명령어 세트(Instruction Set)

- CPU의 기능은 이들에 의해 결정된다.
- 그들의 수와 종류는 CPU에 따라 많이 다르다.

→ **CPU 기능을 위해서 정의된 명령어들의 집합**

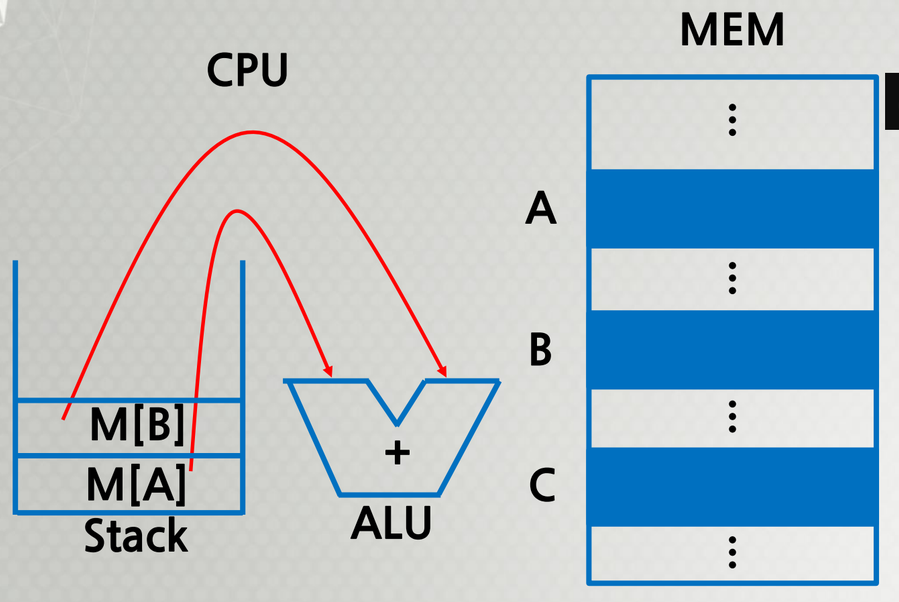
명령어 세트(Instruction Set)

명령어 세트 정의를 위해 결정되어야 할 항목

오퍼랜드의 CPU 기억장소	• 스택(Stack) • 범용 레지스터(GPR) • 누산기(Accumulator)
연산 명령어	• CPU 명령어가 수행할 연산들의 수와 종류
오퍼랜드/명령어	• 일반적인 명령어가 처리 가능한 오퍼랜드의 수
오퍼랜드의 위치	• CPU 의 외부 혹은 내부 • Reg-to-Reg, Mem-to-Reg, Mem-to-Mem
오퍼랜드	• 오퍼랜드의 크기와 형태 • 정의 방법

오퍼랜드의 CPU 기억장소

Stack Architecture



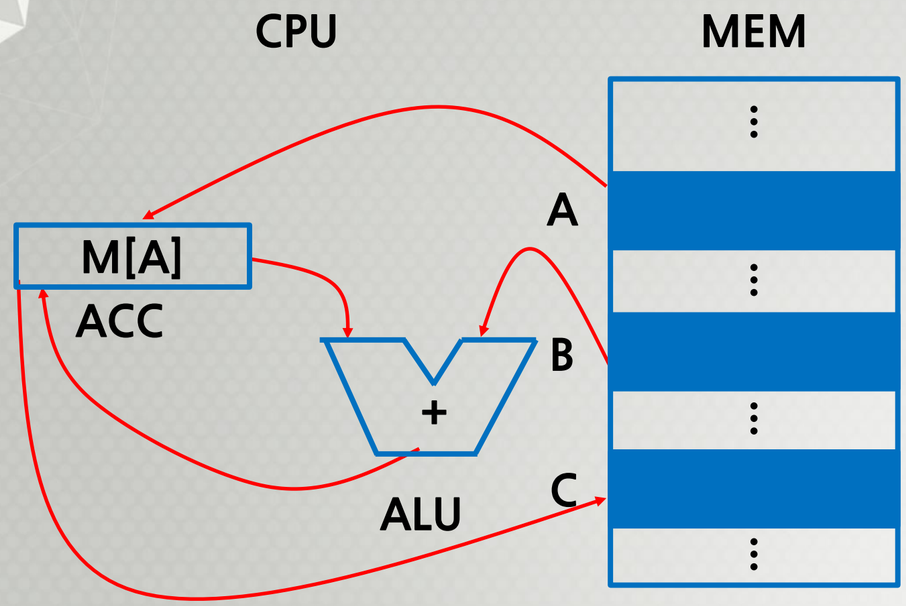
ex) C = A + B ; M[C] ← M[A] + M[B]

- PUSH A
- PUSH B
- ADD
- POP C

Advantages	Disadvantages
• Short instruction-Good code density • Simple model of expression evaluation	• Inefficient code-generation • Bottleneck in stack

오퍼랜드의 CPU 기억장소

Accumulator Architecture



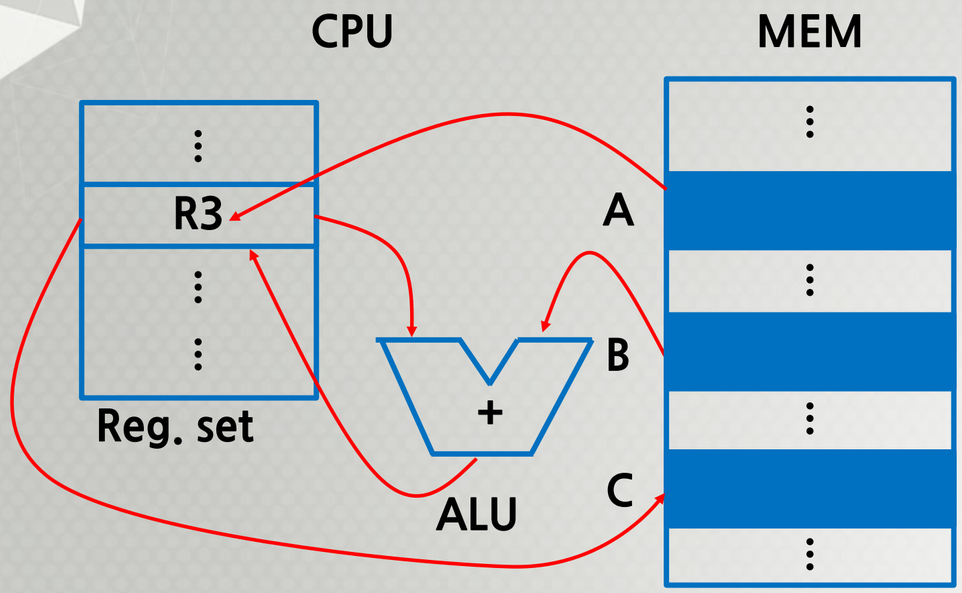
ex) C = A + B ; M[C] ← M[A] + M[B]

- LOAD A
- ADD B
- STORE C

Advantages	Disadvantages
• Short Instruction-Good code density • Minimizes internal state	• Memory Traffic is high

오퍼랜드의 CPU 기억장소

General-purpose register Architecture



ex) C = A + B ; M[C] ← M[A] + M[B]

- LOAD R3, A
- ADD R3, B
- STORE C, R3

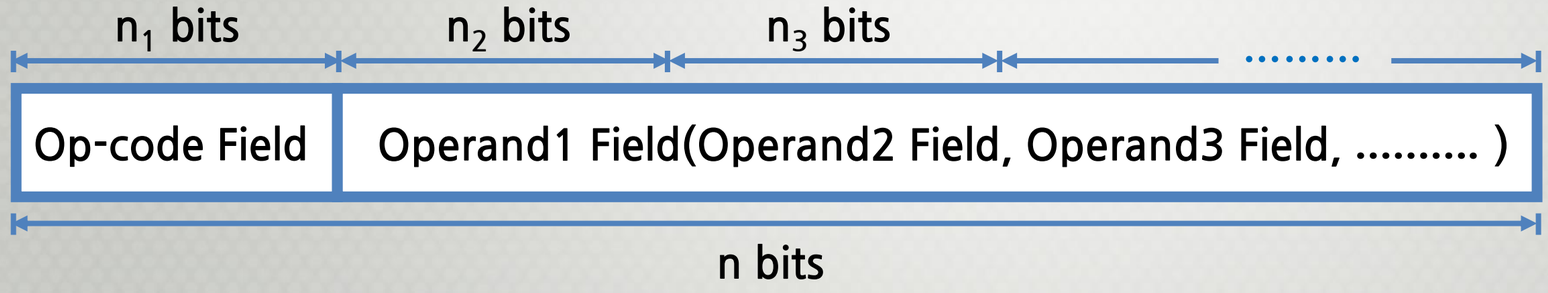
Advantages	Disadvantages
• Most general model	• All named operands lead to long instructions

명령어의 종류

데이터 전송명령	● 레지스터와 레지스터, 레지스터와 기억장치, 기억장치와 기억장치 간에 데이터를 이동시키는 명령
산술 연산명령	● 2의 보수 및 부동소수점 수에 관한 덧셈, 뺄셈, 곱셈 및 나눗셈과 같은 기본적인 산술 연산 명령
논리 연산명령	● 데이터의 각 비트들 간에 대한 AND, OR, NOT 및 Exclusive-OR 와 같은 논리 연산 명령
입출력(I/O) 명령	● CPU와 외부 I/O 장치들 간의 데이터를 이동시키는 명령
프로그램 제어명령	● 각 명령어의 실행 순서를 변경하는 분기(Branch)명령과 서브루틴 호출(Subroutine Call)및 리턴 명령

명령어의 형식

- 명령어는 CPU가 한번에 처리할 수 있는 비트 수의 크기(단어; Word)로 정의된다.
- 명령어를 구성하는 비트는 용도에 따라 몇 개의 필드(Field)로 나누어진다.
- 기본적으로는 Op-code 필드와 Operand 필드로 구성된다.
- Operand 필드는 컴퓨터의 처리 능력에 따라 여러 개의 Operand 필드로 구성된다.



명령어의 형식

명령어의 기본구성 요소

오퍼레이션 코드 (Op-code)	● CPU에서 실행될 연산 지정한다. ● LOAD/STORE, ADD, JUMP, …….
오퍼랜드 (Operand)	● 연산을 실행하는 데 필요한 데이터 혹은 주소 값을 포함한다.

명령어의 형식

Op-code 및 오퍼랜드 필드의 비트 수 결정

Op-code 필드의 비트 수

- CPU에서 수행될 연산 종류의 수에 따라 비트의 수가 결정된다.
- 4비트 → 2^4=16가지의 연산 정의
- 5비트 → 2^5=32가지의 연산 정의
- 비트의 수가 증가할 수록 많은 연산의 정의가 가능하지만, 반면에 오퍼랜드 필드의 비트 수가 감소한다.

명령어의 형식

Op-code 및 오퍼랜드 필드의 비트 수 결정

Operand 필드의 비트 수

- 오퍼랜드의 종류에 따라 결정된다.
- Immediate Value → 표현 가능한 수의 범위가 결정된다.
- Memory Address → CPU가 직접 주소를 지정할 경우 기억장치 영역의 범위가 결정된다.
- Register No. → 범용 레지스터의 수를 결정한다.

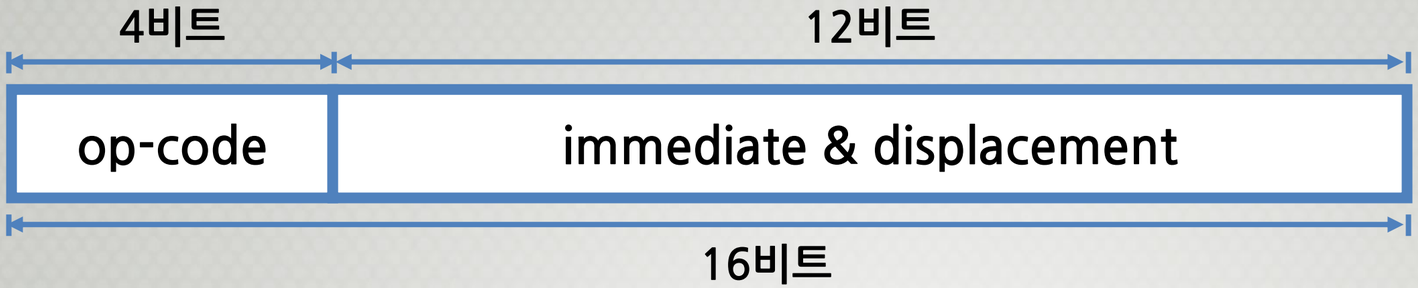
명령어의 형식

명령어의 형식의 예

- 명령어의 길이가 전체 16비트
- Op-code : 4 비트
- 범용 레지스터는 16개

명령어의 형식

1) 1-address instruction

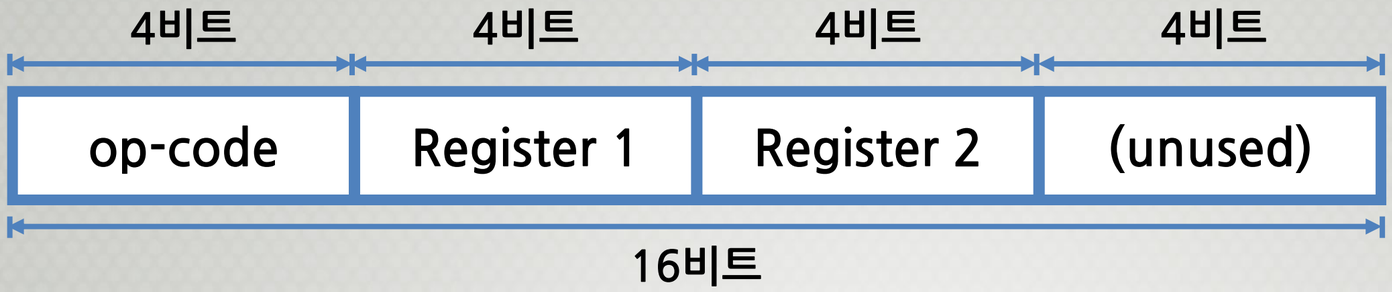


- **Op-code**
  - 4비트 → 2^4 = 16가지의 연산 정의
- **Operand**
  - Memory address : 12비트 → 주소영역 : 0 ~ 2^{12} – 1
  - Immediate value : 12비트 → 표현범위 : -2^{11} ~ 2^{11} – 1

```
ex)  JUMP    1000      ; PC← 1000
     ADD     #1000     ; AC← AC + 1000
```

명령어의 형식

2) 2-address instruction(오퍼랜드가 모두 레지스터인 경우)

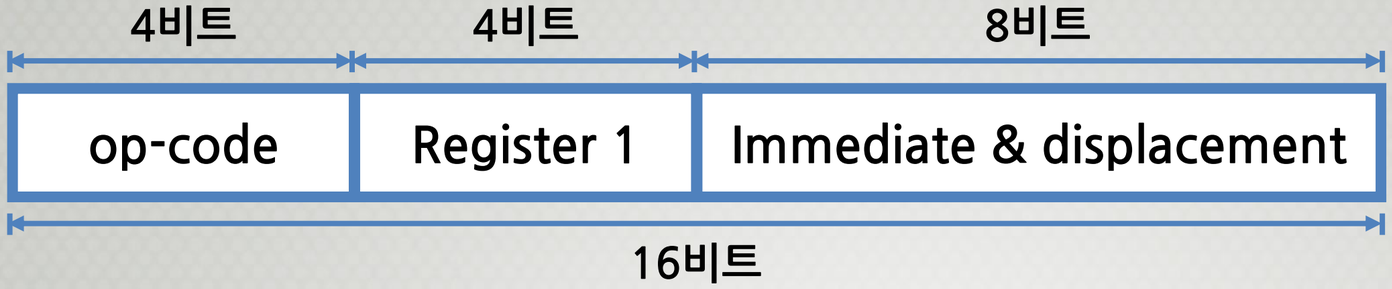


- **Op-code**
  - 4비트 → 2^4 = 16가지의 연산 정의
- **Operand**
  - Register no. : 4비트 → 2^4 = 16 개의 Register
  - Unused → 저장공간의 비효율

```
ex)  ADD     R1, R2      ; R1← R1 + R2
     LOAD    R1, @R2     ; R1← M[R2]
```

명령어의 형식

2) 2-address instruction(오퍼랜드 한 개만 레지스터인 경우)

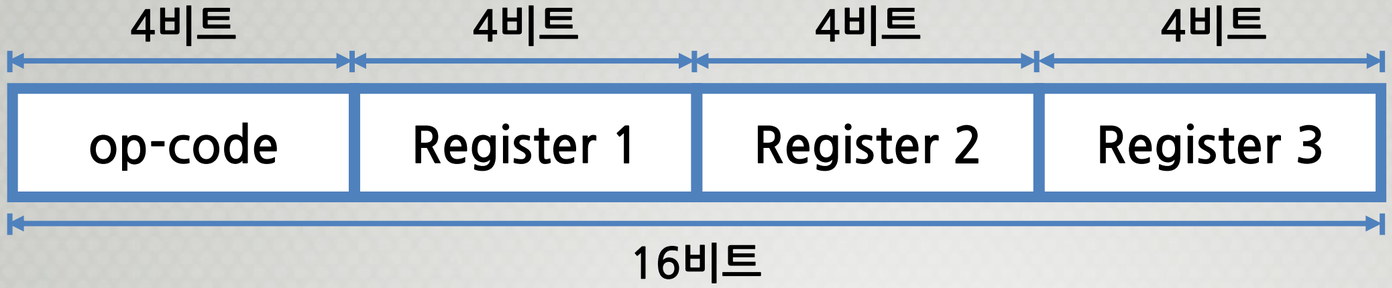


- **Op-code**
  - 4비트 → 2^4 = 16가지의 연산 정의
- **Operand**
  - Register no. : 4비트 → 2^4 = 16 개의 Register
  - Memory address : 8비트 → 주소영역 : 0 ~ $2^8 - 1$
  - Immediate Value : 8비트 → 표현범위 : -2^7 ~ $2^7 - 1$

```
ex)  LOAD    R1, #100     ; R1← 100
     ADD     R1, 100      ; R1← R1 + M[100]
```

명령어의 형식

3) 3-address instruction(모든 오퍼랜드가 레지스터인 경우)

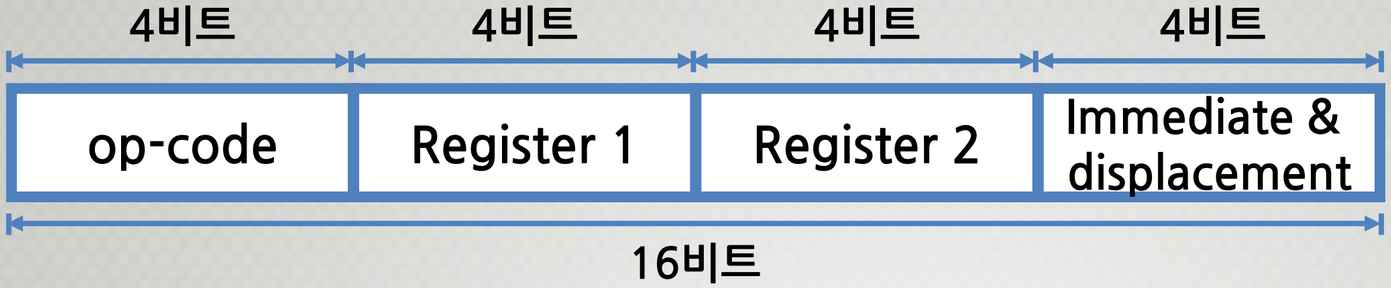


- **Op-code**
  - 4비트 → 2^4 = 16가지의 연산 정의
- **Operand**
  - Register no. : 4비트 → 2^4 = 16 개의 Register

ex) ADD R1, R2, R3 ; R1← R2 + R3

명령어의 형식

3) 3-address instruction(오퍼랜드 두 개는 레지스터인 경우)



- **Op-code**
  - 4비트 → 2^4 = 16가지의 연산 정의
- **Operand**
  - Register no. : 4비트 → 2^4 = 16 개의 Register
  - Memory address : 4비트 → 주소영역 : 0 ~ $2^4 - 1$
  - Immediate Value : 4비트 → 표현범위 : -2^3 ~ $2^3 - 1$

```
ex)  LOAD    R1, 8(R2)          ; R1← M[R2+8]
     ADD     R1, R2, #1000b     ; R1← R2+(− 8)
```

명령어의 형식에 따른 실행 예제

명령어의 다양한 오퍼랜드 형식에 따라 실행될 프로그램의 전체 명령어들의 수량이 다르게 나타난다. 따라서 구성된 명령어의 형식에 따라 **프로그램 실행 시간도 그 수에 비례하여 증가**한다.

- **실행될 프로그램 : X=(A + B) × (C − D)**
- 프로그램을 실행시키는 명령어의 종류
  - ADD : 덧셈
  - SUB : 뺄셈
  - MUL : 곱셈
  - DIV : 나눗셈
  - MOV : 데이터 이동
  - LOAD : 메모리로부터 데이터를 CPU 저장
  - STORE : CPU로부터 메모리에 데이터 저장

명령어의 형식에 따른 실행 예제

1) 1-address instruction를 사용한 프로그램

- 실행될 프로그램 : X=(A + B) × (C − D) → 프로그램의 길이=7

```
LOAD    A    ; AC ← M[A]
ADD     B    ; AC ← AC + M[B]
STORE   T    ; M[T] ← AC
LOAD    C    ; AC ← M[C]
SUB     D    ; AC ← AC − M[D]
MUL     T    ; AC ← AC × M[T]
STORE   X    ; M[X] ← AC
```

Note

- M[x]는 메모리의 'x' 번지에 있는 내용이다.
- T는 중간 연산결과를 저장하기 위한 임시 메모리의 번지이다.

명령어의 형식에 따른 실행 예제

2) 2-address instruction를 사용한 프로그램

- 실행될 프로그램 : X=(A + B) × (C − D) → 프로그램의 길이=6

```
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
```

Note

- M[x]는 메모리의 'x' 번지에 있는 내용이다.
- 중간 연산결과를 저장하기 위해 레지스터를 사용하기 때문에 메모리로 이동할 필요가 없다.

명령어의 형식에 따른 실행 예제

3) 3-address instruction를 사용한 프로그램

- 실행될 프로그램 : X=(A + B) × (C − D) → 프로그램의 길이=3

```
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
```

Note

- M[x]는 메모리의 'x' 번지에 있는 내용이다.
- 오퍼랜드의 수가 많으므로 메모리로부터 직접 해당 번지의 데이터를 CPU로 이동한 후 모든 연산이 끝난 후 저장하기에 실행 명령어의 수를 줄일 수 있다.