not implemented with the same high-performance hardware as is the case for single- and double-precision arithmetic.

As the table of Figure 3.1 indicates, most assembly-code instructions generated by GCC have a single-character suffix denoting the size of the operand. For example, the data movement instruction has four variants: `movb` (move byte), `movw` (move word), `movl` (move double word), and `movq` (move quad word). The suffix '`l`' is used for double words, since 32-bit quantities are considered to be "long words." The assembly code uses the suffix '`l`' to denote a 4-byte integer as well as an 8-byte double-precision floating-point number. This causes no ambiguity, since floating-point code involves an entirely different set of instructions and registers.

3.4 Accessing Information

An x86-64 central processing unit (CPU) contains a set of 16 *general-purpose registers* storing 64-bit values. These registers are used to store integer data as well as pointers. Figure 3.2 diagrams the 16 registers. Their names all begin with `%r`, but otherwise follow multiple different naming conventions, owing to the historical evolution of the instruction set. The original 8086 had eight 16-bit registers, shown in Figure 3.2 as registers `%ax` through `%bp`. Each had a specific purpose, and hence they were given names that reflected how they were to be used. With the extension to IA32, these registers were expanded to 32-bit registers, labeled `%eax` through `%ebp`. In the extension to x86-64, the original eight registers were expanded to 64 bits, labeled `%rax` through `%rbp`. In addition, eight new registers were added, and these were given labels according to a new naming convention: `%r8` through `%r15`.

As the nested boxes in Figure 3.2 indicate, instructions can operate on data of different sizes stored in the low-order bytes of the 16 registers. Byte-level operations can access the least significant byte, 16-bit operations can access the least significant 2 bytes, 32-bit operations can access the least significant 4 bytes, and 64-bit operations can access entire registers.

In later sections, we will present a number of instructions for copying and generating 1-, 2-, 4-, and 8-byte values. When these instructions have registers as destinations, two conventions arise for what happens to the remaining bytes in the register for instructions that generate less than 8 bytes: Those that generate 1- or 2-byte quantities leave the remaining bytes unchanged. Those that generate 4-byte quantities set the upper 4 bytes of the register to zero. The latter convention was adopted as part of the expansion from IA32 to x86-64.

As the annotations along the right-hand side of Figure 3.2 indicate, different registers serve different roles in typical programs. Most unique among them is the stack pointer, `%rsp`, used to indicate the end position in the run-time stack. Some instructions specifically read and write this register. The other 15 registers have more flexibility in their uses. A small number of instructions make specific use of certain registers. More importantly, a set of standard programming conventions governs how the registers are to be used for managing the stack, passing function

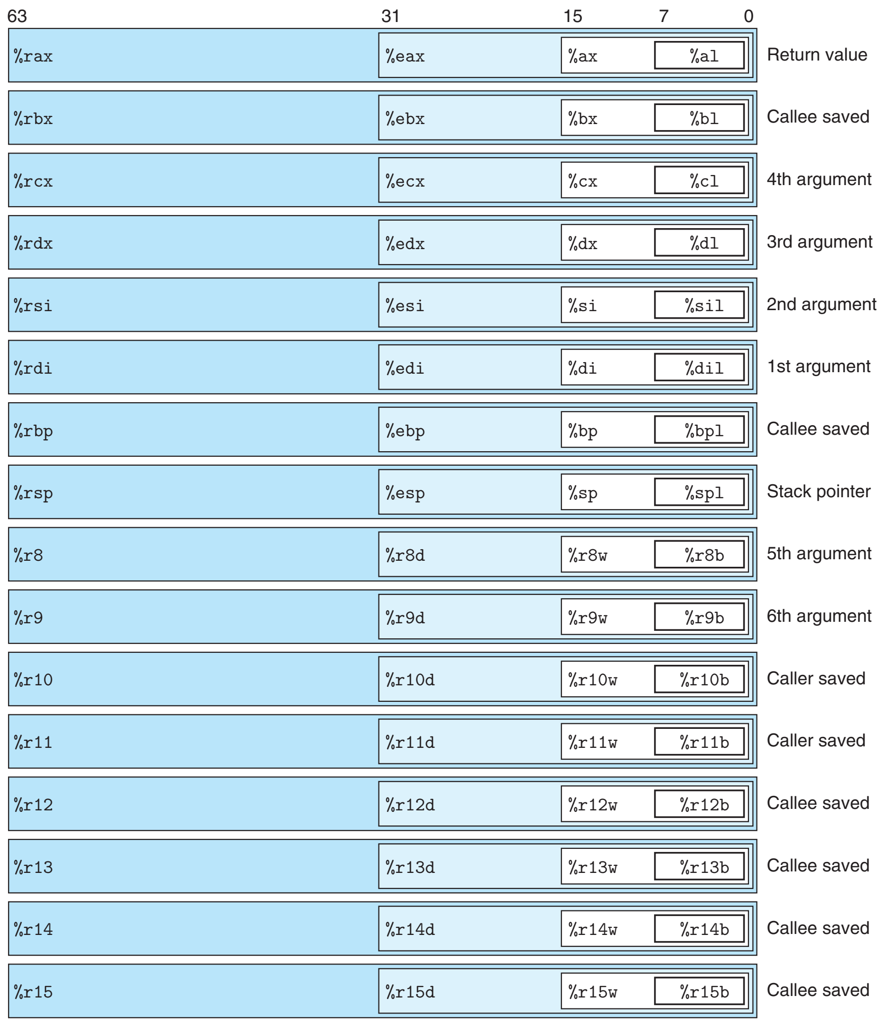
63
31
15
7
0
%rax %eax %ax %al Return value
%rbx %ebx %bx %bl Callee saved
%rcx %ecx %cx %cl 4th argument
%rdx %edx %dx %dl 3rd argument
%rsi %esi %si %sil 2nd argument
%rdi %edi %di %dil 1st argument
%rbp %ebp %bp %bpl Callee saved
%rsp %esp %sp %spl Stack pointer
%r8 %r8d %r8w %r8b 5th argument
%r9 %r9d %r9w %r9b 6th argument
%r10 %r10d %r10w %r10b Caller saved
%r11 %r11d %r11w %r11b Caller saved
%r12 %r12d %r12w %r12b Callee saved
%r13 %r13d %r13w %r13b Callee saved
%r14 %r14d %r14w %r14b Callee saved
%r15 %r15d %r15w %r15b Callee saved

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.

arguments, returning values from functions, and storing local and temporary data. We will cover these conventions in our presentation, especially in Section 3.7, where we describe the implementation of procedures.

3.4.1 Operand Specifiers

Most instructions have one or more *operands* specifying the source values to use in performing an operation and the destination location into which to place the

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	$\mathtt{r}_a$	$R[\mathtt{r}_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	$(\mathtt{r}_a)$	$M[R[\mathtt{r}_a]]$	Indirect
Memory	$Imm(\mathtt{r}_b)$	$M[Imm + R[\mathtt{r}_b]]$	Base + displacement
Memory	$(\mathtt{r}_b,\mathtt{r}_i)$	$M[R[\mathtt{r}_b] + R[\mathtt{r}_i]]$	Indexed
Memory	$Imm(\mathtt{r}_b,\mathtt{r}_i)$	$M[Imm + R[\mathtt{r}_b] + R[\mathtt{r}_i]]$	Indexed
Memory	$(,\mathtt{r}_i,s)$	$M[R[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(,\mathtt{r}_i,s)$	$M[Imm + R[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$(\mathtt{r}_b,\mathtt{r}_i,s)$	$M[R[\mathtt{r}_b] + R[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(\mathtt{r}_b,\mathtt{r}_i,s)$	$M[Imm + R[\mathtt{r}_b] + R[\mathtt{r}_i] \cdot s]$	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.

result. x86-64 supports a number of operand forms (see Figure 3.3). Source values can be given as constants or read from registers or memory. Results can be stored in either registers or memory. Thus, the different operand possibilities can be classified into three types. The first type, *immediate*, is for constant values. In ATT-format assembly code, these are written with a '`$`' followed by an integer using standard C notation—for example, `$-577` or `$0x1F`. Different instructions allow different ranges of immediate values; the assembler will automatically select the most compact way of encoding a value. The second type, *register*, denotes the contents of a register, one of the sixteen 8-, 4-, 2-, or 1-byte low-order portions of the registers for operands having 64, 32, 16, or 8 bits, respectively. In Figure 3.3, we use the notation $\mathtt{r}_a$ to denote an arbitrary register a and indicate its value with the reference $R[\mathtt{r}_a]$, viewing the set of registers as an array R indexed by register identifiers.

The third type of operand is a *memory* reference, in which we access some memory location according to a computed address, often called the *effective address*. Since we view the memory as a large array of bytes, we use the notation $M_b[Addr]$ to denote a reference to the b-byte value stored in memory starting at address $Addr$. To simplify things, we will generally drop the subscript b.

As Figure 3.3 shows, there are many different *addressing modes* allowing different forms of memory references. The most general form is shown at the bottom of the table with syntax $Imm(\mathtt{r}_b,\mathtt{r}_i,s)$. Such a reference has four components: an immediate offset Imm, a base register $\mathtt{r}_b$, an index register $\mathtt{r}_i$, and a scale factor s, where s must be 1, 2, 4, or 8. Both the base and index must be 64-bit registers. The effective address is computed as $Imm + R[\mathtt{r}_b] + R[\mathtt{r}_i] \cdot s$. This general form is often seen when referencing elements of arrays. The other forms are simply special cases of this general form where some of the components are omitted. As we will see, the more complex addressing modes are useful when referencing array and structure elements.

Practice Problem 3.1 (solution page 361)

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value	Register	Value
`0x100`	`0xFF`	`%rax`	`0x100`
`0x104`	`0xAB`	`%rcx`	`0x1`
`0x108`	`0x13`	`%rdx`	`0x3`
`0x10C`	`0x11`		

Fill in the following table showing the values for the indicated operands:

Operand	Value
`%rax`	
`0x104`	
`$0x108`	
`(%rax)`	
`4(%rax)`	
`9(%rax,%rdx)`	
`260(%rcx,%rdx)`	
`0xFC(,%rcx,4)`	
`(%rax,%rdx,4)`	

3.4.2 Data Movement Instructions

Among the most heavily used instructions are those that copy data from one location to another. The generality of the operand notation allows a simple data movement instruction to express a range of possibilities that in many machines would require a number of different instructions. We present a number of different data movement instructions, differing in their source and destination types, what conversions they perform, and other side effects they may have. In our presentation, we group the many different instructions into *instruction classes*, where the instructions in a class perform the same operation but with different operand sizes.

Figure 3.4 lists the simplest form of data movement instructions—MOV class. These instructions copy data from a source location to a destination location, without any transformation. The class consists of four instructions: `movb`, `movw`, `movl`, and `movq`. All four of these instructions have similar effects; they differ primarily in that they operate on data of different sizes: 1, 2, 4, and 8 bytes, respectively.

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.

The source operand designates a value that is immediate, stored in a register, or stored in memory. The destination operand designates a location that is either a register or a memory address. x86-64 imposes the restriction that a move instruction cannot have both operands refer to memory locations. Copying a value from one memory location to another requires two instructions—the first to load the source value into a register, and the second to write this register value to the destination. Referring to Figure 3.2, register operands for these instructions can be the labeled portions of any of the 16 registers, where the size of the register must match the size designated by the last character of the instruction (‘b’, ‘w’, ‘l’, or ‘q’). For most cases, the MOV instructions will only update the specific register bytes or memory locations indicated by the destination operand. The only exception is that when movl has a register as the destination, it will also set the high-order 4 bytes of the register to 0. This exception arises from the convention, adopted in x86-64, that any instruction that generates a 32-bit value for a register also sets the high-order portion of the register to 0.

The following MOV instruction examples show the five possible combinations of source and destination types. Recall that the source operand comes first and the destination second.

```
movl $0x4050,%eax         Immediate--Register, 4 bytes
movw %bp,%sp              Register--Register,  2 bytes
movb (%rdi,%rcx),%al      Memory--Register,    1 byte
movb $-17,(%esp)          Immediate--Memory,   1 byte
movq %rax,-12(%rbp)       Register--Memory,    8 bytes
```

A final instruction documented in Figure 3.4 is for dealing with 64-bit immediate data. The regular movq instruction can only have immediate source operands that can be represented as 32-bit two’s-complement numbers. This value is then sign extended to produce the 64-bit value for the destination. The movabsq instruction can have an arbitrary 64-bit immediate value as its source operand and can only have a register as a destination.

Figures 3.5 and 3.6 document two classes of data movement instructions for use when copying a smaller source value to a larger destination. All of these instructions copy data from a source, which can be either a register or stored

Aside Understanding how data movement changes a destination register

As described, there are two different conventions regarding whether and how data movement instructions modify the upper bytes of a destination register. This distinction is illustrated by the following code sequence:

```
1      movabsq $0x0011223344556677, %rax    %rax = 0011223344556677
2      movb    $-1, %al                     %rax = 00112233445566FF
3      movw    $-1, %ax                     %rax = 001122334455FFFF
4      movl    $-1, %eax                    %rax = 00000000FFFFFFFF
5      movq    $-1, %rax                    %rax = FFFFFFFFFFFFFFFF
```

In the following discussion, we use hexadecimal notation. In the example, the instruction on line 1 initializes register `%rax` to the pattern `0011223344556677`. The remaining instructions have immediate value −1 as their source values. Recall that the hexadecimal representation of −1 is of the form `FF···F`, where the number of `F`'s is twice the number of bytes in the representation. The `movb` instruction (line 2) therefore sets the low-order byte of `%rax` to `FF`, while the `movw` instruction (line 3) sets the low-order 2 bytes to `FFFF`, with the remaining bytes unchanged. The `movl` instruction (line 4) sets the low-order 4 bytes to `FFFFFFFF`, but it also sets the high-order 4 bytes to `00000000`. Finally, the `movq` instruction (line 5) sets the complete register to `FFFFFFFFFFFFFFFF`.

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.

in memory, to a register destination. Instructions in the MOVZ class fill out the remaining bytes of the destination with zeros, while those in the MOVS class fill them out by sign extension, replicating copies of the most significant bit of the source operand. Observe that each instruction name has size designators as its final two characters—the first specifying the source size, and the second specifying the destination size. As can be seen, there are three instructions in each of these classes, covering all cases of 1- and 2-byte source sizes and 2- and 4-byte destination sizes, considering only cases where the destination is larger than the source, of course.

Instruction	Effect	Description
MOVS S, R	$R \leftarrow$ 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` $\leftarrow$ 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`.

Note the absence of an explicit instruction to zero-extend a 4-byte source value to an 8-byte destination in Figure 3.5. Such an instruction would logically be named `movzlq`, but this instruction does not exist. Instead, this type of data movement can be implemented using a `movl` instruction having a register as the destination. This technique takes advantage of the property that an instruction generating a 4-byte value with a register as the destination will fill the upper 4 bytes with zeros. Otherwise, for 64-bit destinations, moving with sign extension is supported for all three source types, and moving with zero extension is supported for the two smaller source types.

Figure 3.6 also documents the `cltq` instruction. This instruction has no operands—it always uses register `%eax` as its source and `%rax` as the destination for the sign-extended result. It therefore has the exact same effect as the instruction `movslq %eax, %rax`, but it has a more compact encoding.

Practice Problem 3.2 (solution page 361)

For each of the following lines of assembly language, determine the appropriate instruction suffix based on the operands. (For example, `mov` can be rewritten as `movb`, `movw`, `movl`, or `movq`.)

```
mov__   %eax, (%rsp)
mov__   (%rax), %dx
mov__   $0xFF, %bl
mov__   (%rsp,%rdx,4), %dl
mov__   (%rdx), %rax
mov__   %dx, (%rax)
```

Aside Comparing byte movement instructions

The following example illustrates how different data movement instructions either do or do not change the high-order bytes of the destination. Observe that the three byte-movement instructions `movb`, `movsbq`, and `movzbq` differ from each other in subtle ways. Here is an example:

```
1    movabsq $0x0011223344556677, %rax    %rax = 0011223344556677
2    movb    $0xAA, %dl                   %dl  = AA
3    movb %dl,%al                         %rax = 00112233445566AA
4    movsbq %dl,%rax                      %rax = FFFFFFFFFFFFFFAA
5    movzbq %dl,%rax                      %rax = 00000000000000AA
```

In the following discussion, we use hexadecimal notation for all of the values. The first two lines of the code initialize registers `%rax` and `%dl` to `0011223344556677` and `AA`, respectively. The remaining instructions all copy the low-order byte of `%rdx` to the low-order byte of `%rax`. The `movb` instruction (line 3) does not change the other bytes. The `movsbq` instruction (line 4) sets the other 7 bytes to either all ones or all zeros depending on the high-order bit of the source byte. Since hexadecimal `A` represents binary value `1010`, sign extension causes the higher-order bytes to each be set to `FF`. The `movzbq` instruction (line 5) always sets the other 7 bytes to zero.

Practice Problem 3.3 (solution page 362)

Each of the following lines of code generates an error message when we invoke the assembler. Explain what is wrong with each line.

```
movb $0xF, (%ebx)
movl %rax, (%rsp)
movw (%rax),4(%rsp)
movb %al,%sl
movq %rax,$0x123
movl %eax,%rdx
movb %si, 8(%rbp)
```

3.4.3 Data Movement Example

As an example of code that uses data movement instructions, consider the data exchange routine shown in Figure 3.7, both as C code and as assembly code generated by GCC.

As Figure 3.7(b) shows, function `exchange` is implemented with just three instructions: two data movements (`movq`) plus an instruction to return back to the point from which the function was called (`ret`). We will cover the details of function call and return in Section 3.7. Until then, it suffices to say that arguments are passed to functions in registers. Our annotated assembly code documents these. A function returns a value by storing it in register `%rax`, or in one of the low-order portions of this register.

(a) C code

```
long exchange(long *xp, long y)
{
    long x = *xp;
    *xp = y;
    return x;
}
```

(b) Assembly code

```
     long exchange(long *xp, long y)
     xp in %rdi, y in %rsi
1    exchange:
2      movq    (%rdi), %rax     Get x at xp. Set as return value.
3      movq    %rsi, (%rdi)     Store y at xp.
4      ret                      Return.
```

Figure 3.7 C and assembly code for exchange routine. Registers `%rdi` and `%rsi` hold parameters `xp` and `y`, respectively.

When the procedure begins execution, procedure parameters `xp` and `y` are stored in registers `%rdi` and `%rsi`, respectively. Instruction 2 then reads `x` from memory and stores the value in register `%rax`, a direct implementation of the operation `x = *xp` in the C program. Later, register `%rax` will be used to return a value from the function, and so the return value will be `x`. Instruction 3 writes `y` to the memory location designated by `xp` in register `%rdi`, a direct implementation of the operation `*xp = y`. This example illustrates how the MOV instructions can be used to read from memory to a register (line 2), and to write from a register to memory (line 3).

Two features about this assembly code are worth noting. First, we see that what we call “pointers” in C are simply addresses. Dereferencing a pointer involves copying that pointer into a register, and then using this register in a memory reference. Second, local variables such as `x` are often kept in registers rather than stored in memory locations. Register access is much faster than memory access.

Practice Problem 3.4 (solution page 362)

Assume variables `sp` and `dp` are declared with types

```
src_t  *sp;
dest_t *dp;
```

where `src_t` and `dest_t` are data types declared with `typedef`. We wish to use the appropriate pair of data movement instructions to implement the operation

```
*dp = (dest_t) *sp;
```

New to C? Some examples of pointers

Function `exchange` (Figure 3.7(a)) provides a good illustration of the use of pointers in C. Argument `xp` is a pointer to a long integer, while `y` is a long integer itself. The statement

```
long x = *xp;
```

indicates that we should read the value stored in the location designated by `xp` and store it as a local variable named `x`. This read operation is known as pointer *dereferencing*. The C operator '`*`' performs pointer dereferencing.

The statement

```
*xp = y;
```

does the reverse—it writes the value of parameter `y` at the location designated by `xp`. This is also a form of pointer dereferencing (and hence the operator `*`), but it indicates a write operation since it is on the left-hand side of the assignment.

The following is an example of `exchange` in action:

```
long a = 4;
long b = exchange(&a, 3);
printf("a = %ld, b = %ld\verb@\@n", a, b);
```

This code will print

```
a = 3, b = 4
```

The C operator '`&`' (called the "address of" operator) *creates* a pointer, in this case to the location holding local variable `a`. Function `exchange` overwrites the value stored in `a` with 3 but returns the previous value, 4, as the function value. Observe how by passing a pointer to `exchange`, it could modify data held at some remote location.

Assume that the values of `sp` and `dp` are stored in registers `%rdi` and `%rsi`, respectively. For each entry in the table, show the two instructions that implement the specified data movement. The first instruction in the sequence should read from memory, do the appropriate conversion, and set the appropriate portion of register `%rax`. The second instruction should then write the appropriate portion of `%rax` to memory. In both cases, the portions may be `%rax`, `%eax`, `%ax`, or `%al`, and they may differ from one another.

Recall that when performing a cast that involves both a size change and a change of "signedness" in C, the operation should change the size first (Section 2.2.6).

`src_t`	`dest_t`	Instruction
`long`	`long`	`movq (%rdi), %rax`
		`movq %rax, (%rsi)`
`char`	`int`	________
		________

`char`	`unsigned`	__________
		__________
`unsigned char`	`long`	__________
		__________
`int`	`char`	__________
		__________
`unsigned`	`unsigned char`	__________
		__________
`char`	`short`	__________
		__________

Practice Problem 3.5 (solution page 363)

You are given the following information. A function with prototype

```
void decode1(long *xp, long *yp, long *zp);
```

is compiled into assembly code, yielding the following:

```
  void decode1(long *xp, long *yp, long *zp)
  xp in %rdi, yp in %rsi, zp in %rdx
decode1:
  movq    (%rdi), %r8
  movq    (%rsi), %rcx
  movq    (%rdx), %rax
  movq    %r8, (%rsi)
  movq    %rcx, (%rdx)
  movq    %rax, (%rdi)
  ret
```

Parameters `xp`, `yp`, and `zp` are stored in registers `%rdi`, `%rsi`, and `%rdx`, respectively.

Write C code for `decode1` that will have an effect equivalent to the assembly code shown.

3.4.4 Pushing and Popping Stack Data

The final two data movement operations are used to push data onto and pop data from the program stack, as documented in Figure 3.8. As we will see, the stack plays a vital role in the handling of procedure calls. By way of background, a stack is a data structure where values can be added or deleted, but only according to a "last-in, first-out" discipline. We add data to a stack via a *push* operation and remove it via a *pop* operation, with the property that the value popped will always be the value that was most recently pushed and is still on the stack. A stack can be implemented as an array, where we always insert and remove elements from one

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.

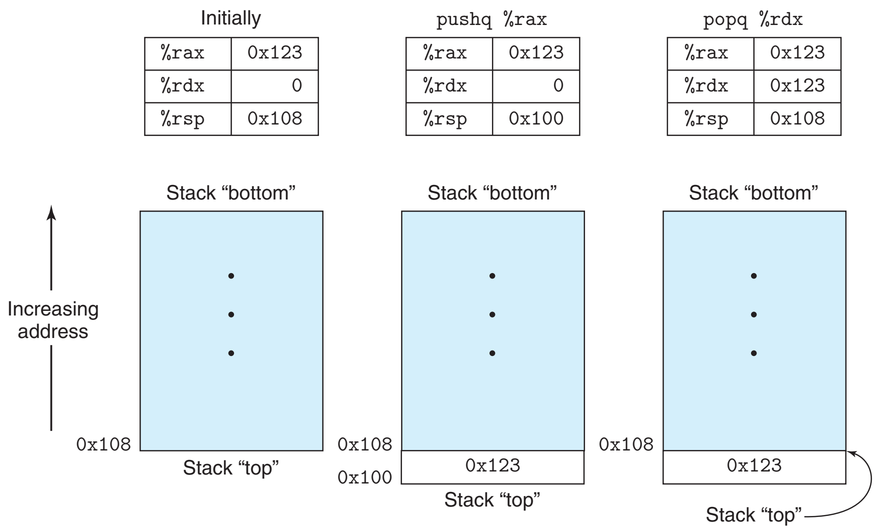

Figure 3.9 Illustration of stack operation. By convention, we draw stacks upside down, so that the "top" of the stack is shown at the bottom. With x86-64, stacks grow toward lower addresses, so pushing involves decrementing the stack pointer (register `%rsp`) and storing to memory, while popping involves reading from memory and incrementing the stack pointer.

end of the array. This end is called the *top* of the stack. With x86-64, the program stack is stored in some region of memory. As illustrated in Figure 3.9, the stack grows downward such that the top element of the stack has the lowest address of all stack elements. (By convention, we draw stacks upside down, with the stack "top" shown at the bottom of the figure.) The stack pointer `%rsp` holds the address of the top stack element.

The `pushq` instruction provides the ability to push data onto the stack, while the `popq` instruction pops it. Each of these instructions takes a single operand—the data source for pushing and the data destination for popping.

Pushing a quad word value onto the stack involves first decrementing the stack pointer by 8 and then writing the value at the new top-of-stack address.

Therefore, the behavior of the instruction `pushq %rbp` is equivalent to that of the pair of instructions

```
subq $8,%rsp              Decrement stack pointer
movq %rbp,(%rsp)          Store %rbp on stack
```

except that the `pushq` instruction is encoded in the machine code as a single byte, whereas the pair of instructions shown above requires a total of 8 bytes. The first two columns in Figure 3.9 illustrate the effect of executing the instruction `pushq %rax` when `%rsp` is `0x108` and `%rax` is `0x123`. First `%rsp` is decremented by 8, giving `0x100`, and then `0x123` is stored at memory address `0x100`.

Popping a quad word involves reading from the top-of-stack location and then incrementing the stack pointer by 8. Therefore, the instruction `popq %rax` is equivalent to the following pair of instructions:

```
movq (%rsp),%rax          Read %rax from stack
addq $8,%rsp              Increment stack pointer
```

The third column of Figure 3.9 illustrates the effect of executing the instruction `popq %edx` immediately after executing the `pushq`. Value `0x123` is read from memory and written to register `%rdx`. Register `%rsp` is incremented back to `0x108`. As shown in the figure, the value `0x123` remains at memory location `0x104` until it is overwritten (e.g., by another push operation). However, the stack top is always considered to be the address indicated by `%rsp`.

Since the stack is contained in the same memory as the program code and other forms of program data, programs can access arbitrary positions within the stack using the standard memory addressing methods. For example, assuming the topmost element of the stack is a quad word, the instruction `movq 8(%rsp),%rdx` will copy the second quad word from the stack to register `%rdx`.

3.5 Arithmetic and Logical Operations

Figure 3.10 lists some of the x86-64 integer and logic operations. Most of the operations are given as instruction classes, as they can have different variants with different operand sizes. (Only `leaq` has no other size variants.) For example, the instruction class ADD consists of four addition instructions: `addb`, `addw`, `addl`, and `addq`, adding bytes, words, double words, and quad words, respectively. Indeed, each of the instruction classes shown has instructions for operating on these four different sizes of data. The operations are divided into four groups: load effective address, unary, binary, and shifts. *Binary* operations have two operands, while *unary* operations have one operand. These operands are specified using the same notation as described in Section 3.4.

3.5.1 Load Effective Address

The *load effective address* instruction `leaq` is actually a variant of the `movq` instruction. It has the form of an instruction that reads from memory to a register,