ICS 笔记（以章节划分）

第二章

一 • 基本概念

2.1.1 十六进制

- 1.字节：8 位的块，最小的可寻址的内存单位
- 2.虚拟：机器级程序将内存视为一个非常大的字节数组
- 3.地址：内存的每个字节都由 一个唯一的数字来标识
- 4.虚拟地址空间：所有可能地址的集合
- 5.程序对象：即程序数据、指令和控制信息
- 6.C 语言中一个指针的值都是某个存储块的第一个字节的虚拟地址。
- 7.十六进制表示法优点：二进制表示法太冗长，而十进制表示法与位模式的互相转化很麻烦
- 8.C 语言中，以 0x 或 0X 开头的数字常量被认为是十六进制的值

2.1.2 字数据大小

- 9.字长：处理器一次能处理的字节数，指明指针数据的标称大小，决定虚拟地址空间的最大大小（字长w ~虚拟地址2^w-1）
- 10.
  - 32 位机器：字长为 4 字节，虚拟地址空间为2^{32} ~4GB，经过伪指令 `linux> gcc -m32 prog.c` 编译后，就可在 32/64 位机器上运行
  - 64 位机器：字长为 8 字节，虚拟地址空间为2^{64} ~16EB，经过伪指令 `linux> gcc -m64 prog.c` 编译后，就只能在 64 位机器上运行

C声明		字节数	
有符号	无符号	32位	64位
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8

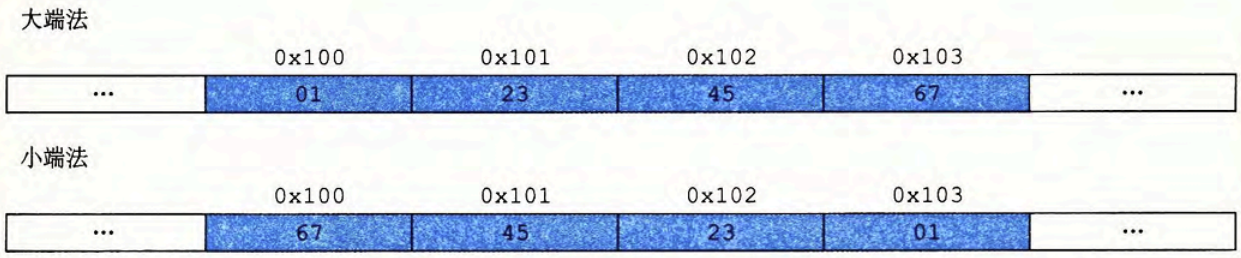
图 2-3 基本 C 数据类型的典型大小(以字节为单位)。分配的字节数受程序是如何编译的影响而变化。本图给出的是 32 位和 64 位程序的典型值

- 11.
  - 有符号：可以表示为 R；
  - 无符号：只能为非负数
  - 大部分数据类型都编码为有符号数值，除非有 `unsigned` 或 `u-int32_t`；`char` 是一个例外，尽管大多数编译器和机器将它们视为有符号数，但 C 标准不保证这一点
- 12. `unsigned long/unsigened long int/long unsigned/long unsigned int` 都是一个意思
- 13. 可移植性的一个方面就是使程序对不同数据类型的确切大小不敏感。假设一个声明为 int 类型的程序对象能被用来存储一个指针。这在大多数 32 位的机器上能正常工作，但是在一台 64 位的机器上却会导致问题。

2.1.3 寻址和字节顺序

- 14. 多字节对象被存储为连续的字节序列，对象的地址为所使用字节中最小的地址
- 15.
  - 一个w位整数，位表示为$[x_{w-1}, x_{w-2}, ..., x_1, x_0]$，其中$x_{w-1}, x_0$分别为最高/低有效位。假设$w$为 8 的倍数，那这些位就能被分组为字节，其中最高有效字节位$[x_{w-1}, x_{w-2}, ..., x_{w-8}]$，最低有效字节位$[x_7, x_6, ..., x_0]$
  - 小端法：内存中按照最低有效字节到最高有效字节的顺序存储对象
  - 大端法：从最高往最低
  - 双端法：可以配置成作为大端或小端的机器运行

假设变量 x 的类型为 int，位于地址 0x100 处，它的十六进制值为 0x01234567。地址范围 0x100～0x103 的字节顺序依赖于机器的类型：



注意，在字 0x01234567 中，高位字节的十六进制值为 0x01，而低位字节值为 0x67。

- 16. `4004d3: 01 05 43 0b 20 00  add  %eax,0x200b43(%rip)`意为：十六进制字节串 `01 05 43 0b 20 00` 是一条指令的字节级表示，这条指令是把一个字长的数据加到一个值上，该值的存储地址由 `0x200b43` 加上当前程序计数器的值得到，当前程序计数器的值即为下一条要执行指令的地址。若取出该序列最后四个字节并按相反方向写出 `00 20 0b 43`，去掉开头的 0，得到 `0x200b43`，这就是右边的值。当阅读像此类小端法机器生成的机器级程序表示时，经常会将字节按照相反的顺序显示。书写字节序列的自然方式是最低位字节在左边，而最高位字节在右边，这正好和通常书写数字时最高有效位在左边，最低有效位在右边的方式相反。
- 17. C 中可通过强制类型转换或联合来允许以一种数据类型来引用一个对象，而这种数据类型与创建这个对象时定义的数据类型不同。
- 18. 使用强制类型转换来访问和打印不同程序对象的字节表示

```c
#include<stdio.h>

typedef unsigned char* byte_pointer;

void show_bytes(byte_pointer start,size_t len){
    size_t i;
    for(int i=0;i<len;i++){
        printf("%.2x",start[i]);
        printf("\n");
    }
}

void show_int(int x){
    show_bytes((byte_pointer)& x,sizeof(int));
}

void show_float(float x){
    show_bytes((byte_pointer)& x,sizeof(float));
}

void show_pointer(void* x){
    show_bytes((byte_pointer)& x,sizeof(void*));
}
```

//传递给show bytes一个指向它们参数的x的指针&x，且这个指针被强制类型转换为unsigned char*。。这种强制类型转换告诉编译器，程序应该把这个指针看成指向一个字节序列，而不是指向一个原始数据类型的对象。然后，这个指针会被看成是对象使用的最低字节地址。

```
//sizeof(T)返回存储一个类型为T的对象所需字节数

void test_show_bytes(int val){
    int ival=val;
    float fval=(float) ival;
    int* pval=&ival;
    show_int(ival);
    show_float(fval);
    show_pointer(pval);
}
```

机器	值	类型	字节（十六进制）
Linux 32	12 345	int	39 30 00 00
Windows	12 345	int	39 30 00 00
Sun	12 345	int	00 00 30 39
Linux 64	12 345	int	39 30 00 00
Linux 32	12 345.0	float	00 e4 40 46
Windows	12 345.0	float	00 e4 40 46
Sun	12 345.0	float	46 40 e4 00
Linux 64	12 345.0	float	00 e4 40 46
Linux 32	&ival	int *	e4 f9 ff bf
Windows	&ival	int *	b4 cc 22 00
Sun	&ival	int *	ef ff fa 0c
Linux 64	&ival	int *	b8 11 e5 ff ff 7f 00 00

不同数据值的字节表示。除了字节顺序以外，int 和 float 的结果是一样的。指针值与机器相关

2.1.4 表示字符串

- 1.C 语言中字符串被编码为一个以 null(其值为 0)字符结尾的字符数组
- 2.$t_{(10)}$的 ASC 码是 0x3t，终止字节是 0x00，例如 12345~0x31 0x32 0x33 0x34 0x35 0x00 在使用 ASCII 码作为字符码的任何系统上都将得到相同的结果，与字节顺序和字大小规则无关

2.1.5 表示代码

- 1.我们发现指令编码是不同的。不同的机器类型使用不同的且不兼容的指令和编码方式。即使是完全一样的进程，运行在不同的操作系统上也会有不同的编码规则，因此二进制代码是不兼容的。二进制代码很少能在不同机器和操作系统组合之间移植。

2.1.6 布尔代数简介

- 1.运算符：
  - ~：非
  - &：与
  - |：或
  - ^：异或$(A+B=1)$
- 2.位向量（固定长度 w，由 0，1 构成）
  - 运算：若定义$a=[a_{w-1},a_{w-2},...,a_0],b=[b_{w-1},b_{w-2},...,b_0]$，则定义运算$a\&b=[a_{w-1}\&b_{w-1},...,a_1\&b_1]$，其余运算类似

- 应用：编码有限集合，如$a=[01101001]$表示集合$A=\{0,3,5,6\}$

2.1.7 C 语言中的位级运算

- 1. 掩码：0xFF；~ 0等等

2.1.8 C 语言中的逻辑运算

- 1. 如果对第一个参数求值就能确定表达式的结果，那么逻辑运算符就不会对第二个参数求值。因此，例如，表达式 `a&&5/a` 将不会造成被零除，而表达式 `p&&*p++`也不会导致间接引用空指针。

2.1.9 C 语言中的移位运算

- 1.

2.4.1 二进制小数

- 1. 形如$b_m b_{m-1}...b_1 b_0.b_{-1}b_{-2}...b_{-n}$的数，$b=\sum_{i=-n}^{m} 2^i \times b_i$

2.4.2 IEEE 浮点表示

- 1. $V=(-1)^s \times M \times 2^E$
  - 符号：s，由一个单独的符号位s直接编码
  - 尾数：M是一个二进制小数，范围是$1 \sim 2-\varepsilon \vee 0 \sim 1-\varepsilon$，由$n$位的小数字段$frac=f_{n-1}...f_1 f_0$编码
  - 阶码：E的作用是对浮点数加权，由k位的阶码字段$exp=e_{k-1}...e_1 e_0$

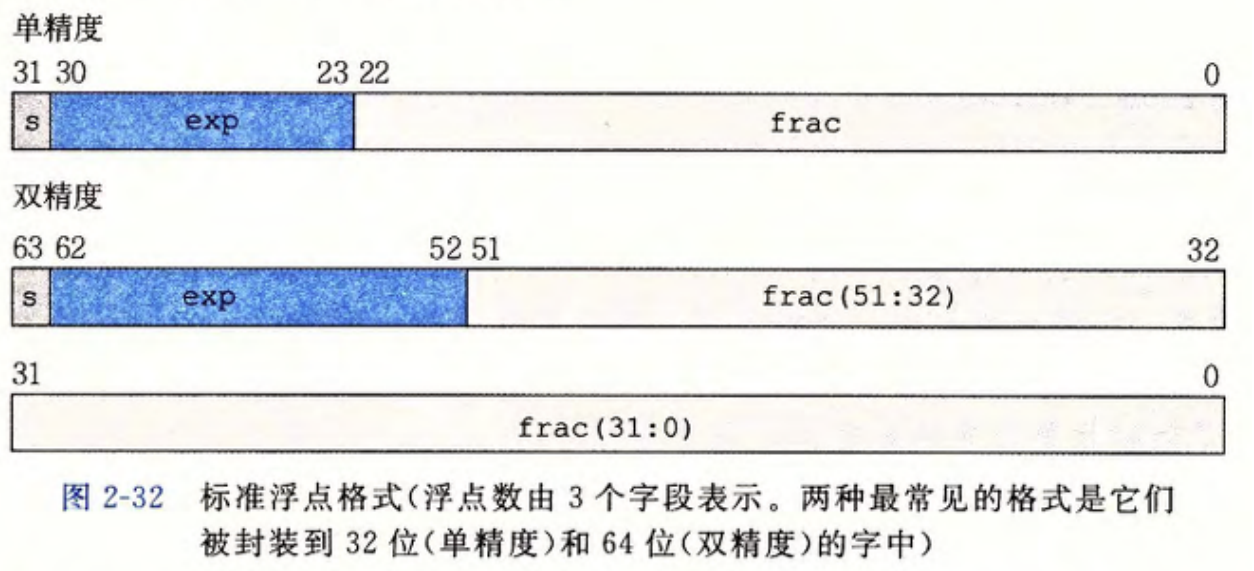


图 2-32 标准浮点格式(浮点数由 3 个字段表示。两种最常见的格式是它们被封装到 32 位(单精度)和 64 位(双精度)的字中)

- 2. float 值的分类
  - 规格化的值
    - 阶码的值是$E=e-Bias, e$是无符号数，位表示为$e_{k-1}...e_1 e_0$；而$Bias=2^{k-1}-1$
    - $frac$被解释为描述小数值$f(0 \le f < 1), M=1+f$

- 非规格化的值
  - $E=1-Bias, M=f$
  - 用途：表示 0；表示非常接近 0 的数
- 特殊值
  - 小数域全 0，表示∞
  - 小数域非 0，表示$\sqrt{-1}, \infty-\infty$等数

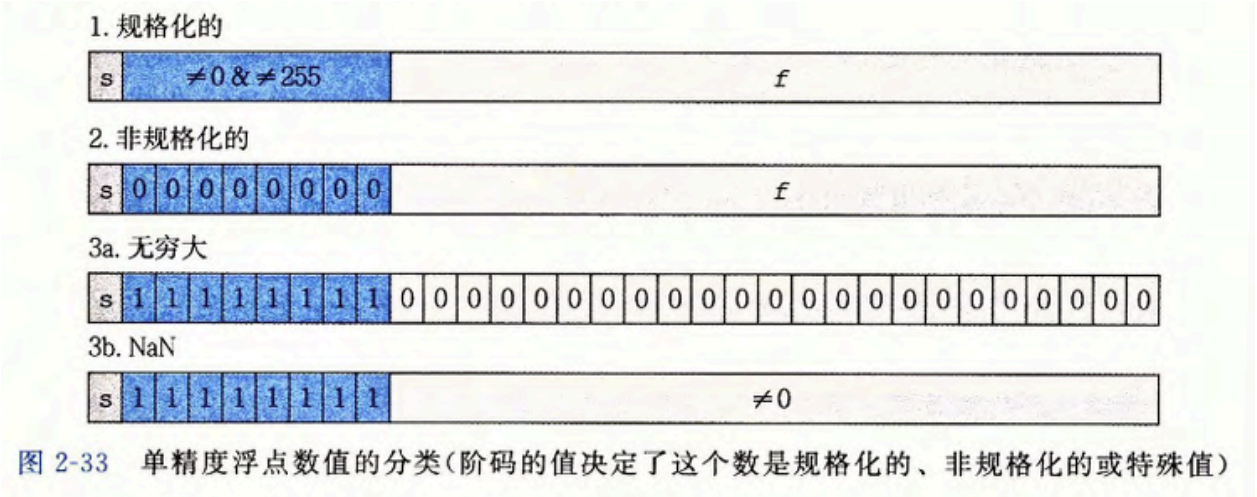


图 2-33　单精度浮点数值的分类(阶码的值决定了这个数是规格化的、非规格化的或特殊值)

2.4.3 数字示例

1. 一般属性

- +0.0 总有一个全为 0 的位表示
- 最小的正非规格化值：$M=f=2^{-n}, E=2-2^{k-1}, V=2^{-n-2^{k-1}+2}$
- 最大的非规格化值：$M=f=1-2^{-n}, E=2-2^{k-1}, V=(1-2^{-n})\times\left(2^{2-2^{k-1}}\right)$
- 最小的正规格化值：$M=1, E=2-2^{k-1}, V=2^{2-2^{k-1}}$
- 值 1.0：$E=0, M=1$
- 最大的规格化值：$f=1-2^{-n}, M=1+f=2-2^{-n}, E=e-Bias=2^k-2-\left(2^{k-1}-1\right)=2^{k-1}-1$

描　述	exp	frac	单精度		双精度	
			值	十进制	值	十进制
0	00 … 00	0 … 00	0	0.0	0	0.0
最小非规格化数	00 … 00	0 … 01	$2^{-23}\times2^{-126}$	1.4×10^{-45}	$2^{-52}\times2^{-1022}$	4.9×10^{-324}
最大非规格化数	00 … 00	1 … 11	$(1-\varepsilon)\times2^{-126}$	1.2×10^{-38}	$(1-\varepsilon)\times2^{-1022}$	2.2×10^{-308}
最小规格化数	00 … 01	0 … 00	1×2^{-126}	1.2×10^{-38}	1×2^{-1022}	2.2×10^{-308}
1	01 … 11	0 … 00	1×2^{0}	1.0	1×2^{0}	1.0
最大规格化数	11 … 10	1 … 11	$(2-\varepsilon)\times2^{127}$	3.4×10^{38}	$(2-\varepsilon)\times2^{1023}$	1.8×10^{308}

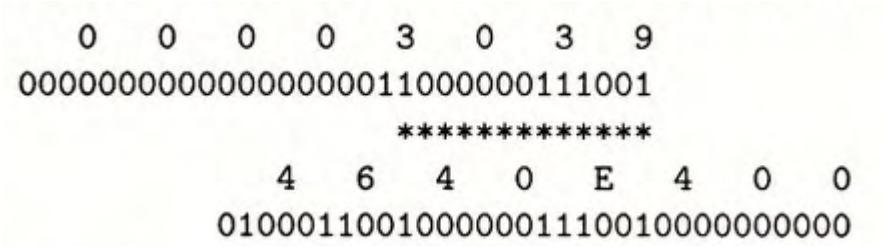
2. 整型转浮点型

【例】$12345\rightarrow12345.0(float)$

解：

- $12345\sim[11000000111001]=1.1000000111001\times2^{13}$
- 为构造小数字段，丢弃开头的 1，末尾添加 10 个 0
- 为构造阶码字段，用 13 加上偏置量 127，得 140

- 加上符号位 0
- 即可得$[0-10001100-10000001110010000000000]$



2.4.4 舍入

1. 四种舍入方式（仅针对两个最紧有效值的中间值）
- 向偶数舍入（默认）
  - 它将数字向上或者向下舍入，使得结果的最低有效数字是偶数
  - $1.5, 2.5 \to 2$
  - $1.235, 1.245 \to 1.24$
  - $10.11100_2\left(2\frac{7}{8}\right) \to 11.00, 10.10100_2\left(2\frac{5}{8}\right) \to 10.10_2\left(2\frac{1}{2}\right)$
- 向 0 舍入
- 向上舍入
- 向下舍入

2.4.5 浮点运算

1. 溢出

2. 加法和乘法都具有单调性

2.4.6 C 语言中的浮点数

1. 生成特殊值

2. 强制类型转换
- `int -> float`:不会溢出，可能舍入
- `int/float -> double`:精准
- `double -> float`:可能溢出，可能舍入
- `float/double -> int`:向 0 舍入，可能溢出

二·公式定理

1.

三·精选例题

1. 进制转换

(1)将 0xD5E4C 转换为二进制

(2)将二进制 10011011100111110110101 转换为十六进制

(3)将十进制 2048 转换为十六进制

解：

(1) 查表法，11010101111001001100

(2) 从右至左，每四位一组，不足四位的高位补 0，得到 0010 0110 1110 0111 1011 0101，对应的十六进制数为 0X26E7B

(3) $2048 = 2^{11}, 11 = 3 + 4 * 2$，故十六进制为 0x800

2. 思考下面对 show_bytes 的三次调用：

```
int val=0x87654321;
byte_pointer valp=(byte_pointer) &val;
show_bytes(valp,1);/* A */
show_bytes(valp,2);/* B */
show_bytes(valp,3);/* C */
```

指出在小端法和大端法机器上，每次调用的输出值

解：

- A. 小端法：21；大端法：87
- B. 小端法：21 43；大端法：87 65
- C. 小端法：21 43 65；大端法：87 65 43

3. 使用 show_int,show_float，我们确定整数 3510593 的十六进制表示为 0x00359141，而浮点数 3510593.0 的十六进制 0x4A564504

- A. 写出这两个二进制表示
- B. 移动这两个二进制串相对位置，使得他们相匹配位数最多
- C. 串中什么部分不匹配？

解：

- A. 0000 0000 0011 0101 1001 0001 0100 0001
  - 0100 1010 0101 0110 0100 0101 0000 0100
- B. 对齐后发现了 21 位 是相匹配的
- C. 头部

4. 下面对 show_bytes 的调用将输出什么结果？

```
const char* s="abcdef";
show_bytes((byte_pointer) s,strlen(s));
```

字母 a-z 的 ASCII 码为 0x61-0x7A。

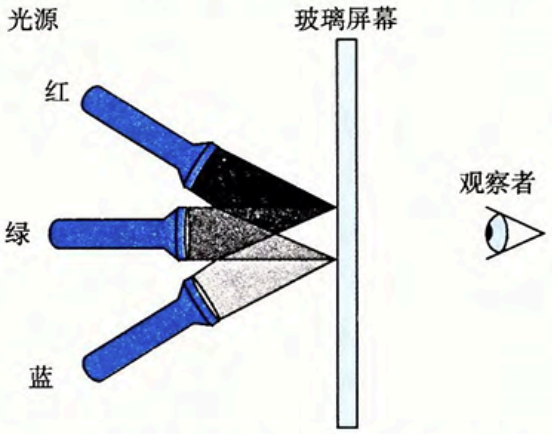
解：61 62 63 64 65 66（因为打印长度为 6，不包括结尾的 0）

5. 运算位向量

运算	结果
a	[01101001]
b	[01010101]
$\sim a$	[10010110]
$\sim b$	[10101010]
$a\&b$	[01000000]
$a\|b$	[01111101]
$a\hat{}b$	[00111100]

6.

练习题 2.9 通过混合三种不同颜色的光(红色、绿色和蓝色)，计算机可以在视频屏幕或者液晶显示器上产生彩色的画面。设想一种简单的方法，使用三种不同颜色的光，每种光都能打开或关闭，投射到玻璃屏幕上，如图所示：



那么基于光源 R(红)、G(绿)、B(蓝)的关闭(0)或打开(1)，我们就能够创建 8 种不同的颜色：

R	G	B	颜色	R	G	B	颜色
0	0	0	黑色	1	0	0	红色
0	0	1	蓝色	1	0	1	红紫色
0	1	0	绿色	1	1	0	黄色
0	1	1	蓝绿色	1	1	1	白色

这些颜色中的每一种都能用一个长度为 3 的位向量来表示，我们可以对它们进行布尔运算。

A. 一种颜色的补是通过关掉打开的光源，且打开关闭的光源而形成的。那么上面列出的 8 种颜色每一种的补是什么？

B. 描述下列颜色应用布尔运算的结果：

蓝色 | 绿色 =

黄色 & 蓝绿色 =

红色 ^ 红紫色 =

解：

A. 黑-白；蓝-黄；绿-红紫；蓝绿-红

B. 蓝绿；绿；蓝

7.

练习题 2.10 对于任一位向量 a，有 $a \hat{} a = 0$。应用这一属性，考虑下面的程序：

```
1   void inplace_swap(int *x, int *y) {
2       *y = *x ^ *y;  /* Step 1 */
3       *x = *x ^ *y;  /* Step 2 */
4       *y = *x ^ *y;  /* Step 3 */
5   }
```

正如程序名字所暗示的那样，我们认为这个过程的效果是交换指针变量 x 和 y 所指向的存储位置处存放的值。注意，与通常的交换两个数值的技术不一样，当移动一个值时，我们不需要第三个位置来临时存储另一个值。这种交换方式并没有性能上的优势，它仅仅是一个智力游戏。

以指针 x 和 y 指向的位置存储的值分别是 a 和 b 作为开始，填写下表，给出在程序的每一步之后，存储在这两个位置中的值。利用 $\hat{}$ 的属性证明达到了所希望的效果。回想一下，每个元素就是它自身的加法逆元($a \hat{} a = 0$)。

步骤	*x	*y
初始	a	b
第1步		
第2步		
第3步		

解：

步骤	*x	*y
初始	a	b
第一步	a	a^b
第二步	a^(a^b)=b	a^b
第三步	b	a

8.

练习题 2.11 在练习题 2.10 中的 inplace_swap 函数的基础上，你决定写一段代码，实现将一个数组中的元素头尾两端依次对调。你写出下面这个函数：

```
1   void reverse_array(int a[], int cnt) {
2       int first, last;
3       for (first = 0, last = cnt-1;
4            first <= last;
5            first++,last--)
6            inplace_swap(&a[first], &a[last]);
7   }
```

当你对一个包含元素 1、2、3 和 4 的数组使用这个函数时，正如预期的那样，现在数组的元素变成了 4、3、2 和 1。不过，当你对一个包含元素 1、2、3、4 和 5 的数组使用这个函数时，你会很惊奇地看到得到数字的元素为 5、4、0、2 和 1。实际上，你会发现这段代码对所有偶数长度的数组都能正确地工作，但是当数组的长度为奇数时，它就会把中间的元素设置成 0。

A. 对于一个长度为奇数的数组，长度 cnt＝$2k+1$，函数 reverse_array 最后一次循环中，变量 first 和 last 的值分别是什么？

B. 为什么这时调用函数 inplace_swap 会将数组元素设置为 0？

C. 对 reverse_array 的代码做哪些简单改动就能消除这个问题？

解：

A.

B.

C.

第三章

一 • 基本概念

3.1 历史观点

- 1.
- 2.

3.2.1 机器级代码

- 1.

3.2.2 代码示例

- 1.

```
//mstore.c
long mult2(long,long);

void multstore(long x,long y,long *dest){
    long t=mult2(x,y);
    *dest=t;
}
```

在命令行上使用-s 选项，即可看到 C 语言编译器产生的汇编代码：

```
linux> gcc -Og -S mstore.c
```

这会使 GCC 运行编译器，产生一个汇编文件 mstore.s

```
multstore:
  pushq   %rbx  //每行都对应一条机器指令，如本行表示将rbx内容压入栈中
  movq    %rdx,%rbx
  call    mult2
  movq    %rax,(%rbx)
  popq    %rbx
  ret
```

若

```
linux> gcc -Og -c mstore.c
```

这就会产生目标代码文件(二进制)mstore.o，其中有一段 14 字节的序列： 53 48 89 d3 e8 00 00 00 00 48 89 03 5b c3 机器执行的程序只是一个字节序列，它是对一系列指令的编码。机器对产生这些指令的源代码几乎一无所知。

- 2.反汇编器

```
linux> objdump -d mstore.o
```

结果如下

```
0000000000000000 <multstore>:
  0:  53                    push  %rbx
  1:  48 89 d3              mov   %rdx,%rbx
  4:  e8 00 00 00 00        callq 9 <mulstore+0x9>
  9:  48 89 03              mov   %rax,(%rbx)
  c:  5b                    pop   %rbx
  d:  c3
```

我们看到按照前面给出的字节顺序排列的 14 个十六进制字节值，它们分成了若干组，每组有1 ~ 5个字节。每组都是一条指令，右边是等价的汇编语言。

- 3.机器代码和反汇编表示的特性：
  - x86-64 指令长度 1-15 字节，常用的指令以及操作数较少的指令所需的字节数少
  - 设计指令格式的方式是，从某个给定位置开始，可以将字节唯一地解码成机器指令。例如，只有指令 pushq %rbx 是以字节值 53 开始的
  - 反汇编器只是基于机器代码文件中的字节序列来确定汇编代码。它不需要访问该程序的源代码或汇编代码

- 反汇编器使用的指令命名规则与 GCC 生成的汇编代码使用的有些细微的差别。 在我们的示例中，它省略了很多指令结尾的 q。这些后缀是大小指示符，在大多数情况中可以省略。相反，反汇编器给 call 和 ret 指令添加了 q 后缀，同样，省略这些后缀也没有问题

- 4. 假设在文件 main.c 中有下面的函数：

```
#include<stdio.h>

void multstore(long,long,long*);

int main(){
    long d;
    multstore(2,3,&d);
    printf("2*3-->%ld\n",d);
    return 0;
}
long mult2(long a,long b){
    long s=a*b;
    return s;
}
```

然后生成可执行文件 prog：

```
linux> gcc -Og -o prog main.c mstore.c
```

文件 prog 变成了 8655 个字节，因为它不仅包含了两个过程的代码，还包含了用来启动和终止程序的代码，以及用来与操作系统交互的代码。我们也可以反汇编 prog 文件

```
linux> objdump -d prog
```

反汇编器会抽出各种代码序列，包括：

```
0000000000000000 <multstore>:
  400540:  53                   push  %rbx
  400541:  48 89 d3             mov   %rdx,%rbx
  400544:  e8 00 00 00 00       callq 40058b <mult2>
  400549:  48 89 03             mov   %rax,(%rbx)
  40054c:  5b                   pop   %rbx
  40054d:  c3                   retq
  40054e:  90                   nop
  40054f:  90                   nop
```

与 mstore.c 反汇编主要区别是

- 左边列出地址不同;
- 链接器填上了 callq 指令调用函数 mult2 需要使用的地址（反汇编代码第 4 行）链接器的任务之一就是为函数调用找到匹配的函数的可执行代码的位置。
- 多了两行代码（第 8 和 9 行）。这两条指令对程序没有影响，因为它们出现在返回指令后面（第 7 行）。插入这些指令是为了使函数代码变为 16 字节，使得就存储器系统性能言，能更好地放置下一个代码块

3.2.3 格式注解

- 1.

```
void multstore(long x,long y,long *dest)
x in %rdi,y in %rsi,dest in %rdx
1   multstore:
2     pushq   %rbx          Save %rdx
3     movq    %rdx,%rbx     Copy dest to %rdx
4     call    mult2         Call mult2(x,y)
5     movq    %rax,(%rbx)   Store result at *dest
6     popq    %rbx          Restore %rbx
7     ret                   Return
```

3.3 数据格式

- 1.

C声明	Intel数据类型	汇编代码后缀	大小(字节)
char	字节	b	1
short	字	w	2
int	双字	l	4
long	四字	q	8
char*	四字	q	8
float	单精度	s	4
double	双精度	l	8

图 3-1 C语言数据类型在 x86-64 中的大小。在 64 位机器中，指针长 8 字节

3.4.1 访问信息

- 1.通用目的寄存器

图 3-2 整数寄存器。所有 16 个寄存器的低位部分都可以作为字节、字(16 位)、双字(32 位)和四字(64 位)数字来访问

- 字节级操作可以访问最低的字节， 16 位操作可以访问最低的 2 个字节， 32 位操作可以访问最低的 4 个字节，而 64 位操作可以访问整个寄存器。
- 对于生成小于 8 字节结果的指令，寄存器中剩下的字节会怎么样，对此有两条规则：生成 1 字节和 2 字节数字的指令会保持剩下的字节不变；生成 4 字节数字的指令会把高位 4 个字节置为 0
- 2. 操作数
  - 立即数：表示常数值，如 `$-577/$0x1F`

- 寄存器：表示某个寄存器的内容
- 内存引用：根据计算出来的有效地址访问内存位置

类型	格式	操作数值	名称
立即数	$*Imm*	*Imm*	立即数寻址
寄存器	r_a	$R[r_a]$	寄存器寻址
存储器	*Imm*	$M[Imm]$	绝对寻址
存储器	(r_a)	$M[R[r_a]]$	间接寻址
存储器	$Imm(r_b)$	$M[Imm+R[r_b]]$	（基址＋偏移量）寻址
存储器	(r_b, r_i)	$M[R[r_b]+R[r_i]]$	变址寻址
存储器	$Imm(r_b, r_i)$	$M[Imm+R[r_b]+R[r_i]]$	变址寻址
存储器	$(,r_i, s)$	$M[R[r_i] \cdot s]$	比例变址寻址
存储器	$Imm(,r_i,s)$	$M[Imm+R[r_i] \cdot s]$	比例变址寻址
存储器	(r_b, r_i, s)	$M[R[r_b]+R[r_i] \cdot s]$	比例变址寻址
存储器	$Imm(r_b, r_i, s)$	$M[Imm+R[r_b]+R[r_i] \cdot s]$	比例变址寻址

图 3-3　操作数格式。操作数可以表示立即数(常数)值、寄存器值或是来自内存的值。比例因子 s 必须是 1、2、4 或者 8

3.4.2 数据传送指令

- 1.MOV 指令类

指令	效果	描述
MOV　　S，D	D←S	传送
movb		传送字节
movw		传送字
movl		传送双字
movq		传送四字
movabsq　　I，R	R←I	传送绝对的四字

图 3-4　简单的数据传送指令

- 常规的 movq 指令只能以表示为 32 位补码数字的立即数作为源操作数，然后把这个值符号扩展得到 64 位的值，放到目的位置。 movabsq 指令能够以任意 64 位立即数值作为源操作数，并且只能以寄存器作为目的
- 2.下面的指令给出了五种可能的组合，第一个是源操作数，第二个是目的操作数

```
1    movl $0x4050,%eax          Immediate--Register, 4 bytes
2    movw %bp,%sp               Register--Register,  2 bytes
3    movb (%rdi,%rcx),%al       Memory--Register,    1 byte
4    movb $-17,(%rsp)           Immediate--Memory,   1 byte
5    movq %rax,-12(%rbp)        Register--Memory,    8 bytes
```

- 3.MOVZ 指令类：将较小的源值复制到较大的目的时使用。所有这些指令都把数据从源（在寄存器或内存中）复制到目的寄存器。 MOVZ 类中的指令把目的中剩余的字节填充为 o，而 MOVS 类中的指令通过符号扩展来填充，把源操作的最高位进行复制。可以观察到，每条指令名字的最后两个字符都是大小指示符：第一个字符指定源的大小，而第二个指明目的的大小。正如看到的那样，这两个类中每个都有三条指令，包括了所有的源大小为 1 个和 2 个字节、目的大小为 2 个和 4 个的情况，当然只考虑目的大于源的情况

指令	效果	描述
MOVZ S, R	R←零扩展(S)	以零扩展进行传送
movzbw		将做了零扩展的字节传送到字
movzbl		将做了零扩展的字节传送到双字
movzwl		将做了零扩展的字传送到双字
movzbq		将做了零扩展的字节传送到四字
movzwq		将做了零扩展的字传送到四字

图 3-5 零扩展数据传送指令。这些指令以寄存器或内存地址作为源，以寄存器作为目的

指令	效果	描述
MOVS S, R	R←符号扩展(S)	传送符号扩展的字节
movsbw		将做了符号扩展的字节传送到字
movsbl		将做了符号扩展的字节传送到双字
movswl		将做了符号扩展的字传送到双字
movsbq		将做了符号扩展的字节传送到四字
movswq		将做了符号扩展的字传送到四字
movslq		将做了符号扩展的双字传送到四字
cltq	%rax ←符号扩展(%eax)	把%eax 符号扩展到%rax

图 3-6 符号扩展数据传送指令。MOVS 指令以寄存器或内存地址作为源，以寄存器作为目的。cltq 指令只作用于寄存器%eax 和%rax

- 4.

旁注 理解数据传送如何改变目的寄存器

正如我们描述的那样，关于数据传送指令是否以及如何修改目的寄存器的高位字节有两种不同的方法。下面这段代码序列会说明其差别：

```
1    movabsq $0x0011223344556677, %rax    %rax = 0011223344556677
2    movb    $-1, %al                     %rax = 00112233445566FF
3    movw    $-1, %ax                     %rax = 001122334455FFFF
4    movl    $-1, %eax                    %rax = 00000000FFFFFFFF
5    movq    $-1, %rax                    %rax = FFFFFFFFFFFFFFFF
```

在接下来的讨论中，我们使用十六进制表示。在这个例子中，第 1 行的指令把寄存器%rax 初始化为位模式 0011223344556677。剩下的指令的源操作数值是立即数值－1。回想－1 的十六进制表示形如 FF…F，这里 F 的数量是表述中字节数量的两倍。因此 movb 指令(第 2 行)把%rax 的低位字节设置为 FF，而 movw 指令(第 3 行)把低 2 位字节设置为 FFFF，剩下的字节保持不变。movl 指令(第 4 行)将低 4 个字节设置为 FFFFFFFF，同时把高位 4 字节设置为 00000000。最后 movq 指令(第 5 行)把整个寄存器设置为 FFFFFFFFFFFFFFFF。

- 5.注意图 3-5 中并没有一条明确的指令把 4 字节源值零扩展到 8 字节目的。这样的指令逻辑上应该被命名为 movzlq，但是并没有这样的指令。不过，这样的数据传送可以用以寄存器为目的的 movl 指令来实现。这一技术利用的属性是，生成 4 字节值并以寄存器作为目的的指令会把高 4 字节置为 0。对于 64 位的目标，所有三种源类型都有对应的符号扩展传送，而只有两种较小的源类型有零扩展传送。
- 6.

旁注 字节传送指令比较

下面这个示例说明了不同的数据传送指令如何改变或者不改变目的的高位字节。仔细观察可以发现，三个字节传送指令 movb、movsbq 和 movzbq 之间有细微的差别。示例如下：

```
1    movabsq $0x0011223344556677, %rax    %rax = 0011223344556677
2    movb    $0xAA, %dl                   %dl  = AA
3    movb %dl,%al                         %rax = 00112233445566AA
4    movsbq %dl,%rax                      %rax = FFFFFFFFFFFFFFAA
5    movzbq %dl,%rax                      %rax = 00000000000000AA
```

在下面的讨论中，所有的值都使用十六进制表示。代码的头 2 行将寄存器%rax 和%dl 分别初始化为 0011223344556677 和 AA。剩下的指令都是将%rdx 的低位字节复制到%rax 的低位字节。movb 指令(第 3 行)不改变其他字节。根据源字节的最高位，movsbq 指令(第 4 行)将其他 7 个字节设为全 1 或全 0。由于十六进制 A 表示二进制值 1010，符号扩展会把高位字节都设置为 FF。movzbq 指令(第 5 行)总是将其他 7 个字节全都设置为 0。

3.4.3 数据传送示例

- 1.

```
long exchange(long *xp, long y)
{
    long x = *xp;
    *xp = y;
    return x;
}
```

a）C语言代码

```
    long exchange(long *xp, long y)
    xp in %rdi, y in %rsi
1   exchange:
2     movq    (%rdi), %rax    Get x at xp. Set as return value.
3     movq    %rsi, (%rdi)    Store y at xp.
4     ret                     Return.
```

b）汇编代码

- 2.两个值得注意的点

- C 语言中所谓的”指针”其实就是地址。间接引用指针就是将该指针放在一个寄存器中，然后在内存引用中使用这个寄存器。
- 像 x 这样的局部变量通常是保存在寄存器中，而不是内存中。访问寄存器比访问内存要快得多

• 3.

给 C 语言初学者　指针的一些示例

函数 exchange(图 3-7a)提供了一个关于 C 语言中指针使用的很好说明。参数 xp 是一个指向 long 类型的整数的指针，而 y 是一个 long 类型的整数。语句

```
long x = *xp;
```

表示我们将读存储在 xp 所指位置中的值，并将它存放到名字为 x 的局部变量中。这个读操作称为指针的间接引用(pointer dereferencing)，C 操作符* 执行指针的间接引用。

语句

```
*xp = y;
```

正好相反——它将参数 y 的值写到 xp 所指的位置。这也是指针间接引用的一种形式(所以有操作符*)，但是它表明的是一个写操作，因为它在赋值语句的左边。

下面是调用 exchange 的一个实际例子：

```
long a = 4;
long b = exchange(&a, 3);
printf("a = %ld, b = %ld\n", a, b);
```

这段代码会打印出：

```
a = 3, b = 4
```

C 操作符 &(称为“取址”操作符)创建一个指针，在本例中，该指针指向保存局部变量 a 的位置。然后，函数 exchange 将用 3 覆盖存储在 a 中的值，但是返回原来的值 4 作为函数的值。注意如何将指针传递给 exchange，它能修改存在某个远处位置的数据。

3.4.4 压入和弹出栈数据

• 1.

指令	效果	描述
pushq S	R[%rsp]←R[%rsp]−8; M[R[%rsp]]←S	将四字压入栈
popq D	D←M[R[%rsp]]; R[%rsp]←R[%rsp]+8	将四字弹出栈

图 3-8　入栈和出栈指令

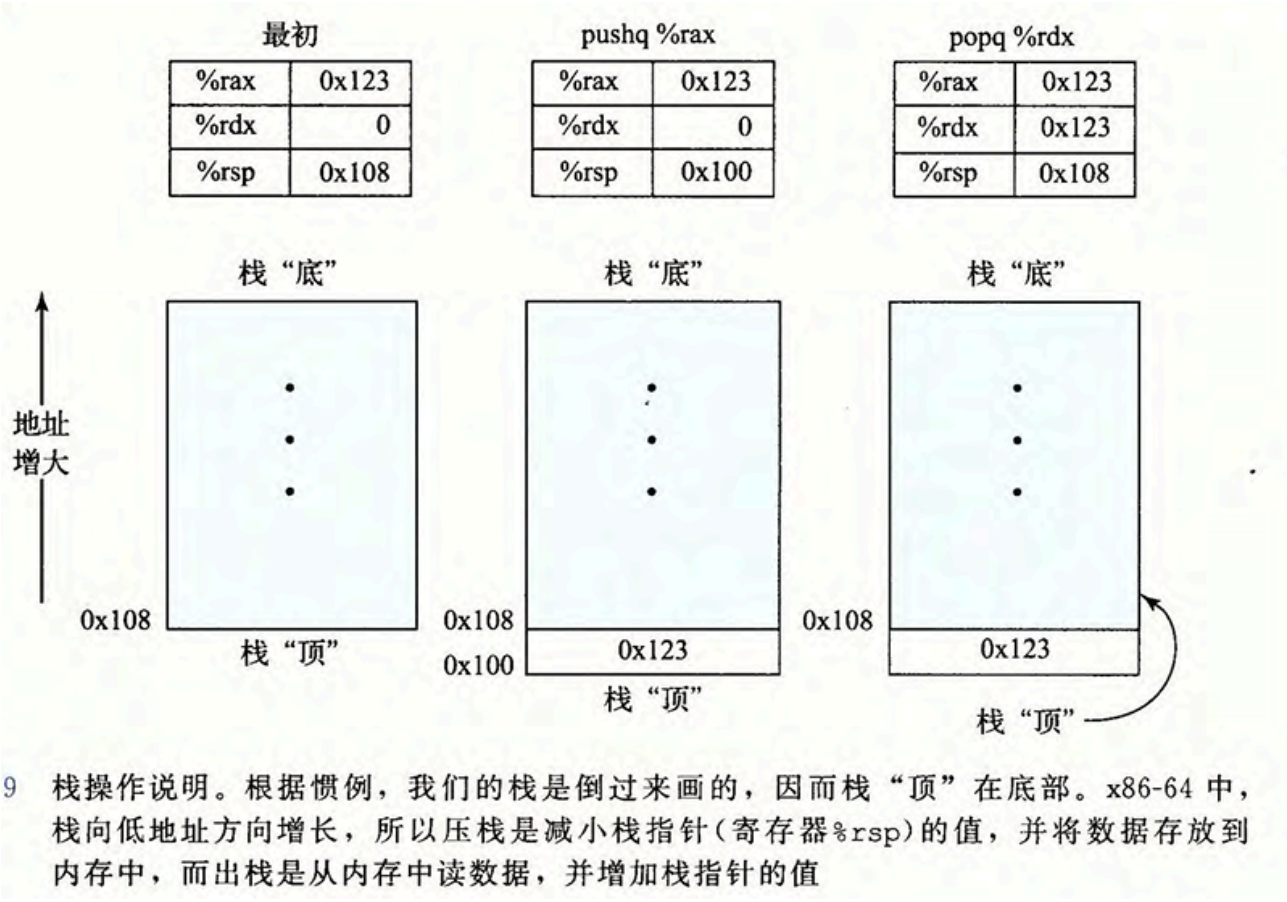
栈指针 %rsp 保存着栈顶元素的地址

```
pushq %rbp
```

等价于

```
subq $8,%rsp
movq %rbq,(%rsp)
```

它们之间的区别是在机器代码中 pushq 指令编码为 1 个字节，而上面那两条指令一共需要 8 个字节



9 栈操作说明。根据惯例，我们的栈是倒过来画的，因而栈“顶”在底部。x86-64 中，栈向低地址方向增长，所以压栈是减小栈指针（寄存器%rsp）的值，并将数据存放到内存中，而出栈是从内存中读数据，并增加栈指针的值

弹出一个四字的操作包括从栈顶位置读出数据，然后将栈指针加 8。因此，指令 popq %rax 等价千下面两条指令：

```
movq (%rsp),%rax
addq $8,%rsp
```

值 0x123 仍然会保持在内存位置 0x100 中，直到被覆盖（例如被另一条入栈操作覆盖）。

- 2.因为栈和程序代码以及其他形式的程序数据都是放在同一内存中，所以程序可以用标准的内存寻址方法访问栈内的任意位置。例如，假设栈顶元素是四字，指令 movq 8 (%rsp), %rdx 会将第二个四字从栈中复制到寄存器 %rdx

3.5.1 加载有效地址

- 1.每个指令类都有对这四种不同大小数据的指令。这些操作被分为四组：加载有效地址、一元操作、二元操作和移位

指令		效果	描述
leaq	S, D	$D \leftarrow \&S$	加载有效地址
INC	D	$D \leftarrow D+1$	加1
DEC	D	$D \leftarrow D-1$	减1
NEG	D	$D \leftarrow -D$	取负
NOT	D	$D \leftarrow \sim D$	取补
ADD	S, D	$D \leftarrow D+S$	加
SUB	S, D	$D \leftarrow D-S$	减
IMUL	S, D	$D \leftarrow D*S$	乘
XOR	S, D	$D \leftarrow D \hat{} S$	异或
OR	S, D	$D \leftarrow D \mid S$	或
AND	S, D	$D \leftarrow D \& S$	与
SAL	k, D	$D \leftarrow D << k$	左移
SHL	k, D	$D \leftarrow D << k$	左移（等同于SAL）
SAR	k, D	$D \leftarrow D >>_A k$	算术右移
SHR	k, D	$D \leftarrow D >>_L k$	逻辑右移

整数算术操作。加载有效地址(leaq)指令通常用来执行简单的算术操作。其余的指令是更加标准的一元或二元操作。我们用$>>_A$和$>>_L$来分别表示算术右移和逻辑右移。注意，这里的操作顺序与 ATT 格式的汇编代码中的相反

- 2.

```
long scale(long x,long y,long z){
    long t=x+4*y+12*z;
    return t;
}
```

编译时，

```
scale:
  leaq   (%rdi,%rsi,4),%rax
  leaq   (%rdx,%rdx,2),%rdx
  leaq   (%rax,%rdx,4),%rax
  ret
```

3.5.2 一元和二元操作

- 1.第二组中的操作是一元操作，只有一个操作数，既是源又是目的。这个操作数可以是一个寄存器，也可以是一个内存位置。
- 2.第三组是二元操作，其中，第二个操作数既是源又是目的。这种语法让人想起 C 语言中的赋值运算符)。第一个操作数可以是立即数、寄存器或是内存位置。第二个操作数可以是寄存器或是内存位置。注意，当第二个操作数为内存地址时，处理器必须从内存读出值，执行操作，再把结果写回内存。

3.5.3 移位操作

- 1.移位量可以是一个立即数，或者放在单字节寄存器 %cl 中。（这些指令很特别，因为只允许以这个特定的寄存器作为操作数。）
- 2. x86-64 中，移位操作对 w 位长的数据值进行操作，移位最是由 %cl 寄存器的低 m 位决定的，这里$2^m = w$。高位会被忽略。所以，例如当寄存器 %cl 的十六进制值为 0xFF 时，指令 salb 会移 7 位， salw 会移 15 位， sall 会移 31 位，而 salq 会移 63 位。

3.5.4 讨论

- 1.

```
long arith(long x, long y, long z)
{
    long t1 = x ^ y;
    long t2 = z * 48;
    long t3 = t1 & 0x0F0F0F0F;
    long t4 = t2 - t3;
    return t4;
}
```

a）C语言代码

```
    long arith(long x, long y, long z)
    x in %rdi, y in %rsi, z in %rdx
1   arith:
2     xorq    %rsi, %rdi              t1 = x ^ y
3     leaq    (%rdx,%rdx,2), %rax     3*z
4     salq    $4, %rax                t2 = 16 * (3*z) = 48*z
5     andl    $252645135, %edi        t3 = t1 & 0x0F0F0F0F
6     subq    %rdi, %rax              Return t2 - t3
7     ret
```

b）汇编代码

图 3-11　算术运算函数的 C 语言和汇编代码

编译器产生的代码中，会用一个寄存器存放多个程序值，还会在寄存器之间传送程序值。

3.5.5 特殊的算术操作

- 1.两个 64 位有符号或无符号整数相乘得到的乘积需要 128 位来表示。 x86-64 指令集对 128 位(16 字节）数的操作提供有限的支持。延续字(2 字节)、双字(4 字节）和四字(8 字节）的命名惯例， Intel 把 16 字节的数称为八字

指令	效果	描述
imulq S mulq S	R[%rdx]：R[%rax]←S×R[%rax] R[%rdx]：R[%rax]←S×R[%rax]	有符号全乘法 无符号全乘法
clto	R[%rdx]：R[%rax]←符号扩展(R[%rax])	转换为八字
idivq S	R[%rdx]←R[%rdx]：R[%rax] mod S R[%rdx]←R[%rdx]：R[%rax]÷S	有符号除法
divq S	R[%rdx]←R[%rdx]：R[%rax] mod S R[%rdx]←R[%rdx]：R[%rax]÷S	无符号除法

图 3-12 特殊的算术操作。这些操作提供了有符号和无符号数的全 128 位乘法和除法。一对寄存器%rdx 和%rax 组成一个 128 位的八字

- 2.imulq 指令有两种不同的形式。
  - 其中一种，如图 3-10 所示，是 IMUL 指令类中的一种。 这种形式的 imulq 指令是一个“双操作数＂乘法指令。 它从两个 64 位操作数产生一个 64 位乘积
  - 此外， x86-64 指令集还提供了两条不同的“单操作数”乘法指令，以计算两个 64 位值的全 128 位乘积—— 一个是无符号数乘法(mulq)，而另一个是补码乘法(imulq)。这两条指令都要求一个参数必须在寄存器 %rax 中，而另一个作为指令的源操作数给出。然后乘积存放在寄存器 %rdx(高 64 位）和 %rax(低 64 位）中。虽然 imulq 这个名字可以用于两个不同的乘法操作，但是汇编器能够通过计算操作数的数目，分辨出想用哪条指令

```
   void store_uprod(uint128_t *dest, uint64_t x, uint64_t y)
   dest in %rdi, x in %rsi, y in %rdx
1  store_uprod:
2    movq    %rsi, %rax      Copy x to multiplicand
3    mulq    %rdx            Multiply by y
4    movq    %rax, (%rdi)    Store lower 8 bytes at dest
5    movq    %rdx, 8(%rdi)   Store upper 8 bytes at dest+8
6    ret
```

这段代码针对的是小端法机器，所以高位字节存储在大地址，正如地址 8(%rdi)表明的那样

- 3.。有符号除法指令过 ivl 将寄存器 %rdx(高 64 位）和 %rax(低 64 位） 中的 128 位数作为被除数，而除数作为指令的操作数给出。指令将商存储在寄存器 %rax 中，将余数存储在寄存器 %rdx 中。对千大多数 64 位除法应用来说，除数也常常是一个 64 位的值。这个值应该存放在 %rax 中， %rdx 的位应该设置为全 0(无符号运算）或者 %rax 的

符号位（有符号运算）。后面这个操作可以用指令 cqto 来完成。这条指令不需要操作数——它隐含读出 %rax 的符号位，并将它复制到 %rdx 的所有位。

- 4.

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

对应汇编代码

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

在上述代码中，必须首先把参数 qp 保存到另一个寄存器中（第 2 行），因为除法操作 要使用参数寄存器 %rdx。接下来，第 3 4 行准备被除数，复制并符号扩展 x。除法之后，寄存器 %rax 中的商被保存在 qp(第 6 行)，而寄存器 %rdx 中的余数被保存在 rp(第 7 行)。 无符号除法使用 divq 指令。通常，寄存器 %rdx 会事先设置为 0。

3.6.1 条件码

- 1.机器代码提供两种基本的低级机制来实现有条件的行为：测试数据值，然后根据测试的结果来改变控制流或者数据流。用 jump 指令可以改变一组机器代码指令的执行顺序
- 2.条件码寄存器
  - CF：进位标志。最近的操作使最高位产生了进位。可用来检查无符号操作的溢出
  - 2F：零标志。最近的操作得出的结果为 0
  - SF：符号标志。最近的操作得到的结果为负数
  - OF：溢出标志。最近的操作导致一个补码溢出——正溢出或负溢出
- 3.

比如说，假设我们用一条 ADD 指令完成等价于 C 表达式 t=a+ b 的功能，这 a、b 和 t 都是整型的。然后，根据下面的 C 表达式来设置条件码：

CF	(unsigned) t < (unsigned) a	无符号溢出
ZF	(t ==0)	零
SF	(t<0)	负数
OF	(a<0==b<0) && (t<0 !=a<0)	有符号溢出

- 4.

指令		基于	描述
CMP	S_1，S_2	S_2-S_1	比较
cmpb			比较字节
cmpw			比较字
cmpl			比较双字
cmpq			比较四字
TEST	S_1，S_2	S_1&S_2	测试
testb			测试字节
testw			测试字
testl			测试双字
testq			测试四字

图 3-13 比较和测试指令。这些指令不修改任何寄存器的值，只设置条件码

3.6.2 访问条件码

- 1.

三 • 精选例题

1. 已知

地址	值
0x100	0xFF
0x104	0xAB

寄存器	值
%rax	0x100
%rcx	0x1

0x108	0x13
0x10C	0x11

%rdx	0x3

填写下表

操作数	值
%rax	
0x104	
$0x108	
(%rax)	
4(%rax)	
9(%rax,%rdx)	
260(%rcx,%rdx)	
0xFC(,%rcx,4)	
(%rax,%rdx,4)	

解：

操作数	值
%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

2.对于下面汇编代码的每一行，根据操作数，确定适当的指令后缀。

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

解：

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

3.当我们调用汇编器的时侯，下面代码的每一行都会产生一个错误消息。解释每一行都是哪里出了错。

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

解：

- 1.在 64 位模式下，只有 64 位通用寄存器（% rax、% rbx、% rcx、% rdx、% rsi、% rdi、% rbp、% rsp、% r8-% r15）可以作为基址寄存器。
- 2.movl 是 32 位指令（操作 32 位数据），但源操作数 %rax 是 64 位寄存器
- 3.x86 汇编的 mov 指令不支持内存到内存的直接数据传输，必须有一个操作数是寄存器（作为临时中转），两个内存操作数直接传递是不允许的
- 4.%sl 不是 x86-64 架构中的有效寄存器。对应 16 位寄存器 %si 的 8 位低字节应为 %sil（64 位模式下），%sl 是无效的寄存器名称。
- 5.立即数（$0x123）不能作为 mov 指令的目的操作数。立即数是常量，无法被修改，目的操作数必须是寄存器或内存地址。
- 6.movl 是 32 位指令（操作 32 位数据），但目的操作数 %rdx 是 64 位寄存器，寄存器位数（64 位）与指令后缀指定的操作数大小（32 位）不匹配，应使用 32 位寄存器 %edx
- 7.movb 是 8 位指令（操作 8 位数据），但源操作数 %si 是 16 位寄存器，寄存器位数（16 位）与指令后缀指定的操作数大小（8 位）不匹配，应使用 8 位寄存器 %sil

4.

练习题 3.4 假设变量 sp 和 dp 被声明为类型

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

这里 src_t 和 dest_t 是用 typedef 声明的数据类型。我们想使用适当的数据传送指令来实现下面的操作

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

假设 sp 和 dp 的值分别存储在寄存器%rdi 和%rsi 中。对于表中的每个表项，给出实现指定数据传送的两条指令。其中第一条指令应该从内存中读数，做适当的转换，并设置寄存器%rax 的适当部分。然后，第二条指令要把%rax 的适当部分写到内存。在这两种情况中，寄存器的部分可以是%rax、%eax、%ax 或%al，两者可以互不相同。

记住，当执行强制类型转换既涉及大小变化又涉及 C 语言中符号变化时，操作应该先改变大小(2.2.6 节)。

src_t	dest_t	指令
long	long	movq(%rdi),%rax movq %rax,(%rsi)
char	int	
char	unsigned	
unsigned char	long	
int	char	
unsigned	unsigned char	
char	short	

5.

练习题 3.5 已知信息如下。将一个原型为

```
void decode1(long *xp, long *yp, long *zp);
```

的函数编译成汇编代码，得到如下代码：

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

参数 xp、yp 和 zp 分别存储在对应的寄存器%rdi、%rsi 和%rdx 中。

请写出等效于上面汇编代码的 decode1 的 C 代码。