

Microprocessor (W1)

- Basics of Microprocessor -

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

Contents

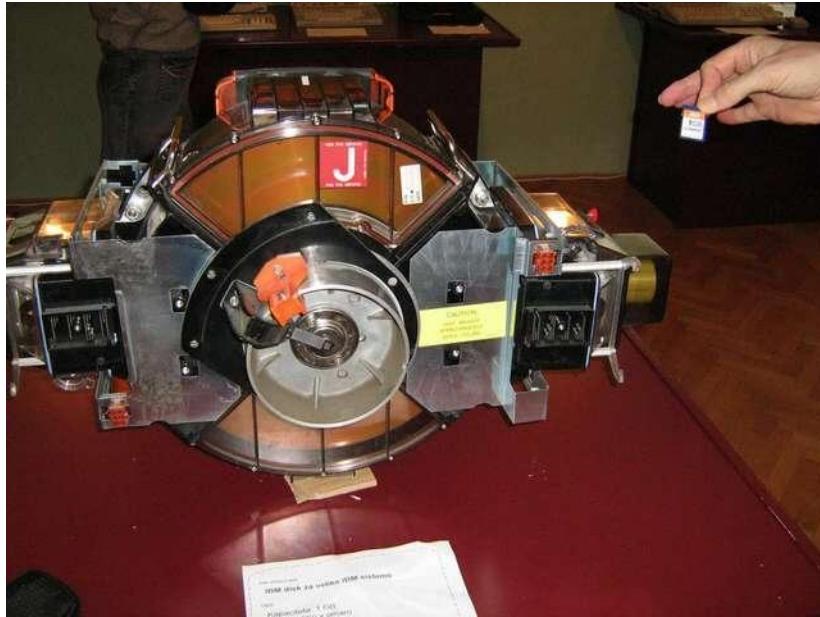
학습목표

- 컴퓨터 하드웨어의 구성 요소와 소프트웨어의 개념을 이해할 수 있다.
- 컴퓨터에서 사용하는 고급 언어와 기계어의 관계를 살펴본다.
- 새로운 하드웨어 부품의 출현에 따른 컴퓨터의 세대별 발전 과정을 알아본다.
- 여러 측면에서 컴퓨터를 분류하고 설명할 수 있다.
- 폰 노이만 구조와 하버드 구조를 비교하여 설명할 수 있다.

내용

- 01 컴퓨터 시스템의 구성
- 02 컴퓨터의 역사
- 03 컴퓨터의 분류
- 04 폰 노이만, 비 폰 노이만, 하버드 구조

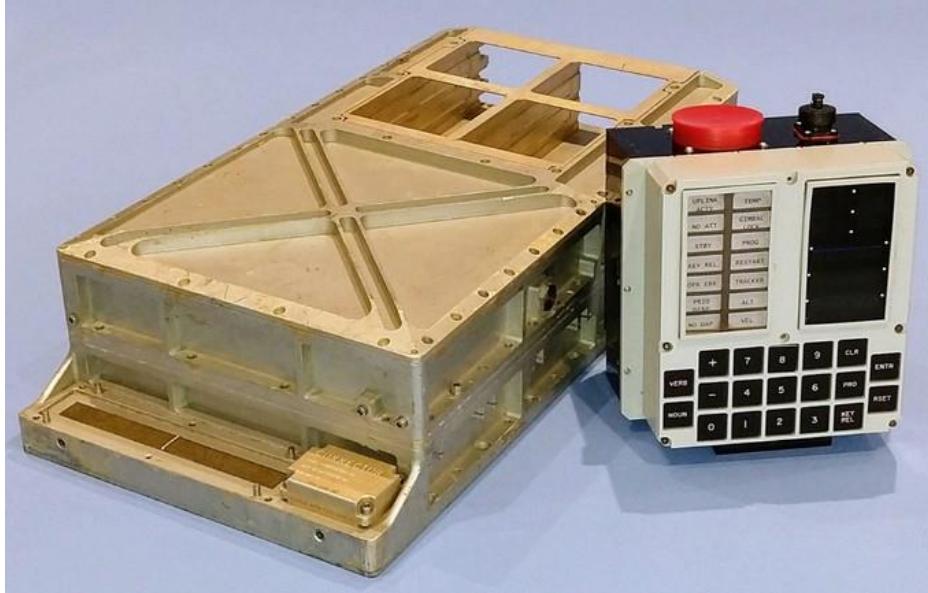
과거와 현재



<1980년의 1GB 하드디스크>



과거와 현재



아폴로 11호에서 달 착륙을 유도하는 데 쓰인 컴퓨터의 클럭 스피드는 1.024MHz였다. 램은 정보의 저장 단위인 한 비트에 한 문자를 넣는다고 가정했을 때 4KB였고, 저장 공간은 72KB였다. 아폴로 11호에는 유도 컴퓨터 2대와 발사체 발사용 컴퓨터, 탈출 유도 시스템 등 4대의 컴퓨터가 쓰인 것으로 알려져 있다.

미국 반도체기업 사이프러스의 'CYPD4225' 마이크로칩을 쓰고 있는 걸로 나타났다. 이 칩의 계산 속도를 나타내는 클럭 스피드는 48MHz다. 램은 8KB를, 저장 공간은 128KB로 나타났다. 이 충전기의 가격은 54.99달러(약 6만 4800원)이다.

<https://www.epnc.co.kr/news/articleView.html?idxno=90911>
<https://www.dongascience.com/news.php?idx=34277>

01 컴퓨터 시스템의 구성

□ 컴퓨터의 기본 구성

- **하드웨어** : 컴퓨터에서 각종 정보를 입력하여 처리하고 저장하는 동작이 실제 일어나게 해 주는 물리적인 실체
- **소프트웨어** : 정보 처리의 종류를 지정하고 정보의 이동 방향을 결정하는 동작이 일어나는 시간을 지정해 주는 명령(command)들의 집합(프로그램)

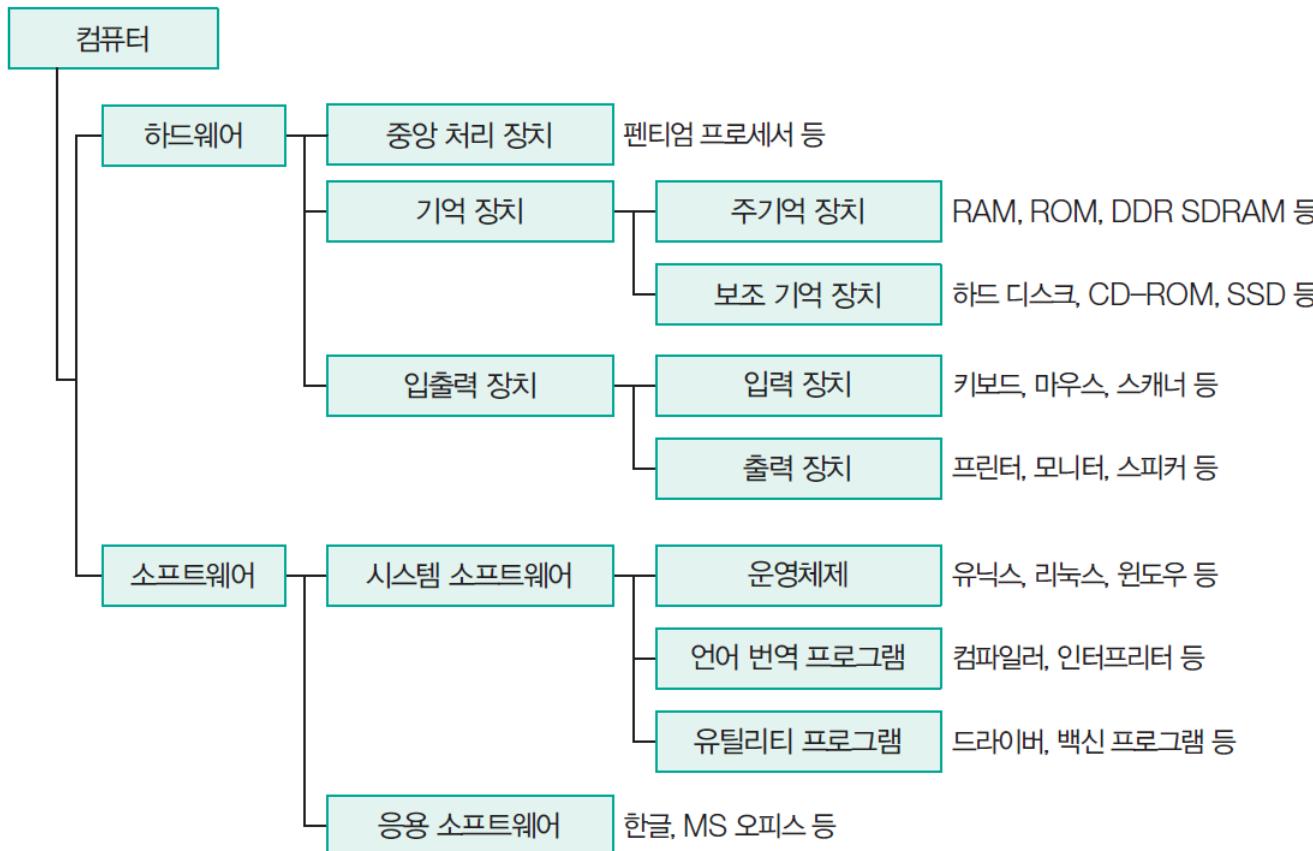


그림 1-1 컴퓨터의 기본 구성

01 컴퓨터 시스템의 구성

1 하드웨어

- 컴퓨터 하드웨어는 중앙 처리 장치, 주기억 장치, 보조 기억 장치, 입력 장치, 출력 장치, 시스템 버스로 구성된다.
- 컴퓨터의 기능을 수행하기 위해 각 구성 요소들은 시스템 버스를 통해 상호 연결되어 있다.

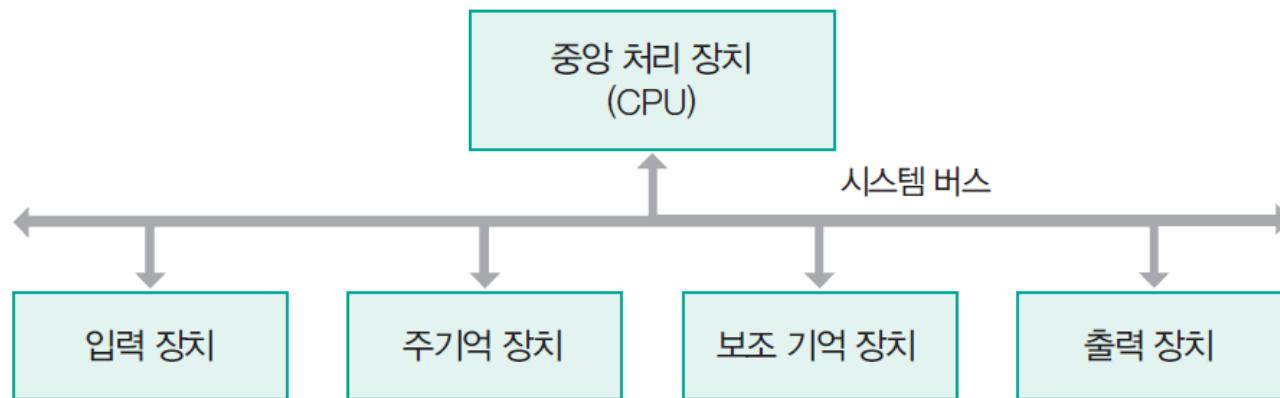


그림 1-2 컴퓨터의 하드웨어 구성

01 컴퓨터 시스템의 구성

□ 중앙 처리 장치

- **중앙 처리 장치**(Central Processing Unit, CPU) : 사실상 컴퓨터의 특성을 결정하며, 컴퓨터의 핵심 기능인 프로그램 실행과 데이터 처리를 담당한다.
- 중앙 처리 장치를 **프로세서**(processor) 또는 **マイクロプロセッサー**(microprocessor)라고도 부른다.
- **산술 논리 연산 장치**(Arithmetic and Logic Unit, ALU) : 산술 연산 논리 연산, 보수 연산, 시프트 shift 연산을 수행한다.
- **제어 장치**(Control Unit, CU) : 프로그램의 명령어를 해독하여 명령어 실행에 필요한 제어 신호를 발생시키고 컴퓨터의 모든 장치를 제어한다.
- **레지스터**(register) : 중앙 처리 장치 내부에 있는 데이터를 일시적으로 보관하는 임시기억 장치로, 프로그램 실행 중에 사용되며 고속으로 액세스할 수 있다.

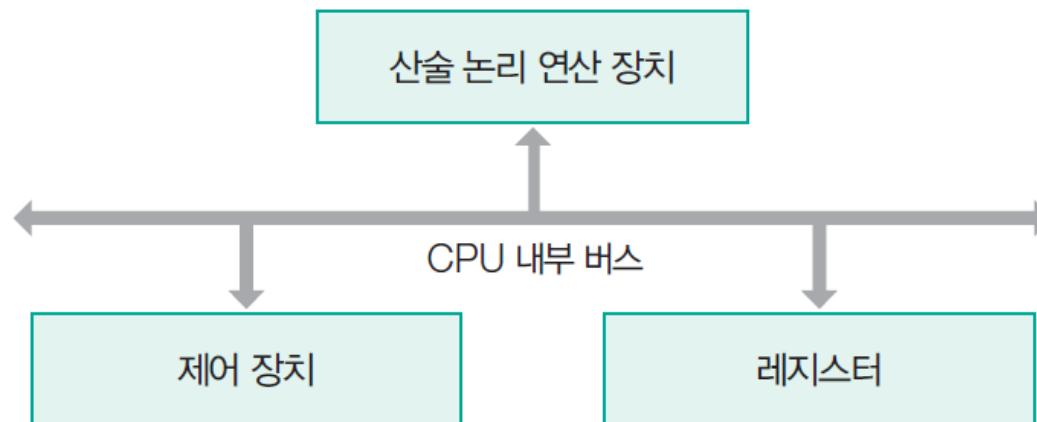


그림 1-3 중앙 처리 장치 내부 구조

01 컴퓨터 시스템의 구성

□ 기억 장치

- **주기억 장치**(main memory) : 반도체 칩으로 구성되어 고속으로 액세스가 가능하지만 고가다. 또한 프로그램 실행 중에 일시적으로만 사용되는 휘발성 메모리다.
- **보조 기억 장치**(auxiliary storage unit) : 하드 디스크나 SSD, CD-ROM 같은 비휘발성 메모리이며, 저장 밀도가 높고 저가이지만 속도가 느리다. 중앙 처리 장치에 당장 필요하지 않은 많은 양의 데이터나 프로그램을 저장하는 장치다.

□ 입출력 장치

- **입력 장치**(input device) : 데이터를 전자적인 2진 형태로 변환하여 컴퓨터 내부로 전달한다. 키보드, 마우스가 있다.
- **출력 장치**(output device) : 중앙 처리 장치가 처리한 전자적인 형태의 데이터를 사람이 이해할 수 있는 데이터로 변환하여 출력한다. LCD 모니터, 프린터, 스피커가 있다.
- 입력 장치와 출력 장치를 합하여 **입출력 장치**(Input/Output device, I/O device)라고도 한다.

01 컴퓨터 시스템의 구성

□ 시스템 버스

- **시스템 버스**(system bus) : 중앙 처리 장치와 기억 장치 및 입출력 장치 사이에 정보를 교환하는 통로이며 주소 버스, 데이터 버스, 제어 버스가 있다.

| | |
|------------------------|---|
| 주소 버스 (address bus) | <ul style="list-style-type: none">• 기억 장치나 입출력 장치를 지정하는 주소 정보를 전송하는 신호 선들의 집합이다.• 단방향이다. |
| 데이터 버스 (data bus) | <ul style="list-style-type: none">• 기억 장치나 입출력 장치 사이에 데이터를 전송하기 위한 신호선들의 집합이다.• 데이터선의 수는 중앙 처리 장치가 한 번에 전송할 수 있는 데이터 비트의 수를 결정한다.• 양방향 전송이 가능해야 한다. |
| 제어 버스 (control bus) | <ul style="list-style-type: none">• 중앙 처리 장치가 시스템 내의 각종 요소의 동작을 제어하는 데 필요한 신호선들의 집합이다.• 기억 장치 읽기와 쓰기 신호, 입출력 장치 읽기와 쓰기 신호 등이 있다.• 단방향이다. |

01 컴퓨터 시스템의 구성

□ 컴퓨터 시스템의 구성

- 주기억 장치는 중앙 처리 장치와 시스템 버스를 통해 연결된다.
- 보조 기억 장치나 입출력 장치는 속도가 느리기 때문에 인터페이스 회로나 제어기를 통해 중앙 처리 장치와 연결되어야 한다.

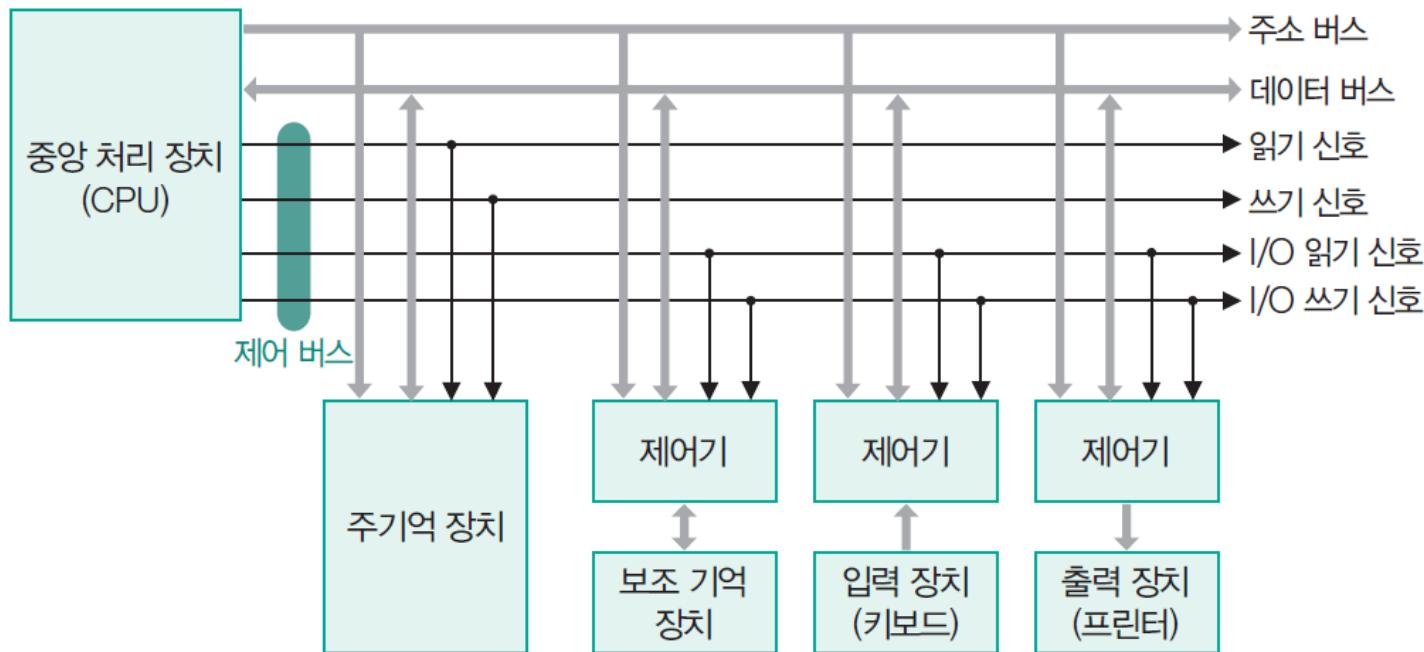


그림 1-4 중앙 처리 장치와 기억 장치 및 입출력 장치의 연결

01 컴퓨터 시스템의 구성

❖ 컴퓨터 시스템의 수행 기능

- **프로그램 실행** : 중앙 처리 장치가 주기억 장치에서 프로그램 코드를 읽어 실행
- **데이터 저장** : 프로그램 실행 결과를 주기억 장치에 저장
- **데이터 이동** : 하드 디스크 같은 보조 기억 장치에 저장되어 있는 명령어와 데이터 블록을 주기억 장치로 이동
- **데이터 입출력** : 사용자가 키보드나 마우스를 통해 입력하는 명령이나 데이터를 입력하거나 중앙 처리 장치가 처리한 결과를 모니터나 프린터로 출력
- **제어** : 프로그램에서 정해진 순서에 따라 실행되도록 각종 제어 신호를 발생

01 컴퓨터 시스템의 구성

2 소프트웨어

- 소프트웨어 : 컴퓨터를 구성하고 있는 하드웨어를 잘 동작시킬 수 있도록 제어하고, 지시하는 모든 종류의 프로그램을 의미한다.
- **프로그램** : 컴퓨터를 사용해 어떤 일을 처리하기 위해 순차적으로 구성된 명령들의 집합이다.
- 소프트웨어는 **시스템 소프트웨어**와 **응용 소프트웨어**로 나눌 수 있다.

□ 시스템 소프트웨어

- 시스템 소프트웨어(system software) : 하드웨어를 관리하고 응용 소프트웨어를 실행하는 데 필요한 프로그램이다.
- 운영체제(OS), 언어 번역 프로그램, 유ти리티 프로그램 등이 이에 속한다.
- 유ти리티 프로그램은 각종 주변 장치들을 구동하는 데 필요한 드라이버 프로그램, 백신 프로그램, TCP/IP 같이 컴퓨터를 네트워크로 연결하는 데 필요한 각종 프로그램 등을 말한다.

01 컴퓨터 시스템의 구성

□ 시스템 소프트웨어(계속)

| | |
|----------------------------|--|
| 운영체제 (Operating System) | <ul style="list-style-type: none">컴퓨터 하드웨어 자원인 중앙 처리 장치, 기억 장치, 입출력 장치, 네트워크 장치 등을 제어하고 관리하는 역할을 한다.종류 : 유닉스 UNIX, 리눅스 LINUX, 윈도우 Windows, 맥 OS MAC OS, OS, 안드로이드 등 |
| 언어 번역 프로그램 | <ul style="list-style-type: none">고급 언어 프로그램을 컴퓨터가 이해할 수 있는 기계어로 변환하는 프로그램인터프리터(interpreter)는 소스 프로그램을 한 줄씩 기계어로 번역하여 실행하기 때문에 실행 속도가 컴파일러보다 느리다. Basic이나 자바스크립트, HTML, SQL, Python 등컴파일러(compiler)는 전체 소스 프로그램을 한 번에 기계어로 직접 번역하여 실행하기 때문에 실행 속도가 빠르며, C, C++, C#, 자바 등 |
| 장치 드라이버 (Device Driver) | <ul style="list-style-type: none">컴퓨터에 온라인으로 연결된 주변 장치를 제어하는 운영체제 모듈을 말한다. |
| 링커 (Linker) | <ul style="list-style-type: none">여러 개로 분할해 작성된 프로그램에 의해 생성된 목적 프로그램 또는 라이브러리 루틴을 결합하여 실행 가능한 하나의 프로그램으로 연결하는 프로그램연결 편집기(Linkage Editor)라고도 한다. |
| 로더 (Loader) | <ul style="list-style-type: none">하드 디스크 같은 저장 장치에 보관된 프로그램을 읽어 주기억 장치에 적재한 후 실행 가능한 상태로 만드는 프로그램이다.로더는 할당, 연결, 재배치, 적재 기능을 수행한다. |

01 컴퓨터 시스템의 구성

□ 응용 소프트웨어

- 컴퓨터 시스템을 일반 사용자들이 특정한 용도에 활용하기 위해 만든 프로그램으로, 애플리케이션 application, 앱, 어플이라고도 한다.

표 1-1 응용 소프트웨어의 용도와 예

| 용도 | 예 |
|------------|-----------------------------------|
| 사무용 | 한글, MS-office 제품군 |
| 그래픽용 | 포토샵, 페인트샵, 일러스트레이터 |
| 멀티미디어용 | WinAMP, GOM Player, PowerDVD |
| 게임용 | 스타크래프트, 웍크래프트, 웨이크 |
| 통신 및 네트워크용 | 인터넷 익스플로러, 크롬, 모질라, MSN 메신저, 카카오톡 |

01 컴퓨터 시스템의 구성

3 프로그램 처리 과정

- 프로그램은 고급 언어 → 어셈블리어 → 기계어 순으로 변환



그림 1-5 고급 언어 프로그램의 변환 과정과 예

01 컴퓨터 시스템의 구성

❖ 기계어 구조 예

① LOAD A, X

- 메모리 10번지의 내용을 읽어 레지스터 A에 로드(LOAD)하라는 의미

② ADD A, Y

- 레지스터 A의 내용과 메모리 11번지의 내용을 더해 그 결과를 레지스터 A에 로드하라는 의미

③ STOR Z, A

- 레지스터 A의 내용을 메모리 12번지에 저장하라는 의미

| | 연산 코드 | 오퍼랜드 |
|-----------|-------|-------|
| LOAD A, X | 001 | 01010 |
| ADD A, Y | 011 | 01011 |
| STOR Z, A | 010 | 01100 |

3비트 5비트

그림 1-6 기계어 구조 예

❖ 연산 코드(opcode)

- CPU가 수행할 연산을 지정하는 비트들
- 비트 수 = 3이면, 지정할 수 있는 연산의 최대 수 : $2^3 = 8$ 개

❖ 오퍼랜드(operand)

- 연산에 사용될 데이터가 저장되어 있는 기억 장치 주소
- 비트 수 = 5 이면, 주소지정할 수 있는 기억 장소의 최대 수 : $2^5 = 32$ 개

01 컴퓨터 시스템의 구성

❖ 프로그램과 데이터가 주기억 장치에 저장되어 있는 형태

- 명령어와 데이터는 지정된 기억 장소에 저장
- 워드(word) 단위로 저장. 워드는 CPU에 의해 한 번에 처리될 수 있는 비트들의 그룹(8, 16, 32, 64비트)

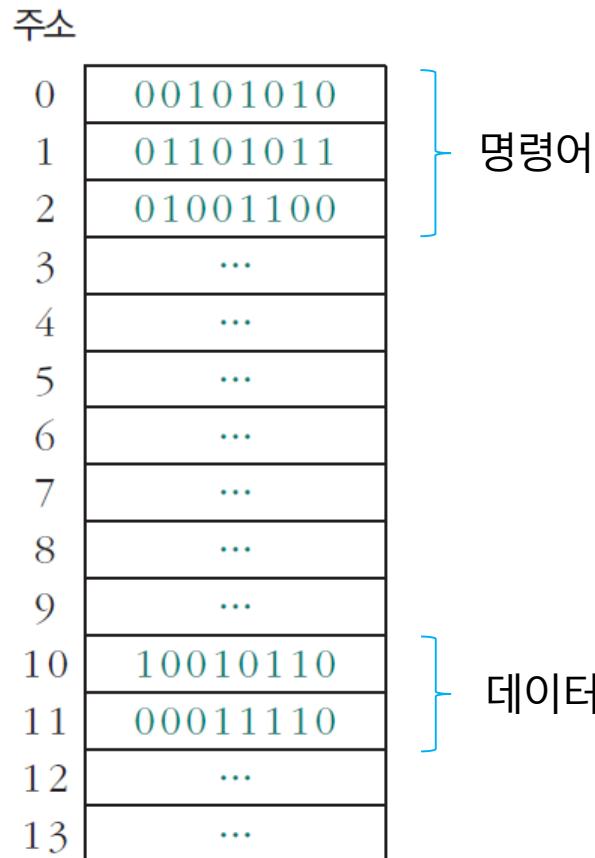
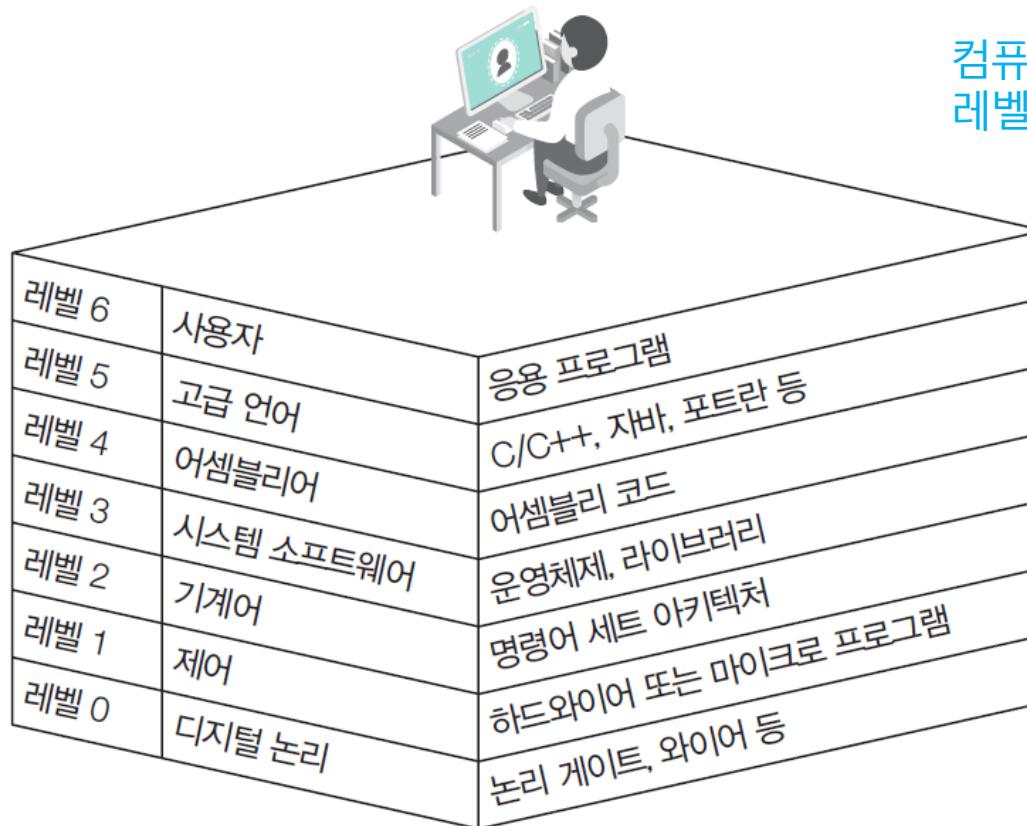


그림 1-7 프로그램과 데이터가 주기억 장치에 저장되어 있는 예

01 컴퓨터 시스템의 구성

4 컴퓨터 시스템의 계층 레벨

- **레벨 6**: 사용자 레벨이며 응용 프로그램들로 구성된다. 이 레벨에서 사용자들은 워드 프로세서, 그래픽 패키지, 게임 등과 같은 프로그램을 동작시킨다.
- **레벨 5**: C/C++, JAVA, FORTRAN 같은 고급 언어로 구성되어 있다. 고급 언어 → 어셈블리어 → 기계어 순으로 변환된다.



컴퓨터 구조에서는
레벨 0부터 레벨 2까지 다룬다.

그림 1-8 컴퓨터 시스템의 추상화 레벨

01 컴퓨터 시스템의 구성

- **레벨 4** : 어셈블리어를 포함한다. 어셈블리어 명령어 1개가 정확히 기계어 명령어 1개로 변환된다.
- **레벨 3** : 시스템 소프트웨어 레벨로 운영체제와 라이브러리들로 구성된다. 이 레벨은 다중 프로그래밍, 메모리 할당, 프로세스 관리와 기타 중요 기능을 담당한다.
- **레벨 2** : 명령어 세트 아키텍처(Instruction Set Architecture, ISA) 레벨인 컴퓨터 시스템의 특정 구조에 의해 인식되는 기계어로 구성되어 있다.
- **레벨 1**은 제어 레벨로 제어 장치가 명령어들을 해독(decode)하고 실행하며, 데이터들을 이동할 시간과 장소들을 결정한다. 제어 장치는 레벨 2에서 기계어 명령어를 한번에 하나씩 읽어서 필요한 동작을 수행하기 위한 신호들을 발생시킨다. 제어 장치는 하드웨어 hardwired 방식과 마이크로 프로그램med 방식 중 하나로 구현할 수 있다(5장에서 설명).

1 컴퓨터의 발전 과정

□ 초기의 계산 도구

- 계산을 하는 도구로서 가장 간단한 것은 주판이며, 기원전 약 3000년 전 고대 메소포타미아 인들이 가장 먼저 사용했다고 추정된다.
- 주판을 제외하면 17세기에 이르도록 계산을 위한 특별한 도구가 없었다.

□ 기계식 계산기

- **톱니바퀴를 이용한 수동 계산기** : 기어로 연결된 바퀴판들로 덧셈과 뺄셈 수행(1642년 파스칼)
- 1671년 독일의 라이프니츠는 이를 개량하여 곱셈과 나눗셈도 가능한 계산기를 발명하였다. 또 라이프니츠는 기계 장치에 더 적합한 진법을 연구해서 17세기 후반에 2진법을 창안했다.
- **차분 기관**(difference engine) : 1823년 영국의 배비지가 삼각함수를 유효숫자 5자리까지 계산하여 종이에 표로 인쇄하였다.

02 컴퓨터의 역사

□ 기계식 계산기(계속)

- **해석 기관**(analytical engine) : 방정식을 순차적으로 풀 수 있도록 고안된 기계식 계산기(1833년, 베비지). 오늘날 사용하는 컴퓨터의 기본 요소를 모두 갖춘 것이었다.

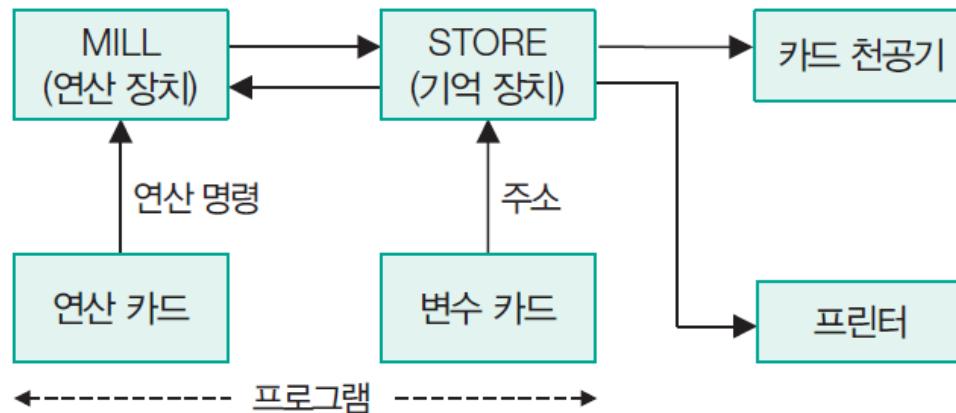


그림 1-9 해석 기관의 기본 구조

- **천공 카드 시스템**(Punch Card System, PCS) : 1889년 미국의 홀러리스는 인구조사에 활용하기 위한 시스템을 만들었다. 종이 카드에 구멍을 뚫어 자료를 처리하는 시스템으로 수 백 장의 카드를 읽고 기억하는 데 1분도 걸리지 않았다. 그 결과 1890년에 시행된 인구조사에서는 7년 이상이 걸리던 통계 처리 시간을 3년 이하로 단축시켰다.

02 컴퓨터의 역사

□ 전기 기계식 계산기

- **MARK-I** : 1944년 하버드 대학의 에이킨이 개발. MARK-I은 배비지의 해석 기관을 실현시킨 것으로서 미해군의 탄도 계산에 이용되었으며, 수많은 수학이나 과학 문제를 해결하는 데 공헌했다.

□ 전자식 계산기

- **에니악**(ENIAC, Electronic Numerical Integrator And Computer) : 진공관을 사용한 최초의 전자식 컴퓨터(1946년, 에커트와 모클리)
- **에드삭**(EDSAC, Electronic Delay Storage Automatic Calculator) : 10진수 체계와 프로그램 내장 방식의 계산기(1949년, 월키스)
- **에드박**(EDVAC, Electronic Discrete Variable Automatic Computer) : 2진수 체계와 프로그램 내장 방식을 적용(1951년, 폰 노이만)
- **유니박**(UNIVAC, UNIVersal Automatic Computer) : 최초의 상용 컴퓨터, 인구조사통계국에 설치(1951년, 모클리와 에커트)

컴퓨터의 발전 과정에 있어서 진공관 및 프로그램 내장 방식의 사용은 근대에서 현대로 넘어 오게 되는 분기점이 되었다.

02 컴퓨터의 역사

2 컴퓨터의 세대별 분류

- 컴퓨터의 발전 과정을 세대별로 명확하게 설명하기는 어렵지만 새로운 하드웨어 부품의 출현을 기준으로 분류되고 있다.

표 1-2 컴퓨터 세대 구분

| 구분 \ 세대 | 1세대 | 2세대 | 3세대 | 4세대 | 5세대 |
|----------|--------------------|------------------------------------|--|--------------------------------|-----------------------|
| 주요 소자 | 진공관 | 트랜지스터 | SSI, MSI | LSI, VLSI | VLSI, ULSI |
| 주 기억 장치 | 자기 드럼, 수은 지연 회로 | 자기 코어 | IC(RAM, ROM) | LSI, VLSI | VLSI |
| 보조 기억 장치 | 천공 카드, 종이 테이프 | 자기 드럼, 자기 디스크 | 자기 디스크, 자기 테이프 | 자기 디스크, 자기 테이프 | 자기 디스크, 광 디스크 |
| 처리 속도 | $ms(10^{-3})$ | $\mu s(10^{-6})$ | $ns(10^{-9})$ | $ps(10^{-12})$ | $fs(10^{-15})$ |
| 사용 언어 | 기계어, 어셈블리어 | 고급 언어(COBOL, FORTRAN, ALGOL) | 고급 언어(LISP, PASCAL, BASIC, PL/I) | 고급 언어 (ADA 등), 문제 지향적 언어 | 객체 지향 언어 (C++, 자바) |



릴레이



진공관



트랜지스터



IC

02 컴퓨터의 역사

❖ 집적 회로의 분류

- 집적 회로(Integrated Circuit, IC)가 개발된 3세대 이후에는 컴퓨터 세대에 대한 정의가 분명하지 않다.

| | |
|--------------------------------|---|
| SSI (Small Scale IC) | <ul style="list-style-type: none">• 트랜지스터 수십 개가 집적된 소규모 IC로, 기본 게이트 기능과 플립 플롭이 이에 해당한다. |
| MSI (Medium Scale IC) | <ul style="list-style-type: none">• 트랜지스터 수백 개가 집적된 중규모 IC로, 디코더, 인코더, 멀티플렉서, 디멀티플렉서, 카운터, 레지스터, 소형 기억 장치 등의 기능을 포함한다. |
| LSI (Large Scale IC) | <ul style="list-style-type: none">• 트랜지스터 수천 개가 집적된 대규모 IC로, 8비트 마이크로프로세서나 소규모 반도체 기억 장치 칩이 이에 해당한다. |
| VLSI (Very Large Scale IC) | <ul style="list-style-type: none">• 트랜지스터 수만에서 수십만 개 이상 집적된 초대규모 IC로, 대용량 반도체 메모리, 1만 게이트 이상의 논리 회로, 단일 칩 마이크로프로세서 등이 있다. |
| ULSI (Ultra Large Scale IC) | <ul style="list-style-type: none">• 트랜지스터가 수백만 개 이상 집적된 극대규모 IC로, 인텔의 486이나 팬티엄, 수백 메가바이트 이상의 반도체 기억 장치 칩 등이 이에 해당한다.• VLSI와 ULSI 사이의 정확한 구분은 사실 모호하다. |

02 컴퓨터의 역사

□ 1세대 컴퓨터

- 전공관을 사용함에 따라 컴퓨터 크기가 매우 크며, 열 발생량이 많고 전력 소모가 크다.
- 폰 노이만이 제안한 프로그램 내장의 개념을 도입했다.
- 수치 계산, 통계 등에 사용되었다.
- 컴퓨터 언어는 기계어와 어셈블리어를 사용했다.
- 대표적인 컴퓨터 : ENIAC, UNIVAC, EDSAC, EDVAC

□ 2세대 컴퓨터

- 자기 드럼이나 자기 디스크 같은 대용량의 보조 기억 장치가 사용되었다.
- 운영체제의 개념을 도입했다.
- 다중 프로그래밍 기법을 사용했다.
- 온라인 실시간 처리 방식을 도입했다.
- 과학 계산, 일반 사무용으로 사용되었다.
- 트랜지스터를 사용함으로써 컴퓨터는 더 고속화되고, 기억 용량이 늘었으나 소형화되었다.
- 소프트웨어 개발에 주력한 시기로, 사용된 언어로는 FORTRAN, ALGOL, COBOL 등이 있다.

02 컴퓨터의 역사

□ 3세대 컴퓨터

- 캐시(cache) 기억 장치가 등장했다.
- OMR, OCR, MICR이 도입되었다.
- 패밀리(family) 개념의 출현에 따라 프로그램의 호환성이 이루어졌다.
- 시분할 처리를 통해 멀티프로그래밍을 지원했다.
- 경영 정보 시스템(Management Information System, MIS)이 확립되었다.

□ 4세대 컴퓨터

- 마이크로프로세서가 개발되었다.
- 가상 기억 장치의 개념이 도입되었다.
- 컴퓨터 네트워크가 발전되었다.
- 개인용 컴퓨터 PC가 등장하여 대중화를 이루었다.
- 온라인 실시간 처리 시스템이 보편화되었고 기존 시스템에 비해 빠른 처리 속도를 갖게 되었다.

02 컴퓨터의 역사

□ 5세대 컴퓨터

- 5세대는 아직 명확하게 구분되지 않고 있으나 부품의 집적도보다는 획기적인 응용 소프트웨어의 출현에 의해 정의될 가능성이 있다.
- 현 시점에서는 VLSI, ULSI를 기본 소자로 하여 초미니, 초고속을 추구하며 기존 시스템의 수준을 벗어나 경영 정보, 지식 정보, 인공지능, 신경망, 퍼지, 멀티미디어, 가상 현실을 목표로 하고 있다.
- 컴퓨터와 인간의 인터페이스를 좀 더 인간에게 편리하도록 하기 위한 GUI 환경을 구현하고 있다.
- 컴퓨터의 성능을 향상시키기 위해 다중 프로세서를 사용한 병렬 처리 컴퓨터 시스템, 광 컴퓨터, 신경망 컴퓨터 등의 개발과 인공지능의 연구가 활발히 진행되고 있다.
- 비 폰 노이만 Non-Von Neumann 컴퓨터 구조가 제안되었다.
- 고도의 사람 대 기계 man-machine 인터페이스가 개발되었다.
- 객체 지향 프로그래밍 언어가 사용되고, 문자, 음성, 영상 정보가 통합되는 멀티미디어 시대가 도래했다.

3 무어의 법칙과 황의 법칙

□ 무어의 법칙

- 반도체 집적 회로의 트랜지스터 수가 12개월마다 2배로 증가한다는 법칙이다.
- 1970년대 중반부터 이 기간은 12개월에서 18개월로 늘어났다. 이럴 경우에 10년 동안 증가율은 1,000배가 아니라 100배 정도가 되었다. 1990년대부터는 18개월이 다시 24개월로 늘어났는데, 이 경우에 10년간 증가율은 32배 정도로 줄어들었다.
- 무어의 법칙은 한계에 다다르고 있다. 이것은 기술적인 한계뿐만 아니라 경제적인 한계도 있었다.

□ 황의 법칙

- 집적 회로를 뛰어넘는 메모리의 발전으로 인해서 앞으로는 1년에 2배씩 용량이 뛰어오를 것이라고 주장한 것
- 이는 계속해서 NAND flash 계열의 메모리가 지속적으로 발전하면서 실제로 증명되었다. 하지만 2010년, 불과 8년 만에 황의 법칙은 깨졌다.

03 컴퓨터의 분류

□ 컴퓨터의 분류

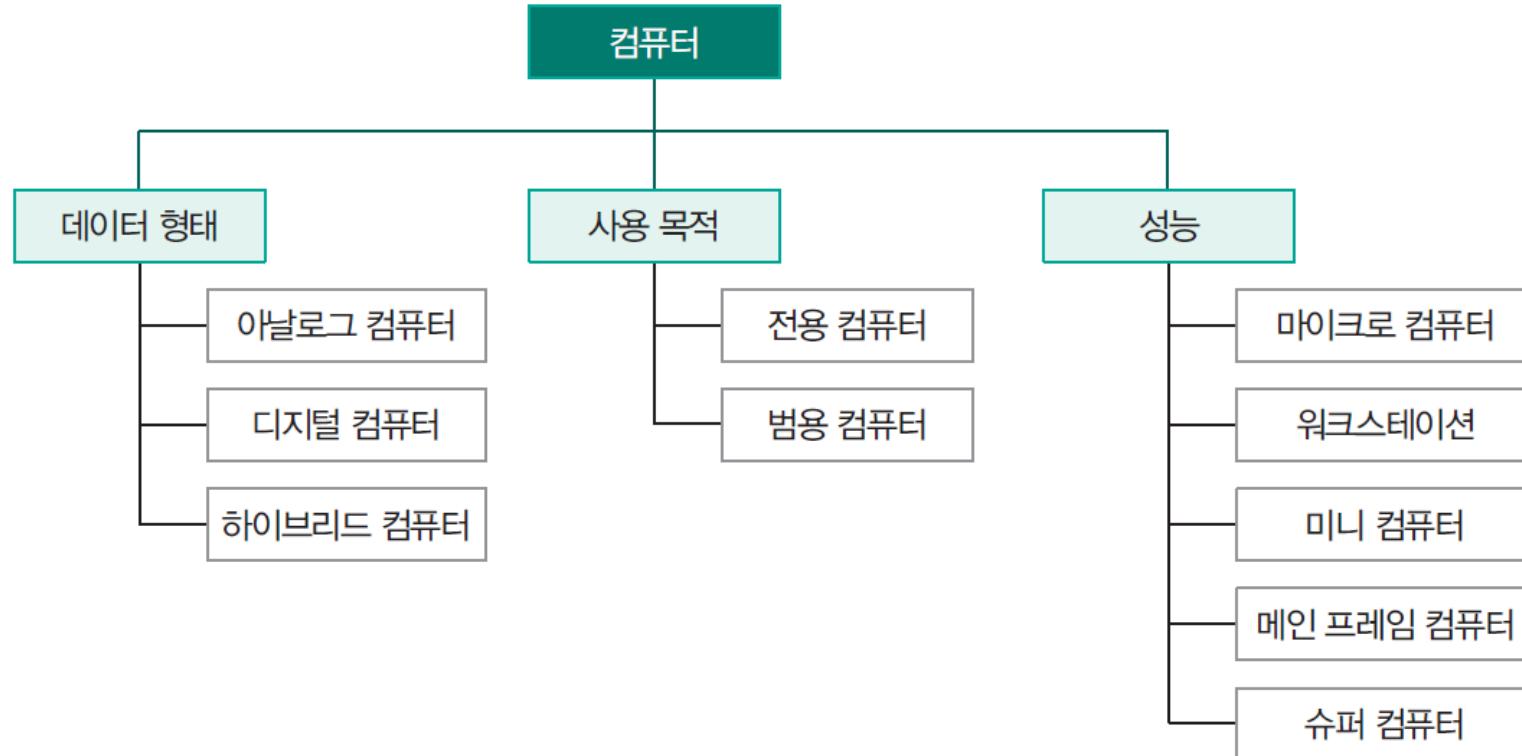


그림 1-10 컴퓨터의 분류

03 컴퓨터의 분류

1 데이터 형태에 따른 분류

□ 아날로그 컴퓨터(analog computer)

- 연속적인 변량을 사용하여 계산을 수행한다.
- 신속한 입력과 그 상태에 대한 즉각적인 반응을 얻을 수 있으므로 이 유형의 컴퓨터는 프로세스 제어 process control에 적합하다.
- 아날로그 컴퓨터에서는 전압, 전류, 온도, 압력 등의 데이터를 처리한다.

□ 디지털 컴퓨터(digital computer)

- 숫자나 문자를 코드화하여 필요한 정밀도까지의 결과를 얻을 수 있다.
- 디지털 컴퓨터는 숫자나 문자를 코드화하여 사용하며, 데이터를 분석하고 종합하여 처리한 결과를 숫자나 문자 등으로 정확히 구분할 수 있게 해준다.

□ 하이브리드 컴퓨터(hybrid computer)

- 아날로그나 디지털 신호를 모두 처리할 수 있으며 필요에 따라 아날로그 신호나 디지털 신호로 변환한다.
- 컴퓨터에서 처리한 결과는 필요에 따라 A/D 변환기나 D/A 변환기에 의해서 데이터를 아날로그 형태나 디지털 형태로 얻을 수 있다.

2 사용 목적에 따른 분류

□ 전용 컴퓨터(special purpose computer)

- 특수한 목적으로 사용하기 위한 컴퓨터로 주로 군사용, 기상 예보용, 천문학, 원자핵 물리 분야 등의 특정 업무에 사용된다.
- 전용 컴퓨터는 주로 고정 프로그램과 일정한 데이터만을 취급할 수 있도록 구성되어 있다.

□ 범용 컴퓨터(general purpose computer)

- 여러 업무에 광범위하게 사용할 수 있는 일반 목적용 컴퓨터다.
- 과학 계산, 통계 데이터 처리, 생산 관리, 사무 관리 등 광범위한 분야에 사용할 수 있다.
- 범용 컴퓨터는 여러 형태의 데이터를 취급할 수 있는 유연성, 기억 용량의 증대, 처리 속도의 신속화, 입출력 장치의 다양화 등 언제든지 필요한 시기에 확장 또는 조정할 수 있는 기능이 필요하다.

03 컴퓨터의 분류

3 성능과 규모에 따른 분류

- 정량보다는 정성적 측면에서 분류한 것이다.
- 근래에는 분류 기준을 기억 용량, 처리 능력, 가격 측면에 두고 있다.
- 이와 같은 분류는 컴퓨터 관련 기술의 급속한 발전으로 인해 구분이 모호해지고 있다.

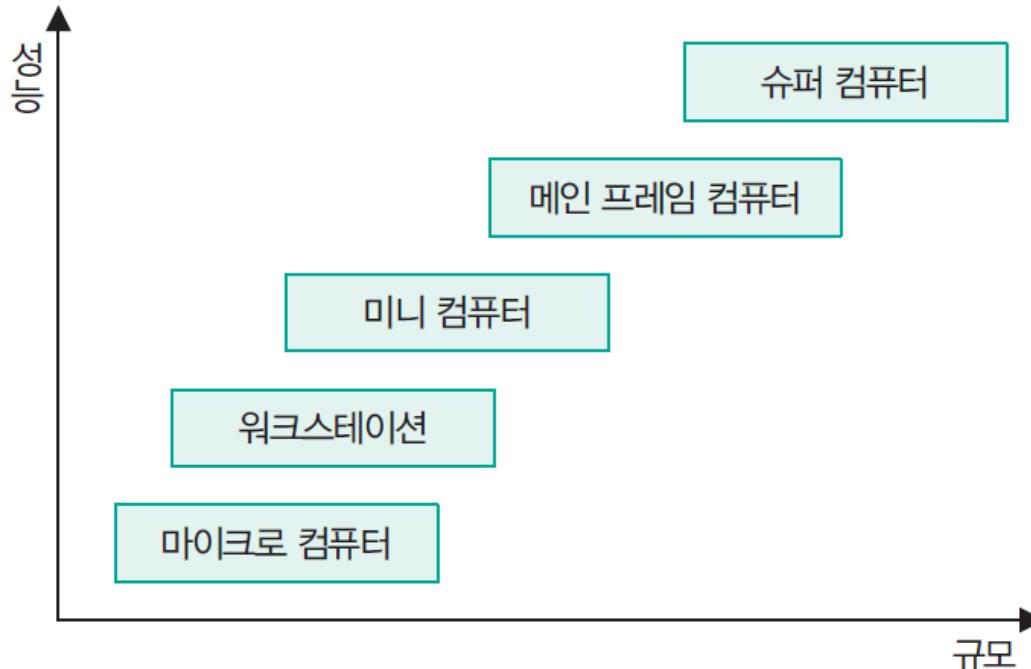


그림 1-11 처리 성능과 규모에 따른 컴퓨터 분류

03 컴퓨터의 분류

□ 마이크로 컴퓨터(microcomputer)

- 최근 가장 널리 사용되고 있는 범용 컴퓨터다.
- 마이크로프로세서를 중앙 처리 장치로 사용하는 컴퓨터를 의미하며, 워크스테이션과 개인용 컴퓨터(PC)가 있다.
- 1970년대 후반에는 개인용 컴퓨터가 등장했다. 1980년대에 들어서면서 마이크로프로세서의 성능이 급속도로 향상되면서 개인용 컴퓨터의 응용 범위도 단순 응용에서부터 매우 복잡한 응용에까지 확산되고 있었다.
- 최근에는 개인용 컴퓨터의 발전으로 워크스테이션과의 구별이 모호해지고 있다.

□ 워크스테이션(workstation)

- 과학자, 공학자, 엔지니어, 애니메이터와 같은 전문직 종사자들이 개인용 컴퓨터보다 더 우수한 성능을 요구하여 만들어진 컴퓨터다.
- 외형은 개인용 컴퓨터와 유사한데, 최근에는 워크스테이션의 가격이 떨어지고 개인용 컴퓨터의 성능이 강력해지기 때문에 워크스테이션과 개인용 컴퓨터의 구분이 사라지고 있다.

03 컴퓨터의 분류

□ 미니 컴퓨터(minicomputer)

- 성능과 규모 측면에서 마이크로 컴퓨터와 메인 프레임 컴퓨터의 중간에 해당한다. 마이크로 컴퓨터보다 대용량 기억 장치와 고속의 주변 장치들로 구성되어 있어서 다수의 사용자가 동시에 한 미니 컴퓨터를 사용할 수 있었다.
- 최근에는 컴퓨팅 서버, 파일 서버, 네트워크 서버 등과 같은 각종 서버의 등장으로 인해 그 자취를 감추게 되었다.

□ 메인 프레임 컴퓨터(main frame computer)

- 대형 컴퓨터라고도 부르는데, 1초에 수십억 개의 명령어를 처리할 수 있는 고속의 컴퓨터다.
- 대기업, 관공서, 대학 등에서 다수의 사용자가 공유하여 사용한다. 메인 프레임에는 단말기 terminal를 통해 접근할 수 있는데, 단말기는 키보드와 모니터가 통합된 장치를 말한다.
- 응용 분야는 은행, 보험, 병원 업무 등에 널리 사용되고 있다.

□ 슈퍼 컴퓨터(super computer)

- 고속의 연산 처리를 위한 중앙 처리 장치, 대규모의 용량을 가진 주기억 장치, 강력한 병렬 처리를 지원하는 소프트웨어로 이루어진 컴퓨터다.
- 고성능 마이크로프로세서를 수십만 개까지 사용하여 작업을 병렬로 처리하기 때문에 초당 수조 개의 명령어들을 처리할 수 있도록 매우 빠르다.
- 슈퍼 컴퓨터는 기상예측, 석유탐사, 원자력 개발 등의 분야에서 사용되고 있다.

04 폰 노이만, 비 폰 노이만, 하버드 구조

1 폰 노이만 구조와 비 폰 노이만 구조

- 사실 폰 노이만 구조 개념을 처음으로 생각한 사람은 모클리와 에커트였다.
- 폰 노이만 구조의 프로그램 처리 과정

- ① 프로그램 카운터를 이용해 메모리에서 실행할 명령어를 인출한다.
- ② 제어 장치는 이 명령어를 해독한다.
- ③ 명령을 실행하는 데 필요한 데이터를 메모리에서 인출하여 레지스터에 저장한다.
- ④ 산술 논리 연산 장치는 명령을 실행하고, 레지스터나 메모리에 결과를 저장한다.

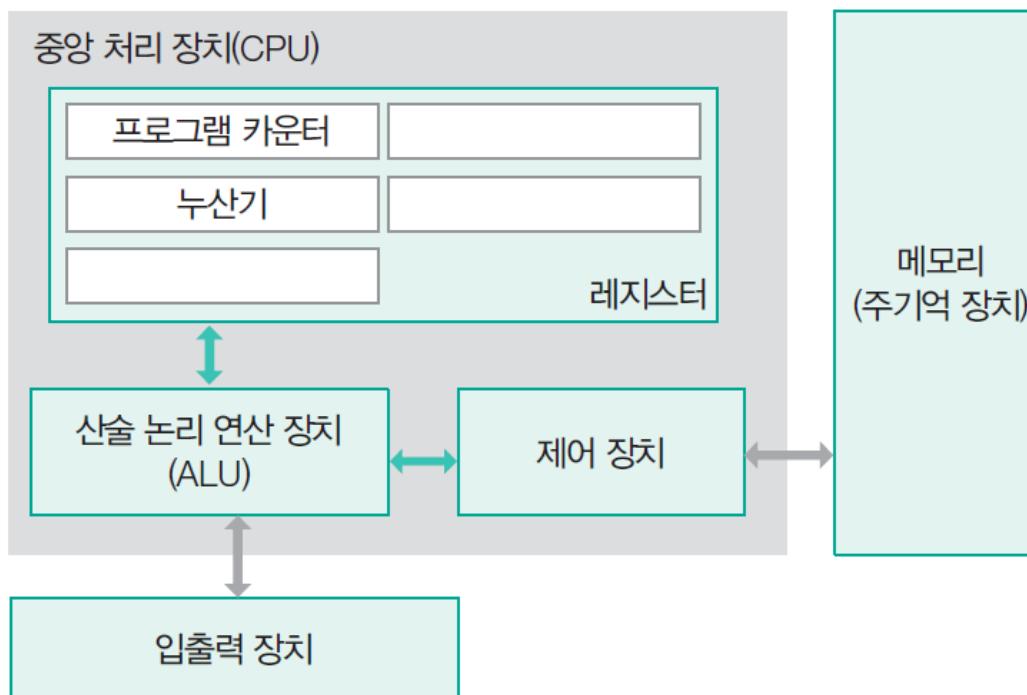


그림 1-12 폰 노이만 구조

04 폰 노이만, 비 폰 노이만, 하버드 구조

❖ 폰 노이만 병목 현상(Von-Neumann Bottleneck)

- 폰 노이만 구조의 컴퓨터는 중앙 처리 장치에서 명령어나 데이터를 메모리에서 가져와 처리한 후, 결과 데이터를 메모리에 다시 보내 저장한다. 또한 저장된 데이터가 필요할 땐 다시 메모리에서 중앙 처리 장치로 불려오는 방식으로 순차적으로 프로그램을 처리하므로 메모리나 시스템 버스에 병목 현상이 생겨 속도가 느려질 수밖에 없다.
- 이것을 데이터 경로의 병목 현상 또는 기억 장소의 지연 현상이라고도 한다.

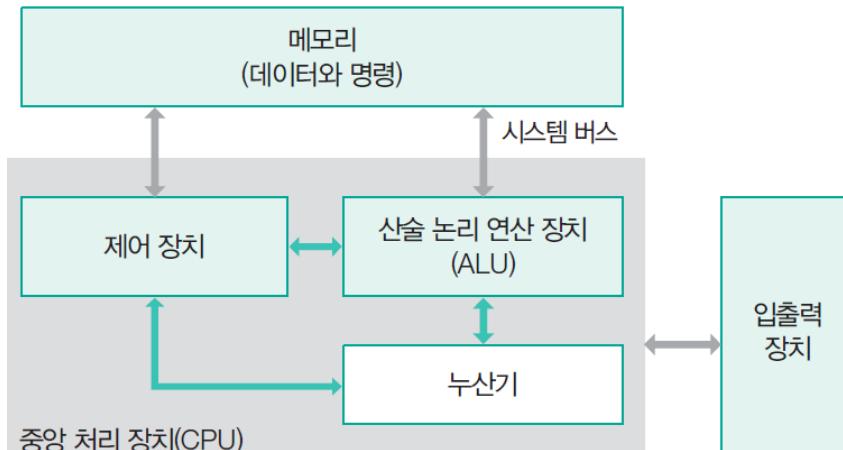
❖ 비 폰 노이만 구조

- 폰 노이만 구조가 아닌 컴퓨터를 통틀어 이르는 말로 데이터 처리의 고속화와 고도화를 위하여 프로그램 일부를 하드웨어화하거나, 병렬 처리 기능, 추론 기구를 채택한 컴퓨터를 가리킨다.
- 연구되고 있는 분야로는 신경망, 유전 알고리즘, 양자 컴퓨터, 병렬 컴퓨터를 비롯한 여러 분야들이 비 폰 노이만 범주에 속한다. 이 중에서 병렬 컴퓨팅이 현재 가장 많이 사용된다.

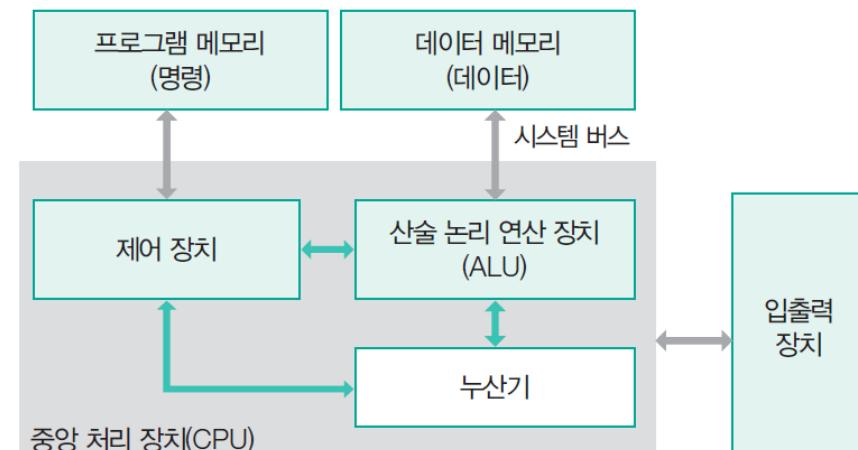
04 폰 노이만, 비 폰 노이만, 하버드 구조

2 폰 노이만 구조와 하버드 구조

- 폰 노이만 구조는 구조적으로 시스템 버스에 병목 현상이 발생한다.
- 하버드 구조는 폰 노이만 구조의 단점을 보완한 개념이다.
- 명령어 메모리 영역과 데이터 메모리 영역을 물리적으로 분리시키고, 각각을 다른 시스템 버스로 중앙 처리 장치에 연결함으로써 명령과 데이터를 메모리로부터 읽는 것을 동시에 처리할 수 있다.
- 그러나 하버드 구조는 비싸고, 공간도 많이 차지하며, 설계가 복잡하다.
- 최근 CPU 설계에서는 폰 노이만 구조와 하버드 구조 양자의 구조를 도입하고 있다.



(a) 폰 노이만 구조



(b) 하버드 구조

그림 1-13 폰 노이만 구조와 하버드 구조

Microprocessor (W2)

- Data Representation -

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

Contents

01 진법과 진법 변환

02 정수 표현

03 실수 표현

04 디지털 코드

05 에러 검출 코드

01 진법과 진법 변환

1 디지털 정보의 단위

- 1 nibble = 4bit
- 1 byte = 8bit
- 1 byte = 1문자(character)
- 영어는 1byte로 1 문자 표현, 한글은 2byte가 필요
- 1워드 : 특정 CPU에서 취급하는 명령어나 데이터의 길이에 해당하는 비트 수
- 워드 길이는 8·16·32·64비트 등 8의 배수가 가능하다.



그림 2-1 비트, 니블, 바이트의 관계

MSB_{Most Significant Bit}: 최상위 비트
LSB_{Least Significant Bit}: 최하위 비트

01 진법과 진법 변환

❖ SI 단위와 IEC 단위 비교

표 2-1 SI 단위와 IEC 단위 비교

| SI(10진수 단위) | | | IEC(2진수 단위) | | | 10진수 변환 크기 |
|----------------------|------|--------|--|----|-------|-----------------------------------|
| 값 | 기호 | 이름 | 값 | 기호 | 이름 | |
| $(10^3)^1 = 10^3$ | k, K | kilo- | $(2^{10})^1 = 2^{10} \cong 10^{3.01}$ | Ki | kibi- | 1,024 |
| $(10^3)^2 = 10^6$ | M | mega- | $(2^{10})^2 = 2^{20} \cong 10^{6.02}$ | Mi | mebi- | 1,048,576 |
| $(10^3)^3 = 10^9$ | G | giga- | $(2^{10})^3 = 2^{30} \cong 10^{9.03}$ | Gi | gibi- | 1,073,741,824 |
| $(10^3)^4 = 10^{12}$ | T | tera- | $(2^{10})^4 = 2^{40} \cong 10^{12.04}$ | Ti | tebi- | 1,099,511,627,776 |
| $(10^3)^5 = 10^{15}$ | P | peta- | $(2^{10})^5 = 2^{50} \cong 10^{15.05}$ | Pi | pebi- | 1,125,899,906,842,624 |
| $(10^3)^6 = 10^{18}$ | E | exa- | $(2^{10})^6 = 2^{60} \cong 10^{18.06}$ | Ei | exbi- | 1,152,921,504,606,846,976 |
| $(10^3)^7 = 10^{21}$ | Z | zetta- | $(2^{10})^7 = 2^{70} \cong 10^{21.07}$ | Zi | zebi- | 1,180,591,620,717,411,303,424 |
| $(10^3)^8 = 10^{24}$ | Y | yotta- | $(2^{10})^8 = 2^{80} \cong 10^{24.08}$ | Yi | yobi- | 1,208,925,819,614,629,174,706,176 |

kibi-: kilobinary, mebi-: megabinary, gibi-: gigabinary,
tebi-: terabinary, pebi-: petabinary, exbi-: exabinary,
zebi-: zettabinary, yobi-: yottabinary

01 진법과 진법 변환

2 진법

❖ 10진법

- 10진수: 기수가 10인 수
- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9의 10개 수로 표현

$$\begin{aligned}9345.35 &= 9 \times 1000 + 3 \times 100 + 4 \times 10 + 5 \times 1 + 3 \times 0.1 + 5 \times 0.01 \\&= 9 \times 10^3 + 3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0 + 3 \times 10^{-1} + 5 \times 10^{-2}\end{aligned}$$

❖ 2진법

- 기수가 2인 수
- 0, 1 두 개의 수로 표현

$$\begin{aligned}1010.1011_{(2)} &= 1 \times 1000_{(2)} + 0 \times 100_{(2)} + 1 \times 10_{(2)} + 0 \times 1_{(2)} \\&\quad + 1 \times 0.1_{(2)} + 0 \times 0.01_{(2)} + 1 \times 0.001_{(2)} + 1 \times 0.0001_{(2)} \\&= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4}\end{aligned}$$

01 진법과 진법 변환

❖ 8진법

- 0~7까지 8개의 수로 표현

$$\begin{aligned}607.36_{(8)} &= 6 \times 100_{(8)} + 0 \times 10_{(8)} + 7 \times 1_{(8)} + 3 \times 0.1_{(8)} + 6 \times 0.01_{(8)} \\&= 6 \times 8^2 + 0 \times 8^1 + 7 \times 8^0 + 3 \times 8^{-1} + 6 \times 8^{-2}\end{aligned}$$

❖ 16진법

- 0~9, A~F까지 16개의 기호로 표현

$$\begin{aligned}6C7.3A_{(16)} &= 6 \times 100_{(16)} + C \times 10_{(16)} + 7 \times 1_{(16)} + 3 \times 0.1_{(16)} + A \times 0.01_{(16)} \\&= 6 \times 16^2 + C \times 16^1 + 7 \times 16^0 + 3 \times 16^{-1} + A \times 16^{-2}\end{aligned}$$

- 8진수보다는 16진수를 사용하는 경우가 더 많은데 실제로 컴퓨터 구조나 어셈블리어에서는 16진수를 많이 쓴다. 자릿수를 더 짧게 표현할 수 있기 때문이다.

01 진법과 진법 변환

❖ 2진수에 해당하는 8진수, 16진수, 10진수 표현

표 2-2 2진수에 해당하는 8진수, 16진수, 10진수 표현

| 2진수 | 8진수 | 10진수 |
|-----|-----|------|
| 000 | 0 | 0 |
| 001 | 1 | 1 |
| 010 | 2 | 2 |
| 011 | 3 | 3 |
| 100 | 4 | 4 |
| 101 | 5 | 5 |
| 110 | 6 | 6 |
| 111 | 7 | 7 |

| 2진수 | 16진수 | 10진수 | 2진수 | 16진수 | 10진수 |
|------|------|------|------|------|------|
| 0000 | 0 | 0 | 1000 | 8 | 8 |
| 0001 | 1 | 1 | 1001 | 9 | 9 |
| 0010 | 2 | 2 | 1010 | A | 10 |
| 0011 | 3 | 3 | 1011 | B | 11 |
| 0100 | 4 | 4 | 1100 | C | 12 |
| 0101 | 5 | 5 | 1101 | D | 13 |
| 0110 | 6 | 6 | 1110 | E | 14 |
| 0111 | 7 | 7 | 1111 | F | 15 |

01 진법과 진법 변환

3 진법 변환

□ 2진수, 8진수, 16진수를 10진수로 변환

① 2진수를 10진수로 변환한 예

$$\begin{aligned}101101.101_{(2)} &= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\&= 32 + 0 + 8 + 4 + 0 + 1 + 0.5 + 0 + 0.125 \\&= 45.625_{(10)}\end{aligned}$$

② 8진수를 10진수로 변환한 예

$$\begin{aligned}364.35_{(8)} &= 3 \times 8^2 + 6 \times 8^1 + 4 \times 8^0 + 3 \times 8^{-1} + 5 \times 8^{-2} \\&= 3 \times 64 + 6 \times 8 + 4 \times 1 + 3 \times 0.125 + 5 \times 0.015625 \\&= 192 + 48 + 4 + 0.375 + 0.078125 \\&= 244.453125_{(10)}\end{aligned}$$

③ 16진수를 10진수로 변환한 예

$$\begin{aligned}A3.D2_{(16)} &= 10 \times 16^1 + 3 \times 16^0 + 13 \times 16^{-1} + 2 \times 16^{-2} \\&= 10 \times 16 + 3 \times 1 + 13 \times 0.0625 + 2 \times 0.00390625 \\&= 160 + 3 + 0.8125 + 0.0078125 \\&= 163.8203125_{(10)}\end{aligned}$$

01 진법과 진법 변환

□ 10진수 - 2진수 변환

- 정수부분과 소수부분으로 나누어 변환
- 정수부분은 2로 나누고, 소수부분은 2를 곱한다.
- 10진수 75.6875를 2진수로 변환한 예

$$\begin{array}{r} 2 \mid 75 \\ 2 \mid 37 \\ 2 \mid 18 \\ 2 \mid 9 \\ 2 \mid 4 \\ 2 \mid 2 \\ 2 \mid 1 \\ 0 \end{array} \quad \text{나머지} \longrightarrow \text{2진수}$$

몫

| 2진수 | 정수 | 소수 |
|--------|-----|------|
| 0.1 | 0. | 6875 |
| | × 2 | |
| 0.10 | 1. | 3750 |
| | × 2 | |
| 0.101 | 0. | 7500 |
| | × 2 | |
| 0.1011 | 1. | 5000 |
| | × 2 | |
| | 1. | 0 |

곱셈 결과 정수를 적는다.

소수 부분이 0이 될 때까지 계산한다.

$$75.6875_{(10)} = 1001011.1011_{(2)}$$

01 진법과 진법 변환

□ 10진수 - 8진수 변환

- 10진수 75.6875를 8진수로 변환
- 8로 나누고, 곱한다.

$$\begin{array}{r} 8 \longdiv{75} & \text{나머지} \longrightarrow 8\text{진수} \\ \hline 8 \longdiv{9} & \cdots 3 \longrightarrow 3 \\ \hline 8 \longdiv{1} & \cdots 1 \longrightarrow 13 \\ \hline 0 & \cdots 1 \longrightarrow 113 \\ \text{몫} & \end{array}$$

$$\begin{array}{r} 8\text{진수} \leftarrow \begin{array}{l} \text{정수} \\ \text{소수} \end{array} \\ \hline 0.6875 \\ \times 8 \\ \hline 5.5000 \\ \times 8 \\ \hline 4.0 \\ \hline \end{array}$$

곱셈 결과 정수를 적는다.

소수 부분이 0이 될 때까지 계산한다.

$$75.6875_{(10)} = 113.54_{(8)}$$

01 진법과 진법 변환

□ 10진수 - 16진수 변환

- 10진수 75.6875를 16진수로 변환
- 16으로 나누고, 곱한다.

$$\begin{array}{r} 16 \longdiv{75} & \text{나머지} \longrightarrow 16\text{진수} \\ \hline 16 \longdiv{4} & \cdots 11 \longrightarrow B \\ 0 & \cdots 4 \longrightarrow 4B \\ \text{몫} & \end{array}$$

$$\begin{array}{r} 16\text{진수} \leftarrow \begin{array}{l} \text{정수} \\ \times \end{array} \begin{array}{l} \text{소수} \\ 16 \end{array} \\ \hline 0.6875 \\ \times 16 \\ \hline 11.0000 \\ 0.B \leftarrow \end{array}$$

곱셈 결과 정수를 적는다.
소수 부분이 0이 될 때까지 계산한다.

$$75.6875_{(10)} = 4B.B_{(16)}$$

01 진법과 진법 변환

□ 2진수 - 8진수 - 10진수 - 16진수 상호 변환

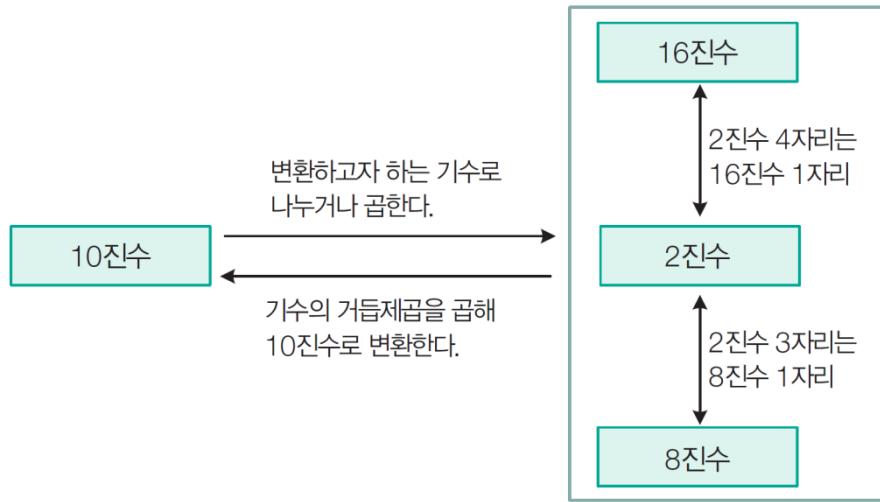


그림 2-2 2진수, 8진수, 10진수, 16진수 상호 변환 개념도

| 10진수 | 2진수 | 8진수 | 16진수 |
|------|------|-----|------|
| 0 | 0000 | 00 | 0 |
| 1 | 0001 | 01 | 1 |
| 2 | 0010 | 02 | 2 |
| 3 | 0011 | 03 | 3 |
| 4 | 0100 | 04 | 4 |
| 5 | 0101 | 05 | 5 |
| 6 | 0110 | 06 | 6 |
| 7 | 0111 | 07 | 7 |
| 8 | 1000 | 10 | 8 |
| 9 | 1001 | 11 | 9 |
| 10 | 1010 | 12 | A |
| 11 | 1011 | 13 | B |
| 12 | 1100 | 14 | C |
| 13 | 1101 | 15 | D |
| 14 | 1110 | 16 | E |
| 15 | 1111 | 17 | F |

01 진법과 진법 변환

- 10진수를 8진수, 16진수로 변환한 예

$$\begin{aligned} 75.6875 &= 1001011.1011_{(2)} \\ &= \textcolor{teal}{001} \ 001 \ 011.101 \ \textcolor{teal}{100}_{(2)} \\ &= \quad 1 \quad 1 \quad 3. \quad 5 \quad 4_{(8)} \end{aligned}$$

$$\begin{aligned} 75.6875 &= 1001011.1011_{(2)} \\ &= \textcolor{teal}{0100} \ 1011.1011_{(2)} \\ &= \quad 4 \quad \text{B.} \quad \text{B}_{(16)} \end{aligned}$$

- 8진수와 16진수를 2진수로 변환한 예

$$\begin{aligned} 3 \quad 6 \quad 7. \quad 7 \quad 5_{(8)} \\ = 011 \ 110 \ 111. \ 111 \ 101_{(2)} \end{aligned}$$

$$\begin{aligned} 9 \quad \text{A} \quad 3. \quad 5 \quad 0 \quad \text{F} \quad 3_{(16)} \\ = 1001 \ 1010 \ 0011. \ 0101 \ 0000 \ 1111 \ 0011_{(2)} \end{aligned}$$

01 진법과 진법 변환

예제 2-1 10진수 48.8125를 2진수, 8진수, 16진수로 변환하여라.

풀이

$$\begin{array}{r} 2 \mid 48 & \text{나머지} \longrightarrow 2\text{진수} \\ 2 \mid 24 \cdots 0 \longrightarrow 0 \\ 2 \mid 12 \cdots 0 \longrightarrow 00 \\ 2 \mid 6 \cdots 0 \longrightarrow 000 \\ 2 \mid 3 \cdots 0 \longrightarrow 0000 \\ 2 \mid 1 \cdots 1 \longrightarrow 10000 \\ 0 \cdots 1 \longrightarrow 110000 \\ \text{몫} \end{array}$$

| 2진수 | 정수 | 소수 |
|--------|----------|------|
| 0. | 8125 | |
| | \times | 2 |
| 0.1 | 1. | 6250 |
| | \times | 2 |
| 0.11 | 1. | 2500 |
| | \times | 2 |
| 0.110 | 0. | 5000 |
| | \times | 2 |
| 0.1101 | 1. | 0 |

$$110000.1101_{(2)} = 110\ 000.\ 110\ 100_{(2)} = 60.64_{(8)}$$

$$= 0011\ 0000.1101_{(2)} = 30.D_{(16)}$$

End of Example

1 보수의 개념과 음수

❖ 최상위비트(MSB)를 부호비트로 사용

- 양수(+) : 0 음수(-) : 1

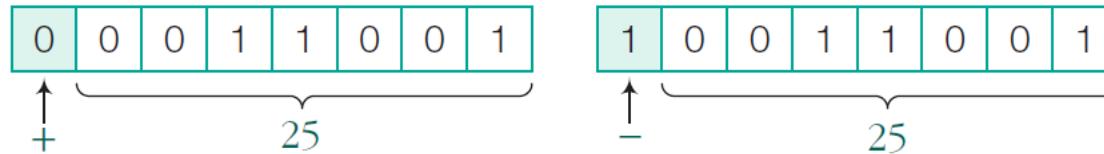
❖ 2진수 음수를 표시하는 방법

- 부호와 절댓값(sign-magnitude)
- 1의 보수(1's complement)
- 2의 보수(2's complement)

02 정수 표현

❖ 부호와 절댓값

- 부호비트만 양수와 음수를 나타내고 나머지 비트들은 같다.



❖ 1의 보수 방식

- $0 \rightarrow 1, 1 \rightarrow 0$ 으로 변환

$$\begin{array}{cccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \downarrow & \downarrow \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \end{array} \leftarrow 1\text{의 보수}$$

❖ 2의 보수 방식

- $1\text{의 보수} + 1 = 2\text{의 보수}$

$$\begin{array}{cccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \downarrow & \downarrow \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \end{array} \leftarrow 1\text{의 보수}$$

$+$

$$\begin{array}{c} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \end{array} \leftarrow 2\text{의 보수}$$

02 정수 표현

❖ 수의 표현 방법에 따른 10진수 대응 값

표 2-3 수의 표현 방법에 따른 10진수 대응 값

| 4비트 2진수 | 부호 없는 수 | 부호와 절댓값 | 1의 보수 | 2의 보수 |
|---------|---------|---------|-------|-------|
| 0000 | 0 | 0 | 0 | 0 |
| 0001 | 1 | 1 | 1 | 1 |
| 0010 | 2 | 2 | 2 | 2 |
| 0011 | 3 | 3 | 3 | 3 |
| 0100 | 4 | 4 | 4 | 4 |
| 0101 | 5 | 5 | 5 | 5 |
| 0110 | 6 | 6 | 6 | 6 |
| 0111 | 7 | 7 | 7 | 7 |
| 1000 | 8 | -0 | -7 | -8 |
| 1001 | 9 | -1 | -6 | -7 |
| 1010 | 10 | -2 | -5 | -6 |
| 1011 | 11 | -3 | -4 | -5 |
| 1100 | 12 | -4 | -3 | -4 |
| 1101 | 13 | -5 | -2 | -3 |
| 1110 | 14 | -6 | -1 | -2 |
| 1111 | 15 | -7 | -0 | -1 |

02 정수 표현

- ❖ 양수를 보수로 바꾸면 음수
- ❖ 음수를 보수로 바꾸면 양수

양수 $\xleftarrow{\text{보수}} \xrightarrow{\text{보수}}$ 음수

- ❖ 2진수와 그 수의 1의 보수와의 합은 모든 bit가 1이 된다.
- ❖ 2진수와 그 수의 2 보수와의 합은 모든 bit가 0이 된다.
(자릿수를 벗어나는 비트는 제외)

02 정수 표현

❖ 2의 보수에 대한 10진수의 표현 범위

표 2-4 n 비트 2의 보수에 대한 10진수의 표현 범위

| 비트 수 | 2의 보수를 사용한 2진 정수의 표현 범위 |
|--------|--|
| n 비트 | $-2^{n-1} \sim +2^{n-1}-1$ |
| 4비트 | $-2^{4-1}(-8) \sim +2^{4-1}-1(+7)$ |
| 8비트 | $-2^{8-1}(-128) \sim +2^{8-1}-1(+127)$ |
| 16비트 | $-2^{16-1}(-32,768) \sim +2^{16-1}-1(+32767)$ |
| 32비트 | $-2^{32-1}(-2,147,483,648) \sim +2^{32-1}-1(+2,147,483,647)$ |
| 64비트 | $-2^{64-1}(-9,223,372,036,854,775,808) \sim +2^{64-1}-1(+9,223,372,036,854,775,807)$ |

02 정수 표현

□ 2의 보수를 10진수로 변환

❖ 2의 보수로 표현된 수 $00101101_{(2)}$ 을 10진수로 변환하는 방법

- 최상위 비트가 0이므로 양수. 따라서 $00101101_{(2)}$ 의 값을 10진수로 변환하여 +부호를 붙인다.

$$\begin{aligned}00101101_{(2)} &= 0 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\&= 0 + 0 + 32 + 0 + 8 + 4 + 0 + 1 \\&= 45\end{aligned}$$

❖ 2의 보수로 표현된 수 $10101100_{(2)}$ 을 10진수로 변환하는 방법

- 최상위 비트가 1이므로 음수다.
- 2의 보수로 바꾼 후 10진수로 변환하여 -부호를 붙이면 -84가 된다.

$$10101100_{(2)} \xrightarrow[\text{음수} \rightarrow \text{양수}]{\substack{\text{1의 보수} + 1 = 01010011 + 1 = 2\text{의 보수}}} 01010100_{(2)}$$

$$\begin{aligned}01010100_{(2)} &= 0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\&= 0 + 64 + 0 + 16 + 0 + 4 + 0 + 0 \\&= 84\end{aligned}$$

02 정수 표현

2 부호 확장

- 부호 확장이란 늘어난 비트 수 만큼 부호를 늘려주는 방법

표 2-5 2진수 표현 방법에 따른 부호 확장

| 2진수 표현 방식 | 부호 확장 방법 | 구분 | 8비트 | 16비트 확장 |
|-----------|------------------------------|----|----------|-------------------|
| 부호와 절댓값 | 부호만 MSB에 복사하고, 나머지는 0으로 채운다. | 양수 | 00101010 | 00000000 00101010 |
| | | 음수 | 10010111 | 10000000 00010111 |
| 1의 보수 | 늘어난 길이만큼 부호와 같은 값으로 모두 채운다. | 양수 | 00101010 | 00000000 00101010 |
| | | 음수 | 10010111 | 11111111 10010111 |
| 2의 보수 | 늘어난 길이만큼 부호와 같은 값으로 모두 채운다. | 양수 | 00101010 | 00000000 00101010 |
| | | 음수 | 10010111 | 11111111 10010111 |

02 정수 표현

3 2진 정수 연산

- 뺄셈의 원리를 보면, A-B 대신에 A+(B의 2의 보수)를 계산하면 된다.
- 뺄셈에서 2의 보수 방식을 사용하는 이유는 뺄셈을 가산기를 사용하여 수행할 수 있다.

1 자리올림수 → 0110000

$$\begin{array}{r} \text{양수} & 49 = 00110001 \\ + \text{양수} & +58 = +00111010 \\ \hline \text{양수} & 107 = 0\ 01101011 \end{array}$$

2 자리올림수 → 1111110

$$\begin{array}{r} \text{양수} & 58 = 00111010 \\ - \text{양수} & -49 = -00110001 \\ \hline \text{양수} & 00111010 \\ + \text{음수} & +11001111 \\ \hline \text{양수} & 9 = 100001001 \end{array}$$

2의 보수

3 자리올림수 → 0000000

$$\begin{array}{r} \text{양수} & 49 = 00110001 \\ - \text{양수} & -58 = -00111010 \\ \hline \text{양수} & 00110001 \\ + \text{음수} & +11000110 \\ \hline \text{음수} & -9 = 011110111 \end{array}$$

2의 보수

4 자리올림수 → 1001110

$$\begin{array}{r} - \text{양수} & -49 = -00110001 \\ - \text{양수} & -58 = -00111010 \\ \hline \text{음수} & 11001111 \\ + \text{음수} & +11000110 \\ \hline \text{음수} & -107 = 1\ 10010101 \end{array}$$

2의 보수

5 자리올림수 → 1000010

$$\begin{array}{r} \text{양수} & 98 = 01100010 \\ + \text{양수} & +74 = +01001010 \\ \hline \text{음수} & -84 = 010101100 \end{array}$$

서로 다른

overflow

6 자리올림수 → 0111110

$$\begin{array}{r} - \text{양수} & -98 = -01100010 \\ - \text{양수} & -74 = -01001010 \\ \hline \text{음수} & 10011110 \\ + \text{음수} & +10110110 \\ \hline \text{양수} & +84 = 1\ 01010100 \end{array}$$

서로 다른

underflow

02 정수 표현

예제 2-2

8비트 연산 $98+74$ 는 오버플로가 발생하여 잘못 계산된 결과를 얻었다. 부호 확장으로 올바른 결과를 얻을 수 있음을 설명하여라.

풀이

부호를 확장하여 연산하는 과정은 다음과 같다.

$$\begin{array}{r} \text{carry} & \underline{0000000} & 1000010 \\ 98 & = & 00000000 & 01100010 \\ + 74 & = & + 00000000 & 01001010 \\ \hline 172 & = & \underline{0} & 10101100 \end{array}$$

직전 캐리와 최종 캐리(밑줄 친 부분)가 같으므로 정상적으로 계산된 것임을 알 수 있고, 그 결과는 $10101100_{(2)} = 128+32+8+4 = 172_{(10)}$ 이다.

End of Example

03 실수 표현

- 컴퓨터의 부동소수점수는 **IEEE 754표준**을 따른다.
- 부호(sign)**, **지수(exponent)**, **가수(mantissa)**의 세 영역으로 표시
- 부호(S)가 0일 때는 양수를 나타내고, 1일 때는 음수를 나타낸다.
- 단정도(single precision) 부동소수점수와 배정도(double precision) 부동소수점수의 두 가지 표현 방법이 있다.

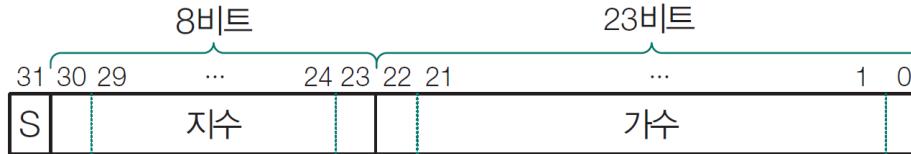
| 구분 | IEEE 754 표준 부동 소수점 수의 비트 할당 | 바이어스 |
|-----------------|---|------|
| 단정도 부동 소수점 수 |  | 127 |
| 배정도 부동 소수점 수 |  | 1023 |

그림 2-5 단정도 및 배정도 부동 소수점 수에 할당된 비트 수

단정도(단일정밀도): 32비트, 배정도(2배정밀도): 64비트,
4배정도(4배정밀도): 128비트

03 실수 표현

□ 정규화(normalization) : 과학적 표기방법

❖ 2진수의 정규화

$$\begin{aligned} 75.6875 &= 1001011.1011_{(2)} \\ &= 1.0010111011_{(2)} \times 2^6 \\ &= 1.0010111011_{(2)} \times 2^{110_{(2)}} \end{aligned}$$

❖ 바이어스(bias) : 지수의 양수, 음수를 나타내기 위한 방법

- IEEE 754 표준에서는 바이어스 127(단정도) 또는 1023(배정도)을 사용
- 표현 지수 = 바이어스 + 2진 지수 값

| 부호 | 지수(바이어스 127) | 가수($1.xxx_{(2)}$) |
|----|------------------------|--------------------------|
| 양수 | 01111111(127) + 110(6) | 1.을 생략한 가수 |
| 0 | 10000101 | 001011101100000000000000 |

03 실수 표현

❖ 10진수 -0.2를 단정도 부동소수점으로 표현

- 2진수로 변환하고 정규화한다.

$$\begin{aligned}-0.2 &= -0.00110011001100110011001\dots_{(2)} \\ &= -1.10011001100110011001\dots_{(2)} \times 2^{-3} \\ &= -1.10011001100110011001\dots \times 2^{-11}_{(2)}\end{aligned}$$

| 부호 | 지수(바이어스 127) | 가수($1.xxx_{(2)}$) |
|----|-----------------------|-------------------------|
| 음수 | 01111111(127) – 11(3) | 1.을 생략한 가수 |
| 1 | 01111100 | 10011001100110011001100 |

03 실수 표현

□ 컴퓨터에서의 부동소수점수의 표현 범위

표 2-6 단정도 및 배정도 부동 소수점 수의 표현 범위

| 구분 | 단정도 부동 소수점 수 | 배정도 부동 소수점 수 |
|-----------|---|--|
| 비정규화된 2진수 | $\sim \pm 2^{-149}$ 에서 $\pm (1 - 2^{-23}) \times 2^{126}$ 까지 | $\sim \pm 2^{-1074}$ 에서 $\pm (1 - 2^{-52}) \times 2^{1022}$ 까지 |
| 정규화된 2진수 | $\sim \pm 2^{-126}$ 에서 $\pm (2 - 2^{-23}) \times 2^{127}$ 까지 | $\sim \pm 2^{-1022}$ 에서 $\pm (2 - 2^{-52}) \times 2^{1023}$ 까지 |
| 10진수 | $\sim \pm 1.40 \times 10^{-45}$ 에서 $\pm 3.40 \times 10^{38}$ 까지 | $\sim \pm 4.94 \times 10^{-324}$ 에서 $\pm 1.798 \times 10^{308}$ 까지 |

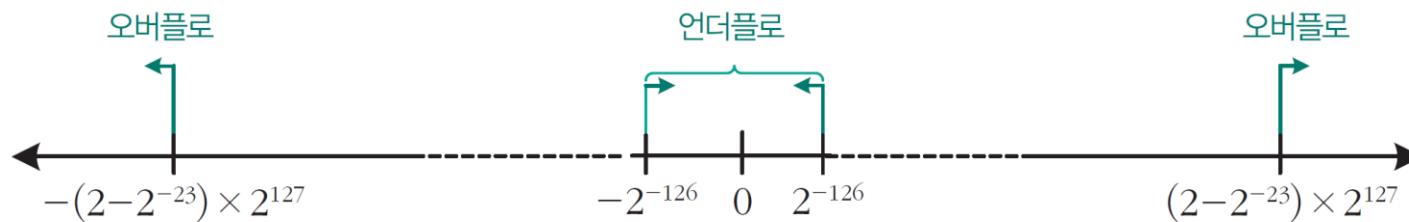


그림 2-6 단정도 부동 소수점 수의 표현 범위

03 실수 표현

예제 2-3

$\frac{1}{256}$ 을 단정도 부동소수점 방식으로 표현하여라.

풀이

$\frac{1}{256}$ 을 정규화된 방법으로 표현하면 $\frac{1}{256} = \frac{1}{2^8} = 2^{-8} = 1.0 \times 2^{-8}$ 이다.

그러므로 $1.0_{(2)} \times 2^{-1000_{(2)}}$ 를 단정도 부동소수점수로 표현하면 다음과 같다.

| 부호 | 지수(bias 127) | 가수($1.\times \times \times_{(2)}$) |
|----|-------------------------|--------------------------------------|
| 양수 | 01111111(127) - 1000(8) | 1.을 생략한 가수 |
| 0 | 01110111 | 00000000000000000000000000000000 |

End of Example

04 디지털 코드

1 BCD 코드(Binary Coded Decimal Code : 2진화 10진 코드, 8421 코드)

- BCD코드는 10진수 0(0000)부터 9(1001)까지를 2진화한 코드
- 표기는 2진수이지만 의미는 10진수
- 1010부터 1111까지 6개는 사용하지 않음

표 2-7 BCD(8421) 코드

| 10진수 | BCD 코드 | 10진수 | BCD 코드 | 10진수 | BCD 코드 |
|------|--------|------|-----------|------|----------------|
| 0 | 0000 | 10 | 0001 0000 | 20 | 0010 0000 |
| 1 | 0001 | 11 | 0001 0001 | 31 | 0011 0001 |
| 2 | 0010 | 12 | 0001 0010 | 42 | 0100 0010 |
| 3 | 0011 | 13 | 0001 0011 | 53 | 0101 0011 |
| 4 | 0100 | 14 | 0001 0100 | 64 | 0110 0100 |
| 5 | 0101 | 15 | 0001 0101 | 75 | 0111 0101 |
| 6 | 0110 | 16 | 0001 0110 | 86 | 1000 0110 |
| 7 | 0111 | 17 | 0001 0111 | 97 | 1001 0111 |
| 8 | 1000 | 18 | 0001 1000 | 196 | 0001 1001 0110 |
| 9 | 1001 | 19 | 0001 1001 | 237 | 0010 0011 0111 |

04 디지털 코드

- 예를 들어 10진수 237을 BCD 코드로 변환하면

| | | | |
|--------|------|------|------|
| 10진수 | 2 | 3 | 7 |
| | ↓ | ↓ | ↓ |
| BCD 코드 | 0010 | 0011 | 0111 |

- BCD 코드의 연산은 다음 예에서 ① ~ ②와 같이 10진수처럼 연산한다.
- 계산 결과가 ③과 같이 9를 초과하여 BCD 코드를 벗어나는 경우에는 결과에 6(0110)을 더한다.

① 10진수 덧셈 BCD 덧셈

$$\begin{array}{r} 6 \\ + 3 \\ \hline 9 \end{array} \quad \begin{array}{r} 0110_{(\text{BCD})} \\ + 0011_{(\text{BCD})} \\ \hline 1001_{(\text{BCD})} \end{array}$$

② 10진수 덧셈 BCD 덧셈

$$\begin{array}{r} 42 \\ + 37 \\ \hline 79 \end{array} \quad \begin{array}{r} 0100\ 0010_{(\text{BCD})} \\ + 0011\ 0111_{(\text{BCD})} \\ \hline 0111\ 1001_{(\text{BCD})} \end{array}$$

③ 10진수 덧셈 BCD 덧셈

$$\begin{array}{r} 8 \\ + 7 \\ \hline \end{array} \quad \begin{array}{r} 1000_{(\text{BCD})} \\ + 0111_{(\text{BCD})} \\ \hline 1111_{(\text{BCD})} \end{array}$$
$$\begin{array}{r} 15 \\ + 0110_{(\text{BCD})} \\ \hline 0001\ 0101_{(\text{BCD})} \end{array}$$

04 디지털 코드

2 3초과 코드(excess-3 code)

- BCD코드(8421코드)로 표현된 값에 3을 더해 준 값으로 나타내는 코드
- 자기 보수의 성질

표 2-8 3초과 코드

| 10진수 | BCD 코드 | 3초과 코드 |
|------|--------|--------|
| 0 | 0000 | 0011 |
| 1 | 0001 | 0100 |
| 2 | 0010 | 0101 |
| 3 | 0011 | 0110 |
| 4 | 0100 | 0111 |
| 5 | 0101 | 1000 |
| 6 | 0110 | 1001 |
| 7 | 0111 | 1010 |
| 8 | 1000 | 1011 |
| 9 | 1001 | 1100 |

보수 관계

3 그레이 코드

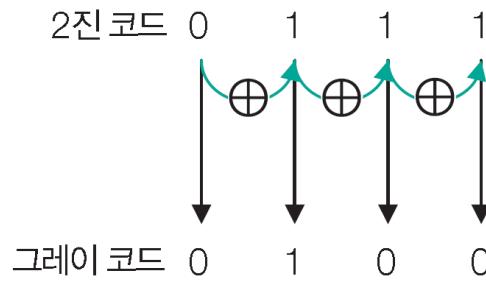
- 가중치가 없는 코드이기 때문에 연산에는 부적당하지만, 아날로그-디지털 변환기나 입출력 장치 코드로 주로 쓰인다.
 - 연속되는 코드들 간에 하나의 비트만 변화하여 새로운 코드가 된다.

표 2-9 4비트 그레이 코드

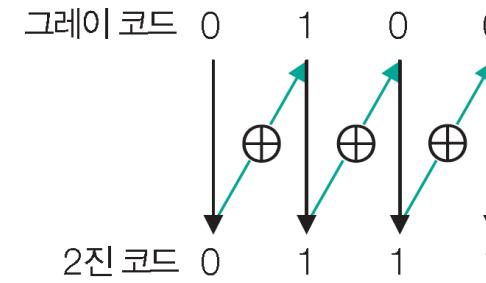
| 10진 코드 | 2진 코드 | 그레이 코드 | 10진 코드 | 2진 코드 | 그레이 코드 | |
|--------|-------|--------|--------|-------|--------|---------|
| 0 | 0000 | 0000 | 8 | 1000 | 1100 | 0 1 1 0 |
| 1 | 0001 | 0001 | 9 | 1001 | 1101 | 0 1 1 1 |
| 2 | 0010 | 0011 | 10 | 1010 | 1111 | |
| 3 | 0011 | 0010 | 11 | 1011 | 1110 | |
| 4 | 0100 | 0110 | 12 | 1100 | 1010 | |
| 5 | 0101 | 0111 | 13 | 1101 | 1011 | |
| 6 | 0110 | 0101 | 14 | 1110 | 1001 | |
| 7 | 0111 | 0100 | 15 | 1111 | 1000 | |

04 디지털 코드

❖ 2진 코드와 그레이 코드의 상호 변환 방법



(a) 2진 코드를 그레이 코드로 변환하는 방법



(b) 그레이 코드를 2진 코드로 변환하는 방법

그림 2-7 2진 코드와 그레이 코드의 상호 변환 방법

XOR의 진리표

| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

$$F = A \oplus B$$

4 다양한 2진 코드

- **가중치 코드** : 각 비트의 위치에 따라 값이 정해진 코드

표 2-10 가중치 코드

84-2-1은 842¹로도 표현한다.

| 10진수 | 8421 코드 (BCD) | 2421 코드 | 5421 코드 | 84-2-1 코드 | 51111 코드 | 바이퀴너리 코드 (biquinary code) 5043210 | 링 카운터 (ring counter) 9876543210 |
|------|------------------|------------|------------|--------------|-------------|---|---------------------------------------|
| 0 | 0000 | 0000 | 0000 | 0000 | 00000 | 0100001 | 0000000001 |
| 1 | 0001 | 0001 | 0001 | 0111 | 00001 | 0100010 | 0000000010 |
| 2 | 0010 | 0010 | 0010 | 0110 | 00011 | 0100100 | 0000000100 |
| 3 | 0011 | 0011 | 0011 | 0101 | 00111 | 0101000 | 0000001000 |
| 4 | 0100 | 0100 | 0100 | 0100 | 01111 | 0110000 | 0000010000 |
| 5 | 0101 | 1011 | 1000 | 1011 | 10000 | 1000001 | 0000100000 |
| 6 | 0110 | 1100 | 1001 | 1010 | 11000 | 1000010 | 0001000000 |
| 7 | 0111 | 1101 | 1010 | 1001 | 11100 | 1000100 | 0010000000 |
| 8 | 1000 | 1110 | 1011 | 1000 | 11110 | 1001000 | 0100000000 |
| 9 | 1001 | 1111 | 1100 | 1111 | 11111 | 1010000 | 1000000000 |

04 디지털 코드

- **비가중치 코드** : 각 위치에 해당하는 값이 없는 코드이며, 이러한 코드들은 데이터 변환 같은 특수한 용도로 사용

표 2-11 비가중치 코드

| 10진수 | 3초과 코드 | 5중 2코드 | 시프트 카운터 | 그레이 코드 |
|------|--------|--------|---------|--------|
| 0 | 0011 | 11000 | 00000 | 0000 |
| 1 | 0100 | 00011 | 00001 | 0001 |
| 2 | 0101 | 00101 | 00011 | 0011 |
| 3 | 0110 | 00110 | 00111 | 0010 |
| 4 | 0111 | 01001 | 01111 | 0110 |
| 5 | 1000 | 01010 | 11111 | 0111 |
| 6 | 1001 | 01100 | 11110 | 0101 |
| 7 | 1010 | 10001 | 11100 | 0100 |
| 8 | 1011 | 10010 | 11000 | 1100 |
| 9 | 1100 | 10100 | 10000 | 1101 |

예제 2-4 10진수 3864를 2421코드로 변환하여라.

풀이

각 자리 별로 변환하면 다음과 같다. 여기서 3, 6, 4는 각각 2가지 경우가 존재한다.

| 3 | 8 | 6 | 4 |
|--------------|------|--------------|--------------|
| 0011 or 1001 | 1110 | 1100 or 0110 | 0100 or 1010 |

End of Example

예제 2-5

74-2-1코드로 표현한 아래의 수를 10진수로 나타내어라.

- ① 0110
- ② 1100
- ③ 1001
- ④ 1011

풀이

$$\textcircled{1} \quad 0110 = 7 \times 0 + 4 \times 1 + (-2) \times 1 + (-1) \times 0 = 4\sim 2 = 2$$

$$\textcircled{2} \quad 1100 = 7 \times 1 + 4 \times 1 + (-2) \times 0 + (-1) \times 0 = 7+4 = 11$$

$$\textcircled{3} \quad 1001 = 7 \times 1 + 4 \times 0 + (-2) \times 0 + (-1) \times 1 = 7\sim 1 = 6$$

$$\textcircled{4} \quad 1011 = 7 \times 1 + 4 \times 0 + (-2) \times 1 + (-1) \times 1 = 7\sim 2\sim 1 = 4$$

End of Example

04 디지털 코드

□ ASCII 코드

- 미국 국립 표준 연구소(ANSI)가 제정한 정보 교환용 미국 표준 코드
- 128가지의 문자를 표현 가능

표 2-13 ASCII 코드

(a) 코드 구성

| b_8 | $b_7 \ b_6 \ b_5$ | $b_4 \ b_3 \ b_2 \ b_1$ |
|-------|-------------------|-------------------------|
| 파리티 | 존 비트 | 디지트 비트 |
| 1 | 3 | 4 |

(b) 표준 ASCII 코드표

| * | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|----|----|----|-----|
| 0 | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS | TAB | LF | VT | FF | CR | SO | SI |
| 1 | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US |
| 2 | ! | " | # | \$ | % | & | ' | (|) | * | + | , | - | . | / | |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5 | P | Q | R | S | T | U | V | W | X | Y | Z | [| ₩ |] | ^ | _ |
| 6 | ' | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7 | p | q | r | s | t | u | v | w | x | y | z | { | | } | ~ | DEL |

□ 유니코드

- ASCII 코드의 한계성을 극복하기 위하여 개발된 인터넷 시대의 표준
- 다양한 운영체제와 최신의 모든 웹 브라우저 및 기타 많은 제품에서 유니코드를 지원한다.
- 미국, 유럽, 동아시아, 아프리카, 아시아 태평양 지역 등의 주요 언어들에 적용될 수 있다.
- 유니코드는 유럽, 중동, 아시아 등 거의 대부분의 문자를 포함하고 있으며, 10만개 이상의 문자로 구성되어 있다.
- 구두표시, 수학기호, 전문기호, 기하학적 모양, 딩벳 기호 등을 포함
- 앞으로도 계속해서 산업계의 요구나 새로운 문자들을 추가하여 나갈 것이다.

05 에러 검출 코드

1 패리티 비트

- 짝수패리티(even parity) : 데이터에서 1의 개수를 짝수 개로 맞춤
- 홀수패리티(odd parity) : 1의 개수를 홀수 개로 맞춤
- 패리티 비트는 데이터 전송과정에서 에러 검사를 위한 추가비트
- 패리티는 단지 에러 검출만 가능하며, 여러 비트에 에러가 발생할 경우에는 검출이 안될 수도 있음

표 2-15 7비트 ASCII 코드에 패리티 비트를 추가한 코드

| 데이터 | 짝수 패리티 | 홀수 패리티 |
|-----|-----------|-----------|
| : | : | : |
| A | 0 1000001 | 1 1000001 |
| B | 0 1000010 | 1 1000010 |
| C | 1 1000011 | 0 1000011 |
| D | 0 1000100 | 1 1000100 |
| : | : | : |

예제 2-6

짝수 패리티 시스템에서 코드 그룹 10110, 11010, 110011, 10101110100, 1100010101011을 수신하였다. 에러가 발생한 그룹을 찾아보아라.

풀이

- 짝수 패리티가 필요하므로 1이 홀수 개인 그룹은 에러 발생
- 10110, 11010, 1100010101011에 비트 에러가 발생하였음

End of Example

05 에러 검출 코드

2 해밍 코드

- 에러를 정정할 수 있는 코드
- 추가적으로 많은 비트가 필요하므로 많은 양의 데이터 전달이 필요
- 데이터 비트와 패리티 비트와의 관계

$$2^p \geq d + p + 1$$

- 예를 들어 $d = 8$ 이면 $2^p \geq 8 + p + 1$ 을 만족하는 p 를 계산하면 4가 된다.
- 해밍코드에서는 짝수 패리티를 사용

표 2-16 해밍 코드에서 패리티 비트의 위치와 패리티 생성 영역

| 비트 위치 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|----------|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|
| 기호 | P_1 | P_2 | D_3 | P_4 | D_5 | D_6 | D_7 | P_8 | D_9 | D_{10} | D_{11} | D_{12} |
| P_1 영역 | ✓ | | ✓ | | ✓ | | ✓ | | ✓ | | ✓ | |
| P_2 영역 | | ✓ | ✓ | | | ✓ | ✓ | | | ✓ | ✓ | |
| P_4 영역 | | | | ✓ | ✓ | ✓ | ✓ | | | | | ✓ |
| P_8 영역 | | | | | | | | ✓ | ✓ | ✓ | ✓ | ✓ |

05 에러 검출 코드

□ 8비트 데이터의 에러 정정 코드

$$P_1 = D_3 \oplus D_5 \oplus D_7 \oplus D_9 \oplus D_{11}$$

$$P_2 = D_3 \oplus D_6 \oplus D_7 \oplus D_{10} \oplus D_{11}$$

$$P_4 = D_5 \oplus D_6 \oplus D_7 \oplus D_{12}$$

$$P_8 = D_9 \oplus D_{10} \oplus D_{11} \oplus D_{12}$$

for example

| P_1 | P_2 | D_3 | P_4 | D_5 | D_6 | D_7 | P_8 | D_9 | D_{10} | D_{11} | D_{12} |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|
| | | 0 | | 0 | 1 | 0 | | 1 | 1 | 1 | 0 |

$$P_1 = D_3 \oplus D_5 \oplus D_7 \oplus D_9 \oplus D_{11} = 0 \oplus 0 \oplus 0 \oplus 1 \oplus 1 = 0$$

$$P_2 = D_3 \oplus D_6 \oplus D_7 \oplus D_{10} \oplus D_{11} = 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 = 1$$

$$P_4 = D_5 \oplus D_6 \oplus D_7 \oplus D_{12} = 0 \oplus 1 \oplus 0 \oplus 0 = 1$$

$$P_8 = D_9 \oplus D_{10} \oplus D_{11} \oplus D_{12} = 1 \oplus 1 \oplus 1 \oplus 0 = 1$$

05 에러 검출 코드

❖ 해밍 코드에서 패리티 비트 생성 과정

표 2-17 원본 데이터가 00101110일 경우 해밍 코드 생성 예

| 비트 위치 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|--------|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|
| 기호 | P_1 | P_2 | D_3 | P_4 | D_5 | D_6 | D_7 | P_8 | D_9 | D_{10} | D_{11} | D_{12} |
| 원본 데이터 | | | 0 | | 0 | 1 | 0 | | 1 | 1 | 1 | 0 |
| 생성된 코드 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |

05 에러 검출 코드

□ 해밍 코드에서 패리티 비트 검사 과정

전송된 데이터 : 010111011110

| P_1 | P_2 | D_3 | P_4 | D_5 | D_6 | D_7 | P_8 | D_9 | D_{10} | D_{11} | D_{12} |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |

☞ 패리티들을 포함하여 검사

$$P_1 = P_1 \oplus D_3 \oplus D_5 \oplus D_7 \oplus D_9 \oplus D_{11} = 0 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 = 1$$

$$P_2 = P_2 \oplus D_3 \oplus D_6 \oplus D_7 \oplus D_{10} \oplus D_{11} = 1 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 = 0$$

$$P_4 = P_4 \oplus D_5 \oplus D_6 \oplus D_7 \oplus D_{12} = 1 \oplus 1 \oplus 1 \oplus 0 \oplus 0 = 1$$

$$P_8 = P_8 \oplus D_9 \oplus D_{10} \oplus D_{11} \oplus D_{12} = 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 0$$

- 검사된 패리티를 $P_8 P_4 P_2 P_1$ 순서대로 정렬
- 모든 패리티가 0이면 에러 없음
- 하나라도 1이 있으면 에러 발생: 결과가 0101이므로 에러 있음
- 0101을 10진수로 바꾸면 5이며, 수신된 데이터에서 앞에서 5번째 비트 0101**1**101110에 에러가 발생한 것이므로 0101**0**1011110으로 바꾸어 주면 에러가 정정된다.

05 에러 검출 코드

표 2-18 해밍 코드에서 에러가 발생한 경우 정정

| 비트 위치 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|--------|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|
| 기호 | P_1 | P_2 | D_3 | P_4 | D_5 | D_6 | D_7 | P_8 | D_9 | D_{10} | D_{11} | D_{12} |
| 해밍 코드 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 패리티 검사 | 1 | 0 | | 1 | | | | 0 | | | | |

$P_8P_4P_2P_1 = 0101_{(2)} = 5_{(10)}$: 5번째 비트에 에러가 발생. 1 → 0으로 정정

| 해밍 코드 수정 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
|----------|---|---|---|---|---|---|---|---|---|---|---|---|
|----------|---|---|---|---|---|---|---|---|---|---|---|---|

05 에러 검출 코드

예제 2-7

다음 해밍 코드 중 에러가 있는지 검사하여라.

1 1 1 1 0 1 0 0 1 0 1 0

풀이

| 비트 위치 | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|----------|---|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|
| 기호 | | P_1 | P_2 | D_3 | P_4 | D_5 | D_6 | D_7 | P_8 | D_9 | D_{10} | D_{11} | D_{12} |
| 해밍 코드 | | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| P_1 계산 | 0 | 1 | | 1 | | 0 | | 0 | | 1 | | 1 | |
| P_2 계산 | 0 | | 1 | 1 | | | 1 | 0 | | | 0 | 1 | |
| P_4 계산 | 0 | | | | 1 | 0 | 1 | 0 | | | | | 0 |
| P_8 계산 | 0 | | | | | | | | 0 | 1 | 0 | 1 | 0 |

$P_8P_4P_2P_1 = 0000$ 이므로 에러 없음

End of Example

05 에러 검출 코드

3 순환 중복 검사(CRC)

- 높은 신뢰도를 확보하며 에러 검출을 위한 오버헤드가 적고, 랜덤 에러나 버스트 에러를 포함한 에러 검출에 매우 좋은 성능을 갖는다.

❖ CRC 발생기 및 검출기

- 수신 측에서는 수신된 $d + k$ 비트의 데이터를 키 값으로 나누었을 때 나머지가 0이면 에러가 없는 것임지만, 0이 아니면 에러가 발생한 것으로 판단한다.

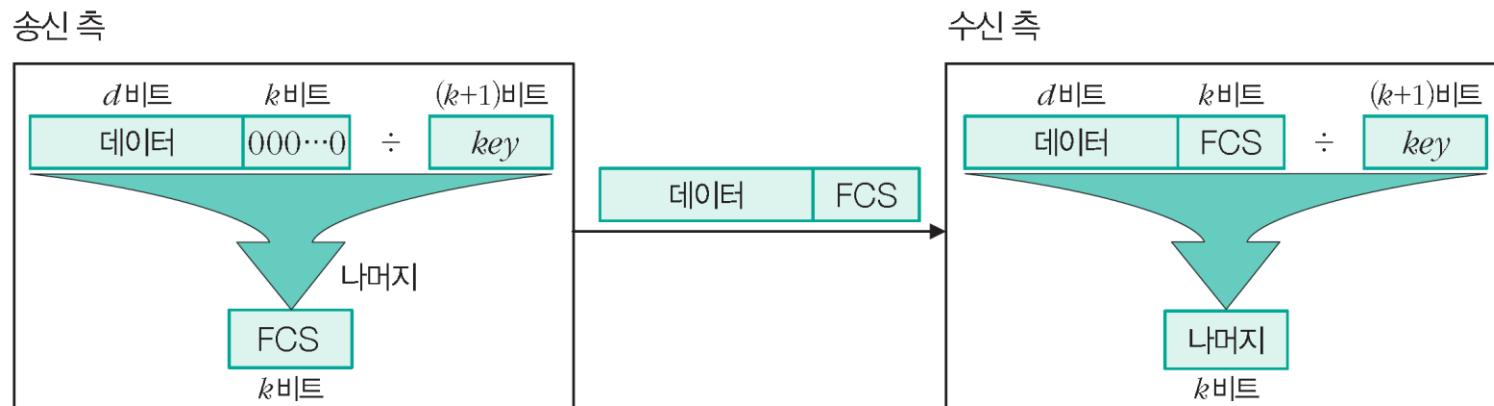


그림 2-8 CRC 발생기와 검사기

cf. FCS: Frame Check Sequence

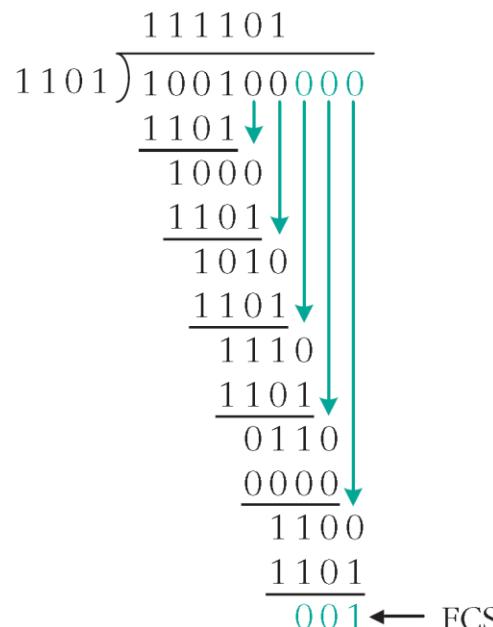
05 에러 검출 코드

❖ CRC 계산에 사용되는 모듈로-2 연산

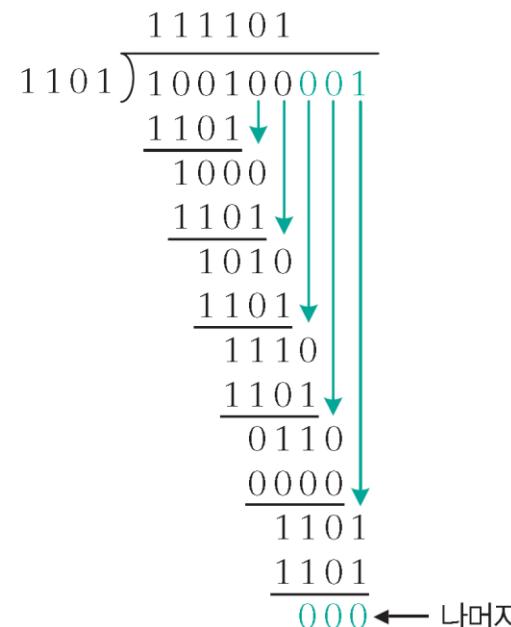
- 사칙 연산에서 캐리는 고려하지 않는다.
- 덧셈 연산은 뺄셈 연산과 결과가 같으며 XOR 연산과도 같다.

- 덧셈 연산: $0+0=0$, $0+1=1$, $1+0=1$, $1+1=0$
- 뺄셈 연산: $0-0=0$, $0-1=1$, $1-0=1$, $1-1=0$

- 데이터가 100100이고, 키 값이 1101인 경우 FCS를 계산하는 예



(a) CRC 발생기에서 계산 과정



(b) 검사기에서 계산 과정

그림 2-9 CRC 발생기와 검사기에서 2진 나눗셈

05 에러 검출 코드

❖ CRC 하드웨어

- 시프트 레지스터와 XOR 게이트(\oplus)를 사용하여 구성할 수 있다.

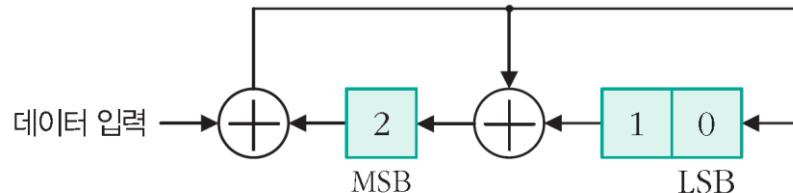


그림 2-10 FCS 생성 회로의 예

- 데이터 100100을 MSB부터 차례로 입력하면, FCS = 001이 된다.

| 데이터 입력 | 2 | 1 | 0 |
|-------------|---|---|---|
| 1 0 0 1 0 0 | 0 | 0 | 0 |
| 1 0 0 1 0 0 | 1 | 0 | 1 |
| 1 0 0 1 0 0 | 1 | 1 | 1 |
| 1 0 0 1 0 0 | 0 | 1 | 1 |
| 1 0 0 1 0 0 | 0 | 1 | 1 |
| 1 0 0 1 0 0 | 1 | 1 | 0 |
| 1 0 0 1 0 0 | 0 | 0 | 1 |

초깃값 ←

05 에러 검출 코드

❖ 생성 다항식

- Key 값을 생성하는 비트를 결정하는 다항식

다항식 표현 $x^7 + x^5 + x^2 + x + 1$

2진수 1 0 1 0 0 1 1 1

- 생성 다항식은 에러 검출 능력을 고려하여 결정되는데, 현재 다음 유형이 사용된다
 - CRC-8: $x^8 + x^2 + x + 1$
 - CRC-16: $x^{16} + x^{12} + x^5 + 1$
 - CRC-32: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$

❖ CRC 테이블 기법

- [그림 2-10]과 같이 비트 단위로 CRC를 계산하면 회로는 간단하지만 처리 속도에 문제가 있다.
- 이를 위해 미리 CRC를 계산하여 메모리에 저장해 놓고 사용한다.

Summary

- 2·8·10·16진수 등의 표현 방법, 상호 변환 방법
- 2진수의 연산과 2진수 음수의 표현 방법
- 부동 소수점 2진수를 IEEE 754 표준 방식으로 표현
- 가중치 코드와 비가중치 코드 이해
- 에러 검출 코드 이해

Microprocessor (W3)

- Logic Circuit Review -

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

Contents

03 조합 논리 회로

04 순서 논리 회로

03 조합 논리 회로

1 데이터 형태에 따른 분류

□ 조합 논리 회로의 개요

- 조합 논리 회로(combational logic circuit)는 현재 입력 값으로 출력이 결정되는 회로

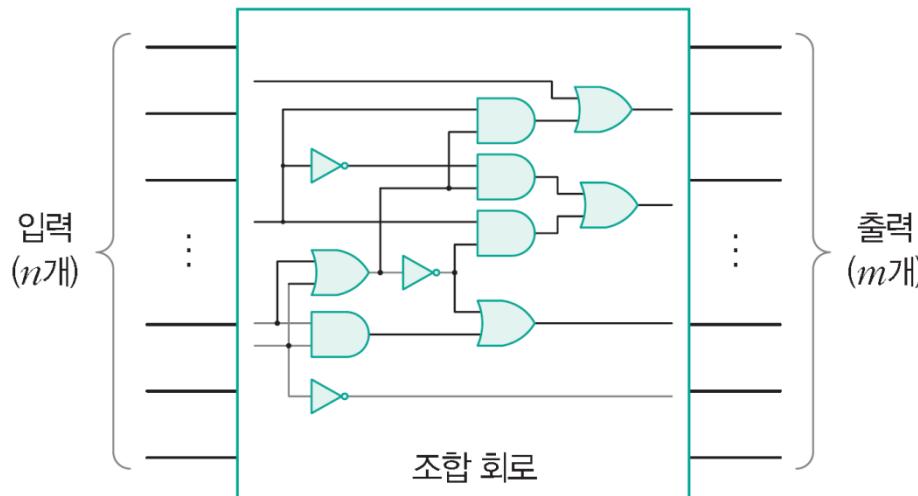


그림 3-32 조합 논리 회로의 개념도

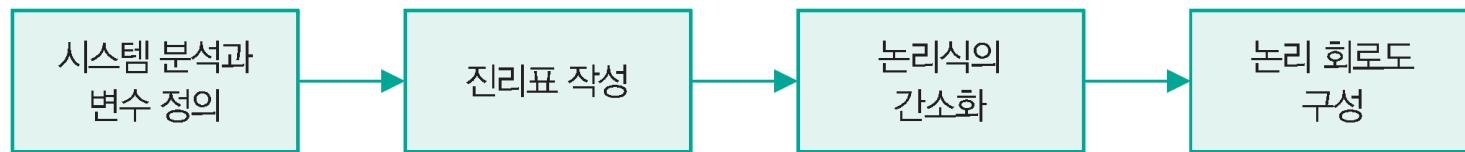


그림 3-33 조합 논리 회로의 설계 과정

03 조합 논리 회로

2 조합 논리 회로의 종류

□ 반가산기

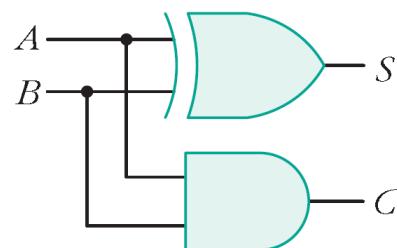
- 반가산기(Half-Adder, HA)는 1자리 2진수 2개를 입력하여 합(S)과 캐리(Carry, C)를 출력하는 조합 논리 회로

$$\begin{array}{r} A & 0 & 0 & 1 & 1 \\ + B & + 0 & + 1 & + 0 & + 1 \\ \hline C & S & 0 & 1 & 0 \\ & & 0 & 1 & 1 \end{array}$$

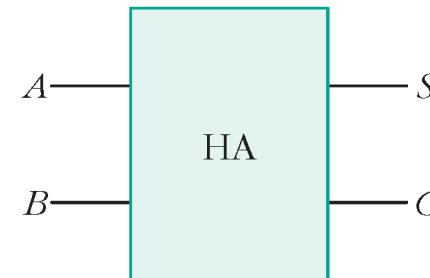
| 입력 | | 출력 | |
|----|---|----|---|
| A | B | S | C |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

$$S = \overline{A}B + A\overline{B} = A \oplus B$$
$$C = AB$$

(a) 진리표와 논리식



(b) 논리 회로



(c) 논리 기호

그림 3-34 반가산기

03 조합 논리 회로

□ 전가산기

- 전가산기(Full-Adder, FA)는 2진수 입력 A, B 와 아랫자리에서 올라온 캐리 C_i 를 포함하여 1자리 2진수 3개를 더하는 조합 논리 회로

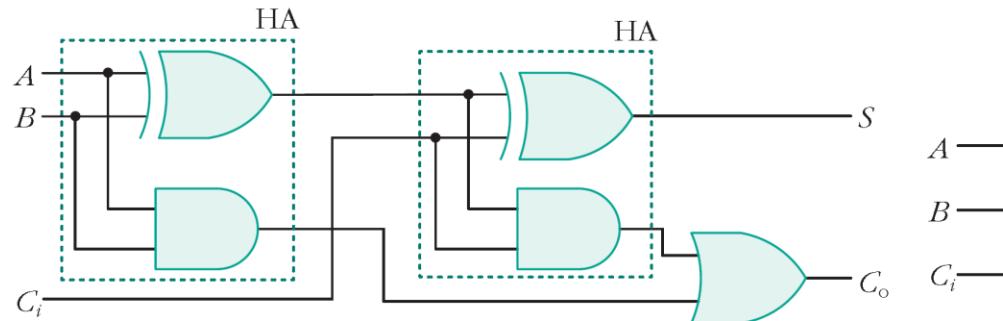
| 입력 | | | 출력 | |
|-----|-----|-------|-----|-------|
| A | B | C_i | S | C_o |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

(a) 진리표

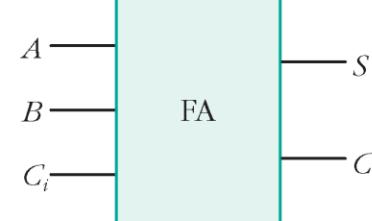
$$\begin{aligned}S &= \overline{ABC}_i + \overline{ABC}_i + A\overline{BC}_i + ABC_i \\&= \overline{A}(\overline{BC}_i + BC_i) + A(\overline{BC}_i + BC_i) \\&= \overline{A}(B \oplus C_i) + A(\overline{B} \oplus \overline{C}_i) \\&= A \oplus (B \oplus C_i) = (A \oplus B) \oplus C_i\end{aligned}$$

$$\begin{aligned}C_o &= \overline{ABC}_i + \overline{ABC}_i + A\overline{BC}_i + ABC_i \\&= C_i(\overline{AB} + AB) + AB(\overline{C}_i + C_i) \\&= C_i(A \oplus B) + AB\end{aligned}$$

(b) 논리식의 간소화



(c) 논리 회로



(d) 논리 기호

그림 3-35 전가산기

03 조합 논리 회로

□ 병렬 가감산기

- **병렬 가산기**(parallel-adder) : 전가산기 여러 개를 병렬로 연결하여 만든 2비트 이상의 가산기

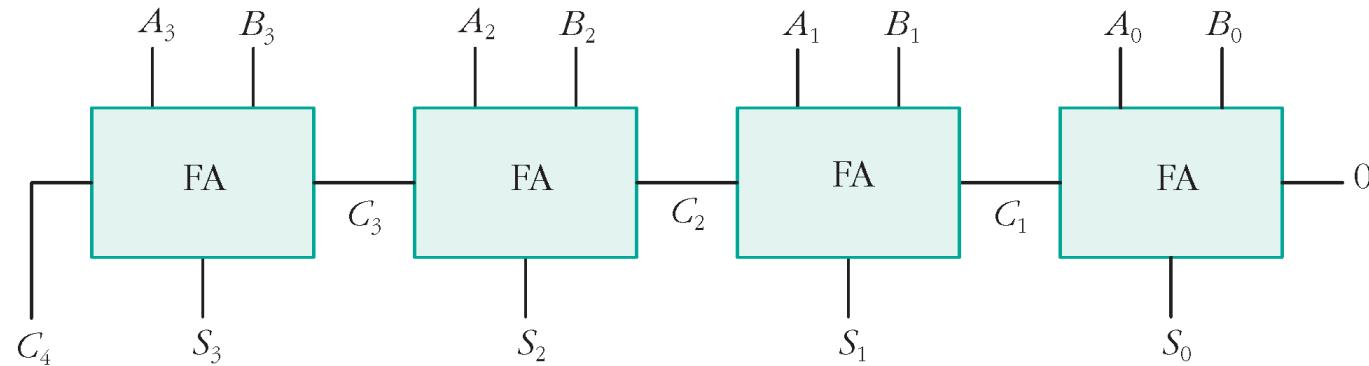


그림 3-38 전가산기를 이용한 병렬 가산기

캐리 예측 가산기(carry lookahead adder)

병렬 가산기는 속도가 매우 느리다. 그 원인은 아랫 단에서 윗단으로 전달되는 캐리 때문이다. 비트가 늘어날수록 지연이 더욱 심해진다. 이를 해결하기 위해 **캐리 예측 가산기**를 사용한다.

03 조합 논리 회로

- **병렬 가감산기**(parallel-adder/subtracter) : 병렬 가산기의 B 입력을 부호 $S(\text{sign})$ 와 XOR하여 전가산기의 입력으로 사용함으로써 덧셈과 뺄셈이 모두 가능한 회로

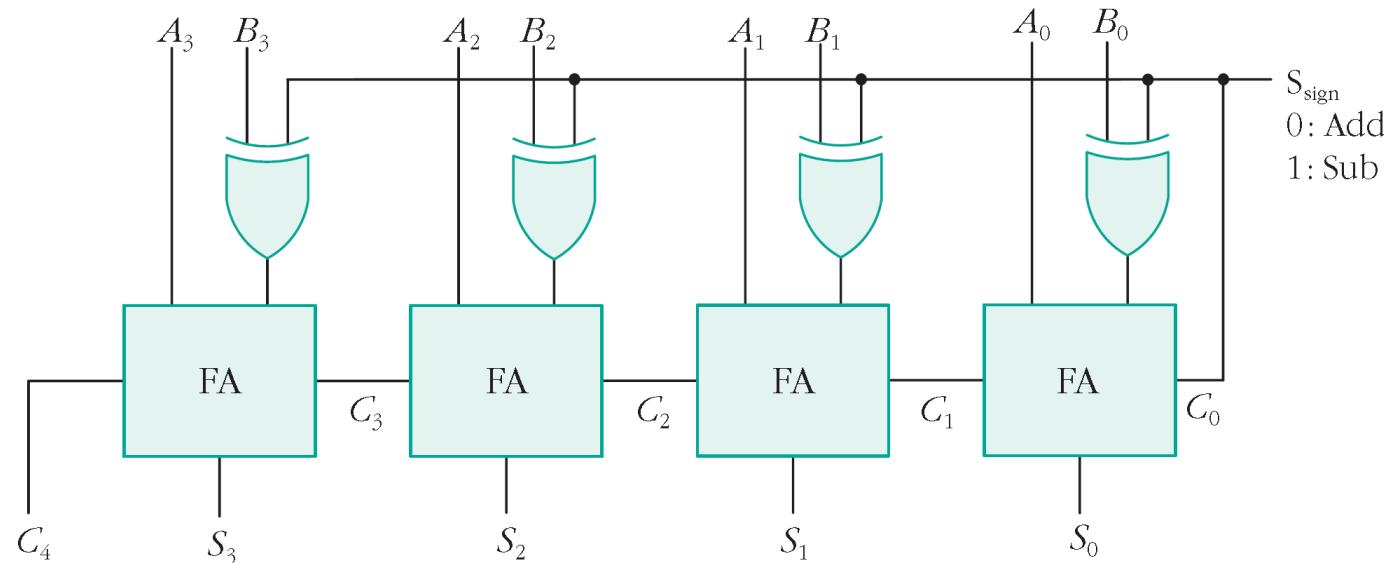


그림 3-39 병렬 가감산기

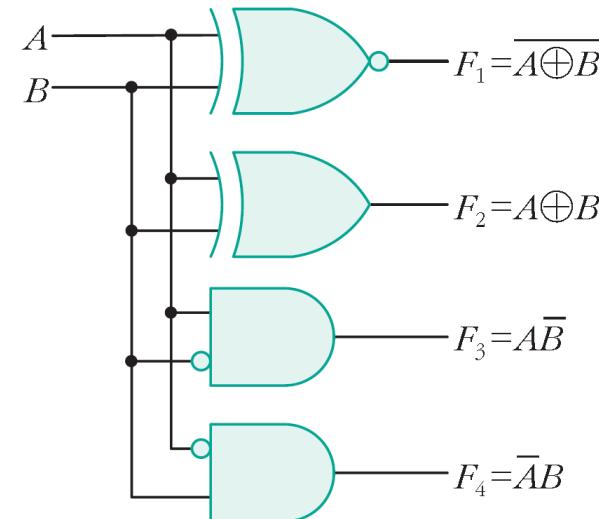
03 조합 논리 회로

□ 비교기

- 2진 비교기(comparator)는 두 2진수 값의 크기를 비교하는 회로다.

| 입력 | | 출력 | | | |
|-----|-----|----------------|---------------------|------------------|------------------|
| A | B | $A=B$ F_1 | $A \neq B$ F_2 | $A > B$ F_3 | $A < B$ F_4 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

(a) 진리표



(b) 논리 회로

그림 3-40 1비트 비교기

03 조합 논리 회로

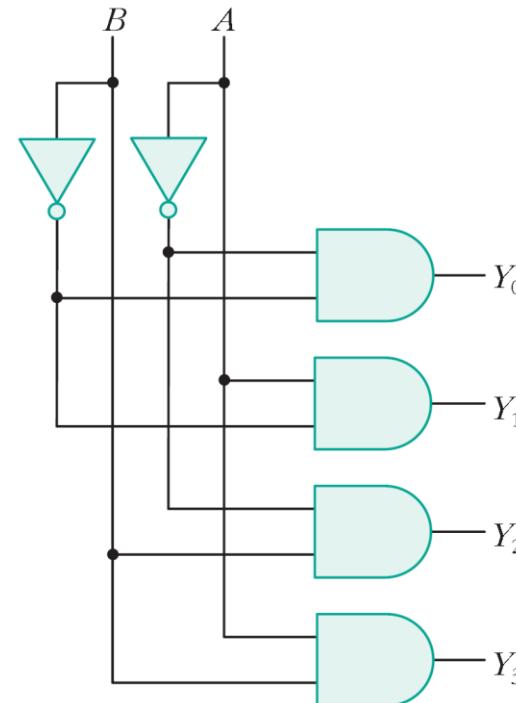
□ 디코더

- **디코더**(decoder) : 입력선에 나타나는 n 비트의 2진 코드를 최대 2^n 개의 서로 다른 정보로 바꿔주는 조합논리회로

| 입력 | | 출력 | | | |
|-----|-----|-------|-------|-------|-------|
| B | A | Y_3 | Y_2 | Y_1 | Y_0 |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

$$Y_0 = \overline{BA}, Y_1 = \overline{B}A, Y_2 = B\overline{A}, Y_3 = BA$$

(a) 진리표와 논리식



(b) 논리 회로

그림 3-41 2×4 디코더

03 조합 논리 회로

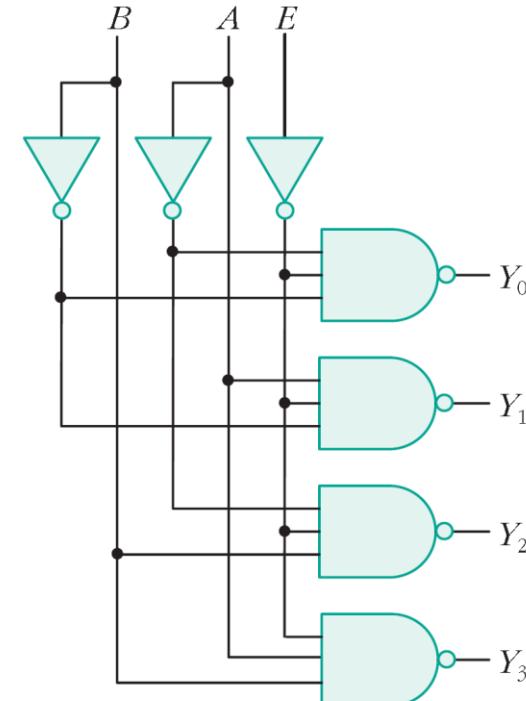
❖ 실제 디코더 회로

- 실제 IC들은 AND 게이트가 아닌 NAND 게이트로 구성되어 출력은 그림과 같이 반대로 된다.
- 또 대부분의 디코더 IC는 인에이블(enable) 입력이 있어 회로를 제어한다.

| 입력 | | | 출력 | | | |
|-----------|-----|-----|-------|-------|-------|-------|
| \bar{E} | B | A | Y_3 | Y_2 | Y_1 | Y_0 |
| 1 | x | x | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 |

$$Y_0 = \overline{\bar{E}\bar{B}\bar{A}}, Y_1 = \overline{\bar{E}\bar{B}A}, Y_2 = \overline{\bar{E}BA}, Y_3 = \overline{\bar{E}BA}$$

(a) 진리표와 논리식



(b) 논리 회로

그림 3-42 인에이블이 있는 2×4 NAND 디코더

03 조합 논리 회로

❖ 3×8 디코더

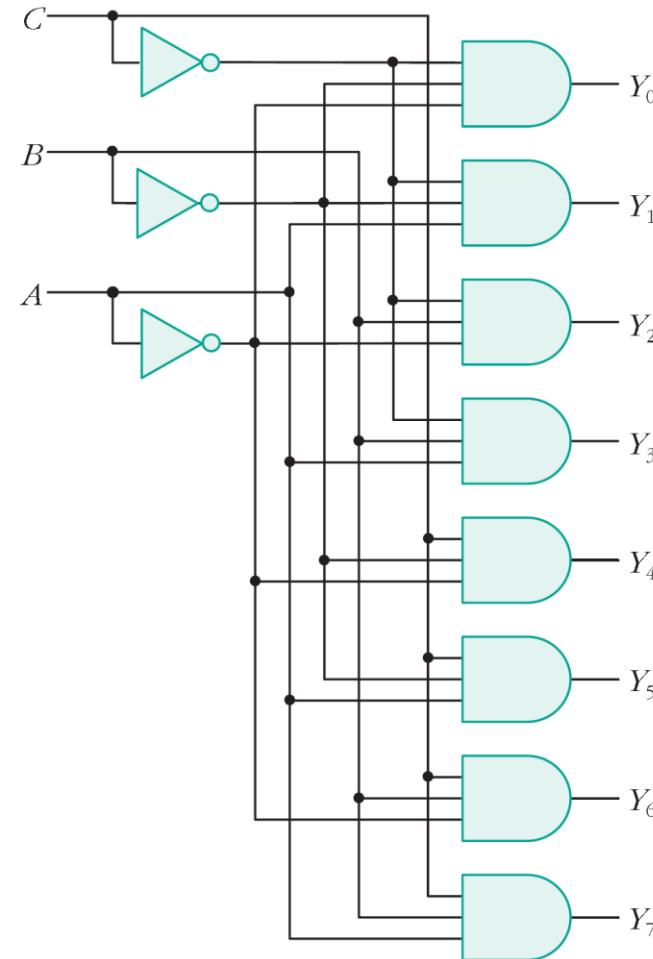
- 3개의 입력에 따라서 8개의 출력 중 하나가 선택

| 입력 | | | 출력 | | | | | | | |
|----|---|---|-------|-------|-------|-------|-------|-------|-------|-------|
| C | B | A | Y_7 | Y_6 | Y_5 | Y_4 | Y_3 | Y_2 | Y_1 | Y_0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

$$Y_0 = \overline{CBA}, Y_1 = \overline{C}\overline{B}A, Y_2 = \overline{C}\overline{B}\overline{A}, Y_3 = \overline{C}BA$$

$$Y_4 = C\overline{B}\overline{A}, Y_5 = C\overline{B}A, Y_6 = C\overline{B}\overline{A}, Y_7 = CBA$$

(a) 진리표와 논리식



(b) 논리 회로

그림 3-43 3×8 디코더 회로

03 조합 논리 회로

❖ 2개의 3×8 디코더로 4×16 디코더를 구성

| | |
|-----|---|
| D=0 | 상위 디코더만 enable되어 출력은 $Y_0 \sim Y_7$ 중의 하나가 1로 되고, 하위 디코더 출력들은 모두 0이 된다. |
| D=1 | 하위 디코더만 enable 되어 출력은 $Y_8 \sim Y_{15}$ 중의 하나가 1로 되고, 상위 디코더 출력들은 모두 0이 된다. |

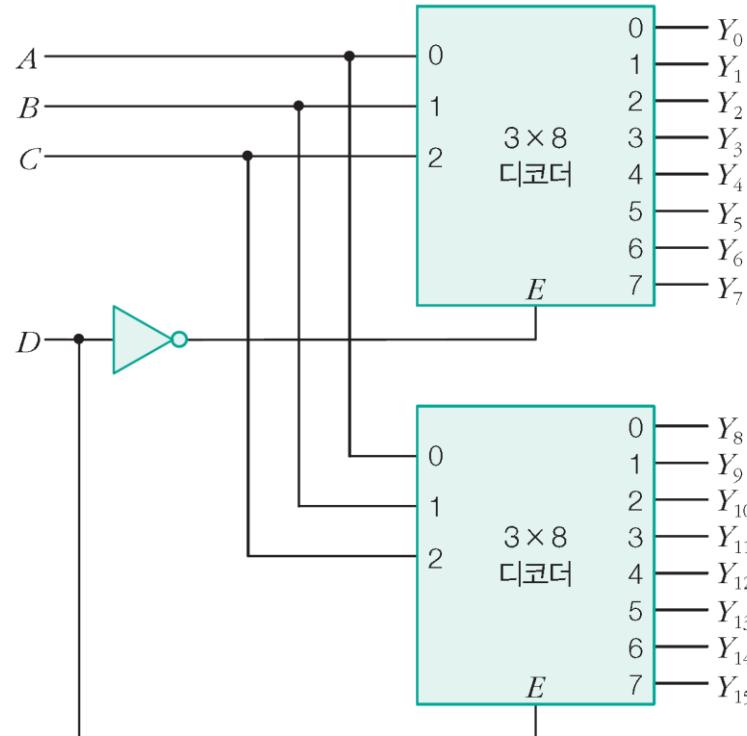


그림 3-44 3×8 디코더 2개를 이용한 4×16 디코더

03 조합 논리 회로

□ 인코더

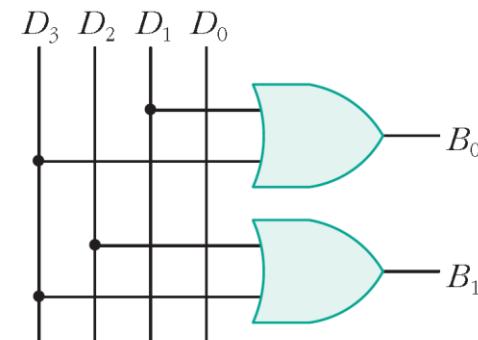
- 부호기라고도 하는 인코더(encoder)는 디코더의 반대 기능을 수행하는 조합 논리 회로로, 2^n 개를 입력받아 n 개를 출력한다.
- 인코더는 2^n 개 중 활성화된 1비트 입력 신호를 받아 그 숫자에 해당하는 n 비트 2진 정보를 출력한다.

| 입력 | | | | 출력 | |
|-------|-------|-------|-------|-------|-------|
| D_3 | D_2 | D_1 | D_0 | B_1 | B_0 |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |

$$B_1 = D_2 + D_3, \quad B_0 = D_1 + D_3$$

(a) 진리표와 논리식

그림 3-45 4×2 인코더



(b) 논리 회로

03 조합 논리 회로

❖ 8×3 인코더

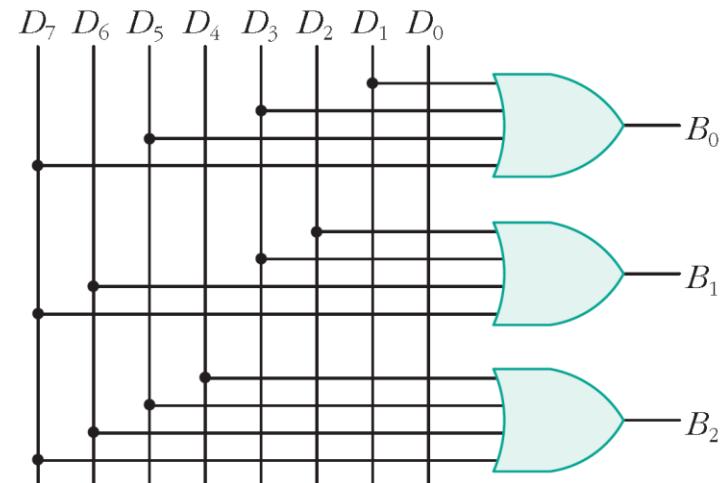
- 8($=2^3$)개의 입력과 3개의 출력을 가지며, 입력의 신호에 따라 3개의 2진 조합으로 출력한다.

| 입력 | | | | | | | | 출력 | | |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| D_7 | D_6 | D_5 | D_4 | D_3 | D_2 | D_1 | D_0 | B_2 | B_1 | B_0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

$$B_2 = D_4 + D_5 + D_6 + D_7, \quad B_1 = D_2 + D_3 + D_6 + D_7$$

$$B_0 = D_1 + D_3 + D_5 + D_7$$

(a) 진리표와 논리식



(b) 논리 회로

그림 3-46 8×3 인코더

03 조합 논리 회로

□ 멀티플렉서

- **멀티플렉서**(multiplexer, MUX)는 여러 개의 입력선들 중에서 하나를 선택하여 출력선에 연결하는 조합논리회로이다. 선택선들의 값에 따라서 특별한 입력선이 선택된다.
- 멀티플렉서는 많은 입력들 중 하나를 선택하여 선택된 입력선의 2진 정보를 출력선에 넘겨주기 때문에 **데이터 선택기**(data selector)라 부르기도 한다.
- **디멀티플렉서**(demultiplexer, DEMUX)는 정보를 한 선으로 받아 2^n 개의 가능한 출력 선들 중 하나를 선택하여, 받은 정보를 전송하는 회로다.

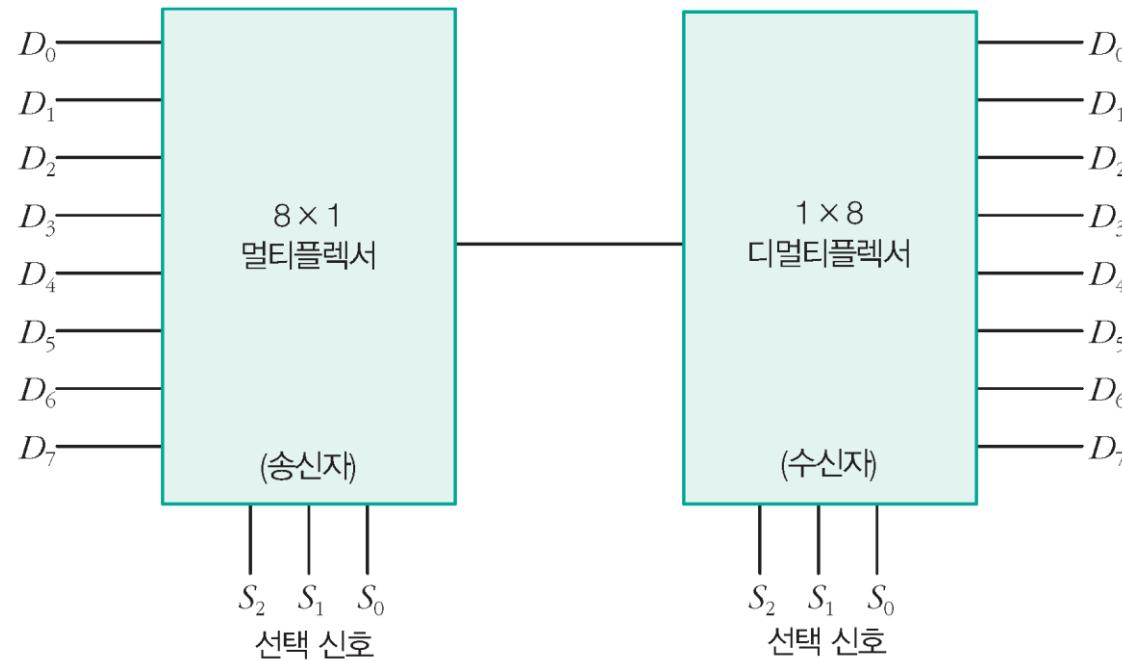


그림 3-47 멀티플렉서와 디멀티플렉서의 역할

03 조합 논리 회로

❖ 2×1 멀티플렉서

- 2($=2^1$)개의 입력중의 하나를 선택선 S 에 입력된 값에 따라서 출력으로 보내주는 조합회로

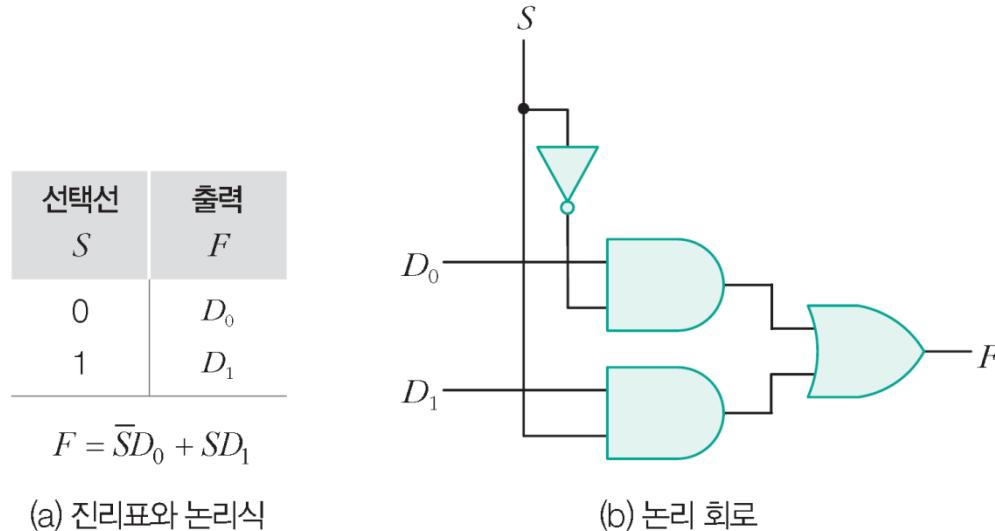


그림 3-48 2×1 멀티플렉서

03 조합 논리 회로

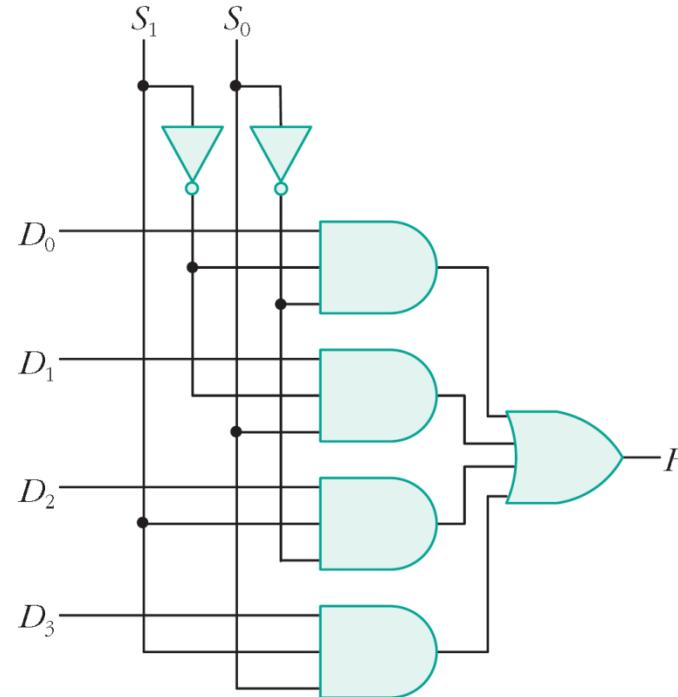
❖ 4×1 멀티플렉서

- 4($=2^2$)개의 입력중의 하나를 선택선 S_1 과 S_0 에 입력된 값에 따라서 출력으로 보내주는 조합회로

| 선택선 | | 출력 |
|-------|-------|-------|
| S_1 | S_0 | F |
| 0 | 0 | D_0 |
| 0 | 1 | D_1 |
| 1 | 0 | D_2 |
| 1 | 1 | D_3 |

$$F = \overline{S}_1 \overline{S}_0 D_0 + \overline{S}_1 S_0 D_1 + S_1 \overline{S}_0 D_2 + S_1 S_0 D_3$$

(a) 진리표와 논리식



(b) 논리 회로

그림 3-49 4×1 멀티플렉서

03 조합 논리 회로

❖ 4×1 멀티플렉서 응용

- 4×1 멀티플렉서를 이용하여 두 입력 A, B 에 대해 AND, OR, XOR, NOT 논리 연산을 수행하는 하드웨어 모듈

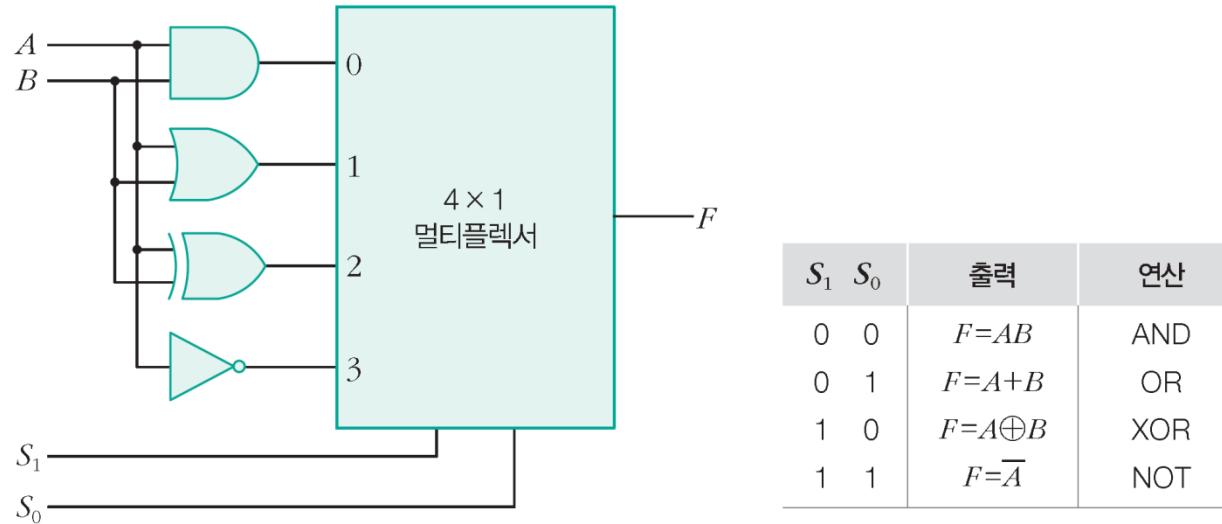


그림 3-50 4×1 멀티플렉서를 이용한 논리 연산 하드웨어 모듈

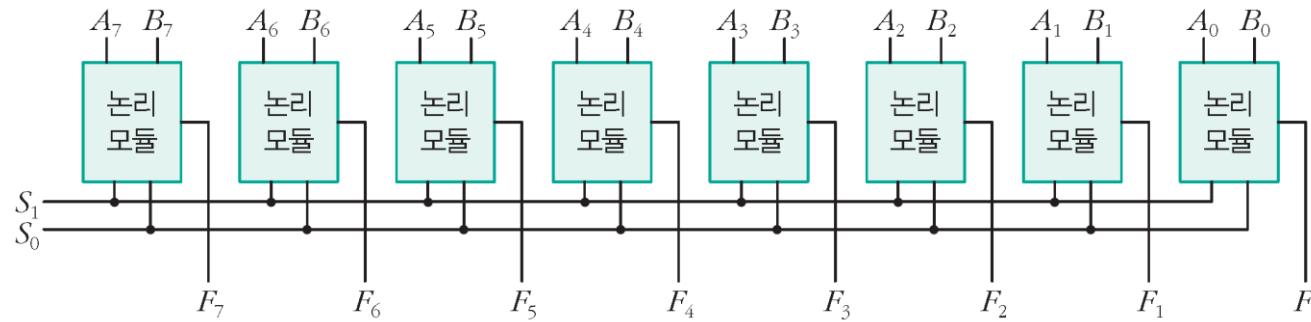


그림 3-51 8비트 논리 연산 장치 구성 예

03 조합 논리 회로

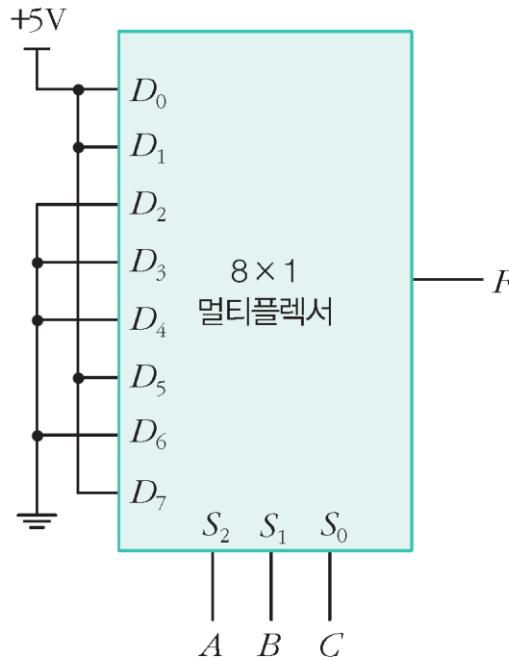
❖ 멀티플렉서를 이용한 조합회로 구현

- $F(A, B, C) = \sum m(0, 1, 5, 7)$ 를 8×1 멀티플렉서로 구현하는 경우
☞ 3개의 선택선을 입력 A, B, C 로 사용

| 입력 | | | 출력 |
|-----|-----|-----|----------|
| A | B | C | F |
| 0 | 0 | 0 | $1(D_0)$ |
| 0 | 0 | 1 | $1(D_1)$ |
| 0 | 1 | 0 | $0(D_2)$ |
| 0 | 1 | 1 | $0(D_3)$ |
| 1 | 0 | 0 | $0(D_4)$ |
| 1 | 0 | 1 | $1(D_5)$ |
| 1 | 1 | 0 | $0(D_6)$ |
| 1 | 1 | 1 | $1(D_7)$ |

$$F(A, B, C) = \sum m(0, 1, 5, 7)$$

(a) 진리표와 논리식



(b) 회로도

그림 3-52 8×1 멀티플렉서를 이용한 회로

03 조합 논리 회로

❖ 멀티플렉서를 이용한 조합회로 구현(계속)

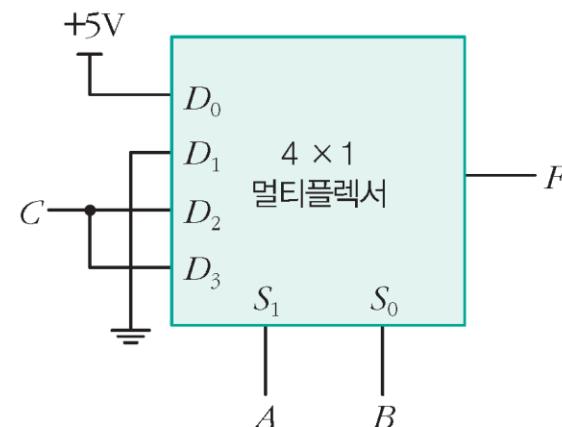
- $F(A, B, C) = \sum m(0, 1, 5, 7)$ 를 4×1 멀티플렉서로 구현하는 경우

☞ A, B는 선택선으로 C는 D_0, D_1, D_2, D_3 을 조합하여 사용

| 입력 | | 출력 | |
|----|---|----|-----------|
| A | B | C | F |
| 0 | 0 | 0 | $D_0 = 1$ |
| | | 1 | 1 |
| 0 | 1 | 0 | $D_1 = 0$ |
| | | 1 | 0 |
| 1 | 0 | 0 | $D_2 = C$ |
| | | 1 | 1 |
| 1 | 1 | 0 | $D_3 = C$ |
| | | 1 | 1 |

$$F(A, B, C) = \sum m(0, 1, 5, 7)$$

(a) 진리표와 논리식



(b) 회로도

그림 3-53 4×1 멀티플렉서를 이용한 회로

03 조합 논리 회로

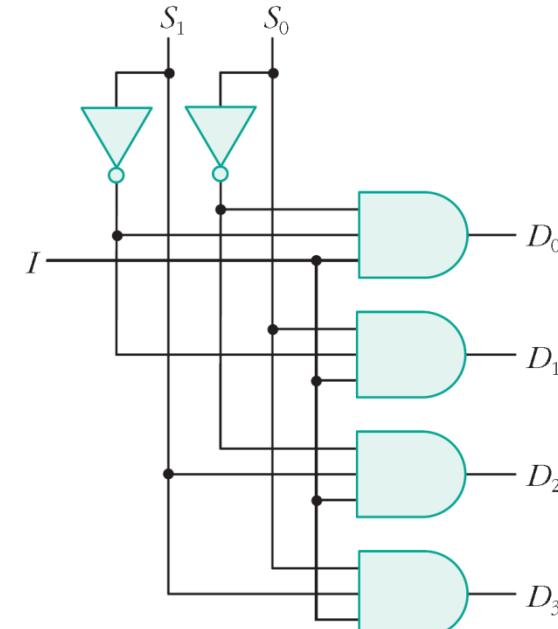
□ 디멀티플렉서

- **디멀티플렉서**(demultiplexer)는 하나의 입력선에 데이터를 입력하면 선택선 n 개로 2^n 개 중 하나를 출력하는 조합 논리 회로다. **데이터 분배기**라고도 한다.
- 1×4 디멀티플렉서는 선택선(S_1, S_0) 2개로 출력(D_3, D_2, D_1, D_0) 4개 중 하나를 선택해 입력(I)을 연결한다.

| 선택선 | | 출력 | | | |
|-------|-------|-------|-------|-------|-------|
| S_1 | S_0 | D_3 | D_2 | D_1 | D_0 |
| 0 | 0 | 0 | 0 | 0 | I |
| 0 | 1 | 0 | 0 | I | 0 |
| 1 | 0 | 0 | I | 0 | 0 |
| 1 | 1 | I | 0 | 0 | 0 |

$$D_0 = \overline{S}_1 \overline{S}_0 I, D_1 = \overline{S}_1 S_0 I, D_2 = S_1 \overline{S}_0 I, D_3 = S_1 S_0 I$$

(a) 진리표와 논리식



(b) 논리 회로

그림 3-54 1×4 디멀티플렉서

03 조합 논리 회로

□ 코드 변환기 (2진 코드 → 그레이 코드 변환)

| 2진 코드(입력) $B_3\ B_2\ B_1\ B_0$ | 그레이 코드(출력) $G_3\ G_2\ G_1\ G_0$ |
|-----------------------------------|------------------------------------|
| 0 0 0 0 | 0 0 0 0 |
| 0 0 0 1 | 0 0 0 1 |
| 0 0 1 0 | 0 0 1 1 |
| 0 0 1 1 | 0 0 1 0 |
| 0 1 0 0 | 0 1 1 0 |
| 0 1 0 1 | 0 1 1 1 |
| 0 1 1 0 | 0 1 0 1 |
| 0 1 1 1 | 0 1 0 0 |
| 1 0 0 0 | 1 1 0 0 |
| 1 0 0 1 | 1 1 0 1 |
| 1 0 1 0 | 1 1 1 1 |
| 1 0 1 1 | 1 1 1 0 |
| 1 1 0 0 | 1 0 1 0 |
| 1 1 0 1 | 1 0 1 1 |
| 1 1 1 0 | 1 0 0 1 |
| 1 1 1 1 | 1 0 0 0 |

(a) 진리표

| B_1B_0 | 00 | 01 | 11 | 10 |
|----------|----|---------|----|----|
| B_3B_2 | 00 | | | |
| | 01 | | | |
| | 11 | 1 1 1 1 | | |
| | 10 | 1 1 1 1 | | |

$$G_3 = B_3$$

| B_1B_0 | 00 | 01 | 11 | 10 |
|----------|----|-----|-----|----|
| B_3B_2 | 00 | | 1 1 | 1 |
| | 01 | 1 1 | | |
| | 11 | 1 1 | | |
| | 10 | | 1 1 | 1 |

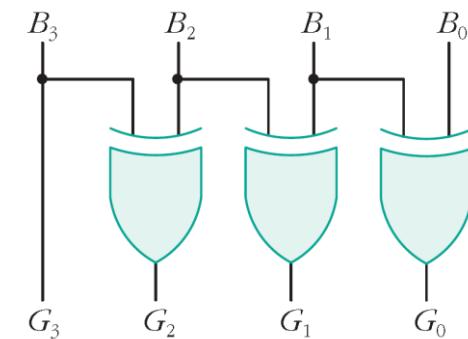
$$G_1 = \bar{B}_2B_1 + B_2\bar{B}_1 \\ = B_2 \oplus B_1$$

| B_1B_0 | 00 | 01 | 11 | 10 |
|----------|----|----|----|----|
| B_3B_2 | 00 | | | |
| | 01 | 1 | 1 | 1 |
| | 11 | | | |
| | 10 | 1 | 1 | 1 |

$$G_2 = \bar{B}_3B_2 + B_3\bar{B}_2 \\ = B_3 \oplus B_2$$

| B_1B_0 | 00 | 01 | 11 | 10 |
|----------|----|----|----|----|
| B_3B_2 | 00 | 1 | | 1 |
| | 01 | 1 | | 1 |
| | 11 | 1 | | 1 |
| | 10 | 1 | | 1 |

$$G_0 = \bar{B}_1B_0 + B_1\bar{B}_0 \\ = B_1 \oplus B_0$$



(c) 논리 회로

그림 3-55 2진 코드를 그레이 코드로 변환하는 회로

03 조합 논리 회로

□ 코드 변환기 (그레이 코드 → 2진 코드 변환)

| 그레이 코드(입력) $G_3\ G_2\ G_1\ G_0$ | 2진 코드(출력) $B_3\ B_2\ B_1\ B_0$ |
|------------------------------------|-----------------------------------|
| 0 0 0 0 | 0 0 0 0 |
| 0 0 0 1 | 0 0 0 1 |
| 0 0 1 0 | 0 0 1 1 |
| 0 0 1 1 | 0 0 1 0 |
| 0 1 0 0 | 0 1 1 1 |
| 0 1 0 1 | 0 1 1 0 |
| 0 1 1 0 | 0 1 0 0 |
| 0 1 1 1 | 0 1 0 1 |
| 1 0 0 0 | 1 1 1 1 |
| 1 0 0 1 | 1 1 1 0 |
| 1 0 1 0 | 1 1 0 0 |
| 1 0 1 1 | 1 1 0 1 |
| 1 1 0 0 | 1 0 0 0 |
| 1 1 0 1 | 1 0 0 1 |
| 1 1 1 0 | 1 0 1 1 |
| 1 1 1 1 | 1 0 1 0 |

| G_3G_2 | 00 | 01 | 11 | 10 |
|----------|----|---------|----|----|
| G_3G_0 | 00 | | | |
| G_3G_2 | 01 | | | |
| G_3G_0 | 11 | 1 1 1 1 | | |
| G_3G_0 | 10 | 1 1 1 1 | | |

$$B_3 = G_3$$

| G_3G_2 | 00 | 01 | 11 | 10 |
|----------|----|-----|-----|----|
| G_3G_0 | 00 | | | |
| G_3G_2 | 01 | 1 1 | | |
| G_3G_0 | 11 | | 1 1 | |
| G_3G_0 | 10 | 1 1 | | |

$$\begin{aligned} B_1 &= G_3 \oplus G_2 \oplus G_1 \\ &= B_2 \oplus G_1 \end{aligned}$$

| G_3G_2 | 00 | 01 | 11 | 10 |
|----------|----|---------|---------|----|
| G_3G_0 | 00 | | | |
| G_3G_2 | 01 | 1 1 1 1 | | |
| G_3G_0 | 11 | | 1 1 1 1 | |
| G_3G_0 | 10 | 1 1 1 1 | | |

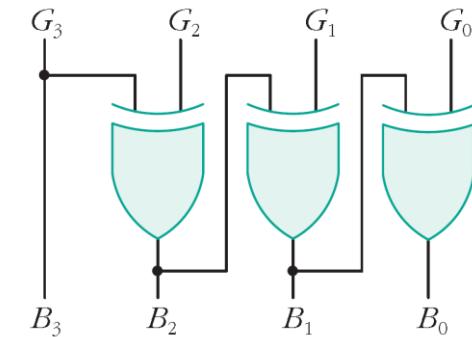
$$\begin{aligned} B_2 &= \bar{G}_3G_2 + G_3\bar{G}_2 \\ &= G_3 \oplus G_2 \end{aligned}$$

| G_3G_2 | 00 | 01 | 11 | 10 |
|----------|----|----|----|----|
| G_3G_0 | 00 | | | |
| G_3G_2 | 01 | 1 | 1 | 1 |
| G_3G_0 | 11 | 1 | 1 | 1 |
| G_3G_0 | 10 | 1 | 1 | |

$$\begin{aligned} B_0 &= G_3 \oplus G_2 \oplus G_1 \oplus G_0 \\ &= B_1 \oplus G_0 \end{aligned}$$

(a) 진리표

(b) 논리식의 간소화



(c) 논리 회로

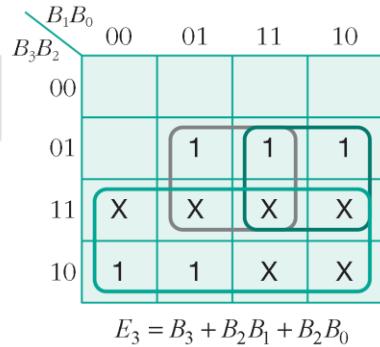
그림 3-56 그레이 코드를 2진 코드로 변환하는 회로

03 조합 논리 회로

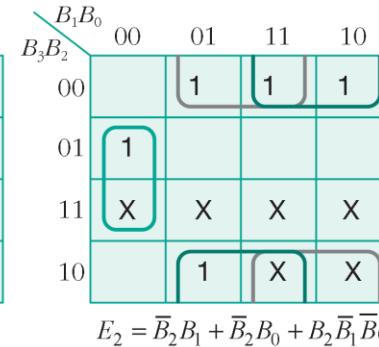
□ 코드 변환기 (BCD 코드 → 3초과 코드 변환)

| BCD 코드(입력) $B_3\ B_2\ B_1\ B_0$ | 3초과 코드(출력) $E_3\ E_2\ E_1\ E_0$ |
|------------------------------------|------------------------------------|
| 0 0 0 0 | 0 0 1 1 |
| 0 0 0 1 | 0 1 0 0 |
| 0 0 1 0 | 0 1 0 1 |
| 0 0 1 1 | 0 1 1 0 |
| 0 1 0 0 | 0 1 1 1 |
| 0 1 0 1 | 1 0 0 0 |
| 0 1 1 0 | 1 0 0 1 |
| 0 1 1 1 | 1 0 1 0 |
| 1 0 0 0 | 1 0 1 1 |
| 1 0 0 1 | 1 1 0 0 |
| 1 0 1 0 | x x x x |
| 1 0 1 1 | x x x x |
| 1 1 0 0 | x x x x |
| 1 1 0 1 | x x x x |
| 1 1 1 0 | x x x x |
| 1 1 1 1 | x x x x |

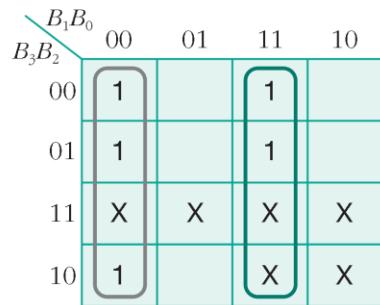
(a) 진리표



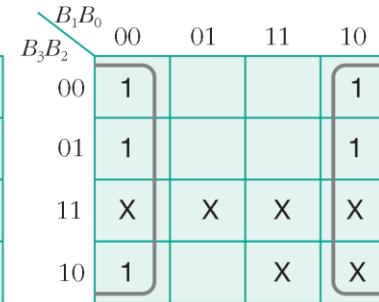
$$E_3 = B_3 + B_2B_1 + B_2B_0$$



$$E_2 = \bar{B}_2B_1 + \bar{B}_2B_0 + B_2\bar{B}_1\bar{B}_0$$

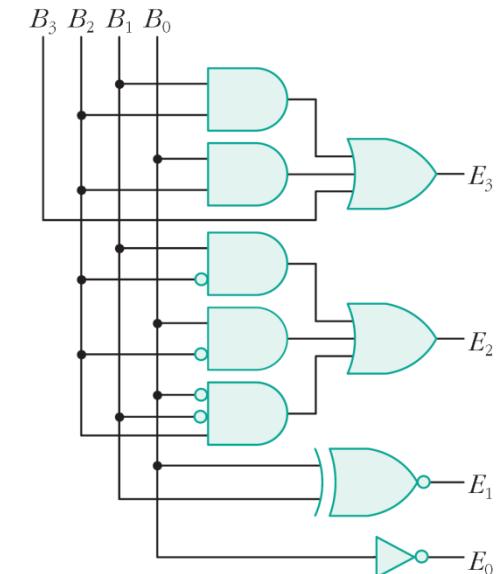


$$E_1 = \bar{B}_1\bar{B}_0 + B_1B_0 \\ = \bar{B}_1 \oplus B_0$$



$$E_0 = \bar{B}_0$$

(b) 논리식의 간소화



(c) 논리 회로

그림 3-57 BCD 코드를 3초과 코드로 변환하는 회로

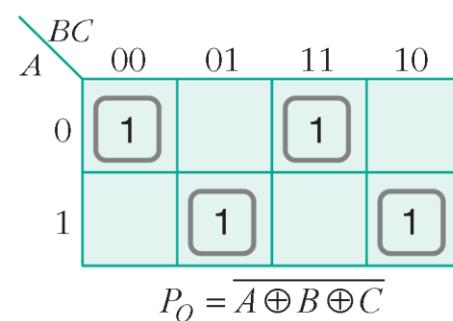
03 조합 논리 회로

□ 패리티 발생기와 검사기

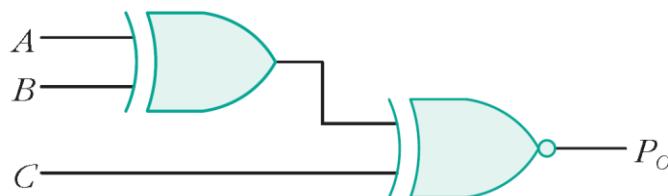
❖ 패리티 발생기

| 입력 | | | 출력 | |
|----|---|---|------------|------------|
| A | B | C | P_O (홀수) | P_E (짝수) |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 |

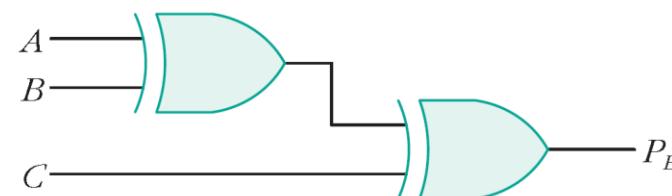
(a) 진리표



(b) 논리식의 간소화



(c) 홀수 패리티 발생기 논리 회로



(d) 짝수 패리티 발생기 논리 회로

그림 3-58 패리티 발생기

03 조합 논리 회로

❖ 패리티 검사기

| 입력 | | | | 출력 | 입력 | | | | 출력 |
|-----|-----|-----|------------|------------|-----|-----|-----|------------|------------|
| A | B | C | P_O (홀수) | Y_O (홀수) | A | B | C | P_E (짝수) | Y_E (짝수) |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

(a) 진리표

03 조합 논리 회로

❖ 패리티 검사기(계속)

- 패리티 검사기 출력이 $Y = 0$ 이면 에러가 발생하지 않았다고 판단하고, $Y = 1$ 이면 에러가 발생했다고 판단한다.

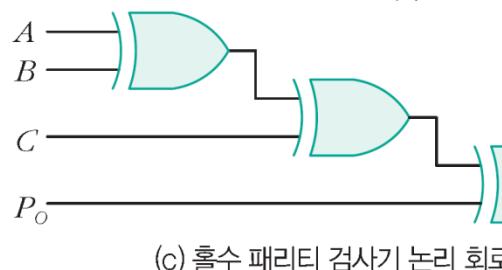
| | | CP_O | | | |
|----|----|--------|----|----|----|
| | | 00 | 01 | 11 | 10 |
| AB | 00 | 1 | | 1 | |
| | 01 | | 1 | | 1 |
| AB | 11 | 1 | | 1 | |
| | 10 | | 1 | | 1 |

$$Y_O = \overline{A \oplus B \oplus C \oplus P_O}$$

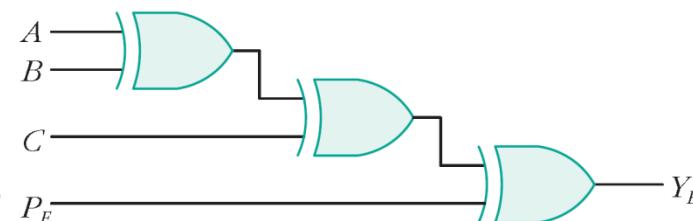
| | | CP_E | | | |
|----|----|--------|----|----|----|
| | | 00 | 01 | 11 | 10 |
| AB | 00 | | 1 | | 1 |
| | 01 | 1 | | 1 | |
| AB | 11 | | 1 | | 1 |
| | 10 | 1 | | 1 | |

$$Y_E = A \oplus B \oplus C \oplus P_E$$

(b) 논리식의 간소화



(c) 홀수 패리티 검사기 논리 회로



(d) 짝수 패리티 검사기 논리 회로

그림 3-59 패리티 검사기

03 조합 논리 회로

❖ 데이터 전송 시스템

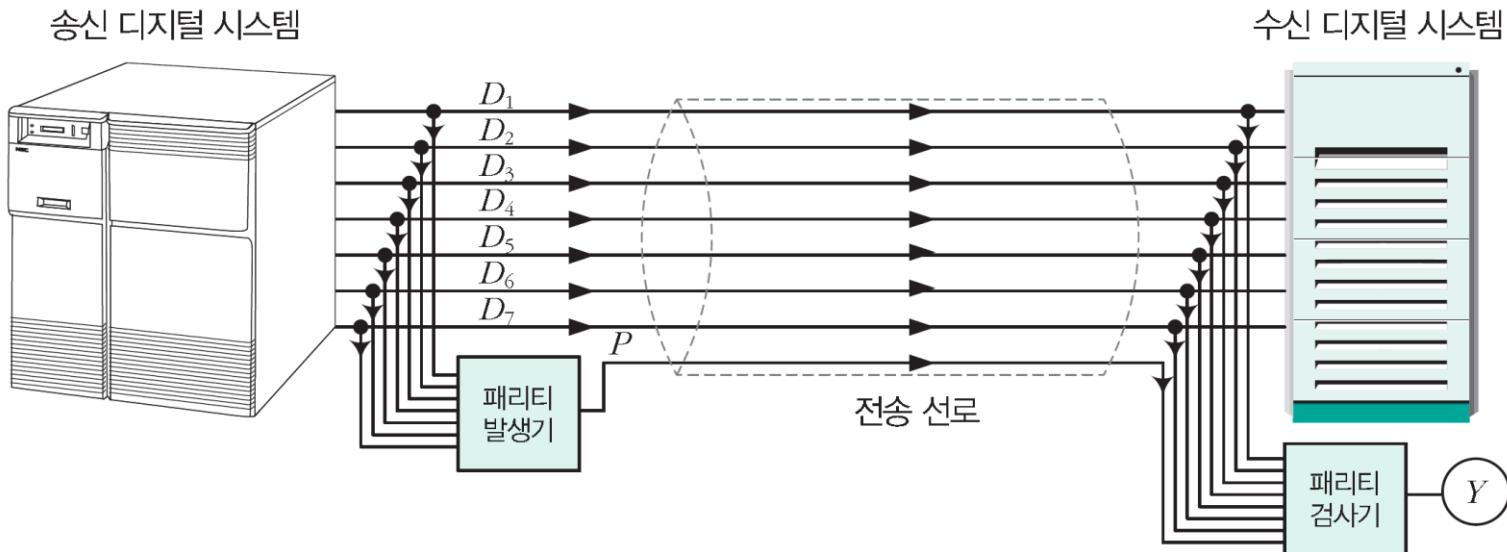


그림 3-60 데이터 전송 시스템에서 패리티 비트를 이용한 에러 검출

04 순서 논리 회로

1 순서 논리 회로의 개요

- 조합 논리 회로(combational logic circuit) : 이전 입력 값에 관계없이 현재 입력 값에 따라 출력이 결정
- 순서 논리 회로(sequential logic circuit) : 현재의 입력 값과 이전 출력 상태에 따라 출력 값이 결정

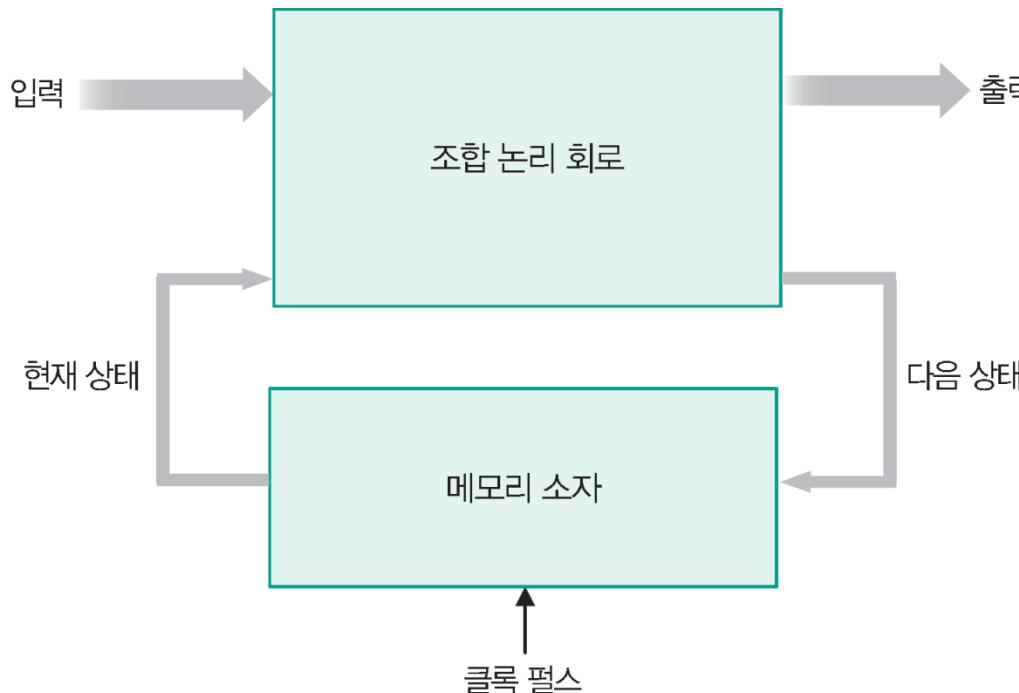


그림 3-62 순서 논리 회로의 구성

04 순서 논리 회로

❖ 순서 논리 회로의 특징

- 순서 논리 회로의 출력은 외부에서 들어온 입력과 이전 출력 상태에 따라 결정된다. 이러한 동작은 클록 펄스가 들어올 때마다 반복해서 일어난다.
- 순서 논리 회로는 기억 기능(플립플롭)이 있다.
- 대표적인 순서 논리 회로에는 플립플롭, 카운터, 레지스터 등이 있다.

□ 클록 펄스

- **플립플롭**(flip-flop)은 **클록 펄스**(Clock Pulse, CP)라는 제어 입력을 가지며, 출력은 클록 펄스에 동기되어 변하고 이러한 변화를 트리거(trigger)되었다고 한다.

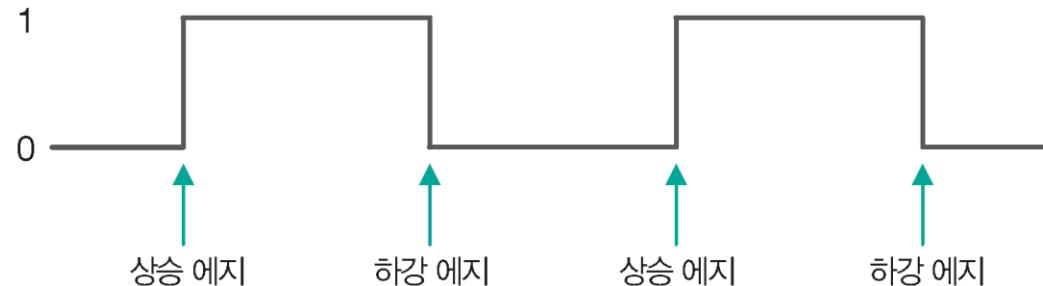
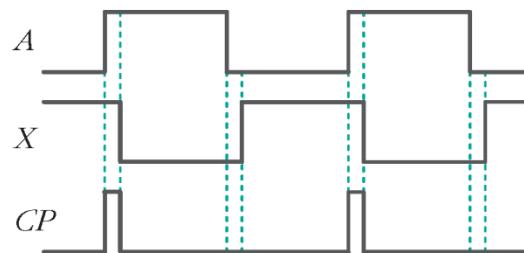
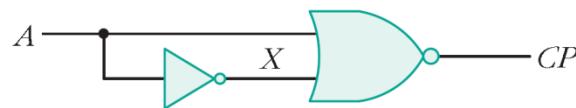
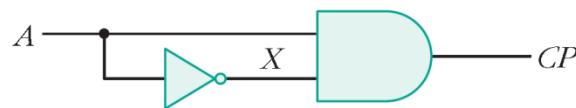


그림 3-63 클록 펄스의 형태

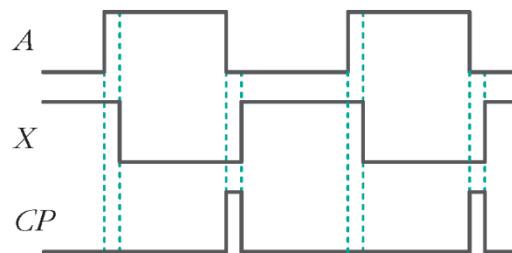
04 순서 논리 회로

❖ 펄스 전이 검출기

- 실제 회로에서 클록 펄스는 상승 에지나 하강 에지에서 순간적으로 1이 되는 날카로운 파형을 만들어 플립플롭을 동작시키는데, 이 파형을 에지 트리거(edge trigger)라고 한다.
- 클록 펄스(구형파)로 에지 트리거를 만들려면 **펄스 전이 검출기**가 필요하다



(a) 상승 에지 트리거



(b) 하강 에지 트리거

그림 3-64 펄스 전이 검출기 구조

04 순서 논리 회로

□ 플립플롭의 종류

❖ 플립플롭의 특징

- 플립플롭은 1비트의 정보를 기억할 수 있는 기억 소자다.
- 플립플롭은 제어 입력인 클록 펄스가 있으며 다음 클록 펄스가 들어올 때까지 현재 상태를 유지 한다.
- 플립플롭은 Q 와 \bar{Q} 로 표시된 출력이 2개 있으며 Q 와 \bar{Q} 의 상태는 서로 보수가 되어야 정상 상태 가 된다.
- 플립플롭은 RAM의 구성 요소로도 사용된다.
- 플립플롭에는 SR 플립플롭, JK 플립플롭, D 플립플롭, T 플립플롭이 있다.

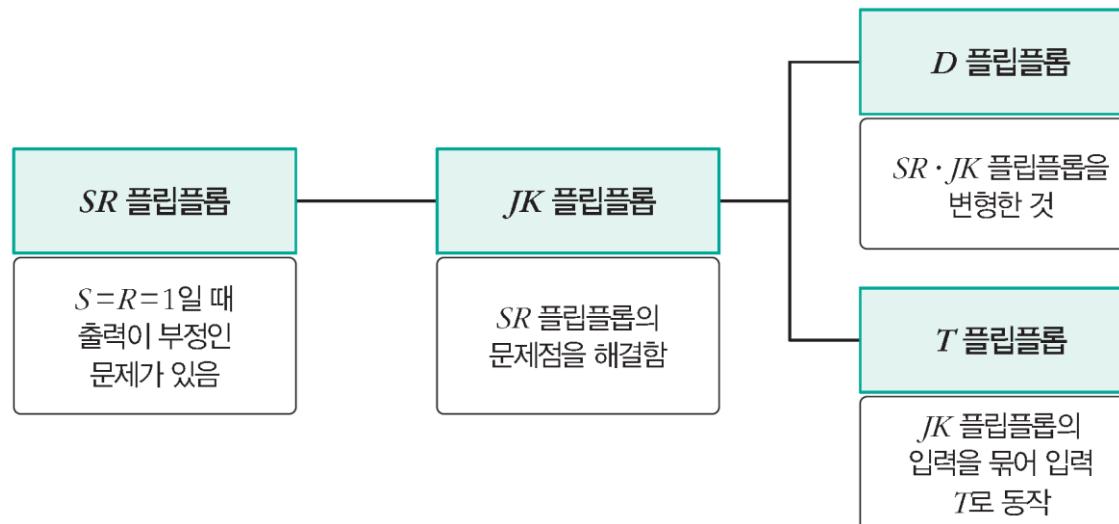
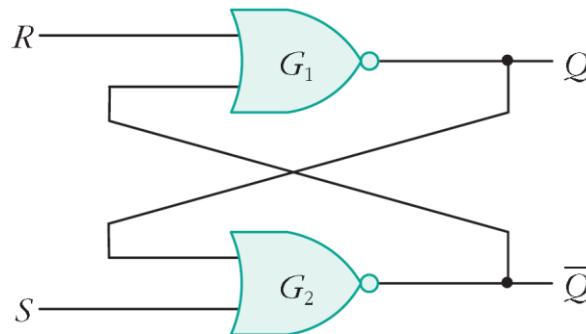


그림 3-65 플립플롭의 종류

04 순서 논리 회로

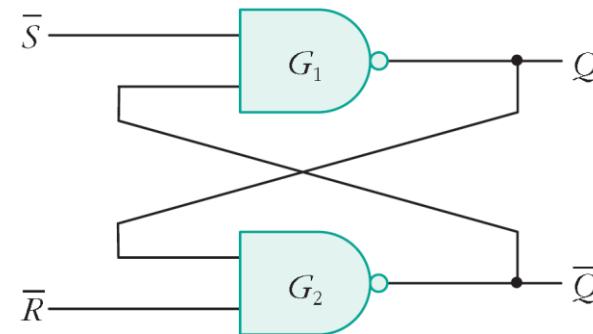
2 플립플롭

□ SR 래치



| S | R | $Q(t+1)$ |
|-----|-----|-------------|
| 0 | 0 | $Q(t)$ (불변) |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 부정 |

(a) NOR 게이트로 구성



| \bar{S} | \bar{R} | $Q(t+1)$ |
|-----------|-----------|-------------|
| 0 | 0 | 부정 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | $Q(t)$ (불변) |

(b) NAND 게이트로 구성

그림 3-66 SR 래치의 논리 회로와 진리표

04 순서 논리 회로

□ SR 플립플롭

- SR 플립플롭 : 클록 펄스가 있을 때만 동작하는 SR 래치를 의미
- 클록 펄스는 **상승 에지 트리거 신호**가 입력된 경우다.

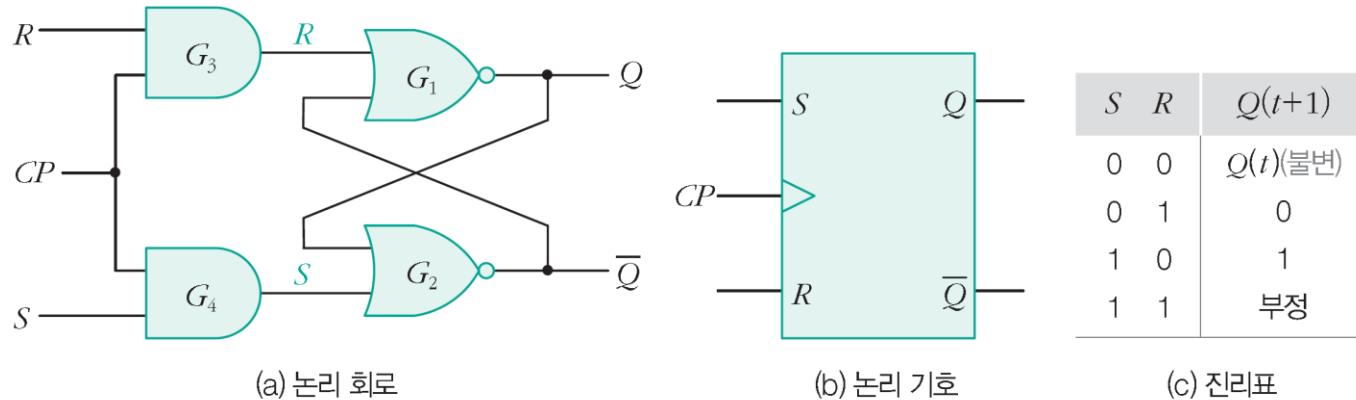


그림 3-67 SR 플립플롭의 구조

| | |
|----------|---|
| CP=0인 경우 | S와 R의 입력에 관계없이 앞단의 AND 게이트 \$G_3\$과 \$G_4\$의 출력이 항상 0이므로 플립플롭의 출력은 불변 |
| CP=1인 경우 | S와 R의 입력이 회로 후단의 NOR 게이트 \$G_1\$과 \$G_2\$의 입력으로 전달되어 SR 래치와 같은 동작을 수행 |

04 순서 논리 회로

❖ SR 플립플롭의 특성표 및 특성 방정식

| S | R | $Q(t)$ | $Q(t+1)$ |
|---|---|--------|----------|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 부정 |
| 1 | 1 | 1 | 부정 |

(a) 특성표

| | | $RQ(t)$ | | | |
|---|--|---------|----|----|----|
| | | 00 | 01 | 11 | 10 |
| S | | 0 | 1 | | |
| 0 | | | | | |
| 1 | | 1 | 1 | X | X |

$$Q(t+1) = S + \bar{R}Q(t), SR = 0$$

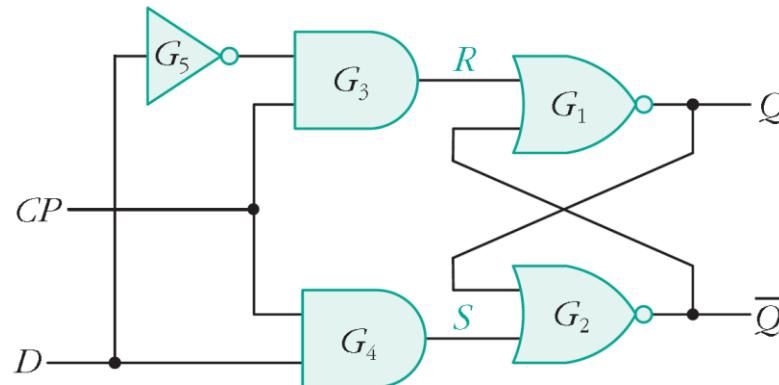
(b) 특성 방정식

그림 3-68 SR 플립플롭의 특성표와 특성 방정식

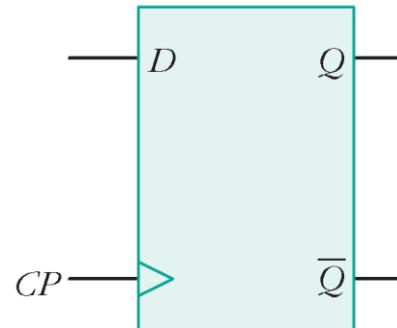
04 순서 논리 회로

□ D 플립플롭

- SR 플립플롭에서 원하지 않는 상태($S=R=1$)를 제거하는 한 가지 방법
- SR 플립플롭을 변형한 것
- 입력신호 D 가 CP 에 동기되어 그대로 출력에 전달되는 특성을 가지고 있음
- D 플립플롭이라는 이름은 데이터(Data)를 전달하는 것과 지연(Delay)하는 역할에서 유래



(a) 논리 회로



(b) 논리 기호

| D | $Q(t+1)$ |
|---|----------|
| 0 | 0 |
| 1 | 1 |

(c) 진리표

그림 3-69 D 플립플롭의 구조

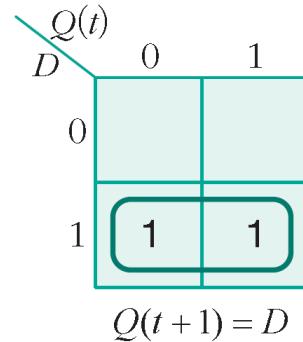
| | |
|-------------|--|
| $CP=1, D=1$ | G_3 의 출력은 0, G_4 의 출력은 1이 된다. 따라서 SR 래치의 입력은 $S=0, R=1$ 이 되므로 결과적으로 $Q=1$ 을 얻는다. |
| $CP=1, D=0$ | G_3 의 출력은 1, G_4 의 출력은 0이 된다. 따라서 SR 래치의 입력은 $S=1, R=0$ 이 되므로 결과적으로 $Q=0$ 을 얻는다. |

04 순서 논리 회로

❖ D 플립플롭의 특성표 및 특성 방정식

| D | $Q(t)$ | $Q(t+1)$ |
|-----|--------|----------|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

(a) 특성표



(b) 특성 방정식

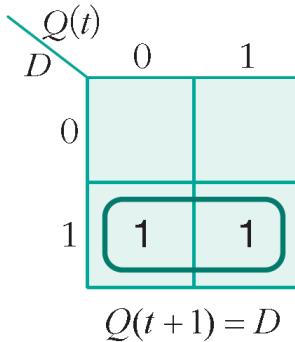


그림 3-70 D 플립플롭의 특성표와 특성 방정식

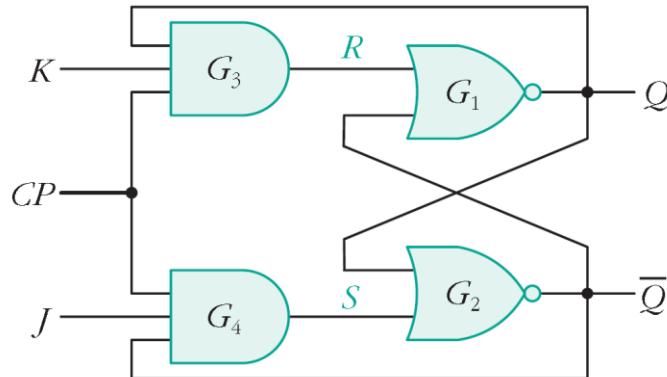
04 순서 논리 회로

□ JK 플립플롭

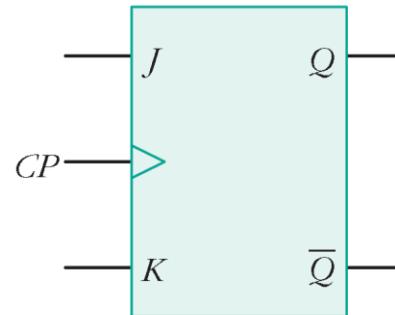
- JK 플립플롭은 SR 플립플롭에서 $S=1, R=1$ 인 경우 출력이 불안정한 상태가 되는 문제점을 개선하여 $S=1, R=1$ 에서도 동작하도록 개선한 회로
- JK 플립플롭의 J 는 S (set)에, K 는 R (reset)에 대응하는 입력
- $J=1, K=1$ 인 경우 JK 플립플롭의 출력은 이전 출력의 보수 상태로 변화
- JK 플립플롭은 플립플롭 중에서 가장 많이 사용되는 플립플롭이다.

04 순서 논리 회로

JK 플립플롭



(a) 논리 회로



(b) 논리 기호

| J | K | $Q(t+1)$ |
|-----|-----|-------------------|
| 0 | 0 | $Q(t)$ (불변) |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | $\bar{Q}(t)$ (토글) |

(c) 진리표

그림 3-71 JK 플립플롭의 구조

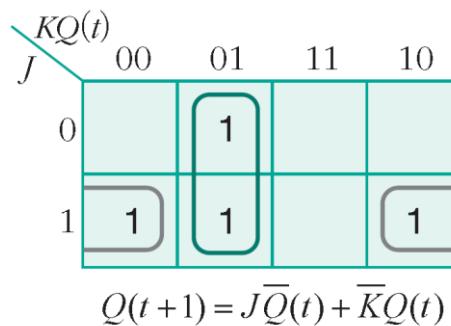
| | |
|------------|--|
| $J=0, K=0$ | G_3 과 G_4 의 출력이 모두 0이므로 G_1 과 G_2 로 구성된 SR 래치는 출력이 변하지 않는다. |
| $J=0, K=1$ | G_4 의 출력은 0이 되고 G_3 의 출력은 $Q(t) \cdot K \cdot CP$ 인데 $K=1, CP=1$ 이므로 $Q(t)$ 가 된다. |
| $J=1, K=0$ | G_3 의 출력은 0이 되고 G_4 의 출력은 $\bar{Q}(t) \cdot J \cdot CP$ 인데 $J=1, CP=1$ 이므로 $\bar{Q}(t)$ 가 된다. |
| $J=1, K=1$ | G_3 의 출력은 $Q(t) \cdot K \cdot CP$ 인데 $K=1, CP=1$ 이므로 $Q(t)$ 가 된다. 또한 G_4 의 출력은 $\bar{Q}(t) \cdot J \cdot CP$ 인데 $J=1, CP=1$ 이므로 $\bar{Q}(t)$ 가 된다. $Q(t)=0$ 인 경우 SR 래치의 $S=1, R=0$ 인 경우와 같으므로 출력은 $Q(t+1)=1$ 이 된다. 마찬가지로 $Q(t)=1$ 인 경우 SR 래치의 $S=0, R=1$ 인 경우와 같으므로 출력은 $Q(t+1)=0$ 이 된다. 따라서 출력은 보수가 된다. |

04 순서 논리 회로

❖ JK플립플롭의 특성표 및 특성 방정식

| J | K | $Q(t)$ | $Q(t+1)$ |
|-----|-----|--------|----------|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

(a) 특성표



(b) 특성 방정식

$$Q(t+1) = J\bar{Q}(t) + \bar{K}Q(t)$$

그림 3-72 JK 플립플롭의 특성표와 특성 방정식

04 순서 논리 회로

□ T플립플롭

- JK 플립플롭의 J 와 K 입력을 묶어서 하나의 입력신호 T 로 동작시키는 플립플롭
- T 플립플롭의 입력 $T=0$ 이면, T 플립플롭은 $J=0, K=0$ 인 JK 플립플롭과 같이 동작하므로 출력은 변하지 않는다. $T=1$ 이면, $J=1, K=1$ 인 JK 플립플롭과 같이 동작하므로 출력은 보수가 된다.

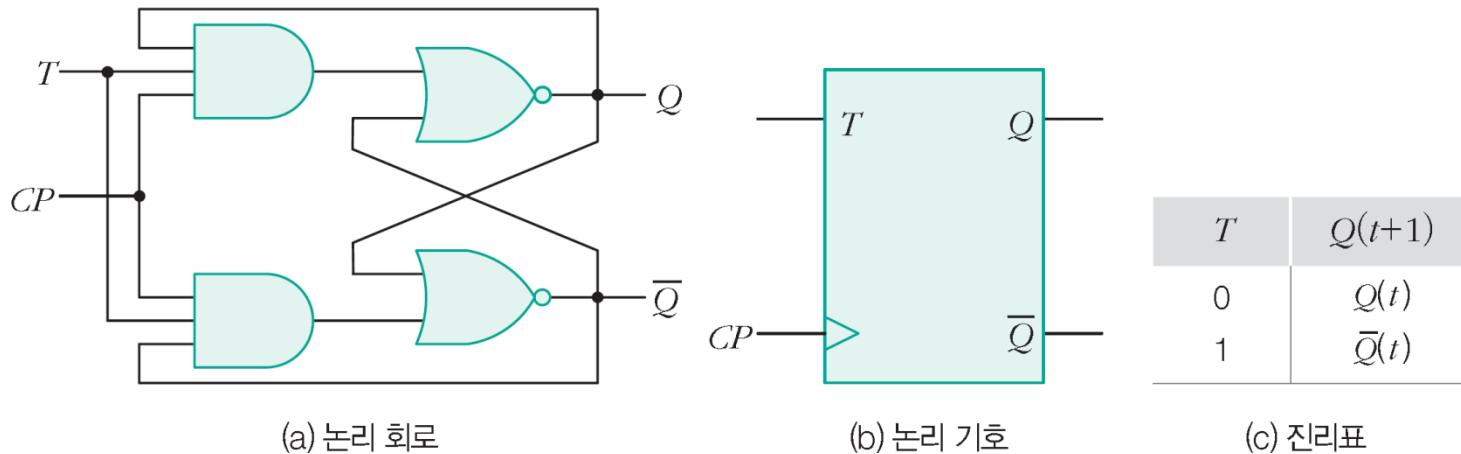
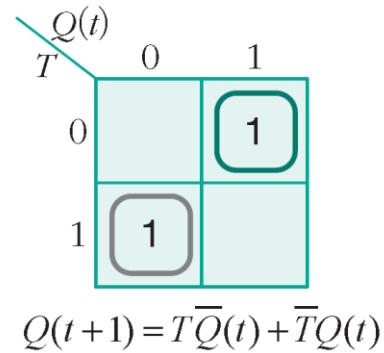


그림 3-73 T 플립플롭

04 순서 논리 회로

❖ T플립플롭의 특성표 및 특성 방정식

| T | $Q(t)$ | $Q(t+1)$ |
|-----|--------|----------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



(a) 특성표

(b) 특성 방정식

그림 3-74 T 플립플롭의 특성표와 특성 방정식

04 순서 논리 회로

□ 주종형 JK 플립플롭

- 레이스 현상 문제(racing problem) : JK 플립플롭은 $J = K = 1$ 인 경우 클록 펄스가 길어지면 출력이 계속 반전되는 현상
- 이를 해결하는 방법은 에지 트리거를 이용하거나 주종형 JK 플립플롭(master-slave JK flip-flop)을 사용하는 것이다.

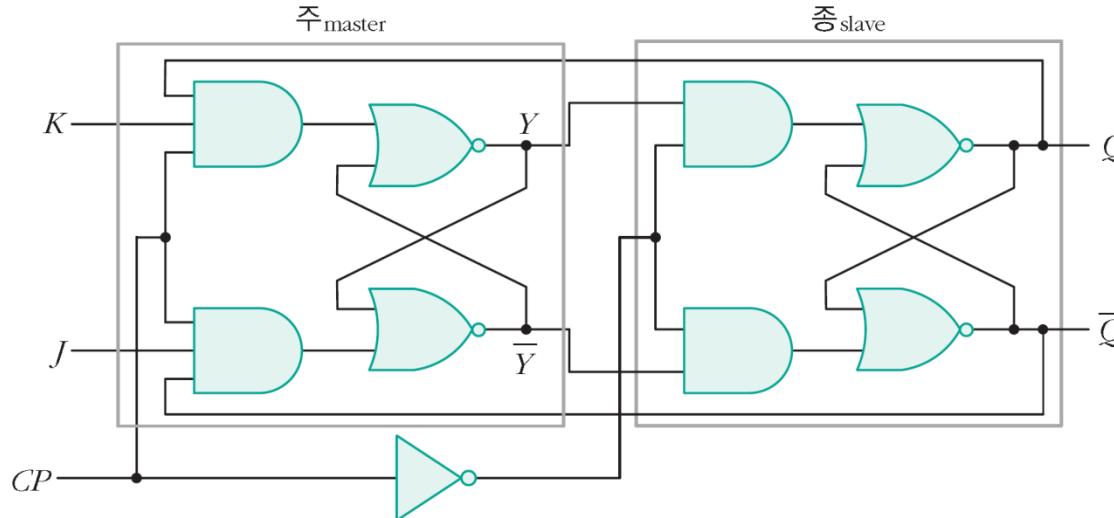


그림 3-75 주종형 JK 플립플롭

| | |
|--------|--|
| $CP=1$ | 외부의 J와 K의 입력이 Master 플립플롭에 전달 Slave 플립플롭은 $CP=0$ 이므로 동작하지 않음 |
| $CP=0$ | Slave 플립플롭이 동작하여 $Q = Y, \bar{Q} = \bar{Y}$ Master 플립플롭은 $CP=0$ 이므로 동작하지 않음 |

04 순서 논리 회로

□ 비동기 입력

- 대부분의 플립플롭은 클록펄스에 의해서 플립플롭의 상태를 변화시킬 수 있는 동기입력이 있고, 클록펄스와 관계없이 비동기적으로 변화시킬 수 있는 비동기 입력인 preset(\overline{PR}) 입력과 clear(\overline{CLR}) 입력이 있다.
- 비동기 입력들은 플립플롭의 초기조건을 결정하는 등 다방면으로 유용하게 사용

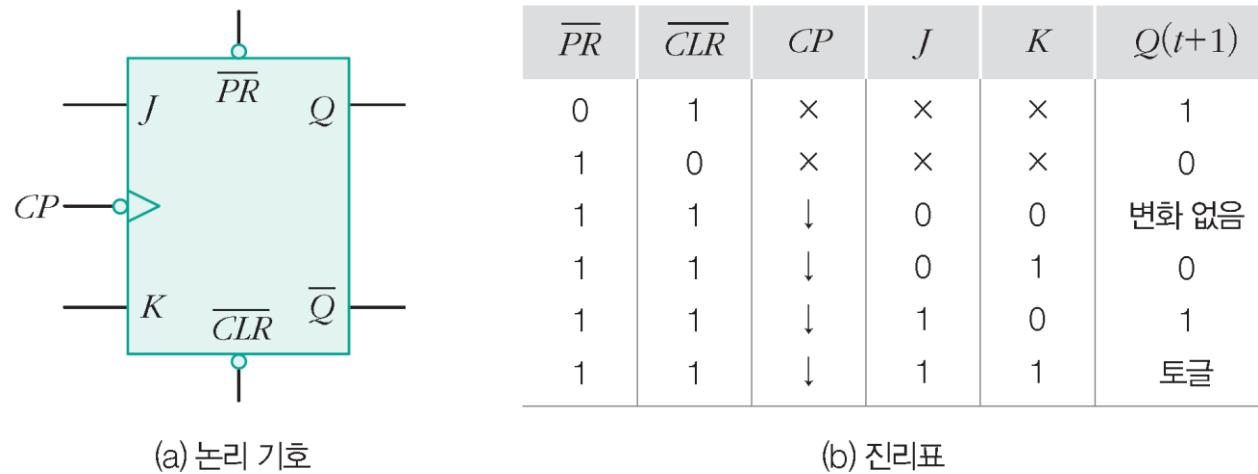


그림 3-76 비동기 입력을 가진 JK 플립플롭의 논리 기호와 진리표

3 순서 논리 회로의 설계

□ 여기표

- 플립플롭의 특성표 : 현재 상태와 입력값이 주어졌을 때, 다음 상태가 어떻게 변하는가를 나타내는 표
- 플립플롭의 여기표(excitation table) : 현재 상태에서 다음 상태로 변했을 때 플립플롭의 입력조건이 어떤 상태인가를 나타내는 표
- 플립플롭의 여기표는 순서논리회로를 설계할 때 자주 사용

04 순서 논리 회로

❖ SR 플립플롭의 예기표

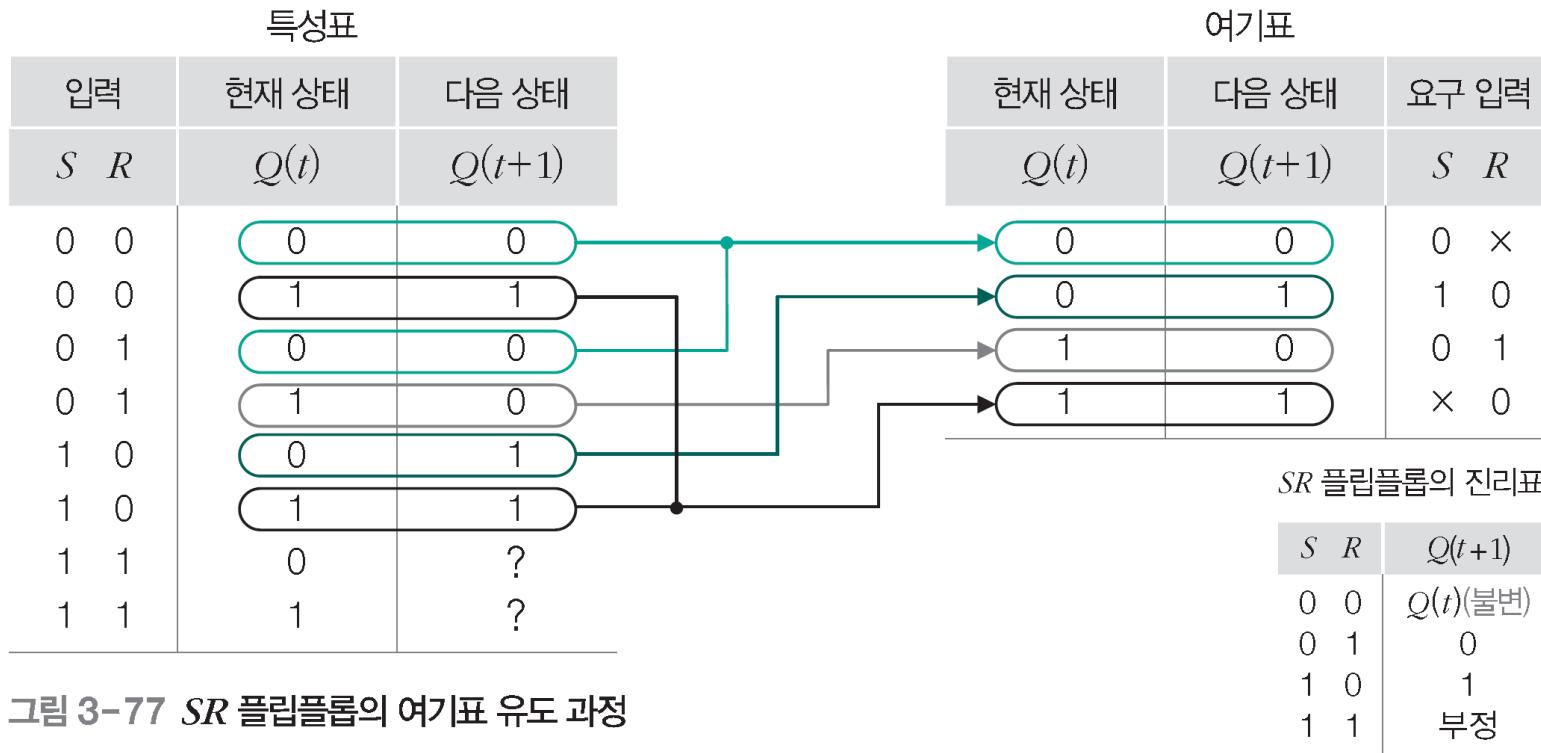


그림 3-77 SR 플립플롭의 예기표 유도 과정

04 순서 논리 회로

❖ JK플립플롭의 예기표

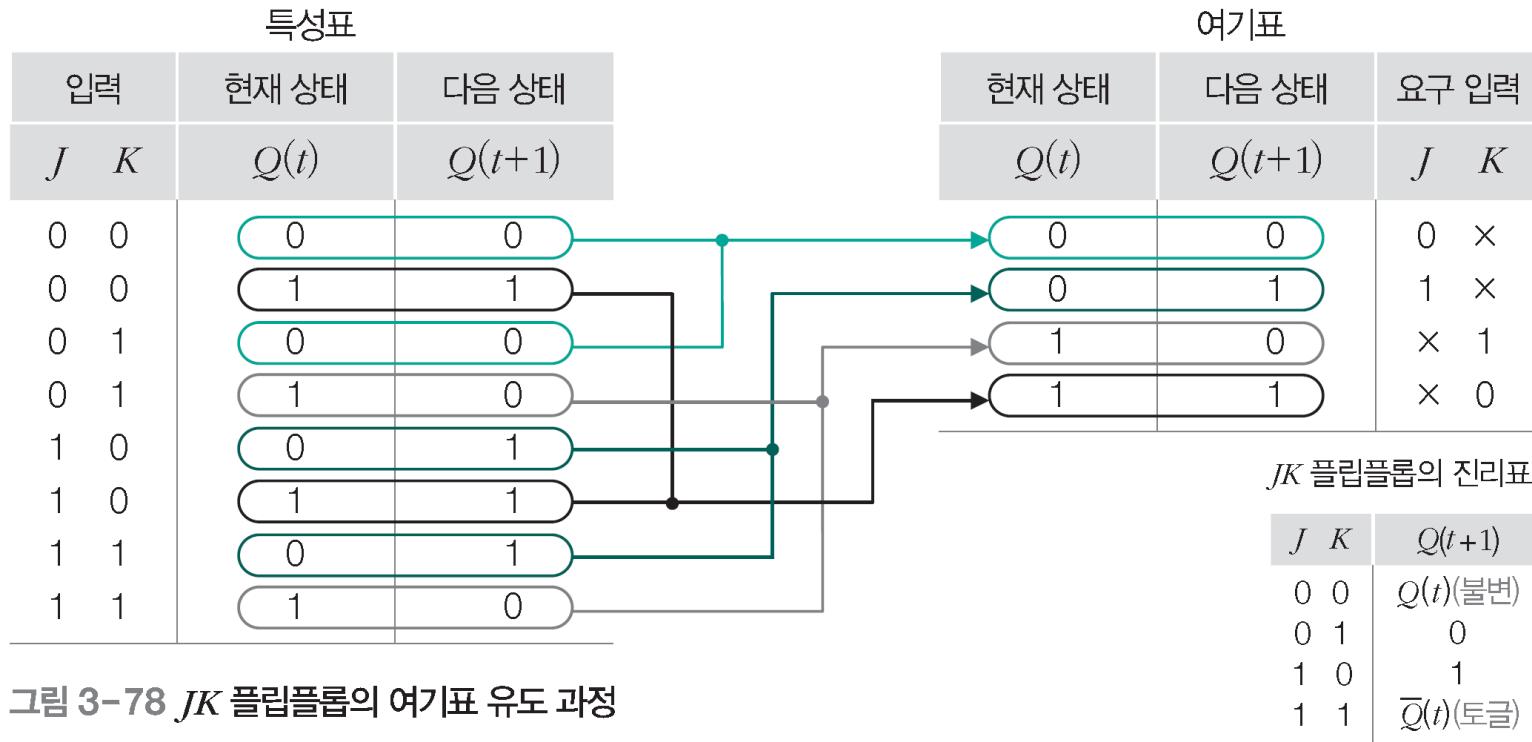


그림 3-78 JK 플립플롭의 예기표 유도 과정

04 순서 논리 회로

❖ D 플립플롭의 여기표

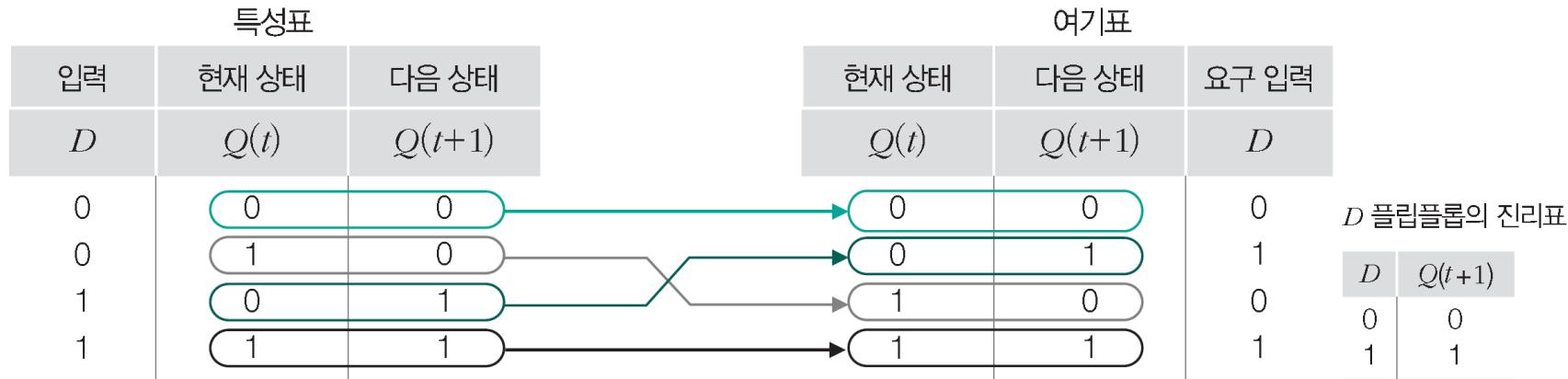


그림 3-79 D 플립플롭의 여기표 유도 과정

❖ T 플립플롭의 여기표

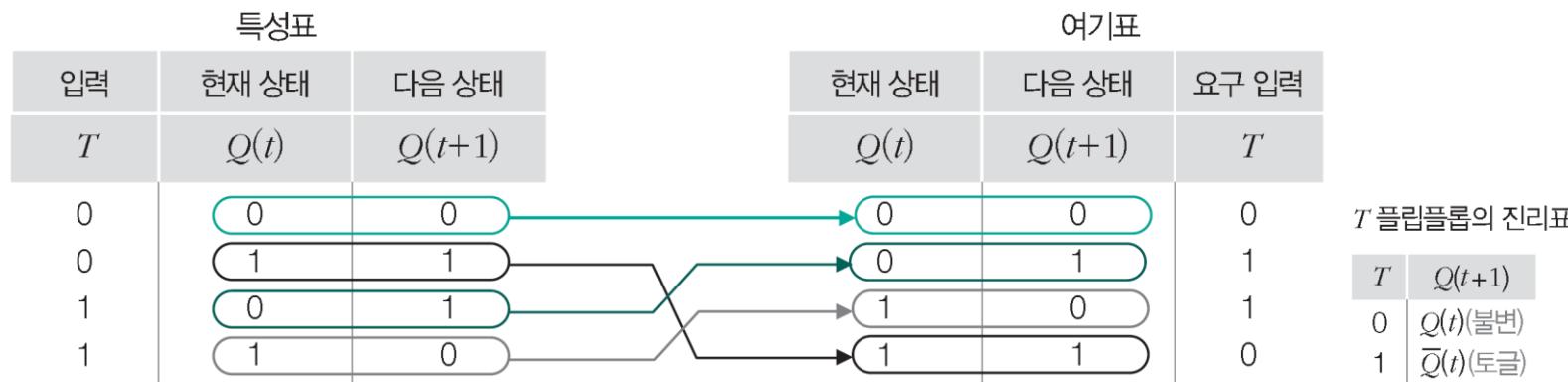


그림 3-80 T 플립플롭의 여기표 유도 과정

04 순서 논리 회로

□ 순서 논리 회로의 설계 과정

- ① 설계 사양으로부터 상태도와 상태표 작성
- ② 플립플롭의 수와 종류 결정
- ③ 플립플롭의 입력, 출력 및 각 상태에 문자 기호 부여
- ④ 상태표를 이용해 회로의 상태 여기표 작성
- ⑤ 간소화 방법을 이용해 출력 함수와 플립플롭의 입력 함수 유도
- ⑥ 순서 논리 회로도 작성

04 순서 논리 회로

① 설계 사양으로부터 상태도와 상태표 작성

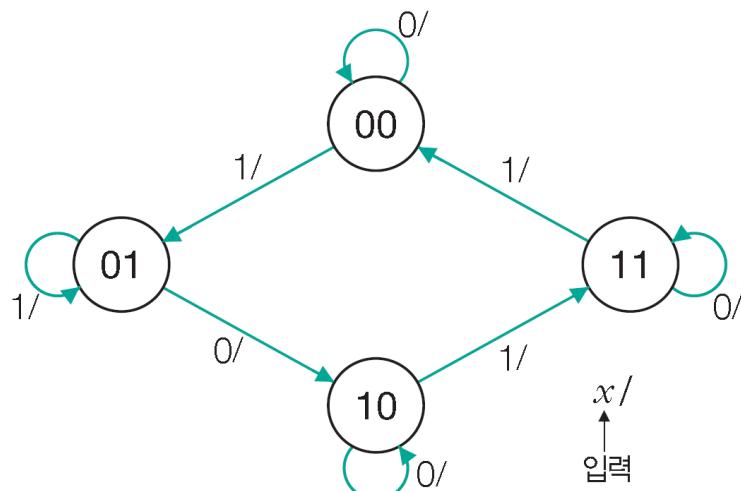


그림 3-81 순서 논리 회로에 대한 상태도

표 3-8 그림 3-81의 상태표

| 현재 상태 | | 입력 | 다음 상태 | |
|-------|---|----|-------|---|
| A | B | x | A | B |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 |

② 플립플롭의 수와 종류 결정하고 ③ 각 상태에 문자 기호 부여

- 네 가지 상태가 있으므로 플립플롭이 2개 필요하며, 각 플립플롭에 문자 A 와 B 를 할당한다.
- JK 플립플롭을 이용한다.

n 개의 서로 다른 상태를 나타내려면 플립플롭이 $\lceil \log_2 n \rceil$ 개 필요하다. 예를 들어 $n=10$ 이면 $\log_2 10 \approx 3.219$ 이므로 플립플롭이 $\lceil \log_2 10 \rceil = 4$ 개 필요하다.

04 순서 논리 회로

④ 상태표를 이용해 회로의 상태 예기표 작성

표 3-9 상태 예기표

| 현재 상태 | | 입력 x | 다음 상태 | | 플립플롭 입력 | | | |
|-------|-----|-----------|-------|-----|---------|-------|-------|-------|
| A | B | | A | B | J_A | K_A | J_B | K_B |
| 0 | 0 | 0 | 0 | 0 | 0 | × | 0 | × |
| 0 | 0 | 1 | 0 | 1 | 0 | × | 1 | × |
| 0 | 1 | 0 | 1 | 0 | 1 | × | × | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | × | × | 0 |
| 1 | 0 | 0 | 1 | 0 | × | 0 | 0 | × |
| 1 | 0 | 1 | 1 | 1 | × | 0 | 1 | × |
| 1 | 1 | 0 | 1 | 1 | × | 0 | × | 0 |
| 1 | 1 | 1 | 0 | 0 | × | 1 | × | 1 |

JK 플립플롭의 예기표

| $Q(t)$ | $Q(t+1)$ | J | K |
|--------|----------|-----|-----|
| 0 | 0 | 0 | × |
| 0 | 1 | 1 | × |
| 1 | 0 | × | 1 |
| 1 | 1 | 1 | 0 |

04 순서 논리 회로

⑤ 간소화 방법을 이용해 출력 함수와 플립플롭의 입력 함수 유도

| | | Bx | 00 | 01 | 11 | 10 | |
|--|--|------|----|----|----|----|---|
| | | A | 0 | | | | 1 |
| | | | 1 | X | X | X | X |
| | | | | | | | |
| | | | | | | | |

$$J_A = B\bar{x}$$

| | | Bx | 00 | 01 | 11 | 10 | |
|--|--|------|----|----|----|----|--|
| | | A | 0 | X | X | X | |
| | | | 1 | | | 1 | |
| | | | | | | | |
| | | | | | | | |

$$K_A = Bx$$

| | | Bx | 00 | 01 | 11 | 10 | |
|--|--|------|----|----|----|----|--|
| | | A | 0 | 1 | X | 1 | |
| | | | 1 | X | X | X | |
| | | | | | | | |
| | | | | | | | |

$$J_B = x$$

| | | Bx | 00 | 01 | 11 | 10 | |
|--|--|------|----|----|----|----|---|
| | | A | 0 | X | X | | 1 |
| | | | 1 | X | X | 1 | |
| | | | | | | | |
| | | | | | | | |

$$K_B = Ax + \bar{A}x = A \odot x$$

그림 3-82 카르노 맵을 이용한 간소화 과정

04 순서 논리 회로

⑥ 순서 논리 회로도 작성

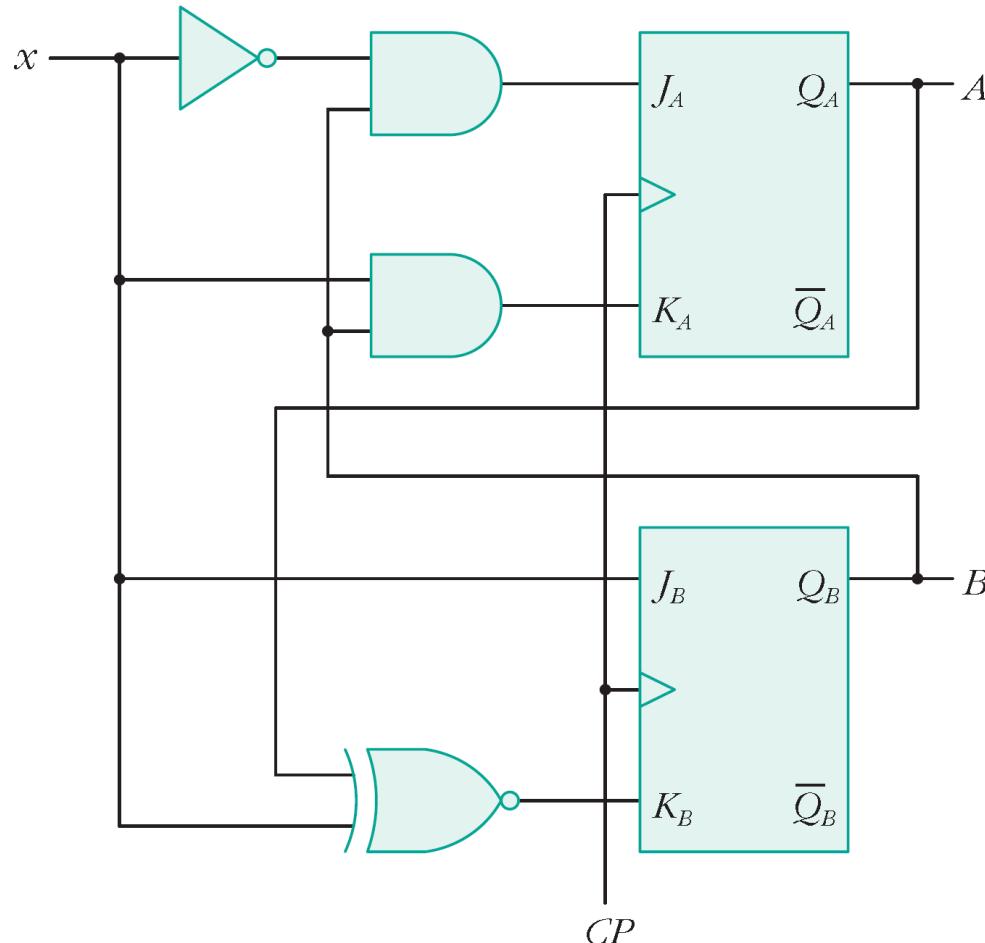


그림 3-83 순서 논리 회로의 구현

04 순서 논리 회로

4 카운터의 설계

- JK 플립플롭을 사용해 3비트 동기식 카운터를 순서 논리 회로 방식으로 설계해 보자.

상태도와 상태표 작성

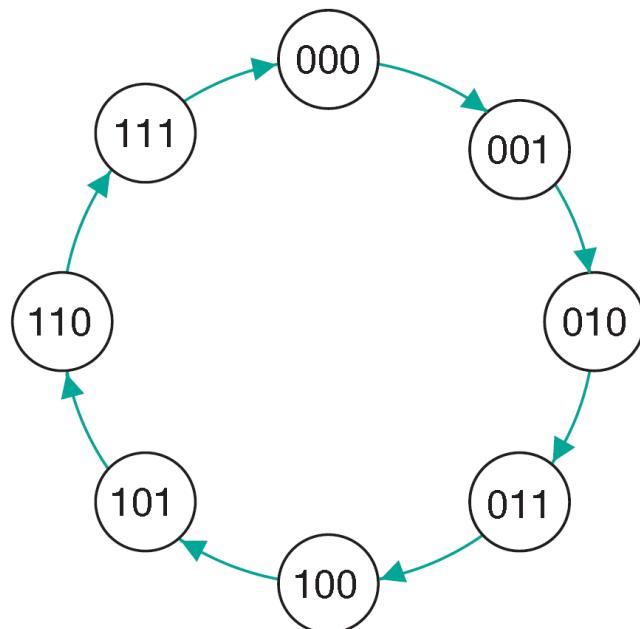


그림 3-84 3비트 동기식 2진 카운터의 상태도

상태표를 이용해 회로의 상태 여기표 작성

표 3-10 3비트 동기식 2진 카운터의 상태 여기표

| 현재 상태 | | | 다음 상태 | | | 플립플롭 입력 | | | | | |
|-------|-------|-------|-------|-------|-------|----------|----------|----------|----------|----------|----------|
| Q_C | Q_B | Q_A | Q_C | Q_B | Q_A | J_C | K_C | J_B | K_B | J_A | K_A |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | \times | 0 | \times | 1 | \times |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | \times | 1 | \times | \times | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | \times | \times | 0 | 1 | \times |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | \times | \times | 1 | \times | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | \times | 0 | 0 | \times | 1 | \times |
| 1 | 0 | 1 | 1 | 1 | 0 | \times | 0 | 1 | \times | \times | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | \times | 0 | \times | 0 | 1 | \times |
| 1 | 1 | 1 | 0 | 0 | 0 | \times | 1 | \times | 1 | \times | 1 |

04 순서 논리 회로

간소화 방법을 이용해 플립플롭의 입력 함수 유도

| | | $Q_B Q_A$ | 00 | 01 | 11 | 10 |
|--|--|-----------|----|----|----|----|
| | | Q_C | 0 | | | |
| | | 0 | | | 1 | |
| | | 1 | X | X | X | X |
| | | | | | | |

$J_C = Q_B Q_A$

| | | $Q_B Q_A$ | 00 | 01 | 11 | 10 |
|--|--|-----------|----|----|----|----|
| | | Q_C | 0 | | | |
| | | 0 | X | X | X | X |
| | | 1 | | | 1 | |
| | | | | | | |

$K_C = Q_B Q_A$

| | | $Q_B Q_A$ | 00 | 01 | 11 | 10 |
|--|--|-----------|----|----|----|----|
| | | Q_C | 0 | | | |
| | | 0 | | | 1 | X |
| | | 1 | 1 | X | X | X |
| | | | | | | |

$J_B = Q_A$

| | | $Q_B Q_A$ | 00 | 01 | 11 | 10 |
|--|--|-----------|----|----|----|----|
| | | Q_C | 0 | | | |
| | | 0 | X | X | 1 | |
| | | 1 | X | X | 1 | |
| | | | | | | |

$K_B = Q_A$

| | | $Q_B Q_A$ | 00 | 01 | 11 | 10 |
|--|--|-----------|----|----|----|----|
| | | Q_C | 0 | | | |
| | | 0 | 1 | X | X | 1 |
| | | 1 | 1 | X | X | 1 |
| | | | | | | |

$J_A = 1$

| | | $Q_B Q_A$ | 00 | 01 | 11 | 10 |
|--|--|-----------|----|----|----|----|
| | | Q_C | 0 | | | |
| | | 0 | 1 | 1 | 1 | X |
| | | 1 | X | 1 | 1 | X |
| | | | | | | |

$K_A = 1$

그림 3-85 3비트 동기식 2진 카운터의 카르노 맵을 이용한 간소화 과정

04 순서 논리 회로

순서 논리 회로도 작성

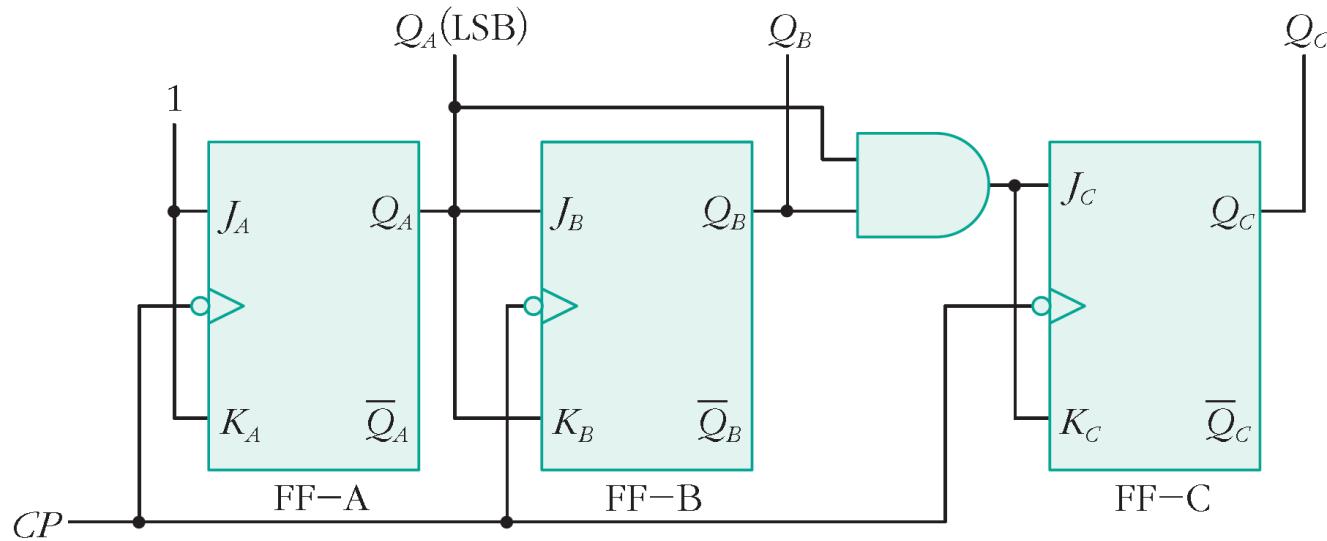


그림 3-86 3비트 동기식 2진 카운터 회로도

04 순서 논리 회로

5 레지스터

- **레지스터**(register)는 기본적으로 데이터 비트를 저장하는 소자
- 대부분의 레지스터에서는 D 플립플롭이 사용되며 각 D 플립플롭에 한 비트씩 저장
- 따라서 n 비트 레지스터는 플립플롭 n 개로 구성되며, n 비트의 2진 정보를 저장할 수 있다.

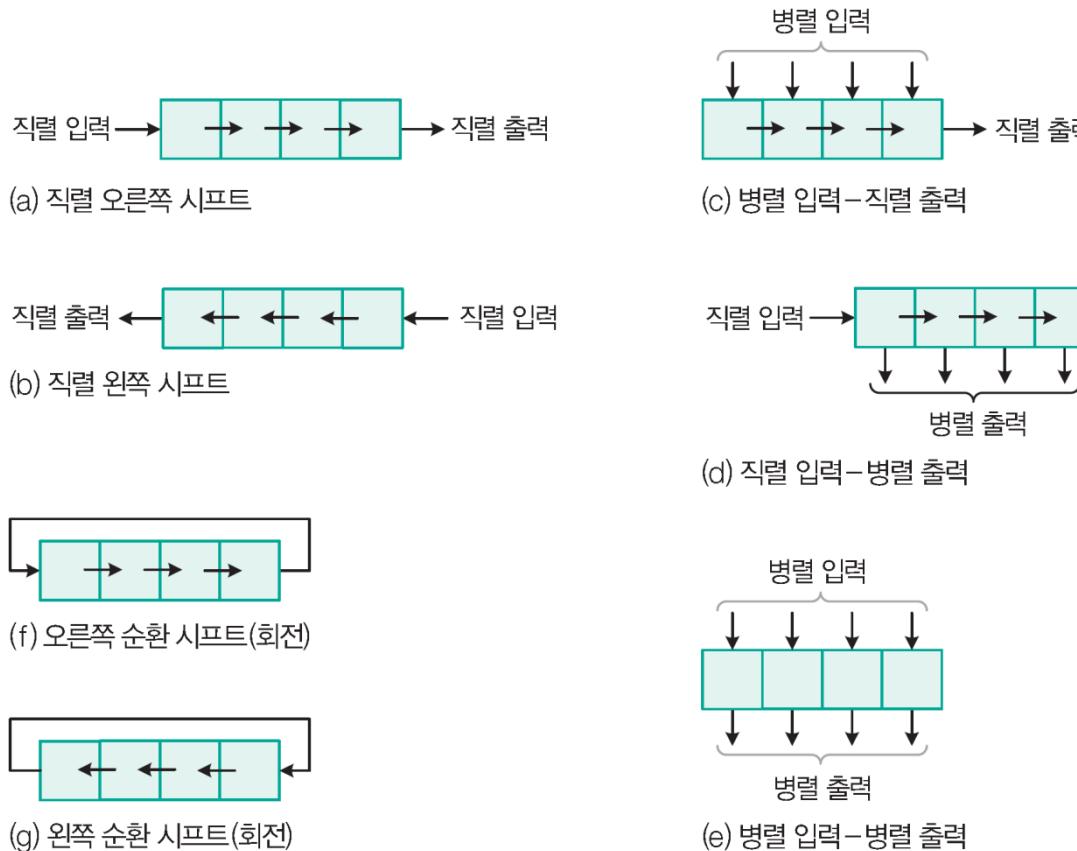


그림 3-87 레지스터 내에서 데이터 이동(4비트인 경우)

04 순서 논리 회로

□ 4비트 레지스터

- 공통 클록 신호의 상승 에지에서 입력 데이터(I_A, I_B, I_C, I_D)가 D 플립플롭 4개에 동시에 저장되며, 출력(O_A, O_B, O_C, O_D)에서는 언제든지 저장된 데이터를 출력할 수 있다.
- $\overline{CLR} = 0$ 이면 클록에 관계없이 언제든지 모든 플립플롭의 출력을 0으로 만들 수 있다.

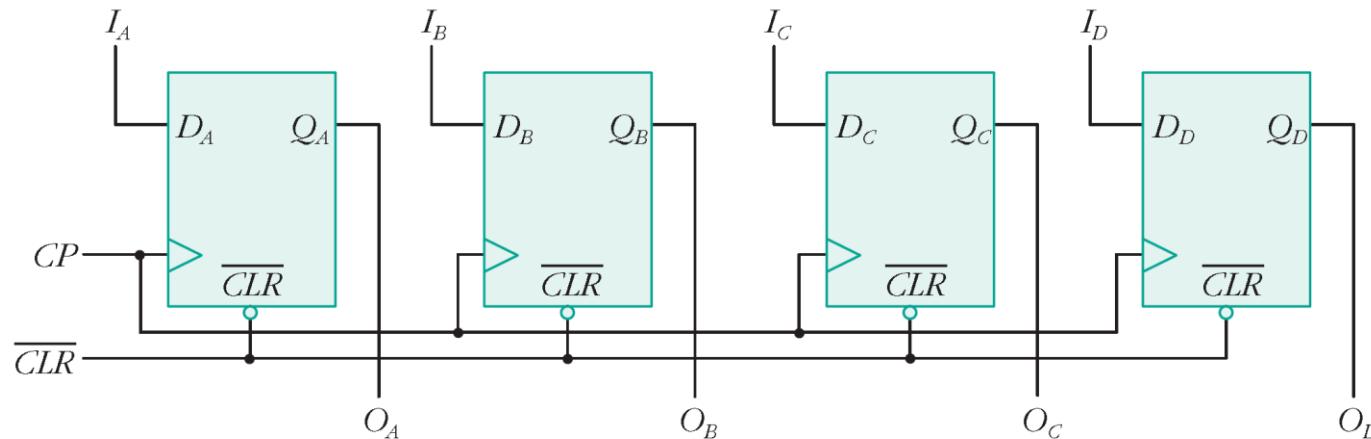


그림 3-88 4비트 레지스터

□ 시프트 레지스터

- 클록 펄스가 입력될 때마다 클록 펄스의 상승 에지에서 입력 데이터가 한 비트씩 오른쪽으로 시프트하면서 저장($I \rightarrow Q_A, Q_A \rightarrow Q_B, Q_B \rightarrow Q_C, Q_C \rightarrow Q_D$)
- 이 과정은 새로운 클록 펄스의 상승 에지마다 반복되므로 네 번째 클록 펄스의 상승 에지에서 처음에 입력된 데이터 비트가 Q_D 에 나타난다.

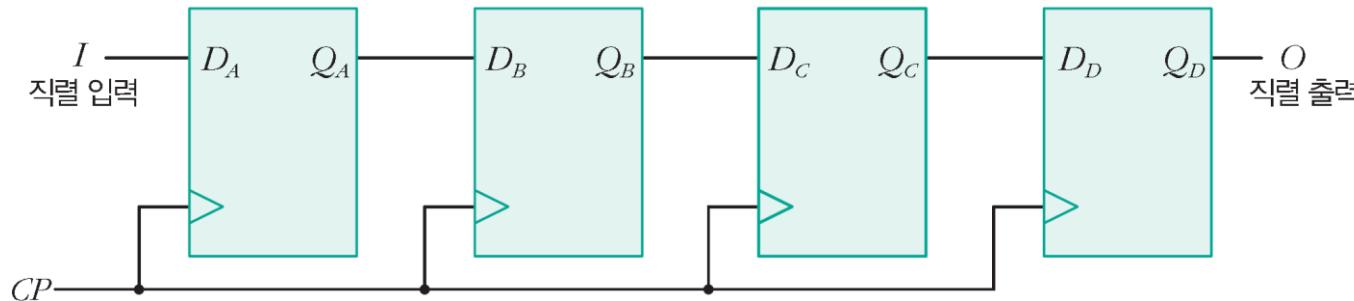


그림 3-89 4비트 시프트 레지스터(직렬 입력 – 직렬 출력)

Summary

- 여러 가지 조합 논리 회로의 동작 원리 이해하고 응용 회로 설계
- 여러 가지 순서 논리 회로의 동작 원리 이해하고 응용 회로 설계

Microprocessor (W4)

- Central Processing Unit (CPU) -

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

Contents

01 프로세서 구성과 동작

02 산술 논리 연산 장치

01 프로세서 구성과 동작

1 컴퓨터 기본 구조와 프로세서

- 컴퓨터의 3가지 핵심 장치 : 프로세서(Processor, CPU), 메모리, 입출력장치
- 버스(Bus) : 장치간에 주소, 데이터, 제어 신호를 전송하기 위한 연결 통로(연결선)

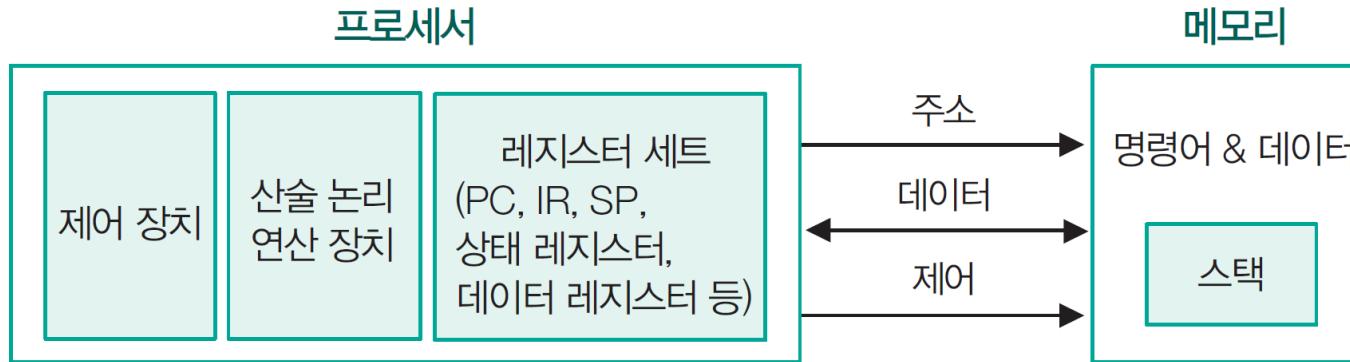


그림 4-1 폰 노이만 컴퓨터의 기본 구조

01 프로세서 구성과 동작

- **버스(Bus)** : 장치간에 주소, 데이터, 제어 신호를 전송하기 위한 연결 통로(연결선)
 - **내부버스(internal bus)** : 프로세서 내부의 장치 연결
 - **시스템 버스(system bus)** : 핵심 장치 및 주변장치 연결

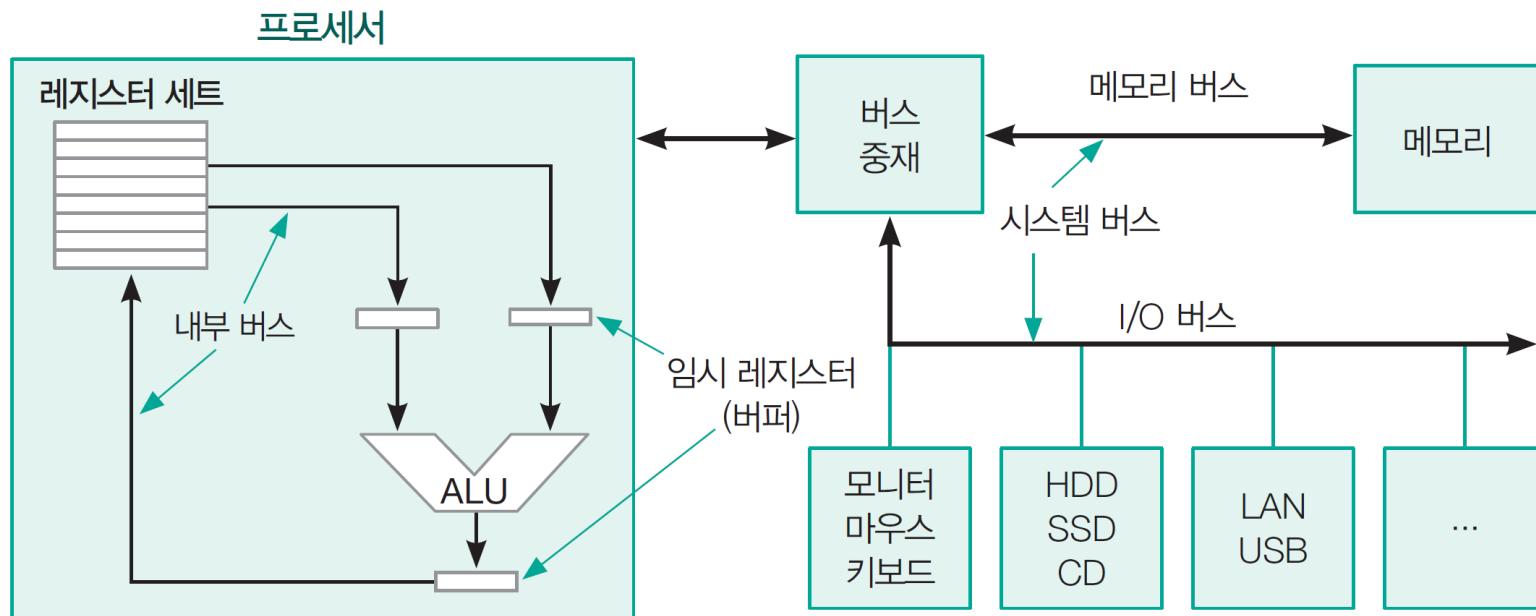


그림 4-2 버스 기반 컴퓨터 구조

2 프로세서 구성 요소

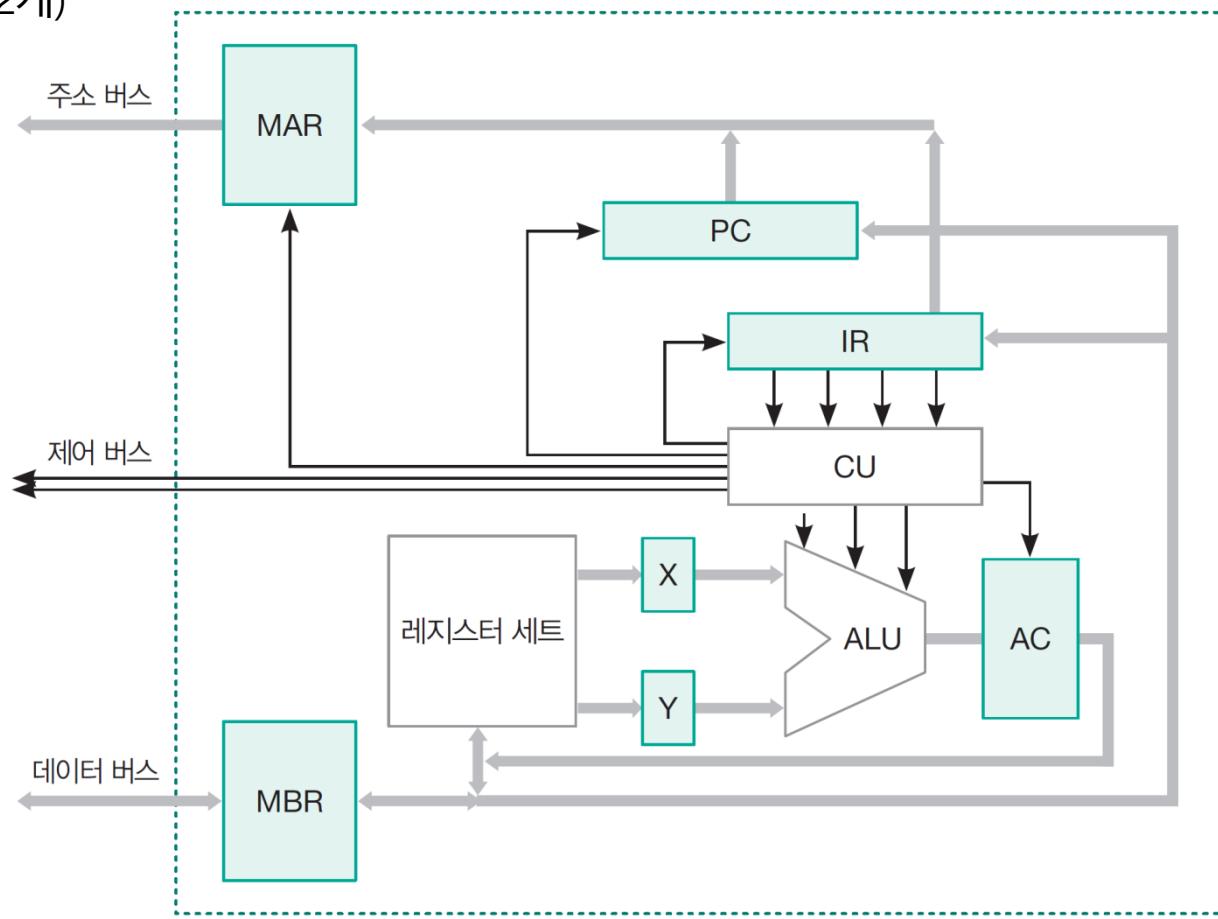
❖ 프로세서 3가지 구성 필수 구성요소

- 산술 논리 연산 장치(Arithmetic Logic Unit, ALU) : 산술 및 논리 연산 등 기본 연산을 수행
- 제어 장치 (Control Unit, CU) : 메모리에서 명령어를 가져와 해독하고 실행에 필요한 장치들을 제어하는 신호를 발생
- 레지스터 세트(register set) : 프로세서 내에 존재하는 용량은 작지만 매우 빠른 메모리, ALU의 연산과 관련된 데이터를 일시 저장하거나 특정 제어 정보 저장
 - 목적에 따라 특수 레지스터와 범용 레지스터로 분류
- 현재는 온칩 캐시(on-chip cache), 비디오 컨트롤러(video controller), 실수보조연산 프로세서(FPU) 등 다양한 장치 포함

01 프로세서 구성과 동작

3 프로세서 기본 구조

- 레지스터 세트(일반적으로 1~32개)
- ALU
- CU
- 이들 장치를 연결하는 버스로 구성



MAR Memory Address Register: 메모리 주소 레지스터

AC ACcumulator: 누산기

IR Instruction Register: 명령 레지스터

PC Program Counter: 프로그램 카운터

MBR(MDR) Memory Buffer(Data) Register: 메모리 버퍼(데이터) 레지스터

ALU Arithmetic Logic Unit: 산술 논리 연산 장치

그림 4-3 프로세서 기본 구조

01 프로세서 구성과 동작

❖ ALU

- 덧셈, 뺄셈 등 연산을 수행하고, 그 결과를 **누산기**(Accumulator, AC)에 저장

❖ 프로세서 명령 분류

| | |
|--------------|--|
| 레지스터-메모리 명령 | <ul style="list-style-type: none">메모리 워드를 레지스터로 가져올(LOAD) 때레지스터의 데이터를 메모리에 다시 저장(STORE)할 때 |
| 레지스터-레지스터 명령 | <ul style="list-style-type: none">레지스터에서 오퍼랜드 2개를 ALU의 입력 레지스터로 가져와 덧셈 또는 논리 AND 같은 몇 가지 연산을 수행하고그 결과를 레지스터 중 하나에 다시 저장 |

4 프로세서 명령 실행

- 프로세서는 각 명령을 더 작은 마이크로 명령(microinstruction)들로 나누어 실행

1단계 다음에 실행할 명령어를 메모리에서 읽어 명령 레지스터(IR)로 가져온다.

2단계 프로그램 카운터(PC)는 그 다음 명령어의 주소로 변경된다.

3단계 방금 가져온 명령어를 해독(decode)하고 유형을 결정한다.

4단계 명령어가 메모리에 있는 데이터를 사용하는 경우 그 위치를 결정한다.

5단계 필요한 경우 데이터를 레지스터로 가져온다.

6단계 명령어를 실행한다.

7단계 1단계로 이동하여 다음 명령어 실행을 시작한다.

- 이 단계를 요약하면 인출(fetch)-해독(decode)-실행(execute) 사이클로 구성 – 주 사이클(main cycle)

01 프로세서 구성과 동작

❖ 해독기(microprogrammed control) : 하드웨어를 소프트웨어로 대체

- 고가의 고성능 컴퓨터는 하드웨어 추가 비용이 크게 부담되지 않아 저가 컴퓨터보다 많은 명령어를 갖게 됨
- 고가인 고성능 컴퓨터의 복잡한 명령어를 저가 컴퓨터에서 실행할 수 있게 하기 위함
- 모리스 윌크스(Maurice Wilkes)가 제안(1951년)
 - 1957년 SDSAC 1.5에 적용
- 1970년대 설계된 거의 모든 컴퓨터가 해독기를 기반
 - Cray-1 같은 매우 고가의 고성능 모델을 제외하고는 1970년대 후반에 해독기를 운영하는 프로세서가 보편적으로 보급
 - 복잡한 명령어에 대한 비용 절감, 훨씬 더 복잡한 명령어 연구
- 제어 기억 장치(control memory)라는 빠른 읽기 전용 메모리

02 산술 논리 연산 장치

- ❖ **산술 논리 연산 장치(Arithmetic Logic Unit, ALU)** : 산술 연산과 논리 연산
 - 주로 정수 연산을 처리
 - 부동 소수(Floating-point Number) 연산 : FPU(Floating-Point Unit)
 - 최근에는 ALU가 부동 소수 연산까지 처리
- ❖ 산술 연산 : 덧셈, 뺄셈, 곱셈, 나눗셈, 증가, 감소, 보수
- ❖ 논리 연산 : AND, OR, NOT, XOR, 시프트(shift)

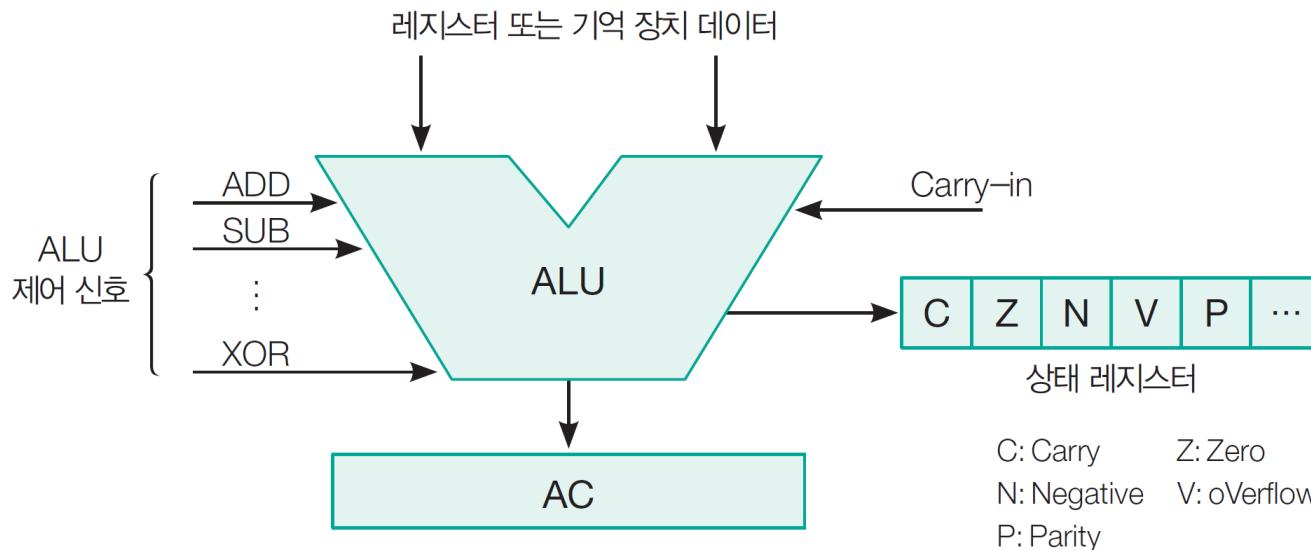


그림 4-4 ALU의 동작

02 산술 논리 연산 장치

1 산술 연산

표 4-1 산술 연산

| 연산 | 8비트 연산 | |
|-----|---------------------------------|---------------------------------|
| | 동작 | 설명 |
| ADD | $X \leftarrow A + B$ | A와 B를 더한다. |
| SUB | $X \leftarrow A + (\sim B + 1)$ | $A + (B\text{의 } 2\text{의 보수})$ |
| MUL | $X \leftarrow A * B$ | A와 B를 곱한다. |
| DIV | $X \leftarrow A / B$ | A와 B를 나눈다. |
| INC | $X \leftarrow A + 1$ | A를 1 증가시킨다. |
| DEC | $X \leftarrow A - 1(0xFF)$ | A를 1 감소시킨다. |
| NEG | $X \leftarrow \sim A + 1$ | A의 2의 보수다. |

02 산술 논리 연산 장치

❖ Booth Algorithm

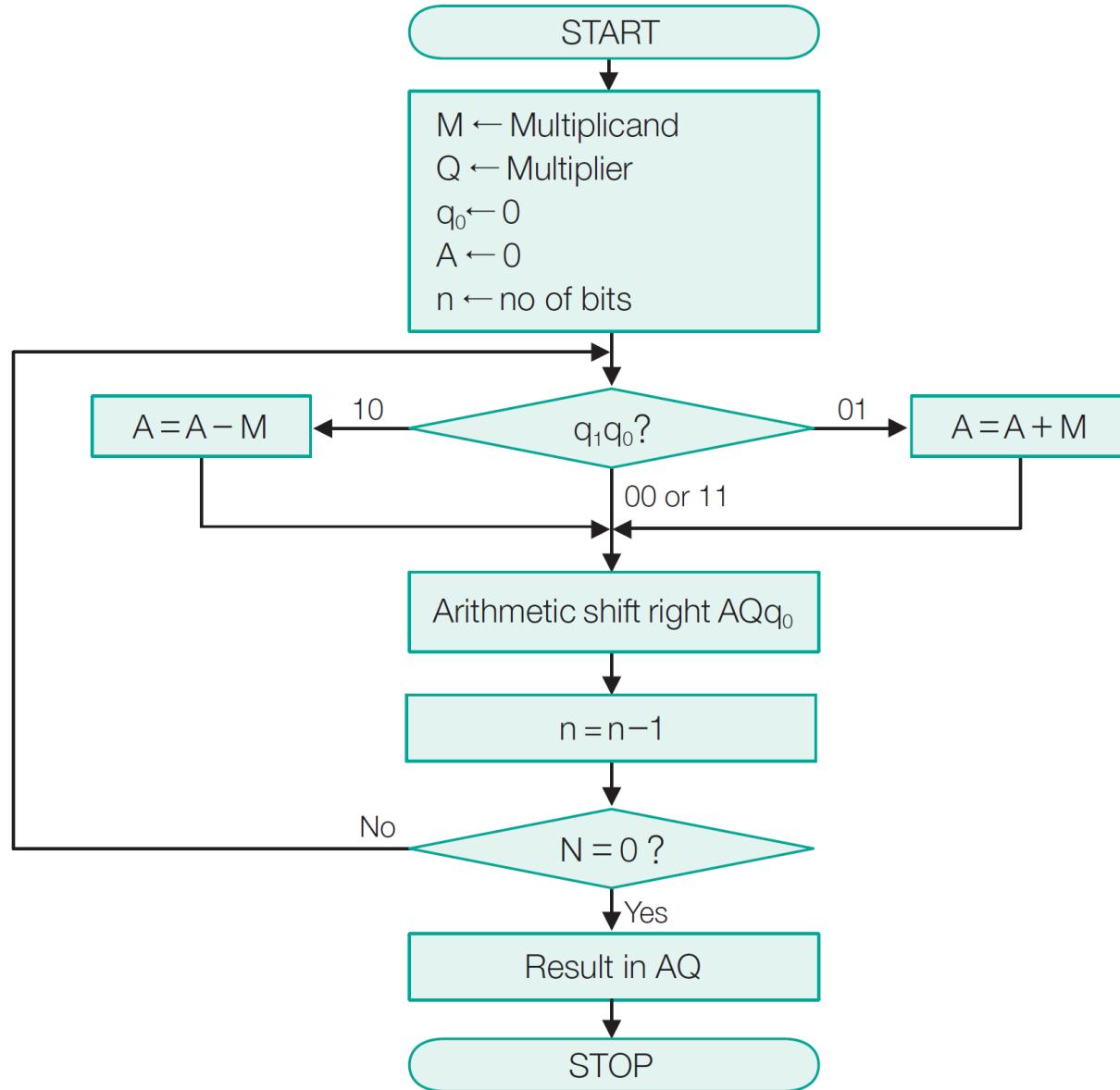


그림 4-5 부스 알고리즘 순서도

02 산술 논리 연산 장치

❖ Booth Algorithm 예 : (-7) * (+3)

| n | A | $Q(q_4q_3q_2q_1)$ | q_0 | 설명 |
|---|------|-------------------|-------|---|
| 4 | 0000 | 0011 | 0 | 초기 상태 |
| | 0111 | 0011 | 0 | q_1q_0 이 10이므로 $A = A - M = 0000 + 0111$ |
| 3 | 0011 | 1001 | 1 | <u>AQq_0을 오른쪽 산술 시프트</u> |
| 2 | 0001 | 1100 | 1 | q_1q_0 이 11이므로 연산 없이 AQq_0 을 오른쪽 산술 시프트 |
| | 1010 | 1100 | 1 | q_1q_0 이 01이므로 $A = A + M = 0001 + 1001$ |
| 1 | 1101 | 0110 | 0 | AQq_0 를 ASR |
| 0 | 1110 | 1011 | 0 | q_1q_0 가 00이므로 연산 없이 AQq_0 을 오른쪽 산술 시프트 |

02 산술 논리 연산 장치

❖ Booth Algorithm 예 : $5 * (-4)$

| n | A | $Q(q_4q_3q_2q_1)$ | q_0 | 설명 |
|---|------|-------------------|-------|--|
| 4 | 0000 | 1100 | 0 | 초기 상태 |
| 3 | 0000 | 0110 | 0 | q_1q_0 이 00이므로 연산 없이 AQq_0 을 오른쪽 산술 시프트 |
| 2 | 0000 | 0011 | 0 | q_1q_0 이 00이므로 연산 없이 AQq_0 을 A오른쪽 산술 시프트 |
| 1 | 1011 | 0011 | 0 | q_1q_0 이 10이므로 $A = A - M = 0000 + 1011$ |
| | 1101 | 1001 | 1 | AQq_0 을 오른쪽 산술 시프트 |
| 0 | 1110 | 1100 | 1 | q_1q_0 이 11이므로 연산 없이 AQq_0 을 오른쪽 산술 시프트 |

02 산술 논리 연산 장치

2 논리 연산과 산술 시프트 연산

표 4-2 논리 연산

| 연산 | 8비트 연산 | |
|------|--|---------------------------------|
| | 동작 | 설명 |
| AND | $X \leftarrow A \& B$ | A와 B를 비트 단위로 AND 연산한다. |
| OR | $X \leftarrow A B$ | A와 B를 비트 단위로 OR 연산한다. |
| NOT | $X \leftarrow \sim A$ | A의 1의 보수를 만든다. |
| XOR | $X \leftarrow A ^ B$ | A와 B를 비트 단위로 XOR 연산한다. |
| ASL | $X \leftarrow A \ll n$ | 왼쪽으로 n비트 시프트(LSL과 같다.) |
| ASR | $X \leftarrow A \gg n, A[7] \leftarrow A[7]$ | 오른쪽으로 n비트 시프트(부호 비트는 그대로 유지한다.) |
| LSL | $X \leftarrow A \ll n$ | 왼쪽으로 n비트 시프트 |
| LSR | $X \leftarrow A \gg n$ | 오른쪽으로 n비트 시프트 |
| ROL | $X \leftarrow A \ll 1, A[0] \leftarrow A[7]$ | 왼쪽으로 1비트 회전 시프트, MSB는 LSB로 시프트 |
| ROR | $X \leftarrow A \gg 1, A[7] \leftarrow A[0]$ | 오른쪽으로 1비트 회전 시프트, LSB는 MSB로 시프트 |
| ROLC | $X \leftarrow A \ll 1, C \leftarrow A[7], A[0] \leftarrow C$ | 캐리도 함께 왼쪽으로 1비트 회전 시프트 |
| RORC | $X \leftarrow A \gg 1, C \leftarrow A[0], A[7] \leftarrow C$ | 캐리도 함께 오른쪽으로 1비트 회전 시프트 |

02 산술 논리 연산 장치

❖ 논리 연산 예 1 : A=46=00101110₍₂₎, B=-75=10110101₍₂₎

| A AND B | | A OR B | | A XOR B | |
|------------|-----|----------|-----|------------|------|
| 00101110 | 46 | 00101110 | 46 | 00101110 | 46 |
| & 10110101 | -75 | 10110101 | -75 | ^ 11111111 | -128 |
| 00100100 | 36 | 10111111 | -65 | 11010001 | -47 |

02 산술 논리 연산 장치

❖ 논리 연산 예 2

| A AND B | A OR B |
|---------------------------------------|---|
| 00101110 & 00001111 상위 4비트 삭제 | 00001110 10110000 상위 4비트 값 설정 |
| 00001110 | 10111110 |

02 산술 논리 연산 장치

❖ 시프트 연산 예

| 연산 | 왼쪽 | | | | 오른쪽 | | | |
|---------------|-------------|-------------------|-------------------|---|-------------|-------------------|-------------------|---|
| 산술 시프트 | MSB ASL | 0 0 0 1 0 1 1 0 | 0 0 1 0 1 1 0 0 | 0 | MSB ASR | 1 0 0 1 0 1 1 0 | 1 1 0 0 1 0 1 1 | 0 |
| 논리 시프트 | MSB LSL | 0 0 0 1 0 1 1 0 | 0 0 1 0 1 1 0 0 | 0 | MSB LSR | 1 0 0 1 0 1 1 0 | 0 1 0 0 1 0 1 1 | 1 |
| 회전 시프트 | MSB ROL | 0 0 0 1 0 1 1 0 | 0 0 1 0 1 1 0 0 | | MSB ROR | 1 0 0 1 0 1 1 0 | 0 1 0 0 1 0 1 1 | |
| 캐리와 함께 회전 시프트 | MSB ROLC | 0 0 0 1 0 1 1 0 0 | 0 0 1 0 1 1 0 0 0 | C | MSB RORC | 0 0 0 1 0 1 1 0 1 | 1 0 0 0 1 0 1 1 1 | 1 |

Summary

- 컴퓨터 프로세서의 기본 구조와 명령 실행 과정 이해
- ALU 구조를 이해하고 프로세서에서의 산술 및 논리 연산을 학습

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

03 레지스터

1 레지스터 동작

- 레지스터는 CPU가 사용하는 데이터와 명령어를 신속하게 읽어 오고 저장하고 전송하는데 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

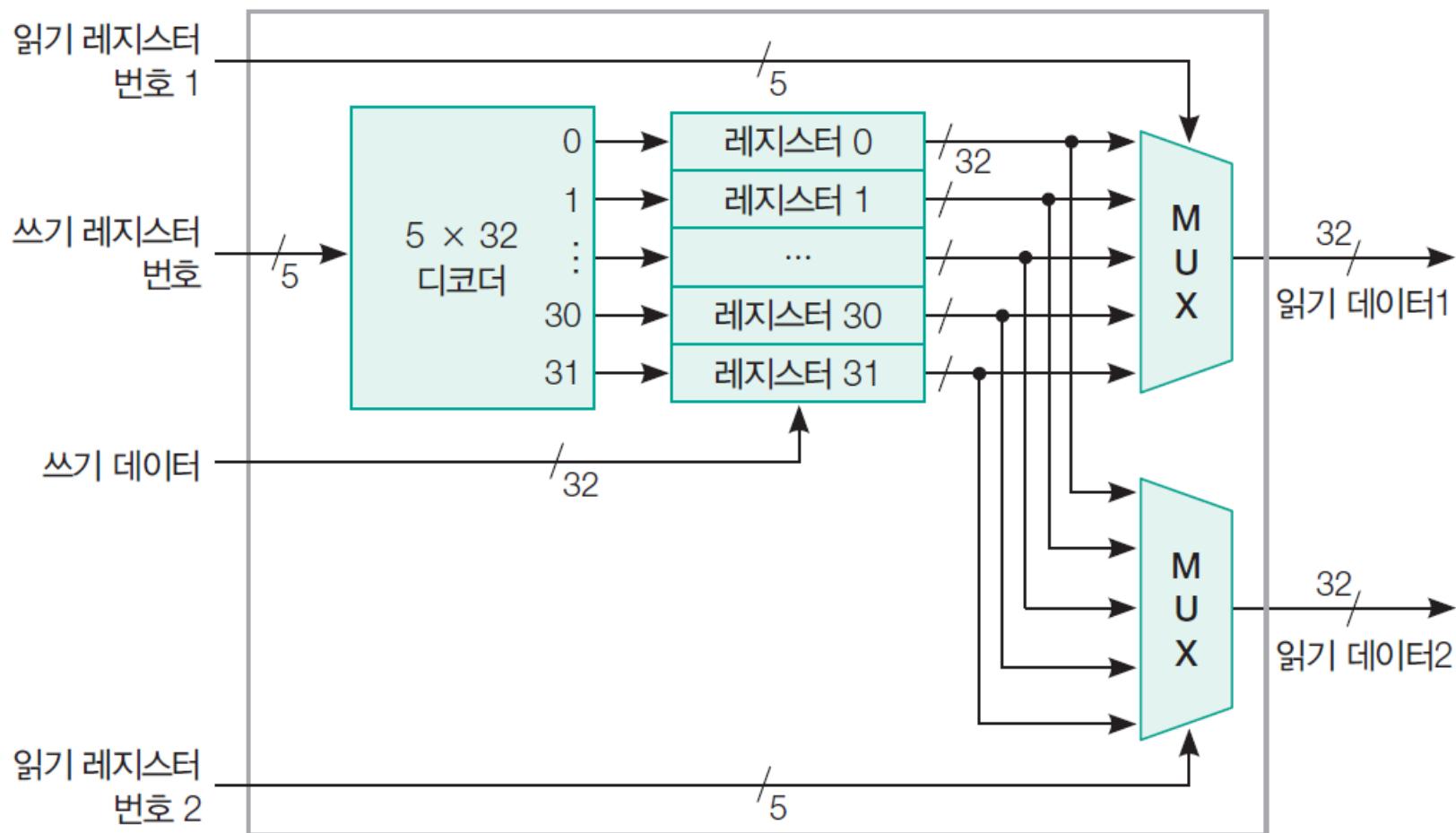


그림 4-6 레지스터 동작 개념

03 레지스터

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

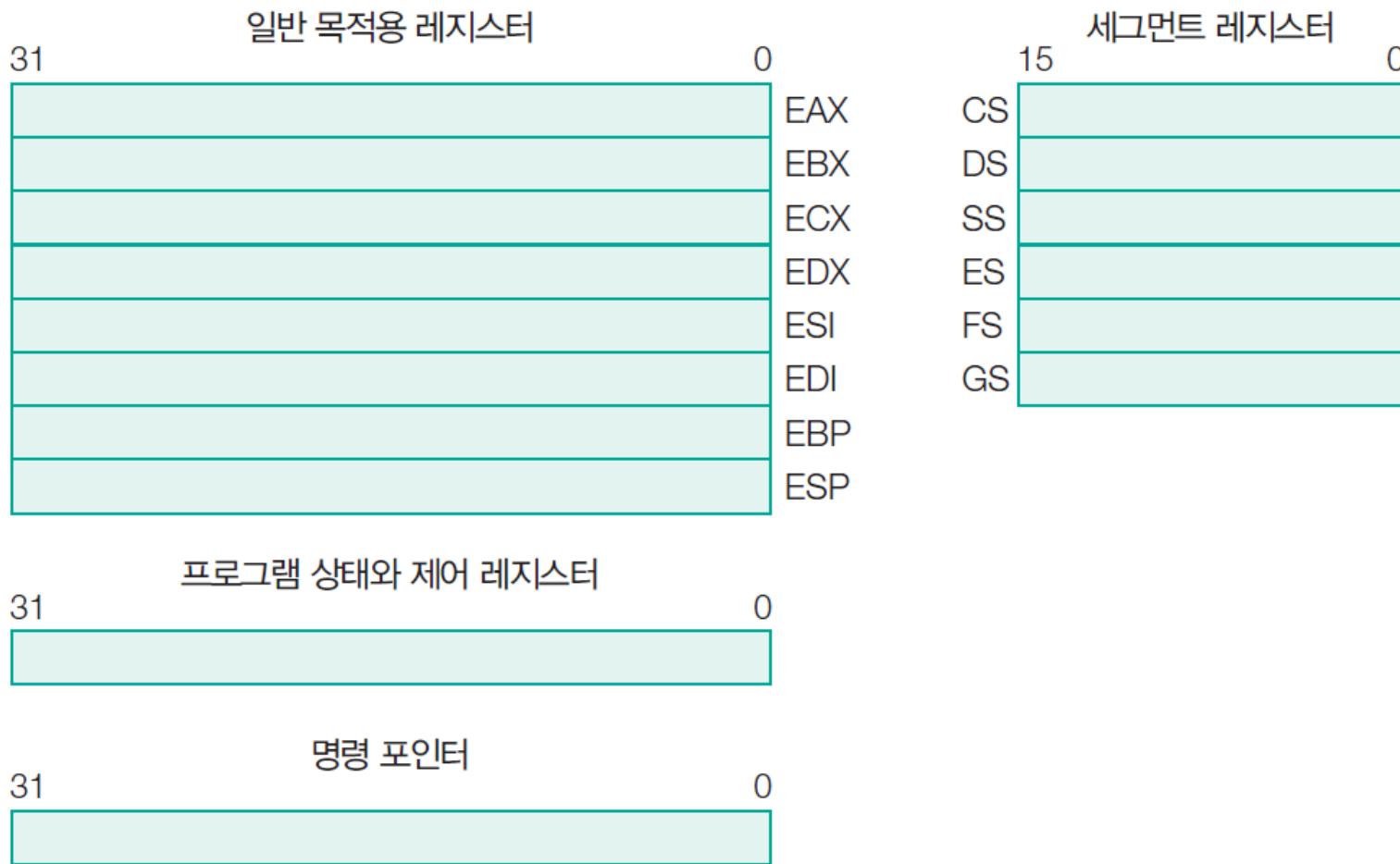


그림 4-7 인텔 x86 레지스터 종류

03 레지스터

3 레지스터 전송(LOAD, STORE, MOVE 명령 등)

- 3가지 레지스터 전송 명령 : LOAD, STORE, MOVE

| | |
|-------|--|
| LOAD | <ul style="list-style-type: none">주기억 장치에서 레지스터로 데이터를 읽음 |
| STORE | <ul style="list-style-type: none">레지스터에서 주기억 장치로 데이터를 저장 |
| MOVE | <ul style="list-style-type: none">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

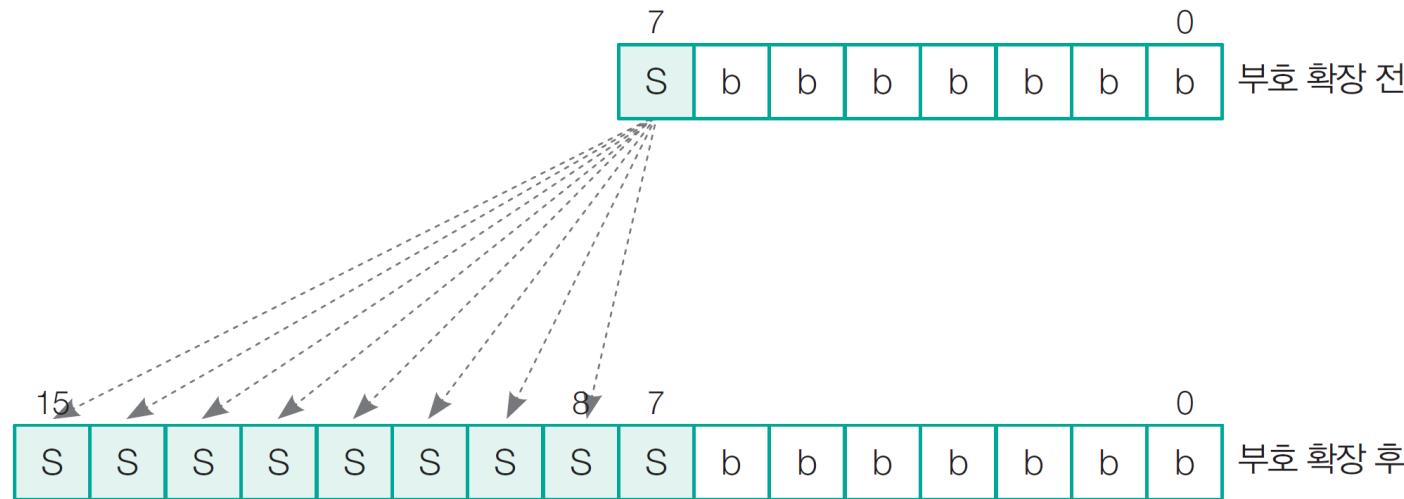


그림 4-8 부호 확장

04 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)

| |
|--------|
| opcode |
|--------|

(a) 0-주소 명령어

| |
|--------|
| opcode |
|--------|

| |
|------|
| addr |
|------|

(b) 1-주소 명령어

| |
|--------|
| opcode |
|--------|

| |
|-------|
| addr1 |
|-------|

| |
|-------|
| addr2 |
|-------|

(c) 2-주소 명령어

| |
|--------|
| opcode |
|--------|

| |
|-------|
| addr1 |
|-------|

| |
|-------|
| addr2 |
|-------|

| |
|-------|
| addr3 |
|-------|

(d) 3-주소 명령어

그림 4-9 명령어 형식

04 컴퓨터 명령어

❖ 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 ; $AC \leftarrow AC + M[X]$

- 오퍼랜드 필드의 모든 비트가 주소 지정에 사용: 보다 넓은 영역의 주소 지정
- 명령워드 : 16비트, Opcode: 5비트, 오퍼랜드(addr): 11비트 $\rightarrow 32 (=2^5)$ 가지의 연산 가능, 2048($=2^{11}$) 개 주소 지정 가능



(b) 1-주소 명령어

04 컴퓨터 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

04 컴퓨터 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 ; $R1 \leftarrow R2 + R3$

| | | | |
|--------|-------|-------|-------|
| opcode | addr1 | addr2 | addr3 |
|--------|-------|-------|-------|

(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 둠
- 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^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

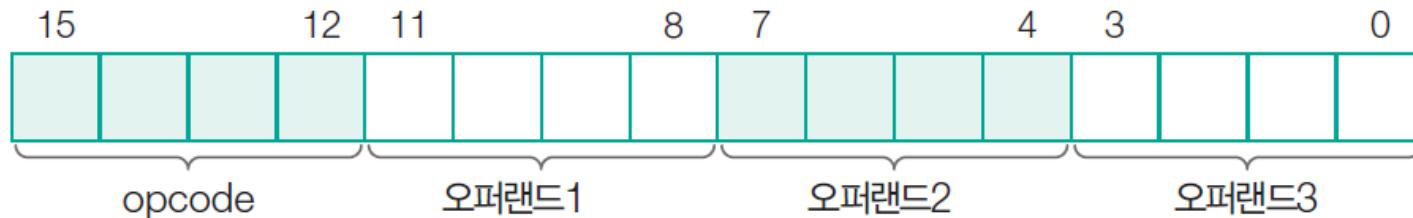


그림 4-10 3-주소 명령어 구조

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

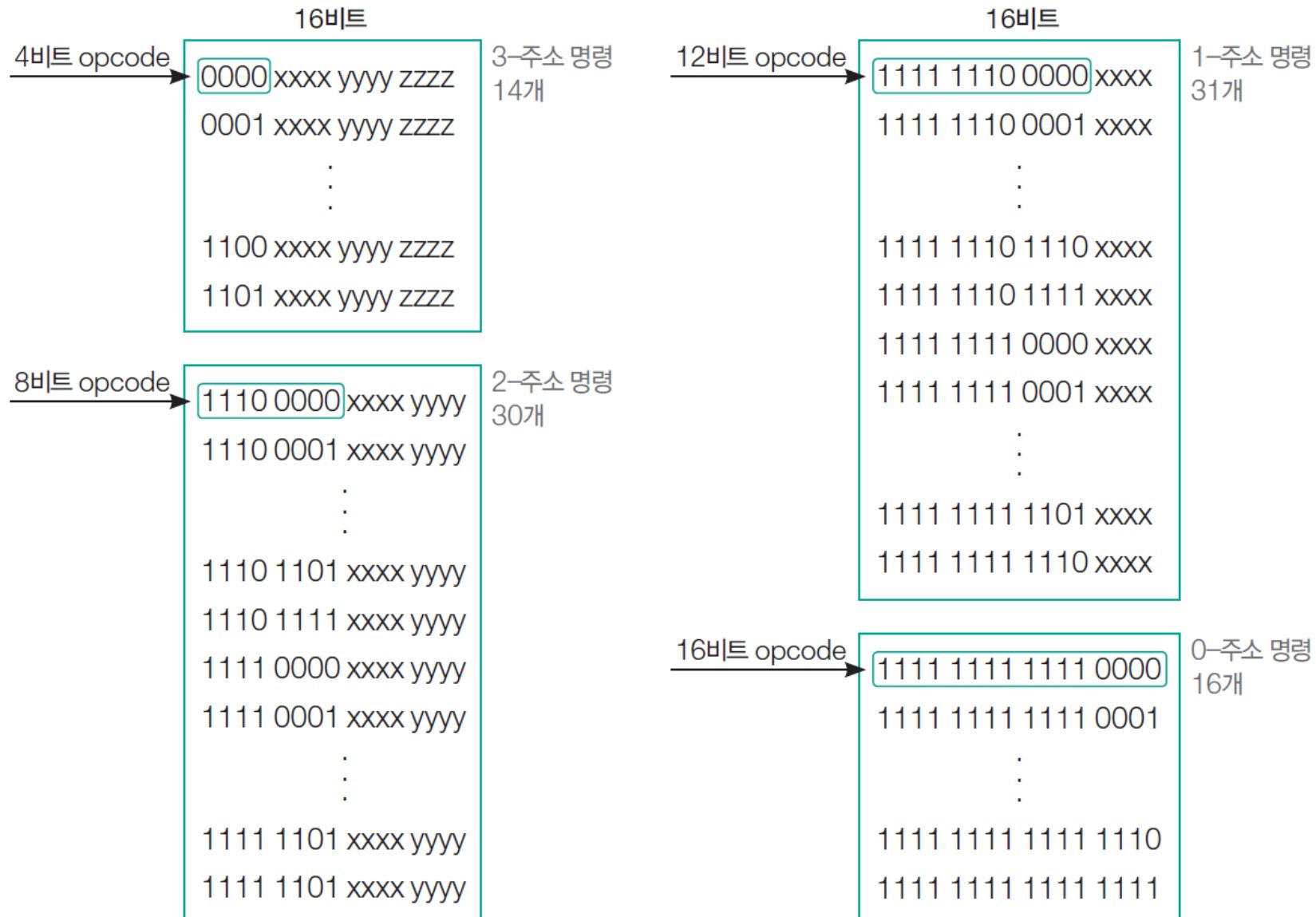


그림 4-11 명령어 설계의 예

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

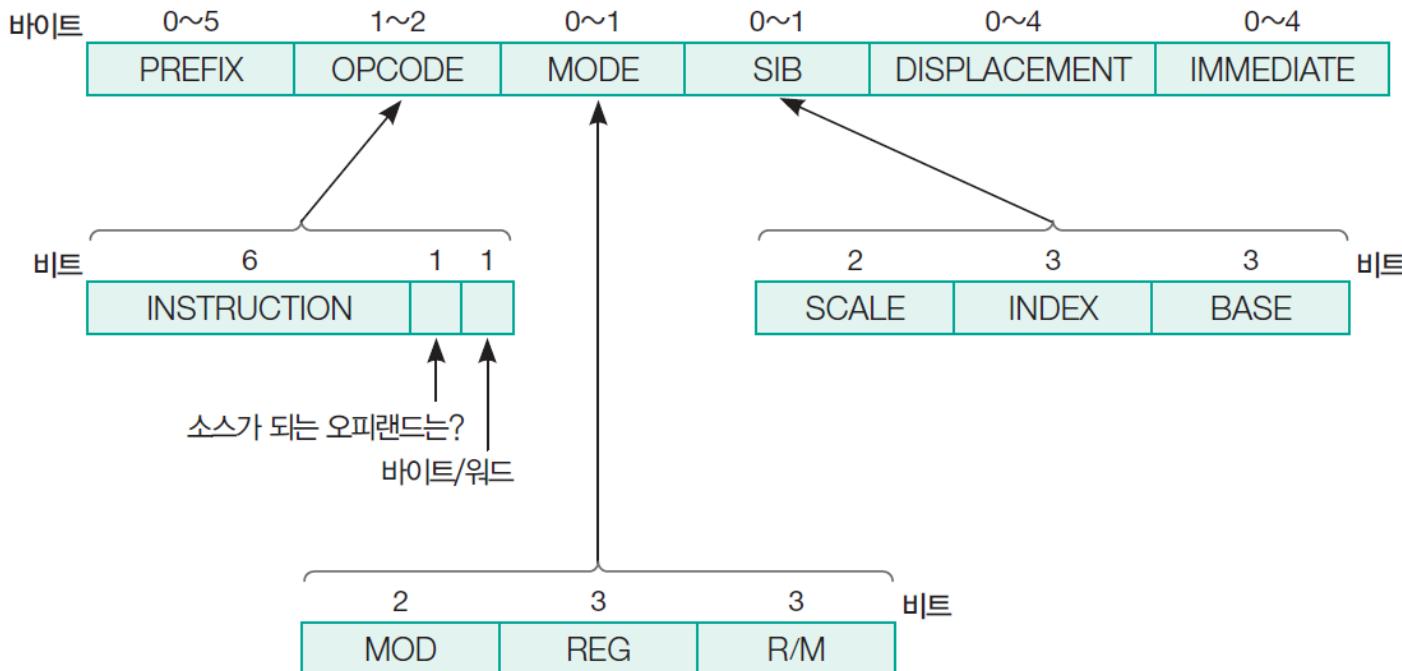
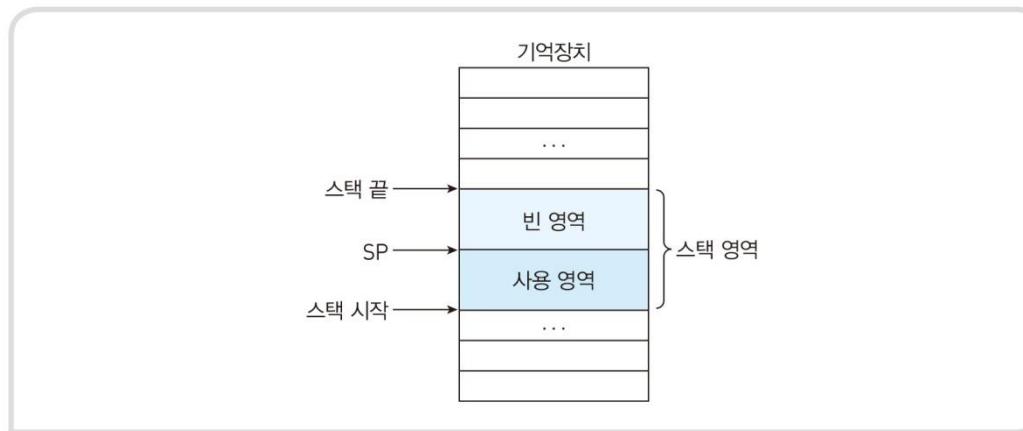


그림 4-12 코어 i7 명령어 형식

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

스택 포인터

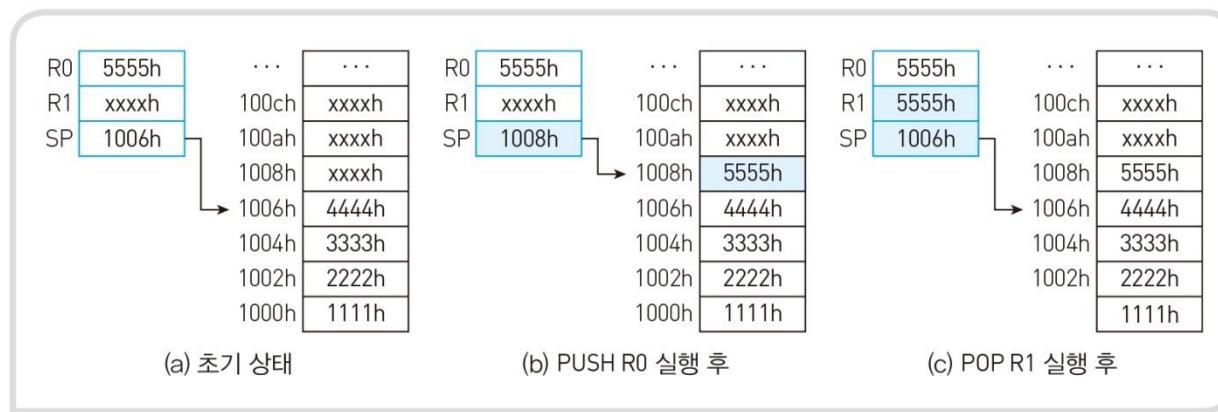
□ PUSH

- 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
 - $\text{MAR} \leftarrow SP, \text{MBR} \leftarrow \text{오퍼런드}$ // MAR, MBR 설정
 - $\text{Mem}[\text{MAR}] \leftarrow \text{MBR}$ // 기억장치에 저장

스택 포인터

□ POP

- 명령어: POP operand // 스택에서 제거한 데이터를 operand에 저장
 - Operand \leftarrow Mem[SP]
 - SP \leftarrow SP - [단어 크기]
- マイ크로오퍼레이션
 - MAR \leftarrow SP // MAR 설정
 - MBR \leftarrow Mem[MAR] // 기억장치 읽기
 - SP \leftarrow SP - [단어의 크기], 오퍼런드 \leftarrow 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

04 컴퓨터 명령어

□ 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

04 컴퓨터 명령어

□ 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

04 컴퓨터 명령어

❖ 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

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

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

- -1 은 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

04 컴퓨터 명령어

□ 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

04 컴퓨터 명령어

❖ 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 총족 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

04 컴퓨터 명령어

□ 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

04 컴퓨터 명령어

□ 입출력 명령

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

04 컴퓨터 명령어

❖ 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

04 컴퓨터 명령어

❖ 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 들
 - 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

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

[05 주소 지정 방식](#)

[06 CISC와 RISC](#)

05 주소 지정 방식

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

05 주소 지정 방식

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

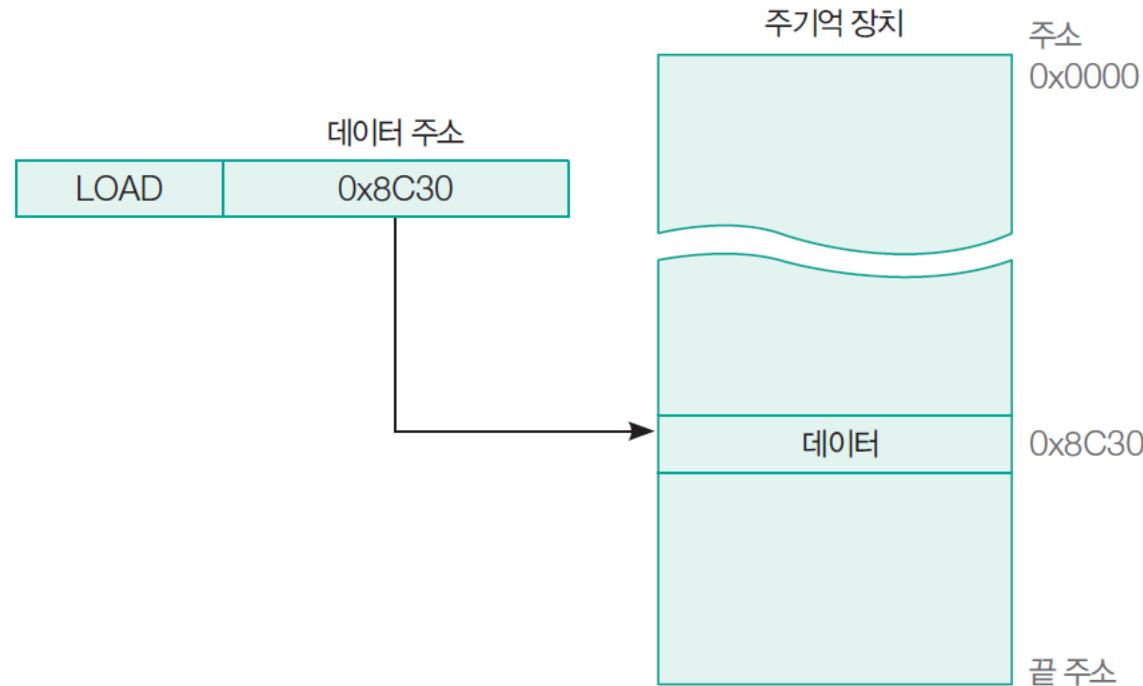


그림 4-14 직접 주소 지정

05 주소 지정 방식

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

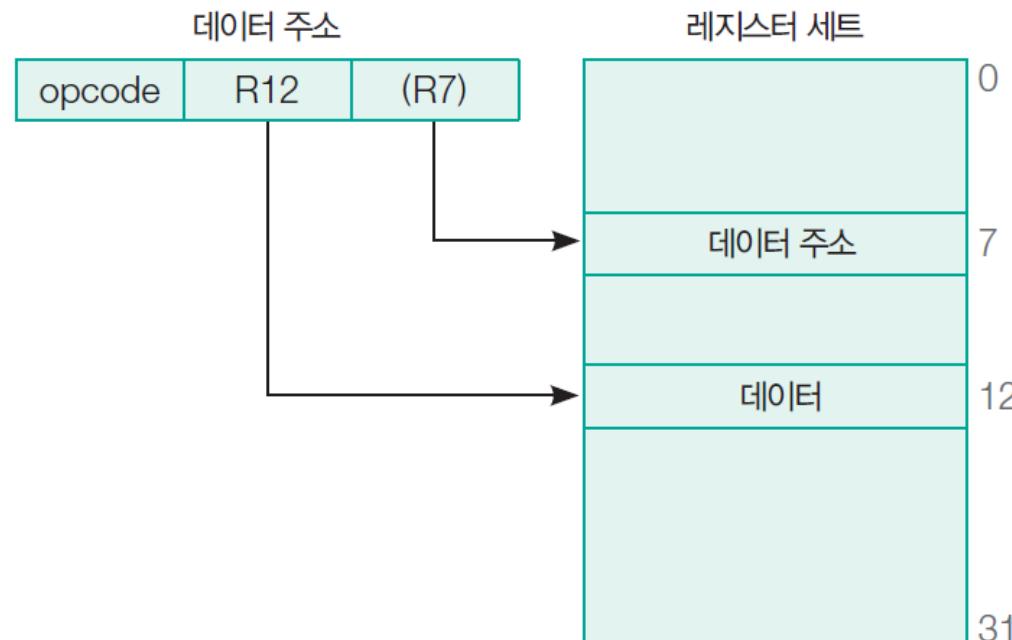


그림 4-15 레지스터 주소 지정

05 주소 지정 방식

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 참조 가능

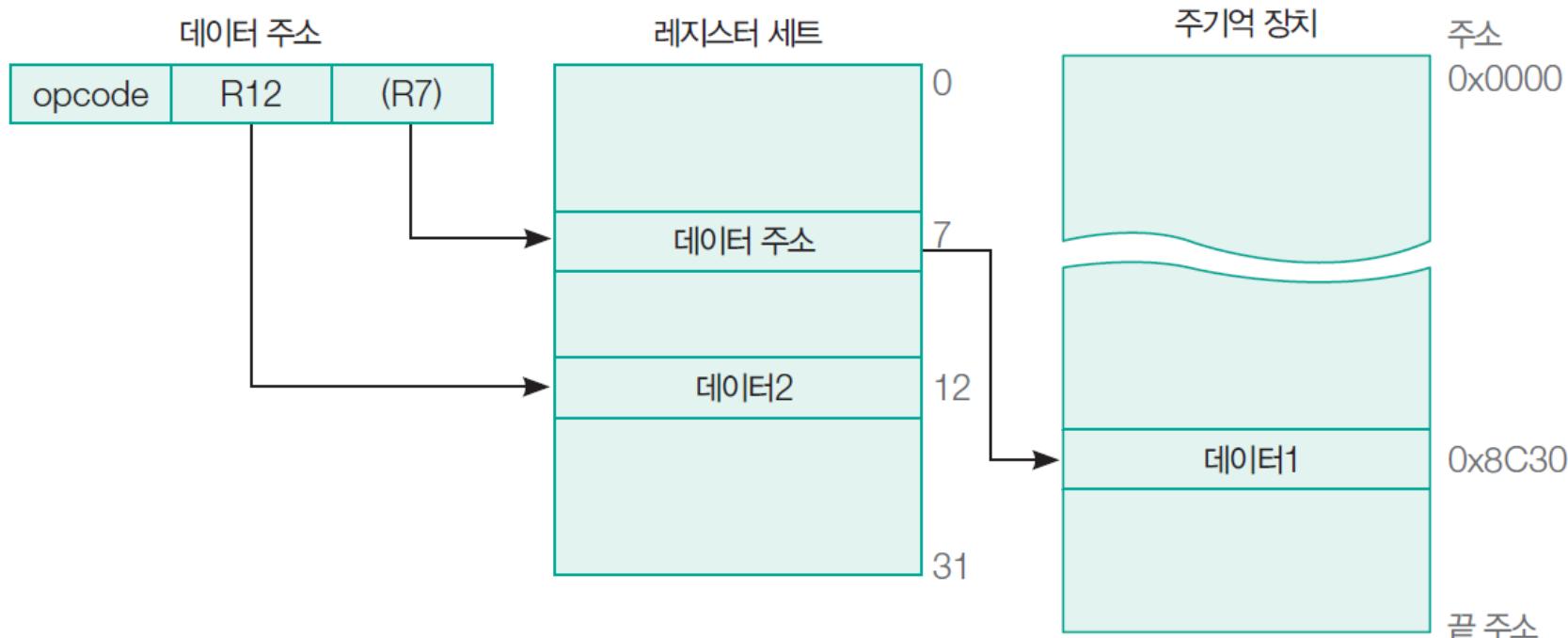


그림 4-16 레지스터 간접 주소 지정

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 초기값 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

05 주소 지정 방식

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

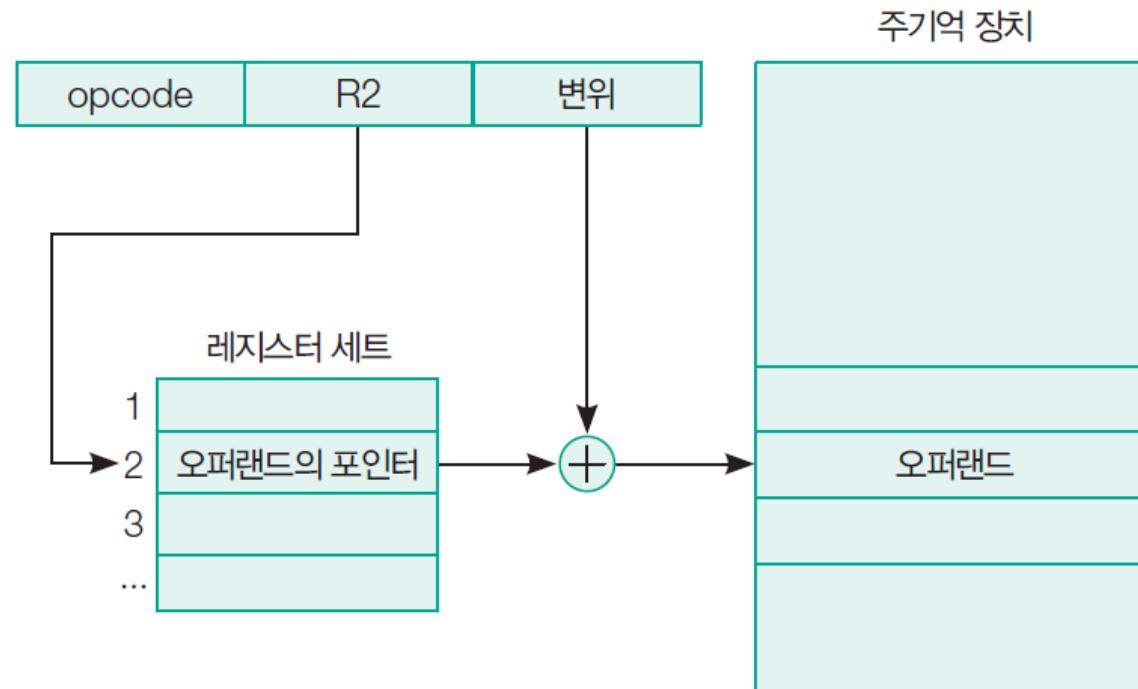


그림 4-17 변위 주소 지정

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, 초기값으로 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

① R1 : A의 누적 합계가 저장된다.

② R2 : 인덱스 레지스터로 배열의 i를 저장한다.

③ R3 : 상수 400

- 명령어 루프에 4개 실행

- 소스 값의 계산은 인덱스 주소지정 사용

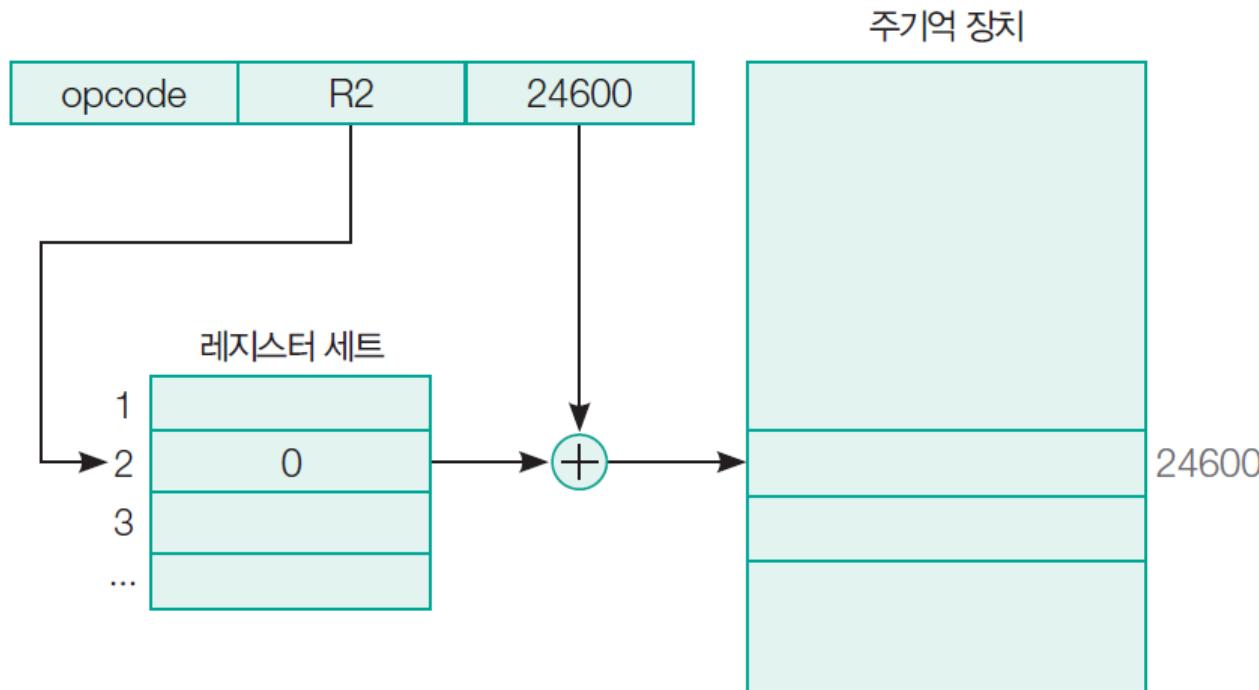
- 배열 A의 인덱스가 저장된 인덱스 레지스터 R2와 상수(0~400)가 더해져 메모리를 참조(A(R2))하는데 사용

- 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



05 주소 지정 방식

□ 상대 주소 지정(relative addressing mode)

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

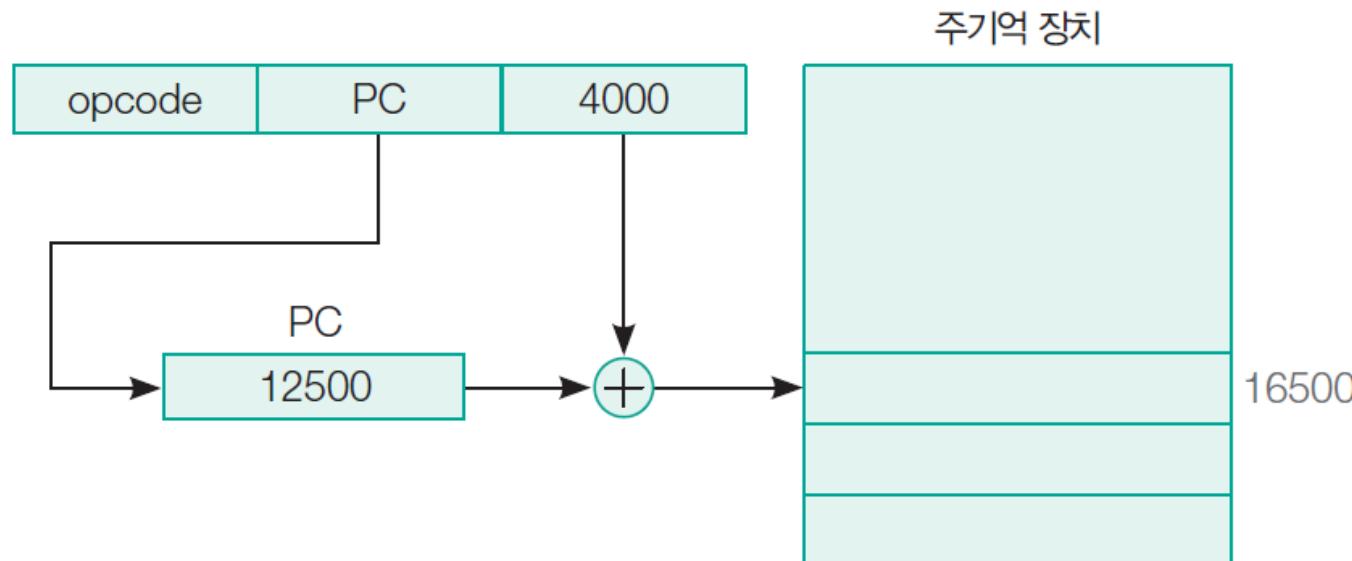


그림 4-19 상대 주소 지정

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

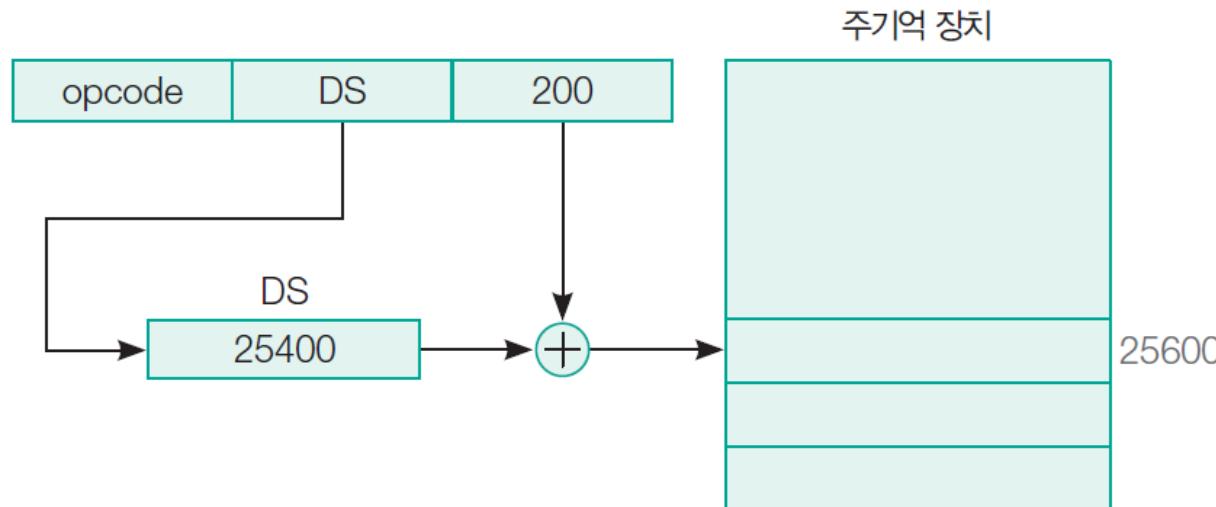


그림 4-20 베이스 주소 지정

05 주소 지정 방식

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

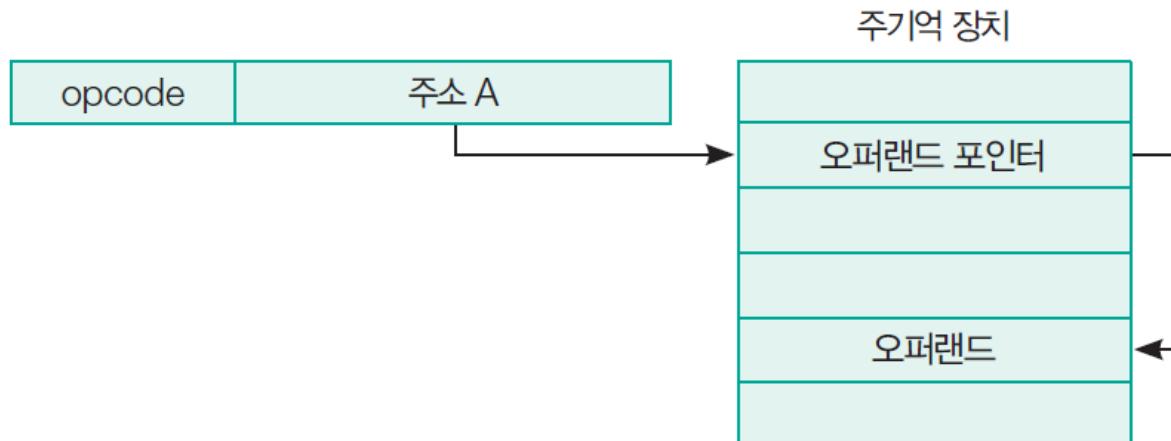


그림 4-21 간접 주소 지정

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

05 주소 지정 방식

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 | | ○ | |

□ CISC(Complex Instruction Set Computer)

❖ 1950년대 초 모리스 윌크스(M. V. Wilkes)

- 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

□ 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 헨네시(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 명령어가 강조
- 추후에는 빠르게 실행할 수 있는 명령어의 설계가 좋은 성능을 보장한다는 것을 깨달음
- 하나의 명령어가 실행되는 데 걸리는 시간보다 초당 얼마나 많은 명령어를 시작할 수 있는지가 더 중요함

❖ 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
 - 주 기억 장치의 속도가 제어 기억 장치의 속도와 비슷해짐으로써 CISC의 해독으로 인한 손실이 더 커짐

06 CISC와 RISC

❖ 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. |

❖ 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

06 CISC와 RISC

❖ 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.