Solix-16 design and ISA

Sean Kim

November 17, 2025

Contents

1 Architecture Overview

This document provides a comprehensive technical specification for a custom 16-bit Central Processing Unit (CPU). The architecture is designed as a streamlined, modular processor utilizing a 16-bit data path and a focused instruction set.

The design adheres to a **Harvard Architecture** model, utilizing separate physical memory spaces for instructions and data to maximize throughput and simplify the control logic. Instructions are fetched from a dedicated Read-Only Memory (ROM), while data operations target a separate Static RAM (SRAM). While primarily intended for educational verification, the rigorous digital logic principles employed allow for future scalability into pipelined or multi-cycle implementations.

1.1 Register File Organization

The processor utilizes a bank of 11 internal registers, each 16 bits in width. The register file is mapped using a 4-bit addressing scheme (supporting up to 16 registers), with addresses 0xB through 0xF reserved for future architectural extensions.

Note: Reads from reserved addresses (0xB-0xF) return undefined values; writes are ignored.

The register allocation is defined as follows:

Address	Mnemonic	Function Description
0x0	**R0**	**Constant Zero:** Hardwired to $0x0000$. Writes are ignored.
0x1 – 0x7	**R1 – R7**	**General Purpose Registers (GPR):** Used for data manipulation.
0x8	**R8 (SP)**	**Stack Pointer:** Points to the top of the current stack frame.
0x9	**R9 (PC)**	**Program Counter:** Holds the address of the next instruction.
0xA	**R10 (FLAGS)**	**Status Register:** Stores condition codes (Z, N, C, O).
0xB – 0xF	**Reserved**	**Future Use:** Undefined behavior if accessed.

Table 1: CPU Register Map

1.2 Memory Architecture

This CPU employs a true Harvard Architecture with distinct physical memories and buses:

- **Instruction Memory (ROM):** A 4K word (4096 × 16-bit) Read-Only Memory stores the system program. It is addressed directly by the Program Counter (R9) via a dedicated 12-bit instruction address bus.
- **Data Memory (SRAM):** A 4K word (4096 × 16-bit) Static RAM stores runtime variables and the stack. It is accessed via a separate data address bus, driven by register values (for LD/ST operations) or the Stack Pointer (R8).

Memory Map:

Address Range	Memory Type	Purpose
0x000 – 0xFFF	ROM (Instruction)	Program code (4096 words)
0x000 – 0xFFF	SRAM (Data)	Variables, stack, heap (4096 words)

Table 2: Harvard Architecture Memory Map

Note: ROM and SRAM share the same address range but are accessed via separate buses. There is no address space conflict.

1.3 Arithmetic Logic Unit (ALU)

The computational core of the CPU is the ALU, which supports 8 distinct operations. The ALU accepts 16-bit operands and generates a 16-bit result alongside relevant status flags. The operation set covers arithmetic, bitwise logic, and shifting capabilities:

- **Arithmetic:** `ADD`, `SUB` (Two's complement).
- **Logic:** `AND`, `OR`, `XOR`, `NOT`.
- **Shift:** `SHL` (Logical Left), `SHR` (Logical Right).

1.4 Design Philosophy

The architecture prioritizes modularity to facilitate debugging and expansion. Key design goals include:

- **Determinism:** Usage of a hardwired zero register (R0) simplifies instruction logic.
- **Observability:** Dedicated Status (R10) and PC (R9) registers allow for transparent control flow monitoring.
- **Scalability:** The 4-bit register addressing space leaves room for future expansion (addresses 0xB to 0xF are reserved).

2 Register File Architecture

The central storage element of the CPU is a unified register file consisting of 11 directly addressable 16-bit registers, with 5 addresses reserved for future expansion.

2.1 Hardware Implementation

Access to the register file is managed through a dedicated decoding and multiplexing network:

- **Read Access:** Operand retrieval is handled by a **16-to-1 Multiplexer**, allowing the contents of any register to be driven onto the primary data bus for ALU operations. Reads from reserved addresses (0xB-0xF) return undefined values (implementation-dependent, typically the last valid register or all zeros).
- **Write Access:** Register updates are controlled by a **4-to-16 Decoder**. When the global `Write Enable` (WE) signal is asserted, the decoder routes the data to the specific register selected by the 4-bit destination address. Write enable signals to reserved addresses (0xB-0xF) are not connected and have no effect.

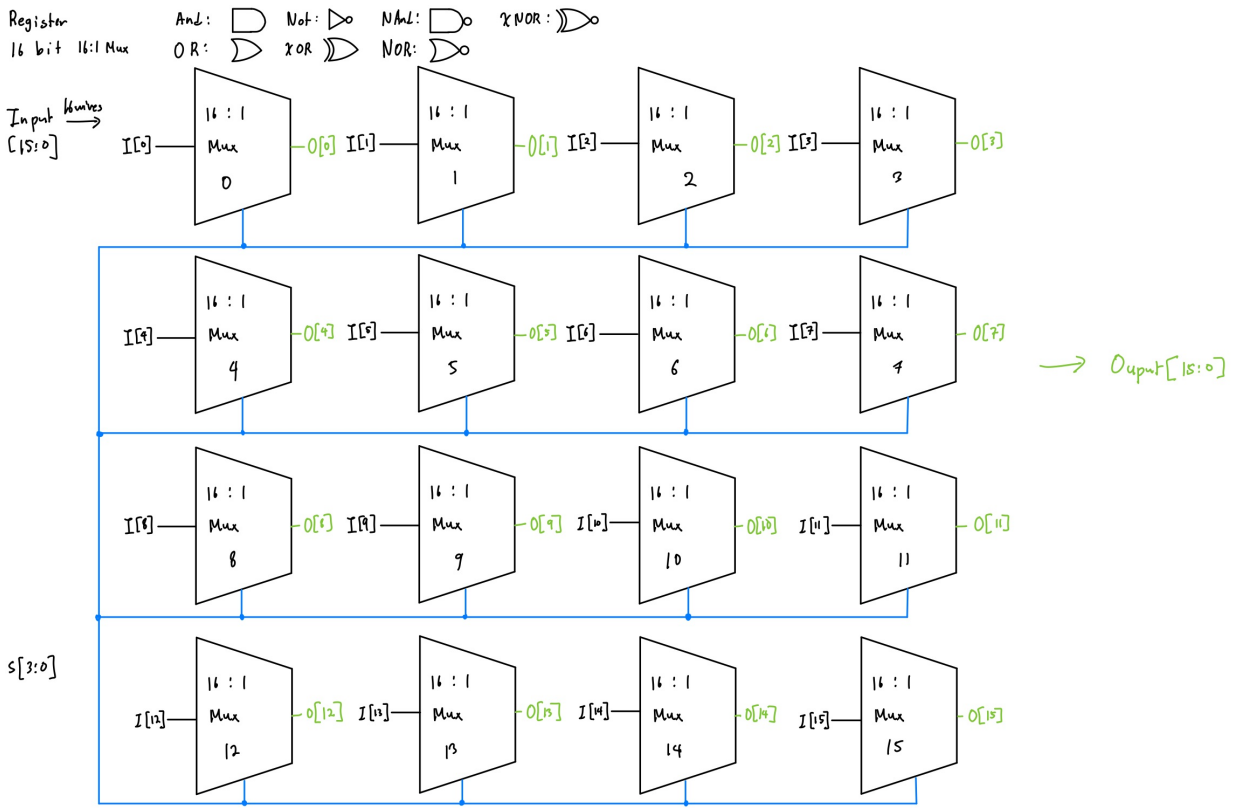

Figure 1: 16-bit Register Read Multiplexer Logic

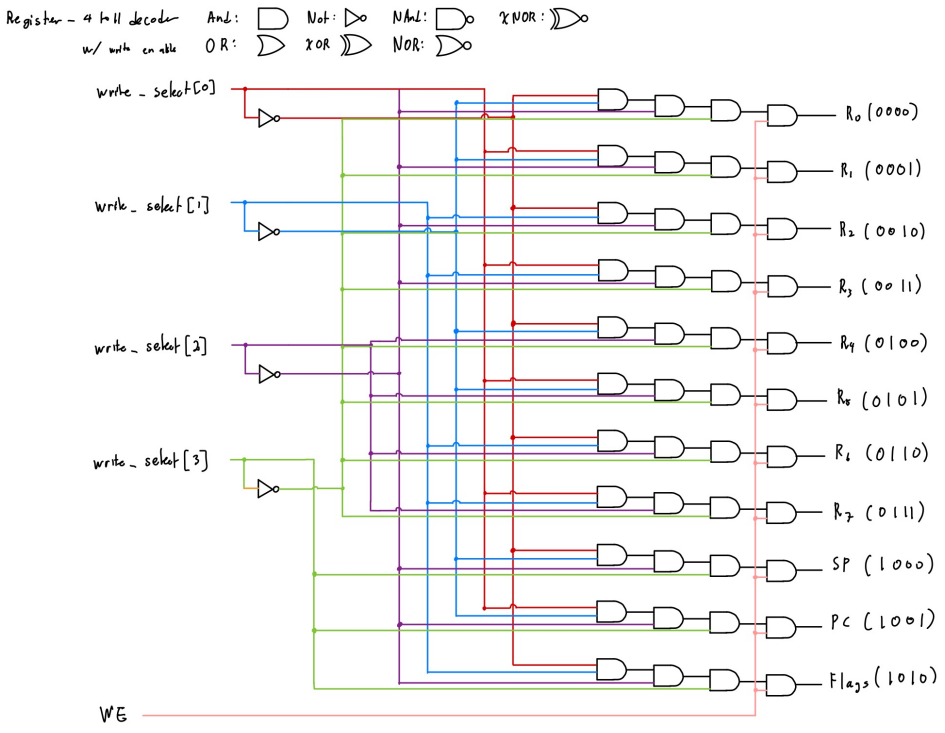

Figure 2: Register Write Decoder Logic

2.2 Register Map

The following table details the allocation of the 4-bit address space. Note that addresses 11 through 15 ($0xB$ to $0xF$) are reserved for future architectural extensions.

Register	Type	Functional Description
r0	Const	**Hardware Zero:** Hardwired to 0x0000 (Read-only)
r1	GPR	General-Purpose Data Register
r2	GPR	General-Purpose Data Register
r3	GPR	General-Purpose Data Register
r4	GPR	General-Purpose Data Register
r5	GPR	General-Purpose Data Register
r6	GPR	General-Purpose Data Register
r7	GPR	General-Purpose Data Register
r8	SP	**Stack Pointer:** Memory address for stack operations
r9	PC	**Program Counter:** Instruction memory pointer
r10	Flags	**Status Register:** ALU condition codes (Z, N, C, O)
0xB-0xF	Reserved	**Future Use:** Reads undefined, writes ignored

Table 3: CPU Register Configuration and Assignments

2.3 Register Definitions

2.3.1 Constant Register (r0)

Register `r0` is strictly hardwired to logical zero ($0x0000$). Any write operations targeting `r0` are ignored by the hardware. This register is essential for:

- Clearing other registers (e.g., `ADD r1, r0, r0`).
- Implementing No-Operation (NOP) instructions.
- Providing a distinct zero value for comparisons.

2.3.2 General-Purpose Registers (r1–r7)

Registers `r1` through `r7` serve as the primary accumulators for the CPU. They are fully read-/write accessible and act as the source and destination operands for all arithmetic, logical, and shift instructions executed by the ALU.

2.3.3 Special-Purpose Registers

The architecture includes three dedicated registers for control flow and state management.

The underlying physical implementation for these registers relies on D-Flip Flops with Write Enable capabilities, as shown below.

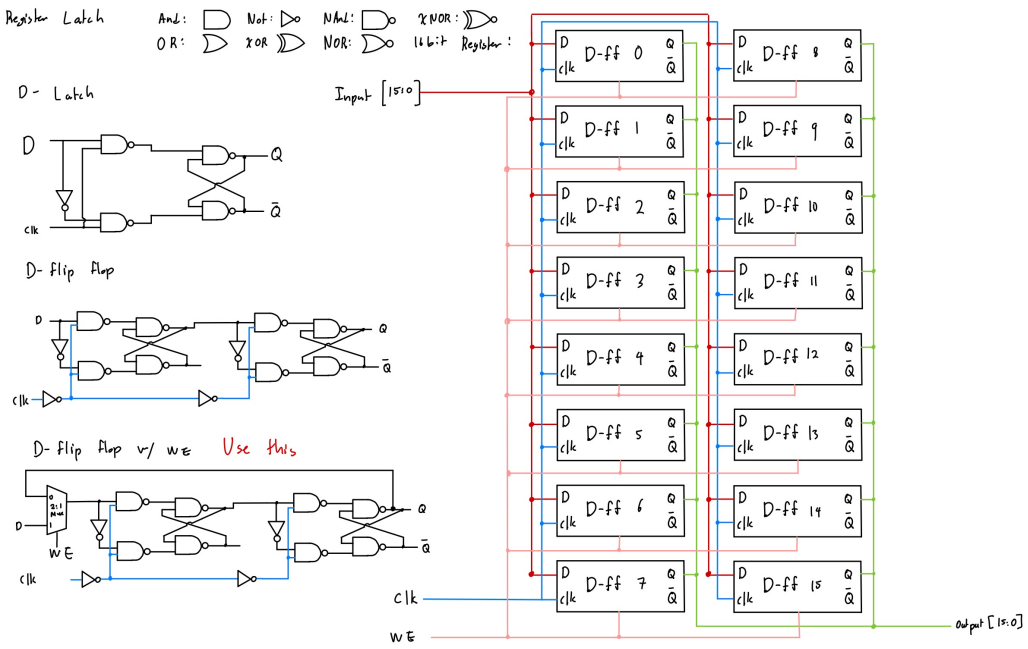

Figure 3: D-Latch and D-Flip Flop Schematics used for Register Storage

- **Stack Pointer (r8):** Maintains the memory address of the current top-of-stack. It is automatically decremented during `PUSH` operations and incremented during `POP` operations (if supported by the ISA). *Note: Current ISA does not include PUSH/POP; SP must be managed manually via ADD/SUB operations.*
- **Program Counter (r9):** Stores the address of the next instruction to be fetched. It is automatically incremented after every fetch cycle, or modified directly by branch and jump instructions.
- **Flags Register (r10):** A bit-field register where specific bits represent the status of the last ALU operation:

2.4 Flags Register Bit Layout

Bit	**15:4**	**3**	**2**	**1**	**0**
Name	Reserved	O	C	N	Z
Description	Always 0	Overflow	Carry	Negative	Zero

Table 4: Flags Register (r10) Bit Assignment

- **Bit 0 (Z - Zero):** Set to 1 if the ALU result equals 0x0000.
- **Bit 1 (N - Negative):** Set to 1 if bit 15 (MSB) of the result is 1, indicating a negative value in two's complement.
- **Bit 2 (C - Carry):** Set to 1 if an arithmetic operation produces a carry-out from bit 15, or if a shift operation shifts out a '1'.
- **Bit 3 (O - Overflow):** Set to 1 if a signed arithmetic operation produces a result outside the representable range (-32768 to +32767).

- **Bits 15:4:** Reserved for future use. Always read as 0.

3 Data Memory Architecture (SRAM)

The CPU's data storage is a **4K × 16-bit Static RAM (SRAM)** module. In the Harvard Architecture design, this memory is dedicated exclusively to data storage, separate from the instruction ROM.

3.1 SRAM Rationale

SRAM was selected over DRAM for this design due to its simplicity and performance characteristics.

- **Simplicity:** SRAM does not require complex refresh controller logic, simplifying the overall CPU design.
- **Fast Access:** It provides low-latency access, which is essential for the single-cycle execution model.
- **Synchronous Fit:** As a synchronous device, it interfaces cleanly with the CPU's global clock.

3.2 Memory Organization

The memory is organized as 4096 words, with each word being 16 bits wide.

- **Data Bus:** 16 bits, matching the CPU's internal datapath.
- **Depth:** 4096 words (4K).
- **Address Lines:** Accessing 4096 unique locations requires $\log_2(4096) = 12$ address lines, labeled `Addr[11:0]`.

The data address bus is driven by register values (via the ALU) for `LD` and `ST` operations, or by the Stack Pointer (R8) for stack operations.

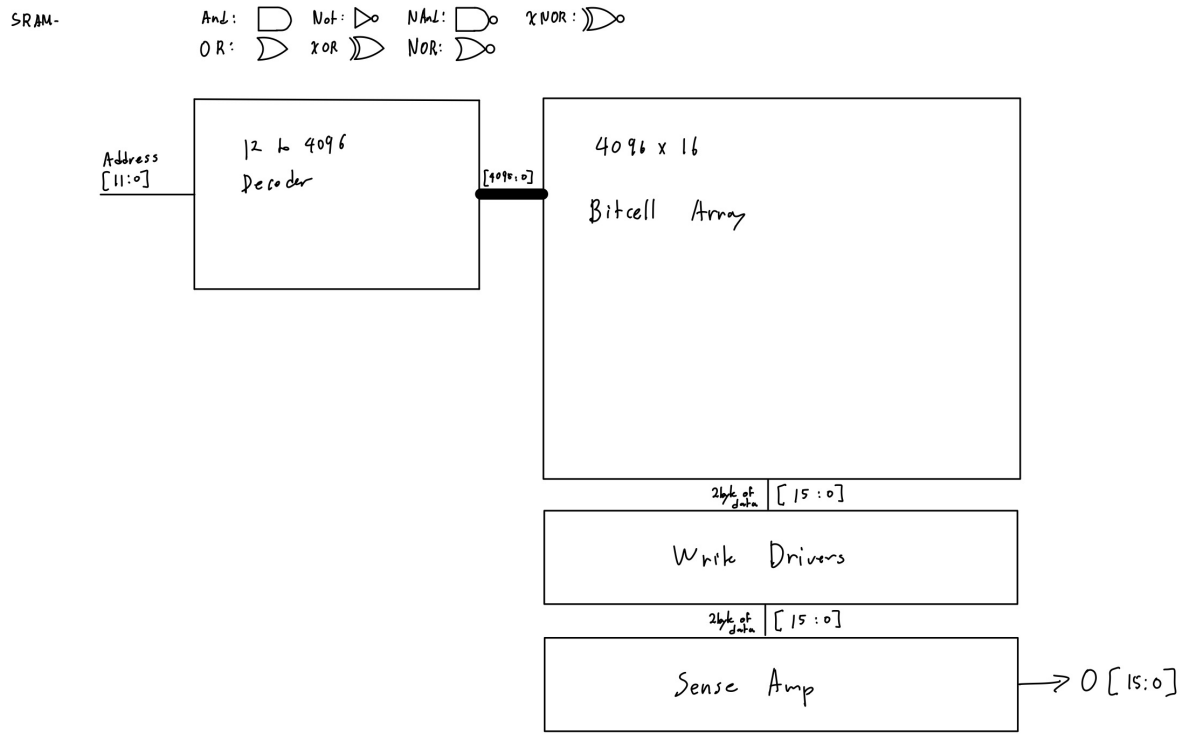

Figure 4: 4096 x 16-bit SRAM Architecture and Interface

3.3 Read/Write Logic

Access is controlled by three primary signals: the 12-bit `Addr` bus, `MemWrite` (Write Enable), and `MemRead` (Read Enable).

- **Write Operation:** A write is synchronous. When `MemWrite` is asserted (1), the 16-bit value on the data bus is written to the location specified by `Addr[11:0]` on the next rising clock edge.
- **Read Operation:** A read is asynchronous. When `MemRead` is asserted (1), the contents of the memory at `Addr[11:0]` are combinationally placed on the data bus. The CPU must wait for the SRAM's access time (T_{acc}) before latching this data.
- **Port:** The design assumes a single-port SRAM. Only one read or one write operation can be performed in a single clock cycle.

3.4 Memory Address Sources

The 12-bit SRAM address can come from multiple sources:

- **LD instruction:** Address = contents of register R_s
- **ST instruction:** Address = contents of register R_s
- **Future stack operations:** Address = Stack Pointer (r8)

3.5 Future Extensions

- **Memory-Mapped I/O (MMIO):** A high-address range (e.g., `0xF00` – `0xFFF`) could be reserved, allowing the CPU to control peripherals by simply reading from or writing to these "memory" addresses.
- **Cache:** A small L1 data cache could be added to improve memory access performance, especially if a slower, larger main memory is added.
- **Stack Operations:** Add dedicated `PUSH` and `POP` instructions that automatically manage the Stack Pointer (r8).

4 Read-Only Memory (ROM) Architecture

The processor's instruction fetch mechanism interfaces with a **16-bit Read-Only Memory**. Unlike the SRAM, the ROM requires no write drivers, simplifying the datapath to a purely combinational read operation.

4.1 ROM Block Diagram

The internal structure consists strictly of an address decoder and the bitcell array, feeding directly into the output buffers.

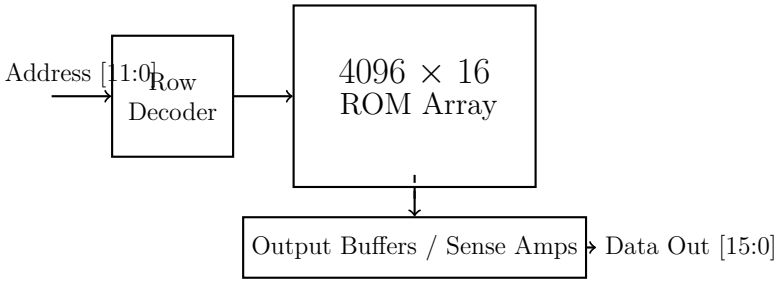

Figure 5: ROM Internal Architecture (Read-Only)

4.2 Purpose and Use Case

The ROM serves as the permanent program store for the CPU. In a typical application, the ROM holds the main executable program, a system bootloader, or a fixed-function operating system. Its non-volatile nature ensures that the program is retained when system power is lost.

4.3 Memory Organization

To seamlessly integrate with the CPU's datapath, the ROM is organized to match the 16-bit instruction width.

- **Width:** 16 bits, matching the instruction word size.
- **Depth:** The architecture supports up to 4K words (4096 instructions).
- **Address Lines:** To address 4K unique locations, $\log_2(4096) = 12$ address lines (`Addr[11:0]`) are required.

- **Access:** The memory is strictly Read-Only. As such, it does not require a Write Enable (`WE`) signal or any write logic, simplifying its interface and the CPU's control unit.

4.4 Read Logic

The ROM employs a simple asynchronous read logic, which minimizes latency.

- **Asynchronous Read:** The device is purely combinational. The 16-bit data word stored at the location specified by the `Addr[11:0]` bus is immediately presented on the 16-bit `Data_Out` bus.
- **Timing:** No clock signal is required for the ROM block itself. The access time (T_{rom_acc}) is determined by the propagation delay from the moment the address lines are stable to the moment the data output is stable.

4.5 Instruction Addressing Constraints

The 12-bit address bus for the ROM is a direct consequence of the 16-bit Instruction Set Architecture (ISA).

- Instruction Width: 16 bits
- Opcode Width: 4 bits
- **Available Address Field:** 16 bits − 4 bits = 12 bits

This 12-bit field, used in `JMP`, `JZ`, and `JNZ` instructions, perfectly maps to the 12-bit address bus. This design choice unifies the instruction format with the physical memory map, allowing any J-Type instruction to target any location within the 4K ROM address space.

4.6 System Integration

The ROM is a central component of the CPU's "Fetch" stage.

1. The 12-bit **Address Bus** of the ROM is directly connected to the output of the **Program Counter (*PC*, r9)**.
2. The 16-bit **Data Bus** of the ROM is connected directly to the input of the **Instruction Register (IR)**, which holds the instruction to be decoded.

4.7 Optional Extensions

- **Flash Memory:** For development and prototyping, the ROM could be replaced with a 16-bit wide, parallel-interface Flash memory chip. This would allow the program to be re-written without fabricating a new ROM.
- **Hybrid Memory System:** A common and powerful extension is a hybrid model. A small ROM (e.g., 256 words) would hold a "bootloader" program. Upon reset, this bootloader would execute and copy the main program from a slower, larger storage device (like Flash or an SD card) into the 4K SRAM. The CPU would then remap its instruction fetch address to the SRAM and execute the main program from there.

5 Instruction Set Architecture (ISA)

The CPU operates on a fixed-length **16-bit instruction word**. The ISA is designed to minimize decoding complexity, utilizing a 4-bit Opcode field located in the most significant bits (MSB) of the instruction.

5.1 Instruction Formats

To optimize the utilization of the 16-bit bus, instructions are categorized into three encoding formats. The bitwise allocation for each format is defined as follows:

R-Type (Register Operations)
Used for arithmetic, logical, and memory access operations involving register operands.

15 12	**11 8**	**7 4**	**3 0**
Opcode (4)	R_d (4)	R_s (4)	R_t (4)

Field Definitions:

- **Opcode [15:12]:** 4-bit operation code
- **Rd [11:8]:** Destination register address (where result is written)
- **Rs [7:4]:** First source register (or memory address for LD/ST)
- **Rt [3:0]:** Second source register (or data source for ST)

Special Cases:

- **LD instruction:** `LD Rd, Rs` $\rightarrow R_d \leftarrow Mem[R_s]$. The Rt field [3:0] is unused and should be set to 0000.
- **ST instruction:** `ST Rs, Rt` $\rightarrow Mem[R_s] \leftarrow R_t$. Note: Rd field [11:8] contains Rs (address register), Rs field [7:4] also contains Rs, Rt [3:0] contains data register.
- **Unary operations (NOT, SHL, SHR):** Only use Rd and Rs. Rt field [3:0] is ignored and should be set to 0000.

I-Type (Immediate Operations)
Used for loading constants into registers. The 8-bit immediate value is zero-extended to 16 bits.

15 12	**11 8**	**7 0**
Opcode (4)	R_d (4)	Immediate (8)

J-Type (Jump Operations)
Used for control flow. The target address is a 12-bit unsigned integer, allowing jumps within a 4K word address space.

15 12	**11 0**
Opcode (4)	Target Address (12)

5.2 Instruction Set Listing

The following table details the complete instruction set, including opcode assignments, ALU operation codes, and operational descriptions in Register Transfer Language (RTL).

Mnemonic	Opcode	Type	ALUOp	Operation / RTL Description
Arithmetic & Logical				
ADD	0000	R	000	$R_d \leftarrow R_s + R_t$
SUB	0001	R	001	$R_d \leftarrow R_s - R_t$
AND	0010	R	010	$R_d \leftarrow R_s \wedge R_t$
OR	0011	R	011	$R_d \leftarrow R_s \vee R_t$
XOR	0100	R	100	$R_d \leftarrow R_s \oplus R_t$
NOT	0101	R	101	$R_d \leftarrow \sim R_s$ (Rt=0000)
SHL	0110	R	110	$R_d \leftarrow R_s \ll 1$ (Rt=0000)
SHR	0111	R	111	$R_d \leftarrow R_s \gg 1$ (Rt=0000)
Data Transfer (Memory & Constants)				
MOV	1000	I	000	$R_d \leftarrow$ ZeroExtend(Imm8)
LD	1100	R	000	$R_d \leftarrow \text{Mem}[R_s]$ (Rt=0000)
ST	1101	R	000	$\text{Mem}[R_s] \leftarrow R_t$
Control Flow				
JMP	1001	J	—	$PC \leftarrow$ Address
JZ	1010	J	—	**if** $(Z == 1)$ $PC \leftarrow$ Address
JNZ	1011	J	—	**if** $(Z == 0)$ $PC \leftarrow$ Address
HLT	1111	-	—	Halt: Stop incrementing PC

Table 5: Instruction Set Definition with Binary ALUOp Encoding

5.3 ALU Operation Encoding

The 3-bit ALUOp control signal selects the ALU function:

ALUOp [2:0]	Operation	Description
000	ADD / PASS	Addition or pass-through (MOV, LD, ST)
001	SUB	Subtraction (two's complement)
010	AND	Bitwise AND
011	OR	Bitwise OR
100	XOR	Bitwise XOR
101	NOT	Bitwise NOT (invert all bits)
110	SHL	Shift left logical by 1 bit
111	SHR	Shift right logical by 1 bit

Table 6: ALU Operation Code Mapping

5.4 Encoding Constraints and Notes

- **Unused Bits:** For R-Type instructions that utilize only one source operand (e.g., `NOT`, `SHL`, `SHR`), the bits corresponding to the R_t field [3:0] are ignored by the control logic and should be set to 0 by the assembler.
- **Indirect Addressing:** The `LD` and `ST` instructions utilize Register Indirect addressing. The memory address is held in a General Purpose Register (R_s), allowing the CPU to access dynamic data structures in the SRAM.
- **Immediate Extension:** The `MOV` instruction loads an 8-bit constant (0-255). To load a full 16-bit value, a combination of `MOV` followed by shift and logical `OR` operations would be required. Example:

```
MOV r1, 0xAB     ; r1 = 0x00AB
SHL r1, r1       ; r1 = 0x0156 (shift 8 times)
...
MOV r2, 0xCD     ; r2 = 0x00CD
OR  r1, r1, r2   ; r1 = 0xABCD
```

- **Jump Range:** J-Type instructions directly access the full 4K ROM address space (0x000 to 0xFFF).
- **Reserved Opcodes:** Opcodes 1110 and 1110-1011 are currently undefined. Executing these results in undefined behavior (implementation-dependent, typically treated as NOP or triggers an error state).

6 Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) serves as the computational core of the processor. It is a purely combinational circuit responsible for executing all arithmetic, logical, and bit-manipulation instructions. The design is modular, consisting of three parallel execution units whose outputs are selected via a central multiplexing stage.

6.1 Arithmetic Operations

The arithmetic unit handles addition and subtraction operations using a 16-bit Ripple Carry architecture.

- **Full Adder Basis:** The core building block is a 1-bit Full Adder. Sixteen instances are chained together, with the Carry-Out (C_{out}) of stage i feeding the Carry-In (C_{in}) of stage $i + 1$.
- **Subtraction Logic:** Subtraction is implemented using Two's Complement arithmetic. When the subtraction control signal is active, the B operand is inverted (bitwise NOT), and the initial Carry-In (C_0) is set to 1, effectively calculating $A + (\sim B) + 1$.

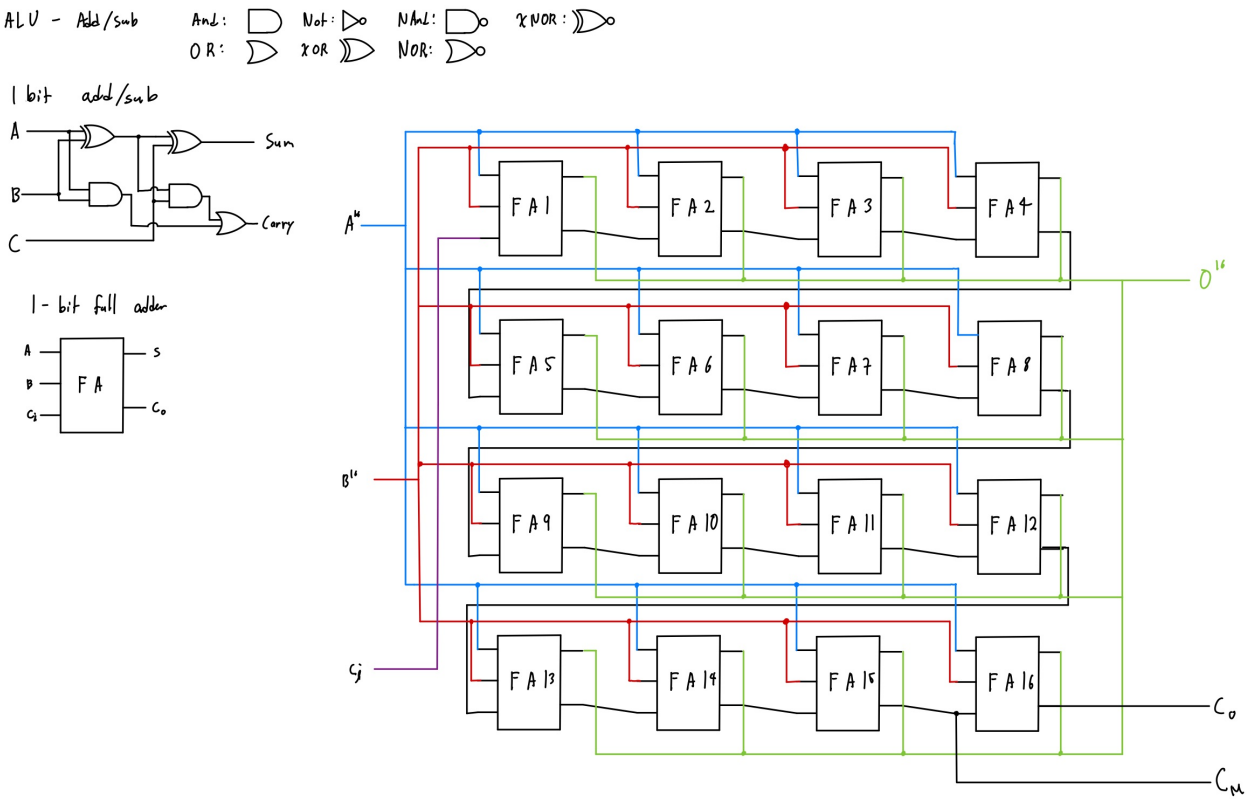

Figure 6: 1-bit Full Adder and 16-bit Ripple Carry Adder/Subtractor Configuration

6.2 Logical Operations

The Logic Unit performs bitwise operations in parallel. Unlike the arithmetic unit, there is no dependency between bits (no carry propagation), resulting in minimal propagation delay.

- **Operations Supported:** AND, OR, XOR, and NOT.
- **Implementation:** The unit consists of arrays of standard logic gates operating on the full 16-bit width of the input vectors.

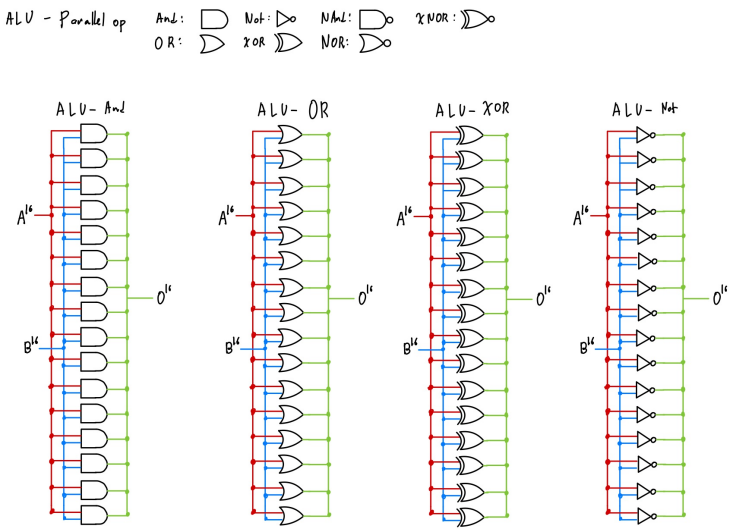

Figure 7: Parallel Bitwise Logic Architecture

6.3 Shifting Operations

Shift operations are critical for efficient multiplication and division by powers of two. The ALU implements fixed-distance logical shifts.

- **SHL (Shift Left):** Bits are shifted towards the MSB. The LSB is filled with 0, and the MSB is shifted out into the Carry flag.
- **SHR (Shift Right):** Bits are shifted towards the LSB. The MSB is filled with 0 (Logical Shift), and the LSB is shifted out.

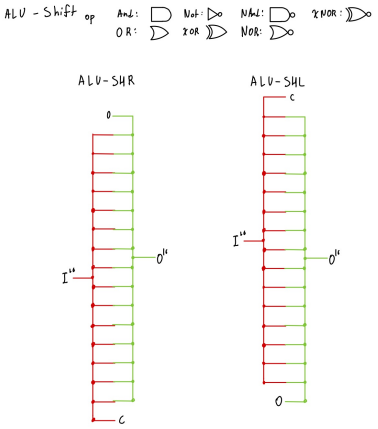

Figure 8: Hardwired Shift Left and Shift Right Wiring

6.4 Top-Level Integration and Flag Generation

The outputs from the Arithmetic, Logic, and Shift units are fed into a global 16-bit multiplexer, controlled by the instruction's ALUOp signal. In parallel with the result selection, the ALU generates four status flags which are latched into the Flags Register ($R10$).

- **Zero (Z):** Generated by a wide 16-input NOR gate on the final result. High if the result is $0x0000$.
- **Negative (N):** Directly connected to the Most Significant Bit (MSB, bit 15) of the result.
- **Carry (C):** The Carry-Out from the MSB of the adder (for arithmetic) or the bit shifted out (for shifts).
- **Overflow (O):** Indicates signed arithmetic overflow. See detailed implementation below.

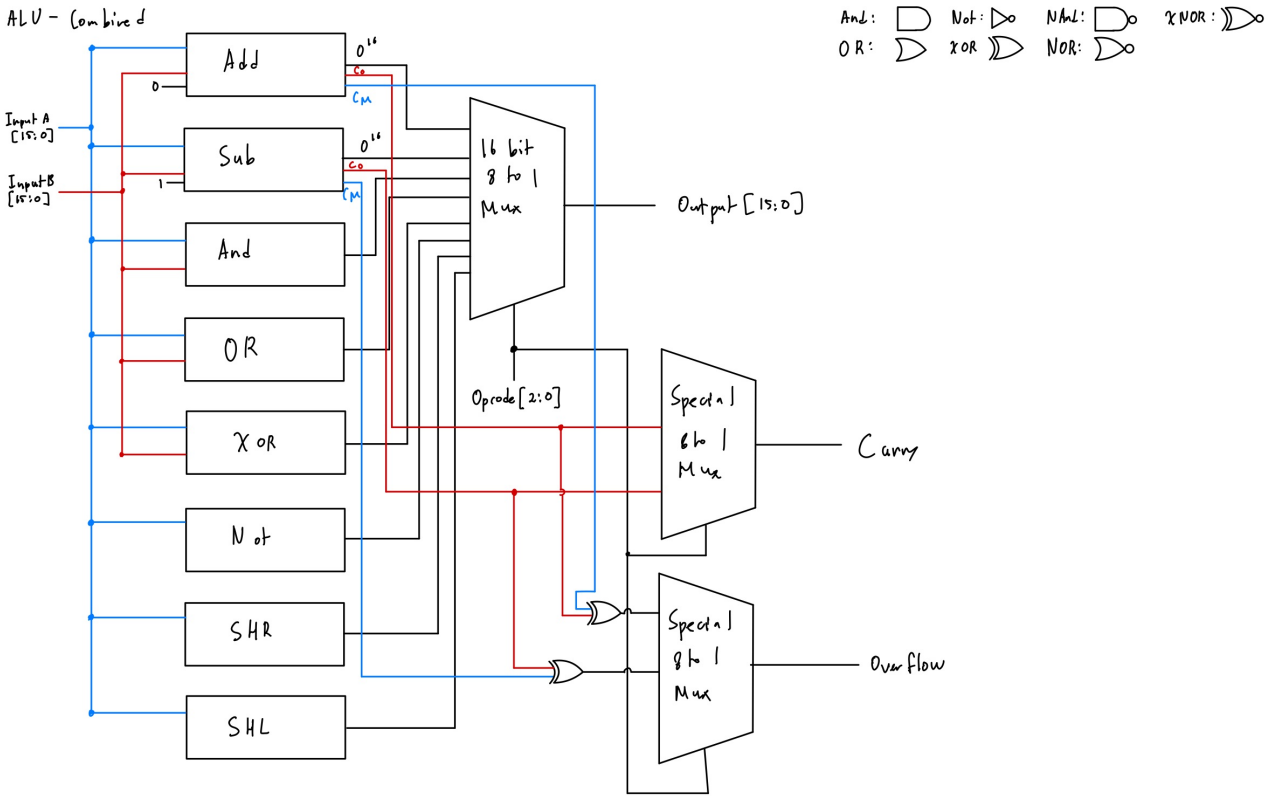

Figure 9: Top-Level ALU Datapath with Result Muxing and Flag Logic

6.5 Corrected Overflow Flag Implementation

The Overflow flag (O) indicates when a signed arithmetic operation produces a result that cannot be represented in 16-bit two's complement format (range: -32768 to +32767).

Incorrect Formula (from original specification): $O = C_{in,15} \oplus C_{out,15}$

This formula is insufficient as it doesn't properly detect all overflow conditions.

Correct Implementation:

For **Addition** ($Result = A + B$): $O = (A_{15} \wedge B_{15} \wedge \overline{Result_{15}}) \vee (\overline{A_{15}} \wedge \overline{B_{15}} \wedge Result_{15})$

For **Subtraction** ($Result = A - B$): $O = (A_{15} \wedge \overline{B_{15}} \wedge \overline{Result_{15}}) \vee (\overline{A_{15}} \wedge B_{15} \wedge Result_{15})$

Alternative unified formula (also correct): $O = (A_{15} \oplus Result_{15}) \wedge (B_{15} \oplus Result_{15})$
This works for addition. For subtraction, B should be inverted first (which happens in two's complement subtraction).

Physical Interpretation:

- **Positive + Positive = Negative:** Overflow (result too large)
- **Negative + Negative = Positive:** Overflow (result too small)
- **Positive + Negative:** Never overflows
- **Negative + Positive:** Never overflows

Truth Table for Overflow Detection (Addition):

A_{15}	B_{15}	$Result_{15}$	**Overflow**	**Condition**
0	0	0	0	Pos + Pos = Pos (OK)
0	0	1	1	Pos + Pos = Neg (OVERFLOW)
0	1	0	0	Pos + Neg = Pos (OK)
0	1	1	0	Pos + Neg = Neg (OK)
1	0	0	0	Neg + Pos = Pos (OK)
1	0	1	0	Neg + Pos = Neg (OK)
1	1	0	1	Neg + Neg = Pos (OVERFLOW)
1	1	1	0	Neg + Neg = Neg (OK)

Table 7: Overflow Detection Truth Table for Addition

7 Control Unit Specification

The Control Unit (CU) acts as the orchestration engine for the processor. It functions as a combinational logic block that decodes the 4-bit **Opcode** from the instruction register and asserts specific control signals to manage data flow, ALU operation, and register updates.

7.1 Control Signal Definitions

The following signals are generated for every clock cycle based on the current instruction state. All signals are Active High logic unless otherwise noted.

- **RegWrite:** Gating signal for the Register File. When asserted (1), data is latched into the register specified by R_d.
- **ALUSrc (ALU Source):** Controls the second operand multiplexer.
  - `0`: Second operand is Register R_t (R-Type).
  - `1`: Second operand is the zero-extended Immediate value (I-Type).
- **ALUOp [2:0]:** A 3-bit selector bus determining the specific arithmetic or logic function to be executed (see Section 5 for encoding).
- **MemRead / MemWrite:** Interface signals for the SRAM.
  - **MemRead:** Enables the SRAM output drivers (for `LD`).

  - **MemWrite:** Enables the SRAM write-enable latch (for `ST`).

- **MemToReg:** Controls the Write-Back multiplexer.
  - `0`: Data written to register comes from the **ALU Result**.
  - `1`: Data written to register comes from **Memory Output**.
- **FlagsWrite:** Enables the update of the Status Register (r10).
- **Branch:** Indicates that the Program Counter should be loaded with a target address rather than the default increment.

7.2 Control Logic Truth Table

The table below defines the state of control signals for each instruction. Note that 'X' denotes a "Don't Care" state.

Instr	Opcode	RegWr	ALUSrc	ALUOp	MemRd	MemWr	MemToReg	Branch
Arithmetic & Logic (R-Type)								
ADD	0000	1	0	000	0	0	0	0
SUB	0001	1	0	001	0	0	0	0
AND	0010	1	0	010	0	0	0	0
OR	0011	1	0	011	0	0	0	0
XOR	0100	1	0	100	0	0	0	0
NOT	0101	1	X	101	0	0	0	0
SHL	0110	1	X	110	0	0	0	0
SHR	0111	1	X	111	0	0	0	0
Memory & Data (I/R-Type)								
MOV	1000	1	1	000	0	0	0	0
LD	1100	1	0	000	1	0	1	0
ST	1101	0	0	000	0	1	X	0
Control Flow (J-Type)								
JMP	1001	0	X	X	0	0	X	1
JZ	1010	0	X	X	0	0	X	*Logic*
JNZ	1011	0	X	X	0	0	X	*Logic*
HLT	1111	0	X	X	0	0	X	0

Table 8: Control Signal Truth Table

7.3 Branching Logic Implementation

While unconditional jumps (`JMP`) assert the Branch signal directly, conditional jumps rely on combinatorial logic involving the Zero flag (Z) from the Status Register.

The **PC Source** logic is defined as: $PC_{Next} = \begin{cases} \text{Target Addr} & \text{if (Op == JMP)} \\ \text{Target Addr} & \text{if (Op == JZ} \land Z == 1) \\ \text{Target Addr} & \text{if (Op == JNZ} \land Z == 0) \\ PC_{Current} + 1 & \text{otherwise} \end{cases}$

7.4 Implementation Notes

- **Memory Addressing:** For `LD` and `ST` instructions, the ALU is configured in `PASS` mode (ALUOp = 000). It takes the address from the source register (R_s) and passes it directly to the SRAM Address Bus.
- **Write Back Path:** The `MemToReg` multiplexer is critical for the `LD` instruction. It ensures that the data written to the destination register (R_d) comes from the Memory Data Bus rather than the ALU Result.

8 Timing and Datapath

The CPU is a fully **synchronous** processor. Its operation is coordinated by a single, global clock signal (`CLK`) that drives all stateful elements (registers, PC, memory).

8.1 Clock Domains

The architecture utilizes a single clock domain. All logic, including the Register File, Control Unit, and SRAM, is synchronous to the rising edge of `CLK`.

- **Setup Time (T_{su}):** Data must be stable at a register's input for a specific duration *before* the rising clock edge to be captured correctly.
- **Hold Time (T_h):** Data must remain stable at the input for a specific duration *after* the rising clock edge.
- **Frequency:** The maximum clock frequency (F_{max}) is determined by the longest combinational delay path between two registers.

8.2 Instruction Cycle

The processor uses a **Single-Cycle** execution model. Every instruction, regardless of complexity, completes all of its stages within a single clock period (T_{clk}).

The five conceptual stages occur sequentially between two rising clock edges:

1. **Fetch:** The PC address is driven to the Instruction Memory (ROM), which outputs the 16-bit instruction word.
2. **Decode:** The instruction's Opcode [15:12] is decoded by the Control Unit to generate signals (RegWrite, ALUSrc, etc.), while register addresses are sent to the Register File.
3. **Execute:** The Register File outputs operands R_s and R_t. The ALU performs the operation specified by `ALUOp`, or calculates a memory address (for `LD`/`ST`).
4. **Memory:** For `LD` and `ST` instructions, the SRAM is accessed using the address calculated in the Execute stage. For all other instructions, this stage is idle.
5. **Write-Back:** The final result—selected from either the ALU output or the Memory output via the `MemToReg` multiplexer—is driven to the Register File's write port.

On the *next* rising edge, the PC updates to the next instruction address, and the destination register R_d latches the new result.

8.3 Signal Timing and Critical Path

The clock period (T_{clk}) must be longer than the worst-case propagation delay through the datapath. For this architecture, the critical path is typically the **Load (LD)** instruction, as it traverses the entire length of the processor:

$T_{clk} \geq T_{pc\_q} + T_{rom\_acc} + T_{reg\_read} + T_{alu} + T_{sram\_acc} + T_{mux} + T_{reg\_su}$

Where:

- T_{pc_q}: PC Clock-to-Q delay.
- T_{rom_acc}: Instruction Memory access time.
- T_{reg_read}: Register File read latency.
- T_{alu}: ALU propagation delay (Address Calculation).
- T_{sram_acc}: Data Memory access time.
- T_{mux}: Final Write-Back Mux delay.
- T_{reg_su}: Register File setup time.

8.4 Future Considerations

The single-cycle design is simple to implement but computationally inefficient, as the clock speed is bottlenecked by the slowest instruction (Load).

- **Pipelining:** The clear evolutionary path is a 5-stage pipeline (IF, ID, EX, MEM, WB). This would dramatically increase throughput by allowing multiple instructions to be in flight simultaneously.
- **Wait States:** To support slower, larger memories without slowing down the core clock, a "Ready/Valid" handshake protocol could be implemented to stall the processor during memory access.

9 System Reset and Initialization

9.1 Reset Behavior

Upon system reset (active-low `RESET_N` signal), the following initialization occurs:

Component	Initial Value	Notes
PC (r9)	0x000	Fetch first instruction from ROM address 0
SP (r8)	0xFFF	Stack grows downward from top of SRAM
FLAGS (r10)	0x0000	All flags cleared (Z=N=C=O=0)
GPRs (r1-r7)	Undefined	Should be initialized by program
r0	0x0000	Always zero (hardwired)

Table 9: Register File State After Reset

Reset Sequence:

1. Assert `RESET_N` low (active)
2. Hold for minimum 2 clock cycles
3. De-assert `RESET_N` high
4. CPU begins fetching from address 0x000 on next rising clock edge

9.2 HLT Instruction Behavior

When the `HLT` instruction (opcode 1111) is executed:

- The PC stops incrementing (remains at current address)
- The CPU re-fetches the HLT instruction indefinitely
- No register or memory state changes occur
- To resume execution, the system must be hard reset

Alternative implementations: Some designs may halt the clock entirely or assert a HALT output signal to external logic for power management.

10 Reference Assembly Sequences

This section provides standard assembly language listings designed to validate the architectural integrity of the CPU. These sequences demonstrate the interaction between the ALU, Register File, Memory, and Control Unit.

10.1 Sequence A: Arithmetic & Flag Verification

This routine performs basic register initialization and arithmetic operations. It validates the ALU's ability to perform two's complement addition and subtraction, as well as the propagation of the **Zero (Z)** and **Negative (N)** flags.

Listing 1: Basic Arithmetic Flow

```
1         ; -- Initialization --
2         MOV r1, 15            ; Load Immediate: r1 = 15 (0x0F)
3         MOV r2, 10            ; Load Immediate: r2 = 10 (0x0A)
4
5         ; -- Addition Test --
6         ADD r3, r1, r2        ; r3 = 15 + 10 = 25 (0x19)
7         ; Flags: Z=0, N=0, C=0, O=0
8
9         ; -- Subtraction & Zero Flag Test --
10        MOV r4, 25            ; Load Expected Result
11        SUB r5, r3, r4        ; r5 = 25 - 25 = 0
12        ; Flags: Z=1 (Result is Zero)
```

10.2 Sequence B: Logic & Bit Manipulation (Corrected)

This sequence demonstrates bitwise operations using the dedicated `NOT` instruction.

Listing 2: Bitwise Logic Operations - Corrected

```
; -- Setup --
MOV r1, 170          ; r1 = 1010 1010 (0xAA)
MOV r2, 255          ; r2 = 1111 1111 (0xFF)

; -- Direct Inversion using NOT instruction --
NOT r3, r1           ; r3 = ~r1 = 0101 0101 (0x55)
; Binary encoding: 0101 0011 0001 0000
;   Opcode=0101 (NOT), Rd=0011 (r3)
;   Rs=0001 (r1), Rt=0000 (ignored)

; -- Alternative: XOR-based inversion (for comparison) --
XOR r4, r1, r2       ; r4 = r1 ^ 0xFF = 0101 0101 (0x55)
; Same result, but requires two registers

; -- Masking (AND) --
MOV r5, 15           ; r5 = 0000 1111 (0x0F) - Lower nibble
   mask
AND r6, r1, r5       ; r6 = 0xAA & 0x0F = 0x0A (0000 1010)
```

10.3 Sequence C: Memory Access (SRAM)

This sequence validates the load/store architecture. It stores a value into the Data Memory (SRAM) and then retrieves it into a different register to verify the path.

Listing 3: SRAM Read/Write Test

```
; -- Setup Pointers and Data --
MOV r1, 100          ; r1 = Memory Address (Pointer)
MOV r2, 42           ; r2 = Data Value to Store

; -- Store to Memory --
ST r1, r2            ; MEM[100] = 42
; Binary encoding: 1101 0001 0001 0010
;   Opcode=1101 (ST), Rd=0001 (contains Rs)
;   Rs=0001 (r1=address), Rt=0010 (r2=data)

; -- Clear Register --
MOV r3, 0            ; r3 = 0 (Verify we aren't cheating)

; -- Load from Memory --
LD r3, r1            ; r3 = MEM[100]
; Binary encoding: 1100 0011 0001 0000
;   Opcode=1100 (LD), Rd=0011 (r3)
;   Rs=0001 (r1=address), Rt=0000 (unused)
; Final State: r3 should equal 42 (0x2A)
```

10.4 Sequence D: Control Flow (Iterative Loop)

This routine implements a "Countdown Loop," summing numbers from 5 down to 1. This tests the **Branching Logic** and the `JNZ` instruction.

Listing 4: Summation Loop with Addresses

```
; Address   | Instruction              | Binary Encoding
;
   ---------|--------------------------|--------------------------------

; 0x000     | MOV r1, 0                | 1000 0001 0000 0000
; 0x001     | MOV r2, 5                | 1000 0010 0000 0101
; 0x002     | MOV r3, 1                | 1000 0011 0000 0001

; 0x003     | loop_start:              |
; 0x003     | ADD r1, r1, r2           | 0000 0001 0001 0010
; 0x004     | SUB r2, r2, r3           | 0001 0010 0010 0011
; This SUB updates the Zero Flag (Z)

; 0x005     | JNZ loop_start           | 1011 0000 0000 0011
; If Z == 0, Jump to address 0x003
; Opcode=1011 (JNZ), Target=0x003

; 0x006     | HLT                      | 1111 0000 0000 0000
; Final State: r1 = 15 (5+4+3+2+1), r2 = 0, FLAGS[Z]=1
```

Binary Encoding Breakdown:

- `JNZ 0x003` encodes as: `1011 0000 0000 0011`
  - Bits [15:12]: `1011` (JNZ opcode)
  - Bits [11:0]: `000000000011` (target address = 3)
- `ADD r1, r1, r2` encodes as: `0000 0001 0001 0010`
  - Bits [15:12]: `0000` (ADD opcode)
  - Bits [11:8]: `0001` (Rd = r1)
  - Bits [7:4]: `0001` (Rs = r1)
  - Bits [3:0]: `0010` (Rt = r2)

10.5 Implementation Constraints

When writing assembly for this architecture, the following hardware constraints must be observed:

- **Immediate Width:** The `MOV` instruction supports only 8-bit unsigned integers ($0-255$). Loading larger values requires a multi-instruction sequence or loading from Data Memory using `LD`.
- **Delay Slots:** This architecture assumes no branch delay slots. The instruction immediately following a branch is only executed if the branch is *not* taken.

- **Stack Operations:** No dedicated PUSH/POP instructions exist. Stack must be managed manually:

```
; Manual PUSH r1 (push value in r1 onto stack)
ST r8, r1            ; Store r1 at address in SP
MOV r7, 1
SUB r8, r8, r7       ; Decrement SP (stack grows
   down)

; Manual POP r1 (pop value from stack into r1)
MOV r7, 1
ADD r8, r8, r7       ; Increment SP
LD r1, r8            ; Load from address in SP
```