2025 Spring

CSED311

컴퓨터구조

Lab 4-2 report

Team ID: 15

팀원 1: 20230345 이성재

팀원 2: 20230355 정지성

목차

1. 서론
2. 디자인
3. 구현
4. 논의 사항
5. 결론

* Compare total cycles of 2-bit global prediction with that of always-taken and always-not-taken

Report (Deadline: 5/13 18:00)

• How to handle branch prediction?

• Describe your design of branch predictor

• If you implement 2-bit global prediction

• Compare total cycles of 2-bit global prediction with that of always-taken and always-not-taken

• If you implement always-taken

• Compare total cycles of always-taken with that of always-not-taken

1. **서론**

이 과제의 목표는 Lab 4-1 에서 구현했던 pipelined cpu를 data hazard 뿐만 아니라 control hazard 도 해결할 수 있도록 변형하는 것이다. 또한 추가 점수를 위해, Branch prediction도 구현을 해야 하고 우리는 Gshare를 만들었다. 이 보고서에서 우리는 먼저 branch prediction을 하지 않은, 즉 always-not-taken 하는 cpu를 구현한 방법과 결과를 다루고 Gshare를 적용한 cpu에 대한 설명을 이후에 다루도록 하겠다.

1. **디자인**
2. **Always-not-taken CPU**

우리는 먼저 control flow instructions, 즉 BEQ, BNE, BLT, BGE, JAL, JALR 명령어들에 대해서 파이프라인이 정상적으로 작동하도록 always-not-taken 방식을 구현했다. IF 단계에서 읽은 명령어가 control instruction이더라도 항상 다음 pc를 pc+4로 예측하는 방식인데, 이는 매번 control instruction 인지 stall로 확인한 후 나아가는 방식보다는 높은 성능을 보이지만, 잘못 예측한 경우 flush 해야 하는 명령어들이 생긴다는 문제점이 있다. 이 과제에서는 특히 EX 단계에서 control flow를 잘못 예측했는지 확인해 잘못 예측한 경우 2개의 버블만 발생하도록 요구한다.

텍스트, 도표, 평면도, 개략도이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

위의 그림은 수업 ppt에 나와 있는 구조도이다. 처음엔 BranchUnit을 만들어 분기 여부와 분기 시의 주소를 해당 unit에서 결정하도록 했지만 ALU에서 한번에 할 수 있는 작업이라, 모듈을 없애고 **branch\_taken** 과 **branch\_addr** 이라는 레지스터 및 추가적인 컨트롤 시그널로 대체했다.

* **ID\_EX\_is\_jal** : 명령어가 jal인 경우 그 정보를 EX 단계까지 갖고 가기 위한 레지스터
* **ID\_EX\_is\_jalr** : 명령어가 jalr인 경우 그 정보를 EX 단계까지 갖고 가기 위한 레지스터
* **ID\_EX\_branch** : opcode가 BRANCH인 경우 1로 설정되는 레지스터

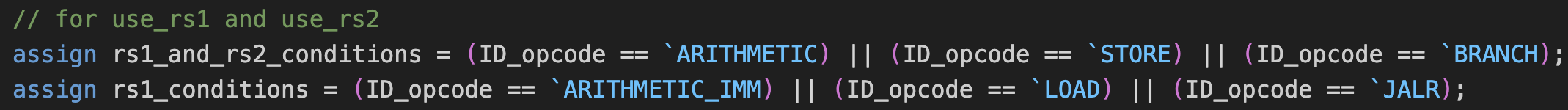
위의 3개의 레지스터가 or 연산자로 연결 돼 **branch\_taken**의 값을 결정한다.

Pc+4 가 잘못된 예측이었을 경우 이를 해결하기 위해 해야할 일은 크게 두가지이다. 먼저 IF/ID 단계에 있는 instruction을 nop로 만들어야 한다. 다음으로 ID/EX 단계에 있는 control signal 들을 0으로 만들어 잘못된 명령어의 결과가 pvs에 반영되지 못하게 해야 한다.

1. **CPU with Branch prediction**
2. **구현**
3. **Always-not-taken CPU**

먼저 jal/jalr/branch 명령어들을 처리해야 함에 따라 일부 수정된 모듈들에 대해 설명하도록 하겠다.

* **StallDetection.v**



Data hazard를 막기 위해 stall이 필요한지를 판단하는 로직에서, rs1과 rs2를 사용하는 명령어의 종류에 BRANCH와 JALR을 추가했다.

* **ControlUnit.v**
  + **is\_jal :** 명령어가 jal인 경우 1로 설정
  + **is\_jalr :** 명령어가 jalr인 경우 1로 설정
  + **branch :** 명령어의 opcode가 BRANCH인 경우 1로 설정
  + **pc\_to\_reg :** 명령어가jal/jalr인 경우 1로 설정

위의 4개의 출력 레지스터가 추가되었다. 또한 case 문에서 이 레지스터들의 값을 결정하는 블록들이 추가되었다.  
**텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

다음으로 EX 단계에서 어떻게 이미 처리하고 있는 명령어들을 flush 시킬지 판단하는 로직을 설명하도록 하겠다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

기존 CPU 구현에서 추가된 부분은 branch\_taken과 branch\_addr을 계산하는 부분이다. 명령어가 jal/jalr 인 경우 무조건 분기를 해야 하고, BRANCH 명령어인 경우 alu\_bcond가 1일 때 분기를 해야 한다. branch\_addr은 branch\_taken인 경우 사용될 주소로, jalr인 경우 pc값에 관계없이 ALU가 연산한 결과를 주소로 이용한다. 그 외의 경우라면 pc 값에 imm 값을 더한 주소를 이용하면 된다.

같은 클럭 사이클에서 EX 단계가 branch\_taken을 1로 설정했다면, IF와 ID에 있는 명령어들은 잘못된 흐름의 명령어들이 된다. IF/ID 단계에서 다음과 같이 처리해주면 IF에서 읽은 명령어를 nop로 바꿀 수 있다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

또한 ID/EX 단계에서는 control signal들을 모두 0으로 만들어줘야 한다.

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

이렇게 두 단계를 통해 정상적으로 flush 처리를 할 수 있다. 마지막으로 jal/jalr 명령어의 경우 자신의 pc에 4를 더한, 즉 다음 명령어를 가리키는 값을 레지스터에 저장해야 한다. 이를 위해 IF/ID 에서부터 ..pc\_plus\_4 라는 레지스터를 만들고 ..pc\_to\_reg 라는 control을 만들었다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

1. **CPU with Branch prediction**
2. **논의 사항**
3. **결론**
4. **Always-not-taken CPU**

**텍스트, 스크린샷, 패턴, 패브릭이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 패턴, 스크린샷, 패브릭이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 패턴, 패브릭이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

**텍스트, 패턴, 스크린샷, 패브릭이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 패턴, 패브릭이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

왼쪽 위에서부터 차례대로 basic, ifelse, loop, non-controlflow, recursive 테스트의 실행결과이다. Control flow를 테스트 하는 테스트의 정답과 1사이클의 차이만 나는 것으로 보아 제대로 구현했음을 알 수 있다.

1. **CPU with Branch prediction**