

제2장 CPU의 구조와 기능

2.1 CPU의 기본 구조

2.2 명령어 실행

2.3 명령어 파이프라이닝

2.4 명령어 세트

CPU의 기능

- ❑ 명령어 인출(Instruction Fetch) : 기억장치로부터 명령어를 읽어온다
- ❑ 명령어 해독(Instruction Decode) : 수행해야 할 동작을 결정하기 위하여 명령어를 해독한다

➔ 위의 두 기능은 모든 명령어들에 대하여 공통적으로 수행

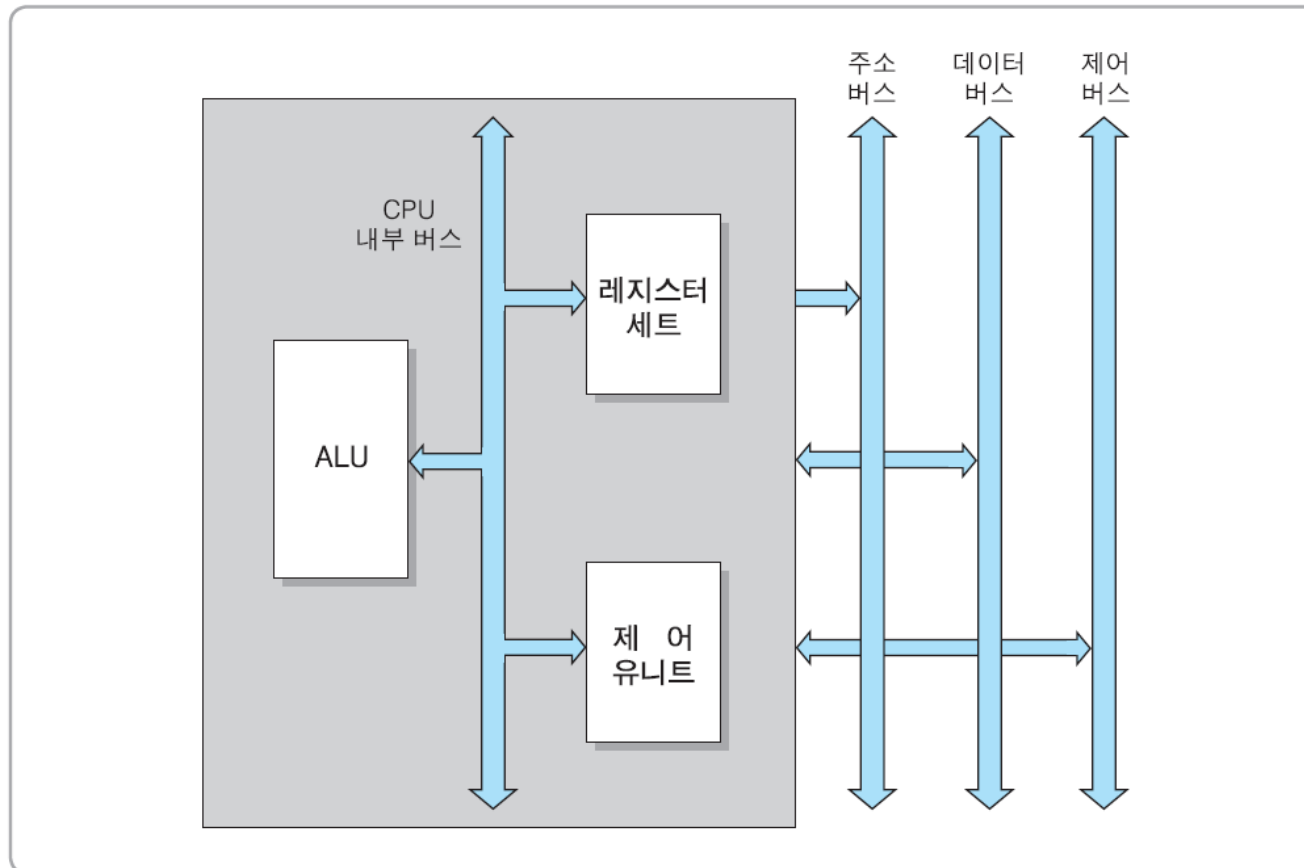
CPU의 기능 (계속)

- ❑ 데이터 인출(Data Fetch) : 명령어 실행을 위하여 데이터가 필요한 경우에는 기억장치 혹은 I/O 장치로부터 그 데이터를 읽어온다
- ❑ 데이터 처리(Data Process) : 데이터에 대한 산술적 혹은 논리적 연산을 수행
- ❑ 데이터 저장(Data Store) : 수행한 결과를 저장

➔ 명령어에 따라 필요한 경우에만 수행

2.1 CPU의 기본 구조

- ❑ 산술논리연산장치(Arithmetic and Logical Unit: ALU)
- ❑ 레지스터 세트(Register Set)
- ❑ 제어 유닛(Control Unit)



CPU의 내부 구성요소

□ ALU

- 각종 산술 연산들과 논리 연산들을 수행하는 회로들로 이루어진 하드웨어 모듈
- 산술 연산 : $+$, $-$, \times , \div
- 논리 연산 : AND, OR, NOT, XOR 등

□ 레지스터(register)

- 액세스 속도가 가장 빠른 기억장치
- CPU 내부에 포함할 수 있는 레지스터들의 수가 제한됨
(특수목적용 레지스터들과 적은 수의 일반목적용 레지스터들)

CPU의 내부 구성요소 (계속)

□ 제어 유닛

- 프로그램 코드(명령어)를 해석하고, 그것을 실행하기 위한 제어 신호들(control signals)을 순차적으로 발생하는 하드웨어 모듈

□ CPU 내부 버스(CPU internal bus)

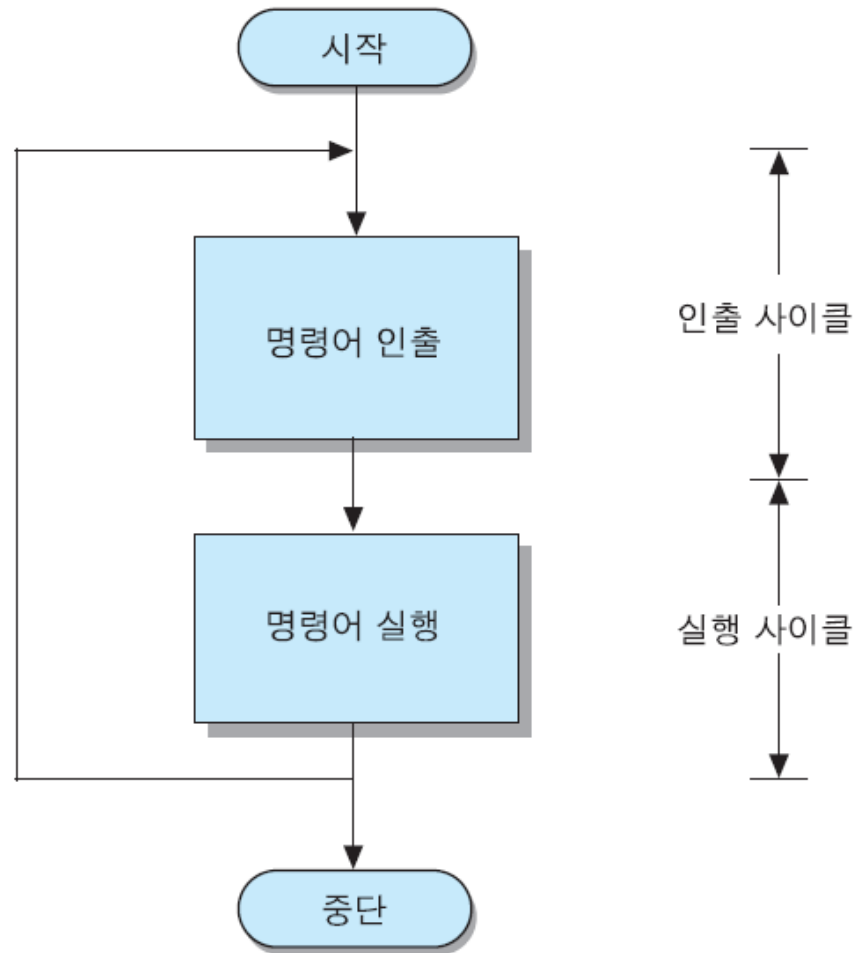
- ALU와 레지스터들 간의 데이터 이동을 위한 데이터 선들과 제어 유닛으로부터 발생하는 제어 신호 선들로 구성된 내부 버스
- 외부의 시스템 버스들과는 직접 연결되지 않으며, 반드시 버퍼 레지스터들 혹은 시스템 버스 인터페이스 회로를 통하여 시스템 버스와 접속

2.2 명령어 실행

- **명령어 사이클(instruction cycle)**: CPU가 한 개의 명령어를 실행하는 데 필요한 전체 처리 과정으로서, CPU가 프로그램 실행을 시작한 순간부터 전원을 끄거나 회복 불가능한 오류가 발생하여 중단될 때까지 반복

- **두 개의 부사이클(subcycle)들로 분리**
 - **인출 사이클(fetch cycle)** : CPU가 기억장치로부터 명령어를 읽어오는 단계
 - **실행 사이클(execution cycle)** : 명령어를 실행하는 단계

기본 명령어 사이클



명령어 실행에 필요한 CPU 내부 레지스터들

□ 프로그램 카운터(Program Counter: PC)

- 다음에 인출할 명령어의 주소를 가지고 있는 레지스터
- 각 명령어가 인출된 후에는 자동적으로 일정 크기(한 명령어 길이)만큼 증가
- 분기(branch) 명령어가 실행되는 경우에는 목적지 주소로 갱신

□ 누산기(Accumulator: AC)

- 데이터를 일시적으로 저장하는 레지스터.
- 레지스터의 길이는 CPU가 한 번에 처리할 수 있는 데이터 비트 수 (단어 길이)와 동일

□ 명령어 레지스터(Instruction Register: IR)

- 가장 최근에 인출된 명령어 코드가 저장되어 있는 레지스터

명령어 실행에 필요한 CPU 내부 레지스터들 (계속)

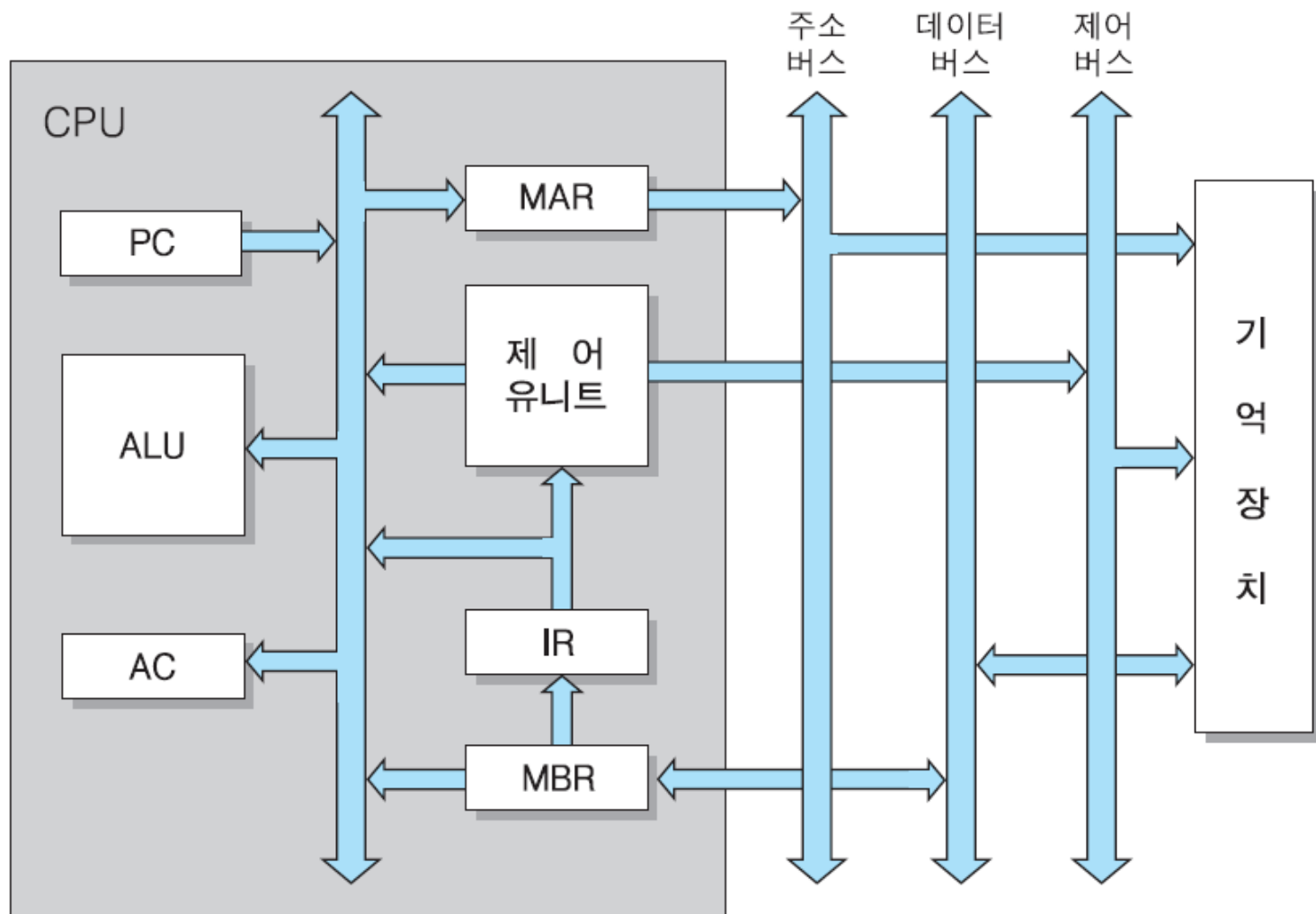
□ 기억장치 주소 레지스터(Memory Address Register: MAR)

- PC에 저장된 명령어 주소가 시스템 주소 버스로 출력되기 전에 일시적으로 저장되는 주소 레지스터

□ 기억장치 버퍼 레지스터(Memory Buffer Register: MBR)

- 기억장치에 쓰여질 데이터 혹은 기억장치로부터 읽혀진 데이터를 일시적으로 저장하는 버퍼 레지스터

데이터 통로가 표시된 CPU 내부 구조



2.2.1 인출 사이클

인출 사이클의 마이크로 연산(micro-operation)

$$t_0 : \text{MAR} \leftarrow \text{PC}$$

$$t_1 : \text{MBR} \leftarrow \text{M}[\text{MAR}], \text{PC} \leftarrow \text{PC} + 1$$

$$t_2 : \text{IR} \leftarrow \text{MBR}$$

단, t_0 , t_1 및 t_2 는 CPU 클록(clock)의 주기

[첫번째 주기] 현재의 PC 내용을 CPU 내부 버스를 통하여 MAR로 전송

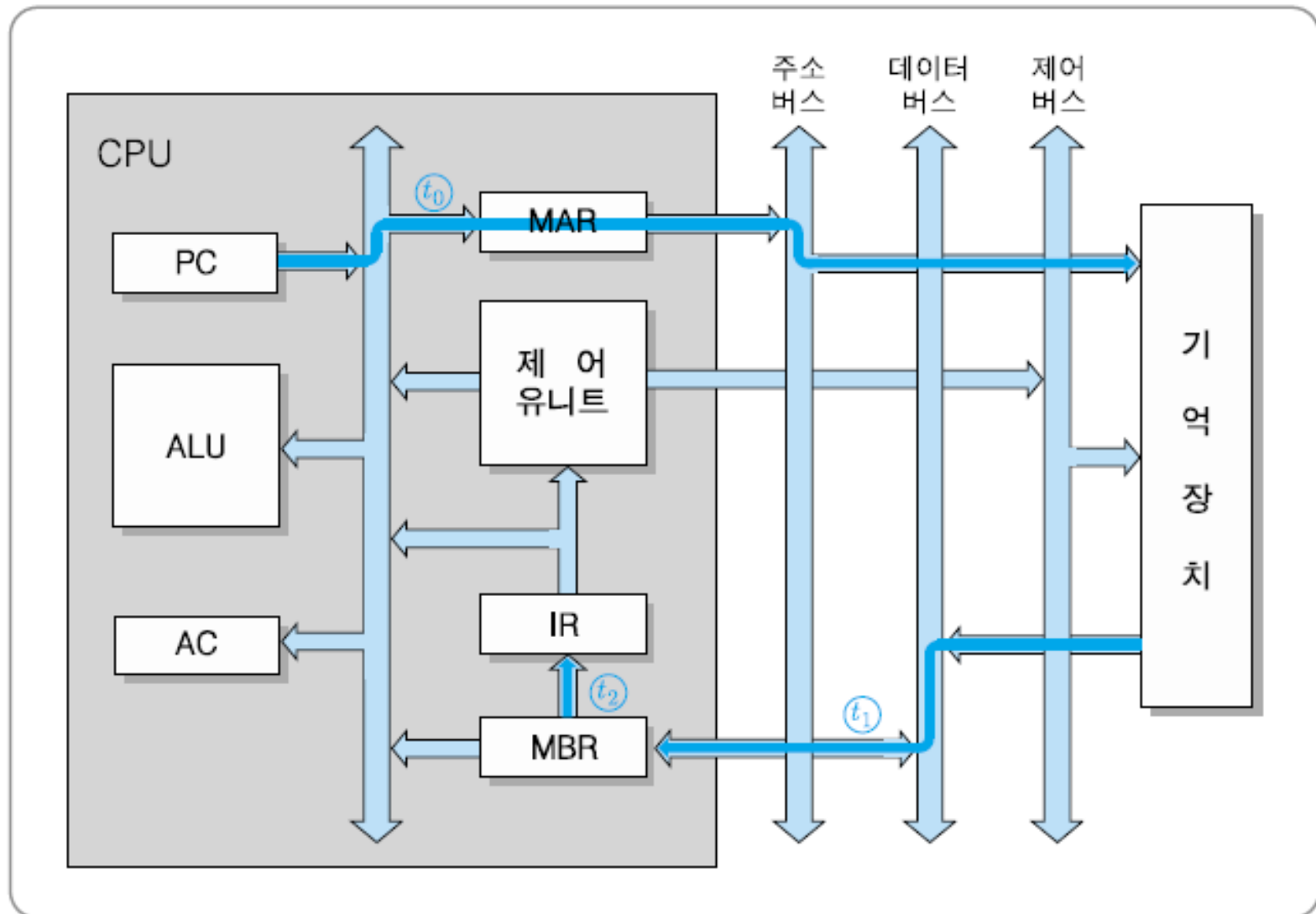
[두번째 주기] 그 주소가 지정하는 기억장치 위치로부터 읽혀진 명령어가 데이터 버스를 통하여 MBR로 적재되며, PC의 내용에 1을 더한다

[세번째 주기] MBR에 있는 명령어 코드가 명령어 레지스터인 IR로 이동

[예] CPU 클록 주파수 = 1GHz (클럭 주기 = 1ns)

→ 인출 사이클 : 1ns x 3 = 3ns 소요

인출 사이클의 주소 및 명령어 흐름도



2.2.2 실행 사이클

- CPU는 실행 사이클 동안에 명령어 코드를 해독(decode)하고, 그 결과에 따라 필요한 연산들을 수행
- CPU가 수행하는 연산들의 종류
 - 데이터 이동 : CPU와 기억장치 간 혹은 I/O장치 간에 데이터를 이동
 - 데이터 처리 : 데이터에 대하여 산술 혹은 논리 연산을 수행
 - 데이터 저장 : 연산 결과 데이터 혹은 입력장치로부터 읽어 들인 데이터를 기억장치에 저장
 - 프로그램 제어 : 프로그램의 실행 순서를 결정
- 실행 사이클에서 수행되는 마이크로-연산들은 명령어의 연산 코드(op code)에 따라 결정됨

기본적인 명령어 형식의 구성

□ 연산 코드(operation code)

- CPU가 수행할 연산을 지정

□ 오퍼랜드(operand)

- 명령어 실행에 필요한 데이터가 저장된 주소(*addr*)



[사례 1] **LOAD** *addr* 명령어

- 기억장치에 저장되어 있는 데이터를 CPU 내부 레지스터인 **AC**로 이동하는 명령어

$t_0 : \text{MAR} \leftarrow \text{IR}(\text{addr})$

$t_1 : \text{MBR} \leftarrow \text{M}[\text{MAR}]$

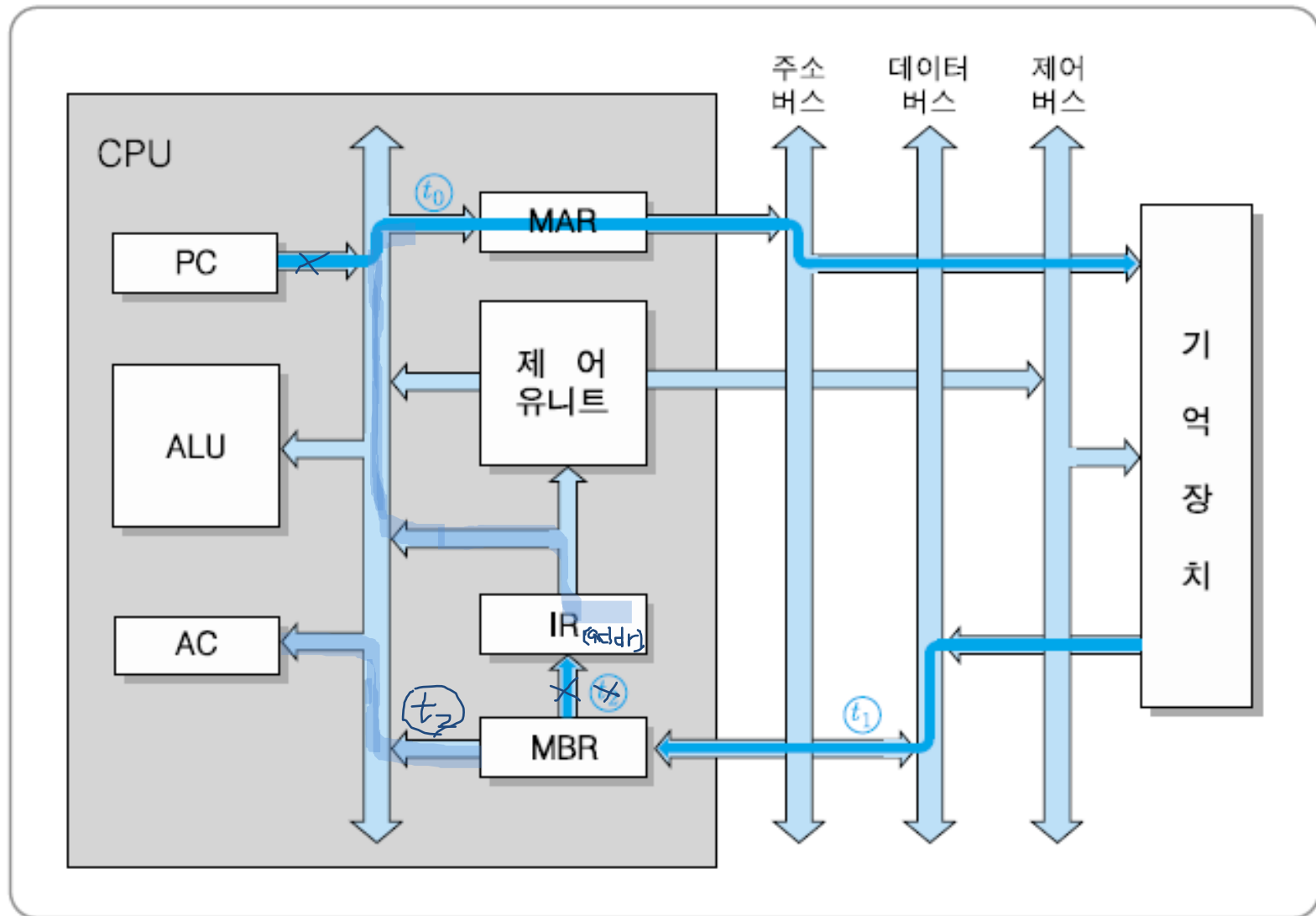
$t_2 : \text{AC} \leftarrow \text{MBR}$

[첫번째 주기] 명령어 레지스터 **IR**에 있는 명령어의 주소 부분을 **MAR**로 전송

[두번째 주기] 그 주소가 지정한 기억장소로부터 데이터를 인출하여 **MBR**로 전송

[세번째 주기] 그 데이터를 **AC**에 적재

인출 사이클의 주소 및 명령어 흐름도



[사례 2] *STA addr* 명령어

□ AC 레지스터의 내용을 기억장치에 저장하는 명령어

$$t_0 : \text{MAR} \leftarrow \text{IR}(\text{addr})$$

$$t_1 : \text{MBR} \leftarrow \text{AC}$$

$$t_2 : \text{M}[\text{MAR}] \leftarrow \text{MBR}$$

[첫번째 주기] 데이터를 저장할 기억장치의 주소를 **MAR**로 전송

[두번째 주기] 저장할 데이터를 버퍼 레지스터인 **MBR**로 이동

[세번째 주기] **MBR**의 내용을 **MAR**이 지정하는 기억장소에 저장

[사례 3] *ADD addr* 명령어

- 기억장치에 저장된 데이터를 AC의 내용과 더하고, 그 결과는 다시 AC에 저장하는 명령어

$$t_0 : \text{MAR} \leftarrow \text{IR}(\text{addr})$$

$$t_1 : \text{MBR} \leftarrow \text{M}[\text{MAR}]$$

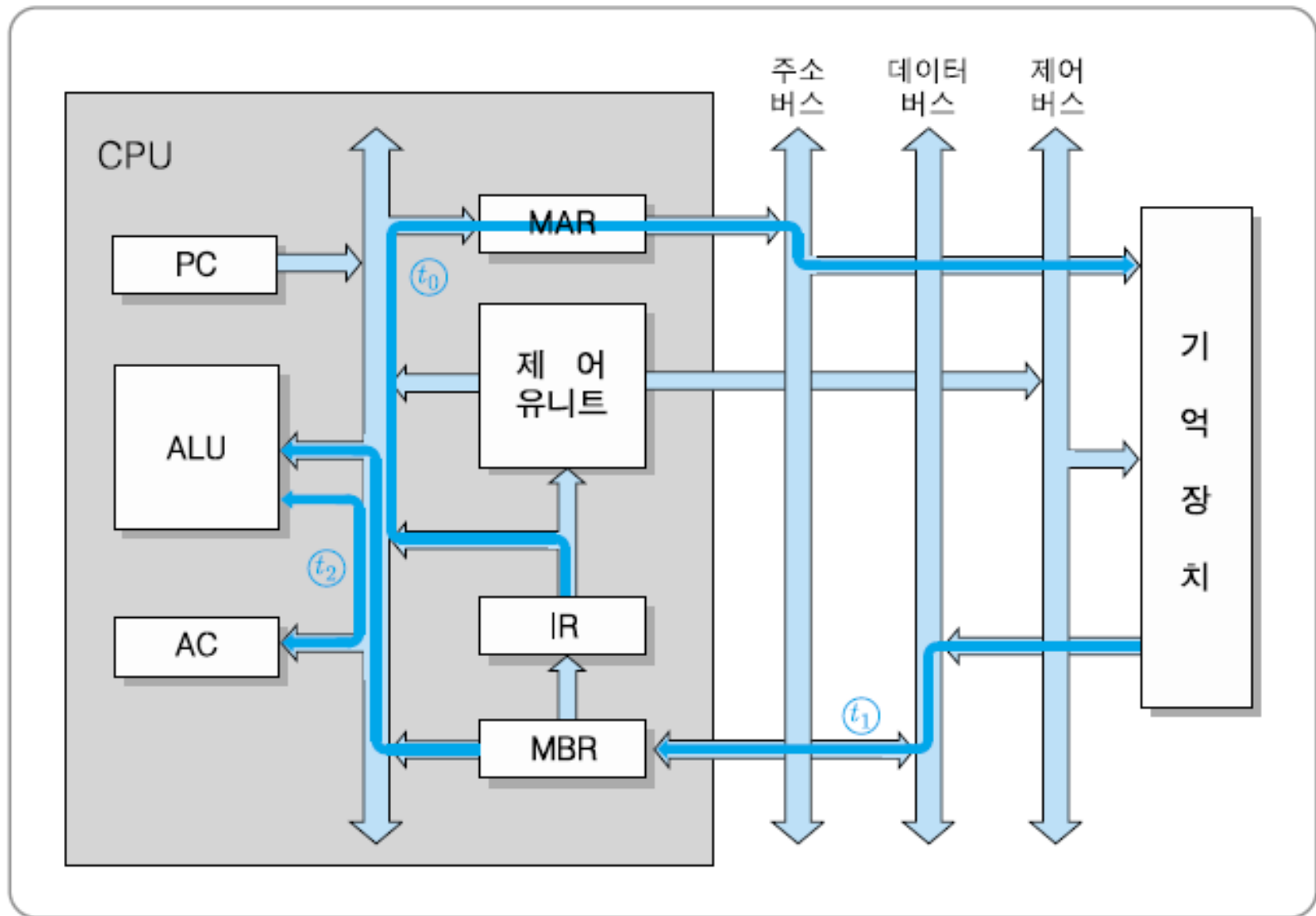
$$t_2 : \text{AC} \leftarrow \text{AC} + \text{MBR}$$

[첫번째 주기] 데이터를 저장할 기억장치의 주소를 MAR로 전송

[두번째 주기] 저장할 데이터를 버퍼 레지스터인 MBR로 이동

[세번째 주기] 그 데이터와 AC의 내용을 더하고 결과값을 다시 AC에 저장

ADD 명령어 실행 사이클 동안의 정보 흐름



[사례 4] **JUMP *addr*** 명령어

- 오퍼랜드(*addr*)가 가리키는 위치의 명령어로 실행 순서를 변경하는 분기(branch) 명령어

$$t_0 : PC \leftarrow IR(addr)$$

- 명령어의 오퍼랜드(분기할 목적지 주소)를 PC에 저장
- 다음 명령어 인출 사이클에서 그 주소의 명령어가 인출되므로, 분기가 발생

어셈블리 프로그램 실행과정의 예

◆ 연산 코드에 임의의 정수 배정

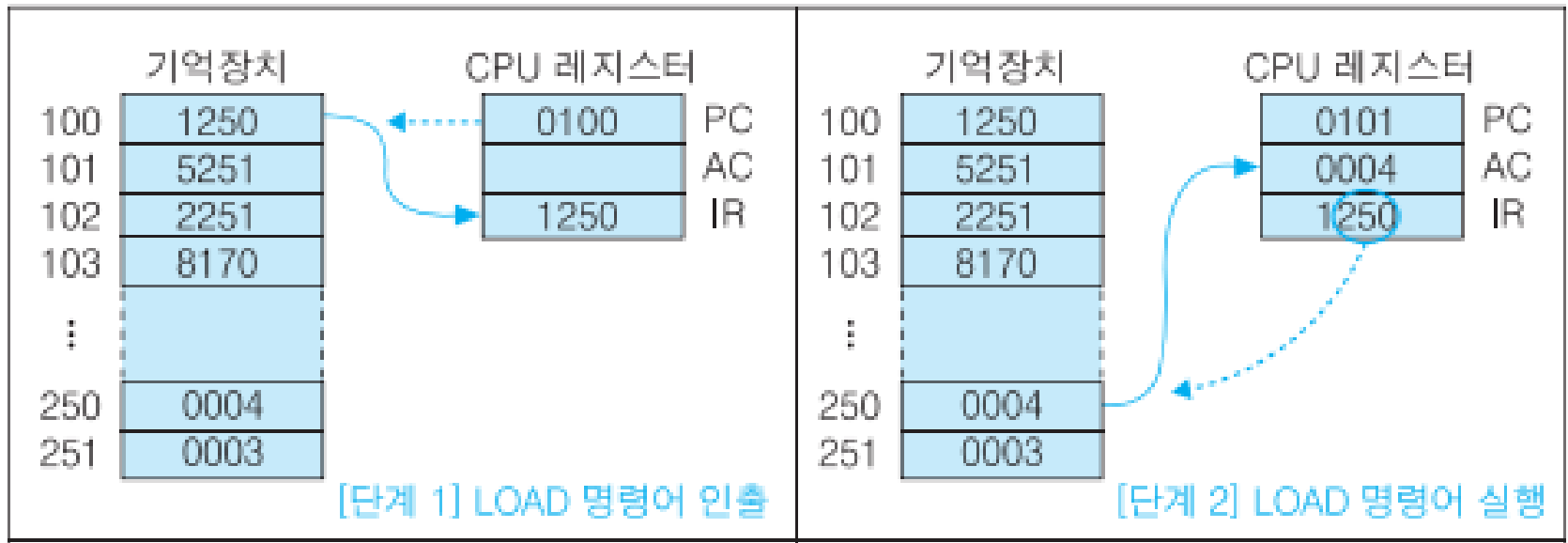
- **LOAD : 1**
- **STA : 2**
- **ADD : 5**
- **JUMP : 8**

[어셈블리 프로그램의 예]

주소	명령어	기계 코드
100	LOAD 250	1250
101	ADD 251	5251
102	STA 251	2251
103	JUMP 170	8170

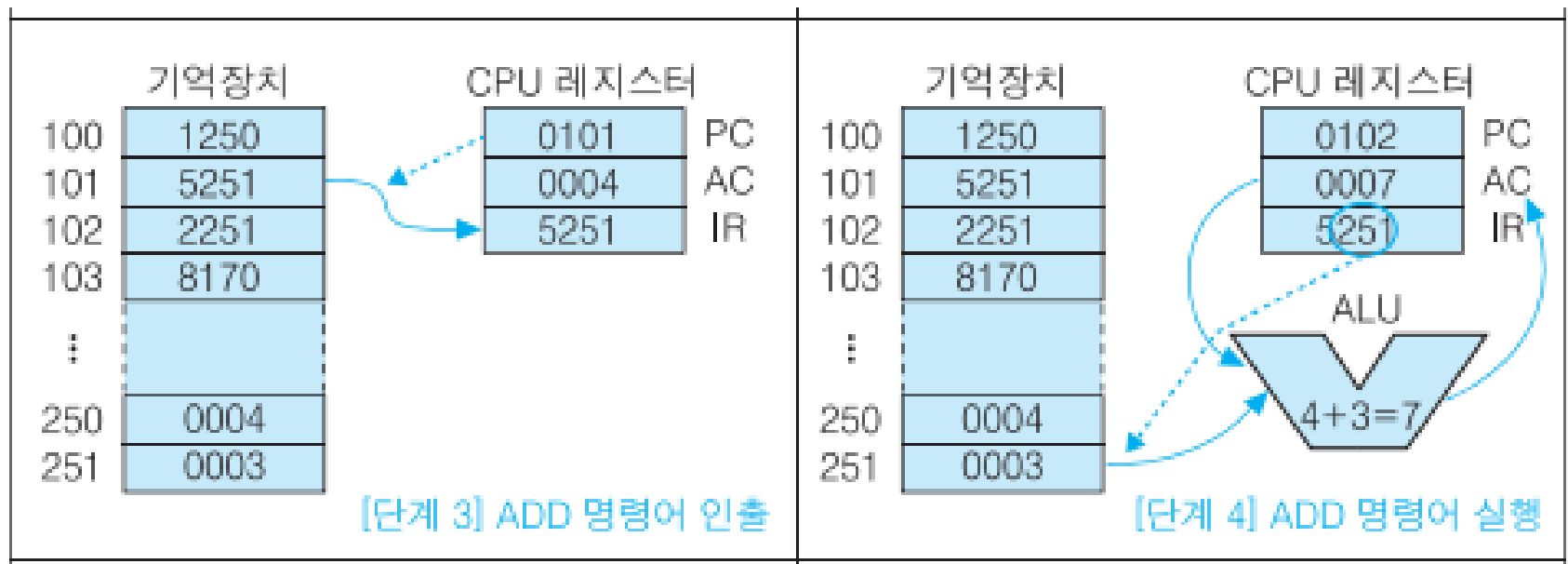
어셈블리 프로그램 실행과정의 예 (계속)

- ❑ 100번지의 첫 번째 명령어 코드가 인출되어 **IR**에 저장
- ❑ 250 번지의 데이터를 **AC**로 이동
- ❑ **PC = PC + 1 = 101**



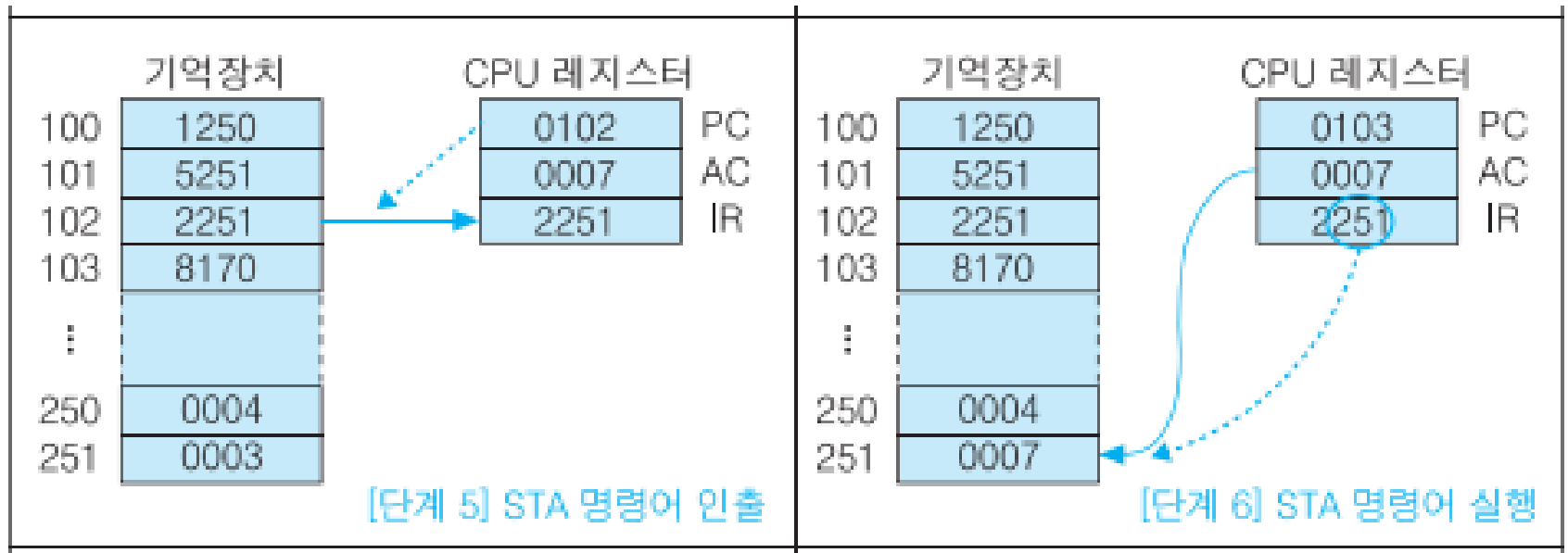
어셈블리 프로그램 실행과정의 예 (계속)

- ❑ 두 번째 명령어가 101번지로부터 인출되어 IR에 저장
- ❑ AC의 내용과 251 번지의 내용을 더하고, 결과를 AC에 저장
- ❑ PC의 내용은 102로 증가



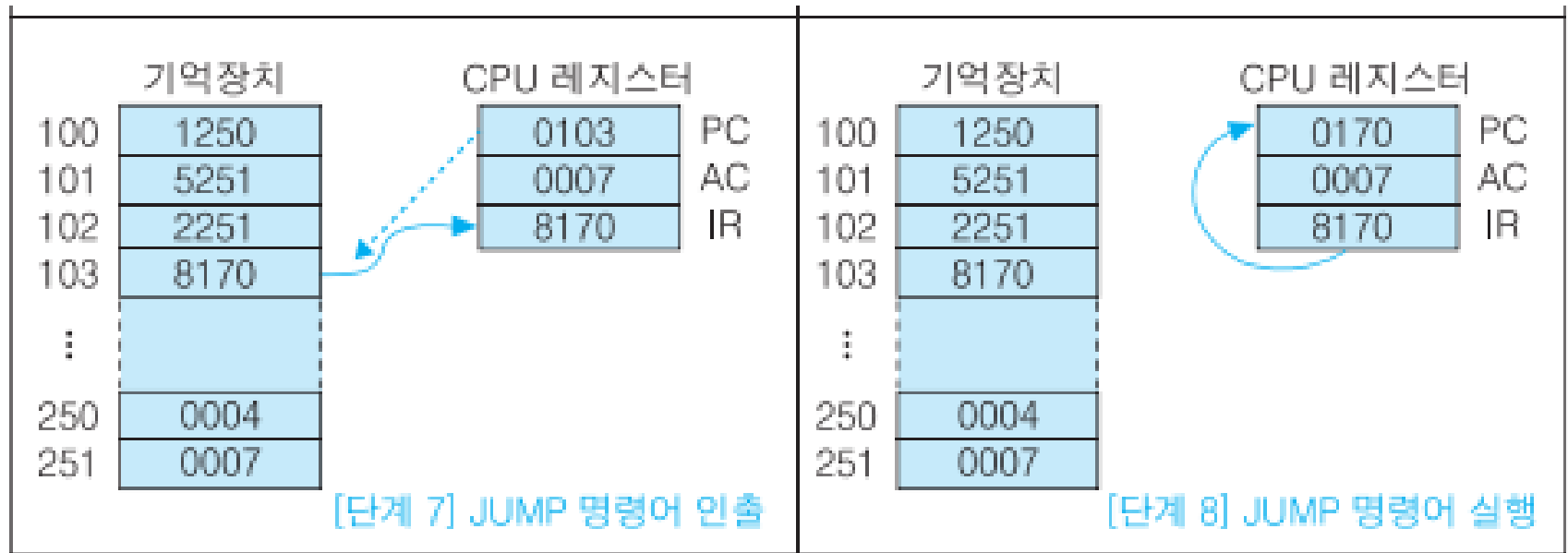
어셈블리 프로그램 실행과정의 예 (계속)

- 세 번째 명령어가 102 번지로부터 인출되어 **IR**에 저장
- **AC**의 내용을 251 번지에 저장
- **PC**의 내용은 103으로 증가



어셈블리 프로그램 실행과정의 예 (계속)

- 네 번째 명령어가 103 번지로부터 인출되어 IR에 저장
- 분기될 목적지 주소, 즉 IR의 하위 부분(170)이 PC로 적재
(다음 명령어 인출 사이클에서는 170 번지의 명령어 인출)



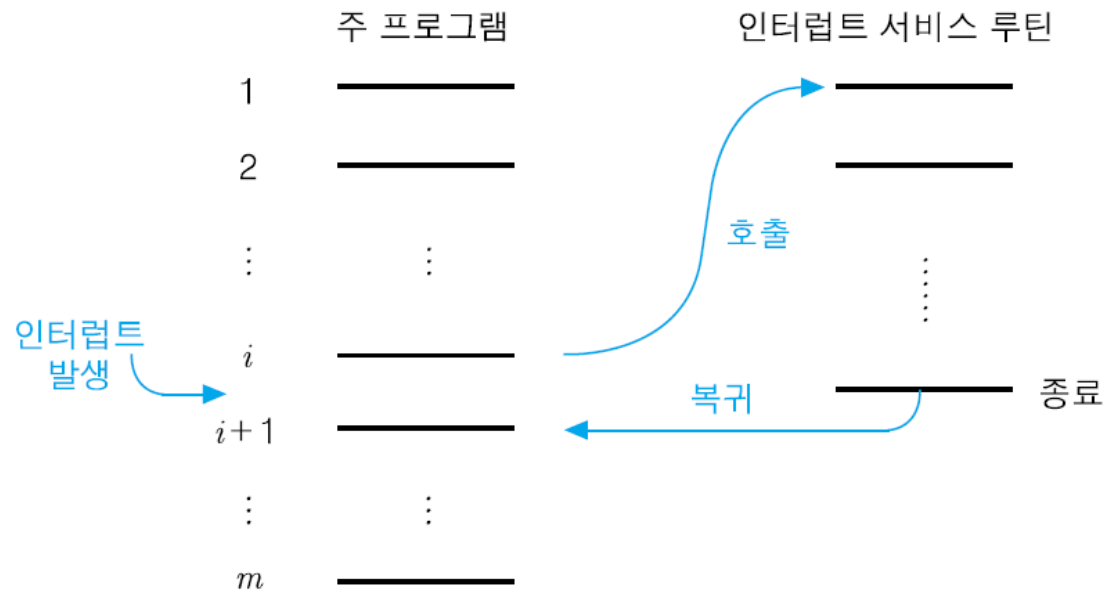
2.2.3 인터럽트 사이클(interrupt cycle)

인터럽트: 프로그램 실행 중에 CPU의 현재 처리 순서를 중단시키고 다른 동작을 수행하도록 요구하는 시스템 동작

- 외부로부터 인터럽트 요구가 들어오면,
 - CPU는 원래의 프로그램 수행을 중단하고,
 - 요구된 인터럽트를 위한 서비스 프로그램을 먼저 수행

- **인터럽트 서비스 루틴(interrupt service routine: ISR) :** 인터럽트를 처리하기 위하여 수행되는 프로그램 루틴

인터럽트에 의한 제어의 이동



인터럽트 처리 과정

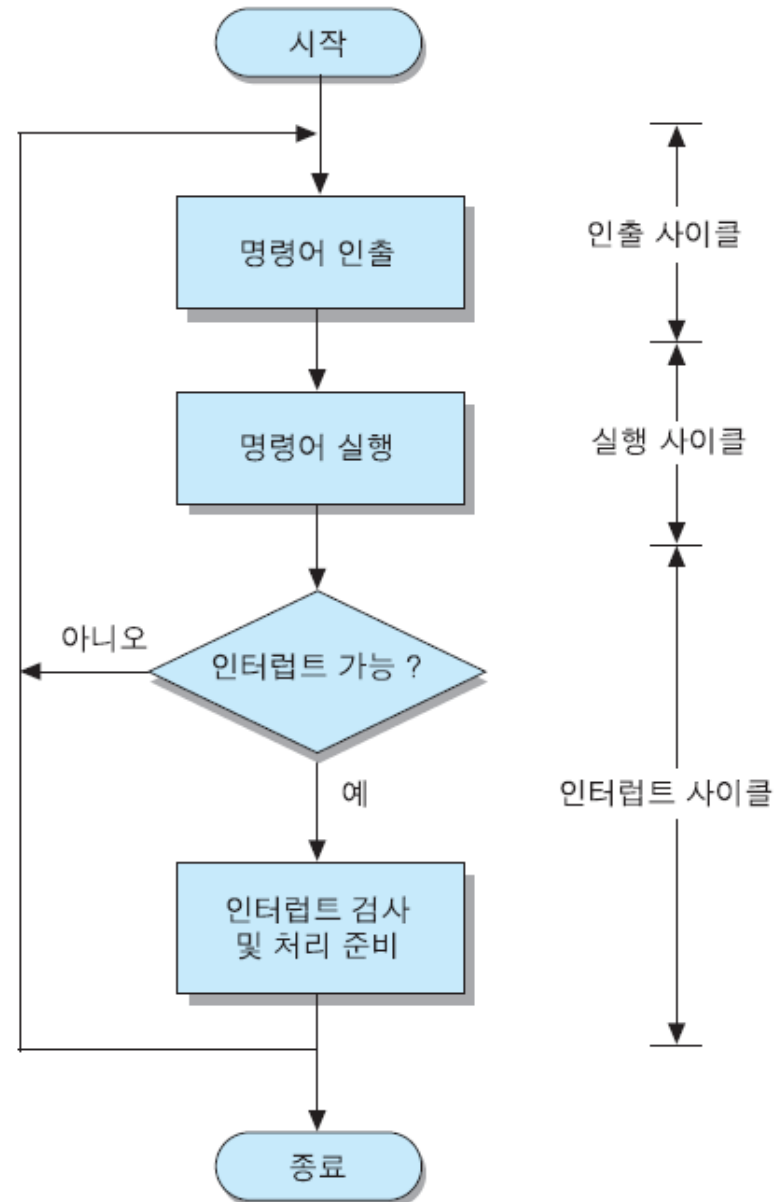
□ 인터럽트가 들어왔을 때 CPU는

- 어떤 장치가 인터럽트를 요구했는지 확인하고, 해당 ISR을 호출
- 서비스가 종료된 다음에는 중단되었던 원래 프로그램의 수행 계속

□ CPU 인터럽트 처리의 세부 동작(인터럽트 사이클 동안 수행)

- ① 현재의 명령어 실행을 끝낸 즉시, 다음에 실행할 명령어의 주소(PC의 내용)를 스택(stack)에 저장 → 일반적으로 스택은 주기억 장치의 특정 부분
- ② ISR을 호출하기 위하여 그 루틴의 시작 주소를 PC에 적재. 이때 시작 주소는 인터럽트를 요구한 장치로부터 전송되거나 미리 정해진 값으로 결정 → 자세한 사항은 7.4절에서 설명

인터럽트 사이클이 추가된 명령어 사이클



인터럽트 사이클의 마이크로 연산

$t_0 : \text{MBR} \leftarrow \text{PC}$

$t_1 : \text{MAR} \leftarrow \text{SP}, \text{PC} \leftarrow \text{ISR의 시작 주소}$

$t_2 : \text{M}[\text{MAR}] \leftarrow \text{MBR}$

단, SP는 스택 포인터(stack pointer)는 스택의 최상위 주소(top of stack: TOS)를 저장하고 있는 레지스터이며, 저장 후에는 1 감소)

[첫번째 주기] PC의 내용을 MBR로 전송

[두번째 주기] SP의 내용을 MAR로 전송하고,

PC의 내용은 인터럽트 서비스 루틴의 시작 주소로 변경

[세번째 주기] MBR에 저장되어 있던 원래 PC의 내용을 스택에 저장

인터럽트 사이클의 마이크로 연산 [예]

- 아래 프로그램의 첫 번째 명령어인 **LOAD 250** 명령어가 실행되는 동안에 인터럽트가 들어왔으며, 현재 **SP = 999** 이고, 인터럽트 서비스 루틴의 시작 주소는 **650** 번지라고 가정

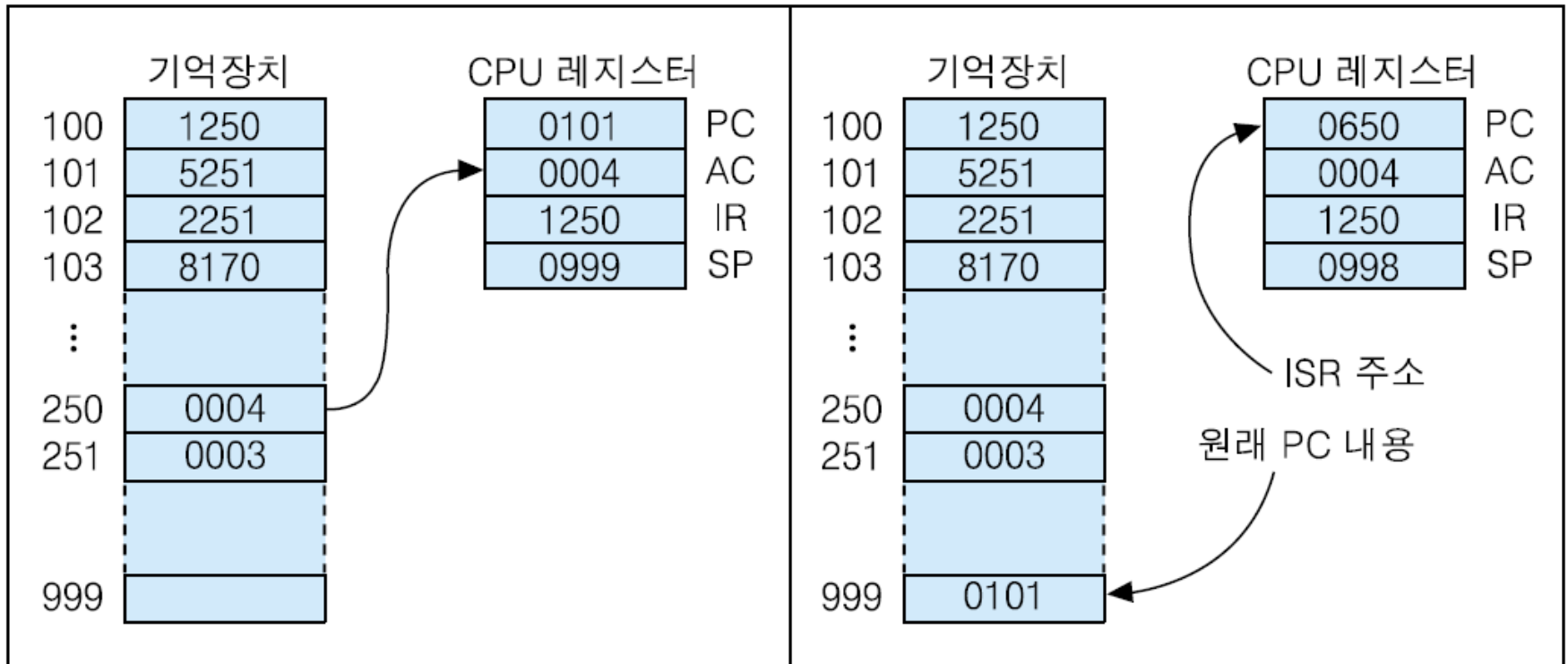
100 **LOAD 250**

101 **ADD 251**

102 **STA 251**

103 **JUMP 170**

인터럽트 요구가 들어온 경우의 상태 변화



(a) LOAD 명령어의 실행 사이클이 종료된 상태

(b) 인터럽트 사이클이 종료된 상태

다중 인터럽트(multiple interrupt)

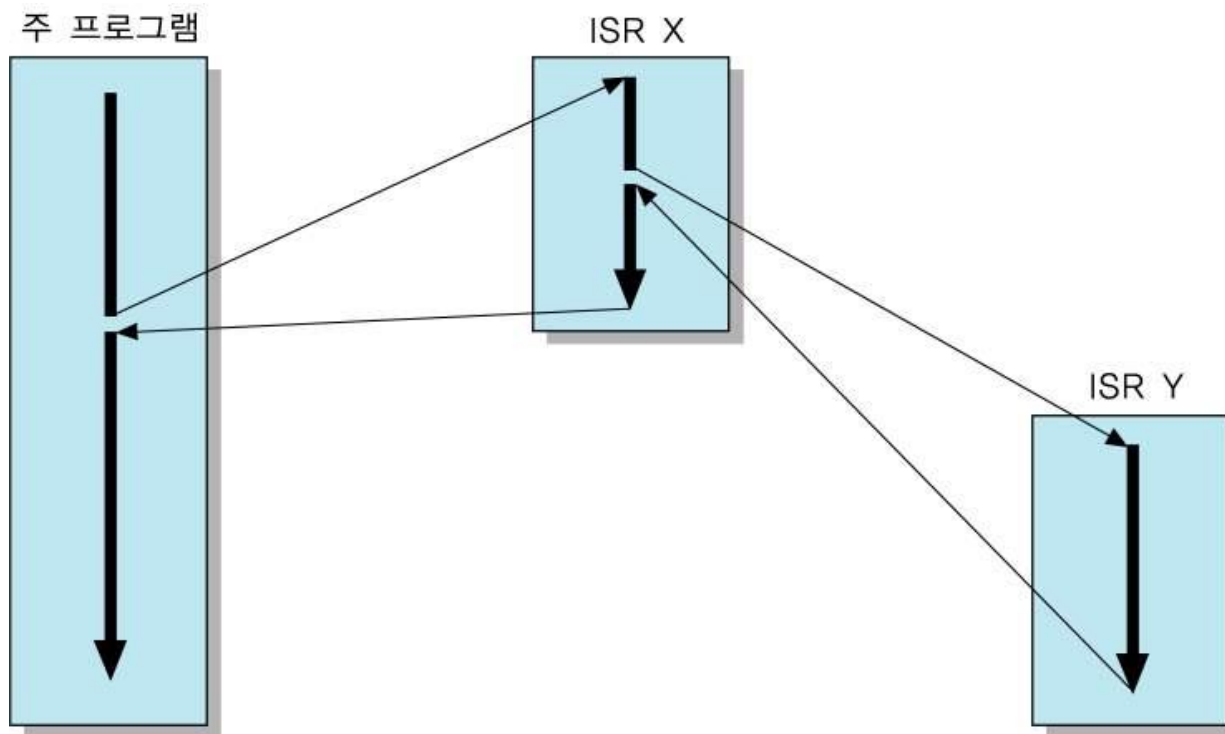
□ 인터럽트 서비스 루틴을 수행하는 동안에 다른 인터럽트 발생

□ 다중 인터럽트의 처리방법 (두 가지)

- ① CPU가 인터럽트 서비스 루틴을 처리하고 있는 도중에는 새로운 인터럽트 요구가 들어오더라도 인터럽트 사이클을 수행하지 않는 방법
 - 인터럽트 플래그(interrupt flag) ← 0 : 인터럽트 불가능(interrupt disabled) 상태
 - 시스템 운영상 중요한 프로그램이나 도중에 중단할 수 없는 데이터 입출력 동작 등을 위한 인터럽트를 처리하는데 사용
- ② 인터럽트의 **우선순위**를 정하고, 우선순위가 낮은 인터럽트가 처리되고 있는 동안에 우선순위가 더 높은 인터럽트가 들어온다면, 현재의 인터럽트 서비스 루틴의 수행을 중단하고 새로운 인터럽트를 먼저 처리

다중 인터럽트 처리 방법

- 장치 X를 위한 ISR X를 처리하는 도중에 우선 순위가 더 높은 장치 Y로부터 인터럽트 요구가 들어와서 먼저 처리되는 경우에 대한 제어의 흐름



2.2.4 간접 사이클(indirect cycle)

- 명령어에 포함되어 있는 주소를 이용하여, 그 명령어 실행에 필요한 데이터의 주소를 인출하는 사이클

→ 간접 주소지정 방식(indirect addressing mode)에서 사용

- 인출 사이클과 실행 사이클 사이에 위치
- 간접 사이클에서 수행될 마이크로-연산

$$t_0 : \text{MAR} \leftarrow \text{IR}(\text{addr})$$

$$t_1 : \text{MBR} \leftarrow \text{M}[\text{MAR}]$$

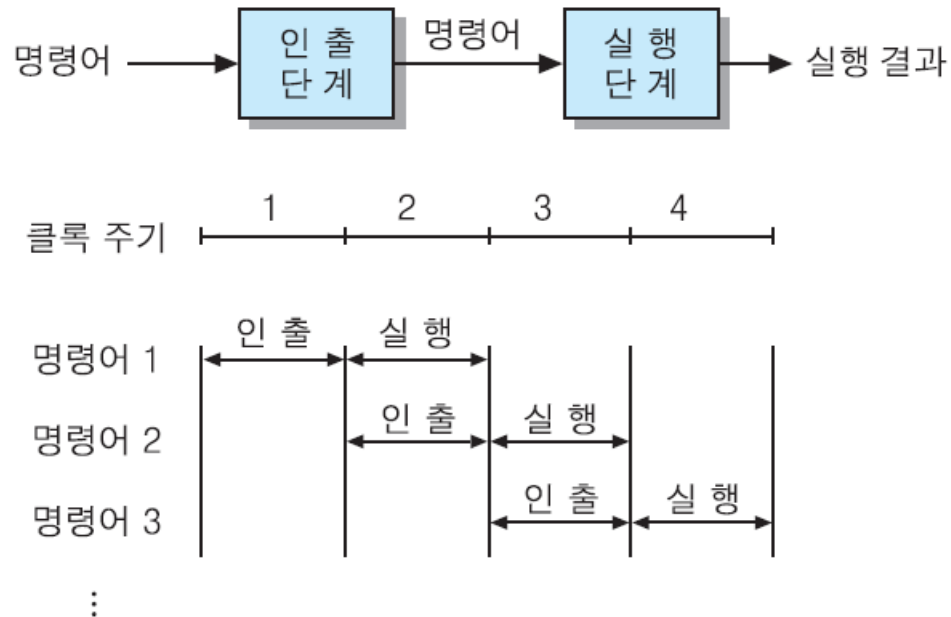
$$t_2 : \text{IR}(\text{addr}) \leftarrow \text{MBR}$$

- 인출된 명령어의 주소 필드 내용을 이용하여 기억장치로부터 데이터의 실제 주소를 인출하여 IR의 주소 필드에 저장

2.3 명령어 파이프라이닝(instruction pipelining)

- CPU의 프로그램 처리 속도를 높이기 위하여 CPU 내부 하드웨어를 여러 단계로 나누어 동시에 처리하는 기술
- 2-단계 명령어 파이프라인(two-stage instruction pipeline)
 - 명령어를 실행하는 하드웨어를 인출 단계(fetch stage)와 실행 단계(execute stage)라는 두 개의 독립적인 파이프라인 모듈로 분리
 - 두 단계들에 동일한 클록을 가하여 동작 시간을 일치시키면,
 - 첫 번째 클록 주기에서는 인출 단계가 첫 번째 명령어를 인출
 - 두 번째 클록 주기에서는 인출된 첫 번째 명령어가 실행 단계로 보내져서 실행되며, 그와 동시에 인출 단계는 두 번째 명령어를 인출(선인출(prefetch))

2-단계 명령어 파이프라인과 시간 흐름도



* 속도향상(S_p) = $6/4 = 1.5$ 배. 실행되는 명령어 수 증가 시, $S_p = 2$ 배에 접근

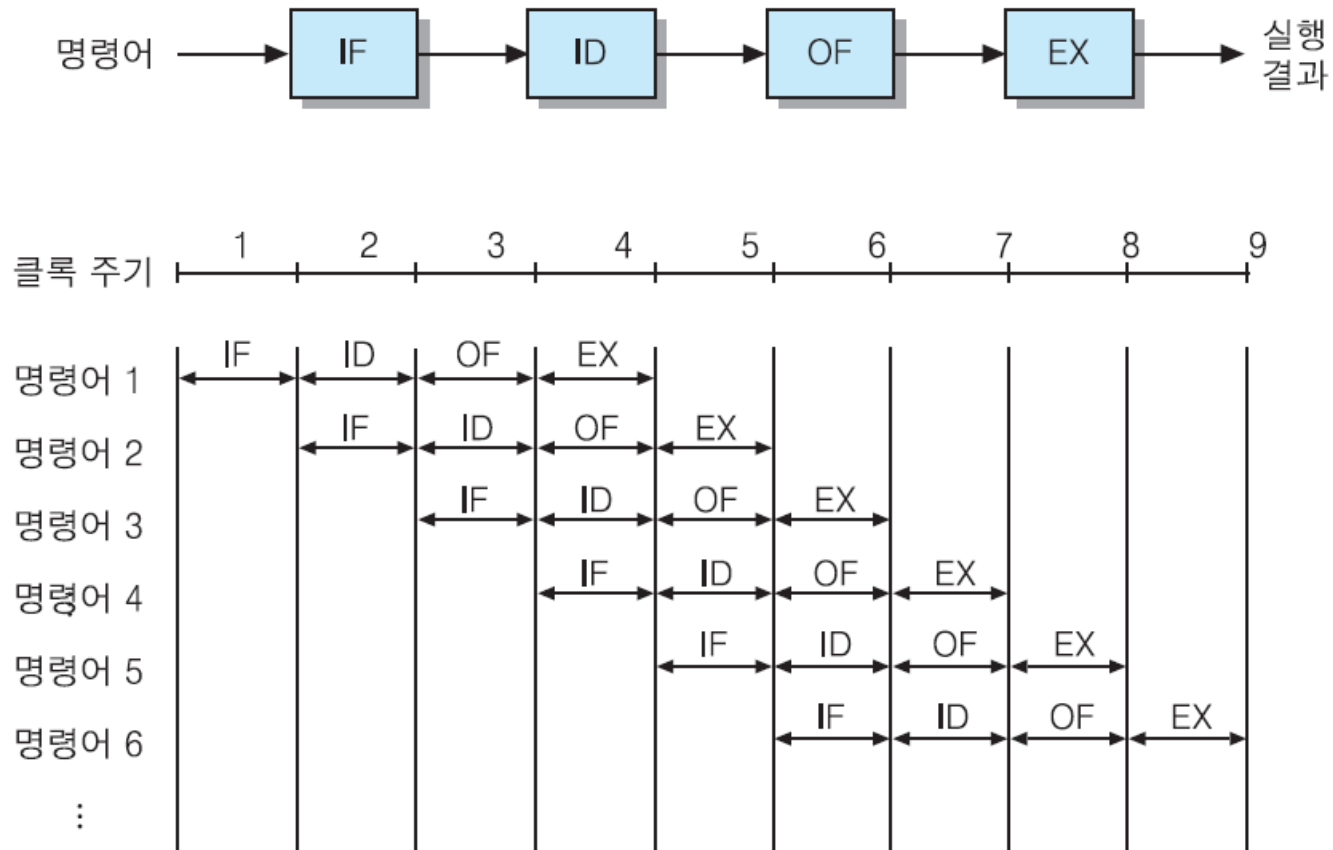
2-단계 명령어 파이프라인 (계속)

- 2-단계 파이프라인을 이용하면 명령어 처리 속도가 두 배 향상
- **문제점** : 두 단계의 처리 시간이 동일하지 않으면 두 배의 속도 향상을 얻지 못함(파이프라인 효율 저하)
 - **해결책** : 파이프라인 단계를 세분하여, 각 단계의 처리 시간을 (거의) 같아지도록 함
 - ➔ 파이프라인 단계의 수를 늘리면 전체적으로 속도향상(speedup)이 더 높아짐

4-단계 명령어 파이프라인

- 명령어 인출(IF) 단계 : 다음 명령어를 기억장치로부터 인출
- 명령어 해독(ID) 단계 : 해독기(decoder)를 이용하여 명령어를 해석
- 오퍼랜드 인출(OF) 단계 : 기억장치로부터 오퍼랜드를 인출
- 실행(EX) 단계 : 지정된 연산을 수행

4-단계 명령어 파이프라인과 시간 흐름도



파이프라인에 의한 전체 명령어 실행 시간

- 파이프라인 단계 수 = k
 - 실행할 명령어들의 수 = N
 - 각 파이프라인 단계가 한 클록 주기씩 걸린다고 가정한다면
- ◆ 파이프라인에 의한 전체 명령어 실행 시간(T_k):

$$T_k = k + (N - 1)$$

즉, 첫 번째 명령어를 실행하는데 k 주기가 걸리고,

나머지 $(N - 1)$ 개의 명령어들은 각각 한 주기씩만 소요

[비교] 파이프라인 되지 않은 경우의 N 개의 명령어들을 실행

시간 (T_1) : $T_1 = k \times N$

파이프라인에 의한 속도 향상(speedup)

$$S_p = \frac{T_1}{T_k} = \frac{k \times N}{k \times (N - 1)}$$

[예제 2-1]파이프라인에 의한 속도 향상

- 파이프라인 단계 수 = 4,
파이프라인 클럭 = 1GHz (각 단계에서의 소요시간 = 1ns)일 때,
10개의 명령어를 실행하는 경우의 속도향상은?

<풀이>

첫 번째 명령어 실행에 걸리는 시간 = 4ns

다음부터는 1ns 마다 한 개씩의 명령어 실행 완료

10개의 명령어 실행 시간 = $4 + (10 - 1) = 13\text{ns}$

➔속도향상(speedup: S_p) = $(10 \times 4) / 13 \doteq 3.08$ 배

N : CPU 가 실행하는 명령어 수라면,

$$N = 100 \text{ 일 때, } Sp = 400/103 = 3.88$$

$$N = 1000 \text{ 일 때, } Sp = 4000/1003 = 3.988$$

$$N = 10000 \text{ 일 때, } Sp = 40000/10003 = 3.9988$$

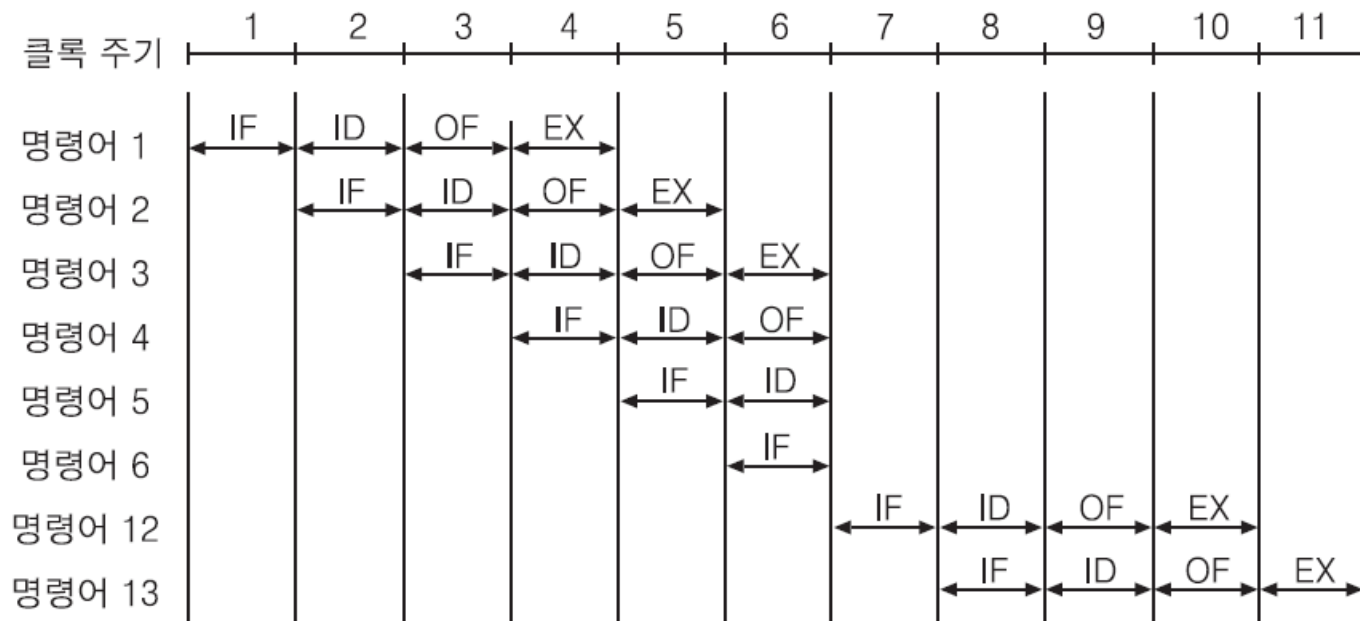
$$N \rightarrow \infty \text{ 일 때는 } Sp \rightarrow 4 \text{ (이론적 속도향상)}$$

파이프라인의 효율 저하 요인들

- 모든 명령어들이 파이프라인 단계들을 모두 거치지는 않는다
 - 어떤 명령어에서는 오퍼랜드를 인출할 필요가 없지만, 파이프라인의 하드웨어를 단순화시키기 위해서는 모든 명령어가 네 단계들을 모두 통과하도록 해야 한다
- 파이프라인의 클록은 처리 시간이 가장 오래 걸리는 단계를 기준으로 결정된다
- IF 단계와 OF 단계가 동시에 기억장치를 액세스하는 경우에, 기억장치 충돌(memory conflict)이 일어나면 지연이 발생한다
- 조건 분기(conditional branch) 명령어가 실행되면, 미리 인출하여 처리하던 명령어들이 무효화된다

조건 분기가 존재하는 경우의 시간 흐름도

[예] 명령어 3: **JZ 12** ; jump (if zero) to address 12



분기 발생에 의한 성능 저하의 최소화 방법

□ 분기 예측(branch prediction)

- 분기가 일어날 것인 지를 예측하고, 그에 따라 명령어를 인출하는 확률적 방법
- 분기 역사 표(branch history table) 이용하여 최근의 분기 결과를 참조

□ 분기 목적지 선인출(prefetch branch target) : 조건 분기가 인식되면, 분기 명령어의 다음 명령어뿐만 아니라 분기의 목적지 명령어도 함께 인출하여 실행하는 방법(조건 확인 결과에 따라 명령어를 선택하여 실행)

□ 루프 버퍼(loop buffer) 사용 : 파이프라인의 명령어 인출 단계에 포함되어 있는 작은 고속 기억장치인 루프 버퍼에 가장 최근 인출된 n개의 명령어들을 순서대로 저장해두는 방법

□ 지연 분기(delayed branch) : 분기 명령어의 위치를 재배치함으로써 파이프라인의 성능을 개선하는 방법

상태 레지스터(status register)

- ❑ 명령어 실행 결과에 따른 조건 플래그(condition flag)들 저장
- ❑ 조건 플래그의 종류
 - **부호(S) 플래그** : 직전에 수행된 산술연산 결과값의 부호 비트를 저장(양수: 0, 음수: 1)
 - **영(Z) 플래그** : 연산 결과값이 0 이면, 1로 세트
 - **올림수(C) 플래그** : 덧셈이나 뺄셈에서 올림수(carry)나 빌림수(borrow)가 발생한 경우에 1로 세트



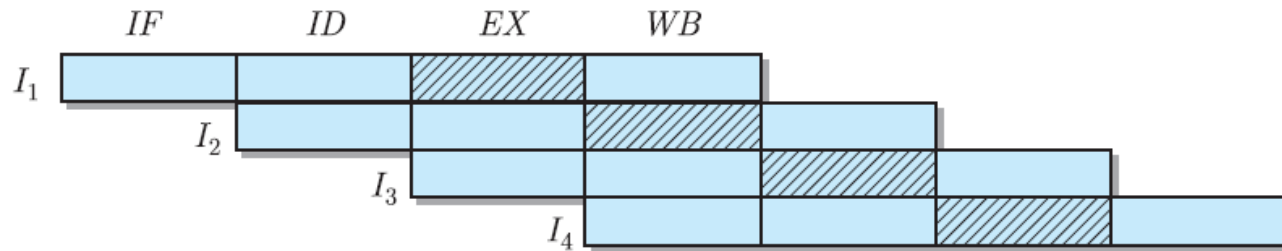
상태 레지스터 (계속)

- 동등(E) 플래그 : 두 수를 비교한 결과가 같게 나왔을 경우에 1로 세트
- 오버플로우(V) 플래그 : 산술 연산 과정에서 오버플로우가 발생한 경우에 1로 세트
- 인터럽트(I) 플래그
 - 인터럽트 가능(interrupt enabled) 상태이면, 0으로 세트
 - 인터럽트 불가능(interrupt disabled) 상태이면, 1로 세트
- 슈퍼바이저(P) 플래그 :
 - CPU의 실행 모드가 슈퍼바이저 모드(supervisor mode)이면, 1로 세트
 - 사용자 모드(user mode)이면, 0으로 세트

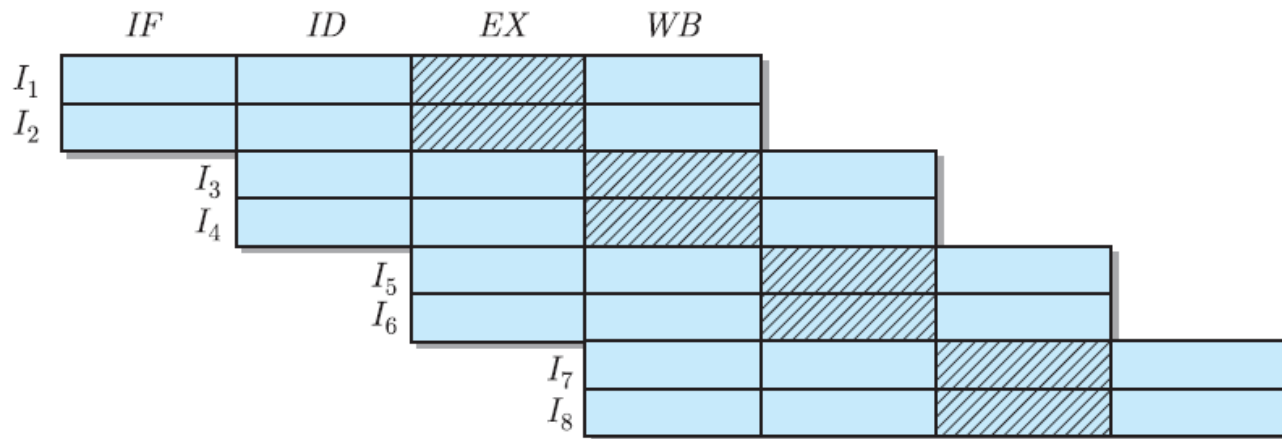
2.3.3 슈퍼스칼라(Superscalar)

- CPU의 처리 속도를 더욱 높이기 위하여 내부에 두 개 혹은 그 이상의 명령어 파이프라인들을 포함시킨 구조
- 매 클럭 주기마다 각 명령어 파이프라인이 별도의 명령어를 인출하여 동시에 실행할 수 있기 때문에, 이론적으로는 프로그램 처리 속도가 파이프라인의 수만큼 향상 가능
- 파이프라인의 수 = m : m -way 슈퍼스칼라

2-way 슈퍼스칼라의 명령어 실행 흐름도



(a) 일반적인 파이프라인의 명령어 실행 시간도



(b) 2-way 슈퍼스칼라의 명령어 실행 시간도

슈퍼스칼라에 의한 속도향상(speedup: Sp)

□ 단일 파이프라인에 의한 실행 시간 (N : 실행할 명령어 수)

- $T(1) = k + N - 1$

□ m -way 슈퍼스칼라에 의한 실행 시간

- $T(m) = k + \frac{N - m}{m}$

□ 속도 향상

- $Sp = \frac{T(1)}{T(m)} = \frac{k + N - 1}{k + (N - m)/m} = \frac{m(k + N - 1)}{N + m(k - 1)}$

□ 명령어 수 $N \rightarrow \infty$, $Sp \rightarrow m$

슈퍼스칼라 (계속)

□ 슈퍼스칼라의 속도 저하 ($S_p < m$) 요인 :

- 명령어들 간의 데이터 의존 관계
- 하드웨어(레지스터, 캐시, 기억장치, 등) 이용에 대한 경합 발생
→ 동시 실행 가능한 명령어 수 $< m$

<해결책>

- 명령어 실행 순서 재배치
→ 명령어들 간의 데이터 의존성 제거
- 하드웨어 추가(중복) 설치
→ 하드웨어(레지스터, 캐시, 등)에 대한 경합 감소

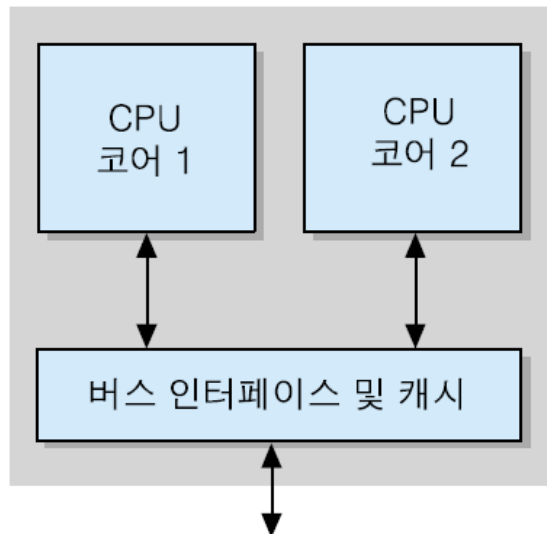
2.3.4 듀얼-코어 및 멀티-코어

- CPU 코어(core) : 명령어 실행에 필요한 CPU 내부의 핵심 하드웨어 모듈(슈퍼스칼라 H/W, ALU, 레지스터 등)
- 멀티-코어 프로세서(multi-core processor): 여러 개의 CPU 코어들을 하나의 칩에 포함시킨 프로세서
 - 듀얼-코어(dual-core): 두 개의 CPU 코어 포함
 - 쿼드-코어(quad-core): 네 개의 CPU 코어 포함
 - 헥사-코어(hexa-core), 옥타-코어(octa-core)도 출시 중
- 칩-레벨 다중프로세서(chip-level multiprocessor) 혹은 단일-칩 다중프로세서(multiprocessor-on-a-chip)이라고도 부름

듀얼-코어 및 멀티-코어 (계속)

□ 듀얼-코어 프로세서

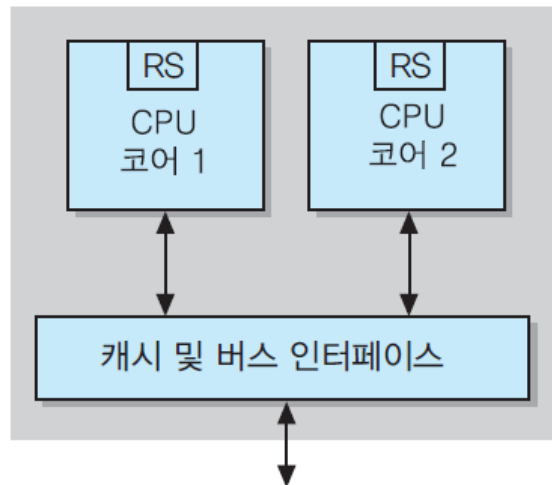
- 단일-코어 슈퍼스칼라 프로세서에 비하여 2배의 속도 향상 기대
- 코어들은 내부 캐시와 시스템 버스 인터페이스만 공유
- 코어 별로 독립적 프로그램 실행 → 멀티-태스킹(multi-tasking) 혹은 멀티-스레딩(multi-threading) 지원



듀얼-코어 및 멀티-코어 (계속)

□ 멀티-스레딩

- 스레드(thread): 독립적으로 실행될 수 있는 최소 크기의 프로그램 단위
- **단일-스레드 모델(그림 (a))**: 각 코어가 스레드를 한 개씩 처리 ---
 - 처리 중의 스레드에 대한 시스템 상태, 데이터 및 주소 정보를 레지스터 세트(register set: RS)에 저장
 - RS: 프로그램 카운터(PC), 스택 포인터(SP), 상태 레지스터, 데이터 레지스터, 주소 레지스터, 등

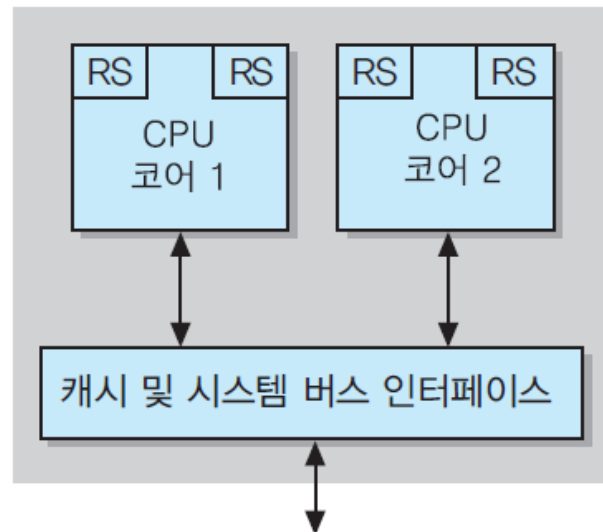


(a) 단일-스레드 모델

듀얼-코어 및 멀티-코어 (계속)

■ 멀티-스레드 모델(그림 (b)):

- 각 코어는 두 개의 RS들을 포함하며, 스레드를 두 개씩 처리
- 두 스레드들이 CPU 코어의 H/W 자원들(ALU, 부동소수점유닛, 온-칩 캐시, TLB, 등)을 공유
- 처리 중인 각 스레드에 대한 시스템 상태, 데이터 및 주소 정보는 서로 다른 레지스터 세트(RS)에 저장



(b) 멀티-스레드 모델

듀얼-코어 및 멀티-코어 (계속)

- 듀얼-코어 멀티-스레드 프로세서: ‘두 개의 물리적 프로세서 (physical processor)들이 네 개의 논리적 프로세서(logical processor)들로 구성되어 있다’라고 정의하기도 함[INTi17 참조]
- 멀티-코어 멀티-스레딩 프로세서의 사례
 - Intel i7-8500Y : 2-코어 4-스레드 프로세서
 - Intel i7-8565U : 4-코어 8-스레드 프로세서

2.4 명령어 세트(instruction set)

- ❑ 어떤 CPU를 위하여 정의되어 있는 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

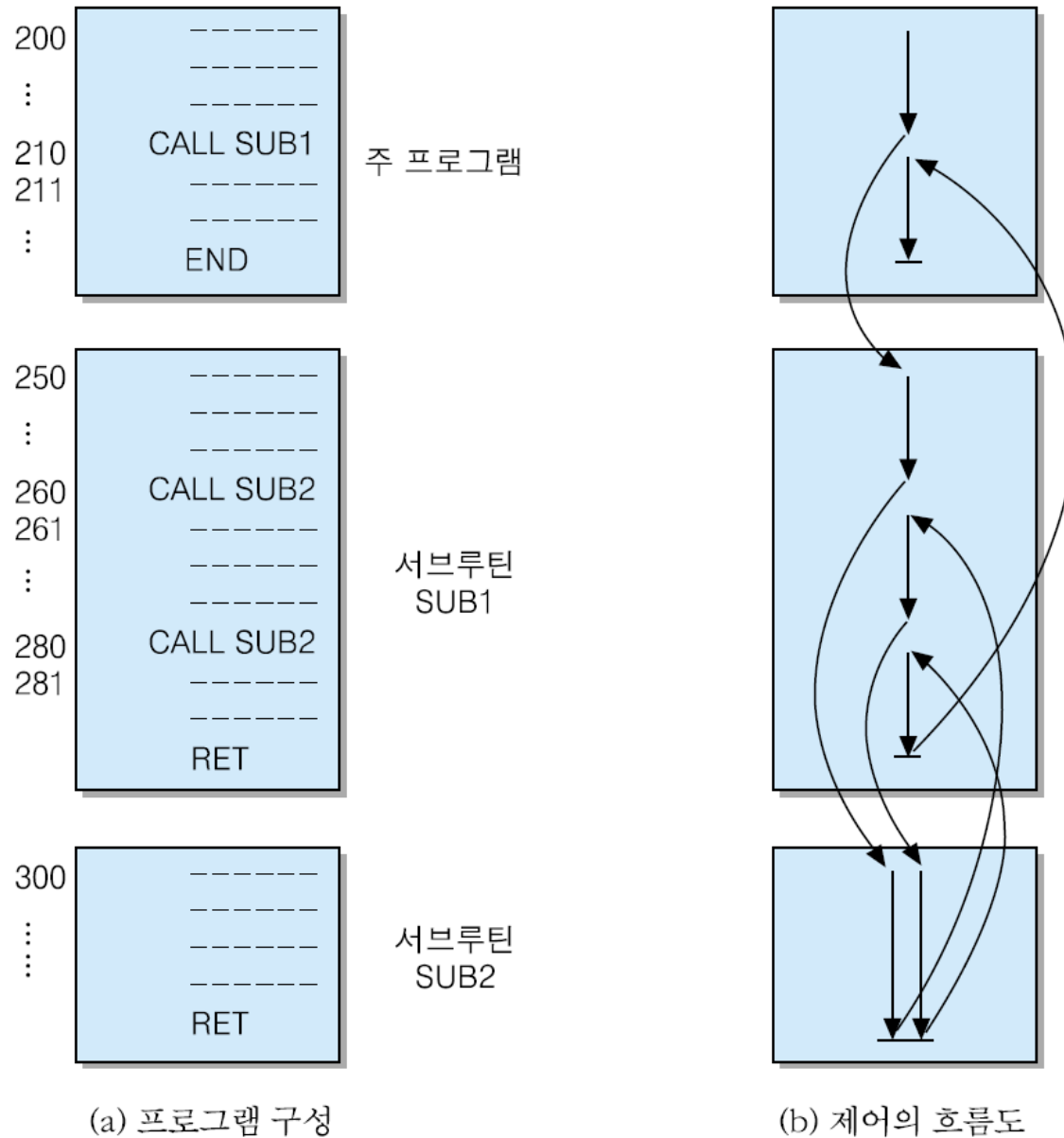
2.4.1 연산의 종류

- **데이터 전송** : 레지스터와 레지스터 간, 레지스터와 기억장치 간, 혹은 기억장치와 기억장치 간에 데이터를 이동하는 동작
- **산술 연산** : 덧셈, 뺄셈, 곱셈 및 나눗셈과 같은 기본적인 산술 연산들
- **논리 연산** : 데이터의 각 비트들 간에 대한 AND, OR, NOT 및 exclusive-OR 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)

서브루틴 호출을 위한 명령어들

- **CALL 명령어** : 현재의 PC 내용을 스택에 저장하고 서브루틴의 시작 주소로 분기하는 명령어
- **RET 명령어** : CPU가 원래 실행하던 프로그램으로 복귀(return)시키는 명령어

서브루틴이 포함된 프로그램이 수행되는 순서



CALL/RET 명령어의 마이크로 연산

□ **CALL X** 명령어에 대한 마이크로-연산

$t_0 : \text{MBR} \leftarrow \text{PC}$

$t_1 : \text{MAR} \leftarrow \text{SP}, \text{PC} \leftarrow \text{X}$

$t_2 : \text{M}[\text{MAR}] \leftarrow \text{MBR}, \text{SP} \leftarrow \text{SP} - 1$

- 현재의 **PC** 내용(서브루틴 수행 완료 후에 복귀할 주소)을 **SP**가 지정하는 스택의 최상위(top of stack: TOS)에 저장
- 만약 주소지정 단위가 바이트이고 저장될 주소는 16비트라면,

$\text{SP} \leftarrow \text{SP} - 2$ 로 변경

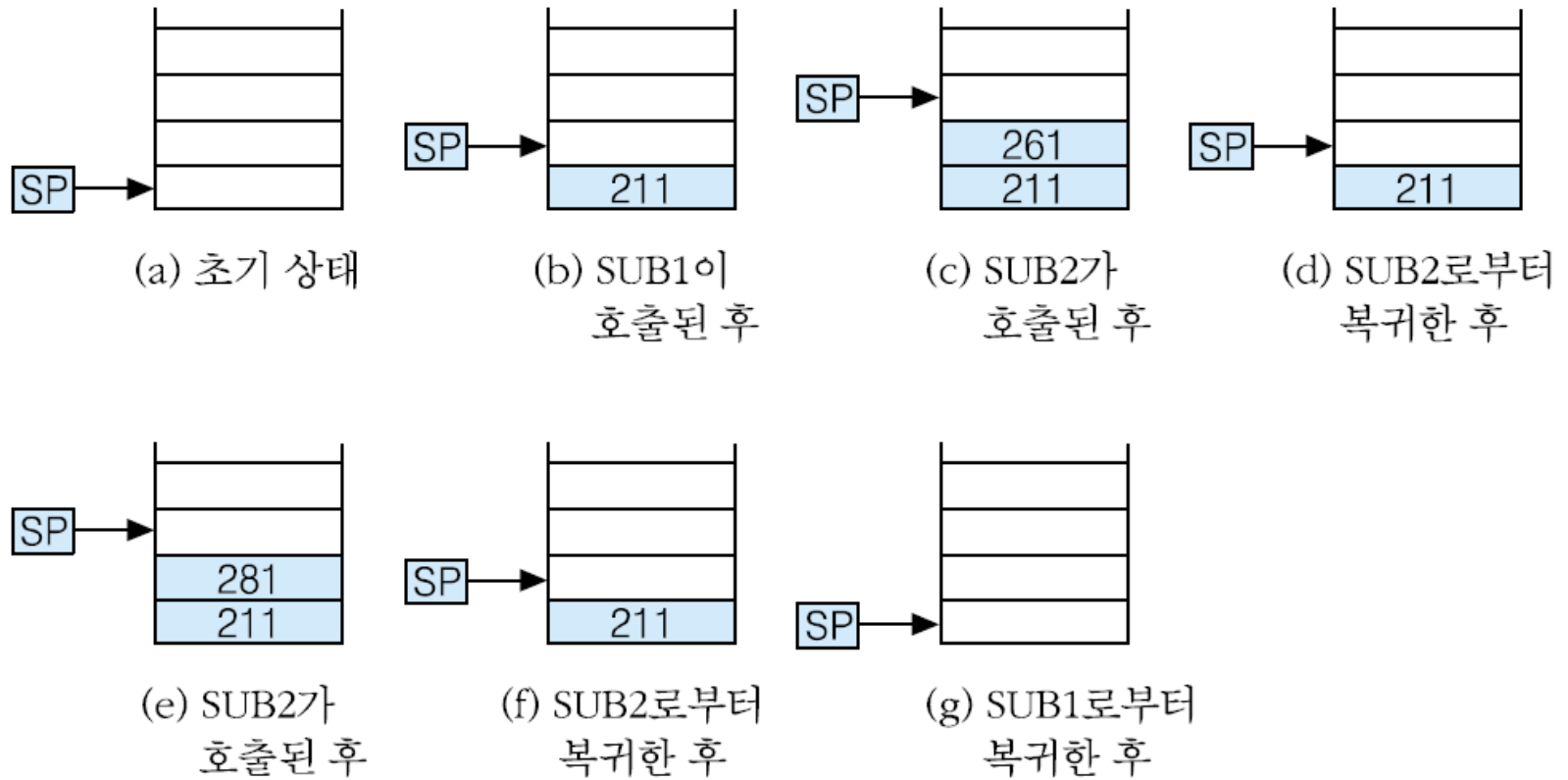
□ **RET** 명령어의 마이크로-연산

$t_0 : \text{SP} \leftarrow \text{SP} + 1$

$t_1 : \text{MAR} \leftarrow \text{SP}$

$t_2 : \text{PC} \leftarrow \text{M}[\text{MAR}]$

그림 2-17의 프로그램 수행 과정에서 스택의 변화



2.4.2 명령어 형식

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 수
- **필드(field)** : 명령어의 각 구성 요소들에 소요되는 비트들의 그룹
- **명령어의 길이** = 단어(word) 길이

[예] 세 개의 필드들로 구성된 16-비트 명령어



명령어 형식의 결정에서 고려할 사항들

□ 연산 코드 필드 길이 : 연산의 개수를 결정

[예] 4 비트 $\rightarrow 2^4 = 16$ 가지의 연산 정의 가능

- 만약 연산 코드 필드가 5 비트로 늘어나면, $2^5 = 32$ 가지 연산들 정의 가능 \rightarrow 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 주소를 지정할 수 있는 기억장치 용량 결정
 - 레지스터 번호 : 데이터 저장에 사용될 수 있는 레지스터의 개수 결정

오퍼랜드 필드 범위의 예

- 오퍼랜드1은 레지스터 번호를 지정하고, 오퍼랜드2는 기억장치 주소를 지정하는 경우
 - 오퍼랜드1 : 4 비트 → 16 개의 레지스터 사용 가능
 - 오퍼랜드2 : 8 비트 → 기억장치의 주소 범위 : 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 지정 가능

오퍼랜드의 수에 따른 명령어 분류

- **1-주소 명령어(one-address instruction)** : 오퍼랜드를 한 개만 포함하는 명령어. (다른 한 오퍼랜드는 묵시적으로 AC가 됨)

[예] **ADD X ; $AC \leftarrow AC + M[X]$**

- **2-주소 명령어(two-address instruction)** : 두 개의 오퍼랜드를 포함하는 명령어.

[예] **ADD R1, R2 ; $R1 \leftarrow R1 + R2$**

MOV R1, R2 ; $R1 \leftarrow R2$

ADD R1, X ; $R1 \leftarrow R1 + M[X]$

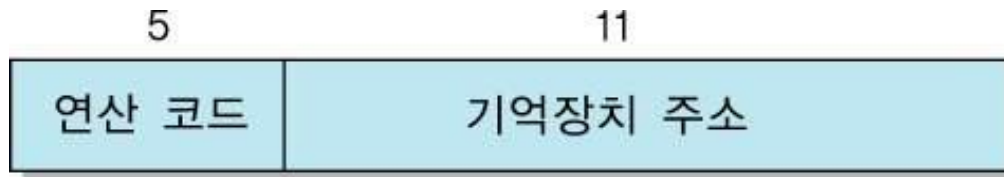
- **3-주소 명령어(three-address instruction)** : 세 개의 오퍼랜드들을 포함하는 명령어.

[예] **ADD R1, R2, R3 ; $R1 \leftarrow R2 + R3$**

1-주소 명령어의 예

- 길이가 16 비트인 1-주소 명령어에서 연산 코드가 5 비트인 경우의 명령어 형식을 정의하고, 주소지정 가능한 기억장치 용량을 결정하라

- 명령어 형식(instruction format)



- 주소지정 가능한 기억장치 용량 : $2^{11} = 2048$ 바이트

2-주소 명령어의 예

- 2-주소 명령어 형식을 사용하는 16-비트 CPU에서 연산 코드가 5 비트이고, 레지스터의 수는 8 개이다. (a) 두 오퍼랜드들이 모두 레지스터 번호인 경우와, (b) 한 오퍼랜드는 기억장치 주소인 경우의 명령어 형식을 정의하라



(a) 두 개의 레지스터 오퍼랜드들을 가지는 경우

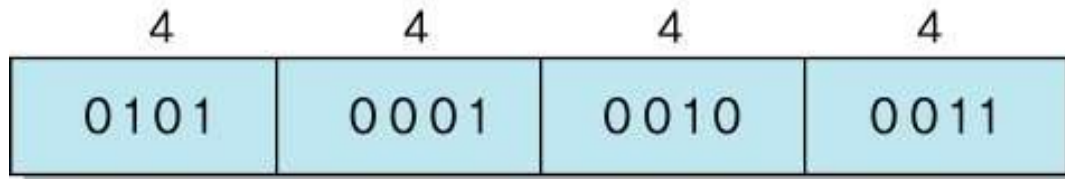


(b) 한 오퍼랜드는 기억장치 주소인 경우

3-주소 명령어 형식의 예



(a) 명령어 형식



(b) ADD R1,R2,R3 명령어의 비트 배열

명령어 형식이 프로그래밍에 미치는 영향 (예)

[예] $X = (A + B) \times (C - D)$ 계산을 위한 어셈블리 프로그램 작성

□ 아래와 같은 니모닉을 가진 명령어들을 사용

- **ADD** : 덧셈
- **SUB** : 뺄셈
- **MUL** : 곱셈
- **DIV** : 나눗셈
- **MOV** : 데이터 이동
- **LOAD** : 기억장치로부터 데이터 적재
- **STOR** : 기억장치로 데이터 저장

1-주소 명령어를 사용한 프로그램

LOAD A ; $AC \leftarrow M[A]$
ADD B ; $AC \leftarrow AC + M[B]$
STOR T ; $M[T] \leftarrow AC$
LOAD C ; $AC \leftarrow M[C]$
SUB D ; $AC \leftarrow AC - M[D]$
MUL T ; $AC \leftarrow AC \times M[T]$
STOR X ; $M[X] \leftarrow AC$

단, $M[A]$ 는 기억장치 A번지의 내용,

T는 기억장치 내 임시 저장장소의 주소

□ 프로그램의 길이 = 7

2-주소 명령어를 사용한 프로그램

MOV R1, A ; $R1 \leftarrow M[A]$
ADD R1, B ; $R1 \leftarrow R1 + M[B]$
MOV R2, C ; $R2 \leftarrow M[C]$
SUB R2, D ; $R2 \leftarrow R2 - M[D]$
MUL R1, R2 ; $R1 \leftarrow R1 \times R2$
MOV X, R1 ; $M[X] \leftarrow R1$

□ 프로그램의 길이 = 6

3-주소 명령어를 사용한 프로그램

ADD R1, A, B ; $R1 \leftarrow M[A] + M[B]$

SUB R2, C, D ; $R2 \leftarrow M[C] - M[D]$

MUL X, R1, R2 ; $M[X] \leftarrow R1 \times R2$

❑ 프로그램의 길이 = 3

❑ 단점

- 명령어의 길이가 증가한다
- 명령어 해독 과정이 복잡해지고, 실행 시간이 길어진다

2.4.3 주소지정 방식(addressing mode)

- 명령어 실행에 필요한 오퍼랜드의 주소를 결정하는 방식
- 다양한 주소지정 방식을 사용하는 이유 : 제한된 수의 명령어 비트들을 이용하여, 사용자(프로그래머)가 여러 가지 방법으로 오퍼랜드의 주소를 결정하도록 해주며, 더 큰 용량의 기억장치를 사용할 수 있도록 하기 위함
- 명령어 내 오퍼랜드 필드의 내용
 - 기억장치 주소 : 데이터가 저장된 기억장치의 위치를 지정
 - 레지스터 번호 : 데이터가 저장된 레지스터를 지정
 - 데이터 : 명령어의 오퍼랜드 필드에 데이터가 포함

□ 기호

- ***EA*** : 유효 주소(Effective Address), 데이터가 저장된 기억장치의 실제 주소
- ***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 *R*의 내용

주소지정 방식의 종류

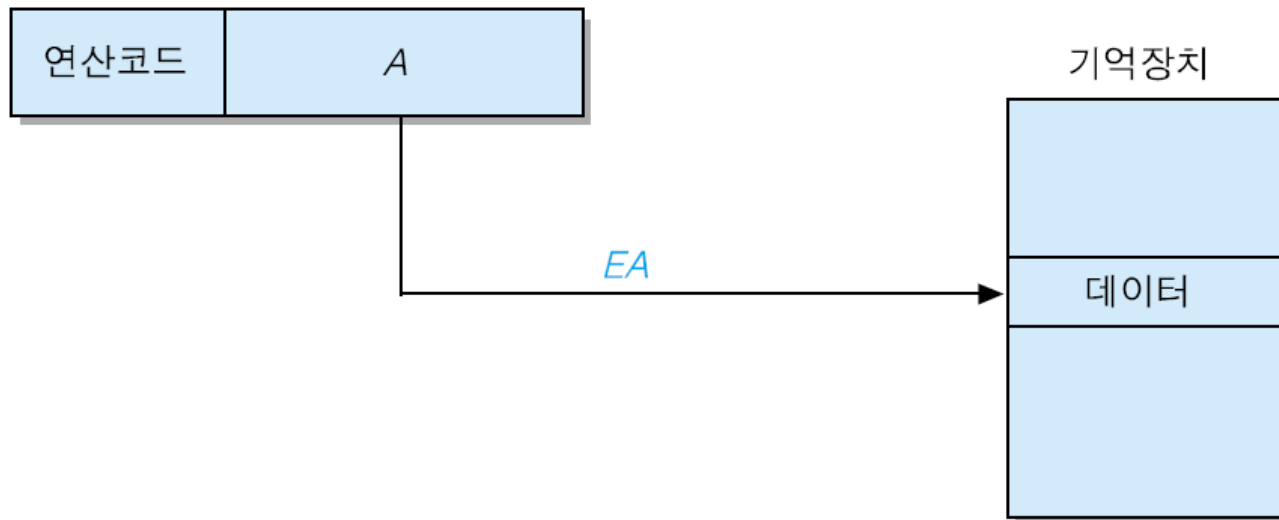
- ❑ 직접 주소지정 방식(direct addressing mode)
- ❑ 간접 주소지정 방식(indirect addressing mode)
- ❑ 묵시적 주소지정 방식(implied addressing mode)
- ❑ 즉시 주소지정 방식(immediate addressing mode)
- ❑ 레지스터 주소지정 방식(register addressing mode)
- ❑ 레지스터 간접 주소지정 방식(register-indirect addressing mode)
- ❑ 변위 주소지정 방식(displacement addressing mode)
 - 상대 주소지정 방식(relative addressing mode)
 - 인덱스 주소지정 방식(indexed addressing mode)
 - 베이스-레지스터 주소지정 방식(base-register addressing mode)

1) 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 기억장소의 수가 제한



예제 2-6

CPU 내부 레지스터들과 주기억장치에 그림 2-24와 같은 값들이 저장되어 있다고 가정하자. 여기서, CPU 레지스터 및 각 기억 장소의 폭(width)은 16비트이며, 그림에서 모든 값들은 편의상 10진수로 표시하였다(그림 2-24는 [예제 2-7] 및 [예제 2-9]에서도 공통적으로 사용됨).

CPU 레지스터		주소	기억장치
PC	450	⋮	
IX	003	150	1234
BR	500	151	5678
R0		172	0202
R1	203	173	—
R2	151	⋮	
R3		⋮	
R4		201	—
⋮		202	3256
⋮		203	4457
		⋮	

그림 2-24 예제 2-6, 2-7, 2-9를 위한 그림

[예제 2-6 (계속)]

- (1) 직접 주소지정 방식을 사용하는 명령어의 주소 필드(A)에 저장된 내용이 150일 때, 유효 주소(EA) 및 그에 의해 인출되는 데이터를 구하라.
- (2) 명령어 길이가 16비트이고 연산 코드가 5비트라면, 이 명령어에 의해 직접 주소 지정 될 수 있는 기억장치 용량은 얼마인가?

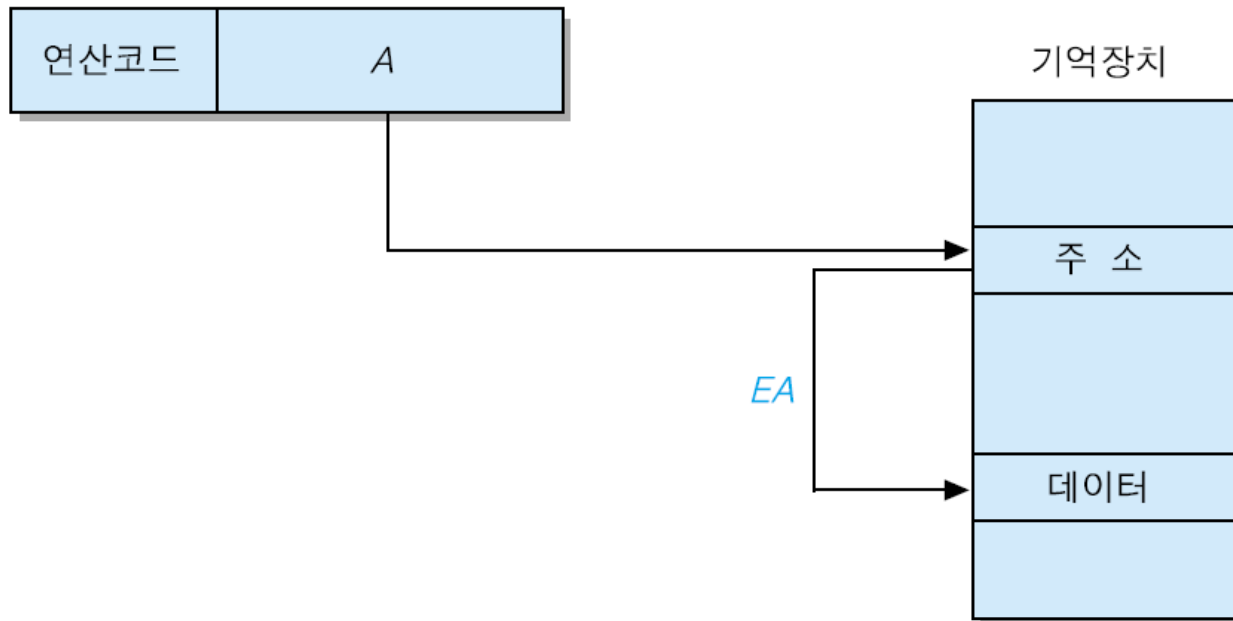
풀이

- (1) $EA = 150$ 이므로, 기억장치 150번지에 저장된 데이터 '1234'가 인출된다.
- (2) 주소 필드가 11비트이므로, 직접 주소지정 할 수 있는 기억장치 용량은 $2^{11} = 2048$ 단어가 된다. 그런데 각 기억 장소에 저장되는 데이터의 비트 수가 16비트 (2바이트)이므로, 기억장치 용량은 4096바이트로 표현할 수도 있다.

2) 간접 주소지정 방식(indirect addressing mode)

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

$$EA = (A)$$



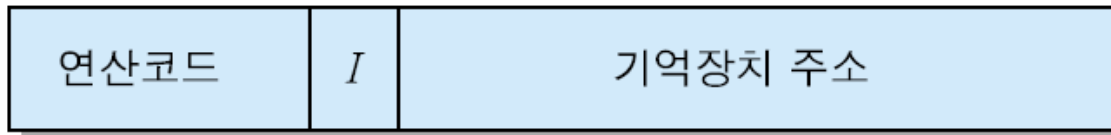
간접 주소지정 방식 (계속)

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

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

간접 주소지정 방식 (계속)

- 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
 - $EA = ((\dots (A) \dots))$

예제 2-7

CPU 레지스터들과 주기억장치에 그림 2-24와 같은 값들이 저장되어 있을 때 아래 물음에 답하라.

- (1) 간접 주소지정 방식을 사용하는 명령어의 주소 필드(A)에 저장된 내용이 '172'라고 가정했을 때, 유효 주소(EA) 및 그에 의해 인출되는 데이터를 구하라.
- (2) 이 명령어에 의해 주소지정 될 수 있는 기억장치 용량은 얼마인가?

풀이

- (1) EA는 그림 2-24의 기억장치 172번지에 저장되어 있는 '202'이다. 따라서 명령어 실행에 사용될 데이터로는 기억장치 202번지에 저장되어 있는 '3256'이 인출된다.
- (2) 주소(EA)의 길이는 16비트가 되므로, 주소지정 할 수 있는 기억장치 용량은 $2^{16} = 64\text{Kword}(128\text{KByte})$ 가 된다.

3) 묵시적 주소지정 방식(implied addressing mode)

- 명령어 실행에 필요한 데이터의 위치가 묵시적으로 지정되는 방식

[예]

- ‘SHL’ 명령어 : 누산기의 내용을 좌측으로 시프트(shift)
- ‘PUSH R1’ 명령어 : 레지스터 R1의 내용을 스택에 저장 (SP가 가리키는 기억 장소에 R1의 내용을 저장한다는 것이 묵시적으로 정해져 있음)

- 장점 : 명령어 길이가 짧다
- 단점 : 종류가 제한된다

4) 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 제한 됨

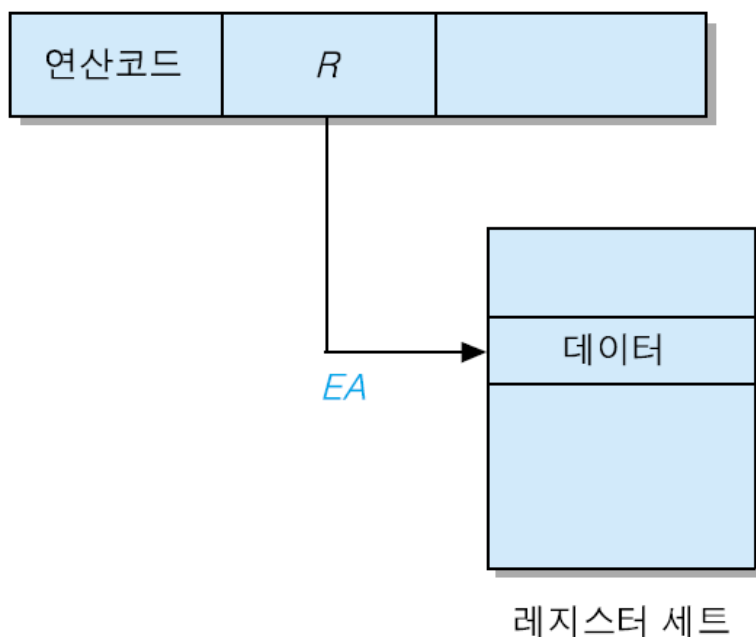


5) 레지스터 주소지정 방식

- 연산에 사용될 데이터가 내부 레지스터에 저장되어 있는 경우로서, 명령어의 오퍼랜드가 해당 레지스터를 가리키는 방식

$$EA = R$$

- 주소지정에 사용될 수 있는 레지스터들의 수 = 2^k 개



(단, k 는 R 필드의 비트 수)

레지스터 주소지정 방식 (계속)

□ 장점

- 오퍼랜드 필드의 비트 수가 적어도 된다
- 데이터 인출을 위하여 기억장치 액세스가 필요 없다

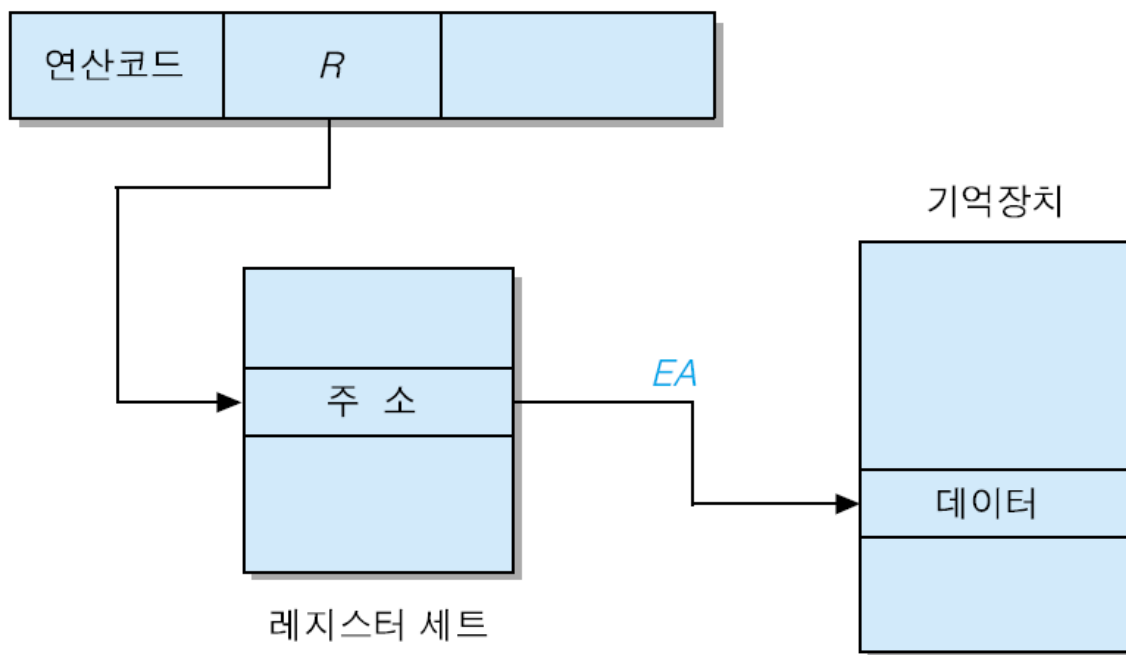
□ 단점

- 데이터가 저장될 수 있는 공간이 CPU 내부 레지스터들로 제한

6) 레지스터 간접 주소지정 방식

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

$$EA = (R)$$



예제 2-9

CPU 내부 레지스터들과 주기억장치에 그림 2-24와 같은 값들이 저장되어 있고, 그림 2-28과 2-29의 명령어 형식에서 레지스터 필드 R에는 '2'가 저장되어 있다고 가정하자.

- (1) 레지스터 주소지정 방식이 사용된다면, 연산 처리 과정에서 어떤 데이터가 사용될 것인가?
- (2) 레지스터 간접 주소지정 방식이 사용된다면, 어떤 데이터가 사용될 것인가?

풀이

- (1) R2에 저장되어 있는 데이터 '151'이 사용된다.
- (2) 기억장치 151번지에 저장되어 있는 데이터 '5678'이 사용된다.

레지스터 간접 주소지정 방식 (계속)

□ **장점** : 주소지정 할 수 있는 기억장치 영역이 확장

- 레지스터의 길이 = 16 비트라면, 주소지정 영역:
 $2^{16} = 64 \text{ Kword}$
- 레지스터의 길이 = 32 비트라면, 주소지정 영역:
 $2^{32} = 4 \text{ Gword}$

7) 변위 주소지정 방식(displacement addressing)

- 직접 주소지정과 레지스터 간접 주소지정 방식의 조합

$$EA = A + (R)$$

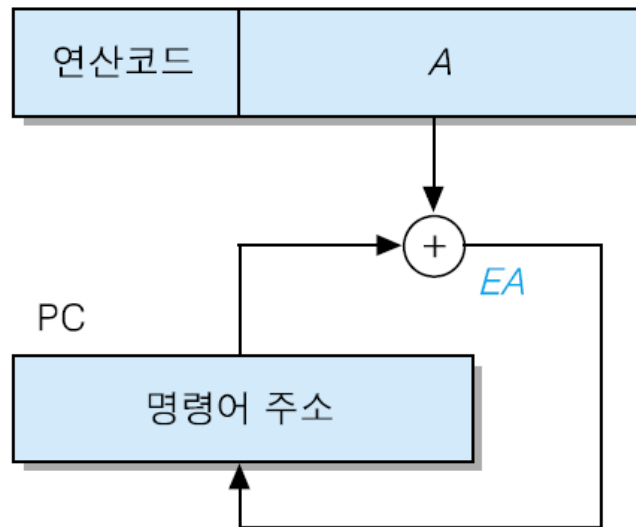
- 사용되는 레지스터에 따라 다양한 변위 주소지정 방식 가능
 - PC → 상대 주소지정 방식(relative addressing mode)
 - 인덱스 레지스터 → 인덱스 주소지정 방식(indexed addressing mode)
 - 베이스 레지스터 → 베이스-레지스터 주소지정 방식(base-register addressing mode)

상대 주소지정 방식(relative addressing mode)

- 프로그램 카운터(PC)를 레지스터로 사용하여 EA를 계산

$$EA = A + (PC) \quad \text{단, } A \text{는 2의 보수}$$

- 주로 분기 명령어에서 사용
 - $A > 0$: 앞(forward) 방향으로 분기
 - $A < 0$: 뒤(backward) 방향으로 분기



예제 2-10

상대 주소지정 방식을 사용하는 JUMP 명령어가 450번지에 저장되어 있다.

- (1) 만약 오퍼랜드 $A = '21'$ 이라면, 몇 번지로 점프하는가?
- (2) 만약 오퍼랜드 $A = '-50'$ 이라면, 몇 번지로 점프하는가?

풀이

- (1) 이 명령어가 인출된 후에는 PC의 내용이 451로 증가된다. 따라서 $451 + 21 = 472$ 번지로 점프하게 된다.
- (2) (1)번과 같은 원리로, $451 - 50 = 401$ 번지로 점프하게 된다.

상대 주소지정 방식 (계속)

- **장점** : 전체 기억장치 주소가 명령어에 포함되어야 하는 일반적인 분기 명령어보다 적은 수의 비트만 필요
- **단점** : 분기 범위가 오퍼랜드 필드의 길이에 의해 제한
(오퍼랜드 비트들로 표현 가능한 2의 보수 범위)

인덱스 주소지정 방식(indexed addressing mode)

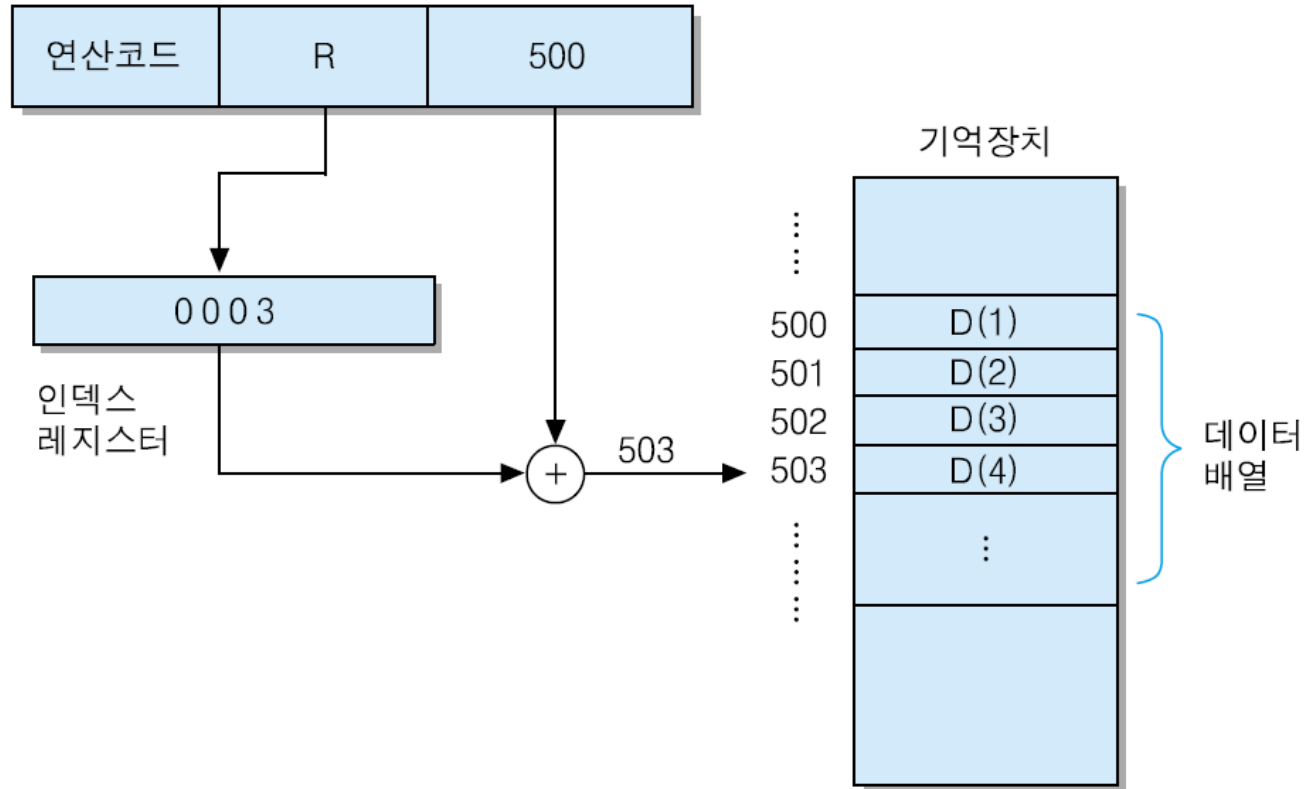
- 인덱스 레지스터의 내용과 변위 A를 더하여 유효 주소를 결정

$$EA = (IX) + A$$

- 인덱스 레지스터(IX) : 인덱스(index) 값을 저장하는 특수-목적 레지스터

- 주요 용도 : 배열 데이터 액세스

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



인덱스 주소지정 방식 (계속)

□ 자동 인덱싱(auto-indexing)

- 명령어가 실행될 때마다 인덱스 레지스터의 내용이 자동적으로 증가 혹은 감소
- 이 방식이 사용된 명령어가 실행되면 아래의 두 연산이 연속적으로 수행됨

$$EA = (IX) + A$$

$$IX \leftarrow IX + 1$$

베이스-레지스터 주소지정 방식

- 베이스 레지스터의 내용과 변위 A를 더하여 유효 주소를 결정

$$EA = (BR) + A$$

- 주요 용도: 프로그램의 위치 지정(혹은 변경)에 사용

[예] 다중프로그래밍(multiprogramming) 환경에서 프로그램 코드 및 데이터를 다른 위치로 이동시켜야 할 때, 분기 명령어나 데이터 액세스 명령어들의 주소 필드 내용을 바꿀 필요 없이, BR 레지스터의 내용만 변경하면 된다.

2.4.4 실제 상용 프로세서들의 명령어 형식

□ CISC(Complex Instruction Set Computer) 프로세서

- 명령어들의 수가 많음
- 명령어 길이가 일정하지 않음(명령어 종류에 따라 달라짐)
- 주소지정 방식이 매우 다양함 → 명령어 실행 시간이 길다

[예] PDP 계열 프로세서, Intel 펜티엄 계열 프로세서

□ RISC(Reduced Instruction Set Computer) 프로세서

- 명령어들의 수를 최소화
- 명령어 길이를 일정하게 고정
- 주소지정 방식의 종류를 단순화

[예] ATmega microcontroller, ARM 계열 프로세서

2.4.4 실제 상용 프로세서들의 명령어 형식

□ PDP-10 프로세서 : 고정 길이의 명령어 형식 사용

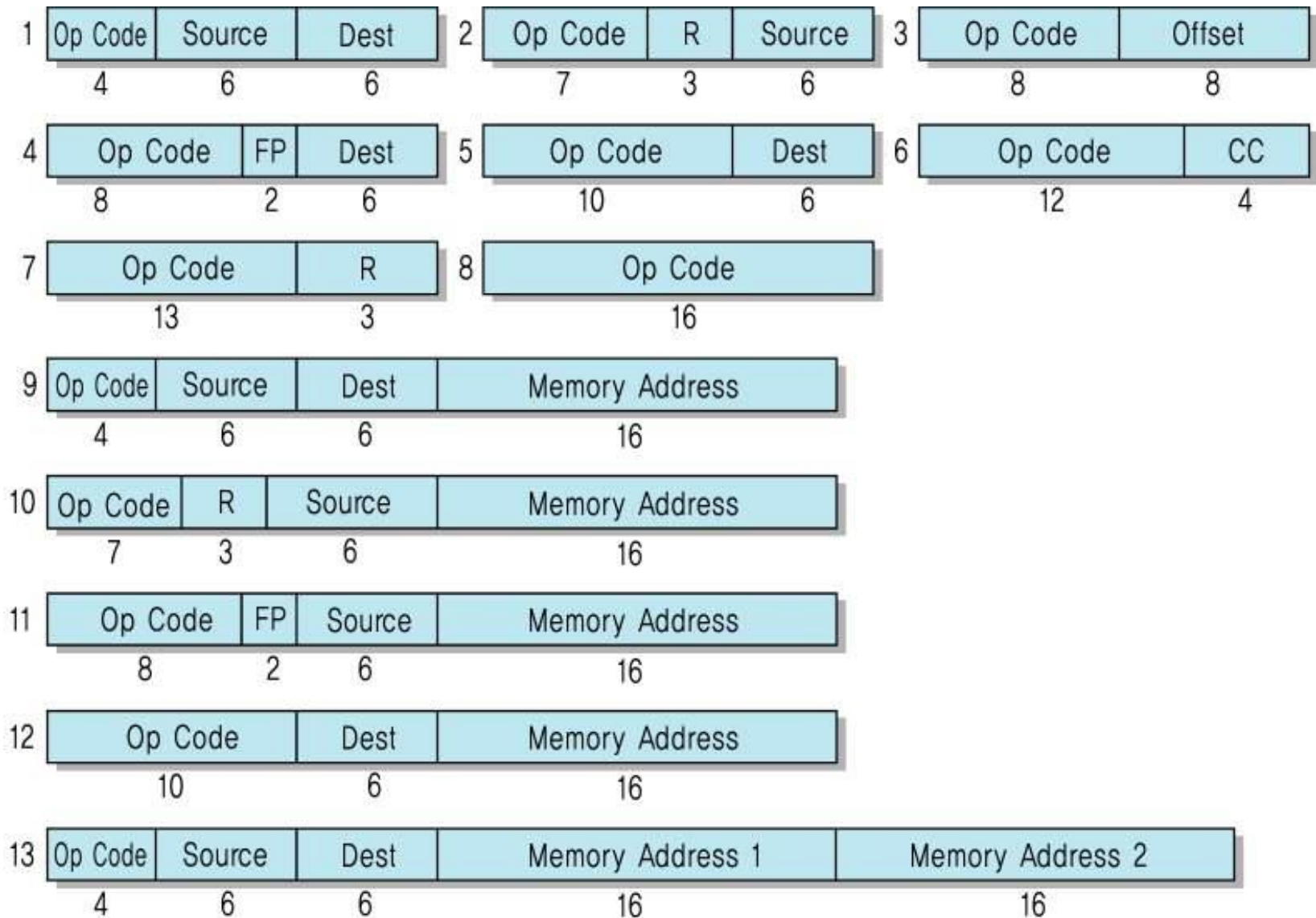
- 단어의 길이 = 36 비트, 명령어의 길이 = 36 비트
- 연산 코드 = 9 비트 → 최대 512 종류의 연산 허용 (실제 365 개)



□ PDP-11 프로세서 : 다양한 길이의 명령어 형식들 사용

- 연산 코드 = 4 ~ 16 비트
- 주소 개수 : 0, 1, 2 개

PDP-11의 명령어 형식들



펜티엄 계열 프로세서의 명령어 형식

- 선형 주소(linear address: **LA**) : 프로세서가 발생하는 주소
= 유효 주소 + 세그먼트의 시작 주소 (각 세그먼트의 시작 주소는 해당 Segment Register(SR)에 저장)

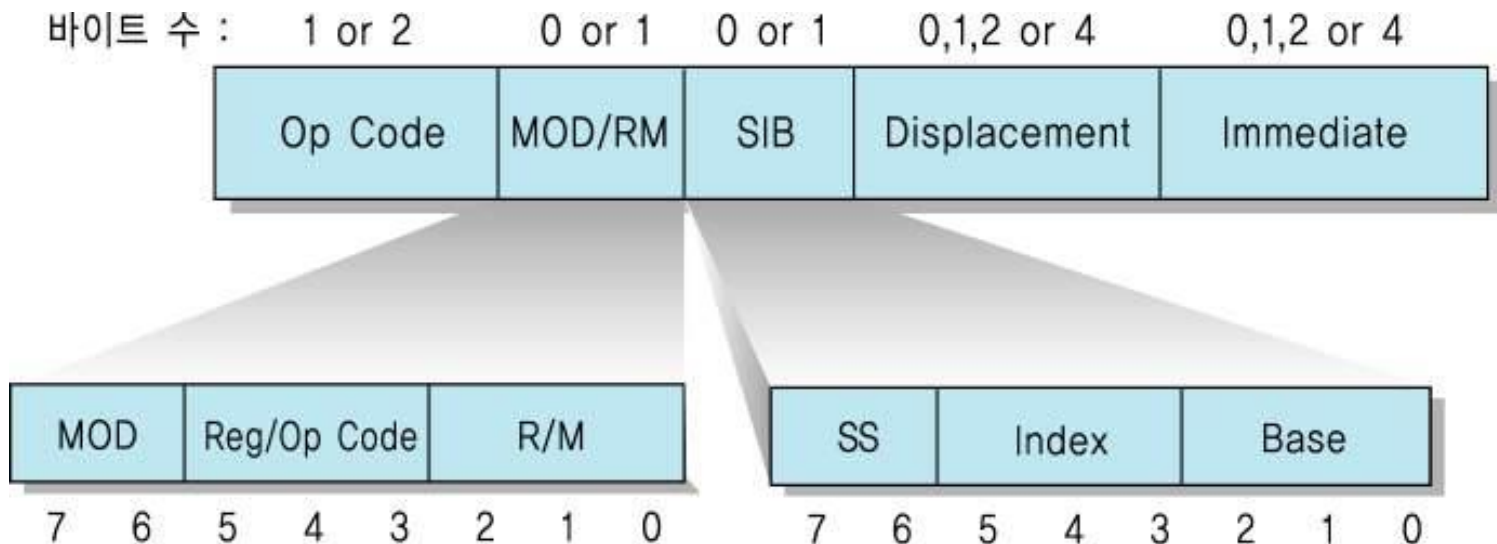
주소지정 방식	유효 주소(EA)	선형 주소(LA)
즉시 방식	데이터 = A	
레지스터 방식	$EA = R$	$LA = R$
변위 방식	$EA = A$	$LA = (SR) + EA$
베이스 방식	$EA = (BR)$	$LA = (SR) + EA$
변위를 가진 베이스 방식	$EA = (BR) + A$	$LA = (SR) + EA$
변위를 가진 인덱스 방식	$EA = (IX) + A$	$LA = (SR) + EA$
인덱스와 변위를 가진 베이스 방식	$EA = (IX) + (BR) + A$	$LA = (SR) + EA$
상대 방식	$EA = (PC) + A$	$LA = EA$

펜티엄 계열 프로세서의 명령어 형식(계속)

- 즉시 방식(immediate mode) : 데이터가 명령어에 포함되는 방식
 - 데이터의 길이 = 바이트, 단어(word) 혹은 2중 단어(double word)
- 레지스터 방식(register mode) : 유효 주소(EA)가 레지스터에 들어 있는 방식
- 변위 방식(displacement mode) : 명령어에 포함된 변위(주소)가 유효 주소로 사용되는 방식으로서, 직접 주소지정 방식에 해당
- 베이스 방식(base mode) : 레지스터 간접 주소지정에 해당
- 상대 방식(relative mode) : 변위값과 프로그램 카운터의 값을 더하여 다음 명령어의 주소로 사용하는 방식

펜티엄 계열 명령어 형식의 필드들

- ❑ 연산 코드(Op code) : 연산의 종류 지정. 길이 = 1 or 2 바이트
- ❑ MOD/RM : 주소지정 방식 지정
- ❑ SIB : MOD/RM 필드와 결합하여 주소지정 방식을 완성
- ❑ 변위(displacement) : 부호화된 정수(변위)
- ❑ 즉시(immediate) : 즉시 데이터



ATmega Microcontroller의 명령어 형식

- ATmega128 microcontroller의 CPU 코어
 - 8-비트 CPU
 - 대부분 명령어들의 길이 : 16비트
 - 기억장치 액세스 명령어들(LOAD, STORE) : 32비트
 - 내부 레지스터 : 32개
 - RISC 프로세서
 - 특징
 - 연산 코드의 비트 수가 명령어에 따라 달라짐
 - 오퍼랜드의 위치가 유동적

[명령어 형식의 예] 그림 2-35

- (a) ADD Rd,Rr ; $Rd \leftarrow Rd + Rs$
- (b) ASR Rd ; 레지스터 Rd에 대하여 산술적 우측 시프트 수행
- (c) JMP K ; K 번지로 무조건 점프
- (d) BRVS K ; V 플래그가 세트 되었다면, K 번지로 분기
- (f) LDS Rd,K ; K 번지의 내용을 읽어서 Rd에 적재

15															0
0	0	0	0	1	1	r	d	d	d	d	d	r	r	r	r

(a) ADD Rd,Rr 명령어

1	0	0	1	0	1	0	d	d	d	d	d	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(b) ASR Rd 명령어

1	0	0	1	0	1	0	k	k	k	k	k	1	1	0	k
k	k	k	k	k	k	k	k	k	k	k	k	k	k	k	k

(c) JMP K 명령어

1	1	1	1	0	0	k	k	k	k	k	k	k	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(d) BRVS K 명령어

1	0	0	1	0	0	0	d	d	d	d	d	0	0	0	0
k	k	k	k	k	k	k	k	k	k	k	k	k	k	k	k

(e) LDS Rd,K 명령어

□ ARM(Advanced RISC Machine) 계열 프로세서

- 32-비트 RISC 프로세서
- 개방형 아키텍처(Open Architecture)
- 모바일 시스템용 프로세서로 널리 사용
- 명령어 필드들
 - 분기조건 필드, 연산 필드, 오퍼랜드 필드, 등
 - 조건 플래그: N, Z, C, V
 - 레지스터 필드: Rn, Rd, Rs
 - P, U, W 비트: 주소지정 방식 결정
 - B 비트: 연산처리 단위 결정 (바이트(B=1) 혹은 단어(B=0))

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0 0 0			opcode		S	Rn				Rd				shift amount				shift	0	Rm							
cond				0 0 0			opcode		S	Rn				Rd				Rs		0	shift	1	Rm								
cond				0 0 1			opcode		S	Rn				Rd				rotate		immediate											
cond				0 1 0			P	U	B	W	L	Rn				Rd				immediate											
cond				0 1 1			P	U	B	W	L	Rn				Rd				shift amount				shift	0	Rm					
cond				1 0 0			P	U	S	W	L	Rn				register list															
cond				1 0 1			L	24-bit offset																							

그림 2-36 ARM 프로세서의 명령어 형식들

- * **Thumb 명령어**: 컴파일 과정에서 원래의 32-비트 명령어들을 분석하여 새로운 패턴의 16-비트 명령어 형식으로 축소
→ 프로그램 기억장치 용량 감소 (각종 임베디드 시스템에서 널리 사용)