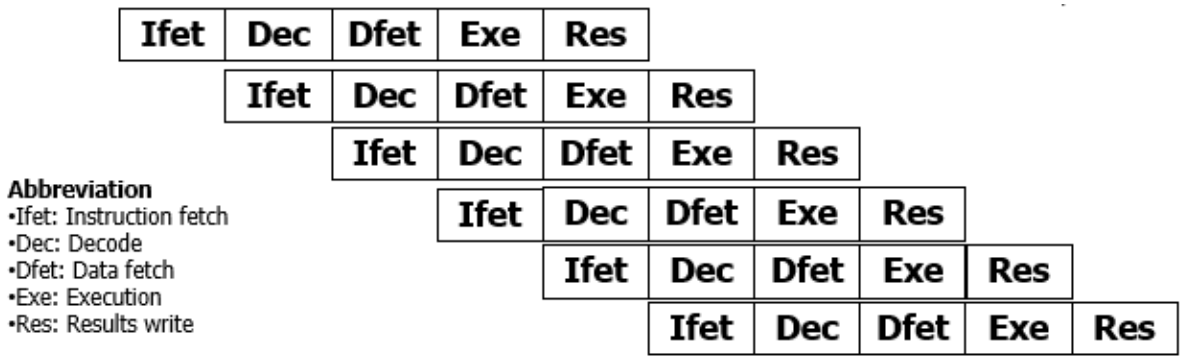
Contents

▶ 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.

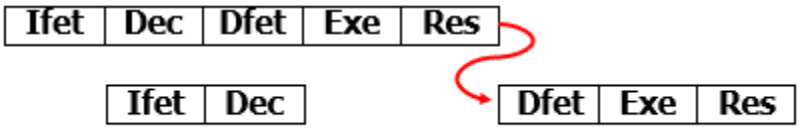


pipeline기법을 사용함으로써 인해 성능 향상이라는 아주 중요한 부분을 취할 수 있지만 역시나 pipeline을 사용함으로써 인해 발생하는 문제점들이 있다. pipeline 기법은 명령어를 여러 단계로 쪼개 병렬적으로 연결함으로써 쪼개는 단계 수에 비례하여 성능이 향상되지만 이 단계 수를 과도하게 쪼개서 클럭 수를 과도하게 늘리게 되면 컴퓨터의 과도한 전력 소모와 열 발생 문제를 초래할 수 있다. 대표적인 예로 Pentium4의 'NetBurst microarchitecture'의 Deep pipelining 기법이 있다.

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.

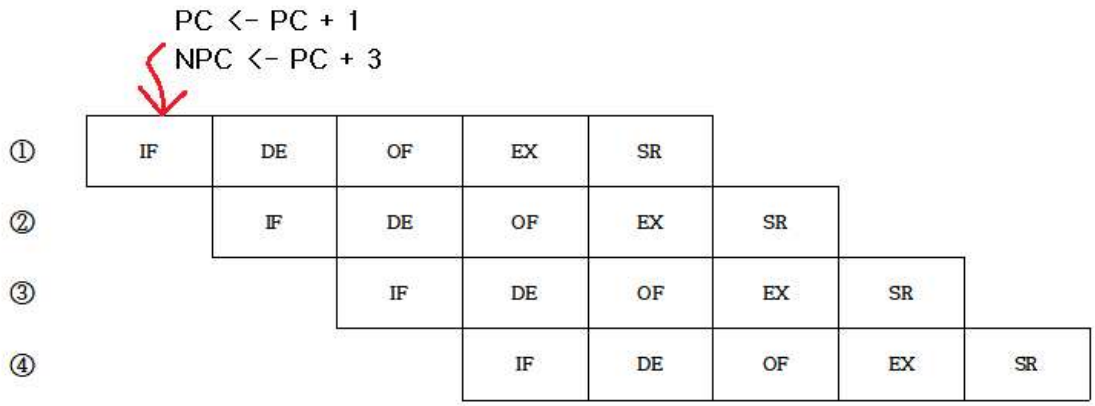
▶ 해결방안 모색

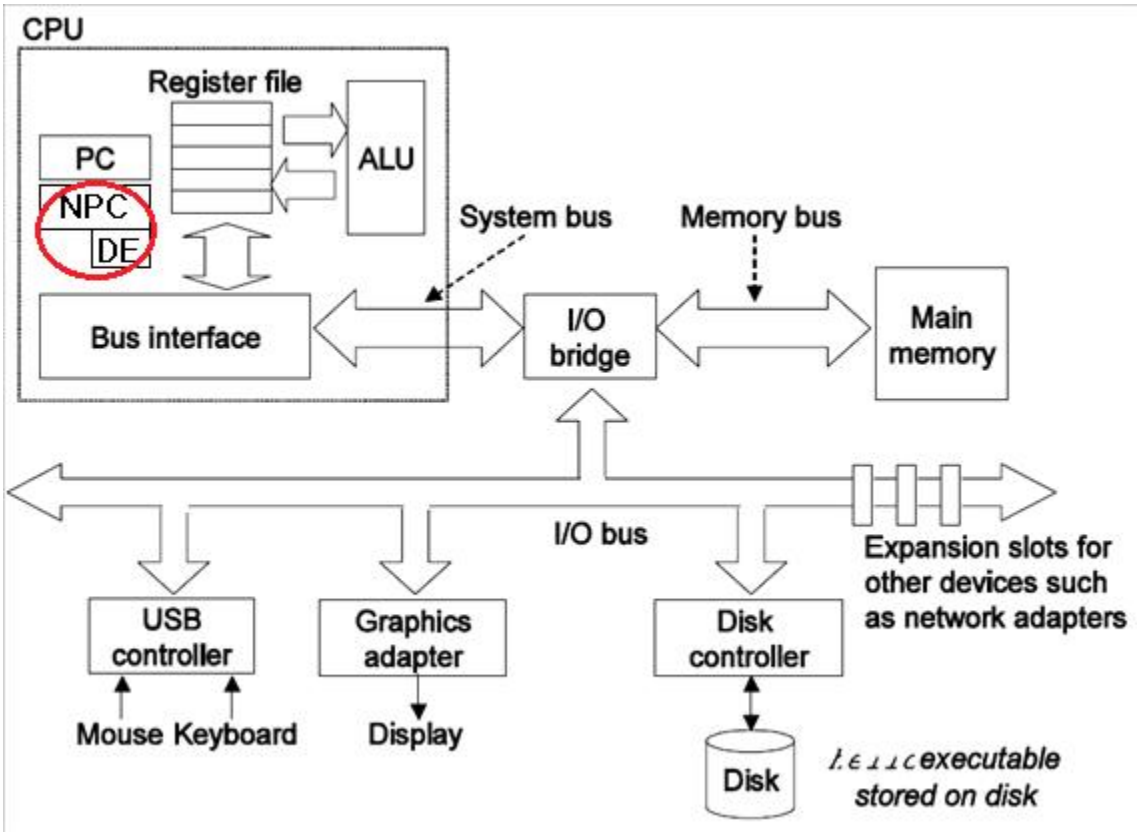
pipeline에서 명령어의 단계를 나누는 개수는 여러 종류의 CPU마다 다른데 우리는 한 명령어를 다섯 단계일 경우로 가정하였다.

PC(Program Counter)는 다음 실행될 명령어의 주소를 저장하는데 우리는 이걸 활용하여 PC를 하나 더 추가하였다.

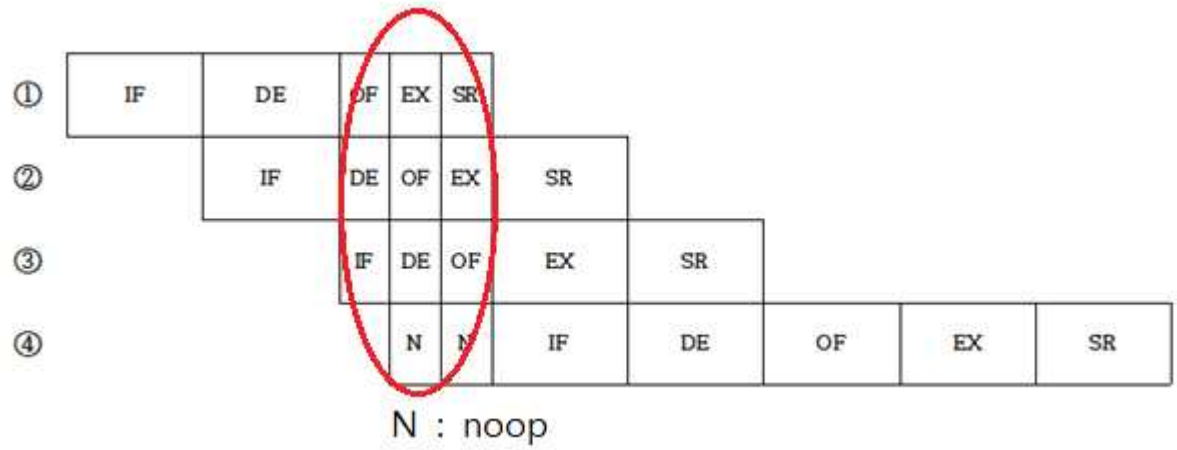
명령어의 주소를 저장하기 위해 새로운 NPC(New Program Counter)라는 레지스터를 만들고 자신보다 3클럭 늦게 수행될 명령어의 opcode를 해석해서 분기문인지 확인하기 위한 용도의 decoder를 추가해준다.

.





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.