Chittagong University of Engineering and Technology

Department of Electronics and Telecommunication Engineering

PROJECT REPORT

Name of the project
Design and Implementation of an SAP-1 Microprocessor with Assembler

Course No:	**ETE 404**
Course Title:	**VLSI Technology Sessional**
Date of Submission:	**06.10.25**

Submitted By:	Submitted To:
Tahmid Fuad Khan **ID: 2008055**	**Arif Istiaque Rupom** Assistant Professor, Dept. of ETE, CUET

1. Introduction

The objective of this project was to design and implement a fully operational SAP-1 (Simple-As-Possible) microprocessor architecture using **Logisim**, integrating both hardware and software components for automatic program execution. The processor was constructed from fundamental digital modules including the general-purpose register, arithmetic logic unit (ALU), 5-to-32 decoder, SRAM memory block, instruction register, ring counter, program counter, and **control sequencer** . Additional subsystems such as the *barrel shifter* and *ring rotator* were implemented to support extended logical and data manipulation operations.

A custom **assembler** was developed in Python within the Visual Studio Code environment to serve as the software front-end of the system. The assembler converts assembly-level source code into 10-bit machine code compatible with the SAP-1 instruction format, producing output in v3.0 hexadecimal word format. This machine code is loaded into the SRAM through a **RAM Auto-Loader** that operates via a debug protocol, enabling seamless program initialization without manual intervention. Once the memory is loaded, the microprocessor executes the program autonomously, reflecting the correct data and status on the output display. The complete system demonstrates a unified design flow starting from instruction compilation to hardware-level execution, showcasing the fundamental principles of computer architecture and automation.

2. Project Overview and Objective

2.1 Goal

The primary goal of this project was to design and implement a fully functional, automatic SAP-1 (Simple-As-Possible) microprocessor system that integrates both hardware and software components into a unified framework. The hardware section was developed in **Logisim**, consisting of fundamental digital modules such as registers, an arithmetic logic unit (ALU), a control sequencer, and memory components. The software section, implemented in Python, acts as a custom **assembler** capable of converting assembly-level programs into 10-bit machine code that conforms to the instruction format of the designed processor. The system is capable of automatic program loading, execution, and output display without manual intervention, reflecting the core principles of computer organization and automation.

2.2 Scope

This project encompasses the complete design flow of a basic computer system — from logical design and simulation to code compilation and execution. The implemented system includes:

- **CPU Design and Control Sequencing:** Incorporates modules such as the general-purpose register, instruction register, ALU, program counter, and control sequencer.
- **RAM Auto-Loading Mechanism:** A debugging-based auto-loader was implemented to load machine code directly into SRAM without manual initialization.
- **Assembler Software:** A Python-based assembler translates human-readable assembly code into 10-bit hexadecimal machine code in v3.0 format.
- **Unique Feature Extension:** Additional modules, including a *barrel shifter* and *ring rotator*, were developed to support enhanced logical and data manipulation operations beyond the classical SAP-1 instruction set.

Overall, this work establishes a complete end-to-end design process — from instruction design and memory interfacing to automatic execution — demonstrating both the theoretical and practical aspects of digital computer architecture.

3. System Architecture

3.1 Overall Block Diagram

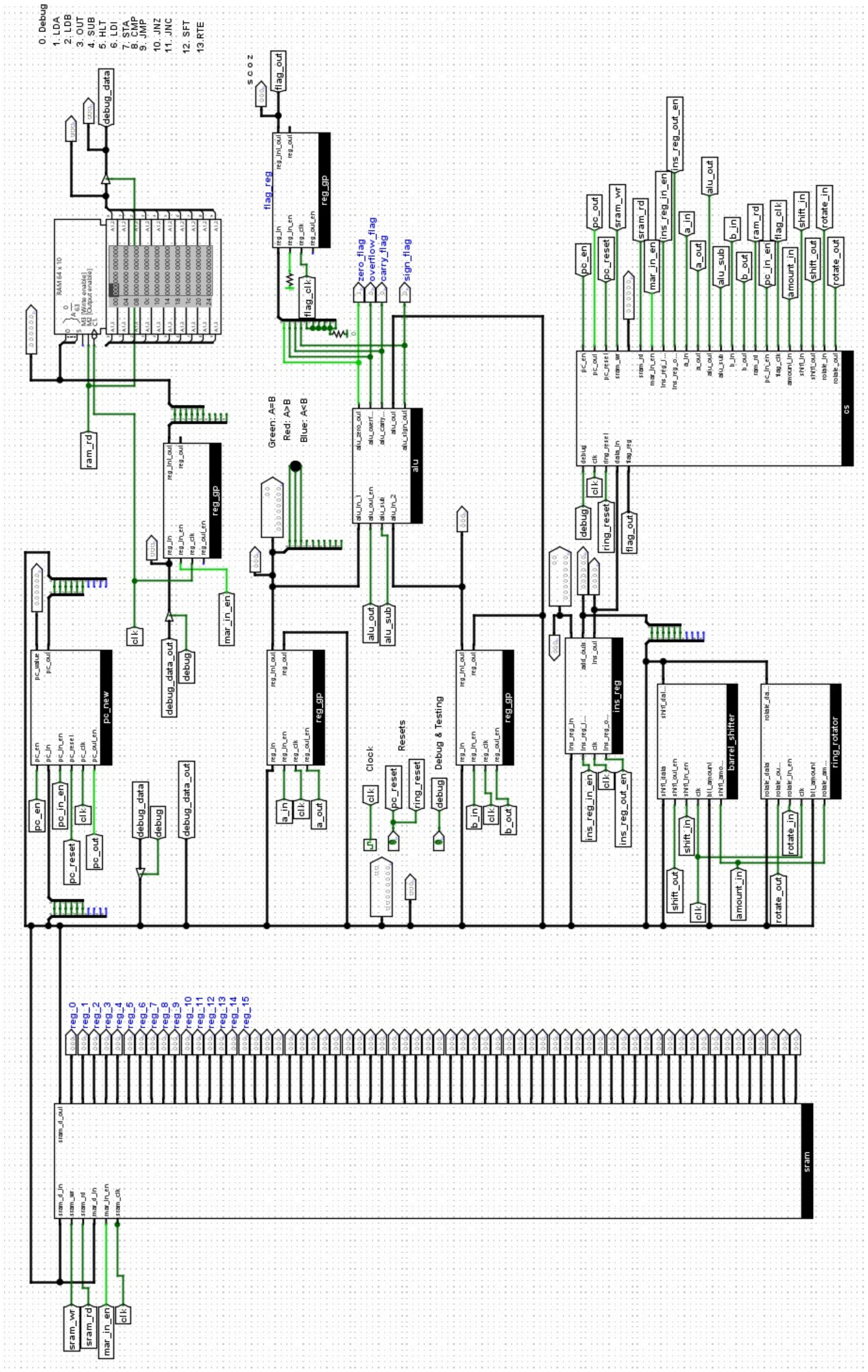


Figure 1: Overall block diagram of the SAP-1 based system

3.2 Major Components

This section describes each core module of the SAP-1 microprocessor in detail. Every component was designed as an individual circuit block in **Logisim** and interconnected through a common 10-bit data bus and 6-bit address bus. The following subsections explain their working principles, design purpose, and interfacing within the complete architecture.

3.2.1 General Purpose Register

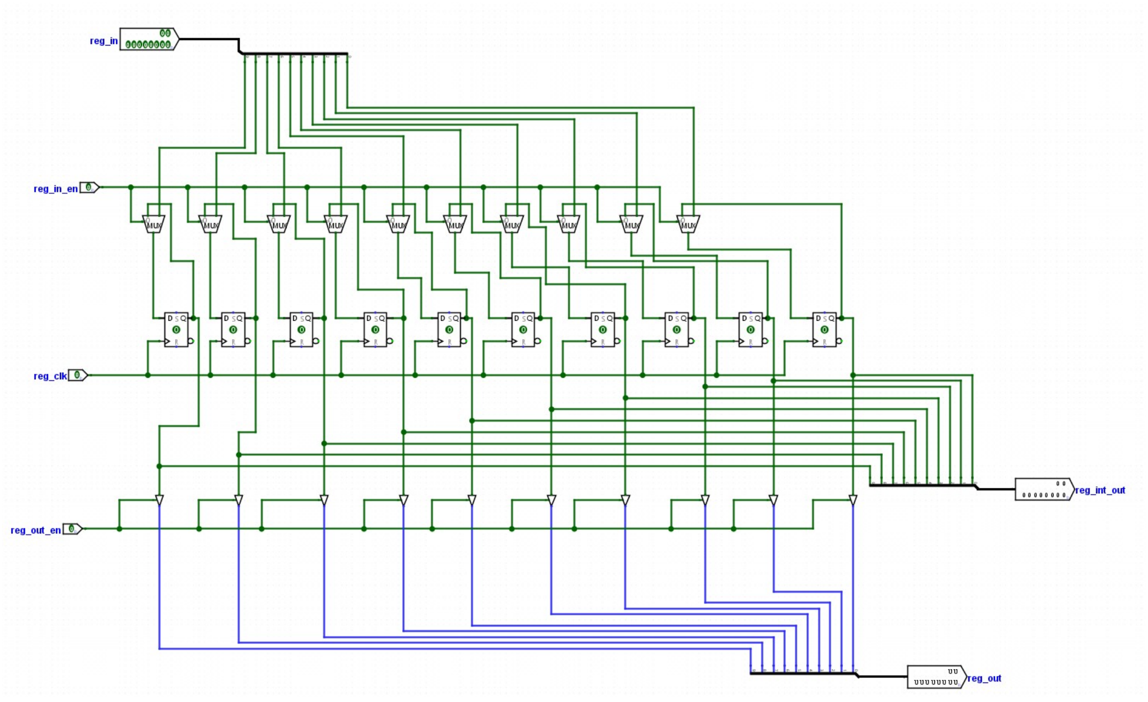


Figure 2: General Purpose Register

Description: General-purpose registers are used for temporary data storage and manipulation within this SAP-1 architecture. The register acts as a fast-access storage element, holding intermediate results, operands, or data fetched from memory. It is implemented using D-type flip-flops arranged in parallel to store each bit of the 10-bit word, allowing synchronous loading and output operations.

Working Principle: The register operates based on control signals that determine when data is loaded from the bus and when it is output back to the bus. When the **reg_in_en** signal is asserted, the data present on the 10-bit bus is latched into the register on the rising edge of the **clk** signal. Conversely, when the **reg_out_en** signal is activated, the contents of the register are driven onto the bus, making them available for other components such as the ALU or memory.

3.2.2 Arithmetic Logic Unit (ALU)

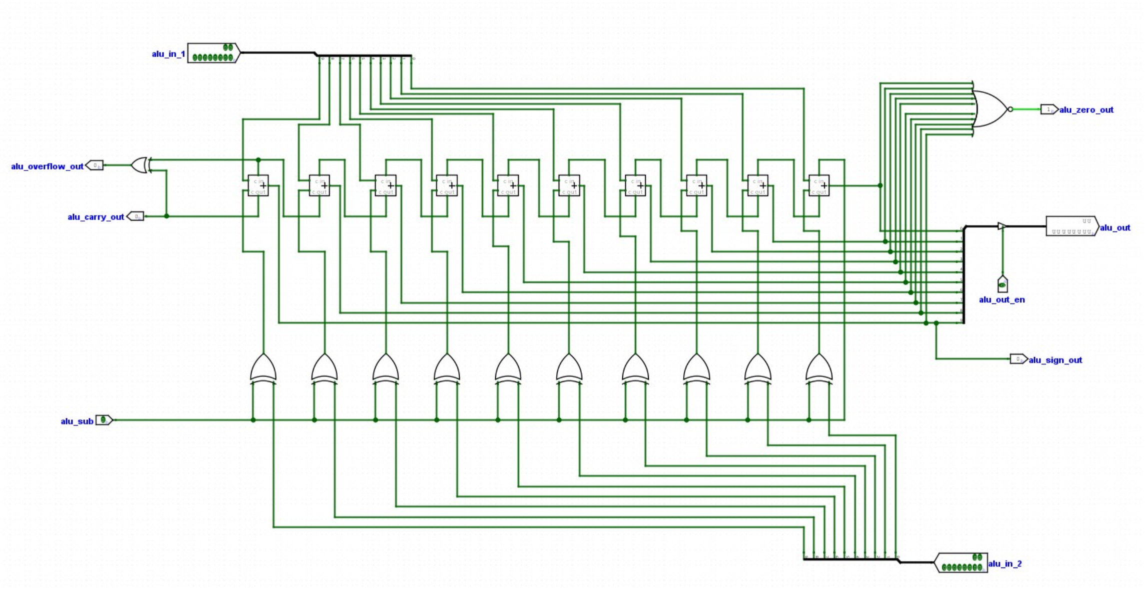


Figure 3: Arithmetic Logic Unit (ALU)

Description: The ALU is responsible for performing arithmetic and logical operations on data. In this SAP-1 design, the ALU supports basic operations such as addition and subtraction, which are essential for executing instructions like ADD and SUB. The ALU is implemented using combinational logic circuits that take two 10-bit inputs (from the accumulator and another register) and produce a 10-bit output along with status flags (Sign, Overflow, Zero, Carry).

Working Principle: The ALU receives two operands via the 10-bit data bus. Control signals determine the specific operation to be performed. For instance, when the **Alu_sub** signal is active, the ALU performs subtraction; otherwise, it defaults to addition. The result of the operation is then made available on the bus when the **Alu_out** signal is asserted. Additionally, the ALU updates status flags based on the result, which can be used for conditional branching in instructions like JNZ and JNC.

3.2.3 Instruction Register

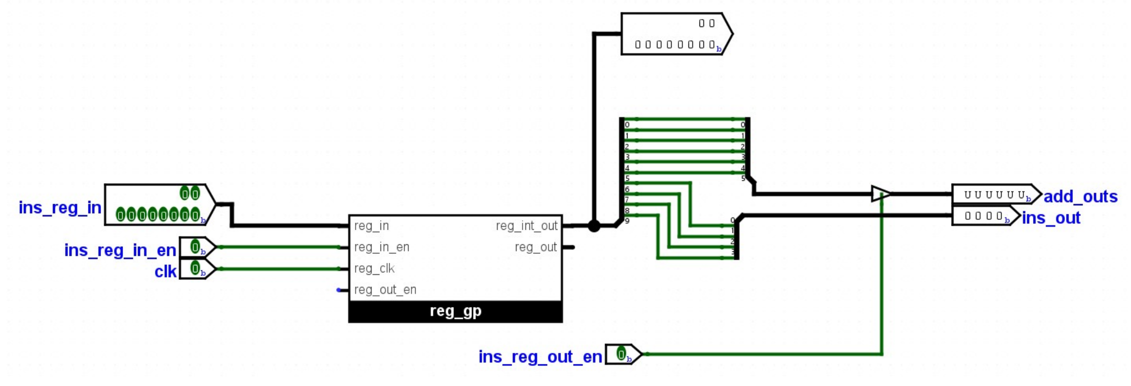


Figure 4: Instruction Register

Description: The Instruction Register (IR) holds the currently executing instruction fetched from memory. It is a 10-bit register that stores both the opcode (first 4 bits) and the operand (last 6 bits) of the instruction. The IR is crucial for decoding and executing instructions, as it provides the necessary information to the control sequencer for generating appropriate control signals.

Working Principle: When the **Ins_in** signal is activated, the instruction fetched from SRAM is loaded into the IR on the rising edge of the **clk** signal. The opcode and operand can then be accessed separately; the opcode is used by the control sequencer to determine which micro-operations to perform, while the operand is typically used as an address or immediate value for data manipulation. The contents of the IR can be output to the bus when the **Ins_out** signal is asserted, allowing other components to access the instruction details.

3.2.4 Program Counter

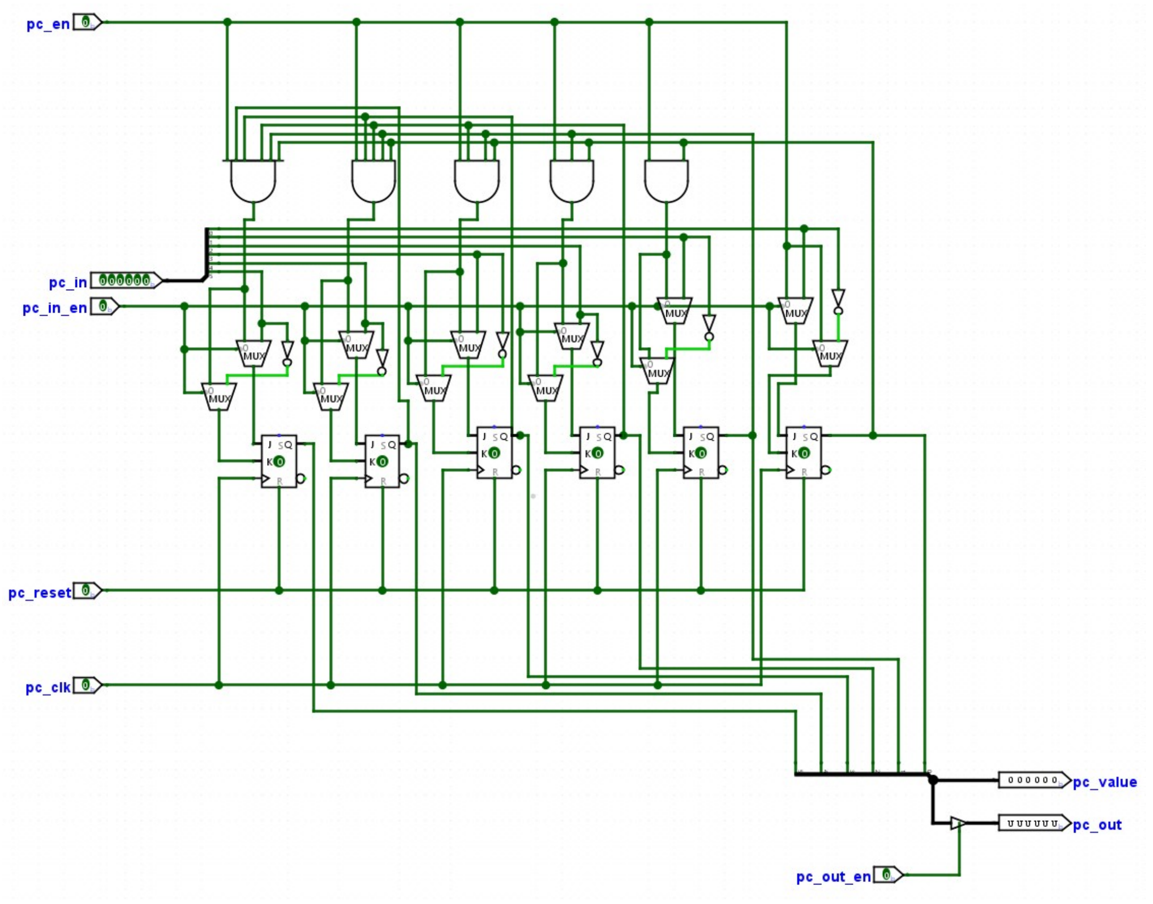


Figure 5: Program Counter

Description: The Program Counter (PC) is a crucial component that keeps track of the address of the next instruction to be executed. The PC increments automatically after each instruction fetch, ensuring sequential execution of instructions unless modified by jump or conditional jump instructions. In which case, the PC is loaded with a new address from the instruction register.

Working Principle: The PC is implemented using a series of flip-flops that store the current address. When the **Pc_en** signal is activated, the PC increments its value by one on the rising edge of the **clk** signal. If a jump instruction is executed, the **Pc_in** signal allows loading a new address directly from the instruction register into the PC. The current value of the PC is output to the memory address register (MAR) when the **Pc_out** signal is asserted, facilitating instruction fetch operations.

,

3.2.5 6-to-64 Decoder

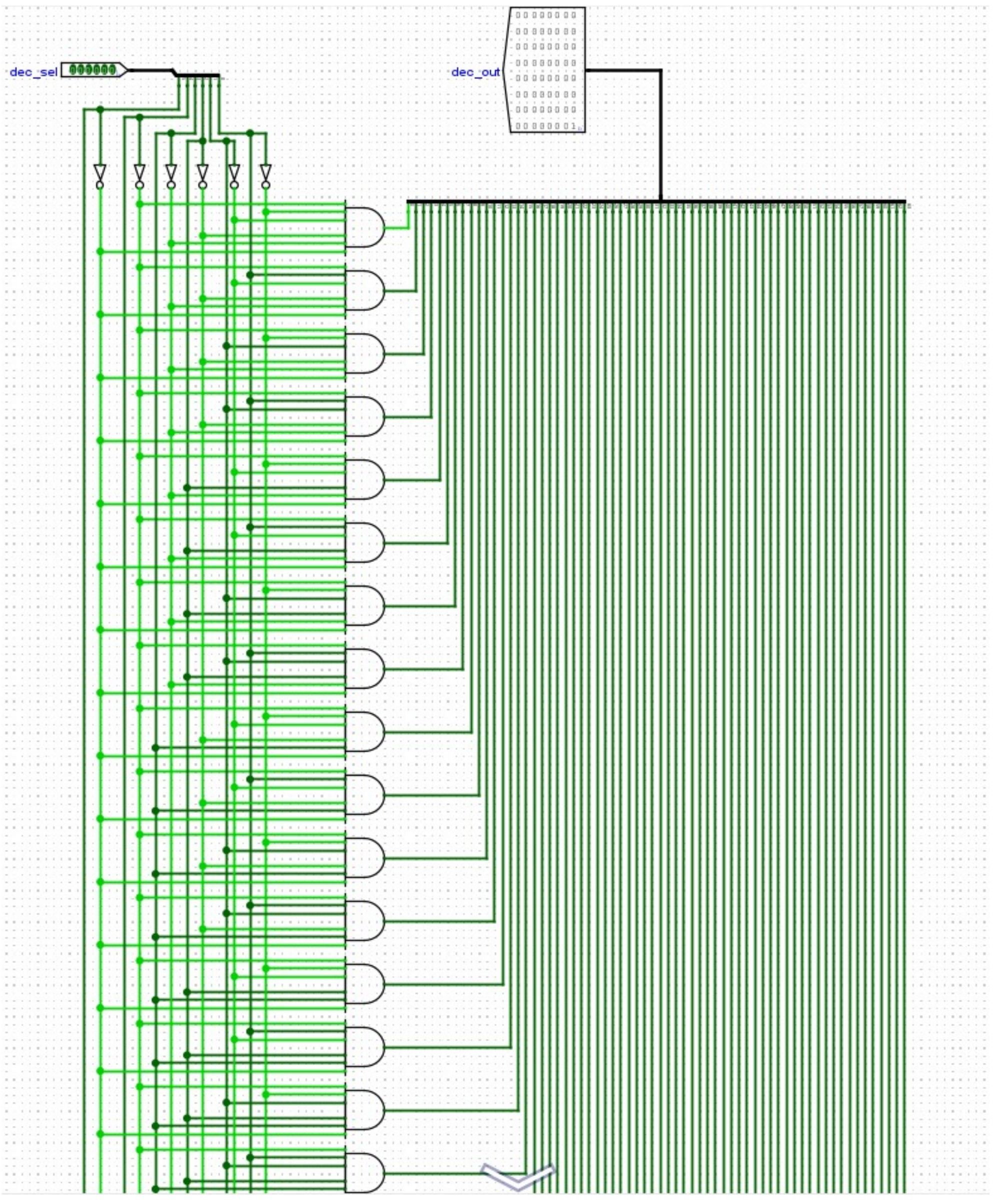


Figure 6: 6-to-64 Decoder

Description: The decoder is used for converting a 6-bit binary number to a one-hot 64-line output. This is also used in sram addressing where the 6-bit address from the instruction register is decoded to select one of the 64 memory locations. The decoder is implemented using combinational logic gates that ensure only one output line is active at any given time based on the binary input.

Working Principle: The decoder takes a 6-bit input and activates one of its 64 output lines corresponding to the binary value of the input. For example, if the input is 000011 (which is 3 in decimal), only the fourth output line (Y3) will be high, while all others remain low. This one-hot encoding is essential for selecting specific memory locations or control signals in the microprocessor architecture.

3.2.6 Memory Section (SRAM)

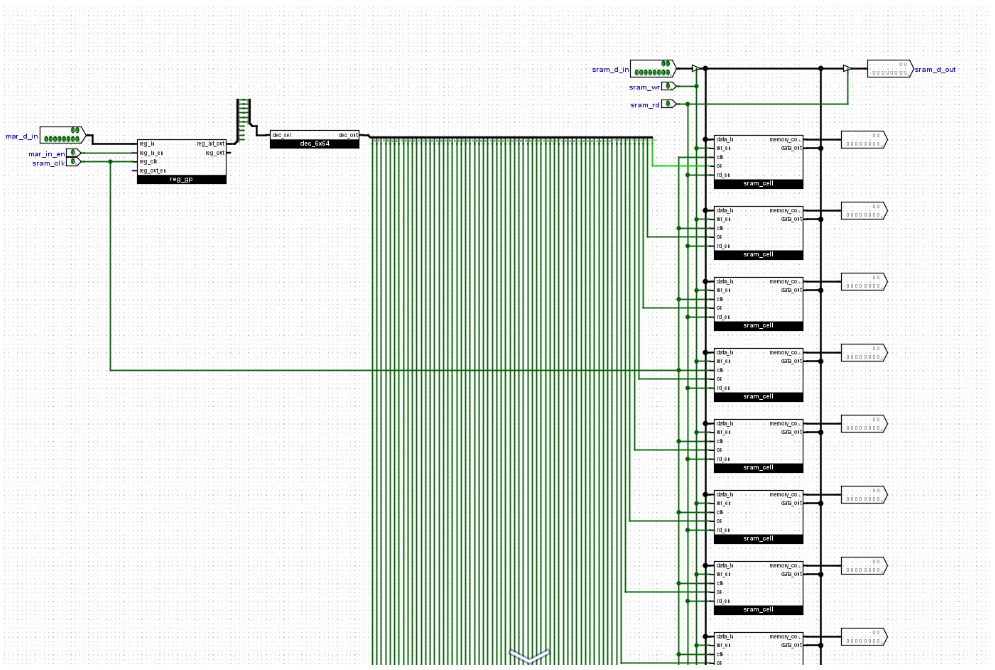


Figure 7: SRAM Memory Block

Description: age for the SAP-1 microprocessor, holding both instructions and data. The memory is organized into 64 words of 10 bits each, allowing the storage of machine code generated by the assembler. The SRAM is implemented using memory cells consist of general purpose registers that can be read from or written to based on control signals. The SRAM (Static Random-Access Memory) serves as the primary memory stor

Working Principle: Sram operates based on address, data, and control signals. The 6-bit address input goes into the Memory Address Register (MAR) which then select among the 64 memory cell using the 6-to-64 decoder. When the **Sram_rd** signal is asserted, the data from the selected memory location is placed on the 10-bit data bus. Conversely, when the **Sram_wr** signal is activated, the data present on the bus is written into the selected memory location. The memory can be initialized using a RAM Auto-Loader that loads machine code directly into SRAM via a debug protocol.

3.2.7 Ring Counter

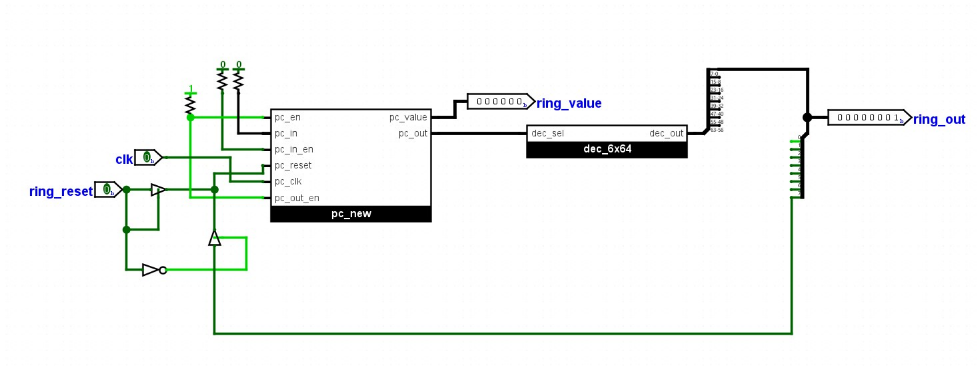
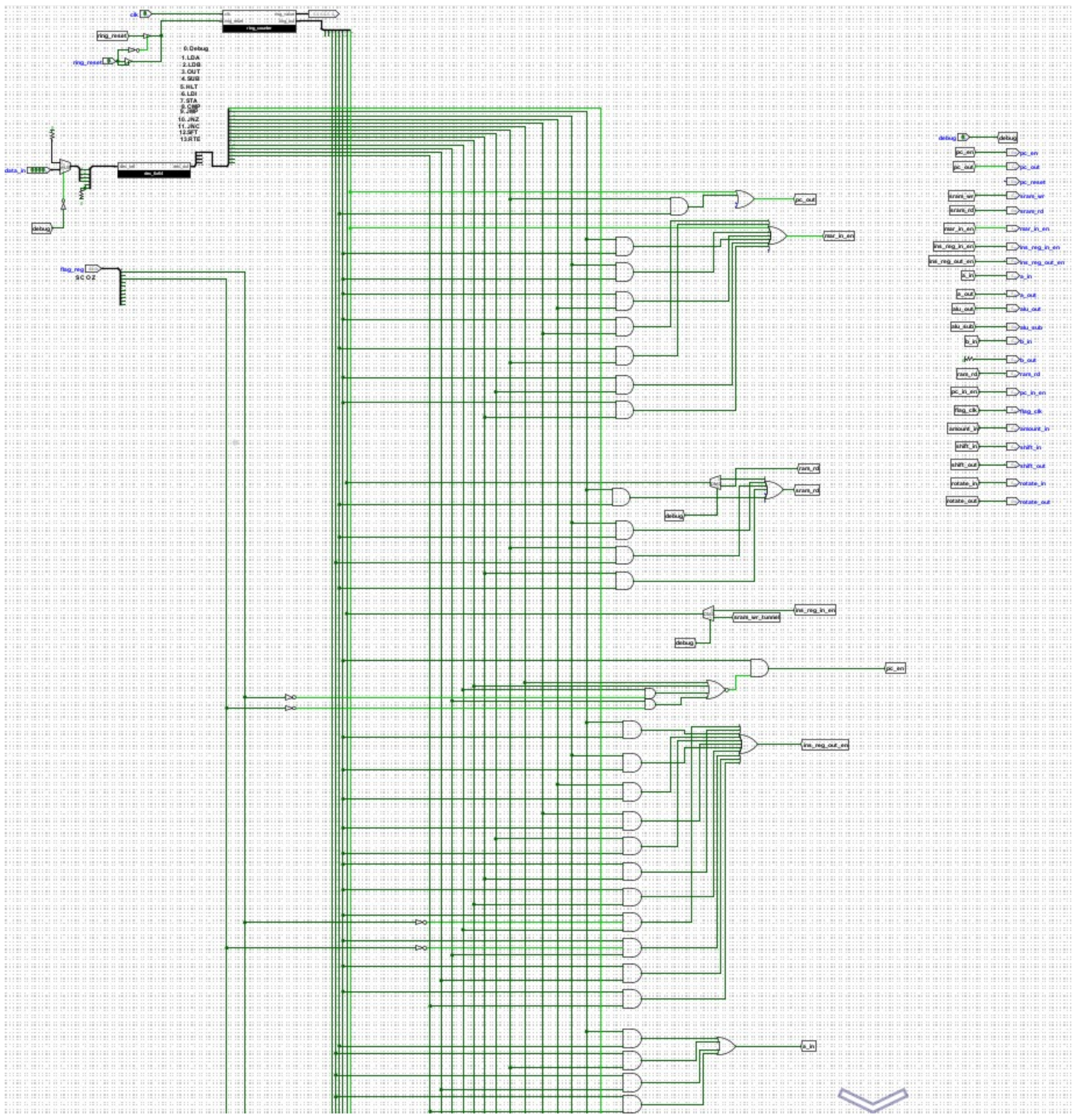


Figure 8: Ring Counter

Description: The ring counter generates a sequence of timing signals (T0 to T5) that orchestrate the fetch-decode-execute cycle of the microprocessor. It consists of program counter circuitry that cycles through six distinct states, each corresponding to a specific phase in the instruction execution process.

Working Principle: The ring counter is implementer using program counter circuitry that advances its state with each clock pulse. It produces six timing signals (T0 to T5) in a cyclic manner, where only one signal is active at any given time. The **ring_reset** signal can be used to reset the counter back to the initial state (T0). These timing signals are crucial for synchronizing the operations of control sequencer, ensuring that data is fetched, decoded, and executed in the correct sequence.

3.2.8 Control Sequencer



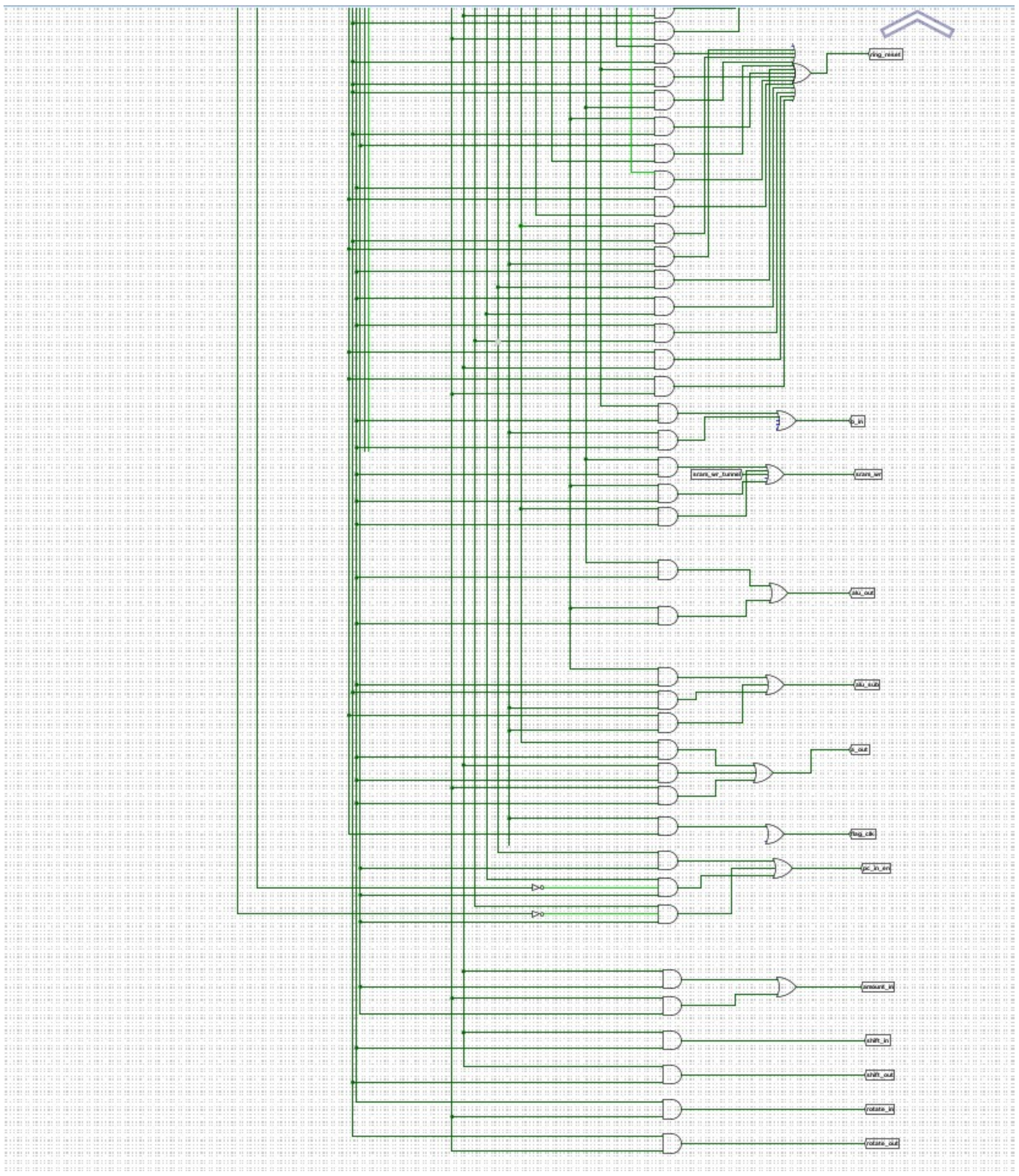


Figure 9: Control Sequencer

Description: The control sequencer is the central coordination unit that manages the timing and execution of instructions in the SAP-1 microprocessor. It orchestrates the flow of data and control signals by interpreting the current instruction and timing state, ensuring that each module (registers, ALU, memory, etc.) performs its required operation at the correct moment. The sequencer is responsible for generating all necessary control signals for data movement, arithmetic/logic operations, and conditional branching. It acts as the "brain" of the processor, translating high-level instructions into a series of micro-operations that are executed in a well-defined sequence. The control sequencer is implemented using a combination of one-hot opcode decoding, timing pulse generation, and logic circuits that combine these signals to drive the appropriate control lines.

Working Principle: The control sequencer receives timing pulses (T0 - T5) from the ring counter, which indicate the current phase of the instruction cycle. It decodes the opcode from the instruction register using a one-hot decoder, activating a unique line for each instruction type. Combinational logic (AND, OR, NOT gates) combines the decoded instruction and timing signals to assert the appropriate control pins for each module, such as enabling data input/output on the bus, triggering ALU operations, or writing to memory. For conditional instructions like JNZ and JNC, the sequencer also monitors status flags (Zero, Carry) to determine whether to branch or continue sequential execution. This design ensures precise synchronization and correct sequencing of micro-operations throughout the processor.

3.2.9 Barrel Shifter

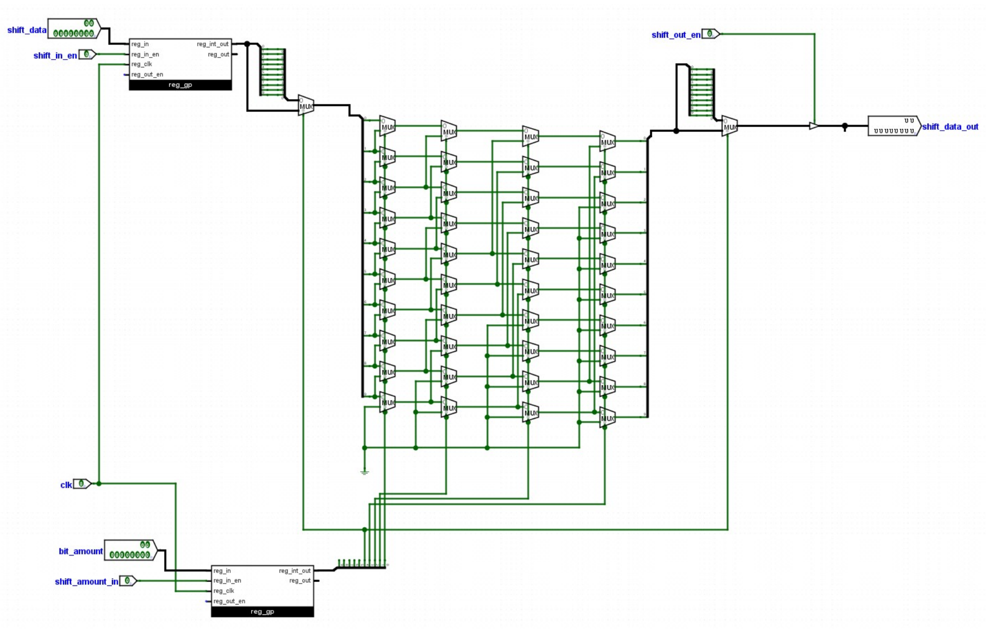


Figure 10: Barrel Shifter

Description: Purpose of the barrel shifter is for executing SFT instruction (logical shift). It can shift upto 16 bits left or right depending on the operand bits in the instruction.

Working Principle: Barrel shifter allows shifting the bits of the accumulator left or right by a specified number of positions. The shift amount and direction are determined by the operand bits in the instruction. When the **Shift_in_en** signal is activated, the current value of the accumulator is output to the barrel shifter. Depending on the direction bit (D) and the number of positions (N[3:0]), the shifter performs a logical shift operation. The shifted result is then loaded back into the data bus when the **Shift_out_en** signal is asserted. This operation effectively moves bits to the left or right, filling vacant positions with zeros. **amount_in** signal is used to load the shift amount from the instruction register to the barrel shifter. This operation is implemented using multiplexers that select the appropriate bits based on the shift amount bits (1,2,4 or 8 bit shifting respectively of the amount bits from LSB to MSB).

3.2.10 Ring Rotator

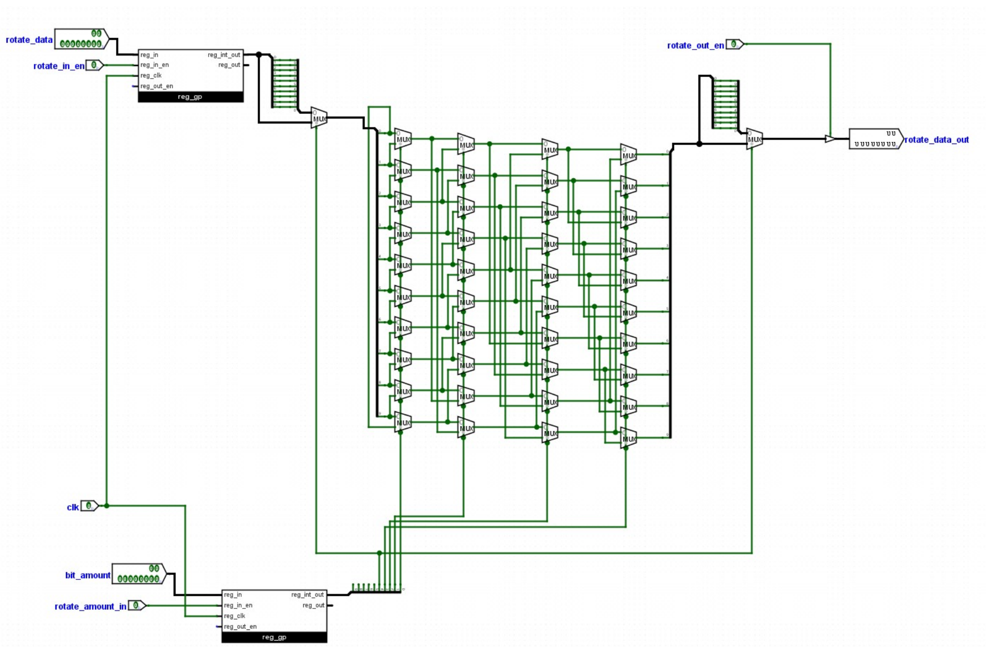


Figure 11: Ring Rotator

Description: The ring rotator is designed to perform bitwise rotation operations, supporting the RTE instruction. It can rotate bits left or right by a specified number of positions, effectively wrapping around the bits that overflow from one end to the other. It can rotate upto 16 bits left or right depending on the operand bits in the instruction.
Working Principle: The ring rotator takes the current value of the accumulator and

rotates its bits based on the direction and amount specified in the instruction. When the **rotate_in_en** signal is activated, the accumulator's value is sent to the rotator. The direction bit (D) determines whether the rotation is to the left or right, while the number of positions (N[3:0]) specifies how many bits to rotate. The rotated result is then placed back onto the data bus when the **Rotate_out_en** signal is asserted. This operation is implemented using multiplexers that select bits based on the rotation amount bits (1,2,4 or 8 bit rotating respectively of the amount bits from LSB to MSB). **amount_in** signal is used to load the rotation amount from the instruction register to the ring rotator.

3.2.11 RAM Auto-Loader

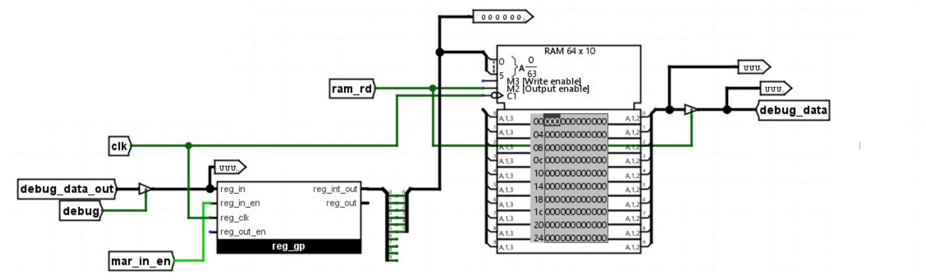


Figure 12: RAM Auto-Loader

Description: The RAM Auto-Loader is a specialized module designed to facilitate the automatic loading of machine code into the SRAM memory of the SAP-1 microprocessor. It operates via a debug protocol, allowing seamless initialization of the memory without manual intervention. The auto-loader is essential for preparing the microprocessor to execute programs by populating the SRAM with the necessary instructions and data.

Working Principle: The RAM Auto-Loader is implemented using Logisim's inbuilt RAM component, which allows direct loading of machine code files from the file manager in v3.0 hexadecimal format. During the auto-loading process, the **Sram_wr** and **ram_rd** control pins are used to write data onto the SRAM. The **debug** pin must be enabled to initiate and synchronize this loading operation. Once the code file is loaded, the microprocessor can execute the program stored in SRAM automatically, without manual intervention.

3.3 Datapath

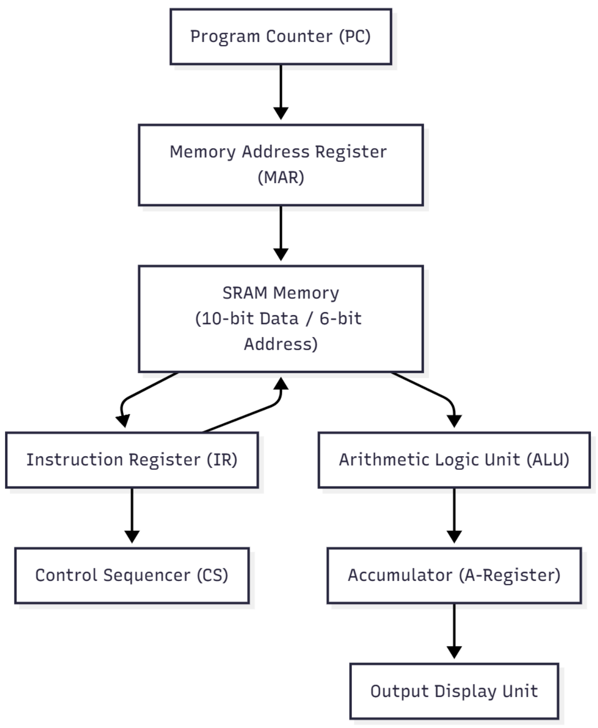

Program Counter (PC)
Memory Address Register (MAR)
SRAM Memory (10-bit Data / 6-bit Address)
Instruction Register (IR)
Arithmetic Logic Unit (ALU)
Control Sequencer (CS)
Accumulator (A-Register)
Output Display Unit

Figure 2: Datapath of the SAP-1 Microprocessor

3.4 Control Path / Sequencer

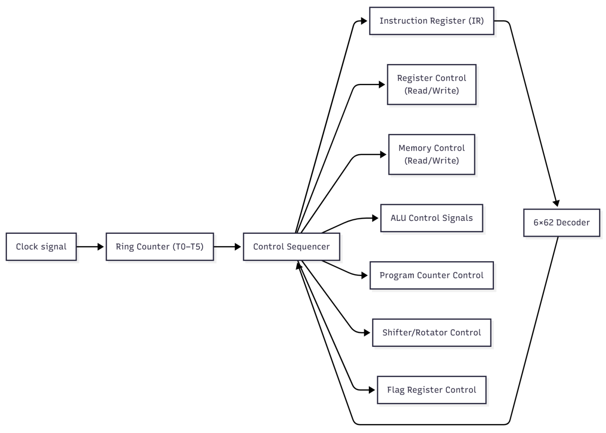


Figure 3: Control Path / Sequencer of the SAP-1 Microprocessor

3.5 Instruction Set and Encoding

Mnemonic	**Opcode (4b)**	**Operand (6b)**	**Notes**
LDA	0001	Address	Load accumulator with value from memory
LDB	0010	Address	Load B-register with value from memory
OUT	0011	Output Address	Send accumulator data to output register
SUB	0100	Output Address	Subtract memory data from accumulator and send to output register
HLT	0101	—	Halt processor operation
LDI	0110	—	Next word is immediate 10-bit literal
STA	0111	Address	Store accumulator value into memory
CMP	1000	Address	Compare accumulator with memory value and update status flags
JMP	1001	Address	Unconditional jump to specified memory location
JNZ	1010	Address	Jump if zero flag $\neq 0$ (not zero)
JNC	1011	Address	Jump if carry flag $\neq 0$ (no carry)
SFT	1100	`X D N3 N2 N1 N0`	Logical shift of accumulator by `N[3:0]` bits; `D` = 0 for left, 1 for right; `X` = don't care
RTE	1101	`X D N3 N2 N1 N0`	Bitwise rotation of accumulator by `N[3:0]` bits; `D` = 0 for left, 1 for right; `X` = don't care

Table 2: Instruction set and encoding of the SAP-1 Microprocessor

3.6 Timing and Micro-Operation Sequence

The execution of each instruction in the designed SAP-1 microprocessor is governed by a fixed six-phase timing cycle generated by the **ring counter**. Each clock pulse advances the system through a specific time state (T0–T5), during which distinct micro-operations are performed. The **Control Sequencer** uses these timing pulses in combination with the decoded opcode to assert the required control lines for registers, memory, and ALU modules.

Instruction	**T0**	**T1**	**T2**	**T3**	**T4**	**T5**
Debug	Pc_out, Mar_in	Sram_wr, ram_rd	Pc_en	ring_reset		
LDA	Pc_out, Mar_in	Sram_rd, Ins_in	Pc_en, Ins_out, Mar_in	Sram_rd, a_in	ring_reset	
LDB	Pc_out, Mar_in	Sram_rd, Ins_in	Pc_en, Ins_out, Mar_in	Sram_rd, B_in	ring_reset	
OUT	Pc_out, Mar_in	Sram_rd, Ins_in	Pc_en, Ins_out, Mar_in	Sram_wr, Alu_out	ring_reset	
SUB	Pc_out, Mar_in	Sram_rd, Ins_in	Pc_en, Ins_out, Mar_in	Sram_wr, Alu_out, Alu_sub	ring_reset	
CMP	Pc_out, Mar_in	Sram_rd, Ins_in	Pc_en, Ins_out, Mar_in	Sram_rd, B_in	Alu_sub	Flag_clk, ring_reset
HLT	Pc_out, Mar_in	Sram_rd, Ins_in	ring_reset			
LDI	Pc_out, Mar_in	Sram_rd, Ins_in	Pc_en	Pc_out, Mar_in	Ram_rd, a_in	ring_reset

Instruction	**T0**	**T1**	**T2**	**T3**	**T4**	**T5**
STA	Pc_out, Mar_in	Sram_rd, Ins_in	Pc_en, Ins_out, Mar_in	Ram_wr, a_out	ring_reset	
JMP	Pc_out, Mar_in	Sram_rd, Ins_in	Ins_out, Pc_in	ring_reset		
JNZ	Pc_out, Mar_in	Sram_rd, Ins_in	If Z=0, Ins_out, Pc_in	ring_reset		
			If Z=1, Pc_en			
JNC	Pc_out, Mar_in	Sram_rd, Ins_in	If C=0, Ins_out, Pc_in	ring_reset		
			If C=1, Pc_en			
SFT	Pc_out, Mar_in	Sram_rd, Ins_in	Pc_en, Ins_out, amount_in	a_out, Shift_in_en	A_in, Shift_out_en	ring_reset
RTE	Pc_out, Mar_in	Sram_rd, Ins_in	Pc_en, Ins_out, amount_in	a_out, rotate_in_en	A_in, rotate_out_en	ring_reset

Table 3: Timing and micro-operation sequence for each instruction

Each instruction follows the same basic fetch–decode–execute structure:

- **T0–T1 (Fetch):** Program Counter sends the address to MAR; instruction is fetched from SRAM and loaded into the Instruction Register.
- **T2 (Decode):** The instruction is decoded and the corresponding control signals are generated.
- **T2–T5 (Execute):** Depending on the instruction type, data transfer, arithmetic/logic,

or conditional jump operations are performed. The sequence then resets the ring counter for the next instruction.

4. Unique Feature Implementation

In addition to the standard SAP-1 architecture, several key design enhancements were implemented to extend functionality, improve programmability, and support advanced instruction execution. The major modifications introduced in this project are summarized below:

1. **Conditional Flag System:** A dedicated *flag register* was introduced to hold the status of key condition signals such as `Zero (Z)` and `Carry (C)`. These flags are automatically updated after arithmetic or logical operations and are utilized by conditional branch instructions such as `JNZ` (Jump if Not Zero) and `JNC` (Jump if No Carry). This enables decision-based program flow, a feature not present in the original SAP-1 design.

2. **Shift and Rotate Functional Units:** Two new dedicated modules — a **barrel shifter** and a **ring rotator** — were incorporated into the ALU subsystem. These execute the newly added `SFT` (shift) and `RTE` (rotate) instructions. Each operand encodes the direction (left/right) and the shift or rotation magnitude, enabling up to 15-bit logical shifts or cyclic rotations within a single instruction cycle.

3. **Program Counter Bus Input (for JMP Instructions):** The Program Counter (PC) module was modified to allow data input directly from the system bus. This enhancement enables the `JMP` instruction to load target addresses dynamically from memory through the 10-bit bus, allowing flexible branching and subroutine jumps.

4. **Expanded Bus and Memory Architecture:** The original 8-bit system bus (4-bit opcode + 4-bit operand) was redesigned into a **10-bit bus** (4-bit opcode + 6-bit operand). This expansion allows addressing of up to 64 unique memory locations instead of 16, significantly increasing program length capacity. The SRAM block was accordingly upgraded from 16 cells to 64 cells (6-bit address width, 10-bit data width). All associated modules, including the instruction register, memory address

register, and control sequencer, were updated to support the new bus width and instruction encoding format.

Collectively, these modifications extend the SAP-1 architecture beyond its educational baseline, transforming it into a programmable and conditionally driven 10-bit microprocessor system with enhanced logical, branching, and data manipulation capabilities.

5. Software Component – Assembler/Compiler

5.1 Overview

The software component of this project consists of a custom **assembler** developed in Python to translate assembly-level mnemonics into 10-bit machine code instructions compatible with the SAP-1 microprocessor. Each instruction consists of a 4-bit opcode and a 6-bit operand field, making the total instruction width 10 bits.

The assembler introduces a **suffix-based numeric system** to ensure clarity and prevent ambiguity in data representation:

- **b** – binary (e.g., `1011b`)
- **d** – decimal (e.g., `11d`)
- **h** – hexadecimal (e.g., `0Ah`)

All numeric literals and operands must include the suffix. Missing suffixes trigger error messages, maintaining strong syntax integrity.

The assembler outputs data in the format:

```
v3.0 hex words addressed
```

which is directly compatible with the RAM Auto-Loader for the SAP-1 system.

5.2 Design Flow

The assembler reads the source assembly file line by line, identifies instruction types, validates operands with suffix enforcement, and encodes them into their respective 10-bit machine representations.

The high-level logic is summarized in the flowchart shown below.

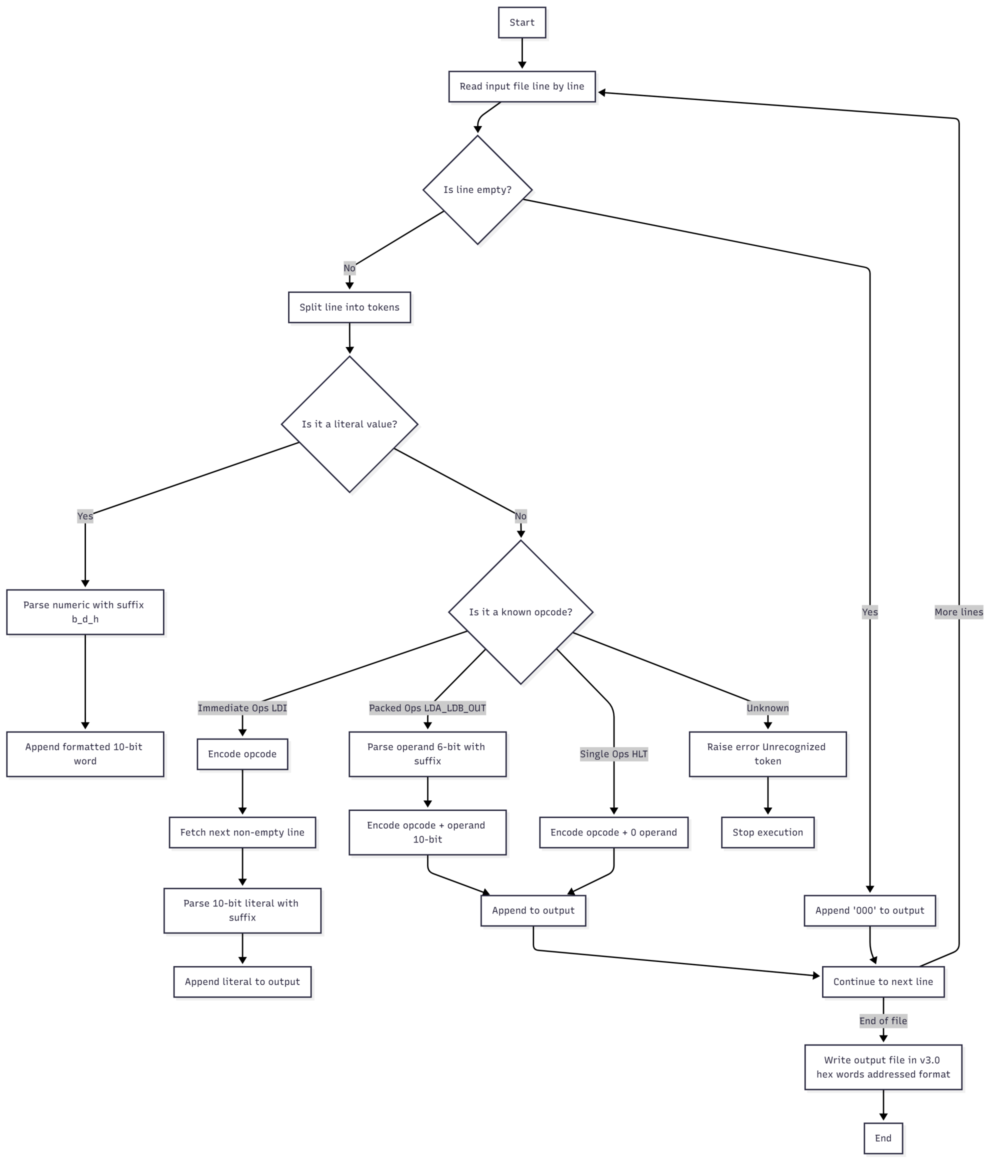

Start
Read input file line by line
Is line empty?
No
Split line into tokens
Is it a literal value?
Yes
No
Parse numeric with suffix b_d_h
Is it a known opcode?
Yes
More lines
Append formatted 10-bit word
Immediate Ops LDI
Packed Ops LDA_LDB_OUT
Unknown
Encode opcode
Parse operand 6-bit with suffix
Single Ops HLT
Raise error Unrecognized token
Fetch next non-empty line
Encode opcode + operand 10-bit
Encode opcode + 0 operand
Stop execution
Parse 10-bit literal with suffix
Append to output
Append '000' to output
Append literal to output
Continue to next line
End of file
Write output file in v3.0 hex words addressed format
End

Figure 9: Flowchart of the Assembler

The major steps of operation are:

1. **Tokenization:** Each line is separated into instruction and operand tokens.
2. **Opcode Mapping:** Mnemonics are mapped to their defined 4-bit opcodes.
3. **Operand Validation:** Operands are validated for suffix, range, and bit-width.
4. **Encoding:** 10-bit instruction words are generated by merging opcode and operand fields.
5. **Output Generation:** Encoded data is written in "v3.0 hex words addressed" format for direct memory loading.

5.3 Assembler Source Code

The full Python implementation of the assembler is provided below. It incorporates the 10-bit instruction format, suffix-based validation, and automatic blank-line handling.

```
import re


# --- Opcode nibbles (4-bit opcodes) ---
OP_NIB = {
    "LDA": 0x1, "LDB": 0x2, "OUT": 0x3, "SUB": 0x4,
    "HLT": 0x5,  # single-op
    "LDI": 0x6,  # loads 10-bit literal from next line
    "STA": 0x7, "CMP": 0x8, "JMP": 0x9,
    "JNZ": 0xA, "JNC": 0xB, "SFT": 0xC, "RTE": 0xD,
}


PACKED_OPS = {"LDA", "LDB", "OUT", "SUB", "STA", "CMP",
              "JMP", "JNZ", "JNC", "SFT", "RTE"}
SINGLE_OPS = {"HLT"}
IMMEDIATE_OPS = {"LDI"}


# ---------------------------------------------------------------
# Numeric parser with suffix enforcement
# ---------------------------------------------------------------


def parse_with_suffix(token: str, bit_limit: int, label: str) -> int:
    """Parse number with suffix b/d/h, validate within bit_limit."""
    tok = token.strip().lower()


    # Validate suffix
    if not re.fullmatch(r"[0-9a-f]+[bdh]", tok):
        raise ValueError(
            f"Invalid {label} '{token}': must end with b, d, or h "
            f"(binary/decimal/hex). Example: 1011b, 11d, 0ah"
        )
    val_str, suffix = tok[:-1], tok[-1]

    try:
        if suffix == "b":
            val = int(val_str, 2)
        elif suffix == "d":
            val = int(val_str, 10)
        elif suffix == "h":
            val = int(val_str, 16)
    except ValueError:
        raise ValueError(f"Bad numeric format in '{token}'")

    # Range check
    if not (0 <= val < (1 << bit_limit)):
        raise ValueError(
            f"{label} '{token}' out of range for {bit_limit}-bit value (0..{(1<<bit_limit)-
        )

    return val

# ---------------------------------------------------------------
# Core encoding helpers
# ---------------------------------------------------------------

def encode_word(op_nib: int, operand6: int) -> str:
    """Encode opcode nibble + 6-bit operand into 3-hex-digit (10-bit) word."""
    if not (0 <= op_nib <= 0xF):
        raise ValueError(f"Opcode nibble out of range: {op_nib}")
    if not (0 <= operand6 <= 0x3F):
        raise ValueError(f"Operand out of range (0..63): {operand6}")
    val10 = (op_nib << 6) | operand6
    return f"{val10:03x}"
```

```
# ------------------------------------------------------------

input_file = "input.txt"
output_file = "output"

with open(input_file, "r", encoding="utf-8") as f:
    raw_lines = f.readlines()

bytes_out = []
i = 0

def get_next_nonempty_line():
    """Get next non-empty token, injecting 000 for all blank lines."""
    global i
    while i < len(raw_lines):
        nxt = raw_lines[i].strip()
        i += 1
        if nxt == "":
            bytes_out.append("000")
            continue
        return nxt
    raise ValueError("Missing literal after LDI")

# ------------------------------------------------------------
# Assembler loop
# ------------------------------------------------------------

while i < len(raw_lines):
    line = raw_lines[i].strip()
    i += 1

    # Blank line → insert 000
    if not line:
        bytes_out.append("000")
    continue

parts = line.split()
up0 = parts[0].upper()

# --- Literal 10-bit data word ---
if len(parts) == 1 and up0 not in OP_NIB:
    try:
        val10 = parse_with_suffix(parts[0], 10, "literal value")
        bytes_out.append(f"{val10:03x}")
        continue
    except ValueError as e:
        raise ValueError(f"Line {i}: {e}")

# --- Packed ops ---
if up0 in PACKED_OPS:
    if len(parts) < 2:
        raise ValueError(f"Missing operand for {up0}")

    operand_token = parts[1].strip().lower()

    # Special rule for SFT / RTE → still 6-bit operand with suffix
    operand_val = parse_with_suffix(operand_token, 6, f"{up0} operand")

    bytes_out.append(encode_word(OP_NIB[up0], operand_val))
    continue

# --- Single ops ---
if up0 in SINGLE_OPS:
    bytes_out.append(encode_word(OP_NIB[up0], 0))
    continue

# --- LDI (next line is a 10-bit literal) ---
if up0 in IMMEDIATE_OPS:
        bytes_out.append(encode_word(OP_NIB[up0], 0))  # operand unused
        literal_line = get_next_nonempty_line()
        lit_val = parse_with_suffix(literal_line, 10, "LDI literal")
        bytes_out.append(f"{lit_val:03x}")
        continue

    raise ValueError(f"Unrecognized opcode or token: '{line}'")

# ------------------------------------------------------------
# Output file
# ------------------------------------------------------------
out_lines = ["v3.0 hex words addressed", f"0: {' '.join(bytes_out)}"]
with open(output_file, "w", encoding="utf-8") as f:
    f.write("\n".join(out_lines))
```

5.4 Sample Input & Output

5.4.1 Example Program 1:

This code can be used to compare two numbers. The RGB LED connected to the accumulator will light up red if the first number is greater, blue if the second number is greater, and green if they are equal. This code demonstrates conditional branching and flag usage. **Input Assembly Code:**

```
LDA 13d
CMP 14d
JNZ 6d
LDI
001h
HLT
JNC ah
LDI
002h
```

```
HLT
LDI
004h
HLT
055h
056h
```

Output Machine Code (output.txt):

```
v3.0 hex words addressed
0: 04d 20e 286 180 001 140 2ca 180 002 140 180 004 140 055 056
```

5.4.2 Example Program 2:

This code is to compare two numbers. Then make the second number as equal to the first number by adding or subtracting 1 from it. This demonstrates loop and conditional branching capabilities. **Input Assembly Code:**

```
LDA 15d
CMP 14d
JNZ 4d
HLT
JNC 9d
LDA 14d
LDB 13d
OUT 14d
JMP 0d
LDA 14d
LDB 13d
SUB 14d
JMP 0d
001h
056h
055h
```

Output Machine Code (output.txt):

```
v3.0 hex words addressed
0: 04f 20e 284 140 2c9 04e 08d 0ce 240 04e 08d 10e 240 001 056 055
```

5.4.3 Example Program 3:

This code demonstrates the use of the shift and rotate instructions. It loads a value into the accumulator, performs 1 bit left shift, then a right 2 bit rotation, and store the results in each steps in the memory. This showcases bit manipulation capabilities. **Input Assembly Code:**

```
LDA 9d
SFT 000001b
STA 8d
SFT 000001b
STA 8d
RTE 010010b
STA 8d
JMP 1d


1h
```

Output Machine Code (output.txt):

```
v3.0 hex words addressed
0: 049 301 1c8 301 1c8 352 1c8 241 000 001
```

5.5 Error Handling

The assembler implements strong syntax and range checking to detect and prevent invalid instruction encoding.

Common handled cases include:

- **Missing suffix:** Raises an error indicating that numeric values must end with `b`, `d`, or `h`.

- **Invalid binary or format:** Detects malformed binary/hex/decimal numbers and halts parsing.
- **Out-of-range operands:** Ensures all operands fit within their defined bit-width (6-bit for operands, 10-bit for literals).

6. Challenges and Debugging

Throughout the design and implementation of the SAP-1 microprocessor system, several technical challenges were encountered and systematically addressed. This section outlines the major issues faced and the corresponding solutions adopted during the development process.

6.1 CMP Instruction and Flag Handling

A significant challenge arose in the implementation of the `CMP` (compare) instruction. Initially, the status flag bits (Zero and Carry) were not reliably stored after comparison operations, leading to incorrect conditional branching behavior. To resolve this, a dedicated **flag register** was introduced, along with a **flag_clk** signal to synchronize flag updates. This ensured that the status flags were correctly latched and available for subsequent instructions, thereby enabling robust conditional execution.

6.2 Program Counter Reliability

The original program counter design utilized a ripple-carry counter constructed from flip-flops. This approach exhibited unreliable behavior due to propagation delays and asynchronous updates, which occasionally resulted in incorrect instruction sequencing. The solution involved redesigning the program counter to generate its output purely through combinational logic based on the current input signals. This modification eliminated timing inconsistencies and improved the overall reliability of instruction flow within the processor.

6.3 Shifter and Rotator Output Stability

During the integration of the barrel shifter and ring rotator modules, it was observed that their outputs needed to be held stable for at least one clock cycle to ensure correct data transfer onto the system bus. Without this, the shifted or rotated values were not consistently available for downstream operations. To address this, a register was added to each module to temporarily store the output value, guaranteeing that the result could be reliably accessed in the subsequent state.

6.4 Assembler Robustness

Developing the Python-based assembler presented its own set of challenges, particularly in enforcing strict syntax and error handling. The assembler was designed to be "idiot-proof," meaning it rigorously checks for missing numeric suffixes, invalid formats, and out-of-range values. Comprehensive validation logic was implemented to provide clear error messages and prevent the generation of malformed machine code, thereby enhancing the usability and reliability of the software component.

These targeted debugging efforts and architectural refinements collectively contributed to the successful realization of a stable, extensible SAP-1 microprocessor system.

7. Demonstration and Repository Links

7.1 Demonstration Video

A demonstration video showcasing the functionality of the designed SAP-1 microprocessor system is available on YouTube.

- **YouTube Link:** https://youtu.be/mreEUxgNvG8

7.2 Project Repository

All project files, including the Logisim design, assembler source code, and documentation, are available on GitHub. Anyone can use the files to rebuild and test the project.

- **GitHub Link:** https://github.com/Tahmid-fuad/Control_sequencer

The repository also contains the full report and sample programs for easy verification.

8. Conclusion

This project successfully demonstrates the complete design and implementation of an enhanced SAP-1 microprocessor system, integrating both hardware and software components into a unified, automated framework. Through modular circuit design in Logisim and the development of a robust Python-based assembler, the system achieves seamless program compilation, memory initialization, and autonomous execution. Key architectural improvements—including expanded bus width, conditional flag support, and advanced bit manipulation units—extend the capabilities of the classical SAP-1, enabling more complex and flexible program control.

The challenges encountered during development, such as reliable flag handling, program counter stability, and robust error checking in the assembler, were systematically addressed through targeted design refinements. The result is a stable, extensible microprocessor platform that not only illustrates fundamental principles of digital computer architecture but also provides a practical foundation for further exploration and experimentation in VLSI and embedded systems.

Overall, this work highlights the importance of a holistic approach to system design, where hardware and software co-development leads to greater functionality, reliability, and ease of use. The project serves as a valuable learning experience in digital logic, processor architecture, and toolchain integration, laying the groundwork for future advancements in custom microprocessor design.