THE AMERICAN UNIVERSITY IN CAIRO
الجامعة الأمريكية بالقاهرة

10/12/2020

Project 2: femTomas Tomasulo Algorithm Simulation

School of Science and Engineering

Submitted to Dr. Cherif Salama

Submitted by:

Salma Aly	900183330
Menna Elzahar	900182968
Mariam Mousa	900183871

CSCE330102 - Computer Architecture (2020 Fall)
Section 02

A Brief Description of our Implementation Including Any Bonus Features Included

Tomasulo's Algorithm is implemented as will be explained later, and a successful trail to build a simple application was conducted; if the text file containing the assembly code to get executed is put in the local folder of the application, we only need to write the text file name and press enter so that the program run. No need to the original code. This was a trial to implement a bonus part.

The Code Is Divided into The Following Classes and Functions:

Tomasulo: the main function of the code. First, it checks that the instruction has not written yet. If it hasn't, it checks which of the three stages it is in based on three flags: issue, execute and write flags. If any of these flags is false, then the instruction is at this stage and the function executes its segment in the code. Also, the function checks if two instructions are trying to issue or write at the same time using the two signals: issue_flag and write_flag. If in issue stage or write stage one of these flags is one, then the instruction is not issued (or not written) because another instruction is getting issued (or writing). Also, the function is divided into a number of cases, each representing one type of instructions. Each type has its own issue, execute and write function.

instrComplete: this function checks whether all instructions have finished or not using the wb_flag vector (of type bool), which stores which instruction has written back or not. This function is used to decide whether there are any other instructions to process or not.

Reservation Station classes:

1. **load_rs:**
   This is the load reservation station. There are two instances of this station. This reservation station has one source register rs1, one destination register rd, and an immediate. Thus, there are Vj, Qj, A, index, result value, and busy flag. These are the class parameters. Also, there are issuing functions and an execution function. Once their conditions are met in the main.cpp, they are called and the changes needed to be conducted by them occur to fill the reservation station elements as described in lectures 18 and 19. Also, the writing back stage of load happens in main.cpp when the conditions (described in the slides) are met.

2. **store_rs**
   This is the store reservation station. There are two instances of this station. This station was implemented in a very similar way to the load_rs. However, because the sw instructions has 2 source registers and an immediate, the station has the following parameters: Vj, Vk, Qj, Qk, A, index, and busy flag. It also has issuing and execution functions that are called when their conditions (described in the slides) are met in the main.cpp. A load_store queue was implemented so that load/store instructions write in order.

3. **arithm_rs**

This is the arithm reservation station. There are two instances of this station. This reservation station has two source registers rs1 and rs2, one destination register rd, and an immediate. Thus, there are Vj, Vk, Qj, Qk, index, result value, and busy flag. These are the class attributes. Also, there are issuing functions and an execution function. Once their conditions are met in the main.cpp, they are called and the changes needed to be conducted by them occur to fill the reservation station elements as described in lectures 18 and 19. Also, the writing back stage of any arithmetic happens in main.cpp when the conditions (described in the slides) are met.

4. Jalr_ret_rs
   This is the jalr and ret reservation station. There is one instance of this station. This reservation station has only one source register rs1 (and an implicit rd, which is r1). Thus, there are Vj, Qj, A, current_PC (represents the PC at which the instruction is now) and busy flag. These are the class attributes. Also, there are issuing functions and an execution function. Once their conditions are met in the main.cpp, they are called and the changes needed to be conducted by them occur to fill the reservation station elements as described in lectures 18 and 19. Also, the writing back stage of jalr (or ret) happens in main.cpp when the conditions (similar to those in the slides) are met.
5. beq_rs
   This is the beq reservation station. There is one instance of this station. This reservation station has two source registers rs1 and rs2 and an immediate. Thus, there are Vj, Vk, Qj, Qk and A, current_PC (represents the PC at which the instruction is now) and busy flag. These are the class attributes. Also, there are issuing functions and an execution function. Once their conditions are met in the main.cpp, they are called and the changes needed to be conducted by them occur to fill the reservation station elements as described in lectures 18 and 19. Also, the writing back stage of jalr (or ret) happens in main.cpp when the conditions (similar to those in the slides) are met.
6. div_rs
   This is the div reservation station. There is one instance of this station. This reservation station has two source registers rs1 and rs2 and a destination register. Thus, there are Vj, Vk, Qj, Qk, result and busy flag. These are the class attributes. Also, there are issuing functions and an execution function. Once their conditions are met in the main.cpp, they are called and the changes needed to be conducted by them occur to fill the reservation station elements as described in lectures 18 and 19. Also, the writing back stage of div (or ret) happens in main.cpp when the conditions (similar to those in the slides) are met.

instr:

A class that represents any instruction. It has all the instruction's operands. In addition, it contains the issue, start and end execution and write time of the instruction. It also has three flags: issue, execute and write that tells if an instruction finished that stage or not. There is also an attribute that indicates the type of the instruction(type) and the reservation station working on that instruction (rs_name). The function initialize takes the string of the instruction and parses it to fill the attributes of the class.

main:

A while loop was created whose condition checks that all instructions were written back to stop progressing and breaks the loop as a sign that the program has ended so that the total execution time shall be available. To make sure branch instructions do not miss with this condition, the write back flags of instructions between a branch and its target address are set to 1 so that they are counted with all other instructions that haven really been executed and written back. Thus, the condition fits. Inside the while loop, a for loop was created to loop over all instructions each iteration of the while loop (that is, each clock cycle) to see what progress or stalling each instruction should encounter. Inside the for loop, Tomasulo function is called to start the test program.

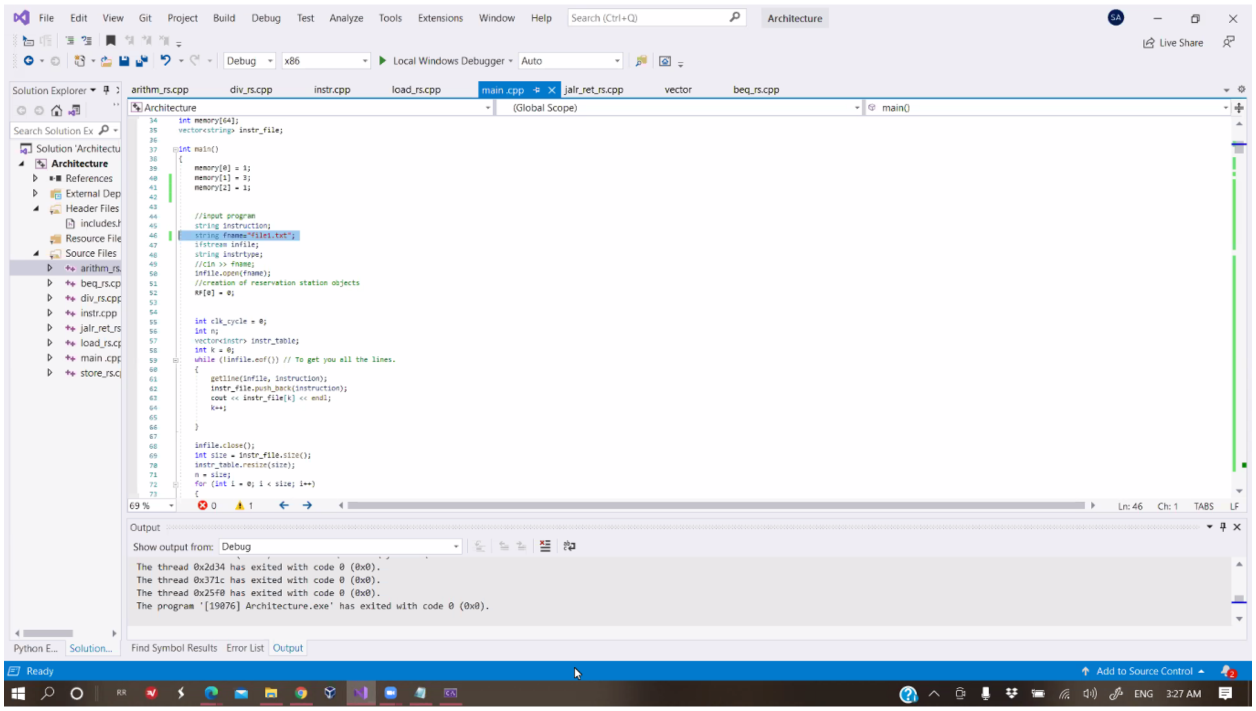
User Guide:

To try the femTomas, first create a .txt file and write your assembly code using the following formats:

```
lw rd, imm(rs1)

sw rs2, imm(rs1)

beq rs1, rs2, imm

jalr rs1

ret

neg rd, rs1

addi rd, rs1, imm

div rd, rs1, rs2
```

Then put the name of the file in the femTomas, then run the code.

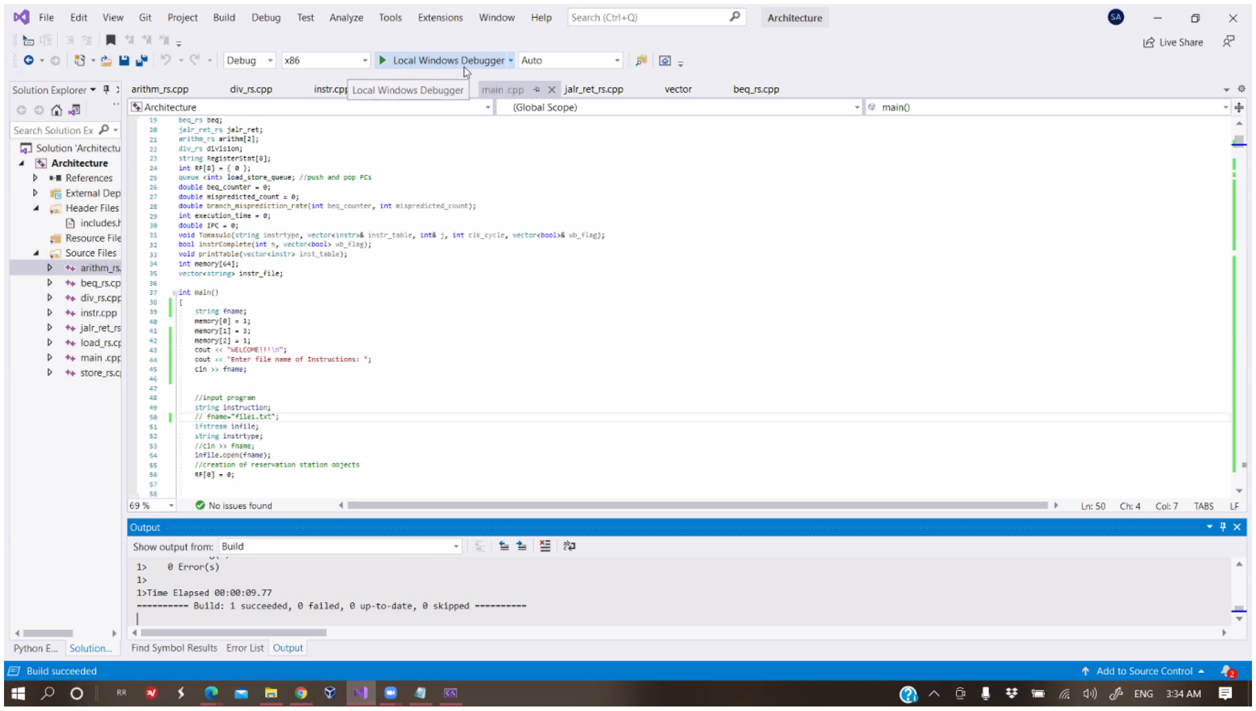
Put the source code in Visual Studio, and make sure the input file name is correct.



```
int memory[64];
vector<string> instr_file;

int main()
{
    memory[0] = 1;
    memory[1] = 3;
    memory[2] = 1;


    //input program
    string instruction;
    string fname="file1.txt";
    ifstream infile;
    string instrtype;
    //cin >> fname;
    infile.open(fname);
    //creation of reservation station objects
    RF[0] = 0;


    int clk_cycle = 0;
    int n;
    vector<instr> instr_table;
    int k = 0;
    while (!infile.eof()) // To get you all the lines.
    {
        getline(infile, instruction);
        instr_file.push_back(instruction);
        cout << instr_file[k] << endl;
        k++;

    }

    infile.close();
    int size = instr_file.size();
    instr_table.resize(size);
    n = size;
    for (int i = 0; i < size; i++)
    {
```

```
The thread 0x2d34 has exited with code 0 (0x0).
The thread 0x371c has exited with code 0 (0x0).
The thread 0x25f0 has exited with code 0 (0x0).
The program '[19076] Architecture.exe' has exited with code 0 (0x0).
```

Press Local Windows Debugger to run the code.



```
beq_rs beq;
jalr_ret_rs jalr_ret;
arithm_rs arithm[2];
div_rs division;
string RegisterStat[8];
int RF[8] = { 0 };
queue <int> load_store_queue; //push and pop PCs
double beq_counter = 0;
double mispredicted_count = 0;
double branch_misprediction_rate(int beq_counter, int mispredicted_count);
int execution_time = 0;
double IPC = 0;
void Tomasulo(string instrtype, vector<instr>& instr_table, int& j, int clk_cycle, vector<bool>& wb_flag);
bool instrComplete(int n, vector<bool> wb_flag);
void printTable(vector<instr> inst_table);
int memory[64];
vector<string> instr_file;

int main()
{
    string fname;
    memory[0] = 1;
    memory[1] = 3;
    memory[2] = 1;
    cout << "WELCOME!!!\n";
    cout << "Enter file name of Instructions: ";
    cin >> fname;


    //input program
    string instruction;
    // fname="file1.txt";
    ifstream infile;
    string instrtype;
    //cin >> fname;
    infile.open(fname);
    //creation of reservation station objects
    RF[0] = 0;

```

```
1>    0 Error(s)
1>
1>Time Elapsed 00:00:09.77
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

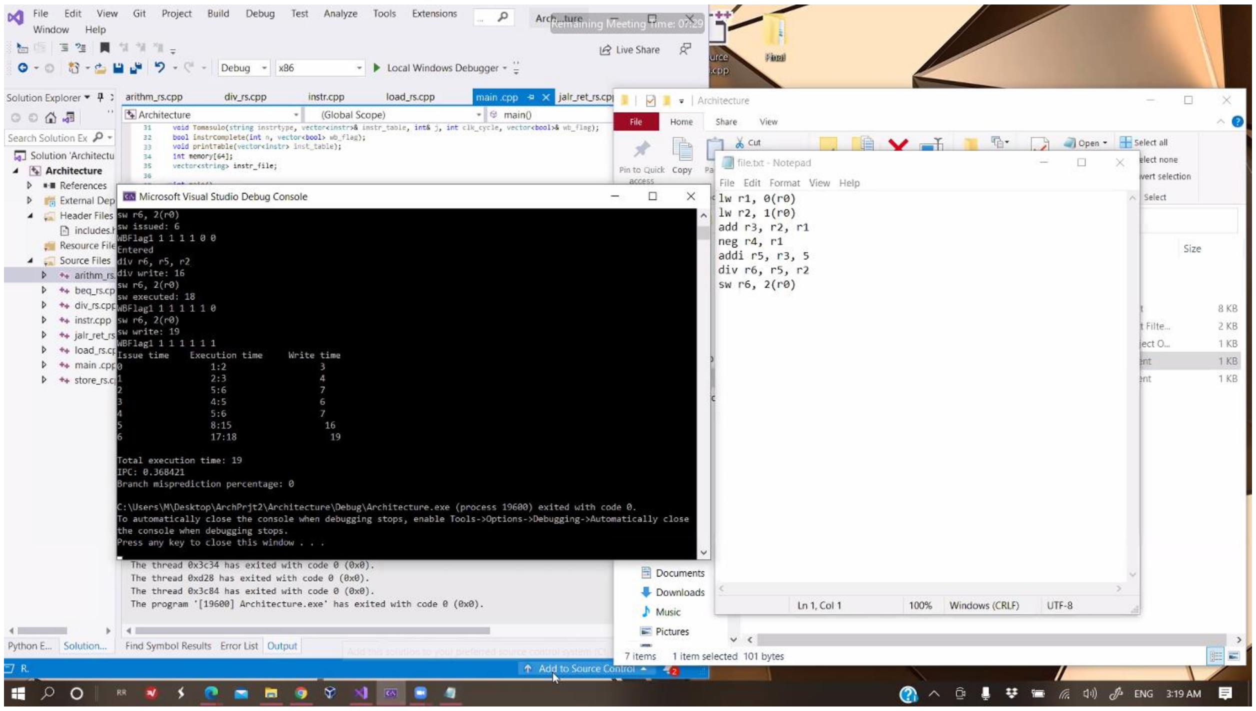
The output shows the following (as illustrated in the results section:

- Issue time, start and end execution time and write time of every instruction.
- IPC
- Total number of cycles

- Branch misprediction ratio

Results:

test1.txt:



```
void Tomasulo(string instrtype, vector<instr>& instr_table, int& j, int clk_cycle, vector<bool>& wb_flag);
bool instrComplete(int n, vector<bool> wb_flag);
void printTable(vector<instr> inst_table);
int memory[64];
vector<string> instr_file;
```

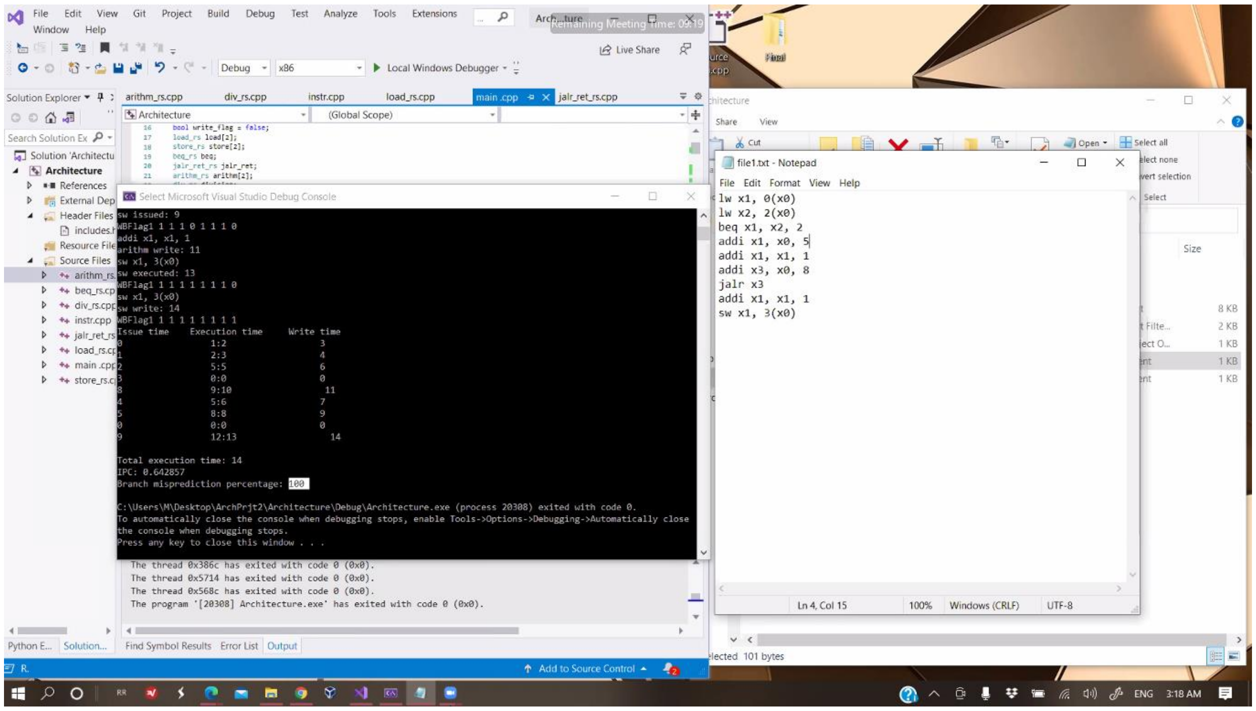
```
sw r6, 2(r0)
sw issued: 6
WBFlag1 1 1 1 1 0 0
Entered
div r6, r5, r2
div write: 16
sw r6, 2(r0)
sw executed: 18
WBFlag1 1 1 1 1 1 0
sw r6, 2(r0)
sw write: 19
WBFlag1 1 1 1 1 1 1
Issue time    Execution time     Write time
0                 1:2                 3
1                 2:3                 4
2                 5:6                 7
3                 4:5                 6
4                 5:6                 7
5                 8:15                 16
6                 17:18                 19

Total execution time: 19
IPC: 0.368421
Branch misprediction percentage: 0

C:\Users\M\Desktop\ArchPrjt2\Architecture\Debug\Architecture.exe (process 19600) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close
the console when debugging stops.
Press any key to close this window . . .
```

```
The thread 0x3c34 has exited with code 0 (0x0).
The thread 0xd28 has exited with code 0 (0x0).
The thread 0x3c84 has exited with code 0 (0x0).
The program '[19600] Architecture.exe' has exited with code 0 (0x0).
```

file.txt - Notepad

```
lw r1, 0(r0)
lw r2, 1(r0)
add r3, r2, r1
neg r4, r1
addi r5, r3, 5
div r6, r5, r2
sw r6, 2(r0)
```

test2.txt:



```
bool write_flag = false;
load_rs load[2];
store_rs store[2];
beq_rs beq;
jalr_ret_rs jalr_ret;
arithm_rs arithm[2];
```

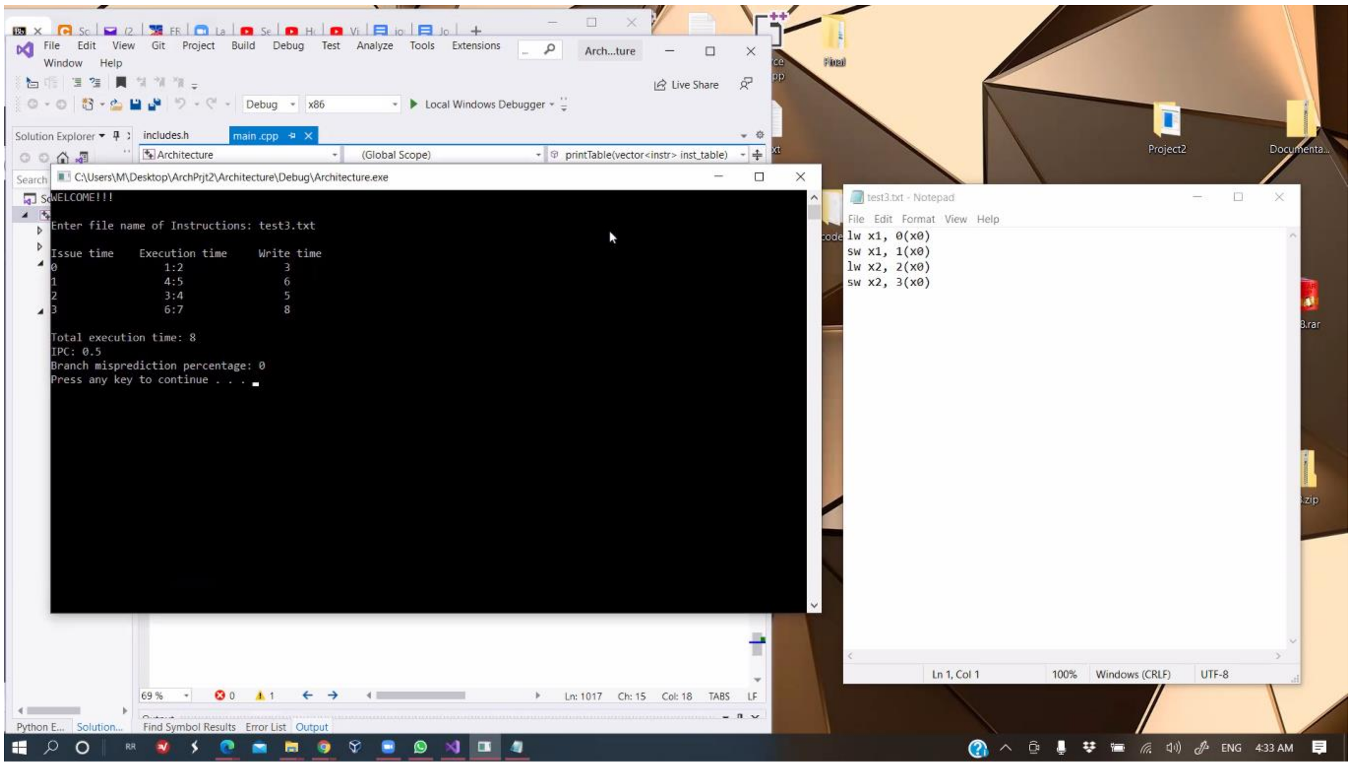
```
sw issued: 9
WBFlag1 1 1 1 0 1 1 1 0
addi x1, x1, 1
arithm write: 11
sw x1, 3(x0)
sw executed: 13
WBFlag1 1 1 1 1 1 1 1 0
sw x1, 3(x0)
sw write: 14
WBFlag1 1 1 1 1 1 1 1 1
Issue time    Execution time     Write time
0                 1:2                 3
1                 2:3                 4
2                 5:5                 6
3                 0:0                 0
8                 9:10                 11
4                 5:6                 7
5                 8:8                 9
0                 0:0                 0
9                 12:13                 14

Total execution time: 14
IPC: 0.642857
Branch misprediction percentage: 100

C:\Users\M\Desktop\ArchPrjt2\Architecture\Debug\Architecture.exe (process 20308) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close
the console when debugging stops.
Press any key to close this window . . .
```

```
The thread 0x386c has exited with code 0 (0x0).
The thread 0x5714 has exited with code 0 (0x0).
The thread 0x568c has exited with code 0 (0x0).
The program '[20308] Architecture.exe' has exited with code 0 (0x0).
```

file1.txt - Notepad

```
lw x1, 0(x0)
lw x2, 2(x0)
beq x1, x2, 2
addi x1, x0, 5
addi x1, x1, 1
addi x3, x0, 8
jalr x3
addi x1, x1, 1
sw x1, 3(x0)
```

test3.txt:


C:\Users\M\Desktop\ArchPrjt2\Architecture\Debug\Architecture.exe
test3.txt - Notepad

```
WELCOME!!!

Enter file name of Instructions: test3.txt

Issue time    Execution time    Write time
0                  1:2               3
1                  4:5               6
2                  3:4               5
3                  6:7               8

Total execution time: 8
IPC: 0.5
Branch misprediction percentage: 0
Press any key to continue . . .
```

```
lw x1, 0(x0)
sw x1, 1(x0)
lw x2, 2(x0)
sw x2, 3(x0)
```

Discussion of results:

test1: here every instruction depends on the one preceding it. No branching instructions are used in this test, hence, the misprediction rate = 0.

test2: in this test, a beq is used to branch to the 5th instruction, hence, the 4th instruction doesn't execute and its execution and write time is printed as 0. Also, a jalr instruction was used to jump to the last instruction (sw), so, the instruction preceding it doesn't issue, execute or write and therefore the issue, execution and write time are printed as 0. There is one issue in this test, which is that the 5th instruction's issue time is late by a few clock cycles. This causes the output to seem like the instructions are not issued in order. However, this is false since the algorithm implemented should force the instructions to be single-issued in order.

test3: in the final test, all dependencies were taken care of. For instance, the first sw only starts to execute when the first lw writes back, and so on.