**Instruction Guide**

## **Steps to run the Project(SEE PHASE 3 AT BOTTOM):**

**Screenshot 2023-03-12 at 17.48.10.png**

* Run the run\_this.bat file in the CSAproject.1/dist/

This is the executable file —>

* Click on the IPL button It will be able to run the set of instruction provided inside the IPL.txt file
* After the IPL.txt file is loaded the program will jump to memory location 20.
* Simply hit run and the program will begin execution.
* It is finished when the result value appears in the console printer.
* See last section for description of the program(WARNING IT DOES TAKE A BIT)!

Then the program will start with the main interface:

1. MachineSimulator interface.

Inside the machine similar interface there is a button to go to DebugGUI and the input/out GUI.

**UI Display:**

In the input panel, the user can enter either a hexadecimal or binary number. If the input is binary, the hexadecimal label will display the input in hexadecimal form. If the number is in hexadecimal format, the binary label will display the number in binary.

Graphical user interface, application

Description automatically generated

* The **GPR0-3 are 4**, 16-bit General Purpose Registers. These can store both memory address and data.
* The **IXR**1-0 are 3, 16- bit Index Registers, they hold the current offset of a memory location. These are used for pointing to operand addresses when running the program.
* **PC: PC** is a 12-bit program counter which has the address of the next instruction to be executed from memory. It is a digital counter needed for faster execution of tasks as well as for tracking the current execution point.
* **MAR: Memory Address Register** is a 12-bit register which is used to access data and instructions from memory during the execution phase of instruction. MAR holds the memory location of data that needs to be accessed. When reading from memory, data addressed byMAR is fed into the MBR (memory buffer register).
* **MFR:** It is a 4-bit Machine Fault Register.
* **MBR: Memory Buffer Register** is a 16-bit register which is used to store the data being transferred to and from the immediate access store. It contains the copy of designated memory locations specified by MAR. It acts as a buffer allowing the processor and memory units to act independently without being affected by minor differences in operation.
* **IR: IR** is a 16- bit register that holds the instruction currently being executed or decoded. Each instruction to be executed is loaded into the instruction register, which holds it while it is decoded, prepared and ultimately executed, which can take several steps.
* **CC:** The overflow and divided by zero flag will be shown in the CC
* **SS: The SS** is the Single Step button, it is used to execute the one step at a time in order to determine functioning.
* **Run:** This button is used to execute all of the instructions specified in the input and produces the final output.
* **INIT:** Initializing PC, Instruction, and memory
* **IPL**: This button is used to run the program

**Design Docs:**

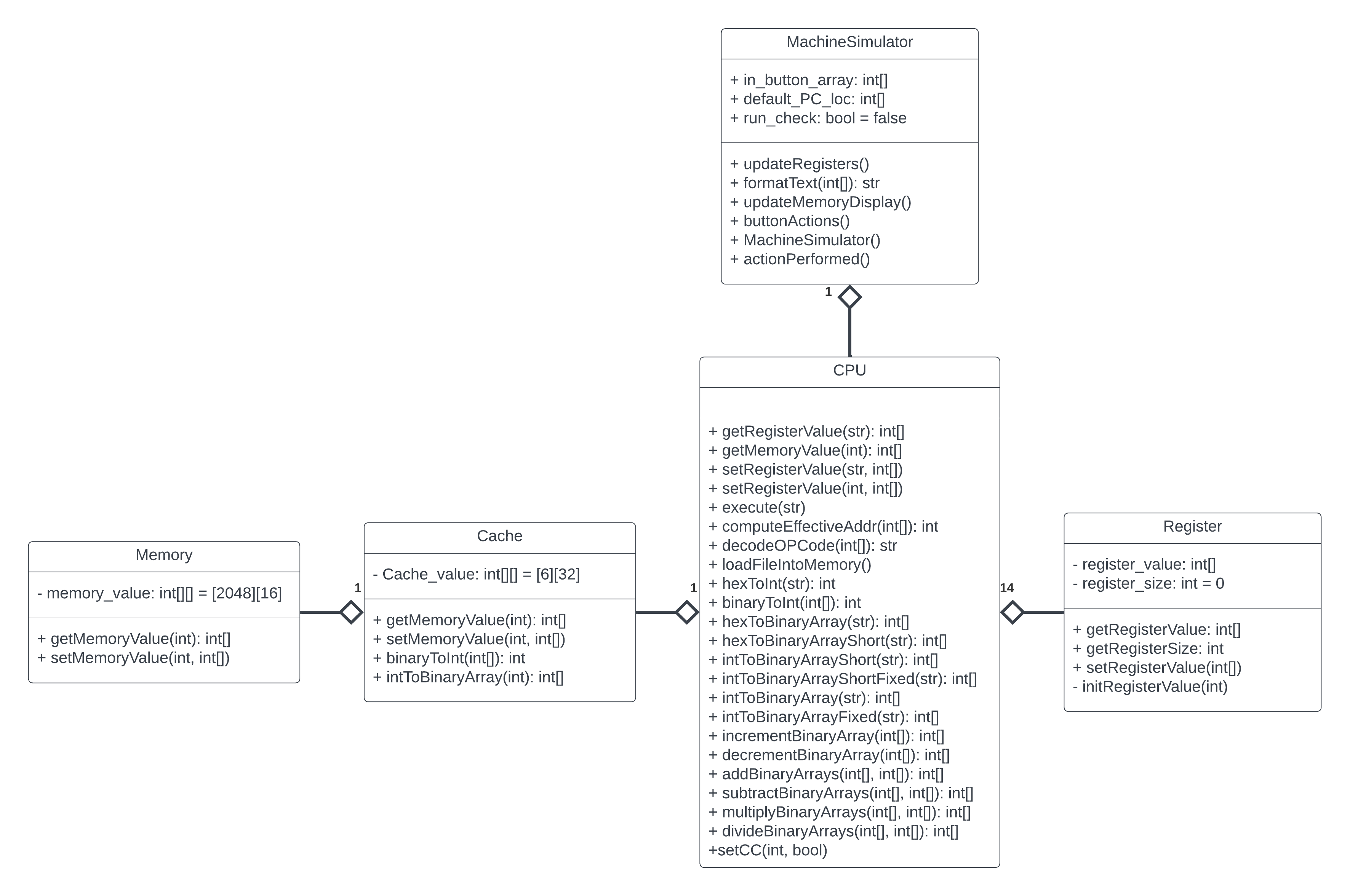
* Cache
  + The Cache is designed to sit right in front of the memory. We designed it in such a way that the CPU class cant tell a difference between the Cache and Main\_Memory. Instead the cache handles all memory operations(shown in figure 1.)

Figure 1.: Class Diagram with Cache(Phase 2)

**Running a Program:**

* This program contains 20 numbers stored at 0x0320-0x0333 and searches through these 20 numbers seeking to find the number closest to the one stored at 0x0190.
* The user is free to change the values here but the provided ones are:
  + Array of numbers to search through: 1, 3842, 3, 212, 1365, 20, 162, 2184, 2313, 161, 273, 3084, 1303, 3598, 14, 4112, 176, 12336, 68, and 80.
  + Number we are looking for: 25.
* The result of the program, printed in the printer window, should be 0000000000010100 or 20(since that is the closest number in the array) when finished.

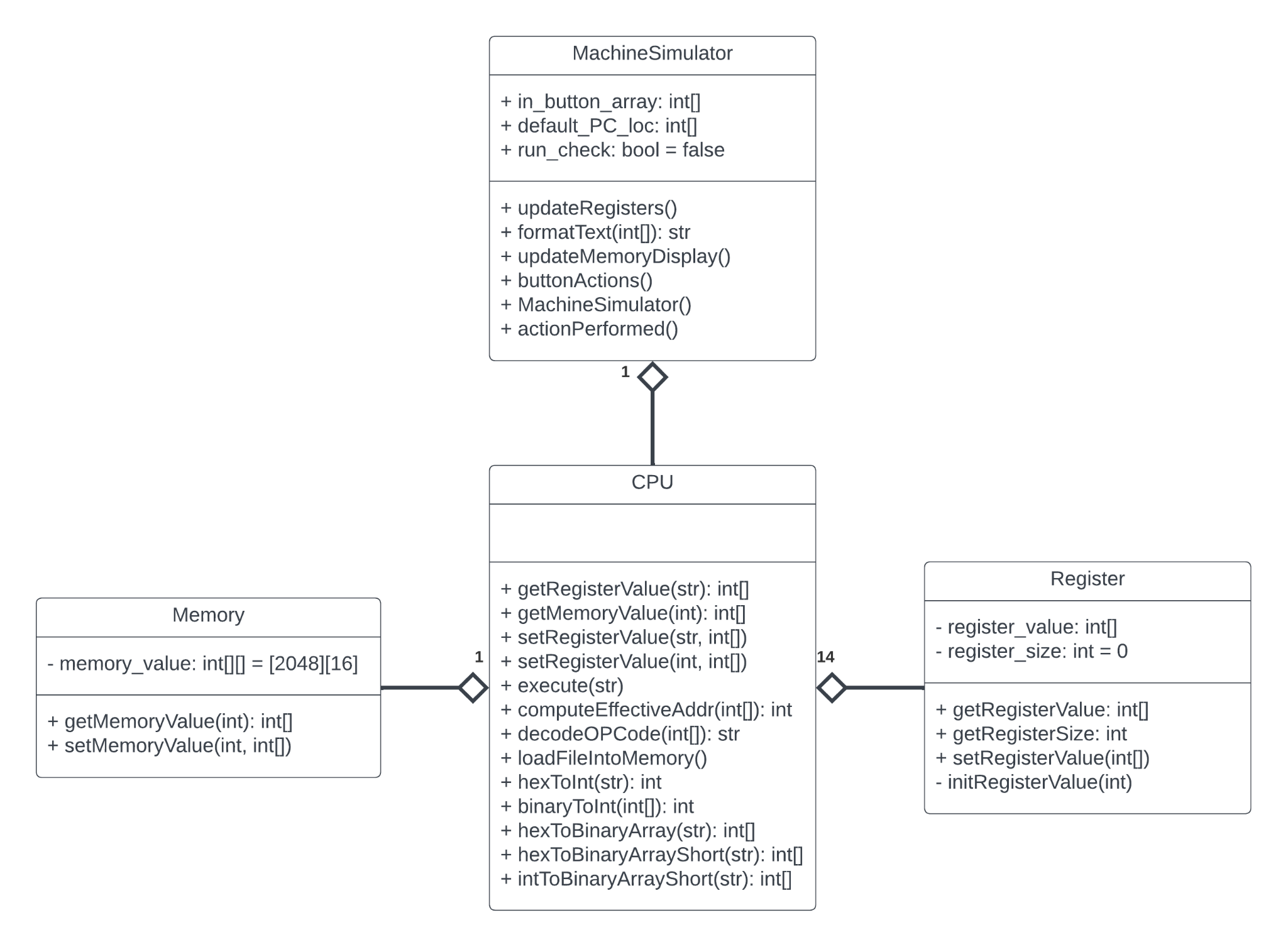
**Phase 2**

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.

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.

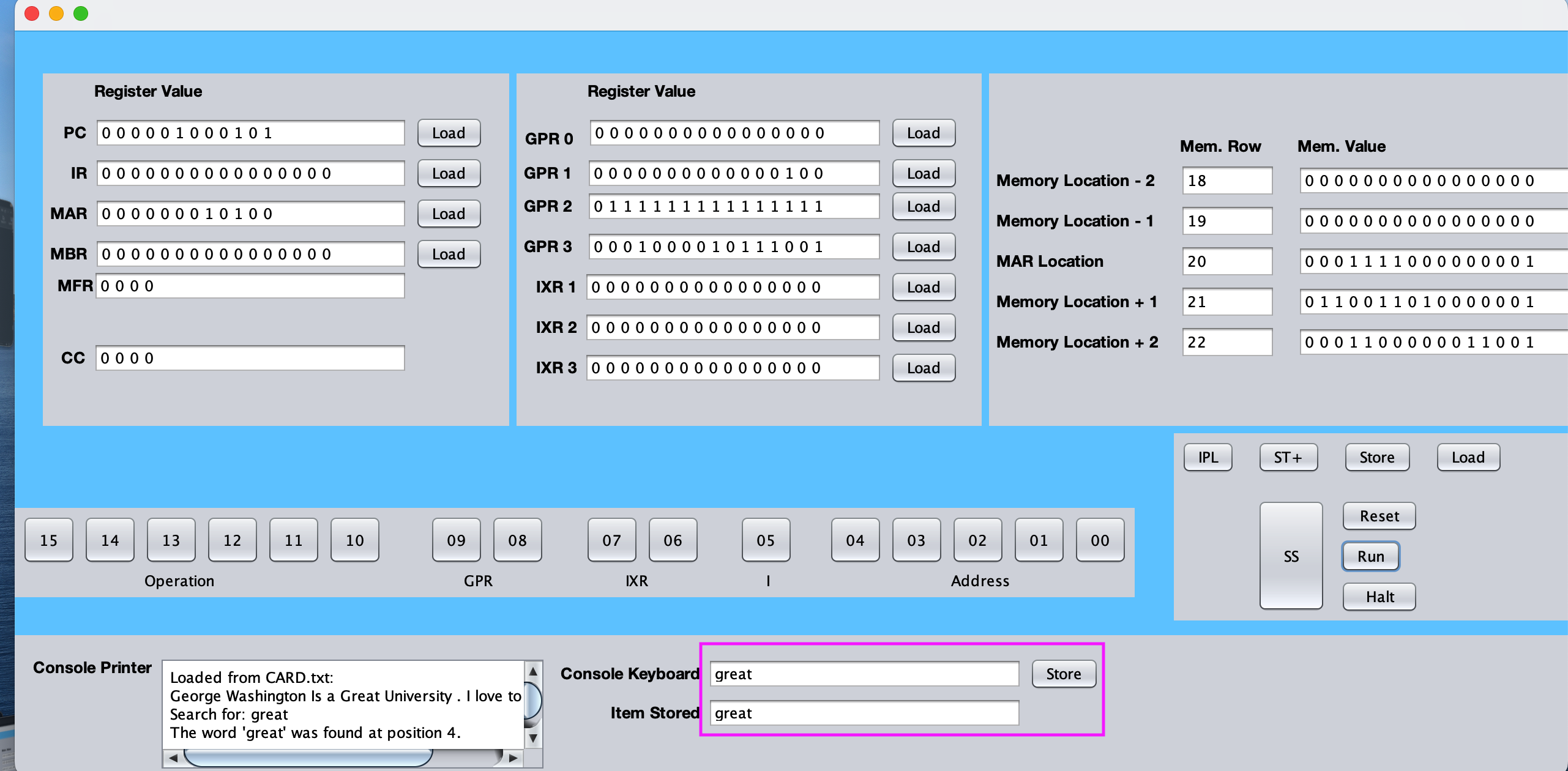
Memory Location 51:  
At this location we have 0x0A4A which is a STR from register 2. This instruction, via indirect addressing but no indexing, stores register 2’s value (“12”) into memory location 534. This location comes from the formula c(IX = 1) + c(Address Field) = 524 + 10 = 534.

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.

**Phase 3**

**New UI elements we have added: “Load Program 2” button:**

When we click on the IPL button, our console will load content from the IPL.txt and CARD.txt to the memory location 0x0E0C to the length of paragraph containing 3 sentences, and the PC will be loaded with the first instruction addresses from the IPL.txt

On Clicking “RUN” button, the IPL.txt instructions will execute.

First we will print the content of the sentences in the console printer, then we will input the word we are going to search into the console keyboard and store into the memory.

Here, each character of the sentence will be compared to each character in the word, if matches will prompt with the message the word found at the which position in the paragraph.

The GPR1 register will output the position of the word.