Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111
Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Figure 2.2 Hexadecimal notation.
Each hex digit encodes one of 16 values.

C declaration		Bytes	
Signed	**Unsigned**	**32-bit**	**64-bit**
`[signed] char`	`unsigned char`	1	1
`short`	`unsigned short`	2	2
`int`	`unsigned`	4	4
`long`	`unsigned long`	4	8
`int32_t`	`uint32_t`	4	4
`int64_t`	`uint64_t`	8	8
`char *`		4	8
`float`		4	4
`double`		8	8

Figure 2.3 Typical sizes (in bytes) of basic C data types.
The number of bytes allocated varies with how the program is compiled. This chart shows the values typical of 32-bit and 64-bit programs.

C declaration	Intel data type	Assembly-code suffix	Size (bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long`	Quad word	`q`	8
`char *`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8

Figure 3.1 Sizes of C data types in x86-64.
With a 64-bit machine, pointers are 8 bytes long.

Figure 3.2 Integer registers.

The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

Type	Form	Operand value	Name
Immediate	`$`*Imm*	*Imm*	Immediate
Register	r_a	R[r_a]	Register
Memory	*Imm*	M[*Imm*]	Absolute
Memory	(r_a)	M[R[r_a]]	Indirect
Memory	*Imm* (r_b)	M[*Imm* + R[r_b]]	Base + displacement
Memory	(r_b,r_i)	M[R[r_b] + R[r_i]]	Indexed
Memory	*Imm* (r_b,r_i)	M[*Imm* + R[r_b] + R[r_i]]	Indexed
Memory	(,r_i,s)	M[R[r_i] · s]	Scaled indexed
Memory	*Imm* (,r_i,s)	M[*Imm* + R[r_i] · s]	Scaled indexed
Memory	(r_b,r_i,s)	M[R[r_b] + R[r_i] · s]	Scaled indexed
Memory	*Imm* (r_b,r_i,s)	M[*Imm* + R[r_b] + R[r_i] · 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
MOV	*S*, *D*	*D* ← *S*	Move
`movb`			Move byte
`movw`			Move word
`movl`			Move double word
`moivq`			Move quad word
`movabsq`	*I*, *R*	*R* ← *I*	Move absolute quad word

Figure 3.4 Simple data movement instructions.

Instruction	Effect	Description
`MOVZ` *S*,*R*	*R* ← 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
`movzbq`		Move zero-extended byte to quad word
`movzwq`		Move zero-extended word to quad word

Figure 3.5 Zero-extending data movement instructions.
These instructions have a register or memory location as the source and a register as the destination.

Instruction	Effect	Description
MOVS *S*,*R*	*R* ← 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
`movsbq`		Move sign-extended byte to quad word
`movswq`		Move sign-extended word to quad word
`movslq`		Move sign-extended double word to quad word
`cltq`	%rax ← SignExtend(%eax)	Sign-extend %eax to %rax

Figure 3.6 Sign-extending data movement instructions.
The MOVS instructions have a register or memory location as the source and a register as the destination. The `cltq` instruction is specific to registers `%eax` and `%rax`.

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

Figure 3.10 Integer arithmetic operations.
The load effective address (leaq) 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`] ← *S* × R[`%rax`]	Signed full multiply
`mulq` *S*	R[`%rdx`]:R[`%rax`] ← *S* × R[`%rax`]	Unsigned full multiply
`cqto`	R[`%rdx`]:R[`%rax`] ← SignExtend(R[`%rax`])	Convert to oct word
`idivq` *S*	R[`%rdx`] ← R[`%rdx`]:R[`%rax`] mod *S*; R[`%rax`] ← R[`%rdx`]:R[`%rax`] ÷ *S*	Signed divide
`divq` *S*	R[`%rdx`] ← R[`%rdx`]:R[`%rax`] mod *S*; R[`%rax`] ← R[`%rdx`]:R[`%rax`] ÷ *S*	Unsigned divide

Figure 3.12 Special arithmetic operations.
These operations provide full 128-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		Based on	Description
CMP	S_1, S_2	$S_2 - S_1$	Compare
`cmpb`			Compare byte
`cmpw`			Compare word
`cmpl`			Compare double word
`cmpq`			Compare quad word
TEST	S_1, S_2	S_1 & S_2	Test
`testb`			Test byte
`testw`			Test word
`testl`			Test double word
`testq`			Test quad word

Figure 3.13 Comparison and test instructions.
These instructions set the condition codes without updating any other registers.

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 3.14 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," that is, 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 3.15 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.

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 3.18 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.

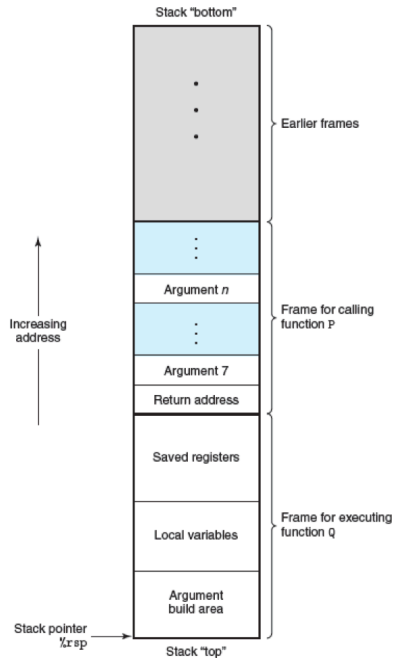

Figure 3.25 General stack frame structure.
The stack can be used for passing arguments, for storing return information, for saving registers, and for local storage. Portions may be omitted when not needed.