Tomasulo Algorithm Simulation Report

1st Maya Hussein
Computer Science & Engineering
The American University in Cairo
Cairo, Egypt
mayamakram@aucegypt.edu

2nd Nada Badawi
Computer Science & Engineering
The American University in Cairo
Cairo, Egypt
nada_badawi@aucegypt.edu

I. Introduction

The Tomasulo's Algorithm Simulation was developed using python. The code consisted of 3 main classes.

1) Reservation Station
2) Tomasulo
3) Main Menu

The Reservation station contained all the necessary attributes for a typical reservation station, including but not limited to index, name, busy, vj, vk, qj, qk, destination register (rd), offset, immediate (A), program counter (pc), result, total executed clock cycles, issue cycle, execute cycle, and executed initially set to False.

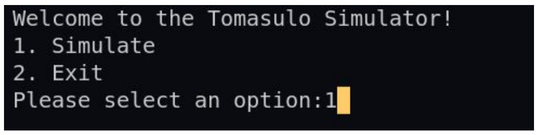
The Tomasulo's Class entails the following attributes: instruction types, instructions, the common data bus (CDB) initially set to True, number of reservation stations (num_rs), clock cycles, global program counter, Register File containing register from R0 to R7, a bool Flush initially set to False. Moreover, the class also holds attributes for the memory including the word size, address size, memory capacity, number of words, an array of memory initially empty or set to zeros. To aid in coding, a branch queue, a bool flag named branch_issued, and another for the jump and link named jal_issued.

The variable hardware bonus feature was implemented. Support a variable hardware organization. The user is able to specify the number of reservation stations for each class of instructions. Additionally, the user also specifies the number of cycles needed by each functional unit type.

II. User Guide

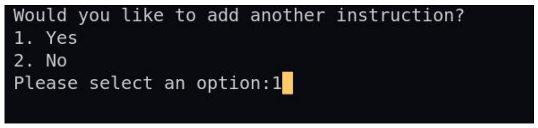
To run the program, the user must have installed the latest version of python installed, and to download and extract the project folder. Upon extraction simply, change directory into the project, and run Tom.py. A command terminal will pop up and have the following interactive interface.

1) The user is prompted to choose between simulating or exiting the program



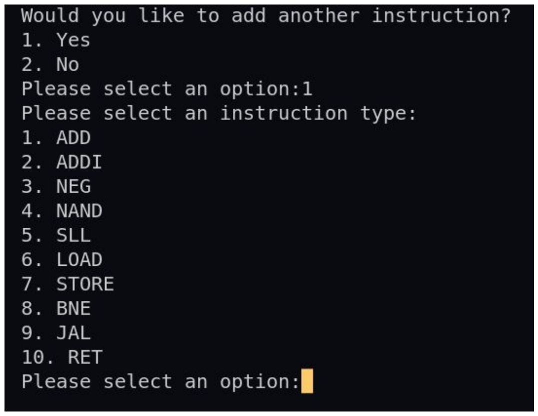
```
Welcome to the Tomasulo Simulator!
1. Simulate
2. Exit
Please select an option:1
```

2) Upon choosing the "simulate" option: the user should choose among 10 options the operation he wishes to select to construct his instruction.

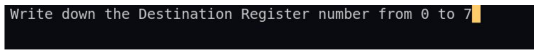


```
Would you like to add another instruction?
1. Yes
2. No
Please select an option:1
```

3) The user will then be given the chance to write a value from 0 to 7 in case of choosing a register or any value to be considered as an offset. The program guides the user with the register type he is currently choosing.

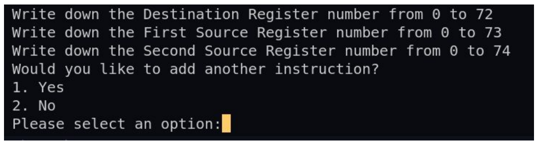


```
Would you like to add another instruction?
1. Yes
2. No
Please select an option:1
Please select an instruction type:
1. ADD
2. ADDI
3. NEG
4. NAND
5. SLL
6. LOAD
7. STORE
8. BNE
9. JAL
10. RET
Please select an option:
```



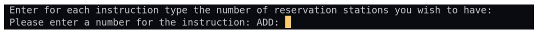
```
Write down the Destination Register number from 0 to 7
```

4) After finishing constructing the instruction by choosing the operation and the source and/or destination registers, the user is asked whether he/she would like to write another instruction. If the user types in a 1, then they are redirected to the same options and steps to construct a new instruction. If the user write a 2, then the user had made a choice to simulate the instructions already written and will see the output of the simulator.



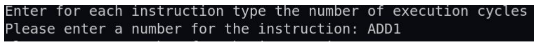
```
Write down the Destination Register number from 0 to 72
Write down the First Source Register number from 0 to 73
Write down the Second Source Register number from 0 to 74
Would you like to add another instruction?
1. Yes
2. No
Please select an option:
```

5) The user is now prompted to enter the number of reservation stations or functional stations that are required for each operation

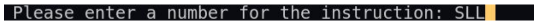


```
Enter for each instruction type the number of reservation stations you wish to have:
Please enter a number for the instruction: ADD:
```

6) The user is now prompted to enter the number of execution cycles for each operation. The operations are listed one by one and the user enters a number to correspond to the listed operation.



```
Enter for each instruction type the number of execution cycles
Please enter a number for the instruction: ADD1
```

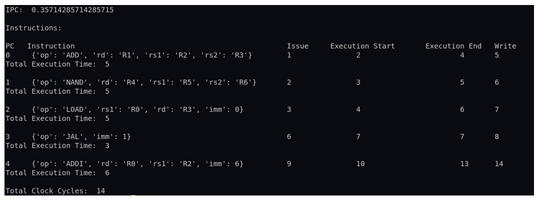


```
 Please enter a number for the instruction: SLL
```

A. Test Results & Discussion

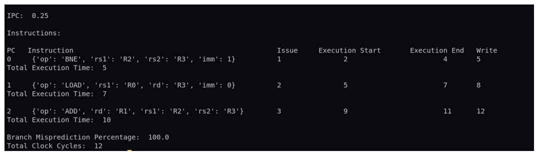
The following were our results to the corresponding test cases found in the test cases folder.

1) In that test case, the first instruction i0, was issued at t1, executed at t2 to t4 as it has 3 clock execution cycles, and writes at t5. The following instructions follow the same methododlogy, but notice how i4 does not issue until JAL is written back. This is because we stall all instractions and prevent them from issuing until we decide where we are going off after the jumping takes place. Bevause of that the program terminates after 14 clock cycles.



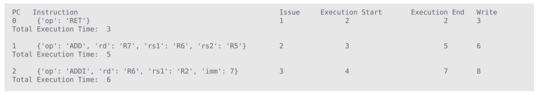
```
IPC:  0.35714285714285715

Instructions:

PC   Instruction                                           Issue     Execution Start       Execution End   Write
0     {'op': 'ADD', 'rd': 'R1', 'rs1': 'R2', 'rs2': 'R3'}       1             2                       4       5
Total Execution Time:  5

1     {'op': 'NAND', 'rd': 'R4', 'rs1': 'R5', 'rs2': 'R6'}      2             3                       5       6
Total Execution Time:  5

2     {'op': 'LOAD', 'rs1': 'R0', 'rd': 'R3', 'imm': 0}         3             4                       6       7
Total Execution Time:  5

3     {'op': 'JAL', 'imm': 1}                                   6             7                       7       8
Total Execution Time:  3

4     {'op': 'ADDI', 'rd': 'R0', 'rs1': 'R2', 'imm': 6}         9             10                      13      14
Total Execution Time:  6

Total Clock Cycles:  14
```

2) Here we see that the Branch Not Equal instruction is in the end take, but the next instructions are automatically issued, but are prevented from execution and its is decided that the branch is taken. This is because the project is based on branch not taken prediction, where the upcoming instruction that have been issued until the execution stage of the branch is over, are placed in a branch queue that allows us to either flush these instructions, or to move forward with the branch not taken prediction. The remaining instructions operate as their normal behavior based on their number of clock cycles.



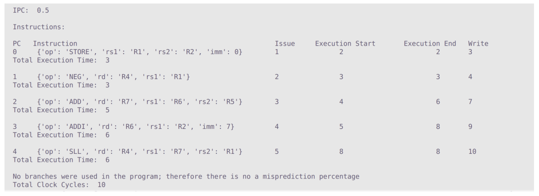
```
IPC:  0.25

Instructions:

PC   Instruction                                           Issue     Execution Start       Execution End   Write
0     {'op': 'BNE', 'rs1': 'R2', 'rs2': 'R3', 'imm': 1}         1             2                       4       5
Total Execution Time:  5

1     {'op': 'LOAD', 'rs1': 'R0', 'rd': 'R3', 'imm': 0}         2             5                       7       8
Total Execution Time:  7

2     {'op': 'ADD', 'rd': 'R1', 'rs1': 'R2', 'rs2': 'R3'}       3             9                       11      12
Total Execution Time:  10

Branch Misprediction Percentage:  100.0
Total Clock Cycles:  12
```

3) In test case 3, we see that return has the same behavior as JAL with the only difference that its address is initially decided in register 1.



```
PC   Instruction                                           Issue     Execution Start       Execution End   Write
0     {'op': 'RET'}                                             1             2                       2       3
Total Execution Time:  3

1     {'op': 'ADD', 'rd': 'R7', 'rs1': 'R6', 'rs2': 'R5'}       2             3                       5       6
Total Execution Time:  5

2     {'op': 'ADDI', 'rd': 'R6', 'rs1': 'R2', 'imm': 7}         3             4                       7       8
Total Execution Time:  6
```

4) These are the remaining instructions that have not been tested in the previous instructions, and they have the same behavior as all the arithmetic instructions that were present in the previous test cases.



```
IPC:  0.5

Instructions:

PC   Instruction                                           Issue     Execution Start       Execution End   Write
0     {'op': 'STORE', 'rs1': 'R1', 'rs2': 'R2', 'imm': 0}       1             2                       2       3
Total Execution Time:  3

1     {'op': 'NEG', 'rd': 'R4', 'rs1': 'R1'}                    2             3                       3       4
Total Execution Time:  3

2     {'op': 'ADD', 'rd': 'R7', 'rs1': 'R6', 'rs2': 'R5'}       3             4                       6       7
Total Execution Time:  5

3     {'op': 'ADDI', 'rd': 'R6', 'rs1': 'R2', 'imm': 7}         4             5                       8       9
Total Execution Time:  6

4     {'op': 'SLL', 'rd': 'R4', 'rs1': 'R7', 'rs2': 'R1'}       5             8                       8       10
Total Execution Time:  6

No branches were used in the program; therefore there is no a misprediction percentage
Total Clock Cycles:  10
```