Exercise 1 - Solution

PROBLEM 1 (SEE PRACTICE PROBLEM 3.3)

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

```
(a)movw %(%rax), 4(%rsp, %rsi, 8)
(b)movb %al, %sl
(c)movq %rdi, $0x111
(d)movl %r8, (%rdx)
(e)movl %ecx, %rdx
(f)movb %si, 8(%rbp)
(g)movb $0xF, (%ebx)
```

(a) Cannot have both source and destination be memory references
(b) No register named %sl
(c) Cannot have immediate as destination
(d) Mismatch between instruction suffix and register ID
(e) Destination operand incorrect size
(f) Mismatch between instruction suffix and register ID
(g) Cannot use %**e**bx as address register

PROBLEM 2 (SEE PRACTICE PROBLEM 3.4)

Assume variables `sp` and `dp` are declared with types

```
src_t *sp;
```

```
dest_t *dp;
```

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

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

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 both 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	movsbl (%rdi), %eax movl %eax, (%rsi)
char	unsigned	movsbl (%rdi), %eax movl %eax, (%rsi)
unsigned char	long	movzbl (%rdi), %eax movq %rax, (%rsi)
int	char	movl (%rdi), %eax movb %al, (%rsi)
unsigned	unsigned char	movl (%rdi), %eax movb %al, (%rsi)
char	short	movsbw (%rdi), %ax movw %ax, (%rsi)

PROBLEM 3

Suppose a 64-bit little endian machine has the following memory and register status. Fill in the blanks using 8-byte value and in hex notation. NOTE: **Instructions are independent**.

Memory status:

Address	Value
0x100	0xf0f0f0f0
0x104	0x78563412
0x108	0x1000

Register status:

Register	Value
%rax	0x104
%rbx	0x1
%rcx	0xffffffff fffffffc
%rdx	0x87654321

Fill in the blanks below:

Operation	Destination	Value
subq (%rax), %rdx	%rdx	0x0f0f0f0f
imulq $2, (%rax, %rbx, 4)	0x110	0x2000
notq (%rax, %rcx)	0x100	0xffffffff 0f0f0f0f
leaq 8(%rax, %rbx, 4), %rdx	%rdx	0x118
movb $0x1, %al	%rax	0x101

PROBLEM 4

Indicate the status (0, 1 or unchanged) of the following flags after each instruction, please write "——" if the flag doesn't change. Assume 3 in %rax, -8 in %rbx. NOTE: **Each instruction works independently and would NOT affect each other**.

Instruction	OF	SF	ZF	CF
addq %rbx, %rax	0	1	0	0
subq %rax, %rbx	0	1	0	0
leaq (%rax, %rax, 2), %rax	--	--	--	--
xorq %rax, %rax	0	0	1	0
salq $2, %rbx	0	1	0	1
cmpq %rax, %rbx	0	1	0	0
testq %rax, %rbx	0	0	1	0

PROBLEM 5 (SEE PRACTICE PROBLEM 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 data types listed in Figure 3.1 and either signed or unsigned. The comparison `COMP` is defined via `#define`.

Suppose a is in some portion of `%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.

A.
```
cmpl %esi, %edi
setl %al
```
B.
```
cmpw %si, %di
setge %al
```
C.
```
cmpb %sil, %dil
setbe %al
```
D.
```
cmpq %rsi, %rdi
setne %al
```

A. The suffix 'l' and the register identifiers indicate 32-bit operands, while the comparison is for a two's-complement <. We can infer that `data_t` must be `int`.
B. The suffix 'w' and the register identifiers indicate 16-bit operands, while the comparison is for a two's-complement >=. We can infer that `data_t` must be `short`.
C. The suffix 'b' and the register identifiers indicate 8-bit operands, while the comparison is for an unsigned <=. We can infer that `data_t` must be `unsigned char`.

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