MIPS Processor for CA Assignment 2

Contributors:

- Rahul Mukundhan (imt2022518)
- Siddharth Palod (imt2022002)

Non-Pipelined MIPS Processor (Processor.java)-

Overview

Overview of a non-pipelined MIPS processor implemented in Java. The processor supports a subset of MIPS instructions and includes key components such as a register file, control paths, data path, and memory.

Register File

The register file consists of 32 general-purpose registers (**$0** to **$31**) and a special register named "LO". Registers are initialized to zero. The "LO" register serves a specific purpose in multiplication operation (MFLO and MULT).
Defined a Java class named **nPipelineRegisters**, representing a set of registers in a pipeline. The class contains a **Map<String, Integer>** named **data** to store register values, where keys are binary strings representing register addresses, and values are initialized to 0. Additionally, there is a **String** field named **writeReg**. The constructor initializes the registers with binary addresses from "00000" to "11111" and includes a special register named "LO," all set to an initial value of 0

Control Paths

The **ControlPaths** class determines control signals for instructions. Signals include register destination (**RegDst**), ALU source (**ALUSrc**), memory register (**MemReg**), register write (**RegWr**), memory read address (**MemRd**), memory write address (**MemWr**), branch (**Branch**), ALU operation (**ALU**), and jump (**JMP**).
Based on ma'am slide we have wrote every instructions control path,
I have added values for ALU to access different functions lateron
Similarly for mem also I have done that as I was trying to implement more functions

The class has several boolean and integer fields, each corresponding to a specific control signal or configuration:

RegDst: Represents the control signal for selecting the destination register.
ALUSrc: Represents the control signal for selecting the ALU (Arithmetic Logic Unit) source.
MemReg: Represents the control signal for memory read.
RegWr: Represents the control signal for register write.

MemRd: Represents the control signal for memory read (as an integer).
MemWr: Represents the control signal for memory write (as an integer).
Branch: Represents the control signal for branching.
ALU: Represents the control signal for the ALU operation (as an integer).
JMP: Represents the control signal for jump instructions.

The class has a parameterized constructor that allows the initialization of these control signals with specific values. The boolean fields (**RegDst**, **ALUSrc**, **MemReg**, **RegWr**, **Branch**, and **JMP**) are initialized with boolean values, while the integer fields (**MemRd**, **MemWr**, and **ALU**) are initialized with corresponding integer values.

Opcode and Function Code Mapping

Instructions are mapped to opcode and function codes using the **iOpcode** and **iFunc** maps.
These mappings facilitate instruction decoding and control signal assignment.
Site used for reference: https://opencores.org/projects/plasma/opcodes

defines two static **HashMaps** in a Java class, named **iOpcode** and **iFunc**. These maps are used to associate MIPS assembly instructions with their corresponding opcode and function code values. Here's a summary:

iOpcode HashMap:

Key: MIPS assembly instruction mnemonic (e.g., "addi", "lw", "j").
Value: Corresponding opcode in binary (e.g., "001000", "100011", "000010").

iFunc HashMap:

Key: MIPS assembly instruction mnemonic (e.g., "addu", "sub", "mflo").
Value: Corresponding function code in binary or **null** for instructions without a function code (e.g., "100001", "100010", "010010").

Static blocks initialize these HashMaps with mappings for various MIPS instructions, associating each instruction with its opcode and function code where applicable.

Fetching Control Path

An instance of the **controlPaths** class named **cp** is created with default values for its fields.
The method then checks the opcode of the input instruction and updates the fields of the **cp** object accordingly based on the opcode.
For R-type instructions (instructions with opcode "000000"):

RegDst, **RegWr**, and **ALUSrc** are set based on the opcode.
MemReg, **MemRd**, **MemWr**, and **Branch** are set to default values.
ALU is determined based on the function code (from the **iFunc** map).
JMP is set to false.

For I-type instructions (e.g., "addi", "lw", "sw", "beq", "bne"):

RegDst, **RegWr**, **ALUSrc**, **MemReg**, **MemRd**, **MemWr**, **Branch**, and **JMP** are set based on the opcode.

ALU is set to a specific value based on the type of I-type instruction.

The method returns the **cp** object with the updated control paths.

Data Path

The **DataPath** class manages the execution stages, including instruction fetch (**IF**), instruction decode (**ID**), execution (**EX**), memory access (**MEM**), and write-back (**WB**). The data path handles the flow of data through these stages, updating the processor's state accordingly.

1. Instruction Fetch (IF)

The **IF** method reads instructions from a machine code file. It uses the program counter (**PC**) to determine the address of the next instruction in memory. Instructions are read in 32-bit chunks (representing one machine instruction) and returned as a string. If the end of the program is reached, it returns null.

String IF(String filename) { // Read machine code instructions from the file
// The program counter (PC) is used to determine the address of the next instruction
// Returns the next instruction as a string or null if the end of the program is reached}

IF (presumably standing for "Instruction Fetch") that takes a file name (**defaultFile**) as input and returns the instruction located at the program counter (PC) position in the file. Here's a summary:

The method attempts to read the contents of the specified file (**defaultFile**) using a **Scanner** and store each line in a **List<String>** named **listLines**.

If the file is not found (FileNotFoundException is caught), an exception is printed, but the program does not terminate; it continues to the next steps.

The lines read from the file are converted to an array of strings (**lines**).

The method then checks if the length of the **lines** array is less than or equal to the calculated index **(PC - 4194380) / 4**. If true, it implies that the program counter is pointing beyond the available instructions in the file. In this case, the method returns **null**.

If there are instructions available at the specified index, the method returns the instruction located at the calculated index in the **lines** array.

2. Instruction Decode (ID)

The **ID** method decodes the instruction, extracting relevant fields such as source and destination registers, immediate values, and control signals. It takes into account whether the instruction is a jump (**JMP**), and if so, it calculates the jump address and updates the program counter (**PC**). The decoded information is then returned as an array.

int[] ID(boolean jmp, String instruct, boolean regDst) { // Decode the instruction, extracting source and destination registers, and immediate values
// If it's a jump instruction, calculate the jump address and update the program counter
// Return an array containing the relevant decoded information }

Jump Instruction Handling (if jmp is true):

If the **jmp** flag is true, it indicates a jump instruction.

The method extracts the jump address from the instruction (**instruct**).

The jump address is formed by concatenating "0000" with bits 6 to 31 of the instruction and appending "00" at the end.

The binary jump address is then converted to an integer.

The program counter (**PC**) is updated to the calculated jump address minus 4 (to account for the default increment of PC).

The method returns an array of three zeros, indicating that no values need to be passed to the next stage.

Regular Instruction Handling (if jmp is false):

The method extracts the source registers (**r1** and **r2**) and the immediate value (**imm**) from the instruction.

If the instruction has a negative immediate value, it is sign-extended.

Depending on the value of **regDst**, the destination register (**r3**) is determined.

The **writeReg** field of the **registerFile** is updated with the destination register (**r3**).

The values of the source registers (**r1** and **r2**) are retrieved from the register file (**registerFile**).

The method returns an array containing the values of the source registers (**res1** and **res2**) and the immediate value (**imm**).

3. Execution (EX)

The **EX** method performs the arithmetic or logical operation specified by the control signals. It takes input values (operands) from the instruction decode stage, performs the operation, and returns the result. This stage also handles branching, updating the program counter based on the control signals.

```
int EX(int output1, int output2, int imm, int aluOp, boolean aluSrc, boolean branch) {
// Perform arithmetic or logical operation based on control signals and input values
// Handle branching, updating the program counter if necessary
// Return the result of the operation }
```

Initialization:

The method initializes local variables **x** and **y** with values from **res1** and **res2**.

A boolean variable **zero** is initialized as false.

ALU Source Handling (aluSrc):

If **aluSrc** is true, it means that the second operand **y** should be replaced with the immediate value **imm**.

ALU Operation (aluOp):

The method performs different operations based on the specified ALU operation code.

If **aluOp** is 0, it performs bitwise AND operation (**x & y**).

If **aluOp** is 1, it performs bitwise OR operation (**x | y**).

If **aluOp** is 2, it performs addition (**x + y**).

If **aluOp** is 3, it performs subtraction (**x - y**). If the result is zero and **branch** is true, it updates the program counter (**PC**) based on the immediate value **imm**.

If **aluOp** is -3, it performs subtraction (**x - y**) with a different condition. If the result is zero and **branch** is true, it updates the program counter (**PC**) based on the immediate value **imm**.

If **aluOp** is 4, it performs a comparison (**x < y ? 1 : 0**) and returns 1 if true and 0 if false.

If **aluOp** is 5, it performs multiplication (**x * y**) and stores the result in the "LO" register.

If **aluOp** is 6, it returns the value stored in the "LO" register.

Return Value:

The method returns the result of the ALU operation.

4. Memory Access (MEM)

The **MEM** method simulates memory access. It takes an address and data as input and performs either a memory read or write operation based on control signals. The method interacts with the simulated memory (**Memory** map) and returns the result of a memory read operation.

int MEM(int address, int writeData, int memRd, int memWr) { // Perform memory read or write operation based on control signals // Interact with the simulated memory (Memory map) // Return the result of a memory read operation }

Memory Read (memRd):

If **memRd** is equal to 1, it indicates a memory read operation.
The method retrieves the value at the specified **address** from a **Memory** data structure (presumably a map or array).

Memory Write (memWr):

If **memWr** is equal to 1, it indicates a memory write operation.
The method updates the value at the specified **address** in the **Memory** data structure with the provided **writeData**.

Return Value:

The method returns the result of the memory read operation (if **memRd** is 1). If no read operation is performed, it returns 0.

5. Write-Back (WB)

The **WB** method updates the processor's state after the execution. It takes the result of the execution, read data from memory, and control signals related to register write (**RegWr**). It updates the program counter and writes the result or memory data back to the register file if necessary.

void WB(int readData, int aluResult, boolean memtoReg, boolean regWr) { // Update the program counter // Write the result or memory data back to the register file if necessary }

Increment Program Counter (PC):

The program counter (**PC**) is incremented by 4. This assumes that each instruction is 4 bytes in size and that the next instruction address follows sequentially.

Check for Register Write (regWr):

If **regWr** is false, indicating that no register write is required, the method returns without performing further operations.

Update Register File:

If **memtoReg** is true, the method updates the register specified by **registerFile.writeReg** with the value of **readData**. This suggests that the data is coming from memory.

If **memtoReg** is false, the method updates the register specified by **registerFile.writeReg** with the value of **aluResult**. This suggests that the data is coming directly from the ALU operation result.

Memory

The processor uses a memory map (**Memory**) to simulate memory. The memory is initialized with addresses ranging from **4** to **112**, with each address holding a 32-bit word. Then run a for loop to make them 0 initially

Execution

The main execution loop in the **main** method fetches instructions from a machine code file, decodes them using control paths, and executes them through the data path. The processor updates register values and memory contents during the execution.

DataPath Initialization:

An instance of the **DataPath** class (**dp**) is created.

Register Initialization (Factorial):

Registers "01001," "01010," and "01011" are initialized with values 3, 4, and 40, respectively.

The value in register "01011" is decremented by 4.

Memory Initialization (Factorial):

A loop is used to read integers from the user and store them in memory locations starting from the value in register "01010."

Instruction Execution Loop:

A loop is entered to execute instructions until there are no more instructions (**machineCode** is null).

Inside the loop, each iteration represents one cycle of the processor.

Instruction Fetch (IF):

The instruction is fetched from the "machinecode2.txt" file.

Instruction Decoding (ID), Execution (EX), Memory Access (MEM), Write Back (WB):

The fetched instruction is processed through the stages of instruction execution using the **ID**, **EX**, **MEM**, and **WB** methods of the **DataPath** object.

Cycle Counting:

The cycle counter is incremented.

Loop Continuation Check:

The loop continues until there are no more instructions (**machineCode** is null).

Print Cycle Count and Final PC Value:

The cycle count and the final value of the program counter (**PC**) are printed.

Print Register Values (Factorial):

The final values of the registers are printed. This is specified for the factorial scenario.

Print Memory Contents (Sorting):

The contents of the memory are printed. This is specified for the sorting scenario.

Example Execution

An example of the execution of a few instructions is provided in the code, demonstrating the processor's functionality and the changes in its state. Code given are machinecode1.txt, machinecode2.txt, which is factorial and sorting respectively

Input/Output Handling

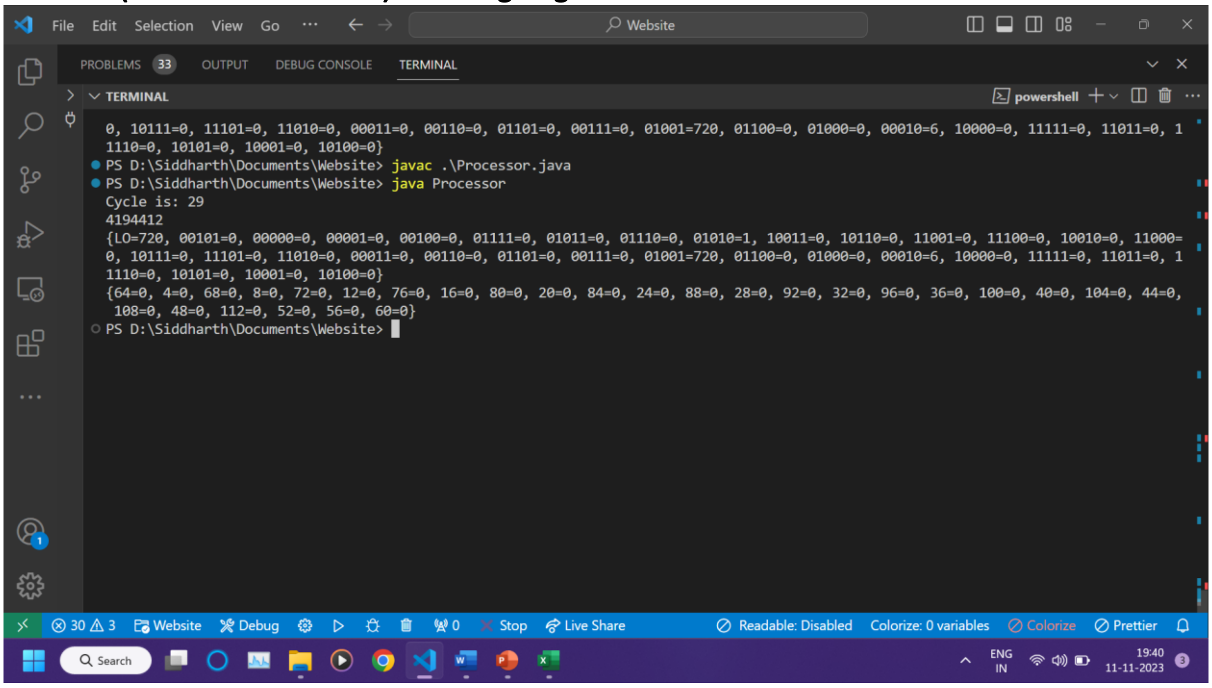
Input is read from a machine code file, and the output includes PC values, register values and memory contents. Machinecode1 demonstrates: factorial and Machinecode2 demonstrates: Sorting

Conclusion

This non-pipelined MIPS processor provides a foundational understanding of processor architecture and execution.

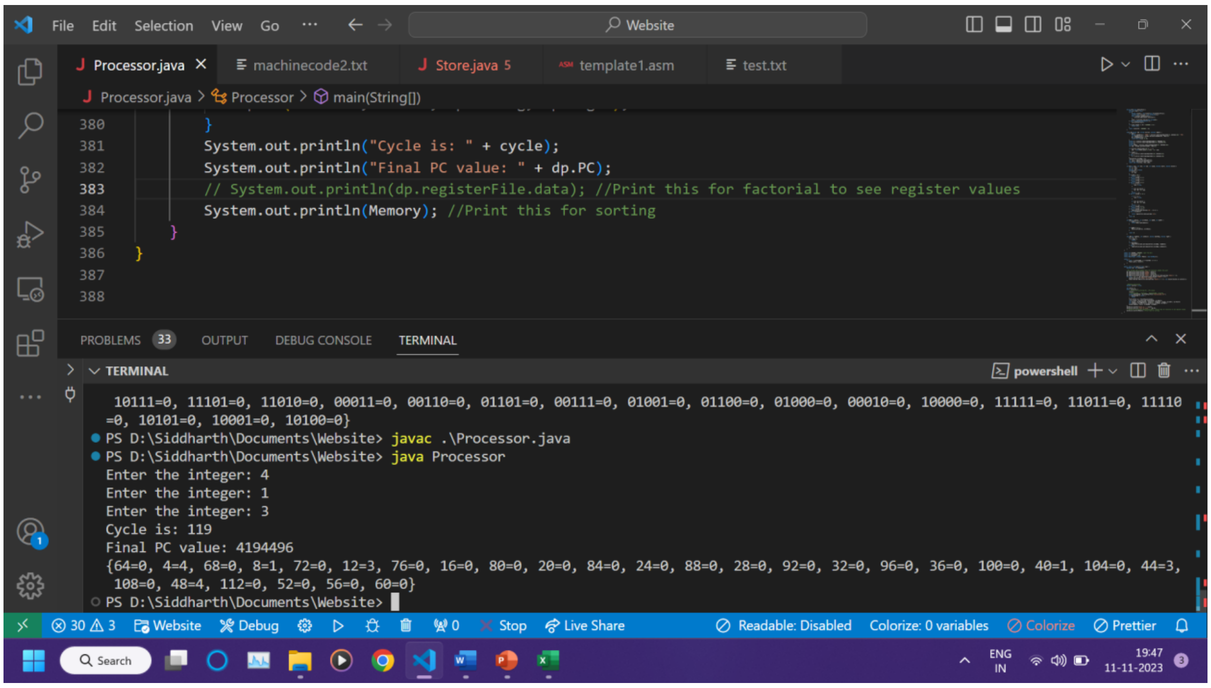
Output

Factorial(machinecode1.txt): Printing Regfile



```
0, 10111=0, 11101=0, 11010=0, 00011=0, 00110=0, 01101=0, 00111=0, 01001=720, 01100=0, 01000=0, 00010=6, 10000=0, 11111=0, 11011=0, 1
1110=0, 10101=0, 10001=0, 10100=0}
PS D:\Siddharth\Documents\Website> javac .\Processor.java
PS D:\Siddharth\Documents\Website> java Processor
Cycle is: 29
4194412
{LO=720, 00101=0, 00000=0, 00001=0, 00100=0, 01111=0, 01011=0, 01110=0, 01010=1, 10011=0, 10110=0, 11001=0, 11100=0, 10010=0, 11000=
0, 10111=0, 11101=0, 11010=0, 00011=0, 00110=0, 01101=0, 00111=0, 01001=720, 01100=0, 01000=0, 00010=6, 10000=0, 11111=0, 11011=0, 1
1110=0, 10101=0, 10001=0, 10100=0}
{64=0, 4=0, 68=0, 8=0, 72=0, 12=0, 76=0, 16=0, 80=0, 20=0, 84=0, 24=0, 88=0, 28=0, 92=0, 32=0, 96=0, 36=0, 100=0, 40=0, 104=0, 44=0,
 108=0, 48=0, 112=0, 52=0, 56=0, 60=0}
PS D:\Siddharth\Documents\Website>
```

Sorting (machinecode2.txt): Printing Memory



```java
        }
        System.out.println("Cycle is: " + cycle);
        System.out.println("Final PC value: " + dp.PC);
        // System.out.println(dp.registerFile.data); //Print this for factorial to see register values
        System.out.println(Memory); //Print this for sorting
    }
}
```

```
 10111=0, 11101=0, 11010=0, 00011=0, 00110=0, 01101=0, 00111=0, 01001=0, 01100=0, 01000=0, 00010=0, 10000=0, 11111=0, 11011=0, 11110
=0, 10101=0, 10001=0, 10100=0}
PS D:\Siddharth\Documents\Website> javac .\Processor.java
PS D:\Siddharth\Documents\Website> java Processor
Enter the integer: 4
Enter the integer: 1
Enter the integer: 3
Cycle is: 119
Final PC value: 4194496
{64=0, 4=4, 68=0, 8=1, 72=0, 12=3, 76=0, 16=0, 80=0, 20=0, 84=0, 24=0, 88=0, 28=0, 92=0, 32=0, 96=0, 36=0, 100=0, 40=1, 104=0, 44=3,
 108=0, 48=4, 112=0, 52=0, 56=0, 60=0}
PS D:\Siddharth\Documents\Website>
```

Pipelined MIPS Processor (pipelineProcessor.py) -

Overview

Overview of a pipelined MIPS processor implemented in Python3(we tried in java but it was not working for us so tried in python). The processor supports a subset of MIPS instructions and has all the same functionalities as non-pipelined code the main changes are in the Mian function of the code

Main

The main execution loop in the **main** method fetches instructions from a machine code file, decodes them using control paths, and executes them through the data path. The processor updates register values and memory contents during the execution.

Hazards detection

This method takes two instructions (current and next) as input.
It extracts relevant parts from the instructions, such as opcodes, source (rs, rt), and destination (rd) registers.
It checks for data hazards based on the extracted information.
If a data hazard is detected, it returns True; otherwise, it returns False.
The conditions for detecting hazards are specific to the MIPS instruction set architecture, checking for certain opcode patterns and register dependencies.

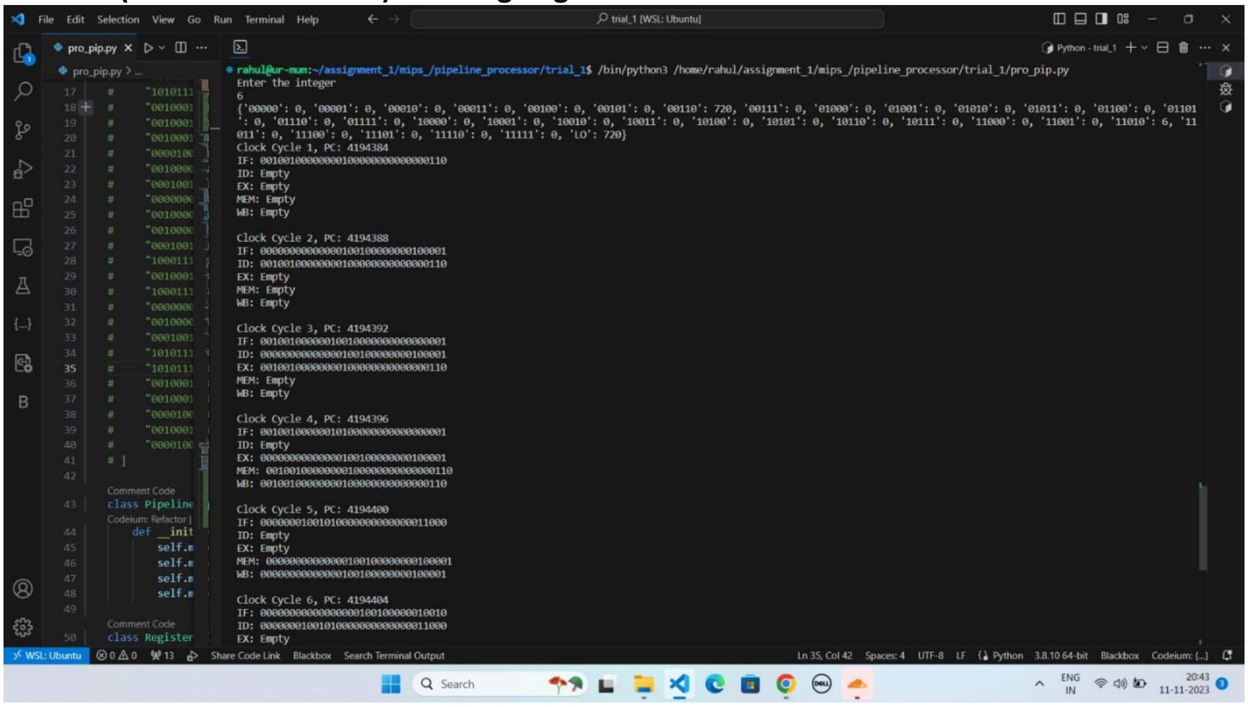
Pipeline Simulation

This method simulates the pipeline stages of a processor.
It runs a loop until all pipeline stages are empty, indicating the end of instruction execution.

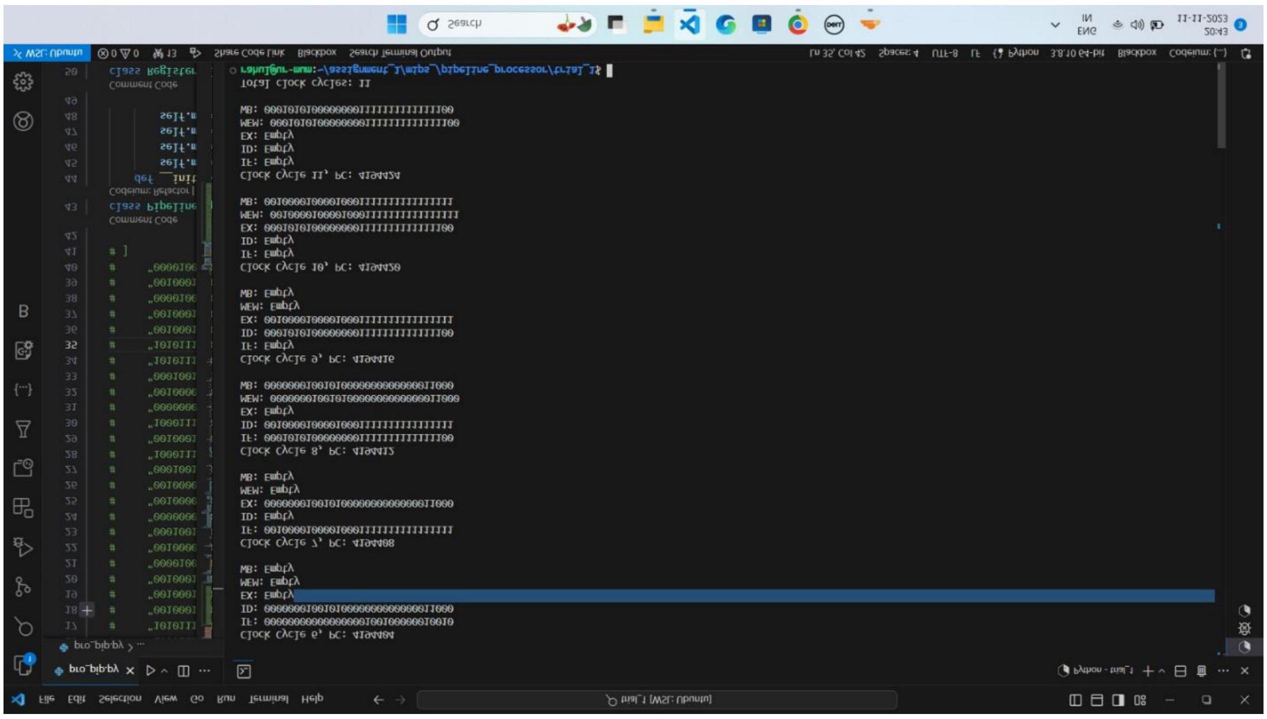
In each cycle, it increments the program counter (PC) and fetches an instruction into the IF stage if available.
It prints the current clock cycle, program counter, and contents of each pipeline stage.
It updates pipeline registers by moving instructions through the pipeline.
It calls the detect_data_hazard method to check for data hazards and inserts stalls (flushes IF stage) if needed.
The simulation loop continues until all pipeline stages are empty.
It prints the total number of clock cycles after the simulation.

Output

Factorial(machinecode1.txt): Printing Regfile

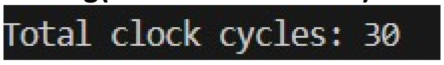

File Edit Selection View Go Run Terminal Help
trial_1 [WSL: Ubuntu]
pro_pip.py
Python - trial_1
WSL: Ubuntu
Share Code Link Blackbox Search Terminal Output
Ln 35, Col 42 Spaces: 4 UTF-8 LF Python 3.8.10 64-bit Blackbox
20:43
11-11-2023

```
rahul@ur-mum:~/assignment_1/mips_/pipeline_processor/trial_1$ /bin/python3 /home/rahul/assignment_1/mips_/pipeline_processor/trial_1/pro_pip.py
Enter the integer
6
{'00000': 0, '00001': 0, '00010': 0, '00011': 0, '00100': 0, '00101': 0, '00110': 720, '00111': 0, '01000': 0, '01001': 0, '01010': 0, '01011': 0, '01100': 0, '01101
': 0, '01110': 0, '01111': 0, '10000': 0, '10001': 0, '10010': 0, '10011': 0, '10100': 0, '10101': 0, '10110': 0, '10111': 0, '11000': 0, '11001': 0, '11010': 6, '11
011': 0, '11100': 0, '11101': 0, '11110': 0, '11111': 0, 'LO': 720}
Clock Cycle 1, PC: 4194384
IF: 00100100000000010000000000000110
ID: Empty
EX: Empty
MEM: Empty
WB: Empty

Clock Cycle 2, PC: 4194388
IF: 00000000000000100100000000100001
ID: 00100100000000010000000000000110
EX: Empty
MEM: Empty
WB: Empty

Clock Cycle 3, PC: 4194392
IF: 00100100000010010000000000000001
ID: 00000000000000100100000000100001
EX: 00100100000000010000000000000110
MEM: Empty
WB: Empty

Clock Cycle 4, PC: 4194396
IF: 00100100000010100000000000000001
ID: Empty
EX: 00000000000000100100000000100001
MEM: 00100100000000010000000000000110
WB: 00100100000000010000000000000110

Clock Cycle 5, PC: 4194400
IF: 00000001001010100000000000011000
ID: Empty
EX: Empty
MEM: 00000000000000100100000000100001
WB: 00000000000000100100000000100001

Clock Cycle 6, PC: 4194404
IF: 00000000000000000100100000010010
ID: 00000001001010100000000000011000
EX: Empty
```

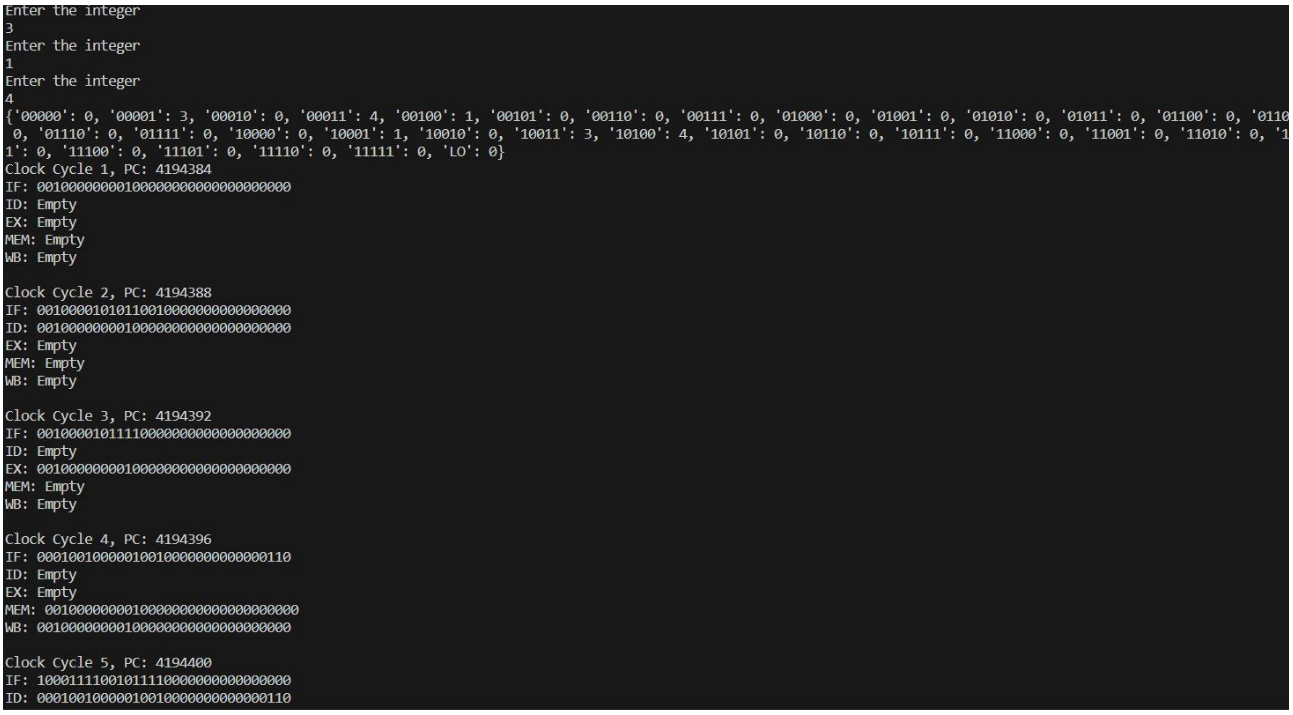


```
Clock Cycle 6, PC: 4194404
IF: 00000000000000000100100000010010
ID: 00000001001010000000000000011000
EX: Empty
MEM: Empty
WB: Empty

Clock Cycle 7, PC: 4194408
IF: 00100001000010001111111111111111
ID: Empty
EX: 00000001001010000000000000011000
MEM: Empty
WB: Empty

Clock Cycle 8, PC: 4194412
IF: 00010101000000001111111111111100
ID: 00100001000010001111111111111111
EX: Empty
MEM: 00000001001010000000000000011000
WB: 00000001001010000000000000011000

Clock Cycle 9, PC: 4194416
IF: Empty
ID: 00010101000000001111111111111100
EX: 00100001000010001111111111111111
MEM: Empty
WB: Empty

Clock Cycle 10, PC: 4194420
IF: Empty
ID: Empty
EX: 00010101000000001111111111111100
MEM: 00100001000010001111111111111111
WB: 00100001000010001111111111111111

Clock Cycle 11, PC: 4194424
IF: Empty
ID: Empty
EX: Empty
MEM: 00010101000000001111111111111100
WB: 00010101000000001111111111111100

Total clock cycles: 11
rahul@ur-mum:~/assignment_1/mips_/pipeline_processor/trial_1$
```

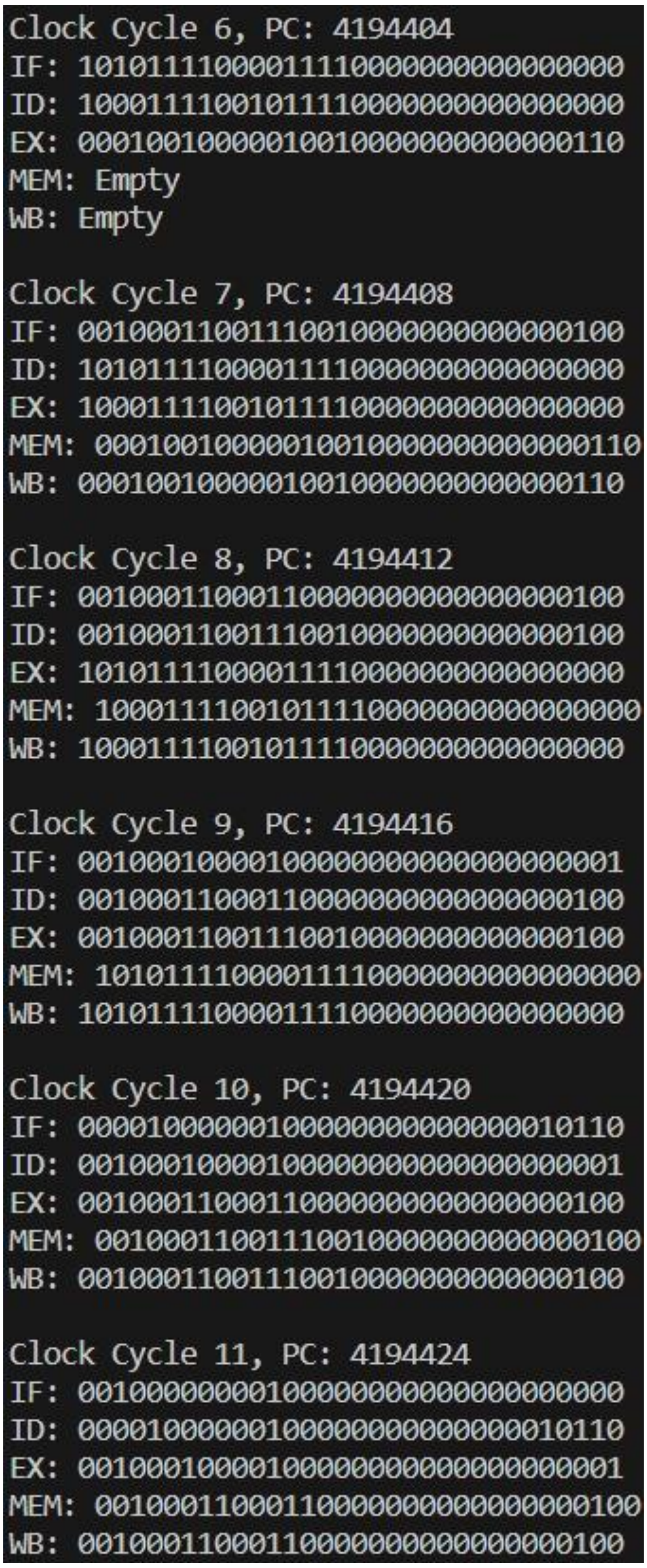
Sorting(machinecode2.txt): Printing



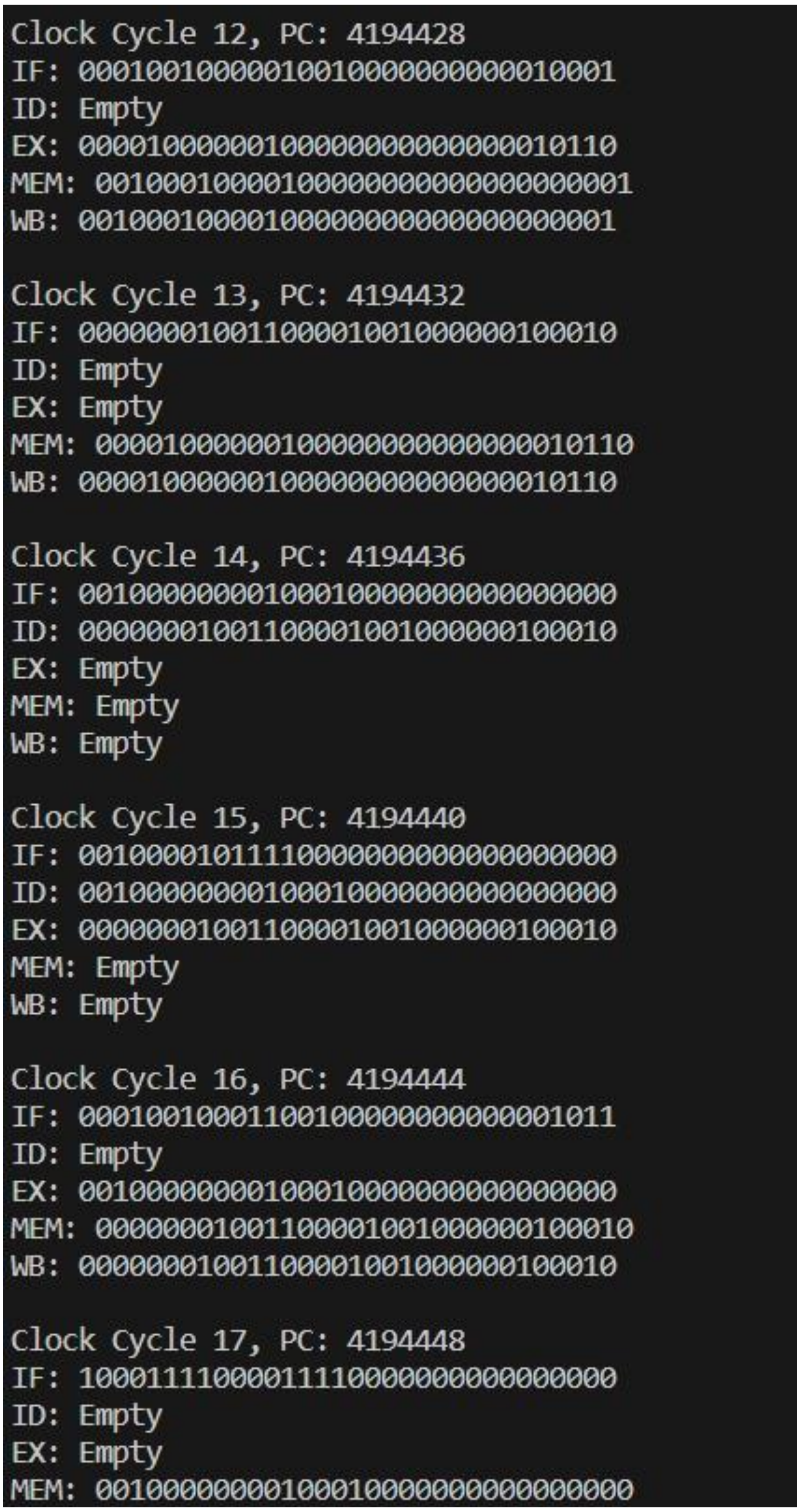
```
Total clock cycles: 30
```



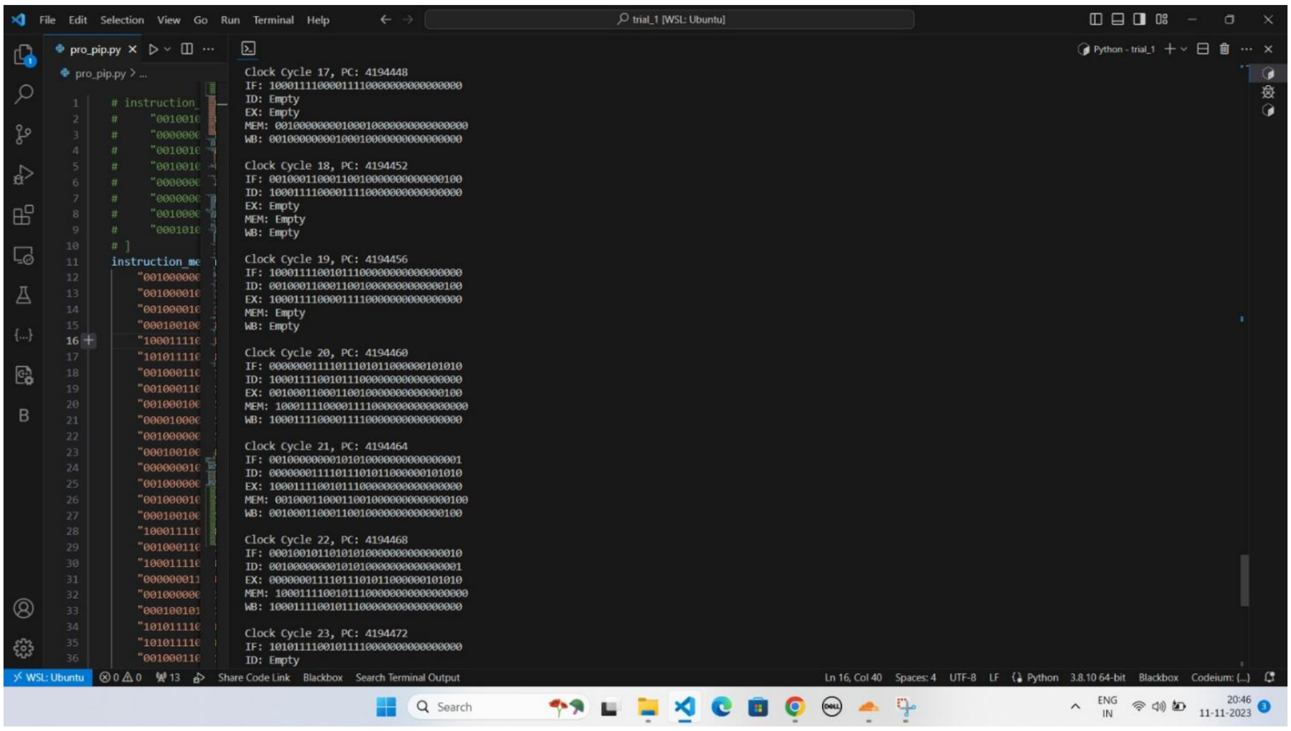
```
Enter the integer
3
Enter the integer
1
Enter the integer
4
{'00000': 0, '00001': 3, '00010': 0, '00011': 4, '00100': 1, '00101': 0, '00110': 0, '00111': 0, '01000': 0, '01001': 0, '01010': 0, '01011': 0, '01100': 0, '0110
 0, '01110': 0, '01111': 0, '10000': 0, '10001': 1, '10010': 0, '10011': 3, '10100': 4, '10101': 0, '10110': 0, '10111': 0, '11000': 0, '11001': 0, '11010': 0, '1
1': 0, '11100': 0, '11101': 0, '11110': 0, '11111': 0, 'LO': 0}
Clock Cycle 1, PC: 4194384
IF: 00100000000100000000000000000000
ID: Empty
EX: Empty
MEM: Empty
WB: Empty

Clock Cycle 2, PC: 4194388
IF: 00100001010110010000000000000000
ID: 00100000000100000000000000000000
EX: Empty
MEM: Empty
WB: Empty

Clock Cycle 3, PC: 4194392
IF: 00100001011110000000000000000000
ID: Empty
EX: 00100000000100000000000000000000
MEM: Empty
WB: Empty

Clock Cycle 4, PC: 4194396
IF: 00010010000010010000000000000110
ID: Empty
EX: Empty
MEM: 001000000001000000000000000000000
WB: 00100000000100000000000000000000

Clock Cycle 5, PC: 4194400
IF: 10001111001011110000000000000000
ID: 00010010000010010000000000000110
```



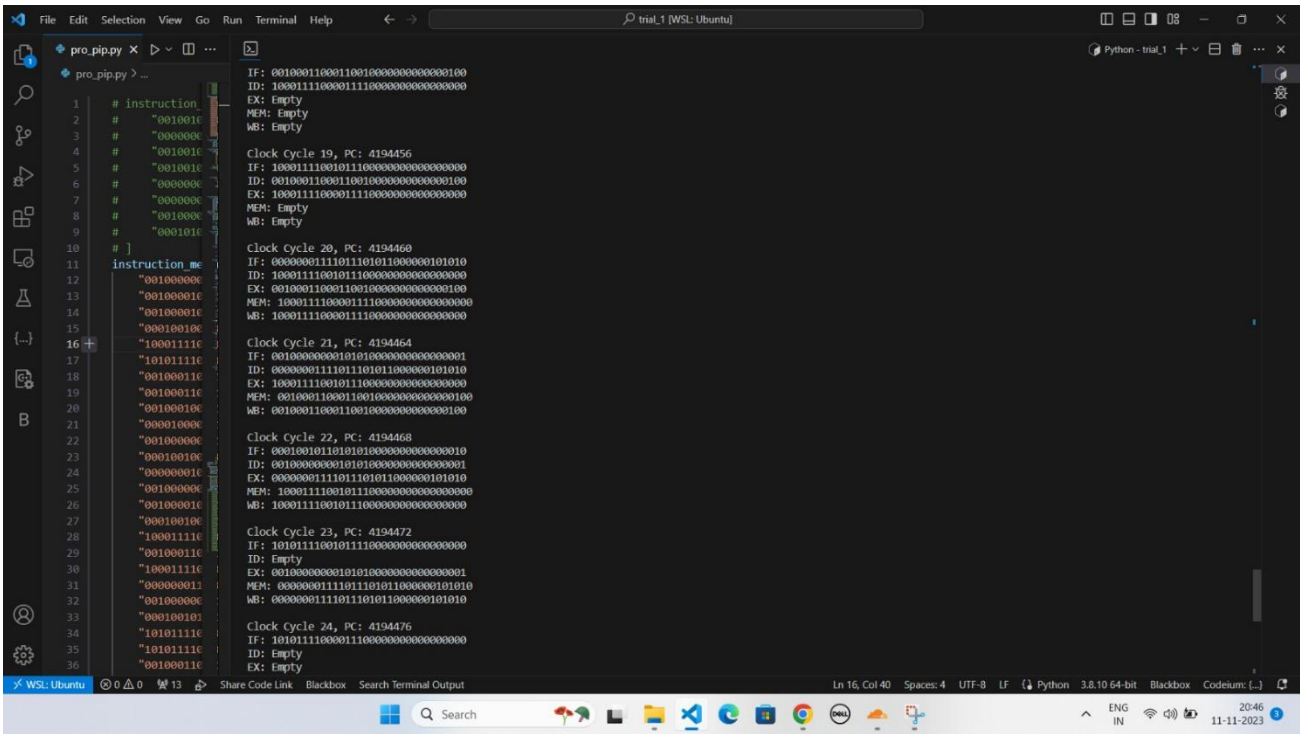
```
Clock Cycle 6, PC: 4194404
IF: 10101111000011110000000000000000
ID: 10001111001011110000000000000000
EX: 00010010000010010000000000000110
MEM: Empty
WB: Empty

Clock Cycle 7, PC: 4194408
IF: 00100011001110010000000000000100
ID: 10101111000011110000000000000000
EX: 10001111001011110000000000000000
MEM: 00010010000010010000000000000110
WB: 00010010000010010000000000000110

Clock Cycle 8, PC: 4194412
IF: 00100011000110000000000000000100
ID: 00100011001110010000000000000100
EX: 10101111000011110000000000000000
MEM: 10001111001011110000000000000000
WB: 10001111001011110000000000000000

Clock Cycle 9, PC: 4194416
IF: 00100010000100000000000000000001
ID: 00100011000110000000000000000100
EX: 00100011001110010000000000000100
MEM: 10101111000011110000000000000000
WB: 10101111000011110000000000000000

Clock Cycle 10, PC: 4194420
IF: 00001000000100000000000000010110
ID: 00100010000100000000000000000001
EX: 00100011000110000000000000000100
MEM: 00100011001110010000000000000100
WB: 00100011001110010000000000000100

Clock Cycle 11, PC: 4194424
IF: 00100000000100000000000000000000
ID: 00001000000100000000000000010110
EX: 00100010000100000000000000000001
MEM: 00100011000110000000000000000100
WB: 00100011000110000000000000000100
```



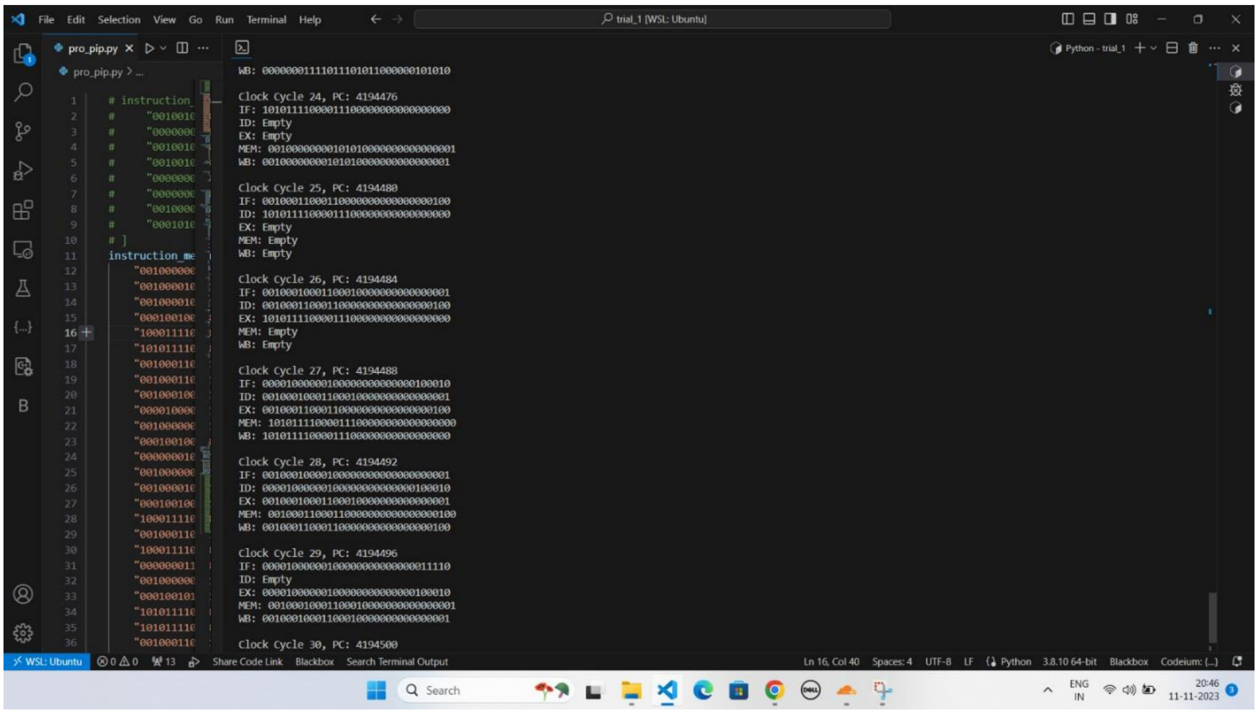
```
Clock Cycle 12, PC: 4194428
IF: 00010010000010010000000000010001
ID: Empty
EX: 00001000000100000000000000010110
MEM: 00100010000100000000000000000001
WB: 00100010000100000000000000000001

Clock Cycle 13, PC: 4194432
IF: 00000001001100001001000000100010
ID: Empty
EX: Empty
MEM: 00001000000100000000000000010110
WB: 00001000000100000000000000010110

Clock Cycle 14, PC: 4194436
IF: 00100000000100010000000000000000
ID: 00000001001100001001000000100010
EX: Empty
MEM: Empty
WB: Empty

Clock Cycle 15, PC: 4194440
IF: 00100001011110000000000000000000
ID: 00100000000100010000000000000000
EX: 00000001001100001001000000100010
MEM: Empty
WB: Empty

Clock Cycle 16, PC: 4194444
IF: 00010010001100100000000000001011
ID: Empty
EX: 00100000000100010000000000000000
MEM: 00000001001100001001000000100010
WB: 00000001001100001001000000100010

Clock Cycle 17, PC: 4194448
IF: 10001111000011110000000000000000
ID: Empty
EX: Empty
MEM: 00100000000100010000000000000000
```



```
Clock Cycle 17, PC: 4194448
IF: 10001111000011110000000000000000
ID: Empty
EX: Empty
MEM: 00100000000100010000000000000000
WB: 00100000000100010000000000000000

Clock Cycle 18, PC: 4194452
IF: 00100011000110010000000000000100
ID: 10001111000011110000000000000000
EX: Empty
MEM: Empty
WB: Empty

Clock Cycle 19, PC: 4194456
IF: 10001111001011110000000000000000
ID: 00100011000110010000000000000100
EX: 10001111000011110000000000000000
MEM: Empty
WB: Empty

Clock Cycle 20, PC: 4194460
IF: 00000001111011101011000000101010
ID: 10001111001011110000000000000000
EX: 00100011000110010000000000000100
MEM: 10001111000011110000000000000000
WB: 10001111000011110000000000000000

Clock Cycle 21, PC: 4194464
IF: 00100000000101010000000000000001
ID: 00000001111011101011000000101010
EX: 10001111001011110000000000000000
MEM: 00100011000110010000000000000100
WB: 00100011000110010000000000000100

Clock Cycle 22, PC: 4194468
IF: 00010010110101010000000000000010
ID: 00100000000101010000000000000001
EX: 00000001111011101011000000101010
MEM: 10001111001011110000000000000000
WB: 10001111001011110000000000000000

Clock Cycle 23, PC: 4194472
IF: 10101111001011110000000000000000
ID: Empty
```



```
IF: 00100011000110010000000000000100
ID: 10001111000011110000000000000000
EX: Empty
MEM: Empty
WB: Empty

Clock Cycle 19, PC: 4194456
IF: 10001111001011110000000000000000
ID: 00100011000110010000000000000100
EX: 10001111000011110000000000000000
MEM: Empty
WB: Empty

Clock Cycle 20, PC: 4194460
IF: 00000001111011101011000000101010
ID: 10001111001011110000000000000000
EX: 00100011000110010000000000000100
MEM: 10001111000011110000000000000000
WB: 10001111000011110000000000000000

Clock Cycle 21, PC: 4194464
IF: 00100000000101010000000000000001
ID: 00000001111011101011000000101010
EX: 10001111001011110000000000000000
MEM: 00100011000110010000000000000100
WB: 00100011000110010000000000000100

Clock Cycle 22, PC: 4194468
IF: 00010010110101010000000000000010
ID: 00100000000101010000000000000001
EX: 00000001111011101011000000101010
MEM: 10001111001011110000000000000000
WB: 10001111001011110000000000000000

Clock Cycle 23, PC: 4194472
IF: 10101111001011110000000000000000
ID: Empty
EX: 00100000000101010000000000000001
MEM: 00000001111011101011000000101010
WB: 00000001111011101011000000101010

Clock Cycle 24, PC: 4194476
IF: 10101111000011110000000000000000
ID: Empty
EX: Empty
```



```
WB: 00000001111011101011000000101010

Clock Cycle 24, PC: 4194476
IF: 10101111000011100000000000000000
ID: Empty
EX: Empty
MEM: 00100000000101010000000000000001
WB: 00100000000101010000000000000001

Clock Cycle 25, PC: 4194480
IF: 00100011000110000000000000000100
ID: 10101111000011100000000000000000
EX: Empty
MEM: Empty
WB: Empty

Clock Cycle 26, PC: 4194484
IF: 00100010001100010000000000000001
ID: 00100011000110000000000000000100
EX: 10101111000011100000000000000000
MEM: Empty
WB: Empty

Clock Cycle 27, PC: 4194488
IF: 00001000000100000000000000100010
ID: 00100010001100010000000000000001
EX: 00100011000110000000000000000100
MEM: 10101111000011100000000000000000
WB: 10101111000011100000000000000000

Clock Cycle 28, PC: 4194492
IF: 00100010000100000000000000000001
ID: 00001000000100000000000000100010
EX: 00100010001100010000000000000001
MEM: 00100011000110000000000000000100
WB: 00100011000110000000000000000100

Clock Cycle 29, PC: 4194496
IF: 00001000000100000000000000011110
ID: Empty
EX: 00001000000100000000000000100010
MEM: 00100010001100010000000000000001
WB: 00100010001100010000000000000001

Clock Cycle 30, PC: 4194500
```

Conclusion

1) With the given codes, we can conclude that the pipeline code runs faster(around 3-4 times), than the non-pipeline code.
2) It can be inferred due to many instructions being executed in a particular clock cycle3)
3) We can compare the results from non-pipeline for example
   - Sorting code cycle becomes from 119 to 30
   - Factorial code cycle becomes from 29 to 11

We were able to understand the significance of using a pipelined processor and how it can reduce latency in the code and make the processes much faster
We were also able to recognize different kinds of hazards and were able to understand how they change the delay in pipelined processor and how to handle them it was a great experience for us and we were able to learn a lot.

Contributors:

- Rahul Mukundhan (imt2022518)
- Siddharth Palod (imt2022002)