8. 设 A_4~A_1 和 B_4~B_1 分别是 4 位加法器的两组输入，C_0 为低位来的进位。当加法器分别采用串行进位和先行进位（即并行进位）时，写出 4 个进位 C_4~C_1 的逻辑表达式。

【分析解答】

串行进位：

$C_1=A_1C_0+B_1C_0+A_1B_1$

$C_2=A_2C_1+B_2C_1+A_2B_2$

$C_3=A_3C_2+B_3C_2+A_3B_3$

$C_4=A_4C_3+B_4C_3+A_4B_4$

并行进位：

$C_1=A_1B_1+(A_1+B_1)C_0$

$C_2=A_2B_2+(A_2+B_2)A_1B_1+(A_2+B_2)(A_1+B_1)C_0$

$C_3=A_3B_3+(A_3+B_3)A_2B_2+(A_3+B_3)(A_2+B_2)A_1B_1+(A_3+B_3)(A_2+B_2)(A_1+B_1)C_0$

$C_4=A_4B_4+(A_4+B_4)A_3B_3+(A_4+B_4)(A_3+B_3)A_2B_2+(A_4+B_4)(A_3+B_3)(A_2+B_2)A_1B_1$
$+(A_4+B_4)(A_3+B_3)(A_2+B_2)(A_1+B_1)C_0$

10. 已知 $x = 10$，$y = -6$，采用 6 位机器数表示。请按如下要求计算，并把结果还原成真值。

（1） 求$[x + y]_{补}$，$[x-y]_{补}$。

（2） 用原码一位乘法计算$[x\times y]_{原}$。

（3） 用布斯乘法计算$[x\times y]_{补}$。

（4） 用不恢复余数法计算$[x/y]_{原}$的商和余数。

【分析解答】

先将 x 和 y 转换为二进制数。x=10=+01010B，y = –6 = –00110B。

（1）$[x]_{补}$=0 01010 B，$[y]_{补}$=1 11010 B，$[-y]_{补}$=0 00110 B。

$[x+y]_{补}$= $[x]_{补}$+$[y]_{补}$=0 01010 B + 1 11010 B =0 00100 B，因此，x+y=4。

$[x-y]_{补}$= $[x]_{补}$+$[-y]_{补}$=0 01010 B + 0 00110 B = 0 10000 B，因此，x–y=+16。

（2）$[x]_{原}$ = 0 01010 B，$[y]_{原}$=1 00110 B。

将符号和数值部分分开处理。乘积的符号为 $0\oplus1=1$，数值部分采用无符号整数乘法算法计算 01010×00110 的乘积。原码一位乘法过程描述如下：初始部分积为 0，在乘积寄存器前增加一个进位位。每次循环首先根据乘数寄存器中最低位决定+X 还是+0，然后将得到的新进位、新部分积和乘数寄存器中的部分乘数一起逻辑右移一位。共循环 5 次，最终得到一个 10 位无

符号数表示的乘积 00001 11100 B。所以，$[x\times y]_{原}$=1 00001 11100 B，因此，x×y= –60。若结果取 6 位原码，则因为高 5 位 00001 是一个非 0 数，所以，结果溢出，即$[x\times y]_{原}\neq$1 11100。验证：6 位原码的表示范围为−31～+31，显然乘积–60 不在其范围内，结果应该溢出。（过程略）

（3）$[x]_{补}$=0 01010 B，$[-x]_{补}$=1 10110 B，　$[y]_{补}$=1 11010 B。

采用 Booth 算法时，符号和数值部分一起参加运算，最初在乘数后面添 0，初始部分积为 0。每次循环先根据乘积寄存器中最低两位决定执行+X、–X 还是+0 操作，然后将得到的新的部分积和乘数寄存器中的部分乘数一起算术右移一位。–X 用+$[-x]_{补}$实现。共循环 6 次。最终得到一个 12 位补码表示的乘积 111111 000100 B，所以，$[x\times y]_{补}$=111111 000100 B，因此，x×y= –60。若结果取 6 位补码，则根据乘积低 6 位 000100 的符号位为 0，而高 6 位为 111111，不等于全 0，说明结果溢出，即$[x\times y]_{补}\neq$0 00100。验证：6 位补码的表示范围为−32～+31，显然乘积–60 不在其范围内，结果应该溢出。（过程略）

（4）$[x]_{原}$ =0 01010 B，$[y]_{原}$=1 00110 B。

将符号和数值部分分开处理。商的符号为 0⊕1=1，数值部分采用无符号整数除法算法计算 01010 B 和 00110 B 的商和余数。无符号整数不恢复余数除法过程描述如下：初始中间余数为 0 00000 01010 0，其中，最高位为添加的符号位，用于判断余数是否大于等于 0；最后一位 0 为第一次上的商，该位商只是用于判断结果是否溢出，不包含在最终的商中。因为结果肯定不溢出，所以该位商可以直接上 0，并先做一次–Y 操作得到第一次中间余数，然后进入循环。每次循环首先将中间余数和商一起左移一位，然后根据上一次上的商（或余数的符号）决定执行+Y 还是–Y 操作，以得到新的中间余数，最后根据中间余数的符号确定上商为 0 还是 1。–Y 采用+$[-y]_{补}$的方式进行。整个循环内执行的要点是“正、1、减；负、0、加”。共循环 5 次。最终得到一个 6 位无符号数表示的商 0 00001 和余数 00100，其中第一位商 0 必须去掉，添上符号位后得到最终的商的原码表示为 1 00001，余数的原码表示为 0 0100。因此，x/y 的商为–1，余数为 4。（过程略）

11. 若一次加法需要 1ns，一次移位需要 0.5ns。请分别计算用一位乘法、两位乘法计算两个 8 位无符号二进制数乘积时所需的时间。

【分析解答】

对于 8 位无符号二进制数相乘，一位乘法需 8 次右移，8 次加法，共计 12ns；二位乘法需 4 次右移，4 次加法，共计 6ns。

16. 对于 3.4.1 节中例 3.8 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。

【分析解答】

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 }
```

17. 假设一次整数加法、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。

【分析解答】

根据表达式 $55*x=(64-8-1)*x=64*x-8*x-x$ 可知，完成 $55*x$ 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。

21. 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。

【分析解答】

不能精确用 IEEE 754 单精度和双精度格式表示的最小正整数分别是 $2^{24}+1$、$2^{53}+1$。

（不超过 24 位的正整数都可以用 float 类型精确表示。

$2^{24}+1=1.0000\ 0000\ 0000\ 0000\ 0000\ 0001\times 2^{24}$）

22. 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？

【分析解答】

IEEE 754 浮点数加、减、乘、除运算结果的尾数不可能大于等于 4，因而尾数溢出的情况只可能是±1x .xx…x 的形式。

(1) 对于结果为±1x .xx…x 的情况，需进行右规。即尾数右移一位，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：a) 尾数右移时，最高位“1”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。b) 阶码加 1 前，先判断阶码是否为全 1 或 1…10，若是，则发生阶码上溢导致结果溢出；否则，直接在阶码末位加 1。

(2) 对于结果为±0.0…01x…x 的情况，需进行左规。即数值位逐次左移，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 $M_b \leftarrow M_b \times 2^{k}$，$E_b \leftarrow E_b - k$。

左规时注意以下几点：a) 尾数左移时数值部分最左 k 个 0 被移出，因此，相对来说，小数点右移了 k 位。b) 每次减 1 前，先判断阶码是否为全 0，若是全 0，则发生阶码下溢，不再进行左规操作，结果为非规格化数（尾数不为 0 时）或 0（尾数为 0 时）；否则，通过执行+11…1 进行减 1 操作。c) 减 1 操作最多 k 次。

25. 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）

【分析解答】

$x=0.75=0.11B=(1.10\cdots0)_2\times2^{-1}$，$y=-65.25=-1000001.01B=(-1.00000101\cdots0)_2\times2^6$。

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。假定用 E_x、E_y 分别表示$[x]_浮$和$[y]_浮$中的阶码，M_x、M_y 分别表示$[x]_浮$和$[y]_浮$中的尾数，即 E_x=0111 1110，M_x=0(1).10…0，E_y=1000 0101，M_y=1(1).000001010…0。尾数 M_x 和 M_y 的小数点前面有两位，第一位为数符，第二位加了括号，是隐藏位“1”。以下是机器中浮点数加/减运算过程（假定保留两位附加位：保护位和舍入位）。

① 对阶。

$[\Delta E]_补=E_x+[-E_y]_补$=0111 1110 + 0111 1011=1111 1001 (mod 2^8)，$\Delta E=-7$，故需对 x 进行对阶，结果为 $E_x=E_y$=1000 0101，M_x=00.00000**1**10…0 **00**，即将 x 的尾数 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，最后移出的高两位保留。

② 尾数相加。

$M_b=M_x+M_y$=00. 00000**1**10…0 **00**+11. 000001010…0 **00** （注意小数点在隐藏位后）。根据原码加/减运算规则（加法运算规则为“同号求和，异号求差”，最后结果可能需要求补），得：00.000000**1**10…0 **00**+11.000001010…0 **00**=11.000000100…0 **00**。上式尾数中最左边第一位是符号位，其余都是数值部分，尾数最后两位是附加位。（0.00000011+0.11111011=0.11111110，没有进位，需求补，求补后为 1.00000010）

③ 规格化。

根据所得尾数的形式，数值部分最高位为 1，所以不需要进行规格化。

④ 舍入。

将结果的尾数 M_b 中最后两位附加位舍入，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。

最终结果为 E_b=1000 0101，M_b=1(1).00000010…0，即结果为$(-1.0000001)_2\times2^6=-64.5$。