诡异的位串行乘法和位串行滤波器设计（直接由字级结构导出位串行结构）

下面将迈入基于 Horner 法则的位串行乘法器设计，以及，位串行滤波器设计！之所以把这两个内容放到一起，是因为他们的设计原理是相同的：**“左移和右移的抵消”**。也许现在你看不懂这个“莫名其妙”的原理，不过没有关系，我会慢慢展示给你看！

为了能把握问题本质，我们从无符号数的 Horner 位串行乘法器开始设计，，之后再转入 2C 位串行乘法器设计，最后比较这两种设计的区别，弄清楚区别产生的原因所在。

无符号数乘法的 Horner 公式如下

$$\begin{aligned} A\cdot B &= A\cdot\left(b_0\cdot 2^0+\underline{b}_1\cdot 2^{-1}+\underline{b}_2\cdot 2^{-2}+\underline{b}_3\cdot 2^{-3}\right) \\ &= A\cdot\left(b_0+\left(\underline{b}_1+\left(\underline{b}_2+\underline{b}_3\cdot 2^{-1}\right)\cdot 2^{-1}\right)\cdot 2^{-1}\right) \\ &= A\cdot b_0+\left(A\cdot\underline{b}_1+\left(A\cdot\underline{b}_2+A\cdot\underline{b}_3\cdot 2^{-1}\right)\cdot 2^{-1}\right)\cdot 2^{-1} \end{aligned} \tag{1}$$

根据公式(21)，画出一个系统框图如下，

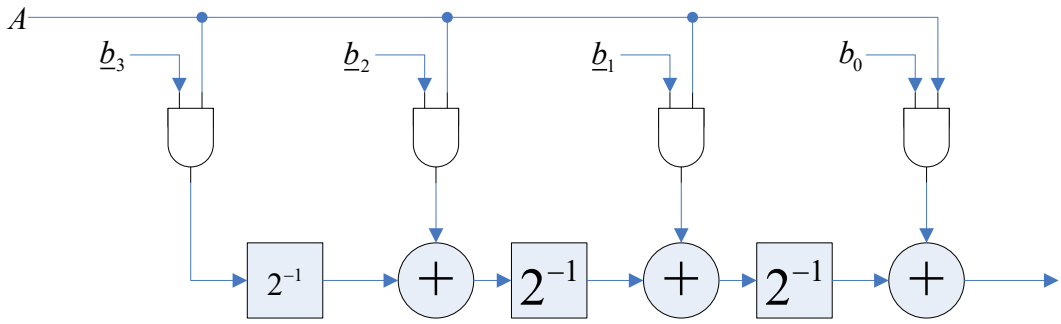


图 1基于 Horner 法则给出的电路雏形

注意，图 33 并非是最终的位串行乘法器电路，仅仅只是一个电路雏形，离位串行结构还有“十万八千里”。。细心的同学可以发现，图 33 的输入端是 A，而不是$a_0\underline{a}_1\underline{a}_2\underline{a}_3$，就是想告诉大家，这还不是位串行电路。那么能不能由图 33 的雏形电路设计出真正的位串行电路呢？

——要做到这一点，得真正开动脑筋了，，下面的过程也许很繁琐，但这就是我一开始的思考过程，

首先考察一下简单的情形，

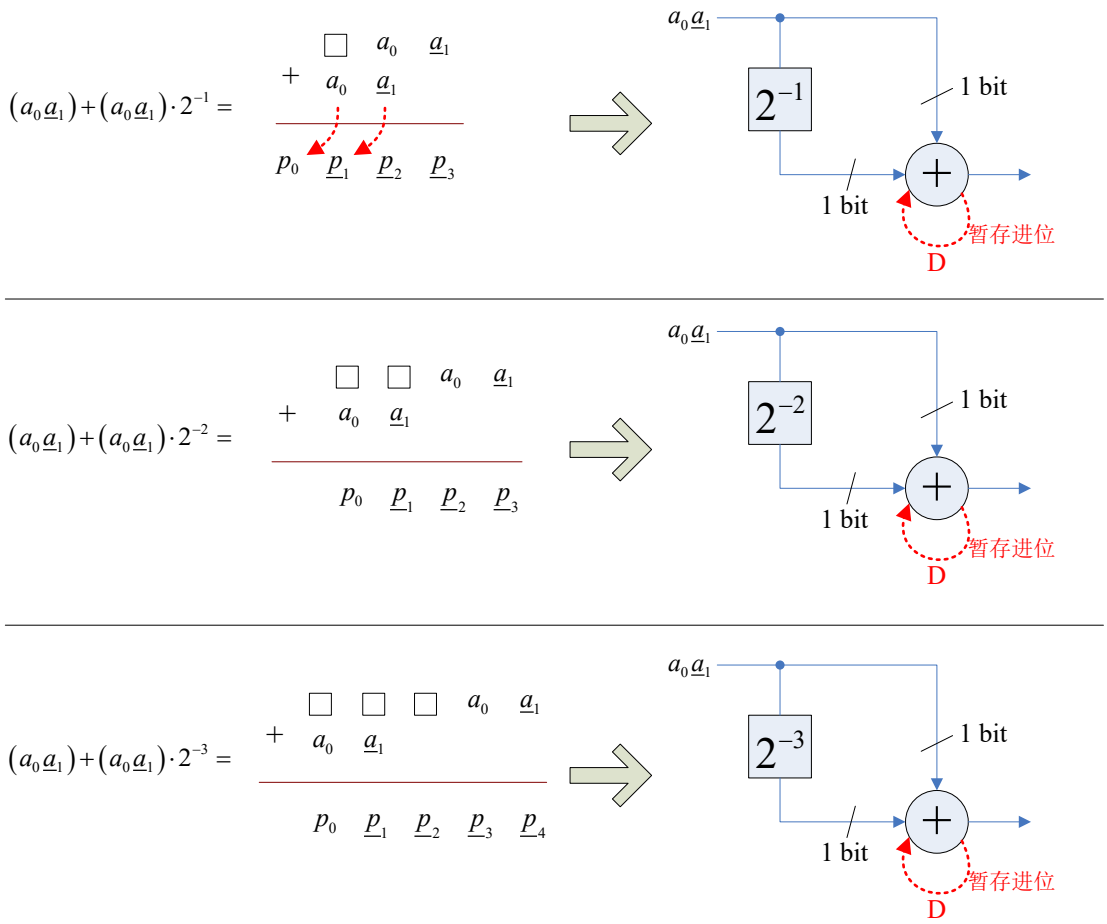


图 2几个简单的位串行设计方案，可能的方案

仔细在想想图 34 电路的工作情况，可以发现，根本不可实现。问题就处在右移单元，而且是一个组合逻辑的右移单元（？），所需要的功能是

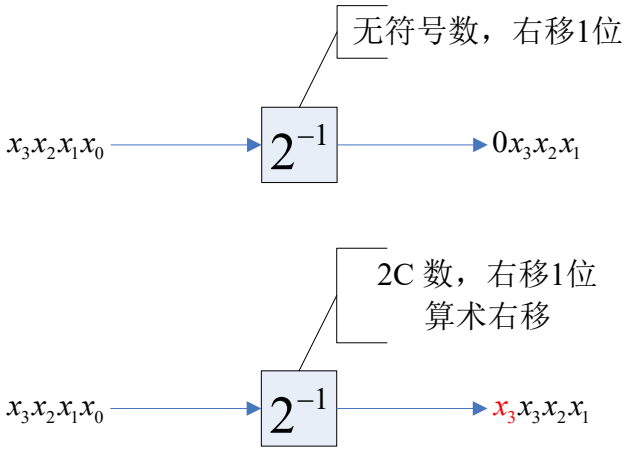


图 3右移单元的功能示意

对于无符号数，右移之后最高位补零；对于 2C 数，右移之后最高位进行符号扩展。如果想用组合逻辑实现图 35 的功能，是做不到的，因为图 35 所示为非因果系统，也就是输入x_0时右移单元必须给出x_1，可是此时x_1还未输入不可能提前输出来。。

我们再来看看右移单元的反面，左移单元：你知道左移单元实际是什么东西吗？

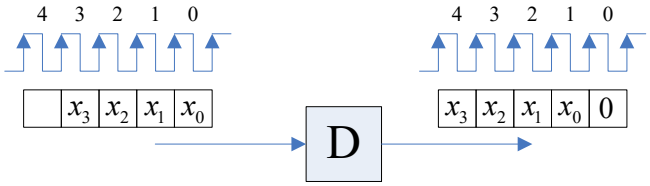


图 4左移单元

所谓的（无符号）左移单元其实就是一个延时器，从图 36 可以看出，输入到输出的确被左移了一位，当然了，延时器的初始值应该为零。

前面说到，图 34 的电路是不照的，比如计算$(a_0\underline{a}_1)+(a_0\underline{a}_1)\cdot 2^{-1}$时，功能上要求

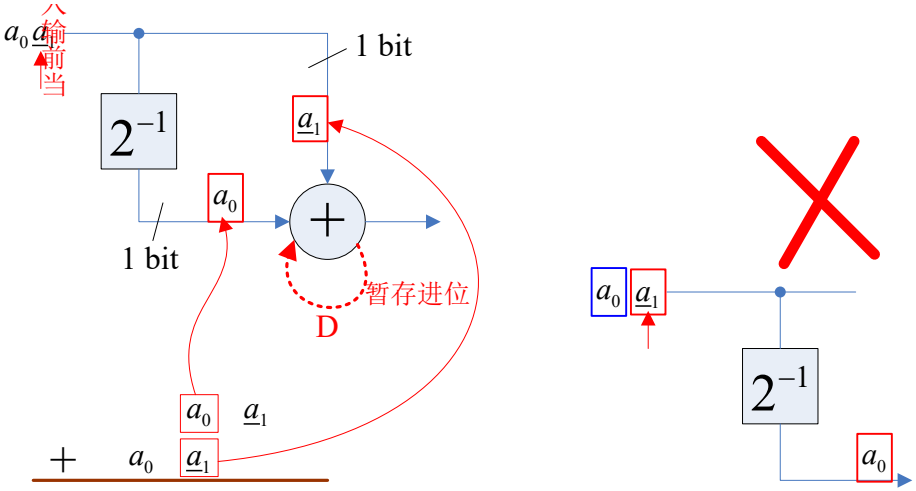
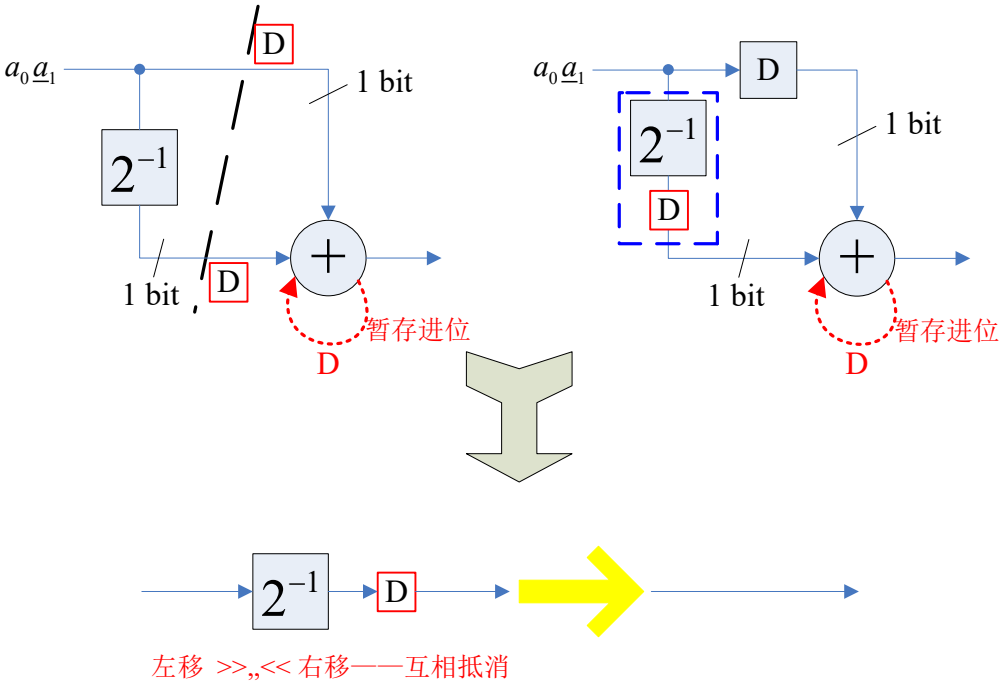


图 5非因果系统，不可实现

图 37 的右移功能单元是实现不了，除非你能把“它”变为因果的，，最简单的方法就是对系统进行延时，也就是



左移 >>„<< 右移——互相抵消

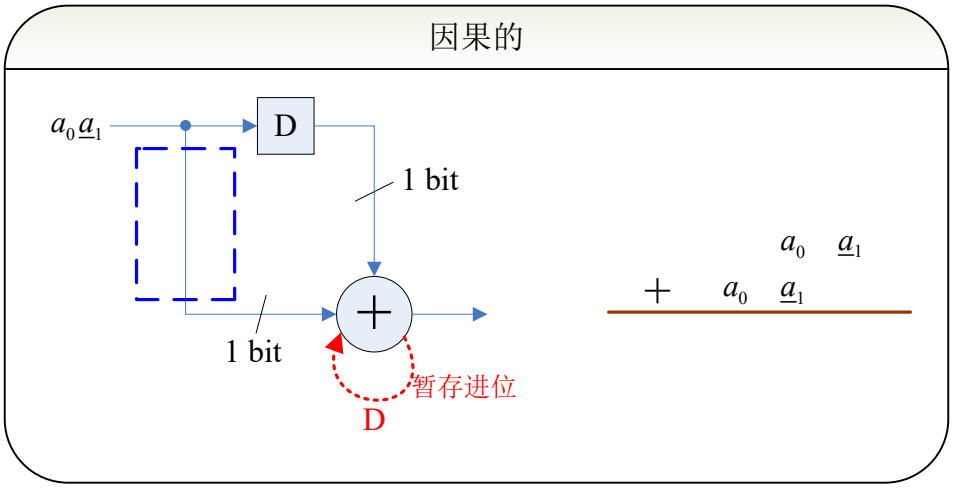


图 6延时得到因果系统（实际上是插入割集流水线）

还记得前面说到的设计原则没：**“左移和右移的抵消”。**这一段看起来并不非常严谨，但是结果的确是对的，大家可以验证一下图 38 的系统。对于图 34 的其他情况有，

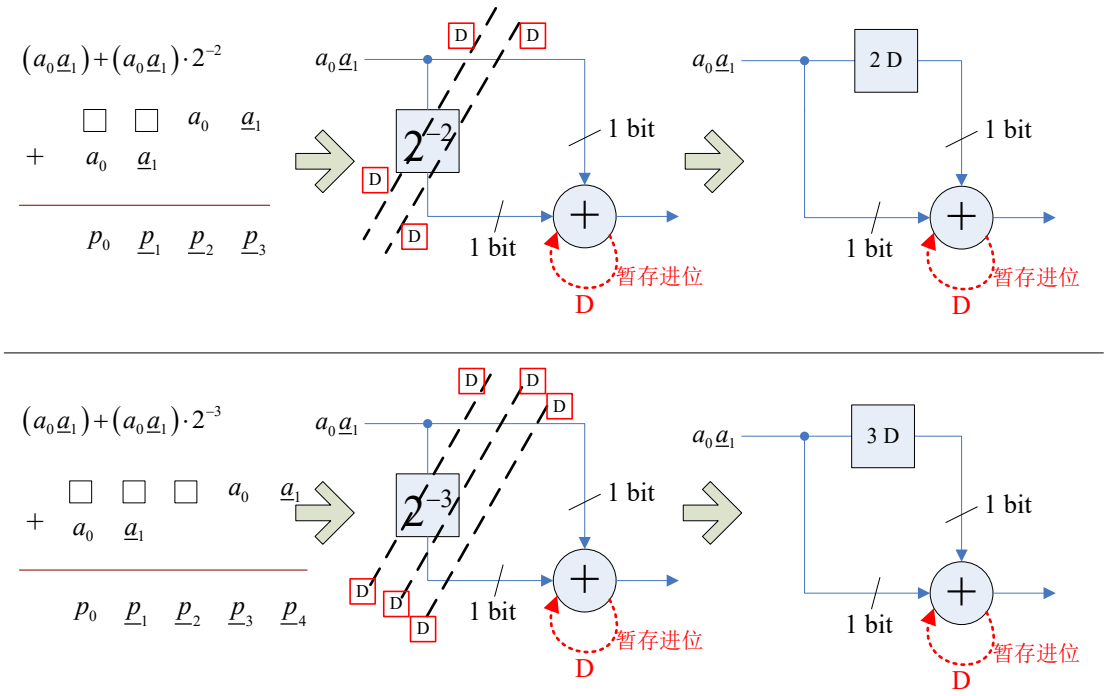


图 7更多无符号数位串行设计例子

大家可以体会体会这种设计的原理，为什么可以这么做？如果你始终不能理解，那也没关系，请记住一点：**“左移和右移的抵消”。**在无符号位串行的设计中，其实就是用一个延时单元（左移）来抵消左移一位（2^{-1}），用两个延时单元抵消左移两位，三个延时抵消左移三位，，，等等如此。

练习题：请把图 39 电路的运行时的中间过程手动在草稿纸上“仿真”一遍！

对于 2C 数的位串行设计仍然遵循**“左移和右移的抵消”，**但最终电路稍微变化，如下图

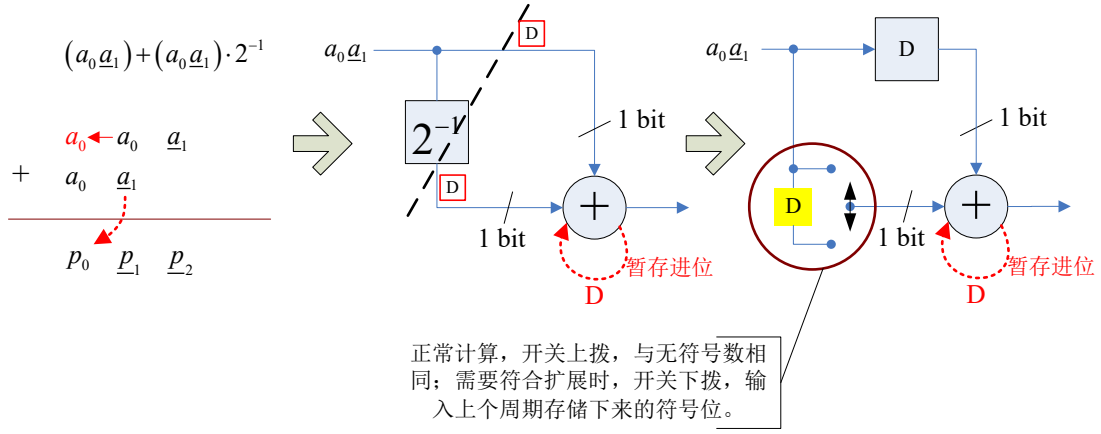


图 82C 数的位串行设计电路，稍区别无符号数的位串行设计

因为 2C 数进行部分积累加时，需要符号对齐，也就是进行符号扩展操作，因此最终电路中要多添加一个寄存器单元，用于存储符号位，以便于在需要时输入。正常计算开关上拨，只有在需要符合位扩展的那个周期，开关下拨，从而输入上个周期存储下来的符号位，实现符

号扩展功能，开关的状态有专门的状态机控制。图中不标出具体开关的时刻，因为这个和具体设计有关，需要根据实际情况来确定。

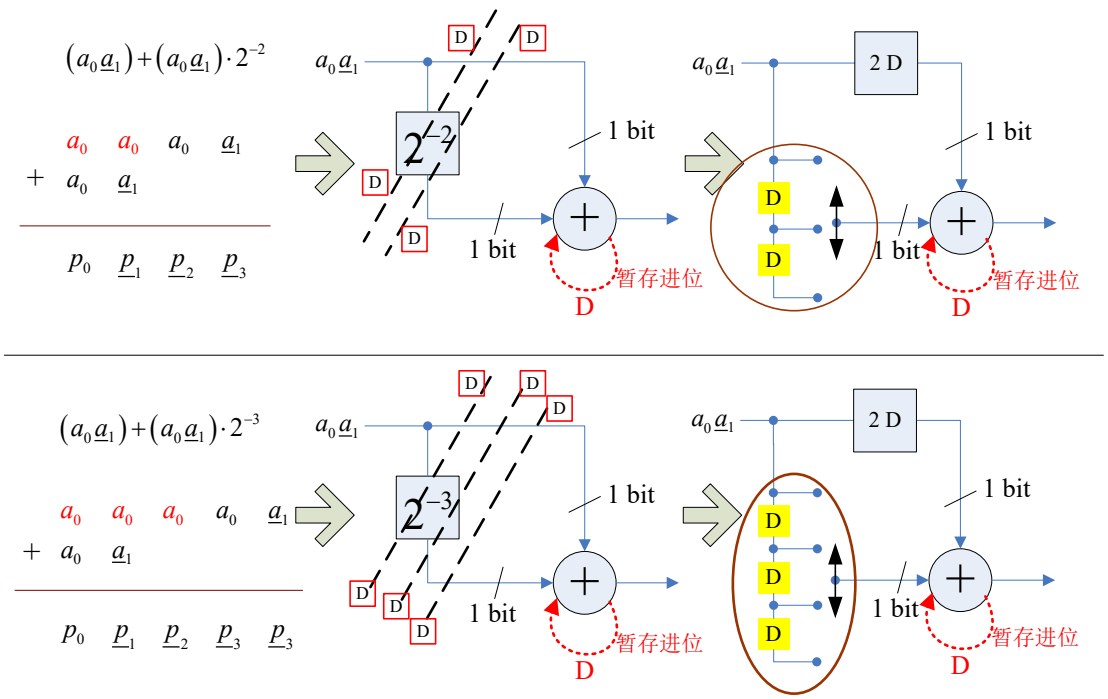


图 9更多 2C 数位串行设计例子

图 41 给出了其他两个 2C 数位串行电路设计的例子，当然了，牛人总是会挑电路的毛病，，注意观察图 41 的电路，符号扩展功能其实可以设计得更为简洁，以便节约寄存器，所带来的坏处可能就是控制逻辑要复杂一些，，

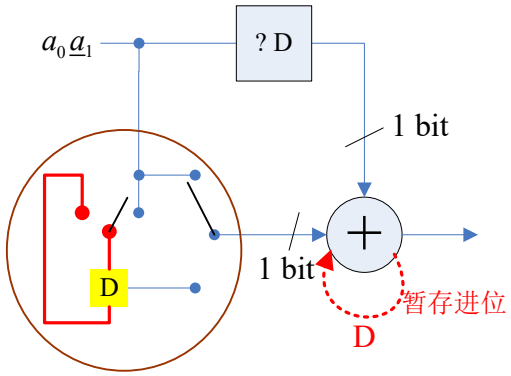


图 10　符号位一出现就接通寄存器进行保存，之后寄存器形成自循环路径（红线）

符号位到来时，接通寄存器进行保存，之后寄存器接成自循环路径（红线所示），这样就能多次使用保留下来的符号位，而不必用多个寄存器来保存。

题外话，图 42 的策略有助于减少功耗，不仅仅是符号寄存器的个数变少，而且其翻转次数也减少了。

明白了以上的设计原则，设计 Lyon 乘法器就容易了，直接看图（4x4 Lyon's multiplier），

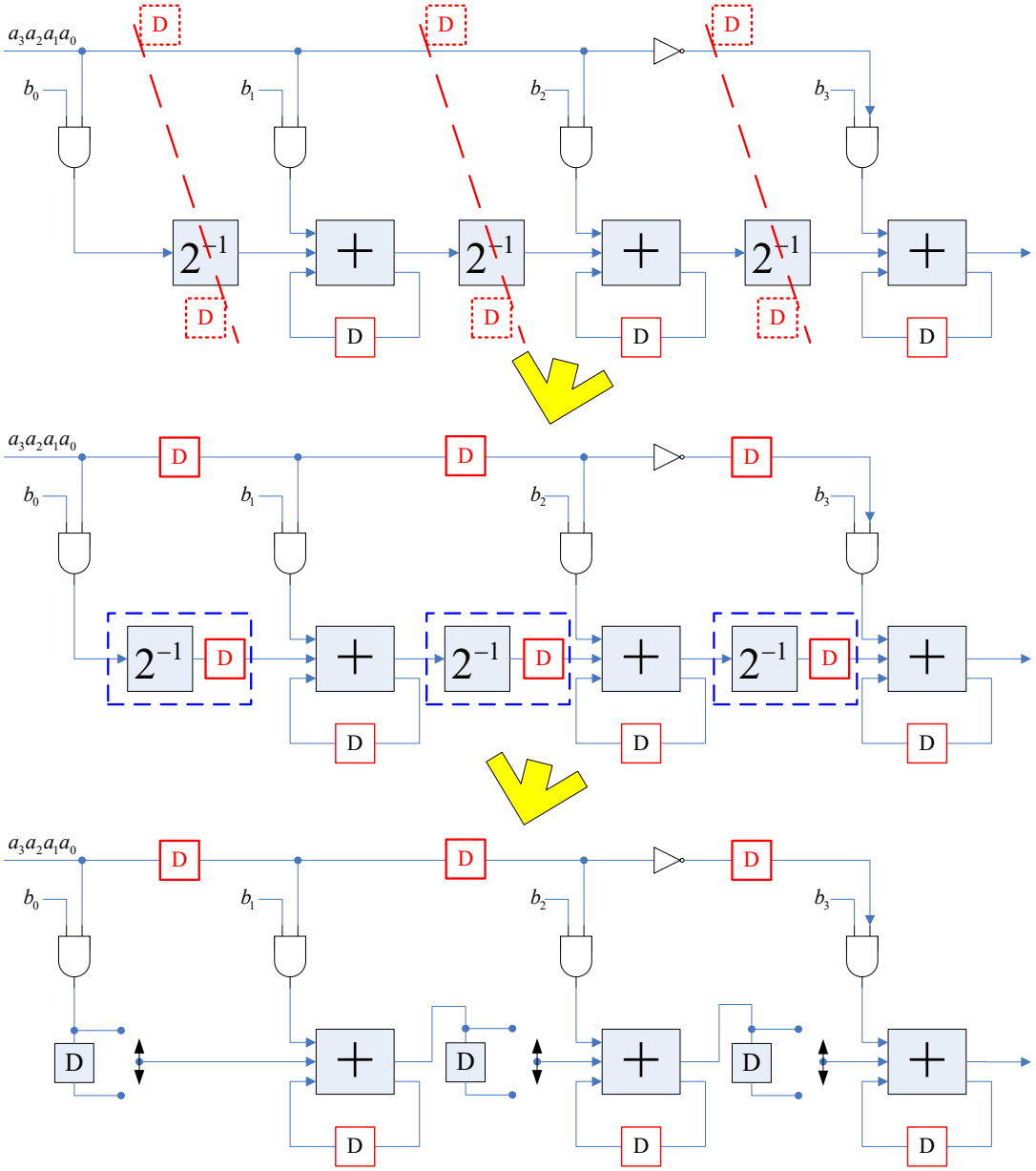


图 11　Lyon's multiplier

图 43 给出 Lyon's multiplier 的设计过程，注意，这只是大致的设计电路，实用化还需进行一些细微的补充：我想这对大家来说不成问题！

上面所讨论的设计方法强大之处不仅仅在于设计一个小小的乘法器，而且能设计各种 FIR 和 IIR 的位串行结构，，下面仍以例子进行学习！

例题 1、FIR 位串行架构设计，假设 FIR 迭代公式如下

$$y(n) = -\frac{7}{8}x(n) + \frac{1}{2}x(n-1) \tag{2}$$

对于定点计算系统，总是能使用移位和加法来代替乘法，所以设计之前要先将迭代式的乘法转化为移位和加法（CSD 编码以及子表达式共享有助于得到更省资源的实现：以后会讨论

到），如公式(23)

$$y(n) = -x(n) + \frac{1}{8}x(n) + \frac{1}{2}x(n-1) \tag{3}$$

字级电路如图 44，

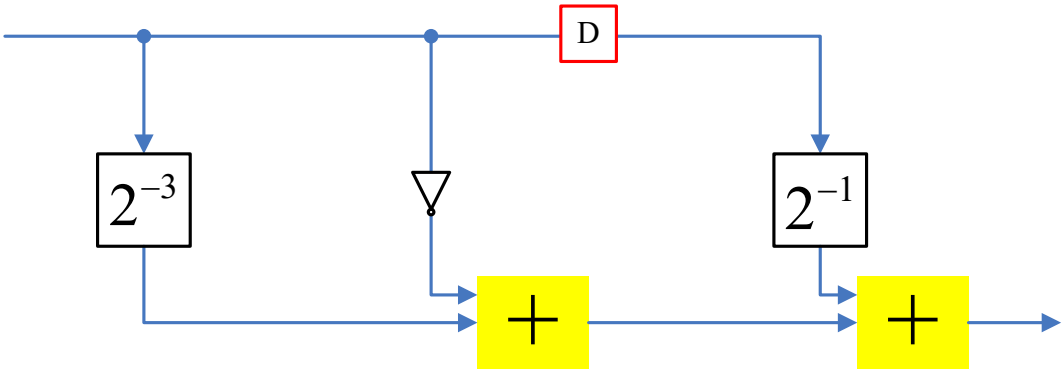


图 12　基于移位和加法器的 FIR 字级电路

这里给出与课本上稍有不同的设计结果，这里输入输出字长均为 W=8。。还有一点值得注意的是图 44 中的延时，指的是字级的延时，转化到位级相当于 W 个延时，所以有，

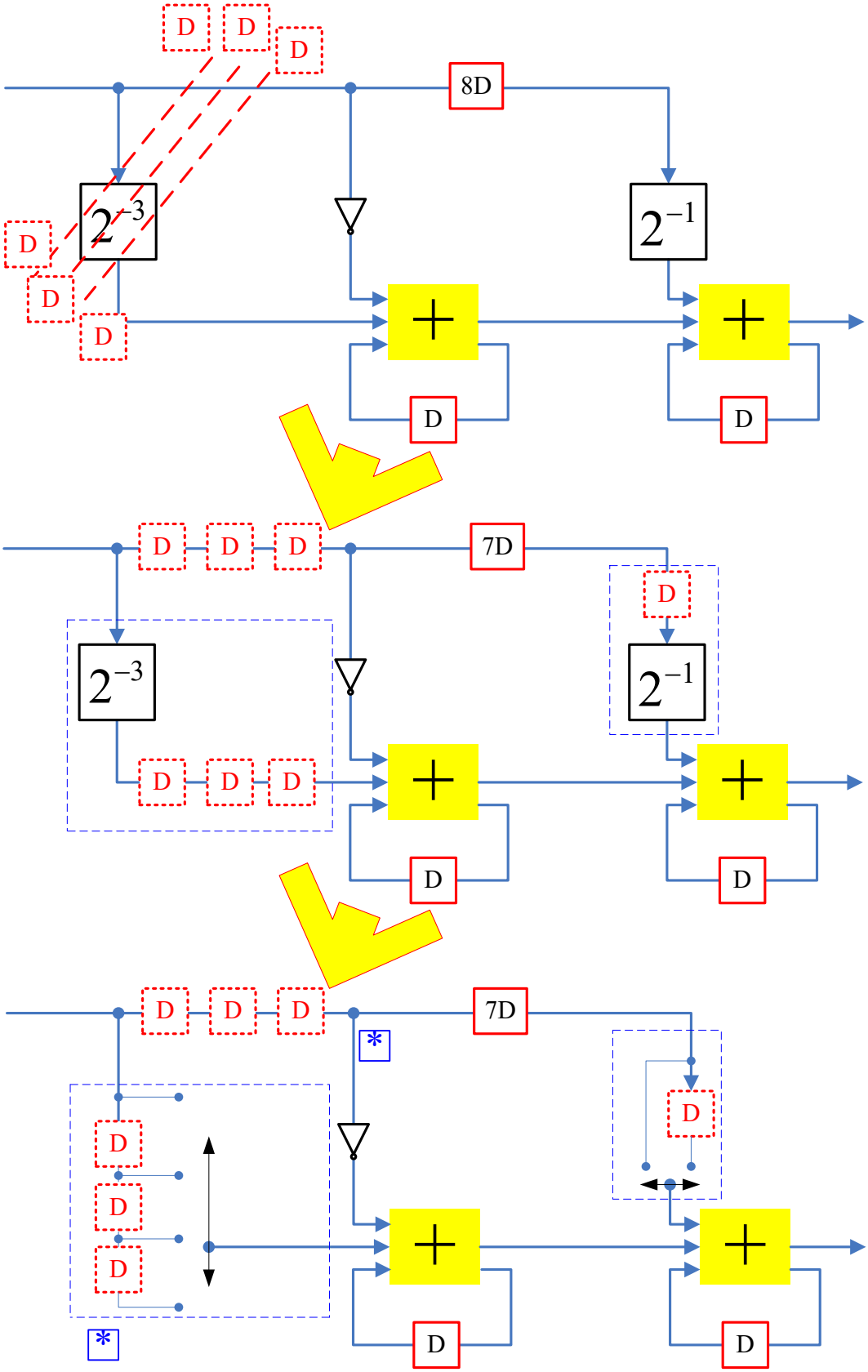


图 13　可实现的位级 FIR 流水线架构

注意到图 45 最后电路的蓝色星号*了吗，这两个节点其实可以合并，这样就能节约 3 个寄

存器单元，最终电路如图 46，

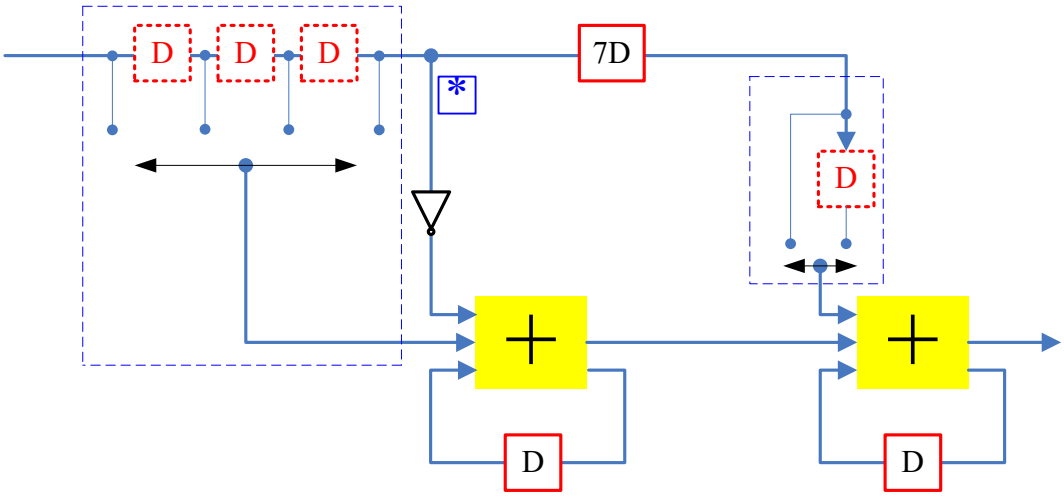


图 14　合并冗余节点，节约寄存器资源

再来看一个 IIR 的例子，课本上对于 IIR 位串行架构的设计引出两个概念：

1） 环路延时同步；

2） 补偿（或 歪斜）。

联系第 10 章，流水线结构的并行自适应递归滤波器，很容易理解这一点：因为 IIR 滤波器的环路可不像 FIR 的前向路径，能找到前馈割集插入任意多的流水线寄存器，，但也不是说 IIR 就不能用多级流水线。要对 IIR 的环路进行流水化，“只能”采用超前计算或者 M 倍降速。

课本上默认并不对 IIR 进行超前处理或者 M 倍降速，也就是说 IIR 环路的延时总数是定值，不可改变。假设字级架构的 IIR 某个环路具有N_D个延时，转化到位级架构就变为$W \cdot N_D$个延时，W 是字长，这个数目是固定不可变的。

——如果“你”在设计 IIR 位级架构的时候，像课本一样将 FIR 部分单独拿出来处理，最后在连接起来，那么就必须保证你所设计的 IIR 位串行架构的每一个环路延时数目匹配到规定的数目。

此外，还有补偿寄存器的数目，，课本上给出了一套法则，经过推导我承认这是对的。注意：课本上默认大家都能理解，所以也没做太多解释。。

关于课本上的 IIR 位级架构设计方法，有点让人混乱，而且摸不着头脑，因此，现在我建议：如果你觉得掌握不了，那就先别采用课本的方法；当然作为补偿，我自己给出一套设计方法（OH，希望我的方法是完备的!! 大家批判的学习学习，错误我可不会负责；另外，当你看懂了下面将要介绍的方法，很容易就能自己推导出课本上的结论，以及作者没有说清楚的细节），，

例题 2、IIR 位串行架构设计，假设 IIR 迭代公式如 下

$$y(n) = -\frac{7}{8}y(n-1) + \frac{1}{2}y(n-2) + x(n) \tag{4}$$

转化为移位和加法的实现，有

$$y(n) = -y(n-1) + \frac{1}{8}y(n-1) + \frac{1}{2}y(n-2) + x(n) \tag{5}$$

字级电路如下，

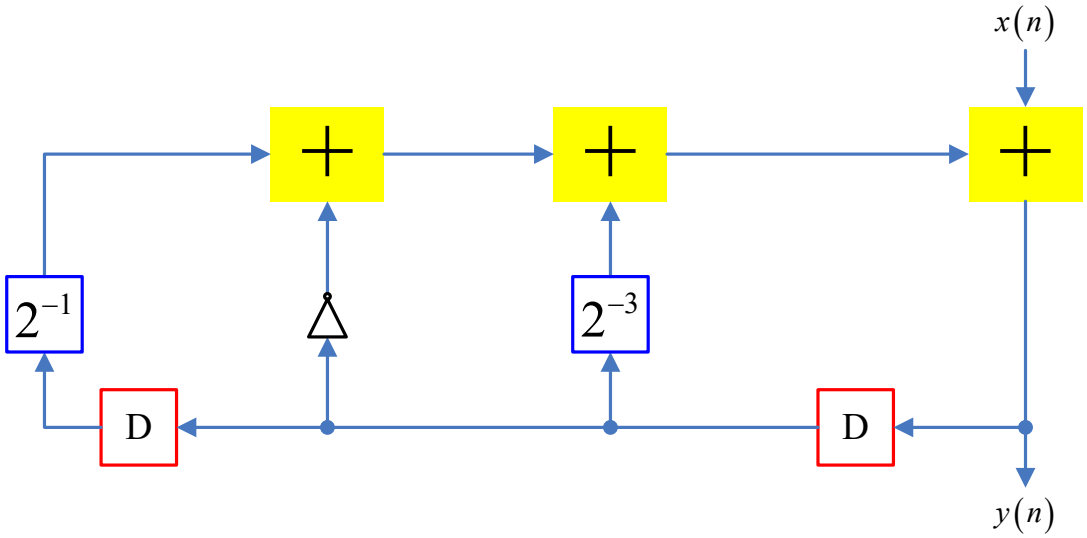


图 15　基于移位和加法器的 IIR 字级电路

将图 47 转为位级架构，并用重定时移动恰当数目延时与右移单元相抵消，具体过程如图 48

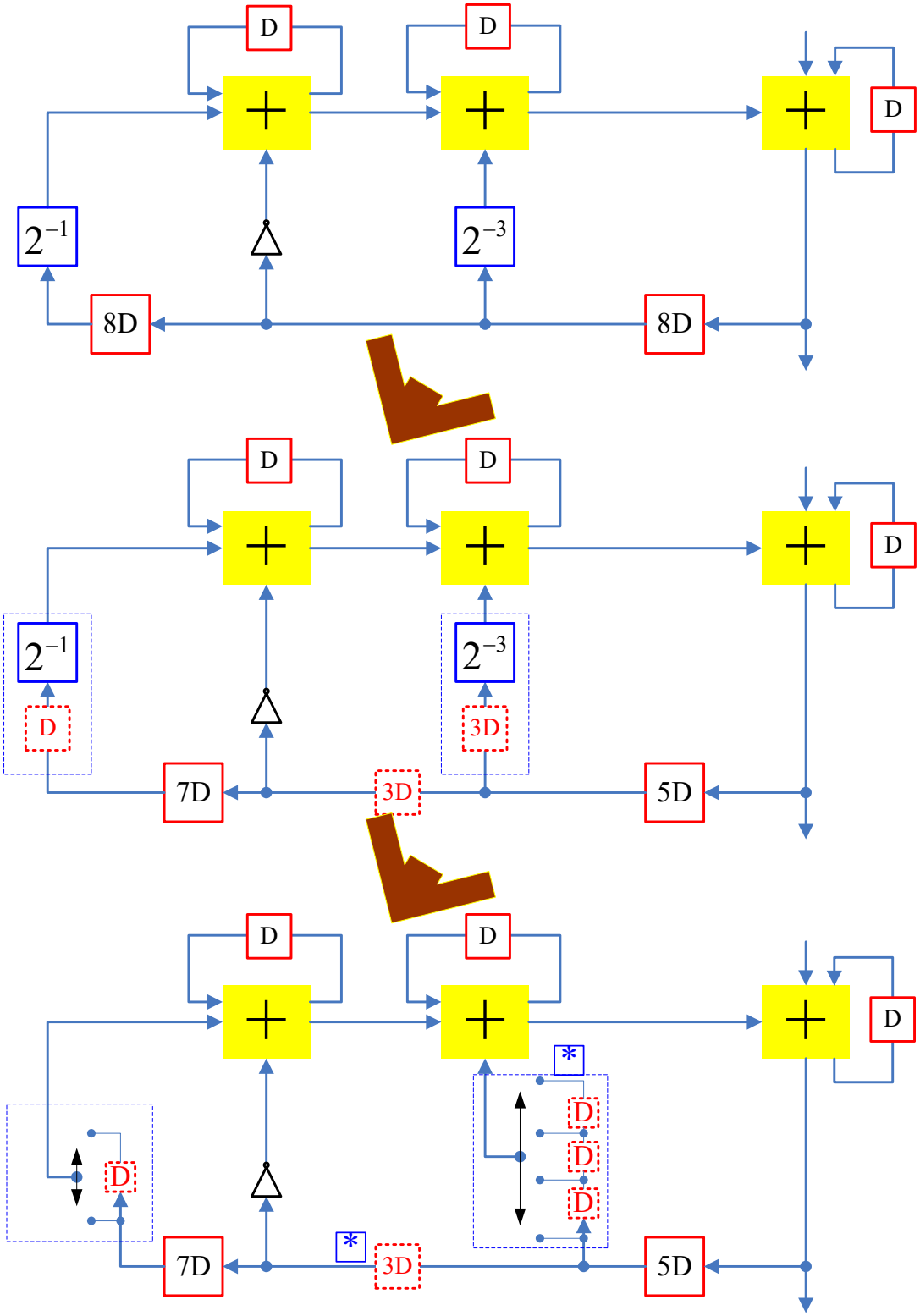


图 16　使用重定时构造的 IIR 位串行电路

同样的，图 48 最后电路也可以进行节点共享，最终可得，

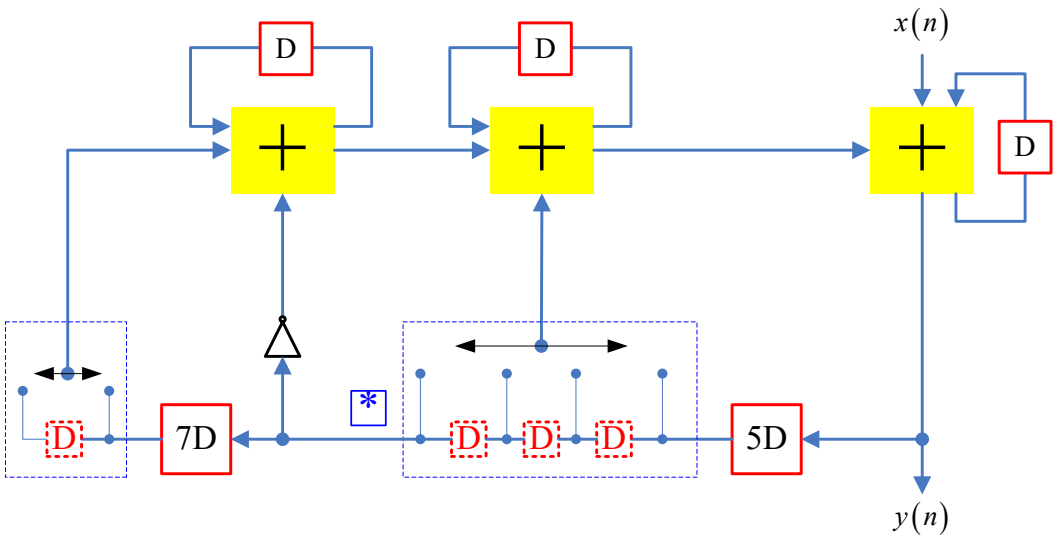


图 17　合并冗余节点，节约寄存器资源

这里所得的结构和课本给出的结构功能“一致”，图 50 对课本结果进行恰当的重定时处理，可以看出最终结果“正是”图 49 的电路。这就说明，我们的设计方法和课本一致!!

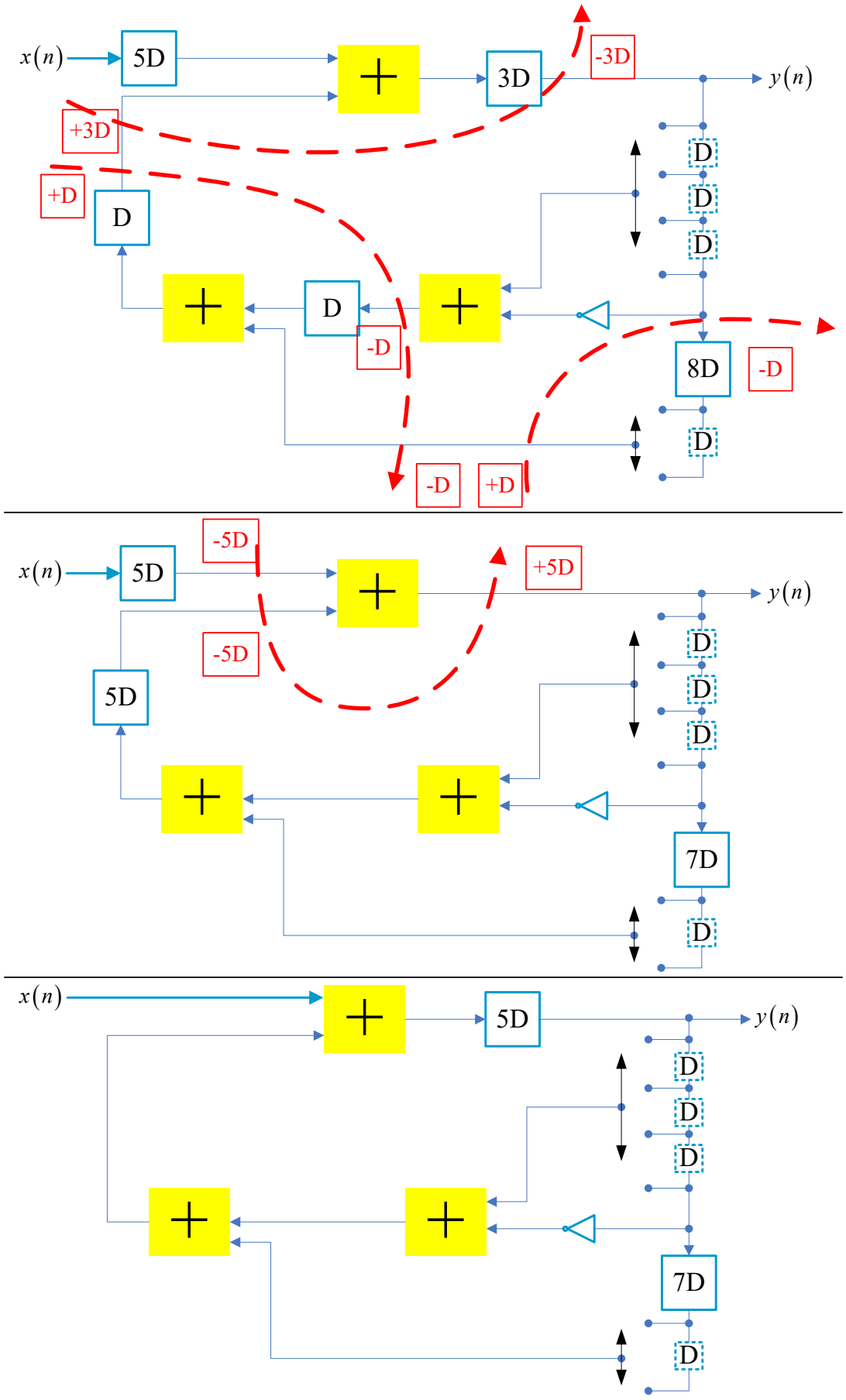

x(n)
5D
3D
-3D
y(n)
+3D
+D
D
D
D
D
D
-D
8D
-D
D
-D
+D
-5D
5D
+5D
-5D
5D
D
D
D
7D
D
x(n)
5D
y(n)
D
D
D
7D
D

图 18　对课本结果进行重定时优化，注意：图中加法器的进位寄存器没画出

再来看看课本上的例 13.5.1,，迭代公式如下

$$y(n) = -\frac{7}{32}y(n-1) + \frac{3}{4}y(n-2) + x(n) \tag{6}$$

转化为移位和加法实现，

$$y(n) = -\frac{1}{4}y(n-1) + \frac{1}{32}y(n-1) + y(n-2) - \frac{1}{4}y(n-2) + x(n) \tag{7}$$

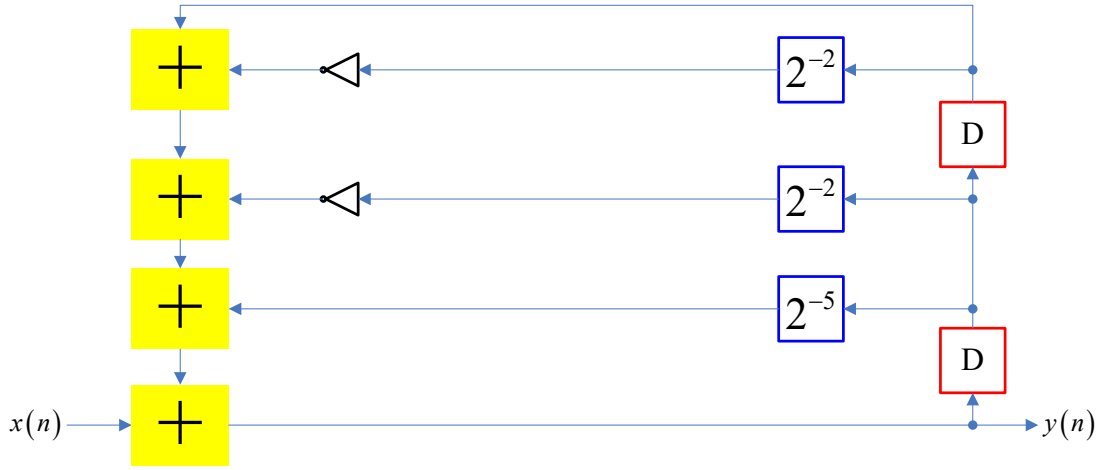


图 19　基于移位和加法器的 IIR 字级电路

请注意，图 51 的画图方法，两条准则：1）反相器位于右移单元之后；2）同一变量的不同右移支路，右移数目多的支路在前，右移数目少的支路在后，比如对 y(n-1)的两条右移支路，2^{-5}的支路在前面，而2^{-2}的支路在后，又比如对 y(n-2)的两条支路，2^{-2}的支路在前，2^{0}支路在后。。按照以上的准则画图，能容易观察出可以合并的节点，从而节约寄存器的使用。。请体会以下的设计过程，

先将图 51 字级电路转化为位级电路（加法器的进位寄存器不再画出！你可以自己补上），

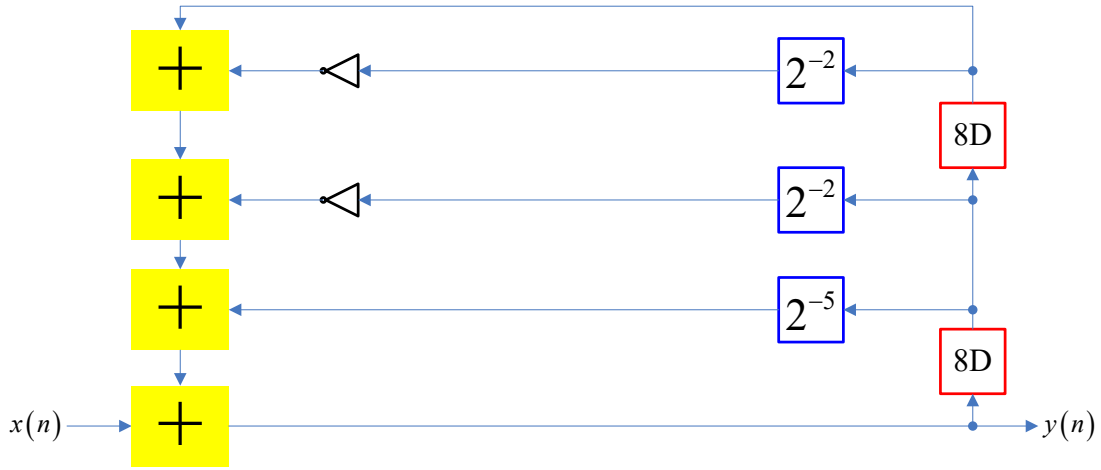


图 20　基于移位和加法器的 IIR 位级电路（加法器的进位寄存器没有画出，可自行补上）

接下来对图 52 位级电路进行恰当重定时，并导出最终位级电路，如下图

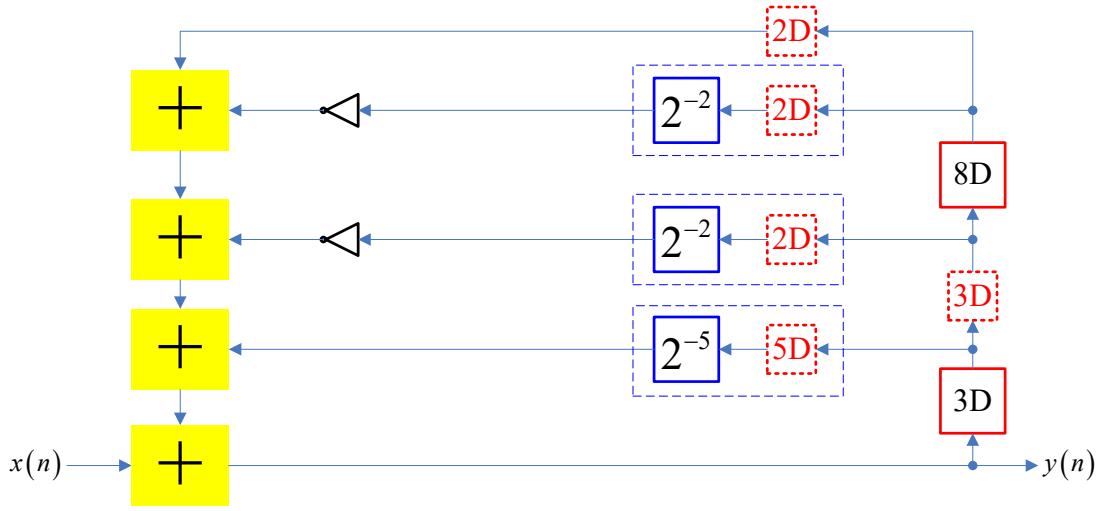


图 21　设计：第一部分，重定时

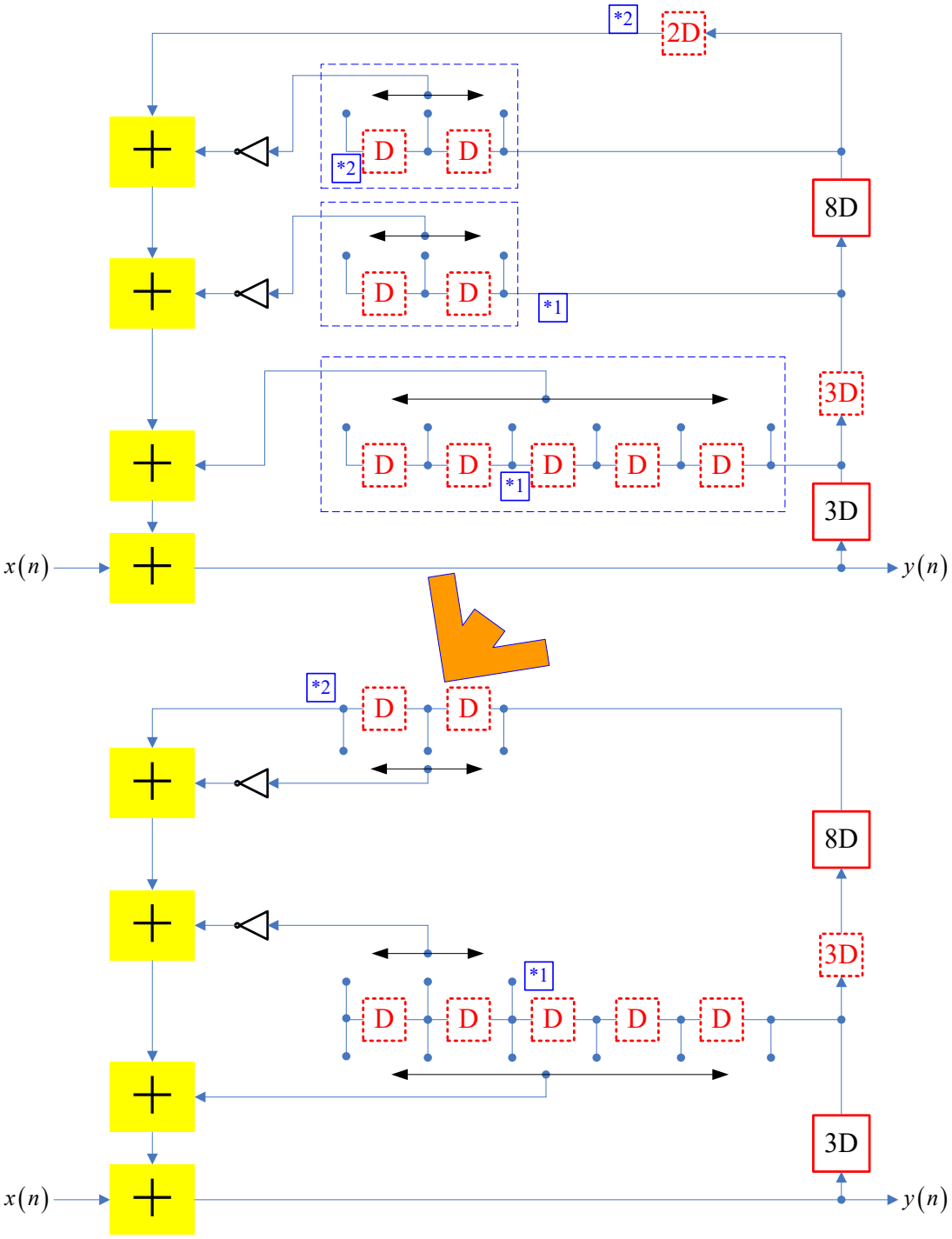


图 22　设计：第二部分，合并节点

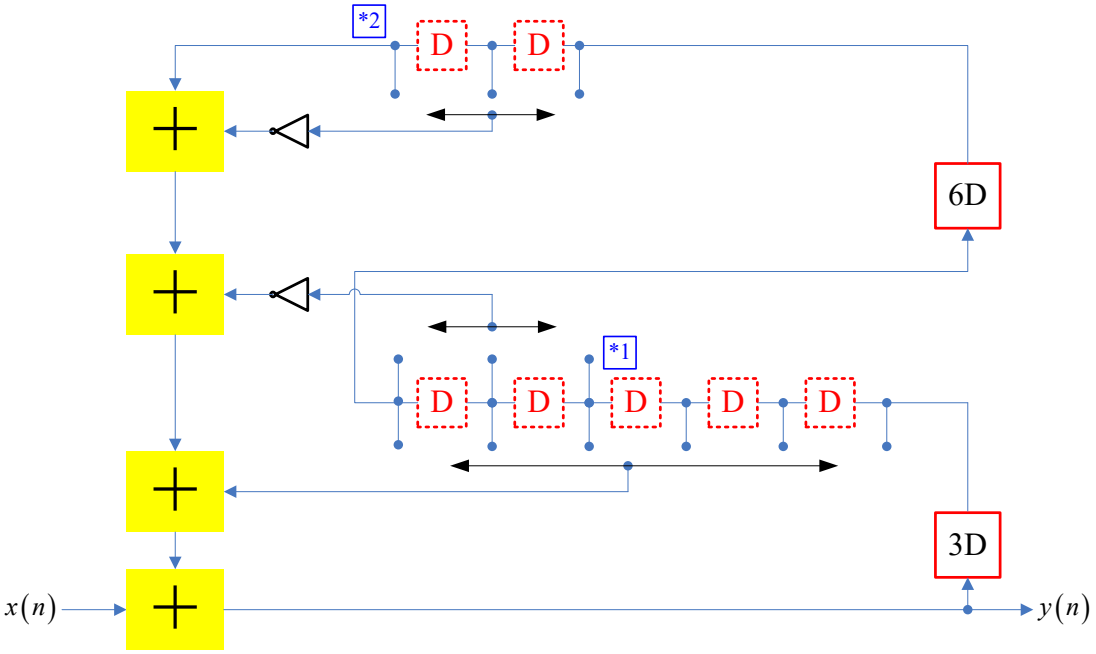


图 23　更进一步的化简

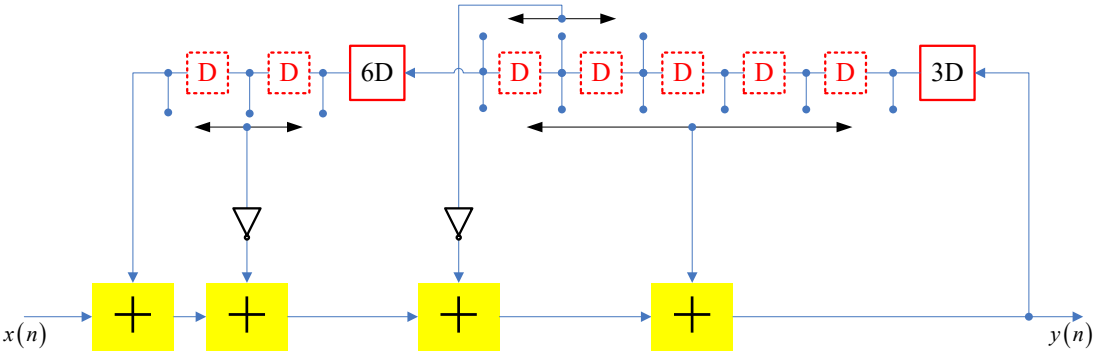


图 24　另一种画法，一眼就能知道为什么寄存器资源可以共享!!（推荐）

图 55 看起来还不是非常直观，而图 56 就非常明了。。如果我们熟练了，以后在进行 IIR 位串行架构设计，直接就能画出图 56 的结构，直接明了。

思考题：会不会出现寄存器不够抵消右移操作的情况呢，如果出现应该如何处理？如果不会出现，请证明之!!

正则符号数 CSD 运算

一般滤波器电路中使用的都是常系数乘法器，也就是说其中一个乘数是固定的常数。类似于 BOOTH 重编码思路，对乘法器的常系数进行重新编码，使得其中非零位最少，将有利于同时减少面积和功耗。

——正则符号数就是这么一种编码，使得常系数的非零位最少，下面通过几个实例先来体验一下，注意比较同一常数的 2C 编码和 CSD 编码非零位数目，思考一下：为什么非零位数

目少就意味着实现面积小和低功耗？

	2C编码	2C非零位（个）	CSD编码	CSD非零位（个）
7 ：	00111 $2^2+2^1+2^0$	3	$0100\bar{1}$ 2^3-2^0	2
-6 ：	11010 $-2^4+2^3+2^1$	3	$0\bar{1}010$ -2^3+2^1	2
15 ：	01111 $2^3+2^2+2^1+2^0$	4	$1000\bar{1}$ 2^4-2^0	2
-9 ：	10111 $-2^4+2^2+2^1+2^0$	4	$00\bar{1}0\bar{1}$ -2^3-2^1	2

图 25　几个 CSD 的编码例子，其中$\bar{1}$表示-1

通过对比图 57 同一个常数 2C 编码和 CSD 编码可知，CSD 编码使用更少的非零比特位，另外一个区别是，CSD 允许使用$\{-1,0,1\}$位集合，而 2C 编码只能使用$\{0,1\}$位集合。其实 CSD 可以看作是二进制编码的一个扩展，如下

正规二进制编码（无符号数）定义公式为

$$A=(a_{N-1}\cdots a_1a_0)_{binary}=\sum_{i=0}^{N-1}a_i\cdot 2^i,\text{while},a_i\in\{0,1\} \tag{8}$$

而 CSD 编码扩展了a_i的取值集合，定义公式如下

$$A=(a_{N-1}\cdots a_1a_0)_{CSD}=\sum_{i=0}^{N-1}a_i\cdot 2^i,\text{while},a_i\in\{-1,0,1\} \tag{9}$$

但，不是每一个由公式(29)得出的编码都是 CSD 编码，还必须满足一下特征要求，

- CSD 数中不存在两个连续的非零位；
- 一个数的 CSD 表示中含有的非零位是所有表示中最少的，因而被称为是正则的；
- 一个数的 CSD 表示是唯一的；
- 对于公式$(a_0.\underline{a}_1\cdots\underline{a}_{W-1})_{CSD}=a_0+\sum_{i=1}^{W-1}\underline{a}_i\cdot 2^{-i},\text{while},a_0,\underline{a}_i\in\{-1,0,1\}$所定义的 CSD 数覆盖区间$\left(-\frac{4}{3},\frac{4}{3}\right)$，其中最有价值的区间是$[-1,1)$；
- 在区间$[-1,1)$的 W 位 CSD 数中，非零比特位数比 2C 编码的非零比特位数少 33%。。

一般情况下，我们只需知道如何将一个实数或者 2C 编码的串转化为 CSD 编码，就能用到实际中，可以不必太纠缠于 CSD 的一些理论讨论。。下面给出课本上 CSD 编码的程序实现，

代码 1　课本所给的 CSD 编码程序实现

```
%
function [digits,weight,err] = csdigit(A,resolution)

s   = sign(A);
ah  = ['0',dec2bin(round(abs(A)*2^resolution))];
W   = length(ah);

%   from VLSI-DSP,,2c to csd
ah  = fliplr([ah(1),ah,'0']-'0');
r   = ah;
a   = zeros(1,W);

for k =1:W
    r(k+1)  = not(r(k))*xor(ah(k+1),ah(k));
    a(k)    = (1-2*ah(k+2))*r(k+1);
end

digits  = s*fliplr(a);
weight  = (W-1:-1:0)-resolution;
err     = A-sum(digits.*2.^weight);
```

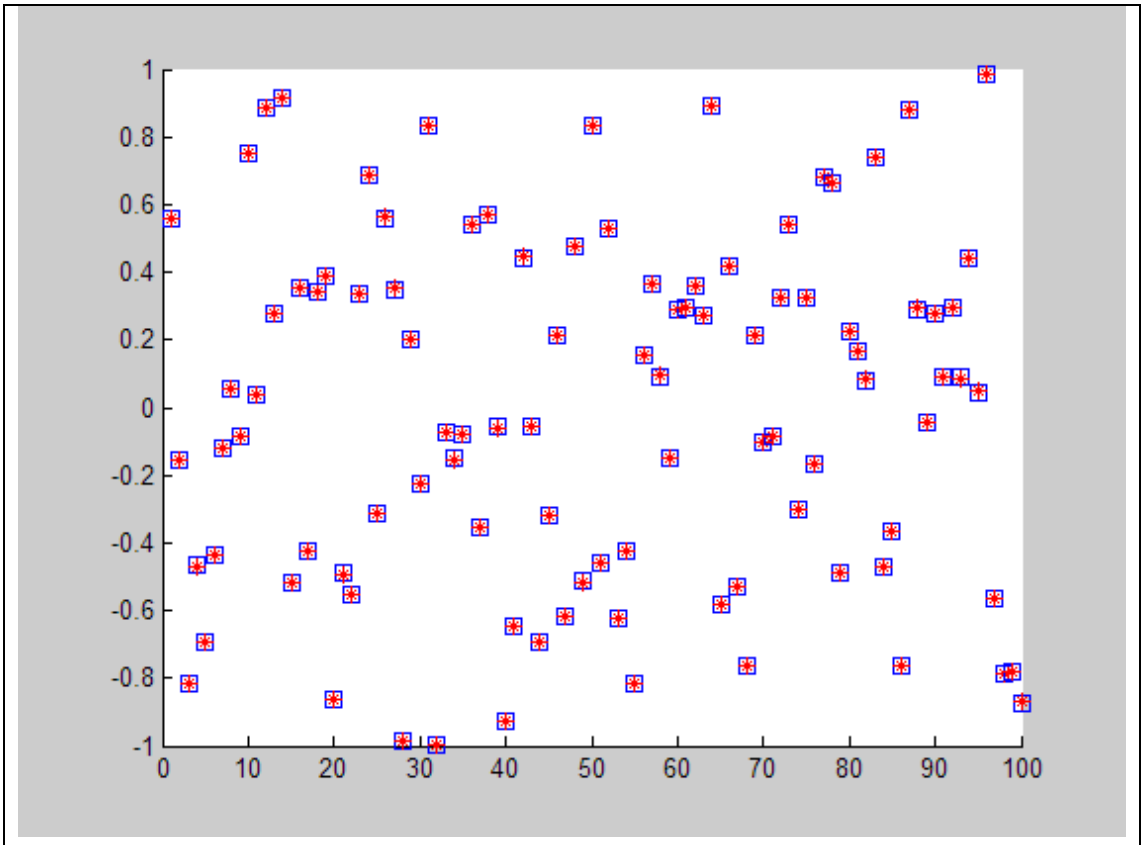
其中输入 A 是一个实数（可正可负），resolution 是截断精度，也就是二进制小数位的位数，返回 digits 位 CSD 编码，以及对应的权值 weight，err 为截断误差。下面是使用示例，

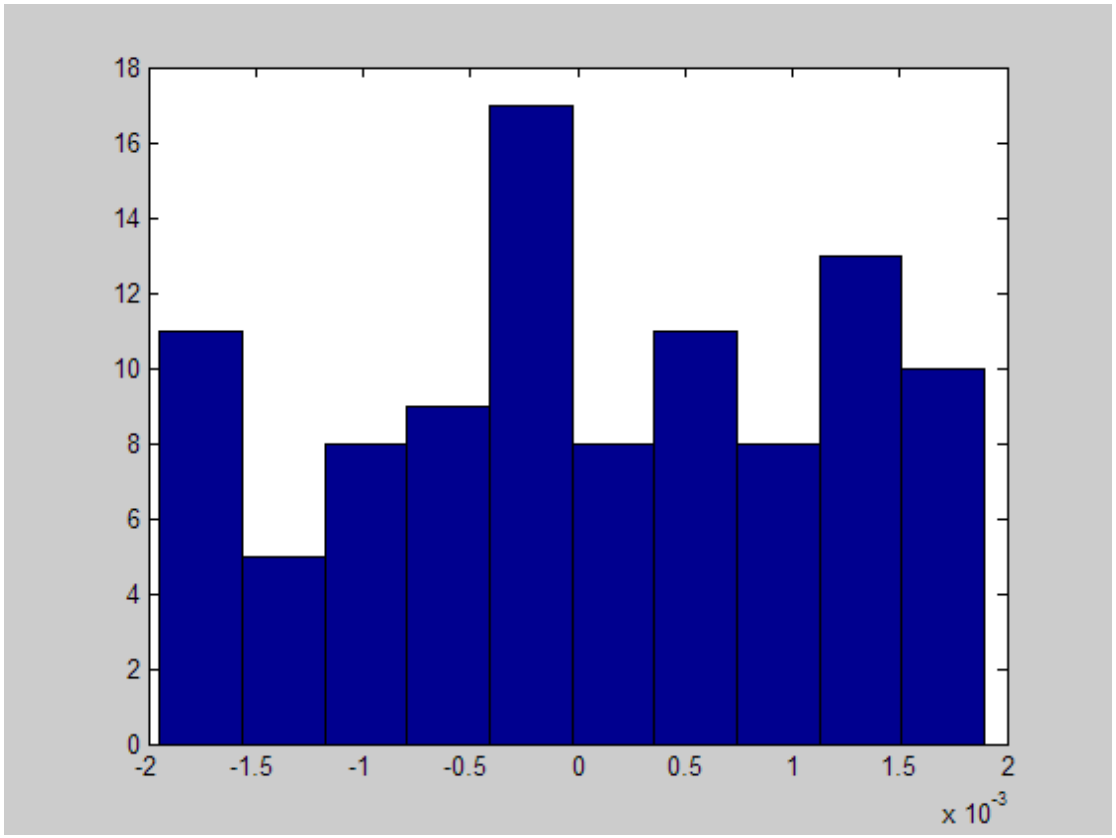
代码 2　CSD 编码示例

```
function csd_demo()
clc
close all

N   = 100;
r   = 2*rand(1,N)-1;
figure; hold on;
E   = zeros(1,N);
for k =1:N
    [digits,weight,E(k)] = csdigit(r(k),8);
    plot(k,r(k),'sb');
    plot(k,sum(digits.*2.^weight),'*r');
end

figure;
hist(E);

for k = 1:5
```

```
    r   = 5*rand-2.5;
    [digits,weight,err] = csdigit(r,8);
    disp(digits);
    disp(weight);
    disp([r,err]);
    disp('------------------------------');
end
```

```
    -1     0     0     0     1     0    -1     0

    -1    -2    -3    -4    -5    -6    -7    -8

   -0.4771   -0.0005

--------------------------------
     0    -1     0     0     0     0    -1     0

    -1    -2    -3    -4    -5    -6    -7    -8

   -0.2581   -0.0003

--------------------------------
    -1     0     1     0     1     0     1     0     0

     0    -1    -2    -3    -4    -5    -6    -7    -8

   -0.6709    0.0010

--------------------------------
     0     1     0     1     0     1     0     0     0     1

     1     0    -1    -2    -3    -4    -5    -6    -7    -8

    1.3175    0.0011

--------------------------------
     0     1     0     1     0     0     1     0     0

     0    -1    -2    -3    -4    -5    -6    -7    -8

    0.6395   -0.0011

--------------------------------
```





%%对 N 个数进行 CSD 编码并统计其误差分布，

示例代码中，对 N 个随机数进行 CSD 编码，并统计其误差分布，，第一个图分别画出原始实数（蓝色方框）和编码后的近似值（红色星号），可以看出红色星号基本落于蓝色方框中，说明编码误差不大，，第二个图给出 N 个近似误差的统计直方图。

题外话：google 也能找到 CSD 编码的 matlab 函数!!如果你对课本的转化方法感兴趣，可以研究这里给出的代码 4，大体思路是，先将实数转化为正整数，然后 CSD 编码，之后根据原始实数的正负性，对 CSD 编码结果修正一下即可得到最终 CSD 编码结果。

利用代码 4 得到常系数的 CSD 编码之后，就是具体的硬件实现过程。但实现过程中还有两个需要考虑的因素：累加的精度和累加的速度。举例说明，如$x \times 0.10100\bar{1}00\bar{1}0\bar{1}00\bar{1}$

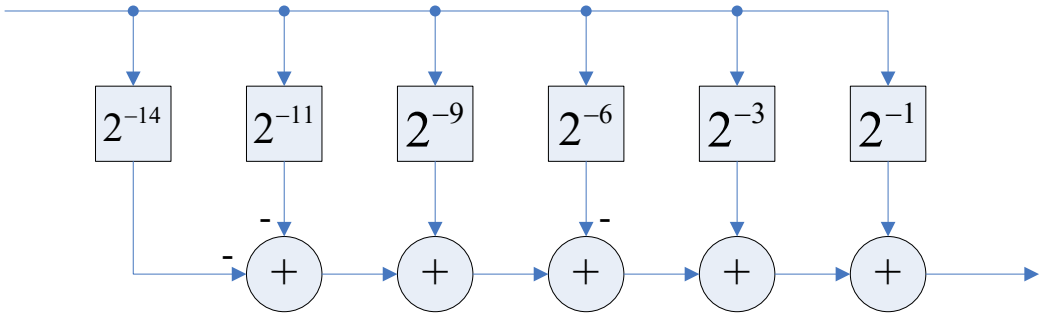


图 26　使用线性排列加法器的 CSD 乘法器实现

利用 Horner 法则可以提高计算精度，这个从课本的图 13-31 可以看出。这里给出图形示意：

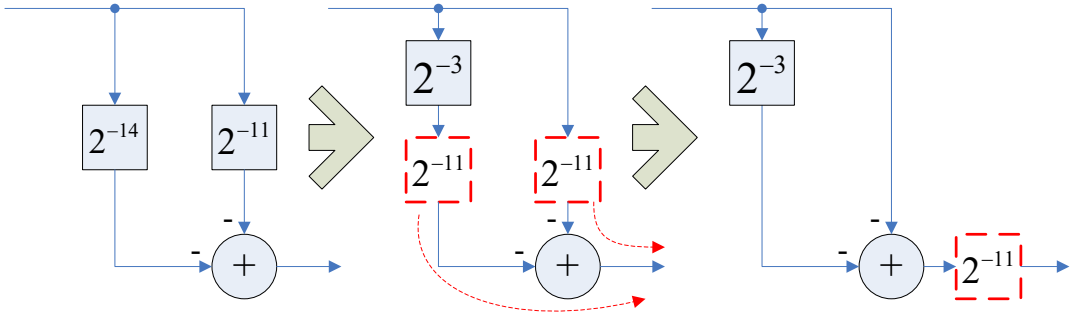


图 27

为什么，图 59 中可以将“一部分”移位单元移到加法器之后呢？下面的计算过程清楚的证明了这样做是对的!!

$$-2^{-14}-2^{-11}=-2^{-3}\cdot 2^{-11}-2^{-11}=\left(-2^{-3}-1\right)\cdot 2^{-11} \tag{10}$$

实际上很多算子都可以进行图 59 的操作而系统功能不变，，在第三章、重定时也属于这种操作！

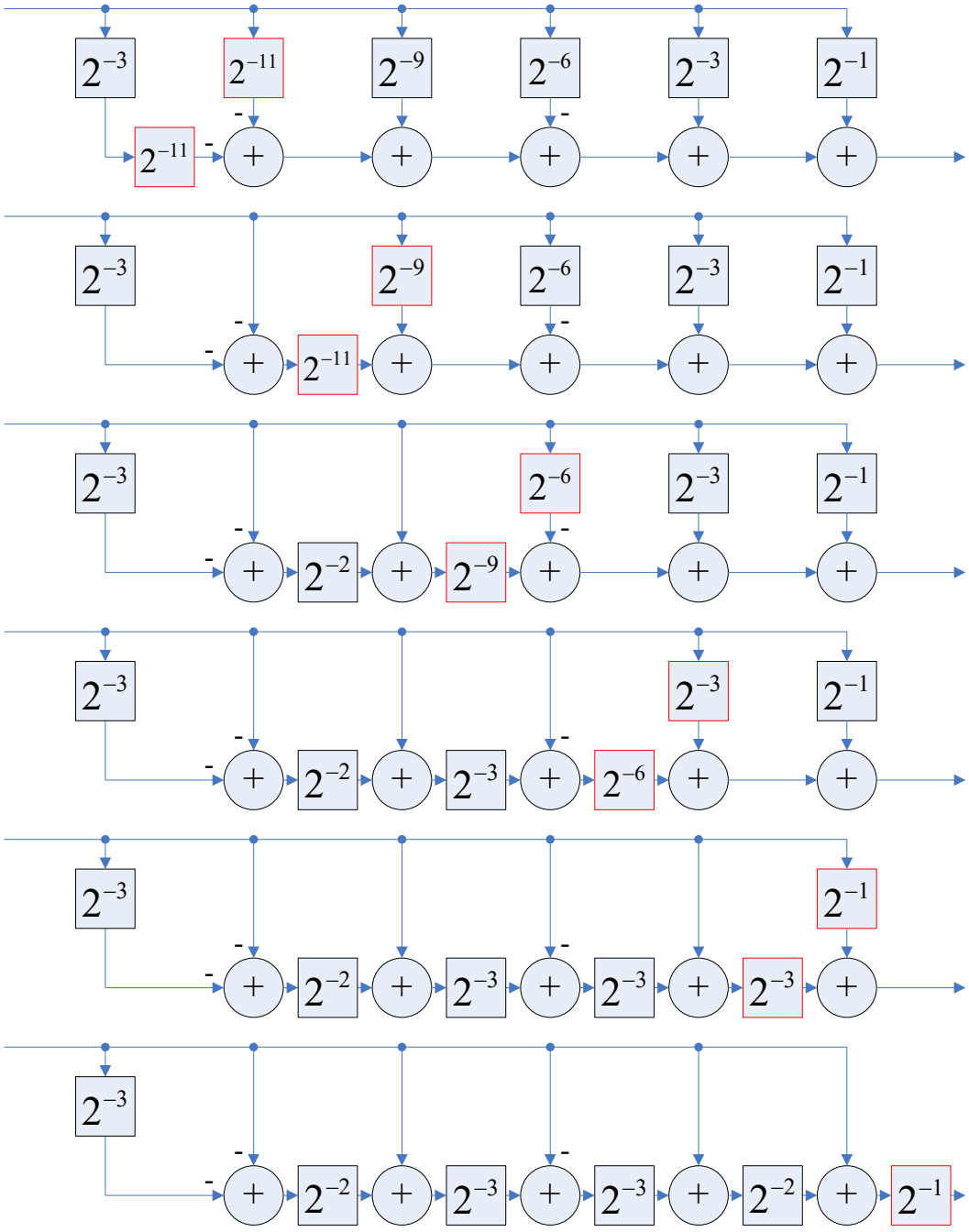


图 28　利用 Horner 法则重新排列 CSD 的部分积累加，以减少截断误差

更进一步，为了加快 CSD 部分积累加的速度，应该采用二叉树加法排列，修改图 58，得

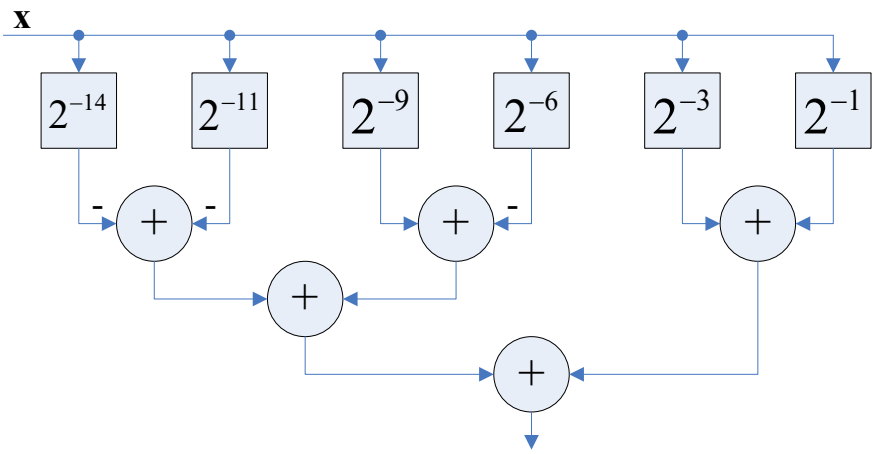


图 29　二叉树型 CSD 部分积累加

同样的，也可以将 Horner 法则应用到图 61，如下图示，

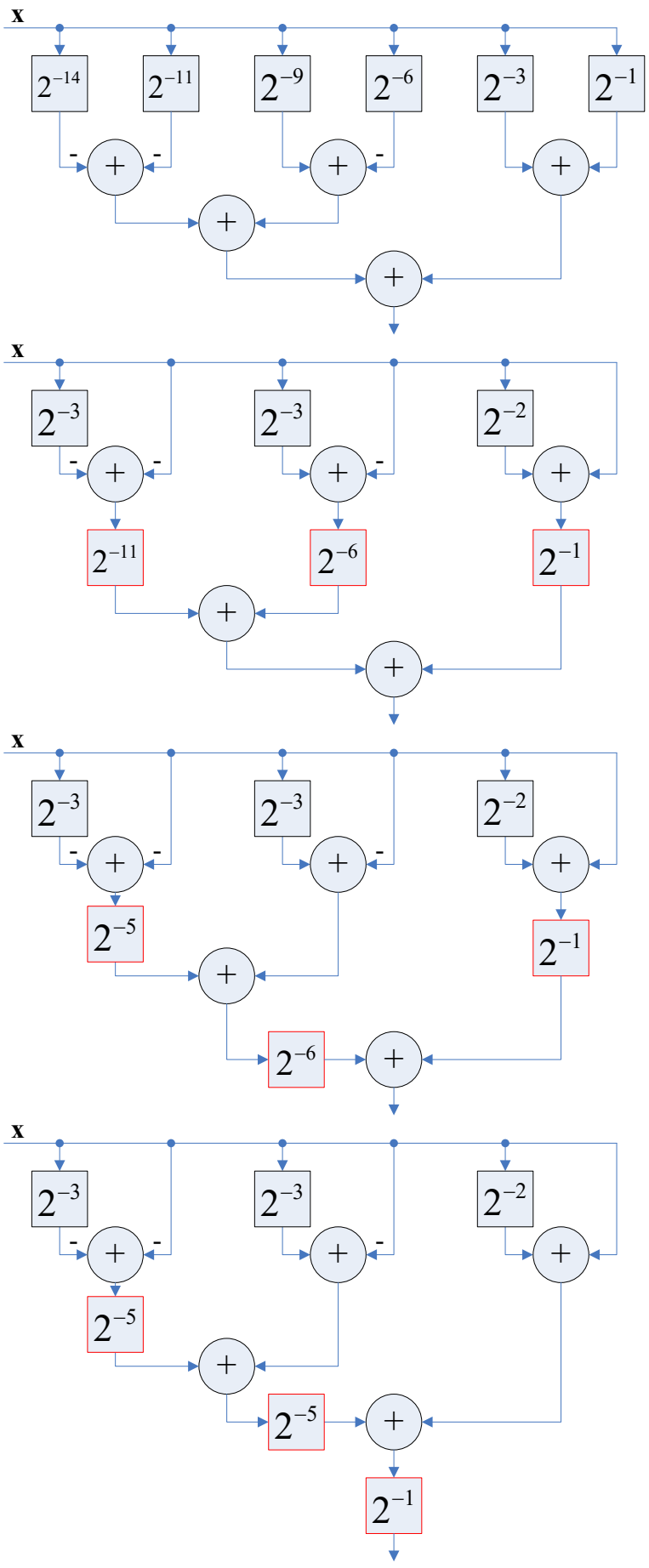

x
2^{-14}
2^{-11}
2^{-9}
2^{-6}
2^{-3}
2^{-1}
x
2^{-3}
2^{-3}
2^{-2}
2^{-11}
2^{-6}
2^{-1}
x
2^{-3}
2^{-3}
2^{-2}
2^{-5}
2^{-1}
2^{-6}
x
2^{-3}
2^{-3}
2^{-2}
2^{-5}
2^{-5}
2^{-1}

图 30　同时应用 Horner 法则和二叉树排列的 CSD 乘法器部分积累加电路

另外，提醒一下，为了防止中间加法计算溢出，有两种方法：1）输入 x 的范围限定在$\left[-\frac{1}{2},\frac{1}{2}\right)$；2）加法的每个操作数在相加之前必须进行恰当的缩放。。

结合前一节介绍的滤波器位串行架构设计，使用 CSD 可以进一步节约加法器资源，下面同时应用基于 Horner 法则和二叉树型排列的 CSD 乘法器来实现课本例 13.5.1 IIR 滤波器的位串行架构设计。

题外话：突然发现，前面对例 13.5.1 的设计中，已经用上 CSD 编码的形式，，但累加形式是线性排列，而且也没考虑计算精度问题。

迭代公式如下，

$$\begin{aligned} y(n) &= -\frac{7}{32}\cdot y(n-1)+\frac{3}{4}\cdot y(n-2)+x(n) \\ &= \left(0.0\bar{1}001\right)_{CSD}\cdot y(n-1)+\left(1.0\bar{1}\right)_{CSD}\cdot y(n-2)+x(n) \end{aligned} \tag{11}$$

这里不给出详细步骤，只给出两个线索（中间设计结果和最终设计结果），大家动手把它不全!!

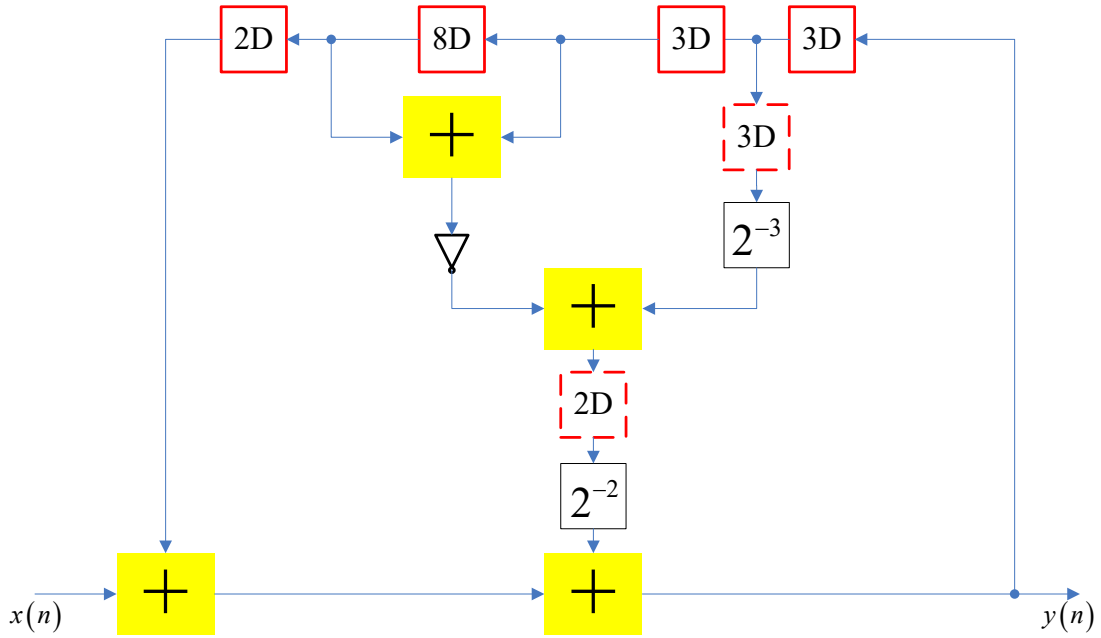


图 31　重定时之后的结构

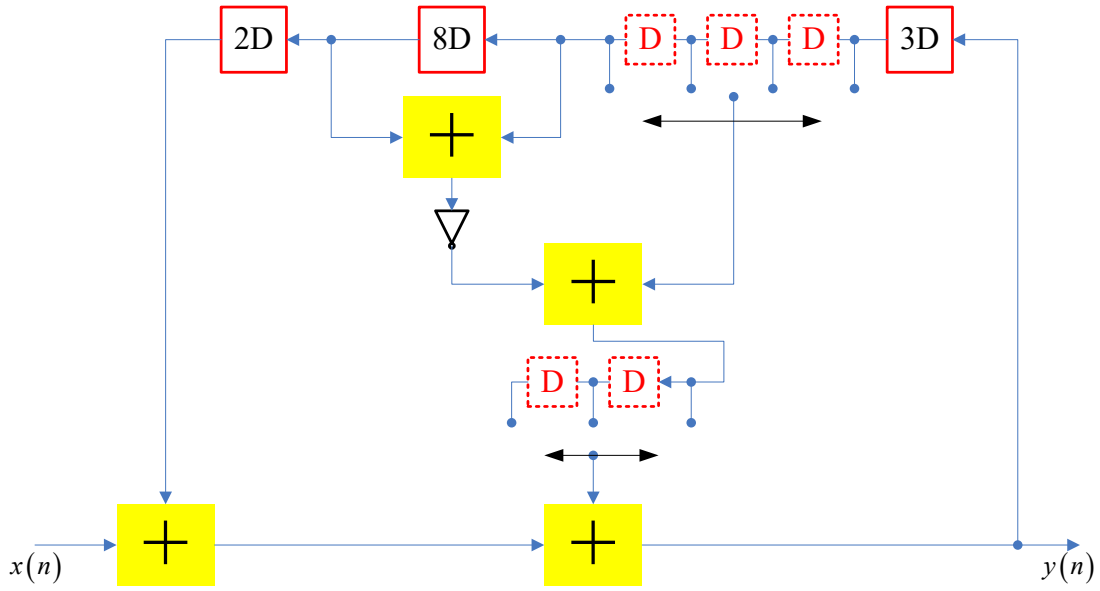


图 32　最终结构

CSD 数的一个缺点，即 CSD 的第一个特征：CSD 数中不存在两个连续的非零位，，比如

$$6 \to (10\bar{1}0)_{CSD}$$

其实编码为

$$6 \to (0110)_{POT}$$

会更好，因为交叠的位数更多，有利于提高计算结果精度，，因此，我编制了一个算法，该算法能给出比 CSD 更好的编码结果，同时非零比特位不比 CSD 编码多，，代码如下

代码 3　POT 编码，使用最少的 2 的幂次来逼近一个给定数 A

```
%%%
function [digits,weight,err] = potdigit(A,resolution)

s   = zeros(1,128);
a   = s;

s(1)= sign(A);
err = A;
E   = 2^(-resolution);

if abs(A)<E
    weight  = 0;
    digits  = 0;
    return;
end

k   = 0;
while (abs(err)>=E)
    k   = k+1;
    t   = log2(abs(err));
```

```
    if abs(err-s(k)*2^floor(t))>abs(s(k)*2^ceil(t)-err)
        a(k)    = ceil(t);
    else
        a(k)    = floor(t);
    end
    err     = err-s(k)*2^a(k);
    s(k+1)  = sign(err);
end

% digits  = s(1:k);
% weight  = a(1:k);

weight  = a(1):-1:a(k);
digits  = zeros(1,a(1)-a(k)+1);
digits(a(1)-a(1:k)+1)   = s(1:k);
```

下面是验证程序，

```
function pot_demo()
clc
close all

N   = 100;
r   = 2*rand(1,N)-1;
figure; hold on;
E   = zeros(1,N);
for k =1:N
    [digits,weight,E(k)] = potdigit(r(k),8);
    plot(k,r(k),'sb');
    plot(k,sum(digits.*2.^weight),'*r');
end

figure;
hist(E);


for k = 1:5
    r   = 5*rand-2.5;
    [digits,weight,err] = potdigit(r,8);
    disp(digits);
    disp(weight);
    disp([r,err]);
    disp('------------------------------');
end
```

1 0 0 1 0 -1 0 0 1

1 0 -1 -2 -3 -4 -5 -6 -7

2.1950 -0.0003

---

1 0 0 0 -1 0 0 0 0 1

1 0 -1 -2 -3 -4 -5 -6 -7 -8

1.8797 0.0008

1

-2

0.2508 0.0008

---

1 0 1 0 0 -1

-1 -2 -3 -4 -5 -6

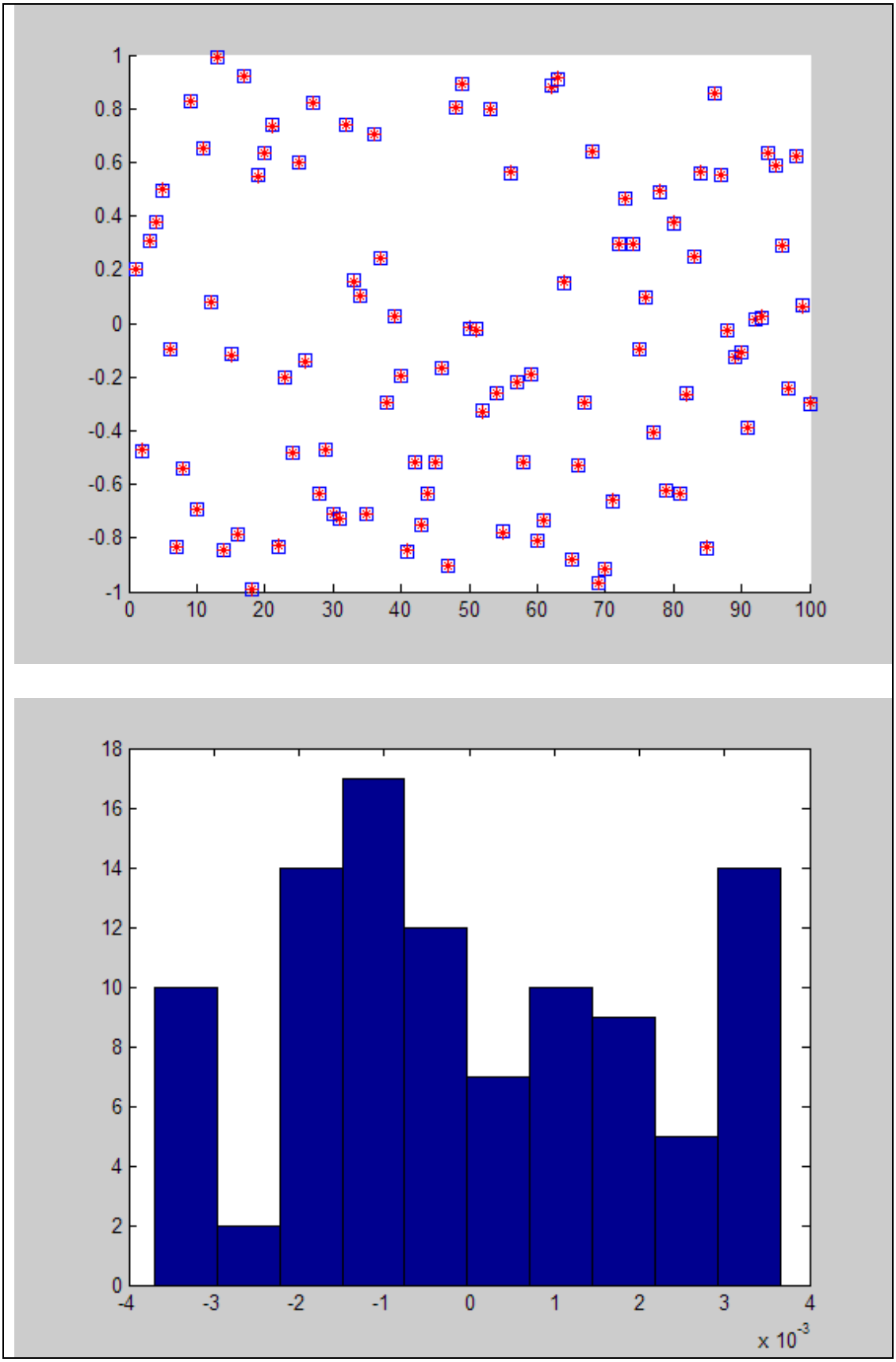
0.6124 0.0030

---

1 0 0 -1

-1 -2 -3 -4

0.4352 -0.0023



关于 CSD 暂时讨论这么多，，扩展阅读：实际应用中，经常将子表达式消除（子表达式共享）技术与 CSD 编码结合，设计更省资源的滤波器结构!!感兴趣的同学可以找文献进行研究。