Chittagong University of Engineering & Technology

Department of Electronics and Telecommunication Engineering

Project Report

Design and Implementation of a SAP-1 Architecture with Control Sequencer Using Logisim Evolution

Course No: ETE 404

Course Title : VLSI Technology Sessional

SUBMITTED BY:

Name : Afsana Dilruba
ID no : 2008014

SUBMITTED TO:

Course Teacher:
Arif Istiaque
Lecturer
Department of Electronics and Telecommunication Engineering, CUET

Contents

1 Introduction **2**

2 Objectives **2**

3 Key System Features **3**

4 Architectural Design and Component Analysis **4**

4.1 System Architecture Overview 4
4.2 Register Implementation (A and B) 5
4.3 Program Counter (PC) 6
4.4 Memory System and Address Register (MAR) 7
4.5 Instruction Register and Opcode Decoder 9
4.6 Arithmetic Logic Unit (ALU) 10
4.7 Timing Control Generator 11
4.8 Automatic Operation Control Logic 13
4.9 Manual/Loader Operation Control 15

5 Instruction Set Architecture **16**

5.1 Instruction Encoding Scheme 16
5.2 Assembler 17

6 Operation **18**

6.1 Fetch–Decode–Execute Cycle 19

7 Running the CPU in Automatic Mode (ADD + SHIFT Program) **19**

8 Conclusion **25**

1 Introduction

This report presents the design and implementation of an enhanced 8-bit SAP-1 (Simple-As-Possible) processor using Logisim Evolution. The classical SAP-1 architecture has been extended with advanced control sequencing, a broader instruction set, and both automatic and manual execution modes. The system has been implemented on a single-bus architecture, ensuring synchronized data transfer and strict single-driver bus discipline.

The extended instruction set includes `LDA, LDB, ADD, SHL A, SHR A` and `HLT`, supporting arithmetic, shift, and branching operations. Instruction execution follows a structured fetch–decode–execute cycle, governed by a six-phase ring counter and an opcode decoder. A manual (loader) mode has been incorporated to allow program transfer from ROM to RAM using handshake-based control signals.

All major components—Program Counter (PC), Arithmetic Logic Unit (ALU), Register Unit, Memory System, Instruction Register, and Control Sequencer—were designed and simulated to ensure proper timing and logical correctness. This implementation provides a valuable educational platform for understanding micro-operations, control signal generation, and instruction-level coordination in a simplified computing model.

2 Objectives

1. To design and implement an enhanced 8-bit SAP-1 architecture in Logisim Evolution with added shift and branch instructions for educational analysis.

2. To establish a single-bus system with an 8-bit data path and 4-bit address space, governed by hardwired control for precise timing.

3. To incorporate two operating modes:

   - **Automatic Mode:** Executes the fetch–decode–execute cycle using a six-phase ring counter.
   - **Manual/Loader Mode:** Enables secure ROM-to-RAM data transfer with diagnostic monitoring.

4. To construct a datapath including dual 8-bit registers, a ripple-carry ALU, Program Counter, MAR, SRAM, and an Instruction Register divided into opcode and operand fields.

5. To ensure proper synchronization and timing across all modules through the control sequencer.

6. To validate system performance through instruction execution and verification of correct data transfer and control signal behavior.

3 Key System Features

1. **Classical SAP-1 foundation (extended):** The processor has been developed on the fundamental SAP-1 framework and expanded with additional functionality. The architecture employs a single 8-bit data bus with a 4-bit address space (16 memory locations) and a hardwired control approach without the use of microcode. This maintains simplicity while providing educational clarity in data flow and instruction execution.

2. **Extended instruction set:** The system supports nine instructions: `LDA, LDB, ADD, SHL A, SHR A, SHL B, SHR B, JUMP,` and `HLT`. These instructions provide arithmetic, shift, and branching operations, extending beyond the classical SAP-1 feature set.

3. **Dual operating modes:** The processor operates in two distinct modes:

   - **Automatic Mode:** Executes programs through a standard fetch–decode–execute cycle driven by a six-phase ring counter (T1–T6).
   - **Manual/Loader Mode:** Facilitates safe program transfer from ROM or manual input to RAM with handshake-based loader control, enabling debugging and instructional demonstration.

4. **Hardwired control unit:** Timing states from the ring counter are combined with decoded opcode signals to generate micro-operation pulses. These pulses govern register loading, ALU activity, memory access, program sequencing, and halting with precise synchronization.

5. **Single-driver bus control:** Strict tri-state control is maintained to ensure that only one source drives the bus during each timing phase, eliminating contention and ensuring accurate data propagation.

6. **Datapath configuration:** The datapath consists of two 8-bit registers (Accumulator A and B), a ripple-carry Arithmetic Logic Unit (ALU) capable of arithmetic and shift operations, a 4-bit Program Counter (PC) with both increment and direct-load modes, a 4-bit Memory Address Register (MAR), a 16×8-bit Static RAM, and an 8-bit Instruction Register (IR) for opcode and operand separation.

7. **Opcode decoder:** A 4-to-16 decoder has been employed to translate opcode bits into one-hot control signals corresponding to the available instructions, facilitating straightforward signal mapping to the control sequencer.

8. **Precise timing control:** Each instruction is executed in a specific number of timing states. Memory-based instructions (`LDA, LDB`) require five timing cycles

(T1–T5), shift and arithmetic instructions (`ADD, SHL, SHR`) require four cycles, and control instructions (`JUMP, HLT`) are completed within four or fewer cycles.

9. **Assembler and verification support:** A custom assembler converts human-readable assembly code (`ORG`, `DEC` directives with mnemonics) into Logisim-compatible hexadecimal files. This enables straightforward program loading and verification through waveform observation and register monitoring.

10. **Extensibility and educational value:** The system is structured for scalability, allowing additional instructions or status flags (e.g., Zero, Carry) to be introduced easily. Its visual design and step-by-step operation make it ideal for demonstrating processor internals in an academic laboratory environment.

4 Architectural Design and Component Analysis

4.1 System Architecture Overview

The processor is built on a single-bus data transfer architecture, enabling all components to communicate through a common pathway. To avoid signal contention, tri-state logic ensures that only one module drives the bus at any given moment, while others remain in a high-impedance state. This bus interconnects the memory, registers, and ALU, coordinating data movement through carefully timed control signals. Typical bus-driving sources include`pc_out`, `sram_rd`, `ins_reg_out_en`, `a_out`, `b_out`, `alu_out`, and `shift_out`. while bus-receiving elements such as `mar_in_en`, `a_in`, `b_in`, and `sram_wr` accept data from the bus when their enable lines are asserted.

Figure 1: Overall SAP-1 processor architecture in automatic mode, illustrating unified single-bus data flow and control signal distribution across all subsystems.

Figure 2: Manual/Loader mode architecture demonstrating ROM-to-RAM program transfer with handshake control for safe and verified data loading.

4.2 Register Implementation (A and B)

The register subsystem comprises two 8-bit general-purpose registers, designated as Register A and Register B, implemented using conventional flip-flop-based storage elements. Each register is equipped with distinct input and output control signals, facilitating independent data read and write operations in accordance with the system's control logic.

.

- **Purpose:** To store operands and intermediate results during instruction execution.
- **Functionality:** Data from the system bus is latched when `a_in` or `b_in` is asserted. Each register can also drive the bus using its respective `a_out` or `b_out` control. Additionally, internal connections provide continuous data to the ALU for computation without occupying the main bus.

Figure 3: A/B Register subsystem showing input/output enable controls and internal connectivity to the ALU for computation without bus usage.

4.3 Program Counter (PC)

The Program Counter is a 4-bit register capable of incrementing sequentially or loading a new address during branching. It drives the system bus through `pc_out` and accepts new data when `pc_en`

- **Purpose:** To hold and manage the address of the next instruction.
- **Operation:**
  - During T1, `pc_out` and `mar_in_en` transfer the PC value to MAR for fetching.
  - During T3, the counter increments (`pc_en`=1).

Figure 4: Program Counter (PC) implementation with increment and load modes for both sequential and jump-based instruction flow control.

4.4 Memory System and Address Register (MAR)

The memory subsystem consists of a 4-bit Memory Address Register (MAR) and a 16×8-bit SRAM array. The MAR captures the address of the instruction or data to be accessed, while SRAM handles the storage and retrieval operations.

- **Purpose:** To provide address-driven access to memory.
- **Operation:**
  - During T1, MAR receives the address from PC for instruction fetch.
  - During T4, MAR is loaded from IR[3:0] during operand addressing for instructions such as `LDA`, `LDB`, and `SHLA SHRA`.
  - `sram_rd`=1 enables read operations, while `sram_wr`=1 performs writes.

Figure 5: SRAM cell implementation showing D flip-flop based storage with read, write, and chip-select control signals.

Figure 6: Complete 16×8 SRAM array connected with MAR, supporting address-based data access through read/write control.

4.5 Instruction Register and Opcode Decoder

The Instruction Register (IR) holds the instruction fetched from memory and separates it into two 4-bit fields: opcode and operand. The opcode field is forwarded to the decoder, which produces one-hot control lines for each instruction.

- **Purpose:** To store and decode the current instruction.
- **Operation:**
  - During T2, the instruction is fetched from memory (`sram_rd=1`) and latched into IR (`ins_reg_in_en=1`).
  - IR[7:4] drives the decoder, while IR[3:0] can be placed on the bus for addressing during T4.

Figure 7: Instruction Register (IR) and Opcode Decoder showing instruction loading, opcode decoding, and operand forwarding paths.

Figure 8: Opcode Decoder converting 4-bit opcodes into one-hot control signals for all nine supported instructions.

4.6 Arithmetic Logic Unit (ALU)

The ALU performs all arithmetic and logical operations. It supports addition and both left and right shift operations for registers A and B, enhancing computational capability.

- **Purpose:** To execute arithmetic and shift operations based on control signals.
- **Operation:** Inputs are directly taken from internal outputs of Registers A and B, reducing bus traffic. The signal `alu_out`=1 places the computed result onto the bus. Shift operations are controlled using `shift_out` and `right_or_left` signals, determining the direction of bit movement.

Figure 9: Arithmetic Logic Unit (ALU) implementing 8-bit addition and bidirectional shift operations, utilizing ripple-carry and tri-state bus interfacing.

4.7 Timing Control Generator

The timing control generator employs a 6-phase ring counter that sequentially activates states (T1–T6) for the execution of micro-operations. Each timing phase corresponds to specific control signal activations, ensuring accurate synchronization between fetching, decoding, and executing instructions.

Figure 10: Timing generator using a six-phase ring counter to coordinate fetch–decode–execute cycles for all instruction types.

Universal Fetch Sequence (for all instructions):

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

Execution Cycles by Instruction:

- **LDA addr:** T4 → `ins_reg_out_en`, `mar_in_en` → MAR ← IR[3:0]; T5 → `sram_rd`, `a_in` → A ← M[MAR].
- **LDB addr:** T4 → `ins_reg_out_en`, `mar_in_en` → MAR ← IR[3:0]; T5 → `sram_rd`, `b_in` → B ← M[MAR].
- **ADD:** T4 → `a_out`, `b_out`, `alu_out`, `a_in` → A ← A + B.
- **SHL A:** T4 → `a_out`, `shift_out`=1, `right_or_left`=0, `a_in` → A shifted left by one bit.
- **SHR A:** T4 → `a_out`, `shift_out`=1, `right_or_left`=1, `a_in` → A shifted right by one bit.
- **HLT:** T4 → `hlt`=1 → Halts the system by disabling the ring counter.

Figure 11: Control Sequencer generating instruction-dependent control pulses synchronized with the six-phase ring counter for both arithmetic and shift operations.

4.8 Automatic Operation Control Logic

The **control logic implementation** was formulated using Boolean equations to define the activation timing and sequencing of all micro-operations for each instruction. In this implementation, the control enable signal (`c_en`) was employed to initiate and sustain the automatic control sequence. When `c_en = 1`, the control sequencer is activated, allowing the processor to progress through the structured fetch–decode–execute cycle. Conversely, when `c_en = 0`, the sequencer halts, suspending all micro-operations until re-enabled.

The `debug` pin was separately utilized to manually control the system during program loading or diagnostic operations. When `debug = 1`, the automatic sequencer remains disabled, permitting manual memory access and signal observation without interference from the control logic. This separation between `c_en` and `debug` ensures safe mode switching and prevents timing conflicts between manual and automatic execution paths.

Figure 12: Gate-level control matrix showing the automatic control signal derivation for each instruction type, combining timing states and decoded opcode signals to achieve precise micro-operation coordination.

The **automatic control logic** was developed to generate precise control signals for each micro-operation by combining timing pulses (T1–T6) with instruction-specific enable lines. All control signals are globally gated by the `c_en` (Control Enable) pin, which activates the control sequencer. When `c_en` is asserted, the fetch–decode–execute cycle proceeds automatically; when it is deasserted, all control outputs remain inactive, ensuring complete isolation between the automatic and manual operation modes.

Listing 1: Fetch Control Equations

```
pc_out = T1
mar_in_en = (T1) | (T4 & (insLDA | insLDB ))
sram_rd = (T2) | (T5 & (insLDA | insLDB))
ins_reg_in_en = T2
pc_en = T3
```

Listing 2: ALU

```
alu_out = T4 & insADD
shift_out = T4 & (insSHLA | insSHRA )
right_or_left = T4 & (insSHRA )
a_in = (T5 & insLDA) | (T4 & (insADD | insSHLA | insSHRA))
a_out = T4 & (insADD | insSHLA | insSHRA)
```

These equations ensure that all register, ALU, and shift operations occur exclusively within their assigned timing windows, maintaining a conflict-free and synchronized bus environment. The hardwired design guarantees deterministic behavior, minimizing the need for sequential microcoding and reducing control latency.

4.9 Manual/Loader Operation Control

In the **Manual/Loader mode**, the control sequence is modified through the activation of the `debug = 1` signal. When this signal is asserted, the automatic fetch–decode–execute control path is disabled, allowing safe manual program loading into RAM from the ROM module. During this period, the processor remains isolated from automatic control signals, preventing unintended data flow through the bus.

Figure 13: Manual/Loader control sequencer design showing debug-controlled program transfer to RAM. The sequencer remains disabled while the debug signal is active, ensuring safe and isolated memory loading.

When `debug` is set HIGH, the `mar_in_manual` and `sram_wr` lines are enabled, allowing data values from the ROM to be written sequentially into RAM according to the address counter. The address counter automatically increments with each clock pulse, ensuring that consecutive ROM locations are transferred to corresponding RAM addresses without overlap. Once all sixteen addresses have been written, the

loader section automatically deactivates the `debug` signal and asserts the `c_en` pin, reactivating the automatic control sequencer.

Figure 14: Automatic transition of control after program loading, where the `c_en` pin is enabled following completion of all 16 address transfers.

This mechanism ensures seamless transition between manual and automatic modes. During manual loading, all automatic control signals remain masked, thereby maintaining complete operational isolation. After the loading process concludes, the system automatically resumes normal execution under the automatic mode without requiring external reset intervention.

5 Instruction Set Architecture

5.1 Instruction Encoding Scheme

The instruction format utilizes the upper four bits (`IR[7:4]`) to represent the opcode and the lower four bits (`IR[3:0]`) to represent the operand or address field. This allows up to 16 unique instructions, of which nine are utilized in the current design.

Table 1: Instruction Set Summary

Address (Binary)	Instruction Code (Binary)	Hex Code	Mnemonic
0000000	00011010	1A	LDA 10
00000001	00101111	2B	LDB 11
00000010	00110000	30	ADD
00000011	01000000	40	SHL A
00000100	01010000	50	SHR A
00000101	01100000	60	HLT

Instruction Descriptions:

- **LDA 10:** Loads the value from memory location `M[10]` into the Accumulator (A).
- **LDB 11:** Loads the value from memory location `M[11]` into Register B.
- **ADD:** Adds the contents of Register B to the Accumulator (A) and stores the result in A.
- **SHL A:** Shifts the contents of Accumulator A one bit to the left.
- **SHR A:** Shifts the contents of Accumulator A one bit to the right.
- **HLT:** Terminates the program execution and halts the processor.

Table 2: Data Values Stored in Memory (RAM)

Address (Binary)	Data (Binary)	Decimal	Hexadecimal
00001010	00001101	13	0D
00001011	00011001	21	15

The instruction set supports arithmetic, shifting, and control operations, allowing both data manipulation and conditional branching within a compact architecture.

5.2 Assembler

A lightweight assembler tool was designed to convert human-readable assembly instructions into hexadecimal machine code compatible with Logisim Evolution. The assembler supports all implemented mnemonics, including `LDA, LDB, ADD, SHL A` and `HLT`. Each instruction is directly translated into its corresponding 8-bit machine code, with the upper nibble representing the opcode and the lower nibble representing the operand or address.

This assembler was primarily used to generate ROM initialization files for automatic program loading. Unlike traditional assemblers, no directives such as `ORG` or `DEC` were required, since instruction and data placement were predefined in the design's memory map. This approach ensured compatibility with the automatic loading mechanism, where program bytes were sequentially transferred from ROM to SRAM under control of the `debug` and `c_en` signals.

```
%%bash
# Create the assembly source
cat > /content/prog.asm <<'ASM'
ORG 0
LDA 1110
LDB 1111
SHL_A
HLT
ORG 14
DATA 0101
DATA 1010
ASM

# Run the assembler on that file
python3 /content/sap1_asm.py /content/prog.asm
```

```
1E 2F 60 B0 00 00 00 00 00 00 00 00 00 00 05 0A
```

Figure 15: Custom web-based assembler translating SAP-1 assembly mnemonics into Logisim-compatible hexadecimal code format.

Table 3: Example Assembly Programs and Corresponding Machine Codes

Example	**Assembly Code**	**Generated Hex Code**
Addition and Shift Program	LDA 10; LDB 11; ADD; SHL A; HLT	1A 2B 30 40 90
Branching Program	LDA 10; LDB 11; ADD; SHL A; HLT	1A 2B 30 40 90

The assembler-generated file can be loaded directly into the ROM component in Logisim Evolution. During execution, the loader mode transfers this ROM content into RAM, ensuring that each instruction and data value is accurately mirrored in memory.

6 Operation

6.1 Fetch–Decode–Execute Cycle

The processor performs all instruction executions through a systematic sequence of fetch, decode, and execute phases, orchestrated by the control sequencer and the six-phase timing generator.

Fetch Phase:

- **T1:** The Program Counter (PC) outputs its current value to the bus, which is captured by the Memory Address Register (MAR).
- **T2:** The SRAM places the instruction stored at MAR on the bus (`sram_rd = 1`), and the Instruction Register (IR) latches this value.
- **T3:** The PC increments its content by one (`pc_en = 1`), preparing for the next instruction.

Decode Phase: The opcode portion (IR[7:4]) is sent to the decoder, which asserts the corresponding instruction line (e.g., `insLDA, insADD, insSHLA`). This decoded output, when combined with the active timing pulse, determines the micro-operations required for the instruction.

Execute Phase: Depending on the instruction, data transfers and computations occur in the subsequent timing states. For instance:

- `LDA` or `LDB`: Executes memory read and register load in T4–T5.
- `ADD`: Executes arithmetic addition in T4.
- `SHL`/`SHR`: Executes bit-shifting operations in T4.
- `HLT`: Disables timing pulses to stop the execution cycle.

The fetch–decode–execute sequence repeats continuously until a `HLT` instruction is encountered, at which point the control sequencer disables further timing signals and halts the processor.

7 Running the CPU in Automatic Mode (ADD + SHIFT Program)

The automatic mode execution of the SAP-1 processor was verified using a program composed of `JUMP`, `ADD`, and shift instructions. The following procedure describes the loading, execution, and output verification process.

1. Initial Setup

[label=(g)]

1. The Logisim Evolution project was opened.
2. The `debug` switch was confirmed to be in the **LOW** state, enabling automatic control.
3. The main clock (`clk`) was turned **OFF** to prevent premature execution.
4. The `pc_reset` pin was pulsed to initialize the Program Counter (PC) to address `0000`.

2. Programming the ROM

1. The ROM contents were edited using the `Edit Contents` option.
2. The following instruction sequence was entered:

   ```
   1A 2B 30 40 90 0D 00
   ```

   representing the instructions `LDA 10, LDB 11, ADD, SHL A, HLT` with data values 13 and 25.

3. Loading the Program into RAM (Bootloader Mode)

1. The `debug` switch was set to **HIGH**, enabling ROM-to-RAM data transfer.
2. Each clock pulse transferred one instruction or data byte sequentially from ROM into SRAM.
3. MAR and Data Bus activity were observed, confirming proper address–data synchronization.

Figure 16: ROM-to-RAM data transfer during bootloader mode. Each instruction and data word was written sequentially into SRAM memory.

4. Terminating Bootloader Operation

1. The `debug` signal was returned to **LOW** to disable the loader circuit.
2. A single clock pulse ensured the bootloader returned to idle state before execution began.

5. Program Execution in Automatic Mode

1. The Program Counter was reset again to `0000`.
2. Clock pulses were applied manually or continuously to begin instruction execution.
3. During each cycle, the values of PC, MAR, IR, A, and B registers were monitored through display outputs.
4. The program execution sequence was as follows:
   - `LDA 10:` Value from address 10 loaded into Register A.

- `LDB 11`: Value from address 11 loaded into Register B.
- `ADD`: Contents of A and B added and stored back in Register A.
- `SHL A` : A shifted left, B shifted right for bit-level operations.
- `HLT`: Processor halted, ending execution.

During the execution of the `LDA`, `LDB`, and `ADD` instructions, the processor successfully fetched the data from memory and performed arithmetic operations within the ALU. Register A and Register B loaded the values `13` and `25` from addresses 10 and 11, respectively. The ALU output showed the correct sum (`38`), stored back into Register A, demonstrating proper data transfer and arithmetic operation.

Figure 17: Execution of LDA and LDB followed by ADD operation. Register A and B load values 13 and 21 respectively, and the ALU outputs the sum (34) stored back into Register A.

Next, the `SHL A` instruction was executed to verify the bitwise manipulation functionality. In this stage, the data within Register A was shifted left and that in Register B was shifted right, confirming correct operation of the shift control logic integrated within the ALU.

Figure 18: Execution of SHL A. Register A contents shifted left

Finally, the HLT instruction was reached, which halted the control sequence and disabled further clock-controlled micro-operations. At this stage, the outputs of all registers and memory were frozen, and the system successfully displayed the final computed results.

Figure 19: Final system state after HLT instruction. The control sequencer stopped operation, and all registers retained their final computed values.

6. Output Verification

Upon completion of the program, all register and memory outputs were examined to verify correct data flow and instruction execution. The values observed in the seven-segment displays and memory units confirmed successful operation of each functional block of the processor.

During the arithmetic phase, the `LDA` and `LDB` instructions correctly loaded the data values `13` and `21` from memory addresses `10` and `11` into Registers A and B, respectively. Subsequently, the `ADD` instruction executed an 8-bit addition within the ALU, producing the correct result of `00100010` (binary), equivalent to decimal **34**. This result was stored back into Register A and later written to memory during program termination.

The `JUMP 2` instruction redirected program control to address `2`, demonstrating accurate branching and instruction looping behavior. This verified that the Program Counter (PC) and instruction decoder operated in synchronization with the control sequencer.

In the subsequent shift operations, the instructions `SHL A` successfully modified the bit patterns of Registers A and B, confirming the correct functioning of the ALU's shift control logic and directional bit manipulation circuits.

Finally, upon reaching the `HLT` instruction, the processor halted execution as designed. All control lines were deactivated, the ring counter ceased operation, and the final data values were retained across registers and memory. The contents of memory address `00001010` (decimal 10) displayed the binary value `00100110`, verifying that arithmetic, branching, and bitwise operations executed flawlessly under automatic mode control.