Instruction	Effect	Description
`pushq` S	R[`%rsp`] $\leftarrow$ R[`%rsp`] $-$ 8; M[R[`%rsp`]] $\leftarrow$ S	Push quad word
`popq` D	D $\leftarrow$ M[R[`%rsp`]]; R[`%rsp`] $\leftarrow$ R[`%rsp`] $+$ 8	Pop quad word

Figure 3.8 Push and pop instructions.

Instruction		Effect	Description
`leaq`	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	$D \leftarrow$ `~`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	$D \leftarrow D$ ^ 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 3.10 **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`]	$\leftarrow$	$S \times$ R[`%rax`]	Signed full multiply
`mulq`	S	R[`%rdx`]:R[`%rax`]	$\leftarrow$	$S \times$ R[`%rax`]	Unsigned full multiply
`cqto`		R[`%rdx`]:R[`%rax`]	$\leftarrow$	SignExtend(R[`%rax`])	Convert to oct word
`idivq`	S	R[`%rdx`]	$\leftarrow$	R[`%rdx`]:R[`%rax`] mod S;	Signed divide
		R[`%rax`]	$\leftarrow$	R[`%rdx`]:R[`%rax`] $\div$ S	
`divq`	S	R[`%rdx`]	$\leftarrow$	R[`%rdx`]:R[`%rax`] mod S;	Unsigned divide
		R[`%rax`]	$\leftarrow$	R[`%rdx`]:R[`%rax`] $\div$ S	

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.

`CF`: Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.

`ZF`: Zero flag. The most recent operation yielded zero.

`SF`: Sign flag. The most recent operation yielded a negative value.

`OF`: Overflow flag. The most recent operation caused a two's-complement overflow—either negative or positive.

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 <=)

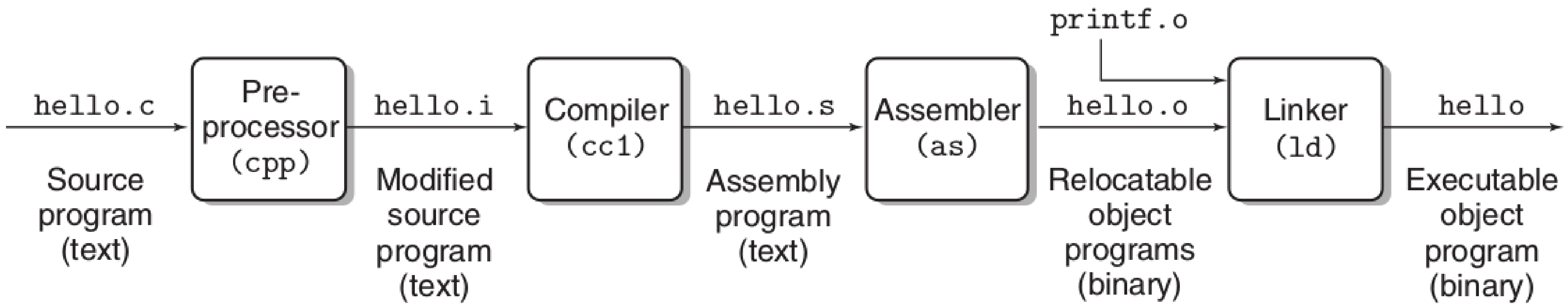
Figure 3.14 **The SET instructions.** Each instruction sets a single byte to 0 or 1 based on some combination of the condition codes. Some instructions have "synonyms," that is, alternate names for the same machine instruction.

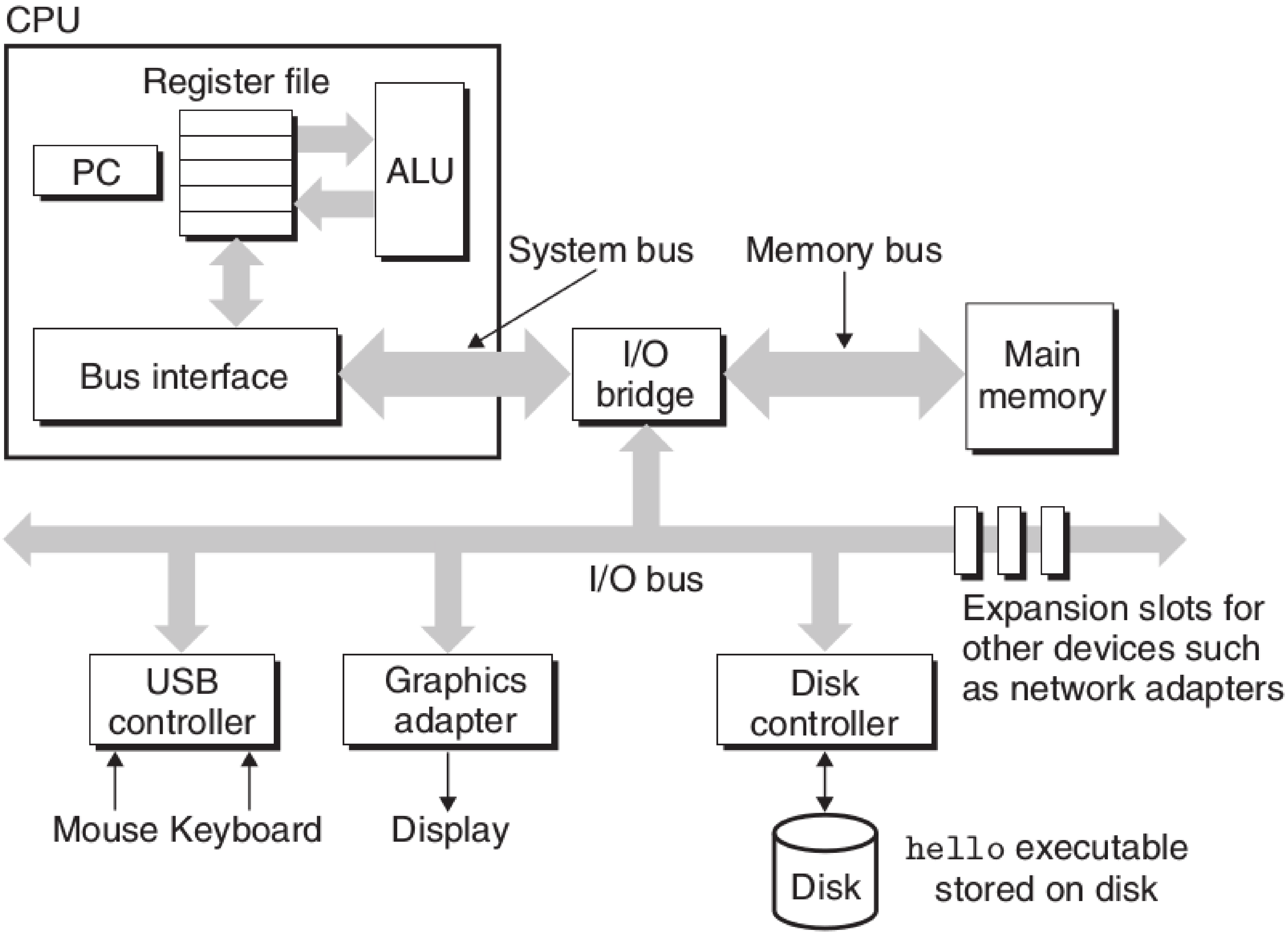
Instruction	Synonym	Jump condition	Description
jmp *Label*		1	Direct jump
jmp *Operand		1	Indirect jump
je *Label*	jz	ZF	Equal / zero
jne *Label*	jnz	~ZF	Not equal / not zero
js *Label*		SF	Negative
jns *Label*		~SF	Nonnegative
jg *Label*	jnle	~(SF ^ OF) & ~ZF	Greater (signed >)
jge *Label*	jnl	~(SF ^ 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.


printf.o
hello.c
Pre-processor (cpp)
hello.i
Compiler (cc1)
hello.s
Assembler (as)
hello.o
Linker (ld)
hello
Source program (text)
Modified source program (text)
Assembly program (text)
Relocatable object programs (binary)
Executable object program (binary)


CPU
Register file
PC
ALU
Bus interface
System bus
Memory bus
I/O bridge
Main memory
I/O bus
Expansion slots for other devices such as network adapters
USB controller
Graphics adapter
Disk controller
Mouse
Keyboard
Display
Disk
hello executable stored on disk

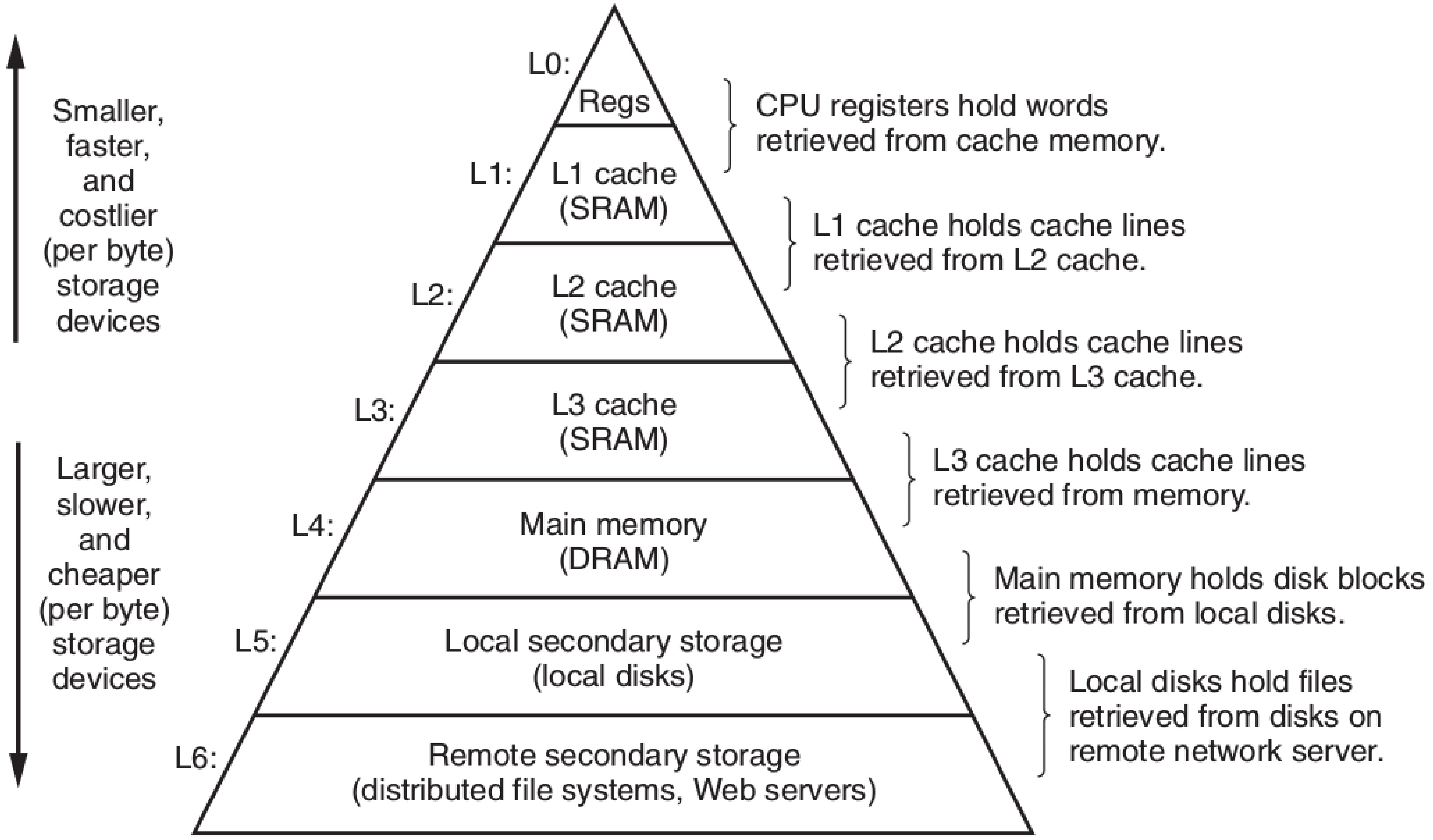

Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
L1:
L1 cache (SRAM)
L2:
L2 cache (SRAM)
L3:
L3 cache (SRAM)
L4:
Main memory (DRAM)
L5:
Local secondary storage (local disks)
L6:
Remote secondary storage (distributed file systems, Web servers)
CPU registers hold words retrieved from cache memory.
L1 cache holds cache lines retrieved from L2 cache.
L2 cache holds cache lines retrieved from L3 cache.
L3 cache holds cache lines retrieved from memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote network server.

Figure 1.9 **An example of a memory hierarchy.**

C data type	Minimum	Maximum
`[signed] char`	−128	127
`unsigned char`	0	255
`short`	−32,768	32,767
`unsigned short`	0	65,535
`int`	−2,147,483,648	2,147,483,647
`unsigned`	0	4,294,967,295
`long`	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
`unsigned long`	0	18,446,744,073,709,551,615
`int32_t`	−2,147,483,648	2,147,483,647
`uint32_t`	0	4,294,967,295
`int64_t`	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
`uint64_t`	0	18,446,744,073,709,551,615

Figure 2.10 **Typical ranges for C integral data types for 64-bit programs.**

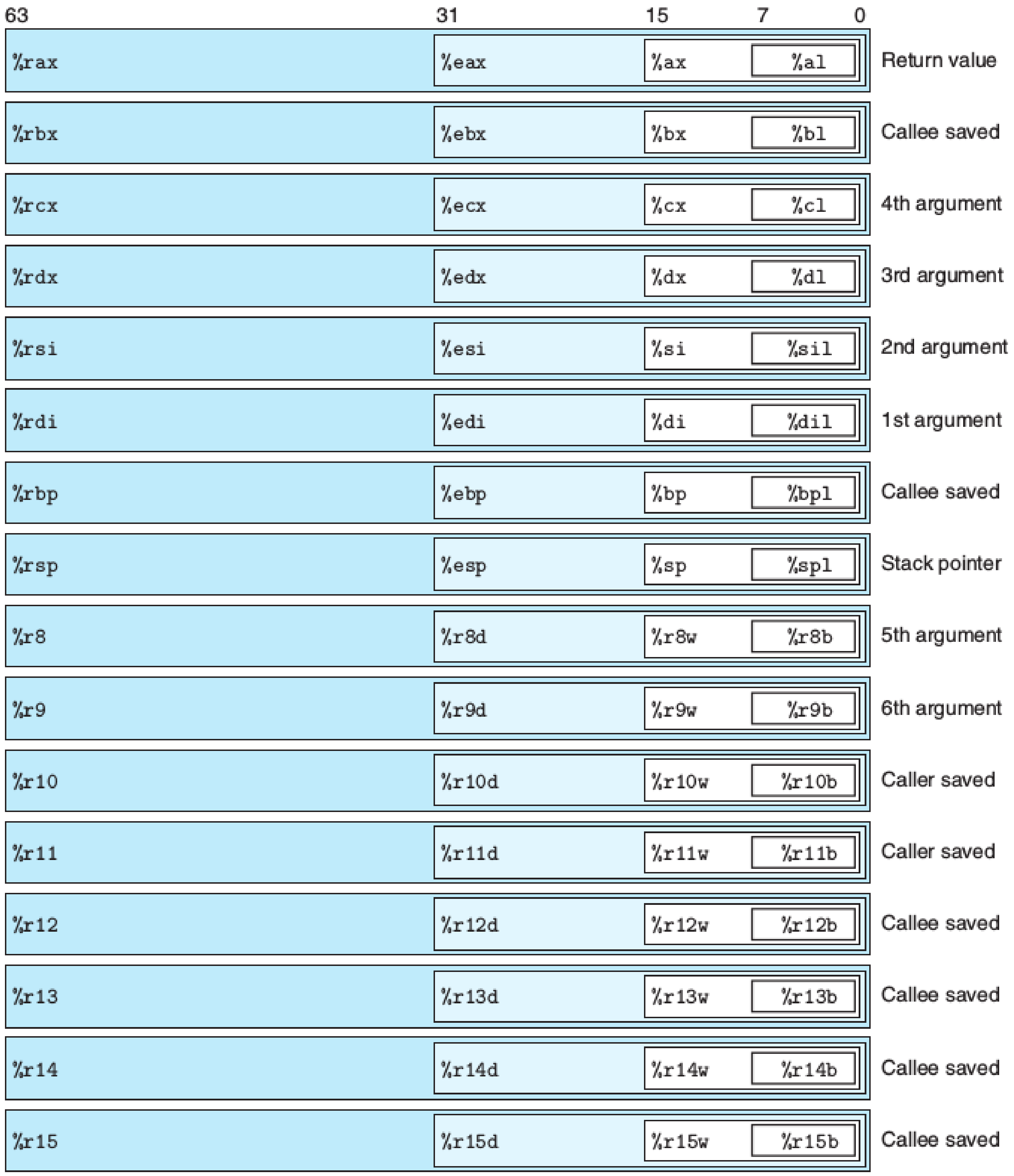


Figure 3.2 **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	$\texttt{r}_a$	$R[\texttt{r}_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	$(\texttt{r}_a)$	$M[R[\texttt{r}_a]]$	Indirect
Memory	$Imm(\texttt{r}_b)$	$M[Imm + R[\texttt{r}_b]]$	Base + displacement
Memory	$(\texttt{r}_b,\texttt{r}_i)$	$M[R[\texttt{r}_b] + R[\texttt{r}_i]]$	Indexed
Memory	$Imm(\texttt{r}_b,\texttt{r}_i)$	$M[Imm + R[\texttt{r}_b] + R[\texttt{r}_i]]$	Indexed
Memory	$(,\texttt{r}_i,s)$	$M[R[\texttt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(,\texttt{r}_i,s)$	$M[Imm + R[\texttt{r}_i] \cdot s]$	Scaled indexed
Memory	$(\texttt{r}_b,\texttt{r}_i,s)$	$M[R[\texttt{r}_b] + R[\texttt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(\texttt{r}_b,\texttt{r}_i,s)$	$M[Imm + R[\texttt{r}_b] + R[\texttt{r}_i] \cdot s]$	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
MOV	S, D	$D \leftarrow S$	Move
`movb`			Move byte
`movw`			Move word
`movl`			Move double word
`movq`			Move quad word
`movabsq`	I, R	$R \leftarrow I$	Move absolute quad word

Figure 3.4 Simple data movement instructions.

Instruction	Effect	Description
MOVZ S, R	$R \leftarrow \text{ZeroExtend}(S)$	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`.