README FILE (Phase 1)

Group-6

Group members:

1. Abhishek Choudhary (2019CSB1061)
2. Pradeep Kumar (2019CSB1107)
3. Nitish Goyal (2019CSB1103)
4. Himanshu Yadav (2019CSB1263)
5. Deepan Maitra (2019CSB1044)

Welcome to the phase 1 of the RISC-V simulator. In this phase, we have tried to simulate the working of Machine Code execution in a non-pipelined approach. The code can be found as `GUI_Phase1.py` (for explicitly seeing the non-GUI version, see `non_GUI_Phase1.py`)

INPUT TO CODE: A file with the machine code (here it is `code.mc`). This file has the machine code divided into two segments: data segment and text segment.

```
0x0    0x10000197
0x4    0x0001A183
0x8    0x10000217
0xc    0xFFC22203
```
Text segment (instructions)

```
0x10000008  0x17020010
0x10000004  0x83A10100
0x10000000  0x97010010
```
Data segment

OUTPUT OF CODE:

In console: Displays the **5 step execution** of each instruction, by displaying the **fields** (*opcode, rs, rd, func3, func7, imm*) of each instruction when applicable. Also, displays the **register values** and **memory state** before and after execution. In order to check successful completion of the code, the user has to check necessary memory and register states at the end of execution.

In GUI simulator window: The register states can also be checked using the display GUI window. (To directly go to the process to use this simulator, jump to ***HOW TO USE*** part of this document)

After execution, all the data memory is written in the `memory.mc` file before the program terminates.

INSTRUCTIONS SUPPORTED BY OUR CODE:

- R format - add, and, or, sll, slt, sra, srl, sub, xor, mul, div, rem
- I format - addi, andi, ori, lb, lh, lw, jalr
- S format - sb, sw, sh
- SB format - beq, bne, bge, blt
- U format - auipc, lui
- UJ format – jal
- This code ***DOES NOT*** support any pseudo instruction.

BRIEF OUTLINE/LOGIC OF THE CODE:

1. There has been OOP implementation in Python 3. The code contains 4 classes: `class execute`, `class register`, `class memory`, `class GUI`

2. `class execute`: This is the driving class of the code. An object is created of this class in the method called `programRun()`, where the machine code is read and successive methods `run()` is called for each decoded instruction. This class also defines methods like `storeInstruction()` (stores each instruction in the memory after reading them, so as to load them one after another in the IR), `fetch()` (fetches the instruction from the memory), `checkFormat()` (decides which format the instruction belongs to), `decode()` (based on format, checks the exact information based on the extracted fields), `memoryAccess()` (the 4th step of execution cycle where values can be stored or loaded from the memory module), `writeRegister()` (the writeback stage of the execution cycle for register value update). In all, this class simulates the whole 5-stage execution cycle. This class also creates instances of the `memory` and `register` classes, to simulate the *Register File* and the *Memory module*

3. `class register:` This class has the methods which are needed for writing and reading the current register values. An instance of this class will behave like a *Register File* which is accessed in the decode and writeback stages of the 5-stage execution cycle.

4. `class memory:` This class has the methods which facilitate the method called `memoryAccess()` which is in charge of reading to and from the memory module, and also storing values in it. An instance of this class will be needed in any instruction that requires the 4th step of the 5-step cycle (memory access stage)

5. `class GUI:` This class is the one that enables the GUI part of the code. Essentially, in the output display window, there will be 32 labels for the 32 registers, each with text fields that contains the stored value.

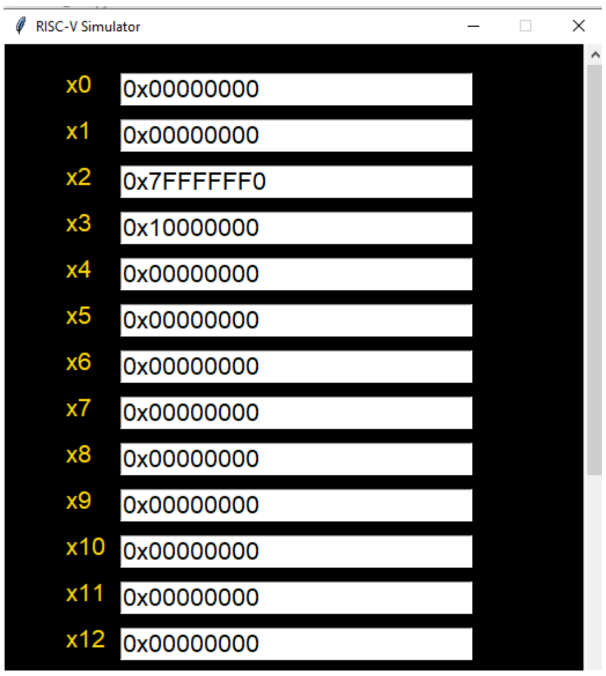
6. The memory and registers are simulated using ***dictionary*** in the Python environment. Dictionary has *keys* and *values*, similar to a list/array. The values (here the stored values) can be accessed by the keys (here, the addresses in memory, or the register numbers in register file)

7. All the instructions are encoded in 32-bit format in `code.mc`, and the memory module is ***byte-addressable*** and ***Big Endian***. Therefore, to read and write to memory, different functions for word, byte etc. has been defined. After execution, all the data memory is written in the `memory.mc` file before the program terminates.

HOW TO USE THE SIMULATOR?

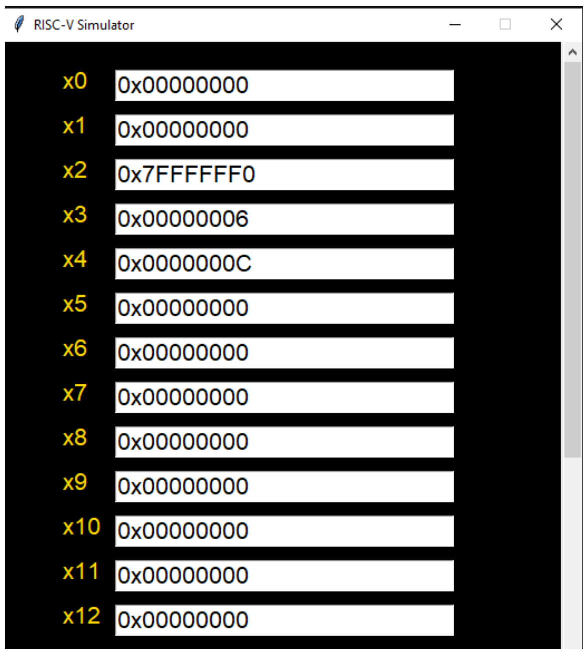
The Simulator has an in-built GUI feature to make the code more user-friendly. The simulator shows the register values, before and after the execution. (Please install `bitstring module` in Python before running this code) The following steps will show how to use the simulator.

Step 1) Run `GUI_Phase1.py` on the console. A display window will pop-up, that lists the default values of registers x0-x31. (Note that the stack pointer(x2) and x3 are pre-loaded with their default values.



```
x0   0x00000000
x1   0x00000000
x2   0x7FFFFFF0
x3   0x10000000
x4   0x00000000
x5   0x00000000
x6   0x00000000
x7   0x00000000
x8   0x00000000
x9   0x00000000
x10  0x00000000
x11  0x00000000
x12  0x00000000
```

Step 2) On waiting for **5 seconds**, the simulator window will close by itself and then pop-up again. The new window will list the registers x0-x31 with their final values (i.e. after the code execution) Note that the user doesn't have to perform any specific operation on the simulator, just has to let the window remain idle for 5+ seconds) **The output can also be seen as text in the console.**



```
x0   0x00000000
x1   0x00000000
x2   0x7FFFFFF0
x3   0x00000006
x4   0x0000000C
x5   0x00000000
x6   0x00000000
x7   0x00000000
x8   0x00000000
x9   0x00000000
x10  0x00000000
x11  0x00000000
x12  0x00000000
```

For example,

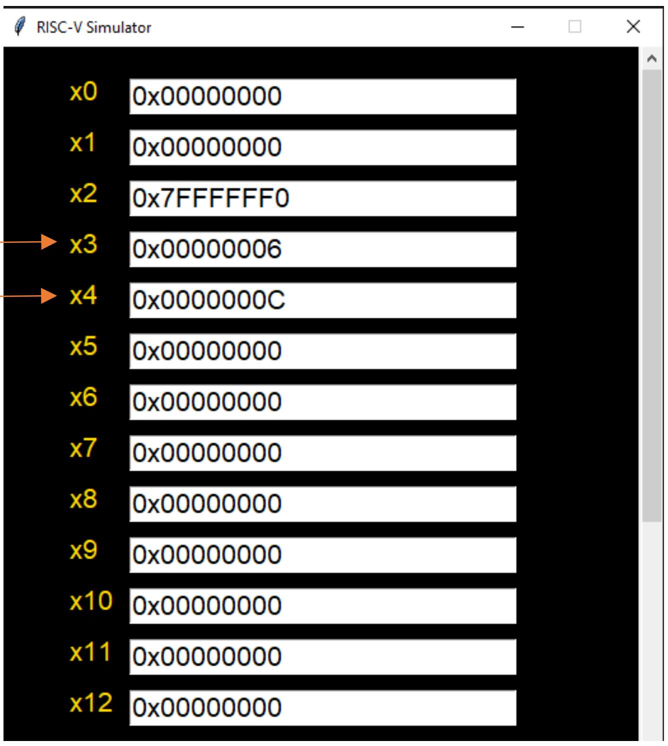
Suppose `code.mc` has the following code written in machine format. The actual instruction also has been shown here to understand.

```
0x0   0x00C00213
0x4   0x00800193
0x8   0x00321463
0xc   0x00500293
0x10  0xFFE18193
```

```
addi x4, x0, 12
addi x3, x0, 8
bne x4, x3, label
addi x5, x0, 5
label: addi x3, x3, -2
```

As an output to the above code, register x4 should have value 12 (0x0000000C) and x3 should have value 6 (0x00000006). Hence, the simulator shows:

Register values updated!



```
x0   0x00000000
x1   0x00000000
x2   0x7FFFFFF0
x3   0x00000006
x4   0x0000000C
x5   0x00000000
x6   0x00000000
x7   0x00000000
x8   0x00000000
x9   0x00000000
x10  0x00000000
x11  0x00000000
x12  0x00000000
```