

AKDENIZ UNIVERSITY

Faculty of Engineering – Computer Engineering Dep.

CSE 206 – Computer Organization Project

**CPU Emulator Final Project Documentation**

**Student Information**

**Name:** Cem Erdemir  
**University:** Akdeniz University  
**Faculty:** Faculty of Engineering  
**Department:** Computer Engineering  
**Course:** Computer Organization (CSE206)  
**Instructor:** Taner Danışman

**Project Overview**

This project simulates a simple CPU emulator capable of executing a set of machine instructions. The system includes memory and a cache subsystem, and it supports basic arithmetic, control flow, and memory operations. The emulator reads binary instructions from program.txt, processes them, and logs execution details, such as changes to the accumulator (AC), program counter (PC), and flag register.

The purpose of the assignment is to gain insight into how CPUs interpret and execute instructions, interact with memory hierarchies like cache, and manage control flow.

**Class Explanations**

**1. CpuEmulator**

This is the main class responsible for simulating the CPU execution.

**Key Variables:**

* AC (Accumulator): Stores results of arithmetic operations.
* PC (Program Counter): Points to the current instruction.
* FLAG: Holds the result of comparisons (negative: -1, zero: 0, positive: 1).
* memory[]: The main memory.
* cache: Instance of a cache class used to simulate memory access speedup.

**Key Methods:**

* execute(): Fetches, decodes, and executes instructions using a switch-case based on opcode.
* readMemWord(int address): Reads a word from memory via the cache system.
* writeMemWord(int address, int value): Writes a word to memory and updates the cache.
* printStats(): Prints final statistics including cache hit rate and final AC value.

**2. Cache**

The Cache class is designed to simulate a direct-mapped cache mechanism. It aims to reduce the access time to memory by keeping recently accessed data.

**Key Features:**

* Tracks cache hits and misses.
* Handles reading and writing from/to memory.

**Methods:**

* read(int address, byte[] memory): Reads a byte from the cache or memory.
* write(int address, byte value, byte[] memory): Writes a byte to the cache and memory.
* Internal logic updates hit and miss counters.

**3. Memory**

The Memory class simulates the main memory as a byte array with support for byte and word operations. It provides a safe and low-level interface for interacting with memory, making it a vital part of the emulator.

**Key Methods:**

* readByte(int address): Returns a byte from the specified memory address. The address is wrapped using 0xFFFF to prevent overflow, simulating 16-bit address space.
* writeByte(int address, byte value): Writes a byte to the specified address, again applying the wrap-around technique.
* readWord(int address): Reads a two-byte word from memory (little-endian). It reads two consecutive bytes and combines them into a 16-bit value.
* writeWord(int address, int value): Writes a two-byte word to memory. It splits the integer into two bytes (low and high) and stores them.

These methods allow the emulator to safely read and write both individual bytes and complete 16-bit words from memory, an essential feature for accurately simulating CPU operations.

**4. Main**

This class is used to initialize and start the emulator by loading the program, setting the base address, and invoking the execute() method.

**Main Responsibilities:**

* Load the program.txt into memory.
* Set initial values for PC and base address.
* Display overall execution outcome.

**Functionality and Instruction Set**

The emulator supports the following opcodes:

| **Opcode** | **Operation** | **Description** |
| --- | --- | --- |
| 0x0 | START | Begin execution |
| 0x1 | LOAD | Load immediate value into AC |
| 0x2 | LOADM | Load memory content into AC |
| 0x3 | STORE | Store AC value into memory |
| 0x4 | CMPM | Compare AC with memory value, set FLAG |
| 0x5 | CJMP | Conditional jump if FLAG > 0 |
| 0x6 | JMP | Unconditional jump |
| 0x7 | ADD | Add immediate value to AC |
| 0x8 | ADDM | Add value from memory to AC |
| 0xD | DISP | Display AC value |
| 0xE | HALT | Terminate the program execution |

Each instruction is 16 bits long: 4 bits opcode + 12 bits operand.

**Why program.txt Was Modified**

Initially, the instruction sequence in program.txt led to an infinite loop and incorrect accumulator values (e.g., 1030 instead of 210). This occurred due to a logic flaw in the program flow, particularly in the loop and memory write operations.

Upon inspection, it was discovered that the memory addresses used for limit, counter, and sum were too close to each other (e.g., 0x0C8, 0x0C9, 0x0CA). These addresses mapped to the same cache block, which caused frequent cache conflicts. As a result, stored values were being overwritten unintentionally during execution.

**Modifications Made:**

To prevent cache collisions and ensure accurate computation, these memory addresses were updated:

| **Variable** | **Old Address** | **New Address** |
| --- | --- | --- |
| limit | 0x0C8 | 0x0100 |
| counter | 0x0C9 | 0x0102 |
| sum | 0x0CA | 0x0104 |

By aligning each variable to a separate and even memory address, each one maps to a distinct cache block. This significantly reduced cache interference and ensured memory integrity throughout the loop.

These adjustments corrected the behavior, resulting in the expected AC value of **210** and a cache hit ratio of **68.96%**

**Conclusion**

This project provided hands-on experience in CPU simulation, instruction decoding, control flow, and memory hierarchy via cache implementation. By building this emulator, we understood:

* How assembly-level instructions are processed.
* The significance of memory alignment and address translation.
* The impact of cache on performance.

All functionalities were tested and verified with sample input, and the final output matched expected results. The revised program.txt ensured correct termination and cumulative arithmetic execution.