

3

Machine-Level Representation of Programs

- 3.1 A Historical Perspective 202
- 3.2 Program Encodings 205
- 3.3 Data Formats 213
- 3.4 Accessing Information 215
- 3.5 Arithmetic and Logical Operations 227
- 3.6 Control 236
- 3.7 Procedures 274
- 3.8 Array Allocation and Access 291
- 3.9 Heterogeneous Data Structures 301
- 3.10 Combining Control and Data in Machine-Level Programs 312
- 3.11 Floating-Point Code 329
- 3.12 Summary 345
 - Bibliographic Notes 346
 - Homework Problems 347
 - Solutions to Practice Problems 361

Computers execute *machine code*, sequences of bytes encoding the low-level operations that manipulate data, manage memory, read and write data on storage devices, and communicate over networks. A compiler generates machine code through a series of stages, based on the rules of the programming language, the instruction set of the target machine, and the conventions followed by the operating system. The gcc C compiler generates its output in the form of *assembly code*, a textual representation of the machine code giving the individual instructions in the program. Gcc then invokes both an *assembler* and a *linker* to generate the executable machine code from the assembly code. In this chapter, we will take a close look at machine code and its human-readable representation as assembly code.

When programming in a high-level language such as C, and even more so in Java, we are shielded from the detailed machine-level implementation of our program. In contrast, when writing programs in assembly code (as was done in the early days of computing) a programmer must specify the low-level instructions the program uses to carry out a computation. Most of the time, it is much more productive and reliable to work at the higher level of abstraction provided by a high-level language. The type checking provided by a compiler helps detect many program errors and makes sure we reference and manipulate data in consistent ways. With modern optimizing compilers, the generated code is usually at least as efficient as what a skilled assembly-language programmer would write by hand. Best of all, a program written in a high-level language can be compiled and executed on a number of different machines, whereas assembly code is highly machine specific.

So why should we spend our time learning machine code? Even though compilers do most of the work in generating assembly code, being able to read and understand it is an important skill for serious programmers. By invoking the compiler with appropriate command-line parameters, the compiler will generate a file showing its output in assembly-code form. By reading this code, we can understand the optimization capabilities of the compiler and analyze the underlying inefficiencies in the code. As we will experience in Chapter 5, programmers seeking to maximize the performance of a critical section of code often try different variations of the source code, each time compiling and examining the generated assembly code to get a sense of how efficiently the program will run. Furthermore, there are times when the layer of abstraction provided by a high-level language hides information about the run-time behavior of a program that we need to understand. For example, when writing concurrent programs using a thread package, as covered in Chapter 12, it is important to understand how program data are shared or kept private by the different threads and precisely how and where shared data are accessed. Such information is visible at the machine-code level. As another example, many of the ways programs can be attacked, allowing malware to infect a system, involve nuances of the way programs store their run-time control information. Many attacks involve exploiting weaknesses in system programs to overwrite information and thereby take control of the system. Understanding how these vulnerabilities arise and how to guard against them requires a knowledge of the machine-level representation of programs. The need for programmers to learn

machine code has shifted over the years from one of being able to write programs directly in assembly code to one of being able to read and understand the code generated by compilers.

In this chapter, we will learn the details of one particular assembly language and see how C programs get compiled into this form of machine code. Reading the assembly code generated by a compiler involves a different set of skills than writing assembly code by hand. We must understand the transformations typical compilers make in converting the constructs of C into machine code. Relative to the computations expressed in the C code, optimizing compilers can rearrange execution order, eliminate unneeded computations, replace slow operations with faster ones, and even change recursive computations into iterative ones. Understanding the relation between source code and the generated assembly can often be a challenge—it's much like putting together a puzzle having a slightly different design than the picture on the box. It is a form of *reverse engineering*—trying to understand the process by which a system was created by studying the system and working backward. In this case, the system is a machine-generated assembly-language program, rather than something designed by a human. This simplifies the task of reverse engineering because the generated code follows fairly regular patterns and we can run experiments, having the compiler generate code for many different programs. In our presentation, we give many examples and provide a number of exercises illustrating different aspects of assembly language and compilers. This is a subject where mastering the details is a prerequisite to understanding the deeper and more fundamental concepts. Those who say “I understand the general principles, I don't want to bother learning the details” are deluding themselves. It is critical for you to spend time studying the examples, working through the exercises, and checking your solutions with those provided.

Our presentation is based on x86-64, the machine language for most of the processors found in today's laptop and desktop machines, as well as those that power very large data centers and supercomputers. This language has evolved over a long history, starting with Intel Corporation's first 16-bit processor in 1978, through to the expansion to 32 bits, and most recently to 64 bits. Along the way, features have been added to make better use of the available semiconductor technology, and to satisfy the demands of the marketplace. Much of the development has been driven by Intel, but its rival Advanced Micro Devices (AMD) has also made important contributions. The result is a rather peculiar design with features that make sense only when viewed from a historical perspective. It is also laden with features providing backward compatibility that are not used by modern compilers and operating systems. We will focus on the subset of the features used by gcc and Linux. This allows us to avoid much of the complexity and many of the arcane features of x86-64.

Our technical presentation starts with a quick tour to show the relation between C, assembly code, and machine code. We then proceed to the details of x86-64, starting with the representation and manipulation of data and the implementation of control. We see how control constructs in C, such as `if`, `while`, and `switch` statements, are implemented. We then cover the implementation of procedures, including how the program maintains a run-time stack to support the

Web Aside ASM:IA32 IA32 programming

IA32, the 32-bit predecessor to x86-64, was introduced by Intel in 1985. It served as the machine language of choice for several decades. Most x86 microprocessors sold today, and most operating systems installed on these machines, are designed to run x86-64. However, they can also execute IA32 programs in a backward compatibility mode. As a result, many application programs are still based on IA32. In addition, many existing systems cannot execute x86-64, due to limitations of their hardware or system software. IA32 continues to be an important machine language. You will find that having a background in x86-64 will enable you to learn the IA32 machine language quite readily.

passing of data and control between procedures, as well as storage for local variables. Next, we consider how data structures such as arrays, structures, and unions are implemented at the machine level. With this background in machine-level programming, we can examine the problems of out-of-bounds memory references and the vulnerability of systems to buffer overflow attacks. We finish this part of the presentation with some tips on using the GDB debugger for examining the run-time behavior of a machine-level program. The chapter concludes with a presentation on machine-program representations of code involving floating-point data and operations.

The computer industry has recently made the transition from 32-bit to 64-bit machines. A 32-bit machine can only make use of around 4 gigabytes (2^{32} bytes) of random access memory. With memory prices dropping at dramatic rates, and our computational demands and data sizes increasing, it has become both economically feasible and technically desirable to go beyond this limitation. Current 64-bit machines can use up to 256 terabytes (2^{48} bytes) of memory, and could readily be extended to use up to 16 exabytes (2^{64} bytes). Although it is hard to imagine having a machine with that much memory, keep in mind that 4 gigabytes seemed like an extreme amount of memory when 32-bit machines became commonplace in the 1970s and 1980s.

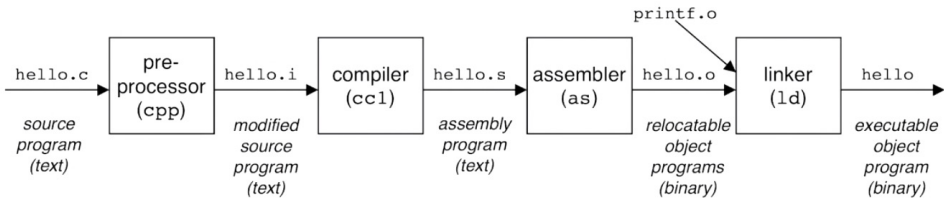
Our presentation focuses on the types of machine-level programs generated when compiling C and similar programming languages targeting modern operating systems. As a consequence, we make no attempt to describe many of the features of x86-64 that arise out of its legacy support for the styles of programs written in the early days of microprocessors, when much of the code was written manually and where programmers had to struggle with the limited range of addresses allowed by 16-bit machines.

3.1 A Historical Perspective

The Intel processor line, colloquially referred to as *x86*, has followed a long evolutionary development. It started with one of the first single-chip 16-bit microprocessors, where many compromises had to be made due to the limited capabilities of integrated circuit technology at the time. Since then, it has grown to take ad-

Typical Compilation System

```
gcc -o hello hello.c
```



Source: Bryant & O'Hallaron

Programming Abstractions

We can program a microprocessor using

- a) Instruction opcodes (also called Machine Code)
- b) Assembly language ✓
- c) High level programming languages

- ☐ The level of **abstraction** increases from Top to Bottom.
- ☐ As the level of abstraction increases, ease of programmability also increases!
- ☐ Hmm, but we may lose the fine-grained control over the underlying hardware?

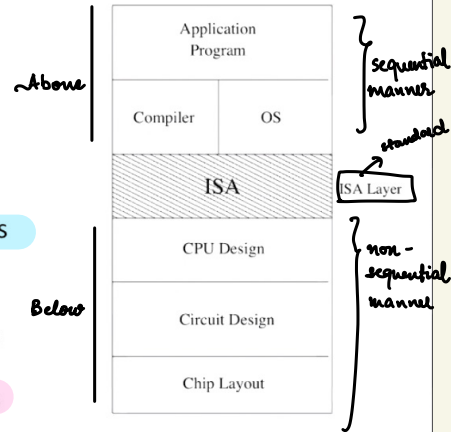
Instruction Set Architecture (ISA)

Assembly Language View

- Processor state: registers, memory, etc.
- Instructions and how instructions are encoded

Layer of Abstraction

- Above: how to program machine, processor executes instructions sequentially
- Below: What needs to be built
 - Use variety of tricks to make it run faster, e.g., execute multiple instructions simultaneously
 - Safeguard to ensure that the overall behavior matches the sequential operation dictated by the ISA.



(*) ISA provides the standard

(*) x86 - IA32 - family of 32 bit processor from Intel
Instruction Set Architecture

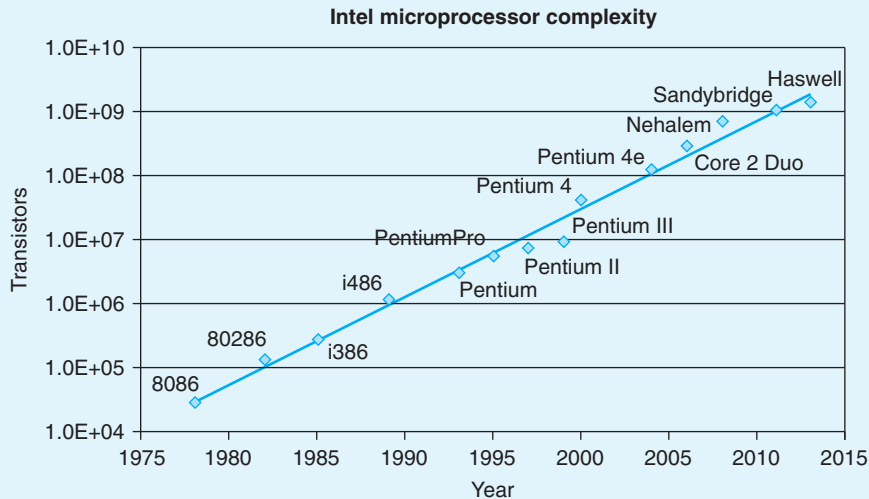
32 bit proc

x86 - IA32

Instruction Set Architecture

- x86-64, describe the behavior of a program as if each instruction is executed in sequence, with one instruction completing before the next one begins.
- The processor hardware is far more elaborate, executing many instructions concurrently, but they employ safeguards to ensure that the overall behavior matches the sequential operation dictated by the ISA.
- Machine-level program use a virtual address space, providing a memory model that appears to be a very large byte array.
- The actual implementation of the memory system involves a combination of multiple hardware memories and operating system software

Aside Moore's Law



If we plot the number of transistors in the different Intel processors versus the year of introduction, and use a logarithmic scale for the y-axis, we can see that the growth has been phenomenal. Fitting a line through the data, we see that the number of transistors increases at an annual rate of approximately 37%, meaning that the number of transistors doubles about every 26 months. This growth has been sustained over the multiple-decade history of x86 microprocessors.

In 1965, Gordon Moore, a founder of Intel Corporation, extrapolated from the chip technology of the day (by which they could fabricate circuits with around 64 transistors on a single chip) to predict that the number of transistors per chip would double every year for the next 10 years. This prediction became known as *Moore's Law*. As it turns out, his prediction was just a little bit optimistic, but also too short-sighted. Over more than 50 years, the semiconductor industry has been able to double transistor counts on average every 18 months.

Similar exponential growth rates have occurred for other aspects of computer technology, including the storage capacities of magnetic disks and semiconductor memories. These remarkable growth rates have been the major driving forces of the computer revolution.

with the introduction of SSE2. Although we see vestiges of the historical evolution of x86 in x86-64 programs, many of the most arcane features of x86 do not appear.

3.2 Program Encodings

Suppose we write a C program as two files `p1.c` and `p2.c`. We can then compile this code using a Unix command line:

```
linux> gcc -Og -o p p1.c p2.c
```

The command `gcc` indicates the gcc C compiler. Since this is the default compiler on Linux, we could also invoke it as simply `cc`. The command-line option `-Og`¹ instructs the compiler to apply a level of optimization that yields machine code that follows the overall structure of the original C code. Invoking higher levels of optimization can generate code that is so heavily transformed that the relationship between the generated machine code and the original source code is difficult to understand. We will therefore use `-Og` optimization as a learning tool and then see what happens as we increase the level of optimization. In practice, higher levels of optimization (e.g., specified with the option `-O1` or `-O2`) are considered a better choice in terms of the resulting program performance.

The `gcc` command invokes an entire sequence of programs to turn the source code into executable code. First, the C *preprocessor* expands the source code to include any files specified with `#include` commands and to expand any macros, specified with `#define` declarations. Second, the *compiler* generates assembly-code versions of the two source files having names `p1.s` and `p2.s`. Next, the *assembler* converts the assembly code into binary *object-code* files `p1.o` and `p2.o`. Object code is one form of machine code—it contains binary representations of all of the instructions, but the addresses of global values are not yet filled in. Finally, the *linker* merges these two object-code files along with code implementing library functions (e.g., `printf`) and generates the final executable code file `p` (as specified by the command-line directive `-o p`). Executable code is the second form of machine code we will consider—it is the exact form of code that is executed by the processor. The relation between these different forms of machine code and the linking process is described in more detail in Chapter 7.

3.2.1 Machine-Level Code

As described in Section 1.9.3, computer systems employ several different forms of abstraction, hiding details of an implementation through the use of a simpler abstract model. Two of these are especially important for machine-level programming. First, the format and behavior of a machine-level program is defined by the *instruction set architecture*, or ISA, defining the processor state, the format of the instructions, and the effect each of these instructions will have on the state. Most ISAs, including x86-64, describe the behavior of a program as if each instruction is executed in sequence, with one instruction completing before the next one begins. The processor hardware is far more elaborate, executing many instructions concurrently, but it employs safeguards to ensure that the overall behavior matches the sequential operation dictated by the ISA. Second, the memory addresses used by a machine-level program are *virtual addresses*, providing a memory model that

1. This optimization level was introduced in gcc version 4.8. Earlier versions of gcc, as well as non-GNU compilers, will not recognize this option. For these, using optimization level one (specified with the command-line flag `-O1`) is probably the best choice for generating code that follows the original program structure.

appears to be a very large byte array. The actual implementation of the memory system involves a combination of multiple hardware memories and operating system software, as described in Chapter 9.

The compiler does most of the work in the overall compilation sequence, transforming programs expressed in the relatively abstract execution model provided by C into the very elementary instructions that the processor executes. The assembly-code representation is very close to machine code. Its main feature is that it is in a more readable textual format, as compared to the binary format of machine code. Being able to understand assembly code and how it relates to the original C code is a key step in understanding how computers execute programs.

The machine code for x86-64 differs greatly from the original C code. Parts of the processor state are visible that normally are hidden from the C programmer:

- The *program counter* (commonly referred to as the *PC*, and called `%rip` in x86-64) indicates the address in memory of the next instruction to be executed.
- The *integer register file* contains 16 named locations storing 64-bit values. These registers can hold addresses (corresponding to C pointers) or integer data. Some registers are used to keep track of critical parts of the program state, while others are used to hold temporary data, such as the arguments and local variables of a procedure, as well as the value to be returned by a function.
- The *condition code registers* hold status information about the most recently executed arithmetic or logical instruction. These are used to implement conditional changes in the control or data flow, such as is required to implement `if` and `while` statements.
- A set of *vector registers* can each hold one or more integer or floating-point values.

Whereas C provides a model in which objects of different data types can be declared and allocated in memory, machine code views the memory as simply a large byte-addressable array. Aggregate data types in C such as arrays and structures are represented in machine code as contiguous collections of bytes. Even for scalar data types, assembly code makes no distinctions between signed or unsigned integers, between different types of pointers, or even between pointers and integers.

The program memory contains the executable machine code for the program, some information required by the operating system, a run-time stack for managing procedure calls and returns, and blocks of memory allocated by the user (e.g., by using the `malloc` library function). As mentioned earlier, the program memory is addressed using virtual addresses. At any given time, only limited subranges of virtual addresses are considered valid. For example, x86-64 virtual addresses are represented by 64-bit words. In current implementations of these machines, the upper 16 bits must be set to zero, and so an address can potentially specify a byte over a range of 2^{48} , or 64 terabytes. More typical programs will only have access to a few megabytes, or perhaps several gigabytes. The operating system manages

Data Formats

- Intel uses the term "word" to refer to a 16-bit data type.
- Hence, 32-bit quantities as "double words" and 64-bit quantities as "quad words."

C declaration	Intel data type	Assembly code suffix	Size (bytes)
char	Byte	<u>b</u>	1
short	<u>Word</u>	<u>w</u>	<u>2</u>
int	Double word	l	4
long int	Double word	l	4
long long int	—	—	4
<u>char *</u>	Double word	<u>l</u>	4
float	Single precision	<u>s</u>	4
double	Double precision	<u>l</u>	8
long double	Extended precision	<u>t</u>	10/12

movb
 movw
 movl
 movq
 movd
 movq

for 32
 for 64

Sizes of C data types in IA32.

The MOVL instruction was generated because you put two int (i and j variables), MOVL will perform a MOV of 32 bits, and integer' size is 32 bits.

a non exhaustive list of all MOV* exist (like MOVD for doubleword or MOVQ for quadword) to allow to optimize your code and use the better expression to gain most time as possible.

Operand Specifiers

movb \$0x3, %eax
0011 0001

- Most instructions have one or more operands, used as source and destination references
- Three types of operand exists
 - Constant (Immediate)
 - Register
 - Memory

Type	Form	Operand value	Name
Immediate	<u>\$Imm</u>	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] · s]	Scaled indexed
Memory	Imm(, E _i , s)	M[Imm + R[E _i] · s]	Scaled indexed
Memory	(E _b , E _i , s)	M[R[E _b] + R[E _i] · s]	Scaled indexed
Memory	Imm(E _b , E _i , s)	M[Imm + R[E _b] + R[E _i] · s]	Scaled indexed

Operand Specifiers

Address	Value	Register	Value
0x100	0xFF	%eax	0x100
0x104	0xAB	%ecx	0x1
0x108	0x13	%edx	0x3
0x10C	0x11		

Operand

Value

%eax	0x100
0x104	0xAB
\$0x108	0x108
(%eax)	0xFF
4(%eax)	
9(%eax,%edx)	
260(%ecx,%edx)	
0xFC(,%ecx,4)	
(%eax,%edx,4)	

value stored at register %eax
↓
Register

value stored at address 0x104

take the literal value of 0x108 not address
\$ ⇒ this hexadecimal itself is the value
this hexadecimal is not address

value at memory which is value of eax → 0x100 → value stored at 0x100 is 0xFF

mov memory → memory x
mov memory → constant x
mov memory → register ✓

mov register/
constant
↓
memory

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 = FFFFFFFF7FAA
5  movzbq  %dl,%rax                    %rax = 000000000000AA
```

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.
```

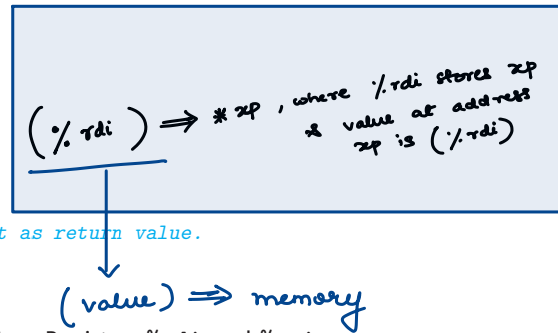


Figure 3.7 C and assembly code for exchange routine. Registers `%rdi` and `%rsi` hold parameters `xp` and `y`, respectively.

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	<hr/> <hr/>

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

Instructions

Data movement

<code>movq Src, Dest</code>	<code>Dest = Src</code>
<code>movsbq Src, Dest</code>	<code>Dest (quad) = Src (byte), sign-extend</code>
<code>movzbq Src, Dest</code>	<code>Dest (quad) = Src (byte), zero-extend</code>

$(\%edx)$
↓
 $(*xp)$

(a) C code

```
1 int exchange(int *xp, int y)
2 {
3     int x = *xp;
4
5     *xp = y;
6     return x;
7 }
```

(b) Assembly code

```
1 movl 8(%ebp), %edx      Get xp
2 movl (%edx), %eax      Get x at xp
3 movl 12(%ebp), %ecx    Get y
4 movl %ecx, (%edx)      Store y at xp
```

By copying to %eax below, x becomes the return value

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

Figure 3.8 Push and pop instructions.

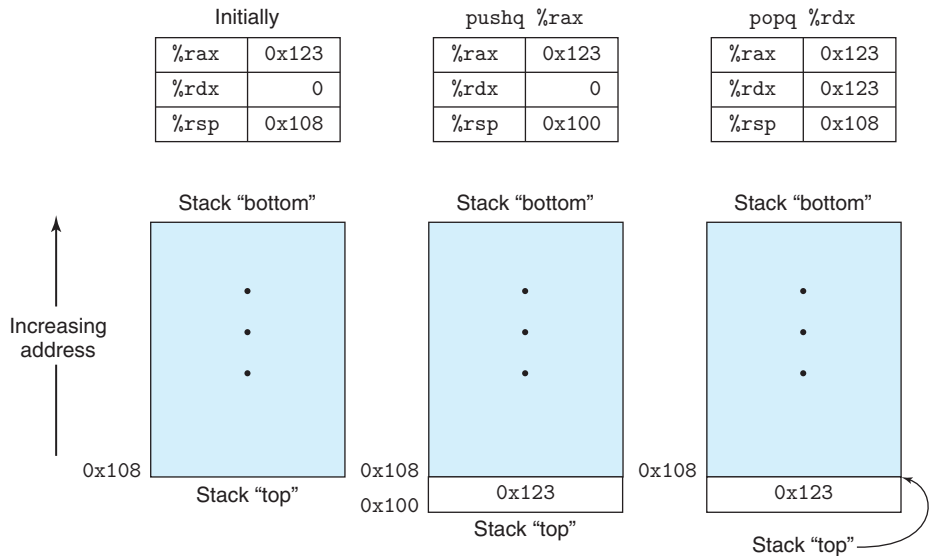


Figure 3.9 Illustration of stack operation. By convention, we draw stacks upside down, so that the “top” of the stack is shown at the bottom. With x86-64, stacks grow toward lower addresses, so pushing involves decrementing the stack pointer (register `%rsp`) and storing to memory, while popping involves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

<code>subq \$8,%rsp</code>	<i>Decrement stack pointer</i>
<code>movq %rbp, (%rsp)</code>	<i>Store %rbp on stack</i>



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:

<code>movq (%rsp), %rax</code>	<i>Read %rax from stack</i>
<code>addq \$8,%rsp</code>	<i>Increment stack pointer</i>

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 (same as $x++$)
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 (subtract source from destination)
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \wedge S$	Exclusive-or
OR	S, D	$D \leftarrow D \vee S$	Or
AND	S, D	$D \leftarrow D \& S$	And
SAL	k, D	$D \leftarrow D \ll k$	Left shift
SHL	k, D	$D \leftarrow D \ll k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D \gg_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D \gg_L k$	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 \gg_A and \gg_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
<code>leaq 9(%rdx), %rax</code>	_____
<code>leaq (%rdx,%rbx), %rax</code>	_____
<code>leaq (%rdx,%rbx,3), %rax</code>	_____
<code>leaq 2(%rbx,%rbx,7), %rax</code>	_____

lea — Load effective address

The lea instruction places the *address* specified by its second operand into the register specified by its first operand. Note, the *contents* of the memory location are not loaded, only the effective address is computed and placed into the register. This is useful for obtaining a pointer into a memory region.

Syntax
lea <reg32>, <mem>

Examples
lea edi, [ebx+4*esi] — the quantity EBX+4*ESI is placed in EDI.
lea eax, [var] — the value in *var* is placed in EAX.
lea eax, [val] — the value *val* is placed in EAX.

leaq vs. movq example (solution)

Registers

%rax	0x110
%rbx	0x8
%rcx	0x4
%rdx	0x100
%rdi	0x100
%rsi	0x1

Memory

	Address
0x400	0x120
0xf	0x118
0x8	0x110
0x10	0x108
0x1	0x100

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

Instruction	Effect	Description
leaq <i>S, D</i>	$D \leftarrow \&S$	Load effective address

Arithmetic and Logical Operations

$Y = 8X + 9$
 $\text{leal } 9(\%eax, \%eax, 7) \quad \%edx$
 $X \quad \%eax$
 $Y \quad \%edx$
 $(X + 7X) + 9$

Instruction	Effect	Description
<u>leal</u> <u>S, D</u>	$D \leftarrow \&S$	Load effective address

Load effective address by (leal)

$\text{leal } 8(\%ebp, \%eax, 4) \quad \%edx$
 $(8 + \%ebp + 4 * \%eax) \rightarrow 0xfcff$
 A plus name which is eight X + 9.

Arithmetic and Logical Operations

$\text{movl } 8(\%ebp, \%eax, 4) \quad \%edx$
 $0xfcff$
 $\%edx$

Instruction	Effect	Description
<u>leal</u> <u>S, D</u>	$D \leftarrow \&S$	Load effective address

Load effective address by (leal)

$\text{leal } 8(\%ebp, \%eax, 4) \quad \%edx$
 $\rightarrow 0xfcff$

not the store is zero XCFE not the value

Arithmetic Example

```

long arith(long x, long y, long z)
{
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
    
```

```

arith:
    leaq    (%rdi,%rsi), %rax
    addq    %rdx, %rax
    leaq    (%rsi,%rsi,2), %rdx
    salq    $4, %rdx
    leaq    4(%rdi,%rdx), %rcx
    imulq   %rcx, %rax
    ret
    
```

Register	Use(s)
%rdi	1 st argument (x)
%rsi	2 nd argument (y)
%rdx	3 rd argument (z)

- ❖ Interesting Instructions
- leaq: “address” computation
 - salq: shift
 - imulq: multiplication
 - Only used once!

6

Arithmetic Example

```

long arith(long x, long y, long z)
{
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
    
```

Register	Use(s)
%rdi	x
%rsi	y
%rdx	z, t4
%rax	t1, t2, rval
%rcx	t5

```

arith:
    leaq    (%rdi,%rsi), %rax      # rax/t1    = x + y
    addq    %rdx, %rax             # rax/t2    = t1 + z
    leaq    (%rsi,%rsi,2), %rdx    # rdx       = 3 * y
    salq    $4, %rdx               # rdx/t4    = (3*y) * 16
    leaq    4(%rdi,%rdx), %rcx     # rcx/t5    = x + t4 + 4
    imulq   %rcx, %rax             # rax/rval  = t5 * t2
    ret
    
```

7

```
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
```

basically will calculate
(%rdi,%rsi,4) & then
give a memory address
but here we want the
value itself not treat
it as
address

so `leaq` useful
since it does what commands like
ADD, MUL, ...
instead use syntax
(base, index, scale) &
gets this value
this value is not used
as determining in `leaq`,
so it becomes useful
since they don't go & get
value at memory but
use calculated value
directly.

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
<code>addq %rcx, (%rax)</code>	_____	_____
<code>subq %rdx, 8(%rax)</code>	_____	_____
<code>imulq \$16, (%rax,%rdx,8)</code>	_____	_____
<code>incq 16(%rax)</code>	_____	_____
<code>decq %rcx</code>	_____	_____
<code>subq %rdx,%rax</code>	_____	_____

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 %c1. (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 %c1, where $2^m = w$. The higher-order bits are ignored. So, for example, when register %c1 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.

FF = 1111 1111
salb \Rightarrow
byte \rightarrow 8 bits
 $2^3 = 8 \Rightarrow m = 3$
so 111 = 7

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 \gg_A (arithmetic) and \gg_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.

General Purpose Registers



Arithmetic Right Shifts

When shifting right with an **arithmetic right shift**, the least-significant bit is lost and the most-significant bit is *copied*.

Languages handle arithmetic and logical right shifting in different ways. Java provides two right shift operators: `>>` does an *arithmetic* right shift and `>>>` does a *logical* right shift.

```
1011 >> 1 → 1101
1011 >> 3 → 1111

0011 >> 1 → 0001
0011 >> 2 → 0000
```

The first two numbers had a 1 as the most significant bit, so more 1's were inserted during the shift. The last two numbers had a 0 as the most significant bit, so the shift inserted more 0's.

If a number is encoded using two's complement, then an arithmetic right shift preserves the number's sign, while a logical right shift makes the number positive.

```
// Arithmetic shift
1011 >> 1 → 1101
1011 is -5
1101 is -3

// Logical shift
1111 >>> 1 → 0111
1111 is -1
0111 is 7
```

(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 $x \wedge y$. 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.

- Explain the effect of this particular EXCLUSIVE-OR instruction and what useful operation it implements.
 - What would be the more straightforward way to express this operation in assembly code?
 - 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
<code>imulq</code>	<code>S</code>	$R[\%rdx]:R[\%rax] \leftarrow S \times R[\%rax]$	Signed full multiply
<code>mulq</code>	<code>S</code>	$R[\%rdx]:R[\%rax] \leftarrow S \times R[\%rax]$	Unsigned full multiply
<code>cqto</code>		$R[\%rdx]:R[\%rax] \leftarrow \text{SignExtend}(R[\%rax])$	Convert to oct word
<code>idivq</code>	<code>S</code>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Signed divide
<code>divq</code>	<code>S</code>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	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 $*_{64}^u$ and $*_{64}^s$ 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    %rax               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~~^{qp} 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,

both statements in C and instructions in machine code are executed *sequentially*, in the order they appear in the program. The execution order of a set of machine-code instructions can be altered with a *jump* instruction, indicating that control should pass to some other part of the program, possibly contingent on the result of some test. The compiler must generate instruction sequences that build upon this low-level mechanism to implement the control constructs of C.

In our presentation, we first cover the two ways of implementing conditional operations. We then describe methods for presenting loops and switch statements.

3.6.1 Condition Codes

In addition to the integer registers, the CPU maintains a set of single-bit *condition code* registers describing attributes of the most recent arithmetic or logical operation. These registers can then be tested to perform conditional branches. These condition codes are the most useful:

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.

For example, suppose we used one of the `ADD` instructions to perform the equivalent of the C assignment `t = a+b`, where variables `a`, `b`, and `t` are integers. Then the condition codes would be set according to the following C expressions:

CF	<code>(unsigned) t < (unsigned) a</code>	Unsigned overflow
ZF	<code>(t == 0)</code>	Zero
SF	<code>(t < 0)</code>	Negative
OF	<code>(a < 0 == b < 0) && (t < 0 != a < 0)</code>	Signed overflow

The `leaq` instruction does not alter any condition codes, since it is intended to be used in address computations. Otherwise, all of the instructions listed in Figure 3.10 cause the condition codes to be set. For the logical operations, such as `XOR`, the carry and overflow flags are set to zero. For the shift operations, the carry flag is set to the last bit shifted out, while the overflow flag is set to zero. For reasons that we will not delve into, the `INC` and `DEC` instructions set the overflow and zero flags, but they leave the carry flag unchanged.

In addition to the setting of condition codes by the instructions of Figure 3.10, there are two instruction classes (having 8-, 16-, 32-, and 64-bit forms) that set condition codes without altering any other registers; these are listed in Figure 3.13. The `CMP` instructions set the condition codes according to the differences of their two operands. They behave in the same way as the `SUB` instructions, except that they set the condition codes without updating their destinations. With AT&T format,

Instruction		Based on	Description
CMP	S_1, S_2	$S_2 - S_1$	Compare
cmpb			Compare byte
cmpw			Compare word
cmpd			Compare double word
cmpq			Compare quad word
TEST	S_1, S_2	$S_1 \& S_2$	Test
testb			Test byte
testw			Test word
testd			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.

the operands are listed in reverse order, making the code difficult to read. These instructions set the zero flag if the two operands are equal. The other flags can be used to determine ordering relations between the two operands. The **TEST** instructions behave in the same manner as the **AND** instructions, except that they set the condition codes without altering their destinations. Typically, the same operand is repeated (e.g., `testq %rax, %rax` to see whether `%rax` is negative, zero, or positive), or one of the operands is a mask indicating which bits should be tested.

3.6.2 Accessing the Condition Codes

Rather than reading the condition codes directly, there are three common ways of using the condition codes: (1) we can set a single byte to 0 or 1 depending on some combination of the condition codes, (2) we can conditionally jump to some other part of the program, or (3) we can conditionally transfer data. For the first case, the instructions described in Figure 3.14 set a single byte to 0 or 1 depending on some combination of the condition codes. We refer to this entire class of instructions as the **SET** instructions; they differ from one another based on which combinations of condition codes they consider, as indicated by the different suffixes for the instruction names. It is important to recognize that the suffixes for these instructions denote different conditions and not different operand sizes. For example, instructions `setl` and `setb` denote “set less” and “set below,” not “set long word” or “set byte.”

A **SET** instruction has either one of the low-order single-byte register elements (Figure 3.2) or a single-byte memory location as its destination, setting this byte to either 0 or 1. To generate a 32-bit or 64-bit result, we must also clear the high-order bits. A typical instruction sequence to compute the C expression `a < b`, where `a` and `b` are both of type `long`, proceeds as follows:

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

not of less than

negative which is not overflow? $a < b$
negative overflow $\Rightarrow a > b$

basically
<, >, =
but
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.

$a < b$

```
int comp(data_t a, data_t b)
a in %rdi, b in %rsi
```

```
1  comp:
2      cmpq    %rsi, %rdi    Compare ba      cmp b a    a - b
3      setl    %al           Set low-order byte of %eax to 0 or 1
4      movzbl  %al, %eax     Clear rest of %eax (and rest of %rax)
5      ret
```

Note the comparison order of the `cmpq` instruction (line 2). Although the arguments are listed in the order `%rsi` (*b*), then `%rdi` (*a*), the comparison is really between *a* and *b*. Recall also, as discussed in Section 3.4.2, that the `movzbl` instruction (line 4) clears not just the high-order 3 bytes of `%eax`, but the upper 4 bytes of the entire register, `%rax`, as well.

For some of the underlying machine instructions, there are multiple possible names, which we list as “synonyms.” For example, both `setg` (for “set greater”) and `setnle` (for “set not less or equal”) refer to the same machine instruction. Compilers and disassemblers make arbitrary choices of which names to use.

Although all arithmetic and logical operations set the condition codes, the descriptions of the different SET instructions apply to the case where a comparison instruction has been executed, setting the condition codes according to the computation $t = a - b$. More specifically, let *a*, *b*, and *t* be the integers represented in two’s-complement form by variables *a*, *b*, and *t*, respectively, and so $t = a -_w^t b$, where *w* depends on the sizes associated with *a* and *b*.

Consider the `sete`, or “set when equal,” instruction. When $a = b$, we will have $t = 0$, and hence the zero flag indicates equality. Similarly, consider testing for signed comparison with the `setl`, or “set when less,” instruction. When no overflow occurs (indicated by having `OF` set to 0), we will have $a < b$ when $a -^t_w b < 0$, indicated by having `SF` set to 1, and $a \geq b$ when $a -^t_w b \geq 0$, indicated by having `SF` set to 0. On the other hand, when overflow occurs, we will have $a < b$ when $a -^t_w b > 0$ (negative overflow) and $a > b$ when $a -^t_w b < 0$ (positive overflow). We cannot have overflow when $a = b$. Thus, when `OF` is set to 1, we will have $a < b$ if and only if `SF` is set to 0. Combining these cases, the `EXCLUSIVE-OR` of the overflow and sign bits provides a test for whether $a < b$. The other signed comparison tests are based on other combinations of `SF` \wedge `OF` and `ZF`.

For the testing of unsigned comparisons, we now let a and b be the integers represented in unsigned form by variables `a` and `b`. In performing the computation $t = a - b$, the carry flag will be set by the `CMP` instruction when $a - b < 0$, and so the unsigned comparisons use combinations of the carry and zero flags.

It is important to note how machine code does or does not distinguish between signed and unsigned values. Unlike in C, it does not associate a data type with each program value. Instead, it mostly uses the same instructions for the two cases, because many arithmetic operations have the same bit-level behavior for unsigned and two’s-complement arithmetic. Some circumstances require different instructions to handle signed and unsigned operations, such as using different versions of right shifts, division and multiplication instructions, and different combinations of condition codes.

Practice Problem 3.13 (solution page 366)

The C code

```
int comp(data_t a, data_t b) {
    return a COMP b;
}
```

shows a general comparison between arguments `a` and `b`, where `data_t`, the data type of the arguments, is defined (via `typedef`) to be one of the integer data types listed in Figure 3.1 and either signed or unsigned. The comparison `COMP` is defined via `#define`.

Suppose `a` is in some portion of `%rdx` while `b` is in some portion of `%rsi`. For each of the following instruction sequences, determine which data types `data_t` and which comparisons `COMP` could cause the compiler to generate this code. (There can be multiple correct answers; you should list them all.)

- A. `cmpl %esi, %edi`
 `setl %al`

- B. `cmpw %si, %di`
 `setge %al`

- C. `cmpb %sil, %dil`
 `setbe %al`

 - D. `cmpq %rsi, %rdi`
 `setne %a`
-

Practice Problem 3.14 (solution page 366)

The C code

```
int test(data_t a) {
    return a TEST 0;
}
```

shows a general comparison between argument `a` and 0, where we can set the data type of the argument by declaring `data_t` with a `typedef`, and the nature of the comparison by declaring `TEST` with a `#define` declaration. The following instruction sequences implement the comparison, where `a` is held in some portion of register `%rdi`. For each sequence, determine which data types `data_t` and which comparisons `TEST` could cause the compiler to generate this code. (There can be multiple correct answers; list all correct ones.)

- A. `testq %rdi, %rdi`
 `setge %al`

 - B. `testw %di, %di`
 `sete %al`


 - C. `testb %dil, %dil`
 `seta %al`

 - D. `testl %edi, %edi`
 `setle %al`
-

3.6.3 Jump Instructions

Under normal execution, instructions follow each other in the order they are listed. A *jump* instruction can cause the execution to switch to a completely new position in the program. These jump destinations are generally indicated in assembly code by a *label*. Consider the following (very contrived) assembly-code sequence:

<code>movq \$0,%rax</code>	<i>Set %rax to 0</i>
<code>jmp .L1</code>	<i>Goto .L1</i>
<code>movq (%rax),%rdx</code>	<i>Null pointer dereference (skipped)</i>
<code>.L1:</code>	
<code>popq %rdx</code>	<i>Jump target</i>



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

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.

The instruction `jmp .L1` will cause the program to skip over the `movq` instruction and instead resume execution with the `popq` instruction. In generating the object-code file, the assembler determines the addresses of all labeled instructions and encodes the *jump targets* (the addresses of the destination instructions) as part of the jump instructions.

Figure 3.15 shows the different jump instructions. The `jmp` instruction jumps unconditionally. It can be either a *direct* jump, where the jump target is encoded as part of the instruction, or an *indirect* jump, where the jump target is read from a register or a memory location. Direct jumps are written in assembly code by giving a label as the jump target, for example, the label `.L1` in the code shown. Indirect jumps are written using ‘`*`’ followed by an operand specifier using one of the memory operand formats described in Figure 3.3. As examples, the instruction

```
jmp %rax
```

uses the value in register `%rax` as the jump target, and the instruction

```
jmp *(%rax)
```

reads the jump target from memory, using the value in `%rax` as the read address.

The remaining jump instructions in the table are *conditional*—they either jump or continue executing at the next instruction in the code sequence, depending on some combination of the condition codes. The names of these instructions

and the conditions under which they jump match those of the `SET` instructions (see Figure 3.14). As with the `SET` instructions, some of the underlying machine instructions have multiple names. Conditional jumps can only be direct.

3.6.4 Jump Instruction Encodings

For the most part, we will not concern ourselves with the detailed format of machine code. On the other hand, understanding how the targets of jump instructions are encoded will become important when we study linking in Chapter 7. In addition, it helps when interpreting the output of a disassembler. In assembly code, jump targets are written using symbolic labels. The assembler, and later the linker, generate the proper encodings of the jump targets. There are several different encodings for jumps, but some of the most commonly used ones are *PC relative*. That is, they encode the difference between the address of the target instruction and the address of the instruction immediately following the jump. These offsets can be encoded using 1, 2, or 4 bytes. A second encoding method is to give an “absolute” address, using 4 bytes to directly specify the target. The assembler and linker select the appropriate encodings of the jump destinations.

As an example of PC-relative addressing, the following assembly code for a function was generated by compiling a file `branch.c`. It contains two jumps: the `jmp` instruction on line 2 jumps forward to a higher address, while the `jg` instruction on line 7 jumps back to a lower one.

```
1    movq    %rdi, %rax
2    jmp     .L2
3    .L3:
4    sarq    %rax
5    .L2:
6    testq   %rax, %rax
7    jg      .L3
8    rep; ret
```

The disassembled version of the `.o` format generated by the assembler is as follows:

1	0:	48 89 f8	mov	%rdi,%rax
2	3:	eb 03	jmp	8 <loop+0x8>
3	5:	48 d1 f8	sar	%rax
4	8:	48 85 c0	test	%rax,%rax
5	b:	7f f8	jg	5 <loop+0x5>
6	d:	f3 c3	repz retq	

In the annotations on the right generated by the disassembler, the jump targets are indicated as `0x8` for the jump instruction on line 2 and `0x5` for the jump instruction on line 5 (the disassembler lists all numbers in hexadecimal). Looking at the byte encodings of the instructions, however, we see that the target of the first jump instruction is encoded (in the second byte) as `0x03`. Adding this to `0x5`, the

Aside What do the instructions `rep` and `repz` do?

Line 8 of the assembly code shown on page 243 contains the instruction combination `rep; ret`. These are rendered in the disassembled code (line 6) as `repz retq`. One can infer that `repz` is a synonym for `rep`, just as `retq` is a synonym for `ret`. Looking at the Intel and AMD documentation for the `rep` instruction, we find that it is normally used to implement a repeating string operation [3, 51]. It seems completely inappropriate here. The answer to this puzzle can be seen in AMD’s guidelines to compiler writers [1]. They recommend using the combination of `rep` followed by `ret` to avoid making the `ret` instruction the destination of a conditional jump instruction. Without the `rep` instruction, the `jg` instruction (line 7 of the assembly code) would proceed to the `ret` instruction when the branch is not taken. According to AMD, their processors cannot properly predict the destination of a `ret` instruction when it is reached from a jump instruction. The `rep` instruction serves as a form of no-operation here, and so inserting it as the jump destination does not change behavior of the code, except to make it faster on AMD processors. We can safely ignore any `rep` or `repz` instruction we see in the rest of the code presented in this book.

address of the following instruction, we get jump target address `0x8`, the address of the instruction on line 4.

Similarly, the target of the second jump instruction is encoded as `0xf8` (decimal `-8`) using a single-byte two’s-complement representation. Adding this to `0xd` (decimal 13), the address of the instruction on line 6, we get `0x5`, the address of the instruction on line 3.

As these examples illustrate, the value of the program counter when performing PC-relative addressing is the address of the instruction following the jump, not that of the jump itself. This convention dates back to early implementations, when the processor would update the program counter as its first step in executing an instruction.

The following shows the disassembled version of the program after linking:

1	4004d0:	48 89 f8	<code>mov</code>	<code>%rdi,%rax</code>
2	4004d3:	eb 03	<code>jmp</code>	<code>4004d8 <loop+0x8></code>
3	4004d5:	48 d1 f8	<code>sar</code>	<code>%rax</code>
4	4004d8:	48 85 c0	<code>test</code>	<code>%rax,%rax</code>
5	4004db:	7f f8	<code>jg</code>	<code>4004d5 <loop+0x5></code>
6	4004dd:	f3 c3	<code>repz</code>	<code>retq</code>

The instructions have been relocated to different addresses, but the encodings of the jump targets in lines 2 and 5 remain unchanged. By using a PC-relative encoding of the jump targets, the instructions can be compactly encoded (requiring just 2 bytes), and the object code can be shifted to different positions in memory without alteration.