Chittagong University of Engineering and Technology

Department of Electronics and Telecommunication Engineering

PROJECT REPORT

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

Course No:	**ETE 404**
Course Title:	**VLSI Technology Sessional**
Level:	**04**
Term:	**I**

Submitted By:	**Submitted To:**
Sayeda Rahman Ananna ID: 2008044	Arif Istiaque Lecturer Dept. of ETE, CUET

Contents

1 Project Overview

This document reports on the design and implementation of a functionally enhanced 8-bit SAP-1 computer within the Logisim Evolution environment. The implementation preserves the foundational single-bus architecture while incorporating hardwired control logic and a broader instruction set. Two distinct operational modes are supported: an automatic execution mode governed by a structured control unit, and a manual mode dedicated to the secure loading of programs. Development was complemented by a lightweight web-based assembler to facilitate program creation. Functional verification through benchmark programs validated the timing of micro-operations, establishing the system as a robust and extensible platform for instructional use in computer architecture.

2 Objectives

The principal aims of this research endeavor are as follows:

1. To realize a functionally augmented SAP-1 (Simple As Possible) 8-bit computing architecture within the Logisim Evolution environment, integrating advanced capabilities to facilitate comprehensive educational verification and systematic architectural analysis.

2. To conceive the system's foundation upon a conventional single-bus topology, characterized by an 8-bit data path, a 4-bit addressing capability, and a 16-byte memory subsystem, all managed by a hardwired control unit that directs the fundamental fetch-decode-execute sequence.

3. To establish two distinct operational modalities:

   - **Automatic Execution Mode**, governed by a six-phase ring counter (T1–T6) and an opcode decoding unit to ensure accurately synchronized instruction processing.
   - **Manual/Loader Mode**, enabling regulated program input, either from ROM or via direct user entry, into system RAM, incorporating diagnostic signaling and reliable loader handshake protocols.

4. To architect a centralized datapath featuring dual 8-bit general-purpose registers (Accumulator and B-register), a ripple-carry Arithmetic Logic Unit (ALU) supporting ADD and SUB functions, a 4-bit Program Counter with inherent increment and parallel load functionalities, a 4-bit Memory Address Register (MAR), a 16x8 static RAM module, and an Instruction Register (IR) configured for separate opcode and operand storage—all operating under a strict single-driver bus protocol to prevent signal contention.

3 Key Features

1. **Extended Classical SAP-1 Architecture** Implements a foundational single 8-bit data bus with 4-bit addressing (16-byte memory space). Utilizes a hardwired control system without microcode, maintaining architectural clarity for educational purposes.

2. **Enhanced Instruction Set Architecture** Builds upon the SAP-1 baseline with an expanded instruction set including LDA, LDB, ADD, SUB, STA, JMP, and HLT, providing comprehensive memory-operand and arithmetic capabilities.

3. **Dual-Mode Operational Flexibility** Features two distinct operational modes: Automatic execution using a six-phase ring counter (T1–T6) for standard fetch-decode-execute sequencing, and Manual/Loader mode enabling secure program transfer from ROM or direct input to RAM with handshake protocols and debugging support.

4. **Hardwired Control System Implementation** Integrates timing states from a ring counter with instruction decoding logic to generate precise control signals for register operations, memory access, ALU functions, program flow control, and system halting.

5. **Rigorous Bus Management Protocol** Enforces strict single-driver bus discipline through tri-state outputs on all bus-connected components, ensuring conflict-free operation by activating only one data source per timing state.

6. **Comprehensive Datapath Design** Incorporates dual 8-bit working registers (Accumulator and B-register) with tri-state interfaces, a ripple-carry ALU supporting ADD/SUB operations, 4-bit Program Counter with increment/load capabilities, Memory Address Register, 16×8 SRAM module, and an Instruction Register configured for separate opcode-operand storage.

7. **Optimized Instruction Timing** Achieves efficient execution timing with memory-operand instructions (LDA, LDB, STA) completing in T5 cycles, while arithmetic (ADD, SUB) and control-flow (JMP, HLT) instructions finish in T4 cycles.

8. **Software Development Support** Includes a lightweight web-based assembler tool that translates human-readable assembly code (using ORG, DEC directives and mnemonics) into Logisim v2.0 compatible HEX format for direct memory loading.

9. **Verification and Debugging Capabilities** Designed with comprehensive observability, allowing step-through examination of PC, MAR, IR, register contents, ALU outputs, bus states, and memory contents for effective debugging and educational analysis.

10. **Extensible Architecture Framework** Structured with expandable control logic and decoding systems to facilitate future instruction set extensions and potential addition of status flags (Zero, Carry).

4 Architecture and Functional Block 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.

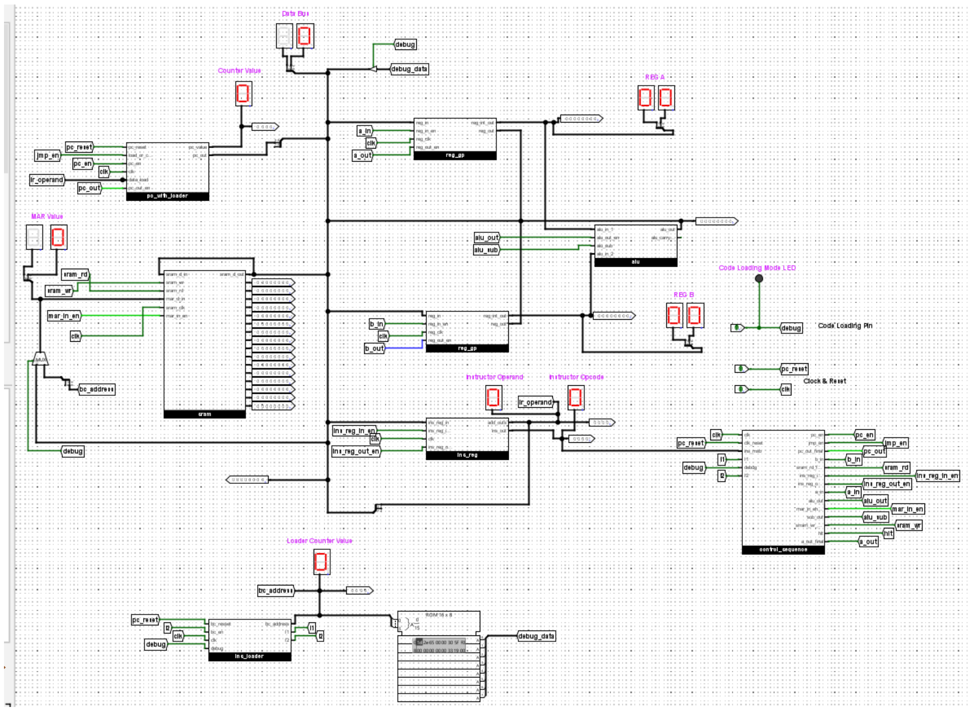


Figure 1: Automatic mode operation of the control sequencer showing fetch–decode–execute sequencing.

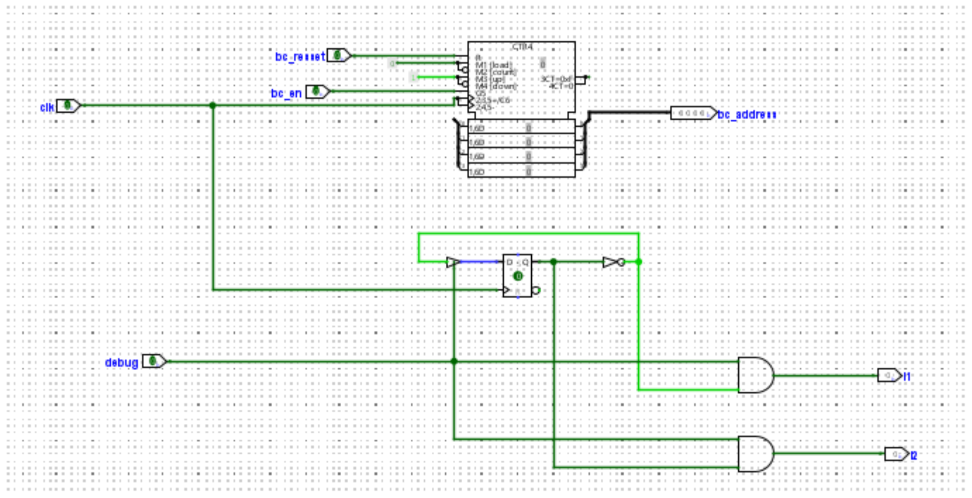


Figure 2: Manual/Loader mode operation of the control sequencer showing secure program loading with debug and handshake signals.

4.2 Register Implementation (A, B)

The A and B registers are realized using standardized `reg_gp` modules, each capable of storing 8-bit data. The design incorporates three distinct interfaces that enable efficient communication within the datapath:

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.

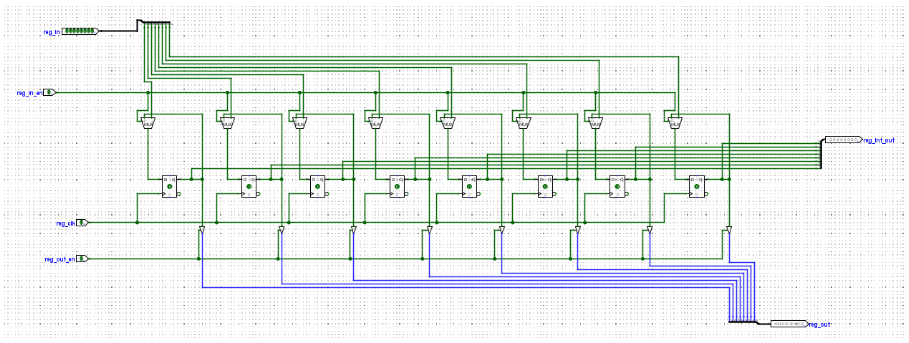


Figure 3: Schematic representation of the A/B register subsystem, highlighting input, output, and internal interfaces with tri-state control and direct datapath connectivity.

This structured interface arrangement allows the ALU to access register contents directly, thereby enhancing execution efficiency while preserving strict single-driver bus discipline.

4.3 Program Counter (PC) Implementation

The **Program Counter (PC)** is designed with dual operational modes that enable both sequential execution and program flow control. Its functionality can be described as follows:

- **Increment Mode:** At timing state `T3`, when `pc_en = 1`, the counter updates as $PC \leftarrow PC + 1$, thereby supporting sequential instruction progression.
- **Jump Mode:** During the execution of a `JMP` instruction, at timing state `T4`, when `jump_en = 1`, the lower nibble of the Instruction Register (IR) is placed on the system bus and loaded into the PC. This allows program control to be transferred to the specified target address.
- **Bus Interface:** At timing state `T1`, when `pc_out = 1`, the current PC value is driven onto the system bus and loaded into the Memory Address Register (MAR), thereby initiating the instruction fetch cycle.

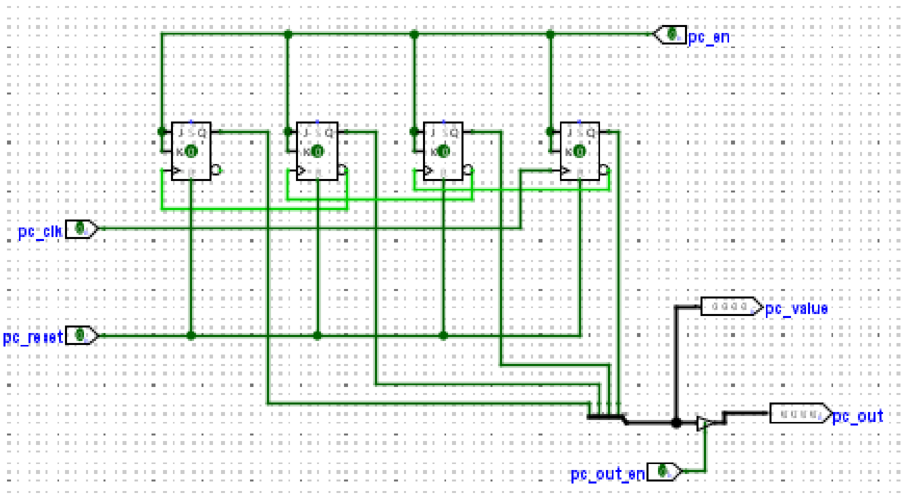


Figure 4: Program Counter (PC) circuit implementation using JK flip-flops, showing clock input, reset, enable control, and bus output interface (pc value, pc quencing.

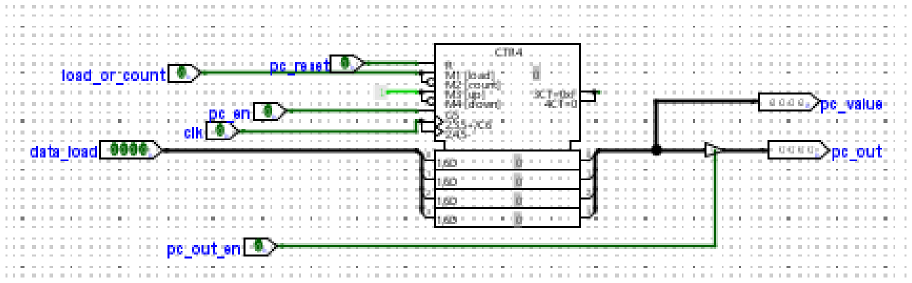


Figure 5: Program Counter implementation showcasing dual operational modes, including se quential increment and direct load functionality, to ensure comprehensive program flow control.

4.4 Memory System and Address Register

The memory subsystem includes a 4-bit Memory Address Register (MAR) which captures addresses from the system bus under the control of `mar_in_en`.

- **Instruction Fetching:** During the T1 phase, the signals `pc_out` and `mar_in_en` load the program counter value into the MAR, enabling instruction fetch.
- **Operand Addressing:** During the T4 phase of LDA, LDB, STA, or JMP instructions, `ins_reg_out_en` together with `mar_in_en` transfers the operand address (IR[3:0]) into the MAR.

The SRAM operates in two modes:

- **Read Mode:** When `sram_rd = 1`, the value stored at RAM[MAR] is placed on the bus during T2 for instruction fetch and during T5 for LDA and LDB instructions.
- **Write Mode:** When `sram_wr = 1`, the data on the bus is written into RAM[MAR] during the T5 phase of the STA instruction.

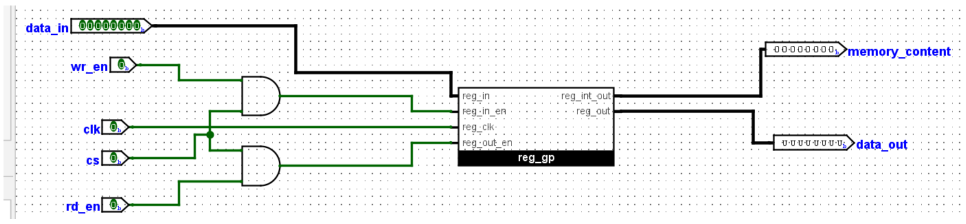


Figure 6: Register-based memory element showing data input (`data_in`), write enable (`wr_en`), read enable (`rd_en`), clock, and chip select (`cs`) control signals. The stored value is available as `memory_content`, while the output to the bus is provided through `data_out`.

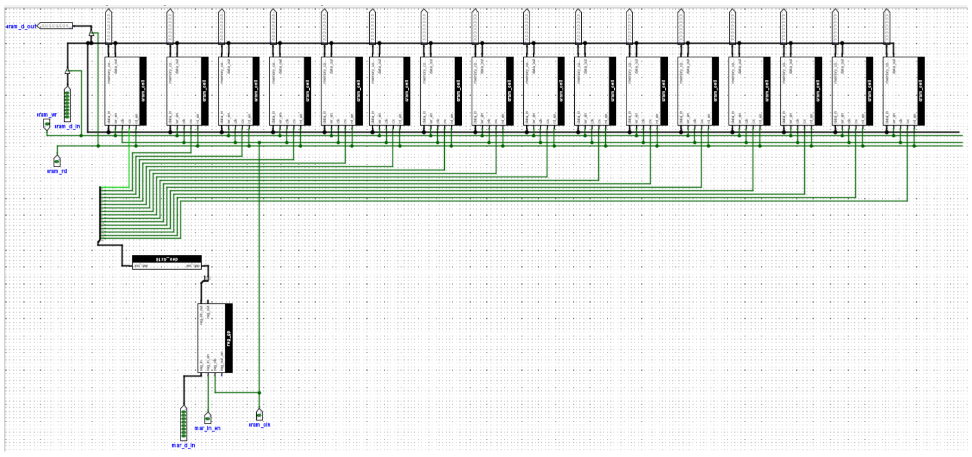


Figure 7: Memory subsystem showing MAR operation and SRAM interface with read-/write timing for reliable data access.

4.5 Instruction Register and Opcode Decoder

The Instruction Register (IR) is designed with a dual functional role, enabling both instruction storage and operand handling. Its operation can be described as follows:

[label=v.]

1. **Instruction Loading**: During the T2 phase, the signals `sram_rd=1` and `ins_reg_in_en=1` activate the transfer `IR` ← `M[MAR]`, thereby capturing the instruction from memory.

2. **Opcode Handling**: The upper nibble of the register, IR[7:4], is routed to the opcode decoder (`ins_tab`), which generates one-hot control signals corresponding to the decoded instruction.

3. **Operand Handling**: The lower nibble, IR[3:0], can be placed onto the system bus when `ins_reg_out_en=1` is asserted, typically during T4, to provide operand addresses or target values required by memory and jump instructions.

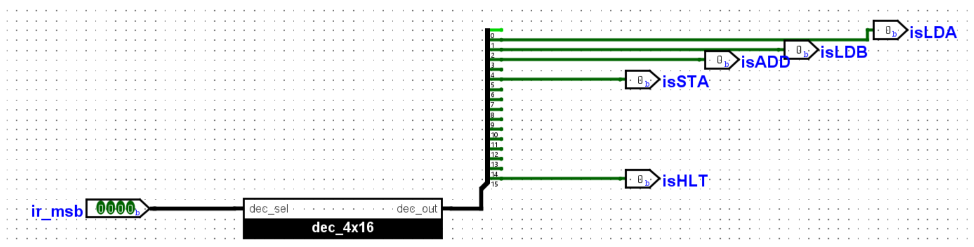


Figure 8: Architecture of the instruction register and opcode decoder, showing instruction loading, opcode routing to the decoder, and operand forwarding for execution.

The opcode decoder (`ins_tab`) implements a 4-to-16 decoding scheme, producing one-hot activation lines such as `insLDA, insLDB, insADD, insSUB, insSTA, insJMP, insHLT`, with additional unused outputs reserved for future instruction set expansion.

4.6 Arithmetic Logic Unit (ALU) 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.

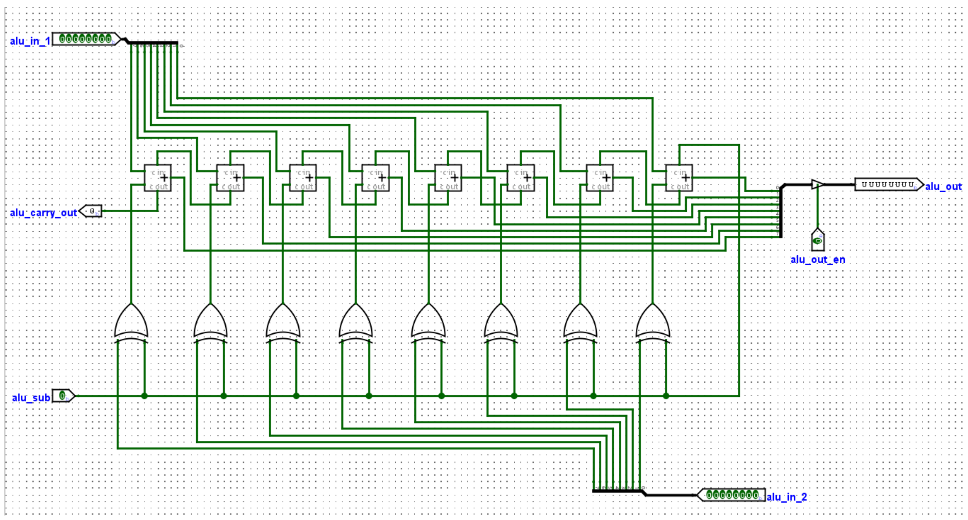


Figure 9: Arithmetic Logic Unit (ALU) implementation illustrating a ripple-carry architecture with a tri-state bus interface and structured operation control, enabling efficient and reliable 8-bit arithmetic processing.

4.7 Boot/Loader Counter and Phase Generation

The boot/loader subsystem (`ins_loader`) is responsible for secure transfer of program data from ROM to RAM during Manual/Loader mode. Its operation can be characterized through the following functions:

1. **Functional Role**: Provides safe program loading from ROM to RAM when the system is placed in Manual/Loader mode (`debug=1`), ensuring 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 ¬Φ), which synchronize data transfer and ensure contention-free operation.

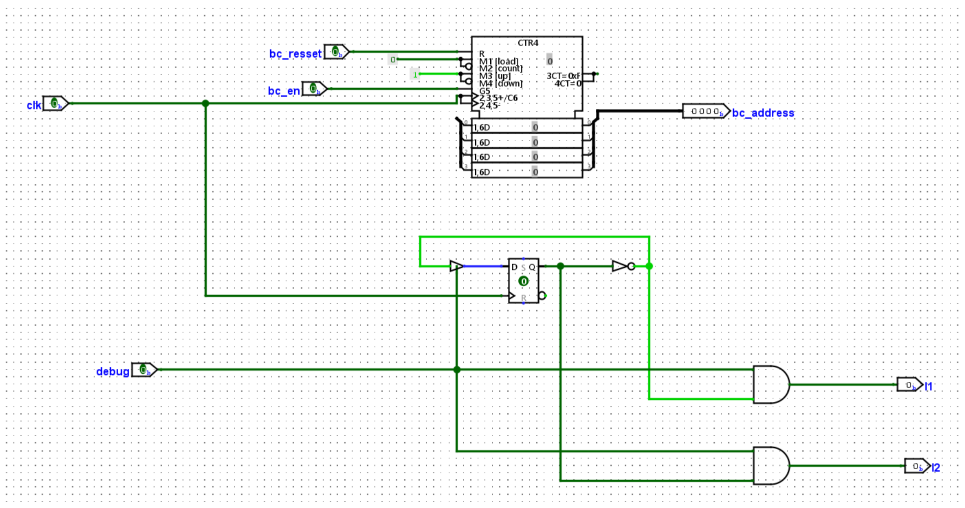


Figure 10: Architecture of the boot/loader subsystem, showing sequential address generation and dual-phase clocking for reliable ROM-to-RAM transfer in Manual/Loader mode.

5 Control System Design

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

- **Ring Counter (rc)** generating timing states `T1..T6`
- **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 paradigms corresponding to the system's **Manual Mode** and **Automatic Execution Mode**. Each operational mode applies distinct control pathways optimized for its intended functionality while maintaining strict signal integrity and timing synchronization.

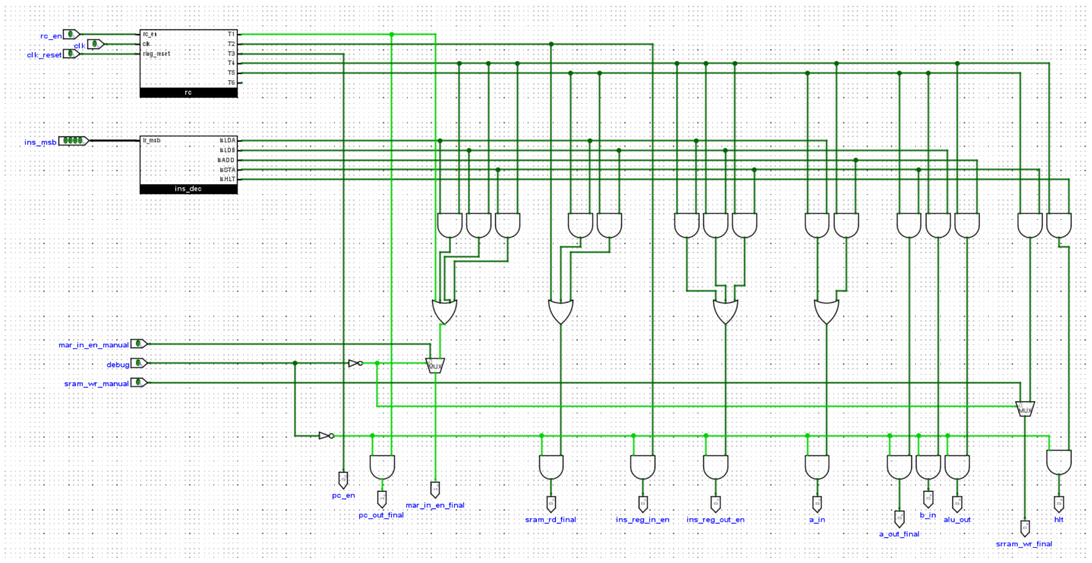


Figure 11: Manual mode control sequencer architecture supporting ADD instruction execution with basic timing coordination for step-wise validation.

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

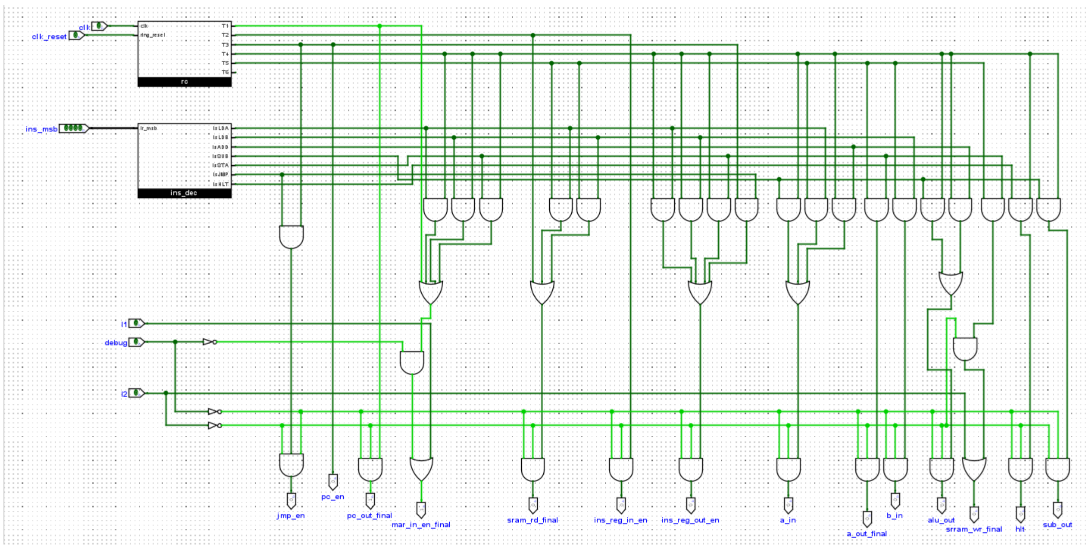


Figure 12: Automatic mode control sequencer demonstrating full instruction set support with ADD, SUB, and JMP execution pathways using hardwired control logic.

The **Automatic mode control sequencer** manages the complete fetch-decode-execute cycle and supports ADD, SUB, and JMP instructions. Instruction execution is orchestrated by combining ring counter timing states with opcode decoding logic, producing deterministic micro-operation sequences. This implementation provides comprehensive instruction execution capabilities with efficient bus utilization and precise timing coordination across all program phases.

5.1 Timing Control Generator

The timing subsystem employs a ring counter that generates six distinct phases (T1–T6), which collectively orchestrate the fetch–decode–execute cycle. This arrangement provides deterministic sequencing of micro-operations and ensures proper synchronization between datapath components.

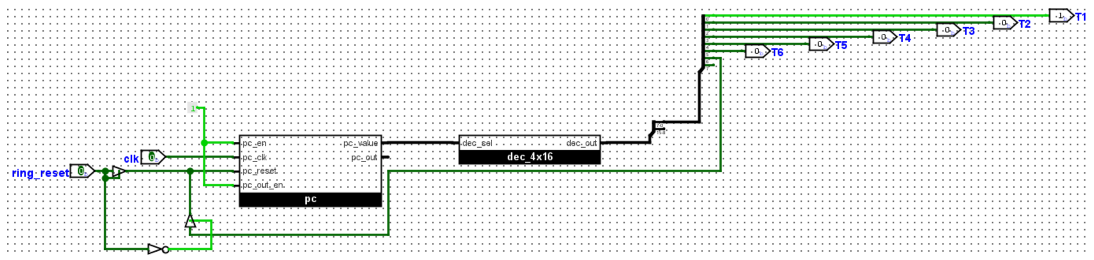


Figure 13: Six-phase ring counter implementation for timing control, enabling systematic fetch–decode–execute sequencing and precise micro-operation coordination.

The operation of the ring counter can be classified into two major categories: the universal fetch sequence, common to all instructions, and instruction-specific execution sequences.

Universal Fetch Sequence

For every instruction, the following sequence is applied:

1. **T1**: The program counter output (`pc_out`) is enabled and the memory address register (`mar_in_en`) captures its value, resulting in `MAR` ← `PC`.
2. **T2**: A memory read operation (`sram_rd`) is initiated and the instruction register input enable (`ins_reg_in_en`) is asserted, transferring `IR` ← `M[MAR]`.
3. **T3**: The program counter is incremented via `pc_en`, updating its value as `PC` ← `PC + 1`.

Representative Execute Sequences

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

1. **LDA addr**:
   - **T4**: The operand address (IR[3:0]) is placed on the bus (`ins_reg_out_en`) and loaded into the memory address register (`mar_in_en`), executing `MAR` ← `IR[3:0]`.
   - **T5**: The memory value is read (`sram_rd`) and latched into register A (`a_in`), performing `A` ← `M[MAR]`.
2. **ADD**: **T4**: The A and B register outputs (`a_out, b_out`) drive the ALU, whose output (`alu_out`) is fed back into register A (`a_in`) with subtraction disabled (`alu_sub=0`).

3. **SUB**: **T4**: Similar to ADD, except with subtraction enabled (`alu_sub=1`), performing `A` ← `A - B`.
4. **JMP addr**: **T4**: The operand (IR[3:0]) is placed onto the bus (`ins_reg_out_en`) and loaded into the program counter (`pc_en`), executing `PC` ← `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):

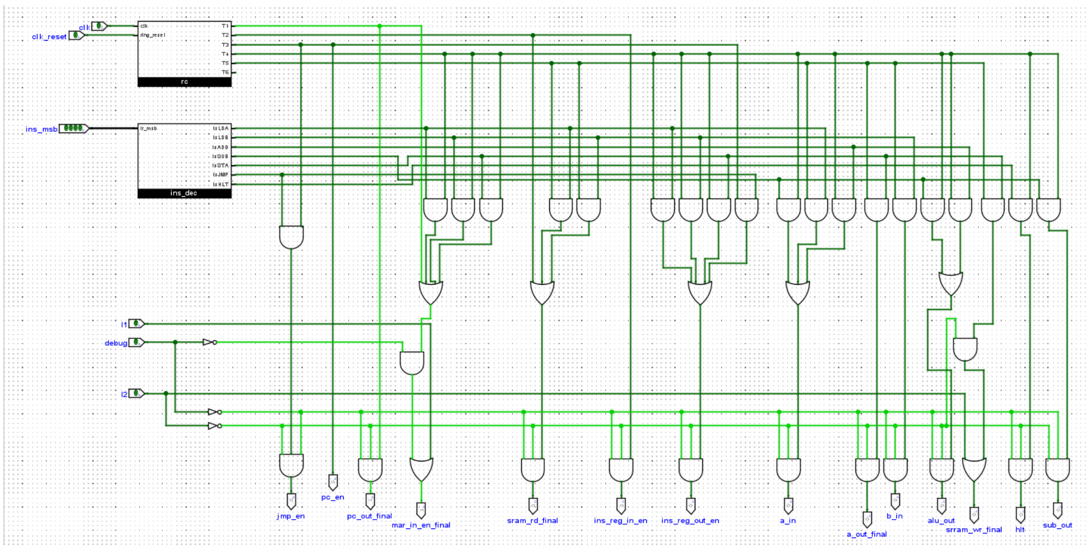


Figure 14: Gate-level control matrix demonstrating comprehensive control equation implementation with automatic mode logic and precise timing coordination for instruction execution sequencing.

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.

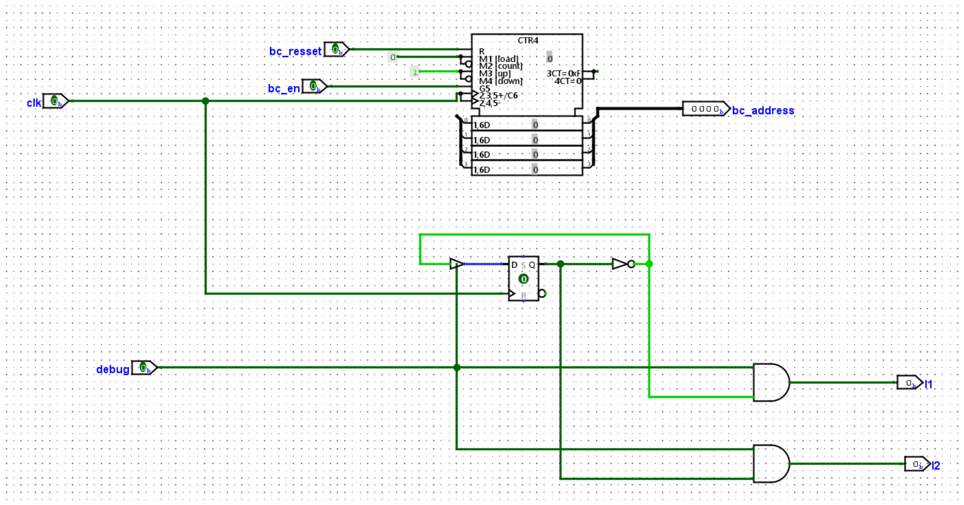


Figure 15: Architecture of the Manual/Loader control system, illustrating debug-based masking and handshake signaling for secure and conflict-free program loading.

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

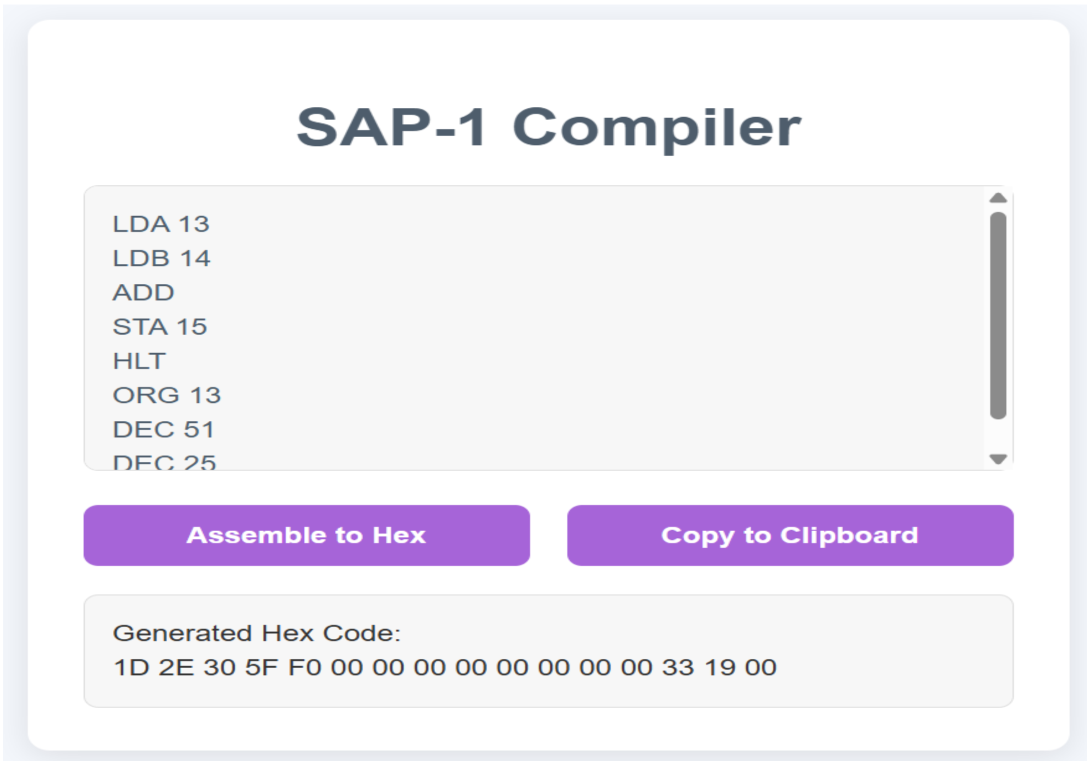
Address	Instruction	Hex	Mnemonic & Explanation
00000000	00011101	1D	LDA 13 (Load A from M[13])
00000001	00101110	2E	LDB 14 (Load B from M[14])
00000010	01100101	65	JMP 5 (PC $\leftarrow$ 5)
00000011	00110000	30	ADD (A $\leftarrow$ A + B)
00000100	01000000	40	SUB (A $\leftarrow$ A – B)
00000101	01011111	5F	STA 15 (M[15] $\leftarrow$ A)
00000110	11110000	F0	HLT (Stop execution)

Table 2: Data Values in RAM

Address (Binary)	Data (Binary)	Decimal	Hex
00001101	00101100	44	2C
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, and 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
ADD
STA 15
HLT
ORG 13
DEC 51
DEC 25
```

Generated Hex Code:

```
1D 2E 30 5F F0 00 00 00 00 00 00 00 00 33 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, LDA 14, ADD, STA 15, HLT, ORG 13, DEC 51, DEC 25	1D 2E 30 5F F0 00 00 00 00 00 00 00 00 33 19 00
JMP + ADD Program	LDA 13, LDA 14, JMP 5, ORG 5, ADD, STA 15, HLT, ORG 13, DEC 51, DEC 25	1D 2E 65 00 00 30 5F F0 00 00 00 00 00 33 19 00

7 Operation:

The CPU functions through a repetitive sequence of operations controlled by the system clock, commonly known as the *fetch–decode–execute cycle.* This process can be divided into three main stages:

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:
     (a) Set address: Use `debug_data` to define the 8-bit memory address.
     (b) Load address into MAR: Pulse `mar_in_en_manual`.
     (c) Set instruction/data: Use `debug_data` to provide the 8-bit value.
     (d) 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:** Check RAM address `00001111` (decimal 15). The expected content is `01000101` (decimal 69),obtained from adding decimal 44 and decimal 25.

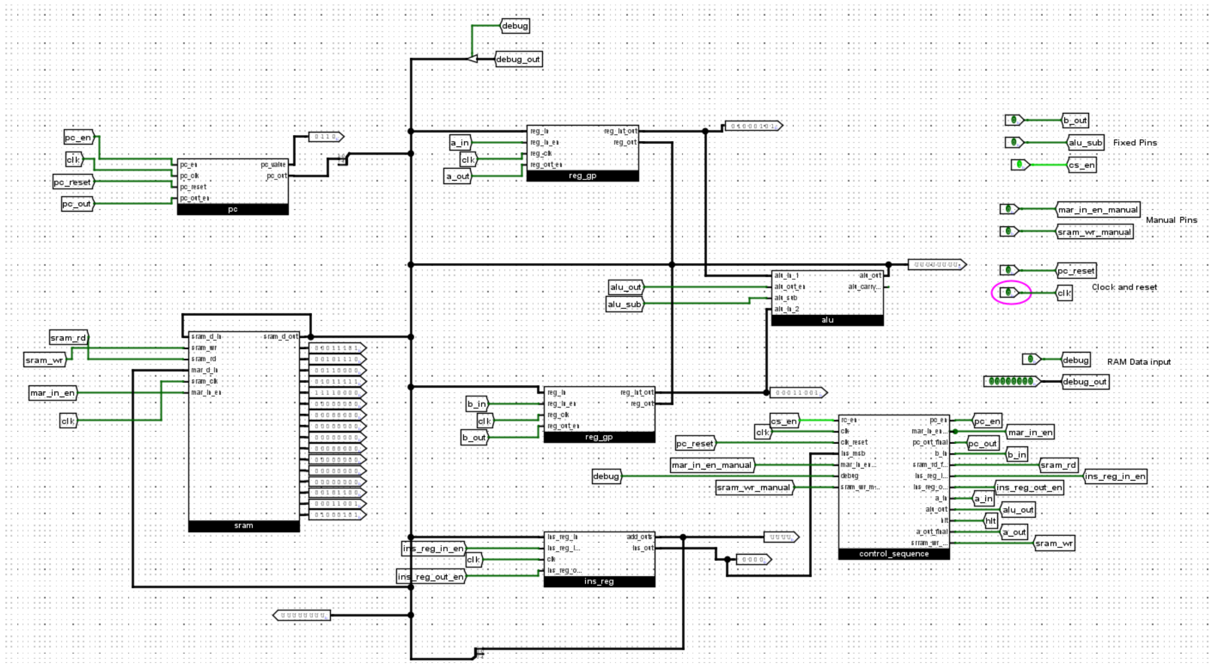


Figure 17: SAP-1 CPU circuit implementation in Logisim Evolution, highlighting debug signals, control pins, and RAM verification for program execution.

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

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

1. **Initial Setup**
   (a) Ensure the `debug` pin is set to LOW.
   (b) Ensure the main clock (`clk`) is OFF.
   (c) Pulse the `pc_reset` pin once to reset the Program Counter to `0000`.
2. **Program the ROM**
   (a) Right-click the ROM component and select `Edit Contents...`.
   (b) Enter the following hex code sequence into the ROM memory:

   ```
   1D 2E 65 00 00 30 5F 00 00 00 00 00 00 2C 19 00
   ```

   (c) Alternatively, upload the prepared `instruction_code_jmp_add` file (if provided).
3. **Load Program to RAM (Bootloader Mode)**
   (a) Set the `debug` pin to HIGH. The `Code Loading Mode` LED will turn ON.
   (b) With each `clk` pulse, the CPU will copy the program from ROM into RAM. Two clock pulses are required per instruction/data value.
   (c) Allow the CPU to complete writing all instructions and data into RAM.
   (d) Observe MAR and Data Bus activity on the 7-segment displays during this phase.
4. **Stop the Bootloader**
   (a) Set the `debug` pin back to LOW.
   (b) Pulse the main `clk` once to ensure the bootloader process stops fully.
5. **Run the Program**
   (a) Pulse `pc_reset` again to reset the Program Counter to `0000`.
   (b) Provide clock pulses (manual clicking or continuous clock) to let the CPU execute.
   (c) Observe PC, MAR, IR, Register A, and Register B values in the 7-segment displays through the Fetch–Decode–Execute cycle.
   (d) 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, stopping the CPU.

6. **Verify Result**
   (a) After execution, check RAM at address `1111` (decimal 15).
   (b) Expected result: Register A and RAM[15] contain the sum of `DEC 44 (0x2C)` and `DEC 25 (0x19)`, i.e., `0x45 (69 decimal)`.

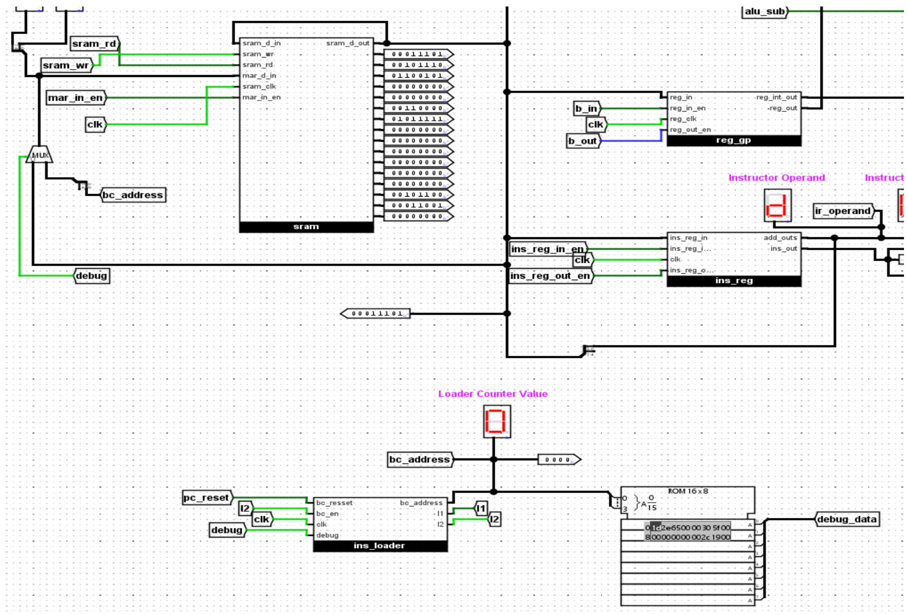


Figure 18: After loading all instruction and data values into RAM memory.

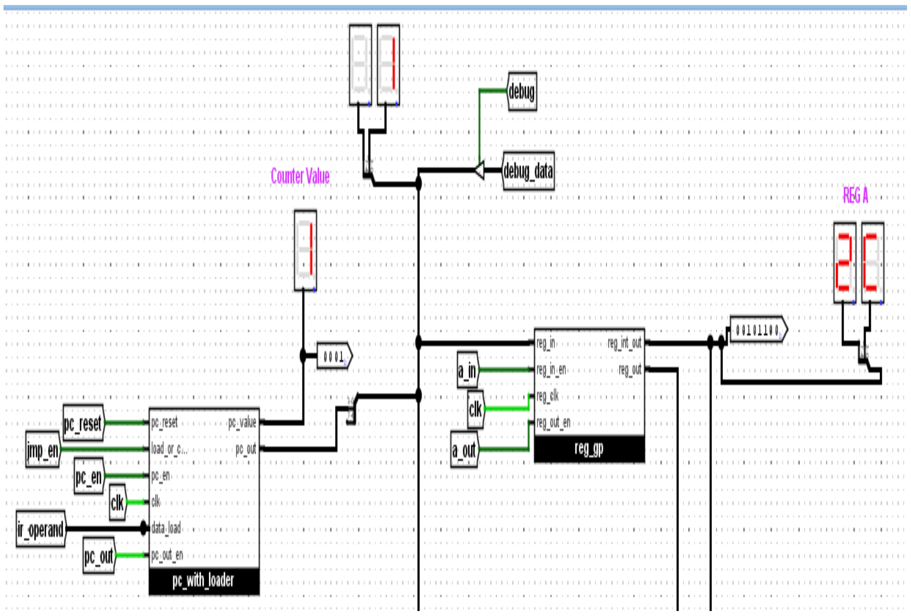


Figure 19: After executing LDA 13: the value 44 is loaded from memory address 13 into Register A.

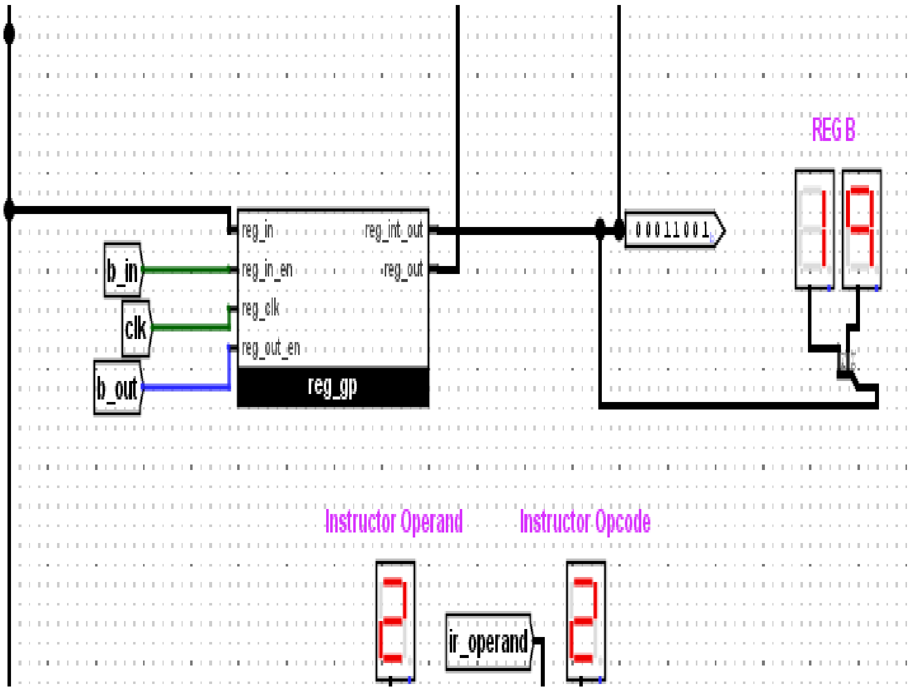


Figure 20: After executing LDA 14: the value 25 is loaded from memory address 14 into Register B.

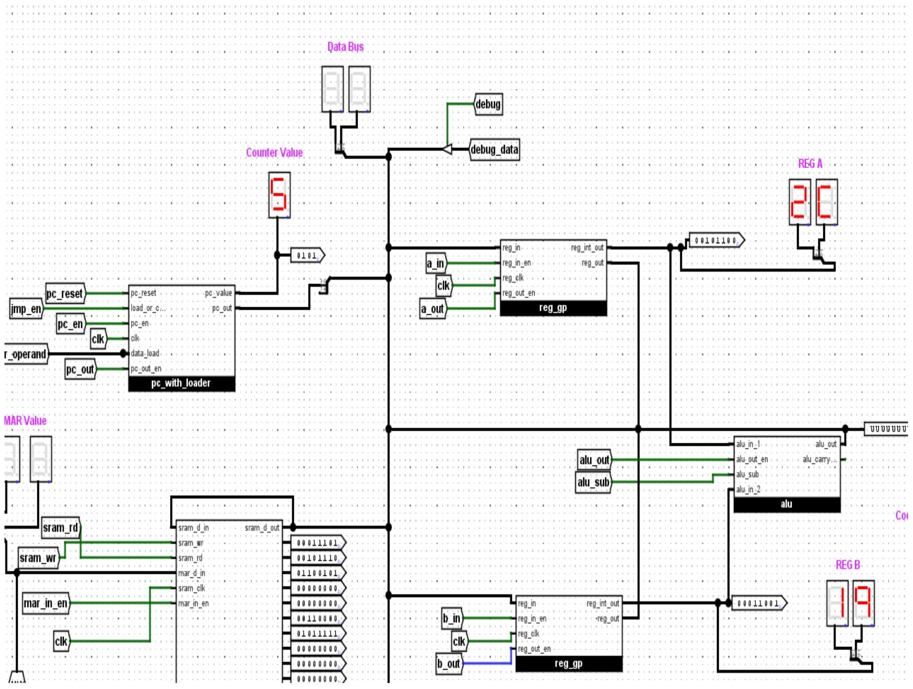


Figure 21: After executing JMP 5: the Program Counter is updated to address 5, redirecting the execution flow.

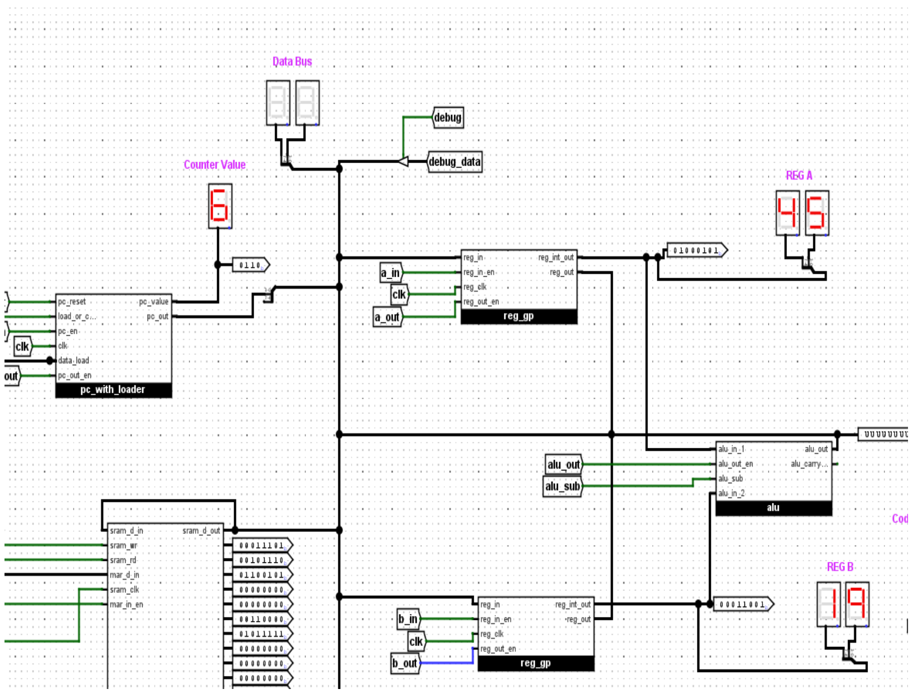


Figure 22: After executing ADD: the contents of Register A and Register B are added, and the result (69) is stored back into Register A.

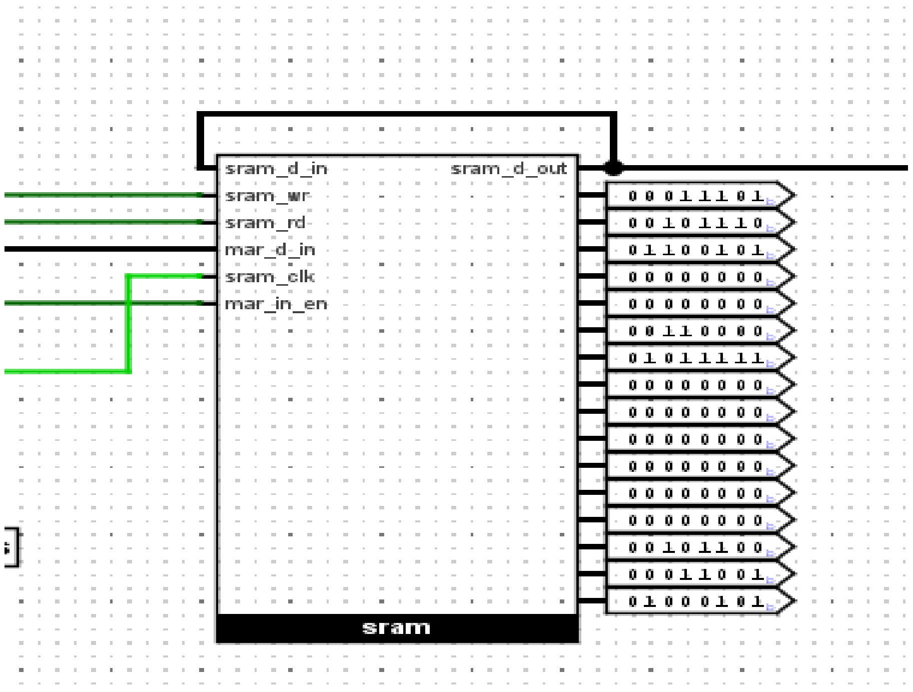


Figure 23: After executing STA 15: the result from Register A (69) is written into memory address 15.

8 Future Work

Potential directions for extending the present implementation include the integration of additional architectural and functional enhancements. Status flags such as the Zero (Z) and Carry (C) flags may be introduced to support more sophisticated control mechanisms. These flags would enable the design and execution of conditional branch instructions (e.g., JZ, JC), thereby extending the control-flow capabilities of the processor.

Further improvements may include support for multi-byte memory addressing and instruction formats, enabling immediate data loading and richer program structures. The adoption of a microcoded control unit could also facilitate systematic ISA (Instruction Set Architecture) expansion, simplifying the incorporation of new operations such as shift and rotate instructions.

Finally, development of an expanded assembler with symbolic labels, expressions, and enhanced directive support would improve programmability, usability, and experimental validation in instructional environments.

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.