

RISC-V Architecture II

'22H2

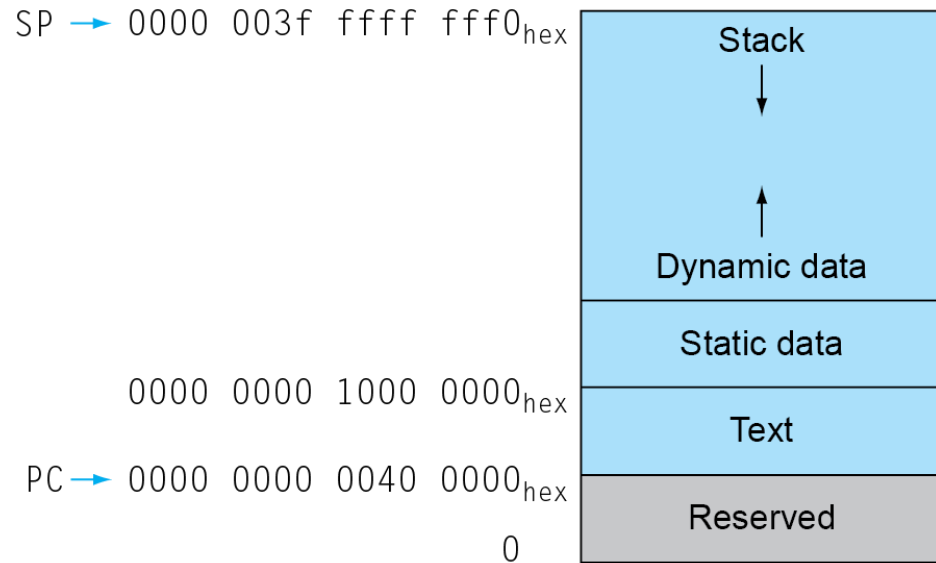
송 인 식

Outline

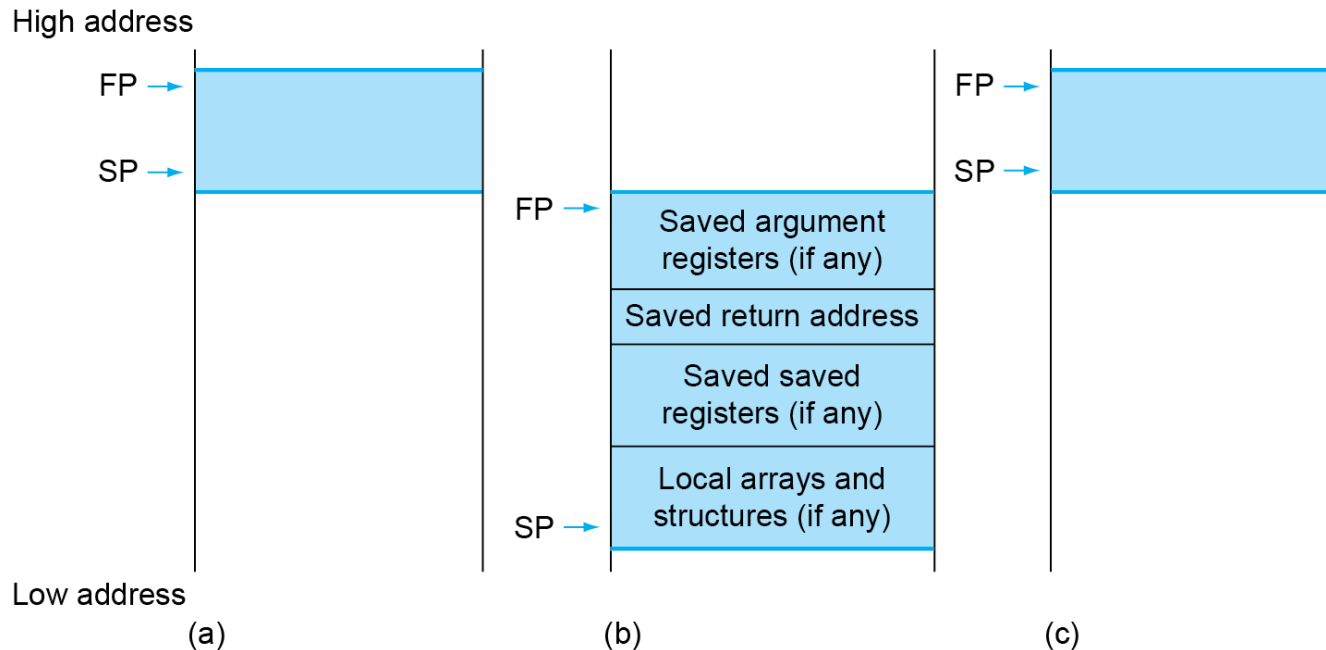
- Machine-level Representation of Programs
- Other ISAs

Memory Layout

- Text: program code
- Static data: global variables
 - e.g., static variables in C, constant arrays and strings
 - x3 (global pointer) initialized to address allowing \pm offsets into this segment
- Dynamic data: heap
 - e.g., malloc in C, new in Java
- Stack: automatic storage

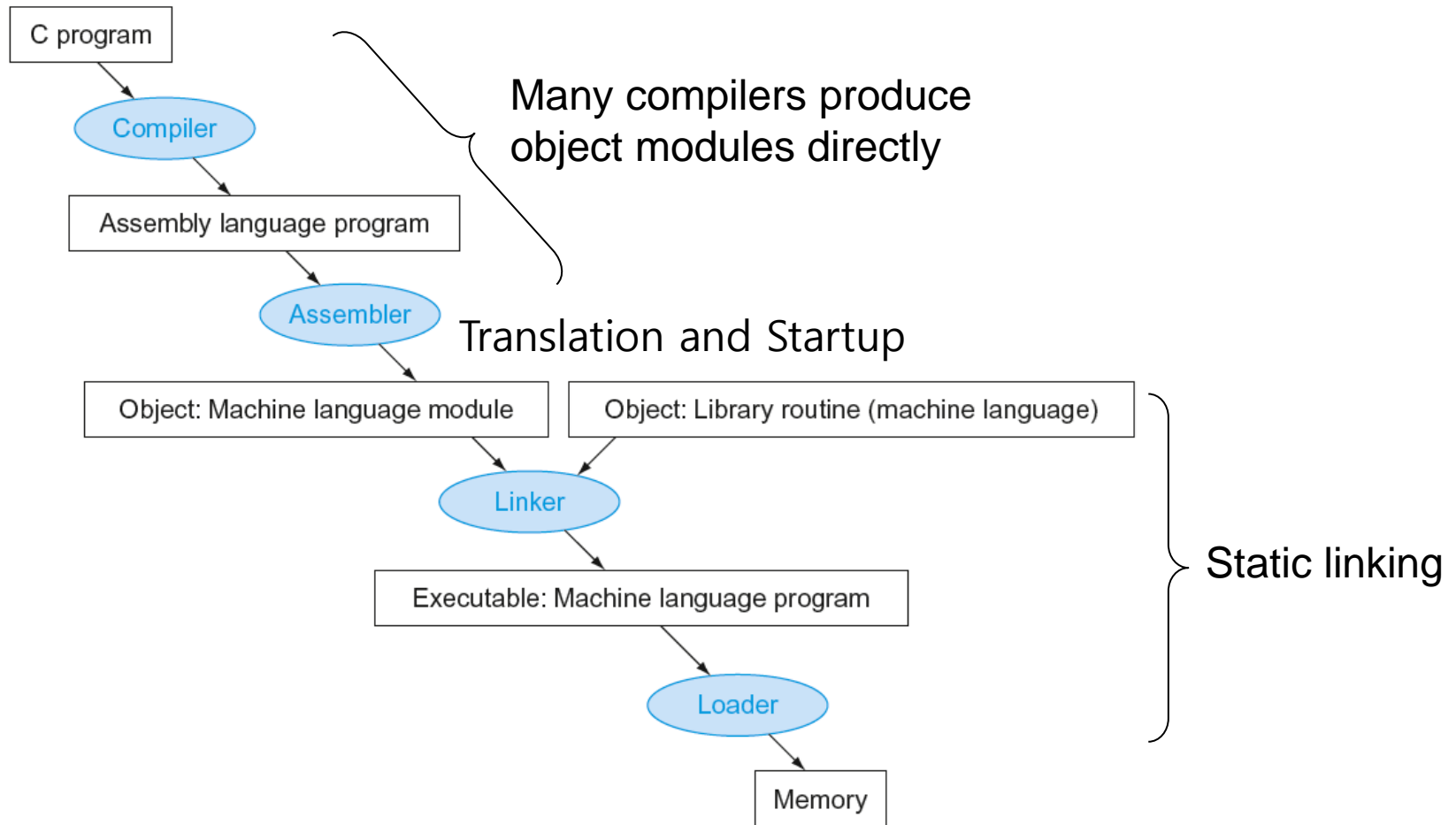


Local Data on the Stack



- Local data allocated by callee
 - e.g., C automatic variables
- Procedure frame (activation record)
 - Used by some compilers to manage stack storage

Translation and Startup



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 x10, ... 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

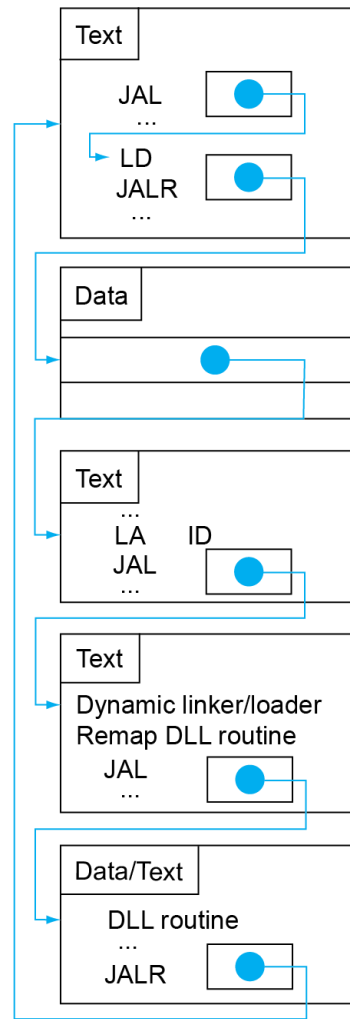
Lazy Linkage

Indirection table

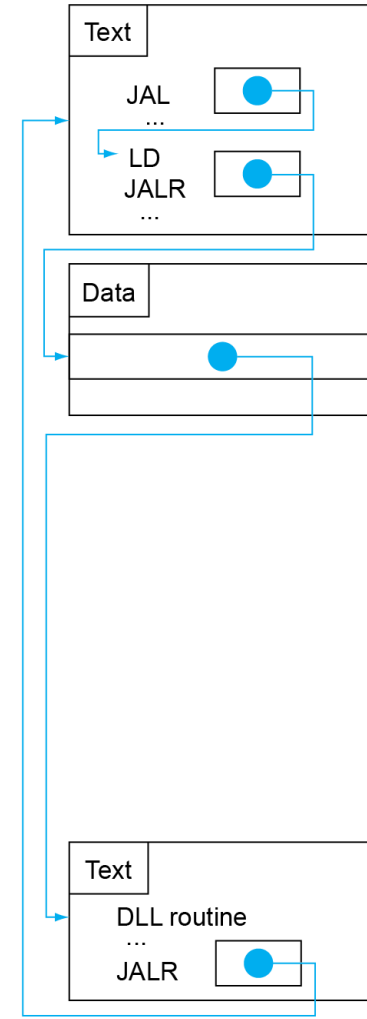
Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

Dynamically
mapped code

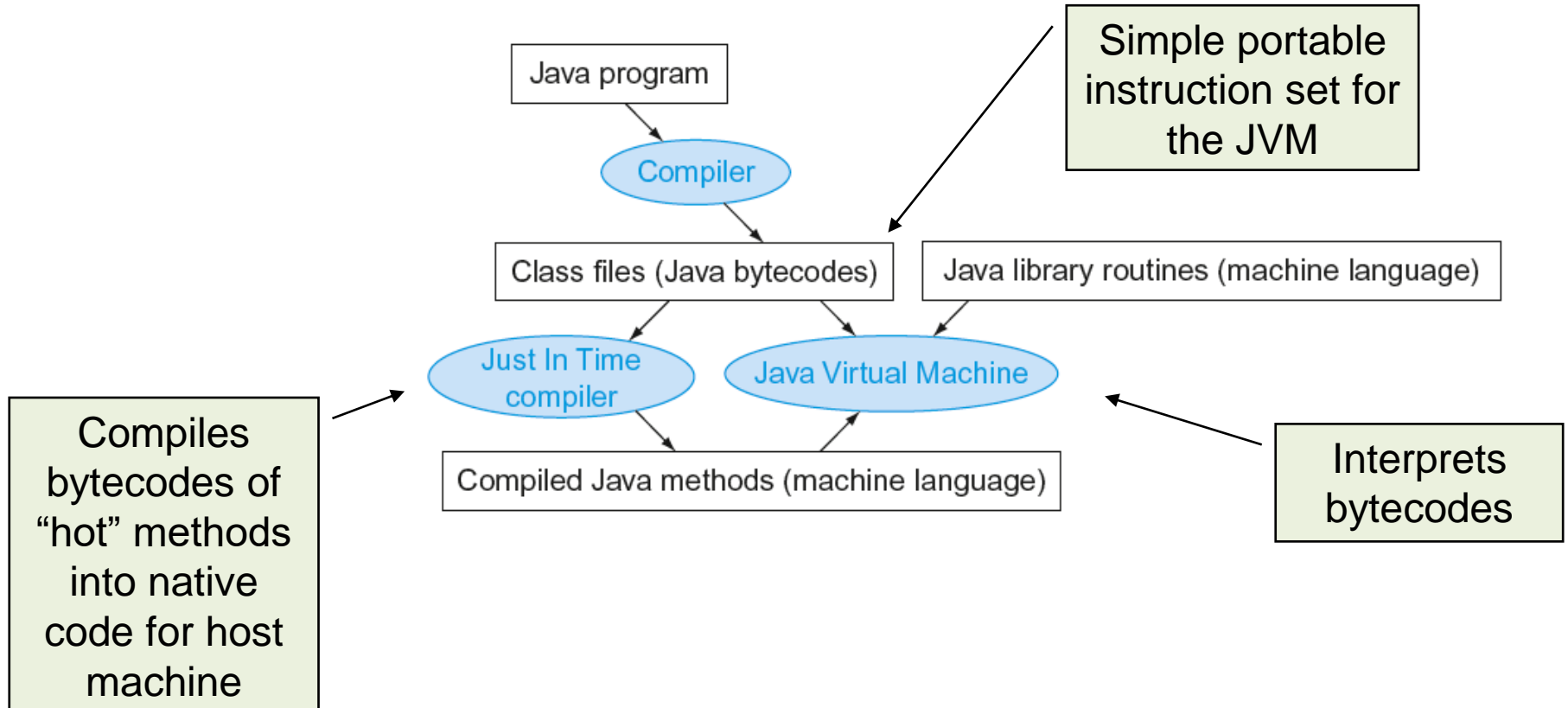


(a) First call to DLL routine



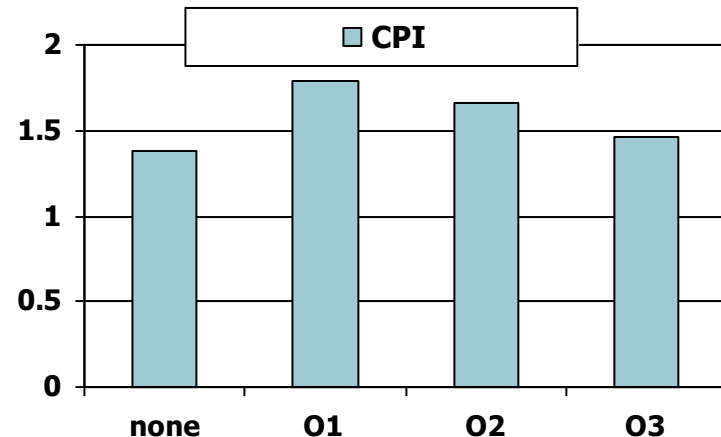
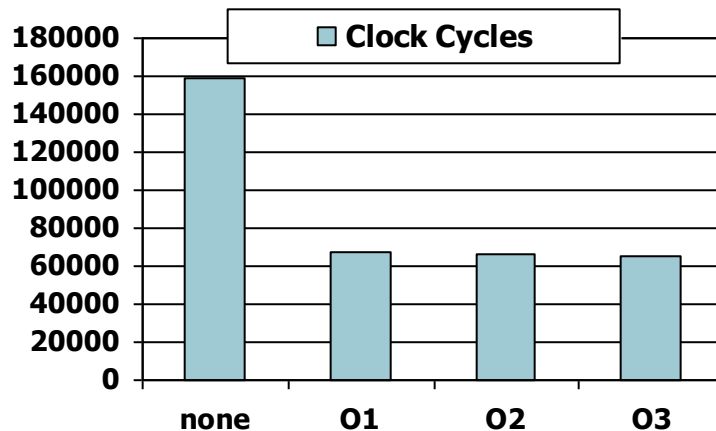
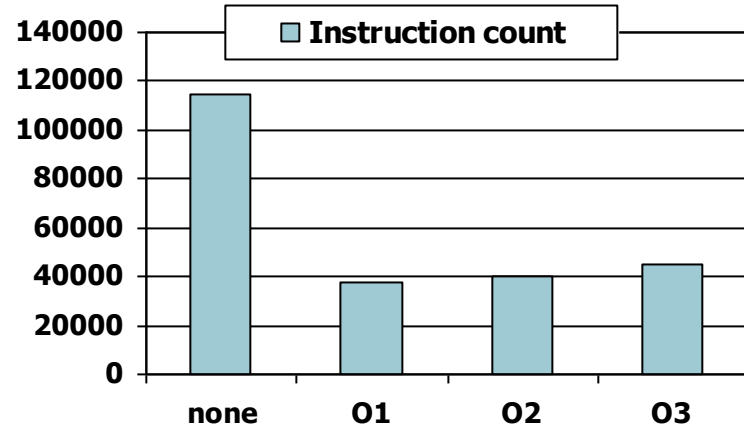
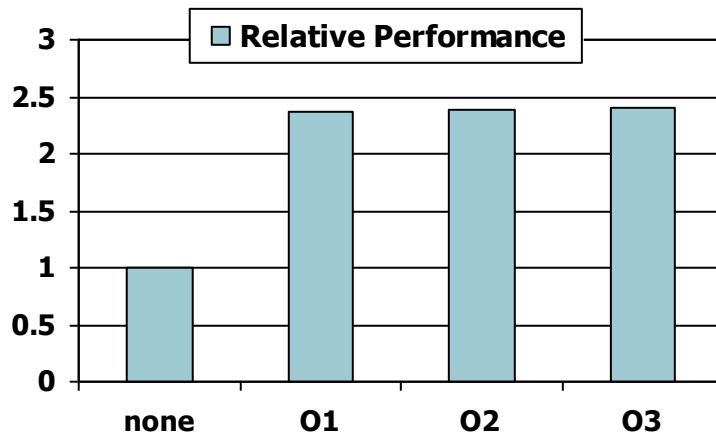
(b) Subsequent calls to DLL routine

Starting Java Applications

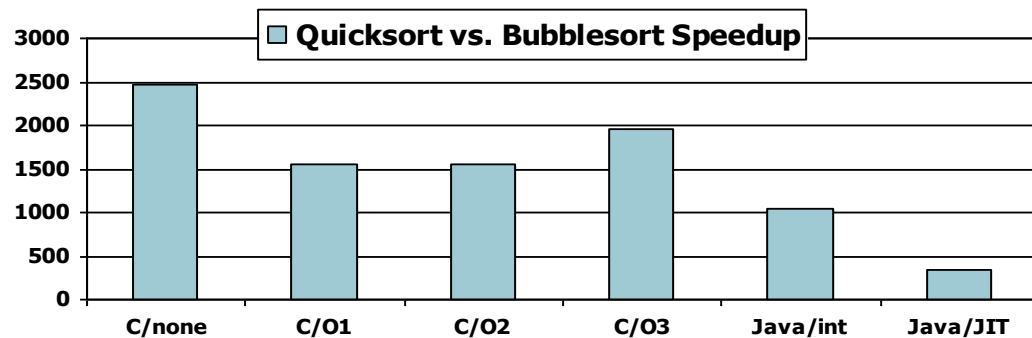
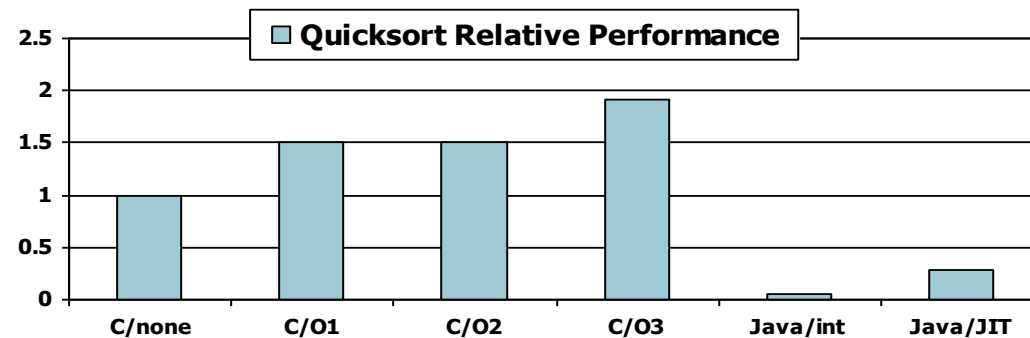
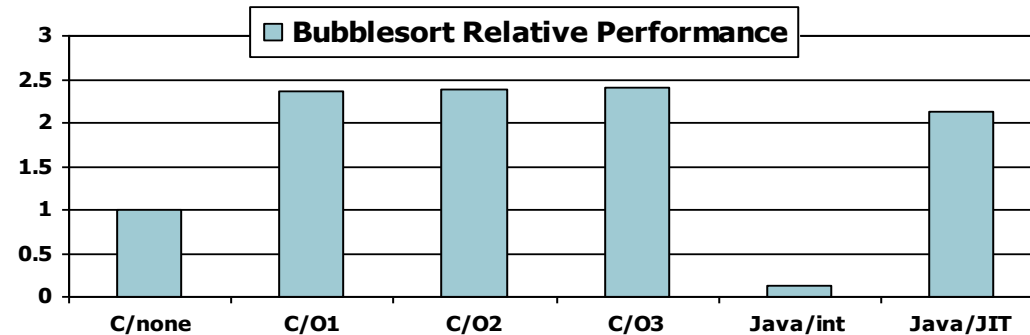


Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux



Effect of Compiler Optimization



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!

Outline

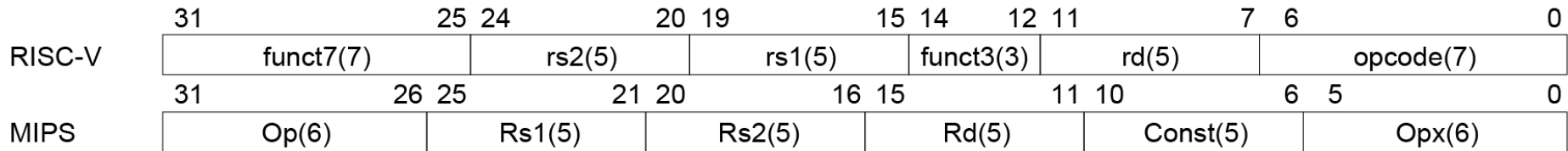
- Machine-level Representation of Programs
- Other ISAs

MIPS Instructions

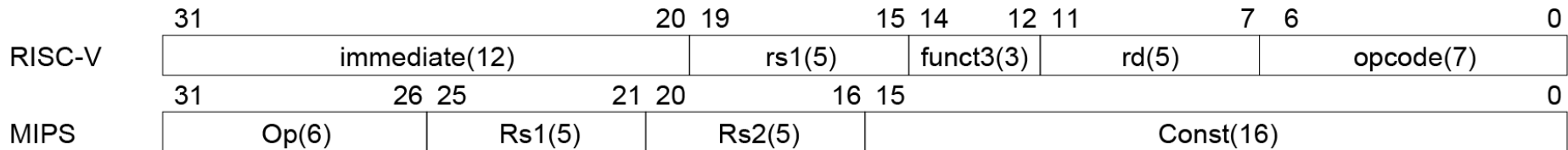
- MIPS: commercial predecessor to RISC-V
- Similar basic set of instructions
 - 32-bit instructions
 - 32 general purpose registers, register 0 is always 0
 - 32 floating-point registers
 - Memory accessed only by load/store instructions
 - Consistent use of addressing modes for all data sizes
- Different conditional branches
 - For $<$, \leq , $>$, \geq
 - RISC-V: `blt`, `bge`, `bltu`, `bgeu`
 - MIPS: `slt`, `sltu` (set less than, result is 0 or 1)
 - Then use `beq`, `bne` to complete the branch

Instruction Encoding

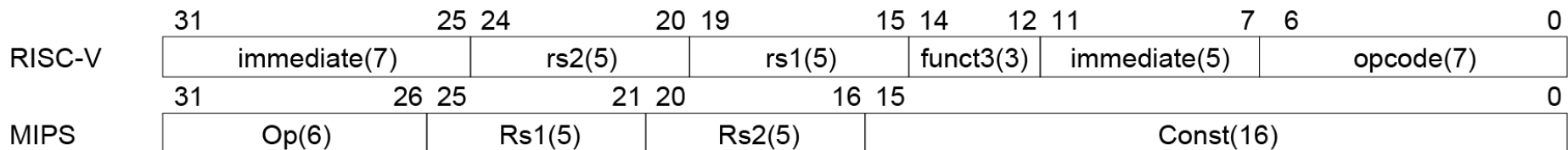
Register-register



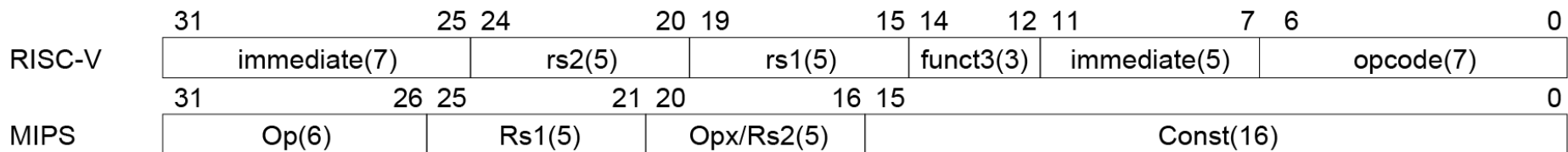
Load



Store



Branch



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

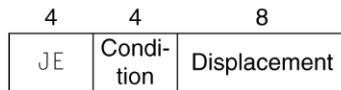
- Two operands per instruction

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)
 - Address = $R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}} + \text{displacement}$

x86 Instruction Encoding

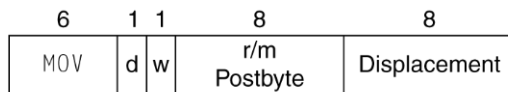
a. JE EIP + displacement



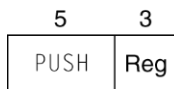
b. CALL



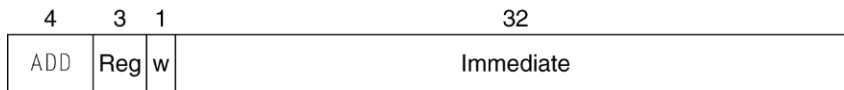
c. MOV EBX, [EDI + 45]



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

CISC (Complex Instruction Set Computer)

- Add instructions to perform “typical” programming tasks
 - DEC PDP-11 & VAX, IBM System/360, Motorola 68000, IA-32, Intel 64, ...
- Stack-oriented instruction set
 - Use stack to pass arguments, save program counter, etc.
 - Explicit push and pop instructions
- Arithmetic instructions can access memory
 - Requires memory read and write during computation
 - Complex addressing modes
- Instructions have varying lengths
- Condition codes
 - Set as side effect of arithmetic and logical instructions

RISC (Reduced Instruction Set Computer)

- Philosophy: Fewer, simple instructions
 - Might take more to get given task done
 - Can execute them with small and fast hardware
 - Stanford MIPS, UCB RISC-V, Sun SPARC, IBM Power/PowerPC, ARM, SuperH, ...
- Register-oriented instruction set
 - Many more (typically 32+) registers
 - Use for arguments, return address, temporaries
- Only load and store instructions can access memory
- Each instruction has fixed size
- No condition codes
 - Test instructions return 0/1 in register

CISC vs. RISC

- Original debate
 - CISC proponents – easy for compiler, fewer code bytes
 - RISC proponents – better for optimizing compilers, can make run fast with simple chip design
- Current status
 - For desktop/server processors, choice of ISA not a technical issue
 - With enough hardware, can make anything run fast
 - Code compatibility more important
 - x86-64 adopted many RISC features
 - More registers, use them for argument passing
 - Hardware translates instructions to simpler μ ops
 - For embedded processors, RISC makes sense: smaller, cheaper, less power

Other RISC-V Instructions

- Base integer instructions (RV64I)
 - Those previously described, plus
 - `auipc rd, imm` // `rd = (imm<<12) + pc`
 - follow by `jalr` (adds 12-bit imm) for long jump
 - `slt`, `sltu`, `slti`, `sltui`: set less than (like MIPS)
 - `addw`, `subw`, `addiw`: 32-bit add/sub
 - `sllw`, `srlw`, `srlw`, `slliw`, `srliw`, `sraiw`: 32-bit shift
- 32-bit variant: RV32I
 - registers are 32-bits wide, 32-bit operations

Instruction Set Extensions

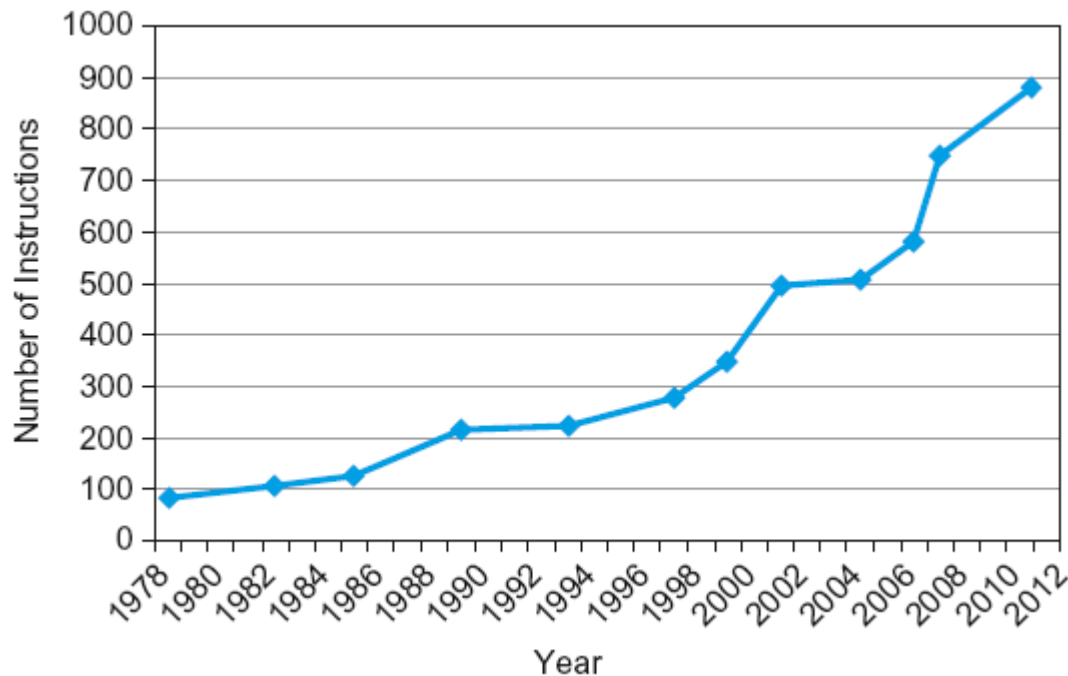
- M: integer multiply, divide, remainder
- A: atomic memory operations
- F: single-precision floating point
- D: double-precision floating point
- C: compressed instructions
 - 16-bit encoding for frequently used instructions

Fallacies

- Powerful instruction → 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 → more errors and less productivity

Fallacies

- Backward compatibility → instruction set doesn't change
 - But they do accrete more instructions



x86 instruction set

Concluding Remarks

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

Questions?