MICROPROCESSORS

and

MICROCONTROLLERS

PROJECT REPORT

on

Design and Implementation of a Custom 8-bit Processor

Pranav Aathrey S | 24BCE5375

Submitted on 5th November, 2025

Abstract

This project presents the design and implementation of a custom 8-bit processor developed to study CPU operation and datapath design.

Built using Logisim Evolution, the processor follows a Harvard architecture with separate ROM and RAM, a 32-bit instruction format, and a compact RISC-based instruction set. It features an ALU, six registers, and 16 I/O pins, executing each instruction in a single cycle.

A C-based assembler was created for program execution. Tests confirmed correct arithmetic, logic, and control operations.

Introduction

This project involves the design of a custom processor built primarily to understand how a CPU executes instructions at the hardware level. Modern processors like ARM or x86 are highly complex, but by creating a simplified version, we can study how data moves through registers, how control logic operates, and how the instruction set architecture (ISA) defines system behavior.

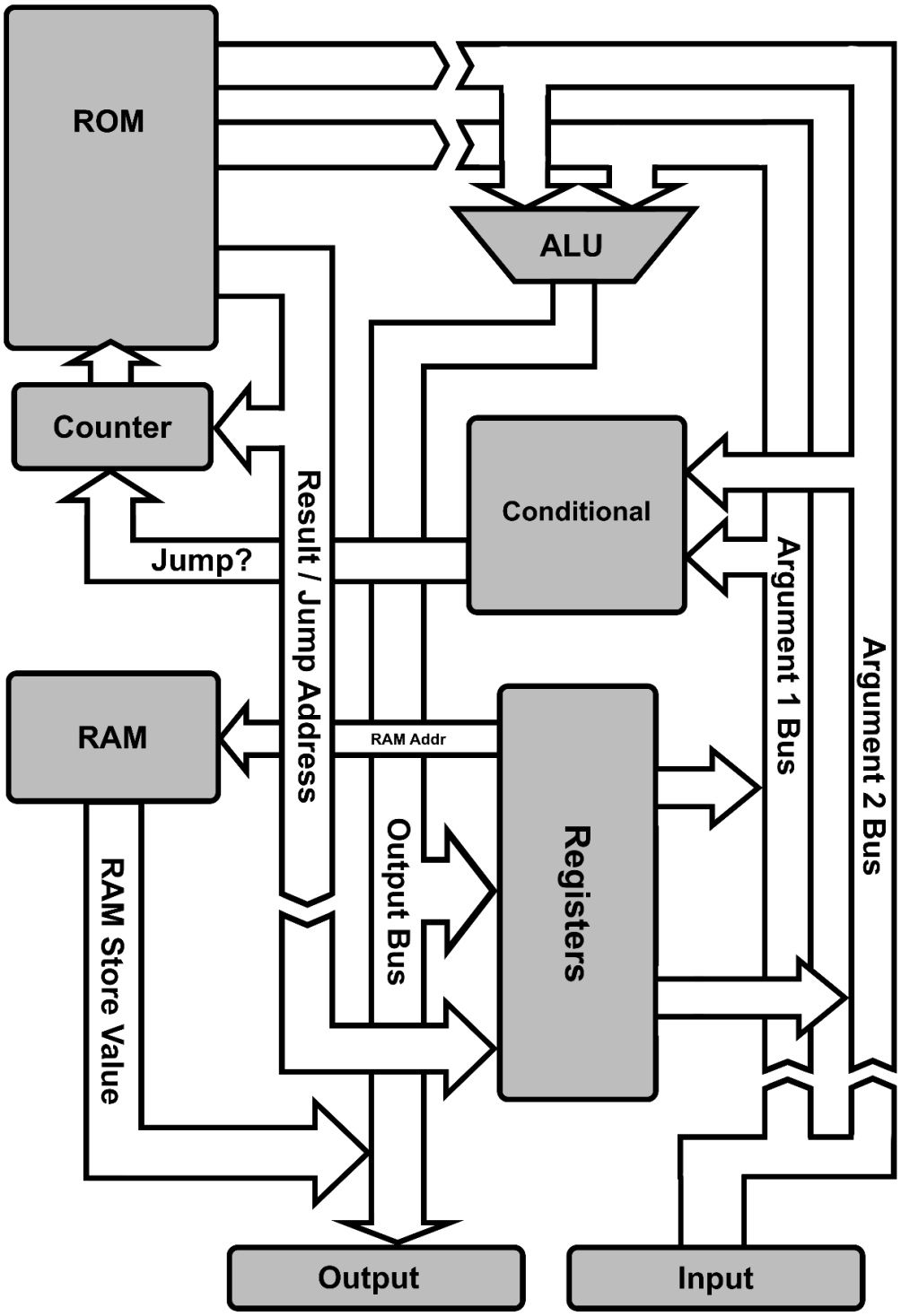
The goal is to design a small, functional processor that models real CPU operations while remaining easy to grasp. It serves as a learning tool to explore instruction decoding, data flow, and control signals in a minimal environment.

The processor is 8-bit, with separate RAM and ROM connected through distinct data and instruction buses – an arrangement inspired by the ARM (Harvard) architecture. Its ISA is 32 bits wide, RISC-based, divided into one byte each for opcode, two arguments, and a result or jump destination.

The processor features 16 I/O pins (8-bit input and 8-bit output) and supports its own assembly language, enabling users to write and run custom programs.

Its simplicity, however, introduces certain limitations: it lacks a stack (restricting functions), uses only 8-bit registers (limiting addressable memory to 256 bytes), has no interrupts, and has a basic control structure without pipelining or caching.

Block Diagram of Architecture



Note: The chevrons in the arrows here (⟩⟩) indicate that there isn't a direct connection here; but a wire that enables a connection (or directly sends in an immediate value).

Architecture Overview

High-level summary

The architecture is an 8-bit datapath, Harvard-style processor with a fixed 32-bit instruction word. Each instruction is fetched from ROM, presented to the core in parallel (all 32 bits at once), decoded, and executed in a single clock cycle.

The ISA uses one opcode byte, two argument bytes, and a final result/jump byte. Two MSBs of the instruction determine whether ARG_1 and ARG_2 are immediate literals or address references.

The design includes 6 writable registers (5 general-purpose + 1 RAM address register), several special-function registers (product high byte, input, output), separate RAM (256 bytes) and ROM (1024 bytes), and 16 I/O pins (8 input, 8 output). Control transfer uses explicit conditional opcodes that evaluate a 1-bit result and, if true, overwrite the ROM program counter (8-bit) with the final instruction byte.

Instruction format (32 bits)

- Total width: **32 bits (4 × 8-bit bytes)**, fetched from ROM and presented in parallel.
- Byte 0 (bits 31–24): **OPCODE** – directly used by decode logic.
- Byte 1 (bits 23–16): **ARG_1** – either an immediate value or an address specifier / register index or a RAM-load token.
- Byte 2 (bits 15–8): **ARG_2** – same encoding possibilities as ARG_1.
- Byte 3 (bits 7–0): **DEST/JUMP** – interpreted as: destination register index / special token (e.g., `ram_save`) or ROM jump destination when executing conditional transfer instructions.

Immediate-mode encoding: the two most significant bits of the instruction are reserved to indicate immediate mode for ARG_1 and ARG_2. When set, the corresponding argument byte is used as the operand value directly; otherwise that byte is interpreted as an index/address to read a value from a register or as a token (e.g., `ram_load`).

All 32 bits are input to the decode stage at clock edge; decode + execute complete in the same cycle (*single-cycle* model).

Registers and special-function registers

Writable registers (6 total):

- **R0 – RAM Address Register (special):** contains an 8-bit address (0–255). When an argument requests `ram_load`, the data bus sources the byte stored in RAM at address R0. When DEST == `ram_save`, ARG_1 is written to RAM at address [R0].
- **R1–R5 – General-purpose registers (GPRs):** 8-bit each, used for arithmetic, logic and data movement.

Additional special-function registers:

- **PROD_HI (product upper byte):** holds the upper 8 bits of a 16-bit multiplication result. Accessed by `mult_hi_byte` opcode (reads only).
- **INPUT_REG:** stores the last input byte sampled from the external 8-bit input pins.
- **OUTPUT_REG:** stores the last value driven to the external 8-bit output pins. This register is indirectly writable.

Register file semantics: all registers are 8-bit wide; read ports must be able to present R1 and R2 values into the argument buses simultaneously within the single cycle.

Memory organization (ROM and RAM)

ROM (program memory):

- *Capacity*: **1024 bytes** (256 instructions × 4 bytes per instruction).
- *Addressing*: The program counter (PC) is an 8-bit counter that increments every clock cycle (mod 256). Each increment represents one instruction, and since each instruction occupies 4 bytes, the PC value is multiplied by 4 to yield the 10-bit ROM address used for fetching the next instruction. Thus, the ROM address space spans 0–

1023, and only the lower 10 bits of the 16-bit product are utilized for addressing the ROM.

- *Fetch model*: full 32-bit word fetched and presented to the instruction bus in a single memory read. There is no decode unit.

RAM (data memory):

- Capacity: **256 bytes** (addressable by 8-bit R0 register).
- Access model: RAM is separate from ROM (Harvard). Reads are triggered when an argument encodes `ram_load` – ALU argument source mux selects RAM output. Writes occur only when DEST == `ram_save`, writing ARG_1 value into RAM at address [R0] at the end of the clock cycle.

I/O interface

- **16 I/O pins** arranged into two 8-bit ports: IN[7:0] and OUT[7:0].
- Not bit-addressable. Must mask bits with `AND` operations.
- Reading input updates INPUT_REG (driven into the argument bus when requested). Writing output updates OUTPUT_REG and also drives the OUT pins.

ALU features and behavioural details

Supported arithmetic operations:

- `add` and `sub` – produce an 8-bit result (mod 256). Overflow/carry bits are discarded (no flags).
- `mult` – full 8×8 multiplication producing a 16-bit product. The low byte may be written to a destination register; the upper byte is stored in PROD_HI and can be accessed via `mult_hi_byte` opcode.
- `div` – division producing an 8-bit quotient (low byte) and storing remainder (upper byte) – consistent with the stated convention: quotient in low byte, remainder in high byte.

Logical operations:

- Bitwise `AND, OR, XOR, NOT` (unary). No shifts/rotates, no barrel shifter.

Status flags:

- **None present.** There is no flag register (no carry, zero, sign, overflow tracking). Conditional operations rely purely on direct comparisons of argument values, performed by the conditional unit.

Implications:

- Arithmetic wraps modulo 256; higher-order effects (carry) are irrecoverable except via `mult` capturing high byte.
- Some algorithms (multi-word arithmetic, precise overflow detection) require software workarounds or use of `mult_hi_byte` where applicable.

Control flow and conditional execution

Conditional unit:

A dedicated comparator evaluates ARG_1 and ARG_2 for relational opcodes (e.g., `if_equal_ri`, `if_greater_rr`, etc.). The comparator outputs a single boolean bit. On true, the instruction's final byte is interpreted as a ROM jump target and the PC is updated to that target for the next cycle; otherwise PC increments normally.

Opcode naming and addressing suffixes:

Convention such as `_ri` means ARG_1 is a register reference and ARG_2 is immediate. The two MSBs of the instruction override these textual suffixes at hardware level by selecting immediate vs address/register sources.

Example walkthrough:

```
if_equal_ri reg_1 39 back
```

This instruction reads R1 into ARG_1 bus, immediate 39 into ARG_2 bus, and the comparator returns 1 if equal. If true, PC <- back (label). Otherwise if false, PC <- PC + 1. The returned bit decides the overwriting of PC.

Datapath and control unit (Harvard, single-cycle)

Harvard datapath characteristics:

- Separate instruction and data buses: ROM fetches instructions while RAM is used for data loads/stores via a different bus.
- Parallelism: ROM provides full instruction in one read; RAM provides data as requested by the argument encoding.

Single-cycle execution model (one clock per instruction):

- **Cycle timeline (conceptual):**
    1. **Fetch:** ROM outputs the 32-bit instruction word onto the instruction bus (instantaneous at the start of cycle).
    2. **'Decode' & read operands:** The circuitry interprets opcode and immediate-mode bits; register file read ports and/or RAM/INPUT_REG are put into ARG_1/ARG_2 buses.
    3. **Execute:** ALU or comparator performs the requested operation using ARG_1 and ARG_2.
    4. **Writeback / Side-effects:** ALU output is written to DEST register or RAM (if ram_save), PROD_HI register updated for multiplications, OUTPUT_REG driven if an output operation occurred, and PC is either incremented or set to DEST/JUMP.

Because all of these steps happen within one cycle, the clock period must be long enough to accommodate the sum of ROM access time + decode + register/RAM read + ALU op + writeback + PC update. This simplifies control logic (no multi-cycle control FSM or pipeline hazards) but forces low clock frequencies or slow cycle rates.

Limitations and design trade-offs

- **8-bit word size** limits addressing and data width; RAM limited to 256 bytes; complex algorithms requiring large memory or wide arithmetic are difficult.
- **No stack** means subroutine calls and recursion are non-trivial; function-like behavior must be implemented via explicit register

convention and manual return-address storage in RAM or a fixed register.

- **Heavy use of Equality-checkers** leaves us with an excess of gates and delay, increasing time for an instruction to execute.
- **No interrupts** meaning interfacing with external I/O components is more power-intensive.
- **No flags** impedes efficient conditional arithmetic chaining; comparisons are explicit operations.
- **Single-cycle execution** simplifies design but reduces clock throughput; the cycle time is dominated by the slowest combined path.

Implementation

Platform and design methodology

The entire processor architecture was designed and implemented using **Logisim Evolution**, a digital logic simulation environment which has sufficient tools for building basic custom CPUs. The program's ability to simulate and visualize signal changes in real time proved invaluable during the debugging and refinement stages.

The initial design began with the **core datapath**, consisting of the ALU, the registers, the conditional unit, and bus configuration. Each component was created as an independent subcircuit, tested in isolation, and then integrated into the main processor layout. After that, the **control logic** responsible for instruction decoding and sequencing was added. Finally, the ROM and RAM interconnection logic was implemented to complete the Harvard architecture design.

Clocking and timing considerations

Clocking was an important aspect of the simulation. The processor was generally run at **64 Hz**, a frequency that provided a balance between observability and responsiveness. At this rate, it was possible to watch

individual instructions being fetched, decoded, and executed within Logisim's visual interface. This slow and steady clock allowed for detailed analysis of data propagation, instruction flow, and bus activity. It also made it easier to detect logical errors, such as unintended bus contention or incorrect register writes.

Because the processor follows a **single-cycle execution model**, every instruction — including arithmetic, logic, memory, and conditional operations — completes in exactly one clock cycle. This design choice, while limiting achievable clock frequency in hardware, made the simulation straightforward and allowed for clear visualization of the full instruction execution path within each cycle.

Assembly Language Instruction Set

The processor's assembly language provides a structured, readable interface for programming the architecture directly. It uses mnemonic keywords that map to specific binary opcodes, which the assembler translates into executable machine code. Each instruction corresponds to a single operation in the processor's control logic, allowing for precise control of arithmetic, logic, memory, and control-flow operations.

The assembler—implemented in C (referenced in the appendix)—processes assembly programs written in plain-text .txt files, which can include **labels**, **comments**, and **constant definitions**. It outputs a .bin binary file suitable for direct loading into the processor's ROM, enabling straightforward simulation and testing.

Each instruction typically follows the format:

```
<OPCODE> <ARG_1> <ARG_2> <DEST>
```

Depending on the operation, arguments may represent **registers**, **immediate values**, or **memory locations**. Variants with suffixes such as `_ii`, `_ri`, and `_ir` denote whether operands are **immediate** (`i`) or **register-based** (`r`), but in this description, these suffixes are omitted to focus on base operations.

Arithmetic Instructions

These perform basic and compound arithmetic between registers and/or immediate values.

Mnemonic	Description	Opcode (Hex)	Binary
`add`	Adds two arguments, result stored in destination register	00	0000 0000
`sub`	Subtracts `ARG_1` from `ARG_2`, stores in destination	01	0000 0001
`mult`	Multiplies arguments, storing low byte of result	0E	0000 1110
`mult_hi_byte`	Retrieves high byte of multiplication result	0F	0000 1111
`div`	Divides `ARG_1` by `ARG_2`, quotient in destination	10	0001 0000
`mod`	Puts the remainder of division of the arguments in destination	11	0001 0001

Arithmetic variants exist for immediate and register combinations (e.g., add_ri, add_ir), but operationally they share the same semantics. The two most significant bits in the Opcode indicate whether they're immediate or not: 7th bit high, ARG_1 immediate, and so on.

Logical Instructions

The Logical operations here manipulate binary data for bitwise computation and control.

Mnemonic	Description	Opcode (Hex)	Binary
`AND`	Bitwise AND between arguments	02	0000 0010
`OR`	Bitwise OR between arguments	03	0000 0011
`NOT`	Bitwise negation of `ARG_1` (unary)	04	0000 0100
`XOR`	Bitwise exclusive OR	05	0000 0101

When it comes to immediate addressing, it is a similar story for these Opcodes and all that follow this as well.

Conditional and Control Flow Instructions

These govern program control and conditional branching, forming the backbone of decision-making logic. (DEST is jump address here)

Mnemonic	Description	Opcode (Hex)	Binary
`if_equal`	Branches if arguments are equal	20	0010 0000
`if_nt_eql`	Branches if arguments are not equal	21	0010 0001
`if_less`	Branches if `ARG_1` < `ARG_2` (unsigned)	22	0010 0010
`if_le_eq`	Branches if `ARG_1` ≤ `ARG_2` (unsigned)	23	0010 0011
`if_more`	Branches if `ARG_1` > `ARG_2` (unsigned)	24	0010 0100
`if_mo_eq`	Branches if `ARG_1` ≥ `ARG_2` (unsigned)	25	0010 0101
`if_less_s`	Signed comparison: branch if less	26	0010 0110
`if_le_eq_s`	Signed comparison: branch if ≤	27	0010 0111
`if_more_s`	Signed comparison: branch if >	28	0010 1000
`if_mo_eq_s`	Signed comparison: branch if ≥	29	0010 1001
`jump`	Unconditional branch to target address	E0	1110 0000
`end`	Marks program termination	3A	0011 1010

Memory Management

These handle data movement between registers and memory.

Mnemonic	Description	Opcode (Hex)	Binary
`copy`	Copy data between registers	40	0100 0000
`copy_i`	Copy immediate value into register	C0	1100 0000
`ram_load`	Load data from RAM into register	3F	0011 1111
`ram_save`	Save register data into RAM (only used as a destination)	3E	0011 1110

There aren't any immediate-value modifiers for these. (except `copy_i`)

Address and Register Mapping

The architecture features a small register set and fixed I/O addresses for predictable low-level control.

Label	Address (Hex)	Description
`reg_0` (or) `ram_address`	00	Stores the current RAM address, points to data in RAM.
`reg_1–reg_5`	01–05	General-purpose registers
`counter`	06	Loop or iteration counter
`input`	07	Input port
`output`	07	Output port (shared address)

Syntax Features and Assembly Structure

The assembler supports the following syntax elements:

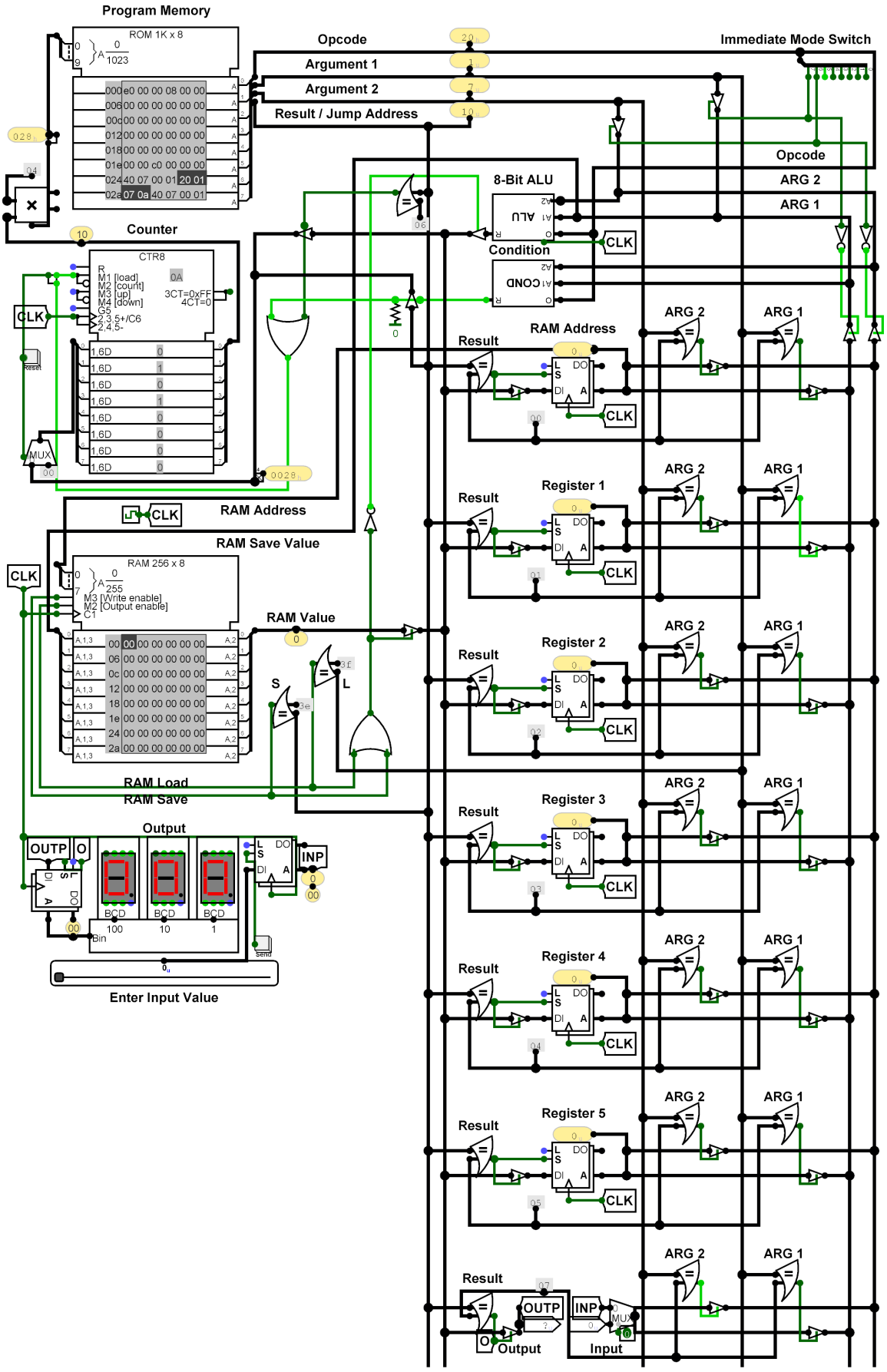
- **Labels:** Define code locations for jumps and calls.
  For example: `label jump_here`
- **Comments:** Begin with a delimiter (e.g., // or #) and are ignored by the assembler. For example: `# this is a comment`
- **Constants:** Defined using keywords or macros for improved readability. For example: `const pi 3`

These are all ***Assembly directives***, meaning they will not show up as bytes in the assembled binary that is to be flashed on to the ROM.

Another feature is **filler keywords**, such as `to`, `from`, `loc`, etc – which are used when there is no need for a keyword in that spot in the instruction line (which has a fixed width of 4 bytes). The assembler simply interprets this as to insert `00` in the assembled binary.

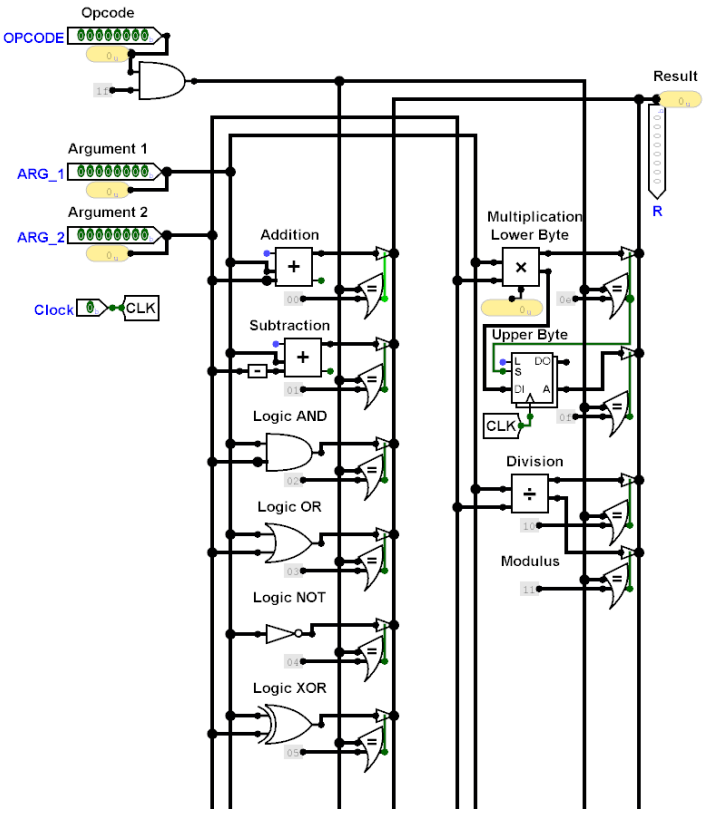
Once assembled, the output binary file can be loaded into ROM, where the processor sequentially fetches and executes instructions. This design allows developers to write compact, readable assembly code that interfaces cleanly with the processor's control logic and memory subsystems.

Architecture Implementation in Logisim

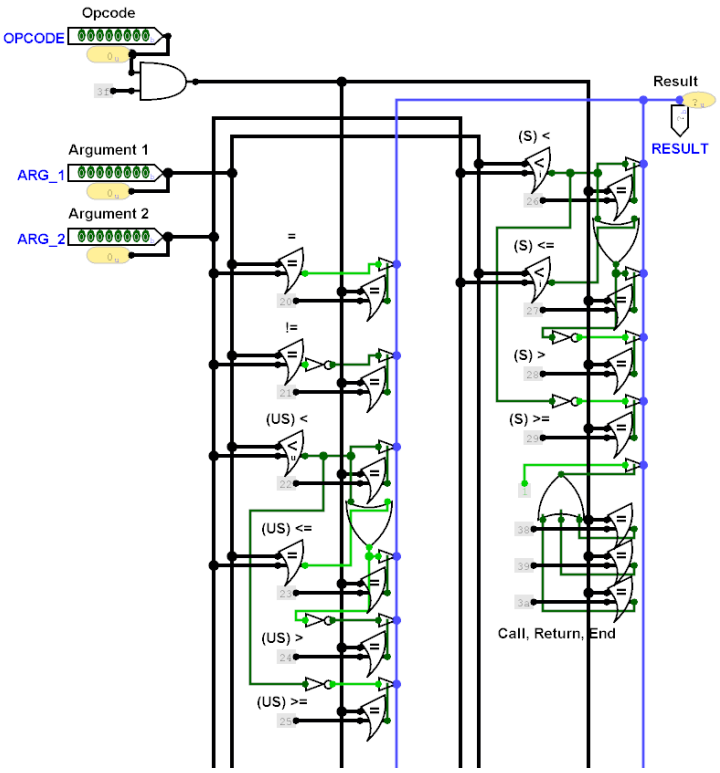


Implementation of Architectural Components

The Arithmetic Logic Unit (ALU)



The Conditional Unit (COND)



Testing approach and verification

Testing was carried out primarily through iterative simulation in Logisim. After each major subsystem was added (ALU, register file, RAM/ROM interface, conditional unit), targeted tests were performed using small assembly snippets to verify functionality. Once the processor achieved full operational stability, more complex programs were developed to demonstrate its real-world behavior.

Two key assembly programs were used for validation and demonstration:

(i) Bubble Sort Algorithm:

This program highlights the processor's memory management and conditional comparison features. It requires repeated use of the RAM address register, conditional branching, and arithmetic operations, effectively exercising the datapath and control unit.

The Assembly code for it:

```
# load to-be-sorted values into RAM
label load_RAM
      copy_i 0 to ram_address
      copy_i 15 to ram_save
      add_ri ram_address 1 ram_address
      copy_i 14 to ram_save
      add_ri ram_address 1 ram_address
      copy_i 13 to ram_save
      add_ri ram_address 1 ram_address
      copy_i 12 to ram_save
      add_ri ram_address 1 ram_address
      copy_i 11 to ram_save
      add_ri ram_address 1 ram_address
      copy_i 10 to ram_save
      add_ri ram_address 1 ram_address
      copy_i 9 to ram_save
      add_ri ram_address 1 ram_address
      copy_i 8 to ram_save
      add_ri ram_address 1 ram_address
      copy_i 7 to ram_save
      add_ri ram_address 1 ram_address
      copy_i 6 to ram_save
      add_ri ram_address 1 ram_address
      copy_i 5 to ram_save
      add_ri ram_address 1 ram_address
      copy_i 4 to ram_save
      add_ri ram_address 1 ram_address
      copy_i 3 to ram_save
      add_ri ram_address 1 ram_address
      copy_i 2 to ram_save
      add_ri ram_address 1 ram_address
      copy_i 1 to ram_save
```

```
# our sorted flag
copy_i 0 to reg_1

# we're doing bubble sort
label sort_items
      # check if sorted
      if_equal_ri reg_1 1 exit

      copy_i 0 to ram_address
      # initially assume it as sorted
      copy_i 1 to reg_1

      label inner_loop
            if_equal_ri ram_address 14 sort_items

            # get our concerned values
            copy ram_load to reg_2
            add_ri ram_address 1 ram_address
            copy ram_load to reg_3

            # check if they're in order
            if_le_eq reg_2 reg_3 continue
            # otherwise, swap them
            sub_ri ram_address 1 ram_address
            copy reg_3 to ram_save
            add_ri ram_address 1 ram_address
            copy reg_2 to ram_save
            # update flag to swapped (= 0)
            copy_i 0 to reg_1

      label continue
            jump to loc inner_loop

label exit
      jump to loc exit
```

(ii) Find Maximum Among 10 Inputs:

This program demonstrates I/O interaction and decision-making capabilities. Inputs are to be read through the input register, compared sequentially, and the maximum value was written to the output register.

The Assembly code for it:

```
label initialize
      copy_i 0 to ram_address
      copy input to reg_1

# getting the input values
label check_input
      if_equal reg_1 input check_input
      # otherwise,
      copy input to reg_1
      copy reg_1 to ram_save
```

```
      add_ri ram_address 1 ram_address
      if_equal_ri ram_address 10 assume_max
      # else
      jump to loc check_input


# actually finding the maximum
label assume_max
      copy_i 0 to ram_address
      # holds our current biggest
      copy ram_load to reg_1

label find_max
      add_ri ram_address 1 ram_address
      if_equal_ri ram_address 10 out

      # comparing values
      copy ram_load to reg_2
      if_more reg_2 reg_1 new_max
      jump to loc find_max

label new_max
      # update new maximum
      copy ram_load to reg_1
      jump to loc find_max

label out
      copy_i 0 to ram_address
      copy reg_1 to output

label exit
      jump to loc exit
```

Each test program was flashed into ROM, and results were verified by monitoring register and memory contents after execution.

The ability to observe bus states, register outputs, and control line transitions in Logisim was crucial in ensuring that every instruction executed exactly as intended.

Conclusion

The custom 8-bit processor represents a basic implementation of a functional CPU architecture, realized from specification to simulation in Logisim Evolution.

The design includes a 32-bit instruction format, a compact register file, a functional ALU, and distinct RAM and ROM modules – illustrating a clear Harvard architecture model. The processor successfully executed assembly-level programs, demonstrating arithmetic operations, control flow, and memory access.

A minimal yet comprehensive *instruction set architecture* (ISA) was developed, supporting arithmetic, logic, branching, and memory operations with both immediate and register addressing. A C-based assembler translated assembly programs into binary code, integrating software and hardware design for seamless execution within the simulation environment.

The project provided insight into datapath organization, control timing, clock synchronization, and modular design. Component-level testing of the ALU, register file, memory, and control unit reflected standard digital system workflows. Simulation clarified how instructions propagate through the datapath and how control logic governs execution and branching.

The key challenges while designing this architecture – RAM addressing, instruction decoding, and synchronization – offered practical understanding of architectural trade-offs. Potential extensions include pipelining, cache integration, I/O control, and advanced features such as interrupts and stack management to enhance realism and performance.

Appendix

The source code for the Assembler made for this architecture, along with all the Logisim evolution files, assembly language programs, and assembled binaries for those files can be found in the following repo:

https://github.com/pranavaathrey/Processor-Architecture