Institute of
Business Administration
Karachi

Leadership and Ideas for Tomorrow

16-Bit CPU Design and Implementation Guide

By Abdulllah Tariq

December 12, 2024

Contents

1 Introduction

1.1 Purpose

The purpose of this guide is to provide a comprehensive overview of the design and implementation of a 16-bit CPU. This guide will cover the fundamental components of a CPU, including the arithmetic logic unit (ALU), control unit, memory, and input/output interfaces. By following this guide, you will gain a deeper understanding of how CPUs work and how to design and build your own 16-bit CPU. This will be a step-by-step guide, starting from the basic building blocks of a CPU and gradually progressing to more complex components. Each section will focus on a specific aspect of CPU design, providing detailed explanations, diagrams, and examples to help you understand the concepts and principles involved. By the end of this guide, you will have the knowledge and skills to design and implement a fully functional 16-bit CPU in a digital circuit simulator.

1.2 Prerequisites

I have tried to explain most of the concepts in detail, but it is recommended to have a basic understanding of digital logic, binary arithmetic, and computer architecture. Familiarity with logic gates, binary numbers, and basic arithmetic operations will be beneficial in understanding the concepts presented in this guide. Additionally, experience with digital circuit design tools such as Logisim or similar software will be helpful in implementing the CPU components.

1.3 Tools

For this guide, we will be using Logisim Evolution, a free and open-source digital design tool that allows you to create and simulate digital circuits. Logisim provides a user-friendly interface for designing circuits using logic gates, flip-flops, multiplexers, and other digital components. You can download Logisim from the following link: `https://github.com/logisim-evolution/logisim-evolution/releases/tag/v3.9.0`.

1.4 Overview

We will following the following architecture for our 16-bit CPU:

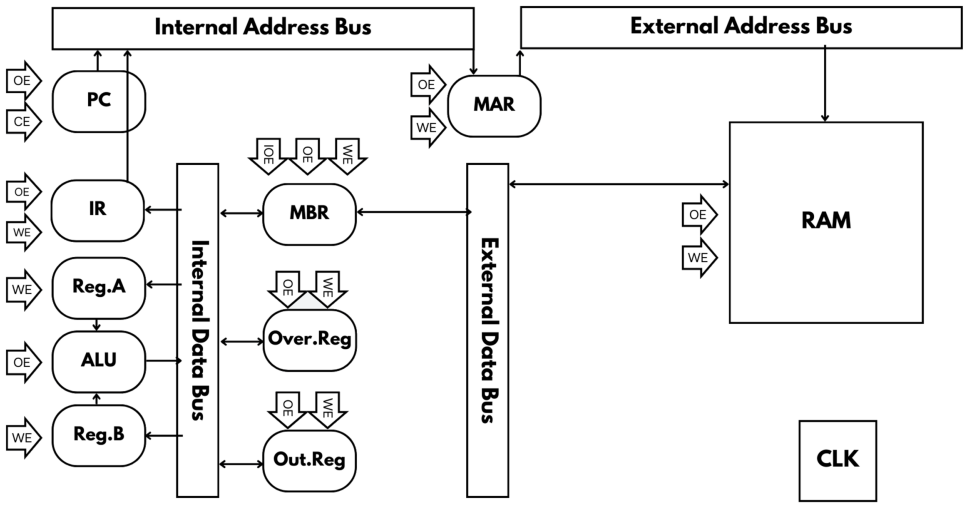

Figure 1: 16-bit CPU Architecture

If you want to go even simpler first then you can start with a 8 bit CPU and then move to 16 bit CPU. The architecture of a 8 bit CPU is same as the 16 bit CPU but the components are half in number.
The link for the 8 bit CPU guide is: `https://www.youtube.com/playlist?list=PL26O2JyrmxV7CWzV9hCCJWMyiOlVMIF8z`.

2 16-Bit Adder/Subtractor

Our first component for the CPU is the **16-bit Adder/Subtractor**. This component, commonly referred to as a *full adder*, is responsible for performing both addition and subtraction operations. We will build up to the 16-bit version, starting from fundamental concepts such as bit addition, overflow, and two's complement subtraction, and ending with the construction of a full adder.

2.1 Bit Addition

Binary addition is the foundation of all arithmetic operations in digital systems. In binary, each bit position can hold a value of either 0 or 1. When two bits are added, the result produces a sum and, if the sum exceeds 1, a carry bit.

Bit A	Bit B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 1: Truth Table for Single-Bit Addition

```
Example:
1 + 1 = 0 (Sum), Carry = 1
1 + 0 = 1 (Sum), Carry = 0
```

2.2 Overflow

Overflow occurs when the sum of two binary numbers exceeds the range that can be represented by the available bits. For unsigned addition, overflow is detected when a carry is generated in the most significant bit (MSB).

```
Example:
Binary addition of 1111 (15 in decimal) + 1 results in overflow as the sum is 10000.
```

2.3 Signed Bit Representation

In signed bit representation, the most significant bit (MSB) represents the sign of the number (0 for positive, 1 for negative).

```
Example:
In 8-bit signed numbers, 1111 1111 represents -1, while 0111 1111 represents +127.
```

2.4 One's Complement

One's complement is another method of representing negative binary numbers, where each bit is inverted.

```
Example:
Binary 5 = 00000101, one's complement of -5 = 11111010.
```

2.5 Two's Complement

Two's complement is the most common method of representing signed binary numbers and is used to handle negative numbers in digital circuits.

```
Example:
Binary 5 = 00000101, two's complement of -5 = 11111011 (invert bits and add 1).
```

2.6 Half Adder

A half adder is the most basic building block in digital arithmetic. It takes two single-bit binary inputs and produces a sum and a carry output.

Input A	Input B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 2: Truth Table for Half Adder

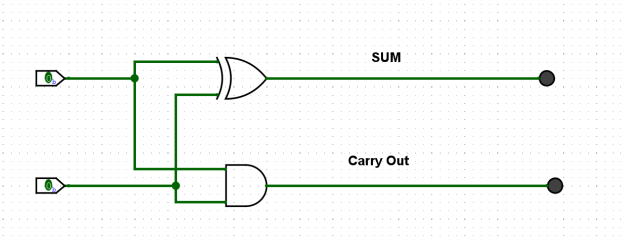

Figure 2: Circuit Diagram for a 2-bit Half Adder

As shown in the truth table, the half adder can only add two single-bit numbers and lacks the ability to account for any carry-in from previous bits, which is why it is called a ”half” adder.

2.7 Full Adder

The full adder builds upon the half adder by including an additional input for a carry-in bit, allowing it to add three single bits (two input bits plus a carry-in). The outputs are a sum and a carry-out.

Input A	Input B	Carry-in	Sum	Carry-out
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Table 3: Truth Table for Full Adder

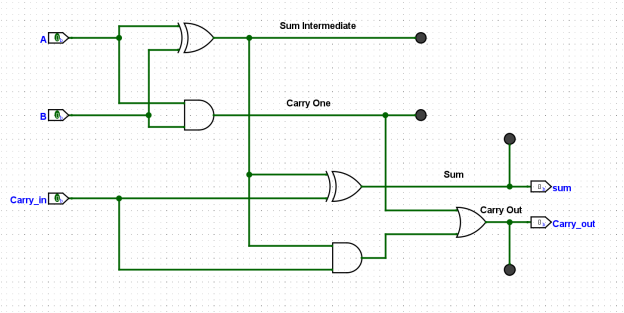

Figure 3: Circuit Diagram for a 2-bit Full Adder

The full adder can manage the carry bit from the previous addition, making it suitable for multi-bit addition.

2.8 4-bit Full Adder

To extend the addition capability, we can combine four full adders to create a 4-bit full adder. Each bit from the inputs A and B is added with the corresponding carry-in from the previous bit.

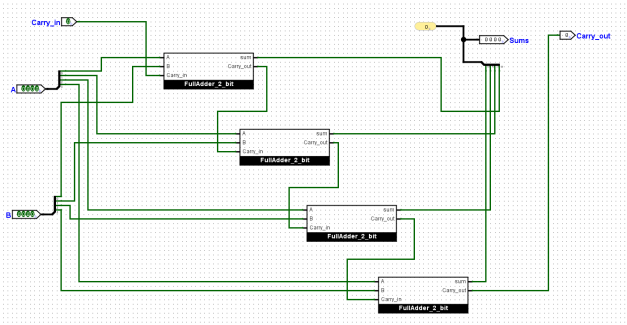

Figure 4: Circuit Diagram for a 4-bit Full Adder

The 4-bit full adder outputs a 4-bit sum and a final carry-out bit. This layout will serve as the foundation for constructing higher-bit adders.

2.9 16-bit Full Adder

Finally, by combining four 4-bit adders, we can construct a 16-bit full adder capable of handling larger binary numbers. This 16-bit adder is critical for our CPU, as it allows operations on 16-bit data. Below you can see that I have made a third output which is called ”Negative”, it is because if two negative numbers are added the answer is also negative, but since we have only 16 bits to display the answer their will be a overflow bit so I merged that $Carry_{Out}$ with the 16 bit Sum to make a 17 bit number which accurately displays the negative numbers.

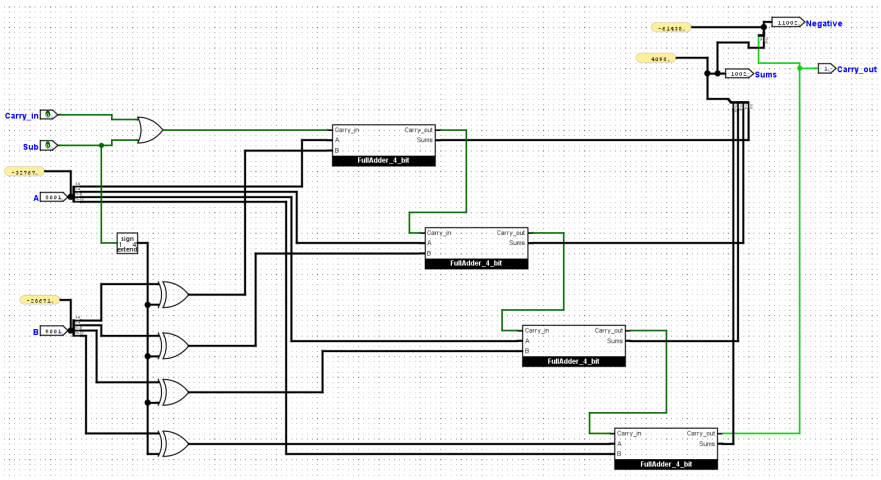

Figure 5: Circuit Diagram for a 16-bit Full Adder

The 16-bit full adder works by connecting the carry-out of each 4-bit adder to the carry-in of the next, creating a seamless addition across all 16 bits.

2.10 Adding Subtraction Capability

To enable subtraction, we can use the two's complement method. By inverting the bits of B and adding 1 (by setting the initial carry-in to 1), we effectively perform $A - B$ as $A + (-B)$.

- **Step 1:** Invert all bits of B.
- **Step 2:** Set the carry-in of the least significant bit (LSB) to 1.
- **Step 3:** Add A and $-B$.

```
Example:
To subtract 5 from 9 in binary:
9  = 1001
5  = 0101
Two's complement of 5: 1011 (invert and add 1)
9 + (-5) = 1001 + 1011 = 10100 (result: 4, discarding overflow bit)
```

This configuration allows the 16-bit adder to perform both addition and subtraction based on the initial carry and inverted input values, making it a versatile component for our CPU.

As shown in the diagram, each bit in 'B' is first passed through an XOR gate before entering the adder. This is because we need the ability to perform both addition and subtraction within the same circuit. The XOR gates act as selectors: they control whether 'B' is used as-is (for addition) or inverted (for subtraction) depending on the state of the control signal called `Sub`.

The `Sub` signal operates as follows:

- When `Sub` is set to 0, the XOR gates output the bits of 'B' without any change, allowing for straightforward addition.
- When `Sub` is set to 1, the XOR gates invert each bit of 'B'. This inversion is part of the two's complement operation, which, along with setting the initial carry-in to 1, allows us to perform subtraction.

Since 'B' is a 16-bit value, it needs to be processed in segments, with each segment of 4 bits feeding into the 4-bit adders in sequence. Each 4-bit segment of 'B' is paired with a corresponding 4-bit input to the XOR gates, controlled by the `Sub` signal. However, `Sub` is only a single bit. To align it with each 4-bit segment, we use a **Bit Extender**.

The Bit Extender replicates the `Sub` bit, effectively converting it from a single-bit signal to a 4-bit signal. This allows each XOR gate within a 4-bit segment to consistently receive the same value from `Sub`, ensuring uniform addition or subtraction across all 16 bits of 'B'.

In summary:

- Each bit in 'B' passes through an XOR gate to determine whether it will remain unchanged or be inverted, based on the `Sub` control signal.
- The 16 bits of 'B' are divided into four 4-bit segments, which are processed by the corresponding 4-bit adders.
- The `Sub` bit is extended to 4 bits using a Bit Extender, ensuring that all XOR gates receive the same control input within each 4-bit segment.

3 Comparator

3.1 Definitions

The comparator is a crucial component in our CPU, responsible for determining the relationship between two binary values, A and B. It evaluates three primary conditions:

- $A < B$: The comparator will output a signal indicating that A is less than B.
- $A = B$: The comparator will output a signal indicating that A is equal to B.
- $A > B$: The comparator will output a signal indicating that A is greater than B.

We will start with a simple 2-bit comparator and progressively build up to a 4-bit and, ultimately, a 16-bit comparator.

3.2 2-bit Comparator

A 2-bit comparator takes two 2-bit binary numbers, A and B, and compares them. It produces three outputs: $A < B$, $A = B$, and $A > B$.

A (Binary)	B (Binary)	A < B	A = B	A > B
00	00	0	1	0
00	01	1	0	0
00	10	1	0	0
00	11	1	0	0
01	00	0	0	1
01	01	0	1	0
01	10	1	0	0
01	11	1	0	0

Table 4: Partial Truth Table for 2-bit Comparator

This table shows that the 2-bit comparator can correctly identify whether A is less than, equal to, or greater than B.

3.3 4-bit Comparator

The 4-bit comparator extends the comparison capability to 4-bit binary values. It combines multiple 1-bit comparators to evaluate each bit sequentially from the most significant bit (MSB) to the least significant bit (LSB).

In a 4-bit comparator, each bit of A and B is compared from highest to lowest significance. The output signals are calculated based on the values of the preceding bits.

A (Binary)	B (Binary)	A < B	A = B	A > B
0000	0000	0	1	0
0000	0001	1	0	0
0000	0010	1	0	0
0000	0011	1	0	0
0001	0000	0	0	1
0001	0001	0	1	0
0001	0010	1	0	0
0001	0011	1	0	0

Table 5: Partial Truth Table for 4-bit Comparator

For brevity, only the first 8 rows of the truth table are shown. The comparator logic extends similarly for other combinations.

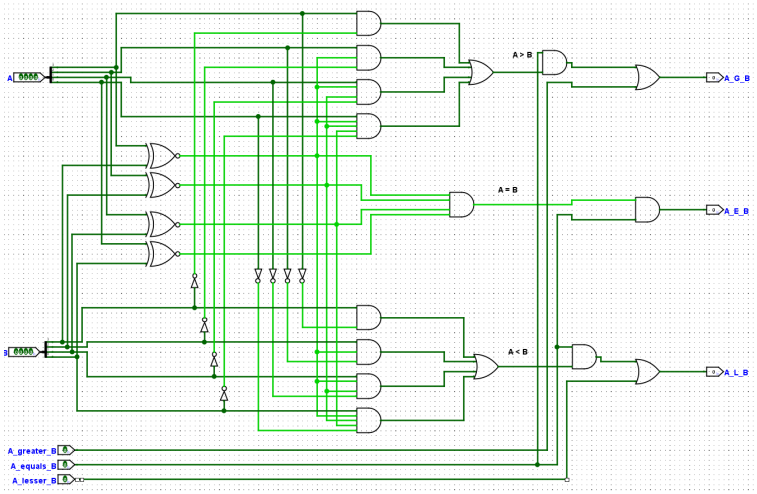

Figure 6: Circuit Diagram for a 4-bit Comparator

Here you can observe that three outputs are made such that:

- $A = B$: Each A's bit is compared to the equivalent bit of B by an XNOR gate. All four outputs become the input of an AND gate, which determines if all bits are equal or not.
- $A > B$: Similarly, for this comparison, A's each bit is compared to B's each negated bit, and all outputs become the input of an OR gate. The equation is given by:

$$A > B \iff (A_4 \wedge \overline{B_4}) \vee (A_3 \wedge \overline{B_3}) \vee (A_2 \wedge \overline{B_2}) \vee (A_1 \wedge \overline{B_1})$$

- $A < B$: For this comparison, each bit of A is compared to the equivalent bit of B, where B's bits are negated, and an OR gate is used. The equation can be represented as:

$$A < B \iff (\overline{A_4} \wedge B_4) \vee (\overline{A_3} \wedge B_3) \vee (\overline{A_2} \wedge B_2) \vee (\overline{A_1} \wedge B_1)$$

I'll explain the upper four AND gates and the same is applicable for the lower four AND gates: you can see that the second gate has 3 inputs and then 3^{th} has 4 inputs and so on.. that is because in order to check the number to be greater then the other we check starting from the MSB bit by bit, the condition of $A > B$ can be possible in the following four cases:

1. If A_3 = 1 and B_3 = 0
2. If A_3 = B_3 and A_2 = 1 and B_2 = 0
3. If A_3 = B_3, A_2 = B_2 and A_1 = 1 and B_1 = 0
4. If A_3 = B_3, A_2 = B_2, A_1 = B_1 and A_0 = 1 and B_0 = 0

That is why you will see the wires from the equivalent positioned bit of $A = B$ comparison to those greater than bits. This is the same for $A < B$ comparison.

Now we have all three outputs but something is not complete. It is that previous bits are also checked since we are connecting 4 four bits comparators together and the first comparator will be checking the most significant four bits of the whole 16 bit number it will send one of the three signals and we have to adjust them too so for that we will connect them like these:

- Previous $A = B$ bit to the current $A = B$ output using an AND gate, since in order for all the bits to be the same, both the previous and current bits need to be the same.
- Previous $A > B$ bit is combined with the current output using an OR gate, where the current output is first processed through an AND gate. The second input to this AND gate is the previous $A = B$ output since if all previous bits are equal and the current bit is greater, the overall condition remains true.
- Previous $A < B$ bit is processed similarly. The output is taken from the current $A < B$ comparison and fed into an OR gate, where the other input is the previous $A = B$ output. This ensures that if the most significant bits indicate a previous decision of $A < B$, any subsequent bits will not affect this decision.

Why did we join them using OR gates? That is because if the previous most significant bits are already decided to be greater than or less than, then the new ones do not matter, and thus the decision should not change. This is crucial in digital circuits as it allows for efficient comparison without the need to re-evaluate all bits once a definitive comparison has been made. This ensures that the circuit remains efficient and responsive, reducing unnecessary computations for the comparison.

In the 4-bit comparator:

- $A < B$: The output $A < B$ is set if the most significant bit where A and B differ has a 0 in A and a 1 in B.
- $A = B$: The output $A = B$ is set if all four bits of A and B match, bit by bit.
- $A > B$: The output $A > B$ is set if the most significant bit where A and B differ has a 1 in A and a 0 in B.

3.4 16-bit Comparator

The 16-bit comparator is built by cascading four 4-bit comparators. It evaluates each 4-bit segment of A and B progressively from the most significant to the least significant, preserving the priority of more significant bits in the comparison.

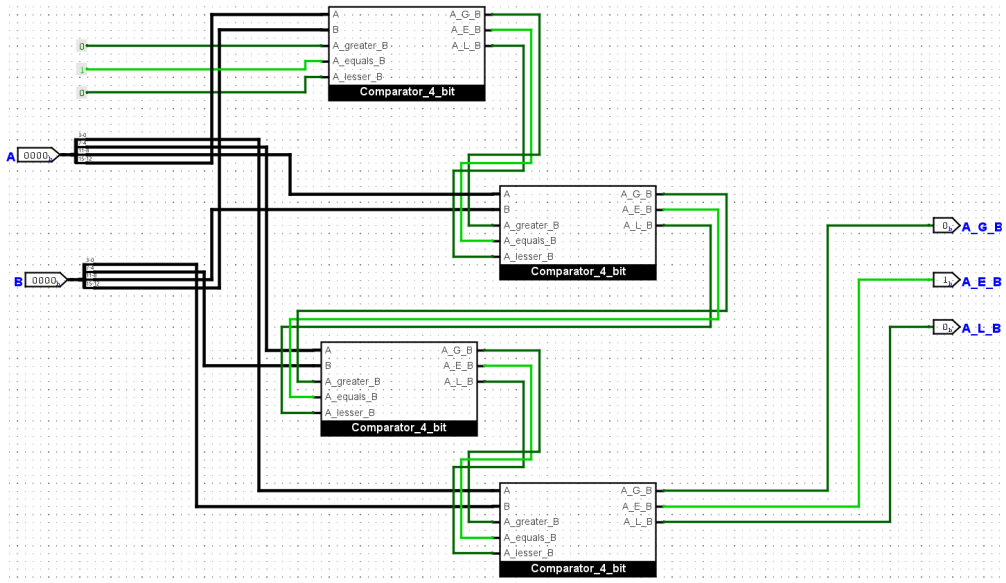

Figure 7: Circuit Diagram for a 16-bit Comparator

Here the initial inputs for $(A < 0)B$ and $(A > B)$ are always 0 and for $(A = B)$ 1 since two numbers at start will be similar until compared. The 16-bit comparator follows the same logic as the 4-bit comparator but on a larger scale:

- The highest 4 bits are compared first, and if a difference is detected (either $A > B$ or $A < B$), the comparison result for the lower bits is ignored.
- If the highest 4 bits are equal, the next 4 bits are compared, and so on.

Explanation of Comparison Outputs For each 4-bit segment:

- $A < B$: Set if the highest significant differing bit has $A = 0$ and $B = 1$.
- $A = B$: Set if all four bits are identical across A and B.
- $A > B$: Set if the highest significant differing bit has $A = 1$ and $B = 0$.

This method ensures that once a comparison result is determined in higher bits, lower bit comparisons do not change the outcome, providing a reliable 16-bit comparison suitable for our CPU. All four bit comparators are stacked an a final output is given with it being either $A > B$, $A = B$ or $A < B$.

4 Multiplier

The 8-by-8 multiplication process builds on the principles of binary multiplication, similar to 4-by-4 multiplication but with a higher bit-width. This process involves generating partial products for each bit in the multiplier and summing these partial products to get the final result. The output of an 8-bit multiplication is a 16-bit product. Why are we making a 8 by 8 multiplier? It is because it's result is 16 bit and if we were to make a 16 by 16 bit then the result would be 32 and we won't have the space to store it.

4.1 Understanding the Multiplication Process

In binary multiplication, each bit of the multiplier affects the entire multiplicand. The multiplication process consists of the following core operations:

- **Generating Partial Products**: Each bit in the multiplier multiplies the entire multiplicand, creating partial products. If the bit is '1', the entire multiplicand is included; if it's '0', it contributes nothing (or '0').

- **Shifting Partial Products**: Each partial product is shifted left based on the position of the corresponding bit in the multiplier. This is similar to multiplying by powers of two.
- **Summing Partial Products**: The final step is to sum all the shifted partial products together to obtain the final result.

4.2 Step-by-Step Process

Let's work with specific binary values for the multiplicand and multiplier.

Assume we are multiplying:

$$\text{Multiplicand} = \text{A} = 11010110_2 \quad (214_{10})$$

$$\text{Multiplier} = \text{B} = 10111011_2 \quad (187_{10})$$

4.2.1 Step 1: Generating Partial Products

Each bit in the multiplier B multiplies the entire multiplicand A, creating 8 rows of partial products. The partial product corresponds to the multiplicand A when the bit in the multiplier B is '1', and '0' otherwise.

Let's take $A = 1110$ and $B = 0101$ for a small multiplication example.

```
    1110
  * 0101
  ------
    1110      (Partial Product 1: 1110 shifted by 0)
   0000       (Partial Product 2: 1110 shifted by 1)
  1110        (Partial Product 3: 1110 shifted by 2)
+ 0000        (Partial Product 4: 1110 shifted by 3)
  ------
 1000110      (Final Sum)
```

Below is shown something similar to this but using the $8 - bit$ inputs A and B above.

Multiplier Bit	Partial Product	Shifted Partial Product
$B_0 = 1$	11010110	11010110
$B_1 = 1$	11010110	110101100
$B_2 = 0$	11010110	0000000000
$B_3 = 1$	11010110	1101011000
$B_4 = 1$	11010110	11010110000
$B_5 = 0$	11010110	000000000000
$B_6 = 1$	11010110	1101011000000
$B_7 = 1$	11010110	11010110000000

Table 6: Partial Products Generated from Multiplier

4.2.2 Step 2: Shifting Partial Products

Next, we take each of the partial products and shift them according to the position of the corresponding bit in the multiplier. The shifting aligns them for addition:

Partial Product 0:	11010110	(no shift)
Partial Product 1:	11010110	(shift left by 1)
Partial Product 2:	00000000	(shift left by 2)
Partial Product 3:	11010110	(shift left by 3)
Partial Product 4:	11010110	(shift left by 4)
Partial Product 5:	00000000	(shift left by 5)
Partial Product 6:	11010110	(shift left by 6)
Partial Product 7:	11010110	(shift left by 7)

4.2.3 Step 3: Summing Partial Products

Now we will write the shifted partial products in a column format, similar to how addition is typically done on paper. We will add these together to get the final product:

```
  11010110    (Partial Product 0)
+ 110101100   (Partial Product 1)
+ 0000000000  (Partial Product 2)
+ 1101011000  (Partial Product 3)
+ 11010110000 (Partial Product 4)
+ 000000000000 (Partial Product 5)
+ 1101011000000 (Partial Product 6)
+ 11010110000000 (Partial Product 7)
--------------------
  1001011101111100 (Final Product)
```

Thus, the product of A and B is 1001011101111100_2, which corresponds to $214 \times 187 = 40058$ in decimal.

4.3 4 by 4 Bits Multiplier

Below in the circuit, it can be seen that there are 2 four-bit inputs, A and B, and each four-bit number is split into single bits. Here, each bit in A is multiplied with every bit in B, resulting in a total of 16 AND gates. The outputs of these AND gates produce the partial products required for multiplication.

Notice that the rightmost AND gate's output is directly connected to the final output. This is because it represents the least significant bit (LSB) of the multiplication and does not need to be shifted, as explained in the 4 by 4 bits multiplication example above. Each subsequent row of AND gates generates partial products that require shifting based on the position of the corresponding bit in the multiplier B.

The partial products are then fed into an adder circuit to sum them up, resulting in the final product. The circuit is efficient as it allows for parallel computation of the products, leveraging the binary nature of multiplication.

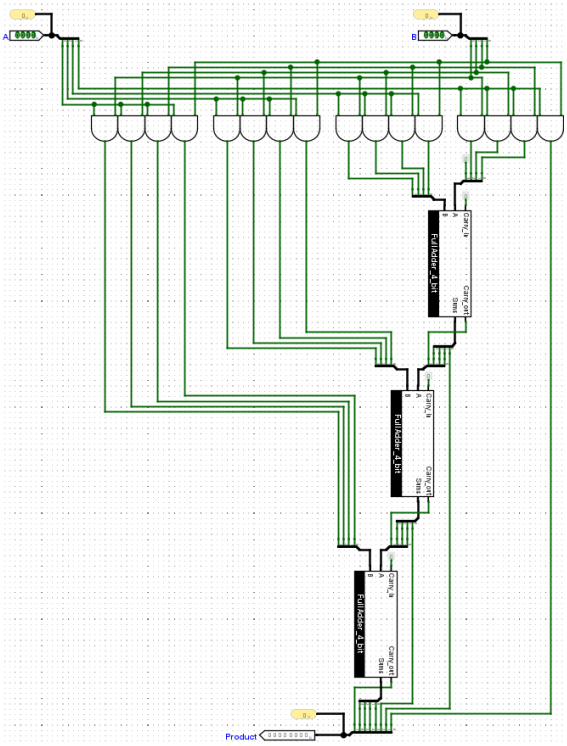

Figure 8: 4-by-4 Bit Multiplier Circuit

4.4 8 by 8 Bits Multiplier

The 8-by-8 bit multiplier circuit utilizes 4 four-bit multipliers, effectively splitting both 8-bit inputs into two 4-bit segments each: the lower part (denoted as A_{low} and B_{low}) and the higher part (denoted as A_{high} and B_{high}). This approach allows for the systematic calculation of partial products, which are then combined to produce the final 16-bit product.

- **Low and High Parts:**
  - The lower part refers to the least significant 4 bits of the 8-bit number. For instance, if $A = A_{\text{high}}A_{\text{low}}$ where $A_{\text{high}} = A_7A_6A_5A_4$ and $A_{\text{low}} = A_3A_2A_1A_0$, then $A_{\text{low}} = A_3A_2A_1A_0$.
  - The higher part refers to the most significant 4 bits of the 8-bit number. Thus, $A_{\text{high}} = A_7A_6A_5A_4$.

To compute the product of the two 8-bit numbers A and B, we generate the following partial products:

$$P_0 = A_{\text{low}} \times B_{\text{low}}$$
$$P_1 = A_{\text{low}} \times B_{\text{high}}$$
$$P_2 = A_{\text{high}} \times B_{\text{low}}$$
$$P_3 = A_{\text{high}} \times B_{\text{high}}$$

Partial Products Explained:

1. P_0: This partial product is generated by multiplying the lower 4 bits of A with the lower 4 bits of B. It represents the product of the least significant parts of both numbers.
2. P_1: This product results from multiplying the lower 4 bits of A with the higher 4 bits of B. The result of this multiplication needs to be shifted left by 4 positions (or 4 bits) because it contributes to the next higher place value in the final product.

3. P_2: Here, the higher 4 bits of A are multiplied with the lower 4 bits of B. Similar to P_1, this partial product is also shifted left by 4 positions due to its contribution to a higher place value.

4. P_3: The product of the higher 4 bits of both A and B yields this partial product. It is shifted left by 8 positions, as it contributes to the most significant bits of the final product.

Final Combination of Partial Products:
The final product P is calculated by summing all the partial products:

$$P = P_0 + (P_1 << 4) + (P_2 << 4) + (P_3 << 8)$$

This systematic approach allows for efficient multiplication of larger binary numbers while leveraging the simplicity of the four-bit multipliers. The circuit's design ensures that all calculations are performed in parallel, improving speed and efficiency.

If you observe here I have fetched the MSB (most significant bit) of the both A and B and sent them as inputs to a XOR gate, that is because if one negative and second positive numbers are multiplied the answer is negative but while adding the 16 numbers after from the 4 by 4 bit multiplication which originally had been split in two, the numbers are not recognized as negative since they are extrapolated too in this process. So in the final adder we have to sent in the carry bit to tell it that the number has a previous carry. This can be done if we use the MSB of both A and B since if they are both not the same number (example A = 1 and B = 0) the XOR gate will output 1 meaning there is a carry bit and that is true since one number is negative and the other is positive which can be identified by the MSB. THis way the the final adder incorporates the carry bit for the final correct answer.

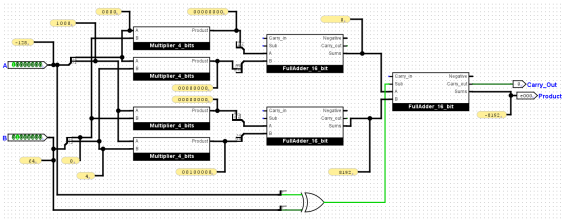

Figure 9: 8-by-8 Bit Multiplier Circuit

5 Multiplexers (MUX)

A **multiplexer (MUX)** is a digital circuit that selects one of several input signals and forwards the selected input into a single line. Essentially, a MUX acts like a digital switch, choosing data from multiple sources based on the selection lines and allowing only the selected data to pass through to the output. This device is commonly used in situations where data needs to be transmitted from multiple sources but only over a single shared line or communication channel.

5.1 Key Components and Operation of a MUX

- **Input Lines:** A MUX has multiple input lines from which it selects one as its output. Each input line can carry different data values, and the MUX's job is to decide which input should be outputted based on the control signals. The number of input lines is usually a power of 2 (e.g., 2, 4, 8, 16, etc.), allowing it to connect multiple data sources.

- **Selection Lines:** The selection lines, also known as control signals, determine which input line is connected to the output. For a MUX with 2^n inputs, there are n selection lines. Each combination of control signals corresponds to one input line, selecting it as the active output.

- **Output Line:** A MUX has a single output line where the selected input line's data appears. Only one input can be passed to the output at any time, controlled by the selection line combination.

5.2 4-to-1 1-bit Multiplexer

A **4-to-1 1-bit multiplexer** takes four different single-bit inputs (denoted as I_0, I_1, I_2, I_3 and selects one of them to be sent to a single output based on the values of two selection lines (let's call them S_0 and S_1).

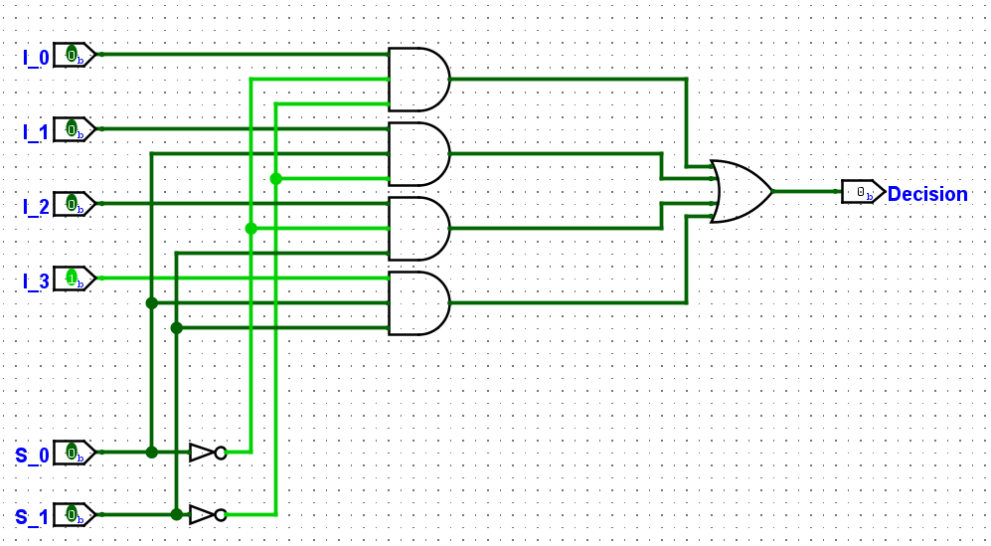

Figure 10: Circuit Diagram for a 4 to 1 1-bit Multiplexer

Inputs: The inputs are I_0, I_1, I_2, I_3.
Selection Lines: There are two selection lines S_0 and S_1, which can represent the binary values 00, 01, 10, and 11.
Output: The output line will carry the value of the selected input.
Operation: When the selection lines are set, the multiplexer routes the chosen input to the output. For instance, if $S_1S_0 = 00$, I_0 is connected to the output; if $S_1S_0 = 01$, I_1 is selected, and so forth. This allows the circuit to effectively choose one bit from multiple inputs.

5.3 4-to-1 4-bit Multiplexer

The **4-to-1 4-bit multiplexer** operates similarly to the 1-bit version but is designed to handle four 4-bit inputs (each denoted as I_0, I_1, I_2, I_3. The output is also 4 bits.

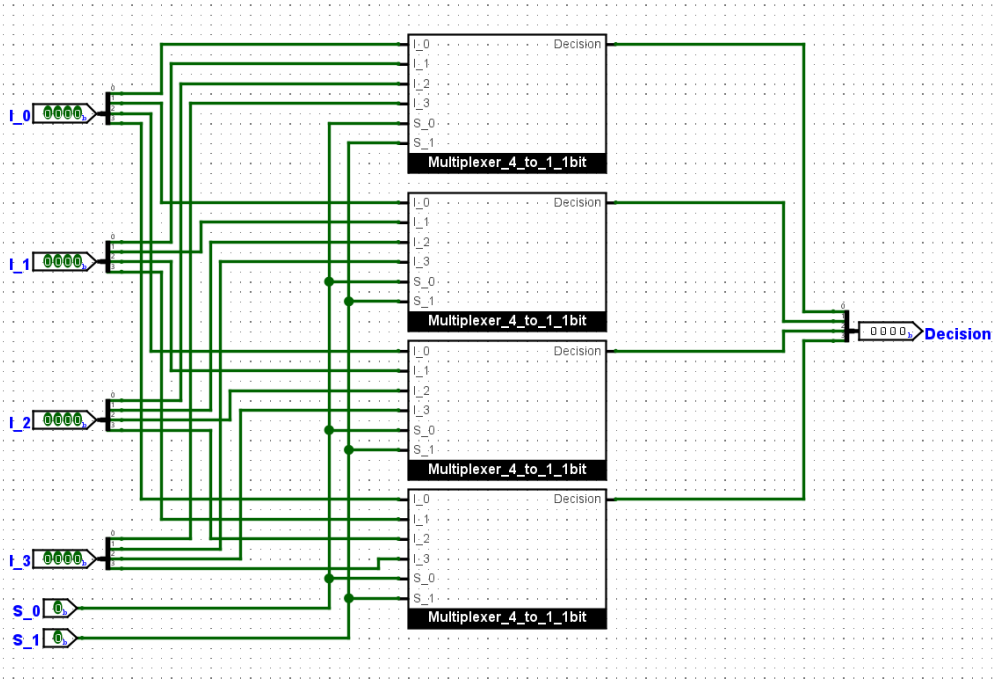

Figure 11: Circuit Diagram for a 4 to 1 4-bit Multiplexer

Inputs: The inputs are four groups of 4 bits, such as $I_0 = [I0_0, I0_1, I0_2, I0_3]$, $I_1 = [I1_0, I1_1, I1_2, I1_3]$, $I_2 = [I2_0, I2_1, I2_2, I2_3]$, and $I_3 = [I3_0, I3_1, I3_2, I3_3]$.
Selection Lines: The same two selection lines S_0 and S_1 are used.
Output: The output consists of the selected 4-bit input.
Operation: The multiplexer determines which 4-bit group to output based on the selection lines. For example, if the selection lines are set to 01, the output will be I_1 (the second group of bits). This allows data to be handled in larger chunks compared to the 1-bit multiplexer.

5.4 4-to-1 16-bit Multiplexer

The **4-to-1 16-bit multiplexer** is a more complex device that selects one of four different 16-bit inputs.

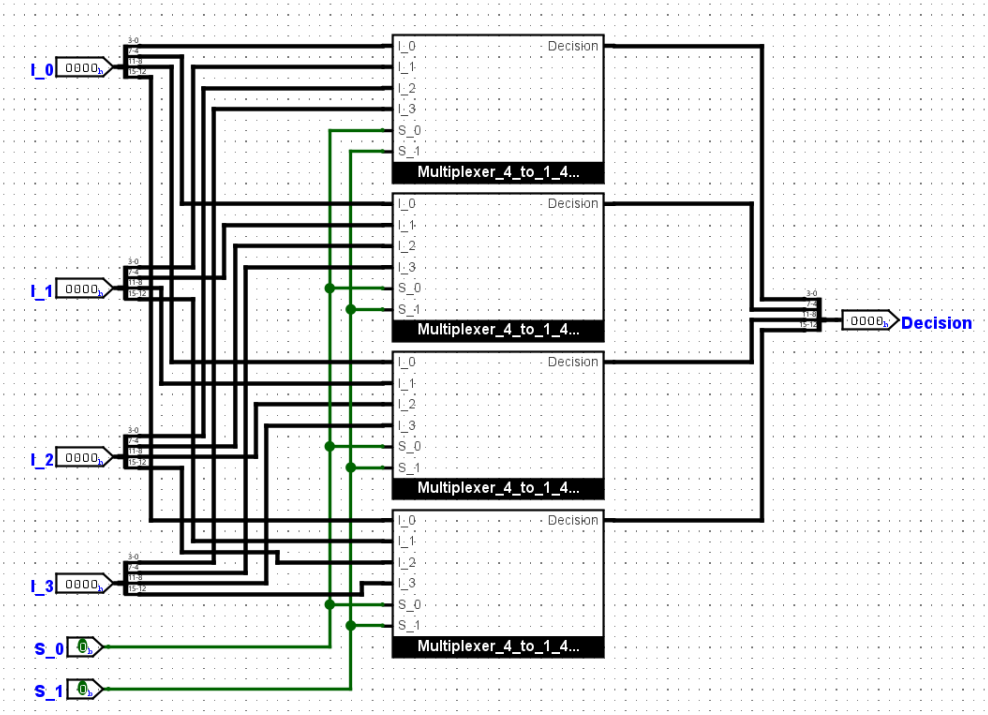

Figure 12: Circuit Diagram for a 16 to 1 16-bit Multiplexer

Inputs: The inputs are four groups of 16 bits, such as I_0 through I_3 (each I_i consists of 16 bits: $I_i = [Ii_0, Ii_1, Ii_2, \ldots, Ii_{15}]$).
Selection Lines: This multiplexer uses four selection lines (S_0, S_1), allowing for 4 possible combinations (from 00 to 11).
Output: The output is a 16-bit group from the selected input.
Operation: Depending on the combination set on the selection lines, one of the four inputs is routed to the output. For example, if $S_1S_0 = 01$, the output will be the 16-bit value from I_1. This multiplexer is particularly useful in digital circuits requiring the selection of larger data widths and multiple data sources.

5.5 Applications in ALUs

Multiplexers play a vital role in the functionality of Arithmetic Logic Units (ALUs) within digital computing systems. They allow the ALU to select different inputs for arithmetic and logical operations based on control signals. By using multiplexers, an ALU can efficiently route various data inputs to the operation units, simplifying the design and increasing flexibility. This enables the ALU to perform multiple operations without the need for extensive wiring and control logic, making the overall system more efficient.

6 AND Operation

For this section we just need to take a two 16 bits inputs and then split each input in four four bits we will have four groups of 4 bits for example: $A_0A_1A_2A_3$ and $B_0B_1B_2B_3$ and so on. Now for each group place a AND gate in front of it, its settings should be that 4 bit in and 4 bit out. After all outputs are received, merge them using a splitter together to form the final 16 bit output.

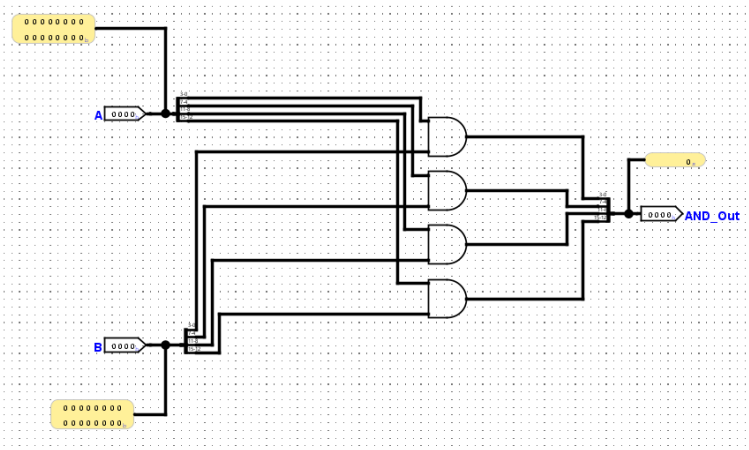

Figure 13: Circuit Diagram for a 16 bit AND operation

7 OR Operation

For this section we just need to take a two 16 bits inputs and then split each input in four four bits we will have four groups of 4 bits for example: $A_0A_1A_2A_3$ and $B_0B_1B_2B_3$ and so on. Now for each group place a OR gate in front of it, its settings should be that 4 bit in and 4 bit out. After all outputs are received, merge them using a splitter together to form the final 16 bit output.

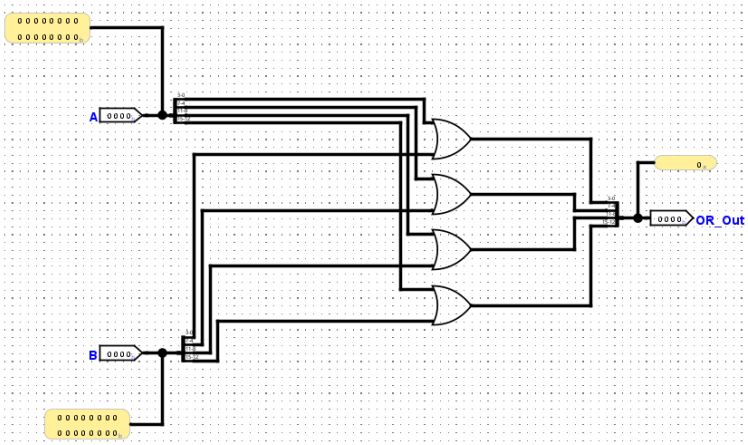

Figure 14: Circuit Diagram for a 16 bit OR operation

8 XOR Operation

For this section we just need to take a two 16 bits inputs and then split each input in four four bits we will have four groups of 4 bits for example: $A_0A_1A_2A_3$ and $B_0B_1B_2B_3$ and so on. Now for each group place a XOR gate in front of it, its settings should be that 4 bit in and 4 bit out. After all outputs are received, merge them using a splitter together to form the final 16 bit output.

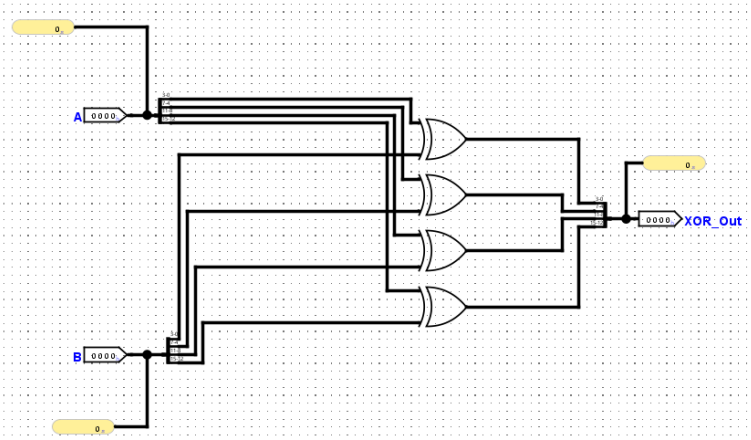

Figure 15: Circuit Diagram for a 16 bit XOR operation

9 NOT Operation

For this section we just need to take a one 16 bit input and split each bit of it, reverse it using a NOT gate then merge them using a splitter back in a 16 bit output.

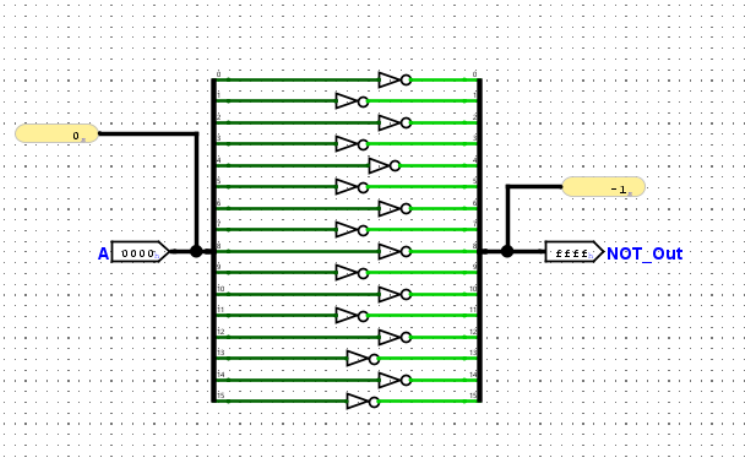

Figure 16: Circuit Diagram for a 16 bit NOT operation

10 Comparator Encoder

A comparator encoder will be used at the output of the comparator within the ALU since the output of the comparator will be three where only one of them will be active. We can encode them for now which later can decoded too:

1. $A < B = 0000\ 0000\ 0000\ 0001$
2. $(A = B) = 0000\ 0000\ 0000\ 0010$
3. $A > B = 0000\ 0000\ 0000\ 0100$

So what I did is that simply extended the one bit output signal to 16 bits and AND operated with:

1. $A < B = 0 \times 1(hex)$

2. $(A = B) = 0 \times 2(hex)$
3. $A > B = 0 \times 4(hex)$

This will provide the one of the outputs provided above as in encoded form. Then we perform OR operation on it so whichever output was active can go forward.

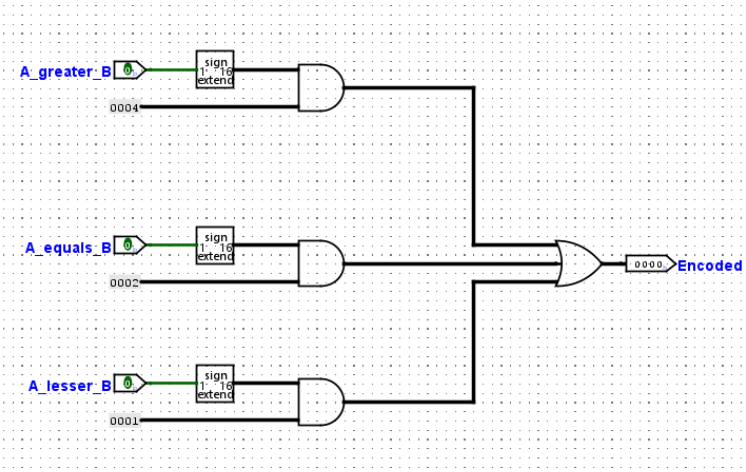

Figure 17: Circuit Diagram for a 16 bit Comparator Encoder

11 ALU

Now time to sum up all the previous components in the ALU all together. First we will connect the 16 bit inputs to all the operation circuits such as multiplier and adder / subtractor. After this each output from each operation will go in a 16 bit MUX, currently we have 8 operations so for this we will used 2 MUX's then for their outputs another MUX. As you can see below that we have used the comparator encoder on the outputs of the encoder.

We have two new things, first is the signal, as we talked earlier that the signal will decide which input should become the output similarly here we will use the signal to decide which shall become the output.

11.1 Operation Mapping

1. Adder means 0001
2. Subtractor mean 0001 (where the flag bit will be 1)
3. Multiplier means 0011
4. Comparator means 1010
5. AND mean 1011
6. XOR mean 1100
7. NOT mean 1101

The rest of the signals will be kept empty as they are for other purposes, but the main essence of this ALU and signaling is that whatever OPcode you receive is where the output of each operation should map to. Here I have not connected the OR operation circuit since I don't need to use but let's say it was needed and was mapping to 14 (1110) we could simply connect it's output to the I_2 of the fourth MUX. Here it can be seen if I have done the OR operation on both the multiplication carry and the Adder carry since at a time one operation will be performed and may cause carry the other will have 0 as it's carry hence doing OR will not change the result.

11.2 Connecting the operations inside ALU

We will connect the first four outputs to the first MUX and the last four outputs to the second MUX. The signal bits used here will be S_0S_1 since these represent the numbers from 0 to 3. After this MUX will output one one output each will be then connected to the inputs of a third MUX, we can leave the rest two empty as the fulfill no purpose currently. We will now add the signal bits: S_2S_3 as they decide the MSB inputs so this way the ALU outputs the correct operations output based on the signals which will come from the OpCode. Here in the end an OE signal is attached so the output is only getting out of the ALU on command no other way.

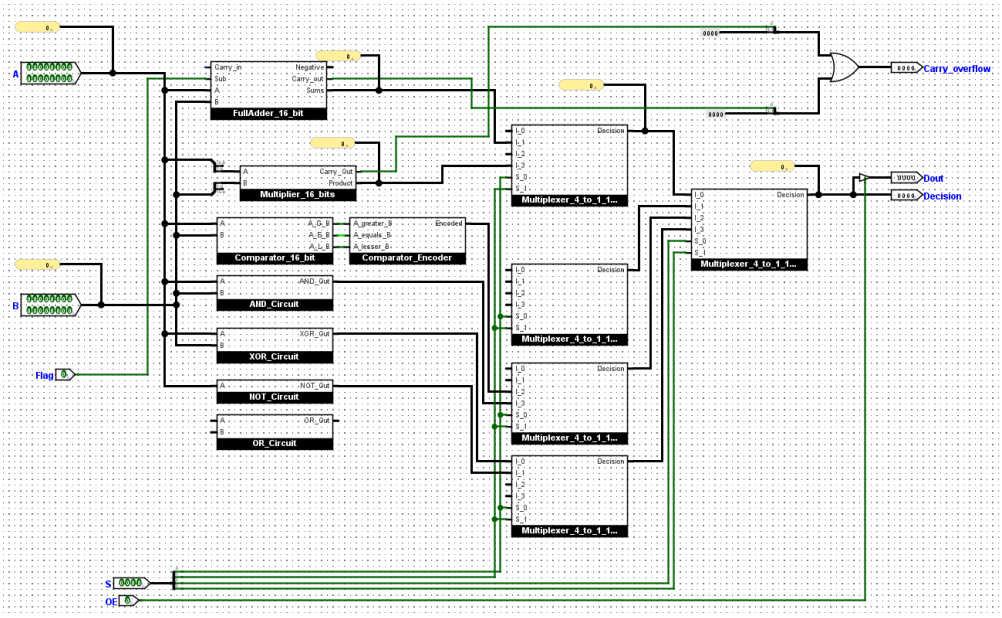

Figure 18: Circuit Diagram for a 16 bit ALU

12 Memory Cell

A memory cell is a fundamental unit of storage in computer architecture, capable of holding a single bit or a small amount of data temporarily. It serves as a building block for larger memory structures and allows the CPU to access and manipulate data efficiently. Each memory cell stores data temporarily and can either be in an active or inactive state, allowing the stored information to be written, read, or cleared based on the CPU's requirements.

To support CPU operations, specific components such as the **Memory Address Register (MAR)** and **Memory Buffer Register (MBR)** utilize memory cells to temporarily store and transfer data. The **MAR** holds the address of the memory location to be accessed, while the **MBR** temporarily stores the data being transferred to or from the specified location in memory. These registers work together to manage data flow within the CPU, ensuring that data is available at precise moments for processing.

In essence, memory cells are crucial in ensuring smooth communication between the CPU and memory, as they hold data briefly and release it when needed, facilitating efficient processing.

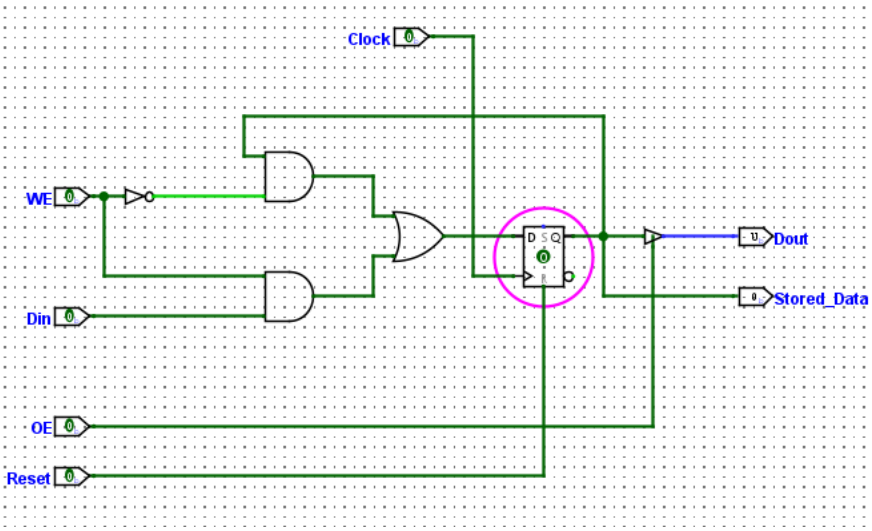

Figure 19: Circuit Diagram for a 1 bit memory cell

12.1 Memory Cell Design

The memory cell described here is a fundamental storage unit built around a **D flip-flop** (data or delay flip-flop), capable of storing a single bit of information. This memory cell includes five primary inputs:

- **Clock Signal (Clock)**
- **Data Input (Din)**
- **Write Enable (WE)**
- **Output Enable (OE)**
- **Reset**

Each input has a specific function that influences how the cell stores or releases data, controlled by the D flip-flop within the cell.

12.2 Input Details and Their Roles

1. **Clock Signal (Clock)**: The clock signal synchronizes the operation of the memory cell. The D flip-flop updates its output only on a specific edge (typically the rising edge) of the clock signal, ensuring data storage and release happen at precise times.
2. **Data Input (Din)**: This is the actual data (a single bit) that the memory cell may store. When Write Enable is active, the Din value is captured by the D flip-flop on the clock edge, and it becomes the new stored value in the cell.
3. **Write Enable (WE)**: This input controls whether the data at DataIn is written to the memory cell. When **Write Enable** is active (often a high signal), the D flip-flop will capture the DataIn value on the next clock edge. If Write Enable is low, the memory cell retains its current value, ignoring DataIn.
4. **Output Enable (OE)**: Output Enable determines whether the stored value in the memory cell is accessible for reading. When **OE** is active, the stored bit is available as an output. If OE is inactive, the output is disconnected, preventing the cell from outputting its data.
5. **Reset**: Reset clears the stored value in the memory cell, typically setting it to 0. When activated, it overrides other inputs and forces the output of the D flip-flop to 0, ensuring the memory cell starts in a known state.

12.3 How the D Flip-Flop Works in the Memory Cell

The **D flip-flop** is central to this memory cell's functionality. It has a data input (D), a clock input (Clock), and an output (Q). On the clock edge, if Write Enable is active, the flip-flop captures the Din value and stores it at Q. The Q output remains constant until the next clock edge where a new value may be captured.

Steps for Operation:

- **Data Write**: When WE is active, and a clock edge occurs, the D flip-flop takes in the value of Din and stores it.
- **Data Hold**: If WE is inactive, the D flip-flop ignores Din and keeps its previous value.
- **Data Output**: When OE is active, the stored data at Q is available as the cell's output.
- **Reset**: When Reset is triggered, Q is set to 0, clearing the memory cell immediately.

13 Registers

A register is a memory circuit capable of storing multiple bits and enabling these bits to be read out or taken in as input based on certain control signals. In this section, we will implement two types of registers: a 5-bit register and a 16-bit register. The 5-bit register will be used as the **Memory Address Register (MAR)**, since our 16-bit CPU can have a 5-bit address space. The 16-bit register will serve as a general-purpose register, which in specific cases will be referred to as the **Memory Buffer Register (MBR)**.

Each register will have the following four inputs:

1. D_{in}: Data input, which carries the bits to be stored in the register.
2. Clock input: Synchronizes the storing and releasing of data in the register.
3. WE (Write Enable): Controls when the data can be written into the register.
4. OE (Output Enable): Controls when the stored data can be output from the register.

These inputs are the same as those used in memory cells, as registers make use of memory cells to store each bit individually. Each bit input is divided accordingly and sent to its respective memory cell, allowing the register to store the desired number of bits (either 5 or 16 bits).

13.1 MAR

The Memory Address Register (MAR) is a 5-bit register specifically used to store memory addresses. In a 16-bit CPU architecture with a 5-bit address space, the MAR holds the address of the memory location that is either to be read from or written to.

The MAR's 5-bit width is appropriate for addressing within a limited memory range, typically used to fetch data or instructions by pointing to specific addresses. This register is crucial in CPU operations as it facilitates communication with memory by storing the address of data for the CPU to access.

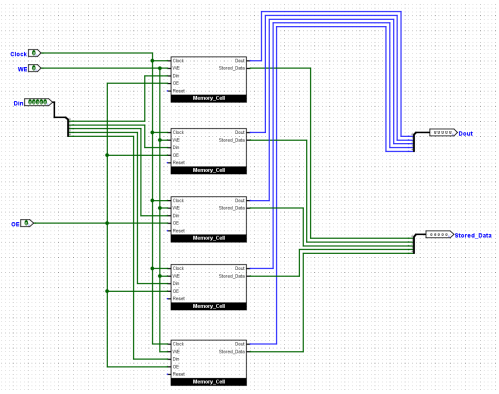

Figure 20: Circuit Diagram for a 5-bit MAR

13.2 MBR

The Memory Buffer Register (MBR) is a 16-bit register, which serves as a temporary storage for data that is either read from or written to memory. When the CPU retrieves data from memory, it is stored in the MBR before being processed. Similarly, when data is to be stored in memory, it is first placed in the MBR before the memory access operation.

With 16 bits, the MBR can store a full word of data in a 16-bit architecture, facilitating smooth data transfer and temporary storage for operations that involve memory I/O.

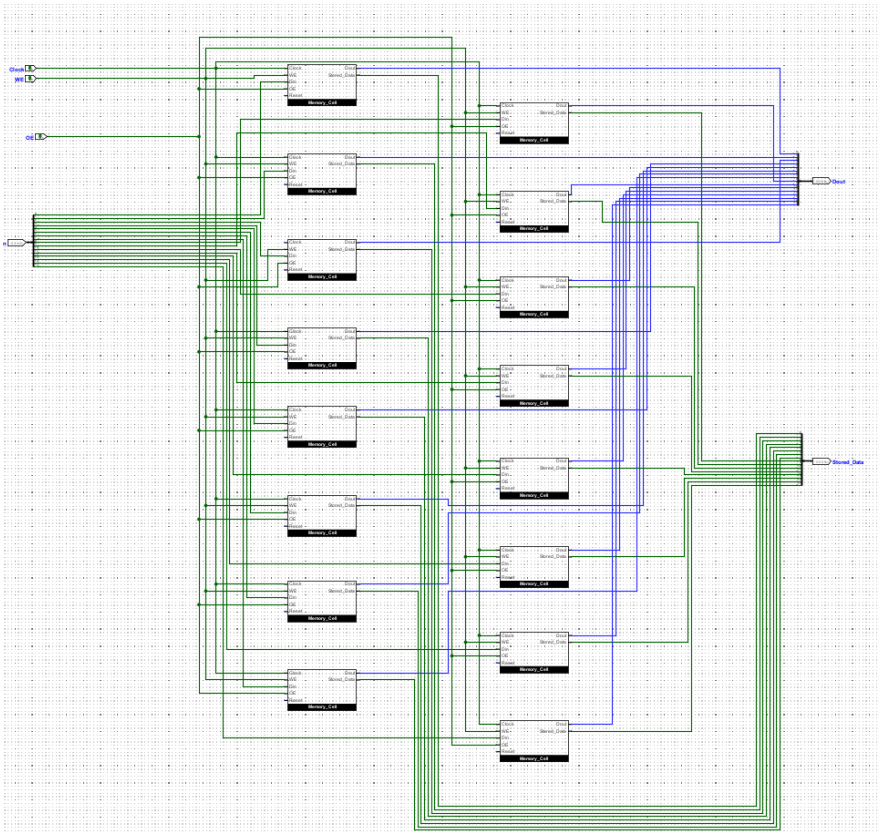

Figure 21: Circuit Diagram for a 16-bit MBR

14 Instruction Splitter

Instruction splitter by it's meaning is a splitter for the 16 bit instruction which we will split in four part; address of A, address of B, Flag bit and the OPcode.

The order will be:

1. (0 - 4) bits are the address of B
2. (5 - 9) bits are the address of A
3. (10 - 14) bits are the OPcode
4. 15 bit is the Flag bit

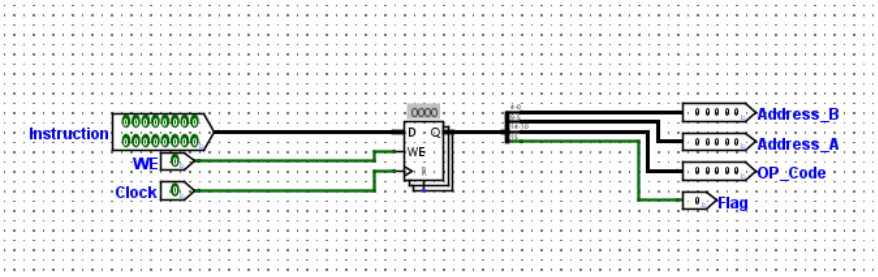

Figure 22: Circuit Diagram for a 16-bit Instruction Splitter

We have put a WE signal and a register so we can save the instruction for temporary purpose.

15 Matrix Multiplication

Matrix multiplication is a fundamental operation in linear algebra. It involves the multiplication of two matrices, where the number of columns in the first matrix must equal the number of rows in the second matrix. The result is a new matrix, where each element is computed as the dot product of the corresponding row from the first matrix and the column from the second matrix.

15.1 Definition

Given two matrices A and B:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \quad B = \begin{bmatrix} e & f \\ g & h \end{bmatrix},$$

their product $C = A \cdot B$ is given by:

$$C = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}.$$

15.2 Example

Suppose:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}.$$

The product is:

$$C = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 7 & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & 3 \cdot 6 + 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}.$$

15.3 Generalization

For two matrices A (of size $m \times n$) and B (of size $n \times p$), the resulting matrix C will have size $m \times p$, with each element c_{ij} calculated as:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

15.4 Matrix Multiplication in Logisim

To implement matrix multiplication in Logisim, we will focus on the multiplication of 2×2 matrices. Each element in the matrices will be represented using 8 bits. However, since we receive each number as a 16-bit input, only the 8 most significant bits (MSBs) will be used for the computation.

15.4.1 Input Representation

Consider the input matrices A and B:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \quad B = \begin{bmatrix} e & f \\ g & h \end{bmatrix}.$$

Each element (a, b, c, d, e, f, g, h) is received as a 16-bit number, where only the MSBs (most significant 8 bits) are used in the calculations. This allows for consistent and efficient multiplication within the 8-bit constraint.

15.4.2 Circuit Design Overview

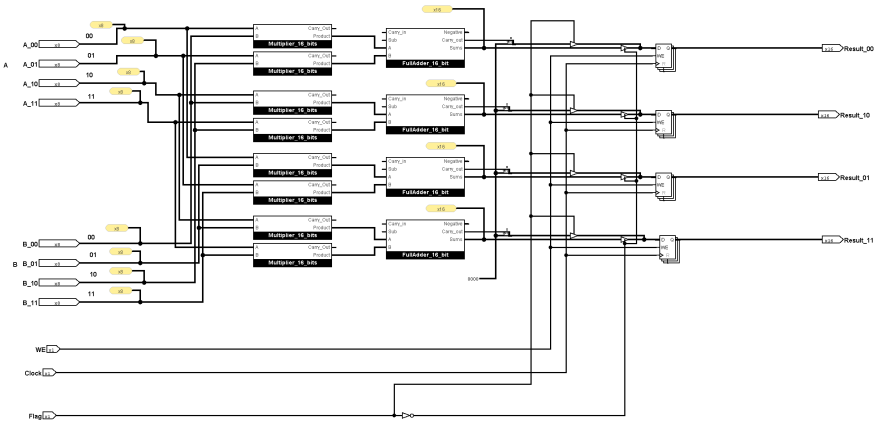

Figure 23: Circuit Diagram for Matrix Multiplication

In the circuit: - Eight partial products are generated by multiplying pairs of elements from A and B. - Each partial product involves multiplying two 8-bit numbers, resulting in a 16-bit product. - These partial products are then summed using 16-bit adders to compute the final results.

15.4.3 Addition and Result Storage

The eight partial products are added to compute the four elements of the resulting matrix:

$$C = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}.$$

- **Adders**: Since each product is 16 bits, 16-bit adders are used to perform the summations.
- **Result Storage**: The four computed values are stored in registers, representing the final resulting matrix.

15.4.4 Flag Bit Mechanism

To handle special cases, a flag bit is integrated into the circuit: - **Flag Bit = 0**: When the flag bit is 0, the resultant values of the matrix multiplication are stored in the registers. - **Flag Bit = 1**: When the flag bit is 1, any overflow that occurs during the calculations is stored in the registers instead. This provides additional diagnostic information and ensures that the circuit can handle potential overflow gracefully.

15.4.5 Key Design Notes

- **Efficiency**: Using 8 MSBs ensures that we work within an 8-bit framework, which simplifies multiplication and addition while reducing resource usage. - **Overflow Management**: The inclusion of a flag bit allows for a dual-purpose design, where both results and overflow information can be stored as needed. - **Scalability**: While this design is specific to 2×2 matrices, it can be extended to larger matrices with minimal changes to the circuit structure.

15.4.6 Final Output

After processing, the circuit outputs four values that represent the elements of the resulting matrix. These values can be further utilized in subsequent computations or visualized as part of the simulation.
This design demonstrates how Logisim can be used to implement and visualize matrix multiplication efficiently, taking advantage of modular arithmetic and circuit design principles.

16 Bus Terminal

The terminal will be used to facilitate the transfer of data from one data bus to another. Currently, RAM A receives data from MBR_A, and the same applies to other RAMs. However, our objective is to enable data flow between different data buses so that one bus can transfer data to another RAM.

For instance, if we want data from MBR_A to flow to RAM B, we must redirect the data to MBR_B so that it can be stored in RAM B. The terminal receives three data inputs, one from each of the MBRs, along with control signals:

1. MBR_Z
2. MBR_A
3. MBR_B
4. Destination (2 separate bits indicating the target RAM)
5. Write Enable (WE) signals for the registers
6. 3 buffer signals to control the data flow of the registers

16.1 Destination Bit Mapping

The destination is determined by a 2-bit combination, which maps as follows:

Destination Bit 1	Destination Bit 2	Mapped Location
0	0	RAM B
0	1	RAM A
1	0	System/Instruction RAM
1	1	RAM B

Table 7: Destination Bit Mapping for the Bus Terminal

This mapping ensures that the terminal can correctly route data to the desired RAM based on the control signals provided. By using this configuration, the bus terminal effectively manages data flow across the system.

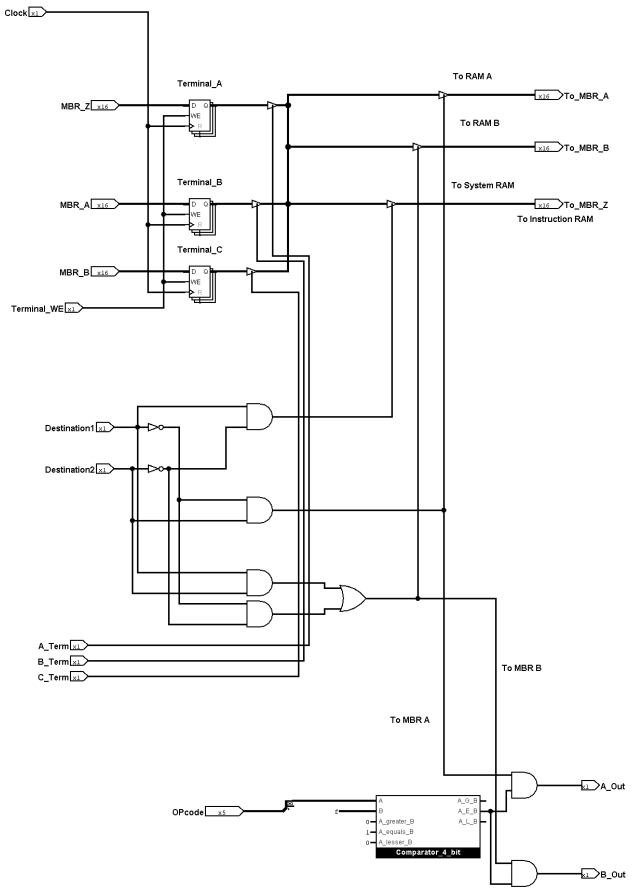

Figure 24: Circuit Diagram for a Terminal

Let's understand the operation of the terminal through an example. Suppose we have data incoming from MBR_A and need to send it to RAM B. To achieve this, the data must first be redirected to MBR_B, and its address must be sent to MAR_B. This way, the data flows correctly to RAM B.

16.2 Data Flow Example

1. **Writing into the registers**:
   - First we enable the WE signal of the registers so any incoming data from the MBR's can be written in the registers.
2. **Input Pin Activation**:
   - Since the data is coming from MBR_A, it will be received at input pin **B**.
   - To allow this data to pass to the registers, the controlled buffer corresponding to input **B** is enabled. This ensures that only the data from MBR_A flows through.
3. **Destination Signal**:
   - To send the data to MBR_B, we set the destination signal to $11\,(\text{binary}) = 3\,(\text{denary})$.
   - This configuration activates only the third AND gate in the circuit.
4. **Output Selection**:

- With the buffer enabled of the output the data will flow through that wire only and towards that MBR too.
- **Notice**: The vertical line in the middle connecting all inputs is crucial. It allows the data from any input to be sent to any register, depending on the destination signal.

5. **Data Forwarding**:
    - This output wire connects back to MBR_B, which then forwards the data to RAM B.

Outcome

- If the data were to flow to the second output pin, it would connect back to MBR_A, which would then send it to RAM A.
- Similarly, if directed to the first or other outputs, the flow would connect appropriately to the respective MBR and RAM.

Summary

- The terminal enables seamless switching of data between inputs and outputs.
- Data originates from one of the three input pins.
- It is directed to one of the registers based on the destination signal.
- Finally, the data flows back to the corresponding MBR and RAM, completing the transfer process.

This design ensures flexibility and efficiency in data handling, as any input can be redirected to any output based on the destination configuration.

17 Address Switcher

The address switcher operates similarly to the terminal but is specifically used for address switching. It takes one input (data), one signal, and one output.

1. Address of A
2. Destination Bit 1
3. Destination Bit 2

17.1 Destination Bit Mapping

The destination is determined by a 2-bit combination, which maps as follows:

Destination Bit 1	Destination Bit 2	Mapped Location
0	0	MAR Z
0	1	MAR Z
1	0	Nowhere
1	1	Nowhere

Table 8: Destination Bit Mapping for the Address Switcher

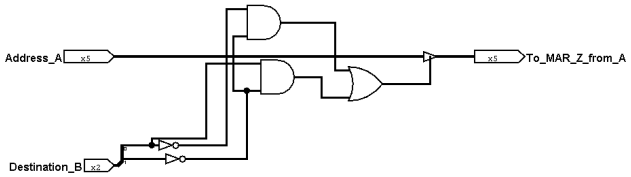

Figure 25: Circuit Diagram for a Address Switcher

17.2 Why Only Two Mapped Combinations?

The address switcher is designed to handle cases where the address from A (normally sent to MAR_A) needs to be redirected to MAR_Z due to specific conditions. For example:

- Under regular operation, $Address_A$ is sent directly to MAR_A.
- However, when the terminal is simultaneously in use, certain conditions may require $Address_A$ to be rerouted to MAR_Z. This ensures that the terminal has access to the required address for further processing.

The mapping is limited to two valid combinations (00 and 01) because these represent the scenarios where MAR_Z is required:

- 00 and 01 both direct the address to MAR_Z, ensuring compatibility with terminal operations.
- 10 and 11 are invalid or unused because there is no logical routing to other destinations in this configuration.

17.3 Controlled Buffer for Data Flow

Data from $Address_A$ flows through an output line, which is controlled by a buffer outside this component. This buffer ensures:

- Proper routing of the address data based on the destination bits.
- Prevention of undesired interference between address lines.

By combining the address switcher with the terminal, the system maintains flexibility and ensures data and addresses are routed efficiently according to the required logic.

18 Assembling the main circuit

We have now designed all the components required for the 16-bit CPU. The main circuit will consist of the following components:

1. Arithmetic Logic Unit (ALU)
2. Registers (MAR and MBR)
3. Other Registers (for temporary storage)
4. Instruction Splitter
5. Matrix Multiplication Unit
6. Bus Terminal
7. Address Switcher
8. RAM's (System, Instruction, RAM A, RAM B)
9. Control Unit (ROM based)
10. Clock
11. Register Swapper
12. MAR's Incrementer

The components will be interconnected to form a complete 16-bit CPU. The control unit will manage the flow of data and instructions, while the ALU will perform arithmetic and logical operations. The RAMs will store data and instructions, and the registers will facilitate data transfer and temporary storage. The matrix multiplication unit will handle matrix operations, and the bus terminal and address switcher will manage data and address routing. The clock will synchronize the operations, ensuring that the CPU functions correctly.

We will start with the RAM's, assembling them first, then connecting the ALU, registers, and other components to complete the 16-bit CPU. We will have four RAM's, each with a specific purpose:

1. **System RAM**: For general data storage and processing.
2. **Instruction RAM**: To store instructions for the CPU.
3. **RAM A**: For specific data storage related to operations (address A).
4. **RAM B**: For additional data storage related to operations (address B).

The System and Instruction RAM's will be connected to MAR Z and MBR Z, respectively, while RAM A and RAM B will be connected to MAR A and MAR B, respectively. This configuration ensures that data and instructions are stored and accessed correctly within the CPU while only the authorized components can access the required memory locations and no else.

18.1 RAM's Connection

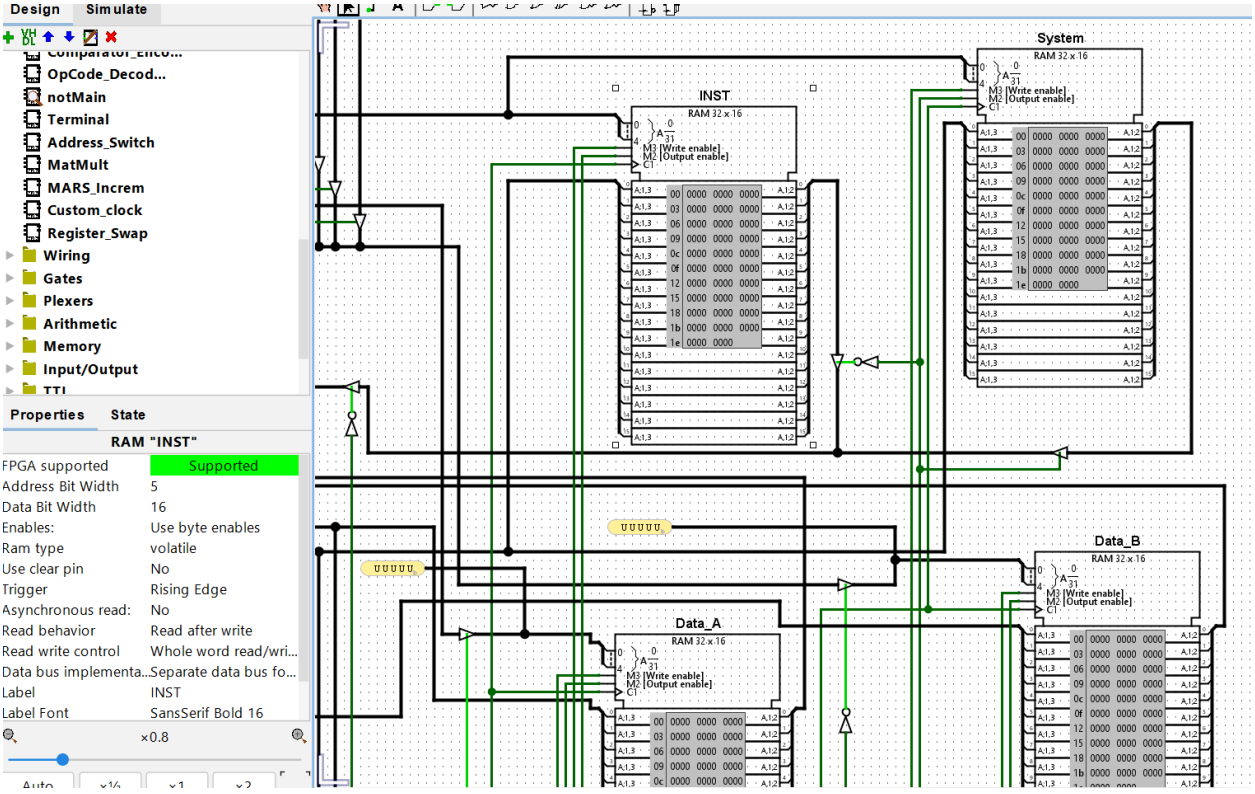

Figure 26: Circuit Diagram for RAM's Connection

As you can see that the all the RAM's are of address width 5 bits and data width 16 bits. The data is coming from the MBR's and the address is coming from the MAR's. The data is written in the RAM's when the WE signal is high and

the data is read when the OE signal is high. The data is read from the RAM's and stored in the MBR's. Here in each RAM on the left their are five wires coming in, let's talk about them in order from top to bottom:

1. Address In
2. WE
3. OE
4. Clock
5. Data In

The address in is the address which is coming from the MAR's, the WE signal is the write enable signal, the OE signal is the output enable signal, the clock signal is the clock signal and the data in is the data which is coming from the MBR's. Notice here that the address coming in to System and Instruction RAM is from the same line, this is because the address is coming from the MAR Z, both the RAM's are connected to the same address line.

The wire on the right side of the RAM's are the data out which is going to the MBR's. As you can see that I have connected the data out of the System and Instruction RAM to the same output line, this is because the data is coming from the same line (from MBR Z) and is going to the same line (to MBR Z). The data out of the RAM A and RAM B is going to the MBR A and MBR B respectively. That we will see in the next section.

Notice:

You can also see that I have connected two buffers in the output wire of the System and Instruction RAM, that is because if one RAM is currently outputting the data the other should not be able to output the data. This is done to prevent the data from being corrupted. This is simply done with a combination of buffers and NOT gates. I have connected the buffers signal with the OE of System RAM so basically when data is being outputted from the System RAM the buffer signal will be high and the NOT gate will make it low so the Instruction RAM will not be able to output the data. The same is done for the Instruction RAM. This is because by default the RAM will never output the data unless the OE signal is high, but in some cases data keeps floating in the output wire hence this is done to prevent that.

18.2 MAR's and MBR's Connection

There will be 3 MAR's and 3 MBR's. The MAR's will be connected to the RAM's address input pin whereas the MBR will be conncted to the RAM's data input pin. The MAR's will be connected to the RAM's address input pin so the address can be written in the RAM's. The MBR's will be connected to the RAM's data input pin so the data can be written in the RAM's. The data outputted from the RAM will go back again in the MBR's but this time from their data input pin.

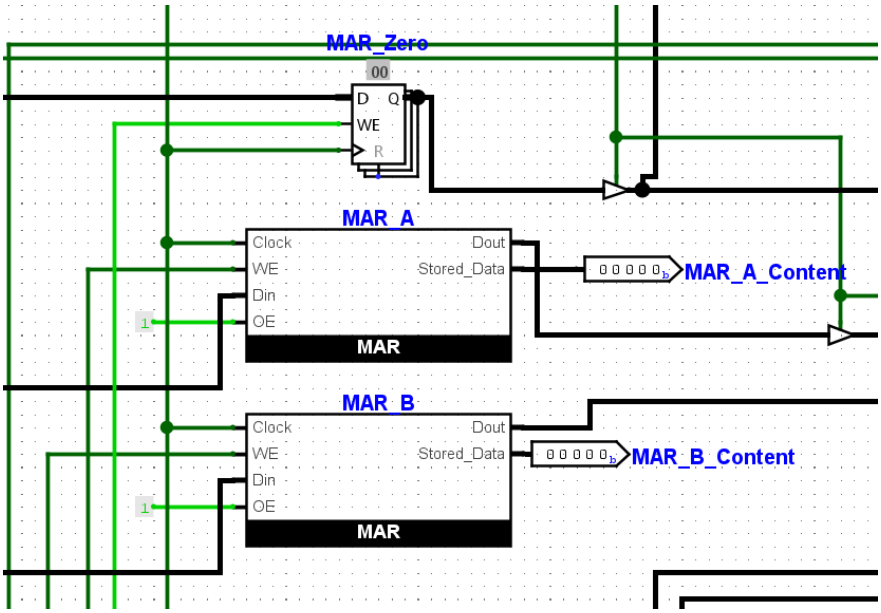

Figure 27: Circuit Diagram for MAR's

Here I am using some self made MAR's and some built-in so if you want to use any of it, it is your choice. For the built-in MAR's you can use the built-in register and for the self made you can use the one I have made. The built-in register will be labeled as MAR Z as I have used MAR A and B for the self made ones. Since the MAR Z will also carry the address, it will have data width of 5 bits. All of the MAR's will receive their data(addresses) from the instruction splitter. **Notice:** The address in MAR Z will also be coming from the counter which will be incremented by the clock signal so it can fetch new instructions from the instruction RAM.This is a special case.

The other two MAR's will receive their address from the instruction splitter. The data outputted from the RAM's will be stored in the MBR's. These MAR's output wires are those wires which you saw in the RAM's connection section. RAM's receive their address from these registers only.

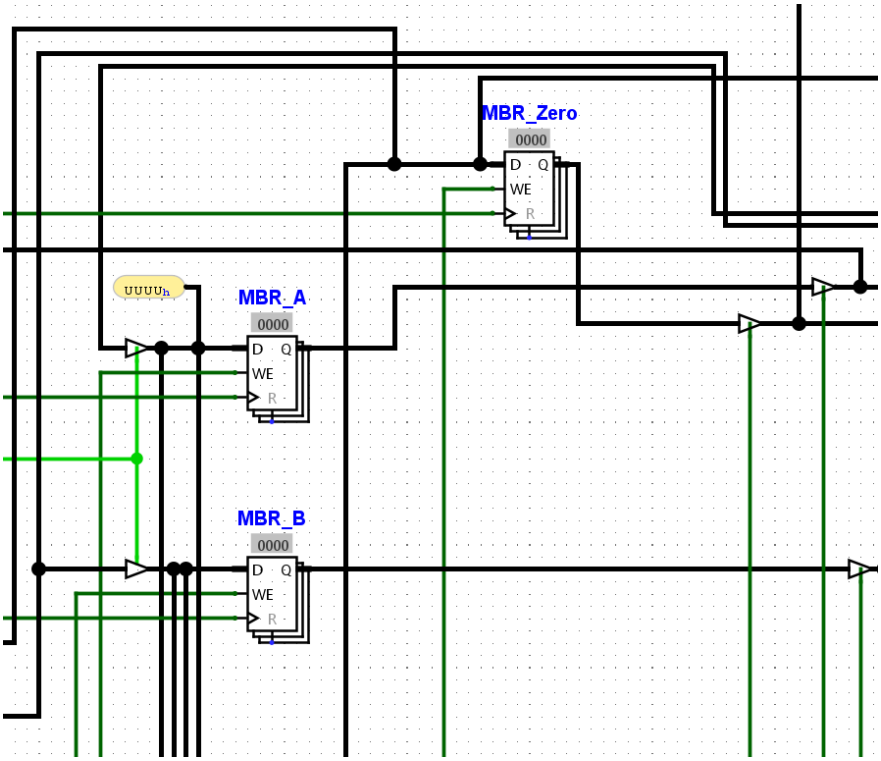

Figure 28: Circuit Diagram for MBR's

The data sent for the address of MAR's will be sent from the MBR's. I have used built-in registers for these and set the data width to 16 bits since the data in our 16 bit CPU is of 16 bits. The data in each MBR is coming from two sources currently where one source will be the RAM and the second will be the terminal. The data from the RAM's will be stored in the MBR's and the data from the terminal will be stored in the MBR's. The data from the terminal will be sent to the RAM's. We will talk about the terminal later. The data from the MBR can be sent back to the RAMs or to the terminal or even to the ALU. The data from the MBR's will be sent to the ALU for the operations.

Notice:

There are buffers before the MBR's, it is because since the data is coming from more than one source it cane cause it collide with each other hence the buffers are used to prevent that. If from one source data is coming and it is allowed than the other source will be blocked from sending the data. This is done to prevent the data from being corrupted. Even after the MBR's you can see the buffers that is because at certain times we need to send the data to be sent to the RAM's not always so this we can control that flow too.

18.3 Program Counter

The program counter is a counter component openly found in logisim which is basically like a register but can be incremented or decremented. The program counter will be used to fetch the instructions from the instruction RAM. The program counter will be connected to the MAR Z directly so it can fetch the instructions from the instruction RAM. The program counter will be incremented by the clock signal. The program counter will be of 5 bits since the instruction RAM is of 5 bits. The program counter will reset automatically when it reaches the maximum value which will be it's own address $1f$ (31 in decimal).

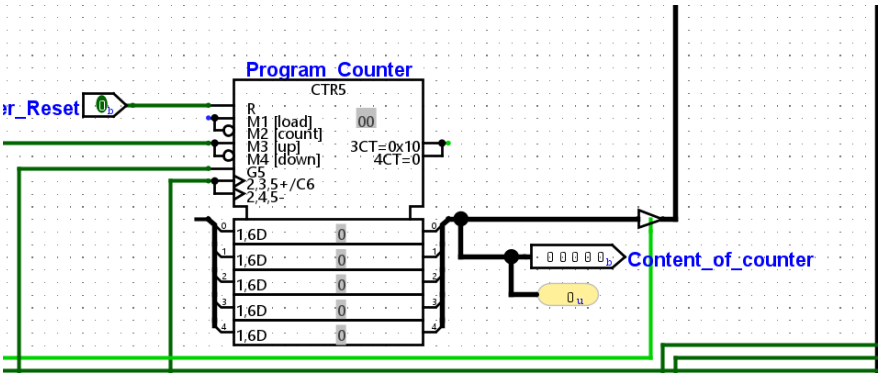

Figure 29: Circuit Diagram for Program Counter

Let's talk about the inputs and output of the program counter in order from top to bottom:

1. Reset (R): This is the reset signal which will reset the program counter when it reaches the maximum value. **We don't need it.**
2. Load (L): This is the load signal which will load the data in the program counter. **We don't need it.**
3. Up and Down (U/D): This is the up and down signal which will increment or decrement the program counter. when 1 it increments and when 0 it decrements. So whenever incrementing the clock signal will be 1 at that moment this signal has to be 1, otherwise it will decrement. **We don't need it.**
4. Enabler of the counter: This is the enabler of the counter which will enable the counter to work. **We don't need it.**
5. Clock (CLK): This is the clock signal which will increment the program counter. **We need it.**
6. Output: This is the output of the program counter which will be connected to the MAR Z. **We need it.**

I placed a buffer in front of the output of the program counter, this is because the program counter will only output the data when the clock signal is high. This is done to prevent the data from being corrupted since the data to MAR Z can also come from the instruction splitter (that we will see in later cases).

18.4 Instruction Splitter Connection

The instruction splitter will be connected to the MAR's since it outputs four things: address of A, address of B, Flag bit and the OPcode. The address of A and B will be sent to the MAR A and B respectively (can be sent to MAR Z in some cases). The Flag bit will be sent to the ALU, MatMult, Terminal and more. The OPcode will be sent to the control unit via a register. As you can see it has the WE signal since it write in a register first then outputs the data. The data is coming from the instruction RAM via MBR Z.

On the left you can see a register with labelling as **'current OPcode'** or **'initial OPcode'**, this is because the OP-code will be sent to the control unit via a register. The OPcode will be stored in the register and then sent to the control unit. This is done to prevent the OPcode directly to the ROM (Control Unit) which will jump otherwise which we don't want to happen indefinitely. The current OPcode will be the one coming from instruction splitter and will only be received when the instruction is received in the instruction splitter. The initial OPcode will be sent to it to reset the ROM (Control Unit) when the instruction is completed, this will be a constant with value 00. We will see this in the later sections.

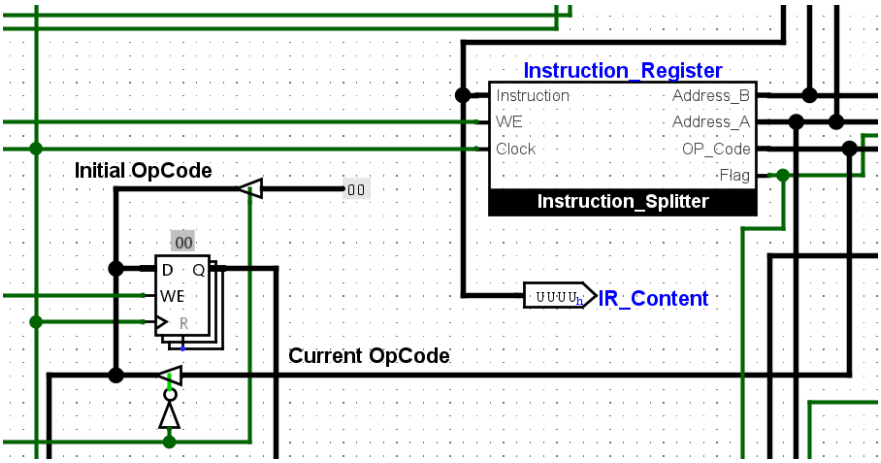

Figure 30: Circuit Diagram for Instruction Splitter

The OPcode here is labeled as **Signal** and it will be not coming from the register mentioned in the previous section. The OPcode will be coming from the instruction splitter directly. I have used a splitter to split the five bits of the OPcode into 4 and 1 bit where only the 4 bits will be utilized in the ALU as talked earlier in the ALU section.

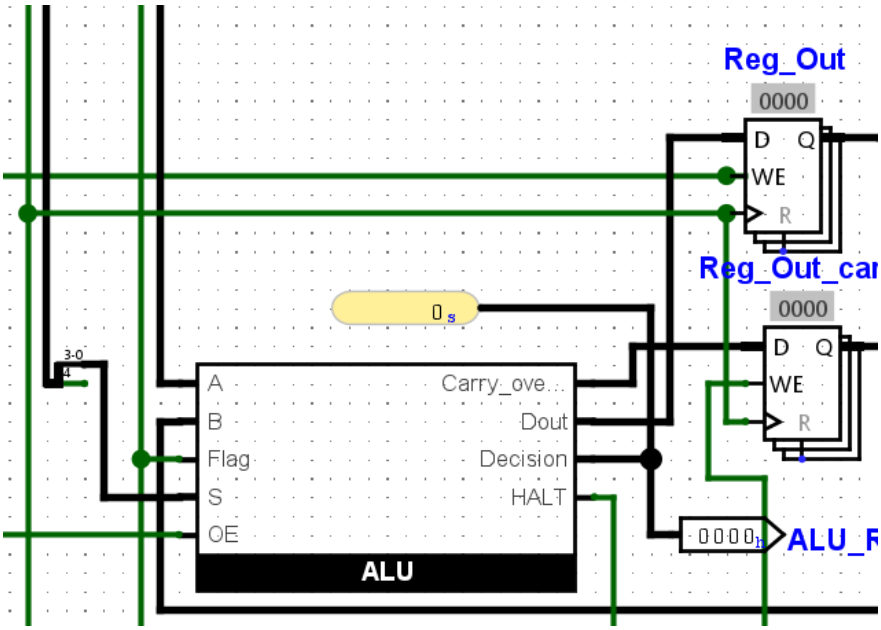

Figure 31: Circuit Diagram for ALU

Here you can see that their are two registers in front of the ALU, this is because the output of the ALU that we talked about are 2; one is the result and the other is the overflow which primarily comes from the adder or the multiplier. The result will be stored in the register and the overflow will be stored in the other register. Both the data will be sent throu one stream of output based on the flag bit of the **Store Instruction**. The data will be sent to the MBR Z and then to the System RAM. From MBR Z it can be sent to other RAM's if wanted to by the terminal.

Notice:

Their is 1 more output of the ALU which is the **HALT** which is one of the operation this 16 bit CPU can do. It is basically just stopping the whole circuit. Will talk about that later.

18.5 Address Switcher Connection

The address switcher will be connected betweent the instruction splitter and the MAR's. The address switcher will receive the address of A from the instruction splitter and will send it to the MAR Z based on the destination input explained earlier. The destination signal basically is the address of B that is why it has been splitter into 2 bits. As you can see that I have added some other buffers. The two buffers on the right side of the address switcher are for the address going to MAR Z both of A and B since in some instructions both the addresses are needed in the MAR Z. The buffer on the left sides are to control the flow of the data to the MAR's, if the data is flowing from address of A to MAR B, if needed, then data going from address of B to MAR B should be blocked. This is done to prevent the data from being corrupted.

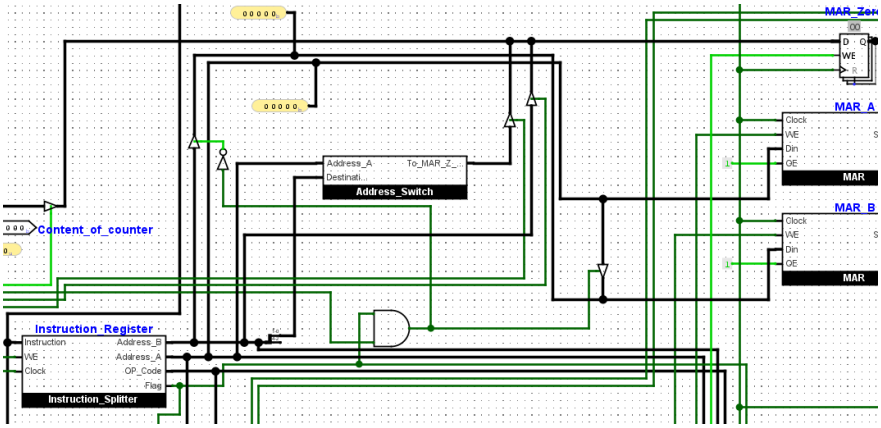

Figure 32: Circuit Diagram for Address Switcher

18.6 MAR's Incrementer Connection

First let's introduce the incrementer and then we will connect it to the MARs.

18.6.1 MARs Incrementer

The MARs Incrementer is a component made up of three temporary registers to hold the values of the three MAR's at the same time and then increment them 3 times as needed for two of the instructions; **Block Load** and **Block Store**. The incrementer will take it's inputs from MAR A, MAR B and MAR Z and then increment them 3 times. The incrementer's outputs will be connected to the MAR A, MAR B and MAR Z. The incrementer will be connected to the MAR's via a buffer so the data can be sent to the MAR's only when needed.

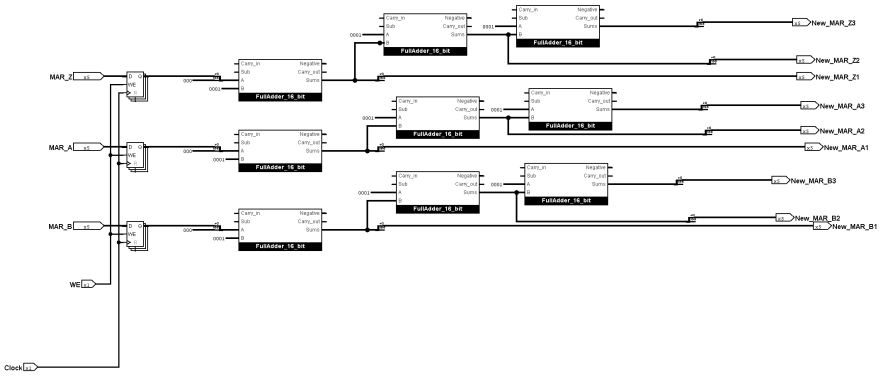

Figure 33: Circuit Diagram for MARs Incrementer

As you can see that I have used 9 adders, 3 for each MAR. Each address will get incremented three times and be sent back to the MAR's but when needed the buffers may get opened and the data will be sent to the MAR's. In each adder I have placed a constant with value **0001** which means 1 in decimal. This is to increment it by 1 each time.

18.6.2 Connection

Why are we incrementing the address 3 times? This is because in the **Block Load** and **Block Store** instructions we need to increment the address 3 times to get the next address. The data is stored or will be stored in 4 consecutive addresses, hence we need to increment the address 3 times to get the next address, where the first (initial) address is the one coming from the MARs directly and next 3 will be the incremented ones. The incrementer's outputs are 9; 3 for each MAR. The output of the incrementer will be sent to the MAR's via 9 buffers, grouped into 3 groups of 3 buffers each.

- The outputs **New MAR Z1, New MAR A1, New MAR B1** will be sent to the MAR Z, MAR A and MAR B respectively with the first buffer group.
- The outputs **New MAR Z2, New MAR A2, New MAR B2** will be sent to the MAR Z, MAR A and MAR B respectively with the second buffer group.
- The outputs **New MAR Z3, New MAR A3, New MAR B3** will be sent to the MAR Z, MAR A and MAR B respectively with the third buffer group.

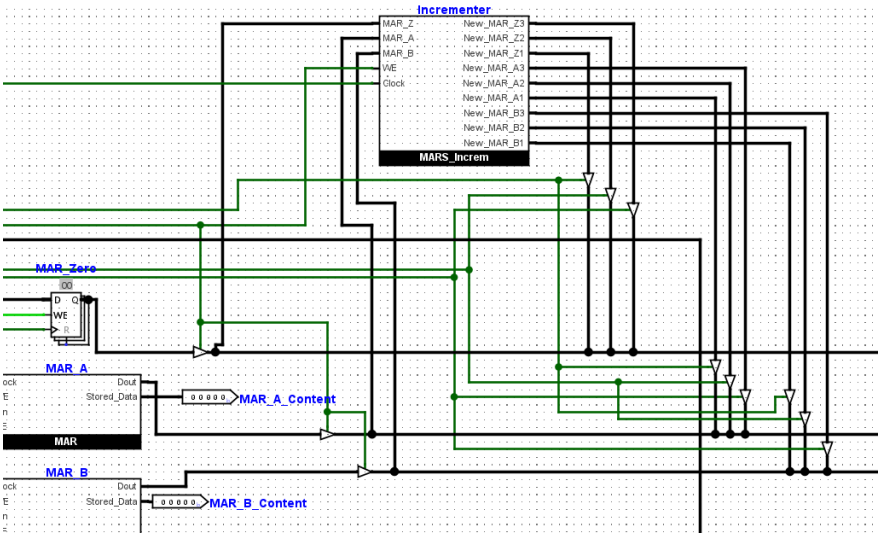

Figure 34: Circuit Diagram for MARs Incrementer Connection

From here onwards these incremented addresses will directly be sent to the RAM's for the data to be stored or fetched. So basically when the first address is need the buffers directly in front of the MAR's will be opened and the address goes to the RAMs and to the Incrementer where the WE signal is enabled hence the data gets saved and incremented, not yet outputted. If the instruction demands the next addresses then the buffer groups will be opened so for example if the instruction demands the next address of MAR Z then the buffer group of first incrementation will be opened which will output all three MAR data whereas needed is only one. This will not cause any problem since the RAM A and B will not accept the values MAR A and B (incremented) since we don't need them at the momemnt. While the data is being outputted throught the incrementer, the buffer in front of the MAR's will be closed so the data doesn't get corrupted.

18.7 Register Bank A and B

Register Banks A and B both contain 4 registers each. The data from the MBR's will be stored in the registers of the Register Banks A and B. The data from the Register Banks A and B will be sent to the ALU for the operations.

Basically the banks will act as temporary station for the data from where it can be sent to the ALU, MatMult or for register swapping. More specifically the banks are for the block load instruction where the data is stored in the bank A for values of RAM A and bank B for values of RAM B, which will be most likely be used in MatMult or Register Swapping.

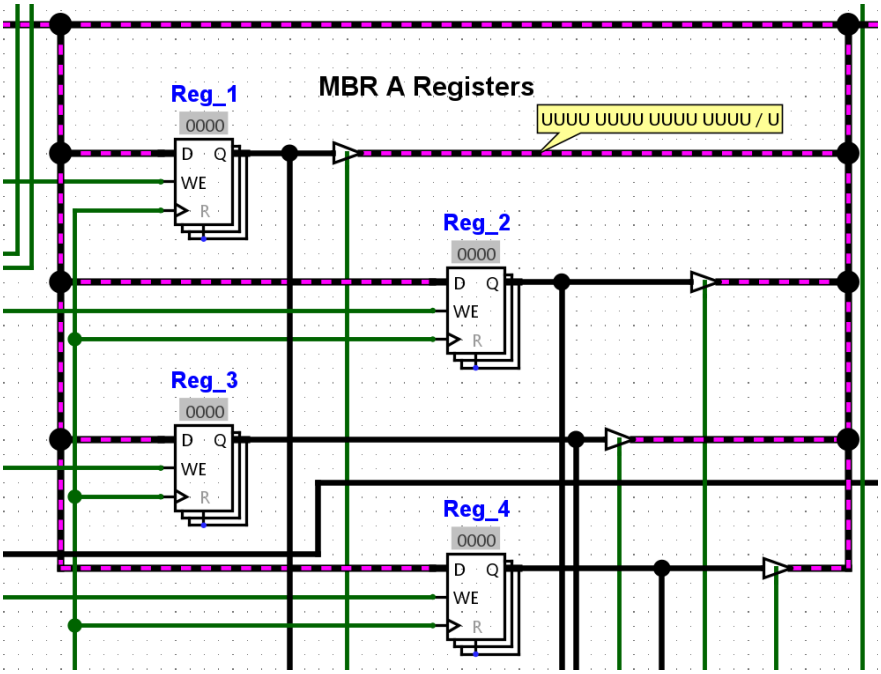

Figure 35: Circuit Diagram for Register Bank A

Here you can see that the data is coming from the right from the MBR A's output and can be stored in any one of the registers from 1 to 4. Each register will have their own WE signal since we don't want multiple registers to store same data and they will also have their own buffer in front of them to control their outputs since each register can output the data at the same time, but we want one at a time. All the registers after their buffers follow the same line and connect back to the same wire where they came from, meaning going back to their inputs or further towards the ALU.

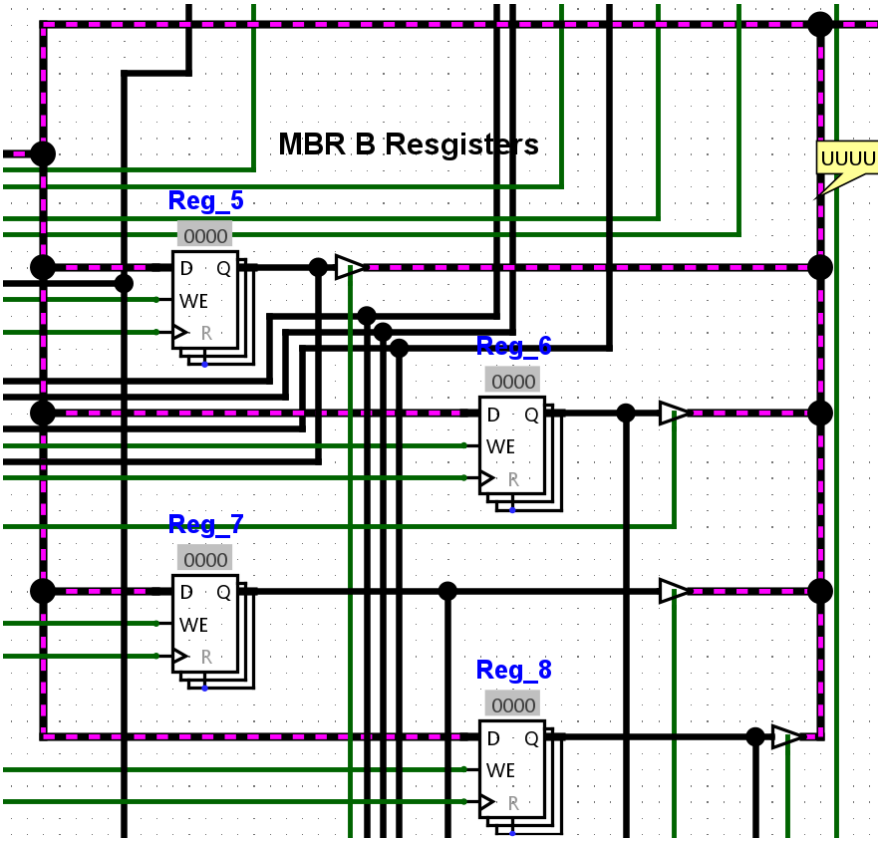

Figure 36: Circuit Diagram for Register Bank B

Similar to the bank A the bank B also has 4 registers and the data is coming from the MBR B's output. The data can be stored in any one of the registers from 5 to 8. Each register will have their own WE signal since we don't want multiple registers to store same data and they will also have their own buffer in front of them to control their outputs since each register can output the data at the same time, but we want one at a time. All the registers after their buffers follow the same line and connect back to the same wire where they came from, meaning going back to their inputs or further towards the ALU.

So basically both banks can work simultaneously and can store the data from both MBR's together in their respective banks.

18.8 Matrix Multiplication Unit Connection

The Matrix Multiplication Unit will be connected to the Register Banks A and B. The data from the Register Banks A and B will be sent to the Matrix Multiplication Unit for the matrix multiplication. Since we have already discussed the matrix mutliplication unit we know that we made some registers inside the MatMult so we don't need to save data outside the component or in Bank A or B. The four results one by one will be sent to the MBR Z to be stored in the system RAM through the same line where the output of the ALU is going through.

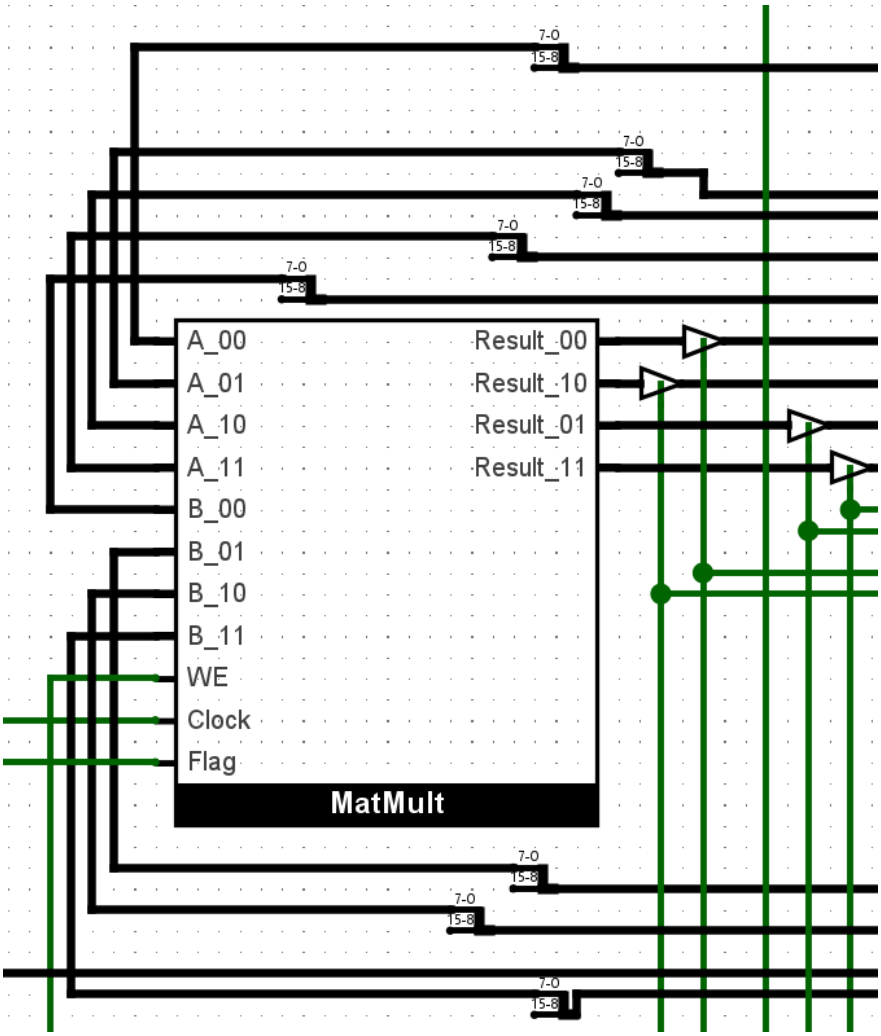

Figure 37: Circuit Diagram for Matrix Multiplication Unit Connection

I have placed this component right below the ALU, where it can have close and easy access to the registers of the Register Banks A and B and the output stream line. Here you can see that there are multiple inputs in the MatMult where I have written their names as the coordinates these values lies on:

- A_{00}: This is the value of the first register of the Register Bank A.
- A_{01}: This is the value of the second register of the Register Bank A.
- A_{10}: This is the value of the third register of the Register Bank A.
- A_{11}: This is the value of the fourth register of the Register Bank A.
- B_{00}: This is the value of the fifth register of the Register Bank B.
- B_{01}: This is the value of the sixth register of the Register Bank B.
- B_{10}: This is the value of the seventh register of the Register Bank B.
- B_{11}: This is the value of the eighth register of the Register Bank B.
- WE signal: This is the write enable signal which will enable the MatMult to store the data in the registers inside it to perfrom the computation.

- Clock: This is the clock signal which will increment the MatMult to perform the computation.
- Flag Bit: This is the flag bit which will be sent to the ALU, MatMult and the Terminal.

You can see that each input is passing throug a splitter first, that is because the data coming from the banks are of 16 bits but we have made the MatMult component to handle 8 bit numbers hence we need to remove the left 8 bits (MSB's). Now lets talk about the outputs of the MatMult:

- $Result_{00}$: This is the result of the first element of the matrix multiplication.
- $Result_{01}$: This is the result of the second element of the matrix multiplication.
- $Result_{10}$: This is the result of the third element of the matrix multiplication.
- $Result_{11}$: This is the result of the fourth element of the matrix multiplication.

Since the registers that are inside the MatMult will be holding the result will be constantly outputting the data we need to control them and let us get one value at a time when needed we can use buffers in front of it.

18.9 Register Swapper Connection

The Register Swapper will be connected to the Register Banks A and B. The data from the Register Banks A and B will be sent to the Register Swapper for the swapping of the data. The swapped data will be sent back to the register banks and into their registers via the MBR's.

18.9.1 Register Swapper

The Register Swapper will recieve its data from the 8 registers directly but will send data one by one after swapping to them.

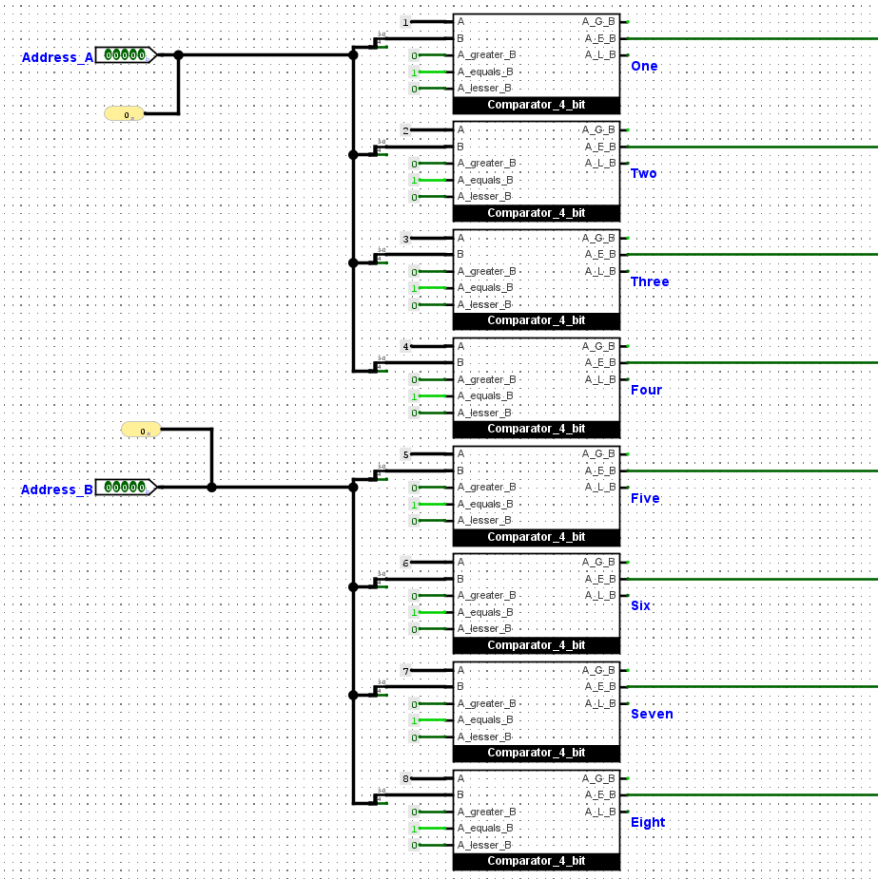

Figure 38: Circuit Diagram for Register Swapper Comparators

Here you can see I have received the Address of A and B which is from the instruction splitter directly. I then compare the two addresses with their respective conparators, since the address of A can be from 0001 to 0004 and the address of B can be from 0005 to 0008. You can clearly see that each comparator has one input from either address A or B and the second is a constant with the values from 1 to 4 for the address of A and 5 to 8 for the address of B. Basically I have also divided the comparators in 2 groups of 4.

Data Entry:

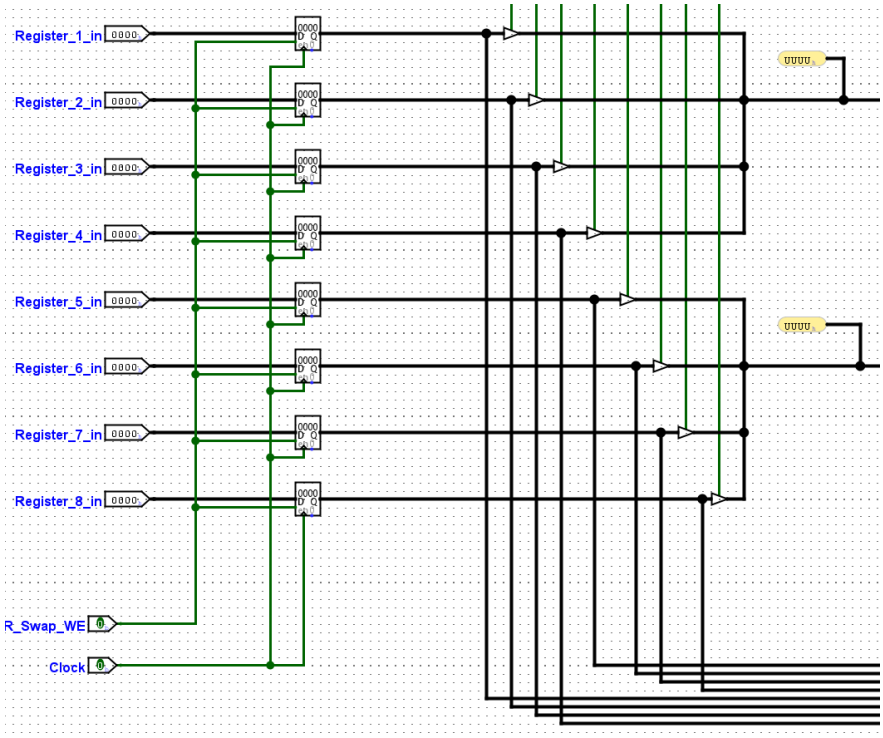

Figure 39: Circuit Diagram for Register Swapper Data Entry

This is the second part of the register swapper, here the data is actually entering from the 8 registers. The small boxes that you see are built-in registers, it's just their shape which is changed to old style. All registers will receive data from their respective banks registers at the same time hence for all of them the WE signal is same. After that you can see that I have placed a buffer in front of the output of the registers, this is because data can only flow from those registers whose swapping is needed. The buffers signals are coming from the previous comparators that we talked about above. So since Address A is from 0001 to 0004 the buffer signals will be high for the first 4 registers and since Address B is from 0005 to 0008 the buffer signals will be high for the last 4 registers. Meaning from both banks only one register will output the data at a time and all others will be blocked. They all combine to one wire after it since they need to be swapped now.

Data Swapping:

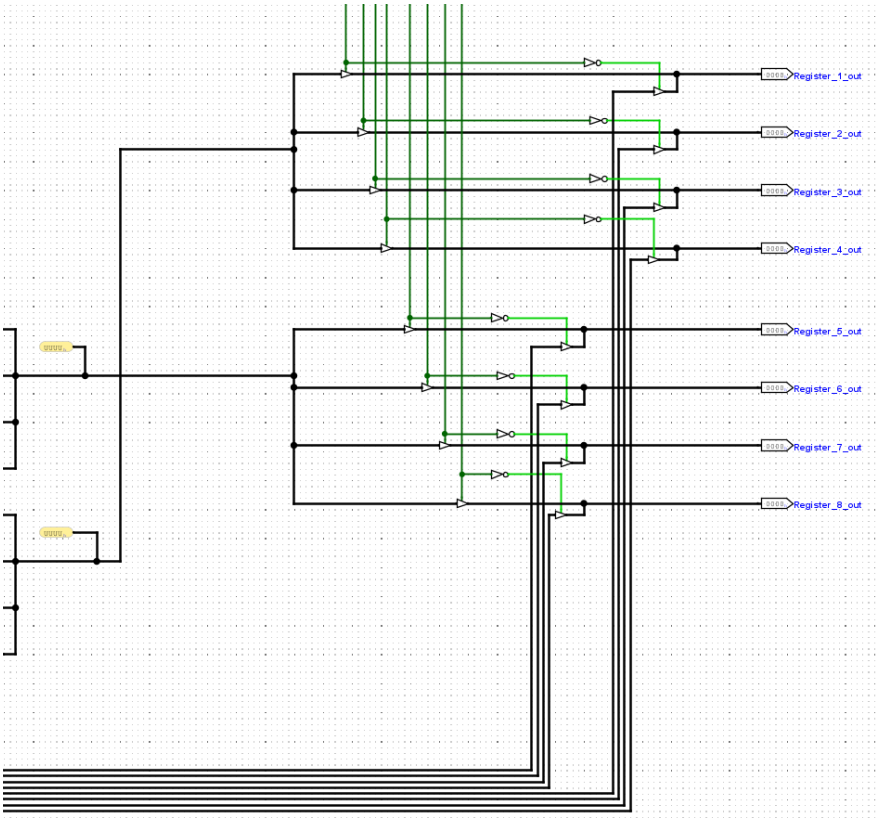

Figure 40: Circuit Diagram for Register Swapper Data Swapping

This is the last part of the register swapper, here uptill now we have two data inputs chosen based on the address of A and B. Now we need to swap the data. As you can see the above data is from first four registers (Bank A) and the below data is from the last four registers (Bank B). The data from the first four registers will be sent to the last four registers and the data from the last four registers will be sent to the first four registers. But how do we know which of the output will receive the incoming data? This is also done by the buffer signals from the comparators that we talked about earlier. The signals will be high for one of the first four and last four registers and only those which needs to be swapped. So now we need to send the incoming swapped data to only those outputs whose address we recieved and others shall receive their original data. This will be done with a combination of buffers, NOT gates and AND gates. if you observe anyone of the output, you can see that the swapped data is coming and is connected to a buffer directly from those comparators while the same output is also receiving the original data from the input registers directly too but they are also connected to the buffers whose signals are coming from the comparators but first passes through the NOT gate.

Data Output: Let's understand the concept of the whole component from an example:

Address A	Address B	Register Bank A	Register Bank B
0001	0005	Active	Active
0002	0006	Inactive	Inactive
0003	0007	Inactive	Inactive
0004	0008	Inactive	Inactive

Table 9: Example Data

Now in this example only the first register of the Register Bank A and the fifth register of the Register Bank B are active. The data will be first compared and then entered in the registers. The data will be swapped and then sent to their banks outputs. In the last steps it will check which of the outputs are active and in our case only the first and the

fifth are active. The other outputs will be blocked in receiving the swapped data and will only get their original data back. The data will be sent to the MBR's and back to the register banks.

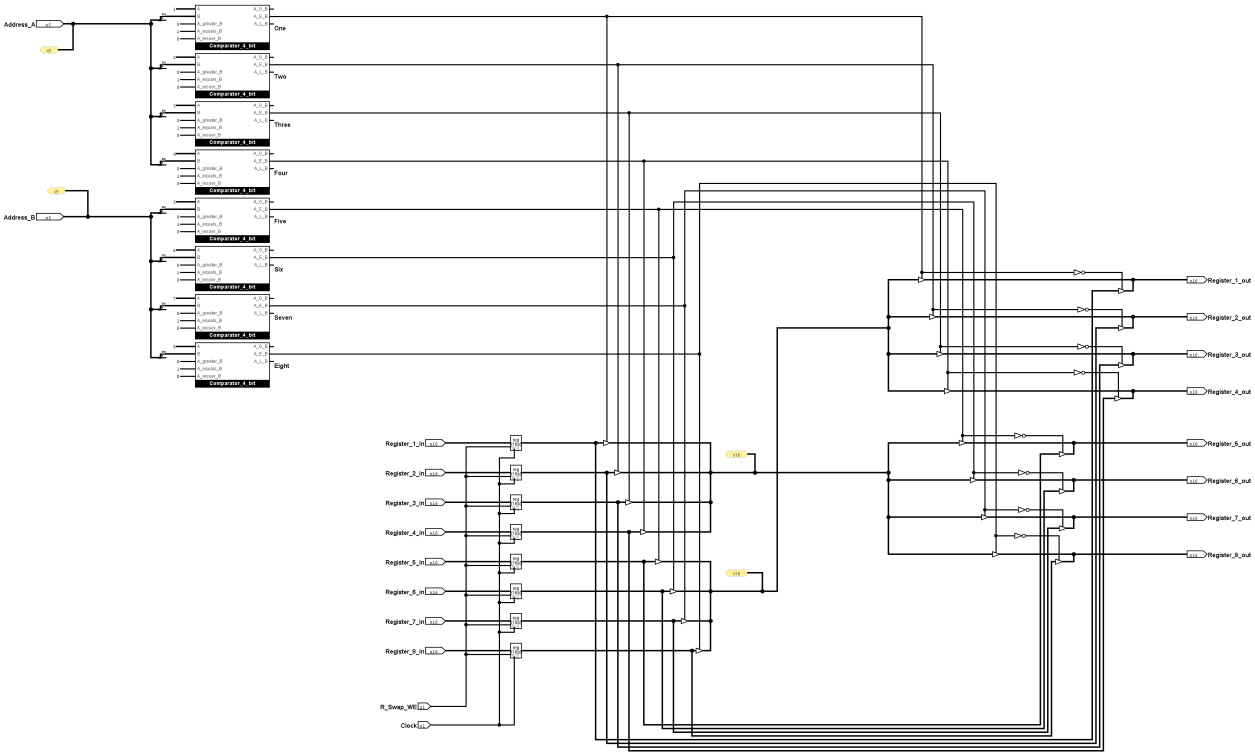

Figure 41: Circuit Diagram for full Register Swapper

18.9.2 Connection Inside The Main Circuit

The register swapper will get its 8 registers data directly from the banks. Remember the buffers in front of each register, they will control the flow of the data out of them, since we need all of them all at once we will get the data even before the buffers.

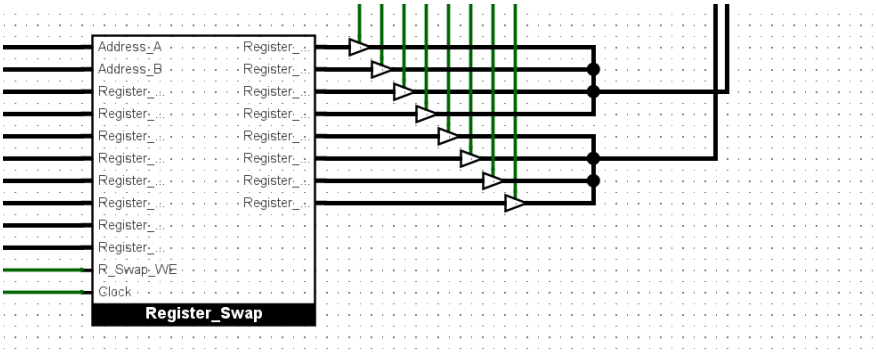

Figure 42: Circuit Diagram for Register Swapper Connection

Here it can be seen that their are 8 outputs as told earlier and they are connected to even more buffers, which will control their outward flows. Since both MBR A and B can have one data at a time we will send one one data at a time simultaneously to both MBR A and B.

Notice:

The data outputted are in groups of 2 where the first group is the data from the first four registers and the second group is the data from the last four registers. Hence, the first four outputs are merging into one wire after the buffers since they will going to the same source. The same is done for the last four outputs. To signal the output of the data or basically controlling the buffers I reused the MatMult output signals since these two operations will never clash together.

18.10 Terminal Connection

The terminal will be only connected to the MBR's where it receives data from their outputs and sends data back to them through their inputs. We have already discussed the terminal in the previous sections. Below is an attached image of the terminal connection. It is placed below the MBR's.

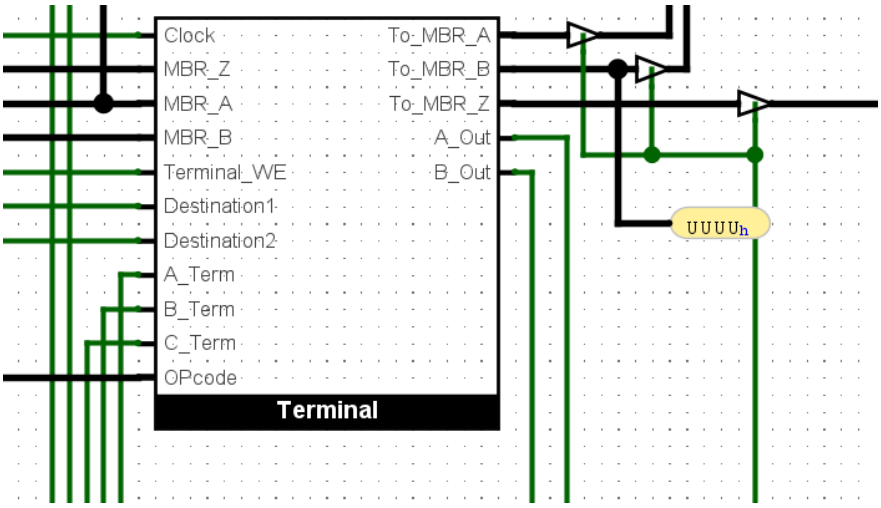

Figure 43: Circuit Diagram for Terminal Connection

I have placed buffers on each output so I control their output when needed and also you can see that all three buffers are connected to one wire, that is so I can save control lines since releasing data from one wire where the other will have no data won't cause any error.

18.11 Control Unit Connection

18.11.1 ROM

The control unit is basically the ROM, it is ROM based. The control unit will output different amount of control lines which will be connected through out the circuit to such as the **WE** signals or RAMs, or the **OE** signals of the RAMs, or the **U/D** signal of the program counter and more. In the current implementation I have used two ROMs of address width 8 bits and data width 32 bits meaning each ROM can take a 8 bit address and can hold a 32 bit data which is to say 32 control lines at a time. I have chosen two ROMs since it seemed easy to manage two rather than one being too large, you can use 1 ROM of 64 bits data width if you want to.

The ROM will receive just a 8 bit address but it is from two different sources, one is the OPcode of the current operation coming from the instruction splitter (the register below it) and the other is from the ROM counter which will be incremented by the clock signal. The ROM counter will be of 4 bits and the OPcode coming will also be of 4 bits combined they will make a 8 bit address. The ROM counter will reset when the instruction is completed and the OPcode will be reset with a constant value of 00. The Current OPcode becomes the the leftmost 4 bits of the address and the ROM counter becomes the rightmost 4 bits of the address, as you can see in the image below. Currently you are seeing only one ROM, but their is one more right below it.

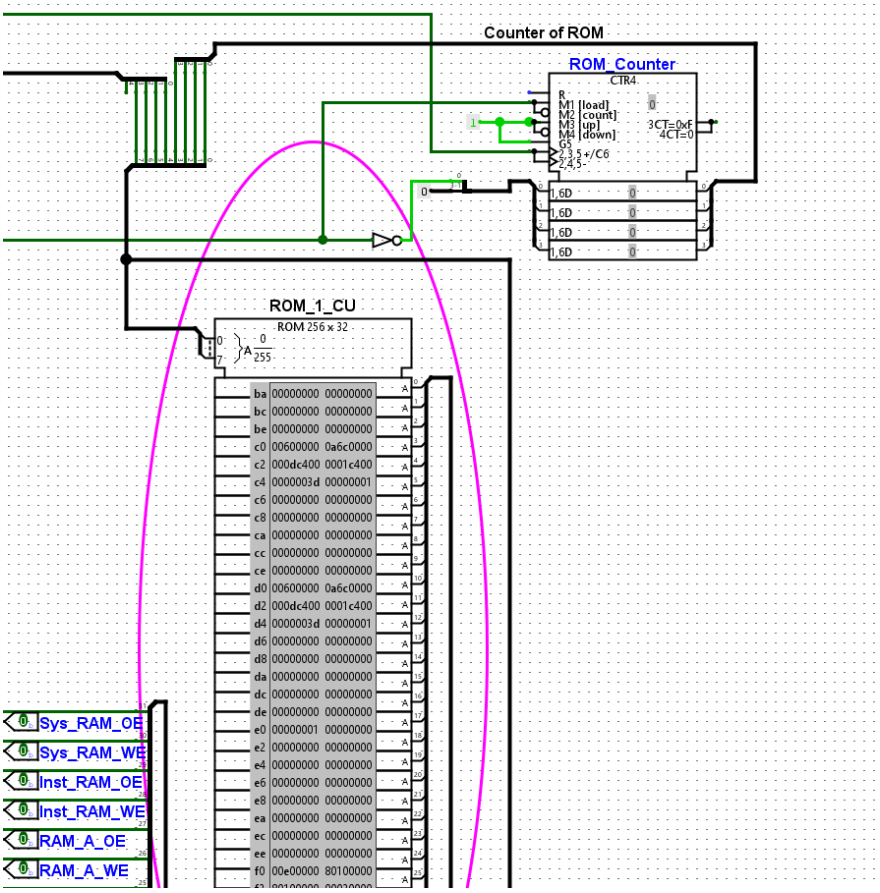

Figure 44: Circuit Diagram for Control Unit Connection

18.11.2 ROM Connection

Here some logic is built for the reset of the counter as you can see. A not gate is placed where I have simply put a reset control line coming from the ROM to the Counter's **Load** input and at the same time the data in the counter is set to 00. So when the ROM outputs the reset signal the counter will reset and the data in the counter will be set to 00, which is equivalent to the reset of the counter. Second ROM is connected in a very similar way since it also needs the same address as the first ROM. The data outputted from the ROM will be sent to the control lines of the circuit. The control lines may be different but the way it works, the input will be same for both.

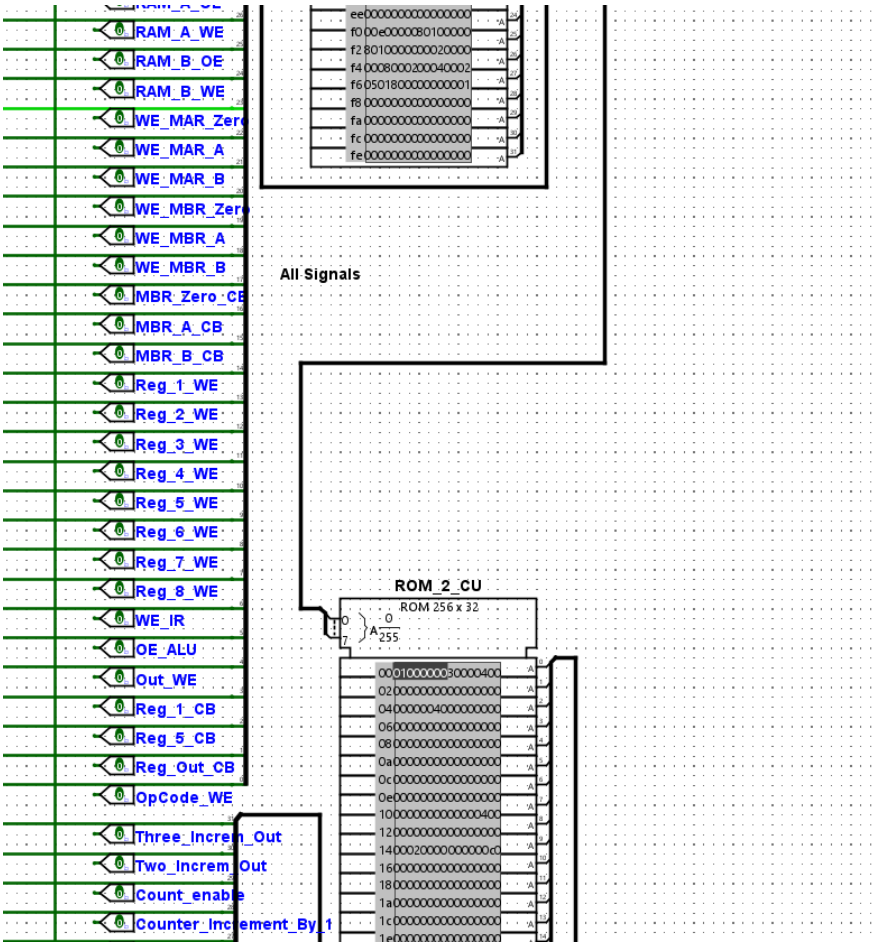

Figure 45: Circuit Diagram for Control Unit Connection

18.11.3 Understanding the ROM

How is the ROM working for all operations? Basically the initial OPcode which is 00 and since the ROM counter is also 00 the ROM will be pointing to it's 00 position wher from that to $0f$ it will have initial fetch instructions. Then as the moment comes when the OPcode is received in the instruction splitter and then into the register below it, the ROM jumps to its sector, meaning the ROM counter will reset while the OPcode will change from 00 to the current one. So for example the OPcode is 02 (hexa) and at that moment the ROM counter will also be reset hence it will be at 00. Combining this to form a 8 bit address the ROM will jump to the 20 sector where now the ROM counter will start incrementing with each clock cycle and will go to $2f$ maximum. This is how the ROM will work for all operations.

18.12 MAIN Circuit

Now that we have discussed all the components and assembeled them too, let's talk about the main circuit. The main circuit will have all the components connected to each other as discussed above.

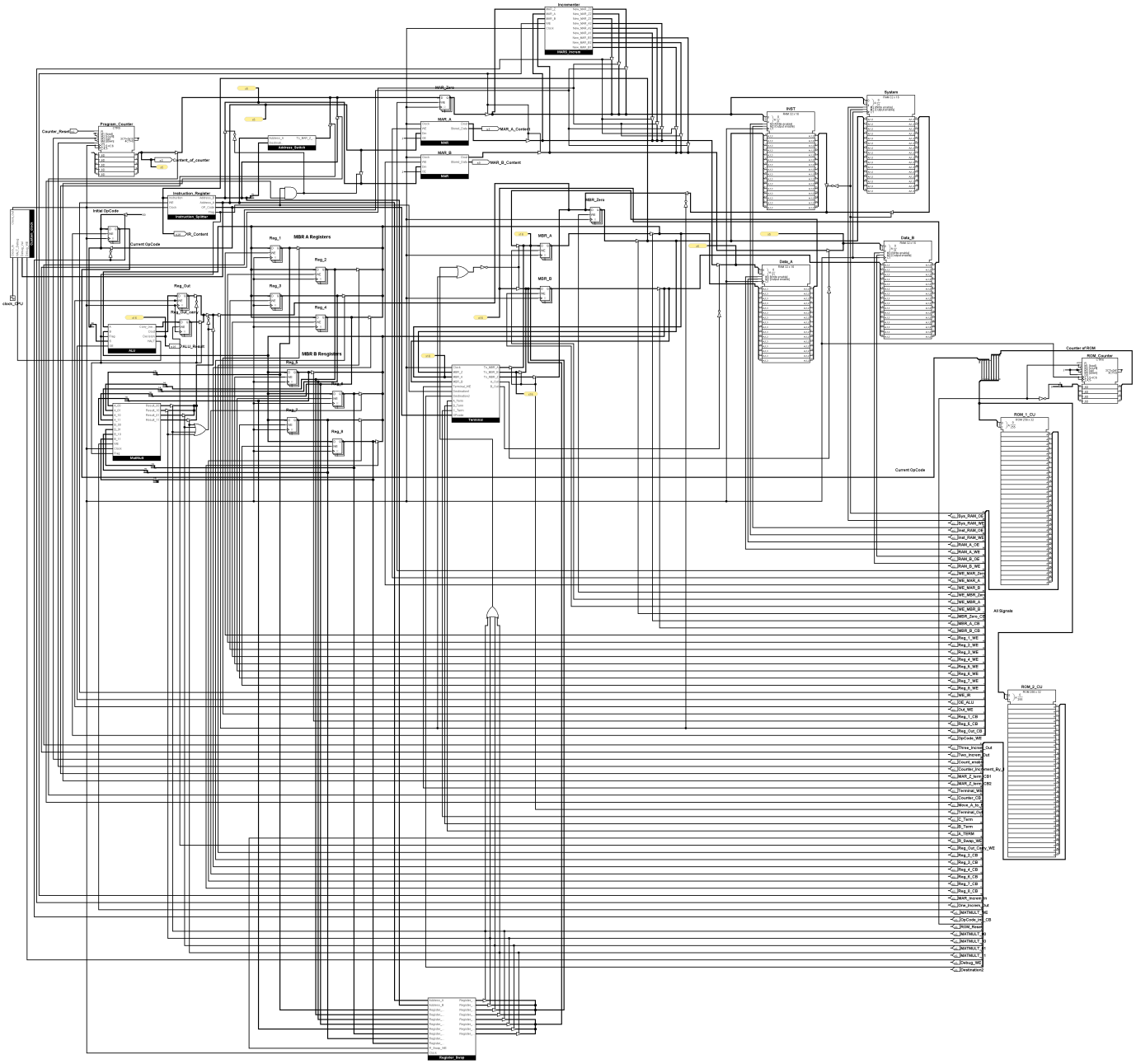

Figure 46: Circuit Diagram for Main Circuit

18.12.1 All Operations

We will discuss most of the operations that this circuit was supposed to do. The operations are as follows:

Operation	OPcode	Flag	Description	Flow
Reserved	0000	0	Reserved for future use	-
Add	0001	0	Adds the data from address A and B from RAM A and B	Inst splitter → MAR's → RAM's → MBR's → Register Bank A and B → ALU → Output Register.
Subtract	0001	1	Subtracts the data from address A and B from RAM A and B	Inst splitter → MAR's → RAM's → MBR's → Register Bank A and B → ALU → Output Register.
Multiply	0011	0	Multiplies the data from address A and B from RAM A and B	Inst splitter → MAR's → RAM's → MBR's → Register Bank A and B → ALU → Output Register.
Comparision	1010	0	Compares the data from address A and B from RAM A and B	Inst splitter → MAR's → RAM's → MBR's → Register Bank A and B → ALU → Output Register.
AND	1011	0	AND's the data from address A and B from RAM A and B	Inst splitter → MAR's → RAM's → MBR's → Register Bank A and B → ALU → Output Register.
NOT A	1101	0	NOT's the data from address A from RAM A	Inst splitter → MAR's → RAM's → MBR's → Register Bank A and B → ALU → Output Register.
XOR	1100	0	XOR's the data from address A and B from RAM A and B	Inst splitter → MAR's → RAM's → MBR's → Register Bank A and B → ALU → Output Register.
NOP	1110	0	No operation	-
Store	0100	0	Stores the data from the output register in the system RAM at address A	Inst splitter → MAR's → RAM's → MBR's → System RAM.
Store Overflow	0100	1	Stores the overflow data from the output register in the system RAM at address A	Inst splitter → MAR's → RAM's → MBR's → System RAM.
Mem Swap	0101	0	Swaps the data from the address A and B from RAM A and B	Inst splitter → MAR's → RAM's → MBR's → Terminal → MBRs → RAMs.
Reg Swap	0110	0	Swaps the data from the register bank A and B (one value both each)	Inst splitter → Register Bank A and B → Register Swapper → MBR's → Register Bank A and B.
Block Load	0111	0	Loads four consecutive data from RAM A and B starting from address A and B into banks	Inst splitter → MAR's → RAM's → MBR's → MAR's Incrementer → RAM's → Register Bank A and B.
Block Store	1000	0	Stores the data from the banks into the four consecutive data from the MatMults output bank to system RAM starting from Address A	Inst splitter → MAR's → RAM's → System RAM → MAR's Incrementer → RAM's → System RAM.
MatMult	1001	0	Multiplies the data from the banks and stores the result in the MatMults output bank	Inst splitter → MAR's → RAM's → Register Bank A and B → MatMult.
Halt	1110	1	Stops the circuit	-
Move A,B	1111	0	Move value in System Ram, at Address B to value in Ram A address A.	Inst splitter → MAR's → RAM's → MBR's → Terminal → MBR's → RAM's.
Move A,B(with flag).	1111	1	Move value in System Ram, Address B to value in Ram B address A.	Inst splitter → MAR's → RAM's → MBR's → Terminal → MBR's → RAM's.

Table 10: Operations

19 Conclusion

In this project, we designed a 16-bit CPU using Logisim, detailing its components, interconnections, and operations. The CPU supports a wide range of functionalities, including arithmetic, logical, memory, and block operations. It performs tasks such as matrix multiplication, register and memory swapping, data transfer between memory locations, and comparison operations. Additional capabilities include bitwise operations like NOT, XOR, and AND, as well as NOP (no operation) and halt functions.

The CPU can load and store data blocks in memory, manage overflow data, and execute operations with flags. It supports efficient memory management, such as moving data between locations and swapping memory blocks or register values. With features like matrix multiplication and halting the circuit, the CPU demonstrates versatility in processing tasks. Its robust design incorporates essential operations for efficient data manipulation and control flow, making it suitable for complex computational tasks.

20 Future Work

The 16-bit CPU can be further enhanced with additional features and optimizations. Future work may include:

- **Instruction Set Expansion:** Introduce new instructions for advanced operations or specialized tasks.
- **Optimization:** Enhance the CPU's performance by optimizing its components or operations.
- **Parallel Processing:** Introduce parallel processing capabilities for faster computation such as Pipelining.
- **User Interface:** Develop a user-friendly interface for input/output operations or data visualization.

References

[1] Magnitude Comparator in Digital Logic. GeeksforGeeks.

[2] Arithmetic operations in logisim

[3] Partial Register Swapping logic by Umama Muhammad

[4] Overall guide in the making of the CPU by Sir Jawwad Farid