SAP-1 Microprocessor Documentation Report

Manual & Automatic (Auto Loader) Implementation in Logisim

Project Name: SAP-1 CPU

Course: VLSI Technology Sessional
Course No: ETE 404

Name: Yeasir Mahmud
ID: 2008036
Electronics and Telecommunication Engineering
Chittagong University of Engineering and Technology

GitHub Repository:
`https://github.com/yeasirmahmud01/SAP_1_2008036`

Date: October 6, 2025

Contents

1 Project Overview

This project implements the Simple-As-Possible (SAP-1) microprocessor in Logisim with two versions:

- Manual Circuit (`sap1_2008036.circ`)
- Automatic Circuit with Instruction Loader (`sap1_2008036_auto.circ`)

The design includes all standard SAP-1 components, plus a RAM auto-loader and an extended instruction set.

2 Video Tutorials

Two demonstration videos have been prepared to illustrate the complete working of the SAP-1 Microprocessor project. The first video focuses on the **Manual Mode**, explaining every subcircuit in sequence, and the second video presents the **Automatic Mode**, featuring the Instruction Loader and Control Sequencer operation.

- **Video 1 – Manual SAP-1 Implementation and Working:**
  `https://youtu.be/Qg41prMnh7M?si=I5qifFfr43NMmPTR`
- **Video 2 – Automatic SAP-1 (Instruction Loader & Control Sequencer):**
  `https://youtu.be/Fg9NOlhEnMM?si=H7XgJqqebU`$_c U_0 Q$

Both videos demonstrate the detailed operation of the **Ring Counter**, **Instruction Decoder**, **Control Sequencer**, and **Instruction Loader**, which are the key functional modules designed in this project.

3 Features

- Complete SAP-1 implementation in Logisim (manual & auto).
- Hardwired control unit: ring counter + instruction decoder + sequencer.
- Python assembler to generate machine code for the RAM auto-loader.
- Extended instruction set: LDA, LDB, ADD, SUB, ST, JMP, CMP, OUT, HLT.

4 Circuits

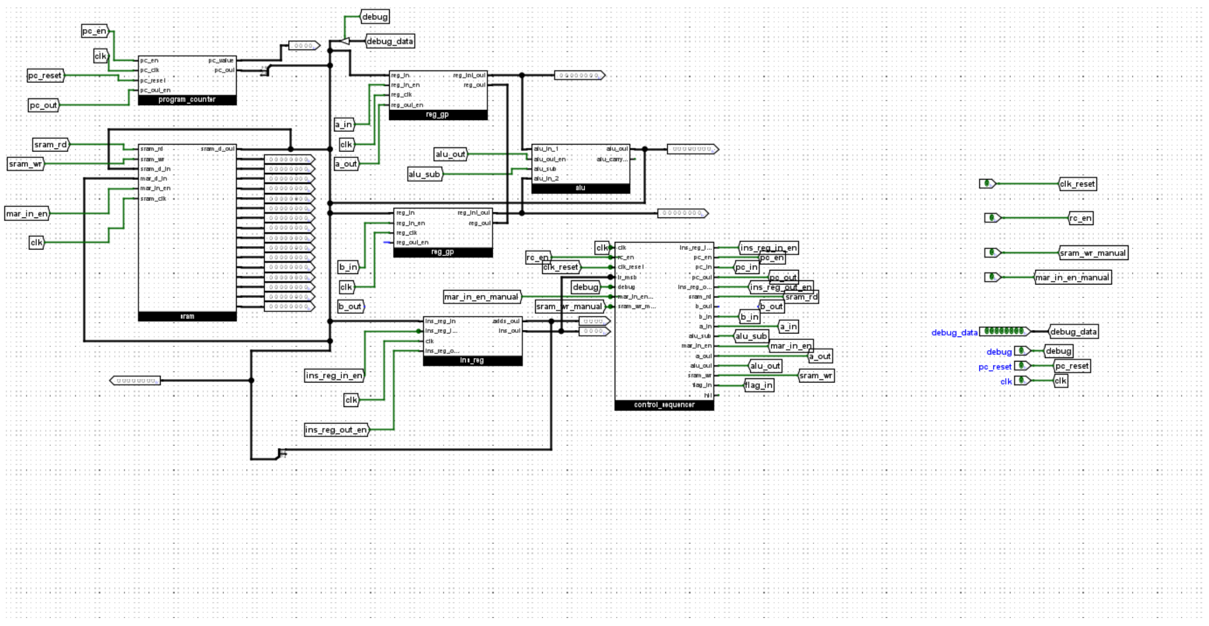


Figure 1: Manual SAP-1 top-level circuit.

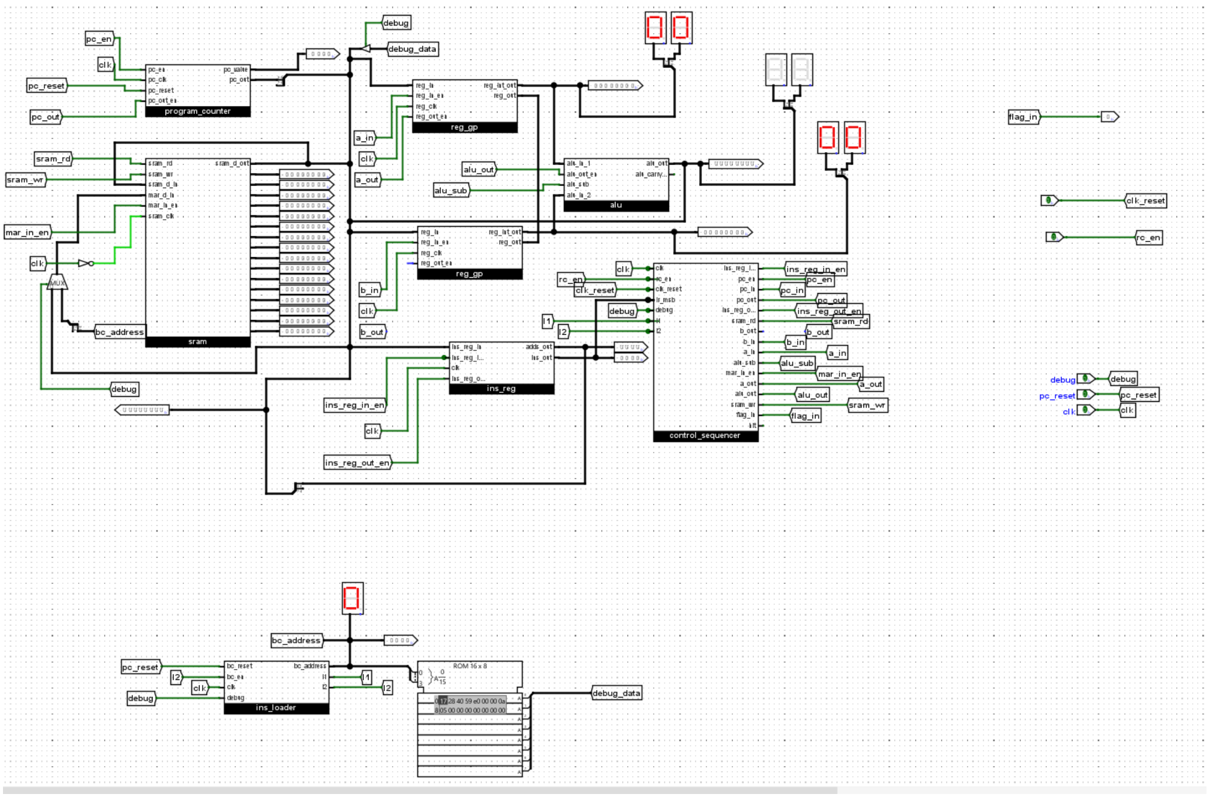


Figure 2: Automatic SAP-1 top-level circuit with instruction loader.

5 Architecture Components

The SAP-1 CPU is composed of several fundamental building blocks. Each block performs a specific function within the fetch–decode–execute cycle. The following subsections describe each component in detail.

5.1 Accumulator / General Purpose Register (reg_gp)

The accumulator is an 8-bit register used for arithmetic and logic operations. Data can be loaded from the bus into register A or B using `a_in`/`b_in`. The accumulator (A) can also drive the bus with `a_out`. Register B serves as a temporary operand for the ALU.

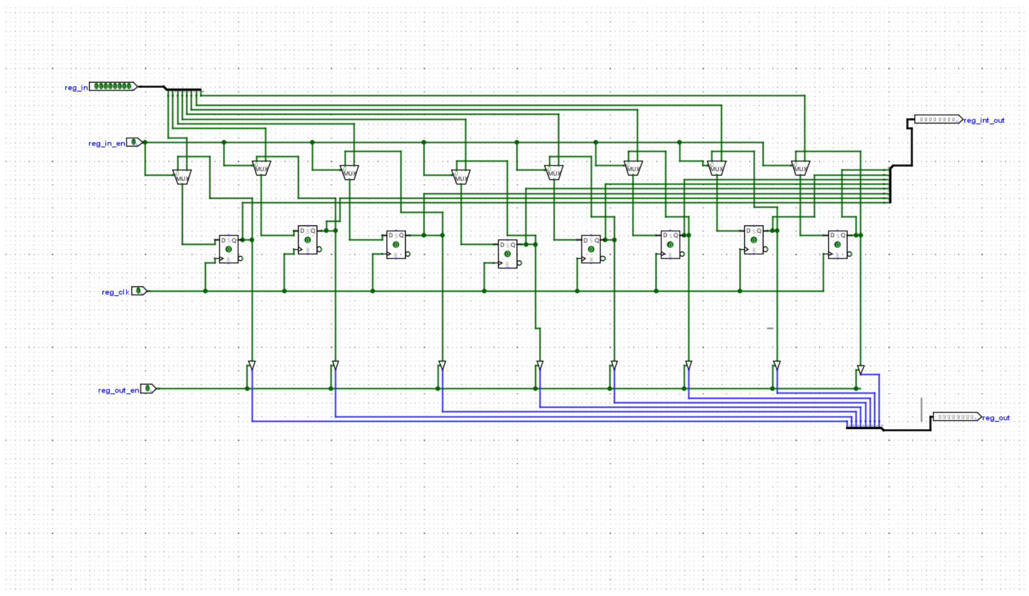


Figure 3: Accumulator and general-purpose register circuit.

5.2 Arithmetic Logic Unit (ALU)

The ALU performs arithmetic operations such as addition and subtraction. It receives inputs from registers A and B and outputs results via `alu_out`. The subtraction mode is enabled with the `alu_sub` control. Results are typically written back into register A.

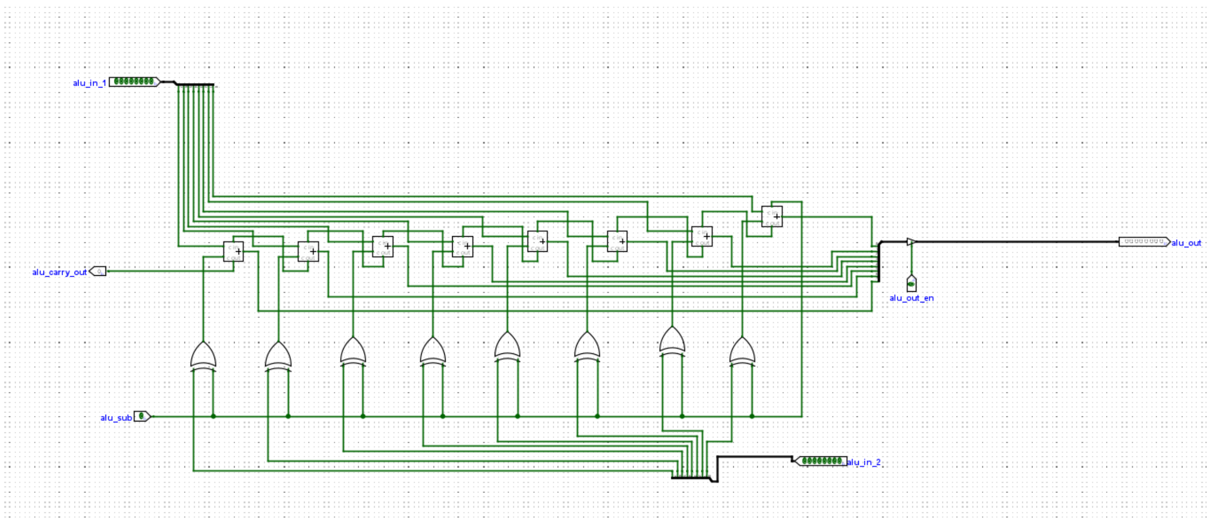


Figure 4: Arithmetic Logic Unit performing add/sub operations.

5.3 Program Counter (PC)

The Program Counter is a 4-bit counter that stores the memory address of the next instruction. It increments automatically after each fetch cycle. It can also be directly loaded during a JMP instruction.

Inputs/Outputs:

- `pc_clk` — system clock input.
- `pc_reset` — resets PC to 0000.
- `pc_en` — enables incrementing.
- `pc_out_en` — places the current address on the bus.

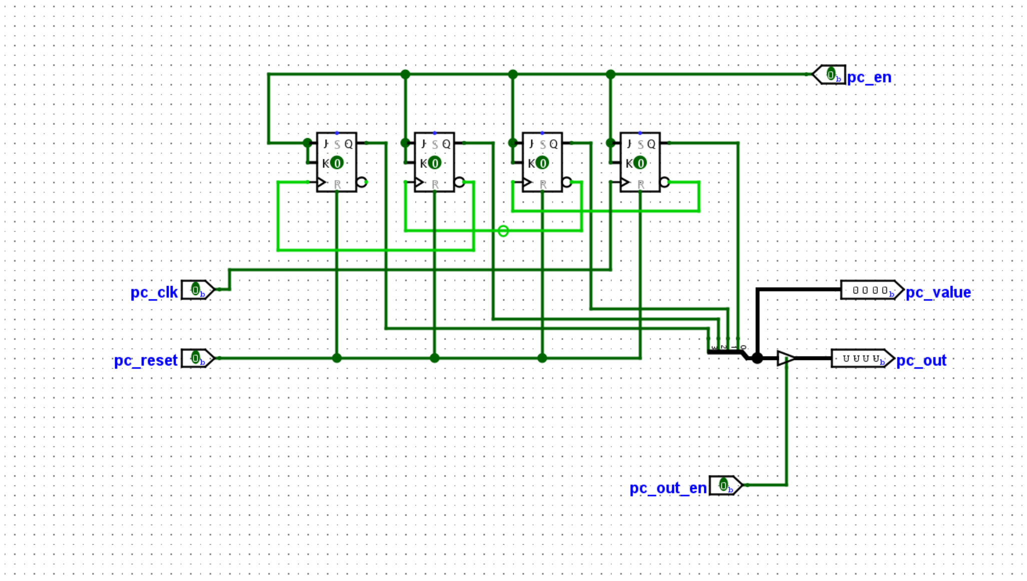


Figure 5: Program Counter circuit.

5.4 Decoder

The `dec` subcircuit implements a 4-to-16 binary decoder. It takes a 4-bit input (`dec_sel`) and produces a one-hot output on the 16 lines of `dec_out`. Only one output line is high at any time, corresponding to the binary value of the input.

This decoder is used as a fundamental block inside the instruction decoder and for other control gating operations. By converting compact binary values into one-hot signals, it allows the control sequencer to directly activate individual instruction or timing lines.

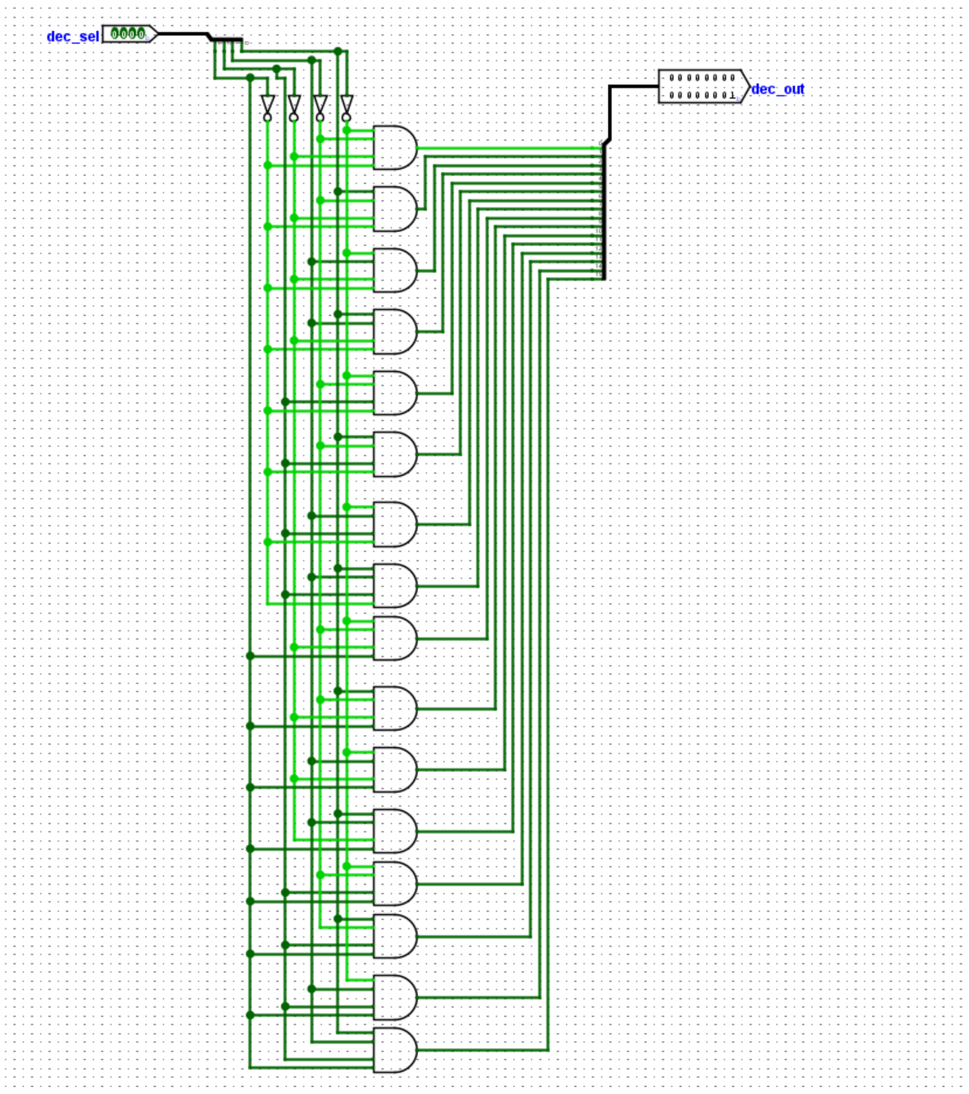


Figure 6: 4-to-16 decoder subcircuit (`dec`).

5.5 SRAM Cell

The `sram_cell` subcircuit is the fundamental building block of the memory. It implements a single-bit memory cell with read and write enable lines. Each cell is capable of storing one bit of data, which can be accessed when the appropriate word line and control signals are active.

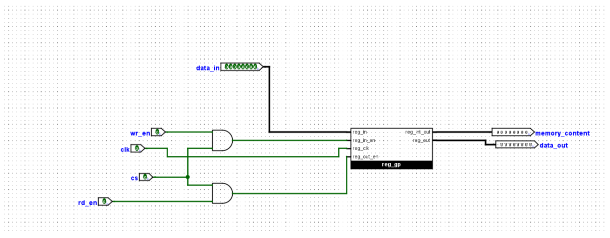


Figure 7: Single-bit SRAM cell used as the base memory element.

5.6 SRAM Block

The `sram` subcircuit integrates multiple `sram_cell` units into a memory array. It stores both instructions and data for the SAP-1. Access is controlled through the `sram_rd` (read enable) and `sram_wr` (write enable) signals. Because the SRAM design is large in height, it is shown in two parts for clarity.

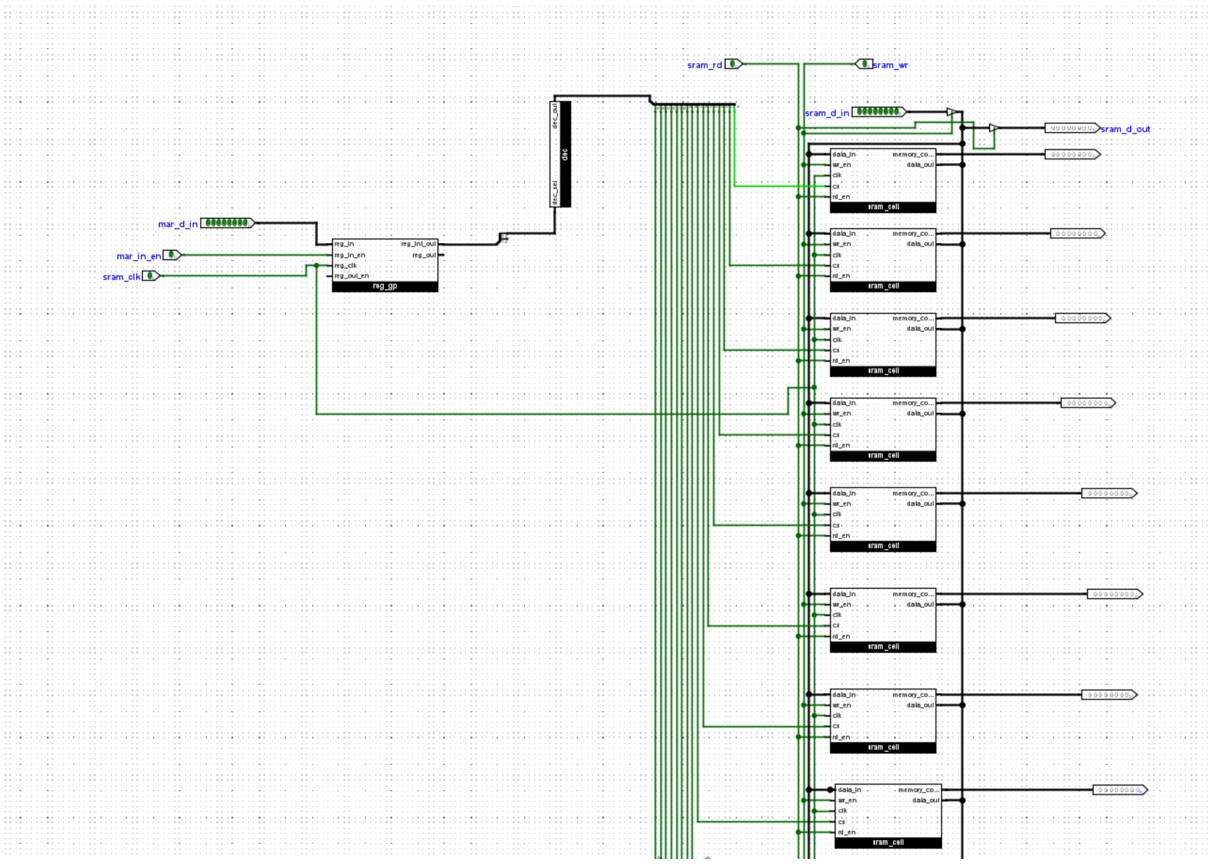


Figure 8: SRAM block (top half) showing address decoding and word line selection.

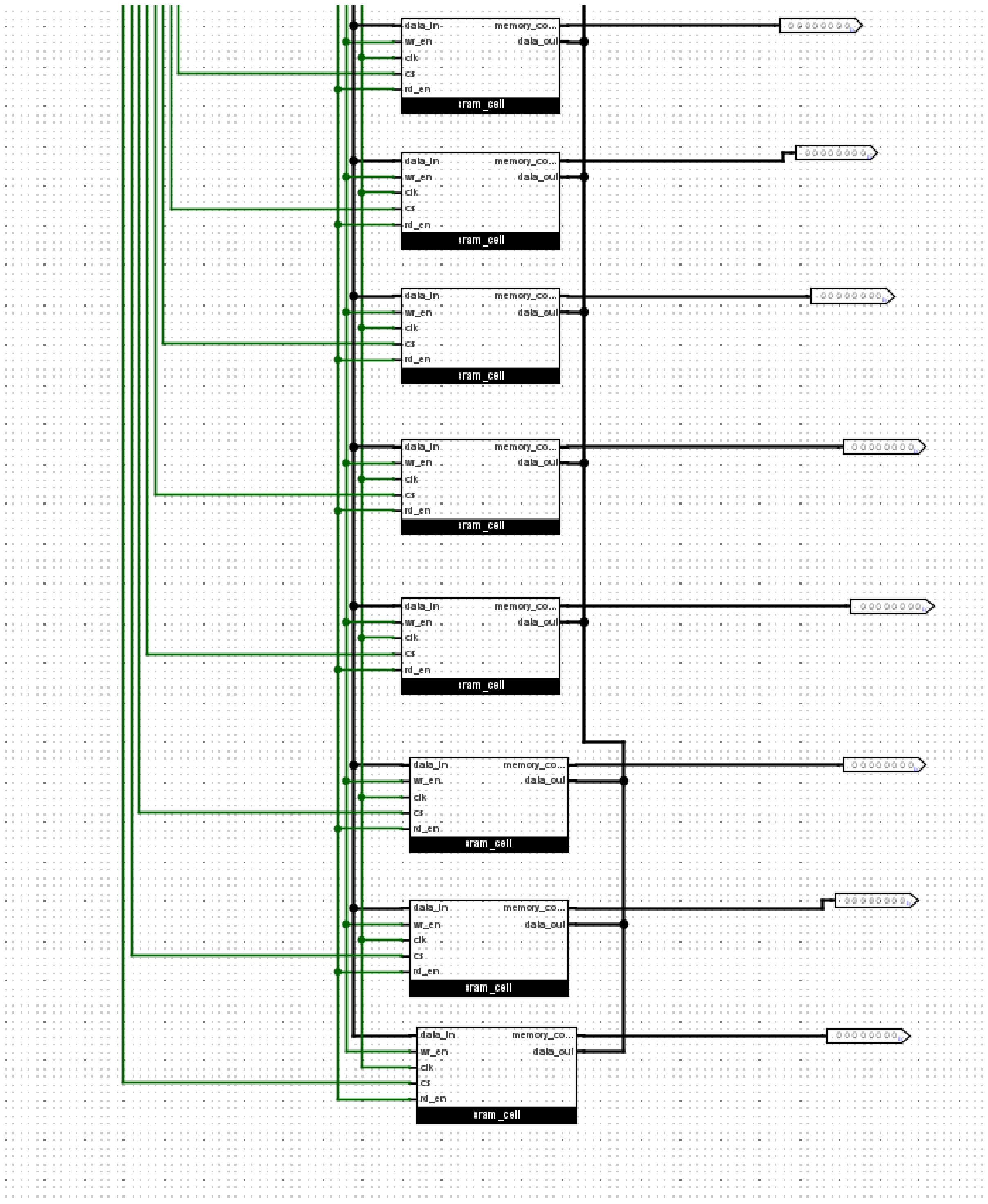


Figure 9: SRAM block (bottom half) showing data storage cells and I/O control.

5.7 Instruction Register (IR)

The IR holds the current instruction fetched from memory. The higher nibble (MSB) is decoded to identify the opcode, while the lower nibble (LSB) provides the operand address.

Controls:

- `ins_reg_in_en` — enables loading the instruction from bus.
- `ins_reg_out_en` — places operand address on the bus.

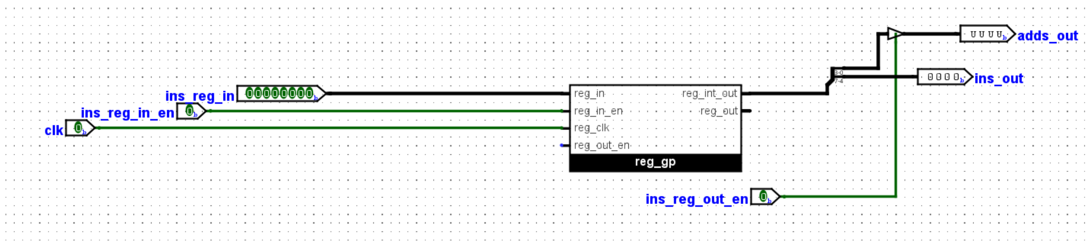


Figure 10: Instruction Register circuit.

5.8 Instruction Decoder (ins_dec)

The instruction decoder is a 4-to-16 line decoder. It converts the 4-bit opcode (from the instruction register) into one-hot control signals. Each active line corresponds to a specific instruction, which drives the control sequencer.

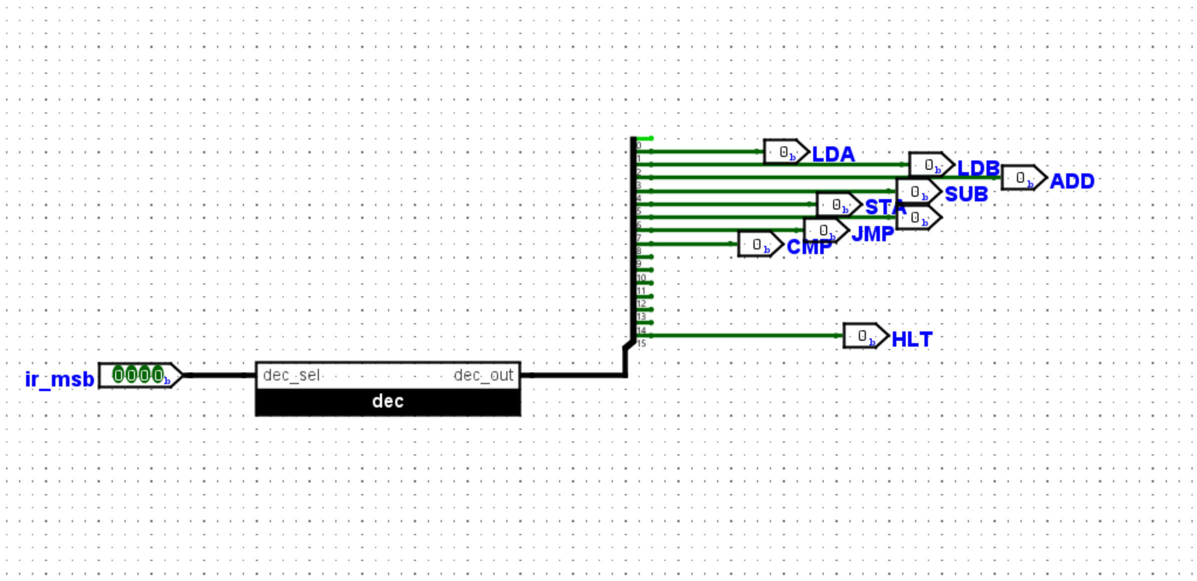


Figure 11: Instruction Decoder circuit (opcode to one-hot).

The mapping between 4-bit opcodes and the decoded instruction signals is shown in Table 1.

Table 1: Opcode to Instruction Mapping (Instruction Decoder Output)

Opcode (Binary)	**Opcode (Hex)**	**Instruction**
0000	0x0	LDA (Load Accumulator)
0001	0x1	LDB (Load B Register)
0010	0x2	ADD (A $\leftarrow$ A + B)
0011	0x3	SUB (A $\leftarrow$ A - B)
0100	0x4	ST (Store Accumulator to memory)
0110	0x6	JMP (Jump to operand address)
0111	0x7	CMP (Compare A and B $\rightarrow$ Flags)
1110	0xE	OUT (Output A)
1111	0xF	HLT (Halt execution)
Others	—	Not used / Reserved

5.9 Ring Counter

The ring counter generates timing signals T_1 through T_6 required for sequencing fetch, decode, and execute phases. It is implemented with D flip-flops connected in a ring configuration.

Controls:

- `rc_en` — enables counter progression.
- `rc_reset` — resets counter back to T_6.

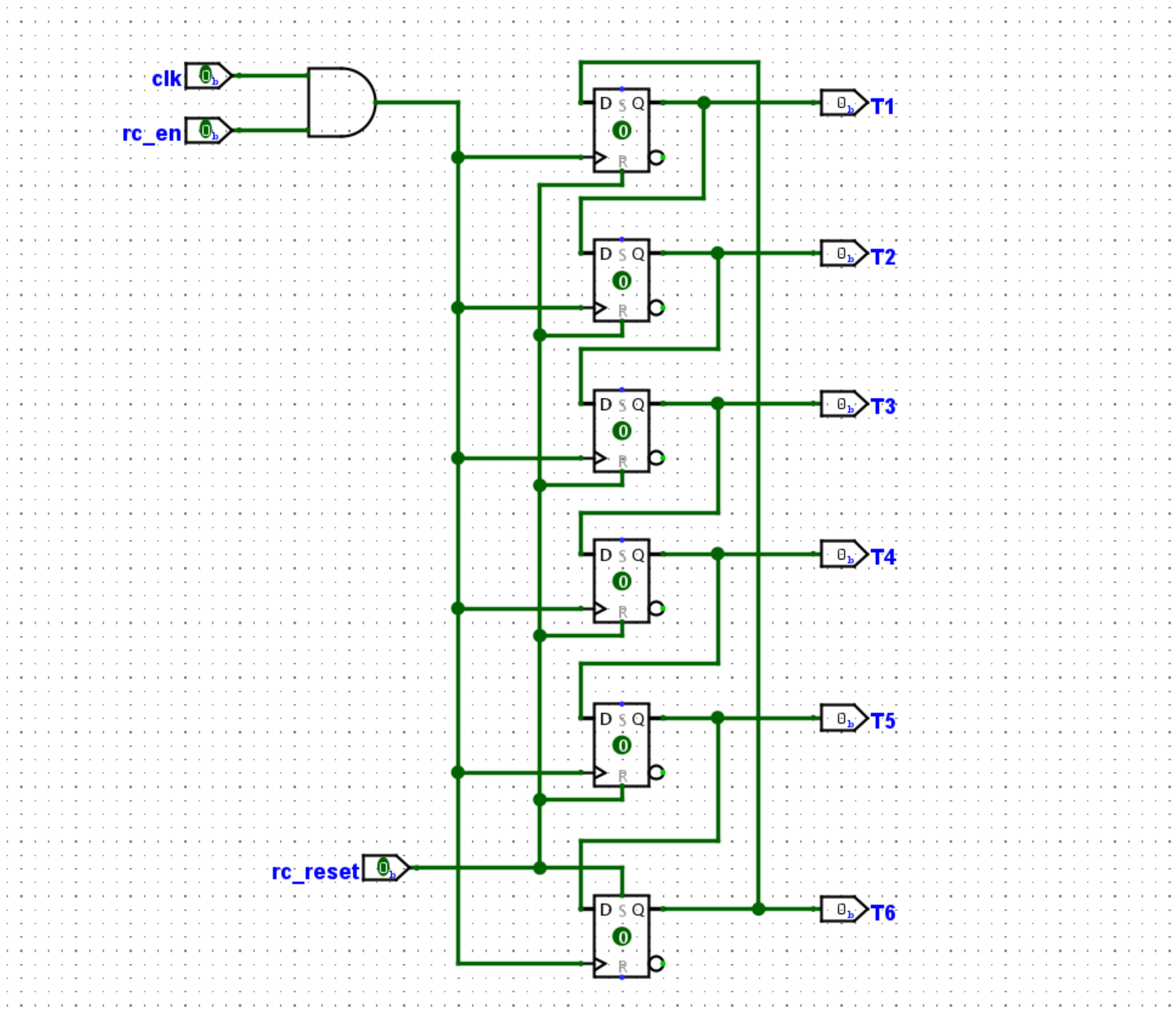


Figure 12: Ring Counter generating timing states T_1–T_6.

5.10 Control Sequencer (Manual Version)

The manual control sequencer combines timing states from the ring counter and opcode signals from the decoder to generate the register enables, memory controls, and ALU functions. It follows the original SAP-1 hardwired model, where all instruction execution is driven directly by the $T_1 \ldots T_6$ pulses.

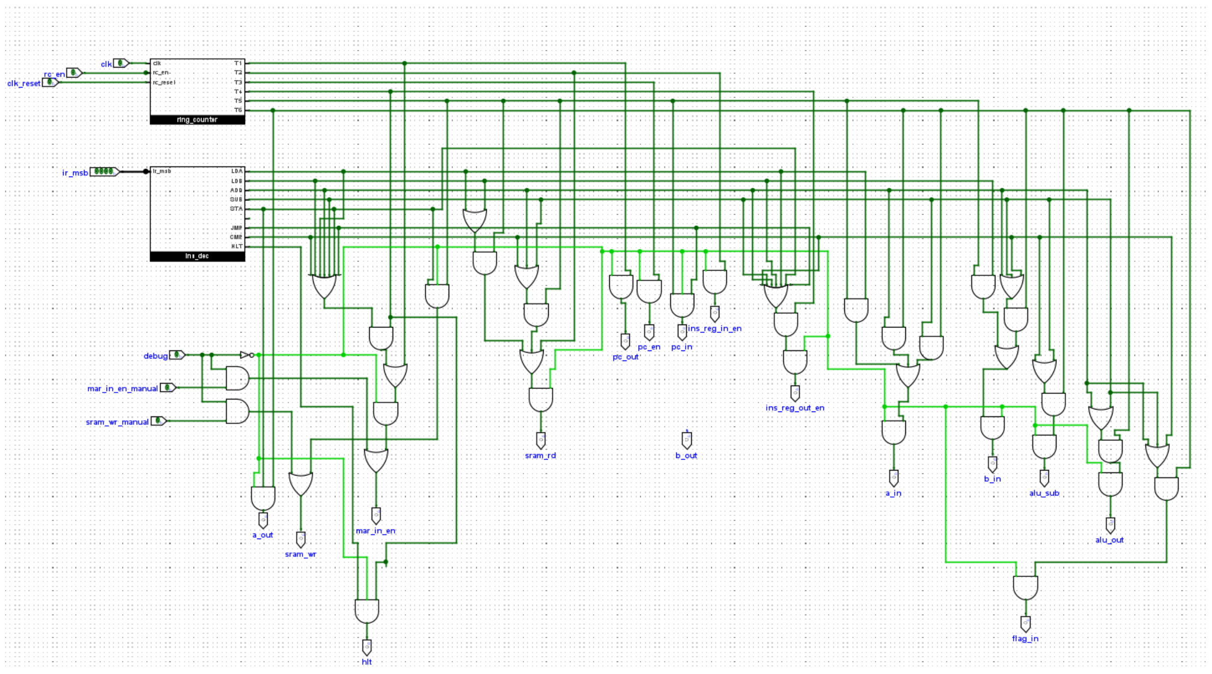


Figure 13: Manual control sequencer (hardwired gating of decoder + T-states).

In this version, the CPU operates only in manual mode. Instructions must be loaded into RAM beforehand (through the memory editor), since the control logic does not include auto-loading features.

5.11 Control Sequencer (Automatic Version)

The automatic control sequencer is an extended version of the manual design. In addition to the timing states ($T_1 \dots T_6$) and opcode decoder signals, it also incorporates signals from the instruction loader: I_1, I_2, and `debug`. These additional inputs ensure that while the loader is active, the CPU control signals remain disabled, preventing conflicts on the bus. Once the loader completes copying the program from ROM into SRAM, control is handed over to the CPU sequencer automatically.

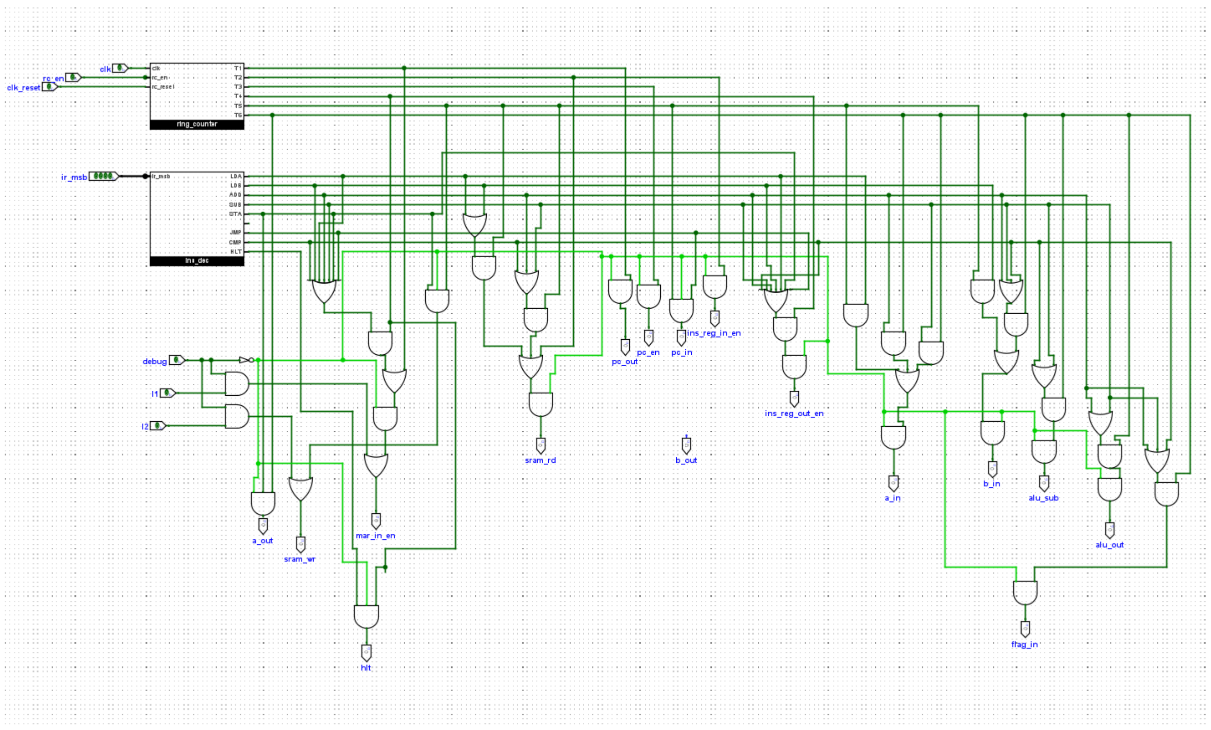


Figure 14: Automatic control sequencer with additional gating for loader handshakes.

Table 2: SAP-1 Control Sequencer Outputs with Debug/Manual Logic

Output	**Logic Expression**	**Description**
mar_in_en	(debug · mar_in_en_manual) + ($\overline{\text{debug}}$ · (T1·PC_OUT) + (T4·(LDA + LDB + ADD + SUB + STA + CMP + JMP)))	Enables MAR to latch address from PC (T1) or IR operand (T4); manual override via debug
sram_wr	(debug · sram_wr_manual) + ($\overline{\text{debug}}$ · (T5·STA))	Enables SRAM write for STA instruction; manual write if debug
sram_rd	$\overline{\text{debug}}$ · ((T2·(LDA + LDB + ADD + SUB + CMP)) + (T5·(LDA + LDB + ADD + SUB + CMP)))	Read memory during fetch/execute in auto mode
pc_out	$\overline{\text{debug}}$ · T1	PC outputs address during fetch
pc_en	$\overline{\text{debug}}$ · T3	PC increments after fetch
ins_reg_in_en	$\overline{\text{debug}}$ · T2	IR latches opcode + operand during fetch
ins_reg_out_en	$\overline{\text{debug}}$ · (T4·(LDA + LDB + ADD + SUB + STA + CMP + JMP))	IR outputs operand address to bus during execution
a_in	$\overline{\text{debug}}$ · ((LDA·T5) + (ADD·T6) + (SUB·T6))	Register A loads memory (LDA) or ALU result (ADD/SUB)
b_in	$\overline{\text{debug}}$ · ((LDB·T5) + (ADD·T5) + (SUB·T5) + (CMP·T5))	Register B loads memory operand
a_out	$\overline{\text{debug}}$ · ((T5·(ADD + SUB + CMP)) + (T4·(other instructions if needed)))	Register A drives bus during ALU operation
b_out	$\overline{\text{debug}}$ · (T5·(ADD + SUB + CMP))	Register B drives bus as ALU operand
alu_out	$\overline{\text{debug}}$ · (T6·(ADD + SUB))	ALU result drives bus during write-back
alu_sub	$\overline{\text{debug}}$ · (T6·(SUB + CMP))	Select subtract mode in ALU
flag_in	$\overline{\text{debug}}$ · (T6·(ADD + SUB + CMP))	Update flags after ALU operation
hlt	$\overline{\text{debug}}$ · HLT	Stops execution when HLT instruction is reached; inactive in manual mode

Logical Equations (sum of products with debug/manual).

Summary. Thus, the difference between manual and automatic control sequencing lies in the additional gating with `debug`, I_1, and I_2 signals, which synchronize the instruction loader with the CPU execution pipeline.

6 Instruction Set and Example

6.1 Instruction Set

Mnemonic	Opcode	Function
LDA	0000	Load Accumulator (A) from memory
LDB	0001	Load B register from memory
ADD	0010	A ← A + B
SUB	0011	A ← A - B
ST	0100	Store A to memory
JMP	0110	PC ← operand (jump)
CMP	0111	Compare A,B → flags
OUT	1110	Output A
HLT	1111	Halt

6.2 Example Program

The following example program loads two numbers from memory, adds them, and stores the result.

Table 3: Example Program: Add Two Numbers

Address	**Assembly**	**Binary**	**Hex**	**Explanation**
0	LDA 7	0000 0111	0x07	Load the value from memory location 7 into the Accumulator (A).
1	LDB 8	0001 1000	0x18	Load the value from memory location 8 into Register B.
2	ADD	0010 0000	0x20	Add contents of A and B; result stored back into A.
3	ST 9	0100 1001	0x49	Store contents of A into memory location 9.
4	HLT	1111 0000	0xF0	Halt program execution.
7	DATA 7 = 10	0000 1010	0x0A	Data constant: decimal 10 stored at memory location 7.
8	DATA 8 = 5	0000 0101	0x05	Data constant: decimal 5 stored at memory location 8.
9	DATA 9 = 0	0000 0000	0x00	Storage location initialized to 0; result (15) will be written here.

7 Assembler / Compiler

A Python-based assembler was developed using `Streamlit` to automatically convert SAP-1 assembly programs into binary machine code and Logisim ROM formats. This eliminates the need for manual binary conversion and allows programs to be directly loaded into the automatic SAP-1 via the instruction loader.

7.1 Compiler Source Code

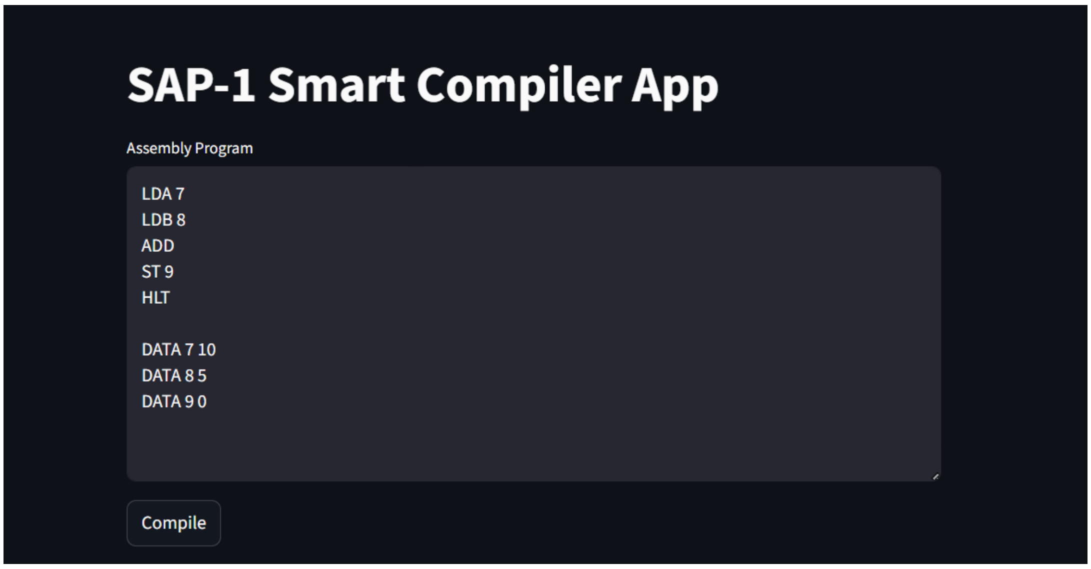
The assembler is implemented in Python. The code defines the instruction set architecture (ISA), parses mnemonics, and outputs binary, hex, and Logisim ROM paste formats.

Listing 1: SAP-1 Compiler Source Code

```
# sap_1_compiler.py
# SAP-1 Smart Compiler App using Streamlit

import streamlit as st

# ---------- ISA (aligned with your decoder) ----------
ISA = {
    "LDA": (0x1, True),    # 0001
    "LDB": (0x2, True),    # 0010
    "ADD": (0x3, False),   # 0011
    "SUB": (0x4, False),   # 0100
    "ST" : (0x5, True),    # 0101
    "CMP": (0x6, False),   # 0110
    "JMP": (0x7, True),    # 0111
    "HLT": (0xE, False),   # 1110
}

def bin8(n): return format(n & 0xFF, "08b")
def hex2(n): return format(n & 0xFF, "02X")

def rom_to_logisim_hex(rom):
    """Convert␣ROM␣into␣Logisim␣grid␣format␣(2␣rows␣ ␣8␣values)."""
    out = []
    for i in range(0, len(rom), 8):
        chunk = rom[i:i+8]
        out.append("␣".join(f"{v:02X}" for v in chunk))  # force 2-
            digit hex
    return "\n".join(out)

def assemble(src: str):
    rom = [0] * 16   # fixed 16x8 ROM
    pc = 0
    listing = []

    for line in src.splitlines():
        line = line.strip()
        if not line:
            continue

        parts = line.split()
        mnem = parts[0].upper()

        # ---------- Handle DATA ----------
        if mnem == "DATA":
            if len(parts) < 3:
                continue
            addr, val = int(parts[1]), int(parts[2])
            if not (0 <= addr < len(rom)):
                raise ValueError(f"DATA␣address␣{addr}␣out␣of␣range␣
                   (0-{len(rom)-1})")
            rom[addr] = val & 0xFF
            listing.append([f"{addr:04b}", bin8(val), f"DATA␣{addr}␣{
               val}"])
            continue

        # ---------- Handle ISA ----------
        if mnem in ISA:
            opcode, has_operand = ISA[mnem]
            imm = 0
            if has_operand and len(parts) > 1 and parts[1].isdigit():
                imm = int(parts[1])

            # force no-operand instructions to 0000
            if mnem in ["ADD", "SUB", "CMP", "HLT"]:
                imm = 0

            code = (opcode << 4) | (imm & 0xF)
            if pc >= len(rom):
                raise ValueError("Program␣too␣large␣for␣16x8␣ROM")
            rom[pc] = code
            listing.append([f"{pc:04b}", bin8(code), f"{mnem}␣{imm␣if␣
               has_operand␣else␣''}"])
            pc = (pc + 1) & 0xF
            continue

        # ---------- Ignore junk ----------
        continue

    return listing, rom

# ---------- Streamlit UI ----------
st.title("SAP-1␣Smart␣Compiler␣App")

default_src = """LDA␣7
LDB␣8
ADD
ST␣9
HLT

DATA␣7␣10
DATA␣8␣5
DATA␣9␣0
"""

src = st.text_area("Assembly␣Program", value=default_src, height=300)

if st.button("Compile"):
    try:
        listing, rom = assemble(src)

        st.subheader("Assembler␣Listing")
        st.table(listing)

        st.subheader("ROM␣Contents␣(16x8,␣linear)")
        st.text("␣".join(f"{v:02X}" for v in rom))   # fixed 2-digit
           hex

        st.subheader("Logisim␣ROM␣Paste␣Format")
        logisim_hex = rom_to_logisim_hex(rom)
        st.text(logisim_hex)

        # Download buttons
        st.download_button("Download␣binary", data="\n".join(bin8(v)
            for v in rom), file_name="binary.txt")
        st.download_button("Download␣hex", data="\n".join(f"{v:02X}"
            for v in rom), file_name="hex.txt")
        st.download_button("Download␣Logisim␣ROM", data=logisim_hex,
            file_name="logisim_rom.txt")

    except Exception as e:
        st.error("Compilation␣failed:␣" + str(e))
```

7.2 Application Interface

The compiler was implemented as a Streamlit web app. Users can enter their program in assembly format and generate the binary/hex outputs by clicking `Compile`.

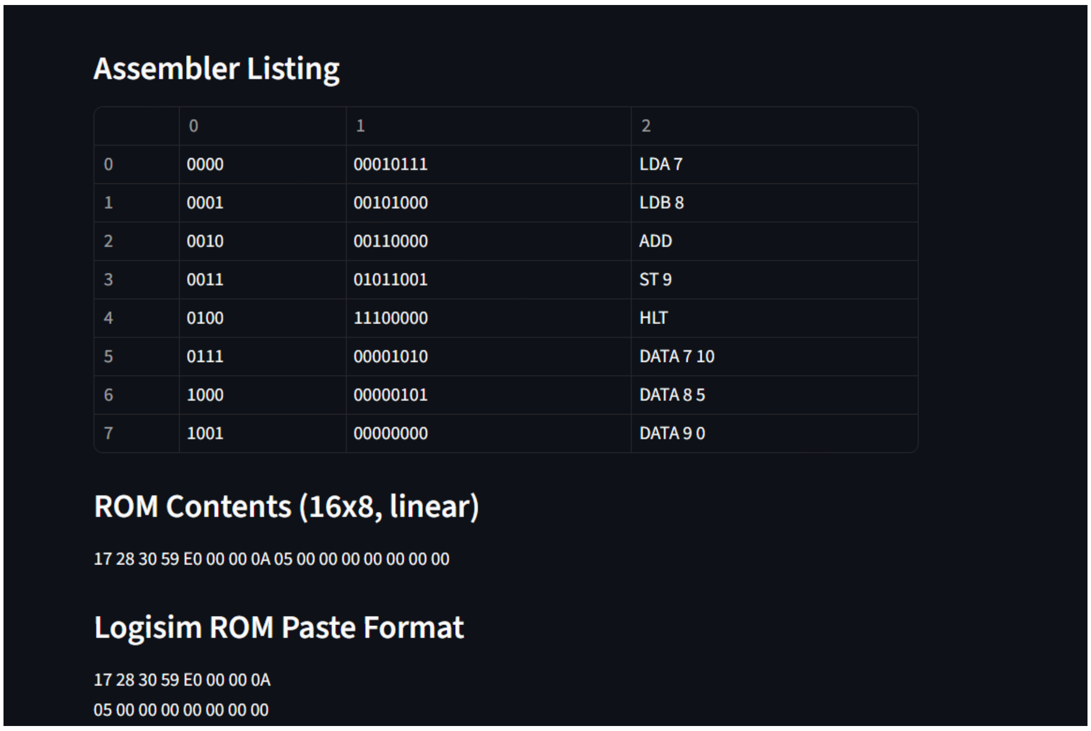


SAP-1 Smart Compiler App

Assembly Program

```
LDA 7
LDB 8
ADD
ST 9
HLT

DATA 7 10
DATA 8 5
DATA 9 0
```

Figure 15: SAP-1 Smart Compiler App with input assembly program.



Assembler Listing

	0	1	2
0	0000	00010111	LDA 7
1	0001	00101000	LDB 8
2	0010	00110000	ADD
3	0011	01011001	ST 9
4	0100	11100000	HLT
5	0111	00001010	DATA 7 10
6	1000	00000101	DATA 8 5
7	1001	00000000	DATA 9 0

ROM Contents (16x8, linear)

```
17 28 30 59 E0 00 00 0A 05 00 00 00 00 00 00 00
```

Logisim ROM Paste Format

```
17 28 30 59 E0 00 00 0A
05 00 00 00 00 00 00 00
```

Figure 16: Assembler listing and ROM contents automatically generated by the compiler.

8 Testing the CPU

8.1 Manual Mode

In manual mode, the program is entered into SRAM manually using the debug interface. The following procedure shows the exact pin-level steps used to load the program, verify SRAM contents, reset the system, and execute instructions.

Step 1: Manual Loading of Program

- Turn on `debug` pin.
- Pulse the `pc_reset` pin.
- Set `debug_data` = `0000 0000` (address 0).
- Toggle `mar_in_en_manual` and give a clock pulse.
- Turn off `mar_in_en_manual`.
- Set `debug_data` = `0000 0111` (LDA 7).
- Toggle `sram_wr_manual` and give a clock pulse.
- Turn off `sram_wr_manual`; verify LDA stored at 0000.
- Set `debug_data` = `0000 0001` (address 1).

- Toggle `mar_in_en_manual` and pulse clock.
- Set `debug_data` = `0001 1000` (LDB 8).
- Toggle `sram_wr_manual` and pulse clock.
- Verify LDB stored at 0001.
- Set `debug_data` = `0000 0010` (address 2).
- Toggle `mar_in_en_manual` and pulse clock.
- Set `debug_data` = `0010 0000` (ADD).
- Toggle `sram_wr_manual` and pulse clock.
- Verify ADD stored at 0010.
- Set `debug_data` = `0000 0011` (address 3).
- Toggle `mar_in_en_manual` and pulse clock.
- Set `debug_data` = `0100 1001` (ST 9).
- Toggle `sram_wr_manual` and pulse clock.
- Verify ST stored at 0011.
- Set `debug_data` = `0000 0100` (address 4).
- Toggle `mar_in_en_manual` and pulse clock.
- Set `debug_data` = `1111 0000` (HLT).
- Toggle `sram_wr_manual` and pulse clock.
- Verify HLT stored at 0100.
- Set `debug_data` = `0000 0111` (address 7).
- Toggle `mar_in_en_manual` and pulse clock.
- Set `debug_data` = `0000 1010` (value 10).
- Toggle `sram_wr_manual` and pulse clock.
- Verify data 10 stored at 0111.
- Set `debug_data` = `0000 1000` (address 8).
- Toggle `mar_in_en_manual` and pulse clock.
- Set `debug_data` = `0000 0101` (value 5).
- Toggle `sram_wr_manual` and pulse clock.
- Verify data 5 stored at 1000.

- Set `debug_data` = `0000 1001` (address 9).
- Toggle `mar_in_en_manual` and pulse clock.
- Set `debug_data` = `0000 0000` (value 0).
- Toggle `sram_wr_manual` and pulse clock.
- Verify data 0 stored at 1001.
- Turn off `debug` pin.

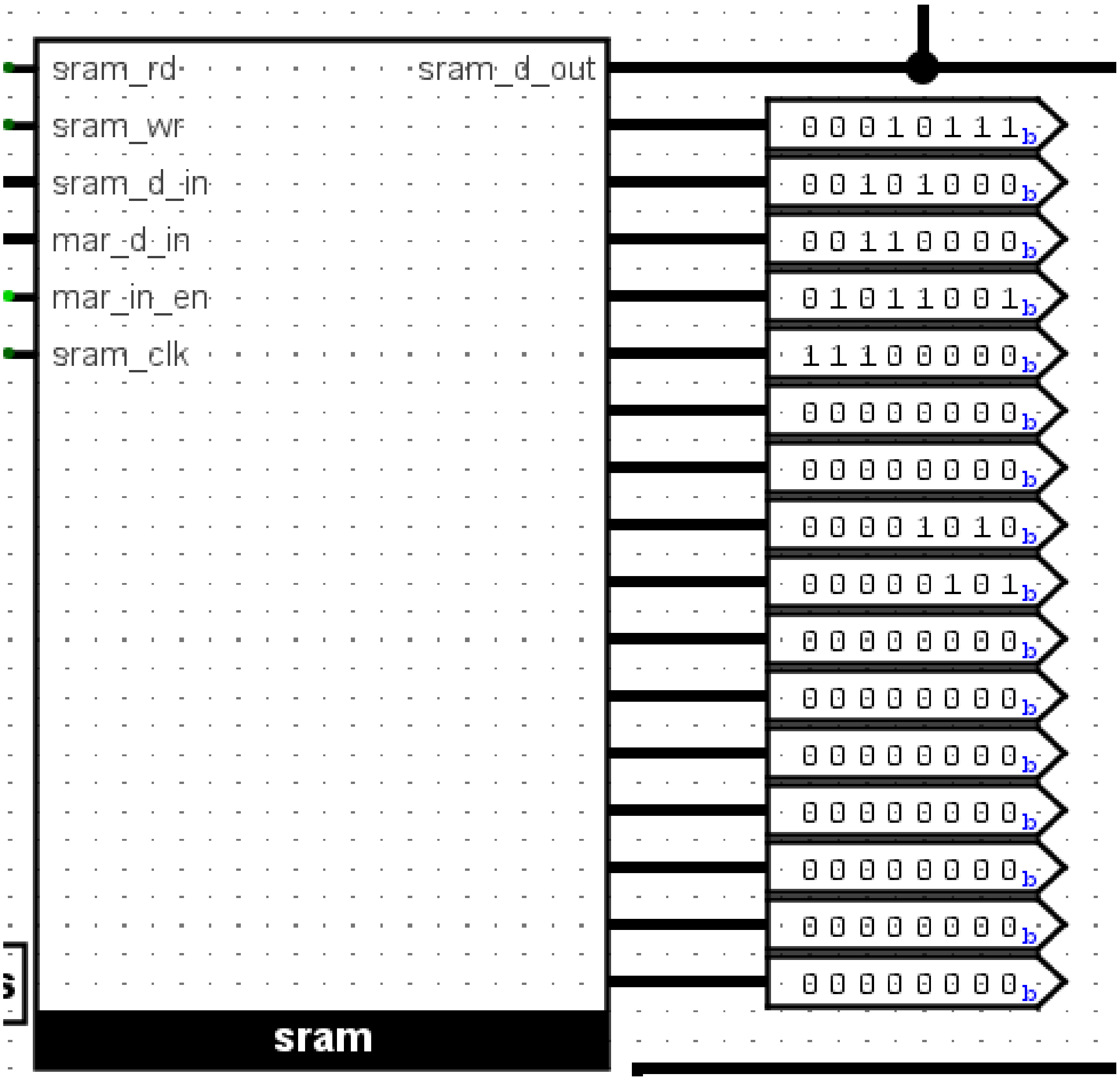


Figure 17: SRAM contents after manual program loading.

Step 2: Reset and Initialization

- Apply `clk_reset` and `pc_reset`.

- Enable `rc_en` (ring counter).
- Begin execution by applying clock pulses.

Step 3: Program Execution After loading and reset, the CPU executes instructions by enabling `rc_en` and applying clock pulses. Each instruction requires six clock pulses (T_1–T_6). The program runs step by step as follows:

0.45

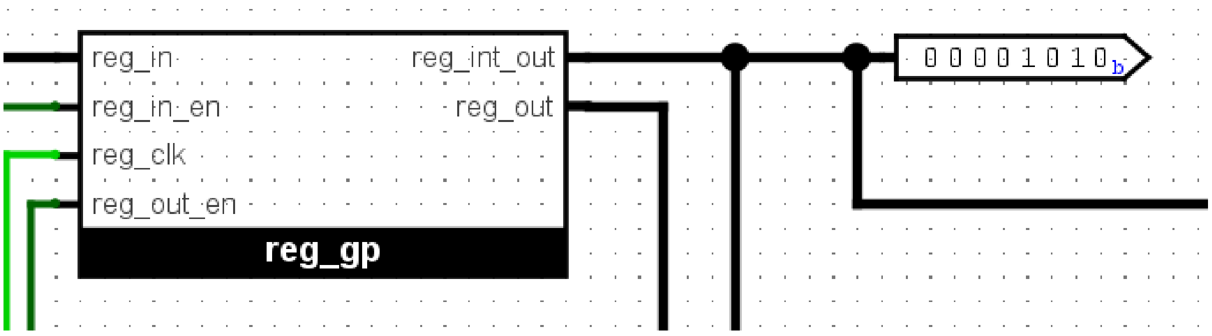


Figure 18: LDA 7: A = 10

0.45

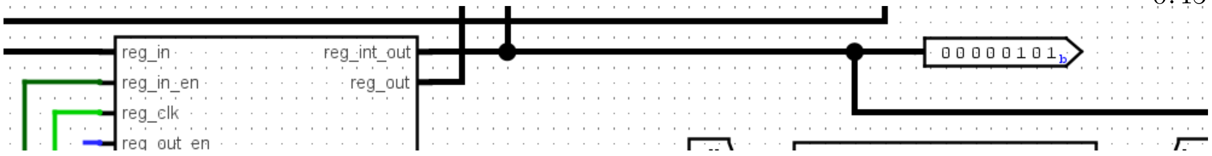


Figure 19: LDB 8: B = 5

0.45

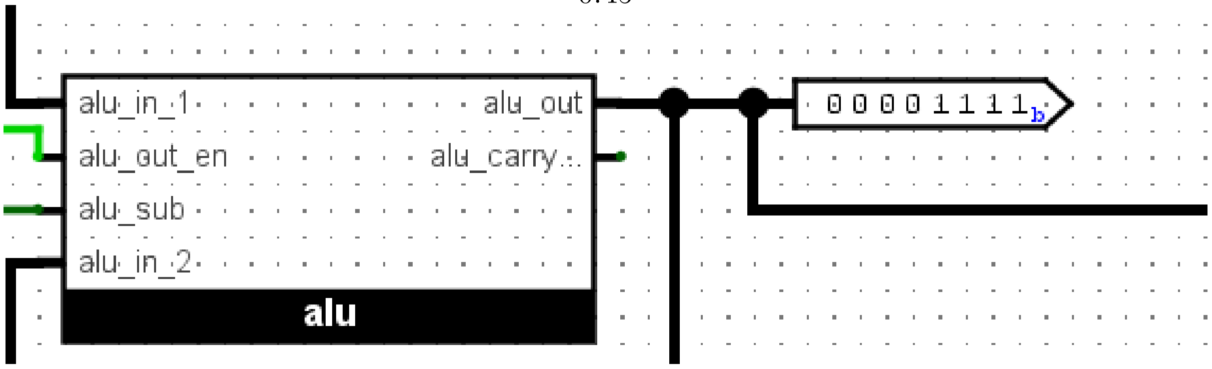


Figure 20: ADD: A = 15

Finally, after the next 6 pulses, the HLT instruction is executed and the CPU halts.

8.2 Auto Mode

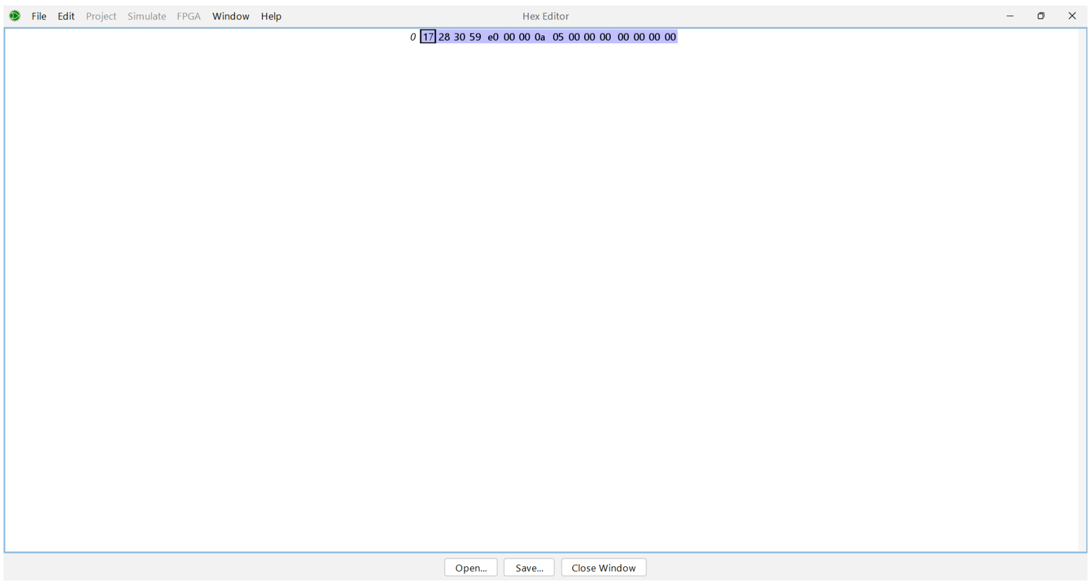
In auto mode, the program is first placed into the ROM (generated from the Python compiler). The instruction loader then copies the ROM contents into the SRAM automatically. Once loading is complete, the CPU begins execution as in manual mode. The following procedure was followed:

1) Initial Setup

- Open the `sap1_2008036_auto.circ` design in Logisim Evolution.
- Ensure the `debug` pin is OFF (LOW).
- Ensure the main clock (`clk`) is OFF.
- Pulse the `pc_reset` pin once to reset the Program Counter (PC) to 0000.

2) Program the ROM

- Right-click the ROM component and select `Edit Contents...`.
- Enter the hex values of the program as obtained from the compiler (see Example Program section).
- For example:
  - ADD Program: `07 18 20 49 F0 00 00 00 0A 05 00`
  - SUB Program: `07 18 30 49 F0 00 00 00 0A 05 00`
- Alternatively, load the ROM contents from a pre-generated file exported by the compiler.



```
0 17 28 30 59 e0 00 00 0a 05 00 00 00 00 00 00 00
```

Figure 21: ROM contents programmed with the Example Program (via compiler output).

3) Load Program to RAM (Bootloader Mode)

- Turn ON the `debug` pin (HIGH). The "Loader Active" indicator LED switches ON.
- With each `clk` pulse, the loader automatically transfers one instruction/data word from ROM into SRAM. Each instruction takes two clock pulses: one for ROM output, one for SRAM write.

- Allow the CPU to cycle through all addresses until all program/data values are copied.
- Observe the MAR address and data bus values on the 7-segment display.

4) Stop the Bootloader

- Turn OFF the `debug` pin (LOW).
- Pulse the `clk` once more to ensure the loader process halts completely.

5) Run the Program

- Pulse the `pc_reset` pin again to reset the PC to 0000.
- Enable `rc_en` (ring counter) and apply clock pulses.
- Execution now proceeds exactly as in manual mode : LDA loads A=10, LDB loads B=5, ADD computes 15, ST stores result at address 9, HLT stops execution.

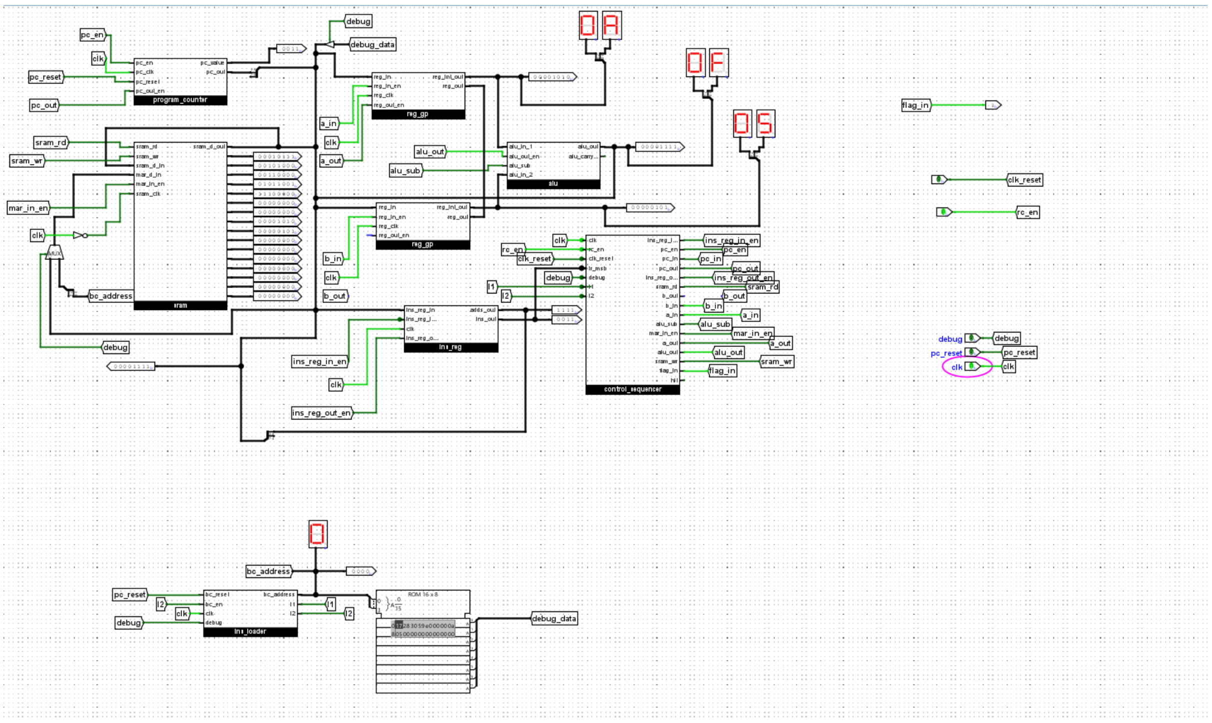


Figure 22: Execution in Auto Mode (same sequence as manual: A=10, B=5, A=15, Mem[9]=15).

9 Conclusion

This project successfully implemented the Simple-As-Possible (SAP-1) microprocessor architecture in Logisim, with both manual and automatic versions. The manual circuit demonstrated the fundamental operation of a hardwired CPU, including fetch–decode–execute cycles, register transfers, and memory operations. The automatic version extended this design with a bootloader and instruction loader, enabling programs to be loaded directly from ROM into SRAM without manual intervention. A Python-based assembler/compiler was also developed, which translates assembly programs into binary and hexadecimal machine code compatible with the SAP-1 design.

Testing verified that the system executed arithmetic and data transfer programs correctly. For example, the sample program (LDA 7, LDB 8, ADD, ST 9, HLT) successfully added two numbers stored in memory and placed the result back in SRAM. This demonstrates not only the correctness of the CPU datapath and control logic but also the seamless integration of the assembler, loader, and execution pipeline.

Overall, this project provides a complete educational CPU system — from instruction set definition and compiler support to hardware-level simulation and execution.

10 Future Improvements

While the SAP-1 design achieved its intended goals, there are several directions for extension and improvement:

- **Additional Instructions:** Implement PUSH/POP with a stack pointer and extend CMP with zero/negative flags for conditional branching.
- **Extended Datapath:** Upgrade the 8-bit SAP-1 to a 16-bit SAP-2 style design with larger memory and registers.
- **Microcoded Control:** Replace the hardwired control sequencer with a microcoded control store to simplify the addition of new instructions.
- **I/O Integration:** Add input/output devices such as a keyboard or serial port to allow interactive programs.
- **Assembler Enhancements:** Expand the Python compiler with labels, variables, and error-checking for a more user-friendly programming model.
- **Pipeline Experiments:** Investigate instruction pipelining and hazard handling to move the design toward more advanced CPU architectures.