# CSC 411

**Computer Organization (Fall 2024)**
**Lecture 15: Compiling, interpreting, and running programs**

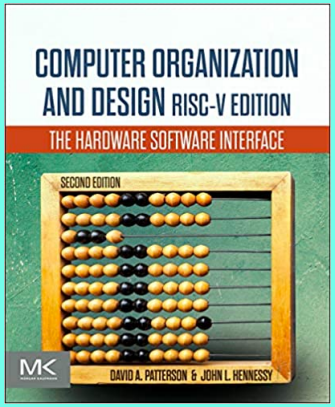
**Prof. Marco Alvarez, University of Rhode Island**

# Disclaimer

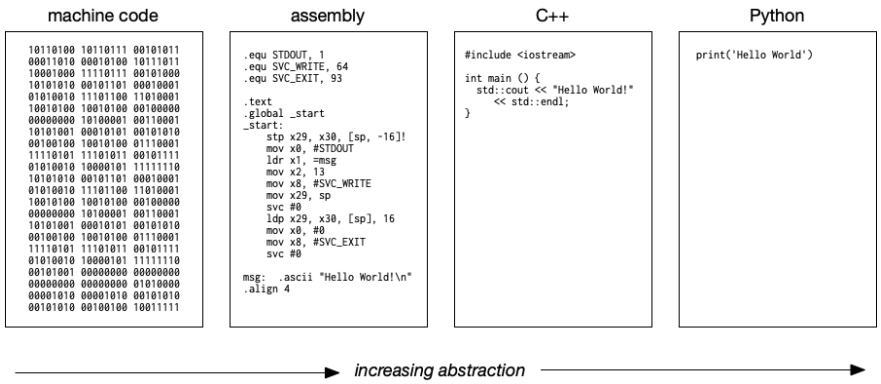
Some figures and slides are adapted from:

Computer Organization and Design (Patterson and Hennessy)

The Hardware/Software Interface




# Context




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

# Program execution approaches

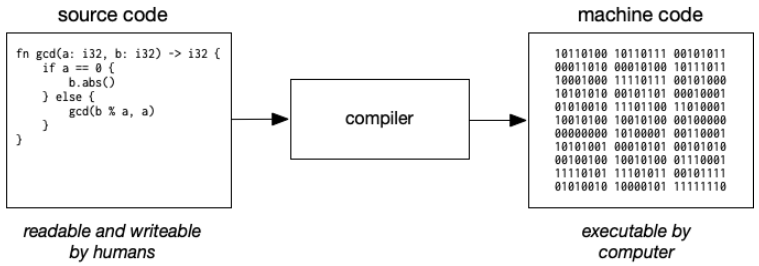
- Compilation
  - high level source **translated** into another language
    - most of the time into a low-level language
  - as code is translated at once, compilers can perform **optimizations** to make the code more efficient, resulting in faster execution (higher performance)
  - e.g. C/C++ compilers
- Interpretation
  - 'executing' a program directly from source
    - read code line by line, translate it into machine code, and execute
    - any language can be interpreted
  - 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)

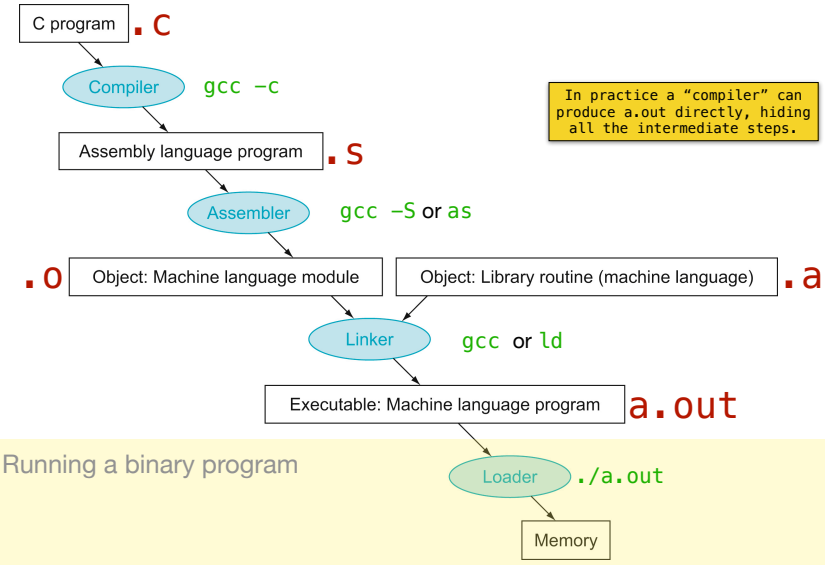
- Typically, "compiling" a program refers to the process of generating machine code from source code
  - the process takes several steps: compile, assemble, link



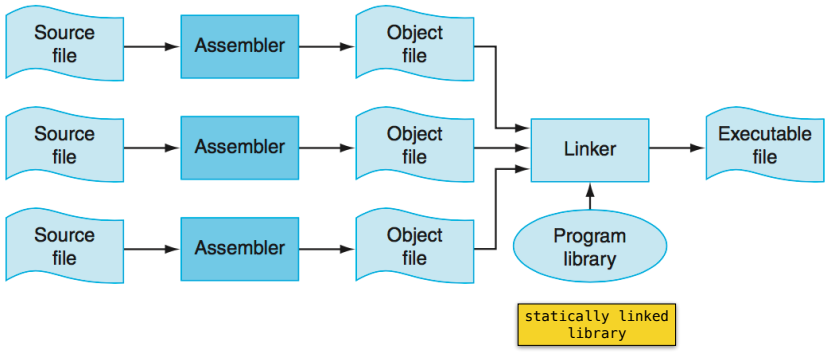

`Source Code → [Preprocessor] → [Frontend] → [Optimizer] → [Backend] → Object Code`

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

## Compiling/linking/running C programs



# Compiling/linking/running C programs




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

# 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 –01, –02, and –03, which progressively increase the level of optimization performed by the compiler
    - other optimization flags include `–00`, `–0s`, `–0fast`, `–0g`, `–0z`

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

# GCC optimizations

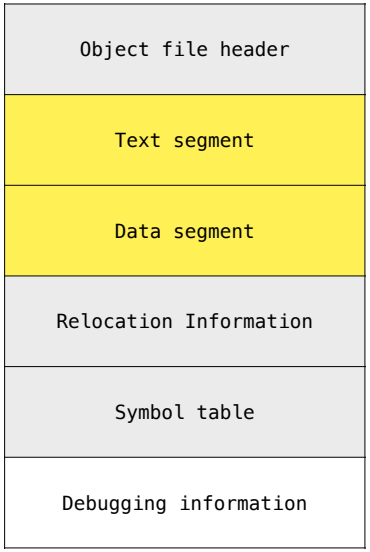
| Optimization name | Explanation | gcc level |
|---|---|---|
| ***High level*** | ***At or near the source level; processor independent*** | |
| Procedure integration | Replace procedure call by procedure body | 03 |
| ***Local*** | ***Within straight-line code*** | |
| Common subexpression elimination | 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 the 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 |

# 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

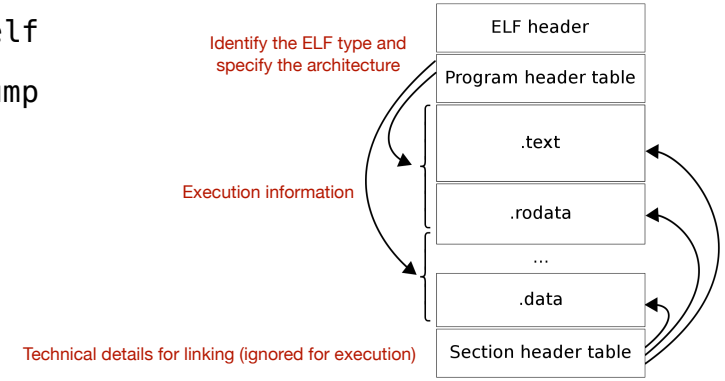



# Linker

- 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 mande to a source file do not require recompiling all others

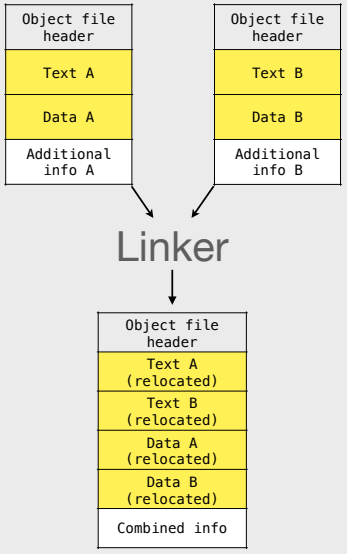
# ELF files

- Executable and Linkable Format
- Useful commands:
  - `readelf`
  - `objdump`
  - `nm`




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

# 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

```
xxd -g4 -c 32 sum_int.o
```

```
7f454c46 01010100 00000000 00000000 0100f300 01000000 00000000 00000000
44020000 00000000 34000000 00002800 0a000900 130101fe 232e8100 13040102
2326a4fe 2324b4fe 0327c4fe 832784fe 6350f702 0327c4fe 832784fe 3307f700
b7070000 83a70700 b307f740 6f000001 0327c4fe 832784fe b307f700 13850700
0324c101 13010102 67800000 00474343 3a202867 63383931 64386463 32336529
2031332e 322e3000 411b0000 00726973 63760001 11000000 04100572 76333269
32703100 00000000 00000000 00000000 00000000 01000000 00000000 00000000
0400f1ff 00000000 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
03000600 1e000000 00000000 58000000 12000100 22000000 00000000 00000000
10000000 0073756d 5f696e74 2e630024 78727633 32693270 31002e4c 32002e4c
33007375 6d006261 73656c69 6e650000 1c000000 10060000 00000000 2c000000
1a0b0000 00000000 2c000000 33000000 00000000 30000000 1b0b0000 00000000
30000000 33000000 00000000 38000000 11070000 00000000 002e7379 6d746162
002e7374 72746162 002e7368 73747274 6162002e 72656c61 2e746578 74002e64
61746100 2e627373 002e636f 6d6d656e 74002e72 69736376 2e617474 72696275
74657300 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 20000000 01000000 06000000 00000000 34000000
58000000 00000000 00000000 04000000 00000000 1b000000 04000000 40000000
00000000 b0010000 48000000 07000000 01000000 04000000 0c000000 26000000
01000000 03000000 00000000 8c000000 00000000 00000000 00000000 01000000
00000000 2c000000 05000000 03000000 00000000 8c000000 00000000 00000000
00000000 01000000 00000000 31000000 01000000 30000000 00000000 8c000000
1c000000 00000000 00000000 01000000 01000000 3a000000 03000070 00000000
00000000 a8000000 1c000000 00000000 00000000 01000000 00000000 01000000
02000000 00000000 00000000 c4000000 c0000000 08000000 0a000000 04000000
10000000 09000000 03000000 00000000 00000000 84010000 2b000000 00000000
00000000 01000000 00000000 11000000 03000000 00000000 00000000 f8010000
4c000000 00000000 00000000 01000000 00000000
```

```
objdump -d a.out
```

```
000101d0 <sum>:
   101d0:   fe010113    addi    sp,sp,-32
   101d4:   00812e23    sw  s0,28(sp)
   101d8:   02010413    addi    s0,sp,32
   101dc:   fea42623    sw  a0,-20(s0)
   101e0:   feb42423    sw  a1,-24(s0)
   101e4:   fec42703    lw  a4,-20(s0)
   101e8:   fe842783    lw  a5,-24(s0)
   101ec:   00f75e63    bge a4,a5,10208 <sum+0x38>
   101f0:   fec42703    lw  a4,-20(s0)
   101f4:   fe842783    lw  a5,-24(s0)
   101f8:   00f70733    add a4,a4,a5
   101fc:   d3c1a783    lw  a5,-708(gp) # 13564 <baseline>
   10200:   40f707b3    sub a5,a4,a5
   10204:   0100006f    j   10214 <sum+0x44>
   10208:   fec42703    lw  a4,-20(s0)
   1020c:   fe842783    lw  a5,-24(s0)
   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
```

```
objdump -d sum_int.o
```

```
00000000 <sum>:
   0:   fe010113    addi    sp,sp,-32
   4:   00812e23    sw  s0,28(sp)
   8:   02010413    addi    s0,sp,32
   c:   fea42623    sw  a0,-20(s0)
  10:   feb42423    sw  a1,-24(s0)
  14:   fec42703    lw  a4,-20(s0)
  18:   fe842783    lw  a5,-24(s0)
  1c:   02f75063    bge a4,a5,3c <.L2>
  20:   fec42703    lw  a4,-20(s0)
  24:   fe842783    lw  a5,-24(s0)
  28:   00f70733    add a4,a4,a5
  2c:   000007b7    lui a5,0x0
  30:   0007a783    lw  a5,0(a5) # 0 <sum>
  34:   40f707b3    sub a5,a4,a5
  38:   0100006f    j   48 <.L3>

0000003c <.L2>:
  3c:   fec42703    lw  a4,-20(s0)
  40:   fe842783    lw  a5,-24(s0)
  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
```

# 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

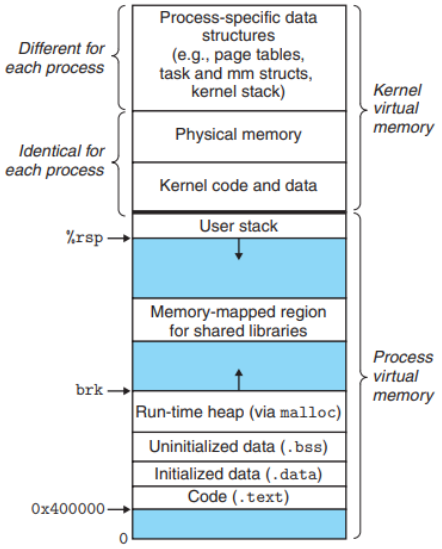
(4) copies the command line arguments (if any) onto the stack

(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

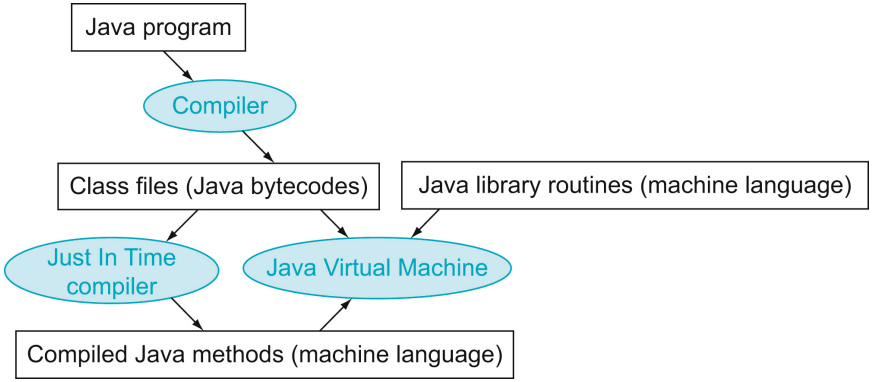



# Dynamically linked libraries

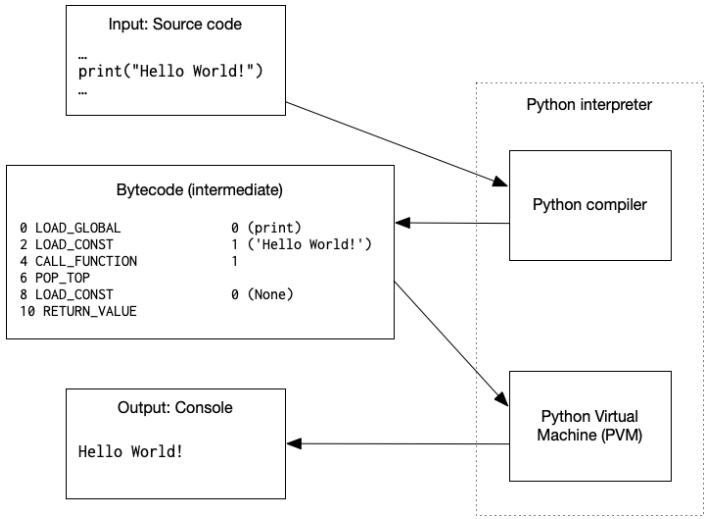
- Statically linked libraries
  - library procedures become part of the executable code
    - if new versions are released code must be recompiled otherwise executable keeps using the old versions
    - loads procedures in the library even if those calls are not executed
- Dynamically linked libraries (DLLs)
  - only link/load library procedure when it is called
  - automatically picks up new library versions
  - requires procedure code to be relocatable

# Interpreters

# Java applications



# Python interpreter




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

# Python interpreter

`python -m dis <file>`

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

myslice = mylist[3:]

print(myslice)
```

```
1   0 BUILD_LIST      0
    2 LOAD_CONST      0 ((1, 2, 3, 'hello', 'world'))
    4 LIST_EXTEND     1
    6 STORE_NAME      0 (mylist)

2   8 LOAD_NAME       0 (mylist)
   10 LOAD_CONST      1 (3)
   12 LOAD_CONST      2 (None)
   14 BUILD_SLICE     2
   16 BINARY_SUBSCR
   18 STORE_NAME      1 (myslice)

3   20 LOAD_NAME      2 (print)
   22 LOAD_NAME       1 (myslice)
   24 CALL_FUNCTION   1
   26 POP_TOP
   28 LOAD_CONST      2 (None)
   30 RETURN_VALUE
```

# 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;
}
```

```
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
```

```
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);
        }
    }
}
```

| 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 |
| | for1tst:bge x19, x22, exit1 | # go to exit1 if i >= n |
| Inner loop | addi x20, x19, -1 | # j = i - 1 |
| | for2tst: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 for2tst |
| | jal, x0 for2tst | # go to for2tst |
| Outer loop | exit2: addi x19, x19, 1 | # i += 1 |
| | jal, x0 for1tst | # go to for1tst |

| 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 |
| 01 (medium) | 2.37 | 66,990 | 37,470 | 1.79 |
| 02 (full) | 2.38 | 66,521 | 39,993 | 1.66 |
| 03 (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 | 01 | 2.37 | 1.50 | 1562 |
| | Compiler | 02 | 2.38 | 1.50 | 1555 |
| | Compiler | 03 | 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

# 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

# Example: clearing an array

```
void clear1(int array[], int size) {
    int i;
    for (i = 0 ; i < size ; i += 1)
        array[i] = 0;
}
```

```
void clear2(int *array, int size) {
    int *p = array;
    int *end = p + size;
    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)