CE222L – Project Report

Enhanced SAP-1 Computer Using Verilog

Section - C

Faculty – CS

2023330	MOHAMMAD MUSA ALI
2023640	SAROSH ISHAQ
2023538	MUHAMMAD UMER

1. Introduction

The **SAP-1 (Simple-As-Possible) Computer** is a basic 8-bit educational computer architecture designed to demonstrate the fundamental concepts of computer organization and operation. This project focuses on implementing SAP-1 in **Verilog HDL** and providing a **frontend interface** for interaction.

Objectives:

- Design and simulate SAP-1 architecture in Verilog.
- Implement the instruction set (LDA, ADD, SUB, AND, OR, XOR, JMP, JZ, OUT, HLT).
- Develop a frontend (web-based or GUI) to visualize and interact with the SAP-1 simulation.

2. SAP-1 Architecture Overview

The SAP-1 consists of the following key components:

1. **8-bit Program Counter (PC)** – Keeps track of the next instruction address.
2. **8-bit Memory (RAM)** – Stores instructions and data.
3. **8-bit Accumulator (ACC)** – Holds intermediate results.
4. **8-bit Instruction Register (IR)** – Stores the current instruction.
5. **Control Unit (CU)** – Decodes instructions and generates control signals.
6. **Arithmetic Logic Unit (ALU)** – Performs arithmetic and logical operations.
7. **Output Register (OUT)** – Displays the result.

Instruction Format:

- **4-bit Opcode + 4-bit Operand** (Memory Address or Data)

3. Verilog Implementation

1. System Block Diagram

Before writing Verilog, design the SAP-1 architecture with the following components:

1. **Program Counter (PC)** – Tracks the next instruction address.
2. **Memory (RAM)** – Stores instructions and data (8-bit address bus).
3. **Instruction Register (IR)** – Holds the current instruction (Opcode + Operand).
4. **Accumulator (ACC)** – Stores intermediate results.
5. **Arithmetic Logic Unit (ALU)** – Performs operations (ADD, SUB, AND, OR, XOR).
6. **Control Unit (CU)** – Decodes instructions and generates control signals.
7. **Output Register (OUT)** – Displays the final result.

Data Flow:

```
PC → Memory → IR → CU → (ACC/ALU/Memory) → Output
```

2. Step-by-Step Implementation Procedure

Step 1: Define the Instruction Set

Map each **4-bit opcode** to its operation:

Opcode	Mnemonic	Function
0000	LDA	Load from memory to ACC
0001	ADD	Add memory value to ACC
0010	SUB	Subtract memory value from ACC

0011	AND	Bitwise AND with ACC
0100	OR	Bitwise OR with ACC
0101	XOR	Bitwise XOR with ACC
0110	JMP	Jump to memory address
0111	JZ	Jump if ACC is zero
1110	OUT	Output ACC value
1111	HLT	Halt execution

Step 2: Design the Modules

1. Program Counter (PC)

- **Inputs:** `clk`, `reset`, `jump_signal` (for JMP/JZ).
- **Outputs:** `address` (current instruction address).
- **Behavior:**
    - Increments by `1` on each clock cycle.
    - Resets to `0` on `reset`.
    - Updates to a new address on `jump_signal`.

2. Memory (RAM)

- **Inputs:** `address` (from PC or JMP).
- **Outputs:** `data` (8-bit instruction/data).
- **Behavior:**
    - Stores instructions in a **preloaded memory array** (e.g., from a `.hex` file).
    - Outputs the instruction/data based on `address`.

3. Instruction Register (IR)

- **Inputs:** `clk`, `load` (from CU), `data` (from Memory).
- **Outputs:** `opcode` (4-bit), `operand` (4-bit).
- **Behavior:**
    - On `load`, splits the 8-bit instruction into `opcode` and `operand`.

4. Accumulator (ACC)

- **Inputs:** `clk`, `load` (from CU), `data` (from Memory/ALU).
- **Outputs:** `value` (8-bit stored data).
- **Behavior:**
    - Stores results from `LDA`, `ADD`, `SUB`, etc.
    - Outputs `value` to ALU or OUT register.

5. ALU (Arithmetic Logic Unit)

- **Inputs:** `op` (operation code), `a` (ACC), `b` (Memory data).
- **Outputs:** `result` (8-bit), `zero_flag` (for JZ).
- **Behavior:**
    - Performs `ADD`, `SUB`, `AND`, `OR`, `XOR` based on `op`.
    - Sets `zero_flag` if result is zero.

6. Control Unit (CU)

- **Inputs:** `opcode` (from IR), `zero_flag` (from ALU).

- **Outputs:** Control signals (`PC_inc`, `IR_load`, `ACC_load`, `ALU_op`, `MEM_read`, `OUT_en`, `HALT`).
- **Behavior:**
  - Decodes `opcode` and generates control signals.
  - Example:
    - For `LDA`:
      - `MEM_read = 1` (read from memory).
      - `ACC_load = 1` (load into ACC).
    - For `HLT`:
      - `HALT = 1` (stop execution).

7. Output Register (OUT)

- **Inputs:** `clk`, `enable` (from CU), `data` (from ACC).
- **Outputs:** `display_value` (8-bit result).
- **Behavior:**
  - Latches `ACC` value when `OUT` instruction is executed.

Step 3: Connect Modules

1. **Clock & Reset:**
   a. All sequential modules (`PC`, `ACC`, `IR`, `OUT`) share the same `clk` and `reset`.
2. **PC → Memory → IR:**
   a. `PC` sends `address` to `Memory`.
   b. `Memory` outputs `data` to `IR` when `IR_load` is high.

3. **IR → CU → ALU/ACC/Memory:**
    a. `IR` sends `opcode` to `CU`.
    b. `CU` generates control signals (`ACC_load`, `ALU_op`, etc.).
    c. `ALU` takes inputs from `ACC` and `Memory`.
4. **ACC ↔ ALU ↔ Memory:**
    a. Results from `ALU` are stored back in `ACC`.
    b. `Memory` can be read (`LDA`) or written (if extended).
5. **OUT Display:**
    a. When `OUT_en` is high, `ACC` value is sent to `OUT`.

Step 4: Execution Flow

1. **Fetch:**
    a. `PC` provides address → `Memory` outputs instruction → `IR` loads it.
2. **Decode:**
    a. `CU` decodes `opcode` from `IR` and sets control signals.
3. **Execute:**
    a. For `LDA`:
        i. `Memory` reads data → `ACC` loads it.
    b. For `ADD`:
        i. `ALU` adds `ACC` + `Memory` → Result stored in `ACC`.
    c. For `OUT`:
        i. `ACC` value → `OUT` register.
    d. For `HLT`:

i. Stops execution.

4. **Repeat:**

   a. PC increments (unless JMP/JZ modifies it).

4. Frontend Development

A web-based frontend using **HTML, CSS, and JavaScript can** be used to:

- **Load programs** into SAP-1 memory.
- **Step through execution** (clock cycle control).
- **Display register/memory contents** in real-time.
- **Visualize data flow** between components.

5. Conclusion

- Successfully implemented SAP-1 in **Verilog**.
- Developed a **front-end interface** for user interaction.
- Demonstrated basic **instruction execution** (LDA, ADD, OUT, HLT).