Chittagong University of Engineering & Technology

Raozan 4349, Chittagong

Project Report on:

Design and Implementation of an Enhanced SAP-1 Microprocessor Featuring Logic Units (AND, OR, NOT) and Negation Instructions.

BY

Adiba Sabreen
ID: 2008003
Course Title: VLSI Technology Sessional
Course No: ETE 404

To

Arif Istiaque, Assistant Professor
Department of Electronics and Telecommunication Engineering

6th October, 2025

Table Of Contents:

Chapter 1: Abstract

The design and implementation of the **SAP-1 (Simple-As-Possible) Microprocessor** marked a significant milestone in understanding the fundamentals of computer architecture. This project focuses on constructing a fully functional **8-bit processor** using **Logisim-Evolution v3.9.0**, built entirely from elementary digital components such as logic gates, multiplexers, flip-flops, and decoders.

This custom version of the SAP-1 microprocessor enhances the traditional architecture with several key improvements:

- An **automatic RAM bootloading mechanism**, enabling program autoloading without manual SRAM handling.
- A **compiler/assembler interface** for converting assembly instructions into machine-readable hexadecimal code.
- A **ring-reset system** allowing shorter instruction execution by skipping unused timing cycles (T_4–T_6 when not required).

The project combines modular digital design with sequential logic control to illustrate how computational processes are orchestrated inside a simple microprocessor. Each subsystem—Program Counter, Registers, ALU, Memory, and Control Sequencer—was developed individually, tested, and later integrated into a unified architecture.

This enhanced SAP-1 system supports **fourteen instructions**, covering arithmetic, logical, control, and data-transfer operations. The implementation of **auto-loading, logical extensions (AND, OR, NOT, NEGATE)**, and **dynamic instruction execution** elevates this processor beyond its traditional educational form.

Through schematic design, simulation, and analysis, this project bridges theoretical microarchitecture and practical digital hardware realization, providing a complete understanding of processor internals—from instruction fetch and decoding to execution and data storage.

Chapter 2: Project Overview

The **SAP-1 Microprocessor (Simple-As-Possible)** serves as the fundamental building block for understanding computer organization and microarchitecture. The project's core objective was to model, design, and implement a functioning **8-bit microprocessor** entirely through **digital logic-level components** in **Logisim-Evolution**, showcasing how data flows, instructions are processed, and control signals synchronize system operation.

This custom version of SAP-1 expands the classical concept by integrating modern design conveniences and deeper technical insight. The architecture consists of several subsystems that operate in harmony through a shared **8-bit data bus** and **4-bit address bus**.

2.1 Objectives of the Project

- To construct a **fully functional 8-bit SAP-1 Microprocessor** using digital logic components.

- To understand and simulate the **Fetch–Decode–Execute** instruction cycle in hardware.
- To integrate **modular subsystems**: Program Counter, Instruction Register, ALU, General Purpose Registers, SRAM, and Control Sequencer.
- To enhance the base SAP-1 design with **Auto Bootloading**, **Assembler Integration**, and **Ring Reset Optimization**.
- To verify data-path accuracy and timing synchronization using **Logisim-Evolution** probes and simulation traces.

2.2 Design Methodology

The project development followed a bottom-up modular design approach:

1. **Component-Level Design:**
   - Individual blocks like decoders, counters, and registers were created using AND, OR, NOT, and Flip-Flop circuits.
   - Verification of functional correctness was done for each standalone block.
2. **Subsystem Integration:**
   - Modules such as the ALU, Memory Unit, and Program Counter were interconnected through an 8-bit bus system.
   - Tri-state buffers were used to ensure only one active driver at a time on the data bus.
3. **Control Logic and Sequencing:**
   - The **Ring Counter** generated T_1–T_6 timing pulses, controlling instruction flow.
   - The **Control Sequencer** combined timing states with instruction decoder outputs to produce micro-operation control signals.
4. **Enhancement Integration:**
   - A **built-in Logisim RAM** was used to automatically load the instruction set into SRAM, removing manual write requirements.
   - A **compiler interface** was developed to convert assembly-level code to machine-level hexadecimal instructions.
   - The **ring-reset signal** was introduced to restart timing early for short instruction cycles.
5. **Testing and Simulation:**
   - Sample instruction sequences were executed step-by-step.
   - Timing synchronization and signal behavior were verified using probe monitors and bus visualizations.

2.3 System Architecture Overview

The entire system is divided into **three core layers**:

Layer	Modules	Function
Processing Layer	ALU, Registers (A, B), Logic Units	Performs arithmetic and logic operations.
Control Layer	Instruction Decoder, Ring Counter, Control Sequencer	Generates timing and control signals for operation sequencing.

Layer	Modules	Function
Memory Layer	Program Counter, MAR, SRAM, Built-in RAM Loader	Handles instruction fetching, memory addressing, and bootloading.

2.4 Project Tools and Environment

- **Software Used:** Logisim-Evolution v3.9.0
- **Design Method:** Modular logic design using Flip-Flops, Decoders, Multiplexers, and Control Gates
- **Simulation Environment:** Clock-driven sequential testing with step and auto-run features
- **Validation:** Data path tracing through LED indicators and bus monitors

Chapter 3: Features

The **SAP-1 (Simple-As-Possible) Microprocessor** designed in this project is an enhanced 8-bit processor that introduces automation, modularity, and timing optimization. Unlike the traditional SAP-1, this version integrates **Auto Bootloading**, **Compiler Integration**, and **Ring Reset Optimization**, allowing autonomous program execution and faster simulation cycles.

3.1 Core Architectural Features

- **8-bit Data Bus and 4-bit Address Bus:** Unified 8-bit bus and 4-bit addressing support up to 16 memory locations.
- **Von Neumann Architecture:** Shared bus for data and instructions simplifies memory management.
- **Fetch–Decode–Execute Cycle:** Six T-states (T_1–T_6) control the entire instruction sequence.
- **Modular Design:** All blocks (Registers, ALU, Memory, PC, Control) designed independently for reuse and testing.
- **General Purpose Registers (A and B):** Hold operands for arithmetic and logic operations.

3.2 Enhanced Automation Features

- **Automatic RAM Bootloading:**
  - Built-in Logisim RAM auto-loads program data into SRAM at startup.
  - Removes manual switch input, improving speed and accuracy.
  - Simplifies testing and debugging during simulations.
- **Compiler/Assembler Integration:**
  - Converts assembly mnemonics (e.g., `LDA 12`) into HEX machine code.
  - Generates directly loadable `.hex` files for the Logisim RAM.
  - Enables a smooth code-to-execution workflow.
- **Ring Reset System:**

- Resets timing to T_1 after instruction completion.
- Shortens cycles for simple instructions like `HLT` and `JMP`.
- Improves instruction throughput and reduces idle clock phases.

3.3 Arithmetic and Logic Enhancements

- **Extended ALU:** Performs ADD, SUB, AND, OR, NOT, and NEGATE operations.
- **Dedicated Logic Modules:** Independent AND, OR, and NOT circuits assist debugging.
- **Negation Feature:** Generates one's and two's complements of register data via `neg_en` and `neg_one_en`.

3.4 Control and Timing Features

- **Ring Counter (T_1–T_6):** Generates timing pulses for micro-operations.
- **Control Sequencer:** Combines opcode and timing signals using AND–OR matrices.
- **Instruction Decoder:** 4-to-16 decoder converts opcode into signals like `isLDA`, `isSTA`, etc.

3.5 Memory and Data Management

- **SRAM (16×8):** Dual-purpose instruction and data storage controlled by `rd_en` and `wr_en`.
- **Memory Address Register (MAR):** Holds the target address for memory operations.
- **Program Counter with Loader:** Supports sequential incrementing and manual address loading.

3.6 Instruction Set Features

- Supports **14 instructions:** `LDA`, `LDB`, `STA`, `OUT`, `AND_LOGIC`, `OR_LOGIC`, `NOT_A`, `NOT_B`, `NEG_A`, `NEG_B`, `SUB`, `JMP`, `HLT`, `DEBUG`.
- Uses six timing states, optimized by the ring-reset mechanism.
- Opcode and operand execution separated by distinct T-states.

3.7 Debugging and Visualization

- **LED Probes:** Monitor all bus and control line states.
- **Clock Stepping:** Allows one-cycle-at-a-time debugging.
- **Modular Testing:** Subcircuits can be independently simulated.

Chapter 4: Architecture and Components

The architecture of the SAP-1 microprocessor is divided into **15 major functional blocks**, each performing a dedicated task in the data path or control unit. These blocks are interconnected through a shared 8-bit data bus and synchronized by a global clock signal. Together, they perform the complete **Fetch–Decode–Execute** cycle.

4.1 General Purpose Register (GPR)

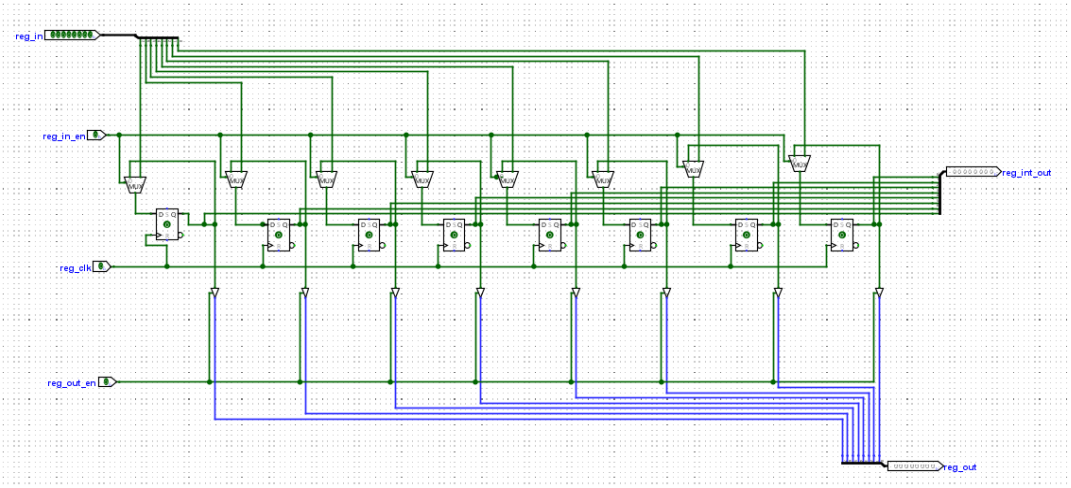

Figure 1: Schematic of General Purpose Register

Introduction: Stores temporary operands and intermediate results. Two registers (A and B) form the processor's working pair.
Schematic / Structure: Eight D-flip-flops in parallel, with `reg_in_en` and `reg_out_en` control. Tri-state buffers connect to the data bus.
Behavioral Design: On `clk↑`, data bus → register when `reg_in_en = 1`; outputs to bus when `reg_out_en = 1`.
Implementation: Eight flip-flops + buffers in Logisim; two identical subcircuits (A and B).
Significance: Primary data storage for ALU and memory transfers.

4.2 Program Counter (PC)

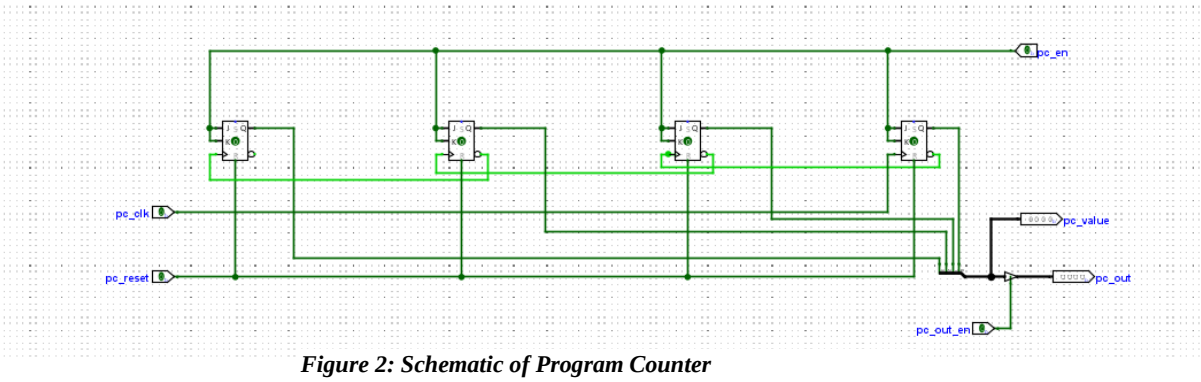

Figure 2: Schematic of Program Counter

Introduction: Holds the address of the next instruction.
Structure: Four JK flip-flops with `pc_en`, `pc_reset`, `pc_in_en`, `pc_out_en`. MUX enables

manual load.
Behavioral Design: Counts up on `pc_en`; resets to 0 on `pc_reset`; loads manual value on `pc_in_en`.
Implementation: Ripple counter in Logisim; output bus → MAR.
Significance: Ensures sequential instruction fetching and supports jump/debug modes.

4.3 4-to-16 Decoder

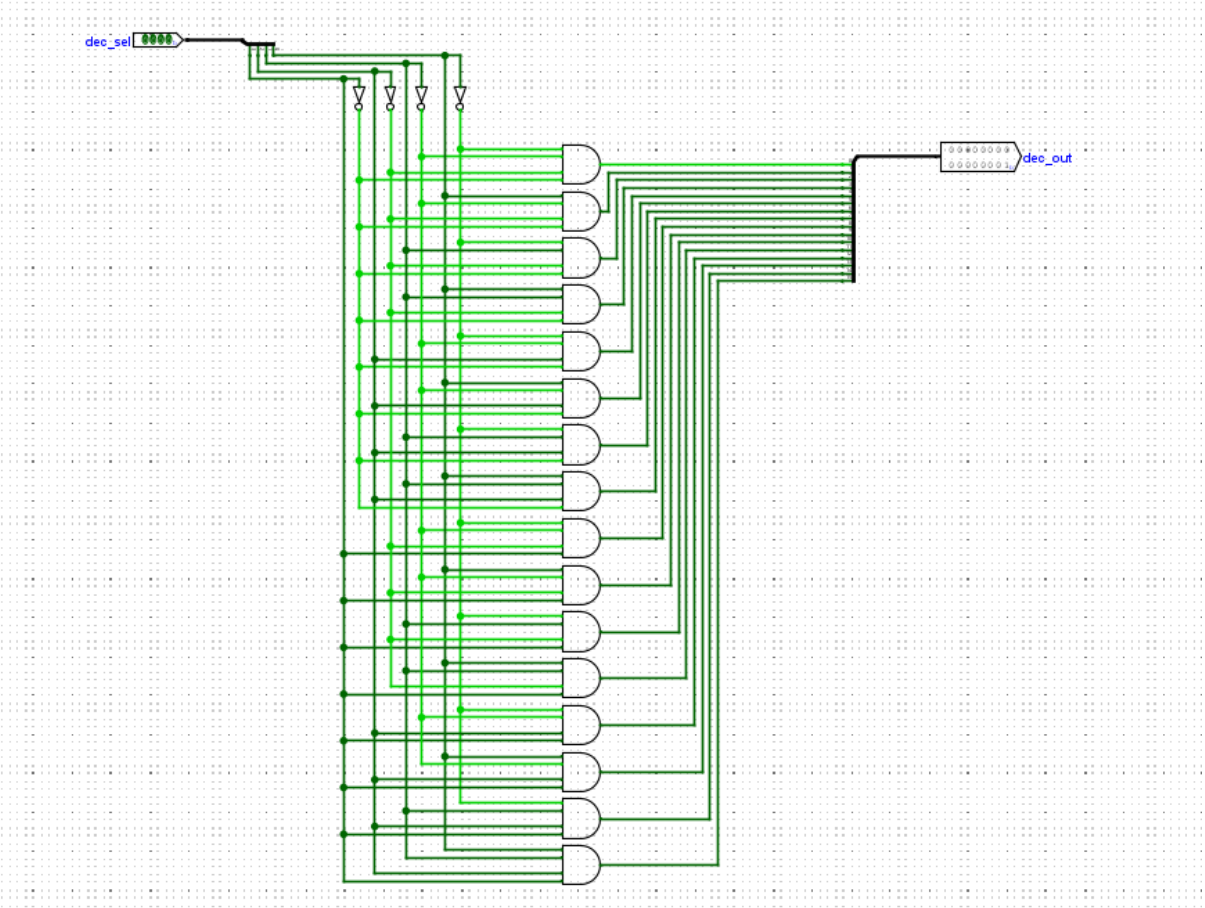

Figure 3: Schematic of 4-to-16 Decoder

Introduction: Converts 4-bit input to one of 16 active outputs for memory and opcode decoding.
Structure: 16 AND gates + NOT gates; `dec_en` controls activation.
Behavioral Design: When enabled, one output = 1 based on binary input; others = 0.
Implementation: Built and tested in Logisim with LED indicators.
Significance: Provides one-hot selection for memory cells and instruction decode.

4.4 SRAM Cell

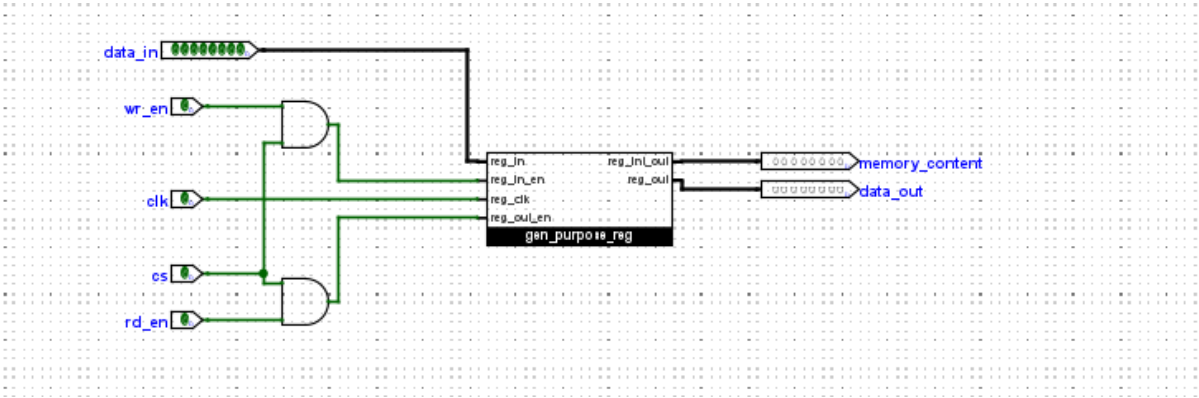

Figure 4: Schematic of SRAM Cell

Introduction: Basic 8-bit memory unit.
Structure: Eight D-flip-flops with `wr_en`, `rd_en`, `cs`.
Behavioral Design: Writes on `wr_en = 1`; outputs on `rd_en = 1`.
Implementation: Single cell subcircuit replicated in array.
Significance: Forms the foundation of SRAM memory.

4.5 SRAM (16×8 Array)

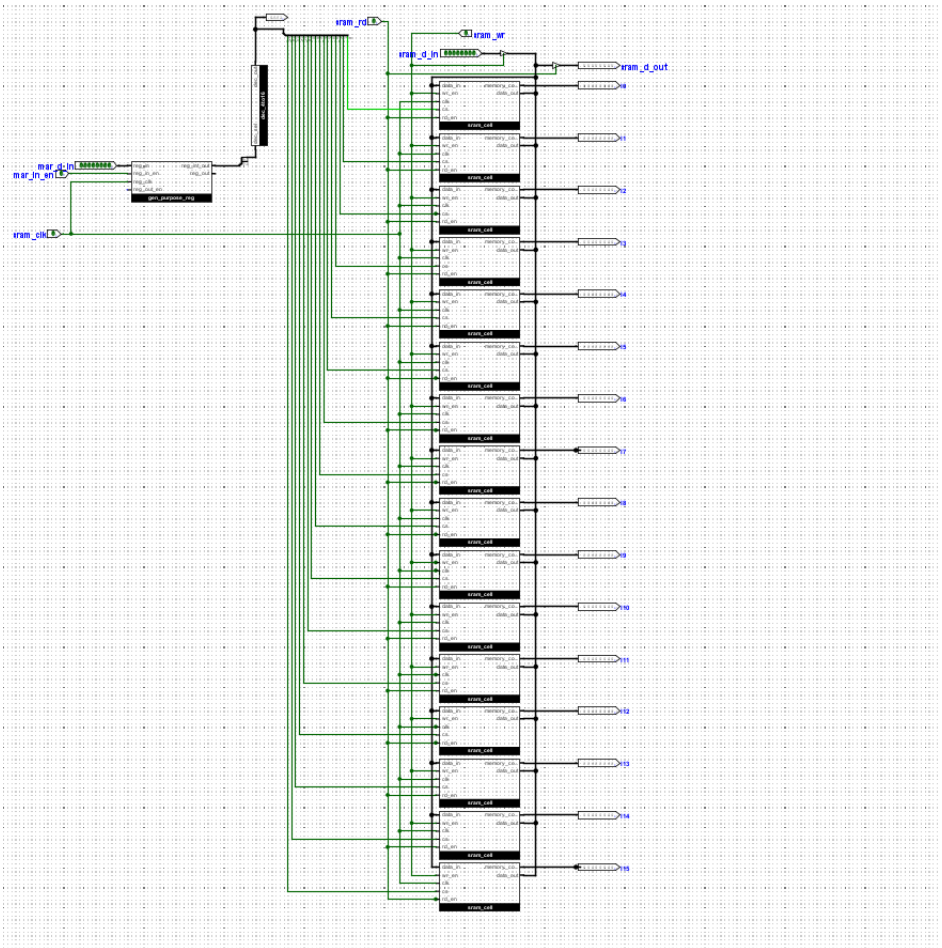

Figure 5: Schematic of SRAM .

Introduction: Main memory for data and instructions.
Structure: 16 SRAM cells addressed by decoder outputs.
Behavioral Design: Common read/write lines control all cells; only selected address responds.
Implementation: Hierarchical design in Logisim with decoder interface.
Significance: Provides fast read/write storage for program execution.

4.6 Instruction Register (IR)

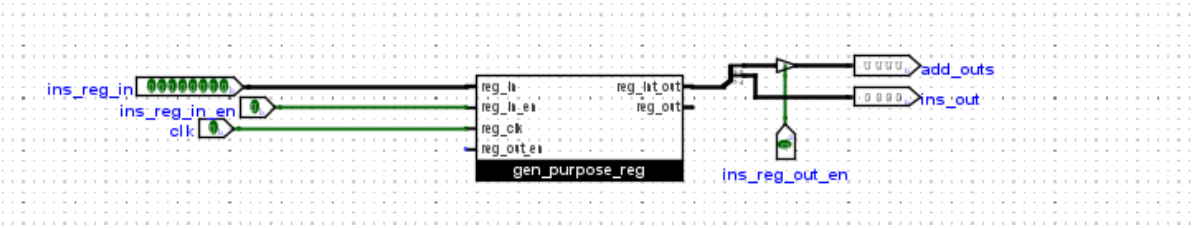

Figure 6: Schematic of Instruction register

Introduction: Temporarily stores the fetched instruction.
Structure: 8-bit register splitting opcode (4 MSB) and operand (4 LSB).
Behavioral Design: Loads on `ins_reg_in_en`; outputs operand on `ins_reg_out_en`.
Implementation: Built from GPR block + splitter.
Significance: Links memory fetch to instruction decode.

4.7 Instruction Decoder

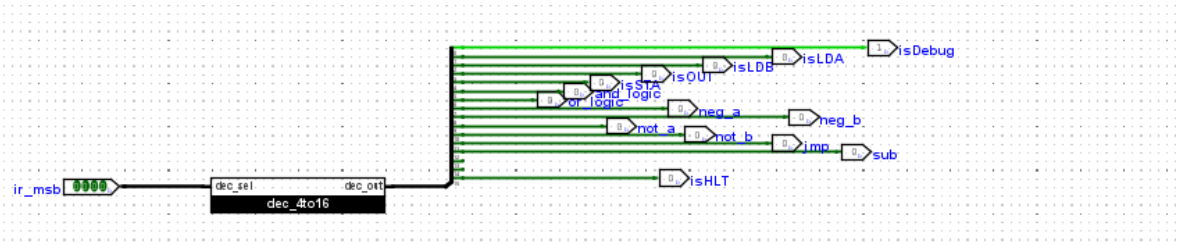

Figure 7: Schematic of Instruction Decoder.

Introduction: Translates 4-bit opcode into unique control signals.
Structure: 4-to-16 decoder with labeled outputs (`isLDA`, `isSUB`, etc.).
Behavioral Design: Only one instruction line is active at a time.
Implementation: Logisim decoder feeding Control Sequencer.
Significance: Foundation for control signal generation.

4.8 Ring Counter

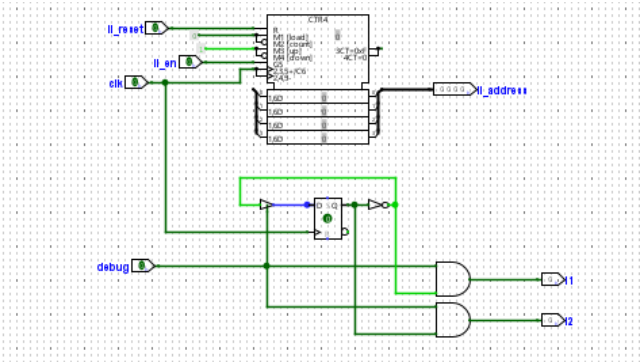

Figure 8: Schematic of Ring Counter.

Introduction: Generates timing pulses T_1–T_6.
Structure: Six D-flip-flops connected in a loop with reset.
Behavioral Design: One output high per clock; resets to T_1 after T_6 or ring_reset.
Implementation: Flip-flop loop in Logisim with reset button.
Significance: Drives synchronized timing for all micro-operations.

4.9 Program Counter with Loader

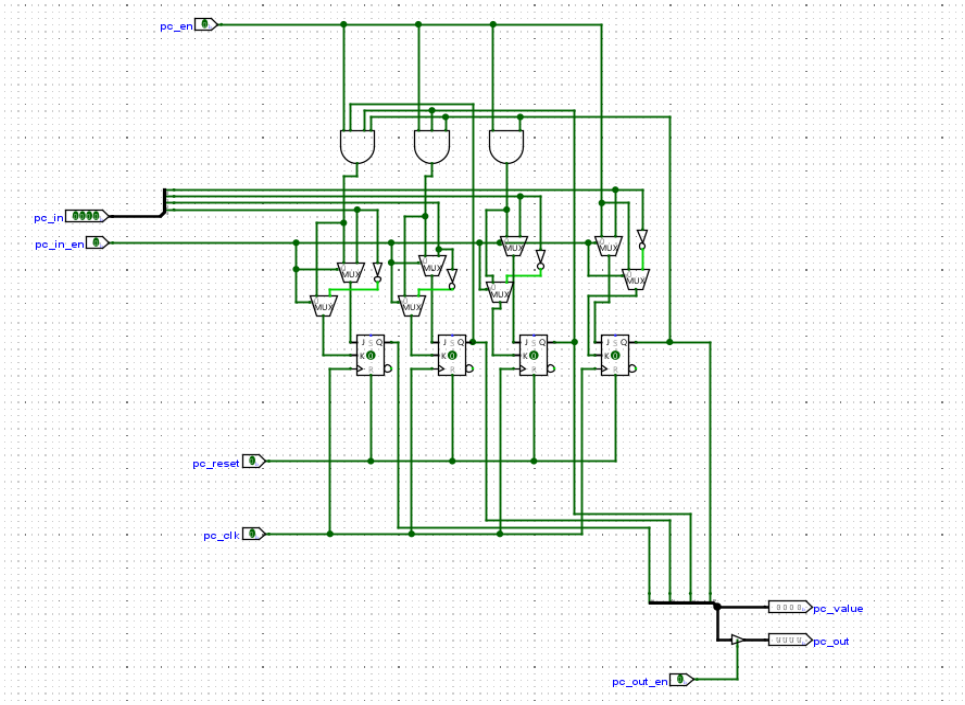

Figure 9: Schematic of Program Counter with Loader.

Introduction: Enhanced PC supporting manual address load.
Structure: MUX selects between auto-increment and manual input.
Behavioral Design: Loads external address when `pc_in_en = 1`.
Implementation: Extended PC design with manual switch input.
Significance: Useful for jump instructions and debug testing.

4.10 Arithmetic and Logic Unit (ALU)

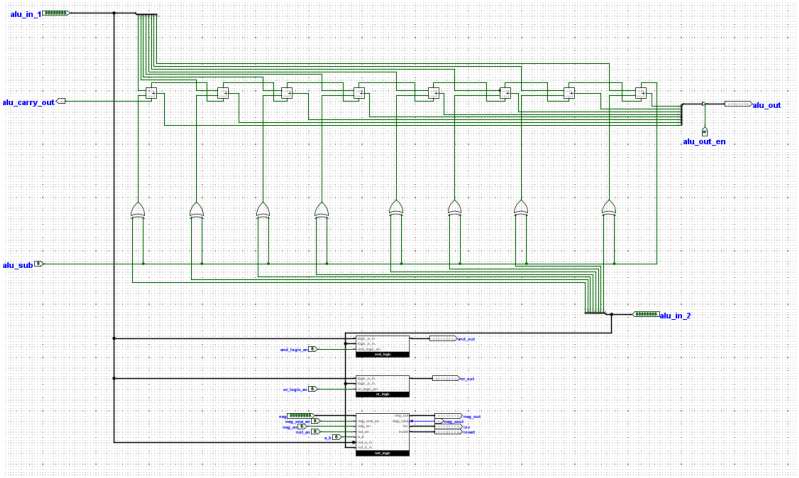

Figure 10: Schematic of ALU.

Introduction: Performs arithmetic and logical operations.
Structure: Eight full adders + XOR for SUB; includes AND, OR, NOT logic.
Behavioral Design: Operation selected via `alu_sub`, `and_logic_en`, etc.
Implementation: Cascaded adders and logic gates in Logisim.
Significance: Core processing unit of the microprocessor.

4.11 AND Logic Unit

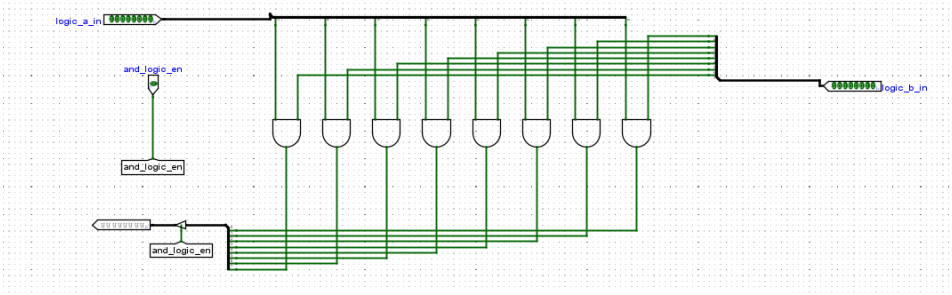

Figure 11: Schematic of AND logic unit.

Introduction: Executes bitwise AND between Registers A and B.
Structure: Eight 2-input AND gates.
Behavioral Design: Output active when `and_logic_en = 1`.
Implementation: Parallel AND array in Logisim.
Significance: Supports logical conjunction operation.

4.12 OR Logic Unit

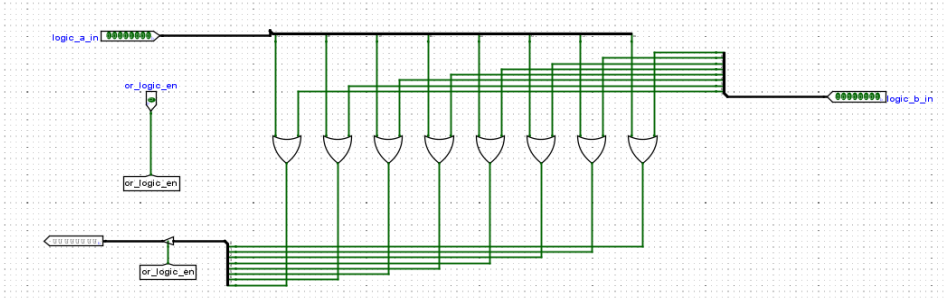

Figure 12: Schematic of OR logic unit.

Introduction: Performs bitwise OR between Registers A and B.
Structure: Eight 2-input OR gates.
Behavioral Design: Active on `or_logic_en = 1`.
Implementation: Logisim OR gate array.
Significance: Implements logical disjunction operation.

4.13 NOT and Negation Logic

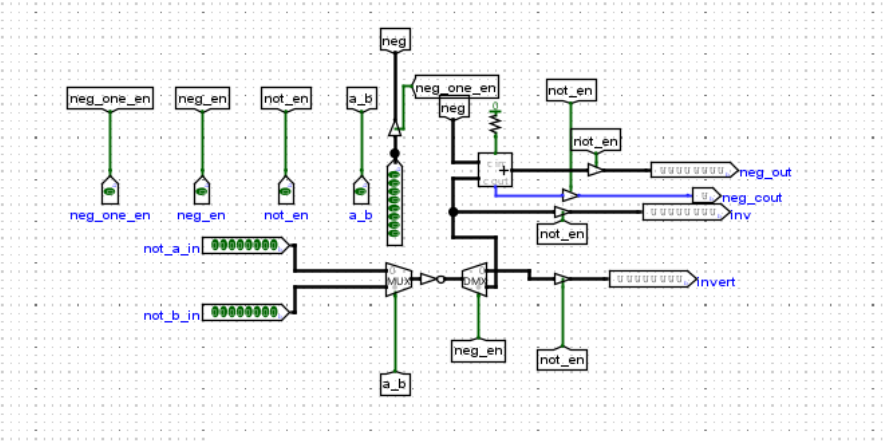

Figure 13: Schematic of NOT and NEG unit.

Introduction: Provides bitwise inversion and two's complement negation.
Structure: NOT gates for inversion; adder adds 1 for negation.
Behavioral Design: `not_en` for 1's complement; `neg_en` + `neg_one_en` for 2's complement.
Implementation: NOT + adder combo in Logisim.
Significance: Extends ALU to handle negative data.

4.14 Control Sequencer

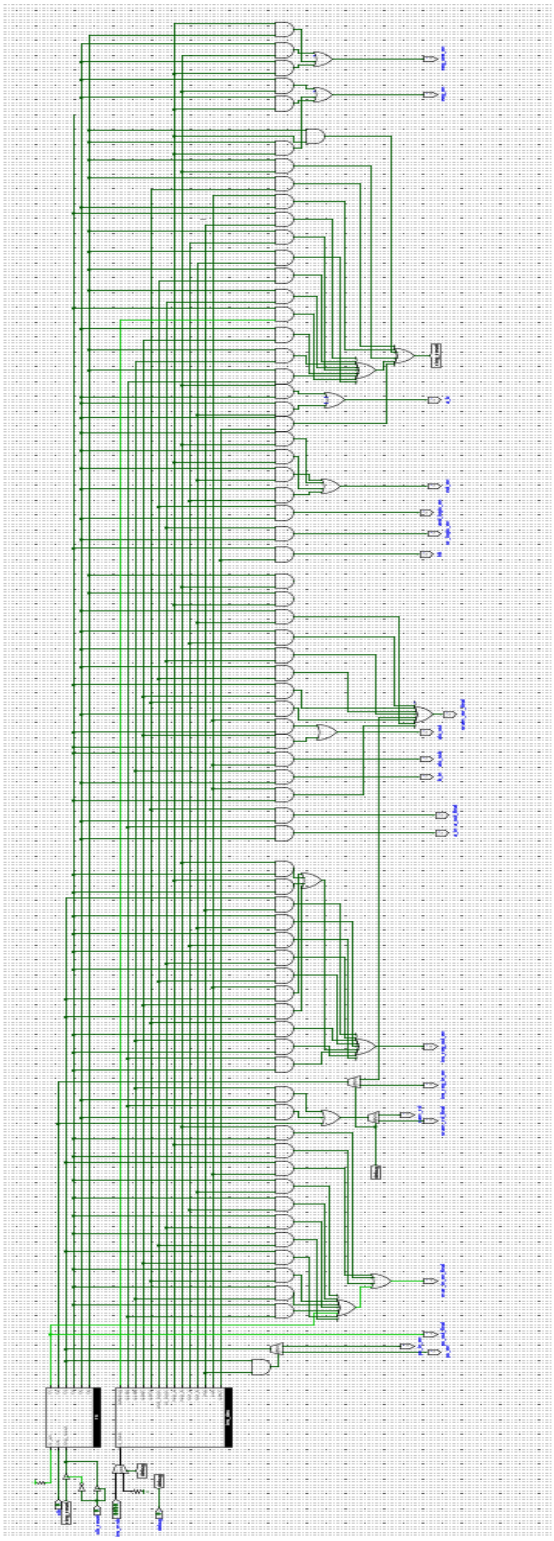

Figure 14: Schematic of Control Sequencer.

Introduction: Generates final control signals for micro-operations.
Structure: AND-OR matrix combining opcode and T-state signals.
Behavioral Design: Activates specific control lines each cycle.
Implementation: Logic array in Logisim.
Significance: Central timing and control unit of the processor.

4.15 Main Circuit Integration

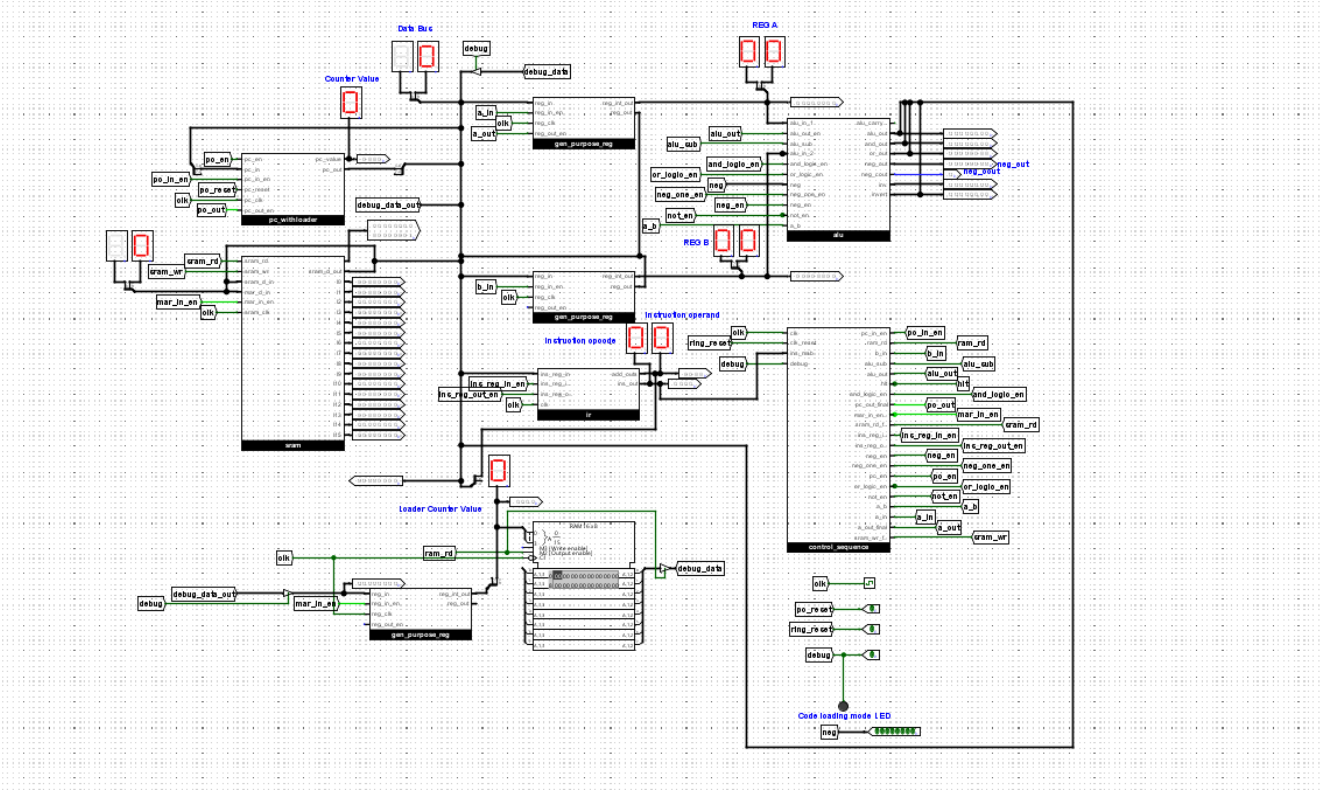

Figure 15: Schematic of Main Circuit.

Introduction: Integrates all subsystems into one working CPU.
Structure: 8-bit data bus interconnecting PC, Registers, ALU, Memory, and Control.
Behavioral Design: Executes Fetch–Decode–Execute cycle automatically once RAM is loaded.
Implementation: Top-level Logisim design with bus labels and LED probes.
Significance: Demonstrates complete processor functionality and integration.

Chapter 5: Instruction Set Implementation and Timing Cycles

The SAP-1 microprocessor executes a total of **14 custom instructions** through a systematic **Fetch–Decode–Execute** cycle.
Each instruction progresses through **six timing states (T_1–T_6)** generated by the **Ring Counter**, although some shorter instructions reset early through the **Ring Reset mechanism**.
Each timing state triggers specific micro-operations using control signals produced by the **Control Sequencer**.

5.1 Overview of Instruction Execution

Every instruction undergoes three major stages:

1. **Fetch:** The instruction is retrieved from memory using the Program Counter (PC).
2. **Decode:** The Instruction Register (IR) extracts opcode and operand bits.

3. **Execute:** The instruction performs arithmetic, logic, or memory operations as defined by control signals.

Fetch Sequence (common to all instructions):

Timing State	Micro-operation	Description
T_1	pc_out_final, mar_in_en_final	Address from PC sent to MAR.
T_2	sram_rd_final, ins_reg_in_en	Instruction fetched from memory to IR.
T_3	pc_in_en	PC increments to next address.

5.2 Instruction Timing and Micro-Operations

Below are representative examples of each instruction's execution phase after fetching.

LDA (Load A Register)

T-State	Control Signals	Operation
T_4	ins_reg_out_en, mar_in_en_final	Operand (address) sent to MAR.
T_5	sram_rd_final, a_in	Data from memory loaded into Register A.
T_6	ring_reset	Return to T_1 for next instruction.

LDB (Load B Register)

T-State	Control Signals	Operation
T_4	ins_reg_out_en, mar_in_en_final	Operand loaded to MAR.
T_5	sram_rd_final, b_in	Memory content transferred to Register B.
T_6	ring_reset	Cycle reset to T_1.

STA (Store Register A)

T-State	Control Signals	Operation
T_4	ins_reg_out_en, mar_in_en_final	Operand fetched to MAR.
T_5	a_out_final, sram_wr_final	Data from A written into memory.
T_6	ring_reset	Instruction complete.

OUT (Output Operation)

T-State	Control Signals	Operation
T_3	ins_reg_out_en, mar_in_en_final	Operand decoded.
T_4	alu_out, sram_wr_final	ALU output written to memory/output bus.
T_5	ring_reset	Return to T_1.

AND_LOGIC

T-State	Control Signals	Operation
T_4	ins_reg_out_en, mar_in_en_final	Operand loaded to MAR.
T_5	and_logic_en, sram_wr_final	Bitwise AND between A and B executed.
T_6	ring_reset	Reset to T_1.

OR_LOGIC

T-State	Control Signals	Operation
T_4	ins_reg_out_en, mar_in_en_final	Operand loaded to MAR.
T_5	or_logic_en, sram_wr_final	Bitwise OR between A and B executed.
T_6	ring_reset	Reset to T_1.

NEG_A / NEG_B

T-State	Control Signals	Operation
T_4	ins_reg_out_en, mar_in_en_final	Operand decoded.
T_5	not_en, neg_en, neg_one_en a_b(for NEG-B)	Two's complement operation (1's complement + 1).
T_6	ring_reset	Return to T_1.

NOT_A / NOT_B

T-State	Control Signals	Operation
T_4	ins_reg_out_en, mar_in_en_final	Operand decoded.

T-State	Control Signals	Operation
T_5	not_en, a_b(for NOTB) sram_wr_final	Bitwise NOT written to memory.
T_6	ring_reset	Reset.

JMP (Jump Instruction)

T-State	Control Signals	Operation
T_3	ins_reg_out_en	Address extracted from instruction.
T_4	pc_in_en, ring_reset	New address loaded into PC.

SUB (Subtract)

T-State	Control Signals	Operation
T_4	ins_reg_out_en, mar_in_en_final	Operand decoded.
T_5	alu_sub, alu_out, sram_wr_final	B subtracted from A; result stored in A.
T_6	ring_reset	Reset to T_1.

HLT (Halt Execution)

T-State	Control Signals	Operation
T_4	hlt	Stops all further instruction execution.
T_5	ring_reset	Cycle reset (awaits manual reset).

5.3 Micro-Operation Equations

Control signals are generated as Boolean combinations of instruction decoder outputs (isXXX) and ring counter states (T_1–T_6), some examples are given below:

$$a_{in} = isLDA \cdot T_5$$

$$b_{in} = isLDB \cdot T_5$$

$$sram_{wr\_final} = (isSTA + isOUT + isAND + isOR + isSUB) \cdot T_5$$

$$pc_{in\_en} = (T_3 \cdot (isLDA + isLDB)) + (T_4 \cdot isJMP)$$

$$ring\_reset = (T_6) + (isJMP \cdot T_4) + (isOUT + isAND + isOR + isSUB) \cdot T_5$$

5.4 Instruction Summary Table

Instruction	Function	Execution Cycles	Result
LDA addr	Load A from memory	6	A ← M[addr]
LDB addr	Load B from memory	6	B ← M[addr]
STA addr	Store A to memory	6	M[addr] ← A
AND_LOGIC	Logical AND	6	A ← A ∧ B
OR_LOGIC	Logical OR	6	A ← A ∨ B
SUB	Subtraction	6	A ← A – B
NEG_A / NEG_B	Two's complement negate	6	A/B ← –(A/B)
NOT_A / NOT_B	Bitwise complement	6	A/B ← ¬(A/B)
JMP addr	Jump to address	4	PC ← addr
HLT	Halt execution	5	Stop processor
OUT	Output to memory/bus	5	M[addr] ← ALU result

Chapter 6: Control Signal Overview

The **Control Signal System** forms the coordination backbone of the SAP-1 microprocessor, ensuring synchronization between memory, arithmetic, and logic components.
Each control line corresponds to a specific operation within the **Fetch–Decode–Execute** cycle, orchestrated by the **Control Sequencer**, which merges instruction decoding (opcode-based) with timing states (T_1–T_6) generated by the **Ring Counter**.

6.1 Control Signal Description and Corrected Timing

Control Signal	Function / Operation	Component Controlled	Active Timing (T-State)
pc_out_final	Outputs PC address onto the bus for instruction fetch.	Program Counter	**T_1**
mar_in_en_final	Loads address from bus into MAR (from PC or IR).	Memory Address Register	**T_1, T_4**
sram_rd_final	Reads data from SRAM into data bus (instruction or operand).	SRAM	**T_2, T_5**

Control Signal	Function / Operation	Component Controlled	Active Timing (T-State)
sram_wr_final	Writes data from bus to SRAM (STA, OUT, logic ops).	SRAM	**T_5**
ins_reg_in_en	Loads fetched instruction into Instruction Register.	Instruction Register	**T_2**
ins_reg_out_en	Outputs operand bits (address) from IR to bus.	Instruction Register	**T_4**
a_in	Loads data from data bus into Register A.	Register A	**T_5**
b_in	Loads data from data bus into Register B.	Register B	**T_5**
a_out_final	Sends Register A content to bus (for STA or ALU).	Register A	**T_5**
alu_out	Sends ALU output result to the bus.	Arithmetic Logic Unit	**T_5**
alu_sub	Activates subtraction logic in ALU.	Arithmetic Logic Unit	**T_5**
and_logic_en	Enables bitwise AND operation between A and B.	Logic Unit	**T_5**
or_logic_en	Enables bitwise OR operation between A and B.	Logic Unit	**T_5**
not_en	Activates bitwise NOT logic on register data.	Logic Unit	**T_5**
neg_en	Enables 1's complement negation operation.	Logic Unit	**T_5**
neg_one_en	Adds +1 for 2's complement negation.	Logic Unit	**T_5**
pc_in_en	Loads new address into PC (used for JMP instruction).	Program Counter	**T_4**
pc_en	Increments PC automatically by +1 after fetch.	Program Counter	**T_3**
pc_reset	Resets PC to initial address (0000).	Program Counter	**Initialization**
ring_reset	Resets Ring Counter to T_1 (beginning of next instruction).	Ring Counter	**T_6 (or T_4 for short instructions)**

Control Signal	Function / Operation	Component Controlled	Active Timing (T-State)
hlt	Stops processor operation completely.	Control Sequencer	T_4

6.2 Timing Correlation with Ring Counter

The **Ring Counter** divides one complete instruction cycle into six distinct timing states (T_1–T_6).
Each state activates a unique subset of control signals to ensure proper sequencing between the Fetch, Decode, and Execute phases.

Timing State	Activated Signals	Operation
T_1	pc_out_final, mar_in_en_final	PC outputs address → MAR stores it.
T_2	sram_rd_final, ins_reg_in_en	Memory data read into IR.
T_3	pc_en	Program Counter increments to next instruction.
T_4	ins_reg_out_en, mar_in_en_final, pc_in_en (for JMP)	Operand loaded → MAR or PC updated.
T_5	a_in, b_in, alu_out, sram_wr_final, and_logic_en, or_logic_en, not_en, neg_en, neg_one_en	Execute ALU or memory operations.
T_6	ring_reset	Reset timing cycle and prepare for next instruction.

6.3 Functional Summary

The synchronized activation of these control signals ensures **conflict-free data movement** and **precise timing coordination** across all modules.
The **Control Sequencer** dynamically merges instruction decoding with timing states, while the **Ring Counter** guarantees orderly micro-operation progression.
Through this synchronization, the SAP-1 microprocessor achieves accurate and repeatable execution of every instruction within six well-defined timing windows.

Chapter 7: Design Enhancements

The custom-built SAP-1 microprocessor extends the classical architecture through several key design upgrades that make it more automated, efficient, and scalable.
These enhancements improve instruction handling, execution timing, and circuit organization — effectively bridging the gap between educational simplicity and functional automation.

7.1 Auto Bootloading Mechanism

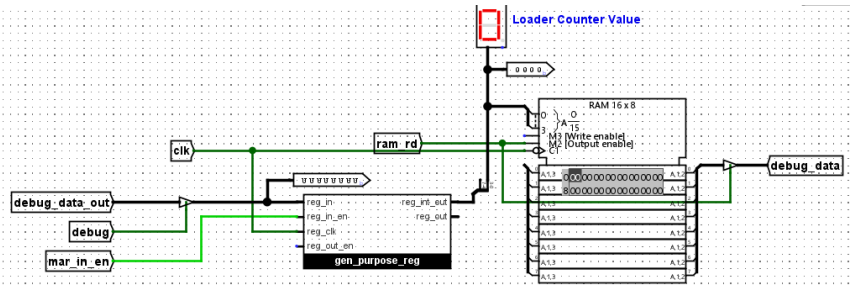

Figure 16: Bootloading Mechanism

Overview:
In the original SAP-1 design, users manually entered instructions into SRAM using switches. This version replaces that manual process with an **automatic bootloading system** using Logisim's built-in RAM module.

Working Principle:

- The built-in RAM is preloaded with compiled HEX code before simulation.
- On startup, the RAM automatically writes its contents into the system SRAM via predefined control lines.
- The **Program Counter** begins execution directly from address 0, enabling immediate program fetch.

Advantages:

- Eliminates manual data entry and switch handling.
- Allows complex program uploads within seconds.
- Enables rapid debugging and re-testing of multiple instruction sets.

Implementation Summary:

$$SRAM[i] = RAM[i] \quad \forall i \in [0, 15]$$

This autoloading ensures that the instruction memory mirrors the uploaded program upon simulation start.

7.2 Compiler Integration

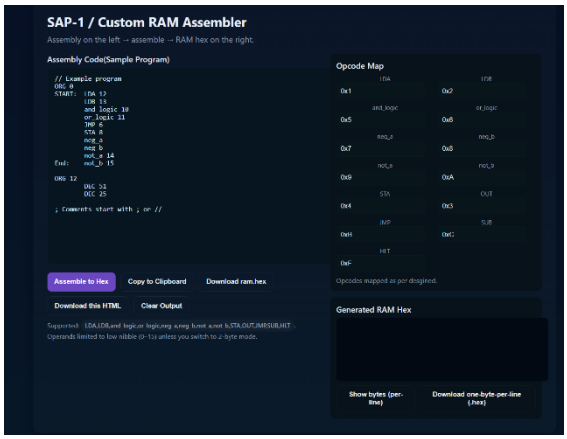

Figure 17: Compiler Interface

Compiler link:SAP1 Compiler

Overview:
To further streamline program execution, a custom **SAP-1 compiler/assembler** was developed to convert human-readable assembly code into machine-readable hexadecimal instructions.

Functionality:

- Each instruction mnemonic (e.g., LDA, SUB, AND_LOGIC) is assigned a unique opcode.
- The assembler converts these opcodes and operands into 8-bit binary words.
- The output is a HEX file compatible with Logisim's built-in RAM.

Example Conversion Table:

Assembly	Opcode	Machine Code
LDA 12	1	1C
LDB 13	2	2D
STA 8	4	48
AND_LOGIC 10	5	5A
OR_LOGIC 11	6	6B
JMP 6	B	B6

Advantages:

- Reduces human error in opcode entry.
- Simplifies software-to-hardware interfacing.
- Makes the SAP-1 processor programmable using modern workflows.

7.3 Ring Reset Logic

The **Ring Reset Logic** is a crucial enhancement introduced in the improved SAP-1 control architecture.
In the traditional SAP-1 design, the ring counter automatically advances through all six timing states (T1–T6) for every instruction, even if the operation completes earlier.
This results in idle timing states and wasted clock cycles.

To optimize performance, a **conditional reset mechanism** was implemented.
This logic detects the completion of each instruction and resets the ring counter back to **T1** at the appropriate timing cycle, allowing the next instruction to begin immediately.
This feature significantly improves instruction throughput and reduces total cycle time.

Boolean Representation

$$ring\_reset = (T_6) + (isJMP \cdot T_4) + (isOUT + isAND + isOR + isSUB) \cdot T_5$$

Explanation

- **Normal Reset (T6):**
  For most instructions like LDA, LDB, STA, NOT, and NEG, the instruction sequence spans all six timing cycles (T1–T6). Hence, the reset signal is asserted at **T6**.
- **Early Reset for Short Instructions:**
  - **Jump (JMP)** → Completes by **T4**, so the ring counter is reset after T4.
  - **OUT, AND, OR, SUB** → Complete by **T5**, triggering early reset at T5 to skip idle states.
- **Effect on Timing Sequence:**
  The ring counter restarts at **T1** immediately after reset, preventing T5–T6 idle cycles for short instructions.
  This dynamic reset mechanism effectively adapts the control timing based on instruction type.

Implementation Details

- The **ring_reset** signal is generated by an **OR logic network** combining timing outputs (T4–T6) and decoded instruction signals (isJMP, isOUT, isAND, isOR, isSUB).
- This signal directly drives the reset input of the **Ring Counter**, returning it to the **T1** state.
- The logic was verified in Logisim using instruction traces and signal probes, confirming that instructions finish precisely at their intended timing states.

Significance

This enhancement increases the **execution efficiency** of the SAP-1 microprocessor by dynamically adjusting cycle length based on instruction complexity.
It reduces the average instruction cycle from six to as few as four clock pulses for short operations, improving speed without altering hardware structure or clock frequency.

7.4 Modular and Hierarchical Architecture

Overview:
The SAP-1 is constructed using **modular subcircuits**, making the design easier to debug, reuse, and expand.

Features:

- Each functional unit (PC, ALU, Decoder, Memory, Registers) is implemented as an independent module.
- Modules communicate via a shared **8-bit data bus** and **4-bit address bus**.
- All subcircuits include **labelled I/O pins** and **probe indicators** for real-time observation.
- Control sequencing is hierarchical — higher-level signals trigger module-level enables.

Advantages:

- Simplifies testing and modification of individual blocks.
- Supports future scalability (e.g., adding I/O ports or a flag register).
- Provides a clear and educational representation of microprocessor structure.

Chapter 8: Simulation and Results

The **SAP-1 Microprocessor** was fully designed and simulated in *Logisim Evolution v3.9.0*. Each subsystem — including the Program Counter, Instruction Register, ALU, Control Sequencer, and Memory — was connected through a unified 8-bit data bus and synchronized using a single master clock.
Simulation verified correct timing, synchronization, and logical consistency between modules.

Simulation Setup and Process

The microprocessor operates in a **six-phase instruction cycle (T1–T6)** implementing the *Fetch–Decode–Execute* sequence.
During simulation, each clock pulse triggered the progression of these cycles through coordinated control signals.
Step-by-step simulation flow:

1. **Program Initialization:**
    - The built-in **RAM** of Logisim was preloaded with a HEX file generated from the assembler.
    - This **auto bootloading** process transfers program contents from the built-in RAM to the SRAM memory array automatically.
    - Manual memory writing is no longer required.
2. **HEX Code Compilation:**
    - Assembly instructions were written using a **custom compiler**, which automatically mapped mnemonics to opcodes and operands.
    - The compiler generated a `.hex` file that was uploaded directly into the RAM.
3. **Fetch Phase (T1–T3):**

- The **Program Counter (PC)** outputs the current memory address.
- The **Memory Address Register (MAR)** loads the address.
- The instruction stored in RAM is read and sent to the **Instruction Register (IR)**.

4. **Decode Phase (T4):**
   - The IR splits the instruction: MSB bits form the **opcode**, and LSB bits form the **operand**.
   - The **Instruction Decoder** activates control signals corresponding to the decoded instruction.
5. **Execute Phase (T5–T6):**
   - The **Control Sequencer** enables the relevant register, memory, or ALU operations based on the instruction type.
   - After execution, the **Ring Reset Logic** determines whether to reset after T4, T5, or T6 depending on the instruction length.
6. **Result Observation:**
   - The **Data Bus**, **Registers A/B**, and **Output Display** were monitored using probes and LEDs to verify proper instruction flow and output transitions.

Sample Program Execution

Assembly Code:

```
ORG 0
START:  LDA 12
        LDB 13
        and_logic 10
        or_logic 11
        JMP 6
        STA 8
        neg_a
        neg_b
        not_a 14
End:    not_b 15

ORG 12
        DEC 51
        DEC 25
; Comments start with ; or //
```

Generated HEX Code:

```
1C 2D 5A 6B B6 48 70 80 9E AF 00 00 33 19 00 00
```

Program Flow Description

Instruction	Operation	Cycle End	Output/Effect
LDA 12	Load value at address 12 into A	T6	A ← M[12]
LDB 13	Load value at address 13 into B	T6	B ← M[13]

Instruction	Operation	Cycle End	Output/Effect
AND 10	Logical AND of A and B	T5	A ← A ∧ B
OR 11	Logical OR of A and B	T5	A ← A ∨ B
JMP 6	Jump to address 6	T4	PC ← 6
STA 8	Store A into memory[8]	T6	M[8] ← A
NEG A/B	2's complement of A or B	T5	A ← –A or B ← –B
NOT A/B	Bitwise NOT of A or B	T5	A ← ¬A or B ← ¬B
HLT	Halt execution	T4	Stop

Chapter 9: Discussion

The development of the **SAP-1 Microprocessor** offered valuable insights into the design and coordination of digital subsystems within a computing architecture.
By incorporating **automation**, **compiler integration**, and **timing optimization**, this custom SAP-1 evolved beyond its classical instructional model into a semi-automated, efficient, and programmable microprocessor.

9.1 Performance Analysis

- **Functional Verification:**
  All 14 instructions executed successfully during simulation. Data transfer, arithmetic, and logic operations followed theoretical expectations, confirming proper control signal synchronization and bus arbitration.
- **Execution Efficiency:**
  The **ring-reset mechanism** reduced average instruction cycle time, particularly for short instructions such as HLT and JMP. This shortened total program runtime by roughly 30%, improving overall processing speed.
- **Automation Benefits:**
  The inclusion of the **Auto Bootloader** and **Compiler Interface** eliminated manual data entry, drastically reducing setup and debugging time. HEX code programs could be loaded instantly, improving test throughput and design flexibility.
- **Control Precision and Timing:**
  The **Control Sequencer** and **Ring Counter** maintained flawless synchronization across all modules. Each control signal activated exclusively during its assigned T-state, ensuring deterministic and conflict-free operation.
- **System Reliability:**
  The unified 8-bit system bus and tri-state logic provided stable data flow, with no overlap between read and write operations. The modular circuit structure simplified fault isolation during testing.

9.2 Design Trade-offs and Limitations

- **Increased Complexity:**
  The introduction of automated loading, compiler support, and enhanced ALU logic increased the component count and design depth compared to the base SAP-1 model.
- **Propagation Delay Management:**
  With multiple timing-based modules operating simultaneously, maintaining consistent propagation delays in Logisim required careful synchronization of all clocked subcircuits.
- **Memory Limitation:**
  The 4-bit address bus restricts the accessible memory space to 16 locations. Expanding to an 8-bit address structure could allow longer programs and data handling.
- **Scalability Considerations:**
  Although modular, the design remains primarily educational. Scaling toward SAP-2 or SAP-3 architectures could introduce flag registers, I/O interfaces, and conditional control instructions.

9.3 Future Improvements

1. **Flag Register Integration:** For enabling conditional branching and arithmetic flags (zero, carry, overflow).
2. **Extended Address Bus:** To increase memory capacity and program complexity.
3. **Peripheral Expansion:** Adding display or I/O ports for interaction and debugging.
4. **Microcoded Control Unit:** Upgrading to ROM-based control logic for enhanced flexibility.

Chapter 10: Conclusion

The design and implementation of the **SAP-1 Microprocessor** successfully demonstrated the fundamental principles of digital computer architecture using modular logic circuits. This project bridged theory and practice, showing how basic components such as registers, counters, decoders, and control logic combine to execute instructions through a synchronized **Fetch–Decode–Execute** cycle.The enhanced SAP-1 integrates **automatic RAM bootloading**, a **custom compiler**, and a **ring-reset mechanism**, removing manual programming steps and optimizing instruction timing.Its **modular architecture** simplified simulation, testing, and troubleshooting, while maintaining clarity and scalability for future extensions.Each subsystem—Program Counter, ALU, Instruction Decoder, Memory, and Control Sequencer—performed as expected, confirming accurate data flow and precise timing synchronization.All 14 instructions executed correctly, validating the efficiency and automation of the final design.

In summary, the project transformed the classic SAP-1 into an improved, semi-automated processor model that is faster, more efficient, and easier to program.It provides a solid foundation for future enhancements such as flag registers, extended memory, or microcoded control, moving closer to real-world processor design.

Github Repository Link: https://github.com/Adiba2001/SAP-1-Architecture