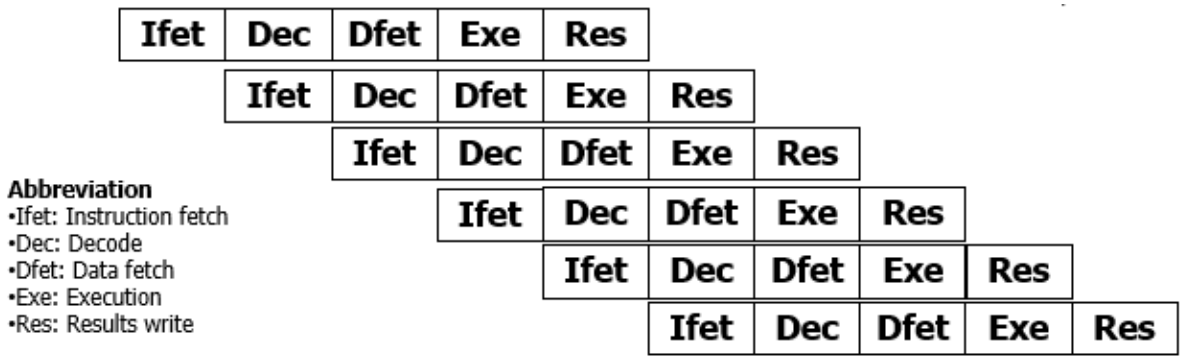
시스템프로그래밍
레포트#5

과 목 명	시스템프로그래밍
교 수	최 종 무
학 번	32163006 32164420
이 름	이 건 욱 조 정 민

Contents

▶ 문제제시

우리는 명령어를 수행하는데 있어서 RISC(Reduced Instruction Set Computer)를 사용하고 더 적은 시간에 더 많은 명령어를 수행하기 위해 즉 throughput를 향상시키기 위해 대표적으로 pipeline기법을 사용한다. pipeline 기법은 명령어를 여러 단계로 나누고 나눈 명령어들을 한 개의 clock마다 병렬적으로 수행하는 것을 말한다.

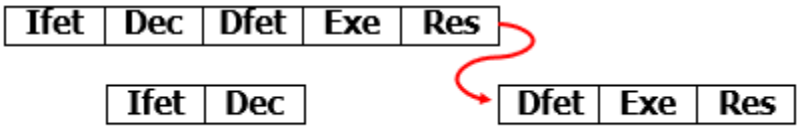


pipeline기법을 사용함으로써 인해 성능 향상이라는 아주 중요한 부분을 취할 수 있지만 역시나 pipeline을 사용함으로써 인해 발생하는 문제점들이 있다. pipeline 기법은 명령어를 여러 단계로 쪼개 병렬적으로 연결함으로써 쪼개는 단계 수에 비례하여 성능이 향상되지만 이 단계 수를 과도하게 쪼개서 클럭 수를 과도하게 늘리게 되면 컴퓨터의 과도한 전력 소모와 열 발생 문제를 초래할 수 있다. 대표적인 예로 Pentium4의 'NetBurst microarchitecture'의 Deep pipelining 기법이 있다.

또 다른 문제점으로는 hazard가 있다. 대표적인 hazard로는 resource hazard와 data hazard가 있는데 resource hazard는 이미 파이프라인에 들어와 있는 두 개(혹은 그 이상)의 명령어들이 동일한 자원을 필요로 할 때 발생한다. 예를 들면 IF(Instruction Fetch)와 OF(Operand Fetch) 모두 메모리에 엑세스해야 하지만 동시에 두 개의 접근은 불가능한 경우이다. 또 data hazard는 오퍼랜드 위치에 대한 액세스에 충돌이 있을 때 발생한다. 예를 들면 어떠한 명령어가 ADD 명령을 수행중일 때 그 다음 명령어가 ADD명령을 수행한 결과를 Fetch하려 할 때는 아직 전 명령어가 ADD명령을 수행중이기 때문에 Fetch가 불가능하다.

< Resource Hazard >

< Data Hazard >

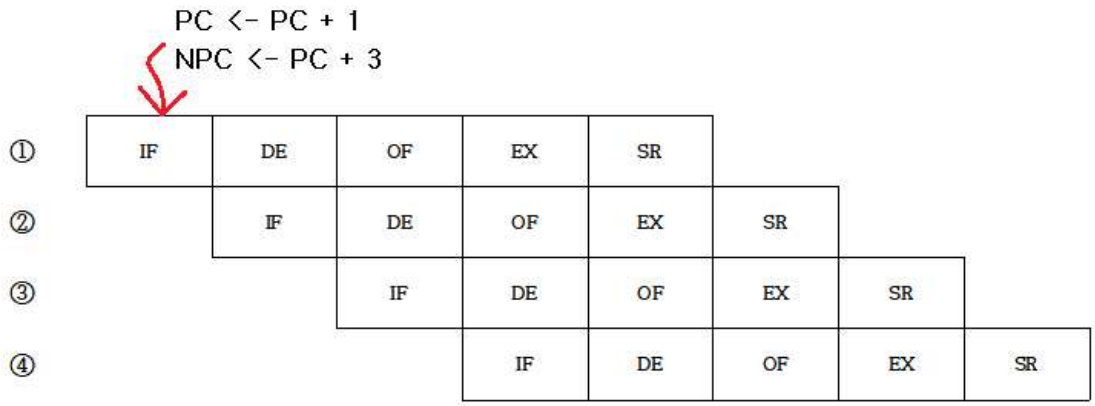


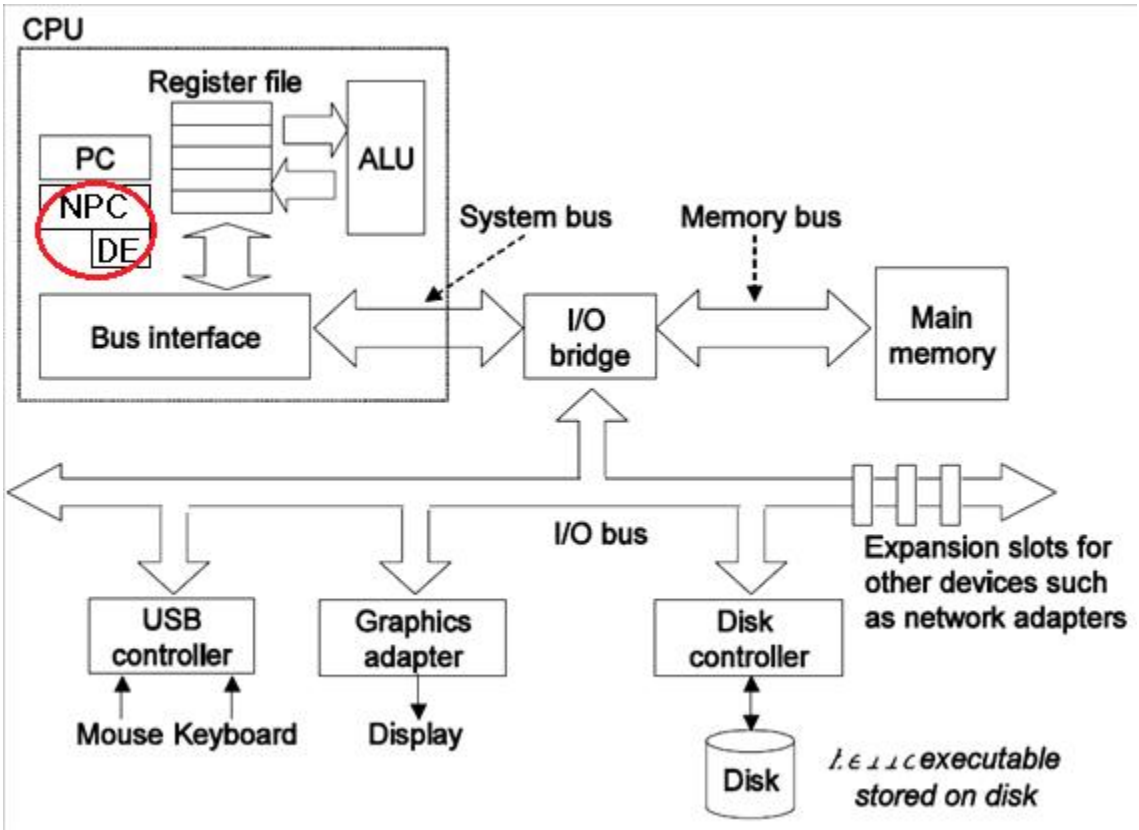
resource hazard의 대표적인 해결책으로는 Out of Order Execution(무순서 실행)이나 Cache를 활용하는 방법이 있고 data hazard의 대표적인 해결책으로는 Branch Prediction(분기 예측)이 있다. 하지만 Branch Prediction은 지금까지의 실행결과들을 토대로 예상되는 결과를 예측하여 미리 실행하는 것이기 때문에 정확한 해결방법이라고 말할 수는 없다. 그래서 우리는 이 Branch Prediction보다 더 나은 방법에 대해 생각해보기로 했고 우리들의 수준에서 완벽하고 논리정연한 방법을 구현할 수는 없겠지만 지금까지 강의시간에 배웠던 내용들을 바탕으로 새로운 해결방법을 모색해 보았다.

▶ 해결방안 모색

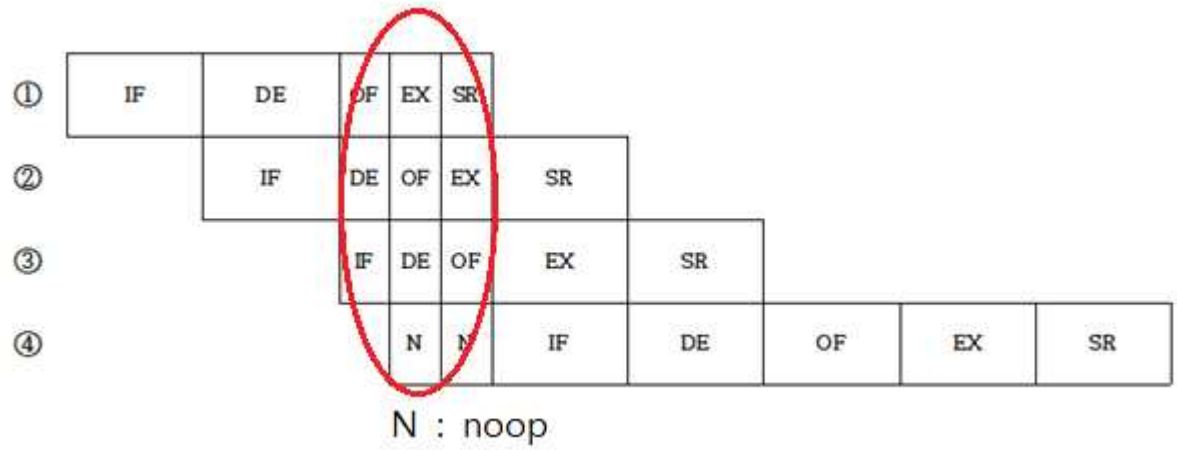
pipeline에서 명령어의 단계를 나누는 개수는 여러 종류의 CPU마다 다른데 우리는 한 명령어를 다섯 단계일 경우로 가정하였다.
PC(Program Counter)는 다음 실행될 명령어의 주소를 저장하는데 우리는 이걸 활용하여 PC를 하나 더 추가하였다.
명령어의 주소를 저장하기 위해 새로운 NPC(New Program Counter)라는 레지스터를 만들고 자신보다 3클럭 늦게 수행될 명령어의 opcode를 해석해서 분기문인지 확인하기 위한 용도의 decoder를 추가해준다.

.





NPC에 저장되는 명령어의 주소에서 해당 명령어의 opcode를 해석한 뒤 분기문이면 첫 번째 Instruction에서 OF부분의 clock을 순간적으로 3배로 튀겨준다.



다시 말해서, ① 명령어의 DE에서 ④ 명령어가 분기문임을 확인하게 되면 ① 명령어의 OF가 3단계로 쪼개지게 되고 그렇게 되면 ④ 명령어까지 쪼개지게 된다. 하지만 ④ 명령어까지 쪼개졌다 하여도 ③ 명령어에 저장된 데이터를 가져오는데 delay가 여전히 발생한다. 그래서 생각한 방법이 ④ 명령어의 IF 앞부분에 비어있는 공간을 넣어주는 것이다. 그러면 delay없이 이전 명령어의 저장된 데이터를 가져와서 사용할 수 있다.

하지만 여기서 문제점이 발생한다. 만약 이후에 수행되는 명령어에서 또 분기문이 나오게 된다면 3단계로 쪼개진 클럭이 또 쪼개지게 되면서 문제 제기에서 언급했었던 pipeline을 과도하게 쪼개면서 생길 수 있던 문제가 발생할 수 있게 된다. 그래서 우리가 추가적으로 제시한 해결방법은 짧은 루프에서는 Branch Prediction과 함께 사용되어야 하지만 루프가 길어지며 분기문이 긴 간격을 두고 나올 때에는 Branch Prediction을 사용해야 할 빈도가 줄어들고 성능을 더 향상시킬 수 있을 것이라고 예상했다.

▶ 기대효과 및 예상되는 문제 발생 원인

우리가 제시한 방법들로 인해 data hazard를 해결하고 branch prediction의 비확실성으로 인한 시스템 성능 저하를 최소화 시킬 수 있을 거라고 예상한다. 하지만 분기문을 예상하고 clock이 쪼개진 상태에서 바로 또 분기문을 확인하였을 때에는 어쩔 수 없이 결국 다시 branch predition을 사용할 수밖에 없는 점이 아쉬운 것 같다. 이러한 원인으로 우리가 처음에 예상했던 것 만큼의 기대효과를 얻기는 힘들 것 같지만 우리의 생각으로는 분기문은 보통 연달아 나오지 않고 코드가 매우 길고 그 코드의 루프문들의 내용들도 길어지는 경우가 많기 때문에 clock의 쪼개짐이 겹쳐지는 경우가 드물거라는게 우리의 예상이다.

▶ 역할분담

이 건 욱	조 정 민
아이디어 및 문제 제시	아이디어 구현 및 설계
문제 해결 방안 모색	문제점 보완
문서 작성 및 해결 방안 토의	

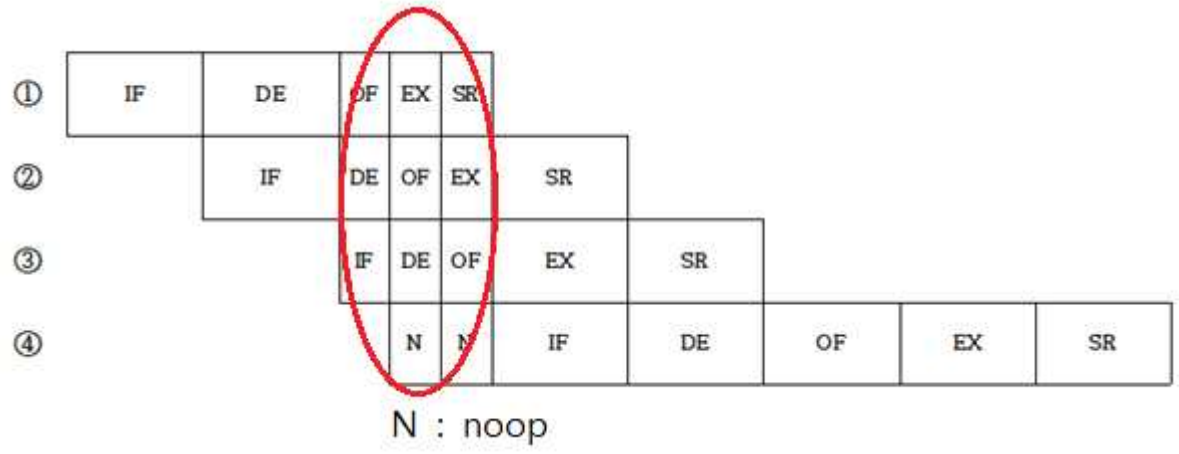
▶ Discussion

[이건욱]

처음 이 과제를 받고 너무 막막하였다. 문제점이나 좀 더 신박한 방법을 생각해내서 아이디어를 낸다는 것 자체가 전반적인 내용들을 상당 수준 이해하고 있어야 한다는 전제가 있다고 생각했고 태어날 때부터 아이디어란 것은 내본 적이 없는 나로썬 너무 막막하였다. 하지만 막상 시작하고 나니 괜찮지 않았다. 모든 시스템프로그래밍 수업자료와 컴퓨터구조 수업자료를 뒤지다보니 그나마 너무나 반복적으로 많이 공부했던 pipeline에서 더 효율적인 방법을 찾고 싶었다. 평소에 NetBurst microarchitecture의 문제점인 전력소모와 열 발생을 해결할 수 있는 방법이 없을까란 생각을 조금 해왔었기에 처음에는 이 점을 어떻게 하면 해결할 수 있을까라고 생각했지만 포기했다. 그래서 현재의 pipeline 활용과 Netburst를 조금 섞어보면 조금 더 효율적이지 않을까란 생각에 이 방법을 찾아내게 되었다.

Pipeline의 대표적인 문제점이라면 hazard가 있고 그 중에 data hazard의 해결방법인 branch prediction은 내가 보기에 대책답지 않은 대책이라고 생각했다. 그래서 branch prediction을 대신할 아이디어를 생각해 보았다. 일단 data hazard를 해결하기 위해선 분기문인 명령어에서 operand를 fetch하기전에 전 명령어가 store까지 완료되어야 한다고 생각했고 그러면 전 명령어들을 빨리 끝내버릴 방법을 찾던중 순간적으로 cycle을 쪼개는 방법을 생각하게 되었다. 처음에는 바로 전 단계 명령어의 decode를 쪼개려고 했으나 전 단계 명령어의 IF 이후의 단계를 쪼개면 결국 분기문 명령어의 단계도 쪼개지기 때문에 의미를 잃는걸 깨닳았다. 그래서 그 전전단계의 명령어 전전전단계의 명령어까지 올라가게 되었고 전전전단계에서 다다다음명령어가 분기문인지 확인시킬 방법을 찾던 중 pc를 활용하면 될 것 같단 생각이 들어 pc를 하나 더 추가하게 되었다. 또 pc만 추가한다고 분기문 명령어가 분기문인지 확인할 수 없기 때문에 다다다음 명령어의 opcode가 분기문인지 decode할 수 있는 opcode 전용 decoder를 추가하였다.

하지만 역시나 좀 수월하다 싶으면 문제가 발생한다. 역시 인생은 쉬울 리가 없었다. 1단계씩 차근차근 그려나가면서 수행했어야 하지만 4단계를 한번에 그려놓고 문제를 해결하려고 하다보니 결국



①의 OF를 쪼개면 4번은 멀쩡할거라 생각했지만 뭔가 이상해 들여다보니 아니다 다를까 ④번도 땡겨질 것 이라는걸 그냥 놓쳐버렸다. 아무리 생각해도 다다음명령어 확인하는 것도 좀 억지스러운데 다다다음에 다를 하나 더 붙이기는 싫어서 일단 NOOP를 넣기로 의견을 모았다고 NOOP을 넣어도 우리가 생각하기에는 충분한 효율을 증가시킬 수 있을 것이라고 예상하였다.

난 솔직히 이번 과제 점수에 큰 욕심은 없다. 아이디어가 많이 참신하다고 생각하지도 해결방법이 참신하다고 생각하지도 않는다. 하지만 그래도 막상 과제를 해결하고 나니 과제의 취지가 예상되었고 지금까지 배웠던 내용들을 되돌아보고 전반적인 나의 현 상태를 확인할 수도 있는 계기가 되었던 것 같다. 팀원과 이건 이거다 저건 저거다 서로 아는거 모르는거 다 끌어모으며 과제를 해결해 나가는 과정이 나름 재미있었던 것 같았다. 처음 시스템프로그래밍을 시작할 때는 내가 뭘 모르는지 조차 모르는 이게 소프트웨어학과 학생인가란 생각을 했었지만 억지로 집어넣고 노력하며 공부하다보니 많은 걸 알게 되어 뜻 깊은 한 학기를 보낸 것 같다. 끝!

[조정민]

막상 과제를 시작하려니 어떤 주제를 선택해서 과제 진행을 해야할 지 정말 고민을 많이 했다. 강의자료를 모두 뒤져봐도 반짝 떠오르는 아이디어가 없어서 힘들었다. 결국 Racd condition, pipeline, fork() 중 pipeline에 관련하여 아이디어를 생각하였고 pipeline hazard를 기존에 알고 있던 방식과는 다른 방법을 사용하여 해결해보는 것은 어떤지 의견을 모았다. 주제를 정해놓고 서로 머리를 맞대며 기발한 생각을 하려고 하였지만 생각보다 쉽지 않았다. 강의를 통해 배운 내용, 기존에 있던 방식을 고수하는 고정관념 때문인 것 같았다. 여러 생각을 한 끝에 우리는 data hazard를 해결하기 위한 새로운 방법에 대하여 과제 진행하는 것에 초점을 맞췄다. data hazard를 해결하기 위해서 우리는 Branch prediction을 사용한다. 우리는 Branch prediction의 한계에 대해서 생각했고 Instruction의 특정 부분의 clock을 순간적으로 높이는 부분에 대해서 연구하였다. 시스템프로그래밍 과목과 컴퓨터구조 과목이 겹치는 내용이 많았고 pipeline 부분도 겹쳤다. 우리는 한 Instruction의 단계를 두 과목에서 공통적으로 사용하는 다섯 단계로 지정하였다. data hazard는 이전 명령어의 결과를 가져와 data fetch를 해야하는데 이러한 과정에서 delay가 생겨 발생되는 hazard이다. 우리 수준에서 생각해 낼 수 있었던 방법은 Instruction의 디코딩 부분에서 +3번째 Instruction이 분기문인지 아닌지 확인하는 것이었고, 이를 새로운 레지스터(NPC)를 통해 확인하는 것이라 생각하였다. +3번째 Instruction이 분기문이면 순간적으로 OF부분에서 clock을 높여준다. 그리고 +3번째 instruction의 IF 앞부분에 NOOP를 두클럭으로 넣어주면 바로 이전 Instruction의 data를 delay없이 가져올 수 있다고 생각하였다. 우리가 생각한 방법이 정상적으로 수행될지는 확신도 서지 않고 억지라는 생각도 들었지만, 이번 과제를 통해서 우리에게 개발자의 입장에서 컴퓨터를 들여다보는 시야가 생겼다고 느꼈다. 또한 팀원과 함께 과제를 하면서 서로의 지식을 공유하고 강의 때 배운 내용들을 복습까지 하면서 시스템프로그래밍이란 과목에 더 쉽게 접근할 수 있었고 놓친 부분에 대해서 다시 생각할 수 있었다. 직접 이런 아이디어를 생각해내고 문제점을 찾고 새로운 방법으로 아이디어를 구현해내려고 한다는 것 자체가 뜻깊은 시간이었던 것 같다. 이런 팀과제를 하다보면 팀원과의 의견 충돌이 있을 거라 생각하였는데 충돌 하나 없이 수월하게 과제를 진행할 수 있었다. 이번 과제를 통해 pipeline을 배웠던 부분 중에서 놓쳤던 부분, 다시 복습하며 배워야할 부분에 대해서 다시 한 번 짚고 넘어갈 수 있는 좋은 계기가 되었고, 팀플레이를 통해서 팀워크의 중요성을 다시 한 번 느낄 수 있었다.