CMPE-220 Sec 01
CPU Design Report

SUBMITTED BY TEAM 09
Bhavika Sodagum (017506567)
Om Bharatbhai Kapadiya (018282355)
Shreya Pudukkottai (017416724)
Parth Gala (018198661)

Date : Nov 29, 2025

1. INTRODUCTION

The primary objective of this project was to design and implement a simplified Central Processing Unit (CPU) using the C programming language. To achieve this, we decomposed the CPU into a collection of modular components, each responsible for a clearly defined function. These foundational modules included the Arithmetic Logic Unit (ALU), the Control Unit, main memory, general-purpose registers, special-purpose registers, and a dedicated flag-handling system associated with the ALU.

By isolating these components into individually maintained building blocks, we ensured that each subsystem could focus on its own responsibilities rather than combining unrelated operations into a single unit. This approach not only enhanced the clarity of the design but also improved code modularity, maintainability, and scalability.

2. METHODOLOGY

We implemented the CPU entirely in C, adopting a structure-driven and modular development strategy. C's struct mechanism served as the foundation for modeling each subsystem, allowing us to encapsulate attributes and behaviors associated with specific CPU functionalities. Throughout development, we intentionally progressed module-by-module, validating each subsystem independently before integrating it into the larger architecture. This incremental testing strategy minimized dependency-related errors and ensured that downstream components received reliable input.

Each conceptual CPU component was represented as an independent C struct, analogous to defining distinct classes in an object-oriented paradigm. This design choice enabled us to embed smaller structs inside larger ones, reflecting the hierarchical nature of an actual processor architecture.

The key modules we structured were:

- ALU (Arithmetic Logic Unit)
- ALUFlags (control and status flags for ALU operations)
- Main Memory (RAM)
- GPRs (General-Purpose Registers)
- SPRs (Special-Purpose Register, later extended for recursion-related behavior)
- Control Unit (CU)
- CPU (top-level struct integrating all components)

By following this layered design pattern, we achieved a clean separation of responsibilities while maintaining a coherent representation of a functioning CPU.

2.1 INSTRUCTION SET ARCHITECTURE (ISA) Specification

To demonstrate a functioning fetch–decode–execute cycle, we developed a compact instruction set tailored to basic arithmetic, logical, and control-flow operations. These instructions provided the CPU with the essential capabilities expected from a minimal yet effective processor model.

TABLE 1: Supported Instructions and Corresponding Opcodes

Instruction	**Opcode**
NOP	0000
MOV	0001
ADD	0010
SUB	0011
AND	0100
OR	0101
MUL	0110
DIV	0111
JMP	1000
JZ	1001
CALL	1010
RET	1011
HALT	1100

After finalizing the instruction set, we designed the control and status flags required for ALU operations. These flags dictated how the ALU interpreted inputs, which operations it performed, and how it produced and classified results. Prior to computation, the Control Unit configured the ALU’s control flags, effectively guiding the ALU on how to treat inputs and outputs.

TABLE 2: Overview of ALU Control Flags

Control Flag	**Function**
zx	Zero-out input x
nx	Negate input x
zy	Zero-out input y
ny	Negate input y
f	Select arithmetic vs. logical operation
no	Negate ALU output

We also introduced a complementary set of status flags, housed within the same ALUFlags struct to maintain organizational simplicity. These indicators were updated only after an ALU computation completed and were used to represent the characteristics of the computed output.

TABLE 3: Summary of ALU Status Flags

Status Flag	**Meaning**
zr	Output is zero
ng	Output is negative
ov	Signed overflow occurred during addition
cy	Unsigned carry-out from addition

3. CODE

Github repository link : https://github.com/shreyapbk0622/CMPE220_Project.git

4. ASSEMBLY LANGUAGE DESIGN

For the assembly-level layer of our CPU, we implemented the instruction set previously outlined in Table 1. To test how these instructions would operate within the emulator, we constructed a small demonstration program inside the main function. This sample program relied on a helper function, encodeI, which converts each assembly instruction—along with its operands—into a 16-bit machine-level representation.

The encodeI function receives four parameters: the opcode (mapped through an enumerated index), destination register r1, source register r2, and an offset or immediate field. The function then assigns each parameter to specific bit positions within the 16-bit instruction format, ensuring that the final output conforms to the CPU's defined encoding scheme. Once encoded, all instructions are stored sequentially in main memory, forming the executable portion of the program. These encoded instructions are then retrieved during the CPU's fetch stage.

In subsequent stages, particularly during decode, each 16-bit instruction is broken apart to recover the operation type, register operands, and immediate values. This reconstructed information is then used during execution.

Instruction Cycle: Fetch, Decode, Execute, Store

Fetch Stage

The fetch phase is implemented through a helper function named fetch. During this step, the Control Unit loads the first instruction of the program—stored in main memory—into the Instruction Register (IR). After loading the instruction, the Instruction Pointer (IP) is automatically advanced to reference the next instruction in memory.

Decode Stage

The decode phase is driven by interpreting the contents of the IR. Using bit shifting and masking operations, we extract the opcode, register operands (r1 and r2), and any immediate value embedded within the 16-bit instruction. Each field occupies designated bit positions, and decoding restores the original structure of the assembly instruction.

Execute + Store Phases

Execution and storage were coupled into a unified process within the fetchDecodeExecute method. Using a structured sequence of conditional statements, the Control Unit determines the currently decoded operation and configures the ALU's inputs accordingly. Operands x and y are assigned based on the source registers, and all necessary control flags are initialized.

Once configured, the alu_compute function is invoked. Since it receives a reference to the CPU, it can directly access the Control Unit's current flag configuration. Based on the flags and selected operation, the ALU performs the appropriate arithmetic or logical computation.

After computation, the result is written back to the appropriate general-purpose register (typically r1). Simultaneously, the ALUFlags structure is updated to reflect the status of the latest operation—zero, negative, overflow, or carry conditions.

5. MEMORY LAYOUT

The memory model of our CPU emulator is divided into three logical regions, each serving a dedicated purpose.

TABLE 3: Segmented Memory Layout for the Virtual CPU

Segment	**Purpose**	**Address Range**	**Example**
Code Segment	Holds program instructions at program start	0x0000 – 0x0010	Binary code
Data Segment	Static and global variable storage (reserved)	0x0100 – 0x01FF	Constants
Stack Segment	Dynamic stack for function calls and recursion (downward growth)	0x01FF – 0x0000	Return addrs

Recursion Handling

Our CPU emulator is capable of demonstrating recursive execution, accomplished through a small recursive program. The recursive mechanism behaves similarly to standard architectures by maintaining a call stack and unwinding it correctly via return instructions.

A typical recursive flow in our emulator proceeds as follows:

1. A MOV instruction initializes the counter (e.g., R0 = 3).
2. A CALL instruction invokes the recursive routine.
3. Inside the recursive function:
    - The counter is decremented via SUB R0, 1.
    - The Zero flag is evaluated using JZ to determine whether the base case is reached.
    - If the counter is not zero, another CALL is issued, deepening the recursion.
4. Upon meeting the base condition, successive RET instructions unwind the stack frame-by-frame.

Each CALL pushes the return address onto the stack, while each RET pulls the last address off, restoring execution to the appropriate point.

TABLE 4: Illustration of Stack Behavior During Recursion

Step	**Stack Contents (Top to Bottom)**	**SP Value**
Initial	—	399
After CALL 1	0x0001	398
After CALL 2	0x0004, 0x0001	397
After CALL 3	0x0006, 0x0004, 0x0001	396
After RET from Level 3	0x0004, 0x0001	397
After RET from Level 2	0x0001	398
After RET from Level 1	empty	399

Stack Mechanics
The stack is fundamental to recursion in our CPU architecture. It is used to preserve the return address after every CALL instruction and restore it when a RET occurs.

- Before any CALL occurs: SP = 399
- After each call: SP decreases by 1
- The top of the stack stores the current IP, serving as the return address
- During a return:
  - The SP increases
  - The return address is loaded back into the IP

This mechanism ensures that recursive and nested function calls execute and unwind properly.

6. Register and Flag Behavior During Program Execution

To verify the correctness of our CPU's execution pipeline, we observed the state of all general-purpose registers, the stack pointer, and the ALU status flags after each instruction. The table below summarizes the evolving machine state throughout a sample run.

Table 5: Register States and Flag Outputs Across Execution

Instruction	**Register Snapshot (R0–R7, SP)**	**Flags (Z, N, OV, CY)**
NOP	R0=0, R1=0, R2=0, R3=0, R4=0, R5=0, R6=0, SP=399	Z=0, N=0, OV=0, CY=0

MOV R0, 10	R0=10, R1=0, R2=0, R3=0, R4=0, R5=0, R6=0, SP=399	Z=0, N=0, OV=0, CY=0
MOV R1, 5	R0=10, R1=5, R2=0, R3=0, R4=0, R5=0, R6=0, SP=399	Z=0, N=0, OV=0, CY=0
ADD R2, 3	R0=10, R1=5, R2=3, R3=0, R4=0, R5=0, R6=0, SP=399	Z=1, N=0, OV=0, CY=0
MOV R3, 12	R0=0, R1=5, R2=3, R3=12, R4=0, R5=0, R6=0, SP=399	Z=1, N=0, OV=0, CY=0
MOV R4, 9	R0=0, R1=5, R2=3, R3=12, R4=9, R5=0, R6=0, SP=399	Z=1, N=0, OV=0, CY=0
AND R3, R4	R0=0, R1=5, R2=3, R3=8, R4=9, R5=0, R6=0, SP=399	Z=0, N=0, OV=0, CY=0
OR R4, R3	R0=0, R1=5, R2=3, R3=8, R4=9, R5=0, R6=0, SP=399	Z=0, N=0, OV=0, CY=0
MUL R1, R0	R0=0, R1=0, R2=3, R3=8, R4=9, R5=0, R6=0, SP=399	Z=1, N=0, OV=0, CY=0
MOV R5, 36	R0=0, R1=0, R2=3, R3=8, R4=9, R5=36, R6=0, SP=399	Z=1, N=0, OV=0, CY=0
MOV R6, 5	R0=0, R1=0, R2=3, R3=8, R4=9, R5=36, R6=5, SP=399	Z=1, N=0, OV=0, CY=0
DIV R5, R6	R0=0, R1=0, R2=3, R3=8, R4=9, R5=7, R6=5, SP=399	Z=0, N=0, OV=0, CY=0
JMP 15	State unchanged	—
CALL 20	R0=0, R1=0, R2=3, R3=8, R4=9, R5=7, R6=0, SP=398	—
MOV R7, 42	R7=42	—
RET	Returned to caller	—
HALT	Program terminated	—

TABLE 6:

No.	Objective	Instruction Sequence	Explanation
1	Recursive factorial of 3	<pre> MOV R0, 3 CALL 3 HALT (At address 3:) SUB R0, R0, 1 JZ 9 MUL R0, R1 RET (At address 9:) MOV R0, 1 RET</pre>	- Start with R0 = 3 - If R0 == 0, return 1 - Otherwise, recursively multiply
2	Sum of numbers from 1 to 5	MOV R0, 5 CALL 3 HALT (At address 3:) JZ 9 MOV R1, R0 SUB R0, R0, 1 CALL 3 ADD R0, R1 RET (At address 9:) Z=0, N=0, OV=0, CY=0 Z=1, N=0, OV=0, CY=0 Z=1, N=0, OV=0, CY=0 Z=1, N=0, OV=0, CY=0 Z=1, N=0, OV=0, CY=0 MOV R0, 0 RET	– Sum 5 + 4 + 3 + 2 + 1 – Uses recursion
3	Simple function call-return without recursion	<pre> MOV R0, 5 CALL 3 HALT (At address 3:) ADD R0, R0, 0 RET</pre>	- - Call a function at address 3 - Do simple operation and return

4	Double a number using CALL	<pre> MOV R0, 4 CALL 3 HALT (3:) ADD R0, R0, R0 RET</pre>	Doubles the value of R0
5	Nested CALL inside CALL	<pre> MOV R0, 5 CALL 3 HALT (3:) CALL 6 ADD R0, R1 RET (6:) MOV R1, 10 RET</pre>	Function at 3 calls another function at 6
6	Multi-level Recursion (Countdown)	<pre> MOV R0, 3 CALL 3 HALT (3:) JZ 8 SUB R0, R0, 1 CALL 3 RET (8:) RET</pre>	Calls itself until R0 reaches 0

7	Swap two registers using CALL	<pre> MOV R0, 5 MOV R1, 10 CALL 5 HALT (5:) MOV R2, R0 MOV R0, R1 MOV R1, R2 RET</pre>	Swaps values between R0 and R1
8	Return immediately (no-op CALL)	<pre> CALL 2 HALT (2:) RET</pre>	CALL immediately returns without doing anything
9	Call different function based on condition	<pre> MOV R0, 0 JZ 6 CALL 3 HALT (3:) MOV R1, 10 RET (6:) MOV R1, 20 RET</pre>	Conditional CALL to different logic
10	Recursive multiplication by addition	<pre> MOV R0, 3 MOV R1, 4 CALL 5 HALT (5:) JZ 9 SUB R0, R0, 1 CALL 5	Computes 3×4 = 12 using repeated addition recursively

7. CONCLUSION

This project required us to think deeply about CPU architecture and collaborate effectively as a team. By dividing responsibilities across well-defined phases, we ensured that each subsystem—from the ALU to recursion support—worked cohesively within the broader virtual CPU. The modular structure provided clarity while enabling parallel development.

In the future, we aim to extend our CPU with more sophisticated instructions and enhanced memory management capabilities, allowing it to support a wider range of computational tasks.

8. TEAM CONTRIBUTIONS

Parth Gala
Contributed to ISA design, struct-level modularity, and initial arithmetic operations. He also assisted with the fetch stage, debugging, and report preparation.

Bhavika Sodagum
Developed the recursion system, analyzed stack behavior and memory layout, and worked on the execution phase. She also contributed heavily to testing and documentation.

Om Bharatbhai Kapadiya
Resolved major bugs, implemented full CALL/RET recursion, refined stack pointer behavior, and validated HALT/IP interaction. Verified execution across all CPU stages and contributed to the Store phase and report.

Shreya Pudukkottai
Implemented ALU flag logic, developed multi-bit arithmetic operations, and worked on decode/execute integration. Also supported testing and report development.

We used Chat-gpt occasionally during the development process to brainstorm implementation approaches and clarify technical concepts. This support helped us refine certain ideas while ensuring that all design decisions and final code were understood and written by the team.