

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 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 |
|------------------------------|-------|
| <code>%rax</code> | 0x100 |
| <code>0x104</code> | 0xAB |
| <code>\$0x108</code> | 0x108 |
| <code>(%rax)</code> | 0xFF |
| <code>4(%rax)</code> | 0xAB |
| <code>9(%rax, %rdx)</code> | 0x11 |
| <code>260(%rcx, %rdx)</code> | 0x13 |
| <code>0xFC(,%rcx,4)</code> | 0xFF |
| <code>(%rax,%rdx,4)</code> | 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 an 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, it 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 indicated 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 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 “signedness” 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 signedness. 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 | <code>movq (%rdi), %rax</code> <code>movq %rax, (%rsi)</code> |
| char | int | <code>movsbl (%rdi), %eax</code> <code>movl %eax, (%rsi)</code> |
| char | unsigned | <code>movsbl (%rdi), %eax</code> <code>movl %eax, %rsi</code> |
| unsigned char | long | <code>movzbl (%rdi), %rax</code> <code>movq %rax, (%rsi)</code> |
| int | char | <code>movb (%rdi), %al</code> <code>movb %al, (%rsi)</code> |
| unsigned | unsigned char | <code>movb (%rdi), %al</code> <code>movb %al, (%rsi)</code> |
| char | short | <code>movsbw (%rdi), %ax</code> <code>movw %ax, (%rsi)</code> |

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 reference 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 |
|--|--------|
| <code>leaq 6(%rax), %rdx</code> | _____ |
| <code>leaq (%rax,%rcx), %rdx</code> | _____ |
| <code>leaq (%rax,%rcx,4), %rdx</code> | _____ |
| <code>leaq 7(%rax,%rax,8), %rdx</code> | _____ |
| <code>leaq 0xA(,%rcx,4), %rdx</code> | _____ |
| <code>leaq 9(%rax,%rcx,2), %rdx</code> | _____ |

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 |
|--|-----------------------|
| <code>leaq 6(%rax), %rdx</code> | $x + 6$ |
| <code>leaq (%rax,%rcx), %rdx</code> | $x + y$ |
| <code>leaq (%rax,%rcx,4), %rdx</code> | $x + 4y$ |
| <code>leaq 7(%rax,%rax,8), %rdx</code> | $7 + x + 8x = 9x + 7$ |
| <code>leaq 0xA(,%rcx,4), %rdx</code> | $10 + 4y$ |
| <code>leaq 9(%rax,%rcx,2), %rdx</code> | $9 + x + 2y$ |

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 | <code>%rax</code> | 0x100 |
| 0x108 | 0xAB | <code>%rcx</code> | 0x1 |
| 0x110 | 0x13 | <code>%rdx</code> | 0x3 |
| 0x118 | 0x11 | | |

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

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

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 |
|--|-------------------|-------|
| <code>addq %rcx, (%rax)</code> | 0x100 | 0x100 |
| <code>subq %rdx, 8(%rax)</code> | 0x108 | 0xA8 |
| <code>imulq \$16, (%rax,%rdx,8)</code> | 0x118 | 0x110 |
| <code>incq 16(%rax)</code> | 0x110 | 0x14 |
| <code>decq %rcx</code> | <code>%rcx</code> | 0x0 |
| <code>subq %rdx, %rax</code> | <code>%rax</code> | 0xFD |

Exercise 3.9. Suppose we want to generate assembly code for the following C function:

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

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

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

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

Solution:

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

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

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

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

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

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

Solution: The instruction `orq %rsi, %rdi` calculates $t1 = x \mid y$. The instruction `sarq $3, %rdi` calculates $t2 = t1 \gg 3$, or equivalently $t2 = 8 * t1$. The instruction `notq %rdi` calculates $t3 = \sim t2$. 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.

- Explain the effect of this particular *exclusive-OR* instruction and what useful operation it implements.
- What would be the more straightforward way to express this operation in assembly code?

- (c) Compare the number of bytes to encode these two different implementations of the same operation.

Solution:

- (a) Noting that $0 \wedge 0$ and $1 \wedge 1$ are both 0, the resulting operation is to yield a value 0 and place it in `%rdx`. This is essentially zeroing the register.
- (b) The straightforward way to write this would be `imul $0, %rdx`, or `movq $0 %rdx`.
- (c)

Exercise 3.12. Consider the following function for computing the quotient and remainder of two unsigned 64-bit numbers:

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

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

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

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

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

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

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

Exercise 3.13. The C code

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

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

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

- (a) `cmpl %esi, %edi`
`setl %al`
- (b) `cmpw %si, %di`
`setge %al`
- (c) `cmpb %sil, %dil`
`setbe %al`
- (d) `cmpq %rsi, %rdi`
`setne %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 `setne` 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 | <code>je</code> | XXXXXX |
| 4003fc: ff d0 | <code>callq</code> | *%rax |

- (b) What is the target of the `je` instruction below?

| | | |
|---------------|------------------|--------|
| 40042f: 74 f4 | <code>je</code> | XXXXXX |
| 400431: 5d | <code>pop</code> | %rbp |

- (c) What is the address of the `ja` and `pop` instructions?

| | | |
|---------------|------------------|-------------------|
| XXXXXX: 77 02 | <code>ja</code> | 400547 |
| XXXXXX: 5d | <code>pop</code> | <code>%rbp</code> |

- (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 | <code>jmp</code> | XXXXXX |
| 4005ed: 90 | <code>nop</code> | XXXXXX |

Solution:

- (a) The `0x02` in `74 02` must be added to `4003fc`, the address of the following instruction, to yield `4003fe` as the jump target of `je`.
- (b) The `f4` in `74 f4` must be added to `400431`, the address of the next instruction. Since `f4` is a single byte has decimal value -12 (in two's complement), we subtract 12 from the hex address `400431` and obtain `400425`.
- (c) Adding `0x02` from `77 02` to `XXXXXX` should give `400547`, and since `0x02` is decimal 2, we get that the `XXXXXX` that follows the jump instruction must be `400545`. Since the instruction `5d` at that location is 2 bytes after the first instruction in the `ja` line, we subtract 2 bytes to get `400543` for the byte address instruction of the `ja` line.
- (d) The `73 ff ff ff` is written in little-endian (least to most significant) bit, so we can re-write it as `ff ff ff 73` (from most to least significant). This is a negative number since the most significant bit is 1, and it is a sign extension of `01 73`. Since `01 00` is 256 and `00 73` is 115, this means we have $-256 + 115 = -141$. Therefore we add -141 to the following address, `4005ed`, to get the jump target address which. The `ed` portion 237, and $237 - 141 = 96$, or `0x60`. Therefore the jump target address is `400560`. has value

Exercise 3.16. When given the C code

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

gcc generates the following code:

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

```

    jge    .L1
    movq   %rdi, (%rsi)
.L1:
    rep; ret

```

- (a) Write a goto version in C that performs the same computation and mimics the control flow of the assembly code, in the style shown in Figure 3.16(b). You might find it helpful to first annotate the assembly code as we have done in our examples.
- (b) Explain why the assembly code contains two conditional branches even though the C code has only one if statement.

Solution:

- (a) The annotated assembly is below:

```

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

```

The goto version is below:

```

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

```

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

Exercise 3.17. An alternate rule for translating if statements into goto code is as follows:

```

t = test-expr;
if (t)

```

```

        goto true;
    else-statement
        goto done;
true:
    then-statement
done:

```

- (a) Rewrite the goto version of `absdiff_se` based on this alternate rule.
- (b) Can you think of any reasons for choosing one rule over the other?

Solution:

- (a)

```

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

```

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

Exercise 3.18. Starting with C code of the form

```

long test(long x, long y, long z) {
    long val = _____;
    if (_____) {
        if (_____)
            val = _____;
        else
            val = _____;
    } else if (_____)
        val = _____;
    return val;
}

```

gcc generates the following assembly code:

```

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

```

Fill in the missing expressions in the C code.

Solution: First we can annotate the assembly:

```

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

```



```

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

```

From this, the C code is

```

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

```

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

- (a) What is the appropriate miss penalty?
- (b) How many cycles would the function require when the branch was mispredicted?

Solution:

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

$$T_{avg}(p) = (1 - p)T_{OK} + p(T_{OK} + T_{MP}) = T_{OK} + pT_{MP}$$

$$T_{MP} = \frac{1}{p} (T_{avg}(p) - T_{OK})$$

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 $T_{OK} + T_{MP} = 46$ cycles.

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

```

#define OP _____ /* Unknown operator */

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

```

When compiled, gcc generated the following assembly code:

```

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

```

- (a) What operation is OP?
- (b) Annotate the code to explain how it works.

Solution:

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

- (b)

- ```

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

```
- 

**Exercise 3.21.** Starting with C code of the form

---

```

long test(long x, long y) {
 long val = _____;
 if (_____) {
 if (_____)
 val = _____;
 else
 val = _____;
 } else if (_____)
 val = _____;
 return val;
}

```

---

gcc generates the following assembly code:

---

```

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

```

---

```

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

```

---

Fill in the missing expressions in the C code.

**Solution:** We can first annotate the assembly:

---

```

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

```

---

This reveals that the C code is as follows:

---

```

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

```

```
 return val;
}
```

---

**Exercise 3.22.**

- (a) What is the maximum value of  $n$  for which we can represent  $n!$  with a 32-bit `int`?
- (b) What about a 64-bit `long`?

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