Dynamic Scheduler: Tomasulo's Algorithm

Sara Mohamed 900203032
Mohamed Noureldin 900203758

Introduction

This project simulates a dynamic scheduler operating on Tomasulo's algorithm, scheduling instructions on a RiSC-16 architecture. This scheduler operates on Tomasulo's algorithm *without* speculation; and supports

Implementation

To implement the scheduler, we needed to implement the following functions. We will go through them file by file.

global.h

The `global.h` file contains the structs and data variables accessed throughout the project. These include:

- `struct RS`: A struct for each reservation station (i.e. a row from the reservation stations table). Note that this does not differ between different functional units. The fields that are not used by certain functional units are simply ignored respectively. The struct includes:
    1. `string name`: name of the functional unit/ reservation station (e.g. `load1` for the first load functional unit or `addaddi3` for the third add/addi functional unit).
    2. `bool busy`: indication of whether or not the functional unit is busy.
    3. `string op`: the operation being carried out.
    4. `int vj`: value of the first source register for the instruction. If instruction has no source register or it is not ready, this is -1.
    5. `int vk`: value of the second source register for the instruction. If instruction has no second source register or it is not ready, this is -1.
    6. `string qj`: in the case that the first source register is not ready, this contains the name of the functional unit that is currently computing the value of the register. If the register is ready, or not needed for the instruction, this is an empty string.
    7. `string qk`: in the case that the second source register is not ready, this contains the name of the functional unit that is currently computing the value of the register. If the register is ready, or not needed for the instruction, this is an empty string.
    8. `int A`: address or immediate for instructions that require it. If not required, this is equal to -1.
    9. `int instructionIndex`:
    10. `int execDuration`: duration for the execution of the functional unit (obviously differs depending on type of functional unit, this is editable by the user).
    11. `int result`: result of computation of the functional unit.

    This struct also has a default constructor, and a constructor to intialize the name and `execDuration`.

- `struct Inst`: A struct for instructions. The struct includes:
    1. `string type`: type or "opcode" of instruction; is simply the first word in the instruction (e.g. `load` for load instructions, `neg` for negation instructions, etc.)
    2. `enum status status`: status of the instruction, defined using an enum "status"; will be defined later in this section.
    3. `int issue`: time t when the instruction is issued (initialized to -1).
    4. `int execFinishTime`: time t when the instruction ends execution (initialized to -1).

5. `int execStartTime`: time t when the instruction begins executing (initialized to -1).
6. `int execRemainingDuration`: time left for instruction to finish executing (initialized to -1).
7. `int write`: time t when the instruction is written (initialized to -1).
8. `int rsIndex`: index of the reservation stations table to which the instruction is assigned.
9. `int rA`: the `rA` of the instruction, if applies.
10. `int rB`: the `rB` of the instruction, if applies.
11. `int rC`: the `rC` of the instruction, if applies.
12. `int imm`: the `imm` of the instruction, if applies.

This struct also has a default constructor, and a constructor to intialize the type, `rA`, `rB`, `rC` and `imm`.

- `instructions`: a vector of instructions, holding the instructions entering the simulator.
- `[type]RS`: a vector of `RS`'s for the instruction type; i.e. there exists a vector `loadRS`, `storeRS`, `addAddiRS`, etc. each for the respective functional unit associated to it. Our big reservation station table is a concatenation of all of these.
- `regMap`: an unordered_map⟨int, string⟩ for the register status table. The key is the index of the register, and the value is the Qi.
- `regMapQueue`: an unordered_map⟨int, queue⟨string⟩⟩ for the register status table. The key is the index of the register, and the value is a queue of functional unit names trying to write to the register.
- `registers`: a vector⟨int⟩ for the registers.
- `memory`: an unordered_map⟨int, int⟩ for the memory. The key is the memory location, and the value is the value at that location.
- `PC`: the PC for the program, initialized at 0.
- `cycles`: the current clock cycle, initialized at 1.
- `afterBranch`: bool for deciding whether or not we are in the instructions after a conditional branch; so we can flush them.
- `waitingForBranch`: vector⟨int⟩ of functional unit names waiting for branch to be written.
- `numberofFinishedInstructions`: int initialized to 0.
- `numberofBranchesEncountered`: int initialized to 0.
- `numberofBranchesMispredicted`: int initialized to 0.
- `enum status`: enum for the status of instructions, contains `NOTYET, ISSUED, EXECUTING, EXECUTED, WRITTEN` options.

deps.cpp

- `bool checkAllFinished()`: loops over reservation stations to check if they are all empty. If so, it checks the last instruction to see if it has been written and returns true if that is the case; else returns false.
- `void emptyRS(int rsIndex, enum rsType rsVector)`: initializes given RStype vector to empty.
- `void notifyAll(string finishedRSName, int resultValue)`: for an instruction that is writing, it calls this function to notify all registers and reservation stations that it has written in this cycle, so that they can make use of the value if needed.
- `void init_RS`: takes as parameters the count and number of cycles for each type of functional units. Initializes that number of functional units with that number of execution cycles (bonus).
- `void init_registers()`: initialized registers to 0, and clears `regMap`.
- `void print_updates()`: used to print updates for the user, prints the reservation tables, register statuses, register contents, PC, cycle number, memory contents, and instruction and branch stats.

parseload.cpp

This file contains some dependencies for the parsing and loading of code and memory into the program.

- `Inst parse_code_line(string instruction)`: takes line from file entered by user, and parses it to return it as an `Inst`.
- `void load_code(string file)`: takes file name and opens it if possible, takes line by line and sends it to `parse_code_line`, pushing the resulting `Inst` into the `instructions` vector.
- `load_memory(string file)`: takes file name and opens it if possible, takes line by line and saves it into the `memory` unordered_map.

issue.cpp

This file contains only the canIssue function.

`bool canIssue(Inst& inst, int t)` takes the instruction to be issued and the current clock cycle t. It first checks if we are after a branch, i.e. if `afterBranch` is true. If so, it pushes the instruction onto the `waitingforBranch` queue and continues.

For each instruction, according to its relevant registers; it checks that a reservation station is available. If so, it then checks the source registers in the `regMap` to see if they are ready. If they are, it puts their value in the `vj` or the `vk` fields of the reservation station row. If not, it puts the name of the functional unit currently producing the value of the register in the `qj` or the `qk` fields of the reservation station row. Then, it sets the destination register, if applicable, in the `regMap` to contain the name of the functional unit. It then sets the `issue` parameter of the instruction to t, and returns true.

If no free functional unit was found, it returns false.

execute.cpp

This file manages the execution of instructions. It contains the following functions:

- `bool canExecute(Inst inst, int t)`: this function takes the issued instruction with the cycle and determines whether or not the instruction is ready to be executed. It first checks if the instruction is waiting for a branch to be executed. Then, it checks the conditions for each instruction type and determines whether or not it can be executed; and returns the associated boolean value.
- `void execute(Inst& inst, int t)`: this function executes the instruction, after it has been sent to `canExecute`, and assigns to the `execStartTime` and `execFinishTime` parameters of the instruction the associated cycle number. It also carries out the necessary operations for the instruction.
- `void executeAfterBranch(bool branched)`: this function executes instructions that come after a branch instruction, depending on the branched boolean passed. If branched is true, then we “flush” the instructions. Otherwise, we remove them from the waitingForBranch queue, and execute them normally.

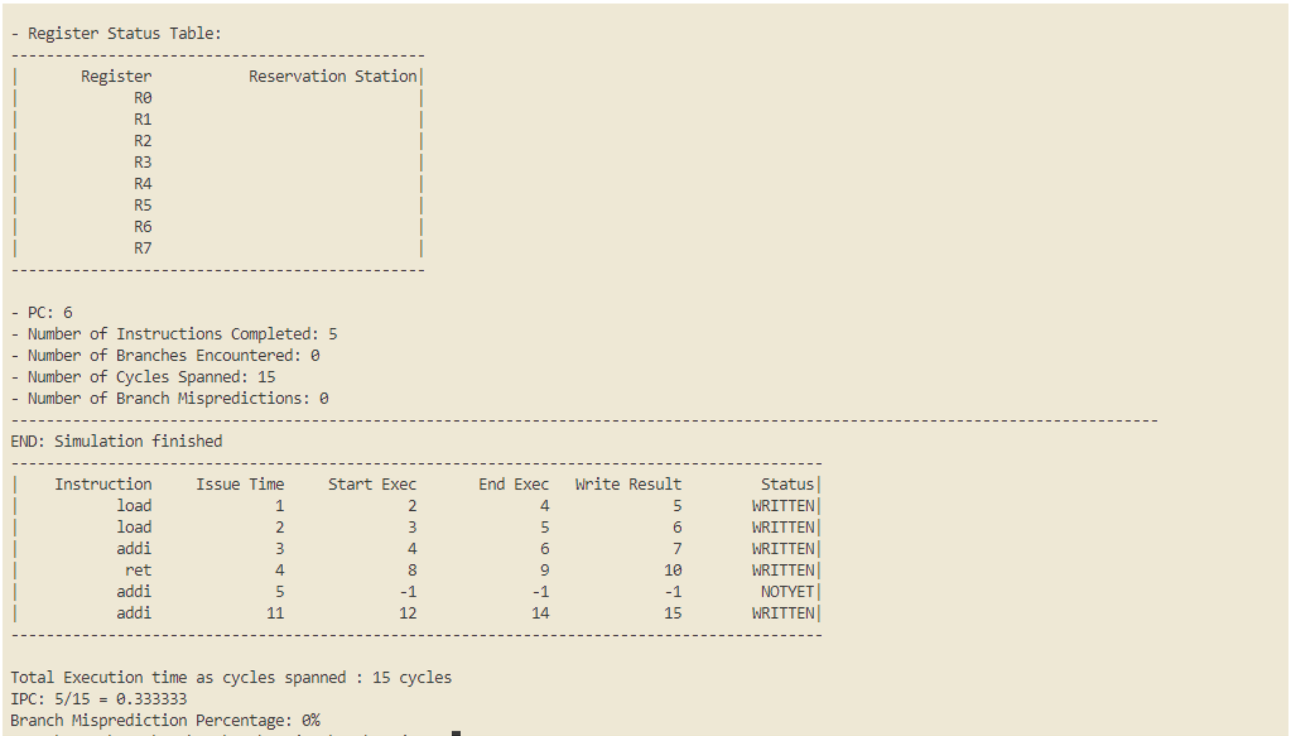
write.cpp

This file manages the writing back of instructions, and contains only one function.
`void write(Inst& inst, int t)`: this function, called after execution of instructions, writes the result of the computations in the corresponding places (memory, register, etc.).

Testing

The submission folder contains a folder with 5 test cases for the program. These test cases cover all 10 instructions, and the simulation of the test cases are correct and accurate, after being traced by us. Below are screenshots of the execution of the test program in test5.txt:

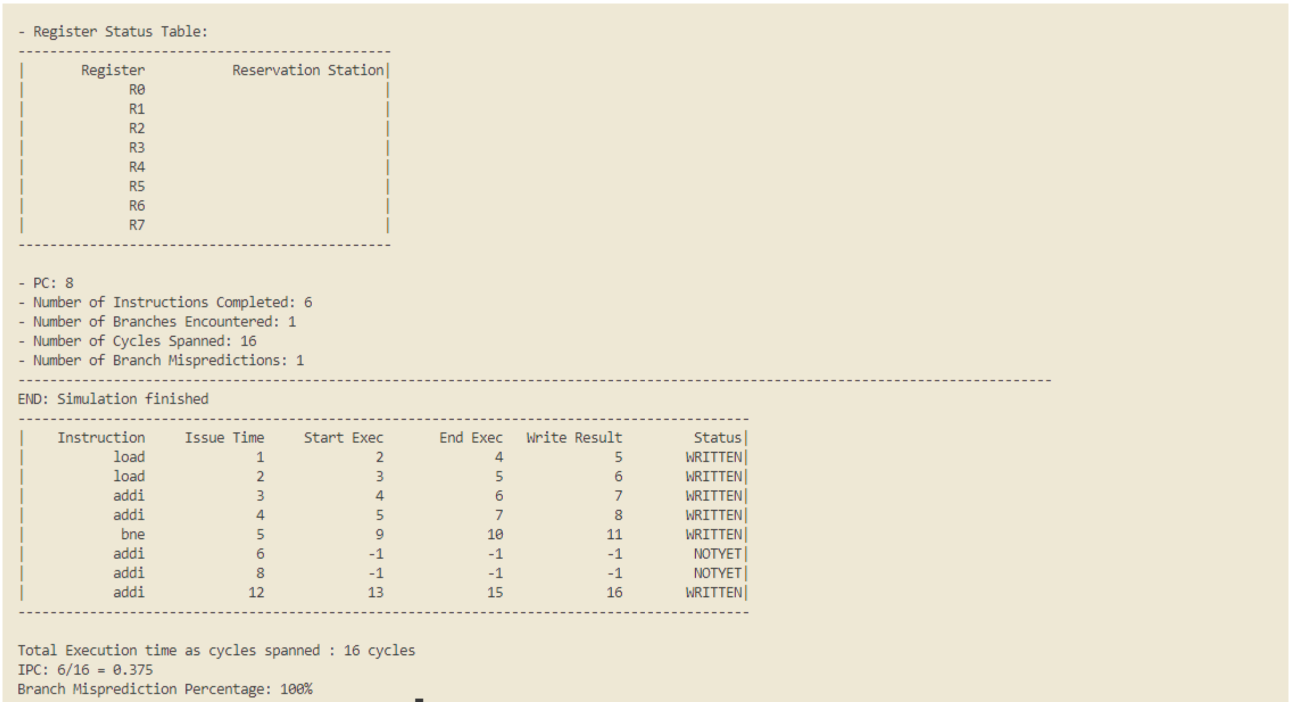
test1.txt



```
- Register Status Table:
--------------------------------------------------
|       Register          Reservation Station|
|             R0                              |
|             R1                              |
|             R2                              |
|             R3                              |
|             R4                              |
|             R5                              |
|             R6                              |
|             R7                              |
--------------------------------------------------

- PC: 6
- Number of Instructions Completed: 5
- Number of Branches Encountered: 0
- Number of Cycles Spanned: 15
- Number of Branch Mispredictions: 0
------------------------------------------------------------------------------------------------------------------------------------------
END: Simulation finished
-----------------------------------------------------------------------------------------------
|    Instruction     Issue Time     Start Exec      End Exec   Write Result        Status|
|           load              1              2             4              5       WRITTEN|
|           load              2              3             5              6       WRITTEN|
|           addi              3              4             6              7       WRITTEN|
|            ret              4              8             9             10       WRITTEN|
|           addi              5             -1            -1             -1        NOTYET|
|           addi             11             12            14             15       WRITTEN|
-----------------------------------------------------------------------------------------------

Total Execution time as cycles spanned : 15 cycles
IPC: 5/15 = 0.333333
Branch Misprediction Percentage: 0%
```

The test program contains 2 load instructions followed by an `addi` that depends on the value loaded in the first instruction. The `addi` result is stored in the R1 register and the `ret` instruction depends on the R1 register and hence the `ret` doesn't start execution before the `addi` writes its result. The `ret` instruction causes the PC to jump the last instruction and hence the `addi` instruction in the middle is issued and then flushed; therefore it only has an issue time and is not issued, executed or written again at the end of the program. The program takes a total of 15 cycles actually to execute only 5 of the 6 instructions. The program contains no branches, and hence no branch misprediction occurred so branch misprediction percentage is 0%

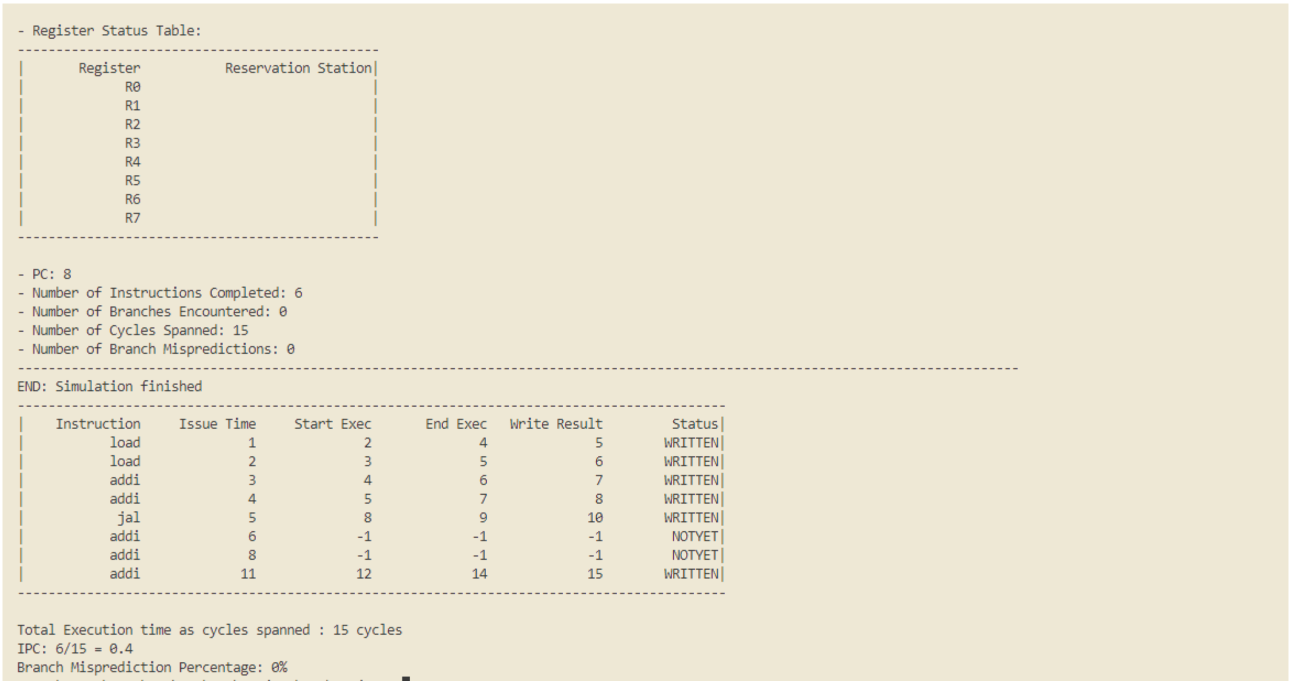
test2.txt



```
- Register Status Table:
--------------------------------------------------
|       Register          Reservation Station|
|             R0                              |
|             R1                              |
|             R2                              |
|             R3                              |
|             R4                              |
|             R5                              |
|             R6                              |
|             R7                              |
--------------------------------------------------

- PC: 8
- Number of Instructions Completed: 6
- Number of Branches Encountered: 1
- Number of Cycles Spanned: 16
- Number of Branch Mispredictions: 1
------------------------------------------------------------------------------------------------------------------------------------------
END: Simulation finished
-----------------------------------------------------------------------------------------------
|    Instruction     Issue Time     Start Exec      End Exec   Write Result        Status|
|           load              1              2             4              5       WRITTEN|
|           load              2              3             5              6       WRITTEN|
|           addi              3              4             6              7       WRITTEN|
|           addi              4              5             7              8       WRITTEN|
|            bne              5              9            10             11       WRITTEN|
|           addi              6             -1            -1             -1        NOTYET|
|           addi              8             -1            -1             -1        NOTYET|
|           addi             12             13            15             16       WRITTEN|
-----------------------------------------------------------------------------------------------

Total Execution time as cycles spanned : 16 cycles
IPC: 6/16 = 0.375
Branch Misprediction Percentage: 100%
```

The test program contains 2 `load` instructions followed by two `addi` instructions that depend on the values loaded in the first two instruction. The `addi` results are stored in the R1 and the R2 registers, respectively. Then, we have a `bne` instruction which jumps to three instructions ahead if the value of R1 is not the same as that of R2. This should happen, and it does; so the program skips the two in between instructions, and executes only the last instruction: an `addi` instruction which puts value 15 in register R4. As a result, the program takes 16 cycles to execute only 6 of the 8 instructions. The program contains a branch that is encountered once and that is taken; so we have a branch misprediction percentage of 100%.

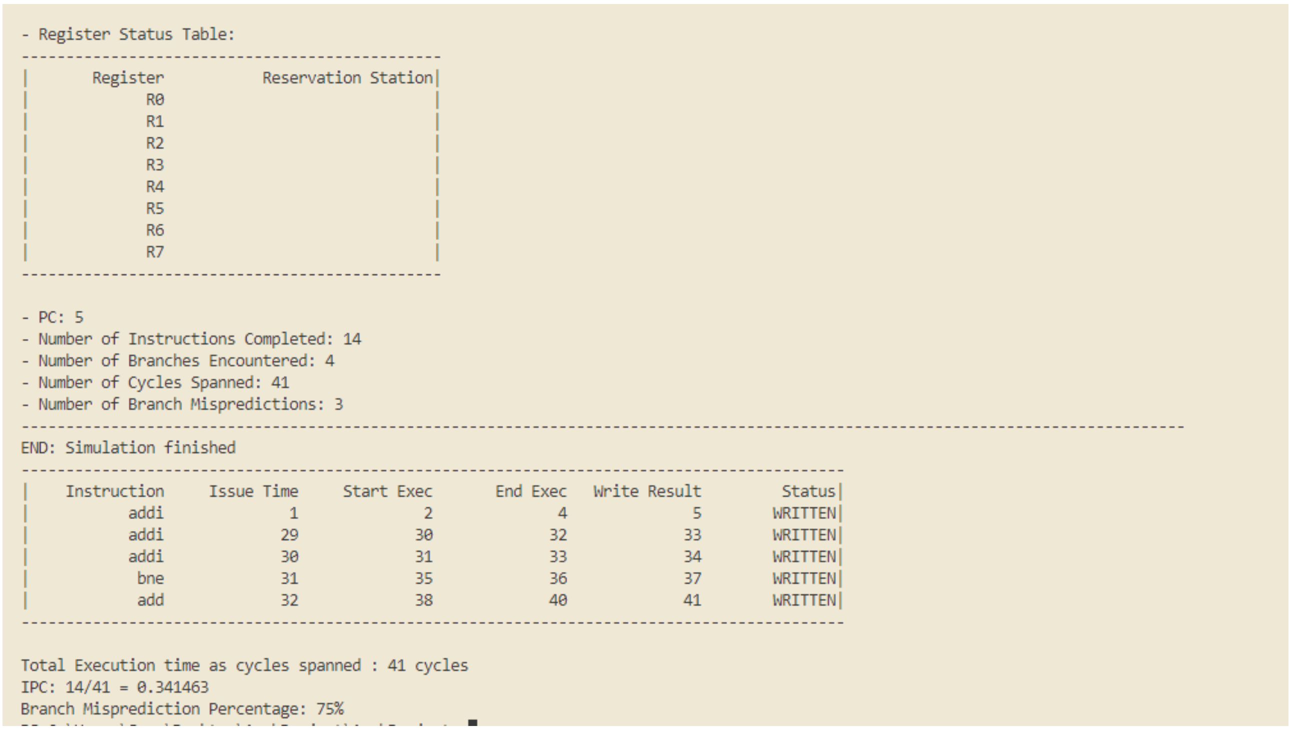
test3.txt



```
- Register Status Table:
----------------------------------------------
|       Register          Reservation Station|
|             R0                             |
|             R1                             |
|             R2                             |
|             R3                             |
|             R4                             |
|             R5                             |
|             R6                             |
|             R7                             |
----------------------------------------------

- PC: 8
- Number of Instructions Completed: 6
- Number of Branches Encountered: 0
- Number of Cycles Spanned: 15
- Number of Branch Mispredictions: 0
------------------------------------------------------------------------------------------------------------------------------------
END: Simulation finished
-------------------------------------------------------------------------------------------
|    Instruction     Issue Time     Start Exec       End Exec   Write Result         Status|
|           load              1              2              4              5        WRITTEN|
|           load              2              3              5              6        WRITTEN|
|           addi              3              4              6              7        WRITTEN|
|           addi              4              5              7              8        WRITTEN|
|            jal              5              8              9             10        WRITTEN|
|           addi              6             -1             -1             -1         NOTYET|
|           addi              8             -1             -1             -1         NOTYET|
|           addi             11             12             14             15        WRITTEN|
-------------------------------------------------------------------------------------------

Total Execution time as cycles spanned : 15 cycles
IPC: 6/15 = 0.4
Branch Misprediction Percentage: 0%
```

The test program aims to test the `jal` instruction along with some other instructions. The program loads 2 values from the memory in registers 1 and 2. The program then overwrites the values of these registers using `addi` instructions to set the actual value of these registers. The program then jumps to the final instruction using `jal` ignoring the 2 `addi` instructions in the middle. Due to the `jal` instruction, the 2 instructions in the middle are flushed and hence they only have issue time however, their state is reverted back to NOTYET. The final `addi` was also issued before the writing of the `jal` result and hence it was reissued again at cycle 11. The program spends a total of 15 cycles and executes 6 instructions out of the 8. The program doesn't contain any branches and hence the branch misprediction percentage is equal to 0

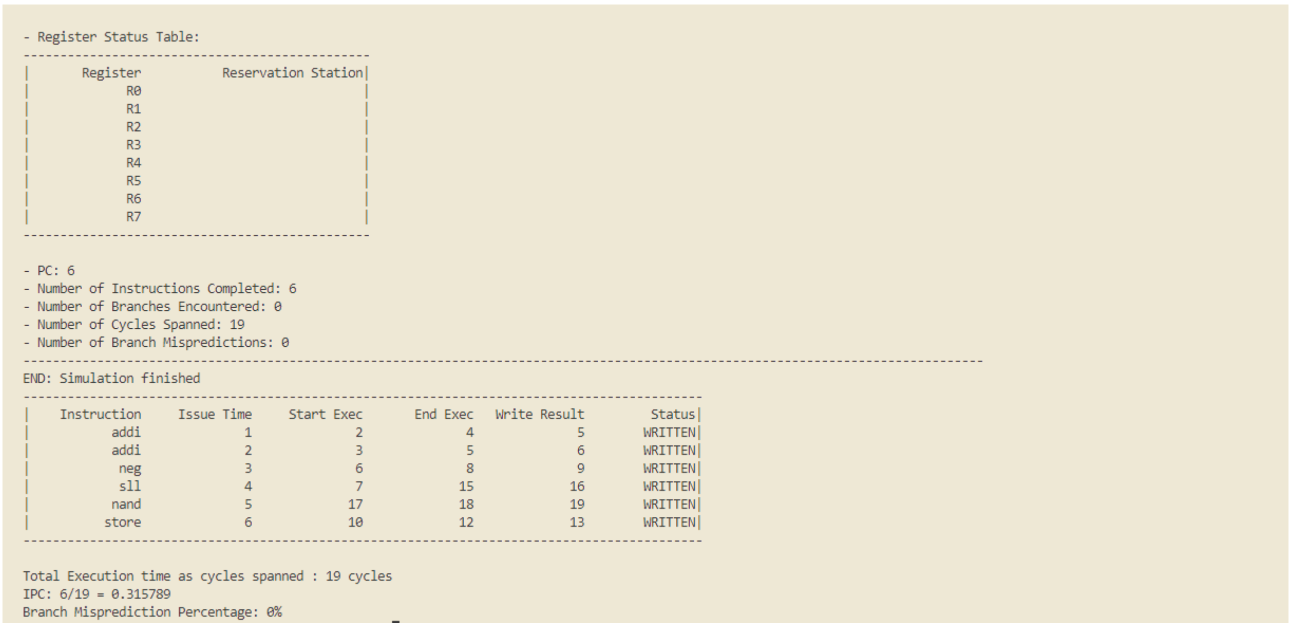
test4.txt



```
- Register Status Table:
-----------------------------------------------
|      Register          Reservation Station|
|            R0                              |
|            R1                              |
|            R2                              |
|            R3                              |
|            R4                              |
|            R5                              |
|            R6                              |
|            R7                              |
-----------------------------------------------

- PC: 5
- Number of Instructions Completed: 14
- Number of Branches Encountered: 4
- Number of Cycles Spanned: 41
- Number of Branch Mispredictions: 3
---------------------------------------------------------------------------------------------------------------------------------
END: Simulation finished
------------------------------------------------------------------------------------------
|    Instruction     Issue Time     Start Exec      End Exec   Write Result         Status|
|           addi              1              2             4              5        WRITTEN|
|           addi             29             30            32             33        WRITTEN|
|           addi             30             31            33             34        WRITTEN|
|            bne             31             35            36             37        WRITTEN|
|            add             32             38            40             41        WRITTEN|
------------------------------------------------------------------------------------------

Total Execution time as cycles spanned : 41 cycles
IPC: 14/41 = 0.341463
Branch Misprediction Percentage: 75%
```

The test program contains 3 `addi` instructions followed by a `bne` instruction intended to create a loop that loops four times. After the `bne` instruction is another `add` instruction which saves in R4 twice the value of R3. The loop is executed four times, which should happen. As a result, the program takes 41 cycles to execute all instructions, including looping (executing some instructions more than once). The program contains a branch that is encountered four times and that is taken three of those times; so we have a branch misprediction percentage of 75%.

test5.txt



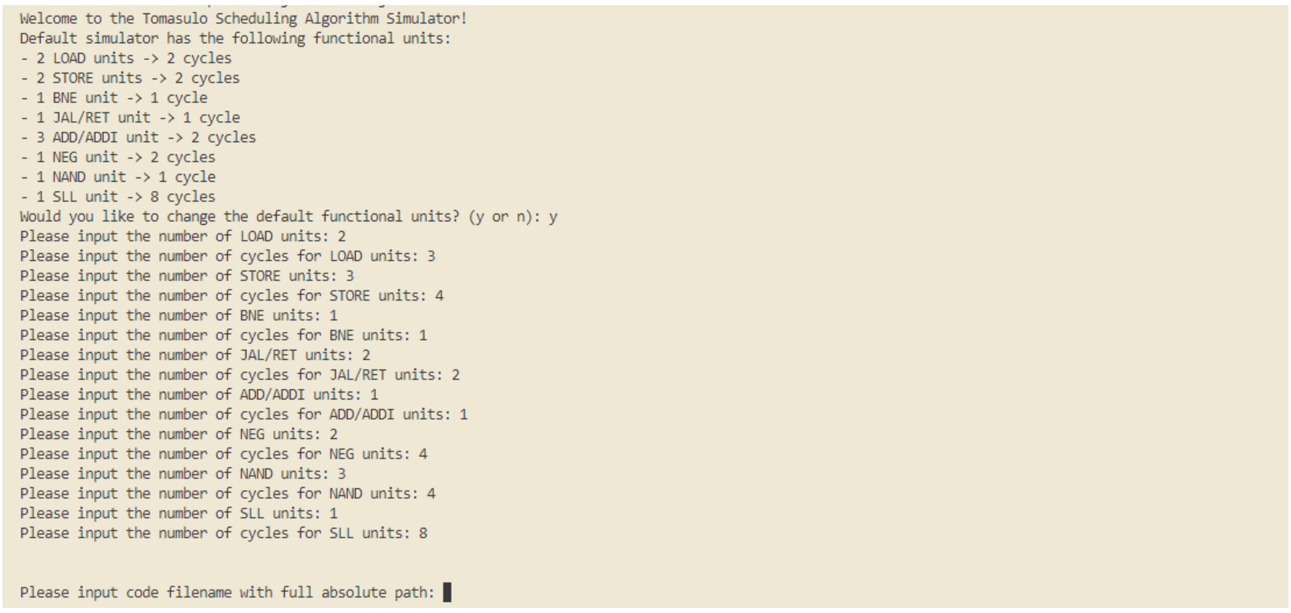
```
- Register Status Table:
-----------------------------------------------
|      Register          Reservation Station|
|            R0                              |
|            R1                              |
|            R2                              |
|            R3                              |
|            R4                              |
|            R5                              |
|            R6                              |
|            R7                              |
-----------------------------------------------

- PC: 6
- Number of Instructions Completed: 6
- Number of Branches Encountered: 0
- Number of Cycles Spanned: 19
- Number of Branch Mispredictions: 0
---------------------------------------------------------------------------------------------------------------------------------
END: Simulation finished
------------------------------------------------------------------------------------------
|    Instruction     Issue Time     Start Exec      End Exec   Write Result         Status|
|           addi              1              2             4              5        WRITTEN|
|           addi              2              3             5              6        WRITTEN|
|            neg              3              6             8              9        WRITTEN|
|            sll              4              7            15             16        WRITTEN|
|           nand              5             17            18             19        WRITTEN|
|          store              6             10            12             13        WRITTEN|
------------------------------------------------------------------------------------------

Total Execution time as cycles spanned : 19 cycles
IPC: 6/19 = 0.315789
Branch Misprediction Percentage: 0%
```

This program aims to test of some of the instructions that were not covered in previous test cases like the `neg`, `sll`, `nand` and `store` instructions. The program starts by putting values in registers 5 and 3 using the `addi` instructions. The negation (2's complement) of the value in register 5 is then computed and since register 5 is being affected by the first `addi` instruction, the execution of the `neg` instruction doesn't start before cycle 6 when the `addi` have already written its result. A `sll` instruction is then issued that shifts the value in the R5 register by the value in the R3 register. Since the value of the R3 register is affected by the second `addi` instruction, the `sll` instruction is not executed before cycle 7 after the `addi` is written. Again the `nand` instruction depends on R6 and R7. The R7 register is updated by the sll instruction and hence the nand is not executed before cycle 17 because the sll is written at cycle 16. The store instruction is executed out of order and hence it is executed at instruction 10 when the value of the R6 is written from the neg instruction. The program uses a total of 19 cycles. The program executes all the 6 instructions. Since the program contains no branches so the branch misprediction percentage is equal to 0%

User Guide

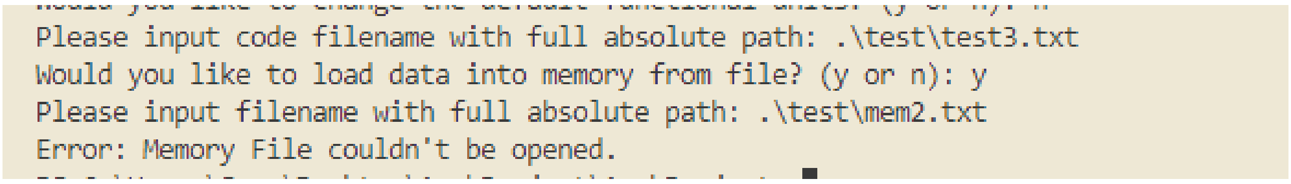
We executed the ***variable hardware organization*** bonus. The user can input the number of each functional unit and the number of cycles required for each. Below are screenshots of the user flow of input. The program also prints the reservation stations, the register statuses, the content of the registers, the content of the memory, the PC, the number of instructions completed, branches encountered, branch mispredictions, cycle spans; at every cycle

Entering functional unit info



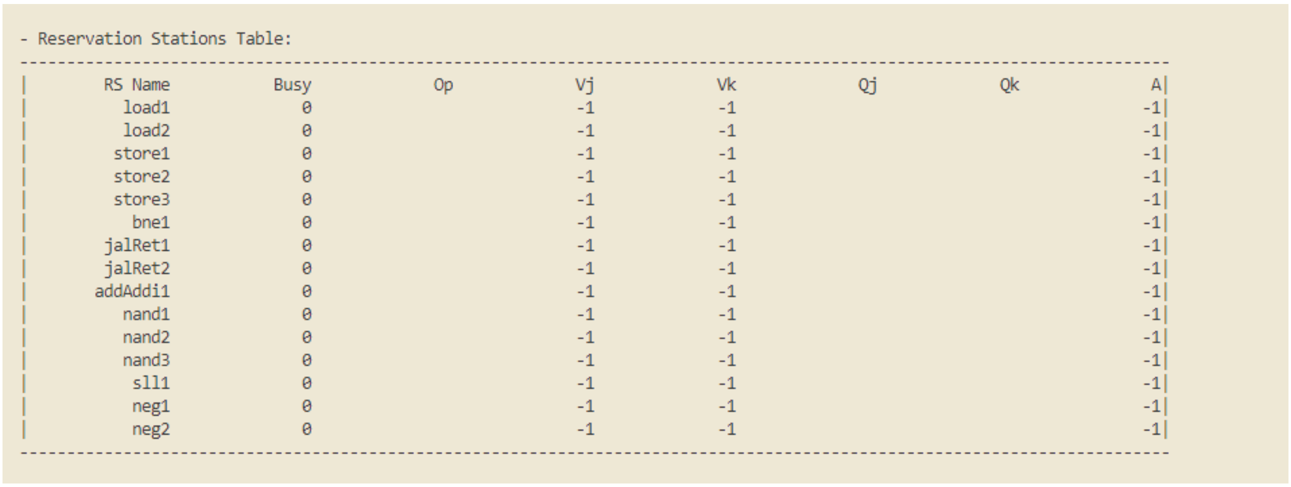
```
Welcome to the Tomasulo Scheduling Algorithm Simulator!
Default simulator has the following functional units:
- 2 LOAD units -> 2 cycles
- 2 STORE units -> 2 cycles
- 1 BNE unit -> 1 cycle
- 1 JAL/RET unit -> 1 cycle
- 3 ADD/ADDI unit -> 2 cycles
- 1 NEG unit -> 2 cycles
- 1 NAND unit -> 1 cycle
- 1 SLL unit -> 8 cycles
Would you like to change the default functional units? (y or n): y
Please input the number of LOAD units: 2
Please input the number of cycles for LOAD units: 3
Please input the number of STORE units: 3
Please input the number of cycles for STORE units: 4
Please input the number of BNE units: 1
Please input the number of cycles for BNE units: 1
Please input the number of JAL/RET units: 2
Please input the number of cycles for JAL/RET units: 2
Please input the number of ADD/ADDI units: 1
Please input the number of cycles for ADD/ADDI units: 1
Please input the number of NEG units: 2
Please input the number of cycles for NEG units: 4
Please input the number of NAND units: 3
Please input the number of cycles for NAND units: 4
Please input the number of SLL units: 1
Please input the number of cycles for SLL units: 8


Please input code filename with full absolute path:
```

Loading data and instruction memory example with invalid memory file



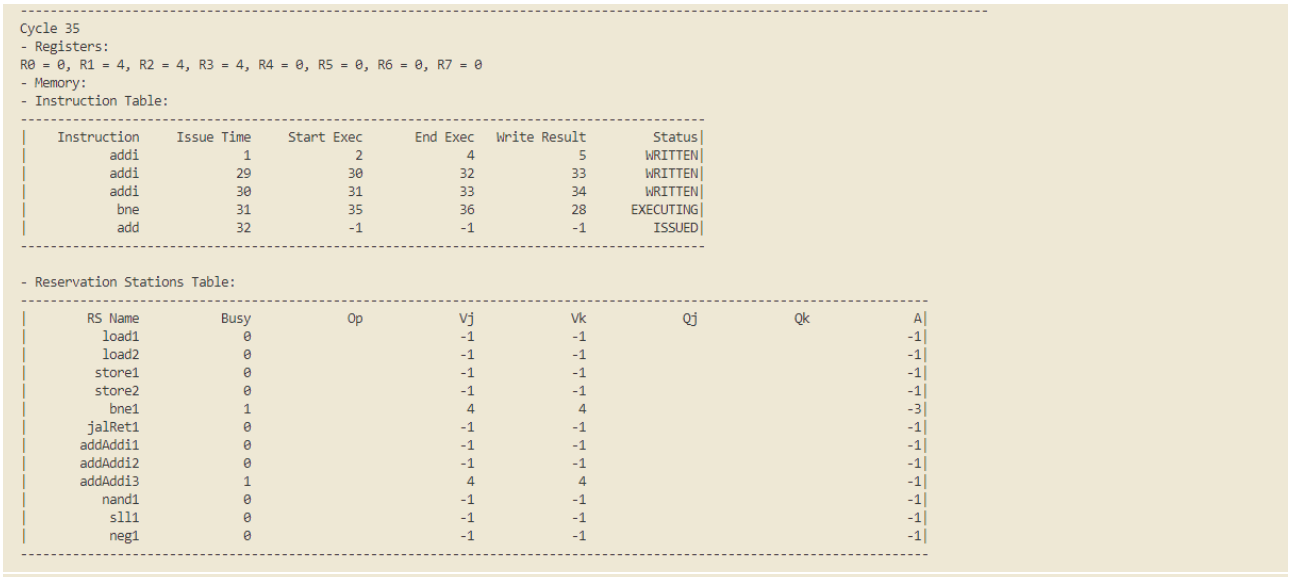
```
Please input code filename with full absolute path: .\test\test3.txt
Would you like to load data into memory from file? (y or n): y
Please input filename with full absolute path: .\test\mem2.txt
Error: Memory File couldn't be opened.
```

Resulting Reservation Stations according to input

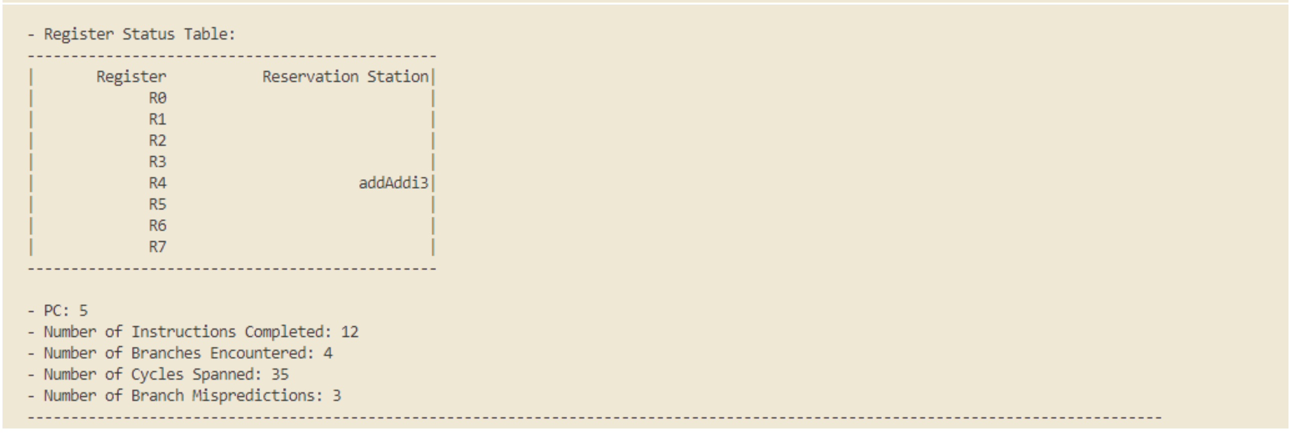


```
- Reservation Stations Table:
------------------------------------------------------------------------------------------------------------------------
|        RS Name          Busy            Op             Vj             Vk             Qj             Qk              A|
|          load1             0                           -1             -1                                           -1|
|          load2             0                           -1             -1                                           -1|
|         store1             0                           -1             -1                                           -1|
|         store2             0                           -1             -1                                           -1|
|         store3             0                           -1             -1                                           -1|
|           bne1             0                           -1             -1                                           -1|
|        jalRet1             0                           -1             -1                                           -1|
|        jalRet2             0                           -1             -1                                           -1|
|       addAddi1             0                           -1             -1                                           -1|
|          nand1             0                           -1             -1                                           -1|
|          nand2             0                           -1             -1                                           -1|
|          nand3             0                           -1             -1                                           -1|
|           sll1             0                           -1             -1                                           -1|
|           neg1             0                           -1             -1                                           -1|
|           neg2             0                           -1             -1                                           -1|
------------------------------------------------------------------------------------------------------------------------
```

Example of printing at cycle 35 in test 4



```
------------------------------------------------------------------------------------------------------------------------------------
Cycle 35
- Registers:
R0 = 0, R1 = 4, R2 = 4, R3 = 4, R4 = 0, R5 = 0, R6 = 0, R7 = 0
- Memory:
- Instruction Table:
------------------------------------------------------------------------------------------
|    Instruction    Issue Time    Start Exec      End Exec  Write Result            Status|
|           addi             1             2             4             5           WRITTEN|
|           addi            29            30            32            33           WRITTEN|
|           addi            30            31            33            34           WRITTEN|
|            bne            31            35            36            28         EXECUTING|
|            add            32            -1            -1            -1            ISSUED|
------------------------------------------------------------------------------------------

- Reservation Stations Table:
------------------------------------------------------------------------------------------------------------------------
|        RS Name          Busy            Op             Vj             Vk             Qj             Qk              A|
|          load1             0                           -1             -1                                           -1|
|          load2             0                           -1             -1                                           -1|
|         store1             0                           -1             -1                                           -1|
|         store2             0                           -1             -1                                           -1|
|           bne1             1                            4              4                                           -3|
|        jalRet1             0                           -1             -1                                           -1|
|       addAddi1             0                           -1             -1                                           -1|
|       addAddi2             0                           -1             -1                                           -1|
|       addAddi3             1                            4              4                                           -1|
|          nand1             0                           -1             -1                                           -1|
|           sll1             0                           -1             -1                                           -1|
|           neg1             0                           -1             -1                                           -1|
------------------------------------------------------------------------------------------------------------------------
```



```
- Register Status Table:
----------------------------------------------
|       Register         Reservation Station|
|             R0                             |
|             R1                             |
|             R2                             |
|             R3                             |
|             R4                     addAddi3|
|             R5                             |
|             R6                             |
|             R7                             |
----------------------------------------------

- PC: 5
- Number of Instructions Completed: 12
- Number of Branches Encountered: 4
- Number of Cycles Spanned: 35
- Number of Branch Mispredictions: 3
------------------------------------------------------------------------------------------------------------------------------------
```