

Microprocessor (W5)

- Central Processing Unit (CPU) 2 -

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

Contents

03 레지스터

04 컴퓨터 명령어

1 레지스터 동작

- 레지스터는 CPU가 사용하는 데이터와 명령어를 신속하게 읽어 오고 저장하고 전송하는 데 사용
- 레지스터는 메모리 계층의 최상위에 있으며 시스템에서 가장 빠른 메모리
- 매우 단순한 마이크로프로세서는 누산기(AC) 레지스터 1개로만 구성 가능
- 레지스터 용도에 따른 종류
 - 누산기(Accumulator, AC)
 - 프로그램 카운터(Program Counter, PC)
 - 명령 레지스터(Instruction Register, IR)
 - 인덱스 레지스터(Index Register, IX)
 - 스택 포인터(Stack Pointer, SP)
 - 메모리 데이터 레지스터(Memory Data Register : MDR, Memory Buffer Register : MBR)
 - 메모리 주소 레지스터(Memory Address Register, MAR)
- 데이터(범용) 레지스터는 보통 8~32개 정도, 많으면 128개 이상인 경우도 있음
- 특수 레지스터는 8~16개 정도

03 레지스터

❖ 레지스터 동작 개념

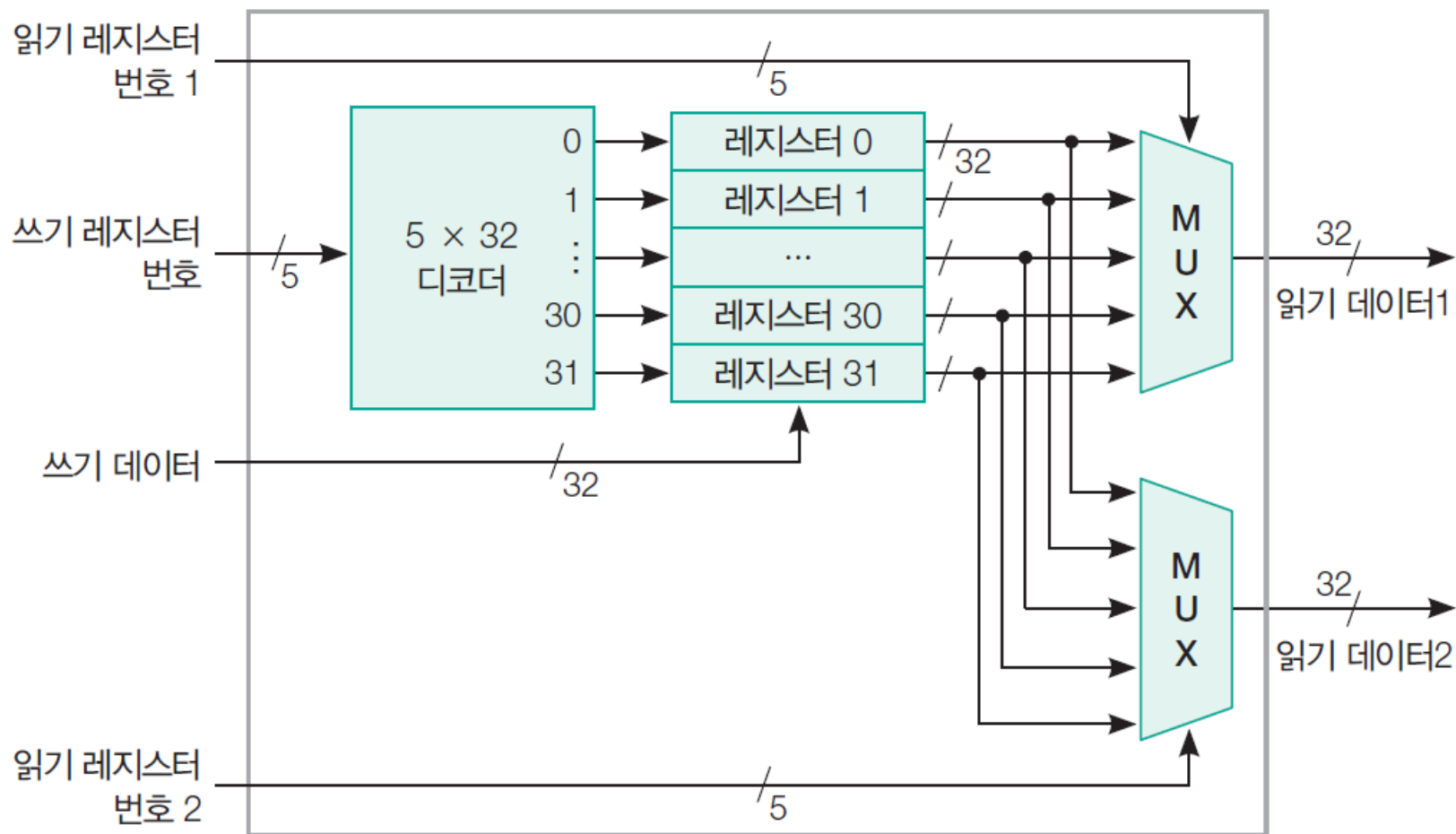


그림 4-6 레지스터 동작 개념

2 레지스터 종류

- ❖ **메모리 주소 레지스터**(Memory Address Register, MAR)
 - CPU가 읽고 쓰기 위한 데이터의 메모리 주소 저장
 - 메모리에 데이터를 저장하거나 읽을 때 필요한 메모리 위치의 주소를 MAR로 전송
- ❖ **메모리 버퍼 레지스터**(Memory Buffer Register, MBR, MDR)
 - 메모리에서 데이터를 읽거나 메모리에 저장될 명령의 데이터를 일시적 저장
 - 명령어 내용은 명령 레지스터로 전송되고, 데이터 내용은 누산기 또는 I/O 레지스터로 전송
- ❖ **입출력 주소 레지스터**(I/O Address Register, I/O AR)
 - 특정 I/O 장치의 주소를 지정하는 데 사용
- ❖ **입출력 버퍼 레지스터**(I/O Buffer Register, I/O BR)
 - I/O 모듈과 프로세서 간에 데이터를 교환하는 데 사용

03 레지스터

❖ 프로그램 카운터(PC)

- 명령 포인터 레지스터라고도 하며, 실행을 위해 인출(fetch)할 다음 명령의 주소를 저장하는데 사용
- 명령어가 인출되면 PC 값이 단위 길이(명령 크기)만큼 증가
- 항상 가져올 다음 명령의 주소 유지

❖ 명령 레지스터(Instruction Register, IR)

- 주기억 장치에서 인출한 명령어 저장
- 제어 장치는 IR에서 명령어를 읽어 와서 해독하고 명령을 수행하기 위해 컴퓨터의 각 장치에 제어신호 전송

❖ 누산기(ACcumulator register, AC)

- ALU 내부에 위치하며, ALU의 산술 연산과 논리 연산 과정에 사용
- 제어 장치는 주기억 장치에서 인출된 데이터 값을 산술 연산 또는 논리 연산을 위해 누산기에 저장
- 이 레지스터는 연산할 초기 데이터, 중간 결과 및 최종 연산 결과 저장
- 최종 결과는 목적지 레지스터나 MBR을 이용하여 주기억 장치로 전송

03 레지스터

❖ 스택 제어 레지스터(Stack Control Register, Stack Pointer)

- 메모리의 한 블록이며, 데이터는 후입 선출(Last In-First Out, LIFO)로 검색
- 메모리 스택을 관리하는 데 사용
- 크기는 2 또는 4바이트

❖ 플래그 레지스터(Flag Register, FR)

- CPU가 작동하는 동안 특정 조건의 발생을 표시하는 데 사용
- 1바이트 또는 2바이트인 특수 목적 레지스터
- 예를 들어 산술 연산 또는 비교 결과로 제로 값이 누산기에 입력되면 제로 플래그를 1로 설정
- 상태 레지스터(Status Register, SR), 프로그램 상태 워드(Program Status Word, PSW)라고도 함

❖ 데이터 레지스터(Data Register, 범용 레지스터)

- CPU내의 데이터를 일시적으로 저장하기 위한 레지스터
- 고정 소수, 부동 소수로 구분하여 따로 저장하는 경우도 있으며,
- 어떤 프로세서는 상수 0 또는 1을 저장할 수 있도록 하는 레지스터도 있다.

03 레지스터

❖ 인텔 x86 레지스터 종류

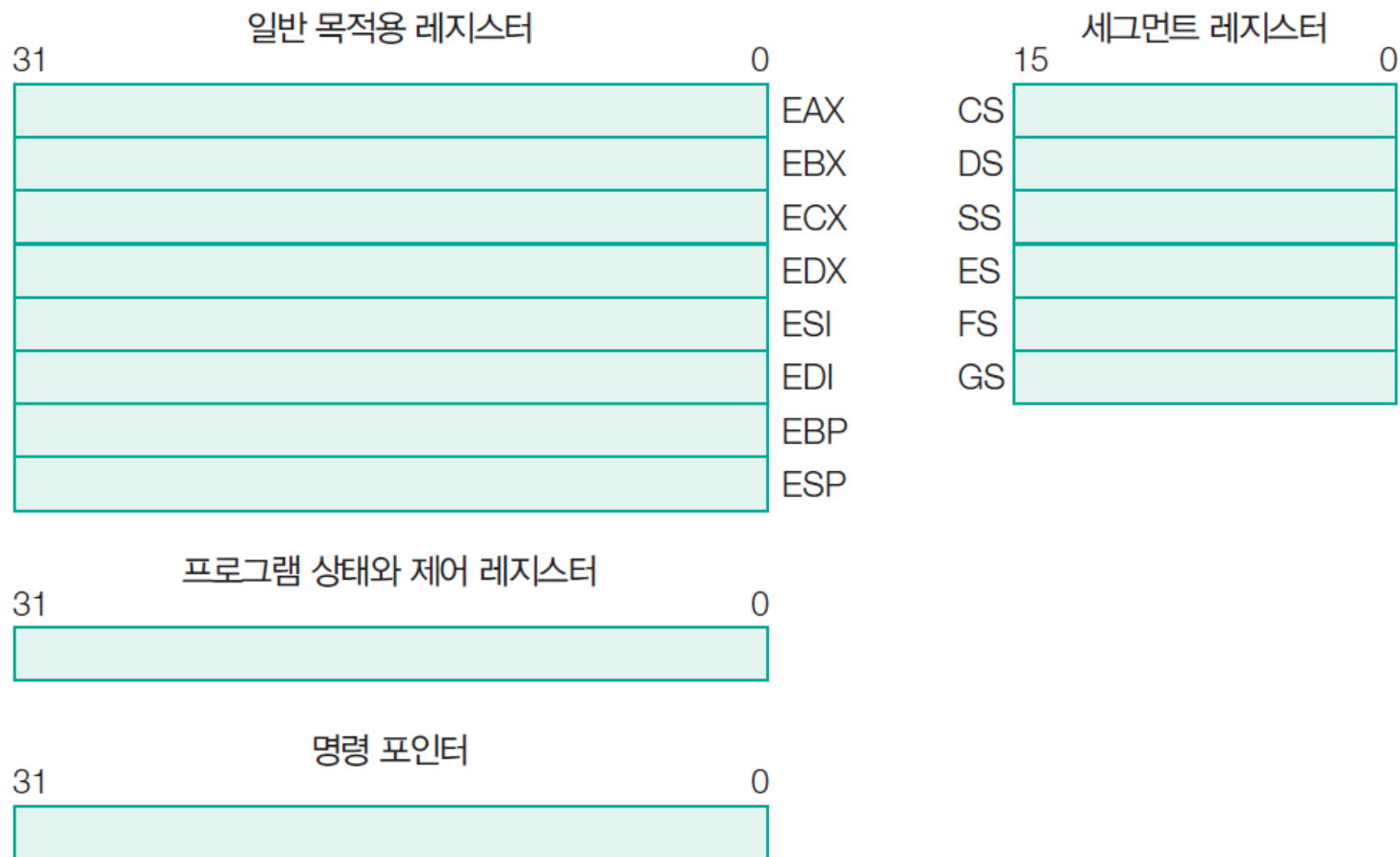


그림 4-7 인텔 x86 레지스터 종류

03 레지스터

3 레지스터 전송(LOAD, STORE, MOVE 명령 등)

- 3가지 레지스터 전송 명령 : LOAD, STORE, MOVE

LOAD	• 주기억 장치에서 레지스터로 데이터를 읽음
STORE	• 레지스터에서 주기억 장치로 데이터를 저장
MOVE	• 레지스터에서 레지스터로 데이터를 이동

- 인텔 프로세서는 이 세 가지를 MOVE 명령으로 모두 처리한다.
- MOVE 명령어를 사용하여 데이터 교환이나 데이터형 변환이 가능하다.

03 레지스터

❖ 데이터 교환

- MOVE 명령을 세 번 사용하여 두 오퍼랜드를 교환할 수 있다.
- 바이트 교환을 이용하여 빅 엔디안을 리틀 엔디안으로 또는 그 반대로 만들 수 있다.

❖ 데이터형 변환

- 크기가 작은 레지스터에 저장된 정수를 큰 레지스터로 이동하여 데이터형을 변환한다.
- 8비트→16비트, 16비트→32비트, 32비트→64비트 등

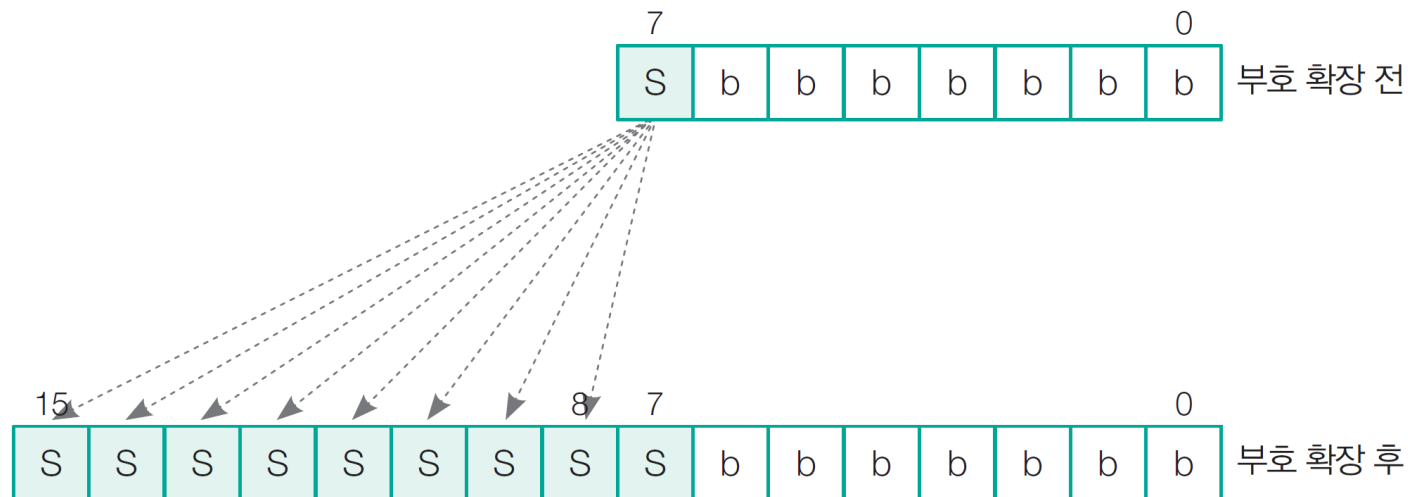


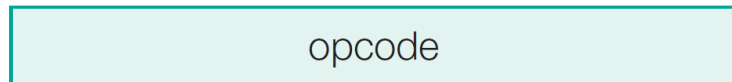
그림 4-8 부호 확장

1 명령어 형식

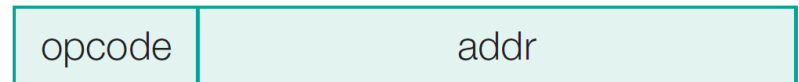
- 연산 코드(opcode), 오퍼랜드(operand), 피연산자 위치, 연산 결과의 저장 위치 등 여러 가지 정보로 구성

❖ 0-주소 명령어

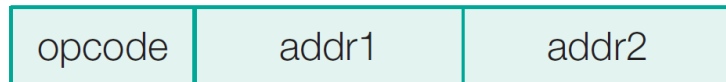
- 연산에 필요한 오퍼랜드 및 결과의 저장 장소가 묵시적으로 지정된 경우 : 스택(stack)을 갖는 구조(PUSH, POP)
- 스택 구조 컴퓨터에서 수식 계산 : 역 표현 (reverse polish)



(a) 0-주소 명령어



(b) 1-주소 명령어



(c) 2-주소 명령어

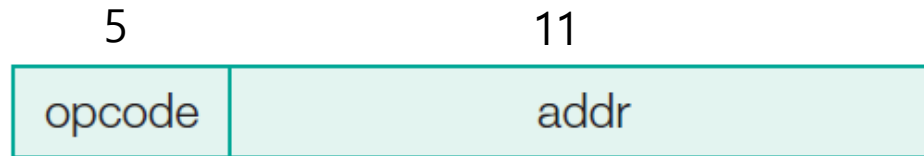


(d) 3-주소 명령어

그림 4-9 명령어 형식

❖ 1-주소 명령어

- 연산 대상이 되는 2개 중 하나만 표현하고 나머지 하나는 묵시적으로 지정: 누산기(AC)
- 기억 장치 내의 데이터와 AC 내의 데이터로 연산
- 연산 결과는 AC에 저장
- 다음은 기억 장치 X번지의 내용과 누산기의 내용을 더하여 결과를 다시 누산기에 저장
$$\text{ADD } X \quad ; \text{AC} \leftarrow \text{AC} + \text{M}[X]$$
- 오퍼랜드 필드의 모든 비트가 주소 지정에 사용: 보다 넓은 영역의 주소 지정
- 명령워드 : 16비트, Opcode: 5비트, 오퍼랜드(addr): 11비트 → 32(=2⁵)가지의 연산 가능, 2048(=2¹¹)개 주소 지정 가능

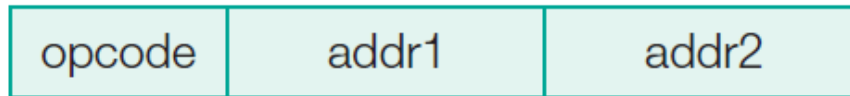


(b) 1-주소 명령어

❖ 2-주소 명령어

- 연산에 필요한 두 오퍼랜드 중 하나가 결과 값 저장
- 레지스터 R1과 R2의 내용을 더하고 그 결과를 레지스터 R1에 저장
- R1 레지스터의 기존 내용은 지워짐

ADD R1, R2 ; $R1 \leftarrow R1 + R2$



(c) 2-주소 명령어

❖ 3-주소 명령어

- 연산에 필요한 오퍼랜드 2개와 결과 값의 저장 장소가 모두 다름
- 레지스터 R2와 R3의 내용을 더하고 그 결과 값을 레지스터 R1에 저장하는 명령어다.
- 연산 후에도 입력 데이터 보존
- 프로그램이 짧아짐
- 명령어 해독 과정이 복잡해짐

ADD R1, R2, R3 ; $R1 \leftarrow R2 + R3$



(d) 3-주소 명령어

04 컴퓨터 명령어

❖ 0-주소, 1-주소, 2-주소, 3-주소 명령을 사용하여 $Z=(B+C) \times A$ 를 구현한 예

- 니모닉(mnemonic)

ADD : 덧셈

MUL : 곱셈

MOV : 데이터 이동(레지스터와 기억 장치 간)

LOAD : 기억 장치에서 데이터를 읽어 누산기에 저장

STOR : AC의 내용을 기억 장치에 저장

0-주소	1-주소	2-주소	3-주소
PUSH B	LOAD B	MOV R1, B	ADD R1, B, C
PUSH C	ADD C	ADD R1, C	MUL Z, A, R1
ADD	MUL A	MUL R1, A	
PUSH A	STOR Z	MOV Z, R1	
MUL			
POP Z			

2 명령어 형식 설계 기준명령어 형식

1. 첫 번째 설계 기준 : 명령어 길이

- 메모리 공간 차지 비율 감소
- 명령어 길이를 최소화하려면 명령어 해독과 실행 시간에 비중을 둬
- 짧은 명령어는 더 빠른 프로세서를 의미: 최신 프로세서는 동시에 여러 개의 명령을 실행하므로 클럭 주기당 명령어를 여러 개 가져오는 것이 중요

2. 두 번째 설계 기준 : 명령어 형식의 공간

- 2^n 개를 연산하는 시스템에서 모든 명령어가 n 비트보다 크다.
- 향후 명령어 세트에 추가할 수 있도록 opcode를 위한 공간을 남겨 두지 않음

3. 세 번째 설계 기준 : 주소 필드의 비트 수

- 8비트 문자를 사용하고, 주기억 장치가 2^{32} 개
- 메모리의 기본 단위
 - 4바이트(32비트)로 해야 한다고 주장하는 팀 : 0, 1, 2, 3, ..., 4,294,967,295인 2^{32} **바이트** 메모리 제안
 - 30비트로 해야 한다고 주장하는 팀: 0, 1, 2, 3, ..., 1,073,741,823인 2^{30} **워드** 메모리 제안

3 확장 opcode

- ❖ 8비트 연산 코드와 24비트 주소를 가진 32비트 명령어
 - 이 명령어는 연산 $2^8(=256)$ 개와 주소 지정 $2^{24}(=16M)$ 개 메모리
- ❖ 7비트 연산 코드와 25비트 주소를 가진 32비트 명령어
 - 명령어 개수는 절반인 128개이지만 메모리는 2배인 $2^{25}(=32M)$ 개
- ❖ 9비트 연산 코드와 23비트 주소일 때
 - 명령어 개수는 2배(256), 주소는 절반인 $2^{23}(=8M)$ 개 메모리

04 컴퓨터 명령어

❖ 명령어 길이 16비트, 오퍼랜드 4비트 시스템

- 모든 산술 연산이 레지스터(따라서 4비트 레지스터 주소) 16개에서 수행되는 시스템
- 한 가지 설계 방법은 4비트 연산 코드와 오퍼랜드가 3개 있는 3-주소 명령어를 16개 가지는 것

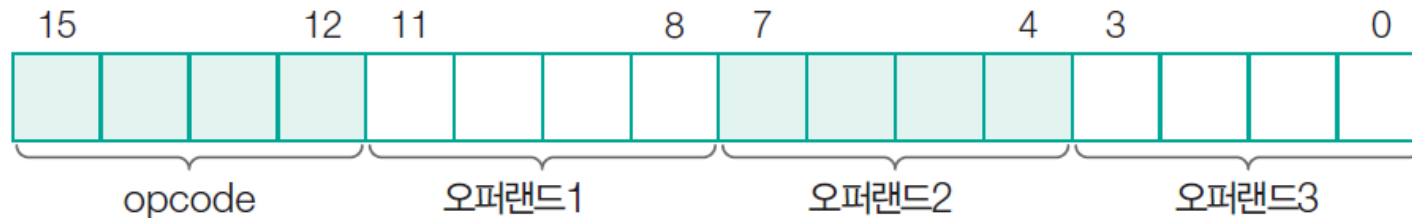


그림 4-10 3-주소 명령어 구조

04 컴퓨터 명령어

1. 3-주소 명령어는 14개, 2-주소 명령어는 30개, 1-주소 명령어는 31개, 0-주소 명령어는 16개가 필요하다면
 - [그림 4-11]과 같이 3-주소 명령어로 opcode 0~13을 사용
2. opcode 14~15를 다르게 해석
 - opcode 14와 opcode 15는 opcode 비트가 12~15(4비트)가 아닌 8~15(8비트)를 의미
 - 비트 0~3과 비트 4~7은 오퍼랜드(주소)를 2개 지정
 - 2-주소 명령어 30개는 왼쪽 4비트가 1110일 때, 비트 8~11은 0000에서 1111까지의 숫자를 지정하고, 왼쪽 4비트가 1111일 때는 비트 8~11은 0000에서 1101까지의 숫자 지정

04 컴퓨터 명령어

3. 가장 왼쪽 4비트가 1111이고, 비트 8~11이 1110 또는 1111인 1주소 명령어
 - 비트 4~15가 opcode(12비트)임
 - 12비트인 opcode가 32개가 가능하지만 12비트 모두가 1인 1111 1111 1111은 또 다른 명령어로 지정
4. 상위 12비트가 모두 1인 명령어 16개를 0-주소 명령어로 지정 ⇒ 이 방법에서는 opcode가 계속해서 길어짐
 - 3-주소 명령어는 4비트 opcode
 - 2-주소 명령어는 8비트 opcode
 - 1-주소 명령어는 12비트 opcode
 - 0-주소 명령어는 16비트 opcode

04 컴퓨터 명령어

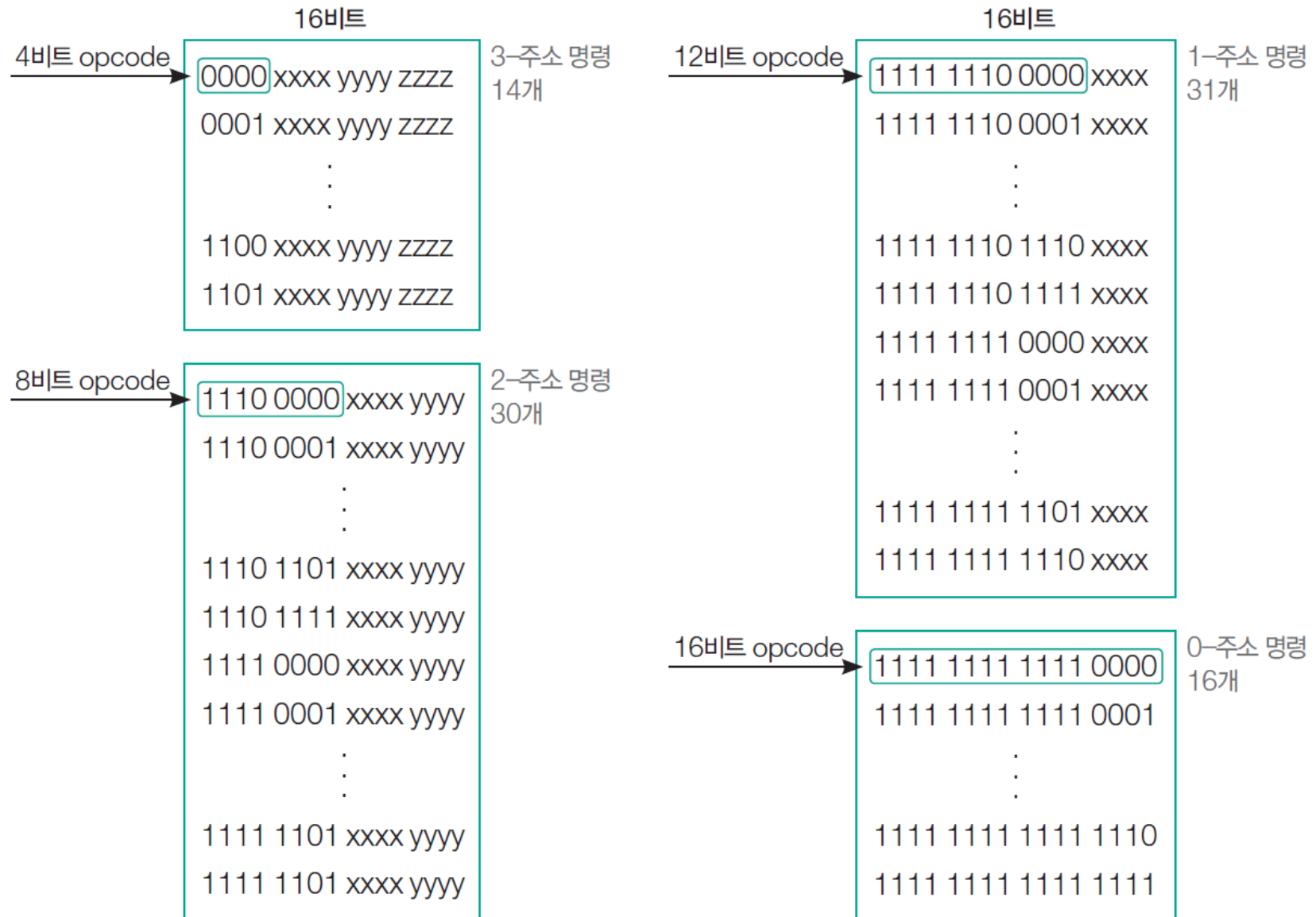


그림 4-11 명령어 설계의 예

04 컴퓨터 명령어

- 확장 opcode는 opcode 공간과 다른 정보 공간 간의 균형을 보여 줌
- opcode를 확장하는 것이 예처럼 명확하고 규칙적이지 않음
- 다양한 크기의 opcode를 사용하는 기능은 두 가지 방법 중 하나로 활용
 - 첫째, 명령어 길이를 일정하게 유지 가능
 - 둘째, 일반 명령어는 가장 짧은 opcode를, 잘 사용되지 않는 명령어는 가장 긴 opcode를 선택
- 장점 : 평균 명령어 길이 최소화
- 단점 : 다양한 크기의 명령어를 초래하여 신속한 해독이 불가하거나 또 다른 역효과

4 코어 i7 명령어 형식

- 코어 i7 명령어 형식은 매우 복잡하고 불규칙
- 가변 길이 필드가 최대 6개 있으며 그 중 5개는 선택적
- CPU 구조가 여러 세대에 걸쳐 발전했고 초기의 잘못된 선택 때문
- 이전 버전과 호환성 고려로 되돌릴 수 없는 결과 발생

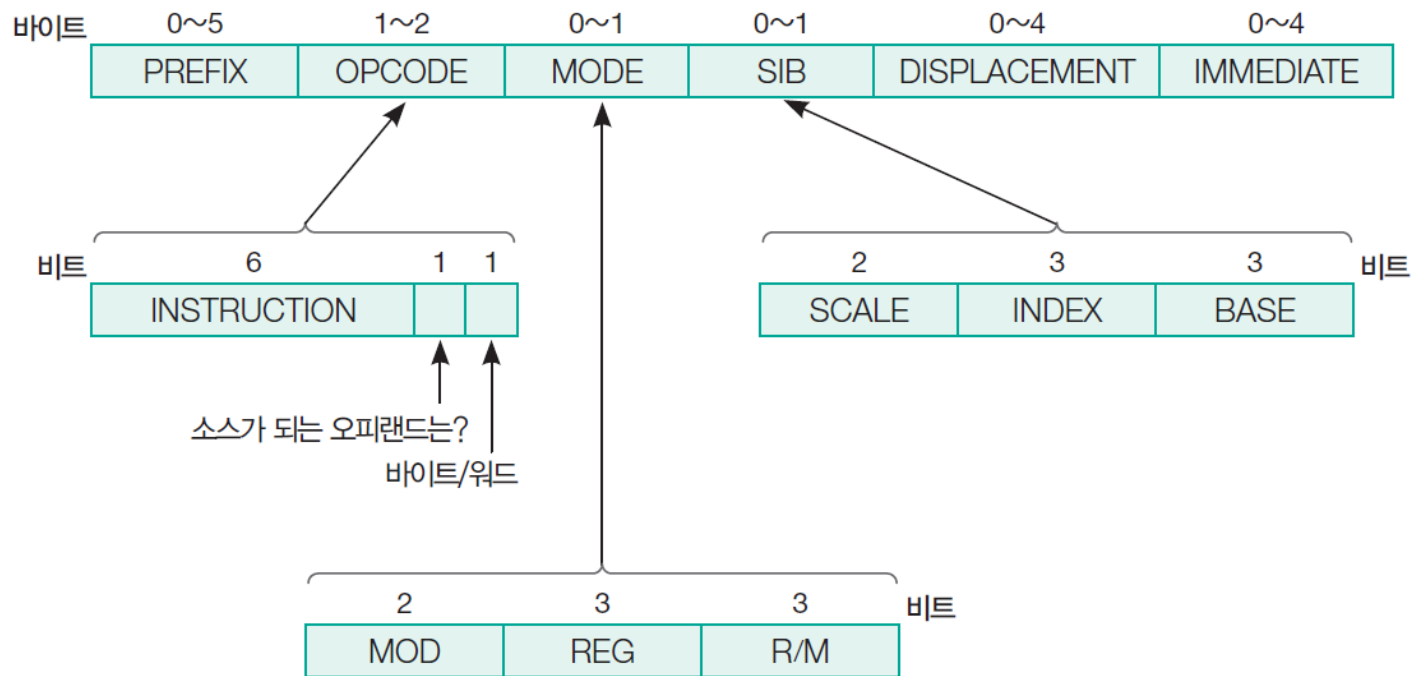
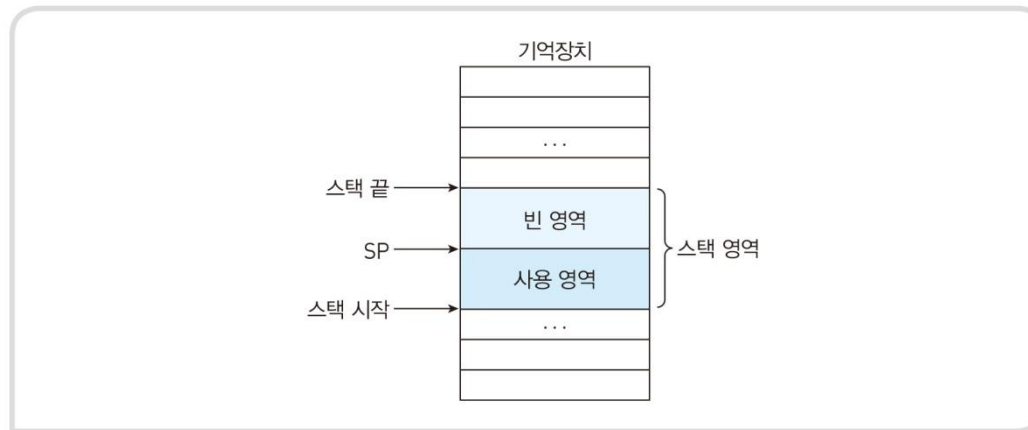


그림 4-12 코어 i7 명령어 형식

스택 포인터

□ 스택(Stack)

- LIFO(Last-In, First-Out)
- 기억장치의 특정 영역을 스택 영역으로 활용
- SP(Stack pointer): point to the top of stack



□ 스택 동작

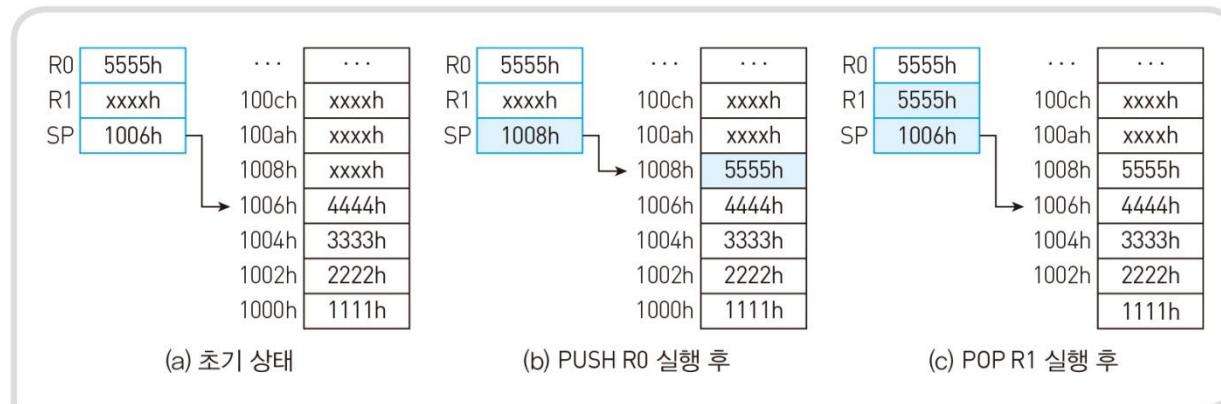
- PUSH: 스택에 데이터 저장
- POP: 스택에서 데이터 제거
- 스택에 저장되는 데이터 크기 = 레지스터 크기 = 단어(word) 크기

❑ PUSH

- 명령어: PUSH operand // operand를 스택에 저장
 - $SP \leftarrow SP + [\text{단어 크기}]$
 - $\text{Mem}[SP] \leftarrow \text{operand}$
- 마이크로오퍼레이션
 - $SP \leftarrow SP + [\text{단어의 크기}]$ // SP 증가
 - $MAR \leftarrow SP, MBR \leftarrow \text{오퍼랜드}$ // MAR, MBR 설정
 - $\text{Mem}[MAR] \leftarrow MBR$ // 기억장치에 저장

POP

- 명령어: POP operand // 스택에서 제거한 데이터를 operand에 저장
 - $\text{Operand} \leftarrow \text{Mem}[\text{SP}]$
 - $\text{SP} \leftarrow \text{SP} - [\text{단어 크기}]$
- 마이크로오퍼레이션
 - $\text{MAR} \leftarrow \text{SP}$ // MAR 설정
 - $\text{MBR} \leftarrow \text{Mem}[\text{MAR}]$ // 기억장치 읽기
 - $\text{SP} \leftarrow \text{SP} - [\text{단어의 크기}]$, 오퍼랜드 $\leftarrow \text{MBR}$ // SP 감소, 레지스터 적재



5 명령어 종류

□ ISA(Instruction Set Architecture) 컴퓨터의 명령어 : 6개의 그룹

- 컴퓨터에는 이전 모델과 호환성을 위해 추가된 몇 가지 특이한 명령어
- 설계자의 좋은 아이디어 추가
- 특정 기관에서 비용을 지불하고 명령어 추가

1. 데이터 이동 명령
2. 2항 연산
3. 단항 연산
4. 비교와 조건 분기 명령
5. 프로시저 호출 명령
6. 루프 제어 명령

□ 데이터 이동 명령

- 가장 기본이 되는 작업 : 원본과 동일한 새로운 객체를 만드는 복사
- 원래 위치에 그대로 두고 다른 장소에 복사본 생성

❖ 데이터를 복사하는 이유

1. 변수에 값 할당 : $A=B$ 는 메모리 주소 B의 값(데이터)을 A 장소로 복사한다는 의미다.
2. 데이터의 효율적인 액세스 및 사용: 메모리와 레지스터 간에 데이터를 이동하여 프로그램 실행을 효율적으로 수행하기 위해서다.
 - LOAD 명령 : 메모리에서 레지스터로 이동
 - STORE 명령 : 레지스터에서 메모리로 이동
 - MOVE 명령 : 하나의 레지스터에서 다른 레지스터로 이동단, 메모리 간 이동은 일반적으로 사용하지 않음

□ 2항 연산

- 2항 연산은 오퍼랜드 2개를 결합하여 결과 생성
- 산술 연산(덧셈, 뺄셈, 곱셈, 나눗셈) 및 논리 연산(AND, OR, XOR, NOR, NAND 등)

❖ AND 연산

- 워드에서 특정 비트를 추출하는 용도로 사용
- 예를 들어 8비트 문자가 4개 저장된 32비트 워드에서 3번째 문자(11010110) 만 남기고 나머지 세 문자를 제거하고 오른쪽으로 8비트 시프트
 - 먼저 3번째 문자(11010110)를 추출: 마스크 상수와 AND 연산
 - 단어의 오른쪽 끝에 추출할 문자를 분리: 오른쪽으로 8비트 시프트

10110011	11000101	11010110	10000101	A
00000000	00000000	11111111	00000000	B (마스크)
00000000	00000000	11010110	00000000	A AND B
00000000	00000000	00000000	11010110	A >> 8

04 컴퓨터 명령어

- OR를 마스크 연산과 함께 사용하여 원하는 위치에 값 교체 : 예를 들어 상위 24비트는 그대로 두고 하위 8비트 변경
 - 필요 없는 8비트를 마스크 처리하여 없애고 새 문자를 OR 연산

10111110	11100011	10101010	01010101	A
11111111	11111111	11111111	00000000	B (마스크)
10111110	11100011	10101010	00000000	A AND B
00000000	00000000	00000000	10001110	C
10111110	11100011	10101010	10001110	(A AND B) OR C

- AND 연산은 1을 제거하는 마스크 연산
- OR 연산은 1을 삽입하는 연산
- XOR 연산은 대칭적이며, 어떤 값을 1로 XOR하면 반대(대칭) 값을 생성
 - 0과 1에 대칭적이라는 것은 때로 유용: 의사 난수 생성에 사용

□ 단항 연산

- 단항 연산 : 오퍼랜드가 1개, 결과도 1개
- 2항 연산보다 명령이 짧지만, 명령에 다른 정보를 지정해야 할 때가 많음

❖ 시프트(shift)

- 비트를 왼쪽이나 오른쪽으로 이동하는 작업
- 워드의 끝부분에서 비트 손실 발생

❖ 회전(rotation)

- 한쪽 끝에서 밀린 비트가 다른 쪽 끝에서 다시 나타나는 이동
- 시프트와 회전의 차이

00000000	00000000	00000000	01110011	A
00000000	00000000	00000000	00011100	A를 오른쪽으로 2비트 시프트
11000000	00000000	00000000	00011100	A를 오른쪽으로 2비트 회전

04 컴퓨터 명령어

❖ 오른쪽 시프트는 흔히 부호와 함께 수행

- 즉, 워드의 MSB 부호는 그대로 유지한 채 오른쪽으로 시프트
- 특히 음수인 경우 그대로 음수 유지
- 2비트 오른쪽 시프트 예

11111001	11100011	01101011	10110000	A
00111110	01111000	11011010	11011100	A를 부호 없이 2비트 오른쪽 시프트
11111110	01111000	11011010	11011100	A를 부호와 같이 2비트 오른쪽으로 시프트

❖ 시프트의 중요한 용도

- 2의 제곱수를 곱하는 것과 나누는 것
- 양의 정수가 왼쪽으로 k 비트 시프트되었을 때 오버플로가 발생하지 않았다면 원래 수에 2^k 을 곱한 것
- 양의 정수를 오른쪽으로 k 비트 시프트했을 때 결과는 원래 수를 2^k 로 나눈 것

❖ 시프트는 특정 산술 연산의 속도를 높이는 데 사용

- 예를 들어 어떤 양의 정수 n 에 대해 $24 \times n$ 을 계산
- $24 \times n = (16+8) \times n = 2^4 \times n + 2^3 \times n$ 이므로,
- n 을 4비트 왼쪽으로 시프트하면 $16 \times n$ 이 되고,
- n 을 왼쪽으로 3비트 시프트하면 $8 \times n$
- 두 값의 합이 $24 \times n$
- 시프트 두 번 과 덧셈으로 계산되므로 곱셈보다 빠름

04 컴퓨터 명령어

❖ 음수를 시프트하면 다른 결과가 됨

- -1의 2의 보수: 부호 확장을 사용하여 오른쪽으로 6비트 시프트하면 그대로 -1

11111111	11111111	11111111	11111111	2의 보수로 표시된 -1
11111111	11111111	11111111	11111111	-1을 오른쪽을 6비트 시프트 = -1

- -1은 더 이상 오른쪽으로 시프트할 수 없음
- 왼쪽 시프트는 한 비트씩 이동할 때마다 2를 곱한 결과

❖ 회전 연산은 워드의 모든 비트 테스트할 경우

- 한 번에 1비트씩 워드를 회전하면 각 비트를 MSB에 순서대로 배치하여 쉽게 테스트 가능
- 모든 비트가 테스트된 후에는 워드가 원래 값으로 복원
- 또는 레지스터 값을 직렬화할 때도 유용함

❖ 다른 단항 연산은 INC(1 증가), DEC(1 감소), NEG(2의 보수), NOT(비트 반전) 등

- NEG는 비트를 반전한 후 1을 더한 2의 보수
- NOT은 단순한 비트 반전으로 1의 보수

□ 비교와 조건 분기 명령

❖ 조건이 충족되면 특정 메모리 주소로 분기

- 검사에 사용되는 일반적인 방법 : 특정 비트가 0인지 확인
- 음수인지 알아보기 위해 부호 비트 검사 : 1이면 분기

❖ 상태 코드 비트

- 특정 조건 표시

❖ 오버플로 비트:

- 산술 연산의 결과 데이터가 표현 범위를 벗어났을 때 1로 설정
- 오버플로 발생 : 에러 루틴 및 수정 조치

❖ 캐리 비트

- 맨 왼쪽 비트에서 데이터가 넘칠 때 세트
- 가장 왼쪽 비트의 캐리는 정상 연산에서도 발생하므로 오버플로와 혼동하면 안됨
- 다중 비트 연산: 정수가 워드 2개 이상으로 표현되는 경우 연산을 수행하려면 캐리 비트 점검

❖ 0 검사

- 루프 및 기타 여러 용도 유용
- 1이 하나라도 들어 있는지를 나타내는 비트를 제공하기 위해 OR 회로를 사용
- Z 비트는 ALU의 모든 출력 비트를 OR한 후 반전

❖ 두 수의 비교

- 두 워드나 문자 비교: 같은지, 아닌지 또는 그렇지 않은 경우 어떤 단어가 더 큰지 확인
- 정렬(sorting)할 때 중요
- 주소 3개 필요 : 2개는 데이터 항목, 1개는 조건이 참일 경우 분기할 주소
- 두 정수가 같은지 비교하려면 XOR 사용
- 어떤 수가 큰지 작은지를 비교: 뺄셈을 사용 가능하지만, 아주 큰 양수와 음수를 비교할 때 두 수를 뺄셈하면 그 결과는 오버플로됨
- 비교 명령 : 테스트 충족 여부 결정 및 오버플로가 발생하지 않는 정확한 답 반환 해야 함

□ 프로시저 호출(procedure call) 명령

- 특정 작업을 수행하는 명령 그룹: 프로그램 내 어디서든 호출 가능
- 어셈블리에서는 서브루틴(subroutine), C 언어에서는 함수(function), 자바에서는 메서드(method)라고 함
- 프로시저가 작업을 완료하면 호출 명령 바로 다음 명령으로 복귀
 - 복귀 주소를 프로시저에 전송하거나 복귀할 때 찾을 수 있도록 어딘가에 저장
 - 복귀 주소: 메모리, 레지스터, 스택 세 군데에 배치 가능
 - 프로시저는 여러 번 호출 가능하므로 프로시저 여러 개가 직접 또는 간접적으로 다중 호출되어도 프로그램이 정상 순서로 수행되어야 함
 - 프로시저를 반복할 경우, 복귀 주소를 호출할 때마다 다른 위치에 두어야 함
 - 프로시저 호출 명령이 복귀 주소와 함께하는 가장 좋은 방법은 스택
 - 프로시저가 끝나면 스택에서 반환 주소를 꺼내 프로그램 카운터 저장
 - 프로시저가 자기 자신 호출 기능 : 재귀(recursion), 스택을 사용하면 재귀 기능 정상 동작
 - 복귀 주소는 이전 복귀 주소가 파손되지 않도록 자동 저장

□ 루프 제어 명령

- 명령 그룹을 정해진 횟수만큼 실행해야 하는 경우
- 루프(loop)를 통해 매번 일정하게 증가 시 또는 감소시키는 카운터 소유
 - 루프를 반복할 때마다 종료 조건을 만족하는지 검사
 - 보통 루프 밖에서 카운터를 초기화한 후 루프 코드 실행 시작
 - 루프의 마지막 명령에서 카운터 업데이트
 - 종료 조건을 아직 만족하지 않으면 루프의 첫 번째 명령으로 분기
 - 반면 종료 조건이 만족되면 루프 종료, 루프를 벗어난 첫 번째 명령이 실행

□ 루프 제어 명령

- 종점 테스트(test-at-the-end 또는 post-test)
 - 조건이 루프의 끝에서 이루어지므로 루프가 무조건 한 번 이상 실행
- 종료 검사를 사전에 수행하도록 루프 구성: 루프의 시작 시점에서 검사
 - 처음부터 조건 만족 : 루프에 포함된 내용을 한 번도 실행하지 않음
- C 언어의 for 처럼 정해진 횟수만큼 반복 루프 가능
- 모든 루프는 한 가지로 표현 가능
 - 용도에 맞는 형태로 사용

□ 입출력 명령

- 입출력 장치 다양한 만큼 입출력 명령도 다양
- 개인용 컴퓨터 : 세 가지 입출력 방식 사용
 - 프로그래밍에 의한 입출력
 - 인터럽트 구동interrupt-driven 입출력
 - DMA 입출력

❖ 프로그래밍에 의한 입출력

- 가장 단순함
- 임베디드 시스템 또는 실시간 시스템 같은 저사양 마이크로프로세서에서 일반적으로 사용
- 주요 단점 : 장치가 준비되기를 기다리는 긴 시간을 CPU가 낭비하게 됨
 - 사용 대기(busy waiting)라 함
 - CPU가 할 일이 하나밖에 없다면 문제되지 않음
 - 단, 여러 개의 이벤트 동시 모니터링할 경우 낭비되므로 다른 입출력 방법이 적용

❖ 인터럽트 구동 입출력

- 프로세서가 입출력 장치에 작업을 지시하고 완료되면 인터럽트를 생성하도록 명령
- 장치 레지스터에 인터럽트 활성화 비트를 설정 : 입출력이 완료되면 하드웨어가 신호를 제공하도록 요청
- 프로그래밍 입출력보다 개선: 완벽하지 않음
 - 전송된 모든 문자에 인터럽트가 필요하므로 처리 비용이 많이 듦
 - 인터럽트의 많은 부분을 제거하는 방법이 요구됨

❖ DMA(Direct Memory Access) 입출력

- 버스에 직접 액세스할 수 있는 방법: 시스템에 DMA 제어기 추가
- DMA 칩은 내부에 레지스터 최소 4개 보유 : 프로세서에서 실행되는 소프트웨어로 로드 가능
 1. 첫 번째는 읽거나 쓸 메모리 주소 포함
 2. 두 번째는 얼마나 많은 바이트(또는 워드)가 전송되는지 계산
 3. 세 번째는 사용할 장치 번호 또는 입출력 공간 주소를 지정
 4. 네 번째는 입출력 장치에서 데이터를 읽거나 쓰는 여부를 지정
- 프로세서 입출력의 부담을 크게 덜어 줌 : 여전히 완전히 자유롭지 못함
- 디스크 같은 고속 장치가 DMA로 실행되는 경우 메모리 참조 및 장치 참조를 위한 버스 사이클이 많이 필요: 이 사이클 동안 CPU는 대기(입출력 장치는 종종 지연을 용인할 수 없으므로 DMA는 항상 CPU보다 높은 버스 우선순위를 가짐)
- **사이클 스틸링**(cycle stealing) : DMA 제어기가 CPU에서 버스 사이클을 제거
 - 사이클 스틸링으로 인한 이득이 인터럽트로 인한 손실 보다 큼

Summary

- 프로세서 내의 레지스터의 종류와 용도를 이해
- 프로세서 명령어 형식의 종류를 구분하고 명령의 동작을 이해