Phase 1 Quick Start and Design Docs

Members: Yanwei Ding , Shahin Madani, Thomas Stanton

Table of contents

Usage Instructions

Steps:

1. Source code
    a. Download CSCI6461_Project_Part1.zip from black board
2. Preparation Instructions
    a. Unzip the program files and navigate to the folder /CSAproject.1/dist
    b. Please make sure to have the newest version of java installed as we have had issues with people not being able to run on older versions.
    c. In the “dist” folder there will be 4 files. The important ones are “run_this.bat” and “IPL.txt”. The first is the file that you click to run our program and the second is the user input file. WARNING: if you want to add your own program it has to be named “IPL.txt”. We have included a program that demonstrates our machine(see end of this document).
3. Operation of Simulator
    a. The code starts execution at memory address 0x0030.
    b. The IPL file may be replaced under /dst and may be loaded by clicking the IPL button. We have included a program that demonstrates our machine(see end of this document).
    c. The program can take single steps by pressing the ‘SS’ button or run by pressing the ‘run’ button. If running, the program will stop when it reaches a HLT instruction, a bad memory call is made, the Halt button is pressed, or the run button is pressed again.
    d. Memory can be read on the display panel which shows two memory addresses in front and behind the MAR.
    e. Finally we had some issues with the front panel buttons. They still work and toggle bits on and off, they just don't visually display if they are depressed or not.

Design

The code was designed in NetBeans Java IDE so the main "MachineSimulator" is mostly auto generated save the actionPerformed() loop and some custom functions to interface with the CPU class. Our design goal was heavily based around the UML class diagram(figure 1) we made and updated as we went along. The idea we had was to make a central "CPU" object that implemented several Registers and a single Memory object. The main_CPU as we call it is in charge of decoding operations, then processing them, and finally updating register values. The main_CPU is instantiated in the MachineSimulator program that controls the GUI. The most important part of the MachineSimulator is the actionPerformed() loop which every 500 milliseconds checks for changes in register values and tells the CPU to process one instruction if applicable.

We decided that the main data structure in this machine would be int arrays. These int[] are 16 to 4 ints long(depending on the register) and contain either 0 or 1 in each index(default all 0s). Our memory is set up to be a d2 array of 2048 rows by 16 columns, meaning that each memory address is guaranteed to have 16 indexes available (the size of our biggest register). Since our memory is indexed off ints (each row is represented by one integer value), we had to create ways to convert int arrays of 1s and 0s to integers.

If we could improve upon one thing it would probably be to improve upon our data structures as it leads to some confusion changing between arrays or different sizes, ints, and hex values for user inputs.

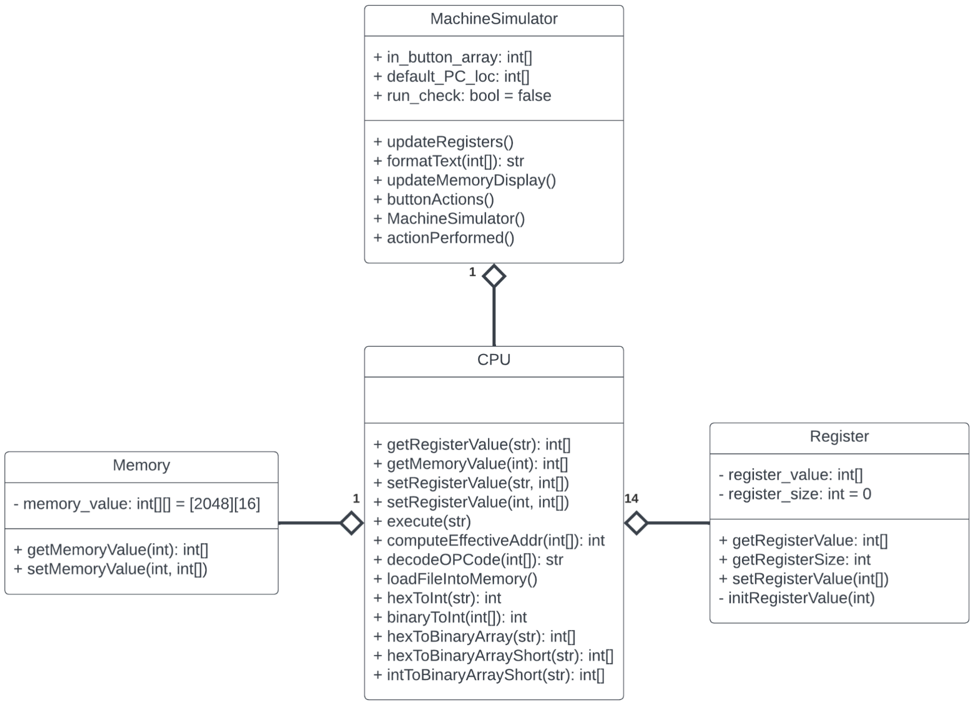
MachineSimulator
+ in_button_array: int[]
+ default_PC_loc: int[]
+ run_check: bool = false
+ updateRegisters()
+ formatText(int[]): str
+ updateMemoryDisplay()
+ buttonActions()
+ MachineSimulator()
+ actionPerformed()
1
CPU
+ getRegisterValue(str): int[]
+ getMemoryValue(int): int[]
+ setRegisterValue(str, int[])
+ setRegisterValue(int, int[])
+ execute(str)
+ computeEffectiveAddr(int[]): int
+ decodeOPCode(int[]): str
+ loadFileIntoMemory()
+ hexToInt(str): int
+ binaryToInt(int[]): int
+ hexToBinaryArray(str): int[]
+ hexToBinaryArrayShort(str): int[]
+ intToBinaryArrayShort(str): int[]
Memory
- memory_value: int[][] = [2048][16]
+ getMemoryValue(int): int[]
+ setMemoryValue(int, int[])
1
14
Register
- register_value: int[]
- register_size: int = 0
+ getRegisterValue: int[]
+ getRegisterSize: int
+ setRegisterValue(int[])
- initRegisterValue(int)

Figure 1:UML Class Diagram of Part 1

Included Program

The included program("IPL.txt") consists of two parts. It first loads several values into memory locations 12, 15, 20 and 25 (The other values in memory are there to show that other values can exist in memory). It then loads a sequence of 6 instructions into memory values 48-54. These instructions, as discussed below, demonstrate our program's ability to execute all load and store instructions with all addressing types.

Memory Location 48:

> At this location we have 0x0E0C which is a LDA to register 2. This instruction, with no need to touch memory, loads "12" into register 2.

Memory Location 49:

> At this location we have 0x0A0F which is a STR from register 2. This instruction,via no indexing and no indirect addressing, stores register 2's value ('12") into memory location 15.

Memory Location 50:

> At this location we have 0x8454 which is a LDX to index register 1. This instruction, via no indexing and no indirect addressing, loads memory location 20's value ("524") into index register 1.

Memory Location 51:

> At this location we have 0x0A4A which is a STR from register 2. This instruction, via indirect addressing but no indexing, stores register 2's value ("12") into memory location 534. This location comes from the formula c(IX = 1) + c(Address Field) = 524 + 10 = 534.

Memory Location 52:

> At this location we have 0x0439 which is a LDR to register 0. This instruction,via indirect addressing but no indexing, loads memory location 12's value ("1280") into register 0. This is done via the formula c(c(Address Field)) = c(c(25)) = c(534) = 12.

Memory Location 53:

> At this location we have 0x8875 which is a STX from index register 1. This instruction,via indirect addressing and indexing, stores index register 1 into memory location 21. This is done via the formula c(c(IX=1) + c(Address Field)) = c(524+21) = c(545) = 20.

Memory Location 54:

> At this location we have 0x0575 which is a LDR to register 1. This instruction,via indirect addressing and indexing, loads memory location 545's value ("12") into register 1. This is done via the formula c(c(IX=1) + c(Address Field)) = c(524+21) = c(545) =25.