Tomasulo Algorithm Simulation

Report

https://github.com/F23-Omar-Bavly-Architecture/TomasuloSimulator

Name (s):
Omar
Elfouly,
Bavly
Remon,

1.0 Introduction

Purpose:

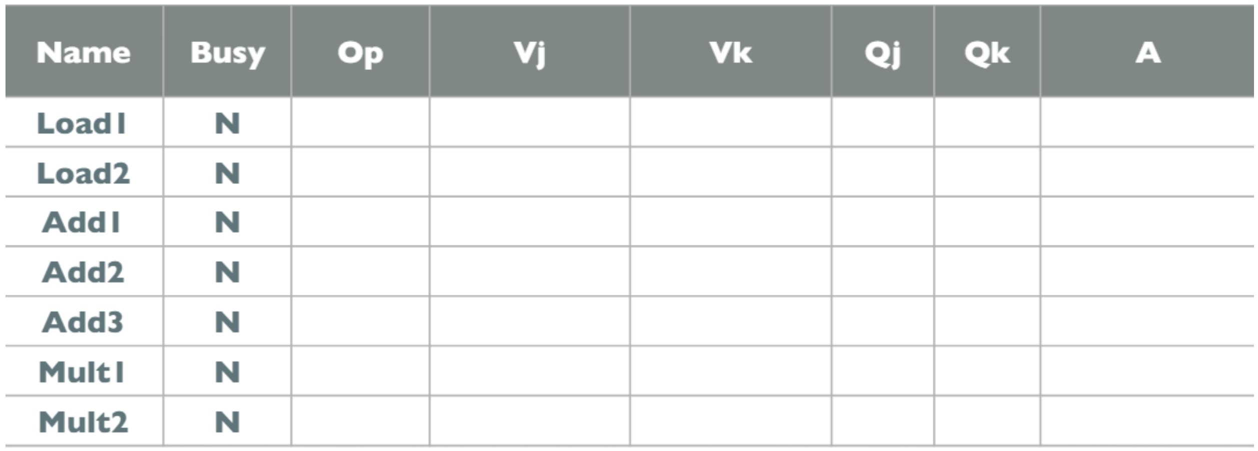
The goal of this project is to implement an architectural simulator capable of assessing the performance of a simplified out-of-order 16-bit RISC processor that uses Tomasulo's algorithm without speculation.

2.0 Implementation and Design

The simulator assumes a simplified RISC ISA. The word size of this computer is 16- bit. The processor has 8 general-purpose registers R0 to R7 (16-bit each). The register R0 always contains the value 0 and cannot be changed. Memory is word addressable and uses a 16-bit address (as such the memory capacity is 128 KB). We support the 9 Instructions those being LOAD, STORE, BNE, CALL, RET, ADDI, NAND, DIV.

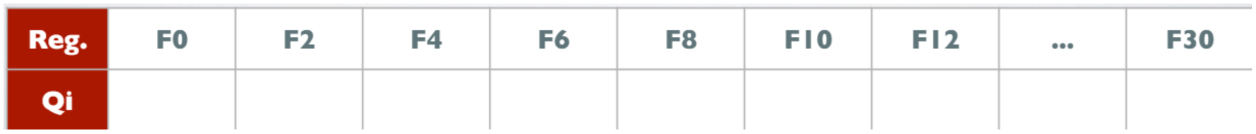
According to Tomasulo's Algorithm, we have 3 stages being issue, execute and write back. We assume that all the instructions are already fetched and decoded and are ready in the instruction queue.

The actual conditions for when to issue, execute or write back was straightforward, the main issue was the data structures that we used to store all the information that we needed We had a reservation station class that contained a map that maped every reservation station name to a reservationstationentry which is one entry in the table we had in the slides.



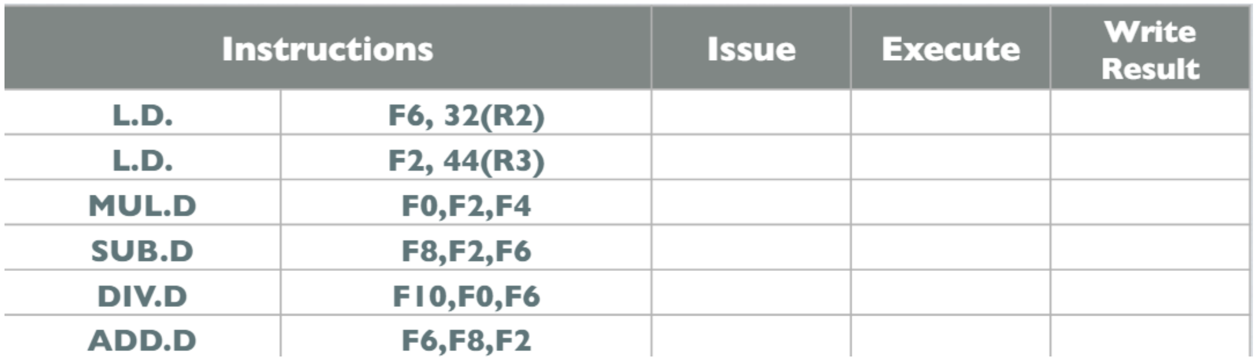
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	N						
Load2	N						
Add1	N						
Add2	N						
Add3	N						
Mult1	N						
Mult2	N						

We also had the register status table that was also a map that mapped each register to the unit producing it



Reg.	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi									

Throughout the program we keep track of when every instruction is issued, executed and written back, and at the end we display a table representing this, though in out program we mention both start and end of execution



Instructions		Issue	Execute	Write Result
L.D.	F6, 32(R2)			
L.D.	F2, 44(R3)			
MUL.D	F0,F2,F4			
SUB.D	F8,F2,F6			
DIV.D	F10,F0,F6			
ADD.D	F6,F8,F2			

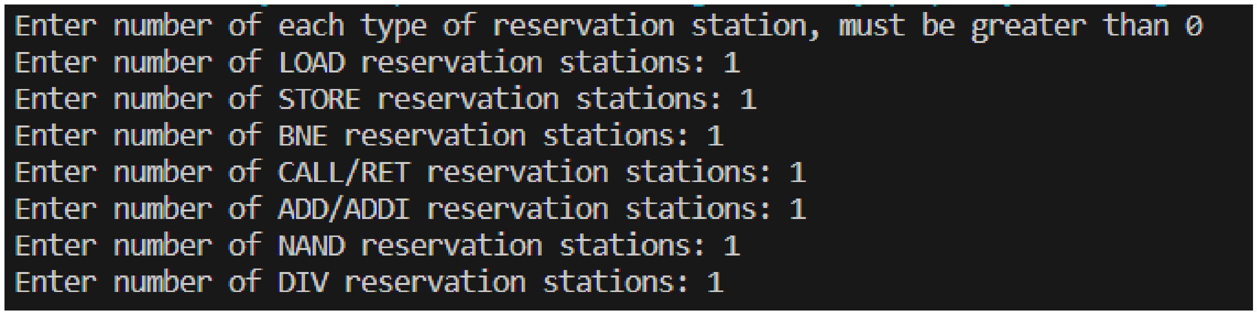
It is also worth noting that no known issues exist in our program, all issues and bugs we faces were

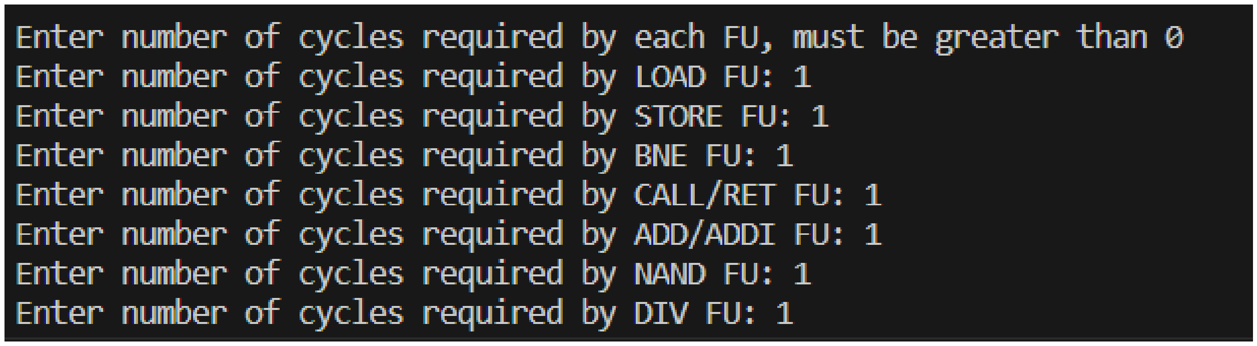
Bonus and Extra Features:

- We Support a variable hardware organization where the user is prompted to input the number of reservation stations per reservation station type and the number of cycles taken to finish execution for each functional unit.
- Our program also prompts the user to enter instructions one at a time or using a file
- We also implemented the load and store buffer to handle load hazards
- Our program covers a very specific edge case that is possible only because variable hardware organisation means a branch can take much longer than a single cycle which means it is possible for the following to occur:
    - A BNE is issued followed by an ADD then another BNE then a DIV
    - The ADD, BNE, and DIV are issued because we predicted a not taken for the first BNE
    - The DIV can also be considered to have been predicted by the second BNE not being taken
    - If it turns out that the first BNE is indeed not taken then we should only begin executing the ADD, and second BNE and not the DIV since it is also a prediction
    - This requires us to keep track of what we are referring to as a nested prediction

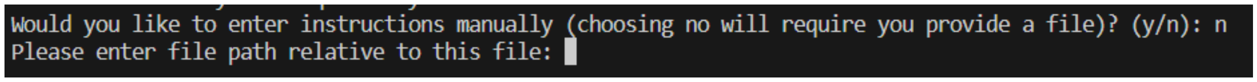
3.0 Testing and Simulating

When first ran you are prompted to enter the number of reservation stations for each type and the number of cycle each functional unit will take

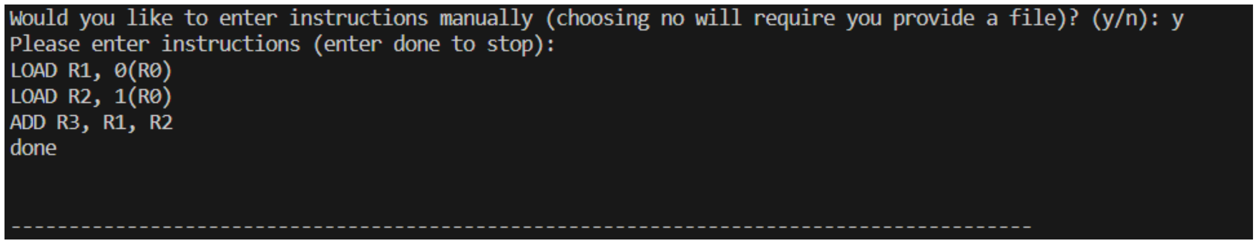

```
Enter number of each type of reservation station, must be greater than 0
Enter number of LOAD reservation stations: 1
Enter number of STORE reservation stations: 1
Enter number of BNE reservation stations: 1
Enter number of CALL/RET reservation stations: 1
Enter number of ADD/ADDI reservation stations: 1
Enter number of NAND reservation stations: 1
Enter number of DIV reservation stations: 1
```


```
Enter number of cycles required by each FU, must be greater than 0
Enter number of cycles required by LOAD FU: 1
Enter number of cycles required by STORE FU: 1
Enter number of cycles required by BNE FU: 1
Enter number of cycles required by CALL/RET FU: 1
Enter number of cycles required by ADD/ADDI FU: 1
Enter number of cycles required by NAND FU: 1
Enter number of cycles required by DIV FU: 1
```

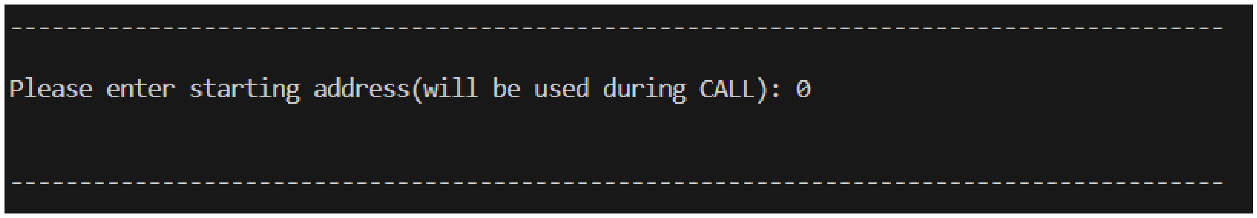
You are then given the choice to either enter your instructions one by one or to use a file if you choose to use a file you would need to give the location of the file


```
Would you like to enter instructions manually (choosing no will require you provide a file)? (y/n): n
Please enter file path relative to this file:
```

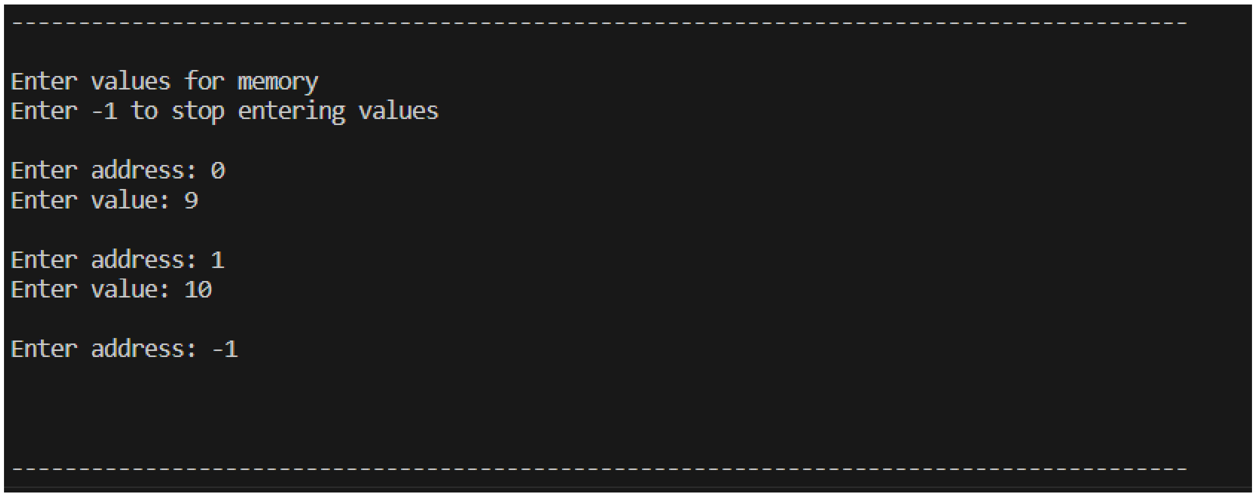
If you choose to enter instructions manually you would enter them line by line till you entered done


```
Would you like to enter instructions manually (choosing no will require you provide a file)? (y/n): y
Please enter instructions (enter done to stop):
LOAD R1, 0(R0)
LOAD R2, 1(R0)
ADD R3, R1, R2
done


--------------------------------------------------------------------------------
```

Next you would be prompted to enter a starting address


```
--------------------------------------------------------------------------------

Please enter starting address(will be used during CALL): 0


--------------------------------------------------------------------------------
```

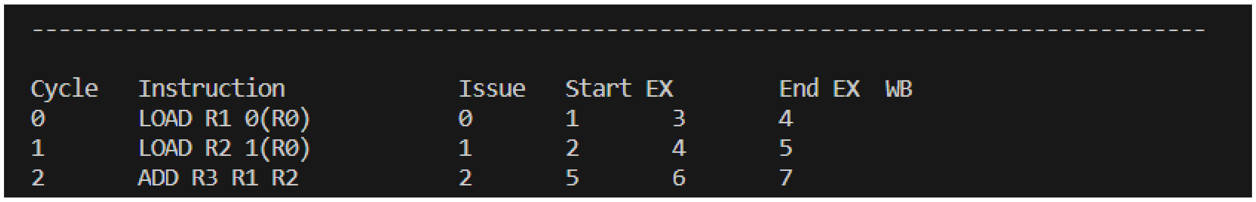
And then you can initialize certain memory locations with any data you like



```
-------------------------------------------------------------------------------------

Enter values for memory
Enter -1 to stop entering values

Enter address: 0
Enter value: 9

Enter address: 1
Enter value: 10

Enter address: -1



-------------------------------------------------------------------------------------
```

To exit and stop inputting data to the memory just type -1

At the end a table like this would be printed



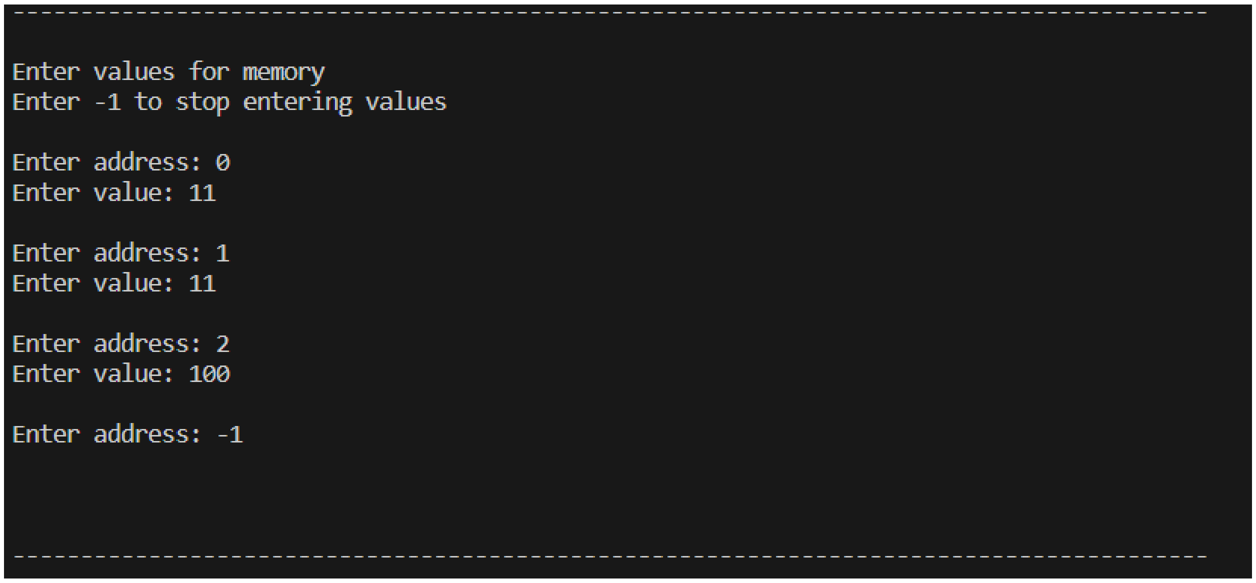
```
 -------------------------------------------------------------------------------------

 Cycle   Instruction             Issue   Start EX       End EX  WB
 0       LOAD R1 0(R0)           0       1       3       4
 1       LOAD R2 1(R0)           1       2       4       5
 2       ADD R3 R1 R2            2       5       6       7
```

Testing:

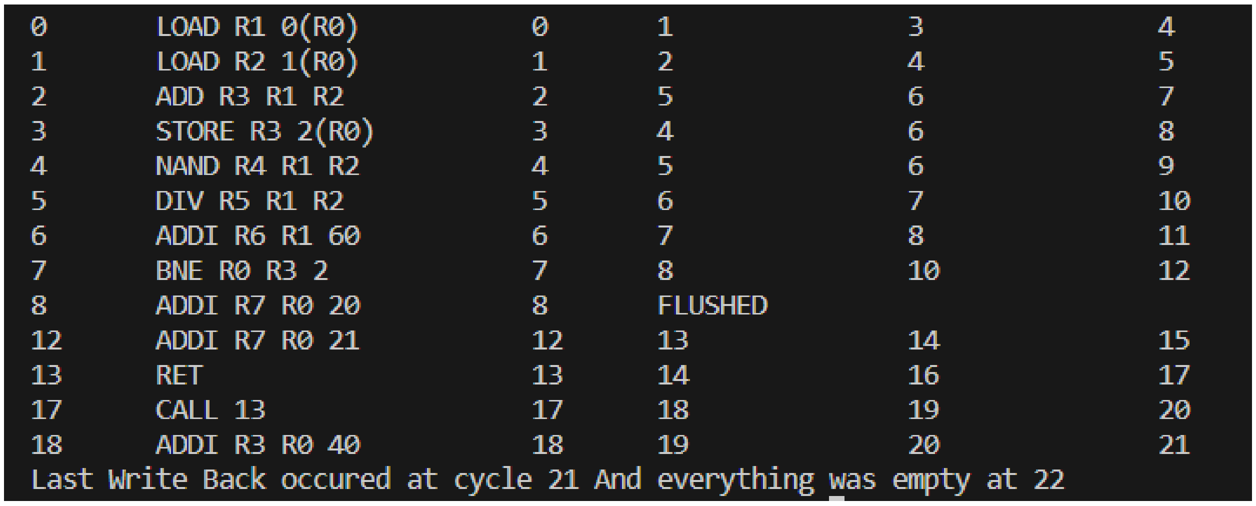
We included some test programs to check the validity of our program:

Test1:

```
LOAD R1, 0(R0)
LOAD R2, 1(R0)
ADD R3, R1, R2
STORE R3, 2(R0)
NAND R4, R1, R2
DIV R5, R1, R2
ADDI R6, R1, 60
BNE R0, R3, 2
ADDI R7, R0, 20
ADDI R7, R0, 21
RET
CALL 13
ADDI R7, R0, 20
ADDI R3, R0, 40
```



```
--------------------------------------------------------------------------------

Enter values for memory
Enter -1 to stop entering values

Enter address: 0
Enter value: 11

Enter address: 1
Enter value: 11

Enter address: 2
Enter value: 100

Enter address: -1



--------------------------------------------------------------------------------
```

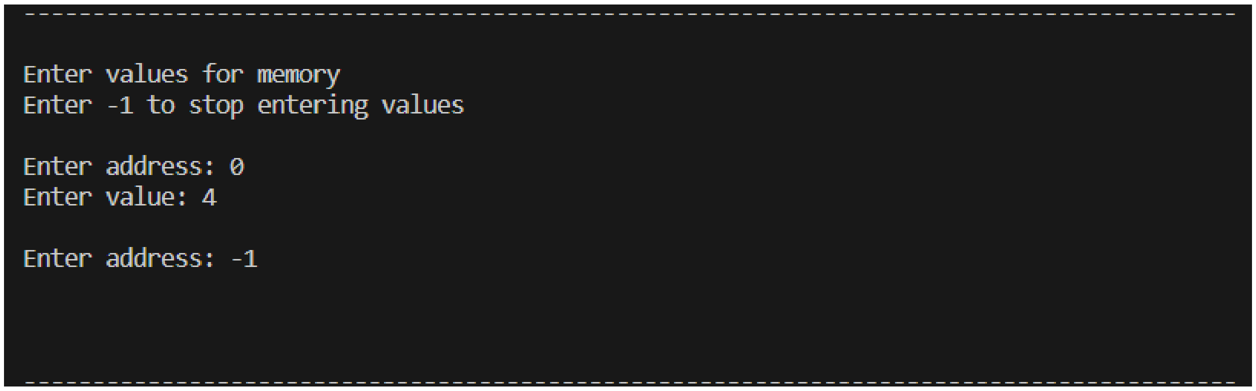


```
 0       LOAD R1 0(R0)          0       1              3              4
 1       LOAD R2 1(R0)          1       2              4              5
 2       ADD R3 R1 R2           2       5              6              7
 3       STORE R3 2(R0)         3       4              6              8
 4       NAND R4 R1 R2          4       5              6              9
 5       DIV R5 R1 R2           5       6              7              10
 6       ADDI R6 R1 60          6       7              8              11
 7       BNE R0 R3 2            7       8              10             12
 8       ADDI R7 R0 20          8       FLUSHED
 12      ADDI R7 R0 21          12      13             14             15
 13      RET                    13      14             16             17
 17      CALL 13                17      18             19             20
 18      ADDI R3 R0 40          18      19             20             21
 Last Write Back occured at cycle 21 And everything was empty at 22
```

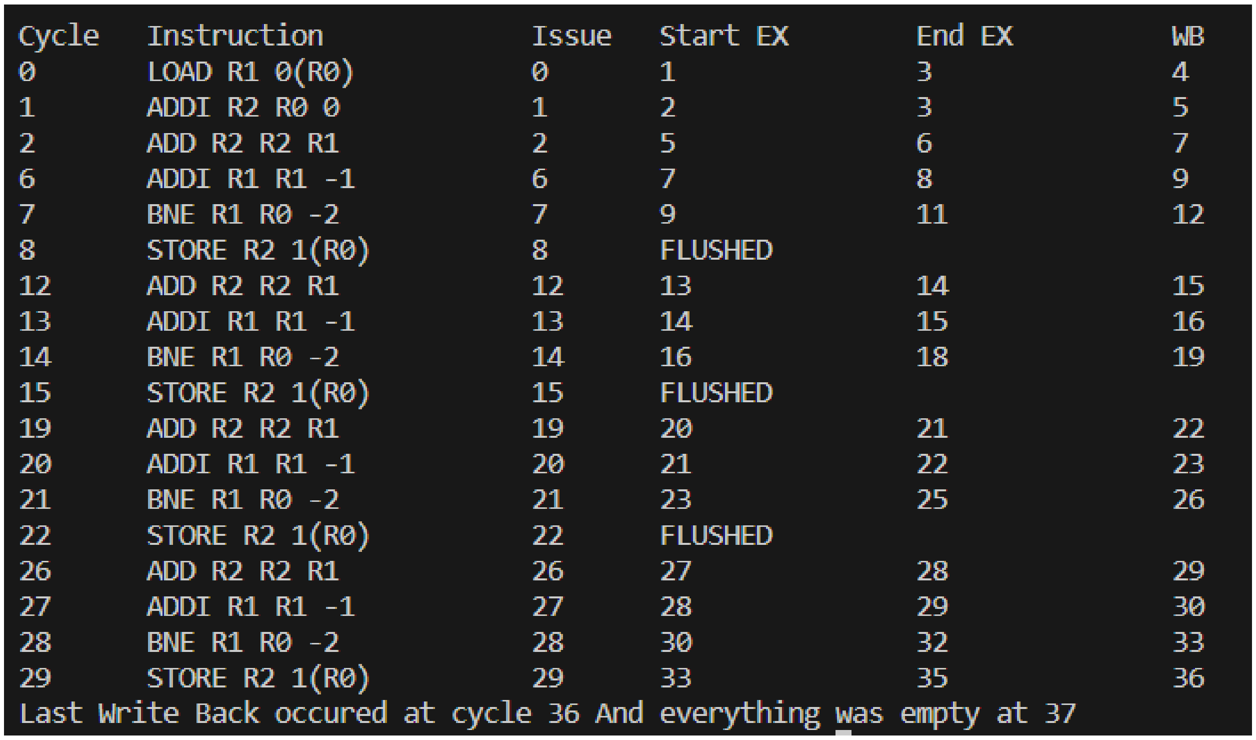
This is a random test code that tests all possible instructions are working as intended and was used for debugging any issues we had. We can see here instructions being flushed due to incorrect prediction we can also see ret jumping as well as some other dependencies like instructions waiting for the common bus to be free to write back(in order)

Test2

```
LOAD R1, 0(R0)
ADDI R2, R0, 0
ADD R2, R2, R1
ADDI R1, R1, -1
BNE R1, R0, -2
STORE R2, 1(R0)
```



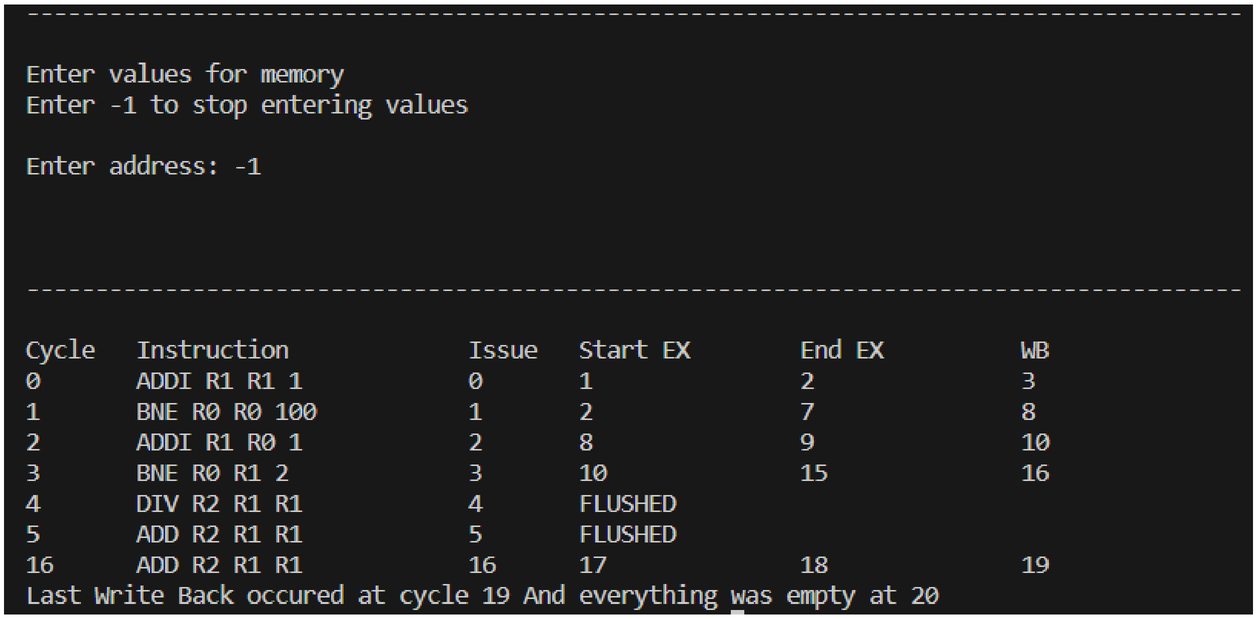
```
------------------------------------------------------------------------------------------

Enter values for memory
Enter -1 to stop entering values

Enter address: 0
Enter value: 4

Enter address: -1



------------------------------------------------------------------------------------------
```



```
Cycle   Instruction            Issue   Start EX        End EX          WB
0       LOAD R1 0(R0)          0       1               3               4
1       ADDI R2 R0 0           1       2               3               5
2       ADD R2 R2 R1           2       5               6               7
6       ADDI R1 R1 -1          6       7               8               9
7       BNE R1 R0 -2           7       9               11              12
8       STORE R2 1(R0)         8       FLUSHED
12      ADD R2 R2 R1           12      13              14              15
13      ADDI R1 R1 -1          13      14              15              16
14      BNE R1 R0 -2           14      16              18              19
15      STORE R2 1(R0)         15      FLUSHED
19      ADD R2 R2 R1           19      20              21              22
20      ADDI R1 R1 -1          20      21              22              23
21      BNE R1 R0 -2           21      23              25              26
22      STORE R2 1(R0)         22      FLUSHED
26      ADD R2 R2 R1           26      27              28              29
27      ADDI R1 R1 -1          27      28              29              30
28      BNE R1 R0 -2           28      30              32              33
29      STORE R2 1(R0)         29      33              35              36
Last Write Back occured at cycle 36 And everything was empty at 37
```

This Program computes the sum of the first n natural numbers. This obviously includes a loop that adds all the numbers till n that is inputted in the first memory location, the result is then stored back in the memory.

Test 3
ADDI R1, R1, 1
BNE R0, R0, 100
ADDI R1, R0, 1
BNE R0, R1, 2
DIV R2, R1, R1
ADD R2, R1, R1



```
------------------------------------------------------------------------------------------

Enter values for memory
Enter -1 to stop entering values

Enter address: -1




------------------------------------------------------------------------------------------


Cycle   Instruction             Issue   Start EX        End EX          WB
0       ADDI R1 R1 1            0       1               2               3
1       BNE R0 R0 100           1       2               7               8
2       ADDI R1 R0 1            2       8               9               10
3       BNE R0 R1 2             3       10              15              16
4       DIV R2 R1 R1            4       FLUSHED
5       ADD R2 R1 R1            5       FLUSHED
16      ADD R2 R1 R1            16      17              18              19
Last Write Back occured at cycle 19 And everything was empty at 20
```

In this program we demonstrate how 2 consecutive branches wouldnt break the program with the prolonged execution time, here we made the branch execution time 5 instead of 2. This is handled correctly as mentioned above.