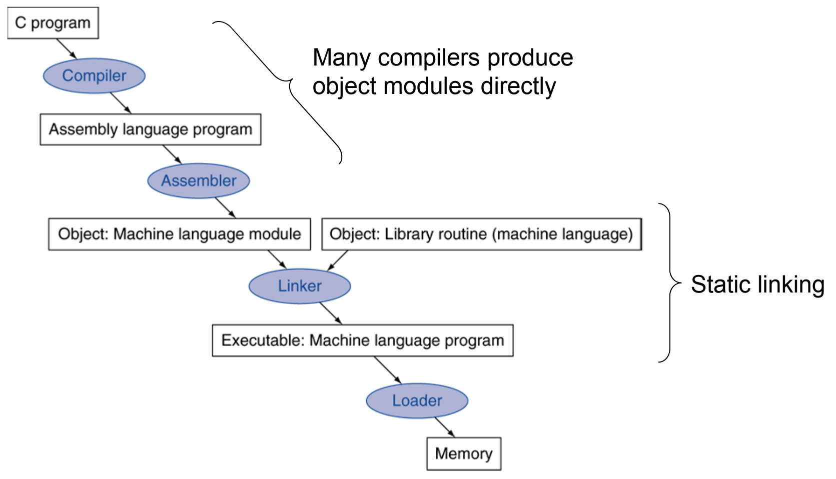
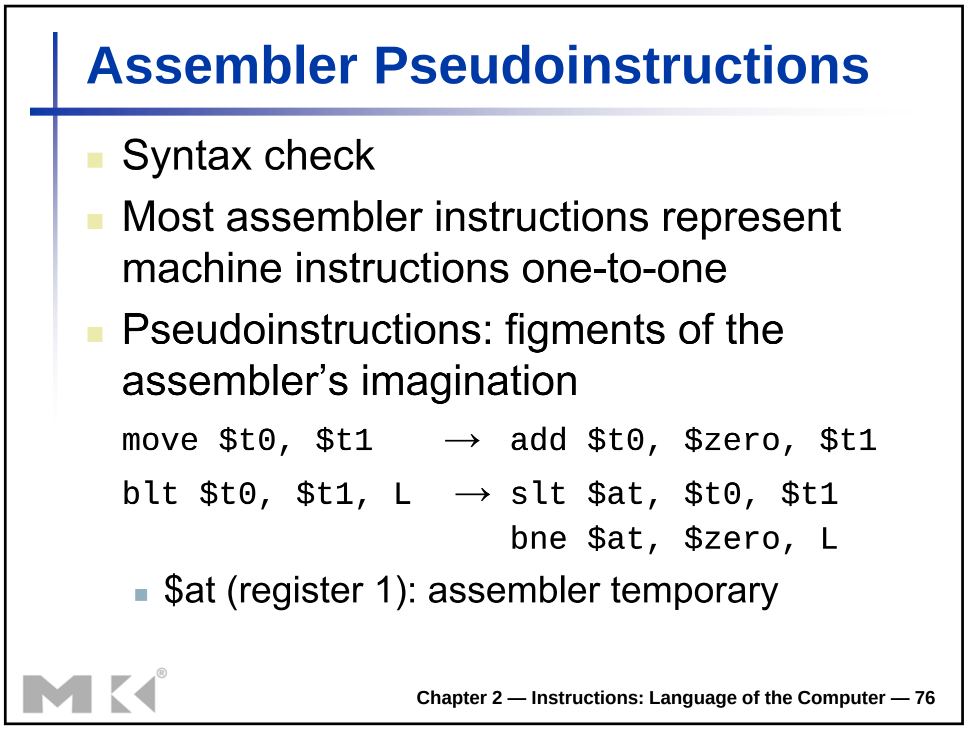
Translation and Startup


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

Compiler

- Transforms HLL C programs into assembly
- Why HLL
  - Fewer lines of code
  - Easier to understand and debug
- Today's optimizing compilers can produce assembly code nearly as good as an assembly language programming expert and often better for large programs



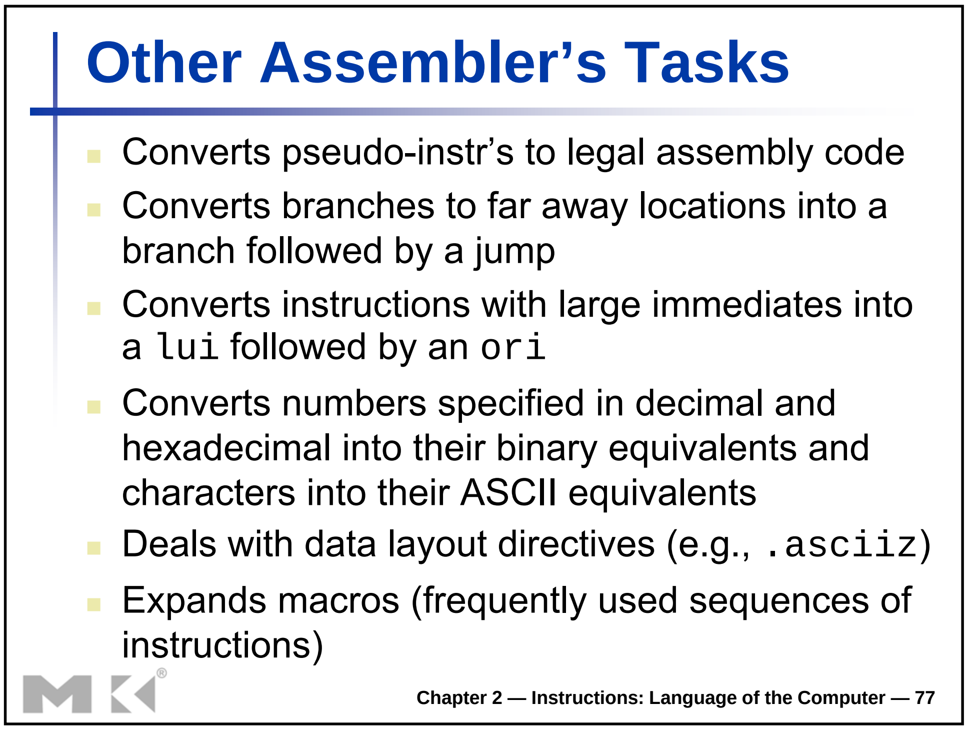
Assembler Pseudoinstructions

- Syntax check
- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: 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

Chapter 2 — Instructions: Language of the Computer — 76



Other Assembler's Tasks

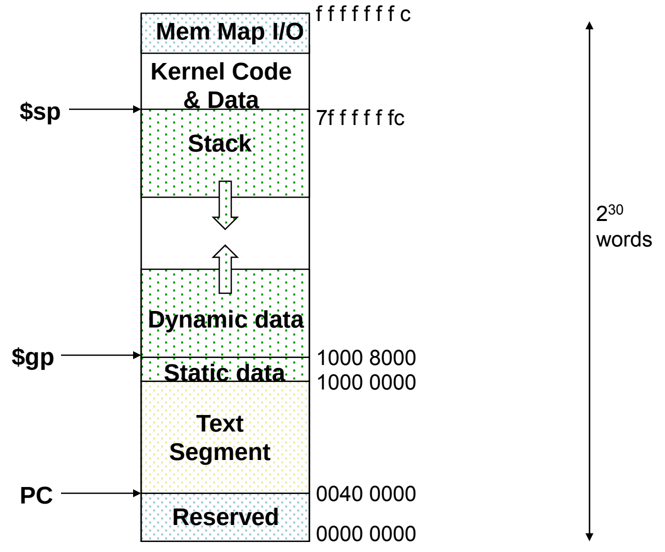
- Converts pseudo-instr's to legal assembly code
- Converts branches to far away locations into a branch followed by a jump
- Converts instructions with large immediates into a `lui` followed by an `ori`
- Converts numbers specified in decimal and hexadecimal into their binary equivalents and characters into their ASCII equivalents
- Deals with data layout directives (e.g., `.asciiz`)
- Expands macros (frequently used sequences of instructions)

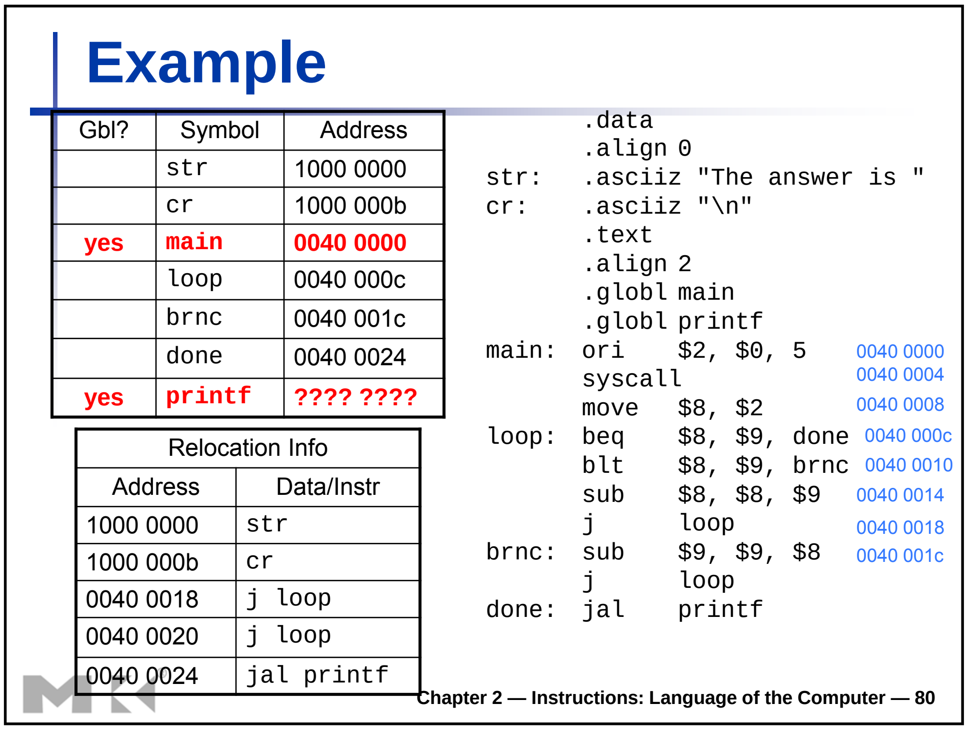
Chapter 2 — Instructions: Language of the Computer — 77

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
    - On MIPS, `j`, and `jal`, also `lw   $t1, 100($zero)`
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

MIPS (spim) memory Allocation





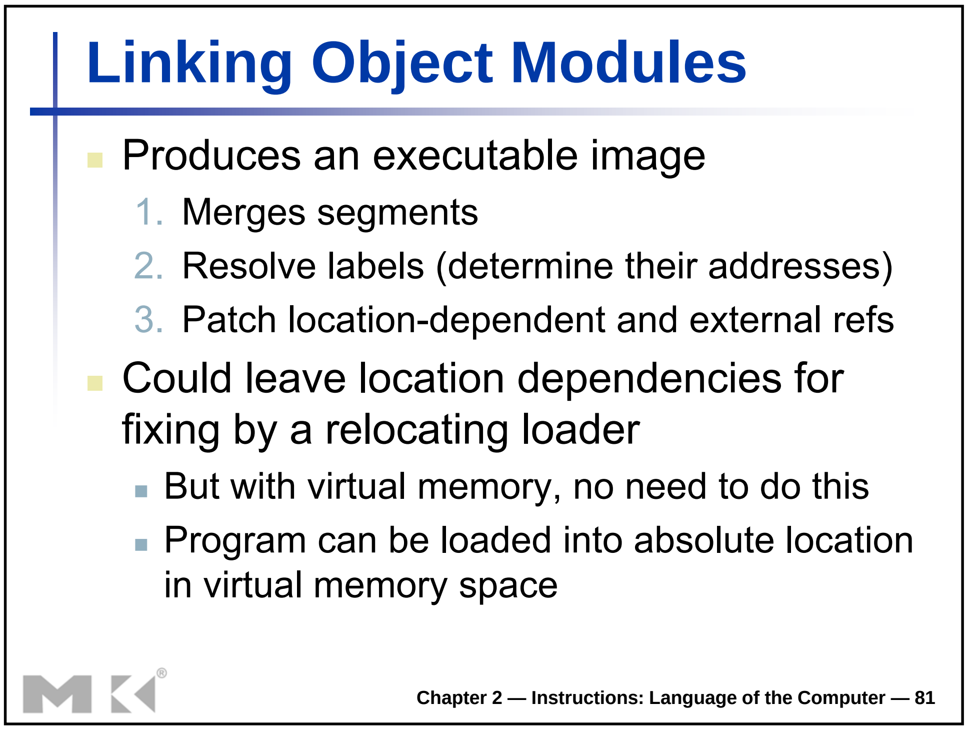
Example

Gbl?	Symbol	Address
	str	1000 0000
	cr	1000 000b
yes	main	0040 0000
	loop	0040 000c
	brnc	0040 001c
	done	0040 0024
yes	printf	???? ????

Relocation Info	
Address	Data/Instr
1000 0000	str
1000 000b	cr
0040 0018	j loop
0040 0020	j loop
0040 0024	jal printf

```
       .data
       .align 0
str:   .asciiz "The answer is "
cr:    .asciiz "\n"
       .text
       .align 2
       .globl main
       .globl printf
main:  ori     $2, $0, 5        0040 0000
       syscall                  0040 0004
       move    $8, $2           0040 0008
loop:  beq     $8, $9, done     0040 000c
       blt     $8, $9, brnc     0040 0010
       sub     $8, $8, $9       0040 0014
       j       loop             0040 0018
brnc:  sub     $9, $9, $8       0040 001c
       j       loop
done:  jal     printf
```

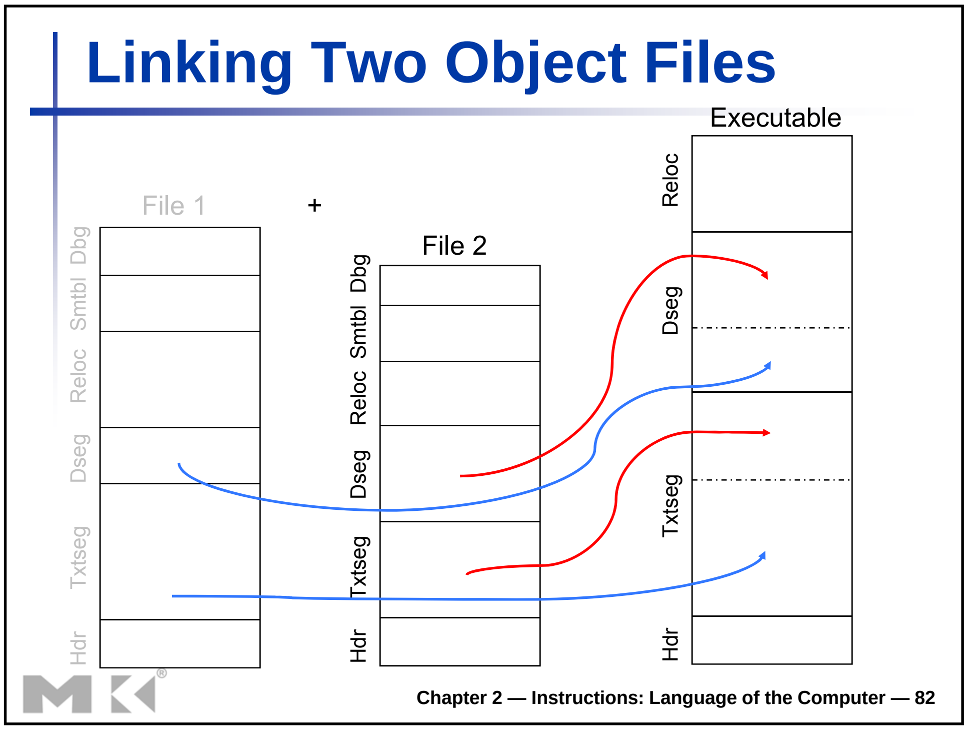
Chapter 2 — Instructions: Language of the Computer — 80



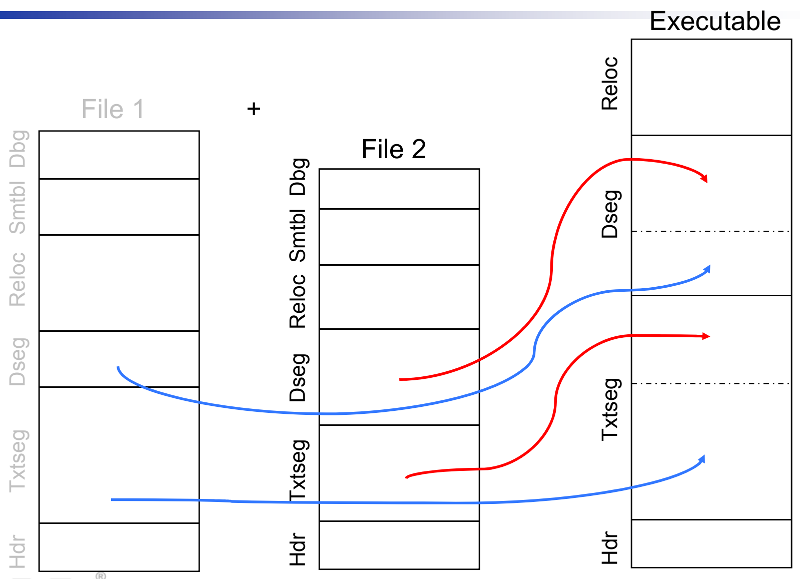
Linking Object Modules

- 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

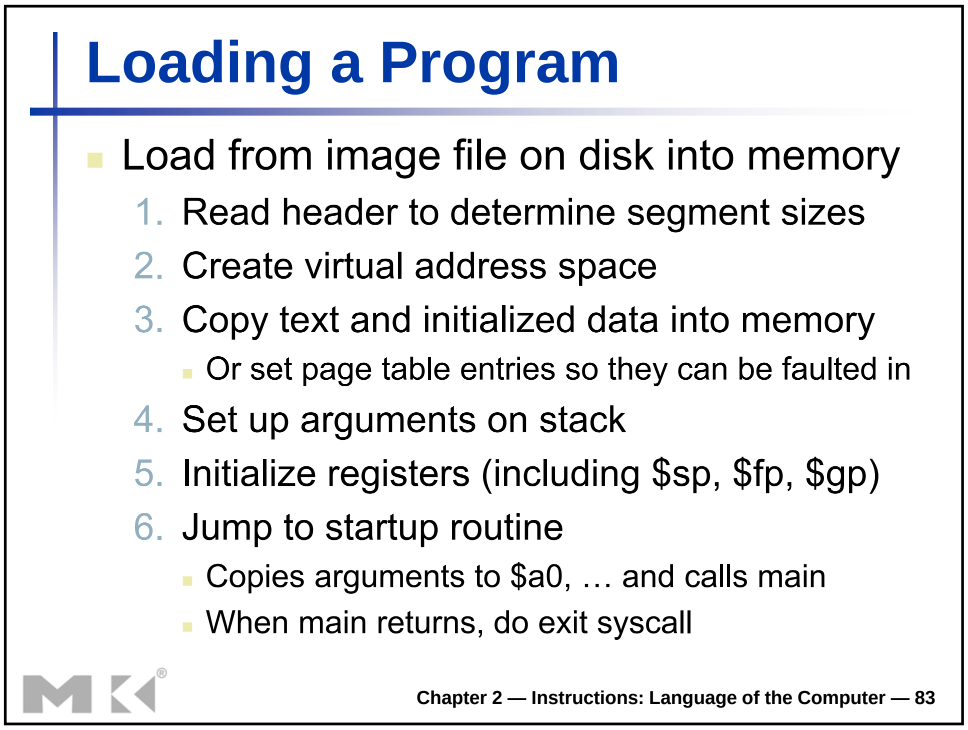
Chapter 2 — Instructions: Language of the Computer — 81



Linking Two Object Files



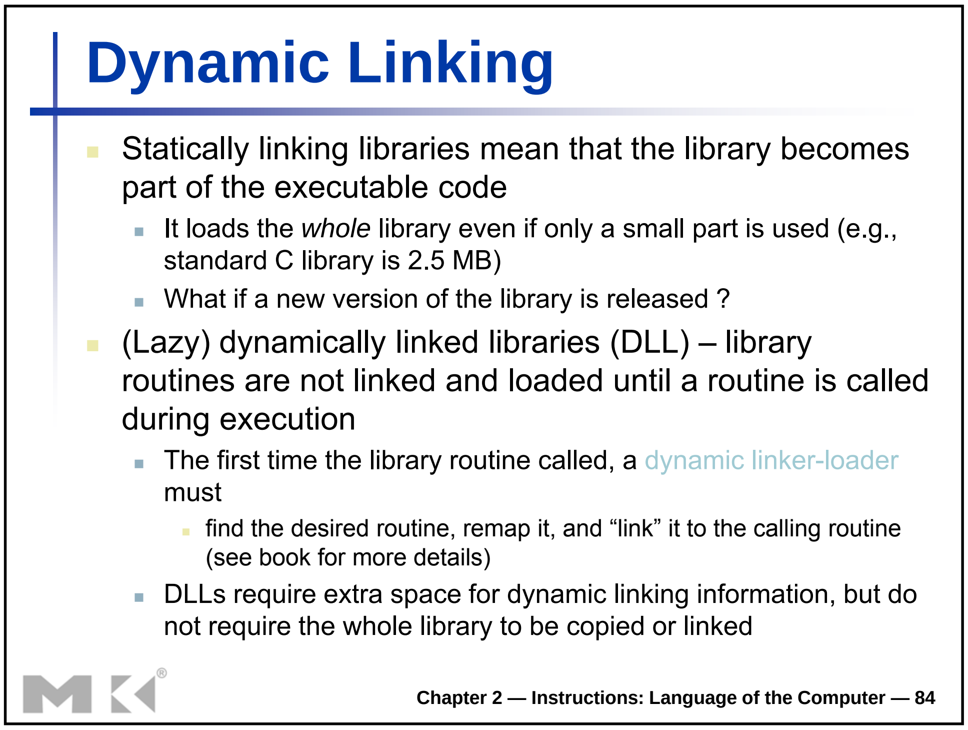
Chapter 2 — Instructions: Language of the Computer — 82



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

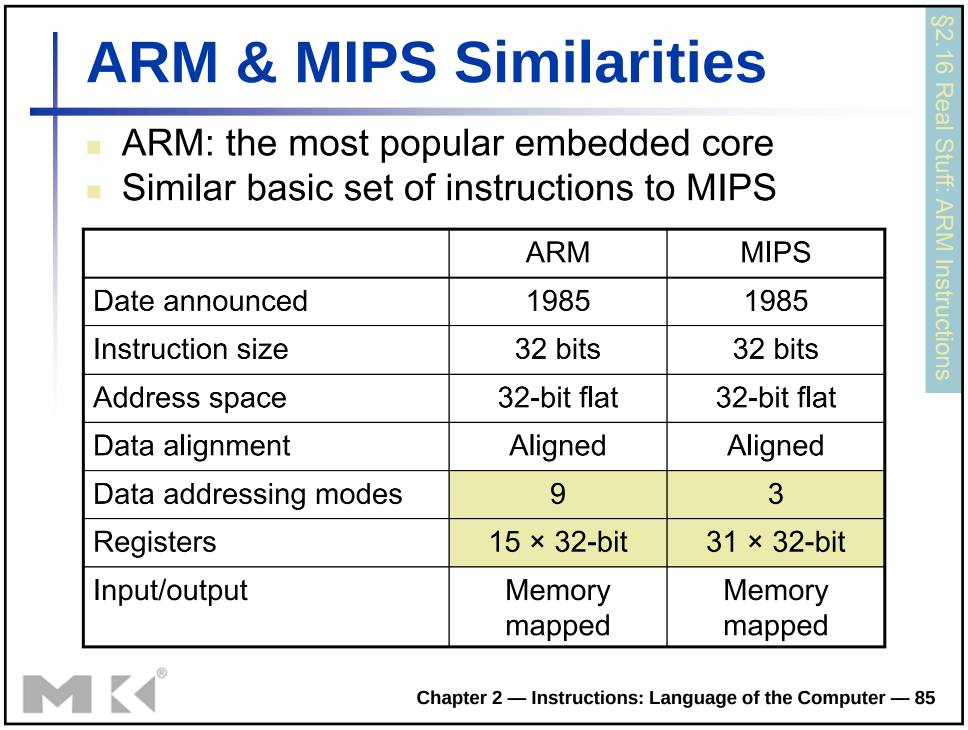
Chapter 2 — Instructions: Language of the Computer — 83



Dynamic Linking

- Statically linking libraries mean that the library becomes part of the executable code
  - It loads the *whole* library even if only a small part is used (e.g., standard C library is 2.5 MB)
  - What if a new version of the library is released ?
- (Lazy) dynamically linked libraries (DLL) – library routines are not linked and loaded until a routine is called during execution
  - The first time the library routine called, a dynamic linker-loader must
    - find the desired routine, remap it, and "link" it to the calling routine (see book for more details)
  - DLLs require extra space for dynamic linking information, but do not require the whole library to be copied or linked

Chapter 2 — Instructions: Language of the Computer — 84



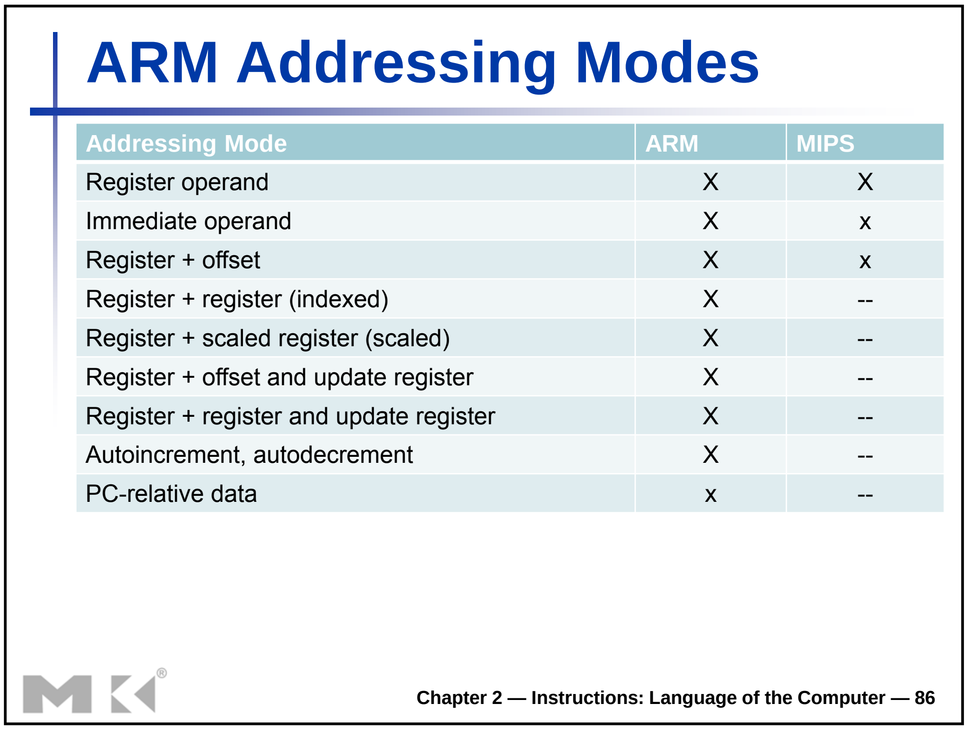
ARM & MIPS Similarities

§2.16 Real Stuff: ARM Instructions

- 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

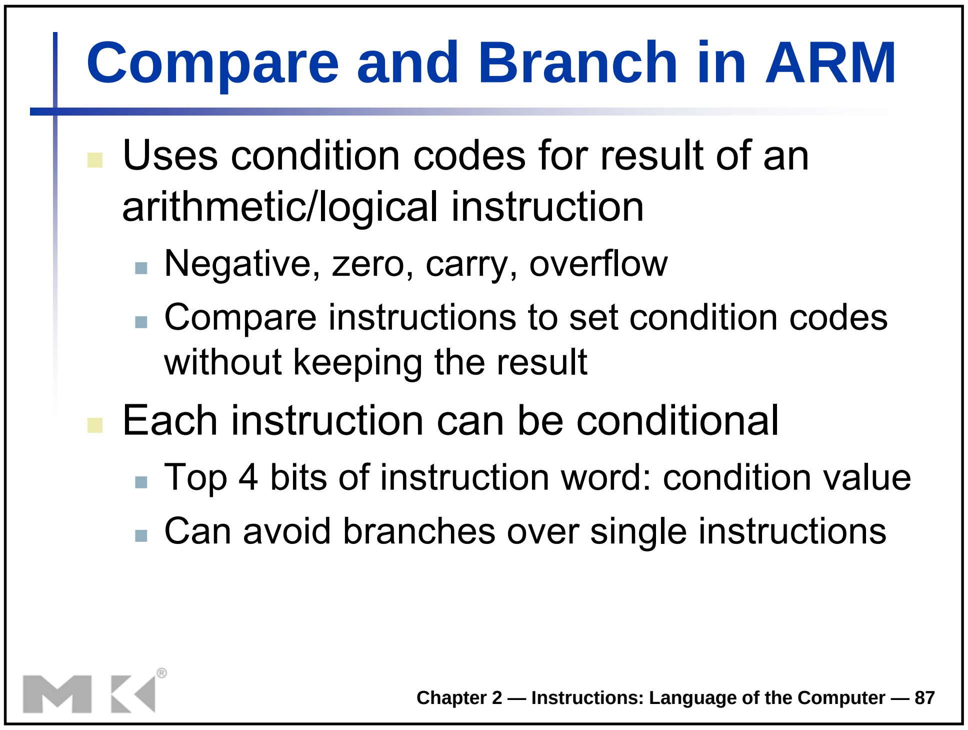
Chapter 2 — Instructions: Language of the Computer — 85



ARM Addressing Modes

Addressing Mode	ARM	MIPS
Register operand	X	X
Immediate operand	X	x
Register + offset	X	x
Register + register (indexed)	X	--
Register + scaled register (scaled)	X	--
Register + offset and update register	X	--
Register + register and update register	X	--
Autoincrement, autodecrement	X	--
PC-relative data	x	--

Chapter 2 — Instructions: Language of the Computer — 86



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

Chapter 2 — Instructions: Language of the Computer — 87

Conditional Execution

Unconditional

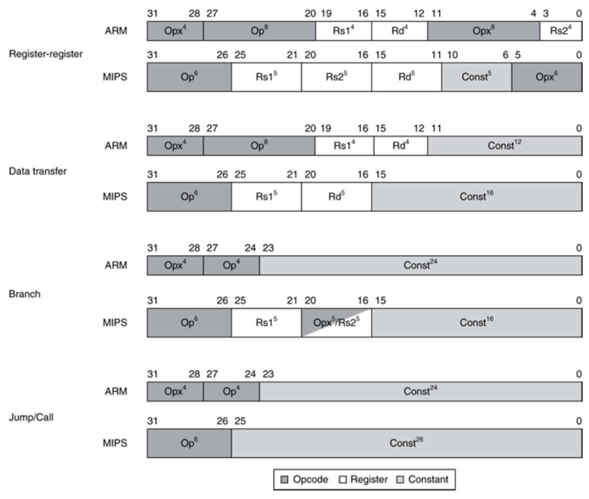
```
gcd     CMP     r0, r1
        BEQ      end
        BLT     less
        SUBS     r0, r0, r1 ;
        B       gcd
 less
        SUBS    r1, r1, r0 ;
        B       gcd
end
```

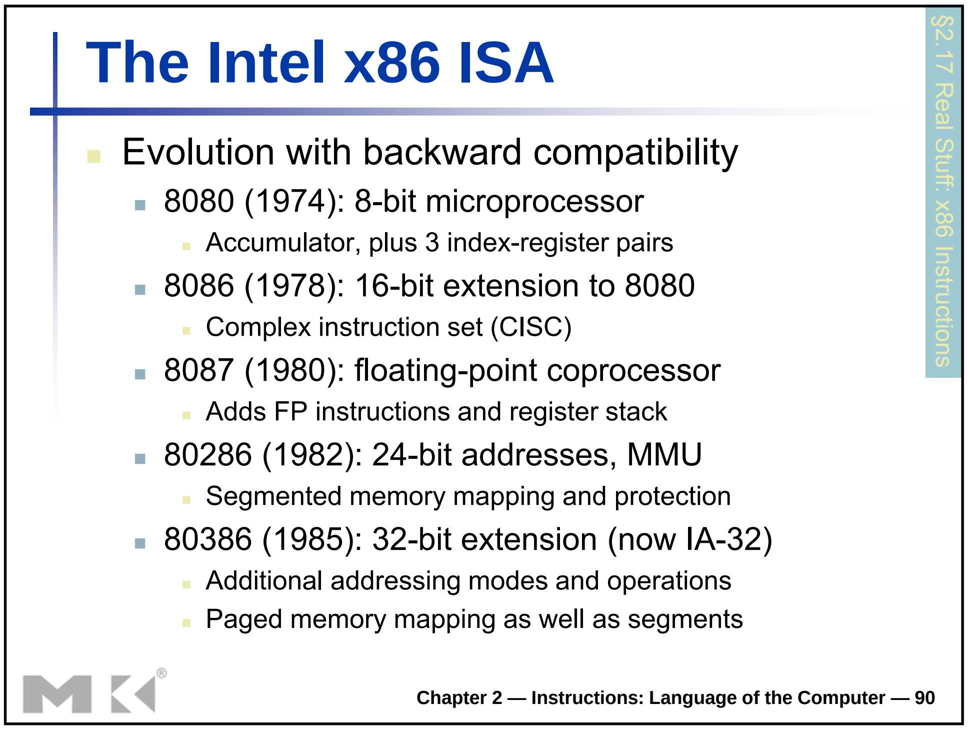
Conditional

```
gcd
        CMP             r0, r1
        SUBGT           r0, r0, r1
        SUBLE           r1, r1, r0
        BNE             gcd
```

```
int gcd(int a, int b)
{
while (a != b) {
    if (a > b) a = a -
b;
    else b = b - a;
    }
 return a;
 }
```

Instruction Encoding



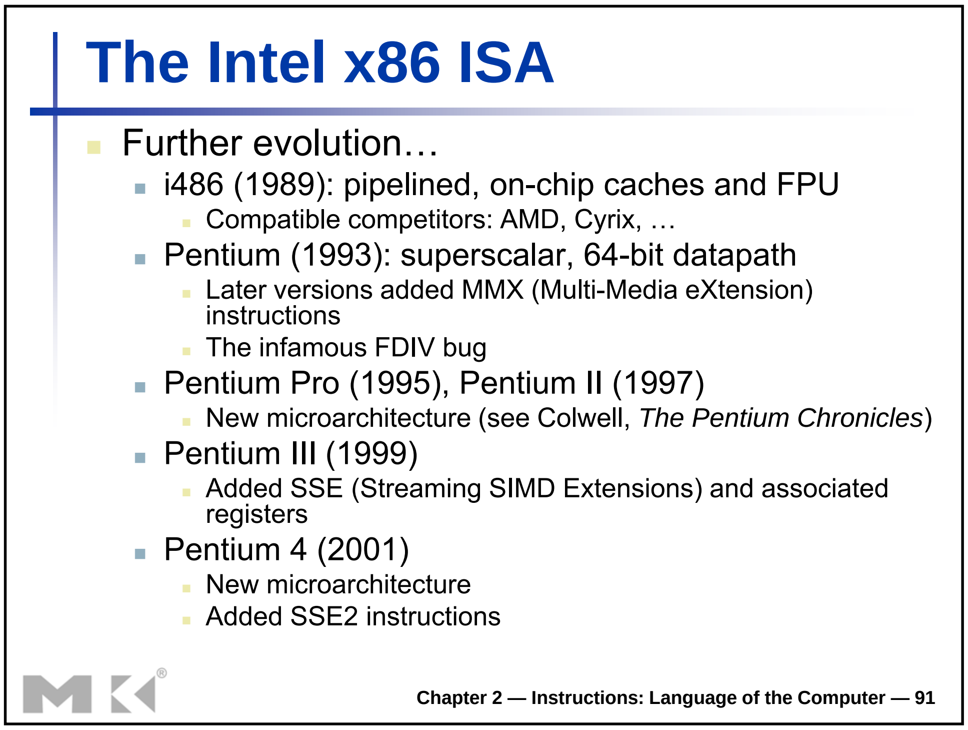


The Intel x86 ISA

§2.17 Real Stuff: x86 Instructions

- 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

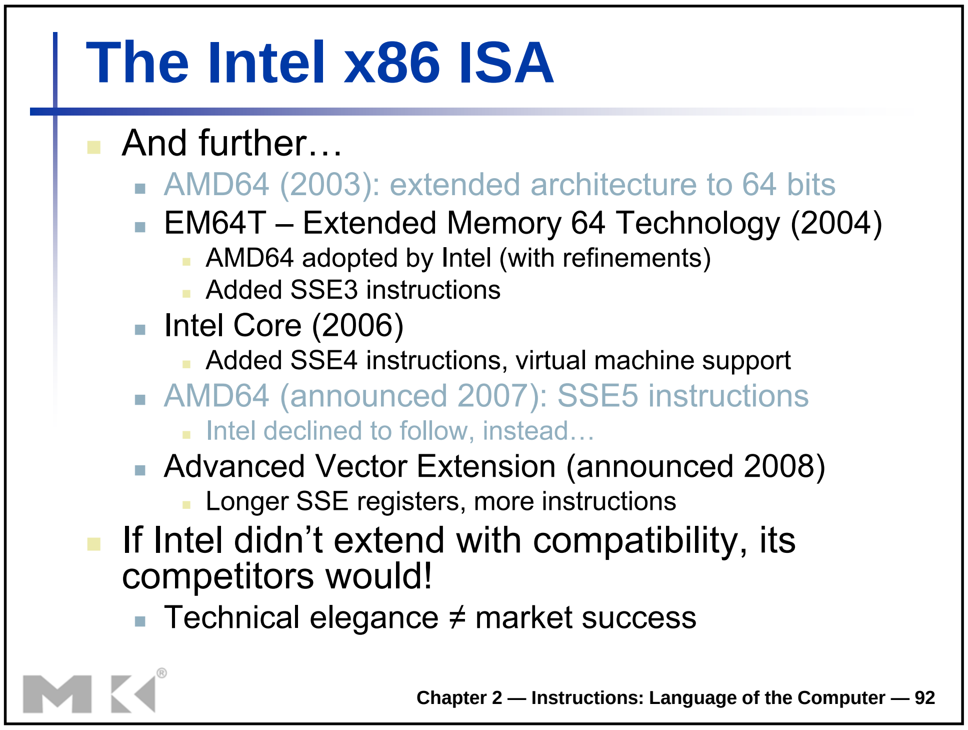
Chapter 2 — Instructions: Language of the Computer — 90



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

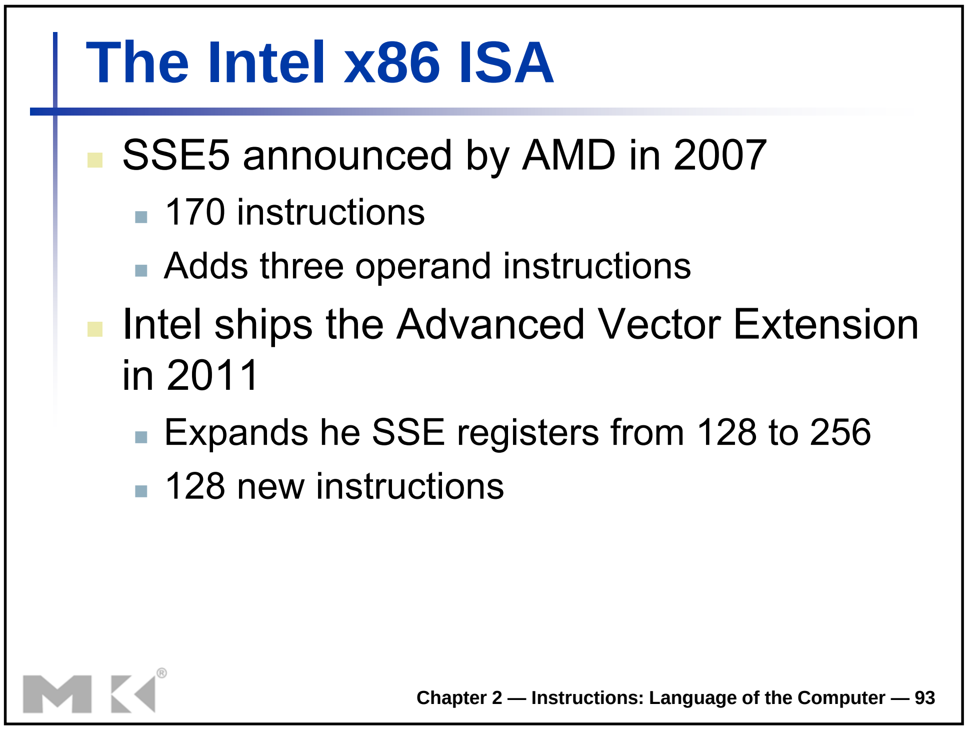
Chapter 2 — Instructions: Language of the Computer — 91



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

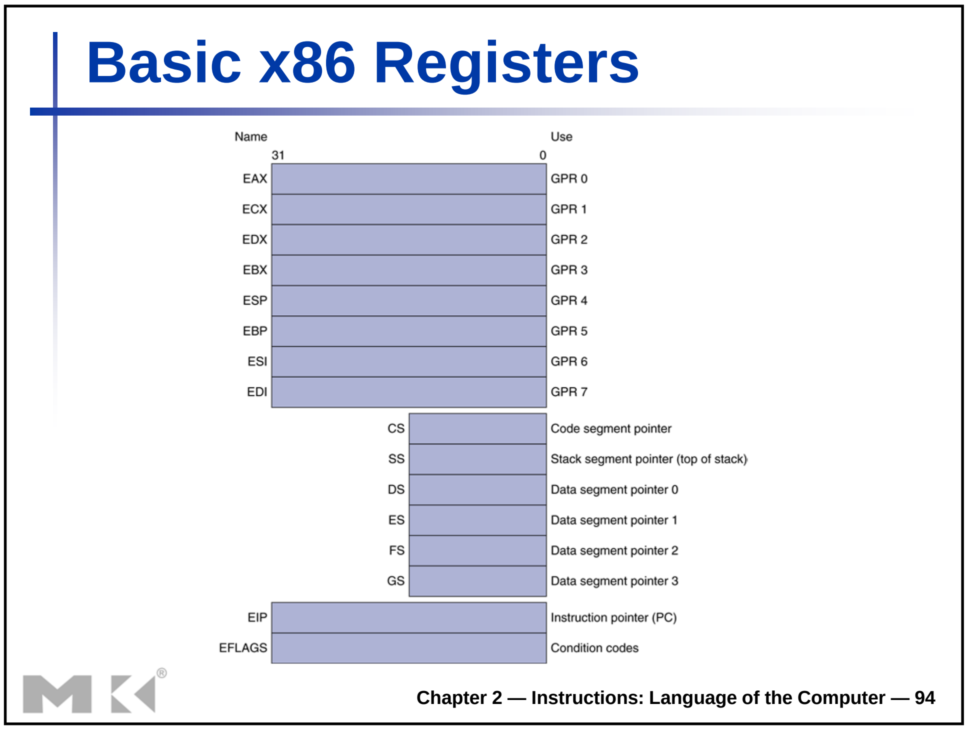
Chapter 2 — Instructions: Language of the Computer — 92



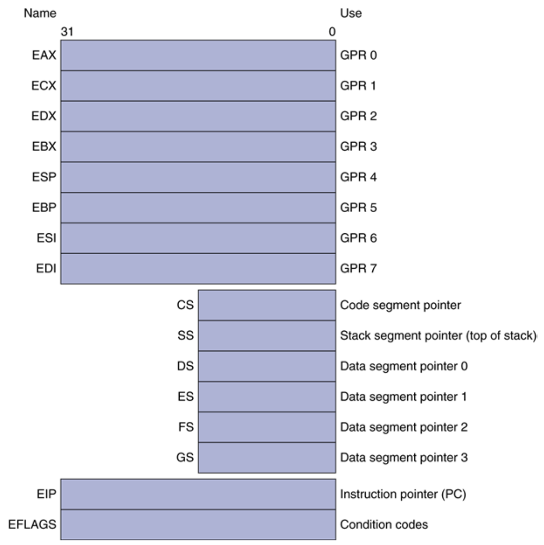
The Intel x86 ISA

- SSE5 announced by AMD in 2007
  - 170 instructions
  - Adds three operand instructions
- Intel ships the Advanced Vector Extension in 2011
  - Expands he SSE registers from 128 to 256
  - 128 new instructions

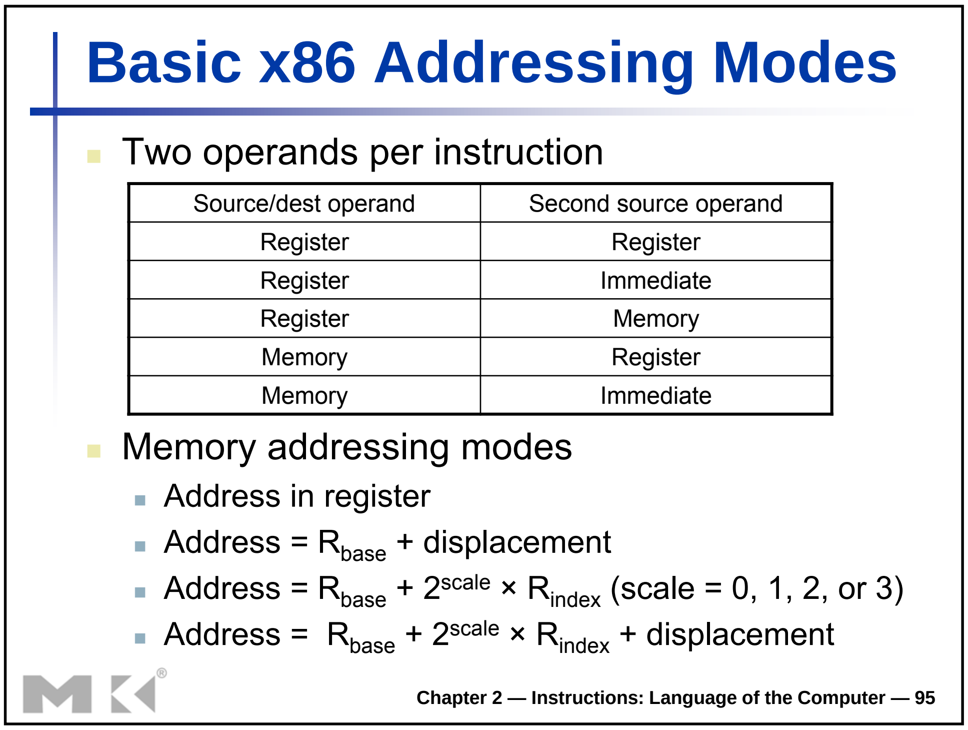
Chapter 2 — Instructions: Language of the Computer — 93



Basic x86 Registers



Chapter 2 — Instructions: Language of the Computer — 94



Basic x86 Addressing Modes

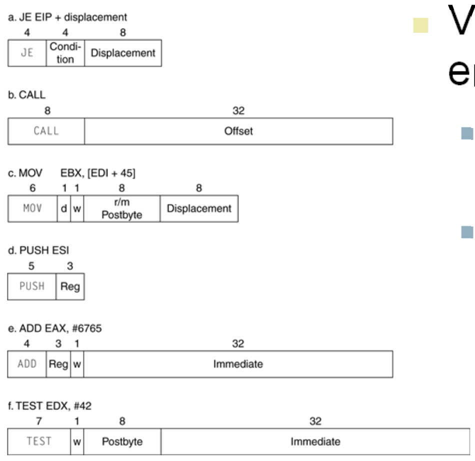
- Two operands per instruction

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

Chapter 2 — Instructions: Language of the Computer — 95

x86 Instruction Encoding



- 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