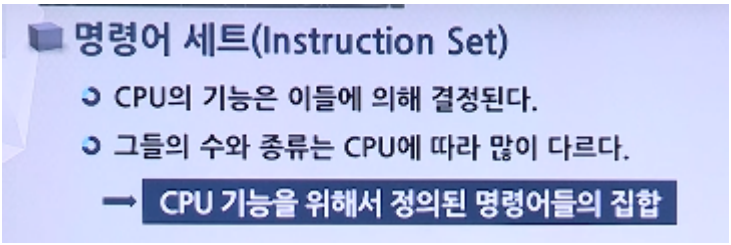
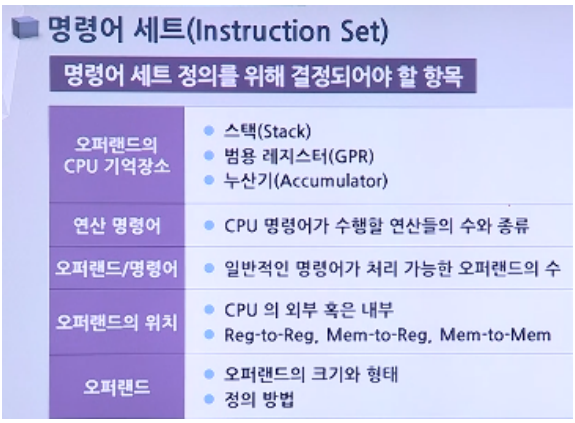
11강_명령어 세트





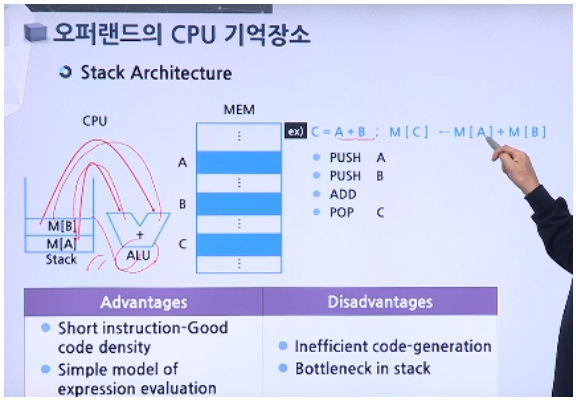
CPU에 **저장된 장소**가 Accumulator이면 이것은 Accumulator 중심으로 된 아키텍쳐이다.
Stack에면 Stack-Based Architecture.
범용 레지스터→ 범용 아키텍쳐
CPU를 위한 명령어 세트에 정의를 하기 위해서는 이것들이 결정이 되야 된다

연산 명령어의 수와 종류가 결정되어야 OP 코드의 비트수가 결정됨

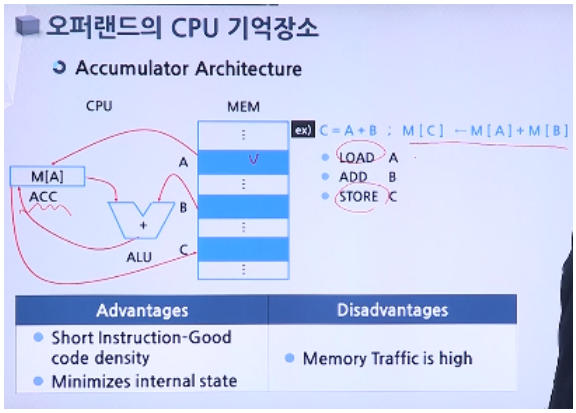
오퍼랜드 : 일반적인 명령어가 처리 가능한 오퍼랜드를 어떻게 할 것이냐.

오퍼랜드의 위치 : CPU 외부이냐 내부이냐, 레지스터와 레지스터 사이에서만 할 것이냐, 메모리-레지스터이냐, 메모리=메모리이냐.
중요하다! : 메모리를 많이 액세스 하게 되면 속도가 그만큼 떨어져서

Stack-Based Architecture 장점 : Instruction이 매우 짧다 == 코드 밀도(Density)가 아주 좋다 == 모델이 간단하다. / 단점 : 비효율 적이다. 스택을 많이 사용해서 병목현상이 생김

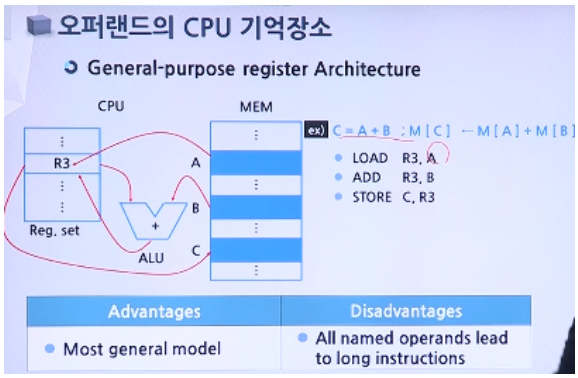


push add pop



Accumulator Architecture
load add store

장점 : 코드 밀도가 좋다, Internal state 중간 단계가 복잡하지 않다
단점 : 메모리에 계속 왔다갔다 해서 트래픽이 많이 발생한다.
== memory latency(메모리에서 발생하는 지연)이 많아진다
== CPU가 빨라도 메모리에 액세스를 계속해서 전체적인 속도는 느려진다.

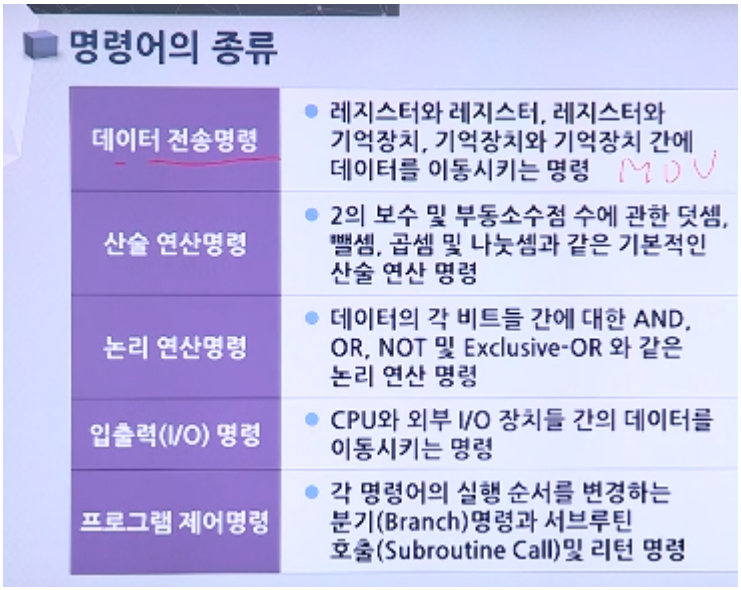


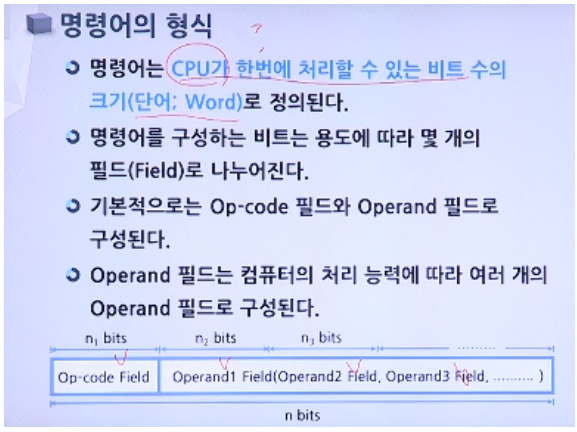
General-purpose register Architecture
장점 : 코드가 명확하고, 대부분의 컴퓨터 아키텍쳐들이 이런 모델을 사용한다(편리해서)
단점 : 모든 Operand들에 이름을 부여해야 해서 코드가 길어진다(long instruction)

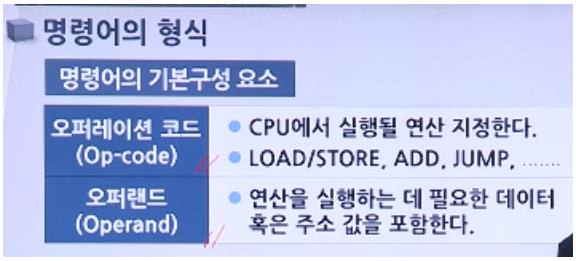
▼ 설명 이해 안감

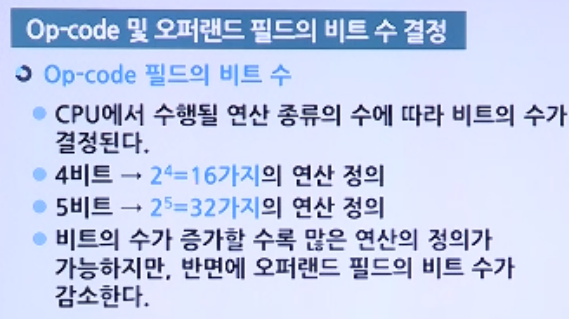
1. 다음은 이제 General Purpose Register를 근간으로 한 아키텍쳐다 이런 뜻이 되는데, 즉 여기 Register Set 에 R1, R2, R3 이런 레지스터들이 있습니다. 여기 아무거나 사용해도 되는 것이죠. 그래서 똑같은 이런 표현들을 저 레지스터를 생각해서 그냥 A에 있는 내용을 R3 에다 또는 R2, R4 아무데나 갖다 둬도 되는 것이죠.

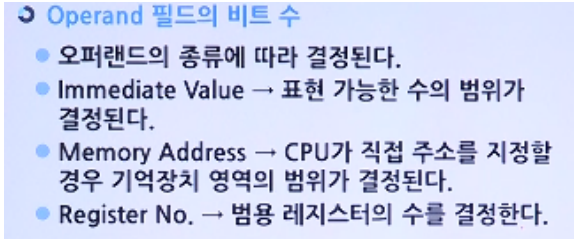
2. 그 다음에 ADD 하면 R3 하고, 그대신 여기 아까 ADD B 였지만 얘는 레지스터를 여러가지를 사용 할 수 있기 때문에 이제 R3를 여기다 쓰게 되는 것이죠. 그래서 이내용은 R3 + B를 해서 다시 R3에 두어라 이런 뜻입니다. 이것이. 그 다음에 이제 R3에 있는 내용을 메모리에 저장한다.

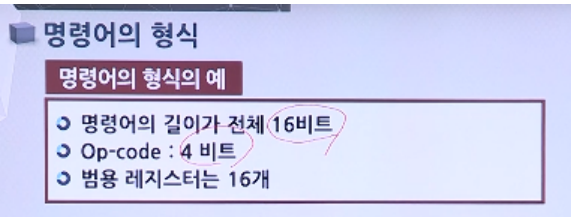




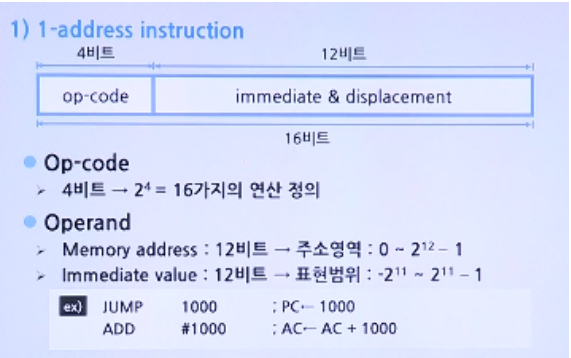




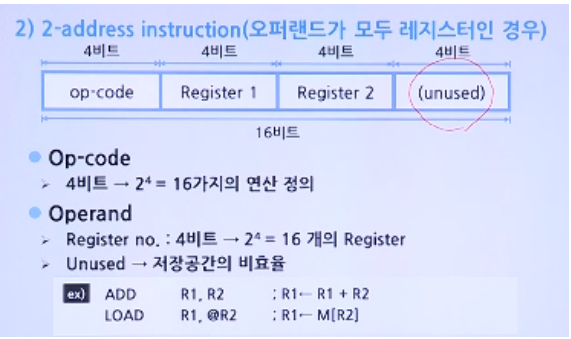




명령어 길이가 16 비트, OP code 4비트라 가정, 범용 레지스터는 16개 가정
이렇게 가정을 하고 다음과 같이~

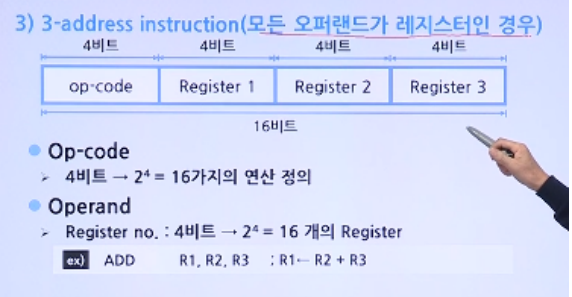


OP code가 4비트 이니까 16가지의 연산 정의 가능.
Memory address : 오퍼랜드가 12비트이고 (메모리 어드레스는 음수가 아니니까) 주소 영역이 0부터 2^12 - 1 까지 정의가능
Immediate value : 12비트 표현 범위 2의 보수~ 마이너스 표현해야 함.



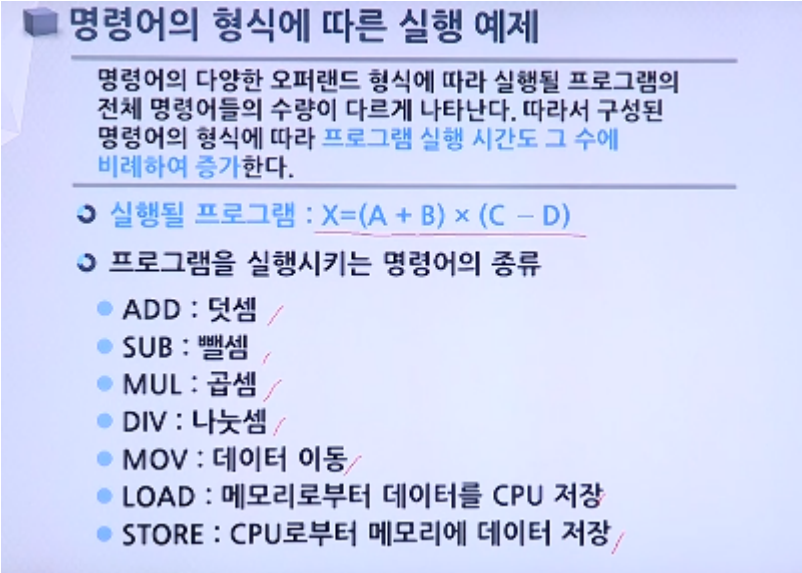
Immediate Value는 없고 오퍼랜드가 전부 레지스터인 경우...
OP code 4비트 전부 똑같이 16가지이고, 오퍼랜드는 레지스터만 결정하게 되니까 4비트면 16개의 레지스터 정의할 수 있게 됨

▼ 설명

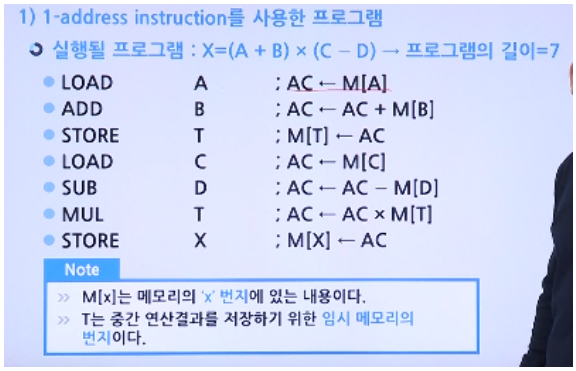


1. 그 다음 이제 3 Address Instruction 이것도 모든 오퍼랜드가 레지스터인 경우 여기는 Immediate Value 숫자 이런 것은 아무것도 없는 거죠. OP code 마찬가지 이고 오퍼랜드 레지스터이니까 전부 16개 구성할 수 있으며 3개니까 오퍼랜드가 1, 2, 3개가 되겠죠. 그래서 계산하면 R2 하고 R3 더해서 R1 두는 이런 형식입니다. 이걸 우리가 3주소 명령 형식 뭐 이렇게 얘기하는 것이죠.
2. 다음은 오퍼랜드 두 개는 레지스터인 경우에. 즉 이거하고 이거하고 2개만 레지스터 이고 나머지 또 Immediate Value나 Displacement로 쓸 수 있는 경우가 되겠죠. 그래서 다 똑같고 메모리 어드레스 4비트 이니까 주소 영역이 이것 밖에 안되겠죠. 이것 같고는 주소영역을 크게 표현할 수 없으니까 이럴 때 쓰는 것들이 Addressing Mode를 많이 쓰게 되는거죠. 여러가지 방식을.
3. 그러면 Immediate Value도 마찬가지로 숫자가 굉장히 작아지는 거죠. 그래서 이런 레지스터가 2개인 경우 이렇게 숫자가 오퍼랜드가 하나 들어가고 그 다음에 레지스터가 2개 들어가는 이런 형태. 그리고 여기에 지금 우물정자 있는 것은 숫자를 의미 하는데 1000b라고 해서 저희가 마이너스 8 이라고 쓴 것은 이거는 2에 보수죠. 그래서 이게 최상의 비트

가 1이니까 음수를 나타내니까 마이너스 8. 뭐, 이런 식으로 예를 볼 수 있겠습니다.



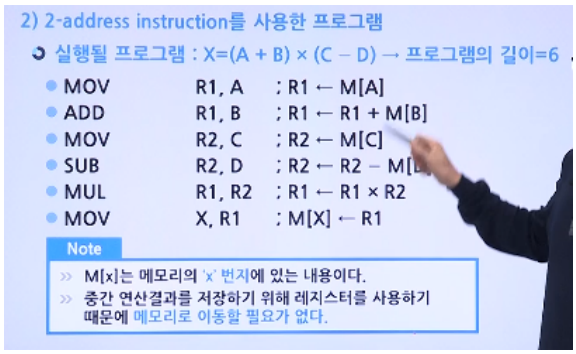
몇 주소냐에 따라서 프로그램의 길이가 달라짐. 3주소를 사용하는 프로그램이 속도가 점점 더 빠르다



▼ 1 설명

1. 첫번째 1주소 명령인 경우에 프로그램이 어떻게 되냐? 즉 A값을 로드해야 되니까 CPU로 가져와야 되니까 LOAD A 하겠죠. 의미는 이렇게 되고 그 다음에 A에 가져와서 Accumulator에 이미 있으니까 그걸 B하고 더해야 하니까 이런거에요. ADD B. 그 다음에 그 결과가 현재 Accumulator에 있습니다. 그러면 다음에 이 계산을 하게 되면 Accumulator 내용이 바뀌게 되니까 일시적으로 다른 데 저장을 하게 되겠죠.

2. 그래서 STORE T 라는 것은 다른 곳으로 보내고 다시 LOAD C 하고 그 다음에 빼기 SUB D를 하고 그 다음에 이제 2개를 곱해야 되는 것이죠. 그래서 지금 여기 C 마이너스 D 결과 이거죠. 이것은 지금 Accumulator 값을 가지고 있으니까 즉 곱할 내용 T로 보낸 거. 곱한 T를 가져와서 곱하면 결론은 지금 최종적으로 Accumulator에 들어가 있는 것이죠.
3. 원하는 것은 우리가 X라는 메모리에 집어 넣야 되니까 Accumulator 내용을 메모리 X로 보내는 이러한 과정 입니다. 그래서 이것을 꼭 길이로 따진다면 명령어수가 프로그램 길이에서 7개 이렇게 얘기를 하는 것이죠. 즉 1 주소 명령어는 똑같은 것을 했을 때 7이 걸렸다. 이렇게 볼 수 있는 것이죠. 여기 Note는 M라는 해서 x는 번지. 메모리에 있는 번지에 있는 내용을 말하는 것이고 T는 중간 연산결과를 저장 하기 위한 임시 메모리의 번지를 의미하고 있습니다.

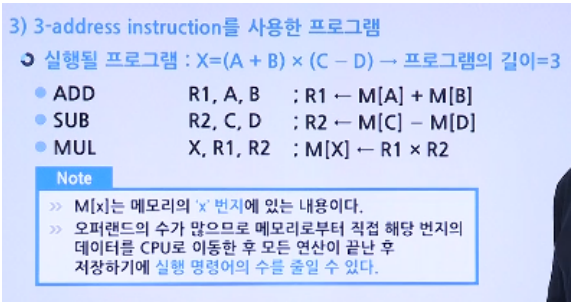


▼ 2 설명

1. 다음에 이제 2 어드레스 인스트럭션을 사용하면 어떻게 되나? 똑같은 연산입니다. 그러면 2 어드레스 이니까 오퍼랜드가 2개가 되겠죠. 그래서 먼저 A값을 R1에 집어 넣고 그 다음에 B를 가져와서 R1에다 더하니까 이런 모양 그래서 R1, B 똑같이 그러면 현재 R1의

결과를 가지고 있는 거죠. A 플러스 B는. 다음에 똑같이 C, D도 R2을 이용해서 하게 되면 R2에 이 연산결과가 들어가 있는 것이 되겠고. 이 2개를 곱하면 되는 거니까 MUL 해서 R1, R2 결과는 지금 R1안에 있습니다.

2. 우리가 최종적으로는 X쪽에 보내야 되니까 X에. MOV R1을 X에 보내는 이런 형태. 그래서 이것은 세어 보니까 1,2,3,4,5,6. 그래서 프로그램 길이 6개 이렇게 얘기하고 있죠. 그래서 얘는 2개의 오퍼랜드가 있으니까 중간 연산결과를 저장하기 위해 레지스터를 사용하니까 메모리로 이동할 필요가 없다. 이런 결론이 나오게 되는 것이죠.



▼ 3 설명

1. 그 다음에 이제 마지막으로 3주소 명령을 사용하면 어떻게 되나 똑같은 연산 입니다. 그러니까 이것은 오퍼랜드가 1,2,3개죠. 그래서 A 더하기 B, A 하고 B 하고 그 메모리에 내용을 더해서 R1에다 결과를 두고, 또 C 하고 D 해서 빼서 R2 에다 결과를 두고 R1, R2가 그 값을 현재 가지고 있기 때문에 곱하면 되니까 2개를. 즉 R1과 R2를 곱해서 X라는 메모리에 두어라. 결국 이렇게 하니까 프로그램의 길이가 3개로 그래서 아까 7개, 6개, 3개.

2. 이 얘기는 프로그램을 많이 실행하게 되면 이렇게 3주소를 사용하는 프로그램이 결론은 속도가 굉장히 빠르다. 이렇게 얘기할 수 있는 것이죠. 그래서 여기서 보여주는 예는 아마 그런 예. 상당히 빠르다. 라는 것을 보여주고 있는 것이죠. 마찬가지로 Note 내용은 앞에서 다 말씀 드린 것이고 결론적으로 얘기하면 오퍼랜드 수가 많으므로 메모리로부터 직접 해당 번지의 데이터를 CPU로 이동한 후 모든 연산이 끝난 후 저장하기에 이 말은 메모리에 지연되는 시간을 많이 없앴다. 결론적으로 이렇게 말씀드릴 수 있겠습니다.