第 二 章　习　题 答 案

1. 给出以下概念的解释说明。

真值　机器数　数值数据　非数值数据　无符号整数　带符号整数
定点数　原码　补码　变形补码　溢出　浮点数
尾数　阶　阶码　移码　阶码下溢　阶码上溢
规格化数　左规　右规　非规格化数　机器零　非数(NaN)
BCD 码　逻辑数　ASCII 码　汉字输入码　汉字内码　机器字长
大端方式　小端方式　最高有效位　最高有效字节(MSB)　最低有效位
最低有效字节(LSB)　掩码　算术移位　逻辑移位　0 扩展
符号扩展　零标志 ZF　溢出标志 OF　符号标志 SF 进位/借位标志 CF

2. 简单回答下列问题。

（1）为什么计算机内部采用二进制表示信息？既然计算机内部所有信息都用二进制表示，为什么还要用到十六进制或八进制数？

（2）常用的定点数编码方式有哪几种？ 通常它们各自用来表示什么？

（3）为什么现代计算机中大多用补码表示带符号整数？

（4）在浮点数的基数和总位数一定的情况下，浮点数的表示范围和精度分别由什么决定？两者如何相互制约？

（5）为什么要对浮点数进行规格化？有哪两种规格化操作？

（6）为什么有些计算机中除了用二进制外还用 BCD 码来表示数值数据？

（7）为什么计算机处理汉字时会涉及到不同的编码（如，输入码、内码、字模码）？说明这些编码中哪些用二进制编码，哪些不用二进制编码，为什么？

3. 实现下列各数的转换。

（1） $(25.8125)_{10}=(?)_{2}=(?)_{8}=(?)_{16}$

（2） $(101101.011)_{2}=(?)_{10}=(?)_{8}=(?)_{16}=(?)_{8421}$

（3） $(0101\ 1001\ 0110.0011)_{8421}=(?)_{10}=(?)_{2}=(?)_{16}$

（4） $(4E.C)_{16}=(?)_{10}=(?)_{2}$

参考答案：

（1） $(25.8125)_{10}=(1\ 1001.1101)_{2}=(31.64)_{8}=(19.D)_{16}$

（2） $(101101.011)_{2}=(45.375)_{10}=(55.3)_{8}=(2D.6)_{16}=(0100\ 0101.0011\ 0111\ 0101)_{8421}$

（3） $(0101\ 1001\ 0110.0011)_{8421}=(596.3)_{10}=(1001010100.01001100110011\ldots)_{2}=(254.4CCC\ldots)_{16}$

（4） $(4E.C)_{16}=(78.75)_{10}=(0100\ 1110.11)_{2}$

4. 假定机器数为 8 位（1 位符号，7 位数值），写出下列各二进制数的原码表示。

+0.1001，–0.1001，+1.0，–1.0，+0.010100，–0.010100，+0，–0

参考答案：

（后面添 0）	原码
+0.1001：	0.1001000
–0.1001：	1.1001000
+1.0：	**溢出**
–1.0：	**溢出**
+0.010100：	0.0101000
–0.010100：	1.0101000
+0：	0.0000000
–0：	1.0000000

5. 假定机器数为 8 位（1 位符号，7 位数值），写出下列各二进制数的补码和移码表示。

+1001，–1001，+1，–1，+10100，–10100，+0，–0

参考答案：（假定移码的偏置常数为 128）

（前面添 0）	移码	补码
+1001：	10001001	00001001
–1001：	01110111	11110111
+1：	10000001	00000001
–1：	011111111	11111111
+10100：	10010100	00010100
–10100：	01101100	11101100
+0：	10000000	00000000
–0：	10000000	00000000

6. 已知 $[x]_{补}$，求 x

（1）$[x]_{补}$=11100111　（2）$[x]_{补}$=10000000　（3）$[x]_{补}$=01010010　（4）$[x]_{补}$=11010011

参考答案：

（1）$[x]_{补}$=11100111　　x = –0011001B = –25

（2）$[x]_{补}$=10000000　　x = –10000000B = –128

（3）$[x]_{补}$=01010010　　x = +1010010B = 82

（4）$[x]_{补}$=11010011　　x = – 0101101B = – 45

7. 某 32 位字长的机器中带符号整数用补码表示，浮点数用 IEEE 754 标准表示，寄存器 R1 和 R2 的内容分别为 R1：0000108BH，R2：8080108BH。不同指令对寄存器进行不同的操作，因而不同指令执行时寄存器内容对应的真值不同。假定执行下列运算指令时，操作数为寄存器 R1 和 R2 的内容，则 R1 和 R2 中操作数的真值分别为多少？

（1） 无符号整数加法指令

（2） 带符号整数乘法指令

（3） 单精度浮点数减法指令

参考答案：

R1 = 0000108BH = 0000 0000 0000 0000 0001 0000 1000 1011B

R2 = 8080108BH = 1000 0000 1000 0000 0001 0000 1000 1011B

（1）对于无符号数加法指令，R1 和 R2 中是操作数的无符号数表示，因此，其真值分别为 R1：108BH，R2：8080108BH。

（2）对于带符号整数乘法指令，R1 和 R2 中是操作数的带符号整数，即补码表示，由最高位可知，R1 为正数， R2 为负数。R1 的真值为+108BH，R2 的真值为–(111 1111 0111 1111 1110 1111 0111 0100b + 1b) = –7F7FEF75H。

（3）对于单精度浮点数减法指令，R1 和 R2 中是操作数的 IEEE 754 单精度浮点数表示。在 IEEE 754 标准中，单精度浮点数的位数为 32 位，其中包含 1 位符号位，8 位阶码，23 位尾数。

由 R1 中的内容可知，其符号位为 0，说明为正数，阶码为 0000 0000，尾数部分为 000 0000 0001 0000 1000 1011，故其为非规格化浮点数，**指数为–126**，尾数中没有隐藏的 1，用十六进制表示尾数为+0.002116H，故 R1 表示的真值为**+0.002116H × 2^{-126}**。

由 R2 中的内容可知，其符号位为 1，表示其为负数，阶码为 0000 0001，尾数部分为 000 0000 0001 0000 1000 1011，故其为规格化浮点数，**指数为 1–127 = –126**，尾数中有隐藏的 1，用十六进制表示尾数为–1.002116H，故 R2 表示的真值为**–1.002116H × 2^{-126}**

8. 假定机器 M 的字长为 32 位，用补码表示带符号整数。表 2.12 中第一列给出了在机器 M 上执行的 C

语言程序中的关系表达式，请参照已有的表栏内容完成表中后三栏内容的填写。

表 2.12 题 8 用表

关系表达式	运算类型	结果	说明
0 == 0U			
–1 < 0			
–1 < 0U	无符号整数	0	11…1B (2^{32}–1) > 00…0B(0)
2147483647 > –2147483647 – 1	有符号整数	1	011…1B (2^{31}–1) > 100…0B (–2^{31})
2147483647U > –2147483647 – 1			
2147483647 > (int) 2147483648U			
–1 > –2			
(unsigned) –1 > –2			

参考答案：

关系表达式	运算类型	结果	说明
0 == 0U	无符号整数	1	00…0B = 00…0B
–1 < 0	有符号整数	1	11…1B (–1) < 00…0B (0)
–1 < 0U	无符号整数	0	11…1B (2^{32}–1) > 00…0B(0)
2147483647 > –2147483647 – 1	有符号整数	1	011…1B (2^{31}–1) > 100…0B (–2^{31})
2147483647U > –2147483647 – 1	无符号整数	0	011…1B (2^{31}–1) < 100…0B(2^{31})
2147483647 > (int) 2147483648U	有符号整数	1	011…1B (2^{31}–1) > 100…0B (–2^{31})
–1 > –2	有符号整数	1	11…1B (–1) > 11…10B (–2)
(unsigned) –1 > –2	无符号整数	1	11…1B (2^{32}–1) > 11…10B (2^{32}–2)

9. 在 32 位计算机中运行一个 C 语言程序，在该程序中出现了以下变量的初值，请写出它们对应的机器数（用十六进制表示）。

（1）int x=−32768　　（2）short y=522　　（3）unsigned z=65530

（4）char c='@'　　（5）float a=−1.1　　（6）double b=10.5

参考答案：

（1）-2^{15}=−1000 0000 0000 0000B，故机器数为 1…1 1000 0000 0000 0000=FFFF8000H

（2）522=10 0000 1010B，故机器数为 0000 0010 0000 1010=020AH

（3）65530=2^{16}-1-5=1111 1111 1111 1010B，故机器数为 0000FFFAH

（4）'@'的 ASCII 码是 40H

（5）−1.1=−1.00011 [0011]…B=−1.000 1100 1100 1100 1100 1100B，阶码为 127+0=01111111，故机器数为 1 01111111 000 1100 1100 1100 1100 1100=BF8CCCCCH

（6）10.5=1010.1B=1.0101B×2^3，阶码为 1023+3=100 0000 0010，故机器数为 0 100 0000 0010 0101 [0000]=40250000 00000000H

10. 在 32 位计算机中运行一个 C 语言程序，在该程序中出现了一些变量，已知这些变量在某一时刻的机器数（用十六进制表示）如下，请写出它们对应的真值。

（1）int x：FFFF0006H　　（2）short y：DFFCH　　（3）unsigned z：FFFFFFFAH

（4）char c：2AH　　5）float a：C4480000H　　（6）double b：C024800000000000H

参考答案：

（1）FFFF0006H=1…1 0000 0000 0000 0110B，故 x= −1111 1111 1111 1010B= −(65535−5)=−65530

（2）DFFCH=1101 1111 1111 1100B=−010 0000 0000 0100B

故 y=−(8192+4)=−8196

（3）FFFFFFFAH=1…1 1010B，故 $z=2^{32}-6$

（4）2AH=0010 1010B，故 c=42，若 c 表示字符，则 c 为字符'*'

（5）C4480000H=1100 0100 0100 1000 0…0B，阶码为 10001000，阶为 136−127=9，尾数为−1.1001B，故 $a=-1.1001B\times2^9=-11\ 0010\ 0000B=-800$

（6）C024800000000000H=1100 0000 0010 0100 1000 0 0…0B，阶码为 100 0000 0010，阶为 1026−1023=3，尾数为 1.01001B，故 $b=-1.01001B\times2^3=-1010.01B=-10.25$

11. 以下给出的是一些字符串变量在内存中存放的字符串机器码，请根据 ASCII 码定义写出对应的字符串。指出代码 0AH 和 00H 对应的字符的含义。

（1）char *mystring1：68H 65H 6CH 6CH 6FH 2CH 77H 6FH 72H 6CH 64H 0AH 00H

（2）char *mystring2：77H 65H 20H 61H 72H 65H 20H 68H 61H 70H 70H 79H 21H 00H

参考答案：

字符串由字符组成，每个字符在内存中存放的是对应的 ASCII 码，因而可根据表 2.5 中的 ASCII 码和字符之间的对应关系写出字符串。

（1）mystring1 指向的字符串为：hello,world\n

（2）mystring2 指向的字符串为：we are happy!

其中，ASCII 码 00001010B=0AH 对应的是“换行”字符'\n'（LF）。每个字符串在内存存放时最后都会有一个“空”字符'\0'（NUL），其 ASCII 码为 00H。

12. 以下给出的是一些字符串变量的初值，请写出对应的机器码。

（1）char *mystring1="./myfile"　　（2）char *mystring2="OK, good!"

参考答案：

（1）mysring1 指向的存储区存放内容为：2EH 2FH 6DH 79H 66H 69H 6CH 65H 00H

（2）mysring2 指向的存储区存放内容为：4FH 4BH 2CH 67H 6FH 6FH 64H 21H 00H

13. 已知 C 语言中的按位异或运算（“XOR”）用符号“^”表示。对于任意一个位序列 a，$a\hat{}a=0$，C 语言程序可以利用这个特性来实现两个数值交换的功能。以下是一个实现该功能的 C 语言函数：

```
1  void xor_swap(int *x, int *y)
2   {
3       *y=*x ^ *y;   /* 第一步 */
4       *x=*x ^ *y;   /* 第二步 */
5       *y=*x ^ *y;   /* 第三步 */
6  }
```

假定执行该函数时*x 和*y 的初始值分别为 a 和 b，即*x=a 且*y=b，请给出每一步执行结束后，x 和 y 各自指向的内存单元中的内容分别是什么？

参考答案：

第一步结束后，x 和 y 指向的内存单元内容各为 a 和 a^b

第二步结束后，x 和 y 指向的内存单元内容各为 b 和 a^b

第三步结束后，x 和 y 指向的内存单元内容各为 b 和 a

14. 假定某个实现数组元素倒置的函数 reverse_array 调用了第 13 题中给出的 xor_swap 函数：

```
1  void reverse_array(int a[], int len)
2  {
3      int left, right=len-1;
4      for (left=0; left<=right; left++, right--)
5          xor_swap(&a[left], &a[right]);
6  }
```

当 *len* 为偶数时，reverse_array 函数的执行没有问题。但是，当 *len* 为奇数时，函数的执行结果不正确。请问，当 *len* 为奇数时会出现什么问题？最后一次循环中的 *left* 和 *right* 各取什么值？最后一次循环中调用 xor_swap 函数后的返回值是什么？对 reverse_array 函数作怎样的改动就可消除该问题？

参考答案：

当 len 为奇数时，最后一次循环执行的是将最中间的数与自己进行交换，即 left 和 right 都指向最中间数组元素，因而在调用 xor_swap 函数过程中的每一步执行*x ^ *y 时结果都是 0，并将 0 写入到了最中间的数组元素，从而改变了原来的数值。

可以将 for 循环中的终止条件改为“left<right”，这样，在 len 为奇数时最中间的数组元素不动。

15. 假设以下表 2.13 中的 *x* 和 *y* 是某 C 语言程序中的 char 型变量，请根据 C 语言中的按位运算和逻辑运算的定义，填写表 2.13，要求用十六进制形式填写。

表 2.13 题 15 用表

x	*y*	*x^y*	*x&y*	*x|y*	*~x|~y*	*x&!y*	*x&&y*	*x || y*	*!x || !y*	*x&&~y*
0x5F	0xA0									
0xC7	0xF0									
0x80	0x7F									
0x07	0x55									

参考答案：

表 2.13 题 15 用表

x	*y*	*x^y*	*x&y*	*x|y*	*~x|~y*	*x&!y*	*x&&y*	*x || y*	*!x || !y*	*x&&~y*
0x5F	0xA0	0xFF	0x00	0xFF	0xFF	0x00	0x01	0x01	0x00	0x01
0xC7	0xF0	0x37	0xC0	0xF7	0x3F	0x00	0x01	0x01	0x00	0x01
0x80	0x7F	0xFF	0x00	0xFF	0xFF	0x00	0x01	0x01	0x00	0x01
0x07	0xFF	0x07	0x07	0xFF	0xF8	0x00	0x01	0x01	0x00	0x00

16. 对于一个 *n*（*n*≥8）位的变量 *x*，请根据 C 语言中按位运算的定义，写出满足下列要求的 C 语言表达式。

（1）*x* 的最高有效字节不变，其余各位全变为 0。

（2）*x* 的最低有效字节不变，其余各位全变为 0。

（3）*x* 的最低有效字节全变为 0，其余各位取反。

（4）*x* 的最低有效字节全变 1，其余各位不变。

参考答案：

（1）(x>>(n-8))<<(n-8)

（2）x & 0xFF

（3）((x^ ~0xFF) >>8)<< 8

（3）x | 0xFF

17. 以下是一个由反汇编器生成的一段针对某个小端方式处理器的机器级代码表示文本，其中，最左边是指令所在的存储单元地址，冒号后面是指令的机器码，最右边是指令的汇编语言表示，即汇编指令。已知反汇编输出中的机器数都采用补码表示，请给出指令代码中划线部分表示的机器数对应的真值。

```
80483d2: 81 ec b8 01 00 00      sub   &0x1b8, %esp
80483d8: 8b 55 08               mov   0x8(%ebp), %edx
80483db: 83 c2 14               add   $0x14, %edx
80483de: 8b 85 58 fe ff ff      mov   0xfffffe58(%ebp), %eax
80483e4: 03 02                  add   (%edx), %eax
80483e6: 89 85 74 fe ff ff      mov   %eax, 0xfffffe74(%ebp)
```

```
80483ec: 8b 55 08              mov   0x8(%ebp), %edx
80483ef: 83 c2 44              add   $0x44, %edx
80483f2: 8b 85 c8 fe ff ff     mov   0xfffffec8(%ebp), %eax
80483f8: 89 02                 mov   %eax, (%edx)
80483fa: 8b 45 10              mov   0x10(%ebp), %eax
80483fd: 03 45 0c              add   0xc(%ebp), %eax
8048400: 89 85 ec fe ff ff     mov   %eax, 0xfffffeec(%ebp)
8048406: 8b 45 08              mov   0x8(%ebp), %eax
8048409: 83 c0 20              add   $0x20, %eax
```

参考答案：

b8 01 00 00：机器数为 000001B8H，真值为+1 1011 1000B = 440

14：机器数为 14H，真值为+1 0100B = 20

58 fe ff ff：机器数为 FFFFFE58H，真值为-1 1010 1000B = -424

74 fe ff ff：机器数为 FFFFFE74H，真值为-1 1000 1100B = -396

44：机器数为 44H，真值为+100 0100B=68

c8 fe ff ff：机器数为 FFFFFEC8H，真值为-1 1000 1100B = -312

10：机器数为 10H，真值为+10000B=16

0c：机器数为 0CH，真值为+1100B=12

ec fe ff ff：机器数为 FFFFFEECH，真值为-1 0001 0100B = -276

20：机器数为 20H，真值为+00100000B=32

18. 假设以下 C 语言函数 compare_str_len 用来判断两个字符串的长度，当字符串 *str*1 的长度大于 *str*2 的长度时函数返回值为 1，否则为 0。

```
1   int compare_str_len(char *str1, char *str2)
2   {
3       return strlen(str1) - strlen(str2) > 0;
4   }
```

已知 C 语言标准库函数 strlen 原型声明为“size_t strlen(const char *s);”，其中，size_t 被定义为 unsigned int 类型。请问：函数 compare_str_len 在什么情况下返回的结果不正确？为什么？为使函数正确返回结果应如何修改代码？

参考答案：

因为 size_t 被定义为 unsigned int 类型，因此，库函数 strlen 的返回值为无符号整数。函数 compare_str_len 中的返回值是 strlen(str1) - strlen(str2) > 0，这个关系表达式中>号左边是两个无符号数相减，其差还是无符号整数，因而总是大于等于 0，也即在 str1 的长度小于 str2 的长度时结果也为 1。显然，这是错误的。

只要将第 3 行语句改为以下形式即可：

```
3       return strlen(str1) > strlen(str2) ;
```

19. 考虑以下 C 语言程序代码：

```
1   int func1(unsigned word)
2   {
3       return  (int) (( word <<24) >> 24);
4   }
5   int func2(unsigned word)
6   {
7       return  ( (int) word <<24 ) >> 24;
8   }
```

假设在一个 32 位机器上执行这些函数，该机器使用二进制补码表示带符号整数。无符号数采用逻辑移位，带符号整数采用算术移位。请填写表 2.14，并说明函数 func1 和 func2 的功能。

表 2.14 题 19 用表

w	func1(*w*)	func2(*w*)

机器数	值	机器数	值	机器数	值
	127				
	128				
	255				
	256				

参考答案：

函数 func1 的功能是把无符号数高 24 位清零（左移 24 位再逻辑右移 24 位），当成带符号整数返回时，结果一定是正数；函数 func2 的功能是把无符号数的高 24 位都变成和第 25 位一样，因为左移 24 位后进行算术右移，高 24 位补符号位（即第 25 位），当成带符号整数返回时，结果可能是正数也可能是负数。

W		func1(w)		func2(w)	
机器数	值	机器数	值	机器数	值
0000 007FH	127	0000 007FH	+127	0000 007FH	+127
0000 0080H	128	0000 0080H	+128	FFFF FF80H	–128
0000 00FFH	255	0000 00FFH	+255	FFFF FFFFH	–1
0000 0100H	256	0000 0000H	0	0000 0000H	0

20．填写表 2.15，注意对比无符号整数和带符号整数的乘法结果，以及截断操作前、后的结果。

表 2.15 题 20 用表

模式	*x*		*y*		*x*×*y*（截断前）		*x*×*y*（截断后）	
	机器数	值	机器数	值	机器数	值	机器数	值
无符号	110		010					
带符号	110		010					
无符号	001		111					
带符号	001		111					
无符号	111		111					
带符号	111		111					

参考答案：

模式	x		y		x×y（截断前）		x×y（截断后）	
	机器数	值	机器数	值	机器数	值	机器数	值
无符号数	110	6	010	2	001 100	12	100	4
二进制补码	110	–2	010	+2	111 100	–4	100	–4
无符号数	001	1	111	7	000 111	7	111	7
二进制补码	001	+1	111	–1	111 111	–1	111	–1
无符号数	111	7	111	7	110 001	49	001	1
二进制补码	111	–1	111	–1	000 001	+1	001	+1

21．以下是两段 C 语言代码，函数 arith( )是直接用 C 语言写的，而 optarith( )是对 arith( )函数以某个确定的 *M* 和 *N* 编译生成的机器代码反编译生成的。根据 optarith( )，可以推断函数 arith( ) 中 *M* 和 *N* 的值各是多少？

```
#define    M
```

```
#define     N
int     arith (int x,    int y)
{
        int    result = 0 ;
        result = x*M + y/N;
        return result;
}

int optarith ( int x,    int y)
{
        int    t = x;
        x << = 4;                          x'<-X*16
        x – = t;                           x'<-X*15
        if ( y < 0 )    y += 3;
        y=>>2;                             y'<-Y/4
        return x+y;
}
```

参考答案：

可以看出 x*M 和“int t = x; x << = 4; x-=t;”三句对应，这些语句实现了 x 乘 15 的功能（左移 4 位相当于乘以 16，然后再减 1），因此，M 等于 15；

y/N 与“if (y < 0) y += 3; y>>2;”两句对应，第二句“y 右移 2 位”实现了 y 除以 4 的功能，因此 N 是 4。而第一句“if (y < 0) y += 3;”主要用于对 y= –1 时进行调整，若不调整，则–1>>2= –1 而–1/4=0，两者不等；调整后 –1+3=2，2>>2=0，两者相等。

22. 下列几种情况所能表示的数的范围是什么？

（1） 16 位无符号整数

（2） 16 位补码表示的带符号整数

（3） 下述格式的浮点数（基数为 2，移码的偏置常数为 128）

数符	阶码	尾数
1 位	8 位移码	7 位原码数值部分

参考答案：

（1）16 位无符号整数：0～65535

（2）16 位补码表示的整数：–32768 ～ +32767

（3）浮点数：负数：$-(1-2^{-7})\times 2^{+127}$ ～ $-2^{-7}\times 2^{-128}$

正数：$+2^{-7}\times 2^{-128}$ ～ $(1-2^{-7})\times 2^{+127}$

23. 以 IEEE 754 单精度浮点数格式表示下列十进制数。

+1.75，+19，–1/8，258

参考答案：

+1.75 = +1.11B = 1.11B × 2^0，故阶码为 0+127=01111111B，数符为 0，尾数为 1.110…0，小数点前为隐藏位，所以+1.7 表示为 0 01111111 110 0000 0000 0000 0000 0000，用十六进制表示为 3FE00000H。

+19 = +10011B = +1.0011B × 2^4，故阶码为 4+127 = 10000011B，数符为 0，尾数为 1.00110…0，所以+19 表示为 0 10000011 001 1000 0000 0000 0000 0000，用十六进制表示为 41980000H。

–1/8 = – 0.125 = – 0.001B = – 1.0 × 2^{-3}，阶码为–3+127 = 01111100B，数符为 1，尾数为 1.0…0，所以–1/8 表示为 1 01111100 000 0000 0000 0000 0000 0000，用十六进制表示为 BE000000H。

258=100000010B=1.0000001B × 2^8，故阶码为 8+127=10000111B，数符为 0，尾数为 1.0000001，所以

258 表示为 0 10000111 000 0001 0000 0000 0000 0000，用十六进制表示为 43810000H。

24. 设一个变量的值为 4098，要求分别用 32 位补码整数和 IEEE 754 单精度浮点格式表示该变量（结果用十六进制形式表示），并说明哪段二进制位序列在两种表示中完全相同，为什么会相同？

参考答案：

4098 = +1 **0000 0000 0010**B = +1. 0000 0000 001 × 2^{12}

32 位 2-补码形式为：0000 0000 0000 0000 0001 **0000 0000 0010** （00001002H）

IEEE754 单精度格式为：0 10001011 **0000 0000 0010** 0000 0000 000 （45801000H）

粗体部分为除隐藏位外的有效数字，因此，在两种表示中是相同的位序列。

25. 设一个变量的值为–2 147 483 647（提示：2 147 483 647=2^{31}–1），要求分别用 32 位补码整数和 IEEE754 单精度浮点格式表示该变量（结果用十六进制形式表示），并说明哪种表示其值完全精确，哪种表示的是近似值。

参考答案：

–2147483647 = –111 1111 1111 1111 1111 1111 1111 1111B

= –1.11 1111 1111 1111 1111 1111 1111 1111 × 2^{30}

32 位补码形式为：1000 0000 0000 0000 0000 0000 0000 0001 （80000001H）

IEEE 754 单精度格式为：1 10011101 1111 1111 1111 1111 1111 111 （CEFFFFFFH）

32 位补码形式能表示精确的值，而浮点数表示的是近似值，因为低 7 位被截断了。

26. 下表给出了有关 IEEE 754 浮点格式表示中一些重要的非负数的取值，表中已经有最大规格化数的相应内容，要求填入其他浮点数格式的相应内容。

表 2.16 题 26 用表

项目	阶码	尾数	单精度		双精度	
			以 2 的幂次表示的值	以 10 的幂次表示的值	以 2 的幂次表示的值	以 10 的幂次表示的值
0						
1						
最大规格化数	11111110	1…11	$(2-2^{-23})\times2^{127}$	3.4×10^{38}	$(2-2^{-52})\times2^{1023}$	1.8×10^{308}
最小规格化数						
最大非规格化数						
最小非规格化数						
+∞						
NaN						

参考答案：

表 2.16 题 26 用表

项目	阶码	尾数	单精度		双精度	
			以 2 的幂次表示的值	以 10 的幂次表示的值	以 2 的幂次表示的值	以 10 的幂次表示的值
0	00000000	0…00	0	0	0	0
1	01111111	0…00	1	1	1	1
最大规格化数	11111110	1…11	$(2-2^{-23})\times2^{127}$	3.4×10^{38}	$(2-2^{-52})\times2^{1023}$	1.8×10^{308}
最小规格化数	00000001	0…00	1.0×2^{-126}	1.2×10^{-38}	1.0×2^{-1022}	2.2×10^{-308}
最大非规格化数	00000000	1…11	$(1-2^{-23})\times2^{-126}$	1.2×10^{-38}	$(1-2^{-52})\times2^{-1022}$	2.2×10^{-308}
最小非规格化数	00000000	0…01	$2^{-23}\times2^{-126}=2^{-149}$	1.4×10^{-45}	$2^{-52}\times2^{-1022}$	4.9×10^{-324}
+∞	11111111	0…00	–	–	–	–

NaN	11111111	非全 0	–	–	–	–

27．已知下列字符编码：A 为 100 0001，a 为 110 0001，0 为 011 0000，求 E、e、f、7、G、Z、5 的 7 位 ACSII 码和在第一位前加入奇校验位后的 8 位编码。

参考答案：

E 的 ASCII 码为‘A’ + (‘E’ – ‘A’) = 100 0001 + 100 = 100 0101，奇校验位 P = 0，第一位前加入奇校验位后的 8 位编码是 0 100 0101。

e 的 ASCII 码为‘a’+ (‘e’ – ‘a’) = 110 0001 + 100 = 110 0101，奇校验位 P = 1，第一位前加入奇校验位后的 8 位编码是 1 110 0101。

f 的 ASCII 码为‘a’+ (‘f’ – ‘a’) = 110 0001 + 101 = 110 0110，奇校验位 P = 1，第一位前　加入奇校验位后的 8 位编码是 1 110 0110。

7 的 ASCII 码为‘0’+ (7 – 0)　= 011 0000 + 111 = 011 0111，奇校验位 P = 0，第一位前加入奇校验位后的 8 位编码是 0 011 0111。

G 的 ASCII 码为‘A’+ (‘G’ – ‘A’) = 100 0001 + 0110 = 100 0111，奇校验位 P = 1，第一位前加入奇校验位后的 8 位编码是 1 100 0111。

Z 的 ASCII 码为‘A’ +(‘Z’ – ‘A’) = 100 0001 + 11001 = 101 1010，奇校验位 P = 1，第一位前加入奇校验位后的 8 位编码是 1 101 1010。

5 的 ASCII 码为‘0’+(5 – 0) = 011 0000 + 101 = 011 0101，奇校验位 P = 1，第一位前加入奇校验位后的 8 位编码是 1 011 0101。

28．假定在一个程序中定义了变量 *x*、*y* 和 *i*，其中，*x* 和 *y* 是 float 型变量（用 IEEE754 单精度浮点数表示），*i* 是 16 位 short 型变量（用补码表示）。程序执行到某一时刻，*x*= –0.125、*y*=7.5、*i*=100，它们都被写到了主存（按字节编址），其地址分别是 100，108 和 112。请分别画出在大端机器和小端机器上变量 *x*、*y* 和 *i* 中每个字节在主存的存放位置。

参考答案：

$-0.125 = -0.001B = -1.0 \times 2^{-3}$

x 在机器内部的机器数为：1　01111100　00…0 (BE00 0000H)

$7.5 = +111.1B = +1.111 \times 2^{2}$

y 在机器内部的机器数为：0　10000001　11100…0 (40F0 0000H)

100=64+32+4=1100100B

i 在机器内部表示的机器数为：0000 0000 0110 0100（0064H）

	大端机	小端机
地址	内容	内容
100	BEH	00H
101	00H	00H
102	00H	00H
103	00H	BEH
108	40H	00H
109	F0H	00H
110	00H	F0H
111	00H	40H
112	00H	64H
113	64H	00H

29．对于图 2.6，假设 *n*=8，机器数 *X* 和 *Y* 的真值分别是 *x* 和 *y*。请按照图 2.6 的功能填写表 2.17，并给出对每个结果的解释。要求机器数用十六进制形式填写，真值用十进制形式填写。

表 2.17 题 29 用表

表示	*X*	*x*	*Y*	*y*	*X*+*Y*	*x*+*y*	OF	SF	CF	*X*−*Y*	*x*−*y*	OF	SF	CF
无符号	0xB0		0x8C											
带符号	0xB0		0x8C											
无符号	0x7E		0x5D											
带符号	0x7E		0x5D											

参考答案：

表 2.17 题 29 用表

表示	*X*	*x*	*Y*	*y*	*X*+*Y*	*x*+*y*	OF	SF	CF	*X*−*Y*	*x*−*y*	OF	SF	CF
无符号	0xB0	176	0x8C	140	0x3C	60	1	0	1	0x24	36	0	0	0
带符号	0xB0	-80	0x8C	-116	0x3C	60	1	0	1	0x24	36	0	0	0
无符号	0x7E	126	0x5D	93	0xDB	219	1	1	0	0x21	33	0	0	0
带符号	0x7E	126	0x5D	93	0xDB	-37	1	1	0	0x21	33	0	0	0

（1）无符号整数 176+140=316，无法用 8 位表示，即结果应有进位，CF 应为 1。验证正确。
（2）无符号整数 176−140=36，可用 8 位表示，即结果没有进位，CF 应为 0。验证正确。
（3）带符号整数−80+(−116)=−316，无法用 8 位表示，即结果溢出，OF 应为 1。验证正确。
（4）带符号整数−80−(−116)=36，可用 8 位表示，即结果不溢出，OF 应为 0。验证正确。
（5）无符号整数 126+93=219，可用 8 位表示，即结果没有进位，CF 应为 0。验证正确。
（6）无符号整数 126−93=33，可用 8 位表示，即结果没有进位，CF 应为 0。验证正确。
（7）带符号整数 126+93=219，无法用 8 位表示，即结果溢出，OF 应为 1。验证正确。
（8）带符号整数 126−93=33，可用 8 位表示，即结果不溢出，OF 应为 0。验证正确。
无符号整数的加减运算的结果是否溢出，通过进位/借位标志 CF 来判断，而带符号整数的加减运算结果是否溢出，通过溢出标志 OF 来判断。

30. 在字长为 32 位的计算机上，有一个函数其原型声明为“int ch_mul_overflow(int x, int y);”，该函数用于对两个 int 型变量 *x* 和 *y* 的乘积判断是否溢出，若溢出则返回 1，否则返回 0。请使用 64 位精度的整数类型 long long 来编写该函数。

参考答案：

使用 64 位精度的乘法实现两个 32 位带符号整数相乘（专门的补码乘法运算），可以通过乘积的高 32 位和低 32 位的关系来进行溢出判断。判断规则是：**若高 32 位中每一位都与低 32 位的最高位相同，则不溢出；否则溢出。**

```
1    int ch_mul_overflow(int x, int y)
2    {
3        long long prod_64= (long long) x*y;
4        return prod_64 != (int) prod_64;
5    }
```

第 3 行赋值语句的右边采用强制类型转换，使得 x 和 y 相乘的结果强制以 64 位乘积的形式保留在 64 位 long long 型变量 prod_64 中。在第 4 行关系运算符!=右边的强制类型转换，将一个 64 位乘积的高 32 位丢弃，然后，在进行关系运算时，因为关系运算符!=的左边是一个 64 位整数，所以，右边的 32 位数必须再进行符号扩展以转换为 64 位整数，然后再与左边的整数进行比较。若乘积没有溢出，则丢弃的高 32 位和后面符号扩展的 32 位相同，因而比较结果一定是相等，返回为 0；若乘积有溢出，则比较结果一定不相等，返回为 1。

若第 3 行赋值语句改成如下形式，则 prod_64 得到的是低 32 位乘积进行符号扩展后的 64 位值，因此，当结果溢出时，prod_64 中得到的并不是正确的 64 位乘积。

3　　long long prod_64= x*y;

31．对于第 2.7.5 节中例 2.31 存在的整数溢出漏洞，如果将其中的第 5 行改为以下两个语句：

unsigned long long arraysize=count*(unsigned long long)sizeof(int);

int *myarray = (int *) malloc(arraysize);

已知 C 语言标准库函数 malloc 的原型声明为“void *malloc(size_t size);”，其中，size_t 定义为 unsigned int 类型，则上述改动能否消除整数溢出漏洞？若能则说明理由；若不能则给出修改方案。

参考答案：

上述改动无法消除整数溢出漏洞，这种改动方式虽然使得 arraysize 的表示范围扩大了，避免了 arraysize 的溢出，不过，当调用 malloc 函数时，若 arraysize 的值大于 32 位的 unsigned int 的最大可表示值，则 malloc 函数还是只能按 32 位数给出的值去申请空间，同样会发生整数溢出漏洞。

程序应该在调用 malloc 函数之前检测所申请的空间大小是否大于 32 位无符号整数的可表示范围，若是，则返回-1，表示不成功；否则再申请空间并继续进行数组复制。修改后的程序如下：

```
1   /* 复制数组到堆中，count 为数组元素个数 */
2   int copy_array(int *array, int count) {
3       int i;
4    /* 在堆区申请一块内存 */
5       unsigned long long arraysize=count*(unsigned long long)sizeof(int);
6       size_t myarraysize=(size_t) arraysize;
7       if (myarraysize!=arraysize)
8           return -1;
9       int *myarray = (int *) malloc(myarraysize);
10      if (myarray == NULL)
11          return -1;
12      for (i = 0; i < count; i++)
13          myarray[i] = array[i];
14      return count;
15  }
```

32. 已知一次整数加法、一次整数减法和一次移位操作都只需一个时钟周期，一次整数乘法操作需要 10 个时钟周期。若 x 为一个整型变量，现要计算 55*x，请给出一种计算表达式，使得所用时钟周期数最少。

参考答案：

$55*x=(64-8-1)*x=64*x-8*x-x$

根据上述表达式，只要两次移位操作和两次减法操作，共 4 个时钟周期。

若将 55 分解为 32+16+4+2+1，则需要 4 次移位操作和 4 次加法操作，共 8 个时钟周期。

上述两种方式都比直接执行一次乘法操作所用的时钟周期数少。

33．假设 x 为一个 int 型变量，请给出一个用来计算 $x/32$ 的值的函数 div32。要求不能使用除法、乘法、模运算、比较运算、循环语句和条件语句，可以使用右移、加法以及任何按位运算。

参考答案：

根据第 2.7.6 节内容可知，带符号整数 x 除以 2^k 的值可以用移位方式实现。若 x 为正数，则将 x 右移 k 位得到商；若 x 为负数，则 x 需要加一个偏移量(2^k-1)后再右移 k 位得到商。因此，在执行右移操作前必须先计算偏移量，计算公式如下：

$$b=\begin{cases}0 & x>=0\\ 31 & x<0\end{cases}$$

x 的符号位在最左边，因此，表达式 x>>31 的计算结果得到 32 个符号位，x 小于 0 时为 32 个 1，否则为 32 个 0。偏移量 b 可以通过用掩码的方式得到。

函数 div32 的 C 语言源代码如下：

```
int div32(int x)
{    /* 根据 x 的符号得到偏移量 b */
     int b=(x>>31) & 0x1F;
     return (x+b)>>5;
}
```

34. 无符号整数变量 *ux* 和 *uy* 的声明和初始化如下：

```
unsigned ux=x;
unsigned uy=y;
```

若 sizeof(int)=4，则对于任意 int 型变量 *x* 和 *y*，判断以下关系表达式是否永真。若永真则给出证明；若不永真则给出结果为假时 *x* 和 *y* 的取值。

（1）(x*x) >= 0　　（2）(x−1<0) || x>0

（3）x<0 || −x<=0　　（4）x>0 || −x>=0

（5）x&0xf!=15 || (x<<28)<0　　（6）x>y==(−x<−y)

（7）~x+~y==~(x+y)　　（8）(int) (ux−uy) == −(y−x)

（9）((x>>2)<<2) <= x　　（10）x*4+y*8==(x<<2)+(y<<3)

（11）x/4+y/8==(x>>2)+(y>>3)　　（12）x*y==ux*uy

（13）x+y==ux+uy　　（14）x*~y+ux*uy==−x

参考答案：

（1）(x*x)>=0

非永真。例如，x=65 534 时，则 $x*x=(2^{16}-2)*(2^{16}-2)=2^{32}-2*2*2^{16}+4 \pmod{2^{32}}=-(2^{18}-4)=-262140$。x 的机器数为 0000FFFEH，x*x 的机器数为 FFFC0004H。

（2）(x−1<0) || x>0

非永真。当 x=−2 147 483 648 时，显然，x<0，机器数为 80000000H，x−1 的机器数为 7FFFFFFFH，符号位为 0，因而 x−1>0。 此时，(x-1<0) 和 x>0 两者都不成立。

（3）x<0 || −x<=0

永真。若 x>0，x 符号位为 0 且数值部分为非 0（至少有一位是 1），从而使−x 的符号位一定是 1，即则−x<0；若 x=0，则−x=0。综上，只要 x<0 为假，则−x<=0 一定为真，因而是永真。

（4）x>0 || −x>=0

非永真。当 x=−2 147 483 648 时，x<0，且 x 和−x 的机器数都为 80000000H，即−x<0。此时，x>0 和−x>=0 两者都不成立。

（5）x&0xf!=15 || (x<<28)<0

非永真。这里!=的优先级比&（按位与）的优先级高。因此，若 x=0，则 x&0xf!=15 为 0，(x<<28)<0 也为 0，所以结果为假。

（6）x>y==(−x<−y)

非永真。当 x=−2 147 483 648、y 任意（除−2 147 483 648 外），或者 y=−2 147 483 648、x 任意（除−2 147 483 648 外）时不等。因为 int 型负数−2 147 483 648 是最小负数，该数取负后结果仍为−2 147 483 648，而不是 2 147 483 648。

（7）~x+~y==~(x+y)

永假。$[-x]_{补}=\sim[x]_{补}+1$，$[-y]_{补}=\sim[y]_{补}+1$，故$\sim[x]_{补}+\sim[y]_{补}=[-x]_{补}+[-y]_{补}-2$。

$[-(x+y)]_{补}=\sim[x+y]_{补}+1$，故$\sim[x+y]_{补}=[-(x+y)]_{补}-1=[-x]_{补}+[-y]_{补}-1$。

由此可见，左边比右边少 1。

（8）(int) (ux−uy) ==−(y−x)

永真。(int) ux−uy=$[x-y]_{补}$=$[x]_{补}$+$[-y]_{补}$=$[-y+x]_{补}$= $[-(y-x)]_{补}$

（9）((x>>2)<<2) <= x

永真。因为右移总是向负无穷大方向取整。

(10)x*4+y*8==(x<<2)+(y<<3)

永真。因为带符号整数 x 乘以 2^k 完全等于 x 左移 k 位，无论结果是否溢出。

(11)x/4+y/8==(x>>2)+(y>>3)

非永真。当 x=−1 或 y=−1 时，x/4 或 y/8 等于 0，但是，因为−1 的机器数为全 1，所以，x>>2 或y>>3 还是等于−1。此外，当 x 或 y 为负数且 x 不能被 4 整除或 y 不能被 8 整除，则 x/4 不等于x>>2，y/8 不等于 y>>3。

(12)x*y==ux*uy

永真。根据第 2.7.5 节内容可知，x*y 的低 32 位和 ux*uy 的低 32 位是完全一样的位序列。

(13)x+y==ux+uy

永真。根据第 2.7.4 节内容可知，带符号整数和无符号整数都是在同一个整数加减运算部件中进行运算的，x 和 ux 具有相同的机器数，y 和 uy 具有相同的机器数，因而 x+y 和 ux+uy 具有完全一样的位序列。

(14)x*~y+ux*uy==−x　　~y=-uy-1　x*~y=ux*（-uy-1)=-ux*uy-ux

永真。−y=~y+1，即~y=−y−1。而 ux*uy=x*y，因此，等式左边为 x*(−y−1)+x*y=−x。

35. 变量 *dx*、*dy* 和 *dz* 的声明和初始化如下：

```
double dx = (double) x;
double dy = (double) y;
double dz = (double) z;
```

若 float 和 double 分别采用 IEEE 754 单精度和双精度浮点数格式，sizeof(int)=4，则对于任意 int 型变量 *x*、*y* 和 *z*，判断以下关系表达式是否永真。若永真则给出证明；若不永真则给出结果为假时 *x* 和 *y* 的取值。

（1）dx*dx >= 0

（2）(double)(float) x == dx

（3）dx+dy == (double) (x+y)

（4）(dx+dy)+dz == dx+(dy+dz)

（5）dx*dy*dz == dz*dy*dx

（6）dx/dx == dy/dy

参考答案：

（1）dx*dx >= 0

永真。double 型数据用 IEEE 754 标准表示，尾数用原码小数表示，符号和数值部分分开运算。不管结果是否溢出都不会影响乘积的符号。

（2）(double)(float) x == dx

非永真。当 int 型数据 x 的有效位数比 float 型可表示的最大有效位数 24 更多时，x 强制转换为 float 型数据时有效位数丢失，而将 x 转换为 double 型数据时没有有效位数丢失。也即等式左边可能是近似值，而右边是精确值。

（3）dx+dy == (double) (x+y)

非永真。因为 x+y 可能会溢出，而 dx+dy 不会溢出。

（4）(dx+dy)+dz == dx+(dy+dz)

永真。因为 dx、dy 和 dz 是由 32 位 int 型数据转换得到的，而 double 类型可以精确表示 int 类型数据，并且对阶时尾数移位位数不会超过 52 位，因此尾数不会舍入，因而不会发生大数吃小数的情况。

但是，如果 dx、dy 和 dz 是任意 double 类型数据，则非永真。

（5）dx*dy*dz == dz*dy*dx

非永真。相乘的结果可能产生舍入。

（6）dx/dx == dy/dy

非永真。dx 和 dy 中只要有一个为 0、另一个不为 0 就不相等。

36. 在 IEEE 754 浮点数运算中，当结果的尾数出现什么形式时需要进行左规，什么形式时需要进行右规？如何进行左规，如何进行右规？

参考答案：

（1）对于结果为±1x. xx…x 的情况，需要进行右规。右规时，尾数右移一位，阶码加 1。右规操作可以表示为：$M_b \leftarrow M_b \times 2^{-1}$，$E_b \leftarrow E_b+1$。右规时注意以下两点：

① 尾数右移时，最高位“1”被移到小数点前一位作为隐藏位，最后一位移出时，要考虑舍入。

② 阶码加 1 时，直接在末位加 1。

（2）对于结果为±0.00…01x…x 的情况，需要进行左规。左规时，数值位逐次左移，阶码逐次减 1，直到将第一位“1”移到小数点左边。假定 k 为结果中“±”和最左边第一个 1 之间连续 0 的个数，则左规操作可以表示为：$M_b \leftarrow M_b \times 2^k$，$E_b \leftarrow E_b-k$。左规时注意以下两点：

① 尾数左移时数值部分最左边 k 个 0 被移出，因此，相对来说，小数点右移了 k 位。因为进行尾数相加时，默认小数点位置在第一个数值位（即：隐藏位）之后，所以小数点右移 k 位后被移到了第一位 1 后面，这个 1 就是隐藏位。

② 执行 $E_b \leftarrow E_b-k$ 时，每次都在末位减 1，一共减 k 次。

37. 在 IEEE 754 浮点数运算中，如何判断浮点运算的结果是否溢出？

参考答案：

浮点运算结果是否溢出，并不以尾数的溢出情况来判断，而主要看阶码是否溢出。尾数溢出时，可通过右规操作进行纠正。阶码上溢时，说明结果的数值太大，无法表示；阶码下溢时，说明结果数值太小，可以把结果近似为 0。

在进行对阶、规格化、舍入和浮点数的乘/除运算等过程中，都需要对阶码进行加、减运算，因而可能会发生阶码上溢或阶码下溢的情况，因此，必须对阶码进行溢出判断。

（有关对阶码进行溢出判断的方法可参见教材中相关章节。）

38. 分别给出不能精确用 IEEE 754 单精度和双精度格式表示的最小正整数。

参考答案：

一个整数如果有效位数大于浮点表示格式中可表示的有效位数时就不能用浮点格式精确表示。因此，当一个整数的有效位数大于 24，则不能用 IEEE 754 单精度格式精确表示，有效位数大于 24 的最小正整数是 $10\cdots01=2^{24}+1=16\ 777\ 217$；当一个整数的有效位数大于 53，则不能 IEEE 754 双精度格式精确表示。有效位数大于 53 的最小正整数是 $10\cdots01=2^{53}+1$。

39. 采用 IEEE 754 单精度浮点数格式计算下列表达式的值。

（1）0.75+(– 65.25)　　（2）0.75–(– 65.25)

参考答案：

$x = 0.75 = 0.110...0B = (1.10...0)_2 \times 2^{-1}$，$y = – 65.25 = – 1000001.01000...0B = (–1.00000101...0)_2 \times 2^6$

用 IEEE 754 标准单精度格式表示为：

$[x]_{浮}$ = 0 01111110 10...0　　$[y]_{浮}$ = 1 10000101 000001010...0

所以，E_x= 01111110，M_x= 0 (1). 1...0 ，E_y= 10000101，M_y = 1(1).000001010...0

尾数 M_x 和 M_y 中小数点前面有两位，第一位为数符，第二位加了括号，是隐藏位“1”。

以下是计算机中进行浮点数加减运算的过程（假定保留 2 位附加位：保护位和舍入位）

（1）0.75+ (– 65.25）

① 对阶：$[\Delta E]_{补}=E_x+[–E_y]_{补} \pmod{2^n} = 0111\ 1110 + 0111\ 1011 = 1111\ 1001$，即 $\Delta E = –7$。

根据对阶规则可知需对 x 进行对阶，结果为：$E_x = E_y = 10000101$，$M_x = 00.000000110...000$

M_x 右移 7 位，符号不变，数值高位补 0，隐藏位右移到小数点后面，最后移出的 2 位保留

② 尾数相加：$M_b = M_x + M_y = 00.000000110...000 + 11.000001010...000$ （注意小数点在隐藏位后）

根据原码加/减法运算规则，得：$00.000000110...000 + 11.000001010...000 = 11.000000100…000$

上式尾数中最左边第一位是符号位，其余都是数值部分，尾数后面两位是附加位（加粗）。

③ 规格化：尾数数值部分最高位为 1，因此不需要进行规格化。

④ 舍入：把结果的尾数 M_b 中最后两位附加位舍入掉，从本例来看，不管采用什么舍入法，结果都一样，都是把最后两个 0 去掉，得：$M_b = 11.000000100…0$

⑤ 溢出判断：在上述阶码计算和调整过程中，没有发生“阶码上溢”和“阶码下溢”的问题。因此，阶码 $E_b = 10000101$。

最后结果为 $E_b = 10000101$，$M_b = 1(1).00000010…0$，即：– 64.5。

（2） 0.75–(– 65.25）

① 对阶：$[\Delta E]_{补} = E_x + [-E_y]_{补} \pmod{2^n} = 0111\ 1110 + 0111\ 1011 = 1111\ 1001$，$\Delta E = -7$。

根据对阶规则可知需要对 x 进行对阶，结果为：$E_x = E_y = 10000110$，$M_x = 00.000000110...000$

M_x 右移一位，符号不变，数值高位补 0，隐藏位右移到小数点后面，最后移出的位保留

② 尾数相加：$M_b = M_x - M_y = 00.000000110...000 - 11.000001010...000$ （注意小数点在隐藏位后）

根据原码加/减法运算规则，得：$00.000000110...000 - 11.000001010...000 = 01.00001000…000$

上式尾数中最左边第一位是符号位，其余都是数值部分，尾数后面两位是附加位（加粗）。

③ 规格化：尾数数值部分最高位为 1，不需要进行规格化。

④ 舍入：把结果的尾数 M_b 中最后两位附加位舍入掉，从本例来看，不管采用什么舍入法，结果都一样，都是把最后两个 0 去掉，得：$M_b = 01.00001000…0$

⑤ 溢出判断：在上述阶码计算和调整过程中，没有发生“阶码上溢”和“阶码下溢”的问题。因此，阶码 $E_b = 10000101$。

最后结果为 $E_b = 10000101$，$M_b = 0(1).00001000…0$，即：+66。

40. 以下是函数 fpower2 的 C 语言源程序，它用于计算 2^x 的浮点数表示，其中调用了函数 u2f，u2f 用于将一个无符号整数表示的 0/1 序列作为 float 类型返回。请填写 fpower2 函数中的空白部分，以使其能正确计算结果。

```
1   float fpower2(int x)
2   {
3       unsigned exp, frac, u;
4
5       if (x<_______) {    /* 值太小，返回 0.0 */
6           exp = _____________;
7           frac = _____________;
8       } else if (x<_______) {       /* 返回非规格化结果 */
9           exp = _____________;
10          frac = _____________;
11      } else if (x<_______) {       /* 返回规格化结果 */
12          exp = _____________;
13          frac = _____________;
14      } else {        /* 值太大，返回+∞ */
15          exp = _____________;
16          frac = _____________;
17      }
```

```
18          u = exp << 23 | frac;
19          return u2f(u);
20      }
```

参考答案：

```
1       float fpower2(int x)
2       {
3           unsigned exp, frac, u;
4
5           if (x<__-149__) {     /* 值太小，返回 0.0 */
6               exp = ___0___;
7               frac = ___0___;
8           } else if (x<__-126__) {       /* 返回非规格化结果 */
9               exp = ___0___;
10              frac = ___0x400000>>(-x-127)___;
11          } else if (x<__128____) {          /* 返回规格化结果 */
12              exp = ___x+127___;
13              frac = ___0___;
14          } else {          /* 值太大，返回+∞ */
15              exp = ___255_____;
16              frac = ___0___;
17          }
18          u = exp << 23 | frac;
19          return u2f(u);
20      }
```

41. 以下是一组关于浮点数按位级进行运算的编程题目，其中用到一个数据类型 float_bits，它被定义为 unsigned int 类型。以下程序代码必须采用 IEEE 754 标准规定的运算规则，例如，舍入应采用就近舍入到偶数的方式。此外，代码中不能使用任何浮点数类型、浮点数运算和浮点常数，只能使用 float_bits 类型；不能使用任何复合数据类型，如数组、结构和联合等；可以使用无符号整数或带符号整数的数据类型、常数和运算。要求编程实现以下功能并进行正确性测试，需要针对参数 *f* 的所有 32 位组合情况进行处理。

（1）计算浮点数 *f* 的绝对值|*f*|。若 *f* 为 NaN，则返回 *f*，否则返回|*f*|。函数原型为：

```
float_bits float_abs(float_bits f);
```

（2）计算浮点数 *f* 的负数−*f*。若 *f* 为 NaN，则返回 *f*，否则返回−*f*。函数原型为：

```
float_bits float_neg(float_bits f);
```

（3）计算 0.5**f*。若 *f* 为 NaN，则返回 *f*，否则返回 0.5**f*。函数原型为：

```
float_bits float_half(float_bits f);
```

（4）计算 2.0**f*。若 *f* 为 NaN，则返回 *f*，否则返回 2.0**f*。函数原型为：

```
float_bits float_twice(float_bits f);
```

（5）将 int 型整数 *i* 的位序列转换为 float 型位序列。函数原型为：

```
float_bits float_i2f(int i);
```

（6）将浮点数 *f* 的位序列转换为 int 型位序列。若 *f* 为非规格化数，则返回值为 0；若 *f* 是 NaN 或±∞或超出 int 型数可表示范围，则返回值为 0x80000000；若 *f* 带小数部分，则考虑舍入。函数原型为：

```
int float_f2i(float_bits f);
```

参考答案：

（1）计算浮点数 f 的绝对值$|f|$。若 f 为 NaN，则返回 f，否则返回$|f|$。

```
float_bits float_abs(float_bits f) {
    unsigned sign=f>>31;
    unsigned exp=f>>23&0xFF;
    unsigned frac=f&0x7FFFFF;
    if (exp==0xFF)&&(frac!=0) || (sign==0)   /* f 为 NaN  或正数*/
        return f;
    else   /* f 为负数*/
        return f & 0x7FFFFFFF;
}
```

（2）计算浮点数 f 的负数$-f$。若 f 为 NaN，则返回 f，否则返回$-f$。

```
float_bits float_neg(float_bits f) {
    unsigned exp=f>>23&0xFF;
    unsigned frac=f&0x7FFFFF;
    if (exp==0xFF)&&(frac!=0)   /* f 为 NaN */
        return f;
    else
        return f ^ 0x80000000;
}
```

（3）计算 0.5*f。若 f 为 NaN，则返回 f，否则返回 0.5*f。

```
float_bits float_half(float_bits f) {
    unsigned sign=f>>31;
    unsigned exp=f>>23&0xFF;
    unsigned frac=f&0x7FFFFF;
    if (exp==0xFF)&&(frac!=0)   /* f 为 NaN */
        return f;
    else if ((exp==0)||(exp==0xFF)) && (frac==0)   /* f 为 0  或∞*/
        return f;
    else if (exp==0) && (frac!=0)   /* f 为非规格化数  */
        return sign<<31 | frac>>1;
    else {   /* f 为规格化数  */
            exp=exp+0xFF;
            if (exp!=0)   /* 0.5*f 为规格化数*/
                return   sign<<31 | exp << 23 | frac;
            else   /* 0.5*f 为非规格化数*/
                return sign<<31 | (frac | 0x800000)>>1;
    }
}
```

（4）计算 2.0*f。若 f 为 NaN，则返回 f，否则返回 2.0*f。

```
float_bits float_twice(float_bits f) {
    unsigned sign=f>>31;
    unsigned exp=f>>23&0xFF;
    unsigned frac=f&0x7FFFFF;
    if (exp==0xFF)&&(frac!=0)   /* f 为 NaN */
        return f;
    else if ((exp==0)||(exp==0xFF)) && (frac==0)   /* f 为 0  或∞*/
        return f;
    else if (exp==0) && (frac!=0)   {   /* f 为非规格化数  */
            if (frac&0x400000)   /* f 的尾数第一位为 1 */
                return sign<<31 | 1<<23 | (frac&0x3FFFFF)<<1;
            else   /* f 的尾数第一位为 0 */
```

```
                return sign<<31 | frac<<1;
        }
    else {   /* f 为规格化数  */
        exp=exp+0x01;
        if (exp!=0xFF)   /* 2.0*f 为规格化数  */
            return   sign<<31 | exp << 23 | frac;
        else   /* 2.0*f 发生阶码溢出*/
            return sign<<31 | exp<<23 ;
    }
}
```

（5）将 int 型整数 *i* 的位序列转换为 float 型位序列。

```
float_bits float_i2f(int i) {
    unsigned pre_count=30;
    unsigned pos_count=31;
    unsigned sign=(unsigned) i >>31;
    unsigned neg_i;
    if (i==0)   /* i 为 0 */
        return i;
    if (sign==0)   {   /* i 为正数  */
        while (i>>pre_count==0)   pre_count--;
        return sign<<31 | (127+pre_count) << 23 | (unsigned) (i<< (32-pre_count))>>23;
    }
    else   {   /* i 为负数  */
        while (i<<pos_count==0)   pos_count--;
        neg_i=(~(i>>(32-pos_count)) << (32-pos_count)) | (1<< (31-pos_count));
        while (neg_i>>pre_count==0)   pre_count--;
        return sign<<31 | (127+pre_count) << 23 | neg_i<< (32-pre_count) >>23;
    }
}
```

（6）将浮点数 *f* 的位序列转换为 int 型位序列。若 *f* 为非规格化数，则返回值为 0；若 *f* 是 NaN 或±∞或超出 int 型数范围，则返回值为 0x80000000；若 *f* 带小数部分，则考虑舍入。

```
int float_f2i(float_bits f)   {
    unsigned sign=f>>31;
    unsigned exp=f>>23&0xFF;
    unsigned frac=f&0x7FFFFF;
    unsigned exp_value = exp -127;
    unsigned neg_i;
    unsigned pos_count=31;
    if ((exp==0xFF) || (exp_value>30))   /* f 为 NaN 或∞或值太大  */
        return 0x80000000;
    else if ((exp==0) || (exp_value <0))   /* f 为非规格化数或 0 或值太小  */
        return 0;
    else if (sign==0)     /* f 为正的规格化数  */
        return   (1<<30 | frac<<7) >> (30-exp_value);
    else {     /* f 为负的规格化数  */
        neg_i = (1<<30 | frac<<7) >> (30-exp_value);
        while (neg_i<<pos_count==0)   pos_count--;
        return (~(neg_i>>(32-pos_count)) << (32-pos_count)) | (1<< (31-pos_count));
    }
}
```