**Design Details: Functional Simulator for Subset of 32 bit RISC V instruction set**

The document describes the design aspect of RISC V, a functional simulator for subset of RISC V instruction set.

# Input/Output

## Input

Input to the simulator is .MC file contains the encoded instruction and the corresponding address at which the instruction is supposed to be stored, separated by space. For example:

0x0 0xE3A0200A

0x4 0xE3A03002

0x8 0xE0821003

## Functional Behavior and output

The simulator reads the instruction from instruction memory, decodes the instruction, reads the register, executes the operation, and writes back to the register file. The instruction set supported is the same as given in the lecture notes.

The execution of instruction continues till it reaches instruction “0xFFFFFFFF”; the simulator stops and writes the updated memory contents onto a memory text file.

The simulator also prints messages for each stage; for example, for the third instruction above following messages are printed.

* Fetch prints:
  + “FETCH: Fetch instruction 0xE3A0200A from address 0x0”
* Decode
  + “DECODE: Operation is ADD, first operand R2, Second operand R3, destination register R1”
  + “DECODE: Read registers R2 = 10, R3 = 2”
* Execute
  + “EXECUTE: ADD 10 and 2”
* Memory
  + “MEMORY: No memory operation.”
* Writeback
  + “WRITEBACK: write 12 to R1”

# Design of Simulator

## Data structure

Registers, memories, and intermediate output for each stage of instruction execution are declared as global. The variables are not visible outside the file, thus, making the data encapsulated in the myRISCSim.py.

## Simulator flow:

There are two steps:

1. First, memory is loaded with an input memory file.
2. Simulator executes instructions one by one.

For the second step, there is an infinite loop, which simulates all the instructions till the instruction sequence reads “0xFFFFFFFF”.

Next, we describe the implementation of the fetch, decode, execute, memory, and write-back function.

## Fetch:

The implementation of instruction begins with this phase. Using the Program counter, PC, which effectively carries that instruction's address, the instruction is now retrieved from memory.

Here, the instruction\_word of the instruction is updated. Instruction\_word will now contain the hex code of the instruction to be executed.

## Decode:

The second stage of the total implementation is the decoding process. The hexadecimal value that was previously retrieved from the PC is decoded in this phase. To put it another way, information about the opcode, function 3, and function 7 and immediate is retrieved. As a result, we can determine the operation that must be carried out as well as the relevant registers and immediately after the operation. Moreover, if necessary, the register values are read. The value of operand1 and operand2 is updated on which the operations would be performed in the execute stage.

In addition, we set the RF\_Write signal to true when we need to write the data back into the destination register, such as with R, I, U, and J-type instructions.

## Execute:

The third stage of an instruction's execution is this. Since we now know what ALU operations need to be performed, it may be regarded as the primary phase in the total execution. This is a description of each sort of instruction in further detail:

**R type Instructions:** In this case, the ALU result is calculated using operand1 and operand2 by operating them with add, subtract, shift etc. and the result is stored in the ALUResult variable for the write-back and memory access procedure. In case of SLT like instructions ALU result is set to 1 or zero according the give conditions.

**I type Instructions** In this case, the ALU result is calculated using operand1 and operand2 by operating them with add in case of addi, and operator in case of andi , or in case of ori operation. Load operation’s ALUResult is also calculated in this stage and stored in ALUResult.

**S type Instructions:** In this case, the ALU Result gives the memory address by addition of operand1 and operand2.

**B type Instructions:** In this case, we subtract operand1 and operand2 and store in ALU Result and then determine if the branch needs to be taken or not according to the instructions.

**U type Instructions:** In this case, the ALU Result is calculated either by shifting operand2 by 12 bits or adding the shift to PC accordingly if the instruction is lui or auipc.

**J type Instructions:** This case involves only jal instruction. Here, the ALU result will always be zero.

In all these isBranch control signal is set on the basis that if the branch is taken or not. Result\_select control signal is also used and it takes value 0, 1, 2. Result\_select= 0 defines that we have to write ALU result to the destination register, result\_select=1 defines that we have to write PC+ imm to the destination register, and result\_select=1 defines that we have to write rs1+ immediate.

## Memory Access:

The process's fourth step is this one. At this stage, the result is read or loaded from the memory location that has been provided. As there is no need to write or load any data back to or from memory, nothing is done in this stage for the majority of instructions, including add, sub, and, or, div, rem, addi, andi, and many more. In such circumstances, the execution flow merely advances to the next stage. The relevant memory operations are carried out in the case of other instructions like lb, lh, and lw.

We have implemented Data Memory as a dictionary to read and write values . In lb, lh, lw the data is loaded from Data memory accordingly. For S type instructions, values are written in the data memory.

## Write Back:

The implementation of the instruction ends with this step. If the operation being carried out at the time calls for a register update, register updates are made in this phase. We have used the RF\_Write signal to control it. If the instruction is of type R, U, J, or I, the value of this variable is maintained as True; otherwise, if the instruction is of type S or B, the value is maintained as False. This variable indicates whether the current instruction requires a write-back operation.

Lastly, if the value of write back signal is equal to false when execution control reaches the write back() method, nothing is done. On the other hand, if the value is True, register data is updated in the target register (which contains the result of instruction execution obtained in the executing stage or obtained during the memory operation). The execution of an instruction is finished at this point.

Result\_select control unit is used to decide what needs to be written back in the registers.

Control signals are updated as and when required in all these stages.

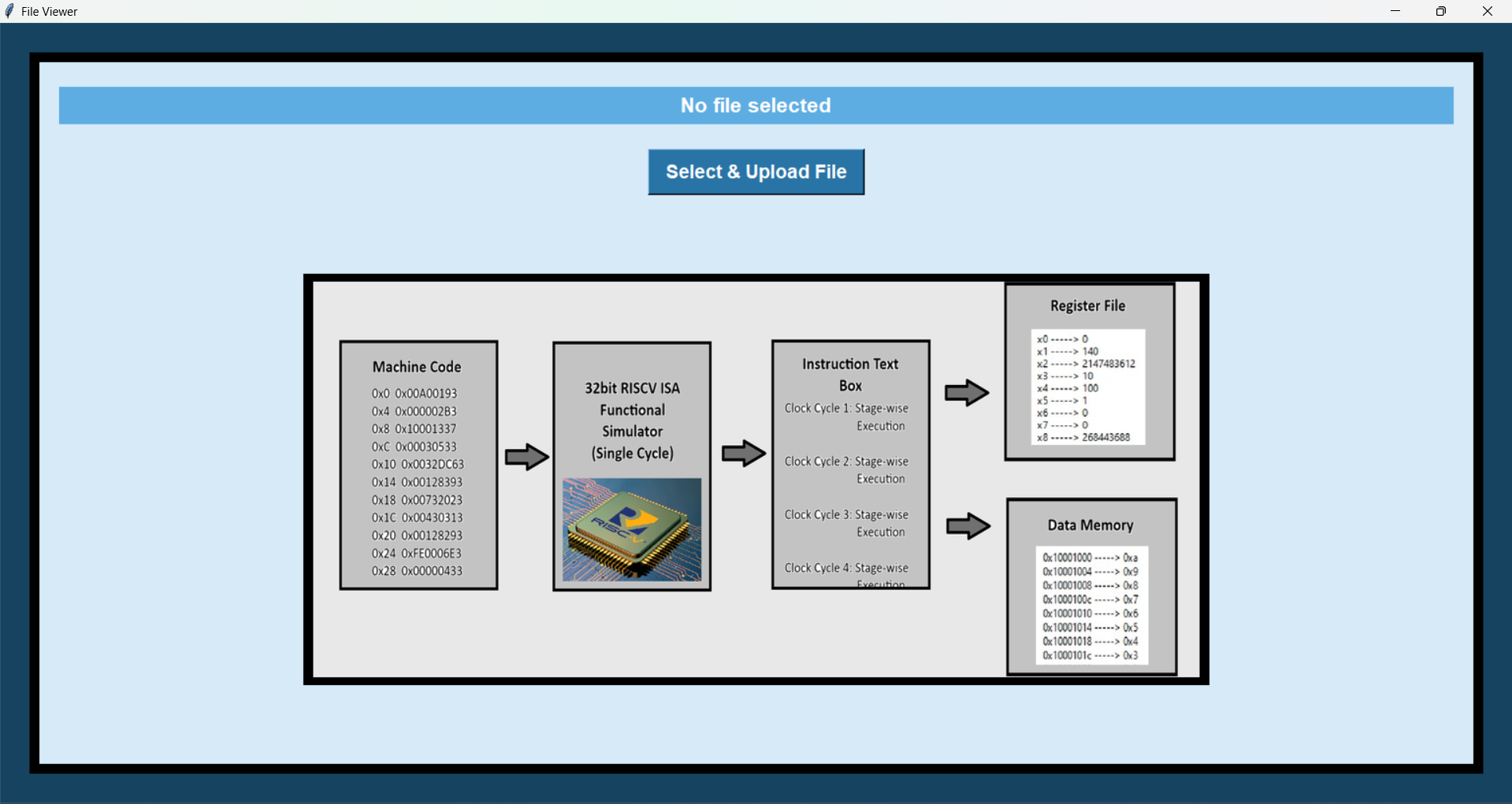
# Test plan

We test the simulator with following assembly programs:

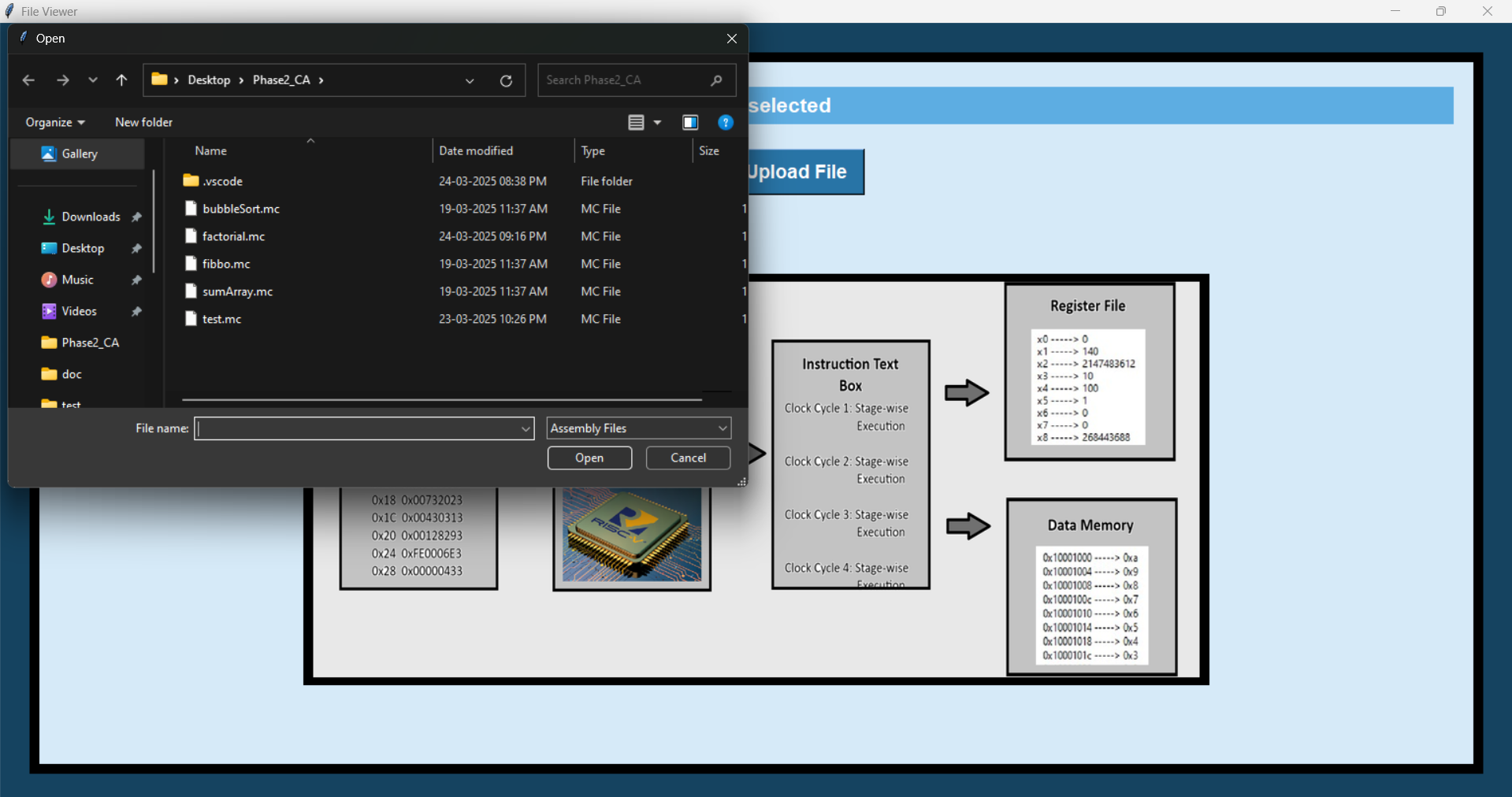
* Fibonacci Program
* Factorial
* Bubble Sort

# GUI

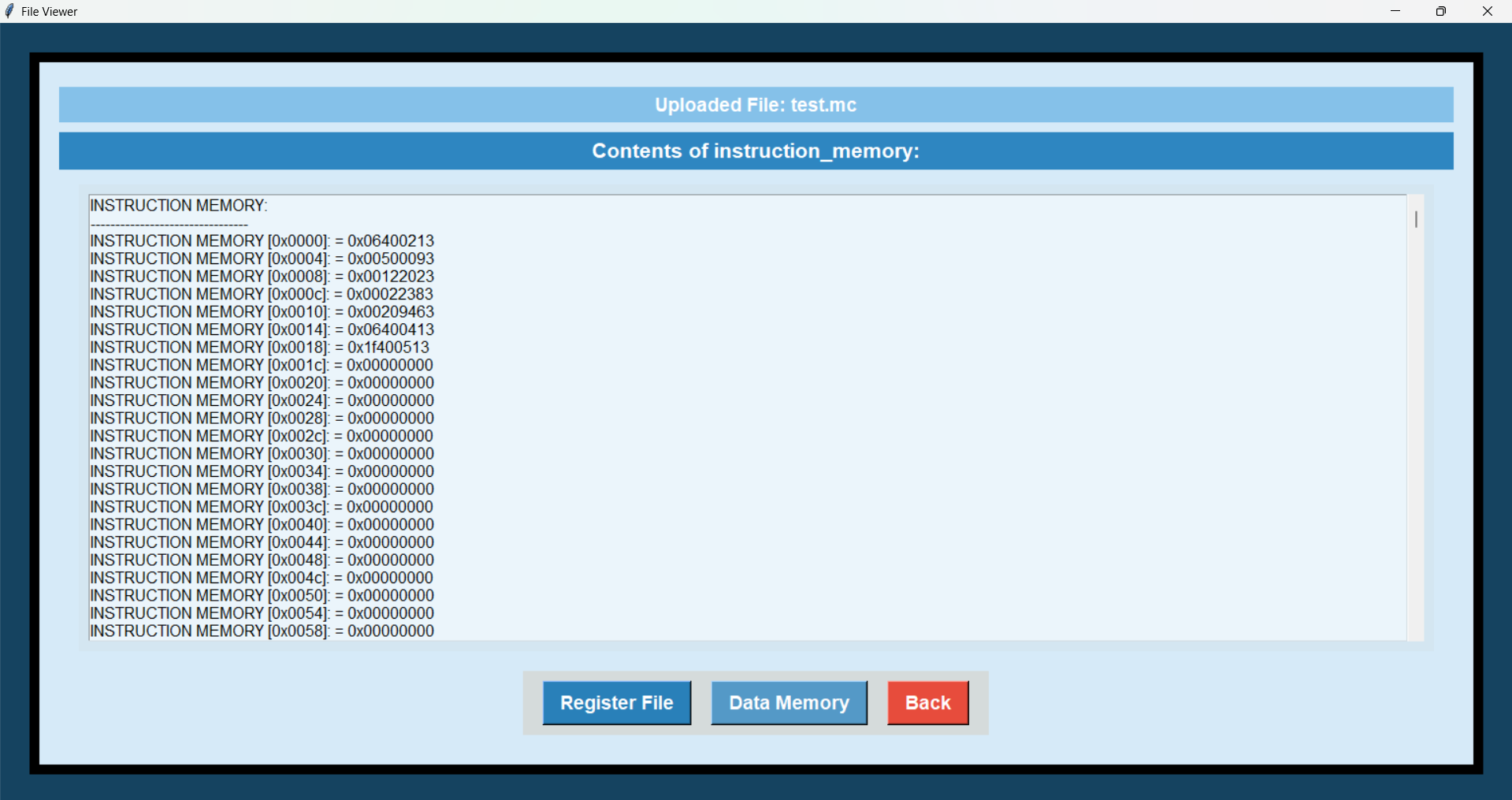
1. The first GUI window allows the user to select the input file present same directory as code.



On selecting “Input File”, all the files in the drop-down menu will be displayed, from which the user can select anyone (as shown below).



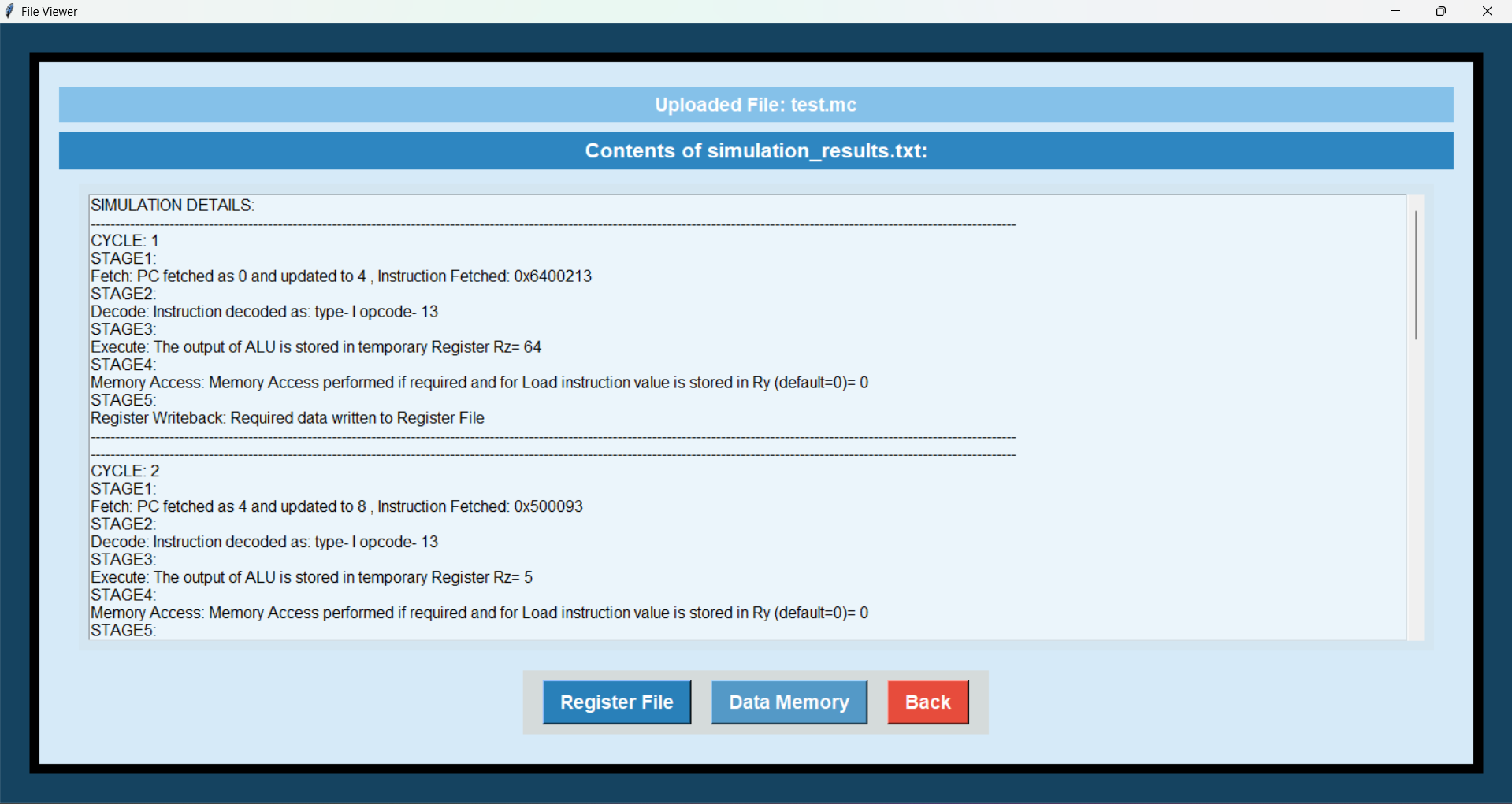
1. The instruction text Box appears after selecting the input file that contains all the stages the instruction is going through, along with the clock cycle and PC values

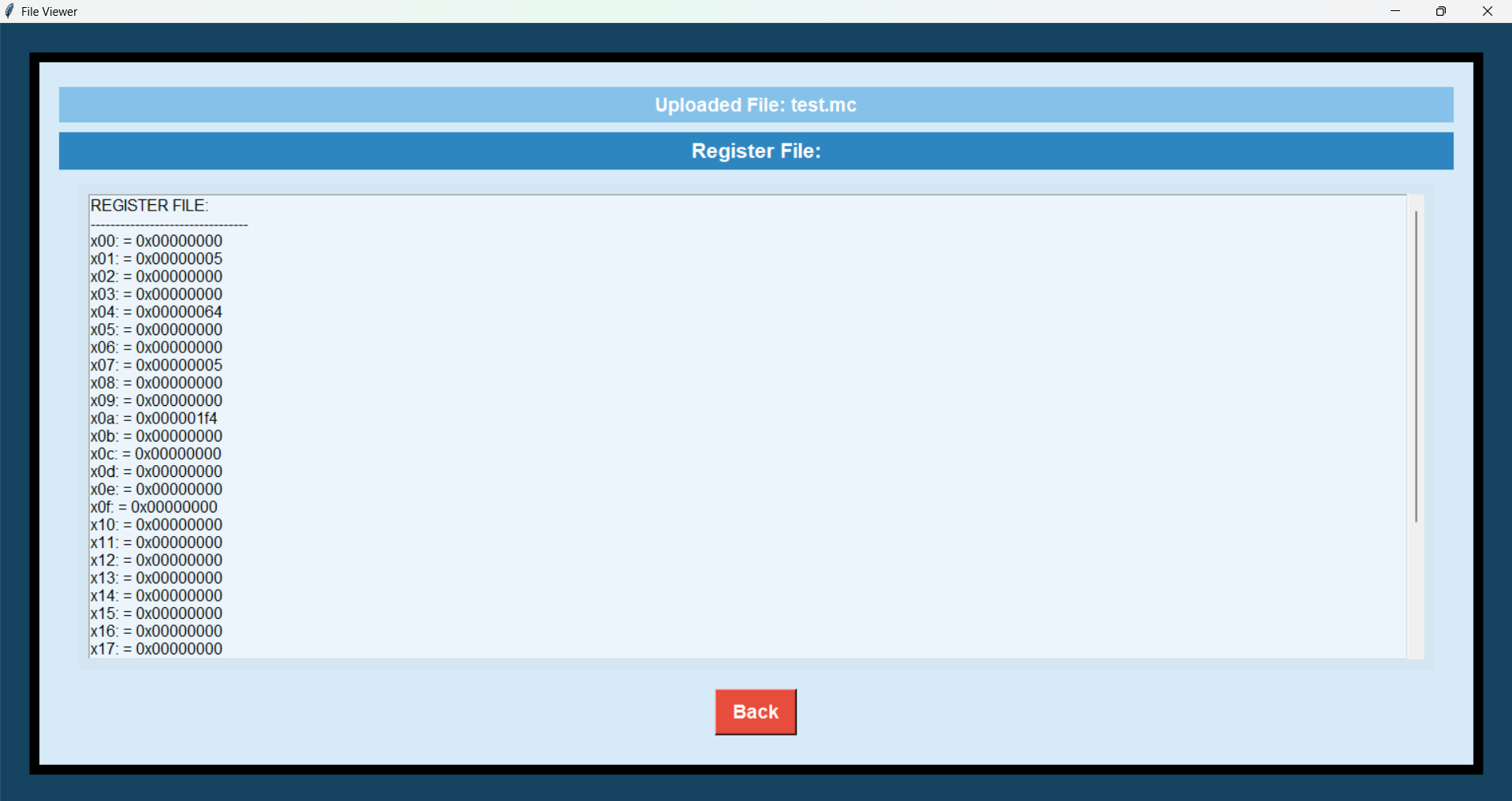


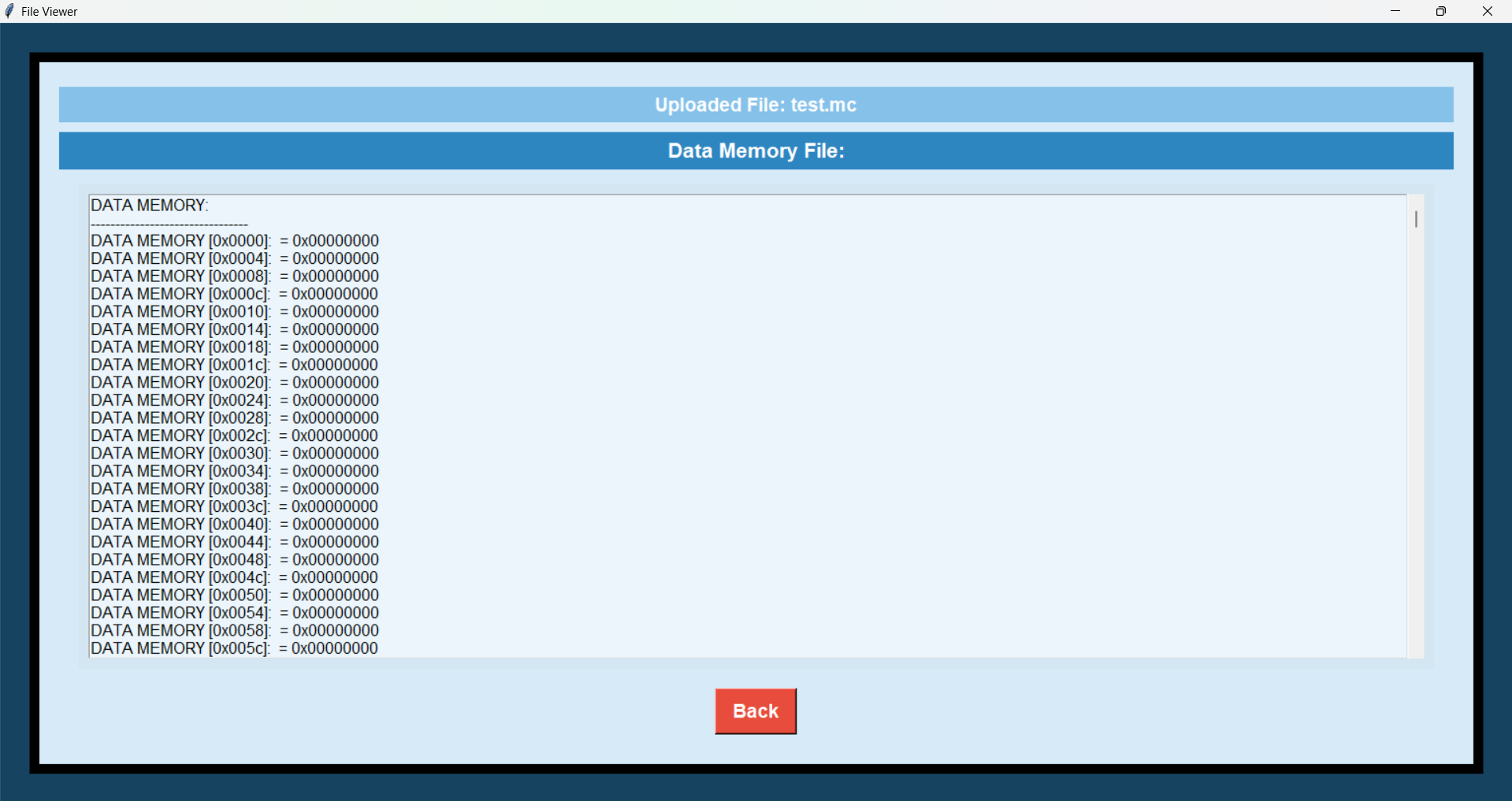
1. The Instruction text Box has two buttons at the end, one for Register File and the other for Data Memory

The Register File button opens the register File window containing the registers and their values.

The Data memory contains the Memory registers along with their values.







We have added back button at every stage for better user interface.

# Codes used:

1. Factorial:

Assembly:

Machine Code:

0x0 0x00A00E13 addi x28 x0 10 addi x28,x0,10

0x4 0x00A00293 addi x5 x0 10 addi x5,x0,10

0x8 0x10000EB7 lui x29 65536 li x29,0x10000100

0xc 0x100E8E93 addi x29 x29 256 li x29,0x10000100

0x10 0x03C05063 bge x0 x28 32 blez x28,n\_0 # if n = 0

0x14 0x000E1263 bne x28 x0 4 bnez x28, compute\_fact

0x18 0xFFFE0E13 addi x28 x28 -1 addi x28,x28,-1 # x28 will hold current n-1

0x1c 0x000E0663 beq x28 x0 12 beqz x28,store\_fact # n-1 = 0, implies now current n=1, we can now store the computed factorial value in x5 to address n\_fact and can exit

0x20 0x03C282B3 mul x5 x5 x28 mul x5,x5,x28 # x5 = x5\*x28

0x24 0xFE0E1AE3 bne x28 x0 -12 bnez x28,compute\_fact

0x28 0x005EA023 sw x5 0(x29) sw x5,0(x29) # store computed factorial in x5 to address pointed by x29 i.e. n\_fact

1. Fibonacci:

Assembly:

addi x28,x0,20

li x29,0x10000100

blez x28,exit # if n <= 0, just directly exit

li x5,0 # load register x5 with immediate value 0

sw x5,0(x29) # store immediate value in x5 to address pointed by x29, i.e. f(1) = 0

addi x28,x28,-1 # decrement n by 1, as f(1) is stored now

beqz x28,exit # if n was initially 1, now it's 0. i.e., only f(1) was to be stored that is done and now we can exit

li x6,1 # load register x6 with immediate value 1

sw x6,4(x29) # store immediate value in x6 to address pointed by x29 +4, i.e. f(2) = 1

addi x28,x28,-1 # decrement n by 1 again, as f(2) is stored now

beqz x28,exit # if n was initially 2, now it's 0. i.e., only till f(2) was to be stored that is done and now we can exit

bnez x28,fibonacci # if still n>0, jump to fibonacci and execute

fibonacci:

lw x5,0(x29) # load f(i-2) to x5

lw x6,4(x29) # load f(i-1) to x6

add x7,x6,x5 # x7 = x6 + x5 || f(i) = f(i-1) + f(i-2)

sw x7,8(x29) # store f(i) to ptr + 2 mem\_adr,

addi x28,x28,-1 # decrement n by 1

addi x29,x29,4 # increment pointer by 1

bnez x28,fibonacci # if still n>0, jump to fibonacci and execute

jal exit # if n=0, exit

exit:

Machine:

0x0 0x01400E13 addi x28 x0 20 addi x28,x0,20

0x4 0x10000EB7 lui x29 65536 li x29,0x10000100

0x8 0x100E8E93 addi x29 x29 256 li x29,0x10000100

0xc 0x05C05463 bge x0 x28 72 blez x28,exit # if n <= 0, just directly exit

0x10 0x00000293 addi x5 x0 0 li x5,0 # load register x5 with immediate value 0

0x14 0x005EA023 sw x5 0(x29) sw x5,0(x29) # store immediate value in x5 to address pointed by x29, i.e. f(1) = 0

0x18 0xFFFE0E13 addi x28 x28 -1 addi x28,x28,-1 # decrement n by 1, as f(1) is stored now

0x1c 0x020E0C63 beq x28 x0 56 beqz x28,exit # if n was initially 1, now it's 0. i.e., only f(1) was to be stored that is done and now we can exit

0x20 0x00100313 addi x6 x0 1 li x6,1 # load register x6 with immediate value 1

0x24 0x006EA223 sw x6 4(x29) sw x6,4(x29) # store immediate value in x6 to address pointed by x29 +4, i.e. f(2) = 1

0x28 0xFFFE0E13 addi x28 x28 -1 addi x28,x28,-1 # decrement n by 1 again, as f(2) is stored now

0x2c 0x020E0463 beq x28 x0 40 beqz x28,exit # if n was initially 2, now it's 0. i.e., only till f(2) was to be stored that is done and now we can exit

0x30 0x000E1263 bne x28 x0 4 bnez x28,fibonacci # if still n>0, jump to fibonacci and execute

0x34 0x000EA283 lw x5 0(x29) lw x5,0(x29) # load f(i-2) to x5

0x38 0x004EA303 lw x6 4(x29) lw x6,4(x29) # load f(i-1) to x6

0x3c 0x005303B3 add x7 x6 x5 add x7,x6,x5 # x7 = x6 + x5 || f(i) = f(i-1) + f(i-2)

0x40 0x007EA423 sw x7 8(x29) sw x7,8(x29) # store f(i) to ptr + 2 mem\_adr,

0x44 0xFFFE0E13 addi x28 x28 -1 addi x28,x28,-1 # decrement n by 1

0x48 0x004E8E93 addi x29 x29 4 addi x29,x29,4 # increment pointer by 1

0x4c 0xFE0E14E3 bne x28 x0 -24 bnez x28,fibonacci # if still n>0, jump to fibonacci and execute

0x50 0x004000EF jal x1 4 jal exit # if n=0, exit

1. Bubble Sort:

Assembly:

Machine: