ASSIGNMENT 2
COMPUTER ARCHITECTURE
IMT2022021 AND IMT2022098
RUTUL and HEMANG

Introduction->

Here in this question we have made our own processors of the MIPS .asm code. We have done factorial and sorting in our code. Using python we have made our processor. It works in both non pipeline and pipeline type. We chose taking input in the form of Hexadecimal format and also doing operations on it

What we Observed->

No. of clock cycles in Non-pipeline Factorial = 85
No. of clock cycles in Pipeline Factorial = 21
No. of clock cycles in Non-pipeline Sorting = 1670
No. of clock cycles in Pipeline Sorting = 338

What can we Infer from it->

1. The ratio of pipeline/non-pipeline clock cycles is approximately 0.22-0.25
2. Pipeline approach turns out to be better as in this case the number of clock cycles are effectively reduced.
3. Pipeline allows running of multiple instructions simultaneously which gives a better efficiency (also note that we considered forwarding and flushing duly)

Description->

Q-1 Non-Pipeline Factorial = Here it calculates the values of the factorial using the code of the factorial asm file
Here we have provided the given screenshot

```
.text
main:
    addi $t0,$zero 5
    addi $t1,$zero 1
    addi $t2,$zero 1
    factorial_loop:
        mul $t1, $t1, $t0
        addi $t0, $t0, -1
        bne $t0, $t2, factorial_loop
        addi $v0,$zero 1       # Load the service code 1 for printing an integer
        addi $a0,$t1, 0      # Load the integer value you want to print into $a0
        #syscall
```

```
5
IF Stage
Instruction address: 0x00400010
Instruction Binary Code: 00100001000010001111111111111111
CLOCK: 21
-----X-----
ID phase
I type instruction
opcode : 001000
rs = 01000 rt = 01000 immediate = 65535
Instruction = addi $t0,$t0,65535
CLOCK : 22
-----X-----
EXE Stage
I type instruction
addi
CLOCK : 23
-----X-----
MEM Stage
Nothing is being stored
00100001000010001111111111111111
CLOCK : 24
-----X-----
Write Back Stage
$t0 : 4
CLOCK : 25
-----X-----
```

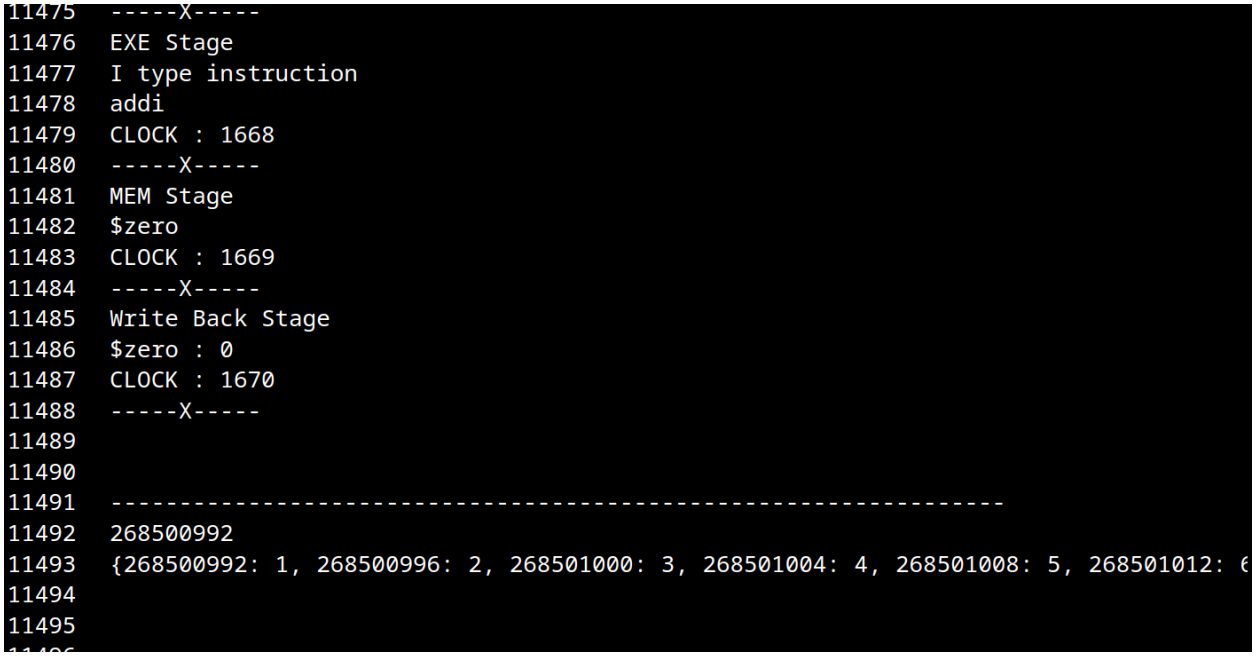
```
120



{'$zero': 0, '$at': 0, '$v0': 1, '$v1': 0, '$a0': 120, '$a1': 0, '$a2': 0,
```

This is how we have shown the output

Q-1 Non-Pipeline Sorting = Here using the standard beq conditions and some loops it sorts the given numbers and prints the corresponding final dictionary to it.

```
.data
ar:.word 9 15 6 5 4 3 2 1
.text

        li $t1,0x10010000
        addi $t8 $t8,7
        addi $t9,$t9, 0
        addi $t5,$t5,0#i
        addi $s5,$s5,-1
for:
        beq $t5,$t8,done
        sub $t4,$t8,$t5#n-i-1
        addi $t6,$zero,0#j
        addi $t3,$t1,0 #&(ar[i])
        addi $t5,$t5,1 #i++
        beq $zero, $zero, for2

for2:
        beq $t6,$t4,for
        lw $s1,0($t3)
        addi $t3, $t3,4
        lw $s2, 0($t3)
        addi $t6,$t6,1

        blt $s1, $s2, for2
        sw $s1,0($t3)
        sw $s2,-4($t3)
        beq $zero, $zero, for2

done:
        addi $t4,$zero,0
```

```
4817  IF Stage
4818  Instruction address: 0x00400040
4819  Instruction Binary Code: 00100001110011100000000000000001
4820  CLOCK: 701
4821  -----X-----
4822  ID phase
4823  I type instruction
4824  opcode : 001000
4825  rs = 01110 rt = 01110 immediate = 1
4826  Instruction = addi $t6,$t6,1
4827  CLOCK : 702
4828  -----X-----
4829  EXE Stage
4830  I type instruction
4831  addi
4832  CLOCK : 703
4833  -----X-----
4834  MEM Stage
4835  $t6
4836  CLOCK : 704
4837  -----X-----
4838  Write Back Stage
4839  $t6 : 6
4840  CLOCK : 705
4841  -----X-----
4842
```



This is how the final corresponding output is coming when we sort the given instructions.

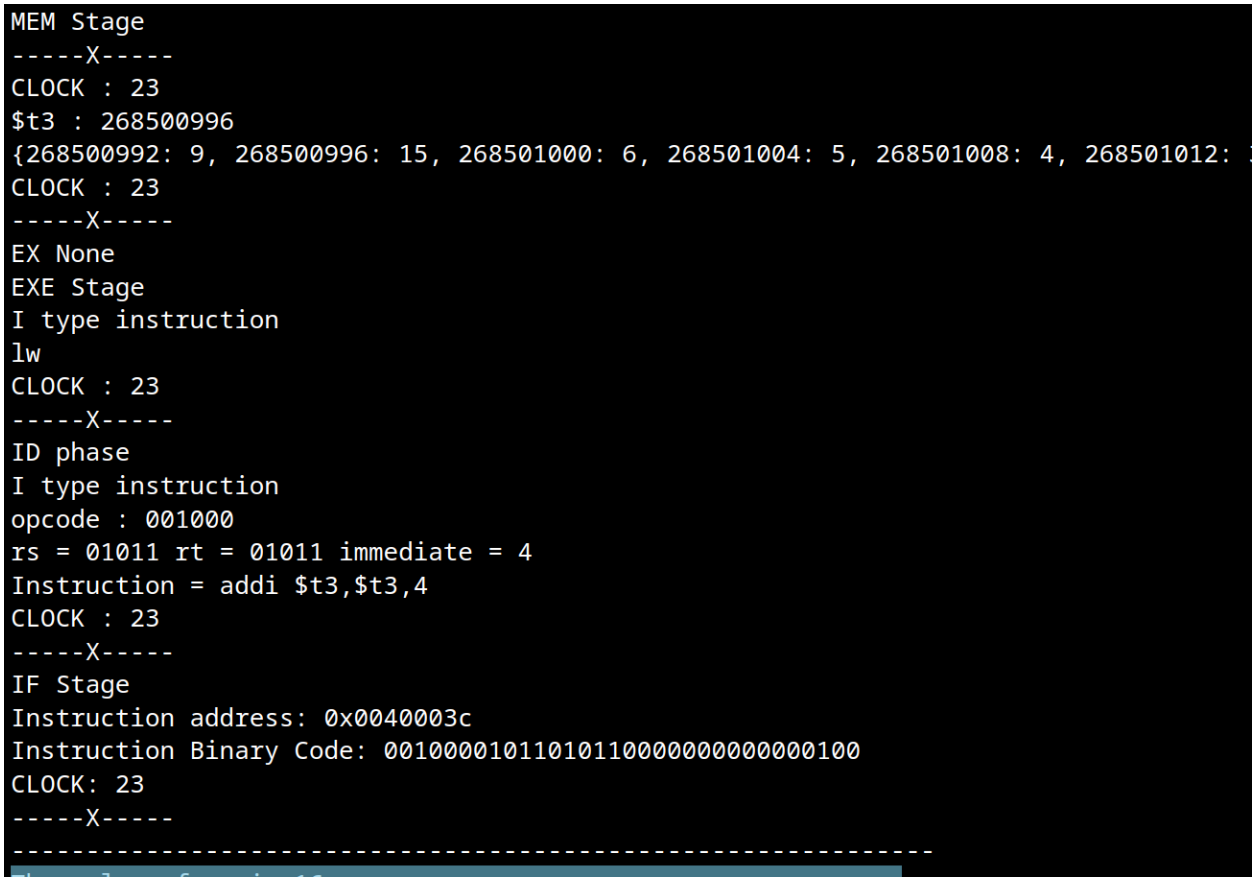
Q-2 Pipeline Factorial = Here using the concept of classes and OOPS we have done the cases of forwarding and how it can multiple instructions simultaneously at a given point of time.

```
60
------------------------------------------------------------------
The value of pc is 5
MEM Stage
-----X-----
CLOCK : 17
EX None
EXE Stage
mul
CLOCK : 17
-----X-----
ID phase
I type instruction
opcode : 001000
rs = 01000 rt = 01000 immediate = 65535
Instruction = addi $t0,$t0,65535
CLOCK : 17
-----X-----
IF Stage
Instruction address: 0x00400014
Instruction Binary Code: 00100001000010001111111111111111
CLOCK: 17
-----X-----
```

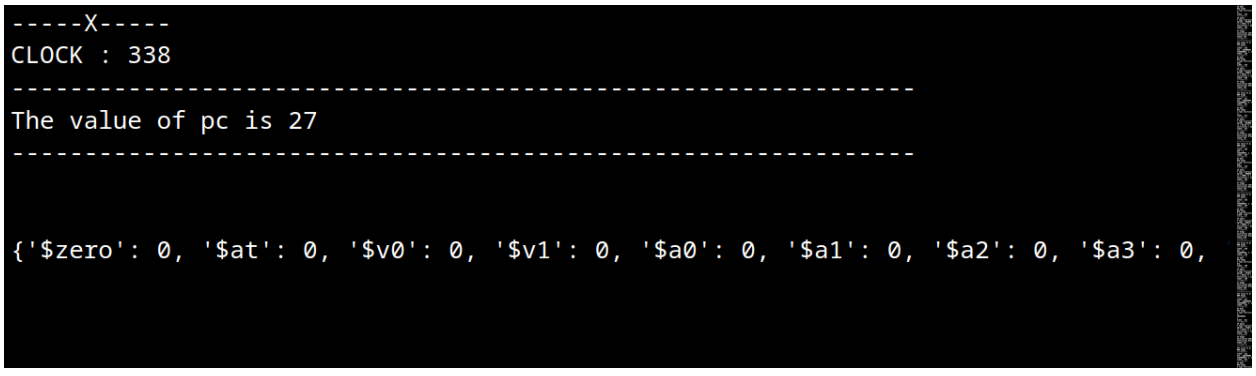
```
120
------------------------------------------------------------
The value of pc is 10



120
------------------------------------------------------------


{'$zero': 0, '$at': 0, '$v0': 0, '$v1': 0, '$a0': 0, '$a1': 0, '$a2': 0, '$a3': 0, '
```

This is how the output is coming and is being showed.

Q-2 Pipeline Sorting = Here also we have mainly used the concept of classes and here multiple instructions are being simultaneously handled.



Here we can see multiple instructions being run during one fetch



Final Output

Problems Faced->

The major Problem in Non-Pipeline factorial and sorting was that the final answer was coming, however the values were not being updated to the given registers. Updating the values individually to the dictionary was initially tricky but was managed efficiently…

The major problem faced in Pipeline factorial and sorting was the handling of the multiple instructions simultaneously. Multiple loops had to be run and the values needed to be continuously at an updating this however was efficiently managed by classes

How To Run->

Select the Option and the txt file containing the output will be shown. ALL OF OUR OUTPUT is printed in there.

Snippets From The Code->

```
    rt = bin_instruction[11:16]
    rd = bin_instruction[16:21]
    shamt = int(bin_instruction[21:26], 2)
    funct = bin_instruction[26:32]
    print(f'opcode : {bin_instruction[:6]}',file=answer_file_output)
    print(f"rs = {mips_registers[rs]} rt = {mips_registers[rt]} rd = {mips_regist
    print(f"Instruction = {opcode_of_commands[funct]} {mips_registers[rd]},{mips_

elif bin_instruction[:6] == "000010":
    print("J type instruction",file=answer_file_output)
    print(f'opcode : {bin_instruction[:6]}',file=answer_file_output)
    address = int(bin_instruction[6:32], 2)
    print(f"address = {address}",file=answer_file_output)
    # int initial = 0x40000000
    print(f"Instruction = {address}",file=answer_file_output)

else:
    print("I type instruction",file=answer_file_output)
    opcode = bin_instruction[:6]
    print(f'opcode : {bin_instruction[:6]}',file=answer_file_output)
    rs = bin_instruction[6:11]
    rt = bin_instruction[11:16]
    #print(rs,rt,end=" ")
    immediate = int(bin_instruction[16:32],2)
    print(f"rs = {rs} rt = {rt} immediate = {immediate}",file=answer_file_output
    print(f"Instruction = {opcode_of_commands[opcode]} {mips_registers[rt]},{mips
```

```
def exe_stage(bin_instruction, mips_registers, mips_registers_values, opcode_of_comma
    print('EXE Stage',file=answer_file_output)
    result = 0
    answer = 0
    global pc
    stringtocheck = format(int(bin_instruction,2),"08x")

#HERE WE ARE STORING THE ANSWERS AND UPDATING THE GIVEN RESULTS
#APART FROM THAT HERE WE ARE ALSO DOING THE NECESSARY VALUE UPDATAL OF THE GIVEN REG

    # if(stringtocheck=="0x3c011001"):
    #     pc+=2
    #     rs = format(int(bin_instruction[6:11], 2), "05b")
    #     rt = format(int(bin_instruction[11:16], 2), "05b")
    #     immediate = int(bin_instruction[16:32], 2)
    #     mips_registers_values[mips_registers[rt]] = mips_registers_values[mips_reg


    if (bin_instruction[:6] == "011100" and RegWrite==1 and RegDst==1):
        print("mul",file=answer_file_output)
        rs = bin_instruction[6:11]
        rt = bin_instruction[11:16]
```

```
    if immediate == 65535:
        immediate = -1
    if immediate == 65533:
        immediate = -3
    if immediate == 65532:
        immediate = -4
    # print(f"rs = {mips_registers[rs]} rt = {mips_registers[rt]} immediate = {ir
    nameOfOpcode = opcode_of_commands[opcode]
    print(nameOfOpcode,file=answer_file_output)
    if nameOfOpcode == 'lw':
        mips_registers_values[mips_registers[rt]] = instruction_dictionary_sortir
        # print(f"{mips_registers[rt]}: {mips_registers_values[mips_registers[rt]
    if nameOfOpcode == 'lui':
        mips_registers_values[mips_registers[rt]] = immediate << 16
        # print(f"{mips_registers[rt]}: {mips_registers_values[mips_registers[rt]
    if nameOfOpcode == 'ori':
        mips_registers_values[mips_registers[rt]] = mips_registers_values[mips_re
        # print(f"{mips_registers[rt]}: {mips_registers_values[mips_registers[rt]
    if nameOfOpcode == 'bne':
        if mips_registers_values[mips_registers[rs]] != mips_registers_values[mip
            string_immediate = bin_instruction[16:32]
            inverted_string = ''.join(['1' if bit == '0' else '0' for bit in stri
            #WE REVERSED THE DICTIONARY OVER HERE
```

```
while(pc<len(instruction_memory_sorting) and a==2):
    # if_stage(instruction_memory_sorting,clk1)
    if(a==2):
        clk+=1
        if_stage(instruction_memory_sorting,clk)

    # print("IF Stage")
    # instruction = instruction_memory[pc]
    # inst = format((0x40000000+(pc*4)),"08x")
    # print("Instruction address: "+"0x"+inst)

    # print("Instruction Binary Code: "+format(instruction,"032b"))
    # clk+=1

    # print(f"CLOCK : {clk}")
    # print(pc)
        clk+=1
        id_phase(instruction_memory_sorting, mips_registers, opcode_of_commands,clk)
    # print(pc)
        clk+=1
        exe_stage(bin_instruction, mips_registers, mips_registers_values, opcode_of_
        if(a==1):
            clk+=1
            print("MEM Stage",file=answer_file_output)
            print("Nothing is being stored",file=answer_file_output)
```

This is from Non-Pipeline

```
0x21ce0001,
0x0232082a,
0x1420fff9,
0xad710000,
0xad72fffc,
0x1000fff6,
0x200c0000,

]

#THE ALLOCATED VALUES FOR THE DATA OF DICTIONARY SORTING

instruction_dictionary_sorting={
0x10010000 : 0x00000009,
0x10010004 : 0x0000000f,
0x10010008 : 0x00000006,
0x1001000C : 0x00000005,
0x10010010 : 0x00000004,
0x10010014 : 0x00000003,
0x10010018 : 0x00000002,
0x1001001C : 0x00000001,
}

# mips_registers_values1 = {
#     "$zero": 0,
#     "$at": 0
```

```
class Pipeline_instructions_used:
    def __init__(self) -> None:
        self.list=[if_stage_instruction,id_stage_instruction,ex_stage_instruction,mer
        self.values_initialsed=[False, False, False, False, False]
        self.instructionsGet=None
        self.i=0
        #commparing the given values of list

        #print(self.list)
        #for i in range(len(list)):
            #print(list[i])

        self.clock=clk
        self.pc=pc
        #print(self.pc)




    def run(self,pc=None):
        if self.i==0:
            self.instructionsGet=self.list[self.i](pc)
            #this is how we used the run instructionds

        else:
            self.list[self.i](self.instructionsGet)
```

```
class pipelineprocessor:
    # pc=0x400000
    # global pc

    def __init__(self):
        self.pipelineprocessor=[]

    def refresh(self):
        if self.pipelineprocessor and self.pipelineprocessor[-1].values_initialsed[-1
            self.pipelineprocessor.pop()
            #this is used for refreshing the PipeLine
            #print(self.pipelineprocessor)


    def append(self,instructions_used:Pipeline_instructions_used):
        self.refresh()
        #for appending the given Instructions
        if len(self.pipelineprocessor)<5:
            self.pipelineprocessor=[instructions_used]+self.pipelineprocessor


    def run(self):
```

```
global pc
while pc<len(instruction_memory)-1 or self.pipelineprocessor:
    self.refresh()
    instructionsGet=Pipeline_instructions_used()
    print(f"The value of pc is {pc}",file=output)
    for i in self.pipelineprocessor[::-1]:
        i.run(pc)
    if pc<len(instruction_memory)-1:
        #FINAL RUN FUNCTIONN
        self.append(instructionsGet)
        self.pipelineprocessor[0].run(pc)
        # if pc < len(instruction_memory) - 1:
        # self.append(instructionsGet)
        # self.pipelineprocessor[0].run(pc)
    pc+=1
    clk+=1
    if(a==2): #considering the differnet cases....
        # print(mips_registers_values,file=output)
        print('-------------------------------------------------------
    elif(a==1):
        print("\n",file=output)
        print("\n",file=output)
        print(mips_registers_values['$t1'],file=output)
        print('-------------------------------------------------------

ineprocessor()
```

This is from Pipeline

How is Our Different->

Using the concept of function definition and using of classes only for the cases of Pipeline. Minimalistic logic and using a simpler .asm code.Output layout and printing and defining the functions wherever necessary.

->Hemang Seth IMT2022098
->Rutul Patel IMT2022021