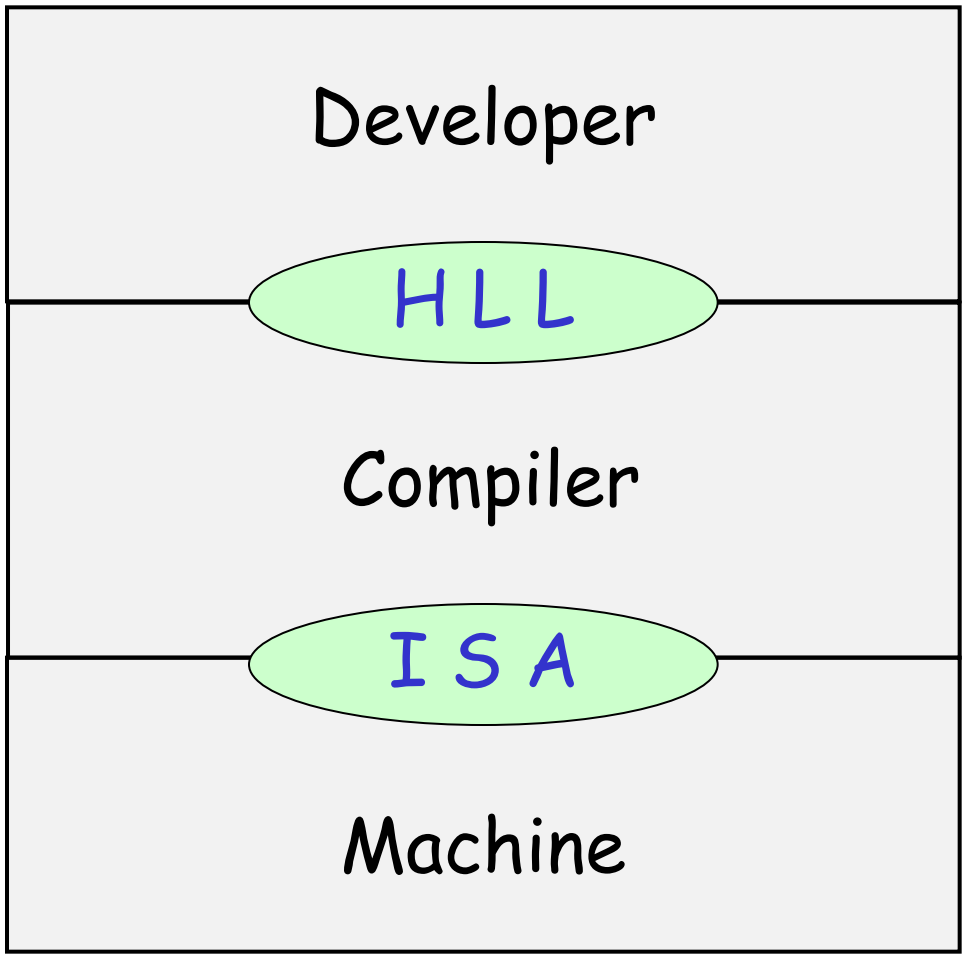
Chapter 2

Instructions: Language of the Computer

Part 3:

- Run C programs
- Procedure calls

Program Execution



Program Execution

- ISA perspective
  - Instruction-level (at hardware/software interface)
  - Processor internal-level
    - Fetch, decode, execute
- HLL perspective
  - Statement-level statement 하나씩 실행
  - Procedure -level procedure call/return 실행
    - Another important level of abstraction
    - † Function: a single abstract operation
    - † C programming: procedure-oriented SW design

Leaf Procedure Example

```
int  leaf_example (int g, int h, int i, int j)
{
    int f;

    f = (g + h) - (i + j);
    return  f;
}
```

- What is leaf procedure? 제일 끝 child
- Caller versus callee
  호출하는 함수 호출받는 함수(leaf는 callee만 가능)

Supporting Leaf Procedures

- Work to do
  1. Place parameters so that called procedure can access them
  2. Transfer control (change PC – jump and link)
  3. Acquire needed resources
  4. Perform desired task
  5. Place results so that calling procedure can access them
  6. Return control to caller (change PC – jump register)

Supporting Leaf Procedures

- Issues
  - How to place parameters, return values
  - How to specify return address
    - Returning to right instruction
  - Coordination of register usage (caller and callee)
- Registers are fastest: use them if possible
  - $a0 - $a3: four argument registers to pass parameter
    benchmark 결과 4개면 충분하다.
  - $v0 - $v1: two registers to return values
    double은 64bit이기 때문에 register 2개 필요
  - $ra: register to store return address

MIPS Register Convention

preserved on call	Name	Register number	Usage
N/A	$zero	0	the constant value 0
no	$v0-$v1	2-3	values for results and expression evaluation
no	$a0-$a3	4-7	arguments
no	$t0-$t7	8-15	temporaries
yes	$s0-$s7	16-23	saved
no	$t8-$t9	24-25	more temporaries
yes	$gp	28	global pointer
yes	$sp	29	stack pointer
yes	$fp	30	frame pointer
N/A	$ra	31	return address
no	$at	1	reserved for the assembler
N/A	$k0-$k1	26-27	reserved for the operating system

To Appear Soon

- Separation of working registers into $s and $t
  - Coordination of register usage
    - What if need more than 18 working registers?
- $gp, $sp, $fp
  - Issue of memory allocation

Supporting Leaf Procedures

- Additional support: jump and link (jal) instruction
  - Caller place parameters in $a0 - $a3
  - jal ProcedureAddress
    - Save PC+4 in register $ra
  - Callee need registers to work; save them in stack
  - Callee does the work, place results in $v0 - $v1
  - Callee restore registers from stack
  - jr $ra

† What if there are more than four parameters?

† What if want to return more than one item?

- Call by reference, global variables and spaghetti code

Leaf Example

```
int  leaf_example (int g, int h, int i, int j)
{
    int f;

    f = (g + h) - (i + j);
    return  f;
}
```

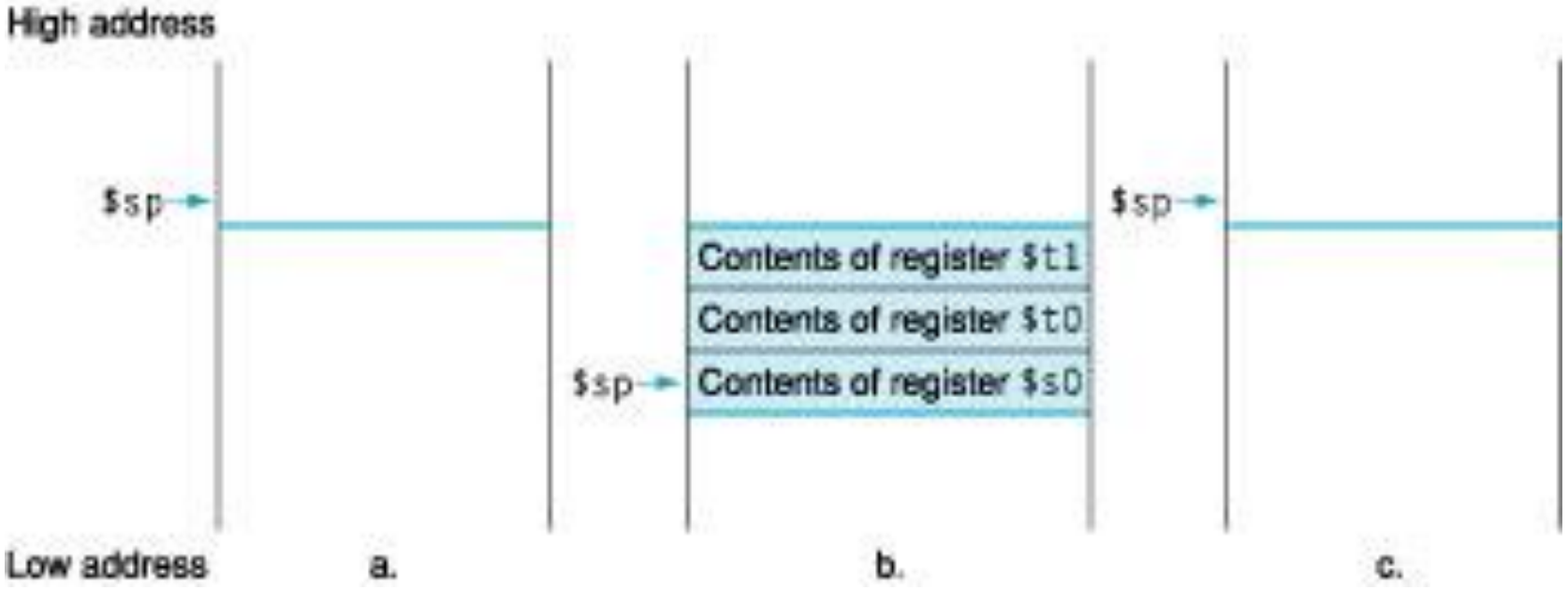
Leaf Example 실제 구현x

- g, h, i, j: $a0, $a1, $a2, $a3 f: $s0

```
addi $sp, $sp, -12      # save $t1, t0, s0 in stack
sw $t1, 8($sp)
sw $t0, 4($sp)
sw $s0, 0($sp)
add $t0, $a0, $a1       # calculate f
add $t1, $a2, $a3
sub $s0, $t0, $t1
add $v0, $s0, $zero     # set return register
lw, $s0, 0($sp)         # restore old registers
lw, $t0, 4($sp)
lw, $t1, 8($sp)
addi $sp, $sp, 12
jr $ra                  # jump back to caller
```

```
add a0,a0,a1
add a2,a2,a3
add r0,a0,a2
jr ra
```

Stack (before/after procedure call)



† What is stack?

String Copy Example

- C code (naïve):
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  - Addresses of x, y in $a0, $a1
  - i in $s0

String Copy Example

- MIPS code:

```
strcpy:
    addi $sp, $sp, -4       # adjust stack for 1 item
    sw   $s0, 0($sp)        # save $s0
    add  $s0, $zero, $zero  # i = 0
L1: add  $t1, $s0, $a1      # addr of y[i] in $t1
    lbu  $t2, 0($t1)        # $t2 = y[i]
    add  $t3, $s0, $a0      # addr of x[i] in $t3
    sb   $t2, 0($t3)        # x[i] = y[i]
    beq  $t2, $zero, L2     # exit loop if y[i] == 0
    addi $s0, $s0, 1        # i = i + 1
    j    L1                 # next iteration of loop
L2: lw   $s0, 0($sp)        # restore saved $s0
    addi $sp, $sp, 4        # pop 1 item from stack
    jr   $ra                # and return
```

Nested Procedure Call

- How to deal with non-leaf procedures
  - Extreme case: recursion

```
int  fact (int n)
{
    if (n < 1)  return (1);
        else  return ( n * fact (n – 1) );
}
```

procedure call 느리다
factorial 계산에서 recursion 은 비효율적
-> for문이 적절하다.
hanoi tower의 경우 recursion을 쓰지 않으면 code가 길어짐!

Recursion

- f(3) = 3 * f(2)

  f(2) = 2 * f(1)

  f(1) = 1 * f(0)

  f(0) = 1

- Recursive solution
  - Sequentially reduce computation
  - Specify terminal condition
- Programs or CPU not tell self-call from calling others
- Performance and elegance
  - Iteration, tail-recursion

Nested Procedure Call

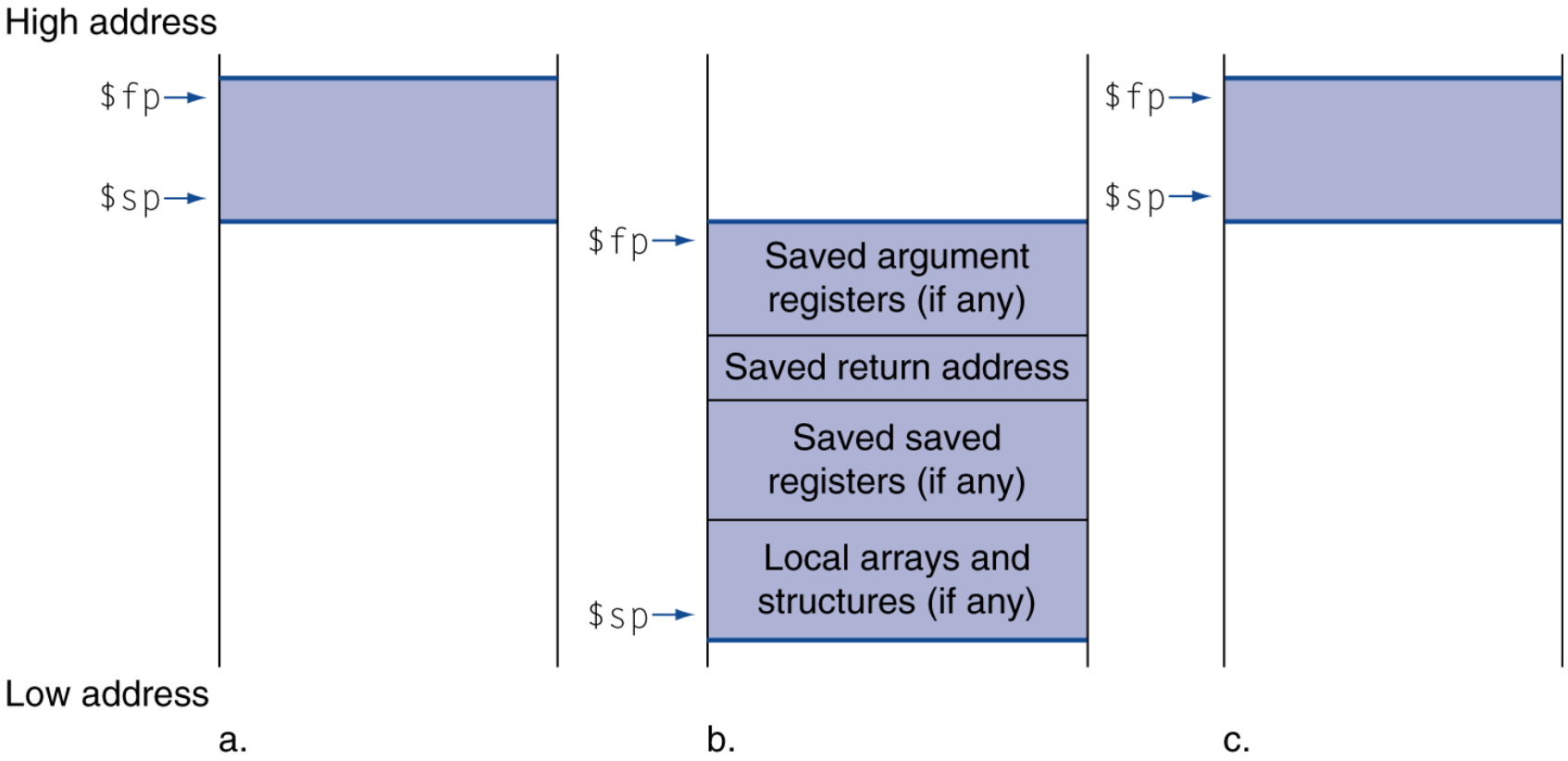
- How to deal with non-leaf procedures
  - Extreme case: recursion
- How to protect $a, $ra
- How to protect temporary register data
  - Save them in stack before use (i.e., procedure call)
  - Restore from the stack after the call

Runtime Stack

1. 해당 procedure가 call 되면 local variable이 만들어짐
2. 해당 procedure가 return되면 local variable이 없어짐

-> 다른 프로그램이 알 수 없다. (local variable의 위치 예측 불가능 (runtime일때만 알 수 있다.)

- Procedure frame, activation record (push and pop frame)



More on Runtime Stack

메모리 사용 효율적 - 필요할때만 만들기 때문에 미리 할당하면 필요한 메모리가 너무 많고 특정 시점에서 쓰이는 것은 적다.`

- Local variables are automatic variables
  - Dynamic allocation, but allocation/deallocation in sync with procedure call and return
    - † Heap – fully dynamic memory allocation
- Why use stack for storing local variables?
  - Storage efficiency, recursion
- Compiler access local variables: $fp + offset
  - Displacement addressing mode: ld $t0, 32($fp)
- Stack (procedure call) trace debug할때 쓰임(stack의 상태를 보고 분석)
  - Problem diagnosis (core dump, patch, rediscoveries)

crash가 일어날때 stack trace를 할 수 있게 해주는 기법

Recursion: fact (n)

```
Fact:   addi $sp, $sp, -8       # save $ra, $a0 in stack
        sw $ra, 4($sp)
        sw $a0, 0($sp)
            slti $t0, $a0, 1    # test if n < 1
            beq $t0, $zero, L1
                addi $v0, $zero, 1      #  return 1 if n < 1
                addi $sp, $sp, 8
                jr $ra
    L1:             addi $a0, $a0, -1   # recursive call:  fact(n-1) if n >= 1
                    jal fact
                        lw, $a0, 0($sp)         # restore old registers
                        lw, $ra, 4($sp)
                        addi $sp, $sp, 8
                            mult $v0, $a0, $v0    # n * fact (n-1)
                            jr $ra                # return to caller
```

Procedure Call Instructions

- Procedure call: jump and link
  `jal ProcedureLabel`
  - Address of following instruction put in $ra
  - Jumps to target address
- Procedure return: jump register
  `jr $ra`
  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Coordinating Register usage

- Problem
  - Callee may destroy caller's data in registers
- Caller saving
  - Caller save register data to use after procedure call
    - Compiler not know what will happen after the call
- Callee saving
  - Callee save registers before use
    - Compiler has no idea of what they contain, be safe

† Both approaches are pessimistic

Coordinating Register usage

- MIPS policy
  - Saved registers ($s): callee backup o caller backup x
    - Preserved across call from caller perspective
      - Callee must save them before use
  - Temporary registers ($t): callee backup x caller backup o
    - Not preserved across call from caller perspective
      - Caller must save them if want to use it after call
- Better than caller or callee saving?
  - Use $t first; store value to be used after "call" in $s

  † Can you think of other strategies? (technical paper)
- Go back to "leaf example" and "recursion example"

Runtime Environment

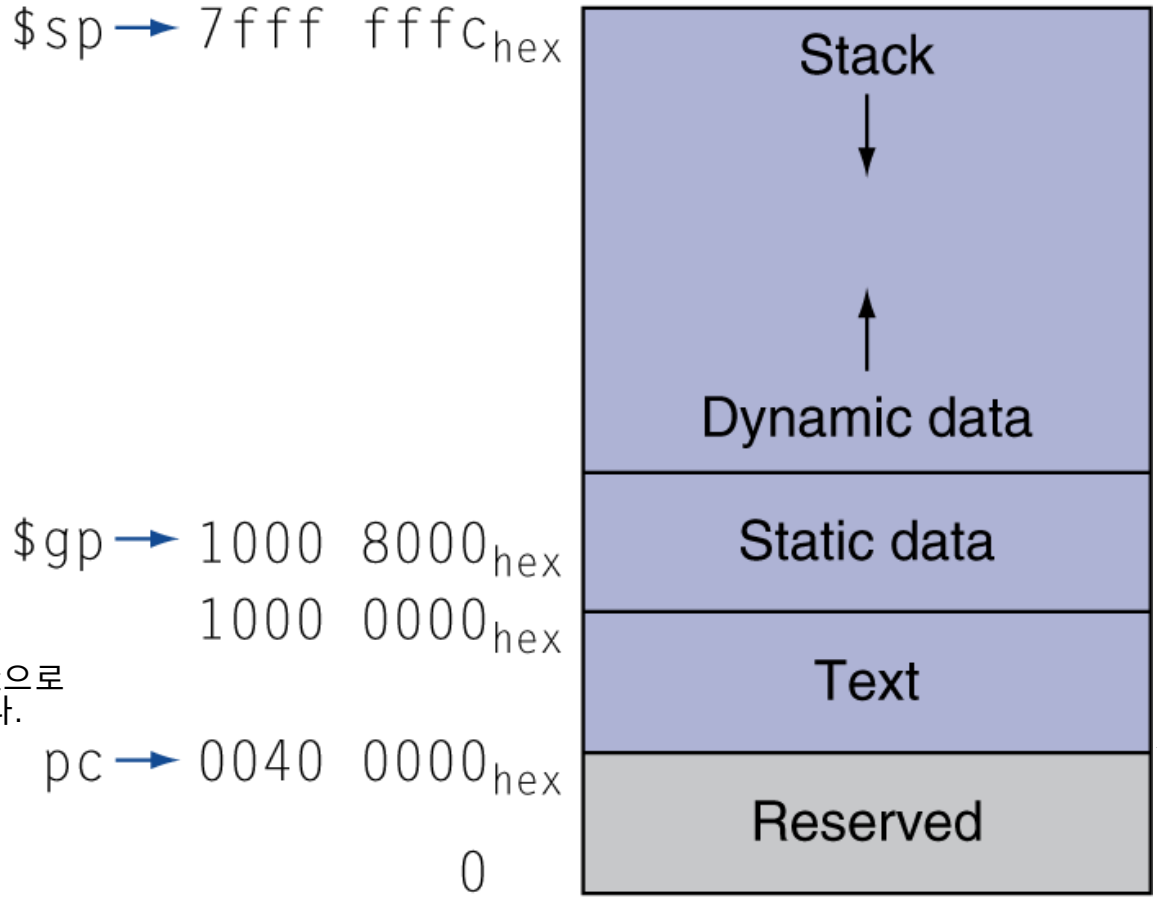
- How to organize registers and memory to maintain information needed by executing process
  - Agreement between CPU, OS, and compiler

Register Usage

- $a0 – $a3: arguments (reg's 4 – 7)
- $v0, $v1: result values (reg's 2 and 3)
- $t0 – $t9: temporaries (reg's 8–15, 24–25)
  - Can be overwritten by callee
- $s0 – $s7: saved (reg's 16–23)
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

Memory layout

- MIPS memory allocation for program and data
  - Static, automatic (runtime stack), dynamic (heap)

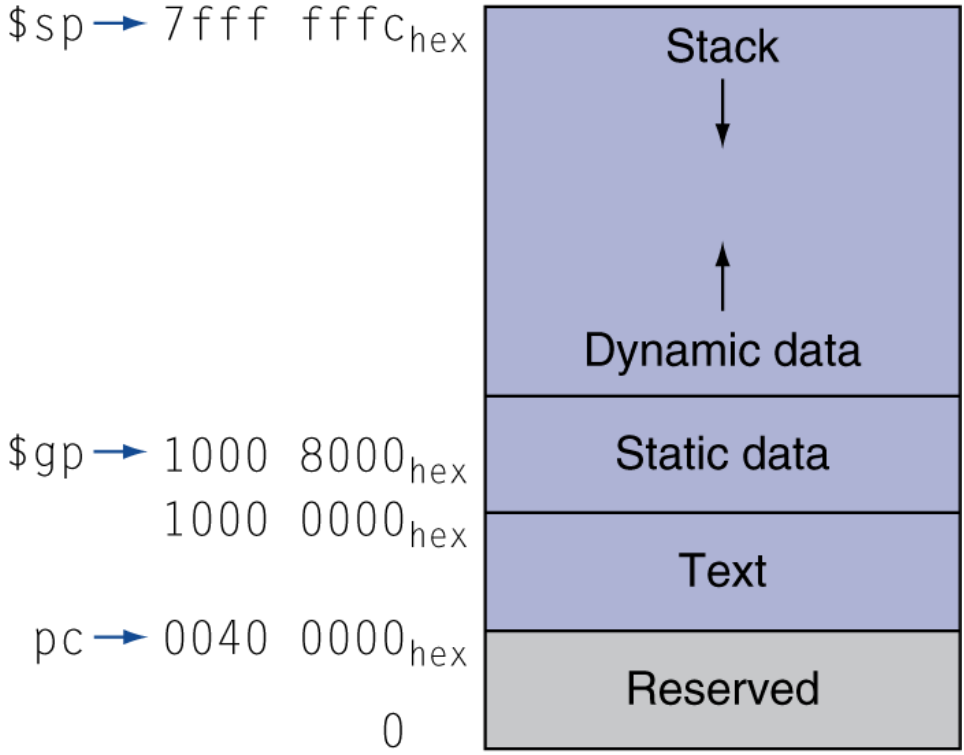


1000 0000
~ 1001 0000 에
global(static) data가
존재하기 때문에
$gp 기준으로 16bit offset으로
모든 static에 접근가능하다.
(I-format)

-> 상위 4bit 모두 0이다
따라서 pseudo direct
jump가 가능하다.

Memory Layout

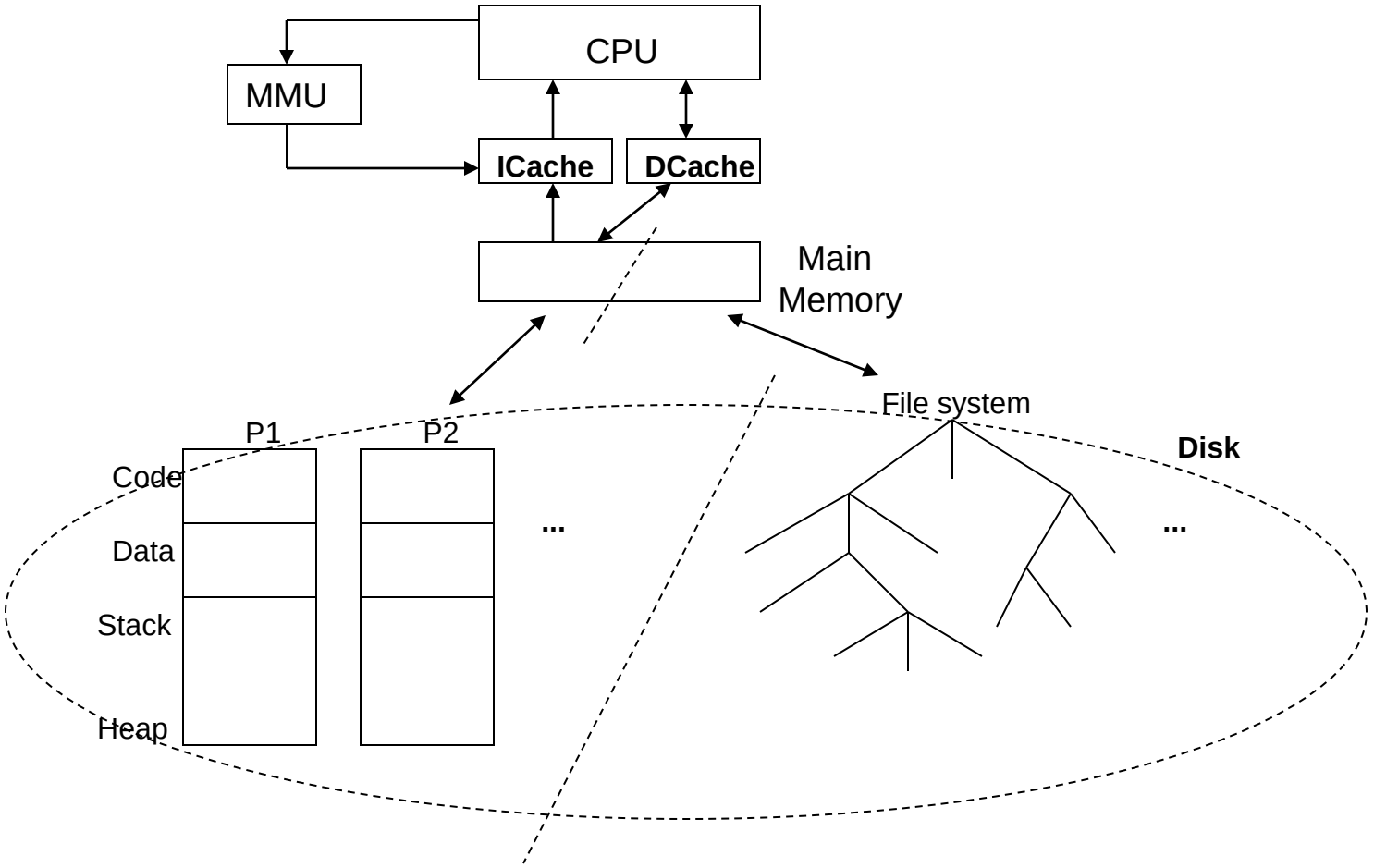
- Text: program code
- Static data: global variables
  - e.g.,static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



Virtual Memory

- Where is the address space located?
  - How do we use disk space?
    - To support process execution
    - Permanent storage (file system)
- Disk access very slow
  - OS (software) use part of main memory as a cache
    - Page table
  - Main memory slower than CPU
    - Use smaller faster cache memory
- How does it compare with physical memory system?

General-Purpose Computer

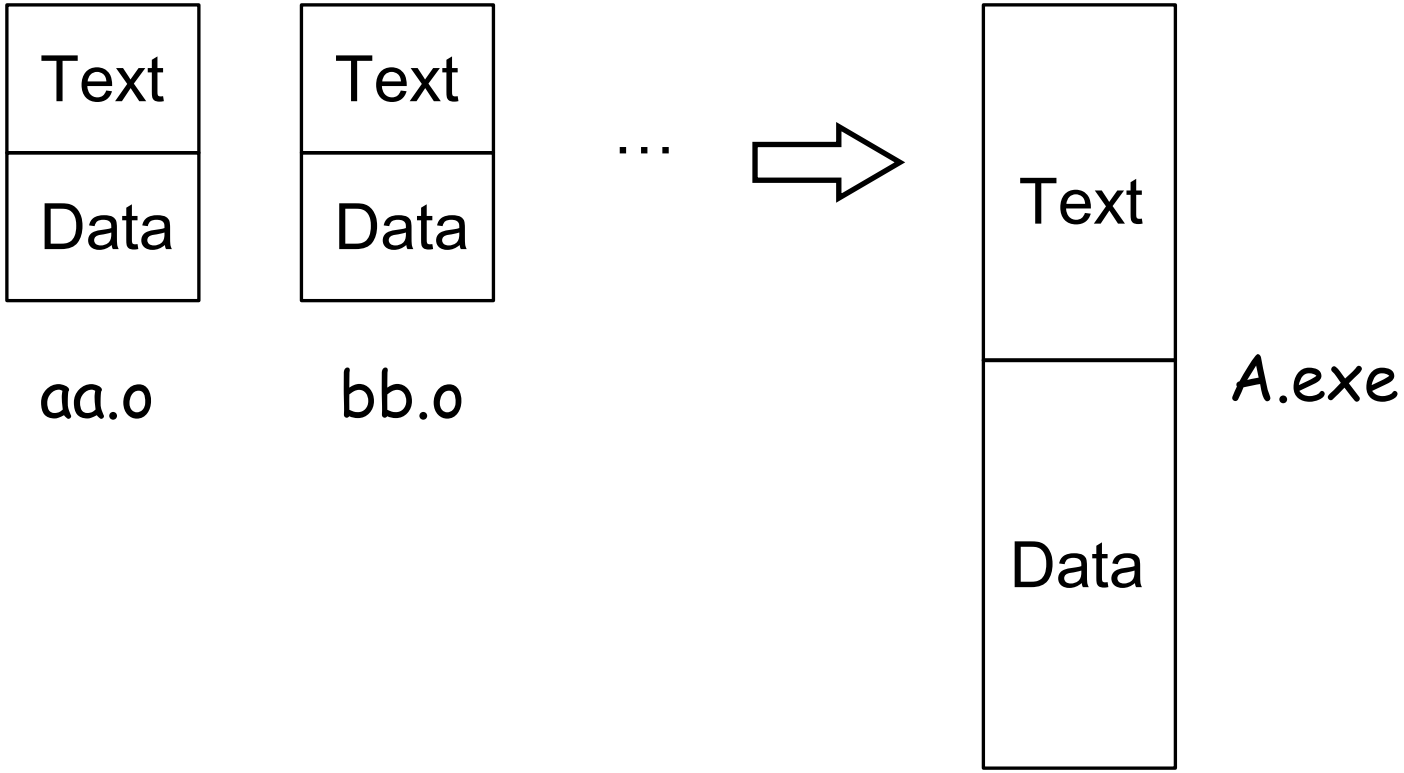


Compile, Link, and Run

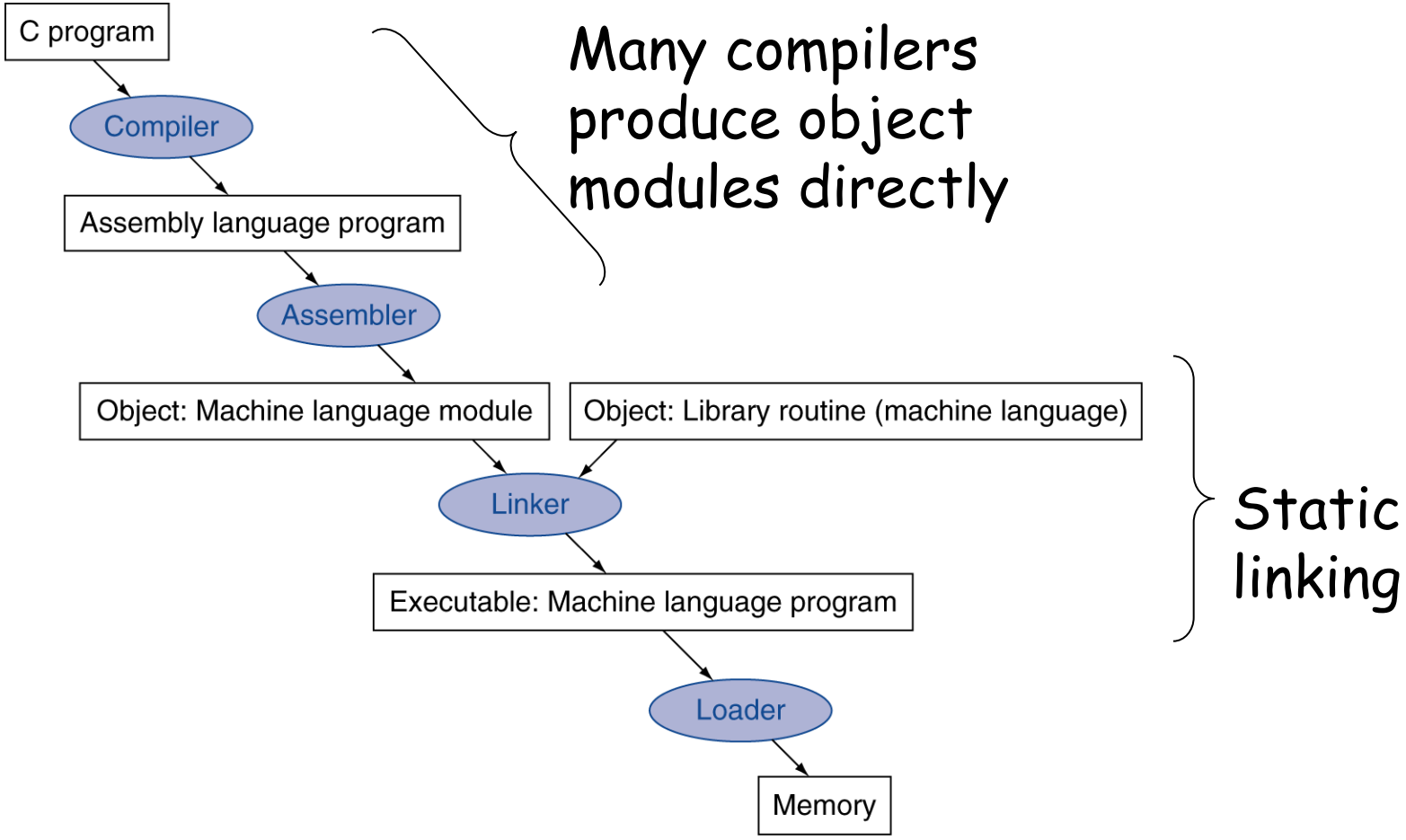
Separate compilation

- Modular development
- Relocation or link editing or linking

<separate coding>
1. 프로그램 개발 혼자서 불가능
- source code 나눠서 coding
2. object file 구매 가능



Translation and Startup



Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Linking Object Files

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	lw $a0, 0($gp)	
	4	jal 0	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	—	
	B	—	

최종 위치를 모른다.

위치를 알지 못한다.
- linker을 위한 정보
정확한 위치를 모르는 것들은 relocation후 linker가 해결해주어야한다.

Linking Object Files

Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sw $a1, 0($gp)`	
	4	`jal 0`	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`sw`	Y
	4	`jal`	A
Symbol table	Label	Address	
	Y	—	
	A	—	

Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs

Linking Object Files

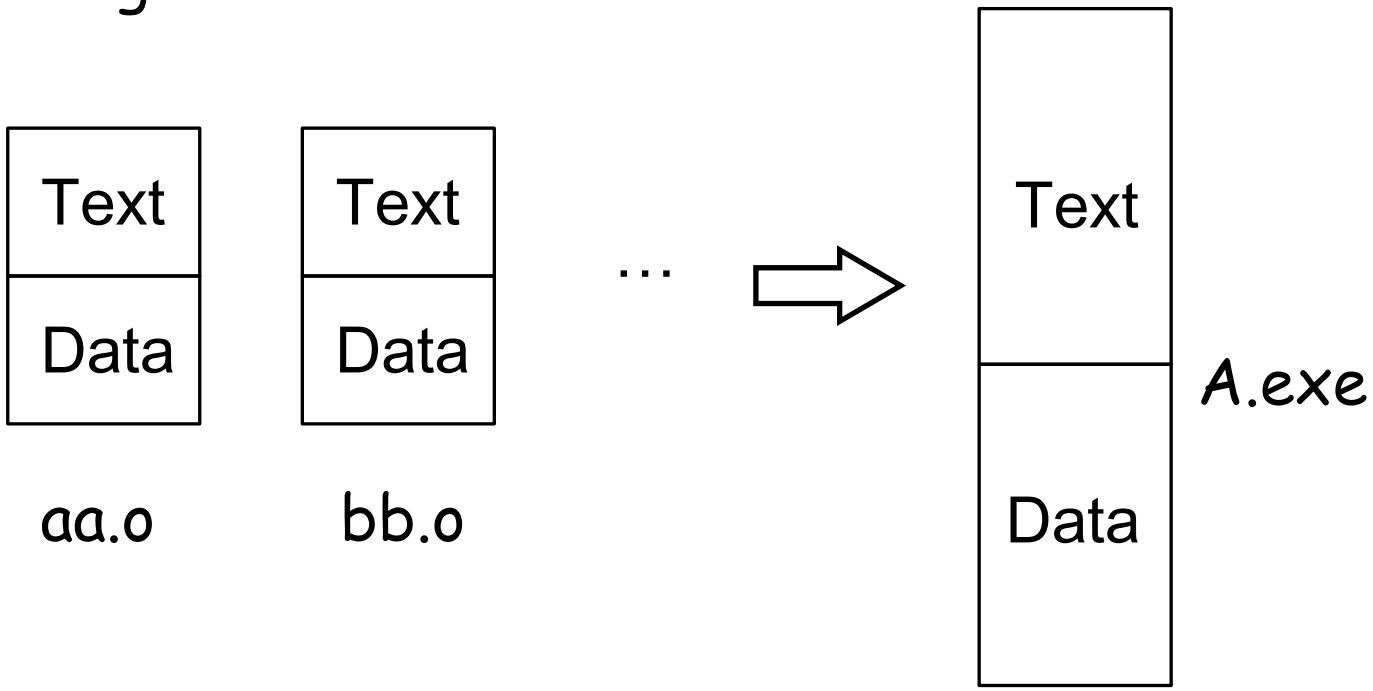
Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	lw $a0, 8000_{hex}($gp)
	$0040\ 0004_{hex}$	jal 40 0100_{hex}
	...	...
	$0040\ 0100_{hex}$	sw $a1, 8020_{hex}($gp)
	$0040\ 0104_{hex}$	jal 40 0000_{hex}
	...	...
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	...	...
	$1000\ 0020_{hex}$	(Y)
	...	...

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine
     - Copies arguments to $a0, … and calls main
     - When main returns, do exit syscall

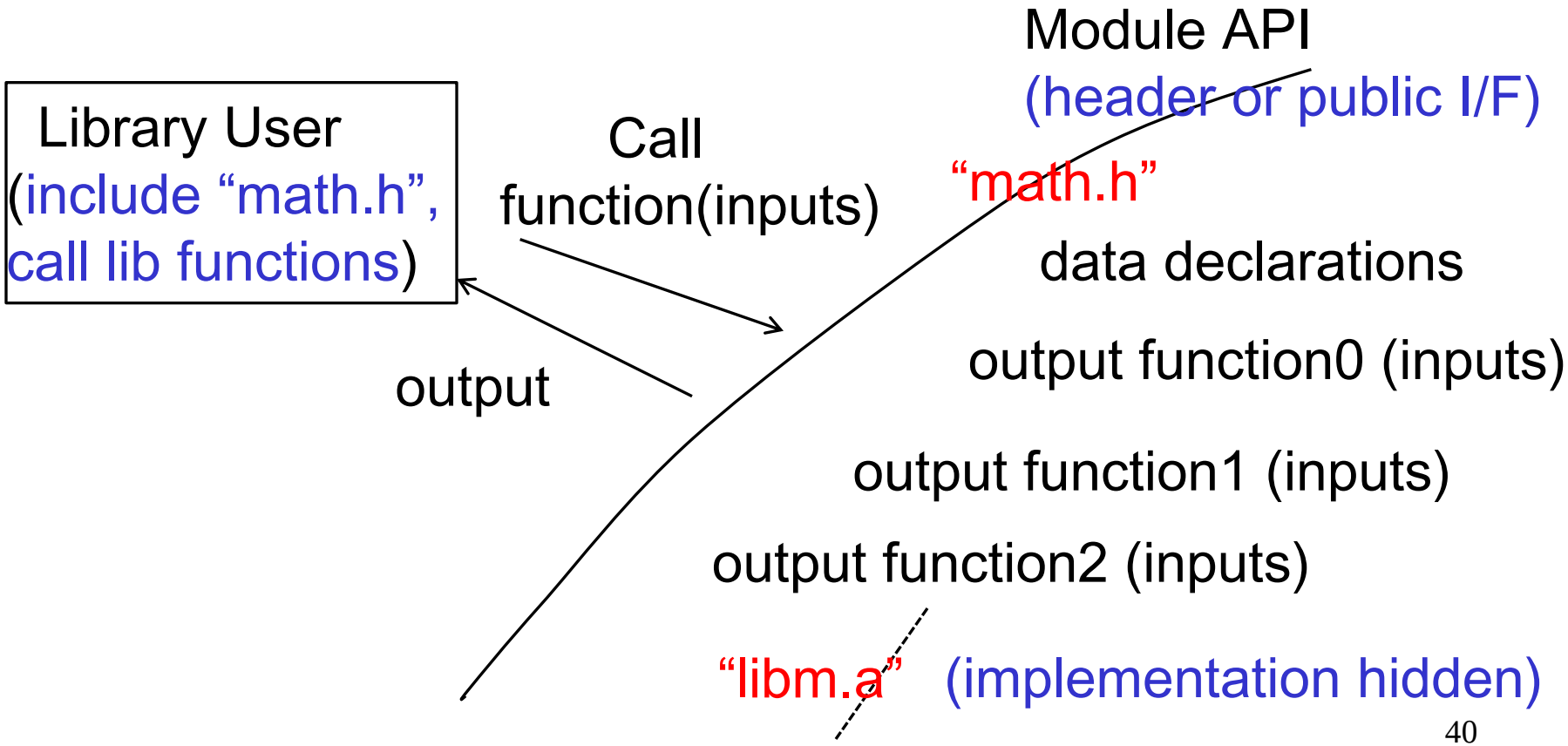
To Think about

- What does "data" mean? (static, stack or heap)
- Examples of static data in C program
- Linking in Java
- Dynamic linking



What is library or API?

- Collection of functions
  - Composite data as arguments and return values



Static vs. Dynamic Linking Library

"hello.c"

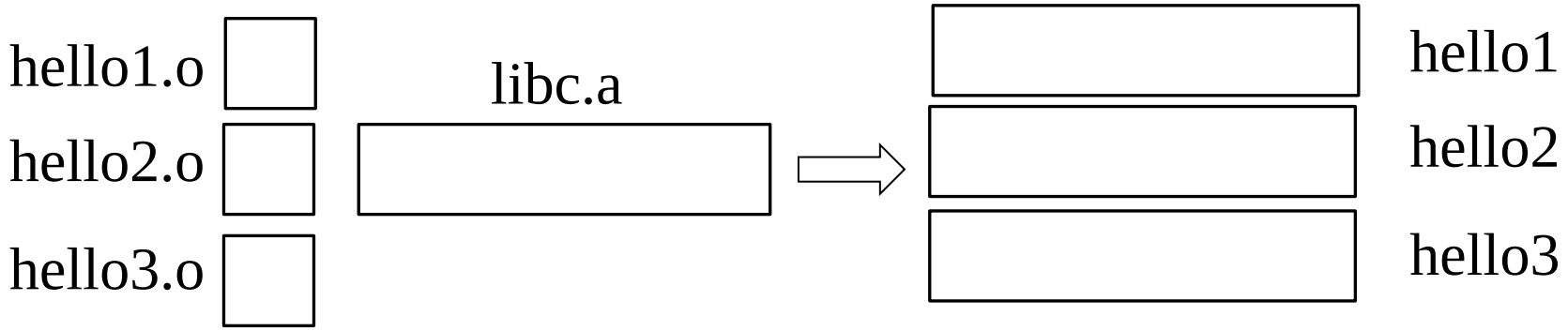
```
#include <stdio.h>

void main()
{
   printf ("Hello, world!\n");
}
```

- Compile and link (your system has "libc.a" and libc.so")

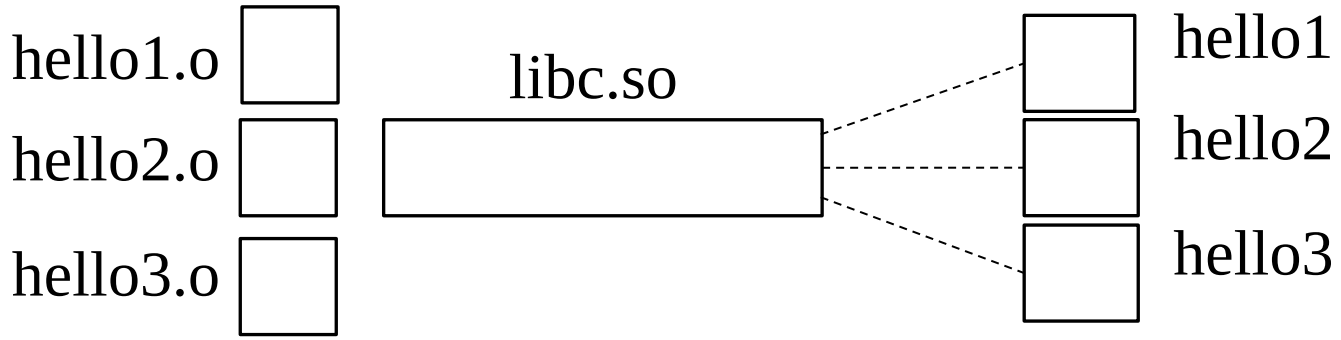
```
$ gcc hello.c –o hello
$ gcc hello.c –o hellostatic -static
```

Static vs. Dynamic Linking Library

- Think about: hello1.c, hello2.c, hello3.c, …
  - Static linking



  - Dynamic linking (default choice): library is shared object



Dynamic Linking

- Only link/load library procedure when it is called (early vs. lazy)
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

부풀다.
파일 크기 커지는 현상

<dynamic linking 장점>
1. exe file 크기 작아진다.
2. software 업데이트 편하다.

1. early linking
프로그램 시작할때 linking
2. lazy linking
함수 쓸대 linking

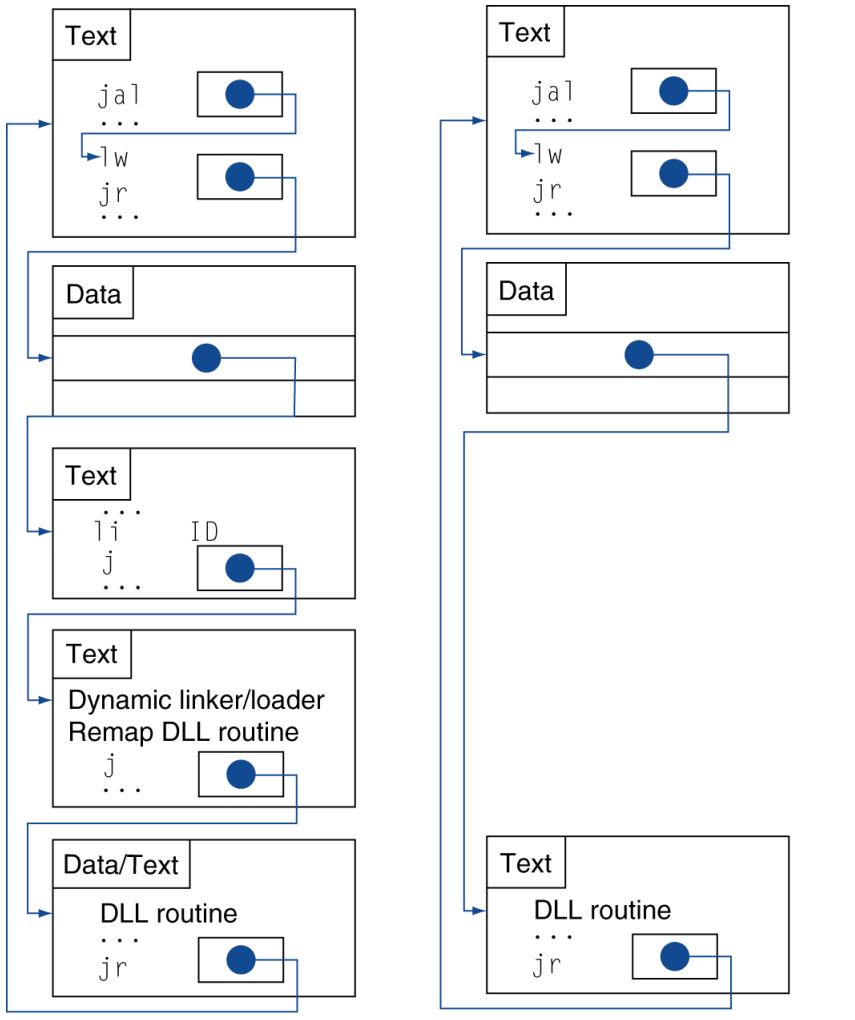
Lazy Linkage

Indirection table

Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

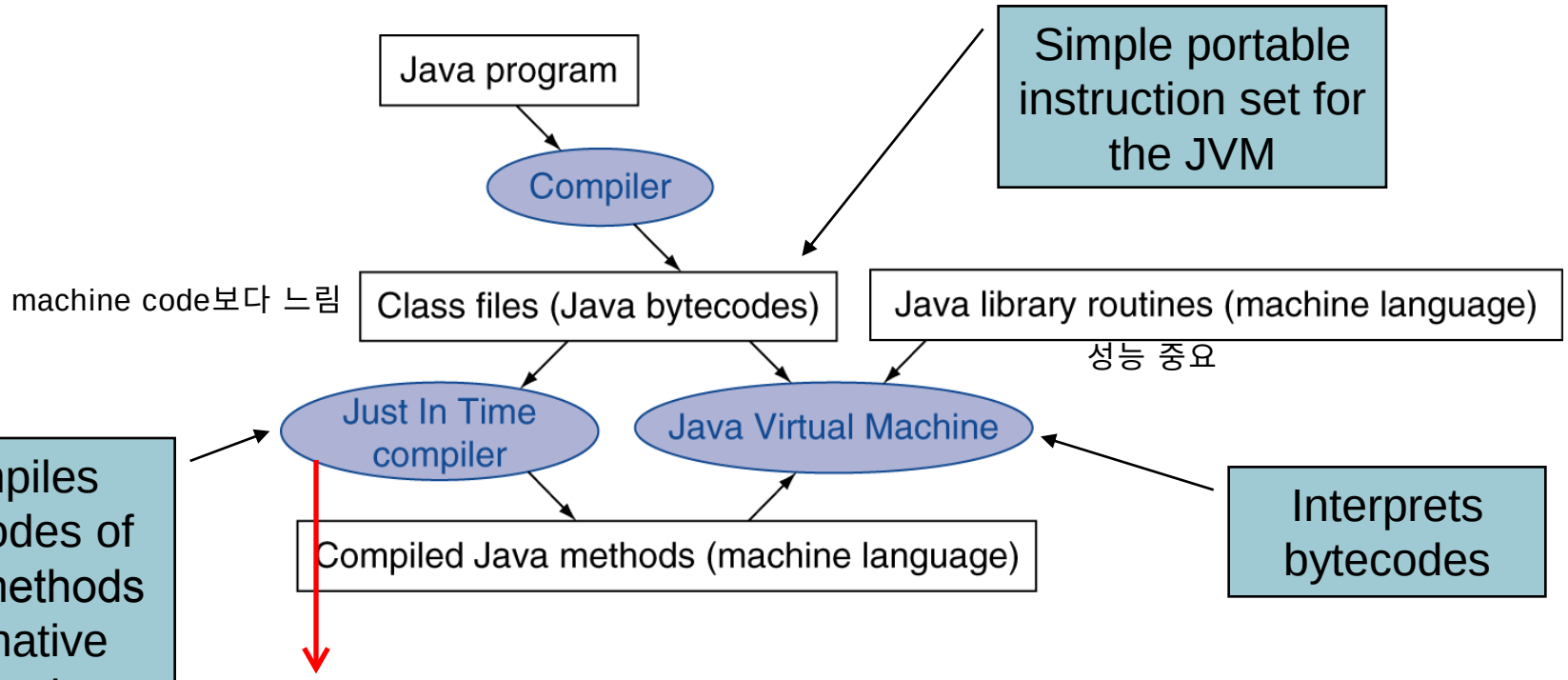
Dynamically mapped code



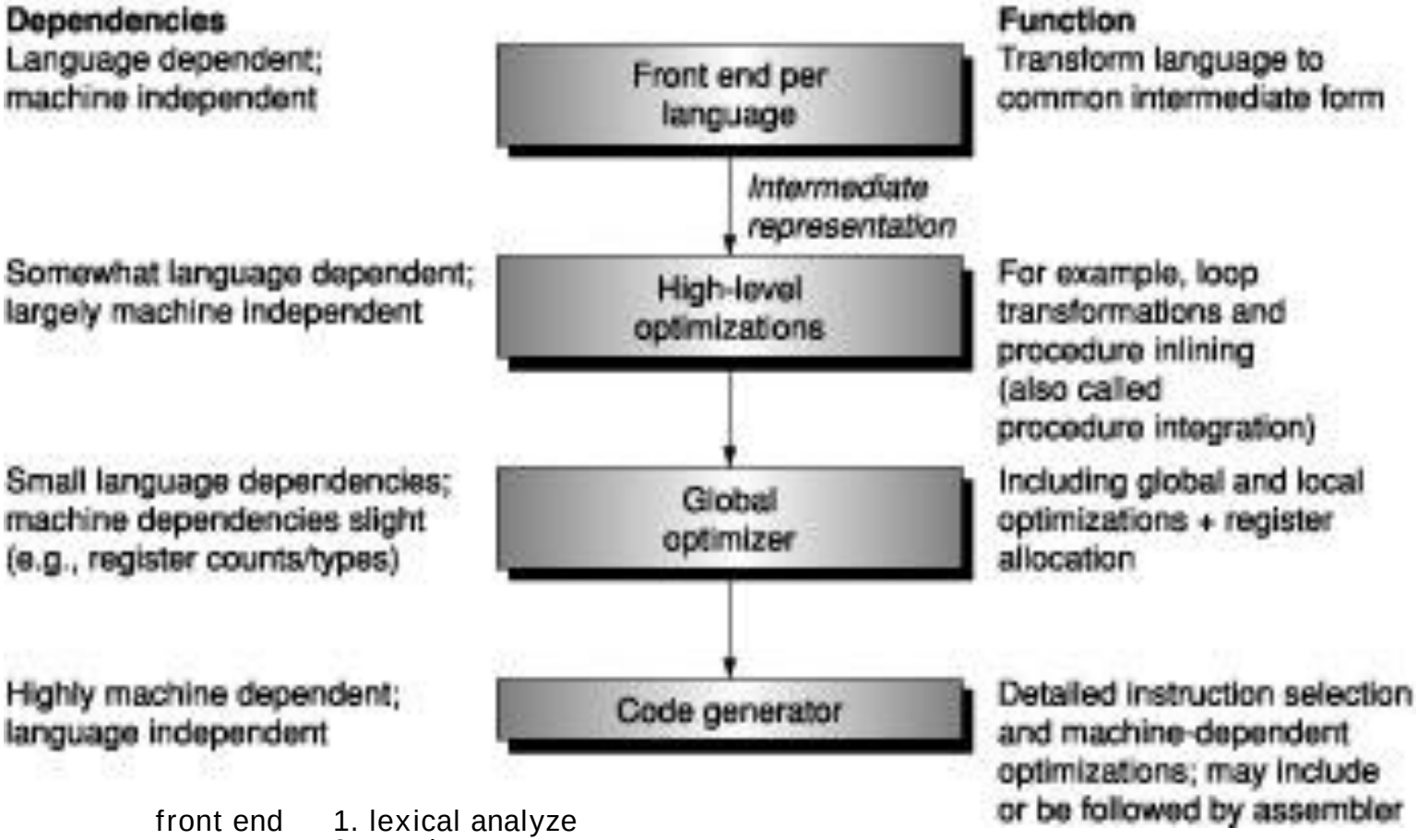
a. First call to DLL routine

b. Subsequent calls to DLL routine

Starting Java Applications



Compiler Optimization



front end 1. lexical analyze
2. parsing
3. intermediate code generation (가상 assembly)

backend optimization(컴퓨터 cpu에 맞게 optimization 시킴)

양산품 - 버그때문에 correctness 중요!

c debug — system debug

source level debug — machine level debug

“gcc” Optimization Level

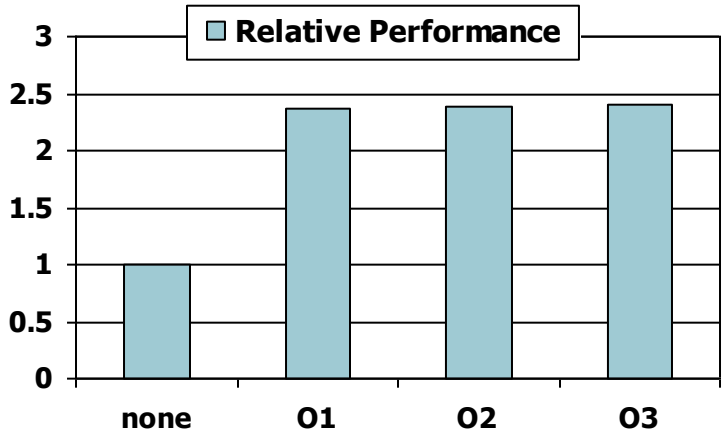
Optimization name	Explanation	gcc level
High level	*At or near the source level; processor independent*	
Procedure integration	Replace procedure call by procedure body	03
Local	*Within straight-line code*	
Common subexpression elimination	Replace two instances of the same computation by single copy	01
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	01
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	01
Global	*Across a branch*	
Global common subexpression elimination	Same as local, but this version crosses branches	02
Copy propagation	Replace all instances of a variable *A* that has been assigned *X* (i.e., *A* = *X*) with *X*	02
Code motion	Remove code from a loop that computes same value each iteration of the loop	02
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	02
Processor dependent	*Depends on processor knowledge*	
Strength reduction	Many examples; replace multiply by a constant with shifts	01
Pipeline scheduling	Reorder instructions to improve pipeline performance	01
Branch offset optimization	Choose the shortest branch displacement that reaches target	01

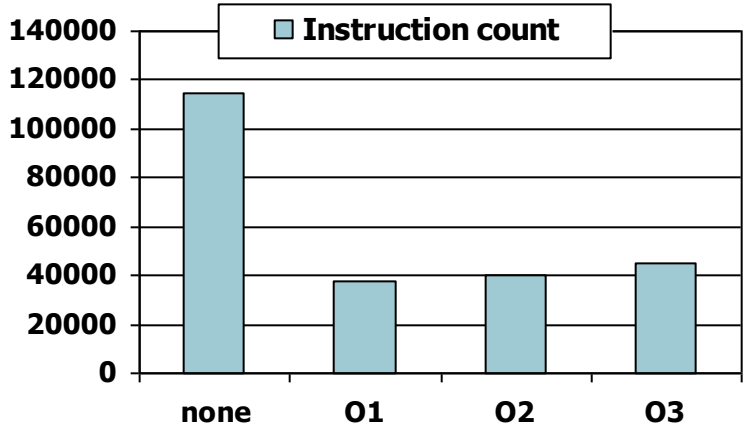
bug 때문에 잘 이용x

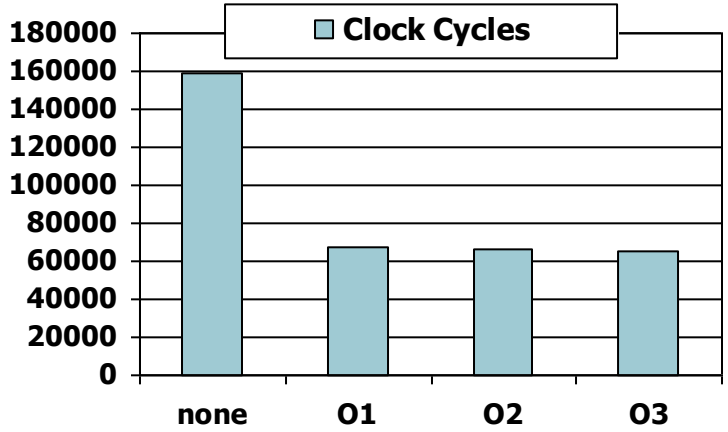
FIGURE 2.32 Major types of optimizations and examples in each class. The third column shows when these occur at different levels of optimization in gcc. The Gnu organization calls the three optimization levels medium (O1), full (O2), and full with integration of small procedures (O3).

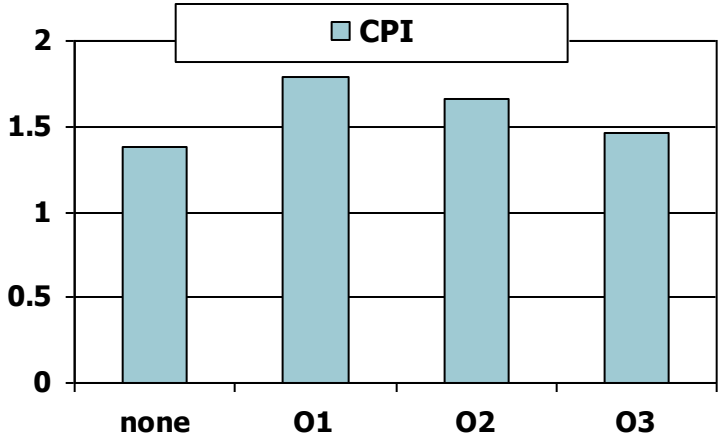
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

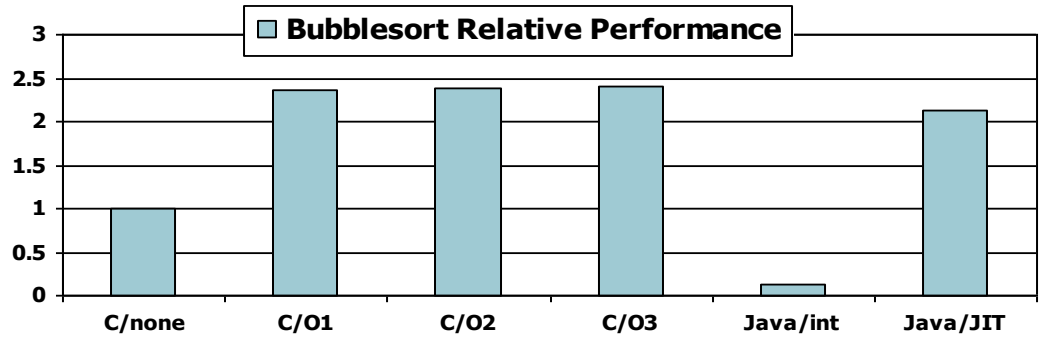


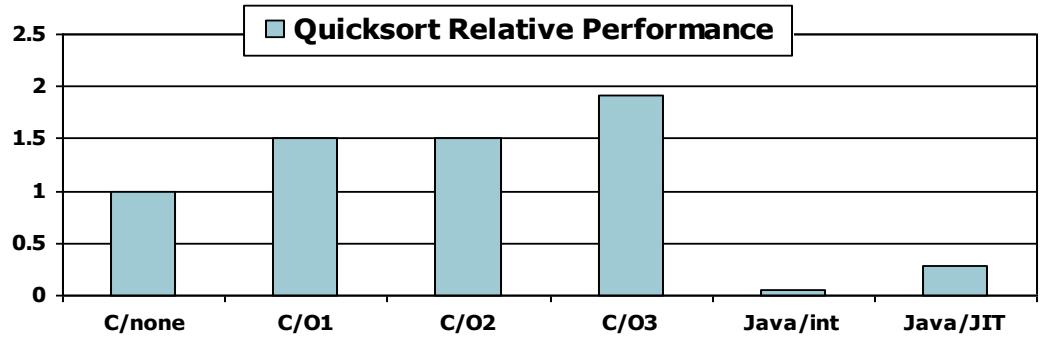


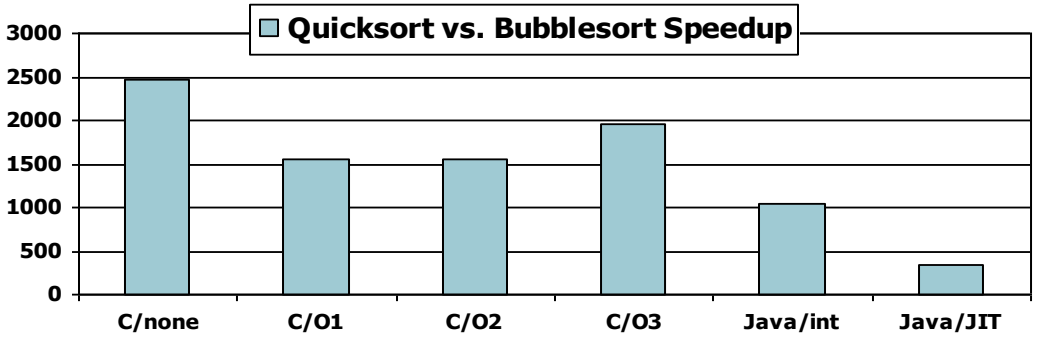




Effect of Language and Algorithm







Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

Remaining Parts of Chapter 2

- (optional) Alternative architectures: ARM, IA-32
- (optional) Parallelism and instructions: synchronization