1.完成《计算机科学导论》练习题 2.3.1，2.3.3，2.3.4，2.3.5，2.3.6

2.3.1

（1）100100010_2

（2）10001111_2

（3）11000001001_2

2.3.3

-16 补码：11110000

2.3.4

-124 补码：10000100

2.3.5

$127-3 = 01111100_2$

2.3.6

$(-4)-4 = 11111000_2$

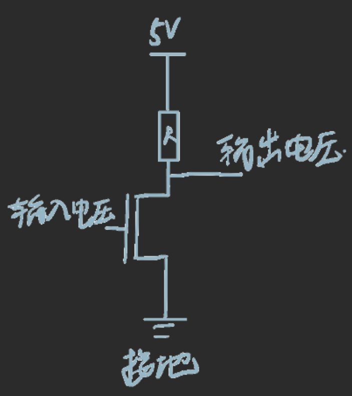
2. 完成《计算机科学导论》练习题 2.4.2，2.4.3，2.4.4，2.4.6

2.4.2

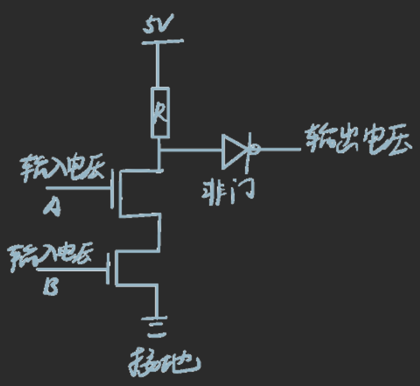
有与（AND），或（OR），非（NOT）三种。真值表如下：

A	B	$\overline{A}$	AND	OR
0	0	1	0	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	1

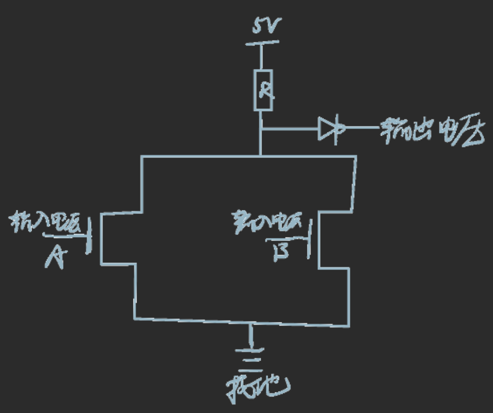
电路示意图如下



非门



与门



或门

2.4.3

8 种；2^n 种

2.4.4

（1）S=1

（2）S=1

（3）S=0

2.4.6

借用 C 语言中的位运算符来表示

0110 = ~1001

1111 = 1001 | 1111

3.完成《计算机科学导论》练习题 2.5.1，2.5.2，2.5.3，2.5.4

2.5.1

1KB = 1024B = 8 * 1024 b=8192b

2.5.2

1GB = 1024KB

5GB = 5120KB

2.5.3

假设要买的硬盘大小为 XGB，根据硬盘厂商的 10 进制和计算机的 2 进制定义，有：

$$X \cdot 10^9 = 500 \cdot 2^{30}$$

计算出至少要买 536.870912 GB 的硬盘（应该没有厂商会生产这种容量的硬盘吧……）

2.5.4

1TB 的数量级约为 10^{12}；1EB 的数量级约为 10^{18}

*4. 请问如何快速检查一个四进制数是 5 的倍数？例如：121,123321,102031,132231112332 都是 5 的倍数。请证明你的方法。请用 **Python** 实现你的思路并分析其时间复杂度。*

类比我们对十进制数是否能被 5 整除的研究，我们可以有以下定理判断一个四进制数能否被 5 整除。

对于一个整数$n = (\overline{a_n a_{n-1} \dots a_1 a_0})_4$，如果满足：

$$\sum_{i=0}^{n} (-1)^i \cdot a_i = 0$$

那么这个四进制数能够被 5 整除。

证明：

考虑其另一种表现形式

$$n = 4^n \cdot a_n + 4^{n-1} \cdot a_{n-1} + \cdots + 4^1 \cdot a_1 + 4^0 \cdot a_0$$

将 n 对 5 取模，由取模运算的性质，可以有

$$n \equiv (-1)^n \cdot a_n + (-1)^{n-1} \cdot a_{n-1} + \cdots + (-1)^1 \cdot a_1 + (-1)^0 \cdot a_0 (mod 5)$$

当$n \equiv 0(mod5)$时，说明 n 能够被 5 整除。

证毕。

根据这个定理，代码如下：

```
def isDiviByFive(x):
    x = list(map(int, list(x)))
    cri = 0
    for i in range(-1, -len(x) - 1, -1):
        if (i * (-1) - 1) % 2 == 0:
            cri -= x[i]
        else:
            cri += x[i]
    if cri == 0:
        ret = True
    else:
        ret = False
    return ret


num = input()
print(isDiviByFive(num))
```

Shell:

Case0:

```
121
True

Process finished with exit code 0
```

Case1:

```
123321
True

Process finished with exit code 0
```

Case2:

```
1020
False

Process finished with exit code 0
```

Case3:

```
10210102003
False

Process finished with exit code 0
```

可以看出，对于任何 n 位数，这个算法的时间复杂度为$O(n)$

5.

a.完成《计算机科学导论》习题 2.10。请用以下几组样例测试你的程序:

x=128,y=8;　　x=129,y=8;　　x=1,y=12;　　x=2047,y=12.

b.类似《计算机科学导论》习题 2.7，但是输入 x 和 y 是介于-128 到 127 间的任意整数。转为二进制后，做加法。如果有溢出，需要返回错误，否则返回十进制的结果。此题的重点是负数补码，二进制加法和判断溢出。判断溢出必须用书上的方法：在二进制的结果上直接判断。请用以下几组样例测试你的程序:

64+65　　100+10　　100+(-128)　　10+(-100)　　(-30)+(-100)　　(-127)+127

a.

```
x, y = map(int, input().split())
Bin = []
digit = 0
while digit < y:
    if x % 2 == 0:
        Bin = [0] + Bin
    else:
        Bin = [1] + Bin
    x //= 2
    digit += 1

if x > 0 or (Bin[0] == 1 and 1 in Bin[1:]):
    print("False", end='')
else:
    for i in range(len(Bin)):
        if Bin[i] == 0:
            Bin[i] = 1
        else:
            Bin[i] = 0
    Bin[-1] += 1
    for i in range(-1, -len(Bin) - 1, -1):
        if Bin[i] == 2:
            Bin[i] = 0
            try:
                Bin[i - 1] += 1
            except IndexError:
                pass
        else:
            break
    for i in range(len(Bin)):
        print(Bin[i], end='')
```

Case0:

```
128 8
10000000
Process finished with exit code 0
```

Case1:

```
129 8
False
Process finished with exit code 0
```

Case2:

```
1 12
111111111111
Process finished with exit code 0
```

Case3:

```
2047 12
100000000001
Process finished with exit code 0
```

b.

虽然 Python 中有许多很方便很方便的函数，但为了尽可能地像 CPU 那样运算，写了很多很基本的东西（从一个全加器开始），以下是主函数种用到的函数

```python
def full_adder(a, b, c):
    carry = (a and b) or (b and c) or (c and a)
    somme = (a and b and c) or (a and (not b) and (not c)) \
        or ((not a) and b and (not c)) or ((not a) and (not b) and c)
    return int(carry), int(somme)

def add(x, y):
    while len(x) < 8:
        x = [0] + x
    while len(y) < 8:
        y = [0] + y
    ret = []
    carry = 0
    for i in range(len(x) - 1, -1, -1):
        carry, somme = full_adder(x[i], y[i], carry)
        ret = [somme] + ret
    return ret

def complement(x):
    ret = x[:]
    for i in range(len(ret)):
        if ret[i] == 0:
            ret[i] = 1
        else:
            ret[i] = 0
    ret = add(ret, [1])
    return ret

def dec_to_bio(x):
    ret = []
    if x < 0:
        sgn = -1
        x *= -1
    else:
        sgn = 1
    for i in range(8):
```

```
        if x % 2 == 0:
            ret = [0] + ret
        else:
            ret = [1] + ret
        x //= 2
    if sgn == -1:
        ret = complement(ret)
    return ret

def bio_to_dec(x):
    ret = 0
    sgn = 1
    if x[0] == 1:
        sgn = -1
        x = complement(x)
    for i in x:
        ret = ret * 2 + i
    return sgn * ret
```

主函数部分如下:

```
if __name__ == "__main__":
    a, b = map(int, input().split())
    a = dec_to_bio(a)
    b = dec_to_bio(b)
    res = add(a, b)
    if a[0] == b[0] and res[0] != a[0]:
        print("溢出错误")
    else:
        print(bio_to_dec(res))
```

Case0:

```
64 65
溢出错误

Process finished with exit code 0
```

Case1:

```
100 10
110

Process finished with exit code 0
```

Case2:

```
100 -128
-28

Process finished with exit code 0
```

Case3:

```
10 -100
-90

Process finished with exit code 0
```

Case4:

```
-30 -100
溢出错误

Process finished with exit code 0
```

Case5:

```
-127 127
0

Process finished with exit code 0
```

6. 输入 2 个十进制的整数，你的程序将它们转为二进制后，编写二进制乘法函数，函数返回二进制值，再转为十进制输出。我们现在有正负和位数限制。假设 CPU 是 N 位(你的程序要适用给定任意 N 介于 8 到 20),用补位法表示正负值，例如 N=12,则可表示范围是-2048 到 2047 之间的任意值。输入 2 个十进制的整数，检查介于可表示范围内，就马上转为二进制表示，编写二进制乘法函数 mult(),限制在 N 位，函数返回是否溢出，如果没有溢出返回正确的结果二进制值，再转为十进制输出。

例如如给定 N=12 时：

当 x=-12，y=100,最后输出是-1200,

当 x=-30，y=-10,最后输出是 300,

当 x=4， y=-512,最后输出是-2048,

当 x=4， y=512,最后输出是"溢出错误",

当 x=40，y=-80,最后输出是"溢出错误"。

函数 Mult(a,b,N):输入 a,b 是补位后二进制（可以是字符串），N 代表最大限制位数。例如，N=12.建议如果 a,b 都是负数，将它们转为正数，再做正数相乘；假如 a 正 b 负，则 a,b 可以交换。让 b 作为正数，然后一位位看 b 的位是否 0，1。需要时加上移位后的 a,如同多个 N 位负数相加，注意移位后要检查是否溢出，只需留下 N 位部分作加法。在函数内检查是否有溢出，绝对不能转为十进制来检查，这不是 CPU 内使用的方式，在 CPU 内的运算都是二进制，所以必须在二进制中检查

代码有些长，但是为了面面俱到，再加上写了很基本的东西，所以长是很必然的。

而且为了尝试用对象（虽然感觉没有这个必要），弄得 Mult 函数反而没有一个乘法器对象重要了。

简单说一下代码的过程。首先是定义了一个 Multiplier 对象，采用的乘法计算方式是将输入数转化为正数，符号按照输入的情况保留在另一个变量中。之后，mult 函数里创建了一个 multiplier 对象创建对象时，其__init__()方法中将会调用 dec_to_bio()方法，这时将会检查第一种溢出：输入数溢出。然后，用 Multiplier.multiplier 方法，利用全加器组成的加法器运算出乘法结果，期间一直检查是否溢出，之后离开这个方法前，补充检查特殊情况(-2048)，检查完之后将其转换为 10 进制。然后，在函数中检查是否两种溢出都没有发生，从而输出。

```python
class Multiplier:
    def __init__(self, x, y, n):
        self.isOverflow = False
        self.NumOverflow = False
        self.isMultiplying = False
        self.digit = n
        self.x, self.sgn_x = self.dec_to_bio(x)
        self.y, self.sgn_y = self.dec_to_bio(y)
        # Here we use unsigned integers, and the sign is saved in another variable.
        # Using unsigned can simplify the code
        self.produit = [0, ]
        self.carry = 0
        self.ret = 0
        if self.sgn_x == self.sgn_y:
            self.sgn = 1
        else:
            self.sgn = -1

    def full_adder(self, a, b, c):
        carry = (a and b) or (b and c) or (c and a)
        somme = (a and b and c) or (a and (not b) and (not c)) \
                or ((not a) and b and (not c)) or ((not a) and (not b) and c)
        return int(carry), int(somme)
```

```python
def add(self, x, y):
    while len(x) < len(y):
        x = [0] + x
    while len(y) < len(x):
        y = [0] + y
    ret = []
    self.carry = 0
    for i in range(len(x) - 1, -1, -1):
        self.carry, somme = self.full_adder(x[i], y[i], self.carry)
        ret = [somme] + ret
    if 1 in ret[:1 - self.digit] and self.isMultiplying:
    # If we are not in the process of multiplying and we use the add method, we may
    # do wrong judgement about the overflow.
        self.isOverflow = True
    # If overflow happened, we can find 1 out of the range of digit,
    # so we can detect whether 1 out of the range.
    return ret[- self.digit:]

def multiplier(self):
    self.isMultiplying = True
    # IN
    for i in range(-1, -self.digit - 1, -1):
        if self.y[i]:
            self.produit = self.add(self.produit, self.x)
            if self.carry:
                self.produit = [self.carry] + self.produit
        self.x += [0]
    self.isMultiplying = False
    # OUT
    if self.sgn == -1 and self.produit[0] == 1 and (1 not in self.produit[1:]):
        self.isOverflow = False
    if self.sgn == -1:
        self.produit = self.complement(self.produit)
    self.ret = self.bio_to_dec(self.produit)

def complement(self, x):
    ret = x[:]
    for i in range(len(ret)):
        if ret[i] == 0:
            ret[i] = 1
        else:
            ret[i] = 0
    ret = self.add(ret, [1])
    return ret

def dec_to_bio(self, x):
    ret = []
    if x < 0:
```

```
            sgn = -1
            x *= -1
        else:
            sgn = 1
        for i in range(self.digit):
            if x % 2 == 0:
                ret = [0] + ret
            else:
                ret = [1] + ret
            x //= 2
        if x > 0:
            self.NumOverflow = True
        return ret, sgn

    def bio_to_dec(self, x):
        ret = 0
        sgn = 1
        if x[0] == 1:
            sgn = -1
            x = self.complement(x)
        for i in x:
            ret = ret * 2 + i
        return sgn * ret


def multi(a, b, N):
    multiplier = Multiplier(a, b, N)
    multiplier.multiplier()
    if multiplier.isOverflow or multiplier.NumOverflow:
        print("溢出错误")
    else:
        print(multiplier.ret)


if __name__ == "__main__":
    a, b, N = map(int, input().split())
    multi(a, b, N)
```

Case0:

```
-12 100 112
-1200

Process finished with exit code 0
```

Case1:

```
-30 -10 12
300

Process finished with exit code 0
```

Case2:

```
4 -512 12
-2048

Process finished with exit code 0
```

Case3:

```
4 512 12
溢出错误

Process finished with exit code 0
```

Case4:

```
40 -80 12
溢出错误

Process finished with exit code 0
```

Case5:

```
100 200 20
20000

Process finished with exit code 0
```

Case6:

```
10215102507 1 20
溢出错误

Process finished with exit code 0
```

Case7:

```
3080 3090 20
溢出错误

Process finished with exit code 0
```

Case8:

```
120 2 8
溢出错误

Process finished with exit code 0
```

Case 9:

```
-34 -3 8
102

Process finished with exit code 0
```