CSCI6461 - Computer System Architecture

Machine Simulator Part - 2

Team 3 (Lekha, Oliver, and Rutvik)

12 December, 2022

This file contains both our instruction manual for use of our simulation as well as our design notes.

Instruction Manual

Basic Overview

Our simulator works simply. All the registers (for this initial and relatively simple version of the machine) are given on the user console. The contents or any register can be modified directly by clicking on radio buttons to change individual bits. This means that there is no need to load from a main set of buttons into the register, but instead that all the registers can be directly and individually modified more easily.

Several buttons are provided with this version of the machine. The “load” and “store” buttons work as specified for the project. The “init” button prompts the user to load a text file which is of the form where each line indicates first in hexadecimal an address and then a space and then in hexadecimal a memory contents. After selecting a file, the contents are loaded at the specified addresses.

Running the simulator is also very simple, the user may double-click on the jar file or they can run the “***java -jar arch-project.jar***” from the command line interface.

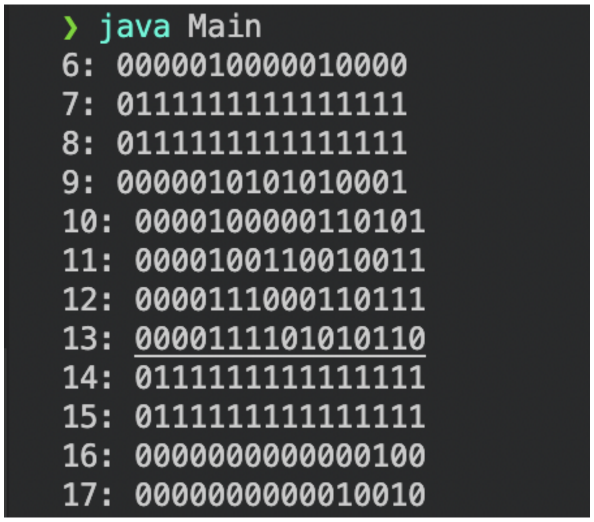
Updating Register Values in the User Interface

We include this section to make clear the use of our user interface specifically with regards to the task of updating the values of registers. Each register is a k-bit binary sequence of 1’s and 0’s, where k is some value usually 4 to 16. Each register is displayed on the user console as a sequence of k radio buttons with buttons toggled to the “selected” state if they represent a binary 1 and toggled to the “unselected” state if they represent a binary 0. At any point in use of the machine simulator, the current value of any register can be modified *simply by clicking on the desired radio button to toggle it from 0 to 1 or 1 to 0* (from its current value to the other).

In this way, we avoid the need for separate user console loads and stores and we make the UI very easy for the user to use to update register values. Of course, all the instructions are implemented and can be used as described to manipulate the machine's state in addition to this user-interface functionality that we have provided.

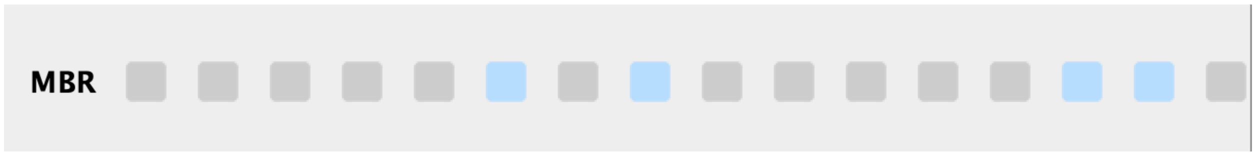
Steps to execute an instruction:

1. Assuming the user has something in the memory they must use the MBR to give instruction and store to into the memory using the store button



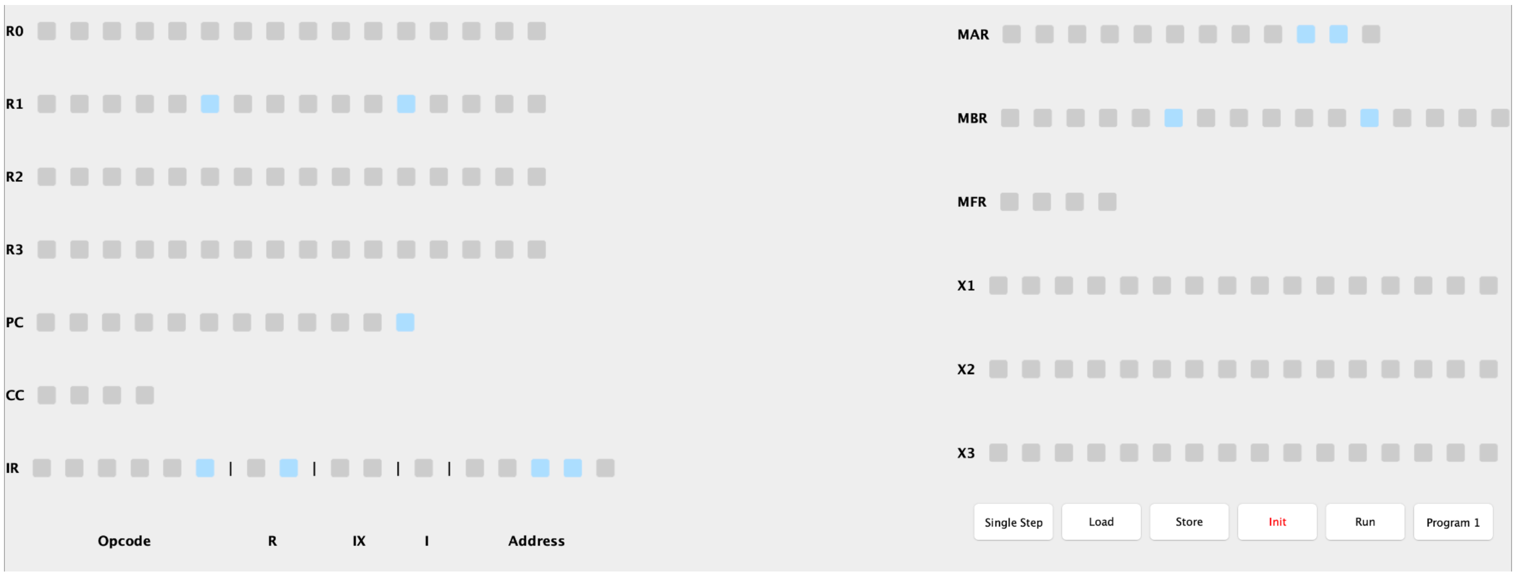
```
❯ java Main
6: 0000010000010000
7: 0111111111111111
8: 0111111111111111
9: 0000010101010001
10: 0000100000110101
11: 0000100110010011
12: 0000111000110111
13: 0000111101010110
14: 0111111111111111
15: 0111111111111111
16: 0000000000000100
17: 0000000000010010
```

Output of the terminal once user loads the memory with the contents of IPL.txt file using the init button



Here the first 6 bits are for the opcode (000001 for the LDR instruction), next 2 bits are for the GPRs (register R1 in this case), next 2 bits are for the index registers, next 1 bit is for indirect addressing and last 5 bits are for specifying the address (address at location 6 in this case)

2. Once the user has store the instruction into the memory than they must point the PC to that location into the memory and load that instruction using the load button
3. Once the user has loaded the instruction from the memory they must hit the single-step button to execute that instruction
4. Executing the instruction will load register R1 with the contents of address of location 6 in the memory



Simulator after executing the LDR instruction

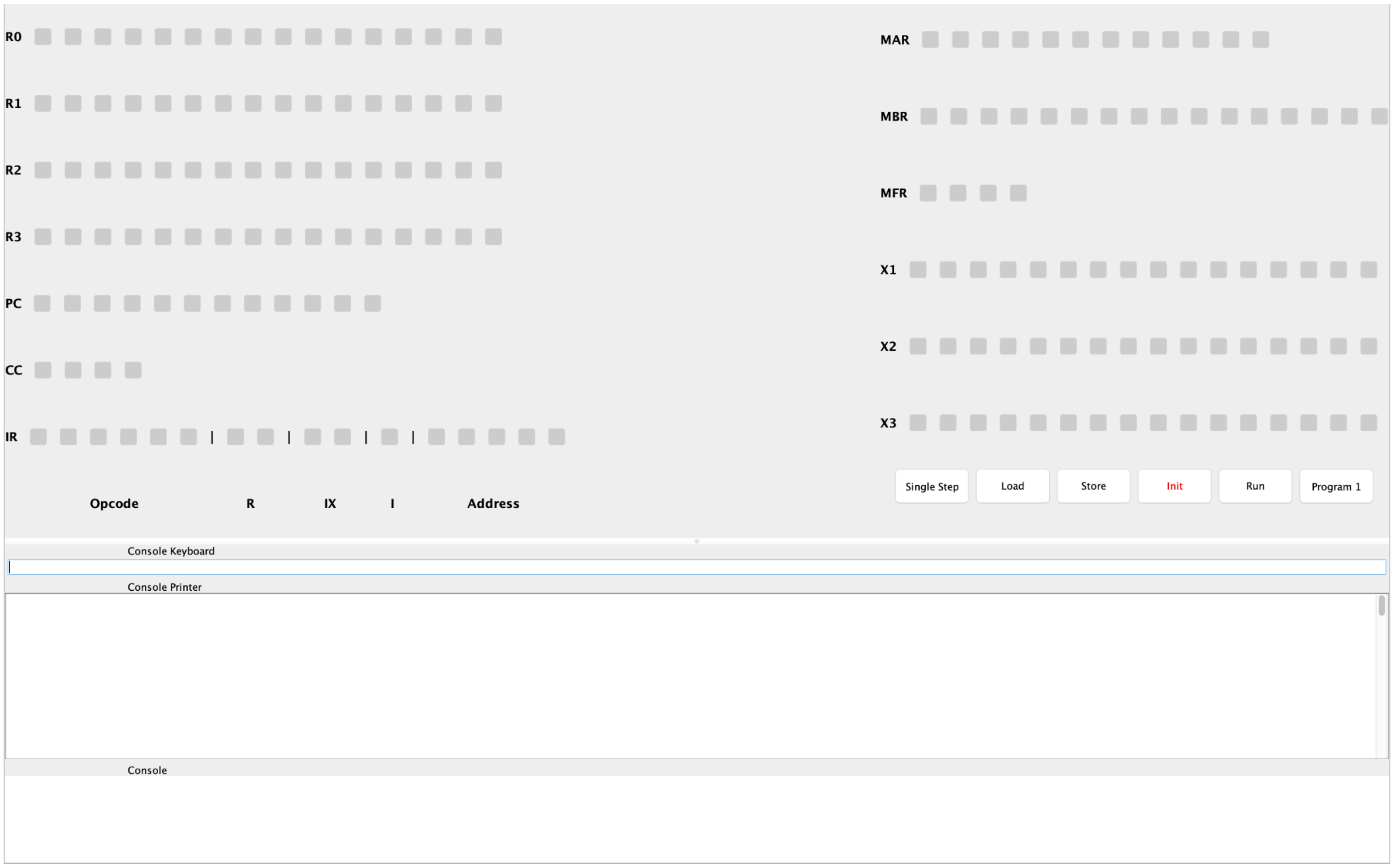
5. Executing the instruction (using the single-step button) will also update all the registers, increment the PC to point at the next instruction, update the IR, MAR and the MBR.

Using the cache functionality (and checking that it works):

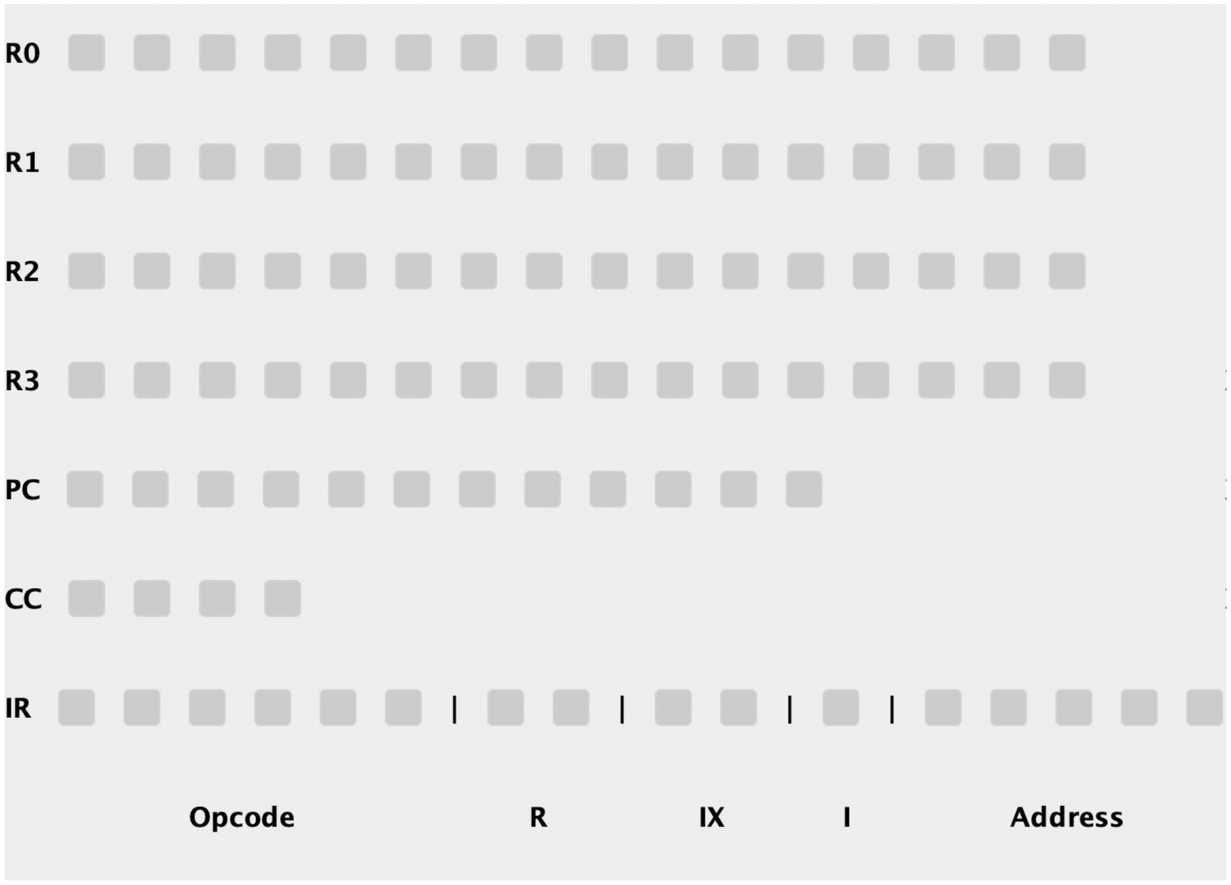
The specifications for this project ask that we think about how to demonstrate that caching works. The easiest way to see that it works is to manually load and store results. We implemented the cache such that every occurrence of a *cache hit* is accompanied by a print to the command line. Note that this print is among the trace contents, rather than that text consoles on the UI, and this is a design decision based on the fact that with such frequent *cache hits*, the text console in the UI would become busy and distracting during normal use, and so we have the machine print to the command line shell being used to run the program. Thus it is easy for the user to check that in fact cache hits are occurring at desired and correct times. For example it is easy to access 16 different memory addresses and then see that accessing any one of those 16 results in a cache hit whereas any different memory access does not.

Console Layout:

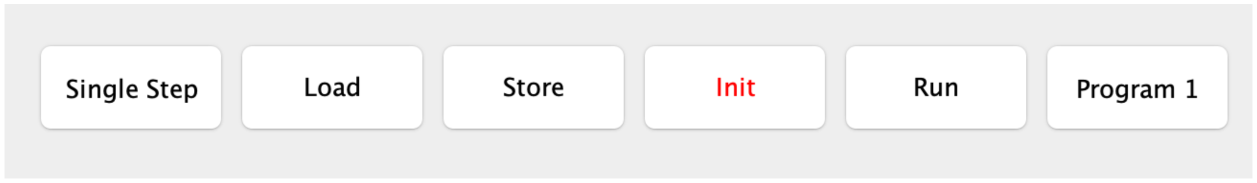
The Console is divided into 3 panels, it contains 4 GPRs (R0-R3), PC, CC, IR, MAR, MBR, MFR and 3 Index Registers (X1-X3). The console further contains radio buttons for these registers depending upon their sizes, which eliminates the need of a load button for every register and makes the process of entering values faster and allows us to utilize the space more efficiently. Lastly, the console also contains Single-step, Load, Store and Init buttons to execute instructions and perform load, store operations onto the memory.



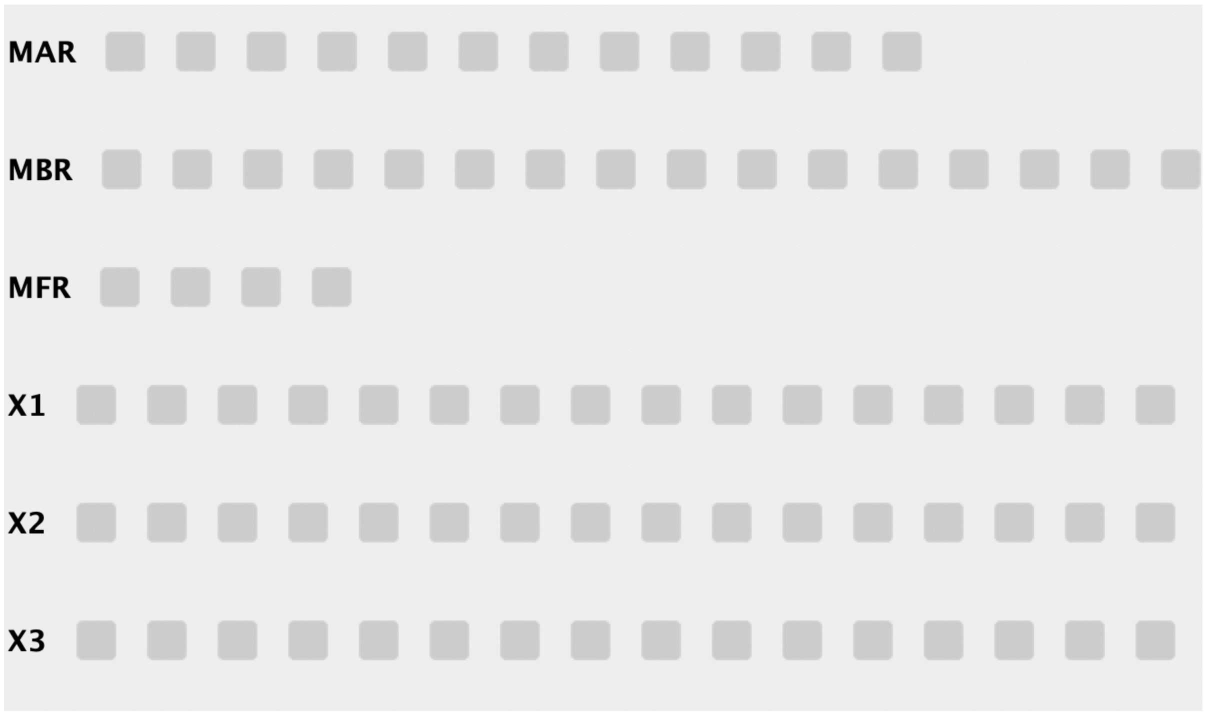
The Register panel contains all general purpose registers, the program counter, the condition code, and the instruction register, where the first 6 bits of the IR is the opcode which can be used to specify instructions, next 2 bits are for selecting the general purpose registers, next 2 bits are for selecting the index registers, next one bit is for toggling between direct and indirect addressing and the last 5 bits are for specifying the address.



The buttons panel consists of all the buttons in the frame, it houses single-step, load, store, run and init buttons, for executing instruction, performing loading/storing operations and reading a file to load its contents onto the memory to its corresponding address.



The Memory panel contains memory buffer register, memory address register, machine fault register and the index registers.



Store Button

- Used to store the value specified in MBR in the address specified in MAR
- Can be used to store instructions

Single Step Button

- Initially address from PC will be copied to MAR and then, PC will be incremented by one
- Next, contents in the address stored in MAR will be copied to MBR
- Then, contents from MBR will be copied to IR
- Now, instruction in the IR will be parsed in executeInstruction() function and will be executed based on the opCode
- If opCode=1, then ldr instruction will be executed wherein value from the given address will be copied to targetRegister and the display will be updated.
- If opCode=2, then str instruction will be executed wherein value from the register will be copied to target Address

Load Button

- Used to load contents of the address currently pointed by the PC

Program 1 Button

- Used to initiate the Program 1 functionality, allowing the user to input 20 numbers in hexadecimal using the console keyboard and a target number and output the closest number to the target number from the first 20 numbers entered by the user.

Program 2 Button

- Used to initiate the Program 2 functionality, allowing the user to search for a word from a text file that the user may load using the **Program 2** button.

Init Button

- Used to load the memory with the contents of the IPL.txt file

Load Instruction Button

- Used to load instruction into the memory from a space delimited text file, where the first part is the address where the instruction should be stored in the memory and the second part is the instructions in binary format.

Instructions Implemented for Phase 2:

All the opcodes for instructions are converted to decimal from octal number. All the fields for the instructions are also given in binary using the radio buttons

JZ - 001000 R, IX, I, Address

JNE - 001001 R, IX, I, Address

JCC - 001010 R, IX, I, Address

JMA - 001011 R, IX, I, Address

JSR - 001100 R, IX, I, Address

RFS - 001101 R, IX, I, Address

SOB - 001110 R, IX, I, Address

JGE - 001111 R, IX, I, Address

MLT - 010000 R, IX, I, Address

DVD - 010001 R, IX, I, Address

TRR - 010010 R, IX, I, Address

AND - 010011 R, IX, I, Address

OR - 010100 R, IX, I, Address

NOT - 010101 R, IX, I, Address

SRC - 011001 R, IX, I, Address

RRC - 011010 R, IX, I, Address

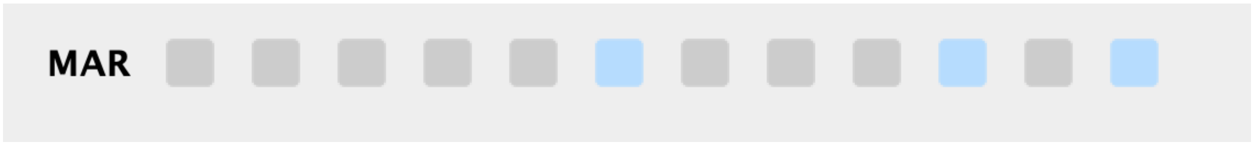
IN - 110001 R, IX, I, Address

OUT - 110010 R, IX, I, Address

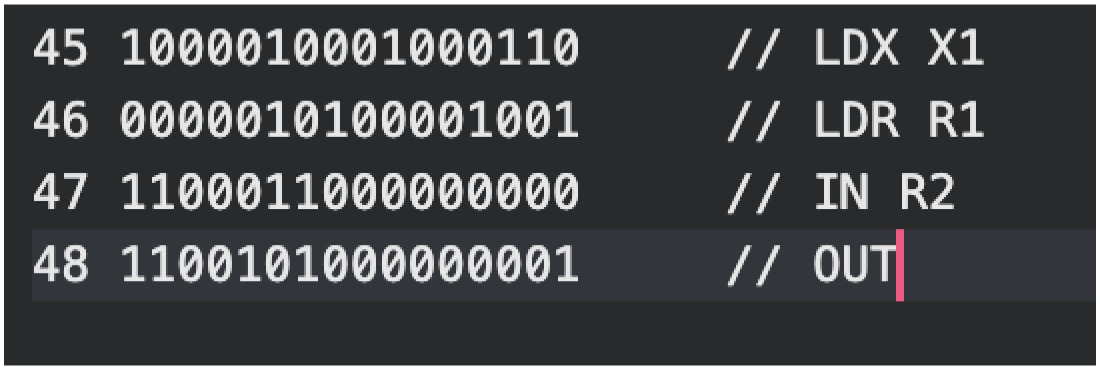
Phase 3:

Loading instruction from a file:

1. Considering the user already has something stored in the memory (loading the IPL.txt file into the memory, it is the **Input.txt** file for our project, using the **Init button**), they can load instructions from a file using the **Load Instruction** button.
2. Once the instructions are loaded into the memory, the user must first load a instruction from the memory:
    a. To load an instruction from the memory the user must use MAR to point to the location where the instruction is stored (as shown below) and hit the load button.



    b. Once the user hits the **Load** button the simulator will load the instruction in the MBR and point the PC to point at the location where the instruction is stored
    c. Now the user may hit the single-step button to execute the instruction, once the instruction is executed the PC will be incremented to point to the next instruction in the memory.
    d. The user also has the option to use the **Run** button once they have loaded the instruction as mentioned in the previous step, hitting the run button will execute all the instructions (from the location where the PC is pointing) until the simulator has finished executing all the instructions and then it goes into halted state.



```
45 1000010001000110     // LDX X1
46 0000010100001001     // LDR R1
47 1100011000000000     // IN R2
48 1100101000000001     // OUT
```

Contents of Instruction.txt file.

NOTE: <u>The first part is the address (in hexadecimal) where the instruction will be stored in the memory and the second part is the instruction in binary format. The last part is added to let you know what these instructions are and they are not included in the actual file.</u>

Program 2:

To execute program 2 the user can hit the program 2 button and load the text file (p2.txt provided within the zip) and when prompted they must enter the word they're looking for, if the loaded file contains that word it will be displayed on the console output.

Phase 4 Branch Prediction:

For phase 4 of this project, we selected the second of the two options, namely, to implement simple branch prediction speculative execution. Since the logic we implemented was covered in lecture, we provide only a brief overview of how our implementation of branch prediction was handled in JAVA. In particular, we specify in this document a few minor implementation details and how the user can confirm the changes are affecting the simulator during use which simplifies to modified terminal outputs to provide the user information about when branches were taken or not and how this modified the branch prediction bits.

We use a simple 2 bit branch predictor as described in the course, and we choose to implement it simply with a single integer in JAVA such that the values 0, 1, 2, and 3 respectively correspond to the 2-bit values of 00, 01, 10, and 11 respectively. Branches being taken or not then correspond to addition and subtraction respectively in the values we stored in the *branchPredict* variable in our simulator (Simulator.java).

Design Notes:

We implement our simulator in ***Java***, and here we provided some notes on the design. More extensive documentation is available in two forms. First, in the *docs* folder of our publicly accessible GitHub repository there is extensive JavaDocs-style documentation of all classes, functions, etc. Accordingly, the source code we provide is well documented and commented. The JavaDocs-style comments are throughout the source code explaining individual method stubs, classes, et cetera, but in addition to this there are many comments (denoted with “//” rather than “/**<...>*/”) which explain step by step logic and algorithmic aspects of the code to make it readable/interpretable.

Classes:

The source code is divided into Classes:

- Interface
- InterfaceActionListener
- Register
- RegisterListener
- Main
- Simulator
- Memory
- Utilities

Interface Class:

A Graphical User Interface class for using the simulated computer. This class inherits from the JFrame class, it contains a frame which houses 3 panels each used to house radio-buttons for all registers and buttons for different operations like single-step, load, store, init, halt and run.

InterfaceActionListener Class:

This is a simple action listener class for various buttons of the interface, which can be responded to by calling the corresponding simulator functions.

Register Class:

This is the class to represent the Register which have different names, values which are arrays of zeros and ones and their corresponding sizes which are in number of bits.

Main Class:

The Main class which initializes the simulator and gives the simulator its interface.

Simulator Class:

A class that simulates the operations of the computer. This class is responsible for creating the simulator instance with a memory size, it also contains some important functions like load, store, step, init, executeInstructions, updateRegister, incrementPC, and getRegister.

RegisterListener Class:

A listener class for the register buttons, used to update various register's values and in turn also update the simulator.

Memory Class:

This class is responsible for creating the memory of the simulator. Its implementation is relatively simple containing the expected *get* and *set* methods as well as the implementation of our **Cache** which is simulated in a nice and simple way that allows for easy counting of *cache-hits and cache-misses*. The cache is implemented as a data structure abiding by a first-in-first-out (FIFO) policy which tracks the memory addresses currently in cache. When initialized, the cache is empty[1], and as memory accesses (calls to *get* function) are performed, each address accessed is added to cache (removing the oldest address in the structure each time). Whenever a *get* is performed on the memory and the address is already in cache, this is caught as a *cache-hit* and printed as an output of the program as part of the trace data. In this way, it would be easy to count *cache-hit* rates (and analogously *cache-miss* rates).

1

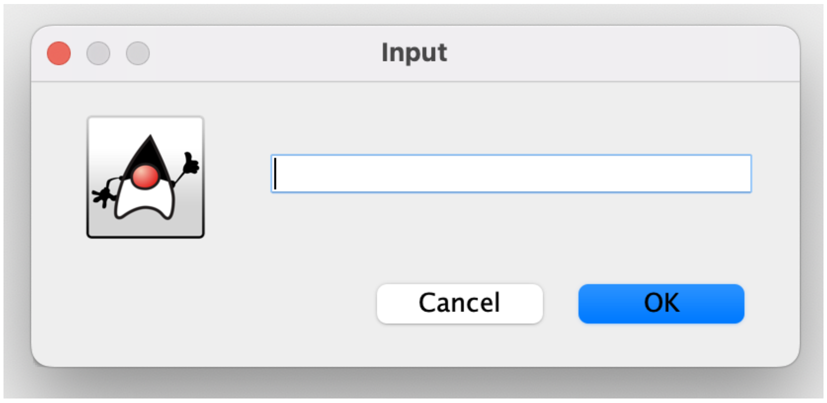
Utilities Class:

This class contains static functions that are useful for the simulator. It contains functions converting hexa-decimal numbers to binary numbers, binary numbers to decimal numbers and decimal numbers to binary numbers.

Additional Design Notes

We also will note here some design decisions we have made. We opted for decisions that made the resulting architecture *simple* whenever possible and a decision was not specified in the project description provided.

- For this part of the project (part 2), we do not have machine faults implemented, and so for now we do not bother to reserve the first six positions of memory. Rather, we allow use of all positions in memory to make the architecture simpler at this point in the project. (We will reserve the required parts of memory for parts of the project when it becomes necessary.)
- Another design decision that we made is regarding the IN, OUT and taking input for the target number for the Program 1, where we are using an input dialog box instead of taking such inputs using the text field (console keyboard), however we're still using the text field (console keyboard) to take 20 numbers from the user and the text area (console printer) for printing the output to the user as mentioned in the project description.



- Since there is significant work to be done in actually carefully implementing machine code for the simulator, we also chose to include a demonstration of how this process is carried out. We include in addition to the *program1.bin* file which includes the machine code for program 1, a file called *program1_SOURCE.pdf* which contains a table of information that shows a snapshot mid-development of program 1 including machine instructions written to be readable/interpretable by a person, as well the conversion to binary, hexadecimal, and octal, and the ultimate strings of characters that are copied into the *program1.bin* file.