Ejercicios Tema 2

```
    void multstore(long x, long y, long *dest)
    x in %rdi, y in %rsi, dest in %rdx
1   multstore:
2     pushq   %rbx              Save %rbx
3     movq    %rdx, %rbx        Copy dest to %rbx
4     call    mult2             Call mult2(x, y)
5     movq    %rax, (%rbx)      Store result at *dest
6     popq    %rbx              Restore %rbx
7     ret                       Return
```

```
long mult2(long, long);

void multstore(long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

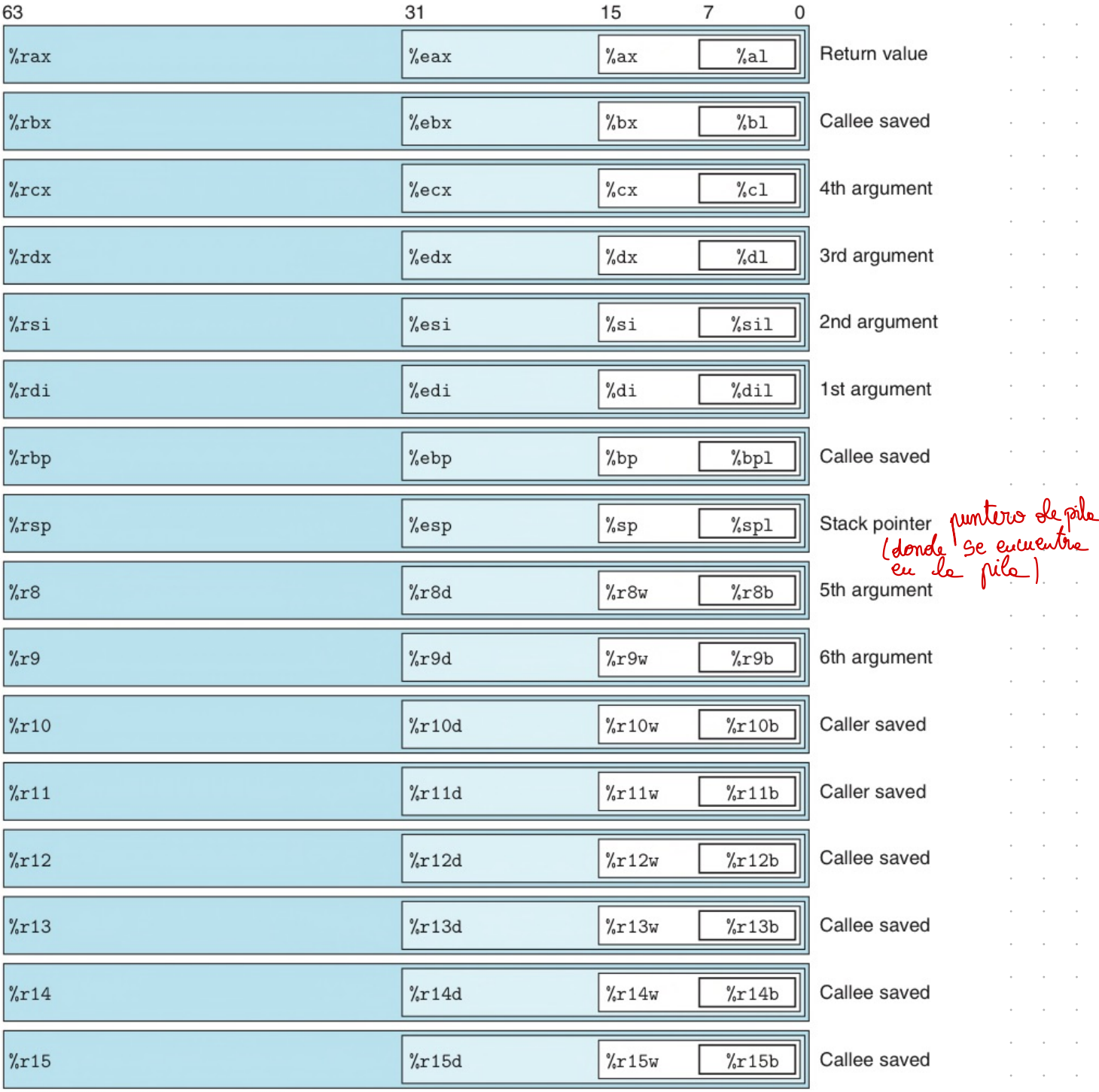


Figure 3.2 **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

(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.

Practice Problem 3.1 (solution page 361)

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

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

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

Operand	Value
%rax	0x100
0x104	0xAB
$0x108	0x108
(%rax)	0xFF
4(%rax)	0xAB
9(%rax,%rdx)	0x11
260(%rcx,%rdx)	0x13
0xFC(,%rcx,4)	0xff
(%rax,%rdx,4)	0x11

4 + 100 = 104

260 = 104 16

FC 12 + 4 = 100

104 + 4 = 108

Solution to Problem 3.1 (page 218)

This exercise gives you practice with the different operand forms.

Operand	Value	Comment
%rax	0x100	Register
0x104	0xAB	Absolute address
$0x108	0x108	Immediate
(%rax)	0xFF	Address 0x100
4(%rax)	0xAB	Address 0x104
9(%rax,%rdx)	0x11	Address 0x10C
260(%rcx,%rdx)	0x13	Address 0x108
0xFC(,%rcx,4)	0xFF	Address 0x100
(%rax,%rdx,4)	0x11	Address 0x10C

Practice Problem 3.2 (solution page 361)

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

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

→ La sol. la he hecho según la teoría. Pero en la sol. no sale lo mismo ??

Solution to Problem 3.2 (page 221)

As we have seen, the assembly code generated by GCC includes suffixes on the instructions, while the disassembler does not. Being able to switch between these two forms is an important skill to learn. One important feature is that memory references in x86-64 are always given with quad word registers, such as `%rax`, even if the operand is a byte, single word, or double word.

Here is the code written with suffixes:

```
movl   %eax, (%rsp)
movw   (%rax), %dx
movb   $0xFF, %bl
movb   (%rsp,%rdx,4), %dl
movq  (%rdx), %rax
movw   %dx, (%rax)
```

↓ Esto es la parte teórica que nos permite resolver el ej. Hay que sabérselo de Memoria

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

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

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

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

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)
```

Solution to Problem 3.3 (page 222)

Since we will rely on GCC to generate most of our assembly code, being able to write correct assembly code is not a critical skill. Nonetheless, this exercise will help you become more familiar with the different instruction and operand types.

Here is the code with explanations of the errors:

Source Destination

```
movb $0xF, (%ebx)        Cannot use %ebx as address register
movl %rax, (%rsp)        Mismatch between instruction suffix and register ID
movw (%rax),4(%rsp)      Cannot have both source and destination be memory references
movb %al,%sl             No register named %sl
movl %eax,$0x123         Cannot have immediate as destination
movl %eax,%dx            Destination operand incorrect size
movb %si, 8(%rbp)        Mismatch between instruction suffix and register ID
```

Es verdad, si nos fijamos en la tabla de registros eBx no aparece

q

w

q

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;
```

Solution to Problem 3.4 (page 223)

This exercise gives you more experience with the different data movement instructions and how they relate to the data types and conversion rules of C. The nuances of conversions of both signedness and size, as well as integral promotion, add challenge to this problem.

src_t	dest_t	Instruction	Comments
long	long	movq (%rdi), %rax	Read 8 bytes
		movq %rax, (%rsi)	Store 8 bytes
char	int	movsbl (%rdi), %eax	Convert char to int
		movl %eax, (%rsi)	Store 4 bytes
char	unsigned	movsbl (%rdi), %eax	Convert char to int
		movl %eax, (%rsi)	Store 4 bytes
unsigned char	long	movzbl (%rdi), %eax	Read byte and zero-extend
		movq %rax, (%rsi)	Store 8 bytes
int	char	movl (%rdi), %eax	Read 4 bytes
		movb %al, (%rsi)	Store low-order byte
unsigned	unsigned	movl (%rdi), %eax	Read 4 bytes
	char	movb %al, (%rsi)	Store low-order byte
char	short	movsbw (%rdi), %ax	Read byte and sign-extend
		movw %ax, (%rsi)	Store 2 bytes

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

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

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.

Solution to Problem 3.5 (page 225)

Reverse engineering is a good way to understand systems. In this case, we want to reverse the effect of the C compiler to determine what C code gave rise to this assembly code. The best way is to run a "simulation," starting with values x, y, and z at the locations designated by pointers xp, yp, and zp, respectively. We would then get the following behavior:

```
  void decode1(long *xp, long *yp, long *zp)
  xp in %rdi, yp in %rsi, zp in %rdx
decode1:
  movq    (%rdi), %r8     Get x = *xp
  movq    (%rsi), %rcx    Get y = *yp
  movq    (%rdx), %rax    Get z = *zp
  movq    %r8, (%rsi)     Store x at yp
  movq    %rcx, (%rdx)    Store y at zp
  movq    %rax, (%rdi)    Store z at xp
  ret
```

From this, we can generate the following C code:

```
void decode1(long *xp, long *yp, long *zp)
{
    long x = *xp;
    long y = *yp;
    long z = *zp;

    *yp = x;
    *zp = y;
    *xp = z;
}
```

PUSH Y POP

Para realizar este procedimiento usamos una pila. Para añadir datos lo hacemos con la orden push y para eliminar datos se hace con la orden pop.

Instruction	Effect	Description
pushq *S*	R[**%rsp**] ← R[**%rsp**] − 8; M[R[**%rsp**]] ← *S*	Push quad word
popq *D*	*D* ← M[R[**%rsp**]]; R[**%rsp**] ← R[**%rsp**] + 8	Pop quad word

El puntero pila se guarda en %rsp.

Poner un valor en la pila, implica disminuir el puntero en 8 y luego escribir el valor en memoria.

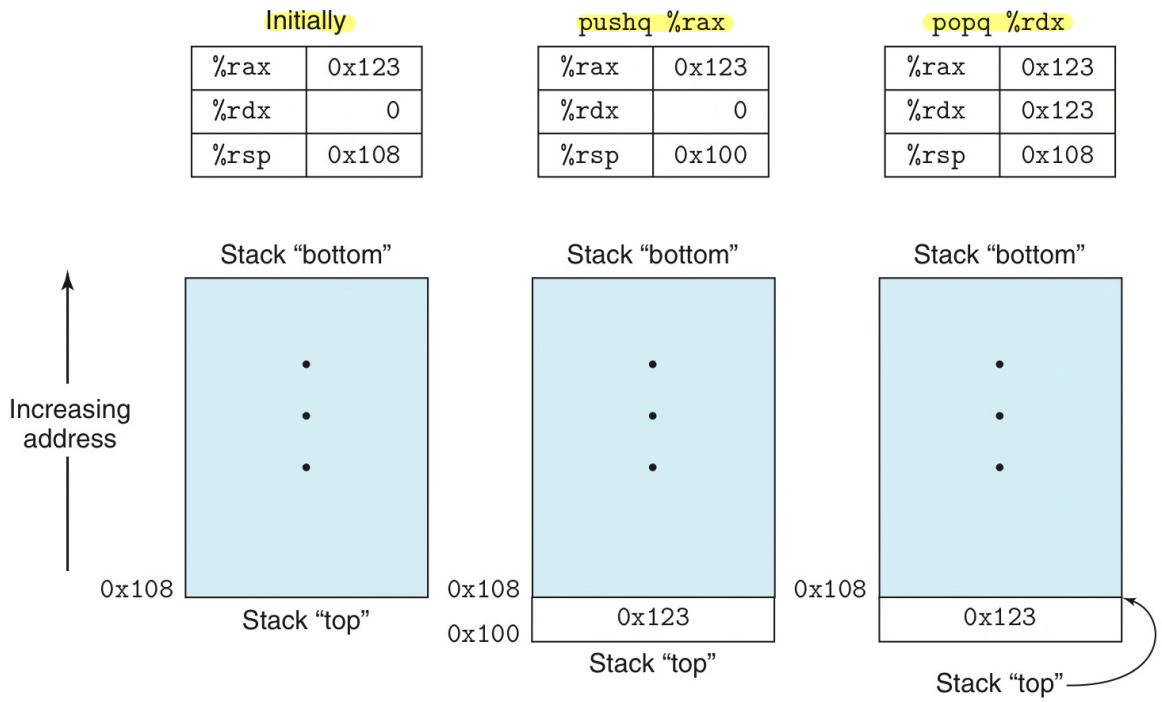
Ejemplo:

Hacer la instrucción pushq %rbp equivale a subq $8, %rsp
movq %rbp, (%rsp)

La **diferencia** es esto es que la instrucción push usa 1 byte mientras que subq y movq juntas requieren 8 bytes.

La instrucción para quitar un elemento de la pila consiste en un incremento del puntero de pila en 8, entonces:

Hacer la instrucción popq %rbp equivale a movq (%rsp), %rax // lee %rax de la pila
addq $8, %rsp



Algunas Operaciones Aritméticas

- **Instrucciones de Dos Operandos:**

todas las operaciones modifican los flags de estado.

Formato		*Operación*†	
`addq`	*Src,Dest*	Dest = Dest + Src	
`subq`	*Src,Dest*	Dest = Dest – Src	
`imulq`	*Src,Dest*	Dest = Dest * Src	
`salq`	*Src,Dest*	Dest = Dest << Src	*También llamada shlq*
`sarq`	*Src,Dest*	Dest = Dest >> Src	*Aritméticas*
`shrq`	*Src,Dest*	Dest = Dest >> Src	*Lógicas*
`xorq`	*Src,Dest*	Dest = Dest ^ Src	
`andq`	*Src,Dest*	Dest = Dest & Src	
`orq`	*Src,Dest*	Dest = Dest | Src	

- **¡Cuidado con el orden de los argumentos!** (Intel vs. AT&T)
- **No se distingue entre enteros con/sin signo (*¿por qué?*)**

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	$D \leftarrow \sim D$	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	$D \leftarrow D \hat{} S$	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

INPUT		OUTPUT
A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

INPUT		OUTPUT
A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

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.

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:

%rbx = p %rdx = q

Instruction	Result
`leaq 9(%rdx), %rax`	$9+q$
`leaq (%rdx,%rbx), %rax`	$p+q$
`leaq (%rdx,%rbx,3), %rax`	$3p+q$
`leaq 2(%rbx,%rbx,7), %rax`	$8p+2$
`leaq 0xE(,%rdx,3), %rax`	$(3q)+14$
`leaq 6(%rbx,%rdx,7), %rax`	$(7p+p)\ 6$

Solution to Problem 3.6 (page 228)

This exercise demonstrates the versatility of the `leaq` instruction and gives you more practice in deciphering the different operand forms. Although the operand forms are classified as type "Memory" in Figure 3.3, no memory access occurs.

Instruction	Result
`leaq 9(%rdx), %rax`	$9+q$
`leaq (%rdx,%rbx), %rax`	$q+p$
`leaq (%rdx,%rbx,3), %rax`	$q+3p$
`leaq 2(%rbx,%rbx,7), %rax`	$2+8p$
`leaq 0xE(,%rdx,3), %rax`	$14+3q$
`leaq 6(%rbx,%rdx,7), %rdx`	$6+p+7q$

Ejemplo del uso de leaq

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 = z + y*(x+10) ;
  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
```

valor a guardar

%rbx = t = 9y + y = 10y

t = 10y = z + 10y

t = y*x + z + 10y = z + y(x+10)

Fill in the missing expression in the C code.

Solution to Problem 3.7 (page 229)

Again, reverse engineering proves to be a useful way to learn the relationship between C code and the generated assembly code.

The best way to solve problems of this type is to annotate the lines of assembly code with information about the operations being performed. Here is a sample:

```
  short scale3(short x, short y, short z)
  x in %rdi, y in %rsi, z in %rdx
scale3:
  leaq    (%rsi,%rsi,9), %rbx       10 * y
  leaq    (%rbx,%rdx), %rbx         10 * y + z
  leaq    (%rbx,%rdi,%rsi), %rbx    10 * y + z + y * x
  ret
```

From this, it is easy to generate the missing expression:

```
short t = 10 * y + z + y * x;
```

TABLA DE CONVERSIÓN DE BASES

(binario, octal, decimal, hexadecimal y otras)

Binario	B3	B4	B5	B6	B7	Oct	B9	Dec	B11	B12	B13	B14	B15	Hex
1	1	1	1	1	1	1	1	**1**	1	1	1	1	1	1
10	2	2	2	2	2	2	2	**2**	2	2	2	2	2	2
11	10	3	3	3	3	3	3	**3**	3	3	3	3	3	3
100	11	10	4	4	4	4	4	**4**	4	4	4	4	4	4
101	12	11	10	5	5	5	5	**5**	5	5	5	5	5	5
110	20	12	11	10	6	6	6	**6**	6	6	6	6	6	6
111	21	13	12	11	10	7	7	**7**	7	7	7	7	7	7
1000	22	20	13	12	11	10	8	**8**	8	8	8	8	8	8
1001	100	21	14	13	12	11	10	**9**	9	9	9	9	9	9
1010	101	22	20	14	13	12	11	**10**	A	A	A	A	A	A
1011	102	23	21	15	14	13	12	**11**	10	B	B	B	B	B
1100	110	30	22	20	15	14	13	**12**	11	10	C	C	C	C
1101	111	31	23	21	16	15	14	**13**	12	11	10	D	D	D
1110	112	32	24	22	20	16	15	**14**	13	12	11	10	E	E
1111	120	33	30	23	21	17	16	**15**	14	13	12	11	10	F
10000	121	100	31	24	22	20	17	**16**	15	14	13	12	11	10
10001	122	101	32	25	23	21	18	**17**	16	15	14	13	12	11
10010	200	102	33	30	24	22	20	**18**	17	16	15	14	13	12
10011	201	103	34	31	25	23	21	**19**	18	17	16	15	14	13
10100	202	110	40	32	26	24	22	**20**	19	18	17	16	15	14
10101	210	111	41	33	30	25	23	**21**	1A	19	18	17	16	15
10110	211	112	42	34	31	26	24	**22**	20	1A	19	18	17	16
10111	212	113	43	35	32	27	25	**23**	21	1B	1A	19	18	17
11000	220	120	44	40	33	30	26	**24**	22	20	1B	1A	19	18
11001	221	121	100	41	34	31	27	**25**	23	21	1C	1B	1A	19
11010	222	122	101	42	35	32	28	**26**	24	22	20	1C	1B	1A
11011	1000	123	102	43	36	33	30	**27**	25	23	21	1D	1C	1B
11100	1001	130	103	44	40	34	31	**28**	26	24	22	20	1D	1C
11101	1002	131	104	45	41	35	32	**29**	27	25	23	21	1E	1D
11110	1010	132	110	50	42	36	33	**30**	28	26	24	22	20	1E
11111	1011	133	111	51	43	37	34	**31**	29	27	25	23	21	1F
100000	1012	200	112	52	44	40	35	**32**	2A	28	26	24	22	20
100001	1020	201	113	53	45	41	36	**33**	30	29	27	25	23	21
100010	1021	202	114	54	46	42	37	**34**	31	2A	28	26	24	22
100011	1022	203	120	55	50	43	38	**35**	32	2B	29	27	25	23
100100	1100	210	121	100	51	44	40	**36**	33	30	2A	28	26	24
100101	1101	211	122	101	52	45	41	**37**	34	31	2B	29	27	25
100110	1102	212	123	102	53	46	42	**38**	35	32	2C	2A	28	26
100111	1110	213	124	103	54	47	43	**39**	36	33	30	2B	29	27
101000	1111	220	130	104	55	50	44	**40**	37	34	31	2C	2A	28
101001	1112	221	131	105	56	51	45	**41**	38	35	32	2D	2B	29
101010	1120	222	132	110	60	52	46	**42**	39	36	33	30	2C	2A
101011	1121	223	133	111	61	53	47	**43**	3A	37	34	31	2D	2B
101100	1122	230	134	112	62	54	48	**44**	40	38	35	32	2E	2C
101101	1200	231	140	113	63	55	50	**45**	41	39	36	33	30	2D
101110	1201	232	141	114	64	56	51	**46**	42	3A	37	34	31	2E
101111	1202	233	142	115	65	57	52	**47**	43	3B	38	35	32	2F

Pasar de hex → dec.

Hexadecimal: F 1 2 A 4

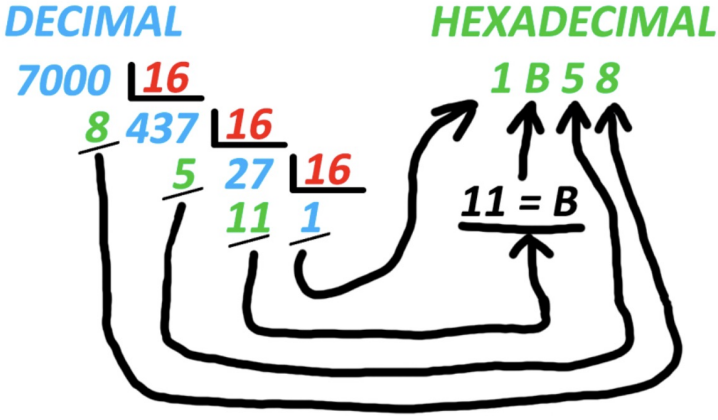
Equivalente decimal: 15 1 2 10 4

Potencias: 16^4 16^3 16^2 16^1 16^0

Resultado: 983040 + 4096 + 512 +160 + 4

F12A4(16) = 987812(10)

Pasar del dec. → hex.



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
① `addq %rcx,(%rax)`	0x100	0x100
② `subq %rdx,8(%rax)`	0x108	0xA8
③ `imulq $16,(%rax,%rdx,8)`	0x118	0x110
`incq 16(%rax)`		
`decq %rcx`		
`subq %rdx,%rax`		

Solution to Problem 3.8 (page 230)

This problem gives you a chance to test your understanding of operands and the arithmetic instructions. The instruction sequence is designed so that the result of each instruction does not affect the behavior of subsequent ones.

Instruction	Destination	Value
`addq %rcx,(%rax)`	0x100	0x100
`subq %rdx,8(%rax)`	0x108	0xA8
`imulq $16,(%rax,%rdx,8)`	0x118	0x110
`incq 16(%rax)`	0x110	0x14
`decq %rcx`	%rcx	0x0
`subq %rdx,%rax`	%rax	0xFD

① 0xff + 0x1 = ff (=15) + 1 = 100 ; $16_{10} = 10_{16}$

② 8(%rax) = 8 + 0xff = ff (=15) + 8 = 113 → = 8 + 0x100 = 0x108 → 0xA8

0xAB − 0x3 = AB (=11) − 3 = A8

③ (%rax, %rdx, 8) = (8 × 3)$_{10}$ + 0x100 = 24$_{10}$ + 0x100 = 0x18 + 0x100 = 0x118

(16) x (0x11) = 16 × (16^1 + 16^0) = 16 × 17 = 272$_{10}$ → 272 | 16 → 0, 17 | 16 → 1, 1 | 16 → 1 → sol: 0x110

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
① addq %rcx,(%rax)	0x100	0x100
② subq %rdx,8(%rax)	0x108	0xA8
③ imulq $16,(%rax,%rdx,8)	0x118	0x110
④ incq 16(%rax)	0x110	0x14
decq %rcx	%rcx	0x0
⑥ subq %rdx,%rax	%rax	0xFD

④ 0x100 + 16 = 0x100 + 0x010 = 0x110 → y el valor que tiene lo suma 1: 0x13 + 1 = 0x14

$16_{10} = 10_{16}$

⑥ 0x100 − 0x003 } 0x100 = 16^2 = 256 / = 0x3 = 3 } ⇒ 256 − 3 = 253

253 | 16 → 13 | 15 → FD

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;
}
```

salq	*Src,Dest*	Dest = Dest << Src	*También llamada shlq*
sarq	*Src,Dest*	Dest = Dest >> Src	*Aritméticas*
shrq	*Src,Dest*	Dest = Dest >> Src	*Lógicas*

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
  salq    $4 ; %rax      x <<= 4
  movl    %esi, %ecx     Get n (4 bytes)
  sarq    $n ; %rax      x >>= n
```

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

Solution to Problem 3.9 (page 231)

This exercise gives you a chance to generate a little bit of assembly code. The solution code was generated by GCC. By loading parameter n in register %ecx, it can then use byte register %cl to specify the shift amount for the sarq instruction. It might seem odd to use a movl instruction, given that n is eight bytes long, but keep in mind that only the least significant byte is required to specify the shift amount.

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

(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
```

Se comporta como 2^4

Figure 3.11 C and assembly code for arithmetic function.

La figura 3.11 muestra un ejemplo de una función que realiza operaciones aritméticas y su traducción a código ensamblador. Los argumentos x, y y z se almacenan inicialmente en los registros% rdi,% rsi y% rdx, respectivamente. Las instrucciones del código ensamblador se corresponden estrechamente con las líneas del código fuente de C. La línea 2 calcula el valor de x ^ y. Las líneas 3 y 4 calculan la expresión z * 48 mediante una combinación de instrucciones leaq y shift. La línea 5 calcula el y de t1 y 0x0F0F0F0F. La resta final se calcula con la línea 6. Dado que el destino de la resta es el registro% rax, este será el valor devuelto por la función.
En el código ensamblador de la figura 3.11, la secuencia de valores en el registro% rax corresponde a los valores de programa 3 * z, z * 48 y t4 (como valor de retorno). En general, los compiladores generan código que usa registros individuales para múltiples valores de programa y mueve los valores de programa entre los registros.

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.

Solution to Problem 3.10 (page 232)

This problem is fairly straightforward, since the assembly code follows the structure of the C code closely.

```
short p1 = y | z;
short p2 = p1 >> 9;
short p3 = ~p2;
short p4 = y - p3;
```

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.

Solution to Problem 3.11 (page 233)

A. This instruction is used to set register `%rcx` to zero, exploiting the property that $x \hat{} x = 0$ for any x. It corresponds to the C statement `x = 0`.

B. A more direct way of setting register `%rcx` to zero is with the instruction `movq $0,%rcx`.

C. Assembling and disassembling this code, however, we find that the version with `xorq` requires only 3 bytes, while the version with `movq` requires 7. Other ways to set `%rcx` to zero rely on the property that any instruction that updates the lower 4 bytes will cause the high-order bytes to be set to zero. Thus, we could use either `xorl %ecx,%ecx` (2 bytes) or `movl $0,%ecx` (5 bytes).

Explique el efecto de esta instrucción o exclusiva exclusiva y la operación útil que implementa.
B. ¿Cuál sería la forma más sencilla de expresar esta operación en código ensamblador?
C. Compare el número de bytes para codificar dos de estas tres implementaciones diferentes de la misma operación.

Esta instrucción se usa para establecer el registro% rcx en cero, explotando la propiedad de que x ^ x = 0 para cualquier x. Corresponde a la declaración de C x = 0.
B. Una forma más directa de establecer el registro% rcx en cero es con la instrucción movq $ 0,% rcx.
C. Sin embargo, al ensamblar y desensamblar este disco, encontramos que la versión con xorq requiere solo 3 bytes, mientras que la versión con movq requiere 7. Otras formas de establecer% rcx en cero se basan en la propiedad de que cualquier instrucción que actualice los 4 bytes inferiores provocará los bytes de orden superior. para ser puesto a cero. Por lo tanto, podríamos usar xorl% ecx,% ecx (2 bytes) o movl $ 0,% ecx (5 bytes).

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.

Solution to Problem 3.12 (page 236)

We can simply replace the `cqto` instruction with one that sets register `%rdx` to zero, and use `divq` rather than `idivq` as our division instruction, yielding the following code:

```
    void uremdiv(unsigned long x, unsigned long y,
                 unsigned long *qp, unsigned long *rp)
    x in %rdi, y in %rsi, qp in %rdx, rp in %rcx
1   uremdiv:
2     movq    %rdx, %r8       Copy qp
3     movq    %rdi, %rax      Move x to lower 8 bytes of dividend
4     movl    $0, %edx        Set upper 8 bytes of dividend to 0
5     divq    %rsi            Divide by y
6     movq    %rax, (%r8)     Store quotient at qp
7     movq    %rdx, (%rcx)    Store remainder at rp
8     ret
```

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.

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
```

Solution to Problem 3.13 (page 240)

It is important to understand that assembly code does not keep track of the type of a program value. Instead, the different instructions determine the operand sizes and whether they are signed or unsigned. When mapping from instruction sequences back to C code, we must do a bit of detective work to infer the data types of the program values.

A. The suffix '`l`' and the register identifiers indicate 32-bit operands, while the comparison is for a two's-complement `<`. We can infer that `data_t` must be `int`.

B. The suffix '`w`' and the register identifiers indicate 16-bit operands, while the comparison is for a two's-complement `>=`. We can infer that `data_t` must be `short`.

C. The suffix '`b`' and the register identifiers indicate 8-bit operands, while the comparison is for an unsigned `<=`. We can infer that `data_t` must be `unsigned char`.

D. The suffix '`q`' and the register identifiers indicate 64-bit operands, while the comparison is for `!=`, which is the same whether the arguments are signed, unsigned, or pointers. We can infer that `data_t` could be either `long`, `unsigned long`, or some form of pointer.

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
```

Solution to Problem 3.14 (page 241)

This problem is similar to Problem 3.13, except that it involves TEST instructions rather than CMP instructions.

A. The suffix '`q`' and the register identifiers indicate a 64-bit operand, while the comparison is for >=, which must be signed. We can infer that `data_t` must be `long`.

B. The suffix '`w`' and the register identifier indicate a 16-bit operand, while the comparison is for ==, which is the same for signed or unsigned. We can infer that `data_t` must be either `short` or `unsigned short`.

C. The suffix '`b`' and the register identifier indicate an 8-bit operand, while the comparison is for unsigned >. We can infer that `data_t` must be `unsigned char`.

D. The suffix '`l`' and the register identifier indicate 32-bit operands, while the comparison is for <. We can infer that `data_t` must be `int`.

Practice Problem 3.15 (solution page 366)

In the following excerpts from a disassembled binary, some of the information has been replaced by X's. Answer the following questions about these instructions.

A. What is the target of the `je` instruction below? (You do not need to know anything about the `callq` instruction here.)

```
4003fa: 74 02                   je     XXXXXX
4003fc: ff d0                   callq  *%rax
```

B. What is the target of the `je` instruction below?

```
40042f: 74 f4                   je     XXXXXX
400431: 5d                      pop    %rbp
```

C. What is the address of the `ja` and `pop` instructions?

```
XXXXXX: 77 02                   ja     400547
XXXXXX: 5d                      pop    %rbp
```

D. In the code that follows, the jump target is encoded in PC-relative form as a 4-byte two's-complement number. The bytes are listed from least significant to most, reflecting the little-endian byte ordering of x86-64. What is the address of the jump target?

```
4005e8: e9 73 ff ff ff          jmpq   XXXXXXX
4005ed: 90                      nop
```

Solution to Problem 3.15 (page 245)

This exercise requires you to examine disassembled code in detail and reason about the encodings for jump targets. It also gives you practice in hexadecimal arithmetic.

A. The `je` instruction has as its target `0x4003fc` + `0x02`. As the original disassembled code shows, this is `0x4003fe`:

```
4003fa: 74 02                   je     4003fe
4003fc: ff d0                   callq  *%rax
```

B. The `je` instruction has as its target `0x0x400431` − 12 (since `0xf4` is the 1-byte two's-complement representation of −12). As the original disassembled code shows, this is `0x400425`:

```
40042f: 74 f4                   je     400425
400431: 5d                      pop    %rbp
```

C. According to the annotation produced by the disassembler, the jump target is at absolute address `0x400547`. According to the byte encoding, this must be at an address `0x2` bytes beyond that of the `pop` instruction. Subtracting these gives address `0x400545`. Noting that the encoding of the `ja` instruction requires 2 bytes, it must be located at address `0x400543`. These are confirmed by examining the original disassembly:

```
400543: 77 02                   ja     400547
400545: 5d                      pop    %rbp
```

D. Reading the bytes in reverse order, we can see that the target offset is `0xffffff73`, or decimal −141. Adding this to `0x0x4005ed` (the address of the `nop` instruction) gives address `0x400560`:

```
4005e8: e9 73 ff ff ff          jmpq   400560
4005ed: 90                      nop
```

Practice Problem 3.16 (solution page 367)

When given the C code

```
void cond(short a, short *p)
{
    if (a && *p < a)
        *p = a;
}
```

GCC generates the following assembly code:

```
  void cond(short a, short *p)
  a in %rdi, p in %rsi
cond:
  testq   %rdi, %rdi
  je      .L1
  cmpq    %rsi, (%rdi)
  jle     .L1
  movq    %rdi, (%rsi)
.L1:
  rep; ret
```

A. Write a goto version in C that performs the same computation and mimics the control flow of the assembly code, in the style shown in Figure 3.16(b). You might find it helpful to first annotate the assembly code as we have done in our examples.

B. Explain why the assembly code contains two conditional branches, even though the C code has only one `if` statement.

Solution to Problem 3.16 (page 248)

Annotating assembly code and writing C code that mimics its control flow are good first steps in understanding assembly-language programs. This problem gives you practice for an example with simple control flow. It also gives you a chance to examine the implementation of logical operations.

A. Here is the C code:

```
void goto_cond(short a, short *p) {
    if (a == 0)
        goto done;
    if (a >= *p)
        goto done;
    *p = a;
 done:
    return;
}
```

B. The first conditional branch is part of the implementation of the `&&` expression. If the test for `a` being non-null fails, the code will skip the test of `a >= *p`.

Practice Problem 3.17 (solution page 367)

An alternate rule for translating `if` statements into goto code is as follows:

```
    t = test-expr;
    if (t)
        goto true;
    else-statement
    goto done;
true:
    then-statement
done:
```

A. Rewrite the goto version of `absdiff_se` based on this alternate rule.

B. Can you think of any reasons for choosing one rule over the other?

Solution to Problem 3.17 (page 248)

This is an exercise to help you think about the idea of a general translation rule and how to apply it.

A. Converting to this alternate form involves only switching around a few lines of the code:

```
long gotodiff_se_alt(long x, long y) {
    long result;
    if (x < y)
        goto x_lt_y;
    ge_cnt++;
    result = x - y;
    return result;
 x_lt_y:
    lt_cnt++;
    result =  y - x;
    return result;
}
```

B. In most respects, the choice is arbitrary. But the original rule works better for the common case where there is no `else` statement. For this case, we can simply modify the translation rule to be as follows:

```
    t = test-expr;
    if (!t)
        goto done;
    then-statement
done:
```

A translation based on the alternate rule is more cumbersome.

Practice Problem 3.18 (solution page 368)

Starting with C code of the form

```
short test(short x, short y, short z) {
    short val = _________;
    if (_________) {
        if (_________)
            val = _________;
        else
            val = _________;
    } else if (_________)
        val = _________;
    return val;
}
```

GCC generates the following assembly code:

```
  short test(short x, short y, short z)
  x in %rdi, y in %rsi, z in %rdx
test:
  leaq    (%rdx,%rsi), %rax
  subq    %rdi, %rax
  cmpq    $5, %rdx
  jle     .L2
  cmpq    $2, %rsi
  jle     .L3
  movq    %rdi, %rax
  idivq   %rdx, %rax
  ret
.L3:
  movq    %rdi, %rax
  idivq   %rsi, %rax
  ret
.L2:
  cmpq    $3, %rdx
  jge     .L4
  movq    %rdx, %rax
  idivq   %rsi, %rax
.L4:
  rep; ret
```

Fill in the missing expressions in the C code.

Solution to Problem 3.18 (page 249)

This problem requires that you work through a nested branch structure, where you will see how our rule for translating `if` statements has been applied. On the whole, the machine code is a straightforward translation of the C code.

```
short test(short x, short y, short z) {
    short val = z+y-x;
    if (z > 5) {
        if (y > 2)
            val = x/z;
        else
            val = x/y;
    } else if (z < 3)
        val = z/y;
    return val;
}
```

Practice Problem 3.19 (solution page 368)

Running on a new processor model, our code required around 45 cycles when the branching pattern was random, and around 25 cycles when the pattern was highly predictable.

A. What is the approximate miss penalty?

B. How many cycles would the function require when the branch is mispredicted?

Solution to Problem 3.19 (page 252)

This problem reinforces our method of computing the misprediction penalty.

A. We can apply our formula directly to get $T_{\mathrm{MP}} = 2(45 - 25) = 40$.

B. When misprediction occurs, the function will require around $25 + 40 = 65$ cycles.

Practice Problem 3.20 (solution page 369)

In the following C function, we have left the definition of operation `OP` incomplete:

```
#define OP _____ /* Unknown operator */

short arith(short x) {
    return x OP 16;
}
```

When compiled, GCC generates the following assembly code:

```
  short arith(short x)
  x in %rdi
arith:
  leaq    15(%rdi), %rbx
  testq   %rdi, %rdi
  cmovns  %rdi, %rbx
  sarq    $4, %rbx
  ret
```

A. What operation is `OP`?

B. Annotate the code to explain how it works.

Solution to Problem 3.20 (page 255)

This problem provides a chance to study the use of conditional moves.

A. The operator is '/'. We see this is an example of dividing by a power of 4 by right shifting (see Section 2.3.7). Before shifting by $k = 4$, we must add a bias of $2^k - 1 = 15$ when the dividend is negative.

B. Here is an annotated version of the assembly code:

```
  short arith(short x)
  x in %rdi
arith:
  leaq    15(%rdi), %rbx    temp = x+15
  testq   %rdi, %rdi        Text x
  cmovns  %rdi, %rbx        If x>= 0, temp = x
  sarq    $4, %rbx          result = temp >> 4 (= x/16)
  ret
```

The program creates a temporary value equal to $x + 15$, in anticipation of x being negative and therefore requiring biasing. The `cmovns` instruction conditionally changes this number to x when $x \geq 0$, and then it is shifted by 4 to generate $x/16$.

Practice Problem 3.21 (solution page 369)

Starting with C code of the form

```
short test(short x, short y) {
    short val = _________;
    if (_________) {
        if (_________)
            val = _________;
        else
            val = _________;
    } else if (_________)
        val = _________;
    return val;
}
```

GCC generates the following assembly code:

```
  short test(short x, short y)
  x in %rdi, y in %rsi
test:
  leaq    12(%rsi), %rbx
  testq   %rdi, %rdi
  jge     .L2

  movq    %rdi, %rbx
  imulq   %rsi, %rbx
  movq    %rdi, %rdx
  orq     %rsi, %rdx
  cmpq    %rsi, %rdi
  cmovge  %rdx, %rbx
  ret
.L2:
  idivq   %rsi, %rdi
  cmpq    $10, %rsi
  cmovge  %rdi, %rbx
  ret
```

Fill in the missing expressions in the C code.

Solution to Problem 3.21 (page 255)

This problem is similar to Problem 3.18, except that some of the conditionals have been implemented by conditional data transfers. Although it might seem daunting to fit this code into the framework of the original C code, you will find that it follows the translation rules fairly closely.

```
short test(short x, short y) {
    short val = y + 12;
    if (x < 0) {
        if (x < y)
            val = x * y;
        else
            val = x | y;
    } else if (y > 10)
        val = x / y;
    return val;
}
```

Practice Problem 3.22 (solution page 369)

A. Try to calculate 14! with a 32-bit int. Verify whether the computation of 14! overflows.

B. What if the computation is done with a 64-bit long int?

Solution to Problem 3.22 (page 257)

A. The computation of 14! would overflow with a 32-bit `int`. As we learned in Problem 2.35, when we get value x while attempting to compute $n!$, we can test for overflow by computing x/n and seeing whether it equals $(n-1)!$ (assuming that we have already ensured that the computation of $(n-1)!$ did not overflow). In this case we get $1{,}278{,}945{,}280/14 = 91353234.286$. As a second test, we can see that any factorial beyond 10! must be a multiple of 100 and therefore have zeros for the last two digits. The correct value of 14! is 87,178,291,200.

Further, we can build up a table of factorials computed through 14! with data type `int`, as shown below:

n	$n!$	OK?
1	1	Y
2	2	Y
3	6	Y
4	24	Y
5	120	Y
6	720	Y
7	5,040	Y
8	40,320	Y
9	362,880	Y
10	3,628,800	Y
11	39,916,800	Y
12	479,001,600	Y
13	1,932,053,504	N
14	1,278,945,280	N

B. Doing the computation with data type `long` lets us go up to 20!, thus the 14! computation does not overflow.

Practice Problem 3.23 (solution page 370)

For the C code

```
short dw_loop(short x) {
    short y = x/9;
    short *p = &x;
    short n = 4*x;
    do {
        x += y;
        (*p) += 5;
        n -= 2;
    } while (n > 0);
    return x;
}
```

GCC generates the following assembly code:

```
    short dw_loop(short x)
    x initially in %rdi
1   dw_loop:
2     movq    %rdi, %rbx
3     movq    %rdi, %rcx
4     idivq   $9, %rcx
5     leaq    (,%rdi,4), %rdx
6   .L2:
7     leaq    5(%rbx,%rcx), %rcx
8     subq    $1, %rdx
9     testq   %rdx, %rdx
10    jg      .L2
11    rep; ret
```

A. Which registers are used to hold program values `x`, `y`, and `n`?

B. How has the compiler eliminated the need for pointer variable `p` and the pointer dereferencing implied by the expression `(*p)+=5`?

C. Add annotations to the assembly code describing the operation of the program, similar to those shown in Figure 3.19(c).

Solution to Problem 3.23 (page 258)

The code generated when compiling loops can be tricky to analyze, because the compiler can perform many different optimizations on loop code, and because it can be difficult to match program variables with registers. This particular example demonstrates several places where the assembly code is not just a direct translation of the C code.

A. Although parameter `x` is passed to the function in register `%rdi`, we can see that the register is never referenced once the loop is entered. Instead, we can see that registers `%rbx`, `%rcx`, and `%rdx` are initialized in lines 2–5 to `x`, `x/9`, and `4*x`. We can conclude, therefore, that these registers contain the program variables.

B. The compiler determines that pointer `p` always points to `x`, and hence the expression `(*p)+=5` simply increments `x`. It combines this incrementing by 5 with the increment of `y`, via the `leaq` instruction of line 7.

C. The annotated code is as follows:

```
    short dw_loop(short x)
    x initially in %rdi
1   dw_loop:
2     movq    %rdi, %rbx              Copy x to %rbx
3     movq    %rdi, %rcx
4     idivq   $9, %rcx                Compute y = x/9
5     leaq    (,%rdi,4), %rdx         Compute n = 4*x
6   .L2:                            loop:
7     leaq    5(%rbx,%rcx), %rcx      Compute y += x + 5
8     subq    $2, %rdx                Decrement n by 2
9     testq   %rdx, %rdx              Test n
10    jg      .L2                     If > 0, goto loop
11    rep; ret                        Return
```

Practice Problem 3.24 (solution page 371)

For C code having the general form

```
short loop_while(short a, short b)
{
    short result = _________;
    while (_________) {
        result = _________;
        a = _________;
    }
    return result;
}
```

GCC, run with command-line option `-Og`, produces the following code:

```
    short loop_while(short a, short b)
    a in %rdi, b in %rsi
1   loop_while:
2     movl    $0, %eax
3     jmp     .L2
4   .L3:
5     leaq    (,%rsi,%rdi), %rdx
6     addq    %rdx, %rax
7     subq    $1, %rdi
8   .L2:
9     cmpq    %rsi, %rdi
10    jg      .L3
11    rep; ret
```

We can see that the compiler used a jump-to-middle translation, using the `jmp` instruction on line 3 to jump to the test starting with label `.L2`. Fill in the missing parts of the C code.

Solution to Problem 3.24 (page 260)

This assembly code is a fairly straightforward translation of the loop using the jump-to-middle method. The full C code is as follows:

```
short loop_while(short a, short b)
{
    short result = 0;
    while (a > b) {
        result = result + (a*b);
        a = a-1;
    }
    return result;
}
```

Practice Problem 3.25 (solution page 371)

For C code having the general form

```
long loop_while2(long a, long b)
{
    long result = _____;
    while (_____) {
        result = _____;
        b = _____;
    }
    return result;
}
```

GCC, run with command-line option `-O1`, produces the following code:

```
    a in %rdi, b in %rsi
1   loop_while2:
2     testq   %rsi, %rsi
3     jle     .L8
4     movq    %rsi, %rax
5   .L7:
6     imulq   %rdi, %rax
7     subq    %rdi, %rsi
8     testq   %rsi, %rsi
}
```

Solution to Problem 3.25 (page 262)

While the generated code does not follow the exact pattern of the guarded-do translation, we can see that it is equivalent to the following C code:

```
long loop_while2(long a, long b)
{
    long result = b;
    while (b > 0) {
        result = result * a;
        b = b-a;
    }
    return result;
}
```

We will often see cases, especially when compiling with higher levels of optimization, where GCC takes some liberties in the exact form of the code it generates, while preserving the required functionality.

Practice Problem 3.26 (solution page 372)

A function `test_one` has the following overall structure:

```
short test_one(unsigned short x) {
    short val = 1;
    while ( ... ) {
        :
        :
    }
    return ...;
}
```

The GCC C compiler generates the following assembly code:

```
     short test_one(unsigned short x)
     x in %rdi
1    test_one:
2      movl    $1, %eax
3      jmp     .L5
4    .L6:
5      xorq    %rdi, %rax
6      shrq    %rdi            Shift right by 1
7    .L5:
8      testq   %rdi, %rdi
9      jne     .L6
10     andl    $0, %eax
11     ret
```

Reverse engineer the operation of this code and then do the following:

A. Determine what loop translation method was used.

B. Use the assembly-code version to fill in the missing parts of the C code.

C. Describe in English what this function computes.

Solution to Problem 3.26 (page 264)

Being able to work backward from assembly code to C code is a prime example of reverse engineering.

A. We can see that the code uses the jump-to-middle translation, using the `jmp` instruction on line 3.

B. Here is the original C code:

```
short test_one(unsigned short x) {
    short val = 1;
    while (x) {
        val ^= x;
        x >>= 1;
    }
    return val & 0;
}
```

C. This code computes the *parity* of argument `x`. That is, it returns 1 if there is an odd number of ones in `x` and 0 if there is an even number.

Practice Problem 3.27 (solution page 372)

Write goto code for a function called `fibonacci` to print fibonacci numbers using a `while` loop. Apply the guarded-do transformation.

Solution to Problem 3.27 (page 267)

This exercise is intended to reinforce your understanding of how loops are implemented.

```
long fibonacci_gd_goto(long n)
{
    long i = 2;
    long next, first = 0, second = 1;
    if (n <= 1)
        goto done;
 loop:
    next = first + second;
    first = second; second = next;
    i++;
    if (i <= n)
        goto loop;
 done:
    return n;
}
```

Practice Problem 3.28 (solution page 372)

A function `test_two` has the following overall structure:

```
short test_two(unsigned short x) {
    short val = 0;
    short i;
    for ( ... ; ... ; ... ) {
        .
        .
        .
    }
    return val;
}
```

The GCC C compiler generates the following assembly code:

```
    test fun_b(unsigned test x)
    x in %rdi
1   test_two:
2     movl    $1, %edx
3     movl    $65, %eax
4   .L10:
5     movq    %rdi, %rcx
6     andl    $1, %ecx
7     addq    %rax, %rax
8     orq     %rcx, %rax
9     shrq    %rdi            Shift right by 1
10    addq    $1, %rdx
11    jne     .L10
12    rep; ret
```

Reverse engineer the operation of this code and then do the following:

A. Use the assembly-code version to fill in the missing parts of the C code.

B. Explain why there is neither an initial test before the loop nor an initial jump to the test portion of the loop.

C. Describe in English what this function computes.

Solution to Problem 3.28 (page 267)

This problem is trickier than Problem 3.26, since the code within the loop is more complex and the overall operation is less familiar.

A. Here is the original C code:

```
long fun_b(unsigned long x) {
    long val = 0;
    long i;

    for (i = 64; i != 0; i--) {
        val = (val << 1) | (x & 0x1);
        x >>= 1;
    }
    return val;
}
```

B. The code was generated using the guarded-do transformation, but the compiler detected that, since i is initialized to 64, it will satisfy the test $i \neq 0$, and therefore the initial test is not required.

C. This code reverses the bits in `x`, creating a mirror image. It does this by shifting the bits of `x` from left to right, and then filling these bits in as it shifts `val` from right to left.

Practice Problem 3.29 (solution page 373)

Executing a `continue` statement in C causes the program to jump to the end of the current loop iteration. The stated rule for translating a `for` loop into a `while` loop needs some refinement when dealing with `continue` statements. For example, consider the following code:

```
/* Example of for loop containing a continue statement */
/* Sum even numbers between 0 and 9 */
long sum = 0;
long i;
for (i = 0; i < 10; i++) {
    if (i & 1)
        continue;
    sum += i;
}
```

A. What would we get if we naively applied our rule for translating the `for` loop into a `while` loop? What would be wrong with this code?

B. How could you replace the `continue` statement with a `goto` statement to ensure that the `while` loop correctly duplicates the behavior of the `for` loop?

Solution to Problem 3.29 (page 268)

Our stated rule for translating a `for` loop into a `while` loop is just a bit too simplistic—this is the only aspect that requires special consideration.

A. Applying our translation rule would yield the following code:

```
/* Naive translation of for loop into while loop */
/* WARNING: This is buggy code */
long sum = 0;
long i = 0;
while (i < 10) {
    if (i & 1)
        /* This will cause an infinite loop */
        continue;
    sum += i;
    i++;
}
```

This code has an infinite loop, since the `continue` statement would prevent index variable `i` from being updated.

B. The general solution is to replace the `continue` statement with a `goto` statement that skips the rest of the loop body and goes directly to the update portion:

```
/* Correct translation of for loop into while loop */
long sum = 0;
long i = 0;
while (i < 10) {
    if (i & 1)
        goto update;
    sum += i;
update:
    i++;
}
```

Practice Problem 3.30 (solution page 374)

In the C function that follows, we have omitted the body of the `switch` statement. In the C code, the case labels did not span a contiguous range, and some cases had multiple labels.

```
void switch2(short x, short *dest) {
    short val = 0;
    switch (x) {
        .
        .   Body of switch statement omitted
        .
    }
    *dest = val;
}
```

In compiling the function, GCC generates the assembly code that follows for the initial part of the procedure, with variable `x` in `%rdi`:

```
    void switch2(short x, short *dest)
    x in %rdi
1   switch2:
2     addq    $2, %rdi
3     cmpq    $8, %rdi
4     ja      .L2
5     jmp     *.L4(,%rdi,8)
```

It generates the following code for the jump table:

```
1   .L4:
2     .quad   .L9
3     .quad   .L5
4     .quad   .L6
5     .quad   .L7
6     .quad   .L2
7     .quad   .L7
8     .quad   .L8
9     .quad   .L2
10    .quad   .L5
```

Based on this information, answer the following questions:

A. What were the values of the case labels in the `switch` statement?

B. What cases had multiple labels in the C code?

Solution to Problem 3.30 (page 272)

This problem gives you a chance to reason about the control flow of a `switch` statement. Answering the questions requires you to combine information from several places in the assembly code.

- Line 2 of the assembly code adds 2 to x to set the lower range of the cases to zero. That means that the minimum case label is -2.
- Lines 3 and 4 cause the program to jump to the default case when the adjusted case value is greater than 8. This implies that the maximum case label is $-2 + 8 = 6$.
- In the jump table, we see that the entry on lines 6 (case value 2) and 9 (case value 5) have the same destination (`.L2`) as the jump instruction on line 4, indicating the default case behavior. Thus, case labels 2 and 5 are missing in the `switch` statement body.
- In the jump table, we see that the entries on lines 3 and 10 have the same destination. These correspond to cases -1 and 6.
- In the jump table, we see that the entries on lines 5 and 7 have the same destination. These correspond to cases 1 and 3.

From this reasoning, we draw the following conclusions:

A. The case labels in the `switch` statement body have values -2, -1, 0, 1, 3, 4, and 6.

B. The case with destination `.L5` has labels -1 and 6.

C. The case with destination `.L7` has labels 1 and 3.

Practice Problem 3.31 (solution page 374)

For a C function `switcher` with the general structure

```
void switcher(long a, long b, long c, long *dest)
{
    long val;
    switch(a) {
    case __________:       /* Case A */
        c = __________;
        /* Fall through */
    case __________:       /* Case B */
        val = __________;
        break;
    case __________:       /* Case C */
    case __________:       /* Case D */
        val = __________;
        break;
    case __________:       /* Case E */
        val = __________;
        break;
    default:
        val = __________;
    }
    *dest = val;
}
```

GCC generates the assembly code and jump table shown in Figure 3.24.

Fill in the missing parts of the C code. Except for the ordering of case labels `C` and `D`, there is only one way to fit the different cases into the template.

Solution to Problem 3.31 (page 273)

The key to reverse engineering compiled `switch` statements is to combine the information from the assembly code and the jump table to sort out the different cases. We can see from the `ja` instruction (line 3) that the code for the default case has label `.L2`. We can see that the only other repeated label in the jump table is `.L5`, and so this must be the code for the cases C and D. We can see that the code falls through at line 8, and so label `.L7` must match case A and label `.L3` must match case B. That leaves only label `.L6` to match case E.

The original C code is as follows:

```
void switcher(long a, long b, long c, long *dest)
{
    long val;
    switch(a) {
    case 5:
        c = b ^ 15;
        /* Fall through */
    case 0:
        val = c + 112;
        break;
    case 2:
    case 7:
        val = (c + b) << 2;
        break;
    case 4:
        val = a;
        break;
    default:
        val = b;
    }
    *dest = val;
}
```