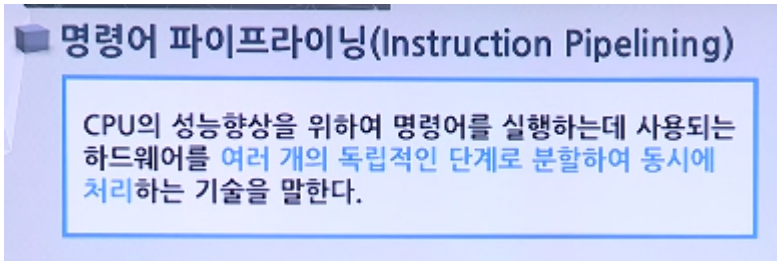
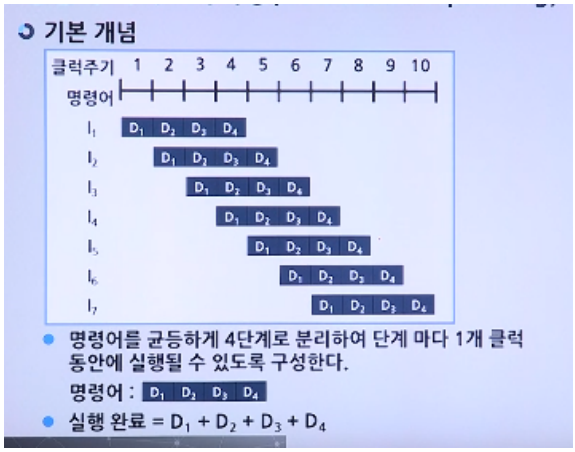
18_명령어 파이프라이닝

Created	@Aug 12, 2020 5:25 PM
Tags	

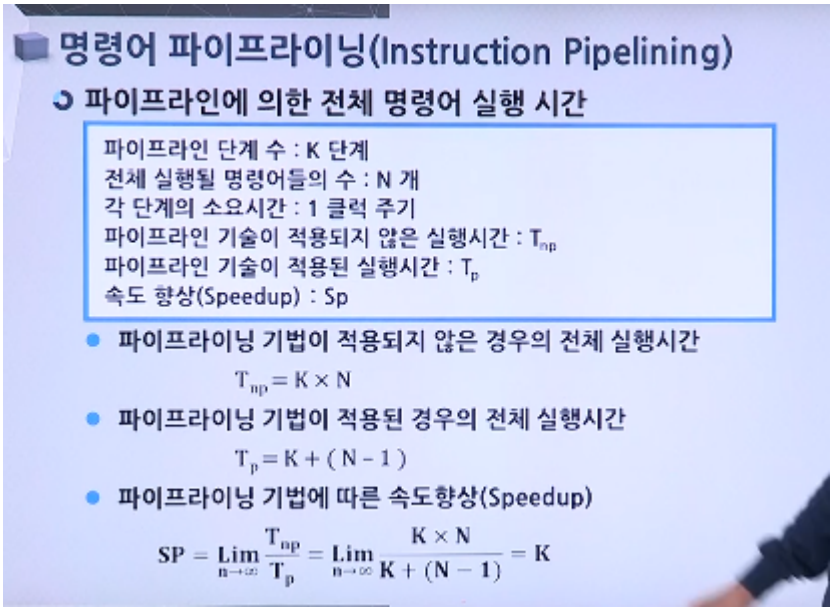


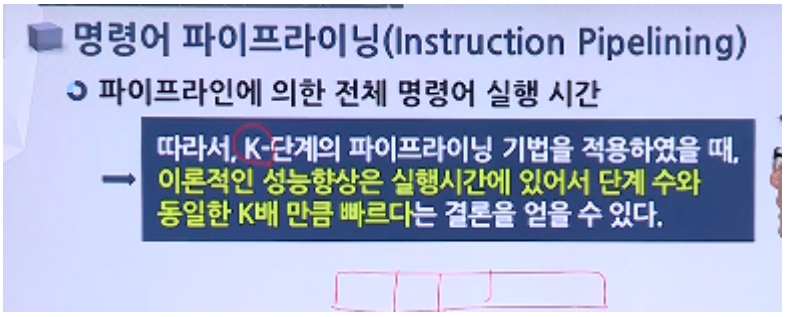


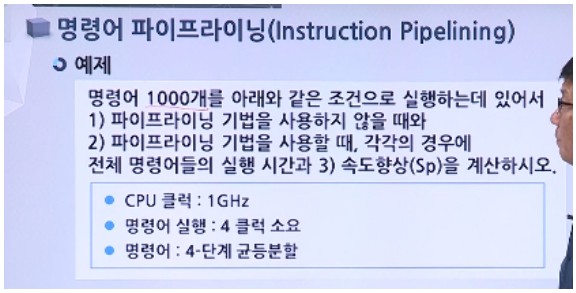
명령어가 처리되는 과정 인스트럭션 1, 2, 3...

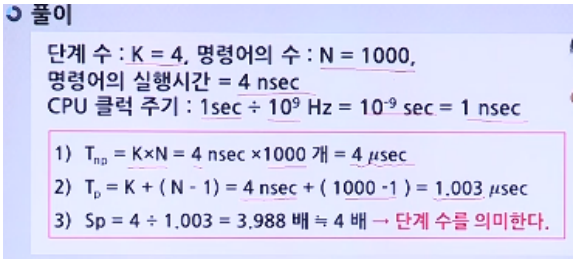
파이프라인 적용 시
처음에는 4개의 클락이 걸리지만, 그 다음부터는 하나씩 늘어난다 ..... //좋은건가? 이게 더 빠르다고.

적용 안함
하나당 4개씩 걸려서 4*N 명령 걸림


명령어 파이프라이닝(Instruction Pipelining)
파이프라인에 의한 전체 명령어 실행 시간
파이프라인 단계 수 : K 단계
전체 실행될 명령어들의 수 : N 개
각 단계의 소요시간 : 1 클럭 주기
파이프라인 기술이 적용되지 않은 실행시간 : T_{np}
파이프라인 기술이 적용된 실행시간 : T_p
속도 향상(Speedup) : Sp
파이프라인 기법이 적용되지 않은 경우의 전체 실행시간
$T_{np} = K \times N$
파이프라인 기법이 적용된 경우의 전체 실행시간
$T_p = K + (N - 1)$
파이프라인 기법에 따른 속도향상(Speedup)
$SP = \lim_{n \to \infty} \frac{T_{np}}{T_p} = \lim_{n \to \infty} \frac{K \times N}{K + (N - 1)} = K$


명령어 파이프라이닝(Instruction Pipelining)
파이프라인에 의한 전체 명령어 실행 시간
따라서, K-단계의 파이프라이닝 기법을 적용하였을 때,
이론적인 성능향상은 실행시간에 있어서 단계 수와
동일한 K배 만큼 빠르다는 결론을 얻을 수 있다.


명령어 파이프라이닝(Instruction Pipelining)
예제
명령어 1000개를 아래와 같은 조건으로 실행하는데 있어서
1) 파이프라이닝 기법을 사용하지 않을 때와
2) 파이프라이닝 기법을 사용할 때, 각각의 경우에
전체 명령어들의 실행 시간과 3) 속도향상(Sp)을 계산하시오.
CPU 클럭 : 1GHz
명령어 실행 : 4 클럭 소요
명령어 : 4-단계 균등분할


풀이
단계 수 : K = 4, 명령어의 수 : N = 1000,
명령어의 실행시간 = 4 nsec
CPU 클럭 주기 : 1sec ÷ 10^9 Hz = 10^{-9} sec = 1 nsec
1) T_{np} = K×N = 4 nsec ×1000 개 = 4 μsec
2) T_p = K + (N - 1) = 4 nsec + (1000 -1) = 1.003 μsec
3) Sp = 4 ÷ 1.003 = 3.988 배 ≒ 4 배 → 단계 수를 의미한다.

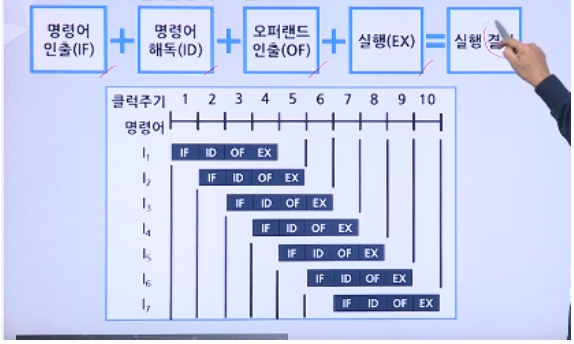
4-단계의 명령어 파이프라이닝 구성

D1 : 명령어 인출(IF)	다음에 실행될 명령어를 기억장치로부터 인출하는 단계
D2 : 명령어 해독(ID)	제어 유니트에서 해독기(Decoder)를 이용하여 명령어의 op-code 를 해석하는 단계
D3 : 오퍼랜드 인출(OF)	연산에 필요한 오퍼랜드의 데이터를 기억장치로부터 인출하는 단계
D4 : 실행(EX)	해독기에서 정해진 연산을 수행하는 단계

- 단계 분할 조건
  - 각 단계의 실행 시간은 동일해야 한다.
  - 분할된 단계의 가장 많이 소요되는 시간과 동일하게 설정되어야 한다.
  - 비효율적이다.

4-단계의 명령어 파이프라이닝 구성

- 4-단계별 실행과 흐름도



명령어 파이프라이닝 실행 장애(Hazard)

구조적 장애(Structural Hazards)

- 계속되는 명령어 수행에 있어서 한 명령어는 메모리부터 명령어를 가져오는 단계에 있고 또 다른 명령어는 메모리로부터 필요한 데이터를 가져오는 단계에 있다면, 메모리가 하나인 경우에 두 개의 서로 다른 명령어의 메모리 접근 충돌 때문에 동시에 처리할 수 없는 구조적 해저드가 발생한다.

데이터 장애(Data Hazards)

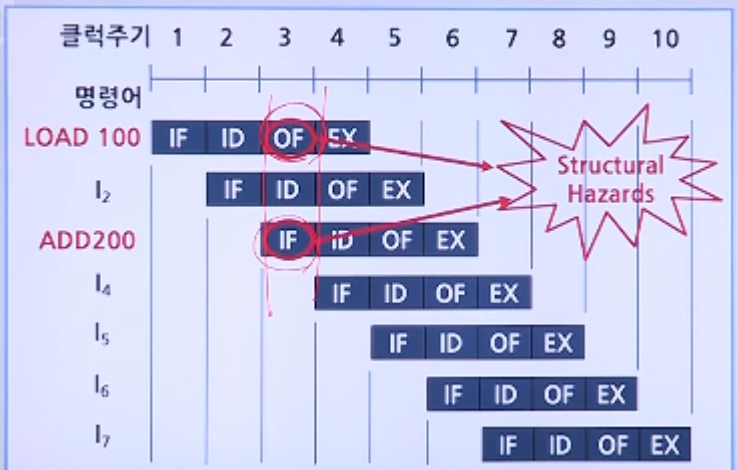
- 한 명령어 수행 결과가 다른 명령어의 연산에 사용될 때 파이프라이닝의 단계가 지연되어야만 하는 경우에 발생하는 것을 의미하는데, 이는 명령어 처리 결과 사이에 데이터의 종속성이 존재할 때 데이터 해저드가 발생한다.

제어 장애(Control Hazards)

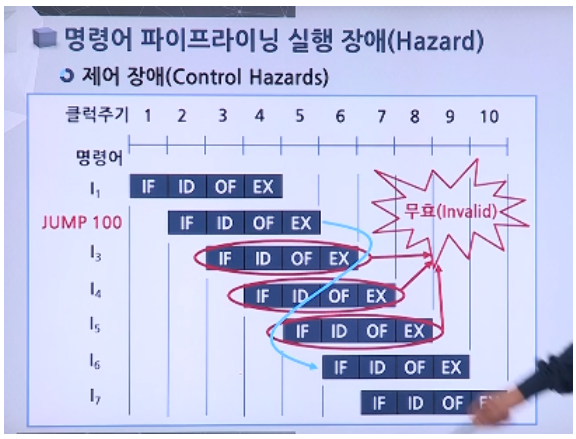
- 다른 명령어들이 실행들이 앞선 명령어의 결과값에 따라 필요하지 않을 때 발생한다. 주로 분기 명령어에서 이러한 경우가 발생한다.

명령어 파이프라이닝 실행 장애(Hazard)

- 구조적 장애(Structural Hazards)



한 단계씩 밀려서 진행되는 데이터 장애



▼ 스크립트

1. 다음은 이제 명령어 파이프라이닝. 이것은 컴퓨터의 성능향상을 위해서 사용되는 기법인데요, 그 파이프라이닝 하는 기법을 알아보도록 하겠습니다.
2. 여기 내용 보시면 "CPU의 성능향상을 위해서 명령어를 실행하는데 사용되는 하드웨어를 여러 개의 독립적인 단계. 여러 개의 독립적인 단계라는 이 의미. 즉 분할해서 동시에 처리하는 기술을 말한다.
3. 즉 명령어가 하나가 있으면 이 명령어들을 이렇게 분할을 한다는 의미가 되겠죠. 물론 여기서는 제가 4단계로 구분을 했는데요. 그래서 처음에 여기 처리 하고 그 다음, 그다음, 그다음 처리 하는 방식. 이렇게 4단계로 구분할때, 또 다른 명령어는 얘가 끝나고 나면 가져와서 또 이런식으로.. 그래서 명령어의 이 부분을 처리할

때, 다른 명령어에 또 이 부분을 처리하는 방식. 이런 단계적으로 이렇게 짤라가지고 연속적으로 처리하는 방식을 저희가 명령어 파이프라이닝 이라고 얘기하고 있습니다.

4. 기본개념은 여기 그림에 보시면 클락 주기가 1,2,3,4 이렇게 되어 있고, 그 다음에 여기 이제 명령어가 처리 되는 과정을 보여 주고 있는데, 이거는 인스트럭션 1, 2, 3 이렇게 의미하고 있습니다.
5. 이렇게 이게 하나의 명령어 인데, 명령어를 이렇게 D1, D2, D3, D4 4개로 나눈다는 의미가 되겠습니다.
6. 그래서 클락별로 여기에 첫 번째 클락에 이것 처리하고, 두 번째 클락에 D2 할 때, 인스트럭션2 라는 다음 명령어에 D1을 처리하고, 이렇게 진행을 하다보면 세 번째 클락에는 3개를 동시에 처리하게 되는 거죠. 그 다음에 네 번째 클락 되면 하나 둘 셋 넷. 그래서 이 상황에서 네 번째 클락이 다 끝나게 되면 비로서 명령어 하나가 끝나는 형태입니다.
7. 그러면서 인스트럭션 2번째 3번째 4번째도 일부분을 처리하는 형태. 이렇게 점진적으로 진행하다 보면 속도가 굉장히 빠르다. 그래서 이런 걸 우리가 파이프라이닝 기법 이라고 얘길 하고 있습니다.
8. 그러니까 파이프라이닝 기법을 쓰지 않는 경우에는 이 그림에서 보시면 인스트럭션 하나를 처리하는데 4클락 걸리고, 그 다음에 인스트럭션2 를 처리할 때는, 또 여기에 이어서 4개만큼 들어가겠죠. 여기가 이제 I2가 되겠죠. 그 기법을 적용 하지 않으면.. 그래서 지금 그림에서 보시는 바와 같이 그 다음에 또 3번째 인스트럭션. 이렇게 하다 보면 이렇게 처리하는 방식보다 이 방식이 훨씬 시간이 많이 걸린다는 걸 아마 그림에서 눈으로 확인하실 수 있겠습니다.
9. 그러면 이 명령어 파이프라이닝 방식을 적용하면 어느정도 빠른가? 즉 이렇게 한 개씩 처리하는 방식에 비해서, 즉 명령어 파이프라이닝 방식을 적용하지 않는 방식에 비해서 얼마나 빠른가? 하는 것이 가장 중요한 이슈가 되겠죠.
10. 그래서 보시는 바와 같이 명령어 하나 처리하는데 이 그림에서 보시면 4개 클락이, 그 다음에 이런 경우에 8클락. 2개 처리하는데 하지만 파이프라이닝을 적용하면 2개 끝나는데 이 하나 끝난 다음에 한 클락만 있으면 두개가 끝나게 되죠. 또 다음 명령어도 한 클락 더 가면 3개 끝나고 또 한 클락 더 가면 4개 끝나고, 이와 같이 파이프 라인을 적용하면 처음에 4개의 클락이 걸리지만 그 다음 명령어 부터는 하나, 그 다음에도 하나 이렇게 늘어난다는 것이죠.
11. 그래서 저희가 N개의 명령어가 있다고 보면, N개. 처음에 명령어가 4개 클락 걸리고 하나를 먼저 처리했으니까 제일 마지막에 걸리는 이 끝까지 걸리는.. 이 끝까지 걸리는 실행시간을 보면, 하나는 이미 4개를 썼고, 나머지 N 마이너스 1개는,

한 개 처리할 때마다 한 클락이 걸리니까 여기다가 N 마이너스 1 이정도의 시간이 걸린다는 것입니다.

12. 그리고 파이프라인 적용하지 않은 경우을 보면 하나당 4개씩 걸리니까 4 곱하기 N개의 명령이 걸리는 것이죠. 그래서 파이프라인을 적용했을 때 시간하고, 그렇지 않을 때 시간을 비교해서 저희가 어느정도 빠르다 이렇게 얘기하는 것이 바로 이 파이프라이닝이라는 것입니다.

13. 이제 내용을 한번 또 세부적으로 살펴 보기로 하죠. 여기 써 있는 내용은 "명령어를 균등하게 4단계로 분리하여 단계마다 1개 클럭동안에 실행될 수 있도록 구성한다." 이미 제가 설명을 드렸죠? 그 다음에 명령어는 이미 이렇게 구성이 되어 있다. 즉 4단계로 균등하게 분할이 되어 있고, 그래서 한 명령어를 실행하는 데는 D1, D2, D3, D4 이렇게 4개를 해야 하나가 끝난다 이런 의미가 되겠습니다.

14. 그래서 이 내용은 지금 파이프라이닝을 적용했을 때 시간이 어떻게 되나 한번 우리가 수식으로 살펴 보는 그런 내용이 되겠습니다.

15. 그래서 파이프라인 단계수를 예를 들어 K단계. 앞에서는 제가 4단계라고 말씀을 드렸었죠. 이걸 일반적으로 표현 했을 때 K단계 라고 하고, 전체 실행될 명령어들의 수. 그것을 우리가 N개. 이렇게 표현하고, 그 다음에 각 단계마다 소요 되는 것을 이제 1클락 주기. 이렇게 잡고.. 파이프라인 기술이 적용되지 않은 실행시간을 우리가 그 변수로 Tnp 이렇게 쓰고 즉 논(NON)파이프 라인 이라는 뜻입니다. 그 다음에 파이프라인을 적용했을 때 시간은 Tp. 파이프라인을 적용했다. 이런 뜻이 되겠죠. 그랬을 때 속도 향상은 어떻게 하느냐? 저희가 보통 Speedup 이라고 얘길 하는데 기호로써 Sp 라고 이렇게 쓰고 있습니다.

16. 그래서 계산을 해보면 앞에서 제가 예를 들어 설명 드린 것처럼 파이프라이닝 기법이 적용되지 않은 경우, 그런 경우에는 명령어가 N개니까 각 명령어 마다 단계가 K니까 단계에 1클락씩 걸린다고 그랬죠? 그래서 Tnp는 K 곱하기 N이 되겠고, 그 다음에 이제 파이프라이닝이 적용된 경우에는, 첫 번째 명령어가 끝나는 시간이 즉 단계수 만큼 걸리니까 K가 되겠고, 명령어 수가 N개니까 첫 번째 것을 제외한 것은 N 마이너스 1. 그래서 K 플러스 N 마이너스 1 이렇게 쓸 수 있습니다.

17. 그래서 파이프라이닝 기법에 따른 속도향상 Speedup을 계산해 보면, Speedup은 그러니까 파이프라이닝 적용하지 않았을 때하고, 파이프라인 적용했을 때를 이제 비교해보는 것이죠. 그래서 비교하면 즉 K 곱하기 N을 K 플러스 N 마이너스 1로 나누어 주면 이 K가 된다. 이것은 여기 N이 이제 무한대로 되어 있다.. 이 말은 명령어가 무수히 많다 이런 뜻이 되겠습니다. N개 라는 게 엄청나게 많은 명령어를 처리하게 되면 결국 이 값은 K값이 나온다. 이런 뜻이.. 이것은 결국은 무슨 말이냐? 이 K값 하고 저희가 말한 이 K 값 하고, 즉 이것은 단계를 말하겠죠. 단계. 그래서 이 파이프라이닝을 적용하게 되면 일반적으로 이렇게 수식적으로 이

것은 이론적이 되겠죠. 실질적이기 보다는 이론적으로는 그 단계수 만큼 빠르다. 이렇게 저희가 통상적으로 얘길 하고 있습니다.

18. 그래서 지금까지 설명드린 이 파이프라이닝에 대한 전체 실행시간을 결론 지어서 얘기하면, 따라서.. K단계의 파이프라이닝 기법. 즉 K단계 한 명령어를 이렇게 한 명령어 길이가 있으면 이것을 분할하는데 이것을 이 수가 즉 K개라는 뜻이 되겠죠.

19. K단계의 파이프라이닝 기법을 적용했을 때, 이론적인 성능향상은 실행시간에 있어서 단계수와 동일한 K배 만큼 빠르다. 라는 결론을 얻는 것이죠. 여기 말씀드린 것처럼 이것은 이론적인 성능향상 입니다. 실질적인 것은 여러가지 원인 때문에 K배의 성능이 나오지 않는다는 것이죠. 그래서 파이프라인을 적용하면서 그 기술들이 이런 이론적인 성능에 가깝게 성능향상이 될 수 있도록 많은 기법들을 개발하고 있는 현실입니다.

20. 이 예제는 앞에서 제가 말씀드린 그 파이프라이닝 기법을 적용했을 때 과연 얼만큼 속도 향상이 되는가를 한 번 확인해 보는 그런 내용이 되겠습니다.

21. 예를 들어 명령어 1000개. 1000개가 있다. 아래와 같은 조건으로 실행하는데 있어서, 1번 파이프라이닝 기법을 사용하지 않을 때와, 2번 파이프라이닝 기법을 사용할 때 각각의 경우에 속도향상을 한번 비교해 보자.. 이런 얘기가 되겠죠. 그리고 조건은 CPU 클럭이 1GHz, 그 다음에 명령어 실행. 이것은 4클럭이 소요된다. 단계를 4단계로 나누면 1단계에 1클럭이 소요된다.. 이런 의미도 있습니다. 그래서 명령어는 우리가 4단계로 균등분할 했다.

22. 이렇게 이제 가정을 했을 때, 여기서 단계수 K는 4다. 그 다음에 명령어의 수는 1000. 그래서 명령어 실행시간. 이게 이제 명령어 실행하는데 시간이 4nsec. CPU 클럭주기는 이제 이정도니까 1초를 10에 9승으로 나눠주면 10의 마이너스 9승 sec. 즉 1nsec. 즉 CPU 클럭주기가 1나노라는 거죠.

23. 그래서 앞에서 제가 설명드린, 이 파이 프라이닝 하지 않았을 때.. 논파이프라이닝 이죠. 그때 계산을 해보면 K 곱하기 N. K는 4고 명령어수는 1000개니까 곱하면 4천. 4천 나노니까 단위 환산을 하면, 나노는 이 μsec. 그래서 4 μsec가 되는 것이고.

24. 그 다음에 파이프라이닝을 적용하게 되면 K. 처음에 걸리는 시간이 K 이고 나머지는 하나씩 추가가 되니까, K 플러스 N 마이너스 1이 되는거죠. 그래서 4 nsec 에다가 1000 에서 1뺀 값을 더하면 1.003 μsec. 그래서 이건 이제 SP. 즉, 스피드업이죠? 이거를 이제 계산을 해보면, 즉 4를 1.003 으로 나누면 3.988배. 즉 약 4배 정도 된다는 뜻입니다. 즉 4배라는 것은 저희가 단계수를 4로 설정을 했기 때문에 거의 4에 가깝게 스피드업이 된다. 그래서 이 명령어 파이프라이닝을 적용하면 단계수만큼 빨라진다. 이렇게 이제 말씀드릴 수 있겠습니다.

25. 4단계 명령어 파이프라이닝 구성. 이걸 이제 하나 하나씩 저희가 살펴 보면, 그럼 단계를 어떻게 나누냐? 그런 뜻이 되겠죠.
26. 그래서 D1은 저희가 앞에서 말씀드린 명령어 인출을 해야 되니까 저희가 명령어 인출 단계. 이렇게 얘기를 하고 IF라고 표현을 하고요.
27. 그 다음에 D2는 명령어 해독. 즉, 제어 유닛에서 그 해독기. 디코더를 이용해서 명령어의 OP code를 알아내는 그런 과정이 되겠고.
28. 그 다음에 3번째는 오퍼랜드 인출. 즉 오퍼랜드 패치가 되겠죠. 연산에 필요한 오퍼랜드의 데이터를 기억장치로 부터 인출하는 그런 단계.
29. 그 다음에 마지막으로 D4는 실행하는, 즉 해독기에서 정해진 연산을 수행하는 그런 단계. 이렇게 4단계로 구분을 하는 거죠.
30. 그런데 이렇게 단계를 나눌 때 조건들이 있습니다. 첫 번째 각 단계의 실행시간은 동일해야 한다. 이 말은 명령어를 이렇게 4단계로 나눴을 때, 이렇게 4단계로 나눴을 때.. 이렇게 4단계로 나눴을 때, 예를 들면 여기는 1nsec 여기는 2nsec 여기는 1nsec 또 여기는 2nsec 이렇게 되면 안된다 .이런 얘기죠. 그러니까 1 2 1 2. 즉 단계마다 서로 다른거죠. 그러면 이렇게 되었을 때 파이프라인을 적용할 수 없다는 것이죠.
31. 그러면 어떻게 해야 되느냐? 이 분할된 단계. 여기 보시면 분할된 단계의 가장 많이 소요되는 시간과 동일하게 설정되야 한다. 이 말은 지금 1보다는 2가 더 많이 걸리니까 ,이 1도 2로 고쳐야 하고 이 1도 2로 고쳐서, 그래서 4개가 모두 다 동일한 시간이 걸리도록 해줘야 한다. 그런 얘기가 되겠습니다.
32. 그럼 제가 말씀드린 것처럼 1 2 1 2 이렇게 걸리는 시간을 다 더해 보면 6 인데, 이거를 제일 긴 거에 맞추다 보니까 8로 늘어나는 거죠. 그래서 저희가 이것을 비효율적이다. 이렇게 얘기할 수 있습니다. 그래서 이 단계는 가급적이면 짧게 걸리게 끔 만들어 주는 것이 우선적인 조건이 되겠습니다. 그래서 짧게 걸리는 조건에 다 맞도록 세분화시키는 그런 형태로 만들어야 된다는 것이죠.
33. 흐름도를 보시면, 아까 말씀 드린 것처럼 명령어 인출하고 해독하고 오퍼랜드 인출하고 마지막으로 실행해서 우리가 결과를 얻는 이런 과정입니다.
34. 클럭 주기가 1 2 3 4 진행될 때마다 명령어 하나 여기서 끝나고, 그 다음에 2개 끝나고 3개 끝나고. 이런 식이죠? 그러면 지금 제가 여기 줄을 그어 놓은 게 이 칸이 다 동일하게 맞아야 된다는 것이죠.
35. 그러니까 이것이 인스트럭션 인출이나 패치나 해독이나 또 오퍼랜드 인출 이런 것들이 시간이 동일하지 않으면 이런 표를 형성할 수 없다는 것이죠. 그래서 중간이 길어지면 이 밑에 있는 그 명령어들이 영향을 받게 되니까, 반드시 이 4개는

똑같해야 된다. 그 단계 걸리는 시간이. 그것을 정확하게 표현해 주는 그림이라고 보시면 되겠습니다.

36. 그러면 이제 이 명령어 파이프라이닝은 제가 말씀드린 것처럼 이론적인 것은 그 단계수 만큼 성능 향상이 된다.. 이랬죠. 단계수. 이만큼 성능 향상이 된다! 그러면 얼마나 좋겠습니까? 그렇지만 결코 그렇게 안되는 이유들이 있습니다.

37. 지금 제가 설명드리는 것은 무엇이냐면, 이런 명령어파이프라이닝에 실행을 하는데 있어서 과연 장애. Hazard가 뭐냐? 저희가 한 세가지로 구분 할 수 있는데요. 하나씩 한번 살펴 보기로 하죠.

38. 이 구조적 장애 라는 것이 먼저 있는데, 이것은 내용에 계속되는 명령어 수행에 있어서 한 명령어에는 메모리로 부터 명령어를 가져오는 단계. 또 다른 명령어에는 메모리로 부터 필요한 데이터를 가져오는 단계. 서로 다른거죠? 명령어를 가져오고, 데이터를 가져오고. 이런 경우에 메모리가 하나에요? 두 개가 아니고 하나. 하나인 경우에 두 개의 서로 다른 명령어 때문에 메모리 접근 하는데 있어서 충돌. conflict가 발생한다는 겁니다. 그래서 충돌이 발생하니까 동시에 처리할 수가 없어요. 그럼 하나는 반드시 기다려야 됩니다. 딜레이가 되는 거죠. 이런 것들을 우리가 구조적 장애. 이렇게 얘기합니다. 구조적 해저드가 발생한다. 이렇게 얘길 하고 있는 것이죠.

39. 또다른 장애가 있는데, 그것은 이제 데이터 장애. 이것은 뭐냐하면 한 명령어의 수행결과가 다른 명령어의 연산에 사용될 경우. 즉 미처 결과가 나오기도 전에 다른 연산에 사용된다는 뜻입니다. 그래서 이런 경우에 파이프라이닝의 단계가 지연되어야만 하는 경우. 발생하는 것을 의미하는데, 이는 명령어 처리결과 사이에 데이터 종속성. 즉, dependency가 발생할 때 그런 것들이 있을 때 데이터 해저드가 발생한다. 이런 걸 우리가 데이터 해저드라고 얘길 합니다. 이 내용은 좀 있다가 제가 예를 통해서 다시 설명을 드릴 것입니다.

40. 그 다음에 마지막으로 제어 장애 Control 장애라는 것은, 다른 명령어들이 실행들이 앞선 명령어의 결과값에 따라 필요하지 않을 경우 발생하는 것. 주로 이제 분기 명령어에서 이러한 경우가 발생하게 됩니다. 즉 분기 명령이라는 건 이제 JUMP 이런 것들이 있겠죠. JUMP. 이런 명령에서 발생을 하게 되는 거죠. 즉 명령을 계속 이렇게 인출해 왔는데 단계별로.. 여기서 JUMP가 되면 미리 인출한 것이 뒤에 인출한 것, 실행은 여기서 이루어 지기 때문에 미리 인출한 게 소용이 없다. 그런 뜻이 되는 것입니다.

41. 그러면 이제 이런 구조적 장애 라던지, 데이터 장애 또는 제어장애. 이것에 대한 예를 한번 살펴 보도록 하겠습니다.

42. 이런 순서대로 제가 말씀대로 순서대로 명령어가 쭉 처리가 되면, 정말 이론적인 그런 성능향상이 나오겠는데.. 지금 이걸 보시면, 예를들어 첫 번째 명령이 LOAD

100 이다. 그러면 그다음에 ADD는 200 인데 여기서 이제 명령어을 만나게 되는데, 여기 보시면 LOAD 100 이라는 건 100번지에 있는 내용을 가져와라. 즉 오퍼랜드 패치할 때 이 시간하고, 그 다음에 ADD 200 그러면 200을 가져.. 200번지에 있는 내용을 가져와서 accumulator에 더해야 되는 것입니다.

43. 여기서 지금 가져오고 여기도 가져오고 이럴 때 보면 동시에 어떤 현상이 발생하느냐면 메모리를 접근해야 되는 현상이 벌어지게 됩니다. 그러면 이럴 때 얘를 먼저 할꺼냐? 얘를 먼저 할꺼냐? 양자간에 선택해서 하나는 기다려야 됩니다. 그래서 발생하는 것이 제가 스트럭쳐. 구조적장애라고 얘기 했죠.

44. 그래서 이런 식으로 명령어가 하나 데이타 명령어 이런 걸 가져올 때 생기는 것이 바로 구조적 장애라고 제가 말씀을 드렸습니다. 메모리가 물론 2개면 데이터를 위한 메모리가 있고, 명령어을 위한 메모리가 따로 있고 이러면은 그런 문제를 해결할 수 있지만 결과적으로는 그 cost가 올라간다는 그런 얘기가 되겠죠.

45. 그 다음에 이 부분이 이제 데이터 장애에 관한 데이터 해저드라고 그랬죠? 장애에 관한 예를 보여주는 것인데, 여기 보시면 지금 ADD R1, R2 이렇게 되어 있습니다. 저 뜻은 지금 R1하고 R2하고 더해서 R1에다 결과를 놔라.. 이런 뜻이 되겠고, SUB R3, R1은 그 R3에서 R1을 뺀 결과를 R3에다 놔라. 이런 뜻이 되겠습니다.

46. 그래서 두 개를 가만히 보시면 ADD라는 명령어는 즉, R1 이라는 결과를.. 최종적인 결과를 얻는 데에는 이 실행단계.. 여기서 얻게 되는 것이죠. 그런데 그 밑에 연산에서 보시면, SUB R3, R1에서 R1은 오퍼랜드값을 이제 가져오게 되는 것입니다. 그러니까 여기서는 결과가 막 진행되고, 결과가 나올려고 하는데 여기서는 가져온다는 뜻이 되죠. 그래서 여기서 결과가 나온 값이 아닌 원래 R1에 있던 값을 그냥 가져와서 R3에서 빼버리니까 결과적으로는 R3가 잘못된 결과를 얻게 되는 것입니다.

47. 그래서 이 얘기는 지금 이 텀에서.. 이 클럭에서 서로 데이터가 종속관계에 있기 때문에, 올바른 데이터를 저희가 구할 수 없는 것이죠. 그래서 이런 경우 우리가 이거를 데이터 장애라고 하고 있습니다.

48. 이런 데이터 장애를 극복하기 위해서는 지금 인스트럭션을 여기서 인출해서 이렇게 실행한 뒤, 4 클럭에 실행하는 거를 그 장애를 없애려면, 이 두 번째는 이쪽에서 인스터럭션을 패치하는 게 아니라, 여기서부터 시작해서 이렇게 시작을 해야 아마 그 장애를 해소할 수 있을 것입니다. 그래서 이 빈공간에다가 저희가 stall 이라는 이런 거를 하나 집어 넣게 되는 것이죠.

49. 결과적으로 보면, 차근차근 한 단계씩 이렇게 밀려서 진행되는 것이 이런 장애 때문에.. 데이터 장애 때문에, 하나가 밀려 들어가서 지연이 되는.. 이런 형태로 나타나는 걸 저희가 이제 데이터 장애라고 하고 있습니다.

50. 그러면 마지막으로 제어 장애는 뭔가? 한번 살펴 보도록 하겠습니다.

51. 이 제어 장애는, 여기서 JUMP가 명령어가 실행되는데, 여기서 이제 실행될 때 JUMP니까 JUMP 100 그것은 프로그램 카운터를 100으로 바꿔줘라 그 얘기죠. 그러면 여기서 알 수 있는데 그 전에, 알기 전에 그 JUMP 라는 명령을 디코딩할 때 그 다음 명령어를 인출하고 ,또 그것을 오퍼랜드 어쩌고 하기 전에 또 그 다음 명령어를 또 인출하게 됩니다.
52. 근데 실질적으로 이 내용을 보면 100번지로 가라. 그 소리는 그 밑에 있는 명령어을 실행 안 하겠다는 뜻이거든요. 그러니까 실행 안 하고 그냥 100번지로 가니까 결국은 이 위치에서 100번지에 있는 명령을 인출해야 되니까 여기에 있는 게 유효하다. 즉 이 3개의 명령은 무효가 된거죠. 그래서 이 사이에는 전부 기다려야 되는.이 stall라는.. 기다리는 표현으로 저희가 stall 이란 걸 쓰는데, stall을 여기다 집어 넣야 되는 것이죠.
53. 그리고 그전에 제가 설명드린 이 화면도 이런 문제가 생겼을 때, 즉 여기에 결과값을 얻기 위해서는 정확한 결과을 얻기 위해서는 실행하는 단계에서 이 명령어도 하나씩 밀어야 되는 이런 형태가 되는 거죠.
54. 그러니까 이럴 때도 마찬가지로 여기도 stall를 넣어야 됩니다. 그러니까 stall를 넣다는 얘기는 여기서 이렇게 stall를 여기도 마찬가지고.. 이 화면도, 이런 해저드 때문에 stall 집어 넣야 된다는 거죠. 결국 단계적으로 수행이 안되고 하나씩 밀어져야 되는 이런 형태 이런것들이 전부 해저드 라고 저희가 얘길 하고 있습니다.
55. 그래서 이 데이터 해저드나 이 control hazards 이런 문제들을 우리가 해결하기 위해서 명령어 파이프라인 기법에는 여러가지 그 기법들을 쓰고 있습니다.
56. 예를 들면 이 제어 장애 에서는 명령어를 실행할 때 앞에 선인출되는 다음에 인출되는 그런 명령어들이 무엇인가를 과연 볼 수 있는 그런 버퍼들을 마련해 놓고 미리 본다든지, 이런 기법들을 사용해서 이런 제어 장애를 없애고, 또 데이터 해저드 같은 경우에는 그 결과값을 쓰기 위해서 그 프로그램 명령어 프로그램 순서를 바꾸다 든지. 즉 바꾼다는 것은 데이터에 종속성이 없어야 되겠죠.
57. 그런 경우에 실행순서를 바꾸므로써 파이프라인을 적용할 수 있게 만든다던지, 그 얘기들은 다 제가 말씀드린 이 stall 를 최대한 줄인다는 얘기가 되겠습니다. 그랬을 경우에 저희가 얻을 수 있는 게 이제 명령어 파이프라이닝의 이론적인 그 스피드업을 얻을 수 있다 이렇게 말씀드릴 수 있겠습니다.
58.