Microprocessor (W6)
- Central Processing Unit (CPU) -

Dong Min Kim
Department of IoT
Soonchunhyang University
dmk@sch.ac.kr

Contents

학습목표

- 명령의 주소 지정 방식을 이해하고 동작 원리를 학습한다.
- CISC와 RISC의 차이점에 대해 학습한다.

내용

1 즉시 주소 지정(immediate addressing mode, 즉치 주소 지정)

- 오퍼랜드를 지정하는 가장 간단한 방법
  - 명령어 자체에 오퍼랜드를 포함
  - 오퍼랜드가 포함되어 명령어가 인출될 때 오퍼랜드도 자동으로 인출
  - 즉시(즉치) 오퍼랜드 : 즉시 사용 가능
- 레지스터 R1에 상수 4를 저장하는 즉시 주소 지정 명령어의 예

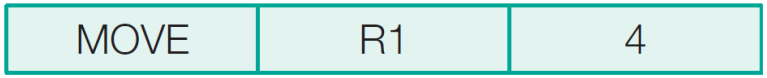

그림 4-13 즉시 주소 지정

- 장점 : 오퍼랜드를 인출을 위한 메모리 참조가 필요 없음
- 단점 : 상수만 가능, 상수 값의 크기가 필드 크기로 제한
- 작은 값의 정수를 지정하는 데 많이 사용

2 직접 주소 지정(direct addressing mode)

- 메모리에 위치한 오퍼랜드의 전체 주소 지정
- 직접 주소 지정도 즉시 주소 지정처럼 사용 제한
  - 명령어는 항상 정확히 동일한 메모리 위치 액세스
  - 값이 변할 수는 있지만 위치는 변할 수 없음
  - 컴파일할 때 알려진 주소의 전역 변수에 액세스하는 데만 사용 가능

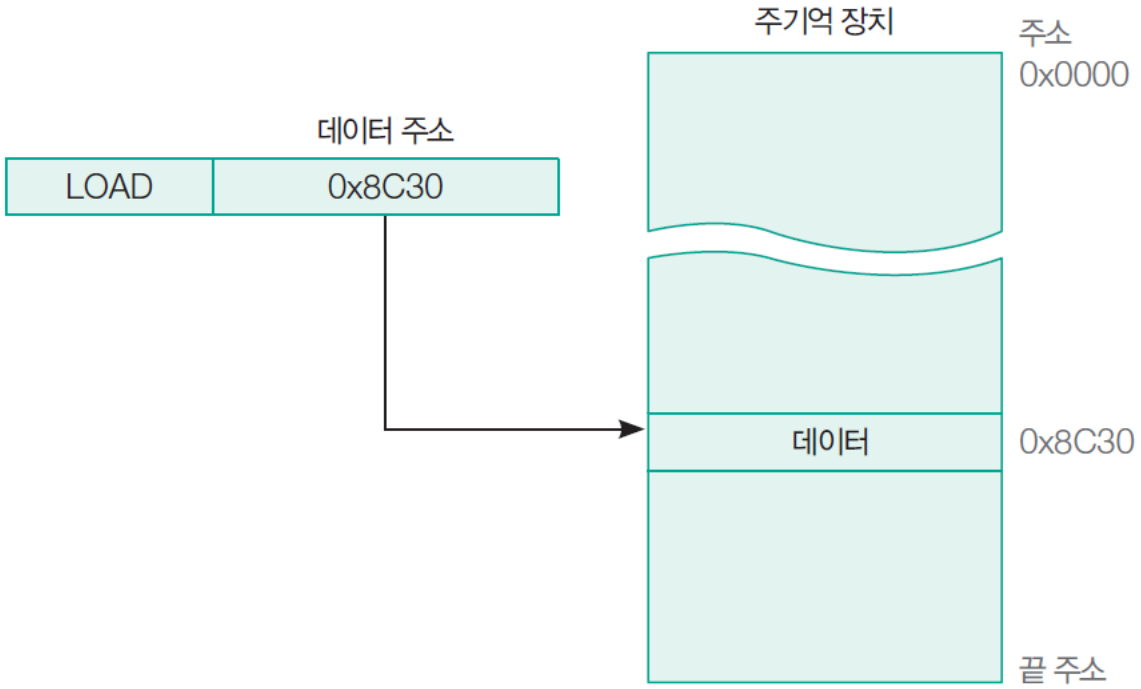

그림 4-14 직접 주소 지정

3 레지스터 주소 지정(register addressing mode)

- 직접 주소 지정과 개념은 같고 그 위치가 메모리 대신 레지스터
- 가장 일반적인 주소 지정 방식:
  - 레지스터는 액세스가 빠르고 주소가 짧기 때문
  - 대부분의 컴파일러는 루프 인덱스처럼 가장 자주 액세스할 변수를 레지스터에 넣기 위해 많은 노력을 기울임
- 많은 프로세서에서 사용
- RISC 등에서는 LOAD, STORE 명령을 제외하고 대부분의 명령어에서 레지스터 주소 지정 방식만 사용
- LOAD나 STORE 명령어
  - 한 오퍼랜드는 레지스터고, 다른 한 오퍼랜드는 메모리 주소

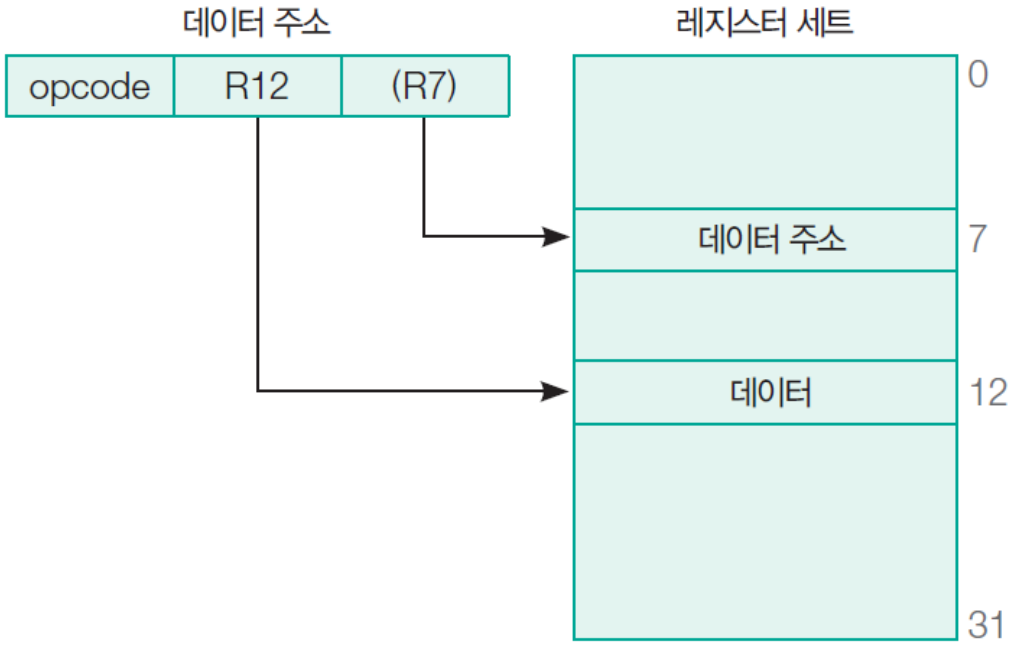

그림 4-15 레지스터 주소 지정

4 레지스터 간접 주소 지정(register indirect addressing mode)

- 직접 주소를 명령어에는 포함하지 않음
  - 메모리의 주소는 레지스터에 저장: 포인터(pointer)
  - 레지스터 간접 주소 지정의 가장 큰 장점 : 명령어에 전체 메모리 주소가 없어도 메모리 참조가능

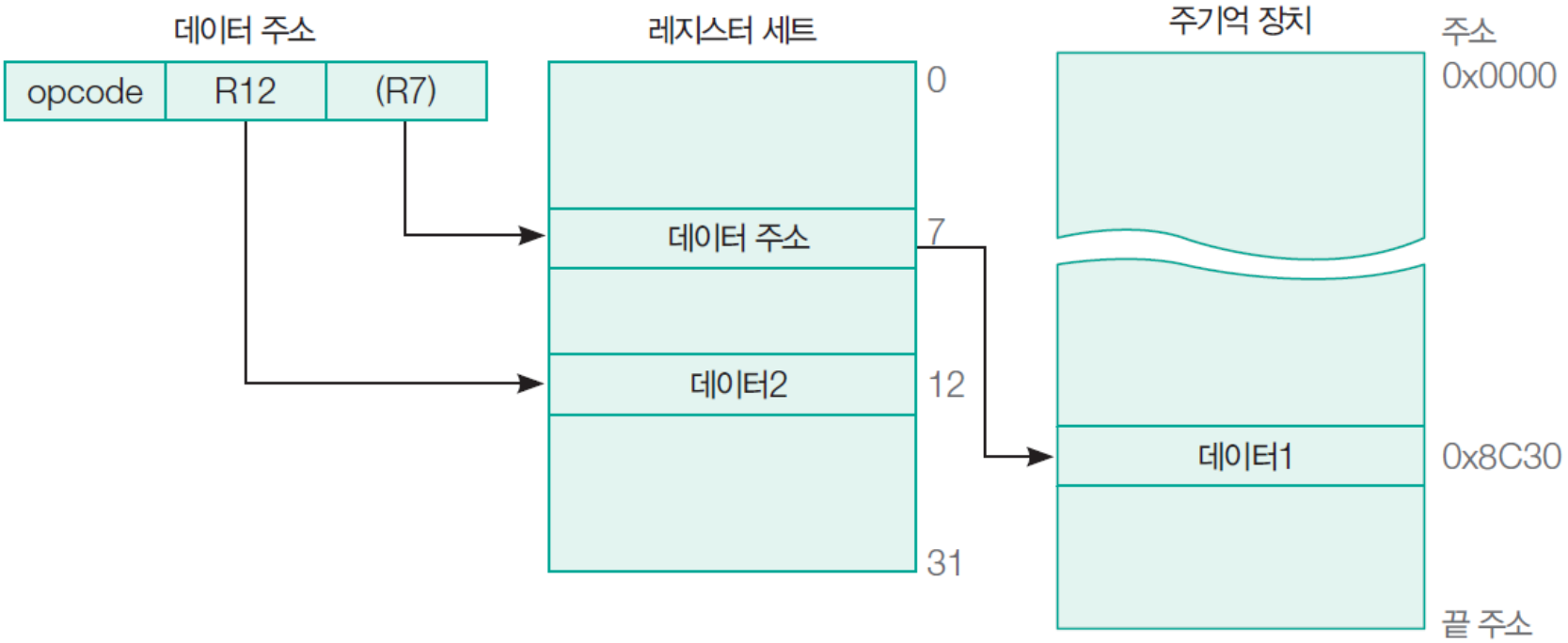

그림 4-16 레지스터 간접 주소 지정

05 주소 지정 방식

❖ 레지스터 R1에 있는 요소의 합계 계산 예

요소가 100개인 1차원 정수 배열의 요소를 단계별로 설명하는 루프를 생각해 보자. 루프 외부에서는 R2 같은 다른 레지스터를 배열의 첫 번째 요소를 가리키도록 설정할 수 있으며 다른 레지스터, 예를 들어 R3은 배열을 벗어나는 첫 번째 주소를 가리키도록 설정할 수 있다. 배열이 4바이트(32비트 정수)인 정수 100개가 있는 경우 배열이 A에서 시작하면 배열을 벗어나는 첫 번째 주소는 A+400이 된다. 이 계산을 수행하는 일반적인 어셈블리 코드는 다음과 같다.

❖ 여러 주소 지정 방식 사용

- 첫 번째 명령어에서 오퍼랜드(목적지) 하나는 레지스터 주소 지정이고, 다른 오퍼랜드는 즉시 주소 지정(상수)
- 두 번째 명령어는 A의 주소를 R2에 저장
- 세 번째 명령어는 배열 A를 벗어나 나타나는 첫 번째 워드 주소

05 주소 지정 방식

❖ 레지스터 R1에 있는 요소의 합계 계산 예

```
            MOVE R1, 0          ; 계산 결과가 저장될 R1에 초깃값 0 저장
            MOVE R2, A          ; R2는 배열 A의 주소
            MOVE R3, A+400      ; R3은 배열 A를 벗어나는 첫 번째 주소
LOOP:       ADD R1, (R2)        ; R2를 이용하여 간접 주소를 지정하고 피연산자를 가져옴
            ADD R2, 4           ; R2 레지스터를 4만큼 증가시킴(4바이트), 바이트 단위 주소 지정
            CMP R2, R3          ; R2와 R3를 비교, 즉 끝에 도달였는가를 판단하기 위함
            BLT LOOP            ; R2< R3이면 LOOP로 가서 반복함
```

- 특이한 점 : 루프 자체에 메모리 주소가 포함되지 않음
- 네 번째 명령어 : 레지스터 주소 지정과 레지스터 간접 주소 지정
- 다섯 번째 명령어 : 레지스터 주소 지정과 즉시 주소 지정을
- 여섯 번째 명령어 : 둘 다 레지스터 주소 지정

- BLT(Branch Less Than) : 메모리 주소를 사용 가능하지만, BLT 명령어 자체에 상대적인 8비트 변위로 분기할 주소를 지정할 때가 많음
- 메모리 주소의 사용을 완전히 피함으로써 짧고 빠른 루프 가능

5 변위 주소 지정(displacement addressing mode)

- 특정 레지스터에 저장된 주소에 변위(offset: 오프셋)을 더해 실제 오퍼랜드가 저장된 메모리 위치 지정
- 특정 레지스터가 무엇인지에 따라 여러 주소 지정 방식 가능
- 예 : 인덱스 주소 지정 방식은 인덱스 레지스터가 되고, 상대 주소 지정 방식에서는 PC가 특정 레지스터로 지정

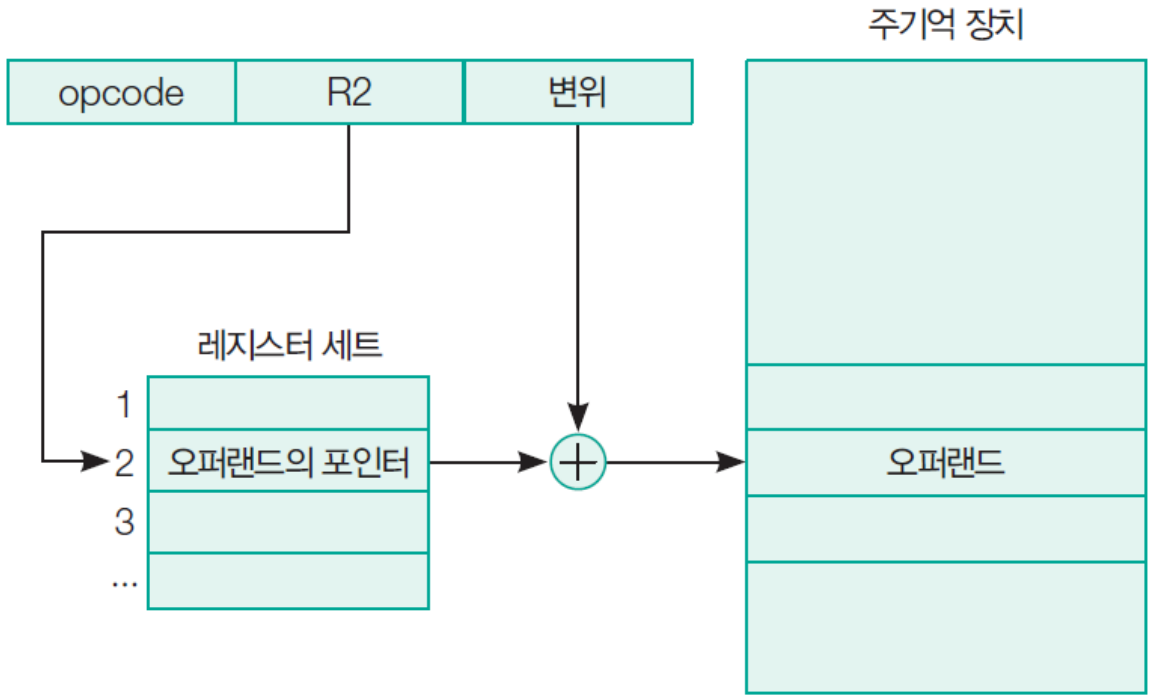

그림 4-17 변위 주소 지정

05 주소 지정 방식

❑ 인덱스 주소 지정(indexed addressing mode)

- 레지스터(명시적 또는 암시적)에 일정한 변위를 더해 메모리 주소 참조
- 특정 레지스터가 무엇인지에 따라 여러 주소 지정 방식 가능
- 예 : 인덱스 주소 지정 방식은 인덱스 레지스터가 되고, 상대 주소 지정 방식에서는 PC가 특정 레지스터로 지정

```
                MOVE R1, 0          ; R1은 합이 저장될 장소이며, 초깃값으로 0을 설정
                MOVE R2, 0          ; R2는 배열 A의 인덱스 i가 저장될 장소이며, 4씩 증가됨
                MOVE R3, 400        ; R3 400이 될 때까지 4씩 증가함
LOOP:           ADD R1, A(R2)       ; R1 = R1 + A[i]
                ADD R2, 4           ; i = i + 4(워드 크기 = 4bytes)
                CMP R2, R3          ; 100개 모두 계산되었는지 비교
                BLT LOOP
```

05 주소 지정 방식

- 프로그램의 알고리즘 : 단순하며 3개의 레지스터 필요

  ❶ R1 : A의 누적 합계가 저장된다.

  ❷ R2 : 인덱스 레지스터로 배열의 i를 저장한다.

  ❸ R3 : 상수 400

- 명령어 루프에 4개 실행

  - 소스 값의 계산은 인덱스 주소지정 사용
  - 배열 A의 인덱스가 저장된 인덱스 레지스터 R2와 상수(0~400)가 더해져 메모리를 참조(A(R2))하는 데 사용
  - 덧셈 연산은 메모리 이용 : 사용자가 볼 수 있는 레지스터에는 저장되지 않음

```
MOV R1, A(R2)
```

05 주소 지정 방식

- R1이 목적지인 레지스터 주소 지정
- 소스는 R2가 인덱스 레지스터이고, A가 변위인 인덱스 주소 지정

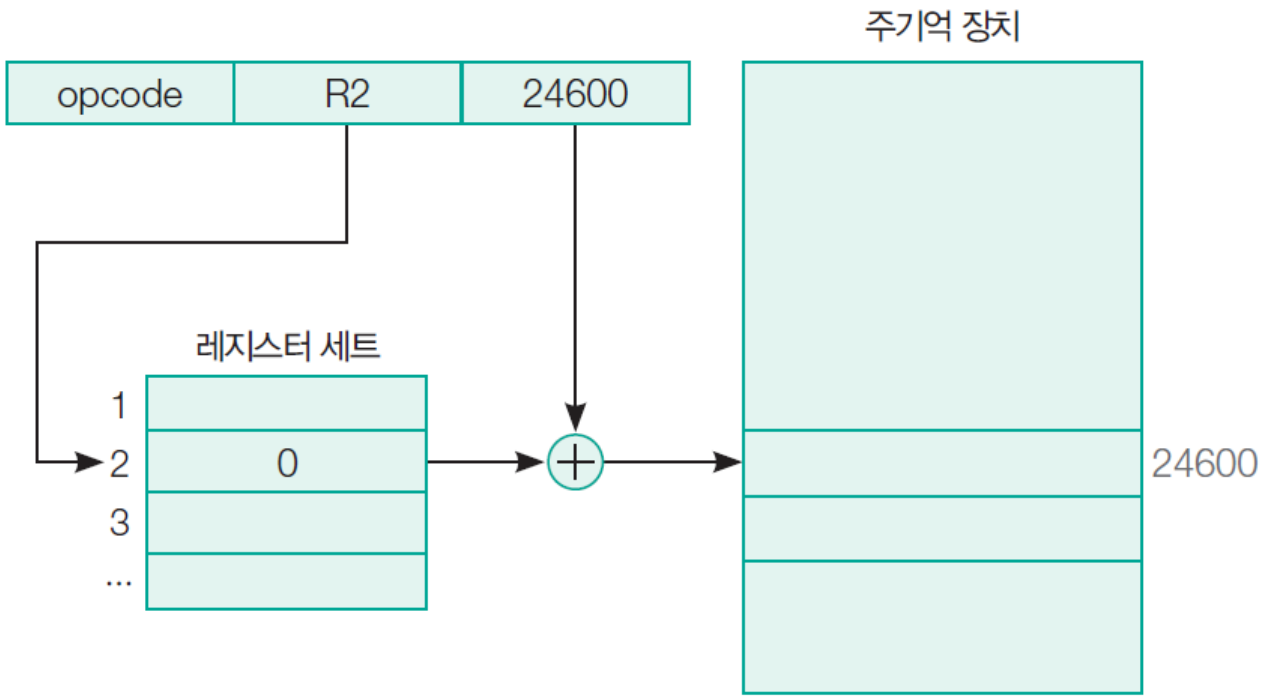

그림 4-18 인덱스 주소 지정

❑ 상대 주소 지정(relative addressing mode)

- PC 레지스터 사용
- 현재 프로그램 코드가 실행되고 있는 위치에서 앞 또는 뒤로 일정한 변위만큼 떨어진 곳의 데이터 지정

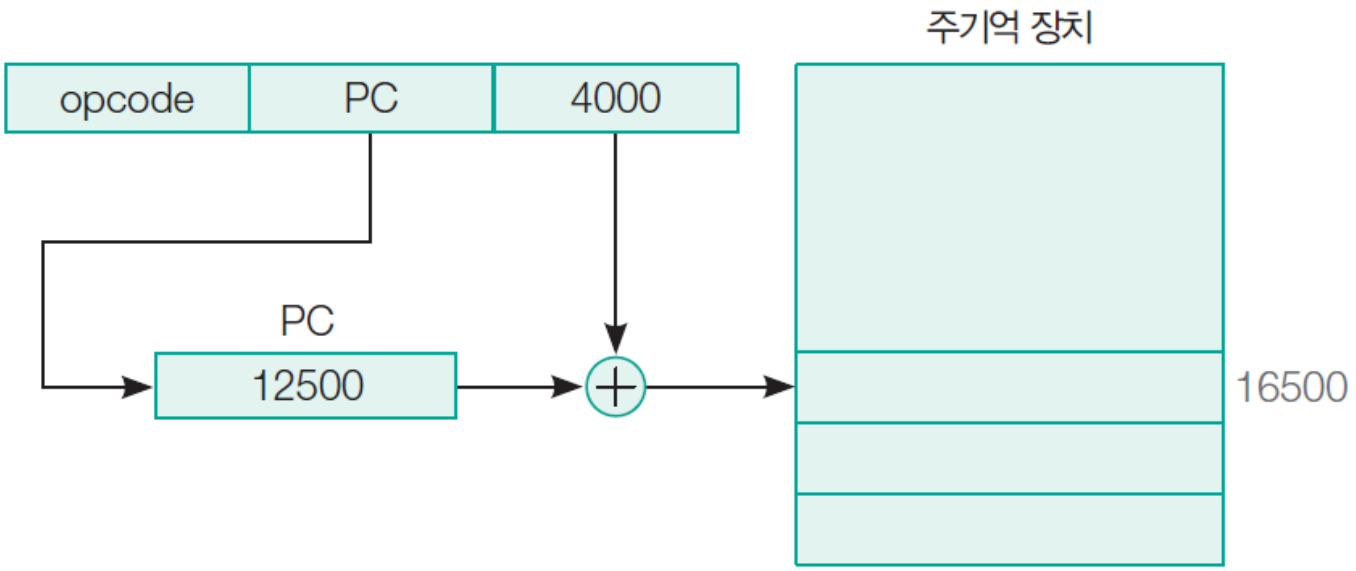

그림 4-19 상대 주소 지정

05 주소 지정 방식

❑ 베이스 주소 지정(base addressing mode)

- 인텔 프로세서에는 세그먼트 레지스터가 6개 있음
- 이 중 하나를 베이스 레지스터로 하고 이 레지스터에 변위를 더해 실제 오퍼랜드가 있는 위치를 찾는 방식
- SS(stack segment): 스택 데이터가 저장되어 있는 스택 위치에 대한 포인터
- CS(code segment): 프로그램 코드가 저장되어 있는 시작 위치에 대한 포인터
- DS(data segment): 데이터 영역에 대한 시작 포인터
- ES, FS, GS: 엑스트라 세그먼트(extra segment)에 대한 포인터

  - 엑스트라 세그먼트는 데이터 세그먼트의 확장 영역

05 주소 지정 방식

- 이 레지스터 중 하나를 베이스로 사용하여 실제 오퍼랜드 위치 지정

  - 데이터 세그먼트를 베이스로 사용: 오프셋 200만큼 떨어진 주소 25600에서 오퍼랜드 위치함

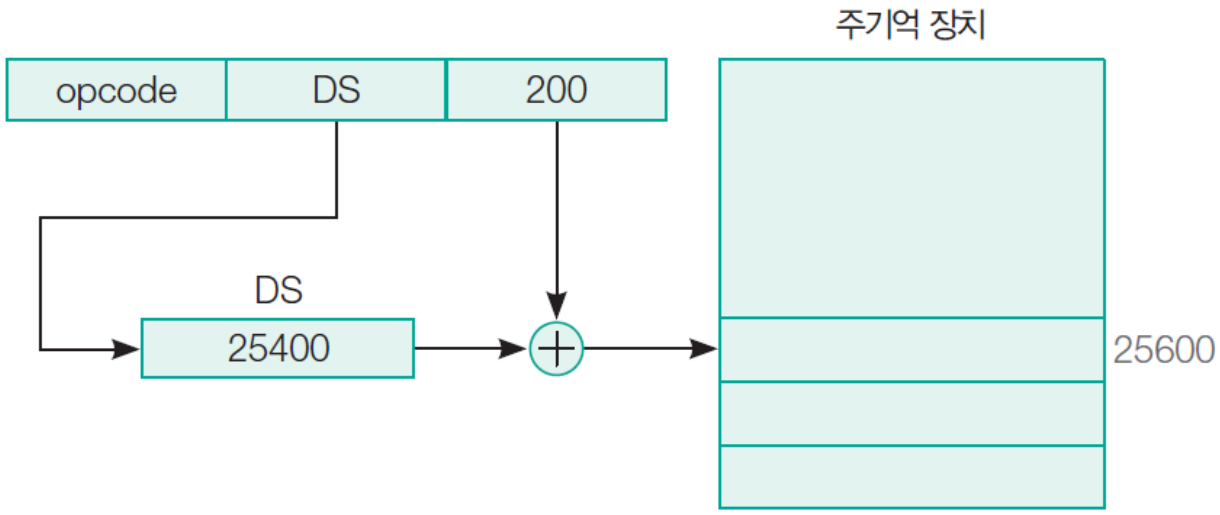

그림 4-20 베이스 주소 지정

6 간접 주소 지정(indirect addressing mode)

- 메모리 참조가 두 번 이상 일어나는 경우
- 데이터를 가져오는 데 많은 시간 소요
- 프로세서와 주기억 장치간의 속도 차가 많은 현재의 프로세서의 경우

  - 오퍼랜드를 인출하는 데 오래 걸리므로 전체 프로그램의 수행 시간은 길어짐
  - 현재는 간접 주소 지정을 지원하는 프로세서는 거의 없음

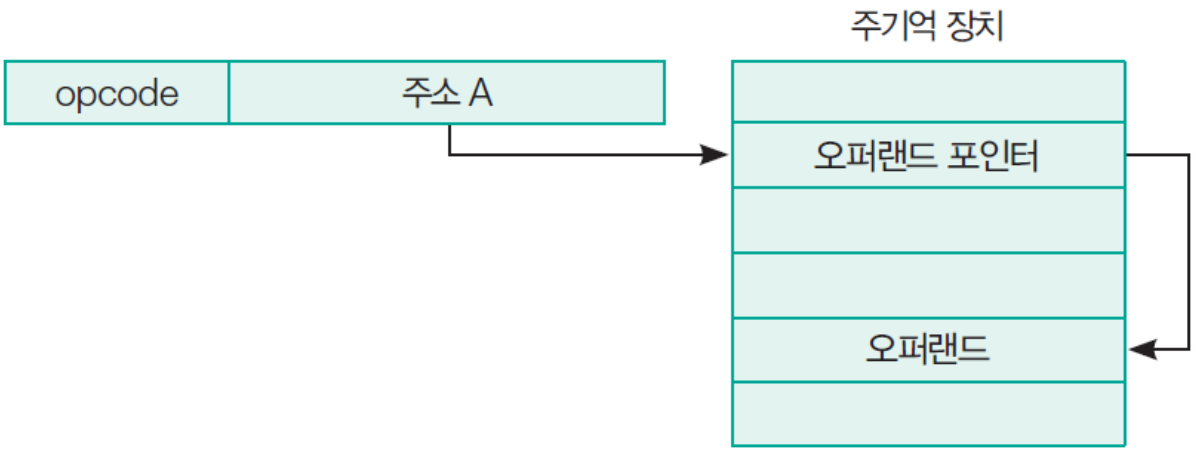

그림 4-21 간접 주소 지정

7 묵시적 주소 지정(implied addressing mode)

- 오퍼랜드의 소스나 목적지를 명시하지 않음
- 암묵적으로 그 위치를 알 수 있는 주소 지정 방식
- 예를 들어 서브루틴에서 호출한 프로그램으로 복귀할 때 사용하는 RET 명령
  - 명령어 뒤에 목적지 주소가 없지만 어디로 복귀할지 자동으로 알 수 있음
- PUSH, POP 등 스택 관련 명령어
  - 스택이라는 목적지나 소스가 생략
  - PUSH R1 : 레지스터 R1의 값을 스택에 저장
  - POP : 스택의 TOP에 있는 값을 AC로 인출
- 누산기를 소스나 목적지로 사용하는 경우도 생략 가능

9 실제 프로세서에서 주소 지정 방식

- Core i7, ARM 및 AVR에서 사용되는 주소 지정 방식

표 4-4 주요 프로세서의 주소 지정 방식 비교

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

06 CISC와 RISC

❑ CISC(Complex Instruction Set Computer)

❖ 1950년대 초 모리스 윌크스(M. V. Willkes)

- 제어 장치를 소프트웨어적인 방법(마이크로 프로그래밍)으로 구현
- 당시에는 하드웨어가 아닌 소프트웨어적 구성이 사람의 흥미를 끌었으나 빠르고 저렴한 제어기억 장치가 필요하여 구현의 어려움

❖ 1964년 4월 IBM 시스템/360

- 가장 큰 모델을 제외하고 모두 마이크로 프로그래밍 사용
- CISC(Complex Instruction Set Computer) 프로세서의 제어 장치를 구현하는 데 널리 사용

❖ 1970년대 후반

- 명령어가 매우 복잡한 컴퓨터가 연구
- 인텔 계열 : 계속적인 명령어 추가 - 대표적인 CISC 프로세서

❖ CISC 프로세서

- 해독기 : 기계어를 제어 기억 장치에 저장된 마이크로 명령 루틴으로 실행

06 CISC와 RISC

❑ **RISC**(Reduced Instruction Set Computer)

❖ 1980년 버클리 그룹

- 데이비드 패터슨(David Patterson) & 카를로 세퀸(Carlo Sequin)

❖ 마이크로 명령을 사용하지 않는 VLSI CPU 칩 설계

❖ 1981년 스탠포드 대학교: 존 헤네시(John Hennessy)

- MIPS라는 다른 칩 설계&제작
- SPARC 및 MIPS로 각각 발전

❖ 기존 제품과 호환할 필요가 없었으므로 시스템 성능을 극대화할 수 있는 새로운 명령어 세트를 자유롭게 선택

- 초기 : 빠르게 실행할 수 있는 단순 명령이 강조
- 추후에는 빠르게 실행할 수 있는 명령의 설계가 좋은 성능을 보장한다는 것을 깨달음
- 하나의 명령어가 실행되는 데 걸리는 시간보다 초당 얼마나 많은 명령어를 시작할 수 있는지가 더 중요함

06 CISC와 RISC

❖ RISC 설계 당시 특징

- 상대적으로 적은 개수의 명령어
- 하나의 명령어가 실행되는 데 걸리는 시간보다 초당 얼마나 많은 명령어를 시작할 수 있는지가 더 중요함
- 명령어 개수가 대략 50개
  - 기존 VAX나 대형 IBM 메인 프레임의 명령어 개수인 200~300개보다 훨씬 적음

❖ VAX, 인텔, 대형 IBM 메인 프레임에 반기

- 컴퓨터를 설계하는 가장 좋은 방법은 레지스터 2개를 어떻게든 결합하고
- 명령어를 적게 하여
- 결과를 레지스터에 다시 저장하는 것
- 반기의 근거 : RISC 시스템은 선호할 가치가 있음
  - CISC가 명령어 1개로 처리하는 일을 RISC는 명령어 4~5개로 처리하더라도 기계어를 마이크로 명령으로 해독하지 않아도 되므로 10배 빠르면 이길 수 있음
  - 주기억 장치의 속도가 제어 기억 장치의 속도와 비슷해짐으로써 CISC의 해독으로 인한 손실이 더 커짐

❖ CISC와 RISC 특징 비교

표 4-5 CISC와 RISC의 특징

CISC	RISC
하드웨어를 강조한다.	소프트웨어를 강조한다.
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.

06 CISC와 RISC

❖ RISC 기술의 기능적인 이점에도 CISC 시스템을 무너뜨리지는 못한 이유

- 첫째, 이전 버전과 호환성 문제와 소프트웨어에 수십억 달러를 투자한 인텔 계열 회사들 때문
- 둘째, 인텔이 CISC 아키텍처에도 RISC 아이디어를 사용할 수 있었기 때문
  - 인텔 CPU에는 486부터 RISC 코어 포함
  - RISC 코어는 단일 CISC 방식으로 복잡한 명령어를 해석하면서 단일 데이터 경로 사이클에서 가장 단순한(보통 가장 일반적인) 명령어 실행
  - 일반 명령어는 빠르지만 일반적이지 않은 명령어는 느림
  - 하이브리드 방식 : 순수한 RISC 설계만큼 빠르지는 않지만 전반적인 성능 향상 - 기존 소프트웨어를 수정하지 않고 실행
- CISC는 하나의 프로그램에 사용되는 명령어 개수 최소화, 명령어 사이클 개수를 희생하는 접근법
- RISC는 명령어 사이클 개수를 줄이고 프로그램당 명령어 개수에 가치 부여

❖ CISC와 RISC의 비교

표 4-6 CISC와 RISC의 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

❑ 현대 컴퓨터 시스템의 주요 설계 원칙

- 모든 명령어는 하드웨어가 직접 실행한다.
- 어떤 명령어가 시작되었을 때 최대 효율을 발휘하는가?
- 명령어는 쉽게 해석할 수 있어야 한다.
- 읽기와 쓰기만 메모리를 참조하여야 한다.
- 많은 레지스터를 제공해야 한다.