CSE312 – Semester Project

Muhammet Talha Memişoğlu

210104004009

Introduction

This project aims to simulate a simplified computing system by implementing a custom CPU and operating system from scratch. Using a custom instruction set (GTU-C312), we developed:

- A C++ CPU simulator that executes 17 unique instructions.
- An assembly-based OS that supports cooperative multitasking through a thread table and round-robin scheduling.
- Several user-level threads including sorting and searching routines.
- Debugging modes to monitor memory and thread execution.

Through this project, we gained hands-on experience in CPU simulation, OS design, and low-level systems programming. The following sections describe the architecture, implementation details, and results.

CPU Simulation

The CPU component forms the core of our system and is responsible for executing instructions and switching between user and kernel modes. It simulates a simplified processor that interprets our custom GTU-C312 instruction set within a virtualized environment.

Memory
The simulator uses two main memory structures:

- **DataMemory:** A vector stores data values.
- **InstructionMemory:** A vector of Instruction objects storing the parsed instruction set.

Program Loading
The loadProgram() function parses input files and populates data and instructions for each thread. It isolates each thread's memory space using thread-based offsets.

Instruction Execution
The execute() method fetches the next instruction using the program counter (PC_REG), checks for validity, and calls executeInstruction() to perform the operation. It also handles debug modes for tracing memory and thread states. Additionally, it checks whether the instruction has permission to access the specified address. If the CPU is in user mode and the instruction attempts to access kernel space, the thread is

terminated. The thread is also restricted to accessing only its own memory space, and this is verified as well.

Instruction Set

Instruction	Type	Explanation	Example
SET B A	Direct Set	Sets the memory at address A with value B.	SET -20, 100 → Memory[100] = -20
CPY A1 A2	Direct Copy	Copies the contents of memory at A1 into A2.	CPY 100, 120 → Memory[120] = Memory[100]
CPYI A1 A2	Indirect Copy	Interprets A1 as a pointer. Copies content at address stored in A1 into A2.	CPYI 100, 120 → if Memory[100] = 200 → Memory[120] = Memory[200]
CPYI2 A1 A2	Double Indirect Copy	Copies value from address pointed by A1 to address pointed by A2.	CPYI2 100, 120 → if Memory[100] = 200 & Memory[120] = 300 → Memory[300] = Memory[200]
ADD A B	Arithmetic	Adds value B to the content at memory location A.	ADD 100, 5 → Memory[100] += 5
ADDI A1 A2	Indirect Add	Adds the value at address A2 to the value at address A1.	ADDI 100, 120 → Memory[100] += Memory[120]
SUBI A1 A2	Indirect Subtract	Subtracts value at A2 from value at A1, result stored in A2.	SUBI 100, 120 → Memory[120] = Memory[100] - Memory[120]
JIF A C	Conditional Jump	If value at memory A ≤ 0, jump to instruction at C (change program counter).	JIF 50, 200
PUSH A	Stack Operation	Pushes the value at address A onto the stack. Stack grows upward.	PUSH 100
POP A	Stack Operation	Pops the top value from the stack into memory address A.	POP 120
CALL C	Subroutine Call	Jumps to instruction C and pushes return address onto the stack.	CALL 300
RET	Subroutine Return	Pops the return address from the	RET

		stack and jumps there.	
HLT	CPU Control	Halts the CPU execution.	HLT
USER A	Mode Switch	Switches to user mode and jumps to the address stored at location A.	USER 150
SYSCALL PRN A	System Call	Prints the value at memory A followed by newline. Blocks thread for 100 instructions.	SYSCALL PRN 100
SYSCALL HLT	System Call	Terminates the thread execution.	SYSCALL HLT
SYSCALL YIELD	System Call	Yields the CPU to allow the OS to schedule other threads.	SYSCALL YIELD

Instruction Decoding

The parseInstruction() method converts textual instructions into Instruction objects, resolving memory addresses in thread-local contexts and handling syntax validation.

System Calls

SYSCALL instructions (e.g., PRN, HLT, YIELD) are handled by jumping to fixed OS handler locations. The simulator supports kernel transitions to manage these interactions securely.

Debugging Support

A configurable debugLevel controls how much information is printed, ranging from full memory dumps to step-by-step thread context displays.

Debug Level	Description
0	Show memory state **only once** at the end of execution.
1	Print full memory state after **every instruction**.
2	Same as Level 1 but **pauses** after each instruction for manual inspection.
3	Shows thread **context switch only**—prints thread ID, start time, instruction count, state, and registers when threads change.

(If the instruction memory is null, it is not printed. Similarly, if the data memory is zero, it is also omitted from the output.)

Operating System

The operating system in this project is implemented using the GTU-C312 instruction set and resides entirely within the instruction memory of the simulator. It manages thread creation, scheduling, system call handling, and thread context switching in a cooperative multitasking environment.

Thread Table Initialization

- The OS initializes 11 thread tables: 1 for the OS and 10 for user threads.
- Each thread table consists of 25 memory cells, including:
    - Thread ID
    - Start time
    - Instruction count
    - State (0: Halted, 1: Ready, 2: Running, 3: Blocked)
    - 21 registers for thread context

These tables are stored in memory starting from address 30 and separated by offsets of 25.

System Call Handling

The OS handles three system calls:

System Call	Description
SYSCALL PRN	Prints a value to the console
SYSCALL HLT	Halts the current thread
SYSCALL YIELD	Yields execution to the next ready thread

Each system call triggers a context switch to the OS using predefined jump points (e.g., address 80 for HLT, 86 for YIELD, 90 for PRN).

A portion of the PRN system call is implemented in C++, as mentioned in the requirements document.

After each YIELD and HALT operation, a message indicating which thread performed the operation is printed. Additionally, after the PRN system call, the ID of the calling thread is printed for informational purposes

Context Switching

When a system call (HLT, YIELD, or PRN) is triggered, control transfers to the OS using fixed addresses (e.g., 80, 86, 90). The OS then performs the following steps:

1. Saves the current thread's context (registers, state, instruction count) into its thread table.
2. Loads its own context from the OS thread table (thread 0) to continue executing safely.
3. Before switching to the next thread, the OS also saves its own registers back into its thread table, just like any thread, to preserve state for future execution.
4. Selects the next ready thread using a round-robin strategy, skipping any that are halted or blocked.
5. Loads the next thread's context and marks it as Running.
6. Transfers control to the new thread via the USER instruction.

If no threads remain, the OS restores its own context again and halts the system.

User Threads

The project includes multiple user-level threads that demonstrate various computational tasks.

Thread 1 – Summation

- Calculates the sum of integers from 10 down to 1.
- Uses a loop with ADDI and ADD instructions.
- Contains SYSCALL YIELD to voluntarily give up control during the loop.
- After the loop ends, prints the result using SYSCALL PRN and halts with SYSCALL HLT.

Thread 2 – Linear Search

- Searches for a key (14) in an array of 5 elements. (You can change key value and number of elements)

- Implements a loop using indirect memory access to compare values.
- Uses SYSCALL YIELD to simulate thread cooperation during search iterations.
- If the key is found, prints the index using SYSCALL PRN and halts with SYSCALL HLT.

Thread 3 – Bubble Sort

- Sorts an array of 5 integers using a bubble sort algorithm.
- Uses nested loops, indirect memory access, and swapping logic.
- Yields execution with SYSCALL YIELD after each outer loop pass.
- Once sorted, prints each element using SYSCALL PRN, then halts with SYSCALL HLT.

Threads 4 to 10 – Empty Tasks

- Each of these threads contains only a single SYSCALL HLT instruction.
- They terminate immediately when scheduled.

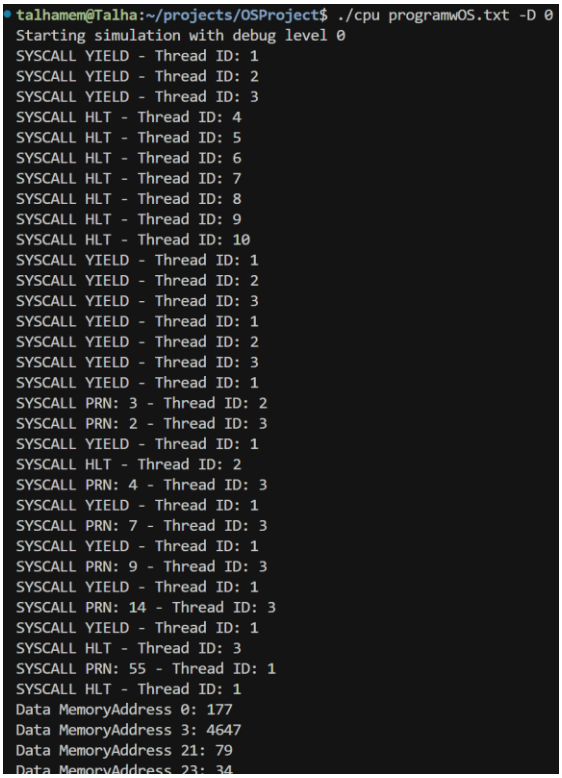
Usage of Program

./cpu programwOS.txt -D <debugmode>

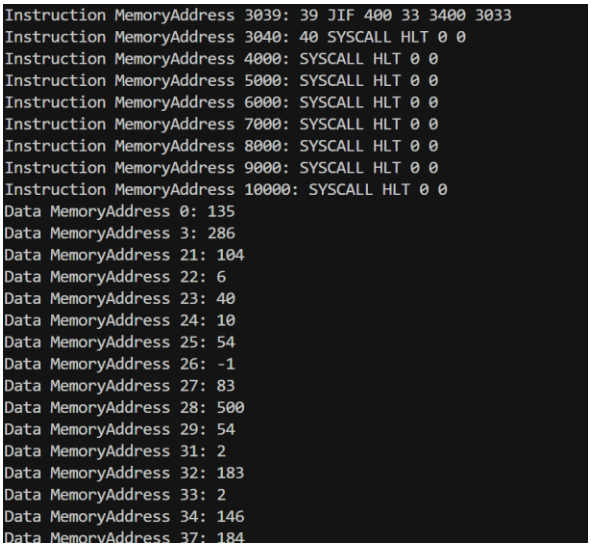
Screenshots of Program

(If the instruction memory is null, it is not printed. Similarly, if the data memory is zero, it is also omitted from the output in order to track easily.)

Debug Mode 0



Debug mode 1



Debug Mode 2

```
Instruction MemoryAddress 3007: 7 CPYI 101 107 3101 3107
Instruction MemoryAddress 3008: 8 SUBI 106 107 3106 3107
Instruction MemoryAddress 3009: 9 JIF 107 14 3107 3014
Instruction MemoryAddress 3010: 10 CPYI 101 106 3101 3106
Instruction MemoryAddress 3011: 11 CPYI2 100 101 3100 3101
Instruction MemoryAddress 3012: 12 SET 3106 107 3106 3107
Instruction MemoryAddress 3013: 13 CPYI2 107 100 3107 3100
Instruction MemoryAddress 3014: 14 ADD 100 1 3100 1
Instruction MemoryAddress 3015: 15 ADD 101 1 3101 1
Instruction MemoryAddress 3016: 16 CPY 100 106 3100 3106
Instruction MemoryAddress 3017: 17 SUBI 102 106 3102 3106
Instruction MemoryAddress 3018: 18 JIF 106 20 3106 3020
Instruction MemoryAddress 3019: 19 JIF 400 6 3400 3006
Instruction MemoryAddress 3020: 20 CPY 25 100 3025 3100
Instruction MemoryAddress 3021: 21 CPY 100 101 3100 3101
Instruction MemoryAddress 3022: 22 ADD 101 1 3101 1
Instruction MemoryAddress 3023: 23 ADD 102 -1 3102 -1
Instruction MemoryAddress 3024: 24 CPY 100 106 3100 3106
Instruction MemoryAddress 3025: 25 SUBI 102 106 3102 3106
Instruction MemoryAddress 3026: 26 JIF 106 29 3106 3029
Instruction MemoryAddress 3027: 27 SYSCALL YIELD 0 0
Instruction MemoryAddress 3028: 28 JIF 400 6 3400 3006
Instruction MemoryAddress 3029: 29 CPY 25 106 3025 3106
Instruction MemoryAddress 3030: 30 CPY 25 107 3025 3107
Instruction MemoryAddress 3031: 31 ADDI 24 107 3024 3107
Instruction MemoryAddress 3032: 32 ADD 107 -1 3107 -1
Instruction MemoryAddress 3033: 33 CPYI 106 108 3106 3108
Instruction MemoryAddress 3034: 34 SYSCALL PRN 108 3108 0
Instruction MemoryAddress 3035: 35 CPY 106 109 3106 3109
Instruction MemoryAddress 3036: 36 SUBI 107 109 3107 3109
Instruction MemoryAddress 3037: 37 JIF 109 40 3109 3040
Instruction MemoryAddress 3038: 38 ADD 106 1 3106 1
Instruction MemoryAddress 3039: 39 JIF 400 33 3400 3033
Instruction MemoryAddress 3040: 40 SYSCALL HLT 0 0
Instruction MemoryAddress 4000: SYSCALL HLT 0 0
Instruction MemoryAddress 5000: SYSCALL HLT 0 0
Instruction MemoryAddress 6000: SYSCALL HLT 0 0
Instruction MemoryAddress 7000: SYSCALL HLT 0 0
Instruction MemoryAddress 8000: SYSCALL HLT 0 0
Instruction MemoryAddress 9000: SYSCALL HLT 0 0
Instruction MemoryAddress 10000: SYSCALL HLT 0 0
Press Enter to continue...
```

Debug Mode 3

```
Register[ 5]      :                       0
Register[ 6]      :                       0
Register[ 7]      :                       0
Register[ 8]      :                       0
Register[ 9]      :                       0
Register[10]      :                       0
Register[11]      :                       0
Register[12]      :                       0
Register[13]      :                       0
Register[14]      :                       0
Register[15]      :                       0
Register[16]      :                       0
Register[17]      :                       0
Register[18]      :                       0
Register[19]      :                       0
Register[20]      :                       0
========================================

SYSCALL HLT - Thread ID: 1

========================================
         Thread 0 Context Table
========================================
Thread ID         :                       0
Starting Time     :                       2
Instruction Count :                    4541
State             :            2 (Running)
----------------------------------------
Register[ 0]      :                     146
Register[ 1]      :                       0
Register[ 2]      :                       0
Register[ 3]      :                    4542
Register[ 4]      :                       0
Register[ 5]      :                       0
Register[ 6]      :                       0
Register[ 7]      :                       0
Register[ 8]      :                       0
Register[ 9]      :                       0
Register[10]      :                       0
Register[11]      :                       0
Register[12]      :                       0
Register[13]      :                       0
Register[14]      :                       0
Register[15]      :                       0
Register[16]      :                       0
Register[17]      :                       0
Register[18]      :                       0
Register[19]      :                       0
Register[20]      :                       0
========================================

Simulation complete!
```

Use of AI Assistance

During the development of this project, I made limited use of AI tools such as ChatGPT and Claude to seek guidance on CPU design decisions and to help structure parts of the simulator logic. These tools were helpful for clarifying general ideas, creating roadmaps, discussing design strategies - particularly for the CPU simulation components- and preparing the report.

However, due to the highly customized nature of the GTU-C312 architecture and its unique instruction set, AI models were unable to generate valid GTU-C312 code. As a result, I had to write the entire OS logic and all user-level threads manually, which proved to be the most challenging part of the project. Debugging and designing the round-robin scheduler, implementing system calls, and managing thread context switching required extensive effort and deep understanding.

While I also used Cursor (an AI-powered code editor) during development, it is not possible to export the suggestions or corrections it provided. Nevertheless, all critical implementation work—especially the OS routines and thread programs—was designed and written entirely by me.

In the end, despite limited external help, I successfully completed the project and met all its requirements.

Links of chats between me and AI Tools:

https://chatgpt.com/share/6841b016-aed0-8008-a426-e6cbbd859acb *(I used that chat a lot)*

https://chatgpt.com/share/6841b1ec-5040-8008-9e7d-67d13aacdd8d (for preparing report)

https://claude.ai/share/a1d21fe9-d39f-4612-a6e4-4e4a54c88423 *(It was for only gaining opinion. In fact, it is not complete and right anyway.)*

https://claude.ai/share/65fea789-070d-4090-9c11-d6a7e415adab *(It was for only gaining opinion. In fact, it is not complete and right anyway.)*

Finally, there is no way to export cursor chats as I mentioned above so I couldn’t share cursor chats.

Conclusion

This project was a challenging but rewarding journey through low-level system design. While AI tools like ChatGPT and Claude helped me with planning and design decisions, they were not capable of generating valid GTU-C312 assembly code due to the custom nature of the instruction set.

As a result, writing the OS and thread programs from scratch was the most difficult part of the project. I spent significant time understanding the architecture, debugging instruction behavior, and carefully implementing every part of the OS logic manually.

Despite these challenges, I successfully built a working CPU simulator, designed a functional operating system, and implemented multiple user threads—all meeting the project’s requirements. This experience deepened my understanding of CPU execution, thread management, and operating system design.