MIPS Simulator

목차

1. MIPS_ISA.py 설명

구현한 함수 및 클래스

1. Processor Class

```
# MIPS ISA Processor
class Processor:
    R_format = ["ADD", "OR"]
    I_format = ["ADDI", "ORI"]
    I_memory = ["SW", "LW"]

    def __init__(self, Instructions):
        # 4 Pipeline Register
        self.Pipelines = [IFIDRegister(), IDEXRegister(), EXMEMRegister(), MEMWBRegister()]

        # IFID Register에 Instruction을 Fetching 하기 위해 IFIDRegister 클래스로 NextInstrution 선언
        self.NextInstruction = IFIDRegister()

        # InstructionMemory Which Contains All Instructions
        self.InstructionMemory = Instructions
        self.InstructionCount = len(Instructions)

        #Forward Unit
        self.ForwardA = '00'
        self.ForwardB = '00'

        #Load Use Data Hazard Unit
        self.PCWrite = '1'
        self.IFIDWrite = '1'
        self.IDEXFlush = '0'

        self.Trial = 0
        self.ClockCount = len(Instructions) - 1 + 5
```

Pipelines는 각 Pipeline Register를 의미하며 리스트로 각 Registers를 묶어서 선언했다. Instruction Memory는 Processor 객체가 선언될 때 전달되는 Instruction들을 리스트 형식으로 저장한 것이며, Instruction Memory에서 Instruction들은 순서대로 NextInstruction에 저장되어 Fetching 된다. Clock Count는 전달된 Instruction을 병렬로 처리하는데 필요한 총 Clock Count의 수이다. Load Use Data Hazard에 의해 Clock Count와 Trial의 값이 영향을 받는다. 또한 Processor에서 지원하는 ADD, OR, ADDI, ORI, SW, LW는 각 형식에 맞추어 3가지 방법으로 분류했다. I-format에서 상수 연산과 메모리 연산을 분리한 이유는 연산의 종류에 따라서 regWrite와 memRead를 다르게 해주기 위함이다.

A. Initialize(self)

매 Clock Cycle 마다 Processor의 ForwardA와 ForwardB 신호를 00으로 초기화 하기 위해서 구현한 메소드이다.

```
def Initialize(self):
    self.ForwardA = '00'
    self.ForwardB = '00'
```

B. MUX(self, a, b, code)

코드의 Boolean 값에 따라서 Input 값 2개 중 한 개를 Return 하는 메소드이다. EXMEM Register의 registerRd 값을 정해줄 때 사용된다.

```
def MUX(self, a, b, code):
    return b if code else a
```

C. Fetch(self, IF)

IF는 Instruction Fetching의 약자이다. Fetch 메소드는 Clock Rise 시점에서 Processor의 Instruction Memory에 할당되어 있는 Instruction을 순서에 맞게 NextInstruction 속성으로 Fetch 하는 역할을 한다. Fetch를 하면서 Instruction에 해당하는 Rs, Rt, Rd 값을 지정해준다. Fetch는 3가지 경우의 수가 존재한다.

1. Instruction Memory에 있는 모든 Instruction을 다 실행했다면 IF에 None 값이 전달된다. 이때에는 Rs, Rt, Rd 모두 0 값을 가진다.
2. Instruction이 R-format의 Instruction이라면 위치에 맞도록 각각 Rs, Rt, Rd값을 지정했다.
3. Instruction이 I-format의 Instruction이라면 위치에 맞도록 각각 Rs, Rt 값을 지정했고, Rd 값은 0 값으로 두었다.

```
# Fetching Instruction
def Fetch(self, IF=None):
    # Empty Instruction For None Instruction
```

```
        if IF == None:
            self.NextInstruction.OPcode = None
            self.NextInstruction.registerRs = '0'
            self.NextInstruction.registerRt = '0'
            self.NextInstruction.registerRd = '0'
        # Instruction with R-format
        elif IF[0] in self.R_format:
            self.NextInstruction.OPcode = IF[0]
            self.NextInstruction.registerRs = IF[2]
            self.NextInstruction.registerRt = IF[3]
            self.NextInstruction.registerRd = IF[1]
        # Instruction with I-format
        elif IF[0] in self.I_memory + self.I_format:
            self.NextInstruction.OPcode = IF[0]
            self.NextInstruction.registerRs = IF[2]
            self.NextInstruction.registerRt = IF[1]
            self.NextInstruction.registerRd = '0'
```

D. ControlUnit(self, IF)

ControlUnit은 IFID 레지스터에 저장된 Instruction의 Operator에 따라서 regWrite와 memRead 값을 정해주는 역할을 한다. ControlUnit은 5가지의 경우의 수가 있다.

1. IFID에서 전달되는 Opcode가 None이라면 regWrite, memRead 모두 0 값을 가진다.
2. IFID에서 전달되는 Opcode가 R-format (ADD, OR) 이라면 regWrite 값은 1, memRead 값은 0이 된다.
3. IFID에서 전달되는 Opcode가 상수를 사용하는 I-format (ADDI, ORI) 이라면 regWrite 값은 1, memRead 값은 0이된다.
4. IFID에서 전달되는 Opcode가 LW연산이라면 regWrite 값은 1, memRead 값은 1이 된다.
5. IFID에서 전달되는 Opcode가 SW연산이라면 regWrite 값은 0, memRead 값은 0이 된다.

```
    # Deciding Control Signal in ID stage
    def ControlUnit(self, reg):
        #returning regWrtie, memRead
        if reg.OPcode == None:
            return ('0', '0')
        elif reg.OPcode in Processor.R_format:
            return ('1', '0')
```

```
        elif reg.OPcode in Processor.I_format:
            return ('1', '0')
        elif reg.OPcode == "LW":
            return ('1', '1')
        elif reg.OPcode == "SW":
            return ('0', '0')
```

E. ForwardUnit(self)

ID/EX, EX/MEM, MEM/WB의 속성 값들을 이용해서 Data Hazard를 탐지하여 Processor의 ForwardA, ForwardB 값을 바꾸어 주는 메소드이다. EXMEM의 조건을 먼저 비교한 후에 MEMWB의 조건을 비교하기 때문에 Double Data Hazard 역시 처리 가능하다.

```
    # Detecting Forward Unit
    def ForwardUnit(self):
        IDEX, EXMEM, MEMWB = self.Pipelines[1], self.Pipelines[2], self.Pipelines[3]

        # Forward A
        if EXMEM.regWrite == '1' and EXMEM.registerRd == IDEX.registerRs and EXMEM.registerRd != '0':
            self.ForwardA = "10"
        elif MEMWB.regWrite == '1' and MEMWB.registerRd == IDEX.registerRs and MEMWB.registerRd != '0':
            self.ForwardA = "01"
        else:
            self.ForwardA = "00"

        # Forward B
        if EXMEM.regWrite == '1' and EXMEM.registerRd == IDEX.registerRt and EXMEM.registerRd != '0':
            self.ForwardB = "10"
        elif MEMWB.regWrite == '1' and MEMWB.registerRd == IDEX.registerRt and MEMWB.registerRd != '0':
            self.ForwardB = "01"
        else:
            self.ForwardB = "00"
```

F. Flush(self)

전달받은 레지스터의 Rs, Rt, Rd, regWrite, memRead, RegDst 속성을 모두 0으로 초기화하고 해당 레지스터를 돌려준다.

```
# Flushing
def Flush(self, reg):
    reg.registerRs = '0'
    reg.registerRt = '0'
    reg.registerRd = '0'
    reg.regWrite = '0'
    reg.memRead = '0'
    reg.RegDst = '0'
    return reg
```

G. HazardDetectionUnit(self)

Load Use Data Hazard의 발생여부를 파악해서 Processor의 PCWrite, IFIDWrite, IDEXFlush 값을 정해주는 역할을 한다. 해당 Cycle에서 IDEX Register에 할당된 Instruction의 연산 종류가 메모리에 접근해서 값을 읽어오는 LW연산이고, 메모리상의 값이 저장되는 레지스터를 IFID에서 Source Register으로 사용할 경우에 PCWrite 와 IFIDWrite는 0, IDEXFlush는 1값을 할당한다.

```
# Detecting Load Use Data Hazard
def HazardDetectionUnit(self):
    IDEX, IFID = self.Pipelines[1], self.Pipelines[0]
    if IDEX.memRead == '1' and (IDEX.registerRt == IFID.registerRs or IDEX.registerRt == IFID.registerRt):
        return ('0', '0', '1')
    else:
        return ('1', '1', '0')
```

H. Clock(self, IF)

한 번의 Clock Cycle 동안 Processor내에서 일어나는 일들과 Pipeline Register들의 상태를 그에 맞추어 변화시켜주는 메소드이다. Clock는 실행될 때 Instruction을 파라미터로 받는데 이 Instruction이 Processor의 Fetch 메소드를 통하여 NextInstruction 속성 값으로 할당된다. 이후에 Initialize 메소드를 이용해서 Processor의 ForwardA, ForwardB 속성 값을 00으로 초기화해주고 reg이라는 변수에 호출된 시점의 Pipeline Register들을 할당한다.

이전 Clock Cycle에서 Load Use Data Hazard가 발생했다면, 다음 Clock Cycle 시점에는 IDEXFlush 값이 1일 것이고 Flush가 일어나야 한다. 따라서 IDEXFlush 값을 통해서 Load Use Data Hazard의 발생 여부를 파악하고 Flush를 해야 한다면 Flush를 진행한다.

IDEXFlush 값이 1이 아니라면 각 단계의 Pipeline Register 값들을 다음 Register에 전달해준다. 이때

IDEX Register의 RegDst 와 registerRd, registerRt값을 MUX를 이용해서 EXMEM registerRD을 할당한다. 또한 IDEX Register로 IFID Register에 저장된 Instruction의 Opcode를 이용해서 Control Signal을 정하여 넘겨준다. 마지막으로, Clock이 호출되었을 때 Fetch된 Instruction은 현재 NextInstruction에 있으므로 NextInstruction을 IFID Register로 할당하면서 Pipeline Register의 연산을 마친다.

이후에 Load Use Data Hazard와 Data Hazard의 여부를 파악하여 적절한 조치를 취한 뒤 reg를 Processor의 Pipelines 속성에 할당하면서 상태를 저장한다.

```
    # Clock Cycle
    def Clock(self, IF=None):
        self.Fetch(IF)
        self.Initialize()
        reg = self.Pipelines

        # Execute Flush
        if self.IDEXFlush == '1':
            self.PCWrite = '1'
            self.IFIDWrite = '1'
            self.IDEXFlush = '0'
            reg[3].regWrite, reg[3].memRead, reg[3].registerRd = reg[2].regWrite, reg[2].memRead, reg[2].registerRd
            reg[2].regWrite, reg[2].memRead = reg[1].regWrite, reg[1].memRead
            # Deciding RegDst
            reg[2].registerRd = self.MUX(reg[1].registerRd, reg[1].registerRt, reg[1].RegDst == "0")
            # Flushing
            reg[1] = self.Flush(reg[1])
            self.ClockCount += 1
            self.Pipelines = reg
            return

        # Send Info to Next Level Register
        reg[3].regWrite, reg[3].memRead, reg[3].registerRd = reg[2].regWrite, reg[2].memRead, reg[2].registerRd
        reg[2].regWrite, reg[2].memRead = reg[1].regWrite, reg[1].memRead
        # Deciding RegDst
        reg[2].registerRd = self.MUX(reg[1].registerRd, reg[1].registerRt, reg[1].RegDst == "0")
        # Deciding Control Unit Code
        reg[1].regWrite, reg[1].memRead = self.ControlUnit(reg[0])
        reg[1].registerRs, reg[1].registerRt, reg[1].registerRd =reg[0].registerRs, reg[0].registerRt, reg[0].registerRd
        # Deciding RegDst Control Signal
        reg[1].RegDst = '1' if reg[0].registerRd != '0' else '0'
        reg[0].OPcode, reg[0].registerRd = self.NextInstruction.OPcode, self.NextInstruction.registerRd
        reg[0].registerRs, reg[0].registerRt = self.NextInstruction.registerRs, self.NextInstruction.registerRt

        # Detecting Load Use Data Hazard
```

```
        self.PCWrite, self.IFIDWrite, self.IDEXFlush = self.HazardDetectionUnit()

        # Detection Data Hazard
        self.ForwardUnit()

        self.Pipelines = reg
        self.Trial += 1
```

2. Register Class

IDEXRegister, EXMEMRegister, MEMWBRegister가 모두 공통으로 가지는 regWrite, memRead 속성을 Register Class의 속성으로 지정한 뒤, 다른 파이프라인 레지스터들이 Register클래스를 상속받아 사용하도록 했다.

```
class Register:
    regWrite = '0'
    memRead = '0'
```

A. IFIDRegister

Opcode, registerRs, registerRt, registerRd를 속성으로 가진다. Opcode는 IDEX로 Opcode를 전달한 뒤에 Control Unit에서 regWrite와 memRead를 정해주기 위해서 저장하는 속성이다.

```
# IF/ID Register
class IFIDRegister():
    def __init__(self):
        self.OPcode = None
        self.registerRs = '0'
        self.registerRt = '0'
        self.registerRd = '0'
```

B. IDEXRegister

registerRs, registerRt, registerRd, RegDst를 속성으로 가지고 Register 클래스에서 상속받은 regWrite, memRead 속성을 추가로 가지고 있다. 우선 IFID에서 전달되는 Instruction의 OPCode를 참고해서 Control Signal을 결정한 뒤에 regWrite와 memRead에 Control Signal을 할당한다.

RegDst는 두 가지 값을 가질 수 있다.

1. R-format Instruction의 경우에는 RegDst의 속성 값이 1이다.
2. I-format Instruction의 경우에는 RegDst 속성 값이 0이다.

Clock 메소드가 실행되는 시점에서 IDEX Register에서 EXMEM Register로 값이 전달될 때 Processor의 MUX 메소드에 RegisterRd와 RegisterRt값을 전달하여 적절한 값을 EXMEM Register의 registerRd 값을 할당한다. RegDst 값이 0이면 해당 Instruction은 I-format이므로 registerRt 값을 EXMEM Register의 registerRd 값으로 정하고, RegDst가 1이면 registerRd 값을 EXMEM Register의 registerRd로 할당한다.

```
# ID/EX Register

class IDEXRegister(Register):
    def __init__(self):
        self.registerRs = '0'
        self.registerRt = '0'
        self.registerRd = '0'
        # Register Destination which would be send to EX/MEM Register
        # output of MUX(RegDst, RegisterRt)
        self.RegDst = '0'
```

C. EXMEMRegister

registerRd속성을 가지고 있고 regWrite와 memRead 속성을 상속받는다.

```
# EX/MEM Regitser
class EXMEMRegister(Register):
    def __init__(self):
        self.registerRd = '0'
```

D. MEMWBRegister

registerRd 속성을 가지고 있고 regWrite와 memRead 속성을 상속받는다.

```
# MEM/WB Register
class MEMWBRegister(Register):
    def __init__(self):
        self.registerRd = '0'
```

2. Run.py 설명

구현한 함수 및 클래스

1. ReadCode(fname)

파일 이름을 입력 받아서 해당 파일이 경로에 존재한다면 내용을 읽어서 파일 이름과 함께 반환 해주고, 파일이 존재하지 않는다면 프로그램을 종료 시킨다.

```
# Reading Input File
def ReadCode(fname):
    try:
        f = open(fname, "r")
    except IOError:
        print("File not exist. Check file name")
        exit(1)

    lines = f.readlines()
    f.close()

    return lines, fname
```

2. WriteSimulation(processor, registers)

매 Cycle마다 processor과 pipeline register들의 상태 값을 리스트 형식으로 받아서 반환한다. 이때 반환하는 리스트는 미리 선언해둔 output DataFrame에 추가한다.

```
# Writing Simulation
def WriteSimulation(processor, registers):
    CC = []
    CC.append(registers[0].registerRs)
    CC.append(registers[0].registerRt)
    CC.append(registers[0].registerRd)

    CC.append(registers[1].registerRs)
    CC.append(registers[1].registerRt)
    CC.append(registers[1].registerRd)
    CC.append(registers[1].regWrite)
    CC.append(registers[1].memRead)
    CC.append(registers[1].RegDst)
```

```
    CC.append(registers[2].registerRd)
    CC.append(registers[2].regWrite)

    CC.append(registers[3].registerRd)
    CC.append(registers[3].regWrite)

    CC.append(processor.ForwardA)
    CC.append(processor.ForwardB)
    CC.append(processor.PCWrite)
    CC.append(processor.IFIDWrite)
    CC.append(processor.IDEXFlush)

    return CC
```

실행 코드

```
from MIPS_ISA import *
import pandas as pd
import sys

# DataFrame Index
df_index = ["IF/ID.registerRs", "IF/ID.registerRt", "IF/ID.registerRd",
            "ID/EX.registerRs", "ID/EX.registerRt", "ID/EX.registerRd", "ID/EX.regWrite","ID/EX.memRead","ID/EX.RegDst",
            "EX/MEM.registerRd", "EX/MEM.regWrite",
            "MEM/WB.registerRd", "MEM/WB.regWrite",
            "ForwardA", "ForwardB", "PC.Write", "IF/ID.Write", "ID/EX.Flush"]

# Making Empty DataFrame
output = pd.DataFrame([], index=df_index)


# Loading Instructions
if len(sys.argv) == 1:
    fname = input("Enter File Name: ")
    instructions, fname = ReadCode(fname)
else:
    instructions, fname = ReadCode(sys.argv[1])
Instructions = []


for instruction in instructions:
    instruction = instruction.strip().split(" ")
```

```
    if instruction[0] in Processor.I_memory:
        offset, rt = tuple(instruction[-1].split("("))
        instruction[-1:] = [rt, offset]

    for i in range(len(instruction)):
        instruction[i] = instruction[i].strip(",$)")

    Instructions.append(instruction)

# Loading Processor
MIPS = Processor(Instructions)


# Running Assembly Code
trial = 0
while trial < MIPS.ClockCount:

    # Recording Each Clock Cycle's Status in DataFrame
    simulation = WriteSimulation(MIPS, MIPS.Pipelines)
    k = pd.Series(simulation, name="CC" + str(trial + 1), index=df_index)
    output = pd.concat([output, k], axis=1)

    # Clock Cycle
    if MIPS.InstructionCount > MIPS.Trial:
        MIPS.Clock(MIPS.InstructionMemory[MIPS.Trial])
    else:
        MIPS.Clock()

    trial += 1

# Showing Output in Console
print("Simulation Result")
print(output)

# Extracting Output DataFrame to CSV File
fname = fname.split(".")[0]
output.to_csv(fname + "_output.csv", encoding="utf-8")
```

알고리즘

1. Simulation의 결과를 저장할 DataFrame을 선언한다.
2. ReadCode 함수를 이용해서 Assembly Code를 저장한다. 읽어온 Assembly Code는 추가 정제 과정을 거쳐

다음과 같은 형식으로 저장되어 Processor에 전달된다.

예시

Before	ADD $1, $2, $5
After	[ADD, 1, 2, 5]
Before	ADDI $1, $2, 20
After	[ADDI, 1, 2, 20]
Before	LW $1, 20($2)
After	[LW, 1, 2, 20]

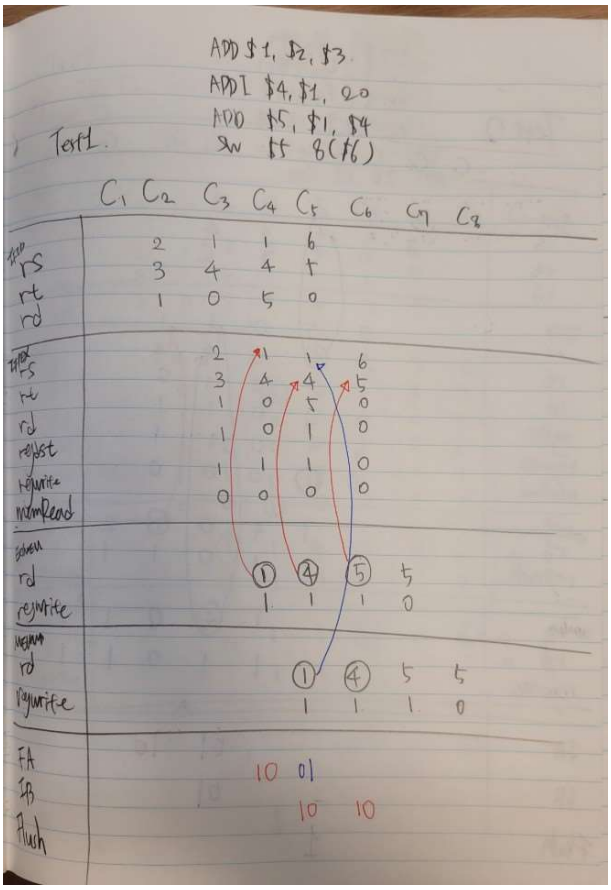
3. MIPS를 Processor의 인스턴스로 선언한다.
4. MIPS의 Instruction들에 따른 Clock Cycle 만큼동안 While문이 동작하면서 MIPS와 MIPS의 Pipeline Register들의 상태를 DataFrame에 추가한다. 이후에 MIPS의 Instruction Memory에 있는 명령어들을 차례대로 실행하는데 더 이상 실행될 명령어가 없다면 Clock 메소드의 인자로 아무것도 전달하지 않으면서 None 이 Fetch되도록 한다.
5. 시뮬레이션 결과를 콘솔창에 출력하고 csv파일로 내보낸다.

주의사항

1. 프로그램은 두가지 방법으로 실행이 가능하다. 실행 시 파일 이름을 실행 인자를 함께 전달해줄 수 있고 프로그램 실행 후에 콘솔 창에서 파일 이름을 입력해도 된다.
2. 본 프로그램은 pandas의 DataFrame을 이용하여 결과를 출력하기 때문에 pandas를 import하도록 되어있다. 프로그램 실행 시 pandas의 설치가 필요하다.
3. Test 할 File은 Run.py와 같은 경로에 위치시켜야 하며, 경로에 없는 파일 이름을 입력하면 프로그램이 종료된다.
4. 해당 파일의 결과는 같은 경로에 파일이름_output.csv로 생성된다. CSV 파일로 변환하는 과정에서 Forward A와 Forward B 값 00, 01 이 0, 1로 저장된다. 프로그램 콘솔 창에는 정상적으로 출력된다.

3. 실행 결과

3.1. Test1



Assembly Code

ADD $1, $2, $3

ADDI $4, $1, 20

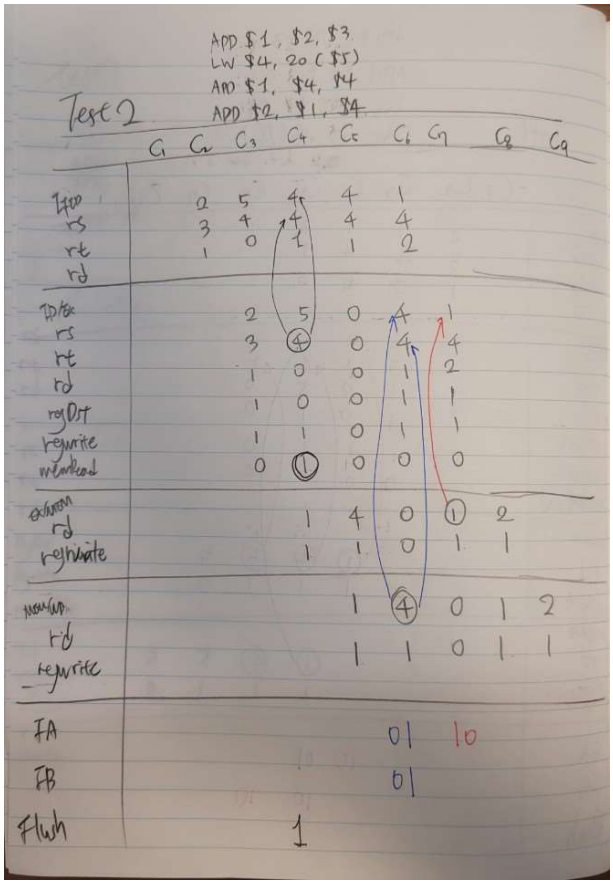
ADD $5, $1, $4

SW $5 8($6)

Output

```
Simulation Result
                CC1 CC2 CC3 CC4 CC5 CC6 CC7 CC8
IF/ID.registerRs  0   2   1   1   6   0   0   0
IF/ID.registerRt  0   3   4   4   5   0   0   0
IF/ID.registerRd  0   1   0   5   0   0   0   0
ID/EX.registerRs  0   0   2   1   1   6   0   0
ID/EX.registerRt  0   0   3   4   4   5   0   0
ID/EX.registerRd  0   0   1   0   5   0   0   0
ID/EX.regWrite    0   0   1   1   1   0   0   0
ID/EX.memRead     0   0   0   0   0   0   0   0
ID/EX.RegDst      0   0   1   0   1   0   0   0
EX/MEM.registerRd 0   0   0   1   4   5   5   0
EX/MEM.regWrite   0   0   0   1   1   1   0   0
MEM/WB.registerRd 0   0   0   0   1   4   5   5
MEM/WB.regWrite   0   0   0   0   1   1   1   0
ForwardA         00  00  00  10  01  00  00  00
ForwardB         00  00  00  00  10  10  00  00
PC.Write          1   1   1   1   1   1   1   1
IF/ID.Write       1   1   1   1   1   1   1   1
ID/EX.Flush       0   0   0   0   0   0   0   0
```

3.2. Test2



Assembly Code

ADD $1, $2, $3

LW $4, 20($5)

ADD $1, $4, $4

ADD $2, $1, $4

Output

Simulation Result

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
IF/ID.registerRs	0	2	5	4	4	1	0	0	0
IF/ID.registerRt	0	3	4	4	4	4	0	0	0
IF/ID.registerRd	0	1	0	1	1	2	0	0	0
ID/EX.registerRs	0	0	2	5	0	4	1	0	0
ID/EX.registerRt	0	0	3	4	0	4	4	0	0
ID/EX.registerRd	0	0	1	0	0	1	2	0	0
ID/EX.regWrite	0	0	1	1	0	1	1	0	0
ID/EX.memRead	0	0	0	1	0	0	0	0	0
ID/EX.RegDst	0	0	1	0	0	1	1	0	0
EX/MEM.registerRd	0	0	0	1	4	0	1	2	0
EX/MEM.regWrite	0	0	0	1	1	0	1	1	0
MEM/WB.registerRd	0	0	0	0	1	4	0	1	2
MEM/WB.regWrite	0	0	0	0	1	1	0	1	1
ForwardA	00	00	00	00	00	01	10	00	00
ForwardB	00	00	00	00	00	01	00	00	00
PC.Write	1	1	1	0	1	1	1	1	1
IF/ID.Write	1	1	1	0	1	1	1	1	1
ID/EX.Flush	0	0	0	1	0	0	0	0	0

3.3. Test3

Assembly Code

ADD $1, $2, $3

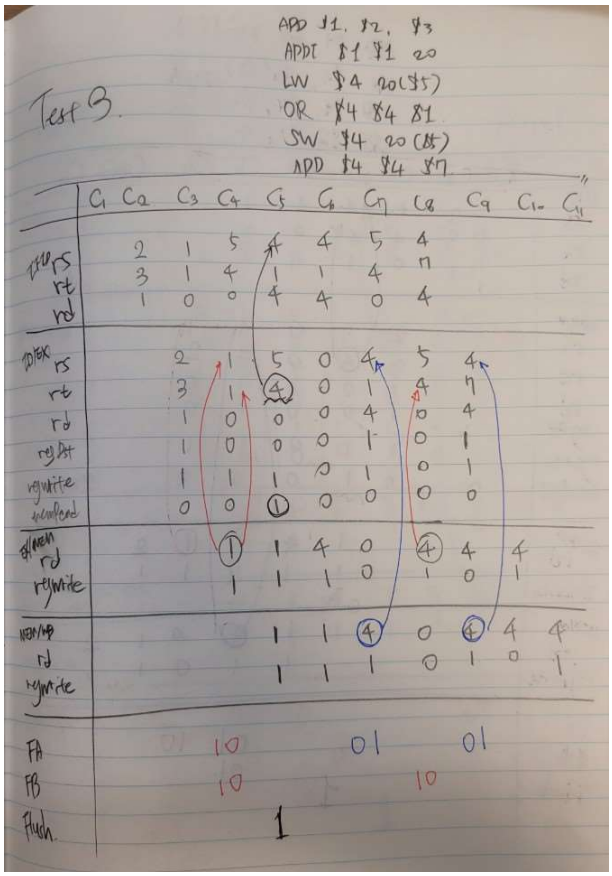
ADDI $1, $1, 20

LW $4 20($5)

OR $4, $4, $1

SW $4, 20($5)

ADD $4, $4, $7



Output

```
Simulation Result
                  CC1 CC2 CC3 CC4 CC5 CC6 CC7 CC8 CC9 CC10 CC11
IF/ID.registerRs    0   2   1   5   4   4   5   4   0    0    0
IF/ID.registerRt    0   3   1   4   1   1   4   7   0    0    0
IF/ID.registerRd    0   1   0   0   4   4   0   4   0    0    0
ID/EX.registerRs    0   0   2   1   5   0   4   5   4    0    0
ID/EX.registerRt    0   0   3   1   4   0   1   4   7    0    0
ID/EX.registerRd    0   0   1   0   0   0   4   0   4    0    0
ID/EX.regWrite      0   0   1   1   1   0   1   0   1    0    0
ID/EX.memRead       0   0   0   0   1   0   0   0   0    0    0
ID/EX.RegDst        0   0   1   0   0   0   1   0   1    0    0
EX/MEM.registerRd   0   0   0   1   1   4   0   4   4    4    0
EX/MEM.regWrite     0   0   0   1   1   1   0   1   0    1    0
MEM/WB.registerRd   0   0   0   0   1   1   4   0   4    4    4
MEM/WB.regWrite     0   0   0   0   1   1   1   0   1    0    1
ForwardA           00  00  00  10  00  00  01  00  01   00   00
ForwardB           00  00  00  10  00  00  00  10  00   00   00
PC.Write            1   1   1   1   0   1   1   1   1    1    1
IF/ID.Write         1   1   1   1   0   1   1   1   1    1    1
ID/EX.Flush         0   0   0   0   1   0   0   0   0    0    0
```

3.4. Test4

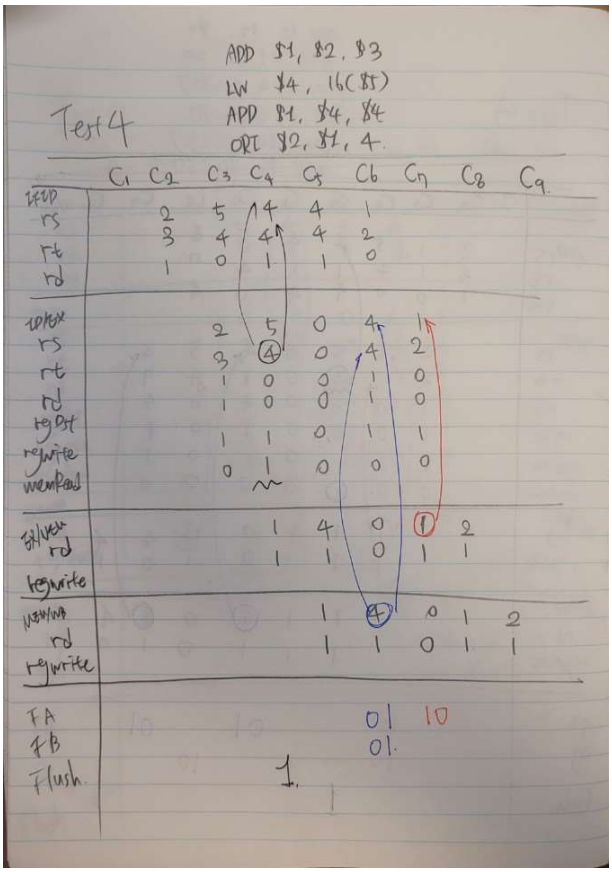
Assembly Code

ADD $1, $2, $3

LW $4, 16($5)

ADD $1, $4, $4

ORI $2, $1, 4

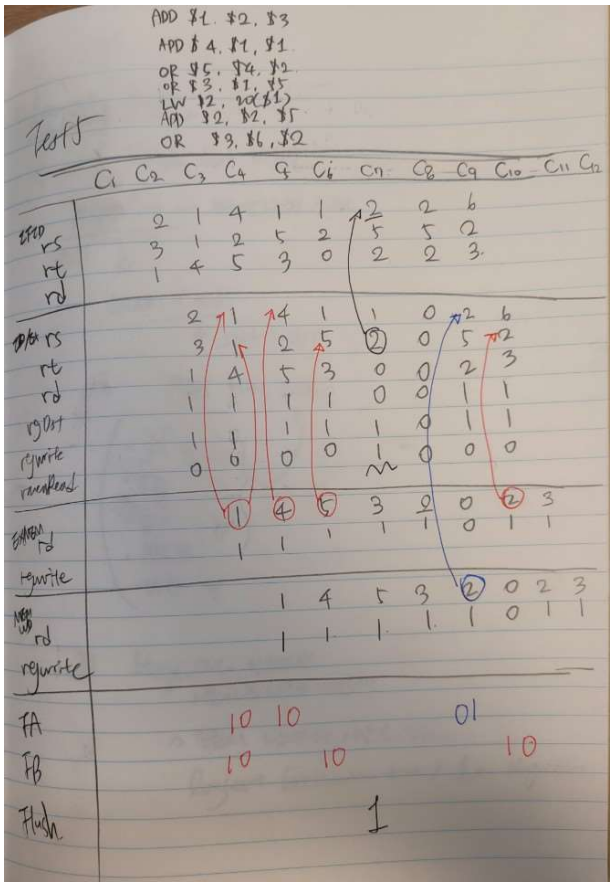


Output

Simulation Result

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
IF/ID.registerRs	0	2	5	4	4	1	0	0	0
IF/ID.registerRt	0	3	4	4	4	2	0	0	0
IF/ID.registerRd	0	1	0	1	1	0	0	0	0
ID/EX.registerRs	0	0	2	5	0	4	1	0	0
ID/EX.registerRt	0	0	3	4	0	4	2	0	0
ID/EX.registerRd	0	0	1	0	0	1	0	0	0
ID/EX.regWrite	0	0	1	1	0	1	1	0	0
ID/EX.memRead	0	0	0	1	0	0	0	0	0
ID/EX.RegDst	0	0	1	0	0	1	0	0	0
EX/MEM.registerRd	0	0	0	1	4	0	1	2	0
EX/MEM.regWrite	0	0	0	1	1	0	1	1	0
MEM/WB.registerRd	0	0	0	0	1	4	0	1	2
MEM/WB.regWrite	0	0	0	0	1	1	0	1	1
ForwardA	00	00	00	00	00	01	10	00	00
ForwardB	00	00	00	00	00	01	00	00	00
PC.Write	1	1	1	0	1	1	1	1	1
IF/ID.Write	1	1	1	0	1	1	1	1	1
ID/EX.Flush	0	0	0	1	0	0	0	0	0

3.5. Test5



Assembly Code

ADD $1, $2, $3

ADD $4, $1, $1

OR $5, $4, $2

OR $3, $1, $5

LW $2, 20($1)

ADD $2, $2, $5

OR $3, $6, $2

Output

Simulation Result

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11	CC12
IF/ID.registerRs	0	2	1	4	1	1	2	2	6	0	0	0
IF/ID.registerRt	0	3	1	2	5	2	5	5	2	0	0	0
IF/ID.registerRd	0	1	4	5	3	0	2	2	3	0	0	0
ID/EX.registerRs	0	0	2	1	4	1	1	0	2	6	0	0
ID/EX.registerRt	0	0	3	1	2	5	2	0	5	2	0	0
ID/EX.registerRd	0	0	1	4	5	3	0	0	2	3	0	0
ID/EX.regWrite	0	0	1	1	1	1	1	0	1	1	0	0
ID/EX.memRead	0	0	0	0	0	0	1	0	0	0	0	0
ID/EX.RegDst	0	0	1	1	1	1	0	0	1	1	0	0
EX/MEM.registerRd	0	0	0	1	4	5	3	2	0	2	3	0
EX/MEM.regWrite	0	0	0	1	1	1	1	1	0	1	1	0
MEM/WB.registerRd	0	0	0	0	1	4	5	3	2	0	2	3
MEM/WB.regWrite	0	0	0	0	1	1	1	1	1	0	1	1
ForwardA	00	00	00	10	10	00	00	00	01	00	00	00
ForwardB	00	00	00	10	00	10	00	00	00	10	00	00
PC.Write	1	1	1	1	1	1	0	1	1	1	1	1
IF/ID.Write	1	1	1	1	1	1	0	1	1	1	1	1
ID/EX.Flush	0	0	0	0	0	0	1	0	0	0	0	0