Project 3: Simulation Pipelined Execution

Submitter: Jiwon Jang (202211167)

Contents Table

---

1. Compile and Execution

언어는 C++를 사용하였고, 컴파일러는 g++ 9.5.0을 이용하였다. 다음 명령어를 치면 컴파일할 수 있다.

```
g++ -o homework3 homework3.cpp
```

이후 파일 실행은 다음과 같이 진행했다. ./homework3 sample.o 순서로 입력하면, 프로그램이 binary file 읽어와 모든 인스트럭션이 수행되고난 후의 결과를 출력을 해준다. 이 외에도 -m, -n, -d, -p, -atp, -antp 옵션을 입력하면 assignment에서 요구한 출력이 나온다.

-m begin_memory:end_memory: begin_memory 부터 end_memory에 담겨있는 내용 출력

-n num_of_instruction: num_of_instructruction 만큼의 instruction 수행 후 레지스터 값 출력

-d: 매 사이클 마다 레지스터 값 출력

-p: 매 사이클 마다 각 파이프라인 단계에서 실행되고 있는 instruction의 PC 값 출력

ex. {0x400000|0x400004||||}

-atp: always taken 분기 예측기 사용

-antp: always not taken 분기 예측기 사용

<주의 사항>

-n 0 option과 -d option이 함께오면 아무것도 출력되지 않는다.

-n num option을 입력하면 WB stage가 num개 만큼 실행되었을 때의 결과를 출력한다

ex. -n 3: {7|6|5|4|3} *(3번째 instruction이 WB에 왔을 때 종료)*

-n option을 사용하지 않으면 completion cycle을 출력한다. 이 때 completion 조건은 {|||||}, 즉 파이프 라인이 다 비었을 때 이다.

-m option과 -d option이 함께오면 매 출력마다 메모리가 출력된다.

-p option 사용시 stall된 instruction은 빈 칸 으로 표현된다.

2. Definition of Class and Function

이번 과제에서 구현한 클래스와 함수에 대한 설명이다. 코드의 흐름을 설명하기 위한 배경 설명이다.

<library>

총 5개의 library를 이용하였다.

iostream: 표준 입출력을 지원한다.

fstream: 파일 생성 및 read/write를 지원한다.

string: 문자열 형식을 지원한다.

vector: vector 자료형을 지원한다.

bitset: 이진수 변환을 지원한다.

<class>

class는 Memory(instruction, data), Register(register, state reg), Program Counter를 관리하기 위해서 사용하였다.

1) PROGRAMCOUNTER class definition

private:

int program_counter: PC값을 담고있다.

vector<DATA> data_list*: 해당 라벨에 속해 있는 data들이 담겨있는 리스트이다.

string line: 해당 라벨이 속해있는 '줄'이 담겨있다.

public:

PROGRAMCOUNTER(): PRGRAMCOUNTER 클래스의 constructor이다. constructor에서는 initializing 진행한다. PC를 0x400000로 초기화 하고, PC_pointer 값을 해당 instance로 초기화 한다.

static PROGRAMCOUNTER PCPointer*: 모든 함수에서 PC값에 접근할 수 있게 하기 위한 static pointer 변수이다.

void UpdateProgramCounter(int update_address): PC값을 업데이트 해주는 함수이다.

int GetPC(): PC값을 가져오는 함수이다.

2) REGISTER class definition

private:

int register_num: 레지스터 번호가 담겨있다.

int data: 레지스터에 담겨있는 데이터가 담겨있다.

public:

REGISTER(int register_num): REGISTER 클래스의 constructor이다. constructor에서는 레지스터 번호를 초기화 하고, register_list에 해당 instance를 추가해 준다.

static vector<REGISTER>register_list*: RGISTER class의 instance들이 담겨있는 리스트이다.

int ReadRegNum(): 레지스터 번호를 읽기 위한 함수이다.

void UpdateData(int update_data): 레지스터 값을 업데이트하기 위한 함수이다.

int ReadData(): 레지스터 데이터를 읽기 위한 함수이다.

3) STATE_REGISTER class definition

private:

int register_num: state 레지스터 번호가 담겨있다.

int data: state 레지스터에 담겨있는 데이터가 담겨있다.

public:

STATE_REGISTER(int register_num): STATE_REGISTER 클래스의 constructor이다. constructor에서는 state 레지스터 번호를 초기화 하고, register_list에 해당 instance를 추가해 준다.

*static vector< STATE_REGISTER *>register_list*: *STATE_REGISTER* class의 instance들이 담겨있는 리스트이다.

int ReadRegNum(): 레지스터 번호를 읽기 위한 함수이다.

void UpdateData(int update_data): 레지스터 값을 업데이트하기 위한 함수이다.

int ReadData(): 레지스터 데이터를 읽기 위한 함수이다.

3) TEXT_MEMORY class definition

private:

string line: Text가 담겨있다.

int address: Text의 주소가 담겨있다.

public:

TEXT_MEMORY(string line, int address): TEXT_MEMORY 클래스의 constructor이다. constructor에서는 line과 address를 초기화 해주고, text_memory_list에 해당 instance를 추가해 준다.

static vector<TEXT_MEMORY>text_memory_list*: TEXT_MEMORY class의 instance들이 담겨있는 리스트이다.

string ReadData(): line(Text)을 읽기 위한 함수이다.

int ReadAddress(): line(Text)가 담겨있는 주소를 읽기 위한 함수이다.

void UpdateData(string update_data): line(Text)을 업데이트 하기 위한 함수이다.

3) DATA_MEMORY class definition

private:

string line: Data가 담겨있다.

int address: Data의 주소가 담겨있다.

public:

DATA_MEMORY(string line, int address): DATA_MEMORY 클래스의 constructor이다. constructor에서는 line과 address를 초기화 해주고, data_memory_list에 해당 instance를 추가해 준다.

static vector<DATA_MEMORY>data_memory_list*: DATA_MEMORY class의 instance들이 담겨있는 리스트이다.

string ReadData(): line(Data)을 읽기 위한 함수이다.

int ReadAddress(): line(Data)가 담겨있는 주소를 읽기 위한 함수이다.

void UpdateData(string update_data): line(Data)을 업데이트 하기 위한 함수이다.

<other functions>

main 함수를 제외한 함수의 종류는 크게 일곱 가지로 구분할 수 있다.

- 진수 변환을 하는 함수들: *Hexadecimaler(string num), Binaryer(int num, int size)*
- 출력을 위한 함수들: *PrintRgisterData(), PrintMemoryContents(int begin_address, int end_address), PrintPipeLineContents(int a, int b, int c, int d, int e)*
- 산술 연산을 함수: *ALU()*
- 레지스터와 메모리에 접근해 데이터를 읽고, 쓰는 함수들: *int ReadReg(int RegNum), void WriteReg(int RegNum, int Data), string ReadMem(int Address), void WriteMem(int Address, string Data), int ReadStateReg(int RegNum), void WriteStateReg(int RegNum, int Data)*
- 각 instruction을 관리하는 함수들: int R_format(int int_line), int Instruction(string line)
- Pipeline의 각 stage를 담당하는 함수들: *int IF_stage(string line), vector<int> ID_stage(int 32bit_line, int atp), void EX_stage(vector<int> control_signal), void MEM_stage(vector<int> control_signal, int atp), void WB_stage(vector<int> control_signal)*
- Emulator 함수: *void Emulator(ifstream& input, int d, int n, int p, int atp, int num_of_instruction, int begin_address_int, int end_address_int, int m)*

이를 조금 더 자세히 알아보자.

1) 진수 변환을 하는 함수들

Hexadecimaler(string num): 2진수를 16진수로 변환하는 함수이다. 이 때 16진수의 자리를 고정하지 않는다.

Binaryer(int num, int size): 10진수를 2진수로 변환하는 함수이다. bitset 라이브러리를 사용하였다.

2) 출력을 위한 함수들

PrintRgisterData(): 레지스터를 읽어와 레지스터의 정보를 assignment가 요구하는 출력 형식으로 출력하는 함수이다.

PrintMemoryContents(int begin_address, int end_address): 메모리를 읽어와 메모리의 정보를 assignment가 요구하는 출력 형식으로 출력하는 함수이다.

PrintPipeLineContents(int a, int b, int c, int d, int e): 각 파이프라인 스테이지에 들어있는 instruction의 PC값을 assignment가 요구하는 출력 형식으로 출력하는 함수이다.

3) 산술 연산을 함수

ALU(): ALU control(ex. 0000: AND, 0001: OR … etc …)과 인자 두 개를 받아서 ALU 연산을 하는 함수이다.

4) 레지스터와 메모리에 접근해 데이터를 읽고, 쓰는 함수들

int ReadReg(int RegNum): 레지스터의 값을 읽는 함수이다.

void WriteReg(int RegNum, int Data): 레지스터의 값을 업데이트하는 함수이다.

string ReadMem(int Address): 메모리의 값을 읽는 함수이다.

void WriteMem(int Address, string Data): 메모리의 값을 업데이트하는 함수이다.

int ReadStateReg(int RegNum): 레지스터의 값을 읽는 함수이다.

void WriteStateReg(int RegNum, int Data): 레지스터의 값을 업데이트하는 함수이다.

6) 각 instruction을 관리하는 함수들

int R_format(int int_line): function field와 state reg들을 바탕으로 instruction들을 처리하는 함수이다.

int Instruction(string line): opcode와 state reg들을 바탕으로 instruction들을 처리하는 함수이다.

6) Emulator 함수

void Emulator(ifstream& input, int d, int n, int p, int atp, int num_of_instruction, int begin_address_int, int end_address_int, int m): assignment 3의 요구에 맞게 파이프라이닝을 하여 병렬적으로 instruction을 실행시키고, 시스템(reg, mem)을 관리하는 함수이다.

다음은 Emulator 함수의 일부이다.

```
       control_signal_WB = control_signal_MEM;

            if (pipelining[4] != 0) { // ==0 인 경우는 noop 가 들어온 상황
(= stall)
                WB_stage(control_signal_WB);
                pipelining_finish[4] = 1;
                instruction_counter++;
            }
            control_signal_MEM = control_signal_EX;
```

```
            if (pipelining[3] != 0) { // ==0 인 경우는 noop 가 들어온 상황
(= stall)
                MEM_stage(control_signal_MEM, atp);
                pipelining_finish[3] = 1;
            }
            control_signal_EX = control_signal_ID;
            if (pipelining[2] != 0) { // ==0 인 경우는 noop 가 들어온 상황
(= stall)
                EX_stage(control_signal_EX);
                pipelining_finish[2] = 1;
            }
            if (pipelining[1] != 0) { // ==0 인 경우는 noop 가 들어온 상황
(= stall)
                control_signal_ID = ID_stage(ReadStateReg(0), atp);
                pipelining_finish[1] = 1;
            }
            if (inst_end_checker == 0) { // instruction 메모리 끝나면
동작 X -> 그 전까지만 동작
                IF_stage(TEXT_MEMORY::text_memory_list[inst]-
>ReadData());
                // 파이프 라이닝 벡터 첫 번째 원소 이거 주소로 바꾸기
                pipelining_finish[0] = 1;
             }
```

위 코드 처럼 pipelining Emulator가 실행되면 각 instruction은 순차적으로 다섯 개의 stage를 거치게 된다. 이 때 stall(bubble, 또는 flush)인 경우에는 pipelining vector에 0이 들어가게 되고, 따라서 해당 pipeline stage를 실행하지 않는다.

```
if (pipelining_finish[0] == 0 && pipelining_finish[1] == 0 &&
pipelining_finish[2] == 0 && pipelining_finish[3] == 0 &&
pipelining_finish[4] == 0)
```

그리고 모든 stage가 비게 된다면 Emulator가 종료된다.

<main function>

main 함수는 input 값을 해석하여, 해석 결과(n, m, d ,p, option의 여부, 및 각 option에서의 정보)를 Emulator 함수에 전달하는 역할을 한다.

3. Code idea

assignment에 사용한 몇 가지 코딩 아이디어이다.

1. State Reg의 활용

   이번 assignment에서는 총 24개의 state register를 활용하였다. 다음 표는 각 state reg가 어떤 역할

을 하는 지에 대한 설명이 담긴 표이다. (가독성을 위해 stage 별로 정렬하였다. 코드에서는 필요한 stage reg를 필요한 시점에서 만들어 사용했기 때문에 stage 순서대로 정렬이 되어 있지 않다)

State Reg	Description
IF_ID.Instr	Fetching 한 instruction을 Decoding 단계로 전달하기 위해 사용하는 레지스터: *ID 단계에서는 이 32bit를 instruction을 전송 받아서 디코딩한다.*
IF_ID.NPC	Fetching 단계의 PC값을 Decoding 단계로 전달하기 위해 사용하는 레지스터: *이 PC값은 bne, beq target 연산 등에 사용된다.*
ID_EX.NPC	Decoding 단계의 PC값을 Execution 단계로 전달하기 위해 사용하는 레지스터: *이 PC값은 bne, beq target 연산 등에 사용된다.*
ID_EX.Opcode	Decoding 단계에서 해석한 opecode를 Execution 단계로 전달하기 위해 사용하는 레지스터: *이 값 바탕으로 Execution 단계에서 각 instruction 별 연산을 수행한다.*
ID_EX.rs	Decoding 단계에서 해석한 rs에 들어있는 값을 Execution 단계로 전달하기 위해 사용하는 레지스터: *이 값 바탕으로 Execution 단계에서 각 instruction 별 연산을 수행한다.*
ID_EX.rt	Decoding 단계에서 해석한 rt에 들어있는 값을 Execution 단계로 전달하기 위해 사용하는 레지스터: *이 값 바탕으로 Execution 단계에서 각 instruction 별 연산을 수행한다*
ID_EX.IMM	Decoding 단계에서 해석한 imm, target, .. 등의 값을 Execution 단계로 전달하기 위해 사용하는 레지스터: *이 값 바탕으로 Execution 단계에서 각 instruction 별 연산을 수행한다*
ID_EX.rd	Decoding 단계에서 해석한 rd 번호를 Execution 단계로 전달하기 위해 사용하는 레지스터: *이 값 바탕으로 Execution 단계에서 각 instruction 별 연산을 수행한다*
ID_EX.shamt	Decoding 단계에서 해석한 shamt를 Execution 단계로 전달하기 위해 사용하는 레지스터: *이 값 바탕으로 Execution 단계에서 각 instruction 별 연산을 수행한다*
ID.EX.function_field	Decoding 단계에서 해석한 function_field를 Execution 단계로 전달하기 위해 사용하는 레지스터: *이 값 바탕으로 Execution 단계에서 각 instruction 별 연산을 수행한다*
ID.EX.WB	Decoding 단계에서 해석한 WB를 Execution 단계로 전달하기 위해 사용하는 레지스터: *이 값은 WB까지 forwarding되고, 이를 바탕으로 WB 단계에서 어느 레지스터에 write할 지 결정한다.*
ID.EX.rs_num	Decoding 단계에서 해석한 rs 번호를 Execution 단계로 전달하기 위해 사용하는 레지스터: *이 값 바탕으로 Hazard를 체크한다.*
ID.EX.rt_num	Decoding 단계에서 해석한 rt 번호를 Execution 단계로 전달하기 위해 사용하는 레지스터: *이 값 바탕으로 Hazard를 체크한다.*
EX_MEM.ALU_OUT	Execution 단계에서 얻은 결과를 MEM 단계로 전달하기 위해 사용하는 레지스터: *이 값 바탕으로 MEM 단계 연산을 수행하거나, forwarding한다.*
EX_MEM.BR_TARGET	Execution 단계에서 얻은 Target을 MEM 단계로 전달하기 위해 사용하는 레지스터: *MEM 단계 에서 atp, antp의 T/F 여부에 따라 이 값으로 분기한다. (T면 분기)*
EX.MEM.IMM	Execution 단계로 forwarding된 IMM을 다시 MEM 단계로 전달하기 위해 사용하는 레지스터: *MEM 단계 에서 sw, sb 연산을 수행하기 위해 사용된다.*
EX.MEM.rt	Execution 단계로 forwarding된 rt에 들어있는 값을 다시 MEM 단계로 전달하기 위해 사용하는 레지스터: *MEM 단계 에서 sw, sb 연산을 수행하기 위해 사용된다.*
EX.MEM.rs_num	Execution 단계로 forwarding된 rs 번호를 다시 MEM 단계로 전달하기 위해 사용하는 레지스터: *이 값 바탕으로 Hazard를 체크한다.*
EX.MEM.rt_num	Execution 단계로 forwarding된 rt 번호를 다시 MEM 단

	계로 전달하기 위해 사용하는 레지스터: *이 값 바탕으로 Hazard를 체크한다.*
EX.MEM.WB	Execution 단계로 forwarding된 WB를 다시 MEM 단계로 전달하기 위해 사용하는 레지스터: *이 값은 WB까지 forwarding되고, 이를 바탕으로 WB 단계에서 어느 레지스터에 write할 지 결정한다.*
EX.MEM.NPC:	Execution 단계의 PC값 다시 MEM 단계로 전달하기 위해 사용하는 레지스터: *이 PC값은 bne, beq target 연산 등에 사용된다.*
MEM_WB.ALU_OUT	Mem 단계로 forwarding된 연산 결과를 다시 WB 단계로 전달하기 위해 사용하는 레지스터: *이 값 바탕으로 WB 단계에서 reg write를 한다.*
MEM_WB.MEM_OUT	Mem 단계에서 얻은 결과를 WB 단계로 전달하기 위해 사용하는 레지스터: *이 값 바탕으로 WB 단계에서 reg write를 한다*
MEM_WB.WB	Mem 단계로 forwarding된 WB를 다시 WB 단계로 전달하기 위해 사용하는 레지스터: *이 값은 어느 레지스터에 write할 지 결정한다.*

위 표를 보면 ID_EX.rs, ID.EX.rt도 있고, ID.EX.rs_num, ID.EX.rt_num도 있는 것을 확인할 수 있다.

: reg num은 Hazard를 판단하기 위해, reg value는 forwarding을 위해서 사용하였다. forwarding을 하면 reg value state reg에 값이 전달되게 되고, 이를 바탕으로 EX 단계에서 업데이트 된 값으로 연산을 수행한다.

2. Control signal

이번 과제에서는 총 10개의 control signal을 사용하였다. 각 control signal의 역할은 밑 표와 같다.

(-1은 X 표시와 같은 역할을 한다)

	RegDst	ALUSrc	Brch	MemRead	MemWrite	RegWrite	MemToReg	PCSrc	WordByteCheck	BeqBneChecker
R format (jr X)	1	0	0	0	0	1	0	0	-1	-1
R format (jr)	1	0	0	0	0	0	0	0	-1	-1
sw	-1	1	0	0	1	0	-1	0	1	-1
sb	-1	1	0	0	1	0	-1	0	0	-1
lw	0	1	0	1	0	1	0	0	1	-1
lb	0	1	0	1	0	1	0	0	0	-1
beq	-1	0	1	0	0	0	-1	1	-1	1
bne	-1	0	1	0	0	0	-1	1	-1	0
j	-1	0	0	0	0	0	0	1	-1	-1
jal	-1	0	0	0	0	0	0	1	-1	-1
else	0	0	0	0	0	1	0	0	0	-1

이 때 새롭게 추가한 WordByteCheck는 lw, sw, 그리고 lb, sb를 구분하기 위한 신호이고, BeqBneChecker는 Beq와 Bne를 구분하기 위한 신호이다.

3. Hazard 처리

A. Data Harzard

EX/MEM to EX forwarding: MEM단계에서 사용하는 reg를 EX단계의 instruction에서 사용한

다면, stall 후 forwarding해준다.

```
    // DataHazard: EX/MEM to EX Forwarding
        if (control_signal[5] == 1) {
            if (ReadStateReg(16) == ReadStateReg(20)) { // 16:
    EX_MEM.WB, 4: ID_EX.rs & WB 일 때
                // WB 될 rt, 또는 rd 가 rs 와 같다면 -> ALU 값을
    forwarding
                WriteStateReg(4, ReadStateReg(11));
                // Forwarding
            }
            if (ReadStateReg(16) == ReadStateReg(21)) { // 16:
    EX_MEM.WB, 5: ID_EX.rt & WB 일 때
                // WB 될 rt, 또는 rd 가 rt 와 같다면
                WriteStateReg(5, ReadStateReg(11));
                // Forwarding
            }
        }
```

MEM/WB to EX forwarding: WB단계에서 사용하는 reg를 EX단계의 instruction에서 사용한다면, stall 후 forwarding해준다.

```
// DataHazard: MEM/WB to EX Forwarding
    if(control_signal[5] == 1) {
        if (ReadStateReg(17) == ReadStateReg(20)) { // 17:
MEM_WB.WB, WB 일 때
            if (ReadStateReg(16) != ReadStateReg(20)) { // 20 은 굳이
forwarding 될 필요 없는 레지스터 -> state reg 는 다음 cycle 에 값 바뀌는
걸 보존 하기 위한 레지이기 때문!
                WriteStateReg(4, ReadStateReg(13));
                // Forwarding
            }
        }
        if (ReadStateReg(17) == ReadStateReg(21)) { // 17:
MEM_WB.WB, WB 일 때
            if (ReadStateReg(16) != ReadStateReg(21)) {
                WriteStateReg(5, ReadStateReg(13));
                // Forwarding
            }
        }
    }
```

MEM/WB to MEM forwarding (only lw, lb to sw, sb): load 뒤에 store가 오는 경우 forwarding을 해준다.

```
if(control_signal[3] == 1) {
        if (ReadStateReg(17) == ReadStateReg(23)) { // rt 같을 때
            WriteStateReg(19, ReadStateReg(14));
```

```
            // Forwarding
        }
    }
```

Load use stall: load에서 사용하는 reg를 그 다음 instruction에서 사용한다면, stall 후 forwarding해준다. 단 PC.write, ID/IF.write는 유지하여 stall이후에 이 전의 instruction이 재시작될 수 있게 한다.

ex. {inst4| inst3|lw| inst2| inst1} … stall … {inst4| inst3|| lw| inst2}

```
// DataHazard: load use
    if (ReadStateReg(3) == 0x23 || ReadStateReg(3) == 0x20) { // Ex
stage load 명령어가 있을 때
        if ((((line_32bit>>26)&0x3F) != 0x2b) &&
(((line_32bit>>26)&0x3F) != 0x28)) {
            if ((ReadStateReg(20) == ((line_32bit>>21)&0x1F)) ||
(ReadStateReg(21) == ((line_32bit>>16)&0x1F))) { // load use 의 rt 와
ID stage 명령의 rs, 또는 rt 가 같다면
                pipelining_control[1] = 0; // flush => 왜 1 index 를
nope 하는지는 '파이프라인 미루기' 부분 참고
                Load_use_Hazard = 1; // IF stage 유지 신호
                Load_use_Hazard_forwarding = 1; // forwarding 신호
                Load_use_Hazard_pipe = 1;
                line_buffer = line_32bit; // ID stage 유지
            }
        }
    }
```

B. Control Hazard

conditional branch

(ID stage): atp, antp에 따라 Always Taken, Always not Taken한다. 만약 Always Taken이 라면 한 사이클 stall 해준다.

ex.

atp: {|brch|inst3|inst2|inst1}

antp: {inst4| brch|inst3|inst2|inst1}

```
// Control Hazard: Brch
    if ((((line_32bit>>26)&0x3F) == 4) || (((line_32bit>>26)&0x3F)
== 5)) {
        int offset = (line_32bit)&0xFFFF;
        if (atp == 1) { // Always Taken
            pipelining[0] = 0; // 1 cycle stall
            PC_buffer = ReadStateReg(1)+4+offset*4;
        }
        else { // Always not Taken
            // 그냥 진행
```

```
        }
    }
```

(MEM stage): 예측 결과가 맞다면 그대로 진행, 만약 틀리다면 세 사이클 stall 후 PC_buffer에 있는 instruction을 실행 시킨다.

ex.

atp(true): {inst6|inst5| |brch|inst3}

atp(false): {| | |brch|inst3}

antp(true): {inst6|inst5|inst4|brch|inst3}

antp(false): {| | |brch|inst3}

```
if (control_signal[2] == 1) {
        // 이 때는 conditional branch (=beq, bne)
        if (control_signal[9] == 1) { // beq
            if (ReadReg(ReadStateReg(22)) ==
ReadReg(ReadStateReg(23))) { // 분기 O
                if (atp == 1) { // 맞춤
                    // 굿
                }
                else { // 틀림
                    pipelining[0] = 0;
                    pipelining[1] = 0;
                    pipelining[2] = 0;
                    PC_buffer = ReadStateReg(11);
                }
            }
            else { // 분기 X
                if (atp == 1) { // 틀림
                    pipelining[0] = 0;
                    pipelining[1] = 0;
                    pipelining[2] = 0;
                    PC_buffer = ReadStateReg(24)+4;
                }
                else {
                    // 굿
                }
            }
        }
        if (control_signal[9] == 0) { // bne
            if (ReadReg(ReadStateReg(22)) ==
ReadReg(ReadStateReg(23))) { // 분기 X
                if (atp == 1) { // 틀림
                    pipelining[0] = 0;
                    pipelining[1] = 0;
                    pipelining[2] = 0;
```

```
                    PC_buffer = ReadStateReg(24)+4;
                }
                else {
                    // 굿
                }
            }
            else { // 분기 O
                if (atp == 1) { // 맞춤
                    // 굿
                }
                else { // 틀림
                    pipelining[0] = 0;
                    pipelining[1] = 0;
                    pipelining[2] = 0;
                    PC_buffer = ReadStateReg(11);
                }
            }
        }
    }
```

unconditional branch: 한 사이클 stall 후, PC_buffer에 있는 instruction을 실행 시킨다.

ex. {inst4|j|inst3|inst2|inst1}

```
// Control Hazard: j
    // bit 연산자 괄호 주의하기
    if ((((line_32bit>>26)&0x3F) == 2) || (((line_32bit>>26)&0x3F)
== 3) || ((((line_32bit>>26)&0x3F) == 0) && (((line_32bit)&0x3F) ==
8))) { // Id stage 에 jump 명령어가 있을 때
        pipelining[0] = 0; // flush
        if (((line_32bit>>26)&0x3F) == 2) { // j
            PC_buffer = ((line_32bit)&0x3FFFFFF)*4;
        }
        if ((((line_32bit>>26)&0x3F) == 0) && (((line_32bit)&0x3F)
== 8)) {// jr
            PC_buffer = ReadReg((line_32bit>>21)&0x1F);
        }
        if (((line_32bit>>26)&0x3F) == 3) { // jal
            PC_buffer = ((line_32bit)&0x3FFFFFF)*4;
            WriteStateReg(15, 31); // 31 번에 기록
        }
    }
```

C. Structural Hazard

: 이번 과제에서 Structural Hazard는 없다고 가정한다.

4. Code flow

코드는

1) main 함수를 통한 파일 read 및 input 해석

2) Emulator 함수를 통한 input 요구에 맞는 실행 (ex. -n 옵션이 있다면 -n 옵션 뒤에 오는 숫자 만큼 실행)

3) Emulator 함수를 통해 각 stage 실행

5) PC 값 업데이트 (만약 brch, 또는 j라면 PC_buffer 값 읽어서 업데이트, 아니라면 PC+4로 업데이트)

6) Emulator 함수를 통한 input 요구에 맞는 출력 제공 (-d 옵션이 있다면 매 instruction마다 출력, 없다면 대기)

7) 다시 3),4),5),6) 반복

8) PC 값이 non을 가르키고, PC_buffer에도 아무런 값이 없다면 Emulator 종료 후 input 요구에 맞는 출력 제공

과정으로 실행된다.

이 때 프로그램은 {|||||}일 때, 즉 파이프 라인이 모두 비었을 때 completion 되도록 설계 하였다.

5. Results

결과는 다음과 같다.

1. -p option with sample.o

```
./homework3 -p -atp sample.o
```

```
===== Completion cycle: 25 =====

Current Pipiline PC State:
{|||||}

Current register values:
-------------------------------------
PC: 0x400050
Registers:
R0: 0x0
R1: 0x0
R2: 0xa
R3: 0x800
R4: 0x1000000c
R5: 0x4d2
R6: 0x4d20000
R7: 0x4d2270f
R8: 0x4d2230f
R9: 0xfffff3ff
```

```
R10: 0x4ff
R11: 0x269000
R12: 0x4d2000
R13: 0x0
R14: 0x4
R15: 0xfffffb01
R16: 0x0
R17: 0x640000
R18: 0x0
R19: 0x0
R20: 0x0
R21: 0x0
R22: 0x0
R23: 0x0
R24: 0x0
R25: 0x0
R26: 0x0
R27: 0x0
R28: 0x0
R29: 0x0
R30: 0x0
R31: 0x0
```

2. -p, -atp option with sample2.o

```
./homework3 -p -atp sample2.o
```

```
===== Completion cycle: 63 =====

Current Pipiline PC State:
{|||||}

Current register values:
-----------------------------------
PC: 0x400030
Registers:
R0: 0x0
R1: 0x1
R2: 0x0
R3: 0xf
R4: 0xf
R5: 0x0
R6: 0x0
R7: 0x0
R8: 0x10000000
R9: 0x5
```

```
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
R16: 0x0
R17: 0x0
R18: 0x0
R19: 0x0
R20: 0x0
R21: 0x0
R22: 0x0
R23: 0x0
R24: 0x0
R25: 0x0
R26: 0x0
R27: 0x0
R28: 0x0
R29: 0x0
R30: 0x0
R31: 0x400010
```

3. -p, -antp option with sample2.o

```
./homework3 -p -antp sample2.o
```

```
===== Completion cycle: 50 =====

Current Pipiline PC State:
{|||||}

Current register values:
-----------------------------------
PC: 0x400030
Registers:
R0: 0x0
R1: 0x1
R2: 0x0
R3: 0xf
R4: 0xf
R5: 0x0
R6: 0x0
R7: 0x0
R8: 0x10000000
R9: 0x5
```

```
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
R16: 0x0
R17: 0x0
R18: 0x0
R19: 0x0
R20: 0x0
R21: 0x0
R22: 0x0
R23: 0x0
R24: 0x0
R25: 0x0
R26: 0x0
R27: 0x0
R28: 0x0
R29: 0x0
R30: 0x0
R31: 0x400010
```