

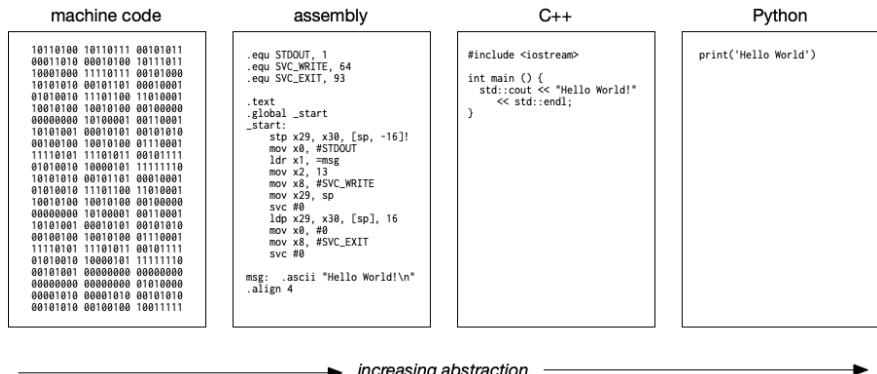
CSC 411

Computer Organization (Spring 2024)

Lecture 16: Compiling, interpreting, and running programs

Prof. Marco Alvarez, University of Rhode Island

Context

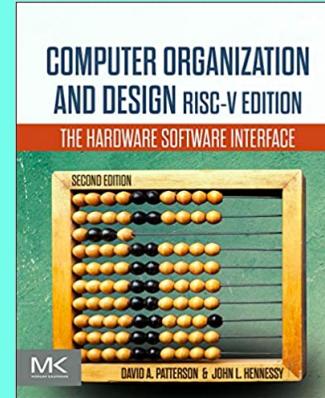


https://www.uvm.edu/~cbcfaier/cs1210/book/02_programming_and_the_python_shell/programming.html

Disclaimer

Some figures and slides are adapted from:

Computer Organization and Design (Patterson and Hennessy)
The Hardware/Software Interface



Compiling vs Interpreting

Compiler

- a program that acts as a **translator**, converting a program from a high-level language into another language
 - most of the time into a low-level language
- because the entire program is translated at once, compilers can perform **optimizations** to make the code more efficient, resulting in faster execution (higher performance)
- e.g. C/C++ compilers

Interpreter

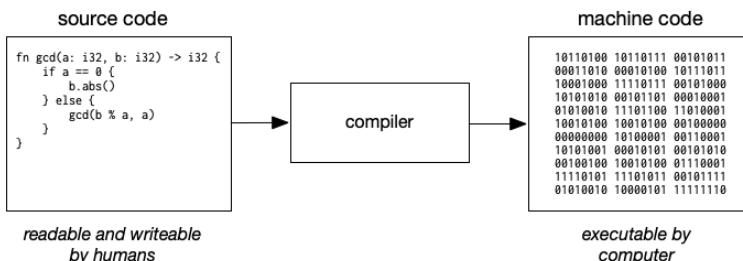
- a program that reads and **translates code on the fly**
 - reads code line by line, translates it into machine code, and executes it
 - any language can be interpreted (including C or assembly)
- preferred when performance is not critical
- e.g. Javascript

Feature	Compiling	Interpreting
Translation Time	Entire program translated before execution	Code translated and executed line by line
Output	Machine code (executable file)	No direct output; relies on interpreter for execution
Execution Speed	Generally faster due to pre-optimization	Generally slower due to on-the-fly translation
Development Speed	Slower due to separate compilation step	Faster; code can be run and tested immediately
Portability	Limited portability (requires recompilation for different architectures)	Higher portability; interpreted code can often run on different systems
Optimizations	Extensive optimizations possible during compilation	Limited optimizations during interpretation
Error Checking	Can perform more comprehensive static analysis	Limited error checking during interpretation
Code Distribution	Compiled machine code (binary files)	Source code in original language

Compiling and linking

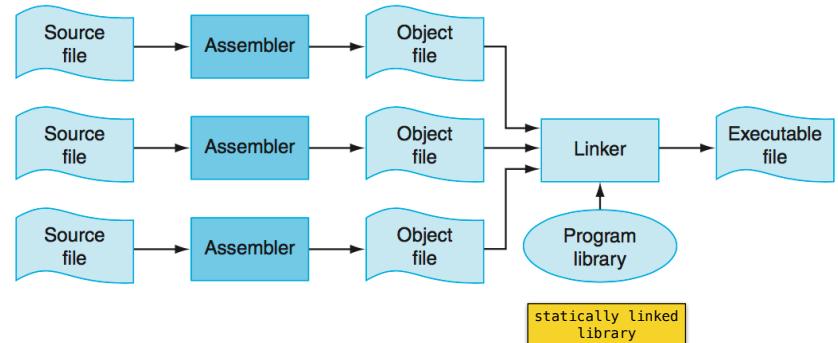
Compiling programs (simplified)

- Typically, “compiling” a program refers to the process of generating machine code from source code
 - the process takes several steps: **compile, assemble, link**



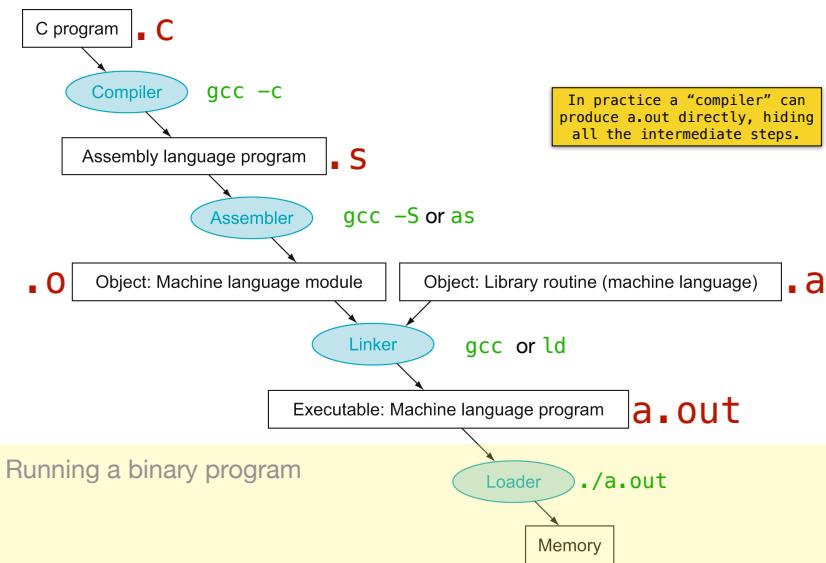
https://www.uvm.edu/~cbcfaier/cs1210/book/02_programming_and_the_python_shell/programming.html

Compiling/linking/running C programs



From Computer Organization and Computer Design: The Hardware/Software Interface

Compiling/linking/running C programs



C Compiler (gcc)

- Translates C language into assembly
 - symbolic form of machine language
 - in the 70s many operating systems were written in assembly language
- Optimizes generated code
 - with the gcc compiler, optimization levels are specified using flags like `-O1`, `-O2`, and `-O3`, which progressively increase the level of optimization performed by the compiler
 - other optimization flags include `-O0`, `-Os`, `-Ofast`, `-Og`, `-Oz`

<https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>

GCC optimizations

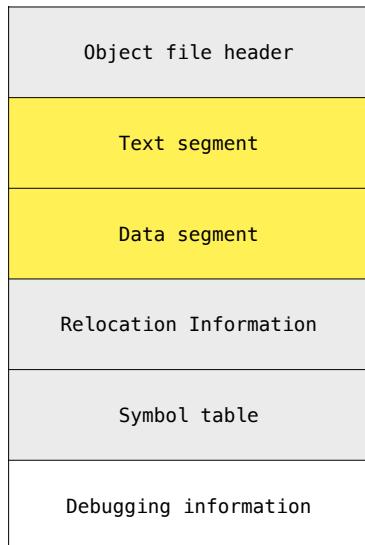
Optimization name	Explanation	gcc level
<i>High level</i>	<i>At or near the source level: processor independent</i>	
Procedure integration	Replace procedure call by procedure body	03
<i>Local</i>	<i>Within straight-line code</i>	
Common subexpression elimination	Replace two instances of the same computation 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
<i>Global</i>	<i>Across a branch</i>	
Global common subexpression elimination	Same as local, but this version crosses branches	02
Copy propagation	Replace all instances of a variable <i>A</i> that has been assigned <i>X</i> (i.e., <i>A = X</i>) with <i>X</i>	02
Code motion	Remove code from a loop that computes the same value each iteration of the loop	02
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	02
<i>Processor dependent</i>	<i>Depends on processor knowledge</i>	
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

Assembler

- Converts pseudo-instructions into actual instructions
- Converts actual assembly instructions into **machine code**
 - must determine memory addresses and offsets corresponding to all labels (requires two passes over program)
- Generates an **object file**
 - combination of machine instructions, data, and information needed to place instructions properly in memory

An object file

- Intermediate file produced by a compiler, typically consisting of the following:
 - object file header:** describes contents (size/position) of other pieces of the object
 - text segment:** translated instructions (machine code)
 - static data segment:** static data in the program
 - relocation information:** for instructions/ data that depend on absolute addresses (updated by the linker later)
 - symbol table:** list of symbols (function names, labels, global variables, etc.) along with their respective addresses that can be used by other objects
 - debugging information:** metadata for associating machine code with original source code



ELF files

Executable and Linkable Format

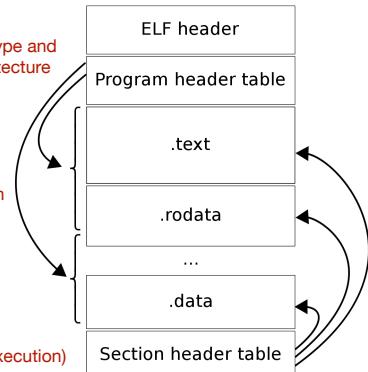
Useful commands:

- readelf**
- objdump**
- nm**

Identify the ELF type and specify the architecture

Execution information

Technical details for linking (ignored for execution)

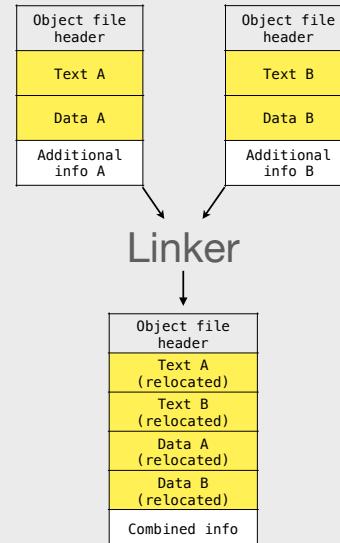


https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

Linker

- Merges all **object files**, producing an **executable**
 - merge code and data segments from all objects
 - resolve all undefined labels (replace them with final addresses)
 - patch location-dependent and external references
- Could leave location dependencies to be fixed by a **relocating loader**
 - with virtual memory, no need to do this
 - program can be loaded into absolute location in virtual memory space
- Allows individual files to be compiled independently
 - changes made to a source file do not require recompiling all others

Linker



Linker can resolve all the TODO items from the independent object modules before generating a final executable

Example

sum_int.h

```
#ifndef __SUM__
#define __SUM__

int sum(int a, int b);
#endif
```

sum_int.c

```
extern int baseline;

int sum(int a, int b) {
    if (a < b)
        return (a + b) - baseline;
    else
        return a + b;
}
```

prog.c

```
#include "sum_int.h"

int baseline = 10;

int main() {
    int a = 10;
    int b = 20;

    int c = sum(a, b);

    return 0;
}
```

Editing instructions

```
7f45c46 01010100 00000000 00000000 0100f300 01000000 00000000 00000000
44020000 00000000 34000000 00025000 0a000000 130101fe 232e8100 13040102
2326a4fe 2324b4fe 0327c4fe 0327d4fe 0327e4fe 0327f4fe 3307f700
b707f700 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0324c101 13010102 67000000 00474343 3a202857 63203031 64286463 32385259
2031332e 322a3000 41160000 00726973 63769001 11000000 04100572 76333269
32763100 00000000 00000000 00000000 01000000 00000000 00000000 00000000
0400f1ff 00000000 00000000 00000000 03000100 00000000 00000000 00000000
03000300 00000000 00000000 00000000 03000400 0b000000 00000000 00000000
00000100 16000000 3c000000 00000000 00000100 1a000000 48000000 00000000
00000100 00000000 00000000 00000000 03000500 00000000 00000000 00000000
00000100 00000000 00000000 00000000 03000500 00000000 00000000 00000000
10000000 0073756d 5f606c74 2c630024 78727632 32693270 31002e4c 32092e4c
33007375 6d006261 73656c69 6e650000 1c000000 10000000 00000000 2c000000
1a000000 00000000 2c000000 33000000 00000000 30000000 1bb00000 00000000
30000000 33000000 00000000 38000000 11070000 00000000 002e7379 6d746162
002e7374 72746162 002e7368 73747274 61620022 72656c61 2e746578 74002664
61746100 2e627373 02e636f7 6d6d655e 74002e72 69736376 2e617474 72696275
74657373 6d6d655e 02e636f7 6d6d655e 74002e72 69736376 2e617474 72696275
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
58000000 00000000 00000000 04000000 04000000 00000000 1b000000 04000000 40000000
00000000 00000000 04000000 04000000 07000000 01000000 04000000 0c000000 26000000
01000000 03000000 00000000 8c000000 00000000 00000000 01000000 00000000 00000000
00000000 00000000 00000000 03000000 00000000 00000000 00000000 00000000 00000000
00000000 01000000 00000000 31000000 01000000 30000000 00000000 0c000000
1c000000 00000000 00000000 01000000 3a000000 03000070 00000000 00000000
00000000 05000000 1c000000 00000000 00000000 01000000 00000000 01000000
00000000 00000000 00000000 00000000 00000000 00000000 0a000000 04000000
10000000 03000000 03000000 00000000 00000000 04000000 00000000 00000000
00000000 01000000 00000000 11000000 03000000 00000000 00000000 75010000
4:000000 00000000 00000000 01000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

xxd -g4 -c 32 sum_int.o

objdump -d sum_int.o

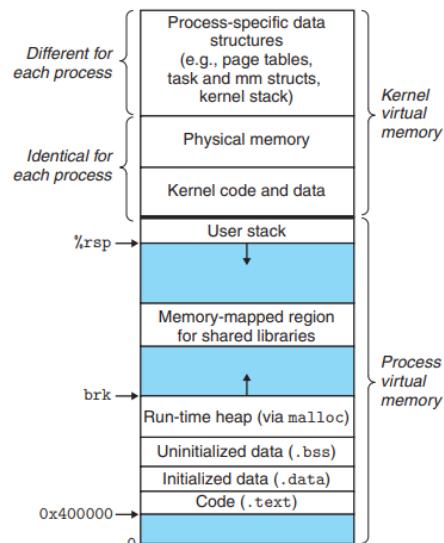
```
00000000 <sum>:
0: fe010113 addi sp,sp,-32
4: 00812e23 sw $0,28(sp)
8: 02010113 addi s0,sp,32
c: fea42623 sw a0,-20($0)
10: feb24223 lw a1,-24($0)
14: fec42703 lw a4,-20($0)
18: fe842783 lw a5,-24($0)
1c: 0275063 bge a4,a5,3c <.L2>
20: fec42703 lw a4,-20($0)
24: feb24223 lw a5,-24($0)
28: fea42623 addi a4,a5
2c: 00007057 lui a5,0x0
30: 0007a783 lw a5,0(a5) # 0 <sum>
34: 49f707b3 sub a5,a4,a5
38: 0100006f j 48 <.L3>
0000003c <.L2>:
3c: fec42703 lw a4,-20($0)
40: fe842783 lw a5,-24($0)
44: 00f707b3 add a5,a4,a5
00000048 <.L3>:
48: 00078513 mv a0,a5
4c: 01c12403 lw s0,28(sp)
50: 02010113 addi sp,sp,32
54: 00008067 ret
000101d0 <sum>:
101d0: fe010113 addi sp,sp,-32
101d4: 00812e23 sw $0,28(sp)
101d8: 02010413 addi s0,sp,32
101dc: fea42623 sw a0,-20($0)
101e0: feb24223 lw a1,-24($0)
101e4: fea42623 sw a2,-20($0)
101e8: feb24223 lw a3,-24($0)
101ec: 00f75e63 bge a4,a5,10208 <sum+0x38>
101f0: fec42703 lw a4,-20($0)
101f4: feb24223 lw a5,-24($0)
101f8: 00f70733 add a4,a4,a5
101fc: d3c1a783 lw a5,-708(gp) # 13564 <baseline>
10200: fea42623 sw a5,a4,a5
10204: 01c12403 lw s0,28(sp)
10208: feb24223 lw a4,-20($0)
1020c: fea42783 lw a5,-24($0)
10210: 00f707b3 add a5,a4,a5
10214: 00078513 mv a0,a5
10218: 01c12403 lw s0,28(sp)
1021c: 02010113 addi sp,sp,32
10220: 00008067 ret
```

Loading

Loader (OS task)

- (1) reads header to determine text/data segment sizes
- (2) creates address space large enough for text/data
- (3) copies text/data from executable into memory
- (4) copies the command line arguments (if any) onto the stack
- (5) initializes registers (including sp, fp, gp)
- (6) jump to startup routine
 - copies arguments to x10 and x11 and calls main
 - when main returns, terminate program with an exit system call

Structure of a Linux process



Dynamically linked libraries

• Statically linked libraries

- library procedures become part of the executable code
 - if new versions are released code must be recompiled otherwise executable keeps using the old versions
 - loads procedures in the library even if those calls are not executed

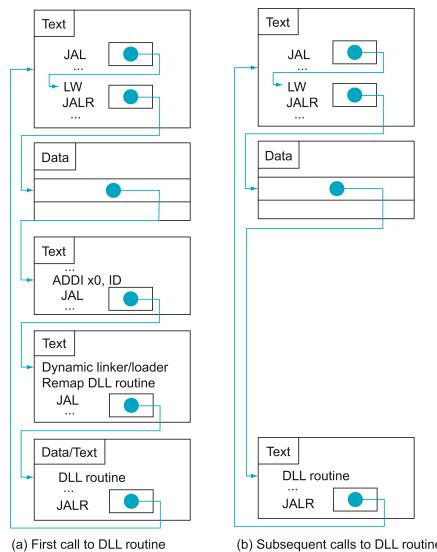
• Dynamically linked libraries (DLLs)

- only link/load library procedure when it is called
- automatically picks up new library versions
- requires procedure code to be relocatable

Lazy linkage

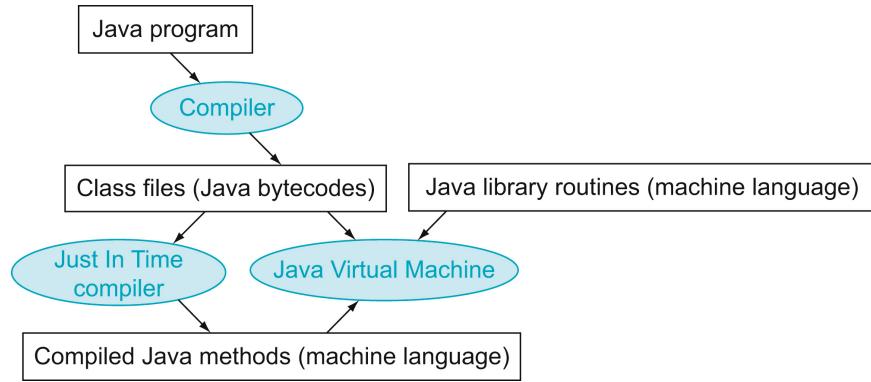
Dynamically linked library via lazy procedure linkage:

- Steps for the first time a call is made to the DLL routine.
- The steps to find the routine, remap it, and link it are skipped on subsequent calls.

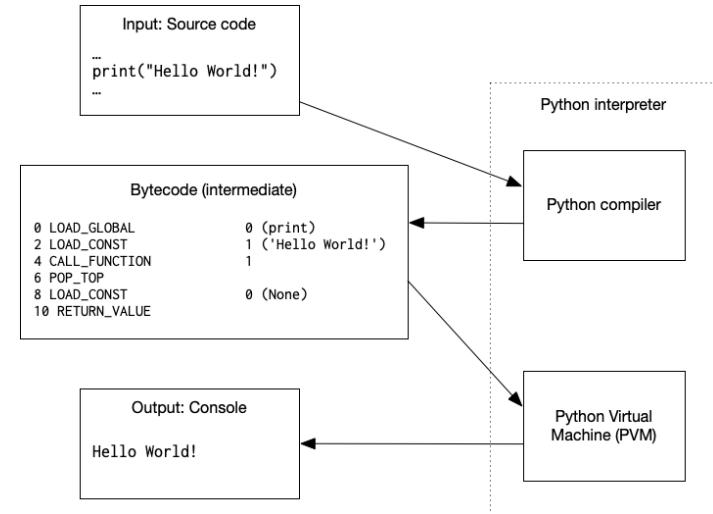


Modern interpreters

Java applications



Python interpreter



Python interpreter

```
python -m dis <file>

mylist = [1, 2, 3, 'hello', 'world']

myslice = mylist[3:]

print(myslice)
```

Annotations from the original image:

- Annotations point to the variable assignment `mylist` and the slice assignment `myslice` in the source code.
- Annotations point to the corresponding bytecode instructions: 1 BUILD_LIST, 2 LOAD_CONST, 4 LIST_EXTEND, 6 STORE_NAME for the first assignment; 8 LOAD_NAME, 10 LOAD_CONST, 12 LOAD_CONST, 14 BUILD_SLICE, 16 BINARY_SUBSCR, 18 STORE_NAME for the slice assignment; and 20 LOAD_NAME, 22 LOAD_NAME, 24 CALL_FUNCTION, 26 POP_TOP, 28 LOAD_CONST, 30 RETURN_VALUE for the print statement.

Impacts on performance

Sort example in C

```

void swap(int v[], size_t k) {
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}

void sort (int v[], size_t n) {
    size_t i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1] ; j -= 1) {
            swap(v, j);
        }
    }
}

```

```

swap:
    slli x6, x11, 2      # reg x6 = k * 4
    add x6, x10, x6      # reg x6 = v + (k * 4)
    lw x5, 0(x6)          # reg x5 (temp) = v[k]
    lw x7, 4(x6)          # reg x7 = v[k + 1]
    sw x7, 0(x6)          # v[k] = reg x7
    sw x5, 4(x6)          # v[k+1] = reg x5 (temp)
    jalr x0, 0(x1)         # return to caller

```

Saving registers	
sort:	addi sp, sp, -20 # make room on stack for 5 registers sw x1, 16(sp) # save return address on stack sw x22, 12(sp) # save x22 on stack sw x21, 8(sp) # save x21 on stack sw x20, 4(sp) # save x20 on stack sw x19, 0(sp) # save x19 on stack
Procedure body	
Move parameters	addi x21, x10, 0 # copy parameter x10 into x21 addi x22, x11, 0 # copy parameter x11 into x22
Outer loop	addi x19,x0, 0 # i = 0 for1st:bge x19, x22, exit1 # go to exit1 if i >= n
Inner loop	addi x20, x19, -1 # j = i - 1 for2st:blt x20, x0, exit2 # go to exit2 if j < 0 slli x5, x20, 2 # x5 = j * 4 add x5, x21, x5 # x5 = v + (j * 4) lw x6, 0(x5) # x6 = v[j] lw x7, 4(x5) # x7 = v[j + 1] ble x6, x7, exit2 # go to exit2 if x6 < x7
Pass parameters and call	addi x10, x21, 0 # first swap parameter is v addi x11, x20, 0 # second swap parameter is j jal x1, swap # call swap
Inner loop	addi x20, x20, -1 # j for2st jal x0, for2st # go to for2st
Outer loop	exit2: addi x19, x19, 1 # i += 1 jal x0, for1st # go to for1st
Restoring registers	
exit1:	lw x19, 0(sp) # restore x19 from stack lw x20, 4(sp) # restore x20 from stack lw x21, 8(sp) # restore x21 from stack lw x22, 12(sp) # restore x22 from stack lw x1, 16(sp) # restore return address from stack addi sp, sp, 20 # restore stack pointer
Procedure return	
	jalr x0, 0(x1) # return to calling routine

Applying compiler optimizations

gcc optimization	Relative performance	Clock cycles (millions)	Instruction count (millions)	CPI
None	1.00	158,615	114,938	1.38
O1 (medium)	2.37	66,990	37,470	1.79
O2 (full)	2.38	66,521	39,993	1.66
O3 (procedure integration)	2.41	65,747	44,993	1.46

The programs sorted 100,000 32-bit words with the array initialized to random values. These programs were run on a Pentium 4 with a clock rate of 3.06 GHz and a 533 MHz system bus with 2 GB of PC2100 DDR SDRAM. It used Linux version 2.4.20.

Comparing sorting algorithms

- Compiler optimizations are sensitive to the algorithm
- Java/JIT significantly faster than JVM interpreted

Language	Execution method	Optimization	Bubble Sort relative performance	Quicksort relative performance	Speedup Quicksort vs. Bubble Sort
C	Compiler	None	1.00	1.00	2468
	Compiler	O1	2.37	1.50	1562
	Compiler	O2	2.38	1.50	1555
	Compiler	O3	2.41	1.91	1955
Java	Interpreter	-	0.12	0.05	1050
	JIT compiler	-	2.13	0.29	338

Nothing can fix slow algorithms

Optimization example

Example: clearing an array

```
void clear1(int array[], int size) {    void clear2(int *array, int size) {  
    int i;        int *p = array;  
    for (i = 0 ; i < size ; i += 1)    int *end = p + size;  
        array[i] = 0;            for ( ; p < end ; p++)  
    }                            *p = 0;  
  
    li x5, 0      # i = 0  
l1: slli x6, x5, 2 # x6 = i * 4  
    add x7, x10, x6 # x7 = &array[i]  
    sw x0, 0(x7) # array[i] = 0  
    addi x5, x5, 1 # i = i + 1  
    blt x5, x11, l1 # if (i<size) go to l1  
  
    add x5, x0, x10 # p = addr of array[0]  
    slli x6, x11, 2 # x6 = size * 4  
    add x7, x10, x6 # x7 = addr of array[size]  
l2: sw x0, 0(x5) # *p = 0  
    addi x5, x5, 4 # p = p + 4  
    bltu x5, x7, l2 # if (p<end) go to l2
```

(*) assembly code above does not consider the case of size = 0

compilers can do this by applying optimizations: **strength reduction** (shift instead of multiply) and **induction variable elimination** (eliminate address calculation inside the loop)

Arrays vs pointers

- Array indexing typically involves
 - multiplying index by element size
 - adding to array base address
- Pointers correspond directly to memory addresses
 - can avoid indexing complexity

Fallacies

- Powerful instructions (x86) ⇒ higher performance
 - complex instructions are hard to implement
 - may slow down all instructions, including simple ones
 - compilers are good at making fast code from simple instructions
- Write assembly code for higher performance
 - modern compilers are better (than most humans) at dealing with modern processors
 - more lines of code ⇒ more errors and less productivity

Fallacies

- Backward compatibility ⇒ instruction set “doesn’t change”
 - x86 keeps adding more instructions (difficult to build compatible processors)

Growth of the x86 instruction set over time

