Table of Contents

13. Appendix
    A. Schematic Snapshots (main, cs, ins_tab, rc, etc.)
    B. Control Equations (consolidated)
    C. Full HEX Listings for Test Programs

1.Abstract

This report presents the design and simulation of an 8-bit **SAP-1** computer implemented in **Logisim Evolution**. The processor follows the classic single-bus architecture with a **hardwired control unit** and extends the baseline instruction set to **LDA, LDB, ADD, SUB, STA, JMP, HLT** alongside **bit-shift and rotate** operations (**SHL, SHR, ROL, ROR**). Two operating modes are provided: **Automatic** mode executes the fetch–decode–execute cycle driven by a ring counter (**T1–T6**) and an opcode decoder, while **Manual/Loader** mode loads programs from ROM into RAM for debugging and demonstrations. The control sequencer asserts non-overlapping bus-enable signals and dedicated shifter controls (**sh_out, sh_dir, sh_rot**) to maintain single-driver discipline on the system bus. A lightweight web-based assembler converts human-readable assembly (with ORG/DEC directives) into Logisim-compatible **v2.0 raw** HEX images, enabling repeatable experiments. Functional verification with arithmetic, control-flow, and bit-manipulation programs confirms correct micro-operation timing—memory-operand instructions complete in **T5**, while shift/rotate complete in **T4**—and correct final states (e.g., stored results at RAM address 0x0F). The project provides a clear, extensible template for undergraduate study of processor control design and micro-architecture.

2.Project Overview:

This project implements a complete **SAP-1 (Simple As Possible)** 8-bit computer in **Logisim Evolution**, extended with modern conveniences for teaching and verification. The design follows the classic single-bus architecture (8-bit data bus, 4-bit address space = 16 bytes of memory) and uses a **hardwired control sequencer** to realize the fetch–decode–execute cycle. In addition to the baseline SAP-1 instructions (**LDA, LDB, ADD, SUB, STA, JMP, HLT**), the processor is extended with **bit-manipulation** instructions: **SHL, SHR, ROL, ROR**.

The CPU operates in two modes:

1. **Automatic mode** (normal execution), where a **ring counter (T1–T6)** and an **opcode decoder** generate time-aligned control pulses for each instruction; and
2. **Manual/Loader mode**, which allows loading a program image from a ROM (or panel inputs) into RAM without bus conflicts, then switching back to run automatically. The mode is controlled by a debug signal, and loader handshakes ensure that fetch logic is masked while memory is being written.

The **datapath** consists of: an 8-bit **A** (accumulator) and **B** register (both built from a reusable reg_gp block with a tri-state bus output and an internal tap), a ripple-carry **ALU** (ADD/SUB selected by alu_sub), a dedicated **8-bit shifter/rotator** path for SHL/SHR/ROL/ROR, a 4-bit **Program Counter (PC)** with increment and **direct load** for JMP, a 4-bit **MAR** (address register), a 16×8 **SRAM**, and an **Instruction Register (IR)** that splits into IR[7:4] (opcode to the decoder) and IR[3:0] (operand address on the bus at execute time). All bus **sources** (PC out, A out, B out, ALU out, IR low nibble out, RAM read, Shifter out) are tri-stated so only **one driver** is enabled per T-state.

The **control unit** (subcircuit cs) is hardwired: it combines the one-hot opcode lines from ins_tab with T-states from the ring counter to assert final control signals such as pc_out, pc_en, mar_in_en, sram_rd, sram_wr, ins_reg_in_en, ins_reg_out_en, a_in, a_out, b_in, b_out, alu_out, alu_sub, jump_en, and hlt. For the new shift/rotate instructions, the controller asserts **sh_out** (tri-state enable of the shifter onto the bus) and selects **direction** (sh_dir) and **rotate/shift** mode (sh_rot) so that **A** captures the transformed value in a single execute cycle (T4). Loader masking (~i2) and cpu_mode = ~debug prevent bus fights during program loading.

To streamline testing, a compact **web-based assembler** was created that accepts simple SAP-1 assembly (with ORG and DEC directives) and emits **Logisim v2.0 raw** HEX images compatible with the ROM/loader. This enables reproducible experiments: arithmetic (addition/subtraction), control flow (jump), and bit-manipulation (shift/rotate) programs are assembled and loaded quickly.

Verification is done by single-stepping the CPU and probing **PC, MAR, IR, A/B, ALU out, Shifter out, Bus**, and **RAM**. Observations confirm correct micro-operation timing: memory-operand instructions complete on **T5** (e.g., LDA, STA), ALU ops complete on **T4** (ADD, SUB), and the new shift/rotate instructions also complete on **T4** with only the shifter driving the bus. Final results (e.g., stored at RAM address 0x0F) match the expected hex listings.

Deliverables include: the Logisim .circ project (auto + manual/loader circuits), annotated schematics for each subcircuit, the assembler HTML tool, reference **HEX** programs for addition, subtraction, jump, shift, rotate, and this documentation describing the architecture, control design, instruction timing, and test outcomes.

3.Features:

- **Classic SAP-1 Architecture (extended)**
  Single shared 8-bit data bus with a 4-bit address space (16 bytes). Hardwired control (no microcode). Clean, didactic design that matches SAP-1 while adding useful extras.
- **Instruction Set (yours)**
  LDA, LDB, ADD, SUB, STA, JMP, HLT + **bit-manipulation**: SHL, SHR, ROL, ROR.
  New shift/rotate execute in a **single T4** cycle using a dedicated shifter path.
- **Two Operating Modes**
  Automatic mode: normal fetch–decode–execute driven by a ring counter (**T1–T6**).
  Manual/Loader mode: safe ROM→RAM program loading (or panel entry) with debug high and loader handshakes i1/i2.
- **Hardwired Control Unit (cs)**
  Combines **T-states** from the ring counter with one-hot opcode lines from ins_tab. Produces final control signals:
  pc_out, pc_en, mar_in_en, sram_rd, sram_wr, ins_reg_in_en, ins_reg_out_en, a_in, a_out, b_in, b_out, alu_out, alu_sub, jump_en, hlt and **new** sh_out, sh_dir, sh_rot.
  Loader masking uses cpu_mode = ~debug and ~i2 to avoid bus fights while loading.
- **Strict Single-Driver Bus Discipline**
  All bus sources are tri-stated; at any T-state only **one** of these can be enabled: pc_out, sram_rd, ins_reg_out_en, a_out, b_out, alu_out, **sh_out**. Prevents contention (no red bus).
- **Datapath Components (reusable blocks)**
  - **A/B Registers (reg_gp)**: 8-bit with tri-state bus output and an always-on internal tap to ALU/Shifter (reg_int_out).
  - **ALU**: ripple adder/subtractor; alu_sub selects SUB; alu_out tri-states result to bus.
  - **Shifter/Rotator Path**: 8-bit shifter + (optionally) 4→1 MUX; output tri-stated by sh_out; controls **sh_dir** (0=left,1=right) and **sh_rot** (0=shift,1=rotate).
  - **PC (4-bit)**: increments at T3 via pc_en; direct load on jump_en at JMP execute.
  - **MAR (4-bit)**: captures bus at mar_in_en (T1 for fetch, T4 for memory-operand exec).

    - **SRAM (16×8)**: separate sram_rd (data→bus) and sram_wr (bus→data) strobes.
    - **IR (8-bit)**: IR[7:4] to ins_tab (opcode); IR[3:0] drives bus via ins_reg_out_en for LDA/LDB/STA/JMP addresses.
- **Opcode Decoder (ins_tab)**
  4→16 decoder generates one-hot lines: insLDA, insLDB, insADD, insSUB, insSTA, insJMP, insHLT, insSHL, insSHR, insROL, insROR (others unused).
- **Precise Timing (micro-ops)**
  Fetch (all): T1: PC→MAR, T2: RAM→IR, T3: PC←PC+1.
  Execute:
  – Memory-operand (LDA/LDB/STA) complete by **T5**.
  – ALU ops (ADD/SUB) complete in **T4**.
  – **SHL/SHR/ROL/ROR** complete in **T4** with sh_out=1 and a_in=1.
  – **JMP** loads PC at **T4** via ins_reg_out_en + jump_en.
  – **HLT** asserts hlt at **T4** and stops the ring counter.
- **Safe Loader Flow**
  In Manual/Loader mode (debug=1), normal fetch paths are masked (~i2) while program bytes move ROM→RAM. After loading, switch debug=0 to run.
- **Assembler/Compiler Support**
  A lightweight web tool converts assembly (ORG, DEC + mnemonics) into **Logisim v2.0 raw** HEX for direct ROM/RAM loading.
- **Verification-Friendly**
  Designed for step-through probing: watch **PC, MAR, IR, A/B, ALU out, Shifter out, Bus**, and **SRAM**. Wire colors (green=1, blue=0) make timing easy to confirm.
- **Extensible Template**
  Clean control matrix and decoder make it straightforward to add new instructions (e.g., immediate load, conditional jumps) or flags (Zero/Carry) later.

4.Architecture & Components:

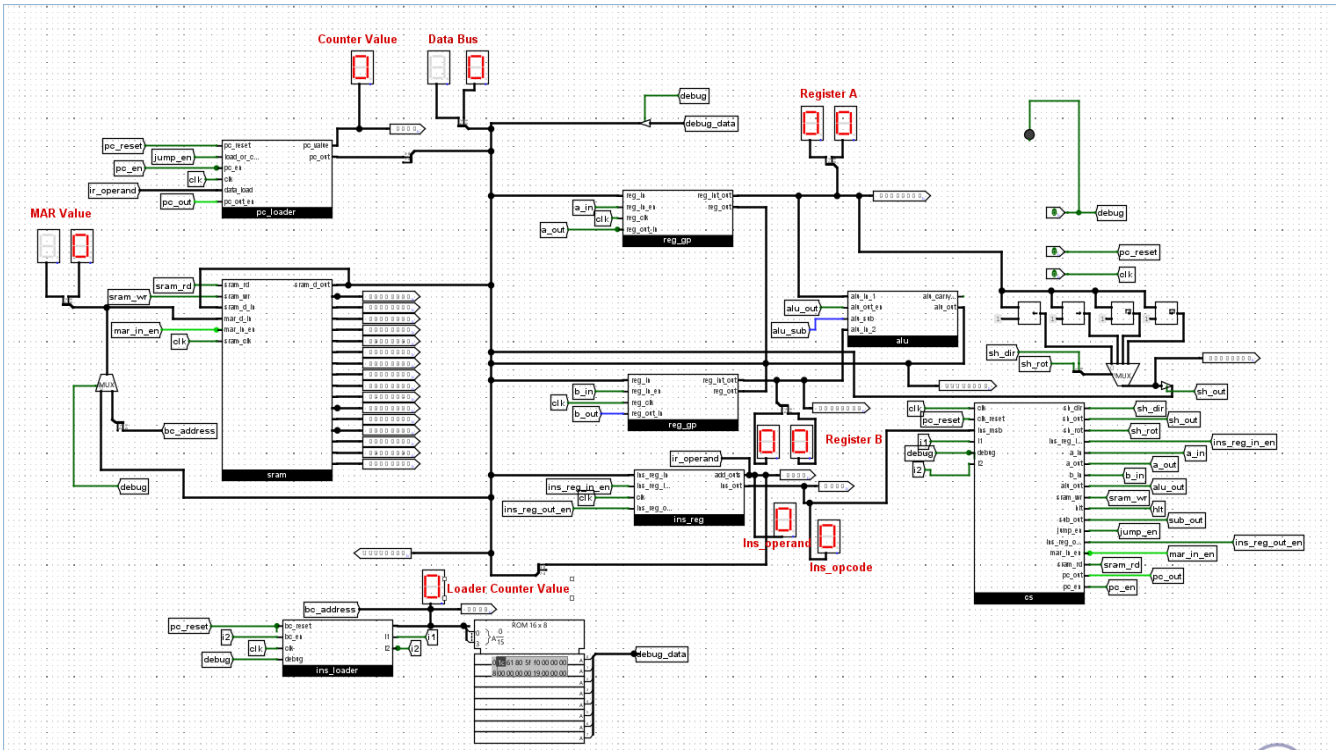


Figure 4.1: Datapath Overview (single-bus, 8-bit)

The CPU uses a single 8-bit data bus with tri-stated sources. Only one source may drive the bus at any T-state: pc_out, sram_rd, ins_reg_out_en, a_out, b_out, alu_out, sh_out. Bus listeners (latches) are mar_in_en, ins_reg_in_en, a_in, b_in, and sram_wr. Addressing is 4-bit (16 bytes). All sequential elements share the system clock.

4.1 Datapath Overview

- **Registers:** A (accumulator) and B (reg_gp blocks) with:
  - reg_in_en ← a_in / b_in
  - reg_out_en ← a_out / b_out
  - reg_int_out (always-on internal tap) feeds ALU/Shifter
- **ALU:** ripple adder/subtractor; alu_sub selects SUB; result tri-stated by alu_out
- **Shifter/Rotator:** 8-bit shifter (or 4× shifters + 4⟶1 MUX) with:
  - sh_dir (0=left, 1=right), sh_rot (0=shift, 1=rotate)
  - amount = 3-bit constant 001

    - output tri-stated by sh_out
- **Memory subsystem:** 16×8 SRAM with MAR (4-bit address register)
- **Control Unit:** ring counter (T1–T6) + opcode decoder (ins_tab) + gate matrix (cs)
- **Program Counter (PC):** increments at T3 (pc_en), direct load on JMP (jump_en)

4.2 Registers (A, B):

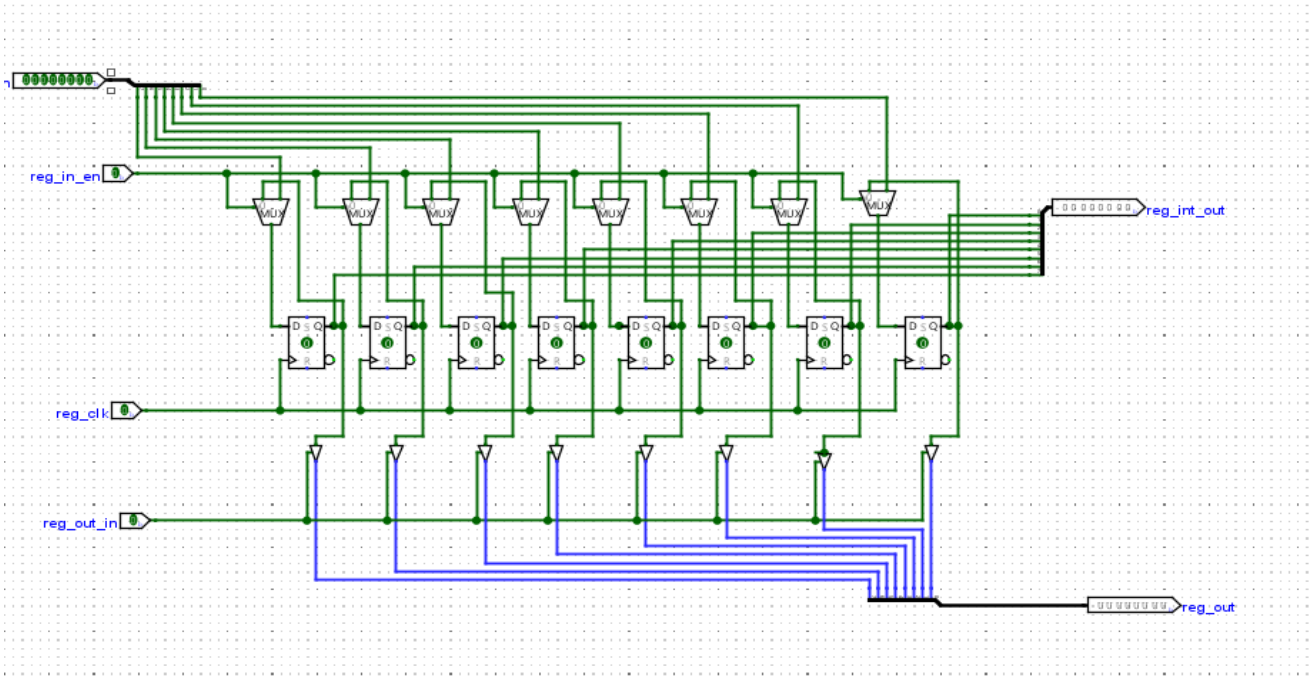
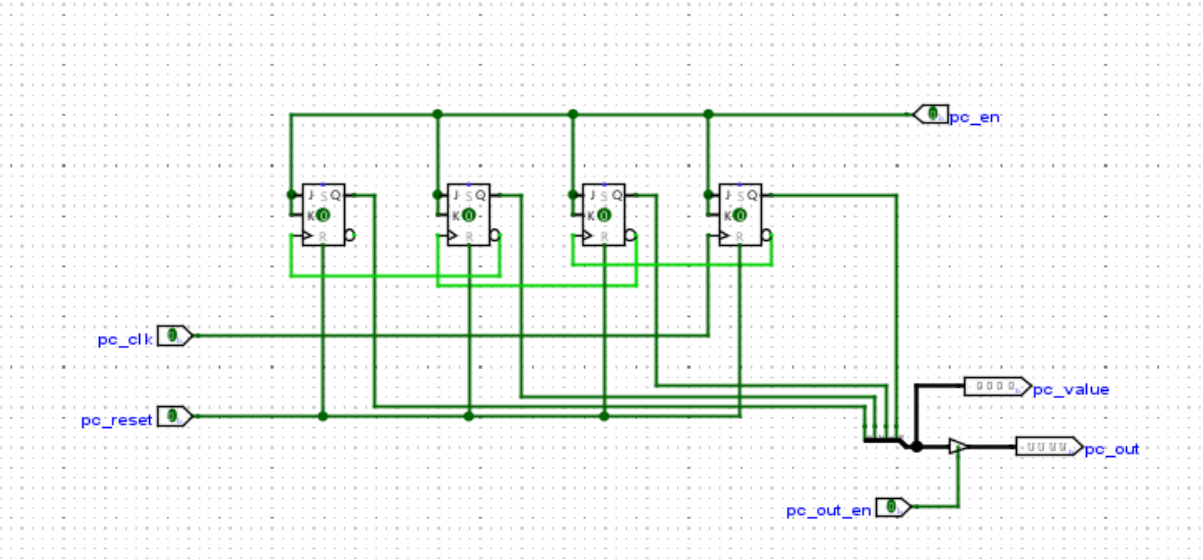


Figure 4.2: A/B Registers (reg_gp) and bus connections

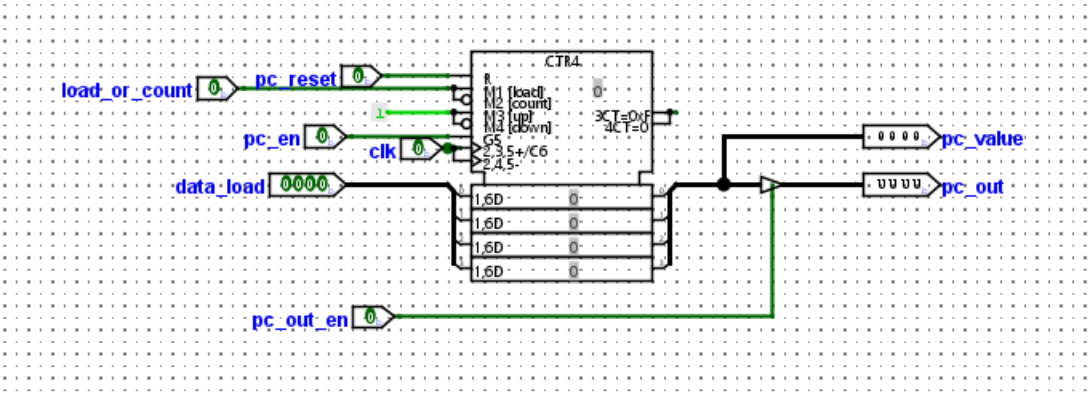
- **Inputs:** reg_in[7:0] from bus, latch on a_in/b_in
- **Outputs:** reg_out[7:0] tri-state to bus on a_out/b_out
- **Internal tap:** reg_int_out[7:0] continuously available (to ALU/Shifter). This lets ALU/Shifter read A/B without turning on the bus.

4.3 Program Counter (PC)

- **T3 (all instructions):** pc_en = 1 → PC ← PC + 1
- **JMP execute (T4):** jump_en = 1 with IR low nibble on bus → PC loads target
- **Bus source:** pc_out = 1 at T1 during fetch so PC can place its value on bus → MAR



(a)

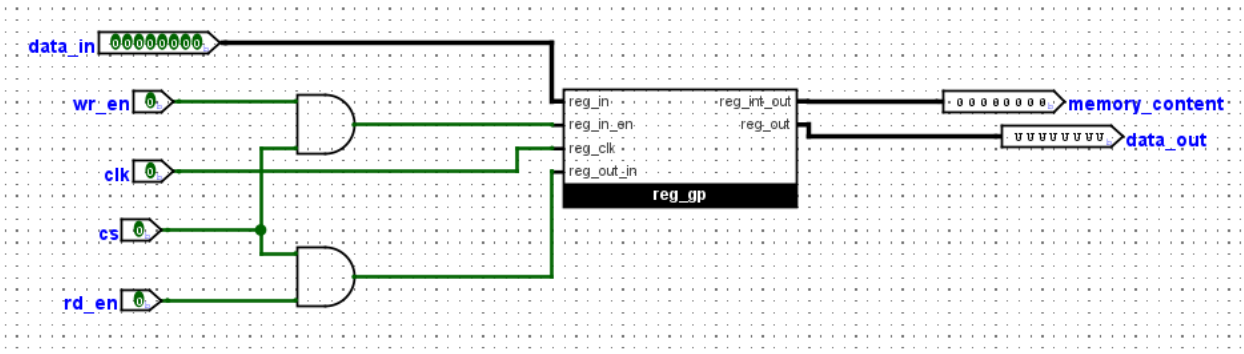


(b)

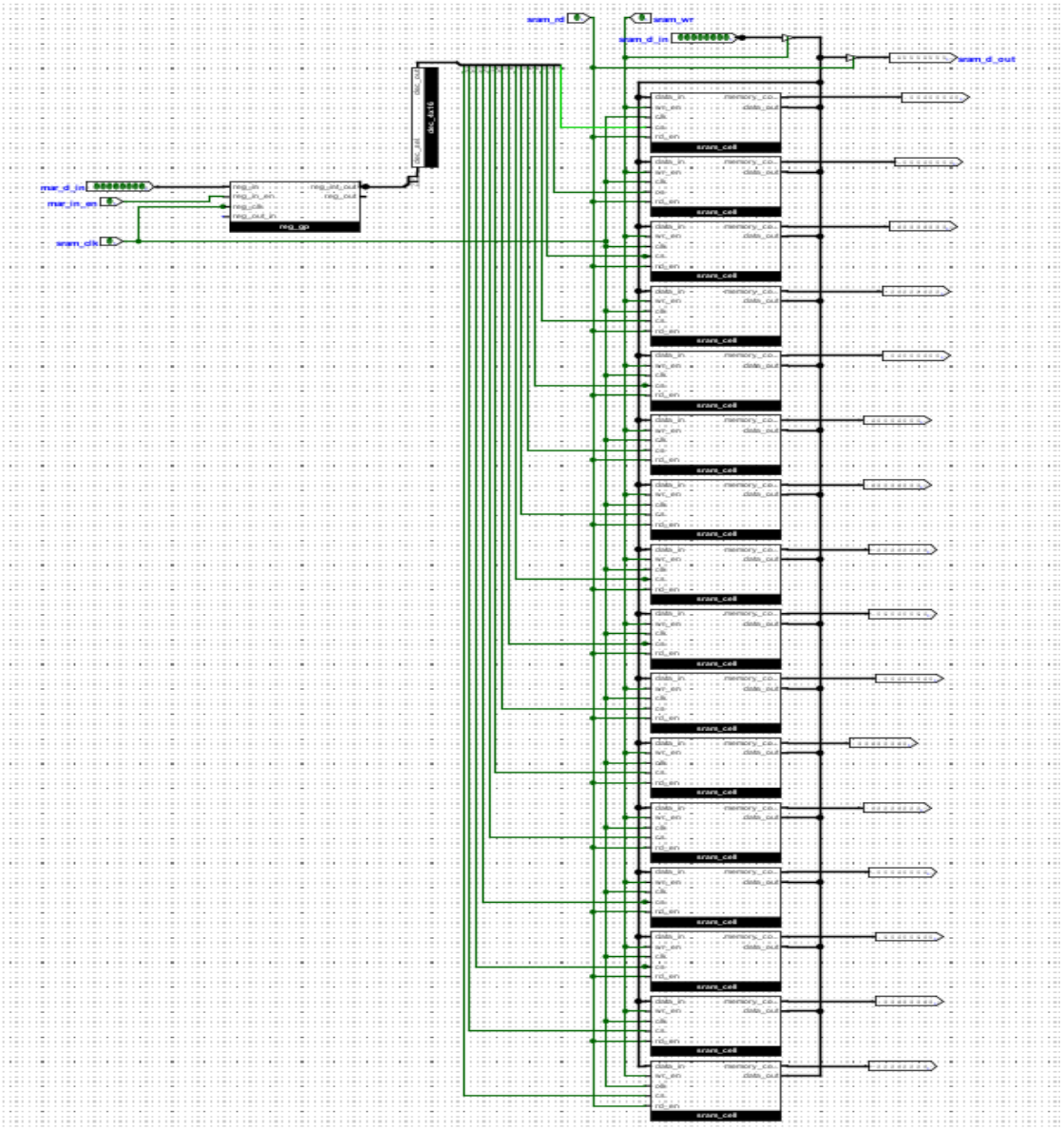
Figure 4.3: Program Counter (increment + jump load)

4.4 Memory & MAR (Address Register)

- **MAR (4-bit):** captures bus on mar_in_en
  - T1: pc_out + mar_in_en → MAR ← PC (fetch address)
  - T4 (LDA/LDB/STA/JMP): ins_reg_out_en + mar_in_en → MAR ← IR[3:0]
- **SRAM read:** sram_rd = 1 → RAM[MAR] drives bus (T2 fetch; T5 for LDA/LDB)
- **SRAM write:** sram_wr = 1 + data on bus (a_out) → RAM[MAR] ← bus (T5 for STA)



(a)



(b)

Figure 4.4: MAR + SRAM (16×8) read/write timing

4.5 Instruction Register (ins_tab)

- **IR load (T2):** sram_rd=1, ins_reg_in_en=1 → IR ← M[MAR]
- **IR[7:4]:** to ins_tab → one-hot lines (above)
- **IR[3:0]:** to bus when ins_reg_out_en=1 (used at T4 for addresses/JMP target)

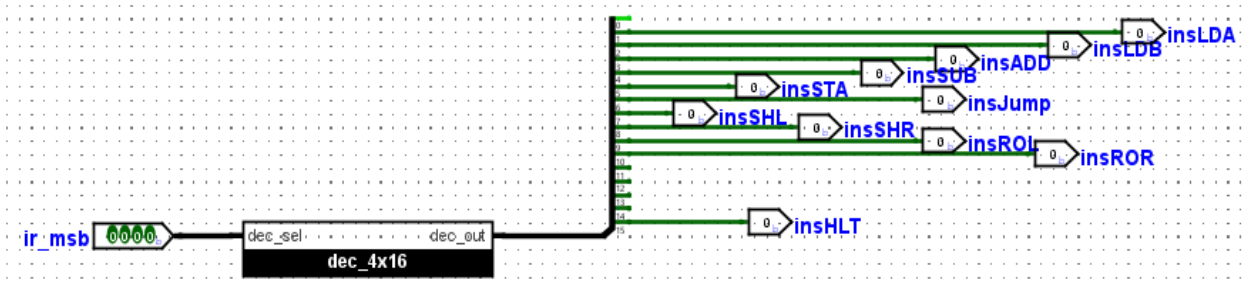


Figure 4.5: IR split and opcode decoder (ins_tab)

4.6 Arithmetic Logic Unit (ALU)

- **Inputs:** A.reg_int_out, B.reg_int_out
- **Control:** alu_sub=1 → A − B, else A + B
- **Execute (T4):** a_out=1, b_out=1, alu_out=1, a_in=1 → A ← A ± B in one step (bus driven only by ALU)

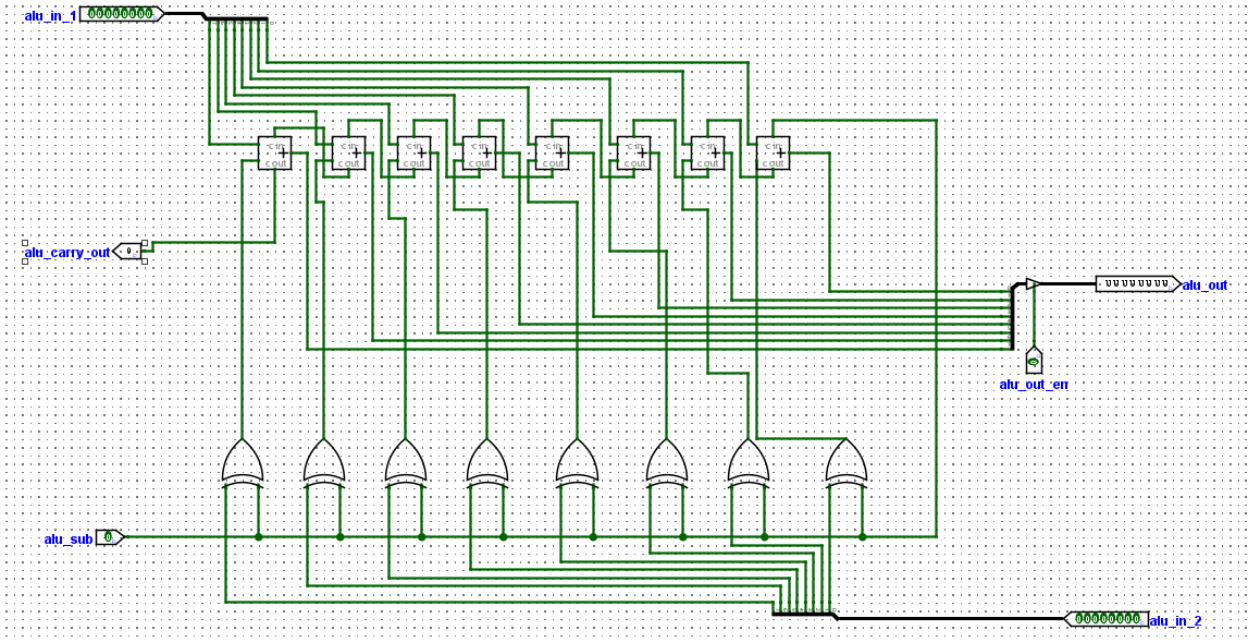


Figure 4.6: ALU (ADD/SUB) and tri-state to bus

4.7 Shifter/Rotator Block (SHL/SHR/ROL/ROR)

- **Inputs:** A.reg_int_out[7:0], amount=3'b001 (1 bit)
- **Control:** sh_dir (0=left,1=right), sh_rot (0=shift zero-fill,1=rotate)
- **Execute (T4):** sh_out=1, a_in=1
  → A ← SHL/SHR/ROL/ROR(A) in one step (bus driven only by Shifter)

4.8 Boot/Loader Counter & Phase Generator (ins_loader)

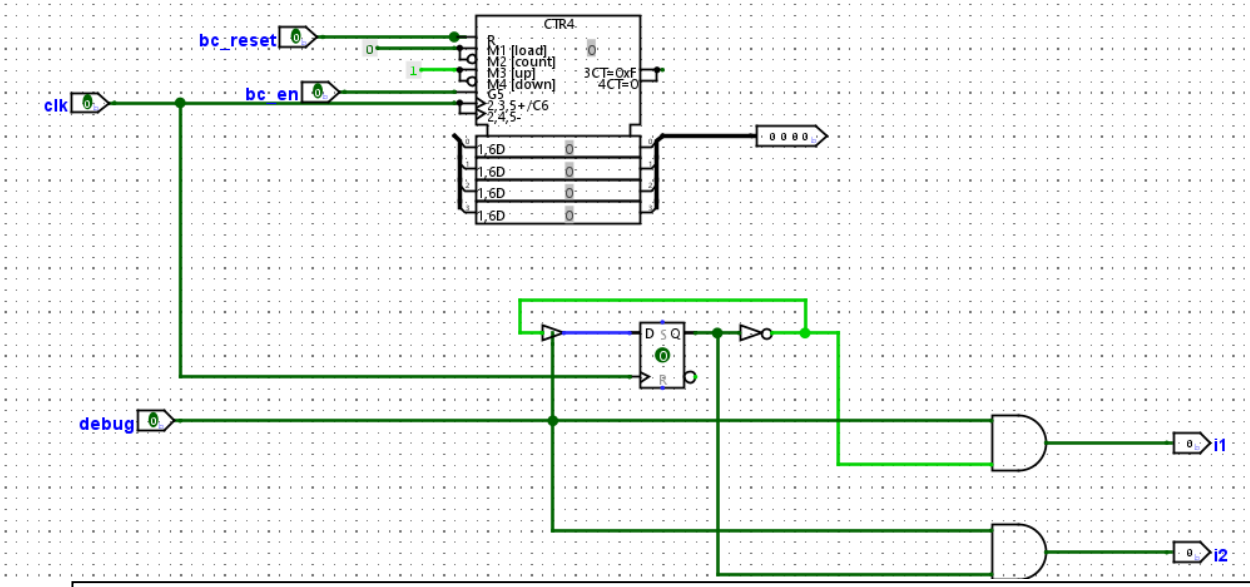


Figure 4.7: Loader address counter and two-phase strobe generator (ins_loader)

- **Role:** provide safe ROM→RAM program loading in Manual/Loader mode (debug = 1)
- **Inputs:** clk, bc_reset (clear counter), bc_en (enable count), debug (mode select)
- **Counter**: CTR4 configured "count up" → bc_address[3:0] = 0000…1111 (write address)
- **Phase generator:** D-FF with feedback + inverter → two non-overlapping phases Φ and ¬Φ
- **Strobes**: i1 = debug ∧ Φ (address phase), i2 = debug ∧ ¬Φ (write phase)
- **Address phase (i1 = 1):** mar_in_en_manual = debug ∧ i1 → MAR ← bc_address
- **Write phase (i2 = 1):** sram_wr_manual = debug ∧ i2 → RAM[MAR] ← ROM_data
- **Repeat: with bc_en =** 1 the counter advances each full clock; i1 then i2 for each address 0…F

- **Run-mode masking:** define cpu_mode = ¬debug; gate normal control in cs with cpu_mode and optionally ¬i2 to prevent overlap with manual writes
- **End of loading:** drop debug to 0 → CPU returns to Automatic mode (fetch–decode–execute)

5. Control Unit — Hardwired Control Sequencer

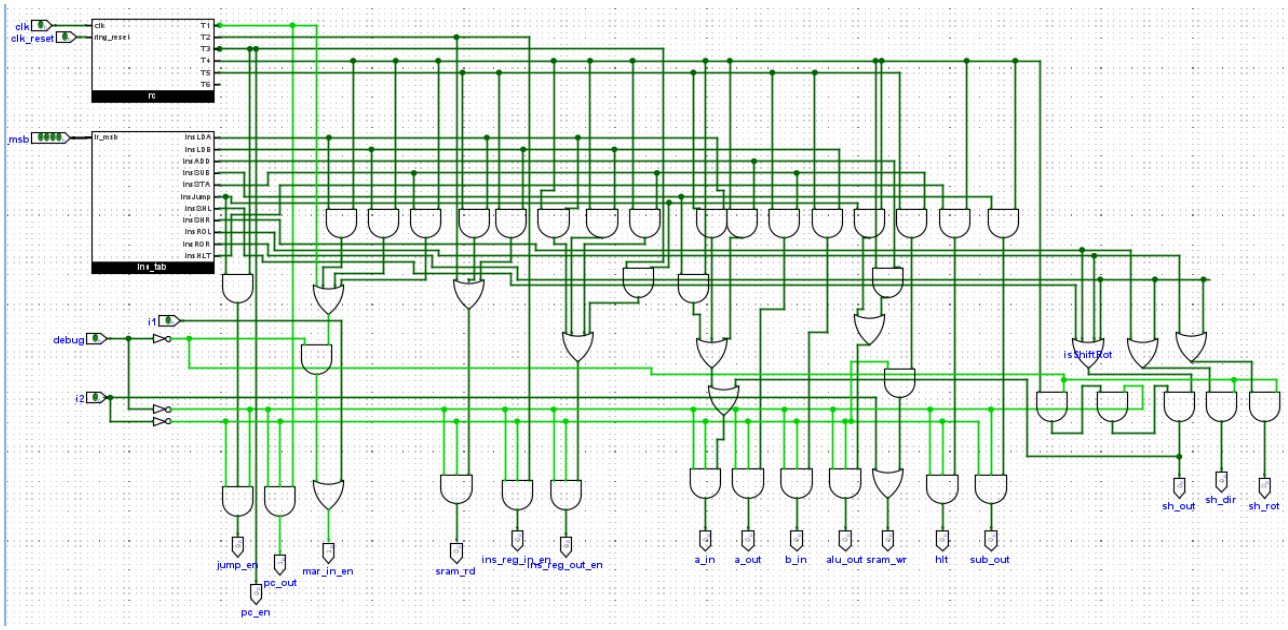


Figure 5.1(a) Control sequencer for Mannal method

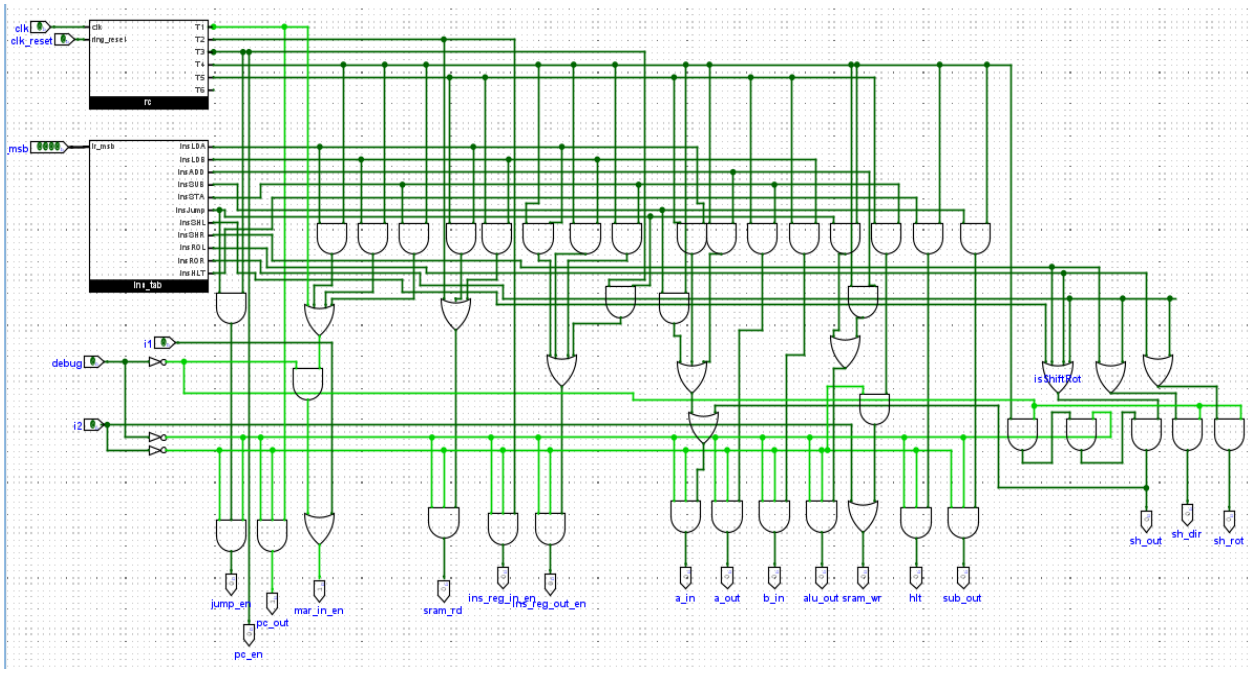


Figure 5.2(b) Control Sequencer for Automatic method.

Role. Convert each instruction into a timed sequence of control pulses that (a) select **one** bus driver, and (b) enable the correct latch(es) each T-state. The unit has:

- **Ring Counter (`rc`)** → T1..T6
- **Opcode Decoder (`ins_tab`)** → `insLDA`, …, `insROR`
- **Mode inputs:** `debug` (manual/loader), `i1/i2` (loader handshakes)
  → define `cpu_mode = ~debug`, mask loader with `~i2`
- **Outputs:**
  `pc_out, pc_en, mar_in_en, sram_rd, sram_wr, ins_reg_in_en, ins_reg_out_en, a_in, a_out, b_in, b_out, alu_out, alu_sub, jump_en, hlt, sh_out, sh_dir, sh_rot`

5.1 Timing Generator (Ring Counter T1–T6)

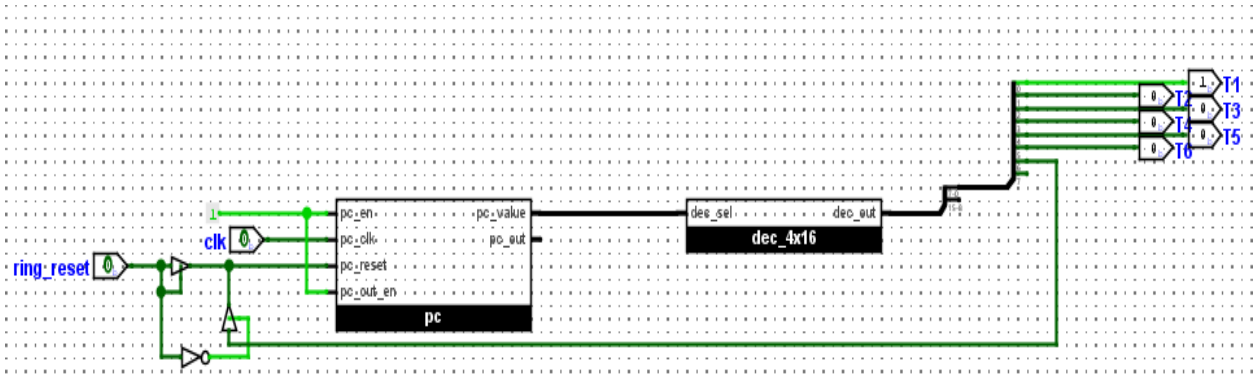


Figure 5.2: Fetch–Decode–Execute timing (T1–T6)

Universal Fetch (all instructions)

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

Execute (as examples)

- **LDA addr**
  T4: ins_reg_out_en, mar_in_en → MAR ← IR[3:0]
  T5: sram_rd, a_in → A ← M[MAR]

- **LDB addr**
  T4: ins_reg_out_en, mar_in_en
  T5: sram_rd, b_in
- **ADD**
  T4: a_out, b_out, alu_out, a_in (and alu_sub=0)
- **SUB**
  T4: a_out, b_out, alu_out, a_in, alu_sub=1
- **STA addr**
  T4: ins_reg_out_en, mar_in_en
  T5: a_out, sram_wr → M[MAR] ← A
- **JMP addr**
  T4: ins_reg_out_en, jump_en → PC ← IR[3:0]
- **HLT**
  T4: hlt=1 → stop ring counter
- **SHL/SHR/ROL/ROR**
  T4: sh_out, a_in (with sh_dir/sh_rot)

5.2 Automatic Mode Control Equations (core minterms)

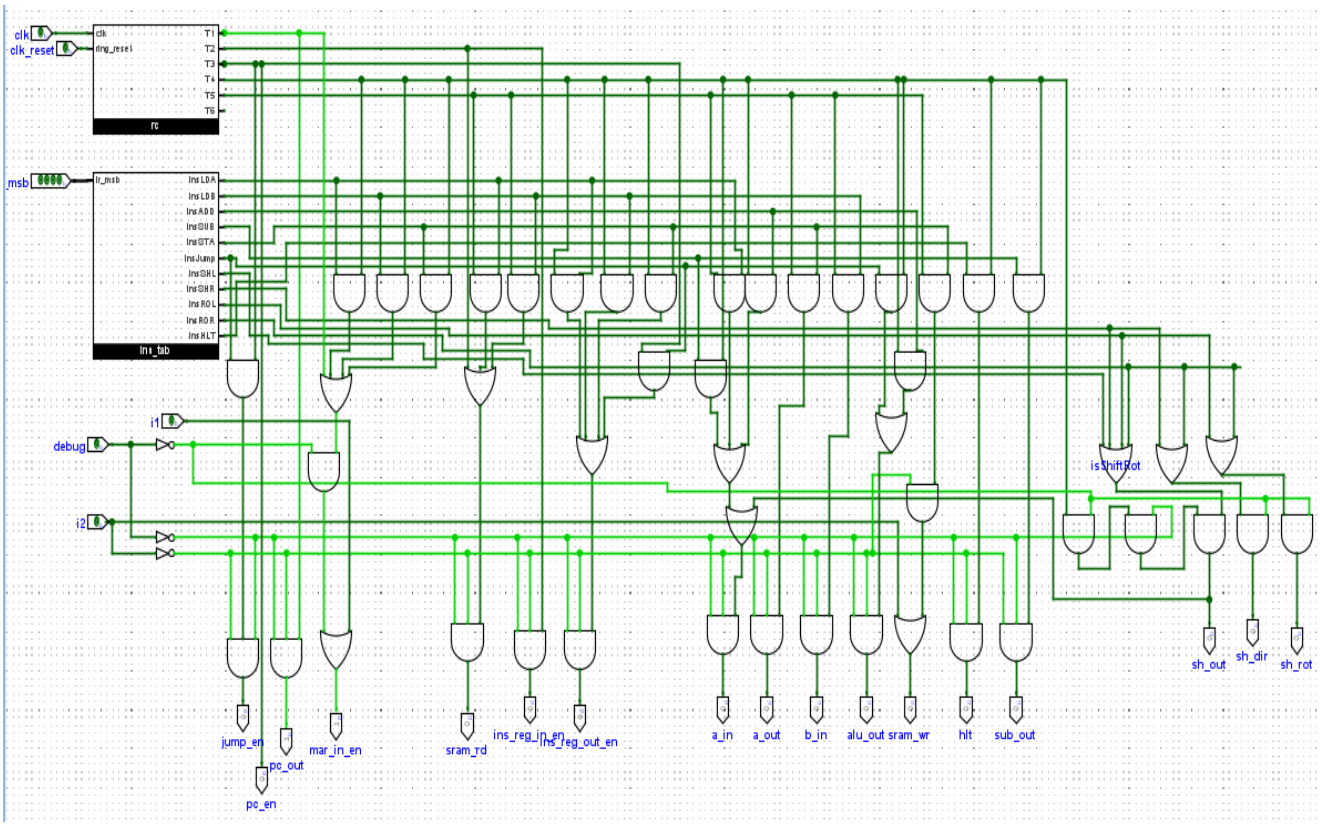


Figure 5.3: Gate-level control matrix (Auto mode)

- Let C = cpu_mode = ~debug, L = ~i2 (loader idle). Representative equations:

```
# Fetch (all)
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

# ALU / Registers
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)

# Memory write
sram_wr       =  T5 & C &  insSTA

# Control flow
ins_reg_out_en =  T4 & C & (insLDA | insLDB | insSTA | insJMP)
jump_en       =  T4 & C &  insJMP
hlt           =  T4 & C &  insHLT
```

5.3 Manual/Loader Mode Control (safe ROM→RAM loading)

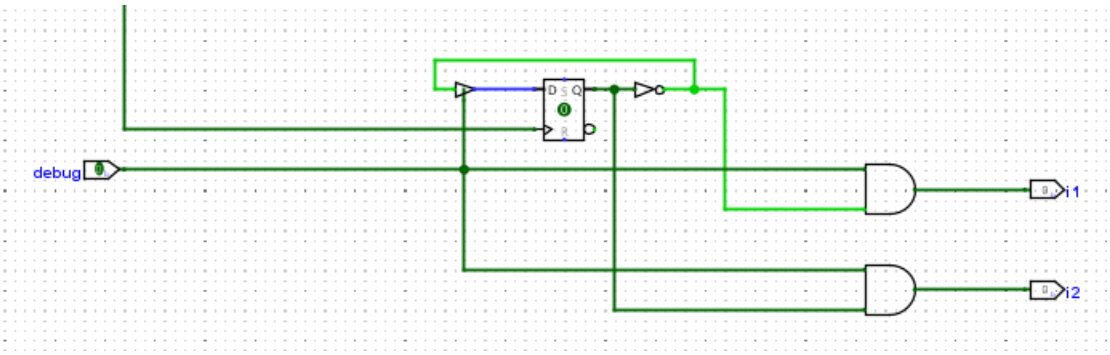


Figure 5.4: Loader gating with debug, i1, i2

- **Mode:** debug=1 → normal CPU cones are masked by C=~debug=0
- **Loader handshakes:** i1/i2 drive MAR write and SRAM write sequences Example (conceptually):

```
mar_in_en = (T1 & C) | (T4 & C & (insLDA|insLDB|insSTA|insJMP)) | (debug & i1)

sram_wr   = (T5 & C & insSTA) | (debug & i2)
```

- When the boot counter finishes, clear debug (or drop i2) to hand control back to Auto mode.

5.4 Jump Path (IR→PC load at execute)

- **Execute (T4):** ins_reg_out_en=1 places IR low nibble on the bus; jump_en=1 makes the PC load from bus instead of incrementing.
- Confirm no other bus drivers at this time (avoid contention).

5.5 Shift/Rotate Control (single-cycle T4)

```
sh_out =  T4 & C & L & (insSHL | insSHR | insROL | insROR)

a_in   =  a_in  | (T4 & C & L & (insSHL | insSHR | insROL | insROR))

sh_dir =  C & (insSHR | insROR)   # 1=right

sh_rot =  C & (insROL | insROR)   # 1=rotate
```

- **Datapath effect:** Shifter drives bus at T4; A latches new value; no memory or ALU used.

- **Discipline:** Ensure `a_out`, `b_out`, `alu_out`, `sram_rd`, `ins_reg_out_en` are **LOW** during T4 of shift/rotate opcodes.

6. Instruction Set

6.1 Opcode Map & Encoding

Table 6.1: Opcode nibble map and byte forms
(Upper nibble = IR[7:4], lower nibble = 4-bit operand/address if used.)

Mnemonic	Upper nibble (hex)	Byte form	Notes
LDA a	**1**	1a	a = 4-bit RAM address
LDB a	**2**	2a	a = 4-bit RAM address
ADD	**3**	30	operand unused
SUB	**4**	40	operand unused
STA a	**5**	5a	a = 4-bit RAM address
JMP a	**6**	6a	**your JMP uses nibble 6**
SHL	**7**	70	shift left by 1 (zero-fill)
SHR	**8**	80	shift right by 1 (zero-fill)
ROL	**9**	90	rotate left by 1
ROR	**A**	A0	rotate right by 1
HLT	**F**	F0	halt

Decoder taps in ins_tab: insJMP, insLDA, insLDB, insADD, insSUB, insSTA, insSHL, insSHR, insROL, insROR, insHLT.

6.2 Micro-operations per Instruction

Fetch (common to all instructions)

- **T1:** pc_out=1, mar_in_en=1 → **MAR ← PC** *(Bus driver: PC)*
- **T2:** sram_rd=1, ins_reg_in_en=1 → **IR ← M[MAR]** *(Bus driver: RAM)*
- **T3:** pc_en=1 → **PC ← PC + 1** *(no bus driver)*

1. **LDA a — Load A from memory**

- **T4:** ins_reg_out_en=1, mar_in_en=1 → **MAR ← IR[3:0]** *(Bus driver: IR low nibble)*
- **T5:** sram_rd=1, a_in=1 → **A ← M[MAR]** *(Bus driver: RAM)*
- **T6:** idle

2. **LDB a — Load B from memory**

- **T4:** ins_reg_out_en=1, mar_in_en=1 → **MAR ← IR[3:0]** *(Bus driver: IR low nibble)*
- **T5:** sram_rd=1, b_in=1 → **B ← M[MAR]** *(Bus driver: RAM)*
- **T6:** idle

3. **ADD — A ← A + B**

- **T4:** a_out=1, b_out=1, alu_sub=0, alu_out=1, a_in=1 → **A ← A + B** *(Bus driver: ALU)*
- **T5/T6:** idle

4. **SUB — A ← A − B**

- **T4:** a_out=1, b_out=1, alu_sub=1, alu_out=1, a_in=1 → **A ← A − B** *(Bus driver: ALU)*
- **T5/T6:** idle

5. **STA a — Store A to memory**

- **T4:** ins_reg_out_en=1, mar_in_en=1 → **MAR ← IR[3:0]** *(Bus driver: IR low nibble)*
- **T5:** a_out=1, sram_wr=1 → **M[MAR] ← A** *(Bus driver: A)*
- **T6:** idle

6. **JMP a — Jump to address**

- **T4:** ins_reg_out_en=1, jump_en=1 → **PC ← IR[3:0]** *(Bus driver: IR low nibble)*
- **T5/T6:** idle

7. **SHL — A ← A << 1 (logical)**

- **T4:** sh_out=1, sh_dir=0, sh_rot=0, a_in=1 → **A ← SHL(A)** *(Bus driver: Shifter)*

- **T5/T6:** idle

8. **SHR — A ← A >> 1 (logical)**

- **T4:** sh_out=1, sh_dir=1, sh_rot=0, a_in=1 → **A ← SHR(A)**
  (Bus driver: Shifter)
- **T5/T6:** idle

9. **ROL — A ← rotate-left 1**

- **T4:** sh_out=1, sh_dir=0, sh_rot=1, a_in=1 → **A ← ROL(A)**
  (Bus driver: Shifter)
- **T5/T6:** idle

10. **ROR — A ← rotate-right 1**

- **T4:** sh_out=1, sh_dir=1, sh_rot=1, a_in=1 → **A ← ROR(A)**
  (Bus driver: Shifter)
- **T5/T6:** idle

11. **HLT — Halt**

- **T4:** hlt=1 → **stop ring counter** (T-states freeze)
- **T5/T6:** n/a

Instruction Set & Program

Address (Binary)	Instruction (Binary)	Hex	Mnemonic & Explanation
00000000	0001 1100	**1C**	**LDA 12** — Load A from address 12
00000001	0010 1101	**2D**	**LDB 13** — Load B from address 13
00000010	0110 0101	**65**	**JMP 5** — Jump to address 5
00000011	0000 0000	**00**	(unused / filler)
00000100	0000 0000	**00**	(unused / filler)
00000101	0011 0000	**30**	**ADD** — A ← A + B *(use this row for addition)*
00000101	0100 0000	**40**	**SUB** — A ← A − B *(use this row instead for subtraction)*
00000110	0101 1111	**5F**	**STA 15** — Store A into address 15

Address (Binary)	Instruction (Binary)	Hex	Mnemonic & Explanation
00000111	1111 0000	**F0**	**HLT** — Halt

Data Values in RAM

Address (Binary)	Data (Binary)	Decimal	Hex
00001100	0011 0011	51	33
00001101	0001 1001	25	19

As example:

HEX (16 bytes)

- **ADD path:** 1C 2D 65 00 00 30 5F F0 00 00 00 00 33 19 00 00
- **SUB path:** 1C 2D 65 00 00 40 5F F0 00 00 00 00 33 19 00 00

Single-cycle shift/rotate options:

Address (Binary)	Hex	Operation	What it does
00000101	**70**	**SHL**	A ← A << 1 (zero in LSB)
00000101	**80**	**SHR**	A ← A >> 1 (zero in MSB)
00000101	**90**	**ROL**	A ← rotate left by 1
00000101	**A0**	**ROR**	A ← rotate right by 1

Example HEX (SHL version):
1C 70 5F F0 00 00 00 00 00 00 00 00 19 00 00 00

7. Programs & Experiments

7.1 Addition & Subtraction Demos (no JMP)

- **HEX (v2.0 raw, 16 bytes):**
  1C 2D 30 5F F0 00 00 00 00 00 00 00 33 19 00 00

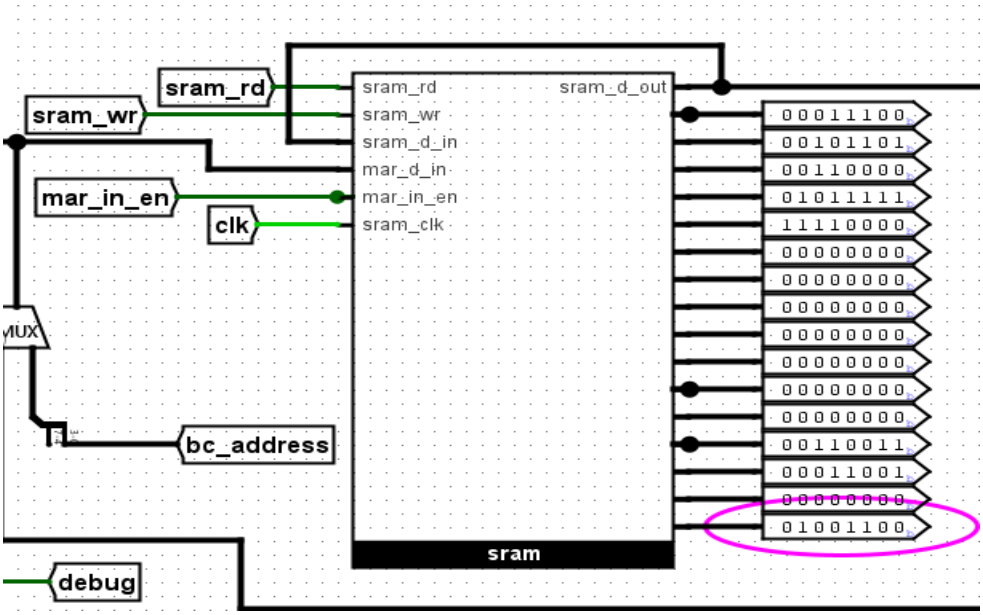


Figure 7.1:(a)addition

- **HEX:**
  1C 2D 40 5F F0 00 00 00 00 00 00 00 33 19 00 00

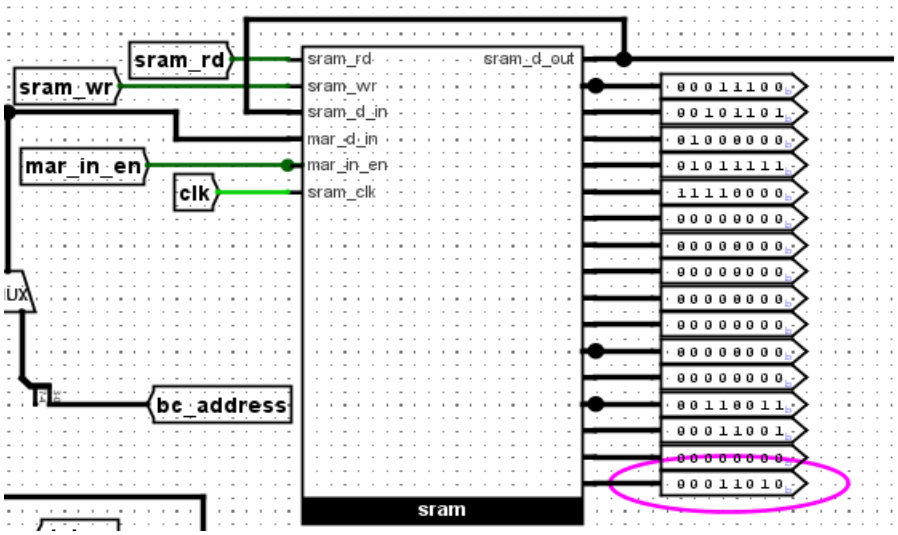


Figure 7.1:(b) Subtraction

7.2 Jump Demo (LDA12, LDB13, then JMP to ADD@5)

- **HEX:**
  1C 2D 65 00 00 30 5F F0 00 00 00 00 33 19 00 00

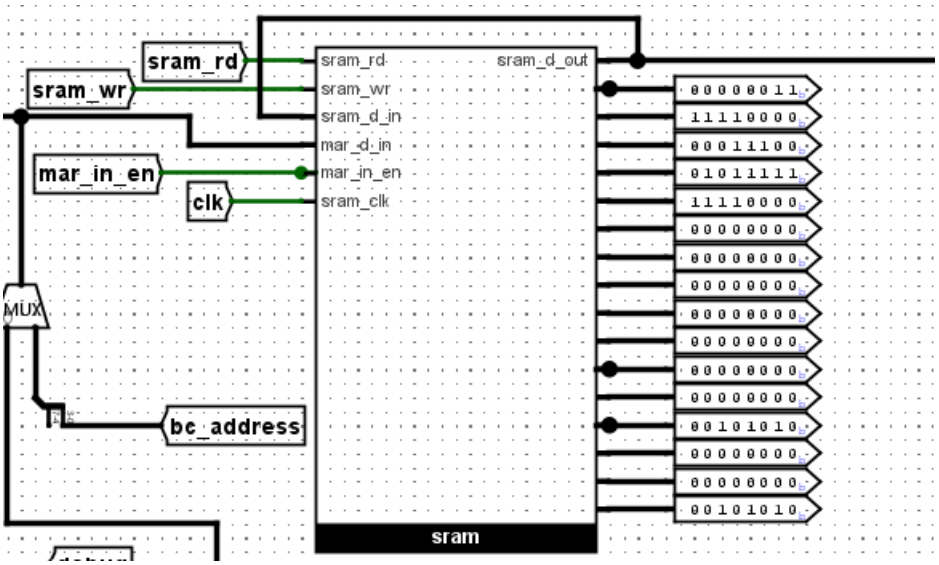


Figure 7.1:(c)Jump

7.3 Shift/Rotate Demos (single-cycle at T4)

- **SHL (logical left by 1)**
- **HEX:** 1C 70 5F F0 00 00 00 00 00 00 00 00 19 00 00 00

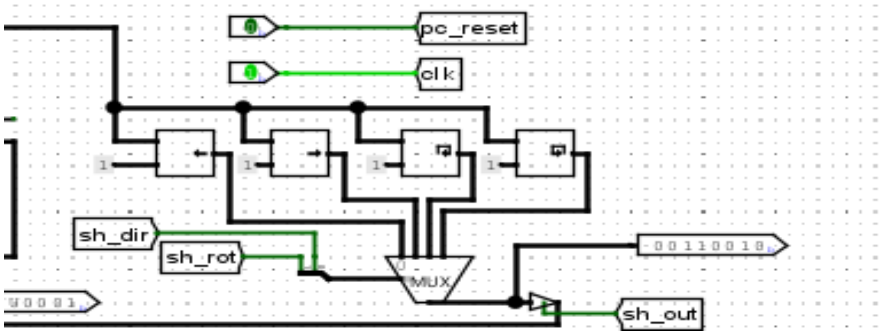


Figure 7.1:(d) shift left

- **SHL ,SHR,ROL,ROR**
- **HEX:** 1C 70 5F 1C 80 5E 1C 90 5D 1C A0 5B 19 00 00 00

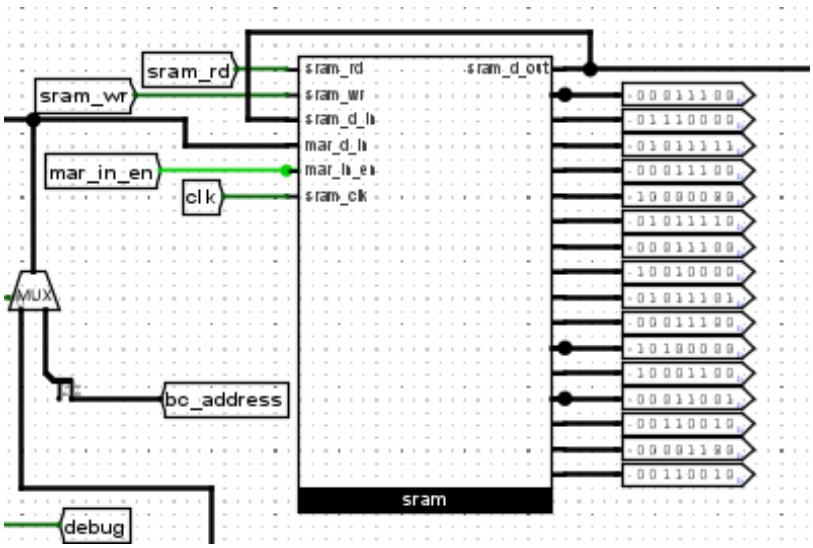


Figure 7.1:(e) SHL, SHR, ROL, ROR

9. Simulation Procedure (Logisim Evolution)

9.1 Running in Automatic Mode

1. Ensure **debug = 0** (run mode).
2. **Reset** PC/RC as your circuit requires (pc_reset, clk_reset).
3. Load the **HEX** into ROM (or prefill RAM).
4. Click **Simulate → Ticks Enabled** or step with **Single Step**.
5. Watch probes for **PC, MAR, IR, A, B, ALU/SH outputs, Bus**.

9.2 Loading in Manual/Loader Mode

1. Set **debug = 1** (enter loader).
2. Enable the boot counter (**bc_en = 1**).
3. On each clock, i1 then i2 fire:
    - i1: MAR ← bc_address
    - i2: RAM[MAR] ← ROM_data

4. After addresses 0…F are written, set **debug = 0** to return to run mode.
   (Run-mode control is masked while debug=1, so no bus fights.)

9.3 Debugging Tips

- **Wire colors:** green=1, blue=0, red=bus contention (two drivers on).
- At any T-state, only **one** of pc_out, sram_rd, ins_reg_out_en, a_out, b_out, alu_out, sh_out may be 1.
- If **JMP** "does nothing," check at **T4**: ins_reg_out_en=1 driving IR[3:0] → bus, and jump_en=1 loading PC.
- For shift/rotate, at **T4**: sh_out=1, a_in=1, no other bus driver high.

10. Results & Verification

10.1 Probe Observations

- **Fetch (all):**
  T1: pc_out=1, mar_in_en=1 → MAR=PC
  T2: sram_rd=1, ins_reg_in_en=1 → IR=RAM[MAR]
  T3: pc_en=1 → PC increments
- **ADD/SUB (T4):** alu_out=1, a_in=1 (and alu_sub=1 only for SUB).
- **STA (T5):** a_out=1, sram_wr=1 → RAM[addr]=A.
- **JMP (T4):** IR[3:0] on bus (ins_reg_out_en=1), jump_en=1 → PC loads target.
- *SH/RO* (T4):** sh_out=1, a_in=1, selects: sh_dir / sh_rot.

10.2 RAM Snapshots (final results at 0x0F)

- **ADD demo:** RAM[0x0F] = **0x4C** (76).
- **SUB demo:** RAM[0x0F] = **0x1A** (26).
- **SHL demo:** RAM[0x0F] = **0x32** (50).
- **ROL demo (0x81):** RAM[0x0F] = **0x03**.
- **ROR demo (0x81):** RAM[0x0F] = **0xC0**.
- **SHR** (0x19 >> 1, zero-fill) → **RAM[0x0F] = 0x0C (12)**

11. Conclusion & Future Work

A compact SAP-1 CPU was implemented with a **hardwired control sequencer** supporting added **shift/rotate** instructions. Automatic and manual/loader modes enable fast testing and safe program loading. Experiments validated correct micro-operation timing and single-driver bus discipline.

Future work: add status flags (Z, C), conditional branches (JZ/JC), multi-byte memory and instructions (immediate loads), microcoded control for easier ISA growth, and an expanded assembler with labels and expressions.