Manuel Pérez González, Mashaal Al Mazro’ei, Pedro Rios

[Email address]

Abstract

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

CSCI 6461 Semester project

CISC Computer Simulator - System Documentation

# Overview

The CISC Simulator is a project to build a basic Computer Simulator that demonstrates the structure of a computer system, executes instructions and operations. The system emphasizes the Instruction Set Architecture (ISA). The project consists of four fundamental components which are the CPU, Memory, Operators Console and the Engineering Console.

The below documentation is the deliverable of Phase I.

# Scope

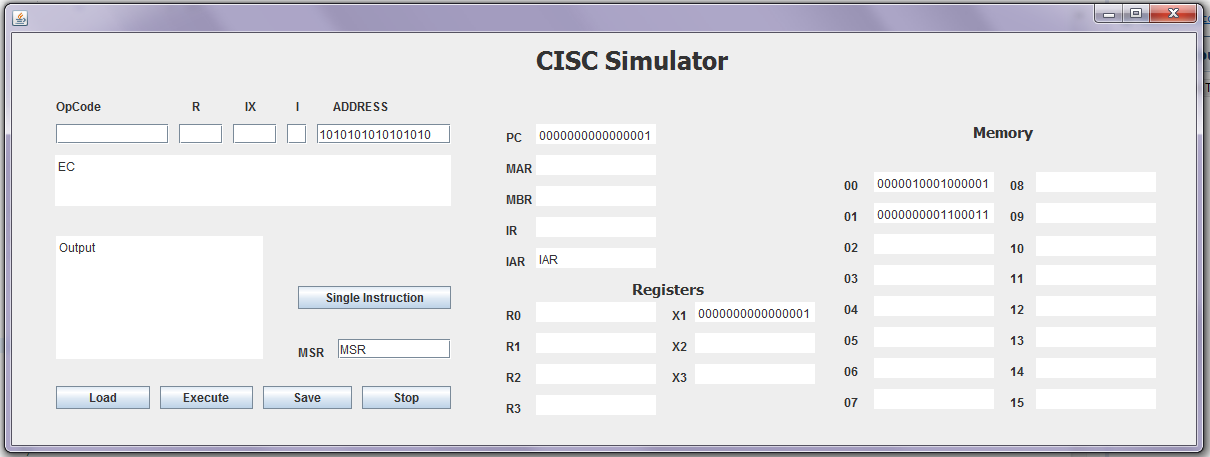
The project is going to be gradually built on four phases starting with the Basic Machine where the Front Panel of the Simulator is designed and implemented along with the Load/Store instructions. Furthermore, in the next phase, the Memory and Cache Operations are going to be implemented. On the following phase, all of the executable instructions are going to be implemented.

# Tools

The Front Panel was initially prototyped using Inkscape (a free and open-source editor to create illustrations and diagrams). The prototype was then developed on Eclipse IDE using Java Swing (an API for providing a graphical user interface (GUI) for Java programs). The LDR/STR instructions were also implemented using Java.

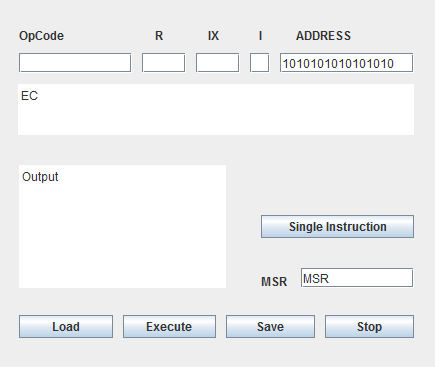
# A Tour of the User Interface

The user interface consists of four main areas; Operator’s Console CPU and Memory. Below is an overview of the Front Panel of the Simulator.



# Operators Console

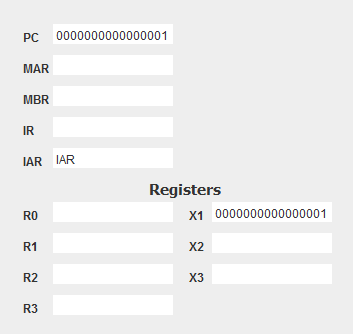
The operators console is the control area for the user to input data, run the programs and display the output. It includes the following components:



* **OPCode Area:** to enter the OPCode to be executed and will display the output.
* **Engineering Console:** o display the status of the simulator during execution.
* **Output Area:** will display the output of the executed program.
* **Single Instruction:** to execute the instruction at the PC address.
* **MSR:** will display the status of the health of the machine.
* **Load Switch:** will load the registers with instructions from the memory using LDR.
* **Execute Switch:** to execute the programs to be implemented. The program is first selected and loaded using the **Load** button, the **Execute** button will run the program.
* **Save Switch:**  store the state of the simulation on a file.
* **Stop Switch**:will Terminate/Halt the program execution.

# Central Processor

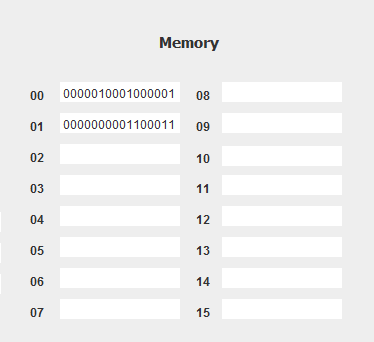
The CPU area contains the Registers and Indexes.



* PC**:** displays the current instruction that is being executed.
* MAR**:** displays the memory address of the data to be fetched.
* MBR**:** displays the data fetched from and to the memory.
* IR**:** displays the currently executing register.
* IAR**:** displays the address of the current executing register.
* Registers R0-R2**:** will be used as General Purpose Registers
* Index Registers X1-X3**:** Registers

# Memory

The memory area contains 16 memory locations.



## Simulator Operation Part I

1. When the simulator runs the simulator will load a hard coded into the PC and one register (memory). (*This was because there are many parts are still not implemented and because of the limited scope of this part*).
2. The **Load** instruction will be executed by clicking the ‘***Single Instruction***’ button.
3. The **Store** instruction will be executed by clicking the ‘***Single Instruction***’ button.

Below is the series of steps that takes place while the simulation is running.

When the Source code is executed, the following operation will run:

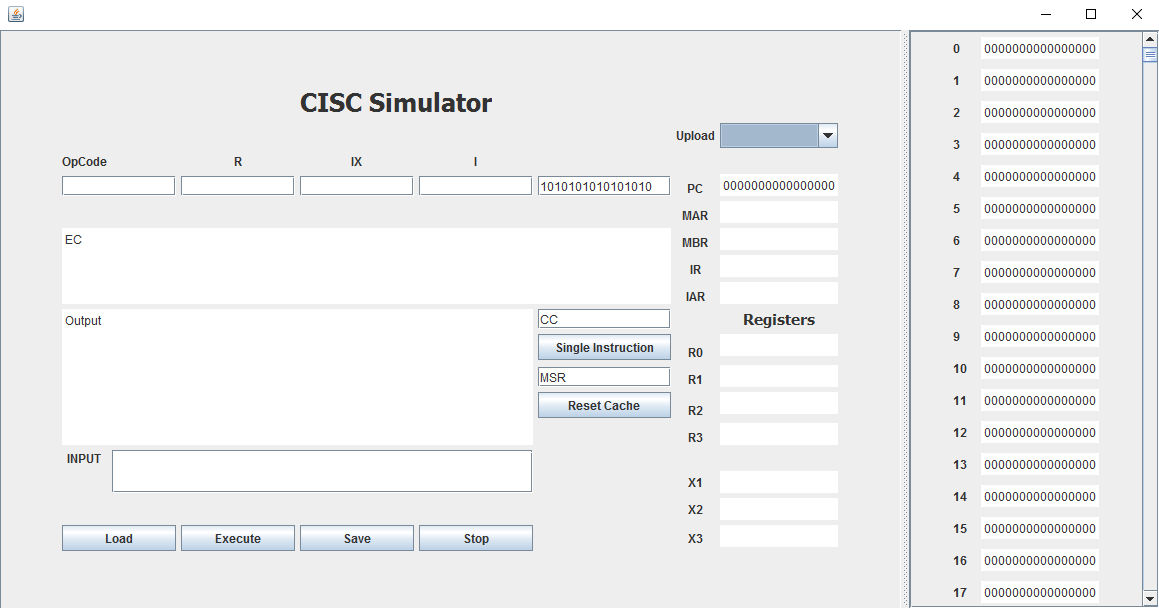
1. The data from Memory 01 and 02 will be fetched and loaded into the MBR and IR
2. The data from the PC will also be fetched and loaded into the MAR.
3. The data from MBR and IR will be stored into the Memory.

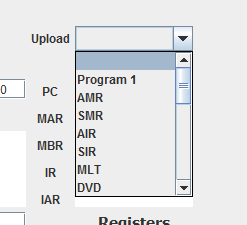
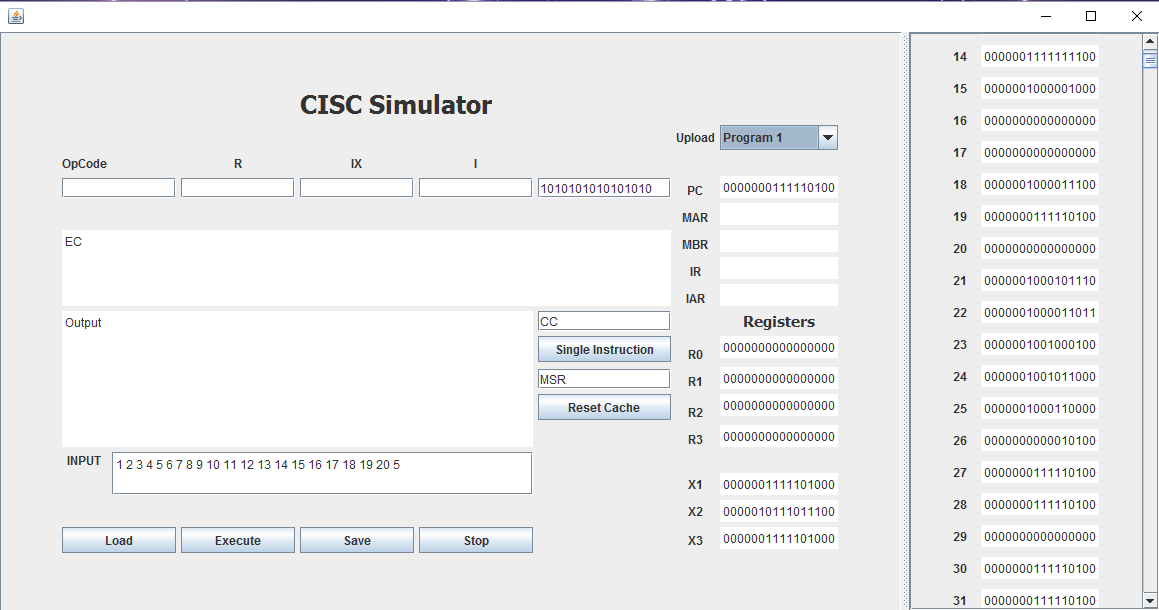
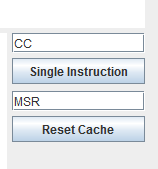
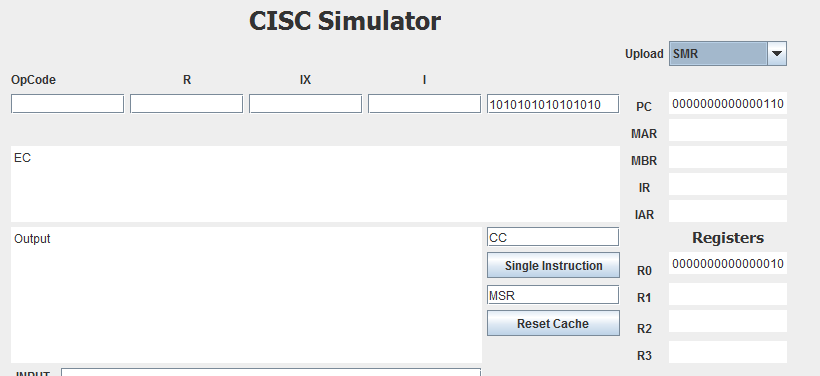
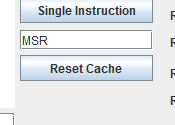
These steps (for this phase) occur when the button ‘Single instruction’ is used to execute the instructions. They also reflect a sequence of steps that can be considered a typical case.

These steps are executed depending on the instruction being processed as in many cases the data in the MBR or IR register does not go directly to Memory but it is used for intermediate steps and thus is saves in one of the general registers (R1-R3) for example.

# Part 2

On this part there has been significant additions to the interface. The main areas are the same as for Part 1 but with enhancements for the user that make easier to test individual instructions or load and execute a program.



1. Main Memory/Cache - The main memory section has been expanded in order to show every memory address location and as part of the implementation of the cache memory.
2. Upload – There is a new ‘Upload’ feature that makes easier to test and load individual instructions into the interface.   
     
     
   **In order to use the ‘Upload’ feature to test an individual instruction or to load and execute a program:**
3. Click on the down arrow on the right side of the field in order to show the available instructions or programs.  
   
4. Once selected the instruction or program will automatically load into the UI.  
     
   
5. In order to execute the program or instruction press the ‘Single Instruction’ button on the bottom left side of the ‘Upload’ section.  
   
6. The display in the UI will update the steps automatically to reflect the results of the instructions with the values that were loaded. The user have the option to enter the values manually for each of the instructions and values using the address field and memory or register/index values for each instruction.   
     
   
7. In the process of using the UI the user also have the option to reset the cache for the current operation by using the ‘Reset Cache’ button.  
     
   

# Part 3

For Part 3 of the Project the UI does not have external changes in term of usability or features as most of the changes are internal.

Features implemented are:

1. Program 2 – From Prof. Lancaster’s Project Description document: “*Program 2: A program that reads a set of a paragraph of 6 sentences from a file into memory. It prints the sentences on the console printer. It then asks the user for a word. It searches the paragraph to see if it contains the word. If so, it prints out the word, the sentence number, and the word number in the sentence.”*
2. TRAP code instruction – The TRAP code instruction is executed if there is an internal issue as a result of executing an instruction. These issues range from accessing a memory location out of range to trying to access the wrong memory location.
3. MSR instruction – The Machine Status Register is basically an indicator that records the status of the machine. The information in the MSR is used in conjunction with the TRAP code to determine how the machine will proceed should it encounter an issue.

Program 2 is available from the ‘Upload’ section as Program 1 on Part 2. Please refer to Part 2 to review the steps.

The difference in Program 2 is that the execution of the program requires that the user enter a word for the program to search. And the paragraph where the word is going to be searched is in the file “TextProgramTwo.txt”. It needs to be in the same folder of the app.jar, so it can find him to run the Program 2. Once the program is loaded the system will prompt the user for a word. Once the user press ‘Enter’ the program will display the results on the output screen.

In the following way: Not found: <word> or Found: <word> S# : <sentence number> W# : <word number in the sequence>.

