컴퓨터구조 프로젝트3 보고서

DGIST 기초학부 201611144 임지훈

1. 작성한 과제에 대한 간략한 설명

(0) 과제 3 설명에 없는 내용은 과제 1과 과제 2를 참조했다. 예를 들어 -m 옵션은 과제 2에서 요구한 출력대로 만들었다.

(1) 메모리와 레지스터를 어떻게 구현했는지

메모리는 전체 가상 메모리를 만드는 것은 공간 낭비가 심하기에 Data Section 메모리와 Text Section 메모리를 크기를 이용해 만들었다. 메모리는 각 바이트에 접근할 수 있기때문에 전부 32 Byte임에도 Char로 만들었다. 이와 달리 명령어 버퍼나 PC 등은 int형으로 만들었다. Register File은 32개의 정수 배열로 만들었다.

(2) 파이프라인 상태레지스터를 어떻게 구현했고 왜 그렇게 했는지

파이프라인의 상태레지스터와 같은 실제 시뮬레이터 상의 내용은 파스칼 표기법 (ex. PascalCase)로, 시뮬레이터의 지원과 표기를 위한 변수는 헝가리안 표기법(ex. nInt)으로 구현하였다. p옵션을 위해 명령어를 전달하기는 하지만 각 파이프라인 단계에서는 명령어를 사용하지 않고 제어비트와 명령어의 나뉜 비트만 사용하였다. 각 상태레지스터는 이름 앞에 해당 부분이 들어가있다.

IFID에는 명령어를 전달하기 위한 IFIDInstruction, PC값을 전달할 IFIDNPC, RS를 위한 IFIDRs, Rt값을 IFIDRt에 담아 ID stall을 가능하게 만들었다.

IDEX 상태 레지스터에는 IDEXRs, IDEXRt는 Rs와 Rt값을 전방전달할지 그리고 IDEXRd와 함께 Write register를 구분하기 위해 들어갔다. IDEXALUControlLines과 IDEXALUSrc, IDEXFunctionField, IDEXShamt는 ALU가 어떤 연산을 할지, 그리고 그 양은 어떻게 될지에 대한 값이 필요해서 들어갔다. IDEXMemtoReg, IDEXRegDst, IDEXRegWrite은 다음 WB에서 필요한 값이라 넘겨주었고 IDEXBranch, IDEXMemRead, IDEXMemWrite으로 MEM에 필요한 값을 IDEXReadData1, IDEXReadData2, IDEXImmediateValue, IDEXNPC을 다음 파이프라인 단계에서 필요할 것이기에 구현했다.

EX/MEM에는 ALU의 결과를 EXMEMALUOut에 담기 위해서 넣었다. EXMEMMemRead는 ID에서 해석한대로 메모리를 읽는 명령어인지 제어하기 위해서, EXMEMImmediateValue는 명령어의 Offset, Immediate Value 부분이 WriteRegister로 들어갈 수 있기 때문에, EXMEMMemtoReg은 메모리에서 읽은 값이나 ALUOut, ImmediateValue 중 어떤 값을 WB할지 결정하기 위해서 필요하다. EXMEMRegWrite은 이 명령어가 WB에서 레지스터를 쓰는지 알아내기 위한 제어비트이고 EXMEMWriteRegister은 WB에서 어디에 쓸지를 알기 위해, EXMEMBranch는 MEM에서 명령어가 Branch 인지 아닌지를 알기 위해 필요하다. 마지막으로 EXMEMMemWrite은 MEM에서 메모리를 쓸지 말지, EXMEMReadData2는 메모리에 쓸 수도 있는 값이라, EXMEMBranchTargetAddress는 Branch시 필요한 값이라서 상태레지스터에 저장된다. EXMEMNPC는 MEM에서 분기 예측 실패시 원래 진행되어야

할 다음 PC값을 얻기 위해서 필요하다.

MEM/WB에는 드디어 사용할만큼 사용한 값을 빼고 앞서 말한 이유들의 값이 다 정리된다. MEMWBALUOut(ALU 결과값), MEMWBMemOut(메모리에서 읽은 값), MEMWBImmdiateValue(Immediate Value값-Lui가 사용)은 WB의 Mux에 필요하다. 이 중에 무엇을 출력할지 MEMWBMemtoReg로 결정하고 어디에 무슨 값을 쓸 지를 MEMWBRegWrite, MEMWBWriteRegister가 각각 결정한다. 그래서 위와같은 상태 레지스터들이 필요하다.

(3) 파이프라인 각 단계를 어떻게 구현했고 왜 그렇게 했는지

파이프라인의 각 단계가 WB-MEM-EX-ID-IF의 순서로 처리되게 만들었다. 뒤에서부터 처리되는 이유는 그렇게 해야 상태레지스터의 침범이 일어나지 않기 때문이다. MEM의 Store 부분은 IF에서 변경된 명령어를 조회하지 않도록 만들기 위해서 처리하는 값을 잠시 저장한다. 과제 설명과 수업 자료(PPT)등에 따르기 위해 <그림 1>과 유사하게 만들었다. 그림 외의 명령어와 해저드 처리를 위해 더 많은 부분이 추가되었다. 그림이 PDF로 변환되면서 화질이 너무 낮아져서 별첨하였다.

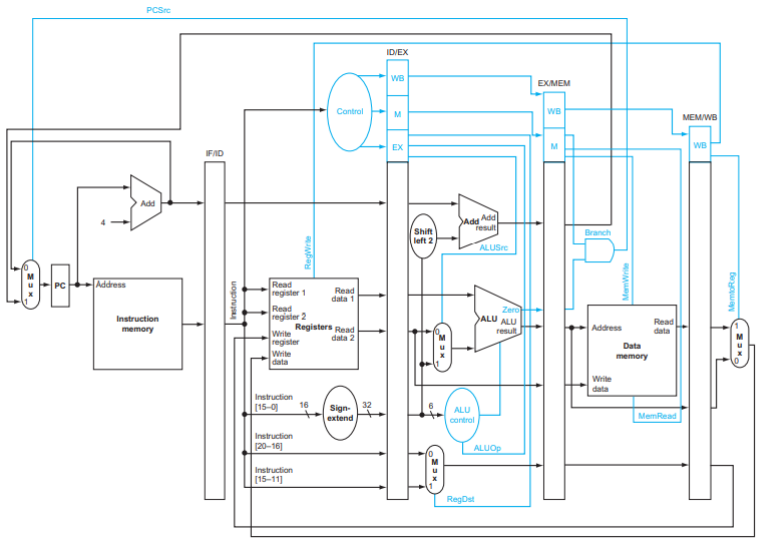

그림 1 기본적 구조 출처 : 교과서

먼저 WB에서는 Mux가 실행되고 EX에서 여기 값을 사용한다면 forwarding해준다. 그리고 레지스터 파일에 Mux 결과값을 저장한다. MEM에서는 EX에 Frowarding하고 메모리에 저장한다. Branch문이 예측대로 됐는지 안됐는지를 조사하여 안됐다면 뒤따르는 3개의 명령어를 Flush한다. 그리고 MEM/WB State Register를 입력한다. EX에서는 R타입 명령어의 컨트롤 라인을 정리하고 ALU를 실행한다. 그리고 EX/MEM State Register를 입력한다. ID에서는 Load등의 MEM/WB to MEM 처리를 Stall로 처리한다. 그리고 Stall이 없다면 뒤따르는 부분의 코드가 작동하게한다. 각 명령에어 맞게 Control Unit이 작동하고 J나 JAL, JR이 처리된다. ID/EX Pipleline State Register를 입력하고 ID 단계가 끝난다. IF에서는 Program Counter에 따라 명령어를 불러오고 IF/ID State Register를 입력한다.

(4) 데이터 해저드와 컨트롤 해저드 해결 방법

데이터 해저드는 책과 PPT의 조건에서 약간의 수정을 거쳐 처리했다. WB에서는 책에서 나온 조건을 사용했으나 EX/MEM to EX의 레지스터와 EX의 레지스터 같은, 그러니까 EX/MEM to EX Forwading이 일어날 때에는 전달하지 않도록 조건을 수정했다. EX/MEM to EX Forwarding은 수업시간에 배운대로 조건을 주었다. ALU의 결과값을 ID/EXReadData 출력에 뒤이어 입력되어 ALUSrc가 복잡해지는 것을 막았다. MEM/WB to MEM은 ID에서 책에 나오는 조건에 Stall을 넣어 한칸 쉬게했다.

컨트롤 해저드는 각각의 명령에서 처리했다. 점프 인스트럭션은 IFID 상태 레지스터를 Flush하여 한 명령어를 쉬게만들었고 Flush라고 IF에 알려 명령어를 읽어오지 않게했다. 조건 분기 인스트럭션은 조금 더 복잡하게 각 상황에 따라 나눠서 처리되었다. Always Not Taken 상황에서는 항상 다음 명령어를 불러온다. 그러나 MEM에서 분기가 일어날 경우 IF/ID, ID/EX, EX/MEM의 상태레지스터 3개를 Flush하여 3사이클 동안 지연이 일어나도록 구현했다. Always Taken 의 경우에는 일단 ID에서 한 Stall을 만들고 TargetAddress를 계산한 다음 그 값을 바로 PC에 전달하여 만들었다. 만약 MEM에서 분기가 일어나지 않는다면 3개의 Flush가 일어난다.

2. 과제의 컴파일 방법 및 컴파일 환경/ 과제의 실행 방법 및 실행 환경

C언어를 사용하여 코딩했고 Ubuntu 18.04 터미널에서 프로젝트 파일에 들어간 뒤 gcc 7.5.0 버전을 이용하여 main.c이 있는 창에서 다음과 같이 컴파일 하였다.

gcc -o runfile main.c

3. 과제의 실행 방법 및 실행 환경

기본적으로 꼭 -atp 나 -antp, binary file(sample.o나 sample2.o)를 꼭 넣고 ./runfile <-atp 나 -antp> [-m addr1:addr2] [-d] [-p] [-n num_instr] <binary file> 형태로 실행한다. 만약 sample.s를 어셈블러로 처리한 sample.o를 분기가 일어날 것이라고 예측하며 실행하기위해서는 터미널에서 다음과 같이 실행하였다.

./runfile -atp sample.o

분기가 일어나지 않을 것이라고 예측할 때에는 다음과 같이 실행하였다.

./runfile -antp sample.o

그리고 다시 sample2.o를 실행할 때에는 다음과 같이 했다.

./runfile -atp sample2.o

다양한 옵션을 테스트하기위해 ./runfile -atp -n 0 sample.o 나 ./runfile -atp -m 0x400000:0x400010 -d -n 3 sample.o 등을 사용해 특정 메모리를 참조할 수 있고(-m 원하는 메모리 영역) 명령어 수행 후 레지스터 결과를 출력할 수 있으며 원하는 갯수만큼 명령어를 실행시킬

수 있다.