数质与编码

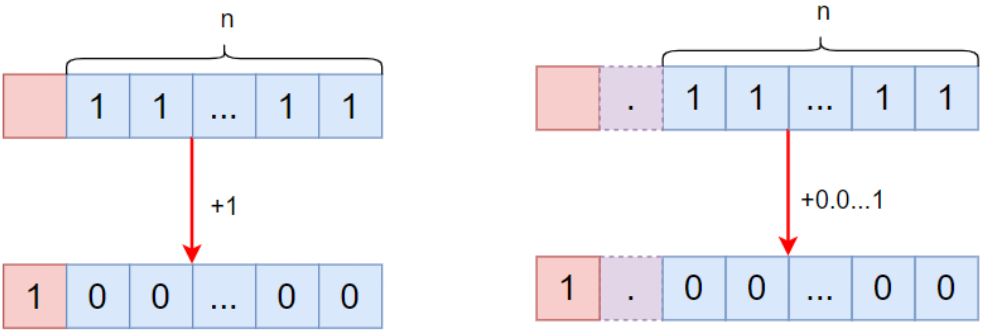
1. 罗马的数字是基于加法原理的，例如$XII=10+1+1=12$、$IX=10-1=9$。
2. 而十进制等是基于乘法原理实现的，其每一位的数字以及每一位数字所在的位置均表示权重。例如96中的就其实表示的是$9\cdot 10^1$。
3. 十进制$D(decimalism)$、十六进制$H(hexadecimal)$。
4. 对于十进制转二进制的原理进行一定程度的说明: 十进制a表示为二进制为 $K_n\ K_{n-1}\ \cdots\ K_2\ K_1\ K_0\ K_{-1}\ K_{-2}\ \cdots\ \ k_{-m+1}\ k_{-m}$
    1. 对于整数部分：$K_n*2^n+K_{n-1}*2^{n-1}+\cdots+K_1*2^1+K_0*2^0$，显然我们每次$\%2$就可以得到一个低位二进制数。
    2. 对于小数部分: $K_{-1}*2^{-1}+K_{-2}*2^{-2}+\cdots+K_{-m+1}*2^{-m+1}+K_{-m}*2^{-m}$，显然我们每次$\cdot 2$就可以得到一个高位二进制数。

BCD码

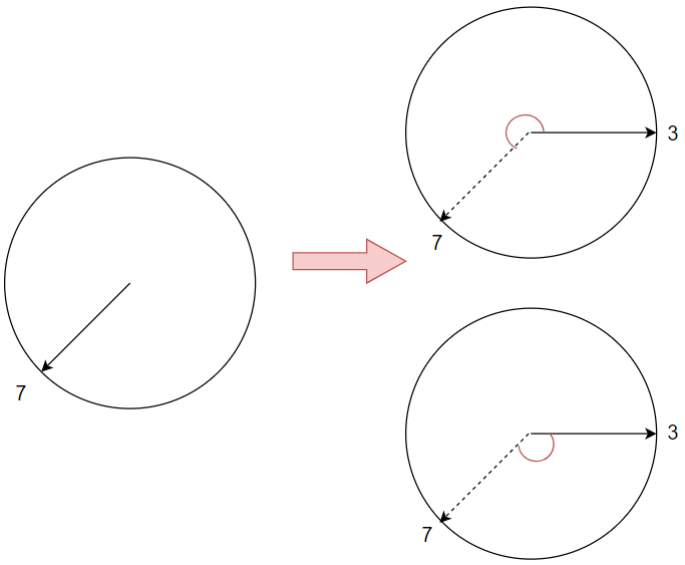
1. 实际上通过上述内容，不难知道将二进制转换为十进制还是比较麻烦的，对此我们可以参照二进制转八进制的思想，我们规定某种编码中，4位二进制表示一位十进制数(当然$4b$能表示16种数字，显然有6多余)。
2. 8421码： 在这种编码方式中每一位二值代码的1都是代表一个固定数值，把每一位的1代表的十进制数加起来，得到的结果就是它所代表的十进制数码。 由于代码中从左到右每一位的1分别表示8，4，2，1，所以把这种代码叫做8421代码。
    1. 8421码是一种有权码。
    2. $0110=2+4=6$显然0110在8421中表示十进制6。
3. 对于8421码的计算：$0101(5)+1000(8)=1101(113)$,但是$13$并不是这样表示的，正确的$13$为$00010011$,此时我们就需要在上述计算结果上进行$+0110(6)$修正(大于$1001(9)$即需要修正)，即$1101+0110=(000)10011$。至于为什么是加$6$修正，上面提过$4b$表示一个个位的十进制数存在$6$中冗余的状态，在上面计算中，我们计算的结果实际上是需要跳过这些冗余状态的。
4. 余3码：其实就是8421码$+0011$,显然是一种无权码。
5. 2421码：有权码，对于十进制5而言，实际上存在多种表示方式：$1011=2+2+1$、$0101=4+1$,为了保证编码的唯一性，规定<5的十进制表示为$0___$,而≥ 5的十进制数表示为$1___$。即5的表示为1011。

编码表示

1. 二进制无法表示所以小数，例如0.3无法通过0.5、0.25、0.125、$\cdots$线性表示。
2. 有符号数和无符号数：对于无符号数，寄存器中的每一位都可以用来存放数值。
3. 原码：以尾数表示真值的绝对值，符号位表示正负(0正1负)
    1. 以$8b$为例：对于原码存在$+0(0,000\ 0000)$和$-0(1,000\ 0000)$，前面给出的是整数，小数也是一样的。
    2. 对于$n+1$位机器字长：
        1. 若表示整数：$[-(2^n-1),(2^n-1)]$
        2. 若表示小数：$[-(1-2^{-n}),(1-2^{-n})]$
        3. 关于上述两个数怎么得出的需要解释一下，对于最大的数，我们采取$+1$或者$+2^{-n}$来计算，于是得到了2^n或者1，给出示意图：

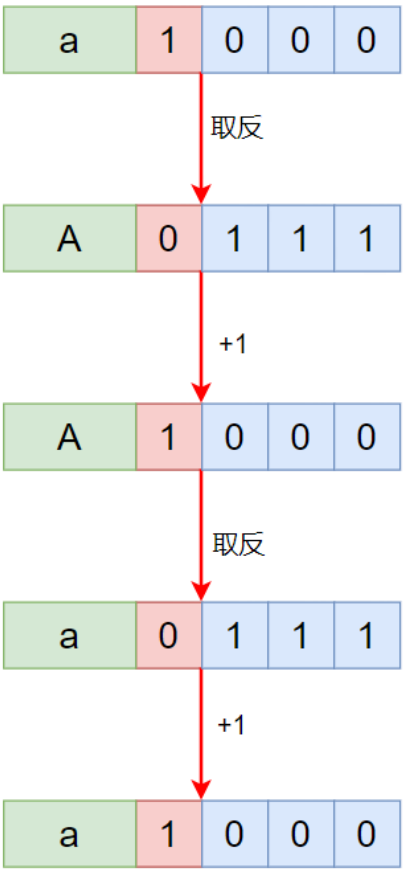


4. 反码：用于原码和补码相互转换的过度。
    1. 反码与原码的关系：反码与原码一一对应。
        1. 正数：反码与补码相同。
        2. 负数：其等于原码除符号位外按位取反。
    2. 以$8b$为例：对于反码存在$+0(0,000\ 0000)$和$-0(1,111\ 1111)$，前面给出的是整数，小数也是一样的。
    3. 对于$n+1$位机器字长：
        1. 若表示整数：$[-(2^n-1),(2^n-1)]$
        2. 若表示小数：$[-(1-2^{-n}),(1-2^{-n})]$
5. 补码：
    1. 实际上在二进制计算的过程中，对与加法和减法需要通过加法器和加法器进行计算，这就会增加底层硬件的设计成本。于是，便有了加法替代减法的操作。那如何使用加法代替减法呢？我们不妨参考一下时钟,如果需要将指针从7拨到3，可以逆时针拨动4下，即对应-4。也可以顺时针拨动8下，即对应$+8$。

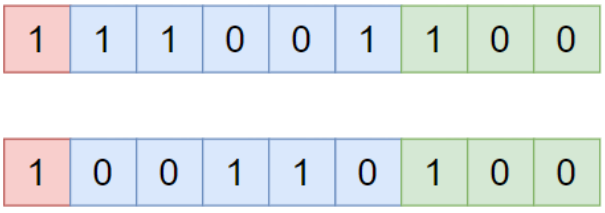


2. 那么对于上述过程，在执行减法$a-4$时其效果等价于$a+8$。这其实是因为时钟最大只能表示12，因此大于12的部分被舍弃。那么依此原理得出原码和补码的关系：
    1. 正数：原码和补码一致。
    2. 负数：补码等于原码除符号位外按位取反后$+1$。
    3. 实际上对于机器字长为$n+1$,$原码+补码=2^n$(这里的原码和补码默认不包含符号位的)。其中2^n可以认为是模。
    4. 实际上，对于一个负数的补码，我们计算时，是将其作为一个正数
3. 补码的补码就是原码。我们不妨从两个角度思考这句话：
    1. 从模的角度来说，$补码=2^n-原码$，同样的$原码=2^n-补码$。而对于二进制来说2^n-*的操作就是对$*$除符号位按位取反然后加一。
    2. 当然，不妨给出下列解释。实际上对原码求反码这个过程很好理解。重点在于此后会对反码执行加一操作得到补码，这个过程会导致从低位开始，反码中的1依次变为0，直到遇见0并将其变为1，则结束得到补码。显然这个过程会将反码末端的$01\cdots1$变回原码形式的$10\cdots0$，而补码前端的A部分仍和原码a相反。那么显然我们再次进行上述两步操作，

就又可以将补码转变为原码。



3. 对于上述的2过程，提供一个由原码求补码的新思路：对于负数，忽略符号位后，保留原码末尾部分的$10\cdots0$(0的个数可以是0),剩余部分全部取反。如下：



4. $+x$的补码等于$-x$的补码包括符号位在内按位取反后末尾加一，其实就是求$-x$的原码然后改变一下符号位。

5. 对于补码而言，真值0只有一种表现形式。以$8b$为例：对于原码的$+0(0,000\ 0000)$和$-0(1,000\ 0000)$，其对应的补码只有$0,000\ 0000$。那么显然，对于补码而言$+0(0,000\ 0000)$和$-0(1,000\ 0000)$都对应原码$0,000\ 0000$。于是乎就多了一个$1,000\ 0000$，对于整数我们将其记为-2^7，对于小数将其记为-1。

6. 那么显然对于机器字长为$n+1$，其原码：

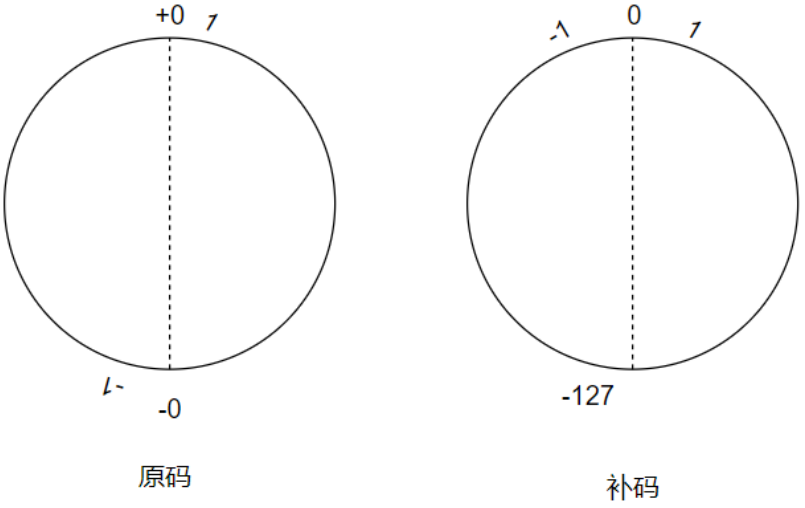
    1. 表示整数：$[-2^n,(2^n-1)]$
    2. 表示小数：$[-1,(1-2^n)]$
    3. 显然-1和-2^n不存在与之对应的原码和补码。

7. 执行加法时，补码的符号位可以一同参与运算。

1. 一旦发生溢出，就很容易被发现，例如$0,111+0,111=1,000$，显然两个正数相加变成负数，说明发生了溢出。当不发生溢出时。对于两个负数相加时，同理。
2. 对于一个负数加上一个正数，我们不妨思考一下，当二者相加为零，显然$+x$的补码等于$-x$的补码包括符号位在内按位取反后末尾加一，也就是说$[+x]_{补}$等于$[-x]_{原}$对符号位取反。那么不难得出$[-x]_{补}+[+x]_{补}=0$(自动舍去一个1)。那么：
    1. 当负数绝对值大于正数，负数对应补码的二进制大小其实会变小，不会进位，即符号位为1,表示负数。
    2. 当正数绝对值大于负数，正数对应当补码的二进制大小会变大，导致进位，即符号位变为0，表示正数。
3. 实际上，我们不难知道：只有同号相加才会导致溢出(负数+负数、正数+正数)，此时由于符号位的参与运算，就会导致符号位变化，从而很容易判断出溢出的存在。

6. 移码：其实就是将补码的符号位取反即可。
    1. 显然对于移码而言，真值0也是唯一的。
    2. 对于补码由于符号位和数据一同存储，我们不好比较大小。但是对于移码，正数的符号位变为1，负数的符号位变为0，此时就可以按位比较大小。
    3. 移码只能表示整数。

7. 小结：
    1. 给出原码和补码的示意图：

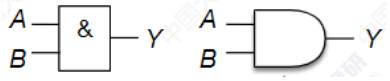


原码　　补码

    2. 再次强调：补码的补码是原码。
    3. 对于正数而言：补码、原码、反码三种相同。

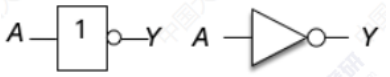
逻辑门电路

1. 逻辑运算：
    1. 与(and)：只有当输入A、B均为1时返回1，其他情况返回0，其实可以将AND视为乘法,只有$1*1=1$。
        1. $Y=A\cdot B$也可以简写为$Y=AB$。
        2. 门电路图形符号：(左边为国标画法，右边为国际常用画法)



    2. 或门(or)：只有当输出A、B均为0时返回0，其他情况返回1，其实可以将OR视为加法,只有$0+0=0$。
        1. $Y=A+B$
        2. 门电路图形符号：(左边为国标画法，右边为国际常用画法)

    3. 非门(not)：输出结果为输入A取反即可。例如$A=1$则输出0。
        1. $Y=\overline{A}$。
        2. 门电路图形符号：(左边为国标画法，右边为国际常用画法)

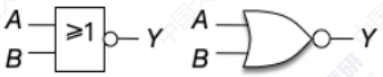


2. 复合逻辑运算：
    1. 与非($NAND$)：本质上就是在AND操作后对结果在执行一次NOT。故而只有当A、B输入均为0时，结果才为0，其他情况输出1。
        1. $Y=\overline{A\cdot B}$,当然也可以简写为$Y=\overline{AB}$。
        2. 门电路图形符号:(左边为国标画法，右边为国际常用画法)

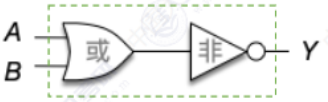
        3. 当然也可以使用一个与门加一个非门表示，只不过常用上述简化写法：

    2. 或非(NOR)：本质上就是在OR操作后对结果在执行一次NOT。故而只有当A、B输入均为0时，结果才为1，其他情况输出0。
        1. $Y=\overline{A+B}$

2. 门电路图形符号：(左边为国标画法，右边为国际常用画法)<b



3. 当然也可以使用一个或门加一个非门表示，只不过常用上述简化写法(封装：屏蔽电路部件的内部实现细节，竞对外部暴露出输入/输出引脚，使用者仅需关注该部件的核心功能即可)：



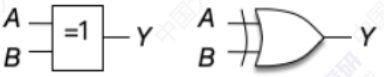
3. 异或(XOR)：输入A、B相异时，输出1，否则输出0。只有$A+B=1$才返回1。

1. $Y=A\oplus B$

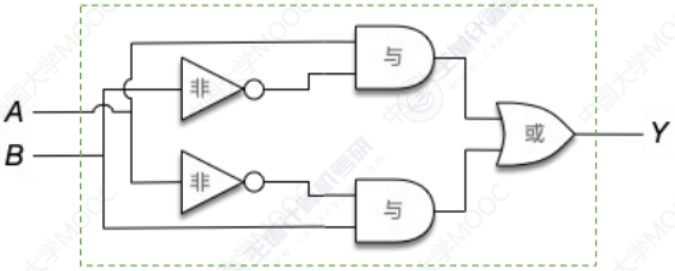
2. 真值表：

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

3. 门电路图形符号：



4. 当然其实$A\oplus B=\overline{A}B+A\overline{B}$,由此可以通过不同逻辑运算组合得到异或：



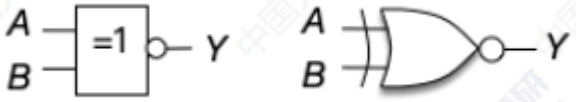
4. 同或($XNOR$)(有时可能会被叫做异或非门)：对异或取反，只有当A、B两个输入相同时返回1，否则返回0。只有$A+B\neq 1$才返回1。

1. $Y=A\odot B$(等价于$Y=\overline{A\oplus B}$)

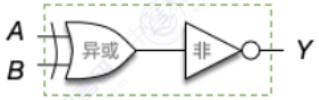
2. 真值表:

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

3. 门电路图形符号：



4. 显然同或也可以使用异或和非构成：



3. 异或运算的妙用：对于$n\ bit$进行异或操作，若其中1的个数为奇数，则异或结果为1，若1的个数为偶数个，则结果为0，该原理运用于后续的奇偶校验发、二进制加法。实际上对于异或操作而言是可以交换运算顺序的，那么若将所有的1放在一起计算就很容易推出上述结论，而0的个数对异或结果无影响，无论多少个0异或的结果都是0。

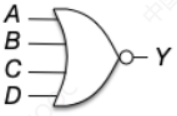
4. 逻辑门电路的补充：

多输入“或门”

Y=A+B+C+D，当且仅当所有输入都为0时，输出才为0

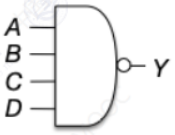
多输入“与门”

Y=A·B·C·D，当且仅当所有输入都为1时，输出才为1



多输入“或非门”

$Y=\overline{A+B+C+D}$，当且仅当所有输入都为0时，输出才为1



多输入“与非门”

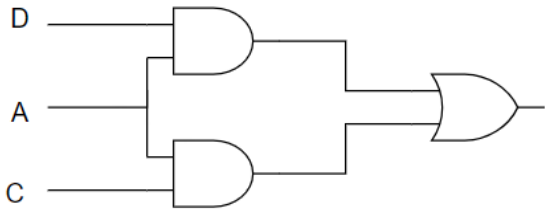
$Y=\overline{A\cdot B\cdot C\cdot D}$当且仅当所有输入都为1时，输出才为0

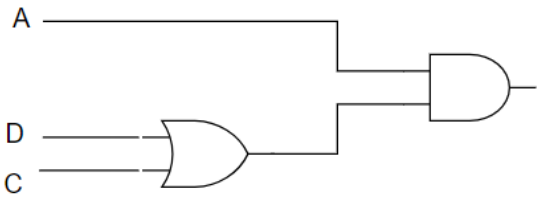
5. 逻辑运算的优先级：非>与或。存在括号，括号的优先级更高。对于非运算符下，可以默认理解为含有一个括号。

6. 逻辑运算的常见公式：

1. $A(C+D)=AC+AD$
2. $ABC+A(BC)$
3. $A+B+C=A+(B+C)$
4. $\overline{A+B}=\overline{A}\cdot\overline{B}$,反演律(德摩根定律)
5. $\overline{A\cdot B}=\overline{A}+\overline{B}$,反演律(德摩根定律)

7. 实际上利用上述公式可以简化电路设计，逻辑表达式是对电路的数学化表达。例如$AC+AD=A(C+D)$可以画出下面两种电路图：

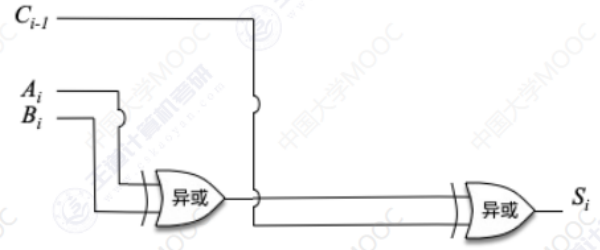




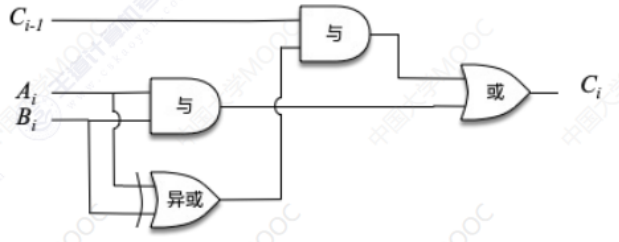
加法器

1. 我们不妨使用上述逻辑电路实现一位全加器：我们不妨假设A_i表示被加数当前参与运算的位置的数，B_i表示加数当前参与运算的位置的数，而C_i表示来自低位的进位，S_i表示当前位加法的结果。

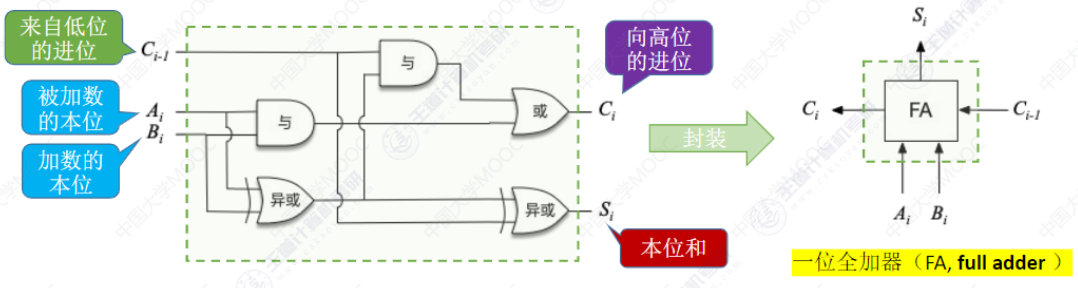
    1. S_i：显然其值与A_i、B_i、C_{i-1}都有关，当三者中存在奇数个1时，则输出1，显然为多位异或操作，即$S_i=A_i\oplus B_i\oplus C_{i-1}$。

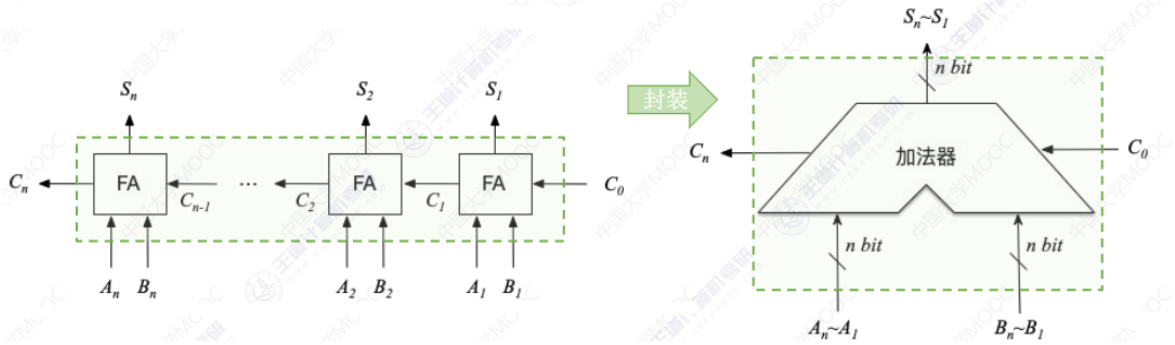
    2. C_i：显然A_i、B_i、C_i三者中至少存在2个1时才会进位，即A_i、B_i都是1，或者A_i、B_i中含有一个1且C_i为1，即$C_i=A_iB_i+(A_i\oplus B_i)C_{i-1}$。

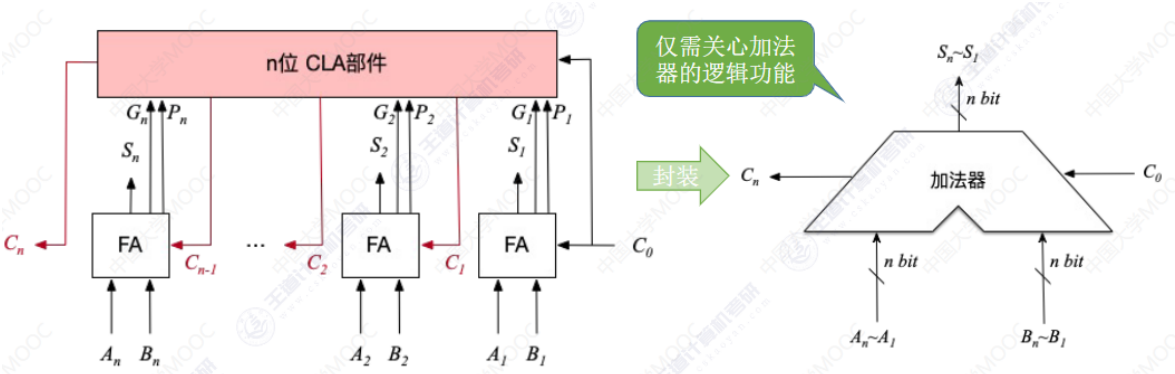
    3. 给出加法器的最终电路实现：

2. 实现一位全加器，那么多位全加器就是多个一位全加器串在一起即可。但是实际上这种全加器的缺点也很明显，即进位信息是串行产生的，每次计算都需要等待来自低位的进位C_{i-1},这就导致位数越多，运算速度越慢。由于两个输入端允许并行输入$n\ bit$，因此这种加法器属于并行加法器。但是由于进位信息是串行产生的，因此从进位方式来看，这种加法器属于串行进位加法器，故而综上所述很多教材称之为串行进位的并行加法器。



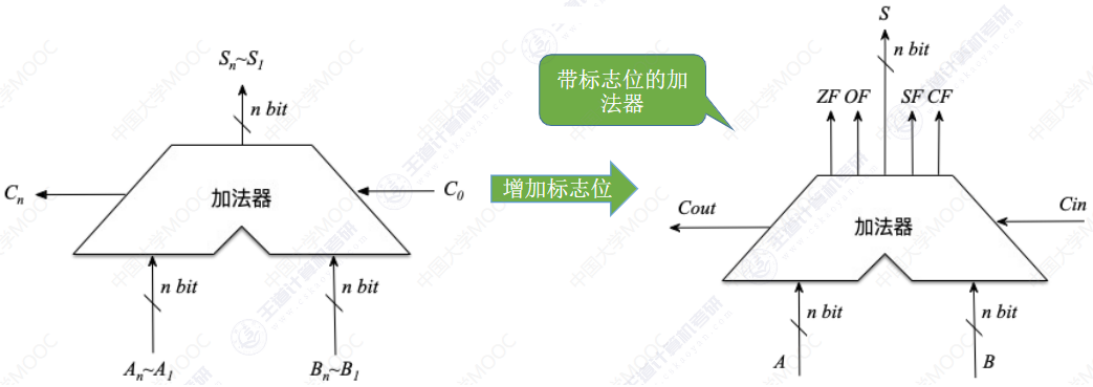
3. 实际上，C_i的数据A_i、B_i都是已知的，而C_{i-1}可以递推求得，故而我们还可以直接在最开始的时候计算出各个位的C_i，这样各位的加法几乎就可以并行进行，而不用等待来自低位的进位。对于这种加法器，其所有的进位信息几乎都是同时产生的，我们称之为并行进位的并行加法器。



4. 实际上加法器还需要一些标志位，来记录是否发生溢出，计算结果的正负等信息(似乎默认状态为0，那么就很好理解SF=1表示负值)：

    1. $OF(overflow\ flag)$: 溢出标志位，用于判断带符号数进行加减法运算时是否溢出。$OF=1$溢出，$OF=0$未溢出。
    2. $SF(symbol\ flag)$: 符号标志, 用于判断带符号数加减法运算结果的正负性。$SF=1$结果为负，$SF=0$结果为正。
    3. $ZF(zero\ flag)$: 零标志位，判断加减法的计算结果是否为0。运算结果的每一位都为0，$ZF=1$，否则$ZF=0$。
    4. $CF(carryflag)$: 进/借位标志位, 用于判断无符号数进行加减运算是否发生溢出。$CF=1$溢出，$CF=0$未溢出。
    5. 实际上上述的标志位在计算结束后，会送往标志寄存器。

6. br>


S_n~S_1
n bit
C_n
加法器
C_0
n bit
n bit
A_n~A_1
B_n~B_1
带标志位的加法器
增加标志位
S
n bit
ZF OF
SF CF
Cout
加法器
Cin
n bit
n bit
A
B