Class Topics (클래스 홈페이지 참조)

- Part 1: Fundamental concepts and principles
  1) Invention of computers and digital logic design
  2) Abstractions to deal with complexity
  3) Data (versus code)
  4) Machines called computers
  5) Underlying technology and evolution since 1945
- Part 2: 빠른 컴퓨터를 위한 설계 (ISA design)
- Part 3: 빠른 컴퓨터를 위한 구현 (ISA implementation)

Machines Called Computers

Part 2

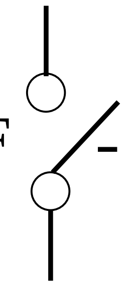
- More on "Abstraction"

(how to deal with complexity)

References:

1. Computer Organization and Design & Computer Architecture, Hennessy and Patterson (slides are adapted from those by the authors)

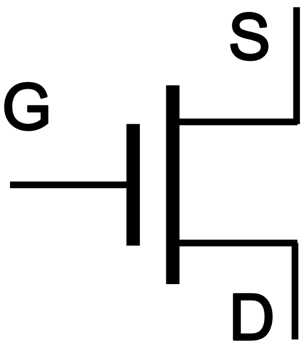
3-Terminal Digital Switches
(구현 기술)

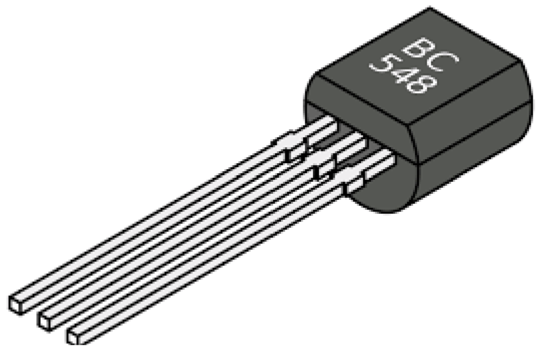


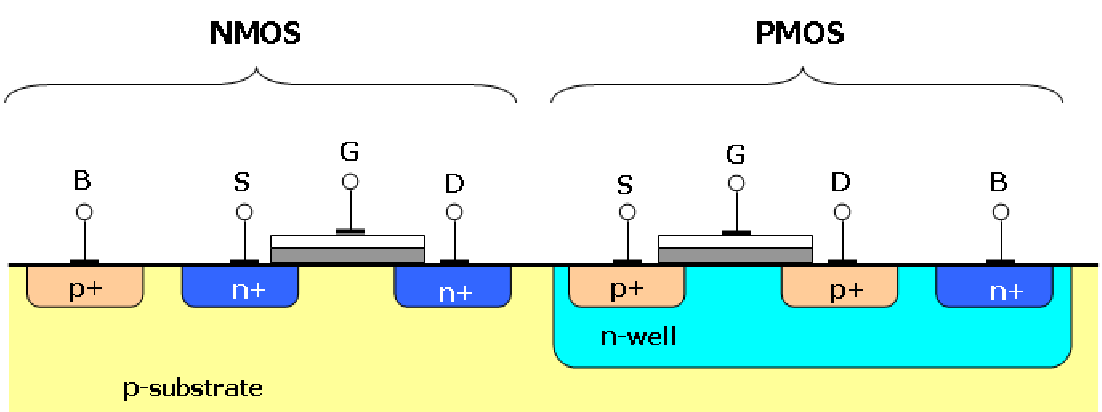
- ❑ None in mechanical era
- ❑ Electromagnetic relay (릴레이)
  - Invented in 1835 (speed: 10^{-3} second)
- ❑ Vacuum tube (진공관; speed: 10^{-6} second)
  - Invented in 1906; first commercial use in 1920
  - 라디오, TV, 오디오, 전화설비, ENIAC, …
- ❑ Transistor – dream device (speed: 10^{-11} second)
  - Invented in 1947; 실용화에 10년 걸림
  - Small, fast, reliable, energy-efficient, inexpensive
  - Integrated Circuits (IC) 형태로 제작 가능

Digital Switches – Transistors

- Solid-state semiconductor devices
  - "Transistors" by Bell Labs. in 1947 (cf. ENIAC in 1946)
  - Integrated circuits in 1958



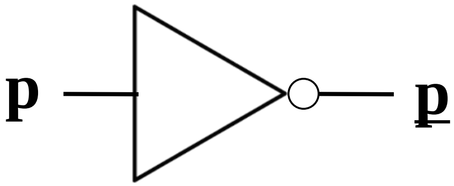


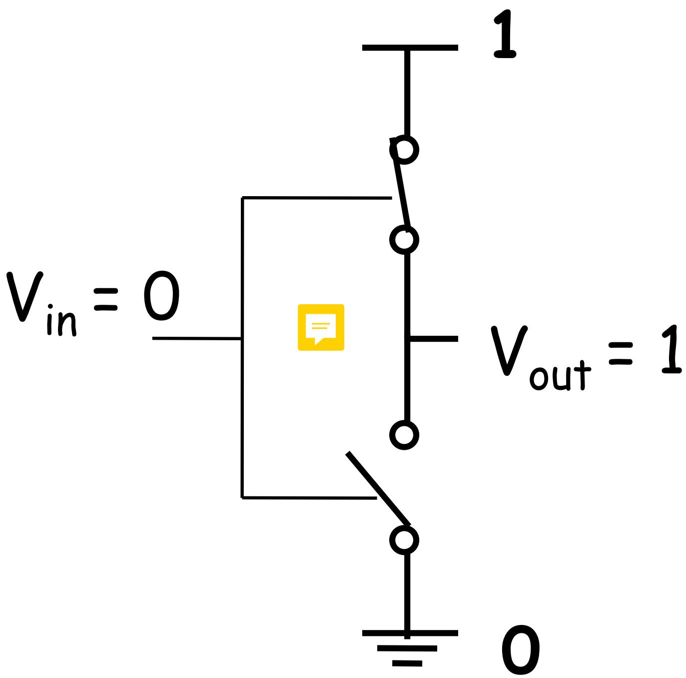


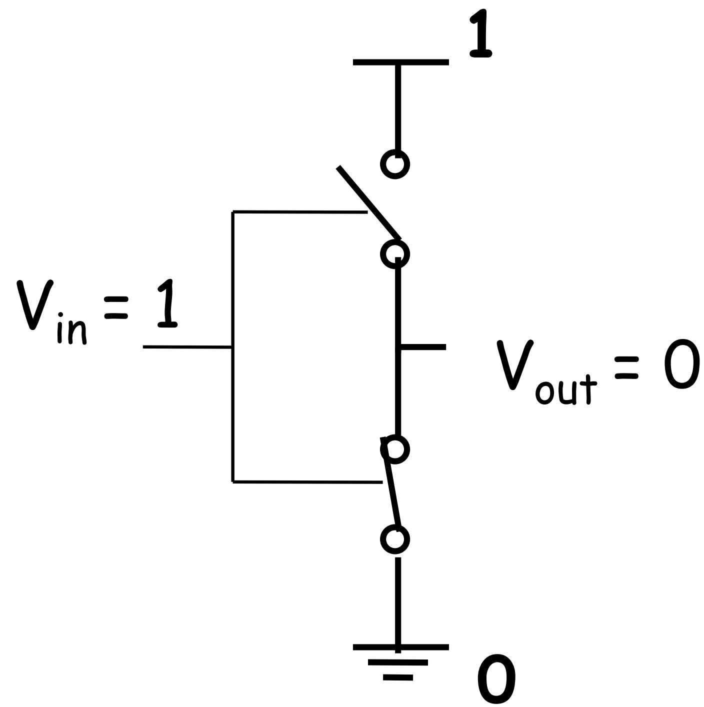
How to implement AND, OR, NOT

- Gate-level of abstraction
  (Digital logic design)
- Transistor-level of abstraction
  (전자공학)

NOT Gate (Inverter)



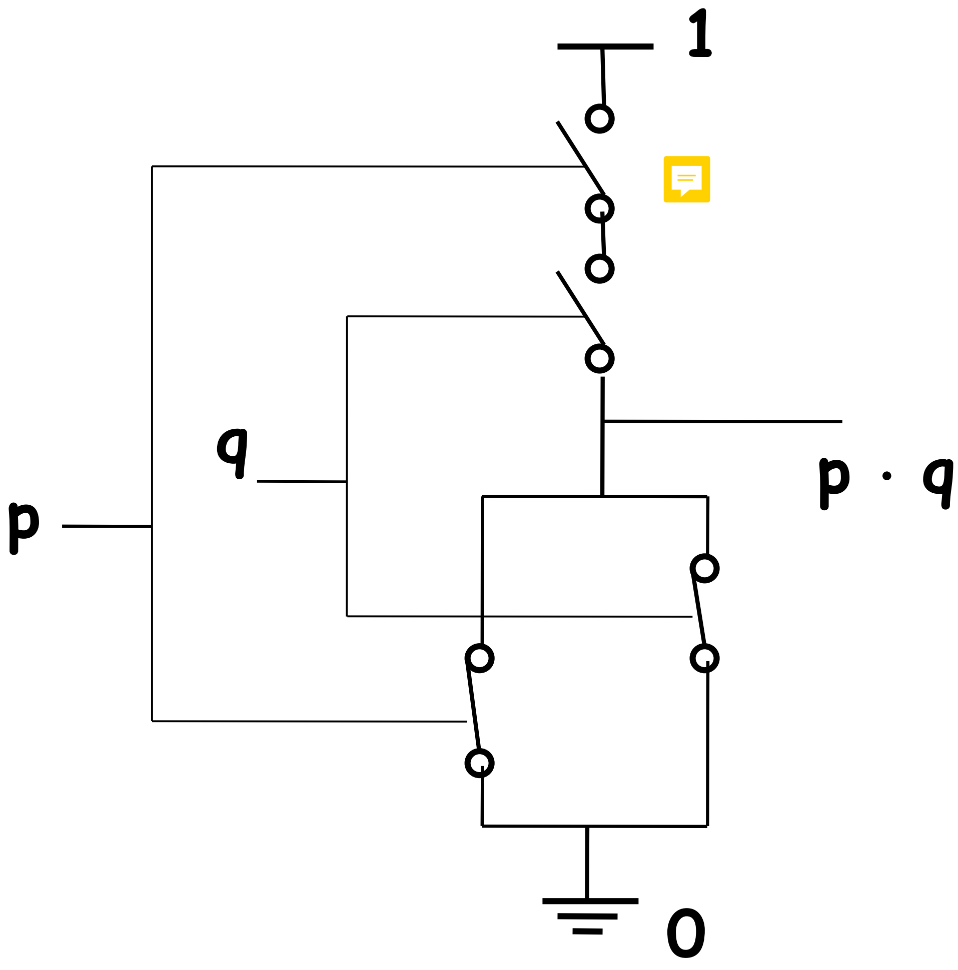


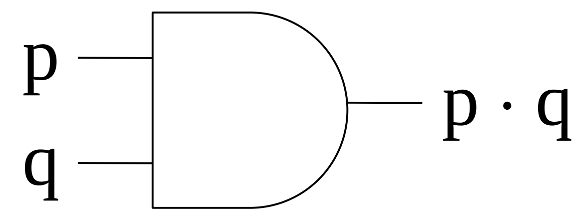


- High = 1.2^V = "1" = True, Low = 0^V = "0" = False

AND Gate

- When p = 0, q = 0

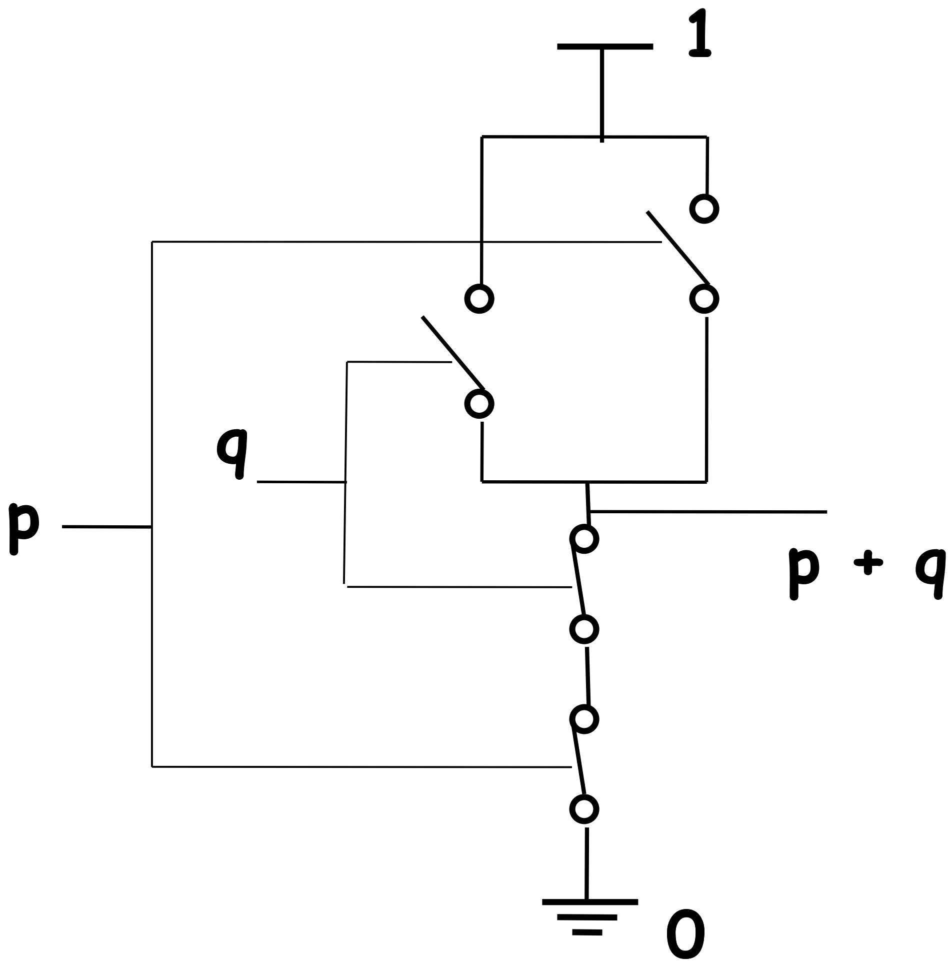


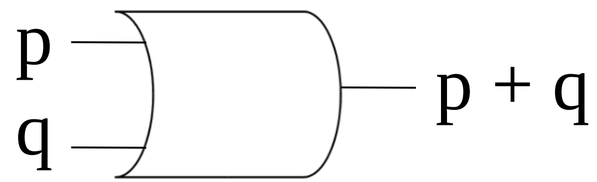


p	q	p · q
1	1	1
1	0	0
0	1	0
0	0	0

OR Gate

- When p = 0, q = 0





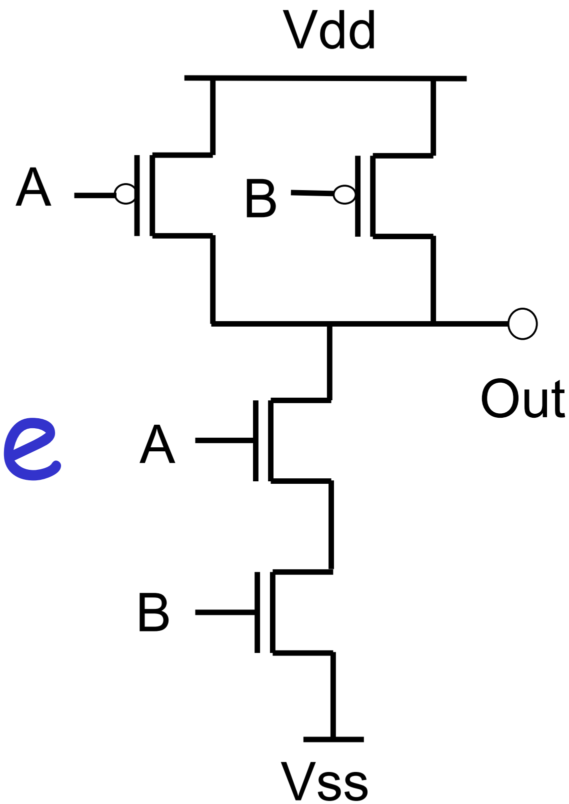
p	q	p + q
1	1	1
1	0	1
0	1	1
0	0	0

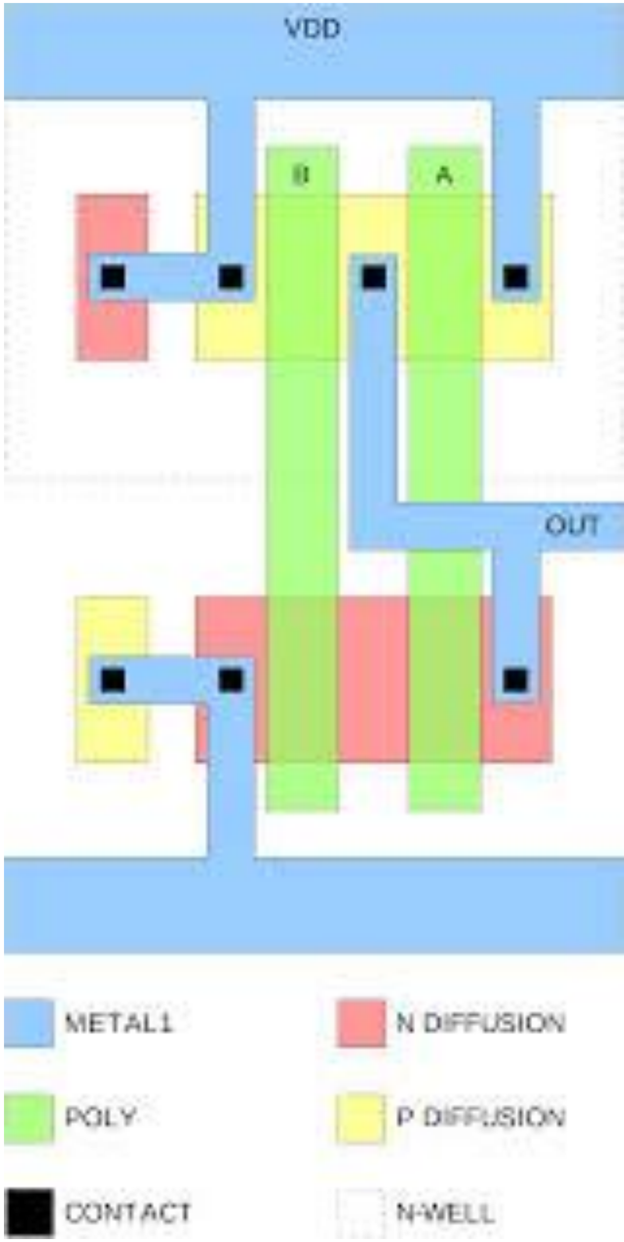
How to implement transistors

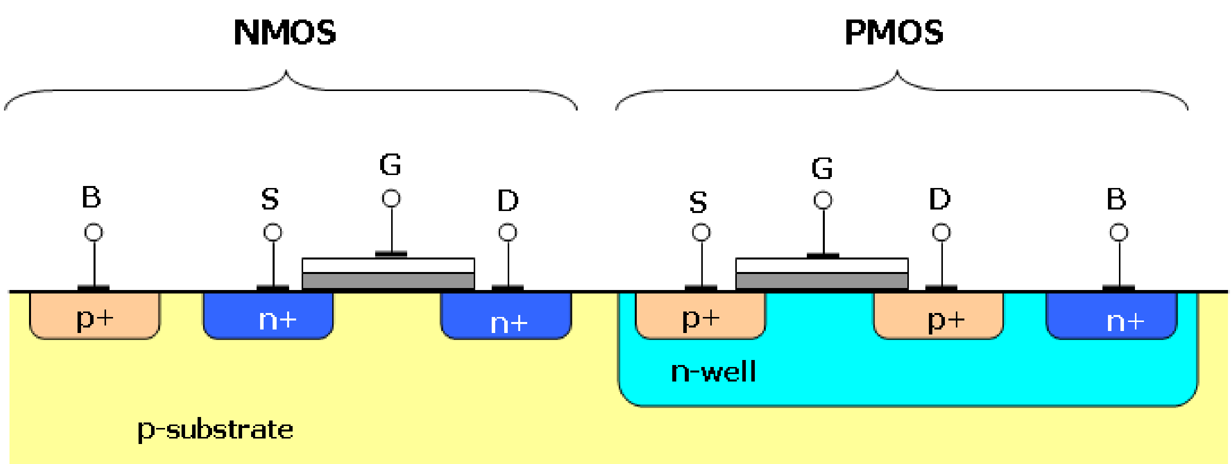
(transistor: abstraction of a complex thing)

(반도체 제조)

CMOS NAND Gate

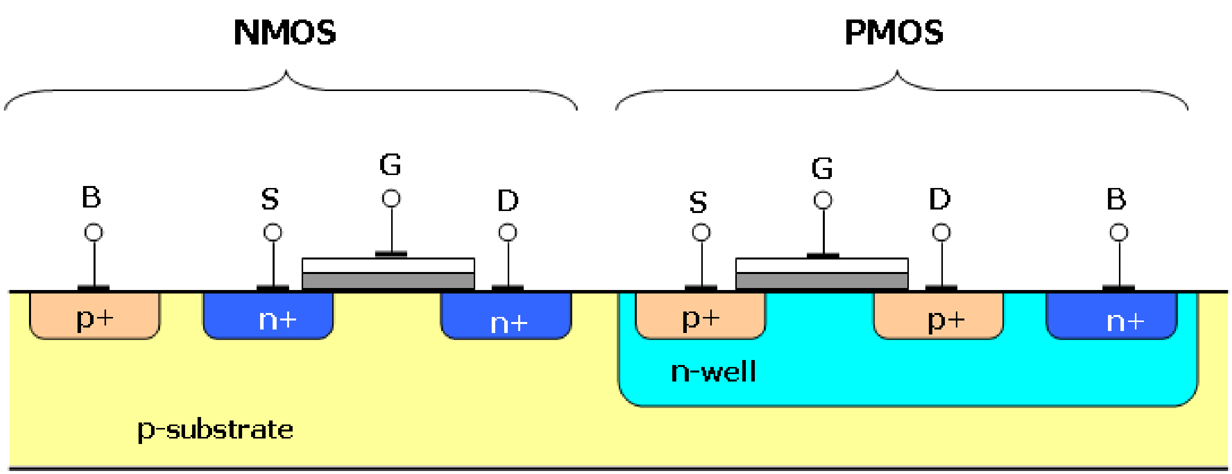


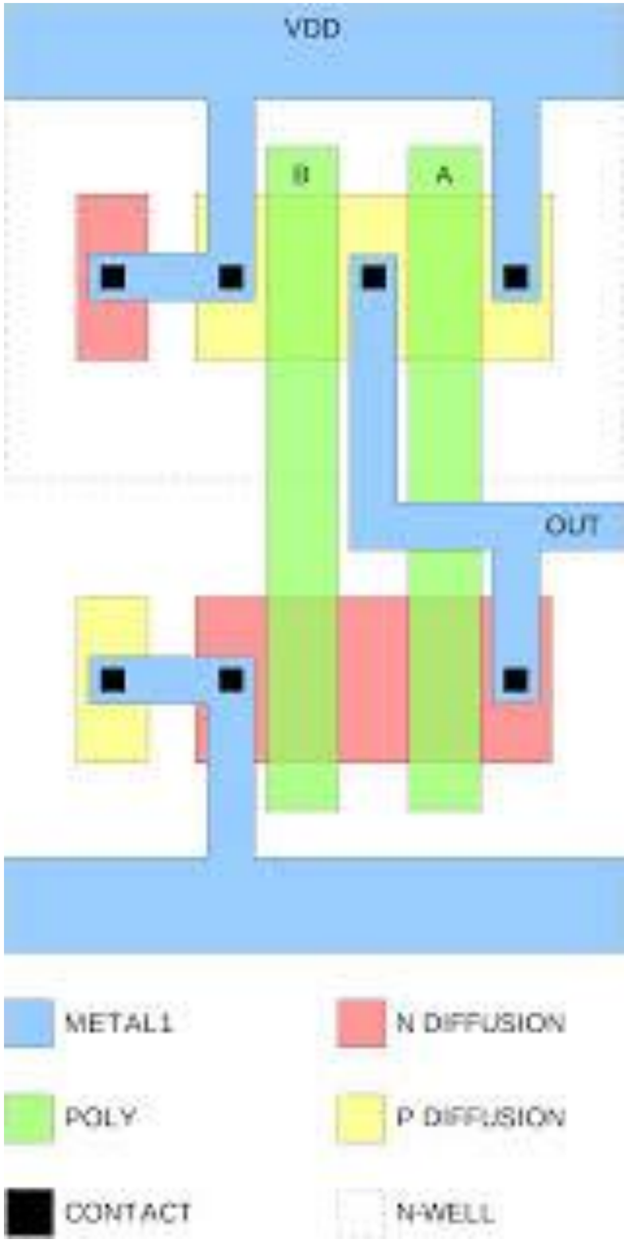




Fabrication;
반도체 제조 공정

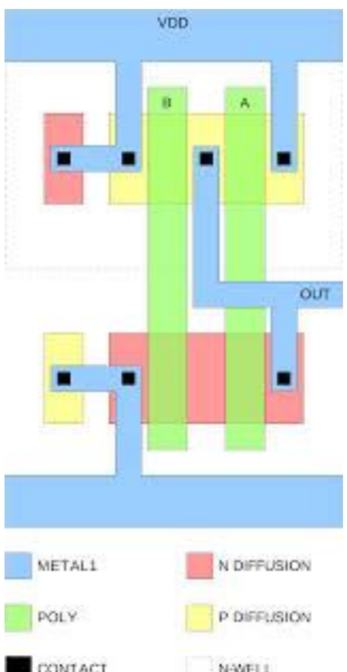
- ❑ 색깔별로 다른 물질, 기능
  - 각각 하나의 반도체 공정
- ❑ CPU: 20~40 공정





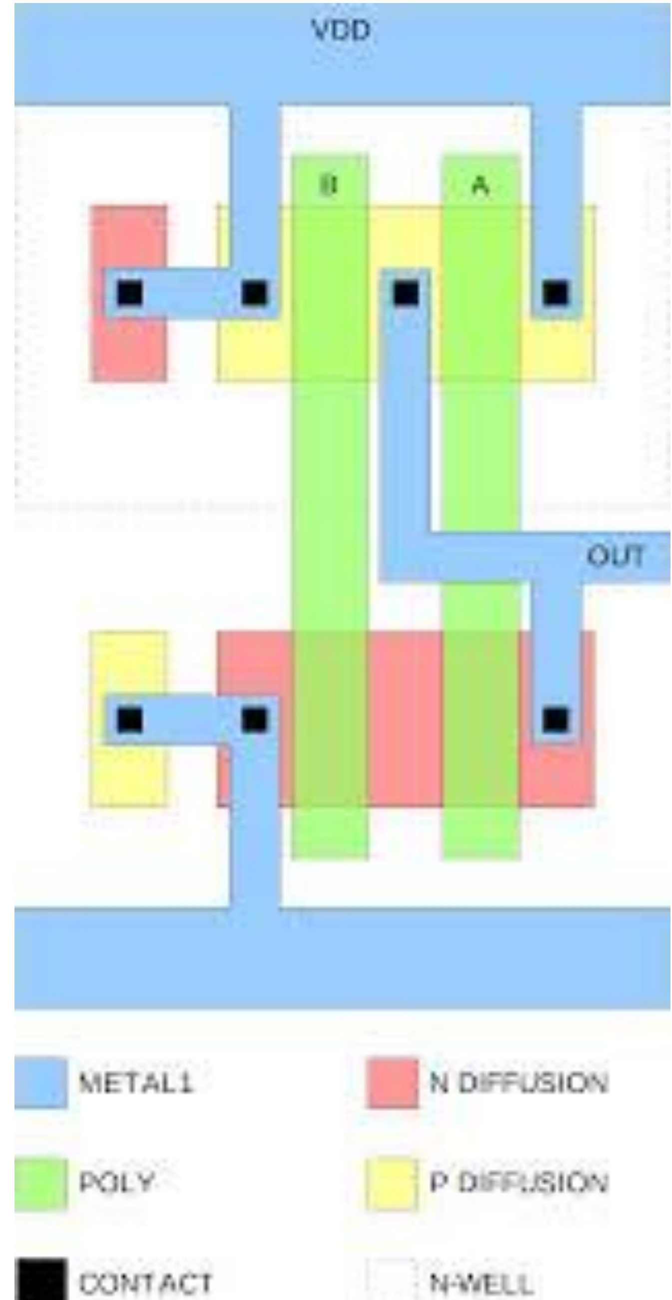
경쟁: Smaller, Faster

- Minimum feature size (최소선폭)
  - 속도, 크기 (집적도)



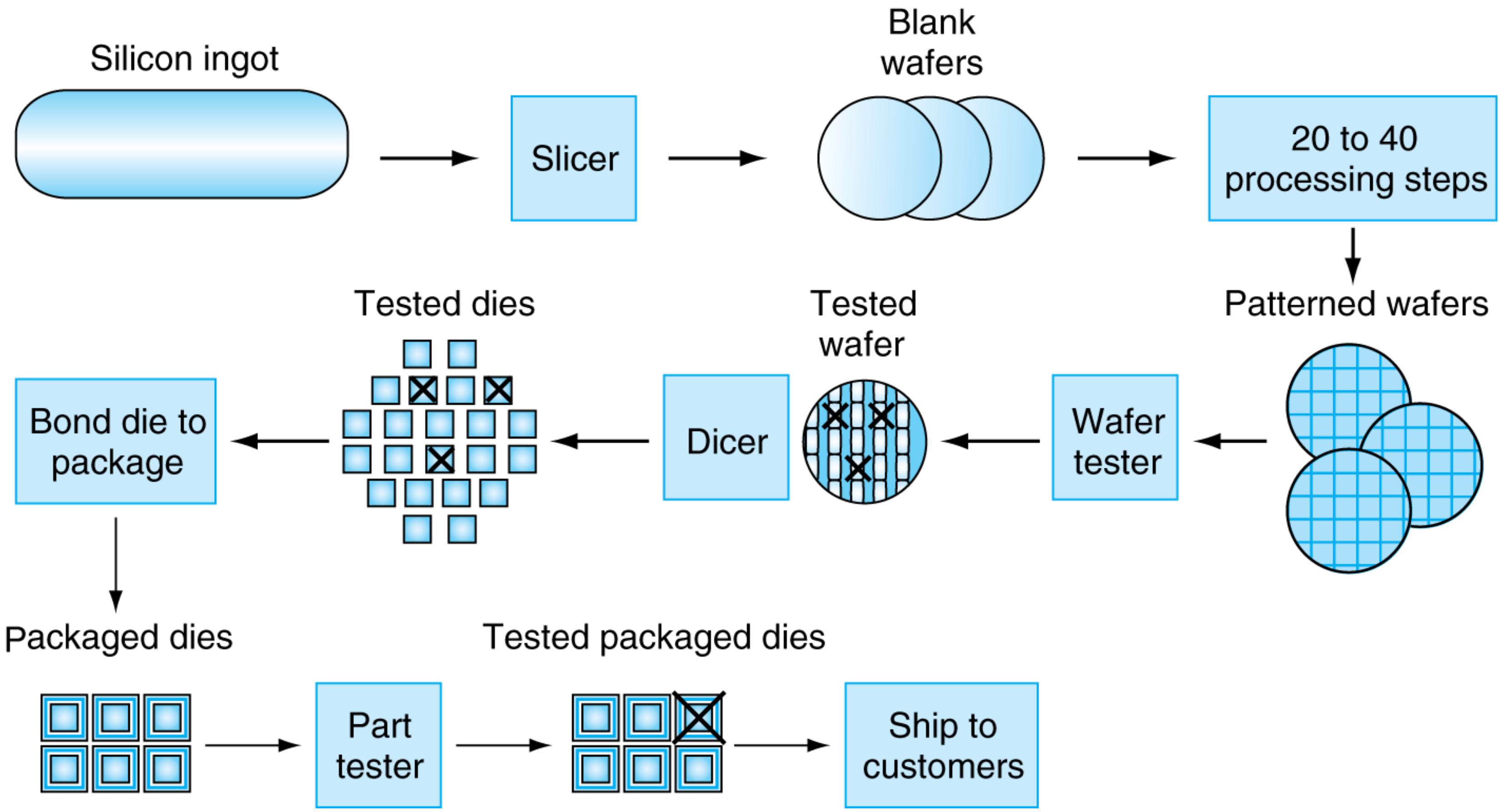
10 nm

20 nm



Manufacturing ICs

(Hennessy and Patterson slide, Computer Organization and Design, Morgan Kaufmann)



- Yield: proportion of working dies per wafer

Science and Engineering

(References: "Godel, Escher, Bach" by Hofstadter, "AI" by Winston)

- Levels of abstraction

Java Language

Machine Instructions

System Biology

Functional

Cell Biology

Gates

Molecular Biology

Transistors

Chemist

Semiconductor Physics

(Electron, orbit)

Atomic Physics

(Atomic nucleus)

Nuclear Physics

(Proton, neutron)

Abstractions in Software

(Primitive-Composition-Abstraction)

What is programming?

- Not syntax
- What we must know about

Case Study: C programming

C Programming Language

- "Small" language (c.f., C++ and Java)
  - Can be described in a small space, and learned quickly
  - Can understand and regularly use the entire language

- Can clearly see primitive-composition-abstraction

C (or High-Level) Programming

- What are the primitives?

  (basic building blocks or atoms)

  - Statements (like sentences in human writing)
    - "atoms" that have meanings

† Variables, constants, operators, expressions, data types

† Compilers translate statements into CPU instructions

C Statements

- Compilers support variety of statements for programmers
  - Variable declaration statements

```
int  a, b, c, d, i, j = 0; // statement end with ;
```

  - Assignment statements

```
a = 3;
```

  - Arithmetic and assignment statements

```
a = (b*3) – (c/d);
```

  - Conditional statements

```
if (i > 0)  x = x*1.1;              // if-else statement
    else  x = x*0.9;                   (indentation)
```

C Statements

- Loop statements

```
a = 0;                      // summation
for (i = 1; i < 5; i = i + 1)
    a = a + i;
```

- Compound statements

```
{ multiple statements }     // treat as single
```

- Function call statements

```
printf("hello, world!\n");  // call OS service
```

-
-
-

C Programming

- We have statements
  - Can write algebraic equation
  - Have English-like control structure
    - Can forget about machine-level details

- Are we ready to handle large software?
  - What if we put 1000 statements in the main function?
  - Need design paradigms to reduce complexity
  - How to perform <u>composition and abstraction</u>?
    - C provides functions

Small C Program - Function

```
#include <stdio.h>
int sum_from_to (int, int);    /* function declaration */
main()                         /* test summation function */
{   int  i;
    for (i = 0; i < 10; i++)
        printf("%d %d \n", i, sum_from_to (0,i));    // function call
}
int sum_from_to (int m, int n)   /* integer sum from m to n */
{
    int  i, sum = m;
    for (i = 0; i < (n - m); ++i)
        sum = sum + (++m);
    return sum;          }
```

Function: Abstraction Mechanism

- Why define functions?
  - Write once, call many times (from different locations)
    - Don't Repeat Yourself (DRY principle)
  - For composition and abstraction (using statements)
    - What is interface?
    - What is implementation?

Function: Abstraction Mechanism

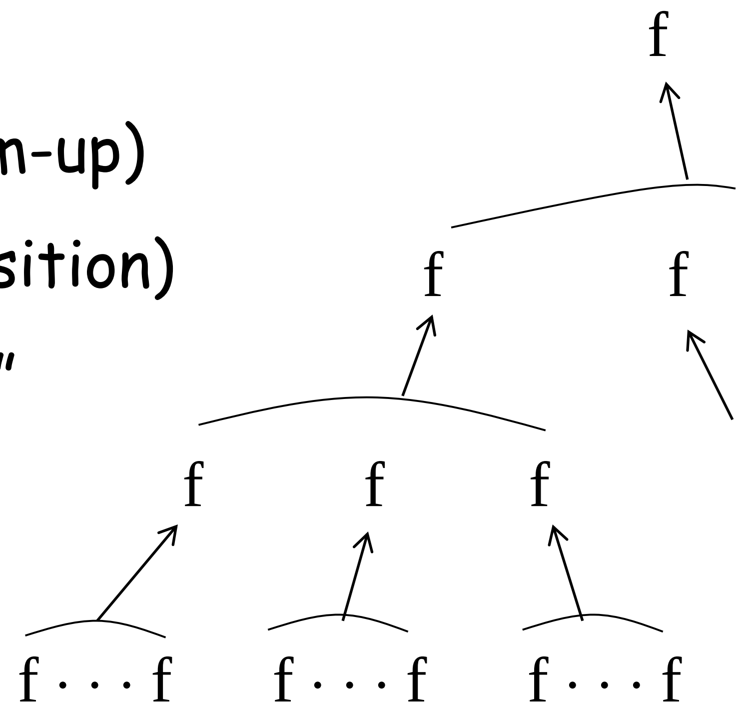
- Once we define a function, all users need to know is
  - Function interface: "int sum_from_to (int, int)"

    a = sum_from_to (2, 3); // function call statement

  - Don't have to know about implementation details
    - Function body: { … }
- sum_from_to function call
  - Look like a single abstract operation
    - Although it may do a lot of work
  - Become a statement (i.e., primitives)

Hierarchical Function Abstractions

- Hierarchical bottom-up function abstraction
  - Critical to deal with complexity
- Design perspective
  - Top-down (rather than bottom-up)
  - Modular design (i.e., decomposition)
  - Keep "dividing and conquering"
- Notion of program structure



Function: Abstraction Mechanism

- Function abstraction
  - Critical to deal with program largeness and complexity

- High-level programming languages
  - Must provide abstraction mechanisms

Primitive-Composition-Abstraction

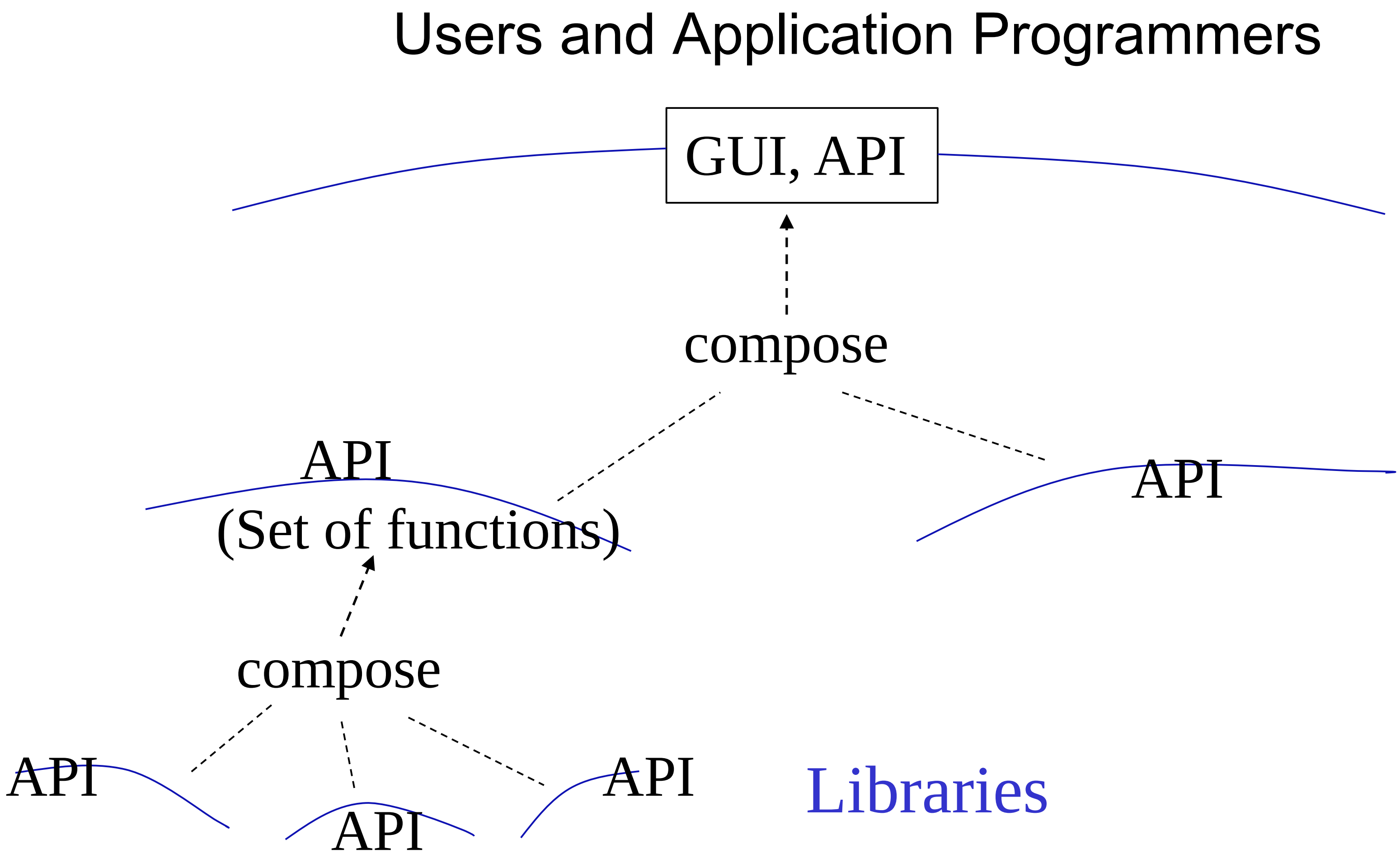
- Fundamental paradigm in high-level programming
  - Primitives: statements
  - Composition: build a function using statements
  - Abstraction
    - Given its interface, can use the function
    - Function becomes a primitive (or statement)
- Function: abstraction building mechanism
- What is high-level programming?
  - Hierarchically build abstractions
    - True in all engineering

Key Concepts in C programming

- Statements
- Functions
- What else?

Software Design

- Hierarchical abstractions



What is library (or API)?

- Library: collection of related functions
- Mathematics library
  - API (Application Programming Interface): 사용법
    - "math.h" ("#include <math.h>" in my code)

```
int power (int, int);
float sin (float);
float log (float);
float sqrt (float);
. . .
```

  - 구현 (또는 물건): compiled code (power.o, sin.o, log.o, ...)
    - Link with my code

What is library (or API)?

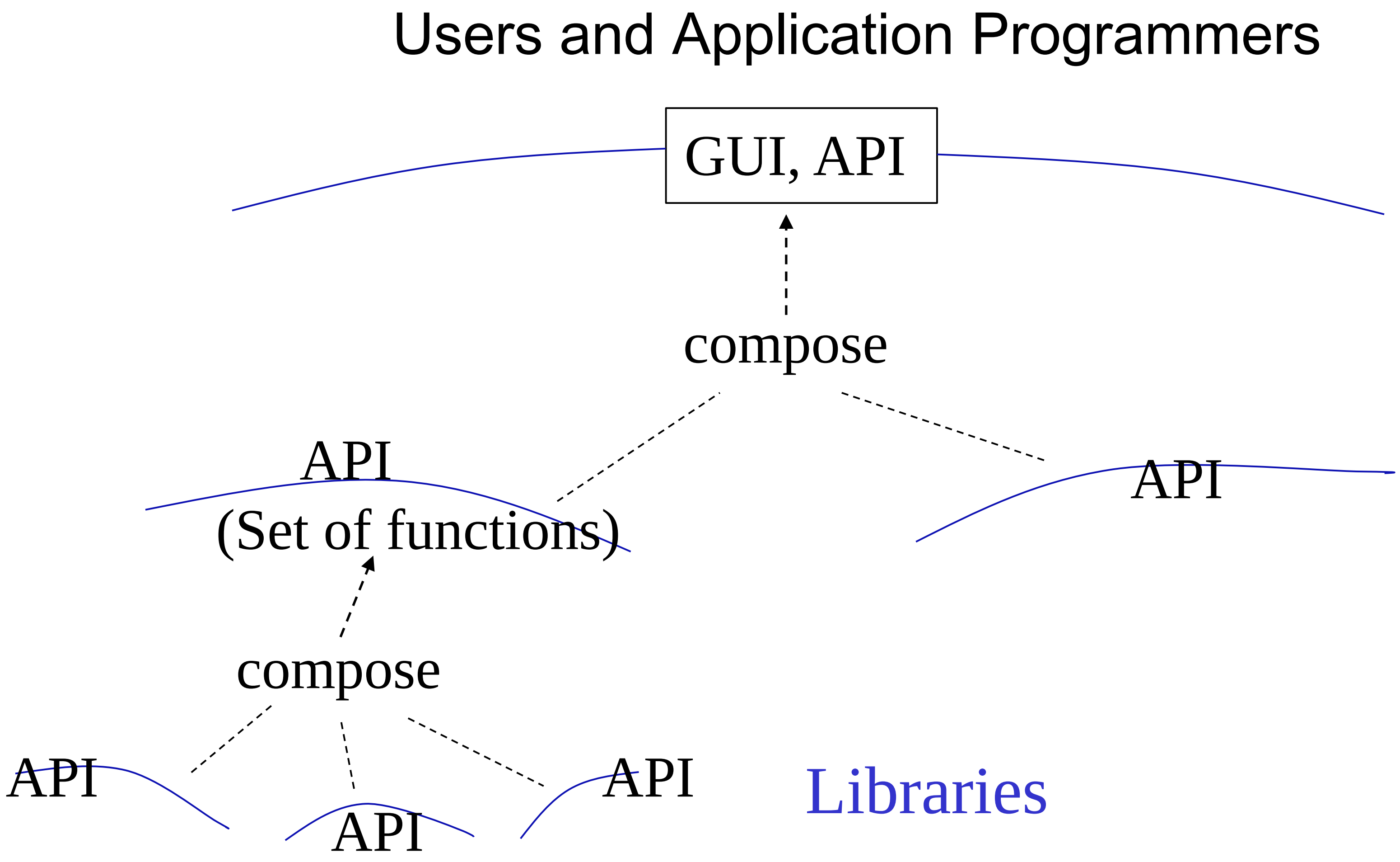
- Library: collection of related functions
- Graphics library
  - API (사용법): "graphics.h"

```
void initGraphics(int width, int height);
void drawImage(string filename, double x, double y);
void drawLine(double x0, double y0, double x1, double y1);
void drawRect(double x, double y, double width, double height);
void drawOval(double x, double y, double width, double height);
void setColor(string color);
. . .
```

  - 구현: compiled code

Software Design (반복)

- Hierarchical abstractions



Science and Engineering (반복)

(References: "Godel, Escher, Bach" by Hofstadter, "AI" by Winston)

- Levels of abstraction

Software Abstraction Layers

C Language

Machine Instructions

System Biology

Functional

Cell Biology

Gates

Molecular Biology

Transistors

Chemist

Semiconductor Physics

(Electron, orbit)

Atomic Physics

(Atomic nucleus)

Nuclear Physics

(Proton, neutron)

Fundamentals of C Programming

- Procedural programming paradigm
  - Functions vs. procedures
- Can you pick three critical concepts in C programming?
  - Statements (and single function C programs)
    - Art: 필요한 statement를 정교하게 완성
  - Functions (and single-file C programs)
    - Art: 우아한 function decomposition
  - Libraries (and multiple-file C programs)
    - Art: 성능 고려한 논리적인 API design

Software Architecture

- Software architecture (or program structure)
- What is it?
  - Set of key interfaces
    - Identification of modules
    - Their interfaces
  - Hierarchical: all the way down to lowest library
- Architects vs. programmers

† The same applies to hardware or any engineering area

Computer Architecture

- Meaning of "Architecture" in Computer Architecture
- Meaning of "A" in ISA (Instruction Set Architecture)
  - Most important interface in computers
  - Interface between hardware and software
- Issues in computer architecture (3대 수업 목표)
  - Fundamental concepts
  - Design of efficient interface (ISA)
  - Key implementation techniques (pipelining, cache)

Two Major Interfaces in CS (반복)

Developers

High-level language		C, C++, Java
	Compilers	
(executable) Machine-level language		Machine instructions (Core, PowerPC)
	Machines (CPUs)	

Key Concepts in C programming

- Statements, functions, libraries
- Abstractions from data perspective

  (지금까지는 abstractions in processing)

Processing vs. Data

- Large software uses complex data structures
- C: separate abstraction of data (and processing)
  - Primitive-composition-abstraction paradigm
    - Primitives
      - int, double/float, char
    - Composition and abstraction
      - struct, array, pointer
  - Hierarchical/recursive abstractions of data

Composition and Abstraction

- struct, array, pointer

```
struct  Student_info {
        char  name[20];
        int  age;
        …  ;
};
struct Student_info  Hong;
struct Student_info  Myclass[50];
struct Student_info  *precord;
```

- Hierarchical/recursive abstractions of data

Size Limit in C programming

- Statements, functions, libraries
- Abstractions from data perspective
- Library use in C

C Programming

Client code

Library code

include header;
allocate ds1;
fn1(&ds1,);

header

ds1;
ds2;
fn1(ds1 *,);
fn2(,);

compiled code

- Data sharing limits project size
  - What is a software project failure?

lib1.h (interface)

```c
struct A { … };
void fn1 (struct A *pA, …);
…
```

lib1.c (implementation)

```c
struct A { … };
struct B { … };
…

void fn1 (struct A *pA, …) { … }
void fn2 (struct B *pB, …) { … }
…
```

Client code

```c
#include "lib1.h"
struct  A *pA = (struct A *) malloc (sizeof(struct A));
fn1 (pA, …);                    // call library function
```

Million Lines of Source Code

Developers

Complexity/
Modularity &
Abstraction

Many design steps
(manual)
to fill semantic gap

High-level
language

C, C++,
Java

Compiler

(executable)
Machine-
level
language

ISA
(Pentium,
PowerPC)

Machine (CPU)

- Most intellectually-challenging tasks

C Language

- Designed for and implemented on UNIX OS on PDP-11
  - D. Ritchie
  - UNIX kernel, C compiler, all UNIX applications
- Since then, C spreads far beyond its origin
  - Popular general-purpose language
    - Kernels, compilers, embedded systems, applications
- Influence many later programming languages
- Standardization
  - K&R C, ANSI C (C89), C99, C11, Embedded C

C Language

- Programming in early 1970s
  - Replace assembly in system programming
    - Compact and fast code is the goal
  - Programmers are computer experts (unsafe)
  - Software size: up to 100K SLOC (limit project size)
- C is a relatively low-level language (high-level assembly)
  - Map language constructs efficiently to CPU instructions
- Terse, small, efficient, relatively unsafe
  - If C program not run efficiently, it's due to design

Why Another Language (C++) ?

- C: limit project size and productivity
  - Library use in C is inconvenient
  - Procedural programming: functions & data, still low level
- Software crisis in 1980s and 1990s
- Solution: object-oriented approach
  - C++: productivity tools
  - Libraries based on objects

OO Programming Paradigm

Functional Programming Paradigm

Object-Oriented Programming

- Additional mechanism for abstraction: object
  - Combined abstraction of data and processing
    - Larger than function

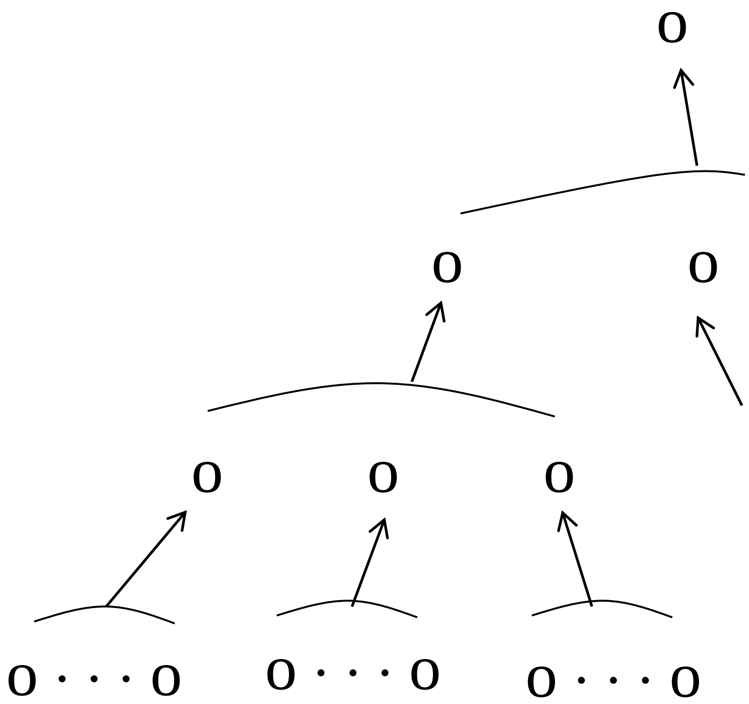
Users access public I/F only
(not share internal states)

ds1
ds2
…
fn1(ds1 *,)
fn2(,)
…

- With data and functions, can model real-life objects

Object-Oriented Programming

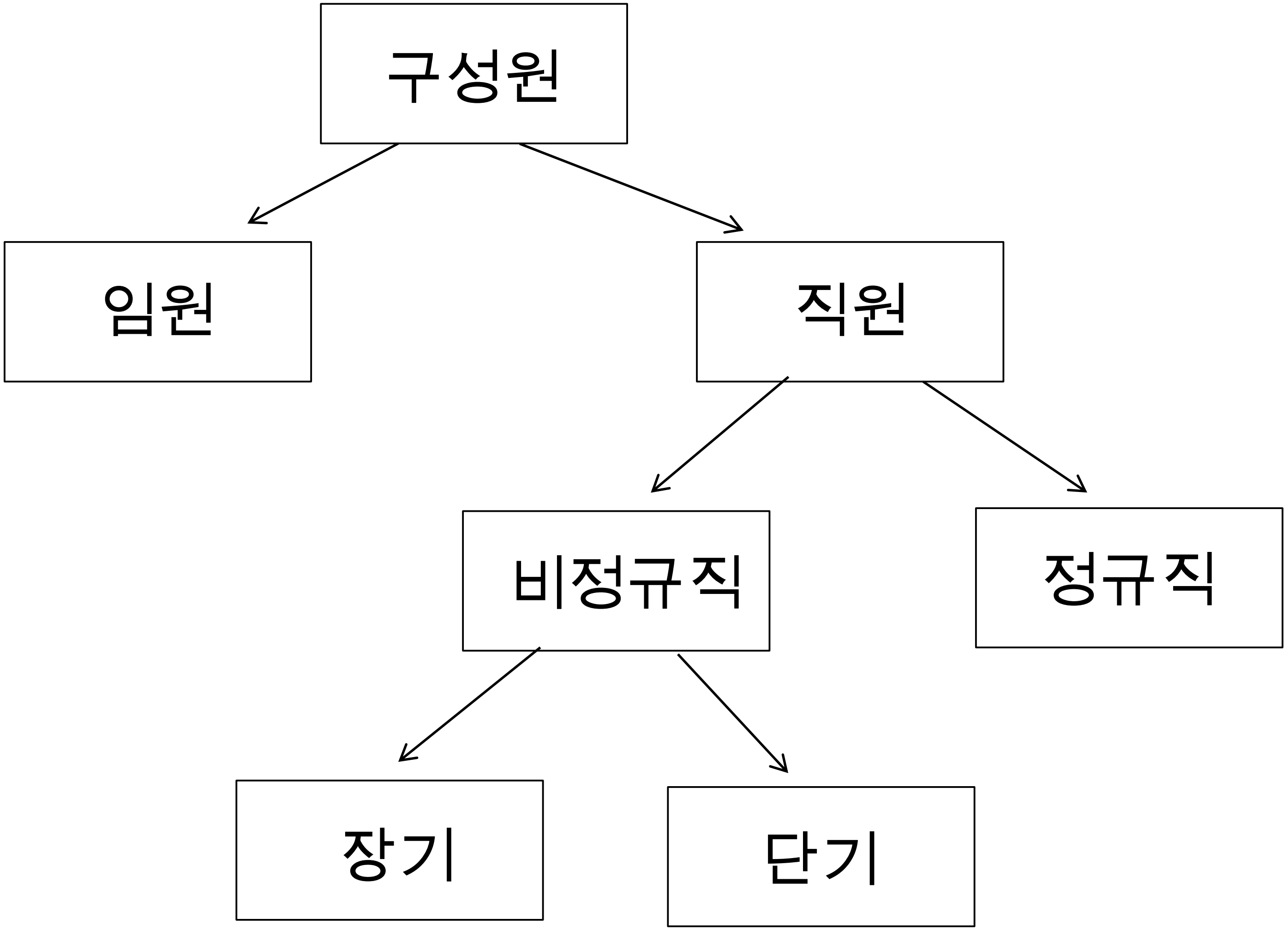
- Hierarchical abstractions with object libraries



Object-Oriented Programming

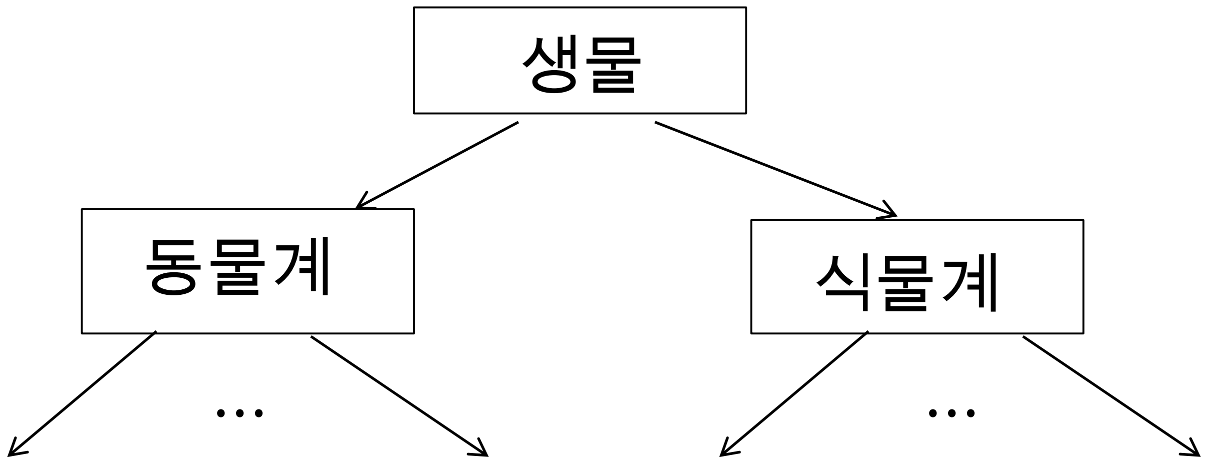
- Object: state and behavior (data and functions)
  - Can model real-life objects (employee, bank account)
  - Why important? (what do you do in C programming?)
    - Cyber space is much like physical universe
    - Solve problems in problem space
- OO design: objects and their interactions
- How to model all relevant real-life objects?
  - Class hierarchy and inheritance
    - Upcasting, "is-a" relationship, dynamic binding

Class Hierarchy (Art)

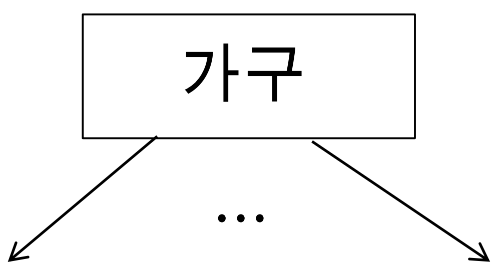


Classification and Inheritance

- Species: 계, 문, 강, 목, 과, 속, 종

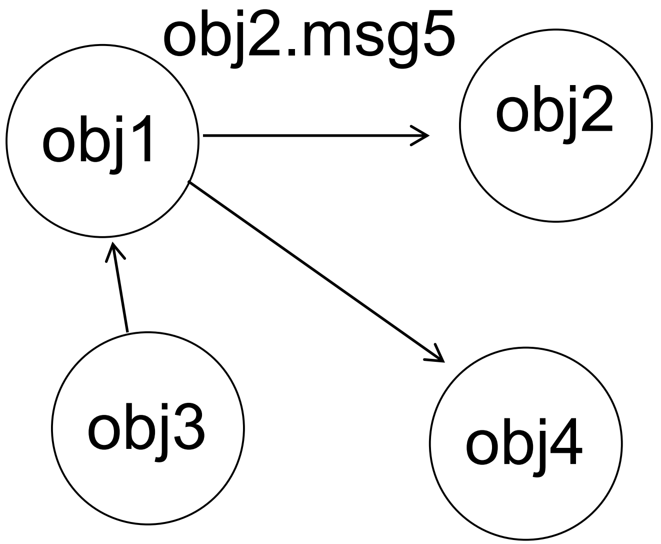


- 비생물



Programming Paradigms

- Paradigms to use: depend on nature of problems
  - C: sequential processing
  - OOP: a bunch of objects sending/receiving messages
    - Flavor of distributed processing
  - Functional
  - Logic



Functional Programming Paradigm

- Functions and procedures in C (procedural programming)
- 이들을 다른 의미로 사용하면
  - Procedures
    - Side-effects (we rely on side-effects)
      - State changes other than return values
        - "printf" function, memory write (assignment)
  - Functions (functional programming)
    - Mathematical (or pure) functions
      - Arguments and return values only $(f = \sin\theta)$
    - Side-effect free (referential transparency)
- 적합한 응용에 적용하면 생산성 높음

Functional Programming (참고사항)

- Based on λ-calculus by Church (1930s)
  - All computations can be modeled with pure functions
- Functions as first-class citizens
  - Higher-order functions
    - Functions as arguments, functions as return value
  - Nested functions
  - Non-local variables and closures
  - Assigning functions to variables
- Lisp, Scheme, OCamel, Haskell, …
  - λ-expressions in Python, C++, Java, …

Big Ideas of Computing

- Abstraction
- Programming paradigms
  - Procedural
  - Object-oriented
  - Functional
  - Logic
- Recursion
- Concurrency and transactions
- Higher-order functions
- Algorithms (complexity), CS 전공교과목, …

OS Abstractions

- 어떤 구체적인 문제들을 풀었나?

- 어떤 solution 들을 만들어 내었나?

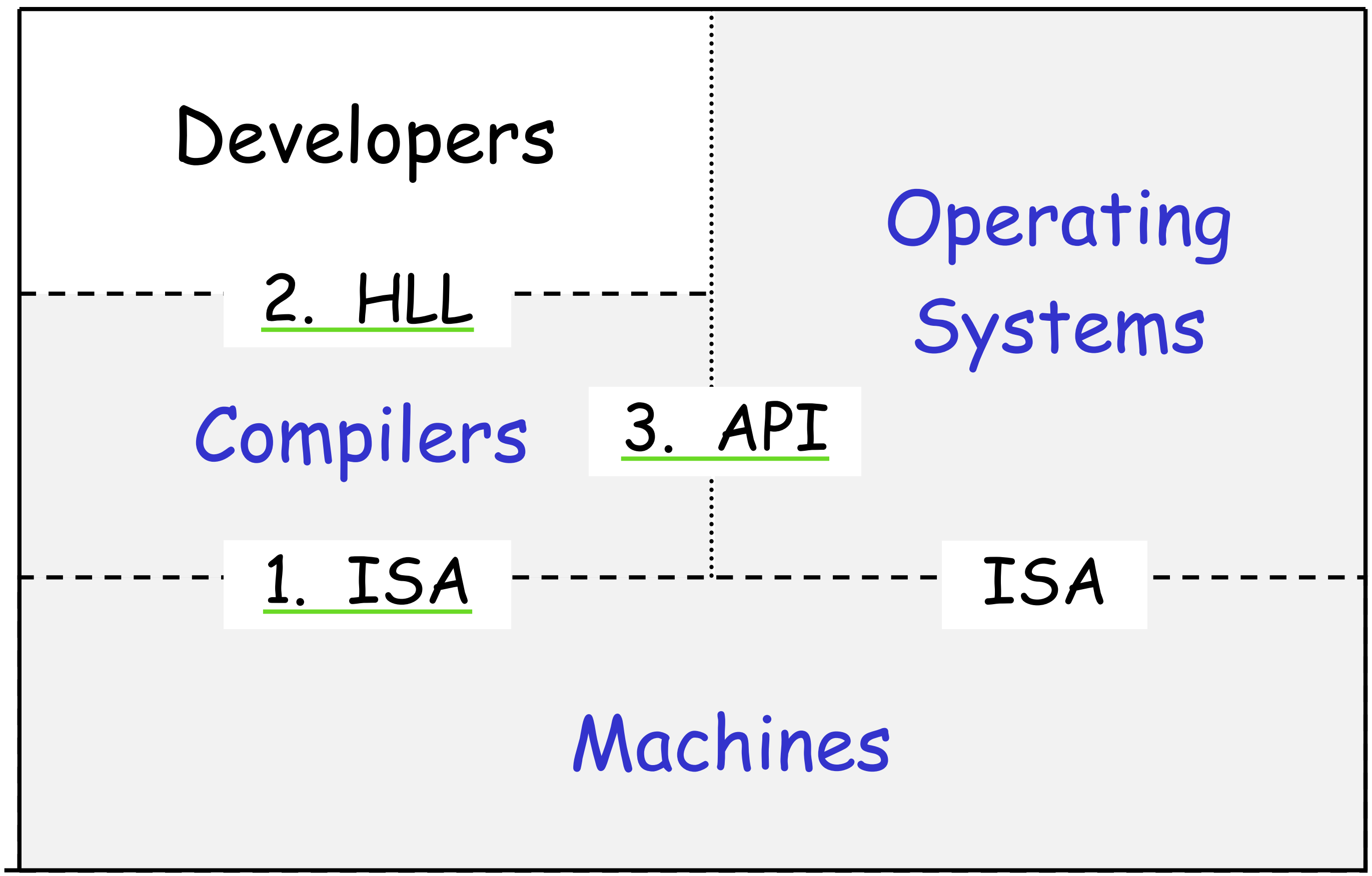
(Concepts, design patterns, algorithms)

- How to improve?

- What are new/unsolved problems?

Three Major Interfaces in CS

- Three key products and their services
  - Three core subjects in computer systems



What is OS? (Abstraction Perspective)

Users

GUI

Utilities Applications

API Library functions

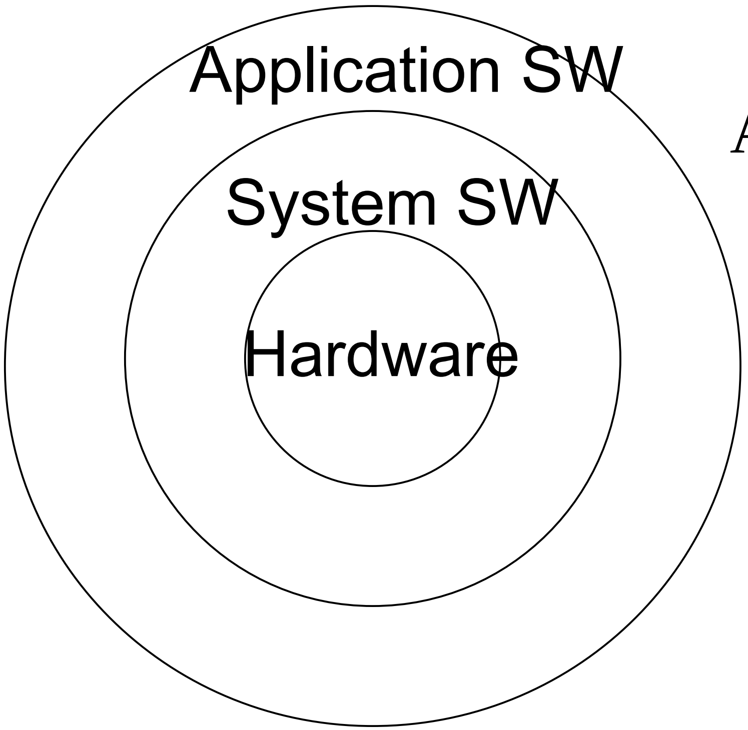
Process (execution)	File (storage)	TCP/IP, peripherals (connect)	Kernel
Processor	Memory	I/O	Hardware

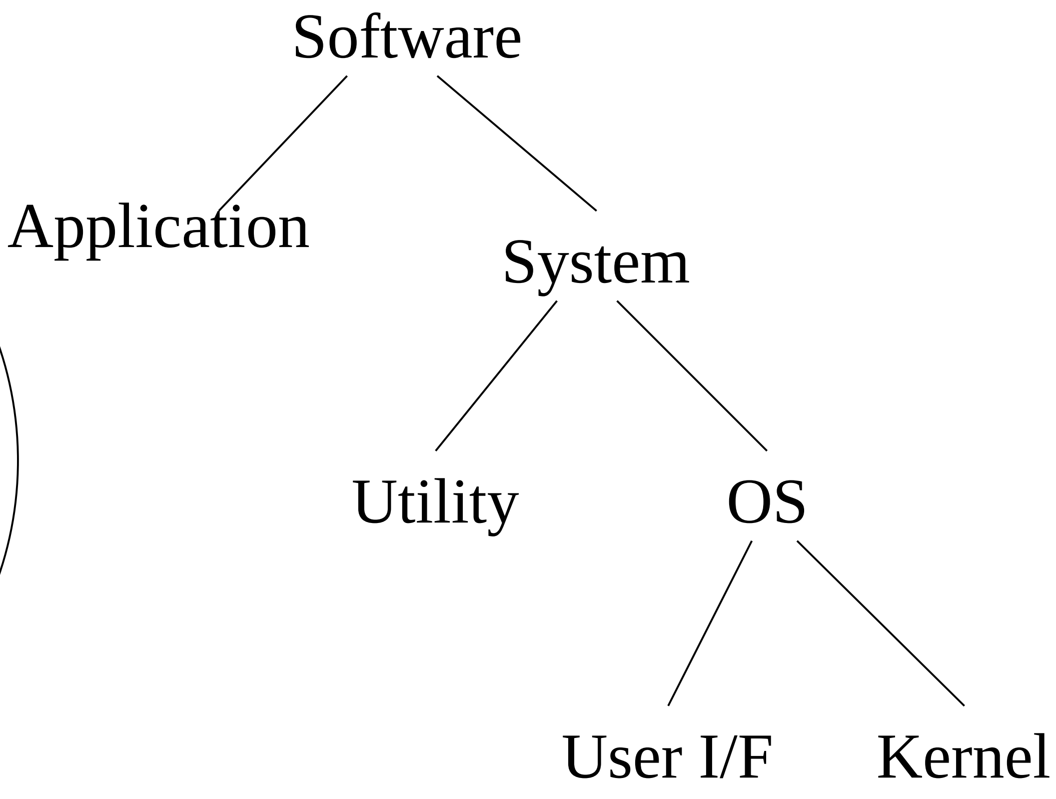
What is OS?

- Make hardware easy to use by providing abstractions
  - Processor (program execution)
    - process_create(), process_kill(), …
  - Memory (storage)
    - file_copy(), delete_folder(), file_rename(), …
  - I/O (connectivity)
    - Socket("naver.com", 80), monitor_write(), …
- GUI, utilities (common functions for all users)
- 공유자원의 사용관리 및 보호

What is OS?

- Map to previous figure





Summary

- Abstractions in software
  - Primitive-composition-abstraction paradigm
  - Procedural programming paradigm
    - C language: statements, functions, libraries, data
  - Object-oriented programming paradigm
  - Functional programming paradigm

Homework #2 (see Class Homepage)

1) Write a summary report about the materials discussed in Topics 0-1 and 0-2 (at least 5 pages of detailed report)

   - 문장으로 써도 좋고 파워포인트 형태의 개조식 정리도 좋음

2) Discuss a real engineering example of the (hierarchical) primitives-composition-abstraction paradigm

   - Book writing and building construction are possible examples; you may try to find other interesting examples based on your imagination

❑ Submit electronically to Blackboard

❖ Study lecture notes - you should be able to give a lecture with them

Class Topics (클래스 홈페이지 참조)

- Part 1: Fundamental concepts and principles
  1) Invention of computers and digital logic design
  2) Abstractions to deal with complexity
  3) Data (versus code)
  4) Machines called computers
  5) Underlying technology and evolution since 1945
- Part 2: 빠른 컴퓨터를 위한 설계 (ISA design)
- Part 3: 빠른 컴퓨터를 위한 구현 (pipelining, cache)