Role of the Compiler

Computer Organization

Sunday, 18 September 2022

TÉCNICO LISBOA

Many slides adapted from:
Computer Organization and Design,
Patterson & Hennessy
5th Edition, © 2014, MK

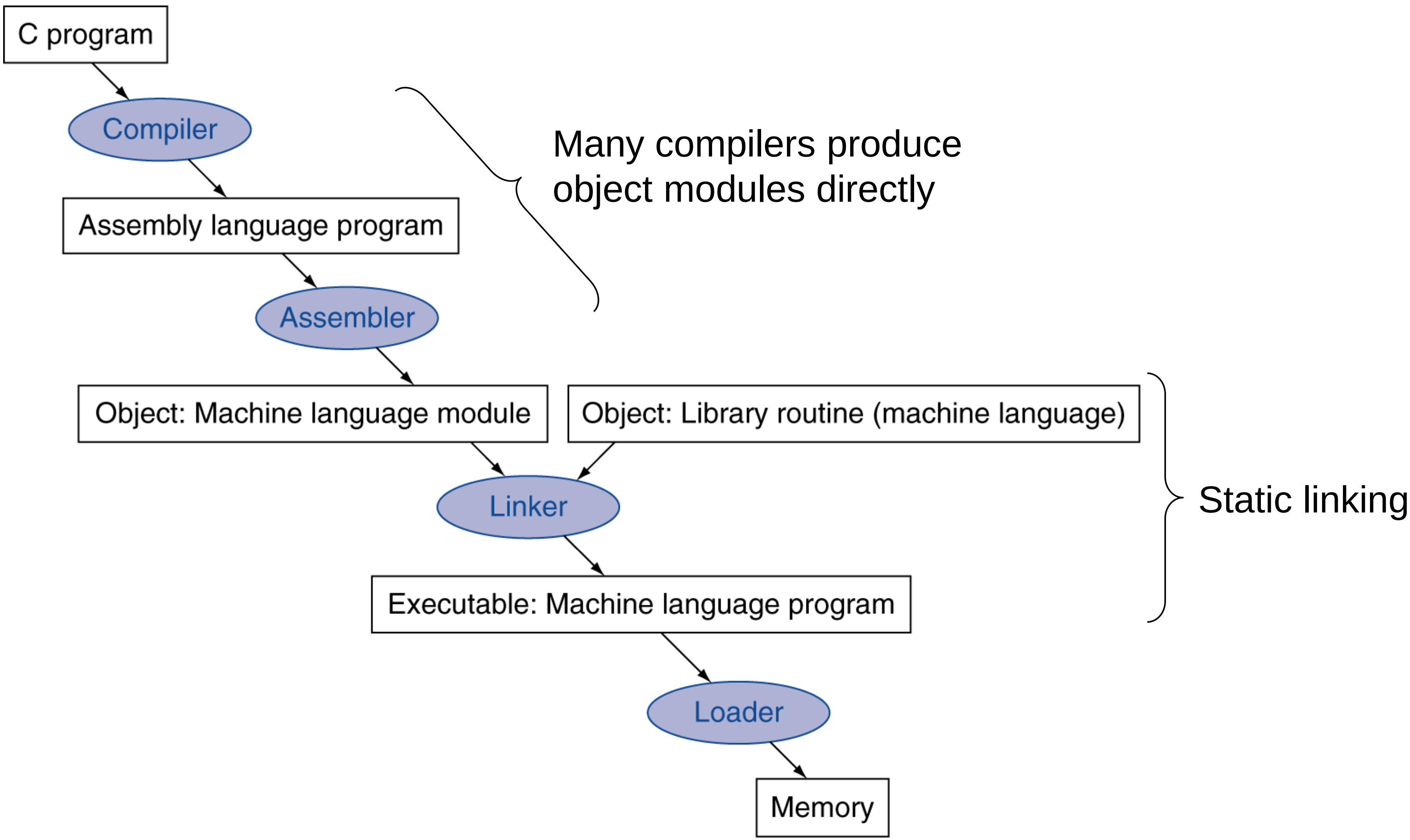
Chap. 2

Summary

- Previous Class
  - MIPS ISA

- Today:
  - Role of the compiler
  - Comparison between ISAs

Translation and Startup



Assembler Pseudo-instructions

- Most assembler instructions represent machine instructions one-to-one

- Pseudo-instructions: figments of the assembler's imagination

```
move $t0, $t1      →  add $t0, $zero, $t1

blt $t0, $t1, L    →  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

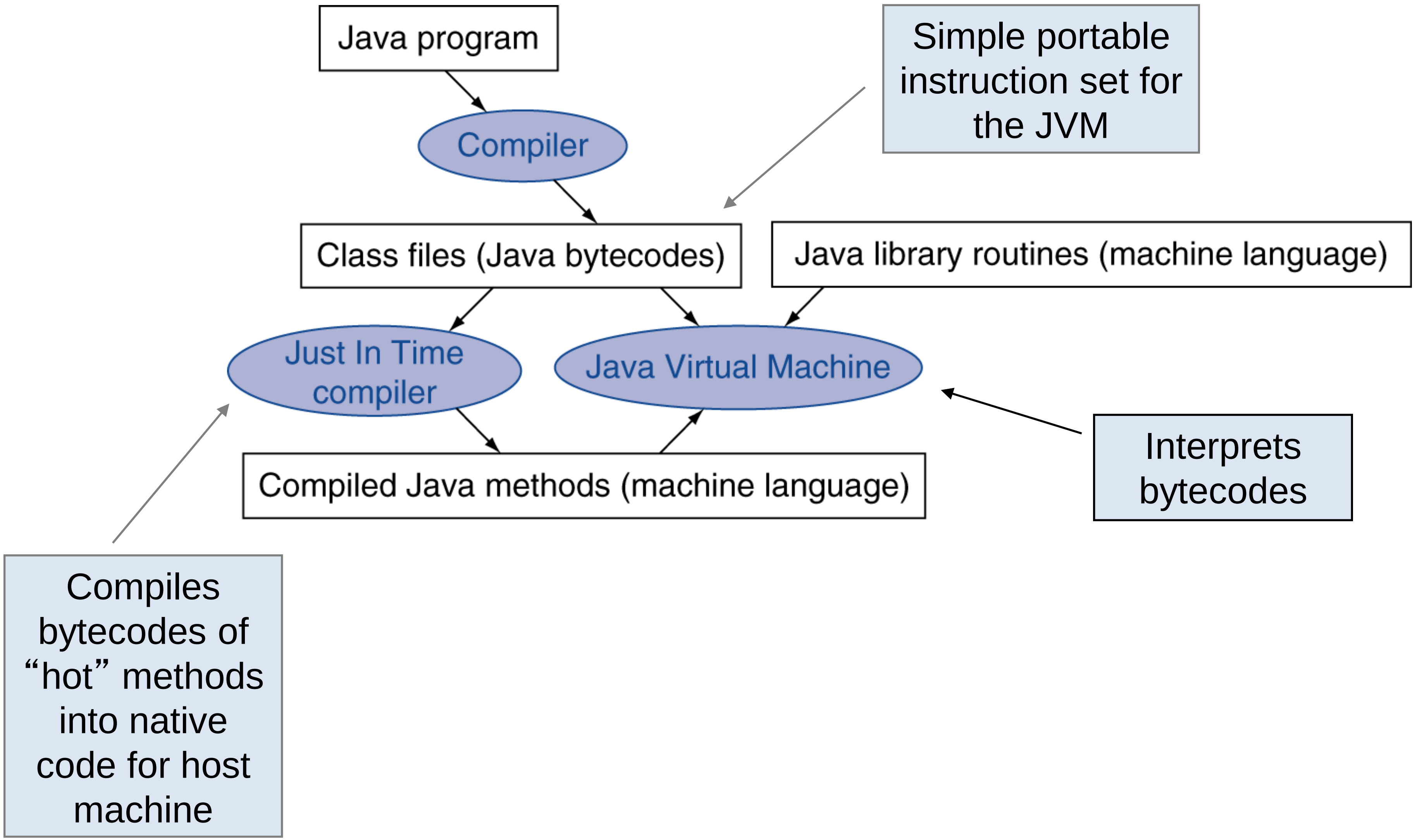
- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including `$sp`, `$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

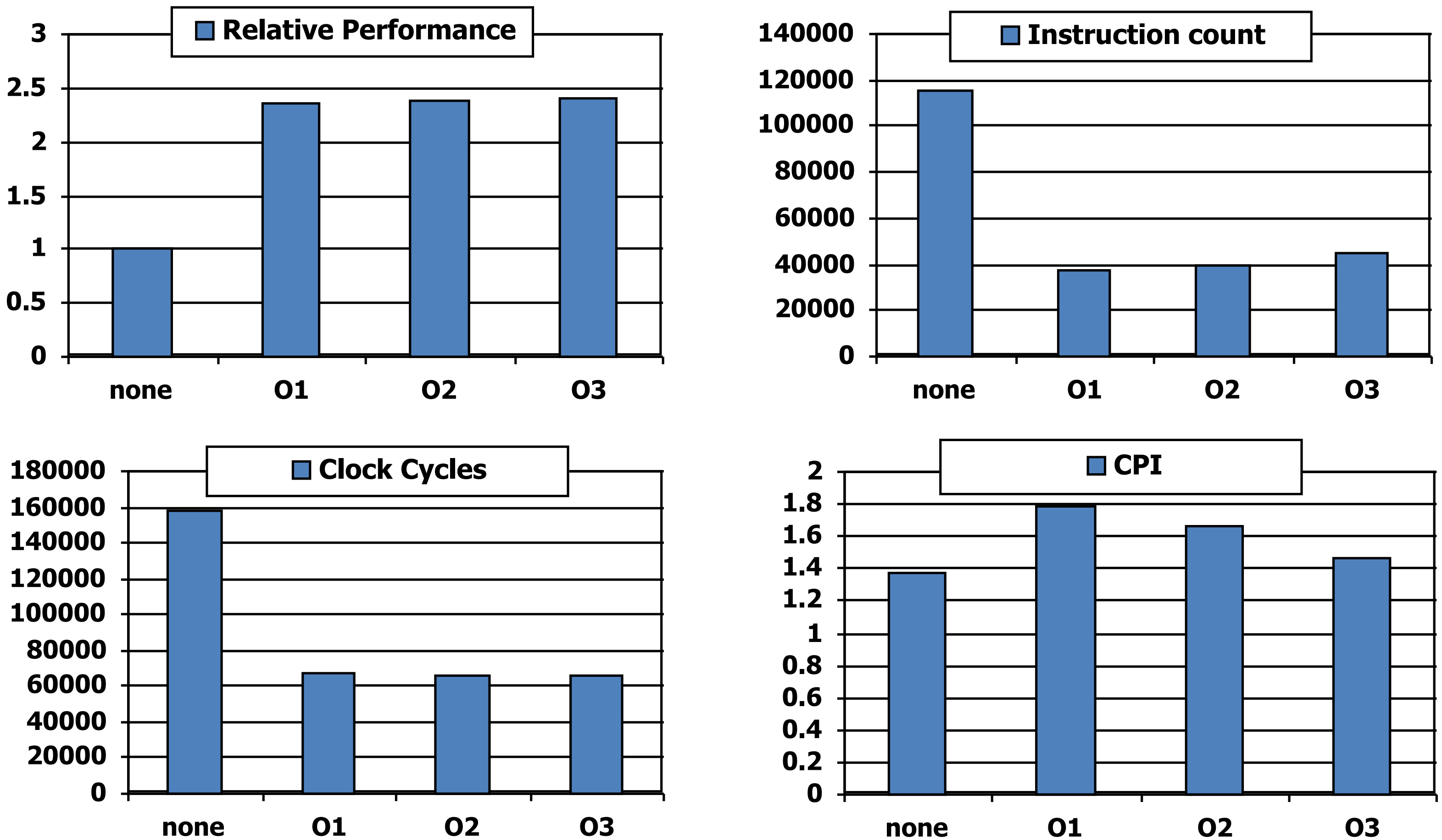
- Lazy Linkage
  - Linkage performed only when function called
  - Only functions actually used are linked

Starting Java Applications

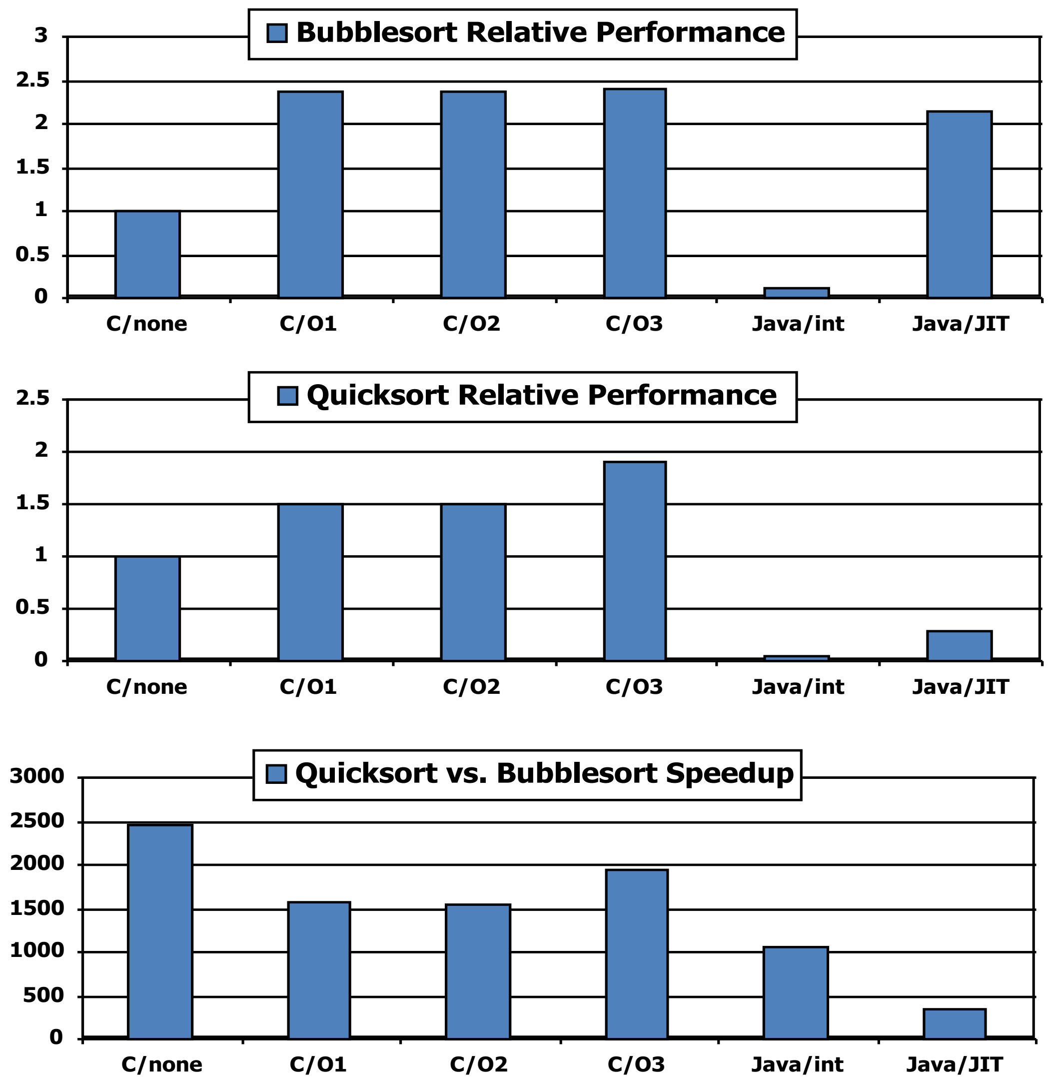


Effect of Compiler Optimization

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;
}
```

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

```
       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 Arrays vs. Pointers

- 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

Check@home: 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

Check@home: 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

Check@home: 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

x86 Instruction Encoding

a. JE EIP + displacement

4	4	8
JE	Condi-tion	Displacement

b. CALL

8	32
CALL	Offset

c. MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r/m Postbyte	Displacement

d. PUSH ESI

5	3
PUSH	Reg

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Immediate

f. TEST EDX, #42

7	1	8	32
TEST	w	Postbyte	Immediate

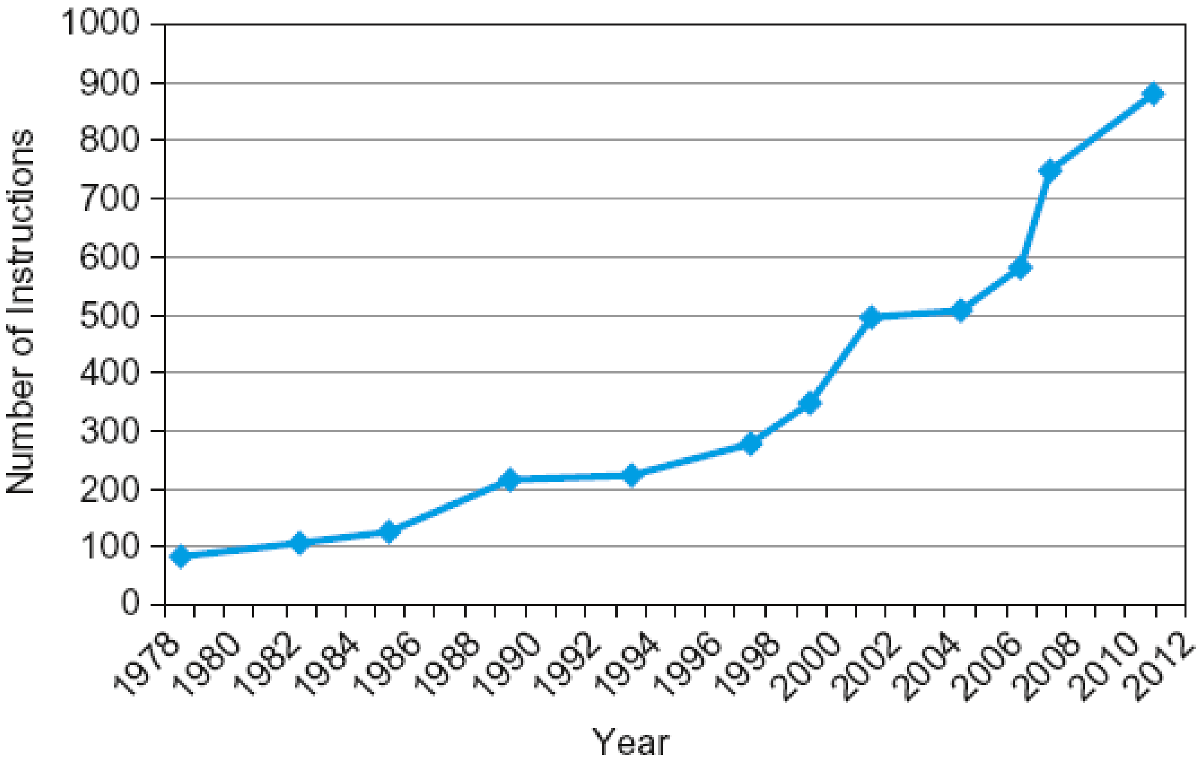
- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …

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

Check@home: Fallacies

- Backward compatibility $\Rightarrow$ instruction set doesn't change



x86 instruction set

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

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

- 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%

Role of the Compiler

Computer Organization

Sunday, 18 September 2022

TÉCNICO LISBOA

Many slides adapted from:
Computer Organization and Design,
Patterson & Hennessy
5th Edition, © 2014, MK