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,

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.

but it does not reference memory at all. Its first operand appears to be a memory reference, but instead of reading from the designated location, the instruction copies the effective address to the destination. We indicate this computation in Figure 3.10 using the C address operator $\&S$. This instruction can be used to generate pointers for later memory references. In addition, it can be used to compactly describe common arithmetic operations. For example, if register `%rdx` contains value x, then the instruction `leaq 7(%rdx,%rdx,4), %rax` will set register `%rax` to $5x + 7$. Compilers often find clever uses of `leaq` that have nothing to do with effective address computations. The destination operand must be a register.

Practice Problem 3.6 (solution page 363)

Suppose register `%rbx` holds value p and `%rdx` holds value q. Fill in the table below with formulas indicating the value that will be stored in register `%rax` for each of the given assembly-code instructions:

Instruction	Result
`leaq 9(%rdx), %rax`	
`leaq (%rdx,%rbx), %rax`	
`leaq (%rdx,%rbx,3), %rax`	
`leaq 2(%rbx,%rbx,7), %rax`	

```
leaq 0xE(,%rdx,3), %rax        ________
leaq 6(%rbx,%rdx,7), %rax      ________
```

As an illustration of the use of `leaq` in compiled code, consider the following C program:

```
long scale(long x, long y, long z) {
    long t = x + 4 * y + 12 * z;
    return t;
}
```

When compiled, the arithmetic operations of the function are implemented by a sequence of three `leaq` functions, as is documented by the comments on the right-hand side:

```
  long scale(long x, long y, long z)
  x in %rdi, y in %rsi, z in %rdx
scale:
  leaq    (%rdi,%rsi,4), %rax     x + 4*y
  leaq    (%rdx,%rdx,2), %rdx     z + 2*z = 3*z
  leaq    (%rax,%rdx,4), %rax     (x+4*y) + 4*(3*z) = x + 4*y + 12*z
  ret
```

The ability of the `leaq` instruction to perform addition and limited forms of multiplication proves useful when compiling simple arithmetic expressions such as this example.

Practice Problem 3.7 (solution page 364)

Consider the following code, in which we have omitted the expression being computed:

```
short scale3(short x, short y, short z) {
  short t = __________;
  return t;
}
```

Compiling the actual function with GCC yields the following assembly code:

```
  short scale3(short x, short y, short z)
  x in %rdi, y in %rsi, z in %rdx
scale3:
  leaq (%rsi,%rsi,9), %rbx
  leaq (%rbx,%rdx), %rbx
  leaq (%rbx,%rdi,%rsi), %rbx
  ret
```

Fill in the missing expression in the C code.

3.5.2 Unary and Binary Operations

Operations in the second group are unary operations, with the single operand serving as both source and destination. This operand can be either a register or a memory location. For example, the instruction `incq (%rsp)` causes the 8-byte element on the top of the stack to be incremented. This syntax is reminiscent of the C increment (++) and decrement (--) operators.

The third group consists of binary operations, where the second operand is used as both a source and a destination. This syntax is reminiscent of the C assignment operators, such as `x -= y`. Observe, however, that the source operand is given first and the destination second. This looks peculiar for noncommutative operations. For example, the instruction `subq %rax,%rdx` decrements register `%rdx` by the value in `%rax`. (It helps to read the instruction as "Subtract `%rax` from `%rdx`.") The first operand can be either an immediate value, a register, or a memory location. The second can be either a register or a memory location. As with the MOV instructions, the two operands cannot both be memory locations. Note that when the second operand is a memory location, the processor must read the value from memory, perform the operation, and then write the result back to memory.

Practice Problem 3.8 (solution page 364)

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

Address	Value	Register	Value
`0x100`	`0xFF`	`%rax`	`0x100`
`0x108`	`0xAB`	`%rcx`	`0x1`
`0x110`	`0x13`	`%rdx`	`0x3`
`0x118`	`0x11`		

Fill in the following table showing the effects of the following instructions, in terms of both the register or memory location that will be updated and the resulting value:

Instruction	Destination	Value
`addq %rcx,(%rax)`		
`subq %rdx,8(%rax)`		
`imulq $16,(%rax,%rdx,8)`		
`incq 16(%rax)`		
`decq %rcx`		
`subq %rdx,%rax`		

3.5.3 Shift Operations

The final group consists of shift operations, where the shift amount is given first and the value to shift is given second. Both arithmetic and logical right shifts are possible. The different shift instructions can specify the shift amount either as an immediate value or with the single-byte register %cl. (These instructions are unusual in only allowing this specific register as the operand.) In principle, having a 1-byte shift amount would make it possible to encode shift amounts ranging up to $2^8 - 1 = 255$. With x86-64, a shift instruction operating on data values that are w bits long determines the shift amount from the low-order m bits of register %cl, where $2^m = w$. The higher-order bits are ignored. So, for example, when register %cl has hexadecimal value 0xFF, then instruction salb would shift by 7, while salw would shift by 15, sall would shift by 31, and salq would shift by 63.

As Figure 3.10 indicates, there are two names for the left shift instruction: SAL and SHL. Both have the same effect, filling from the right with zeros. The right shift instructions differ in that SAR performs an arithmetic shift (fill with copies of the sign bit), whereas SHR performs a logical shift (fill with zeros). The destination operand of a shift operation can be either a register or a memory location. We denote the two different right shift operations in Figure 3.10 as $>>_A$ (arithmetic) and $>>_L$ (logical).

Practice Problem 3.9 (solution page 364)

Suppose we want to generate assembly code for the following C function:

```
long shift_left4_rightn(long x, long n)
{
    x <<= 4;
    x >>= n;
    return x;
}
```

The code that follows is a portion of the assembly code that performs the actual shifts and leaves the final value in register %rax. Two key instructions have been omitted. Parameters x and n are stored in registers %rdi and %rsi, respectively.

```
  long shift_left4_rightn(long x, long n)
  x in %rdi, n in %rsi
shift_left4_rightn:
  movq    %rdi, %rax      Get x
  __________________      x <<= 4
  movl    %esi, %ecx      Get n (4 bytes)
  __________________      x >>= n
```

Fill in the missing instructions, following the annotations on the right. The right shift should be performed arithmetically.

(a) C code

```
long arith(long x, long y, long z)
{
    long t1 = x ^ y;
    long t2 = z * 48;
    long t3 = t1 & 0x0F0F0F0F;
    long t4 = t2 - t3;
    return t4;
}
```

(b) Assembly code

```
    long arith(long x, long y, long z)
    x in %rdi, y in %rsi, z in %rdx
1   arith:
2     xorq    %rsi, %rdi              t1 = x ^ y
3     leaq    (%rdx,%rdx,2), %rax     3*z
4     salq    $4, %rax                t2 = 16 * (3*z) = 48*z
5     andl    $252645135, %edi        t3 = t1 & 0x0F0F0F0F
6     subq    %rdi, %rax              Return t2 - t3
7     ret
```

Figure 3.11 C and assembly code for arithmetic function.

3.5.4 Discussion

We see that most of the instructions shown in Figure 3.10 can be used for either unsigned or two's-complement arithmetic. Only right shifting requires instructions that differentiate between signed versus unsigned data. This is one of the features that makes two's-complement arithmetic the preferred way to implement signed integer arithmetic.

Figure 3.11 shows an example of a function that performs arithmetic operations and its translation into assembly code. Arguments `x`, `y`, and `z` are initially stored in registers `%rdi`, `%rsi`, and `%rdx`, respectively. The assembly-code instructions correspond closely with the lines of C source code. Line 2 computes the value of `x^y`. Lines 3 and 4 compute the expression `z*48` by a combination of `leaq` and shift instructions. Line 5 computes the AND of `t1` and `0x0F0F0F0F`. The final subtraction is computed by line 6. Since the destination of the subtraction is register `%rax`, this will be the value returned by the function.

In the assembly code of Figure 3.11, the sequence of values in register `%rax` corresponds to program values `3*z`, `z*48`, and `t4` (as the return value). In general, compilers generate code that uses individual registers for multiple program values and moves program values among the registers.

Practice Problem 3.10 (solution page 365)

Consider the following code, in which we have omitted the expression being computed:

```
short arith3(short x, short y, short z)
{
    short p1 = ________;
    short p2 = ________;
    short p3 = ________;
    short p4 = ________;
    return p4;
}
```

The portion of the generated assembly code implementing these expressions is as follows:

```
  short arith3(short x, short y, short z)
  x in %rdi, y in %rsi, z in %rdx
arith3:
  orq     %rsi, %rdx
  sarq    $9, %rdx
  notq    %rdx
  movq    %rdx, %bax
  subq    %rsi, %rbx
  ret
```

Based on this assembly code, fill in the missing portions of the C code.

Practice Problem 3.11 (solution page 365)

It is common to find assembly-code lines of the form

```
xorq %rcx,%rcx
```

in code that was generated from C where no EXCLUSIVE-OR operations were present.

A. Explain the effect of this particular EXCLUSIVE-OR instruction and what useful operation it implements.

B. What would be the more straightforward way to express this operation in assembly code?

C. Compare the number of bytes to encode any two of these three different implementations of the same operation.

3.5.5 Special Arithmetic Operations

As we saw in Section 2.3, multiplying two 64-bit signed or unsigned integers can yield a product that requires 128 bits to represent. The x86-64 instruction set provides limited support for operations involving 128-bit (16-byte) numbers. Continuing with the naming convention of word (2 bytes), double word (4 bytes), and quad word (8 bytes), Intel refers to a 16-byte quantity as an *oct word*. Figure 3.12

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; 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 3.12 **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.

describes instructions that support generating the full 128-bit product of two 64-bit numbers, as well as integer division.

The `imulq` instruction has two different forms One form, shown in Figure 3.10, is as a member of the IMUL instruction class. In this form, it serves as a "two-operand" multiply instruction, generating a 64-bit product from two 64-bit operands. It implements the operations $*^{u}_{64}$ and $*^{t}_{64}$ described in Sections 2.3.4 and 2.3.5. (Recall that when truncating the product to 64 bits, both unsigned multiply and two's-complement multiply have the same bit-level behavior.)

Additionally, the x86-64 instruction set includes two different "one-operand" multiply instructions to compute the full 128-bit product of two 64-bit values—one for unsigned (`mulq`) and one for two's-complement (`imulq`) multiplication. For both of these instructions, one argument must be in register `%rax`, and the other is given as the instruction source operand. The product is then stored in registers `%rdx` (high-order 64 bits) and `%rax` (low-order 64 bits). Although the name `imulq` is used for two distinct multiplication operations, the assembler can tell which one is intended by counting the number of operands.

As an example, the following C code demonstrates the generation of a 128-bit product of two unsigned 64-bit numbers `x` and `y`:

```
#include <inttypes.h>

typedef unsigned __int128 uint128_t;

void store_uprod(uint128_t *dest, uint64_t x, uint64_t y) {
    *dest = x * (uint128_t) y;
}
```

In this program, we explicitly declare `x` and `y` to be 64-bit numbers, using definitions declared in the file `inttypes.h` , as part of an extension of the C standard. Unfortunately, this standard does not make provisions for 128-bit values. Instead, we rely on support provided by GCC for 128-bit integers, declared using the name `__int128`. Our code uses a `typedef` declaration to define data type `uint128_t`, following the naming pattern for other data types found in `inttypes.h`. The code specifies that the resulting product should be stored at the 16 bytes designated by pointer `dest`.

The assembly code generated by GCC for this function is as follows:

```
   void store_uprod(uint128_t *dest, uint64_t x, uint64_t y)
   dest in %rdi, x in %rsi, y in %rdx
1  store_uprod:
2    movq    %rsi, %rax       Copy x to multiplicand
3    mulq    %rdx             Multiply by y
4    movq    %rax, (%rdi)     Store lower 8 bytes at dest
5    movq    %rdx, 8(%rdi)    Store upper 8 bytes at dest+8
6    ret
```

Observe that storing the product requires two `movq` instructions: one for the low-order 8 bytes (line 4), and one for the high-order 8 bytes (line 5). Since the code is generated for a little-endian machine, the high-order bytes are stored at higher addresses, as indicated by the address specification `8(%rdi)`.

Our earlier table of arithmetic operations (Figure 3.10) does not list any division or modulus operations. These operations are provided by the single-operand divide instructions similar to the single-operand multiply instructions. The signed division instruction `idivl` takes as its dividend the 128-bit quantity in registers `%rdx` (high-order 64 bits) and `%rax` (low-order 64 bits). The divisor is given as the instruction operand. The instruction stores the quotient in register `%rax` and the remainder in register `%rdx`.

For most applications of 64-bit addition, the dividend is given as a 64-bit value. This value should be stored in register `%rax`. The bits of `%rdx` should then be set to either all zeros (unsigned arithmetic) or the sign bit of `%rax` (signed arithmetic). The latter operation can be performed using the instruction `cqto`.[2] This instruction takes no operands—it implicitly reads the sign bit from `%rax` and copies it across all of `%rdx`.

As an illustration of the implementation of division with x86-64, the following C function computes the quotient and remainder of two 64-bit, signed numbers:

```
void remdiv(long x, long y,
            long *qp, long *rp) {
    long q = x/y;
    long r = x%y;
    *qp = q;
    *rp = r;
}
```

2. This instruction is called `cqo` in the Intel documentation, one of the few cases where the ATT-format name for an instruction does not match the Intel name.

This compiles to the following assembly code:

```
   void remdiv(long x, long y, long *qp, long *rp)
   x in %rdi, y in %rsi, qp in %rdx, rp in %rcx
1  remdiv:
2    movq    %rdx, %r8       Copy qp
3    movq    %rdi, %rax      Move x to lower 8 bytes of dividend
4    cqto                    Sign-extend to upper 8 bytes of dividend
5    idivq   %rsi            Divide by y
6    movq    %rax, (%r8)     Store quotient at qp
7    movq    %rdx, (%rcx)    Store remainder at rp
8    ret
```

In this code, argument rp must first be saved in a different register (line 2), since argument register `%rdx` is required for the division operation. Lines 3–4 then prepare the dividend by copying and sign-extending `x`. Following the division, the quotient in register `%rax` gets stored at `qp` (line 6), while the remainder in register `%rdx` gets stored at `rp` (line 7).

Unsigned division makes use of the `divq` instruction. Typically, register `%rdx` is set to zero beforehand.

Practice Problem 3.12 (solution page 365)

Consider the following function for computing the quotient and remainder of two unsigned 64-bit numbers:

```
void uremdiv(unsigned long x, unsigned long y,
             unsigned long *qp, unsigned long *rp) {
    unsigned long q = x/y;
    unsigned long r = x%y;
    *qp = q;
    *rp = r;
}
```

Modify the assembly code shown for signed division to implement this function.

3.6 Control

So far, we have only considered the behavior of *straight-line* code, where instructions follow one another in sequence. Some constructs in C, such as conditionals, loops, and switches, require conditional execution, where the sequence of operations that get performed depends on the outcomes of tests applied to the data. Machine code provides two basic low-level mechanisms for implementing conditional behavior: it tests data values and then alters either the control flow or the data flow based on the results of these tests.

Data-dependent control flow is the more general and more common approach for implementing conditional behavior, and so we will examine this first. Normally,