Project 1 Documentation

Brysen Pfingsten, Nathaniel Savoury, David Anthony Fields

September 22, 2025

Contents

1 Outline

1.1 Problem Statement

In this project, you will develop a Computer System Simulator that simulates key components of a basic computer system. The purpose of this assignment is to help you understand the fundamental concepts of computer systems (from chapters 1 and 2). You will simulate components such as the CPU, memory, interrupt handling, and Direct Memory Access (DMA).

The goal is to design and simulate a fully functioning computer system by implementing all the components listed below. Your project 1 will cover five modules, each focusing on the simulation of specific system components.

- Module 1: Central Processing Unit
- Module 2: Memory Management System
- Module 3: Instructions Set Architecture
- Module 4: Interrupts
- Module 5: Direct Memory Access

1.2 Introduction

To complete the tasked of simulating a computer system, we elected to us the programming language, `C`, to implement this project. This documentation is structured as follows, the implementation for modules 1 through 5 can be found in Section 2, Section 3, Section 4, Section 5 and, Section 6 respectively. The testing and debugging is found in Section 7. Finally, the simulation of the described computer system is be found in Section 8.

2 Module 1: Central Processing Unit

2.1 Problem Statement

Implement the core components of the CPU including the Program Counter (`PC`), Accumulator (`ACC`), and Instruction Register (`IR`). The CPU will follow the fetch-decode-execute cycle, fetching instructions from memory, decoding them, and performing operations.

Simulate the `PC`, which stores the address of the next instruction to be executed. Implement the `ACC` for performing arithmetic and logic operations, and the IR to hold the current instruction. The Status Register will manage flags like zero, carry, and overflow. The CPU must execute a fetch-decode-execute cycle, where it fetches an instruction from memory, decodes the opcode, and executes it, updating the register values. For instance, the CPU might fetch an `ADD` instruction, decode it, and then update the `ACC` with the result. Implement error handling for invalid instructions, and ensure the `PC` updates correctly after each cycle. Your program execution should simulate real-time instruction processing and register updates, handling valid instructions and interrupts efficiently. It must handle instructions such as `ADD`, `SUB`, `LOAD`, and `STORE`. Also, ensure that the `PC` is updated correctly after each instruction, and implement error handling for invalid instructions.

2.2 Implementation

To implement the core components of the CPU, we first chose to represent the CPU as a structure:

```
typedef struct {
 word PC;
 word ACC;
 word IR;
 Flags flags;
} Cpu;
```

where the `PC`, `ACC` and, `IR` are all represented as `word`s and the flags is a structure:

```
typedef struct {
 int ZERO;
 int CARRY;
 int OVERFLOW;
 int INTERRUPT;
} Flags;
```

The main function for the implementation of the CPU is:

```
// Runs the fetch-execution cycle program_size times
// or until a halt is encountered
void cpu_run(const int program_size, word* mem) {
 for (int i = 0; i < program_size && CPU.PC != CPU_HALT; i++) {
  printf("=== Cycle %d === \n", i + 1);

  if (CPU.PC == CPU_HALT) {
   printf("CPU Halted! \n");
   break;
  }
  fetch();
  execute();
  check_for_interrupt();

  cpu_print_state();
 }
}
```

The design idea behind this function leverages the fact that the CPU runs for at most `program_size` amount of times unless a halt is encounter. Since the CPU cycles a variable amount of times, we decided to use a for-loop. Within the body of the for-loop we print the cycle number, fetch the instruction using `fetch()`, execute the instruction using `execute()`, check for and handle system interrupts using `check_for_interrupt()`, and finally printing the CPU state using `cpu_print_state()`. Within `fetch()`, we read the instruction from the `PC` and load it into the `IR` then increment the `PC` by 1. For `execute()`, we decode the instruction by extracting the opcode and operand then execute the instruction using the extracted opcode and operand. After the instruction has been executed, we check for interrupts and handle any interrupts accordingly.[1] To close the cycle, print the CPU state where we show the values of the `PC`, `ACC`, `IR`, and the flags.

[1]For more information on how we execute instructions and check for interrupts, see Section 4.2 and Section 5.2 respectively.

3 Module 2: Memory Management System

3.1 Problem Statement

Simulate a hierarchical memory system, including RAM as the main memory for storing data and programs, and cache (L1 and L2) for faster memory access. The memory system should allow the CPU to perform read and write operations. Implement the logic cache hits and misses, where the CPU first checks the cache for data and falls back on RAM if necessary.

3.2 Implementation

To simulate a hierarchical memory system, we implemented the RAM as an array of `word` and the L1 and L2 caches are structures:

```
typedef struct {
 Entry* items;
 int front;
 int count;
 int size;
} Cache;
```

where `Entry` is a structure:

```
typedef struct {
 word val;
 mem_addr addr;
} Entry;
```

The two main functions for this module is

4 Module 3: Instructions Set Architecture

4.1 Problem Statement

Define a simple ISA with basic operations like `ADD`, `SUB`, `LOAD`, and `STORE`. Assign opcodes to each instruction, allowing the CPU to interpret and execute them. The ISA should ensure that the CPU can fetch, decode, and execute these instructions, modifying the values in registers or interacting with memory as necessary. Provide mechanisms for error handling in case of undefined opcodes.

4.2 Implementation

5 Module 4: Interrupts

5.1 Problem Statement

Implement a mechanism for interrupts that allows the CPU to pause its current task, service an interrupt through an Interrupt handler, and then return to the previous task. This includes handling hardware interrupts (e.g., timer interrupts) and software interrupts (e.g., system calls). Ensure that the CPU can manage both synchronous and asynchronous events, simulating how real systems handle task prioritization through interrupts.

5.2 Implementation

6 Module 5: Direct Memory Access

6.1 Problem Statement

Simulate a DMA system that allows for data transfer between memory and I/O devices without CPU intervention. Implement a DMA controller that manages data transfers between RAM and devices like disk storage or network interfaces. The CPU will initiate the DMA transfer, but the actual data movement should proceed without using CPU cycles, demonstrating how DMA improves system efficiency by freeing up the CPU for other tasks.

6.2 Implementation

7 Debugging and Testing

8 The Simulation