Sergio Garcia Tapia
Computer Systems: A Programmer's Perspective, by Bryant and O'Hallaron
Chapter 3: Machine-Level Representation of Programs March 25, 2024

Practice Problems

Exercise 3.1. Assume the following values are stored at the indicated memory addresses and registers:

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

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

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

Solution: To start, %rax is a 64-bit register conventionally used to store a return value. Its value is 0x100. Next, 0x104 looks like an immediate but it is not preceded by $, so it is in fact an absolute memory address. Its operand value is 0xAB. Next is $0x108, which is an immediate since it is preceded by a $, so its value is 0x108. The operand (%rax) is a an type of memory reference, specifically an indirect one. Therefore, the value 0x100 of %rax is used as an address, yielding 0xFF. The 4(%rax) operand is a memory operand where 4 is an immediate treated as an offset, and %rax is treated as the base. Therefore the address is 4 added to 0x100, yielding 0x104. Accessing the memory value at that address yields 0xAB. Next, 9(%rax, %rdx) is an indexed memory reference, where %rax is the base, %rdx is the 64-bit index register (normally used as a 3rd argument for a procedure), and 9 is an immediate offset. The memory address is thus 9 + 0x100 + 0x3. The resulting is memory address 0x10C, and the corresponding value is 0x11. Now 260(%rcx, %rdx), which is similar; the address is 260+0x1+0x3 which is 264 or 0x108, and its value is 0x13. Next 0xFC(,%rcx,4), which is a scaled index memory reference. We scale the address in the index register %rcx by 4, so it becomes 0x4, and then add to it the immediate 0xFC to give an address 0x100. The value is now determined to be 0xFF. Finally, (%rax,%rdx,4) is a scaled indexed memory reference, with address 0x100 in the base register %rax and value 0x3 in register %rdx scaled by 4 to give address 0x10C. The value is 0x11.

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

Exercise 3.2. For each of the following lines of assembly language, determine the appropriate suffix based on the operands. (For example, `mov` can be rewritten as `movb`, `movw`, `movl`, or `movq`.)

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

Solution: The `%eax` source register is 32-bit (a double word) and conventionally used as a return value, while the `%rsp` destination register is 64-bit (a quad word) and conventionally used as the stack pointer. Therefore we can use the `movl` instruction, where the `l` suffix indicates we are moving a double word.

The `(%rax)` is a n indirect memory reference using the address in the 64-bit (quad-word) `%rax` source register (conventionally used as a return address), and the destination 16-bit (word) register `%dx` (conventionally the 3rd argument in a procedure). This means we should use `movw`, since we are moving a single word.

The `$0xFF` source operand is an 8-bit immediate, and the destination `%bl` is an 8-bit (byte) register (conventionally callee-saved). For this we use `movb`.

The `(%rsp%rdx,4)` is the source, and it is a scaled index memory reference using the 64-bit (quad word) stack pointer register `%rsp` as the base address, the 64-bit (quad word) 3rd-argument register as the index register, and the scale factor `4`. The destination is `%dl`, the 8-bit (byte) 3rd argument register. For this we must use `movb`.

The `(%rdx)` operand is an indirect memory reference using the 64-bit (quad word) register `%rdx` (conventionally representing the 3rd argument) as the address. The destination is the 64-bit (quad word) register `%rax` normally used for the return value. We can use `movq` in this case.

Finally, we have source operand `%dx`, the 16-bit (word) third argument, and destination indirect memory reference using the address of the 64-bit (quad word) return value register `%rax`. We use `movw` in this case.

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

Exercise 3.3. Each of the following lines of code generates an error 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,     %dx
movq    %si,   8(%rbp)
```

Solution: The instruction `movb $0xF, (%ebx)` has as a destination operand the indirect memory reference `(%ebx)`, where `%ebx` is a 32-bit register. When a register is used in a memory addressing mode, its must be 64-bit; see page 181. We could fix the instruction by changing the destination operand to `(%rbx)`.

For `movl %rax, (%rsp)`, we have 64-bit (quad word) operands, but the `movl` instruction is meant to work with double words (32-bit, as indicated by the suffix `l`).

The instruction `movw (%rax), 4(%rsp)` is meant to work with 16-bit operands, as indicate by the word suffix `w`. However, its values are both 64-bit operands. Nevertheless, both operands are memory references, which is forbidden by x86-64; see page 183.

The instruction `movb %al, %sl` has an invalid register `%sl`. The intention may have been `%spl` for stack pointer or maybe `%sil` for the second argument, but it's not clear.

The instruction `movq %rax, $0x123` has an immediate as a destination, which is not allowed; only a register or a memory reference may be used as a destination.

The instruction `movl %eax, %dx` has a 32-bit source register and a 16-bit destination register. The `movl` instruction works with double words (32-bit) operands, so the destination register is incompatible. A fix would be use to use `movw`, where the `w` suffix indicates a word (16-bits).

The instruction `movq %si, 8(%rbp)` has an 8-bit (byte) source operand register, which is incompatible with `movq` which operates on quad words (64-bit).

Exercise 3.4. Assume variables `sp` and `dp` are declared with types

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

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

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

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 a size change and a change of "signedness" in C, the operation should change the size first (Section 2.2.6).

src_t	dest_t	Instruction
long	long	`movq (%rdi), %rax` `movq %rax, (%rsi)`
char	int	_____ _____
char	unsigned	_____ _____
unsigned char	long	_____ _____
int	char	_____ _____
unsigned	unsigned char	_____ _____
char	short	_____ _____

Solution: We will take `long` to be signed and 64 bit (quad word, 8 bytes), `int` to be signed and 32 bit (double word, 4 bytes), `unsigned` to be 32-bit and unsigned, `char` to be signed and 1 byte (8-bit), and `unsigned char` to be unsigned and 1 byte, and `short` to be 1 word (2 bytes or 16-bits).

Going from a source `char` of 1 byte to a destination `int` of 4 bytes requires using `movzbl`, since both operands are signed. Since the destination is 4 bytes (two words, 32-bit), we use the 32-bit `%eax` register.

From signed `char` of 1 byte to `unsigned` of 4 bytes requires using `movsbl`. This is because the operation should change the size first, so since `char` is signed, we keep its "signness" by using `movsbl` and not `movzbl`. Since the destination is 4 bytes, we use `movl` for the second operation.

From `unsigned char` of 1 byte to `long` which is signed and has 8 bytes (64-bit) requires that we change the size first, maintaining the signness. This suggests we use a move with the `z` suffix, since the source is unsigned so we should zero extend. Since we want a 64-bit result, we could use `movzbq` with `%rax` as the destination register. Then the last move simply uses `movq`. The book also uses `movzbl (%rdi), %eax`. This is valid because whenever the destination register of a `movl` instruction is a register, it also sets the high-order 4 bytes of the register to 0 (see page 183).

From signed `int` of 4 bytes to signed `char` of 1 byte, we truncate by using `movb` to move only the lowest order byte and the 8-bit `%al` register.

From `unsigned` of 4 bytes to `unsigned char` of 1 byte, we truncate again by using `movb` and the `%al` register.

Finally, from (signed) `char` of 1 byte to (signed) `short` of 2 bytes, we sign-extend and we use `movsbw` with the `%ax` register.

src_t	dest_t	Instruction
long	long	movq (%rdi), %rax movq %rax, (%rsi)
char	int	movsbl (%rdi), %eax movl %eax, (%rsi)
char	unsigned	movsbl (%rdi), %eax movl %eax %rsi
unsigned char	long	movzbq (%rdi), %rax movq %rax, (%rsi)
int	char	movb (%rdi), %al movb %al, (%rsi)
unsigned	unsigned char	movb (%rdi), %al movb %al, (%rsi)
char	short	movsbw (%rdi), %ax movw %ax, (%rsi)

Exercise 3.5. You are given the following information. A function with prototype

```
    void decode1(long *xp, long *yp, long *zp)
    xp in \%rdi,
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: The indirect memory reference `(%rdi)` dereferences `xp`, yielding its value `*xp`,and storing it in register `%r8`, conventionally used as the 5th argument of a procedure. This amounts to storing the value in a local variable `t` of the same type `long`. Similarly, `(%rsi)` is an indirect memory references that effectively dereferences `yp`, yielding its value `*yp` and storing it in register `%rcx`, normally used for a procedure's 4th argument. In C, this might be storing its in a local variable `s` of type `long`. The third memory reference `(%rdx)` serves to dereference `zp`, placing its value `*zp` in the `%rax` register, conventionally used for a return value of a procedure. Now the value stored in register `%r8` is stored at the location in memory pointed to by the `%rsi` register. This is equivalent to the assignment statement `*yp = t`. Next, the value in register `%rcx` is moved to the memory location pointed to by `%rdx`, which is equivalent to the statement `*zp = s`. Finally, the value in the return register `%rax` is placed at the memory location pointed to by register `%rdi`, which is equivalent to setting `*xp` to the value initially held by `*zp`.

The program below implements the C equivalent:

```
void decode1(long *xp, long *yp, long *zp) {
    long t = *xp;
    long s = *yp;
    long r = *zp;
    *yp = t;
    *zp = s;
    *xp = r;
    return r;
}
```

Exercise 3.6. Suppose register `%rax` holds value x and `%rcx` holds value y. Fill in the table below with formulas indicating the value that will be stored in register `%rdx` for each of the given assembly-code instructions.

Instruction	Result
`leaq 6(%rax), rdx`	_____
`leaq (%rax,%rcx), %rdx`	_____
`leaq (%rax,%rcx,4), %rdx`	_____
`leaq 7(%rax,%rax,8), %rdx`	_____
`leaq 0xA(,%rcx,4), %rdx`	_____
`leaq 9(%rax,%rcx,2), %rdx`	_____

Solution:

For `leaq 6(%rax), rdx`, the memory address used in the memory reference operand is that stored at `%rax`, which has value x offset by `6`. Therefore, the result is that register `%rdx` has value $x + 6$. The rest can be done similarly.

Instruction	Result
`leaq 6(%rax), %rdx`	$x + 6$
`leaq (%rax,%rcx), %rdx`	$x + y$
`leaq (%rax,%rcx,4), %rdx`	$x + 4y$
`leaq 7(%rax,%rax,8), %rdx`	$7 + x + 8x = 9x + 7$
`leaq 0xA(,%rcx,4), %rdx`	$10 + 4y$
`leaq 9(%rax,%rcx,2), %rdx`	$9 + x + 2y$

Exercise 3.7. Consider the following code, in which we have omitted the expression being computed:

```
long scale2(long x, long y, long z) {
    long t = ___________;
    return t;
}
```

Compiling the actual function with `gcc` yields the following assembly code:

```
    long scale2(long x, long y, long z)
    x in %rdi, y in %rsi, z in %rdx
scale2:
```

```
    leaq    (%rdi,%rdi,4), %rax
    leaq    (%rax,%rsi,2), %rax
    leaq    (%rax,%rdx,8), %rax
```

Solution: The first line places $5x = x + 4x$ in `%rax`. The second line places $5x + 2y$ in `%rax`. The last line places $5x + 2y + 8z$ in `%rax`. The function is therefore as follows:

```
long scale2(long x, long y, long z) {
    long t = 5x + 2y + 8z;
    return t;
}
```

Exercise 3.8. Assume the following values are stored at the indicated memory addresses and registers:

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

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

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

Solution: The `addq %rcx, (%rax)` instruction means that we add the quad stored in register `%rcx`, namely `0x1`, to the value store at the memory location whose address is the value `0x100` stored at `%rax`. This means we add `0x1` and `0xFF`, which results in `0x100`, and store its value at address `0x100`.

The `subq %rdx, 8(%rax)` instruction means we subtract the quad stored in register `%rdx`, which is `0x3`, from the value stored at the memory location whose address is the value stored at `%rax` offset by 8. Since `%rax` has value `0x100`, we add 8 to get `0x108`, and the value at that location is `0xAB`. Subtracting `0x3` results in `0xA8`.

The `imulq $16, (%rax,%rdx,8)` instruction means we multiply by 16 the value stored at the destination address. Since `%rax` is `0x100` and `%rdx` is 3, the destination address is `0x100 + 0x18`, which yields `0x118`. The value at that address is `0x11`, so multiplying by `16` yields `0x110`.

The `incq 16(%rax)` instruction says we increment by 1 the value stored at memory location whose address is that which is stored at `%rax` offset by 16. The address is `0x100 + 16` or `0x110`, so we are incrementing `0x13` by 1 to `0x14`.

The `decq %rcx` instruction decrements the value held in the `%rcx` register by 1, so `%rcx` goes from being `0x1` to being `0x0`.

The `subq %rdx, %rax` instruction subtracts the value stored at `%rdx` from the value at `%rax`. That is, we subtract `0x3` from `0x100`, yielding `0xFD`.

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`

Exercise 3.9. 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.

Solution:

```
    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
```

Exercise 3.10. In the following variant of the function of Figure 3.11(a), the expressions have been replaced by blanks:

```
long arith2(long x, long y, long z)
{
    long t1 = ________;
    long t2 = ________;
    long t3 = ________;
    long t4 = ________;
    return t4;
}
```

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

```
    long arith2(long x, long y, long z)
    x in %rdi, y in %rsi, z in %rdx
arith2:
    orq    %rsi,  %rdi
    sarq   $3,    %rdi
    notq   %rdi
    movq   %rdx,  %rax
    subq   %rdi,  %rax
    ret
```

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

Solution: The instruction `orq %rsi, %rdi` calculates `t1 = x | y`. The instruction `sarq $3, %rdi` calculates `t2 = t1 >> 3`, or equivalently `t2 = 8 * t1`. The instruction `notq %rdi` calculates `t3 = ~t2`. The instructions `movq %rdx, %rax` and `subq %rdi, %rax` together transform to `t4 = z - t3`. The resulting C program is below:

```
long arith2(long x, long y, long z)
{
    long t1 = x | y;
    long t2 = 8 * t1;
    long t3 = ~t2;
    long t4 = z - t3;
    return t4;
}
```

Exercise 3.11. It is common to find assembly-code lines of the form

```
xorq \%rdx \%rdx
```

in the 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 these two different implementations of the same operation.

Solution:

(a) Noting that `0^0` and `1^1` are both 0, the resulting operation is to yield a value 0 and place it in `%rdx`. This is essentially zeroing the register.

(b) The straightforward way to write this would be `imul $0, %rdx`, or `movq $0 %rdx`.

(c)

Exercise 3.12. 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: The text presented an function `remdiv` that was mostly equivalent to `uremdiv`, but with arguments of type `long` instead. In other words, it operated with signed numbers. To achieve it, the following assembly instructions were carried out:

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

The key instruction is `cqto`, which reads the sign bit from `%rax` and copies it across all of `%rdx`. For unsigned division, we instead want all zeros in register `%rdx`, so we replace `%cqto` with `movq $0, %rdx`:

```
    void remdiv(unsigned long x, unsigned long y, unsigned long *qp, unsigned
        long *rp)
    x in %rdi, y in %rsi, qp in %rdx, rp in %rcx
uremdiv:
    movq   %rdx,  %r8    // Copy qp
    movq   %rdi,  %rax   // Move x to lower 8 bytes of dividend
```

```
    moveq   $0,     %rdx    // Zero the register to signify unsigned arithmetic
    idivq   %rsi            // Divide by y
    movq    %rax,   (%r8)   // Store quotient at qp
    movq    %rdx,   (%rcx)  // Store remainder at rp
    ret
```

Exercise 3.13. 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 types listed in Figure 3.1 (`char`, `short`, `int`, `long`, or `char *`) and either signed or unsigned. The comparison `COMP` is defined via `#define`.

Suppose `a` is in some portion of `%rdi` 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
selbe    %al
```

(d)
```
cmpq   %rsi,   %rdi
selne    %al
```

Solution:

(a) The registers `%esi` and `%edi` are the lower 32-bit portions of `%rsi` and `%rdi`, respectively. Therefore, `data_t` is a 32-bit integer. Moreover, the `setl` suggests a signed `<` comparison. Therefore, `COMP` is `<` and `data_t` is `int`.

(b) Here, `cmpw` deals with 16-bit values, which is consistent with the fact that `%si` and `%di` are the lower 16-bit portions of `%rsi` and `%rdi`, respectively. Therefore, `data_t` is `short`. Also, `setge` is the signed `>=` instruction.

(c) Here, `cmpb` deals with 8-bit values, and `%sil` and `%dil` are the lower 8-bit portions of the `%rsi` and `%rdi` registers, respectively. Since `setbe` is the unsigned `<=` instruction, it follows that `data_t` is `unsigned char`.

(d) Here `selne` is the `!=` comparison operator, which applies to both signed and unsigned. Since `%rsi` and `%rdi` are 64-bit registers, it follows that `data_t` can be, signed or unsigned `long`, or any pointer type.

Exercise 3.14. The C code

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

shows a general comparison between `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 registers `%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:

(a) The `setge` indicates signed comparison, so `TEST` is `>=`. The `testq` suggests we are using a quad (64-bit), so `data_t` is a `long`.

(b) The `sete` is used for signed or unsigned equality checks, so `TEST` is `=`. The `testw` suggests we are operating on a word (16-bit), so `data_t` is either `short` or `unsigned short`.

(c) The `seta` is used for unsigned above comparison, so `TEST` is `>`. The `testb` is used for 8-bit comparison, and `%dil` is an 8-bit portion of `%rdi` register, so `data_t` is an `unsigned char`.

(d) The `setle` is used for signed less than or equal comparison, so `TEST` is `<=`. The `%edi` is the lower 32-bit portion of the `%rdi` register, so `data_t` is `int`.

Exercise 3.15. In the following excerpts from a disassembled binary, some of the information has been replaced by `X`'s. Answer the following questions about the 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 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   jmp    XXXXXX
4005ed: 90               nop    XXXXXX
```

Solution:

(a) The `0x02` in `74 02` must be added to `4003fc`, the address of the following instruction, to yield `4003fe` as the jump target of `je`.

(b) The `f4` in `74 f4` must be added to `400431`, the address of the next instruction. Since `f4` is a single byte has decimal value -12 (in two's complement), we subtract 12 from the hex address `400431` and obtain `400425`.

(c) Adding `0x02` from `77 02` to `XXXXXX` should give `400547`, and since `0x02` is decimal 2, we get that the `XXXXXX` that follows the jump instruction must be `400545`. Since the instruction `5d` at that location is 2 bytes after the first instruction in the `ja` line, we subtract 2 bytes to get `400543` for the byte address instruction of the `ja` line.

(d) The `73 ff ff ff` is written in little-endian (least to most significant) bit, so we can re-write it as `ff ff ff 73` (from most to least significant). This is a negative number since the most significant bit is 1, and it is a sign extension of `01 73`. Since `01 00` is 256 and `00 73` is 115, this means we have $-256 + 115 = -141$. Therefore we add -141 to the following address, `4005ed`, to get the jump target address which. The `ed` portion 237, and $237 - 141 = 96$, or `0x60`. Therefore the jump target address is `400560`. has value

Exercise 3.16. When given the C code

```
void cond(long a, long *p)
{
    if (p && a > &p)
        *p = a;
}
```

`gcc` generates the following code:

```
    void cond(long a, long *p)
    a in %rdi, p in %rsi
cond:
    testq  %rsi,  %rsi
    je     .L1
    cmpq   %rdi,  (%rsi)
```

```
    jge     .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:

(a) The annotated assembly is below:

```
    void cond(long a, long *p)
    a in %rdi, p in %rsi
cond:
    testq   %rsi,   %rsi    // Test p
    je      .L1             // If p != NULL (meaning 0) go to done
    cmpq    %rdi,   (%rsi)  // comp (*p):a
    jge     .L1             // if >= goto done
    movq    %rdi,   (%rsi)  // *p = a;
.L1:
    rep; ret                // return
```

The goto version is below:

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

}
```

(b) There are two conditional branches because the condition expression in the `if` statement is the `AND` of two condition expressions.

Exercise 3.17. 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:

(a)

```
void gotov2_absdiff_se(long x, long y)
{
    long result;
    if (x < y)
        goto x_le_y;
    ge_cnt++;
    result = x - y;
    goto done;
x_le_y:
    le_cnt++;
    result = y - x;
done:
    return result;
}
```

(b) I don't know! However the book mention that the first one is preferable when there is no else branch, since it's easier to translate.

Exercise 3.18. Starting with C code of the form

```
long test(long x, long y, long z) {
    long val = _______;
    if (________) {
        if (______)
            val = ______;
        else
            val = ______;
    } else if (______)
        val = _________;
    return val;
}
```

`gcc` generates the following assembly code:

```
   long test(long x, long y, long z)
   x in %rdi, y in %rsi, z in %rdx
test:
   leaq   (%rdi, %rsi), %rax
   addq   %rdx,  %rax
   cmpq   $-3,   %rdi
   jge    .L2
   cmpq   %rdx,  %rsi
   jge    .L3
   movq   %rdi,  %rax
   imulq  %rsi,  %rax
   ret
.L3:
   movq   %rsi,  %rax
   imulq  %rdx,  %rax
   ret
.L2:
   cmpq   $2,    %rdi
   jle    .L4
   movq   %rdi,  %rax
   imulq  %rdx,  %rax
.L4:
   rep; ret
```

Fill in the missing expressions in the C code.

Solution: First we can annotate the assembly:

```
   long test(long x, long y, long z)
   x in %rdi, y in %rsi, z in %rdx
test:
   leaq   (%rdi, %rsi), %rax   // long t = x + y;
   addq   %rdx,  %rax          // long val = t + z;
   cmpq   $-3,   %rdi          // Compare x:-3
   jge    .L2                  // if >= go to .L2
   cmpq   %rdx,  %rsi          // Compare y:z
   jge    .L3                  // if >= goto .L3
   movq   %rdi,  %rax          // val = x;
   imulq  %rsi,  %rax          // val *= y;
   ret                         // return val;
.L3:
   movq   %rsi,  %rax          // val = y;
   imulq  %rdx,  %rax          // val *= z;
   ret                         // return val;
.L2:
   cmpq   $2,    %rdi          // Compare x:2
   jle    .L4                  // if <= goto .L4
   movq   %rdi,  %rax          // val = x;
```

```
    imulq   %rdx,   %rax            // val *= z;
.L4:
    rep; ret                        // return val;
```

From this, the C code is

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

Exercise 3.19. Running on an older processor model, our code required around 16 cycles when the branching pattern was highly predictable, and around 31 cycles when the pattern was random.

(a) What is the appropriate miss penalty?

(b) How many cycles would the function require when the branch was mispredicted?

Solution:

(a) As discussed in the text, if p is the probability of misprediction, T_{OK} is the time to execute the code without misprediction, and T_{MP} is the misprediction penalty, then the average time to execute the code is given by

$$T_{avg}(p) = (1-p)T_{OK} + p(T_{OK} + T_{MP}) = T_{OK} + pT_{MP}$$
$$T_{MP} = \frac{1}{p}\left(T_{avg}(p) - T_{OK}\right)$$

We are given $T_{OK} = 16$, and $T_{ran} = T_{avg}(p) = 31$, so

$$T_{MP} = 2(31 - 16) = 30$$

(b) If mispredicted, the function would require $T_{OK} + TMP = 46$ cycles.

Exercise 3.20. In the following C function, we have left the definition of `OP` incomplete:

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

long arith(long x) {
    return x OP 8;
}
```

When compiled, `gcc` generated the following assembly code:

```
    long arith(long x)
    x in %rdi
arith:
    leaq    7(%rdi),    %rax
    testq   %rdi,       %rdi
    cmovns  %rdi,       %rax
    sarq    $3,         %rax
    ret
```

(a) What operation is `OP`?

(b) Annotate the code to explain how it works.

Solution:

(a) The book explains that `OP` is `/` because dividing by a power of 2 involves first biasing the number so that it rounds towards 0.

(b)
```
    long arith(long x)
    x in %rdi
arith:
    leaq    7(%rdi),    %rax    // int t = x + 7
    testq   %rdi,       %rdi    // test x
    cmovns  %rdi,       %rax    // if >=0 then t = x
    sarq    $3,         %rax    // t >>= 3; or equivalently t /= 8;
    ret                         // return t;
```

Exercise 3.21. Starting with C code of the form

```
long test(long x, long y) {
    long val = ___________;
    if (___________) {
        if (_________)
            val = ____________;
        else
            val = ____________;
    } else if (__________)
        val = ________;
    return val;
}
```

`gcc` generates the following assembly code:

```
    long test(long x, long y)
    x in %rdi, y in %rsi
test:
    leaq    0(,%rdi,8), %rax
```

```
    testq  %rsi,      %rsi
    jle    .L2
    movq   %rsi,      %rax
    subq   %rdi,      %rax
    movq   %rdi,      %rdx
    andq   %rsi,      %rdx
    cmpq   %rsi,      %rdi
    cmovge %rdx,      %rax
    ret
.L2:
    addq   %rsi,      %rdi
    cmpq   $-2,       %rsi
    cmovle %rdi,      %rax
    ret
```

Fill in the missing expressions in the C code.

Solution: We can first annotate the assembly:

```
    long test(long x, long y)
    x in %rdi, y in %rsi
test:
    leaq   0(,%rdi,8), %rax  // long r = x * 8;
    testq  %rsi,      %rsi   // test y
    jle    .L2               // if <= 0 goto .L2
    movq   %rsi,      %rax   // r = y;
    subq   %rdi,      %rax   // r -= x;
    movq   %rdi,      %rdx   // long s = x;
    andq   %rsi,      %rdx   // s &= y;
    cmpq   %rsi,      %rdi   // compare x:y
    cmovge %rdx,      %rax   // if >= then r = s;
    ret                      // return r;
.L2:
    addq   %rsi,      %rdi   // x += y;
    cmpq   $-2,       %rsi   // compare y:-2
    cmovle %rdi,      %rax   // if y <= -2 then r = x;
    ret                      // return r;
```

This reveals that the C code is as follows:

```
long test(long x, long y) {
    long val = x * 8;
    if (y > 0) {
        if (x >= y)
            val = x & y;
        else
            val = y - x;
    } else if (y <= -2)
        val = x + y;
```

```
    return val;
}
```

Exercise 3.22.

(a) What is the maximum value of n for which we can represent $n!$ with a 32-bit `int`?

(b) What about a 64-bit `long`?

Solution: I thought to print a table of values, but not how to determine overflow. The solution provided in the book was to use `tmult_ok` from practice problem 2.35 to check for overflow. I have provided this in `./22-factorial` for the case of `int`, which shows that $n = 12$ is the maximum before overflow, meaning 12! is ok, but 13! overflows.

Exercise 3.23. For the C code

```
long dw_loop(long x) {
    long y = x*x;
    long *p = &x;
    long n = 2*x;
    do {
        x += y;
        (*p)++;
        n--;
    } while (n > 0);
    return x;
}
```

`gcc` generates the following assembly code:

```
    long dw_loop(long x)
    x initially in %rdi
dw_loop:
    movq    %rdi,          %rax
    movq    %rdi,          %rcx
    imulq   %rdi,          %rcx
    leaq    (%rdi,%rdi),   %rdx
.L2:
    leaq    1(%rcx,%rax),  %rax
    subq    $1,            %rdx
    testq   %rdx,          %rdx
    jg      .L2
    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)++`?

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

Solution:

(a) Initially, `x` is in `%rdi`, but then it is placed in `%rax` since it is to be returned after modification. The variable `y` is placed in the `%rcx` register, and `n` is placed in the `%rdx` register.

(b) It has done so through the use of the `leaq` instruction to both increment `x` by 1 (the effect of `(*p)++`) in addition to increment `x` by `y`.

(c) The annotations are below:

```
   long dw_loop(long x)
   x initially in %rdi
dw_loop:
   movq   %rdi,          %rax   // long result = x;
   movq   %rdi,          %rcx   // y = x;
   imulq  %rdi,          %rcx   // y *= x;
   leaq   (%rdi,%rdi),   %rdx   // long n = 2 * x;
.L2:
   leaq   1(%rcx,%rax),  %rax   // result += y + 1;
   subq   $1,            %rdx   // n -= 1
   testq  %rdx,          %rdx   // test n
   jg     .L2                   // if > 0 goto .L2
   rep; ret                     // return result
```

Exercise 3.24. For C code having the general form

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

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

```
   long loop_while(long a, long b)
   a in %rdi, b in %rsi
loop_while:
   movl   $1,           %eax
   jmp    .L2
.L3:
   leaq   (%rdi,%rsi),  %rdx
   imulq  %rdx,         %rax
```

```
    addq    $1,             %rdi
.L2:
    cmpq    %rsi,           %rdi
    jl      .L3
    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 int he missing parts of the C code.

Solution: Below is my annotation of the assembly produced by `gcc`:

```
    long loop_while(long a, long b)
    a in %rdi, b in %rsi
loop_while:
    movl    $1,             %eax    // long result = 1;
    jmp     .L2                     // goto .L2
.L3:
    leaq    (%rdi,%rsi),    %rdx    // long t = a + b;
    imulq   %rdx,           %rax    // result *= t;
    addq    $1,             %rdi    // a += 1;
.L2:
    cmpq    %rsi,           %rdi    // compare a:b
    jl      .L3                     // if < goto .L3
    rep; ret
```

Based on this, I filled in the C code as shown below:

```
long loop_while(long a, long b)
{
    long result = 1;
    while (a < b) {
        result = result * (a + b);
        a = a + 1;
    }
    return result;
}
```

Exercise 3.25. For C code having the general form

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

gcc, when run with command-line option -O1, produces the following code:

```
    a in %rdi, b in %rsi
loop_while2:
    testq  %rsi,  %rsi
    jle    .L8
    movq   %rsi,  %rax
.L7:
    imulq  %rdi,  %rax
    subq   %rdi,  %rsi
    testq  %rsi,  %rsi
    jg     .L7
    rep; ret
.L8:
    movq   %rsi,  %rax
    ret
```

We can see that the compiler used a guarded-do translation, using the `jle` instruction on line 3 to skip over the loop code when the initial test fails. Fill in the missing parts of the C code. Note that the control structure in the assembly code does not exactly match what would be obtained by a direct translation of the C code according to our translation rules. In particular, it has two different `ret` instructions (lines 10 and 13). However, you can fill out the missing portions of the C code in a way that it will have equivalent behavior to the assembly code.

Solution: First I annotated the assembly like so:

```
    a in %rdi, b in %rsi
loop_while2:
    testq  %rsi,  %rsi   // test b
    jle    .L8           // if <= 0 goto .L8
    movq   %rsi,  %rax   // long result = b;
.L7:
    imulq  %rdi,  %rax   // result *= a;
    subq   %rdi,  %rsi   // b -= a;
    testq  %rsi,  %rsi   // test b
    jg     .L7           // if > 0 goto .L7
    rep; ret             // return result;
.L8:
    movq   %rsi,  %rax   // result = b;
    ret                  // return result;
```

The corresponding C then becomes:

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

```
    }
    return result;
}
```

Exercise 3.26. A function `fun_a` has the following overall structure:

```
long fun_a(unsigned long x) {
    long val = 0;
    while ( ... ) {
        .
        .
        .
    }
    return ...;
}
```

The `gcc` C compiler generates the following assembly code:

```
    long fun_a(unsigned long x)
    x in %rdi
fun_a:
    movl    $0,     %eax
    jmp     .L5
.L6:
    xorq    %rdi,   %rax
    shrq    %rdi            // Shift right by 1
.L5:
    testq   %rdi,   %rdi
    jne     .L6
    andl    $1,     %eax
    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: I first annotated the assembly as follows:

```
    long fun_a(unsigned long x)
    x in %rdi
fun_a:
    movl    $0,     %eax    // long result = 0;
    jmp     .L5             // goto .L5
.L6:
    xorq    %rdi,   %rax    // result = result ^ x;
```

```
    shrq    %rdi            // x >>= 1;
.L5:
    testq   %rdi,   %rdi    // test x
    jne     .L6             // if != 0 goto .L6
    andl    $1,     %eax    // result = result & 1;
    ret                     // return result;
```

(a) The `jmp .L5` instruction and the tests and jump in the lines that proceed label `.L5` suggests a jump-to-middle strategy.

(b) The C code can be filled in as follows:

```
long fun_a(unsigned long x) {
    long val = 0;
    while ( x != 0 ) {
        val = val ^ x;
        x = x >> 1;
    }
    return val & 1;
}
```

(c) Since `val` is 0, `0 ^ 0` is `0`, and `0 ^ 1` is `1`, it follows that the initial `val ^ x` sets `val` equal to `x`. Then, shifting `x` results in the least significant bit of `val` being XORed with the least significant bit of `x` after the shift. When the loop ends, the least significant bit of `val` will have the result of XORing all of the bits in `x`, and the `val & 1` yields that value. Since the `x` is an `unsigned long`, which has an even number of bits, and since the `XOR` of an even number of bits yields 0 if a number has an even number of 1 bits (or no 1 bits at all) and 1 otherwise, it follows that this function returns 1 if `x` has an odd number of 1 bits, and 0 otherwise.

Exercise 3.27. Write goto code for `fact_for` based on first transforming it to a `while` loop and then applying the guarded-do transformation.

Solution: The `fact_for` function is given below:

```
long fact_for(long n)
{
    long i;
    long result = 1;
    for (i = 2; i <= n; i++)
        result += i;
    return result;
}
```

The `while` loop version is below:

```
long fact_while(long n)
{
    long result = 1;
```

```
    long i = 2;
    while (i <= n) {
        result *= i;
        i++;
    }
    return result;
}
```

The guarded-do translation follows:

```
long fact_while_guarded_do(long n)
{
    long result = 1;
    long i = 2;
    if (i > n)
        goto done;
loop:
    result *= i;
    i++;
    if (i <= n)
        goto loop;
done:
    return result;
}
```

Exercise 3.28. A function `fun_b` has the following overall structure:

```
long fun_b(unsigned long x) {
    long val = 0;
    long i;
    for ( ... ; ... ; ... ) {
        .
        .
        .
    }
    return val;
}
```

The `gcc` C compiler generates the following assembly code:

```
    long fun_b(unsigned long x)
    x in %rdi
fun_b:
    movl    $64,    %edx
    movl    $0,     %eax
.L10:
    movq    %rdi,   %rcx
    andl    $1,     %ecx
```

```
    addq    %rax,   %rax
    orq     %rcx,   %rax
    shrq    %rdi            // Shift right by 1
    subq    $1,     %rdx
    jne     .L10
    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: I began by annotating the assembly code as follows:

```
long fun_b(unsigned long x)
x in %rdi
fun_b:
    movl    $64,    %edx    // unsigned long t = 64;
    movl    $0,     %eax    // long result = 0;
.L10:
    movq    %rdi,   %rcx    // long v = x;
    andl    $1,     %ecx    // v = v & 1;
    addq    %rax,   %rax    // result = result + result;
    orq     %rcx,   %rax    // result = result | v;
    shrq    %rdi            // x = x >> 1;
    subq    $1,     %rdx    // t = t - 1;
    jne     .L10            // if t != 0 goto .L10
    rep; ret
```

(a) Based on my annotations of the assembly, I deduced the C code to be:

```c
long fun_b(unsigned long x) {
    long val = 0;
    long i;
    for ( i = 64 ; i != 0 ; i-- ) {
        long xlsb = x & 1; // Get least significant bit of x
        val = (2 * val) | xlsb;
        x = x >> 1;
    }
    return val;
}
```

(b) Neither test is present because the loop always iterates 64 times.

(c) In the first iteration, the least significant bit of `val` has the least significant bit of `x`, and all of its other bits are 0. In the next iteration, multiplying `val` by 2 shifts all of the bits of `val` left by 2 while copying in the next-least significant bit of `x` into the least significant position of `val`. The apparent effect is that it reverses the bits of `x`.

Exercise 3.29. 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:

1. We would get the following if we "naively" translated the for loop:

   ```
   /* Naive translation of for loop with continue statement into while loop */
   /* Sum even numbers between 0 and 9 */
   long sum = 0;
   long i = 0;
   while (i < 10) {
       if (i & 1)
           continue;
       sum += i;
       i++;
   }
   ```

   The translation creates an infinite loop. In the first iteration, the test expression in the `if` statement evaluates to false, so `continue` is not evaluated, causing `i++` to execute, thus increasing `i` to 1. With this new value, the condition in the `if` statement now succeeds, causing `continue` to be executed. As a result, we go back to the top of the while loop, but we never reach the `i++` statement thereafter. Thus, the value of `i` remains at 1 indefinitely.

2. We could replace the `continue` with a `goto next_iter`, where `next_iter` is a label under which the `i++;` statement is present.

```
/* Translation of for loop with continue statement into while loop */
/* Sum even numbers between 0 and 9 */
long sum = 0;
long i = 0;
while (i < 10) {
    if (i & 1)
        goto next_iter;
    sum += i;
next_iter:
    i++;
}
```

Exercise 3.30. 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(long x, long *dest) {
    long 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(long x, long *dest)
    x in %rdi
switch2:
    addq    $1,     %rdi
    cmpq    $8,     %rdi
    ja      .L2
    jmp     *.L4(,%rdi,8)
```

It generates the following code for the jump table:

```
.L4:
    .quad   .L9
    .quad   .L5
    .quad   .L6
    .quad   .L7
    .quad   .L2
    .quad   .L7
    .quad   .L8
    .quad   .L2
    .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:

(a) Below I have annotated the initial part of the assembly for the procedure provided:

```
    void switch2(long x, long *dest)
    x in %rdi
switch2:
    addq    $1,     %rdi    // x += 1
    cmpq    $8,     %rdi    // cmp x:8
    ja      .L2             // if > goto .L2 (default case)
    jmp     *.L4(,%rdi,8)   // Go to *jt[index]
```

Based on the annotation, `x` was adjusted so that it would be an index between 0 and 8, inclusive by adding 1. Hence, `x` must have been between -1 and 7. Based on this, we can annotate the jump table assembly snippet given:

```
.L4:
.quad   .L9 // case -1
.quad   .L5 // case 0
.quad   .L6 // case 1
.quad   .L7 // case 2
.quad   .L2 // case 3 (default)
.quad   .L7 // case 4
.quad   .L8 // case 5
.quad   .L2 // case 6
.quad   .L5 // case 7
```

The `.L2` label is reserved for the `default` case. The `.L4` for the jump table addresses. The only `x` values with no matching `case` are when `x` is 3 or 6, because for those values, `switch` transfers control to the `default` branch.

(b) The labels that repeat constitute the cases with multiple labels (except for `.L2`, corresponding to the `default` branch) are the cases with multiple labels. These are `.L5` with cases `0` or `7`, and `.L7` with cases 2 or 4.

Exercise 3.31. 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 below:

```
    void switcher(long a, long b, long c, long *dest)
    a in %rdi, b in %rsi, c in %rdx, d in %rcx
switcher:
    cmpq    $7,             %rdi
    ja      .L2
    jmp     *.L4(,%rdi,8)
    .section                .rodata
.L7:
    xorq    $15,            %rsi
    movq    %rsi,           %rdx
.L3:
    leaq    112(%rdx),      %rdi
    jmp     .L6
.L5:
    leaq    (%rdx,%rsi),    %rdi
    salq    $2,             %rdi
    jmp     .L6
.L2:
    movq    %rsi,           %rdi
.L6:
    movq    %rdi,           (%rcx)
    ret
```

and it also generates the following jump table:

```
.L4:
    .quad   .L3
    .quad   .L2
    .quad   .L5
    .quad   .L2
    .quad   .L6
    .quad   .L7
```

```
    .quad   .L2
    .quad   .L5
```

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: I began by annotating the assembly:

```
void switcher(long a, long b, long c, long *dest)
a in %rdi, b in %rsi, c in %rdx, d in %rcx
switcher:
    cmpq    $7,             %rdi    // compare a:7
    ja      .L2                     // if > 7 goto .L2 (default branch)
    jmp     *.L4(,%rdi,8)           // Go to *jt[index]
    .section                .rodata
.L7:                                // Case A
    xorq    $15,            %rsi    // b = b ^ 15;
    movq    %rsi,           %rdx    // c = b;
                                    // fall through
.L3:                                // Case B
    leaq    112(%rdx),      %rdi    // a = c + 112;
    jmp     .L6                     // break
.L5:                                // Cases C and D
    leaq    (%rdx,%rsi),    %rdi    // a = c + b
    salq    $2,             %rdi    // a <<= 2;
    jmp     .L6                     // break
.L2:                                // default
    movq    %rsi,           %rdi    // a = b;
.L6:                                // just after switch stament, or case E
    movq    %rdi,           (%rcx) // *dest = a;
    ret
```

It seems there is no case E, from which I deduce that case E actually does the same thing as the last statement pf `*dest = val`. That is, case E just sets `val` equal to `a`. Based on this the case values corresponding to the jump table labels are as follows:

```
.L4:
    .quad   .L3 // case 0 (case B)
    .quad   .L2 // case 1 (default)
    .quad   .L5 // case 2 (case C)
    .quad   .L2 // case 3 (default)
    .quad   .L6 // case 4 (case E)
    .quad   .L7 // case 5 (case A)
    .quad   .L2 // case 6 (default)
    .quad   .L5 // case 7 (case D)
```

Based on this, the C code is as follows:

```
void switcher(long a, long b, long c, long *dest)
```

```
{
    long val;
    switch(a) {
        case 5:    /* Case A */
            c = b ^ 15;
            /* Fall through */
        case 0:    /* Case B */
            val = c + 112;
            break;
        case 2:    /* Case C */
        case 7:    /* Case D */
            val = (c + b) << 2;
            break;
        case 4:    /* Case E */
            val = a;
            break;
        default:
            val = b;
    }
    *dest = val;
}
```