CHITTAGONG UNIVERSITY OF ENGINEERING & TECHNOLOGY

Department of Electronics and Telecommunication Engineering

Project Report

Title of the Project

Design and Implementation of Modified SAP-1 Architecture-based Microprocessor using Logisim

Course Code	:	ETE 404
Course Title	:	VLSI Technology Sessional
Date of Submission	:	6th October, 2025

Submitted by-	**Submitted to-**
Name : Farhan Tanvir ID : 2008060 Level : 4 Term : I	Arif Istiaque Rupom Lecturer, Department of Electronics and Telecommunication Engineering

Remarks

1. Objectives

- To design and implement a modified SAP-1 architecture-based microprocessor using Logisim
- To design an Instruction loader for automated instruction loading into the RAM from external memory device.
- To design a Controller Sequencer for full automated control of Fetch-Decode-Execution Cycles.

Required Software: Logisim Evolution v3.9.0

2. Introduction

The SAP-1 (Simple-As-Possible-1) is a very basic educational microprocessor architecture that demonstrates fundamental computer components and operation using an 8-bit design with limited memory (16 bytes) and a basic instruction set (i.e. LDA, ADD, SUB, OUT, HLT).

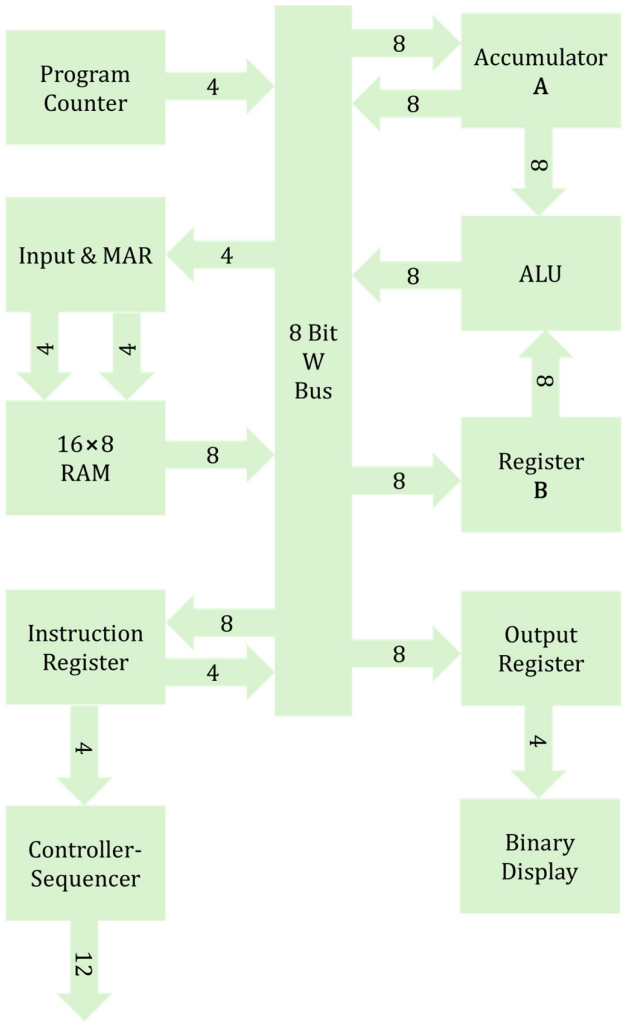

Figure 2.1: Simple as Possible architecture-1 (SAP-1)

Key Components

SAP-1 uses a Von-Neumann architecture with a single 8-bit central bus (W-bus) for data transfer between components. Its main parts include:

- **Program Counter (PC)**: A 4-bit counter that holds the address of the next instruction, incrementing automatically to sequence program execution.
- **Memory Address Register (MAR)**: Stores the 4-bit address from the PC to access RAM.
- **RAM**: 16 × 8-bit memory (16 locations, each holding 8 bits for instructions or data).
- **Instruction Register (IR)**: Holds the fetched 8-bit instruction from RAM.
- **Accumulator (A Register)**: The primary 8-bit register for arithmetic operations.
- **B Register**: A supporting 8-bit register for holding operands during calculations.
- **Adder/Subtractor**: Performs basic addition and subtraction on values from A and B.
- **Output Register**: Displays results via LEDs or similar.
- **Controller/Sequencer**: Manages timing and control signals across fetch and execute cycles.

It uses a simple fetch-execute cycle with a limited instruction set to perform basic operations like loading data, addition, and subtraction, allowing students to understand how a microprocessor interacts with memory and I/O. This project aims to Modify the basic SAP-1 Microprocessor by extending its capabilities for a more in-depth understanding. The design introduces a comprehensive instruction set, a bootloader for program loading via hex files, and a sophisticated control unit, all interconnected via a shared 8-bit tri-state controlled bus.

This microprocessor is ideal for simulating basic computing tasks, such as arithmetic operations and program control, within a constrained yet extensible framework. The design specification and components of our Modified SAP-1 architecture is described below:

Design Specifications

Key features include:

- 8-bit data bus and general-purpose registers, supporting a wider range of operations.
- 4-bit address space with 16 locations of 8-bit SRAM.
- 22-instruction set covering arithmetic, logical, bitwise, shift/rotate, branching, and memory operations.
- Bootloader utilizing Logisim's ROM for preloading SRAM.

- Advanced ALU with flag updates and a 5-stage ring counter for timing control.

Key Components:

- ✓ Arithmetic and Logic Unit
- ✓ General Purpose Register
- ✓ Program Counter
- ✓ SRAM
- ✓ MAR (Memory Address Register)
- ✓ IR (Instruction Register)
- ✓ Central Bus
- ✓ Controller Sequencer
- ✓ Bootloader

The design process of this project solely depends on the instructions set. So, designing the instruction set, and proper utilization of the 8-bit instruction word consisting of opcode and operand is very important. Hence, the instructions can be grouped as:

1. Arithmetic and Logical Operations

- Arithmetic: ADD, SUB, INC, DEC, CMP
  These arithmetic operations don't require any operand; the opcode itself is sufficient to dictate their execution. Because, ADD, SUB and CMP these are always supposed to operate between register A and B. The INC and DEC operator only operate on the Accumulator.

- Logical Operations: Bitwise OR, AND, XOR, NOT, REV, SHL, SHR, ROL, ROR
  These logical operations also don't require any operand where OR, AND, XOR operate on A and B, whereas REV, SHL, SHR, ROL and ROR operate on register A only. But, SHR, SHL, ROL, and ROR these operations need operand for specifying the amount of shifting.

2. Control Flow Operations

- Jump Operations: JMP, JZ, JNZ
  These operations require an operand for specifying the destination of the jump operation.

3. Transfer Operations

- LDA, LDB, STA
  These operations have source or destination, any of them is fixed. Hence, the other location (either source or destination) needs to be specified as the operand.

4. Control Operations

- HLT, NOP
  These operations don't require any operand, they are self-sufficient.

Hence, analyzing the despondency on the operand it can be said that, Shifting & Rotation(4), Jumps(3) and Transfer operations(3) in total 10 operations require the operand and rest of the 12 operations don't require any operand. As the 4 bits-opcode can forms 16 combinations, out of them 10 are fixed 4-bit opcodes, the remaining 6 combinations can be be used to extend the number of operations by borrowing only 1 bit from the operand as they don't require the operand as their argument. The table below demonstrates the 22 combinations of opcode where b4 is borrowed from the operand.

No.	OPCODE	b4	b3	b2	b1	b0	Format	Description
0.	LDA	0	0	0	0	0	LDA addr	Loads memory content of addr to A
1.	LDB	0	0	0	0	1	LDB addr	Loads memory content of addr to B
2.	STA	0	0	0	1	0	STA addr	Stores memory content of A to addr
3.	JMP	0	0	0	1	1	JMP addr	Jumps to instr. addr
4.	JZ	0	0	1	0	0	JZ addr	Jumps to instr. Addr if zero flag is on
5.	JNZ	0	0	1	0	1	JNZ addr	Jumps to instr. Addr if zero flag is off
6.	SHL	0	0	1	1	0	SHL amount	Left shift n bit. n = amount
7.	SHR	0	0	1	1	1	SHR amount	Right shift n bit. n = amount
8.	ROL	0	1	0	0	0	ROL amount	Left rotate n bit. n = amount
9.	ROR	0	1	0	0	1	ROR amount	Right rotate n bit. n = amount
10.	ADD	0	1	0	1	0	ADD	Adds A and B and stores the result to A
11.	SUB	0	1	0	1	1	SUB	Subs A and B and stores the result to A
12.	INC	0	1	1	0	0	INC	Increments A and stores the result to A
13.	DEC	0	1	1	0	1	DEC	Decrements A and stores the result to A
14.	REV	0	1	1	1	0	REV	Reverse the bits of A
15.	HLT	0	1	1	1	1	HLT	Halts the CS
16.	NOP	1	1	0	1	0	NOP	Does nothing for 5 clock pulse.
17.	CMP	1	1	0	1	1	CMP	Subs A and B and updates the flags only.
18.	BW_OR	1	1	1	0	0	BW_OR	Bitwise OR between A and B, Store to A
19.	BW_AND	1	1	1	0	1	BW_AND	Bitwise AND between A and B, Store to A
20.	BW_NOT	1	1	1	1	0	BW_NOT	Bitwise NOT of A, Stores to A
21.	BW_XOR	1	1	1	1	1	BW_XOR	Bitwise XOR between A and B, Store to A

3. Design Process

3.1 Central bus

The Central Bus is the primary data pathway in the Modified SAP-1 Microprocessor, facilitating communication between major components such as the ALU, registers, memory, and I/O. It is an 8-bit bidirectional bus that uses tri-state buffers to manage data transfer, preventing contention by enabling only one component to drive it at a time.

Design and Structure

- **Bit Width**: 8 bits, supporting 8-bit data words.
- **Core Elements**:
  - Comprises eight tri-state buffer lines, controlled by enable signals from the controller.
  - Connects the Accumulator, Register B, Instruction Register, Memory Address Register, and SRAM.
- **Control**: Managed by the Controller-Sequencer, which asserts specific enable lines to direct data flow.

3.2 General Purpose Register (`gp_reg`)

The General-Purpose Register (**`gp_reg`**) is an 8-bit register designed as a fundamental building block for the Modified SAP-1 Microprocessor. This register serves as the basis for various components such as the accumulator (A), secondary register (B), the Instruction Register, the Memory Address Register (MAR) and the SRAM cells within the SRAM subsystem. It is a synchronous, edge-triggered register with input, output, and control mechanisms to interface with the 8-bit central bus.

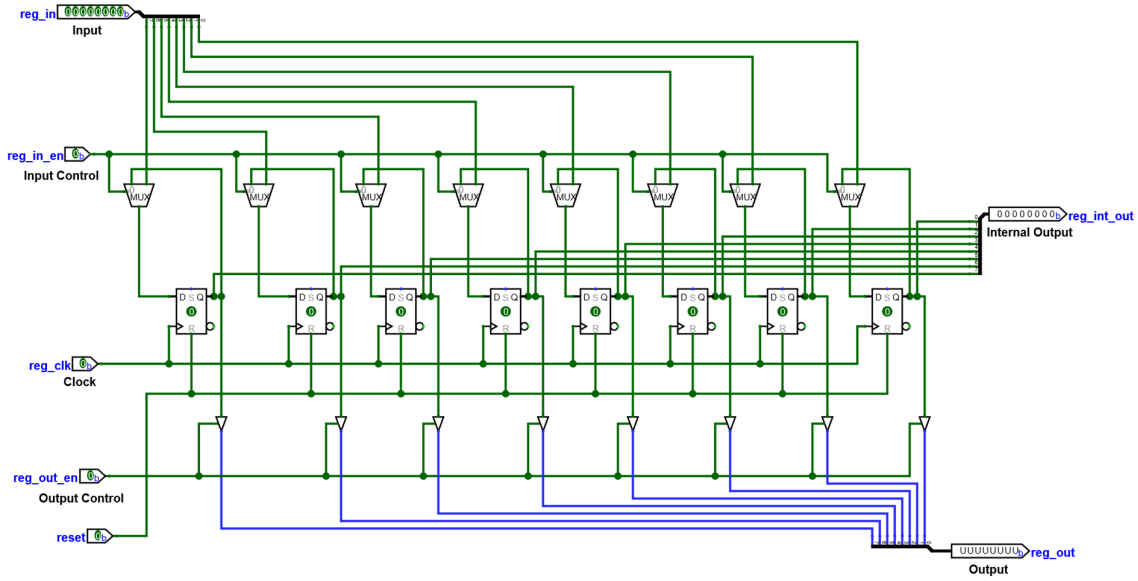

Figure 3.2.1: General Purpose Register (**`gp_reg`**)

Design and Structure

- **Bit Width**: 8 bits, allowing storage and manipulation of 8-bit data words.

- **Core Elements**: The register consists of eight D-type flip-flops, each storing one bit of the 8-bit word. These flip-flops are clocked synchronously to ensure consistent data updates.
- **Input Multiplexing**: Eight 2-to-1 multiplexers (MUX) are used at the input of each flip-flop. The MUX allows selection between the input data (**`reg_in`**) and the current internal state, controlled by the **`reg_in_en`** (input enable) signal. This enables dynamic loading of new data or retention of the existing value.
- **Output Buffering**: Eight tri-state buffers are employed at the output, controlled by the **`reg_out_en`** (output enable) signal. This allows the register to drive the 8-bit bus only when enabled, preventing bus contention in the shared bus architecture.
- **Clocking**: A single **`reg_clk`** (clock) signal drives all flip-flops, ensuring synchronized operation with the microprocessor's clock cycle.
- **Reset**: A reset signal is connected to each flip-flop, clearing the register to zero when asserted.

Pins and Signal Descriptions

- **Inputs**
  - **`reg_in`**: 8-bit input data from the central bus.
  - **`reg_clk`**: Clock signal triggering data updates on the rising edge.
  - **`reset`**: Asynchronous reset signal (active high) to clear the register.
- **Outputs**
  - **`reg_int_out`**: 8-bit internal output, reflecting the current state of the register before tri-state buffering.
  - **`reg_out`**: 8-bit output to the bus when **`reg_out_en`** is active.
- **Control**
  - **`reg_in_en`**: Input enable signal (active high) to load new data into the register.
  - **`reg_out_en`**: Output enable signal (active high) to connect the register to the bus.

Operation

1. **Data Loading**: When **`reg_in_en`** is high, the MUX selects the **`reg_in`** data, which is loaded into the flip-flops on the next rising edge of **`reg_clk`**. If **`reg_in_en`** is low, the current state is retained.
2. **Data Output**: When **`reg_out_en`** is high, the tri-state buffers drive the **`reg_int_out`** value onto the **`reg_out`** line, making it available on the bus. When low, the output is high-impedance, isolating the register from the bus.

3. **Reset**: Asserting **`reset`** asynchronously clears all flip-flops to zero, overriding other inputs.

3.3 Program Counter (pc)

The Program Counter (PC) is a 4-bit synchronous binary up-counter with parallel load capability. It is a critical component of the microprocessor, responsible for storing and updating the memory address of the next instruction to be fetched. The PC supports both incremental counting for sequential execution and parallel loading for jump instructions, ensuring flexible control flow.

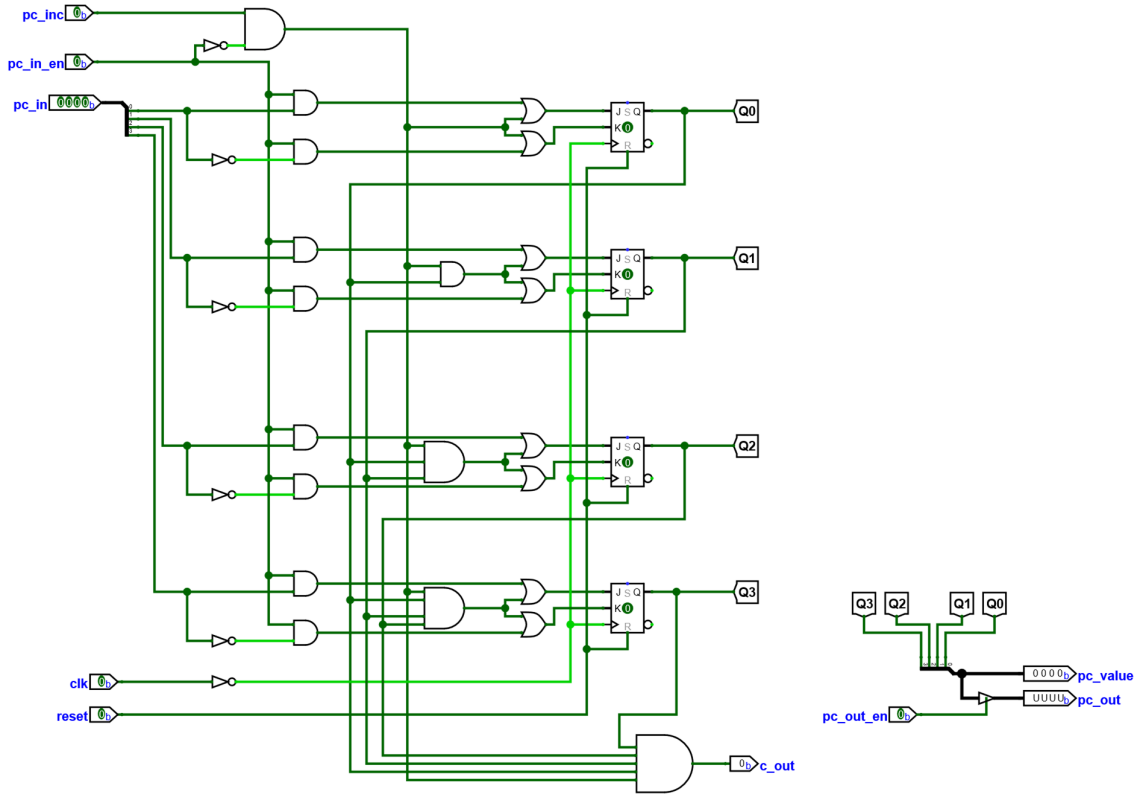

Figure 3.3.1: Program Counter (**pc**)

Design and Structure

- **Bit Width**: 4 bits, allowing the PC to address up to 16 memory locations (0x0 to 0xF).
- **Core Elements**: The PC is constructed using four J-K flip-flops, configured as a synchronous counter. Each flip-flop represents one bit (Q0 to Q3), with the outputs forming the 4-bit address.
- **Increment Logic**: The counter increments by one on each clock cycle when enabled. This is achieved through combinational logic (AND and OR gates) that generates the

J and K inputs for each flip-flop, creating a ripple-carry effect for the up-counting behavior.

- **Parallel Load**: The PC includes a parallel load mechanism using additional gating. When **pc_in_en** is active, the 4-bit input **pc_in** is loaded into the flip-flops, overriding the increment logic.
- **Output Buffering**: The 4-bit output (**pc_out**) is driven through a tri-state buffer controlled by **pc_out_en**, allowing the PC to connect to the 8-bit bus with the upper 4 bits unused or tied off.
- **Clocking**: A single **clk** signal synchronizes all flip-flops, aligning the PC with the microprocessor's clock cycle.
- **Reset**: A **reset** signal clears all flip-flops to zero asynchronously when asserted.

Signal Descriptions

- **Inputs**:
  - **pc_in**: 4-bit parallel input data for loading a new address.
  - **pc_in_en**: Input enable signal (active high) to load **pc_in** into the PC.
  - **clk**: Clock signal triggering updates on the rising edge.
  - **reset**: Asynchronous reset signal (active high) to set the PC to 0x0.
- **Outputs**:
  - **pc_value**: 4-bit internal value representing the current address.
  - **pc_out**: 4-bit output to the bus when **pc_out_en** is active.
- **Control**:
  - **pc_out_en**: Output enable signal (active high) to drive the bus.
  - **pc_inc**: Increment enable signal (active high) to advance the counter.

Operation

1. **Increment Mode**: When **pc_inc** is high and **pc_in_en** is low, the PC increments by one on the rising edge of **clk**, moving to the next memory address for sequential instruction fetch.
2. **Parallel Load Mode**: When **pc_in_en** is high, the **pc_in** value is loaded into the PC on the next **clk** edge, used for jump instructions (e.g., **JMP**, **JZ**, **JNZ**).
3. **Output Control**: The **pc_out_en** signal enables the tri-state buffer, allowing the PC to output its value to the bus for memory addressing.
4. **Reset**: Asserting **reset** sets all flip-flops to zero, initializing the PC to 0x0.

Here, the clock signal **clk** is connected to all four flip-flops simultaneously. This ensures that all flip-flops change state at the same time, eliminating propagation delays. Hence, it provides a reliable output.

3.4 Arithmetic and Logic Unit(`alu`)

The Arithmetic and Logic Unit (ALU) is the central processing component of the Modified SAP-1 Microprocessor, responsible for executing arithmetic, logical, and bit-manipulation operations. It integrates multiple sub-circuits to support a wide range of instructions, including ADD, SUB, INC, DEC, CMP, SHL, SHR, ROL, ROR, REVERSE, and bitwise operations (OR, AND, NOT, XOR).

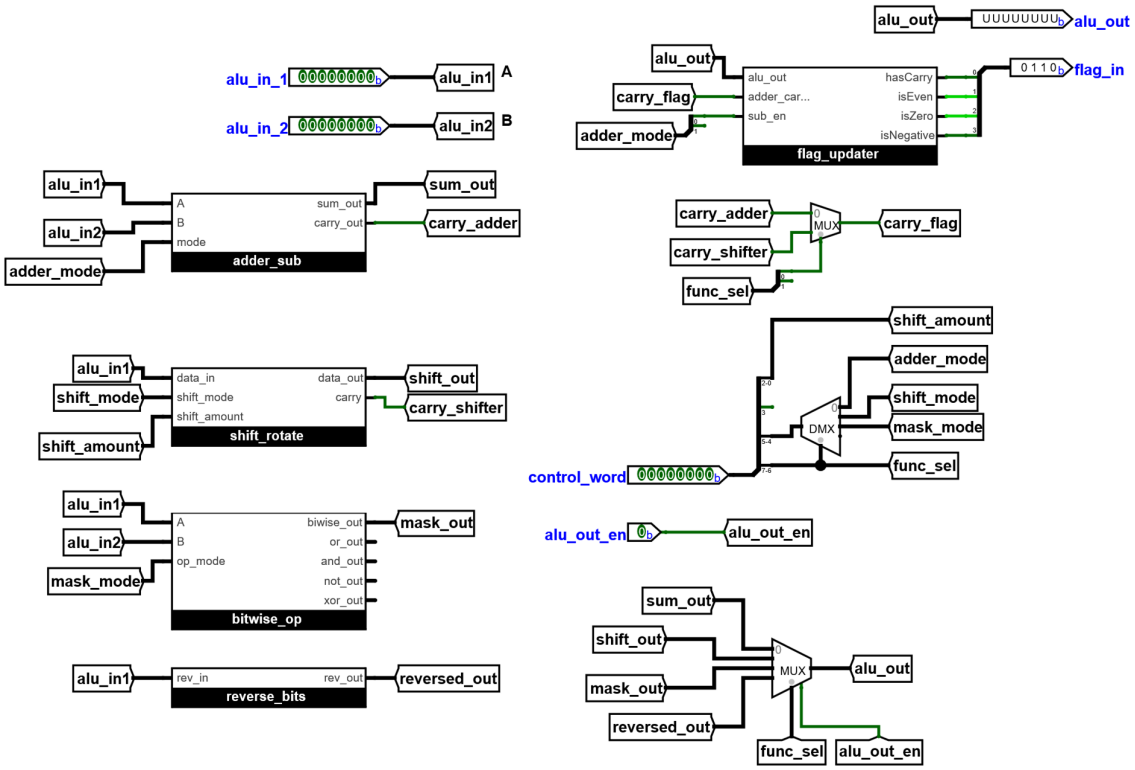

Figure 3.4.1: High level schematic of ALU.

Design and Structure

- **Bit Width**: 8 bits, processing 8-bit inputs to produce an 8-bit result and status flags.
- **Core Elements**:
  - **Adder/Subtractor**: Handles ADD, SUB, INC, DEC and CMP using an 8-bit full adder with mode control (0: ADD, 1: SUB, 2: INC, 3: DEC), the CMP leverages the SUB mode for only updating the flag bits.
  - **Shifting Circuit**: Supports SHL and SHR with a direction signal (0: left, 1: right).

    - **Rotation Circuit**: Performs ROL and ROR with a direction signal (0: left, 1: right).
    - **Reverse Bits**: Inverts bit order for the REVERSE instruction.
    - **Bitwise Operations**: Executes OR, AND, NOT, and XOR using parallel logic gates.
    - **Flag Updater**: Computes Carry (C), Zero (Z), Sign (S), and Parity (P) flags based on the results.
- **Control Logic**:
    - A 4-bit op_code (operation code) selects the operation (e.g., 0000: ADD, 0001: SUB, etc.).
    - Additional control signals (e.g., shift_amount, direction) configure shifter and rotator behavior.
- **Data Path**: Inputs from the Accumulator (A) and Register B (or constants) are routed through multiplexers to the selected sub-circuit.

Signal Descriptions

- **Inputs**:
    - a_in: 8-bit input from the Accumulator.
    - b_in: 8-bit input from Register B (used in ADD, SUB, bitwise ops).
    - op_code 4-bit operation selector.
    - shift_amount 3-bit shift/rotation distance (0 to 7).
    - direction: 1-bit signal (0: left, 1: right for shifts/rotations).
- **Outputs**:
    - alu_out: 8-bit result of the operation.
    - flag_out[3:0]: 4-bit flags [C, Z, S, P] for the Flag Register.

Operation

- **Arithmetic**: For ADD, SUB, INC, or DEC, the adder/subtractor processes a_in and b_in (or a constant) based on op_code, updating flags via the Flag Updater.
- **Shifts**: SHL or SHR uses the Barrel Shifter, shifting a_in by shift_amount with direction control.
- **Rotations**: ROL or ROR uses the Rotation Circuit, rotating a_in by shift_amount with direction control.
- **Bitwise**: OR, AND, XOR, or NOT applies the selected operation to a_in and b_in (NOT uses only a_in).

- **Reverse**: Reverses a_in bit order using the Reverse Bits circuit.
- **Data Flow**: The 8-bit alu_out and 4-bit flag_out are output, with alu_out routed to the bus and flag_out to the Flag Register.

Each subcircuit is detailed below:

A. Adder/ Subtractor Circuit(`adder_sub`)

The Adder/ Subtractor Circuit (adder_sub) is a fundamental component of the ALU. This circuit performs 8-bit addition and subtraction operations, including increment (**INC**) and decrement (**DEC**) modes, supporting the **ADD**, **SUB**, **INC**, and **DEC** instructions. It is designed to handle binary arithmetic with carry propagation and includes a 2-bit mode control to switch between the four operations.

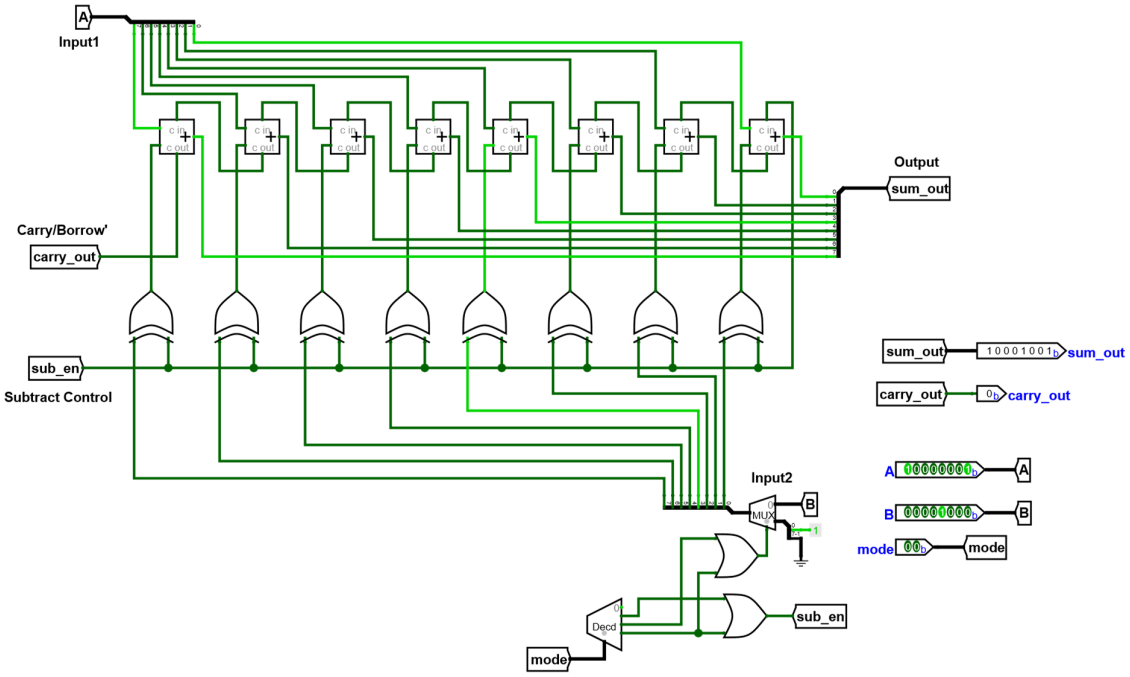

Figure 3.4.2: Adder/ Subtractor Circuit

Design and Structure

- **Bit Width**: 8 bits, processing one or two 8-bit inputs (a_in and optionally b_in) to produce an 8-bit result (sum_out) and a carry-out (carry_out).
- **Core Elements**:
  - Comprises eight full adder stages, each processing one bit with inputs from a_in, a modified b_in or constant, and the previous carry.
  - Uses XOR gates and multiplexers to adjust b_in or apply constants based on mode.

- **Mode Control**:
  - A 2-bit mode signal (mode[1:0]) selects the operation:
    - 00: ADD (a_in + b_in)
    - 01: SUB (a_in - b_in)
    - 10: INC (a_in + 1)
    - 11: DEC (a_in - 1)
  - Carry-in (c_in) is configured as 0 for ADD/INC and 1 for SUB/DEC.
- **Carry Propagation**: Carry bits ripple through the adder chain, with the final carry-out indicating overflow or borrow. When it's SUB or DEC operation the carry_out bit is the inverse of the actual logical borrow bit.

Signal Descriptions

- **Inputs**:
  - a_in: 8-bit input (e.g., from the Accumulator).
  - b_in: 8-bit input (used for ADD and SUB; ignored or set to 00000001 for INC/DEC).
  - mode: 2-bit control (00: ADD, 01: SUB, 10: INC, 11: DEC).
  - c_in: 1-bit carry-in (0 for ADD/INC, 1 for SUB/DEC).
- **Outputs**:
  - sum_out: 8-bit result of the operation.
  - carry_out: 1-bit output indicating the final carry or borrow.

Operation

- **Addition Mode (mode = 00)**: Adds a_in and b_in, with c_in = 0, producing sum_out and carry_out.
- **Subtraction Mode (mode = 01)**: Inverts b_in with XOR and sets c_in = 1 for two's complement subtraction, yielding sum_out with borrow in carry_out.
- **Increment Mode (mode = 10)**: Adds a_in and 1 (b_in ignored or set to 00000001), with c_in = 0, updating sum_out and carry_out.
- **Decrement Mode (mode = 11)**: Subtracts 1 from a_in (b_in inverted to 11111111), with c_in = 1, producing sum_out with borrow in carry_out.
- **Data Flow**: The 8-bit sum_out and carry_out are output for further processing or flag updates.

B. Reverse Bits:

The Reverse Bits circuit is a specialized component of the ALU. It reverses the order of bits in an 8-bit input word, supporting the **REV** instruction. It is designed to manipulate data at the bit level, providing a useful operation for certain algorithms or debugging purposes. It is pure combinational and static circuit which doesn't require additional controlling signals or clock.

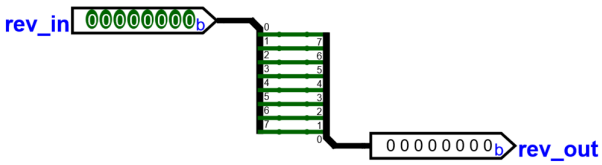

Figure 3.4.3: Bit Reversal sub-circuit.

Design and Structure

- Bit Width: 8 bits, processing an 8-bit input (rev_in) to produce an 8-bit output (rev_out) with bits reversed.
- Core Elements: The circuit uses two bit-splitters to reorder the bits. Each bit position (0 to 7) is connected to its complementary position (7 to 0), effectively flipping the byte.

Signal Descriptions

- **Inputs:**
  rev_in: 8-bit input data to be reversed (e.g., from the Accumulator).

- **Outputs:**
  rev_out: 8-bit output with bits reversed (e.g., bit 0 of rev_in becomes bit 7 of rev_out, and vice versa).

C. Bitwise Operations

The Bitwise Operations circuit is a key component of the Arithmetic Logic Unit (ALU). This circuit performs four fundamental bitwise operations—OR, AND, NOT, and XOR—supporting the **BW_OR, BW_AND, BW_NOT,** and **BW_XOR** instructions. It enables bit-level manipulation of 8-bit data, enhancing the microprocessor's capability for logical and masking operations. The design is combinational, requiring no clocking, with outputs updated instantly based on inputs and mode.

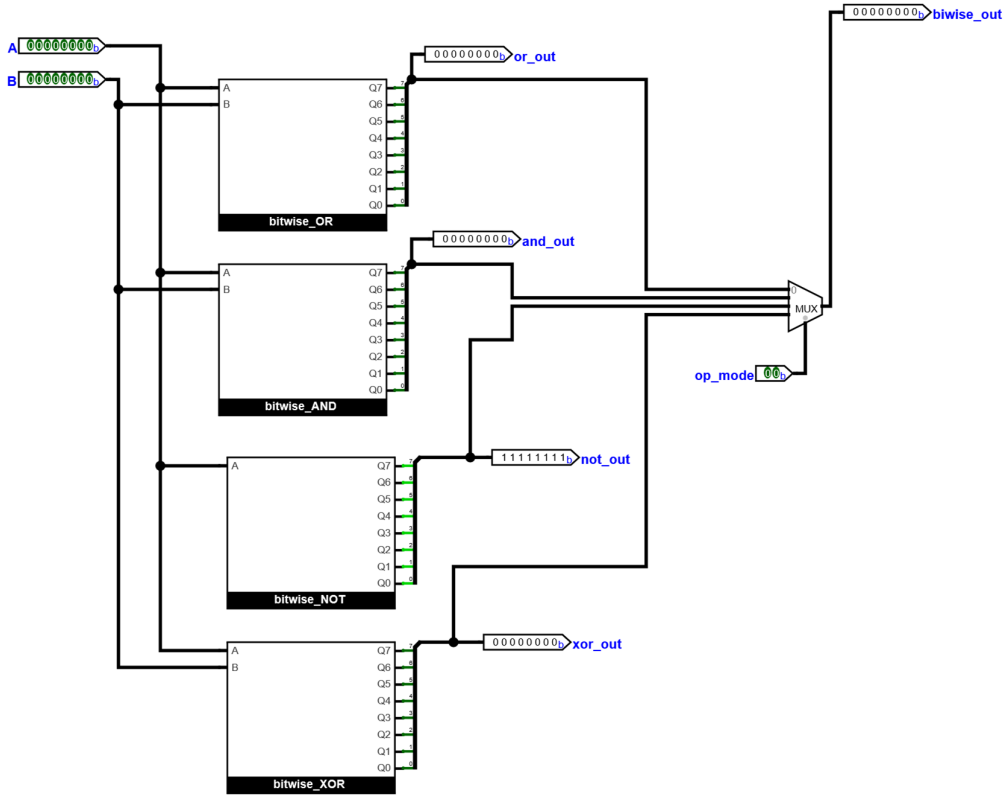

Figure 3.4.4: High Level Circuit of Bitwise Operations.

Design and Structure

- **Bit Width**: 8 bits, processing one or two 8-bit inputs (a_in and optionally b_in) to produce an 8-bit output.
- **Core Elements**:
  - Uses eight parallel logic gates per operation:
    - OR gates for BW_OR.
    - AND gates for BW_AND.
    - NOT gates for BW_NOT.
    - XOR gates for BW_XOR.
  - Each gate operates on corresponding bits of the inputs.
- **Mode Control**:
  - A 2-bit op_mode signal selects the operation:
    - 00: BW_OR
    - 01: BW_AND
    - 10: BW_NOT

        - 11: BW_XOR
    - Multiplexers route the appropriate output based on op_mode.
- **Combinational Nature**: Produces instantaneous results without clocking.

Signal Descriptions

- **Inputs**:
    - a_in: 8-bit input (e.g., from the Accumulator).
    - b_in: 8-bit input (used for OR, AND, XOR; ignored for NOT).
    - op_mode[1:0]: 2-bit control (00: OR, 01: AND, 10: NOT, 11: XOR).
- **Outputs**:
    - bw_out: 8-bit result of the selected bitwise operation.

Operation

- **BW_OR (op_mode = 00)**: Sets each output bit to 1 if either a_in or b_in bit is 1.
- **BW_AND (op_mode = 01)**: Sets each output bit to 1 only if both a_in and b_in bits are 1.
- **BW_NOT (op_mode = 10)**: Inverts each bit of a_in, ignoring b_in.
- **BW_XOR (op_mode = 11)**: Sets each output bit to 1 if a_in and b_in bits differ.
- **Data Flow**: The 8-bit bw_out is output for further processing or storage.

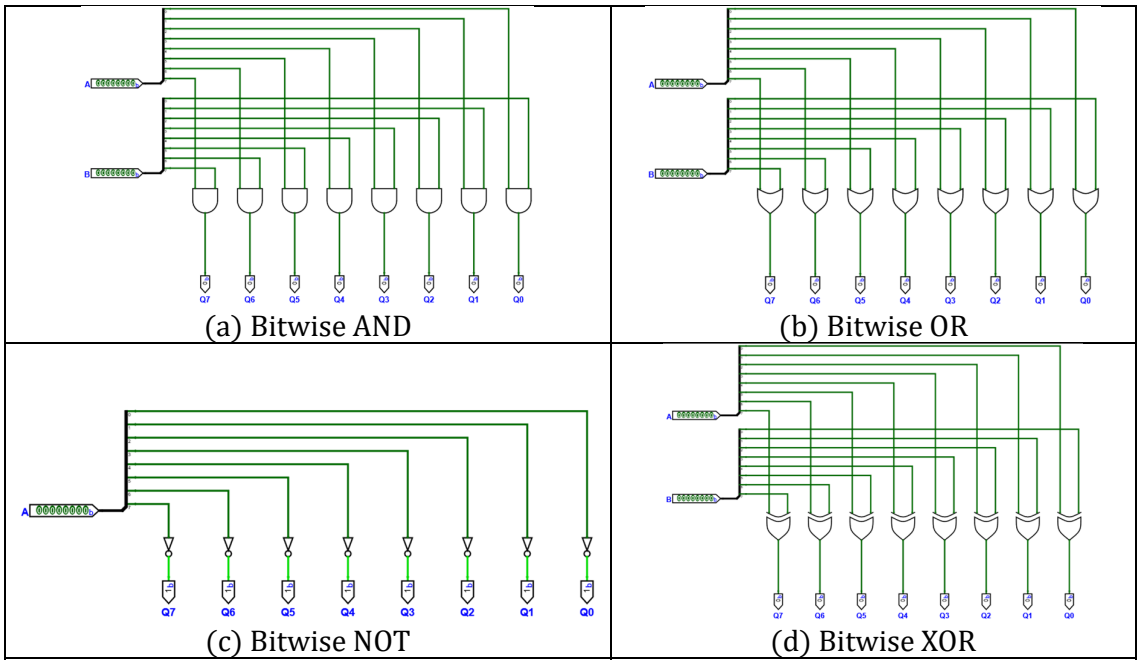

Figure 3.4.5: Sub-circuits of the Bitwise Operations.

D. Shifting and Rotation Circuit

The Shifting and Rotation Circuit is a versatile sub-component of the ALU. It integrates shifting and rotation operations, supporting the **SHL**, **SHR**, **ROL**, and **ROR** instructions. This circuit enables efficient bit manipulation, such as logical shifts for multiplication/division and rotations for cyclic data handling, with configurable mode, direction, and amount.

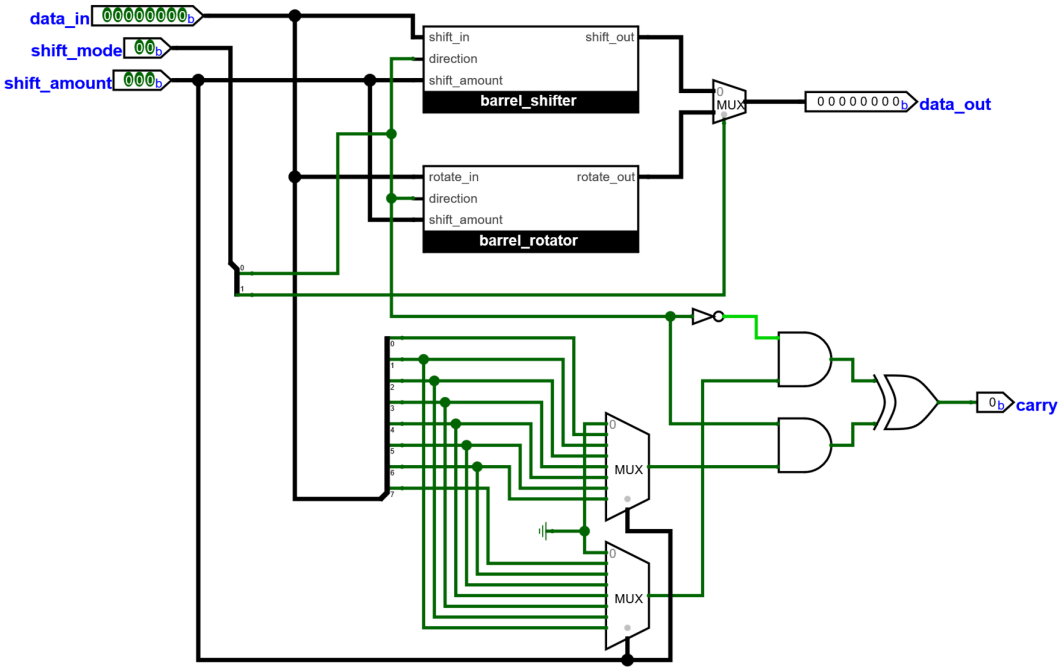

Figure 3.4.6: Shifting and Rotation Circuit

Here, the carry bit is only applicable in case of ROL and ROR as the left most and right most bit is shifted to carry flag also with 1 bit shift. For a shift amount more than 1 bit is hence predicted using a combination logic circuit along with 2 mux.

I. Shifting Circuit

i. Barrel Shifter (Left)

The Barrel Shifter (Left) is a component of the core Shifting Circuit, designed to perform leftward bit shifts on an 8-bit input. It supports the SHL (Shift Left) instruction, enabling efficient multiplication by powers of 2 and bit manipulation for the Accumulator's data.

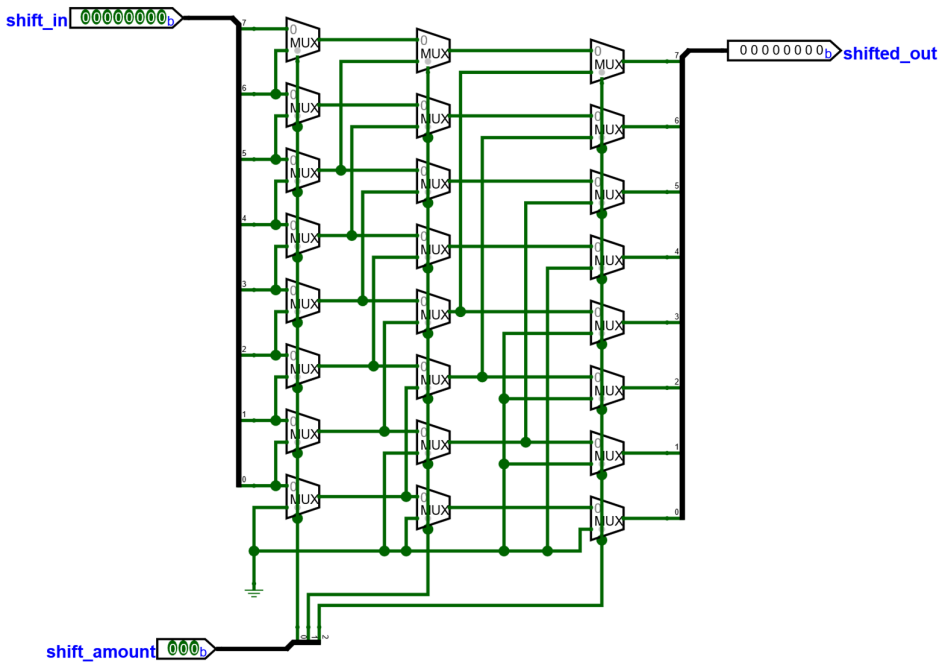

Figure 3.4.7: Barrel Shifter (Left)

Design and Structure

- **Bit Width**: 8 bits, processing an 8-bit input (shift_in) to produce an 8-bit output (shifted_out).
- **Core Elements**:
  - Utilizes a combinational network of multiplexers arranged in a barrel structure to shift bits left by a specified amount.
  - Each stage shifts by one position, with multiple stages controlled by a shift_amount input to achieve shifts from 0 to 7 positions.
- **Shift Control**:
  - A 3-bit shift_amount (0 to 7) determines the number of left shifts.
  - Bits shifted out of the leftmost position are lost, and vacant rightmost positions are filled with zeros.
- **Combinational Nature**: Operates without clocking, providing instantaneous output based on input and shift amount.

Signal Descriptions

- **Inputs**:
  - shift_in: 8-bit input data to be shifted (e.g., from the Accumulator).
  - shift_amount: 3-bit value (0 to 7) specifying the shift distance.

- **Outputs**:
    - shifted_out: 8-bit result after left shift.

Operation

- **Shift Process**: The circuit routes each bit of shift_in through multiplexers based on shift_amount. For example, a shift_amount of 2 moves bit 0 to bit 2, bit 1 to bit 3, etc., filling lower bits with 0.
- **Data Flow**: The shifted 8-bit result is output as shifted_out, available for further ALU operations or register storage.

ii. Shifter with 2 modes

The Shifter with 2 Modes is an enhanced component of the core Shift and Rotate circuit. It supports both left (**SHL**) and right (**SHR**) shifts on an 8-bit input, controlled by a direction signal. This circuit uses the Left Shifting circuit to enable flexible bit manipulation for multiplication, division by powers of 2, and data alignment, supporting the **SHL** and **SHR** instructions.

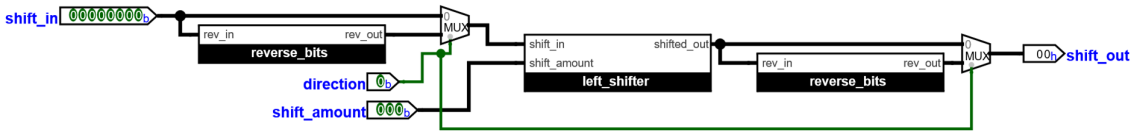

Figure 3.4.8: Main Shifting Circuit

Design and Structure

- **Bit Width**: 8 bits, processing an 8-bit input (shift_in) to produce an 8-bit output (shift_out).
- **Core Elements**:
    - Combines a left-shifter network with a reverse-bits stage to enable right shifts.
    - Uses multiplexers in a barrel configuration, with each stage shifting by one position.
    - A 3-bit shift_amount (0 to 7) controls the shift distance.
- **Mode Control**:
    - A 1-bit direction signal (0 for left shift, 1 for right shift) selects the shift direction.
    - For right shifts, the input is reversed, shifted left, then reversed again to simulate a right shift.

- **Combinational Nature**: Operates without clocking, providing instantaneous output based on inputs and controls.

Signal Descriptions

- **Inputs**:
  - shift_in: 8-bit input data to be shifted (e.g., from the Accumulator).
  - shift_amount: 3-bit value (0 to 7) specifying the shift distance.
  - direction: 1-bit control (0 for left, 1 for right).
- **Outputs**:
  - shift_out: 8-bit result after shifting.

Operation

- **Left Shift (direction = 0)**: Shifts shift_in left by shift_amount positions, filling rightmost bits with 0. Bits shifted out are lost.
- **Right Shift (direction = 1)**: Reverses shift_in, shifts left by shift_amount, then reverses again, effectively shifting right and filling leftmost bits with 0.
- **Data Flow**: The shifted 8-bit result is output as shift_out, available for further ALU operations or register storage.

II. Rotation Circuit

i. Barrel Rotation (Left)

It is a component of the core Rotation Circuit, designed to perform a left rotation on an 8-bit input. It supports the **ROL** (Rotate Left) instruction, where bits shifted out of the leftmost position are reintroduced to the rightmost position, preserving all data without loss.

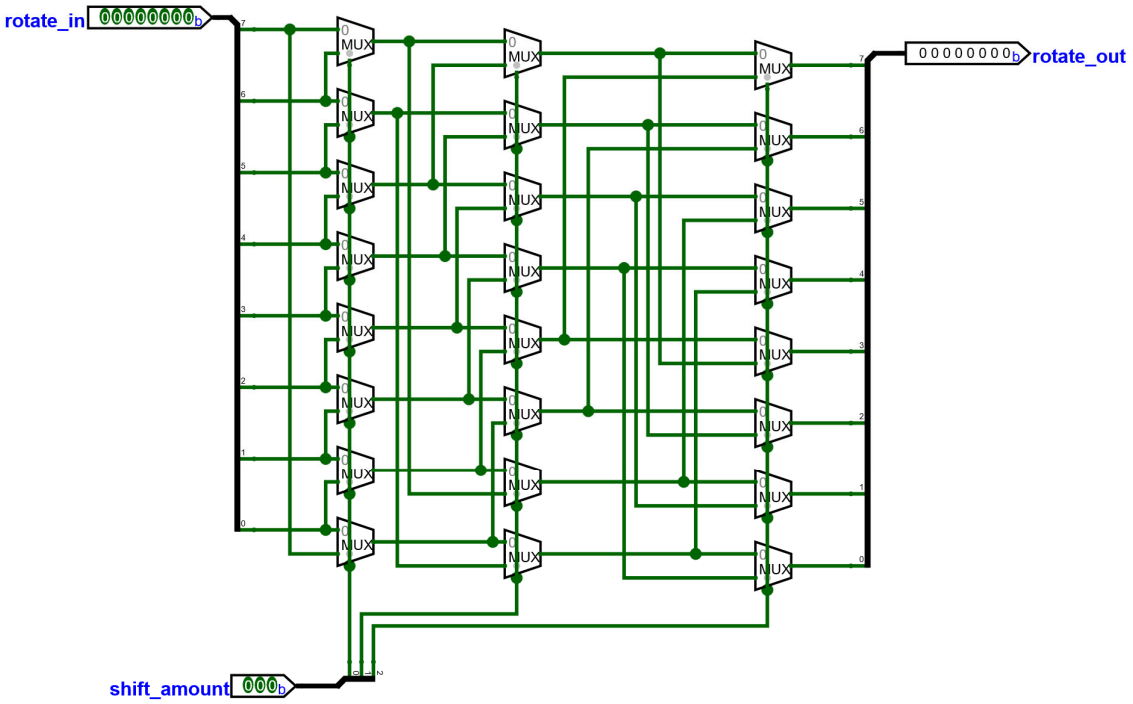

Figure 3.4.9: Barrel Rotator (Left)

Design and Structure

- **Bit Width**: 8 bits, processing an 8-bit input (rotate_in) to produce an 8-bit output (rotate_out).
- **Core Elements**:
  - Uses a combinational network of wire connections to cyclically shift bits.
  - Bits are rotated left by a specified shift_amount, with the leftmost bit wrapping around to the rightmost position.
- **Shift Control**:
  - A 3-bit shift_amount (0 to 7) determines the number of positions to rotate.
  - Implemented using a barrel-like structure with multiplexers to select the rotation distance.
- **Combinational Nature**: Operates without clocking, providing instantaneous output based on input and shift amount.

Signal Descriptions

- **Inputs**:
  - rotate_in: 8-bit input data to be rotated (e.g., from the Accumulator).
  - shift_amount: 3-bit value (0 to 7) specifying the rotation distance.

- **Outputs**:
  - rotate_out: 8-bit result after left rotation.

Operation

- **Rotation Process**: Shifts rotate_in left by shift_amount positions, with the bit shifted out of the leftmost position moving to the rightmost position. For example, with shift_amount = 1, bit 0 moves to bit 7, bit 1 to bit 0, and so on.
- **Data Flow**: The rotated 8-bit result is output as rotate_out, available for further ALU operations or register storage.

ii. Rotation with 2 modes

The Rotation with 2 Modes is an enhanced component of the core Shift and Rotate circuit, designed to perform both left (**ROL**) and right (**ROR**) rotations on an 8-bit input. It supports the **ROL** and **ROR** instructions, where bits shifted out are reintroduced to the opposite end, preserving all data without loss.

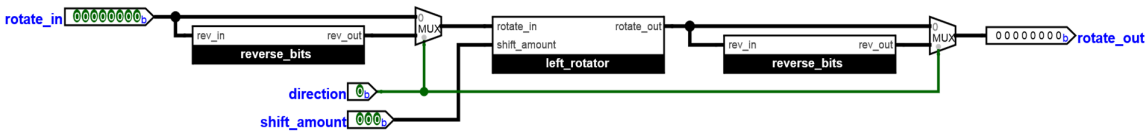

Figure 3.4.9: Rotation Circuit

Design and Structure

- **Bit Width**: 8 bits, processing an 8-bit input (rotate_in) to produce an 8-bit output (rotate_out).
- **Core Elements**:
  - Combines a reverse-bits stage with a left rotator to enable bidirectional rotation.
  - Uses a barrel shifter structure with multiplexers, controlled by shift_amount and direction.
- **Mode Control**:
  - A 1-bit direction signal (0 for left, 1 for right) selects the rotation direction.
  - A 3-bit shift_amount (0 to 7) specifies the rotation distance.
  - For right rotation, the input is reversed, rotated left, then reversed again.
- **Combinational Nature**: Operates without clocking, providing instantaneous output based on inputs and controls.

Signal Descriptions

- **Inputs**:
  - rotate_in: 8-bit input data to be rotated (e.g., from the Accumulator).
  - shift_amount: 3-bit value (0 to 7) specifying the rotation distance.
  - direction: 1-bit control (0 for left, 1 for right).
- **Outputs**:
  - rotate_out: 8-bit result after rotation.

Operation

- **Left Rotation (direction = 0)**: Rotates rotate_in left by shift_amount positions, with the leftmost bit moving to the rightmost position.
- **Right Rotation (direction = 1)**: Reverses rotate_in, rotates left by shift_amount, then reverses again, effectively rotating right with the rightmost bit moving to the leftmost position.
- **Data Flow**: The rotated 8-bit result is output as rotate_out, available for further ALU operations or register storage.

E. Flag Register (`flag_reg`)

The Flag Register (flag_reg) is an additional component of the Modified SAP-1 Microprocessor, designed to store status flags that reflect the outcome of ALU operations. It supports the tracking of carry, zero, sign, and parity conditions, which are used for conditional instructions like JZ and JNZ.

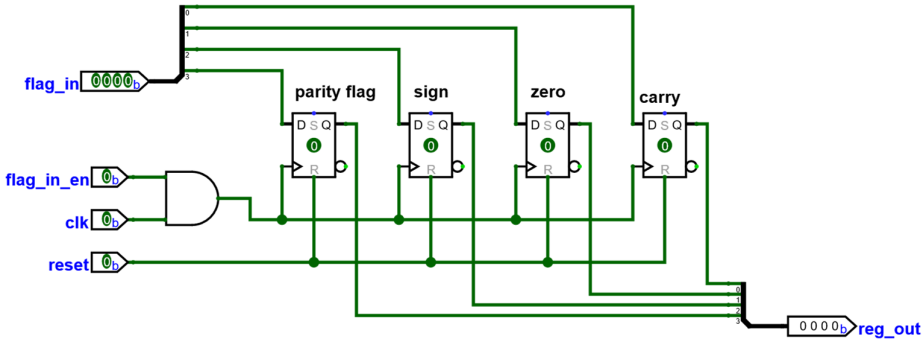

Figure 3.4.10: Flag Register (`flag_reg`)

Design and Structure

- **Bit Width**: 4 bits, representing four flags: Carry (C), Zero (Z), Sign (S), and Parity (P).
- **Core Elements**:
  - Comprises four D-type flip-flops, each storing one flag bit.
  - Operates synchronously with a clock signal for flag updates.
- **Control Logic**:
  - Includes an enable signal (flag_in_en) to load new flag values from flag_in.
  - Reset clears all flags to 0 asynchronously.
- **Output**: Drives flag values to reg_out for use by the controller or ALU.

Signal Descriptions

- **Inputs**:
  - flag_in: 4-bit input data containing new flag values (e.g., from ALU).
  - flag_in_en: 1-bit enable signal (active high) to load flag_in.
  - clk: Clock signal triggering updates on the rising edge.
  - reset: Asynchronous reset signal (active high) to clear flags.
- **Outputs**:
  - reg_out: 4-bit output reflecting the current flag states.

Operation

- **Flag Update**: When flag_in_en is high, flag_in values are loaded into the flip-flops on the next clk rising edge. If low, the current flags are retained.
- **Reset**: Asserting reset sets all flags to 0, overriding other inputs.
- **Data Flow**: The 4-bit flag state is output via reg_out, available for conditional branching or status monitoring.

F. Flag Updater

The Flag Updater is a combinational logic circuit within the AL. It calculates the status flags—Carry (C), Zero (Z), Sign (S), and Parity (P)—based on the result of ALU operations, providing inputs for the Flag Register to support conditional instructions like JZ and JNZ.

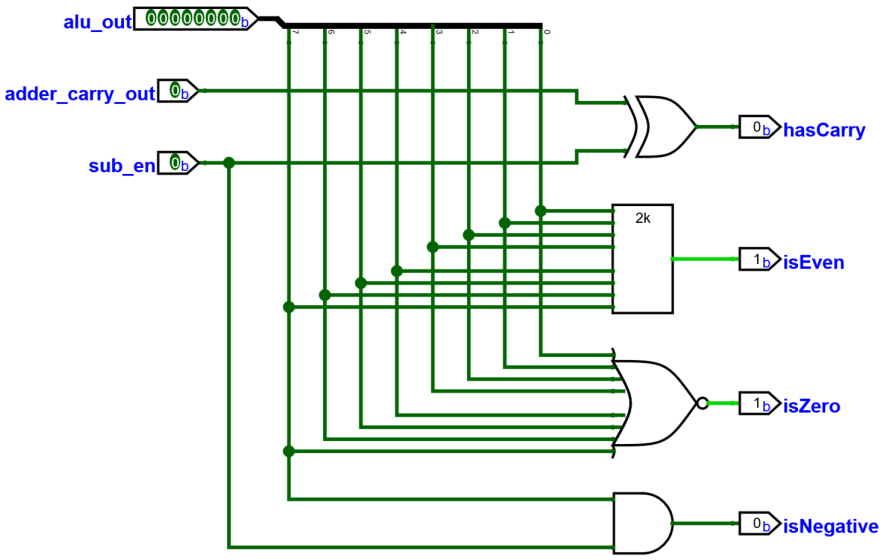

Figure 3.4.11: Flag Updater

Design and Structure

- **Input Width**: Accepts an 8-bit result (alu_result) and a 1-bit carry_out from the ALU.
- **Core Elements**:
  - **Zero Flag (Z)**: Generated by a 8-input NOR gate, set to 1 if all bits of alu_result are 0.
  - **Sign Flag (S)**: Derived from the most significant bit (bit 7) of alu_result, set to 1 for negative values in two's complement.
  - **Carry Flag (C)**: Directly uses the carry_out from the ALU, indicating overflow or borrow.
  - **Parity Flag (P)**: Computed using a built-in even parity encoder of Logisim, set to 1 if the number of 1s in alu_result is even.
- **Combinational Nature**: Produces flag values instantly based on ALU outputs, requiring no clocking.

Signal Descriptions

- **Inputs**:
  - alu_result: 8-bit result of the ALU operation.
  - carry_out: 1-bit carry or borrow output from the ALU.
- **Outputs**:
  - flag_out: 4-bit output containing [C, Z, S, P] flags.

Operation

- **Flag Calculation**:
  - C is set to carry_out's value.
  - Z is 1 if alu_result is 0, else 0.
  - S is 1 if bit 7 of alu_result is 1, else 0.
  - P is 1 if the parity of alu_result is even, else 0.
- **Data Flow**: The 4-bit flag_out is passed to the Flag Register for storage and later use in control logic.

3.5 Instruction Register (`ins_reg`)

The Instruction Register (IR) is a key component of the SAP-1, designed to store the current 8-bit instruction fetched from memory. It serves as a temporary holding register, enabling the Controller-Sequencer to decode and execute the instruction by distributing its opcode and operand to relevant units.

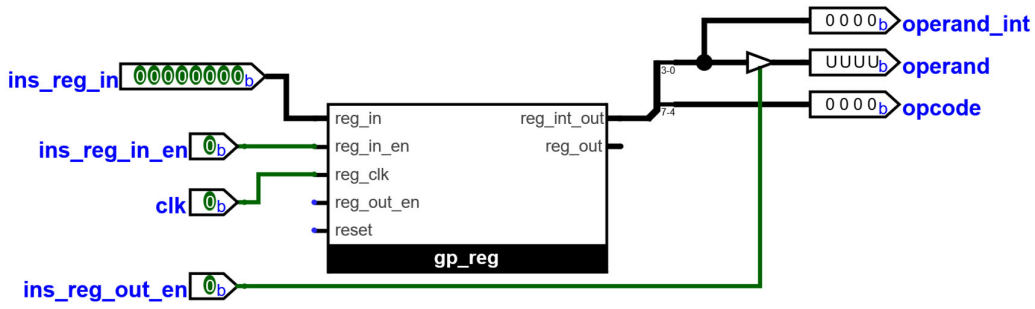

Figure 3.5.1: Instruction Register (`ins_reg`)

Design and Structure

- **Bit Width**: 8 bits, accommodating a complete 8-bit instruction word (e.g., opcode + operand).
- **Core Elements**:
  - Comprises the general-purpose register for synchronous storage of the instruction.
  - Includes tri-state buffers for output to the central bus.
- **Control Logic**:
  - An enable signal (ins_in_en) loads the instruction from the bus.
  - A clock signal (clk) synchronizes updates on the rising edge.
  - A reset signal clears the register asynchronously.

Signal Descriptions

- **Inputs**:
  - ins_in: 8-bit input data from the central bus (fetched instruction).
  - ins_in_en: 1-bit enable signal (active high) to load ins_in.
  - clk: Clock signal triggering updates on the rising edge.
  - reset: Asynchronous reset signal (active high) to clear the register.
- **Outputs**:
  - ins_out: 8-bit output of the stored instruction, driven to the bus when enabled.

Operation

- **Instruction Load**: When ins_in_en is high, the ins_in value is loaded into the flip-flops on the next clk rising edge; otherwise, the current instruction is retained.
- **Reset**: Asserting reset sets all bits to 0, clearing the register.
- **Data Flow**: The 8-bit ins_out is output to the central bus or Controller-Sequencer for decoding and execution.

3.6 SRAM

The SRAm is the main memory of the SAP-1 architecture which consists of 16 SRAM Cells and a MAR for selecting the specific Cell for memory operations.

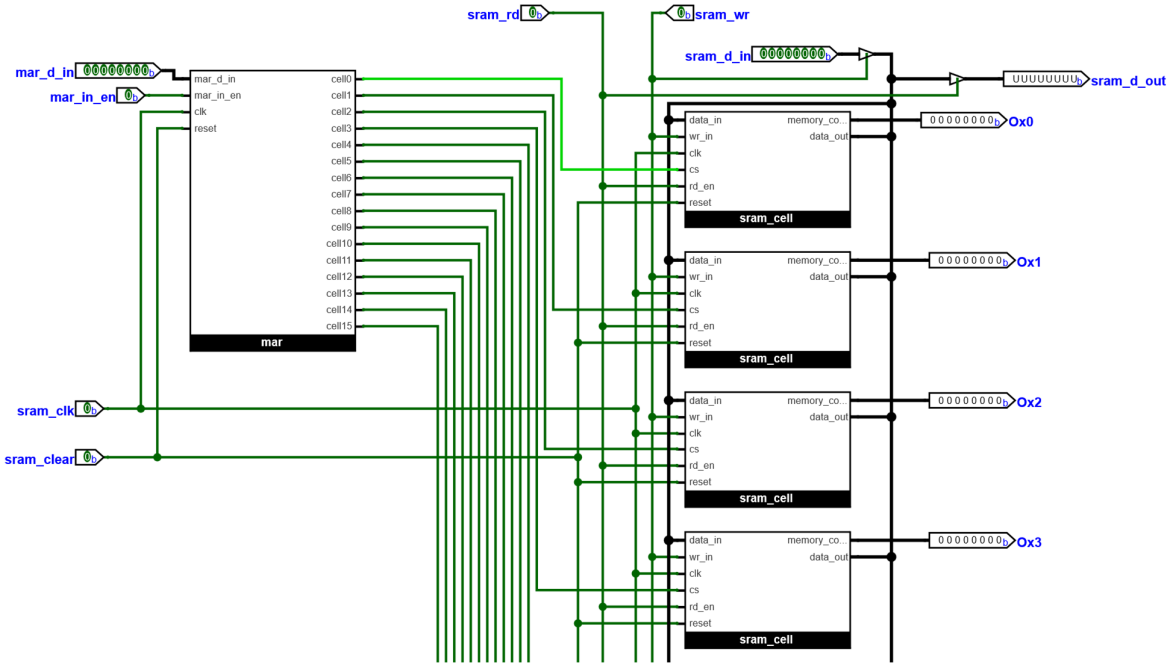

Figure 3.6.1: High Level Overview of the SRAM Circuit (Partial)

A. SRAM Cell

The SRAM Cell is a basic memory element in the SRAM subsystem of the Microprocessor, designed to store a single byte of data. It is part of the 16x8 SRAM array, providing fast, volatile storage for instructions and data.

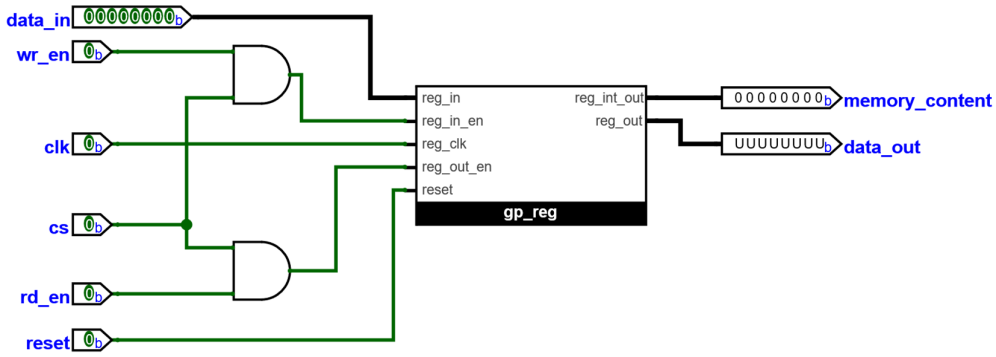

Figure 3.6.2: SRAM Cell

Design and Structure

- **Bit Width**: 8 bits, storing a full byte using the gp_reg.
- **Core Elements**:
  - Based on the gp_reg with input/output controls for read/write access.
  - Includes chip select (cs) for enabling the cell and read enable (rd_en) for output.
- **Control Logic**:
  - Write enable (wr_en) loads data from the bus.
  - Clock (clk) and reset synchronize operations.

Signal Descriptions

- **Inputs**:
  - data_in: 8-bit input data from the central bus.
  - wr_en: Write enable signal (active high).
  - rd_en: Read enable signal (active high).
  - cs: Chip select signal (active high).
  - clk: Clock for synchronous updates.
  - reset: Asynchronous reset to clear data.
- **Outputs**:
  - data_out: 8-bit stored data to the bus when read-enabled.

Operation

- **Write**: When cs and wr_en are high, data_in is loaded into the gp_reg on clk edge.
- **Read**: When cs and rd_en are high, data_out drives the stored value to the bus.
- **Data Flow**: Functions as a byte-addressable memory cell, selected via the 4-to-16 decoder in the SRAM array.

B. Decoder - 4×16 (dec_4x16)

The 4-to-16 Decoder is a critical component of the SRAM subsystem, designed to select one of the 16 SRAM cells (each storing 8 bits) based on a 4-bit address input. It enables precise addressing within the 16x8 SRAM array for read and write operations.

Design and Structure

- **Input Width**: 4 bits, accepting an address to select one of 16 cells.
- **Core Elements**:
  - Comprises combinational logic (AND gates with inverters) to generate 16 output lines.
  - Each output corresponds to one SRAM cell, active when its address matches addr[3:0].
- **Control Logic**:
  - Includes an enable signal (decoder_en) to activate the decoder.
  - Outputs are tied to the chip select (cs) lines of the SRAM cells.

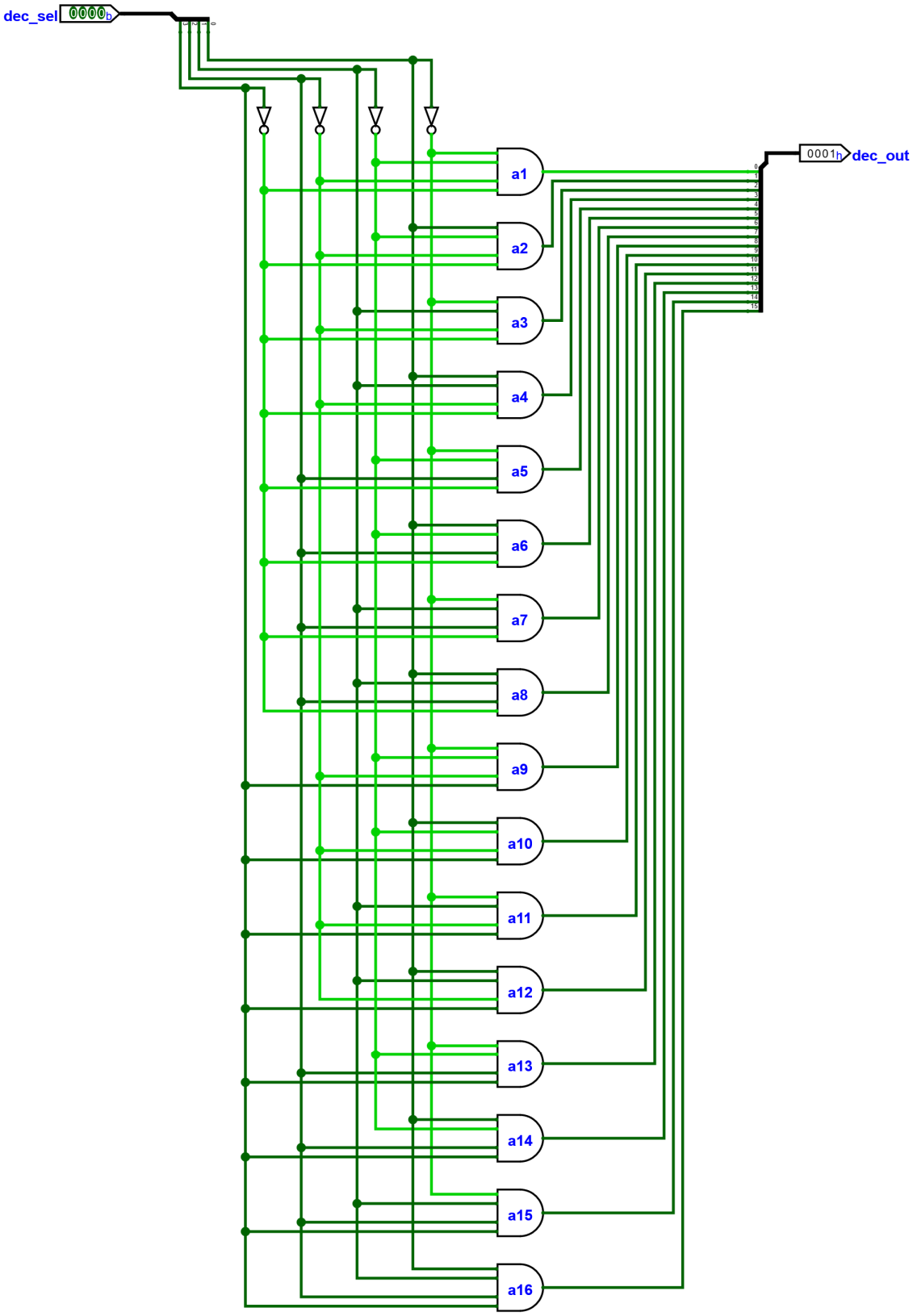

Figure 3.6.3: A 4 to 16 Decoder

C. Memory Address Register (MAR)

The Memory Address Register (MAR) is a key component of the Modified SAP-1 Microprocessor, designed to hold a 4-bit memory address to select one of the 16 SRAM cells in the 16x8 SRAM array. It utilizes the General-Purpose Register (gp_reg) for storage and integrates with the 4-to-16 decoder for cell selection during memory operations.

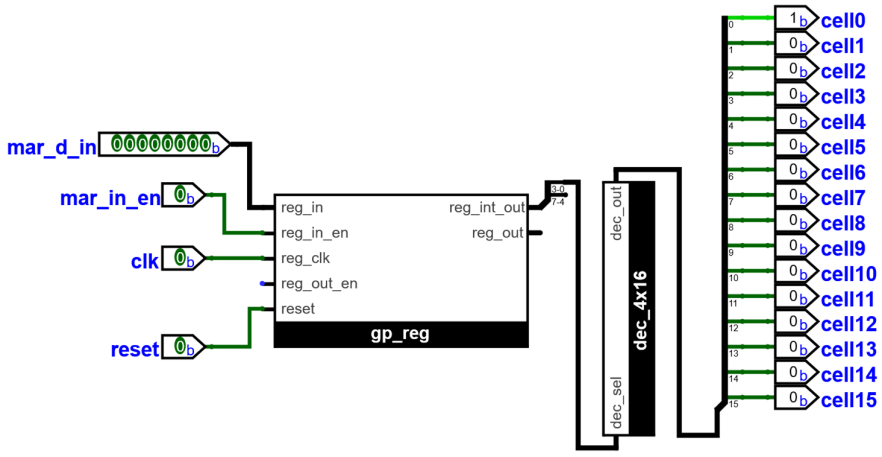

Figure 3.6.4: Memory Address Register

Design and Structure

- **Bit Width**: 4 bits, addressing 16 memory locations (0x0 to 0xF).
- **Core Elements**:
  - Employs the gp_reg as the storage element for the 4-bit address.
  - Connected to a 4-to-16 decoder, which generates 16 chip select (cs) signals based on the address.
- **Control Logic**:
  - An enable signal (mar_in_en) loads the address into the gp_reg.
  - A clock signal (clk) synchronizes updates on the rising edge.
  - A reset signal clears the register asynchronously.
  - The decoder is enabled by a decoder_en signal tied to mar_in_en.

Signal Descriptions

- **Inputs**:
  - mar_d_in: 4-bit input address from the central bus.

- mar_in_en: 1-bit enable signal (active high) to load mar_in and enable the decoder.
- clk: Clock signal for synchronous updates.
- reset: Asynchronous reset signal (active high) to clear the register.

- **Outputs**:
  - 16 chip select lines from the decoder, activating the selected SRAM cell.

Operation

- **Address Load**: When mar_in_en is high, mar_d_in is loaded into the gp_reg on the next clk edge, and the decoder activates the corresponding cs line based on mar_out.
- **Reset**: Asserting reset sets the gp_reg to 0x0, selecting cell0.
- **Data Flow**: The 4-bit reg_out_int drives the decoder, which selects one SRAM cell via its cs line for read or write operations, coordinated by the Controller-Sequencer.

3.7 Bootloader

The bootloader loads the predefined program from the external built-in ROM of Logisim into the custom SRAM before the starting the controller-sequencer.

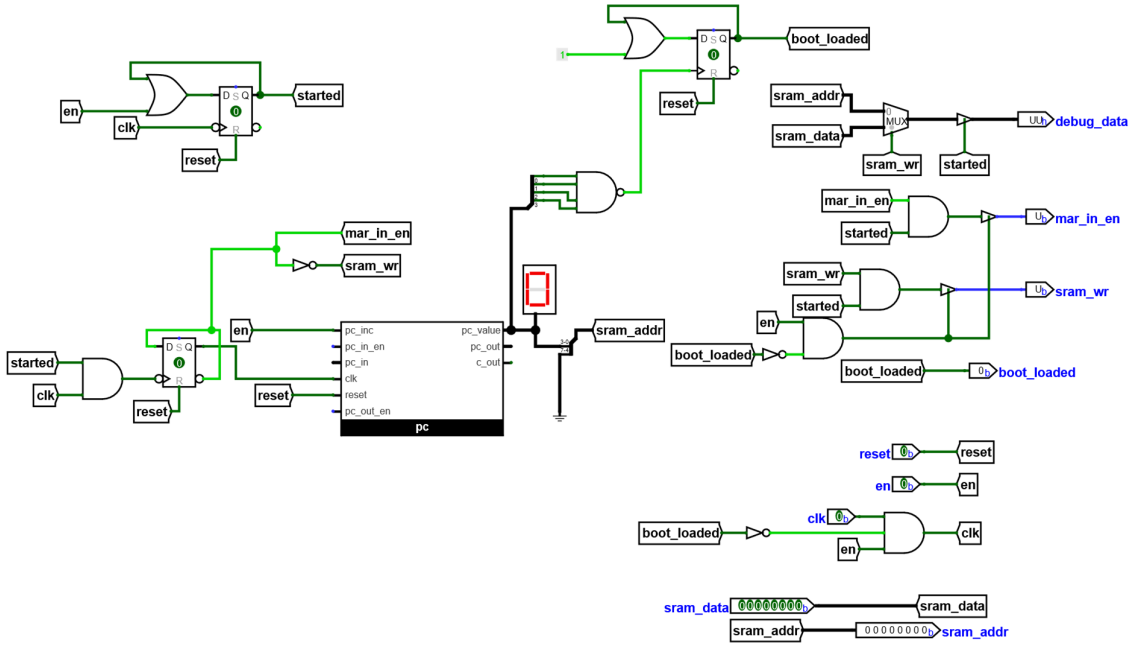

Figure 3.7: Bootloader

3.8 Controller Sequencer

This is the brain of the SAP-1 architecture which emits the control signals for leveraging the fetch-decode-execution cycle. It consists of a instruction decoder to decode the Instruction to be executed in the T4 and T5 states, a ALU Control word generator for forming the control word required to set the specific mode of operation and their corresponding control bits, and a 5-state ring counter for generating the 5 T states. The two table below demonstrates the formation of the control signal for each operation and in the T-states.

Logic Construction using the opcode, operand and T states.

Pin	MOV			JMP			Accumulated									
	LDA	LDB	STA	JMP	JZ	JNZ	SHL	SHR	RHL	RHR	ADD	SUB	INC	DEC	REV	HLT
a_in	**1**						**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	
a_out			**1**													
b_in		**1**														
b_out																
alu_out							**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	
mar_in(T4)	**1**	**1**	**1**													
ram_rd	**1**	**1**														
ram_wr			**1**													
ins_in																
ins_out(T4)	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**						
pc_inc																
pc_in(T4)				**1**	**1**	**1**										
pc_out																
Flag_in							**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	
hlt(T4)																**1**

T	a_in	a_out	b_in	b_out	alu_out	mar_in	ram_rd	ram_wr	ins_in	ins_out	pc_inc	pc_in	pc_out	hlt
T1						1							1	
T2							1		1					
T3											1			
T4						1				1		1		
T5	1	1	1		1			1						
T6														

A. Instruction Decoder

The Instruction Decoder is a vital component of the Controller sequencer, responsible for interpreting the 8-bit instruction stored in the Instruction Register (IR). It selects the specific instruction pin based on the opcode received from instruction register. As discussed before some of the operation pins require 4 bits and some of them require 5 bits for dictating them. The 4-to-16-bit decoder is used for selection of 16 combinations using the 4 bits opcode. 1 Bits is borrowed from the operand for the remaining 6 combinations to form total 22 operational pins. Here, the decode pin is used as output enable pin and it receives 4 bits opcode and 4 bits operand as input signals. These 8 bits along with the decode bit decide the single output pin to be enabled using a combination logic circuit.

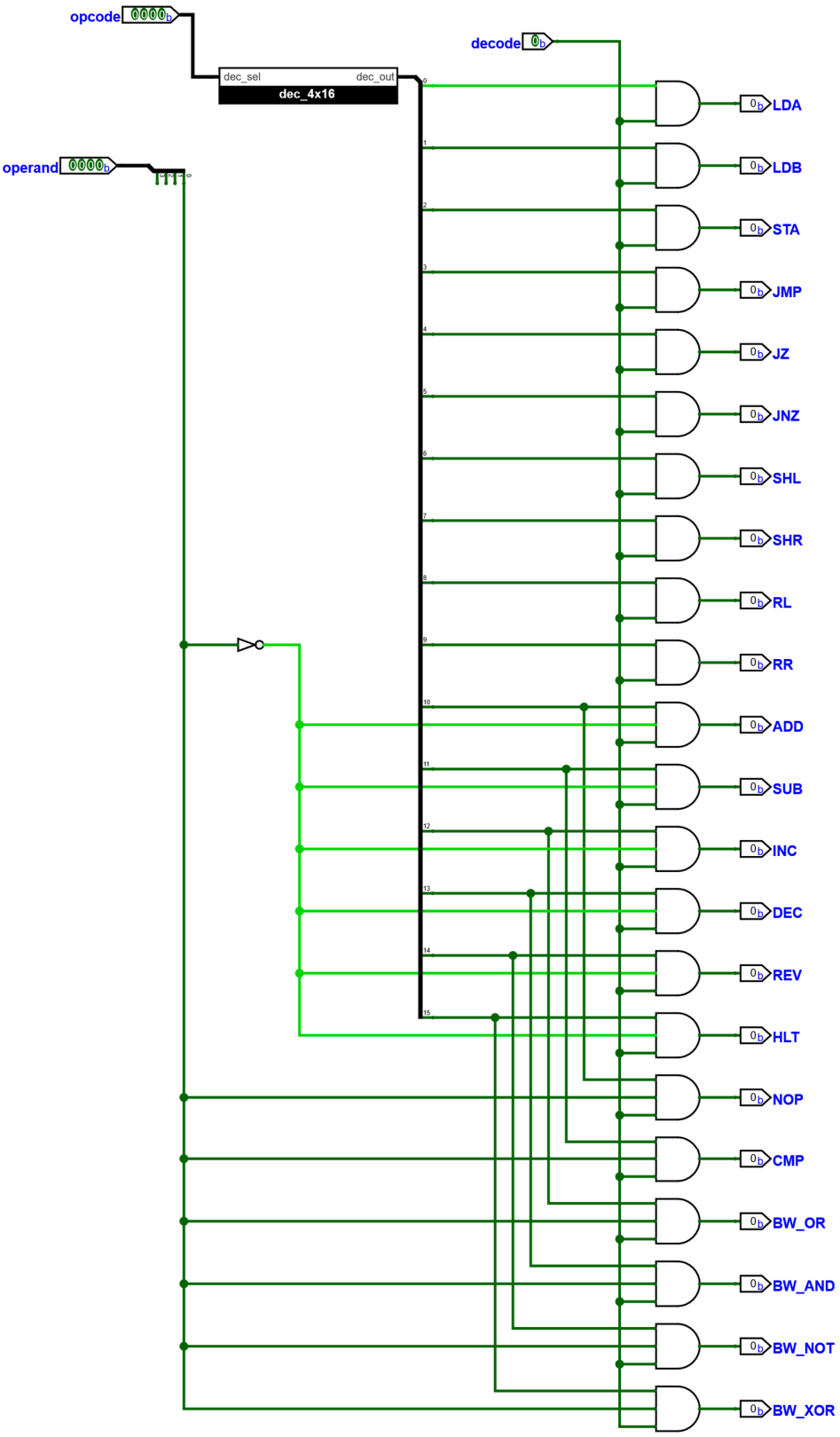

Figure 3.8.1 : Instruction Decoder

ALU Control Word Generator

This subcircuit takes the output pins of the instruction decoder as input pins and is used to generate control word for ALU which decides the mode of operations and the arguments like shifting amount and directions.

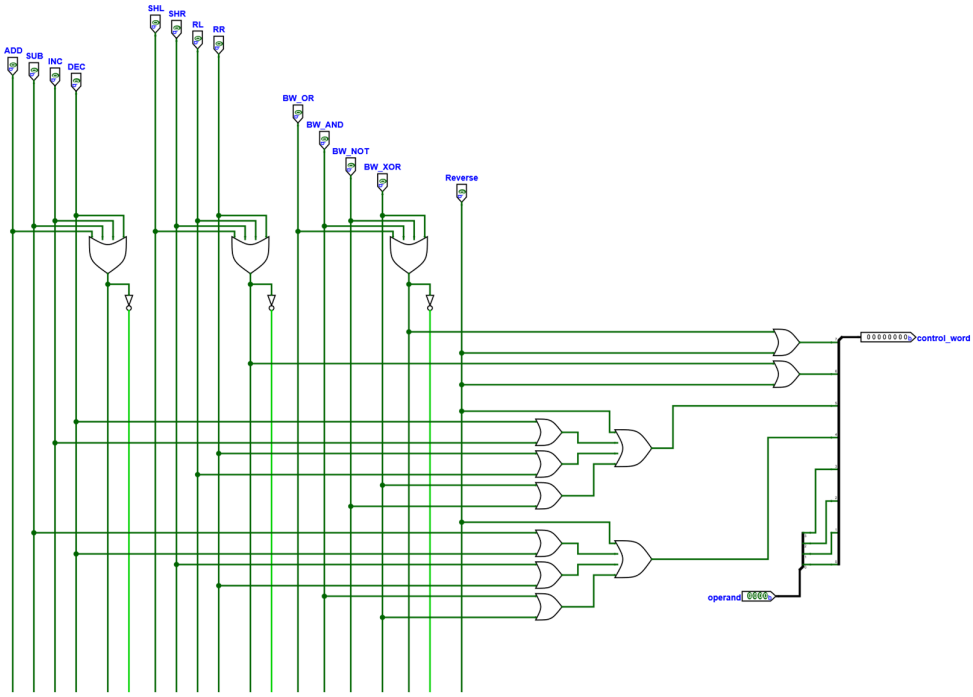

Figure 3.8.2: ALU Control Word Generator

C. Ring Counter:

This ring counter counts the 5-T states required for the Fetch-Decode-Execution cycles of the microprocessor. The Controller-Sequencer generates the control signals based on the T-states. It uses the program counter for counting T states and the decoder is used to select the 5-States using the 1st 5 counts of the PC. In the 5th count the pc_in is enabled the 0001 is loaded as input to retain the T1 state.

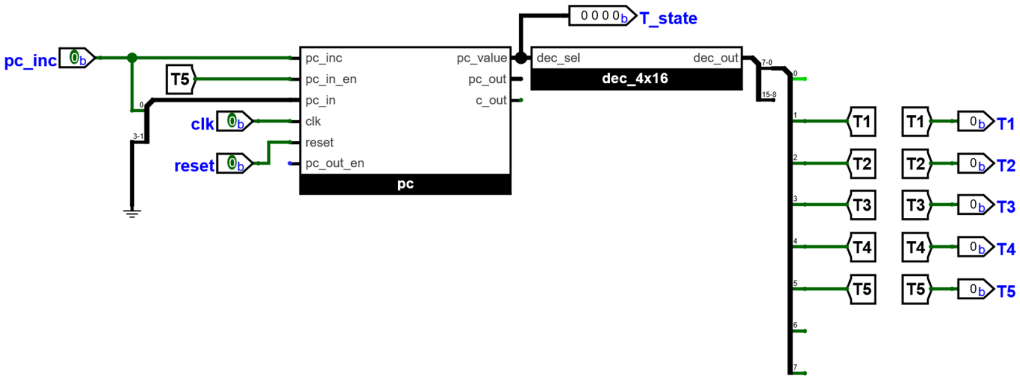

Figure 3.8.3: Ring Counter with 5-T States.

This is the final schematic of the controller sequencer which leverages these 3 components: Instruction Decoder, Ring Counter and ALU Control Word Generator to dictate the control pins using a combinational circuit.

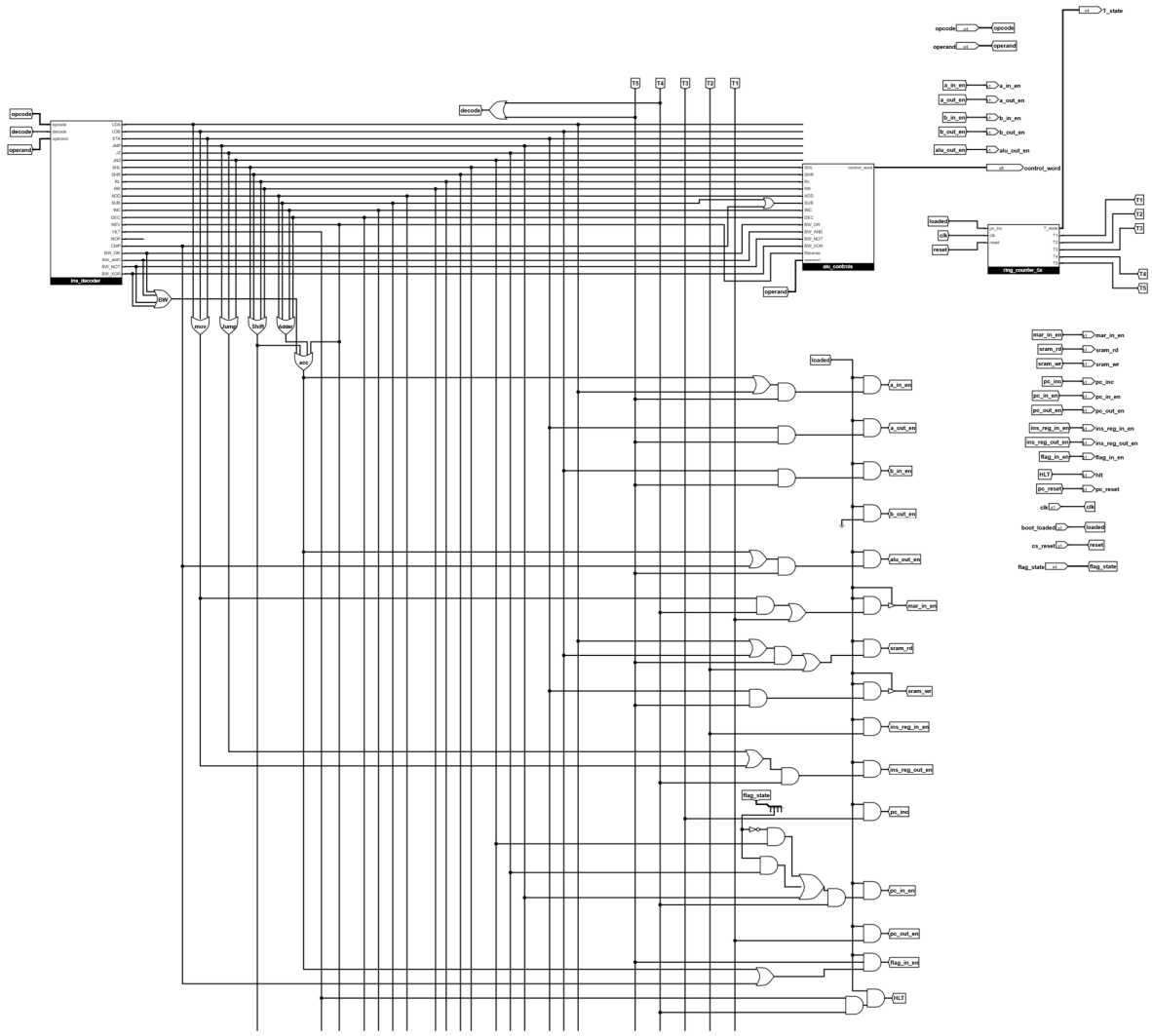

Figure 3.8.3: Controller Sequencer

3.9 Final Schematic

This the final SAP-1 architecture combining all the components such as ALU, General Purpose Registers, Program Counter, SRAM, Instruction Register, Flag Register, Controller Sequencer for automating fetch-decode-execution cycles and Bootloader for autoloading Instructions into the SRAM.

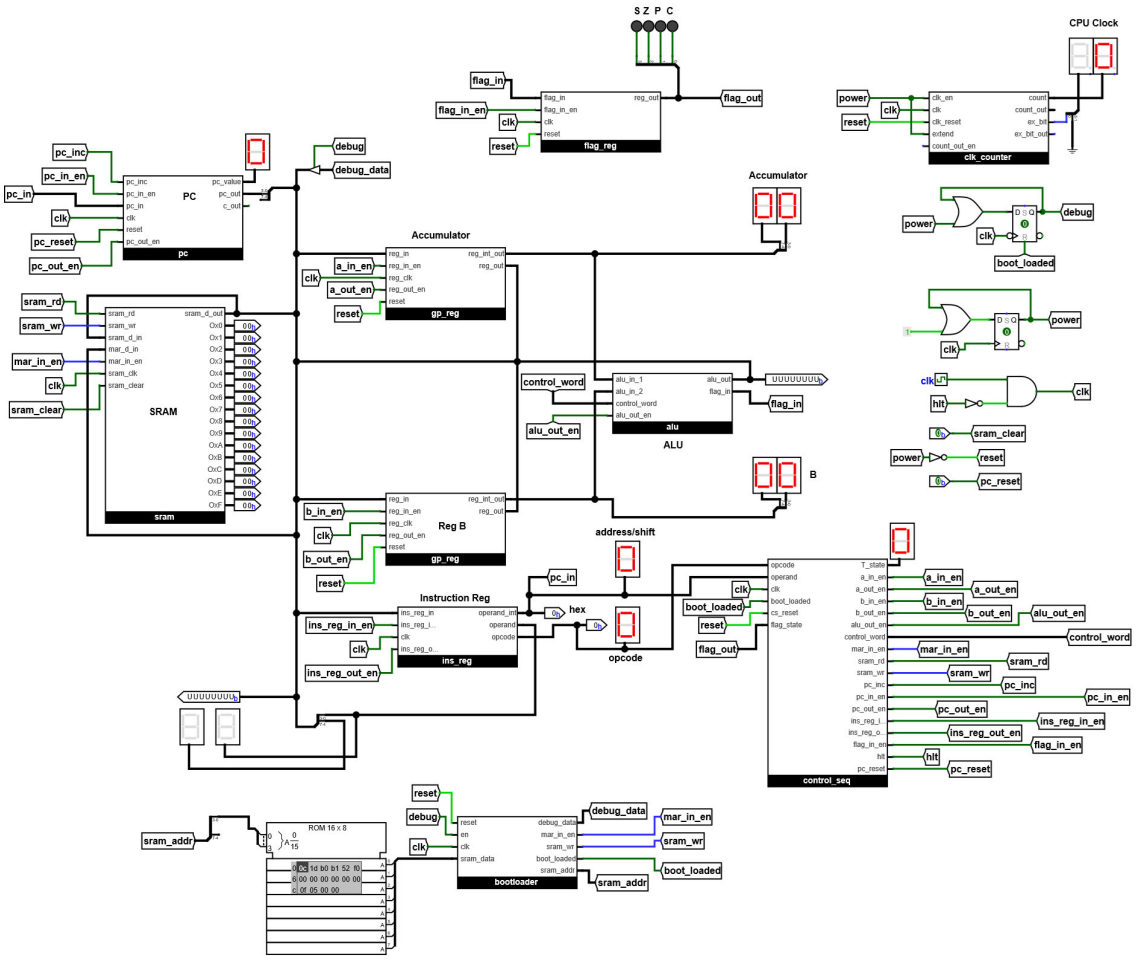

Figure 3.9.1: Final Schematic of the Modified SAP-1 Microprocessor

3.10 Testing

The implemented microprocessor is tested with various sample programs included in the zip file and successfully achieved the desired results showing the Addition, Subtraction, Increment, Decrement, Compare, Conditional Jumps, Sifting and Rotations, Bitwise Operations, the Transfer Operations as well the control operations. The **Learn** page of the website provided in the Reference section includes the step-by-step overview of the rigorously tested programs demonstrating the appropriate behavior.

3.11 Conclusion

The Modified SAP-1 Microprocessor successfully demonstrates a functional 8-bit architecture suitable for educational purposes and small-scale applications. Its modular design, reusing components like the gp_reg and decoder for MAR and SRAM cells, ensures flexibility and simplicity. The ALU's support for diverse operations and the SRAM's reliable addressing via the 4-to-16 decoder highlight its robustness. However, there are more scopes to extend the capabilities of this architecture like stack and subroutine implementation enabling a more advanced feature rich architecture. Overall, this architecture provided a ground for understanding and developing the idea of how complex architectures may work and opens up the door for experimenting with more advanced architectural microprocessor implementations.

3.12 References

Github Repo: https://github.com/FarhanXTanvir/SAP1

Youtube Demo Video: https://tinyurl.com/ete404-sap1

Assembler: https://farhanxtanvir.github.io/SAP1