

Class Topics (클래스 홈페이지 참조)

- ❑ Part 1: Fundamental concepts and principles
- ❑ Part 2: 빠른 컴퓨터를 위한 ISA design
 - Topic 1 Computer performance and ISA design (Ch. 1)
 - Topic 2 RISC (MIPS) instruction set (Chapter 2)
 - 2-1 ALU and data transfer instructions
 - 2-2 Branch instructions
 - 2-3 Supporting program execution
 - Topic 3 Computer arithmetic and ALU (Chapter 3)
- ❑ Part 3: ISA 의 효율적인 구현 (pipelining, cache memory)

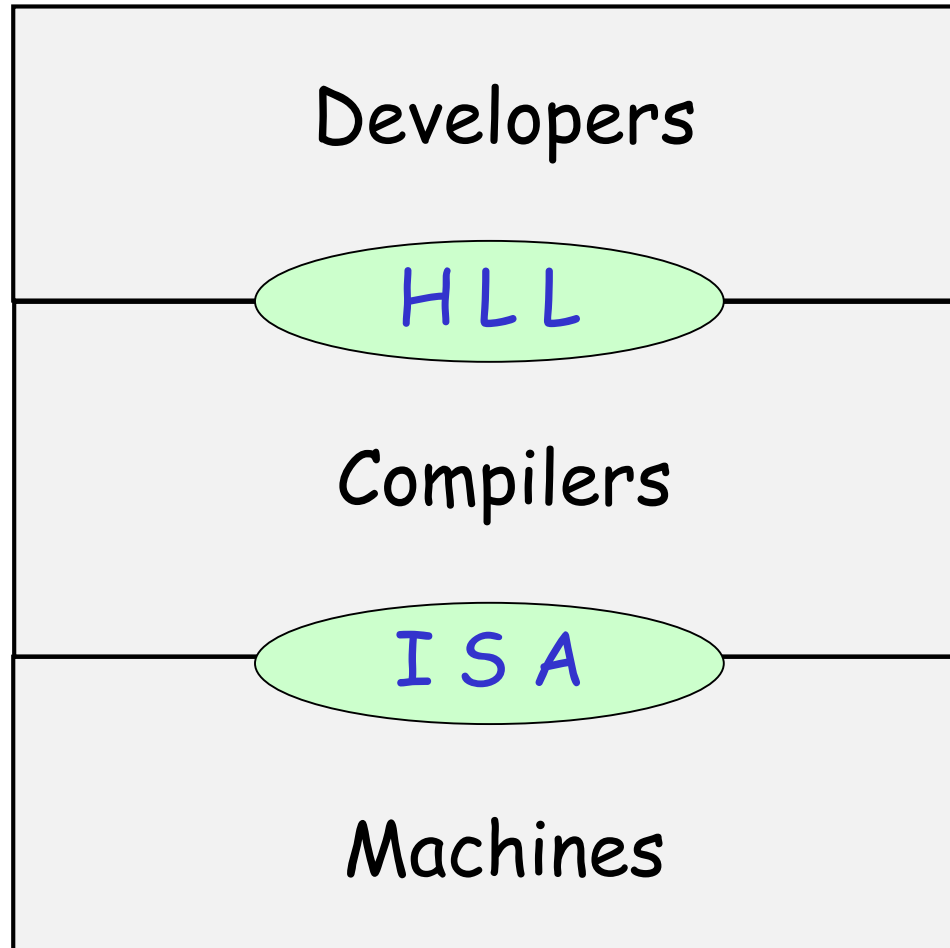
Chapter 2

Instructions: Language of the Computer

Part 3:

- **Run C programs**
- **Procedure calls**

Program Execution (Levels of Abstraction)

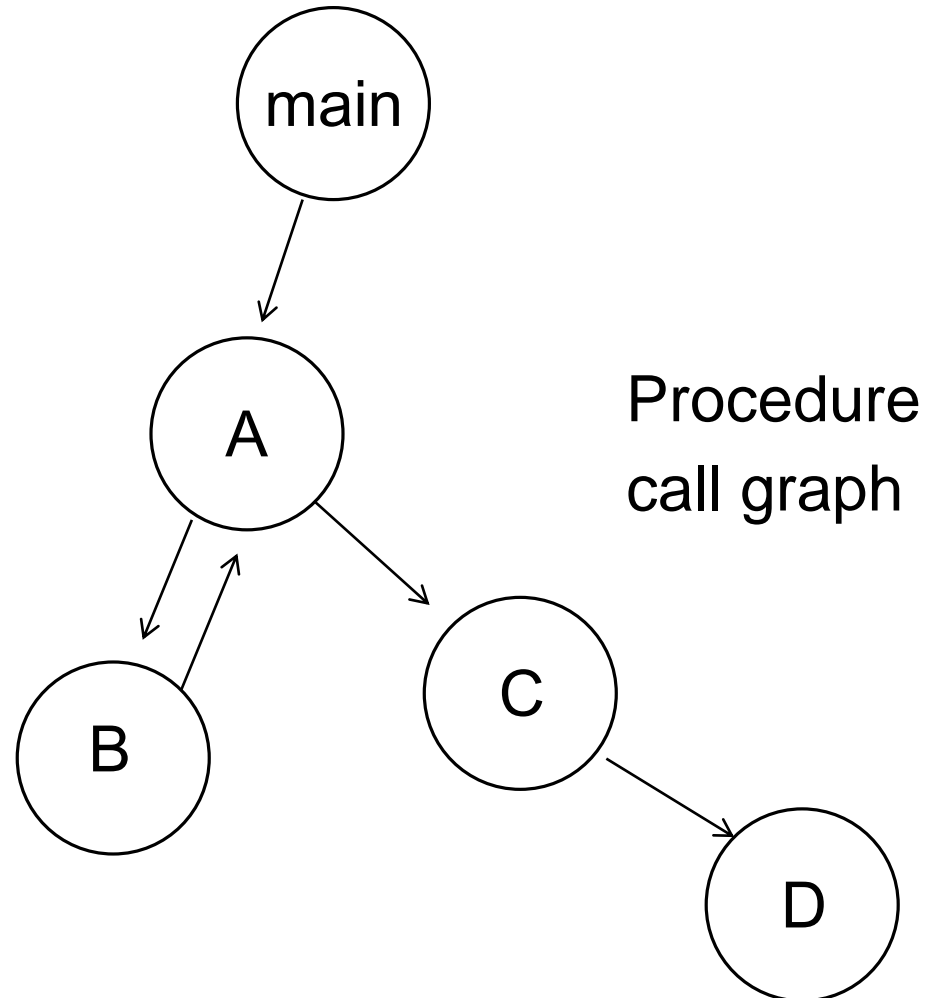


Program Execution (Levels of Abstraction)

- ❑ HLL perspective
 - Object level
 - Procedure level
 - Another important level of abstraction
 - Statement level
- ❑ ISA perspective
 - Machine instruction level (HW-SW interface)
 - Processor internal level (fetch, decode, execute)

Program Execution

- Procedure calls
 - Fundamental
 - Heavy operations
 - HW support



Topic 2 (Chapter 2) Overview

- ❑ Topic 2-1 and 2-2
 - Supporting statements
- ❑ Topic 2-3
 - Function call and return
 - Compile, link and run

Leaf Procedure Example

❑ C code



```
int leaf_example (int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

❑ What is leaf procedure?

❑ Caller versus callee

Supporting Leaf Procedures

- ❑ Work to do
 - Caller
 1. Place parameters so that callee can access them
 2. Jump to callee (change PC; jump and link)
 - Callee (“leaf example”)
 3. Acquire needed registers
 4. Perform desired task
 5. Place results so that caller can access them
 6. Return to caller (change PC; jump register)

Supporting Leaf Procedures

□ Issues

- Where to place parameters, return values
- Where to specify return address
 - Returning to right instruction
- Coordination of register usage (caller and callee)

Supporting Leaf Procedures

- ❑ Where to place parameters and return values?
 - Where to specify return address?
- ❑ Registers are fastest: use them if possible
 - \$a0 - \$a3: 4 argument registers to pass parameters
 - \$v0 - \$v1: two registers to return values
 - \$ra: register to store return address
- ❑ What if there are more than four parameters?
 - Why two return value registers?
- ❑ What if want to return more than one item?

MIPS Register Conventions

(Fig. 2.14)



Name	Register number	Usage	Preserved on call?
\$zero	0	The constant value 0	n.a.
\$v0-\$v1	2-3	Values for results and expression evaluation	no
\$a0-\$a3	4-7	Arguments	no
\$t0-\$t7	8-15	Temporaries	no
\$s0-\$s7	16-23	Saved	yes
\$t8-\$t9	24-25	More temporaries	no
\$gp	28	Global pointer	yes
\$sp	29	Stack pointer	yes
\$fp	30	Frame pointer	yes
\$ra	31	Return address	yes

□ Will explain \$s, \$t, \$gp, \$sp, \$fp soon

Supporting Leaf Procedures

❑ Work to do

- Caller

1. Place parameters so in `$a0 - $a3`
2. Jump & save return address: `jal CalleeAddress`

- Callee ("leaf example")

3. Need registers to work; save them in stack
4. Perform task and put results in `$v0 - $v1`
5. Restore registers from stack
6. Return to caller: `jr $ra`

Leaf Procedure Example


❑ C code

```
int leaf_example (int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

- ❑ Arguments g, \dots, j in $\$a0, \dots, \$a3$ (in caller code)
- ❑ f in $\$s0$ (hence, need to save $\$s0$ on stack)
- ❑ Result in $\$v0$

Leaf Procedure Example (1)

□ 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 $f = (g + h) - (i + j)$`

`add $t1, $a2, $a3`

`sub $s0, $t0, $t1`

`add $v0, $s0, $zero // set return value register`

`lw, $s0, 0($sp) // restore registers`

`lw, $t0, 4($sp)`

`lw, $t1, 8($sp)`

`addi $sp, $sp, 12`

`jr $ra // jump back to caller`

Stack (Fig. 2.10), (before/during/after Procedure Call)

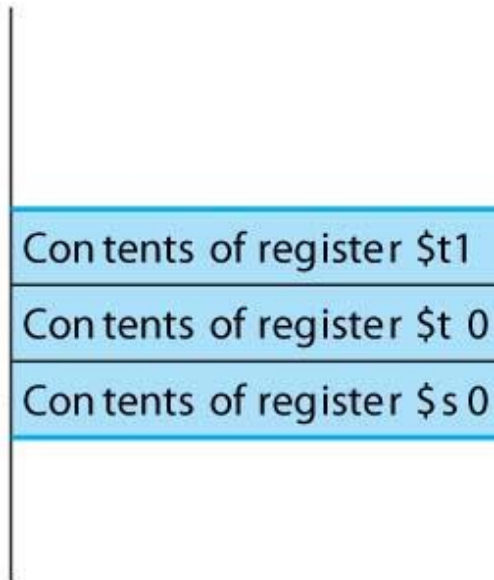
High address

\$sp →



a.

\$sp →



b.

\$sp →



c.

push 3 registers pop 3 registers

Coordinating Register Usage

Coordinating Register Usage

- ❑ Problem: callee may destroy caller's data in registers
- ❑ Callee saving (previous "leaf example")
 - Callee save registers before use
 - Compiler has no idea of what they contain, be safe
- ❑ Caller saving
 - Caller save register data to use after procedure call
 - Compiler not know what will happen after the call
- ❑ Both approaches are pessimistic

Coordinating Register Usage

- ❑ MIPS policy: 계산에 사용되는 register 를 두 그룹으로
 - Saved registers (\$s): callee saving
 - Preserved across call from caller perspective
 - † “main” 제외한 function: \$s 쓰기 전 save in stack
 - Temporary registers (\$t): caller saving
 - Not preserved across call from caller perspective
 - † functions can freely use \$t
- ❑ Within a function, how do we use registers? (compilers)

Coordinating Register Usage

- ❑ MIPS saved registers (\$s): callee saving (반복)
 - Temporary registers (\$t): caller saving
- ❑ Within a function, how do we use registers? (compilers)
 - Use \$t for free
 - If need more registers, save \$s and use them
 - If still need more registers, use memory
 - Values to be used after a call: save \$s and use them
- ❑ Performance better than caller or callee saving?

Leaf Procedure Example (2)

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

`addi $sp, $sp, -4` // save $\$t1, t0, s0$ in stack

`sw $s0, 0($sp)`

`add $t0, $a0, $a1` // calculate $f = (g + h) - (i + j)$

`add $t1, $a2, $a3`

`sub $s0, $t0, $t1`

`add $v0, $s0, $zero` // set return value register

`lw, $s0, 0($sp)` // restore registers

`addi $sp, $sp, 4`

`jr $ra` // jump back to caller

Leaf Procedure Example (3)

□ g, h, i, j : $\$a0, \$a1, \$a2, \$a3$ f : $\$v0$

□ `add $t0, $a0, $a1 // calculate $f = (g + h) - (i + j)$`
`add $t1, $a2, $a3`
`sub $v0, $t0, $t1`
`jr $ra // jump back to caller`

□ What about non-leaf procedures?

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

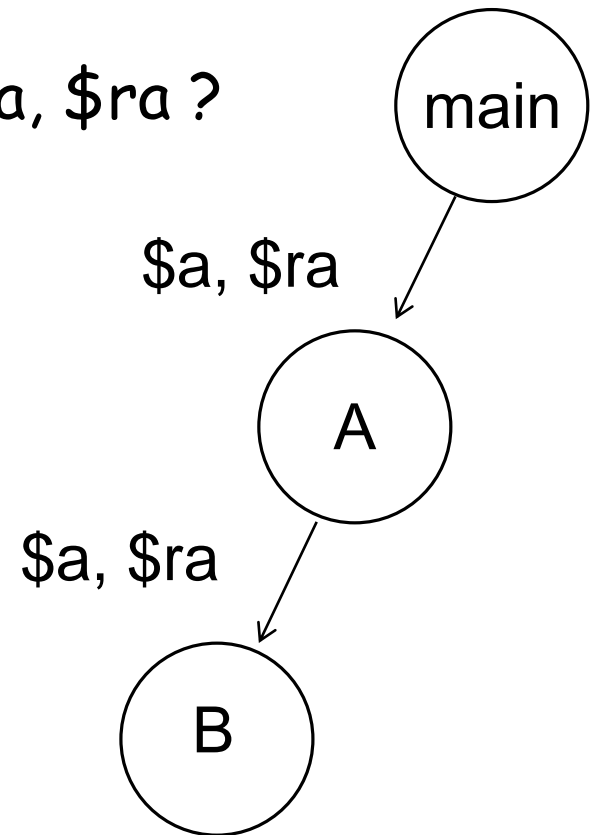
□ MIPS code:

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

Non-Leaf Procedures

Non-Leaf Procedures

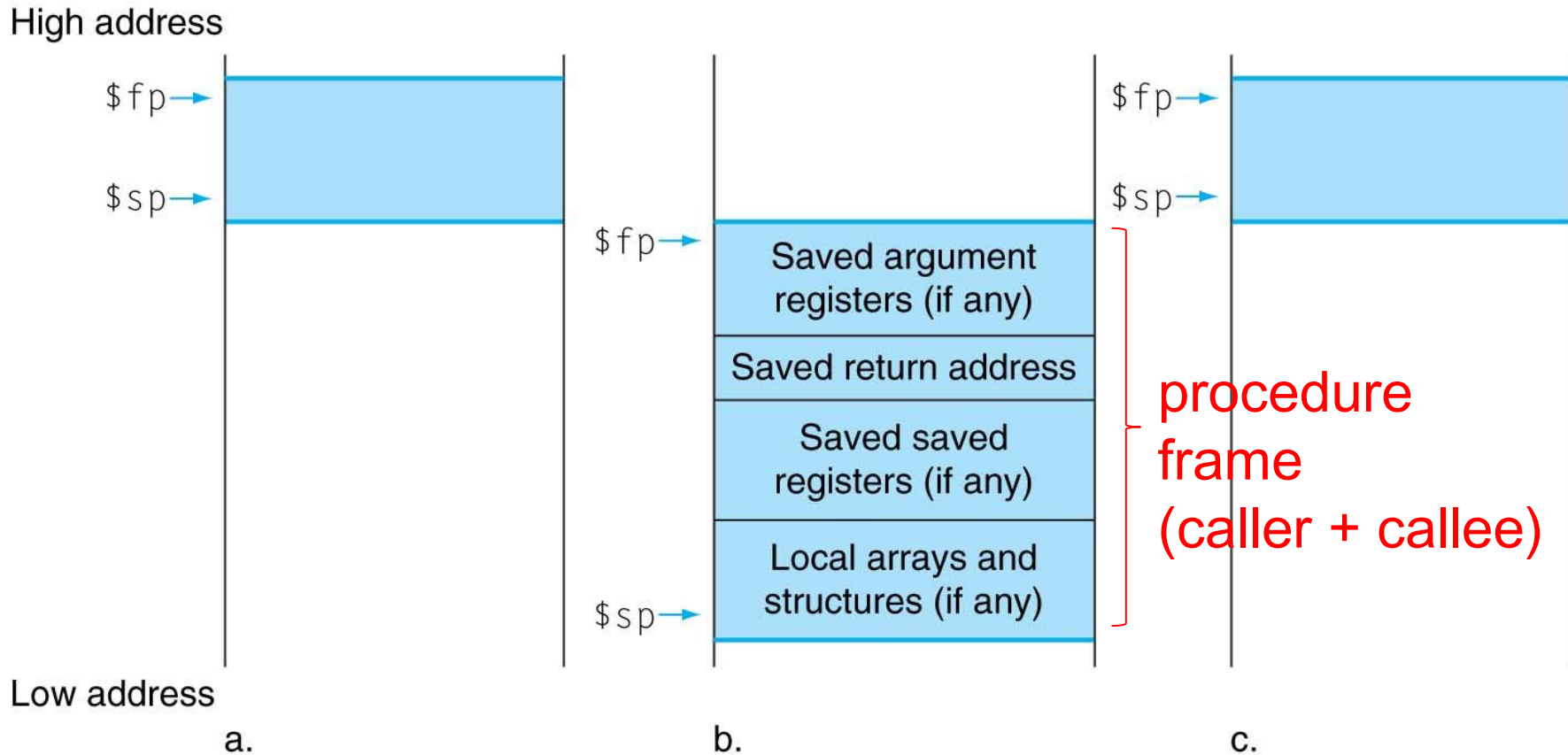
- ❑ Non-leaf procedures
 - Procedures that call other procedures
- ❑ In a nested call, how to protect $\$a$, $\$ra$?



Nested Procedure Call

- ❑ In a nested call, how do we protect $\$a$, $\$ra$?
 - Save $\$a$, $\$ra$ on the stack before the call
 - Restore them after the call
- ❑ Just like how we protect register data ($\$t$, $\$s$)

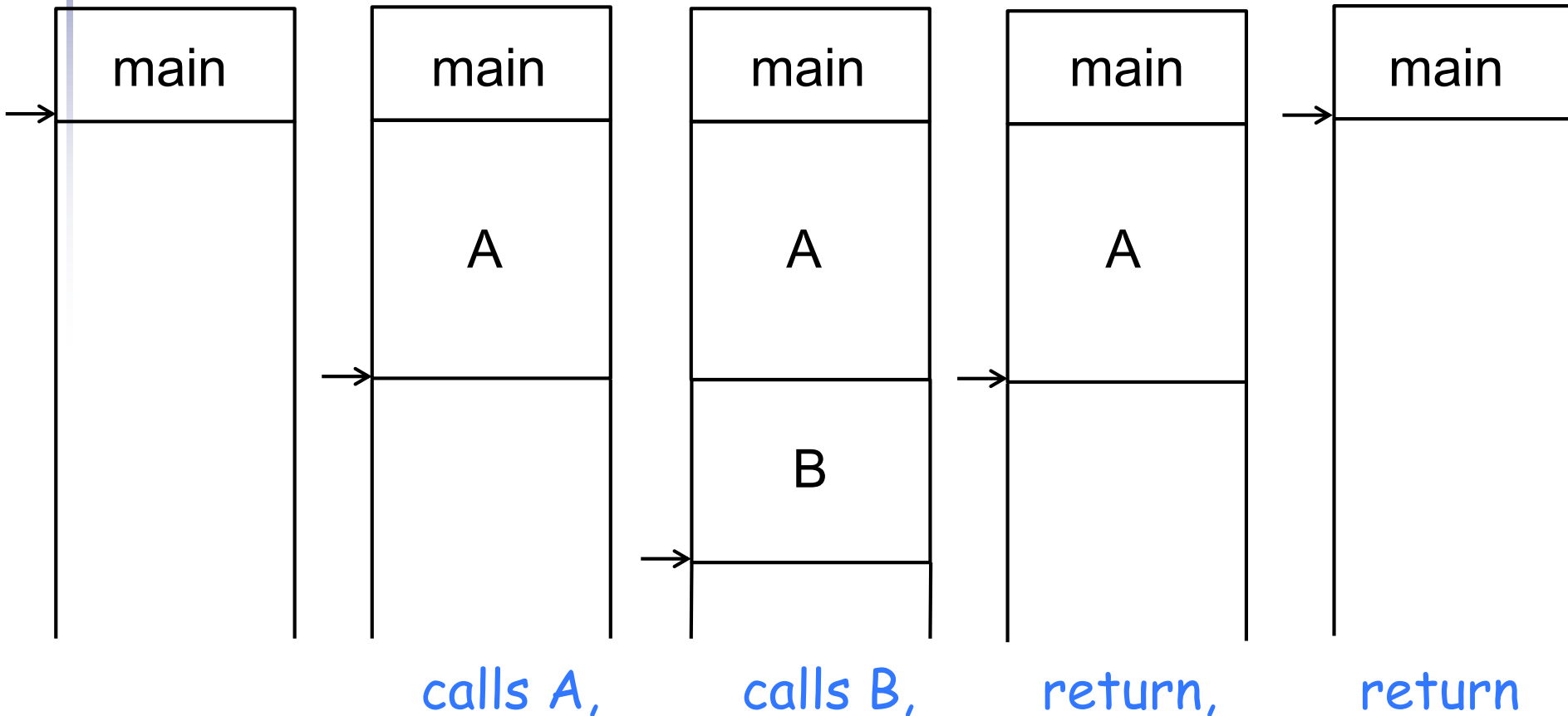
Runtime Stack (Fig. 2.12), before/during/after Procedure Call



❑ Compiler knows the size of each area (\$fp 일단 무시)

Runtime Stack

- Procedure activation record (push and pop frames)



Runtime Stack

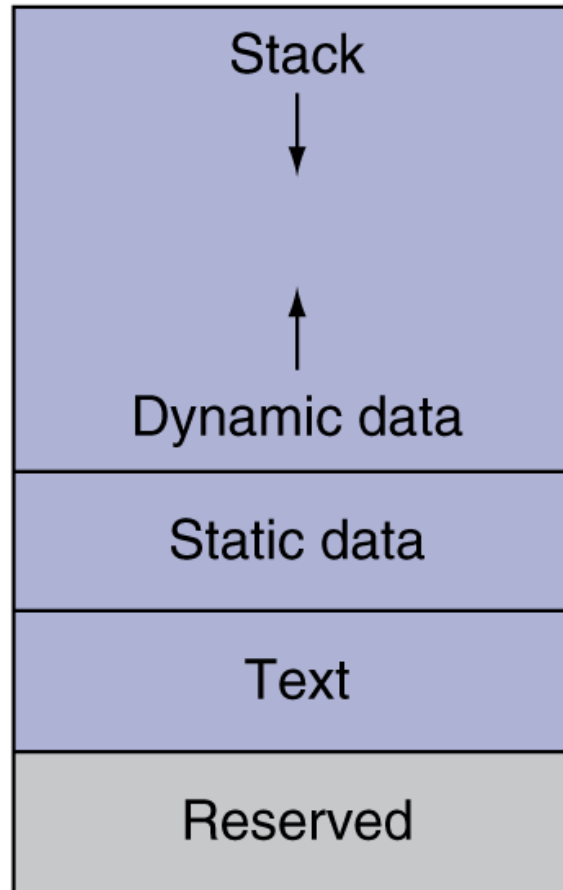
- ❑ Where is the runtime stack located?

\$sp → 7fff fffc_{hex}

\$gp → 1000 8000_{hex}
1000 0000_{hex}

pc → 0040 0000_{hex}

0



Process
virtual
address space

More on Runtime Stack

- ❑ Local variables are automatic variables
 - Dynamic allocation, but allocation/deallocation in sync with procedure call and return (why use stack?)
 - Heap: fully dynamic memory allocation
- ❑ Why use stack for storing local variables?
 - Storage efficiency, recursion
- ❑ Compiler access local variables: $\$sp + \text{offset}$
 - Base addressing mode: `lw $t0, 32($sp)`
- ❑ Stack (or procedure call) trace
 - Problem diagnosis (core dump)

Non-Leaf Procedures

(Example)

Nested Procedure Call

- ❑ Extreme example: recursion
 - Specify terminal condition
 - Sequentially reduce computation

```
int fact (int n)                //  $n! = n * (n-1)!$ 
{
    if (n < 1) return 1;        //  $0! = 1$ 
    else return (n * fact (n - 1));
}
```


Recursion

- ❑ Computational process (machine code? runtime stack?)

$\text{fact}(3) = 3 * \text{fact}(2)$ $// n! = n * (n - 1)!$

$\text{fact}(2) = 2 * \text{fact}(1)$

$\text{fact}(1) = 1 * \text{fact}(0)$

$\text{fact}(0) = 1$ $// 0! = 1$

- ❑ Performance and elegance
 - Iteration, tail-recursion
- ❑ Programs or CPU not tell self-call from calling others

Nested Procedure Call

- ❑ Extreme example: recursion (반복)

```
int fact(int n)                // n! = n * (n-1)!  
{  
    if (n < 1) return 1;        // 0! = 1  
    else return (n * fact(n - 1));  
}
```

- Argument n in $\$a0$, result in $\$v0$
- No local variables
- Not use $\$s$

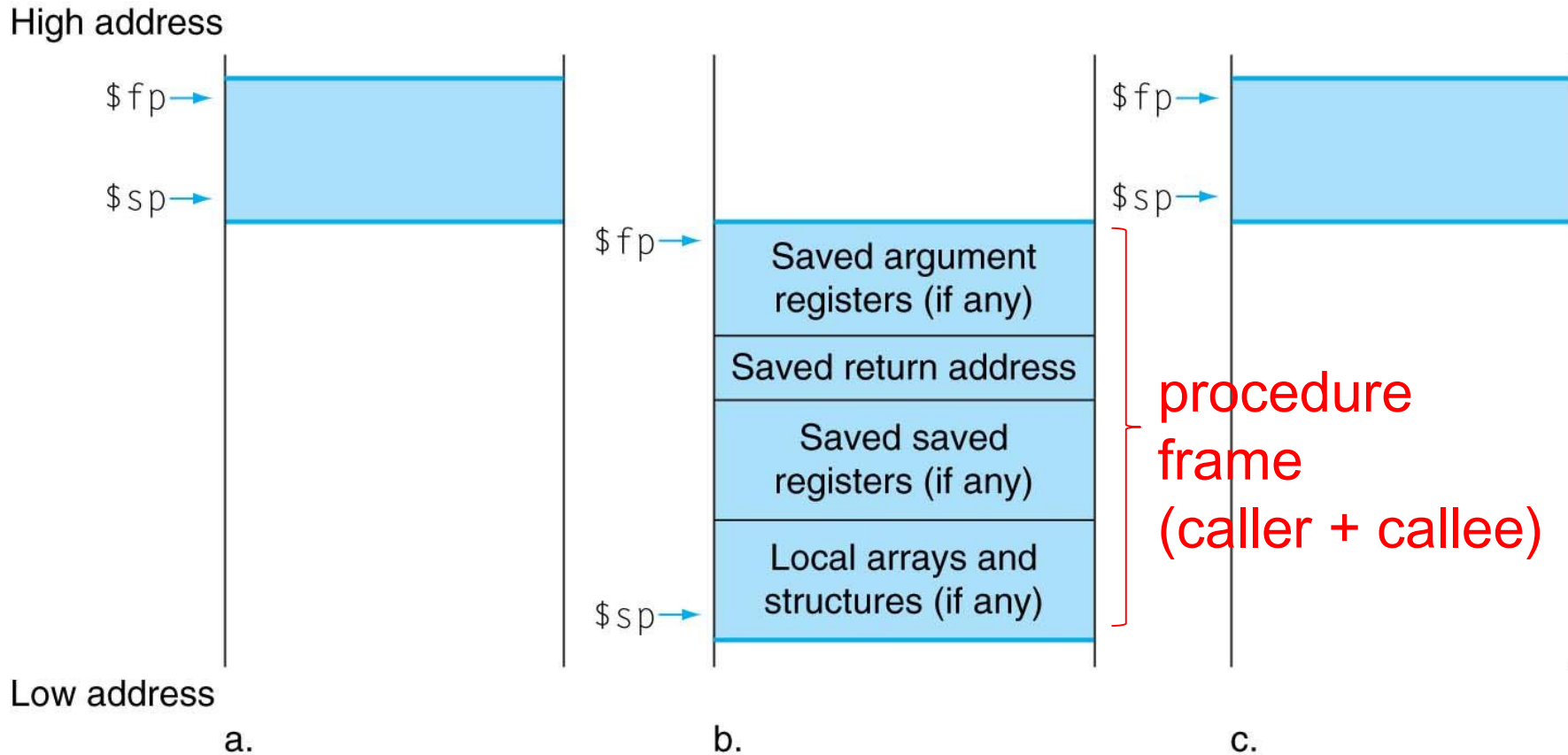
Recursion: fact(n)

❑ MIPS code:

fact:		
addi	\$sp, \$sp, -8	# adjust stack for 2 items
sw	\$ra, 4(\$sp)	# save return address
sw	\$a0, 0(\$sp)	# save argument
slti	\$t0, \$a0, 1	# test for n < 1
beq	\$t0, \$zero, L1	
addi	\$v0, \$zero, 1	# if so, result is 1
addi	\$sp, \$sp, 8	# pop 2 items from stack
jr	\$ra	# and return
L1:	addi \$a0, \$a0, -1	# else decrement n
	jal fact	# recursive call
lw	\$a0, 0(\$sp)	# restore original n
lw	\$ra, 4(\$sp)	# and return address
addi	\$sp, \$sp, 8	# pop 2 items from stack
mul	\$v0, \$a0, \$v0	# multiply to get result
jr	\$ra	# and return

Runtime Stack (Fig. 2.12), (반복)

before/during/after Procedure Call



□ Compiler knows the size of each area

Computing “fact(3)”

$$n! = n * (n - 1)!$$

Register file

\$a0
\$ra

- ❑ No local variables
- ❑ Use \$t (not \$s)

main	
fact(3)	\$a0 = 3 \$ra
fact(2)	\$a0 = 2 \$ra
fact(1)	\$a0 = 1 \$ra
fact(0)	\$a0 = 0 \$ra

$$\$v0 = 3 * 2$$

$$\$v0 = 2 * 1$$

$$\$v0 = 1 * 1$$

$$\$v0 = 1 = 0!$$

Procedure Call Instructions (복습)

- ❑ Procedure call: jump and link

`jal ProcedureLabel`

- Put the address of following instruction in \$ra
- Jump 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

Runtime Environment

Runtime Environment

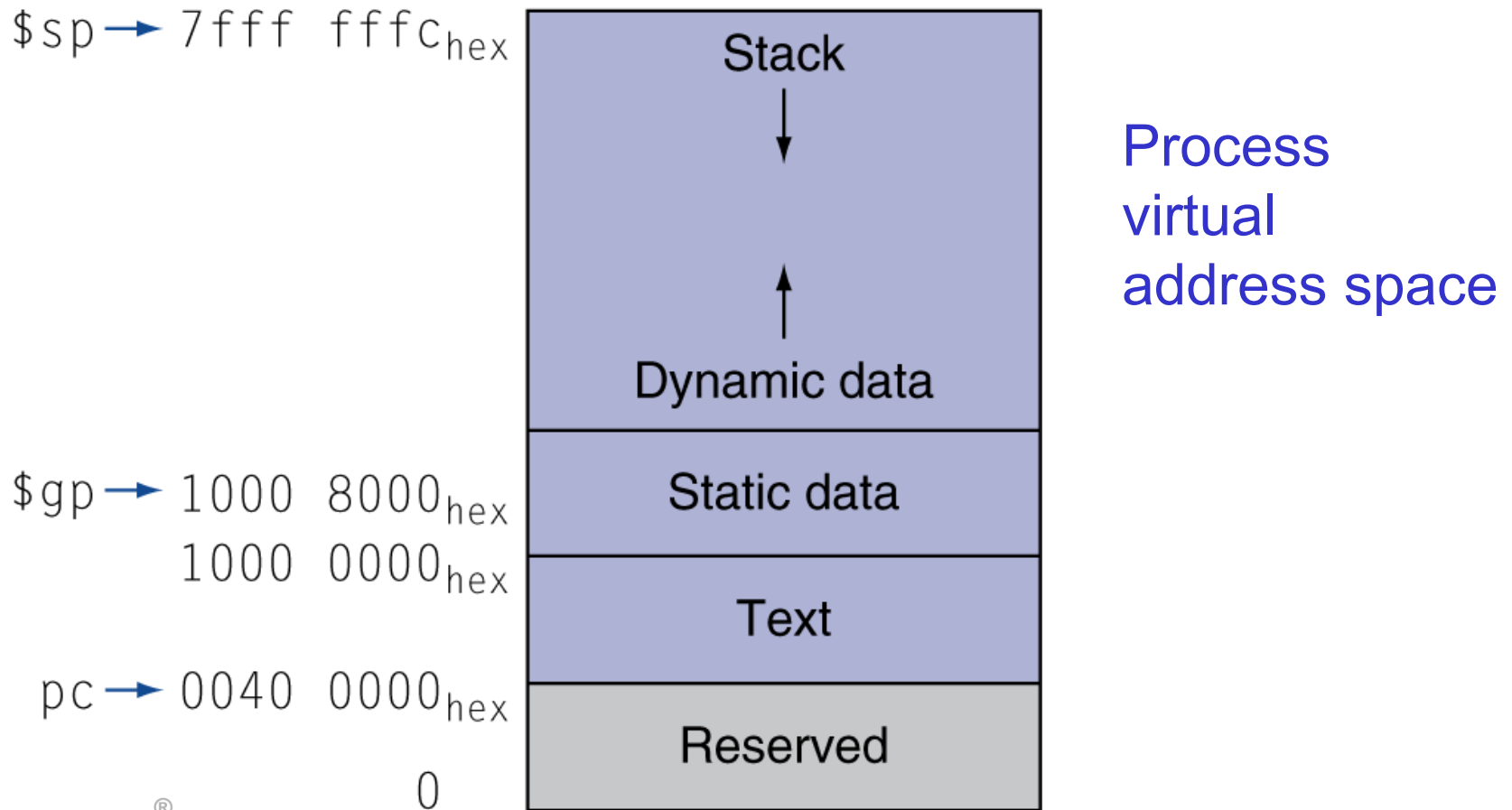
- ❑ Implement *program execution model*
 - How to organize *registers* and *memory* to maintain information needed by executing process
 - Agreement between CPU, OS, and compiler
 - ISA design: many SW-HW interactions
- ❑ Runtime library
 - Standard library, application library
- ❑ What is an environment?
 - Collection of (name, value) pairs

Register Usage (반복)

- ❑ \$a0 – \$a3: arguments (reg's 4 – 7)
- ❑ \$v0, \$v1: result values (reg's 2 and 3)
- ❑ \$t0 – \$t9: temporaries
 - Can be overwritten by callee
- ❑ \$s0 – \$s7: saved
 - 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)

MIPS Memory Layout

- ❑ Static, automatic (runtime stack), dynamic (heap)



MIPS Memory Layout

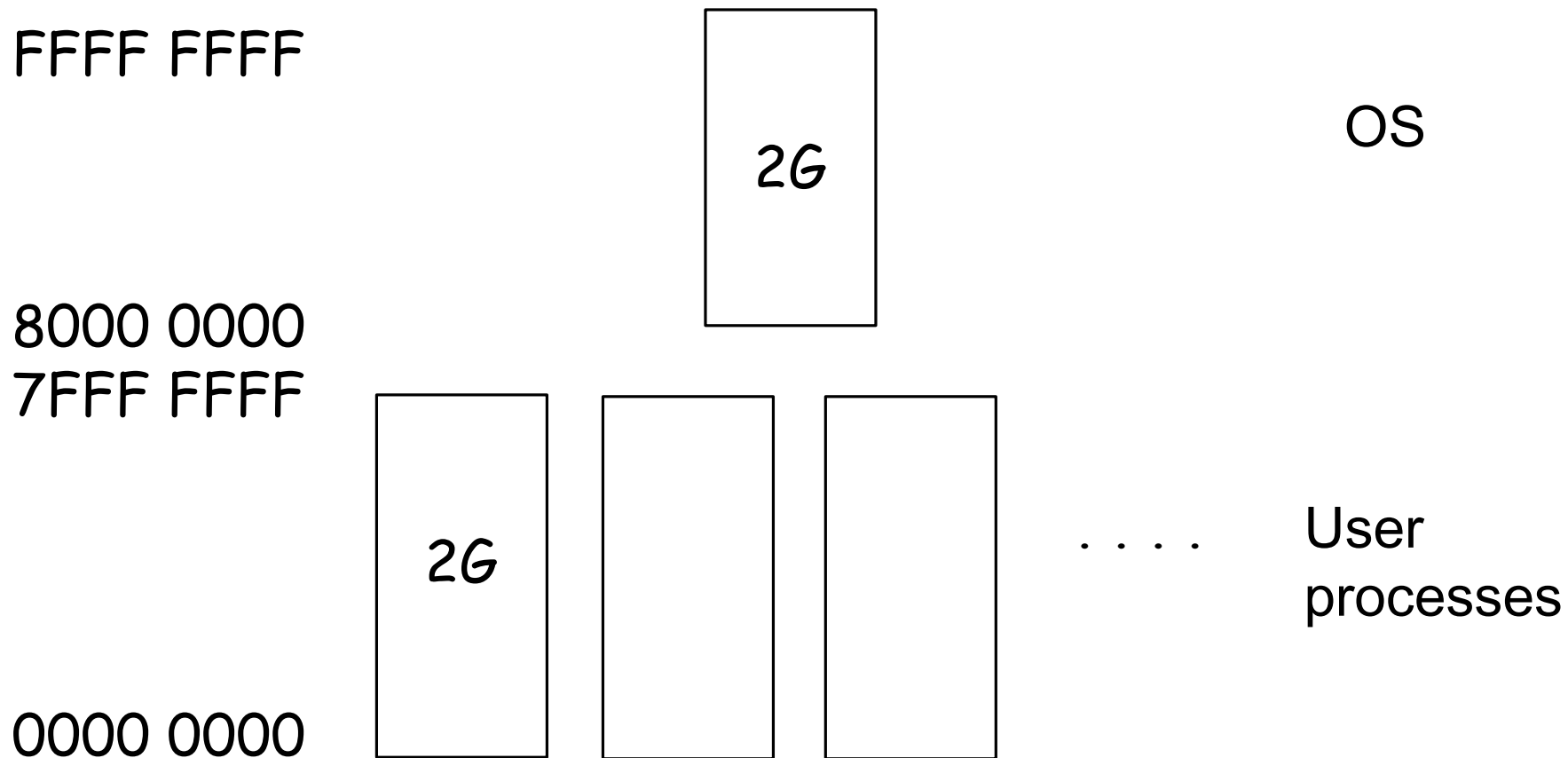
- ❑ Static entities - what's in executable file
- ❑ Meaning of "static" and static data
 - C external variables (and local static)
 - 64KB ($1000\ 0000_{\text{hex}} \sim 1000\ \text{FFFF}_{\text{hex}}$)
 - Base addressing: `lw $t0, 16-bit-offset($gp)`
 - $\$gp = 1000\ 8000_{\text{hex}}$
 - Why do static data have fixed addresses?
- ❑ Text: (static) program code
 - Size: $2^{28} = 256\text{MB}$ (jump with 26bit word offset)

MIPS Memory Layout

- ❑ What will happen during process creation?
- ❑ Dynamic data: heap
 - e.g., malloc in C, new in Java
- ❑ Stack: automatic storage
- ❑ Object lifetime
 - Static object, local object, heap object
- ❑ Why only 2GB for user process?

Virtual Memory

- ❑ Large space: where are they located?



Virtual Memory

- ❑ What is an alternative?

FFFF FFFF

4G

0000 0000

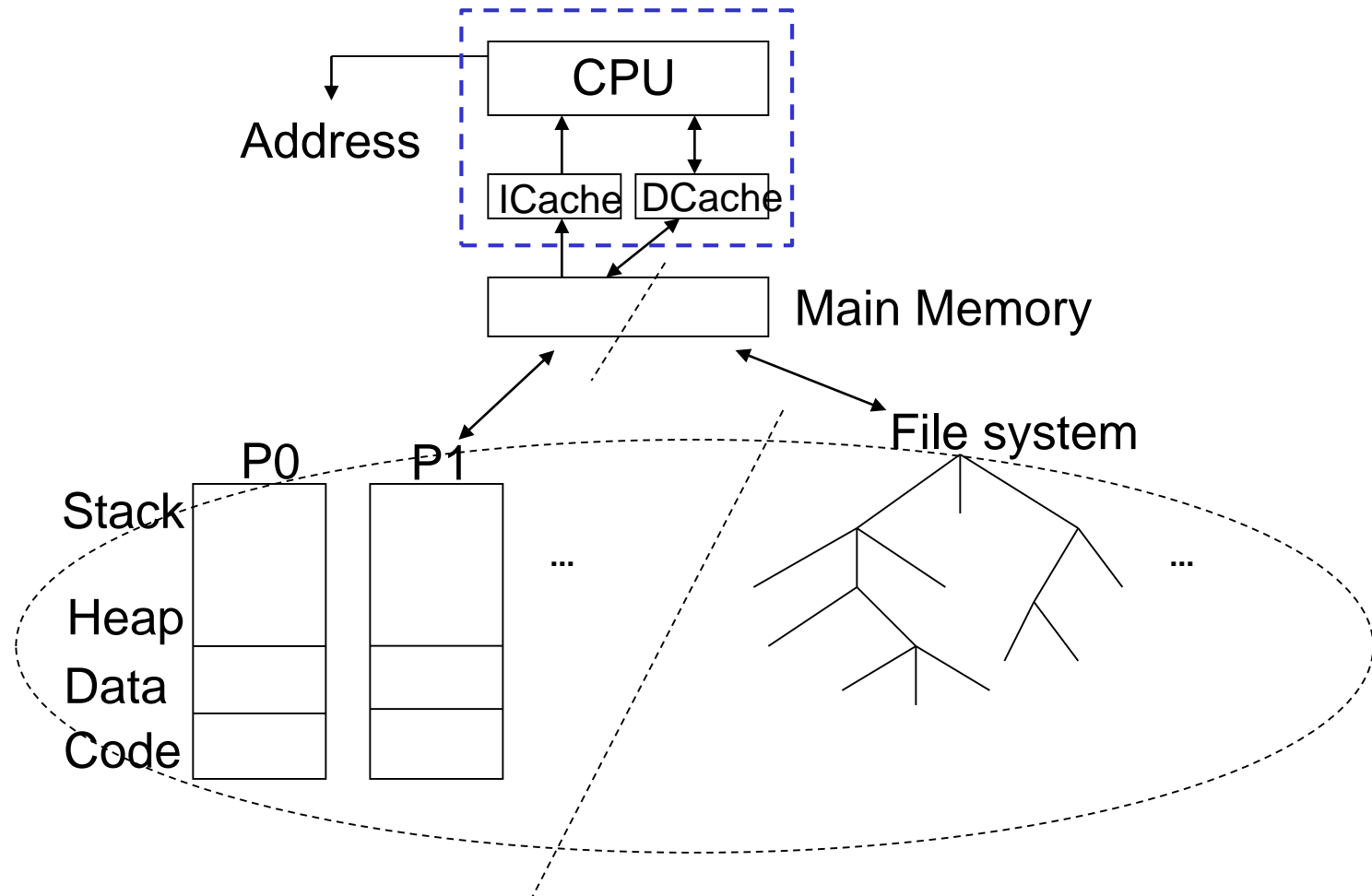
. . . .

OS and
user
processes

Virtual Memory

- ❑ Where is the address space located?
 - How do we use disk space?
 - To support process execution
 - Permanent storage (file systems)
- ❑ Disk access very slow
 - OS (software) use main memory as cache
 - Main memory slower than CPU
 - Use smaller and faster cache memory

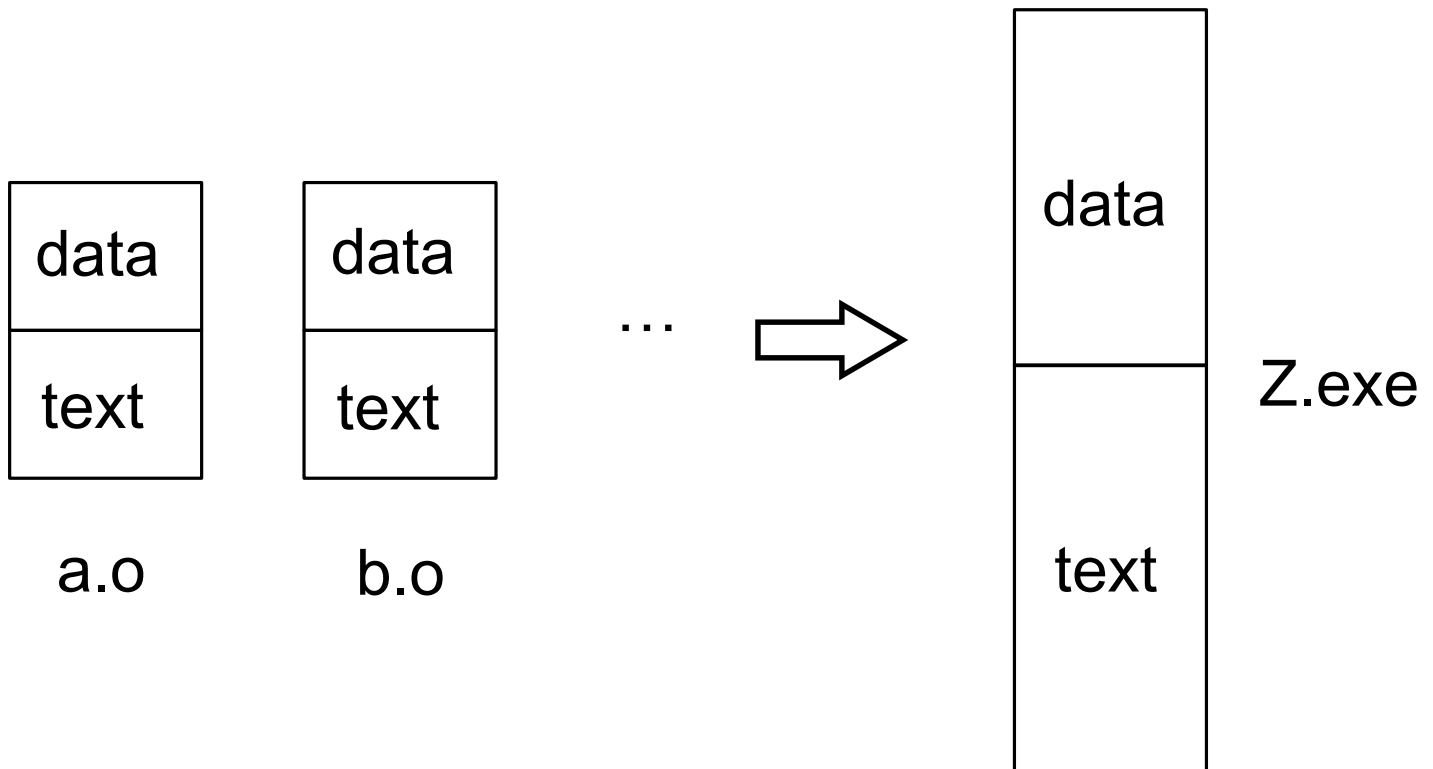
General-Purpose Computers



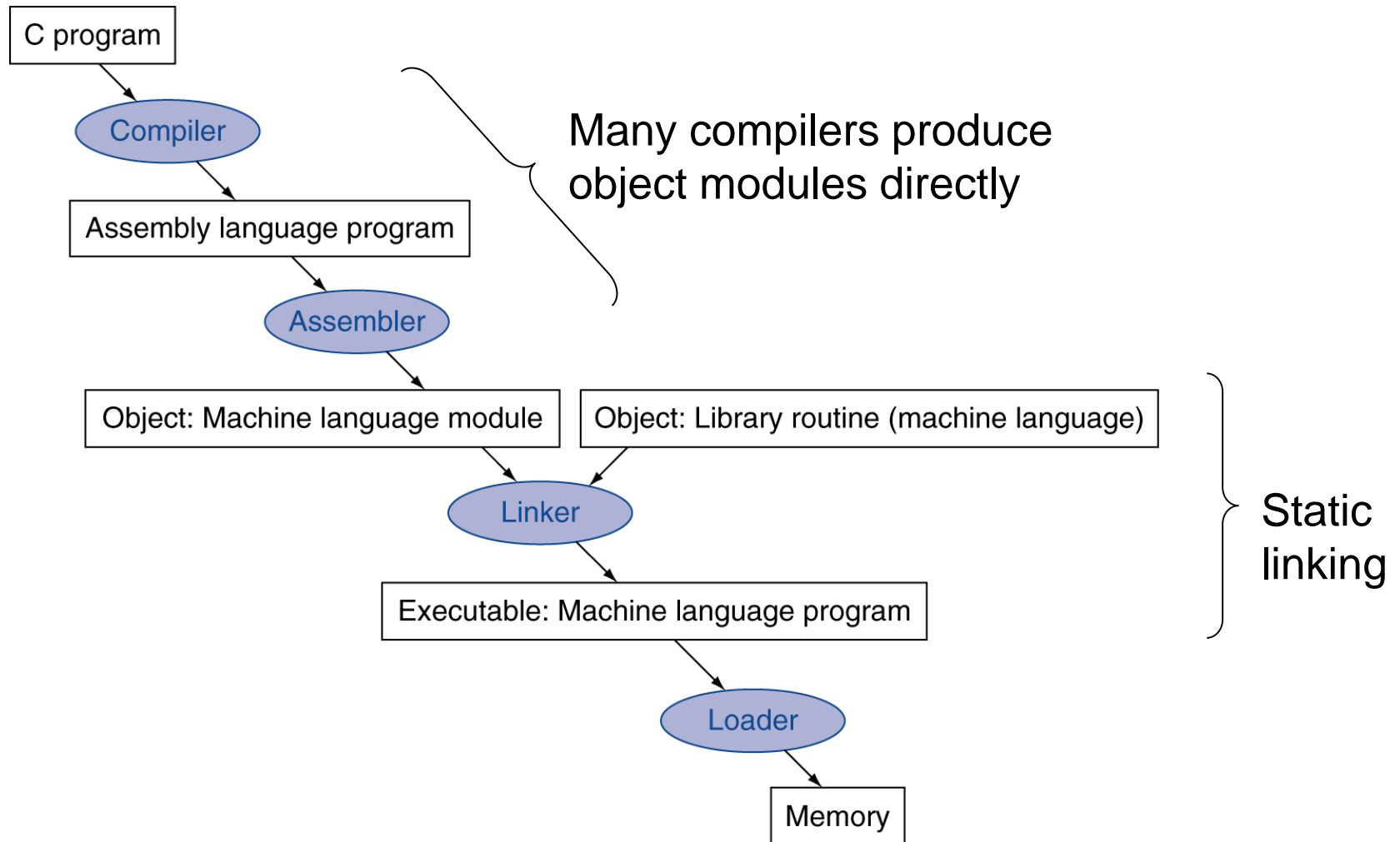
Compile, Link, and Run (1)

Separate Compilation

- ❑ Modular development
- ❑ Linking (or link editing or relocation)



Translation and Startup



Programs to Compile and Link

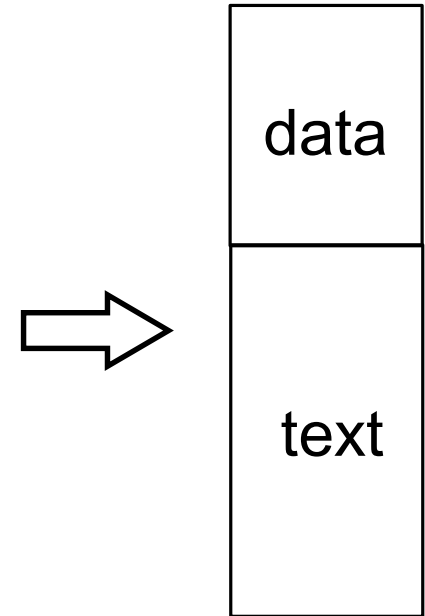
a.src:

```
int X;  
...  
void A(void)  
{  
  B(X);  
  ...  
}  
...
```

b.src:

```
int Y;  
...  
int B(int m, int n)  
{  
  Y = n;  
  A();  
  ...  
} ...  
...
```

Z.exe:



a.o: X

```
...  
lw $a0, "X"($gp)  
jal "B"
```

b.o: Y

```
...  
sw $a1, "Y"($gp)  
jal "A"
```

External symbols'
addresses not
known
at compile time



Linking Object Files (a.o)

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	—	

Linking Object Files (b.o)

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	—	

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 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 Modules

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

Linking Object Files (Z.exe)

Executable file header		
	Text size	300 _{hex}
	Data size	50 _{hex}
Text segment	Address	Instruction
(function A)	0040 0000 _{hex}	lw \$a0, 8000 _{hex} (\$gp)
	0040 0004 _{hex}	jal 40 0100 _{hex}

(function B)	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)

Linking Object Files

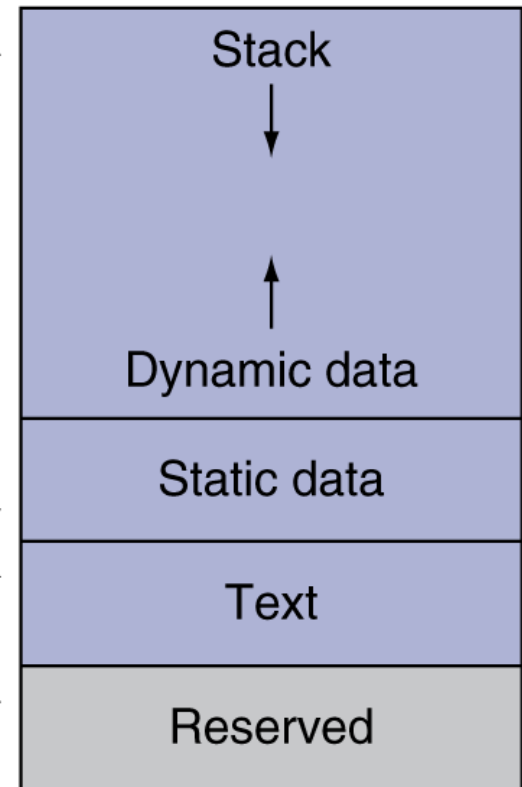
❑ “lw \$a0, 8000_{hex}(\$gp)”

```
$gp:  1000 8000hex
      FFFF 8000hex
      -----
      1000 0000hex
```

\$sp → 7fff fffc_{hex}

\$gp → 1000 8000_{hex}
 1000 0000_{hex}

pc → 0040 0000_{hex}
 0



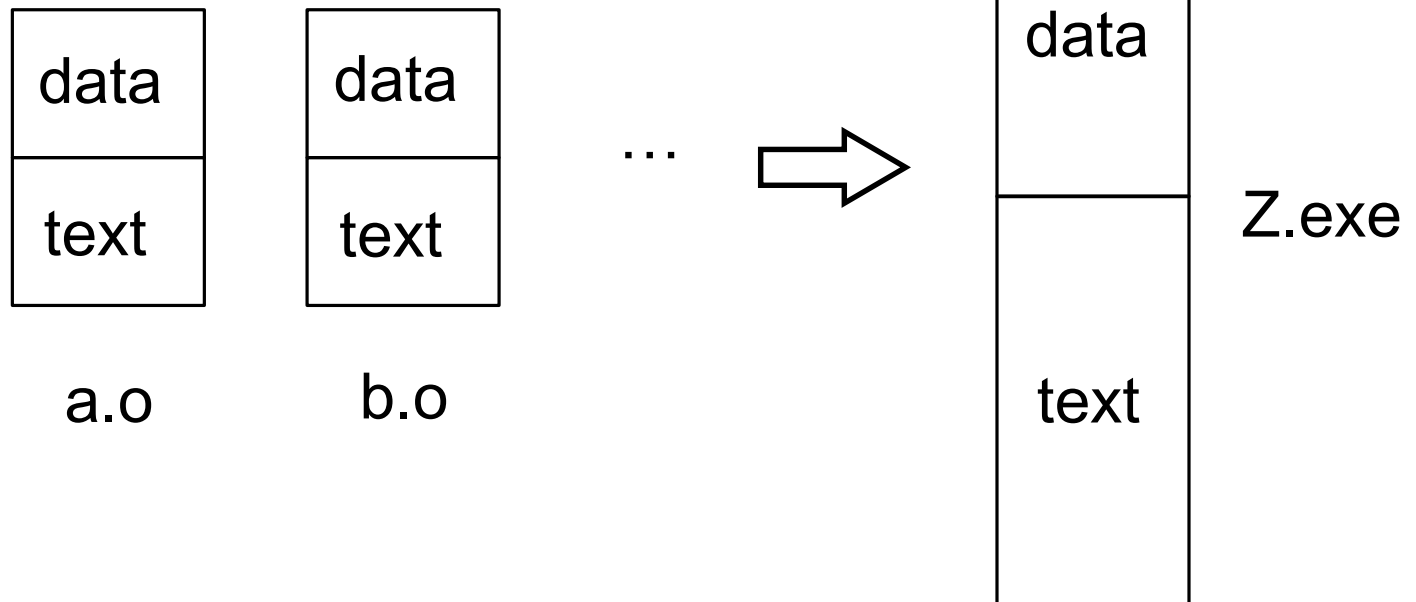
Loading a Program

- ❑ 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

Compile, Link, and Run (2)

To Think about

- ❑ Compile and link
- ❑ Static vs. dynamic linking



Static vs. Dynamic Linking

“hello.c”

```
#include <stdio.h>

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

- ❑ Compile and link (your system has “libc.a” and libc.so”)

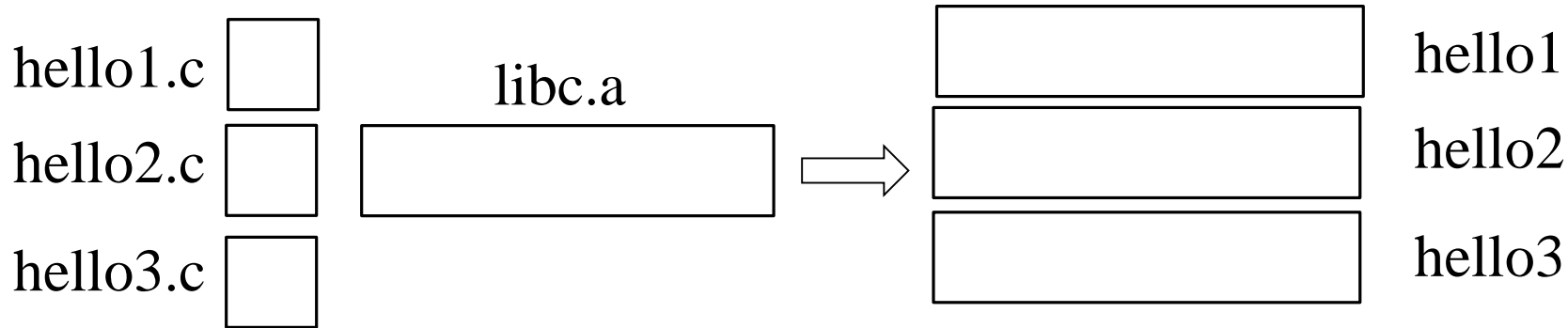
```
$ gcc hello.c -o hello
```

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

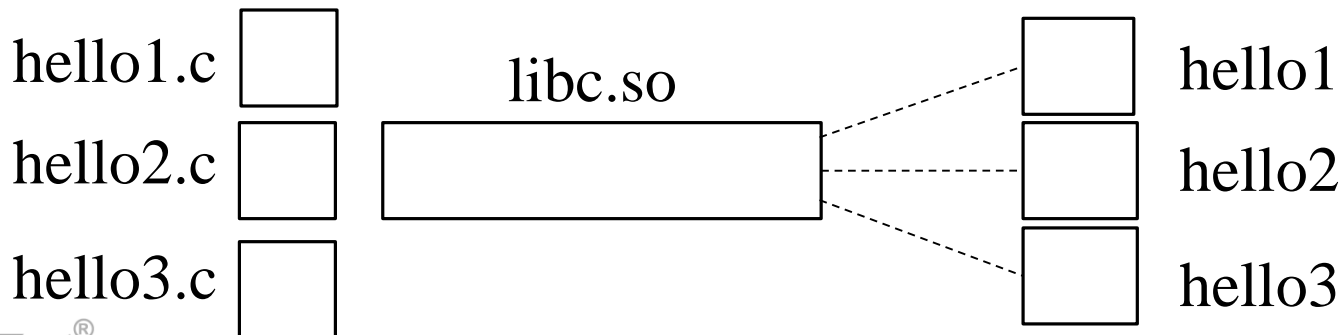
Static vs. Dynamic Linking

❑ Think about: `hello1.c`, `hello2.c`, `hello3.c`, ...

- Static linking

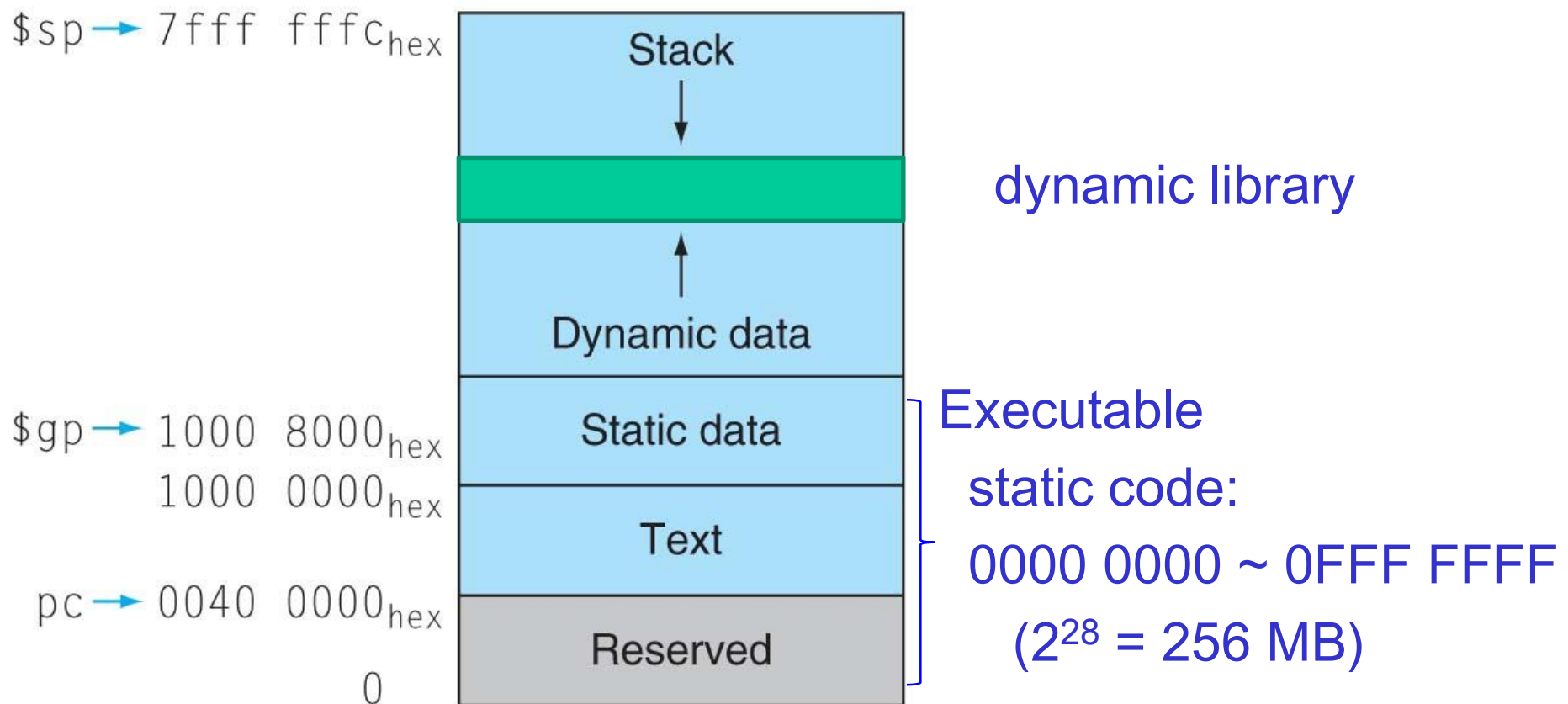


- Dynamic linking (default choice): library is shared object
 - Linking at program start or on actual call



MIPS Memory Layout (반복)

Figure 2.13 MIPS memory allocation for program and data



Dynamic Linking

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

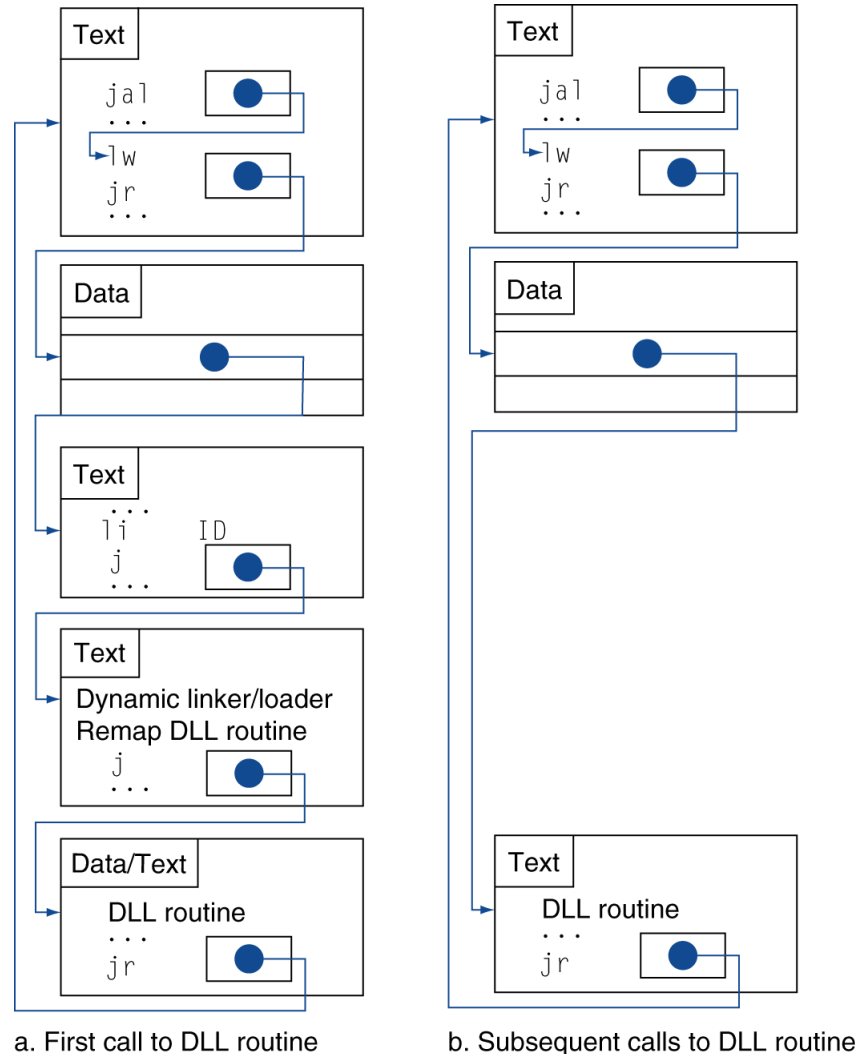
Lazy/Delayed Linkage (skip)

Indirection table

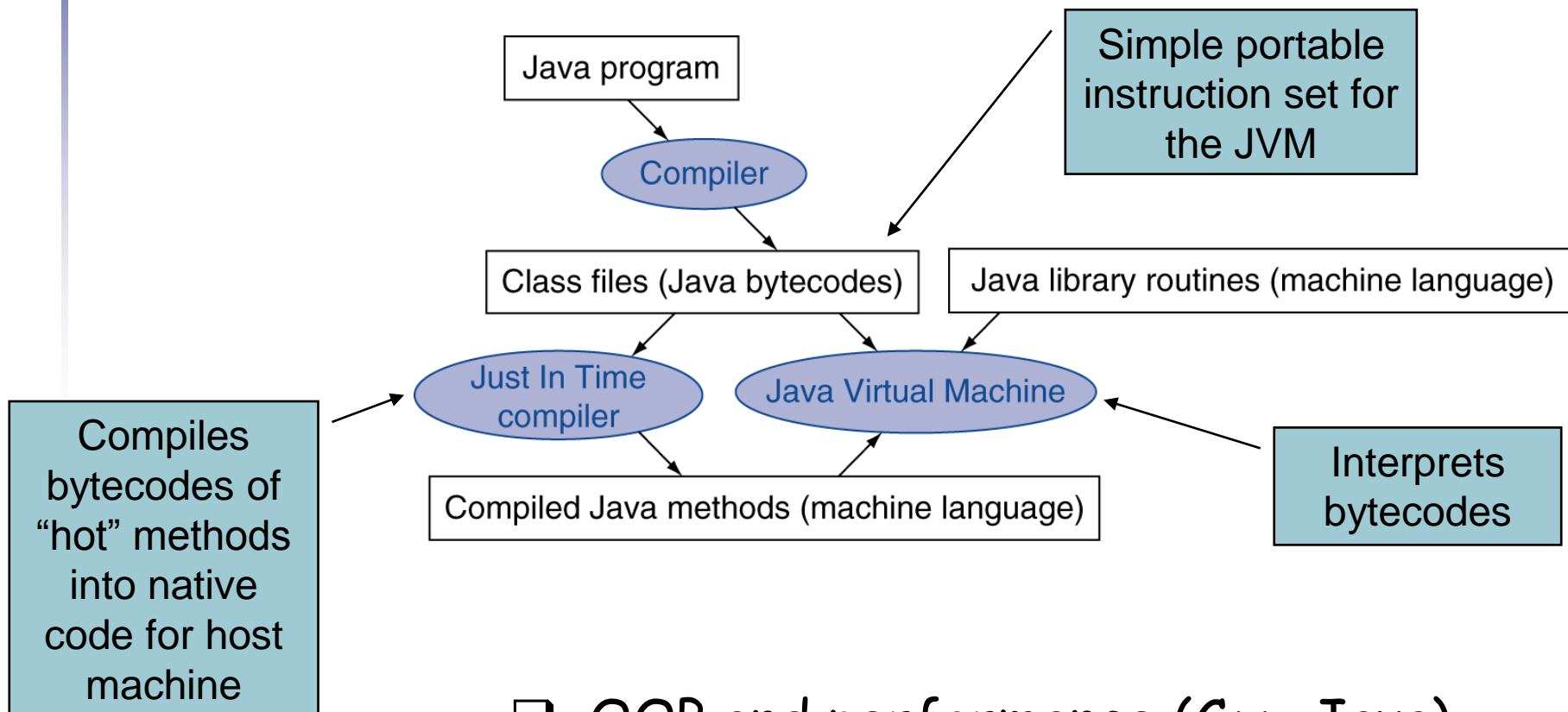
Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

Dynamically
mapped code

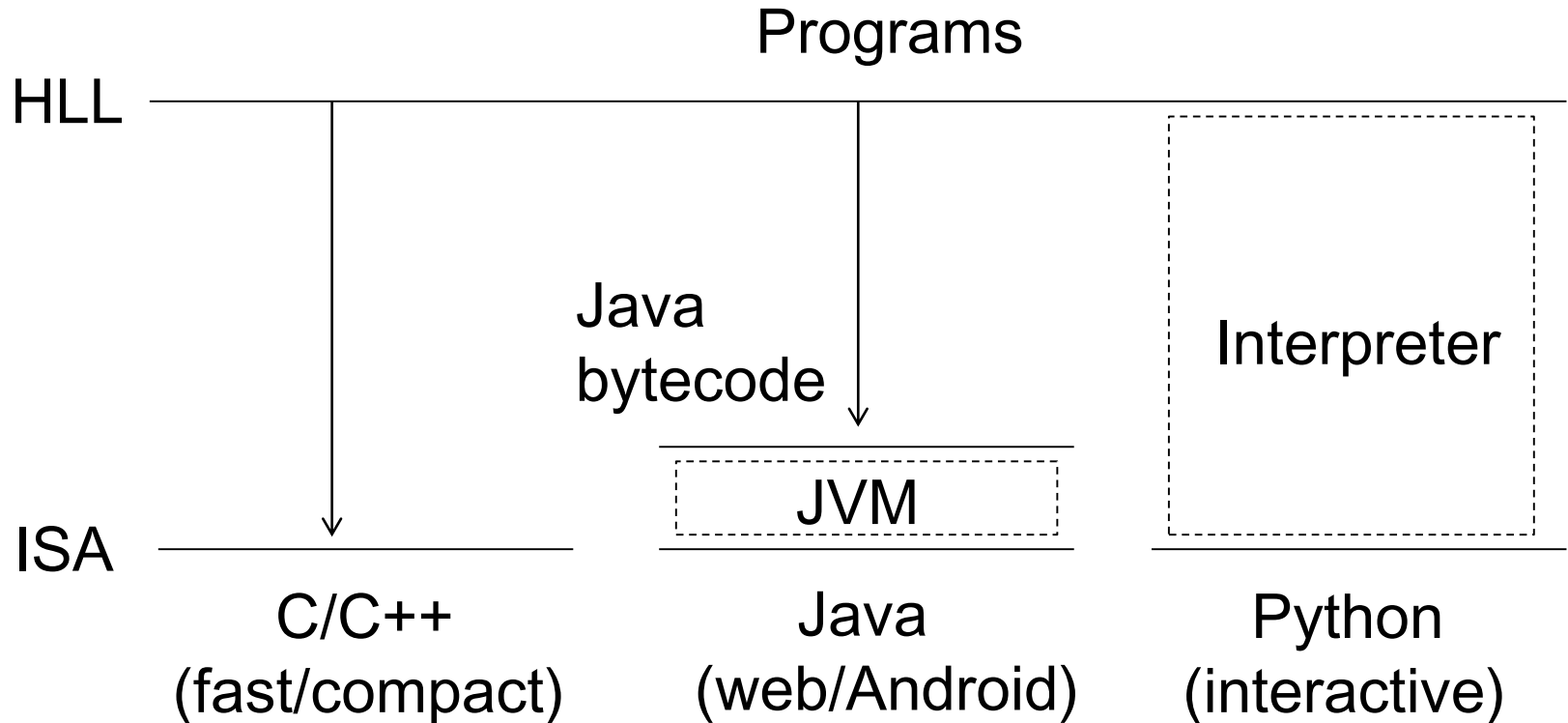


Starting Java Applications



- ❑ OOP and performance (C++, Java)
- ❑ Notion of profiling, linking in java

Program Execution



- ❑ Compiler vs. interpreter approach
 - Native vs. non-native mode (Python machine learning?)
- ❑ Four major programming languages

Compile, Link, and Run (3)

Compiler Optimization

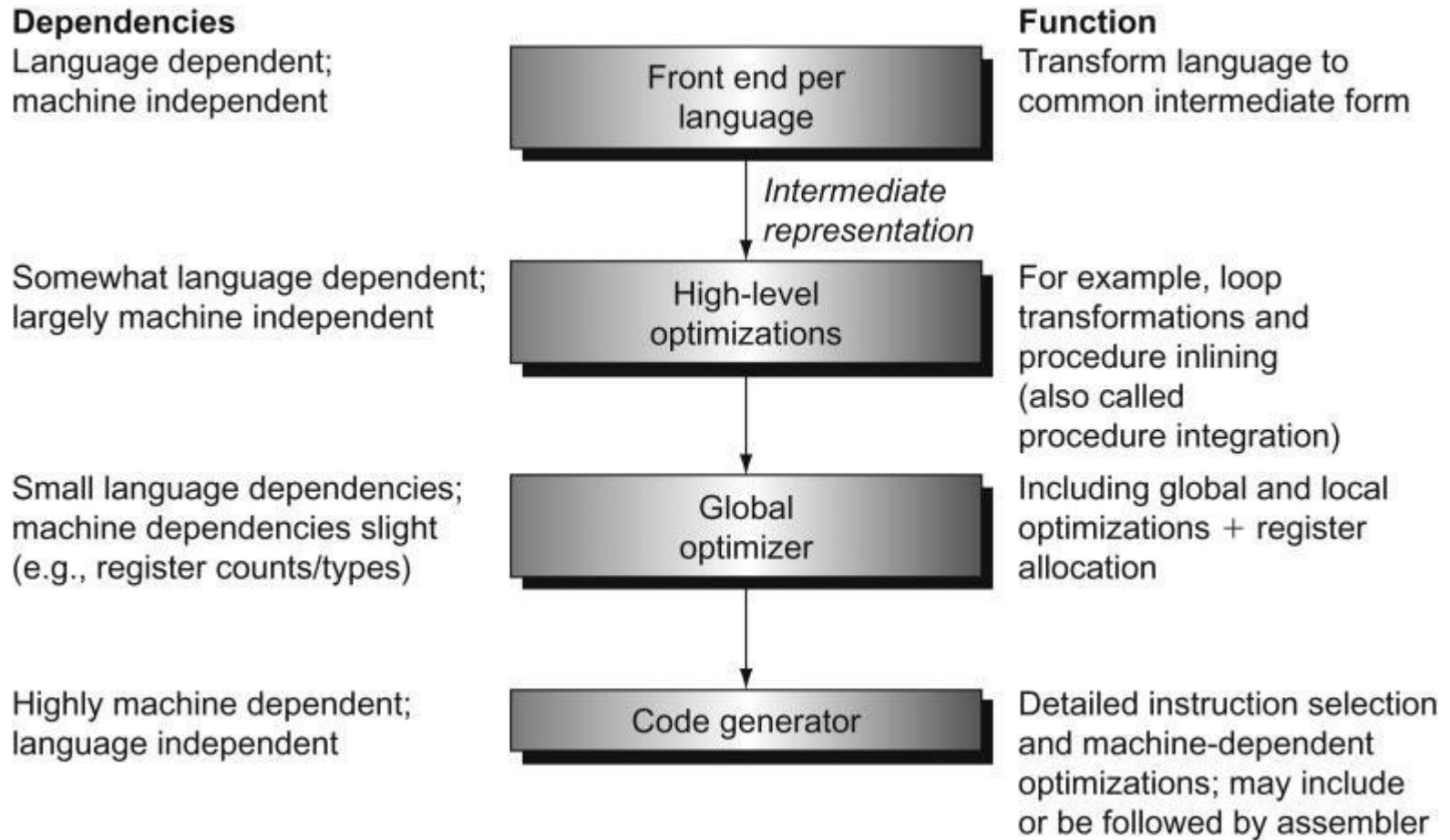
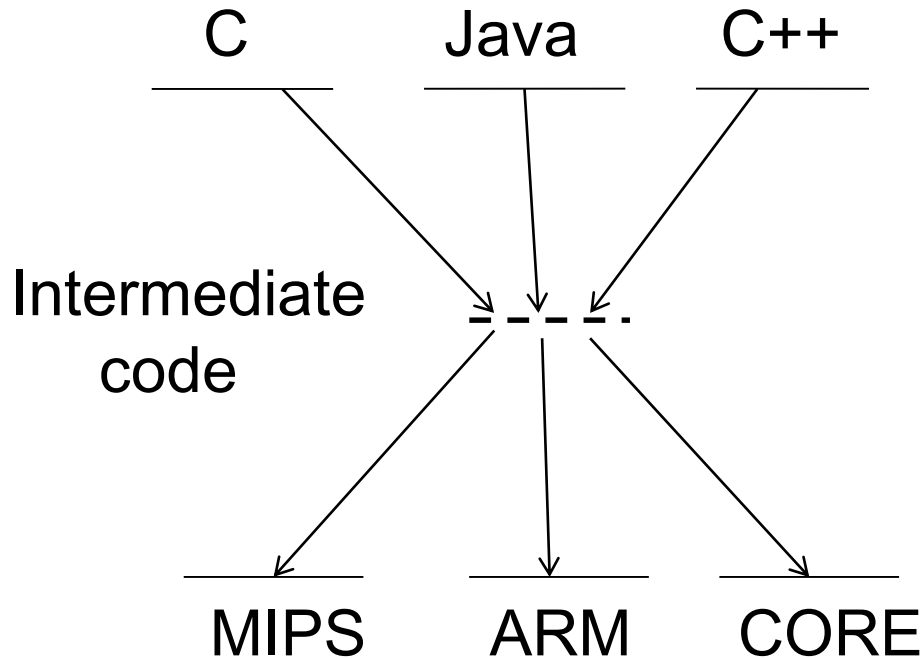


Figure 2.15.1 The structure of a modern optimizing compiler

Compilers



Front-end (parsing)

- Language dependent

Back-end (optimization)

- Machine dependent
- RISC since 1980s

❑ Parsing algorithms

- Automata theory and formal languages (grammars)

❑ Compiler support for GPU and NPU

“gcc” Optimization Level

❑ gcc -O0, -O1, -O2, -O3

Optimization name	Explanation	gcc level
<i>High level</i>	<i>At or near the source level; processor independent</i>	
Procedure integration	Replace procedure call by procedure body	O3
<i>Local</i>	<i>Within straight-line code</i>	
Common subexpression elimination	Replace two instances of the same computation by single copy	O1
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	O1
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	O1
<i>Global</i>	<i>Across a branch</i>	
Global common subexpression elimination	Same as local, but this version crosses branches	O2
Copy propagation	Replace all instances of a variable A that has been assigned X (i.e., A = X) with X	O2
Code motion	Remove code from a loop that computes same value each iteration of the loop	O2
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	O2
<i>Processor dependent</i>	<i>Depends on processor knowledge</i>	
Strength reduction	Many examples; replace multiply by a constant with shifts	O1
Pipeline scheduling	Reorder instructions to improve pipeline performance	O1
Branch offset optimization	Choose the shortest branch displacement that reaches target	O1

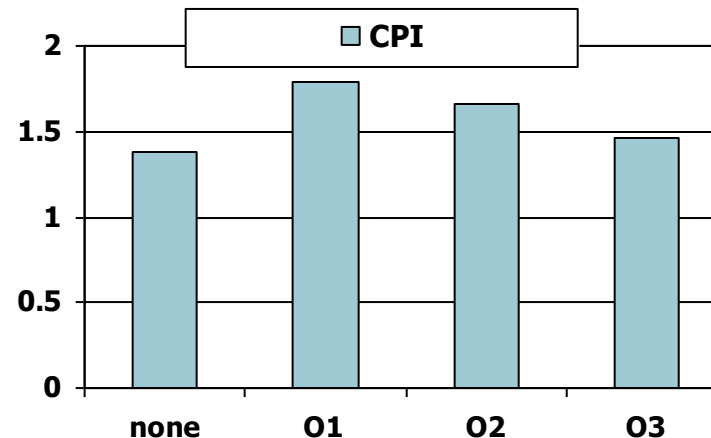
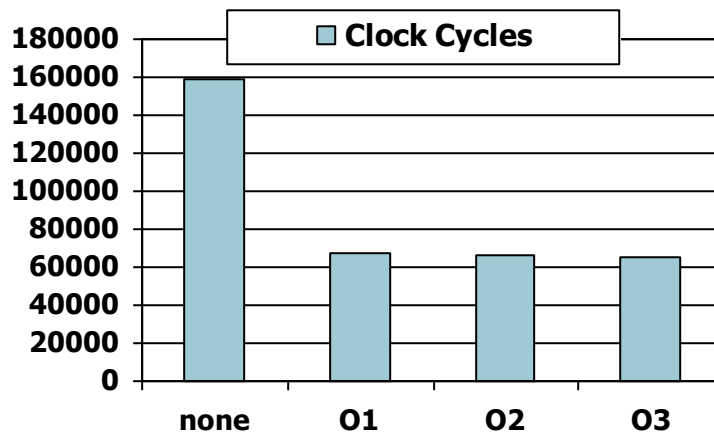
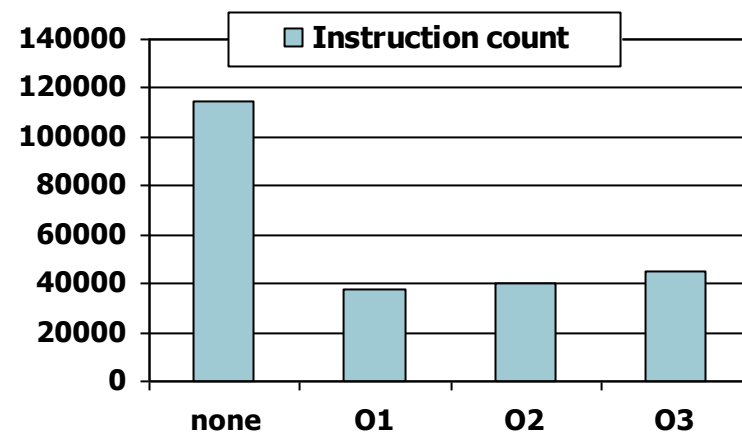
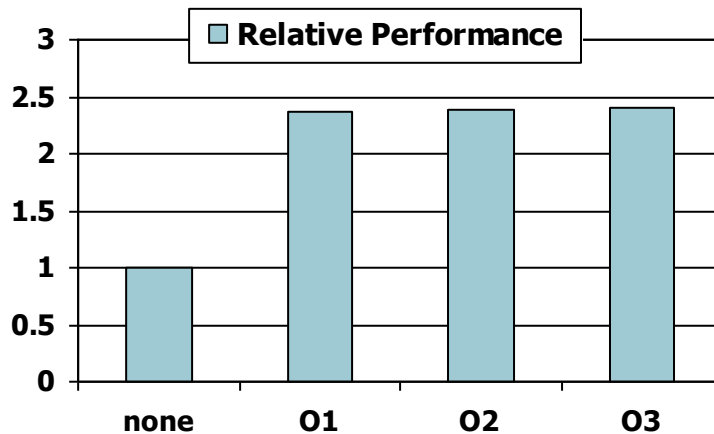
Figure 2.15.7 Major types of optimizations



Effect of Compiler Optimization

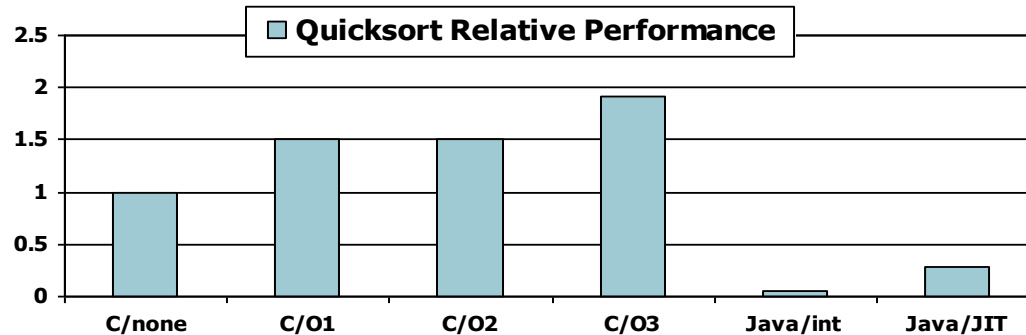
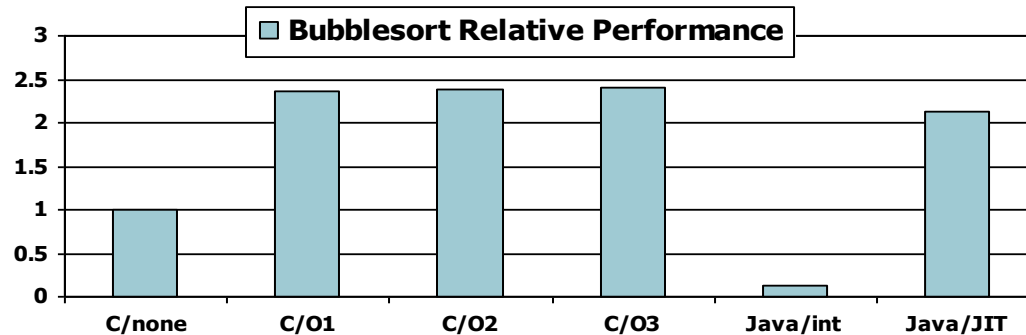
Bubblesort compiled with gcc for Pentium 4 under Linux

(cct 고정)

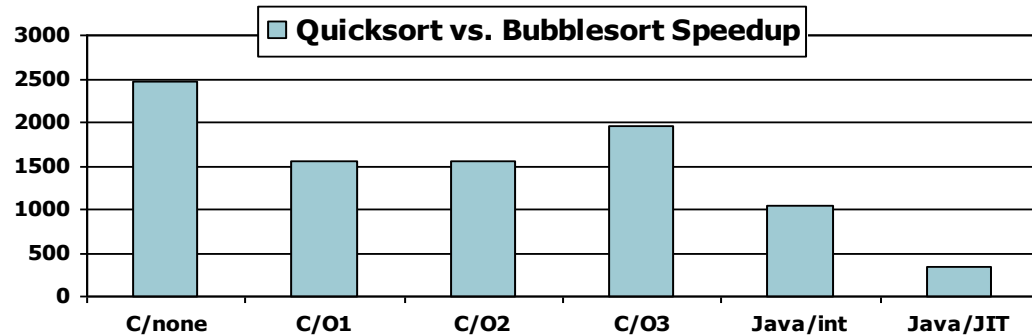


□ Use smart compiler to reduce IC

Effect of Language and Algorithm



(프로그램
고유 특성)



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!

Homework #9 (see Class Homepage)

1) Write a report summarizing the materials discussed in Topic 2-3 (이번 주 수업 내용)

** 문장으로 써도 좋고 파워포인트 형태의 개조식 정리도 좋음

2) Solve Chapter 2 exercises 1, 2, 3, 4, 10, 11, 12, 18, 20, 24, 26, 39, 40, 41, 42, 47

** 위의 문제들 중에서 Homework #8 에서 제출하지 않은 문제들은 Homework #9 로 제출하세요.

□ Due: see Blackboard

- Submit electronically to Blackboard

Class Topics (클래스 홈페이지 참조)

- ❑ Part 1: Fundamental concepts and principles
- ❑ Part 2: 빠른 컴퓨터를 위한 ISA design
 - Topic 1 Computer performance and ISA design (Ch. 1)
 - Topic 2 RISC (MIPS) instruction set (Chapter 2)
 - 2-1 ALU and data transfer instructions
 - 2-2 Branch instructions
 - 2-3 Supporting program execution
 - Topic 3 Computer arithmetic and ALU (Chapter 3)
- ❑ Part 3: ISA 의 효율적인 구현 (pipelining, cache memory)