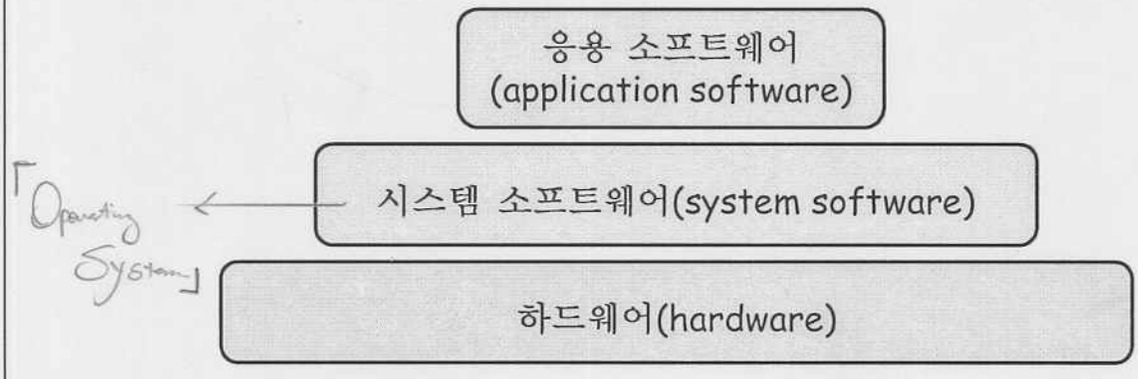
제1장. 컴퓨터시스템 개요

1.1 컴퓨터의 기본 구조

컴퓨터시스템의 구성



하드웨어와 소프트웨어

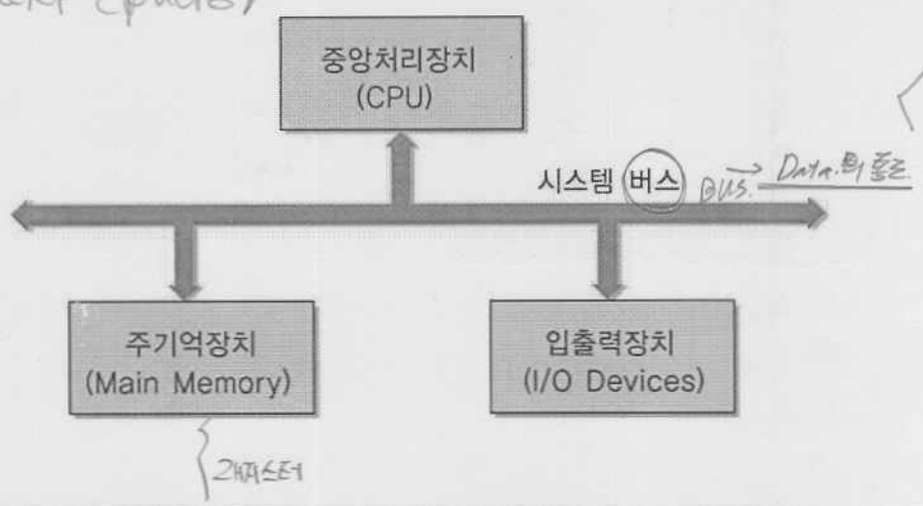
- 하드웨어(hardware)
  - -. 컴퓨터 정보들의 전송 통로를 제공해 주고, 그 정보에 대한 처리가 실제 일어나게 해주는 물리적인 실체들
- 소프트웨어(software)
  - -. 정보들이 이동하는 방향과 정보 처리의 종류를 지정해주고, 그러한 동작들이 일어나는 시간을 지정해주는 명령(command)들의 집합
  - -. 시스템 소프트웨어(system software) : OS(WinXP, Unix, Linux 등)
  - -. 응용 소프트웨어(application software) : 워드프로세서, 웹 브라우저 등

컴퓨터 하드웨어의 주요 요소들

- 컴퓨터 하드웨어
  - 중앙처리장치 : 펜티엄 Ⅳ, UltraSparc, MIPS R8000 등
  - 기억장치
    - 주기억장치 : RAM, ROM
    - 보조저장장치 : 디스크, CD-ROM 등
  - 입출력장치
    - 입력장치 : 키보드, 마우스 등
    - 출력장치 : 프린터, CRT, 스피커 등
  - 전원공급장치, 냉각 송풍기 등

컴퓨터의 기본 구조

- 컴퓨터는 프로그램 코드들을 정해진 순서대로 실행
  필요한 데이터를 읽어서(read), 처리(processing)하고, 저장(store)



컴퓨터의 주요 구성요소들

- 중앙처리장치(Central Processing Unit: CPU)
  - -. 프로세서(processor)
  - -. '프로그램 실행'과 '데이터 처리'라는 중추적인 기능의 수행을 담당하는 요소
- 기억장치(memory)
  - -. CPU가 실행할 프로그램과 데이터를 저장하는 장치
  - (1) 주기억장치(main memory)
    - CPU 가까이 위치하며 반도체 기억장치 칩들로 구성
    - 고속 액세스
    - 가격이 높고 면적을 많이 차지 → 저장 용량의 한계
    - 영구 저장 능력이 없기 때문에 프로그램 실행 중에 일시적으로만 사용

(2) 보조저장장치(auxiliary storage device)

- 2차 기억장치(secondary memory)
- 기계적인 장치가 포함되기 때문에 저속 액세스
- 저장 밀도가 높고, 비트 당 비용이 저가
- 영구 저장 능력을 가진 저장장치 : 디스크, 자기 테이프(magnetic tape) 등

- 입출력장치(I/O device)

  입력 장치(input device), 출력 장치(output device)

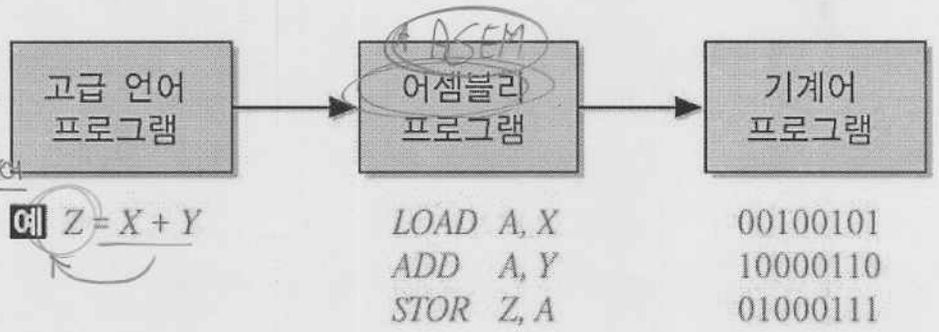
  사용자와 컴퓨터간의 대화를 위한 도구

1.2 정보의 표현과 저장

- 컴퓨터 정보
  - -. 2진수 비트(1과 0)들로 표현된 프로그램 코드와 데이터
- 프로그램 코드
  - -. 기계어(machine language)
    - 기계 코드(machine code), 컴퓨터 하드웨어 부품들이 이해할 수 있는 언어
    - 2진수 비트들로 구성
  - -. 어셈블리 언어(assembly language)
    - 어셈블리 코드(assembly code), 고급 언어와 기계어 사이의 중간 언어
    - 저급 언어(low-level language), 기계어와 1:1 대응
  - -. 고급 언어(high-level language)
    - 영문자와 숫자로 구성되어 사람이 이해하기 쉬운 언어
    - C, PASCAL, FORTRAN, COBOL 등

프로그램 언어의 변환 과정

- Z = X + Y
  - –. LOAD A,X : 기억장치 X번지의 내용을 읽어 레지스터 A에 적재(load)
  - –. ADD A,Y : 기억장치 Y번지 내용을 읽어 레지스터 A에 적재된 값과 더하고 결과를 다시 A에 적재
  - –. STOR Z,A : 그 값을 기억장치 Z 번지에 저장(store)



프로그램 언어 번역 소프트웨어

- 컴파일러(compiler)
  - –. 고급언어 프로그램을 기계어 프로그램으로 번역하는 소프트웨어
- 어셈블러(assembler)
  - –. 어셈블리 프로그램을 기계어 프로그램으로 번역하는 소프트웨어
  - –. 니모닉스(mnemonics)
    - 어셈블리 명령어가 지정하는 동작을 개략적으로 짐작할 수 있도록 하기 위하여 사용된 기호
    - 'LOAD', 'ADD', 'STOR' 등

기계 명령어의 형식

- 연산 코드(op code)

  CPU가 수행할 연산을 지정해 주는 비트들

  비트 수 = 3이면, 지정할 수 있는 연산의 최대 수는 $2^3 = 8$

- 오퍼랜드(operand)

  적재될 데이터가 저장된 기억장치 주소 혹은 연산에 사용될 데이터

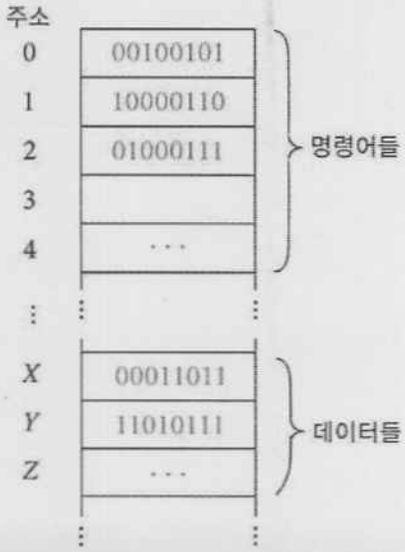
  비트의 수 = 5이면, 주소 지정할 수 있는 기억장소의 최대 수는 $2^5 = 32$

연산코드	오퍼랜드
0 0 1	0 0 1 0 1

프로그램 코드와 데이터의 기억장치 저장

- 단어(word)

  각 기억 장소에 저장되는 데이터의 기본 단위로서, CPU에 의해 한 번에 처리될 수 있는 비트들의 그룹



1.3 시스템의 구성

1.3.1 CPU와 기억장치의 접속

- 시스템 버스(system bus)

  -. CPU와 시스템 내의 다른 요소들 사이에 정보를 교환하는 통로

  -. 기본 구성

    - 주소 버스(address bus)
    - 데이터 버스(data bus)
    - 제어 버스(control bus)

시스템 버스

- 주소 버스(address bus)

  -. CPU가 외부로 발생하는 주소 정보를 전송하는 신호 선들의 집합

  -. 주소 선들의 수는 CPU와 접속될 수 있는 최대 기억장치 용량을 결정

    - 주소 버스의 비트 수 = 16 비트라면,
      최대 2^{16} = 64K 개의 기억 장소들의 주소를 지정 가능

- 데이터 버스(data bus)

  -. CPU가 기억장치 혹은 I/O 장치와의 사이에 데이터를 전송하기 위한 신호 선들의 집합

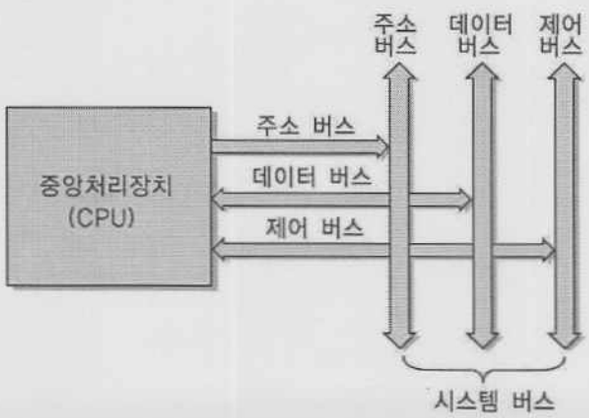
  -. 데이터 선들의 수는 CPU가 한 번에 전송할 수 있는 비트 수를 결정

    - 데이터 버스 폭 = 32 비트라면,
      CPU와 기억장치 간의 데이터 전송은 한 번에 32 비트씩 가능

- 제어 버스(control bus)
  - -. CPU가 시스템 내의 각종 요소들의 동작을 제어하기 위한 신호 선들의 집합
  - -. 기억장치 읽기/쓰기(Memory Read/Write) 신호
  - -. I/O 읽기/쓰기(I/O Read/Write) 신호

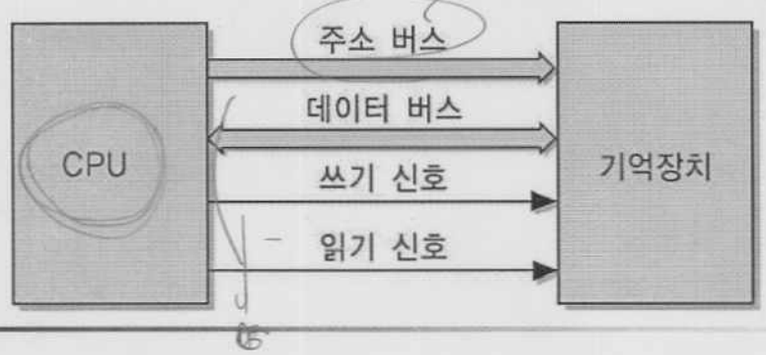
CPU와 시스템 버스

- 주소 버스 : 단방향성(uni-directional bus)
  - -. 주소가 CPU로부터 기억장치 혹은 I/O 장치들로 전송되는 정보이기 때문
- 데이터 버스, 제어 버스 : 양방향성(bi-directional)
  - -. 읽기와 쓰기를 모두 해야 하기 때문



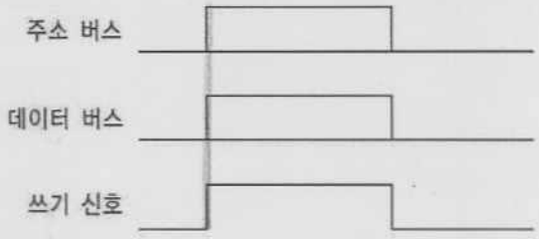
CPU와 기억장치

- 기억장치 쓰기 동작
  - -. CPU가 데이터를 저장할 기억 장소의 주소와 저장할 데이터를 각각 주소 버스와 데이터 버스를 통하여 보내면서 동시에 쓰기 신호를 활성화
  - -. 기억장치 쓰기 시간(memory write time)
    - CPU가 주소와 데이터를 보낸 순간부터 저장이 완료될 때까지의 시간

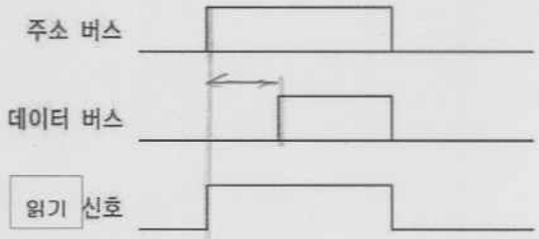


- 기억장치 읽기 동작
  - -. CPU가 기억장치 주소를 주소 버스를 통하여 보내면서 읽기 신호를 활성화
  - -. 일정 지연 시간이 경과한 후에 기억장치로부터 읽혀진 데이터가 데이터 버스 상에 실리며, CPU는 그 데이터를 버스 인터페이스 회로를 통하여 읽음
  - -. 기억장치 읽기 시간(memory read time)
    - 주소를 해독(decode)하는 데 걸리는 시간과 선택된 기억 소자들로부터 데이터를 읽는 데 걸리는 시간을 합한 시간

기억장치 액세스 동작의 시간 흐름도



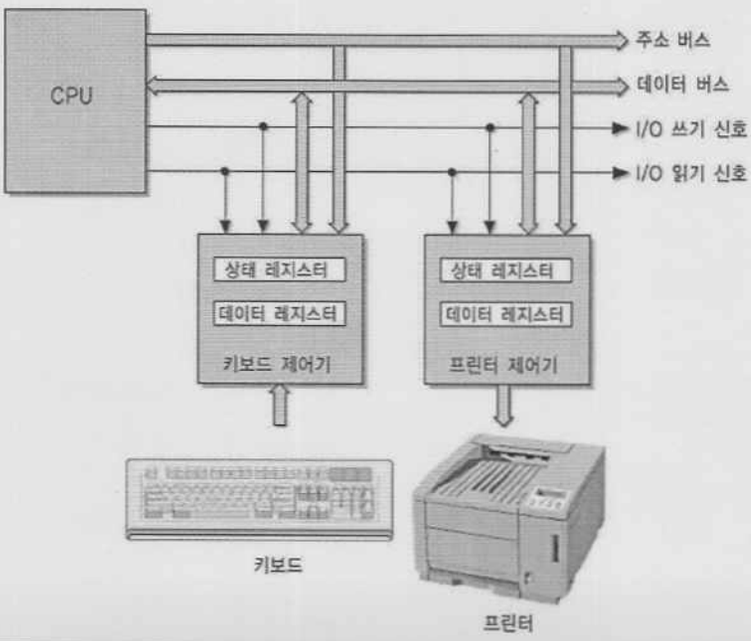
(a) 기억장치 쓰기 동작의 시간 흐름도



(b) 기억장치 읽기 동작의 시간 흐름도

1.3.2 CPU와 I/O 장치의 접속

- CPU – 시스템 버스 – I/O 장치 제어기 – I/O 장치

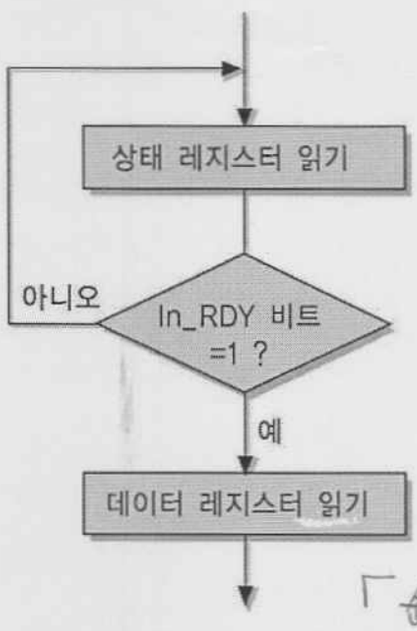


I/O 장치 제어기 (I/O device controller)

- CPU로부터 I/O 명령을 받아서, 해당 I/O 장치를 제어하고, 데이터를 이동함으로써 명령을 수행하는 전자회로 장치
  (키보드 제어기, 프린터 제어기 등)
  - 상태 레지스터
    -. I/O 장치의 현재 상태를 나타내는 비트들을 저장한 레지스터
    -. 준비 상태(RDY) 비트, 데이터 전송확인(ACK) 비트, 등
  - 데이터 레지스터
    -. CPU와 I/O 장치 간에 이동되는 데이터를 일시적으로 저장하는 레지스터

키보드의 데이터 입력 과정

- 키보드 제어기
  -. 키보드의 어떤 한 키(key)를 누르면,
  그 키에 대응되는 ASCII 코드가 키보드 제어기의 데이터 레지스터에 저장되고,
  동시에 상태 레지스터의 In_RDY 비트가 1로 세트
- CPU
  -. 키보드 제어기로부터 상태 레지스터의 내용을 읽어서 In_RDY 비트가 세트 되었는지 검사
  (In_RDY 비트는 데이터 레지스터에 외부로부터 데이터가 적재되었는지를 표시)
  -. 만약 세트 되지 않았으면, 1번을 반복하며 대기.
  만약 세트 되었다면, 데이터 레지스터의 내용을 읽음



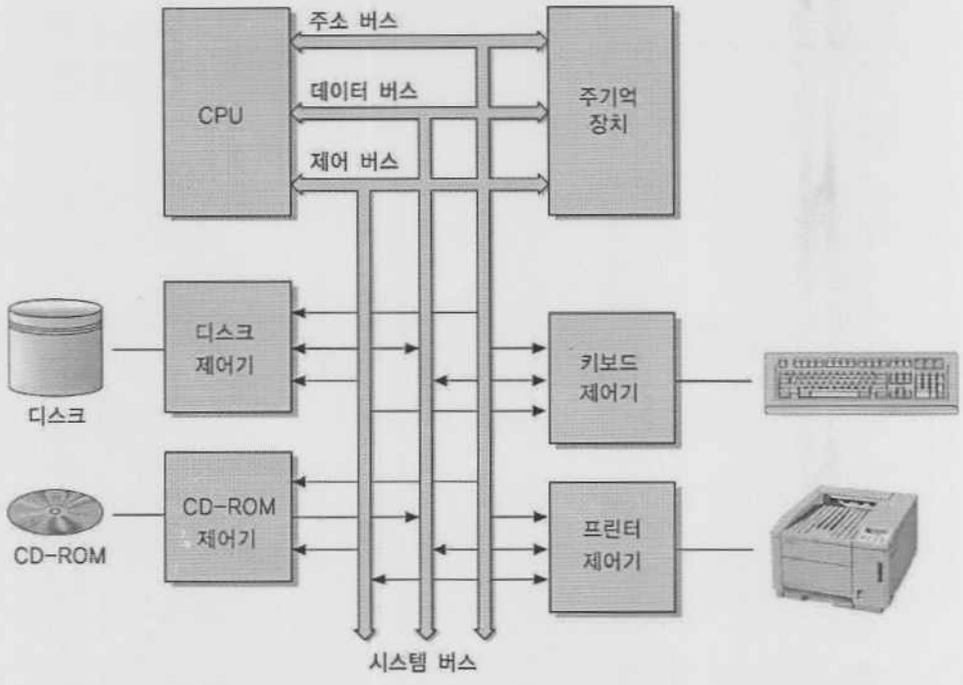
프린터의 데이터 출력 과정

- CPU :
  - -. 프린터 제어기의 상태 레지스터의 내용을 읽어서 Out_RDY 비트 검사 (Out_RDY 비트는 프린터가 출력할 준비가 되었는지를 표시)
  - -. 만약 세트 되지 않았으면, 1번을 반복하며 대기
    만약 세트 되었다면, 프린트할 데이터를 프린터 제어기의 데이터 레지스터에 씀
- 프린터 제어기 :
  - -. 데이터 레지스터의 내용을 프린터로 보내고, 프린터의 하드웨어를 제어하여 인쇄

CPU와 보조저장장치의 접속

- 보조저장장치들(디스크, 플로피 디스켓, CD-ROM 등)도 각 장치를 위한 제어기를 통하여 키보드나 프린터와 유사한 방법으로 접속
- 차이점 : 데이터 전송 단위
  - -. 키보드, 프린터 : 바이트(8 비트) 단위로 전송
  - -. 보조저장장치 : 블록(512/1024/4096 바이트) 단위로 전송
    → 제어기 내에 한 블록 이상을 임시 저장할 수 있는 데이터 버퍼 필요
    - 트랙 버퍼(track buffer)
      - 하드 디스크상의 한 트랙의 내용을 모두 저장할 수 있는 디스크 제어기내의 데이터 버퍼

1.3.3 컴퓨터시스템의 전체 구성

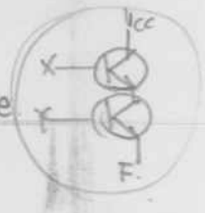


컴퓨터의 기본적인 기능들

- 프로그램 실행
  - -. CPU가 주기억장치로부터 프로그램 코드를 읽어서 실행
- 데이터 저장
  - -. 프로그램 실행 결과로서 얻어진 데이터를 주기억장치에 저장
- 데이터 이동
  - -. 디스크 혹은 CD-ROM에 저장되어 있는 프로그램과 데이터 블록을 기억장치로 이동
- 데이터 입력/출력
  - -. 사용자가 키보드를 통하여 보내는 명령이나 데이터를 읽어 들인다. 또한 CPU가 처리한 결과값이나 기억장치의 내용을 프린터(혹은 모니터)로 출력
- 제어
  - -. 프로그램이 순서대로 실행되도록 또는 필요에 따라 실행 순서를 변경하도록 조정하며, 각종 제어 신호들을 발생

09.09.04-TU.
A1037.

1.4 컴퓨터 구조의 발전 과정



- 주요 부품들의 발전 과정
  - –. 릴레이(relay) → 트랜지스터 → 반도체 집적회로(IC)
  - –. 발전된 특성들:
    - 처리속도 향상
    - 저장용량 증가
    - 크기 감소
    - 가격 하락
    - 신뢰도 향상
- 초기 컴퓨터들의 근본적인 설계 개념과 동작 원리가 현대의 컴퓨터들과 거의 같음

최초의 컴퓨터

- 1642년, Blaise Pascal(프랑스)
- 덧셈과 뺄셈을 수행하는 기계적 카운터
- 다이얼의 위치에 의하여 십진수를 표시하는 6개의 원형판 세트들로 구성
- 각 원형판은 일시적으로 숫자를 기억하는 레지스터로 사용

Leibniz의 기계

- 1671년, Gottfried Leibniz(독일)
- 덧셈과 뺄셈 및 곱셈과 나눗셈도 할 수 있는 계산기
- Pascal의 계산기에 두 개의 원형판들을 추가하여 반복적 방법으로 곱셈과 나눗셈을 수행
- 이후 많은 기계들의 조상이 됨

Difference Engine

- 19세기 초, Charles Babbage(영국, 현대 컴퓨터의 할아버지)
- 표에 있는 수들을 자동적으로 계산하고, 그 결과를 금속천공기를 거쳐서 프린트
- 덧셈과 뺄셈만 수행 가능

Analytical Engine

- 19세기 초, Charles Babbage(영국)
- 주요 특징들
  - -. 어떤 수학 연산도 자동적으로 수행할 수 있는 일반 목적용 계산 기계
  - -. 프로그래밍 가능 : 프로그램 언어 사용
  - -. 프로그램의 실행 순서 변경 가능
    - 수의 부호 검사를 이용한 조건 분기
    - 제어카드 이용을 이용한 실행 순서 변경
- 문제점
  - -. 주요 부품들이 기계적인 장치들이었기 때문에 속도가 느렸고 신뢰도가 낮음

Analytical Engine의 기본 구조

- 산술연산장치 : CPU
- 기억장치 : Store
- 입력장치: 카드판독기
- 출력장치: 프린터, 카드 천공기

ENIAC

- Electronic Numerical Integrator And Computer
- 1940년대 초, von Neumann(폰 노이만)
- 펜실바니아 대학에서 개발한 진공관을 사용한 최초의 전자식 컴퓨터
- 문제점 : 프로그램의 저장과 변경 불가능
- 폰 노이만의 설계 개념(Stored-program 개념)
  - -. 2진수 체계(binary number system)를 사용
  - -. 프로그램과 데이터를 내부에 저장
  - -. EDVAC(Electronic Discrete Variable Computer) 개발을 위하여 1945년에 발표

IAS 컴퓨터

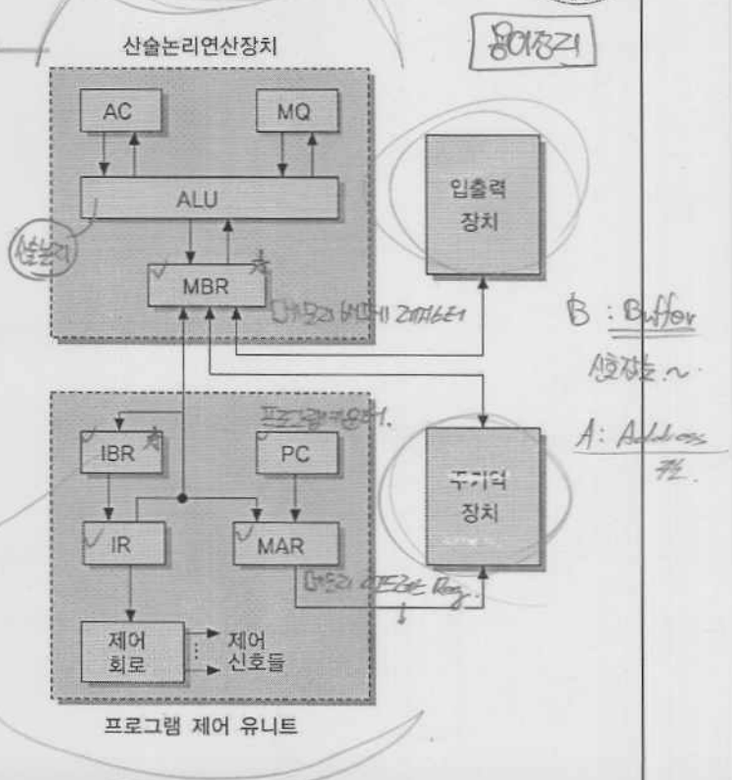
- 1952년, 폰 노이만
- 'stored-program' 컴퓨터
- 주요 구성요소
  - -. 프로그램 제어 유니트(Program Control Unit) : 명령어 인출/해독
  - -. 산술논리연산장치(ALU)
  - -. 주기억장치 : 명령어와 데이터를 모두 저장
  - -. 입출력장치
- 주요 특징
  - -. 주기억장치로부터 한 번에 두 개씩 명령어 인출
    - 하나는 즉시 프로그램 제어 유니트로 보내져서 실행
    - 다른 하나는 명령어 버퍼에 저장되어 있다가 다음 명령어 실행 사이클에서 실행
  - -. 최근 프로세서들의 명령어 선인출(instruction prefetch)과 같은 개념

IAS 컴퓨터의 구조

- 폰 노이만 구조

  von Neumann Architecture,

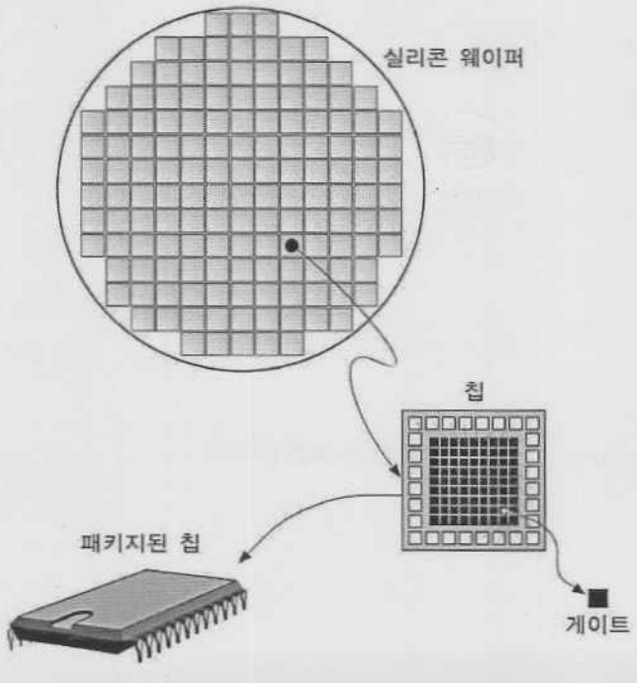
  프로그램 코드들을 기억장치에 저장된 순서대로 실행하며, 그 주소는 CPU의 내부 레지스터인 프로그램 카운터 (program ccunter)에 의하여 지정



1.4.2 주요 컴퓨터 부품들의 발전 경위

- 트랜지스터(transistor)
  - –. 초기(제1세대) 전자식 컴퓨터의 핵심 부품인 진공관을 대체한 전자 부품
  - –. 진공관보다 작고 싸며 더 적은 열을 발산
  - –. 반도체 재료인 실리콘(Si)으로 만들어진 고체(solid-state) 장치
  - –. 제2세대 컴퓨터로 분류 ← 제1세대 컴퓨터들의 부품은 진공관
  - –. 초기 컴퓨터들은 약 1000 개의 트랜지스터들로 구성
- 집적 회로(Integrated Circuit: IC)
  - –. 수만 개 이상의 트랜지스터들을 하나의 반도체 칩에 집적시킨 전자 부품
  - –. 제3세대 컴퓨터로 분류

IC의 제조 과정



집적도에 따른 IC의 분류

- SSI(Small Scale IC)
  - -. 수십 개의 트랜지스터들이 집적되는 소규모 IC
  - -. 최근에는 주로 기본적인 디지털 게이트(digital gate)들을 포함하는 칩
- MSI(Medium Scale IC)
  - -. 수백 개의 트랜지스터들이 집적되는 IC
  - -. 카운터(counter), 해독기(decoder) 또는 쉬프트 레지스터(shift register)와 같은 조합 회로나 순차 회로를 포함하는 칩
- LSI(Large Scale IC)
  - -. 수천 개의 트랜지스터들이 집적되는 대규모 IC
  - -. 8-비트 마이크로프로세서 칩이나 소규모 반도체 기억장치 칩

집적도에 따른 IC의 분류 (계속)

- VLSI(Very Large Scale IC)
  - -. 수만 내지 수십만 개 이상의 트랜지스터들이 집적되는 초대규모 IC
  - -. 마이크로프로세서 칩들과 대용량 반도체 기억장치 칩
- ULSI(Ultra Large Scale IC)
  - -. 수백만 개 이상의 트랜지스터들이 집적되는 32-비트급 이상 마이크로프로세서 칩들과 수백 메가비트 이상의 반도체 기억장치 칩들 및 앞으로 출현할 고밀도 반도체 칩들을 지칭하기 위한 용어지만, 아직 일반적으로 통용되는 용어는 아님

IC 사용에 따른 이점

- 전기적 통로가 짧아짐 → 동작 속도가 크게 상승
- 컴퓨터 크기의 감소
- 칩 내부의 회로들간의 상호연결 → 부품들의 신뢰성 향상
- 전력 소모 감소 및 냉각 장치의 소형화
- 컴퓨터 가격 하락
- 제4세대 컴퓨터 시대의 시작
  - -. 개인용 컴퓨터 출현
  - -. 초고속 슈퍼컴퓨터 개발

1.4.3 컴퓨터시스템의 분류와 발전 동향

1) 개인용 컴퓨터 (PC)

- 특징
  - -. 소형, 저가
  - -. 성능 : 십년 전의 대형 메인프레임 컴퓨터의 성능을 능가
- 주요 발전 동향
  - -. 매 2 ~ 3 년마다 성능이 개선된 새로운 마이크로프로세서가 등장하고, 그에 따라 새로운 PC 모델 출현
  - -. 주변 요소들(캐쉬, MMU, 산술보조프로세서 등)이 CPU 칩에 내장됨에 따라 속도 및 신뢰도가 향상
  - -. CPU 구조가 다수의 ALU들 혹은 명령어 실행 유니트들을 포함하는 슈퍼스칼라(superscalar) 구조로 발전함에 따라, 여러 명령어들의 동시 실행 가능

개인용 컴퓨터 (계속)

-. 분기 예측(branch prediction), 동적 실행(dynamic execution) 기법 등이 사용됨에 따라 하드웨어 이용률이 높아지고, 따라서 처리 속도가 더욱 향상

-. 문자 이외의 다양한 정보들에 대한 입력과 출력, 저장 및 처리 능력을 보유하게 됨에 따라 멀티미디어 PC로 발전

-. 보다 더 편리한 사용자 인터페이스를 제공해 주는 시스템 소프트웨어들 출현 (Windows 95/98/ME/2000/XP)

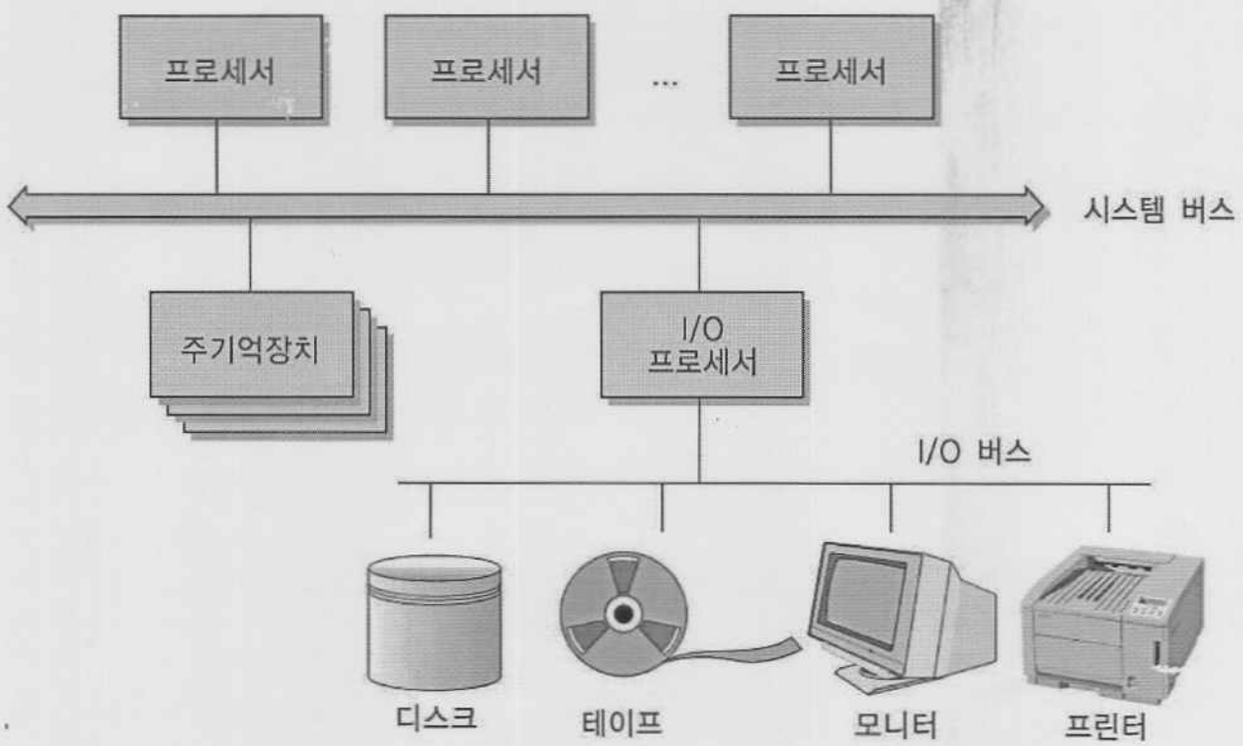
-. 고속 입출력장치들의 인터페이스를 위한 새로운 버스 규격 제안

-. 주기억장치와 보조저장장치의 용량이 크게 증가

2) 중형급 컴퓨터시스템

- 워크스테이션(workstation)
  - -. CPU : 32-비트 혹은 64-비트 마이크로프로세서 사용
  - -. 고속 그래픽 처리 하드웨어 포함
  - -. 주요 응용 : 컴퓨터를 이용한 설계(CAD), 시뮬레이션 등
  - -. OS : UNIX
- 슈퍼미니컴퓨터(Super-minicomputer)
  - -. 시스템 구조 : 다중프로세서(multiprocessor) 구조
  - -. CPU의 수 : 20 ~ 30 개
  - -. 성능 : VAX-11 미니컴퓨터 성능의 수십 배 이상
  - -. OS : UNIX
  - -. 다운사이징(downsizing) 주도
    - → 네트워크에 접속된 다수의 중형급 컴퓨터 시스템들을 응용(혹은 용도)별로 구분하여 사용하는 컴퓨팅 환경이 가능해지게 함

다중프로세서시스템의 구조



3) 메인프레임 컴퓨터(mainframe computer)

- IBM 360 및 370 계열, 3081, 3090 등으로 계속 발전
- 대용량 저장장치 보유
- 다중 I/O 채널을 이용한 고속 I/O 처리 능력 보유
- 대규모 데이터베이스 저장 및 관리용으로 사용
- 최근 성능과 가격면에서 슈퍼미니급 컴퓨터들과 경쟁하고 있으며, 점차적으로 시장 점유율 하락 중

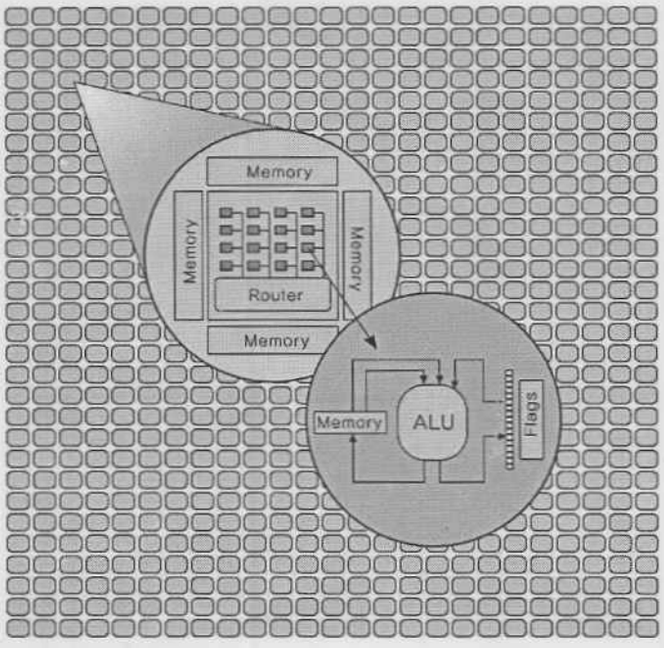
4) 슈퍼컴퓨터(supercomputer)

- 현존하는 컴퓨터들 중에서 처리 속도와 기억장치 용량이 다른 컴퓨터들에 비하여 상대적으로 월등한 컴퓨터 시스템들
- 분류 기준 : 계속적으로 상승
  - -. 최초의 슈퍼컴퓨터인 CRAY-1의 속도는 100 MFLOPS
  - -. 최근의 슈퍼컴퓨터들의 속도는 수백 GFLOPS 이상
- 주요 응용 분야들
  - -. VLSI 회로 설계, 항공우주공학, 천문학(일기 예보), 구조 공학, 유전 탐사, 핵공학, 인공지능, 입체 영상처리 등과 같은 대규모 과학 계산 및 시뮬레이션

슈퍼컴퓨터(계속): 구조에 따른 분류

- 파이프라인 슈퍼컴퓨터(pipeline supercomputer)
  - -. 한 CPU 내에 다수의 연산 장치들이 포함
  - -. 각 연산 장치는 고도의 파이프라이닝 구조를 이용하여 고속 벡터 계산 가능
  - -. 대표적인 시스템들 : CRAY Y-MP, CRAY-2, Fujitsu VP2000, VPP500 등
- 대규모 병렬컴퓨터(massively parallel computer: MPP)
  - -. 한 시스템 내에 상호 연결된 수백 혹은 수천 개 이상의 프로세서들 포함
  - -. 프로세서들이 하나의 큰 작업을 나누어서 병렬로 처리
  - -. 시스템 구조의 예 : Thinking Machine사의 CM-1 시스템
    - 특징 : 프로세서 수 = 최대 65,536(2^{16})개
    - 프로세서 내부 구조
      - 매우 간단한 구조의 프로세서 16개가 상호 연결되어 하나의 칩에 집적되어 있으며, 그러한 칩들이 수백개가 모여 한 프로세서 모듈을 구성

MPP의 예: CM-1 병렬컴퓨터의 프로세서 모듈



제2장 CPU의 구조와 기능

2.1 CPU의 기본 구조
2.2 명령어 실행
2.3 명령어 파이프라이닝
2.4 명령어 세트

CPU의 기능

- 명령어 인출(Instruction Fetch) : 기억장치로부터 명령어를 읽어온다
- 명령어 해독(Instruction Decode) : 수행해야 할 동작을 결정하기 위하여 인출된 명령어를 해독한다

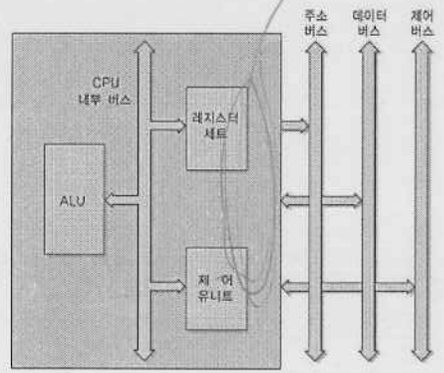
➔ 모든 명령어들에 대하여 공통적으로 수행

- 데이터 인출(Data Fetch) : 명령어 실행을 위하여 데이터가 필요한 경우에는 기억장치 또는 입출력장치로부터 그 데이터를 읽어온다
- 데이터 처리(Data Process) : 데이터에 대한 산술적 또는 논리적 연산을 수행
- 데이터 쓰기(Data Store) : 수행한 결과를 저장

➔ 명령어에 따라 필요한 경우에만 수행

2.1 CPU의 기본 구조

- 산술논리연산장치(Arithmetic and Logical Unit: ALU)
- 레지스터 세트(Register Set)
- 제어 유니트(Control Unit)



CPU의 내부 구성요소

- ALU
  - -. 각종 산술 연산들과 논리 연산들을 수행하는 회로들로 이루어진 하드웨어 모듈
  - -. 산술 연산 : +, -, ×, ÷
  - -. 논리 연산 : AND, OR, NOT, XOR 등
- 레지스터 세트 (register set)
  - -. CPU 내부 레지스터들의 집합.
  - -. 컴퓨터의 기억장치들 중에서 액세스 속도가 가장 빠름
  - -. CPU 내부에 포함할 수 있는 레지스터들의 수가 제한됨 (특수 목적용 레지스터들과 적은 수의 일반 목적용 레지스터들)

CPU의 내부 구성요소 (계속)

- 제어 유니트
  - -. 프로그램 코드(명령어)를 해석하고, 그것을 실행하기 위한 제어 신호들(control signals)을 순차적으로 발생하는 하드웨어 모듈
- 내부 CPU 버스(internal CPU bus):
  - -. ALU와 레지스터들 간의 데이터 이동을 위한 데이터 선들과 제어유니트로부터 발생되는 제어 신호 선들로 구성된 내부 버스
  - -. 외부의 시스템 버스들과는 직접 연결되지 않으며, 반드시 버퍼 레지스터들 혹은 시스템 버스 인터페이스 회로를 통하여 시스템 버스와 접속.

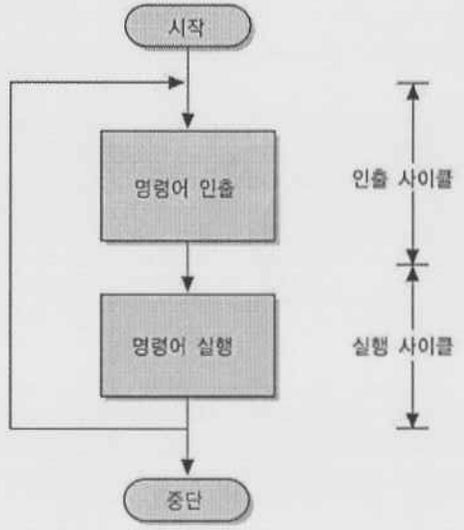
2.2 명령어 실행

명령어 사이클 (instruction cycle)

- CPU가 한 개의 명령어를 실행하는 데 필요한 전체 처리 과정으로서, CPU가 프로그램 실행을 시작한 순간부터 전원을 끄거나 회복 불가능한 오류가 발생하여 중단될 때까지 반복

- 부사이클(subcycle)
  - -. 인출 사이클(fetch cycle) : CPU가 기억장치로부터 명령어를 읽어오는 단계
  - -. 실행 사이클(execution cycle) : 명령어를 실행하는 단계

기본 명령어 사이클



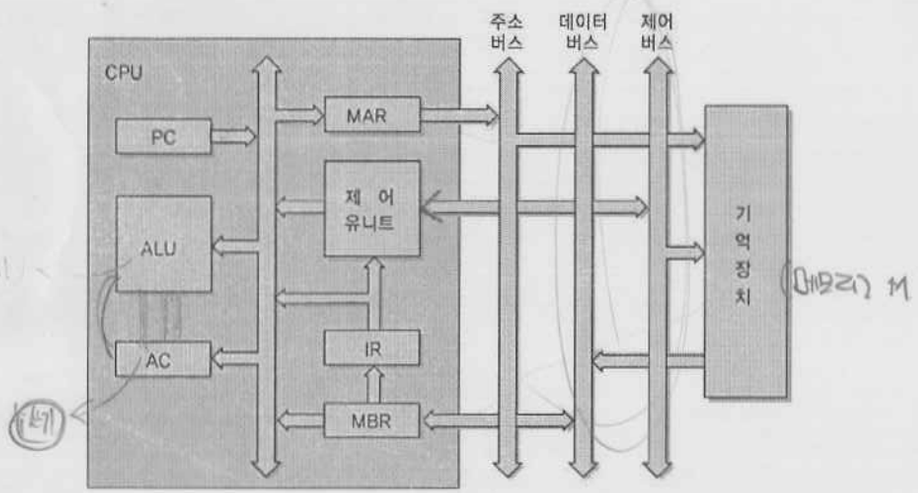
명령어 실행에 필요한 CPU 내부 레지스터들

- 프로그램 카운터(Program Counter: PC)
  - -. 다음에 인출할 명령어의 주소를 가지고 있는 레지스터
  - -. 각 명령어가 인출된 후에는 자동적으로 일정 크기(한 명령어 길이)만큼 증가
  - -. 분기(branch) 명령어가 실행되는 경우에는 목적지 주소로 갱신
- 누산기(Accumulator: AC)
  - -. 데이터를 일시적으로 저장하는 레지스터.
  - -. 레지스터의 크기는 CPU가 한 번에 처리할 수 있는 데이터 비트 수(단어 길이)
- 명령어 레지스터(Instruction Register: IR)
  - -. 가장 최근에 인출된 명령어 코드가 저장되어 있는 레지스터

명령어 실행에 필요한 CPU 내부 레지스터들 (계속)

- 기억장치 주소 레지스터(Memory Address Register: MAR)
  - -. PC에 저장된 명령어 주소가 시스템 주소 버스로 출력되기 전에 일시적으로 저장되는 주소 레지스터
- 기억장치 버퍼 레지스터(Memory Buffer Register: MBR)
  - -. 기억장치에 쓰여질 데이터 혹은 기억장치로부터 읽혀진 데이터를 일시적으로 저장하는 버퍼 레지스터

데이터 통로가 표시된 CPU 내부 구조



2.2.1 인출 사이클

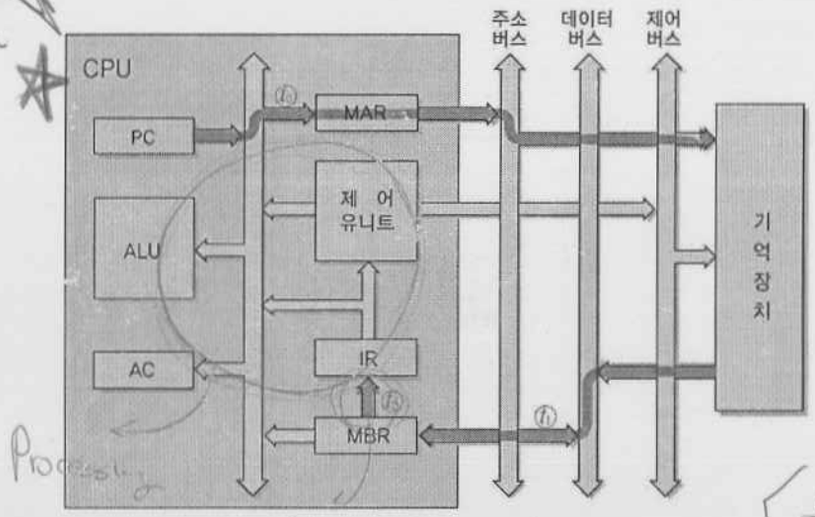
인출 사이클의 마이크로 연산

t0 : MAR ← PC
t1 : MBR ← M[MAR], PC ← PC + 1
t2 : IR ← MBR
단, t0, t1 및 t2는 CPU 클럭의 주기

[첫번째 주기] 현재의 PC 내용을 CPU 내부 버스를 통하여 MAR로 전송
[두번째 주기] 그 주소가 지정하는 기억장치 위치로부터 읽혀진 명령어가 데이터 버스를 통하여 MBR로 적재되며, PC의 내용에 1을 더한다
[세번째 주기] MBR에 있는 명령어 코드가 명령어 레지스터인 IR로 이동

(예) CPU 클럭 = 100㎒ (클럭 주기 = 10ns)
→ 인출 사이클 : 10ns x 3 = 30ns 소요

인출 사이클의 주소 및 명령어 흐름도



2.2.2 실행 사이클

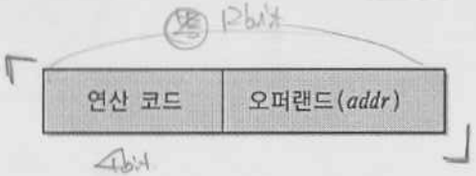
- CPU는 실행 사이클 동안에 명령어 코드를 해독(decode)하고, 그 결과에 따라 필요한 연산들을 수행
- CPU가 수행하는 연산들의 종류
  - -. 데이터 이동 : CPU와 기억장치 간 혹은 I/O장치 간에 데이터를 이동
  - -. 데이터 처리 : 데이터에 대하여 산술 혹은 논리 연산을 수행
  - -. 데이터 저장 : 연산 결과 데이터 혹은 입력장치로부터 읽어 들인 데이터를 기억장치에 저장
  - -. 제어 : 프로그램의 실행 순서를 결정
- 실행 사이클에서 수행되는 마이크로-연산들은 명령어에 따라 다름

기본적인 명령어 형식의 구성

- 연산 코드(operation code)

  CPU가 수행할 연산을 지정
- 오퍼랜드(operand)

  명령어 실행에 필요한 데이터가 저장된 주소(*addr*)



[사례 1] LOAD *addr* 명령어

- 기억장치에 저장되어 있는 데이터를 CPU 내부 레지스터인 AC로 이동하는 명령어

t0 : MAR ← IR(*addr*)
t1 : MBR ← M[MAR]
t2 : AC ← MBR

[첫번째 주기] 명령어 레지스터 IR에 있는 명령어의 주소 부분을 MAR로 전송
[두번째 주기] 그 주소가 지정한 기억장소로부터 데이터를 인출하여 MBR 전송
[세번째 주기] 그 데이터를 AC에 적재

[사례 2] STA *addr* 명령어

- AC 레지스터의 내용을 기억장치에 저장하는 명령어

t0 : MAR ← IR(*addr*)
t1 : MBR ← AC
t2 : M[MAR] ← MBR

[첫번째 주기] 데이터를 저장할 기억장치의 주소를 MAR로 전송
[두번째 주기] 저장할 데이터를 버퍼 레지스터인 MBR로 이동
[세번째 주기] MBR의 내용을 MAR이 지정하는 기억장소에 저장

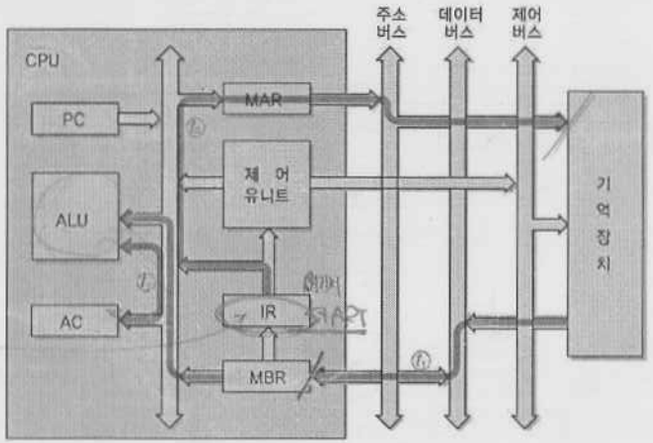
[사례 3] ADD *addr* 명령어

- 기억장치에 저장된 데이터를 AC의 내용과 더하고, 그 결과는 다시 AC에 저장하는 명령어

t0 : MAR ← IR(*addr*)
t1 : MBR ← M[MAR]
t2 : AC ← AC + MBR

[첫번째 주기] 데이터를 저장할 기억장치의 주소를 MAR로 전송
[두번째 주기] 저장할 데이터를 버퍼 레지스터인 MBR로 이동
[세번째 주기] 그 데이터와 AC의 내용을 더하고 결과값을 다시 AC에 저장

ADD 명령어 실행 사이클 동안의 정보 흐름



[사례 4] JUMP *addr* 명령어

- 오퍼랜드(*addr*)가 가리키는 위치의 명령어로 실행 순서를 변경하는 분기(branch) 명령어

t0 : PC ← IR(*addr*)

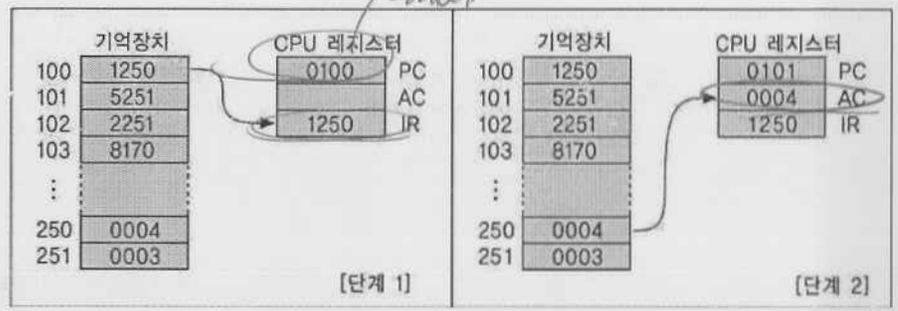
-. 명령어의 오퍼랜드(분기할 목적지 주소)가 PC에 저장
-. 다음 명령어 인출 사이클에서 그 주소의 명령어가 인출되므로 분기가 발생

어셈블리 프로그램 실행과정의 예

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

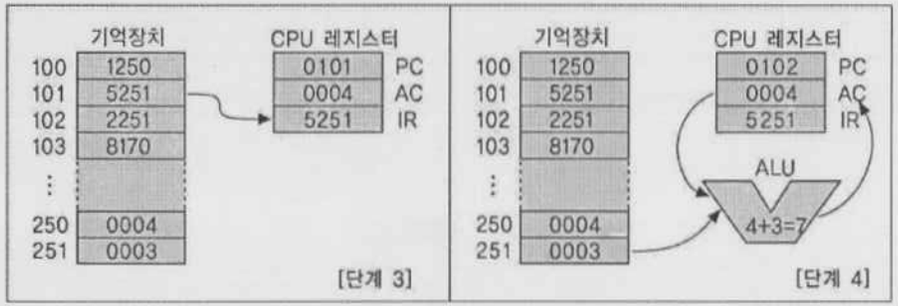
어셈블리 프로그램 실행과정의 예 (계속)

- 100번지의 첫 번째 명령어 코드가 인출되어 IR에 저장
- 250 번지의 데이터를 AC로 이동
- PC = PC + 1 = 101



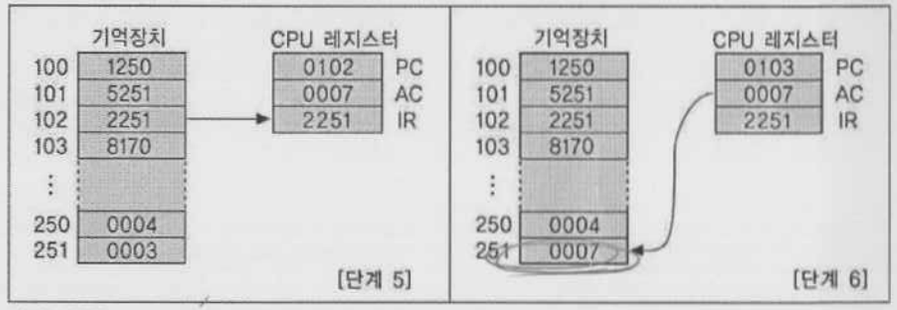
어셈블리 프로그램 실행과정의 예 (계속)

- 두 번째 명령어가 101번지로부터 인출되어 IR에 저장
- AC의 내용과 251 번지의 내용을 더하고, 결과를 AC에 저장
- PC의 내용은 102로 증가



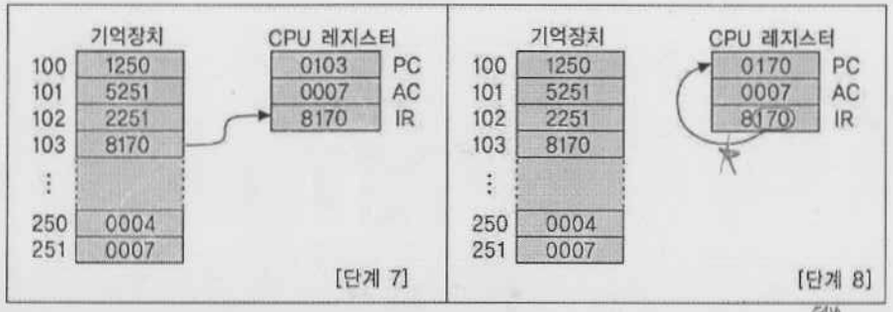
어셈블리 프로그램 실행과정의 예 (계속)

- 세 번째 명령어가 102 번지로부터 인출되어 IR에 저장
- AC의 내용이 251 번지에 저장
- PC의 내용은 103으로 증가



어셈블리 프로그램 실행과정의 예 (계속)

- 네 번째 명령어가 103 번지로부터 인출되어 IR에 저장
- 분기될 목적지 주소, 즉 IR의 하위 부분(170)이 PC로 적재
  (다음 명령어 인출 사이클에서는 170 번지의 명령어가 인출)

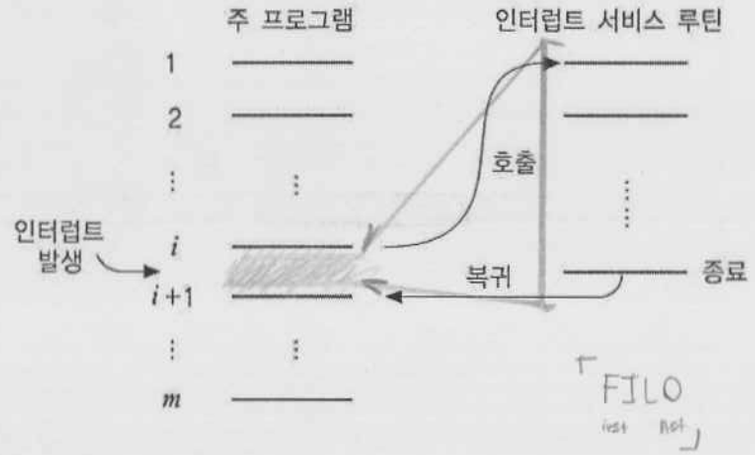


2.2.3 인터럽트 사이클(interrupt cycle)

인터럽트

- 프로그램 실행 중에 CPU의 현재 처리 순서를 중단시키고 다른 동작을 수행하도록 하는 것
- 외부로부터 인터럽트 요구가 들어오면,
  - -. CPU는 원래의 프로그램 수행을 중단하고,
  - -. 요구된 인터럽트를 위한 서비스 프로그램을 먼저 수행
- 인터럽트 서비스 루틴(interrupt service routine: ISR)
  - -. 인터럽트를 처리하기 위하여 수행하는 프로그램 루틴.

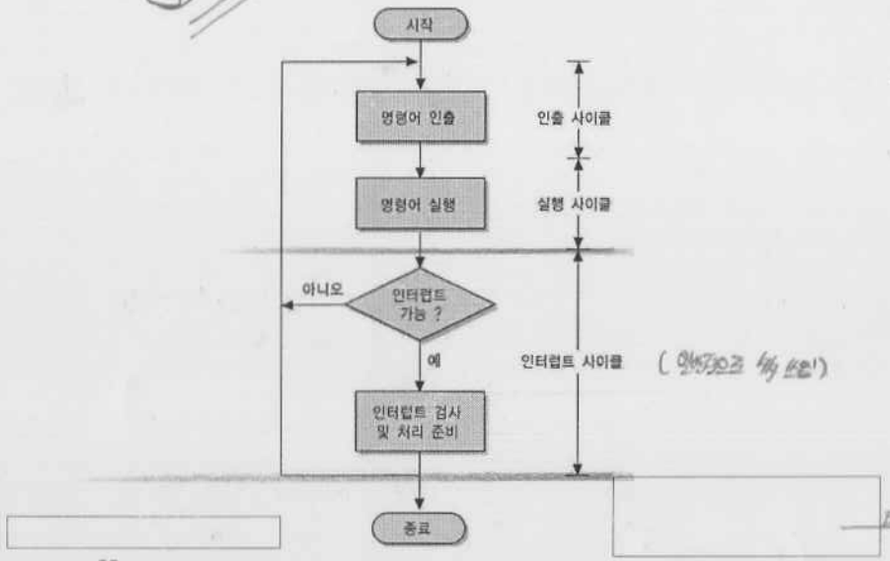
인터럽트에 의한 제어의 이동



인터럽트 처리

- 인터럽트가 들어왔을 때 CPU는
  - -. 어떤 장치가 인터럽트를 요구했는지 확인하여 해당 인터럽트서비스루틴을 호출
  - -. 서비스가 종료된 다음에는 중단되었던 원래 프로그램의 수행을 계속
- CPU의 인터럽트 처리 동작
  - -. 현재의 명령어 실행을 끝낸 즉시, 다음에 실행할 명령어의 주소(PC의 내용)를 스택(stack)에 저장 → 일반적으로 스택은 주기억장치의 특정 부분
  - -. 인터럽트 서비스 루틴을 호출하기 위하여 그 루틴의 시작 주소를 PC에 적재, 이때 시작 주소는 인터럽트를 요구한 장치로부터 전송되거나 미리 정해진 값으로 결정 → 자세한 사항은 제7장에서 설명

인터럽트 사이클이 추가된 명령어 사이클



인터럽트 사이클의 마이크로 연산

t0 : MBR ← PC

t1 : MAR ← SP, PC ← ISR의 시작 주소

t2 : M[MAR] ← MBR

단, SP는 스택 포인터(stack pointer)

[첫번째 주기] PC의 내용이 MBR로 전송

[두번째 주기] SP의 내용이 MAR로 전송되고,
PC의 내용은 인터럽트 서비스 루틴의 시작 주소로 변경

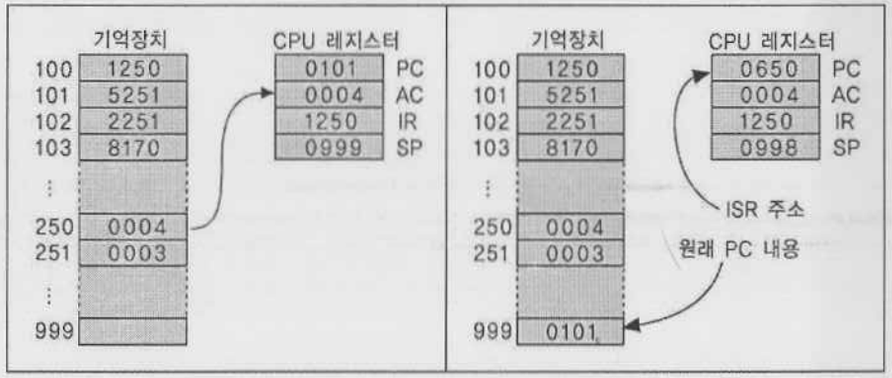
[세번째 주기] MBR에 저장되어 있던 원래 PC의 내용이 스택에 저장

인터럽트 사이클의 마이크로 연산 예

- 아래 프로그램의 첫 번째 명령어인 LOAD 250 명령어가 실행되는 동안에 인터럽트가 들어왔으며, SP = 999, 인터럽트 서비스 루틴의 시작 주소 = 650 번지라고 가정

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


인터럽트 요구가 들어온 경우의 상태 변화



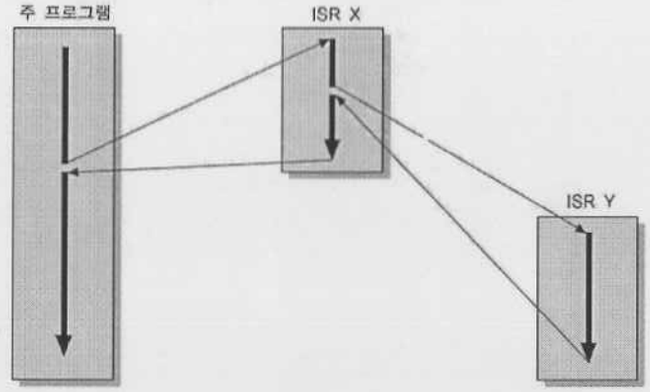
(a) LOAD 명령어의 실행 사이클이 종료된 상태 (b) 인터럽트 사이클이 종료된 상태

다중 인터럽트 (multiple interrupt)

- 인터럽트 서비스 루틴을 수행하는 동안에 다른 인터럽트가 발생하는 것
- 다중 인터럽트의 처리방법
  - CPU가 인터럽트 서비스 루틴을 처리하고 있는 도중에는 새로운 인터럽트 요구가 들어오더라도 CPU가 인터럽트 사이클을 수행하지 않도록 방지
    - 인터럽트 플래그(interrupt flag) = 인터럽트 불가능(interrupt disabled)
    - 시스템 운영상 중요한 프로그램이나 도중에 중단할 수 없는 데이터 입출력 동작 등을 위한 인터럽트를 처리하는데 사용
  - 인터럽트의 우선 순위를 정하고, 우선 순위가 낮은 인터럽트가 처리되고 있는 동안에 우선순위가 더 높은 인터럽트가 들어오면 현재의 인터럽트 서비스 루틴의 수행을 중단하고 새로운 인터럽트를 처리

다중 인터럽트 처리

- 장치 X를 위한 ISR X를 처리하는 도중에 우선 순위가 더 높은 장치 Y로부터 인터럽트 요구가 들어와서 먼저 처리되는 경우에 대한 제어의 흐름



2.2.4 간접 사이클(indirect cycle)

- 명령어에 포함되어 있는 주소를 이용하여, 실제 명령어 실행에 필요한 데이터를 인출하는 사이클
  → 간접 주소지정 방식(indirect addressing mode)에서 사용
- 인출 사이클과 실행 사이클 사이에 위치
- 간접 사이클에서 수행될 마이크로-연산

  t0 : MAR ← IR(addr)

  t1 : MBR ← M[MAR]

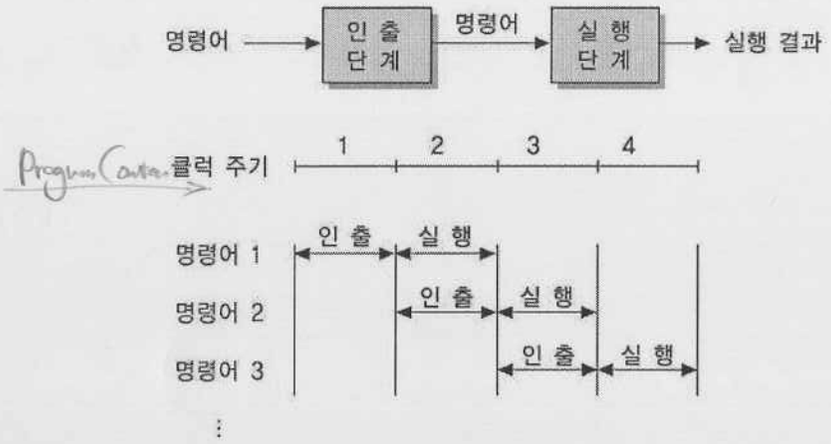
  t2 : IR(addr) ← MBR

  -. 인출된 명령어의 주소 필드 내용을 이용하여 기억장치로부터 데이터의 실제 주소를 인출하여 IR의 주소 필드에 저장

2.3 명령어 파이프라이닝(instruction pipelining)

- CPU의 프로그램 처리 속도를 높이기 위하여 CPU 내부 하드웨어를 여러 단계로 나누어 동시에 처리하는 기술
- 2-단계 명령어 파이프라인(two-stage instruction pipeline)
  -. 명령어를 실행하는 하드웨어를 인출 단계(fetch stage)와 실행 단계(execute stage)라는 두 개의 독립적인 파이프라인 모듈들로 분리
  -. 두 단계들에 동일한 클럭을 가하여 동작 시간을 일치
    - 첫 번째 클럭 주기에서는 인출 단계가 첫 번째 명령어를 인출
    - 두 번째 클럭 주기에서는 인출된 첫 번째 명령어가 실행 단계로 보내져서 실행되며, 그와 동시에 인출 단계는 두 번째 명령어를 인출

2-단계 명령어 파이프라인과 시간 흐름도



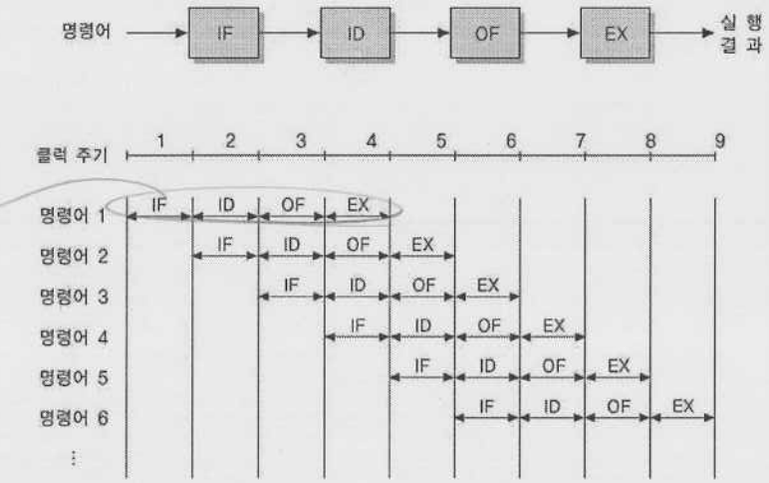
2-단계 명령어 파이프라인

- 2-단계 파이프라인을 이용하면 명령어 처리 속도가 두 배 향상
  (일반적으로 단계 수만큼의 속도 향상)
- 문제점
  -. 두 단계의 처리 시간이 동일하지 않으면 두 배의 속도 향상을 얻지 못함
  (파이프라인 효율 저하)
  ➔ 파이프라인 단계의 수를 증가시켜 각 단계의 처리 시간을 같게 함
  파이프라인 단계의 수를 늘리면 전체적으로 속도 향상도 더 높아짐

4-단계 파이프라인의 예

- 명령어 인출(IF) 단계 : 다음 명령어를 기억장치로부터 인출
- 명령어 해독(ID) 단계 : 해독기(decoder)를 이용하여 명령어를 해석
- 오퍼랜드 인출(OF) 단계 : 기억장치로부터 오퍼랜드를 인출
- 실행(EX) 단계 : 지정된 연산을 수행

4-단계 명령어 파이프라인과 시간 흐름도



파이프라인에 의한 전체 명령어 실행 시간

파이프라인 단계 수 = k,

실행할 명령어들의 수 = N,

각 파이프라인 단계가 한 클럭 주기씩 걸린다고 가정한다면,

파이프라인에 의한 전체 명령어 실행 시간 T :

$$T = k + (N - 1)$$

즉, 첫 번째 명령어를 실행하는데 k 주기가 걸리고,

나머지 (N - 1) 개의 명령어들은 각각 한 주기씩만 소요

파이프라인 되지 않은 경우의 N 개의 명령어들을 실행 시간 T :

$$T = k \times N$$

파이프라인에 의한 속도 향상 (speedup)

$$S_p = \frac{T_1}{T_k} = \frac{k \times N}{k \times (N-1)}$$

[예] k=4일 때, N = 100 이라면, Sp = 400/103 = 3.88

N = 1000 이라면, Sp = 4000/1003 = 3.99

N = 10000 이라면, Sp = 40000/10003 = 3.998

N → ∞ 이라면, Sp = 4

파이프라인에 의한 속도 향상 예

- 파이프라인 단계 수 = 4이고,

  파이프라인 클럭 = 1MHz (각 단계에서의 소요시간 = 1 ㎲)라면,

  첫번째 명령어 실행에 걸리는 시간 = 4 ㎲

  다음부터는 매 1 ㎲ 마다 한 개씩의 명령어 실행 완료,

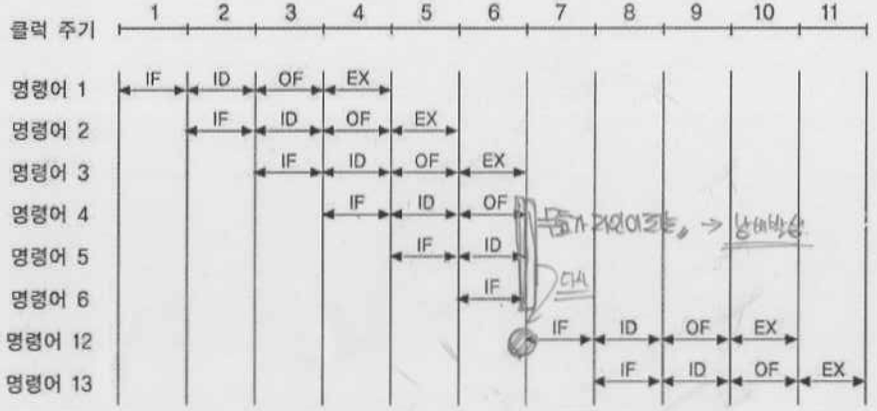
  10개의 명령어 실행 시간 = 4 + (10 - 1) = 13 ㎲

  속도향상 = (10 × 4) / 13 ≒ 3.08 배

파이프라인의 성능 저하 요인들

- 모든 명령어들이 파이프라인 단계들을 모두 거치지는 않는다
  - -. 어떤 명령어에서는 오퍼랜드를 인출할 필요가 없다
  - -. 그러나 파이프라인 하드웨어를 단순화하기 위해서는 모든 명령어가 네 단계들을 모두 통과하도록 해야 한다
- 파이프라인의 클럭은 처리 시간이 가장 오래 걸리는 단계가 기준이 된다
- IF 단계와 OF 단계가 동시에 기억장치를 액세스하는 경우에 기억장치 충돌(memory conflict)이 일어나면 지연이 발생한다
- 조건 분기(conditional branch) 명령어가 실행되면, 미리 인출하여 처리하던 명령어들이 무효화된다

조건 분기가 존재하는 경우의 시간 흐름도



분기 발생에 의한 성능 저하의 최소화 방법

- 분기 목적지 선인출(prefetch branch target)
  - -. 조건 분기가 인식되면, 분기 명령어의 다음 명령어뿐만 아니라 분기의 목적지 명령어도 함께 인출하는 방법
- 루프 버퍼(loop buffer) 사용
  - -. 파이프라인의 명령어 인출 단계에 포함되어 있는 작은 고속 기억장치인 루프 버퍼에 가장 최근 인출된 n개의 명령어들을 순서대로 저장해두는 방법
- 분기 예측(branch prediction)
  - -. 분기가 일어날 것인 지를 예측하고, 그에 따라 명령어를 인출하는 확률적 방법
  - -. 분기 역사 표(branch history table) 이용하여 최근의 분기 결과를 참조
- 지연 분기(delayed branch)
  - -. 분기 명령어의 위치를 재배치함으로써 파이프라인의 성능을 개선하는 방법

상태 레지스터(status register)

- 조건분기 명령어가 사용할 조건 플래그(condition flag)들 저장
- 부호(S)플래그
  - -. 직전에 수행된 산술연산 결과값의 부호비트를 저장
- 영(Z) 플래그
  - -. 연산 결과값이 0 이면, 1
- 올림수(C) 플래그
  - -. 덧셈이나 뺄셈에서 올림수(carry)나 빌림수(borrow)가 발생한 경우에 1로 세트

S	Z	C	X	E	V	I	P

상태 레지스터 (계속)

- 동등(E) 플래그
  - -. 두 수를 비교한 결과가 같게 나왔을 경우에 1로 세트
- 오버플로우(V) 플래그
  - -. 산술 연산 과정에서 오버플로우가 발생한 경우에 1로 세트
- 인터럽트(I) 플래그
  - -. 인터럽트 가능(interrupt enabled) 상태이면 0로 세트
  - -. 인터럽트 불가능(interrupt disabled) 상태이면 1로 세트
- 슈퍼바이저(P) 플래그
  - -. CPU의 실행 모드가 슈퍼바이저 모드(supervisor mode)이면 1로 세트,
  - -. 사용자 모드(user mode)이면 0로 세트

2.4 명령어 세트(instruction set)

- 어떤 CPU를 위하여 정의되어 있는 명령어들의 집합
- 명령어 세트 설계를 위해 결정되어야 할 사항들
  - -. 연산 종류 : CPU가 수행할 연산들의 수와 종류 및 복잡도
  - -. 데이터 형태 : 연산을 수행할 데이터들의 형태, 데이터의 길이(비트 수), 수의 표현 방식 등
  - -. 명령어 형식 : 명령어의 길이, 오퍼랜드 필드들의 수와 길이, 등
  - -. 주소지정 방식 : 오퍼랜드의 주소를 지정하는 방식

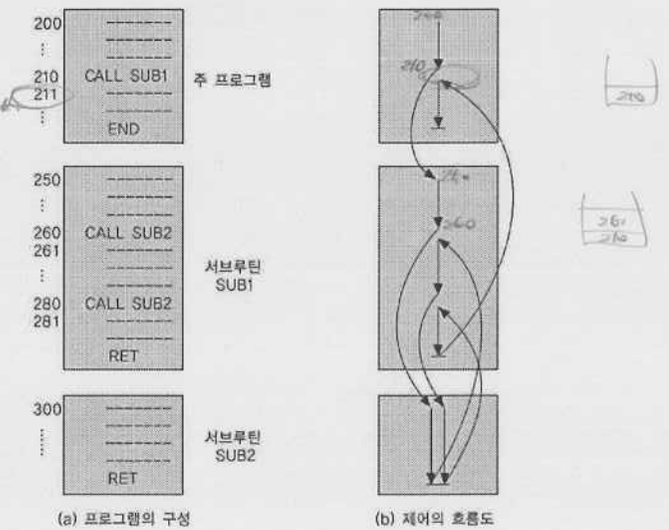
2.4.1 연산의 종류

- 데이터 전송 : 레지스터와 레지스터 간, 레지스터와 기억장치 간, 혹은 기억장치와 기억장치 간에 데이터를 이동하는 동작
- 산술 연산 : 덧셈, 뺄셈, 곱셈 및 나눗셈과 같은 기본적인 산술 연산들
- 논리 연산 : 데이터의 각 비트들 간에 대한 AND, OR, NOT 및 exclusive-OR 연산
- 입출력(I/O) : CPU와 외부 장치들 간의 데이터 이동을 위한 동작들
- 프로그램 제어
  - -. 명령어 실행 순서를 변경하는 연산들
  - -. 분기(branch), 서브루틴 호출(subroutine call)

서브루틴 호출을 위한 명령어들

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

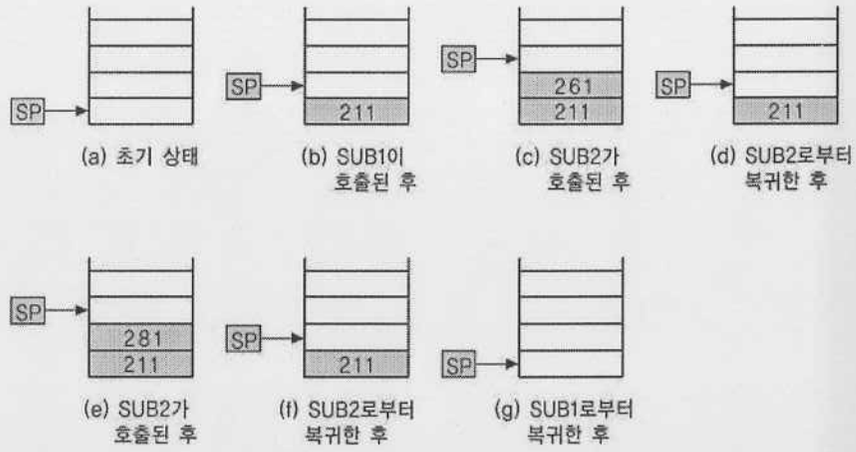
서브루틴들이 포함된 프로그램의 실행 과정



CALL/RET 명령어의 마이크로 연산

- CALL X 명령어에 대한 마이크로-연산:
  - t0 : MBR ← PC
  - t1 : MAR ← SP, PC ← X
  - t2 : M[MAR] ← MBR, SP ← SP - 1
  - -. 현재의 PC 내용(서브루틴 수행 완료 후에 복귀할 주소)을 SP가 지정하는 스택의 최상위(top of stack)에 저장
  - -. 만약 주소지정 단위가 바이트이고 저장될 주소는 16비트라면, SP ← SP - 2
- RET 명령어의 마이크로-연산
  - t0 : SP ← SP + 1
  - t1 : MAR ← SP
  - t2 : PC ← M[MAR]

서브루틴 수행 과정에서 스택의 변화



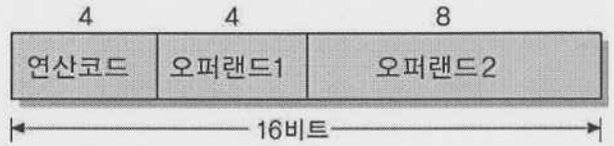
2.4.2 명령어 형식

명령어의 구성요소들

- 연산 코드(Operation Code)
  - -. 수행될 연산을 지정(예: LOAD, ADD 등)
- 오퍼랜드(Operand)
  - -. 연산을 수행하는 데 필요한 데이터 혹은 데이터의 주소
  - -. 각 연산은 한개 혹은 두개의 입력 오퍼랜드들과 한개의 결과 오퍼랜드를 포함
  - -. 데이터는 CPU 레지스터, 주기억장치, 혹은 I/O 장치에 위치
- 다음 명령어 주소(Next Instruction Address)
  - -. 현재의 명령어 실행이 완료된 후에 다음 명령어를 인출할 위치 지정
  - -. 분기 혹은 호출 명령어와 같이 실행 순서를 변경하는 경우에 필요

명령어 형식 (instruction format)

- 필드(field) : 명령어의 각 구성 요소들에 소요되는 비트들의 그룹
- 명령어 형식(instruction format) : 명령어 내 필드들의 수와 배치 방식 및 각 필드의 비트 수
- 명령어의 길이 = 단어 길이
- 명령어 형식의 예 : 세 개의 필드들로 구성된 16-비트 명령어



명령어 형식의 결정에서 고려할 사항들

- 연산 코드 필드 = 4 비트 → 2^4 = 16 가지의 연산들 정의 가능
  - -. 만약 연산 코드 필드가 5 비트로 늘어나면, 2^5 = 32 가지 연산들 정의 가능
    → 다른 필드의 길이가 감소
- 오퍼랜드 필드의 범위는 오퍼랜드의 종류에 따라 결정
  - -. 데이터 : 표현 가능한 수의 크기가 결정
  - -. 기억장치 주소 : CPU가 오퍼랜드 인출을 위하여 직접 주소를 지정할 수 있는 기억장치 영역의 범위가 결정
  - -. 레지스터 번호 : 데이터 저장에 사용될 수 있는 레지스터의 수가 결정

오퍼랜드 필드의 범위 예

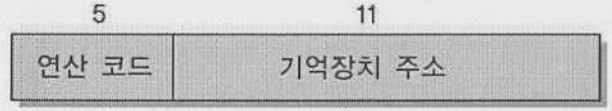
- 오퍼랜드1은 레지스터 번호를 지정하고, 오퍼랜드2는 기억장치 주소를 지정하는 경우
  - -. 오퍼랜드1 = 4 비트 → 16 개의 레지스터 사용 가능
  - -. 오퍼랜드2 = 8 비트 → 기억장치의 주소 범위 : 0 번지 ~ 255 번지
- 위에서 두 오퍼랜드들을 하나로 통합하여 사용하는 경우
  - -. 오퍼랜드가 2의 보수로 표현되는 데이터라면,
    표현 범위 : − 2048 ~ + 2047
  - -. 오퍼랜드가 기억장치 주소라면,
    2^{12} = 4096 개의 기억장치 주소 지정 가능

오퍼랜드의 수에 따른 명령어 분류

- 1-주소 명령어(1-address instruction) : 오퍼랜드를 한 개만 포함하는 명령어
  [예] ADD X ; AC ← AC + M[X]
- 2-주소 명령어(two-address instruction) : 두 개의 오퍼랜드를 포함하는 명령어.
  [예] ADD R1, R2 ; R1 ← R1 + R2
  MOV R1, R2 ; R1 ← R2
  ADD R1, X ; R1 ← R1 + M[X]
- 3-주소 명령어(three-address instruction) : 세 개의 오퍼랜드들을 포함하는 명령어.
  [예] ADD R1, R2, R3 ; R1 ← R2 + R3

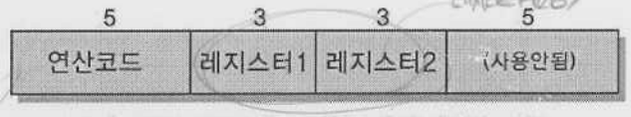
1-주소 명령어의 예

- 길이가 16 비트인 1-주소 명령어에서 연산 코드가 5 비트일 때의 명령어 형식을 정의하고, 주소지정 가능한 기억장치 용량을 결정하라
  - -. 주소지정 가능한 기억장치 용량 : 2^{11} = 2048 바이트
  - -. 명령어 형식

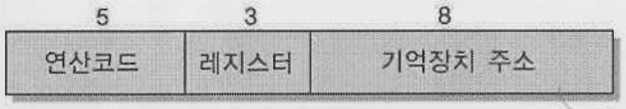


2-주소 명령어의 예

- 2-주소 명령어 형식을 사용하는 16-비트 CPU에서 연산 코드가 4 비트이고, 레지스터의 수는 16 개이다. (a) 두 오퍼랜드들이 모두 레지스터 번호인 경우와, (b) 한 오퍼랜드는 기억장치 주소인 경우의 명령어 형식을 정의하라

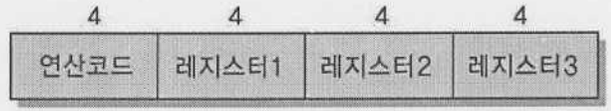


(a) 두 개의 레지스터 오퍼랜드들을 가지는 경우

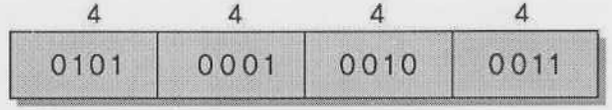


(b) 한 오퍼랜드는 기억장치 주소인 경우

3-주소 명령어 형식의 예



(a) 명령어 형식



(b) ADD R1,R2,R3 명령어의 비트 배열

명령어 형식이 프로그래밍에 미치는 영향 (예)

- X = (A + B) x (C - D)
- 프로그래밍에 다음과 같은 니모닉을 가진 명령어들을 사용
  - ADD : 덧셈
  - SUB : 뺄셈
  - MUL : 곱셈
  - DIV : 나눗셈
  - MOV : 데이터 이동
  - LOAD : 기억장치로부터 데이터 적재
  - STOR : 기억장치로 데이터 저장

1-주소 명령어를 사용한 프로그램

```
LOAD  A     ; AC ← M[A]
ADD   B     ; AC ← AC + M[B]
STOR  T     ; M[T] ← AC
LOAD  C     ; AC ← M[C]
SUB   D     ; AC ← AC - M[D]
MUL   T     ; AC ← AC × M[T]
STOR  X     ; M[X] ← AC
```

-. M[A]는 기억장치 A 번지의 내용, T는 기억장치 내의 임시 저장 장소의 주소

-. 프로그램의 길이 = 7

2-주소 명령어를 사용한 프로그램

```
MOV     R1, A     ; R1 ← M[A]
ADD     R1, B     ; R1 ← R1 + M[B]
MOV     R2, C     ; R2 ← M[C]
SUB     R2, D     ; R2 ← R2 - M[D]
MUL     R1, R2    ; R1 ← R1 × R2
MOV     X, R1     ; M[X] ← R1
```

-. 프로그램의 길이 = 6

3-주소 명령어를 사용한 프로그램

```
ADD   R1, A, B     ; R1 ← M[A] + M[B]
SUB   R2, C, D     ; R2 ← M[C] - M[D]
MUL   X, R1, R2    ; M[X] ← R1 × R2
```

- 프로그램의 길이 = 3
- 단점
  - -. 명령어의 길이 증가한다
  - -. 명령어 해독 과정이 복잡해진다

주소지정 방식

- 다양한 주소지정 방식(addressing mode)을 사용하는 이유 : 제한된 수의 명령어 비트들을 이용하여 사용자(혹은 프로그래머)로 하여금 여러 가지 방법으로 오퍼랜드를 지정하고 더 큰 용량의 기억장치를 사용할 수 있도록 하기 위함
- 기호
  - -. EA : 유효 주소(Effective Address), 즉 데이터가 저장된 기억장치의 실제 주소
  - -. A : 명령어 내의 주소 필드 내용 (오퍼랜드 필드가 기억장치 주소를 나타내는 경우)
  - -. R : 명령어 내의 레지스터 번호 (오퍼랜드 필드가 레지스터 번호를 나타내는 경우)
  - -. (A) : 기억장치 A 번지의 내용
  - -. (R) : 레지스터 R의 내용

주소지정 방식의 종류

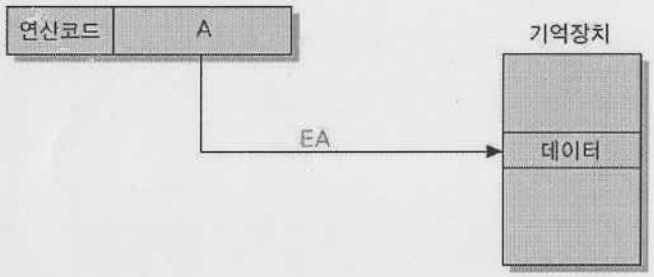
- 직접 주소지정 방식 (direct addressing mode)
- 간접 주소지정 방식 (indirect addressing mode)
- 묵시적 주소지정 방식 (implied addressing mode)
- 즉치 주소지정 방식 (immediate addressing mode)
- 레지스터 주소지정 방식 (register addressing mode)
- 레지스터 간접 주소지정 방식 (register-indirect addressing mode)
- 변위 주소지정 방식 (displacement addressing mode)
  - -. 상대 주소지정 방식(relative addressing mode)
  - -. 인덱스 주소지정 방식(indexed addressing mode)
  - -. 베이스-레지스터 주소지정 방식(base-register addressing mode)

직접 주소지정 방식

- 오퍼랜드 필드의 내용이 유효 주소가 되는 방식

  EA = A

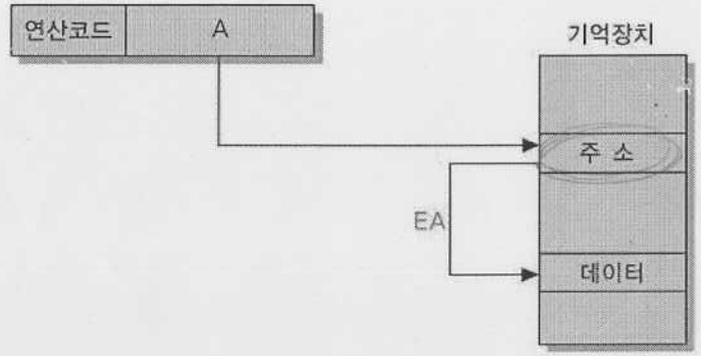
- [장점] 데이터 인출을 위하여 한 번의 기억장치 액세스만 필요
- [단점] 연산 코드를 제외하고 남은 비트들만 주소 비트로 사용될 수 있기 때문에 직접 지정할 수 있는 기억장소의 수가 제한



간접 주소지정 방식

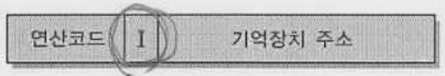
- 오퍼랜드 필드에 기억장치 주소가 저장되어 있지만, 그 주소가 가리키는 기억 장소에 데이터의 유효 주소가 저장되어 있도록 하는 방식

  EA = (A)



간접 주소지정 방식

- [장점] 최대 기억장치용량이 단어의 길이에 의하여 결정 → 확장가능
  - -. 단어 길이가 n 비트라면, 최대 2^n 개의 기억 장소들을 주소지정 가능
- [단점] 실행 사이클 동안에 두 번의 기억장치 액세스가 필요
  - -. 첫 번째 액세스는 주소를 읽어 오기 위한 것
  - -. 두 번째는 그 주소가 지정하는 위치로부터 실제 데이터를 인출하기 위한 것
- 명령어 형식에서 간접비트(I) 필요



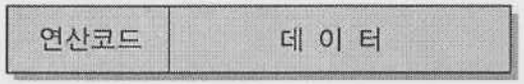
  - -. 만약 I = 0 이면, 직접 주소지정 방식
  - -. 만약 I = 1 이면, 간접 주소지정 방식
- 다단계(multi-level) 간접 주소지정 방식
  - -. EA = ((. . (A) . .))

묵시적 주소지정 방식

- 명령어 실행에 필요한 데이터의 위치가 묵시적으로 지정되는 방식
  - -. 'SHL 명령어' : 누산기의 내용을 좌측으로 쉬프트(shift)
  - -. 'PUSH R1 명령어' : 레지스터 R1의 내용을 스택에 저장
- [장점] 명령어 길이가 짧다
- [단점] 종류가 제한된다

즉치 주소지정 방식

- 데이터가 명령어에 포함되어 있는 방식
  - -. 오퍼랜드 필드의 내용이 연산에 사용할 실제 데이터
- 용도
  - -. 프로그램에서 레지스터들이나 변수의 초기 값을 어떤 상수값 (constant value)으로 세트하는 데 유용하게 사용
- [장점] 데이터를 인출하기 위하여 기억장치를 액세스할 필요가 없음
- [단점] 상수값의 크기가 오퍼랜드 필드의 비트 수에 의하여 제한

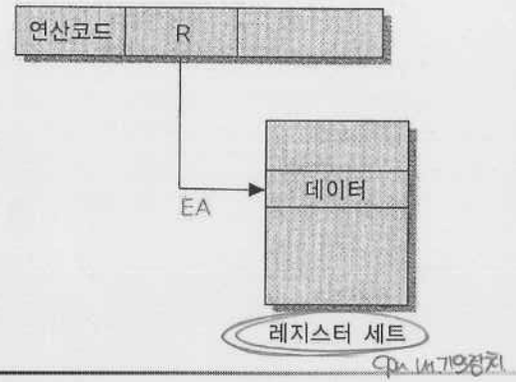


레지스터 주소지정 방식

- 연산에 사용할 데이터가 레지스터에 저장되어 있는 방식

  EA = R
- 주소지정에 사용될 수 있는 레지스터들의 수 = 2^k 개

  (k는 오퍼랜드 비트 수)



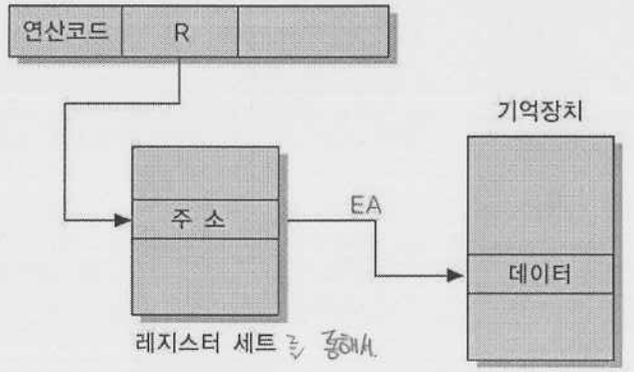
레지스터 주소지정 방식

- [장점]
  - -. 오퍼랜드 필드의 비트 수가 적어도 된다
  - -. 데이터 인출을 위하여 기억장치 액세스가 필요 없다
- [단점]
  - -. 데이터가 저장될 수 있는 공간이 CPU 내부 레지스터들로 제한

레지스터 간접 주소지정 방식

- 오퍼랜드 필드(레지스터 번호)가 가리키는 레지스터의 내용을 유효 주소로 사용하여 실제 데이터를 인출하는 방식

  EA = (R)



레지스터 간접 주소지정 방식

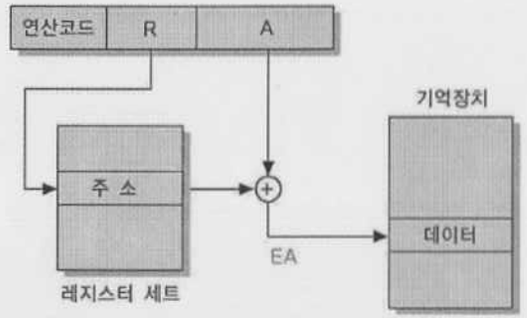
- [장점] 주소지정 할 수 있는 기억장치 영역이 확장
  - -. 레지스터의 길이 = 16 비트라면, 주소지정 영역: 2^{16} = 64K 바이트
  - -. 레지스터의 길이 = 32 비트라면, 주소지정 영역: 2^{32} = 4G 바이트

변위 주소지정 방식

- 직접 주소지정과 레지스터 간접 주소지정 방식의 조합

  EA = A + (R)

  -. 명령어에 포함된 변위 A값과 R이 가리키는 레지스터의 내용을 더하여 유효 주소를 결정



변위 주소지정 방식

- 사용되는 레지스터에 따라 여러 종류의 변위 주소지정 방식 정의
  - -. PC → 상대 주소지정 방식(relative addressing mode)
  - -. 인덱스 레지스터 → 인덱스 주소지정 방식(indexed addressing mode)
  - -. 베이스 레지스터 → 베이스-레지스터 주소지정 방식(base-register addressing mode)

상대 주소지정 방식

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

  EA = A + (PC)

  A는 2의 보수

  -. A ≥ 0 : 앞(forward) 방향으로 분기

  -. A < 0 : 뒷(backward) 방향으로 분기
- [예] JUMP 명령어가 450 번지에 저장. 명령어 인출 후, PC 내용 = 451
  - -. 만약 오퍼랜드 A = + 21 → 분기 목적지 주소 = 451 + 21 = 472 번지
  - -. 만약 오퍼랜드 A = - 50 → 분기 목적지 주소 = 451 - 50 = 401 번지
- [장점] 전체 기억장치 주소가 명령어에 포함되어야 하는 일반적인 분기 명령어보다 적은 수의 비트만 있으면 된다
- [단점] 분기 범위가 오퍼랜드 필드의 길이에 의하여 제한

인덱스 주소지정 방식

- 인덱스 레지스터의 내용과 변위 A를 더하여 유효 주소를 결정

  EA = (IX) + A

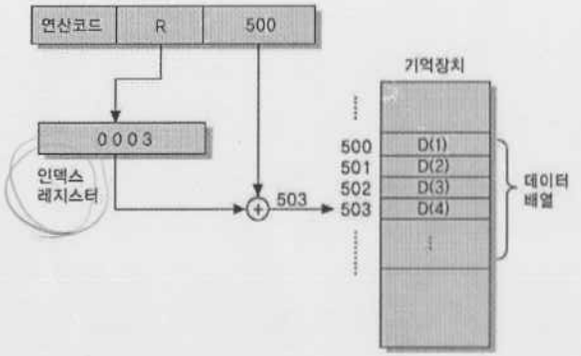
  -. 인덱스 레지스터(IX) : 인덱스(index) 값을 저장하는 특수 레지스터
- [주요 용도] 배열 데이터 액세스
- 자동 인덱싱(autoindexing)
  -. 명령어가 실행될 때마다 인덱스 레지스터의 내용이 자동적으로 증가 혹은 감소
  -. 이 방식이 사용된 명령어가 실행되면 아래의 두 연산이 연속적으로 수행

  EA = (IX) + A

  IX ← IX + 1

인덱스 주소지정 방식의 예

- 데이터 배열이 기억장치의 500 번지부터 저장되어 있고, 명령어의 주소 필드에 500이 포함되어 있을 때, 인덱스 레지스터의 내용 (IX) = 3 이라면 → 데이터 배열의 4 번째 데이터 액세스



베이스-레지스터 주소지정 방식

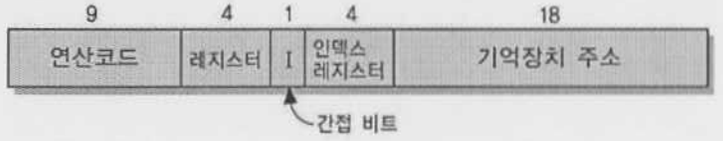
- 베이스 레지스터의 내용과 변위 A를 더하여 유효 주소를 결정

  EA = (BR) + A

- [주요 용도] 서로 다른 세그먼트내 프로그램의 위치를 지정

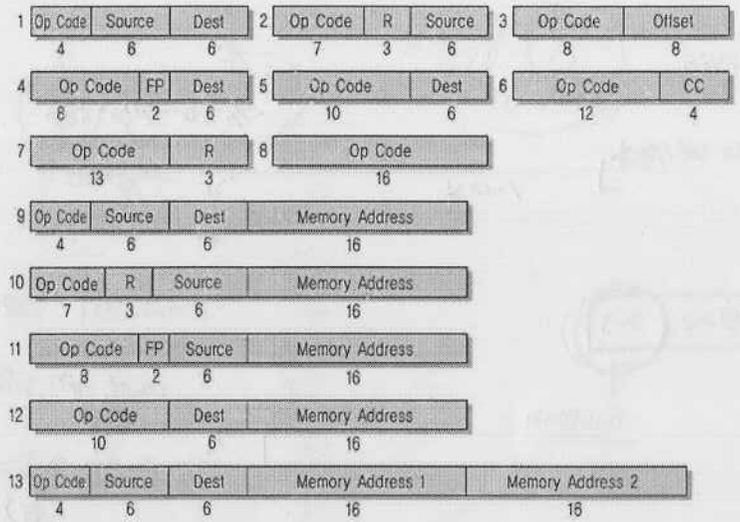
PDP 계열 프로세서의 명령어 형식

- PDP-10 프로세서 : 고정 길이의 명령어 형식 사용
  -. 단어의 길이 = 36 비트, 명령어의 길이 = 36 비트
  -. 연산 코드 = 9 비트 → 최대 512 종류의 연산 허용 (실제 365 개)



- PDP-11 프로세서 : 다양한 길이의 명령어 형식들 사용
  -. 연산 코드 = 4 ~ 16 비트
  -. 주소 개수 : 0, 1, 2 개

PDP-11의 명령어 형식



펜티엄 프로세서의 명령어 형식

- 선형 주소(linear address: LA)
  - 유효 주소 + 세그먼트의 시작 주소
    - 세그먼트의 시작 주소는 세그먼트 레지스터에 저장

주소지정 방식	유효 주소(EA)	선형 주소(LA)
즉치 방식	데이터 = A	
레지스터 방식	EA = R	LA = R
변위 방식	EA = A	LA = (SR)+EA
베이스 방식	EA = (BR)	LA = (SR)+EA
변위를 가진 베이스 방식	EA = (BR)+A	LA = (SR)+EA
변위를 가진 인덱스 방식	EA = (IX)+A	LA = (SR)+EA
인덱스와 변위를 가진 베이스 방식	EA = (IX)+(BR)+A	LA = (SR)+EA
상대 방식	EA = (PC)+A	LA = EA

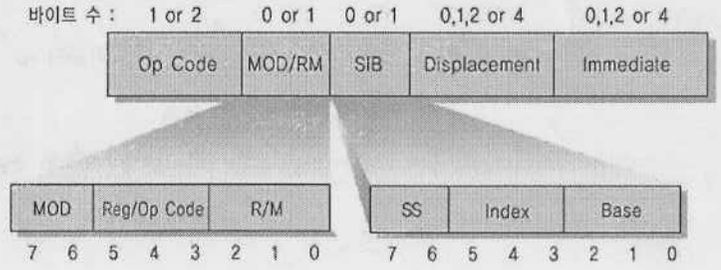
펜티엄 프로세서의 명령어 형식

- 즉치 방식(immediate mode) : 데이터가 명령어에 포함되어 있는 방식

  데이터의 길이 = 바이트, 단어(word) 혹은 2중 단어(double word)
- 레지스터 방식(register mode) : 유효 주소가 레지스터에 들어 있는 방식
- 변위 방식(displacement mode) : 명령어에 포함된 변위값과 세그먼트 레지스터 SR의 내용을 더하여 선형주소 LA를 생성하는 방식
- 베이스 방식(base mode) : 레지스터 간접 주소지정에 해당
- 상대 방식(relative mode) : 변위값과 프로그램 카운터의 값을 더하여 다음 명령어의 주소로 사용하는 방식

펜티엄 명령어 형식의 필드들

- 연산 코드(Op code) : 연산의 종류 지정. 길이 = 1 혹은 2 바이트
- MOD/RM : 주소지정 방식 지정
- SIB : MOD/RM 필드와 결합하여 주소지정 방식을 완성
- 변위(displacement) : 부호화된 정수(변위)를 저장
- 즉치(immediate) : 즉치 데이터를 저장



기억공간과 연관시키는 방법

절대 주소(absolute address)
절대 주소는 기억 장치의 최대 크기가 2m일 때 주소 bus의 개수는 m(bit)개이고 m개를 모두 사용하여 주소를 나타낸다. 즉, 기억 장치 cell의 절대 위치(주소)를 나타낸다. instruction에서 주소를 표시하기 위하여 많은 bit를 사용해야 하므로 메모리를 많이 차지하고 프로그램은 절대 주소에서만 수행되고 재배치(relocation)가 안된다. 따라서 특별한 경우가 아니면 사용하지 않는다.

상대 주소(relative address)
상대 주소는 기억 장치의 최대 크기가 2m일 때 m개의 bit를 모두 사용하지 않고 주소를 나타내는 방식으로 절대 주소보다 명령어의 길이가 짧아진다.
즉, {실제주소=기본주소+상대거리}의 형태이므로 주소 계산을 해야 하는 부담이 따르지만 재배치가 가능하고 명령어의 길이가 짧아 메모리가 절약되고 또한 프로그램의 수행이 다른 컴퓨터에서도 가능해 진다는 이점이 있으므로 실제로 많이 사용하고 있다. 이때 주소 계산은 ALU가 할 수도 있고 주소 계산만 전담하는 장치를 별도로 두어 할 수도 있다.
기본 주소는 레지스터에 기억시키거나 symbol(기호, label, 고급언어의 경우 변수)을 이용하여 표시하고 그 값은 운영 체제가 정해주며 상대 거리는 기본 주소에서 얼마만큼 떨어져 있나를 표시하는 값으로 프로그래머가 정해준다.

계산에 의한 주소
계산에 의한 주소 또는 인덱스된 주소도 상대 주소인데 이는 일반적인 상대 주소와는 달리 실제 주소 계산에 인덱스가 쓰인다는 점이다.
즉 {유효주소=기본 주소+상대거리+인덱스}로 주소 계산을 하는 경우이다. 이때 인덱스 값이 기억된 레지스터를 index register라 한다.

자기 정의 상대 주소
자기정의 상대 주소(self define relative address)는 기본 주소가 아닌 PC 값에서 얼마만큼 떨어져 있나를 표시하는 방법, 즉 현재 명령어에서 떨어져 있는 정도를 표시하는 방법이다. Program Counter에는 다음에 수행할 명령어 주소가 기억되어 있다.

접근 방법
직접 주소
직접 주소(direct address)는 자료가 기억된 장소에 직접 사상(mapping)시킬 수 있는 주소이다. 기억 장치의 주소를 직접(절대주소이든, 상대주소이든) 가리키는 것으로 간결하나 융통성이 없다. 하나의 자료를 가져오기 위해서 기억장치에 1번 접근해야 한다.

간접 주소
간접 주소(indirect address)는 주소 부분의 내용이 직접 자료를 가리키는 것이 아니고 자료가 있는 주소를 가리키는 것으로 융통성은 많으나 하나의 자료를 가져오기 위해 기억 장치에 최소한 두 번 접근해야 하므로 시간이 많이 걸린다. 하지만 짧은 길이로 큰 용량의 기억장치 주소를 나타내는데 적합하다.
또한 bit-flag(mode field)를 두어 직접인지 간접인지 구별하여 쓸 수도 있다.

자료 자신
자료 자신(immediate address)은 operand 자리에 주소를 적는 것이 아니고 자료를 바로 적는 경우로서 다른 어떤 방식보다 속도가 빠르나 자료의 길이에 한계가 있다. 자료 가져오기 위한 메모리 접근은 필요없다.
immediate: 자료 자체, 자료 자신, 즉치, 즉각 등의 용어로 번역된다.

레지스터 지정
자료를 특정 레지스터 내에 기억시켜 놓고 주소대신 그 레지스터의 번호를 적어주는 방식이다. 자료를 가져오기 위한 접근은 할 필요가 없다

기억장치 주소지정 방식(Pattern of Addressing)

오퍼랜드의 주소가 지정되는 방법

▷ CPU가 오퍼랜드에 접근하는 속도에 영향을 미친다.

▷ 오퍼랜드가 위치한 주소를 변경하는 것의 용이함에 영향을 준다.

(용어 설명)

EA: 유효주소(Effective Address) : 명령어 수행에 필요한 데이터가 저장되어 있는 기억장치의 실재 주소를 말한다.

A : 명령어 내의 오퍼랜드 필드가 기억장치를 나타내는 경우 주소 필드의 내용

R : 명령어 내의 오퍼랜드 필드가 레지스터를 나타내는 경우에 명령어 내의 레지스터 번호

(A) : 기억장치 A에 저장되어 있는 데이터를 나타낸다.

(R) : 레지스터 R에 저장되어 있는 데이터를 나타낸다.

(1) 직접 주소지정 방식(Direct Addressing Mode)

명령어의 오퍼랜드 필드의 내용이 데이터의 유효주소이다.

유효주소 = A

◇장점: 단 한 번만 필요하므로 간단하고 데이터를 CPU로 빠르게 가지고 온다.

◇단점: 명령어 형식에서 정해진 오퍼랜드의 비트만을 사용하므로 메모리의 사용이 극히 제한적이다.

예) 16 비트의 명령어 중에서 연산코드 길이가 4 비트인 경우, 오퍼랜드 필드의 길이는 16-4=12 비트이므로, 직접 주소지정 방식을 통하여 지정할 수 있는 메모리의 크기는 212 = 4096 개 이다.

(2) 간접 주소지정 방식(Indirect Addressing Mode)

간접 주소지정 방식에서 유효주소

EA = (A)

오퍼랜드의 값 A가 유효 주소가 아니라, A에 저장되어 있는 값이 유효주소임

◇ 장점 : 액세스되어지는 메모리의 범위가 명령어 형식의 오퍼랜드 필드의 길이에 제한을 받지 않고 확장할 수 있다.

◇ 단점 : 명령어 연산에 필요한 데이터를 얻는데 두 번의 메모리 액세스가 필요하다. 따라서 메모리를 가져오는 지연이 증가한다.

(3) 즉시 주소지정방식(Immediate Addressing Mode)

- 즉치 주소지정 방식에서는 명령어에 데이터가 포함되어있다.

- 명령어의 오퍼랜드가 데이터의 기억장치 주소가 아니라, 명령어 수행에 필요한 데이터 그 자체이다.

- 프로그램의 수행에 있어서 레지스터 혹은 어떤 변수들의 초기 값을 정해 줄 경우에 주로 사용되어 진다.

EA = 현재 수행 중인 명령어가 저장되어 있는 기억장치 주소

(예제)

* PC = 1000

1000 LOAD 500

1001 ADD B

- 현재 수행 중인 명령어는 데이터 500을 누산기로 적재하라는 의미

- 따라서 유효주소는 1000 이다.

◇ 장점 : 연산에 필요한 데이터를 가져오는데 기억장치를 액세스 할 필요가 없으므로 인출 시간 이 절약된다.

◇ 단점 : 사용되어지는 데이터의 표현이 오퍼랜드 필드의 비트에 제한을 받는다. 즉, 명령어 전체에서 연산명령 부분을 뺀 나머지 필드로 데이터를 표현해야 하므로 표현되어지는 수의 범위에 제한을 받는다.

(4) 레지스터 주소지정 방식(Register Addressing Mode)

- 명령어 수행에 필요한 데이터가 명령어의 오퍼랜드에 나타나는 레지스터에 저장되어 있다.

- 유효주소를 연산에 필요한 데이터가 들어있는 기억장치(메모리) 주소라고 할 때, 레지스터 주소 지정 방식에서의 유효 주소는 레지스터이므로 없다.

- 단, 레지스터를 저장장소로 생각한다면 유효주소는 레지스터 R0이 된다.

◇ 장점 : 연산에 필요한 데이터가 CPU 내의 레지스터에 저장되어 있으므로 기억장치 액세스를 필요로 하는 다른 주소지정 방식에 비해 데이터 인출 시간이 적다. 뿐 만 아니라 오퍼랜드필드의 길이가 적다.

◇ 단점 : 데이터가 레지스터에 저장되어지기 때문에 데이터의 크기가 레지스터의 크기에 제한을 받는다.

(5) 레지스터 간접 주소지정 방식(Register-Indirect Addressing Mode)

- 레지스터에 저장된 데이터가 명령어 연산에 필요한 데이터가 아니라, 그 데이터가 저장되어 있는 기억장치의 주소이다.

- 유효 주소

EA = (R)

- 레지스터의 크기에 따라 지정되어지는 기억장치의 주소의 범위가 지정된다.

예) 사용되어지는 레지스터의 길이가 16비트라면, 2^{16} = 64 K 바이트 까지의 주소를 지정할 수 있다.

◇ 장점 : 사용되어지는 레지스터의 길이에 따라 데이터 저장 영역이 확장되어질 수 있다.

◇ 단점 : 레지스터의 크기에 따라 지정되어지는 기억장치의 주소의 범위가 제한되어 질 수 있고, 레지스터 지정 방식과는 달리 데이터 인출을 위하여 한 번의 기억장치 액세스가 필요하다.

(6) 상대 주소지정 방식(Relative Addressing Mode)

- 변위 주소지정 방식(Displacement Addressing Mode)

- 변위 주소지정 방식에서는 아래의 그림과 같이 두 개의 오퍼랜드를 사용하는데, 하나는 레지스 터를 나타내고, 다른 하나는 변위를 나타낸다.

- 상대 주소지정 방식은 PC를 사용하는 변위 주소지정 방식이다.

이 방식의 유효주소는 변위 D에 PC의 값을 더한 값이 된다. 즉,

〈유효주소 = D + (PC)〉

- 변위 값 D는 양수 혹은 음수 값을 가질 수 있다.

- 만약 D의 값이 양수이면 현재의 PC 값에서 D 값 만큼 앞으로 이동(Forwarding)한 주소가 바로 유효주소이고, 만약 D의 값이 음수이면 현재의 PC 값에서 D 값 만큼 뒤로 이동(Backwarding) 한 주소가 바로 유효 주소.

◇ 장점 : 사용되어지는 레지스터가 PC로 묵시적으로 정해지므로 명령어 형식에서 레지스터를 나타내는 필드가 필요하지 않다.

◇ 단점 : 변위의 길이(D) 혹은 범위가 오퍼랜드 필드의 비트 수에 제한을 받는다.

(7) 인덱스 주소지정 방식(Indexed Addressing Mode)

- 인덱스 레지스터를 사용하는 변위 주소지정 방식

- 인덱스 레지스터는 인덱스(Index) 값을 저장하는 특수 레지스터이다.

유효주소 = A + (IX)

- 일반적으로 인덱스 주소지정 방식은 데이터의 배열(Data Array) 연산에 유용

- 인덱스의 내용은 데이터 배열의 시작으로부터 배열 내의 데이터까지의 위치를 나타낸다.

(8) 베이스-레지스터 주소지정 방식(Base-Register Addressing Mode)

- 베이스 레지스터를 사용하는 변위 주소지정 방식이다.

- 베이스 레지스터 주소 방식에서는 기준이 되는 주소 A가 저장되어진다.

- 베이스 레지스터 주소지정 방식은 주로 기억장치 내의 프로그램의 위치를 지정하는데 사용된다.

- 베이스 주소지정 방식의 유효주소 유효주소 = A + (BR) ==========

주소지정방식이란 프로그램 수행 시 오퍼랜드를 지정하는 방식으로서 오퍼랜드를 실제 참조하기 전에 명령어의 주소 필드를 변경하거나 해석하는 규칙을 지정하는 형식

· 의미주소지정
명령어에서 주소필드를 필요로 하지 않는 방식
연산코드필드에 지정된 묵시적 의미의 오퍼랜드를 지정
[예]
ADD :
기억장치스택에서 ADD와 같은 명령어는 스택의 맨 위 항목과 그 아래 항목을 더하여 스택의 맨 위에 저장하는 명령어(오퍼랜드가 스택의 맨 위에 있다는 것을 묵시적으로 가정)

· 즉치주소지정
명령어 자체 내에 오퍼랜드를 지정하고 있는 방식
[예]
LDI ;
레지스터 R1에 데이터 100을 초기화 시키는 것으로서 명령어 LDI 자체 내에 100이라는 오퍼랜드를 포함

· 레지스터와 레지스터 간접주소지정
레지스터방식은 오퍼랜드가 레지스터에 저장되어 있음
레지스터 간접주소지정 방식은 레지스터가 실제 오퍼랜드가 저장된 기억장치의 주소값을 갖고있는 방식

· 직접주소지정과 간접주소지정
(1) 직접주소지정방식(direct-addressing mode)은 명령어의 주소필드에 직접 오퍼랜드의 주소를 저장
(2) 간접주소지정방식(indirect-addressing mode)은 명령어의 주소필드에 유효주소가 저장되어있는 기억장치주소를 기억시키는 주소방식. 제어는 기억장치로부터 명령어를 가져온 후 주소부분을 이용하여 다시 기억장치에 접근하여 유효주소를 읽어 냄

· 상대주소지정
유효주소를 계산하기 위해 처리장치 내에 있는 특정 레지스터의 내용에 명령어 주소필드 값을 더하는 방식
자주 사용되는 레지스터는 PC로서 상대주소지정방식에서 유효주소는 다음과 같이 계산됨
유효주소 = 명령어 주소부분의 내용 + PC의 내용

· 인덱스된 주소지정
인덱스된 주소지정방식에서는 인덱스 레지스터의 내용을 명령어 주소 부분에 더해서 유효주소를 얻음
인덱스 레지스터는 특정한 CPU레지스터나 레지스터 파일에 있는 레지스터가 될 수 있음

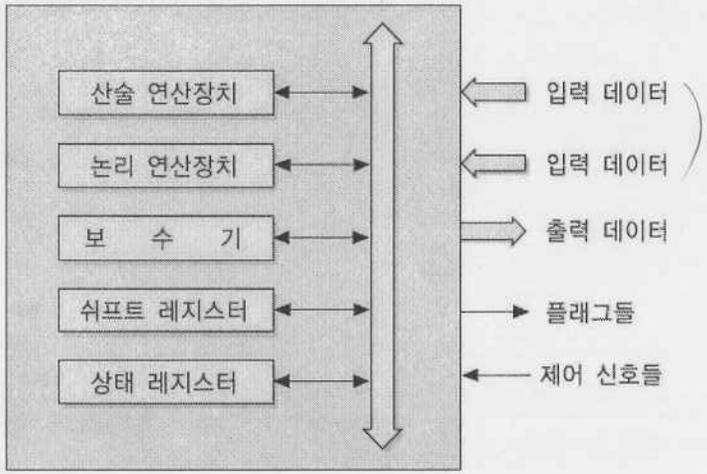
==========

제3장. 컴퓨터 산술과 논리 연산

3.1 ALU의 구성 요소

- 산술 연산장치 : 산술 연산들(+, −, ×, ÷)을 수행
- 논리 연산장치 : 논리 연산들(AND, OR, XOR, NOT 등)을 수행
- 쉬프트 레지스터(shift register) : 비트들을 좌측 혹은 우측으로 이동시키는 기능을 가진 레지스터
- 보수기(complementer) : 2진 데이터를 2의 보수로 변환(음수화)
- 상태 레지스터(status register) : 연산 결과의 상태를 나타내는 플래그(flag)들을 저장하는 레지스터

ALU의 내부 구성 요소들



3.2 정수의 표현

- 2진수 : 0, 1, 부호 및 소수점으로 표현

  $-13.625_{10} = -1101.101_2$

- 부호 없는 정수 표현의 예

  00111001 = 57
  00000000 = 0
  00000001 = 1
  10000000 = 128
  11111111 = 255

- n-비트 2진수를 부호 없는 정수 A로 변환하는 방법

  $A = a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} + \ldots + a_1 \times 2^1 + a_0 \times 2^0$

소수와 음수의 표현

- 최상위 비트인 a_{n-1}의 좌측에 소수점이 있는 소수의 10진수 변환방법

$A = a_{n-1} \times 2^{-1} + a_{n-2} \times 2^{-2} + \ldots + a_1 \times 2^{-(n-1)} + a_0 \times 2^{-n}$

$2^3 \quad 2^2 \quad 2^1 \quad 2^0 \quad 2^{-1} \quad 2^{-2} \quad 2^{-3}$

$1 \quad 1 \quad 0 \quad 1 \quad . \quad 1 \quad 0 \quad 1$

$= 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 + 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3}$

$= 8 + 4 + 1 + 0.5 + 0.125 \quad = 13.625$

- 음수 표현 방법
  - -. 부호화-크기 표현(signed-magnitude representation)
  - -. 1의 보수 표현(1's complement representation)
  - -. 2의 보수 표현(2's complement representation)

3.2.1 부호화-크기 표현

- 맨좌측 비트는 부호 비트, 나머지 n-1 개의 비트들은 수의 크기(magnitude)를 나타내는 표현 방식

[예] + 9 = 0 0001001 + 35 = 0 0100011

− 9 = 1 0001001 − 35 = 1 0100011

- 부호화-크기로 표현된 2진수($a_{n-1}\ a_{n-2} \ldots a_1\ a_0$)를 10진수로 변환

$A = (-1)^{an-1}\ (a_{n-2} \times 2^{n-2} + a_{n-3} \times 2^{n-3} + \ldots + a_1 \times 2^1 + a_0 \times 2^0)$

$0\ 0100011 = (-1)^0(0\times2^6+1\times2^5+0\times2^4+0\times2^3+0\times2^2+1\times2^1+1\times2^0)$

$= (32 + 2 + 1) = 35$

$1\ 0001001 = (-1)^1(0\times2^6+0\times2^5+0\times2^4+1\times2^3+0\times2^2+0\times2^1+1\times2^0)$

$= -(8 + 1) = -9$

부호화-크기 표현 (계속)

- 결점
  - -. 덧셈과 뺄셈을 수행하기 위해서는 부호비트와 크기 부분을 별도로 처리
  - -. 0 표현이 두 개 존재

➔ n-비트 단어로 표현할 수 있는 수들이 2^n 개가 아닌, $(2^n - 1)$개로 감소

0 0000000 = + 0

1 0000000 = − 0

3.2.2 보수 표현

- 1의 보수(1's complement) 표현
  - -. 모든 비트들을 반전 (0 → 1, 1→ 0)
- 2의 보수(2's complement) 표현
  - -. 모든 비트들을 반전하고, 결과값에 1을 더한다

[예]

+ 9 = 0 0001001 + 35 = 0 0100011

− 9 = 1 1110110 (1의 보수) − 35 = 1 1011100 (1의 보수)

− 9 = 1 1110111 (2의 보수) − 35 = 1 1011101 (2의 보수)

8-비트 보수로 표현된 정수들

- 8-비트 2진수로 표현할 수 있는 10진수의 범위
  1의 보수 : $-(2^7-1) \sim +(2^7-1)$
  2의 보수 : $-2^7 \sim +(2^7-1)$

10진수	1의 보수	2의 보수
127	01111111	01111111
126	01111110	01111110
⋮	⋮	⋮
1	00000001	00000001
+0	00000000	00000000
−0	11111111	−
−1	11111110	11111111
−2	11111101	11111110
⋮	⋮	⋮
−126	10000001	10000010
−127	10000000	10000001
−128	−	10000000

2의 보수 → 10진수 변환

- 2의 보수로 표현된 양수($a_{n-1}=0$)를 10진수로 변환하는 방법

  $A = a_{n-2} \times 2^{n-2} + a_{n-3} \times 2^{n-3} + \dots a_1 \times 2^1 + a_0 \times 2^0$

- 2의 보수로 표현된 음수($a_{n-1}=1$)를 10진수로 변환하는 방법

  $A = -2^{n-1} + (a_{n-2}\times 2^{n-2} + a_{n-3}\times 2^{n-3} + \dots a_1 \times 2^1 + a_0 \times 2^0)$

  [예] $10101110 = -128 + (1\times 2^5 + 1\times 2^3 + 1\times 2^2 + 1\times 2^1)$

  $= -128 + (32 + 8 + 4 + 2) = -82$

  [다른 방법] 10101110 → 01010010 으로 먼저 변환한 후,

  $01010010 = -(1\times 2^6 + 1\times 2^4 + 1\times 2^1)$

  $= -(64 + 16 + 2) = -82$

3.2.3 비트 확장 (Bit Extension)

- 데이터의 길이(비트 수)를 늘리는 방법
  -. 목적: 데이터를 더 많은 비트의 레지스터에 저장하거나 더 긴 데이터와의 연산 수행

[예] 8-비트 데이터를 16-비트 데이터로 확장

-. 부호화-크기 표현의 경우 : 부호 비트를 맨좌측 위치로 이동시키고, 그 외의 비트들은 0으로 채운다

+21 = 00010101 (부호화-크기, 8 비트)
+21 = 0000000000010101 (부호화-크기, 16 비트)
−21 = 10010101 (부호화-크기, 8 비트)
−21 = 1000000000010101 (부호화-크기, 16 비트)

-. 2의 보수 표현의 경우 : 확장되는 상위 비트들을 부호 비트와 같은 값으로 세트 = 부호 비트 확장(sign-bit extension)

+21 = 00010101 (2의 보수, 8 비트)
+21 = 0000000000010101 (2의 보수, 16 비트)
−21 = 11101011 (2의 보수, 8 비트)
−21 = 1111111111101011 (2의 보수, 16 비트)

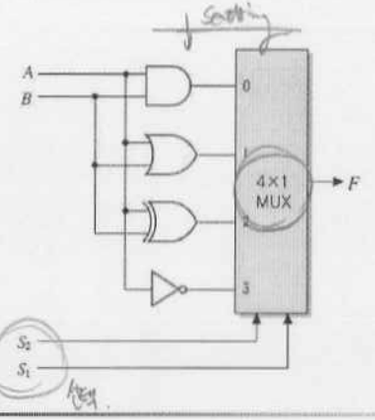
3.3 논리 연산

- 기본적인 논리 연산들

A	B	NOT A	NOT B	A AND B	A OR B	A XOR B
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	1	0

논리 연산을 위한 하드웨어 모듈

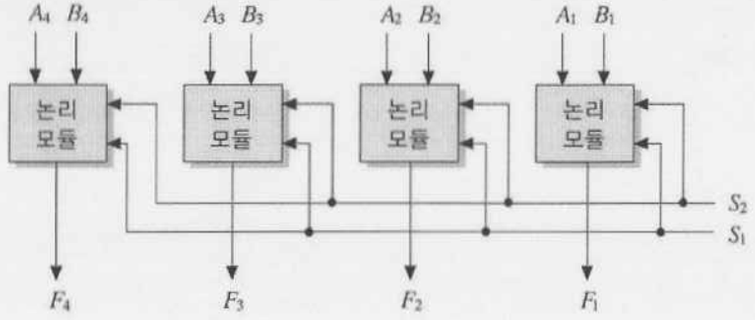
- 하드웨어의 구성
  - -. 입력 비트들은 모든 논리 게이트들을 통과
  - -. 출력 선택 신호들에 의하여 멀티플렉서의 네 입력들 중의 하나를 출력



S_2	S_1	출력	연산
0	0	$F = A \wedge B$	AND
0	1	$F = A \vee B$	OR
1	0	$F = A \oplus B$	XOR
1	1	$F = \overline{A}$	NOT

N-비트 논리 연산장치

- N-비트 데이터들을 위한 논리 연산장치
  - -. 기본 논리 모듈들을 병렬로 접속
- 4-비트 논리 연산장치



AND 연산 / OR 연산

- AND 연산 : 두 데이터 단어들의 대응되는 비트들 간에 AND 연산을 수행

  A = 1011 0101

  B = 0011 1011

  0011 0001 (연산 결과)

- OR 연산 : 두 데이터 단어들의 대응되는 비트들 간에 OR 연산을 수행

  A = 1001 0101

  B = 0011 1011

  1011 1111 (연산 결과)

XOR 연산 / NOT 연산

- XOR 연산 : 두 데이터 단어들의 대응되는 비트들 간에 exclusive-OR 연산을 수행

```
A = 1001 0101
B = 0011 1011
--------------------
    1010 1110   (연산 결과)
```

- NOT 연산 : 데이터 단어의 모든 비트들을 반전(invert)

```
A = 1001 0101   (연산 전)
--------------------------------
    0110 1010   (연산 후)
```


선택적-세트 연산 / 선택적-보수 연산

- 선택적-세트(selective-set) 연산 : B 레지스터의 비트들 중에서 1로 세트된 비트들과 같은 위치에 있는 A 레지스터의 비트들을 1로 세트

<OR 연산 이용>

```
A = 1001 0010   (연산 전)
B = 0000 1111
--------------------
A = 1001 1111   (연산 후)
```

- 선택적-보수(selective-complement) 연산 : B 레지스터의 비트들 중에서 1로 세트된 비트들에 대응되는 A 레지스터의 비트들을 보수로 변환

<XOR 연산 이용>

```
A = 1001 0101   (연산 전)
B = 0000 1111
--------------------
A = 1001 1010   (연산 후)
```


마스크 연산

- 마스크(mask) 연산 : B 레지스터의 비트들 중에서 값이 0인 비트들과 같은 위치에 있는 A 레지스터의 비트들을 0으로 바꾸는(clear하는) 연산

<AND 연산 이용>

-. 용도 : 단어내의 원하는 비트들을 선택적으로 clear하는 데 사용

[예]

```
A = 1101 0101   (연산 전)
B = 0000 1111
--------------------
A = 0000 0101   (연산 후)
```


삽입 연산

- 삽입(insert) 연산 : 새로운 비트 값들을 데이터 단어내의 특정 위치에 삽입

-. 방법 : ① 삽입할 비트 위치들에 대하여 마스크(AND) 연산 수행
② 새로이 삽입할 비트들과 OR 연산을 수행

```
A = 1001 0101
B = 0000 1111   마스크 (AND 연산)
--------------------
A = 0000 0101   첫 단계 결과
B = 1110 0000   삽입 (OR 연산)
--------------------
A = 1110 0101   최종(삽입) 결과
```

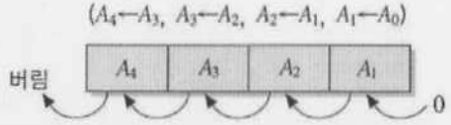
비교 연산

- 비교(compare) 연산
  - -. A와 B 레지스터의 내용을 비교 → exclusive-OR 연산
    - 만약 대응되는 비트들의 값이 같으면, A 레지스터의 해당 비트를 0 세트
    - 만약 서로 다르면, A 레지스터의 해당 비트를 1 세트
    - 모든 비트들이 같으면, Z 플래그를 1 세트

```
A = 1101 0101
B = 1001 0110
-------------------
A = 0100 0011   (연산 결과)
```


3.4 쉬프트(shift) 연산

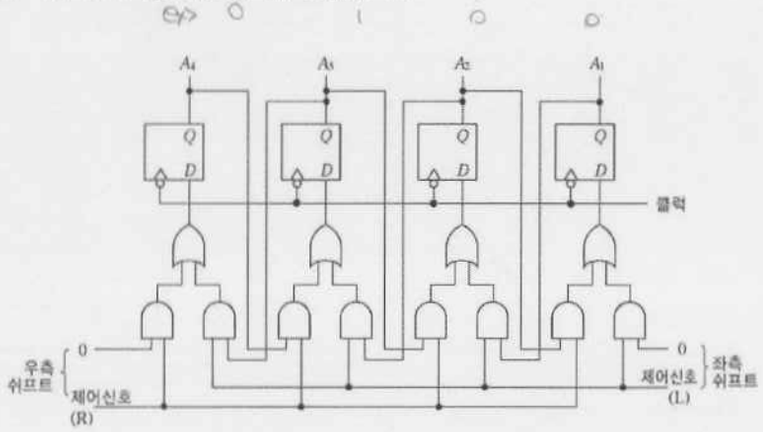
- 논리적 쉬프트 (logical shift) : 레지스터내의 데이터 비트들을 왼쪽 혹은 오른쪽으로 한 칸씩 이동
  - 좌측 쉬프트(left shift)
    - -. 모든 비트들을 좌측으로 한 칸씩 이동
    - -. 최하위 비트(A1)로는 0 이 들어오고, 최상위 비트(A4)는 버림



  - 우측 쉬프트(right shift)
    - -. 모든 비트들이 우측으로 한 칸씩 이동
    - -. 최상위 비트(A4)로 0이 들어오고, 최하위 비트(A0)는 버림

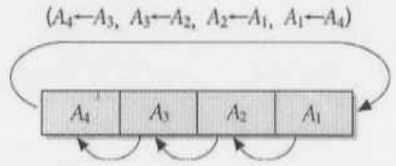
쉬프트 레지스터 (shift register)

- 쉬프트 연산 기능을 가진 레지스터



순환 쉬프트(circular shift)

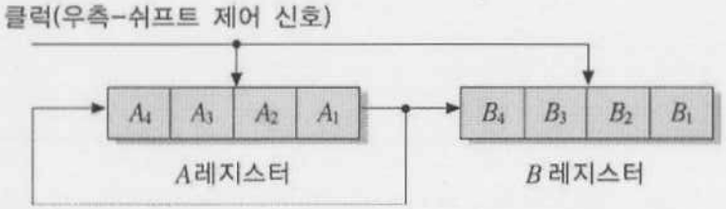
- 순환 쉬프트(circular shift) : 회전(rotate)이라고도 부르며, 최상위 혹은 최하위에 있는 비트를 버리지 않고 반대편 끝에 있는 비트 위치로 이동
  - -. 순환 좌측-쉬프트(circular shift-left)
    - 최상위 비트인 A4가 최하위 비트 위치인 A1으로 이동



  - -. 순환 우측-쉬프트(circular shift-right)
    - $A_4 \rightarrow A_3$, $A_3 \rightarrow A_2$, $A_2 \rightarrow A_1$, $A_1 \rightarrow A_4$

직렬 데이터 전송 (serial data transfer)

- 직렬 데이터 전송 : 쉬프트 연산을 데이터 비트 수만큼 연속적으로 수행함으로써 두 레지스터들 사이에 한 개의 선을 통하여 전체 데이터를 이동하는 동작



4-비트 레지스터들 간의 직렬 전송의 예

	A_4	A_3	A_2	A_1	B_4	B_3	B_2	B_1
초기상태	1	0	1	1	0	0	0	0
t_1	1	1	0	1	1	0	0	0
t_2	1	1	1	0	1	1	0	0
t_3	0	1	1	1	0	1	1	0
t_4	1	0	1	1	1	0	1	1

산술적 쉬프트 (arithmetic shift)

- 수(number)를 나타내는 데이터에 대한 쉬프트
- 방법: 쉬프트 과정에서 부호 비트는 그대로 유지시키고, 수의 크기를 나타내는 비트들만 쉬프트

(1) 산술적 좌측-쉬프트(arithmetic shift-left)

A4 (불변), A3 ← A2, A2 ← A1, A1 ← 0

(2) 산술적 우측-쉬프트(arithmetic shift-right)

A4 (불변), A4 → A3, A3 → A2, A2 → A1

[예] A = 1 1 1 0 (-2) ; 초기 상태

1 1 0 0 (-4) ; 산술적 좌측-쉬프트 결과

1 1 1 0 (-2) ; 산술적 우측-쉬프트 결과

1 1 1 1 (-1) ; 산술적 우측-쉬프트 결과

3.5 정수의 산술 연산

- 기본적인 산술 연산들

A ← A'+ 1 ; 보수화(2의 보수 변환)

A ← A + B ; 덧셈

A ← A - B ; 뺄셈

A ← A x B ; 곱셈

A ← A / B ; 나눗셈

A ← A + 1 ; 증가(increment)

A ← A - 1 ; 감소(decrement)

3.5.1 덧셈

- 2의 보수로 표현된 수들의 덧셈 방법
  - -. 두 수를 더하고, 만약 올림수가 발생하면 버림

(a) (+3) + (+4) = +7

```
  0011
+ 0100
------
  0111 = +7
```

(b) (−3) + (+3) = 0

```
  1101
+ 0011
------
① 0000 = 0
```

(버림)

(c) (−6) + (+2) = −4

```
  1010
+ 0010
------
  1100 = −4
```

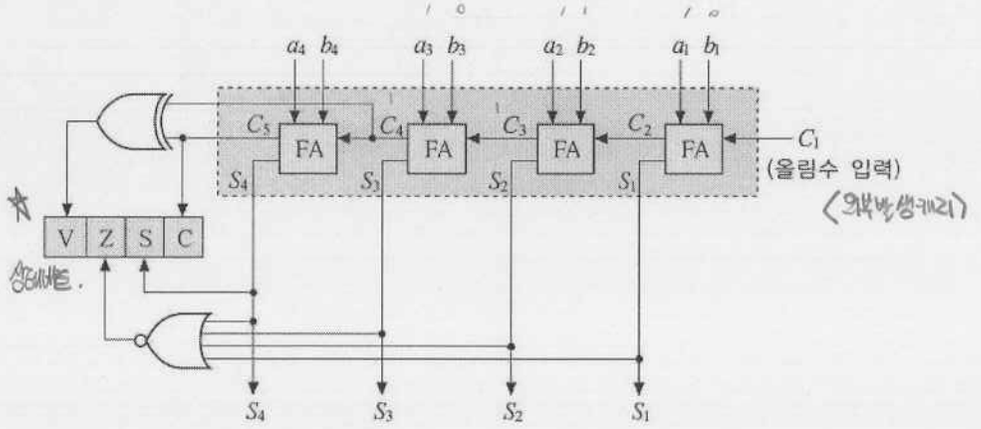
(d) (−4) + (−1) = −5

```
  1100
+ 1111
------
① 1011 = −5
```


병렬 가산기(parallel adder)

- 덧셈을 수행하는 하드웨어 모듈
- 비트 수만큼의 전가산기(full-adder)들로 구성
- 덧셈 연산 결과에 따라 해당 조건 플래그들(condition flags)을 세트
  - -. C 플래그 : 올림수(carry)
  - -. S 플래그 : 부호(sign)
  - -. Z 플래그 : 0(zero)
  - -. V 플래그 : 오버플로우(overflow)

4-비트 병렬 가산기와 상태 비트 제어회로



덧셈 오버플로우

- 덧셈 결과가 그 범위를 초과하여 결과값이 틀리게 되는 상태
- 검출 방법
  - -. 두 올림수(carry)들 간의 exclusive-OR를 이용

    $V = C_4 \oplus C_5$

- 덧셈에서 오버플로우가 발생하는 예

(a) (+6) + (+3) = +9

```
  0110
+ 0011
------
  1001 = −7 (오버플로우)
```

(b) (−7) + (−6) = −13

```
  1001
+ 1010
------
① 0011 = +3 (오버플로우)
```

3.5.2 뺄셈

- 덧셈을 이용하여 수행

  A − (+B) = A + (−B)

  A − (−B) = A + (+B)

  단, A : 피감수(minuend), B : 감수(subtrahend)

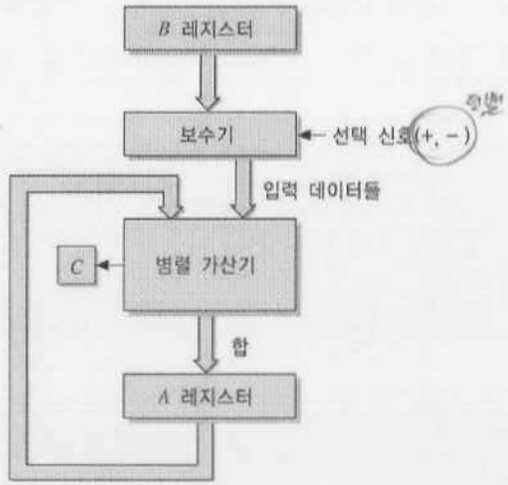
(a) (+2) − (+6) = (+2) + (−6) = −4

```
  0010
+ 1010
------
  1100 = −4
```

(b) (+5) − (+2) = (+5) + (−2) = +3

```
  0101
+ 1110
------
① 0011 = +3
```


덧셈과 뺄셈 겸용 하드웨어의 블록 구성도



뺄셈 오버플로우

- 뺄셈 결과가 그 범위를 초과하여 결과값이 틀리게 되는 상태.
- 검출 방법 : 덧셈과 동일 ($V = C_4 \oplus C_5$)

(a) (+7) − (−5) = (+7) + (+5) = +12

```
  0111
+ 0101
------
  1100 = −4 (오버플로우)
```

(b) (−6) − (+4) = (−6) + (−4) = −10

```
  1010
+ 1100
------
① 0110 = +6 (오버플로우)
```

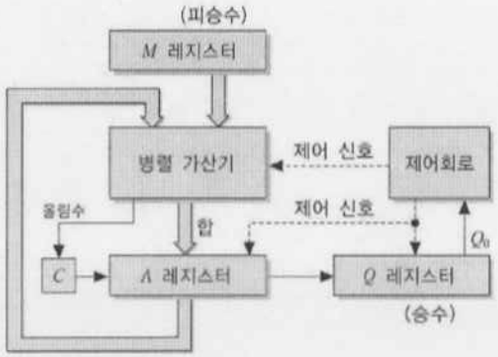

부호 없는 정수의 곱셈

- 부호 없는 정수의 곱셈
  - −. 각 비트에 대하여 부분 적(partial product) 계산
  - −. 부분적들을 모두 더하여 최종 결과를 얻음

```
  1011 (피승수)
× 1101 (승수)
------
  1011
 0000 (부분 적들)
1011
1011
--------
10001111 (최종 결과)
```

부호 없는 정수 승산기의 하드웨어 구성도

- **M** 레지스터 : 피승수(**multiplicand**) 저장
- **Q** 레지스터 : 승수(**multiplier**) 저장
- 두 배 길이의 결과값은 **A** 레지스터와 **Q** 레지스터에 저장



곱셈이 수행되는 과정에서의 레지스터 내용들

	C	A	Q	
[초기 상태]	0	0000	1101	
[사이클 1]	0	1011	1101	; $Q_0 = 1$이므로, $A \leftarrow A+M$.
	0	0101	1110	; 우측 쉬프트$(C-A-Q)$
[사이클 2]	0	0010	1111	; $Q_0 = 0$이므로, 쉬프트$(C-A-Q)$만 한다.
[사이클 3]	0	1101	1111	; $Q_0 = 1$이므로, $A \leftarrow A+M$.
	0	0110	1111	; 우측 쉬프트$(C-A-Q)$
[사이클 4]	1	0001	1111	; $Q_0 = 1$이므로, $A \leftarrow A+M$.
	0	1000	1111	; 우측 쉬프트$(C-A-Q)$

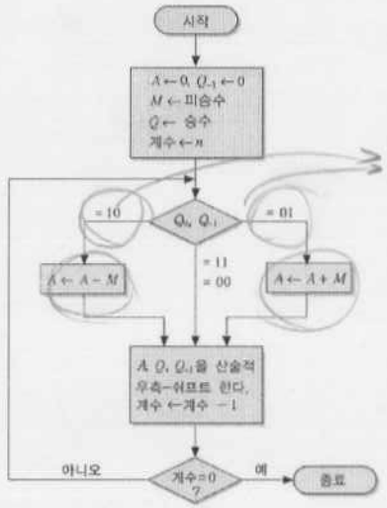
2의 보수들 간의 곱셈

- Booth 알고리즘(Booth's algorithm) 사용
- 하드웨어 구성

  -. 부호 없는 정수 승산기의 하드웨어에 다음 부분을 추가

  - M 레지스터와 병렬 가산기 사이에 보수기(complementer) 추가
  - Q 레지스터의 우측에 Q_{-1} 이라고 부르는 1-비트 레지스터를 추가하고, 출력을 Q_0와 함께 제어 회로로 입력

Booth 알고리즘의 흐름도



Booth 알고리즘을 이용한 곱셈의 예 (−7x3)

(M) 1001 (−7)
(Q) × 0011 Q_{-1} 계수

초기값 : $A = 0000$, $Q_{-1} = 0$, 계수 = 4

(A Q) 0111 0011 0 ; $(Q_0 Q_{-1}) = (1\ 0)$이므로, A로부터 피승수(1001)을 뺀다 (실제로는 그 보수인 0111을 더한다).

0011 1001 1 3 ; AQQ_{-1}을 산술적 우측-쉬프트 하고, 계수에서 1을 뺀다.

0001 1100 1 2 ; $(Q_0 Q_{-1}) = (1\ 1)$이므로, AQQ_{-1}에 대한 산술적 우측-쉬프트만 하고, 계수에서 1을 뺀다.

1010 1100 1 ; $(Q_0 Q_{-1}) = (0\ 1)$이므로, A에 피승수(1001)를 더하고, 계수에서 1을 뺀다.

1101 0110 0 1 ; AQQ_{-1}을 산술적 우측-쉬프트 하고, 계수에서 1을 뺀다.

1110 1011 0 0 ; $(Q_0 Q_{-1}) = (0\ 0)$이므로, AQQ_{-1}을 산술적 우측-쉬프트 한다. 계수에서 1을 빼면 0이므로 계산이 종료되었다.

−21 (곱셈 결과)

3.5.4 나눗셈

- 나눗셈의 수식 표현

  D ÷ V = Q --- R

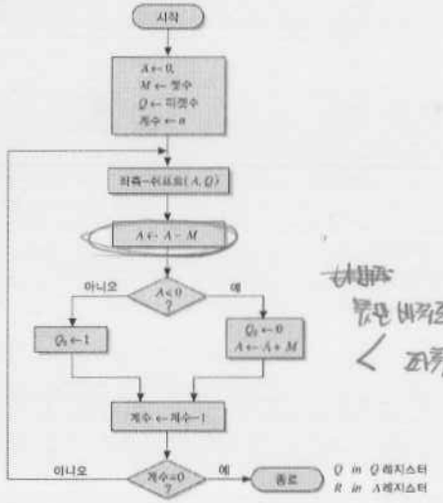
  단, D = 피젯수(dividend), V = 젯수(divisor), Q = 몫(quotient)

  V = 젯수(divisor), R = 나머지 수(remainder)

- 부호 없는 2진 나눗셈

```
          00001101   ← 몫(Q)
젯수(V) → 1011)10010011 ← 피젯수(D)
          -1011
          001110
          -1011
부분      001111
나머지 수 -1011
          100      ← 나머지 수(R)
```


부호 없는 2진 나눗셈 알고리즘의 흐름도



2의 보수 나눗셈 과정

- [초기 상태] 젯수는 M 레지스터에, 피젯수는 A와 Q 레지스터에 저장

  각 레지스터가 n 비트일 때, 피젯수는 2^n 비트 길이의 2의 보수로 표시

- [사이클 1] A와 Q 레지스터를 좌측으로 한 비트씩 쉬프트
- [사이클 2] 만약 M과 A의 부호가 같으면 A ← A − M,

  다르면 A ← A + M을 수행한다.

- [사이클 3] 연산 전과 후의 A의 부호가 같으면 위의 연산은 성공
  - −. 연산이 성공이거나 A = 0 이면, Q_0 ← 1로 세트
  - −. 연산이 실패이고 A ≠ 0 이면, Q_0 ← 0으로 하고 A를 이전의 값으로 복구
- [사이클 4] Q에 비트 자리 수가 남아있다면, 단계 2에서 4까지를 반복
- [사이클 5] 나머지 수는 A. 만약 젯수와 피젯수의 부호가 같으면 몫은 Q의 값이고, 그렇지 않으면 Q의 2의 보수가 몫

2의 보수 나눗셈의 예 (7 ÷ (-3))

A	Q	M = 1101 (-3)
0000	0111	; 초기 상태
0000	1110	; 좌측-쉬프트(A-Q).
1101		; A와 M의 부호가 서로 다르므로, A ← A + M.
0000	1110	; A의 부호가 바뀌었으므로, A의 원래값을 복구.
0001	1100	; 좌측-쉬프트(A-Q).
1110		; A와 M의 부호가 서로 다르므로, A ← A + M.
0001	1100	; A의 부호가 바뀌었으므로, A의 원래값을 복구.
0011	1000	; 좌측-쉬프트(A-Q).
0000		; A와 M의 부호가 서로 다르므로, A ← A + M.
0000	1001	; A = 0이므로, Q_0 ← 1로 세트
0001	0010	; 좌측-쉬프트(A-Q).
1110		; A와 M의 부호가 서로 다르므로, A ← A + M.
0001	0010	; A의 부호가 바뀌었으므로, A의 원래값을 복구.

연산 결과 : 몫 = Q의 내용(0010)에 대한 2의 보수 = 1110 (-2)
나머지 = A의 내용 = 0001 (+1)

3.6 부동소수점 수의 표현

- 부동소수점 표현(floating-point representation) : 소수점의 위치를 이동시킬 수 있는 표현 방법
- 부동소수점 수(floating-point number)의 일반적인 형태

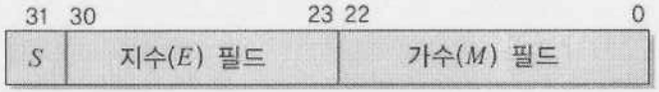
$$N = (-1)^S M \times B^E$$

단, S: 부호(sign), M: 가수(mantissa), B: 기수(base), E: 지수(exponent)

- 2진 부동소수점 수(binary floating-point number)
  - -. 기수 B = 2
  - -. 단일-정밀도(single-precision) 부동소수점 수 : 32 비트
  - -. 복수-정밀도(double-precision) 부동소수점 수 : 64 비트

단일-정밀도 부동소수점 수 형식의 예

- S: 1 비트, E: 8 비트, M: 23 비트
- 지수(E) 필드의 비트 수가 늘어나면, 표현 가능한 수의 범위 확장
- 가수(M) 필드의 비트 수가 늘어나면, 정밀도(precision) 증가



같은 수에 대한 부동소수점 표현

- 같은 수에 대한 부동소수점 표현이 여러 가지가 존재

0.1101×2^5

110.1×2^2

0.01101×2^6

- 정규화된 표현(normalized representation)
  - -. 수에 대한 표현을 한 가지로 통일하기 위한 방법

$$\pm 0.1bbb...b \times 2^E$$

  - -. 위의 예에서 정규화된 표현은 0.1101×2^5

비트 배열의 예 (0.1101×2^5)

- 부호(S) 비트 = 0
- 지수(E) = 00000101
- 가수(M) = 1101 0000 0000 0000 0000 000
- 소수점 아래 첫 번째 비트는 항상 1이므로, 저장할 필요가 없음

  → 가수 23비트를 이용하여 소수점 아래 24 자리 수까지 표현 가능

데이터 표현:

S	E	M
0	00000101	11010000000000000000000

바이어스된 지수 (biased exponent)

- 지수를 바이어스된 수(biased number)로 표현
- 사용 목적

  -. 0에 대한 표현에서 모든 비트들이 0이 되게 하여, 0-검사(zero-test)

  가 용이하게 하기 위함

  → 0-검사가 정수에서와 같은 방법으로 가능

8-비트 바이어스된 지수값들

지수 비트 패턴	절대값	실제 지수값 바이어스=127	실제 지수값 바이어스=128
11111111	255	+128	+127
11111110	254	+127	+126
⋮	⋮	⋮	⋮
10000001	129	+2	+1
10000000	128	+1	0
01111111	127	0	−1
01111110	126	−1	−2
⋮	⋮	⋮	⋮
00000001	1	−126	−127
00000000	0	−127	−128

바이어스된 지수를 사용한 부동소수점 표현의 예

- 바이어스값 = 128일 때, N = − 13.625에 대한 부동소수점 표현

  $13.625_{10} = 1101.101_2 = 0.1101101 \times 24$

  부호(S) 비트 = 1 (−)

  지수(E) = 00000100 + 10000000 = 10000100

  (바이어스 128을 더한다)

  가수(M) = 10110100000000000000000

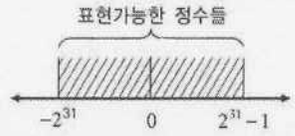
  (소수점 우측의 첫 번째 1은 제외)

S	E	M
1	10000100	10110100000000000000000

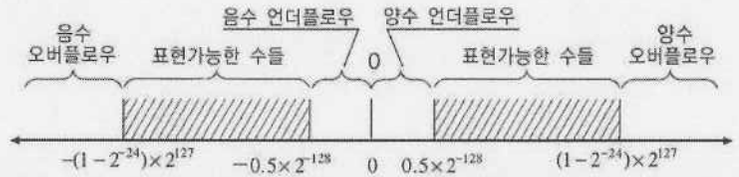
부동소수점 수의 표현 범위

- 부동소수점 수의 표현 범위
  - 0.5×2^{-128}에서 $(1 - 2^{-24}) \times 2^{127}$ 사이의 양수들
    (대략 $1.47 \times 10^{-39} \sim 1.7 \times 10^{38}$)
  - $-(1 - 2^{-24}) \times 2^{127}$ 에서 -0.5×2^{-128} 사이의 음수들
- 제외되는 범위
  - $(1 - 2^{-24}) \times 2^{127}$보다 작은 음수 → 음수 오버플로우(negative overflow)
  - 0.5×2^{-128} 보다 큰 음수 → 음수 언더플로우(negative underflow)
  - 0
  - 0.5×2^{-128} 보다 작은 양수 → 양수 언더플로우(positive underflow)
  - $(1 - 2^{-24}) \times 2^{127}$ 보다 큰 양수→ 양수 오버플로우(positive overflow)

32-비트 데이터 형식의 표현 가능한 수의 범위



(a) 2의 보수 정수의 표현 범위



(b) 부동소수점 수의 표현 범위

IEEE 754 표준 부동소수점 수의 형식

- 부동소수점 수의 표현 방식의 통일을 위하여 미국전기전자공학회(IEEE)에서 정의한 표준
- 표현 방법

$$N = (-1)^S \, 2^{E-127} \, (1.M)$$

-. 가수 : 부호화-크기 표현 사용

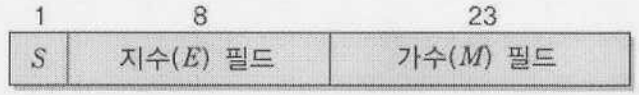
-. 지수 필드 : 바이어스 127 사용

-. $1.M \times 2^E$의 형태를 가지며, 소수점 아래의 M 부분만 가수 필드에 저장
(소수점 왼쪽의 표현되지 않는 1을 hidden bit라고 지칭)

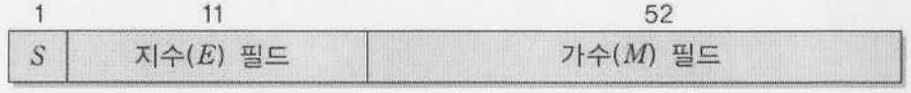
-. 64-비트 복수-정밀도 부동소수점 형식을 사용하는 경우

$$N = (-1)^S \, 2^{E-1023} \, (1.M)$$

IEEE 754 표준 부동소수점 수의 형식



(a) 단일-정밀도 형식(single-precision format)



(b) 복수-정밀도 형식(double-precision format)

IEEE 754 표현 예 (N = − 13.625)

- $13.625_{10} = 1101.101_2 = 1.101101 \times 2^3$

  부호(S) 비트 = 1 (−)

  지수 E = 00000011 + 01111111 = 10000010

  (바이어스 127을 더한다)

  가수 M = 10110100000000000000000

  (소수점 좌측의 1은 제외한다)

S	E	M
0	10000010	10110100000000000000000

예외 경우를 포함한 IEEE 754 표준

- 예외 경우를 포함한 정의 (32-비트 형식)
  - −. 만약 E = 255이고 M ≠ 00이면, N = NaN
  - −. 만약 E = 255이고 M = 00이면, N = $(-1)^S \infty$
  - −. 만약 0 < E < 255 이면, N = $(-1)^S 2^{E-127}$ (1.M)
  - −. 만약 E = 00이고 M ≠ 00이면, N = $(-1)^S 2^{-126}$ (0.M)
  - −. 만약 E = 00이고 M = 00이면, N = $(-1)^S$ 0
- 예외 경우를 포함한 정의 (64-비트 형식)
  - −. 만약 E = 2047이고 M ≠ 00이면, N = NaN
  - −. 만약 E = 2047이고 M = 00이면, N = $(-1)^S \infty$
  - −. 만약 0 < E < 2047 이면, N = $(-1)^S 2^{E-1023}$ (1.M)
  - −. 만약 E = 00이고 M ≠ 00이면, N = $(-1)^S 2^{-1022}$ (0.M)
  - −. 만약 E = 00이고 M = 00이면, N = $(-1)^S$ 0

부동소수점 덧셈 / 뺄셈

- 덧셈과 뺄셈
  - −. 지수들이 일치되도록 조정 (alignment)
  - −. 가수들 간의 연산(더하기 혹은 빼기) 수행
  - −. 결과를 정규화 (normalization)

① 지수 조정

$$0.110100 \times 2^3 \Longrightarrow 0.001101 \times 2^5$$

$$+\ 0.111100 \times 2^5 \qquad +\ 0.111100 \times 2^5$$

② 더하기

$$1.001001 \times 2^5$$

③ 정규화

$$\Longrightarrow 0.1001001 \times 2^6$$

(최종 결과)

부동소수점 산술의 파이프라이닝

- 단계 수만큼의 속도 향상
- 대규모의 부동소수점 계산을 처리하는 거의 모든 슈퍼컴퓨터들에서 채택

부동소수점 곱셈 / 나눗셈

- 2진수 부동소수점 곱셈 과정
  - -. 가수들을 곱한다
  - -. 지수들을 더한다
  - -. 결과값을 정규화
- 2진수 부동소수점 나눗셈 과정
  - -. 가수들을 나눈다
  - -. 피젯수의 지수에서 젯수의 지수를 뺀다
  - -. 결과값을 정규화

[부동소수점 곱셈의 예]

$(0.1011 \times 2^3) \times (0.1001 \times 2^5)$

<가수 곱하기>

$1011 \times 1001 = 01100011$

<지수 더하기>

$3 + 5 = 8$

<정규화>

0.01100011×2^8

$= 0.1100011 \times 2^7$ (결과값)

부동소수점 연산 과정에서 발생 가능한 문제들

- 지수 오버플로우(exponent overflow)
  - -. 양의 지수값이 최대 지수값을 초과
  - -. 수가 너무 커서 표현될 수 없는 상태이므로, $+\infty$ 또는 $-\infty$로 세트
- 지수 언더플로우(exponent underflow)
  - -. 음의 지수값이 최대 지수값을 초과
  - -. 수가 너무 작아서 표현될 수 없는 상태이므로, 0으로 세트

부동소수점 연산 과정에서 발생 가능한 문제들 (계속)

- 가수 언더플로우(mantissa underflow)
  - -. 가수의 소수점 위치 조정 과정에서 비트들이 가수의 우측 편으로 넘치는 경우
    - ➔ 반올림(rounding) 적용
- 가수 오버플로우(mantissa overflow)
  - -. 같은 부호를 가진 두 가수들을 덧셈하였을 때 올림수가 발생하는 경우
    - ➔ 재조정(realignment) 과정을 통하여 정규화

컴퓨터 구조 레포트

* 컴퓨터 산술연산 관련문제입니다. 주어진 문제를 풀이과정을 포함하여 자필로 작성하여주시기 바랍니다.

<유형 1> 부호없는 정수 곱셈

1번> 11 x 13
2번> 7 x 3

<유형 2> 2의 보수들 간의 곱셈 (Booth 알고리즘 이용)

3번> Booth 알고리즘의 흐름도를 그리세요.
4번> (-7) x 3
5번> (-6) x 5
6번> 7 x (-4)

<유형 3> 나눗셈

7번> 부호없는 2진 나눗셈 알고리즘의 흐름도를 그리세요.
8번> 7 / (-3)
9번> 5 / 2
10번> 7 / (-2)