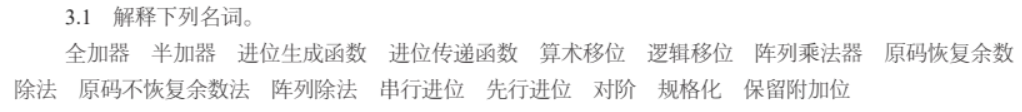
**作业讲解三**

**庾晓萍（20420192201952）**

**3.1**

****

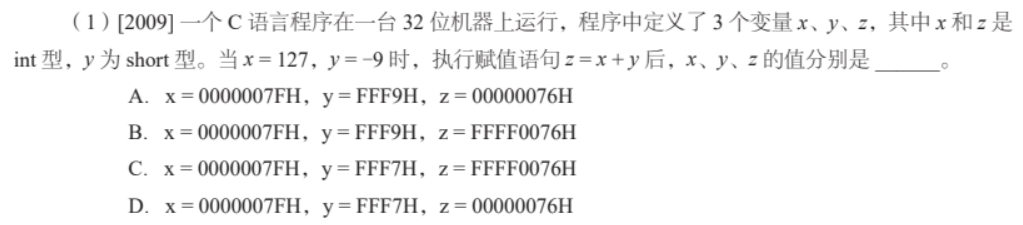
**（1）全加器：设计的带进位的一位加法器，又称全加器。全加器包括3个输入和两个删输出，输入端分别为相加数Xi、Yi，低位进位输入Ci。输出端分别是和数Si、高位进位输出Ci+1。**

**（2）半加器：半加器没有进位输入，所以其内部逻辑只有一个异或门，用于产生和数，一个与门，用于产生进位输出，半加器通常用于没有进位输入的情况，以减少不必要的开销。**

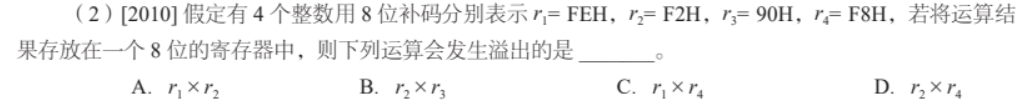
**进位生成函数：**

1. **进位生成函数：n位串行加法电路中和数与进位输出的逻辑表达式如下：和数Si=Xi⊕Yi⊕Ci 进位输出Ci+1=XiYi+(Xi⊕Yi)Ci。设Gi=XiYi，Pi=(Xi⊕Yi)，当Gi=1时（Xi与Yi都为1），Ci+1一定为1，所以将Gi称为进位生成函数。**
2. **进位传递函数：n位串行加法电路中和数与进位输出的逻辑表达式如下：和数Si=Xi⊕Yi⊕Ci 进位输出Ci+1=XiYi+(Xi⊕Yi)Ci。设Gi=XiYi，Pi=(Xi⊕Yi)，当Pi=1时，也就是Xi与Yi相异时，进位输入信号才能传递到进位输出Ci+1处，所以将Pi称为进位传递函数。**
3. **算术移位：算术是带有符号的数据，所以我们不能直接移动所有的位数，这可能会使得符号不正确。算术移位时，正数无论左移还是右移都是补0。而负数的补码左移在右边补0，右移需要在左边补1。**
4. **逻辑移位：对于逻辑移位，就是不考虑符号位，移位的结果只是数据所有的位数进行移位。根据移位操作的目的，左移时，低位补0，右移时，高位补0。**
5. **阵列乘法器：为了提高多个位积求和的速度，采用硬件的方法实现阵列乘法器，基本思想是采用类似手动乘法运算的方法，用大量与门阵列同时产生手动乘法中的各乘积项，同时将大量一位全加器按照手动乘法运算的需要构成全加器阵列。**
6. **原码恢复余数除法：在原码恢复余数法中，比较被除数（余数）与除数的大小是通过减法实现的。对原码除法而言，操作数以绝对值的形式参与运算。因此相减的结果为正（符号位为0）说明够减，商上1；相减结果为负（符号位为1）说明不够减，商上0。由于除数通过减法实现，当商上1的时候，减法得到的差值就是余数，可以继续进行后续的除法操作。但商上0时表示不够减，减法得到的余数是负数，因此就需要将余数加上除数，即将余数恢复成比较操作之前的数值，这种方法就称为恢复余数法。**
7. **原码不恢复余数法：不恢复余数法是对恢复余数法的改进，主要特点是不够减时不需要恢复余数，而根据余数符号不同进行不同的运算处理。当余数为正时，商上1，余数左移一位，减去除数。当余数为负时，商上0，余数左移一位，加上除数。**
8. **阵列除法：为了加快除法运算速度，也可以采用阵列除法器来实现除法运算。为简化运算及阵列除法器的结构，应对参与运算的数据进行适当的处理，使其以正数的形式参加运算。**
9. **串行进位又称波形进位，每级进位直接依赖于前一级的进位，即进位信号是逐级形成的。**
10. **先行进位：引入生成和传递进位的两个进位辅助函数，使得加法器的各个进位之间相互独立并行产生。**
11. **对阶：对阶的原则是小的阶码向大的阶码看齐，这是因为小阶码数值增大时，尾数部分会右移，舍去的是尾数的低位部分，只有很小的精度影响。如果让大阶码向小阶码看齐，则尾数部分需要左移，将会丢失尾数的高位部分，会严重影响运算精度和结果的正确性。对阶又包括求阶差、阶码的调整和尾数的移位两部分。**
12. **规格化：规格化就是使运算结果成为规格化数。为了方便处理，可让尾数的符号位扩展为双符号位，当尾数运算结果不是11.0……或00.1的形式时，应进行相应的规格化处理。当尾数符号位为01或10时，运算结果上溢，需要向右规格化，且只需将尾数右移一位，同时将结果的阶码值减一。当尾数的运算结果为11.1……或00.0……时需要向左规格化，而且左移次数不固定，与运算结果的形式有关。向左规划化时尾数连同符号位一起左移，直到尾数部分出现11.0……或00.1的形式为止。向左规格化时阶码作减法，左移多少位就减多少。**
13. **保留附加位：尾数右移时通常将最低位的移出位暂时保留，称为保留附加位。保留附加位参与中间运算以提高运算精度。尾数运算结束，结果规格化再进行舍入。**

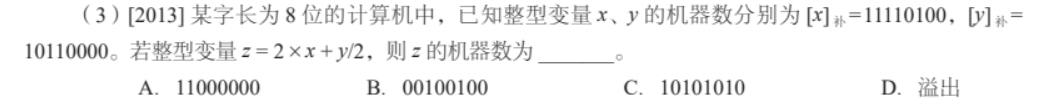
**3.2**

****

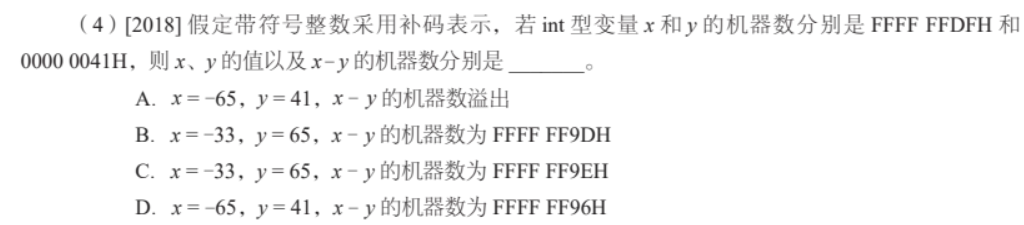
**D：32位机器：int 32位，short 16位。x = 127，原码等于补码：0111 1111。扩展到32位高位补0，结果为0000007FH。Y = -9，负数原码：1000 1001，补码：1111 0111，扩展到16位高位补1，结果为FFF7H。z = x + y = 118，原码：0111 0110，补码：0111 0110，扩展到32位高位补0，结果为00000076H。**

****

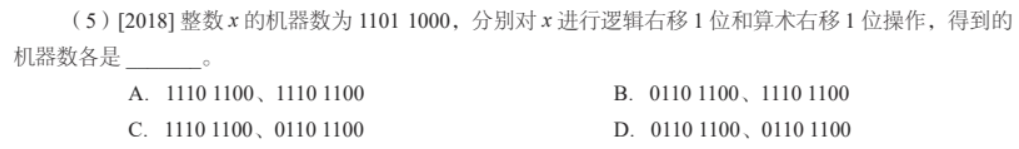
**B：整数用8位补码表示范围在-128 ~ 127，其中r3（10010000）的原码为1111 0000即-112，r2（1111 0010）的原码为1000 1110即-14，r2\*r3>127，故B会发生溢出。**



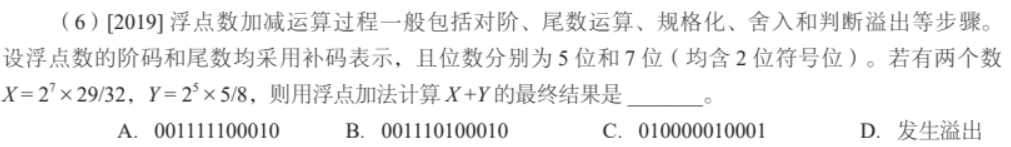
**A：x的原码是1000 1100即-12，y的原码是1101 0000即-80，求得z为-64（1100 0000），补码也为1100 0000。**



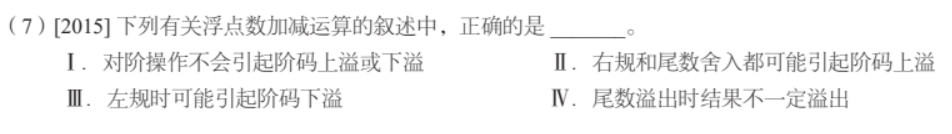
**C：x的补码为1111 1111 1111 1111 1111 1111 1101 1111（原码：1000 0000 0000 0000 0000 0000 0010 0001即-33）y的补码是0000 0000 0000 0000 0000 0000 0100 0001（原码相同，即65）相减为-98（机器数：1111 1111 1111 1111 1111 1111 1001 1110，FFFF FF9E H）**



**B：算数右移：指的是将原来的数向右移X位，前面补符号位。逻辑右移：指的是不考虑符号位，左边直接补0**



**D：X=00111 00.11101，Y=00101 00.10100。对阶，Y=00111 00.00101，尾数求和，得到01.00010。此时需要进行规格化，00111 01.00010= 01000 00.10001，不需要舍入。此时由于阶码符号位出现01，即阶码大于最大阶码，表示发生上溢。**

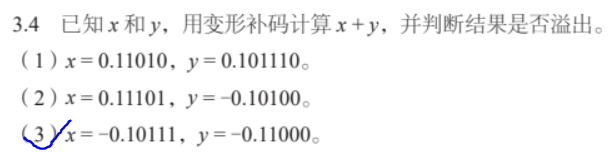


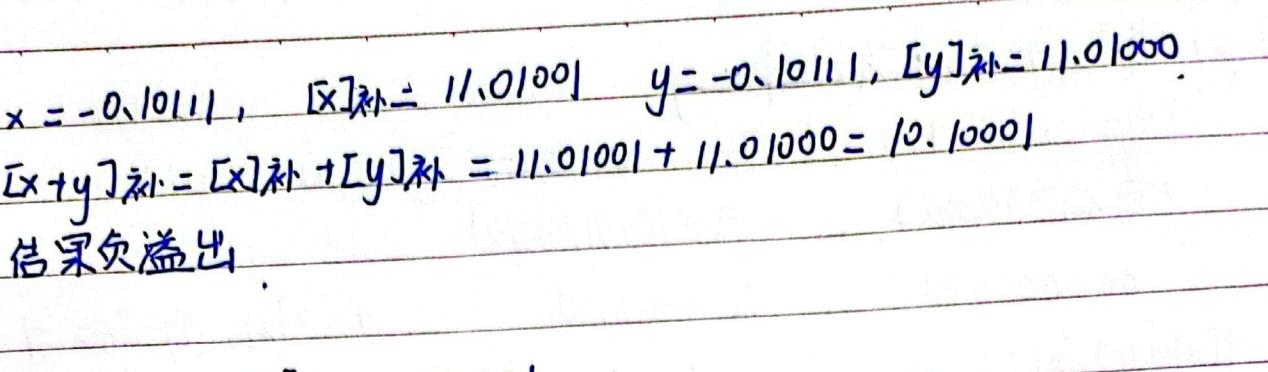


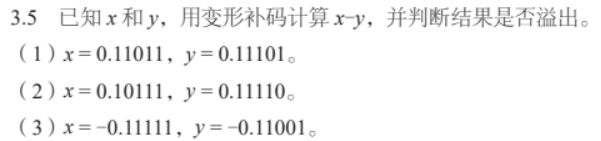
**D:**

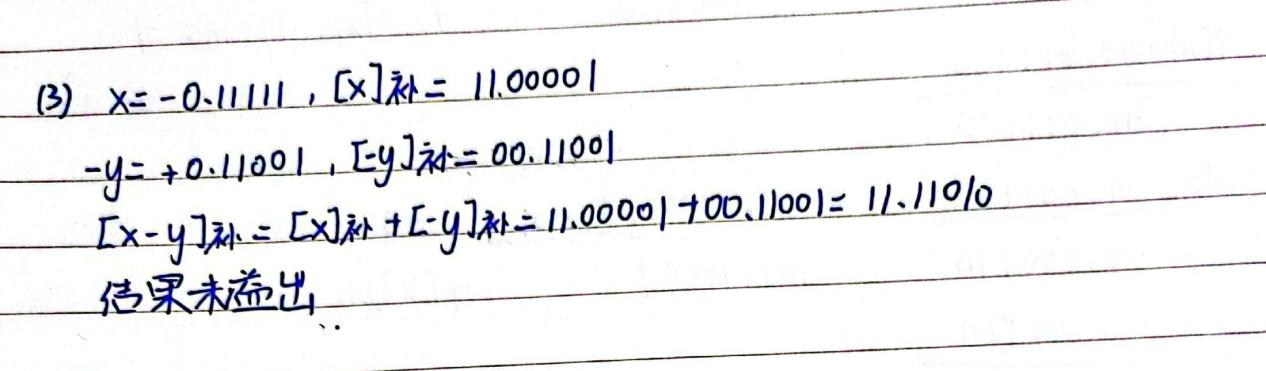
**Ⅰ正确。对阶时，统一保留大的阶数，并不会造成阶码的上溢或者下溢。Ⅱ正确。右规和尾数舍入的过程会造成阶码的增加，因而有可能会引起阶码上溢。Ⅲ正确，左规的过程会造成阶码的减少，因而有可能会引起阶码下溢。 Ⅳ正确，尾数溢出时，阶码不一定会溢出，结果也不一定能溢出。**

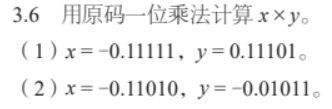
**3.4**

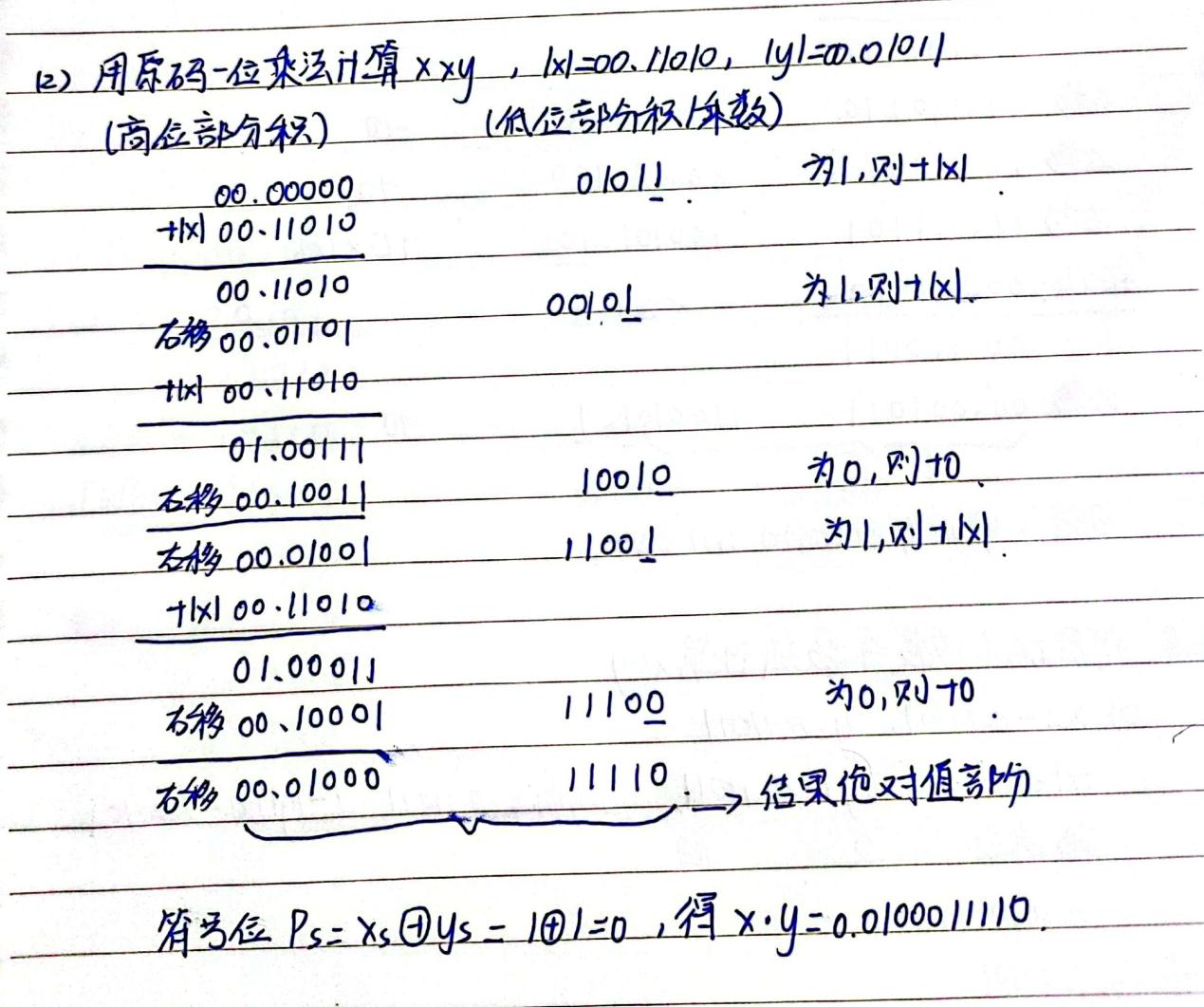


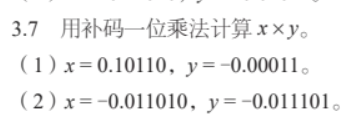


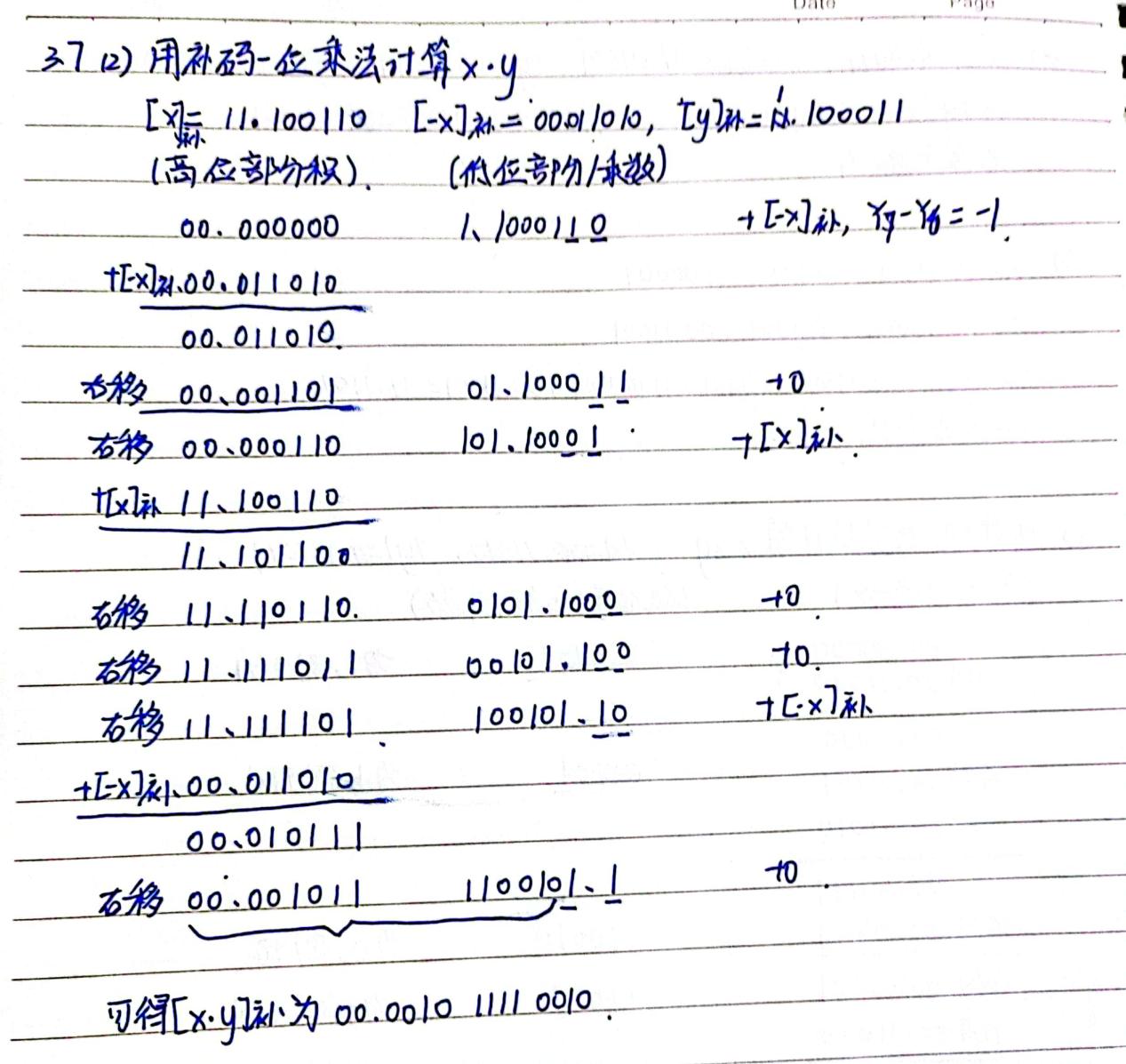


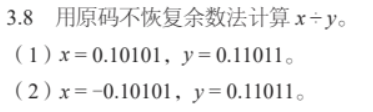


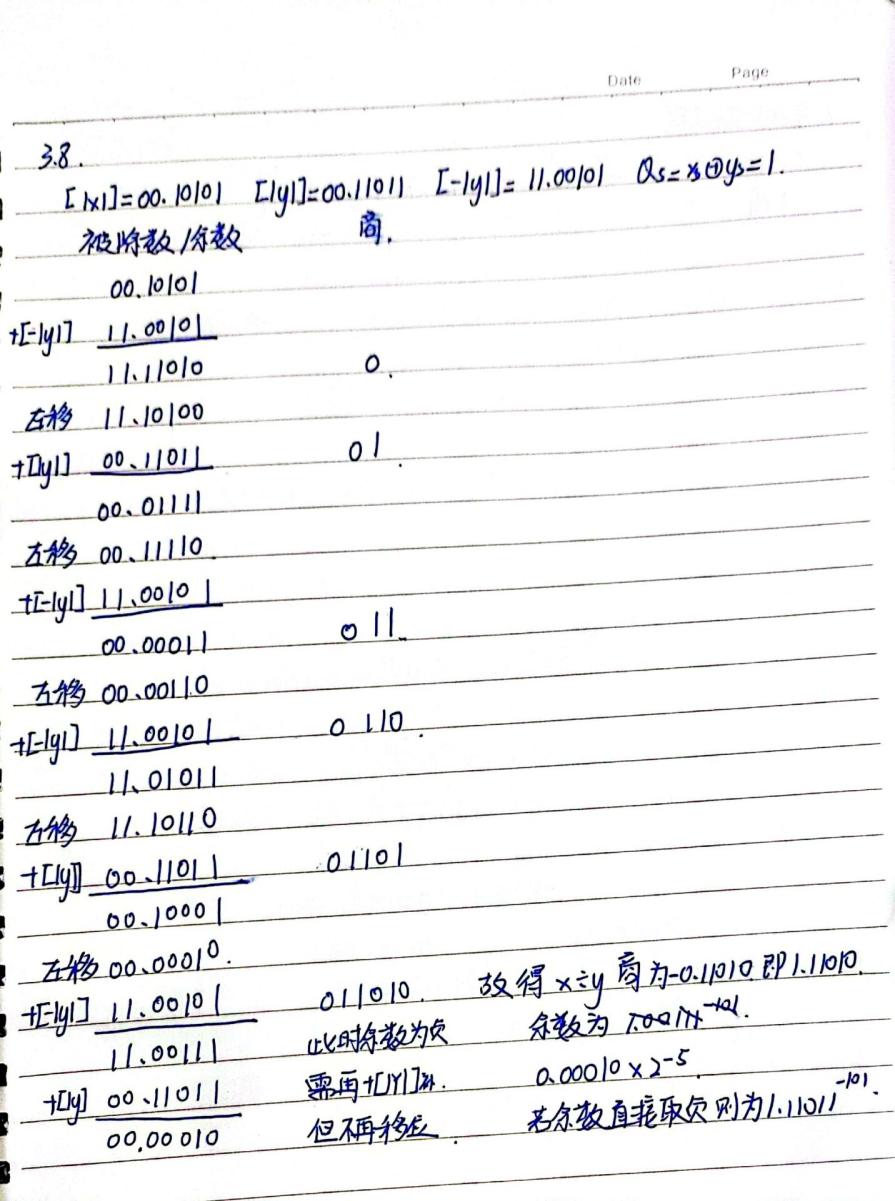


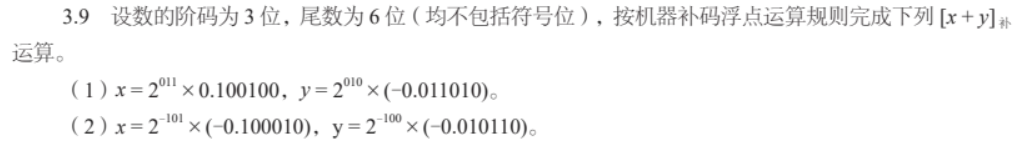


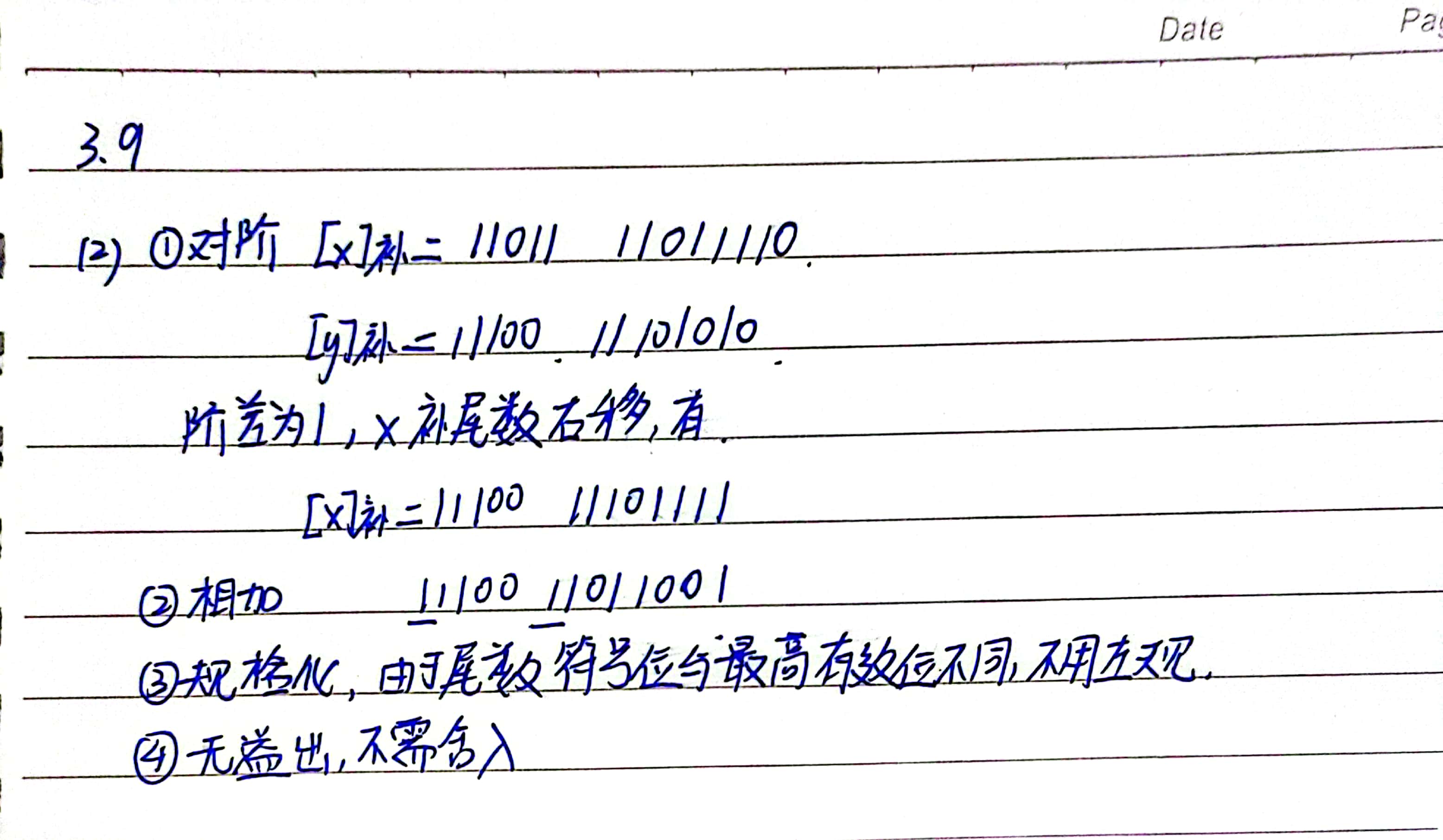


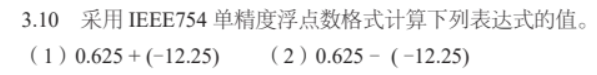


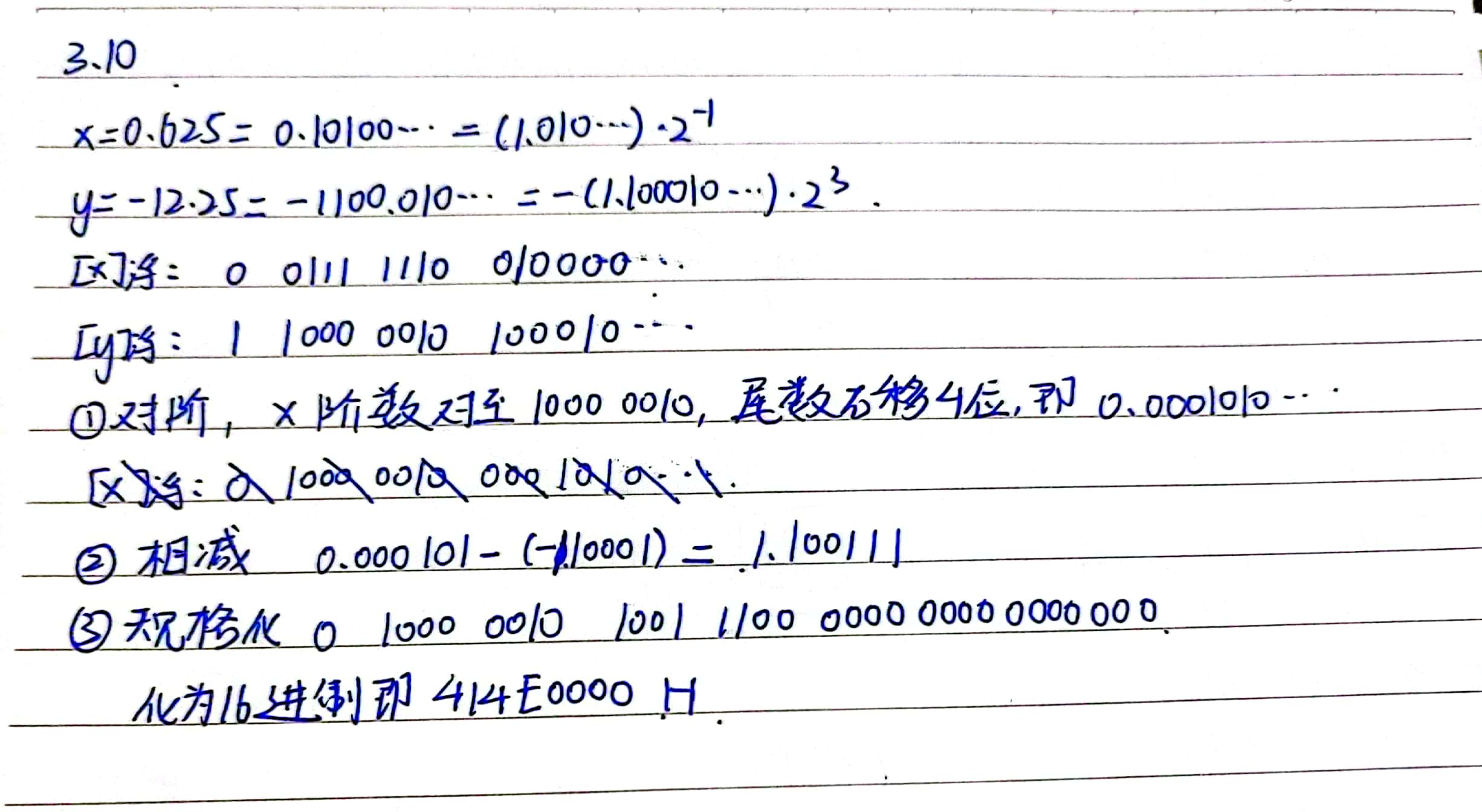




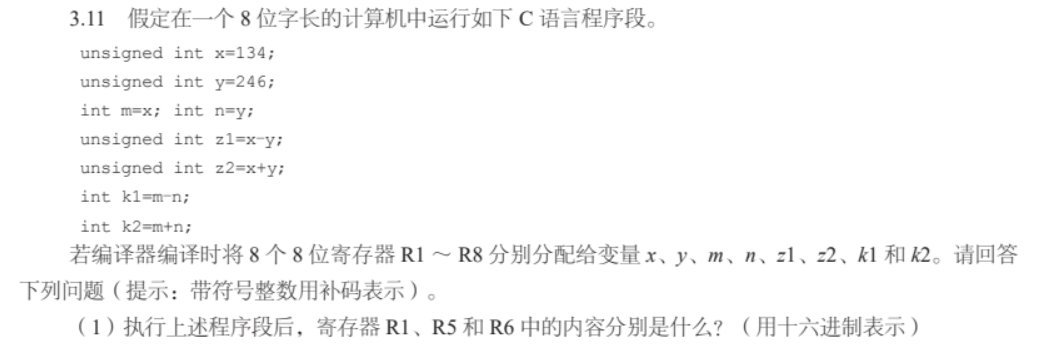


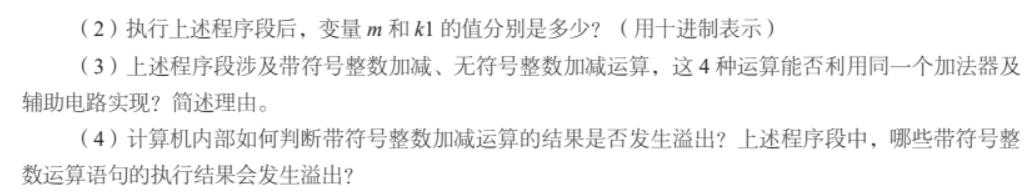






**3.11**





（1）R1存储的是x=134，转换成二进制为1000 0110，即86 H。R5存储的是z1，即x－y的内容，x和y都是无符号整型，两数相减的过程是按照正常的相减方式的。x－y=-112，转换成二进制为1001 0000（补码），即90H。R6存储的是x+y的内容，x+y=380，转换成二进制为（1）0111 1100（前面的进位舍弃），即7C H。由于计算机字长为8位，所以无符号整数能表示的范围为0~255。而x+y=380，故溢出。

（2）m二进制表示为1000 0110，由于m是int型，所以最高位为符号位，所以可以得出m的原码为：1111 1010，即-122。同理n的二进制表示为1111 0110，原码为：1000 1010，转成十进制为-10。所以k1=-122-（-10）=-112.

（3）可以利用同一个加法器及辅助电路实现。因为都是以补码形式存储，所以运算规则都是一样的。但是有一点需要考虑，由于无符号整数和有符号整数的表示范围是不一样的，所以需要设置不一样的溢出电路。

（4）①符号相同操作数的运算后，双符号位、最高位进位与原操作数的符号不同则溢出。②带符号整数只有k2会发生溢出。8位带符号整数的补码取值范围为：-128~+127，而k2=m+n=-122-10=-132，超出范围。