컴퓨터 구조 Project3 보고서

202011125 유선우

1. 컴파일/실행 방법, 환경

코딩과 테스트는 docker를 이용한 ubuntu 20.04에서 진행하였다.

해당 환경에서 g++의 버전은 9.4.0을 사용하였고,

다음 command를 통해서 컴파일과 실행을 할 수 있었다.

$ g++ pipelining.cpp -o runfile

$ ./runfile [-m addr1:addr2] <-atp or -antp> [-p] [-d] [-n num_instruction] <object file>

2. 코드 작동 Flow

Pipelining의 동작을 하기 위해서는

1) Input command parsing
2) Input file parsing
3) Input file의 data, text를 memory에 할당
4) Instruction을 line by line으로 확인하여 1)에서 확인한 method에 맞춰 cycle을 실행시킨다.
   (각 cycle를 실행시키면서 hazard의 발생에 대처한다.)

의 순서를 거쳐야 한다.

1) Input command parsing

[-m addr1:addr2] [-d] [-n num_instruction] 해당 파라미터에 대해서는 과제2와 같은 방식으로 실행한다.

[-p] 파라미터는 매 실행마다 pipeline의 실행 cycle수와 각 stgae의 instruction

PC를 출력한다.

<-atp or -antp>에 대해서 -atp가 들어가면 매 cycle마다 branch에 대해서 prediction이 된 결과를 사용하고, -antp가 들어가면 매 cycle마다 branch에 대해서 prediction하지 않은 pc를 다음 instruction으로 사용한다.

2) Input file parsing

Object file의 모든 line을 한 줄씩 읽으며, 10진수 값으로 변환하여 vector에 push_back해 저장한다.

3) Input file의 data, text를 memory에 할당

Object file의 첫번째, 두번째 줄을 확인하여 text와 data값을 word단위로 메모리에 할당해준다.

4) Instruction을 line by line으로 확인하여 1)에서 확인한 method에 맞춰 cycle을 실행시킨다.
(각 cycle를 실행시키면서 hazard의 발생에 대처한다.)

먼저 각 stage의 실행이 동시에 작동하는 모습을 구현하기 위해
WB->MEM->EX->ID->IF 순서로 stage를 실행시키고 동시에 작동한다고 본다.

각 stage를 실행시키기 위해 IF/ID, ID/EX, EX/MEM, MEM/WB state register 객체를 정의하고, 해당 객체를 생성하기 위해 state register class를 생성한다.

해당 class에는 다음과 같은 element를 포함한다.

```
class state_register
{
public:
    int Instr = 0;                  // 어떤 instruction 인지 저장
    unsigned int NPC = 0;           // 다음으로 실행될 instruction 의 PC 저장
    int rs = 0;                     // 해당 instruction 의 rs
    int rt = 0;                     // 해당 instruction 의 rt
    int IMM = 0;                    // 해당 instruction 의 IMM
    int rd = 0;                     // 해당 instruction 의 rd
    int ALU_OUT = 0;                // 해당 instruction 의 ALU 연산의 결과
    unsigned int BR_TARGET = 0;     // BEQ, BNE 에서 Branch 할 주소 & jump 의
target address
    int shamt = 0;                  // SLL, SRL 에서 shift 할 개수 (shamt)
    unsigned int cur_PC = 0;        // 현재 PC
    unsigned int ex_PC = 0;         // 이전 instruction 의 PC
    int ALUcntrl = 0;               // ALU 에서 어떤 연산을 할지에 대한 state(덧셈
or 뺄셈 or 그 외)
    int RegWrite = 0;               // Register 에 Write 을 할지에 대한 state
    int Branch = 0;                 // BEQ, BNE 일 경우
    int BranchN = 0;                // BNE 의 경우
    int MemWrite = 0;               // Memory 에 Write 할지에 대한 state
    int MemRead = 0;                // Memory 를 Read 할지에 대한 state
    int MemtoReg = 0;               // Memory -> Register 에 대한 state
    int funct = 0;                  // ALU 에서 어떤 연산을 할지에 대한
state(구체적)
    int RegWriteAddr = 0;           // register 에 값을 저장해야할 때, register
주소
    int MemAddr = 0;                // LW, LB 에서 접근할 메모리의 주소
    int RegData = 0;                // SW, SB 에서 저장할 register 의 값
    int MemData = 0;                // LW, LB 하는 메모리의 값
    int rs_F = 0;                   // rs 에 data forwarding 할 value
    int rt_F = 0;                   // rt 에 data forwarding 할 value
    int forwardA = 0;               // data forwarding 을 할지에 대한 state(rs)
    int forwardB = 0;               // data forwarding 을 할지에 대한 state(rt)
    int Jump = 0;                   // Jump 여부에 대한 state
    int Store = 0;                  // data 를 memory 에 store 할지의 state
    int Byte = 0;                   // 단위가 Byte 단위일 경우 (LB, SB)
};
```

이와 같은 구조를 가지며, 매번 cycle의 진행은 수업시간에 다룬 instruction의 동작 순서를 반영하여 구현하였다.

또한 그 과정에서 발생하는 hazard를 아래와 같은 방법으로 해결하였다.

해당 구조로 state register로 구현하여 실제 수업시간에 배운 알고리즘을 그대로 사용하여 data hazard를 구현할 수 있었다.

Ex)

```
if (EX_MEM.RegWrite == 1 && EX_MEM.RegWriteAddr != 0 && EX_MEM.RegWriteAddr == ID_EX.rs){
        ID_EX.forwardA = 1;
        ID_EX.rs_F = EX_MEM.ALU_OUT;
    }
```

해당 방식으로 rs data를 forwarding하는 값을 rs_F에 저장하고, forwarding한다는 정보를 forwardA에 저장한다.

같은 방식으로 나머지 data hazard도 해결하였으며, 추가로 수업시간에 다루지 않은 MEM/WB -> MEM data hazard는 다음과 같이 바로 data를 forwarding해주어 해결하였다.

```
if (MEM_WB.RegWrite == 1 && MEM_WB.RegWriteAddr != 0 && MEM_WB.MemRead == 1 && EX_MEM.MemWrite == 1 && MEM_WB.RegWriteAddr == EX_MEM.rt){
    if (MEM_WB.Byte == 1){
        EX_MEM.RegData = MEM_WB.MemData;
    }
    EX_MEM.RegData = reg[MEM_WB.RegWriteAddr];
}
```

또한 jump instruction이 실행될 때, hazard를 해결하기 위해 한 번의 flush를 하고, 다음 cycle에 jump address의 instruction을 실행하도록 구현하였다.

또한 BEQ, BNE의 경우 cycle을 진행할 때, atp인지 antp인지 여부를 확인하기 위해 taken Boolean 값을 parameter로 받아 진행하였으며 taken일 경우 taget address를 예측하여 넣어주고 (1cycle 손해) 틀리면 flush하여 3cycle을 손해보며 진행한다. Not taken의 경우 target address가 아닌 PC+4로 원래와 같이 진행하며 틀렸을 경우 flush해주어 3cycle을 손해보며 진행한다.