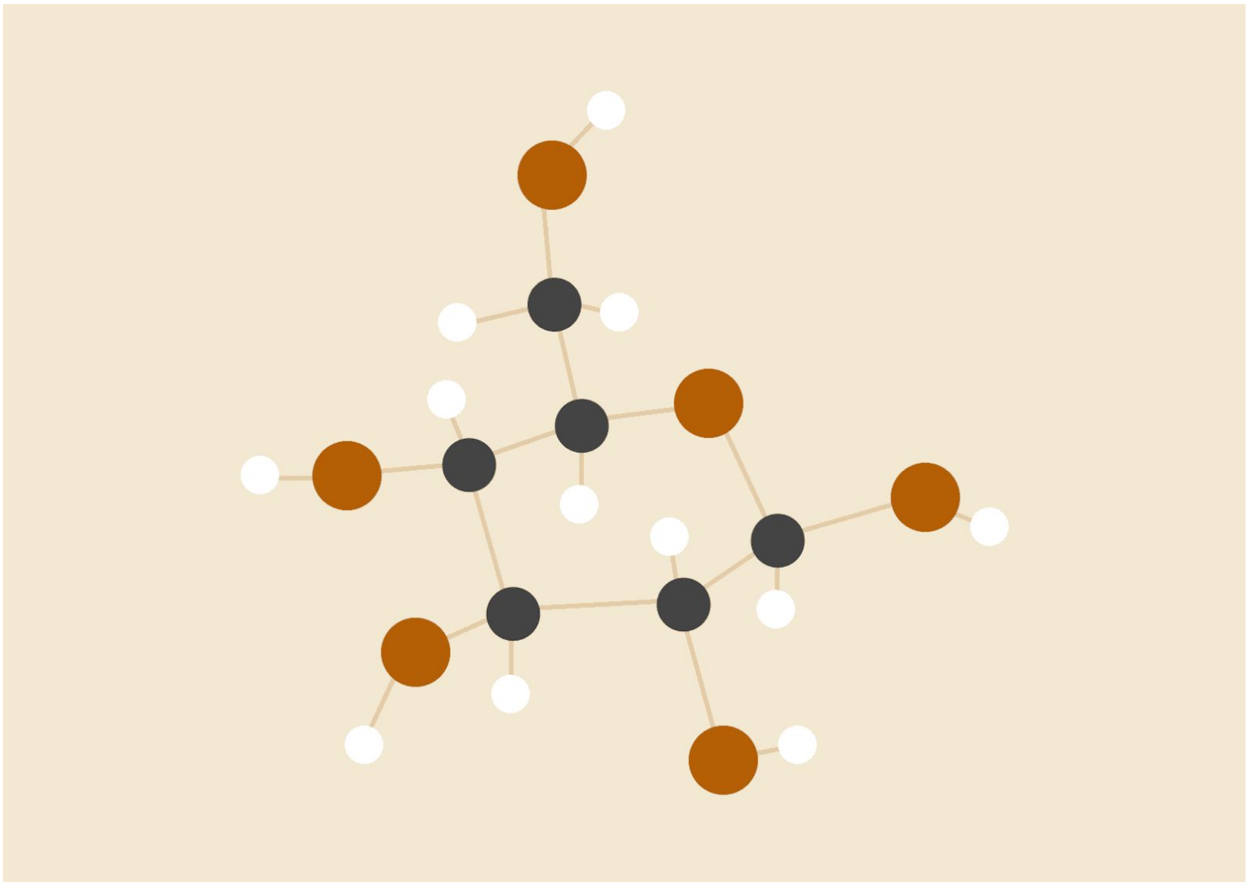
COE403 PROJECT REPORT

M-Architecture assembler/simulator



Team Members	**Faris Hijazi**
	Taha Alsadah
	Rakan Aalsoraye

17/4/2019

COE403

Outline:

Our initial proposal

- what we thought of and how we approached it
- issues faced and solutions we came up with
- Features
    - limitations of the simulator
- list of instructions covered (or we just say all of them if they're all covered)
- ideas that didn't make it to the final product (future expansions, things we didn't have the time to implement/will be implemented later)
- usage (how to use)
- class diagram
- data flow graph (first the file is read, then we clean the lines……)
- contributions
- tools and technologies used

(also check the project proposal on github, most of that stuff can be copied and pasted here)

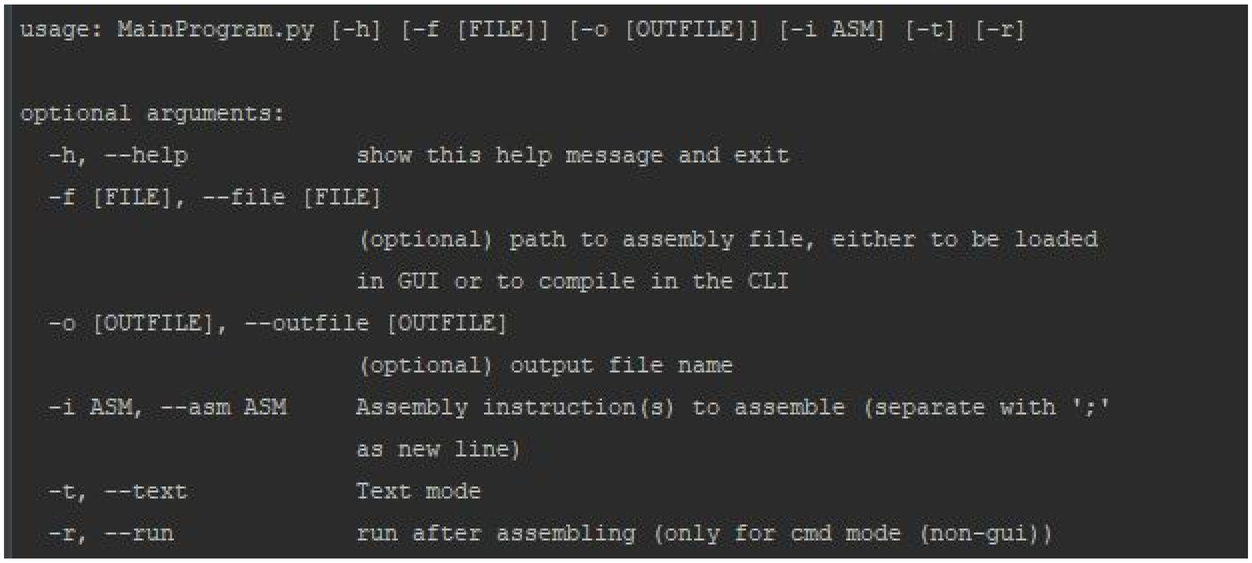
Introduction

Our goal for this course project is to create a software assembler/simulator for the M-Architecture. In this report we will explain our the flow of our code,what we have achieved and our shortcomings.

How to use

<u>**Command Line Interface:**</u>

The code can be ran using the command line interface described below.



```
usage: MainProgram.py [-h] [-f [FILE]] [-o [OUTFILE]] [-i ASM] [-t] [-r]

optional arguments:
  -h, --help            show this help message and exit
  -f [FILE], --file [FILE]
                        (optional) path to assembly file, either to be loaded
                        in GUI or to compile in the CLI
  -o [OUTFILE], --outfile [OUTFILE]
                        (optional) output file name
  -i ASM, --asm ASM     Assembly instruction(s) to assemble (separate with ';'
                        as new line)
  -t, --text            Text mode
  -r, --run             run after assembling (only for cmd mode (non-gui))
```

You can run the entire program in text mode, it will print the memory content and the register file on every change. Also you can pass arguments on what file you start out with, so you wouldn't have to keep loading the file every time you open the program.

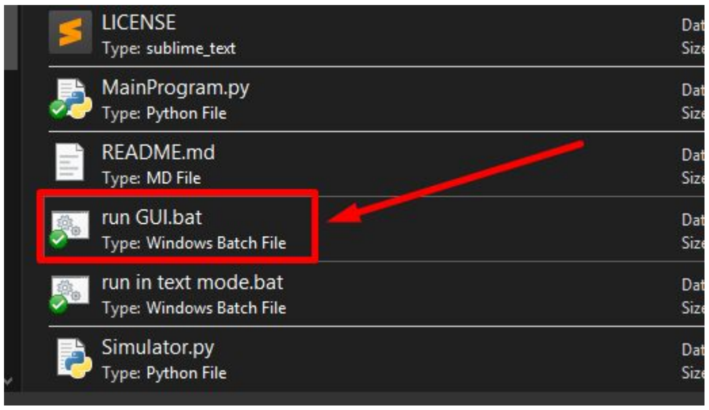
<u>GUI:</u>

The program also has a GUI option which can be ran by typing the following command from the directory of the program file:

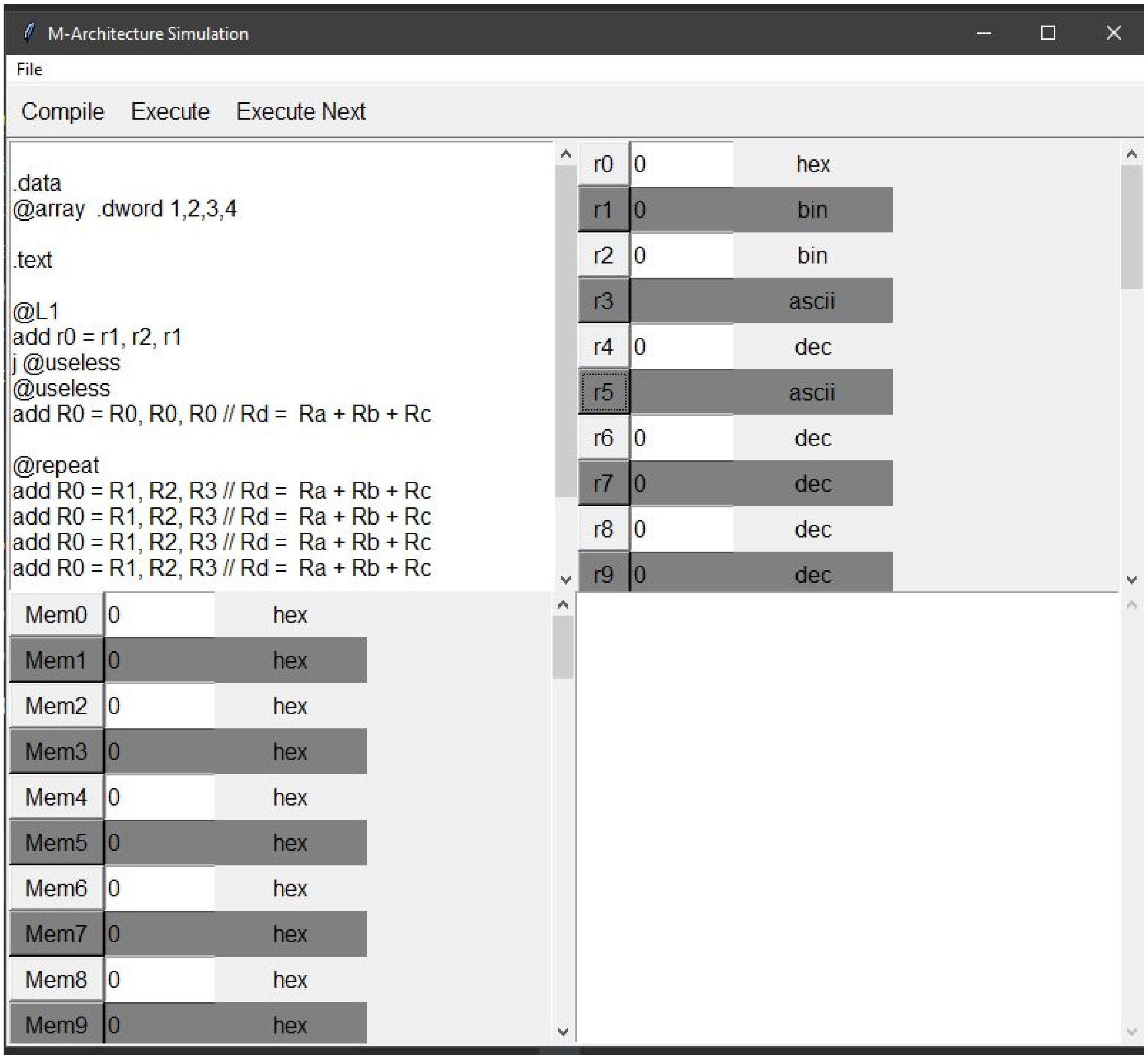
python ./MainProgram.py

OR

By running the file "`run GUI.bat`"



```
LICENSE                                  Dat
Type: sublime_text                       Size
MainProgram.py                           Dat
Type: Python File                        Size
README.md                                Dat
Type: MD File                            Size
run GUI.bat                              Dat
Type: Windows Batch File                 Size
run in text mode.bat                     Dat
Type: Windows Batch File                 Size
Simulator.py                             Dat
Type: Python File                        Size
```



```
.data
@array  .dword 1,2,3,4

.text

@L1
add r0 = r1, r2, r1
j @useless
@useless
add R0 = R0, R0, R0 // Rd =  Ra + Rb + Rc

@repeat
add R0 = R1, R2, R3 // Rd =  Ra + Rb + Rc
add R0 = R1, R2, R3 // Rd =  Ra + Rb + Rc
add R0 = R1, R2, R3 // Rd =  Ra + Rb + Rc
add R0 = R1, R2, R3 // Rd =  Ra + Rb + Rc
```

r0	0	hex
r1	0	bin
r2	0	bin
r3		ascii
r4	0	dec
r5		ascii
r6	0	dec
r7	0	dec
r8	0	dec
r9	0	dec

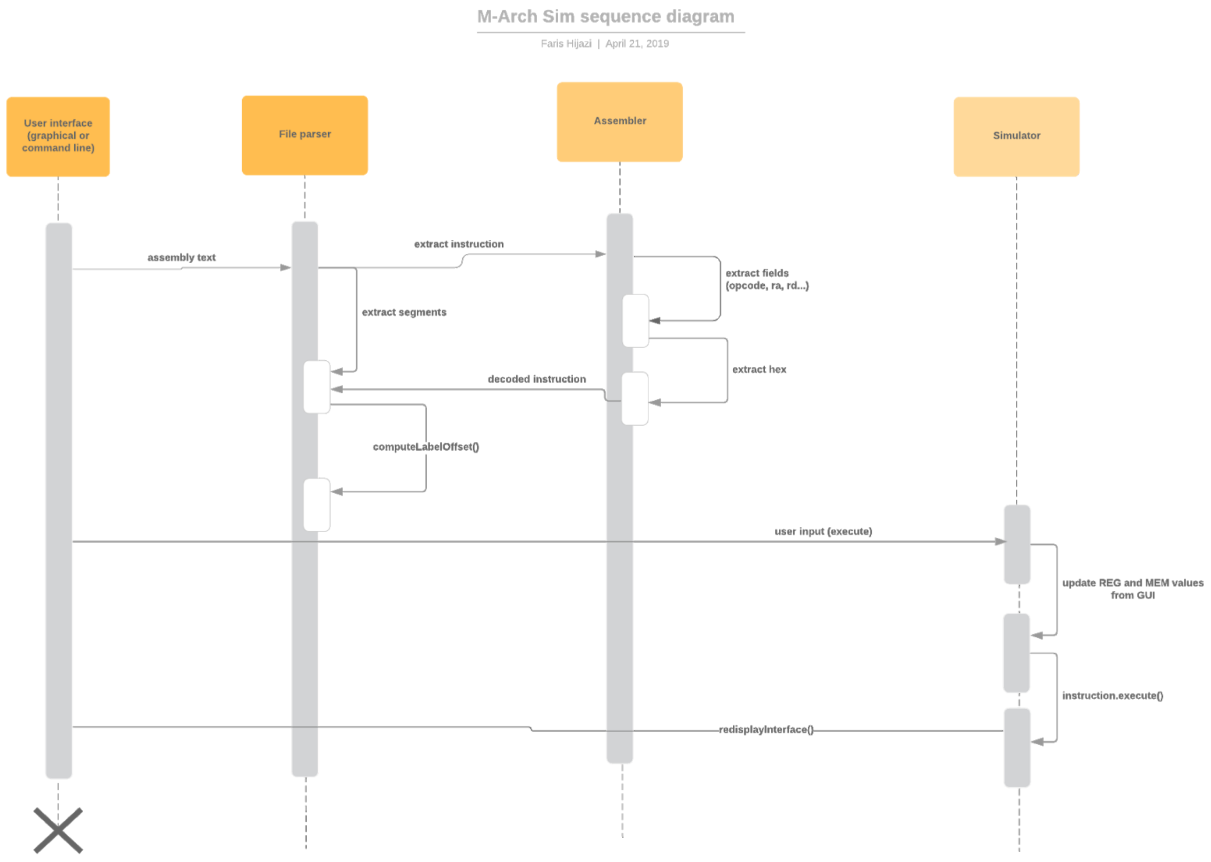
Mem0	0	hex
Mem1	0	hex
Mem2	0	hex
Mem3	0	hex
Mem4	0	hex
Mem5	0	hex
Mem6	0	hex
Mem7	0	hex
Mem8	0	hex
Mem9	0	hex

Sequence and timing

This section gives a brief overview of the sequence or flow (what happens where).

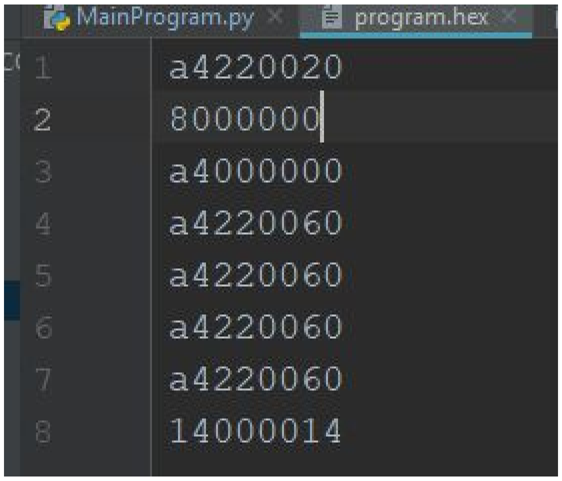
The main tasks are:

- Reading and parsing the file
- Assembling the instructions
- Simulating



The above sequence diagram shows the steps that are taken in the assembler. Note that before executing a single step, the GUI is parsed and the register and memory values are updated, this ensures a consistent experience for the user.

At the end of assembling, 2 files are produced, a hex file containing the decoded instructions as a hex string:

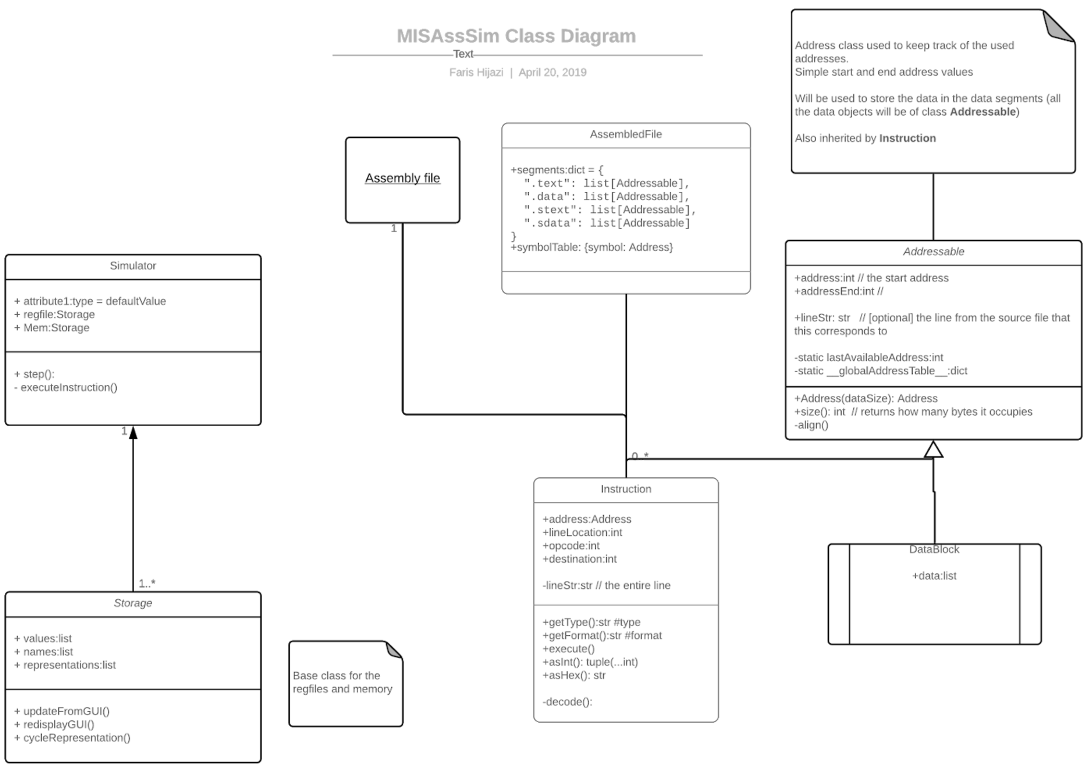


```
a4220020
8000000
a4000000
a4220060
a4220060
a4220060
a4220060
14000014
```

And a binary file which includes the raw hexadecimal values encoded to binary (not string)

Class diagrams and layouts

It is extremely important to plan the entire structure. Below is a class diagram showing the members, fields, and inheritance.


MISAssSim Class Diagram
Text
Faris Hijazi | April 20, 2019
Address class used to keep track of the used addresses.
Simple start and end address values
Will be used to store the data in the data segments (all the data objects will be of class **Addressable**)
Also inherited by **Instruction**
Assembly file
1
AssembledFile
+segments:dict = {
".text": list[Addressable],
".data": list[Addressable],
".stext": list[Addressable],
".sdata": list[Addressable]
}
+symbolTable: {symbol: Address}
Addressable
+address:int // the start address
+addressEnd:int //
+lineStr: str // [optional] the line from the source file that this corresponds to
-static lastAvailableAddress:int
-static __globalAddressTable__:dict
+Address(dataSize): Address
+size(): int // returns how many bytes it occupies
-align()
Simulator
+ attribute1:type = defaultValue
+ regfile:Storage
+ Mem:Storage
+ step():
- executeInstruction()
1
0..*
Instruction
+address:Address
+lineLocation:int
+opcode:int
+destination:int
-lineStr:str // the entire line
+getType():str #type
+getFormat():str #format
+execute()
+asInt(): tuple(...int)
+asHex(): str
-decode():
DataBlock
+data:list
1..*
Storage
+ values:list
+ names:list
+ representations:list
+ updateFromGUI()
+ redisplayGUI()
+ cycleRepresentation()
Base class for the regfiles and memory

The main classes/modules are:

- Assembler
    - The assembler is a helper module with static functions to decode instructions
- Addressable
    - An abstract class to represent anything that contains an address in memory. The class is used with labels, instructions, and *DataBlock* instances (a DataBlock is a block from the *.data* segment, see below).
    - The class keeps a counter of what was the last recently used address, and allows writing of multiple bytes at a time, or a single byte.
    - It also gives the option to **align** the data in memory or not, by defualt, it will align depending on the size of the data element. For example, instructions are aligned to 2^2 (since each instruction is 4 bytes).
    - Also note that the instructions and data are using the same memory space,

effectively making this architecture a Von Nuemann architecture.

- Instruction
  - The instruction class represents a single assembly instruction, for each instruction, an object is created to represent it, the *offset* field is calculated later once the entire file is parsed, this is to allow for detection of labels *after* the current instruction, so the parsing needs 2 sweeps.
    Below is a list of the class members (note that not all of them are used for every single instruction):
    - op
    - rd
    - ra
    - rb
    - rc
    - opcode
    - func
    - rdi
    - rai
    - rbi
    - rci
    - imm
    - p
    - x
    - s
    - n
    - imm_L
    - imm_R

        - Offset

-

Assembler

Our program takes input from the text editor in the gui and decodes it line by line. It checks the mnemonics and extracts the relevant data. It does some error checking, such as checking that the correct number of arguments were passed. The program handles pseudo code and does assembling for all instructions. The result is printed to the terminal.

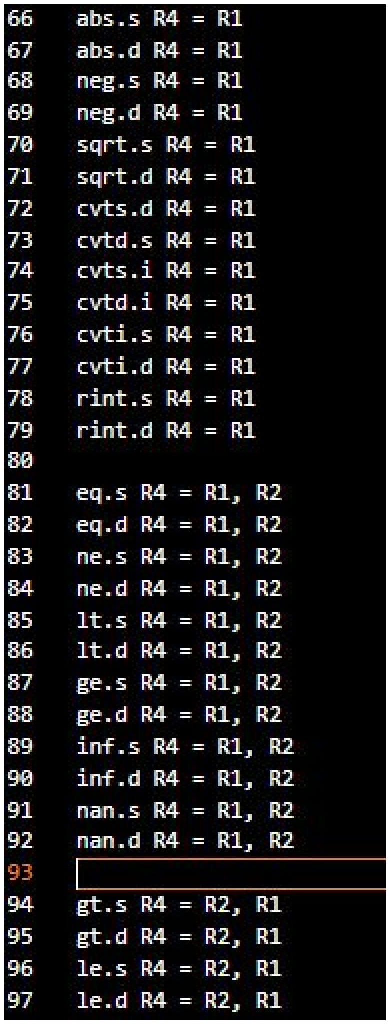
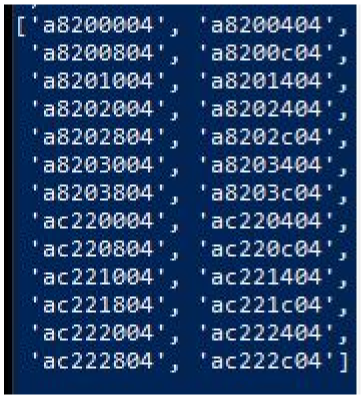
If the line being decoded is a label then the program inserts the newly found label in the symbol table with its corresponding address.

A file named “test_code.txt” contains some instructions that can be put in the gui to check the functionality of the assembler. Click compile once you insert the data in the editor and the hexcode can be found in the terminal.

The assembler differentiates instructions with the same mnemonics based on the number of different parameters passed and checks the type if necessary.

Example:

On the left is user entered text and on the right is the hexcode which can be found in the terminal

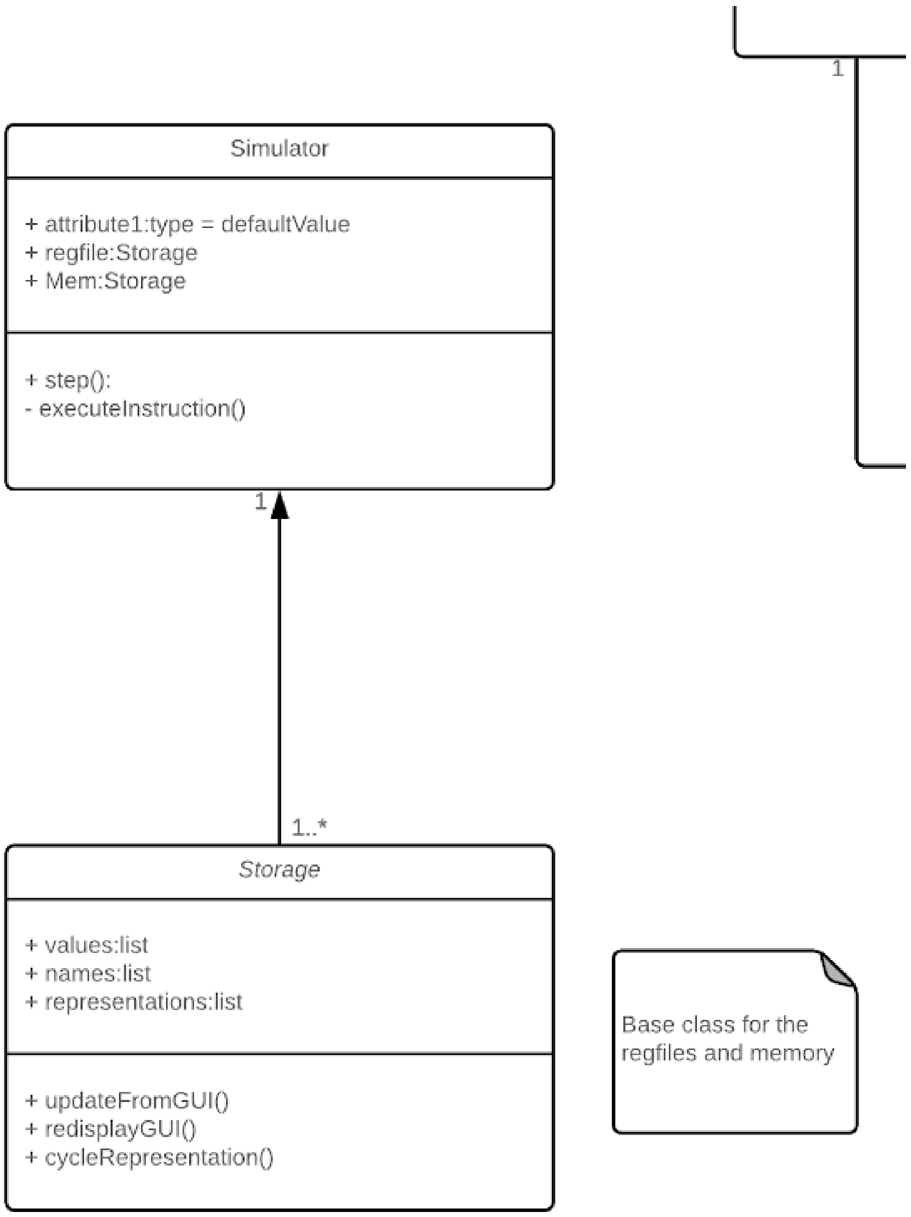

```
abs.s R4 = R1
abs.d R4 = R1
neg.s R4 = R1
neg.d R4 = R1
sqrt.s R4 = R1
sqrt.d R4 = R1
cvts.d R4 = R1
cvtd.s R4 = R1
cvts.i R4 = R1
cvtd.i R4 = R1
cvti.s R4 = R1
cvti.d R4 = R1
rint.s R4 = R1
rint.d R4 = R1

eq.s R4 = R1, R2
eq.d R4 = R1, R2
ne.s R4 = R1, R2
ne.d R4 = R1, R2
lt.s R4 = R1, R2
lt.d R4 = R1, R2
ge.s R4 = R1, R2
ge.d R4 = R1, R2
inf.s R4 = R1, R2
inf.d R4 = R1, R2
nan.s R4 = R1, R2
nan.d R4 = R1, R2

gt.s R4 = R2, R1
gt.d R4 = R2, R1
le.s R4 = R2, R1
le.d R4 = R2, R1
```

```
['a8200004', 'a8200404',
 'a8200804', 'a8200c04',
 'a8201004', 'a8201404',
 'a8202004', 'a8202404',
 'a8202804', 'a8202c04',
 'a8203004', 'a8203404',
 'a8203804', 'a8203c04',
 'ac220004', 'ac220404',
 'ac220804', 'ac220c04',
 'ac221004', 'ac221404',
 'ac221804', 'ac221c04',
 'ac222004', 'ac222404',
 'ac222804', 'ac222c04']
```

Assembling and parsing the file

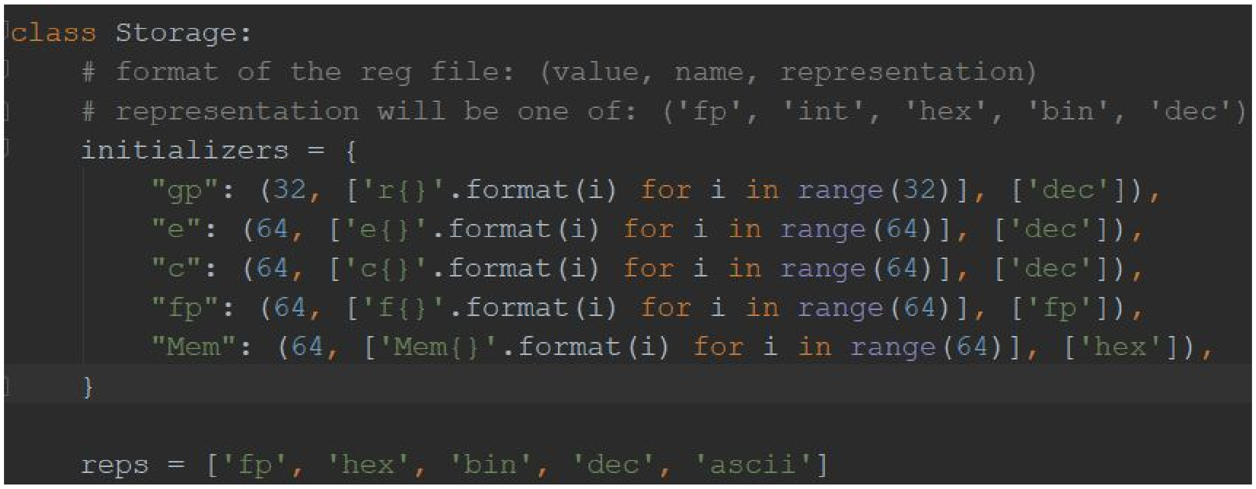
During the assembly of user inputs, instruction objects are also being created and being added to a list. The simulator then takes this list and executes the necessary operation. From the GUI the user can choose to execute the whole program or execute it one line at a time. The user can also change the value of registers by editing them in the GUI.

We have created a class, Regfile, which can be used to created register file objects that can be accessed to retrieve or store information in simulation.

In our simulator we only initialize and use the general purpose registers but we have kept the other types of registers in mind to make future expansions effortless. The programmer can pass the required parameter and start using the newly created Regfile



straight away.

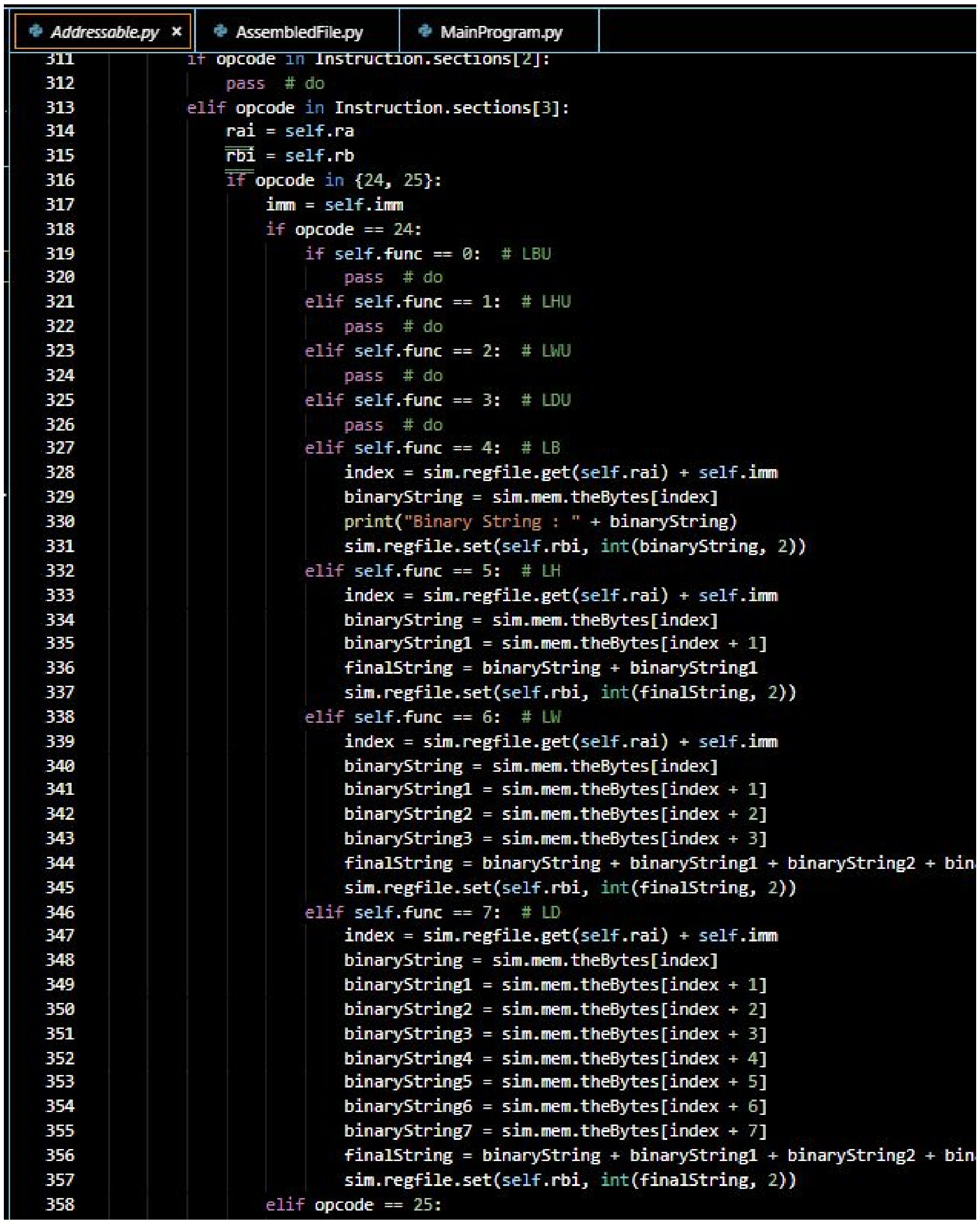


```
class Storage:
    # format of the reg file: (value, name, representation)
    # representation will be one of: ('fp', 'int', 'hex', 'bin', 'dec')
    initializers = {
        "gp": (32, ['r{}'.format(i) for i in range(32)], ['dec']),
        "e": (64, ['e{}'.format(i) for i in range(64)], ['dec']),
        "c": (64, ['c{}'.format(i) for i in range(64)], ['dec']),
        "fp": (64, ['f{}'.format(i) for i in range(64)], ['fp']),
        "Mem": (64, ['Mem{}'.format(i) for i in range(64)], ['hex']),
    }

    reps = ['fp', 'hex', 'bin', 'dec', 'ascii']
```

We have created a *Storage* class. We initialize one memory object for simulation. The memory is byte addressable and we also have a function “write()” to enable the writing of multiple bytes in one instance. For example, This “write()” function can be used in the case we want to write a ascii string to memory, or an array (for example).

Load and Store:

Only signed load is implemented, store is also implemented. Loadu, LoadX and StoreX are not implemented but they can be easily added later within the adressable.py file.



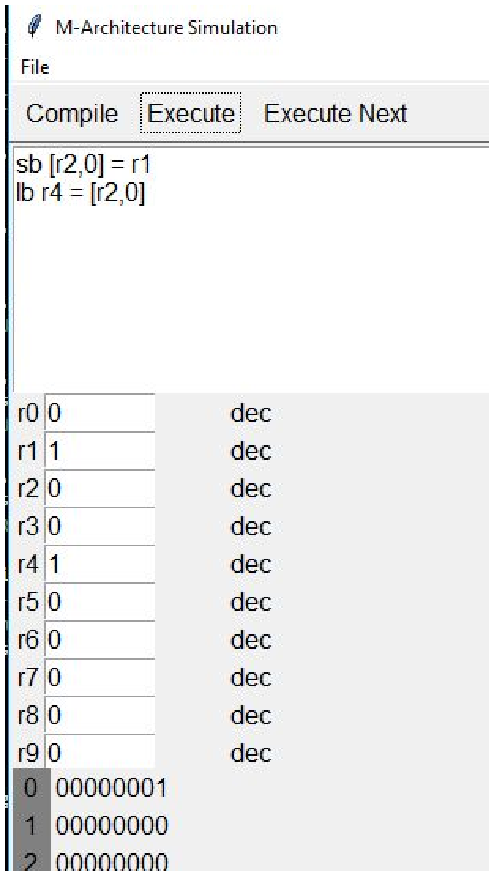
Addressable.py × | AssembledFile.py | MainProgram.py

```
            if opcode in Instruction.sections[2]:
                pass  # do
            elif opcode in Instruction.sections[3]:
                rai = self.ra
                rbi = self.rb
                if opcode in {24, 25}:
                    imm = self.imm
                    if opcode == 24:
                        if self.func == 0:  # LBU
                            pass  # do
                        elif self.func == 1:  # LHU
                            pass  # do
                        elif self.func == 2:  # LWU
                            pass  # do
                        elif self.func == 3:  # LDU
                            pass  # do
                        elif self.func == 4:  # LB
                            index = sim.regfile.get(self.rai) + self.imm
                            binaryString = sim.mem.theBytes[index]
                            print("Binary String : " + binaryString)
                            sim.regfile.set(self.rbi, int(binaryString, 2))
                        elif self.func == 5:  # LH
                            index = sim.regfile.get(self.rai) + self.imm
                            binaryString = sim.mem.theBytes[index]
                            binaryString1 = sim.mem.theBytes[index + 1]
                            finalString = binaryString + binaryString1
                            sim.regfile.set(self.rbi, int(finalString, 2))
                        elif self.func == 6:  # LW
                            index = sim.regfile.get(self.rai) + self.imm
                            binaryString = sim.mem.theBytes[index]
                            binaryString1 = sim.mem.theBytes[index + 1]
                            binaryString2 = sim.mem.theBytes[index + 2]
                            binaryString3 = sim.mem.theBytes[index + 3]
                            finalString = binaryString + binaryString1 + binaryString2 + bin
                            sim.regfile.set(self.rbi, int(finalString, 2))
                        elif self.func == 7:  # LD
                            index = sim.regfile.get(self.rai) + self.imm
                            binaryString = sim.mem.theBytes[index]
                            binaryString1 = sim.mem.theBytes[index + 1]
                            binaryString2 = sim.mem.theBytes[index + 2]
                            binaryString3 = sim.mem.theBytes[index + 3]
                            binaryString4 = sim.mem.theBytes[index + 4]
                            binaryString5 = sim.mem.theBytes[index + 5]
                            binaryString6 = sim.mem.theBytes[index + 6]
                            binaryString7 = sim.mem.theBytes[index + 7]
                            finalString = binaryString + binaryString1 + binaryString2 + bin
                            sim.regfile.set(self.rbi, int(finalString, 2))
                    elif opcode == 25:
```

Bytes are stored in big-endian fashion.

Example:

Loading and storing from memory. R1 was previously set to 1.



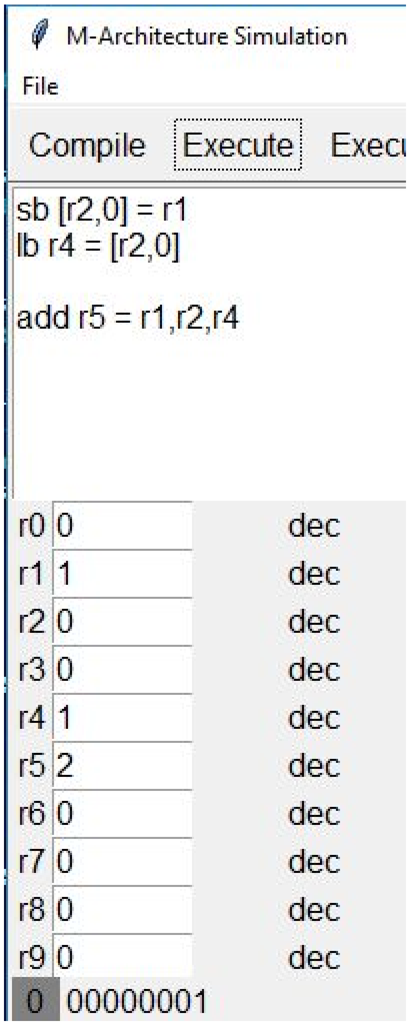
```
sb [r2,0] = r1
lb r4 = [r2,0]
```

```
r0 0    dec
r1 1    dec
r2 0    dec
r3 0    dec
r4 1    dec
r5 0    dec
r6 0    dec
r7 0    dec
r8 0    dec
r9 0    dec
0  00000001
1  00000000
2  00000000
```

Section 5: Three source registers is implemented.

Example:

R1 was previously set to 1.



```
sb [r2,0] = r1
lb r4 = [r2,0]

add r5 = r1,r2,r4
```

```
r0 0          dec
r1 1          dec
r2 0          dec
r3 0          dec
r4 1          dec
r5 2          dec
r6 0          dec
r7 0          dec
r8 0          dec
r9 0          dec
0  00000001
```

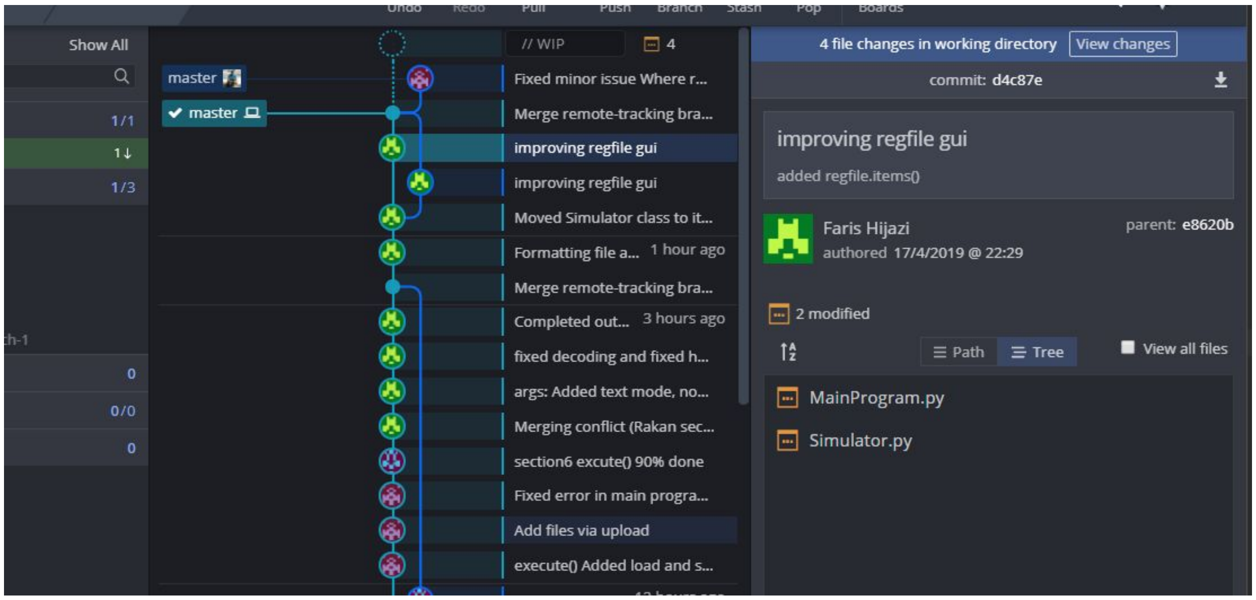
Tools used

Our code is written in python.

Github for version control.

Pycharm - python IDE

Gitkraken a git client



// WIP 4

Fixed minor issue Where r...

Merge remote-tracking bra...

improving regfile gui

improving regfile gui

Moved Simulator class to it...

Formatting file a... 1 hour ago

Merge remote-tracking bra...

Completed out... 3 hours ago

fixed decoding and fixed h...

args: Added text mode, no...

Merging conflict (Rakan sec...

section6 excute() 90% done

Fixed error in main progra...

Add files via upload

execute() Added load and s...

4 file changes in working directory View changes

commit: d4c87e

improving regfile gui

added regfile.items()

Faris Hijazi
authored 17/4/2019 @ 22:29
parent: e8620b

2 modified

MainProgram.py

Simulator.py

PROCEDURE

1.

Contributions

Name	Contribution
Faris Hijazi	● Planning and managing, designing layouts of classes and how the program will work, integrating code across members ● Creating the command line interface ● Parsing code (detecting data segments and detecting labels and instructions), and anything related to parsing/reading text in the files ● GUI buttons for changing representations ● Creation of diagrams and designing the program classes ● Report 40% ● Total hours put into this project: 50+
Taha Alsadah	● Assembler: Sec 2,3,5,6. ● Simulator execution: Sections 3,5, and memory. ● Responsible for GUI
Rakan Aalsoraye	● Assembler: Sec 4. ● Simulator execution: Sections 4,6.

RESULTS

We were able to complete most of the project, due to time constraints (mainly bad time management), not all features work as needed by the time of the deadline.

The objective was to make the code extensible (and it is, only minor changes are needed to add a new instruction)

In addition, the flexibility of our code, gives it a solid advantage of adding many other features. We intend to improve the code a lot especially the gui, it doesn't reflect how powerful our code is.

What works:

- Almost all instructions, regular and pseudo, are accounted for
- Catching all types of errors
- Recognizing capital and small letters

- Custom alignment
- You can manually change registers/memory values even at runtime (using the interface)
- The gui displays memory and registers correctly

What doesn't work:

- SYSCALL
- Branch Instructions
- Single/double precision operations are treated the same way
- Rotating instruction

Note: everything that doesn't work has its own area within the code to be done later.