

# Computer Architecture

## Chapter 2: MIPS – part 3

Dr. Phạm Quốc Cường

Adapted from Computer Organization the Hardware/Software Interface – 5th

# Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, ...
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

# Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case

`lb rt, offset(rs)`      `lh rt, offset(rs)`

- Sign extend to 32 bits in `rt`

`lbu rt, offset(rs)`      `lhu rt, offset(rs)`

- Zero extend to 32 bits in `rt`

`sb rt, offset(rs)`      `sh rt, offset(rs)`

- Store just rightmost byte/halfword

# 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

# 32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant
  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

lhi \$s0, 61

0000 0000 0111 1101	0000 0000 0000 0000
---------------------	---------------------

ori \$s0, \$s0, 2304

0000 0000 0111 1101	0000 1001 0000 0000
---------------------	---------------------

# Branch Addressing

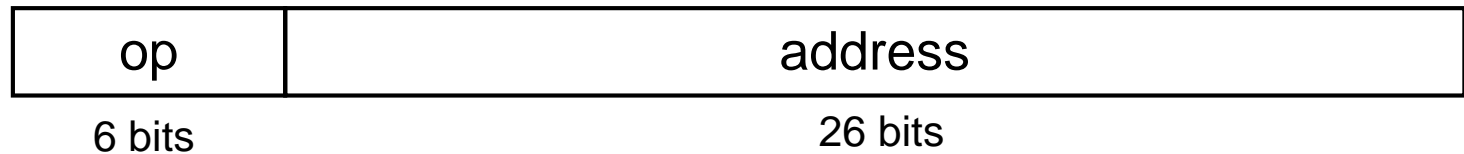
- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward



- PC-relative addressing
  - Target address =  $PC + \text{offset} \times 4$
  - PC already incremented by 4 by this time

# Jump Addressing

- Jump (j and jal) targets could be anywhere in text segment
  - Encode full address in instruction



- (Pseudo)Direct jump addressing
  - Target address =  $PC_{31...28} : (\text{address} \times 4)$

# Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

Loop: sll	\$t1, \$s3, 2	80000	0	0	19	9	4	0
add	\$t1, \$t1, \$s6	80004	0	9	22	9	0	32
lw	\$t0, 0(\$t1)	80008	35	9	8	0		
bne	\$t0, \$s5, Exit	80012	5	8	21	2		
addi	\$s3, \$s3, 1	80016	8	19	19	1		
j	Loop	80020	2	20000				
Exit: ...		80024						



# Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
beq $s0,$s1, L1
```

↓

```
bne $s0,$s1, L2
```

```
j L1
```

```
L2: ...
```

# Addressing Mode Summary

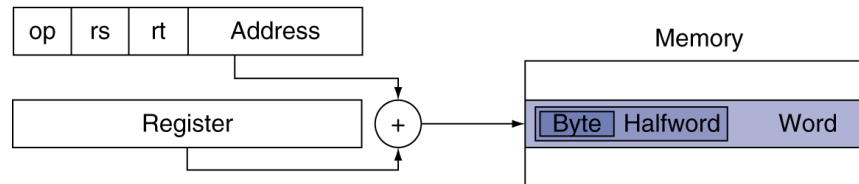
## 1. Immediate addressing



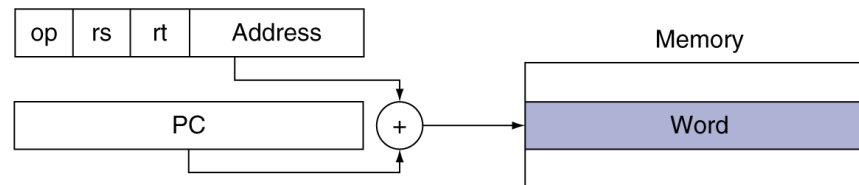
## 2. Register addressing



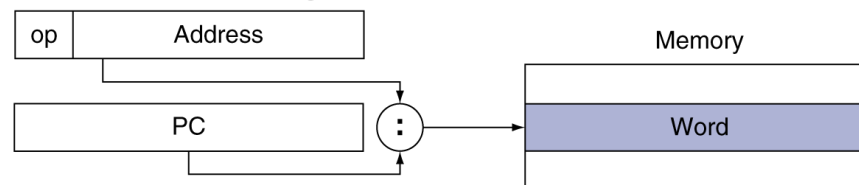
## 3. Base addressing



## 4. PC-relative addressing



## 5. Pseudodirect addressing



# Synchronization

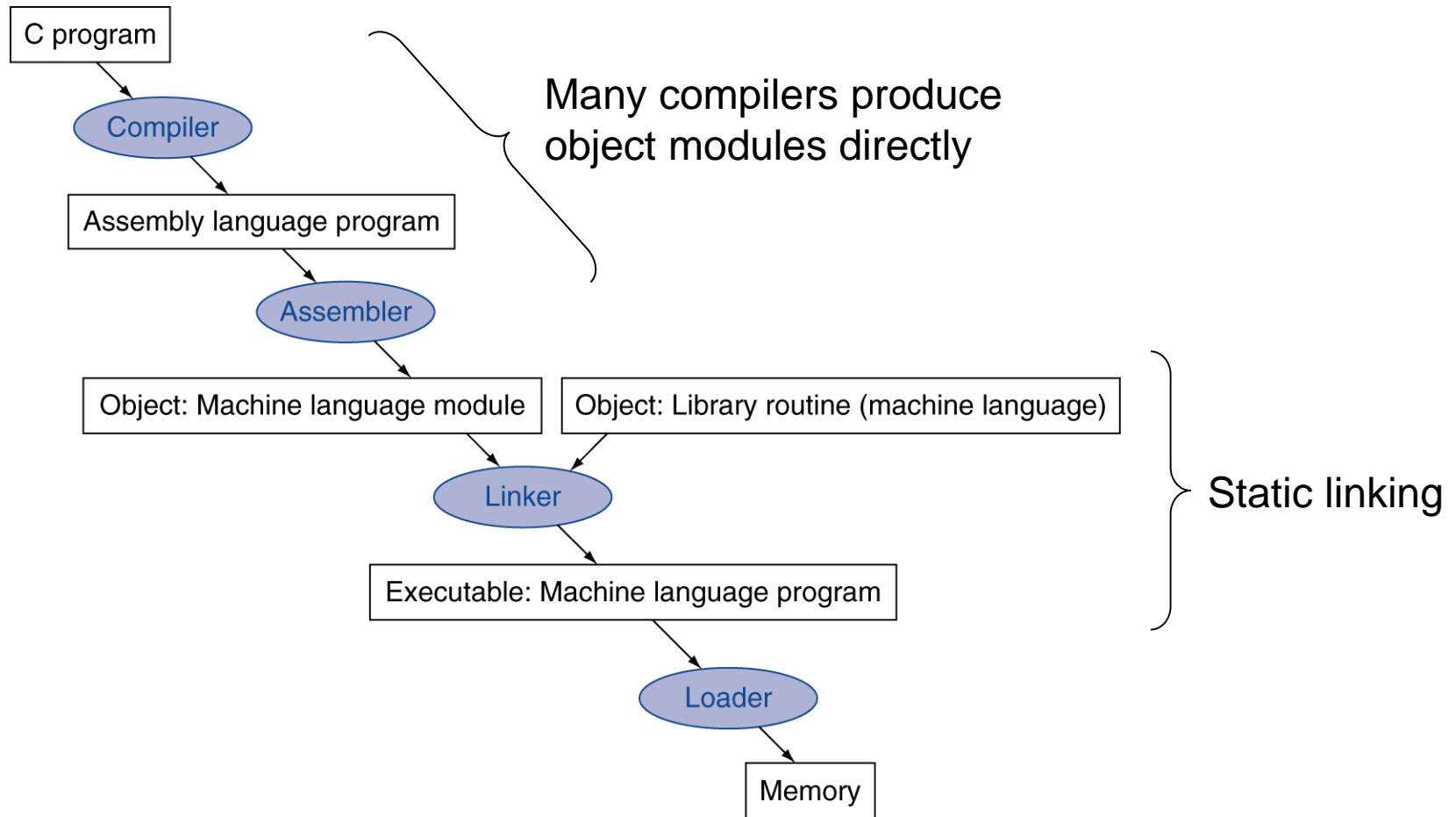
- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register  $\leftrightarrow$  memory
  - Or an atomic pair of instructions

# Synchronization in MIPS

- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in `rt`
  - Fails if location is changed
    - Returns 0 in `rt`
- Example: atomic swap (to test/set lock variable)

```
try: add $t0,$zero,$s4 ;copy exchange value
      ll $t1,0($s1)    ;load linked
      sc $t0,0($s1)    ;store conditional
      beq $t0,$zero,try ;branch store fails
      add $s4,$zero,$t1 ;put load value in $s4
```

# Translation and Startup



# Assembler Pseudoinstructions

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

`move $t0, $t1`       $\rightarrow$    `add $t0, $zero, $t1`

`blt $t0, $t1, L`     $\rightarrow$    `slt $at, $t0, $t1`  
                                 `bne $at, $zero, L`

– `$at` (register 1): assembler temporary

# 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 Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space



# 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

# 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

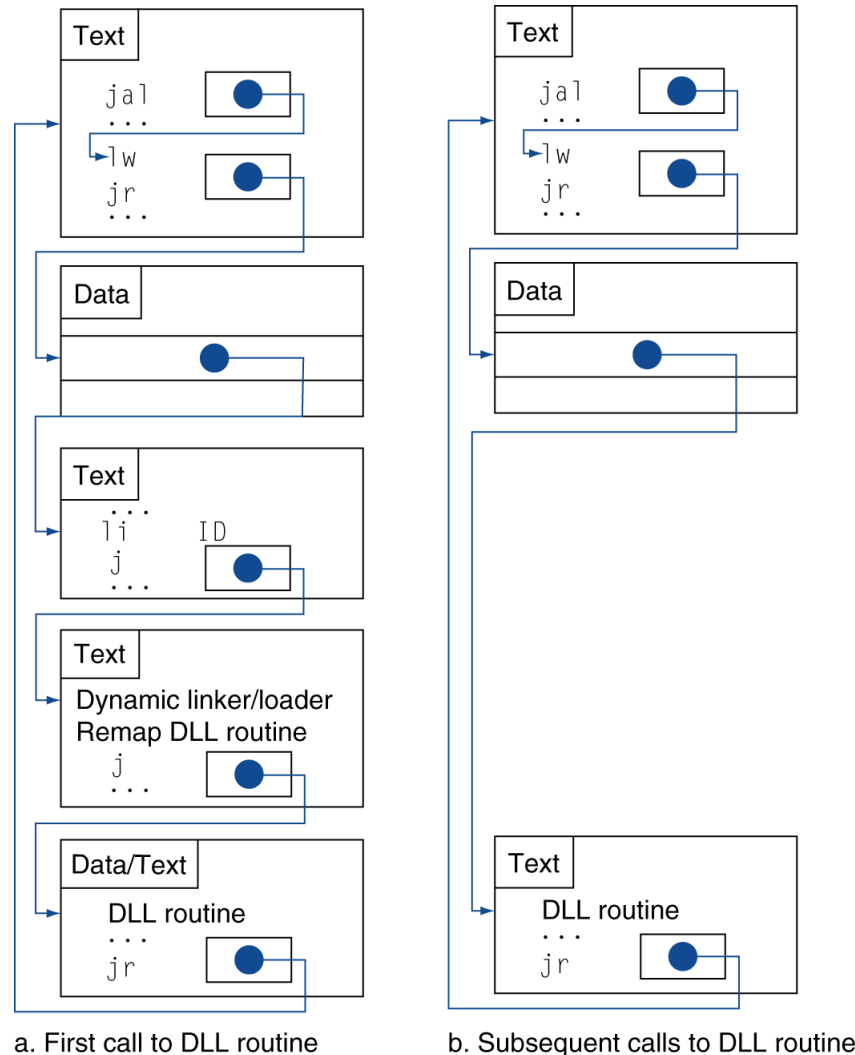
# Lazy Linkage

Indirection table

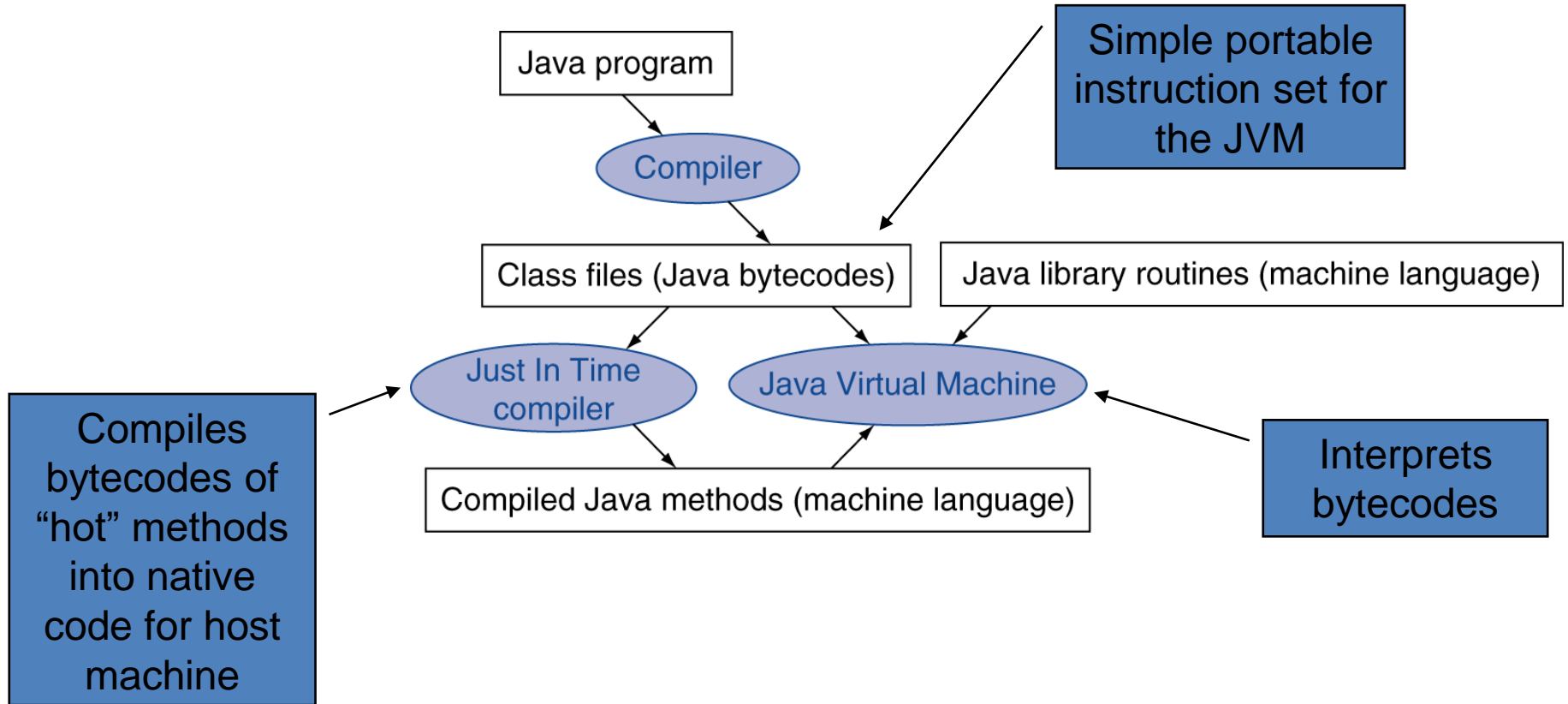
Stub: Loads routine ID,  
Jump to linker/loader

Linker/loader code

Dynamically  
mapped code



# Starting Java Applications



# C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

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

– v in \$a0, k in \$a1, temp in \$t0

# The Procedure Swap

```
swap: sll $t1, $a1, 2    # $t1 = k * 4
      add $t1, $a0, $t1  # $t1 = v+(k*4)
                          # (address of v[k])
      lw $t0, 0($t1)     # $t0 (temp) = v[k]
      lw $t2, 4($t1)     # $t2 = v[k+1]
      sw $t2, 0($t1)     # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)     # v[k+1] = $t0 (temp)
      jr $ra             # return to calling routine
```

# The Sort Procedure in C

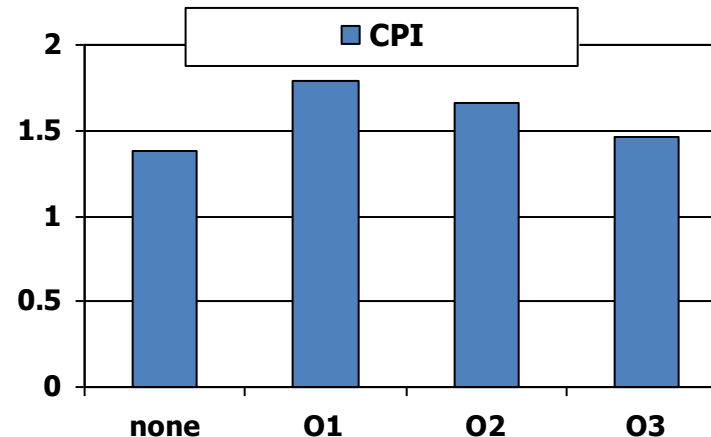
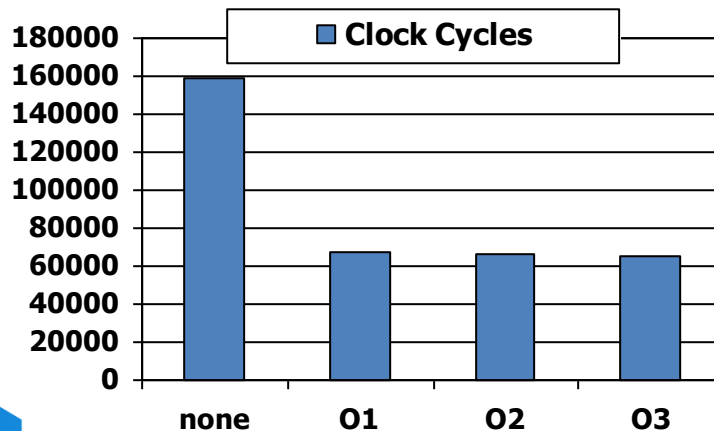
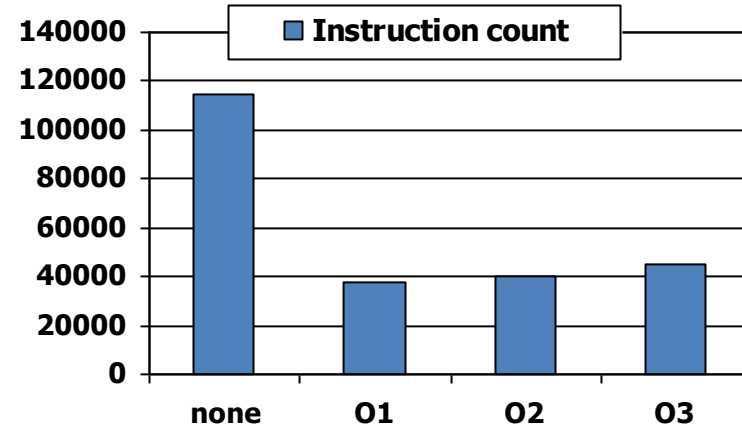
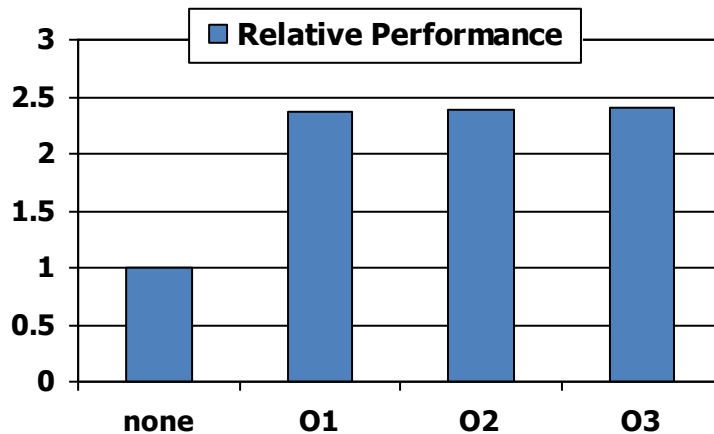
- Non-leaf (calls swap)

```
void sort (int v[], int n)
{
    int 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);
        }
    }
}
```

– v in \$a0, k in \$a1, i in \$s0, j in \$s1

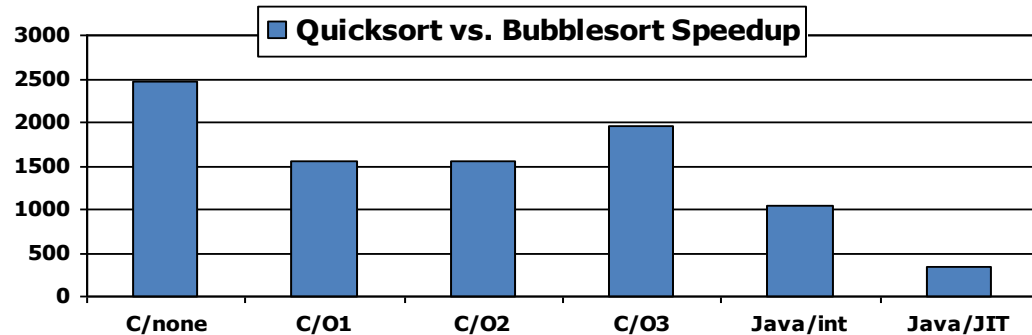
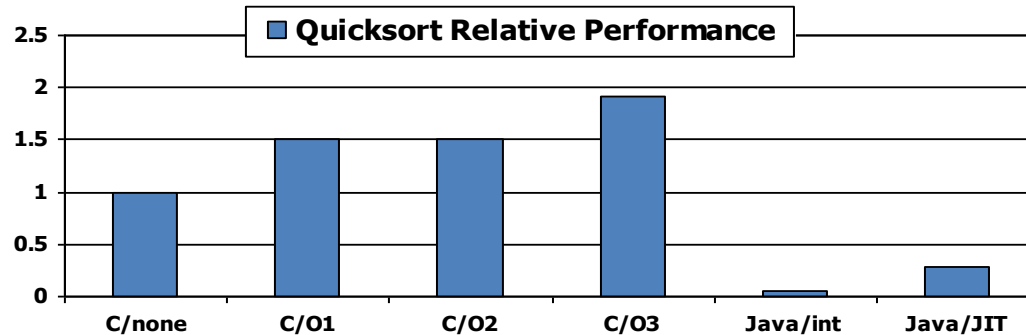
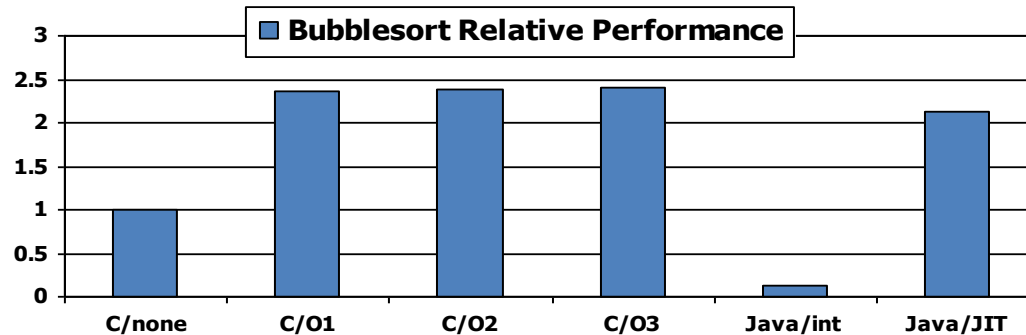
# Effect of Compiler Optimization

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!

# Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

# Example: Clearing and Array

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

```
        move $t0,$zero    # i = 0
loop1: sll $t1,$t0,2      # $t1 = i * 4
        add $t2,$a0,$t1   # $t2 =
                        # &array[i]
        sw $zero, 0($t2)  # array[i] = 0
        addi $t0,$t0,1    # i = i + 1
        slt $t3,$t0,$a1   # $t3 =
                        # (i < size)
        bne $t3,$zero,loop1 # if (...)
                        # goto loop1
```

```
clear2(int *array, int size) {
    int *p;
    for (p = &array[0]; p < &array[size];
        p = p + 1)
        *p = 0;
}
```

```
        move $t0,$a0      # p = & array[0]
        sll $t1,$a1,2      # $t1 = size * 4
        add $t2,$a0,$t1   # $t2 =
                        # &array[size]
loop2: sw $zero,0($t0)    # Memory[p] = 0
        addi $t0,$t0,4     # p = p + 4
        slt $t3,$t0,$t2   # $t3 =
                        # (p < &array[size])
        bne $t3,$zero,loop2 # if (...)
                        # goto loop2
```

# Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented  $i$
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

# ARM & MIPS Similarities

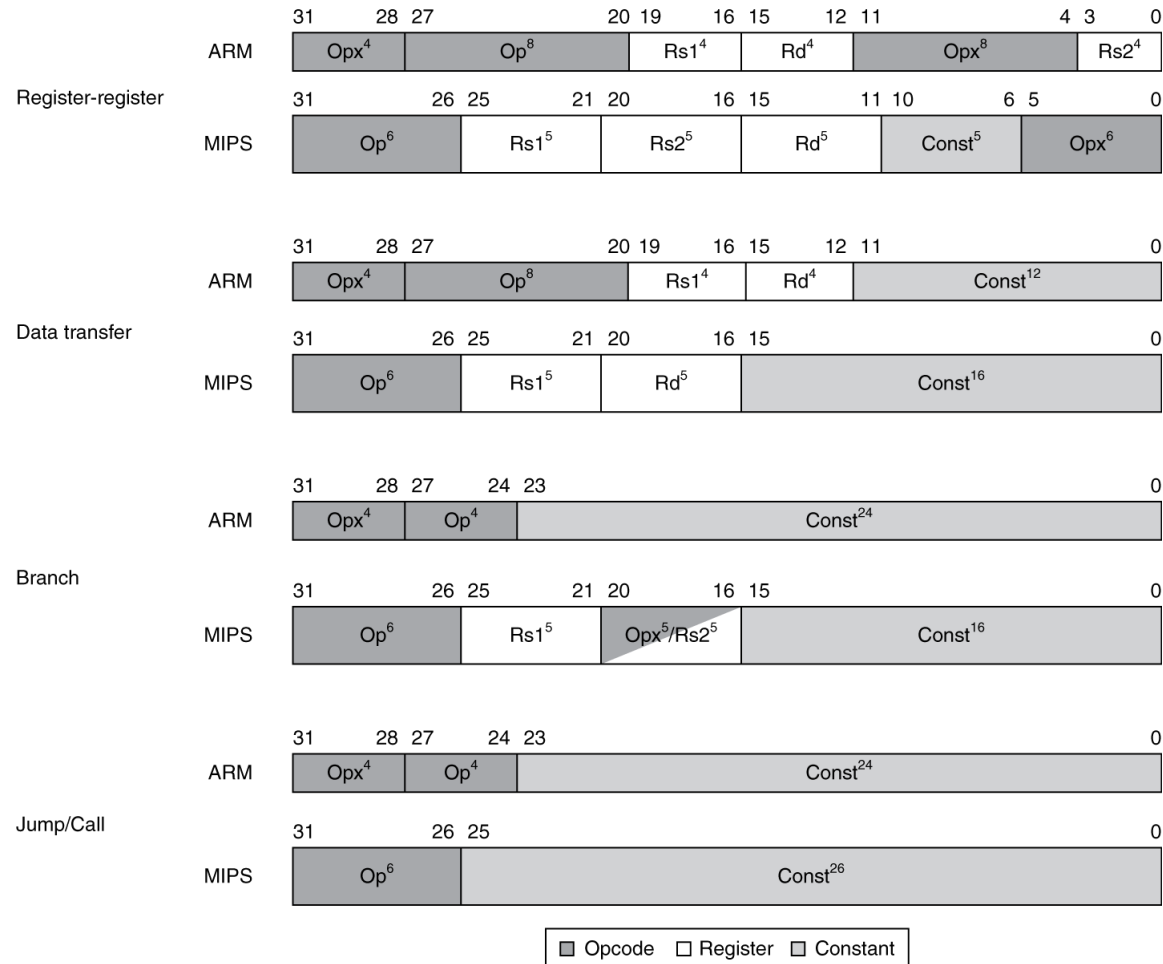
- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

# Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

# Instruction Encoding





# The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

# The Intel x86 ISA

- Further evolution...
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, ...
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

# The Intel x86 ISA

- And further...
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead...
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance  $\neq$  market success

# Basic x86 Registers

Name	31	0	Use
EAX			GPR 0
ECX			GPR 1
EDX			GPR 2
EBX			GPR 3
ESP			GPR 4
EBP			GPR 5
ESI			GPR 6
EDI			GPR 7
	CS		Code segment pointer
	SS		Stack segment pointer (top of stack)
	DS		Data segment pointer 0
	ES		Data segment pointer 1
	FS		Data segment pointer 2
	GS		Data segment pointer 3
EIP			Instruction pointer (PC)
EFLAGS			Condition codes

# Basic x86 Addressing Modes

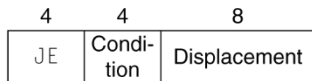
- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- Memory addressing modes
  - Address in register
  - Address =  $R_{\text{base}} + \text{displacement}$
  - Address =  $R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}}$  (scale = 0, 1, 2, or 3)

# x86 Instruction Encoding

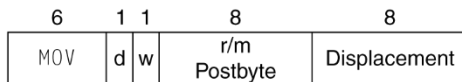
a. JE EIP + displacement



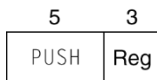
b. CALL



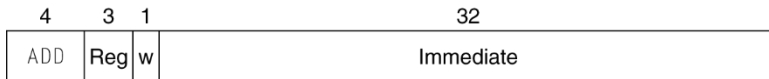
c. MOV EBX, [EDI + 45]



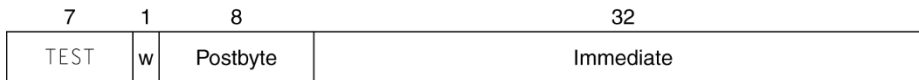
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, ...

# Implementing IA-32

- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions

# ARM v8 Instructions

- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
  - Changes from v7:
    - No conditional execution field
    - Immediate field is 12-bit constant
    - Dropped load/store multiple
    - PC is no longer a GPR
    - GPR set expanded to 32
    - Addressing modes work for all word sizes
    - Divide instruction
    - Branch if equal/branch if not equal instructions

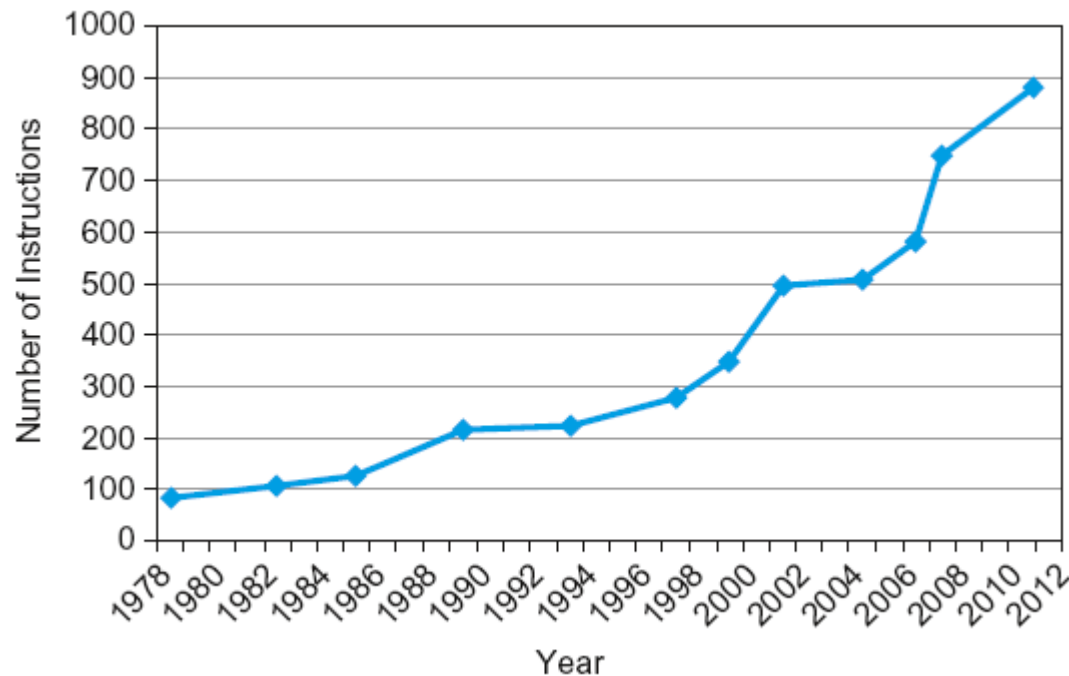


# Fallacies

- Powerful instruction  $\Rightarrow$  higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code  $\Rightarrow$  more errors and less productivity

# Fallacies

- Backward compatibility  $\Rightarrow$  instruction set doesn't change
  - But they do accrete more instructions



x86 instruction set

# Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped

# Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - c.f. x86

# Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	add, sub, addi	16%	48%
Data transfer	lw, sw, lb, lbu, lh, lhu, sb, lui	35%	36%
Logical	and, or, nor, andi, ori, sll, srl	12%	4%
Cond. Branch	beq, bne, slt, slti, sltiu	34%	8%
Jump	j, jr, jal	2%	0%