2025 Spring

CSED311

컴퓨터구조

Lab 4-2 report

Team ID: 15

팀원 1: 20230345 이성재

팀원 2: 20230355 정지성

목차

1. 서론
2. 디자인

* How to handle branch prediction?
* Describe your design of branch predictor

1. 구현
2. 논의 사항
3. 결론

* 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. **Always-taken CPU**

여기서부터는 BTB, 즉 Branch Target Bugger 를 구현해야 한다. 정확히는 Tagged BTB를 이용해 구현했고, 디자인은 수업 ppt의 구조도를 참고했다.

도표, 텍스트, 평면도, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

이 방식은 BTB index와 tag를 이용해 BTB에는 branch, jump 명령에 따른 pc 이동 결과만 저장할 수 있도록 한다. 그림에서 알 수 있듯 Pc의 tag 부분이 tag table의 BTB index에 저장된 값과 같다면 nextPC를 BTB에 저장된 값으로 예측한다. 만약 같지 않다면 pc+4를 다음 pc로 이용한다.

BTB는 모든 entry에 대해 0으로 초기화 되고, clock에 맞춰 업데이트 된다. 업데이트는 Pc가 0이 아닐 때만 일어나며 tag table은 N-bit에 대응되는 entry가 현재 pc의 tag로 업데이트 된다. BTB도 비슷하게 업데이트가 일어난다. 또 교안에는 나와 있지 않지만, valid라고 하는 비트를 적절히 켜줘야 한다. 그렇지 않으면 BTB에 저장된 쓰레기값이(valid == 0인 상태의 값) pc에 들어갈 수도 있기 때문이다.

1. **2-bit global prediction**

이 구현에서는 BTB 뿐만 아니라 PHT도 구현해야 한다. PHT의 구현은 2-bit state machine으로 했으며 교안에 소개된 두 가지 방식으로 모두 구현해보았다.

텍스트, 도표, 스크린샷, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

이전에 구현했던 BranchPredictor 모듈에 PHT 부분을 추가하는 것으로 충분했으며, PHT 구현을 위해서 HysteresisCounter와 SaturationCounter 모듈을 만들었다. State machine의 디자인은 아래의 교안 내용을 참고했다.

텍스트, 도표, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 도표, 폰트, 스크린샷이(가) 표시된 사진

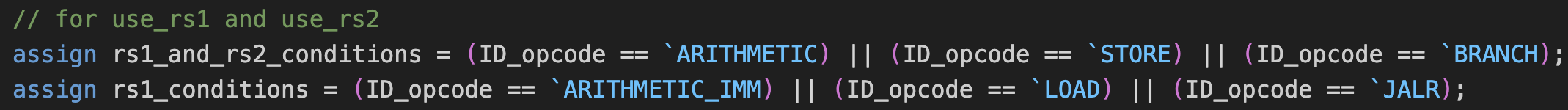
AI가 생성한 콘텐츠는 부정확할 수 있습니다.

1. **Gshare**

1. **구현**
2. **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. **Always-taken CPU**

먼저 BranchPredictor.v 라는 모듈이 추가되었다.

**<BranchPredictor>**

Input:   
**reset, clk  
update\_pc, update\_BTB, update\_taken  
current\_pc**

Output: **predicted\_next\_pc, predicted\_branch\_taken**

여기서 tag, BTB의 사용 및 업데이트가 이뤄진다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

디자인 부분에서 논의했던 것처럼 wire와 레지스터들을 설정하고, 적절한 값들에 대응시켜줬다. 우리는 N을 5로 설정해 총 32개의 entry를 이용할 수 있도록 했다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

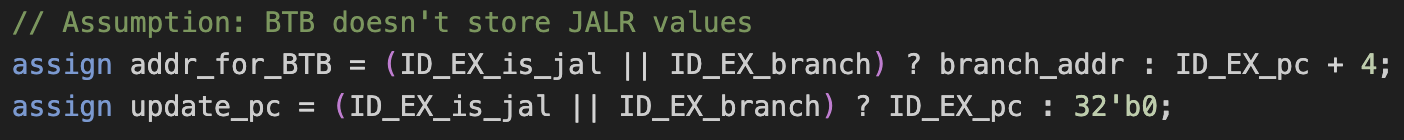
0으로 초기화 하는 작업과 업데이트가 일어난다. 이때 jal과 branch 명령에 대해서만 업데이트를 해야 하는데, 이는 cpu.v에서 update\_pc 값을 변화시키는 방식으로 관리한다. 또 이러한 테이블 업데이트는 synchronous하게 일어나야 한다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

모듈의 출력은 위와 같은 로직으로 결정된다.

이 모듈을 활용하기 위해 cpu.v에도 약간의 수정이 필요했다.



위의 코드는 EX 단계에서 실행되는 코드로, BTB의 입력으로 사용될 pc 주소를 결정한다. Update\_pc는 BTB에 jump와 branch 명령어에 대한 내용만 저장하기 위해, 해당하지 않는 명령어의 경우 update\_pc를 0으로 만들어 모듈 내에서 업데이트를 하지 않도록 한다.

마지막으로 prediction\_wrong 이라는 레지스터로 branch\_taken을 대체 했는데, 이 값은 아래와 같이 계산된다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

1. **2-bit global prediction**

BranchPredictor.v에서 가장 크게 바뀐 부분은 PHT의 정의와 업데이트 로직이 추가 됐다는 점이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

코드는 Saturation counter를 사용하는 것을 기준으로 설명하지만, Hysteresis Counter를 사용하는 것으로 바꿀 수도 있다. BTB와 마찬가지로 0으로 초기화를 하고, PHT의 값을 next\_PHT로 바꾼다. 이때 next\_PHT는 State machine 모듈의 출력이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

해당 모듈 내에서, Saturation counter가 다음과 같이 들어있으며 이 모듈의 출력이 PHT entry에 저장되게 된다. State machine의 코드는 다음과 같이 구현되어 있다.

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

디자인 부분에서 언급한 상태 머신을 그대로 옮긴 것이고, branch를 taken했는지 여부에 따라 state이 달라진다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

HysteresisCounter의 구현도 비슷하지만, 상태 전이 방식이 다르다.

다시 BranchPredictor로 돌아와서, predicted\_next\_pc와 predicted\_branch\_taken을 결정하는 로직은 다음과 같이 변경 되었다. 텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

이전 구현에선 valid 하고, tag가 일치하면 BTB의 값을 꺼내 썼지만 이 방식에선 PHT의 state에 따라 branch를 taken으로 예측할지, not taken으로 예측할지가 달라진다. 교안에서 사용된 다이어그램을 보면 알 수 있듯이, PHT[predict\_index][1]의 값이 1이어야 taken으로 예측하는 상태이다.

cpu.v 모듈에서는 일부 최적화를 위한 수정이 있긴 했지만 로직 상에 변화는 없었다.

1. **Gshare**
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. **Always-taken CPU**

**텍스트, 스크린샷, 패턴, 모노크롬이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 패턴, 패브릭, 모노크롬이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.패브릭, 텍스트, 패턴, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

**텍스트, 패턴, 스크린샷, 패브릭이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.패턴, 패브릭, 스크린샷, 모노크롬이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

1. **2-bit global prediction**

**패턴, 모노크롬, 패브릭, 스티치이(가) 표시된 사진

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

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 패턴, 패브릭, 모노크롬이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 패턴, 스크린샷, 패브릭이(가) 표시된 사진

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

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

1. **Gshare**