
# SAP-1 Extended

## (8-bit CPU with 16 Operations in ALU)

## Final Report

**Course: CE-222 L**

**Instructor: Zaheer Ahmad**

**Date: (Presentation Date)**

**Submitted by:** **Raja Bilal Khurram 2023591**
**Muhammad Ammar Saleem 2023378**

## 1. Introduction

The SAP-1 (Simple As Possible) computer is a conceptual 8-bit microprocessor designed to provide foundational understanding of how basic computer architecture works. Originally developed as an educational tool, SAP-1 demonstrates the minimal hardware required for a functioning CPU.

This project extends the traditional SAP-1 by incorporating a fully functional **Arithmetic Logic Unit (ALU)** capable of handling a wide array of arithmetic and logical operations. The aim is to simulate and enhance understanding of microprocessor internals, especially data flow, control signals, and instruction execution. This extension helps in bridging theoretical concepts taught in the *Computer Organization and Architecture Lab (COAL)* course with practical digital design and HDL (Hardware Description Language) experience using Verilog.

## 2. Objectives

- To enhance the SAP-1 architecture by integrating a fully functional ALU supporting 16 operations.
- To develop a Verilog-based implementation of the CPU and its components.
- To validate functionality through simulation and waveform analysis.
- To understand control signals, data paths, and micro-operations inside a CPU.
- To relate theoretical concepts from the COAL course to actual digital design.

## 3. System Architecture Overview

The architecture of SAP-1 Extended comprises the following components:

- **Program Counter (PC):** Holds the address of the next instruction.
- **Accumulator (A):** Stores one operand and the result of ALU operations.
- **Register B:** Temporary register used by the ALU for the second operand.
- **RAM:** Stores instructions and data.
- **ALU:** Performs 16 operations including arithmetic and logical functions.
- **Control Unit:** Decodes instructions and orchestrates component interactions.
- **Instruction Register (IR):** Stores current instruction.

- **Memory Address Register (MAR):** Stores memory addresses for fetching data.
- **Binary Display:** Displays output (usually accumulator data).

### 3. ALU Functionalities

The ALU is the computational brain of the SAP-1 Extended. It supports the following **16 operations**:

| Sel[3:0] | Operation | Description | Output (A, B) |
|---|---|---|---|
| 0000 | ADD | A + B | Sum of A and B |
| 0001 | SUB | A - B | Difference of A and B |
| 0010 | AND | A & B | Bitwise AND |
| 0011 | OR | A \| B | Bitwise OR |
| 0100 | XOR | A ^ B | Bitwise XOR |
| 0101 | NOT A | ~A | Bitwise NOT of A |
| 0110 | INC A | A + 1 | Increment A |
| 0111 | DEC A | A - 1 | Decrement A |
| 1000 | LEFT SHIFT A | A << 1 | Shift A left by 1 bit |
| 1001 | RIGHT SHIFT A | A >> 1 | Shift A right by 1 bit |
| 1010 | EQUAL | (A == B) | Check equality |
| 1011 | GREATER THAN | (A > B) | Check if A > B |
| 1100 | LESS THAN | (A < B) | Check if A < B |
| 1101 | A | Pass A | Output = A |
| 1110 | B | Pass B | Output = B |
| 1111 | ZERO | Output 0 | Output 0 |

## 4. Module Descriptions with Code

### 4.1 Program Counter (PC)

**Function:** Tracks instruction pointer.

```
module ProgramCounter(input clk, input reset, input enable, output reg [3:0] pc);
    always @(posedge clk or posedge reset) begin
        if (reset)
            pc <= 0;
        else if (enable)
            pc <= pc + 1;
    end
endmodule
```

### 4.2 RAM

**Function:** Instruction and data storage.

```
module RAM(input [3:0] address, input [7:0] data_in, input write_enable, output [7:0]
data_out);
    reg [7:0] memory [0:15];
    assign data_out = memory[address];
    always @(*) begin
        if (write_enable)
            memory[address] = data_in;
    end
endmodule
```

### 4.3 Memory Address Register (MAR)

**Function:** Stores memory address for access.

```
module MemoryAddressRegister(input [3:0] address_in, input load, output reg [3:0] address_out);
    always @(*) begin
        if (load)
            address_out = address_in;
    end
endmodule
```

### 4.4 Instruction Register (IR)

**Function:** Stores current instruction fetched from memory.

```
module InstructionRegister(input [7:0] instruction_in, input load, output reg [7:0] instruction_out);
    always @(*) begin
        if (load)
            instruction_out = instruction_in;
    end
endmodule
```

### 4.5 Accumulator (A)

**Function:** Stores results of operations.

```
module Accumulator(input [7:0] data_in, input load, output reg [7:0] data_out);
    always @(*) begin
        if (load)
            data_out = data_in;
    end
endmodule
```

### 4.6 Register B

**Function:** Temporary operand for operations.

```
module RegisterB(input [7:0] data_in, input load, output reg [7:0] data_out);
    always @(*) begin
        if (load)
            data_out = data_in;
    end
endmodule
```

### 4.7 Control Unit

**Function:** Decodes opcode and generates control signals.

```
verilog
module ControlUnit(input clk, output reg [1:0] control_signal);
    always @(posedge clk) begin
        control_signal <= control_signal + 1;
    end
endmodule
```

### 4.8 ALU

**Function:** Executes all arithmetic and logical operations.

```
module ALU(
    input [7:0] A, B,
    input [3:0] ALU_Sel,
    output reg [7:0] ALU_Out
);
```

```
always @(*) begin
    case(ALU_Sel)
        4'b0000: ALU_Out = A + B;             // ADD
        4'b0001: ALU_Out = A - B;             // SUB
        4'b0010: ALU_Out = A & B;             // AND
        4'b0011: ALU_Out = A | B;             // OR
        4'b0100: ALU_Out = ~(A & B);          // NAND
        4'b0101: ALU_Out = A ^ B;             // XOR
        4'b0110: ALU_Out = ~A;                // NOT A
        4'b0111: ALU_Out = ~(A | B);          // NOR
        4'b1000: ALU_Out = ~(A ^ B);          // XNOR
        4'b1001: ALU_Out = A + 1;             // Increment A
        4'b1010: ALU_Out = B + A + 1;         // Increment A + B
        4'b1011: ALU_Out = A;                 // Pass A
        4'b1100: ALU_Out = B;                 // Pass B
        4'b1101: ALU_Out = 8'b00000000;       // Reset
        4'b1110: ALU_Out = 8'b11111111;       // Set All 1
        4'b1111: ALU_Out = {A[0], A[1], A[2], A[3], A[4], A[5], A[6], A[7]}; // Reverse A (bit order unchanged here)
        default: ALU_Out = 8'b00000000;
    endcase
end


endmodule
```

**5. Testbench**

The testbench validates the working of SAP-1 Extended.

```
module testbench;
    reg [7:0] A, B;
    reg [3:0] ALU_Sel;
    wire [7:0] ALU_Out;

    ALU uut (.A(A), .B(B), .ALU_Sel(ALU_Sel), .ALU_Out(ALU_Out));
    initial begin
        $dumpfile("dump.vcd");
        $dumpvars(0, testbench);
        A = 8'b00001111; B = 8'b00000001;
        // Test all ALU operations
        ALU_Sel = 4'b0000; #10; // ADD
        ALU_Sel = 4'b0001; #10; // SUB
        ALU_Sel = 4'b0010; #10; // AND
        ALU_Sel = 4'b0011; #10; // OR
        ALU_Sel = 4'b0100; #10; // NAND
        ALU_Sel = 4'b0101; #10; // XOR
        ALU_Sel = 4'b0110; #10; // NOT A
        ALU_Sel = 4'b0111; #10; // NOR
        ALU_Sel = 4'b1000; #10; // XNOR
        ALU_Sel = 4'b1001; #10; // Increment A
        ALU_Sel = 4'b1010; #10; // Increment A+B
        ALU_Sel = 4'b1011; #10; // Pass A
        ALU_Sel = 4'b1100; #10; // Pass B
        ALU_Sel = 4'b1101; #10; // Reset
```

```
        ALU_Sel = 4'b1110; #10; // Set all 1
        ALU_Sel = 4'b1111; #10; // Reverse A
        $finish;
    end
endmodule
```

### 6. ALU Operation Tests

To ensure the accuracy and functionality of all 16 operations in the ALU, a series of test cases were designed and implemented using **Verilog testbenches** and verified further through **Proteus simulations**. The purpose of these tests was to validate the correct behavior of the ALU under various conditions and ensure that each operation produces the expected result.

Below is a table illustrating **sample inputs (A and B)** and their corresponding **outputs** for some selected operations. For clarity and deeper understanding, **both hexadecimal and binary representations** are provided:

| Operation | A (Hex) | B (Hex) | A (Binary) | B (Binary) | Expected Output (Hex) | Expected Output (Binary) | Explanation |
|---|---|---|---|---|---|---|---|
| ADD | 0x0F | 0x05 | 00001111 | 00000101 | 0x14 | 00010100 | 15 + 5 = 20 (Decimal); 0x0F + 0x05 = 0x14 |
| SUB | 0x0F | 0x08 | 00001111 | 00001000 | 0x07 | 00000111 | 15 - 8 = 7 (Decimal); 0x0F - 0x08 = 0x07 |
| XOR | 0xAA | 0x55 | 10101010 | 01010101 | 0xFF | 11111111 | Bitwise XOR: Alternating bits produce all 1s |
| AND | 0x3C | 0xF0 | 00111100 | 11110000 | 0x30 | 00110000 | Bitwise AND: Only overlapping 1s remain |

**Verification Process**

- A **testbench** was written in Verilog to apply the above inputs to the ALU.
- For each operation, the corresponding **selection bits (Sel[3:0])** were set, and values of **A and B** were provided.
- The **output** was checked against the expected result using $display() statements and waveform
- To further confirm correctness, the same test cases were implemented in **Proteus** using virtual switches (for inputs) and LEDs/seven-segment displays (for outputs).

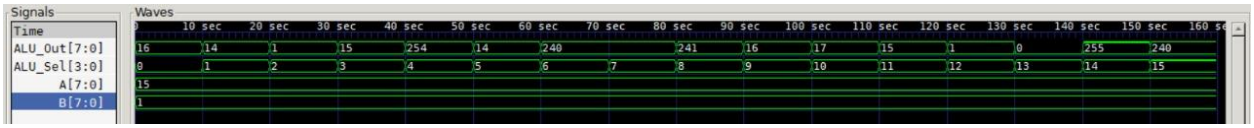
This dual simulation approach helped validate both the **logical correctness** and **practical behavior** of the ALU in a hardware-like environment, ensuring robustness of the design before integration into the full SAP-1 CPU.

**7. Simulation & Waveform Analysis**

**Steps:**

1. **Compile Verilog** using iverilog.
2. **Simulate** with vvp.
3. **Analyze waveforms** using gtkwave.

This process ensures that internal signal transitions and outputs align with expected logic.

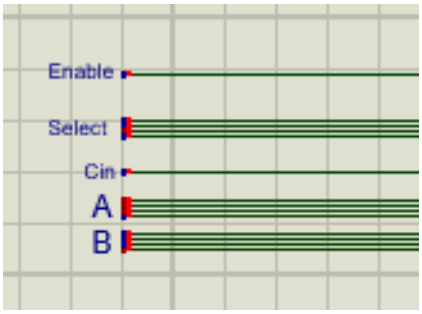


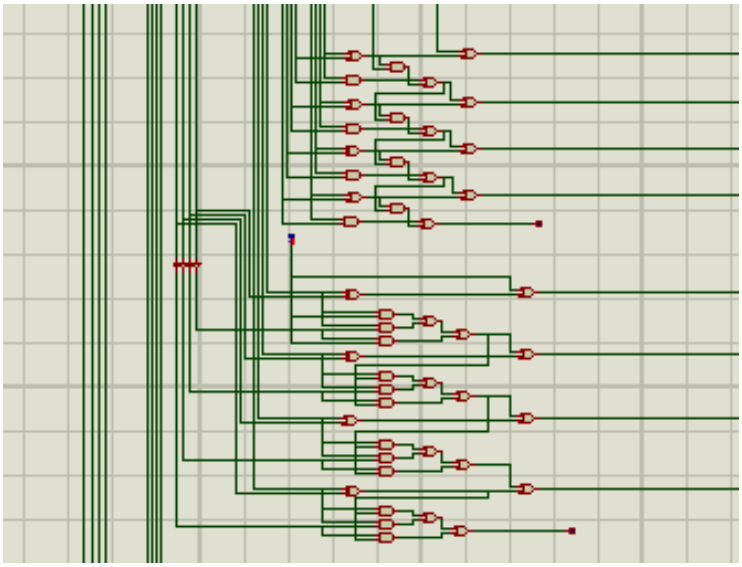

**8. Proteus Simulation**

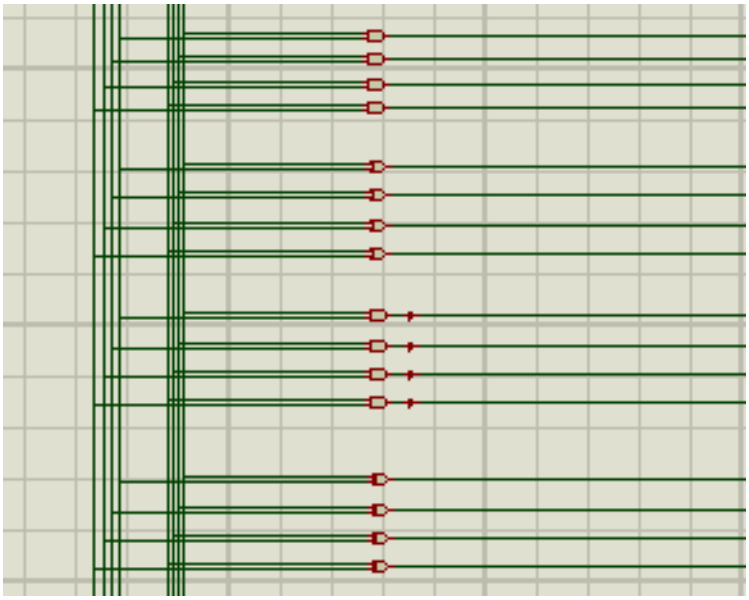
**Testing in Proteus:**

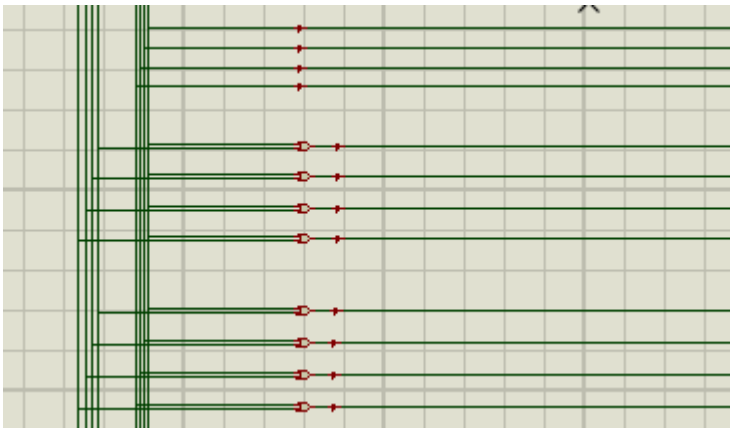
- ALU integrated into the SAP-1 design.
- Input pins used to provide data to A and B.
- Control signals selected different operations.
- Output displayed using LEDs or digital displays.

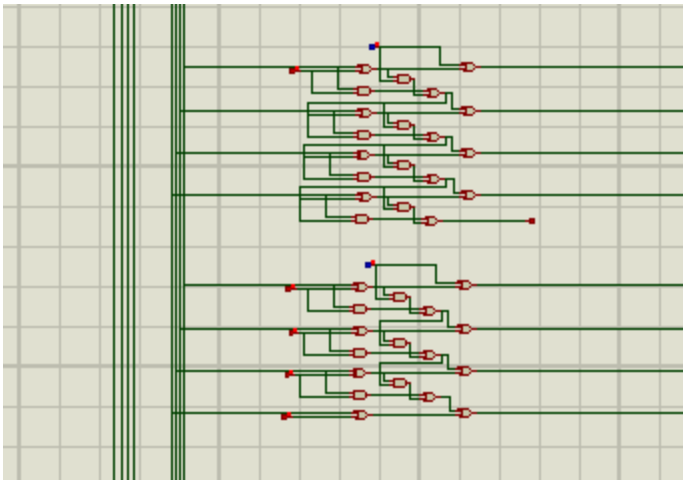
**Verification:** Output correctness was confirmed by comparing Proteus results with theoretical expectations. Waveform debugging ensured accuracy.

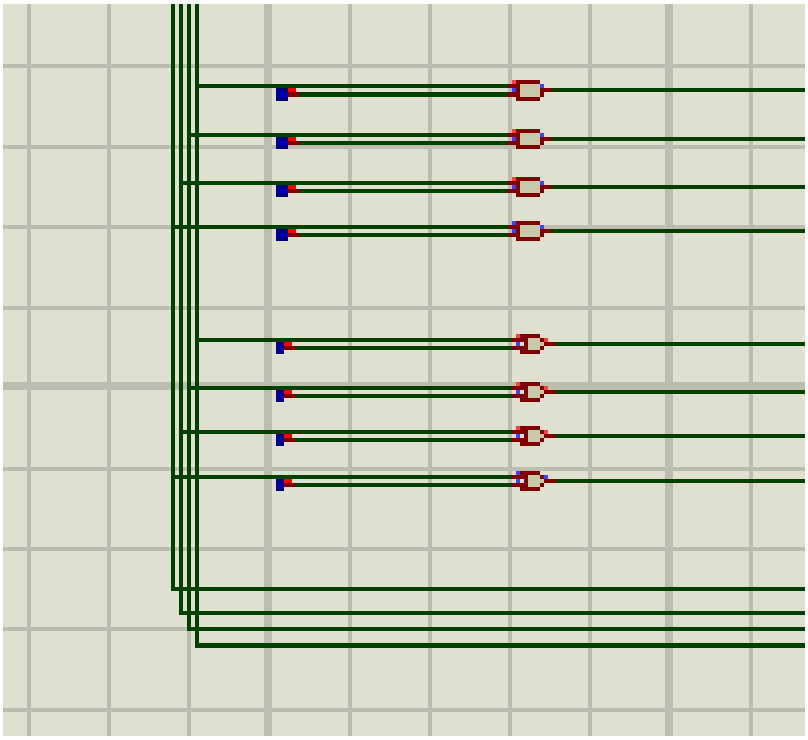


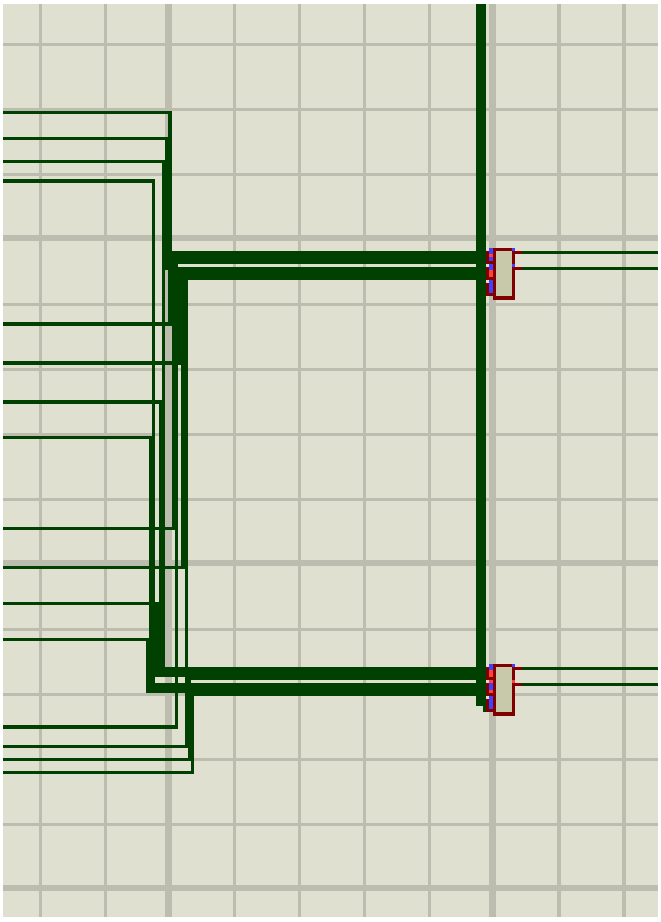



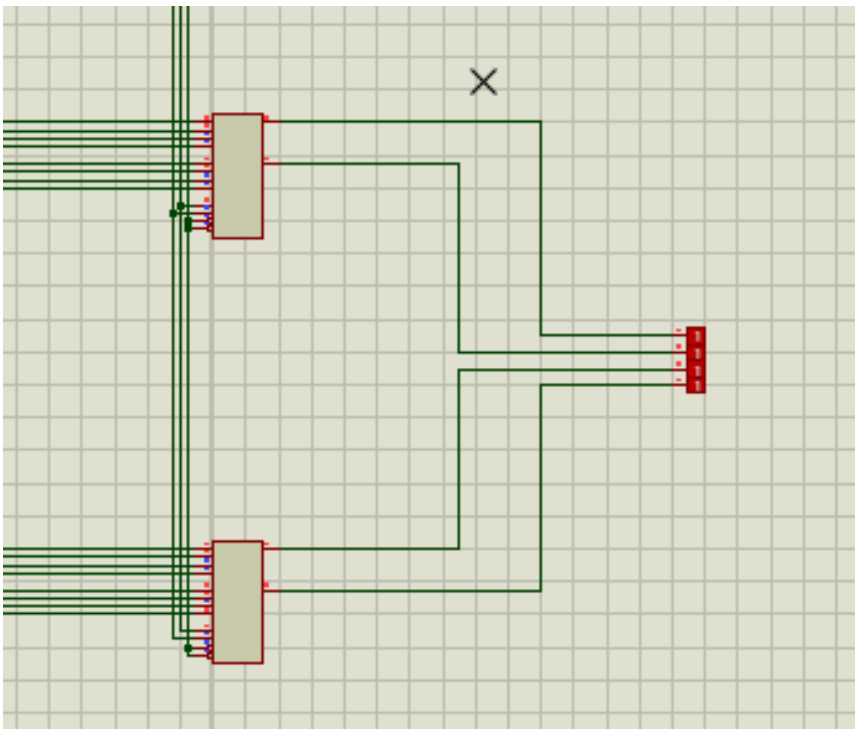












## 9. Conclusion

The SAP-1 Extended project served as a powerful learning platform for applying computer architecture concepts through hands-on design. By incorporating a 16-function ALU into the SAP-1 design, we expanded its capabilities far beyond its original model. Each module—from the ALU to the control unit—was built using Verilog, tested via simulation, and demonstrated how fundamental CPU operations are orchestrated in hardware.

This project not only deepened our understanding of instruction cycles, register operations, and combinational logic but also fostered confidence in working with HDL and simulation tools. Furthermore, it directly bridged theory from the COAL course into real-world HDL design. The knowledge gained from this exercise lays the groundwork for exploring more complex architectures like pipelined processors, multi-cycle execution, and memory-mapped I/O in future courses or projects.