Chittagong University of Engineering & Technology

Department of Electronics and Telecommunication Engineering

Project Report

Name of the Project:

SAP-1 CPU Implementation

Course No.	:	ETE 404
Course Title	:	VLSI Technology Sessional
Level	:	4
Term	:	I
Date of submission	:	06/10/2025

Submitted By:
Turja Talukder
ID: 2008022

Submitted To:
Arif Istiaque Rupom
Assistant Professor,
Dept. of ETE, CUET

Name of the Project

SAP-1 CPU Implementation

Objectives

- To design and implement a fully functional Simple-As-Possible (SAP-1) CPU using Logisim Evolution.
- To demonstrate the core operations of a processor, including the fetch–decode–execute cycle.
- To implement a hardwired control unit capable of managing all CPU operations automatically.
- To integrate a ROM-based bootloader that loads machine code programs into memory automatically.

Required Software

1. Logisim-evolution.

Introduction

The instruction cycle is the repetitive process a CPU follows to run a program, consisting of three main steps: fetch, decode, and execute. During fetch, the CPU retrieves the next instruction from memory; in decode, it interprets the instruction; and in execute, it performs the required operation, involving key components like the Program Counter, MAR, MDR, CIR, Control Unit, and ALU. In this experiment, a basic processor architecture was built in Logisim, incorporating elements such as a 4x16 decoder, SRAM, instruction register, and general-purpose registers, all connected via a common bus. Rather than using an automatic control unit, the processor was manually controlled through switching signals at each T-state, allowing a hands-on understanding of data flow and micro-operations. A simple instruction set was created to support basic data loading and arithmetic operations, demonstrating fundamental CPU functioning.

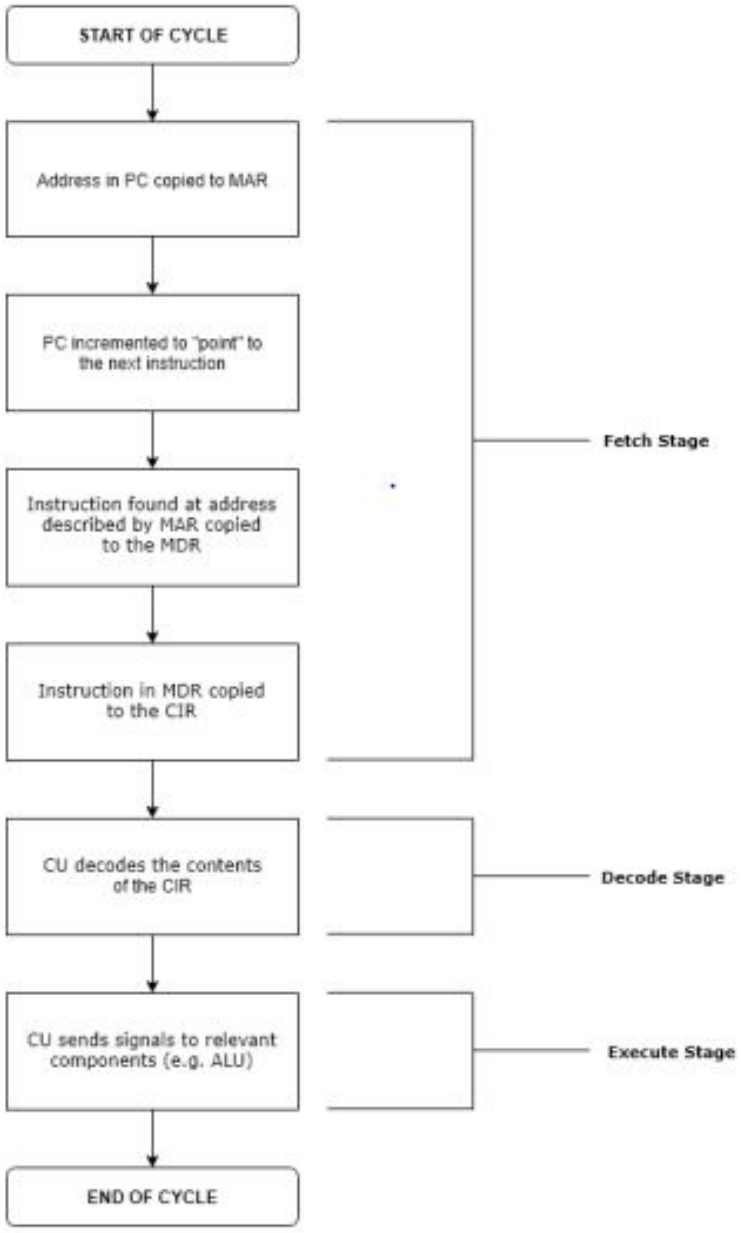

Figure 1: Individual stages in a fetch-decode-execution cycle

Design Process

4-to-16 Decoder

A decoder is a combinational logic circuit that takes n input selection bits and produces up to 2 output lines. For every unique combination of input bits, only one output line is activated at a time. In this experiment, a 4-to-16 decoder is used, which means it has 4 input lines and 16 output lines. These 16 outputs serve as address selectors to access the 16 distinct locations in the RAM, which will be designed next. The circuit diagram of the decoder is illustrated in Figure 2.

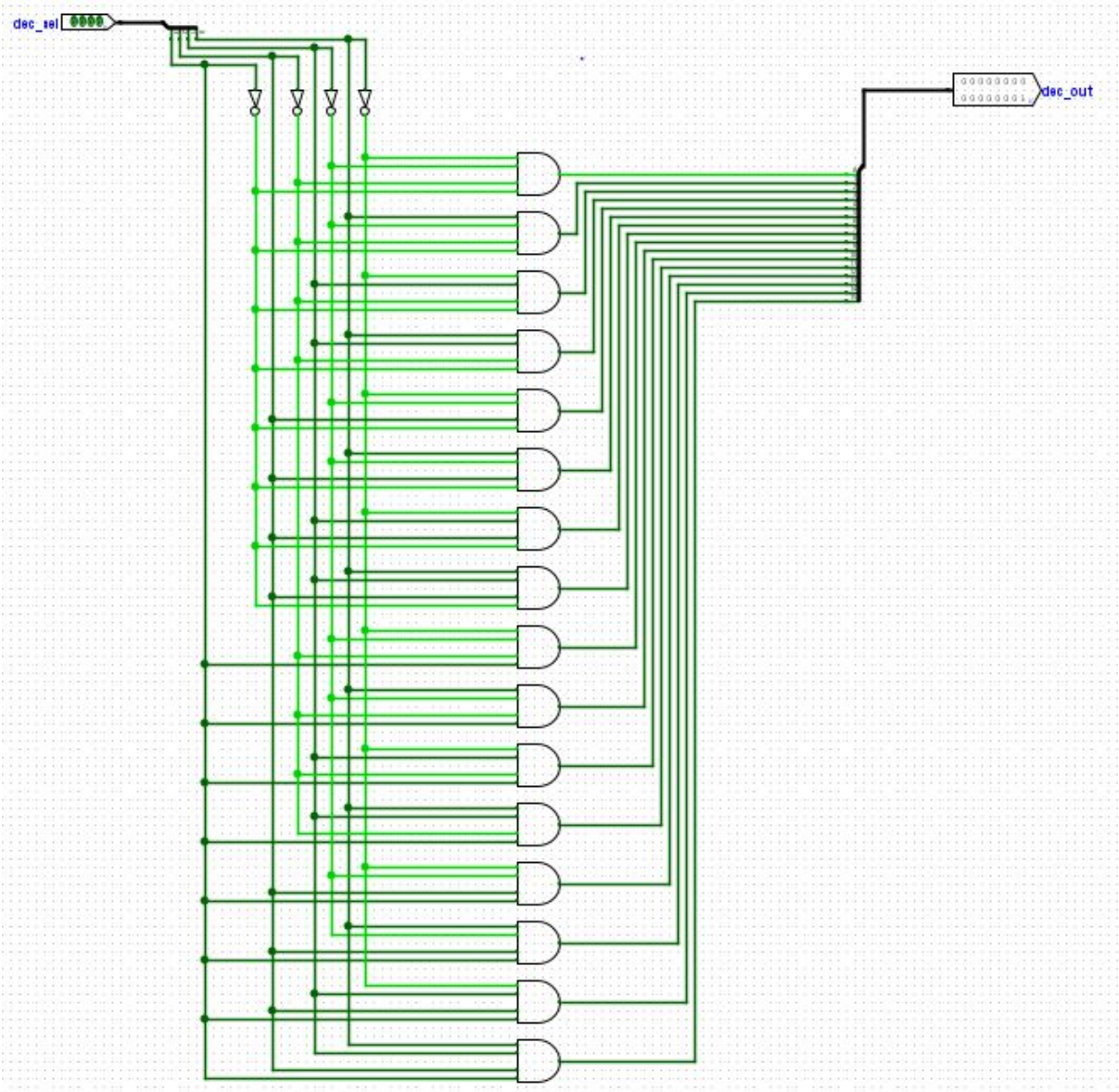

Figure 2: Schematic Diagram of 4-to-16 decoder

Program counter

The Program Counter (PC) shown in the image is a 4-bit counter used to store the address of the next instruction to be fetched from memory. It consists of four flip-flops

connected in a chain, where each flip-flop holds a single bit of the address. The counter is controlled by a clock signal (clk), which increments the PC value on each clock cycle. A reset signal (`pc_reset`) is used to initialize the counter to a predefined value (usually 0), while the `pc_on` control signal allows the counter to function or hold its current value. The `pc_out_en` signal is used to enable the output of the current PC value, which is then sent as the address for memory access. This simple yet effective design ensures that the processor keeps track of the next instruction to be executed during the fetch phase of the fetch-decode-execute cycle.

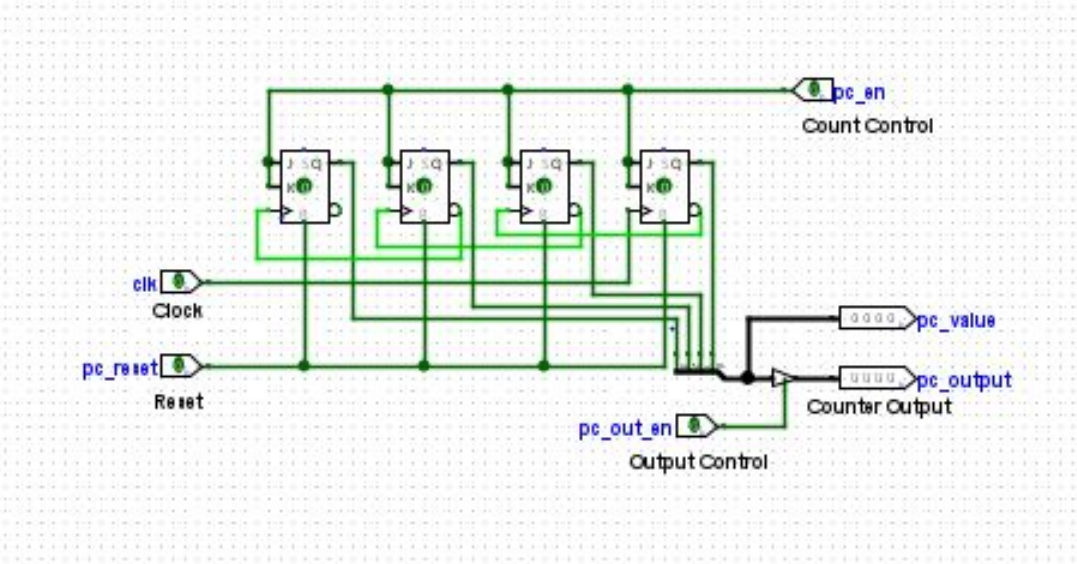

Figure 3: Schematic Diagram of Program Counter

Registers

The architecture shown in the image represents an 8-bit register constructed using eight flip-flops, where each flip-flop holds one bit of data. The register is controlled by three main signals: `reg_in_en`, which enables the input of data into the register; `reg_clk`, the clock signal that synchronizes the operation of each flip-flop; and `reg_out_en`, which enables the output of the stored data. The 8-bit input is provided to the register via the `reg_in` signal, and when `reg_in_en` is active, the data is loaded into the flip-flops. The output, `reg_out`, can be accessed when `reg_out_en` is enabled, allowing the stored data to be sent to `reg_int_out`. This register architecture plays a vital role in storing temporary data or intermediate results within the processor during operations. understanding the core principles of CPU functionality.

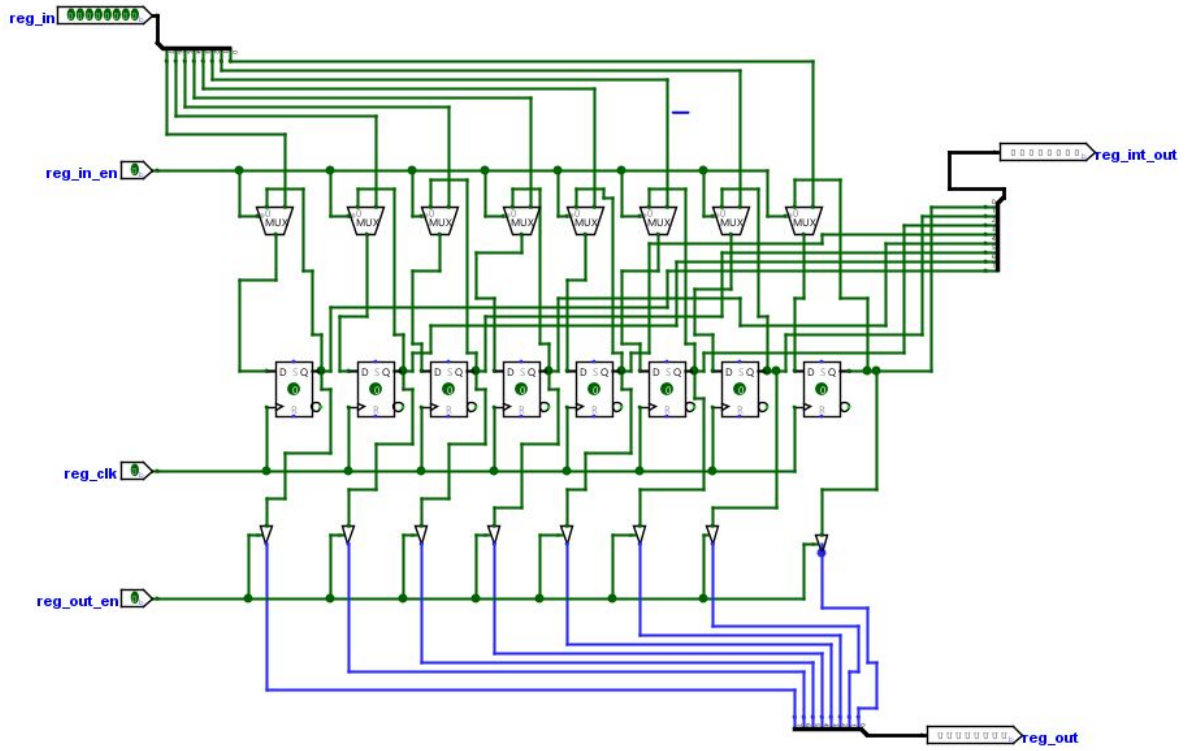

Figure 4: Schematic Diagram of Register

Instruction Register (IR)

The instruction register stores the instruction that is currently being executed. It divides the data from the BUS into two 4-bit sections: the upper 4 bits represent the opcode, while the lower 4 bits correspond to the operand or memory address.

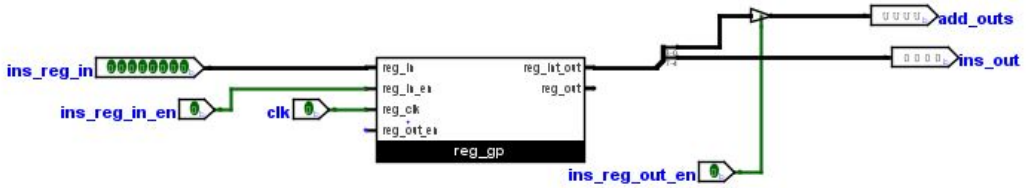

Figure 5: Schematic Diagram of Instruction Register (IR)

Random Access Memory (RAM)

RAM is a memory storage area accessible to the processor during program execution. Before a program runs, it is loaded and stored in RAM, allowing the CPU to quickly read and write data as needed. The RAM implementation began with the design of a single memory cell utilizing an 8-bit register, which included control signals for chip select (cs), write enable (`wr_en`), and read enable (`rd_en`). A 4-to-16 decoder was utilized to choose one memory cell out of sixteen according to a 4-bit address input. All memory cells were linked to a common data bus, with tri-state buffers managing the read operations.

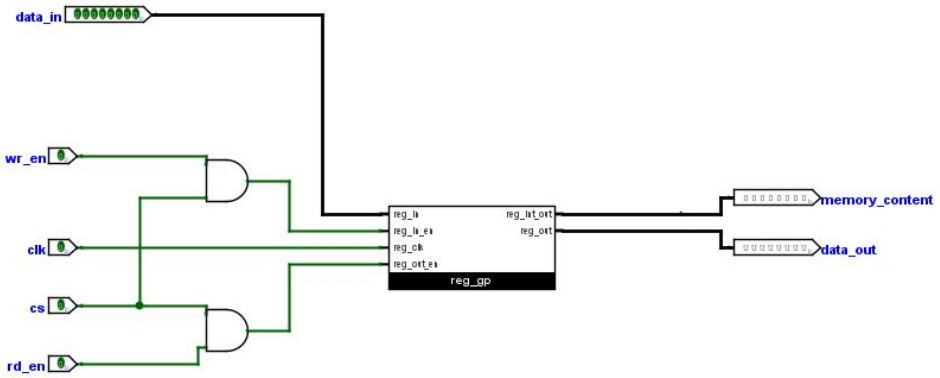

Figure 6: Schematic Diagram of a single SRAM cell

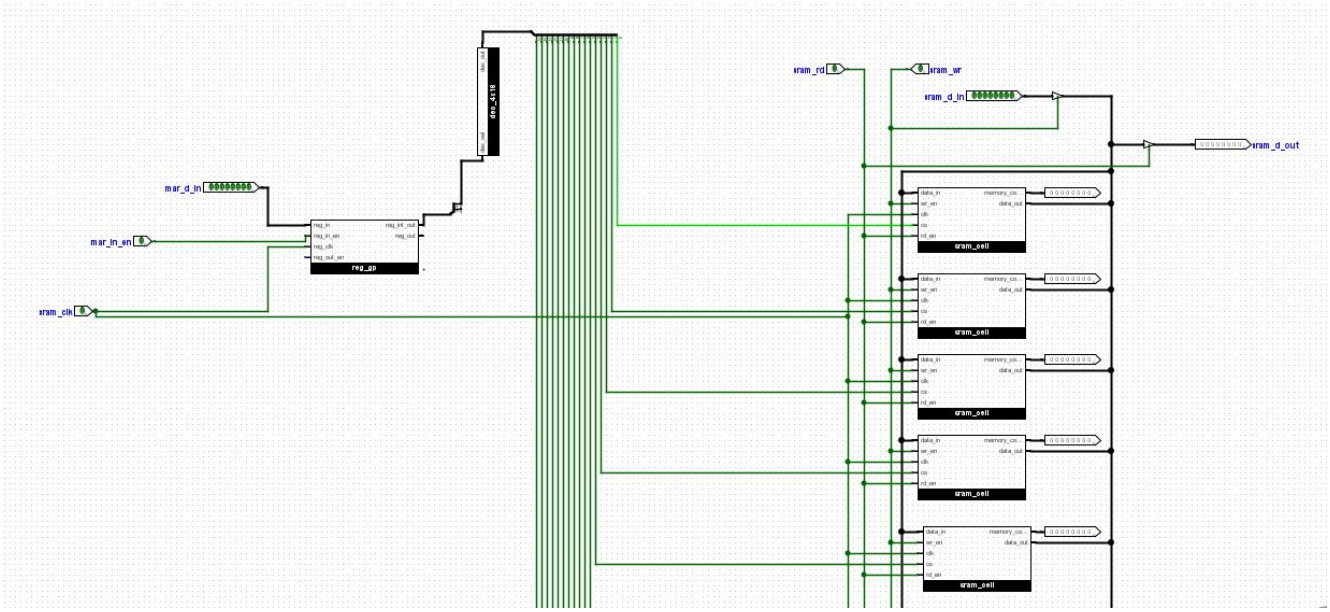

Figure 7: Schematic Diagram of RAM part-I

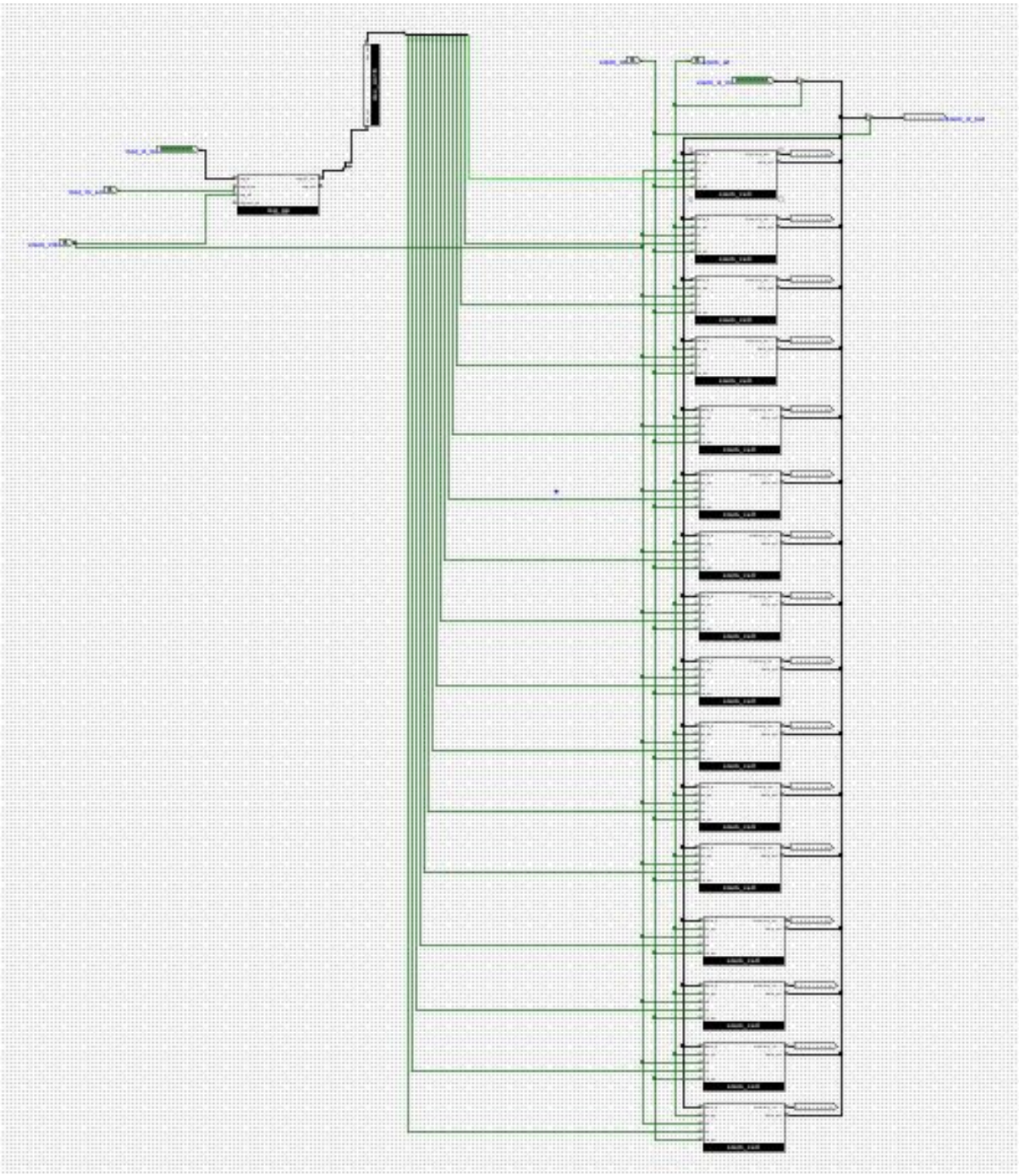

Figure 8: Schematic Diagram of RAM part-II

Arithmatic Logic Unit (ALU)

The Arithmetic Logic Unit including shifter and rotater is responsible for performing arithmetic and logical operations within the processor. It takes two input registers, `reg_a` and `reg_b`, which hold the data to be operated on, and processes them based on the operation. The ALU can perform operations such as addition or subtraction, with the result output as `alu_output`. Additionally, the ALU generates a carry-out signal, `alu_carry_out`, when required for arithmetic operations. The ALU interacts with a shifter and a rotator, which perform shift and rotate operations on the data. The shift operation is controlled by the `shift_direction` and `shift_amount`, while the rotation operation is controlled by the `rotate_direction` and `rotate_amount`. The results of these operations are output as `shift_output` and `rotate_output`, respectively. The entire process is synchronized using the clock signal (`clk`).

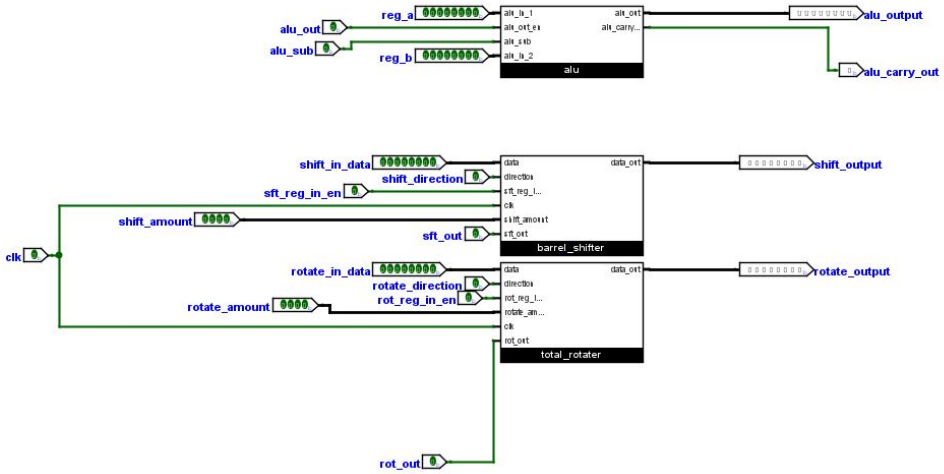

Figure 9: Schematic Diagram of ALU

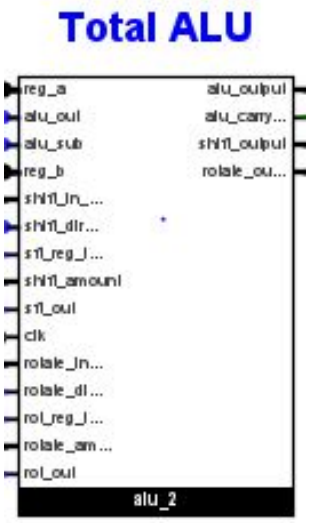

Figure 10: Block of Total ALU

Shift/Rotate amount register

The Shift/Rotate Amount Register is designed to store the shift and rotate amounts for operations such as bit shifting and rotation. The register receives input data through the `reg_in` signal, which is controlled by the `reg_in_en` signal, enabling the loading of the input value. The register operates based on the `reg_clk`, which synchronizes the data transfer, and the `reg_out_en` signal, which controls the output of the stored value. The output data, denoted as `reg_out`, represents the stored shift or rotation amount, which is then used by the shift and rotate units to perform the corresponding operations. This register is a key component for controlling how much to shift or rotate data during processing.

Figure 11: Block of shift/rotate amount register

Shifter

An 8-bit barrel shifter, which shifts the input data bits either left or right based on the value of shift_amount. It uses a series of multiplexers (MUXes) to control the bit positions dynamically. The design allows shifting by any number of positions in a single clock cycle, providing fast and efficient data manipulation.

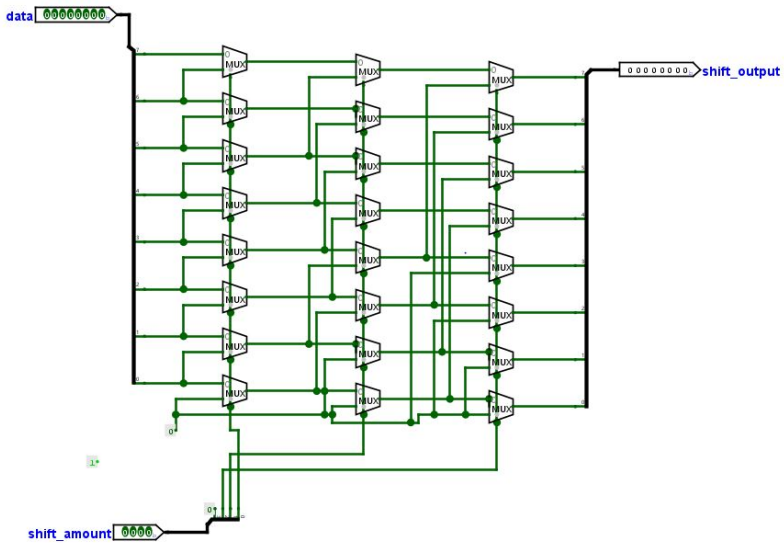

Figure 12: Schematic Diagram of the barrel shifter

To reverse the bits a schematic has been made shown in Figure 13. An 8-bit shifter

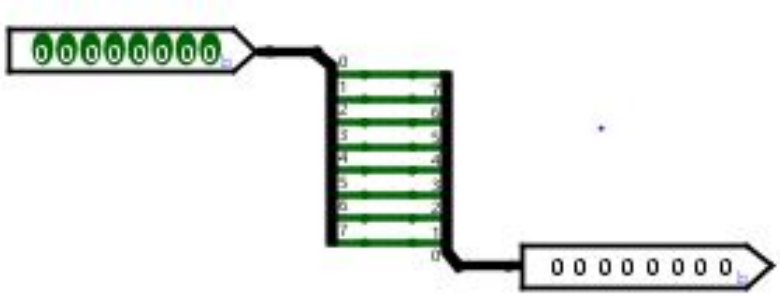

Figure 13: Schematic Diagram of the reverse bits

unit that shifts the input data either left or right based on the specified direction and shift_amount. It uses control signals and registers to perform the shift operation and outputs the shifted result through data_out.

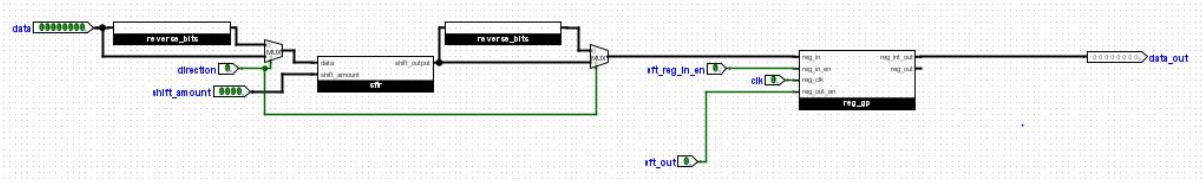

Figure 14: Schematic Diagram of the shifter

Rotater

An 8-bit barrel rotator unit that performs circular bit rotation on input data based on the specified rotate_amount. It uses a network of multiplexers (MUXes) to cyclically shift bits left or right without data loss. The rotated output is produced instantly through rot_output, allowing efficient bit manipulation.

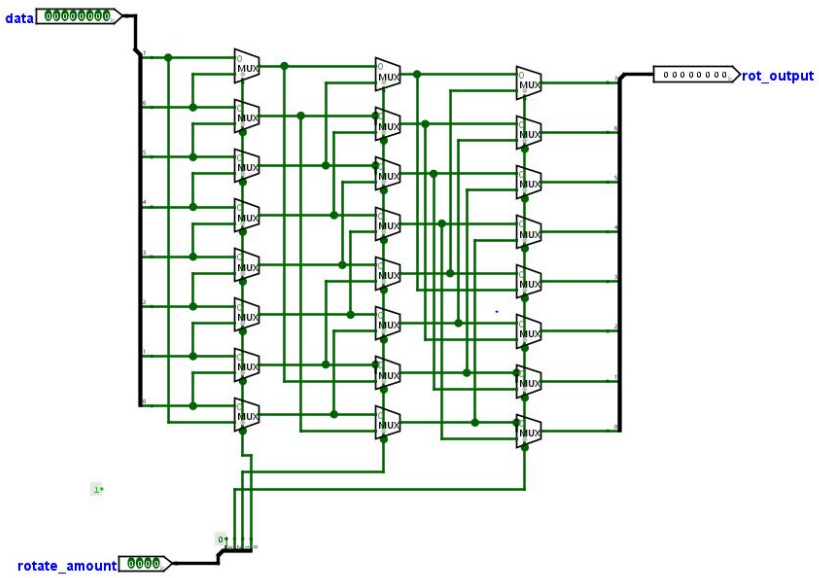

Figure 15: Schematic Diagram of the barrel rotater

To reverse the bits a schematic has been made shown in Figure 16. An 8-bit rotater

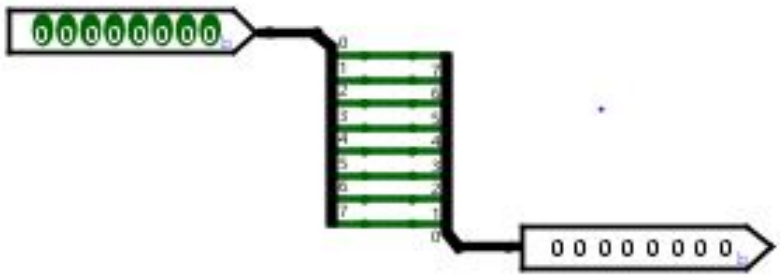

Figure 16: Schematic Diagram of the reverse bits

unit that rotates the input data either left or right based on the specified direction and rotate_amount. It uses control signals and registers to perform the rotate operation and outputs the rotated result through data_out.

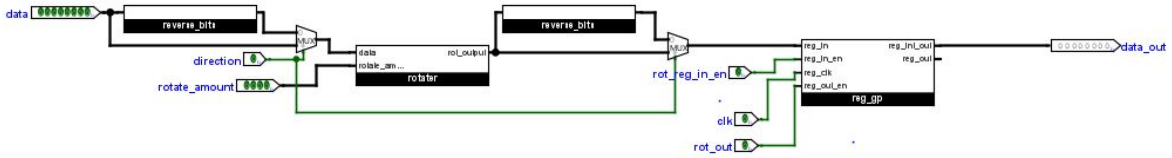

Figure 17: Schematic Diagram of the rotater

T states

T-state represents a discrete time period within the fetch-decode-execute cycle of a CPU, where specific operations are performed. The control logic uses these T-states to generate

control signals that coordinate the operation of components like memory, registers, and the ALU during each phase.

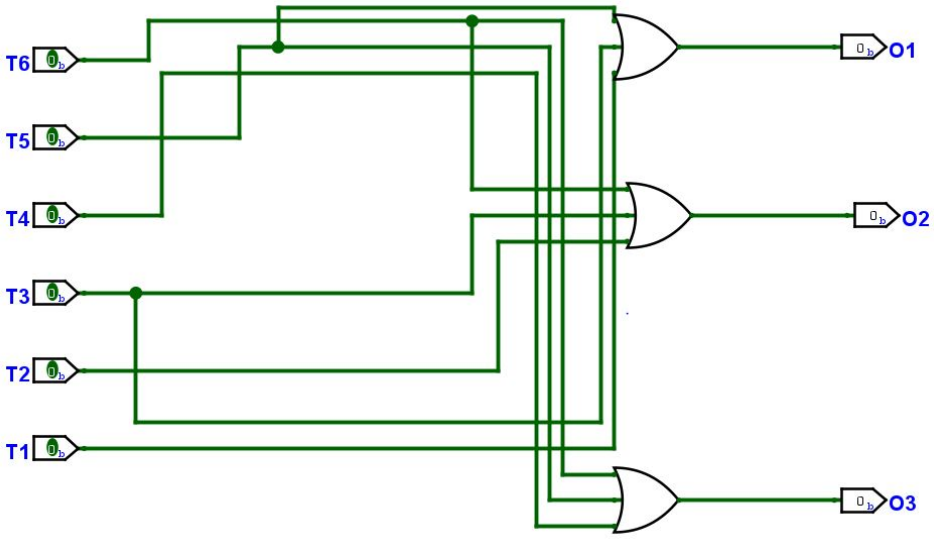

Figure 18: Schematic Diagram of T state

Control Sequencer

The Control Sequencer is a crucial component of the CPU's control unit, responsible for generating the necessary timing and control signals during each phase of the fetch-decode-execute cycle. It takes the clock signal and other inputs to synchronize the various operations within the CPU. The sequencer manages the execution of instructions by controlling signals that enable or disable components like memory, registers, ALU, and other modules. The sequencer ensures that each part of the CPU operates in the correct sequence by transitioning through different T-states (T1, T2, T3, etc.), which trigger specific control actions such as loading values into registers, performing arithmetic operations, or activating other components.

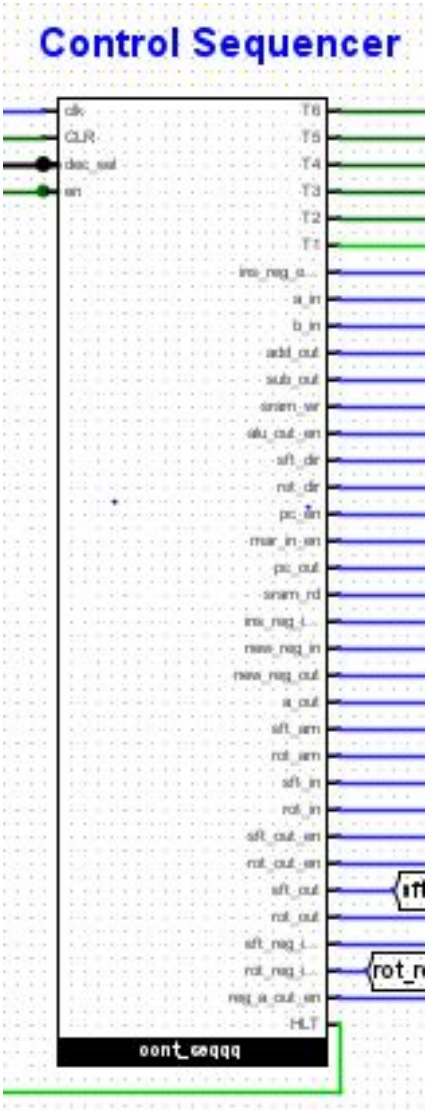

Figure 19: Block of Control Sequencer

Inner connections of control sequencer

This is responsible for generating the appropriate control signals during the CPU's operation. At the top, a Ring Counter tracks the different states in the fetch-decode-execute cycle. The Decoder component interprets the signals from the ring counter and generates corresponding control signals for actions like loading data into registers (load), performing arithmetic operations (add, sub, etc.), and executing operations such as HALT. The States section assigns the states for the different phases (T1, T2, T3, etc.) of the instruction cycle, and the logic gates (AND, OR) combine these signals to generate the final control outputs. These outputs are then used to manage the operation of various CPU components, ensuring that each part of the processor performs the correct function at the right time. This setup is essential for maintaining synchronization and controlling the flow of data during instruction execution.

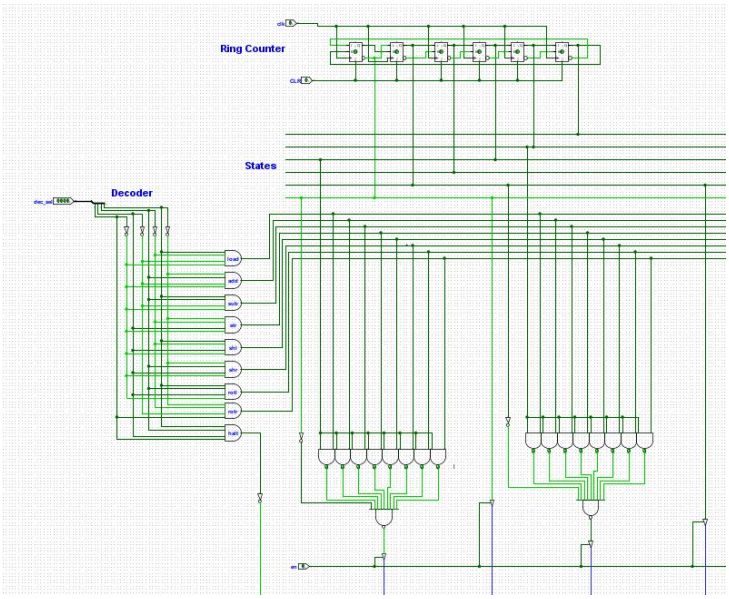

Figure 20: Inner connections of Control Sequencer-I

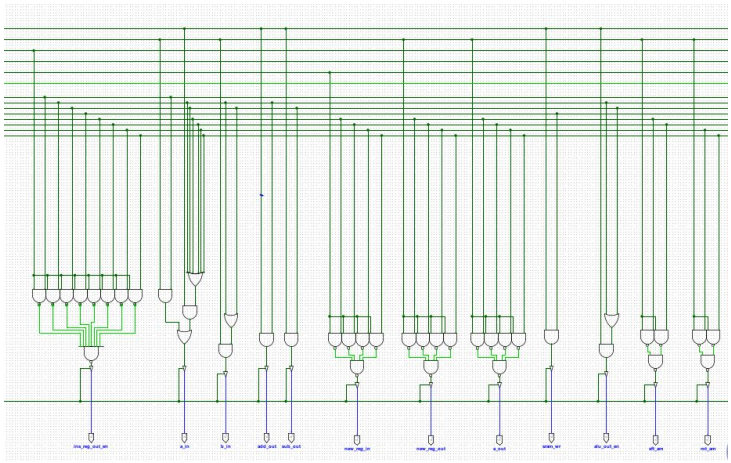

Figure 21: Inner connections of Control Sequencer-II

Bootloader/Data Loader

The Data Loader architecture is responsible for fetching data from memory and loading it into the CPU during execution. In the first image, the Data Loader interacts with the Address Register and ROM to retrieve data, with control signals like `data_load_en` and `debug` managing the flow of data. The `debug_data` signal allows for monitoring the data being loaded. In the second image, the Control Sequencer for Data Loader coordinates the loading process by controlling the Address Register, directing the address to fetch data, and managing memory operations using AND gates to synchronize the data flow. This sequencer ensures that data is loaded at the correct time during the CPU's cycle, facilitating smooth execution of instructions and operations.

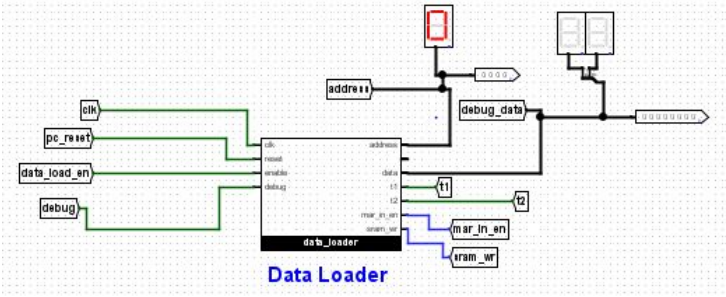

Figure 22: Block of Data loader

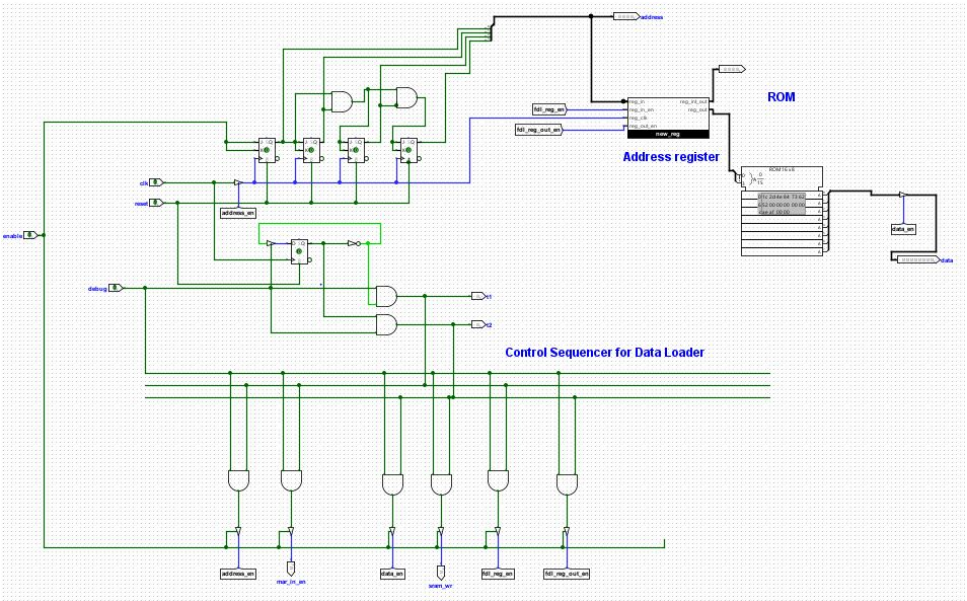

Figure 23: Inner connections of Data loader

Compiler Interface

A SAP-1 Compiler interface has been made that allows users to write assembly code for the SAP-1 CPU and generate the corresponding Hex code. The left side of the screen allows users to input their assembly code, which includes instructions such as LDA, ADD, STA, ROR, HLT, and more. The compiler then translates this assembly code into Hexadecimal machine code on the right, ready to be loaded into the 16-byte ROM of the SAP-1 CPU.

In the example provided, the Assembly Code includes various instructions, such as loading a value into the accumulator (LDA 13), performing an addition (ADD 14), storing a value (STA 15), rotating the accumulator (ROR 4), and halting the execution (HLT). The Generated Hex Code represents the compiled instructions in machine-readable format, which can be directly inserted into the ROM initialization of the Logisim simulation. The Opcode Map shows how each instruction is mapped to its corresponding opcode and operand. This tool helps in quickly generating the machine code for a given SAP-1 assembly program, streamlining the development and testing process.

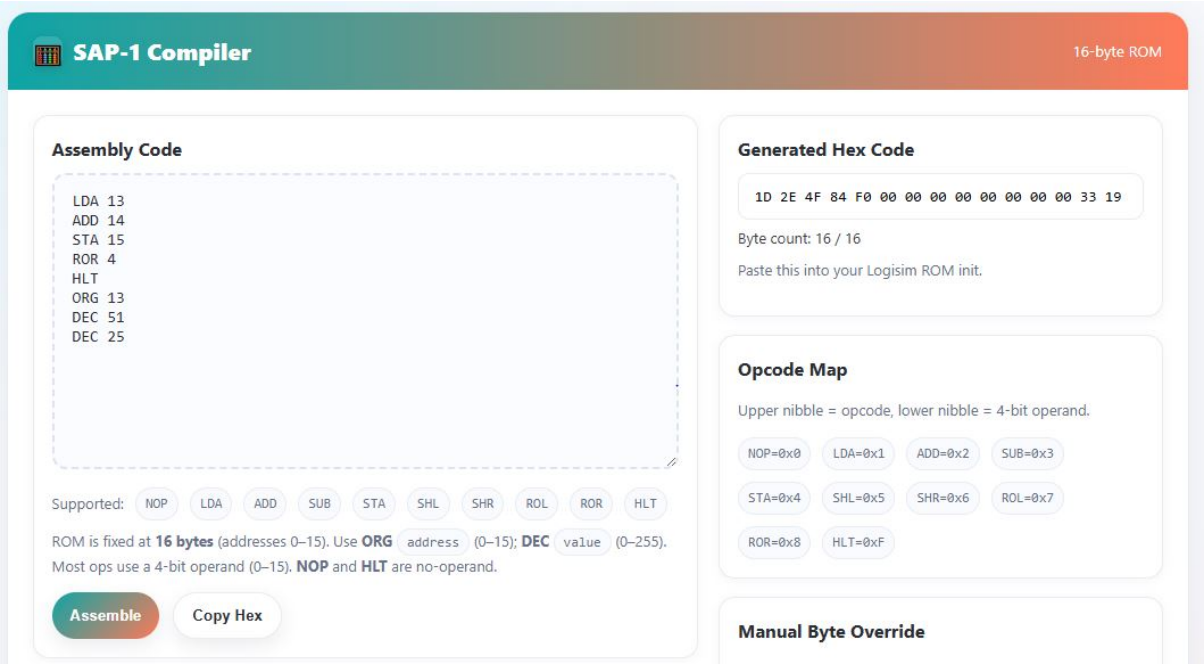

Figure 24: Figure of the compiler interface

Final Circuit

Figure 25 represents the final SAP-1 CPU Architecture.

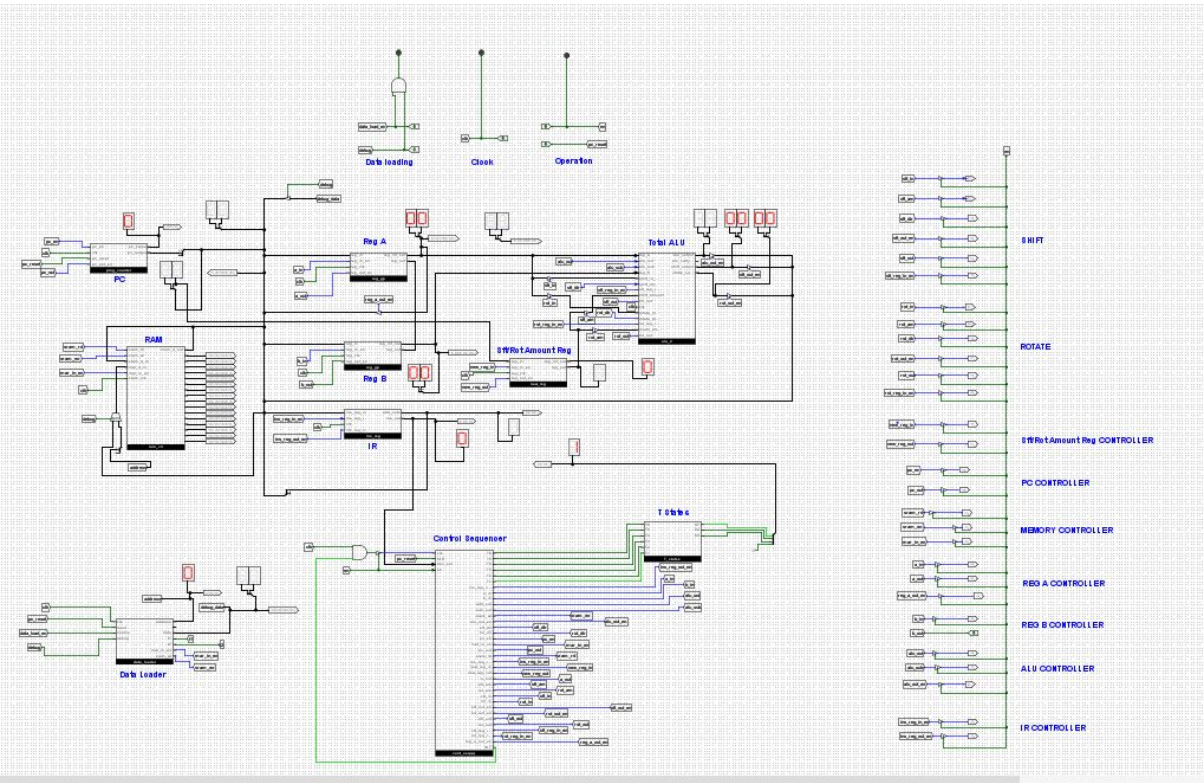

Figure 25: Final SAP-1 CPU Architecture

Control Signals

The Boolean equations provided determine the conditions under which each control pin is activated (set to HIGH). These conditions are directly realized through the use of AND and OR gates within the Control Matrix.

For Data Loading

1. **mar_in_en :** debug AND t1.
2. **address_en :** debug AND t1.
3. **fdl_reg_en :** debug AND t1.
4. **sram_wr :** debug AND t2.
5. **data_en :** debug AND t2.
6. **fdl_reg_out_en :** debug AND t2.

Connections of control sequencer

1. **mar_in_en :** T1 (NOT) NAND (T4 NAND Load) NAND (T4 NAND Add) NAND (T4 NAND Sub) NAND (T4 NAND Str) NAND (T4 NAND Shl) NAND (T4 NAND Shr) NAND (T4 NAND Rotr) NAND (T4 NAND Rotl).
2. **pc_out :** T1.
3. **sram_rd :** T2 (NOT) NAND (T5 NAND Load) NAND (T5 NAND Add) NAND (T5 NAND Sub) NAND (T5 NAND Str) NAND (T5 NAND Shl) NAND (T5 NAND Shr) NAND (T5 NAND Rotr) NAND (T5 NAND Rotl).
4. **ins_reg_in_en :** T2.
5. **pc_en :** (T3 NAND Load) NAND (T3 NAND Add) NAND (T3 NAND Sub) NAND (T3 NAND Str) NAND (T3 NAND Shl) NAND (T3 NAND Shr) NAND (T3 NAND Rotr) NAND (T3 NAND Rotl).
6. **ins_reg_out_en :** (T4 NAND Load) NAND (T4 NAND Add) NAND (T4 NAND Sub) NAND (T4 NAND Str) NAND (T4 NAND Shl) NAND (T4 NAND Shr) NAND (T4 NAND Rotr) NAND (T4 NAND Rotl).
7. **a_in :** (T5 AND Load) OR [T6 AND (Add OR Sub OR Shl OR Shr OR Rotr OR Rotl)] .
8. **b_in :** T5 AND (Add OR Sub).
9. **add_out :** T6 AND Add.
10. **sub_out :** T6 AND Sub.

11. **new_reg_in :** (T2 NAND Shl) NAND (T2 NAND Shr) NAND (T2 NAND Rotr) NAND (T2 NAND Rotl).
12. **new_reg_out :** (T5 NAND Shl) NAND (T5 NAND Shr) NAND (T5 NAND Rotr) NAND (T5 NAND Rotl).
13. **a_out :** (T5 NAND Shl) NAND (T5 NAND Shr) NAND (T5 NAND Rotr) NAND (T5 NAND Rotl).
14. **sram_wr :** T6 AND Str.
15. **alu_out_en :** T6 AND (Add OR Sub).
16. **a_out :** (T5 NAND Shl) NAND (T5 NAND Shr) NAND (T5 NAND Rotr) NAND (T5 NAND Rotl).
17. **sft_in_en :** T5 AND (Shl OR Shr).
18. **sft_in :** (T5 NAND Shl) NAND (T5 NAND Shr).
19. **sft_am :** (T5 NAND Shl) NAND (T5 NAND Shr)
20. **sft_dir :** (T5 AND Shl).
21. **sft_out_en :** T6 AND (Shl OR Shr).
22. **sft_out :** T6 AND (Shl OR Shr).
23. **rot_in_en :** T5 AND (Rotl OR Rotr).
24. **rot_in :** (T5 NAND Rotr) NAND (T5 NAND Rotl). .
25. **rot_am :** (T5 NAND Rotr) NAND (T5 NAND Rotl).
26. **rot_dir :** (T5 AND Rotl).
27. **rot_out_en :** T6 AND (Rotl OR Rotr).
28. **rot_out :** T6 AND (Rotl OR Rotr).
29. **reg_a_out_en :** T6 AND Str.
30. **HALT :** A NOT Gate has been connected with this.

Instruction Set & Program

Instruction (Binary)	HEX code	Description
0001 1100	1C	LDA 12 (Loads the value which is stored in address 12)
0010 1101	2D	ADD 13 (Adds with the value which is stored in address 13)
0011 1101	3D	SUB 13 (Subs with the value which is stored in address 13)
0100 1110	4E	STR 14 (Stores the result)
0101 0100	54	SHL 4 (Performs 4 bits left shift)
0110 0011	63	SHR 3 (Performs 3 bits right shift)
0111 0011	73	ROTL 3 (Performs 3 bits left rotate)
1000 0111	87	ROTR 7 (Performs 7 bits right rotate)
1111 0000	F0	HLT (Program Halts)

Step-by-Step Instruction Execution

Data Loading

To begin the data loading process, pc_reset should be toggled, then debug pin and debug_load_en pin should be high.
t1: At t1 state mar_in_en becomes high, selects the address where the data is needed to be stored.
t2: At t2 state sram_wr pin enables, stores the data at the selected address of RAM.

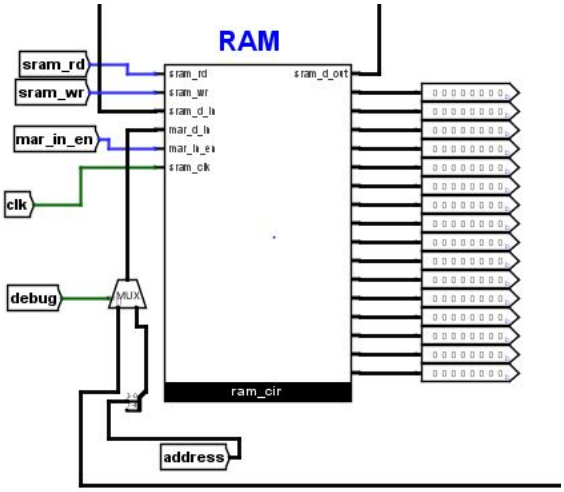

Figure 26: Status of RAM before loading data

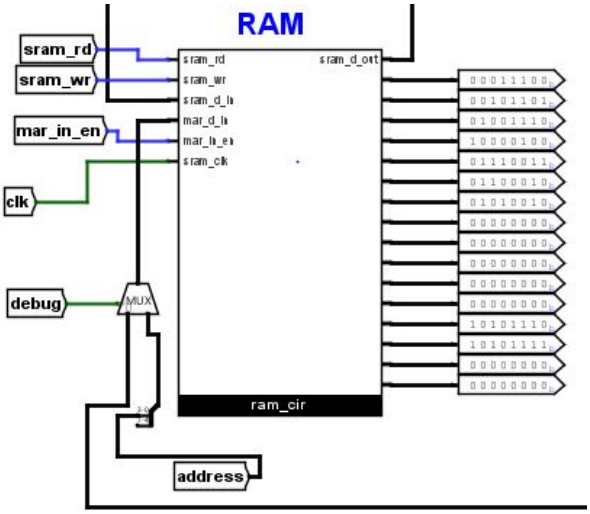

Figure 27: Status of RAM after loading data

Running the program :

When the machine codes have been stored successfully, the program counter (PC) resets to 0000 and the program starts to run. Basically, programs run in three steps. They are : Fetch, Decode and Execute.

i. Fetching :

Fetch stage has 3 T-states, all followed by a clock pulse.

T1 : `pc_out` and `mar_in_en` have been toggles, and a clock pulse has been given. The address of the next instruction to be executed has been sent to `MAR` from `PC`, and the control pins have been turned off.
T2 : `sram_rd` and `ins_reg_in_en` toggles.
T3 : `pc_en` toggles. This causes the counter's value to be incremented to 0001, indicating the location of the next line of code.

ii. Decoding :

Decode stage has no T-states and in pure combinational logic.

iii. Executing :

For example, the instruction is to load value at register A. As the loaded instruction is load A or LDA, it also has 3 T-states as follows:

T4 : `ins_reg_out_en` and `mar_in_en` toggles. This causes the address to be saved into the `MAR` to be fetched.
T5 : `sram_rd` and `a_in` toggles. As a result, the value stores in register A
T6 : This state is unused by LDA.

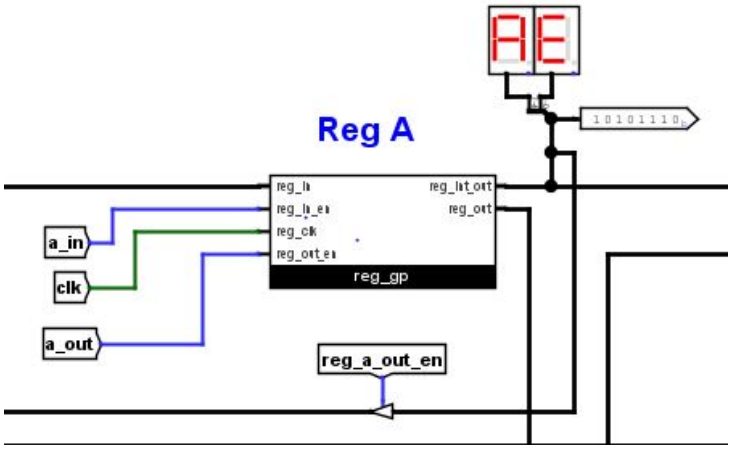

Figure 28: Observation of data load in register A

Testing and Validation

1. Testing the Data Loader to ensure that it correctly loads data from the ROM to RAM during startup and that the data is correctly written to memory for further use during execution.

2. Conducting individual tests on each CPU component, including the Program Counter (PC), ALU, Registers, Control Unit, and RAM, to ensure correct functionality and interaction between parts.

3. Verifying that each instruction in the SAP-1 CPU's instruction set (e.g., LDA, ADD, SUB, STA, SHL, SHR, ROTL, ROTR, HLT) behaves as expected by simulating multiple instruction cycles with known inputs and expected outputs.

4. Running complete programs on the SAP-1 CPU within a simulation environment to verify that the fetch-decode-execute cycle operates correctly and produces expected results.

Future Work and Improvements

1. **Expanding the Instruction Set** – Adding more instructions such as JMP, CALL, RET, and conditional branching to enhance the versatility of the CPU and support more complex programs.
2. **Implementing Pipelining** – Introducing pipelining to increase instruction throughput and reduce the number of cycles needed for executing multiple instructions.
3. **Supporting Stack Operations** – Enabling stack operations for handling function calls and returns, supporting recursive programming and more complex function handling.
4. **Adding Interrupt Handling** – Incorporating interrupt handling mechanisms to allow the CPU to respond to external events or higher-priority tasks, improving real-time processing capabilities.
5. **Expanding Memory** – Increasing the memory size beyond the current 16 bytes and introducing RAM/ROM segmentation to support more extensive applications and data storage.
6. **Enhancing the ALU for Advanced Arithmetic** – Integrating advanced arithmetic operations in the ALU (e.g., multiplication, division, bitwise operations) to handle more complex computational tasks.
7. **Optimizing the Control Unit** – Refining the control unit for better efficiency, reducing gate count, and possibly incorporating microprogramming for more flexible control signal generation.

Discussions

1. The SAP-1 CPU design project has been successful in demonstrating the core principles of computer architecture through the implementation of a simple yet functional processor using Logisim Evolution.

2. The development of key components such as the Program Counter (PC), Arithmetic Logic Unit (ALU), Registers (A, B), and Control Unit has been effective in illustrating how a CPU operates during the fetch-decode-execute cycle.

3. The design has been focused on showcasing the fundamental operations of a CPU, including instruction fetching, decoding, and execution, providing a clear and practical understanding of its functionality.

4. The addition of a ROM-based bootloader has been instrumental in automating the loading of machine code into memory, eliminating manual data entry and improving the programming process.

5. The Data Loader has been implemented to simplify data transfer between ROM and RAM, ensuring smooth and uninterrupted program execution.