Sergio Garcia Tapia
Computer Systems: A Programmer's Perspective, by Bryant and O'Hallaron
Chapter 3: Machine-Level Representation of Programs March 23, 2024

Practice Problems

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Exercise 3.3. Each of the following lines of code generates an error when we invoke the assembler. Explain what is wrong with each line.

```
movb   $0xF,    (%ebx)
movl   %rax,    (%rsp)
movw  (%rax),  4(%rsp)
movb    %al,     %sl
movq   %rax,   $0x123
movl   %eax,     %dx
movq    %si,   8(%rbp)
```

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

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

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

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

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

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

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

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

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

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

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

Assume that the values of `sp` and `dp` are stored in registers `%rdi` and `%rsi`, respectively. For each entry in the table, show the two instructions that implement the specified data movement. The first instruction in the sequence should read from memory, do the appropriate conversion, and set the appropriate portion of register `%rax`. The second instruction should then write the appropriate portion of `%rax` to memory. In both cases, the portions may be `%rax, %eax, %ax,` or `%al`, and they may differ from one another. Recall that when performing a cast that involves a size change and a change of "signedness" in C, the operation should change the size first (Section 2.2.6).

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

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

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

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

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

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

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

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

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

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

```
    void decode1(long *xp, long *yp, long *zp)
    xp in \%rdi,
decode1:
    movq   (%rdi), %r8
    movq   (%rsi), %rcx
    movq   (%rdx), %rax
    movq   %r8,   (%rsi)
    movq   %rcx,  (%rdx)
    movq   %rax,  (%rdi)
    ret
```

Parameters `xp`, `yp`, and `zp` are stored in registers `%rdi`, `%rsi`, and `%rdx`, respectively. Write C code for `decode1` that will have an effect equivalent to the assembly code shown.

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

The program below implements the C equivalent:

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

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

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

Solution:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Solution:

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

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

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

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

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

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

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

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

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

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

in the code that was generated from C where no *exclusive OR* operations were present.

(a) Explain the effect of this particular *exclusive-OR* instruction and what useful operation it implements.

(b) What would be the more straightforward way to express this operation in assembly code?

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

Solution:

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

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

(c)

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

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

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

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

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

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

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

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