MIPS Processor Design

Brief:

The **MIPS processor** is a type of computer chip known for its efficiency in executing a specific set of instructions. It operates on a RISC (Reduced Instruction Set Computing) architecture, emphasizing simplicity and speed by using a smaller set of basic instructions. This design allows the processor to perform tasks quickly and efficiently, making it a reliable choice for various computing operations.

Aim:

1. **Non-pipelined MIPS Processor :** To create a MIPS processor that, as supposed to, takes instructions in hex/binary and runs through all stages such as Instruction Fetch, Instruction decode, Execute, Memory and Write Back in a non-pipelined fashion, runs one instruction per clock cycle.
2. **Pipelined MIPS Processor :** To create a MIPS processor that runs through all stages of IF, ID, EX, Mem, WB and runs instructions parallelly in a pipelined fashion , also handling data hazards and control hazards

Non-Pipelined Processor:

A non-pipelined version of MIPS represents a computer architecture where instructions are executed sequentially, one after another, without overlapping stages in their processing. In this design, each instruction undergoes a series of phases, such as instruction fetch, decode, execute, memory access, and write-back, without parallelism. While this sequential approach simplifies the control and coordination of instructions, it

may result in longer execution times and lower throughput, as the processor cannot begin processing the next instruction until the current one has been fully completed.

In our project we have tried to simulate the non-pipelined processor using Control Signals

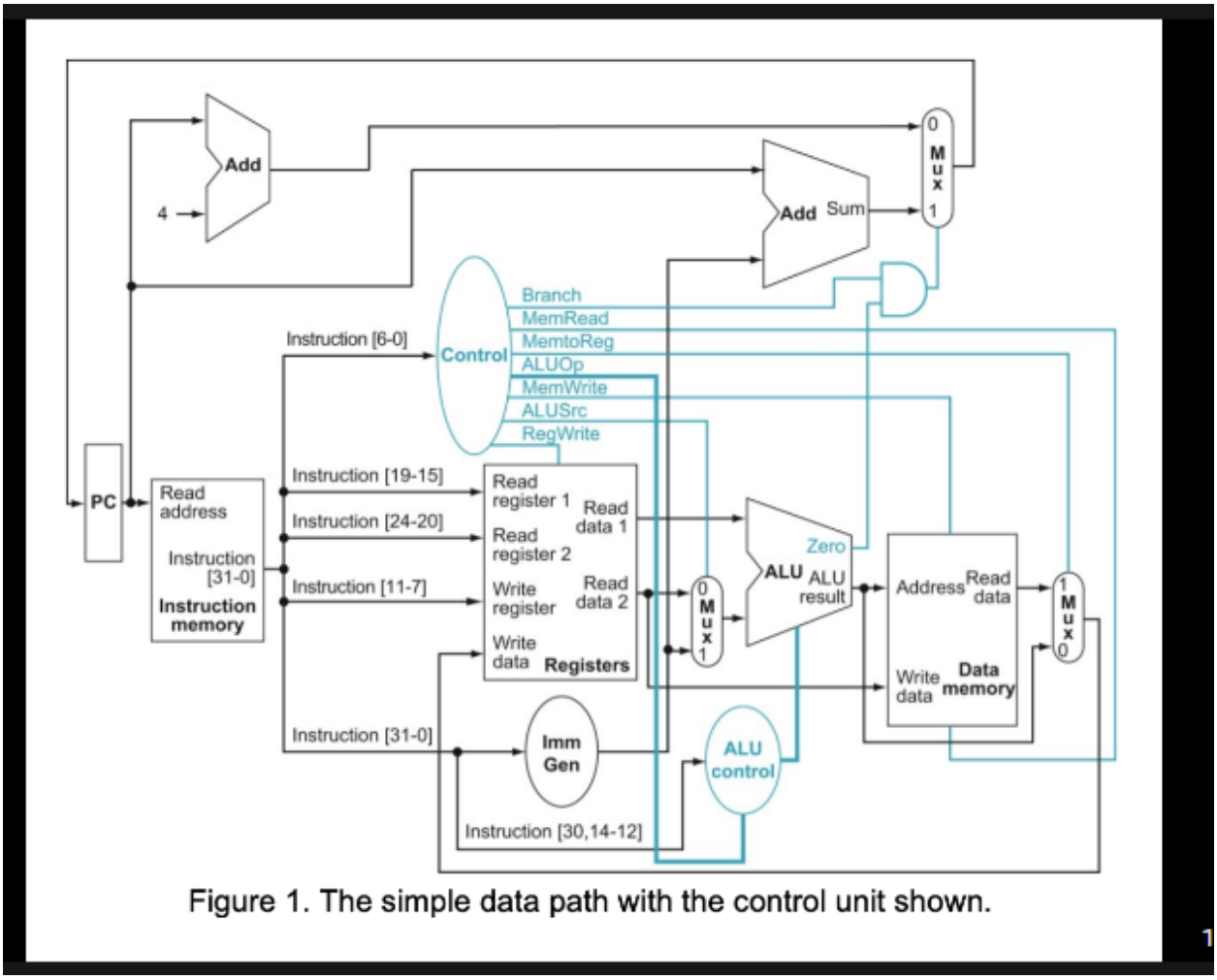


Figure 1. The simple data path with the control unit shown.

STAGES:

Instruction Memory:

```
def Create_Instr_set():
    global num_lines, strt_add
    strt_add = pc
    in_file=open("sorting.txt","r")
    instr_list = in_file.readlines()
    num_lines = len(instr_list)
    for line in instr_list:
        instr_mem[strt_add] = '0x'+line.rstrip('\n')
        strt_add+=4
```

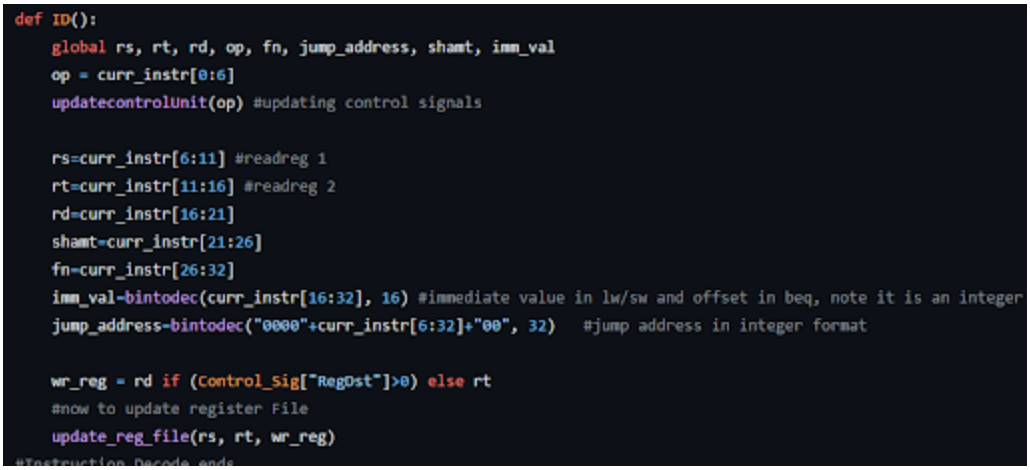
From the appropriate machine code file (**sorting.txt** in this case), **instr_list** creates a list with each instruction as an element. Each instruction is then added to the **Instruction Memory** at appropriate locations using a dummy PC variable (**strt_add**).

Instruction Fetch(IF):

```
def Create_Instr_set():
    global num_lines, strt_add
    strt_add = pc
    in_file=open("sorting.txt","r")
    instr_list = in_file.readlines()
    num_lines = len(instr_list)
    for line in instr_list:
        instr_mem[strt_add] = '0x'+line.rstrip('\n')
        strt_add+=4
```

In this stage, the **pc** and **cycle_count** is incremented and the next instruction to be executed is fetched from **instr_mem** using variable **pc**

Instruction Decode(ID):



```
def ID():
    global rs, rt, rd, op, fn, jump_address, shamt, imm_val
    op = curr_instr[0:6]
    updatecontrolUnit(op) #updating control signals

    rs=curr_instr[6:11] #readreg 1
    rt=curr_instr[11:16] #readreg 2
    rd=curr_instr[16:21]
    shamt=curr_instr[21:26]
    fn=curr_instr[26:32]
    imm_val=bintodec(curr_instr[16:32], 16) #immediate value in lw/sw and offset in beq, note it is an integer
    jump_address=bintodec("0000"+curr_instr[6:32]+"00", 32)   #jump address in integer format

    wr_reg = rd if (Control_Sig["RegDst"]>0) else rt
    #now to update register File
    update_reg_file(rs, rt, wr_reg)
#Instruction Decode ends
```

The **op** (opcode) is retrieved along with all possible fields such as **rs**, **rt**, **rd**, **shamt**, **fn**, **im_val** (immediate value) and **jump_address**. The fields are later used accordingly depending on the format of the instruction used. **updatecontrolUnit** function sets the control signal in response to the opcode decoded. **update_reg_file** function passes all the decoded registers and its data to the **Execute** phase.

Execute Instruction(EX):

```
def EX():
    global pc
    updateAluControl(fn, Control_Sig["aluop"]) #updating alucontrol signals

    in1 = Register_File["rd_data1"]
    in2 = imm_val if (Control_Sig["ALUSrc"]>0) else Register_File["rd_data2"]
    performALU(in1, in2)    #performing ALU operations ALU is updated

    #performing mux1
    if(Control_Sig["Branch"] and ALU["zero"]):
        pc += 4*imm_val
    #performing mux2
    if(Control_Sig["Jump"]):
        pc = jump_address
#Instruction Execute ends
```

According to the control signal (**Control_Sig**) which was set appropriately using the opcode decoded (**op**), and along with the function field (**fn**), the ALU control signals are updated using the **updateAluControl** function.

The ALU performs operation on the register values decoded (**in1**, **in2**) from **Instruction Decode stage** using **performALU** function.

A separate ALU dictionary is used to store the register data decoded (**in1**, **in2**), result of the ALU operation (**res)** and the ALU Control Signal (**aluc**).

The key “**zero**” is to handle **BEQ** and **J** (Jump).

```
#ALU
ALU = {"in1":'', "in2":'', "aluc":aluc, "zero":0, "res":''}
```

```
def updateAluControl(fn, aluop):
    global aluc
    if aluop=='00' or (aluop=='10' and fn=='100000'): #LW/SW add or ADD/ADDI
        aluc = '010'
    elif aluop=='01' or (aluop=='10' and fn == '100010'): #BEQ or SUB
        aluc = '110'
    elif aluop=='10' and fn == '000010':    #MUL
        aluc = '011'    #let MUL alu control be mul
    elif aluop=='10' and fn == '100100':   #and
        aluc = '000'
    elif aluop=='10' and fn == '100101':   #or
        aluc = '001'
    elif aluop=='10' and fn == '101010':  #slt
        aluc = '111'
```

setting alu control according to function field and aluop

Memory Access(MEM):

```
def MEM():
    global rd_data_from_mem
    if(Control_Sig["MemWrite"]==1):
        addr = ALU["res"]
        data_to_be_written = Register_File["rd_data2"]
        data_mem[addr] = data_to_be_written

    if(Control_Sig["MemRead"]==1):
        addr_to_read_from = ALU["res"]
        rd_data_from_mem = data_mem[addr_to_read_from]
#Memory Access ends
```

rd_data_from_mem is initially set to an empty string.

If the instruction proceeds to write data into the memory (**Control_Sig[“MemWrite”]==1**), address (**addr**) is assigned the result from ALU stage (**ALU[“res”]**) and the data to be written is obtained from the Instruction **Decode stage** (**Register_File[“rd_data2”]**).

If the instruction proceeds to read data from the memory (**Control_Sig[“MemRead”]==1**), the address to be read from (**addr_to_read_from**) is assigned from ALU stage (**ALU[“res”]**) and data is fetched from the memory.

Write Back(WB):

```
data_write_back = ''
def WB():
    global data_write_back
    if(Control_Sig["MemtoReg"]):
        data_write_back = rd_data_from_mem
    else:
        data_write_back = ALU["res"]

    if(Control_Sig["RegWrite"]):
        write_into_reg(data_write_back)
#WriteBack ends
```

Depending on the control signals (**Control_Sig[“MemtoReg’]**), data is sent to the register memory from the **memory** or from the **result of the ALU** stage.

write_into_reg function updates the desired output back into the register memory.

MAIN:

```python
def main(): #main
    Create_Instr_set()
    print("Data Memory is:")
    print(data_mem)
    print("Register Memory is:")
    for key in regmem.keys():
        print(f'{regmem_name[key]} : {regmem[key]}')
    while(True):
        if(pc not in instr_mem.keys()):
            break
        IF()
        ID()
        EX()
        MEM()
        WB()
        print(f'Running inst at address : {pc} and clockcyle count: {cycle_count}')
        print("IF ID EX MEM WB")
    print(f'Time taken by processor assuming each clock cycle is 950ps is {cycle_count*950/1000: .2f} ns')
    print("Data Memory is:")
    print(data_mem)
    print("Register Memory is:")
    for key in regmem.keys():
        print(f'{regmem_name[key]} : {regmem[key]}')

if __name__ == "__main__":
    main()
```

Assumptions :

1)Each Instruction takes 950ps to run(avg), then the time the program takes is given as output along with number of cycles

2)pc starts from `0x00400000'.

3)implemented jump in EX stage

4)Also the initial data_memory is to be set directly in the code (but we could store and load the vaues)

The main() first creates an instruction set and then calls each stage (**IF, ID, EX, MEM, WB**) and prints the data in the register after the whole instruction set is completed, ie pc goes out of bound.

Binary Input of Factorial:

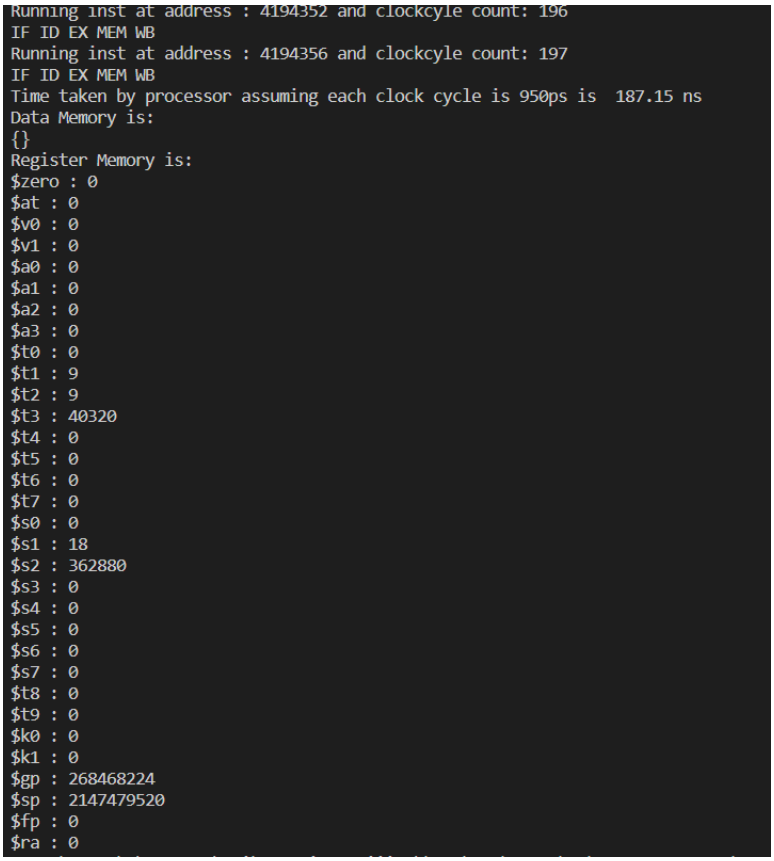
```
00100000000100010000000000001001 00100000000100100000000000000001
00100001001010010000000000000001 00010001001100010000000000001000
00100001001010010000000000000001 00100000000010100000000000000001
00000000000100100101100000100000 00010001001010100000000000000011
00000010010010111001000000100000 00100001010010100000000000000001
00001000000100000000000000000111 00001000000100000000000000000011
00000010001100011000100000100000
```

Input in 32 bit binary codes, this code calculates the value of **9!(9 factorial)**

Output Input of Factorial

```
PS C:\Users\Dhruv Kothari\OneDrive - iiit-b\Desktop\Sem-3\CA\MIPS_Processor\non-pipelined> python3 main.py
Data Memory is:
{}
Register Memory is:
$zero : 0
$at : 0
$v0 : 0
$v1 : 0
$a0 : 0
$a1 : 0
$a2 : 0
$a3 : 0
$t0 : 0
$t1 : 0
$t2 : 0
$t3 : 0
$t4 : 0
$t5 : 0
$t6 : 0
$t7 : 0
$s0 : 0
$s1 : 0
$s2 : 0
$s3 : 0
$s4 : 0
$s5 : 0
$s6 : 0
$s7 : 0
$t8 : 0
$t9 : 0
$k0 : 0
$k1 : 0
$gp : 268468224
$sp : 2147479520
$fp : 0
$ra : 0
Running inst at address : 4194308 and clockcyle count: 1
IF ID EX MEM WB
Running inst at address : 4194312 and clockcyle count: 2
IF ID EX MEM WB
```

initially all registers have value 0 except **$gp** and **$sp.**



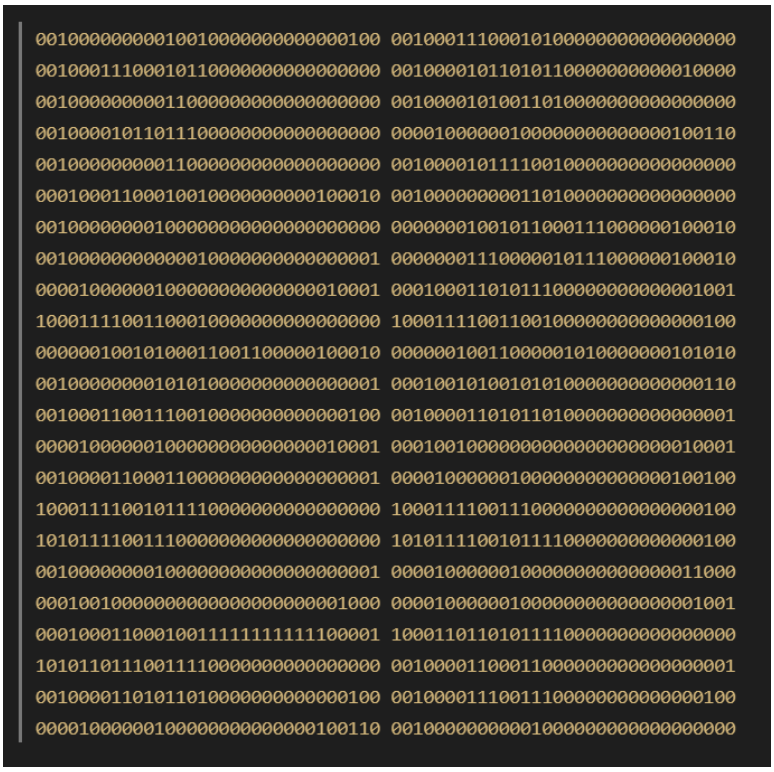
This process takes **186.15ns** and 197 clock cycles, the value of **9!** is later stored in **$s2,** ie 362880, also the value of **$t3** is **8!**

Binary Input for Sorting

Initially we set our data memory

```
data_mem = {0x10008000: 3, 0x10008004: 5, 0x10008008: 4, 0x1000800c : 2}
```

The memory is stored starting from **0x10008000** that **$gp** stores


00100000000010010000000000000100 00100011100010100000000000000000
00100011100010110000000000000000 00100001011010110000000000010000
00100000000011000000000000000000 00100001010011010000000000000000
00100001011011100000000000000000 00001000000100000000000000100110
00100000000011000000000000000000 00100001011110010000000000000000
00010001100010010000000000100010 00100000000011010000000000000000
00100000000100000000000000000000 00000001001011000111000000100010
00100000000000010000000000000001 00000001110000010111000000100010
00001000000100000000000000010001 00010001101011100000000000001001
10001111001100010000000000000000 10001111001100100000000000000100
00000010010100011001100000100010 00000010011000001010000000101010
00100000000101010000000000000001 00010010100101010000000000000110
00100011001110010000000000000100 00100001101011010000000000000001
00001000000100000000000000010001 00010010000000000000000000010001
00100001100011000000000000000001 00001000000100000000000000100100
10001111001011110000000000000000 10001111001110000000000000000100
10101111001110000000000000000000 10101111001011110000000000000100
00100000000100000000000000000001 00001000000100000000000000011000
00010010000000000000000000001000 00001000000100000000000000001001
00010001100010011111111111100001 10001101101011110000000000000000
10101101110011110000000000000000 00100001100011000000000000000001
00100001101011010000000000000100 00100001110011100000000000000100
00001000000100000000000000100110 00100000000001000000000000000000

The code sorts the numbers and starts storing the output from **$gp + 16**

Output of Sorting

```
PS C:\Users\Dhruv Kothari\OneDrive - iiit-b\Desktop\Sem-3\CA\MIPS_Processor\non-pipelined> python3 main.py
Data Memory is:
{268468224: 3, 268468228: 5, 268468232: 4, 268468236: 2}
Register Memory is:
$zero : 0
$at : 0
$v0 : 0
$v1 : 0
$a0 : 0
$a1 : 0
$a2 : 0
$a3 : 0
$t0 : 0
$t1 : 0
$t2 : 0
$t3 : 0
$t4 : 0
$t5 : 0
$t6 : 0
$t7 : 0
$s0 : 0
$s1 : 0
$s2 : 0
$s3 : 0
$s4 : 0
$s5 : 0
$s6 : 0
$s7 : 0
$t8 : 0
$t9 : 0
$k0 : 0
$k1 : 0
$gp : 268468224
$sp : 2147479520
$fp : 0
$ra : 0
Running inst at address : 4194308 and clockcyle count: 1
IF ID EX MEM WB
```

Initially data memory is the above thing

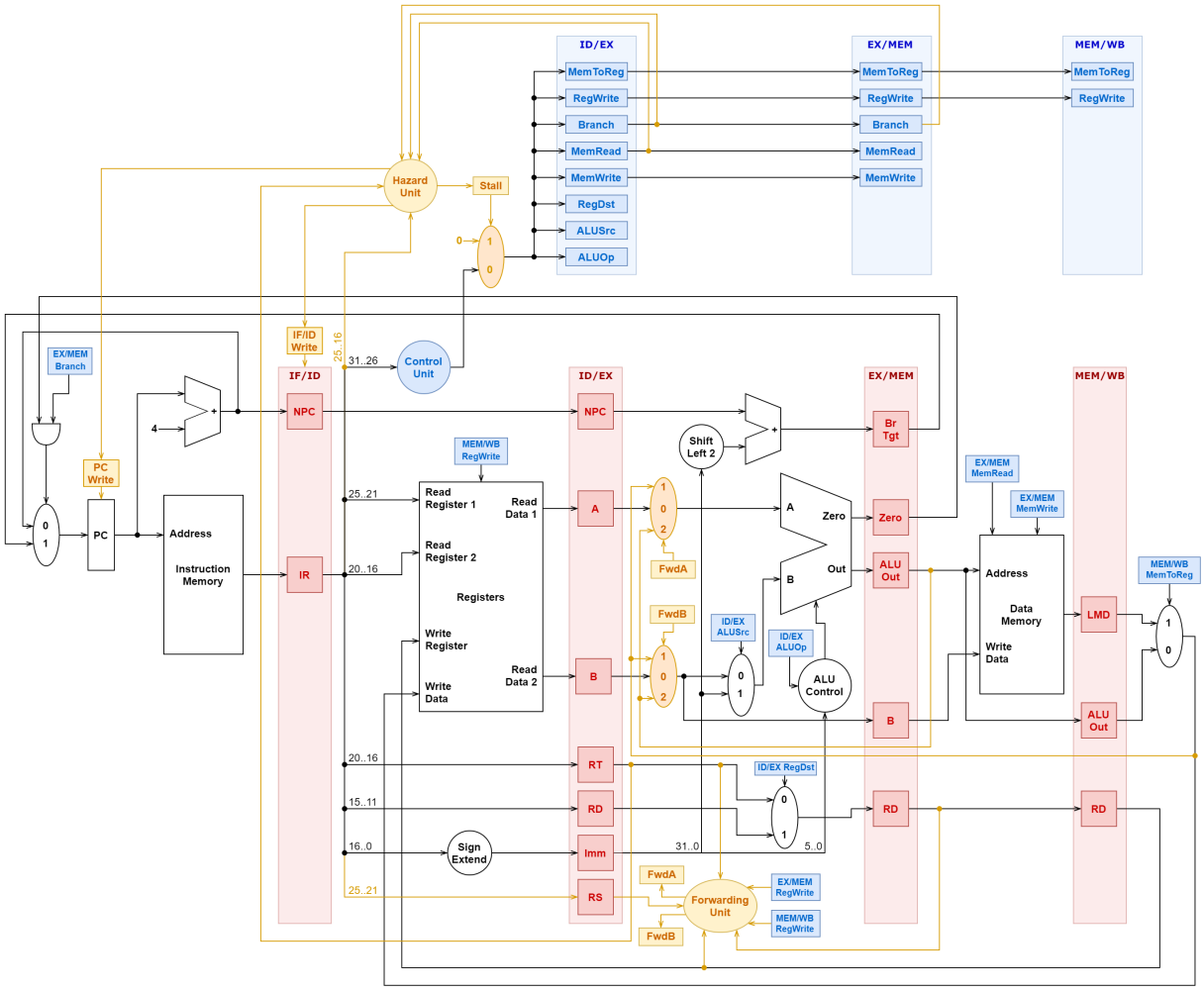
```
Running inst at address : 4194488 and clockcyle count: 175
IF ID EX MEM WB
Time taken by processor assuming each clock cycle is 950ps is  166.25 ns
Data Memory is:
{268468224: 3, 268468228: 5, 268468232: 4, 268468236: 2, 268468240: 2, 268468244: 3, 268468248: 4, 268468252: 5}
Register Memory is:
$zero : 0
$at : 1
$v0 : 0
$v1 : 0
$a0 : 0
$a1 : 0
$a2 : 0
$a3 : 0
$t0 : 0
$t1 : 4
$t2 : 268468224
$t3 : 268468240
$t4 : 3
$t5 : 0
$t6 : 0
$t7 : 3
$s0 : 0
$s1 : 3
$s2 : 2
$s3 : -1
$s4 : 1
$s5 : 1
$s6 : 0
$s7 : 0
$t8 : 2
$t9 : 268468240
$k0 : 0
$k1 : 0
```

The output address starts from **268468240**

Pipelined Processor:

A pipelined processor refers to a computer architecture that employs a pipeline structure to enhance instruction throughput and overall processing speed. In a pipelined design, the execution of instructions is divided into discrete stages, and multiple instructions are processed concurrently at different stages of the pipeline. This parallelism allows for a more efficient use of hardware resources and reduces the overall time taken to complete a sequence of instructions.

We forward IFID to IDEX, IDEX to EXMEM… after every cycle.



Stages:

The stages are the same as non-pipelined, just that we maintain pipelined-registers, ie- (IFID, IDEX, EXMEM, MEMWB), Control Unit, Forwarding Unit and Hazard Unit.

Forwarding:

In a pipelined MIPS processor, forwarding is a technique used to resolve data hazards and ensure correct data flow between pipeline stages. Data hazards occur when an instruction depends on the result of a previous instruction that has not yet completed its execution.

Forwarding allows the processor to bypass the need for waiting for the data to be written back to the register file before it can be used by subsequent instructions. This improves the efficiency of the pipeline by reducing stalls or delays.

The forwarding unit in the pipelined MIPS processor checks for data hazards by comparing the register numbers of the current instruction's source operands (**rs** and **rt**) with the register numbers of the destination operands (**rd**) of the previous instructions in the pipeline. If a match is found, the forwarding unit enables the forwarding path to forward the necessary data from the later pipeline stage **EXMEM** or **MEMWB** directly to the earlier stage **IDEX**.

This reduces the number of stalls that happens in the processor and improves the efficiency of the program.

```python
def updateforwading_sig(rs,rt):
    cntrlA, cntrlB = '00', '00' #from IDEX, no forwarding

    if(MEMWB["reg_write_data"] == rs and MEMWB["Control_Sig"]["RegWrite"]==1 and MEMWB["reg_write_data"]!='' and (EXMEM["reg_write_data"] != rs or
        cntrlA = '01'   #from EXMEM, forwarding
    elif(EXMEM["reg_write_data"] == rs and EXMEM["Control_Sig"]["RegWrite"]==1 and EXMEM["reg_write_data"]!=''):
        cntrlA = '10'   #from MEMWB, forwarding



    if(MEMWB["reg_write_data"] == rt and MEMWB["Control_Sig"]["RegWrite"]==1 and MEMWB["reg_write_data"]!='' and (EXMEM["reg_write_data"] != rt or
        cntrlB = '01'
    elif(EXMEM["reg_write_data"] == rt and EXMEM["Control_Sig"]["RegWrite"]==1 and EXMEM["reg_write_data"]!=''):
        cntrlB = '10'

    return cntrlA, cntrlB

def forward_mul(cntrlsig, rd_data):#depending upon the forwarding signals you return the right value of rd_data
    if(cntrlsig == '10'):
        rd_data = EXMEM["ALU_res"]
    elif(cntrlsig == '01'):
        if(MEMWB["Control_Sig"]["MemtoReg"]==1):
            rd_data = MEMWB["mem_rd_data"]
        else:
            rd_data = MEMWB["ALU_res"]
    return rd_data
```

here **update_forwarding** updates forwarding signals according to the states of IDEX, EXMEM, MEMWB, and **forward_mul** will forward according to forwarding signals and control signals.

Hazards:

During a load dependency in the very next instruction, we need to implement stalling, because the dependent register value is extracted in MEM stage and only then it can be forwarded to EX stage of the next instruction. Stalling solves this efficiently. After stalling we can forward from MEMWB to IDEX phase.

```python
if(EXMEM["Control_Sig"]["MemtoReg"] and (IDEX["rs"]==EXMEM["reg_write_data"] or IDEX["rt"]==EXMEM["reg_write_data"])):
    st = 1  #if EXMEM is load(ie MemtoReg ==1) and rs or rt of IDEX is same as rd in EXMEM then we stall
    print("stalling")
```

st is set to 1 if load dependency in the next instruction. if **st** is 1 then in the next iteration we only move **lw** instruction to **MEM** phase but not move the next instructions, ie implementing a stall.

beq/jump: if beq executes as true or jump is in EX stage then we flush all the next instruction data, that is we flush IFID, IDEX pipelined registers.

```
#BEQ and jump start
if(to_jump):
    print("Flushing after jump/branch")
    to_jump = 0
    flushEXMEM()    #flusing the pipelined registers to get rid of the the values in pipelined reg
    flushIDEX()
    flushIFID()
    pc = pc2    #updating pc
    instr[0] = None #flusing IF and ID
    instr[1] = None
#end
```

to_jump = 1 if we branch or jump and flush if to_jump==1

Implementation:

Implementing a queue, **q** that is always of length 5, initially all set to None, then running such that IF is implemented only if **q[0]** is not None, ID is implemented only if **q[1]** is not None, and so on. At end of running all phases, pushing pc(a new instruction if stall is false) and popping the last thing in the queue. Also after all phases, implementing forwarding from EXMEM or MEMWB to IDEX, if any dependencies occur.

```
while(True):
    if(st):
        instr.insert(2, None)
    elif(pc not in instr_mem.keys()):
        instr.insert(0, None)
    else:
        instr.insert(0, pc) #storing the current instruction
    instr.pop()
    if(instr.count(None)==5):   #breaking if everything is none in instr queue
        break
    Reg_update = {} #dictionary that contains the updates that is to be done for each pipelined register after a clock cycle
    cycle_count+=1
    print(f'Running inst at address : {pc} and clockcyle count: {cycle_count}')
    to_jump = 0
    for i in range(4, -1, -1):
        if(instr[i] is None):
            continue
        if(i==4):
            print("WB", end=" ")
            WB()
        elif(i==3):
            print("MEM", end = " ")
            Reg_update[i] = MEM()
            if(st==1):  #if stall then flushexmem and break, ie dont execute EX, ID, IF
                st=0
                flushEXMEM()
                break
        elif(i==2):
            print("EX", end = " ")
            to_jump, pc2, Reg_update[i] = EX()
        elif(i==1):
            print("ID", end = " ")
            Reg_update[i] = ID()
        elif(i==0):
            print("IF", end = " ")
            pc, Reg_update[i] = IF(pc)
    print()
    #forwarding the right values into IDEX register depending upon the forwarding control signals
    FOWD()
    #updating pipelined registers
    update_pipelined(Reg_update)
```

Binary Input of Factorial:

```
00100000000100010000000000001001 00100000000100100000000000000001
00100001001010010000000000000001 00010001001100010000000000001000
00100001001010010000000000000001 00100000000010100000000000000001
00000000000100100101100000100000 00010001001010100000000000000011
00000010010010111001000000100000 00100001010010100000000000000001
00001000000100000000000000000111 00001000000100000000000000000011
00000010001100011000100000100000
```

Input in 32 bit binary codes, this code calculates the value of **9!(9 factorial)**

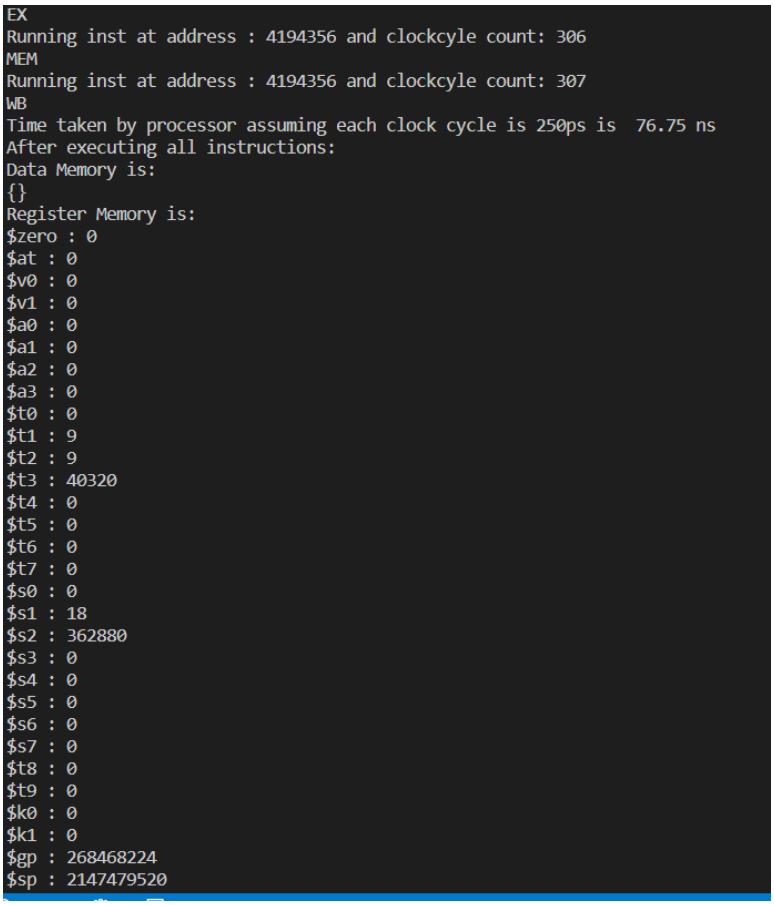
Output Input of Factorial:

```
PS C:\Users\Dhruv Kothari\OneDrive - iiit-b\Desktop\Sem-3\CA\MIPS_Processor\Pipelined> python3 main.
Data Memory is:
{}
Register Memory is:
$zero : 0
$at : 0
$v0 : 0
$v1 : 0
$a0 : 0
$a1 : 0
$a2 : 0
$a3 : 0
$t0 : 0
$t1 : 0
$t2 : 0
$t3 : 0
$t4 : 0
$t5 : 0
$t6 : 0
$t7 : 0
$s0 : 0
$s1 : 0
$s2 : 0
$s3 : 0
$s4 : 0
$s5 : 0
$s6 : 0
$s7 : 0
$t8 : 0
$t9 : 0
$k0 : 0
$k1 : 0
$gp : 268468224
$sp : 2147479520
$fp : 0
$ra : 0
Running inst at address : 4194304 and clockcyle count: 1
IF
Running inst at address : 4194308 and clockcyle count: 2
ID IF
```

initially all registers have value 0 except **$gp** and **$sp.**

```
WB MEM EX ID IF
Flushing after jump/branch
Running inst at address : 4194332 and clockcyle count: 36
WB MEM IF
Running inst at address : 4194336 and clockcyle count: 37
WB ID IF
Running inst at address : 4194340 and clockcyle count: 38
EX ID IF
Flushing after jump/branch
Running inst at address : 4194348 and clockcyle count: 39
MEM IF
Running inst at address : 4194352 and clockcyle count: 40
WB ID IF
Running inst at address : 4194356 and clockcyle count: 41
EX ID
Flushing after jump/branch
Running inst at address : 4194316 and clockcyle count: 42
MEM IF
Running inst at address : 4194320 and clockcyle count: 43
WB ID IF
Running inst at address : 4194324 and clockcyle count: 44
EX ID IF
Running inst at address : 4194328 and clockcyle count: 45
MEM EX ID IF
Running inst at address : 4194332 and clockcyle count: 46
WB MEM EX ID IF
Running inst at address : 4194336 and clockcyle count: 47
WB MEM EX ID IF
Running inst at address : 4194340 and clockcyle count: 48
WB MEM EX ID IF
Running inst at address : 4194344 and clockcyle count: 49
WB MEM EX ID IF
Running inst at address : 4194348 and clockcyle count: 50
WB MEM EX ID IF
Running inst at address : 4194352 and clockcyle count: 51
WB MEM EX ID IF
Flushing after jump/branch
```

printing stalling and flushing when the hazard occurs



This process takes **76.75ns** and 307 clock cycles, the value of **9!** is later stored in **$s2,** ie 362880, also the value of **$t3** is **8!**

Binary Input for Sorting

```
00100000000010010000000000000100 00100011100010100000000000000000
00100011100010110000000000000000 00100001011010110000000000010000
00100000000011000000000000000000 00100001010011010000000000000000
00100001011011100000000000000000 00001000000100000000000000100110
00100000000011000000000000000000 00100001011110010000000000000000
00010001100010010000000000100010 00100000000011010000000000000000
00100000000100000000000000000000 00000001001011000111000000100010
00100000000000010000000000000001 00000001110000010111000000100010
00001000000100000000000000010001 00010001101011100000000000001001
10001111001100010000000000000000 10001111001100100000000000000100
00000010010100011001100000100010 00000010011000001010000000101010
00100000000101010000000000000001 00010010100101010000000000000110
00100011001110010000000000000100 00100001101011010000000000000001
00001000000100000000000000010001 00010010000000000000000000010001
00100001100011000000000000000001 00001000000100000000000000100100
10001111001011110000000000000000 10001111001110000000000000000100
10101111001110000000000000000000 10101111001011110000000000000100
00100000000100000000000000000001 00001000000100000000000000011000
00010010000000000000000000001000 00001000000100000000000000001001
00010001100010011111111111100001 10001101101011110000000000000000
10101101110011110000000000000000 00100001100011000000000000000001
00100001101011010000000000000100 00100001110011100000000000000100
00001000000100000000000000100110 00100000000001000000000000000000
```

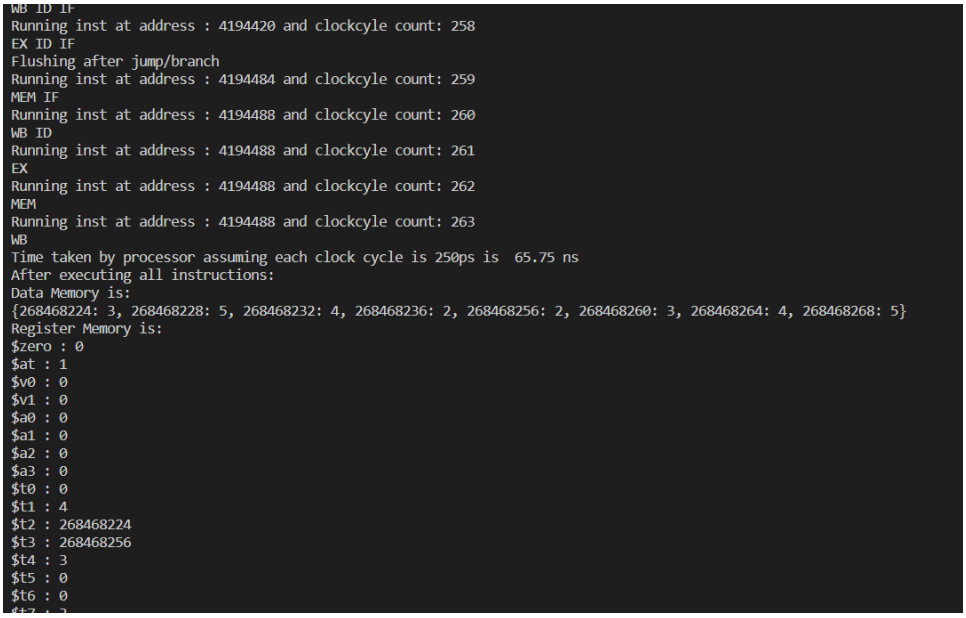
The code sorts the numbers and starts storing the output from **$gp + 16**

```
PS C:\Users\Dhruv Kothari\OneDrive - iiit-b\Desktop\Sem-3\CA\MIPS_Processor\Pipelined> python3 main.py
Data Memory is:
{268468224: 3, 268468228: 5, 268468232: 4, 268468236: 2}
Register Memory is:
$zero : 0
$at : 0
$v0 : 0
$v1 : 0
$a0 : 0
$a1 : 0
$a2 : 0
$a3 : 0
$t0 : 0
$t1 : 0
$t2 : 0
$t3 : 0
$t4 : 0
$t5 : 0
$t6 : 0
$t7 : 0
$s0 : 0
$s1 : 0
$s2 : 0
$s3 : 0
$s4 : 0
$s5 : 0
$s6 : 0
$s7 : 0
$t8 : 0
$t9 : 0
$k0 : 0
$k1 : 0
$gp : 268468224
$sp : 2147479520
$fp : 0
$ra : 0
Running inst at address : 4194304 and clockcyle count: 1
IF
```

initial memory and registers

```
Running inst at address : 4194312 and clockcyle count: 3
EX ID IF
Running inst at address : 4194316 and clockcyle count: 4
MEM EX ID IF
Running inst at address : 4194320 and clockcyle count: 5
WB MEM EX ID IF
Running inst at address : 4194324 and clockcyle count: 6
WB MEM EX ID IF
Running inst at address : 4194328 and clockcyle count: 7
WB MEM EX ID IF
Running inst at address : 4194332 and clockcyle count: 8
WB MEM EX ID IF
Running inst at address : 4194336 and clockcyle count: 9
WB MEM EX ID IF
Running inst at address : 4194340 and clockcyle count: 10
WB MEM EX ID IF
Flushing after jump/branch
Running inst at address : 4194456 and clockcyle count: 11
WB MEM IF
Running inst at address : 4194460 and clockcyle count: 12
WB ID IF
Running inst at address : 4194464 and clockcyle count: 13
EX ID IF
Running inst at address : 4194468 and clockcyle count: 14
MEM EX ID IF
stalling
Running inst at address : 4194472 and clockcyle count: 15
WB MEM
Running inst at address : 4194472 and clockcyle count: 16
WB EX ID IF
```

cycles in between, printing if flushing or stalling and printing the phases that get executed in that iteration



The output address starts from **268468256,** the numbers are sorted

Assumptions:

1)Implementing jump and beq both in EX phase

2)Assumed each clock cycle to be 250ps

Done by:

Dhruv Kothari(IMT2022114)

R Harshavardhan(IMT2022515)