

CSCE 3301 Project 2

Report

Department of Computer Science and Engineering, AUC
Submitted to: Dr. Cherif Salama

Done by:

Omar Elsayed	900183884
Mohammed Zaieda	900181473
Kareem Kassab	900182771

Introduction

In this project we implemented Tomasulo's algorithm without speculation in a C++ program. We followed an object-oriented approach with 3 main classes for the Instructions, Reservation stations, and the stations handler.

Implementation Description

Structure:

Main Classes:

- **Instruction:** this is a very essential class that has a name, names for the parameter registers, and other attributes for its status and the clk and its cycles. Objects of this class are used to be issued and to apply the logic of tomasulo to it according to their clocks and execution times.
- **Reservation Station:** this is what makes the fundamental stations that handle the instructions, we made instantiations of these as per the numbers required. They also have attributes to mark the instructions ready or not to compliment the instruction class to execute the algorithm correctly.
- **Reservation Station Handler:** The object of this class is aware of the number of the free and the busy reservation stations for every instruction to be issued, it decrements the number of reservation stations of that type of instructions, and for every instruction to be written, it increments the number of available stations of that instruction. In addition, it has a Boolean method that returns true or false according to whether a reservation station is available or not

Other components:

There are other components apart from the main classes. These components are still important to the functionality. We used structures such as maps to make such components.

Those components were achieved using maps:

- **Register Status**
- **Register File**

Memory: we consider memory as an array of size 128KB, and since it is word addressable, and each word is two bytes, then number of elements in the array is 64KB

Functionality:

- 1- We read the instructions from a file and place them into a 2D vector
- 2- We read the instructions from the vector in an array of type Instruction, and we set the values of the data members each instruction object

3- Note that every instruction has an attribute that represents its state

we mainly have five states

state 0: before issue

state 1: after issue

state 2: executing

state 3: write back

state 4: finished

4- We make a nested loop where the outer loop, loops over the clocks

and inside this loop, we have two steps:

first: we try to issue a new instruction if the issuing conditions are satisfied

if a new instruction was issued, we append it to another array of instructions called backend array

second: We loop over the backend array, we get every instruction and get its state, then according to its state, we test the conditions to forward it to the next state if it is possible, else it remains in its current state

5- The outer loop is an infinite loop , and its stopping condition is when all the instructions finish writing this condition is tested by looping over the instructions in the backend array and check their states, if there is still any instruction that is not in state 4, then when we enter the loop again, else we break the loop

Connecting Reservation stations to the Register status: We created an array of reservation stations, and for every instruction in the backend array, we get its index and created a reservation station in its array with the same index. then we take this index of the reservation station and put it in a data member inside the instruction object. then we get this data member value and place it into the Register Status map in the item whose key is the destination register of the instruction and its value is the index of the reservation station

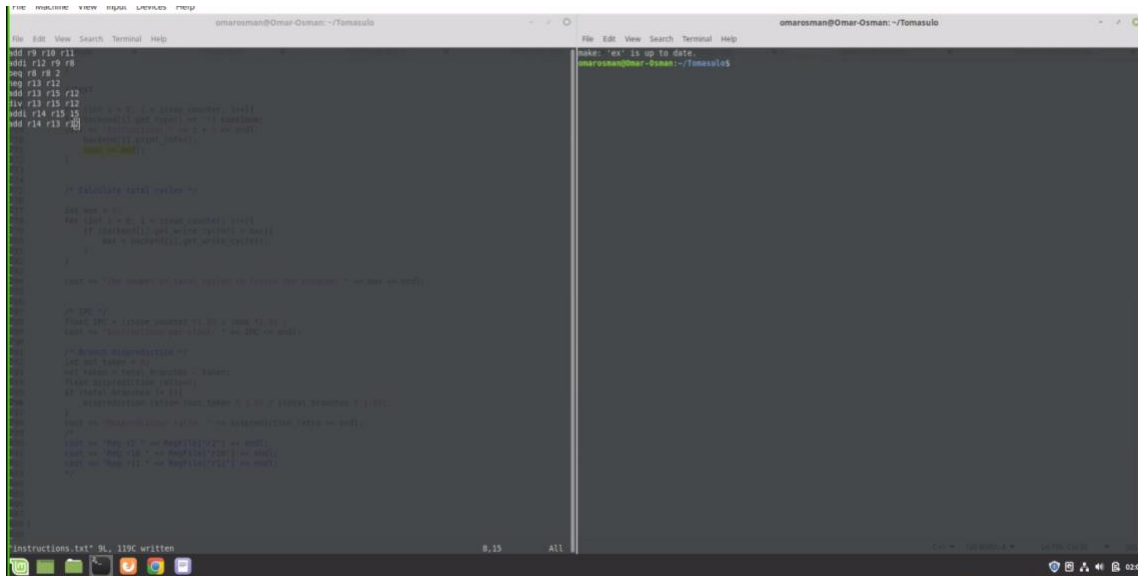
Branch not taken counter: we used that counter to handle the misprediction.

Challenges faced:

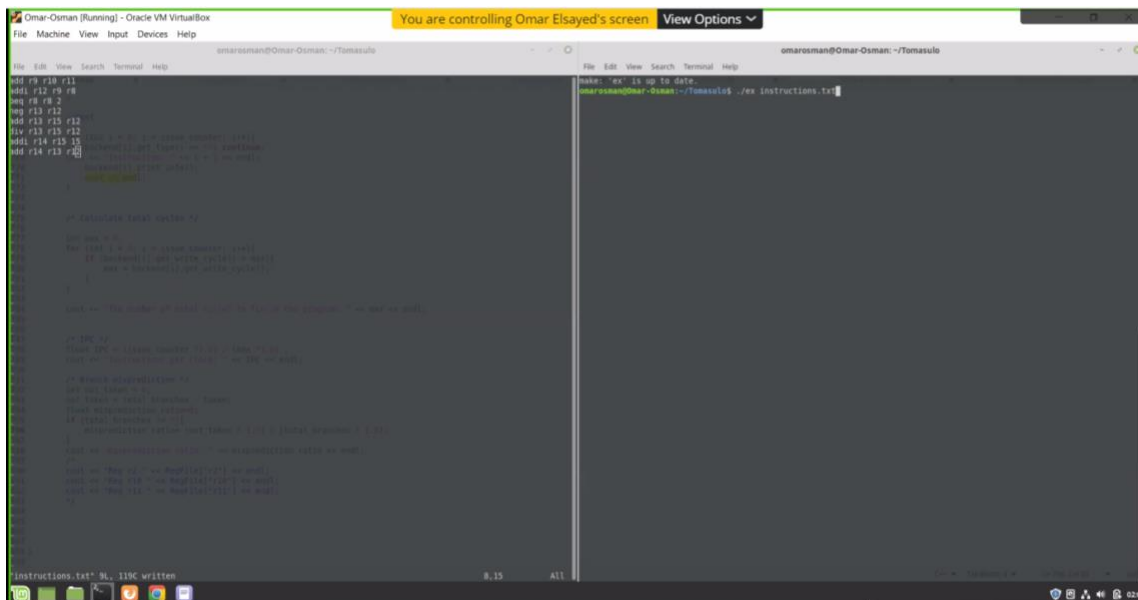
- **RAW hazards:** we used maps for the register status and the register file. In an if condition, simply, we check if one of the source registers is in a currently executing instruction., in the status map, we wait until this value is finished, and then we start executing the instruction. After doing this for each instruction we write the results of the source register, we store the value in the rd and the register file to avoid upcoming RAW hazards and for more program accuracy.
- **PC:** the PC is typically a counter that indicates which instruction is currently issuing; hence, we used the issue counter to represent the PC. This PC value is essential in the beq, jalr, and return instructions. In the beq instructions we wait until the branch result is computed, then we use the next PC which is the next instr. To execute to be the issue counter to represent jumping to the required instruction.
- **Jalr address computation:** for a normal jalr instruction the return address and the jump address are computed in the execute stage

User Guide

- Create a text file
- Write assembly instructions to operate on with the registers separated by a single space
- Save the file

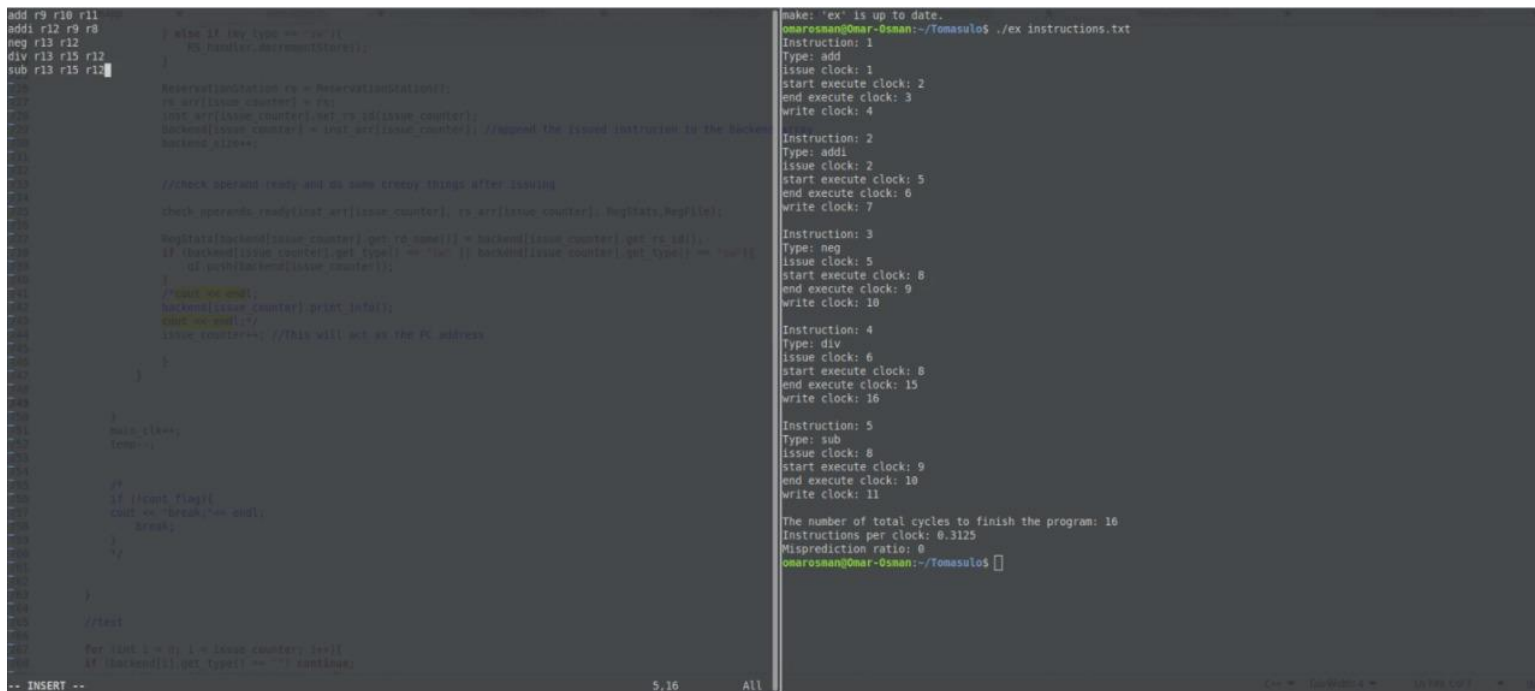


- In a terminal, run the program and pass the file you just saved



-if you press enter, the results will be displayed like the result of the cases shown in the results section.

add, addi, neg, div, subtract:



In this screenshot, we have performed all arithmetic instructions. The first instruction `add` starts issuing at clock cycle 1 and starts execution at 2, ending at 3, and writing in clock cycle 4. The second instruction `addi` starts issuing at clock cycle 2, starts executing at clock cycle 5 handling the RAW hazard of `r9` register to finish from instruction one, and ends executing at clock cycle 6, and writes in clock cycle 7. Third instruction `neg` starts issuing at clock cycle 5 and not 3 because it waited for a reservation station to become empty which is the first instruction's reservation station, starts executing at 8 to handle RAW hazard from instruction 2, and ends at clock cycle 9 and writes at clock cycle 10. Fourth instruction `div` starts issuing at clock cycle 6, starts executing at clock cycle 8 so that it handles RAW hazard of `r12` waiting for writing stage of instruction 2, finish executing at clock cycle 15, and writes at clock cycle 16. The last instruction `sub` starts issuing at clock cycle 8 because it waited for a reservation station to become empty which is instruction 2, starts executing at clock cycle 9 and ends at 10, and then writes at clock cycle 11.

The total number of cycles to finish the program is 16 which is the greatest clock cycle achieved in the tomasulo table. Instructions per clock cycle was calculated by taking the number of clock cycle divided by 11, i.e. $5/16 = 0.3125$ as shown. Branch misprediction is 0 due to no branch instructions added in this test case.

beq:

```

r9 r10 r11
addi r12 r9 r8
beq r8 r0 2
neg r13 r12
add r13 r15 r12
div r13 r15 r12
addi r14 r15 15
add r14 r13 r14

//check operand ready and do some crazy things after issuing
check operands ready(int arxIssueCounter, rx arxIssueCounter, RegStats, RegFile);
RegStats[backendIssueCounter].get rx done() = backendIssueCounter.get rx id();
if (backendIssueCounter.get type() == "rx" || backendIssueCounter.get type() ==
    "si push(backendIssueCounter);
    IssueCounter++; //this will act as the PC address
} else {
    int arxIssueCounter.set status();
    int arxIssueCounter.set issue clock;
    if (my type == "add" || my type == "sub" || my type == "and" || my type == "neg")
        RegHandler.decrementAdd(); //decrement add Reg
    } else if (my type == "div")
        RegHandler.decrementDiv();
    } else if (my type == "mul")
        RegHandler.decrementMul();
    } else if (my type == "shl") || my type == "shr")
        RegHandler.decrementShl();
    } else if (my type == "lwr")
        RegHandler.decrementLwr();
    } else if (my type == "lwr")
        RegHandler.decrementLwr();
    }
}
ReservationStation rx = ReservationStation();
rx arxIssueCounter = rx;
int arxIssueCounter.set rx arxIssueCounter;
backEndIssueCounter = int arxIssueCounter; //append the issued instruction to the backEnd
backEnd.sizes++;

//check operand ready and do some crazy things after issuing
check operands ready(int arxIssueCounter, rx arxIssueCounter, RegStats, RegFile);
RegStats[backEndIssueCounter].get id done() = backendIssueCounter.get rx id();
if (backendIssueCounter.get type() == "rx" || backendIssueCounter.get type() == "si push")
    omarosman@omar-Osman:~/Tomasulo$ ./ex Instructions.txt
Instruction: 1
Type: add
Issue clock: 1
Start execute clock: 2
End execute clock: 3
Write clock: 4

Instruction: 2
Type: addi
Issue clock: 2
Start execute clock: 5
End execute clock: 6
Write clock: 7

Instruction: 3
Type: beq
Issue clock: 3
Start execute clock: 4
End execute clock: 4
Write clock: -1

Instruction: 6
Type: div
Issue clock: 4
Start execute clock: 8
End execute clock: 15
Write clock: 16

Instruction: 7
Type: addi
Issue clock: 5
Start execute clock: 6
End execute clock: 7
Write clock: 8

Instruction: 8
Type: add
Issue clock: 8
Start execute clock: 17
End execute clock: 18
Write clock: 19

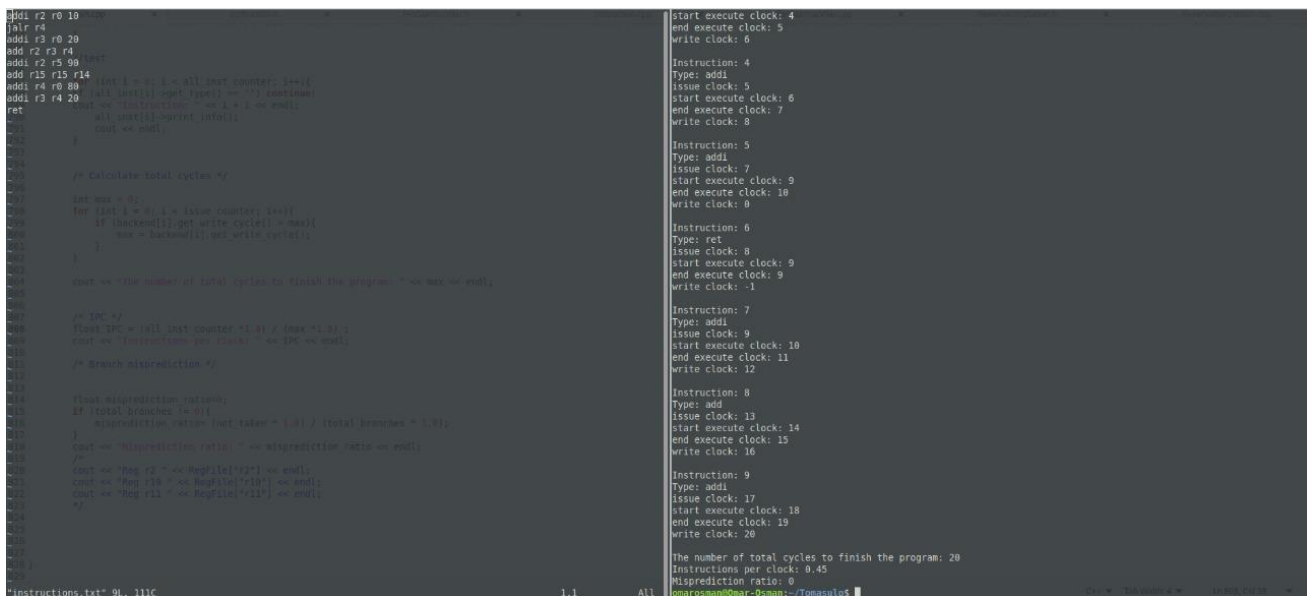
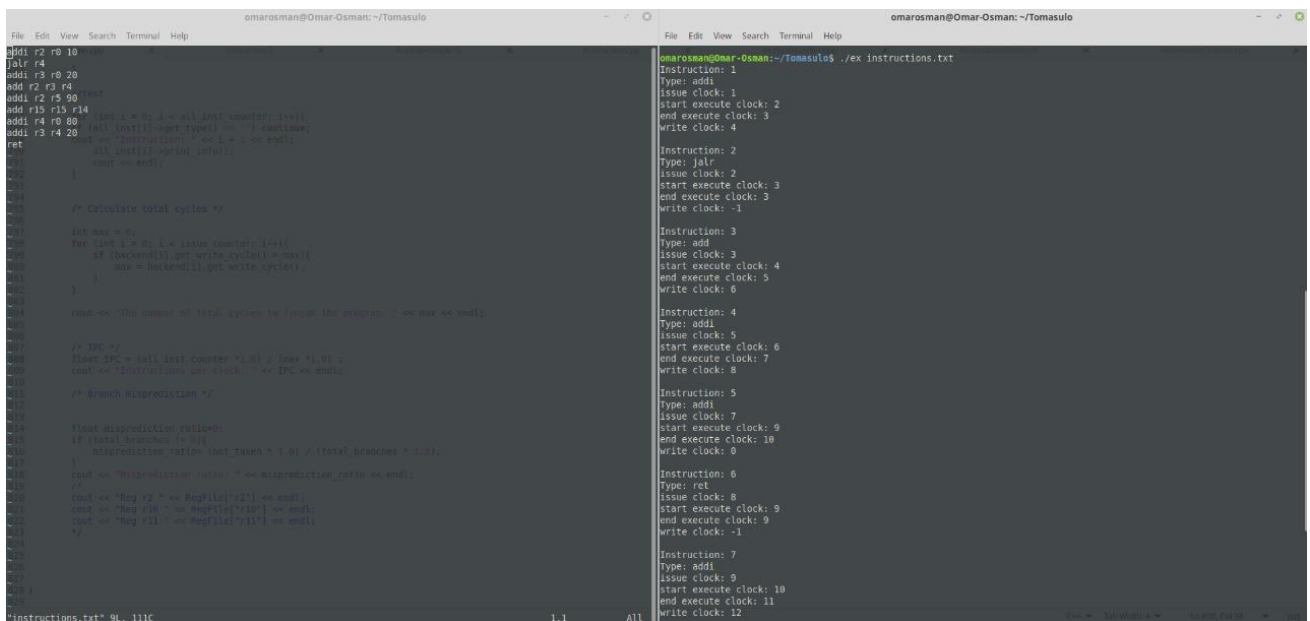
The number of total cycles to finish the program: 19
Instructions per clock: 0.421853
Misprediction ratio: 1
omarosman@omar-Osman:~/Tomasulo$

```

In this program we test the branch logic. The first instruction add is issued in clock cycle 1, starts executing at clock cycle 2, and finishes executing at clock cycle 3, and writes at clock cycle 4. The second instruction starts issuing at clock cycle 2, starts executing at clock cycle 5 because of RAW hazard of register r9 to finish in instruction 1, finishes executing at clock cycle 6, and writes at clock cycle 7. The third instruction branch which starts executing at 3 (has its own reservation station) starts and ends executing at clock cycle 4, and does not write at all, hence the write stage has a negative 1 indicating there is no write back stage. The branch jumps $pc + 2$ instructions which lands on div instruction. Div starts executing at clock cycle 4, starts executing at clock cycle 8 waiting for instruction 2 to finish with register r12 to handle RAW hazards; finished at clock cycle 15 which takes 8 clock cycles as shown, and writes in clock cycle 16. The next instruction addi which starts issuing at clock cycle 5, starts executing at clock cycle 6 and finishes clock cycle 7, and then writes at clock cycle 8. The last instruction starts issuing at clock cycle 8 and not 6 because it waits for an empty reservation station. It starts executing at clock cycle 17 because of RAW hazard from instruction div to finish writing to use register r13, and then finishes executing at clock cycle 18 and writes at clock cycle 19.

The number of total cycles was 19 which is the greatest cycle achieved in the table. The instruction per clock is the total number of instructions divided by 19, which is $8/19 = 0.42$. The misprediction is counted as 1 which is the total not taken branches which is 1 divided by the total branch instructions which is also 1, hence the branch misprediction is 1.

Jalr+ ret:



In this program we test the jalr and ret instructions. The first instruction is addi starts issuing at clock cycle 1, starts executing at clock cycle 2 and finishes at clock cycle 3, and then writes at clock cycle 4. The second instruction jalr jumps to the instruction add which is $pc + 2$. It starts issuing at clock cycle 2, starts and ends executing at clock cycle 3 and does not enter the write back stage as shown is indicated with -1. The add instruction starts issuing at clock cycle 3, starts executing at clock cycle 4 and ends at clock cycle 5, and writes at clock cycle 6. The next instruction addi starts at clock cycle 5

because it awaits for a free reservation station which finishes at clock cycle 4 of instruction 1. The next instruction `addi` starts issuing at clock cycle 7 because it awaits for a reservation station to be free which is instruction 3. It starts executing at clock cycle 9 because it awaits for a RAW hazard of register `r0` from instruction 3, finishes at clock cycle 10 and writes at clock cycle 11 as shown. The next instruction `ret` starts issuing at clock cycle at 8, starts and ends executing at clock cycle 9 and does not write at all hence the the write back stage is indicated with -1. After the `ret` instruction is executed it jumps to the address saved in `r1` which is the instruction after `jalr`. The instructions are issued and executed with the clock cycles shown, and the last instruction is halted and issued at clock cycle 17 to wait for a free reservation station as shown in the screenshot.

The total number of cycles used in this program is the longest clock cycle which is 20. The instruction per clock is calculated by divining the number of instructions: 9 by 20, hence $9/20 = 0.45$ as shown in the picture. The branch misprediction is 0 because there are no branch instructions in this test case.

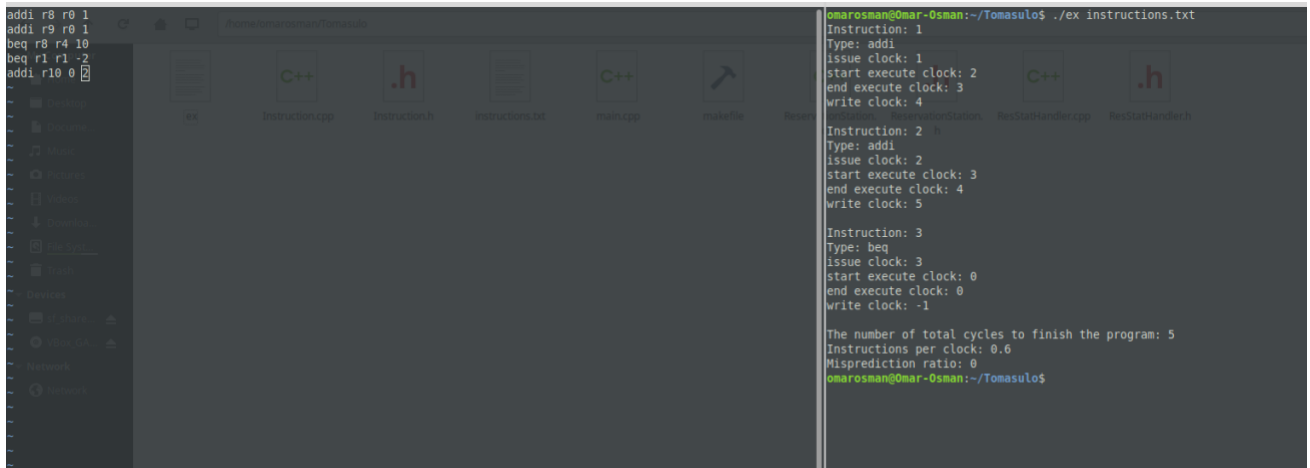
Load

<pre> 1 lw r2 0(r1) 2 addi r2 r0 10 3 add r2 r3 r4 4 jalr r3 5 addi r2 r0 20 6 add r2 r3 r5 7 8 //load 9 else if (backend[i].get_type() == "lw") { 10 backend[i].set_status(2); //executing 11 backend[i].set_start_execute_clk(main_clk + backend[i].get_execution_cycles() - 1); 12 backend[i].set_end_execute_clk(main_clk + backend[i].get_execution_cycles() - 1); 13 issue_counter = RegFile[r1]; //set PC with value in r1 14 } 15 16 //executing stage 17 } else if (backend[i].get_status() == 2) { 18 backend[i].decrement_execution_counter(); //decrement execution cycles 19 if (backend[i].get_execution_counter() == 0) { 20 //finish executing 21 if (backend[i].get_type() == "lw") { 22 RS_handler.incrementReg(); 23 backend[i].set_status(4); 24 } 25 } else if (backend[i].get_type() == "lw") { 26 RS_handler.incrementLoad(); 27 backend[i].set_status(4); 28 } 29 cout << "Register File before: " << RegFile[backend[i].get_rd_name()] << endl; 30 RegFile[backend[i].get_rd_name()] = mem[rs_arr[backend[i].get_rs_id()].getAddr()]; 31 cout << "Register File after: " << RegFile[backend[i].get_rd_name()] << endl; 32 } </pre>	<pre> omarosman@Omar-Osman:~/Tomasulo\$./ex_instructions.txt Type: lw rs1: r1 1 rs2: 0 rd: r2 2 Type: addi rs1: r0 0 rs2: 0 rd: r2 2 Register File before: 2 Register File After: 9 Type: add rs1: r3 3 rs2: r4 4 rd: r2 2 Type: jalr rs1: r3 3 rs2: 0 rd: 0 Type: ret rs1: 0 rs2: 0 rd: 0 </pre>
---	--

Store

<pre> 1 sw r1 0(r1) 2 3 //Convert immediate string to integer and save it in the 2nd instruction 4 //convert string to integer 5 stringToInt(str2Immediate); 6 int imm = 0; 7 temp_str2 = str2; 8 cout << imm << endl; 9 </pre>	<pre> omarosman@Omar-Osman:~/Tomasulo\$ make g++ -g main.cpp Instruction.cpp ReservationStation.cpp ResStatHandler.cpp -o ex omarosman@Omar-Osman:~/Tomasulo\$./ex_instructions.txt Type: sw rs1: r1 1 rs2: r4 4 rd: 0 before store: 9 after store: 4 omarosman@Omar-Osman:~/Tomasulo\$ </pre>
---	---

Loop unrolling



The screenshot shows a code editor with a dark theme. On the left, a file explorer lists files: `main.cpp`, `Instructions.h`, `Instructions.cpp`, `Instructions.txt`, `main.h`, `main.cpp`, `makefile`, and `Results`. The main editor area displays assembly code in the top-left pane and its execution output in the bottom-right pane.

```
addi r8 r0 1
addi r9 r0 1
beq r8 r4 10
beq r1 r1 -2
addi r10 0 2
```

Execution output:

```
omarosman@Omar-Osman:~/Tomasulo$ ./ex instructions.txt
Instruction: 1
Type: addi
issue clock: 1
start execute clock: 2
end execute clock: 3
write clock: 4
Instruction: 2
Type: addi
issue clock: 2
start execute clock: 3
end execute clock: 4
write clock: 5
Instruction: 3
Type: beq
issue clock: 3
start execute clock: 0
end execute clock: 0
write clock: -1
The number of total cycles to finish the program: 5
Instructions per clock: 0.6
Misprediction ratio: 0
omarosman@Omar-Osman:~/Tomasulo$
```

This is a trial to run a loop, but it did not work properly.

Project Experience

We think it was a fruitful experience, but the time was pretty short, as we were approaching the end of the semester. However, we learnt deeply about the Tomasulo dynamic scheduling algorithm.