1 Introduction

This project is done as part of the Computer Architecture Course under Prof. Nanditha Rao. This assignment has two parts to it. The first part is the Non-Pipelined Processor. In this code we ask the user to put Sorting/Factorial/Fibonacci as an input in the terminal which contains the binary format of the Mips code. As the output it creates a '.txt' file which contains the output of the required code in the register memory or the data memory. We have implemented all 5 instruction phases: IF, ID, EX, MEM, WB.

The second part of the assignment was to implement a Pipelined Processor with the resolution of the Data Hazards and Control Hazards. Here again we are taking the Binary format of the Mips code that we get from the assembler as the machine code, as the input. We are first detecting the Data -Hazards with the help of a function that we have created. With the help of this function, we are solving the data hazards that have occurred in the Mips machine code. There is no need for creating another function in order to calculate Control Hazards as Beq and Bne instructions only deal with the Control Hazards. We use Python Programming Language to implement both the Processors. The code is well commented and modularized for easy reading and ease of use.

2 Code Explanation

Both the processors have one python code file of itself and the text files which contain the machine code of the Factorial, Fibonacci and Sorting codes.

Part - A

Here we are declaring all the dictionaries and initialising wherever required for the register memory and the data memory. We are also initialising the function dictionary for the R format type of instructions.

```
import time
instr_mem = {}
data_mem = {}
for i in range(101):
    data_mem[i] = 0
data_mem[0] = 4
data_mem[4] = 1
data_mem[8] = 7
data_mem[12] = 10
data_mem[16] = 6
reg_mem = {
    "00000": 0,"00001": 0,"00010": 0,"00011": 0,"00100": 0,"00101": 0,"00110": 0,"00111": 0,"01000": 0,"01001": 5,
    "01010": 0,"01011": 24,"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": 0,"11011": 0,"11100": 0,"11101": 0,
    "11110": 0,"11111": 0,
}
function_Mem = {
    "100000": "add","100100": "and","1001010": "or","101010": "slt","100010": "subtract","100001" : "addu",
    "000000" : "sll"
}
opcode = {
    "000000": "R","001000": "addi","000100": "beq","000010": "j","000011": "jal","100011": "load","101011": "store",
    "000101": "bne",
}

```

Here we are in the Decode Phase and decoding the instructions and sending the required Control Signals.

```

def decode(instruction, program_Counter):
    # R format
    if instruction[0:6] == "000000":
        regdst = True
        Branch = False
        Mem_read = False
        Mem_to_reg = False
        Mem_wri = False
        RegWr = True
        jump = False
        ALU_src = False
        jump_And_Link = True
        if regdst:
            reg_Dest = instruction[16:21]
        else:
            reg_Dest = instruction[11:16]
        decode_Dict = {
            "opcode": opcode[instruction[0:6]],
            "reg_src_1": reg_mem[instruction[6:11]],
            "reg_src_2": reg_mem[instruction[11:16]],
            "reg_dest": reg_Dest,
            "function": function_Mem[instruction[26:32]],
            "regdst": regdst,
            "Branch": Branch,
            "Mem_read": Mem_read,
            "Mem_to_reg": Mem_to_reg,
            "Mem_wri": Mem_wri,
            "RegWr": RegWr,
            "jump": jump,
            "ALU_src": ALU_src,
            "jump_And_Link" : jump_And_Link,
            "shamt" : instruction[21:26]
        }
        return decode_Dict
    # addi
    if instruction[0:6] == "001000":
```

Here we are in the Execute phase, where we are evaluating the operation as mentioned in the instruction format. We have also implemented the required Muxes wherever required with the help of if-else loops and passing the result of the operation performed to the next phase.

```
274
275  # -------------EXECUTE PHASE-----------------
276
277  def execute(decode_Dict, program_Counter):
278      if decode_Dict["ALU_src"]:
279          if(decode_Dict['immediate'][0] == '0'):
280              ALU_Mux = int(decode_Dict['immediate'][17:32], 2)
281          else:
282              ALU_Mux = (1 << (len(decode_Dict['immediate'][16:32]) - 1)) - int(decode_Dict['immediate'][16:32], 2)
283      elif(decode_Dict["ALU_src"] == False and decode_Dict['jump'] == False):
284          ALU_Mux = decode_Dict["reg_src_2"]
285      # R
286      if decode_Dict["opcode"] == "R":
287          reg_src_1_Value = decode_Dict["reg_src_1"]
288          reg_src_2_Value = ALU_Mux
289          # add
290          if decode_Dict["function"] == "add" or decode_Dict["function"] == "addu":
291              reg_dest_Value = reg_src_1_Value + reg_src_2_Value
292              execute_Dict = {
293                  "program_Counter" : None,
294                  "reg_dest_Value": reg_dest_Value,
295                  "reg_dest_Address": decode_Dict["reg_dest"],
296                  "regdst": decode_Dict["regdst"],
297                  "Branch": decode_Dict["Branch"],
298                  "Mem_read": decode_Dict["Mem_read"],
299                  "Mem_to_reg": decode_Dict["Mem_to_reg"],
300                  "Mem_wri": decode_Dict["Mem_wri"],
301                  "RegWr": decode_Dict["RegWr"],
302                  "jump": decode_Dict["jump"],
303                  "ALU_src": decode_Dict["ALU_src"],
304                  "jump_And_Link" : decode_Dict["jump_And_Link"]
305              }
306              return execute_Dict
307          # subtract
308          if decode_Dict["function"] == "subtract":
309              reg_dest_Value = reg_src_1_Value - reg_src_2_Value
```

Here we are in the Memory Phase.

```
non_pipelined.py X  pipelined.py
non_pipelined.py > memory
543                 "jump_And_Link" : decode_Dict["jump_And_Link"]
544             }
545             return execute_Dict
546
547
548  # -------------MEMORY PHASE-----------------
549
550
551  def memory(execute_Dict, program_Counter):
552      if execute_Dict['Mem_wri'] == True and execute_Dict['Mem_read'] == False: # STORE
553          data_mem[execute_Dict["dest_Address"]] = execute_Dict["value_To_Be_Stored"]
554          return execute_Dict
555      if execute_Dict['Mem_wri'] == False and execute_Dict['Mem_read'] == True:  # LOAD
556          memory_Dict = {
557              "value_To_Be_Stored" : data_mem[execute_Dict["value_To_Be_Stored"]],
558              "reg_dest_Address": execute_Dict["reg_dest_Address"],
559              "Mem_to_reg" : execute_Dict['Mem_to_reg'],
560              "Mem_wri" : execute_Dict['Mem_wri']
561          }
562          return memory_Dict
563      return execute_Dict
564
565
```

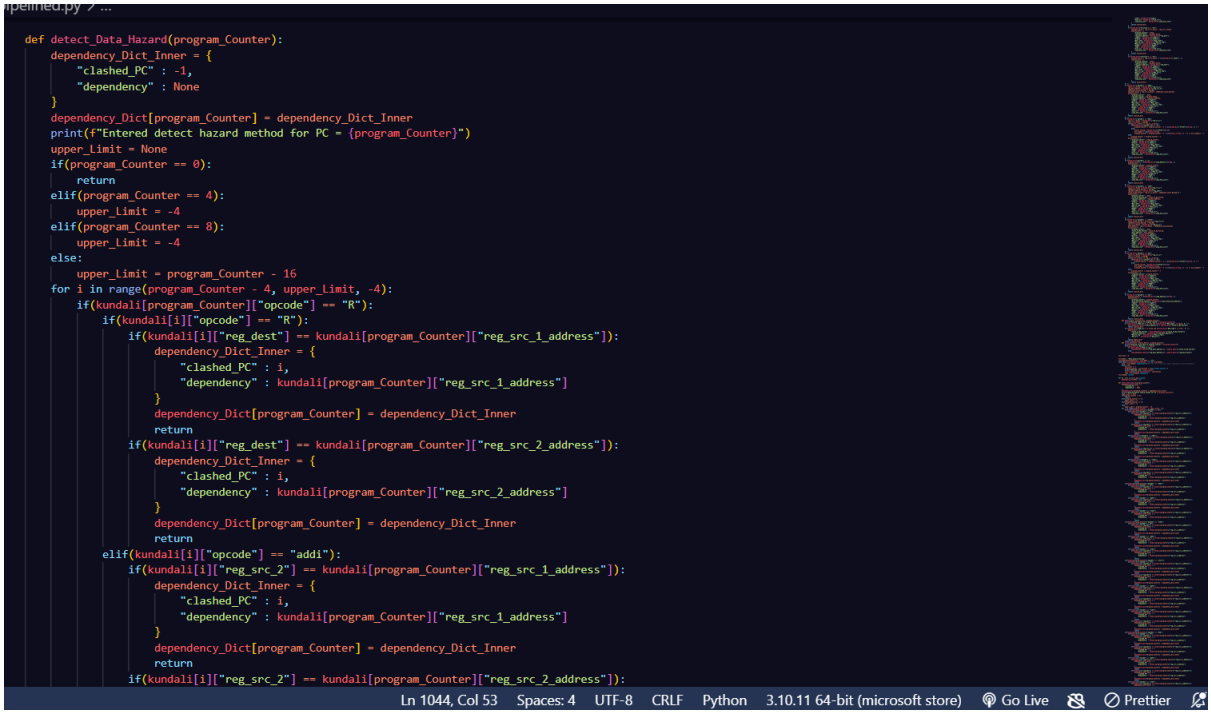
Here we are in the Writeback Phase and we are reading the machine code from the mentioned text file using the File format syntax. In this block of code we are also keeping a track of the number of clock cycles and printing it in the end to verify the result of the Non-Pipelined processor.

```
# ----------------------WRITE BACK PHASE----------------------


def write_Back(memory_Return, program_Counter):
    if(memory_Return['Mem_wri'] == False):
        if memory_Return['Mem_to_reg']:
            reg_mem[memory_Return["reg_dest_Address"]] = memory_Return["value_To_Be_Stored"]
        else:
            reg_mem[memory_Return["reg_dest_Address"]] = memory_Return["reg_dest_Value"]

num_Lines = 0
clock_Cycles = 0
file_Name = input("Write the program name:")
file_Name_With_Extension = file_Name + ".txt"
with open(file_Name_With_Extension, "r") as read_Binary:
    line = read_Binary.readline() # Read the first line and remove leading/trailing whitespaces
    while line:
        num_Lines += 1
        program_Counter, instruction = line.strip().split(" ")
        instr_mem[int(program_Counter)] = instruction
        line = read_Binary.readline()
read_Binary.close()
```

Part – B

The code remains the same as Non-Pipelined processor till the Writeback phase. After it, we have added the required code in order to convert it to a Pipelined Processor with the required hazard solving.

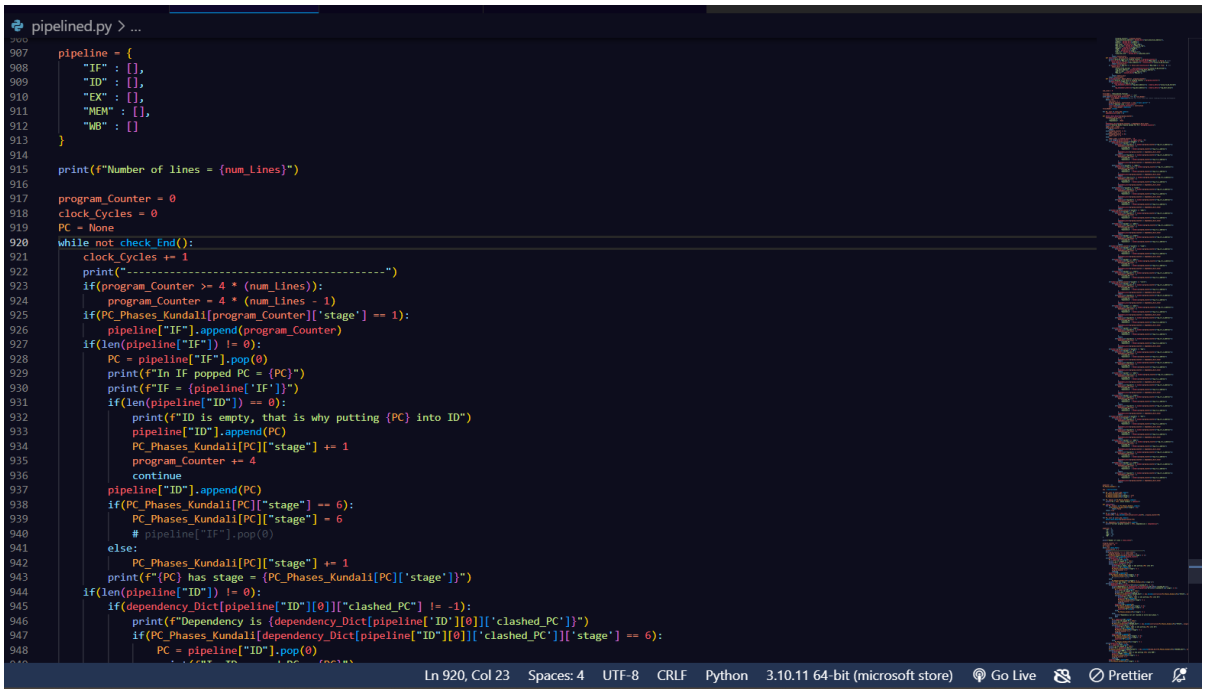
Here, this code finds the dependencies for the Data Hazard and takes the PC count of the dependent instructions into consideration and sends it to the next function for solving the same.



```
def detect_Data_Hazard(program_Counter):
    dependency_Dict_Inner = {
        "clashed_PC" : -1,
        "dependency" : None
    }
    dependency_Dict[program_Counter] = dependency_Dict_Inner
    print(f"Entered detect hazard method for PC = {program_Counter}")
    upper_Limit = None
    if(program_Counter == 0):
        return
    elif(program_Counter == 4):
        upper_Limit = -4
    elif(program_Counter == 8):
        upper_Limit = -4
    else:
        upper_Limit = program_Counter - 16
    for i in range(program_Counter - 4, upper_Limit, -4):
        if(kundali[program_Counter]["opcode"] == "R"):
            if(kundali[i]["opcode"] == "R"):
                if(kundali[i]["reg_dest"] == kundali[program_Counter]["reg_src_1_address"]):
                    dependency_Dict_Inner = {
                        "clashed_PC" : i,
                        "dependency" : kundali[program_Counter]["reg_src_1_address"]
                    }
                    dependency_Dict[program_Counter] = dependency_Dict_Inner
                    return
                if(kundali[i]["reg_dest"] == kundali[program_Counter]["reg_src_2_address"]):
                    dependency_Dict_Inner = {
                        "clashed_PC" : i,
                        "dependency" : kundali[program_Counter]["reg_src_2_address"]
                    }
                    dependency_Dict[program_Counter] = dependency_Dict_Inner
                    return
            elif(kundali[i]["opcode"] == "addi"):
                if(kundali[i]["reg_src_2"] == kundali[program_Counter]["reg_src_1_address"]):
                    dependency_Dict_Inner = {
                        "clashed_PC" : i,
                        "dependency" : kundali[program_Counter]["reg_src_1_address"]
                    }
                    dependency_Dict[program_Counter] = dependency_Dict_Inner
                    return
                if(kundali[i]["reg_src_2"] == kundali[program_Counter]["reg_src_2_address"]):
```

Ln 1044, Col 53 Spaces: 4 UTF-8 CRLF Python 3.10.11 64-bit (microsoft store) Go Live Prettier

Here this code is responsible for the Pipelining of the entire processor. It makes sure that the 5 stages IF, ID, EX, MEM, WB are following the Pipelined form. It also solves the Data Hazards and Control Hazards on the way in order to ensure that the output is matching with the Non-Pipelined processor.



```
pipelined.py > ...
    pipeline = {
        "IF" : [],
        "ID" : [],
        "EX" : [],
        "MEM" : [],
        "WB" : []
    }

    print(f"Number of lines = {num_Lines}")

    program_Counter = 0
    clock_Cycles = 0
    PC = None
    while not check_End():
        clock_Cycles += 1
        print("----------------------------------------")
        if(program_Counter >= 4 * (num_Lines)):
            program_Counter = 4 * (num_Lines - 1)
        if(PC_Phases_Kundali[program_Counter]['stage'] == 1):
            pipeline["IF"].append(program_Counter)
        if(len(pipeline["IF"]) != 0):
            PC = pipeline["IF"].pop(0)
            print(f"In IF popped PC = {PC}")
            print(f"IF = {pipeline['IF']}")
            if(len(pipeline["ID"]) == 0):
                print(f"ID is empty, that is why putting {PC} into ID")
                pipeline["ID"].append(PC)
                PC_Phases_Kundali[PC]["stage"] += 1
                program_Counter += 4
                continue
            pipeline["ID"].append(PC)
            if(PC_Phases_Kundali[PC]["stage"] == 6):
                PC_Phases_Kundali[PC]["stage"] = 6
                # pipeline["IF"].pop(0)
            else:
                PC_Phases_Kundali[PC]["stage"] += 1
            print(f"{PC} has stage = {PC_Phases_Kundali[PC]['stage']}")
        if(len(pipeline["ID"]) != 0):
            if(dependency_Dict[pipeline["ID"][0]]["clashed_PC"] != -1):
                print(f"Dependency is {dependency_Dict[pipeline['ID'][0]]['clashed_PC']}")
                if(PC_Phases_Kundali[dependency_Dict[pipeline["ID"][0]]['clashed_PC']]['stage'] == 6):
                    PC = pipeline["ID"].pop(0)
```

Ln 920, Col 23 Spaces: 4 UTF-8 CRLF Python 3.10.11 64-bit (microsoft store) Go Live Prettier

3 RESULT

Sorting Code

sorting_Output.txt

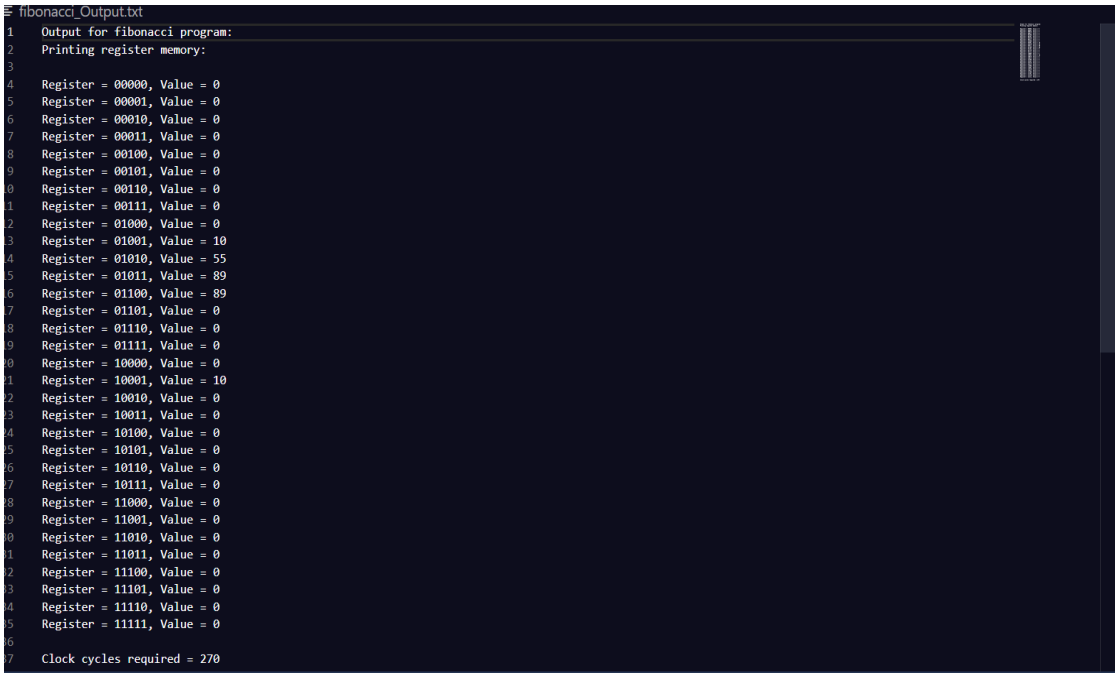
```
Output for sorting program:
Printing data memory:
Address = 0, value = 4
Address = 1, value = 0
Address = 2, value = 0
Address = 3, value = 0
Address = 4, value = 1
Address = 5, value = 0
Address = 6, value = 0
Address = 7, value = 0
Address = 8, value = 7
Address = 9, value = 0
Address = 10, value = 0
Address = 11, value = 0
Address = 12, value = 10
Address = 13, value = 0
Address = 14, value = 0
Address = 15, value = 0
Address = 16, value = 6
Address = 17, value = 0
Address = 18, value = 0
Address = 19, value = 0
Address = 20, value = 0
Address = 21, value = 0
Address = 22, value = 0
Address = 23, value = 0
Address = 24, value = 1
Address = 25, value = 0
Address = 26, value = 0
Address = 27, value = 0
Address = 28, value = 4
Address = 29, value = 0
Address = 30, value = 0
Address = 31, value = 0
Address = 32, value = 6
Address = 33, value = 0
Address = 34, value = 0
Address = 35, value = 0
Address = 36, value = 7
Address = 37, value = 0
Address = 38, value = 0
Address = 39, value = 0
Address = 40, value = 10
Address = 41, value = 0
Address = 42, value = 0
Address = 43, value = 0
Address = 44, value = 0
Address = 45, value = 0
```

Factorial Codes

factorial_Output.txt

```
Output for factorial program:
Printing register memory:

Register = 00000, Value = 0
Register = 00001, Value = 0
Register = 00010, Value = 0
Register = 00011, Value = 0
Register = 00100, Value = 0
Register = 00101, Value = 0
Register = 00110, Value = 0
Register = 00111, Value = 0
Register = 01000, Value = 0
Register = 01001, Value = 120
Register = 01010, Value = 0
Register = 01011, Value = 120
Register = 01100, Value = 0
Register = 01101, Value = 0
Register = 01110, Value = 0
Register = 01111, Value = 0
Register = 10000, Value = 0
Register = 10001, Value = 120
Register = 10010, Value = 0
Register = 10011, Value = 0
Register = 10100, Value = 0
Register = 10101, Value = 0
Register = 10110, Value = 0
Register = 10111, Value = 0
Register = 11000, Value = 0
Register = 11001, Value = 0
Register = 11010, Value = 0
Register = 11011, Value = 0
Register = 11100, Value = 0
Register = 11101, Value = 0
Register = 11110, Value = 0
Register = 11111, Value = 0

Clock cycles required = 360
```

Fibonacci Codes



4 Done BY:

- **Valmik Belgaonkar (IMT2022020)**
- **Vedant Mangrulkar (IMT2022519)**