

# Project Report

Assembly

**Ahmed Jaheen - 900212943**

**Adham Samy - 900213258**

**Abdullah Mahmoud - 900193567**

*Submitted to Dr. Nourhan Zayed*

*This assignment is prepared for CSCE2303, section 2*



**THE AMERICAN  
UNIVERSITY IN CAIRO**

**COMPUTER SCIENCE AND ENGINEERING DEPARTMENT  
THE AMERICAN UNIVERSITY IN CAIRO**

11/05/2023

# Implementation Brief Illustration

Our implementation consisted primarily of four components: parsing the input, executing the instructions, handling errors, and producing the output.

(i) **Parsing the input:**

The input is given mainly through two files, a memory file and instructions file (**Project.txt**, **Memory.txt**). The instructions file is used to store all the instructions in a vector of strings (**vector<string> instructions**) line by line to handle each instruction separately. The memory file takes the input as two columns of values then addresses separated by a space. Then, it stores both of them in two vectors of integers (**vector<int> values**, **vector<int> addresses**). We took care of the spaces that may be there in the input using function **removeemptylines**.

(ii) **Executing the instructions:**

After filling the vectors with the code and the data from the memory, we start from the address of the first instruction, given by the user. Then we access each instruction line by line in the main function to identify the type of the instruction and pass it to the appropriate function to handle it. This function then does the logic behind the instruction and change the PC accordingly. Lastly, we execute the new instruction pointed by PC until one of the three halting instructions were met.

(iii) **Handling the errors:**

We kept track of handling many errors that occur from the user, such as:

- (a) We remove empty lines from the instructions.
- (b) It can handle any number of realistic instructions for most of the known code.
- (c) If you input the wrong number of parameters for a specific instruction, it breaks.
- (d) If you input any unidentified instruction, it output you the line which has the error.
- (e) If you input the memory file with unsorted address, it will handle that and sort them before starting to work.
- (f) It handles having spaces before the instructions and between the variables of a single instruction.

(iv) **Producing the output:**

After executing each instruction, we print the PC, the names of the 32 registers with the values stored in them in Decimal, Binary, and Hexadecimal forms. Moreover, we print the data stored in the Memory with its addresses sorted in an ascending order

# Bonus Features

- (i) Outputting all values in decimal, binary, and hexadecimal (instead of just decimal which is assumed to be the default).
- (ii) Add support for integer multiplication and division to effectively support the full RV32IM instruction set.
- (iii) Including a larger set of test programs (at least 6 meaningful programs) and their equivalent C programs.

# Design Decisions and Assumptions

## 1. **Architecture of the Memory:**

In our memory design, we have opted to store instructions and data in separate components. Due to the large size of the memory, we have implemented it using three vectors: one for instructions and two for data (values and addresses). Furthermore, we have chosen to allow the user to initialize data in memory addresses that are divisible by 4. This decision simplifies the process of loading data from or storing data into memory, ensuring ease of use. Additionally, it guarantees that the user will not inadvertently overwrite different values in the same memory address.

## 2. **Modular Structure:**

To enhance code organization and manage the extensive number of lines, we adopted a modular approach by implementing each instruction as a separate function with its own designated name, such as `branchiflessthan()` and `xorimmediate()`. This design choice allowed us to handle the complexity of the program more effectively. By encapsulating each instruction within its own function, we achieved improved code modularity and enhanced our ability to manage and maintain the codebase.

## 3. **Case Sensitivity work:**

In accordance with the case sensitivity conventions employed by RARS, we adhered to the following assumptions: labeling of labels is case sensitive, meaning that "Main" is considered distinct from "main". However, when it comes to instruction words like "addi", case sensitivity is not taken into account. Therefore, the instruction words are considered case insensitive, allowing for flexibility in their usage regardless of letter casing.

## 4. **Registers:**

The 32 registers are represented by an array with a length of 32. At the start of the program, all registers are initialized to zero. As the instructions are executed, the values of the registers are modified accordingly. During the execution of each instruction, we verify whether the destination register is x0 (register zero) or not. If the destination register is x0, we prevent any write operations into it, thus preserving its initial value of zero. This precaution ensures that the register x0 retains its designated purpose and avoids unintended modifications that could impact the program's functionality.

## 5. **Handling the errors:**

Our program includes extensive error handling to validate all aspects before executing the program. This ensures reliability and minimizes potential issues.

## 6. **Assumptions:**

- (a) Our program does not support comments. Therefore, any lines or sections intended for comments should be omitted from the input files to ensure proper

execution and avoid any potential errors or conflicts.

- (b) Instruction words are not case-sensitive, but labels are.
- (c) There must be starting address for the memory.
- (d) The memory format is of the form that the first number contains the value stored in the address, while the second number is the address.
- (e) If the starting memory address is not divisible by 4, the code will crash.

# Bugs and Issues

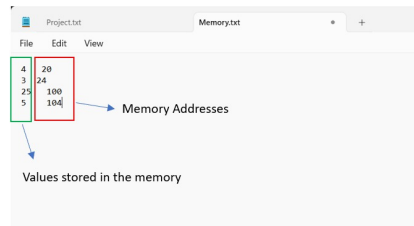
Programme execution crashes due to the missing "**Memory.txt**" file, which provides the program's memory. Without this file, the programme cannot access and retrieve data, causing a fatal error and crash. Moreover, the register labels must have small case letter, these issue causes the program not to identify the registers that is written in the **Project.txt** file. The system will go to infinite loop of memory calling if you don't write one of the exit instructions.

# User Guide

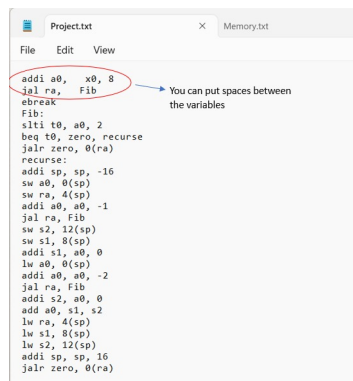
The project is written in C++, allowing you to compile it using any C++ compiler. Once the compilation process is complete, you have to edit **Program.txt** with your implemented RISC-V instructions and edit **Memory.txt** with your memory addresses. You always have to write a memory file even if you will initialize the registers in the instructions file. Also, you have to make sure that the **Program.txt** and the **Memory.txt** are present in the same directory as the .cpp file.

To initiate the program, follow these steps:

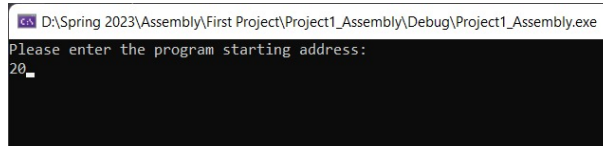
1. Firstly in the memory file, you write two numbers per line; the first number hold the value stored in the memory, and the second number hold the memory address of this value. You can have any number of spaces between the value and its memory address as our code handle it.



2. Secondly in the instructions file, you can write any number of instructions. However, you have to be aware that you cannot write a space before any instruction or have a syntax error as it will not give an output, but it will let you know the line with the error. You can put any number of spaces between the registers. Also, you can use the registers from x0 to x32 or the other definitions of (zero, ra, sp, gp, tp, t0-t6, a0-a7, s0-s11).

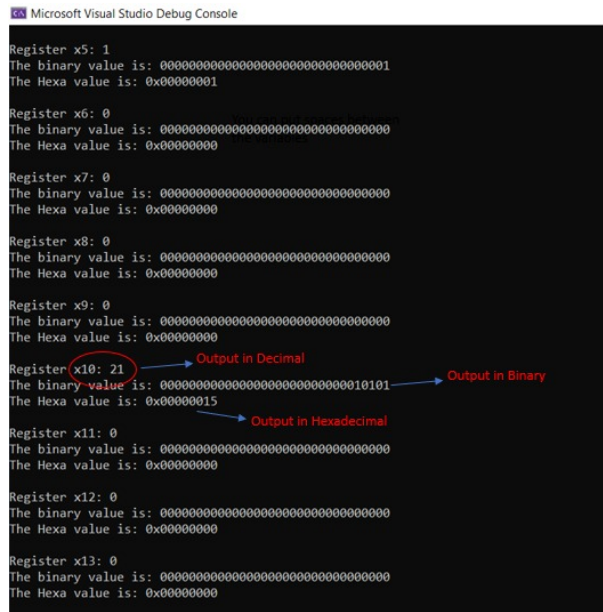


3. Thirdly, when you open the program, you will be prompted to enter the starting address of the program as a decimal integer. Once you have entered the value, press the Enter key. As an example, I have entered the value 20 in this case.



```
D:\Spring 2023\Assembly\First Project\Project1_Assembly\Debug\Project1_Assembly.exe
Please enter the program starting address:
20_
```

4. Fourthly, For each instruction, you should expect to receive output according to the specified format, which will look like this.



```
Microsoft Visual Studio Debug Console

Register x5: 1
The binary value is: 00000000000000000000000000000001
The Hexa value is: 0x00000001

Register x6: 0
The binary value is: 00000000000000000000000000000000
The Hexa value is: 0x00000000

Register x7: 0
The binary value is: 00000000000000000000000000000000
The Hexa value is: 0x00000000

Register x8: 0
The binary value is: 00000000000000000000000000000000
The Hexa value is: 0x00000000

Register x9: 0
The binary value is: 00000000000000000000000000000000
The Hexa value is: 0x00000000

Register x10: 21
The binary value is: 000000000000000000000000000010101
The Hexa value is: 0x00000015

Register x11: 0
The binary value is: 00000000000000000000000000000000
The Hexa value is: 0x00000000

Register x12: 0
The binary value is: 00000000000000000000000000000000
The Hexa value is: 0x00000000

Register x13: 0
The binary value is: 00000000000000000000000000000000
The Hexa value is: 0x00000000
```

Annotations in the image:

- A red circle highlights "x10: 21". A blue arrow points from it to the text "Output in Decimal".
- A red circle highlights "10101" in the binary value of Register x10. A blue arrow points from it to the text "Output in Binary".
- A red circle highlights "0015" in the hexa value of Register x10. A blue arrow points from it to the text "Output in Hexadecimal".



# List of Programs

The list of programs that are provided with our project are:

- (i) **Fibonacci:** This program is designed to calculate and find the nth element in the Fibonacci series, given the value of n.
- (ii) **Sum of element in an array:** This program is designed to calculate the sum of elements within an array. Given the head address and length of the array, the program traverses through the array elements, adding each value to a running total.
- (iii) **Minimum integer in an array:** This program is designed to identify the smallest element within an array. Given the head address and length of the array, the program executes a search algorithm to determine the smallest value present.
- (iv) **GCD:** This program is specifically developed to find the greatest common divisor between two positive numbers. By executing the program, it applies an algorithm to identify the largest positive integer that divides both numbers without leaving a remainder.
- (v) **Sort an array:** This program is sorting an inputted array.