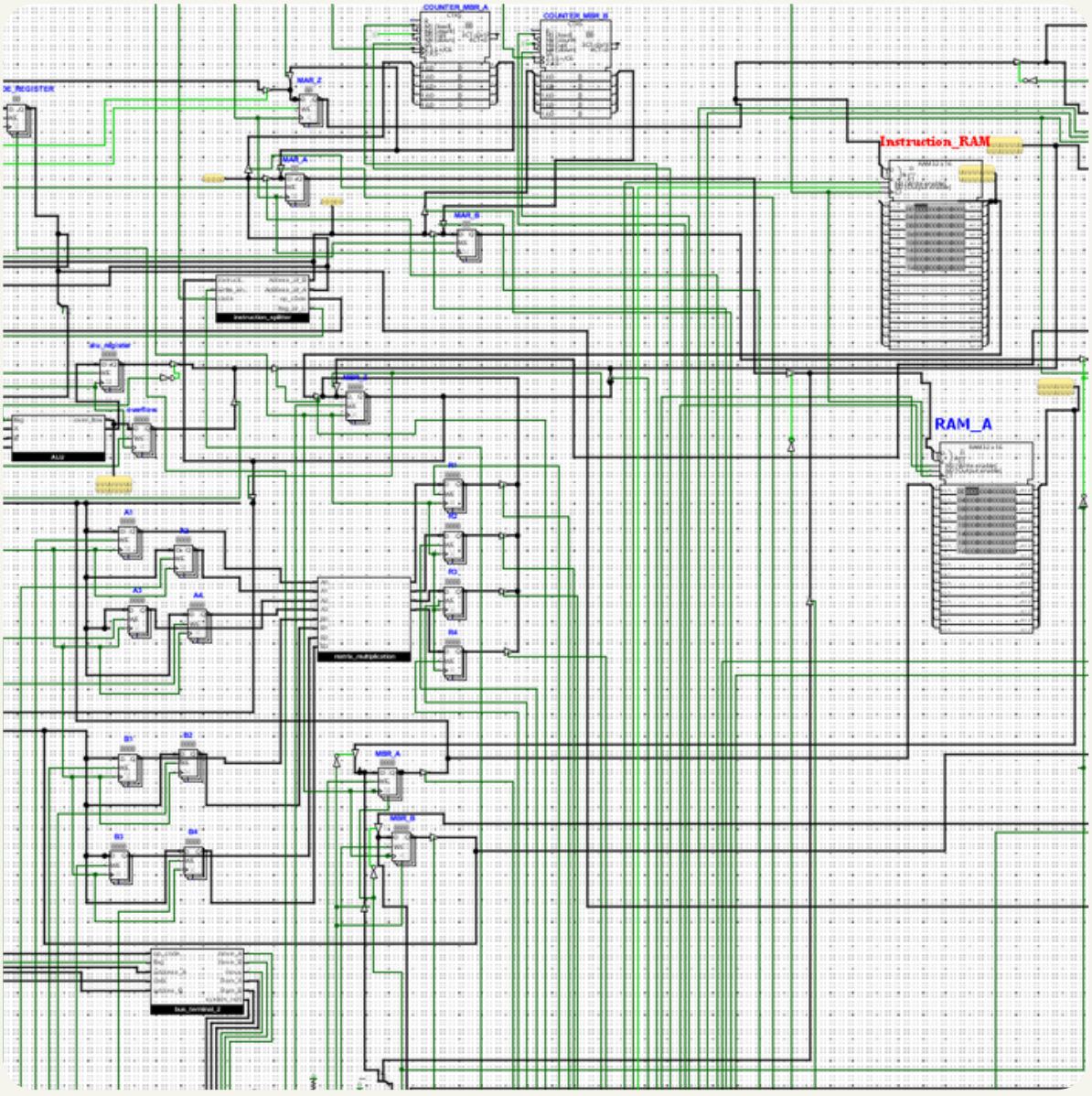
A SHORT GUIDE

TO BUILD 32-BIT CPU IN LOGISIM



BY ABDULLAH SIRAJ KHAN

TABLE OF CONTENTS — CUSTOM CPU GUIDE

3.2	Building an Arithmetic Logic Unit (ALU)	Combining arithmetic & logic operations
4	Instruction Handling & Bus Architecture	Control signal and data flow management
4.1	Implementing an Instruction Splitter	Decoding instructions into opcode and operands
4.2	Utilizing Bus Terminals for Efficient Communication	Interconnecting RAM, ALU, and control logic
5	Advanced Operations	Enhancements beyond basic CPU functionality
5.1	Matrix Multiplication in CPU Design	Using the CPU to multiply 2D matrices (16x16 → 32-bit output)
6	Constructing the Core CPU	Full CPU system design and build process
6.1	Building a 16-bit CPU in Logisim	Step-by-step walkthrough
6.2	Scaling Up to a 32-bit Architecture	Upgrading buses, ALU, RAM, ROM, etc.

1. Introduction

This guide documents the complete design and development process of a custom-built 16-bit CPU, later scaled to 32-bit, using Logisim. It explains how basic logic gates and modules come together to form a fully functional processor capable of performing real computations. The project reflects both learning and application of digital logic and architecture principles.

1.1 Purpose of the Guide

The aim of this guide is to walk readers through the structure, logic, and functionality of a CPU built from the ground up. It serves as a reference for students and enthusiasts who wish to understand how arithmetic units, instruction decoding, and control logic interact to execute meaningful tasks. Every component, from simple adders to matrix multiplication support, is covered with practical insight.

1.2 Benefits of Building a Custom CPU

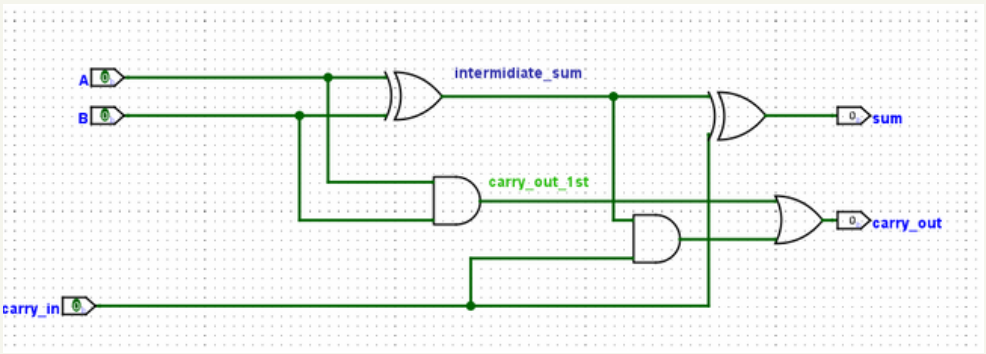
Designing a CPU manually builds a deep understanding of computer architecture, binary logic, data flow, and instruction cycles. It strengthens core concepts like ALUs, registers, memory addressing, and bus systems — knowledge that's essential for both software engineers and hardware designers. Moreover, it sharpens problem-solving, debugging, and system-level thinking.

2. Core Arithmetic Units

The Arithmetic Unit is the heart of any CPU. It performs mathematical operations like addition, subtraction, multiplication, and comparisons. These operations are essential for executing most types of instructions. In this section, we explain how each core unit was designed using basic digital logic in Logisim.

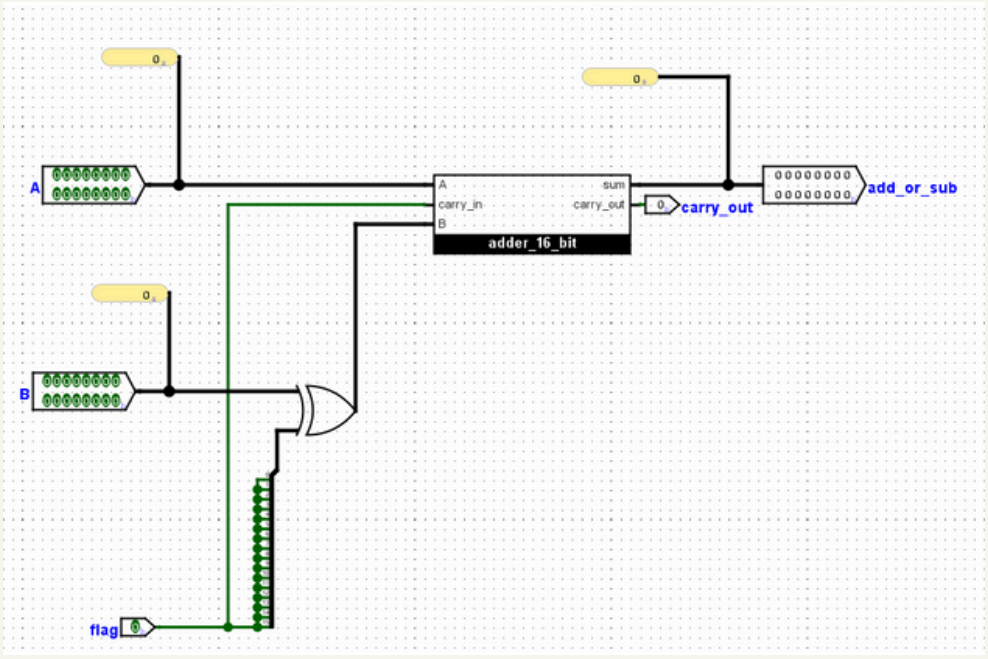
2.1 Designing an Adder

We built a 1-bit full adder as the base unit, then cascaded multiple adders to form 16-bit and 32-bit adders. Each full adder handles sum and carry bits, enabling accurate binary addition across all bit widths. The core idea to built it the fact that an **OR** gate acts like a adder.



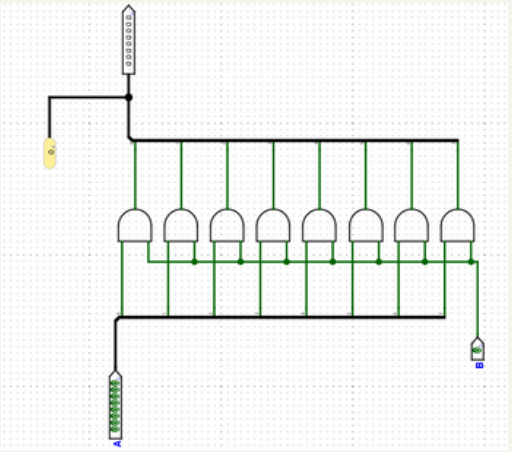
2.2 Implementing a Subtractor

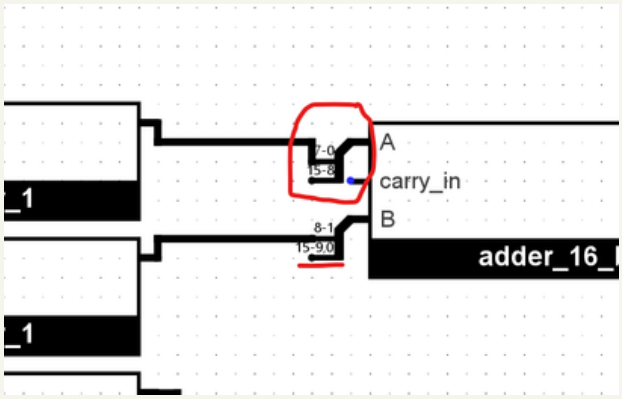
The subtractor was implemented using 2's complement logic. Instead of building a separate module, we reused the adder by inverting the second operand and adding 1, which efficiently performs subtraction.



2.3 Constructing a Multiplier

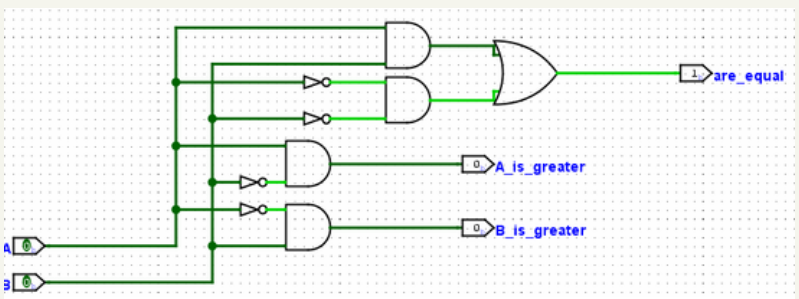
We designed a shift-and-add multiplier that multiplies two binary numbers using a looped structure of adders and shifters. This setup supports both 16-bit and 32-bit multiplication, producing accurate multi-bit results. First we will make a 1-bit by 8-bit multiplier than we will make use it for others. The key idea is the shifting done by using splitters.





2.4 Creating a Comparator

The comparator checks if one binary number is less than, equal to, or greater than another. It was built using subtraction followed by logic gates to set condition flags, useful for branching and decision-making. Our comparator is made using NOT,OR and AND gates.



3. Essential Digital Components

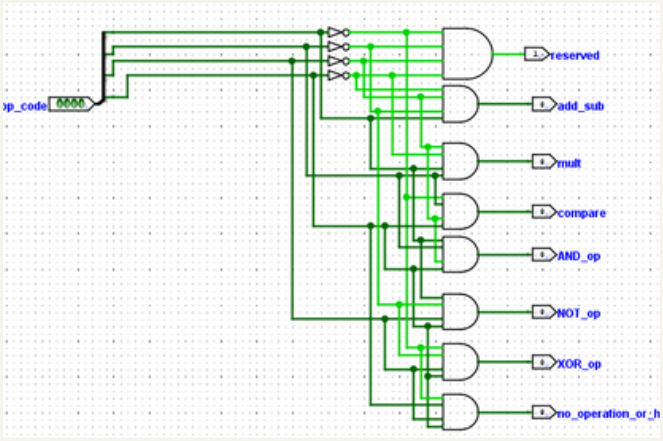
This section explains important logic modules that give the CPU its core processing power. While the Arithmetic Logic Unit (ALU) performs all essential operations, internal elements like multiplexers support its flexible behavior.

3.1 Understanding Multiplexers (MUX)

In this CPU, multiplexers were used **only inside the ALU** to switch between different arithmetic and logic operations. For example:

- Selecting between ADD, SUB, AND, OR based on control input
- Routing internal signals to determine the correct output
- Although MUXes are commonly used in data paths, here they are strictly internal to the ALU and not used elsewhere like register selection or memory routing.

I have build a 4 to 8 MUX . It takes op code of 4 bit as input and selects 1 from 8 outputs of ALU. The instruction format used in my CPU is 1,5,5,5. the MSB is represents flag, and the other sets of 5 bits after it are used for address of B, B and op code respectively.

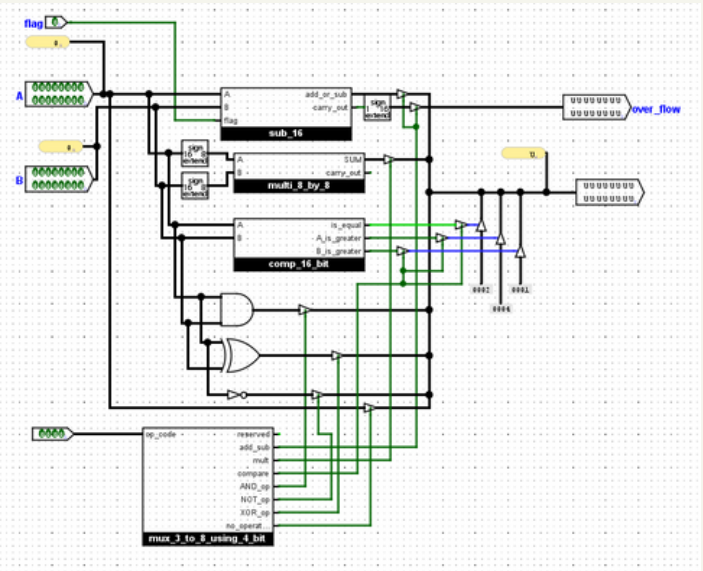


3.2 Building an Arithmetic Logic Unit (ALU)

The ALU handles all core operations such as addition, subtraction, AND, OR, and NOT. It's controlled by a 4-bit opcode:

- Each opcode maps to a specific operation
- MUX inside the ALU direct which logic block to activate
- Flags like zero, negative, and carry can be set for condition checking

The ALU is central to the CPU's processing ability and connects directly to the control unit and register buses.



4. Instruction Handling & Bus Architecture

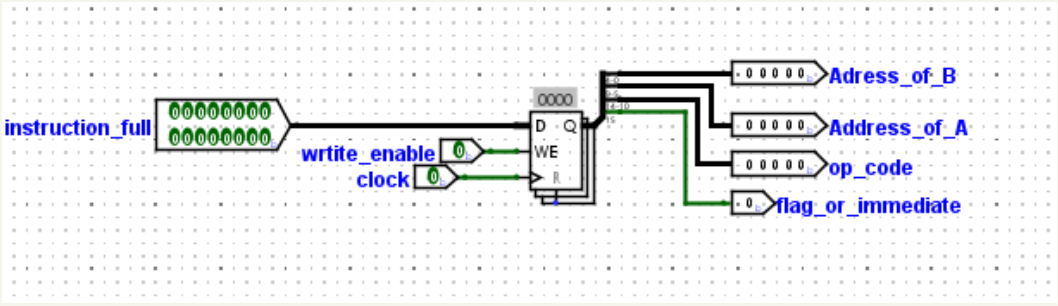
Efficient instruction execution requires decoding opcodes and managing the flow of data between components. This section explains how the CPU splits instructions and uses bus terminals to connect memory, the ALU, and the control unit.

4.1 Implementing an Instruction Splitter

Each instruction is fetched from Instruction_Ram and passed through an **instruction splitter**, which separates:

- The **opcode** (e.g., ADD, SUB, LOAD)
- The **operands** (register addresses or immediate values)

This separation is crucial for decoding and routing the correct control signals. The splitter outputs go to the control logic and directly influence the behavior of the ALU and RAM modules.
Instruction splitter is designed to split the 16 bit instruction.

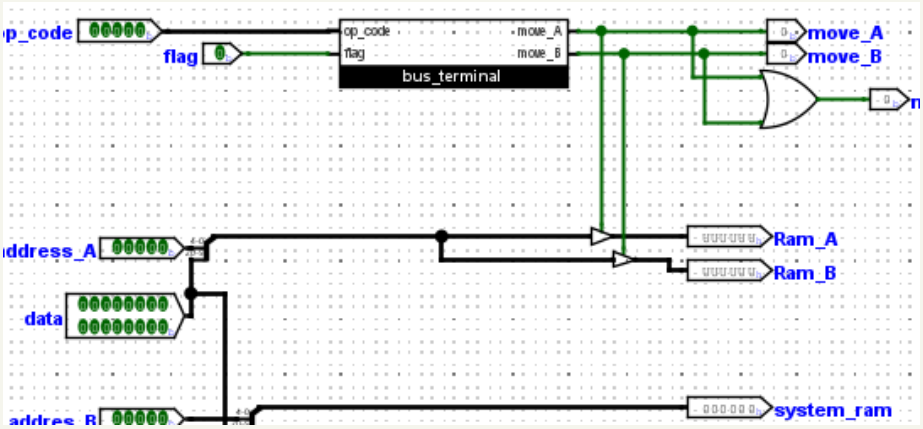


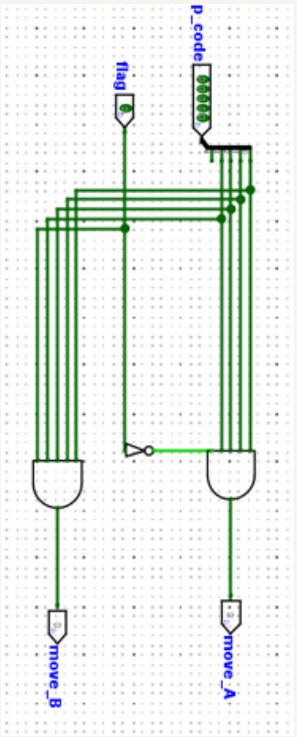
4.2 Utilizing Bus Terminals for Efficient Communication

Bus terminals were used to connect different parts of the CPU:

- **Data buses** transfer values between RAM, ALU, and registers
- **Address buses** specify memory locations
- **Control buses** carry signals to trigger read/write or select operations

Using named bus terminals in Logisim made the design clean and modular, reducing wire clutter and making debugging easier.





5. Advanced Operations

Beyond basic arithmetic and logic, this CPU also supports complex tasks such as matrix multiplication. These advanced operations demonstrate the CPU's flexibility and its ability to process structured data efficiently.

5.1 Matrix Multiplication in CPU Design

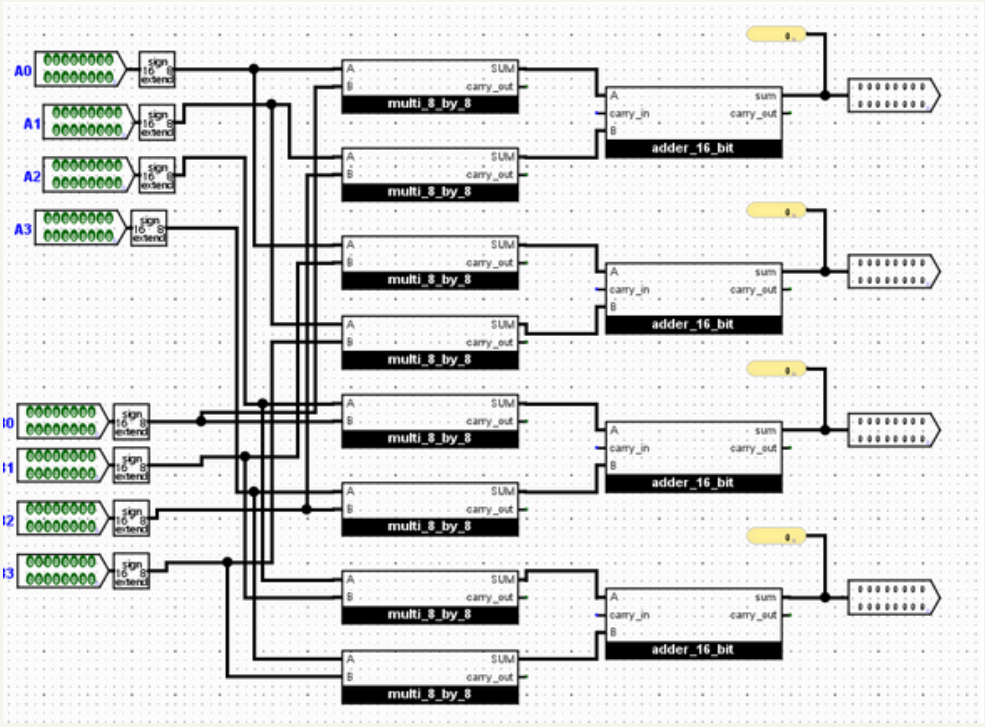
We implemented **16×16-bit matrix multiplication**, where:

- Each matrix is a 2×2 grid of 8-bit integers
- The CPU multiplies row and column elements using internal ALU operations
- The result is a 16-**bit output** per element in the result matrix

This was achieved by sequencing multiple ADD and MUL operations manually via instruction RAM. It simulates how real processors handle structured data in scientific computing or graphics.

Why it matters:

This feature showcases the CPU's ability to perform more than just simple operations — it demonstrates how general-purpose logic can be reused to solve complex tasks with control flow and memory management.



6. Constructing the Core CPU

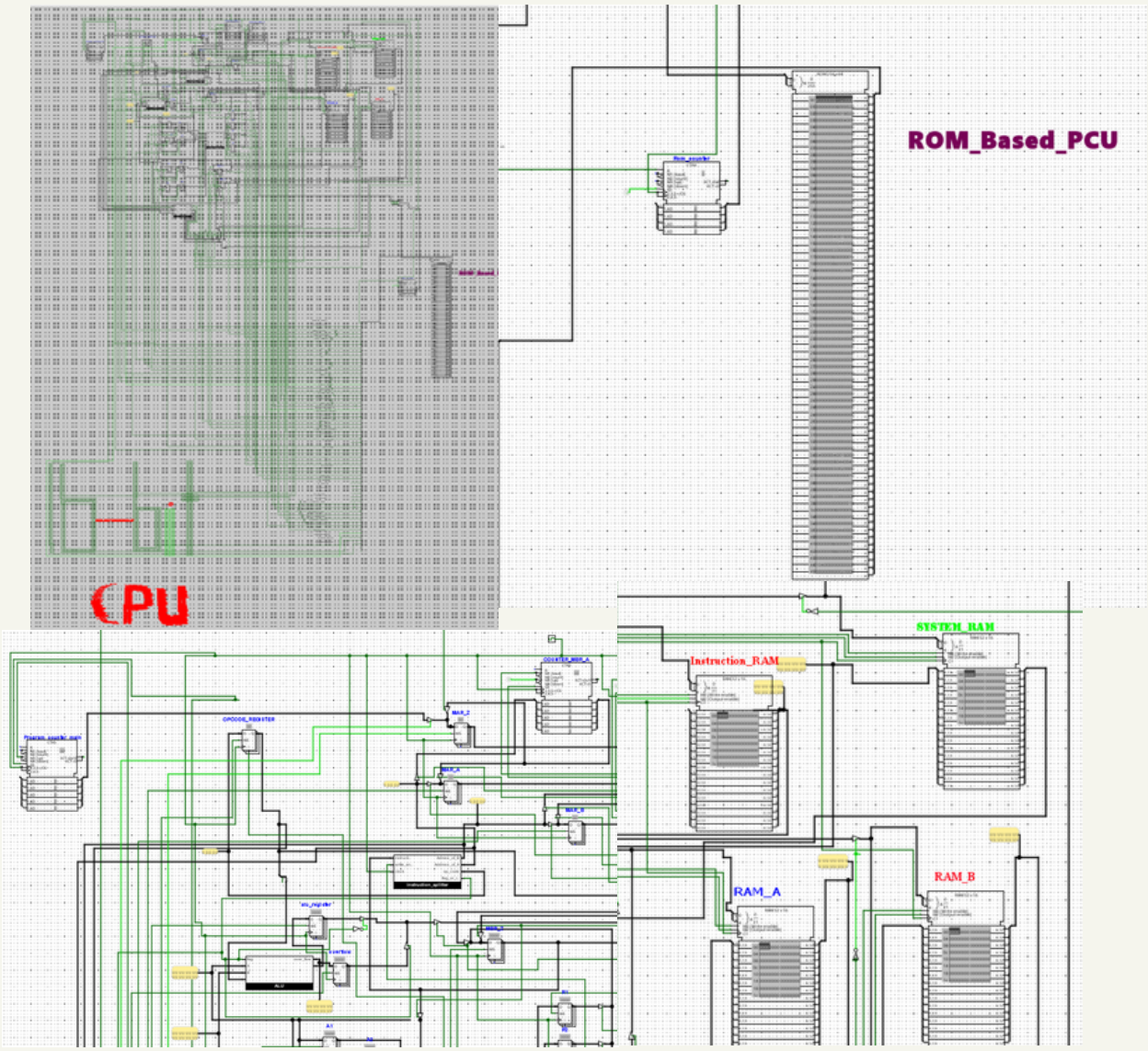
This section outlines how all components — arithmetic units, instruction handling, and memory — were combined to build a functional CPU. It also covers how the design was upgraded from 16-bit to 32-bit architecture.

6.1 Building a 16-bit CPU in Logisim:

The project began with a 16-bit CPU, constructed from scratch in Logisim. Key features included:

- A **16-bit ALU** supporting multiple operations
- **RAM A and B** for data storage
- **instruction RAM for** instruction storage.
- A **ROM-based control unit** that stores instructions, which holds microcoded instructions .
- Registers, instruction splitter, and output display

This version laid the foundation for the CPU's instruction execution cycle and verified the working logic .The data flow starts from the program counter which start from 0, it gives the MAR_Z the address of the instruction from instruction Ram. Then instruction Ram outputs the instruction which goes to MBR_Z and then to splitter. The instruction splitter split the instructions into the defined format and send the respective element to its place. the MAR_A and MAR_B are used to hold addresses of the respected value A and B. then the values of A and B are fecthed from the RAM_A and RAM_B which are stored in register . These value then go to ALU which performs respective task. The result is stored in system ram.

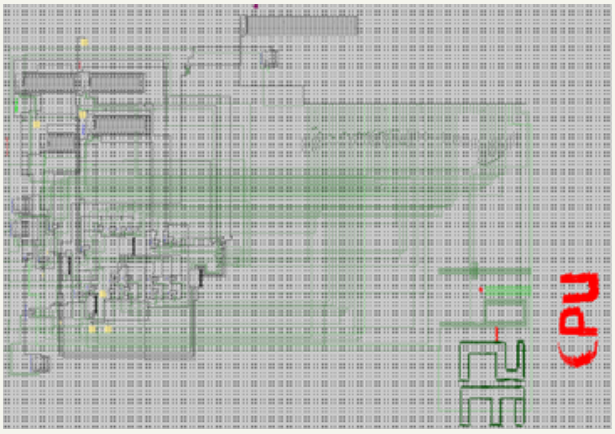


6.2 Scaling Up to a 32-bit Architecture

After successful testing, the CPU was scaled up to 32 bits. This included:

- Doubling the bit width of all buses, ALU, RAM, ROM, and output
- Modifying the **instruction format** for larger operands
- Expanding the control unit to handle wider data paths

Please consider the drawback that 32-bit CPU no longer supports the matrix_multiplication operations due to lagging of logisim evalution, which made me to dissconnect the buses of matrix rigesters.



7.1 Automating Execution and Control Signals

The CPU was made fully automatic using a **ROM-based microcoded control unit**, closely resembling how real-world CPUs operate.

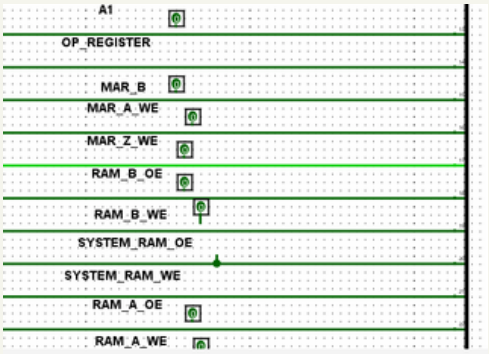
Here's how the automation works:

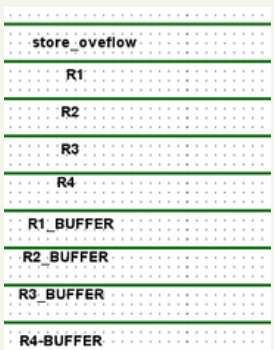
- A **truth table (microcode)** is written into ROM. Each row corresponds to a micro-instruction that activates specific control signals for one clock cycle.
- The **instruction opcode** and an internal **step counter** (micro-instruction counter) are combined to form the address for the ROM.
- At each clock tick, the ROM outputs control signals that:
  - Enable ALU operations
  - Control read/write for RAM A and B
  - Manage data flow to output or registers
- Once all steps of an instruction are executed, the **step counter resets**, and the **program counter** moves to the next instruction.

As a result:

- **Only inputs in RAM A, RAM B, and Instruction ROM are needed**
- The entire CPU runs the program **automatically without any manual switching**
- This mirrors real CPUs where a **control unit orchestrates everything internally**

This design made the CPU highly modular, testable, and realistic in behavior — capable of running complete tasks like matrix multiplication or arithmetic sequences without intervention.





8. Capabilities & Applications

This CPU isn't just a collection of logic gates — it's a fully functional processing unit capable of running structured programs. With automated control and support for both arithmetic and logic operations, it simulates core behavior found in real microprocessors.

8.1 What This CPU Can Do

Here's a summary of the CPU's key capabilities:

✅ **Arithmetic Operations**

- Performs 16-bit and 32-bit addition, subtraction, and multiplication
- Supports complex tasks like matrix multiplication using a sequence of instructions

✅ **Logic Operations**

- Executes bitwise AND, OR, and NOT operations
- Supports condition checking via comparison flags (e.g., zero, carry)

✅ **Instruction Execution**

- Decodes and processes instructions from ROM automatically
- Follows a microcoded control logic for step-by-step execution

✅ **Memory Interaction**

- Reads/writes from **RAM A and B**
- Communicates efficiently using bus terminals

✅ **Automation & Realism**

- Fully automated execution loop
- Mimics behavior of real CPUs via instruction ROM, control ROM, and counters

These features make the CPU suitable for demonstrating **instruction flow, ALU design, control units, and automation concepts** — ideal for learning or showcasing in embedded systems, digital logic courses, or even research portfolios.

9. References & Acknowledgements

Behind every successful project lies a blend of learning, collaboration, and guidance. This section lists the valuable resources and individuals that supported the completion of this custom CPU design.

9.1 Cited Resources

The following references were instrumental in helping understand logic circuits, control units, and CPU architecture:

- **YouTube Playlists & Videos**
  - Ben Eater-style 8-bit CPU Design Playlist – A comprehensive guide to building an 8-bit breadboard CPU.youtube.com/playlistlist=PLowKtXNTBypGqImE405J2565dvjafglHU
  - Jawwad Farid – FinanceLearningAcademy: Building a Simple CPU in Logisim – A practical, beginner-friendly tutorial specifically focused on Logisim-based CPU design.
- **Software Documentation & Forums**
  - *Logisim Evolution* documentation and community forums – for learning component behavior and emulator configuration.

9.2 Acknowledging Contributions

- **Yahya**, for sharing his own CPU design and ideas that sparked improvements in mine.
- **Open-Source Community**
- Gratitude to the creators and contributors of *Logisim Evolution*, and all those who openly share educational content online.
- **AI Support**
- **Everyone Who Inspired me.**