Assignment 2

EG 211 – Computer Architecture

Khadarabad Tahir Mohammed - IMT2022100, Khadarabad.Mohammed@iiitb.ac.in

Sasi Snigdha Yadavalli - IMT2022571, YS.Snigdha@iiitb.ac.in

Non-Pipelined:

```python
#File I/O for getting the machine code as a list of str
def file_handling(file_name:str):
    f = open(file_name, 'r')
    machine_code = []
    for line in f.readlines():
        machine_code.append(line.rstrip(" ")[: -1])#to remove 0x and \n characters
    f.close()
    #print(machine_code)
    return machine_code #returns list of instructions in hexadecimal

#initialising an instruction memory for usage
def create_instruction_memory():
    hexadecimal_str = "00400000"
    addresses = []
    while hexadecimal_str != "00402400":
        hexadecimal_str = bin(int(hexadecimal_str, 16))[2:].zfill(32)
        addresses.append(hexadecimal_str)
        nextpc = str(bin(int(hexadecimal_str, 2) + int('4', 16)))[2:].zfill(32)
        hexadecimal_str = str(hex(int(nextpc, 2))[2:].zfill(8))
    return dict(zip(addresses, [' ' for _ in range(len(addresses))]))#create a dictionary with keys

#initialising register file
def create_register_file():
    register_no = []
    start = "00000"
    while start != "100000":
        register_no.append(start)
        if start == "11111":
            break
        start = bin(int(start, 2) + int("00001", 2))[2:].zfill(5)
    return dict(zip(register_no, [''.join(['0' for _ in range(32)])] + [' ' for i in range(len(regis

#initialising data memory
def create_data_memory():
    start = bin(int("10010000", 16))[2:].zfill(32)
    data = []
    while start != bin(int("1001015c", 16))[2:].zfill(32):
        data.append(start)
        start = bin(int(start, 2) + int("0100", 2))[2:].zfill(32)
    return dict(zip(data, [' ' for i in range(len(data))]))#create a dictionary with keys from 10010000 to 10
```

It returns a list of hexadecimal instructions after reading them from the given file

Creates the instruction memory with the previous list

Creates 32 registers, in the form of a dictionary from “00000” to “11111”

Creates the Data Memory

```
#helpers
register_file = create_register_file()
register_file["lo"] = ' '
data_memory = create_data_memory()
instruction_memory = create_instruction_memory()
register_file[bin(int("29", 10))[2:].zfill(5)] = "11111111111
```

Initialize various memory using the previously defined functions
Seperate opcodes in r, j, and i formats.
Take note of func values.

```
#helpers
r_format = ['000000']
i_format = ["100011", "101011", "000100", "001000", "001010", "000101", "111111", "001010"]
j_format = ["000010"]
code = input_function()

func_ALU = {"100000":"0010", "100010":"0110", "100100":"0000", "100101":"0001", "101010":"0111", "000000":"0100", "011000":"0101", "010010":"100
lw = "100011"
sw = "101011"
beq = "000100"
bne = "000101"
addi = "001000"
subi = "111111"
slti = "001010"
```

Take the input values for the start address, end address, number of integers etc. in Python itself.
Assign these values to Data Memory or Registers.

```
#Taking input for respective code
def input_function():
    name = None
    g = input("Enter type of code: ")
    if g == "S":
        name = "dumpsorting.txt"
        n = int(input("Enter length of array: "))
        inp_add = int(input("Enter input address: "))
        register_file[bin(int("9", 10))[2:].zfill(5)] = bin(n)[2
        register_file[bin(int("10", 10))[2:].zfill(5)] = bin(inp
        #print(register_file)
        start_add = register_file[bin(int("10", 10))[2:].zfill(5
        #print(start_add)
        for i in range(n):
            x = int(input())
            data_memory[start_add] = bin(int(str(x), 10))[2:].zf
```

```
    elif g == "C":
        name = "dumpconvolution.txt"
        n = int(input("Enter length of x: "))
        inp_add = int(input("Enter the x input address: "))
        register_file[bin(int("9", 10))[2:].zfill(5)] = bin(n)[2:].zfill(32)
        register_file[bin(int("10", 10))[2:].zfill(5)] = bin(inp_add)[2:].zfill(32)
        #print(register_file)
        start_add = register_file[bin(int("10", 10))[2:].zfill(5)]
        #print(start_add)
        for i in range(n):
            x = int(input())
            data_memory[start_add] = bin(int(str(x), 10))[2:].zfill(32)
            #print(data_memory[start_add])
            start_add = bin(int(str(start_add), 2) + int("100", 2))[2:].zfill(32)
            #print(start_add)
        int(input("Enter length of h: "))
        add = int(input("Enter h input address: "))
        ster_file[bin(int("11", 10))[2:].zfill(5)] = bin(n)[2:].zfill(32)
        ster_file[bin(int("13", 10))[2:].zfill(5)] = bin(inp_add)[2:].zfill(32)
        nt(register_file)
```

```
    elif g == "M":
        name = "dumpedhexmatrix.txt"
        m = int(input("Enter m: "))
        n = int(input("Enter n: "))
        p = int(input("Enter p: "))
        if n != p:
            print("Matrix Multiplication not possible")
            return
        q = int(input("Enter q: "))
        register_file[bin(int("9", 10))[2:].zfill(5)] = bin(m)[2:].zf
        register_file[bin(int("17", 10))[2:].zfill(5)] = bin(n)[2:].z
        register_file[bin(int("18", 10))[2:].zfill(5)] = bin(q)[2:].z
        x1_add = int(input("Enter first matrix input address: "))
        register_file[bin(int("10", 10))[2:].zfill(5)] = bin(x1_add)[
        start_add = register_file[bin(int("10", 10))[2:].zfill(5)]
        #print(start_add)
        for i in range(n*m):
            x = int(input())
            data_memory[start_add] = bin(int(str(x), 10))[2:].zfill(3
            #print(data_memory[start_add])
            start_add = bin(int(str(start_add), 2) + int("100", 2))[2
            #print(start_add)

        x2_add = int(input("Enter second matrix input address: "))
        register_file[bin(int("13", 10))[2:].zfill(5)] = bin(x2_add)[
        start_add = register_file[bin(int("10", 10))[2:].zfill(5)]
        #print(start_add)
        for i in range(n*q):
            x = int(input())
            data_memory[start_add] = bin(int(str(x), 10))[2:].zfill(3
            #print(data_memory[start_add])
            start_add = bin(int(str(start_add), 2) + int("100", 2))[2
            #print(start_add)
        y_add = int(input("Enter output address: "))
        register_file[bin(int("14", 10))[2:].zfill(5)] = bin(y_add)[2
```

```
    else:
        name = "machinec.txt"
        n = int(input("Enter length of x: "))
        inp_add = int(input("Enter the x input address: "))
        register_file[bin(int("9", 10))[2:].zfill(5)] = bin(n)[2:].zfill(32)
        register_file[bin(int("10", 10))[2:].zfill(5)] = bin(inp_add)[2:].zfill(32)
        #print(register_file)
        start_add = register_file[bin(int("10", 10))[2:].zfill(5)]
        #print(start_add)
        for i in range(n):
            x = int(input())
            data_memory[start_add] = bin(int(str(x), 10))[2:].zfill(32)
            #print(data_memory[start_add])
            start_add = bin(int(str(start_add), 2) + int("100", 2))[2:].zfill(32)
            #print(start_add)
        n = int(input("Enter length of h: "))
        inp_add = int(input("Enter h input address: "))
        register_file[bin(int("11", 10))[2:].zfill(5)] = bin(n)[2:].zfill(32)
        register_file[bin(int("13", 10))[2:].zfill(5)] = bin(inp_add)[2:].zfill(32)
        #print(register_file)
        start_add = register_file[bin(int("13", 10))[2:].zfill(5)]
        #print(start_add)
        for i in range(n):
            x = int(input())
            data_memory[start_add] = bin(int(str(x), 10))[2:].zfill(32)
            #print(data_memory[start_add])
            start_add = bin(int(str(start_add), 2) + int("100", 2))[2:].zfill(32)
            #print(start_add)
        out_add = int(input("Enter output address: "))
        register_file[bin(int("14", 10))[2:].zfill(5)] = bin(out_add)[2:].zfill(32)
    #print(name)
    add_to_memory(name, g)
    return g
```

```python
#Instruction class to denote one particular instruction
class Instruction:
    #initialisation
    def __init__(self, pc):
        self.pc = pc
        self.nextpc = str(str(bin(int(pc, 2) + int('4', 16))))[2:].zfill(32)
        self.opcode = "0"
        self.machine_code = ["0"]
        self.rs = "0"
        self.rt = "0"
        self.rd = "0"
        self.shamt = "0"
        self.funct = "0"
        self.imm = "0"
        self.address = "0"
        self.ALUOp = "0"
        self.ALUa = "0"
        self.ALUb = "0"
        self.ALUc = "0"
        self.ALUout = "0"
        self.ALUSrc = "0"
        self.branch = "0"
        self.regWrite = "0"
        self.regDst = "0"
        self.memWrite = "0"
        self.memToreg = "0"
        self.memRead = "0"
        self.jump = "0"
        self.readData1 = "0"
        self.readData2 = "0"
        self.memReadp = "0"
        self.add = "0"
        self.write = "0"
        self.writeData = "0"
        self.format = 'n'
```

Instructions are defined as objects and have their own attributes, like control signals, registers, immediate, shift amount etc.

We initialize these with certain default values every time the constructor is called. (listed here)

```python
#generate ALU control signals
def generate_ALU_control(self):
    if self.ALUOp == "00":
        self.ALUc = "0010"
    elif self.ALUOp == "01":
        self.ALUc = "0110"
    elif self.ALUOp == "10":
        self.ALUc = func_ALU[self.funct]
    elif self.ALUOp == "11":
        self.ALUc = "0111"

#generate all the control signals - main signals + ALU control signals
def generate_control_signals(self, format):
    if format == 'r':
        self.regWrite = 1
        self.regDst = 1
        self.ALUSrc = 0
        self.branch = "00"
        self.memWrite = 0
        self.memToreg = 0
        self.memRead = 0
        self.ALUOp = "10"
        self.jump = 0
    elif format == 'i':
        if self.opcode == lw:
            self.regWrite = 1
            self.regDst = 0
            self.ALUSrc = 1
            self.branch = "00"
            self.memWrite = 0
            self.memToreg = 1
            self.ALUOp = "00"
            self.memRead = 1
            self.jump = 0
        elif self.opcode == sw:
            self.regWrite = 0
            self.regDst = 0
            self.ALUSrc = 1
            self.branch = "00"
            self.memWrite = 1
```

We generate ALU control signals depending on the ALUOp control signal.

Control signals are initially defined based on the opcode, which falls into three broad categories:

1. r - type
2. i - type
3. j - type

Some functions have control signals explicitly defined for them

```python
#Instruction fetch - retrieves instruction from instruction memory
def fetch(self, pc):
    if instruction_memory[self.pc] == '' or instruction_memory[self.pc] == ' ':
        return
    self.machine_code = bin(int(instruction_memory[self.pc], 16))[2:].zfill(32)
```

Instruction Fetch,
involves setting the attribute
'machine_code' to the 32 bit binary string
corresponding to the current PC.

Instruction Decode,
involves splitting the machine code
previously fetched, based on its
format, then making the necessary
changes in the control signals

```python
#splits the machine code based on the format of instruction
def main_decoder_unit(self, machine_code, opcode):
    if self.opcode == "000000":
        self.rs = bin(int(machine_code[6:11], 2))[2:].zfill(5)
        self.rt = bin(int(machine_code[11:16], 2))[2:].zfill(5)
        self.rd = bin(int(machine_code[16:21], 2))[2:].zfill(5)
        self.shamt = bin(int(machine_code[21:26], 2))[2:].zfill(5)
        self.funct = bin(int(machine_code[26:], 2))[2:].zfill(6)
        return 'r'
    elif self.opcode in i_format:
        self.rs = bin(int(machine_code[6:11], 2))[2:].zfill(5)
        self.rt = bin(int(machine_code[11:16], 2))[2:].zfill(5)
        self.imm = bin(int(machine_code[16:], 2))[2:].zfill(16)
        return 'i'
    elif self.opcode in j_format:
        self.address = bin(int(machine_code[6:], 2))[2:].zfill(26)
        return 'j'
```

```python
#Instruction decode - identifies the opcode, spli
def decode(self, code):
    self.opcode = self.machine_code[:6]
    format = self.main_decoder_unit(self.machine_
    self.format = format
    print(format)
    self.generate_control_signals(format)
    if format == 'i':
        self.imm = self.sign_extend(format, self.
        print(self.imm)
    if format != 'j':
        print(self.rt)
        self.readData1 = register_file[self.rs]
        self.readData2 = register_file[self.rt]
        if self.regDst == 1:
            self.writeReg = self.rd
        else:
            self.writeReg = self.rt
        if self.branch == "10" or self.branch == "01":
            if self.readData1 == self.readData2 and self.jump == 0 and self.branch == "01":
                print(self.branch)
                self.nextpc = str(bin(int(self.nextpc, 2) + int(str(self.imm), 2) * int("100", 2)))[2:].zfill(32)
            elif self.readData1 != self.readData2 and self.jump == 0 and self.branch == "10":
                self.nextpc = str(bin(int(self.nextpc, 2) + int(str(self.imm), 2) * int("100", 2)))[2:].zfill(32)
    elif self.jump == 1:
        '''print(self.address)
        if code == 'C':
            self.address = bin(int("1000B", 16))[2:].zfill(26)
            self.nextpc = "0000" + self.address + "00"
        elif self.machine_code == "00001000000100000000000000010111":
            self.nextpc = "00000000000001000000000000010000"
        elif self.address == "00100000100000000000000100101":
            self.nextpc = " 00000000000001000000000000010000"
        '''
        self.nextpc = "0000" + self.address + "00"
```

In the Execute phase, guided by the ALU control signals, we calculate the required, memory addresses or values meant to be written back to memory, or a register, then return it.

```
#instruction execute - ALU operations
def execute(self):
    if self.format != 'j':
        self.ALUa = self.readData1
        if self.ALUSrc == 1: #ALU Src multiplexor
            self.ALUb = self.imm
        else:
            self.ALUb = self.readData2
            print(self.readData2)
        if self.ALUc == "0010": #add
            print("ALUa : ", self.ALUa)
            print("ALUb : ", self.ALUb)
            self.ALUout = bin(int(self.ALUa, 2) + int(str(self.ALUb), 2))[2:].zfill(32)
        elif self.ALUc == "0110": #sub
            if int(self.ALUa, 2) - int(str(self.ALUb), 2) >= 0:
                self.ALUout = bin(int(self.ALUa, 2) - int(str(self.ALUb), 2))[2:].zfill(32)
            else:
                self.ALUout = ''.join(['1' for _ in range(33 - len(bin(int(self.ALUa, 2) - int(str(self.ALUb),
        elif self.ALUc == "0111": #set less than
            print("ALU a: ", self.ALUa)
            print("ALU b: ", self.ALUb)
            if int(str(self.ALUa), 2) < int(str(self.ALUb), 2):
                self.ALUout = bin(int("1", 10))[2:].zfill(32)
            else:
                self.ALUout = bin(int("0", 10))[2:].zfill(32)
        elif self.ALUc == "0000":
            self.ALUout = bin(int(self.ALUa, 2) & int(str(self.ALUb), 2))[2:].zfill(32)
        elif self.ALUc == "0001":
            self.ALUout = bin(int(self.ALUa, 2) | int(str(self.ALUb), 2))[2:].zfill(32)
        elif self.ALUc == "0100":
            print("ALU b: ", self.ALUb)
            print("shift: ", self.shamt)
            self.ALUout = bin(int(str(self.ALUb), 2) * (pow(int("010",2), int(self.shamt, 2))))[2:].zfill(32)
        elif self.ALUc == "0101":
            self.ALUout = bin(int(self.ALUa, 2) * int(str(self.ALUb), 2))[2:].zfill(32)
        elif self.ALUc == "1000":
            self.ALUout = register_file["lo"]
        print(self.ALUout)
```

In the memory access stage, we read from the memory, or write into it, depending on the instruction at hand.

In the write back stage, we write values back to the register file, only modifying the data stored in the destination register

```
#Memory access - Memory read and write
def memory_access(self):
    if self.format != 'j':
        self.add = self.ALUout
        self.write = register_file[self.rt]
        if self.memWrite == 1:
            data_memory[self.add] = self.write
            print("Memory Updated", self.add, self.write)
        if self.memRead == 1:
            self.memReadp = data_memory[self.add]

#Write back - write to register file
def write_back(self):
    if self.format != 'j':
        if self.regWrite == 1:
            if self.memToreg == 1:
                self.writeData = self.memReadp
            else:
                self.writeData = self.ALUout
            register_file[self.writeReg] = self.writeData
            print("Register File updated", self.writeReg, self.writeData)
```

```python
class Non_Pipelined_Processor:
    def __init__(self):
        self.clock = 0

    def execution_time(self, pc):
        instr = Instruction(pc)
        for i in range(len(instruction_memory)):
            instr.fetch(pc)
            if instr.machine_code == ["0"]:
                print(self.clock)
                return
            print("Machine code: " , instr.machine_code)
            self.clock += 1
            instr.decode(code)
            print("Opcode: ", instr.opcode, end = " ")
            print("rs: ", instr.rs, end = " ")
            print("rt: ", instr.rt, end = " ")
            print("rd: ", instr.rd, end = " ")
            print("imm: ", instr.imm, end = " ")
            print("regDst: ", instr.regDst, end = " ")
            print("regWrite: ", instr.regWrite, end = " ")
            print("ALUOp: ", instr.ALUOp, end = " ")
            print("ALUc: ", instr.ALUc, end = " ")
            print("ALUSrc: ", instr.ALUSrc, end = " ")
            self.clock += 1
            instr.execute()
            print("ALUa: ", instr.ALUa, end = " ")
            print("ALUb: ", instr.ALUb, end = " ")
            print("ALUout: ", instr.ALUout, end = " ")
            self.clock += 1
            instr.memory_access()
            self.clock += 1
            instr.write_back()
            self.clock += 1
            print("PC, clock: ", instr.pc, self.clock)
            if instruction_memory[instr.nextpc] != ' ' or instruction_memory[instr.nextpc] != '':
                instr = Instruction(instr.nextpc)
            else:
                print("Clock: ", self.clock)
                return
```

This class is used to define a Non-Pipelined processor object, that not only executes the instructions one by one, but also keeps track of the execution time.

Here is an example of the processor sorting an array of 4 integers.

Memory shown here on the right (Unsorted above, Sorted below)

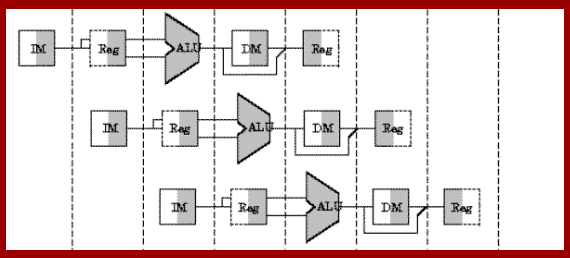
```
Enter type of code: S
Enter length of array: 4
Enter input address: 268501312
1
9
8
2
Enter output address: 268501344
```

```
00010000000000010000000100110000 :
00010000000000010000000100110100 :
00010000000000010000000100111000 :
00010000000000010000000100111100 :
00010000000000010000000101000000 : 00000000000000000000000000000001
00010000000000010000000101000100 : 00000000000000000000000000001001
00010000000000010000000101001000 : 00000000000000000000000000001000
00010000000000010000000101001100 : 00000000000000000000000000000010
00010000000000010000000101010000 :
00010000000000010000000101010100 :
00010000000000010000000101011000 :
00010000000000010000000101011100 :
00010000000000010000000101100000 : 00000000000000000000000000000001
00010000000000010000000101100100 : 00000000000000000000000000000010
00010000000000010000000101101000 : 00000000000000000000000000001000
00010000000000010000000101101100 : 00000000000000000000000000001001
00010000000000010000000101110000 :
00010000000000010000000101110100 :
00010000000000010000000101111000 :
00010000000000010000000101111100 :
00010000000000010000000110000000 :
00010000000000010000000110000100 :
00010000000000010000000110001000 :
00010000000000010000000110001100 :
```

Pipelined:

```
class Pipelined_Processor:#pipelined processor
    def __init__(self, pc):
        self.clock = 0
        self.start_pc = bin(int(pc, 16))[2:].zfill(32)
        print("Start pc: ", self.start_pc)
        self.if_id = {"machine_code": '', "nextpc": '', "rs": '', "rt": '', "rd": ''}
        self.id_ex = {"readData1": '', "readData2": '', "nextpc": self.if_id["nextpc"], "sign_extend": '',
                      "rs": self.if_id["rs"], "rt": self.if_id["rt"], "rd": self.if_id["rd"], "regDst": '',
                      "AluOp": '', "branch": '', "ALUSrc": '', "memRead": '', "memWrite": '', "regWrite": '',
                      "memToreg": ''}
        self.ex_mem = {"Aluout":'', "rd":'',"Alub" : '', "writeReg":'', "branch":self.id_ex["branch"], "ALUSr
        self.mem_wb = {"memReadp" : '', "Aluout":'', "writeReg":self.ex_mem["writeReg"], "regWrite":self.ex_m
        self.forwardA = "00"
        self.forwardB = "00"
        self.pcWrite = 1
        self.if_id_write = 1
        self.if_flush = 0
        self.lock = Lock()
```

This class is used to define a Non-Pipelined processor object, that executes the instructions stage by stage.




```
def hazard_detection_unit(self, pipeline_data, lock, instruction, c):#hazard detection unit
    with lock:
        if c == 1:
            return
        if pipeline_data["id_ex"]["memRead"] and ((pipeline_data["id_ex"]["rt"] == pipeline_data["id_ex"]["rs"]
            self.pcWrite = 0
            instruction.nextpc = instruction.pc
            self.if_id_write = 0
            '''pipeline_data["if_id"] = {"machine_code": '', "nextpc": '', "rs": '', "rt": '', "rd": ''}'''
            instruction.machine_code = instruction.machine_code
            instruction.rs = instruction.rs
            instruction.rt = instruction.rt
            pipeline_data["id_ex"]["regDst"] = 0
            pipeline_data["id_ex"]["AluOp"] = "00"
            pipeline_data["id_ex"]["branch"] = "00"
            pipeline_data["id_ex"]["ALUSrc"] = 0
            pipeline_data["id_ex"]["memRead"] = 0
            pipeline_data["id_ex"]["memWrite"] = 0
            pipeline_data["id_ex"]["regWrite"] = 0
            pipeline_data["id_ex"]["memToreg"] = 0
```

Hazard detection unit- Mainly for stalls after a load instruction(load use hazard since forwarding alone is not enough). Logic used- identifying if the instruction is a load instruction, if yes, retains the pc value and if id register values, all control signals are 0

```
def forwarding_unit(self, pipeline_data, lock, instruction, c):#forwarding unit
    with lock:
        if c == 1:
            return
        if pipeline_data["ex_mem"]["regWrite"] and pipeline_data["ex_mem"]["rd"] != ''.join(['0' for _ in range(5)]) and pipel
            self.forwardA = "10"
            instruction.ALUa = pipeline_data["ex_mem"]["Aluout"]
        if pipeline_data["ex_mem"]["regWrite"] and pipeline_data["ex_mem"]["rd"] != ''.join(['0' for _ in range(5)]) and pipel
            self.forwardB = "10"
            instruction.ALUb = pipeline_data["ex_mem"]["Aluout"]
        if pipeline_data["mem_wb"]["regWrite"] and pipeline_data["mem_wb"]["writeReg"] != ''.join(['0' for _ in range(5)]) and
            self.forwardA = "01"
            if pipeline_data["mem_wb"]["memToreg"] == 0:
                instruction.ALUa = pipeline_data["mem_wb"]["Aluout"]
            else:
                instruction.ALUa = pipeline_data["mem_wb"]["memReadp"]
        if pipeline_data["mem_wb"]["regWrite"] and pipeline_data["mem_
            self.forwardB = "01"
            if pipeline_data["mem_wb"]["memToreg"] == 0:
                instruction.ALUb = pipeline_data["mem_wb"]["Aluout"]
            else:
                instruction.ALUb = pipeline_data["mem_wb"]["memReadp"]
```

Forwarding unit- Tests whether the ex mem Rd field matches any of the source operands of next instruction...if yes, updates the ALU source. Also tests forwarding from mem wb register(to be done only when there is no dependency on the latest ex mem)

```
def flush_unit(self, pipeline_data, lock, instruction, c):#flush unit
    with lock:
        if c == 1:
            return
        if instruction.branch == "01" and instruction.branch_taken:
            with lock:
                self.if_flush = 1
                instruction.nextpc = instruction.branch_target_address
                pipeline_data["if_id"]["machine_code"] = ''.join(["0" for i in range(32)])
                self.pcWrite = 1
                self.if_id_write = 0
                pipeline_data["id_ex"]["regDst"] = 0
                pipeline_data["id_ex"]["AluOp"] = "00"
                pipeline_data["id_ex"]["branch"] = "00"
                pipeline_data["id_ex"]["ALUSrc"] = 0
                pipeline_data["id_ex"]["memRead"] = 0
                pipeline_data["id_ex"]["memWrite"] = 0
                pipeline_data["id_ex"]["regWrite"] = 0
                pipeline_data["id_ex"]["memToreg"] = 0
        if instruction.branch == "10" and instruction.branch_taken:
            with lock:
                self.if_flush = 1
                instruction.nextpc = instruction.branch_target_address
                pipeline_data["if_id"]["machine_code"] = ''.join(["0" for i in range(32)])
                self.pcWrite = 1
                self.if_id_write = 0
                pipeline_data["id_ex"]["regDst"] = 0
                pipeline_data["id_ex"]["AluOp"] = "00"
                pipeline_data["id_ex"]["branch"] = "00"
                pipeline_data["id_ex"]["ALUSrc"] = 0
                pipeline_data["id_ex"]["memRead"] = 0
                pipeline_data["id_ex"]["memWrite"] = 0
                pipeline_data["id_ex"]["regWrite"] = 0
                pipeline_data["id_ex"]["memToreg"] = 0
```

Flush unit- if flush control signal is used to flush pipeline based on branch decision and fast branching unit

This function executes all the stages, while checking for hazards.

```
def execute_instruction(self, pc, pipeline_data, lock, result_queue, c):
    with lock:
        self.clock += 1

    # Create an Instruction object
    instruction = Instruction(pc)

    # Fetch and decode
    #self.hazard_detection_unit(pipeline_data, self.lock, instruction, c)
    #self.flush_unit(pipeline_data, self.lock, instruction, c)
    instruction.fetch(instruction.pc)
    print("fetch done")
    with lock:
        '''
        pipeline_data["if_id"] = {"machine_code": '', "nextpc": '', "rs": '', "rt": '', "rd": ''}
        '''
        pipeline_data["if_id"] =  {"machine_code": instruction.machine_code, "nextpc": instruction.nextpc, "rs": instruction.
        self.clock += 1
        #for i in list(pipeline_data["if_id"].keys()):
        #   print("IF_ID: ", i , pipeline_data["if_id"][i])
    instruction.decode(code)
    print("decode done")
    with lock:
        '''
        pipeline_data["id_ex"] = {"readData1": '', "readData2": '', "nextpc": '', "sign_extend": '',
                                  "rs": '', "rt": '', "rd": '', "regDst": '', "AluOp": '', "branch": '',
                                  "ALUSrc": '', "memRead": '', "memWrite": '', "regWrite": '', "memToreg": ''}
        '''
        pipeline_data["id_ex"] = {"readData1": instruction.readData1, "readData2": instruction.readData2, "nextpc": instructi
                                  "rs": instruction.rs, "rt": instruction.rt, "rd": instruction.rd, "regDst": instruction.regDst,
                                  "ALUSrc": instruction.ALUSrc, "memRead": instruction.memRead, "memWrite": instruction.memWrite,
        #for i in list(pipeline_data["id_ex"].keys()):
            #print("ID_EX: ", i , pipeline_data["id_ex"][i])
        self.clock += 1
    self.forwarding_unit(pipeline_data, self.lock, instruction, c)
    # Execute
    instruction.execute()
    print("Execute done")
    with lock:
        '''
        pipeline_data["ex_mem"] = {"Aluout": '', "rd": '', "Alub": '', "writeReg": '', "branch": '',
                                   "ALUSrc": '', "memRead": '', "memWrite": '', "regWrite": '', "memToreg": ''}
        '''
        pipeline_data["ex_mem"] = {"Aluout": instruction.ALUout, "rd": instruction.rd, "Alub": instruction.ALUb, "writeReg'
                                   "ALUSrc": instruction.ALUSrc, "memRead": instruction.memRead, "memWrite": instruction.memWrit
        #for i in list(pipeline_data["ex_mem"].keys()):
            #print("EX_MEM: ", i , pipeline_data["ex_mem"][i])
        self.clock += 1

    # Memory Access
    instruction.memory_access()
    print("Mem done")
    with lock:
        '''
        pipeline_data["mem_wb"] = {"memReadp": '', "Aluout": '', "writeReg": '', "regWrite": '', "memToreg": ''}
        '''
        pipeline_data["mem_wb"] = {"memReadp": instruction.memReadp, "Aluout": instruction.ALUout, "writeReg": instruction.
        #for i in list(pipeline_data["mem_wb"].keys()):
            #print("MEM_WB: ", i , pipeline_data["mem_wb"][i])
        self.clock += 1

    # Write Back
    instruction.write_back()
    print("Wb done")
    with lock:
        self.clock += 1
        #print("Clock: ", self.clock)
    #print("Execute instr:", instruction.nextpc)
    result_queue.put(instruction.nextpc)
    #result_queue.put(self.clock)
```

```python
def execution_time(self, machine_codes):#execution time to integrate all the processes together
    manager = Manager()
    pipeline_data = manager.dict()

    # Initialize pipeline data
    pipeline_data["if_id"] = {"machine_code": "", "nextpc": "", "rs": "", "rt": "", "rd": ""}
    pipeline_data["id_ex"] = {"readData1": "", "readData2": "", "nextpc": pipeline_data["if_id"]["nextpc"],
                              "sign_extend": "", "rs": pipeline_data["if_id"]["rs"],
                              "rt": pipeline_data["if_id"]["rt"], "rd": pipeline_data["if_id"]["rd"],
                              "regDst": "", "AluOp": "", "branch": "", "ALUSrc": "", "memRead": "",
                              "memWrite": "", "regWrite": "", "memToreg": ""}
    pipeline_data["ex_mem"] = {"Aluout": "", "rd": "", "Alub": "", "writeReg": "", "branch": "",
                               "ALUSrc": "", "memRead": "", "memWrite": "", "regWrite": "", "memToreg": ""}
    pipeline_data["mem_wb"] = {"memReadp": "", "Aluout": "", "writeReg": "", "regWrite": "", "memToreg": "", }
    result_queue = Queue()
    processes = []
    current_pc = self.start_pc
    c = 0
    for i in range(len(instruction_memory)):
        c += 1
        process = Process(target=self.execute_instruction, args=(current_pc, pipeline_data, self.lock, result_queue, c))
        processes.append(process)
        #print("Previous pc: ", current_pc)
        process.start()
        current_pc = result_queue.get()
        #print("Current pc: ", current_pc)

    for process in processes:
        process.join()
    print(self.clock)
```

Function for execution time to integrate all the processes together in a pipelined fashion

Overall, we can see that the Pipelined processor takes lesser time in general than the Non-Pipelined one, based on the number of clock cycles obtained.