컴퓨터 구조

제 10 강 중앙처리장치의 명령어

- 컴퓨터에서 마이크로 프로세서 유형보기
- 어셈블리 프로그램의 이해
- 인터럽트(interrupt)
- 명령어 세트
- 주소지정 방식

학습 목표

- 어셈블리 언어는 기계어와 일대일 대응을 하는 언어로 그 형식과 동작을 이해한다.
- 인터럽트의 동작과 인터럽트 부 사이클이 포함된 명령어 사이클을 공부한다.
- 명령어의 형식은 연산 코드와 오퍼랜드로 구성된다.
- 오퍼랜드 형태에 따라 0 ~ 3주소 명령어 프로그램이 존재한다.
- 명령어 세트에서 연산의 종류를 이해한다.
- 명령어 형식을 결정하는 명령어 길이와 명령어 종류의 수를 공부한다.
- 여러 가지 주소 지정방식을 이해한다.

마이크로 프로세서

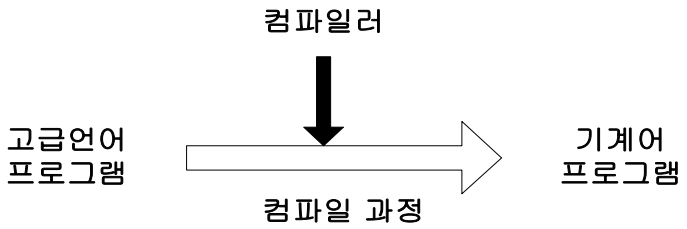

INTEL
PENTIUM4
650 SL7Z7 COSTA RICA
3.40GHZ/2M/800/04A

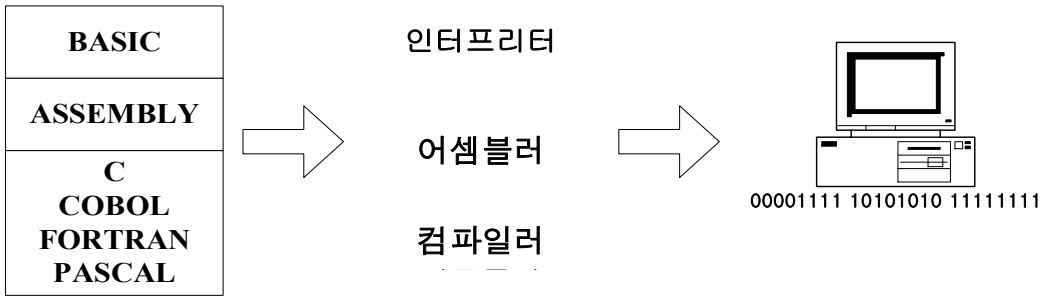

AMD
Opteron

어셈블리 프로그램의 이해

- 고급언어
  - C, COBOL, PASCAL, FORTRAN
- 저급언어
  - 어셈블리어
- 기계어

- 컴파일러
  - 고급 언어로 작성된 프로그램을 하드웨어가 인식할 수 있는 기계어로 변환





어셈블리 프로그램

- 고급언어

Para = 3

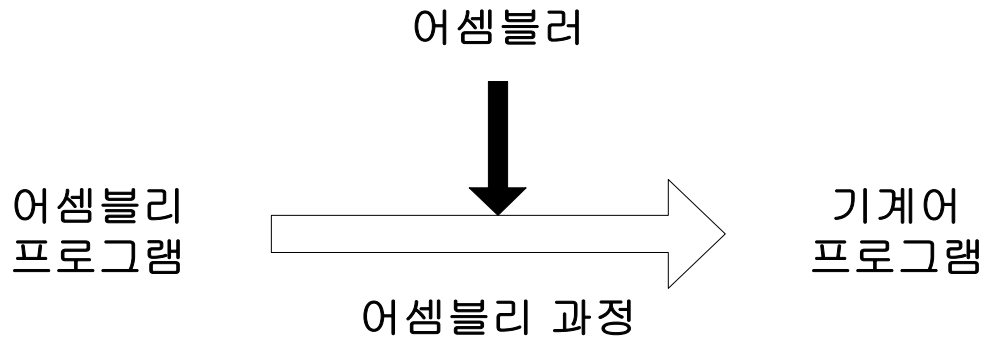
- 기계어

11000111 00000110 00000000 00000000 00000011 00000000

- 어셈블리어

MOV Para, 3

- 어셈블리 과정



8086 어셈블리 언어의 형식

DOSTART: ADD X ;X 와 가산기를 더하고 그 결과를 가산기에 저장

Label 부 Operation 부 Operand 부 Command 부

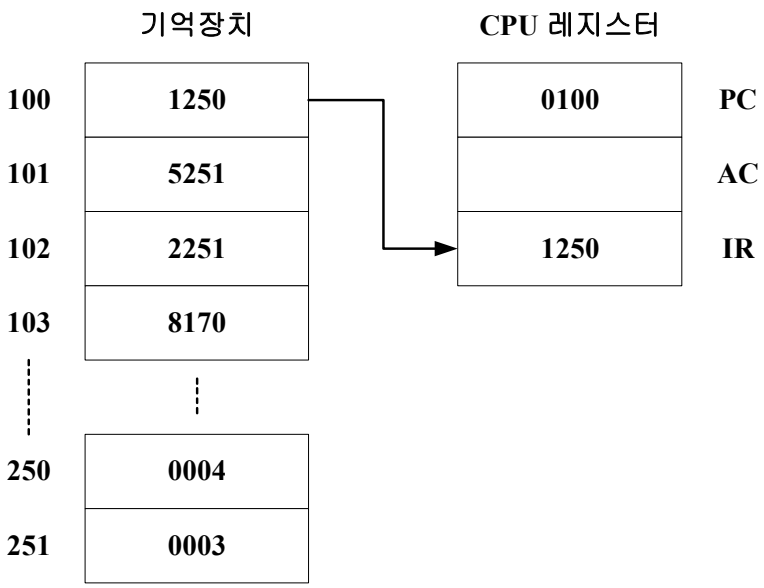
- 레이블
  - JUMP, LOOP와 같은 순환이나 반복 명령에서 해당 레이블로 프로그램 카운터 이동
- 연산
  - 명령의 니모닉 또는 어셈블러 디렉티브 등을 기록
- 오퍼랜드 또는 피연산자
  - 레지스터 이름, 정수, 라벨, 연산자, 주소 등을 기록
- 주석문
  - 프로그램의 설명

어셈블리 프로그램의 실행과정

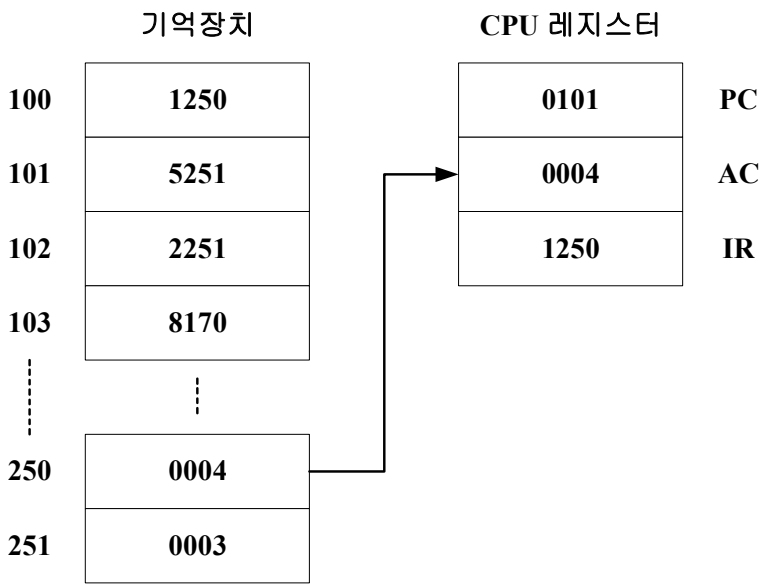
주소	명령어		기계 코드
100	LOAD	250	1250
101	ADD	251	5251
102	STA	251	2251
103	JUMP	170	8170

1. 기억장치 250번지에서 데이터를 가산기에 적재
2. 기억장치 251번지의 데이터와 덧셈을 수행
3. 결과를 다시 가산기에 저장
4. 기억장치 251번지에 그 결과를 저장
5. 프로그램의 주소 170번지로 점프

어셈블리 프로그램의 실행과정 – Load

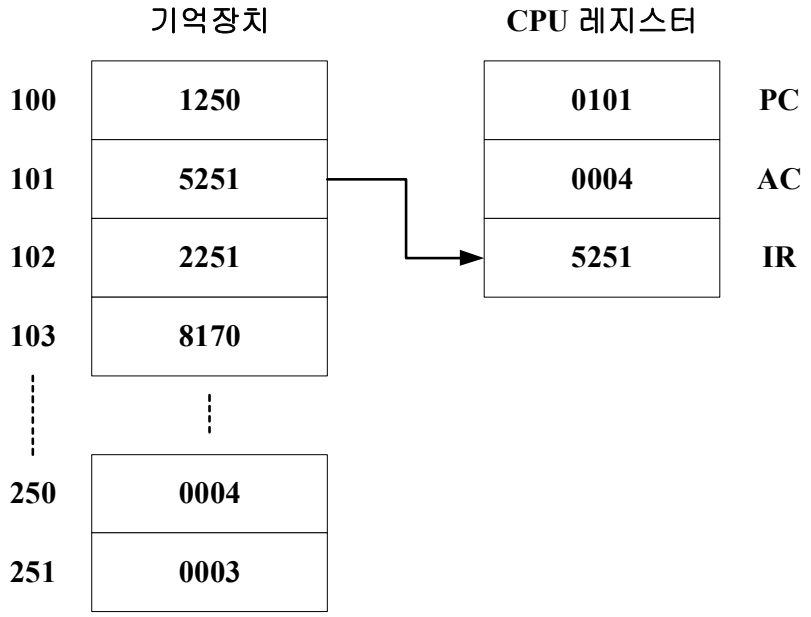


단계 1

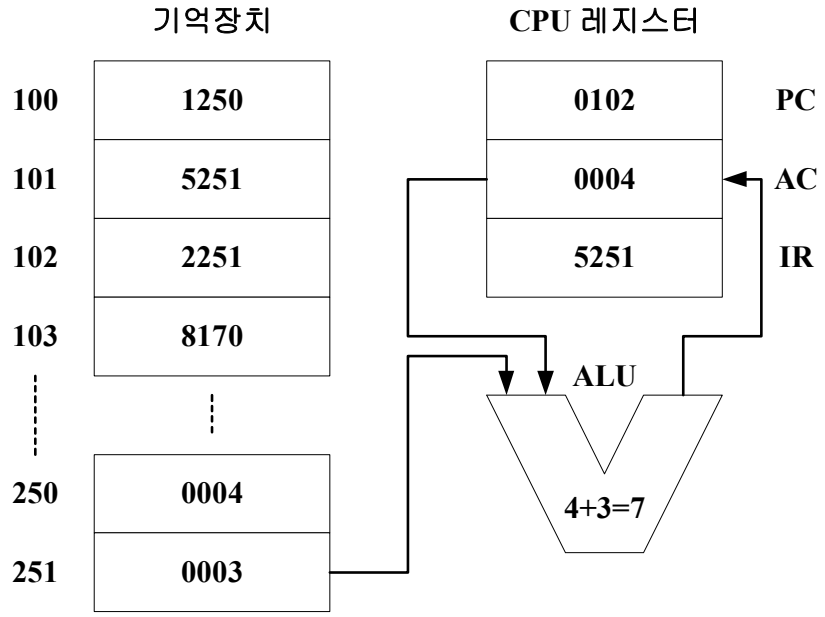


단계 2

어셈블리 프로그램의 실행과정 – ADD

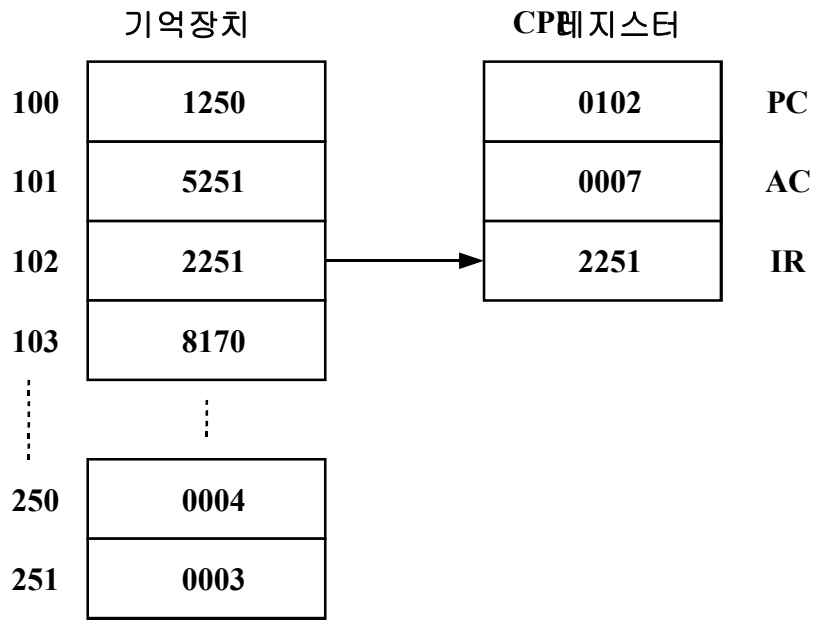


단계 3

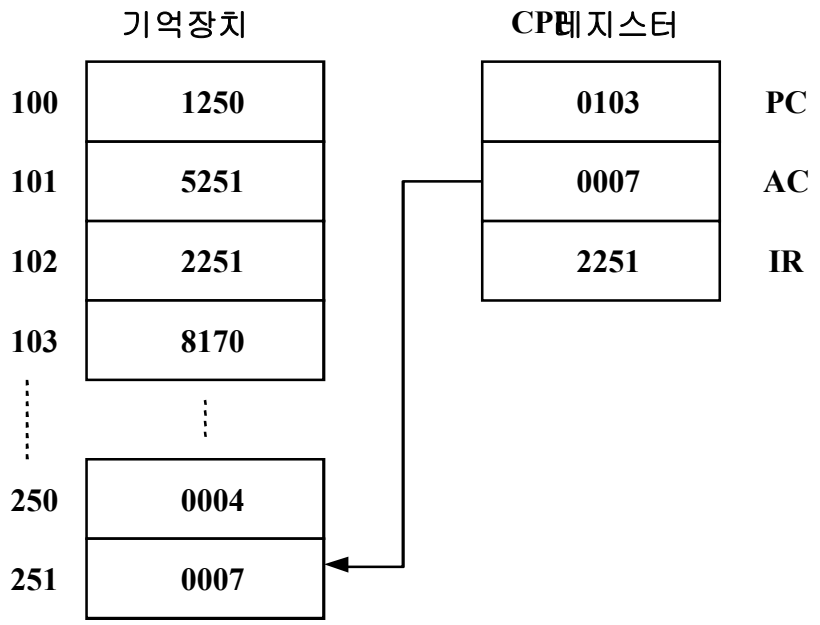


단계 4

어셈블리 프로그램의 실행과정 – STORE

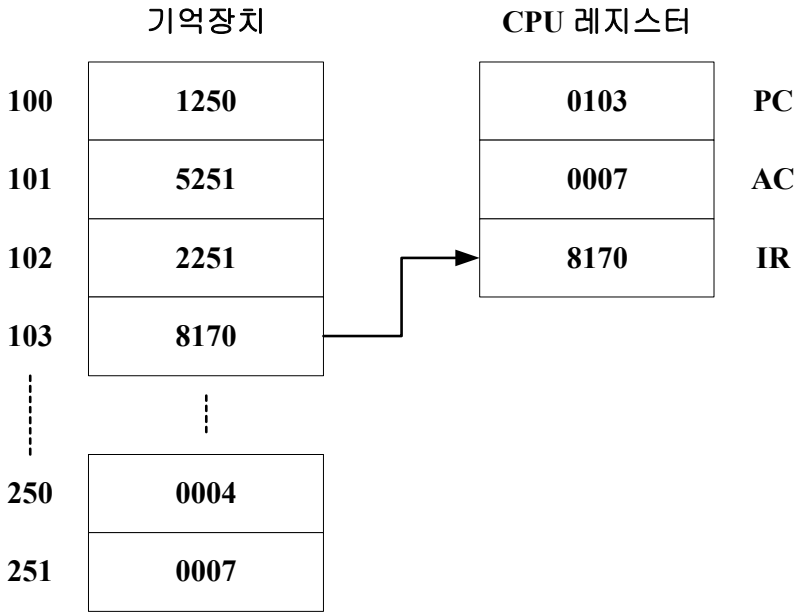


단계 5

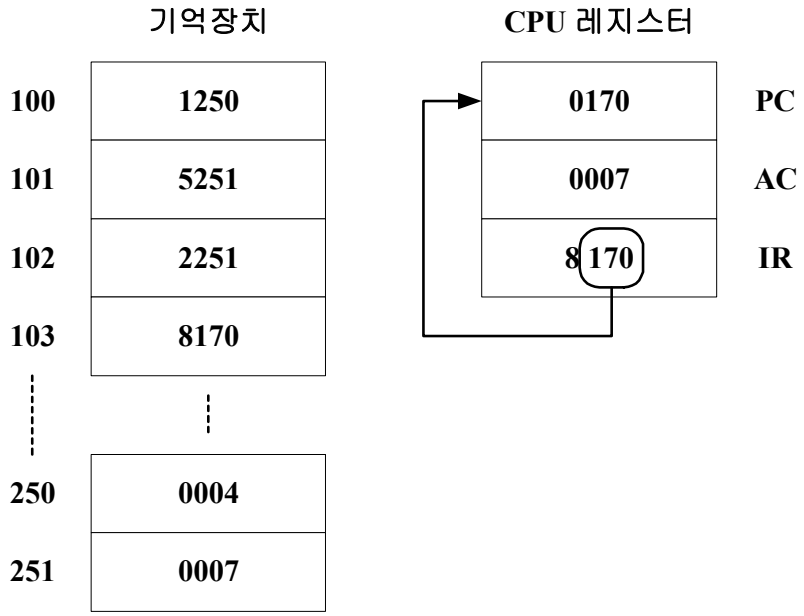


단계 6

어셈블리 프로그램의 실행과정 – JUMP



단계 7

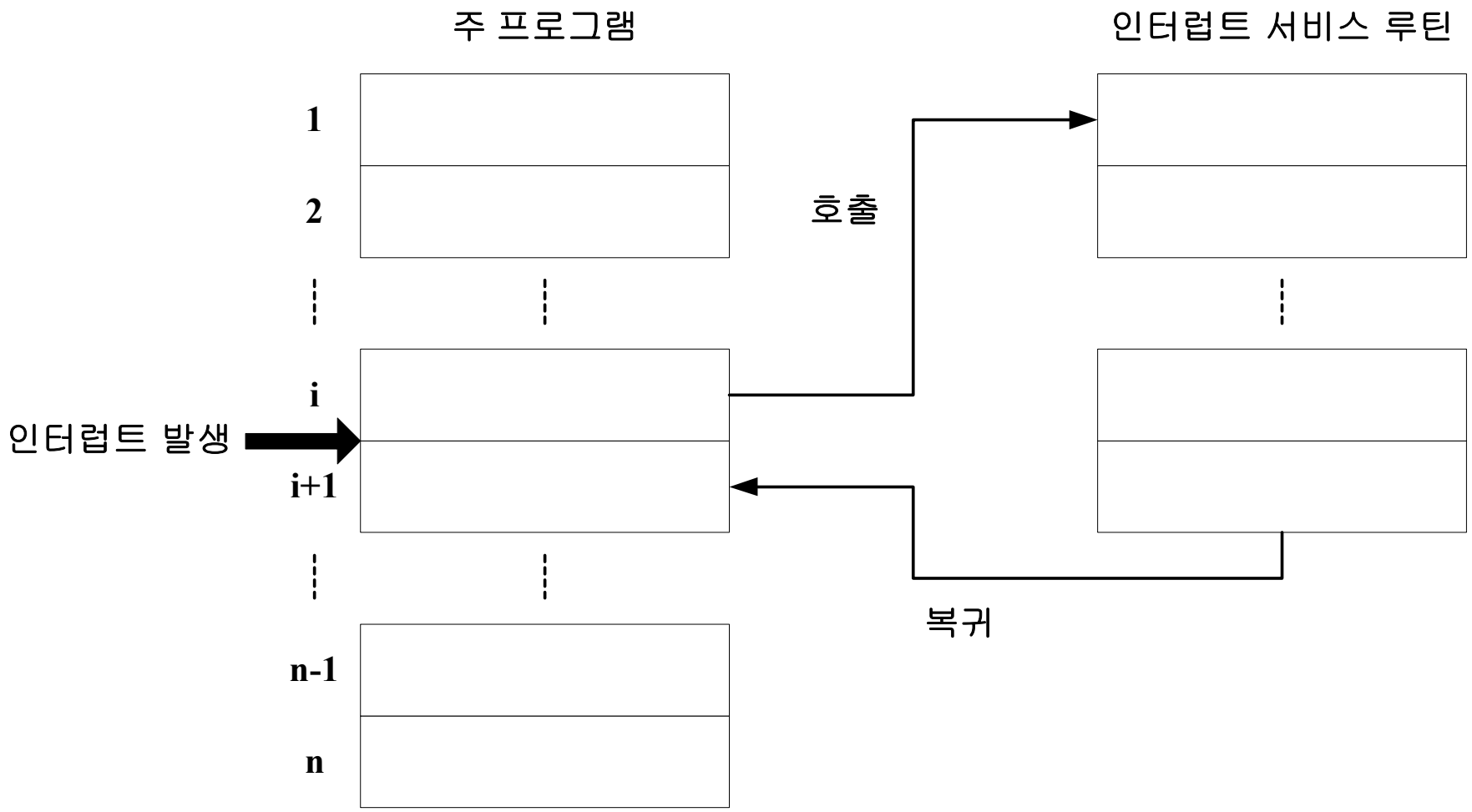


단계 8

- 인터럽트 사이클
- 인터럽트 사이클의 마이크로 연산
- 다중 인터럽트

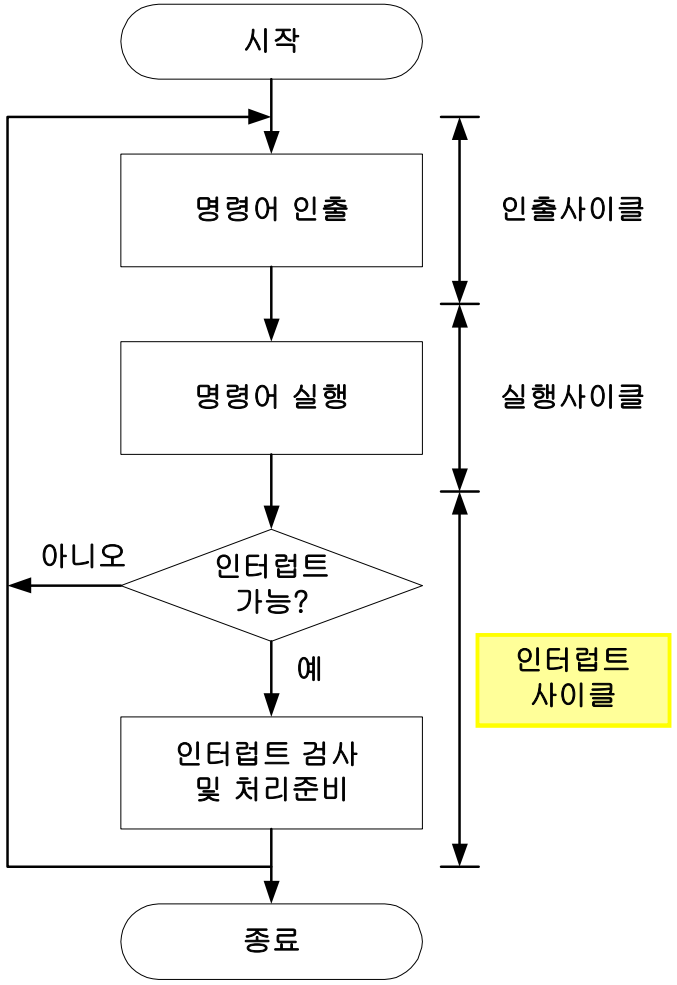
- 프로그램이 실행 중에 프로세서가 현재 처리 순서를 중단시키고 다른 동작을 수행하도록 하는 것
- 외부로부터 인터럽트 요구가 들어오면
  - 프로세서는 원래의 프로그램 수행을 중단
  - 요구된 인터럽트를 위한 서비스 프로그램을 먼저 수행
- 인터럽트 서비스 루틴(interrupt service routine : ISR)
  - 인터럽트를 처리하기 위해 수행하는 프로그램 루틴
- 결과적으로 처리 효율을 향상시키는 방법이다
- 인터럽트의 예
  - 오버플로우(overflow), '0'에 의한 나누기(division by zero) 등이 발생하면 프로그램이 종료

인터럽트에 의한 제어 이동



❒ 프로세서로 하여금 인터럽트 요구가 있는지를 검사하는 과정

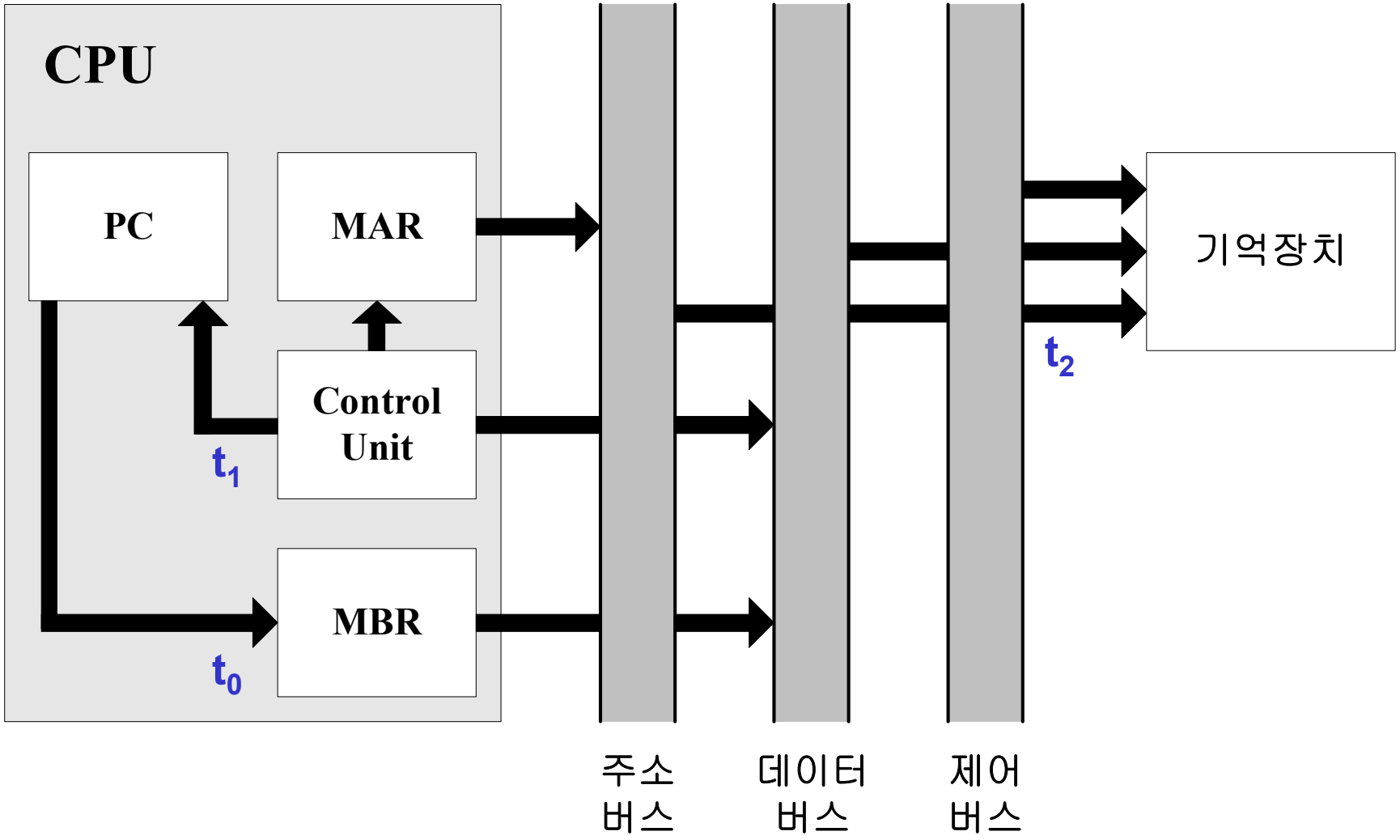
인터럽트 부 사이클이 포함된 명령어 사이클



t_0 : MBR ← PC
t_1 : MAR ← SP, PC ← ISR 시작주소
t_2 : M[MAR] ← MBR

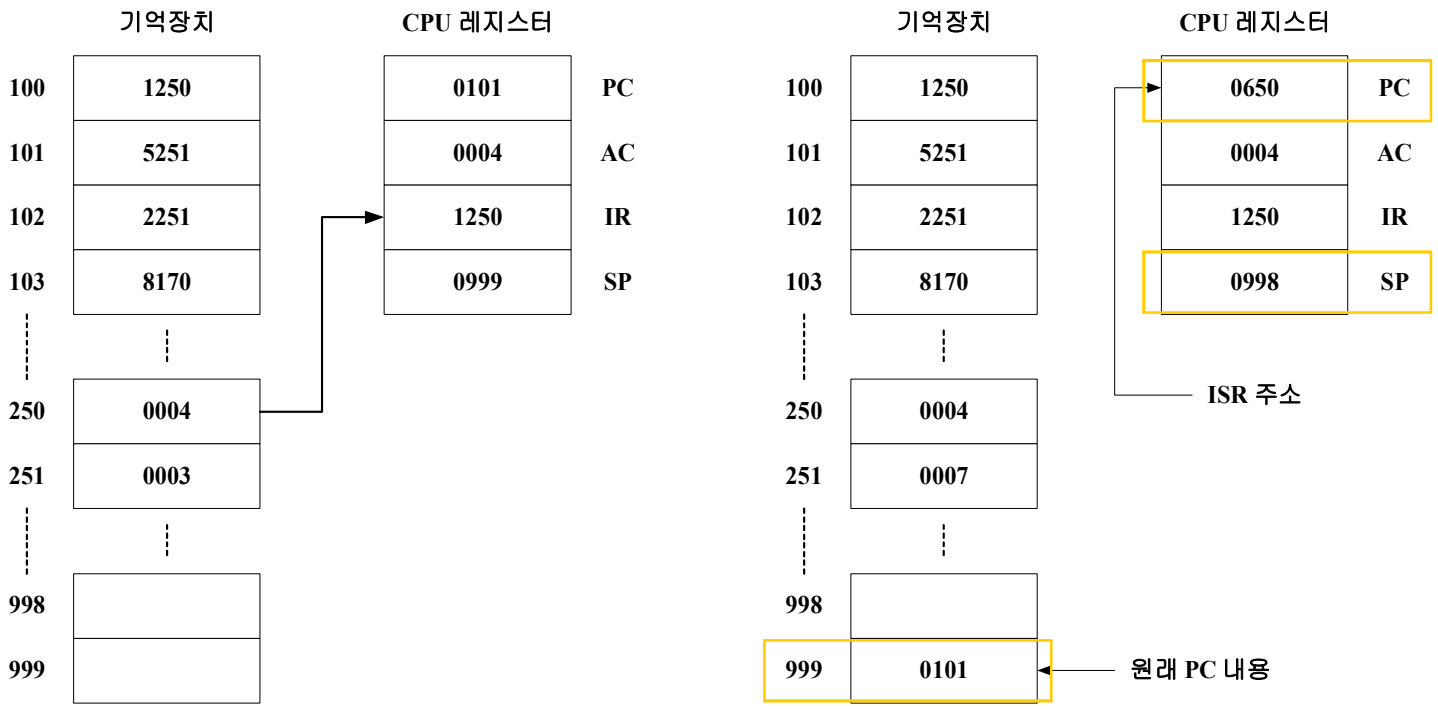
- 세 개의 CPU 클럭으로 구성
- 첫 번째 클럭에서는 PC의 내용이 MBR로 전송
  - PC 내용은 주 프로그램에서 수행될 다음 명령어 주소
- 두 번째 클럭에서는 SP(스택 포인터)의 내용이 MAR로 전송
- PC의 내용은 인터럽트 서비스 루틴의 시작 주소로 변경
  - 스택 포인터는 MBR에 저장되어 있는 내용을 스택에 저장하기 위해서 저장할 위치를 지정하기 위해서 사용
- 세 번째 클럭에서는 MBR에 저장되어 있던 원래 PC의 내용이 스택에 저장

인터럽트 부사이클에서의 데이터 흐름



인터럽트가 발생한 경우의 마이크로 연산

```
100   LOAD 250
101   ADD  251
102   STA  251
103   JUMP 170
```



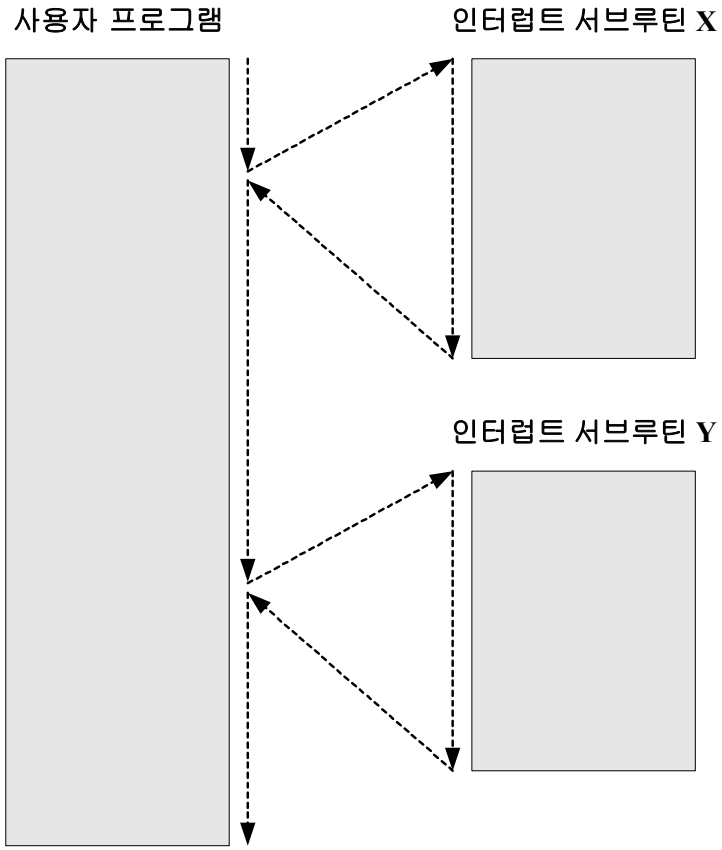
(a) LOAD 명령어의 실행 사이클이 종료된 상태

(b) 인터럽트 사이클이 종료된 상태

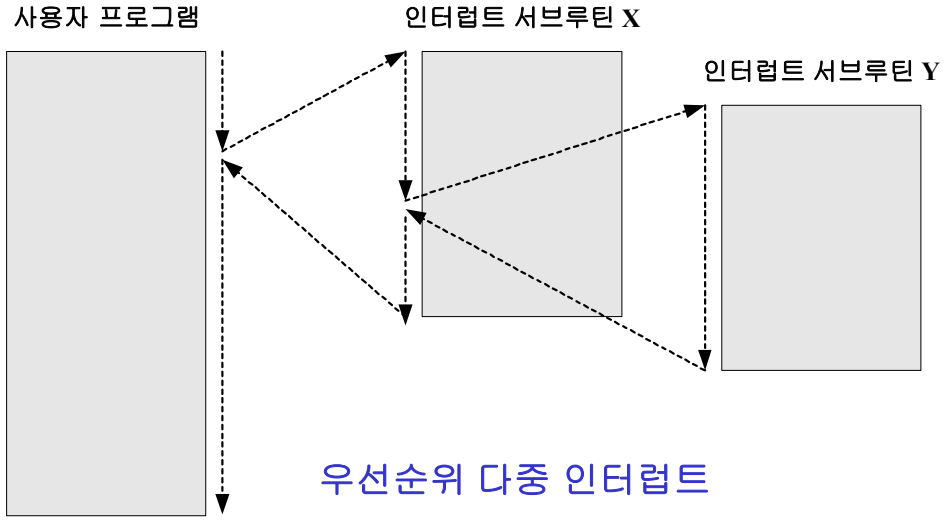
다중 인터럽트

- 인터럽트 서비스 루틴 수행 중 다른 인터럽트가 발생
- 인터럽트 불가능(interrupt Disabled)
  - CPU가 인터럽트 서비스 루틴을 처리하고 있는 도중에는 새로운 인터럽트 요구가 들어오더라도 CPU가 인터럽트 사이클을 수행하지 않도록 방지
  - 인터럽트는 대기하게 되며, 현재의 인터럽트에 대한 처리가 종료된 후에 발생한 순서대로 처리
- 우선 인터럽트(Priority Interrupt)
  - 인터럽트의 우선 순위를 결정
  - 우선 순위가 낮은 인터럽트가 처리되고 있는 동안에 우선순위가 더 높은 인터럽트가 들어오면 현재의 인터럽트 서비스 루틴의 수행을 중단하고 새로운 인터럽트를 처리

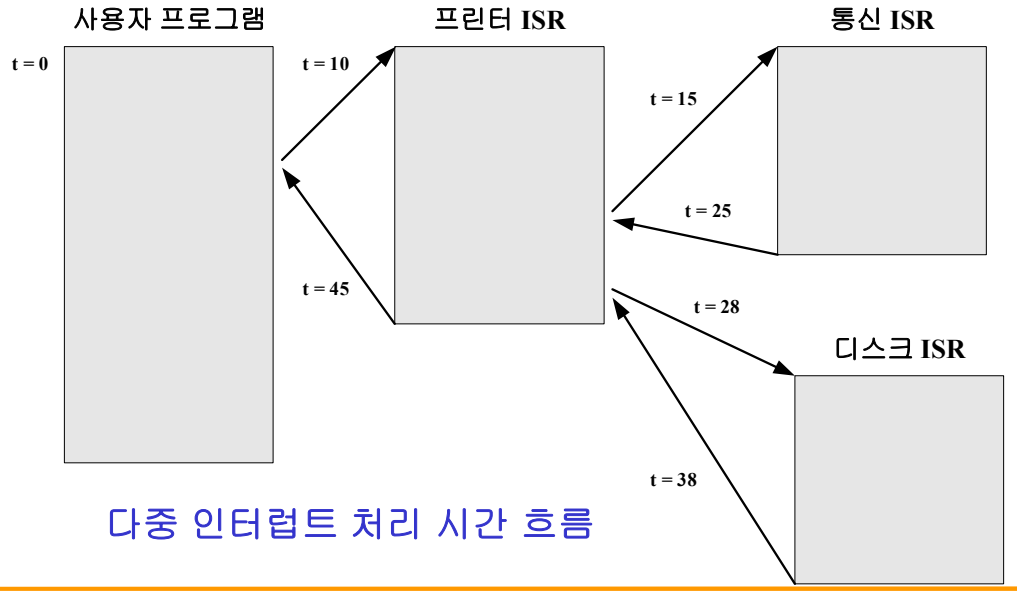
다중 인터럽트



순차적인 다중 인터럽트



우선순위 다중 인터럽트



다중 인터럽트 처리 시간 흐름

명령어 세트

- 명령어 세트의 특징
- 오퍼랜드 형태와 수에 따른 명령어 분류
- 명령어 형식이 프로그래밍에 미치는 영향
- 명령어 세트에서 연산의 종류
- 명령어 형식

명령어 세트 (instruction set)

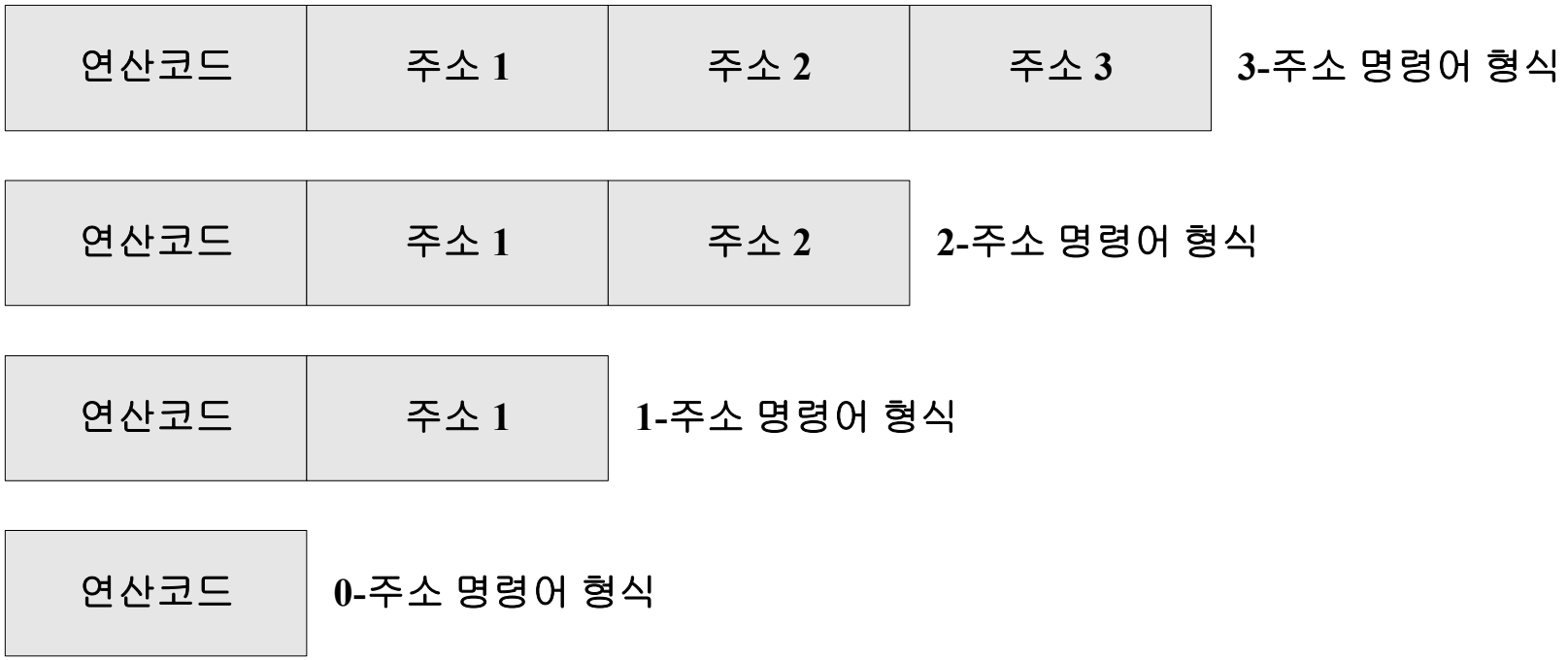
- CPU가 수행할 동작을 정의하는 2진수 코드들의 집합 또는 명령어들의 집합
- 기계 명령어(machine instruction)라고도 함
- 일반적으로 어셈블리 코드(assembly code) 형태로 표현
- CPU의 사용목적, 특성에 따라 결정
- 명령어 세트 설계를 위해 결정되어야 할 사항들
  - CPU가 수행할 연산들의 수와 종류 및 복잡도 등을 결정
  - 데이터 형태
  - 주소지정 방법

명령어 세트의 특징

- 명령어의 구성
  - 연산 코드(Operation Code)
  - 오퍼랜드(Operand)
- 연산 코드는 수행될 연산을 지정(예: LOAD, ADD 등)
- 오퍼랜드(Operand)는 연산을 수행하는 데 필요한 데이터 혹은 데이터의 주소
- 각 연산은 한 개 혹은 두 개의 입력 오퍼랜드들과 한 개의 결과 오퍼랜드를 포함
- 명령어 분류
  - 데이터 처리
  - 데이터 저장
  - 데이터 이동
  - 제어

명령어 형식

❒ 오퍼랜드가 주소를 나타내는 경우의 명령어 분류



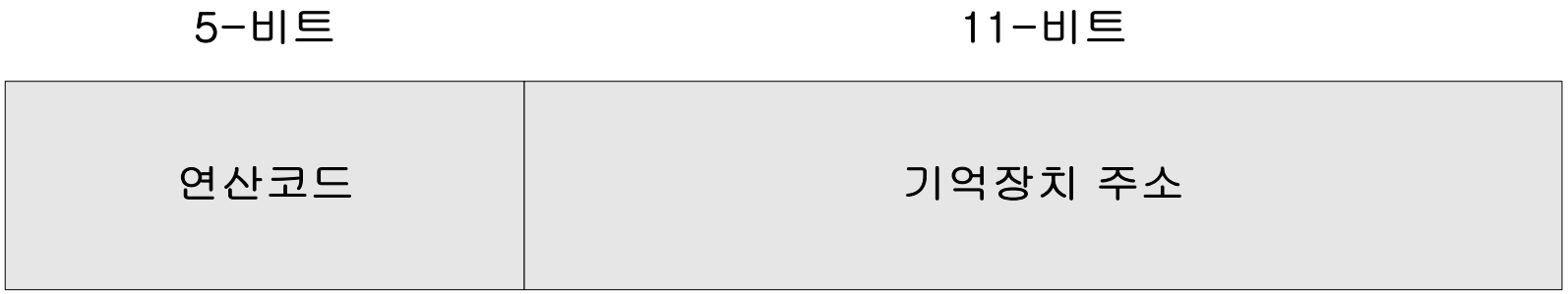
1-주소 명령어(1-Address instruction)

- 오퍼랜드를 한 개만 포함하는 명령어

- [예]

LOAD X ; AC ← M[X]

X 주소 하나만 존재



- 2개의 오퍼랜드를 포함하는 명령어

- [예]

```
MOV  X, Y          ; M[X] ← M[Y]
```

5비트	3비트	3비트	5비트
연산코드	레지스터 1	레지스터 2	(사용안됨)

(a) 두개의 레지스터 오퍼랜드들을 가지는 경우

5비트	3비트	8비트
연산코드	레지스터	오퍼랜드 2

(b) 한 오퍼랜드는 기억장치 주소인 경우

3-주소 명령어(3-Address instruction)

- 3개의 오퍼랜드를 포함하는 명령어

- [예]

ADD X, Y, Z ; M[X] ← M[Y] + M[Z]

4-비트	4-비트	4-비트	4-비트
연산코드	레지스터 1	레지스터 2	레지스터 3

❒어셈블리 명령어

명령어	동작
ADD	덧셈
SUB	뺄셈
MUL	곱셈
DIV	나눗셈
MOV	데이터 이동
LOAD	기억장치로부터 데이터 적재
STOR	기억장치로 데이터 저장

1-주소 명령어를 사용한 프로그램

- X = B×(C+D×E-F/G)

```
100    LOAD F          ; AC ← M[F]
101    DIV G           ; AC ← AC / M[G]
102    STOR T          ; M[T] ← AC
103    LOAD D          ; AC ← M[D]
104    MUL E           ; AC ← AC * M[E]
105    ADD C           ; AC ← AC + M[C]
106    SUB T           ; AC ← AC - M[T]
107    MUL B           ; AC ← AC / M[B]
108    STOR X          ; M[X] ← AC
```

프로그램 길이 : 9

2-주소 명령어를 사용한 프로그램

- X = B×(C+D×E-F/G)

```
100    MOV R1, D          ; M[R1] ← M[D]
101    MUL R1, E          ; M[R1] ← M[R1] * M[E]
102    MOV R2, F          ; M[R2] ← M[F]
103    DIV R2, G          ; M[R2] ← M[R2] / M[G]
104    SUB R1, R2         ; M[R1] ← M[R1] – M[R2]
105    ADD R1, C          ; M[R1] ← M[R1] + M[C]
106    MUL R1, B          ; M[R1] ← M[R1] * M[B]
107    MOV X, R1          ; M[X] ← M[R1]
```

프로그램 길이 : 8

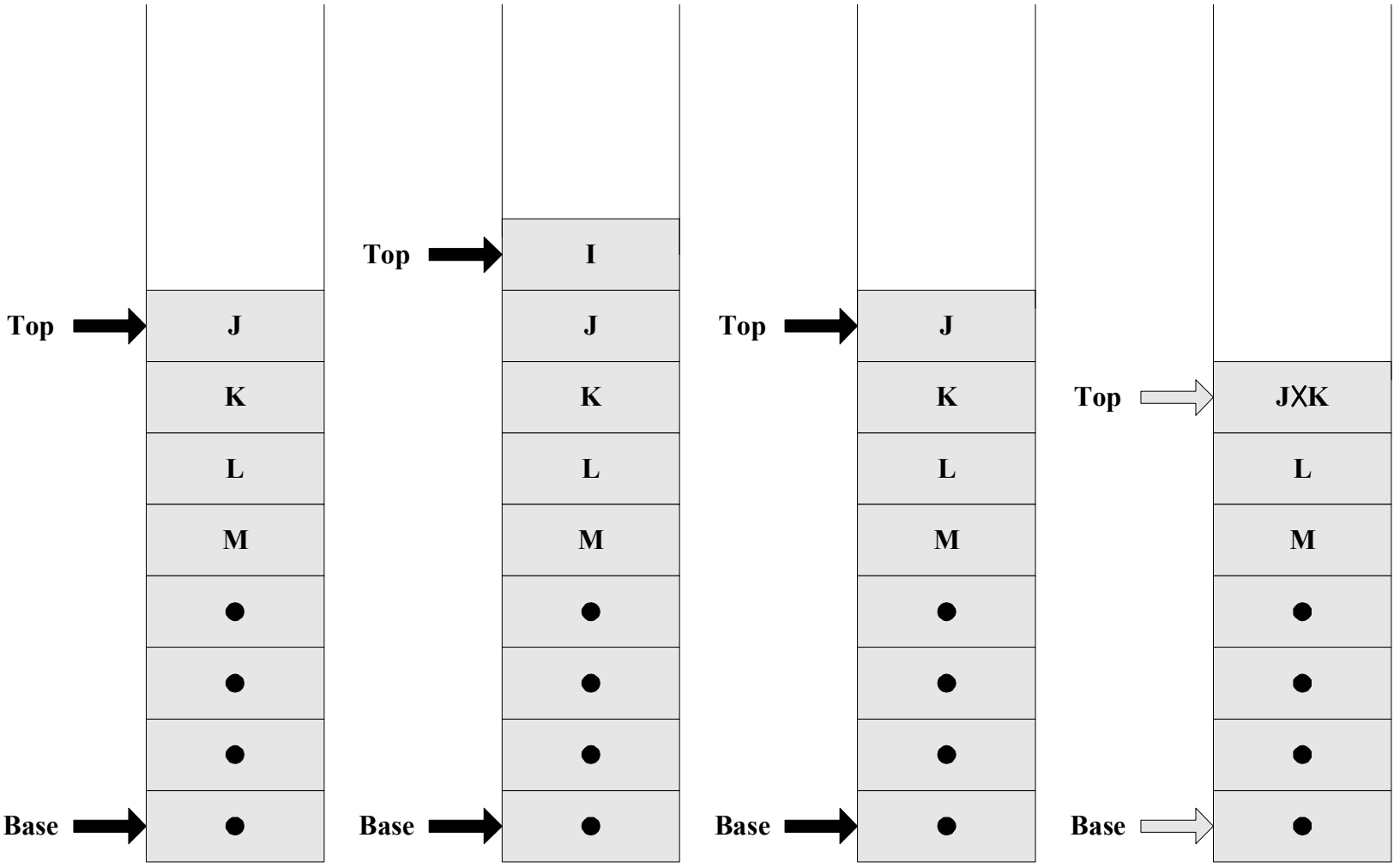
3-주소 명령어를 사용한 프로그램

- X = B×(C+D×E-F/G)

```
100    MUL D, E, R1          ; M[R1] ← M[D] * M[E]
101    ADD C, R1, R1         ; M[R1] ← M[C] + M[R1]
102    DIV F, G, R2          ; M[R2] ← M[F] / M[G]
103    SUB R1, R2, R1        ; M[R1] ← M[R1] - M[R2]
104    MUL B, R1, X          ; M[X] ← M[B] * M[R1]
```

프로그램 길이 : 5

스택(Stack)의 기본동작 – PUSH, POP (LIFO : Last In First Out)



(a) 시작

(b) I를 PUSH한 후 스택의 상태

(c) I를 POP 한 후 스택의 상태

(d) J와 K의 곱셈한 후 스택의 상태

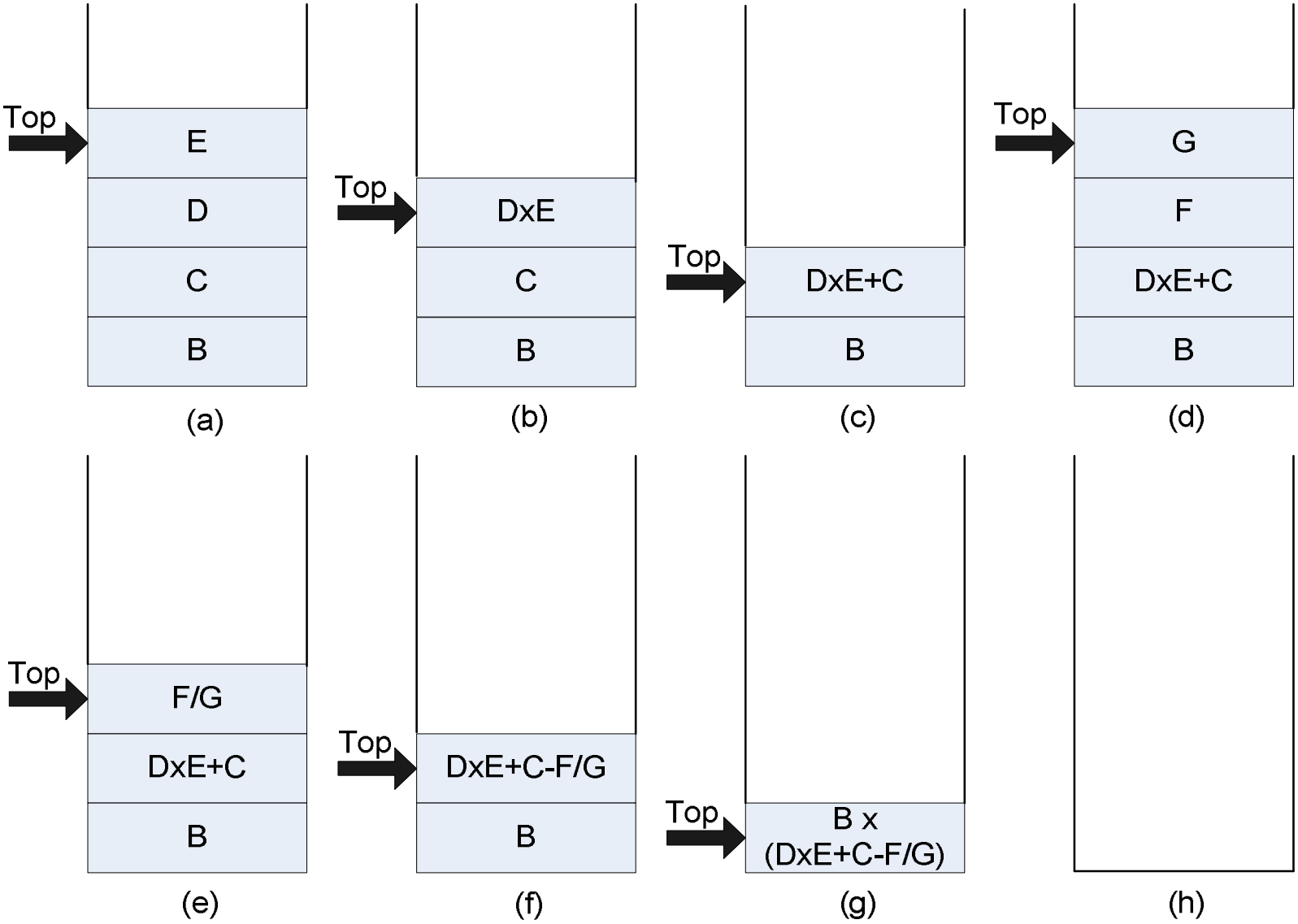
0-주소 명령어를 사용한 프로그램

- X = B×(C+D×E-F/G)

```
100     PUSH B      ; 스택에 B가 입력됨
101     PUSH C      ; 스택에 C가 입력됨
102     PUSH D      ; 스택에 D가 입력됨
103     PUSH E      ; 스택에 E가 입력됨
104     MUL         ; E와 D를 연속해서POP, 곱셈을 수행 후 결과 PUSH
105     ADD         ; E*D의 결과와 C를 연속해서 POP, 덧셈 후 결과 PUSH
106     PUSH F      ; 스택에 F가 입력됨
107     PUSH G      ; 스택에 G가 입력됨
108     DIV         ; G와 F를 연속해서 POP, 나눗셈을 수행한 후 결과 PUSH
109     SUB         ; F/G와 C+E*D를 연속해서 POP, 뺄셈을 수행 후 결과
                      PUSH
110     MUL         ; (C+D*E-F/G)와 B를 연속해서 POP, 곱셈을 수행 후
                      결과 PUSH
111     POP         ; 기억장치 X 번지에 저장하기 위해 결과를 POP
```

프로그램 길이 : 12

스택에서 0-주소 명령어 프로그램의 동작



명령어 세트에서 연산의 종류

- 데이터 전송
- 산술 연산, 논리 연산, 변환
- 입출력 명령어, 프로그램 제어 이동 명령어

데이터 전송

- 레지스터와 레지스터 사이에 데이터를 이동
- 레지스터와 기억장치 사이에 데이터를 이동
- 기억장치와 기억장치 사이에 데이터를 이동
- 데이터 전송 명령어에는 근원지 오퍼랜드(source operand)와 목적지 오퍼랜드 (destination operand)의 위치가 명시
- 전송될 데이터의 길이와 오퍼랜드의 주소지정 방식(addressing mode)등이 명시
- [예] CPU에서 읽기 동작
  - 주소지정 방식에 근거하여 기억장치 주소를 계산, 실제 주소 획득
  - 원하는 데이터가 캐시에 있는 지 검사
  - 캐시 히트상태에서는 원하는 데이터를 캐시로부터 얻어서 CPU로 이동
  - 캐시 미스상태의 경우, 기억장치 모듈로 읽기 명령을 전송하고 기억장치로부터 데이터가 CPU로 전송

- 산술 연산
  - 기본산술 연산
    - 덧셈, 뺄셈, 곱셈, 나눗셈
  - 특징적인 산술 연산
    - 단일-오퍼랜드 연산
      - 절대값(absolute) 연산
      - 음수화(negate)연산
      - 증가(increment) 연산
      - 감소(decrement) 연산
- 논리 연산
  - 비트들 간에 대한 AND, OR, NOT 및 exclusive-OR 연산
- 변환(Conversion)
  - 2진수 → 10진수
  - EBCDIC 코드 → ASCII 코드

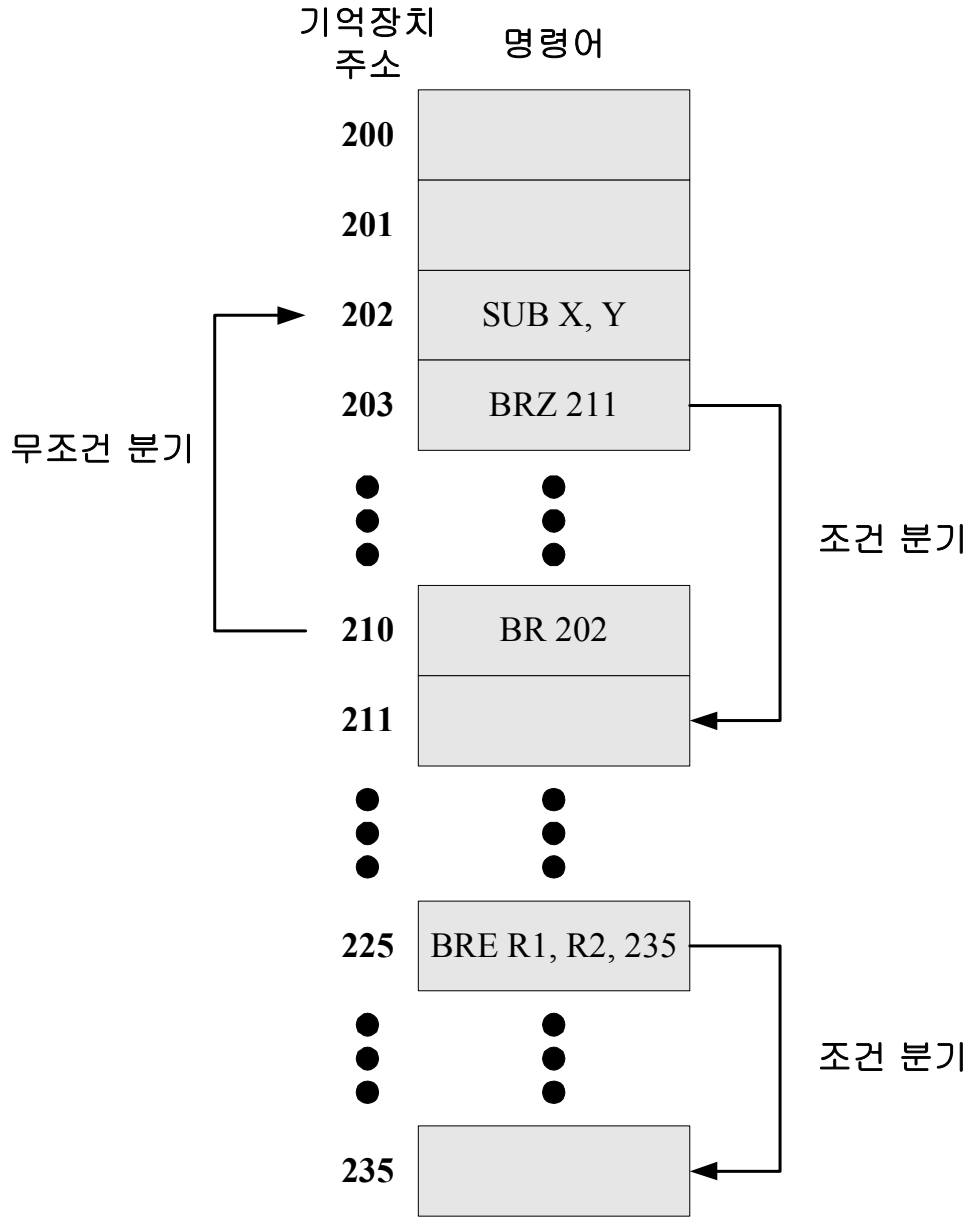
- 입출력 명령어
  - CPU와 외부 장치들간의 데이터 이동을 위한 명령어
  - 분리형 I/O
    - 특정 I/O 명령어 사용
  - 기억장치-사상 I/O
    - 일반적이 데이터 이동 명령어

- 프로그램 제어 이동 명령어
  - 명령어 실행 순서를 변경하는 명령어
  - 종류
    - 분기 명령어
    - 서브루틴 호출 명령어

분기 명령어

- 오퍼랜드가 다음 실행될 명령어의 주소를 가지고 있음
- 명령어 내용에 따라서 무조건 오퍼랜드의 주소로 이동하거나 조건 만족 시에만 이동하는 형태
- 조건 분기에서 연산 결과를 나타내는 조건 코드 (condition code)
  - zero(0)
  - 부호(+, –)
  - 오버플로우 플래그

다양한 분기 형태


기억장치 주소
명령어
200
201
202
SUB X, Y
203
BRZ 211
무조건 분기
조건 분기
210
BR 202
211
225
BRE R1, R2, 235
조건 분기
235

서브루틴 호출 명령어

- 호출 명령어(CALL 명령어)는 현재의 PC 내용을 스택에 저장하고 서브루틴의 시작 주소로 분기하는 명령어
- 복귀 명령어(RET 명령어) CPU가 원래 실행하던 프로그램으로 되돌아가도록 하는 명령어

t_0 : MBR ← PC
t_1 : MAR ← SP, PC ← X
t_2 : M[MAR] ← MBR, SP ← SP – 1

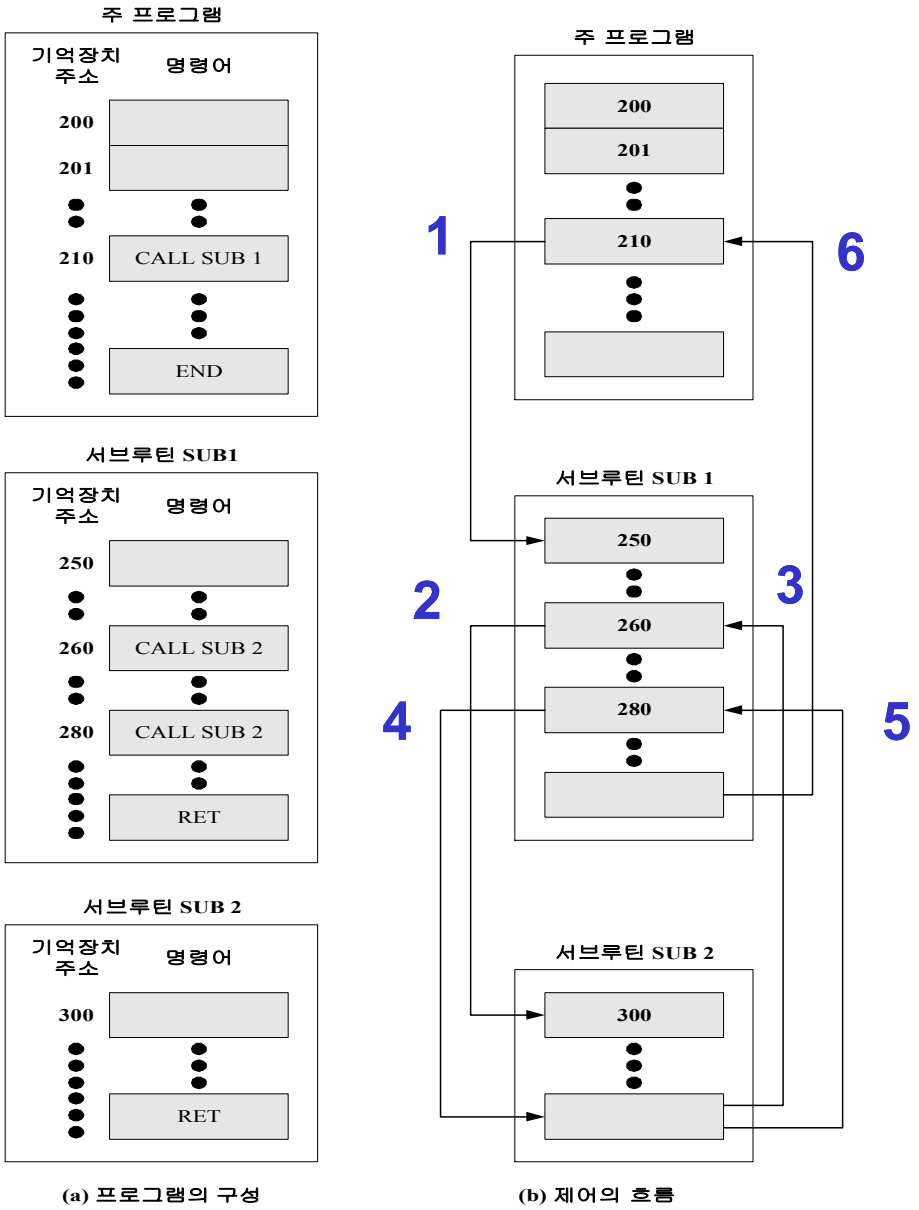
- t_0에서는 PC의 저장된 다음 명령어 주소가 메모리 버퍼 레지스터(MBR)에 저장
  - 서브루틴 수행 완료 후에 복귀할 주소가 저장
- t_1에서는 스택 포인터(SP)가 메모리 주소 레지스터(MAR)에 저장
- PC에는 실행 될 서브루틴의 시작 주소가 저장
- t_2에서는 MBR에 저장되어 있는 복귀 주소가 스택 포인터가 가리키는 스택의 위치에 저장
- 스택 포인터는 스택의 top값을 하나 감소

- 서브루틴을 수행

t_0 : SP ← SP + 1
t_1 : MAR ← SP
t_2 : PC ← M[MAR]

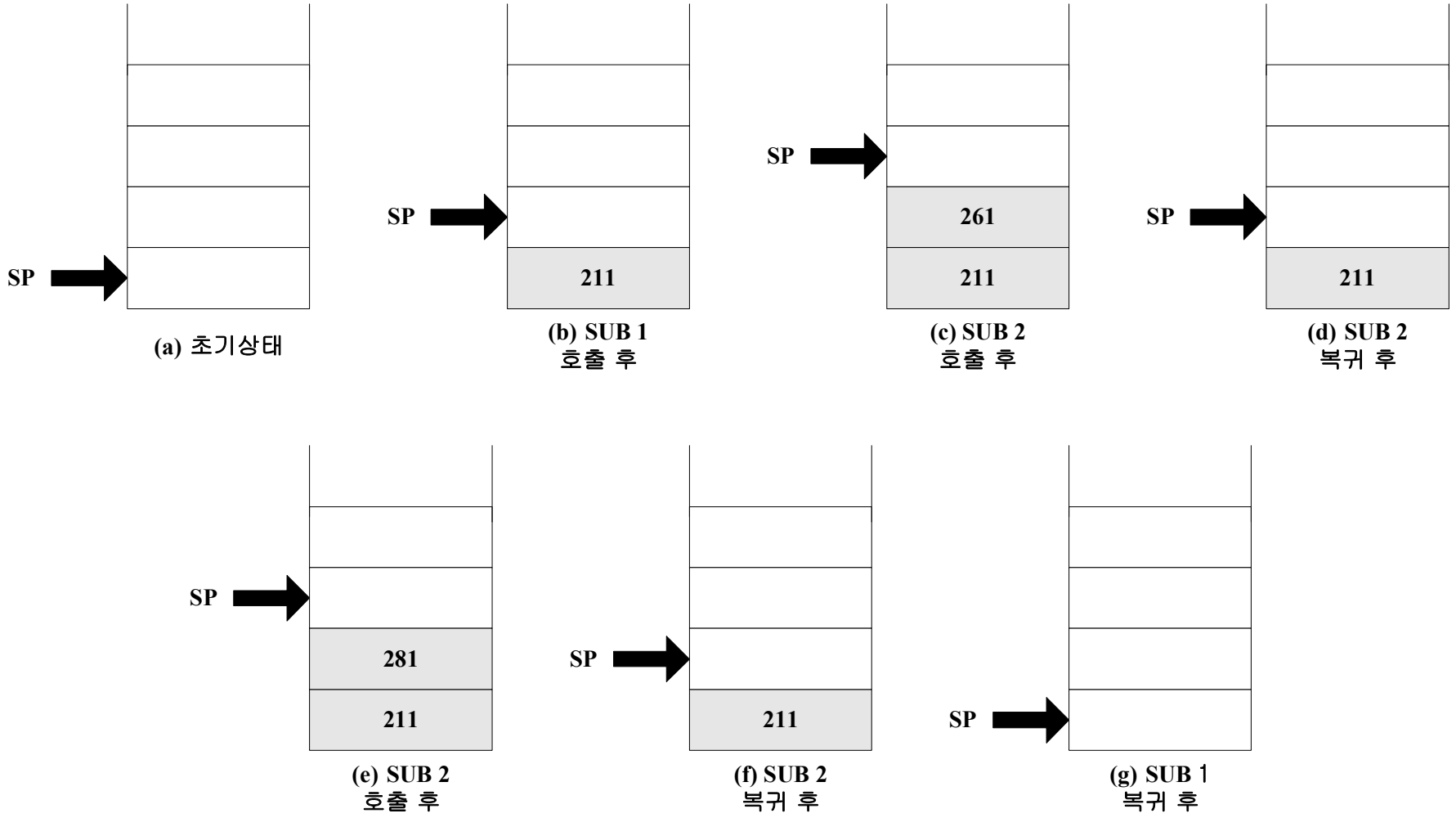
- t_0에서는 스택에 저장에 되어 있는 복귀주소를 POP하기 위해서 스택 포인터를 하나 증가
- t_1에서는 스택 포인터를 메모리 주소 레지스터(MAR)에 저장
- t_2에서는 스택에 저장되어 있는 복귀주소를 POP해서 PC에 저장
- PC에 의해서 원래의 프로그램으로 복귀

서브루틴의 호출과 복귀 과정



(a) 프로그램의 구성 (b) 제어의 흐름

서브루틴 수행 과정에서 스택의 변화



- 명령어 내의 비트 배열
- 명령어 세트에는 한가지 이상의 명령어 형식이 존재
- 연산코드의 비트 길이, 오퍼랜드의 수와 길이에 따라 명령어 형식이 달라질 수 있다.

명령어 길이에 따른 명령어 형식

- 기억장치와 관련된 명령어 길이
  - 기억장치 용량과 기억장치 조직에 의해서 주소를 지정하는 오퍼랜드 부분이 영향
    - 기억장치의 용량이 많은 경우는 주소의 수가 많아지므로 오퍼랜드의 비트수가 많아 져야 한다
- 버스 조직(Bus structure)에 의한 명령어 길이
  - 데이터를 전송하는 명령어의 경우 이에 맞는 명령어의 길이가 필요
- CPU의 복잡도(complexity)와 CPU의 속도에 의한 명령어 길이
  - 명령어는 CPU가 한 번에 읽고 쓸 수 있는 단위로 수행
  - CPU가 한 번에 읽고 쓸 수 있는 비트 수를 단어(Word)
  - 단어의 크기에 따라서 명령어의 길이 결정

명령어의 종류에 따른 명령어 형식

- 연산 코드의 종류와 오퍼랜드가 커지면 프로그램에 유리
- 주기억 장치의 용량이 증가, 가상 기억장치의 사용량이 증가하면 더 큰 기억장치 영역들을 주소 지정 가능
- 연산 코드, 오퍼랜드, 주소지정 방식, 주소 범위는 비트들을 필요로 하므로 명령어가 더 길어진다
- 긴 명령어는 비 효율적으로 사용될 가능성 존재
- 명령어 종류의 수와 비트 수에 대한 적절한 조정 필요
- 명령어 내 비트들의 할당에 영향을 주는 요소들
  - 주소지정 방식의 수
  - 명령어 내 오퍼랜드의 수
  - 오퍼랜드 저장에 사용되는 레지스터의 수
  - 레지스터 세트의 수
  - 주소 영역(address range)
  - 주소 세분화(address granularity)

가변길이 명령어

- 명령어 형식에서 서로 다른 길이를 가지는 경우
- 길이가 서로 다른 더 많은 종류의 연산 코드들을 쉽게 제공
- 레지스터와 기억장치 참조들을 주소 지정 방식들과 다양하게 결합
  - 주소 지정이 더욱 융통적
- CPU의 복잡도가 증가

주소지정방식

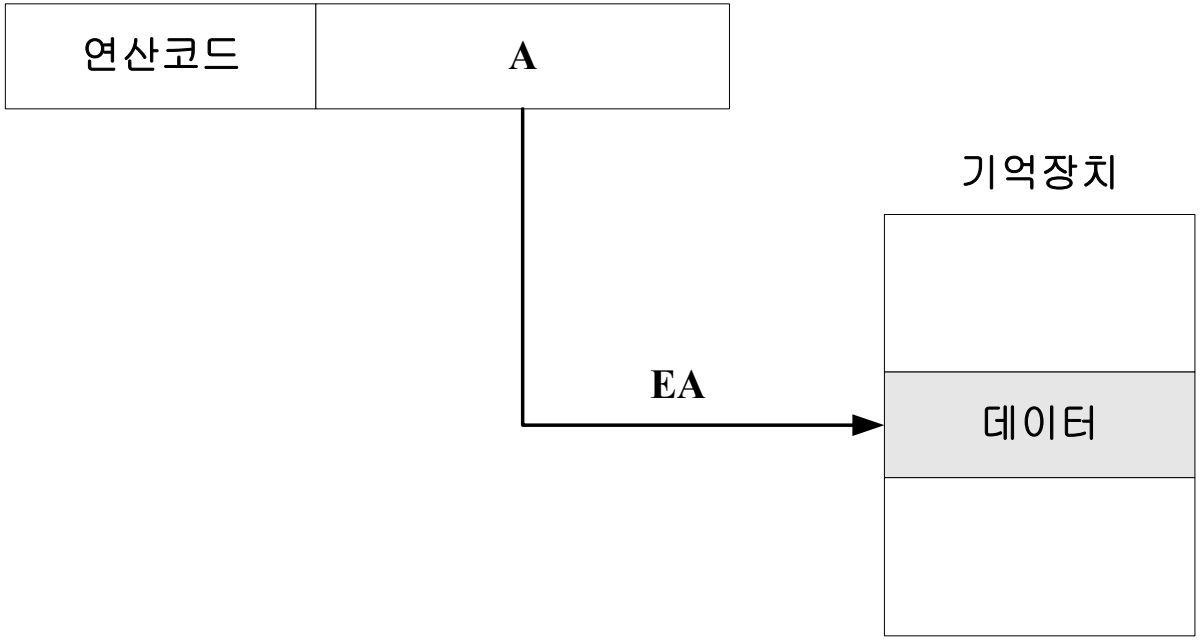
- 다양한 주소지정 방식(addressing mode)을 사용
  - 제한된 명령어 비트들을 적절하게 이용하여 사용자(혹은 프로그래머)로 하여금 여러 가지 방법으로 오퍼랜드를 지정하고 더 큰 용량의 기억장치를 사용할 수 있도록 하기 위함
- 데이터가 저장된 기억장치의 실제 주소를 유효 주소(Effective Address) : EA
- 오퍼랜드 필드가 기억장치 주소를 나타내는 경우 : A
- 오퍼랜드 필드가 레지스터 번호를 나타내는 경우 : R
- 기억장치 A 번지의 내용 : (A), 레지스터 R의 내용 : (R)
- 주소 지정방식
  - 직접 주소지정 방식 (direct addressing mode)
  - 간접 주소지정 방식 (indirect addressing mode)
  - 묵시적 주소지정 방식 (implied addressing mode)
  - 즉치 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)

직접 주소 지정 방식

- 오퍼랜드 필드의 내용이 유효 주소가 되는 방식
- 가장 일반적인 개념의 주소 방식

$$EA = A$$

- 데이터 인출을 위해 한 번만 기억장치에 액세스
- 연산 코드를 제외하고 남은 비트들이 주소 비트로 사용
  - 지정할 수 있는 기억장소의 수가 제한, 많은 수의 주소를 지정 불가능



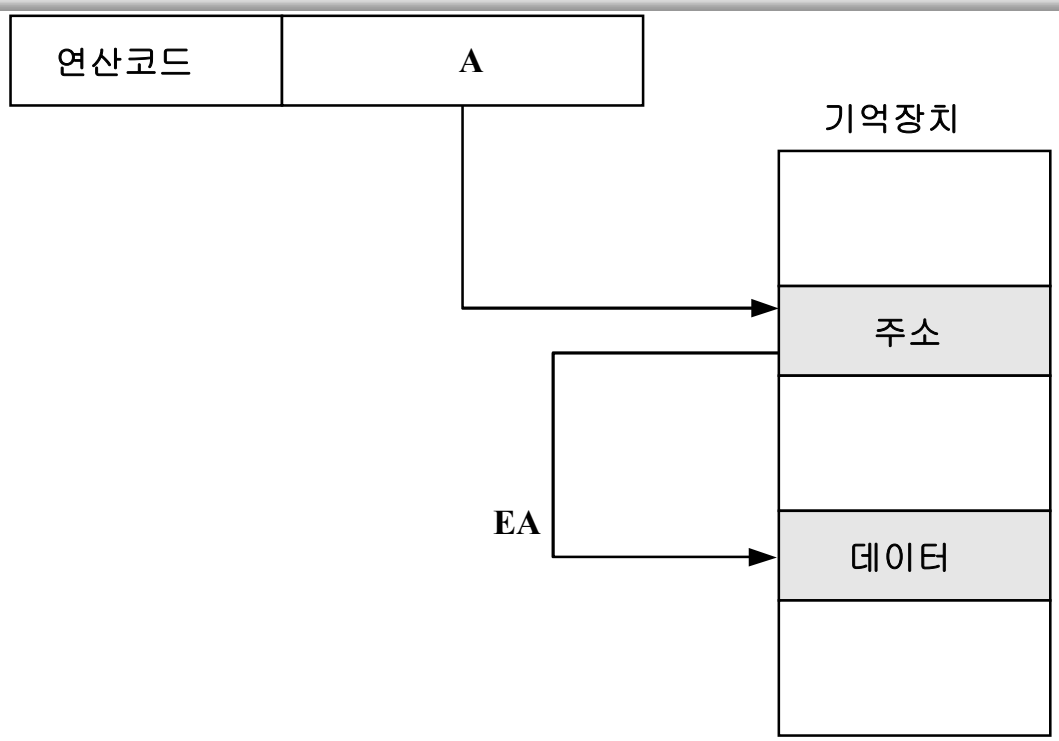
간접 주소 지정 방식

- 오퍼랜드 필드에 데이터 유효 기억장치 주소가 저장되어 있는 주소가 저장
- 그 주소가 가리키는 기억 장소에서 유효 주소 획득

$$EA = (A)$$

- 최대 기억장치용량이 CPU가 한 번에 액세스할 수 있는 단어의 길이에 의하여 결정
  - 기억장치의 구조 변경 등을 통해 확장이 가능
  - 단어 길이가 n 비트라면, 최대 2^n 개의 기억장소들을 주소지정 가능
- 실행 사이클 동안 두 번의 기억장치 액세스가 필요하다는 단점
- 두 번의 액세스
  - 첫 번째 액세스는 주소를 읽기
  - 두 번째는 그 주소가 지정하는 위치로부터 실제 데이터를 인출
- 주소 지정 방식을 표시하는 간접비트(I)필드가 필요

간접 주소 지정 방식



간접주소 지정 방식

연산코드	I	기억장치 주소

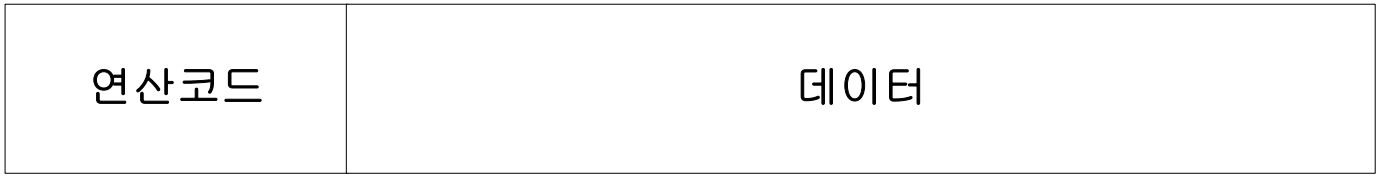
간접 주소 지정 방식에서의 간접 비트 필드

묵시적 주소 지정 방식

- 명령어를 실행하는데 필요한 데이터의 위치가 별도로 지정되어 있지 않음
- 명령어의 연산 코드가 내포하고 있는 방법을 묵시적 주소지정 방식이라고 한다
- 명령어 길이가 짧음
- 명령어의 종류가 제한
- 예> SHL (shift left)

즉치 주소 지정 방식

- 데이터가 명령어에 포함되어 있는 방식
- 오퍼랜드 필드의 내용이 연산에 사용할 실제 데이터
- 프로그램에서 레지스터들이나 변수의 초기 값을 어떤 상수값(constant value)으로 세트하는 데 유용
- 데이터를 인출하기 위하여 기억장치를 액세스할 필요가 없다
- 상수 값의 크기가 오퍼랜드 필드의 비트 수에 의하여 제한



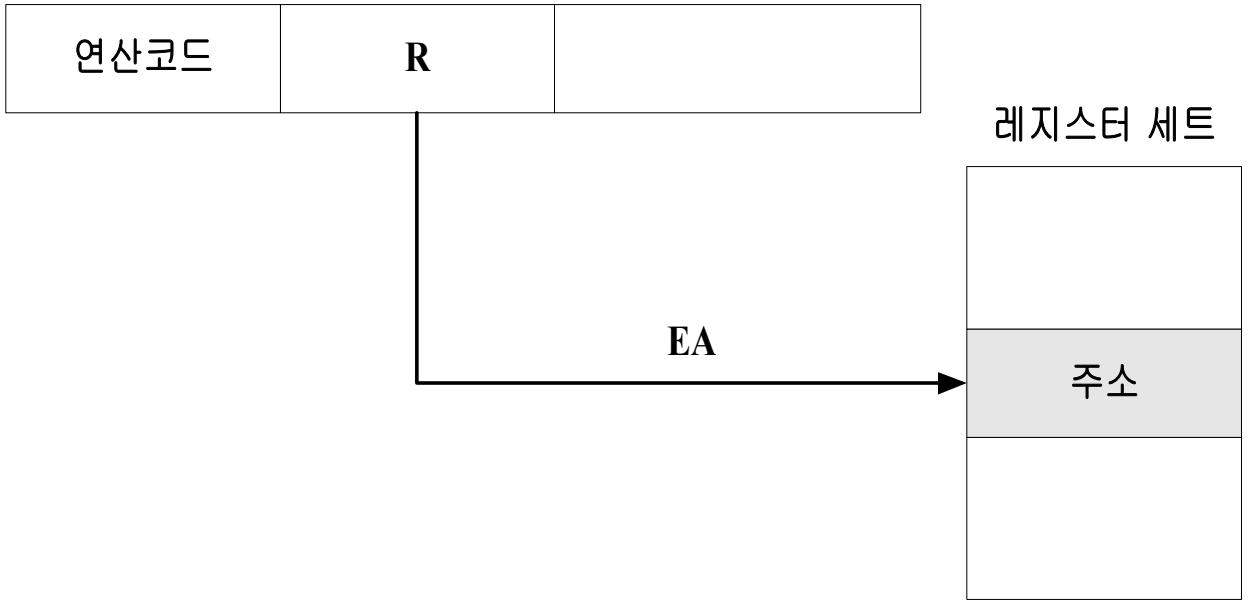
즉치 주소지정 방식에서 오퍼랜드 필드

레지스터 주소 지정 방식

- 연산에 사용할 데이터가 레지스터에 저장
- 오퍼랜드 부분이 레지스터 번호, 유효주소가 레지스터 번호

$$EA = R$$

- 오퍼랜드의 비트수가 k비트, 주소지정에 사용될 수 있는 레지스터들의 수 2^k 개
- 오퍼랜드 필드가 레지스터들의 번호를 나타내기 때문에 비트 수가 적어도 가능
- 데이터 인출을 위하여 기억장치에 액세스 할 필요 없음
- 데이터가 저장될 수 있는 공간이 CPU 내부 레지스터들로 제한



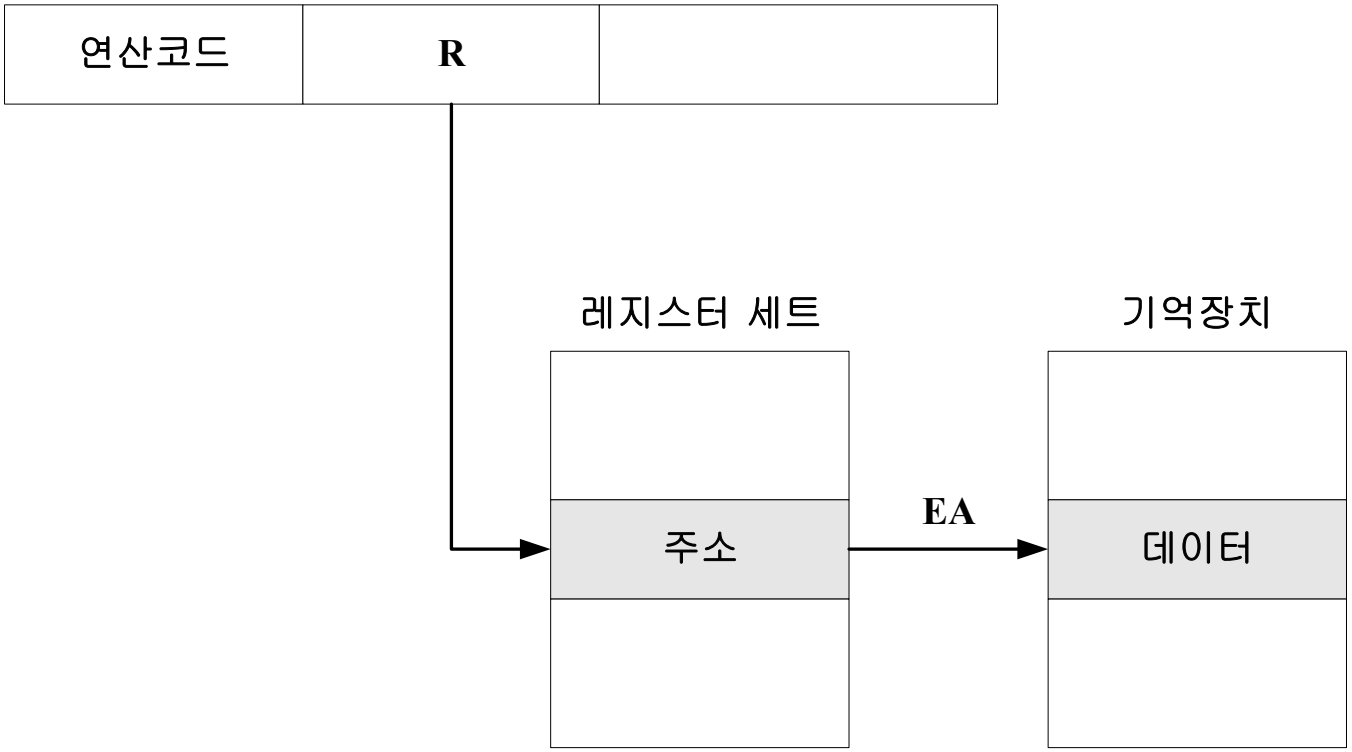
레지스터 간접 주소 지정 방식

- 이 방식은 명령어 형식에서 오퍼랜드 필드가 레지스터 번호를 지정
- 레지스터의 내용이 유효 주소

$$EA = (R)$$

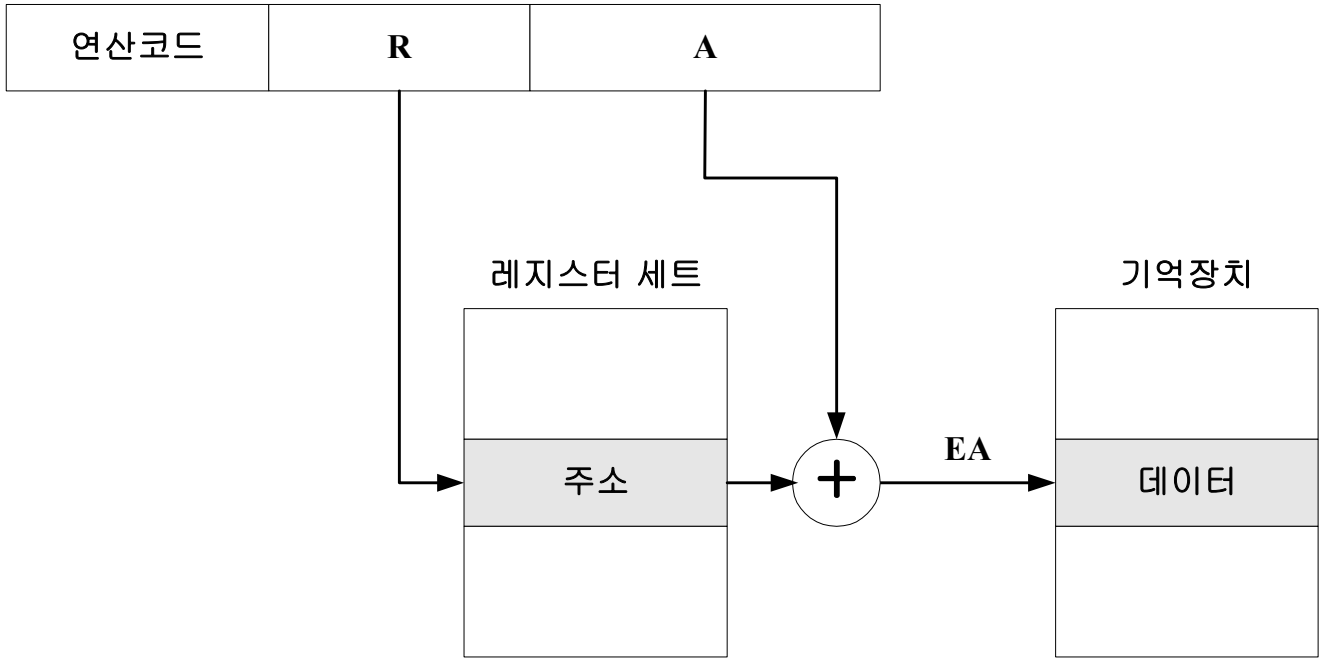
- 주소를 지정 할 수 있는 기억장치 영역이 확장
- 레지스터의 길이에 따라 주소지정 영역이 결정
  - 레지스터의 길이 : 16 비트
  - 주소지정 영역 2^{16} 비트(64K 바이트)
- 한 번의 기억장치 액세스

레지스터 간접 주소 지정 방식



변위(displacement) 주소 지정 방식

- 직접 주소지정 방식과 레지스터 간접 주소지정 방식을 조합한 방식
- 오퍼랜드 필드
  - 레지스터 번호필드
  - 변위 값 필드
  - 두 오퍼랜드의 조합으로 유효 주소가 생성



- 프로그램 카운터(PC)를 레지스터로 사용
- 주로 분기 명령어에서 사용

$$EA = A + (PC)$$

- A는 2의 보수
- A ≥ 0 이면, 앞(forward) 방향으로 분기
- A <0 이면 후(backward) 방향으로 분기

- 전체 기억장치 주소가 명령어에 포함되어야 하는 일반적인 분기 명령어보다 적은 수의 비트 사용
- 분기 범위가 오퍼랜드 필드의 길이에 의하여 제한

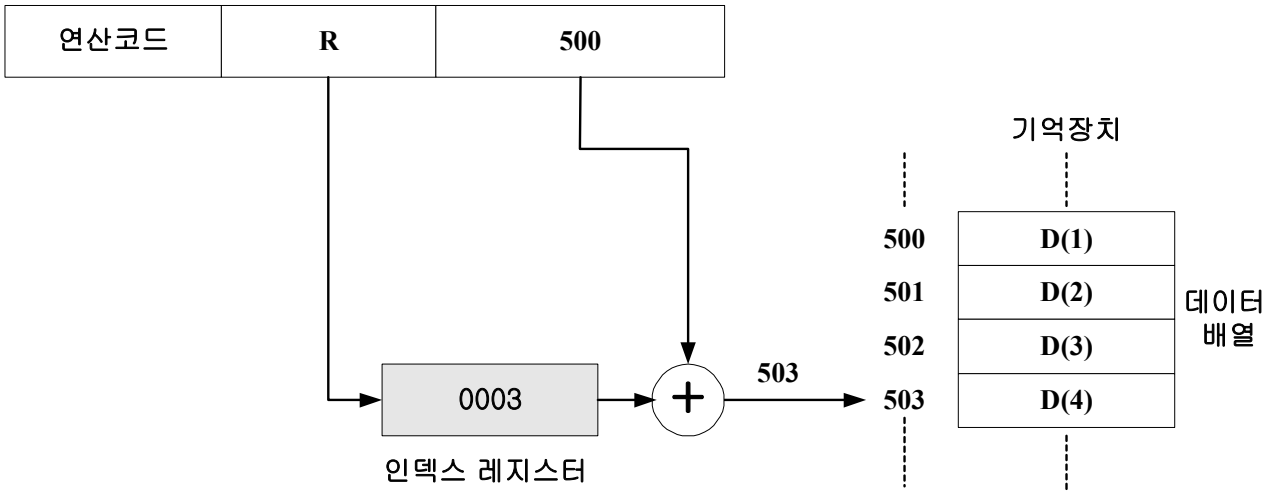
변위 주소 지정 방식-인덱스 주소 지정 방식

- 인덱스 레지스터의 내용과 변위 A를 더하여 유효 주소를 결정하는 방식

$$EA = (IX) + A$$

- 인덱스 레지스터(IX) : 인덱스(index) 값을 저장하는 특수 레지스터
- 방식은 배열 데이터를 액세스할 때 자동 인덱싱(autoindexing)
- 명령어가 실행될 때마다 인덱스 레지스터의 내용이 자동적으로 증가 혹은 감소
- 명령어가 실행되면 아래의 두 연산이 연속적으로 수행

$$EA = (IX) + A$$
$$IX \leftarrow IX + 1$$



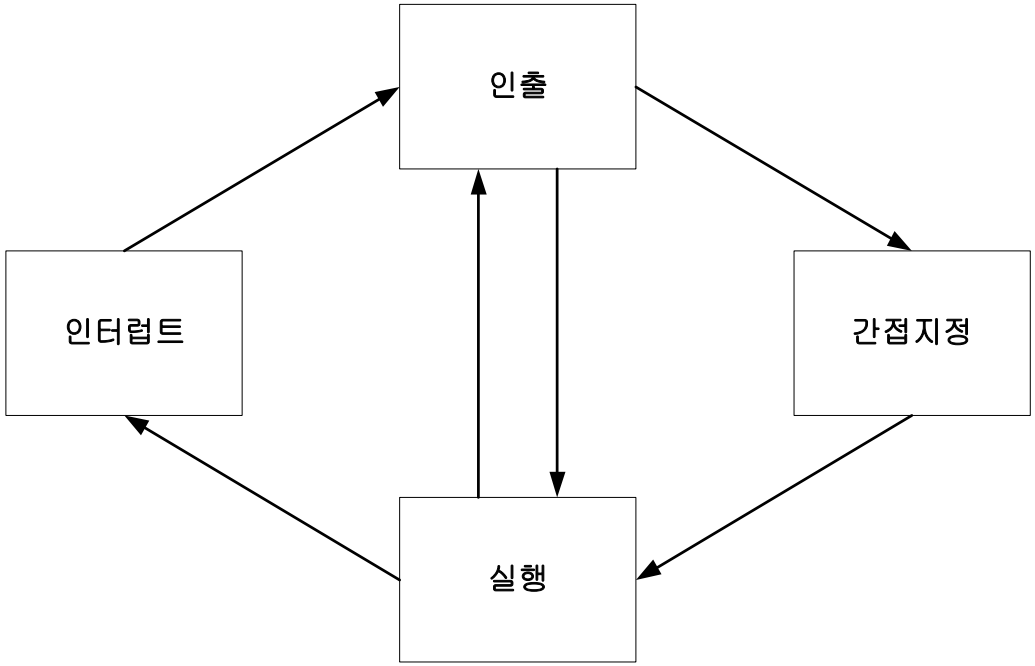
변위 주소 지정 방식 – 베이스-레지스터 주소지정 방식

- 베이스 레지스터의 내용과 변위 A를 더하여 유효 주소를 결정하는 방식
- 서로 다른 세그먼트 내 프로그램의 위치를 지정하는 데 사용

$$EA = (BR) + A$$

간접 사이클

- 간접 주소지정 방식(indirect addressing mode)에서 사용되는 명령어 부 사이클
- 명령어에 포함되어 있는 주소를 이용하여, 실제 명령어 실행에 필요한 데이터를 인출하는 사이클
- 부 사이클은 인출 사이클과 실행 사이클 사이에 위치



간접 사이클에서 수행될 마이크로-연산

t_0 : MAR ← IR(addr)

t_1 : MBR ← M[MAR]

t_2 : IR(addr) ← MBR

- 인출 사이클에서 인출된 명령어가 저장된 명령어 레지스터에서 주소필드 부분을 MAR에 저장
- MAR에 저장된 주소번지의 기억장치에서 실제 주소를 MBR에 적재
- 명령어 레지스터의 주소 필드 부분에 MBR의 내용, 즉 실제 주소를 적재

간접 사이클에서의 데이터 흐름도