CHITTAGONG UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Department of Electronics and Telecommunication Engineering

PROJECT REPORT

Design and Implementation of a SAP-1 Architecture with Control Sequencer Using Logisim Evolution

Course No: ETE 404

Course Title: VLSI Technology Sessional

Submitted By

Name:Ishrat Jahan Mily

Student ID: 2008013

Submitted To

Arif Istiaque

Lecturer

Dept. of ETE,CUET

Contents

1 Introduction

This work presents the design and implementation of an enhanced 8-bit SAP-1 (Simple As Possible) computer using Logisim Evolution. The system preserves the traditional single bus architecture of SAP-1 while extending functionality with hardwired control logic and an expanded instruction set including LDA, LDB, ADD, SUB, STA, JMP, and HLT. Two modes of operation are supported: Automatic mode, which executes programs through a structured fetch-decode-execute cycle driven by a ring counter and opcode decoder, and Manual/Loader mode, which allows secure transfer of programs from ROM to RAM. A lightweight web-based assembler was also developed to generate Logisim-compatible machine code from assembly input. Functional verification through arithmetic and control-flow programs confirmed correct micro-operation timing, with memory-based instructions completing within T5 cycles. This implementation provides a practical and extensible platform for instruction in processor design and micro-architectural concepts

2 Objectives

The primary objectives of this research are as follows:

1. To design and implement an enhanced SAP-1 (Simple As Possible) 8-bit computing architecture in the Logisim Evolution environment, integrating additional features for educational demonstration and systematic architectural evaluation.

2. To establish the system on a conventional single-bus structure, featuring an 8-bit data path, 4-bit addressing capability, and a 16-byte memory unit, coordinated by a hardwired control unit that governs the fetch–decode–execute cycle.

3. To support two distinct modes of operation:

   - **Automatic Execution Mode**, driven by a six-phase ring counter (T1–T6) and an opcode decoder for precise synchronization of instruction execution.
   - **Manual/Loader Mode**, allowing program input from ROM or direct user entry into RAM, with provisions for diagnostic signaling and reliable loader handshake protocols.

4. To construct a centralized datapath comprising dual 8-bit general-purpose registers (Accumulator and B-register), a ripple-carry Arithmetic Logic Unit (ALU) with ADD and SUB capabilities, a 4-bit Program Counter with increment and parallel load features, a 4-bit Memory Address Register (MAR), a 16×8 static RAM module, and an Instruction Register (IR) with separate opcode and operand fields—ensuring safe operation via a single-driver bus protocol that prevents contention.

3 Key System Features

1. **Classical SAP-1 foundation (extended):** Single 8-bit data bus with 4-bit address space (16 bytes). Hardwired control (no microcode), preserving a clean didactic design.

2. **Instruction set (extended):** `LDA, LDB, ADD, SUB, STA, JMP, HLT`. Provides arithmetic and memory-operand support beyond the SAP-1 baseline.

3. **Dual operating modes:** Automatic mode: standard fetch–decode–execute cycle driven by a six-stage ring counter (T1–T6). Manual/Loader mode: safe program transfer from ROM or panel input to RAM with loader handshakes and debug support.

4. **Hardwired control unit:** Combines T-states from the ring counter with opcode decoding. Generates precise control signals for register loading, memory access, ALU operations, program sequencing, and halting.

5. **Strict single-driver bus discipline:** All bus sources are tri-stated, ensuring only one active driver per T-state to prevent contention.

6. **Datapath architecture:** Dual 8-bit registers (Accumulator and B-register) with tri-state outputs. Ripple-carry ALU for ADD/SUB operations. 4-bit Program Counter with increment and direct load capability. 4-bit Memory Address Register, 16×8 SRAM, and an 8-bit Instruction Register partitioned for opcode and operand handling.

7. **Opcode decoder:** 4-to-16 one-hot decoder generates control signals for each instruction.

8. **Precise timing**
   Memory-operand instructions (`LDA, LDB, STA`) complete in T5. Arithmetic (`ADD, SUB`) and control-flow (`JMP, HLT`) instructions execute in T4.

9. **Assembler/compiler support:** Lightweight web-based assembler converts human-readable assembly (`ORG`, `DEC` + mnemonics) into Logisim v2.0 HEX format for direct ROM/RAM loading.

10. **Verification-friendly design:** Step-through probing of PC, MAR, IR, A/B registers, ALU output, bus, and SRAM. Provides clear signal visibility for debugging and educational analysis.

11. **Extensible template:** Control matrix and decoder designed for easy addition of future instructions or flags (e.g., Zero/Carry).

4 Architectural Design and Component Analysis

4.1 System Architecture Overview

The processor architecture uses a unified single-bus design with an 8-bit data pathway controlled by tri-state sources. Bus arbitration ensures that only one driver is active during each T-state, with possible drivers including `pc_out`, `sram_rd`, `ins_reg_out_en`, `a_out`, `b_out`, `alu_out`, and `sh_out`. Bus listener components such as `mar_in_en`, `ins_reg_in_en`, `a_in`, `b_in`, and `sram_wr` allow selective data capture when required.

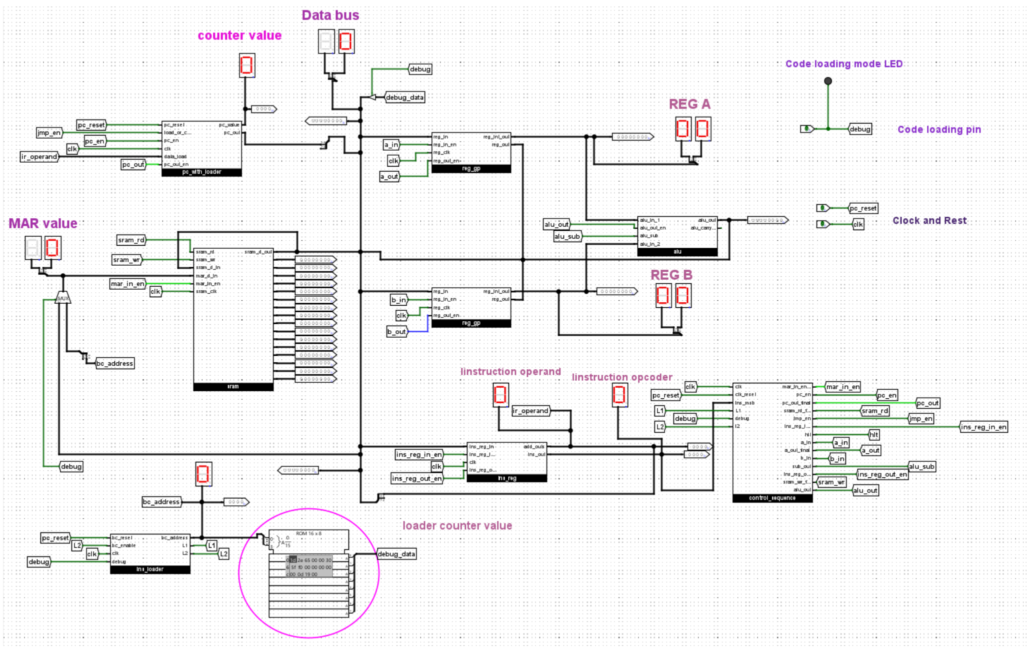


Figure 1: Automatic mode operation of the control sequencer showing fetch–decode–execute sequencing.

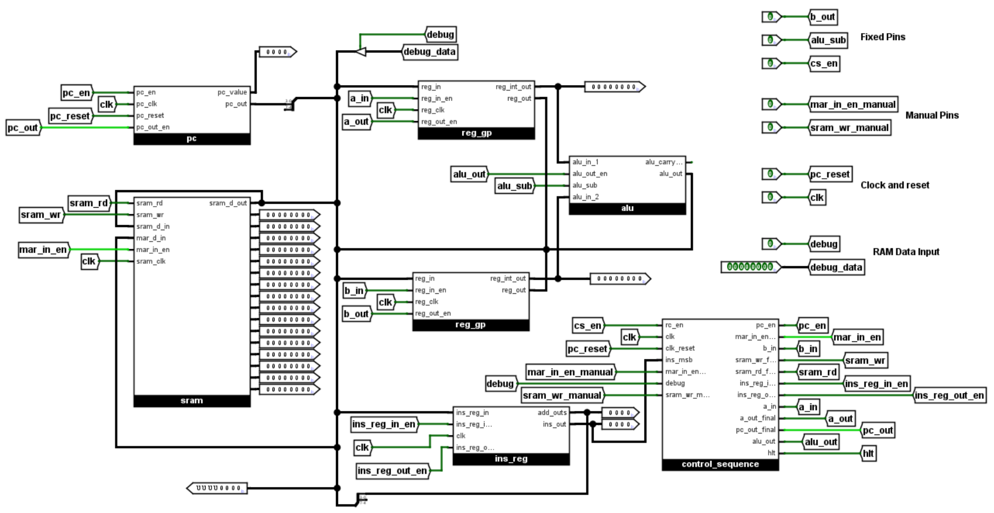


Figure 2: Manual/Loader mode operation of the control sequencer showing secure program loading with debug and handshake signals..

4.2 Register Implementation (A, B)

The **register subsystem** utilizes **standardized reg_gp building blocks** providing **8-bit storage capacity** with **comprehensive interface capabilities**:

1. **Input Interface:** The `reg_in` lines are connected to the system bus, with data transfers governed by the `a_in` and `b_in` control signals. This mechanism ensures proper latching of data into the respective registers.

2. **Output Interface:** The `reg_out` lines provide bus-driving capability through tri-state logic. The `a_out` and `b_out` signals activate this interface, allowing controlled data placement onto the system bus.

3. **Internal Interface:** The `reg_int_out` lines offer continuous data availability to the Arithmetic Logic Unit (ALU) without engaging the shared bus. This feature enables direct computational access and eliminates unnecessary bus utilization.

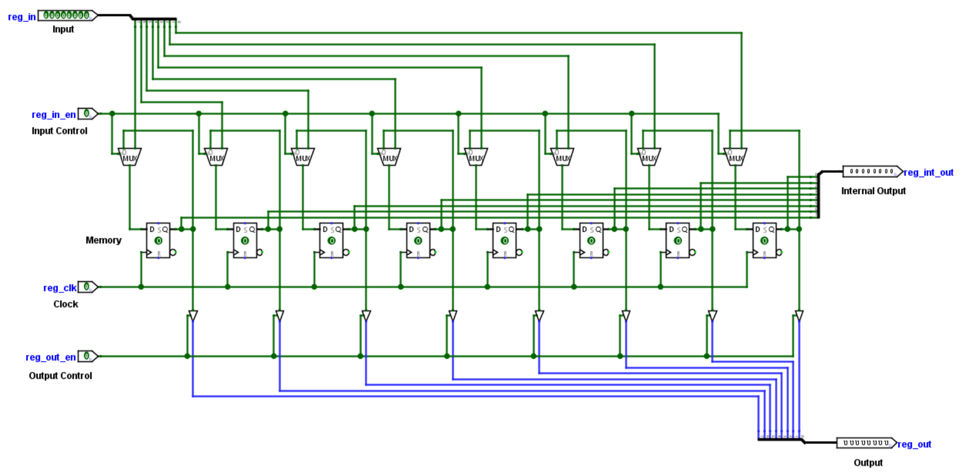


Figure 3: Schematic representation of the A/B register subsystem, highlighting input, output, and internal interfaces with tri-state control and direct datapath connectivity.

This **dual-interface design** enables **ALU/Shifter operations** to **read A/B register contents** without **bus activation**, significantly improving **operational efficiency** and **eliminating bus contention** during **computational phases**.

4.3 Program Counter Design

The **Program Counter implementation** provides **dual operational capabilities**:

- **Increment Mode**: **T3 timing** with `pc_en = 1` executes `PC ← PC + 1` for **sequential instruction progression**
- **Jump Mode**: **JMP instruction execution** at **T4** with `jump_en = 1` and **IR low nibble on bus** executes `PC ← target address` for **program flow control**
- **Bus Interface**: `pc_out = 1` during **T1 fetch operations** enables **PC value placement** on **system bus** for **MAR loading**

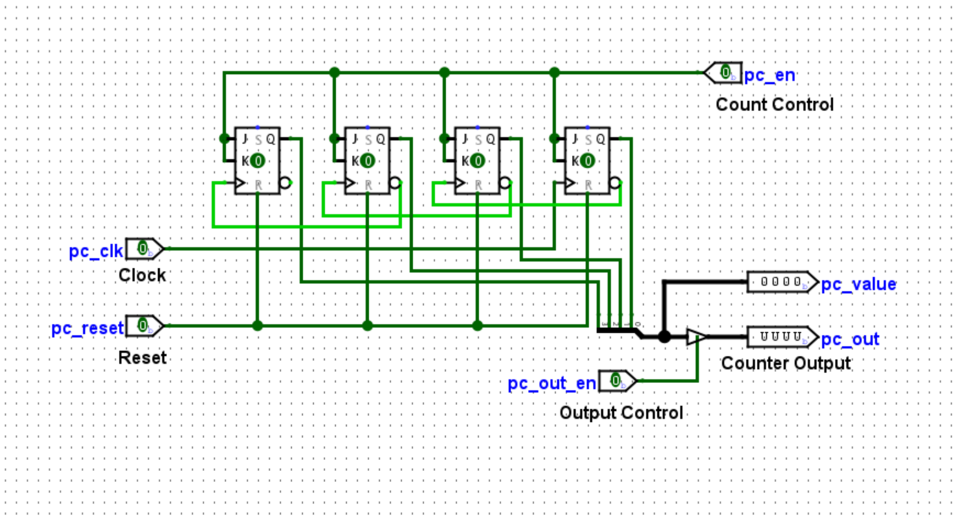


Figure 4: Program Counter implementation showcasing dual operational modes, including sequential increment and direct load functionality, to ensure comprehensive program flow control.

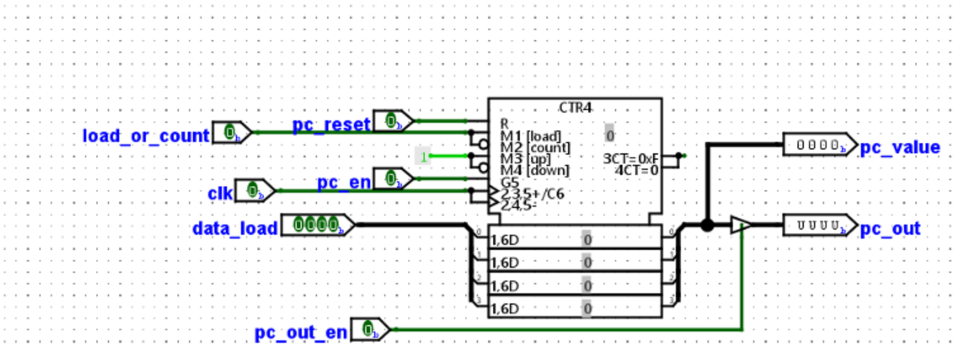


Figure 5: Program Counter implementation showcasing dual operational modes, including sequential increment and direct load functionality, to ensure comprehensive program flow control.

4.4 Memory System and Address Register

The **memory subsystem architecture** incorporates **4-bit MAR (Memory Address Register)** providing **bus capture functionality** via `mar_in_en` control:

- **Fetch Addressing**: **T1 phase** with `pc_out + mar_in_en` executes `MAR ← PC` for **instruction fetch addressing**
- **Operand Addressing**: **T4 phase** for **LDA/LDB/STA/JMP operations** with `ins_reg_out_en + mar_in_en` executes `MAR ← IR[3:0]` for **operand addressing**

SRAM Operation Modes:

- **Read Operations**: `sram_rd = 1` executes `RAM[MAR]` → `bus` during **T2 fetch operations** and **T5 LDA/LDB operations**
- **Write Operations**: `sram_wr = 1` with **bus data** executes `RAM[MAR]` ← `bus` during **T5 STA operations**

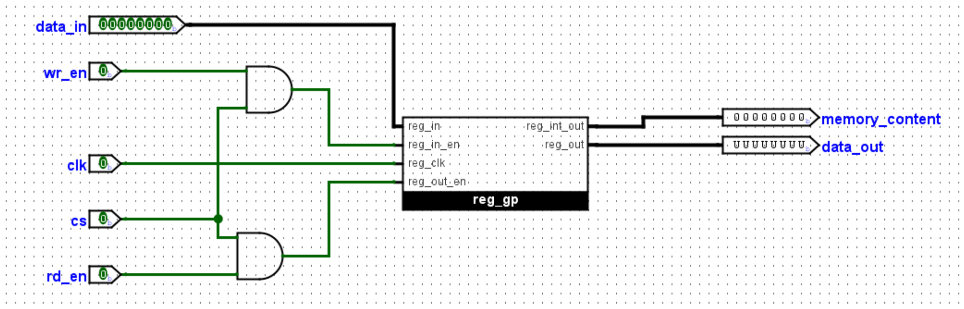


Figure 6: Register-based memory element showing data input (`data_in`), write enable (`wr_en`), read enable (`rd_en`), clock, and chip select (`cs`) control signals. The stored value is available as `memory_content`, while the output to the bus is provided through `data_out`.

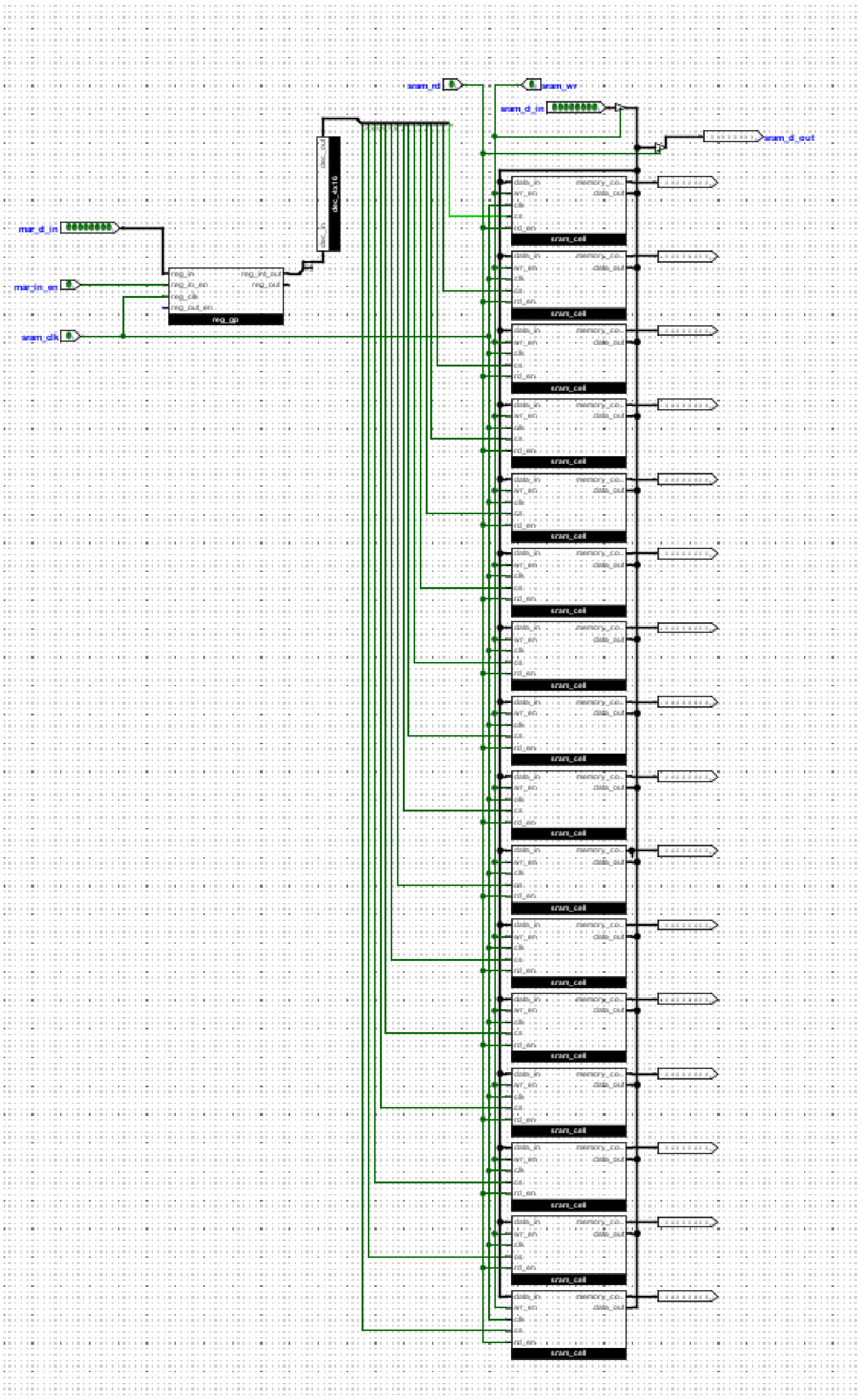


Figure 7: Memory subsystem showing MAR operation and SRAM interface with read-/write timing for reliable data access.

4.5 Instruction Register and Opcode Decoder

The **Instruction Register (IR) architecture** implements **dual-function design**:

- **IR Loading**: **T2 phase** with `sram_rd=1, ins_reg_in_en=1` executes `IR ← M[MAR]` for **instruction capture**
- **Opcode Processing**: **IR[7:4]** routes to **ins_tab decoder** generating **one-hot opcode lines**
- **Operand Processing**: **IR[3:0]** provides **bus driving** via `ins_reg_out_en=1` during **T4 address/target operations**

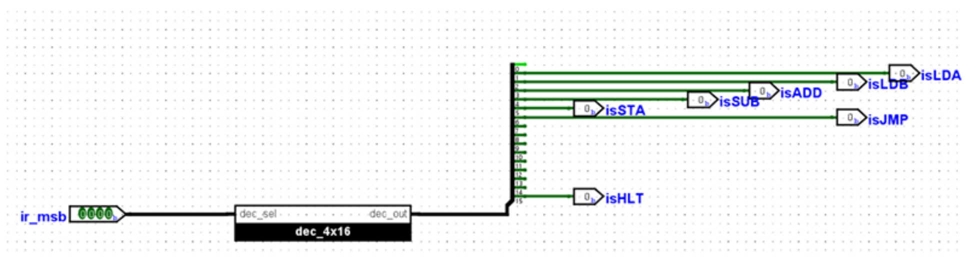


Figure 8: Architecture of the instruction register and opcode decoder, showing instruction loading, opcode routing to the decoder, and operand forwarding for execution.

The **opcode decoder (ins_tab)** implements **4→16 decoding** producing **one-hot activation lines**: `insLDA, insLDB, insADD, insSUB, insSTA, insJMP, insHLT` with **unused decoder outputs** reserved for **future instruction expansion**.

4.6 Arithmetic Logic Unit Implementation

The ALU subsystem performs 8-bit arithmetic processing and is characterized by the following operational features:

- **Input Sources:** The operands are directly supplied from `A.reg_int_out` and `B.reg_int_out`, enabling register-level access without requiring bus utilization.
- **Operation Control:** The control signal `alu_sub = 1` selects the subtraction operation ($A - B$); otherwise, the addition operation ($A + B$) is executed. This binary control allows minimal logic overhead while maintaining flexible arithmetic capability.
- **Execution Protocol:** During the T4 phase, when `a_out = 1`, `b_out = 1`, `alu_out = 1`, and `a_in = 1`, the ALU performs the operation in a single step as $A \leftarrow A \pm B$. This protocol ensures exclusive ALU bus driving and supports efficient one-cycle execution.
- **Architectural Design:** The ALU is implemented using a ripple-carry adder structure with integrated control logic for mode selection. While this approach maintains

hardware simplicity, it introduces a carry propagation delay that scales linearly with operand width. For the current 8-bit design, this delay remains within acceptable performance limits.

- **Bus Interface:** The ALU output is enabled onto the system bus only when `alu_out = 1`, ensuring strict bus discipline and preventing contention with other subsystems. This tri-state interfacing mechanism maintains the integrity of shared data transfers.

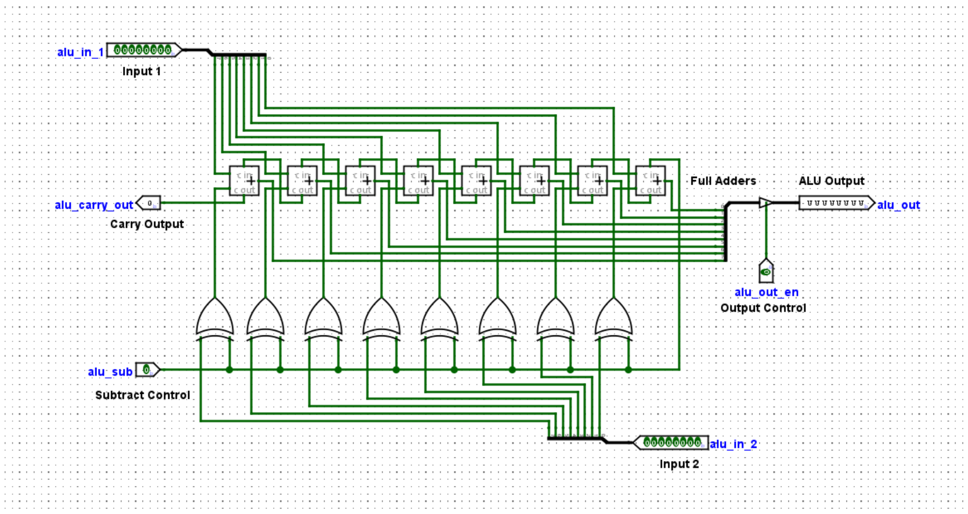


Figure 9: Arithmetic Logic Unit (ALU) implementation illustrating a ripple-carry architecture with a tri-state bus interface and structured operation control, enabling efficient and reliable 8-bit arithmetic processing

4.7 Boot/Loader Counter and Phase Generation

The **loader subsystem (ins_loader)** facilitates **secure ROM-to-RAM program transfer** in **Manual/Loader mode**:

1. **Functional Role:** Provides safe program loading from ROM to RAM when the system is placed in Manual/Loader mode (`debug = 1`), ensuring that normal fetch–execute logic is disabled during this process.

2. **Input Interface:** Accepts standard control signals including `clk`, `bc_reset` for counter reset, `bc_en` for count enable, and the `debug` signal for mode selection.

3. **Address Sequencing:** Employs a 4-bit `CTR4` counter configured for upward counting, producing sequential addresses `bc_address[3:0]` ranging from `0000` to `1111` for systematic RAM write operations.

4. **Phase Control:** Utilizes a D flip-flop with feedback inversion to generate two non-overlapping clock phases (Φ and $\neg\Phi$), which synchronize data transfer and ensure contention-free operation.

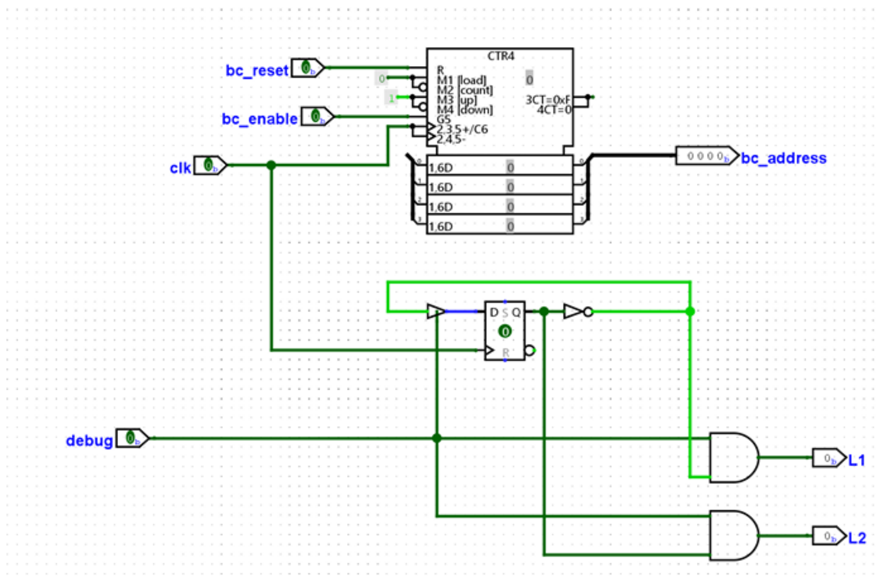


Figure 10: Architecture of the boot/loader subsystem, showing sequential address generation and dual-phase clocking for reliable ROM-to-RAM transfer in Manual/Loader mode.

5 Control System Design

The **control unit architecture** converts **individual instructions** into **precisely timed control pulse sequences** that (a) select **exclusive bus driver activation**, and (b) enable **appropriate latch operations** during **each T-state period**. The **control subsystem** incorporates:

- **Ring Counter (rc)** generating **T1..T6 timing states**
- **Opcode Decoder (ins_tab)** producing activation lines for instruction-specific micro operations
- **Mode Control Inputs**: `debug` (manual/loader selection), `i1/i2` (loader handshake protocols) defining `cpu_mode = ~debug` with **loader masking via** `~i2`

The **control sequencer implementation** operates through **dual operational paradigms** corresponding to the system's **Manual/Loader** and **Automatic execution modes**. Each operational mode utilizes distinct control logic pathways optimized for their respective functional requirements while maintaining comprehensive signal integrity and timing coordination.

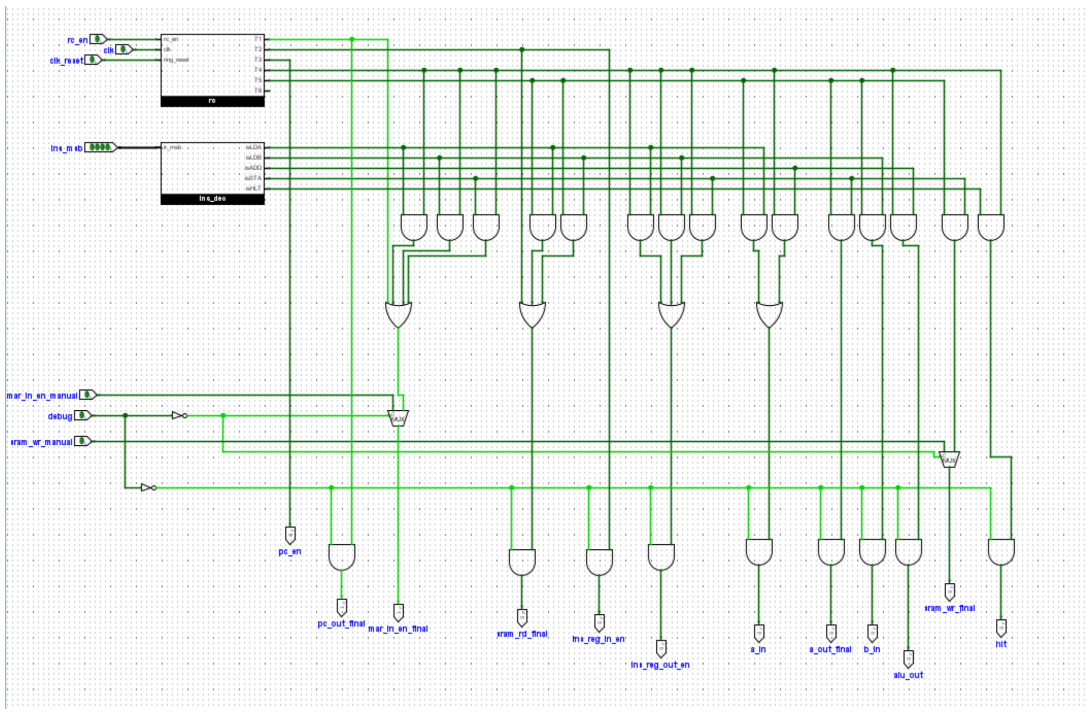


Figure 11: Manual mode control sequencer architecture supporting ADD instruction execution with basic timing coordination for step-wise validation.

The **Manual mode control sequencer** provides a simplified execution pathway, supporting only the ADDinstruction. This minimal configuration is intended for step-wise verification and manual debugging of instruction flow, allowing clear observation of bus activity and control signal sequencing during arithmetic operation

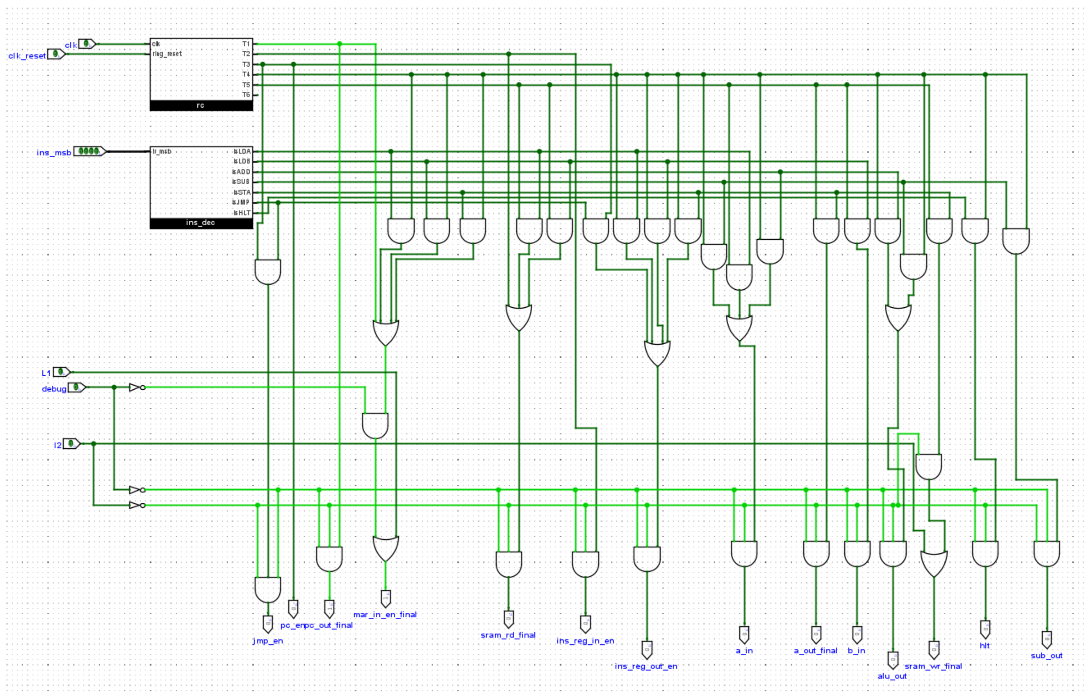


Figure 12: Automatic mode control sequencer demonstrating full instruction set support with ADD, SUB, and JMP execution pathways using hardwired control logic.

The **Automatic mode control sequencer** orchestrates **standard instruction execution** through sophisticated **fetch-decode-execute cycling** with precise **micro-operation timing control**. This architecture integrates **ring counter coordination** with **opcode decoding logic** to generate comprehensive **control signal assertions** for all supported instruction types, ensuring optimal performance and timing precision throughout program execution phases.

5.1 Timing Control Generator

The **Ring Counter implementation** provides **six-phase timing generation (T1-T6)** orchestrating **fetch-decode-execute sequencing**:

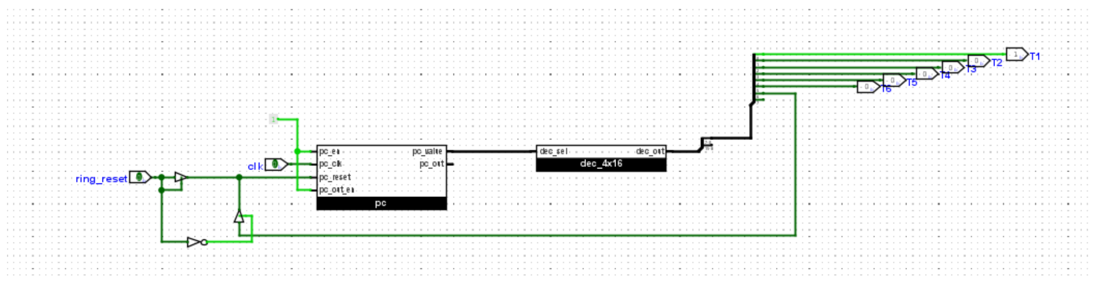


Figure 13: Six-phase ring counter implementation for timing control, enabling systematic fetch–decode–execute sequencing and precise micro-operation coordination.

Universal Fetch Sequence (all instructions):

- **T1**: `pc_out, mar_in_en` executing `MAR ← PC`
- **T2**: `sram_rd, ins_reg_in_en` executing `IR ← M[MAR]`
- **T3**: `pc_en` executing `PC ← PC + 1`

Representative Execute Sequences
Execution timing varies according to instruction type. Representative cases include:

- **LDA addr: T4:** Operand address (`IR[3:0]`) placed on bus (`ins_reg_out_en`) and loaded into MAR (`mar_in_en`), executing $MAR \leftarrow IR[3:0]$. **T5:** Memory value read (`sram_rd`) and latched into A (`a_in`), performing $A \leftarrow M[MAR]$.
- **ADD: T4:** A and B register outputs (`a_out`, `b_out`) drive ALU; output (`alu_out`) fed into A (`a_in`) with subtraction disabled (`alu_sub=0`).
- **SUB: T4:** Similar to ADD, except subtraction enabled (`alu_sub=1`), performing $A \leftarrow A - B$.
- **JMP addr: T4:** Operand (`IR[3:0]`) placed on bus (`ins_reg_out_en`) and loaded into PC (`pc_en`), executing $PC \leftarrow IR[3:0]$.

5.2 Automatic Operation Control Logic

The **control equation implementation** defines **core minterms** with `C = cpu_mode = ~debug` and `L = ~i2` (loader idle):

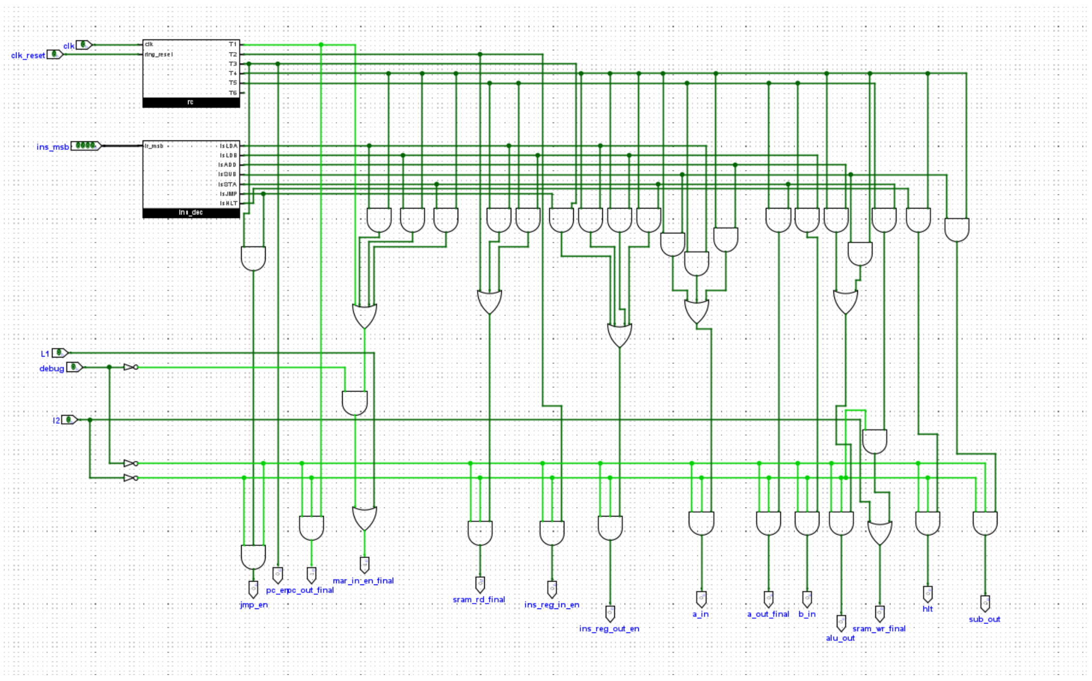


Figure 14: Gate-level control matrix demonstrating comprehensive control equation implementation with automatic mode logic and precise timing coordination for instruction execution sequencing

Listing 1: Fetch Control Equations

```
pc_out = T1 & C
mar_in_en = (T1 & C) | (T4 & C & (insLDA | insLDB | insSTA |
   insJMP))
sram_rd = (T2 & C) | (T5 & C & (insLDA | insLDB))
ins_reg_in_en = T2 & C
pc_en = T3 & C
```

Listing 2: ALU and Register Control Equations

```
alu_out = T4 & C & (insADD | insSUB)
alu_sub = T4 & C & insSUB
a_in = (T5 & C & insLDA) | (T4 & C & (insADD | insSUB))
b_in = T5 & C & insLDB
a_out = (T4 & C & (insADD | insSUB)) | (T5 & C & insSTA)
b_out = T4 & C & (insADD | insSUB)
```

5.3 Manual/Loader Operation Control

In Manual/Loader mode, the control mechanism relies on the debug=1 signal, which masks the normal CPU fetch–decode–execute logic. This ensures that the processor's standard operation is suspended while external program loading is performed safely.

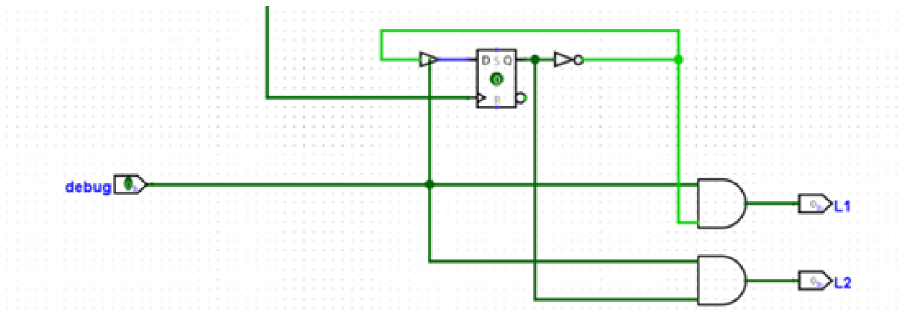


Figure 15: : Architecture of the Manual/Loader control system, illustrating debug-based masking and handshake signaling for secure and conflict-free program loading.

Program transfer is coordinated using a handshake protocol. The signals i1 and i2 govern the sequence of operations, where i1 enables address placement into the Memory Address Register (MAR), and i2 activates write operations into SRAM. This stepwise handshake process guarantees that memory loading occurs without contention, while the debug masking mechanism prevents interference from the normal CPU execution path.

6 Instruction Set Architecture

6.1 Instruction Encoding Scheme

The instruction encoding system utilizes upper nibble = IR[7:4] for opcode specification and lower nibble for 4-bit operand/address when required:

Table 1: Instruction Set & Program

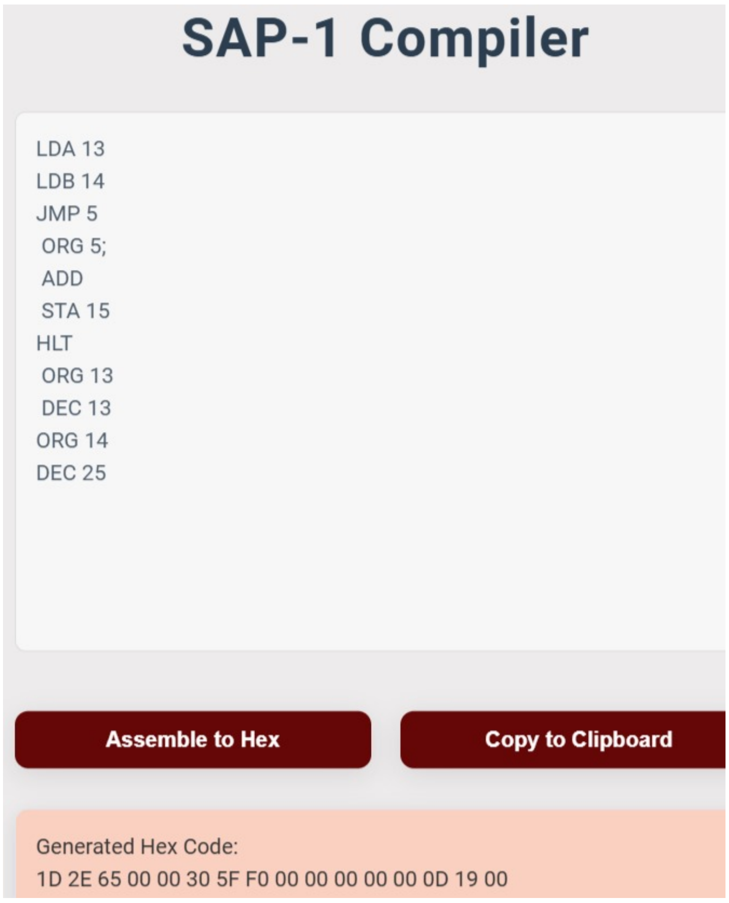
Address	Instruction	Hex	Mnemonic & Explanation
00000000	00011101	1D	LDA 13 (Load A from M[13])
00000001	00101110	2E	LDB 14 (Load B from M[14])
00000010	01100101	65	JMP 5 (PC $\leftarrow$ 5)
00000011	00110000	30	ADD (A $\leftarrow$ A + B)
00000100	01011111	5F	STA 15 (M[15] $\leftarrow$ A)
00000101	11110000	F0	HLT (Stop execution)

Table 2: Data Values in RAM

Address (Binary)	Data (Binary)	Decimal	Hex
00001101	00001101	13	0D
00001110	00011001	25	19

6.2 Assembler

The assembler translates SAP-1 assembly language programs into machine code (hexadecimal) suitable for execution in Logisim. It supports instructions such as **LDA, LDB, ADD, SUB, STA, JMP, HLT**, along with directives like **ORG** and **DEC**. The tool automatically generates Logisim-compatible v2.0 raw hex output, avoiding manual conversion errors. This enables efficient program development, testing, and debugging of the SAP-1 system.



```
LDA 13
LDB 14
JMP 5
 ORG 5;
 ADD
 STA 15
HLT
 ORG 13
 DEC 13
ORG 14
DEC 25
```

```
Generated Hex Code:
1D 2E 65 00 00 30 5F F0 00 00 00 00 00 0D 19 00
```

Figure 16: Web-based SAP-1 assembler interface converting assembly instructions into Logisim-compatible hexadecimal code.

Table 3: Examples of Assembly Programs and Corresponding Hex Codes

Example	Assembly Code	Hex Code
ADD Program	LDA 13; LDB 14; ADD; STA 15; HLT; ORG 13; DEC 13; ORG 14; DEC 25	1D 2E 30 5F F0 0D 19 00
JMP + ADD Program	LDA 13; LDB 14; JMP 5; ORG 5; ADD; STA 15; HLT; ORG 13; DEC 13; ORG 14; DEC 25	1D 2E 65 00 00 30 5F F0 00 00 00 00 00 0D 19 00

7 Operation

7.1 Fetch–Decode–Execute Cycle

Fetch

- **T1:** The Program Counter (PC) outputs the current instruction address onto the bus, which is then stored in the Memory Address Register (MAR).
- **T2:** The memory unit provides the instruction stored at the MAR address onto the data bus, and the Instruction Register (IR) captures this instruction.
- **T3:** The PC is incremented so it is ready to point to the next instruction in sequence.

Decode The opcode portion of the IR is forwarded to the instruction decoder, which activates the appropriate control line (e.g., `LDA`, `ADD`). This decoded output, combined with the active timing state (T-state), defines the exact set of control signals required for execution.

Execute The control unit asserts the relevant signals to carry out the micro-operations of the decoded instruction. The number of clock states required depends on the instruction type—for example, `LDA` typically requires two states, `ADD` also takes two, while `HLT` completes in a single state. This cycle continues automatically, instruction by instruction, until a `HLT` command is reached, at which point the CPU halts and the state counter is stopped.

7.2 Running the CPU in Manual Mode

To run the SAP-1 CPU in Manual/Loader mode, the following steps are followed:

1. Initial Setup

- Ensure the `debug` pin is OFF (LOW) to enable automated control.
- Pulse the `pc_reset` pin once to reset the Program Counter (PC) to 0000.
- Ensure the main clock (`clk`) is OFF. For step-by-step execution, use the manual clock button.
- Set the `cs_en` pin to ON (HIGH) to enable the circuit.

2. Program the RAM (Debug Mode)

- Turn ON the `debug` pin (HIGH). This enables manual RAM programming.
- For each instruction/data:

1. Set address: Use `debug_data` to define the 8-bit memory address.
2. Load address into MAR: Pulse `mar_in_en_manual`.
3. Set instruction/data: Use `debug_data` to provide the 8-bit value.
4. Write to RAM: Pulse `sram_wr_manual`.

- After all instructions/data are loaded, turn OFF the `debug` pin (LOW).
- Pulse `pc_reset` to reset PC to 0000 for program execution.

3. Run the Program

- **Manual stepping:** Press the `clk` button repeatedly to step through the Fetch–Decode–Execute cycle, observing PC, MAR, IR, A/B registers, and RAM.
- **Continuous run:** Enable the continuous clock source for automated execution.

4. Observe HLT When the `HLT` instruction is reached, the CPU halts, stopping the clock or state counter.

5. Verify Results At RAM address `00001111` (decimal 15), the stored value should be `00100110` (binary), corresponding to decimal 38 or hexadecimal 26, which confirms the correct addition of 13 and 25.

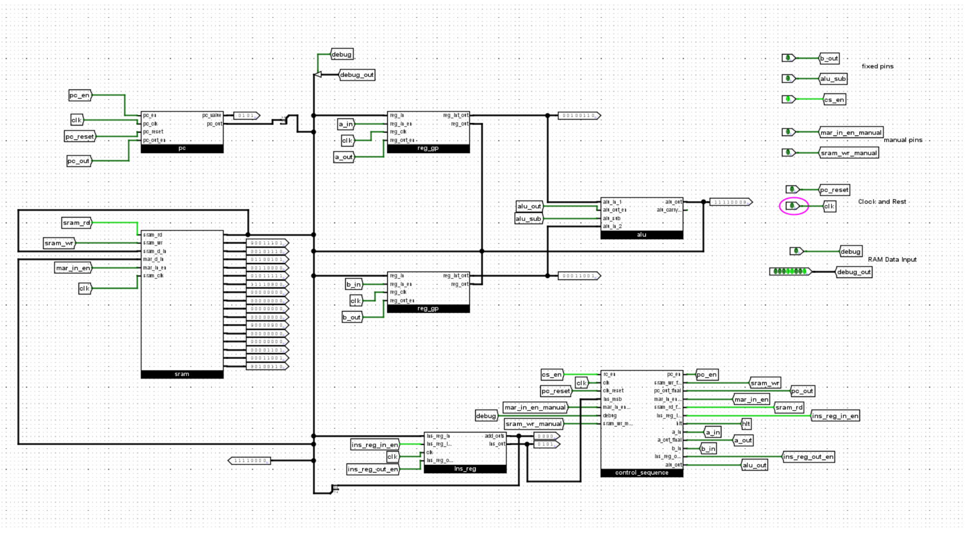


Figure 17: SAP1 CPU circuit implementation in Logisim Evolution,highlighting debug signals, control pins, and RAM verification for program execution.

7.3 Running the CPU in Automatic Mode (JMP + ADD Program)

8 Running the CPU in Automatic Mode (JMP + ADD Program)

To execute the CPU in automatic mode using the program with `JMP` and `ADD` instructions, follow the procedure below:

1. Initial Setup

[label=(h)]

1. Open the file in Logisim Evolution.
2. Ensure the `debug` pin is set to **LOW**.
3. Ensure the main clock (`clk`) is **OFF**.
4. Pulse the `pc_reset` pin once to reset the Program Counter to `0000`.

2. Program the ROM

1. Right-click the ROM component and select `Edit Contents.`
2. Enter the following hex code sequence into the ROM memory:

```
1D 2E 65 00 00 30 5F F0 00 00 00 00 00 0D 19 00
```

3. Alternatively, upload the prepared `instruction_code_jmp_add` file (if provided).

3. Load Program to RAM (Bootloader Mode)

1. Set the `debug` pin to **HIGH**. The *Code Loading Mode* LED will turn ON.
2. With each `clk` pulse, the CPU will copy the program from ROM into RAM. Two clock pulses are required per instruction/data value.
3. Allow the CPU to complete writing all instructions and data into RAM.
4. Observe MAR and Data Bus activity on the 7-segment displays during this phase.

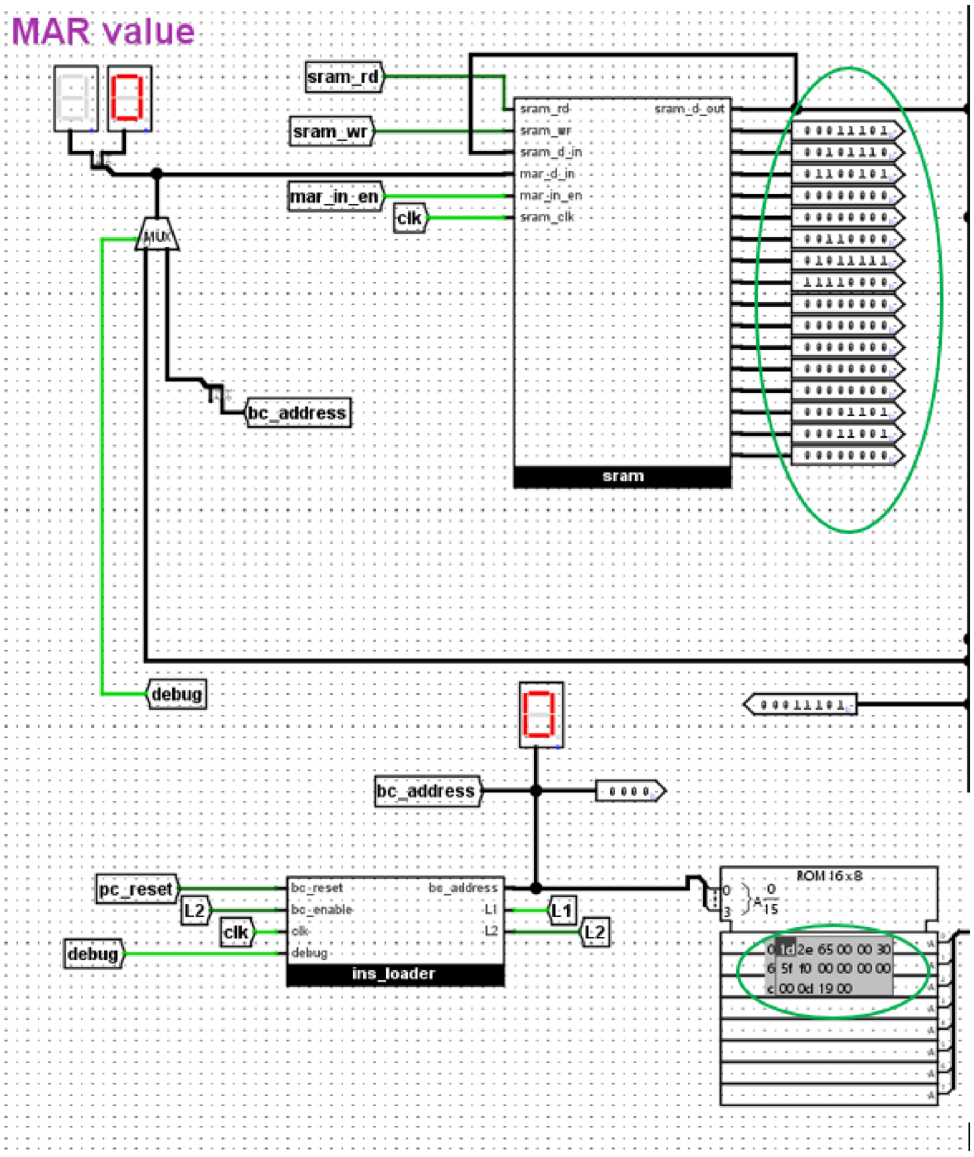


Figure 18: After loading all instruction and data values into RAM memory.

4. Stop the Bootloader

1. Set the `debug` pin back to **LOW**.
2. Pulse the main `clk` once to ensure the bootloader process stops fully.

5. Run the Program

1. Pulse `pc_reset` again to reset the Program Counter to `0000`.
2. Provide clock pulses (manual clicking or continuous clock) to let the CPU execute.
3. Observe PC, MAR, IR, Register A, and Register B values in the 7-segment displays through the **Fetch–Decode–Execute** cycle.

4. Execution sequence:

   - Fetch `LDA(13)`: load value at address 13 into Register A.
   - Fetch `LDA(14)`: load value at address 14 into Register B.
   - Execute `JMP 5`: Program Counter jumps to address 5.
   - At address 5, execute `ADD` (Register A + Register B).
   - Execute `STA(15)`: store the result into address 15.
   - Execute `HLT`: stop the CPU.

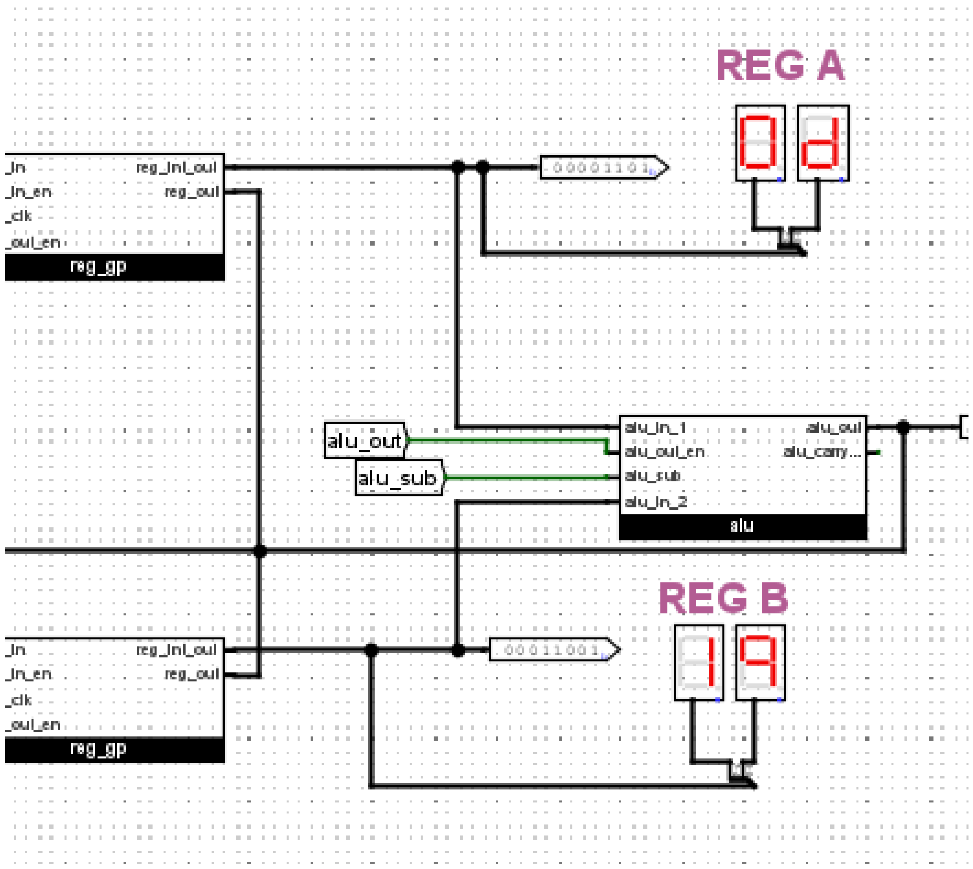


Figure 19: After executing LDA 13 and LDA 14: the value 13 and 25 is loaded from memory address to the RegA and RegB.

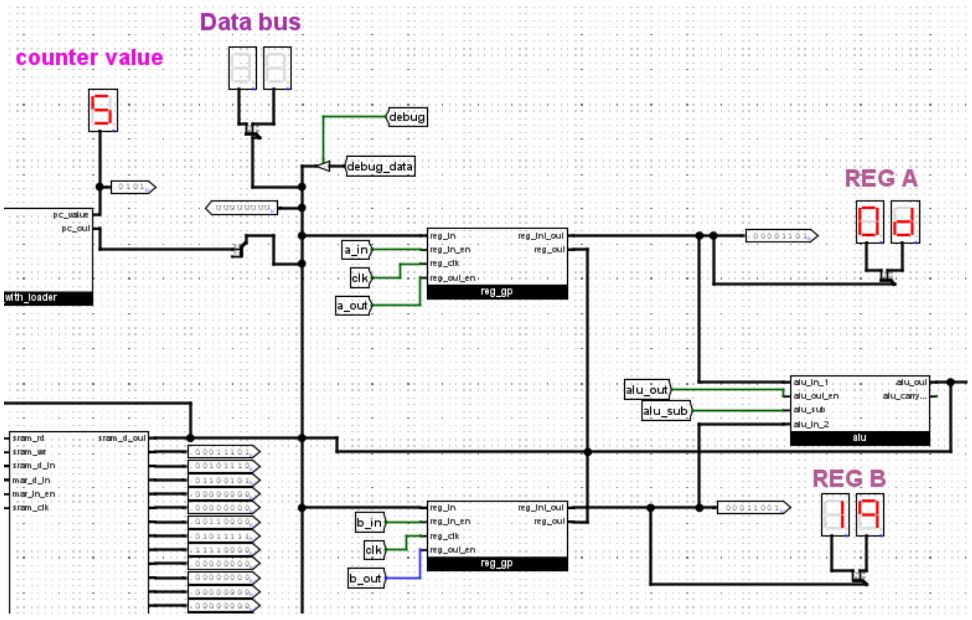


Figure 20: After executing JMP 5: the Program Counter is updated to address 5, redirecting the execution flow.

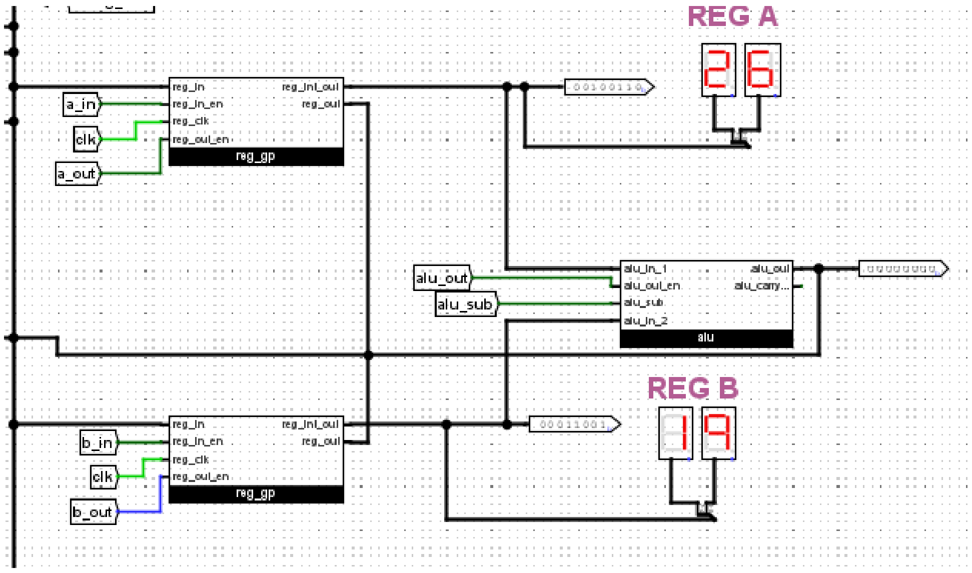


Figure 21: After executing ADD: the contents of Register A and Register B are added, and the result is stored back into Register A.

6. Verify Result

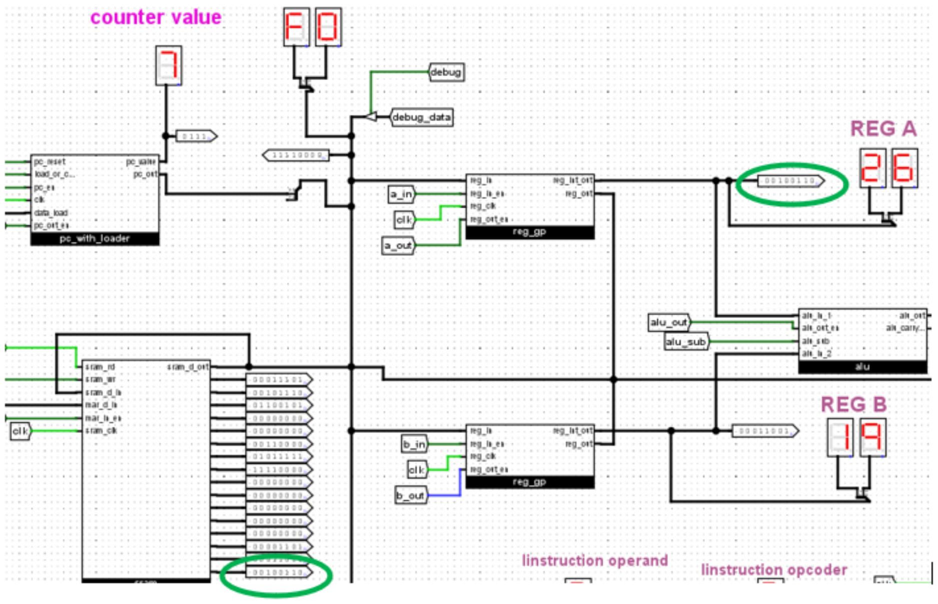


Figure 22: After executing STA 15: the result from Register A is written into memory address 15.

At RAM address `00001111` (decimal 15), the stored value should be `00100110` (binary), corresponding to decimal **38** or hexadecimal `26`. This confirms the correct addition of 13 and 25.

9 Conclusion

The enhanced SAP-1 implementation successfully bridges classical processor design principles with modern simulation-based educational practices. By incorporating dual operational modes, expanded instruction support, and a rigorously structured control sequencer, the system demonstrates both technical soundness and pedagogical clarity. Validation through diverse program executions confirmed the correctness of architectural decisions, control logic design, and timing coordination.

This project establishes a practical and extensible platform for undergraduate education in computer architecture, offering students direct experience with instruction encoding, bus protocols, and micro-operation sequencing. Beyond its educational impact, the modular and open design creates opportunities for further research in control optimization and processor design methodologies.

Overall, the work highlights the continued relevance of foundational processor concepts while providing an adaptable framework for future advancements in both academic instruction and applied computing research.

10 Project Resources

10.1 Source Code Repository

The complete Logisim design files and project resources are available in the GitHub repository: GitHub Repository: SAP-1 CPU Project

10.2 Demonstration Video

The demonstration of the SAP-1 CPU with control sequencer can be viewed at the following link: YouTube Video: SAP-1 CPU Demonstration