## 分布式运算（DA）

内积运算是数字信号处理中最为常见的运算：思考题，哪些信号处理算法可以抽象为内积呢？

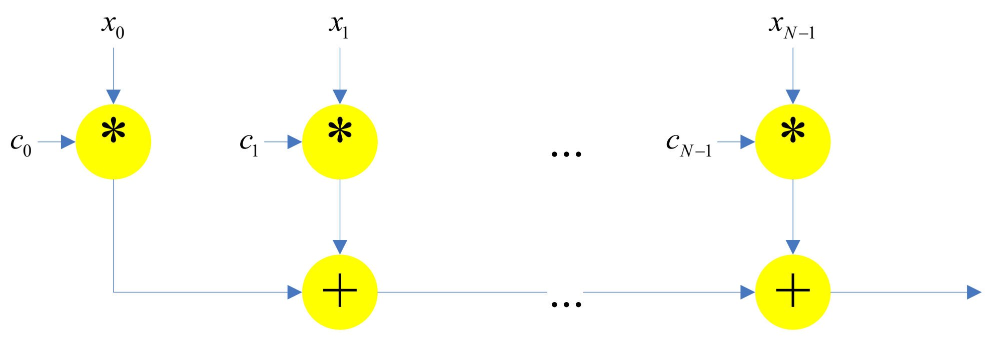



图 1常数向量内积，
其中$C=[c_0,c_1,...,c_{N-1}]$是常数向量，$X=[x_0,x_1,...,x_{N-1}]$是变数向量

首先要明白：一个通用乘法器和一个常系数乘法器的实现代价是不同的，前者比后者需要更多的面积，且功耗更大。而在大多数数字信号处理应用中，需要的只是常系数乘法，比如图 65 的内积结构。

针对图 65，工程上有很多优化的实现技术，其中最为著名的当属分布式算法。这里我们介绍基本的 DA 算法，及其若干改进，每一种改进都有其意味深长的地方，值得玩味！最后，介绍一个 DA 实现 IIR 滤波器的例子并结束本章。

将图 65 的计算过程列出来，如图 66 所示。正常的计算方式：先是一列列单独计算，也就是计算$a_i \cdot x_i$，然后再把所有列的结果累加起来；而 DA 反其道而行之，先是一行行计算（累加同一时段的不同乘法器的部分积），然后再把所有行的累加结果相加，比如第 j 行的计算就如图 67 所示。注意，从图 66 可以看出，行与行之间成 2 倍关系，用$P_j=p_j^5p_j^4p_j^3p_j^2p_j^1p_j^0$表示第 j 行的累加结果，则最终结果可表示为$\sum_{j=0}^{W-1}P_j$，W 为 x 的字长。其实每一行之间的计算都是相似的，不同的只是$\left[x_0^j,x_1^j,...,x_{N-1}^j\right]$，其中$x_i^j$为第 i 路输入$x_i$的第 j 位。针对$\left[x_0^j,x_1^j,...,x_{N-1}^j\right]$的所有组合，可制成 LUT 表，从而直接根据$\left[x_0^j,x_1^j,...,x_{N-1}^j\right]$查表得出每一行的累加结果$P_j$，从而大大加快系统计算速度。

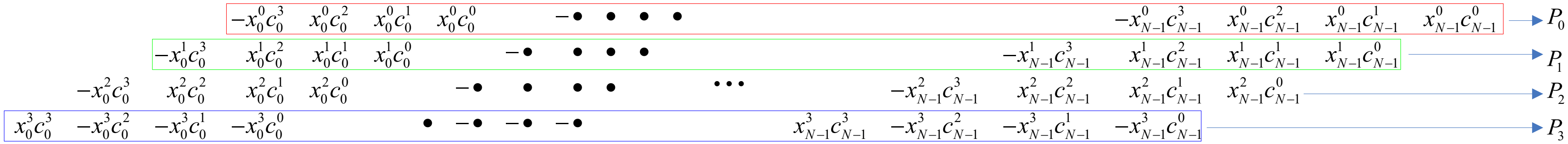


图 2DA 的部分积累加方式，先行后列;；而一般的方式是先列后行

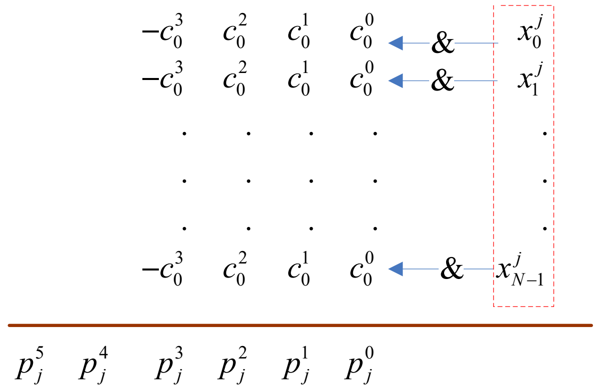


图 3第 j 行累加，一共有 W 行，也就是 j=0,1,…,W

以上是从图示说明 DA 的原理，我们也可以从数学公式的变换中得出 DA 结构（设计硬件电路，有时可借鉴数学上的“数形结合”思维方式，结合图形和解析式来分析，从而找到高效的实现方式）。

假设第 i 个输入为

$$X_i = x_i^{W-1} \dots x_i^1 x_i^0 = -x_i^{W-1} \cdot 2^{W-1} + \sum_{j=0}^{W-2} x_i^j \cdot 2^j \tag{1}$$

第 i 个乘法器的系数为

$$C_i = c_i^{B-1} \dots c_i^1 c_i^0 = -c_i^{B-1} \cdot 2^{B-1} + \sum_{j=0}^{B-2} c_i^j \cdot 2^j \tag{2}$$

注意，$C_i$的位长不必与$X_i$位长相同（实际很多情况下也是不同的），这里$X_i$的位长为 W，而$C_i$位长为 B。

内积输出可表示为

$$Y = \sum_{i=0}^{N-1} X_i \cdot C_i \tag{3}$$

注意，公式(34)给出的计算方式对应的就是先列后行（结合图 66 来分析），那么什么样的才是先行后列呢？请往下看，

$$Y = \sum_{i=0}^{N-1} X_i \cdot C_i = \underbrace{\overbrace{\sum_{i=0}^{N-1} \left( -x_i^{W-1} \cdot C_i \cdot 2^{W-1} + \sum_{j=0}^{W-2} x_i^j \cdot C_i \cdot 2^j \right)}^{\text{后行}}}_{\text{先列}} = \overbrace{-\sum_{i=0}^{N-1} x_i^{W-1} \cdot C_i \cdot 2^{W-1} + \sum_{j=0}^{W-2} \underbrace{\sum_{i=0}^{N-1} x_i^j \cdot C_i \cdot 2^j}_{\text{先行}}}^{\text{后列}}$$

(4)

从解析式(35)来看从一般计算方式到 DA 方式，其实就是交换求和次序。

根据图 66 的计算方式，设计 DA 结构如下，

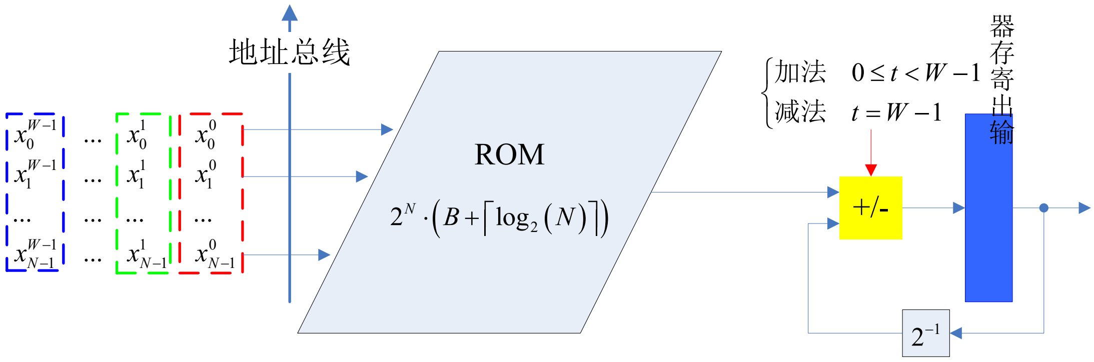



图 4标准的 DA 系统框图

这里，给出生成 ROM 数据的 MATLAB 代码，输入为 N 个系数$C_i, i = 0, 1, ..., N-1$，输出 ROM 的数据（程序中给出实数数据，实现时可用定点转换程序，将数据转化为规定格式定点数），

代码 1　标准 DA 的 ROM 数据生成

```
function [rom,Xi] = da_rom_nor(C)

N  = length(C);
Xi  = 0:2^N-1;
Xi  = dec2bin(Xi,N)-'0';
rom = Xi*C';
```

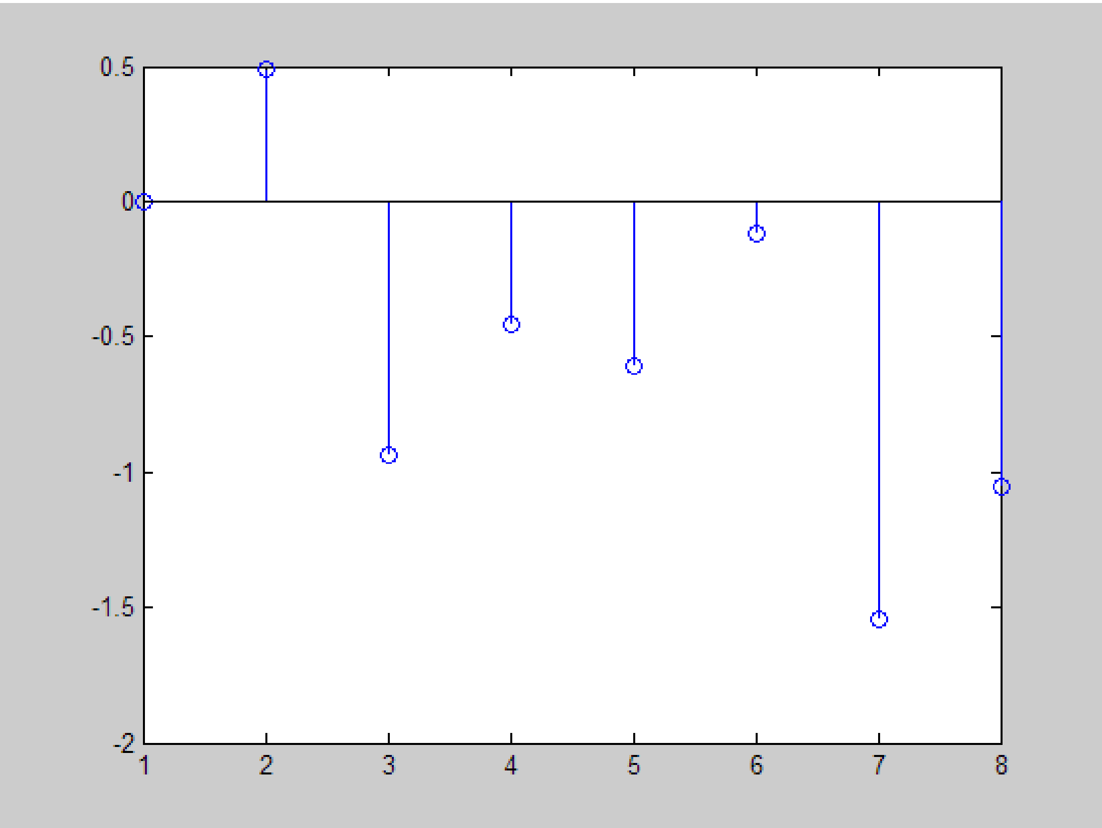
测试程序

代码 2　给定 N 个系数 Ci，输出 ROM 数据

```
function da_nor_demo()
clc
close all

N  = 3;
C  = rand(1,N)*2-1;

[rom,Xi] = da_rom_nor(C);

stem(rom);

disp('C');
disp(C);

disp('rom');
disp(rom);

disp('Xi');
disp(Xi);
```

C

```
   -0.6044    -0.9389     0.4881

rom
         0
    0.4881
   -0.9389
   -0.4508
   -0.6044
   -0.1162
   -1.5433
   -1.0551

Xi
     0     0     0
     0     0     1
     0     1     0
     0     1     1
     1     0     0
     1     0     1
     1     1     0
     1     1     1
```




%ROM 数据示意图

根据我们假设的参数，不难得出 ROM 的容量为$2^N \cdot (B + \lceil \log_2(N) \rceil)$比特，其中与 N 成指

数关系，这几乎是不可接受的；假设 N=32，那么 ROM 容量将是 Gbits 数量级。为了应用 DA，首要解决的减少 ROM 容量，否则 DA 算法就没有实用意义了。

缩减 ROM 容量有两个途径：

1） ROM 分解；

2） 偏移数编码（对称数形式）。

ROM 2 分解的 DA 框图如下

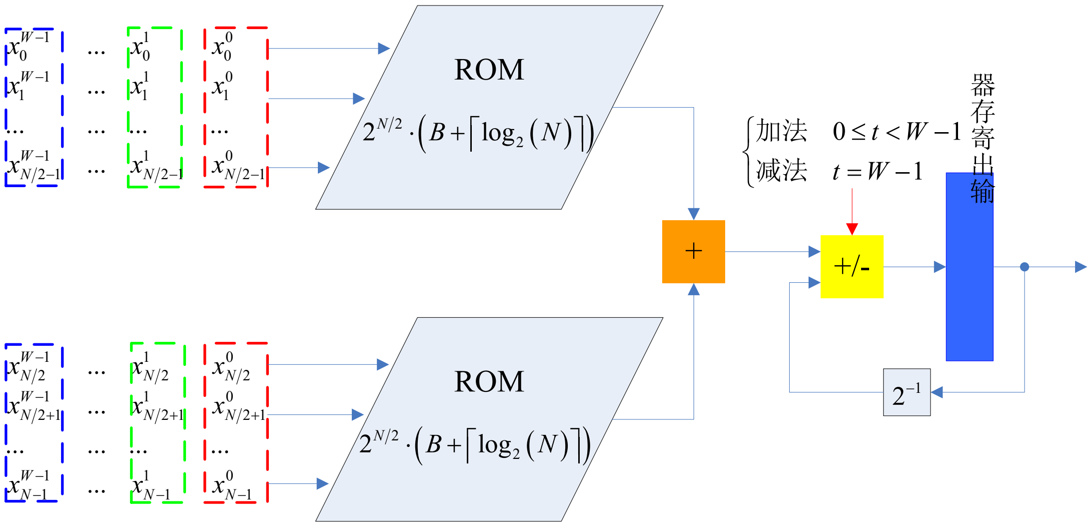



图 5ROM 2 分解 DA 框图

ROM 2 分解 DA 所需 ROM 容量为 $2\cdot 2^{N/2}\cdot(B+\lceil\log_2(N)\rceil)$，缩小程度非常可观，我们来测试一下不同程度分解 ROM 容量缩小的比例，

```
clear all
clc

format short e;

N   = 16;
B   = 16;
B   = B+log2(N);

rmsz= 2^N*B;

%   分解块数
M   = [2 4 8 16];
a   = zeros(1,length(M));
b   = zeros(1,length(M));
c   = zeros(1,length(M));

n   = 0;
for k   = M
```

```
    n   = n+1;
    a(n)= k*2^(N/k)*B;
    b(n)= k*2^(N/k);
    c(n)= 100*((rmsz-a(n))/rmsz);
end

disp([rmsz,2^N]);
disp('---------分解ROM-------------')
disp(M);
disp(a);
disp(b);
disp(c);
```

```
     1310720       65536

--------分解 ROM------------
     2     4     8    16

      10240        1280         640         640

   512    64    32    32

  9.9219e+001  9.9902e+001  9.9951e+001  9.9951e+001
```

考虑 N=32 的情况，分别使用[2,4,8,16]四种不同分解方案，从输出结果可看成，缩小程度都在 99.2%以上。不分解 ROM 需要 65535 个单元，而 2 分解只需 512 个单元，更甚的 8 分解只需 32 个单元。。分解所带来的 ROM 可节省 99.951%,，真是太令人兴奋了。当然，图 69 也说明分解所带来的代价，就是需要而外加法器，将各个分开的 ROM 输出进行合并（相加，橙色加法器）。

高阶 ROM 分解的 DA 实现，需要多个合并加法器，为了加快合并速度，应该使用树形加法方式同时结合 CSA 累加技术,,，或者是在树形加法器中插入流水线——不论怎么说，要设计一个快速合并模块是很容易的！

使用 ROM 分解技术，已经可以极大缩减 ROM 容量，但是“大虾们”还给出了进一步缩减 ROM 容量的方法：偏移数制（OBC）编码——注意，这是一种非常巧妙且难于想到的处理方式，值得大家挑战！

偏移数制编码（我更喜欢称之为 对称数制编码），来源于这么一个设想，

首先来观察

**表格 1　标准 2C 编码 DA 的 ROM 内容**

| $x_0^j$ | $x_1^j$ | $x_2^j$ | $x_3^j$ | ROM 的内容 |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | C3 |
| 0 | 0 | 1 | 0 | C2 |
| 0 | 0 | 1 | 1 | C2+C3 |
| 0 | 1 | 0 | 0 | C1 |
| 0 | 1 | 0 | 1 | C1+C3 |
| 0 | 1 | 1 | 0 | C1+C2 |
| 0 | 1 | 1 | 1 | C1+C2+C3 |
| 1 | 0 | 0 | 0 | C0 |
| 1 | 0 | 0 | 1 | C0+C3 |
| 1 | 0 | 1 | 0 | C0+C2 |
| 1 | 0 | 1 | 1 | C0+C2+C3 |
| 1 | 1 | 0 | 0 | C0+C1 |
| 1 | 1 | 0 | 1 | C0+C1+C3 |
| 1 | 1 | 1 | 0 | C0+C1+C2 |
| 1 | 1 | 1 | 1 | C0+C1+C2+C3 |

其实，从代码 8 的结果图中也能看出，ROM 的数据不具有“对称的规律”，也就是说没有冗余性（对称性思路来源于正弦函数数据的存储，利用对称性总是可以节约资源）。大虾们经过分析思考，发现，之所以 ROM 数据不对称，原因在于 2C 补码编码的不对称，2C 编码数

$$X_i = -x_i^{W-1} \cdot 2^{W-1} + \sum_{j=0}^{W-2} x_i^j \cdot 2^j \tag{5}$$

中，$x_i^j \in \{0,1\}$，也就是说$x_i^j$的取值是不对称的，0 和 1 显然不是对称关系，-1 和 1 倒是刚好对称。

如果能用对称的形式来编码$X_i$，会不会“幸运”地造成 ROM 数据成对称关系呢？经过巧妙的构造，得到一种对称数制编码如下

$$\begin{aligned}
X_i &= \frac{1}{2}\left(X_i - \left(-X_i\right)\right) \\
&= \frac{1}{2}\left(\left(-x_i^{W-1} \cdot 2^{W-1} + \sum_{j=0}^{W-2} x_i^j \cdot 2^j\right) - \left(-\overline{x}_i^{W-1} \cdot 2^{W-1} + \sum_{j=0}^{W-2} \overline{x}_i^j \cdot 2^j + LSB'1\right)\right) \\
&= \frac{1}{2}\left(-\left(x_i^{W-1} - \overline{x}_i^{W-1}\right) \cdot 2^{W-1} + \sum_{j=0}^{W-2}\left(x_i^j - \overline{x}_i^j\right) \cdot 2^j + LSB'1\right) \\
&= \frac{1}{2}\left(-d_i^{W-1} \cdot 2^{W-1} + \sum_{j=0}^{W-2} d_i^j \cdot 2^j + LSB'1\right)
\end{aligned} \tag{6}$$

其中，

$$\begin{cases} d_i^{W-1} = x_i^{W-1} - \overline{x}_i^{W-1} \\ d_i^j = x_i^j - \overline{x}_i^j \end{cases} \tag{7}$$

很巧妙的一点在于，

$$
\begin{array}{llllll}
x_i^{W-1} & = & 0 & \rightarrow & d_i^{W-1} & = & -1 \\
x_i^{W-1} & = & 1 & \rightarrow & d_i^{W-1} & = & 1 \\
x_i^{j} & = & 0 & \rightarrow & d_i^{j} & = & -1 \\
x_i^{j} & = & 1 & \rightarrow & d_i^{j} & = & 1
\end{array}
\tag{8}
$$

从公式(39)可以看出，$d_i^j \in \{-1, 1\}, j = 0, 1, ..., W-1$，形成一种对称编码，使用这种编码来编码$X_i$，将得到偏移数制 DA 实现，

$$
\begin{aligned}
Y &= \sum_{i=0}^{N-1} X_i \cdot C_i = \sum_{i=0}^{N-1} \frac{1}{2}\left(-d_i^{W-1}\cdot 2^{W-1} + \sum_{j=0}^{W-2} d_i^j \cdot 2^j + LSB'1\right)\cdot C_i \\
&= \frac{1}{2}\left(\sum_{i=0}^{N-1}\left(-d_i^{W-1}\cdot 2^{W-1} + \sum_{j=0}^{W-2} d_i^j \cdot 2^j + LSB'1\right)\cdot C_i\right) \\
&= \frac{1}{2}\left(\sum_{i=0}^{N-1}\left(-d_i^{W-1}\cdot 2^{W-1} + \sum_{j=0}^{W-2} d_i^j \cdot 2^j\right)\cdot C_i + \sum_{i=0}^{N-1} LSB'1\cdot C_i\right) \\
&= \frac{1}{2}\left(\overbrace{-\sum_{i=0}^{N-1} d_i^{W-1}\cdot 2^{W-1}\cdot C_i + \underbrace{\sum_{j=0}^{W-2}\sum_{i=0}^{N-1} d_i^j \cdot C_i \cdot 2^j}_{\text{先行}}}^{\text{后列}} + \underbrace{\sum_{i=0}^{N-1} LSB'1\cdot C_i}_{\text{额外一项}}\right)
\end{aligned}
\tag{9}
$$

根据公式(40)设计的结构为

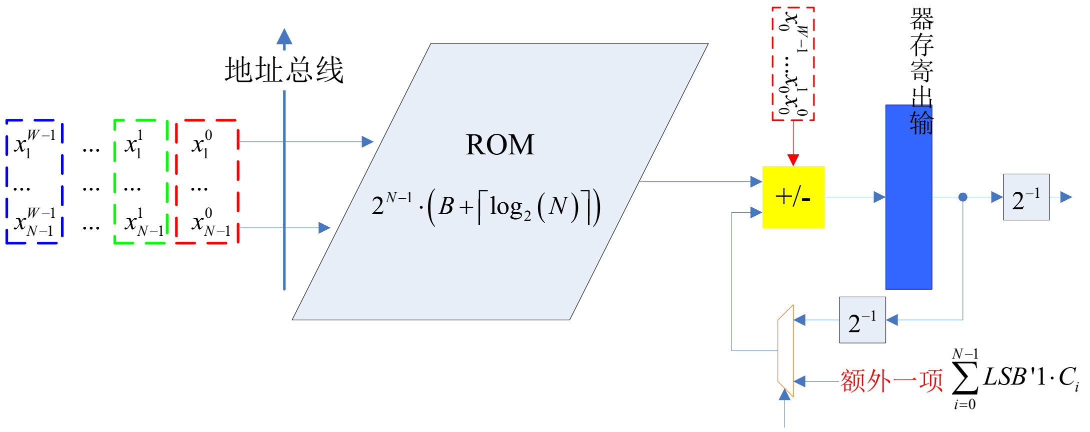



图 6使用偏移数制的 DA 框架

表格 2　偏移数制编码 DA 的 ROM 内容

| 2C 编码 | | | | 偏移数制编码 | | | | ROM 的内容 |
|---|---|---|---|---|---|---|---|---|
| $x_0^j$ | $x_1^j$ | $x_2^j$ | $x_3^j$ | $d_i^0$ | $d_i^1$ | $d_i^2$ | $d_i^3$ | ROM 的内容 |
| 0 | 0 | 0 | 0 | -1 | -1 | -1 | -1 | -C0-C1-C2-C3 |
| 0 | 0 | 0 | 1 | -1 | -1 | -1 | 1 | -C0-C1-C2+C3 |
| 0 | 0 | 1 | 0 | -1 | -1 | 1 | -1 | -C0-C1+C2-C3 |
| 0 | 0 | 1 | 1 | -1 | -1 | 1 | 1 | -C0-C1+C2+C3 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | -1 | 1 | -1 | -1 | -C0+C1-C2-C3 |
| 0 | 1 | 0 | 1 | -1 | 1 | -1 | 1 | -C0+C1-C2+C3 |
| 0 | 1 | 1 | 0 | -1 | 1 | 1 | -1 | -C0+C1+C2-C3 |
| 0 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | -C0+C1+C2+C3 |
| 1 | 0 | 0 | 0 | 1 | -1 | -1 | -1 | -(-C0+C1+C2+C3) |
| 1 | 0 | 0 | 1 | 1 | -1 | -1 | 1 | -(-C0+C1+C2+C3) |
| 1 | 0 | 1 | 0 | 1 | -1 | 1 | -1 | -(-C0+C1-C2+C3) |
| 1 | 0 | 1 | 1 | 1 | -1 | 1 | 1 | -(-C0+C1-C2-C3) |
| 1 | 1 | 0 | 0 | 1 | 1 | -1 | -1 | -(-C0-C1+C2+C3) |
| 1 | 1 | 0 | 1 | 1 | 1 | -1 | 1 | -(-C0-C1+C2-C3) |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | -1 | -(-C0-C1-C2+C3) |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | -(-C0-C1-C2-C3) |

注意到，ROM 中的内容是关于中间黑线上下对称的（互为相反数），也就是说，ROM 只需存储一半的数据即可，，，

这里给出基于偏移数制编码的 ROM 数据生成 MATLAB 程序，

**代码 3　偏移数制编码 DA 的 ROM 数据生成**

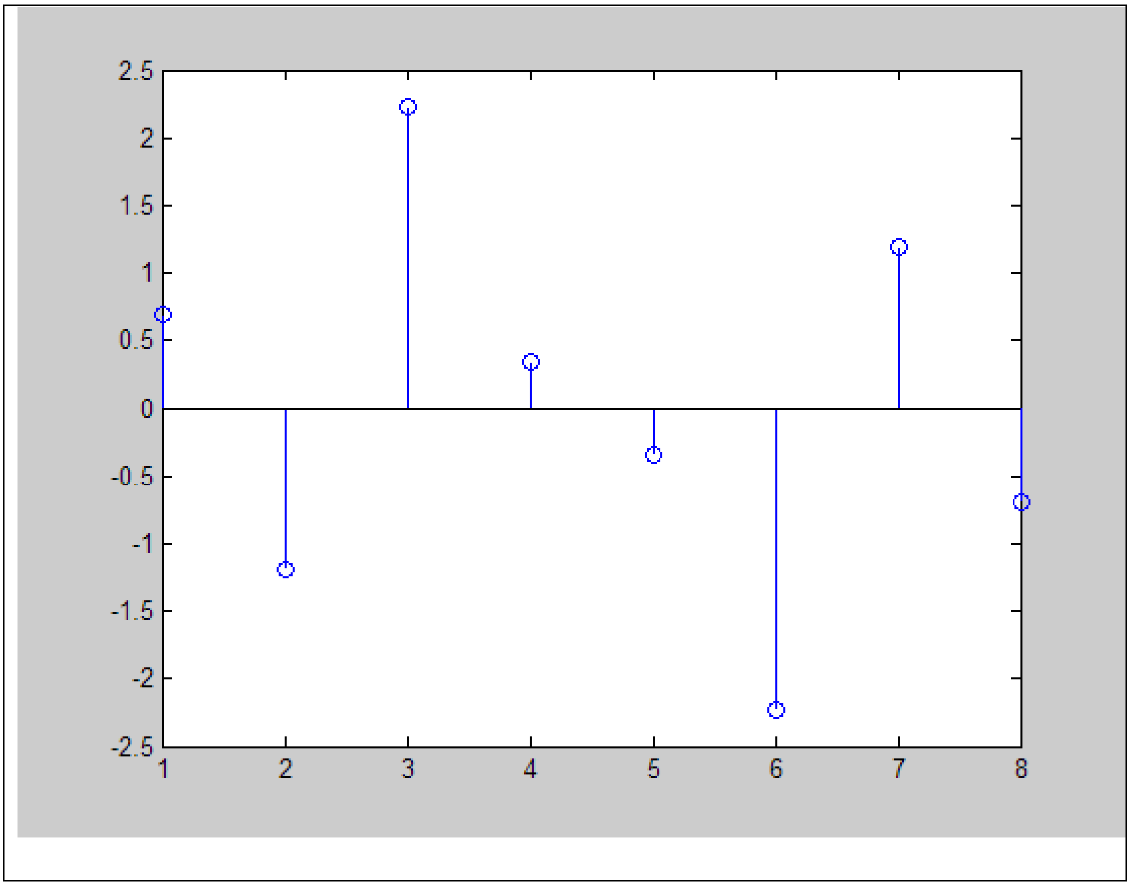
```
function [rom,Xi,Di] = da_rom_obc(C)

N   = length(C);
Xi  = 0:2^N-1;
Xi  = dec2bin(Xi,N)-'0';
Di  = Xi-not(Xi);

rom = Di*C';
```

测试程序，

```
function da_obc_demo()
clc
close all

N   = 3;
C   = rand(1,N)*2-1;

[rom,Xi,Di] = da_rom_obc(C);
stem(rom);

disp('C');
disp(C);
disp('Di');
```

```
disp(Di);

disp('rom');
disp(rom);

disp('Xi');
disp(Xi);
```

```
C
  -5.2014e-001   7.7302e-001 -9.4265e-001

Di
    -1    -1    -1
    -1    -1     1
    -1     1    -1
    -1     1     1
     1    -1    -1
     1    -1     1
     1     1    -1
     1     1     1

rom
   6.8976e-001
  -1.1955e+000
   2.2358e+000
   3.5051e-001
  -3.5051e-001
  -2.2358e+000
   1.1955e+000
  -6.8976e-001

Xi
     0     0     0
     0     0     1
     0     1     0
     0     1     1
     1     0     0
     1     0     1
     1     1     0
     1     1     1
```




，ok，最后在给出一个 IIR DA 的例子，这里只给出设计框图，大家感兴趣可以自行实现之，，

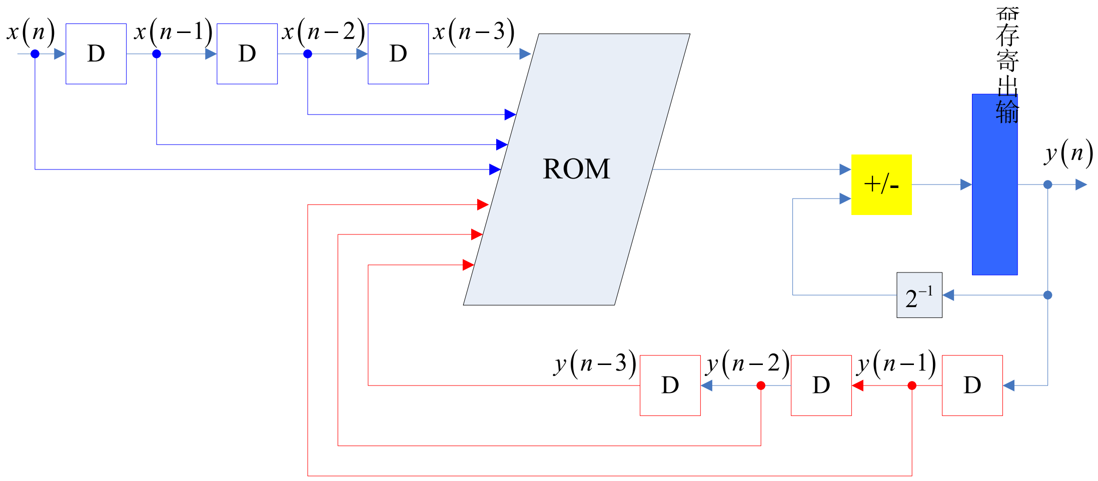



图 7IIR DA 实现示意图

思考题：从图 68 的标准 DA 框架中可知，后半段是一个循环累加的过程，，这个循环累加需要 W 个迭代周期才能完成，试问：如何设计一个“全并行”的 DA，使用树型累加方式来算出结果，而不用循环累加方式（全并行 DA 又称为高阶 DA）？——问题来源于《数字信号处理的 FPGA 实现》，Uwe Meyer-Baese 著。

好了，本章内容到此结束。。GOODLUCK！

# 附录 A 数的表示及其加法运算

虽然本章讨论的是乘法器的位级架构设计，但乘法器实际上等价于多次加法，所以有必要讨论一下基本的加法运算性质，更为详细的知识可以从计算机体系结构或数字设计等书籍中找到（如果你对加法运算已经非常熟悉，可以跳过这一段）。

数字有多种不同表示，下面只介绍本章用到的。数可分为无符号数和有符号数两类，无符号数表示比较简单，如公式(41)

$$a_{N-1}\cdots a_1 a_0.\underline{a}_1\cdots\underline{a}_{M-1}=\sum_{i=0}^{N-1}a_i\cdot 2^i+\sum_{i=0}^{M-1}\underline{a}_i\cdot 2^{-i} \tag{10}$$

有符号数表示稍微复杂，常用原码表示或 2C 表示。

原码表示，最高位为符号位（0 为正数，1 为负数），除最高位之外的其他位表示数字的绝对值大小，如公式(42)

$$\left[x\right]_{原}=a_{N-1}\cdots a_1 a_0=\left(-1\right)^{a_{N-1}}\cdot\sum_{i=0}^{N-2}a_i\cdot 2^i \tag{11}$$

如果是$(-1,1)$之间的小数，注意不包括$\pm1$，如公式(43)

$$\left[x\right]_{原}=a_0.\underline{a}_1\cdots\underline{a}_{N-1}=\left(-1\right)^{a_0}\cdot\sum_{i=1}^{N-1}\underline{a}_i\cdot 2^{-i} \tag{12}$$

根据定义，0 有两种表示，即$0.0\cdots0$或者$1.0\cdots0$。

2C 表示，仍然是最高位为符号位，但具体表示公式不同，如公式(44)和(45)

$$\left[x\right]_{2C}=a_{N-1}\cdots a_1 a_0=-a_{N-1}+\sum_{i=0}^{N-2}a_i\cdot 2^i \tag{13}$$

$$\left[x\right]_{2C}=a_0.\underline{a}_1\cdots\underline{a}_{N-1}=-a_0+\sum_{i=1}^{N-1}\underline{a}_i\cdot 2^{-i} \tag{14}$$

在数字逻辑设计课程中，学习过“原码和 2C 的互相转化”，这里复习一遍，其中会涉及到某些容易混淆的概念，大家务必小心谨慎！注意，通过这种讨论，是可以提高大家对“位”的感觉，从而能更好的讨论位级运算架构！

首先是：原码到 2C 的转化，可分 3 种情况，

1） 若 x 为正数，则

$$\begin{cases}\left[x\right]_{原} = 0.\underline{a}_1\cdots\underline{a}_{N-1} = \left(-1\right)^0\cdot\sum_{i=1}^{N-1}\underline{a}_i\cdot 2^{-i} = \sum_{i=1}^{N-1}\underline{a}_i\cdot 2^{-i}\\ \left[x\right]_{2C} = 0.\underline{a}_1\cdots\underline{a}_{N-1} = -a_0+\sum_{i=1}^{N-1}\underline{a}_i\cdot 2^{-i} = \sum_{i=1}^{N-1}\underline{a}_i\cdot 2^{-i}\end{cases} \tag{15}$$

公式(46)的上下两式结果一样，也就是说正数的原码和 2C 表示是一模一样的，直接 copy 即可。

2）若 x 为 0，那么原码可表示为$0.0\cdots0$或者$1.0\cdots0$，转化为 2C，必须是$0.0\cdots0$；注意 2C 表示的$1.0\cdots0$是-1，而不是 0。

3）若 x 为负数，则

$$[x]_{\text{原}} = 1.\underline{a}_1 \cdots \underline{a}_{N-1} = -\sum_{i=1}^{N-1} \underline{a}_i \cdot 2^{-i} \tag{16}$$

另外，负数的 2C 表示为

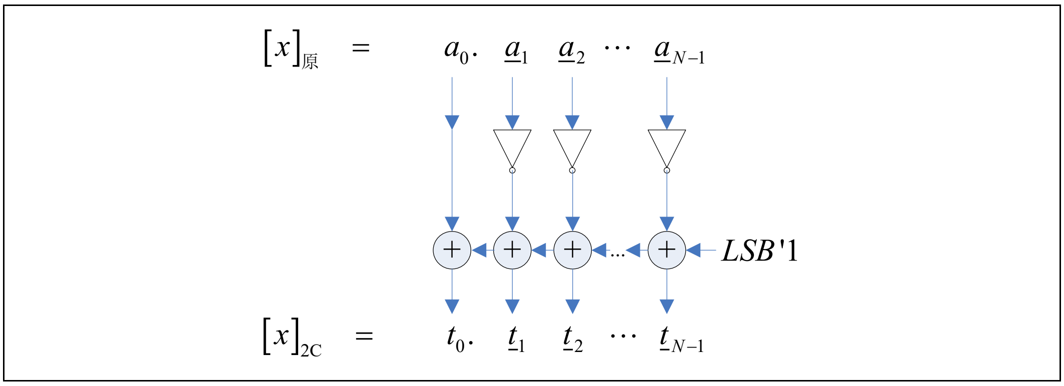
$$[x]_{2C} = 1.\underline{t}_1 \cdots \underline{t}_{N-1} = -1 + \sum_{i=1}^{N-1} \underline{t}_i \cdot 2^{-i} \tag{17}$$

要进行负数原码到 2C 的转化，必须找到$\underline{a}_i$和$\underline{t}_i$的关系（别只是看，你也动手试试），

$$\begin{aligned}
& -\sum_{i=1}^{N-1} \underline{a}_i \cdot 2^{-i} && = && -1 + \sum_{i=1}^{N-1} \underline{t}_i \cdot 2^{-i} \\
\Leftrightarrow\quad & 1 - \sum_{i=1}^{N-1} \underline{a}_i \cdot 2^{-i} && = && \sum_{i=1}^{N-1} \underline{t}_i \cdot 2^{-i} \\
\Leftrightarrow\quad & 0.\underbrace{1\cdots11}_{N-1\text{个}1} + 0.0\cdots0\underset{\text{第N-1位}}{1} - \sum_{i=1}^{N-1} \underline{a}_i \cdot 2^{-i} && = && \sum_{i=1}^{N-1} \underline{t}_i \cdot 2^{-i} \\
\Leftrightarrow\quad & \underbrace{\left( \underbrace{\left( 0.\underbrace{1\cdots11}_{N-1\text{个}1} - \sum_{i=1}^{N-1} \underline{a}_i \cdot 2^{-i} \right)}_{\text{各位取反}} + 0.0\cdots0\underset{\text{第N-1位}}{1} \right)}_{\text{LSB 加 1}} && = && \sum_{i=1}^{N-1} \underline{t}_i \cdot 2^{-i}
\end{aligned} \tag{18}$$

公式(49)推导过程非常清楚地给出了负数原码到 2C 表示的过程，归结如下：保持符号位不变，其余各位取反并在 LSB 上加 1。$\underline{a}_i$取反表示为$\overline{\underline{a}_i}$，即加上划线；在 LSB 上加 1，表示为$+LSB'1$。

> *题外话：进行硬件设计时，有些同学担心$\overline{\underline{a}}_1\overline{\underline{a}}_2\cdots\overline{\underline{a}}_{N-1}+LSB'1$会不会出现计算溢出，下面将证明，不会出现溢出；然后我们来设计一种转化电路。*
>
> 因为 x 为负数（$\neq 0$），所以$\underline{a}_1\underline{a}_2\cdots\underline{a}_{N-1}$不全为 0，因此$\overline{\underline{a}}_1\overline{\underline{a}}_2\cdots\overline{\underline{a}}_{N-1}$不全为 1，所以$\overline{\underline{a}}_1\overline{\underline{a}}_2\cdots\overline{\underline{a}}_{N-1}+LSB'1$不会溢出。
>
> 话又说回来，当$[x]_{\text{原}} = 1.0\cdots0$时，非符号位取反并$+LSB'1$，就会导致向符号位的进位，改进位和符号位相加，得 0 并产生更高位进位;;;如果抛弃更高位进位，将得到$0.0\cdots0$，这恰恰好是我们想要的转化结果，所以负数原码到 2C 的转化电路如下，




小结以上原码到 2C 的转化方式：

1） 符号位为 0，直接 copy；

2） 符号位为 1，除符号位之外的其他各个位取反，然后$+LSB'1$，溢出自然舍弃。

反过来看看 2C 到原码的转化，虽然这种转化可能意义不大，但做为位级硬件设计的练习是不错的，，仍然分三种情况：

1） x 为正数，则 2C 到原码直接 copy 即可；

2） x 为 0，则 $\underbrace{0.0\cdots0}_{2C} \to \underbrace{1.0\cdots0或0.0\cdots0}_{原码}$；

3） x 为负数，那么 $1.\underline{t}_1\underline{t}_2\cdots\underline{t}_{N-1} \xrightarrow{?} 1.\underline{a}_1\underline{a}_2\cdots\underline{a}_{N-1}$应该是什么样的呢？

$$
\begin{aligned}
& -1+\sum_{i=1}^{N-1}\underline{t}_i\cdot 2^{-i} && = -\sum_{i=1}^{N-1}\underline{a}_i\cdot 2^{-i} \\
\Leftrightarrow\ & 1-\sum_{i=1}^{N-1}\underline{t}_i\cdot 2^{-i} && = \sum_{i=1}^{N-1}\underline{a}_i\cdot 2^{-i} \\
\Leftrightarrow\ & 0.\underbrace{1\cdots11}_{N-1个1}+0.0\cdots0\underbrace{1}_{第N-1位}-\sum_{i=1}^{N-1}\underline{t}_i\cdot 2^{-i} && = \sum_{i=1}^{N-1}\underline{a}_i\cdot 2^{-i} \\
\Leftrightarrow\ & \underbrace{\left(\underbrace{\left(0.\underbrace{1\cdots11}_{N-1个1}-\sum_{i=1}^{N-1}\underline{t}_i\cdot 2^{-i}\right)}_{各位取反}+0.0\cdots0\underbrace{1}_{第N-1位}\right)}_{LSB加1} && = \sum_{i=1}^{N-1}\underline{a}_i\cdot 2^{-i}
\end{aligned}
\tag{19}
$$

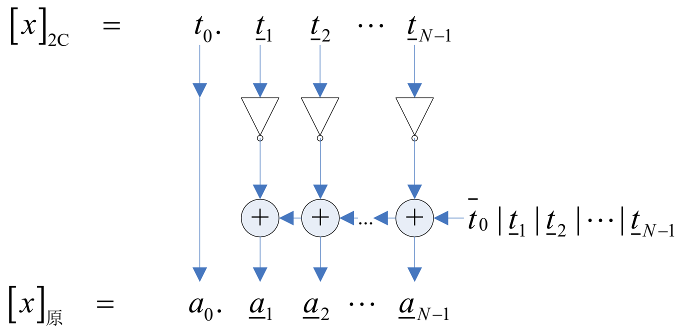
公式(50)的推导过程和公式(49)是相同的，所以负数的 2C 到原码的转化是：除符号位之外的各位取反，并$+LSB'1$。

题外话：同样的，有些同学担心$\bar{t}_1\bar{t}_2\cdots\bar{t}_{N-1}+LSB'1$会不会出现计算溢出，下面分析一下。

$\bar{t}_1\bar{t}_2\cdots\bar{t}_{N-1}+LSB'1$溢出，除非$\bar{t}_1\bar{t}_2\cdots\bar{t}_{N-1}$全为 1，也就是$\underline{t}_1\underline{t}_2\cdots\underline{t}_{N-1}$全为 0，也就是说 x 的 2C 表示为$1.\underbrace{00\cdots0}_{N-1个0}$，此时 x=-1；前面说过，这种情况下的原码表示不了-1。对于溢出的情况，如果“强制符号位不变”，则所得原码为$1.\underbrace{00\cdots0}_{N-1个0}=0$，这是错误的结果，

如果采用自然舍弃，所得原码为$0.\underbrace{00\cdots0}_{N-1个0}=0$，结果都是将-1 转为 0，所以也是错的。

如果允许转化误差，那么可以用原码$1.\underbrace{11\cdots1}_{N-1个0}$近似表示-1。硬件设计时，可以设立检测电路，当检测到输入的 2C 表示为$1.00\cdots0$，对符号位之外的各位取反，但不$+LSB'1$。负数 2C 到原码转化电路如下




输入不为$1.\underbrace{00\cdots0}_{N-1个0}$时，$+LSB'1$起作用；输入为$1.\underbrace{00\cdots0}_{N-1个0}$时，$\bar{t}_0 | t_1 | t_2 | \cdots | t_{N-1} = 0$。

小结以上 2C 到原码的转化方式：

1） 符号位为 0，直接 copy；

2） 符号位为 1，除符号位之外的其他各个位取反，如果不是$x \neq 1.\underbrace{00\cdots0}_{N-1个0}$，$+LSB'1$，否则转化完毕。

---

下面将注意力转移到 2C 表示的一些运算上，其中一个容易混淆又非常重要的是：2C 取相反数，

假设 A 和 B 均为 2C 表示的数，且 B=-A，则有，

$$
\begin{aligned}
B &= -A \\
\Leftrightarrow\quad -b_0+\sum_{i=1}^{N-1}\underline{b}_i\cdot 2^{-i} &= -\left(-a_0+\sum_{i=1}^{N-1}\underline{a}_i\cdot 2^{-i}\right) \\
\Leftrightarrow\quad -b_0+\sum_{i=1}^{N-1}\underline{b}_i\cdot 2^{-i} &= a_0-\sum_{i=1}^{N-1}\underline{a}_i\cdot 2^{-i} \\
\Leftrightarrow\quad -b_0+\sum_{i=1}^{N-1}\underline{b}_i\cdot 2^{-i} &= (a_0-1)+\left(1-\sum_{i=1}^{N-1}\underline{a}_i\cdot 2^{-i}\right) \\
\Leftrightarrow\quad -b_0+\sum_{i=1}^{N-1}\underline{b}_i\cdot 2^{-i} &= (a_0-1)+\underbrace{\left(\underbrace{\left(0.\underbrace{1\cdots 11}_{N-1\text{个}1}-\sum_{i=1}^{N-1}\underline{a}_i\cdot 2^{-i}\right)}_{\text{各位取反}}+0.0\cdots 0\underset{\text{第N-1位}}{1}\right)}_{\text{LSB 加 1}}
\end{aligned}
\tag{20}
$$

分两种情况讨论，

1） 如果$\bar{\underline{a}}_1\bar{\underline{a}}_2\cdots\bar{\underline{a}}_{N-1}+LSB'1$不会溢出，则公式(51)的结果可表示为

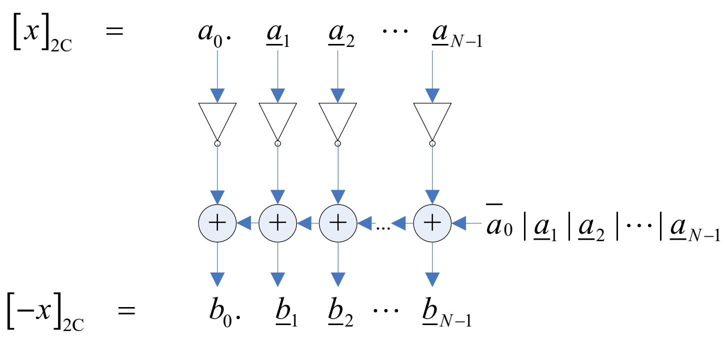
$$
b_0.\underline{b}_1\underline{b}_2\cdots\underline{b}_{N-1}=\bar{a}_0.\bar{\underline{a}}_1\bar{\underline{a}}_2\cdots\bar{\underline{a}}_{N-1}+LSB'1 \tag{21}
$$

2） 如果$\bar{\underline{a}}_1\bar{\underline{a}}_2\cdots\bar{\underline{a}}_{N-1}+LSB'1$溢出，则必有$x=1.\underbrace{00\cdots 0}_{N-1\text{个}0}$或者$0.\underbrace{00\cdots 0}_{N-1\text{个}0}$。硬按公式(52)来进行转化，则会有

$$
\begin{cases}
1.\underbrace{00\cdots 0}_{N-1\text{个}0}\xrightarrow{\text{各位取反}}0.\underbrace{11\cdots 1}_{N-1\text{个}1}\xrightarrow{+LSB'1}1.\underbrace{00\cdots 0}_{N-1\text{个}0} \\
0.\underbrace{00\cdots 0}_{N-1\text{个}0}\xrightarrow{\text{各位取反}}1.\underbrace{11\cdots 1}_{N-1\text{个}1}\xrightarrow{+LSB'1}10.\underbrace{00\cdots 0}_{N-1\text{个}0}\xrightarrow{\text{自然舍弃}}0.\underbrace{00\cdots 0}_{N-1\text{个}0}
\end{cases}
\tag{22}
$$

公式(53)的上式将-1 转化为-1，下式是 0 转化为 0,，注意-1 的相反数为 1，不幸的是此时 2C 表示不了 1，只能表示$[-1,1)$范围内的数,，0 的相反数仍为 0，这个是正确的。总之，我们可用公式(52)来进行取相反数操作，只要是$(-1,1)$结果就不会出错。

思考题：如果用$0.\underbrace{11\cdots 1}_{N-1\text{个}1}$近似作为-1 的相反数，应该如何设计转化电路？




值得注意的是，不要将“2C 码取相反数”和“原码和 2C 的互相转化”弄混淆！！

另外一个重要的 2C 码运算是：符号扩展，也就是说

$$A=\underset{\text{符号位}}{\underline{a}_0}.\underline{a}_1\underline{a}_2\cdots\underline{a}_{N-1}=\underbrace{\underline{a}_0\underline{a}_0}_{\text{符号扩展2位}}.\underline{a}_1\underline{a}_2\cdots\underline{a}_{N-1}=\underbrace{\underline{a}_0\cdots\underline{a}_0}_{\text{符号扩展多位}}.\underline{a}_1\underline{a}_2\cdots\underline{a}_{N-1} \tag{23}$$

证明过程留给大家练习，符号扩展在加法或累加运算中是非常有用的。

---

两个$[-1,1)$内的 N 位 2C 数相加，所得结果为$[-2,2)$内的 N+1 位，从以下两个极端情况可以看出，

$$\begin{cases}0.\underbrace{11\cdots1}_{N-1\text{个}1}+0.\underbrace{11\cdots1}_{N-1\text{个}1} & = & 01.\underbrace{11\cdots10}_{N-2\text{个}1}\\ 1.\underbrace{00\cdots0}_{N-1\text{个}0}+1.\underbrace{00\cdots0}_{N-1\text{个}0} & = & 10.\underbrace{00\cdots00}_{N\text{-}1\text{个}0}\end{cases} \tag{24}$$

正因为两个 N 位 2C 数相加可能会得到 N+1 位的结果，如果限定结果只能为 N 位，那就意味着计算可能会溢出。

直接引用前人的结论，2C 加法的溢出：
1） 只有正数加正数可能产生正向溢出；
2） 只有负数加负数可能产生负向溢出；
3） 正数加负数，不会产生溢出。

下面分三种情况深入分析一下，为什么溢出？溢出到底意味着什么？

1） 正数加正数

$$\begin{array}{cccccccc} & & 0 & . & \underline{a}_1 & \underline{a}_2 & \cdots & \underline{a}_{N-1}\\ + & & 0 & . & \underline{b}_1 & \underline{b}_2 & \cdots & \underline{b}_{N-1}\\ \hline & s_1 & s_0 & . & \underline{p}_1 & \underline{p}_2 & \cdots & \underline{p}_{N-1}\end{array}$$

进位

图 8　　N 位 2C 正数相加，对两个操作数均进行一位符号扩展

小数点右边相加的结果可能会进位到$s_0$，但绝不会影响到$s_1$。如果$s_1s_0=00$，显然没有溢出；如果$s_1s_0=01$，结果溢出;；因为$s_1s_0=11$ or 10是不会出现的，所以暂不理会。

记住结论：$s_1s_0=00$没有溢出，$s_1s_0=01$结果溢出。

2） 负数加负数

$$
\begin{array}{cccccccc}
 & & 1 & . & \underline{a}_1 & \underline{a}_2 & \cdots & \underline{a}_{N-1} \\
+ & & 1 & . & \underline{b}_1 & \underline{b}_2 & \cdots & \underline{b}_{N-1} \\
\hline
 & s_1 & s_0 & . & \underline{p}_1 & \underline{p}_2 & \cdots & \underline{p}_{N-1}
\end{array}
$$

进位

进位

图 9　　N 位 2C 负数相加，对两个操作数均进行一位符号扩展

首先两个符号位相加结果为 10，同时小数点右边相加的结果可能会进位到$s_0$也可能不会进位：如果有向$s_0$的进位，使得$s_0=1$，也就是说结果为$11.\underline{p}_1\underline{p}_2\cdots\underline{p}_{N-1}$，根据符号扩展运算，有$11.\underline{p}_1\underline{p}_2\cdots\underline{p}_{N-1}=1.\underline{p}_1\underline{p}_2\cdots\underline{p}_{N-1}$，后者是标准 N 位 2C 表示，这么说来$s_1s_0=11$表示结果没有溢出，是正确的；如果没有向$s_0$的进位，那么结果为$10.\underline{p}_1\underline{p}_2\cdots\underline{p}_{N-1}$，该数已不能用 N 位 2C 数表示，结果溢出;；同样的$s_1s_0=00\text{ or }01$是不会出现的，暂不理会。

最终结论：$s_1s_0=11$没有溢出，$s_1s_0=10$结果溢出。

3） 正数加负数

$$
\begin{array}{cccccccc}
 & & 1 & . & \underline{a}_1 & \underline{a}_2 & \cdots & \underline{a}_{N-1} \\
+ & & 0 & . & \underline{b}_1 & \underline{b}_2 & \cdots & \underline{b}_{N-1} \\
\hline
 & s_1 & s_0 & . & \underline{p}_1 & \underline{p}_2 & \cdots & \underline{p}_{N-1}
\end{array}
$$

进位　进位

图 10　　一正一负 N 位 2C 数相加，对两个操作数均进行一位符号扩展

如果小数点右边没有向$s_0$进位，则结果为$1.\underline{p}_1\underline{p}_2\cdots\underline{p}_{N-1}$，一个正常的 N 位 2C 负数；如果小数点右边向$s_0$进位，则结果为$10.\underline{p}_1\underline{p}_2\cdots\underline{p}_{N-1}$，进行自然舍弃，得到$0.\underline{p}_1\underline{p}_2\cdots\underline{p}_{N-1}$，恰恰好$0.\underline{p}_1\underline{p}_2\cdots\underline{p}_{N-1}$就是正确的结果（为什么，请看如下证明）。

证明：从理论上分析，$A\in[-1,0]\wedge B\in[0,1)\Rightarrow(A+B)\in[-1,1)$，计算过程如下

$$
P=A+B=-1+\sum_{i=1}^{N-1}\underline{a}_i\cdot 2^{-i}+\sum_{i=1}^{N-1}\underline{b}_i\cdot 2^{-i}=-1+\underbrace{\left(\sum_{i=1}^{N-1}\underline{a}_i\cdot 2^{-i}+\sum_{i=1}^{N-1}\underline{b}_i\cdot 2^{-i}\right)}_{\text{小数点右边相加}} \tag{25}
$$

进位

小数点右边相加结果向$s_0$的进位正好抵消公式(56)的-1 项，这样 P 就是一个纯粹的正小数，所以说$0.\underline{p}_1\underline{p}_2\cdots\underline{p}_{N-1}$是正确的结果。

归纳以上三种情况：如果两个操作数同号，则$s_1s_0 = 01$结果正向溢出，$s_1s_0 = 10$结果负向溢出，$s_1s_0 = 11$ or 00结果正常；如果两个操作数异号，结果总是正确。

$$\begin{array}{rcccccc} & 0 & . & a_1 & a_2 & \cdots & a_{N-1} \\ + & 0 & . & b_1 & b_2 & \cdots & b_{N-1} \\ \hline & 0 & . & p_1 & p_2 & \cdots & p_{N-1} \end{array}$$

没进位

a) 正数加正数：正确

$$\begin{array}{rcccccc} & 0 & . & a_1 & a_2 & \cdots & a_{N-1} \\ + & 0 & . & b_1 & b_2 & \cdots & b_{N-1} \\ \hline & 1 & . & p_1 & p_2 & \cdots & p_{N-1} \end{array}$$

进位

b) 正数加正数：溢出

$$\begin{array}{rrcccccc} & & 1 & . & a_1 & a_2 & \cdots & a_{N-1} \\ + & & 1 & . & b_1 & b_2 & \cdots & b_{N-1} \\ \hline & 1 & 0 & . & p_1 & p_2 & \cdots & p_{N-1} \end{array}$$

没进位

d) 负数加负数：溢出

$$\begin{array}{rrcccccc} & & 1 & . & a_1 & a_2 & \cdots & a_{N-1} \\ + & & 1 & . & b_1 & b_2 & \cdots & b_{N-1} \\ \hline & 1 & 1 & . & p_1 & p_2 & \cdots & p_{N-1} \end{array}$$

进位

c) 负数加负数：正确

$$\begin{array}{rcccccc} & 1 & . & a_1 & a_2 & \cdots & a_{N-1} \\ + & 0 & . & b_1 & b_2 & \cdots & b_{N-1} \\ \hline & 1 & . & p_1 & p_2 & \cdots & p_{N-1} \end{array}$$

没进位

e) 负数加正数：正确

$$\begin{array}{rrcccccc} & & 1 & . & a_1 & a_2 & \cdots & a_{N-1} \\ + & & 0 & . & b_1 & b_2 & \cdots & b_{N-1} \\ \hline & 1 & 0 & . & p_1 & p_2 & \cdots & p_{N-1} \end{array}$$

进位　进位

f) 负数加正数：正确

自然舍弃，结果就正确了

根据上面的结论，可以设计一个溢出检测电路，如下

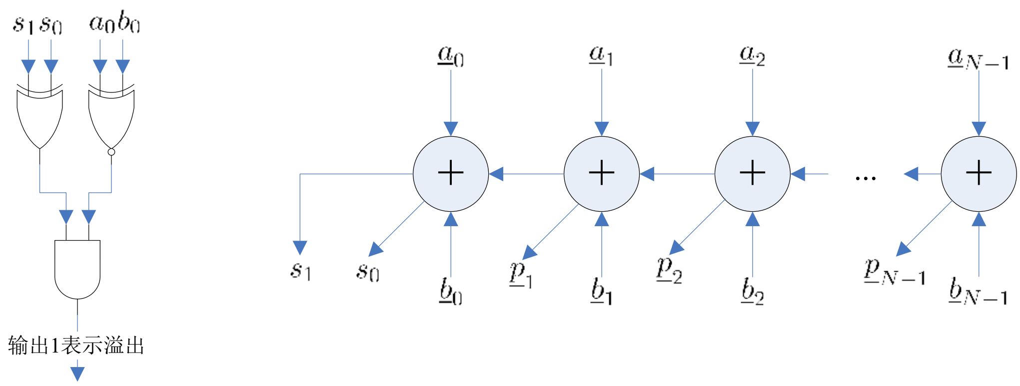



图 11　　　2C 加法溢出检测电路

从以上的分析，不难总结出一点：如果对操作数分别进行一位符号扩展后再相加，结果总是

正确的，不再出现溢出，如图 76，

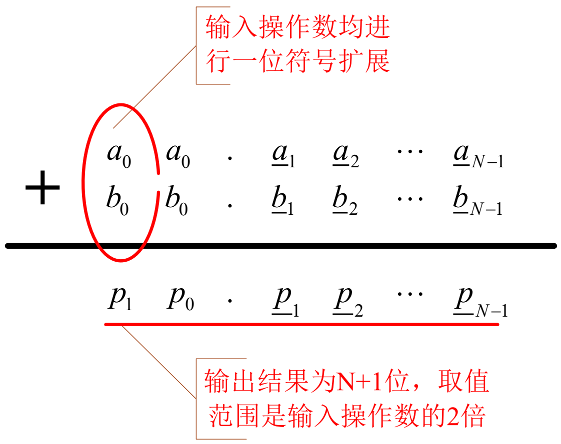



图 12　　　符号扩展一位后，能保证结果总是正确的

反过来，不对操作数进行符号扩展，而是将其取值范围限制为$\left[-\frac{1}{2},\frac{1}{2}\right)$，那么输出结果也不会发生溢出，，，这个留做习题，大家动动手！！

well，关于数的表示及其加法运算的讨论就到这，一时记不住其中结论没有关系，但是讨论中所使用的研究方式请务必理解和掌握，，很多时候我们需要自己动手寻找算法规律，找到规律硬件设计就容易了。对于位级的设计，关键就是寻找和利用运算的规律！　　　！

# 附录 B 粗粒度分段乘法器（并行/流水？）

在第十章的流水线结构的并行自适应递归滤波器设计中，涉及到多通道交织的技术，那种设计的乘法器是“变系数”流水乘法器，不同通道对应不同系数。如何设计这类乘法器呢？这里给出一种粗粒度分段乘法器的实现，作为抛砖引玉，能引导大家去思考，并给出更多更丰富多样的实现！！

以 W=16 为例，如下式

$$\begin{aligned} X \cdot C &= \left(2^8 X_1^8 + X_0^8\right)\cdot\left(2^8 C_1^8 + C_0^8\right) \\ &= 2^{16} X_1^8 C_1^8 + 2^8\left(X_0^8 C_1^8 + X_1^8 C_0^8\right) + X_0^8 C_0^8 \end{aligned} \tag{26}$$

其中，

$$\begin{cases} X &= x_{15}x_{14}\ldots x_0 \\ B &= b_{15}b_{14}\ldots b_0 \end{cases} \tag{27}$$

红色位是符号位，，还有就是

$$\begin{cases} X_1^8 &= x_{15}x_{14}\ldots x_8 \\ X_0^8 &= x_7x_6\ldots x_0 \end{cases} \wedge \begin{cases} C_1^8 &= c_{15}c_{14}\ldots c_8 \\ C_0^8 &= c_7c_6\ldots c_0 \end{cases} \tag{28}$$

下面，分别来讨论公式(57)中各个子项的计算，

1） 最简单的，无符号乘法项

$$X_0^8 C_0^8 = p_{15}^{00} p_{14}^{00} \dots p_0^{00} \tag{29}$$

2） 有符号乘法项 I

$$\begin{cases} X_1^8 C_0^8 & = & p_{15}^{10} p_{14}^{10} \dots p_0^{10} \\ X_0^8 C_1^8 & = & p_{15}^{01} p_{14}^{01} \dots p_0^{01} \end{cases} \tag{30}$$

3） 有符号乘法项 II

$$X_1^8 C_1^8 = p_{14}^{11} p_{13}^{11} \dots p_0^{11} \tag{31}$$

计算流程，根据公式(57)/(60)/(62)，，我们可以不用任何代价就能进行第一项和第三项的相加（其实是一种 Verilog 的拼接操作，不占任何资源）

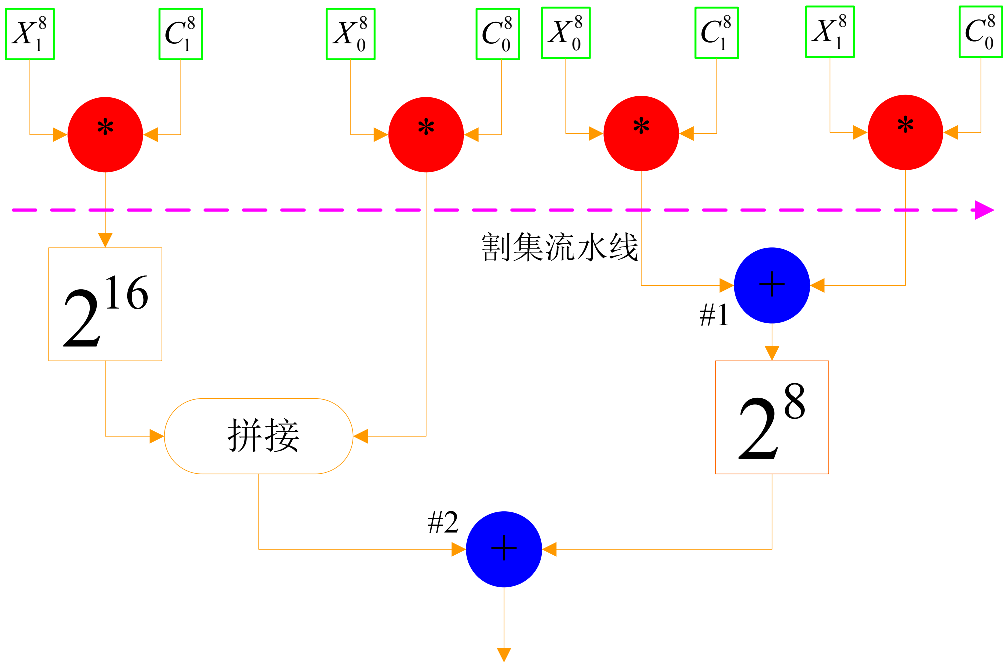



图 13 分段乘法器框架，图中加法单元进行的是 2C 数加法

关于流水线乘法器的方案还有很多，这里给出的只是一种最容易想到的实现！