Sergio Garcia Tapia
Computer Systems: A Programmer's Perspective, by Bryant and O'Hallaron
Chapter 3: Machine-Level Representation of Programs March 31, 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:

```c
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

```c
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

```c
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:

```c
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)

```c
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

```c
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

```c
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:

```c
#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:

```c
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;
```

```c
    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

```c
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:

```c
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

```c
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:

```c
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.

```c
/* 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.

```c
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 = _________;
```

```c
            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;
}
```

Exercise 3.32. The disassembled code for two functions `first` and `last` is shown below, along with the code for a call of `first` by function `main`:

```
    Disassembly of last(long u, long v)
    u in %rdi, v in %rsi
0000000000400540 <last>:
    400540: 48 89 f8          mov    %rdi, %rax     // L1: u
    400543: 48 0f af c6       imul   %rsi, %rax     // L2: u*v
    400547: c3                retq                  // L3: Return

    Disassembly of first(long x)
    x in %rdi
0000000000400548 <last>:
    400548: 48 8d 77 01       lea    0x1(%rdi), %rsi // F1: x+1
    40054c: 48 83 ef 01       sub    $0x1,     %rdi // F2: x-1
    400550: e8 eb ff ff ff    callq  400540 <last> // F3: Call last(x-1,x+1)
    400555: f3 c3             repz retq            // F4: Return
    .
    .
    .
    400560: e8 e3 ff ff ff    callq  400540 <first> // M1: Call first(10)
    400565: 48 89 c2          mov    %rax,%rdx      // M2: Resume
```

Each of these instructions is given a label, similar to those in Figure 3.27(a). Starting with the calling of `first(10)` by `main`, fill in the following table to trace instruction execution through the point where the program returns back to `main`.

Instruction			State values (at beginning)					
Label	PC	Instruction	%rdi	%rsi	%rax	%rsp	*%rsp	Description
M1	0x400560	callq	10	—	—	0x7fffffffe820	—	Call first(10)
F1	____	____	____	____	____	____	____	____
F2	____	____	____	____	____	____	____	____
F3	____	____	____	____	____	____	____	____
L1	____	____	____	____	____	____	____	____
L2	____	____	____	____	____	____	____	____
L3	____	____	____	____	____	____	____	____
F4	____	____	____	____	____	____	____	____
M2	____	____	____	____	____	____	____	____

Solution: Each time a function call is made, the address is pushed onto the stack, so the stack pointer is moved down by 8 bytes. (since an address is 64-bit). For example, when `main` calls `first`, the return address on top of the stack goes from `0x7fffffffe820` to `0x7fffffffe818`, a decrease by 8 bytes. Similarly, when `first` calls `last`, it decreases to `0x7fffffffe810`.

Instruction			State values (at beginning)					
Label	PC	Instruction	%rdi	%rsi	%rax	%rsp	*%rsp	Description
M1	0x400560	callq	10	—	—	0x7fffffffe820	—	Call first(10)
F1	0x400548	lea	10	–	—	0x7fffffffe818	0x400565	Entry of first
F2	0x40054c	sub	10	11	—	0x7fffffffe818	0x400565	Subtract 1 from x
F3	0x400550	callq	9	11	—	0x7fffffffe818	0x400565	Call last(9, 11)
L1	0x400540	mov	9	11	—	0x7fffffffe810	0x400555	Entry of last
L2	0x400543	imul	9	11	9	0x7fffffffe810	0x400555	Multiply u*v
L3	0x400547	retq	9	11	99	0x7fffffffe810	0x400555	Return 99 from last
F4	0x400555	repz retq	9	11	99	0x7fffffffe818	0x400565	Return 99 from first
M2	0x400565	mov	9	116	99	0x7fffffffe820	—	Resume main

Exercise 3.33. A C function `procprob` has four arguments `u`, `a`, `v`, and `b`. Each is either a signed number or a pointer to a signed number, where the numbers have different sizes. The function has the following body:

```
*u += a;
*v += b;
return sizeof(a) + sizeof(b);
```

It compiles to the following x86-64 code:

```
procprob:
    movslq %edi,  %rdi
    addq   %rdi,  (%rdx)
    addb   %sil,  (%rcx)
    movl   $6,    %eax
    ret
```

Determine a valid ordering and types of the four parameters. There are two correct answers.

Solution: The first argument is in `%rdi`, the second in `%rsi`, the third in `%rdx`, and the fourth in `%rcx`. The `addq %rdi, (%rdx)` command corresponds to the statement `*u += a`, so `a` is the first argument, and `u` is the third argument. Similarly, the `addb %rsi, (%rcx)` corresponds to the statement `*v += b`, and `%sil` is the lower byte of `%rsi`, so `b` is the second argument and `v` is the 4th argument

We are given that all arguments are signed integers or pointers to signed integers. The `moveslq %edi, %rdi` sign extends the lower half register of `%rdi`, in this case from 32-bit to 64-bit. Hence, `a` must be a 32-bit integer (an `int`) and `v` must be a pointer to a 64-bit integer (a `long *`). Since `a` is 4 bytes, it must be that `b` is 2 bytes because `%movl $6, %eax` implies the sum of their sizes is 6 bytes, so `b` is a `short`. The `addb %sil, (%rcx)` suggests an 8-bit operand, so `v` is a pointer to a `char` (a `char *`). This means the function signature would be

```
procprob(int a, short b, long *u, char *v);
```

Exercise 3.34. Consider a function `P`, which generates local values, named `a0-a8`. It then calls function `Q` using these generated values as arguments. `gcc` produces the following code for the first part of `P`:

```
    long P(long x)
    x in %rdi
P:
    pushq   %r15
    pushq   %r14
    pushq   %r13
    pushq   %r12
    pushq   %rbp
    pushq   %rbx
    subq    $24, %rsp
    movq    %rdi, %rbx
    leaq    1(%rdi), %r15
    leaq    2(%rdi), %r14
    leaq    3(%rdi), %r13
    leaq    4(%rdi), %r12
    leaq    5(%rdi), %rbp
    leaq    6(%rdi), %rax
    movq    %rax, (%rsp)
    leaq    7(%rdi), %rdx
    movq    %rdx, 8(%rsp)
    movl    $0, %eax
    call    Q
    . . .
```

(a) Identify which local values get stored in callee-saved registers.

(b) Identify which local values get stored on the stack.

(c) Explain why the program could not store all of the local variables in callee-saved registers.

Solution:

(a) The callee-saved registers are `%rbx`, `%rbp`, and `%r12` through `%r15`. After pushing these values onto the stack, `P` stores `x` at `%rbx`, `x + 1` in `%rx15`, `x + 2` in `%rx14`, `x + 3` in `%rx13`, `x + 4` in `%rx12`, and `x + 5` in `%rbp`.

(b) The value `x + 6` is saved at the top of the stack. The value `x + 7` is saved at an 8-byte offset of the top-of-stack.

(c) There are only 6 callee-saved registers, and to store `a0` through `a8` requires 9 registers. Therefore, `P` must store the remaining 3 on the stack. This explains the `subq $24, %rsp` instruction, which allocates space on the stack for 3 local variables.

Exercise 3.35. For a C function having the general structure

```c
long rfun(unsigned long x) {
    if (__________)
        return ____________;
    unsigned long nx = ___________;
    long rv = rfun(nx);
    return _________;
}
```

`gcc` generates the following assembly code:

```
    long rfun(unsigned long x)
    x in %rdi
rfun:
    pushq   %rbx
    movq    %rdi, %rbx
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L2
    shrq    $2, %rdi
    call    rfun
    addq    %rbx, %rax
.L2:
    popq    %rbx
    ret
```

(a) What value does `rfun` store in the callee-saved register `%rbx`?

(b) Fill in the missing expressions in the C code shown above.

Solution:

(a) First we can annotate the assembly code:

```
long rfun(unsigned long x)
x in %rdi
```

```
rfun:
    pushq   %rbx            // Save %rbx
    movq    %rdi, %rbx      // Store x in callee-saved register.
    movl    $0, %eax        // Set return val to 0
    testq   %rdi, %rdi      // Test x
    je      .L2             // if == 0 goto .L2
    shrq    $2, %rdi        // unsigned long nx = x >> 2; (logical right shift)
    call    rfun            // Call rfun(nx)
    addq    %rbx, %rax      // long rv = x + rfun(nx)
.L2:
    popq    %rbx            // Restore %rbx
    ret                     // Return
```

We conclude that `rfun` stores `x`, its sole argument, in the callee-saved register `%rbx`, after saving the existing value on the stack, and restoring it before returning.

(b) The C code is as follows:

```c
long rfun(unsigned long x) {
    if (x == 0)
        return 0;
    unsigned long nx = x >> 2;
    long rv = rfun(nx);
    return x + rv;
}
```

Exercise 3.36. Consider the following declarations

```c
short    S[7];
short   *T[3];
short  **U[6];
int      V[8];
double  *W[4];
```

Fill in the following table describing the element size, the total size, and the address of element i for each of these arrays.

Array	Element Size	Total size	Start address	Element i
S	_____	_____	x_S	_____
T	_____	_____	x_T	_____
U	_____	_____	x_U	_____
V	_____	_____	x_V	_____
W	_____	_____	x_W	_____

Solution: The sizes in the brackets constitute the number of elements. The element size is given by the type, where `short` is 2 bytes, `int` is 4 bytes, and all pointer types are 8 bytes. From this, the table is as follows:

Array	Element Size	Total size	Start address	Element i
S	2	14	x_S	$x_S + 2i$
T	8	24	x_T	$x_T + 8i$
U	8	48	x_U	$x_U + 8i$
V	4	32	x_V	$x_V + 4i$
W	8	32	x_W	$x_W + 8i$

Exercise 3.37. Suppose x_S, the address of short integer array `S`, and long integer index i are stored in registers `%rdx` and `%rcx`, respectively. For each of the following expressions, give its type, a formula for its value, and an assembly-code implementation. The result should be stored in register `%rax` if it is a pointer and register element `%ax` if it has data type `short`.

Expression	Type	Value	Assembly code
`S+1`	______	______	______
`S[3]`	______	______	______
`&S[i]`	______	______	______
`S[4*i+1]`	______	______	______
`S+i-5`	______	______	______

Solution: I used the fact that `short` is 2 bytes, and `long` is 8 bytes. Since 2 bytes is a "word", we use `movw` for data movements.

Expression	Type	Value	Assembly code
`S+1`	`short *`	$x_S + 2$	`leaq 2(%rdx), %rax`
`S[3]`	`short`	$\mathrm{M}[x_S + 6]$	`movw 6(%rdx), %ax`
`&S[i]`	`short *`	$x_S + 2i$	`leaq (%rdx,%rcx,2), %rax`
`S[4*i+1]`	`short`	$\mathrm{M}[x_S + 8i + 2]$	`movw 2(%rdx, %rcx, 8), %ax`
`S+i-5`	`short *`	$x_S + 2i - 10$	`leaq -10(%rdx, %rcx,2), %rax`

Exercise 3.38. Consider the following source code, where M and N are constants declared with `#define`:

```c
long P[M][N];
long Q[N][M];

long sum_element(long i, long j) {
    return P[i][j] + Q[j][i];
}
```

In compiling this program, `gcc` generates the following assembly code:

```
    long sum_element(long i, long j)
    i in %rdi, j in %rsi
sum_element:
    leaq    0(,%rdi,8), %rdx
    subq    %rdi, %rdx
    addq    %rsi, %rdx
    leaq    (%rsi,%rsi,4), %rax
    addq    %rax, %rdi
    movq    Q(,%rdi,8), %rax
```

```
    addq   P(,%rdx,8), %rax
    ret
```

Use your reverse engineering skills to determine the values M and N based on this assembly code.

Solution: Note that `long` is of size 8 bytes. The address `&P[i][j]` is given by $x_P + 8(N \cdot i + j)$, whereas `&Q[j][i]` is given by $x_Q + 8(M \cdot j + i)$. First we can annotate the assembly:

```
    long sum_element(long i, long j)
    i in %rdi, j in %rsi
sum_element:
    leaq   0(,%rdi,8), %rdx      // long a = 8i
    subq   %rdi, %rdx            // a = a - i /* now a is 7i */
    addq   %rsi, %rdx            // a = a + j /* now a is 7i + j */
    leaq   (%rsi,%rsi,4), %rax   // long b = j + 4j /* now it is 5j */
    addq   %rax, %rdi            // long c = i + b /* c = i + 5j */
    movq   Q(,%rdi,8), %rax      // result = M[Q + 8*c]
    addq   P(,%rdx,8), %rax      // result += M[P + 8*a]
    ret
```

Based on the annotations, `&Q[j][i]` is $x_Q + 8(5j + i)$, and `&P[i][j]` is $x_P + 8(7i + j)$. Hence, $N = 7$, and $M = 5$.

Exercise 3.39. Use Equation 3.1 to explain how the computations of the initial values for `Aptr`, `Bptr`, and `Bend` in the C code in Figure 3.37(b) (lines 3-5) correctly describe their computations in the assembly code generated for `fix_prod_ele` (lines 3-5).

Solution: Equation 3.1 says that if T is a data type of size L, then given the array

```
T D[R][C];
```

with R rows and C columns, the array element `D[i][j]` is at memory address

$$\&D[i][j] = x_D + L(C \cdot i + j)$$

where x_D is the address of D. Recall that `A` and `B` are of types `int[16][16]`. To access `A[i][0]`, the first element in the i-th row, we have $L = 4$ because `int` takes up 4 bytes, and C is 16, so

$$\&A[i][0] = x_A + 4(16 \cdot i + 0) = x_A + 64i$$

Similarly, to access `B[0][k]`, the first element in the k-th column we have $L = 4$ because `int` takes up 4 bytes, and C is 16, so

$$\&B[0][k] = x_B + 4(16 \cdot 0 + k) = x_B + 4k$$

Finally, to access the first element beyond the k-th column, namely, the first (thus marking the end of column k of `B`), and recalling that $N = 16$, we have

$$\&B[N][k] = x_B + 4(16 \cdot N + k) = x_B + 4(16 \cdot 16 + k) = x_B + 1024 + 4k$$

Exercise 3.40. The following C code sets the diagonal elements of one of our fixed-sized arrays to `val`:

```c
/* Set all diagonal elements to val */
void fix_set_diag(fix_matrix A, int val) {
    long i;
    for (i = 0; i < N; i++)
        A[i][i] = val;
}
```

When compiled with optimization level `-O1`, `gcc` generates the following assembly code:

```
    void fix_set_diag(fix_matrix A, int val)
    A in %rdi, val in %rsi
fix_set_diag:
    movl    $0, %eax
.L13:
    movl    %esi, (%rdi,%rax)
    addq    $68, %rax
    cmpq    $1088, %rax
    jne     .L13
    rep; ret
```

Create a C code program `fix_set_diag_opt` that uses optimizations similar to those in the assembly code, in the same style as the code in Figure 3.37(b). Use expressions involving the parameter N rather than integer constants, so that your code will work correctly if N is redefined.

Solution: First we can annotate the assembly:

```
    void fix_set_diag(fix_matrix A, int val)
    A in %rdi, val in %rsi
fix_set_diag:
    movl    $0, %eax          // i = 0
.L13:
    movl    %esi, (%rdi,%rax) // M[A + i] = val
    addq    $68, %rax         // i += 16 * 4 + 4
    cmpq    $1088, %rax       // compare i:(4*(16*16 + 16)), N = 16
    jne     .L13              // if != go to .L13
    rep; ret                  // Return
```

Based on the annotations, we can produce the following C code:

```c
/* (Optimized) Set all diagonal elements to val */
void fix_set_diag_opt(fix_matrix A, int val) {
    int i = 0;            /* Running index */
    int last = N * N + N; /* Index is one beyond end */
    do {
        A[i] = val;
        i += (N + 1);
    } while (i != last);
}
```

Exercise 3.41. Consider the following structure declaration:

```
struct prob {
    int *p;
    struct {
        int x;
        int y;
    } s;
    struct prob *next;
};
```

This declaration illustrates that one structure can be embedded within another, just as arrays can be embedded within structures and arrays can be embedded within arrays. The following procedure (with some expressions omitted) operates on this structure:

```
void sp_init(struct prob *sp) {
    sp->s.x = ________;
    sp->p   = ________;
    sp->next = ________;
}
```

(a) What are the offsets (in bytes) of the following fields?

`p`:	______
`s.x`:	______
`s.y`:	______
`next`:	______

(b) How many total bytes does the structure require?

(c) The compiler generates the following assembly code for `sp_init`:

```
    void sp_init(struct prob *sp)
sp_init:
    movl   12(%rdi), %eax
    movl   %eax, 8(%rdi)
    leaq   8(%rdi), %rax
    movq   %rax, (%rdi)
    movq   %rdi, 16(%rdi)
    ret
```

On the basis of this information, fill in the missing expressions in the code for `sp_init`.

Solution:

(a) The offsets are determined by the size of the type of each field, as follows:

Field	Byte Offset
p:	0 bytes
s.x:	8 bytes
s.y:	12 bytes
next:	16 bytes

(b) The structure required 24 bytes.

(c) The annotated assembly is

```
    void sp_init(struct prob *sp)
    sp in %rdi
sp_init:
    movl   12(%rdi), %eax // int y = sp->y;
    movl   %eax, 8(%rdi)  // sp->s.x = y;
    leaq   8(%rdi), %rax  // s = &(sp->s.x)
    movq   %rax, (%rdi)   // sp->p = s
    movq   %rdi, 16(%rdi) // r->next = sp
    ret
```

The corresponding C code is:

```c
void sp_init(struct prob *sp) {
    sp->s.x = sp->y;
    sp->p   = &(sp->s.x);
    sp->next = sp;
}
```

Exercise 3.42. The following code shows the declaration of a structure of type `ELE` and the prototype for a `fun`:

```c
struct ELE {
    long   v;
    struct ELE *p;
};

long fun(struct ELE *ptr);
```

When the code for `fun` is compiled, `gcc` generates the following assembly code:

```
    long fun(struct ELE *ptr)
    ptr in %rdi
fun:
    movl   $0, %eax
    jmp    .L2
.L3:
    addq   (%rdi), %rax
    movq   8(%rdi), %rdi
.L2:
```

```
    testq   %rdi, %rdi
    jne     .L3
    rep; ret
```

(a) Use your reverse engineering skills to write C code for `fun`.

(b) Describe the data structure that this structure implements and the operation performed by `fun`.

Solution: (a) The annotated assembly is below:

```
    long fun(struct ELE *ptr)
    ptr in %rdi
fun:
    movl    $0, %eax         // long result = 0;
    jmp     .L2              // goto .L2
.L3:
    addq    (%rdi), %rax     // result += *ptr;
    movq    8(%rdi), %rdi    // ptr++;
.L2:
    testq   %rdi, %rdi       // test ptr
    jne     .L3              // if != NULL goto .L3
    rep; ret
```

The C implementation of `fun` is:

```c
long fun (struct ELE *ptr) {
    long result = 0;
    while (ptr != NULL) {
        result += *ptr;
        ptr++;
    }
    return result;
}
```

(b) The data type `struct ELE` implements a linked list, and the function `fun` computes the sum of all the elements in it.

Exercise 3.43. Suppose you are given the job of checking that a C compiler generates the proper code for structure and union access. You write the following structure declaration:

```c
typedef union {
    struct {
        long    u;
        short   v;
        char    w;
    } t1;
    struct {
```

```
        int a[2];
        char *p;
    } t2;
} u_type;
```

You write a series of functions of the form

```
void get(u_type *up, type *dest) {
    *dest = expr;
}
```

with different access expressions `expr` and with destination data type `type` set according to the type associated with `expr`. You then examine the code generated when compiling the functions to see if they match your expectations.

Suppose in these functions that `up` and `dest` are loaded into registers `%rdi` and `%rsi`, respectively. Fill in the following table with data type *type* and sequences of one to three instructions to compute the expression and store the result at `dest`.

expr	*type*	Code
`up->t1.u`	`long`	`movq (%rdi), %rax`
		`movq %rax, (%rsi)`
`up->t1.v`	______	________________
		________________
`&up->t1.w`	______	________________
		________________
`up->t2.a`	______	________________
		________________
`up->t2.a[up->t1.u]`	______	________________
		________________
`*up->t2.p`	______	________________
		________________

Solution: The union structure of type `u_type` has a size that is the maximum of its member fields, subject to proper alignment. Its field `t1` is a structure is a structure with a `long`, a `short`, and `char`, whose overall size is $8 + 2 + 1 = 11$ bytes. Its other field, `t2`, has an `int[2]` array and a `char *`, both of which take up 8 bytes, for a total of 16 bytes. Therefore, `u_type` takes up 16 bytes.

In a union, the address of each field is the same as the union object itself. In practice, this means that `up->t1` and `up->t2` both reference the beginning of the data structure. This can be seen in the code for the first entry of the table provided, for the expression `up->t1.u`, whose first associated instruction is `movq (%rdi), %rax`. Because `up` and up-¿t1 both refer to the beginning of the data structure, and because the offset of the first field `u` in the data structure for `t1` is also at the beginning of that data structure, we can see that their addresses are all the same (here, `%rdi`).

Since the first field of `up->t1` is `up->t.u`, an 8-byte `long`, this means that the next field `up->t1.v` is at an 8 byte offset. Next, since `u` and `v` in struct `t1` are an 8-byte `long` and a 2-byte `short`, respectively, it follows that `up->t1.w` is at a 10-byte offset. Moreover, the address operator `&` causes the expression to return a pointer (in this case, a `char *`). The `up->t2.a` refers to the first location as `up`, as explained before. However, `a` is an `int *` pointer. The next expression, `up->t2.a[up->t1.u]`, requires three instructions: one to compute the index given by `up->t1.u`, a memory reference that is stored in a register, and then a move from the register to (%rsi). The latter steps must be broken into two because when a memory location is a destination in an instruction, the source cannot also be a memory location. The last expression, `*up->t2.p`, dereferences the member pointer `p`, so the expression has type `char`. The reference `up->t2.p` is at an 8-byte offset since it follows the 8-byte `int[2]` array in `t2`. Once again we need three instructions, because we need one instruction to compute the value of `p` before the instruction to dereference it, and once again, we need to put its value at a register because a memory reference cannot be the source and destination of an instruction.

expr	*type*	Code
`up->t1.u`	`long`	`movq (%rdi), %rax` `movq %rax, (%rsi)`
`up->t1.v`	`short`	`movw 8(%rdi), %ax` `movw %ax, (%rsi)`
`&up->t1.w`	`char *`	`leaq 10(%rdi), %rax` `movw %rax, (%rsi)`
`up->t2.a`	`int *`	`leaq (%rdi), %rax` `movq %rax, (%rsi)`

`up->t2.a[up->t1.u]`	`int`	`movq (%rdi), %rax`
		`movl (%rdi, %rax, 4), %eax`
		`movl %eax, (%rsi)`

`*up->t2.p`	`char`	`movq 8(%rdi), %rax`
		`movb (%rax), %al`
		`movb %al, (%rsi)`

Exercise 3.44. For each of the following structure declarations, determine the offset of each field, the total size of the structures, and its alignment requirement for x86-64:

(a) `struct P1 { int i; char c; int j; char d; };`

(b) `struct P2 { int i; char c; char d; long j; };`

(c) `struct P3 { short w[3]; char c[3]; };`

(d) `struct P4 { short w[5]; char *c[3]; };`

(e) `struct P5 { struct P3 a[2]; struct P2 t; };`

Solution:

(a) For `struct P1`, the `int i` has 0 offset, the `char c` has offset 4, given that `i` takes up 4 bytes. The `int j` field has alignment 8, where 3 bytes of padding are added right after `char c` to ensure it has a proper 4-byte element. Finally, `char d` has offset 12, given that `int j` has an offset of 8 bytes and its size is 4 bytes. Since the last field is 1 byte, this would suggest a total of 13 bytes, but to satisfy alignment requirements in arrays, a total of 3 bytes of adding is added at the end. Therefore, `struct P1` has 16 bytes total. It has 4 byte alignment to satisfy the

(b) By a similar reasoning, in `struct P2`, the field `int i` has offset 0, `char c` has offset 4, `char d` has offset 5, `long j` has offset 8. Altogether, the structure takes up 16 bytes, with 2 bytes of padding between `char d` and `long j`. It requires 8 byte alignment to satisfy the restriction of all of it fields.

(c) The `short w[3]` field in `struct P3` has offset 0. Since `short` is 2 bytes, it takes up 6 bytes, so the `char c[3]` has an offset of 6 bytes, ensuring that the alignment for `w` is satisfied. Since `char c[3]` takes up 3 bytes, this would imply 9 byte alignment, but to ensure the alignment restrictions are satisfied by fields of `struct P3` when packed into an array, we need 1 byte of padding to achieve 2-byte alignment. Therefore, `struct P3` takes up 10 bytes overall, with 1 byte of padding at the end. It has 2 byte alignment to satisfy the restriction of all of its elements.

(d) In `struct P4`, the field `short w[5]` is at offset 0, and it takes up 10 bytes. Since `char *c[3]` has `char *` elements, each taking up 8 bytes, the overall structure require 8 byte alignment. Therefore, 6 bytes of padding are added after `short w[5]`, so that the offset of `char *c[3]` is 16 bytes. Since `c` has 3 elements, it takes up 24 bytes, and hence the overall structure takes up 40 bytes. Its alignment is 8 bytes.

(e) Since `P3` takes up 10 bytes, the field `struct P3 a[2]` takes up 20 bytes. But `P2` takes up 16 bytes, and due to its field it has an 8-byte alignment restriction. Therefore, 4 bytes of padding are added so that `struct P2 t` has offset 24 bytes. Overall, the structure takes up 40 bytes and has 8 byte alignment.

Exercise 3.45. Answer the following for the following declaration

```
struct {
    char   *a;
    short   b;
    double  c;
    char    d;
    float   e;
    char    f;
    long    g;
    int     h;
} rec;
```

(a) What are the byte offsets of all the fields in the structure?

(b) What is the total size of the structure?

(c) Rearrange the fields of the structure to minimize wasted space, and then show the byte offsets and total size for the rearranged structure.

Solution:

(a) `char *a` has byte offset 0 and takes up 8 bytes. `short b` has byte offset 8 and takes up 2 bytes. It requires 6 bytes of padding to ensure `double c`, the next field, has a required alignment of 8 bytes. Therefore, `double c` has a 16 byte offset and takes up 8 bytes. The next field `char d` has 24 byte offset and takes up 1 byte. For properly aligning the next field, `float e` taking up 4 bytes, 3 bytes of padding are needed. Therefore `float e` has 28 byte offset. The next field, `char f`, has 32 byte offset. Since takes up 1 byte is followed by an 8 byte `long`, it requires 7 bytes of padding. Hence, `long g` has a 40 byte offset, and lastly, `int h` has a 48 byte offset.

(b) Since `int h` takes up 4 bytes, we require 4 bytes of padding at the end to ensure that an array of these structures satisfies the byte alignment for all of it fields. Therefore, the structure takes up 56 bytes.

(c) We can re-arrange the fields and compute the new offsets:

```
struct {
    char   *a; // offset 0
    double  c; // offset 8
    long    g; // offset 16
    float   e; // offset 24
    int     h; // offset 28
    short   b; // offset 32
    char    d; // offset 34
    char    f; // offset 35
};
```

The offsets are shown above. There is no padding in between any of the offsets, and the sum of the sizes of the fields is 36 bytes. Since the structure has fields that require 8 byte alignment, we need to add 4 bytes of padding, making the structure 40 bytes in size.

Exercise 3.46. The C code below: shows a (low-quality) implementation of a function that reads a line from standard input, copies the string to newly allocated storage, and returns a pointer to the result:

```
/* This is very low-quality code
   It is intended to illustrate bad programming practices.
   See Practice Problem 3.46 */
char *get_line()
{
    char buf[4];
    char *result;
    gets(buf);
    result = malloc(strlen(buf));
    strcpy(result, buf);
    return result;
}
```

The disassembly up through call to `gets` is below:

```
//  char *get_line()
0000000000400720 <get_line>:
    400720: 53                  push   %rbx
    400721: 48 83 ec 10         sub    $0x10,%rsp
//  Diagram stack at this point
    400725: 48 89 e7            mov    %rsp,%rdi
    400728: e8 73 ff ff ff      callq  4006a0 <gets>
// Modify diagram to show stack content at this point
```

Consider the following scenario. Procedure `get_line` is called with the return address equal to `0x400776` and register `%rbx` equal to `0x0123456789ABCDEF`. You type the string

`0123456789012345678901234`

The program terminates with a segmentation fault. You run `gdb` and determine that the error occurs during the execution of the `ret` instruction of `get_line`.

(a) Fill in the diagram that follows, indicating as much as you can about the stack just after executing the instruction in line 3 in the disassembly. Label the quantities stored on the stack (e.g., "Return address") on the right, and their hexadecimal values (if known) within the box. Each box represents 8 bytes. Indicate the position of `%rsp`. Recall that the ASCII codes for characters `0-9` are `0x30-0x39`.

00 00 00 00 00 40 00 76	Return Address

(b) Modify your diagram to show the effect of the call to `gets` (line 5).

(c) To what address does the program attempt to return?

(d) What register(s) have corrupted value(s) when `get_line` returns?

(e) Besides the potential for buffer overflow, what two other things are wrong with the code for `get_line`?

Solution:

(a) When `get_line` is called, the return address of its caller is at the top of the stack, as indicated by the first entry in the diagram. The first instruction `pushq %rbx` places the contents of `%rbx` on the top of the stack. The next instruction, `$0x10`, allocates `16` bytes of space on the stack, since hexadecimal `0x10` has decimal value 16. At this point, there may have been a pre-existing value we do not know about.

00 00 00 00 00 40 00 76	Return Address
01 23 45 67 89 AB CD EF	Value of `%rbx`
	Unused
	Address of `buf` (top of stack `%rsp`)

(b) The instruction `mov %rsp, %rdi` on line 4 indicates that the address of `buf` is at the top of the stack, and passed as an argument to `gets`. Since `gets` reads in the string

`0123456789012345678901234`

which is 25 bytes long, plus 1 extra byte due to the null terminator character used in C strings, we find that the space allocated for `buf` is exceeded. The diagram looks as follows now:

00 00 00 00 00 40 00 34	Return Address
33 32 31 30 39 38 37 36	%rbx, overwritten
35 34 33 32 31 30 39 38	Next 8 bytes in buf (8 through 9, 0 through 5)
37 36 35 34 33 32 31 30	Address of buf (top of stack %rsp)

Note that %rbx was completely overwritten, and so were the first two bytes of the return address.

(c) The return address is now 0x0000000000400034.

(d) Register %rbx was completely overwritten with the hexadecimal representation of characters 67890123 from the input string. Since this is a callee-saved register, the get_line function will attempt to restore it right before it returns, and hence it will restore a value different than what the caller expects.

The first byte in register %rsp was overwritten with 0x34, corresponding to the last input 4, and the second byte 0x00 was overwritten by the same value 0x00 corresponding to the string null terminator.

(e) It does not provide the extra character necessary for the null terminator; that is, it should be strlen(buf) + 1 It does not check the return value of gets, which may be NULL. The code does not check whether malloc was successful. When malloc does not execute successfully (may there is no more memory available), it returns NULL. Passing this to strcpy will result in a segmentation fault as it tries to dereference the point while copying bytes.

Exercise 3.47. Running our stack-checking code 10,000 times on a system running Linux version 2.6.16, we obtained addresses ranging from a minimum of 0xffffb754 to a maximum of 0xffffd754.

(a) What is the approximate range of addresses?

(b) If we attempted to overrun with a 128-byte nop slep, about how many attempts would it take to test all starting addresses?

Solution:

(a) Since 0xb754 is 46,932 decimal and 0xd754 is 55,124 Subtracting the addresses yields around 0x2000, which is around $2 \cdot 16^3 = 2^{13} = 8192$.

(b) Since $128 = 2^7$, and a nop-slep essentially starts goes from addresses 0 through 127 in the first attempt, then 128 through 255, and so on, we essentially have $\frac{2^{12}}{2^7} = 2^6 = 64$. It would take 64 attempts to test all starting addresses.

Exercise 3.48. The function intlen, len, and iptoa provide a very convoluted way to compute the number of decimal digits required to represent an integer. We will use this as a way to study some of aspects of the gcc stack protector facility.

```
int len(char *s) {
    return strlen(s);
}
```

```c
void iptoa(char *s, long *p) {
    long val = *p;
    sprintf(s, "%ld", val);
}

int intlen(long x) {
    long v;
    char buf[12];
    v = x;
    iptoa(buf, &v);
    return len(buf);
}
```

The following show portions of the code for `inlen`, compiled both with and without stack pointer protector. First, without protector:

```
    // int intlen(long x)
    // x in %rdi
intlen:
    subq   $40,       %rsp
    movq   %rdi,      24(%rsp)
    leaq   24(%rsp),  %rsi
    movq   %rsp,      %rdi
    call   iptoa
```

With protector:

```
    int intlen(long x)
    x in %rdi
intlen:
    subq   $56,    %rsp
    movq   %fs:40, %rax
    movq   %rax,   40(%rsp)
    xorl   %eax,   %eax
    movq   %rdi,   8(%rsp)
    leaq   8(%rsp), %rsi
    leaq   16(%rsp), %rdi
    call iptoa
```

(a) For both versions: What are the positions in the stack for `buf`, `v`, and (when present) the canary value?

(b) How does the rearranged ordering of the local variables in the protected code provide greater security against a buffer overrun attack?

Solution:

(a) For the version without stack protector, the first instruction `subq $40, %rsp` indicates we are allocating 40 bytes on the stack. The `movq %rdi, 24(%rsp)` is the assignment of`x` to `v`.

Therefore, `v` is as at an offset of 24 bytes from the top of the stack. The line `movq %rsp, %rdi` indicates that we are passing `buf` as the first argument of `iptoa`. This indicates that `buf` is at the top of the stack.

For the version with stack protector, space is allocated on the stack for 56 bytes. In this case, the canary value by the `movq %fs:40, %rax` instruction into the `%rax` register, and its value is loaded into the memory location at a 40 byte offset of the top of the stack, as indicated by `movq %rax, 40(%rsp)`. In this case, `leaq 16(%rsp), %rdi` indicates that buf is at a 16 byte offset from the top of the stack, and `leaq 8(%rsp), %rsi` indicates that `v` is at an 8 byte offset from the top of the stack.

(b) Between the two versions, the location of `buf` and `r` relative to the top of the stack was changed. In particular, `r` is closer to the top of the stack in the version with the stack protector. This means that if `buf` overflows, the value of `v` will not be overwritten. Moreover, `buf` is followed by the canary value. If the canary value is overwritten due to a buffer overflow, the changed value in the canary will be picked up by the compiler, which will flag it as an error and abort.

Exercise 3.49. In this problem, we will explore the logic behind the code in lines 5–11 on Figure 3.43(b), where space is allocated for variable-size array `p`. As the annotations on the code indicate, let s_1 denote the address of the stack pointer after executing the `subq` instruction of line 4. This instruction allocates space for local variable `i`. Let s_2 denote the value of the stack pointer after executing the `subq` instruction on line 7. This instruction allocates the storage for local array `p`. Finally, let p denote the value assigned to registers `%r8` and `%rcx` in the instructions of lines 10–11. Both of these registers are used to reference array `p`.

The right-hand side of Figure 3.44 diagrams the positions of the locations indicated by s_1, s_2, and p. It also shows that there may be an offset of e_2 bytes between the values of s_1 and p. This space will not be used. There may also be an offset of e_1 bytes between the end of array `p` and the position indicated by s_1.

(a) Explain, in mathematical terms, the logic in the computation of s_2 on lines 5–7. *Hint*: Think about the bit-level representation of -16 and its effect in the `andq` instruction of line 6.

(b) Explain, in mathematical terms, the logic in the computation of p on lines 8–10. *Hint*: You may want to refer to the discussion on division by powers of 2 in Section 2.3.7.

(c) For the following values of n and s_1, trace the execution of the code to determine what the resulting values would be for s_2, p, e_1, and e_2.

n	s_1	s_2	p	e_1	e_2
5	$2,065$	_____	_____	_____	_____
6	$2,064$	_____	_____	_____	_____

(d) What alignment properties does this guarantee for the values of s_2 and p?

Solution:

(a) The location s_1 is where the address of the stack pointer after allocating space for i. The instruction on line 5 `leaq 22(,%rdi, 8), %rax` uses the value `n` is in register `%rdi` to store the value $x = 22 + 8n$ in register `%rax`. The instruction `andq $-16, %rax` on line 6 then computes the x AND -16. Note that $-16 = -1 \cdot 16$, which is equivalent to `-1 << 4`. Since

-1 has all 1s in its binary representation, it follows from the left shift operation that -16 has all 1s except for 4 zeroes in its least significant bits. Therefore, the effect when involved in the AND operation is to remote the least significant byte. Put another way, suppose $r = x \mod 16$. Then the instruction on line 16 subtracts the remainder r from x and places it in `%rax`. Now `%rax` has value $y = x - r$ in its register. Note that

$$\begin{aligned} r &= x \mod 16 \\ &= (22 + 8n) \mod 16 \\ &= (6 + 8n) \mod 16 \end{aligned}$$

If n is even, then $n = 2k$, and in that case we get

$$r = (6 + 8n) \mod 16 = (6 + 16k) \mod 16 = 6$$

If n is odd, then $n = 2k + 1$ for some k, and we have

$$r = (6 + 16k + 8) \mod 16 = (14 + 16k) \mod 16 = 14$$

Therefore, r is 6 or 14. Since $x = 22 + 8n$, this means that

$$y = \begin{cases} 16 + 8n & \text{if } n \text{ is even} \\ 8 + 8n & \text{if } n \text{ is odd} \end{cases}$$

Now, the instruction `subq %rax, %rsp` on line 7 subtracts $y = x$ from s_1, since s_1 is the current value on the stack, to compute s_2. Hence, the new value on top of the stack is

$$s_2 = \begin{cases} s_1 - (16 + 8n) & \text{if } n \text{ is even} \\ s_1 - (8 + 8n) & \text{if } n \text{ is odd} \end{cases}$$

(b) The instruction `leaq 7(%rsp), %rax` on line 8 and the instruction `shrq $3, %rax` takes the value s_2 in register `%rax` and replaces it with $(s_2 + 7) >> 3 = (s_2 + (1 << 3) - 1) >> 3$, which is the computation for dividing a two's complement number by $2^3 = 8$. This computes $s_2/8$ and rounds up. The instruction `leaq 0(,%rax,8), %r8` on line 10 multiplies this value by 8. Now `%r8` has value

$$p = \begin{cases} 8\lceil s_1/8 \rceil - (16 + 8n) & \text{if } n \text{ is even} \\ 8\lceil s_1/8 \rceil - (8 + 8n) & \text{if } n \text{ is odd} \end{cases}$$

The result is that the address in p is now a multiple of 8, ensuring its aligned on 8 bytes, as required by its elements of type `long *`. Moreover, it is greater than s_2, so it is above s_2. In short, it is the address closes to s_2 that is a multiple of 8.

(c)

n	s_1	s_2	p	e_1	e_2
5	$2,065$	2017	2024	1	7
6	$2,064$	2000	2000	16	0

(d) p is aligned on 8 bytes with respect to `%rbp` consistent with the fact that `long *` requires 8 byte alignment. Also, s_2 is aligned on 16 bytes with respect to s_1.

Exercise 3.50. For the following C code, the expressions `val1`–`val4` all map to the program values `i`, `f`, `d`, and `l`:

```
double fvct2(int *ip, float *fp, double *dp, long l)
{
    int i = *ip; float f = *fp; double d = *dp;
    *ip = (int)    val1;
    *fp = (float)  val2;
    *dp = (double) val3;
    return (double) val4;
}
```

Determine the mapping, based on the following x86-64 code for the function:

```
    double fvct2(int *ip, float *fp, double *dp, long l)
    ip in %rdi, fp in %rsi, dp in %rdx, l in %rcx
    Result returned in %xmm0
fcvt2:
    movl       (%rdi),    %eax
    vmovss     (%rsi),    %xmm0
    vcvttsd2si (%rdx),    %r8d
    movl       %r8d,      (%rdi)
    vcvtsi2ss  %eax,      %xmm1,  %xmm1
    vmovss     %xmm1,     (%rsi)
    vcvtsi2sdq %rcx,      %xmm1,  %xmm1
    vmovsd     %xmm1,     (%rdx)
    vunpcklps  %xmm0,     %xmm0,  %xmm0
    vcvtps2pd  %xmm0,     %xmm0
    ret
```

Solution: We begin by annotating the assembly:

```
    double fvct2(int *ip, float *fp, double *dp, long l)
    ip in %rdi, fp in %rsi, dp in %rdx, l in %rcx
    Result returned in %xmm0
fcvt2:
    movl       (%rdi),    %eax            // int i = *ip
    vmovss     (%rsi),    %xmm0           // float f = *fp;
    vcvttsd2si (%rdx),    %r8d            // Convert *dp to an integer
    movl       %r8d,      (%rdi)          // *ip = (int) d;
    vcvtsi2ss  %eax,      %xmm1,  %xmm1   // Convert i to single-precision float
    vmovss     %xmm1,     (%rsi)          // *fp = (float) i;
    vcvtsi2sdq %rcx,      %xmm1,  %xmm1   // Convert l to double-precision float
    vmovsd     %xmm1,     (%rdx)          // *dp = (float) l;
    vunpcklps  %xmm0,     %xmm0,  %xmm0   //
    vcvtps2pd  %xmm0,     %xmm0           // Extend *fp to a double
    ret                                   // return (double) f;
```

The C code now looks like:

```
double fvct2(int *ip, float *fp, double *dp, long l)
{
    int i = *ip; float f = *fp; double d = *dp;
    *ip = (int)    d;
    *fp = (float)  i;
    *dp = (double) l;
    return (double) f;
}
```

Exercise 3.51. The following C function converts an argument of type `src_t` to a return value of type `dst_t`, where these two types are defined using `typedef`:

```
dest_t cvt(src_t x)
{
    dest_t y = (dest_t) x;
    return y;
}
```

For an extension on x86-64, assume that argument `x` is either in `%xmm0` or in the appropriately named portion of register `%rdi` (i.e., `%rdi` or `%edi`). One or two instructions are to be used to perform the type conversion and to copy the value to the appropriately named portion of register `%rax` (integer result) or `%mm0` (floating-point result). Show the instruction(s), including the source and destination registers.

T_x	T_y	Instruction(s)
`long`	`double`	`vcvtsi2sdq %rdi, %mm0`
`double`	`int`	________
`double`	`float`	________
`long`	`float`	________
`float`	`long`	________

Solution:

T_x	T_y	Instruction(s)
`long`	`double`	`vcvtsi2sdq %rdi, %xmm0, %xmm0`
`double`	`int`	`vcvttsd2si %xmm0, %eax`
`double`	`float`	`vcvtsd2ss %xmm0, %xmm0, %xmm0`
`long`	`float`	`vcvtsi2sdq %rdi, %xmm0, %xmm0`
`float`	`long`	`vcvttss2siq %xmm0, %rax`

Exercise 3.52. For each of the following function declarations, determine the register assignments for the arguments:

(a) `double g1(double a, long b, float c, int d);`

(b) `double g2(int a, double *b, float *c, long d);`

(c) `double g3(double *a, double b, int c, float d);`

(d) `double g4(float a, int *b, float c, double d);`

Solution:

(a) `a` in `%xmm0`, `b` in `%rdi`, `c` in `%xmm1`, and `d` in `%esi`.

(b) `a` in `%edi`, `b` in `%xmm0`, `c` in `%rsi`, and `d` in `%rdx`.

(c) `a` in `%rdi`, `b` in `%xmm0`, `c` in `%esi`, and `d` in `%xmm1`.

(d) `a` in `%xmm0`, `b` in `%rdi`, `c` in `%xmm1`, and `d` in `%xmm2`.

Exercise 3.53. For the following C function, the types of the four arguments are defined by `typdef`:

```c
double funct1(arg1_t p, arg2_t q, arg3_t r, arg4_t s)
{
    return p/(q+r) - s;
}
```

When compiled, `gcc` generates the following code:

```
    double funct1(arg1_t p, arg2_t q, arg3_t r, arg4_t s)
funct1:
    vcvtsi2ssq    %rsi,  %xmm2, %xmm2
    vaddss        %xmm0, %xmm2, %xmm0
    vcvtsi2ss     %edi,  %xmm2, %xmm2
    vdivss        %xmm0, %xmm2, %xmm0
    vunpcklps     %xmm0, %xmm0, %xmm0
    vcvtps2pd     %xmm0, %xmm0
    vsubsd        %xmm1, %xmm0, %xmm0
    ret
```

Determine the possible combinations of types of the four arguments (there may be more than one).

Solution: The annotated assembly is:

```
    double funct1(arg1_t p, arg2_t q, arg3_t r, arg4_t s)
funct1:
    vcvtsi2ssq    %rsi,  %xmm2, %xmm2  // Convert long to float
    vaddss        %xmm0, %xmm2, %xmm0  // Add two floats
    vcvtsi2ss     %edi,  %xmm2, %xmm2  // Convert int to float
    vdivss        %xmm0, %xmm2, %xmm0  // Divide two floats
    // The next two instructions convert float to double
    vunpcklps     %xmm0, %xmm0, %xmm0
    vcvtps2pd     %xmm0, %xmm0

    vsubsd        %xmm1, %xmm0, %xmm0  // Subtract two doubles
    ret
```

From the last instruction, we infer that `s` is in the `%xmm1` register, since it is subtracted from the value computed so far in `%xmm0`. Since it has undergone no conversions so far, we conclude that `arg4_t` is `double`.

From the vdivss instruction we see that %xmm2 must hold the numerator p of the division. The instruction right before is one that converts an integer stored in %edi to a float. Therefore, we conclude that p must be a 32-bit integer (an int).

Based on the remaining annotations, the two remaining arguments are of type long and float, because they involve the %xmm0 register and the %rsi register. Therefore, arg2_t is a float and arg3_t is a long, or viceversa.

Exercise 3.54. Function funct2 has the following prototype:

```
double funct2(double w, int x, float y, long z);
```

gcc generates the following for the function:

```
    double funct2(double w, int x, float y, long z);
    w in %xmm0, x in %edi, y in %xmm1, z in %rsi
funct2:
    vcvtsi2ss     %edi,  %xmm2, %xmm2
    vmulss        %xmm1, %xmm2, %xmm1
    vunpcklpls    %xmm1, %xmm1, %xmm1
    vcvtps2pd     %xmm1, %xmm2
    vcvtsi2sdq    %rsi,  %xmm1, %xmm1
    vdivsd        %xmm1, %xmm0, %xmm0
    vsubsd        %xmm0, %xmm2, %xmm0
    ret
```

Write a C version of funct2.

Solution: I have annotated the assembly below:

```
    double funct2(double w, int x, float y, long z);
    w in %xmm0, x in %edi, y in %xmm1, z in %rsi
funct2:
    vcvtsi2ss     %edi,  %xmm2, %xmm2  // Convert integer to float: (float) x
    vmulss        %xmm1, %xmm2, %xmm1  // y *= (float) x
    // Convert y from float to double
    vunpcklpls    %xmm1, %xmm1, %xmm1
    vcvtps2pd     %xmm1, %xmm2         // double s = y

    vcvtsi2sdq    %rsi,  %xmm1, %xmm1  // double r = z (long -> double)
    vdivsd        %xmm1, %xmm0, %xmm0  // w /= r;
    vsubsd        %xmm0, %xmm2, %xmm0  // return s - r;
    ret
```

The corresponding C code is below:

```
double funct2(double w, int x, float y, long z)
{
    double s = y * (float) x; // Convert y to float
    double r = w / z;
    return s - r;
```

```
}
```

Exercise 3.55. Show how the numbers declared at label `.LC3` encode the number `32.0`.

Solution: At `.L3` we have:

```
.LC3:
    .long   0               // Low-order 4 bytes of 32.0
    .long   1077936128      // High-order 4 bytes of 32.0
```

Converting `1077936128` by dividing by 16 yields:

$$
\begin{aligned}
1077936128 &= 16 \cdot 67371008 + 0 \\
67371008 &= 16 \cdot 4210688 + 0 \\
4,210,688 &= 16 \cdot 263168 + 0 \\
263,168 &= 16 \cdot 16488 + 0 \\
16488 &= 16 \cdot 1028 + 0 \\
1024 &= 16 \cdot 64 + 4 \\
64 &= 16 \cdot 4 + 0 \\
4 &= 16 \cdot 0 + 4
\end{aligned}
$$

The corresponding number in hex is `0x40400000`. The `0x404` is equivalent to binary `0100 0000 0100`. The leading bit is the sign bit, and 0 indicates a positive number, which is true for `32.0`. The remaining 11 bits yield the number `1028`. Subtracting the bias `1023` yields an exponent of `5`. Since the exponent is nonzero, the number is a normalized floating point, so the significand is $M = 1 + f$, where f is the fractional part. Since the rest of the bits in the binary sequence are 0, this means that $f = 0$ and $M = 1$. The value is therefore $V = 2^E \cdot M = 2^5 \cdot 1 = 32$.

Exercise 3.56. Consider the following C function, where `EXPR` is a macro defined with a `#define`:

```c
double simplefun(double x) {
    return EXPR(x);
}
```

Below, we show the AVX2 code generated for different definitions of `EXPR`, where value `x` is held in `%xmm0`. All of them correspond to some useful operation on floating-point values. Identify what the operations are. Your answers will require you to understand the bit patterns of the constant words being retrieved from memory.

(a)

```
    vmovsd .LC1(%rip),    %xmm1
    vandpd %xmm1,         %xmm0   %xmm0
.LC1:
    .long   4294967295
    .long   2147483647
    .long   0
    .long   0
```

(b)

```
    vxorpd %xmm0, %xmm0, %xmm0
```

(c)

```
    vmovsd .LC2(%rip),    %xmm1
    vxorpd %xmm1,         %xmm0, %xmm0
.LC2:
    .long  0
    .long  -2147483648
    .long  0
    .long  0
```

Solution:

(a) The `4294967295` is $2^{32}-1$, and corresponds to hex value `ff ff ff ff`. Meanwhile `2147483647` is $2^{31}-1$, and corresponds to hex value `7f ff ff ff`. The latter is the higher-order 4 bytes of a double precision number. The three leading bytes `0x7ff` correspond to binary `0111 1111 1111`. The sign bit is 0, and the remaining bits are 1. Since the lower ordered bytes (for the fractional part) are not 0, this number must be `NaN`. Noting the `vandpd` instruction, we must have `NaN & x`. The value of the expression is a bit pattern that is the same as `x`, except that the most significant bit is 0. Therefore it returns the absolute value of `x`.

(b) This computes `x ^ x`, which gives 0.

(c) The number `-2147483648` has hex representation `80 00 00 00`. Hence, the binary representation of the leading three bytes `0x800` is `1000 0000 0000`. This has a sign bit of 1, and all exponent bits are 0. The rest of the bits in the pattern are also 0. Since the operation is for `vxorpd`, which computes exclusive or, this has the effect of flipping the leading bit (since `1^ 1` is 0 and `1^ 0` i s 1), and retaining the rest of the bits. This calculation negates `x`.

Exercise 3.57. Function `funct3` has the following prototype:

```c
double funct2(int *ap, double b, long c, float *dp);
```

For this function, `gcc` generates the following code:

```
    double funct3(int *ap, double b, long c, float *dp)
    ap in %rdi, b in %xmm0, c in %rsi, dp in %rdx
funct3:
    vmovss     (%rdx),    %xmm1
    vcvtsi2sd  (%rdi),    %xmm2, %xmm2
    vucomisd   %xmm2,     %xmm0
    jbe        .L8
    vcvtsi2ssq %rsi,      %xmm0, %xmm0
    vmulss     %xmm1,     %xmm0, %xmm1
    vunpcklps  %xmm1,     %xmm1, %xmm1
    vcvtps2pd  %xmm1,     %xmm0
```

```
    ret
.L8:
    vaddss     %xmm1,     %xmm1, %xmm1
    vcvtsi2ssq %rsi,      %xmm0, %xmm0
    vaddss     %xmm1,     %xmm0, %xmm0
    vunpcklps  %xmm0,     %xmm0, %xmm0
    vcvtps2pd  %xmm0,     %xmm0
    ret
```

Write a C version of funct3.

Solution: The annotated version of the assembly is:

```
    double funct3(int *ap, double b, long c, float *dp)
    ap in %rdi, b in %xmm0, c in %rsi, dp in %rdx
funct3:
    vmovss     (%rdx),    %xmm1          // float d = *dp;
    vcvtsi2sd  (%rdi),    %xmm2, %xmm2   // double v = *ap;
    vucomisd   %xmm2,     %xmm0          // Compare b:v
    jbe        .L8                       // if <= go to .L8
    vcvtsi2ssq %rsi,      %xmm0, %xmm0   // float r = c;
    vmulss     %xmm1,     %xmm0, %xmm1   // d *= r;
    vunpcklps  %xmm1,     %xmm1, %xmm1
    vcvtps2pd  %xmm1,     %xmm0
    ret                                  // return d;
.L8:
    vaddss     %xmm1,     %xmm1, %xmm1   // d += d; // double d
    vcvtsi2ssq %rsi,      %xmm0, %xmm0   // float z = c;
    vaddss     %xmm1,     %xmm0, %xmm0   // z += d;
    vunpcklps  %xmm0,     %xmm0, %xmm0
    vcvtps2pd  %xmm0,     %xmm0
    ret                                  // return z;
```

The corresponding C code is below:

```
double funct3(int *ap, double b, long c, float *dp)
{
    float d = *dp;
    double v = *ap;
    if (b > v) {
        return c * d
    } else {
        return 2 * d + c;
    }
}
```