CMPE 220 - System Software

Program Layout & Execution Final Report

Dec 6, 2025

Team 15

Abhinav Sriharsha Anumanchi

San Jose State University

One Washington Sq, San Jose, CA 95192, USA

abhinavsriharsha.anumanchi@sjsu.edu

Sahithi Chikkela

San Jose State University

One Washington Sq, San Jose, CA 95192, USA

sahithi.chikkela@sjsu.edu

Vineela Mukkamala

San Jose State University

One Washington Sq, San Jose, CA 95192, USA

vineela.mukkamala@sjsu.edu

Vinuta Patil

San Jose State University

One Washington Sq, San Jose, CA 95192, USA

vinuta.patil@sjsu.edu

GITHUB REPOSITORY :

https://github.com/sahithichikkela/CMPE-220-Program-Layout-Execution

The GitHub repository contains all project deliverables and supporting materials:

```
├── cpu.c, cpu.h              # CPU core & architecture
├── assembler.c, assembler.h  # Two-pass assembler
├── main.c                    # Emulator driver
├── factorial_recursive.asm   # Recursive factorial
├── multiply_recursive.asm    # Recursive multiplication
├── factorial.c               # C reference implementation
├── multiply.c                # C reference implementation
├── Makefile
├── video.mp4                 # Recursion explanation video
└── README.md
```

Instructions to access:

1. Navigate to the URL above

2. Click "Code" button

3. Download ZIP or clone using: git clone: https://github.com/sahithichikkela/CMPE-220-Program-Layout-Execution

STEPS TO DOWNLOAD, COMPILE, AND RUN

PREREQUISITES:

- GCC compiler (any recent version)

- Make utility

- macOS, Linux, or Unix-like system

STEP 1: DOWNLOAD THE PROJECT

From GitHub clone the repository using:

git clone **https://github.com/sahithichikkela/CMPE-220-Program-Layout-Execution**

Or :

1. Navigate to the repository URL

2. Click the green "Code" button

3. Select "Download ZIP"

4. Extract the ZIP file to your desired location

STEP 2: COMPILE THE PROJECT

1. Open terminal and navigate to the project directory:

cd "CMPE-220-Program-Layout-Execution"

2. Cleanup using Make clean:

make clean

3. Compile using Make:

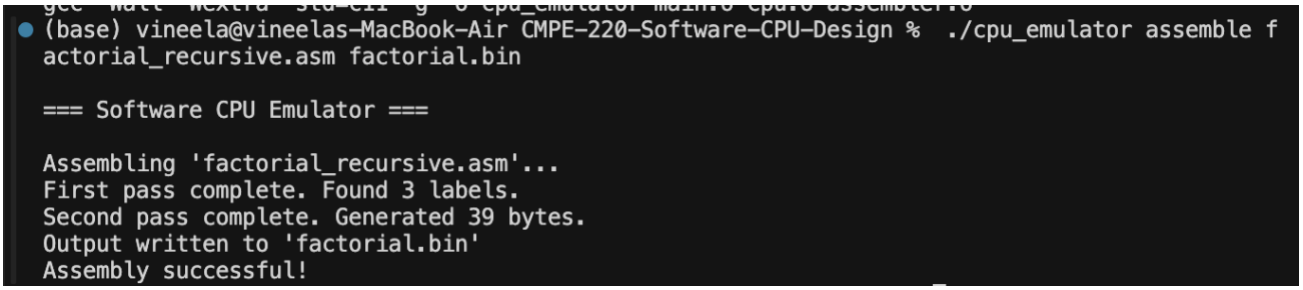
make

STEP 3: RUN THE PROGRAM

Assemble factorial program:

./cpu_emulator assemble factorial_recursive.asm factorial.bin

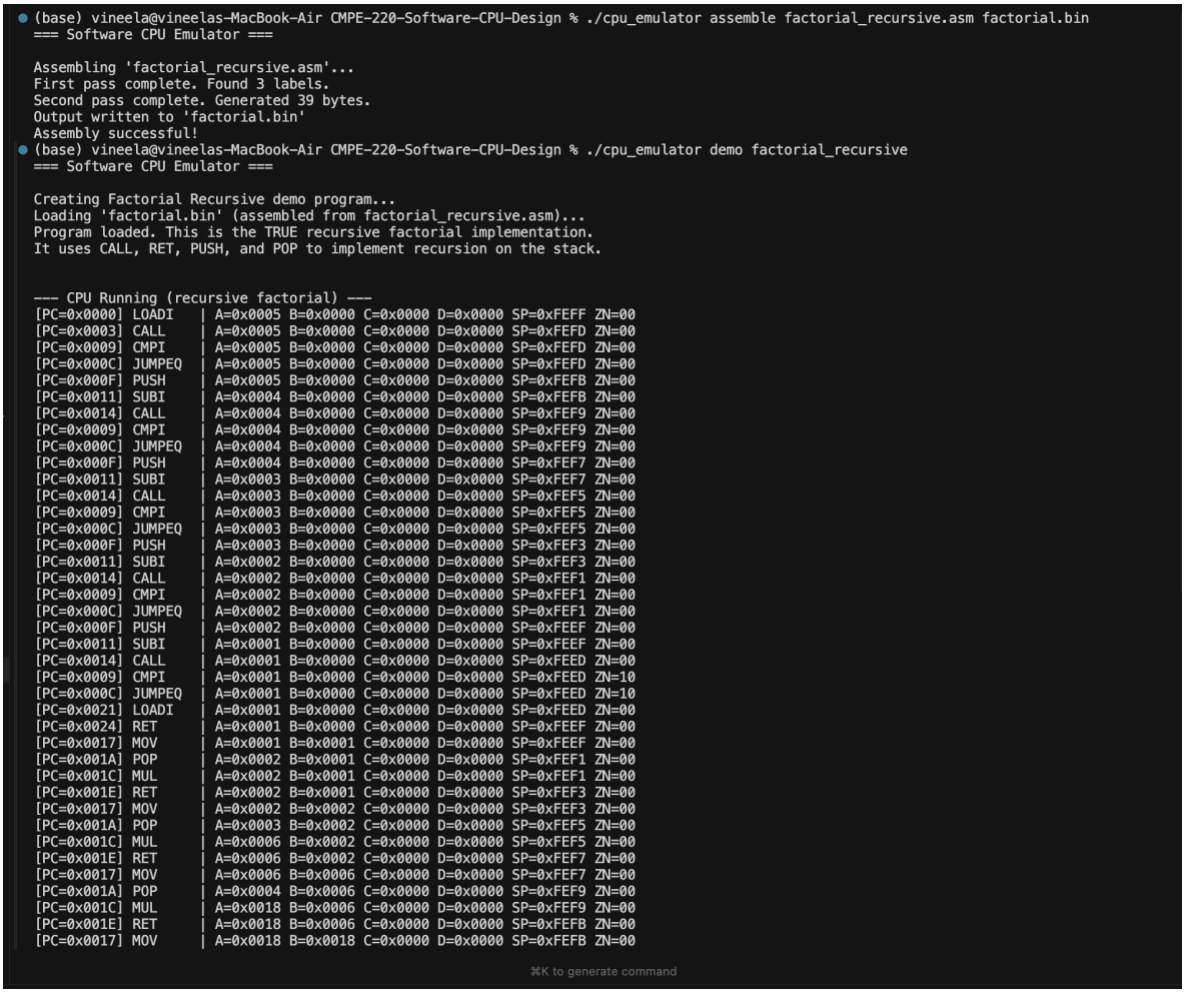
Output:



Factorial program in demo mode:

./cpu_emulator demo factorial_recursive

Output:



```
[PC=0x001A] POP     | A=0x0004 B=0x0006 C=0x0000 D=0x0000 SP=0xFEF9 ZN=00
[PC=0x001C] MUL     | A=0x0018 B=0x0006 C=0x0000 D=0x0000 SP=0xFEF9 ZN=00
[PC=0x001E] RET     | A=0x0018 B=0x0006 C=0x0000 D=0x0000 SP=0xFEFB ZN=00
[PC=0x0017] MOV     | A=0x0018 B=0x0018 C=0x0000 D=0x0000 SP=0xFEFB ZN=00
[PC=0x001A] POP     | A=0x0005 B=0x0018 C=0x0000 D=0x0000 SP=0xFEFD ZN=00
[PC=0x001C] MUL     | A=0x0078 B=0x0018 C=0x0000 D=0x0000 SP=0xFEFD ZN=00
[PC=0x001E] RET     | A=0x0078 B=0x0018 C=0x0000 D=0x0000 SP=0xFEFF ZN=00

[CPU HALTED after 43 cycles]
[PC=0x0006] HALT    | A=0x0078 B=0x0018 C=0x0000 D=0x0000 SP=0xFEFF ZN=00

[CPU HALTED after 43 cycles]

--- CPU Halted ---

=== CPU Registers ===
PC: 0x0007   SP: 0xFEFF
A:  0x0078   B:  0x0018
C:  0x0000   D:  0x0000
FLAGS: 0x80 [H]
Cycles: 43

Recursive factorial calculated!
factorial(5) result in register A: 120 (expected: 120)
Result in hexadecimal: 0x0078

--- Memory Dump (0000 - 003F [Hex]) ---
Addr | 00 01 02 03 04 05 06 07 | ASCII
-------------------------------------------------
0000 | 04 05 00 64 09 00 6C 00 | ...d..l.
0008 | 00 48 01 00 54 21 00 12 | .H..T!..
0010 | 00 1C 01 00 64 09 00 0E | ....d...
0018 | 00 01 16 00 2A 01 68 00 | ....*.h.
0020 | 00 04 01 00 68 00 00 00 | ....h...
0028 | 00 00 00 00 00 00 00 00 | ........
0030 | 00 00 00 00 00 00 00 00 | ........
0038 | 00 00 00 00 00 00 00 00 | ........
-------------------------------------------------
(base) vineela@vineelas-MacBook-Air CMPE-220-Software-CPU-Design %
```

Run binary

./cpu_emulator run factorial.bin

Output:

```
(base) vineela@vineelas-MacBook-Air CMPE-220-Software-CPU-Design % ./cpu_emulator run factorial.bin
=== Software CPU Emulator ===

Running program 'factorial.bin' (39 bytes)...


[CPU HALTED after 43 cycles]


=== CPU Registers ===
PC: 0x0007   SP: 0xFEFF
A:  0x0078   B:  0x0018
C:  0x0000   D:  0x0000
FLAGS: 0x80 [H]
Cycles: 43
(base) vineela@vineelas-MacBook-Air CMPE-220-Software-CPU-Design %
```

RUN RECURSIVE MULTIPLY PROGRAM:

Assemble:

./cpu_emulator assemble multiply_recursive.asm multiply.bin

Output:

```
(base) vineela@vineelas-MacBook-Air CMPE-220-Software-CPU-Design % ./cpu_emulator assemble mu
ltiply_recursive.asm multiply.bin

=== Software CPU Emulator ===

Assembling 'multiply_recursive.asm'...
First pass complete. Found 3 labels.
Second pass complete. Generated 47 bytes.
Output written to 'multiply.bin'
Assembly successful!
```

Run Mode:

./cpu_emulator run multiply.bin

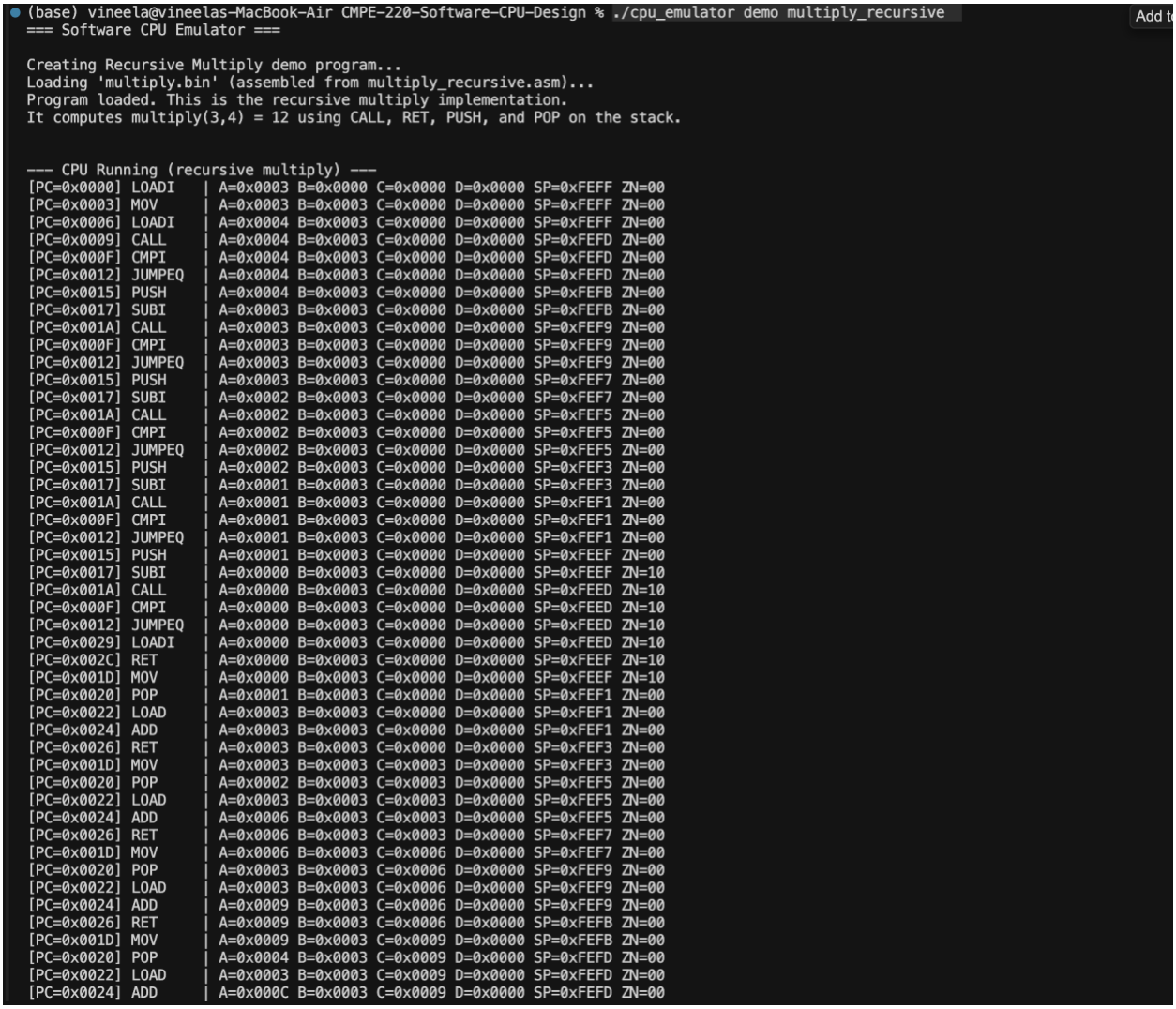
Output:

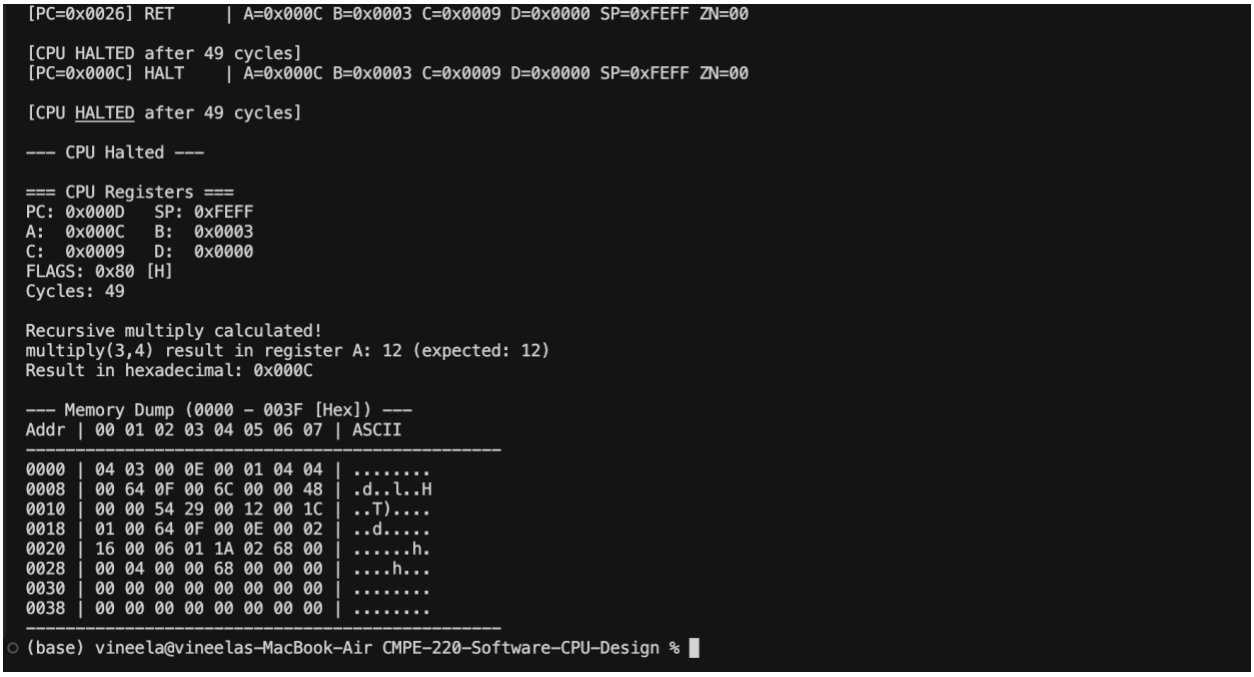
```
(base) vineela@vineelas-MacBook-Air CMPE-220-Software-CPU-Design % ./cpu_emulator run multiply.bin
=== Software CPU Emulator ===

Running program 'multiply.bin' (47 bytes)...


[CPU HALTED after 49 cycles]


=== CPU Registers ===
PC: 0x000D   SP: 0xFEFF
A:  0x000C   B:  0x0003
C:  0x0009   D:  0x0000
FLAGS: 0x80 [H]
Cycles: 49
(base) vineela@vineelas-MacBook-Air CMPE-220-Software-CPU-Design %
```

Multiply Program in demo mode:

./cpu_emulator demo multiply_recursive

Output:




[PC=0x0026] RET | A=0x000C B=0x0003 C=0x0009 D=0x0000 SP=0xFEFF ZN=00
[CPU HALTED after 49 cycles]
[PC=0x000C] HALT | A=0x000C B=0x0003 C=0x0009 D=0x0000 SP=0xFEFF ZN=00
[CPU HALTED after 49 cycles]
--- CPU Halted ---
=== CPU Registers ===
PC: 0x000D SP: 0xFEFF
A: 0x000C B: 0x0003
C: 0x0009 D: 0x0000
FLAGS: 0x80 [H]
Cycles: 49
Recursive multiply calculated!
multiply(3,4) result in register A: 12 (expected: 12)
Result in hexadecimal: 0x000C
--- Memory Dump (0000 - 003F [Hex]) ---
Addr | 00 01 02 03 04 05 06 07 | ASCII
0000 | 04 03 00 0E 00 01 04 04 | ....
0008 | 00 64 0F 00 6C 00 00 48 | .d..l..H
0010 | 00 00 54 29 00 12 00 1C | ..T)....
0018 | 01 00 64 0F 00 0E 00 02 | ..d.....
0020 | 16 00 06 01 1A 02 68 00 | ......h.
0028 | 00 04 00 00 68 00 00 00 | ....h...
0030 | 00 00 00 00 00 00 00 00 | ....
0038 | 00 00 00 00 00 00 00 00 | ....
(base) vineela@vineelas-MacBook-Air CMPE-220-Software-CPU-Design %

RUN EQUIVALENT C PROGRAMS

Compile & run factorial in C

gcc factorial.c -o factorial

./factorial

Output:

```
(base) vineela@vineelas-MacBook-Air CMPE-220-Software-CPU-Design %  gcc factorial.c -o factorial
  ./factorial

=== Recursive Factorial Calculator ===

factorial(0) = 1
factorial(1) = 1
factorial(3) = 6
factorial(5) = 120

=== Detailed Example: factorial(5) ===
Call Stack Visualization:

Step 1: main() calls factorial(5)
  Stack: [main] -> [factorial(5)]

Step 2: factorial(5) calls factorial(4)
  Stack: [main] -> [factorial(5)] -> [factorial(4)]

Step 3: factorial(4) calls factorial(3)
  Stack: [main] -> [factorial(5)] -> [factorial(4)] -> [factorial(3)]

Step 4: factorial(3) calls factorial(2)
  Stack: [main] -> [factorial(5)] -> [factorial(4)] -> [factorial(3)] -> [factorial(2)]

Step 5: factorial(2) calls factorial(1)
  Stack: [main] -> [factorial(5)] -> [factorial(4)] -> [factorial(3)] -> [factorial(2)] -> [factorial(1)]

Step 6: factorial(1) returns 1 (base case)
  Stack: [main] -> [factorial(5)] -> [factorial(4)] -> [factorial(3)] -> [factorial(2)]
  factorial(2) computes: 2 * 1 = 2

Step 7: factorial(2) returns 2
  Stack: [main] -> [factorial(5)] -> [factorial(4)] -> [factorial(3)]
  factorial(3) computes: 3 * 2 = 6

Step 8: factorial(3) returns 6
  Stack: [main] -> [factorial(5)] -> [factorial(4)]
  factorial(4) computes: 4 * 6 = 24

Step 9: factorial(4) returns 24
  Stack: [main] -> [factorial(5)]
  factorial(5) computes: 5 * 24 = 120

Step 10: factorial(5) returns 120
  Stack: [main]
  Final result: 120
```

Compile & run multiplication in C

gcc multiply.c -o multiply

./multiply

Output:

```
(base) vineela@vineelas-MacBook-Air CMPE-220-Software-CPU-Design % gcc multiply.c -o multiply
  ./multiply

=== Recursive Multiplication ===

Test Cases:
multiply(3, 4) = 12 (expected: 12)
multiply(5, 6) = 30 (expected: 30)
multiply(7, 2) = 14 (expected: 14)
multiply(10, 0) = 0 (expected: 0)

=== Detailed Example: multiply(3, 4) ===
Computing: 3 * 4 using recursion

Call Sequence:
1. main() calls multiply(3, 4)
   Stack: [main] -> [multiply(3,4)]

2. multiply(3, 4) calls multiply(3, 3)
   Stack: [main] -> [multiply(3,4)] -> [multiply(3,3)]

3. multiply(3, 3) calls multiply(3, 2)
   Stack: [main] -> [multiply(3,4)] -> [multiply(3,3)] -> [multiply(3,2)]

4. multiply(3, 2) calls multiply(3, 1)
   Stack: [main] -> [multiply(3,4)] -> [multiply(3,3)] -> [multiply(3,2)] -> [multiply(3,1)]

5. multiply(3, 1) calls multiply(3, 0)
   Stack: [main] -> [multiply(3,4)] -> [multiply(3,3)] -> [multiply(3,2)] -> [multiply(3,1)] -> [multiply(3,0)]

Return Sequence:
6. multiply(3, 0) returns 0 (base case)
   Stack: [main] -> [multiply(3,4)] -> [multiply(3,3)] -> [multiply(3,2)] -> [multiply(3,1)]
   multiply(3,1) computes: 3 + 0 = 3

7. multiply(3, 1) returns 3
   Stack: [main] -> [multiply(3,4)] -> [multiply(3,3)] -> [multiply(3,2)]
   multiply(3,2) computes: 3 + 3 = 6

8. multiply(3, 2) returns 6
   Stack: [main] -> [multiply(3,4)] -> [multiply(3,3)]
   multiply(3,3) computes: 3 + 6 = 9

9. multiply(3, 3) returns 9
   Stack: [main] -> [multiply(3,4)]
   multiply(3,4) computes: 3 + 9 = 12

10. multiply(3, 4) returns 12
    Stack: [main]
    Final result: 12
```

TEAM MEMBER CONTRIBUTIONS

Vinuta Patil

Role: CPU Architecture & ISA Support for Recursion

Contributions:

- Verified correct encoding and operation of control-flow instructions
- Defined the calling convention used by recursive programs
- Documented stack frame layout and operand passing for recursion
- Validated ISA behavior for nested function calls
- Base recursive program ("factorial_recursive.asm") for factorial N, demonstrating CALL/RET and stack behavior.

Sahithi Chikkela:

Role: Recursive Programs, Testing & Documentation

Contributions:

- Implemented equivalent C programs for correctness testing
- Authored and refined recursive assembly programs
- Compared assembly recursion outputs against C reference outputs
- Debugged recursion behavior (operand passing, SP movement, RET unwinding)
- Verified ALU operations required for recursion (ADD, SUB, CMP, MUL)
- Supported documentation and README instructions for recursion demos
- Implements ("muliply_recursive.asm") `multiply(a, b) = a + multiply(a, b−1)`, `validating recursive descent & unwind.`

Vineela Mukkamala

Role: Memory Management, Stack Operations & Video

Contributions:

- Implemented and validated stack pointer movement during recursive calls
- Ensured correct memory allocation for stack frames during recursive descent
- Verified CALL/RET operation cycles and memory safety
- Integrated recursion demos into the emulator (demo factorial_recursive , demo multiply_recursive)
- Created and edited the recursion execution demo video
- C reference implementation used to validate the assembly factorial recursion ("factorial.c").

Abhinav Sriharsha Anumanchi

Role: CPU Core Behavior & ALU Operations

Contributions:

- Implemented C reference code for multiply recursion
- Validated correctness between emulator outputs and C outputs
- Performed cycle-by-cycle analysis of recursion trace (PC, SP, CALL/RET behavior)
- Documented recursion flow, stack frames, and emulator outputs
- Prepared explanation materials for report and presentation
- C version of multiply recursion ("multiply.c") used to test correctness of multiply_recursive.asm .