

```

### IMPORTATING STATION ###
from instruction_queue import * #for the timing table
from reservation_station import *
from load_station import *
from register import Registers

### IMPORTING MODULES ###
import numpy as np
import os

#####

### PARAMETERS ###
#####

memory_file_name = "memory.txt"
input_file_name = "custom_input.txt"

## Number of RS, Register entries ##
nb_add = 3
nb_mult = 2
nb_load = 6
nb_store = 6
nb_register = 11

RESVNUMCONFIG = {
    'Add': nb_add,
    'Mult': nb_mult,
    'Load': nb_load,
    'Store': nb_store
}

# clock cycle needed per type of operation ##
cpi_add = 2
cpi_sub = 2
cpi_mul = 6
cpi_div = 12
cpi_load = 3
cpi_store = 3

# Initial Register values ##
val_reg = np.zeros(nb_register)
reg_init = [6.0, 0., 3.5, 0., 10., 0., 0., 0., 7.8, 0.]

for i in range(len(reg_init)):
    val_reg[i] = reg_init[i]

## Initial pc and clock ##
clock = 0
pc = 0

## Creation of RS, Register ##
Add = Add_RS(RESVNUMCONFIG)
Mult = Mul_RS(RESVNUMCONFIG)
Load = Load_Station(RESVNUMCONFIG, memory_file_name)
Store = Store_Station(RESVNUMCONFIG, memory_file_name)
Register = Registers(nb_register, val_reg)

## MESSAGES ##

message_fp_issued = "NEXT FP INSTRUCTION TO BE ISSUED: "

```

```

#####
### MAIN FUNCTION ###
#####

def main():
    global pc
    global clock

    cdb_buffer = [] # treated as a priority queue for the commun data bus
    list_cdb=["",-1,-1]

    # Main iteration
    while not timing_table.check_everything_finished():
        #input("Press enter to simulate a clock")
        clock+=1 # clock cycle added one
        print("\n")
        print("#####")
        print("#####")
        print("\n")
        print("Clock cycle :", clock)
        print("PC :", pc, "\n")

        # Fetch instruction to reservations
        load_instruction(instructions)

        # Check for execution start of instruction in reservation
        started()

        # Load all finished instruction from the reservation station
        cdb_buffer = is_finished()
        timing_table_finished(cdb_buffer)

        # Broadcast Instruction of CDB to Register and RS
        if len(cdb_buffer)>0:
            # Broadcast instruction value of the smallest pc first
            tag_cdb, value_cdb, pc_cdb = cdb_buffer[0]
            list_cdb.append([tag_cdb, value_cdb, pc_cdb])
            # Reset the RS and Register that finished

        else:
            list_cdb.append(["",-1,-1])

        if list_cdb[-2][0] != "" and list_cdb[-2][0] !=list_cdb[-3][0]:
            # Update the timing table for Write back when the instruction is Broadcasted
            # (the one finished in the previous clock [-2])
            cdb_update(list_cdb[-2][0],list_cdb[-2][1])
            timing_table.timing_update_wb(list_cdb[-2][2], clock)
            print("Instruction '"+str(instructions[list_cdb[-2][2]])+"' BROADCASTED in CDB at c")
        # Update
        update()

        # Print the Tables
        timing_table.printList()
        Add.printList()
        Mult.printList()
        Load.printList()
        Register.printList()
        reset(list_cdb[-1][0])
    return("SIMULATION FINISHED")

```

LOAD FUNCTIONS

Load a single instruction of number pc to the corresponding RS if possible

```
def load_instruction(instructions):
```

```
    global pc
```

```
    global clock
```

If pc is bigger then the highest instruction pc no more instruction to issue

```
    if (pc >= len(instructions)):
```

```
        return True
```

Instruction has to be issued

```
    else:
```

```
        instruction = instructions[pc].split(" ") # Splitting the instruction
```

```
        type_op = instruction[0] # First arg is the operation type
```

```
        print(message_fp_issued, ""+str(instructions[pc])+"", "\n")
```

```
        if (type_op == "ADD"): # ADD
```

```
            FetchInstruction(Add,instruction,cpi_add)
```

```
        if (type_op == "SUB"): # SUB
```

```
            FetchInstruction(Add,instruction,cpi_sub)
```

```
        if (type_op == "MULT"): # MULT
```

```
            FetchInstruction(Mult,instruction,cpi_mul)
```

```
        if (type_op == "DIV"): # DIV
```

```
            FetchInstruction(Mult,instruction,cpi_div)
```

```
        if (type_op == "LD"): # LD
```

```
            Fetchload(Load,instruction,cpi_load)
```

We move to the next instruction after successfully issuing one

```
        pc += 1
```

Put the corresponding instruction to the RS entry

```
def FetchInstruction(station,instruction,cpi):
```

```
    global pc
```

```
    global clock
```

```
    type_op = instruction[0] # Type of operation
```

```
    dest = instruction[1] # Destination Register of the instruction
```

```
    reg_valueI = instruction[2] # First operand
```

```
    reg_valuek = instruction[3] # Second operand
```

Get a free Position from the corresponding station (Addition or Multiplication) in order

results[0] = -1 if no free position in RS

```
    result = station.getFreePosition()
```

Update the free RS entries when the destination Register is not Busy

```
    if result[0]>=0 and (not Register.isBusy(dest)):
```

```
        print("Instruction '"+type_op+" "+dest+" "+reg_valueI+" "+reg_valuek+" ISSUED at clock
```

```
        Register.updateRegisterTag(result[1],dest)
```

```
        operand1 = Register.getRegister(reg_valueI)
```

```
        operand2 = Register.getRegister(reg_valuek)
```

```
        station.loadInstruction(operand1[1], operand1[0], operand2[1], operand2[0], result[0], 1)
```

Update of the timing table entries after issuing

```
        timing_table.timing_update_issue(pc, clock)
```

```
        return True
```

Otherwise Stall to wait for free RS entries

```
    pc-=1 # Reduce by one to maintain pc number in load_instruction(instructions)
```

```
    return False
```

```
# Put the corresponding instruction to the RS entry same as FetchInstruction(station,instruction)
# But Less entries needed
```

```
def Fetchload(station,instruction,cpi):
    global pc
    global clock
    type_op = instruction[0]
    dest = instruction[1]
    operand = instruction[2]
    result = station.getFreePosition()
    if result[0]>=0 and (not Register.isBusy(dest)):
        Register.updateRegisterTag(result[1],dest)
        offset = extract_offset_reg(operand)[0]
        reg_value = extract_offset_reg(operand)[1]
        Load.loadInstruction(reg_value,offset, result[0], type_op, pc, cpi)
        timing_table.timing_update_issue(pc, clock)
    return True
```

```
pc-=1
return False
```

```
# Function to get value of the Memory Addresses
```

```
def extract_offset_reg(instruction_text):
    inst_split = instruction_text.replace(')', ' ').split('(')
    offset = inst_split[0]
    #print("offset : ", offset)
    reg_value = inst_split[1][1]
    #print("reg_value : ", reg_value)
    return (offset, reg_value)
```

```
### UPDATE FUNCTIONS ###
```

```
# Update the Time Left by instruction in the RS
```

```
def update():
    Add.update_clock()
    Mult.update_clock()
    Load.update_clock()
```

```
# Update timing table Execution start entries depending on the time Left of execution
```

```
# If Time Left for execution = clock needed per instruction then we know the instruction started
```

```
def started():
    for item in Add.reservation :
        if item.op == "ADDD" and item.time == cpi_add-1 and item.fuState ==1:
            timing_table.timing_update_start(item.ins_pc, clock)
            print("Instruction "+str(timing_table.instructionList[item.ins_pc].ins[0])+" START")
        if item.op == "SUBD" and item.time == cpi_sub-1 and item.fuState ==1:
            timing_table.timing_update_start(item.ins_pc, clock)
            print("Instruction "+str(timing_table.instructionList[item.ins_pc].ins[0])+" START")
    for item in Mult.reservation :
        if item.op == "MULTD" and item.time == cpi_mul-1 and item.fuState ==1:
            timing_table.timing_update_start(item.ins_pc, clock)
            print("Instruction "+str(timing_table.instructionList[item.ins_pc].ins[0])+" START")
        if item.op == "DIVD" and item.time == cpi_div-1 and item.fuState ==1:
            timing_table.timing_update_start(item.ins_pc, clock)
            print("Instruction "+str(timing_table.instructionList[item.ins_pc].ins[0])+" START")
    for item in Load.reservation :
        if item.op == "LD" and item.time == cpi_load-1 and item.fuState ==1:
            timing_table.timing_update_start(item.ins_pc, clock)
            print("Instruction "+str(timing_table.instructionList[item.ins_pc].ins[0])+" START")
```

```

# Check if any RS entries finished executing and are ready to be broadcast
def is_finished():
    list_add = Add.finish()
    list_mult = Mult.finish()
    list_load = Load.finish()
    list_finished = list_load+list_add+list_mult
    return list_finished

# Update timing table Execution finished entries depending of the is_finished() List
def timing_table_finished(list_finished):
    global clock
    for item in list_finished:
        timing_table.timing_update_finish(item[2], clock)
        print("Instruction '"+str(timing_table.instructionList[item[2]].ins[0])+"' FINISHED at (

# When a value is ready in RS => Broadcast
# Input is the tag of the operation and value
def cdb_update(tag, value):
    # Check and update RS
    Add.updateValueByTag(tag, value)
    Mult.updateValueByTag(tag, value)

    # Check and update Register
    Register.updateRegisterByTag(tag,value)

### RESET FUNCTIONS ###

# Reset the RS entries that finished execution and have been broadcasted
def reset(tag):
    if tag == "":
        return True
    tag_name = tag[:-1]
    tag_position = int(tag[-1])
    if tag_name == "Add":
        Add.reset(tag_position)
    if tag_name == "Mult":
        Mult.reset(tag_position)
    if tag_name == "Load":
        Load.reset(tag_position)

### PRINT FUNCTIONS ###

# Initial timing table with instructions inside
def initial_table(instructions):
    timing_table = Timing(instructions)
    print("=====")
    print("Clock cycle :", clock, "\n")
    timing_table.printList()
    Add.printList()
    Mult.printList()
    Load.printList()
    Register.printList()
    return timing_table

# Txt file decoder for instruction List
def input_file_decoder(in_file):
    input_file = open(in_file, 'r')
    instructions = []
    for line_not_split in input_file:
        if(line_not_split != ""):
            line_not_split = line_not_split.split("\n")[0]

```

```
        instructions.append(line_not_split.replace(",", " "))
    return instructions
```

```
#####
### PROGRAM EXECUTION ###
#####
```

```
if __name__ == '__main__':
    #input("Press Enter to Start")
    if len(input_file_name) > 1:
        instructions = input_file_decoder(input_file_name)
        print("INSTRUCTION LIST :","\n")
        print("\n".join(instructions))
        timing_table = initial_table(instructions)
        main()
    else:
        print("Please specify input file!")
        exit(1)
```