|  |
| --- |
| CSCI 6461 Computer Simulator |
| Design Notes |
| Author : Lulwah AlKulaib Amirah Abdulrahman Yichen Zhou Han Wang |
|  |
|  |

## **Project Overview**

The CISC Simulator is a project to build a basic Computer Simulator that demonstrates the structure of a computer system, the breakdown of ISA, executes instructions and operations.

## **Technology and Tools**

* Programming language: Java 1.8
* IDE: IntelliJ IDEA CE 2
* Repository: GitHub

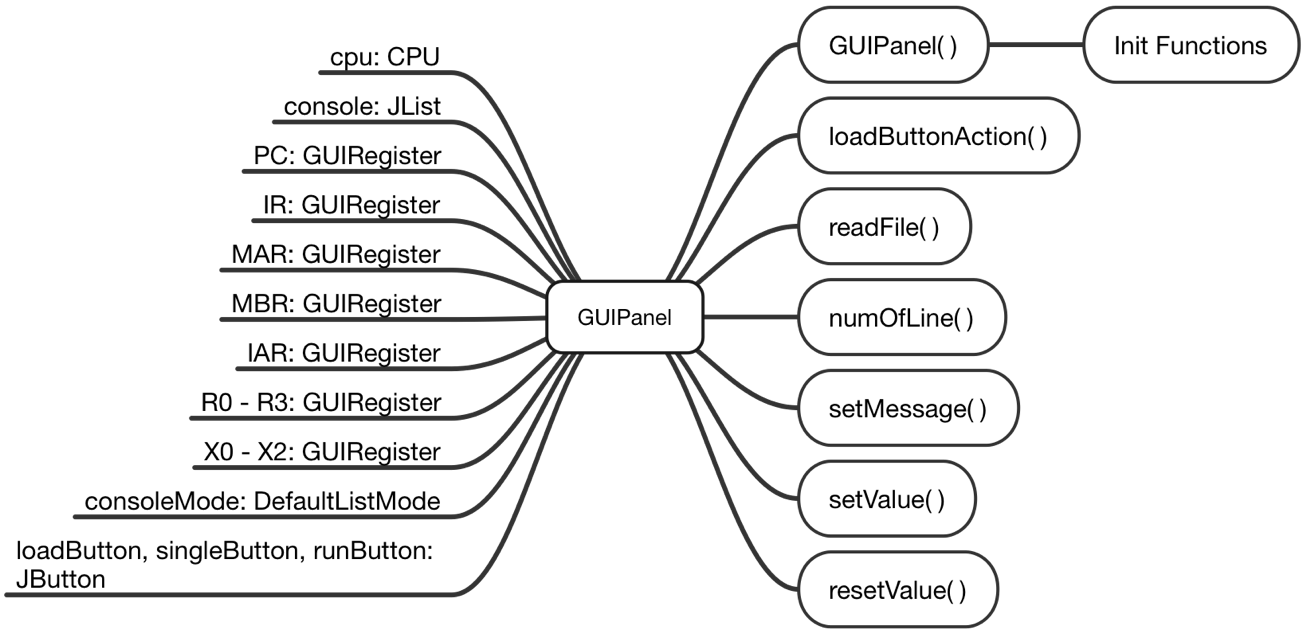
## **Part I: Basic Machine**

### Description: We have implemented a simple memory that you can see represented in an Array. The Simulator has the ability to execute Load and Store instructions using the graphical user interface we have created.

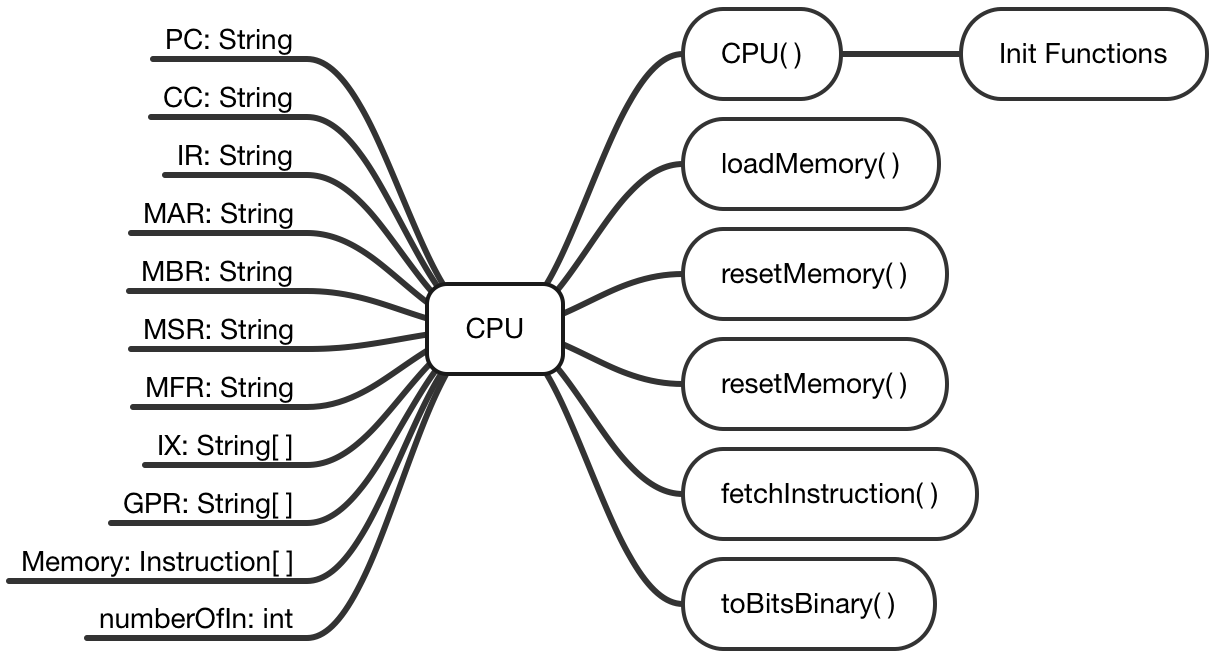
### Input: Input file contains binary instructions.

### Test and Error handling: Exceptions and handlers are used when reading from file and elsewhere needed.

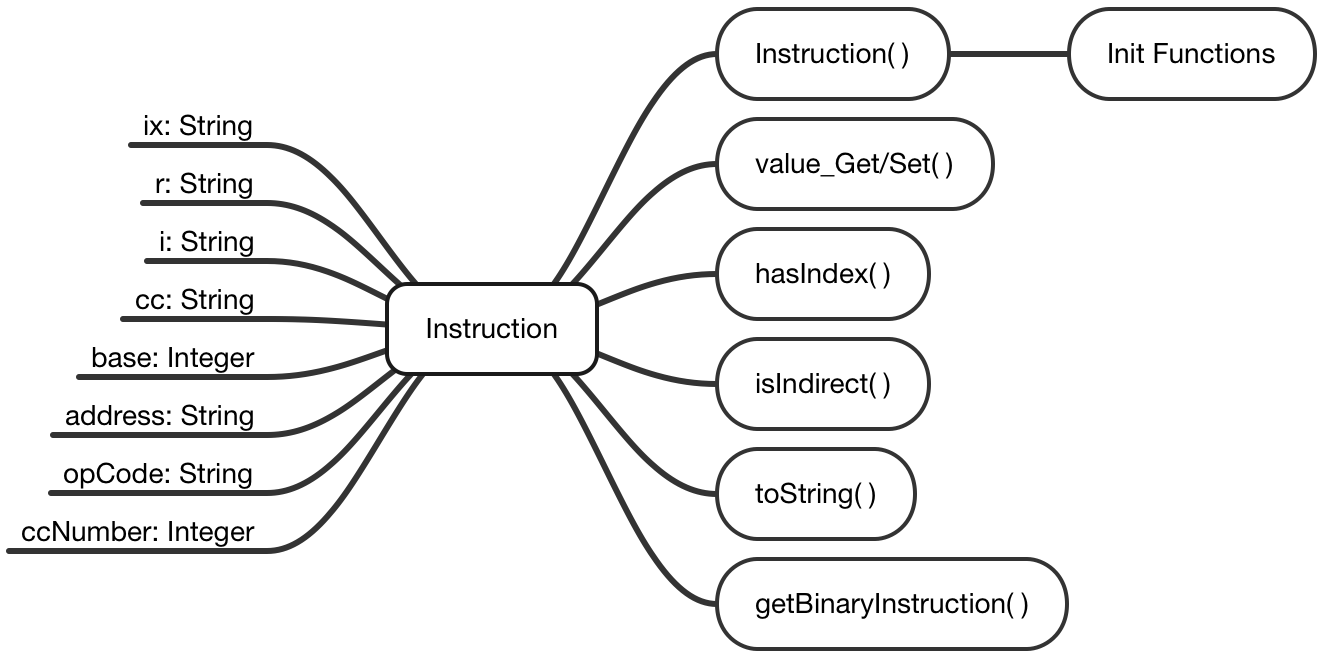
### API Maps: All labels on the left of Class Node are class properties; All labels on the right of Class Node are functions;



GUIPanel Class Node



CPU Class Node



Instruction Class Node

### Classes Overview:

* 1. GUIPanel: Creates and initializes the Graphical User Interface, representing all objects in view, associated with their actions that are supposed to run when buttons are clicked.
  2. GUIRegister: Creates and initializes registers view.
  3. Instruction: A class where Instruction constructors are initialized, an Instruction can be broken down per it’s ISA, setters and getters available accordingly. Will be used more in upcoming phases.
  4. CPU: A class that performs ‘load’ instruction by loading file of binary instructions into memory, displaying the content on the GUI. It also holds the functionality of ‘Single Step’ instruction where a single line is read from memory, and registers values are changed accordingly on GUI.
  5. Main: Loads the GUIPanel and creates a new instance of it. The part that we run every time we run a new instance of the program.

**Part II: Memory and Cache Design**

**OPCode**

In this phase, we have created the OPCodes to perform their correspondent tasks following the project description document.

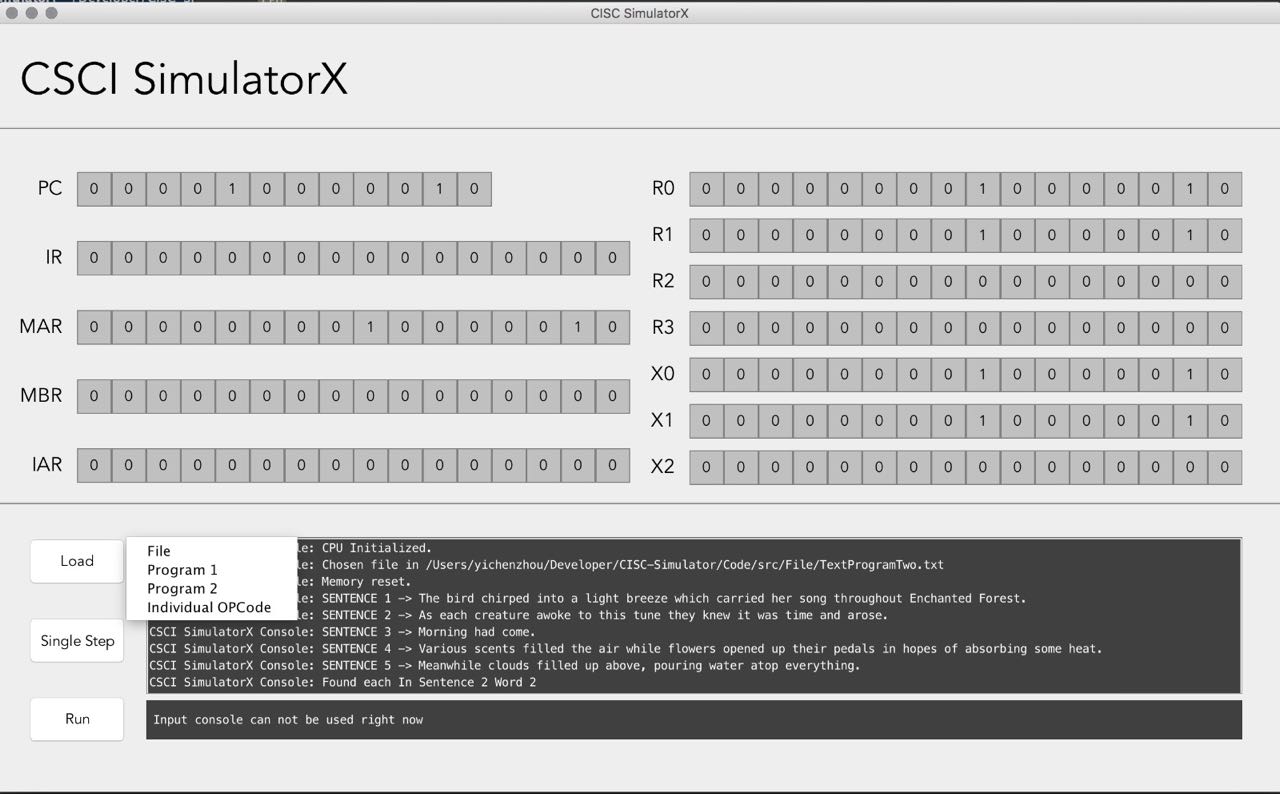
|  |  |  |
| --- | --- | --- |
| OpCode | Name | Function |
| 0 | HLT | Halt Machine |
| 1 | LDR | Load Register from Memory |
| 2 | STR | Store Register to Memory |
| 3 | LDA | Load Accumulator with Memory |
| 4 | AMR | Add Memory to Register |
| 5 | SMR | Subtract Memory to Register |
| 6 | AIR | Add Immediate to Register |
| 7 | SIR | Subtract Immediate from Register |
| 10 | JZ | Jump if Zero |
| 11 | JNE | Jump if Not Equal |
| 12 | JCC | Jump if Conditional Code Specified |
| 13 | JMP | Unconditional Jump |
| 14 | JSR | Jump Subroutine |
| 15 | RFS | Return from Subroutine |
| 16 | SOB | Subtract One and Branch |
| 17 | JGE | Jump If Greater or Equal |
| 20 | MLT | Multiply Register by Register |
| 21 | DVD | Divide Register by Register |
| 22 | TRR | Test if the Contents of Two Registers are Equal |
| 23 | AND | Logical AND |
| 24 | ORR | Logical OR |
| 25 | NOT | Logical NOT |
| 30 | TRAP | Traps to Memory Address 0 |
| 31 | SRC | Shift Register by Count |
| 32 | RRC | Rotate Register by Count |
| 33 | FADD | Floating Add Memory to Register |
| 34 | FSUB | Floating Subtract from Register |
| 35 | VADD | Vector Add |
| 36 | VSUB | Vector Subtract |
| 37 | CNVRT | Convert to Fixed/Floating Point |
| 41 | LDX | Load Index Register from Memory |
| 42 | STX | Store Index Register to Memory |
| 50 | LDFR | Load Floating Register to Memory |
| 51 | STFR | Store Floating Register to Memory |
| 61 | IN | Input Character to Register |
| 62 | OUT | Output Character from Register |
| 63 | CHK | Check Device Status |

**Cache Design**

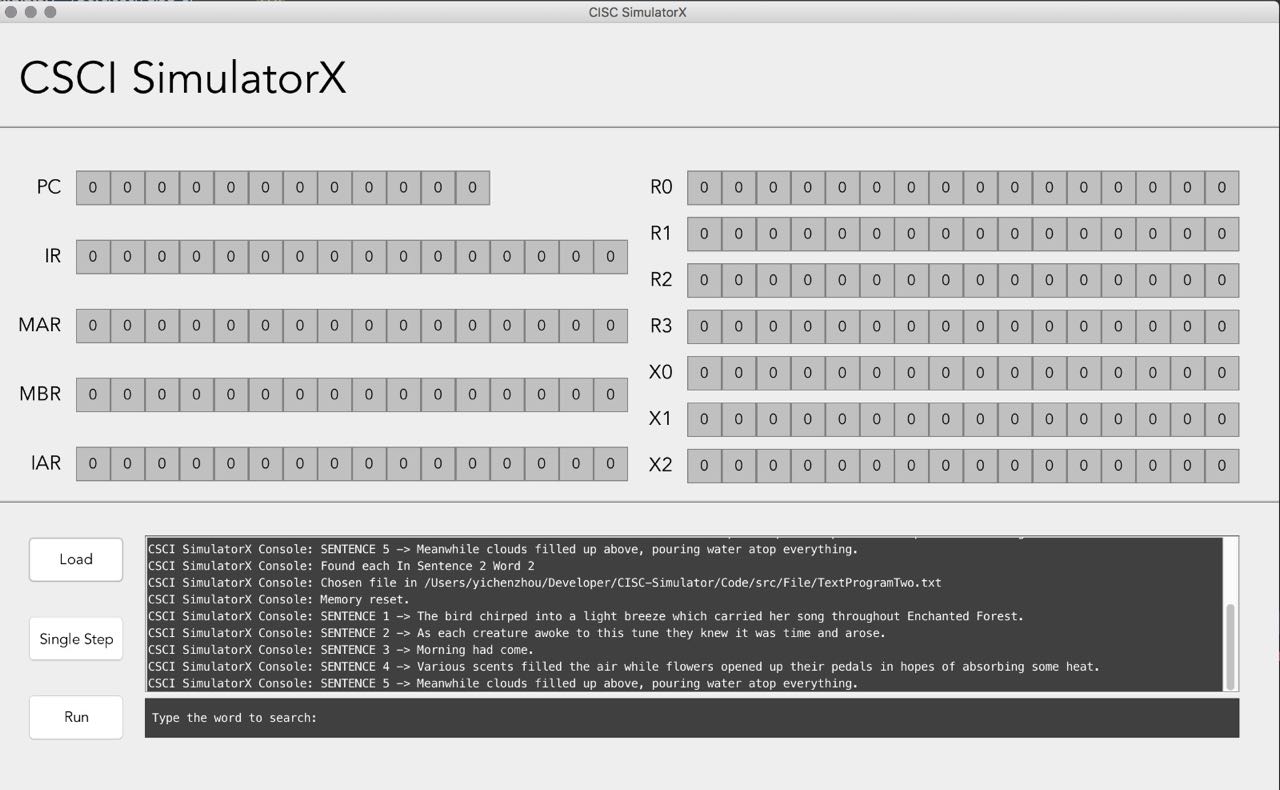
* We created a simple cache between the memory and the rest of the processor that uses a sort queue to store the addresses and the contents of data in memory.
* In each cycle, before searching the memory, we will first search for the address in cache.
* If we cannot find the address, we will then move the search to memory, and put a 16 bits word into cache.
* If the address is found in the cache, we will use the content stored in cache.

## **Part III: Execute All Instructions**

* To fulfill the requirement of program 2 testing, some designs of the UI have been adjusted.
* Program 2 has been added to the pop up menu when clicking on the Load button.
* The results of loading program 2 will appear in the console.
* To search through program 2, we use the input console.



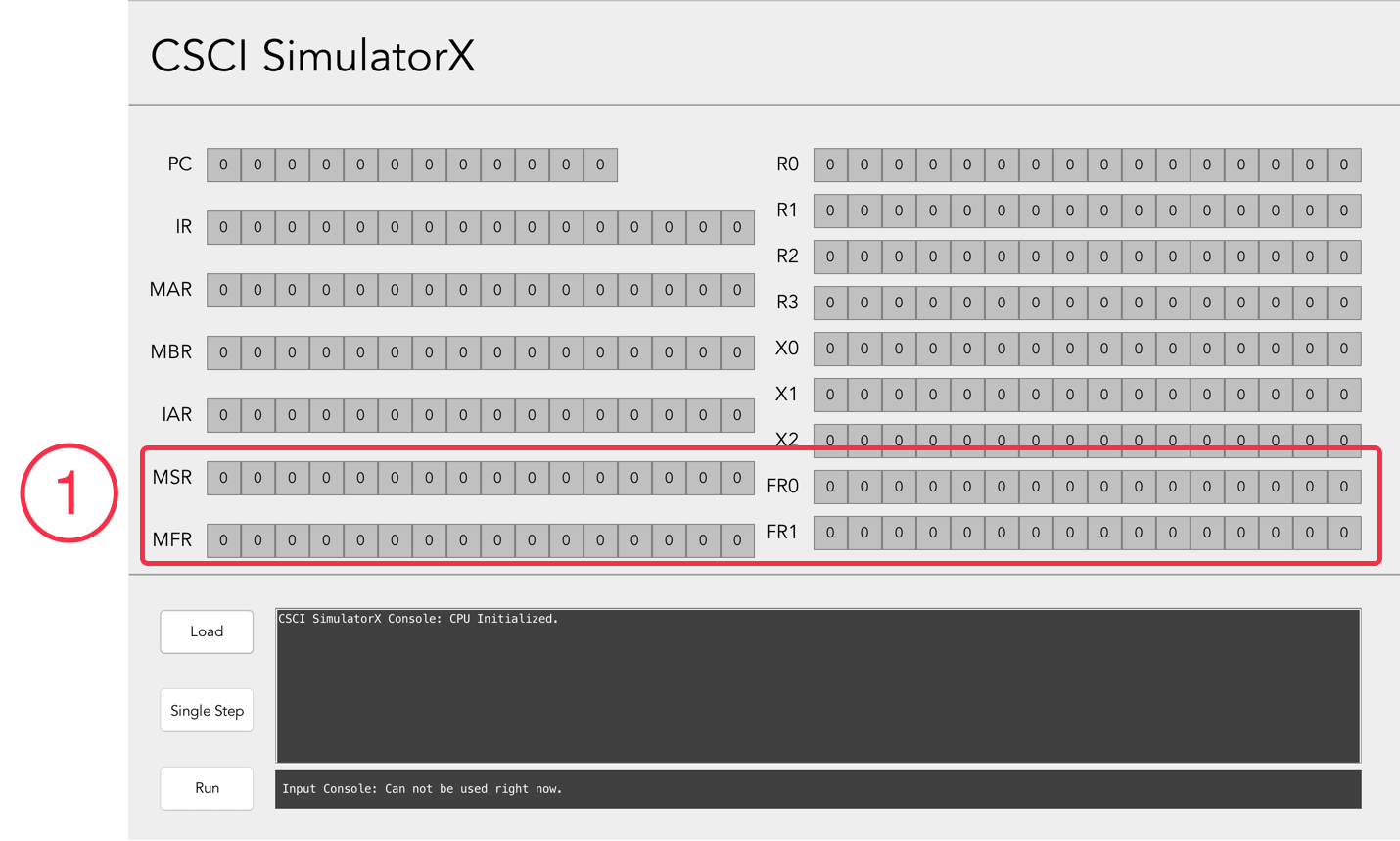
UI adjustments



UI Adjustments II

## **Part IV: Floating Point and Vector Operations**

In this part, we developed floating point and vector operations. Remaining Opcodes were implemented. There are two registers (FR0 and FR1) which store two floating point values. Floating registers Figure shows as below.



We added these files to carry out floating point and vector operation instructions.

|  |  |  |  |
| --- | --- | --- | --- |
| OpCode | Register 1 | Register 2 | Instructions |
| FADD | X0 | FR0 | 100001000001100100 |
| FSUB | X0 | FR0 | 100010000001100100 |
| VADD | X0 | FR0 | 100011000001100100 |
| VSUB | X0 | FR0 | 100100000001100100 |

New classes Pipeline.java and FloatingRepresentation.java were added. A simple Pipeline was implemented by a data structure which is a queue. FloatingRepresentation class performs calculations for the exponent and mantissa components for this program.

## **Version**

|  |  |  |  |
| --- | --- | --- | --- |
| Version No. | Version Date | Author | Description |
| 0.1 | 02-05-2017 | Han Wang, Lulwah AlKulaib | Phase I: Basic Machine |
| 0.5 | 03-04-2017 | Lulwah AlKulaib | Phase II: Memory and Cache Design |
| 0.8 | 03-31-2017 | Lulwah AlKulaib | Phase III: Execute All Instructions |
| 1.0 | 04-40-2017 | Lulwah AlKulaib, Han Wang | Phase IV: Floating Point and Vector Operations |