San Jose State University

Computer Engineering Department

CMPE 220 - System Software

Program Layout & Execution

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)

Instructor: Ishie Eshwar

Semester: Fall 2025

San José State University

One Washington Square, San José, CA 95192, USA

Table of Contents

GitHub Repository and Project Overview

1. GitHub Repository

The complete source code for the CPU Simulator and the recursive C program used in this project is available at:

GitHub Repository

https://github.com/Hsamreen27/CPU_Simulator

This repository contains:

- CPU core implementation (registers, PC, SP, flags and instruction cycle).
- Memory module with code, data, stack, and heap segments.
- Assembler to convert `.asm` files into binary `.bin` files.
- Sample programs, including the recursive program used for this assignment.
- Video Recordings

2. Project Overview

This project demonstrates **program layout and execution** for a **recursive C function** running on our custom CPU simulator designed in the earlier CMPE 220 project. We focus on:

- How the compiled program is **laid out in memory** (code, data, stack, heap).
- How the CPU handles **function calls** using the stack.

- How **recursion** is implemented through repeated function calls and returns.

The CPU architecture and memory segmentation follow the same design as our earlier “CPU Emulator Design” project (4 general-purpose registers R0–R3, PC, SP, flags, and a 1024-byte memory divided into logical segments).

Program Layout, Recursion, and Instructions to Download/Run

3. Recursive C Program and CPU Execution

We use a simple **recursive factorial function** written in C:

```c
// factorial.c

int factorial(int n) {

    if (n <= 1) {

        return 1;

    }

    return n * factorial(n - 1);

}

int main() {

    int n = 4;

    int result = factorial(n);

    // result is printed or stored in a register / memory location

    return 0;

}
```

This C program is translated/compiled into assembly compatible with our CPU simulator, and then assembled into a binary (`factorial.bin`) that the simulator can execute.

```
hsamreen@Hafeezas-MacBook-Air CPU_Simulator_Project_220 % gcc programs/c/factorial.c -o factorial_test
./factorial_test
=== Factorial Calculation Demo ===
Calculating factorial of 5

  Computing: factorial(5) = 5 * factorial(4)
  Computing: factorial(4) = 4 * factorial(3)
  Computing: factorial(3) = 3 * factorial(2)
  Computing: factorial(2) = 2 * factorial(1)
  Base case reached: factorial(1) = 1
  Returning: factorial(2) = 2
  Returning: factorial(3) = 6
  Returning: factorial(4) = 24
  Returning: factorial(5) = 120

Final Result: 5! = 120
Expected: 5! = 5 * 4 * 3 * 2 * 1 = 120
hsamreen@Hafeezas-MacBook-Air CPU_Simulator_Project_220 %
```

3.1 Memory Layout of the Executable

Our simulator uses a **flat 1024-byte memory**, logically divided into four segments similar to the reference CPU project:

- **Code Segment (0x000 – 0x0FF):**

  Contains machine instructions for `main`, `factorial`, and runtime support code.

  - Example contents:
    - Instructions for `main` (loading `n`, calling `factorial`, storing result).
    - Instructions for `factorial` (base case check, multiply, recursive CALL).
- **Data Segment (0x100 – 0x1FF):**

  Holds global/static variables if used. For our simple program, `n` and `result` can be:

  - Stored as local variables in stack frames, or
  - Mapped to fixed locations in the data segment (depending on compiler/translator).

```
PC: 00000008 SP: 00000300 Flags: Z=1 N=0 O=0

Executing instruction at PC: 0000000C
Executing instruction: Opcode=19 Operands=0, 0, 0
HALT instruction executed. Stopping CPU.

Updated Memory Segments:

Memory Layout:
Code Segment (Instructions):
0x00000000: 00 05 00 10 00 00 10 15 00 00 01 1A 00 00 00 19
0x00000010: 00 01 02 10 02 00 02 01 00 00 38 13 00 00 00 17
0x00000020: 00 01 02 10 02 00 00 01 00 00 10 15 00 00 00 18
0x00000030: 01 00 01 02 00 00 00 16 00 01 01 10 00 00 00 16
0x00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000000A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000000B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000000C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000000D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000000E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000000F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Data Segment (Global/Static Variables):
0x00000100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000140: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000150: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000160: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000170: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000190: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000001A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000001B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000001C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000001D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000001E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000001F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

- **Stack Segment (0x200 – 0x2FF):**

  Used to store **stack frames** for:

- ○ Return addresses.
- ○ Saved registers.
- ○ Function parameters.
- ○ Local variables (`n` and temporary values).

- The stack grows **downward** from higher addresses toward lower addresses. The **Stack Pointer (SP)** always points to the top of the current stack frame.
- **Heap Segment (0x300 – 0x3FF):**

  Reserved for dynamic allocation (not used in this simple factorial example).

```
Stack Segment:
0x00000200: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000220: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000230: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000240: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000250: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000260: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000270: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000280: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000290: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000002A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000002B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000002C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000002D0: 00 00 00 00 00 00 00 00 00 00 00 00 2C 00 00 00
0x000002E0: 02 00 00 00 2C 00 00 00 03 00 00 00 2C 00 00 00
0x000002F0: 04 00 00 00 2C 00 00 00 05 00 00 00 08 00 00 00

Heap Segment:

Final CPU state:
R0: 00000005
R1: 00000078
R2: 00000000
R3: 00000000
PC: 0000000C SP: 00000300 Flags: Z=1 N=0 O=0
hsamreen@Hafeezas-MacBook-Air CPU_Simulator_Project_220 % ./build/cpu_simulator run programs/bin/factorial.bin | grep "OUT:"

OUT: R1 = 00000078
hsamreen@Hafeezas-MacBook-Air CPU_Simulator_Project_220 %
```

3.2 Function Call Handling (CALL / RET)

Our CPU supports **CALL** and **RET** instructions to implement function calls:

- **CALL factorial** (from `main`):
  1. The CPU pushes the **return address** (the address of the instruction after CALL) onto the stack.

2. It may push argument(s) (e.g., the value of `n`) or store them in registers (e.g., R0).
3. The **Program Counter (PC)** is updated to the starting address of `factorial`.
4. The **Stack Pointer (SP)** is decremented to reflect the new frame.

- **RET** (from `factorial` back to caller):
  1. The function places the **return value** (e.g., in R0 or a known memory location).
  2. The CPU pops the saved **return address** from the stack.
  3. The PC is set to this return address, and SP is incremented to discard the current frame.
  4. Execution continues in the caller (`main` or the previous `factorial` call).

A typical stack frame for `factorial(n)` looks like:

- Return address to caller.
- Saved registers (e.g., old R0–R3 if needed).
- Parameter `n`.
- Space for local variables / temporaries.

3.3 How Recursion is Carried Out

For **factorial(4)**, the CPU executes the following high-level sequence:

1. **main frame created**
   - `n = 4` is stored (in a register or memory).

- `CALL factorial` with argument `4`.

2. **factorial(4) frame pushed**
   - Stack now has a frame for `factorial(4)`.
   - CPU checks `n <= 1` (false).
   - It prepares `n - 1 = 3` and executes `CALL factorial` again.

3. **factorial(3) frame pushed**
   - New frame for `factorial(3)` with its own parameter `n = 3`.
   - Again, base case is false → `CALL factorial(2)`.

4. **factorial(2) frame pushed**
   - Frame for `factorial(2)`.
   - Base case still false → `CALL factorial(1)`.

5. **factorial(1) frame pushed (base case)**
   - Base case `n <= 1` is true.
   - Returns `1`. Return value stored in a register or on the stack.
   - `RET` pops the frame and jumps back to `factorial(2)`'s return address.

6. **Unwinding the recursion**
   - In `factorial(2)` frame: result = `2 * 1 = 2`; return to caller.

- In `factorial(3)` frame: result = `3 * 2 = 6`; return to caller.
- In `factorial(4)` frame: result = `4 * 6 = 24`; return to `main`.

7. **Back in main**
   - Final result `24` is stored in a register or memory.
   - The program may output the result via an `OUT` instruction or simply halt after storing it.

At each step, our simulator can display:

- **Current PC and SP values.**
- **Register state** (R0–R3) after each instruction.
- **Stack contents**, showing multiple frames for `factorial(4)`, `factorial(3)`, `factorial(2)`, `factorial(1)` during the deepest point of recursion.

This makes recursion and stack frame behavior very clear.

4. How to Download, Compile, and Run the Program

4.1 Prerequisites

- A Linux environment (or WSL on Windows / macOS terminal).
- `gcc` (or any required C compiler) installed.
- `make` installed.

- `git` installed.

4.2 Download the Repository

Clone the repository

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

Move into the project directory

```
cd CPU_Simulator
```

4.3 Build the CPU Simulator

Most of the build is automated using the provided Makefile:

Build the CPU simulator

```
make
```

The built binary will typically be in ./build/

```
ls build/
```

You should see an executable such as `cpu_simulator` (name may vary slightly depending on your repo).

4.4 Add the Recursive Program

1. Place your C file (e.g., `factorial.c`) into the appropriate folder (for example, `programs/c/` if you have one).
2. Use your existing pipeline to translate C → assembly → binary. Depending on your project setup, this could be:
   - A custom script that converts `factorial.c` to `factorial.asm`, or
   - Writing `factorial.asm` manually and assembling it.

Assuming you have `factorial.asm` ready in `programs/asm/`:

Assemble factorial.asm into factorial.bin

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

You can use the same style as your existing examples (like `fib.asm` and `hello.asm`), just adjusting filenames.

4.5 Run the Recursive Program in the Simulator

Run the binary on the CPU simulator

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

If your simulator supports filtered output (like using `OUT` instructions and piping to `grep "OUT:"`), you can do:

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

This will show only the output lines printed by your program.

Team Member Contributions

This project was completed collaboratively by four team members. The work was divided to keep the effort balanced across **CPU architecture understanding**, **program design**, **memory/stack analysis**, and **documentation**.

Hafeeza Samreen

- Integrated the recursive C program (`factorial`) with the existing CPU simulator infrastructure.
- Helped define how the program's code and data map into the code and data segments of the simulator's memory.
- Verified correctness of function call behavior (CALL/RET) and recursive execution by inspecting stack frames and register changes.
- Contributed to writing sections on program layout, function calls, and recursion.
- Handled reproducibility in other machines.

Shriya Reddy Aleti

- Reviewed the CPU and memory architecture from the earlier project and ensured consistency in this report's description (registers, PC, SP, flags, memory segmentation).
- Helped design the step-by-step explanation of how each recursive call pushes a new stack frame and how the stack unwinds on return.

- Assisted in generating and validating test runs of the factorial program on the simulator, including checking intermediate debug output.

Siri Chandana Uppula

- Focused on the **user workflow**: how to **download, compile, assemble, and run** the recursive program using the provided Makefile and simulator binaries.
- Wrote and refined the “How to Download, Compile, and Run” section, including terminal commands and usage notes.
- Ensured that instructions are clear enough for another student to reproduce the results on their own machine.

Sravani Linga

- Led the preparation of the **final report document**, including title page, GitHub page, and the overall structure so that it matches CMPE 220 expectations (double spacing, margins, page numbers).
- Edited and polished the explanations of memory layout, function calls, and recursion to keep them concise and understandable.
- Coordinated team review to ensure that each member’s contribution is reflected and that the report aligns with the assignment rubric.