컴퓨터 공학 개론
Lecture 6

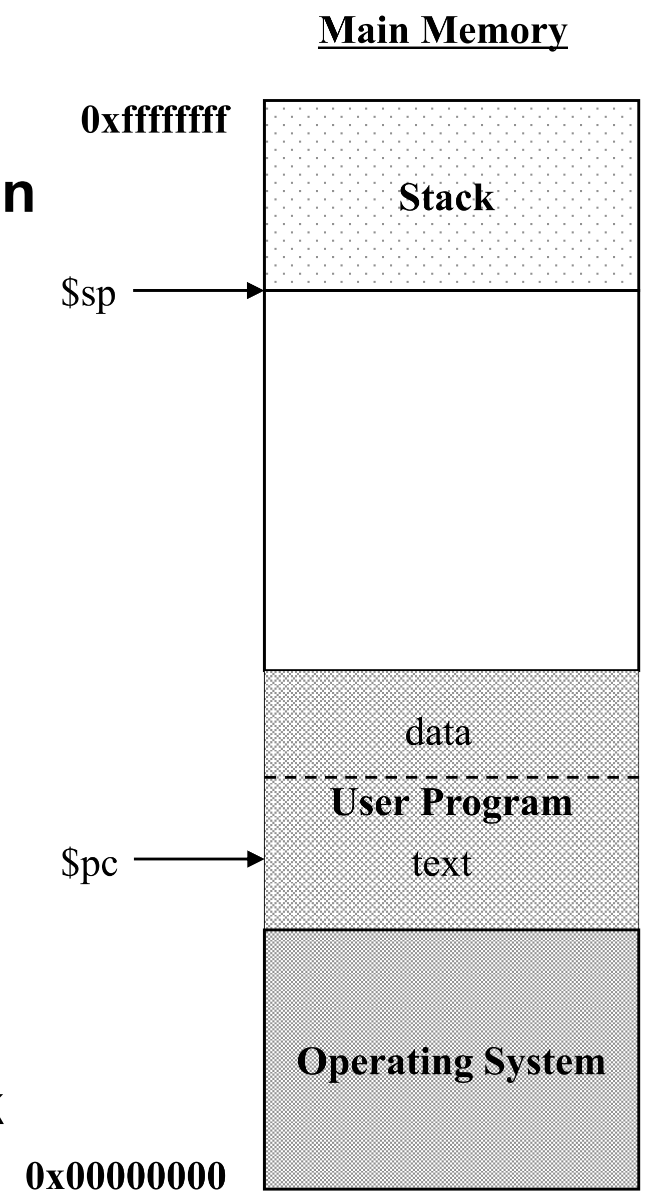
Instructions: Language of the Computer

2017

김태성

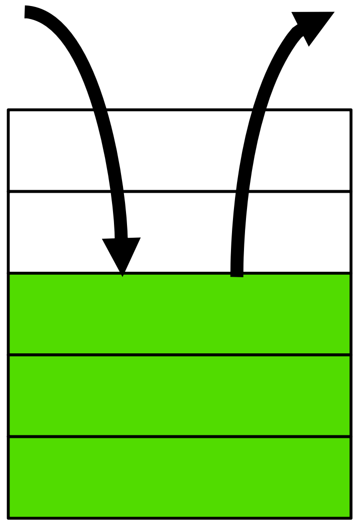
Memory Allocation

- $pc
  - Program Counter
  - register for the address of current instruction
- Stack
  - a last-in-first-out(LIFO) data structure
  - memory reserved for spilling registers
  - grows downward
  - *push: place data onto the stack*
  - *pop: remove data from the stack*
- $sp
  - Stack Pointer
  - register for the stack
  - most recently allocated address in the stack

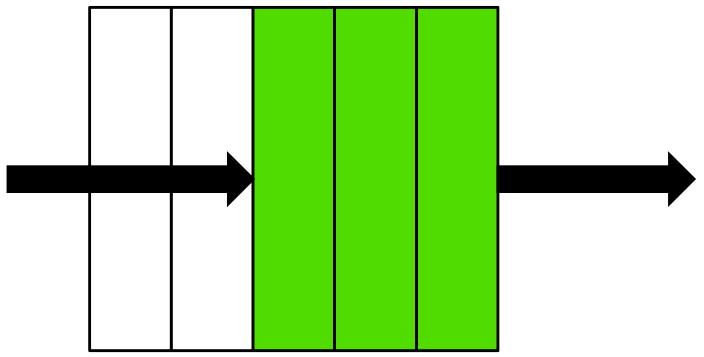


Stack vs. FIFO

- Stack
  - Last-In-First-Out(LIFO) data structure



- FIFO
  - First-In-First-Out data structure



Procedure Calls

° <u>**In the execution of a procedure, the program must follow these six steps:**</u>

1. Place parameters in a place where the procedure can access them

2. Transfer control to the procedure

Caller side

3. Acquire the storage needed for the procedure

4. Perform the procedure operations

5. Place the result value in a place where the calling program can access it

6. Return control to the caller

Callee side

```
main(…){
…
…
flag = bar(a, b);
….
}
…

int bar(x,y){
…
…
return(result);
}
```

Procedure Calls

1. Place parameters in a place where the procedure can access them
    - $a0 - $a3: argument registers to pass parameters
2. Transfer control to the procedure
    - $ra: register for return address
    - jal ProcedureAddress   # saves the address of the next instruction in $ra
      # and jump to “ProcedureAddress”
3. Acquire the storage needed for the procedure
    - save registers into stack
4. Perform the procedure operation

5. Place the result value in a place where the calling program can access it
    - $v0 - $v1: two registers for return value
6. Return control to the caller
    - restore saved registers
    - jr $ra   # ‘jump register’: jump to the address stored in $ra

Calling a Procedure

° C Program:

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

° MIPS Assembly Program:

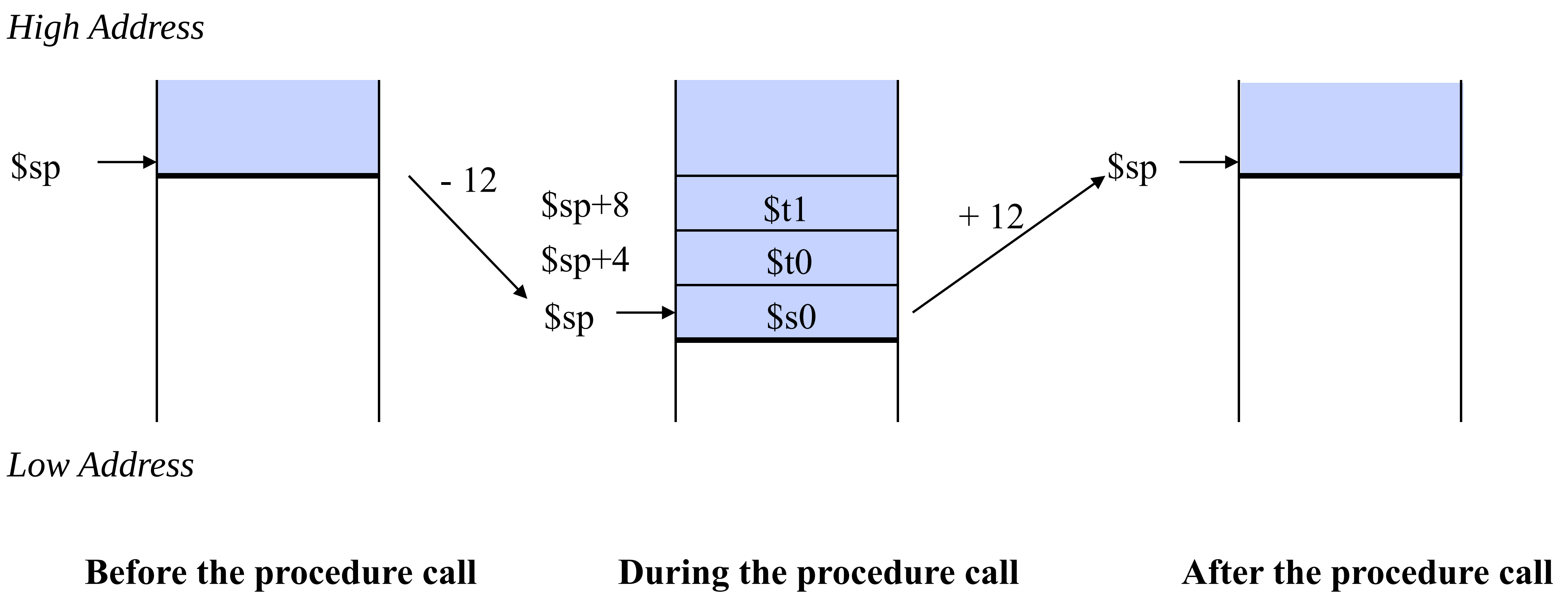
```
                        # step 1, 2 are executed on the caller side
                        # compiler assigned $a0, $a1, $a2, $a3, and $s0
                        # for  g, h, i,  j, and f respectively
bar:                    # procedure starts here executing step 3
addi $sp, $sp, -12      # adjust stack to make room for 3 items
sw $t1, 8($sp)          # save register $t1 for use afterwards
sw $t0, 4($sp)          # save register $t0 for use afterwards
sw $s0, 0($sp)          # save register $s0 for use afterwards
```

Calling a Procedure (cont.)

```
                        # step 4: main body of the procedure
add $t0, $a0, $a1       # register $t0 contains ( g + h )
add $t1, $a2, $a3       # register $t1 contains ( i + j )
sub $s0, $t0, $t1       # f gets ( $t0 - $t1 ), which is ( g + h ) - ( i + j )

add $v0, $s0, $zero            # step 5: returns f ( $v0 = $s0 + 0 )
                        # step 6:
lw $s0, 0($sp)          # restore register $s0 for caller
lw $t0, 4($sp)          # restore register $t0 for caller
lw $t1, 8($sp)          # restore register $t1 for caller
addi $sp, $sp, 12       # adjust stack to delete 3 items
jr $ra                  # jumb back to calling routine
```

Stack Movement



Nested Procedures

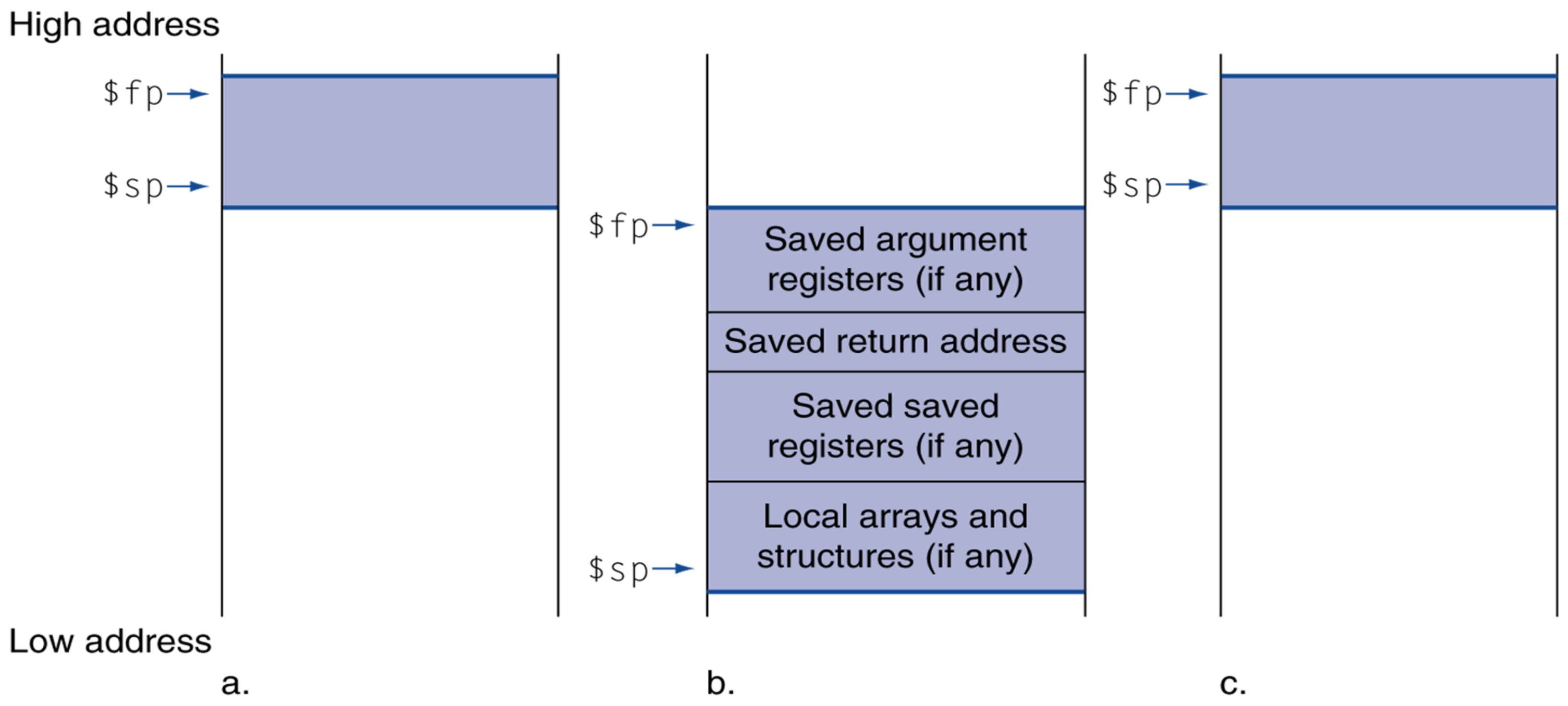
° Procedures that call other procedures

° For nested call, caller needs to save on the stack:

  - Its return address ($ra)
  - Any arguments and temporaries needed after the call ($a0~$a3)

° Restore from the stack after the call

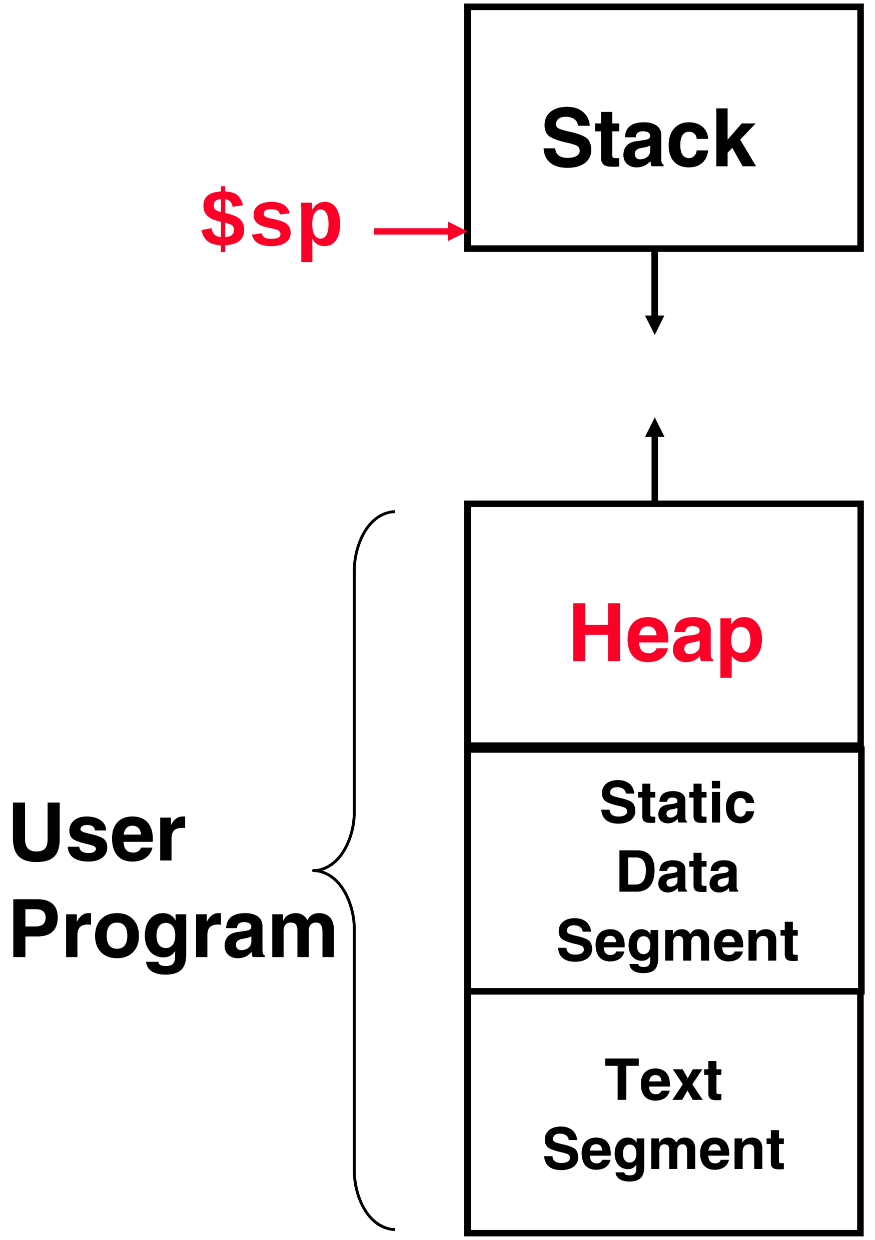
Local Data on the Stack



High address
$fp
$sp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
Low address
a.
b.
c.

- Local data allocated by callee
  - e.g., C automatic variables (<-> external variables)
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage
  - $fp: frame pointer

Inside of User Program



Space for saved procedure information e.g. register values

Dynamic data: e.g., malloc(); C pointers

Static variables and constants

Machine code

MIPS Register Summary

Name	Register Number	Usage
$zero	**0**	**The constant value 0**
$at	*1*	*Reserved for assembler*
$v0 - $v1	**2-3**	**Return value**
$a0 - $a3	**4-7**	**Arguments**
$t0 - $t7	**8-15**	**Temporaries**
$s0 - $s7	**16-23**	**Static variables**
$t8 - $t9	**23-25**	**More temporaries**
$k0 – $k1	*26-27*	*Reserved for OS kernel*
$gp	*28*	*Global pointer*
$sp	**29**	**Stack pointer**
$fp	**30**	**Frame pointer**
$ra	**31**	**Return address**

Kilo, Mega, Giga, Tera, Peta, Exa, Zetta, Yotta

Name	Abbr	Factor	SI (International System of Units) size
Kilo	K	**2^{10} = 1,024**	10^{3} = 1,000
Mega	M	2^{20} = **1,048,576**	10^{6} = 1,000,000
Giga	G	2^{30} = **1,073,741,824**	10^{9} = 1,000,000,000
Tera	T	2^{40} = **1,099,511,627,776**	10^{12} = 1,000,000,000,000
Peta	P	2^{50} = **1,125,899,906,842,624**	10^{15} = 1,000,000,000,000,000
Exa	E	2^{60} = **1,152,921,504,606,846,976**	10^{18} = 1,000,000,000,000,000,000
Zetta	Z	2^{70} = **1,180,591,620,717,411,303,424**	10^{21} = 1,000,000,000,000,000,000,000
Yotta	Y	2^{80} = **1,208,925,819,614,629,174,706,176**	10^{24} = 1,000,000,000,000,000,000,000,000

Kilo, Mega, Giga, Tera, Peta, Exa, Zetta, Yotta

- Common use prefixes in computer engineering
- Confusing! Common usage of “kilobyte” means 1024 bytes, but the “correct” SI value is 1000 bytes
- 4Gb DRAM = 4 x 2^{30} bits = 4,294,967,296 bits
- 1.3GB AVI file = 1.3 x 2^{30} Bytes = 1,395,864,371 Bytes
- Hard Disk manufacturers & Telecommunications use SI factors, so what is advertised as a 300 GB drive will actually only hold about 280 x 2^{30} bytes, and a 1 Mbit/s connection transfers 10^6 bps.

Handling Characters

ASCII: American Standard Code for Information Interchange

Character	ASCII Value
…	…
?	63 (011 1111)
@	64 (100 0000)
A	65 (100 0001)
B	66 (100 0010)
…	…

- **How to Represent Characters?**
  - **English alphabet: 26 letters**
  - **upper/lower case + symbols + formatting (< 128 char.)**

    => can be represented in 7 bits ("ASCII")

    => use 1 byte(8 bits) to store one character
- **MIPS instructions for loading/storing character data**
  - **lb: "load byte" loads a byte from memory and place it in the rightmost 8 bits of a register**
    - **e.g. lb $t0, 0($s1)   // read byte from memory**
  - **sb: "store byte" takes a byte from the rightmost 8 bits of a register and writes it to memory**
    - **e.g. sb $t0, 0($s1)   // write byte to memory**

Handling Characters

- Standard code to cover all the world's languages
  - => 8,16,32 bits   ("Unicode", www.unicode.com)
  - 16-bit encoding (UTF-16) is default
    - UTF: Unicode Transformation Formats
  - UTF-8 (variable-length encoding) keeps the ASCII subset as 8 bits and use 16~32 bits for the other characters
  - UTF-32 uses 32 bits per characters

- MIPS instructions for loading/storing 16-bit character data
  - lh: "load half" loads a halfword from memory and place it in the rightmost 16 bits of a register
    - e.g. lh $t0, 0($s1)   // read halfword (16 bits) from memory
  - sh: "store half" takes a halfword from the rightmost 16 bits of a register and writes it to memory
    - e.g. sh $t0, 0($s1)   // write halfword (16 bits) to memory

Constants

- Small constants are used quite frequently (50% of operands)

  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;

- MIPS Instructions:

```
addi $29, $29, 4
slti   $8, $18, 10
andi $29, $29, 6
ori   $29, $29, 4
```

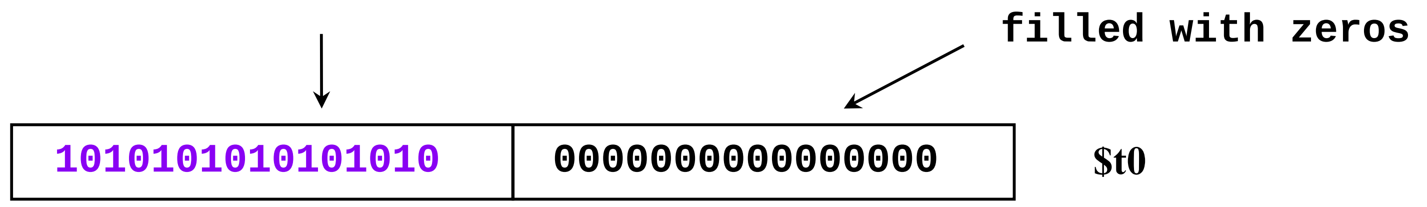
- These are I-type instructions:

op(6)	rs(5)	rt(5)	Immediatae(16)

How about larger constants?

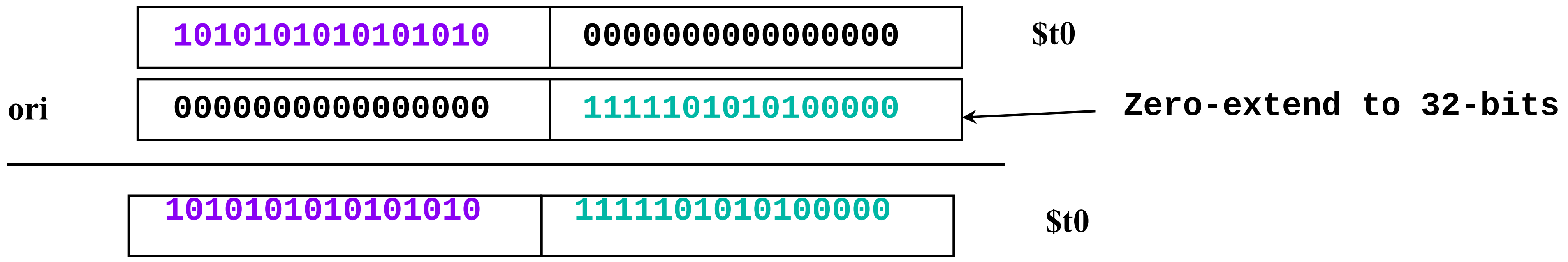
- We load a 32 bit constant into a register first, then operate on the register
- Must use two instructions, new "load upper immediate" instruction

lui $t0, 1010101010101010



- Then must get the lower order bits right, i.e.,

ori $t0, $t0, 1111101010100000



Load Upper Immediate (lui)

- "lui" is an I-type Instruction
- lui $t0, 1010101010101010
  - interpreted as lui $t0, $zero, 1010101010101010
  - similar to addi $t0, $zero, 1010101010101010

    except the immediate operand is considered as the upper 16 bits of a constant

Addresses in Branches and Jumps

- Instructions:
  - bne \$t4,\$t5,Label Next instruction is at Label if \$t4 ≠ \$t5
  - beq \$t4,\$t5,Label Next instruction is at Label if \$t4 = \$t5
  - j Label Next instruction is at Label

- Formats:

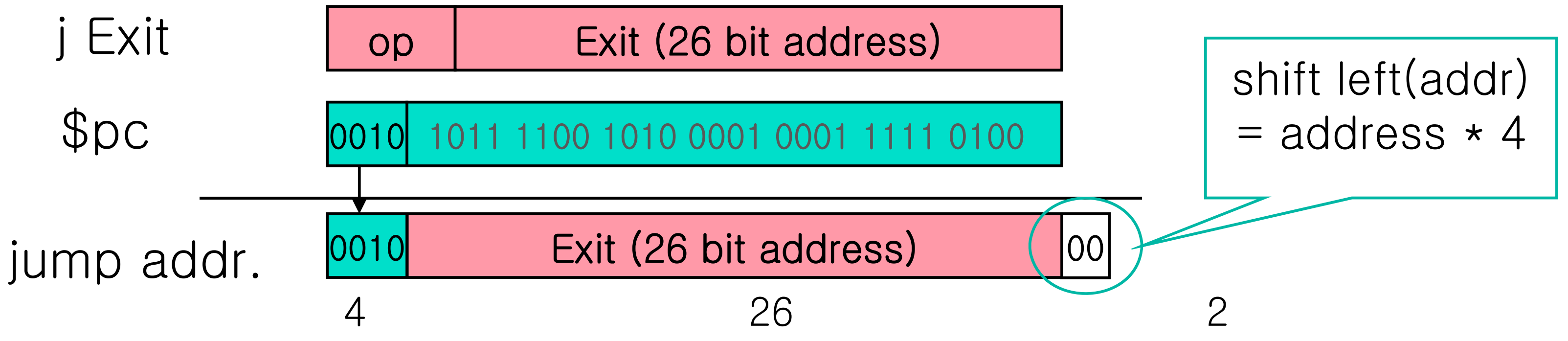
I	op(6)	rs(5)	rt(5)	16 bit address
J	op(6)	26 bit address		

- Addresses are not 32 bits (either 16 bits or 26 bits)
  — How do we handle this?

Addresses in Branches and Jumps

cf. lw $t1, 16($s0)

- **Branches: PC-relative Addressing**
  - **Could specify a register (as in lw and sw) and add it to address**
  - **Use program counter($pc) as a base address**
  - **e.g. beq $s1, $s2, 25 # if ($s1 == $s2) go to ($pc+4)+(4*25)**
  - **# else go to ($pc+4) (next instruction)**
- **Jumps: Pseudodirect Addressing**
  - **Jump instructions just use high order bits of $pc**
  - **the jump address is the 26 bits of the instruction concatenated with the upper 4 bits of the $pc**

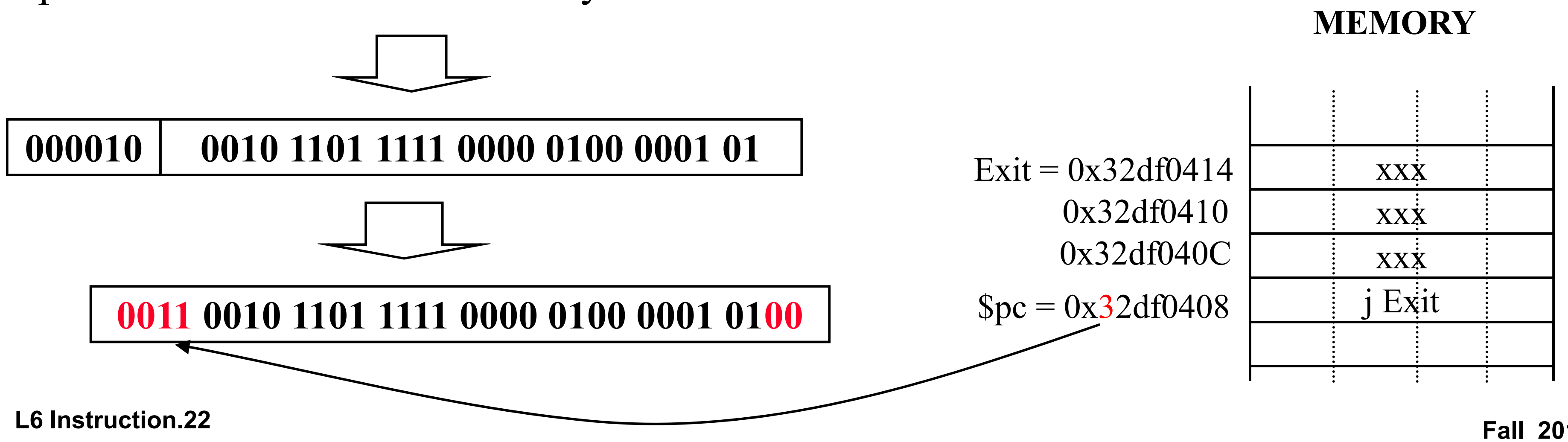


More about Jump

- Current address($pc) = 0x32df0408
  (0011 0010 1101 1111 0000 0100 0000 1000)
- Destination address(Exit) = 0x32df0414
  (0011 0010 1101 1111 0000 0100 0001 0100)
- Instruction ‘j Exit’

000010	0011 0010 1101 1111 0000 0100 0001 0100

6 bits + 32 bits = 38 bits => Too many!
• Last 2 bits are always zero because each instruction consists of 4 bytes.
• First 4 bits remains the same with those in $pc as long as the program is placed within 256MB boundary

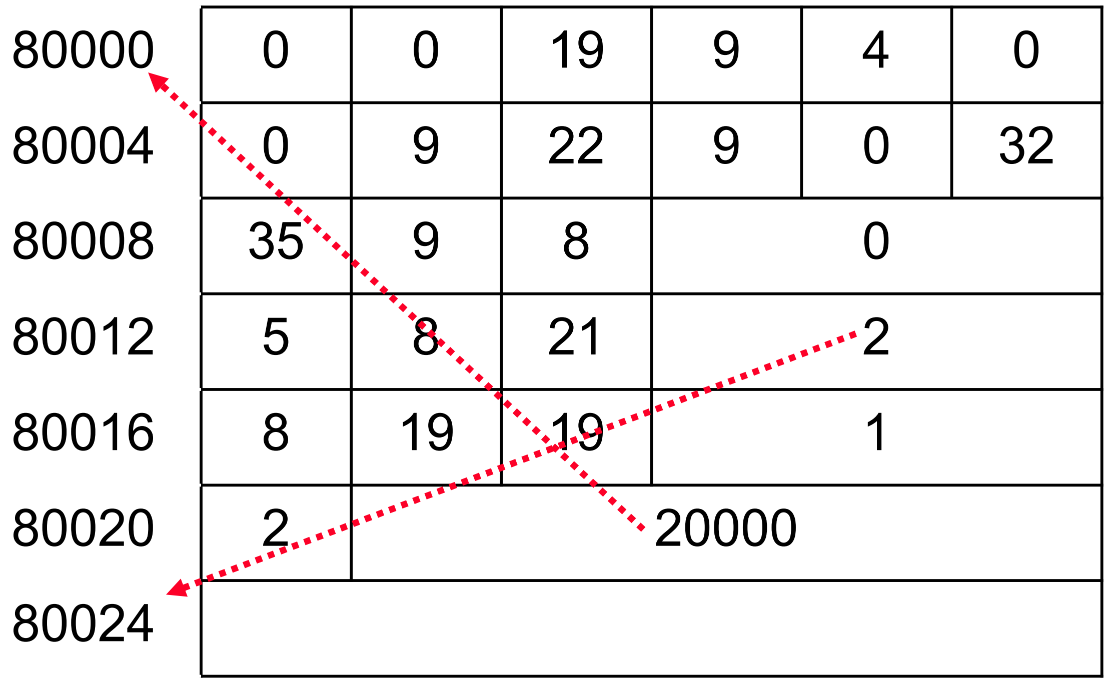


Target Addressing Example

° **Loop code from earlier example**

- **Assume Loop at location 80000**

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```



80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

MIPS Addressing Mode

1. Register addressing
   - operand is a register
   - e.g. add $s0, $s1, $s2 # $s0 = $s1 + $s2
2. Base or Displacement addressing
   - operand = Mem[register value + constant]
   - e.g. lw $s1, 100($s2) # $s1 = Mem[$s2 + 100]
3. Immediate addressing
   - operand is a constant within the instruction
   - e.g. addi $sp, $sp, 12 # $sp = $sp + 12
   - slti $s1, $s2, 100 # $s1 = 1 if ($s2 < 100)
   - # else $s1 = 0
4. PC-relative addressing
   - address is the sum of PC and a constant in the instruction
   - e.g. beq $s1, $s2, 25 # if ($s1 == $s2) go to PC+4+(4*25)
5. Pseudodirect addressing
   - the jump address is the 26 bits of the instruction concatenated with the upper bits of the $pc

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler's imagination

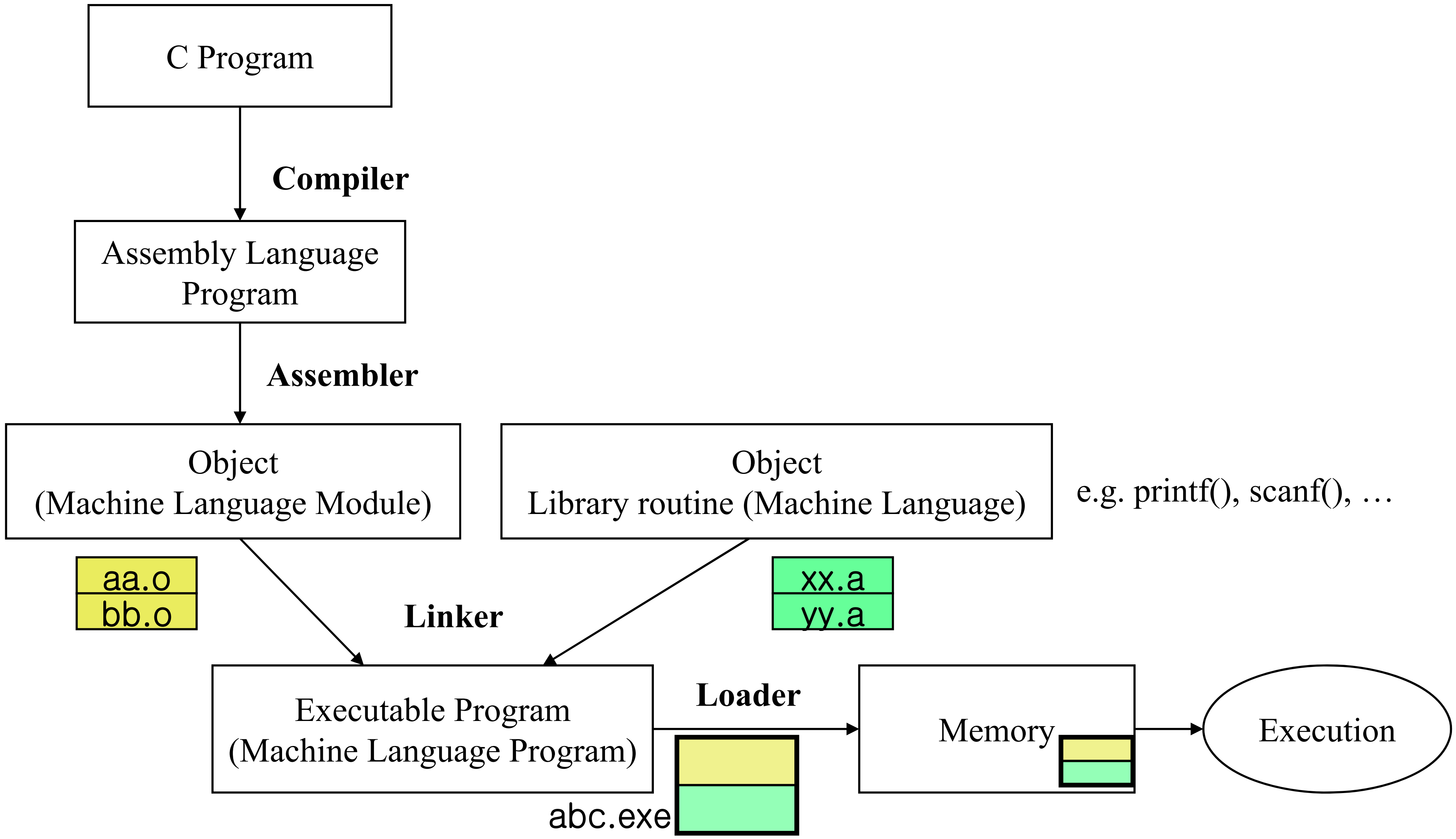
```
move $t0, $t1       →  add $t0, $zero, $t1

blt $t0, $t1, L     →  slt $at, $t0, $t1
                       bne $at, $zero, L
```

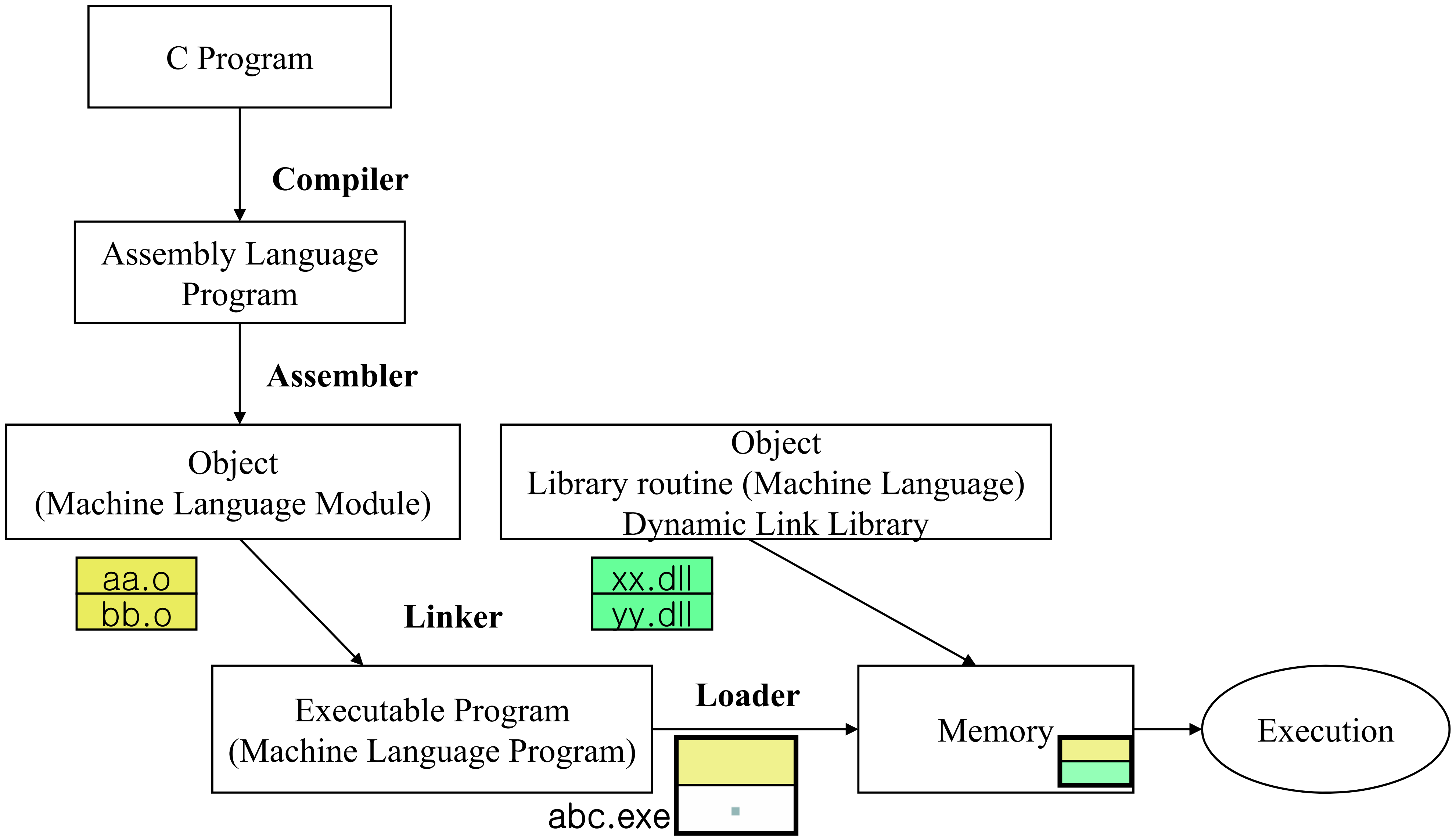
  - $at (register 1): assembler temporary

***figment of imagination: 상상의 산물**

Starting a Program


C Program
Compiler
Assembly Language Program
Assembler
Object (Machine Language Module)
aa.o
bb.o
Object Library routine (Machine Language)
e.g. printf(), scanf(), …
xx.a
yy.a
Linker
Executable Program (Machine Language Program)
abc.exe
Loader
Memory
Execution

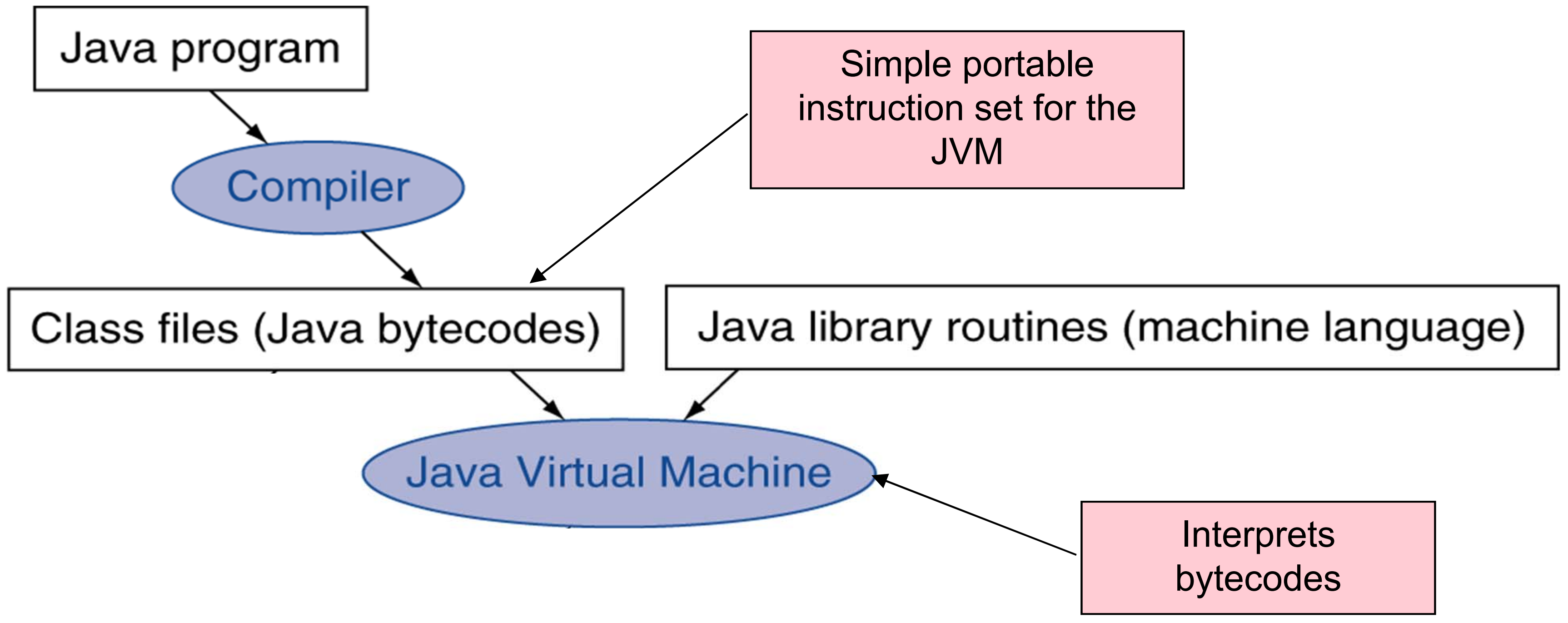
Starting a Program with DLL


C Program
Compiler
Assembly Language Program
Assembler
Object (Machine Language Module)
aa.o
bb.o
Object Library routine (Machine Language) Dynamic Link Library
xx.dll
yy.dll
Linker
Executable Program (Machine Language Program)
abc.exe
Loader
Memory
Execution

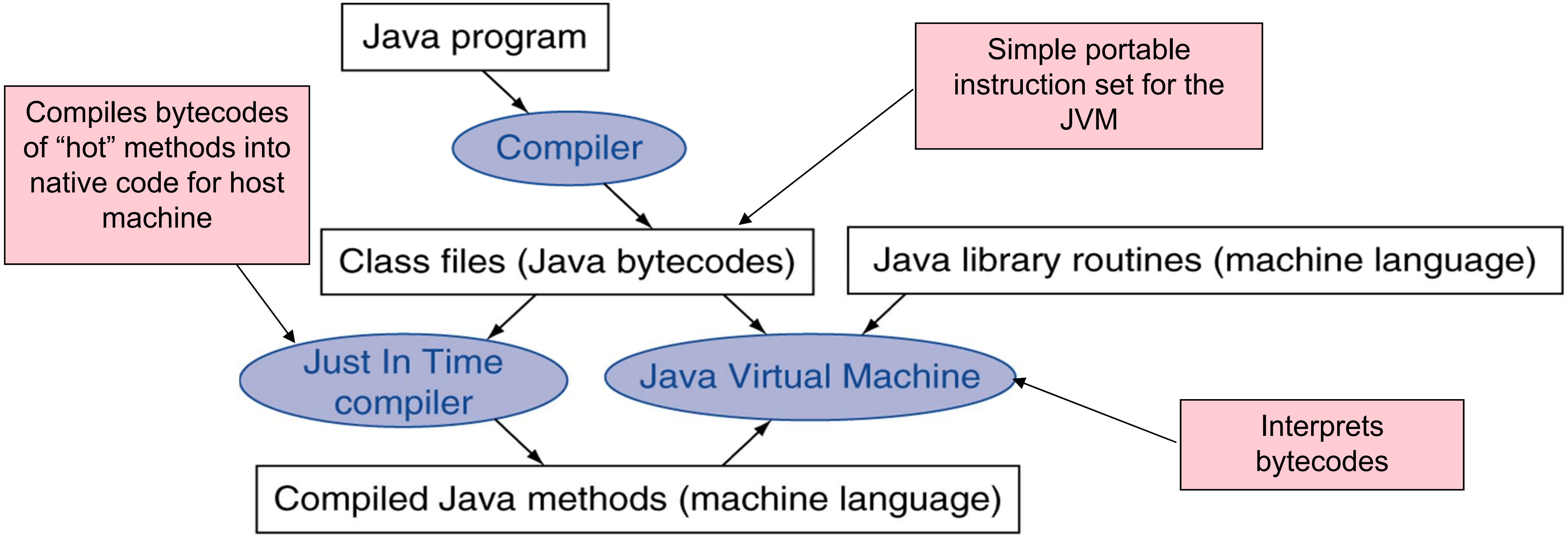
Dynamically Linked Libraries (DLL)

- library routines are not linked into the executable program
- they are linked and loaded when the program is run
- library routines can be bug-fixed and updated independently from application programs
- can save memory while program is running
  - More than one program can share the same dll’s
- MS Windows makes extensive use of DLLs

Starting Java Applications

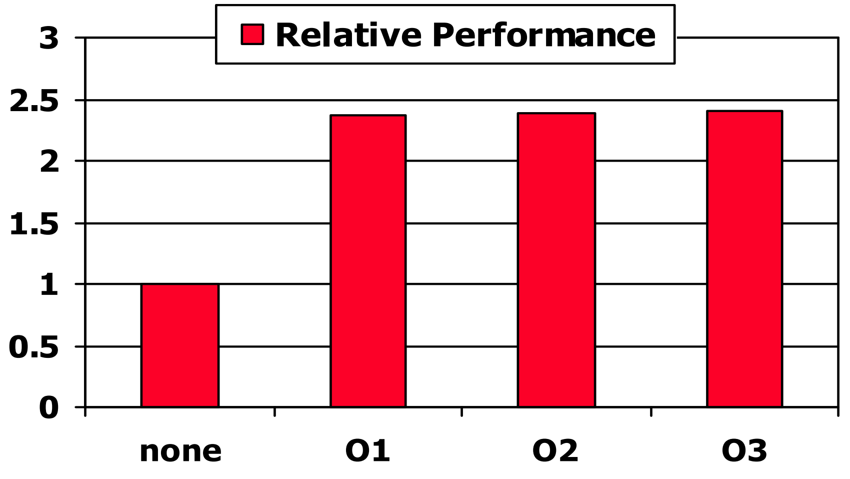

Java program
Compiler
Class files (Java bytecodes)
Java library routines (machine language)
Java Virtual Machine
Simple portable instruction set for the JVM
Interprets bytecodes

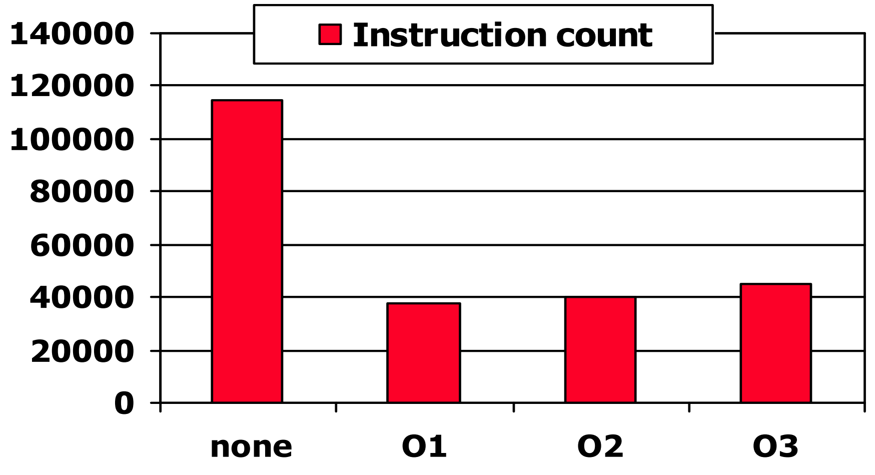
Starting Java Applications

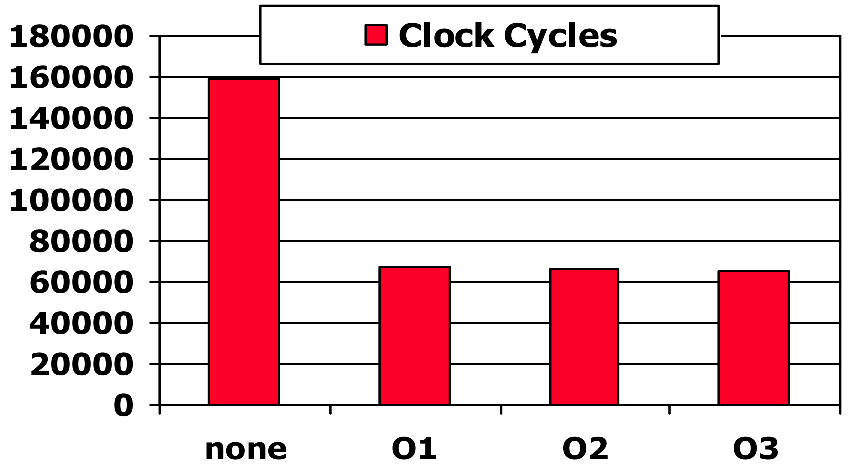

Java program
Compiler
Class files (Java bytecodes)
Java library routines (machine language)
Just In Time compiler
Java Virtual Machine
Compiled Java methods (machine language)
Compiles bytecodes of “hot” methods into native code for host machine
Simple portable instruction set for the JVM
Interprets bytecodes

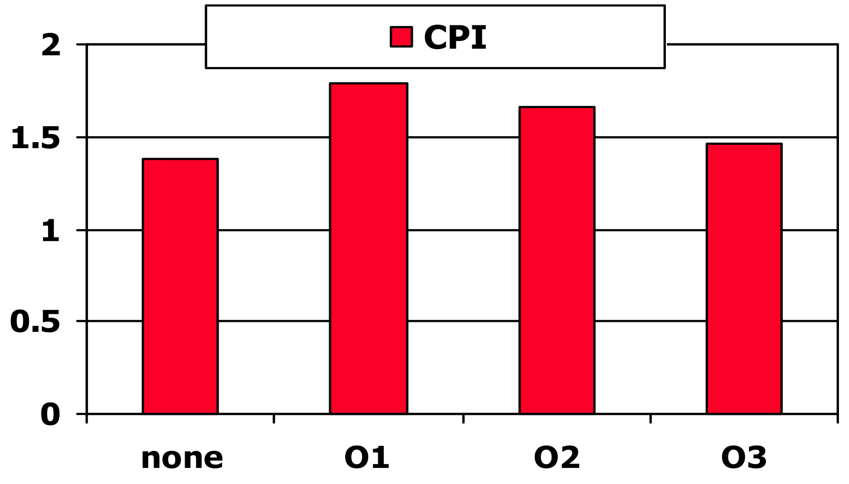
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

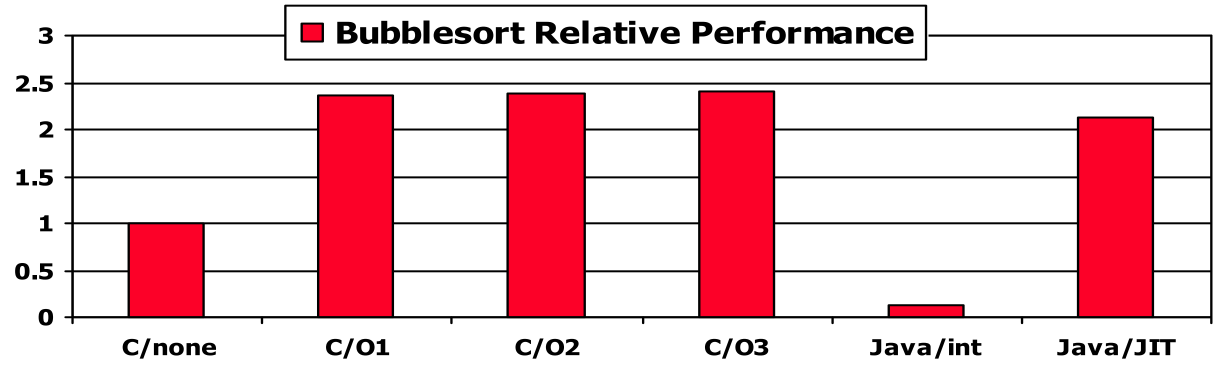


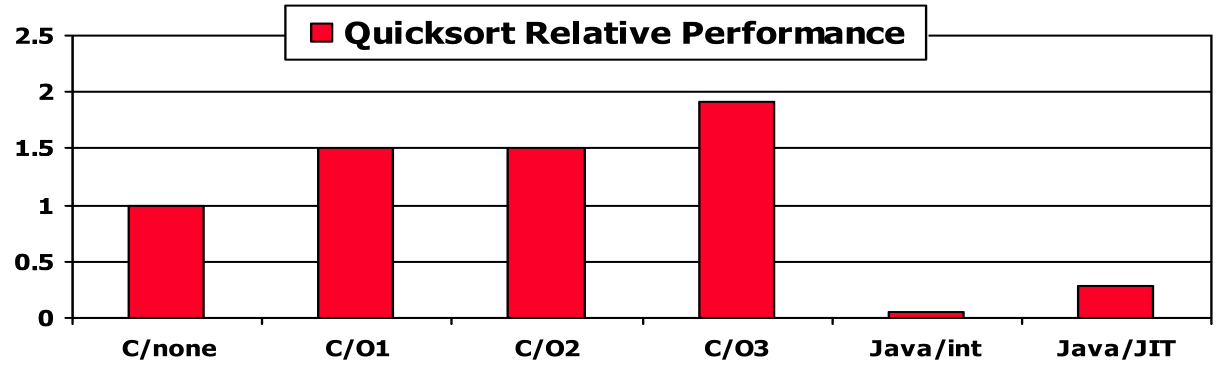


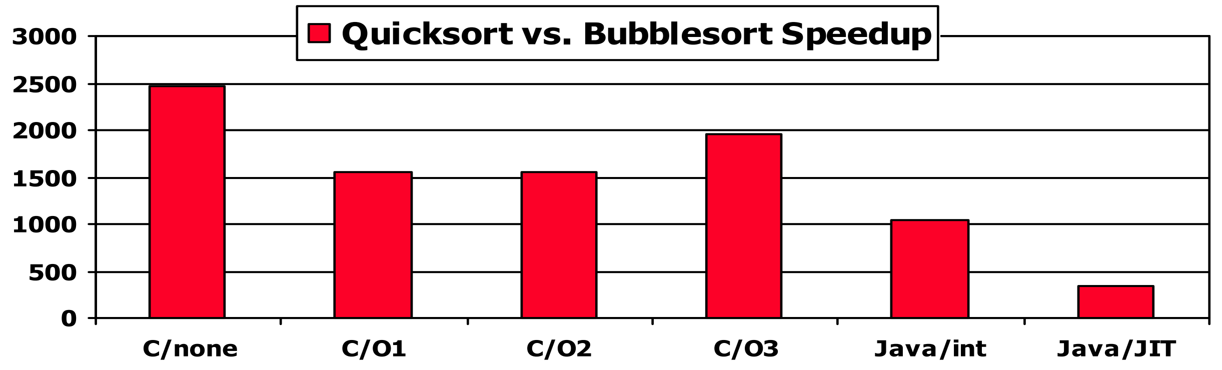




Effect of Language and Algorithm







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!