San Jose State University

Computer Engineering Department

CMPE 220 System Software

Software CPU Design

Team 16

Hafeeza Samreen (hafeeza.samreen@sjsu.edu)

Shriya Reddy Aleti (shriyareddy.aleti@sjsu.edu)

Siri Chandana Uppula (sirichandana.uppula@sjsu.edu)

Sravani Linga (sravani.linga@sjsu.edu)

Table of Contents

Sr no	Section	Page no
1	Github Repository	3
2	Architecture Details	4
3	Instructions to Download, Compile and Run	17
4	Team Members Contributions	21

GitHub Repository

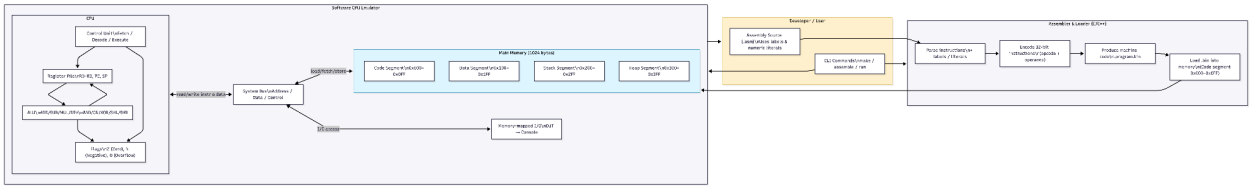
The complete source code, assembler, emulator, instruction set architecture, and demo programs for this CPU Simulator project are available at the following repository:

https://github.com/Hsamreen27/CPU_Simulator

The repository contains:

- Full CPU simulator implementation in C
- Custom Instruction Set Architecture (ISA)
- Assembler with two-pass label resolution
- Memory subsystem with code, data, stack, and heap segments
- Demonstration assembly programs (Timer, Hello World, Fibonacci)
- Build instructions and program execution examples
- Demo video explaining the Fibonacci program

Architecture Details



3.1 High-Level Overview

The CPU Simulator is organized as a small but complete software CPU running on top of C. At the highest level, the system consists of three major parts:

1. **CPU Core** – implements registers, ALU, flags, and the control unit that runs the fetch–decode–execute cycle.

2. **Main Memory and I/O** – a 1024-byte memory divided into code, data, stack, and heap segments, plus a memory-mapped I/O area that behaves like a simple console.

3. **Assembler and Loader** – translates human-readable assembly programs into 32-bit machine code and loads them into the code segment for execution.

Developers interact only with the **assembly source files** and a few **CLI commands** (make, assemble, run). Internally, the assembler and CPU emulator take care of instruction encoding, memory layout, and execution.

3.2 CPU Core

The CPU core lives conceptually inside the left box of the architecture diagram. It is composed of four main elements: the **control unit**, the **register file**, the **ALU**, and the **flags**.

3.2.1 Register File

The **register file** contains four 32-bit general-purpose registers and two special-purpose registers:

- **R0, R1, R2, R3** – used by instructions for arithmetic, logical operations, loop counters, temporary values, function parameters, and return values.

- **PC (Program Counter)** – holds the address of the next instruction to be fetched from memory. The control unit updates PC on every cycle or replaces it when executing jumps and calls.

- **SP (Stack Pointer)** – points to the top of the stack segment in memory and is used by CALL/RET, PUSH/POP, and any stack-based storage.

Keeping the design to four general-purpose registers keeps the ISA simple but still expressive enough to implement loops (Timer), output routines (Hello), and state-dependent algorithms (Fibonacci).

3.2.2 Flags Register

The CPU maintains a small flags register with three condition codes:

- **Z (Zero)** – set when an ALU result is zero.

- **N (Negative)** – set when an ALU result is negative (most-significant bit set).

- **O (Overflow)** – set when a signed arithmetic operation overflows the representable range.

These flags are **updated centrally by the ALU** after every arithmetic or logical operation. Branch instructions such as **JZ** ("jump if zero") and **JNZ** ("jump if not zero") consult these flags to decide whether control flow should change. For example, the Fibonacci loop uses **JNZ** to repeat until a counter reaches zero; the loop termination condition depends entirely on Z being updated correctly.

3.2.3 Arithmetic Logic Unit (ALU)

The **ALU** is responsible for all arithmetic and logical computation. It supports:

- Arithmetic: ADD, SUB, MUL, DIV

- Bitwise logic: AND, OR, XOR, NOT

- Shifts: SHL, SHR

- Comparisons: EQ, NEQ, GT, LT, GE, LE (these primarily update flags)

When the control unit decodes an instruction like ADD R0, R1, R2, it passes the operands to the ALU. The ALU computes the result, writes it back into the destination register, and updates Z, N, and O. A key bug fix in this project was to **always route arithmetic through ALU helper functions** (e.g., alu_add, alu_sub) instead of doing "raw" C arithmetic, so that the flags are always set correctly. Without this, conditional branches in Timer and Fibonacci would not behave as expected.

3.2.4 Control Unit and Fetch–Decode–Execute Cycle

The **control unit** orchestrates everything. Its main loop, implemented in cpu.c, repeatedly executes the following three phases:

1. **Fetch**
    - Read a 32-bit instruction from memory at the address in PC.
    - Store this instruction in a temporary variable for decoding.
2. **Decode**

    Extract the **8-bit opcode** (bits 31–24).
    - Extract the three **8-bit operands** (bits 23–16, 15–8, 7–0).
    - Determine the instruction type (ALU op, memory op, control flow, stack, or I/O).
3. **Execute**

- Invoke the appropriate handler:
  - ALU for arithmetic/logic/shift/compare.
  - Memory subsystem for LOAD/STORE, PUSH/POP.
  - PC/SP updates for jumps, CALL and RET.
  - I/O handler for OUT.
- Update registers, flags, and potentially PC and SP.

Normally, at the end of an instruction the control unit increments PC by 4 to point to the next 32-bit instruction. For **jump-like instructions**, we deliberately **do not** perform this automatic increment if the instruction has already modified PC. Fixing this behavior was another important improvement: previously the emulator incremented PC unconditionally, causing jumps to skip instructions or land at wrong addresses.

The control unit continues this cycle until it encounters the **HALT** instruction, which sets a "halted" flag and stops execution.

3.3 System Bus and Main Memory

Between the CPU core and the rest of the system is a simplified **system bus**. In software this is not a physical bus but a conceptual interface: all instruction fetches, data loads/stores, and I/O accesses go through a small set of memory access functions. This abstraction lets the CPU think in terms of addresses, data, and control signals without knowing exactly how memory is implemented.

3.3.1 Memory Layout

The simulator uses a contiguous **1024-byte main memory**, divided into four segments:

Code Segment (0x000–0x0FF)

Holds the machine code generated by the assembler. When you run

./build/cpu_simulator assemble programs/asm/hello.asm programs/bin/hello.bin

./build/cpu_simulator run programs/bin/hello.bin

- the resulting hello.bin is loaded into this region. PC starts at the beginning of this segment.

- **Data Segment (0x100–0x1FF)**

  Reserved for global and static variables. In the current set of demo programs, most values are held in registers, but this region allows future programs to use memory-based data.

- **Stack Segment (0x200–0x2FF)**

  Used by the **stack pointer (SP)** for pushing and popping values. As is common in many architectures, the stack grows **downward**: SP is initialized near the top of the stack region and moves towards lower addresses on PUSH and back upwards on POP, CALL, and RET.

- **Heap Segment (0x300–0x3FF)**

  Reserved for dynamic allocation. While the current project focuses more on register-based and stack-based behavior, this region allows for extension of the simulator with malloc-style features later.

All memory access goes through functions like read_memory and write_memory. These functions perform **bounds checking** so an instruction cannot access outside the 1024-byte range, avoiding crashes and making debugging easier.

3.4 Memory-Mapped I/O

To keep the I/O model simple and visible, the simulator uses **memory-mapped I/O** for output. Instead of having a separate I/O port instruction, we define:

- An **OUT instruction** that takes a register as operand.

- Internally, OUT sends the value in that register to a pseudo-device that prints to the console (with a prefix like OUT:).

From the program's perspective, this looks like writing to a specific "device address," but in the emulator it simply calls a function that logs output. When we run commands like:

./build/cpu_simulator run programs/bin/fib.bin | grep "OUT:"

we are filtering the console output to show only lines produced by this OUT operation, which makes it very clear what the simulated program is "printing."

The **Hello World** program uses OUT to print ASCII codes corresponding to the characters of "Hello, World". The **Timer** and **Fibonacci** programs use OUT to print numeric values as the counter or sequence progresses.

3.5 Instruction Set Architecture (ISA)

The custom ISA is designed around a **fixed 32-bit instruction format**:

- **Bits 31–24:** 8-bit **opcode**

- **Bits 23–16:** 8-bit operand 1

- **Bits 15–8:** 8-bit operand 2

- **Bits 7–0:** 8-bit operand 3

The meaning of the operands depends on the instruction. For example:

- ADD R0, R1, R2 – opcode selects ADD, operands encode the register indices for destination and sources.

- LOAD R0, 5 – opcode selects LOAD, one operand encodes R0, another operand encodes the immediate value 5.

- JNZ label – opcode selects JNZ, operand encodes the target address (resolved by the assembler using labels).

The ISA includes several groups of instructions:

- **Arithmetic:** ADD, SUB, MUL, DIV

- **Logical:** AND, OR, XOR, NOT

- **Shifts:** SHL, SHR

- **Comparisons:** update flags without necessarily storing a result

- **Memory:** LOAD, STORE

- **Control Flow:** JUMP, JZ, JNZ, CALL, RET

- **Stack Operations:** PUSH, POP

- **I/O:** OUT

- **System:** HALT

Addressing modes are kept **implicit** based on the opcode. For example:

- LOAD treats its second operand as an immediate or memory reference, depending on context.

- ADD and most ALU instructions work in **register-to-register** mode.

This design avoids complicated encoding of addressing modes and keeps decoding logic straightforward.

3.6 Assembler and Program Flow

The right side of your diagram shows the **Assembler & Loader** and the interaction with the developer.

1. **Assembly Source**

   The developer writes assembly files such as hello.asm and fib.asm. These files contain:

   - Instruction mnemonics (ADD, LOAD, JNZ, etc.)
   - Register names (R0, R1, …)
   - Labels (loop:, start:, done:)
   - Numeric literals (e.g., 5, 12, ASCII codes)

Assembler (Two-Pass)

When the user runs:

./build/cpu_simulator assemble programs/asm/fib.asm programs/bin/fib.bin

2. the assembler performs:

   - **First pass:** scans all lines, records label names and their corresponding addresses in the code segment.

- **Second pass:** for each instruction:
  - Parses the mnemonic and operands
  - Looks up label addresses
  - Encodes opcode + operands into the 32-bit binary format
  - Writes the binary words to the .bin file

3. During this stage we also trim whitespace from operands, which fixes the "undefined label" issues that occur when trailing spaces are present.

Loader

When the user then runs:

./build/cpu_simulator run programs/bin/fib.bin

4. the emulator:
   - Clears memory and CPU state

- Loads the binary instructions into the **code segment (0x000–0x0FF)**

- Sets PC to the start of the code segment

- Enters the fetch–decode–execute loop until HALT is reached

5. **Execution Examples**

   - In **hello.asm**, the program consists of a sequence of LOAD and OUT instructions. Each character's ASCII value is loaded into a register and sent to OUT.

   - In **fib.asm**, the program sets up initial values for the Fibonacci sequence in registers, then uses a loop with ADD and SUB to generate new terms. After each term is computed, it calls OUT. A loop counter is decremented; when it reaches zero, Z is set and a JZ or JNZ instruction exits the loop and finally reaches HALT.

This is the level of detail your instructor will expect when they say "architecture details": not just what exists, but **how data and control actually flow through your system** and how that supports your three demo programs.

If you want, I can now **merge this with your objectives, results, and team contributions into one clean report**, or tweak this section to match a specific page/word limit.

Instructions to Download, Compile and Run

1. Clone the GitHub Repository

```
git clone https://github.com/Hsamreen27/CPU_Simulator
cd CPU_Simulator
```

This will download the entire project, including source code, headers, the assembler/emulator, and example programs.

2. Build the Simulator & Assembler

The project uses a Makefile for build automation. From the project root directory, run:

```
make
```

This will compile the source code and produce the executable `cpu_simulator` (or `cpu_simulator.exe` on Windows). After building, you can use `cpu_simulator` both to assemble `.asm` files into `.bin` and to run `.bin` binaries on the simulated CPU.

3. Assembling an Assembly Program

To assemble one of the provided example programs (e.g., Fibonacci or Hello World), use the following command:

```
./build/cpu_simulator assemble programs/asm/<program>.asm programs/bin/<program>.bin
```

```
Binary Instruction: 19000000
Assembly complete: programs/bin/fib.bin
Successfully assembled 'programs/asm/fib.asm' to 'programs/bin/fib.bin'.
```

For example, to assemble the Fibonacci program:

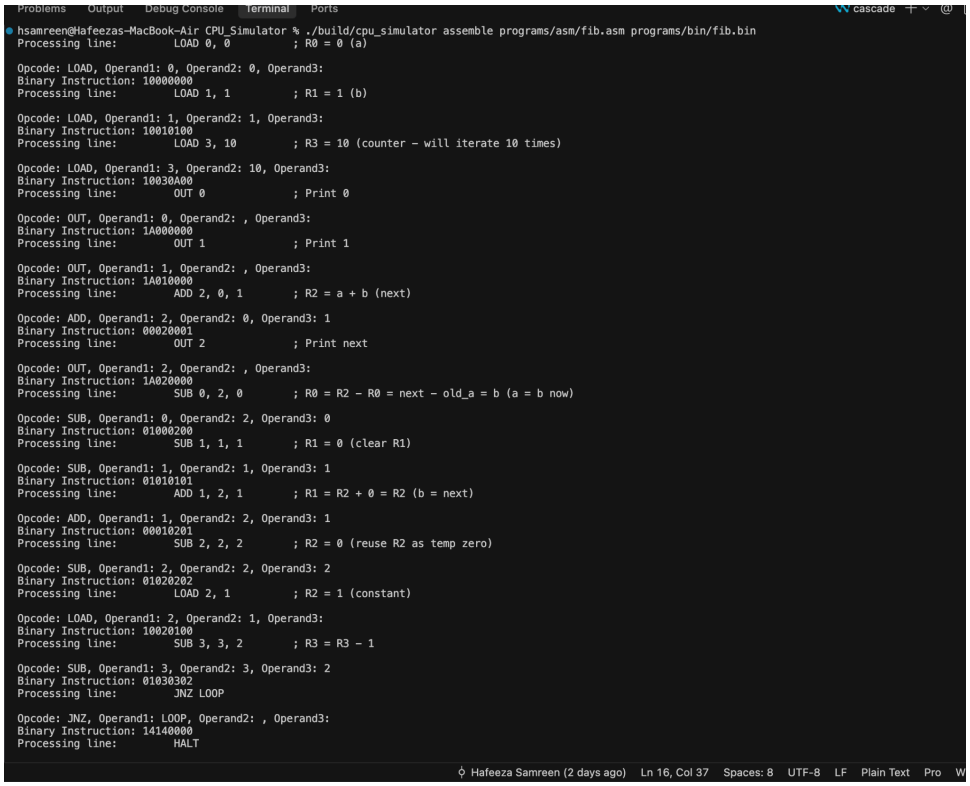
```
./build/cpu_simulator assemble programs/asm/fib.asm programs/bin/fib.bin
```

This will produce `fib.bin` in the `programs/bin/` directory.

4. Running a Compiled Binary

To run the assembled binary on the CPU emulator:

```
./build/cpu_simulator run programs/bin/<program>.bin
```



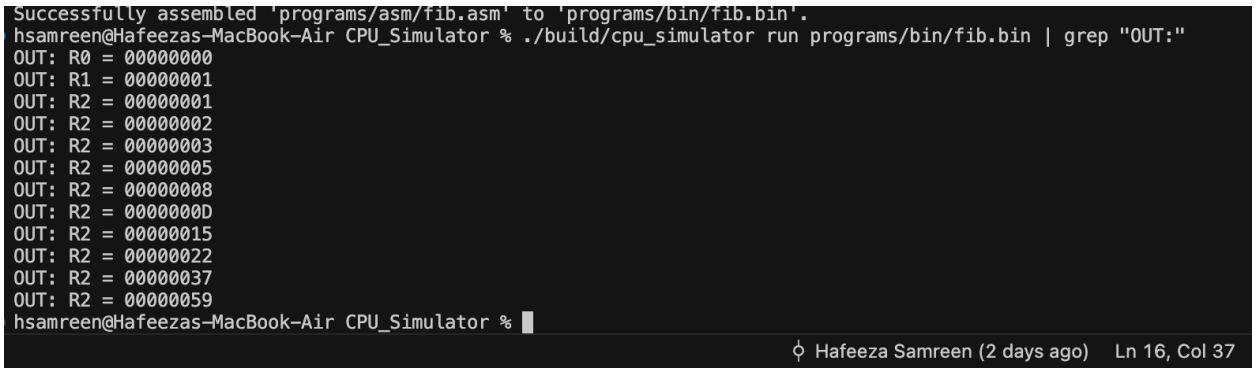
For example:

```
./build/cpu_simulator run programs/bin/fib.bin
```

5. Filtering Output (optional but recommended)

The emulator outputs debug logs and other info; to only view the program's output (e.g., via the `OUT` instruction), you can filter as follows:

```
./build/cpu_simulator run programs/bin/fib.bin | grep "OUT:"
```



Similarly for the Hello World program:

```
./build/cpu_simulator assemble programs/asm/hello.asm
programs/bin/hello.bin \
&& ./build/cpu_simulator run programs/bin/hello.bin | grep
"OUT:"
```

This prints only the lines containing output from the program (e.g., ASCII codes or values printed).

6. Directory Structure Overview

```
CPU_Simulator/
├── include/ # Header files (cpu.h, alu.h, memory.h,
instructions.h, debug.h, linker.h)
├── src/             # Source code files (cpu.c, alu.c,
memory.c, instructions.c, debug.c, linker.c, main.c)
├── programs/
│   ├── asm/         # Assembly source files (timer.asm,
hello.asm, fib.asm)
│   └── bin/         # Assembled binaries (timer.bin,
hello.bin, fib.bin)
├── build/           # Compiled executable (cpu_simulator)
└── Makefile         # Build instructions / automation
```

This modular organization is intended to make the project easy to understand, maintain, and extend — following good documentation & code-organization practices.

Team Members Contributions

We divided the work among four team members to ensure roughly equal distribution of tasks, leveraging each member's strengths. Here is the breakdown:

Hafeeza Samreen (hafeeza.samreen@sjsu.edu)

Role: CPU Core & ALU / Flag & Control-Flow Implementation

- Designed the core CPU components: registers (R0–R3, SP, PC), flags (Z, N, O), and the overall memory model (code, data, stack, heap).

- Implemented the ALU operations (ADD, SUB, MUL, DIV, logic, shift) and ensured they correctly set flags (Zero, Negative, Overflow).

- Integrated ALU with the instruction execution logic, ensuring arithmetic and logical instructions operate correctly.

- Handled branch and control-flow instructions, including correct PC update logic (especially for JUMP, CALL, RET, conditional JZ/JNZ), ensuring proper fetch-decode-execute cycles.

Shriya Reddy Aleti (shriyareddy.aleti@sjsu.edu)

Role: Assembler & Instruction Encoding / Parsing & Label Resolution

- Designed the 32-bit instruction format (8-bit opcode + three 8-bit operands) and defined the full ISA (arithmetic, logical, memory, control flow, stack operations, I/O, HALT).

- Built the two-pass assembler (in linker.c) to parse .asm files, handle labels, numeric literals, whitespace trimming, and operand validation.

- Implemented encoding logic to convert assembly instructions to binary representation compatible with the emulator.

- Ensured assembler handles immediate values, register operands, and memory instructions correctly; tested with example programs (hello.asm, timer.asm, fib.asm).

Siri Chandana Uppula (sirichandana.uppula@sjsu.edu)

Role: Memory Subsystem, Memory-Mapped I/O & Debug / Emulator Infrastructure

- Implemented the memory subsystem (memory.c), dividing memory into segments: code, data, stack, heap (total 1024 bytes).

- Added support for memory operations: LOAD, STORE — enabling data movement between memory and registers.

- Designed and implemented memory-mapped I/O via the OUT instruction to allow programs to output values (e.g., ASCII codes for Hello World, numeric output for Fibonacci, Timer).

- Built debugging utilities (debug.c, debug.h) to trace CPU state, registers, flags, memory dumps — used during development and testing for verification.

Sravani Linga (sravani.linga@sjsu.edu)

Role: Example Programs, Testing, Documentation & Repository Organization

- Wrote the demonstration assembly programs:
    - **timer.asm** — to illustrate fetch–decode–execute cycles and loop / counter behavior
    - **hello.asm** — to output "Hello, World" via ASCII codes using OUT
    - **fib.asm** — to compute the Fibonacci sequence (first 12 numbers) via register-only arithmetic, loops, and conditional branching
- Performed full testing and verification of all programs, ensuring correctness of output (timer counting, ASCII outputs, Fibonacci values).
- Documented the build/run instructions, created README-style instructions, and prepared test commands (assemble + run + output filtering).
- Organized the GitHub repository structure, added comments in code, and ensured consistency across modules for easier comprehension by third parties.