Chapter 2

Instructions: Language of the Computer II

Overview

- Review last lecture
- Branch and jump addressing
- Synchronization
- Linking and loading
- Arrays and pointer
- Sorting example
- MIPS and x86

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

The RISC-V Instruction Set

- Used as the example throughout the book
- Developed at UC Berkeley as open ISA
- Now managed by the RISC-V Foundation (riscv.org)
- Typical of many modern ISAs
  - See RISC-V Reference Data tear-out card
- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …

Register Operands

- Arithmetic instructions use register operands
- RISC-V has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Register x0 – x31
  - 32-bit data is called a "word"
  - 64-bit data is called a "doubleword"
- RISC-V in 32-bit and 64-bit variants
  - Not a big difference in the instructions
  - Book uses 32-bit, lab/Venus uses 32-bit

Register File

- All (but one) register are identical
- Register number 0 (x0) is hardwired to 0
  - Cannot be changed
  - Allows some optimization in the code

- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

Register Operand Example

- C code:

```
f = (g + h) - (i + j);
```

  - f, …, j in x19, x20, …, x23

- Compiled RISC-V code:

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

Memory Operand Example

- C code:

```
A[12] = h + A[8];
```

  - h in x21, base address of A in x22
    - Base address is where the array starts in memory
- Compiled RISC-V code:
  - Index 8 requires offset of 32
    - 4 bytes per word

```
lw      x9, 32(x22)
add     x9, x21, x9
sw      x9, 48(x22)
```

Immediate Operands

- Constant data specified in an instruction

```
addi x22, x22, 4
```

- Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

RISC-V R-format Instructions

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Instruction fields
  - opcode: operation code
  - rd: destination register number
  - funct3: 3-bit function code (additional opcode)
  - rs1: the first source register number
  - rs2: the second source register number
  - funct7: 7-bit function code (additional opcode)

RISC-V I-format Instructions

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

- Immediate arithmetic and load instructions
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2s-complement, sign extended
- *Design Principle 3:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

RISC-V S-format Instructions

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Shift left	<<	<<	`slli`
Shift right	>>	>>>	`srli`
Bit-by-bit AND	&	&	`and, andi`
Bit-by-bit OR	|	|	`or, ori`
Bit-by-bit XOR	^	^	`xor, xori`
Bit-by-bit NOT	~	~	

- Useful for extracting and inserting groups of bits in a word

Conditional Operations

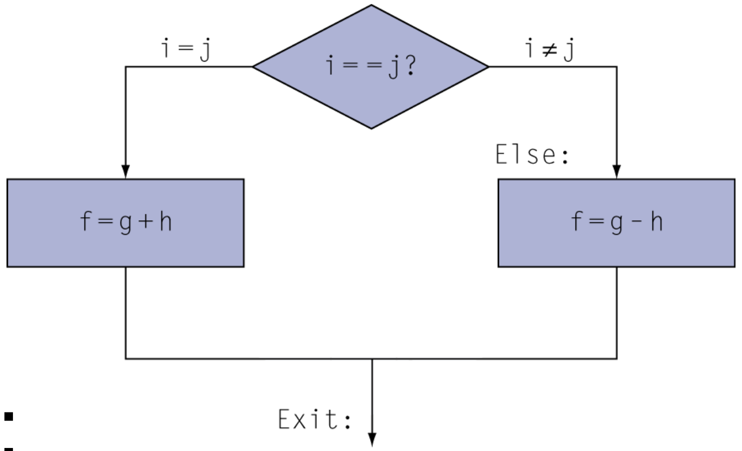
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs1, rs2, L1`
  - if (rs1 == rs2) branch to instruction labeled L1
- `bne rs1, rs2, L1`
  - if (rs1 != rs2) branch to instruction labeled L1

Compiling If Statements

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, … in x19, x20, …



- Compiled RISC-V code:

```
      bne x22, x23, Else
      add x19, x20, x21
      beq x0,x0,Exit // unconditional
Else: sub x19, x20, x21
Exit: …
```

Assembler calculates addresses

More Conditional Operations

- `blt rs1, rs2, L1`
  - if (rs1 < rs2) branch to instruction labeled L1
- `bge rs1, rs2, L1`
  - if (rs1 >= rs2) branch to instruction labeled L1
- Example
  - if (a > b) a += 1;
  - a in x22, b in x23

```
      bge  x23, x22, Exit      // branch if b >= a
      addi x22, x22, 1
Exit:
```

Procedure Call Instructions

- Procedure call: jump and link

  `jal x1, ProcedureLabel`

  - Address of following instruction put in x1
  - Jumps to target address
- Procedure return: jump and link register

  `jalr x0, 0(x1)`

  - Like jal, but jumps to 0 + address in x1
  - Use x0 as rd (x0 cannot be changed)
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Byte/Halfword/Word Operations

- RISC-V byte/halfword/word load/store
  - Load byte/halfword/word: Sign extend to 32 bits in rd
    - `lb rd, offset(rs1)`
    - `lh rd, offset(rs1)`
    - `lw rd, offset(rs1)`
  - Load byte/halfword/word unsigned: Zero extend to 32 bits in rd
    - `lbu rd, offset(rs1)`
    - `lhu rd, offset(rs1)`
    - `lwu rd, offset(rs1)`
  - Store byte/halfword/word: Store rightmost 8/16/32 bits
    - `sb rs2, offset(rs1)`
    - `sh rs2, offset(rs1)`
    - `sw rs2, offset(rs1)`

String Copy Example

- C code:
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ size_t i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

String Copy Example

- RISC-V code:

```
strcpy:
    addi sp,sp,-4// adjust stack for 1 word
    sw   x19,0(sp)       // push x19
    add  x19,x0,x0       // i=0
L1: add  x5,x19,x11      // x5 = addr of y[i]
    lbu  x6,0(x5)        // x6 = y[i]
    add  x7,x19,x10      // x7 = addr of x[i]
    sb   x6,0(x7)        // x[i] = y[i]
    beq  x6,x0,L2        // if y[i] == 0 then exit
    addi x19,x19,1       // i = i + 1
    jal  x0,L1           // next iteration of loop
L2: lw   x19,0(sp)       // restore saved x19
    addi sp,sp,4         // pop word from stack
    jalr x0,0(x1)        // and return
```

32-bit Constants

- Most constants are small
  - 12-bit immediate is sufficient
- For the occasional 32-bit constant

```
lui rd, constant
```

  - Copies 20-bit constant to bits [31:12] of rd
  - Clears bits [11:0] of rd to 0

```
lui x19, 976  // 0x003D0
```

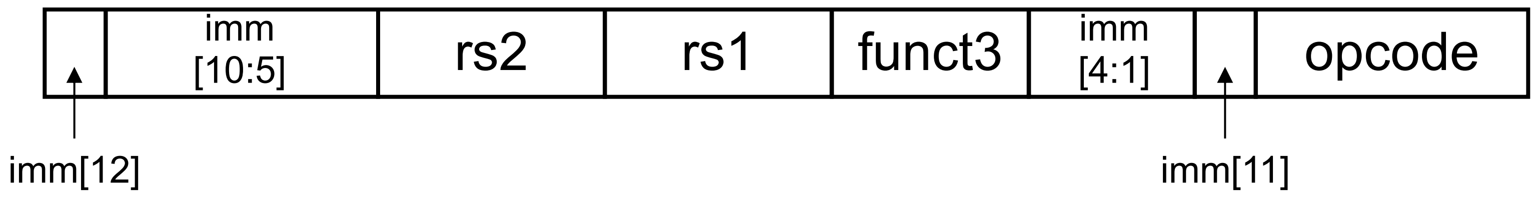
0000 0000 0011 1101 0000	0000 0000 0000

```
addi x19,x19,128  // 0x500
```

0000 0000 0011 1101 0000	0101 0000 0000

Branch Addressing

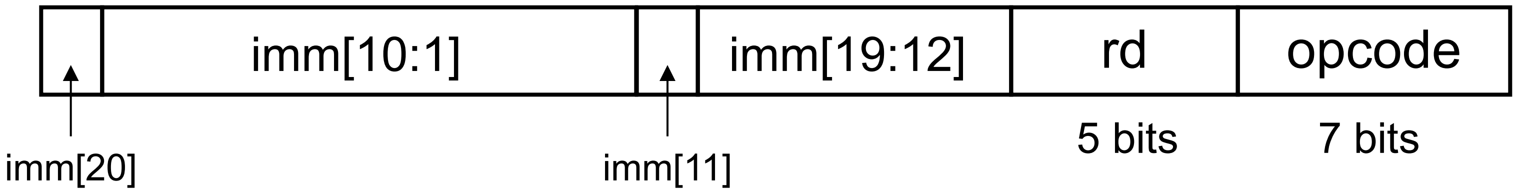
- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward
- SB format:



- PC-relative addressing
  - Target address = PC + immediate × 2

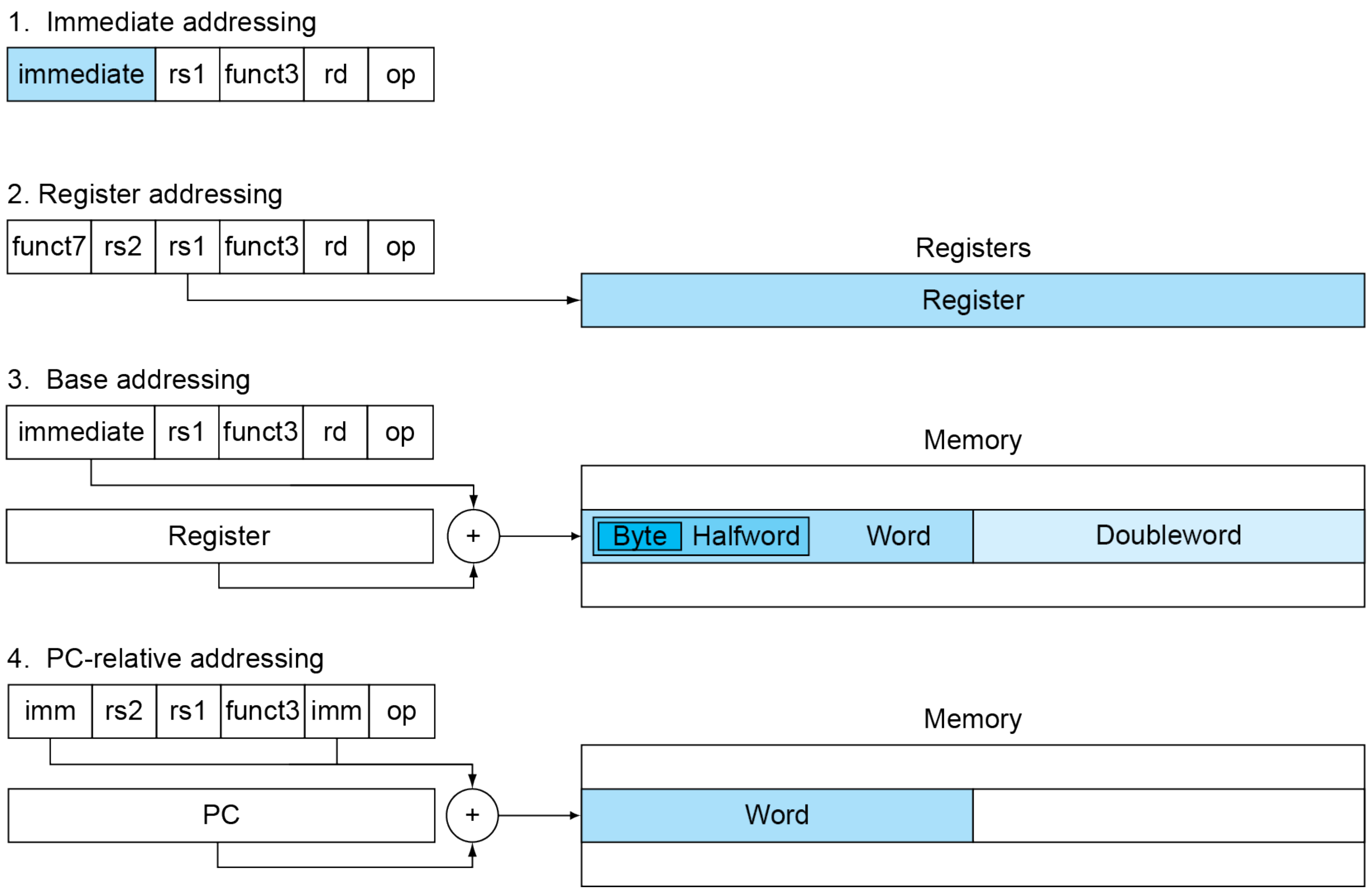
Jump Addressing

- Jump and link (`jal`) target uses 20-bit immediate for larger range (relative to PC)
- UJ format:



- For long jumps, eg, to 32-bit absolute address
  - lui: load address[31:12] to temp register
  - jalr: add address[11:0] and jump to target

RISC-V Addressing Summary



RISC-V Encoding Summary

Name	Field						Comments
(Field Size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

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 ↔ memory
  - Or an atomic pair of instructions

Synchronization in RISC-V

- Load reserved: `lr.w rd,(rs1)`
  - Load from address in rs1 to rd
  - Place reservation on memory address
- Store conditional: `sc.w rd,(rs1),rs2`
  - Store from rs2 to address in rs1
  - Succeeds if location not changed since the `lr.w`
    - Returns 0 in rd
  - Fails if location is changed
    - Returns non-zero value in rd

Synchronization in RISC-V

- Example 1: atomic swap (to test/set lock variable)

```
again:  lr.w x10,(x20)
        sc.w x11,(x20),x23 // X11 = status
        bne  x11,x0,again  // branch if store failed
        addi x23,x10,0     // X23 = loaded value
```

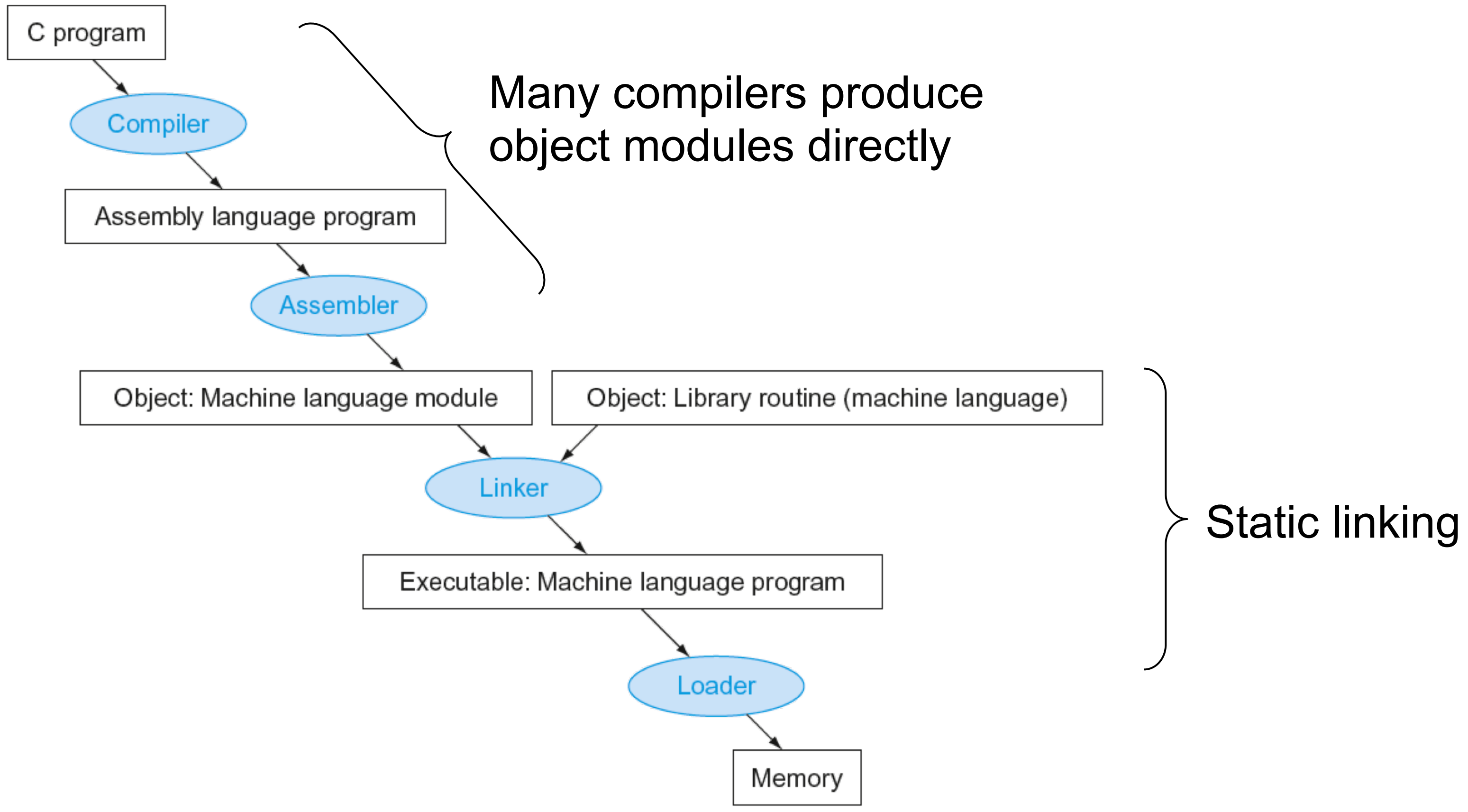
- Example 2: lock

```
        addi x12,x0,1          // copy locked value
again:  lr.w x10,(x20)         // read lock
        bne  x10,x0,again      // check if it is 0 yet
        sc.w x11,(x20),x12     // attempt to store
        bne  x11,x0,again      // branch if fails
```

- Unlock:

```
        sw   x0,0(x20)         // free lock
```

Translation and Startup



C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Many compilers produce object modules directly
Static linking

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

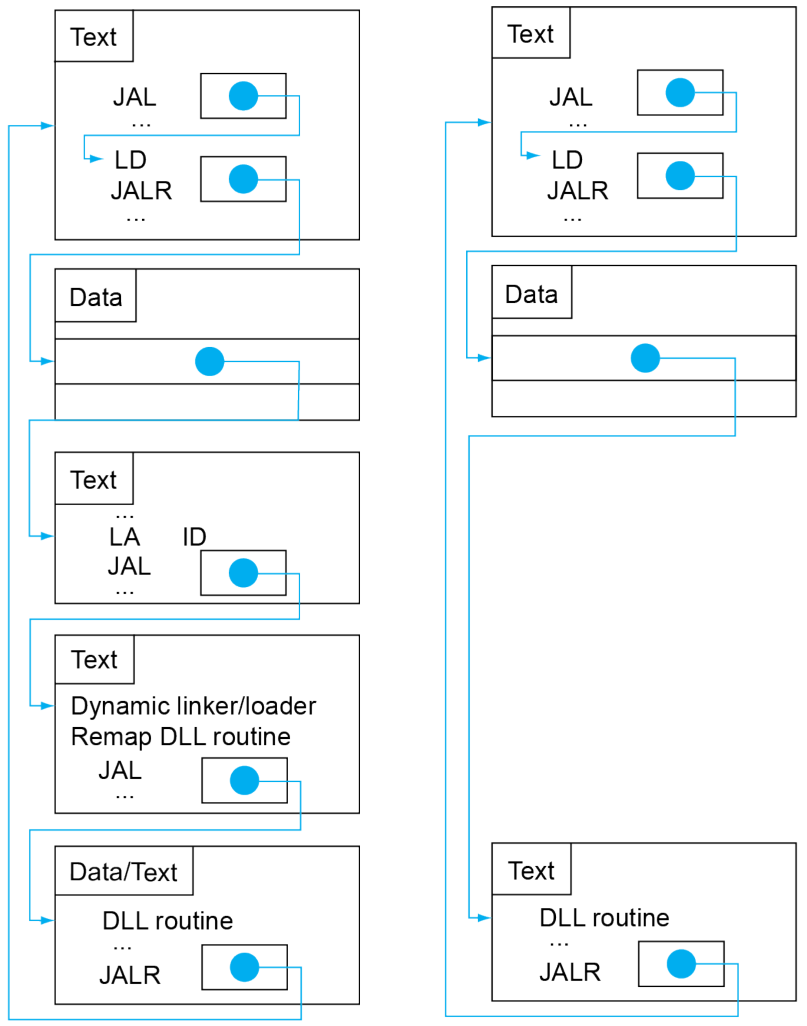
Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

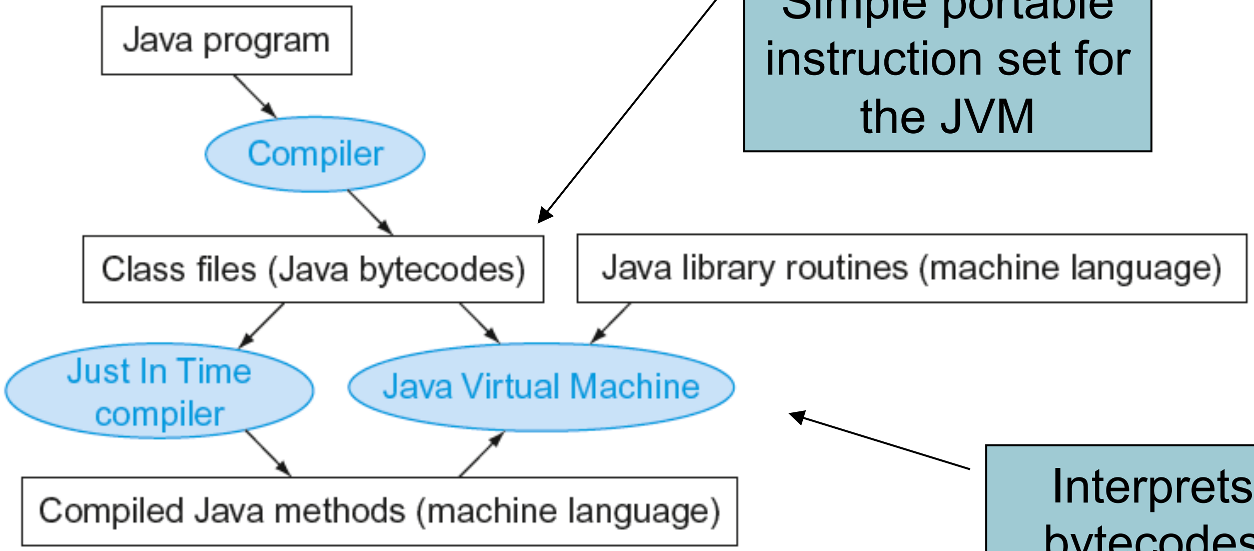
Dynamically
mapped code



(a) First call to DLL routine (b) Subsequent calls to DLL routine

Starting Java Applications

Simple portable instruction set for the JVM


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

Interprets bytecodes

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 x10, k in x11, temp in x5

The Procedure Swap

```
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 calling routine
```

The Sort Procedure in C

- Non-leaf (calls swap)

```
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);
    }
  }
}
```

  - v in x10, n in x11, i in x19, j in x20

The Outer Loop

- Skeleton of outer loop:
  - for (i = 0; i <n; i += 1) {

```
   li    x19,0               // i = 0
for1tst:
   bge   x19,x11,exit1     // go to exit1 if x19 ≥ x11 (i≥n)

   (body of outer for-loop)

   addi x19,x19,1          // i += 1
   j    for1tst            // branch to test of outer loop
exit1:
```

The Inner Loop

- Skeleton of inner loop:

```
    mv   x21, x10      // copy parameter x10 into x21
    mv   x22, x11      // copy parameter x11 into x22
```

  - for (j = i − 1; j >= 0 && v[j] > v[j + 1]; j − = 1) {

```
    addi x20,x19,-1    // j = i –1
for2tst:
    blt  x20,x0,exit2  // go to exit2 if X20 < 0 (j < 0)
    slli x5,x20,2      // reg x5 = j * 4
    add  x5,x10,x5     // reg x5 = v + (j * 4)
    lw   x6,0(x5)      // reg x6 = v[j]
    lw   x7,4(x5)      // reg x7 = v[j + 1]
    ble  x6,x7,exit2   // go to exit2 if x6 ≤ x7
    mv   x10, x21      // first swap parameter is v
    mv   x11, x20      // second swap parameter is j
    jal  x1,swap       // call swap
    addi x20,x20,-1    // j -= 1
    j    for2tst       // branch to test of inner loop
 exit2:
```

Preserving Registers

- Preserve saved registers:

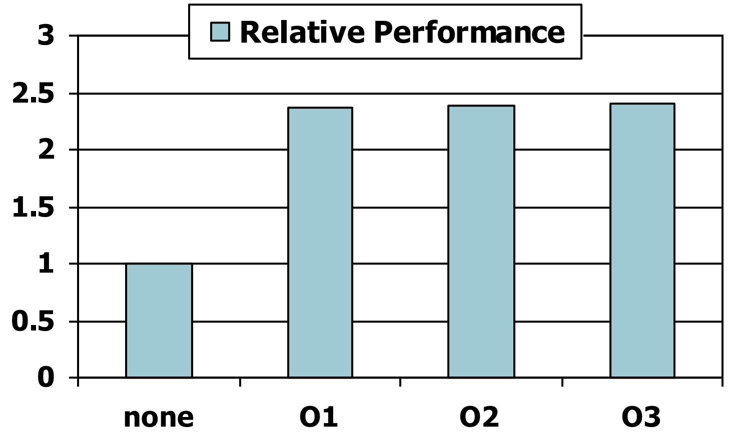
```
  addi sp, sp, -20 // make room on stack for 5 regs
  sw   x1, 16(sp)  // save x1 on stack
  sw   x22, 12(sp) // save x22 on stack
  sw   x21, 8(sp) // save x21 on stack
  sw   x20, 4(sp)  // save x20 on stack
  sw   x19, 0(sp)  // save x19 on stack
```

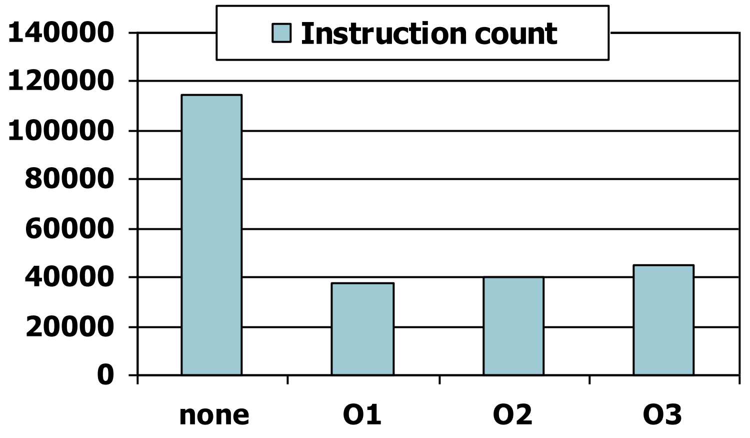
- Restore saved registers:

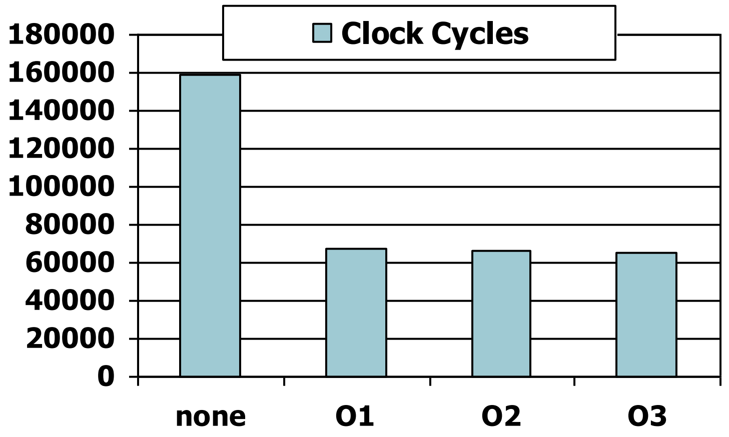
```
exit1:
  lw   x19, 0(sp)   // restore x19 from stack
  lw   x20, 4(sp)   // restore x20 from stack
  lw   x21, 8(sp) // restore x21 from stack
  lw   x22, 12(sp) // restore x22 from stack
  lw   x1, 16(sp)   // restore x1 from stack
  addi sp, sp, 20   // restore stack pointer
  jalr x0, 0(x1)
```

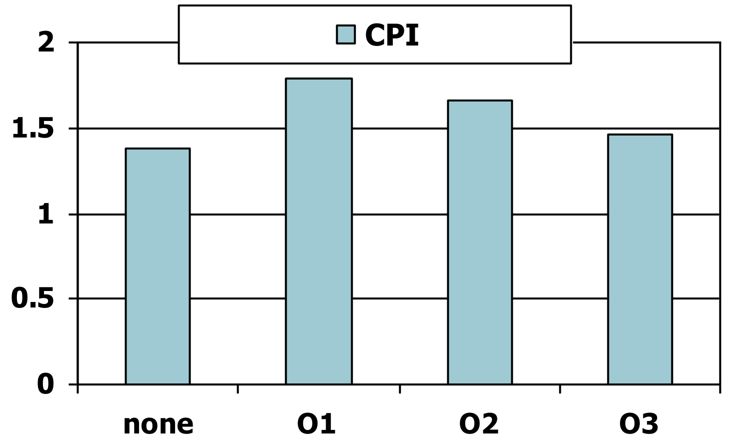
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

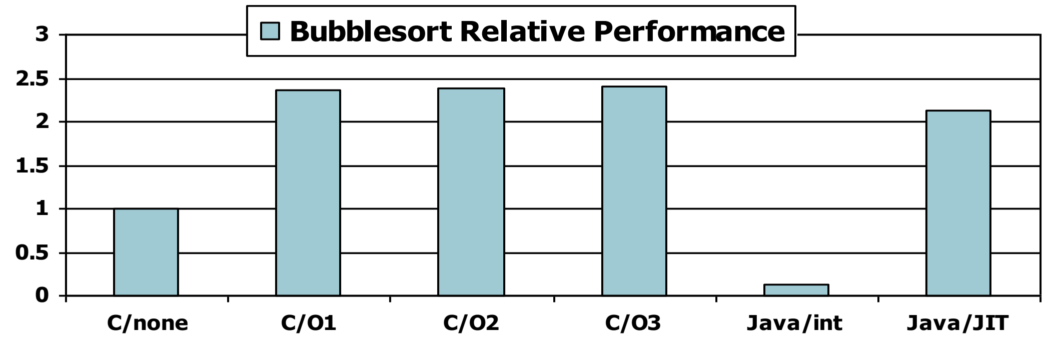


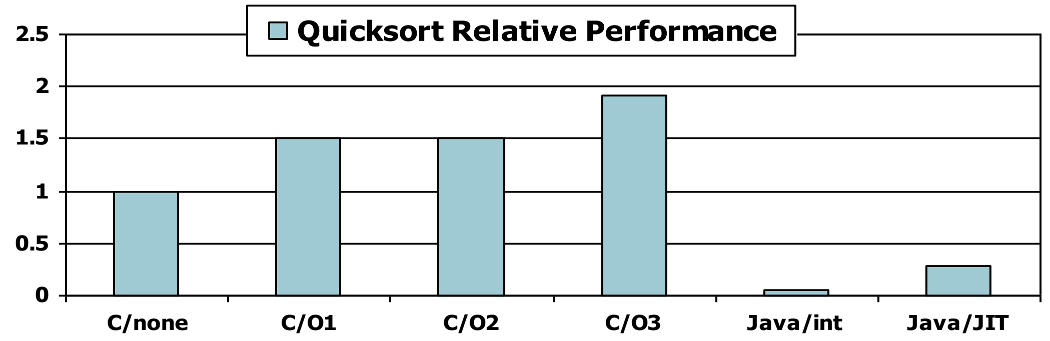


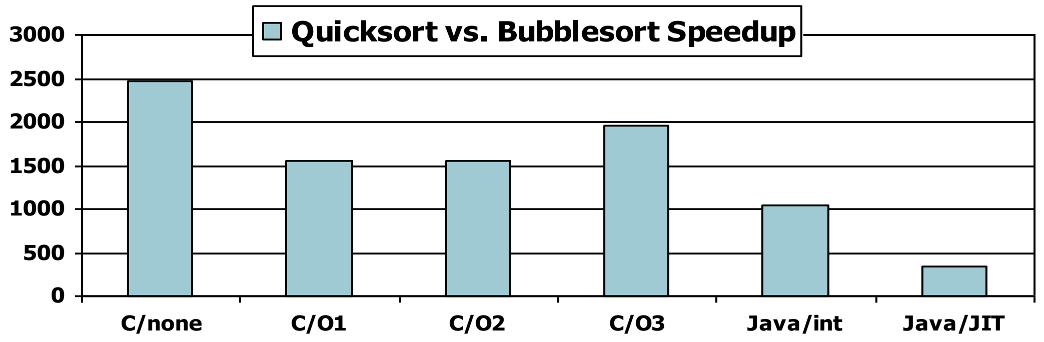




Effect of Language and Algorithm







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 an Array

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```
	```
	clear2(int *array, int size) {
	  int *p;
	  for (p = &array[0]; p < &array[size];
	       p = p + 1)
	    *p = 0;
	}
	```

```
   li   x5,0         // i = 0
loop1:
   slli x6,x5,2     // x6 = i * 4
   add  x7,x10,x6   // x7 = address
                    // of array[i]
   sw   x0,0(x7)    // array[i] = 0
   addi x5,x5,1     // i = i + 1
   blt  x5,x11,loop1  // if (i<size)
                      // go to loop1
```
	```
	   mv x5,x10         // p = address
	                     // of array[0]
	   slli x6,x11,2     // x6 = size * 4
	   add x7,x10,x6     // x7 = address
	                     // of array[size]
	loop2:
	   sw x0,0(x5)       // Memory[p] = 0
	   addi x5,x5,4      // p = p + 4
	   bltu x5,x7,loop2
	                     // if (p<&array[size])
	                     // go to 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

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 <, <=, >, >=
  - 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

	31 25	24 20	19 15	14 12	11 7	6 0
RISC-V	funct7(7)	rs2(5)	rs1(5)	funct3(3)	rd(5)	opcode(7)

	31 26	25 21	20 16	15 11	10 6	5 0
MIPS	Op(6)	Rs1(5)	Rs2(5)	Rd(5)	Const(5)	Opx(6)

Load

	31 20	19 15	14 12	11 7	6 0
RISC-V	immediate(12)	rs1(5)	funct3(3)	rd(5)	opcode(7)

	31 26	25 21	20 16	15 0
MIPS	Op(6)	Rs1(5)	Rs2(5)	Const(16)

Store

	31 25	24 20	19 15	14 12	11 7	6 0
RISC-V	immediate(7)	rs2(5)	rs1(5)	funct3(3)	immediate(5)	opcode(7)

	31 26	25 21	20 16	15 0
MIPS	Op(6)	Rs1(5)	Rs2(5)	Const(16)

Branch

	31 25	24 20	19 15	14 12	11 7	6 0
RISC-V	immediate(7)	rs2(5)	rs1(5)	funct3(3)	immediate(5)	opcode(7)

	31 26	25 21	20 16	15 0
MIPS	Op(6)	Rs1(5)	Opx/Rs2(5)	Const(16)

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
    - Assembly level compatibility
    - 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

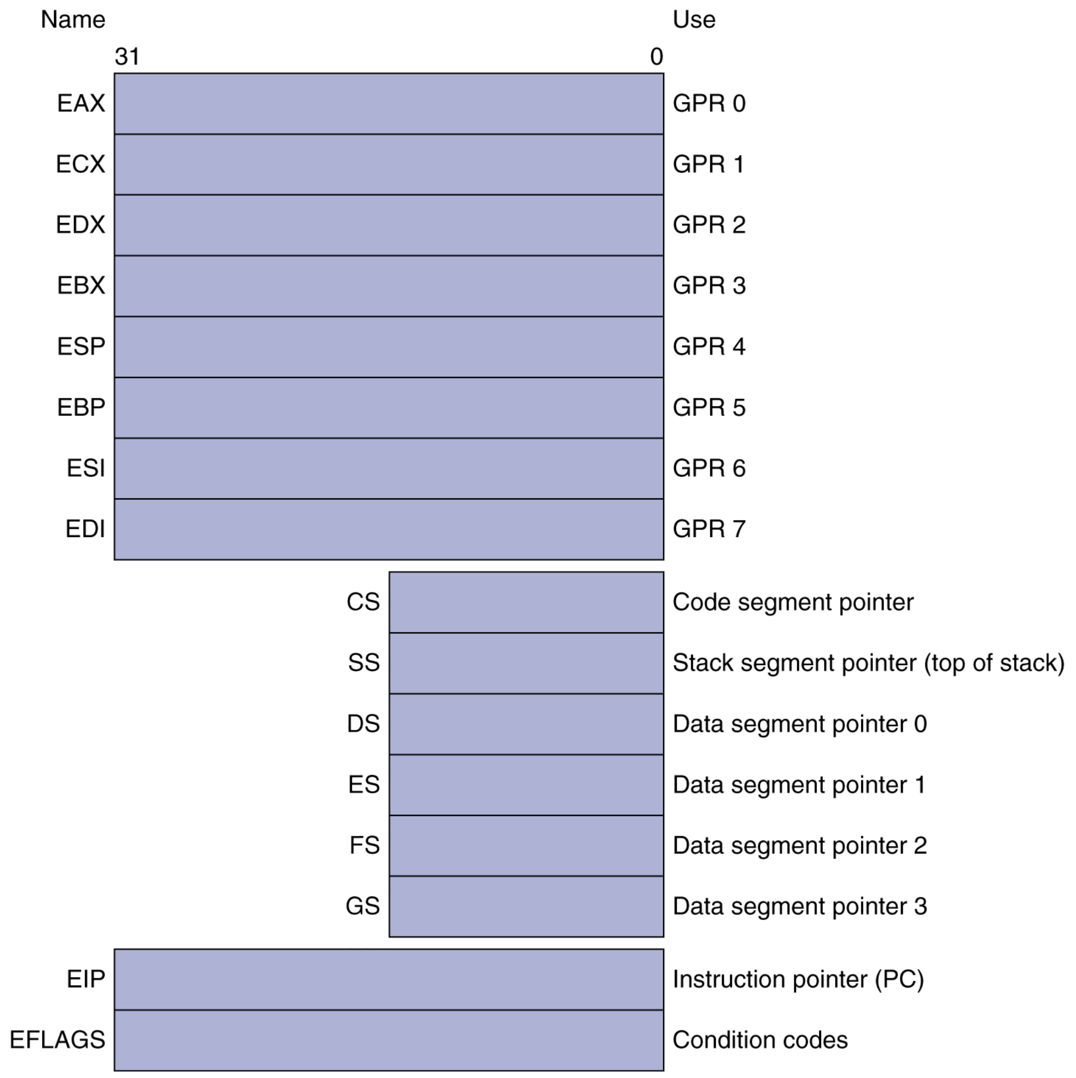
The Intel x86 ISA

- Further evolution…
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar
    - 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 ≠ market success

Basic x86 Registers



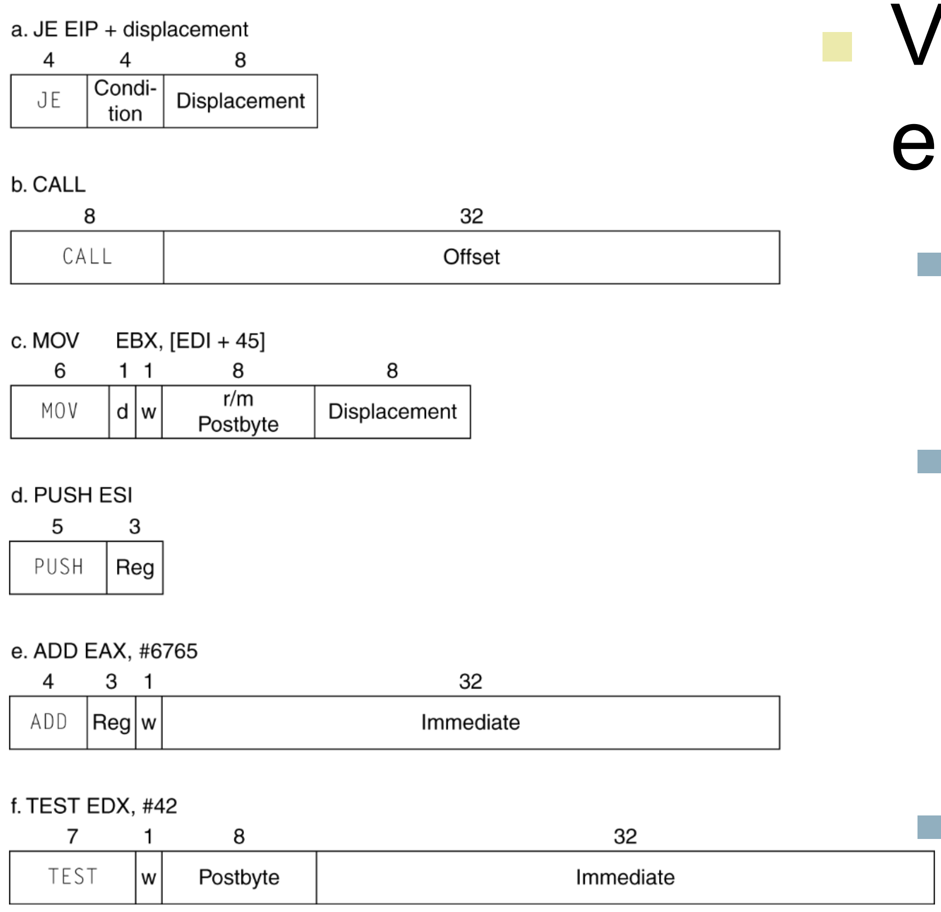
Basic x86 Addressing Modes

- 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_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

x86 Instruction Encoding



- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …
  - 1-15 bytes in length

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

Other RISC-V Instructions

- Base integer instructions (RV32I)
  - Those previously described, plus
  - auipc rd, immed // rd = (imm<<12) + pc
    - follow by jalr (adds 12-bit immed) for long jump
  - slt, sltu, slti, sltui: set less than (like MIPS)
- 64-bit variant: RV64I
  - registers are 64-bits wide, 64-bit operations

Instruction Set Extensions

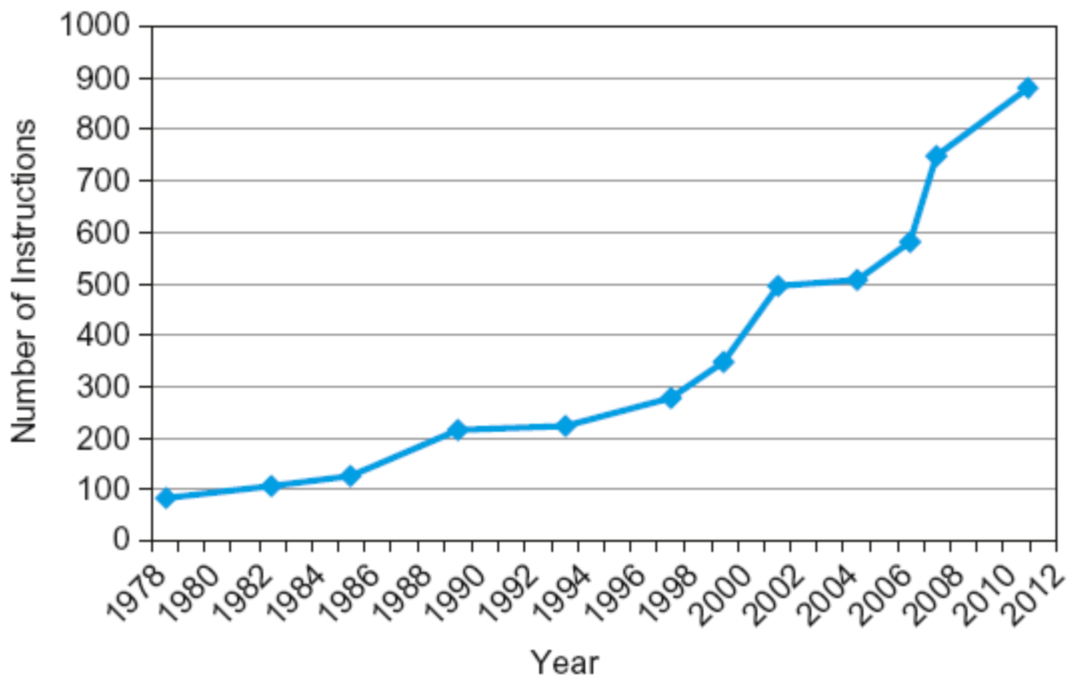
- M: integer multiply, divide, remainder
- A: atomic memory operations
- F: single-precision floating point
- D: double-precision floating point
- C: compressed instructions
  - 16-bit encoding for frequently used instructions

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

Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4/8, 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. 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