컴퓨터 명령어

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)

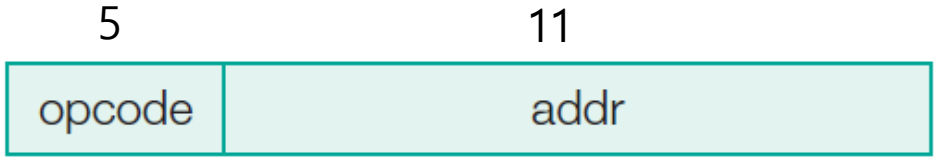
그림 4-9 명령어 형식

보충설명 : Stack(스택) , reverse polish(역 폴리쉬)

❖ 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

  ADD X ; AC ← AC + M[X]
- 오퍼랜드 필드의 모든 비트가 주소 지정에 사용: 보다 넓은 영역의 주소 지정
- 명령워드 : 16비트, Opcode: 5비트, 오퍼랜드(addr): 11비트 ➔ 32(=2^5)가지의 연산 가능, 2048(=2^{11})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 ← 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 ← R2 + R3

(d) 3–주소 명령어

04 컴퓨터 명령어

❖ 0-주소, 1-주소, 2-주소, 3-주소 명령을 사용하여 Z=(B+C)×A를 구현한 예

- 니모닉(mnemonic) : 기계어로 이뤄진 명령어를 상징적인 기호로 변환한 것

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			

Mnemonic :어떤 것을 기억하는 데 쉽게 하도록 도움을 주는 것

2 명령어 형식 설계 기준명령어 형식

1. 첫 번째 설계 기준 : 명령어 길이

- 짧은 명령어는 긴 명령어에 비해 프로그램 실행 시 메모리 공간 차지 비율 감소
- 명령어 길이를 최소화하려면 명령어 해독과 실행 시간에 비중을 둠(명령어 길이를 최소화하면 해독은 어렵고 실행시간은 줄어듬)
- 짧은 명령어는 더 빠른 프로세서를 의미: 최신 프로세서는 동시에 여러 개의 명령을 실행하므로 클록 주기당 명령어를 여러 개 가져오는 것이 중요

2. 두 번째 설계 기준 : 명령어 형식의 공간

- 2^n개를 연산하는 시스템에서 모든 명령어가 n비트보다 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

❖ 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

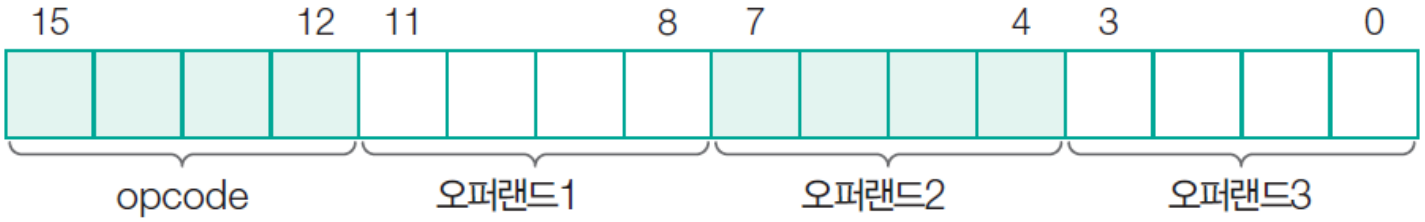


그림 4-10 3-주소 명령어 구조

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

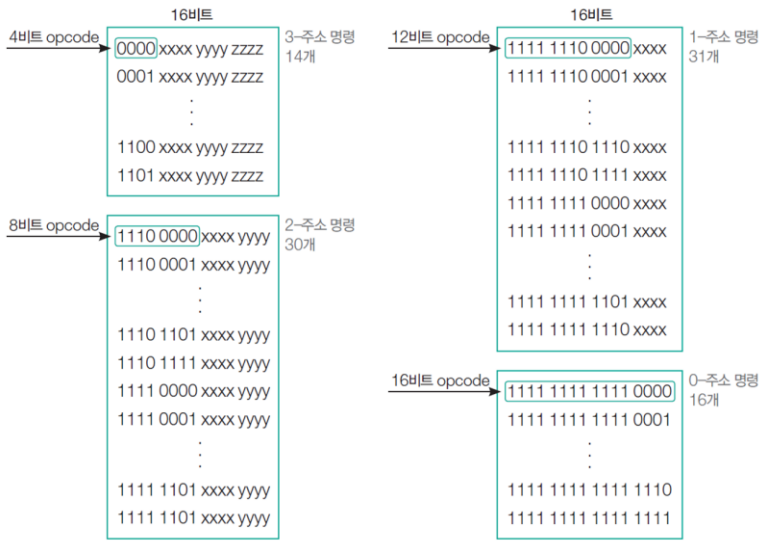


그림 4-11 명령어 설계의 예

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

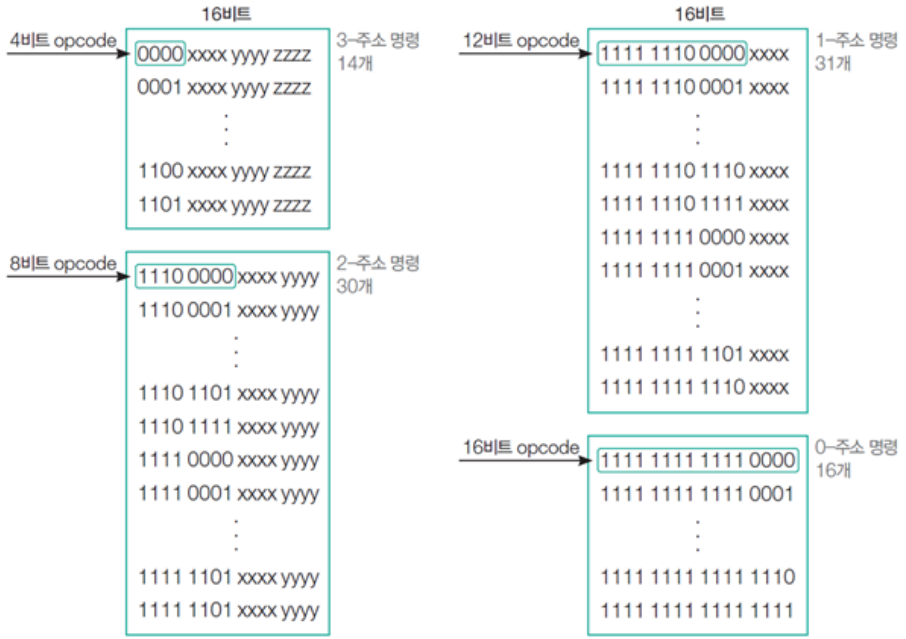


그림 4-11 명령어 설계의 예

04 컴퓨터 명령어

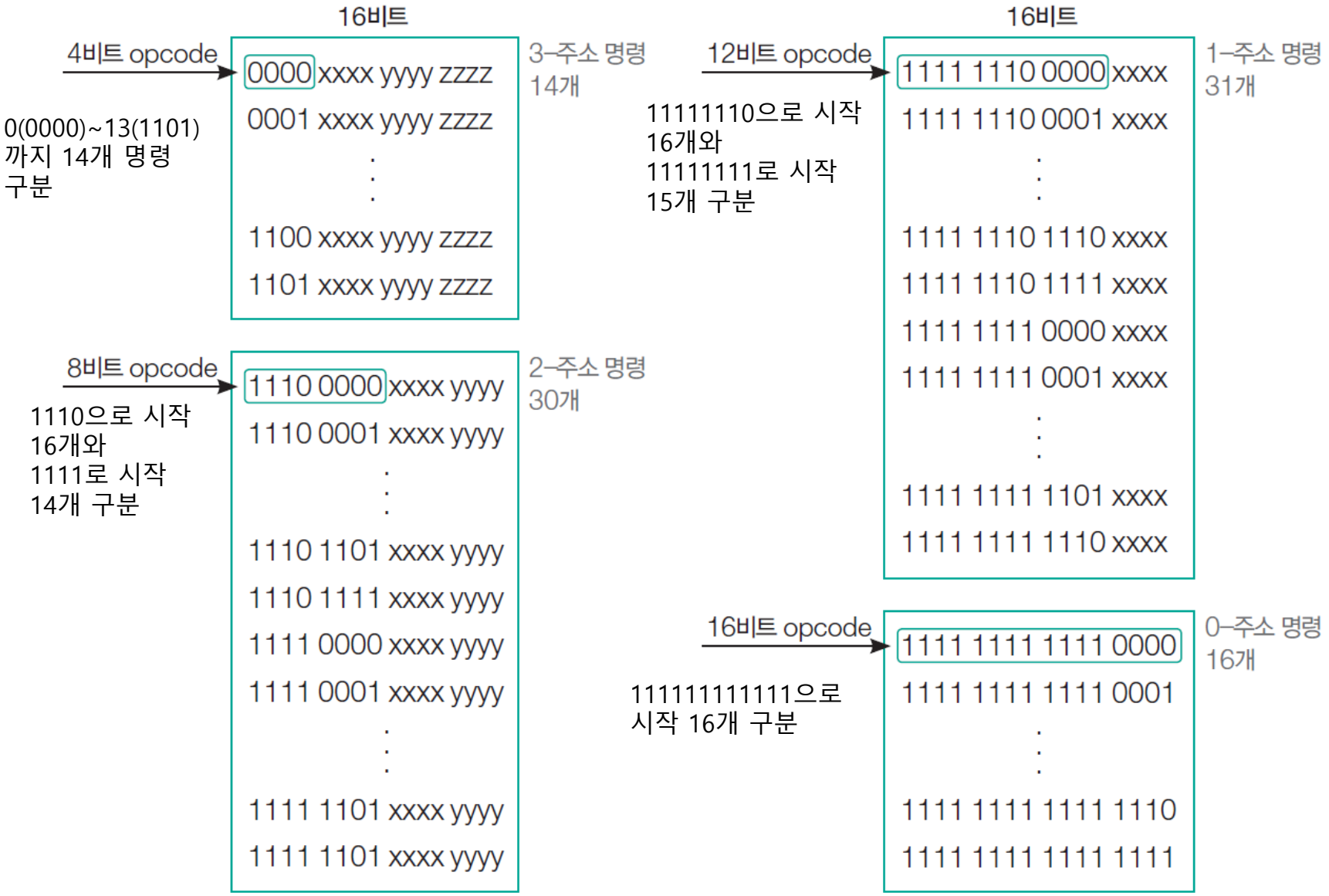


그림 4-11 명령어 설계의 예

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

수고하셨습니다!