C declaration	Intel data type	Assembly code suffix	x86-64 size (bytes)	IA32 Size
`char`	Byte	`b`	1	1
`short`	Word	`w`	2	2
`int`	Double word	`l`	4	4
`long int`	Quad word	`q`	8	4
`long long int`	Quad word	`q`	8	8
`char *`	Quad word	`q`	8	4
`float`	Single precision	`s`	4	4
`double`	Double precision	`d`	8	8
`long double`	Extended precision	`t`	10/16	10/12

Figure 3.34 **Sizes of standard data types with x86-64.** These are compared to the sizes for IA32. Both long integers and pointers require 8 bytes, as compared to 4 for IA32.

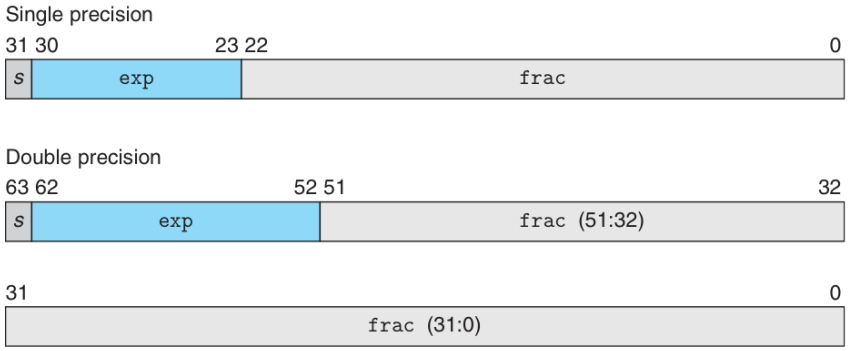

Figure 2.31 **Standard floating-point formats.** Floating-point numbers are represented by three fields. For the two most common formats, these are packed in 32-bit (single precision) or 64-bit (double precision) words.

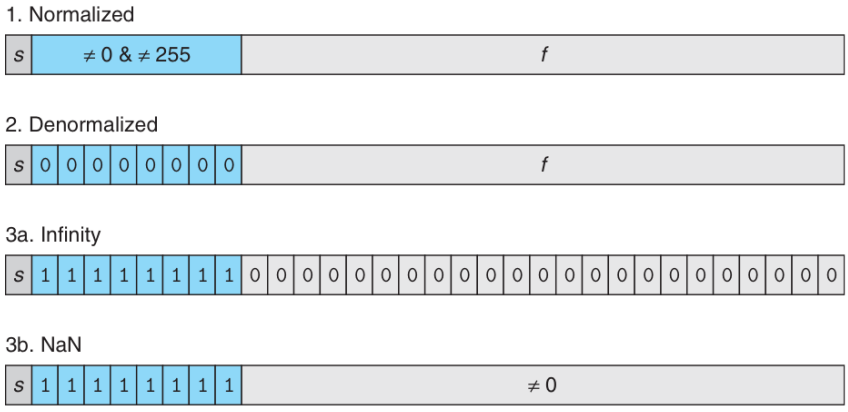

Figure 2.32 **Categories of single-precision, floating-point values.** The value of the exponent determines whether the number is (1) normalized, (2) denormalized, or a (3) special value.

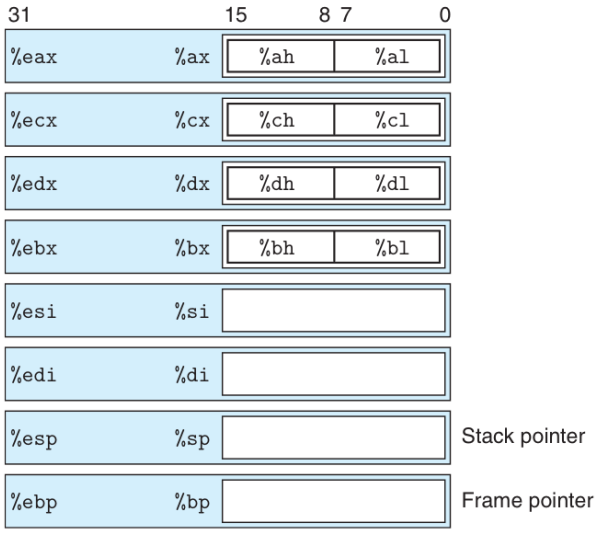

Figure 3.2 **IA32 integer registers.** All eight registers can be accessed as either 16 bits (word) or 32 bits (double word). The 2 low-order bytes of the first four registers can be accessed independently.

Instruction		Based on	Description
CMP	S_2, S_1	$S_1 - S_2$	Compare
`cmpb`		Compare byte	
`cmpw`		Compare word	
`cmpl`		Compare double word	
TEST	S_2, S_1	S_1 & S_2	Test
`testb`		Test byte	
`testw`		Test word	
`testl`		Test double word	

Figure 3.10 **Comparison and test instructions.** These instructions set the condition codes without updating any other registers.

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	E_a	$R[E_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(E_a)	$M[R[E_a]]$	Indirect
Memory	$Imm(E_b)$	$M[Imm + R[E_b]]$	Base + displacement
Memory	(E_b, E_i)	$M[R[E_b] + R[E_i]]$	Indexed
Memory	$Imm(E_b, E_i)$	$M[Imm + R[E_b] + R[E_i]]$	Indexed
Memory	$(, E_i, s)$	$M[R[E_i] \cdot s]$	Scaled indexed
Memory	$Imm(, E_i, s)$	$M[Imm + R[E_i] \cdot s]$	Scaled indexed
Memory	(E_b, E_i, s)	$M[R[E_b] + R[E_i] \cdot s]$	Scaled indexed
Memory	$Imm(E_b, E_i, s)$	$M[Imm + R[E_b] + R[E_i] \cdot s]$	Scaled indexed

Figure 3.3 **Operand forms.** Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

Instruction		Effect	Description
`leal`	S, D	$D \leftarrow \&S$	Load effective address
INC	D	$D \leftarrow D + 1$	Increment
DEC	D	$D \leftarrow D - 1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement
ADD	S, D	$D \leftarrow D + S$	Add
SUB	S, D	$D \leftarrow D - S$	Subtract
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \hat{} S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \,\&\, S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	Logical right shift

Figure 3.7 **Integer arithmetic operations.** The load effective address (`leal`) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use the notation $>>_A$ and $>>_L$ to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.

Instruction		Effect	Description
`imulq`	S	R[%rdx]:R[%rax] $\leftarrow S \times$ R[%rax]	Signed full multiply
`mulq`	S	R[%rdx]:R[%rax] $\leftarrow S \times$ R[%rax]	Unsigned full multiply
`cltq`		R[%rax] $\leftarrow$ SignExtend(R[%eax])	Convert %eax to quad word
`cqto`		R[%rdx]:R[%rax] $\leftarrow$ SignExtend(R[%rax])	Convert to oct word
`idivq`	S	R[%rdx] $\leftarrow$ R[%rdx]:R[%rax] mod S; R[%rax] $\leftarrow$ R[%rdx]:R[%rax] $\div S$	Signed divide
`divq`	S	R[%rdx] $\leftarrow$ R[%rdx]:R[%rax] mod S; R[%rax] $\leftarrow$ R[%rdx]:R[%rax] $\div S$	Unsigned divide

Figure 3.37 **Special arithmetic operations.** These operations support full 64-bit multiplication and division, for both signed and unsigned numbers. The pair of registers %rdx and %rax are viewed as forming a single 128-bit oct word.

Instruction		Synonym	Effect	Set condition
`sete`	*D*	`setz`	*D* ← `ZF`	Equal / zero
`setne`	*D*	`setnz`	*D* ← `~ZF`	Not equal / not zero
`sets`	*D*		*D* ← `SF`	Negative
`setns`	*D*		*D* ← `~SF`	Nonnegative
`setg`	*D*	`setnle`	*D* ← `~(SF ^ OF) & ~ZF`	Greater (signed >)
`setge`	*D*	`setnl`	*D* ← `~(SF ^ OF)`	Greater or equal (signed >=)
`setl`	*D*	`setnge`	*D* ← `SF ^ OF`	Less (signed <)
`setle`	*D*	`setng`	*D* ← `(SF ^ OF) | ZF`	Less or equal (signed <=)
`seta`	*D*	`setnbe`	*D* ← `~CF & ~ZF`	Above (unsigned >)
`setae`	*D*	`setnb`	*D* ← `~CF`	Above or equal (unsigned >=)
`setb`	*D*	`setnae`	*D* ← `CF`	Below (unsigned <)
`setbe`	*D*	`setna`	*D* ← `CF | ZF`	Below or equal (unsigned <=)

Figure 3.11 **The SET instructions.** Each instruction sets a single byte to 0 or 1 based on some combination of the condition codes. Some instructions have "synonyms," i.e., alternate names for the same machine instruction.

Instruction		Effect	Description
MOV	*S*, *D*	*D* ← *S*	Move
`movb`		Move byte	
`movw`		Move word	
`movl`		Move double word	
MOVS	*S*, *D*	*D* ← SignExtend(*S*)	Move with sign extension
`movsbw`		Move sign-extended byte to word	
`movsbl`		Move sign-extended byte to double word	
`movswl`		Move sign-extended word to double word	
MOVZ	*S*, *D*	*D* ← ZeroExtend(*S*)	Move with zero extension
`movzbw`		Move zero-extended byte to word	
`movzbl`		Move zero-extended byte to double word	
`movzwl`		Move zero-extended word to double word	
`pushl`	*S*	R[%esp] ← R[%esp] − 4; M[R[%esp]] ← *S*	Push double word
`popl`	*D*	*D* ← M[R[%esp]]; R[%esp] ← R[%esp] + 4	Pop double word

Figure 3.4 **Data movement instructions.**

Instruction		Synonym	Move condition	Description
`cmove`	*S*, *R*	`cmovz`	`ZF`	Equal / zero
`cmovne`	*S*, *R*	`cmovnz`	`~ZF`	Not equal / not zero
`cmovs`	*S*, *R*		`SF`	Negative
`cmovns`	*S*, *R*		`~SF`	Nonnegative
`cmovg`	*S*, *R*	`cmovnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`cmovge`	*S*, *R*	`cmovnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`cmovl`	*S*, *R*	`cmovnge`	`SF ^ OF`	Less (signed <)
`cmovle`	*S*, *R*	`cmovng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`cmova`	*S*, *R*	`cmovnbe`	`~CF & ~ZF`	Above (unsigned >)
`cmovae`	*S*, *R*	`cmovnb`	`~CF`	Above or equal (Unsigned >=)
`cmovb`	*S*, *R*	`cmovnae`	`CF`	Below (unsigned <)
`cmovbe`	*S*, *R*	`cmovna`	`CF | ZF`	below or equal (unsigned <=)

Figure 3.17 **The conditional move instructions.** These instructions copy the source value *S* to its destination *R* when the move condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	**Operand*		1	Indirect jump
`je`	*Label*	`jz`	`ZF`	Equal / zero
`jne`	*Label*	`jnz`	`~ZF`	Not equal / not zero
`js`	*Label*		`SF`	Negative
`jns`	*Label*		`~SF`	Nonnegative
`jg`	*Label*	`jnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`jge`	*Label*	`jnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`jl`	*Label*	`jnge`	`SF ^ OF`	Less (signed <)
`jle`	*Label*	`jng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`ja`	*Label*	`jnbe`	`~CF & ~ZF`	Above (unsigned >)
`jae`	*Label*	`jnb`	`~CF`	Above or equal (unsigned >=)
`jb`	*Label*	`jnae`	`CF`	Below (unsigned <)
`jbe`	*Label*	`jna`	`CF | ZF`	Below or equal (unsigned <=)

Figure 3.12 **The jump instructions.** These instructions jump to a labeled destination when the jump condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.