Tomasulo’s Algorithm Simulation Report

Omar Miniesy 900202087
Ziad Miniesy 900202283

Overview

This project aims to implement the simulation of Tomasulo's Algorithm. It takes the text file containing the assembly program to be simulated. It takes the number of functional units for the following units: ADD, LOAD, STORE, NEG, NAND, SLL, BNE and JAL. It also takes the number of cycles needed for each operation supported. The program issues, executes the assembly code and writes back the data to the register file or the memory. The supported instructions are: LOAD, STORE, BNE, JAL, RET, ADD, ADDI, NEG, NAND, SLL. At the end of the simulation, a table is displayed showing when each instruction was issued, started execution, finished execution and written back. The total execution time for the whole program represented in the number of cycles is also shown as well as the IPC and the branch misprediction percentage.

Important Data Structures and Functions

- At the beginning, the function read_instructions() reads the instructions from a text file and stores them in a vector. Then those instructions are parsed and stored in a vector of type instructions, which is a struct that includes all the important information of an instruction such as rA, rB, imm…etc.
- There is also a struct for each functional unit which includes Name, Vj, Vk, Qj, Qk and Address where applicable as well as if it is busy or not busy. There is a vector for each functional unit of its type.
- Instructions are parsed using parse_instructions() function. The operands for each instruction are determined and stored in the instruction vector.
- The register file is implemented using an unordered map of type string to int.
- The memory is implemented using an unordered map of type int to int.
- The user can initialize the register file with values using the initialize_registers_file() function.
- The initialize_memory_file() function sets initial values in the memory text file.
- The initialize_memory_map() function stores the contents of the memory file to the memory map it also sets the starting address based on the user's input.
- Terminate() function returns true if all functional units are empty than exit the program, this means that all instructions have been issued, executed and written back.

Issue

To issue an instruction we loop over the instructions vector and check if there are available functional units for this instruction. If yes, then we set the instruction to issued, and fill the functional units. The source registers are inserted in Vj or Vk if ready. If they are not ready the functional unit producing the needed source registers are stored in Qj and Qk. If the functional unit is not free then the instruction is not issued. The destination register is inserted into the register status table along with the functional unit writing to it. When an instruction is issued it is stored in the issued instructions vector. We also keep track of the issue index, it represents the next instruction to be issued exactly like the PC.

Execute

To execute an instruction we loop over the issued instructions vector. We have to check that the issue cycle of the instruction to be executed is not equal to the current cycle. We also have to check that Qj and Qk of the functional unit are empty. Execution decrements the value of the remaining time. It checks each time if this is the last execution cycle, if yes, it sets the instruction to executed. When an instruction is executed we push it to the executed instructions vector.

Write Back

To write back an instruction, we first sort the executed instructions vector by issue time. This ensures that if two instructions finish execution at the same time, the older one will be written back first. To write, we check the functional unit responsible for the instruction, we set the instruction to written, then we store the computed result in the register file or the memory. Finally, we free the reservation station and check if other functional units had the Qj or Qk waiting for the current functional units, if yes then empty Qj/Qk and set Vj/Vk with the register value. For the JAL, RET and BNE instructions, we change the value of the issue index mentioned above. For the JAL and RET, we stall until the required instruction is needed, so the instructions between a JAL and its destination will not be issued. For the BNE, we issue all of them between, but we do not execute them if the branch was taken.

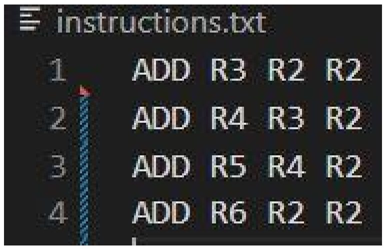
Bonus

The bonus feature implemented was variable length hardware. The user inputs how many functional units exist for each type as well as the number of cycles each instruction consumes. At the beginning of the program the initialize_reservation_stations() function is called with the input parameters entered by the user. This function creates a vector for each functional unit of its type with the size specified by the user. In the execution stage, the number of cycles taken is determined also by the user input. The remaining time for the instruction is initialized with the number of cycles input by the user and is decremented each cycle.

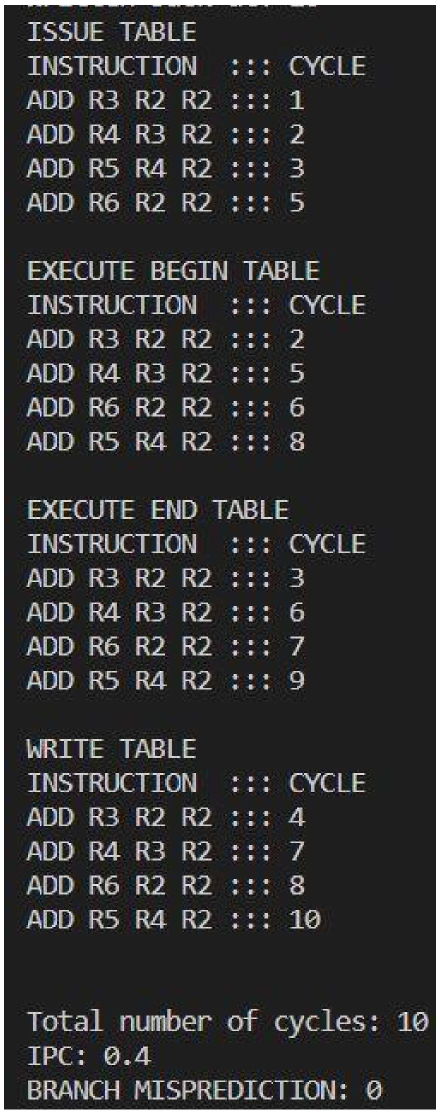
Simulation Output

R3 has the value 3 and R2 has the value 2 in all test cases.

1. First test case to test that the issuing depends on the number of functional units, and the last one won't issue until one of them writes back

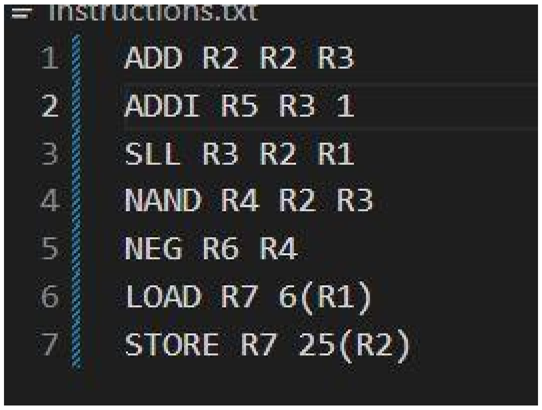


```
instructions.txt
ADD R3 R2 R2
ADD R4 R3 R2
ADD R5 R4 R2
ADD R6 R2 R2
```

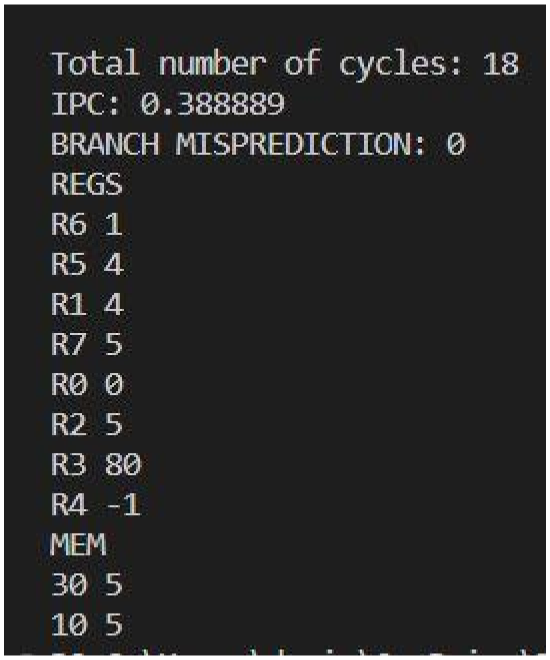


```
ISSUE TABLE
INSTRUCTION  ::: CYCLE
ADD R3 R2 R2 ::: 1
ADD R4 R3 R2 ::: 2
ADD R5 R4 R2 ::: 3
ADD R6 R2 R2 ::: 5

EXECUTE BEGIN TABLE
INSTRUCTION  ::: CYCLE
ADD R3 R2 R2 ::: 2
ADD R4 R3 R2 ::: 5
ADD R6 R2 R2 ::: 6
ADD R5 R4 R2 ::: 8

EXECUTE END TABLE
INSTRUCTION  ::: CYCLE
ADD R3 R2 R2 ::: 3
ADD R4 R3 R2 ::: 6
ADD R6 R2 R2 ::: 7
ADD R5 R4 R2 ::: 9

WRITE TABLE
INSTRUCTION  ::: CYCLE
ADD R3 R2 R2 ::: 4
ADD R4 R3 R2 ::: 7
ADD R6 R2 R2 ::: 8
ADD R5 R4 R2 ::: 10


Total number of cycles: 10
IPC: 0.4
BRANCH MISPREDICTION: 0
```

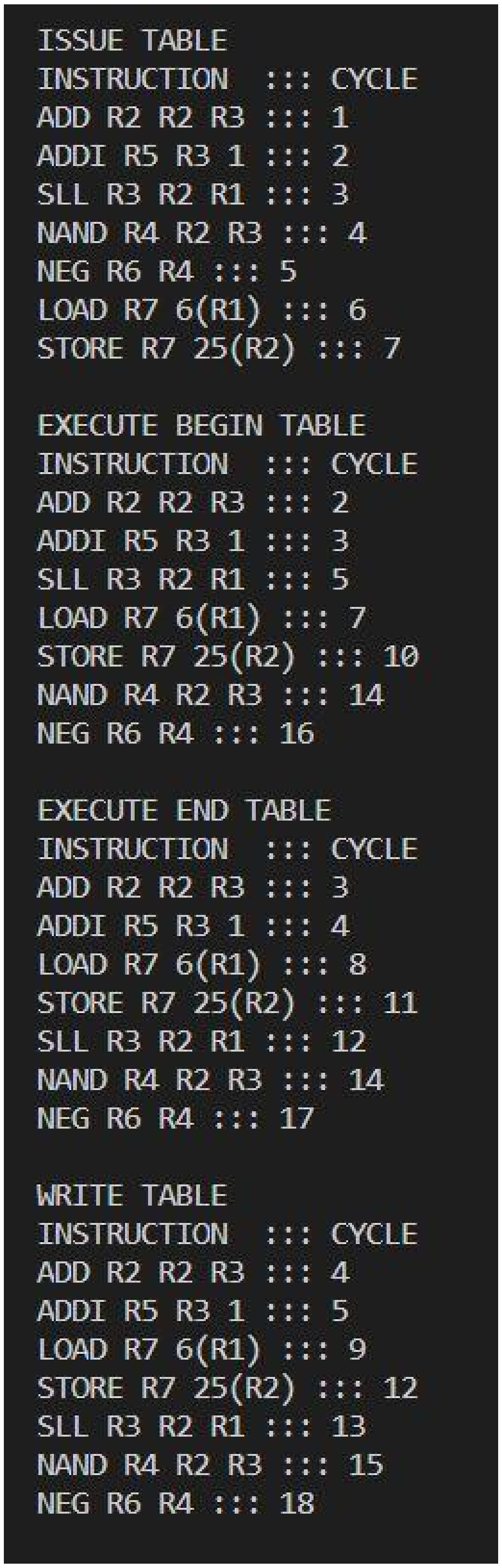
2. Second test case to realize that any instruction that depends on another instruction waits for it to finish to start execution if any two instructions write at the same instant, the one issued first writes first.



```
instructions.txt
ADD R2 R2 R3
ADDI R5 R3 1
SLL R3 R2 R1
NAND R4 R2 R3
NEG R6 R4
LOAD R7 6(R1)
STORE R7 25(R2)
```

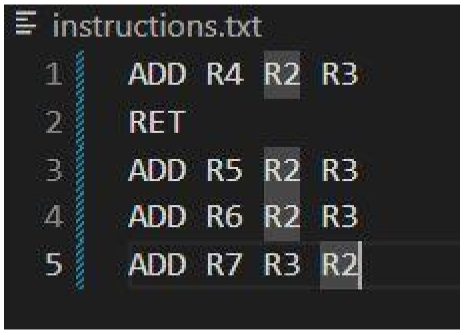


```
Total number of cycles: 18
IPC: 0.388889
BRANCH MISPREDICTION: 0
REGS
R6 1
R5 4
R1 4
R7 5
R0 0
R2 5
R3 80
R4 -1
MEM
30 5
10 5
```

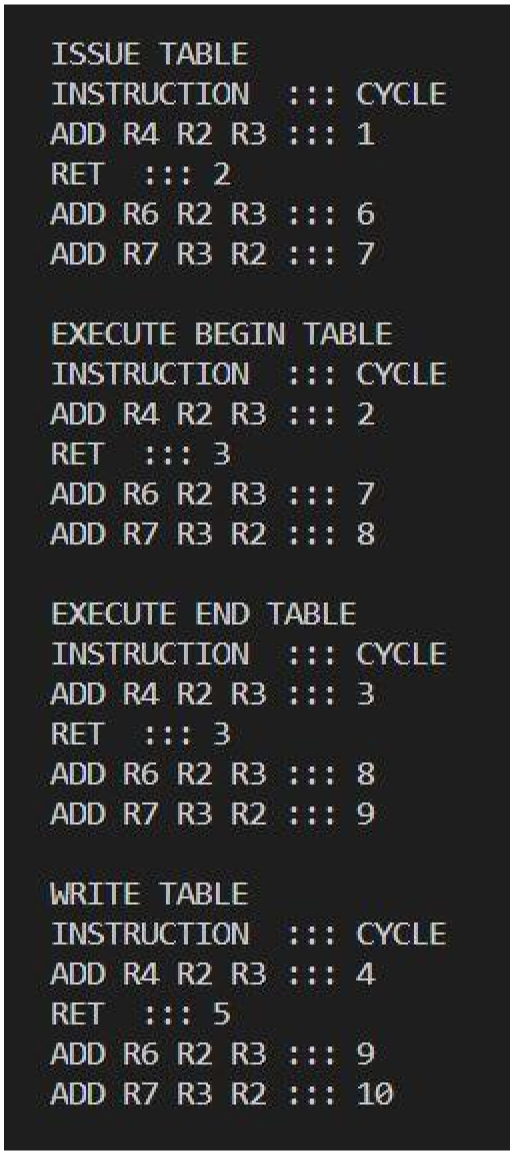


```
ISSUE TABLE
INSTRUCTION  ::: CYCLE
ADD R2 R2 R3 ::: 1
ADDI R5 R3 1 ::: 2
SLL R3 R2 R1 ::: 3
NAND R4 R2 R3 ::: 4
NEG R6 R4 ::: 5
LOAD R7 6(R1) ::: 6
STORE R7 25(R2) ::: 7

EXECUTE BEGIN TABLE
INSTRUCTION  ::: CYCLE
ADD R2 R2 R3 ::: 2
ADDI R5 R3 1 ::: 3
SLL R3 R2 R1 ::: 5
LOAD R7 6(R1) ::: 7
STORE R7 25(R2) ::: 10
NAND R4 R2 R3 ::: 14
NEG R6 R4 ::: 16

EXECUTE END TABLE
INSTRUCTION  ::: CYCLE
ADD R2 R2 R3 ::: 3
ADDI R5 R3 1 ::: 4
LOAD R7 6(R1) ::: 8
STORE R7 25(R2) ::: 11
SLL R3 R2 R1 ::: 12
NAND R4 R2 R3 ::: 14
NEG R6 R4 ::: 17

WRITE TABLE
INSTRUCTION  ::: CYCLE
ADD R2 R2 R3 ::: 4
ADDI R5 R3 1 ::: 5
LOAD R7 6(R1) ::: 9
STORE R7 25(R2) ::: 12
SLL R3 R2 R1 ::: 13
NAND R4 R2 R3 ::: 15
NEG R6 R4 ::: 18
```

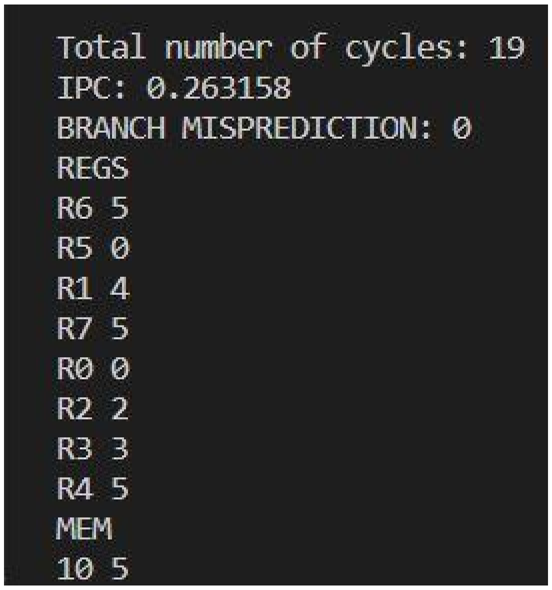
3. Third test case jumps to the 4th instruction, stall until this instruction is issued.



```
instructions.txt
ADD R4 R2 R3
RET
ADD R5 R2 R3
ADD R6 R2 R3
ADD R7 R3 R2
```

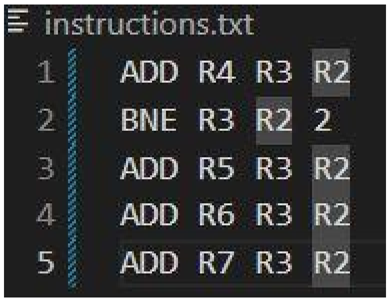


```
ISSUE TABLE
INSTRUCTION  ::: CYCLE
ADD R4 R2 R3 ::: 1
RET  ::: 2
ADD R6 R2 R3 ::: 6
ADD R7 R3 R2 ::: 7

EXECUTE BEGIN TABLE
INSTRUCTION  ::: CYCLE
ADD R4 R2 R3 ::: 2
RET  ::: 3
ADD R6 R2 R3 ::: 7
ADD R7 R3 R2 ::: 8

EXECUTE END TABLE
INSTRUCTION  ::: CYCLE
ADD R4 R2 R3 ::: 3
RET  ::: 3
ADD R6 R2 R3 ::: 8
ADD R7 R3 R2 ::: 9

WRITE TABLE
INSTRUCTION  ::: CYCLE
ADD R4 R2 R3 ::: 4
RET  ::: 5
ADD R6 R2 R3 ::: 9
ADD R7 R3 R2 ::: 10
```

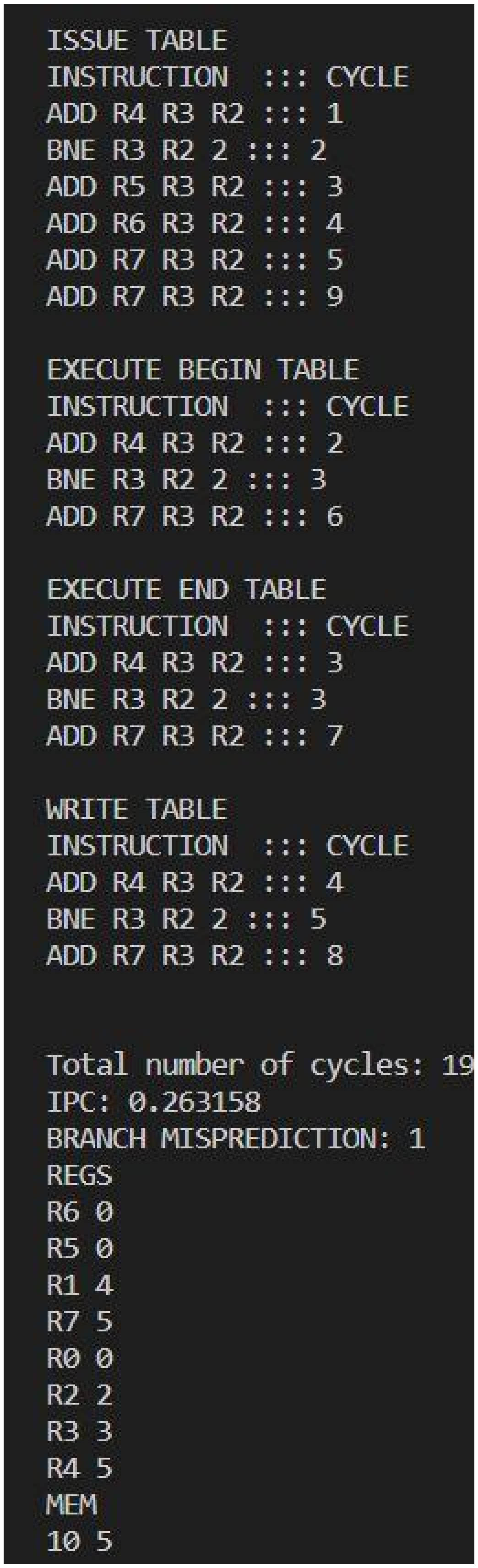


```
Total number of cycles: 19
IPC: 0.263158
BRANCH MISPREDICTION: 0
REGS
R6 5
R5 0
R1 4
R7 5
R0 0
R2 2
R3 3
R4 5
MEM
10 5
```

4. Fourth test case to test BNE. It is supposed to skip the following 2 instructions and branch to instruction 5.

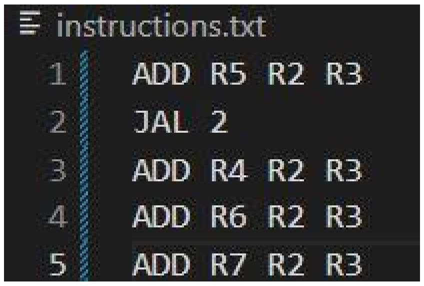


```
instructions.txt
ADD R4 R3 R2
BNE R3 R2 2
ADD R5 R3 R2
ADD R6 R3 R2
ADD R7 R3 R2
```

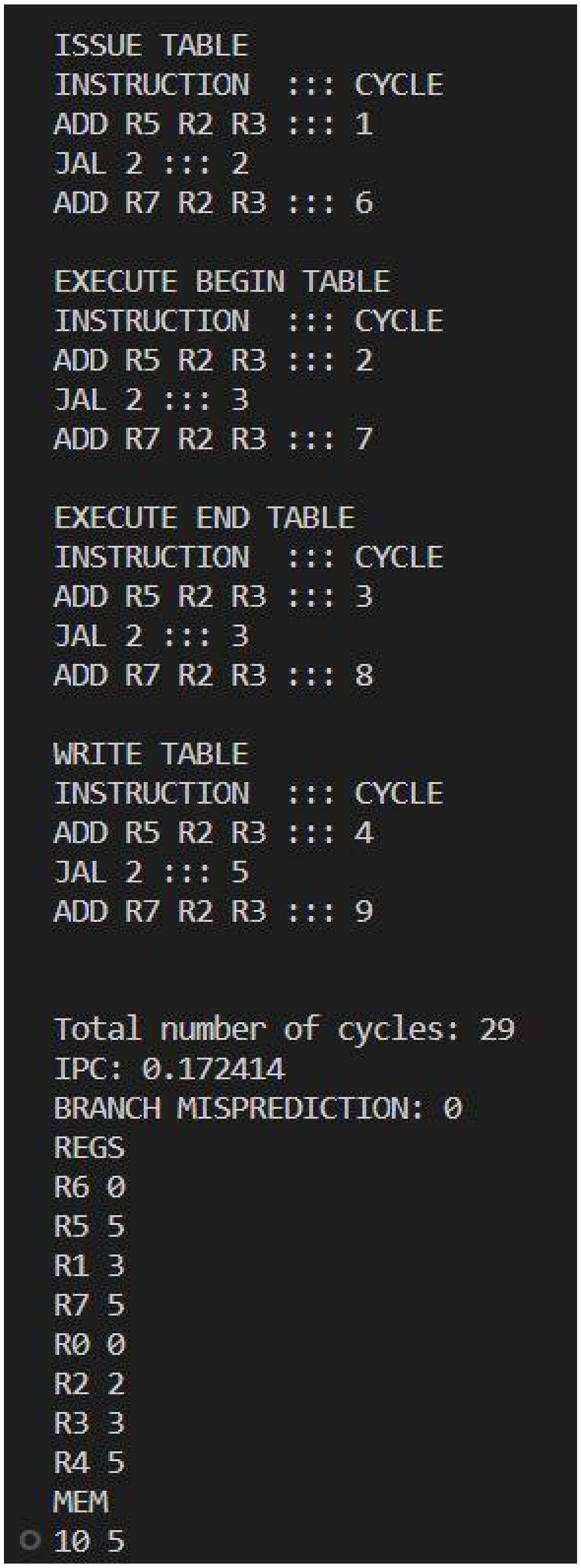


```
ISSUE TABLE
INSTRUCTION  ::: CYCLE
ADD R4 R3 R2 ::: 1
BNE R3 R2 2 ::: 2
ADD R5 R3 R2 ::: 3
ADD R6 R3 R2 ::: 4
ADD R7 R3 R2 ::: 5
ADD R7 R3 R2 ::: 9

EXECUTE BEGIN TABLE
INSTRUCTION  ::: CYCLE
ADD R4 R3 R2 ::: 2
BNE R3 R2 2 ::: 3
ADD R7 R3 R2 ::: 6

EXECUTE END TABLE
INSTRUCTION  ::: CYCLE
ADD R4 R3 R2 ::: 3
BNE R3 R2 2 ::: 3
ADD R7 R3 R2 ::: 7

WRITE TABLE
INSTRUCTION  ::: CYCLE
ADD R4 R3 R2 ::: 4
BNE R3 R2 2 ::: 5
ADD R7 R3 R2 ::: 8


Total number of cycles: 19
IPC: 0.263158
BRANCH MISPREDICTION: 1
REGS
R6 0
R5 0
R1 4
R7 5
R0 0
R2 2
R3 3
R4 5
MEM
10 5
```

5. Fifth test case to test JAL. It is supposed to skip the 2 instructions after it and jump to the last instruction.



```
instructions.txt
ADD R5 R2 R3
JAL 2
ADD R4 R2 R3
ADD R6 R2 R3
ADD R7 R2 R3
```



```
ISSUE TABLE
INSTRUCTION  ::: CYCLE
ADD R5 R2 R3 ::: 1
JAL 2 ::: 2
ADD R7 R2 R3 ::: 6

EXECUTE BEGIN TABLE
INSTRUCTION  ::: CYCLE
ADD R5 R2 R3 ::: 2
JAL 2 ::: 3
ADD R7 R2 R3 ::: 7

EXECUTE END TABLE
INSTRUCTION  ::: CYCLE
ADD R5 R2 R3 ::: 3
JAL 2 ::: 3
ADD R7 R2 R3 ::: 8

WRITE TABLE
INSTRUCTION  ::: CYCLE
ADD R5 R2 R3 ::: 4
JAL 2 ::: 5
ADD R7 R2 R3 ::: 9


Total number of cycles: 29
IPC: 0.172414
BRANCH MISPREDICTION: 0
REGS
R6 0
R5 5
R1 3
R7 5
R0 0
R2 2
R3 3
R4 5
MEM
10 5
```

User Guide

1. To run the program, enter the number of each functional unit and the number of cycles for each operation.
2. Initialize the register file with any values you wish, uninitialized registers will be equal to zero.
3. Initialize the memory file in the text file with the address then the value. E.g: 10 5 means address 10 contains the value 5.
4. Enter the assembly code in the text file with the same format in the project description pdf but without commas.

Limitations

We were supposed to prevent changing R0 by checking the destination register in the execute function. However, it did not work for some reason and we did not have time to fix it.