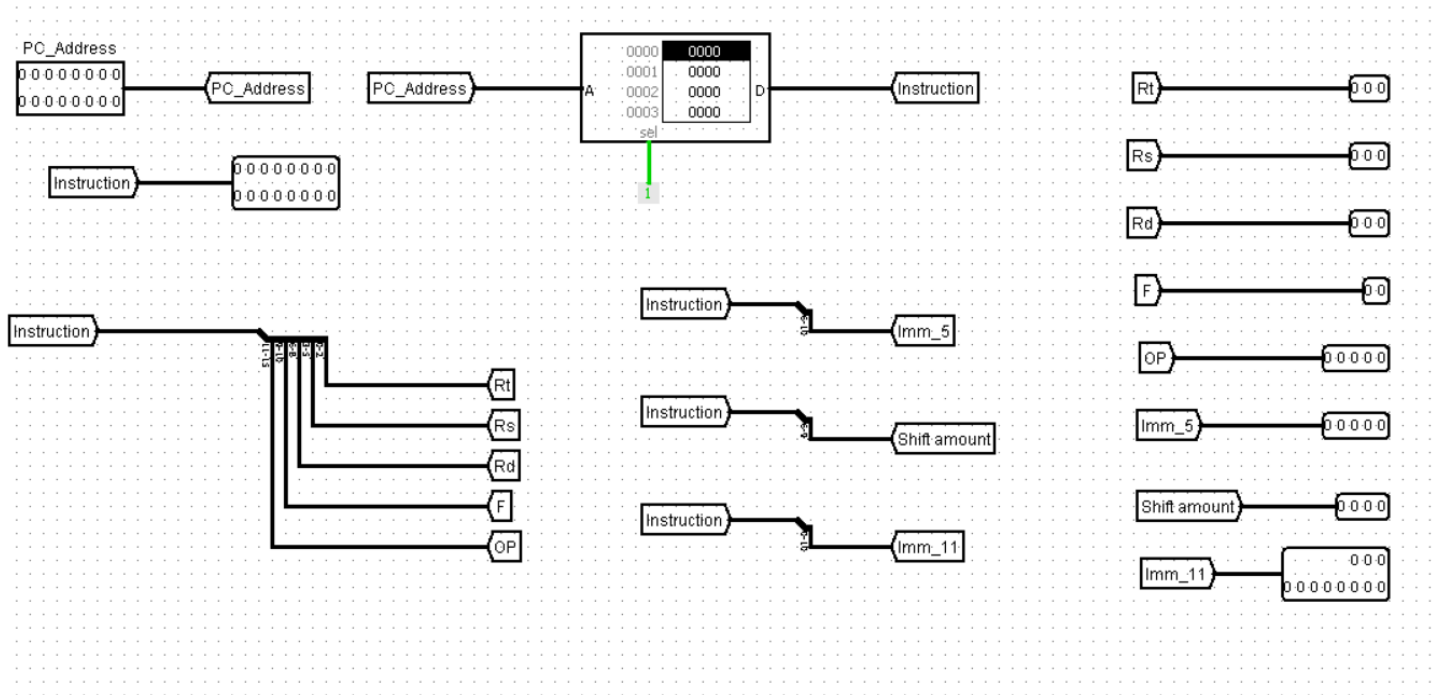


2.1.2 Branch Control:

In the case of a branch instruction the next PC value is either $PC + 1$ or $PC + \text{sign-extend}(\text{Imm5})$. 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.



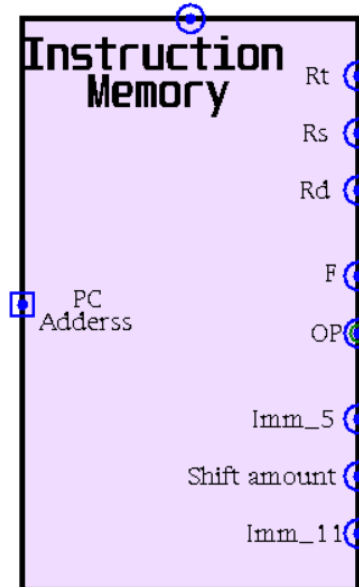
2.2 Instruction Memory:



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.

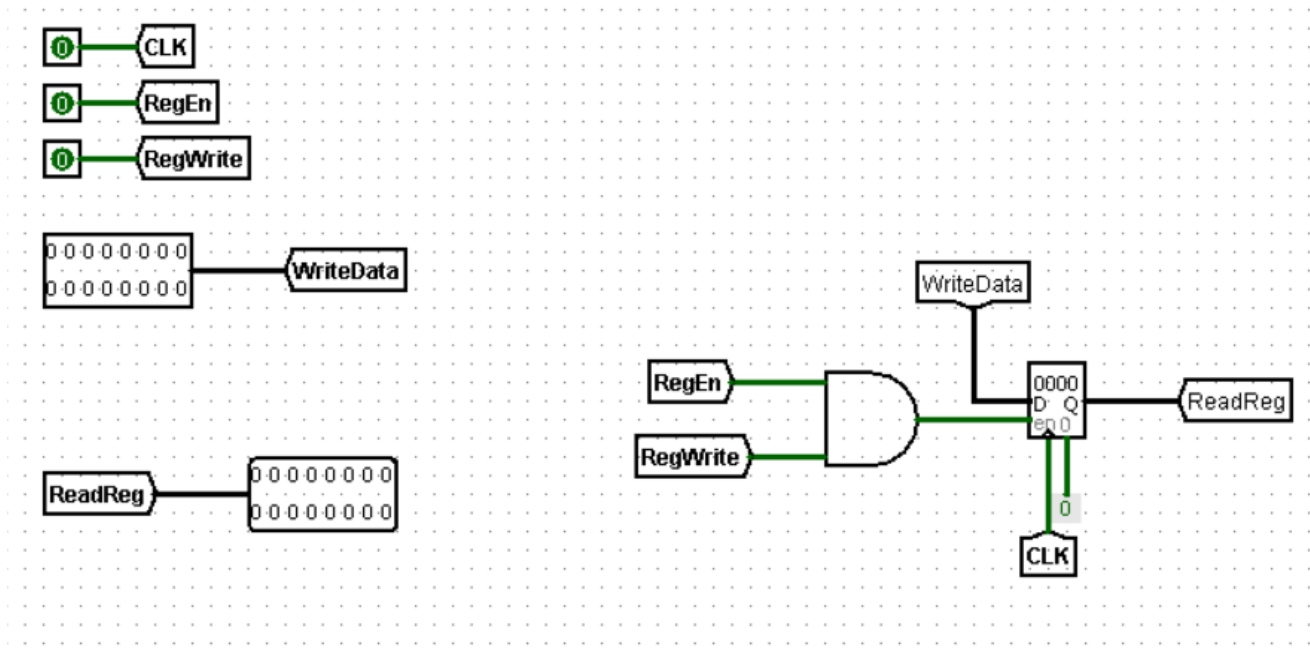


2.3 Register File:

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:

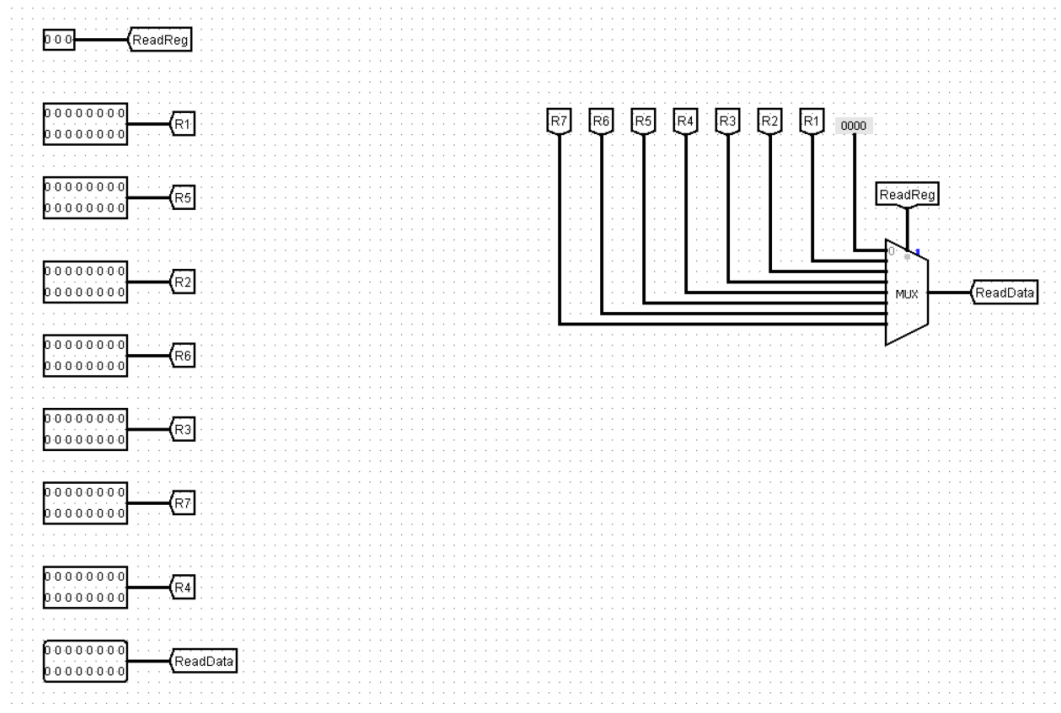
1. The general RegWrite signal must be active (typically logic 1). This signal indicates that a write operation is happening to one of the registers in the data path.
2. The sub register's specific enable signal (often called reg_enable) must also be active (logic 1). This signal selects which specific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

2.2.3. 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.

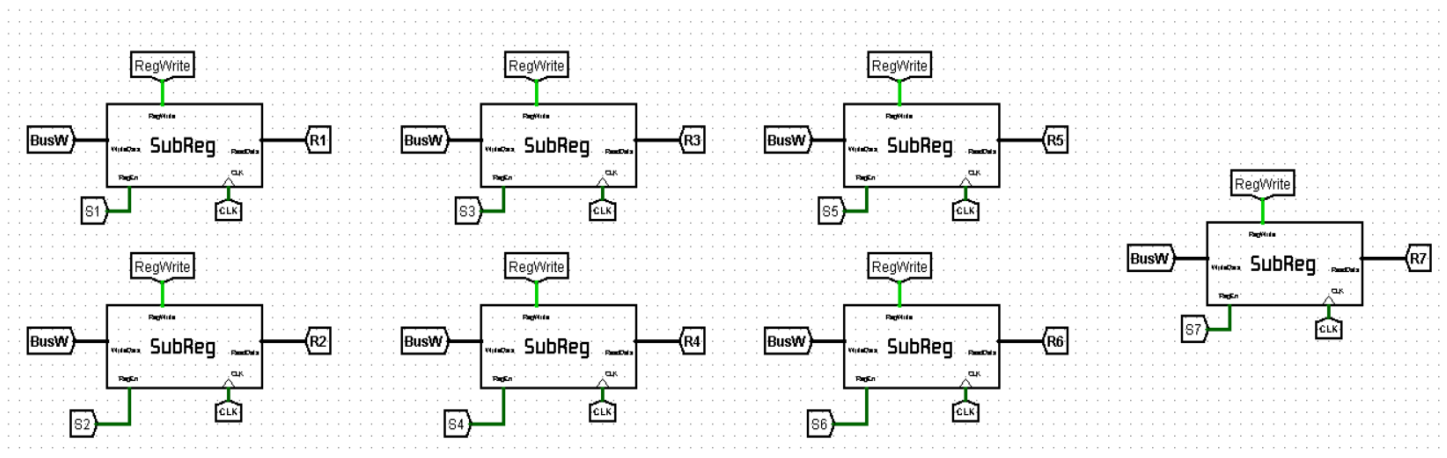
During instruction execution, when the processor needs to read data from a register, the decoded instruction provides the control signal to the MUX. The MUX then routes the data from the selected register's output line to the data output line of the register selector. This data becomes the source for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

Now we build the required sub circuits and can build our register file.

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.

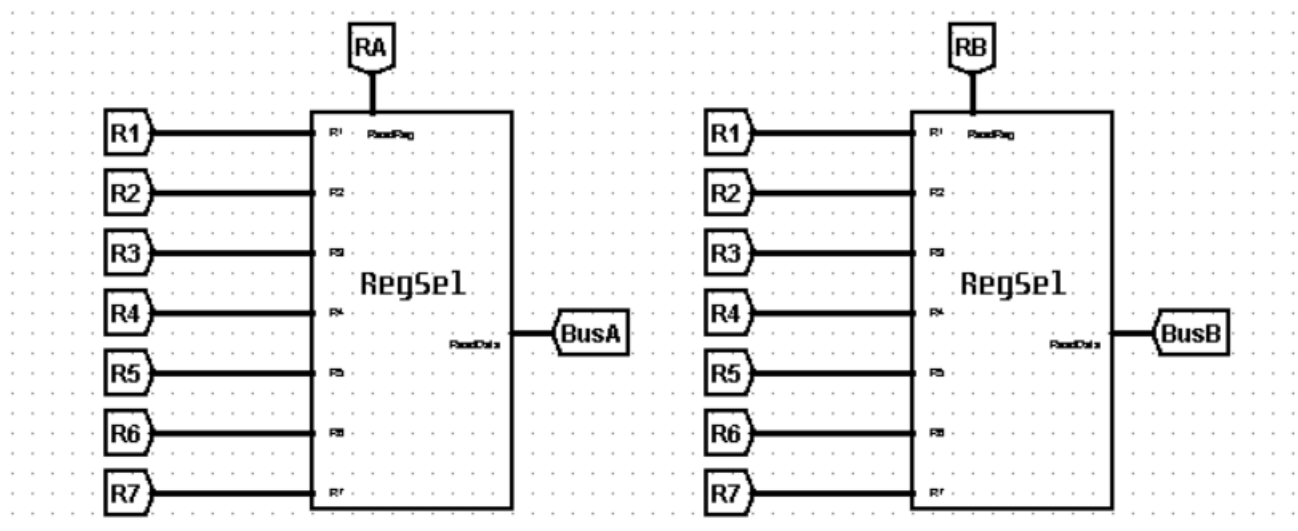
To create a complete register file with eight registers, we'll replicate this subregister seven times. This provides us with the core components for registers r1 through r7.

(register r0 is typically hardwired with zero to ensure consistent behavior and simplify operations within the process)



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.

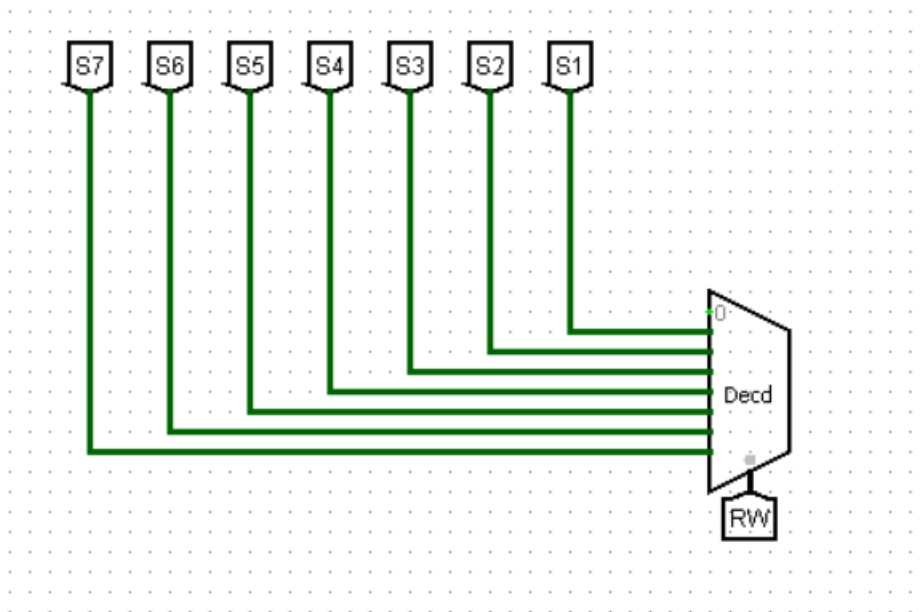


Writing to the Register File

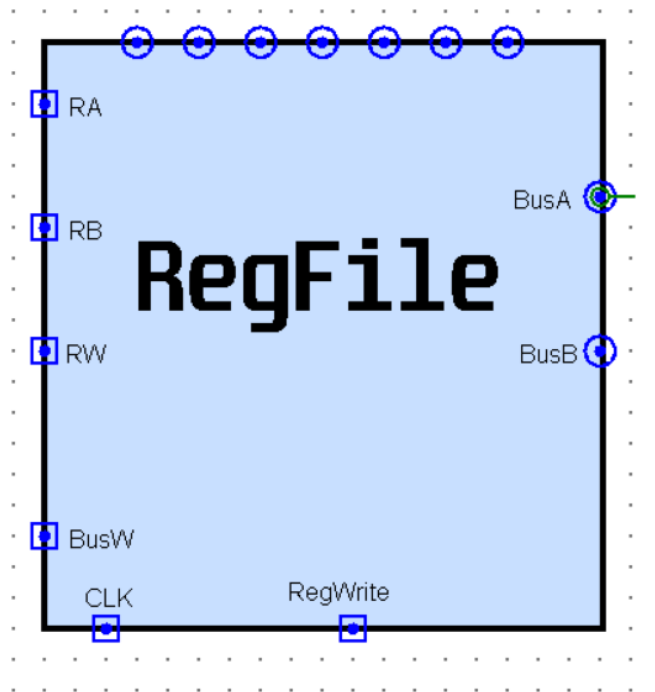
While the register file can receive data through a single input (data.in), only one register can be written to at a time. To ensure this controlled write operation, we utilize two signals:

- **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.



Finally, our register file will be utilized in the single-cycle circuit.

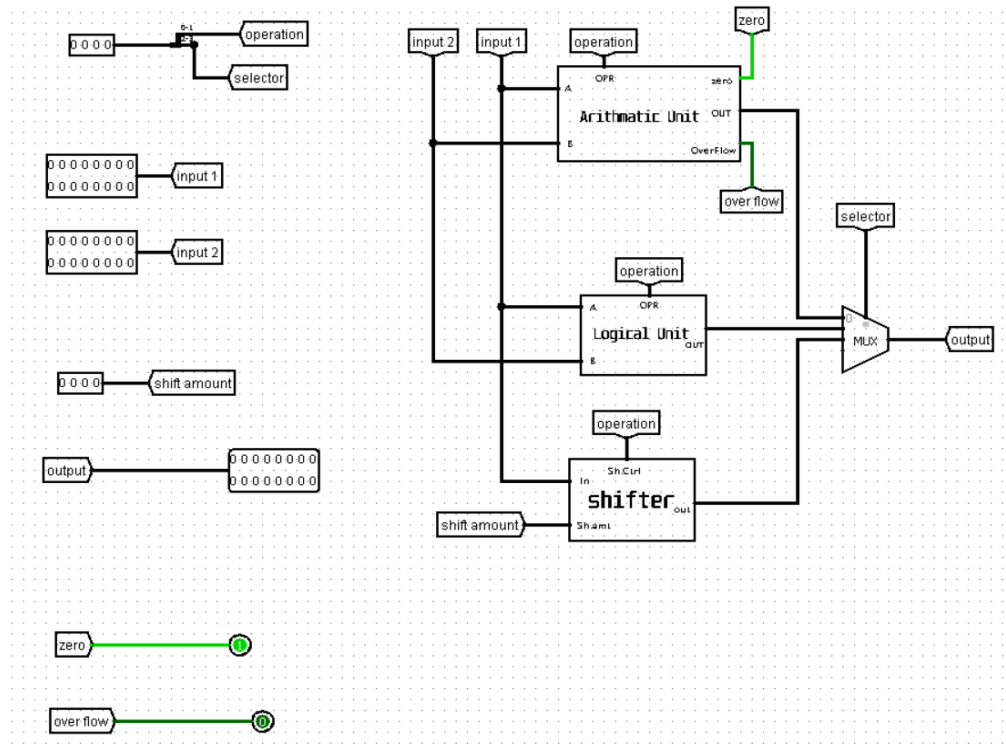


2.4 ALU:

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

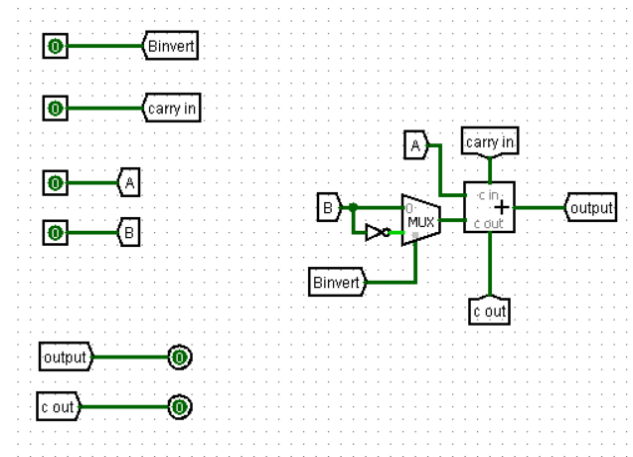
The Arithmetic Unit (AU) is the core component within the ALU responsible for all arithmetic operations. It's essentially a high-speed calculator unit specifically designed for binary data manipulation within the processor.

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 C_{in} (Carry In), dictates the adder's behavior:

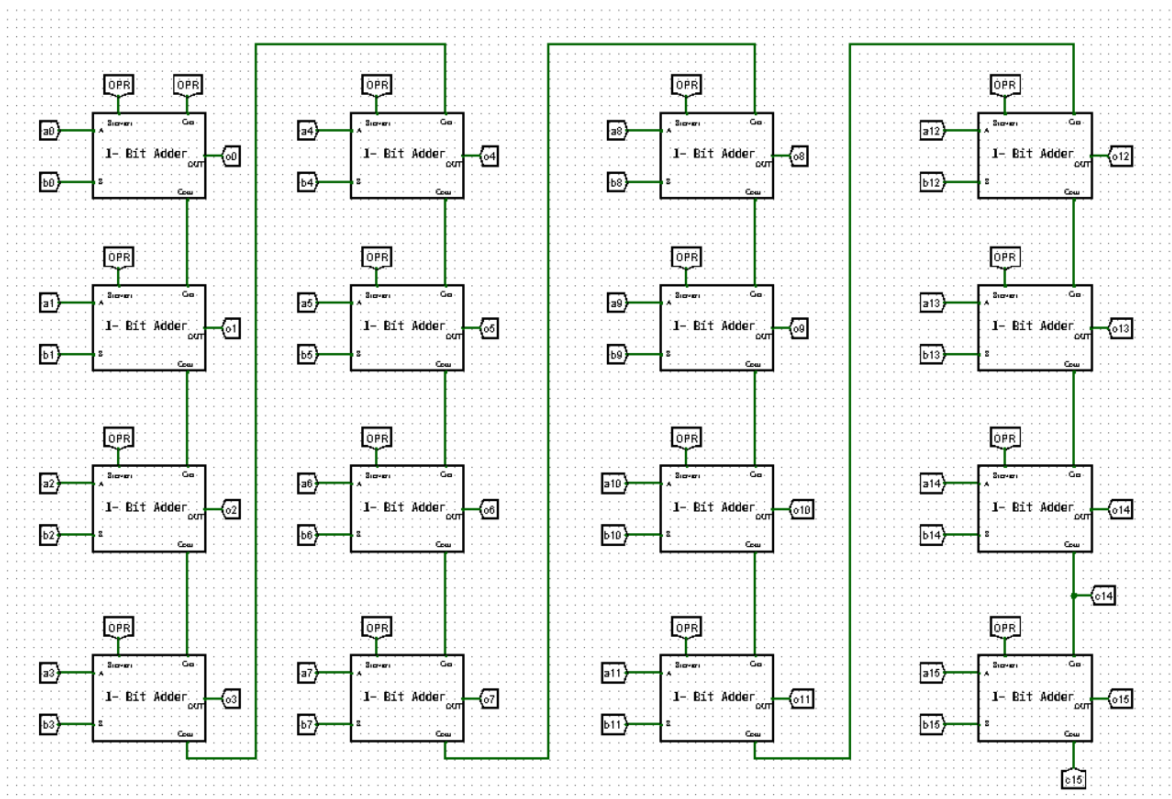
- $C_{in} = 0$: When C_{in} is set to 0, the adder performs a standard addition of the two input bits (A and B).
- $C_{in} = 1$: When C_{in} is set to 1, the adder behaves differently. It takes the complement of the second input bit (B) and adds 1 (effectively performing 2's complement addition). This 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.



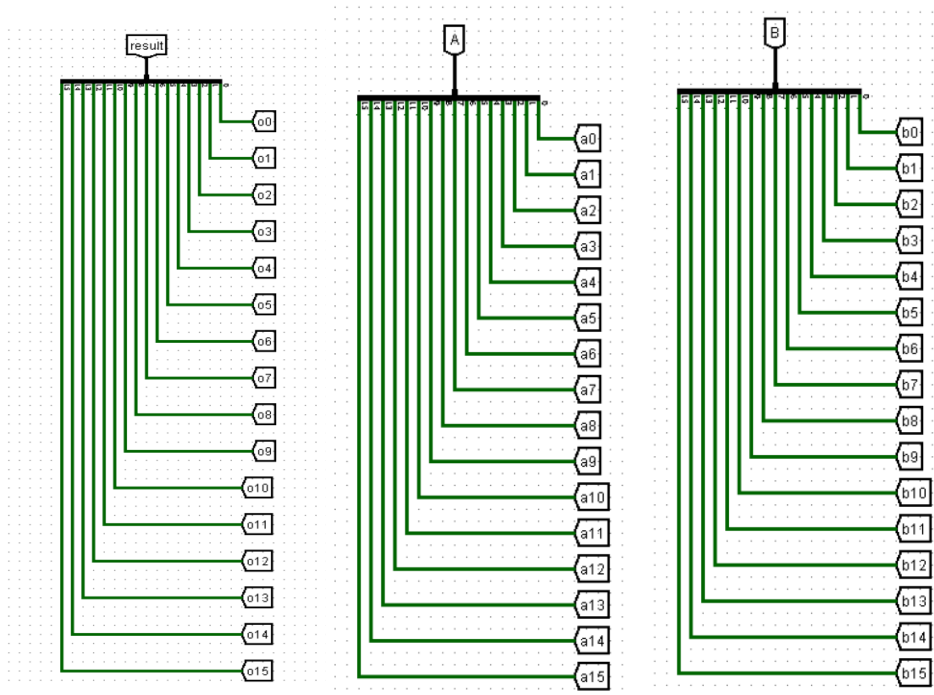
We'll need 16 of these 1-bit adders working together. Why 16? Because we want to handle 16-bit numbers, so each adder tackles one bit of the larger numbers.

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.

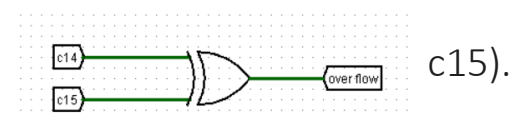
Each adder produces a sum bit, which contributes to the final answer. We need to combine these 16 sum bits into the final result.



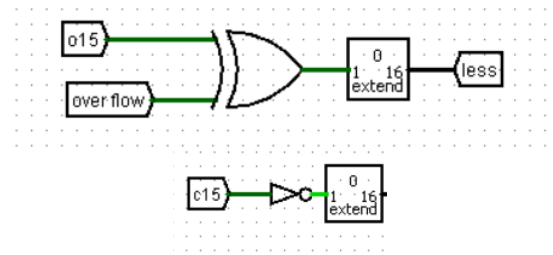
Now, we need a way to choose the final output based on what operation we want to perform (add, subtract, set less than (SLT), or set less than unsigned (SLTU)). Here's where a **multiplexer (MUX)** comes in.

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.

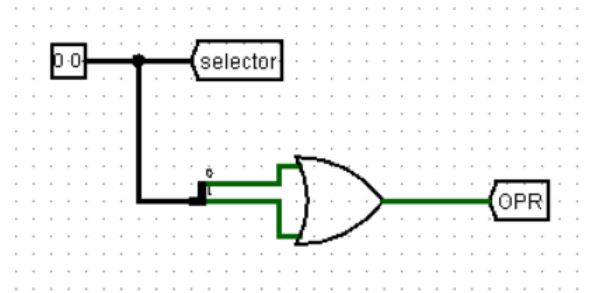
Overflow Flag: To detect overflow (when the result is too large to fit in the designated number of bits), we can use an XOR operation on the last two carry outputs (c14 and



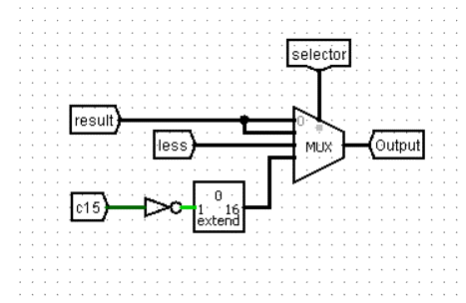
SLT and SLTU: These operations determine if one number is less than another, considering signed or unsigned values. We can use the overflow flag and the last output bit (o15) to calculate SLT, while inverting the last carry (c15) gives us SLTU.



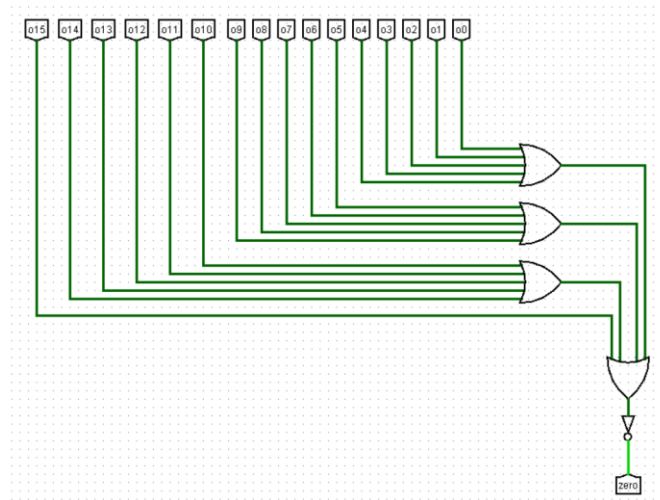
Operation Signal (opSignal): This is a crucial 1-bit signal that tells the first adder (remember, the one with the special Cin control) whether to perform a regular addition (opSignal = 0) or use 2's complement for subtraction. We'll achieve this by cleverly combining the 2-bit control signal using an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



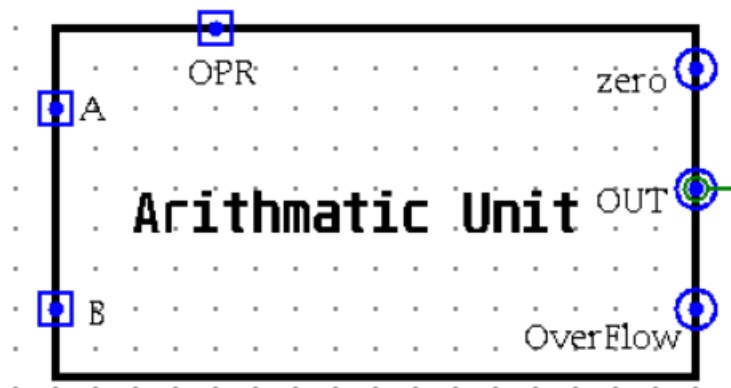
Connecting the MUX: Finally, the MUX connects the results from the 16 adders (for addition or subtraction) and the calculated SLT and SLTU outputs to the final result line based on the control signal.



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.

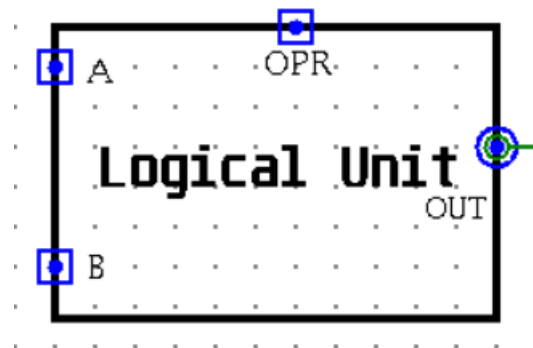
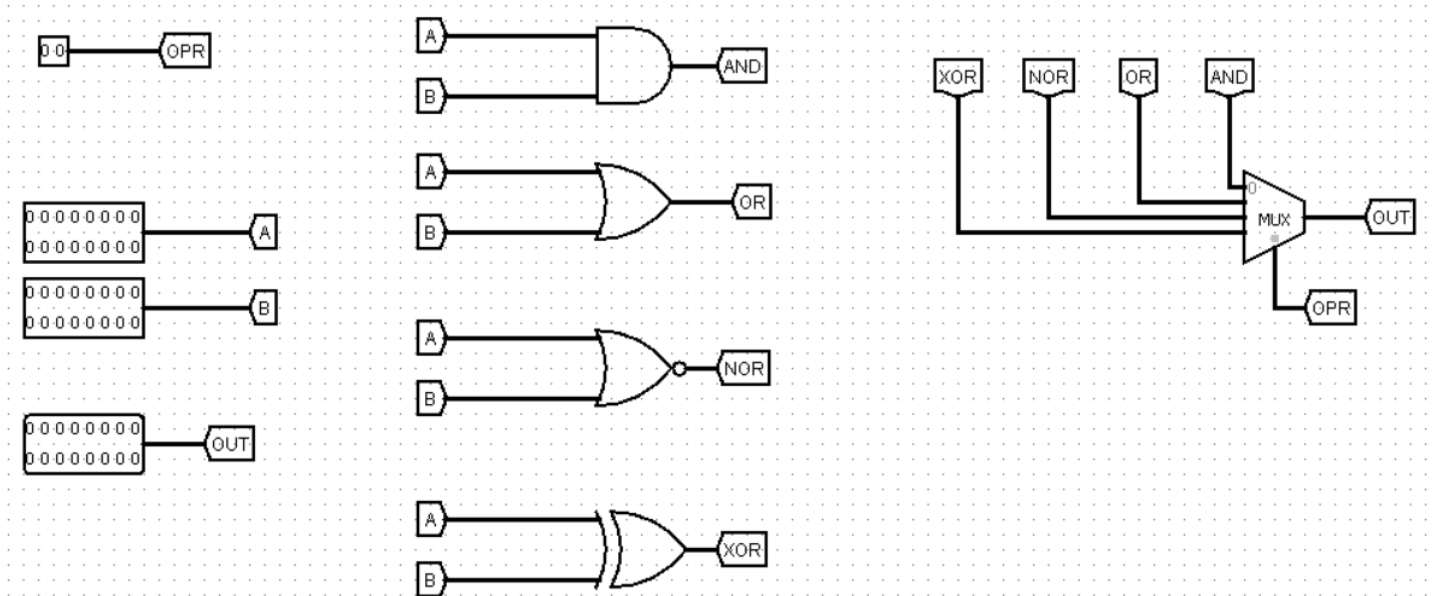


By combining these components and clever control signals, we've created a powerful AU that can perform various arithmetic operations on 16-bit numbers. It's like having a tiny, efficient calculator built right into the processor!



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.



2.4.3 Shift Unit

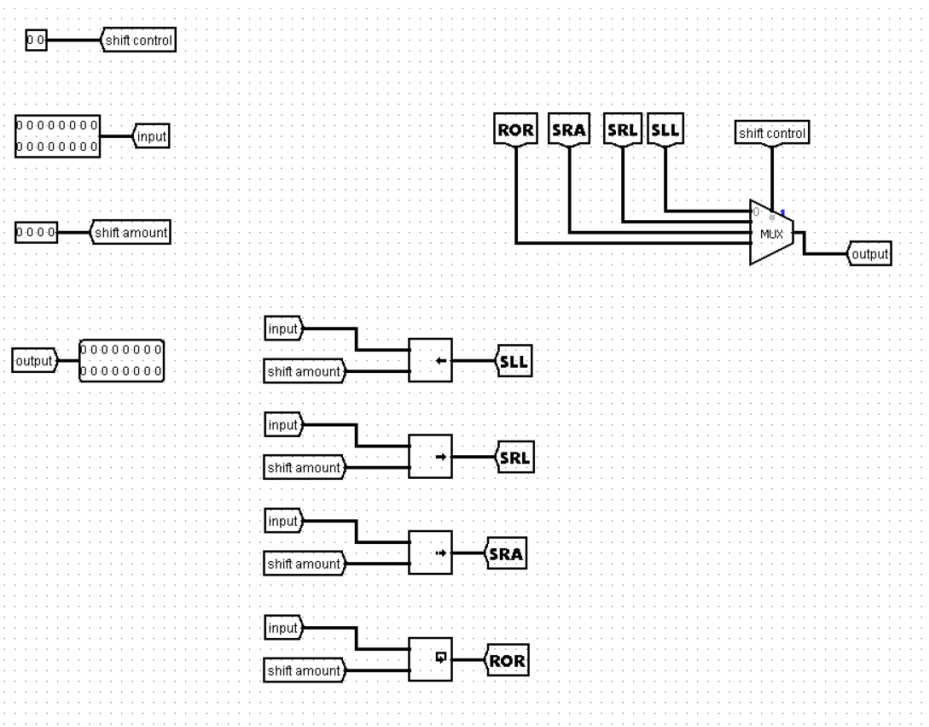
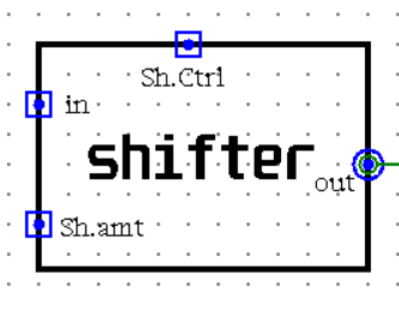
The Shift Unit focuses on one key task: **moving data bits around**.

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.



Now, let's see how they work together!

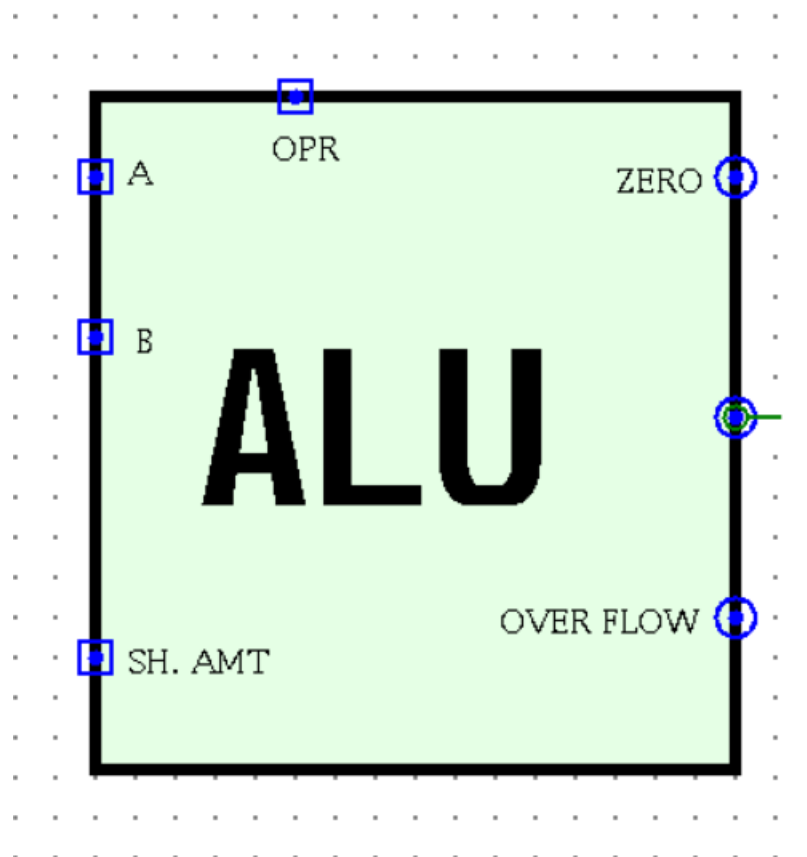
1. The Units Get Their Instructions: Each unit (AU, LU, and Shift) receives a special 2-bit control signal. This signal acts like a tiny code, telling the unit what specific task to perform.
2. The Shifter Gets Specific: In addition to the 2-bit control signal, the Shift Unit has a bonus input – a 4-bit 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.
3. Results Start Flowing: The AU outputs useful flags like zero and overflow to indicate specific conditions during calculations.
4. **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.
5. 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:

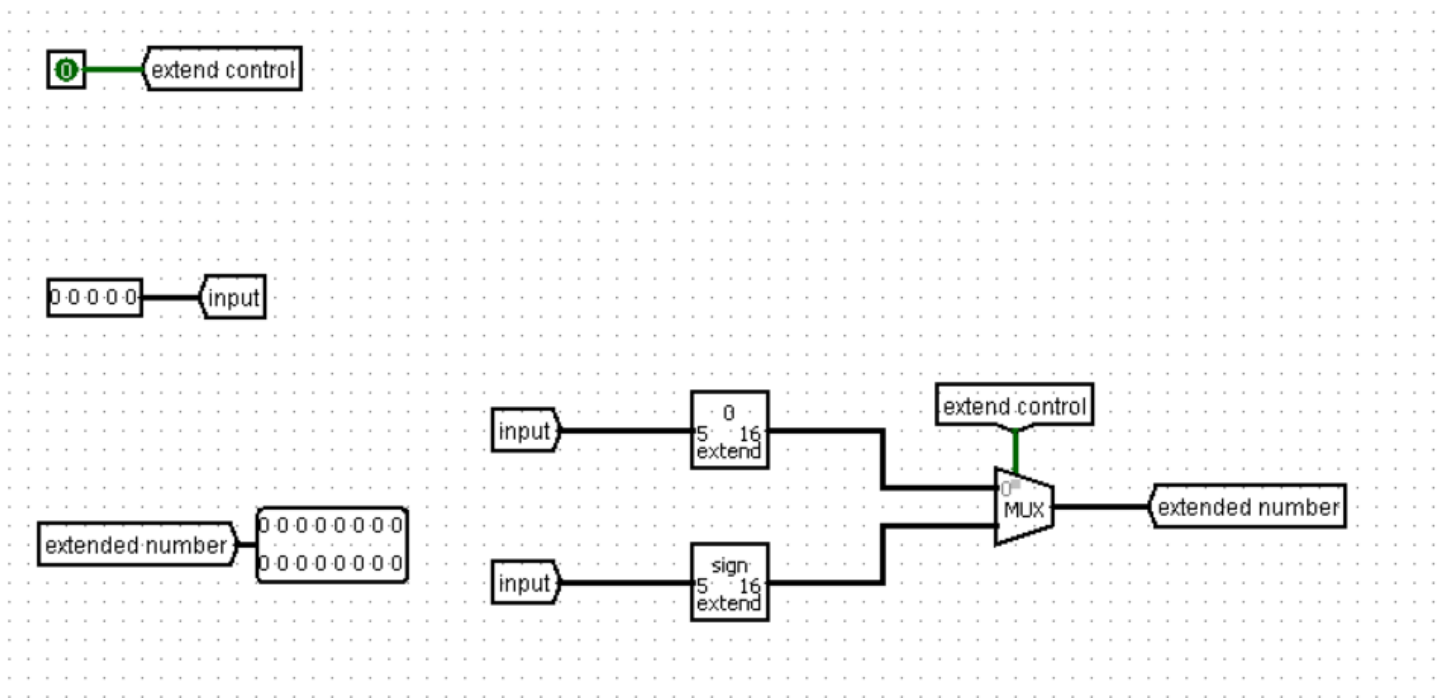
- 2 bits for the unit control signals (AU, LU, or Shift)
- 2 bits for the MUX control signal (which unit's output to use)

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

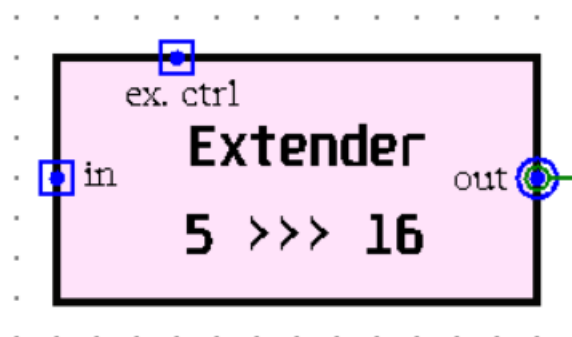
And this is our ALU:



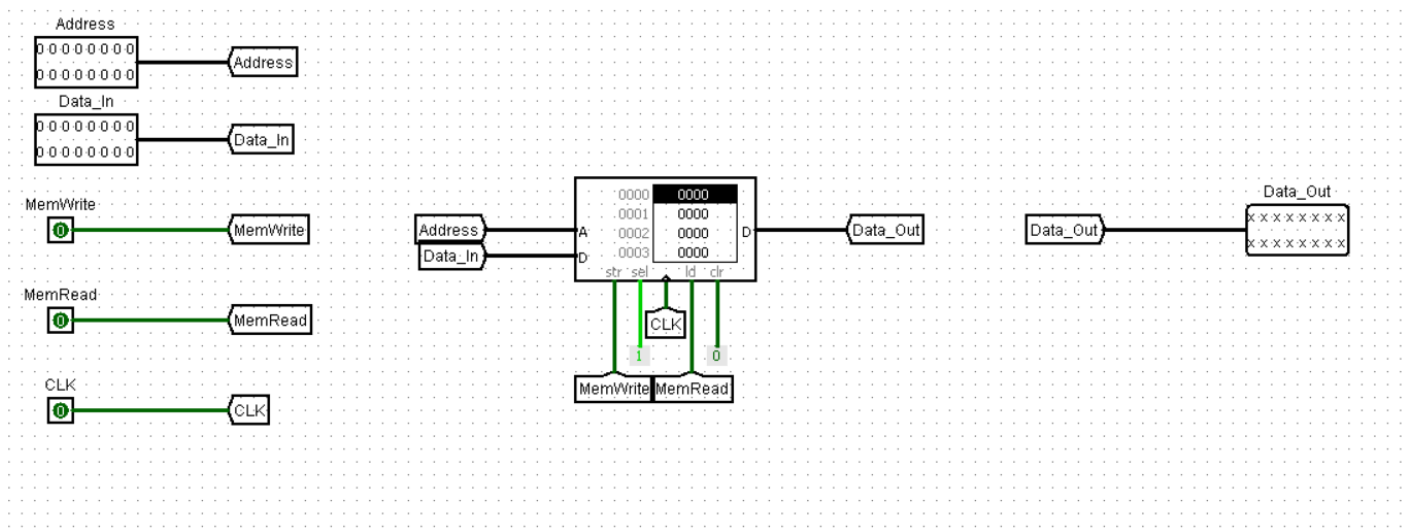
2.5 Extender:



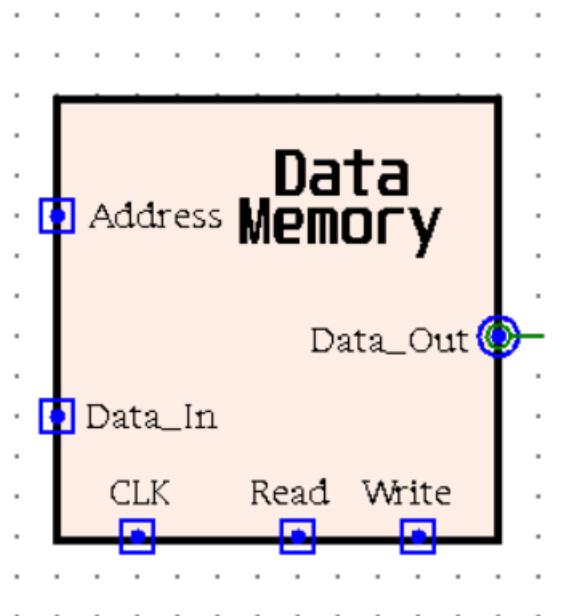
The extender is a special circuit that takes this small immediate value (5 bits in our example) and expands it to a full size (maybe 16 bits, depending on the processor). This gives the tiny value the power to contribute to more complex operations!



2.6 Data Memory:

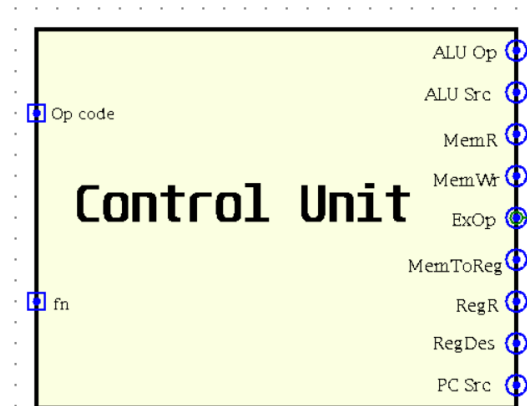


The data memory operates akin to a Random Access Memory (RAM) system, offering both read and write functionalities. It encompasses five inputs: an address pathway for accessing data, a data pathway for inputting data to be stored, signals for MemWrite (activated when writing) and MemRead (activated when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

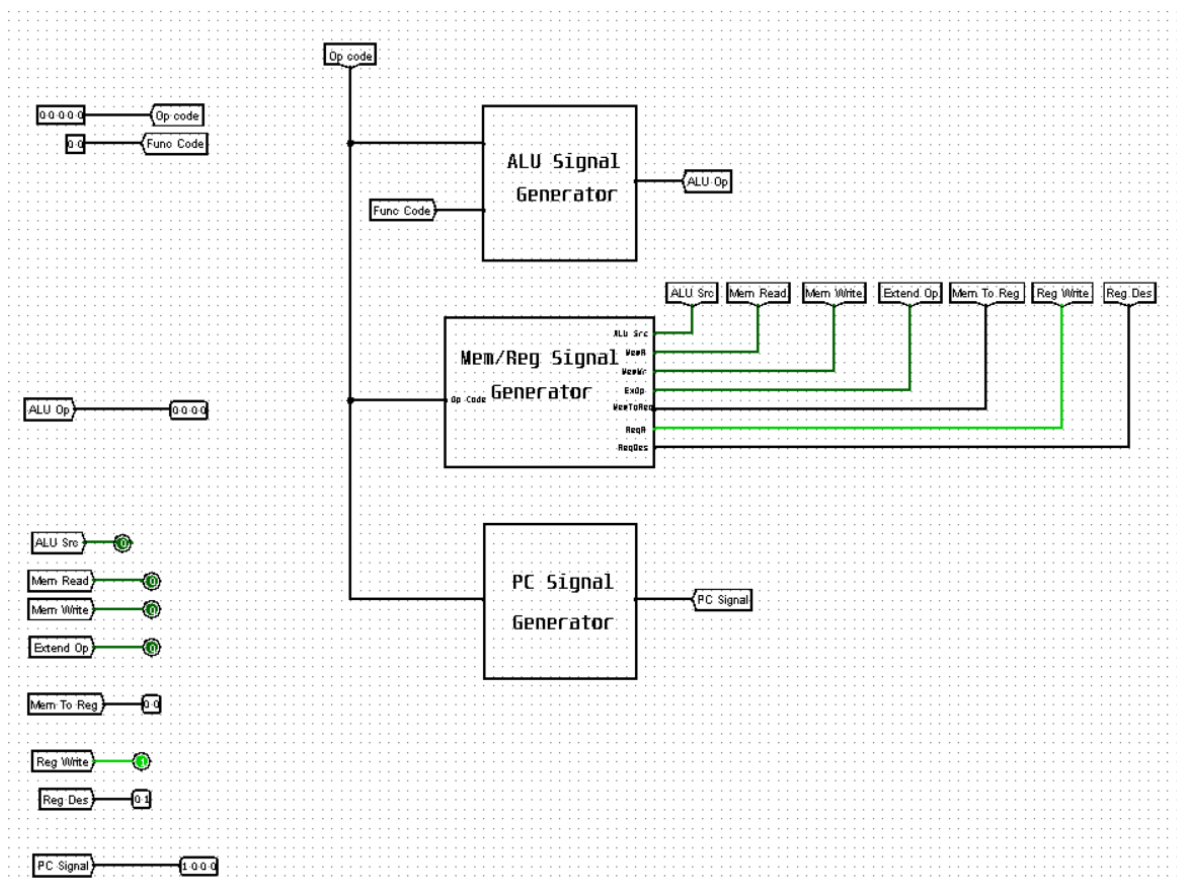


2.7. Control Unit

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



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):

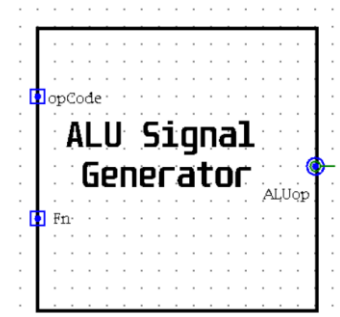
The R-type instruction format utilises the 2-bit function field to precisely control ALU operations.

7.2. Output signals

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

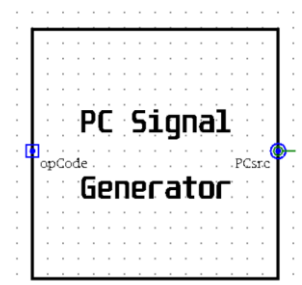
7.2.1 ALU signal generator:

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.).



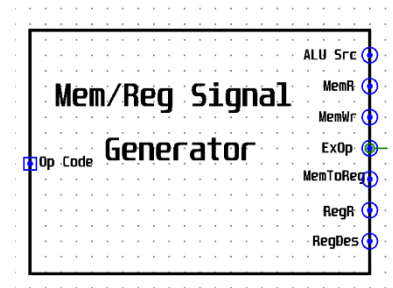
7.2.2 PC signal generator:

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

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

7.4. Signals expressions

<i>ALUop_0</i>	$\sim OP3 \sim OP2 OP0 + \sim OP3 \sim OP2 OP1 + \sim OP3 OP1 OP0 + OP3 OP2 + OP4$
<i>ALUop_1</i>	$OP3 \sim OP2$
<i>ALUop_2</i>	$\sim OP4 \sim OP3 \sim OP2 \sim OP1 F0 + \sim OP3 OP2 \sim OP1 OP0 + OP3 \sim OP2 OP0 + OP3 OP2 OP1$
<i>ALUop_3</i>	$\sim OP3 \sim OP2 \sim OP1 F1 + \sim OP2 OP1 OP0 + \sim OP3 OP2 OP1 \sim OP0 + OP3 \sim OP2 OP1 + OP4$

<i>PCsrc_0</i>	$\sim OP4 \sim OP3 \sim OP2 OP1 + \sim OP4 OP3 OP2 OP1 + OP4 \sim OP1$
<i>PCsrc_1</i>	$OP3 OP2 OP1 + OP4 \sim OP1$
<i>PCsrc_2</i>	$OP0$
<i>PCsrc_3</i>	$\sim OP1$

<i>RegDst_0</i>	$\sim OP3 \sim OP2 \sim OP1 + OP4 OP3$
<i>RegDst_1</i>	$OP4$

<i>MemToReg_0</i>	$OP3 OP2$
<i>MemToReg_1</i>	$OP4$

<i>ALUsrc</i>	$\sim OP3 OP2 + OP3 \sim OP1 + OP4 OP1$
<i>Ext_control</i>	$OP1 OP0 + OP3$
<i>RegWrite</i>	$\sim OP4 \sim OP3 \sim OP1 + \sim OP4 \sim OP1 \sim OP0 + \sim OP4 \sim OP3 OP2 + \sim OP4 OP3 \sim OP2 + OP4 \sim OP3 \sim OP2 OP1 \sim OP0 + OP4 OP3 OP2 OP1 OP0$
<i>MemWrite</i>	$\sim OP4 OP3 OP2 \sim OP1 OP0$
<i>MemRead</i>	$\sim OP4 OP3 OP2 \sim OP1 \sim OP0$

3. Testing and verification

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

3.1. 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:

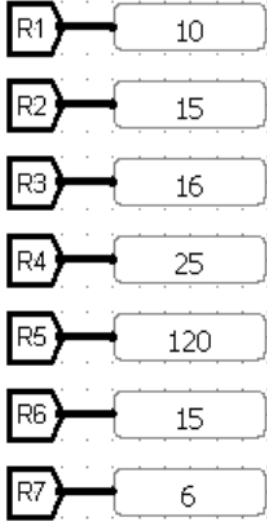
- \$1: Stores the base address of the array in memory.
- \$2: Stores the length of the array.
- \$4: Likely used as a general-purpose argument register to pass data to the functions.
- \$5: Stores the final sum returned by the `sum_array` function.

3.1.1. Assembly code and Instruction code

		Instruction code
1	main:	
2	addi \$1, \$0, 10	# save array address in \$1 3A81
3	addi \$2, \$0, 15	# save array length in \$2 3BC2
4		
5	add \$4, \$0, \$1	# 0901
6	jal init_array	# initialize array F804
7		
8	add \$4, \$0, \$1	# 0901
9	jal sum_array	# sum array F808
10		
11	j exit	# jump exit to end F00F
12		
13	init_array:	
14	addi \$3, \$0, 1	# initialize counter 3843
15		
16	init_loop:	
17	sw \$3, 0(\$4)	# store array element 6823
18	addi \$3, \$3, 1	# increment counter 385B
19	addi \$4, \$4, 1	# increment address 3864
20	bge \$2, \$3, init_loop	# loop until counter is greater than array len 8F53
21	jr \$7	# 1038
22		
23	sum_array:	
24	addi \$3, \$0, 1	# initialize counter 3843
25	add \$5, \$0, \$0	# initialize sum output 0940
26		
27	sum_loop:	
28	lw \$6, 0(\$4)	# load array element 6026
29	add \$5, \$5, \$6	# add to previous elements 096E
30	addi \$3, \$3, 1	# increment counter 385B
31	addi \$4, \$4, 1	# increment address 3864
32	bge \$2, \$3, sum_loop	# loop until counter is greater than array len 8F13
33	jr \$7	# 1038
34		
35	exit:	
36	j exit	# Terminat program F000

3.1.2. Registers Expected vs Actual values

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



Test Pass

3.1.3. Memory Expected vs Actual values

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

Test Pass

3.2 Test No.2:

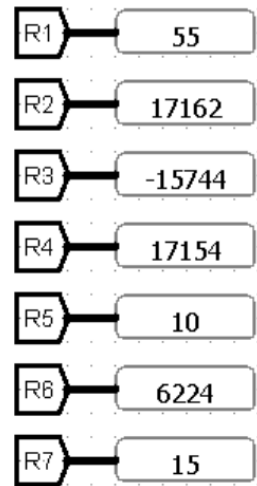
The testing process of the test code sent by Eng. Jihad.

3.2.1. Assembly code and Instruction code

		Instruction code
1	#	9384
2 Lui 900	#	3B4D
3 Addi R5, R1,13	#	04CD
4 Xor R3, R1, R5	#	6001
5 Lw R1, 0(R0)	#reg[1] <- mem[0]	6042
6 Lw R2, 1(R0)	#reg[2] <- mem[1]	6083
7 Lw R3, 2(R0)	#reg[3] <- mem[2]	3AA4
8 Addi R4, R4, 10	#	0B24
9 Sub R4, R4, R4	#	0914
10 L2: Add R4, R2, R4	#	0D93
11 Slt R6, R2, R3	#	70F0
12 Beq R6, R0, L1	#	088A
13 Add R2, R1, R2	#	7700
14 Beq R0, R0, L2	#	6804
15 L1: Sw R4, 0(R0)	#mem[0] <- reg[4]	F804
16 Jal func	#	4193
17 Sll R3, R2, 6	#	58DE
18 L4:ROR R6, R3, 3	#	77C0
19 beq r0,r0,l4	#program is over, keep looping back to here	0353
20 func: or R5, R2, R3	#	6001
21 Lw R1, 0(R0)	#	614A
22 Lw R2, 5(R1)	#	618B
23 Lw R3 ,6(R1)	#	0113
24 And R4, R2, R3	#	6804
25 Sw R4, 0(R0)	#mem[0] <- reg[4]	1038
26 Jr R7	#	

3.2.2. 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

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

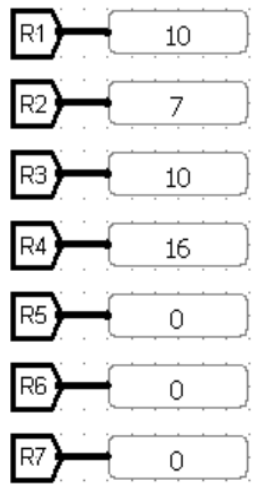
This code implements mechanisms to ensure the proper execution of store, jump, and branch instructions.

3.3.1. Assembly code and Instruction code

			Instruction code
1	#		
2	addi r1, r0, 0	# r1 = a = 0	3801
3	addi r2, r0, 7	# r2 = b = 7	39C2
4	addi r3, r0, 10	# r3 = 10	3A83
5			
6	loop:		
7	bge r1, r3, exit	# if a >= 10 exit	894B
8	add r4, r1, r2	# a + b	090A
9	sw r4, 0(r1)	# mem[a] = a + b	680C
10	addi r1, r1, 1	# a += 1	3849
11	j loop	# repeat	F7FC
12			
13	exit:		
14	j exit	# end program	F000

3.3.2. 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

3.3.3. 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

3.4. 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.

3.4.1. Assembly code and Instruction code

			Instruction code
1	main:	#	
2	addi r1, r0, 1	# arrA address	3841
3	addi r2, r0, 11	# arrB address	3AC2
4			
5	addi r3, r0, 8	#	3A03
6	sw r3, -1(r1)	# arrA len	6FCB
7			
8	addi r3, r3, -1	#	3FDB
9	sw r3, -1(r2)	# arrB len	6FD3
10			
11	add r5, r0, r1	#	0941
12	jal init_array	# initialize arrA	F80E
13			
14	add r5, r0, r2	#	0942
15	jal init_array	# initialize arrB	F80C
16			
17	addi r4, r0, 3	# x value	38C4
18			
19	lui 1619	#	9365
20	ori r3, r1, 9	#	2A4B
21			
22	addi r5, r4, 2	#	38A5
23	add r5, r5, r2	#	096A
24	lw r5, 0(r5)	# r5 = B[x+2]	602D
25			
26			
27	or r5, r5, r3	#	036B
28			
29	addi r6, r4, 3	#	38E6
30	add r6, r6, r1	#	09B1
31	sw r5, 0(r6)	# A[x+3] = r5	6835
32	j exit	#	F008
33			
34	init_array:		
35	add r3, r0, r0	# initialize counter	08C0
36	lw r4, -1(r5)	# load array size	67EC
37	init_loop:		
38	sw r3, 0(r5)	# store array element	682B
39	addi r3, r3, 1	# increment counter	385B
40	addi r5, r5, 1	# increment address	386D
41	blt r3, r4, init_loop	# loop if not reached array size	875C
42	jr r7	#	1038
43			
44	exit:		
45	j exit	#	F000

Test Pass

3.4.2. Registers Expected vs Actual values

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

3.3.3. 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

3.5 Multiplication and Applications Test:

3.5.1 Multiplication Function:

The objective of the implementation of the multiplication function is to be able to use it 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

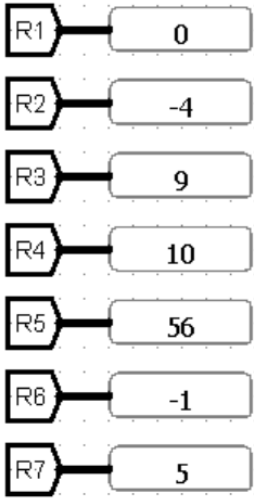
Memory Inputs

```
1  # Multiplication Function
2  # $6 is used as a virtual stack pointer so ITS VALUE IS NOT PRESERVED!
3  # Uses $5 as the address that contain the input 0($5), 1($5) are the inputs
4  # Uses $4, $3 to calculate the output, $2 as a counter but their values are preserved
5  # The output is returned in $5
6
7
8  # Set $2, $3, $4 with random values
9  addi $2, $0, -4
10 addi $3, $0, 9
11 addi $4, $0, 10
12
13 # call mul function
14 addi $5, $0, 6
15 jal mul
16
17 exit: j exit
18
19 mul:
20
21 # Set the stack pointer at the last position in the memory
22 addi $6, $0, -1
23
24 # Save Preserved values
25 sw $4, 0($6)
26 sw $3, -1($6)
27 sw $2, -2($6)
28
29 # Load inputs
30 lw $4, 0($5)
31 lw $3, 1($5)
32
33 addi $2, $0, 0
34 addi $5, $0, 0
35
36 loop:
37 beq $2, $4, return
38 add $5, $5, $3
39 addi $2, $2, 1
40 j loop
41
42 return:
43
44 # Load Preserved value
45 lw $4, 0($6)
46 lw $3, -1($6)
47 lw $2, -2($6)
48
49 jr $7
```

v2.0 raw	v2.0 raw
3985	0000
3F02	0000
3A43	0000
3A84	0000
F802	0000
F000	0008
3FC6	0007
6834	
6FF3	
6FB2	
602C	
606B	
3802	
3805	
7114	
096B	
3852	
F7FD	
6034	
67F3	
67B2	
1038	

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

3.5.2 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

```

1  # Power Function
2  # Calculate 0($0) to the power of 1($0)
3  # The output is stored in $5
4
5
6  # Load the inputs
7  lw $1, 0($0)
8  lw $2, 1($0)
9
10 # Store the default result (1)
11 addi $3, $0, 1
12 sw $3, 1($0)
13
14
15 addi $4, $0, 0
16 power_loop:
17 beq $4, $2, exit
18 addi $5, $0, 0
19 jal mul
20 sw $5, 1($0)
21 addi $4, $4, 1
22 j power_loop
23
24 exit: j exit

```

Memory Inputs

```

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

```

5.2.2. Registers Expected vs Actual values

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

3.5.3 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

```

1  # Calculate the Product of an array
2  # Inputs:
3  # $1 contains the address of the first element
4  # $2 contains the length
5  # The output is stored in $5
6
7
8  # inputs
9  addi $1, $0, 5
10 addi $2, $0, 6
11
12
13 addi $3, $0, 0
14 # Loop over length - 1
15 addi $2, $2, -1
16
17 product_loop:
18 beq $3, $2, exit
19
20 add $4, $1, $3
21
22 addi $5, $4, 0
23 jal mul
24
25 sw $5, 1($4)
26 addi $3, $3, 1
27
28 j product_loop
29
30 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

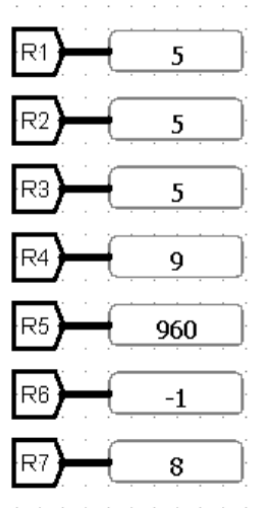
```

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

```

5.3.2. Registers Expected vs Actual values

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



Test Pass

3. Team work

We implemented this processor using GitHub, we have done about 9 meetings since we started on the 1st of this month.

3.1. Work distribution chart

