Design Notes - CSCI 6461 Simulator

Authors: Charles, Ethan, Sam, Harshini

Technologies:

- Eclispe, Visual Studio Code
- Java (with Swing for graphics)
- GitHub, Discord, Email

Background:

The task is to design a Von Neumann Architecture Simulator that respresents the registers, memory, and processing within a procesor.

- Part 1 can be sumarized to the GUI with simple Load and Store functions for interacting with memory.
- Part 2 can be sumarized to contain many instructions for the CPU as well as cache and more robust GUI features.
- Part 3 can be sumarized to contain an additional layer of functionality. Now registers support characters as well as integers, and can handle a more complex program.

GUI

This section covers the files and design within the GUI of our simulator

GUI

The GUI was broken up into distinct panels for different parts of the code responsible for different features and functionality of the simulator. These were mainly: user input, registers, operations, and display.

Input

The Input from the user lies within the switches along the top of the simulator. A user can interact and set each of these switches in a binary state. On/Pressed is 1, Off/Deselected is 0. Even with labels, the user is not able to overload a register if the register cannot contain all of the switches inputted by the user.

DEVID

This a file of constants that various operations need to distinguish where results of the operation should be placed. Registers, Memory, Display, etc.

CPU

This section covers the files and design within the CPU of our simulator

CPU

This file contains most of the functionality of the CPU. This layer interacts with the registers on a GUI level, and uses the Instructions below to process each request. The CPU runs each program until the program halts; at which time the user can continue or stop running the program.

ALU

The ALU is referred to as the Arithmetic Logic Unit of the processor. This serves as a mathematical feature of the cpu. To perform functions like ADD, AND, NOT, etc. the ALU serves as a special subset of instruction that contain functionality to interact with multiple input and output to a desired location in the computer.

Instruction Set

All instructions implemented will be stored in the Instruction Set. This is to mirror the definition of an Instruction Set Architecture where you can define new functions easily as long as they meet requirements of input from the processor.

Every instruction contains the ability to decode and execute the command. Even instructions that do not require extra steps to decode the provided input still use a bare function in case further implementation is required.

Register

Each register could be rewritten as an implementation of an abstract class, but this design lies closer to a factory pattern. Each register in the CPU is an instance of this register object and contain the same properties.

If a register supports signed numbers, if a value is a character and not the ASCII value of a character, etc. can all be supported within the design of this class.

Memory

This section covers the files and design within the Memory of our simulator

Memory

The memory module follows standard design to store 2048 memory addresses in memory. Simply put, this memory is a lookup table just like the memory in a computer. Given an address, the simulator can store or return the value at the location.

Cache

Cache acts as a smaller, more convinient lookup table for memory. If recent addresses are stored in cache, then each load, store, and interaction with memory can check cache first rather than lookup into memory. We can support hits and misses, but there does not seem like pressing need for this feature at the moment.

---

programs

This section covers the files and design within the programs our simulator runs

boot

This boot program will be loaded into memory automatically at the start of the program so the simulator can run immediately.

program1

The goal of the test program 1 is to take in 20 numbers from the user input (range of 0 to 65,535) and then one number to test against the previous 20. The first 20 numbers will be saved and compared against the last number. The goal of this program is to return which of the 20 numbers provided is closest to the final number.

- In a sequence of number 10, 20, 30, ... 190, 200, for example, if 12 was inputted as the 21st number, the console would display 10. If you run the program through once, memory will still be set to your program, so you must use the IPL to reload instruction and memory for the program to be executed again.

program2

The goal of program2 is to read a paragraph of text, load it into memory, then ask the user for a word and find said word within said test.

- Press IPL Button. Locate Program2.txt
    - If it has loaded successfully, press okay and the prompt will come up to find the paragraph
    - Locate the paragraph of text.
- From here, with the program and text loaded, press the run button. This will have read the text into memory and now display it on the developer console with enter characters and all.
- Now, the prompt will show up to end a character (or number).
- Please enter the character for your word one letter at a time.
    - For example, "GWU" would be
    - "G">enter "W">enter "U">enter "">enter
    - When your word is done, please hit enter without any input. This indicates that you have entered the whole word.
- The program will print your word in the developer console and search for it in the text
- The output will contain sentence number, and word number. Please note that the indexing may be 1 off from what you expect due to extra characters that may have emerged in your paragraph. Spaces, etc. can all throw off the count.