컴퓨터 구조

2장 CPU의 구조와 기능4

안형태

anten@kumoh.ac.kr

디지털관 139호

CPU의 구조와 기능

4. 명령어 세트

명령어 세트

- **명령어 세트(instruction set)**
  - 어떤 CPU를 위하여 정의된 명령어들의 집합

- 명령어 세트 설계를 위해 결정되어야 할 사항들
  - **연산 종류(operation repertoire):** CPU가 수행할 연산들의 수와 종류 및 복잡도
  - **데이터 형태(data type):** 연산을 수행할 데이터들의 형태, 데이터의 길이(비트 수), 수의 표현 방식 등
  - **명령어 형식(instruction format):** 명령어의 길이, 오퍼랜드 필드들의 수와 길이 등
  - **주소지정 방식(addressing mode):** 오퍼랜드의 주소를 지정하는 방식

연산의 종류

- **데이터 전송:** 레지스터와 레지스터 간, 레지스터와 기억장치 간, 혹은 기억장치와 기억장치 간에 데이터를 이동하는 동작
  - MOVE, LOAD, STORE 등
- **산술 연산:** 덧셈, 뺄셈, 곱셈 및 나눗셈과 같은 기본적인 산술 연산들
- **논리 연산:** 데이터의 각 비트들 간에 대한 AND, OR, NOT, XOR 등 연산
- **입출력(I/O):** CPU와 외부 장치들 간의 데이터 이동을 위한 동작들
- **프로그램 제어:** 명령어 실행 순서를 변경하는 연산들
  - 분기(branch), 서브루틴 호출(subroutine call)

명령어의 구성요소들

- **연산 코드(Operation Code):** 수행될 연산을 지정
  - **[예]** LOAD, ADD 등

- **오퍼랜드(Operand)**
  - 연산을 수행하는 데 필요한 **데이터** 혹은 **데이터의 주소**
    - 각 연산은 한 개 혹은 두 개의 입력 오퍼랜드들과 한 개의 결과 오퍼랜드를 포함
    - 데이터는 CPU 레지스터, 주기억장치, 혹은 I/O 장치에 위치

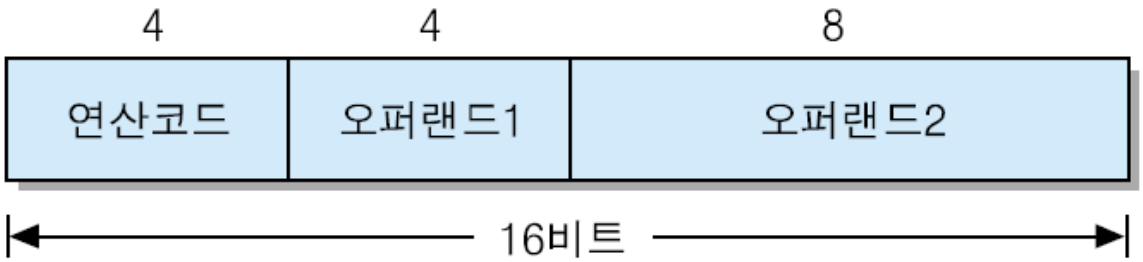
- **다음 명령어 주소(Next Instruction Address)**
  - 현재의 명령어 실행이 완료된 후에 다음 명령어를 인출할 위치 지정
  - 분기 혹은 호출 명령어와 같이 실행 순서를 변경하는 경우에 필요

명령어 형식

- **명령어 형식(instruction format):** 명령어 내 필드들의 수와 배치 방식 및 각 필드의 비트 수를 정의

- **필드(field):** 명령어의 각 구성 요소들에 소요되는 비트들의 그룹

- 일반적으로 **명령어의 길이 = 단어(word) 길이**
  - **[예]** 세 개의 필드들로 구성된 16-비트 명령어



명령어 형식의 결정에서 고려할 사항들

❑ 연산 코드 필드의 길이: 연산의 개수를 결정

- **[예]** 4 비트 → $2^4 = 16$ 가지의 연산 정의 가능
  - 만약 연산 코드 필드가 5 비트로 늘어나면, $2^5 = 32$가지 연산들 정의 가능 → 하지만, 다른 필드의 길이가 감소

❑ 오퍼랜드 필드의 길이: 오퍼랜드의 범위 결정

- 오퍼랜드의 종류에 따라 범위가 달라짐
  - **데이터:** 표현 가능한 수의 범위 결정
  - **기억장치 주소:** CPU가 오퍼랜드 인출을 위하여 **직접 주소**를 지정할 수 있는 기억장치 용량 결정
  - **레지스터 번호:** 데이터 저장에 사용될 수 있는 범용 레지스터의 개수 결정

오퍼랜드 필드 범위

- **[예]** 오퍼랜드1은 레지스터 번호를 지정하고, 오퍼랜드2는 기억장치 주소를 지정하는 경우
  - 오퍼랜드1: 4 비트 → **16 개의 레지스터** 사용 가능
  - 오퍼랜드2: 8 비트 → 기억장치의 주소 범위는 **0 ~ 255 번지**

- 두 개의 오퍼랜드들을 하나로 통합하여 사용하는 경우
  - 오퍼랜드가 2의 보수로 표현되는 데이터라면,
    - **표현 범위: -2048 ~ +2047**
  - 오퍼랜드가 기억장치 주소라면,
    - **2^{12} = 4096 개의 기억장치 주소들 직접 지정 가능**

오퍼랜드의 수에 따른 명령어 분류

0-주소 명령어(zero-address instruction)

- 연산에 필요한 오퍼랜드 및 결과의 저장 장소가 묵시적으로 지정된 경우
  - **[예] 스택(stack)**을 갖는 구조(PUSH, POP)
- 수식 계산시 후위 표기법(postfix) 활용
  - 피연자를 먼저 표시하고 연산자를 나중에 표시하는 방법
  - 중위 표기법:(A+B)×(C-D) → 후위 표기법: AB+CD-×

1-주소 명령어(one-address instruction): 오퍼랜드를 한 개만 포함하는 명령어

- 다른 한 오퍼랜드는 묵시적으로 AC가 됨
  - 연산 결과는 AC에 저장
- **[예]** ADD X ; AC ← AC+M[X]

오퍼랜드의 수에 따른 명령어 분류

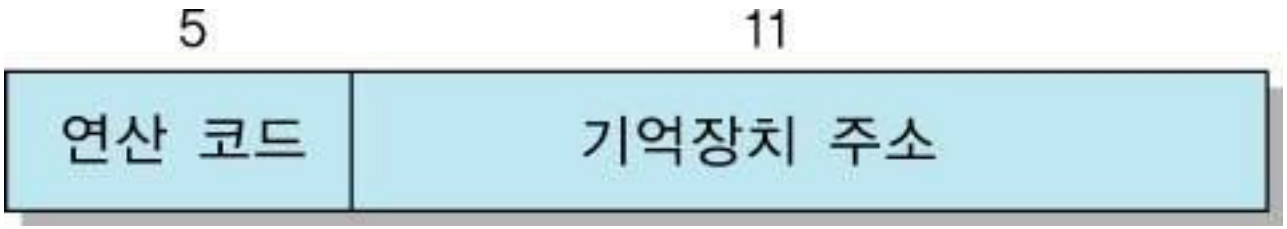
- **2-주소 명령어(two-address instruction):** 두 개의 오퍼랜드를 포함하는 명령어
  - **[예]** ADD R1, R2 ; R1 ← R1+R2
  - **[예]** MOV R1, R2 ; R1 ← R2
  - **[예]** ADD R1, X ; R1 ← R1+M[X]

- **3-주소 명령어(three-address instruction):** 세 개의 오퍼랜드들을 포함하는 명령어
  - **연산 후에도 입력 데이터 보존**
  - **[예]** ADD R1, R2, R3 ; R1 ← R2+R3

1-주소 명령어

- **[예제]** 길이가 16 비트인 1-주소 명령어에서 연산 코드가 5 비트인 경우의 명령어 형식을 정의하고, 주소지정 가능한 기억장치 용량을 결정하라.
  - 단, 주소가 지정되는 각 기억 장소에는 한 바이트씩 저장된다고 가정한다.

- **[풀이]**
  - 명령어 형식(instruction format)



  - 주소지정 가능한 기억장치 용량 : $2^{11} \times 1\text{Bytes} = 2048\text{Bytes}$

2-주소 명령어

- **[예제]** 2-주소 명령어 형식을 사용하는 16-비트 CPU에서 연산 코드가 5 비트이고, 레지스터의 수는 8 개이다.
  - **(a)** 두 오퍼랜드들이 모두 레지스터 번호인 경우의 명령어 형식은?
  - **(b)** 오퍼랜드 하나는 레지스터 번호이고, 나머지는 기억장치 주소인 경우의 명령어 형식은?

- **[풀이]**

(a) 두 개의 레지스터 오퍼랜드들을 가지는 경우

(b) 한 오퍼랜드는 기억장치 주소인 경우

3-주소 명령어 형식

- **[예제]** 다음의 명령어를 3-주소 명령어 형식을 나타내라.

ADD R1, R2, R3 ; R1 ← R2+R3

  - 여기서 명령어의 길이는 16 비트이며, 필요한 연산의 종류는 12개이며, 레지스터의 숫자는 10개이다.

- **[풀이]**

(a) 명령어 형식

명령어 형식이 프로그래밍에 미치는 영향

- **[예]** X=(A+B)×(C-D) 계산을 위한 어셈블리 프로그램 작성
  - 아래와 같은 니모닉(Mnemonic)을 가진 명령어들을 사용
    - **ADD** : 덧셈
    - **SUB** : 뺄셈
    - **MUL** : 곱셈
    - **DIV** : 나눗셈
    - **MOV** : 데이터 이동(레지스터와 기억장치 간)
    - **LOAD** : 기억장치로부터 데이터 읽어서 AC에 적재
    - **STOR** : AC의 내용을 기억장치로 데이터 저장

명령어 형식이 프로그래밍에 미치는 영향

- **[예]** X=(A+B)×(C-D) 계산을 위한 어셈블리 프로그램 작성
  - **1-주소 명령어**를 사용한 프로그램(단, M[*i*]는 기억장치 *i*번지의 내용, *T*는 기억장치 내 임시 저장 장소의 주소)
    - **LOAD A ; AC ← M[A]**
    - **ADD B ; AC ← AC + M[B]**
    - **STOR T ; M[T] ← AC**
    - **LOAD C ; AC ← M[C]**
    - **SUB D ; AC ← AC - M[D]**
    - **MUL T ; AC ← AC × M[T]**
    - **STOR X ; M[X] ← AC**
  - 프로그램의 길이 = **7**

명령어 형식이 프로그래밍에 미치는 영향

- **[예]** X=(A+B)×(C-D) 계산을 위한 어셈블리 프로그램 작성
  - **2-주소 명령어**를 사용한 프로그램(레지스터 R1, R2 활용)
    - **MOV R1, A ; R1 ← M[A]**
    - **ADD R1, B ; R1 ← R1 + M[B]**
    - **MOV R2, C ; R2 ← M[C]**
    - **SUB R2, D ; R2 ← R2 - M[D]**
    - **MUL R1, R2 ; R1 ← R1 × R2**
    - **MOV X, R1 ; M[X] ← R1**
  - 프로그램의 길이 = **6**

명령어 형식이 프로그래밍에 미치는 영향

- **[예]** X=(A+B)×(C-D) 계산을 위한 어셈블리 프로그램 작성
  - **3-주소 명령어**를 사용한 프로그램(레지스터 R1, R2 활용)
    - **ADD R1, A, B ; R1 ← M[A] + M[B]**
    - **SUB R2, C, D ; R2 ← M[C] - M[D]**
    - **MUL X, R1, R2 ; M[X] ← R1 × R2**
  - 프로그램의 길이 = **3**
  - **장점:** 프로그램의 길이가 짧아지며, 입력 데이터(A, B, C, D)가 보존됨
  - **단점:** 명령어 해독 과정이 복잡함

주소지정 방식

주소지정 방식(addressing mode)

- 명령어 실행에 필요한 오퍼랜드의 유효 주소를 결정하는 방식
  - 주소지정방식의 종류와 수는 다양하고, CPU마다 다름

다양한 주소지정 방식을 사용하는 이유:

- 일반적으로 명령어 비트의 수는 CPU가 처리하는 단어의 길이와 같도록 제한
- 제한된 수의 명령어 비트들을 이용하여, 사용자(프로그래머)가 다양한 방법으로 오퍼랜드의 주소를 결정하도록 해주며, **더 큰 용량의 기억장치를 사용**하기 위해

명령어 내 오퍼랜드 필드의 내용

- **기억장치 주소:** 데이터가 저장된 기억장치의 위치를 지정
- **레지스터 번호:** 데이터가 저장된 레지스터를 지정
- **데이터:** 명령어의 오퍼랜드 필드에 데이터가 포함

주소지정 방식

- 기호
  - ***EA*: 데이터가 저장된 기억장치의 실제 주소(유효 주소, Effective Address)**
  - ***A*:** 명령어 내의 주소 필드 내용(오퍼랜드 필드의 내용이 기억장치 주소인 경우)
  - ***R*:** 명령어 내의 레지스터 번호(오퍼랜드 필드의 내용이 레지스터 번호인 경우)
  - **(*A*):** 기억장치 A 번지의 내용
  - **(*R*):** 레지스터 R의 내용

주소지정 방식

- 주소지정 방식의 종류
  - 직접 주소지정 방식
  - 간접 주소지정 방식
  - 묵시적 주소지정 방식
  - 즉시 주소지정 방식
  - 레지스터 주소지정 방식
  - 레지스터 간접 주소지정 방식
  - 변위 주소지정 방식
    - 상대 주소지정 방식
    - 인덱스 주소지정 방식
    - 베이스-레지스터 주소지정 방식

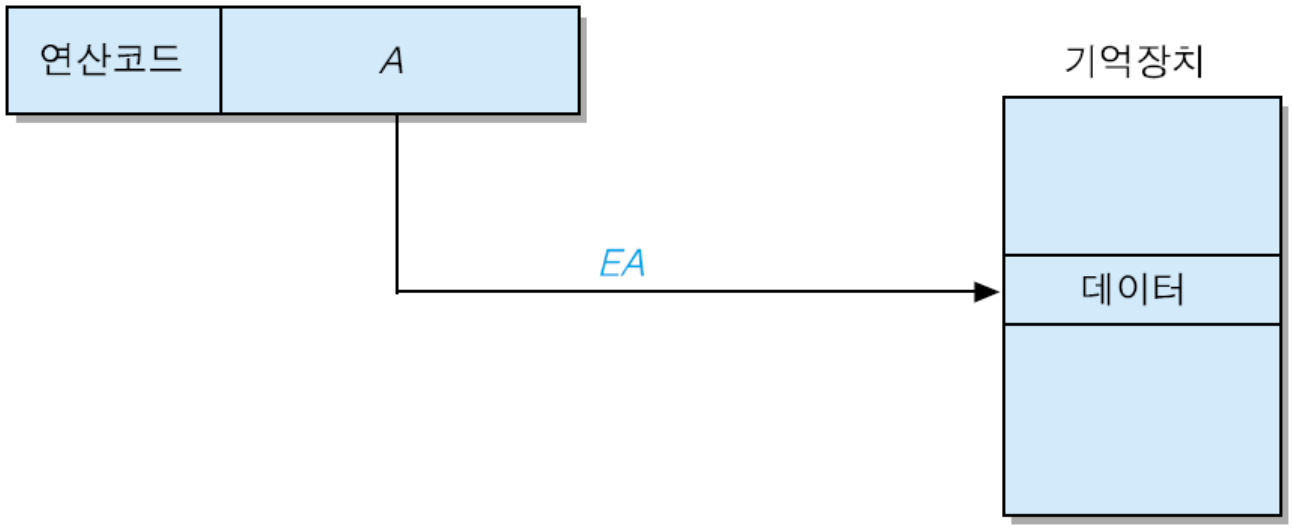
1) 직접 주소지정 방식

❑ 직접 주소지정 방식(direct addressing mode)

- 오퍼랜드 필드의 내용이 유효 주소(*EA*)가 되는 방식

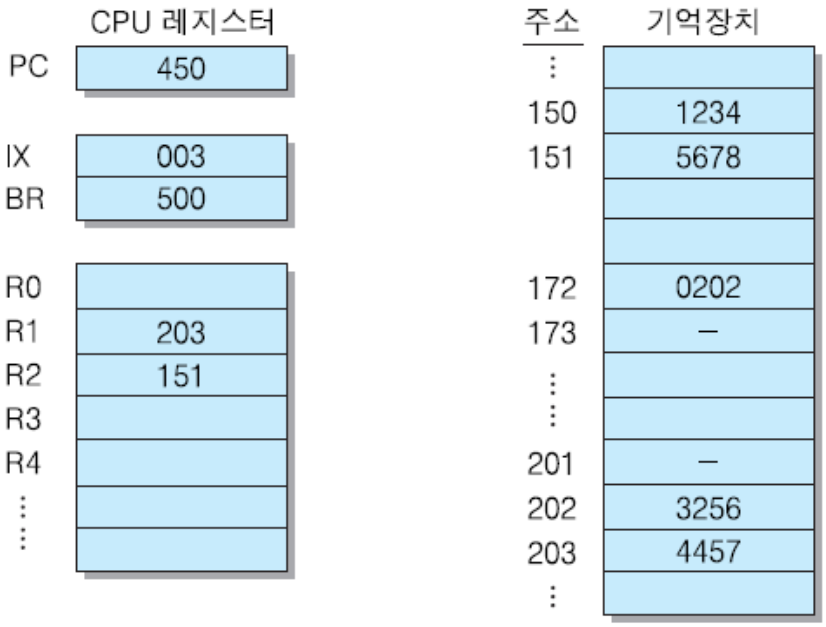
$$EA = A$$

- **장점:** 데이터 인출을 위하여 한 번의 기억장치 액세스만 필요
- **단점:** 연산 코드를 제외하고 남은 비트들만 주소 비트로 사용될 수 있기 때문에 직접 지정할 수 있는 기억 장치의 주소가 제한



1) 직접 주소지정 방식

- **[예제]** CPU 레지스터 길이와 주소지정 단위는 16비트로 가정
  - **[1]** 직접 주소지정 방식을 사용하는 명령어의 주소필드(*A*)에 저장된 내용이 150일때, 유효주소(*EA*) 및 그에 인출되는 데이터는?
  - **[2]** 명령어 길이가 16비트이고 연산 코드가 5비트라면, 이 명령어에 의해 직접 주소 지정 될 수 있는 기억장치 용량(Bytes)은 얼마인가?



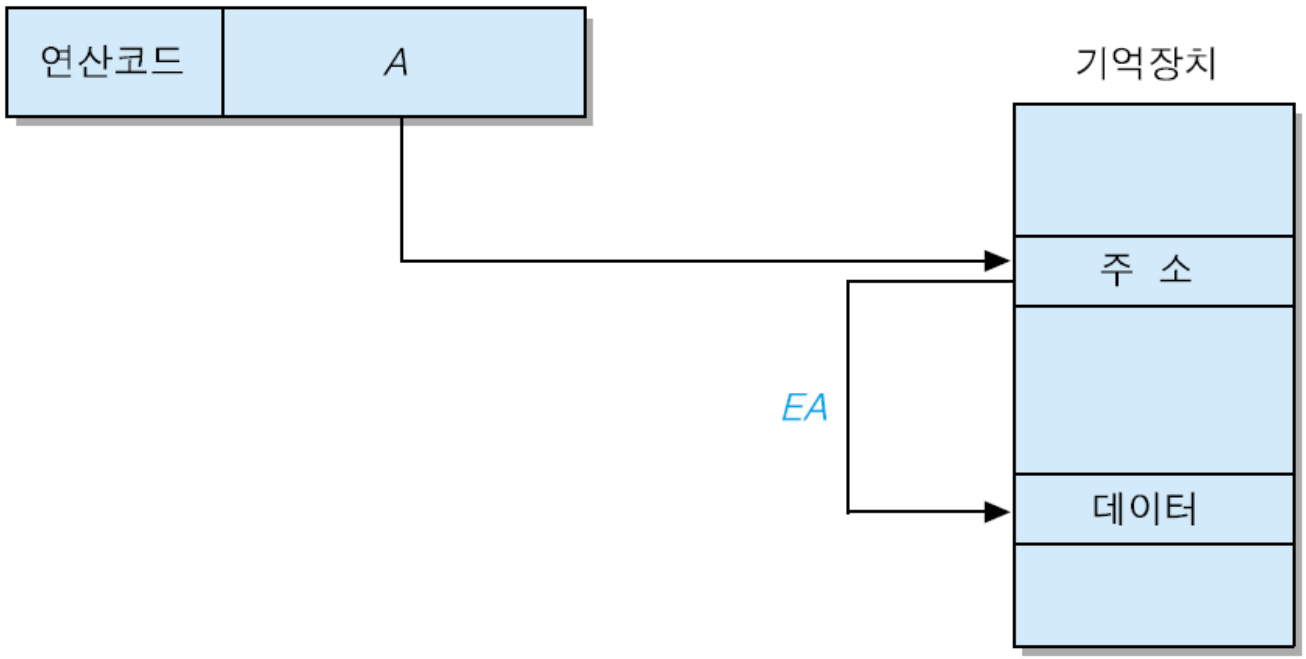
- **[풀이]**
  - **[1]** *EA* = 150이므로, 기억장치에 저장된 데이터 '1234'가 인출된다.
  - **[2]** 주소 필드가 11비트이므로, 총 2^{11}(2048)개를 기억 장소를 지정할 수 있다. 각 기억 장소에 저장되는 데이터의 비트 수가 16비트(2Bytes)이므로, 기억장치 용량은 4096Bytes이다.

2) 간접 주소지정 방식

간접 주소지정 방식(indirect addressing mode)

- 오퍼랜드 필드에 기억장치 주소가 저장되어 있지만, 그 주소가 가리키는 기억 장소에 데이터의 유효 주소를 저장해두는 방식

$$EA = (A)$$



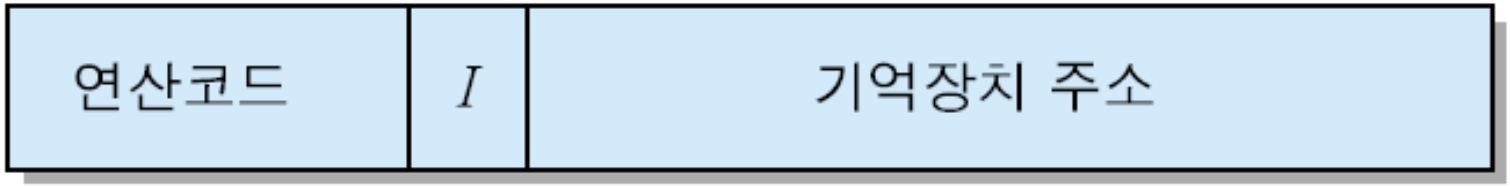
2) 간접 주소지정 방식

- **장점:** 최대 기억장치용량이 단어의 길이에 의하여 결정
  - 주소지정이 가능한 기억장치 용량 확장
  - 단어 길이가 n 비트라면, 최대 2^n개의 기억 장소에 대한 주소지정이 가능

- **단점:** 간접 및 실행 사이클 동안에 두 번의 기억장치 액세스가 필요
  - 첫 번째 액세스: 유효 주소 인출
  - 두 번째 액세스: 해당 주소가 지정하는 기억 장소로부터 실제 데이터 인출

2) 간접 주소지정 방식

- **명령어 형식에 간접비트(I) 필요**
  - 만약 I = 0 이면, 직접 주소지정 방식
  - 만약 I = 1 이면, 간접 주소지정 방식 → 간접 사이클 실행

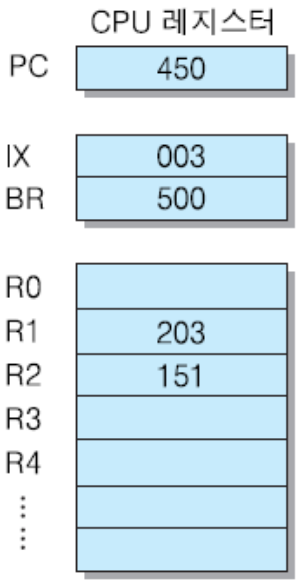


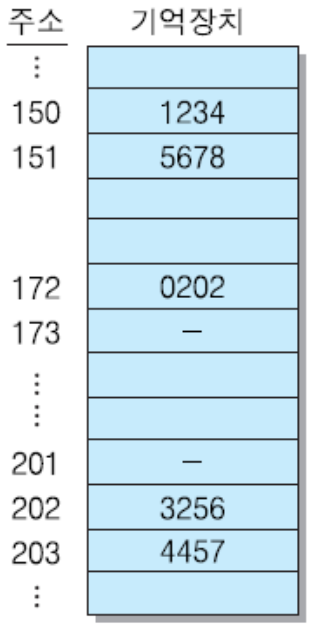
- 다단계(multi-level) 간접 주소지정 방식

$$EA = ((..(A)..))$$

2) 간접 주소지정 방식

- **[예제]** CPU 레지스터 길이와 주소지정 단위는 16비트로 가정
  - **[3]** 간접 주소지정 방식을 사용하는 명령어의 주소 필드(*A*)에 저장된 내용이 '172'라고 가정했을 때, 유효 주소(*EA*) 및 그에 의해 인출되는 데이터는?
  - **[4]** 이 명령어에 의해 주소지정 될 수 있는 기억장치 용량(Bytes)은 얼마인가?





- **[풀이]**
  - **[3]** *EA*는 기억장치 172번지에 저장되는 있는 '202'이다. 따라서 명령어 실행에 사용될 데이터로는 기억장치 202번지에 저장되어 있는 '3256'이 인출된다.
  - **[4]** 주소(*EA*)의 길이는 16비트가 되므로, 주소지정 할 수 있는 기억장치 용량은 2^{16}(64K) × 2Bytes = 128KBytes가 된다.

3) 묵시적 주소지정 방식

묵시적 주소지정 방식(implied addressing mode)

- 명령어 실행에 필요한 데이터의 위치가 묵시적으로 지정되는 방식
- **[예] 'PUSH R1' 명령어:** 레지스터 R1의 내용을 **스택**에 저장
  - SP가 가리키는 기억 장소(TOS)에 R1의 내용을 저장한다는 것이 묵시적으로 정해짐
- **[예]** 'POP' 명령어: TOS에 있는 값을 누산기로 인출
- **[예]** 'SHL' 명령어: 누산기의 내용을 좌측으로 시프트(shift)
- **장점:** 오퍼랜드의 수가 0 또는 1이기 때문에 명령어 길이가 짧음
- **단점:** 종류가 제한됨

4) 즉시 주소지정 방식

❑ 즉시 주소지정 방식(immediate addressing mode)

- 데이터가 명령어에 포함되어 있는 방식
  - 즉, **오퍼랜드 필드의 내용이 연산에 사용할 실제 데이터**
- **용도:** 프로그램에서 레지스터나 변수의 초기 값을 어떤 상수값(constant value)으로 세트 하는 데 사용
- **장점:** 데이터를 인출하기 위하여 기억장치를 액세스할 필요가 없음
- **단점:** 상수값의 크기가 오퍼랜드 필드의 비트 수에 의해 제한됨

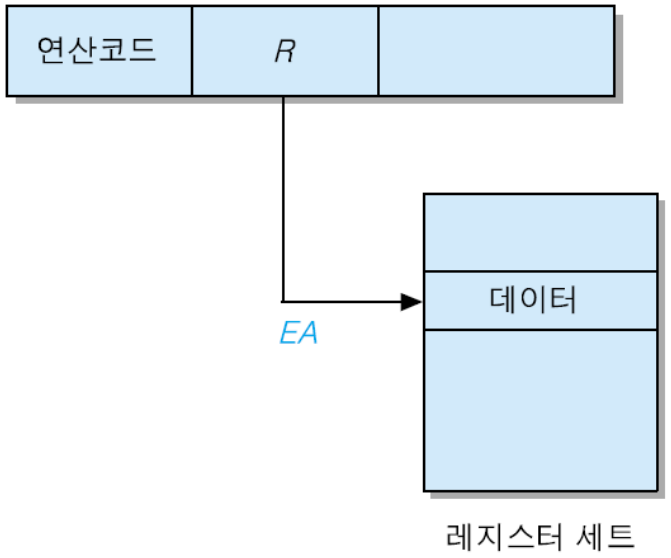
5) 레지스터 주소지정 방식

❑ 레지스터 주소지정 방식(register addressing mode)

- 연산에 사용될 데이터가 CPU 내부 레지스터에 저장되어 있는 경우에 사용
  - 명령어의 오퍼랜드가 해당 레지스터를 가리키는 방식

$$EA = R$$

- 주소지정에 사용될 수 있는 레지스터들의 수 = 2^k개
  - k는 R 필드의 비트수



5) 레지스터 주소지정 방식

장점:

- 오퍼랜드 필드의 비트 수가 적어도 됨
- 데이터 인출을 위하여 주기억장치 액세스가 필요 없음
  - CPU 내부 레지스터에 대한 액세스 시간이 주기억장치 액세스 시간보다 훨씬 짧기 대문에, 명령어 실행 시간이 짧아짐

단점:

- 데이터가 저장될 수 있는 공간이 CPU 내부 레지스터들로 제한

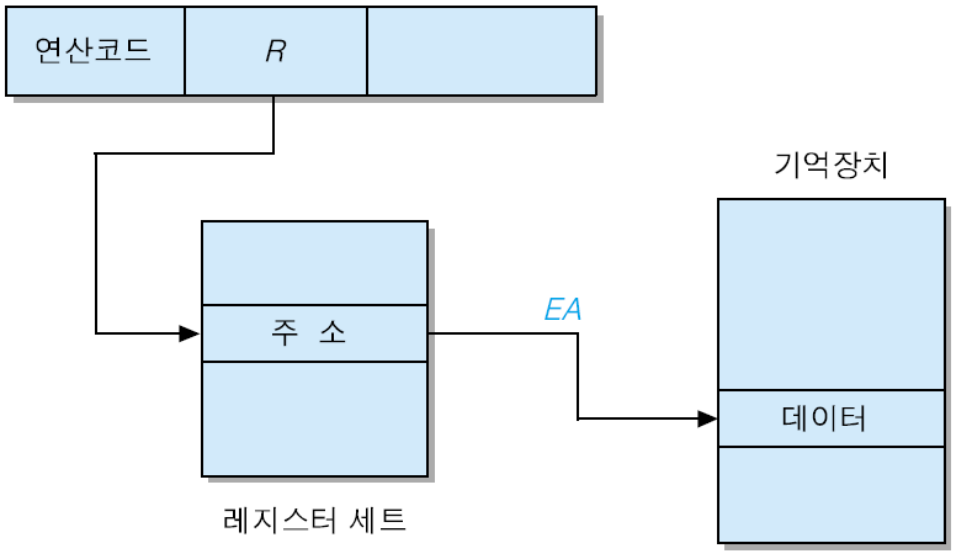
6) 레지스터 간접 주소지정 방식

❑ 레지스터 간접 주소지정 방식(register-indirect address mode)

- 오퍼랜드 필드(레지스터 번호)가 가리키는 레지스터의 내용을 유효 주소로 사용하여 실제 데이터를 인출하는 방식

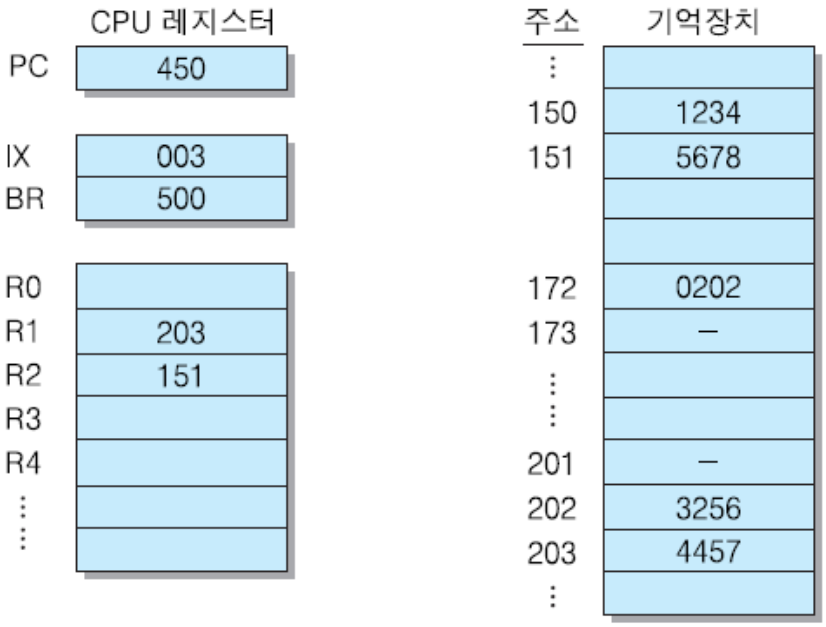
$$EA = (R)$$

- **장점:** 주소지정 할 수 있는 기억장치 영역이 확장
  - 레지스터의 길이 = 16 비트라면, 지정 가능한 기억장치의 주소: 2^{16} = 64K개
  - 레지스터의 길이 = 32 비트라면, 지정 가능한 기억장치의 주소: 2^{32} = 4G개
- **단점:** 데이터 인출을 위하여 한번의 기억장치 액세스가 필요



6) 레지스터 간접 주소지정 방식

- **[예제]** 명령어의 레지스터 필드 R에 '2'가 저장됨
  - **[5]** 레지스터 주소지정 방식이 사용된다면, 연산 처리 과정에서 어떤 데이터가 사용될 것인가?
  - **[6]** 레지스터 간접 주소지정 방식이 사용된다면, 연산 처리 과정에서 어떤 데이터가 사용될 것인가?



- **[풀이]**
  - **[5]** R2에 저장되어 있는 데이터 '151'이 사용됨
  - **[6]** 기억장치 151번지에 저장되어 있는 데이터 '5678'이 사용됨

7) 변위 주소지정 방식

❑ 변위 주소지정 방식(displacement addressing)

- 직접 주소지정과 레지스터 간접 주소지정 방식의 조합
- 지정된 레지스터에 저장된 주소에 명령어 오퍼랜드의 변위(offset)를 더하여 유효 주소를 결정하는 주소지정 방식

$$EA = A + (R)$$

- 사용되는 레지스터에 따라 다양한 변위 주소지정 방식 가능
  - **PC:** 상대 주소지정 방식(relative addressing mode)
  - **인덱스 레지스터:** 인덱스 주소지정 방식(indexed addressing mode)
  - **베이스 레지스터:** 베이스-레지스터 주소지정 방식(base-register addressing mode)

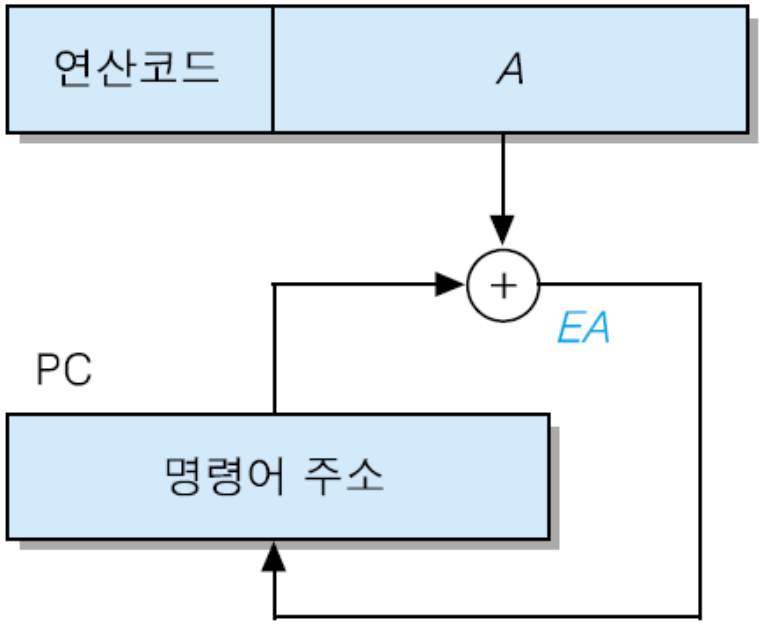
7-1) 상대 주소지정 방식

❑ 상대 주소지정 방식(relative addressing mode)

- 프로그램 카운터(PC)를 레지스터로 사용하여 EA를 계산

$$EA = A + (PC) \qquad \textit{단, A는 2의 보수}$$

- 주로 **분기 명령어**에서 사용
  - A > 0: 앞(forward) 방향으로 분기
  - A < 0: 뒷(backward) 방향으로 분기



7-1) 상대 주소지정 방식

- **장점:** 전체 기억장치 주소가 명령어에 포함되어야 하는 일반적인 분기 명령어보다 적은 수의 비트만 필요
  - PC가 묵시적으로 지정될 수 있으므로 레지스터 필드가 필요 없음
- **단점:** 분기 범위가 오퍼랜드 필드의 길이에 의해 제한
  - 오퍼랜드 비트들로 표현 가능한 2의 보수 범위

❑ **[예제]** 상대 주소지정 방식을 사용하는 JUMP 명령어가 450번지에 저장됨

- **[1]** 만약 오퍼랜드 A = '21'이라면, 몇 번지로 분기하는가?
- **[2]** 만약 오퍼랜드 A = '-50'이라면, 몇 번지로 분기하는가?

❑ **[풀이]**

- **[1]** 이 명령어가 인출 된 후에는 PC의 내용이 451로 증가되므로, 451 + 21 = 472번지로 분기하게 됨
- **[2]** 위와 같은 원리로, 451-50 = 401번지로 분기하게 됨

7-2) 인덱스 주소지정 방식

인덱스 주소지정 방식(indexed addressing mode)

- 인덱스 레지스터의 내용과 변위 A를 더하여 유효 주소를 결정
  - **인덱스 레지스터(IX):** 인덱스(index) 값을 저장하는 특수-목적 레지스터

$$EA = (IX) + A$$

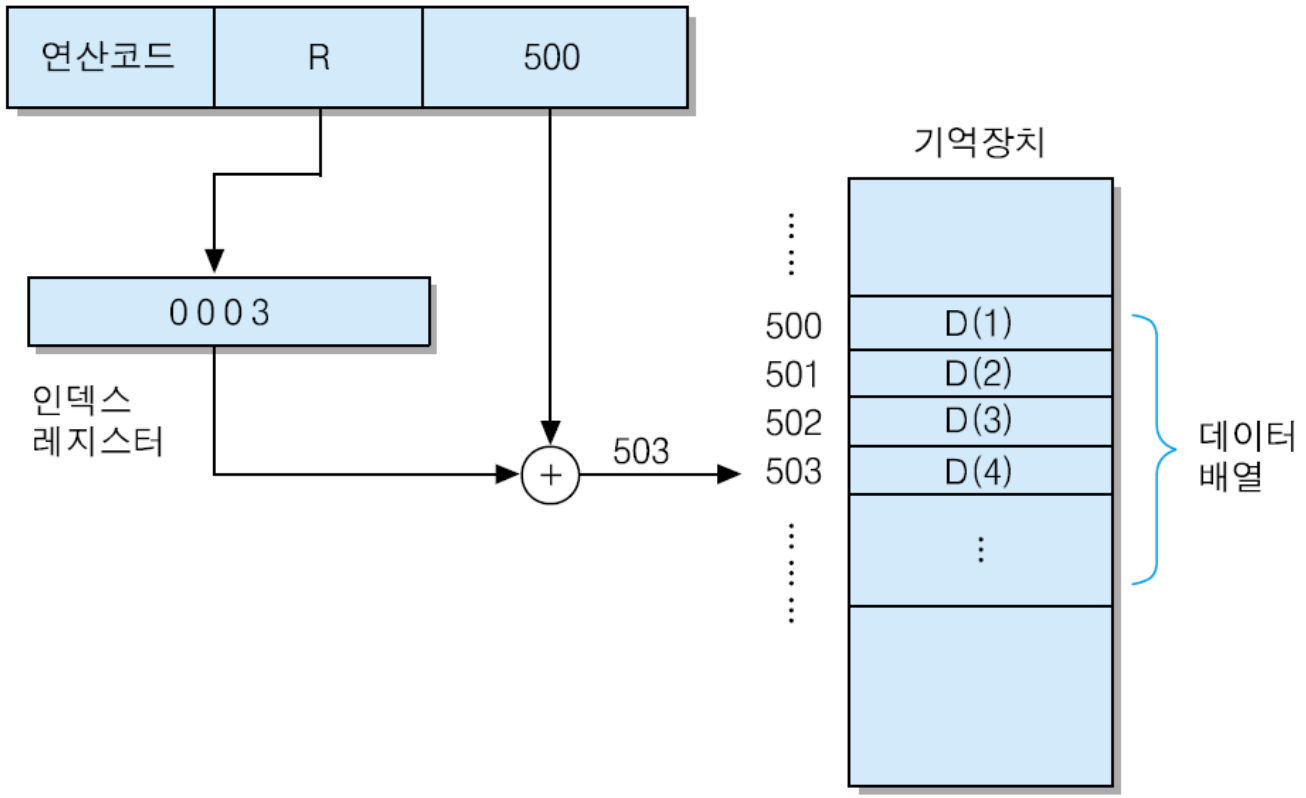
- **주요 용도: 배열 데이터 액세스**
  - 주소 *A*는 기억장치에 저장된 데이터 배열의 시작 주소
- **자동 인덱싱(auto-indexing)**
  - 명령어가 실행될 때마다 인덱스 레지스터의 내용이 자동적으로 증가 혹은 감소
  - 이 방식을 사용하는 명령어가 실행되면 아래의 두 연산이 연속적으로 수행됨

$$EA = (IX) + A$$

$$IX \leftarrow IX + 1$$

7-2) 인덱스 주소지정 방식

- **[예]** 데이터 배열이 기억장치의 500번지부터 저장되어 있고, 명령어의 주소 필드에 '500' 이 포함되어 있을 때, 인덱스 레지스터의 내용(IX) = 3 이면,
  - 데이터 배열의 네 번째 데이터 액세스



7-3) 베이스-레지스터 주소지정 방식

베이스-레지스터 주소지정 방식(base-register addressing mode)

- 베이스 레지스터의 내용과 변위 A를 더하여 유효 주소를 결정
  - **베이스 레지스터(BR):** 기준(base)이 되는 프로그램이나 데이터 블록의 시작 주소를 저장

$$EA = (BR) + A$$

- 주요 용도: 프로그램의 시작 위치 지정하거나 변경하는 데 사용
  - 다중 프로그래밍(multiprogramming) 환경에서 프로그램 코드 및 데이터를 메모리의 다른 위치로 이동시킬 때, 분기 명령어나 데이터를 액세스하는 명령어들의 주소 필드 내용을 수정할 필요 없이 베이스 레지스터만 변경하면 되므로 주소 재배치가 쉬움

주소지정 방식

- 주소지정 방식의 종류
  - 직접 주소지정 방식
  - 간접 주소지정 방식
  - 묵시적 주소지정 방식
  - 즉시 주소지정 방식
  - 레지스터 주소지정 방식
  - 레지스터 간접 주소지정 방식
  - 변위 주소지정 방식
    - 상대 주소지정 방식
    - 인덱스 주소지정 방식
    - 베이스-레지스터 주소지정 방식

실제 상용 프로세서들의 주소지정 방식

주소 지정 방식	X86	ARM	AVR
즉시 주소	○	○	○
직접 주소	○		○
레지스터 주소	○	○	○
레지스터 간접 주소	○	○	○
인덱스 주소	○	○	
베이스-인덱스 주소		○	

명령어 세트 설계 개념

- **CISC(Complex Instruction Set Computer): 복잡 명령어 집합 컴퓨터**
  - 명령어 형식이 복잡함
    - 많은 명령어들을 지원 → 명령어들의 수가 많음
    - 명령어 길이가 일정하지 않음 → 명령어 종류에 따라 달라짐
    - 주소지정 방식이 매우 다양함 → 명령어 실행 시간이 김
  - 프로그램 길이가 감소

- **RISC(Reduced Instruction Set Computer): 축소 명령어 집합 컴퓨터**
  - 명령어 형식이 간단함
    - 명령어들의 수를 최소화
    - 명령어 길이를 일정하게 고정
    - 주소지정 방식의 종류를 단순화
  - 프로그램 길이가 증가

CISC와 RISC 특징 비교

구분	CISC			RISC	
컴퓨터	IBM-360/168	VAX-11/780	Intel 80486	SPARC	MIPS R4000
개발 연도	1973년	1978년	1989년	1987년	1991년
명령어 개수	208개	303개	235개	69개	94개
명령어 크기	2~6바이트	2~57바이트	1~11바이트	4바이트	4바이트
범용 레지스터	16개	16개	8개	40~250개	32개
제어 메모리	420K비트	480K비트	246K비트	-	-
캐시 크기	64K바이트	64K바이트	8K바이트	32K바이트	128K바이트

CISC	RISC
명령어 크기와 형식이 다양하다.	명령어 크기가 동일하고 형식이 제한적이다.
명령어 형식이 가변적이다.	명령어 형식이 고정적이다.
레지스터가 적다.	레지스터가 많다.
주소 지정 방식이 복잡하고 다양하다.	주소 지정 방식이 단순하고 제한적이다.
마이크로 프로그래밍(CPU)이 복잡하다.	컴파일러가 복잡하다.
프로그램 길이가 짧고 명령어 사이클이 길다.	모든 명령어는 한 사이클에 실행되지만 프로그램의 길이가 길다.
파이프 라인이 어렵다.	파이프 라인이 쉽다.

[TMI] x86 아키텍처 vs ARM 아키텍처

특징	x86 아키텍처	ARM 아키텍처
명령어 세트	**CISC** 기반	**RISC** 기반
사용 목적	데스크탑, 노트북, 서버 등 고성능 컴퓨팅용	모바일 장치, 임베디드 시스템, IoT 장치 등
전력 소비	상대적으로 높은 전력 소비	저전력 소비(전력 대비 성능 우수)
성능 최적화	복잡한 계산, 고성능 멀티태스킹, 대용량 데이터 처리	효율성, 배터리 수명, 최신 ARM 칩은 고성능 제공
레거시 소프트웨어 하위 호환성	강력한 하위 호환성 제공 (Windows, Linux, 대부분의 PC 소프트웨어는 x86 기반)	모바일 운영체제(iOS, 안드로이드)와 높은 호환성
설계 철학	복잡한 기능을 하드웨어에서 지원	단순한 명령어로 소프트웨어 최적화에 집중, 모듈식 설계 및 확장성 제공
대표 회사	인텔, AMD 등	애플(Silicon), 삼성전자(엑시노스), 퀄컴(스냅드래곤) 등

Q & A

End!