CPU의 구조와 기능

연세대학교

CPU의 명령어 수행 과정

연세대학교

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

➔ 모든 명령어들에 대하여 공통적으로 수행

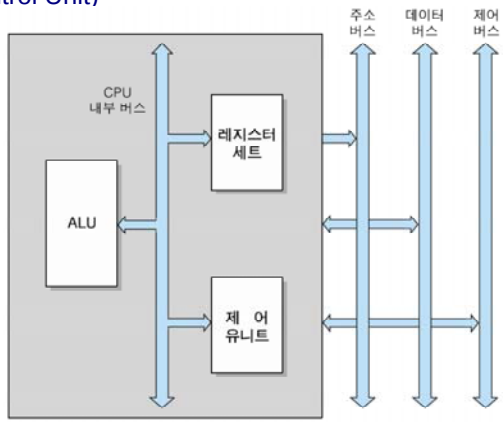
- 데이터 인출(Data Fetch) :
  - 명령어 실행을 위하여 데이터가 필요한 경우에는 기억장치 혹은 I/O 장치로부터 그 데이터를 읽어온다
- 데이터 처리(Data Process) :
  - 데이터에 대한 산술적 혹은 논리적 연산을 수행
- 데이터 쓰기(Data Store) :
  - 수행한 결과를 저장

➔ 명령어에 따라 필요한 경우에만 수행

2.1 CPU의 기본 구조

연세대학교

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



CPU의 내부 구성요소

연세대학교

- ALU
  - 각종 산술 연산들과 논리 연산들을 수행하는 회로들로 이루어진 하드웨어 모듈
  - 산술 연산 : 덧셈, 뺄셈, 곱셈, 나눗셈
  - 논리 연산 : AND, OR, NOT, XOR 등

- 레지스터(register)
  - CPU내부에 위치한 액세스 속도가 가장 빠른 기억장치
  - 레지스터들의 수가 제한됨
    (특수 목적용 레지스터들과 적은 수의 일반 목적용 레지스터들)

CPU의 내부 구성요소 (계속)

연세대학교

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

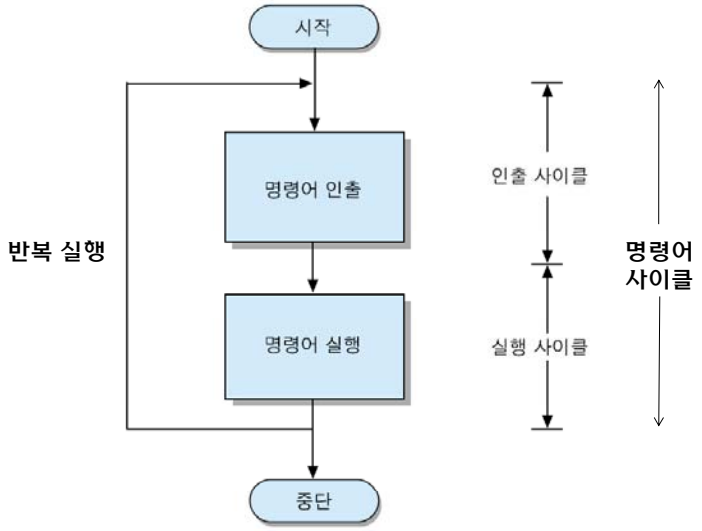
2.2 명령어 실행

연세대학교

- 명령어 사이클 (instruction cycle)
  - CPU가 한 개의 명령어를 실행하는 데 필요한 전체 처리 과정
  - 명령어 인출과 명령어 실행의 부사이클로 구성됨
- 부사이클(subcycle)
  - 인출 사이클(fetch cycle) :
    - CPU가 기억장치로부터 명령어를 읽어오는 단계
  - 실행 사이클(execution cycle) :
    - 명령어를 실행하는 단계

기본 명령어 사이클

연세대학교



간단한 CPU

연세대학교

- 명령어 집합

명령어	동작
LOAD addr	AC ← M[addr]
STA addr	M[addr] ← AC
ADD addr	AC ← AC + M[addr]
JUMP addr	PC ← addr

- 명령어 형식

	opcode
LOAD	1(0001)
STA	2(0010)
ADD	5(0101)
JUMP	8(1000)

CPU 레지스터

연세대학교

- CPU 레지스터
  - AC (Accumulator,누산기)
  - PC (Program Counter)
  - IR (Instruction Register)

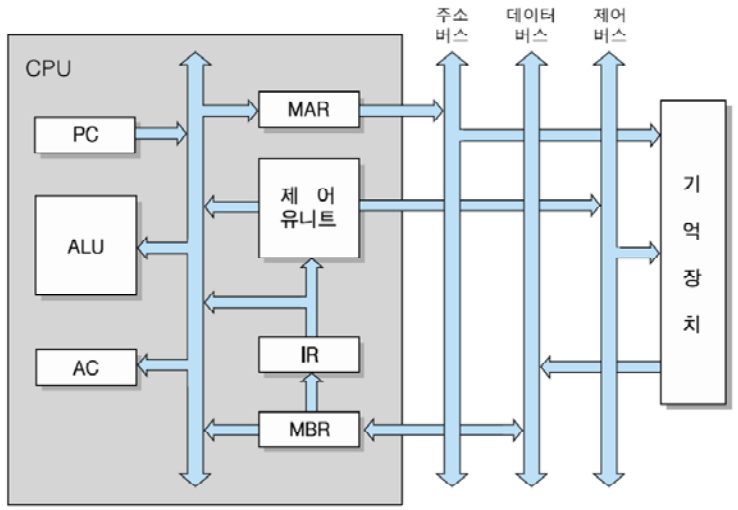
  } CPU 내부에 위치

  - MAR (Memory Address Register)
  - MBR (Memory Buffer Register) 또는 MDR(Memory Data Reg)

  } 시스템 버스와의 인터페이스에 위치

데이터 경로가 표시된 CPU 내부 구조

연세대학교



CPU 내부 레지스터들

연세대학교

- 프로그램 카운터(Program Counter: PC)
  - 다음에 인출할 명령어의 주소를 가지고 있는 레지스터
  - 각 명령어가 인출된 후에는 명령어 길이만큼 주소를 증가
  - 분기(jump) 명령어가 실행되는 경우에는 목적지 주소로 갱신됨
- 명령어 레지스터(Instruction Register: IR)
  - 가장 최근에 인출된 명령어 코드가 저장되어 있는 레지스터
- 누산기(Accumulator: AC)
  - 데이터를 일시적으로 저장하는 레지스터.
  - 레지스터의 크기는 CPU가 한 번에 처리할 수 있는 데이터 비트수

CPU 내부 레지스터들 (계속)

연세대학교

- 기억장치 주소 레지스터(Memory Address Register: MAR)
  - 명령어 또는 데이터의 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

인출 사이클

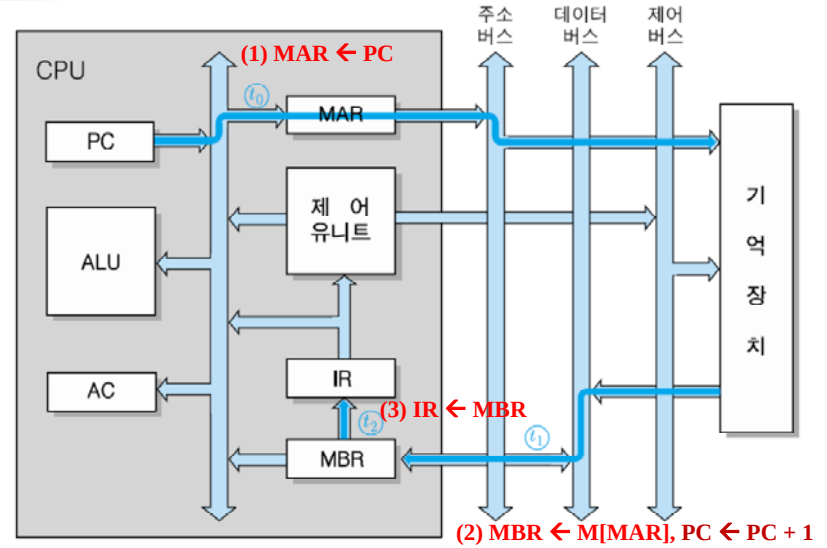
연세대학교

- 마이크로 연산(micro -operation)
  - 한 CPU 클럭 주기에 수행되는 CPU 내의 동작
- 기계어 명령어의 실행
  - 여러 단계의 마이크로 연산들로 이루어짐

- 인출 사이클의 마이크로 연산 (IR ← M[MAR])
  - t0 : MAR ← PC
  - t1 : MBR ← M[MAR], PC ← PC + 1
  - t2 : IR ← MBR

인출 사이클의 주소 및 명령어 흐름도

연세대학교



실행 사이클

연세대학교

- CPU는 실행 사이클 동안에 명령어 코드를 해독(decode)하고, 그 결과에 따라 필요한 연산들을 수행
  - 실행 사이클에서 수행되는 마이크로연산들은 명령어에 따라 다름

- CPU가 수행하는 연산들의 종류
  - 데이터 이동 : CPU와 기억장치 혹은 I/O장치 간에 데이터를 이동
    - LOAD addr, STA addr
  - 데이터 처리 : 데이터에 대하여 산술 혹은 논리 연산을 수행
    - ADD addr
  - 프로그램 제어 : 프로그램의 실행 순서를 결정
    - JMP addr

Load, Store 명령어

연세대학교

- *LOAD addr* (AC ← M[addr])
  - 기억장치에 저장되어 있는 데이터를 AC로 이동하는 명령어
  - 실행사이클의 마이크로연산
    - t0 : MAR ← IR(addr)
    - t1 : MBR ← M[MAR]
    - t2 : AC ← MBR
- *STA addr* (M[addr] ← AC)
  - AC 레지스터의 내용을 기억장치에 저장하는 명령어
  - 실행사이클의 마이크로연산
    - t0 : MAR ← IR(addr)
    - t1 : MBR ← AC
    - t2 : M[MAR] ← MBR

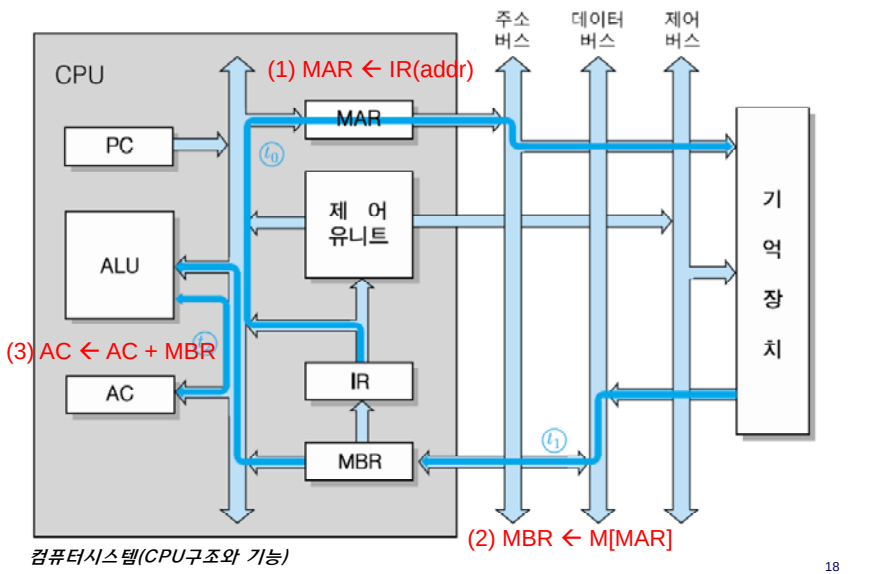
ADD 명령어

연세대학교

- *ADD addr* (AC ← AC + M[addr])
  - 기억장치에 저장된 데이터를 AC의 내용과 더하고, 그 결과를 다시 AC에 저장하는 명령어
  - 실행사이클의 마이크로연산
    - t0 : MAR ← IR(addr)
    - t1 : MBR ← M[MAR]
    - t2 : AC ← AC + MBR

ADD 명령어 실행 사이클

연세대학교



JUMP 명령어

연세대학교

- *JUMP addr* (PC ← addr)
  - 오퍼랜드 필드(addr)가 가리키는 주소의 명령어로 실행 순서를 변경하는 분기(branch) 명령어
  - 실행사이클의 마이크로 연산
    - t0 : PC ← IR(addr)

어셈블리 프로그램 실행과정의 예

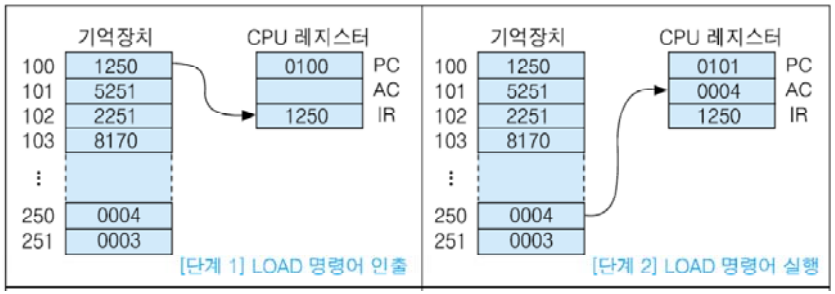
연세대학교

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)

연세대학교

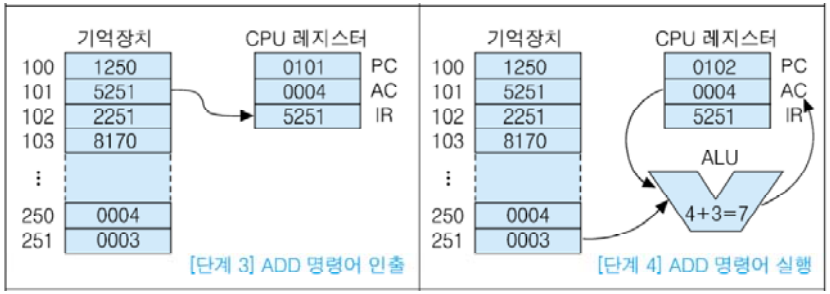
- LOAD 250
  - 인출: 100번지의 첫 번째 명령어 코드가 인출되어 IR에 저장
    PC = PC + 1 = 101
  - 실행: 250번지의 데이터(0004)를 AC로 이동



어셈블리 프로그램 실행과정의 예(계속)

연세대학교

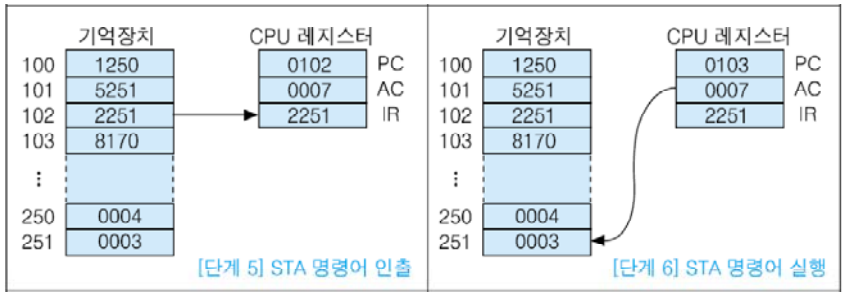
- ADD 251
  - 인출: 두 번째 명령어가 101번지로부터 인출되어 IR에 저장
    PC = PC + 1 = 102
  - 실행: AC와 251 번지의 내용을 더하고, 결과를 AC에 저장



어셈블리 프로그램 실행과정의 예(계속)

연세대학교

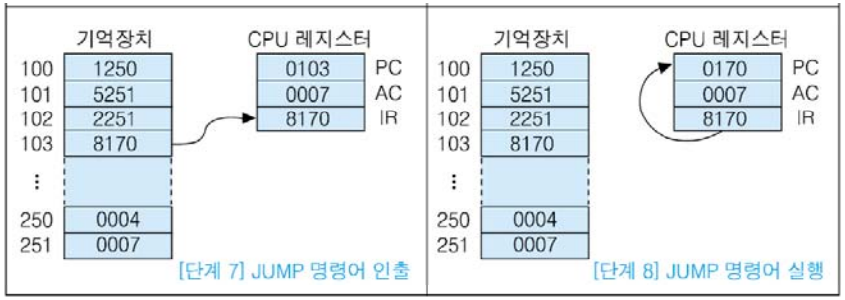
- STA 251
  - 인출: 세 번째 명령어가 102 번지로부터 인출되어 IR에 저장
    PC = PC + 1 = 103
  - 실행: AC의 내용이 251 번지에 저장



어셈블리 프로그램 실행과정의 예(계속)

연세대학교

- JUMP 170
  - 인출 : 네 번째 명령어가 103 번지로부터 인출되어 IR에 저장
    PC = PC + 1 = 104
  - 실행: 분기될 목적지 주소(IR의 오퍼랜드 부분)가 PC로 적재, 즉,
    PC = 170 → 다음에 170 번지의 명령어 인출



간접 주소지정 방식

연세대학교

- 간접주소지정 방식(indirect addressing)
  - 명령어에 데이터의 주소가 아니라 데이터의 주소가 저장된 메모리 주소를 저장하는 방식
- 간접 주소지정을 포함하는 명령어 형식

  - I=0: 직접주소지정 방식
    - addr = A, 오퍼랜드 = M[A]
  - I=1: 간접주소지정방식
    - addr = M[A], 오퍼랜드 = M[addr] = M[M[A]]
    - 오퍼랜드를 얻기 위해서 메모리를 두 번 읽어야 함.

ADD **[250]** ; AC ← AC + M[**M[250]**]

간접 사이클(indirect cycle)

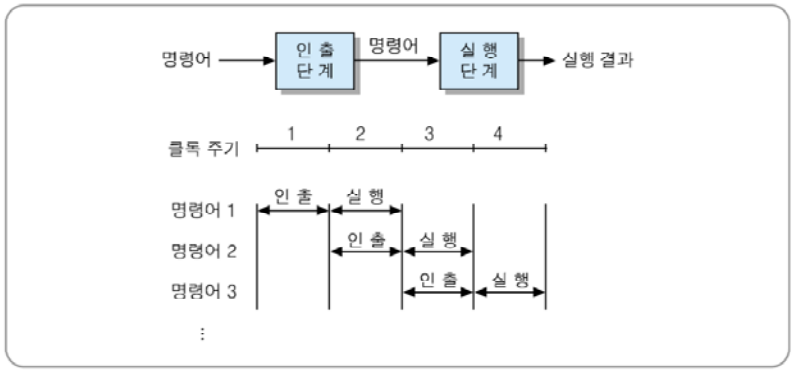
연세대학교

- 간접 사이클
  - 간접 주소지정 방식을 사용하는 명령어 실행에서 메모리에 저장된 데이터의 주소를 인출하는 사이클
  - 명령어의 간접 비트(I)가 1인 경우에 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

연세대학교

- 명령어 파이프라이닝(instruction pipelining)
  - CPU의 프로그램 처리 속도를 높이기 위하여 CPU 내부 하드웨어를 여러 단계로 나누어 여러 명령어를 중첩하여 처리하는 기술
- 2-단계 명령어 파이프라인



2-단계 명령어 파이프라인 (계속)

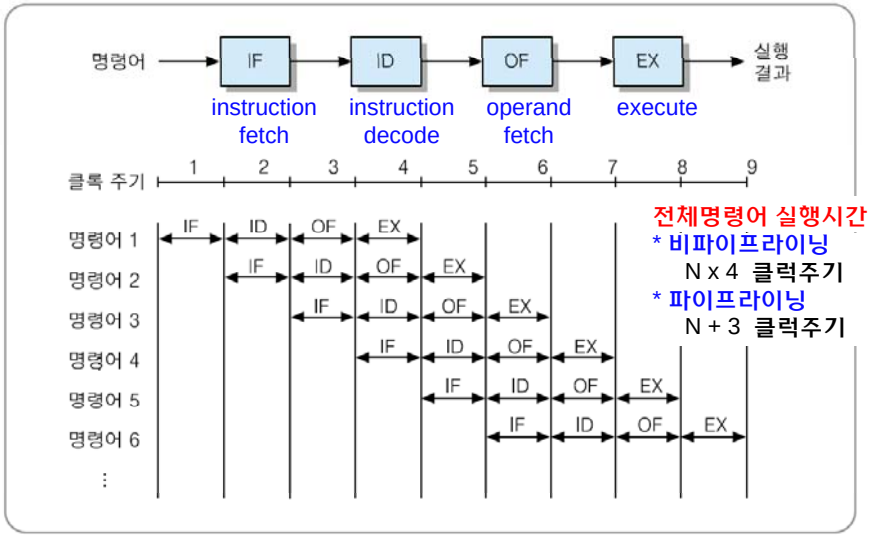
연세대학교

- 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)
- 해결책 :
  - 파이프라인 단계의 수를 증가시켜 각 단계의 처리 시간을 같게 함

(파이프라인 단계의 수를 늘리면 전체적으로 속도 향상도 더 높아짐)

4-단계 명령어 파이프라인

연세대학교



파이프라인에 의한 속도 향상

연세대학교

- 가정
  - 파이프라인 단계 수 = k
  - 실행할 명령어들의 수 = N
  - 각 파이프라인 단계가 한 클럭 주기씩 걸린다고 가정
- 파이프라인에 의한 전체 명령어 실행 시간 T :

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

파이프라인 효율 저하 요인 및 해결책

연세대학교

- 특정 단계의 소요시간 증가
  - 메모리 접근 시에 소요시간이 증가함 → 내부 캐시 메모리 사용
  - 복잡한 연산 실행 → 파이프라인 단계 증가, 연산을 여러 단계로 나누어서 실행
- 기억장치 접근 충돌
  - IF 단계와 OF 단계가 동시에 메모리를 액세스하는 경우에 기억장치 충돌(memory conflict)이 일어나면 지연이 발생한다
    → 명령어 캐시와 데이터 캐시 메모리를 분리하여 사용
- 파이프라인 무효화
  - 조건 분기(conditional branch) 명령어가 실행되면, 미리 인출하여 처리하던 명령어들이 무효화된다

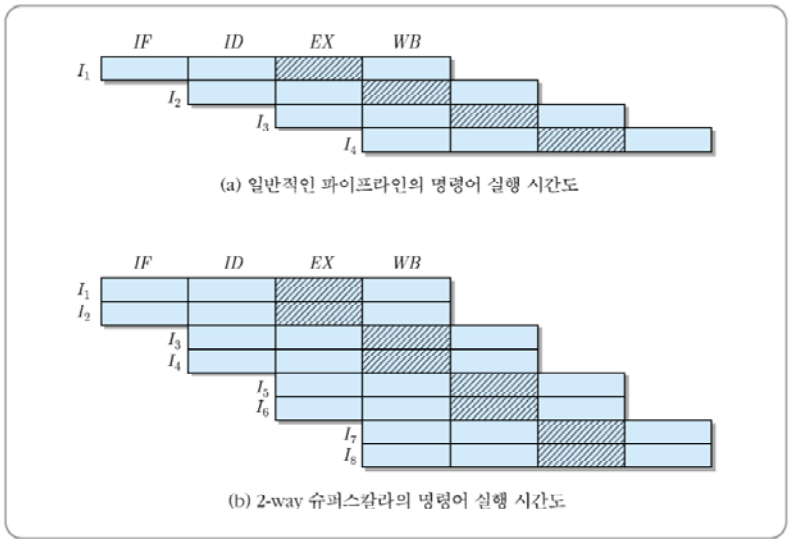
  → 지연분기 (분기 명령어 다음의 명령어 실행 후 분기함),
  분기예측 (분기 명령 다음에 실행할 명령어를 예측하여 인출),
  분기목적지 선인출(다음 명령어와 분기목적지 명령어 함께 인출)

슈퍼스칼라(Superscalar)

연세대학교

- CPU의 처리 속도를 더욱 높이기 위하여 내부에 여러 개의 명령어 파이프라인들을 포함시킨 구조
- m-way 슈퍼스칼라 구조
  - m개의 명령어 파이프라인을 가진 CPU 구조
- 매 클록 주기마다 각 명령어 파이프라인이 별도의 명령어를 인출하여 동시에 실행할 수 있기 때문에,이론적으로는 프로그램 처리 속도가 파이프라인의 수만큼 향상 가능
- 슈퍼스칼라의 속도 저하 (Sp < m) 요인 및 해결책
  - 동시 실행 가능한 명령어 수가 m개 미만
  - 해결책:
    - 명령어 실행 순서 재배치: 명령어들 간의 데이터 의존성 제거

2-way슈퍼스칼라 명령어 실행 흐름도



예: Pentium

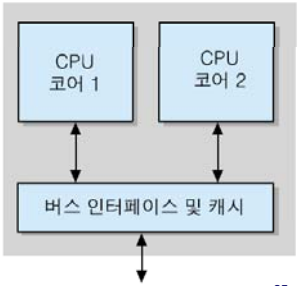
intel

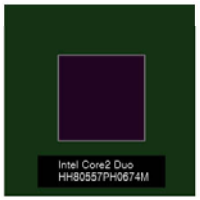
PENTIUM® PROCESSOR

- **Compatible with Large Software Base**
  - **MS-DOS*, Windows*, OS/2*, UNIX***
- **32-Bit CPU with 64-Bit Data Bus**
- **Superscalar Architecture**
  - **Two Pipelined Integer Units Are Capable of 2 Instructions/Clock**
  - **Pipe-lined Floating Point Unit**
- **Separate Code and Data Caches**
  - **8-Kbyte Code, 8-Kbyte Write Back Data**
  - **MESI Cache Protocol**
- **Advanced Design Features**
  - **Branch Prediction**
  - **Virtual Mode Extensions**

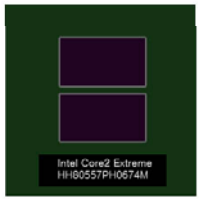
멀티-코어

- CPU 코어(core) :
  - 명령어 실행에 필요한 CPU 내부의 핵심 하드웨어(슈퍼스칼라) 모듈
- 멀티-코어 프로세서(multi-core processor): 여러 개의 CPU 코어들을 하나의 칩에 포함시킨 프로세서
  - 듀얼-코어(dual-core): 두 개의 CPU 코어 포함
  - 쿼드-코어(quad-core): 네 개의 CPU 코어 포함
- 각 CPU 코어는 시스템 버스와 캐쉬 공유
- 각 코어는 독립적으로 프로그램을 실행함



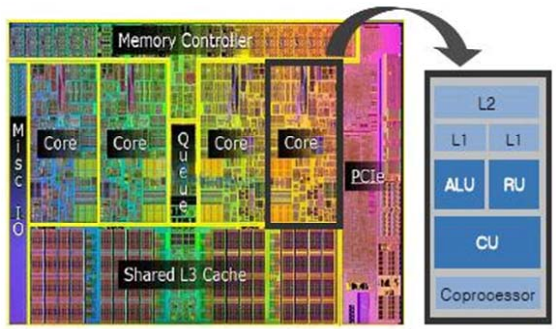


single die (dual core)



dual die (quad core)

dual die (dual core)



주: Intel Core i7 Lynnfield