第十三、位级运算架构

在 DSP 程序中，最为常见的运算就是乘法和加法。在高级设计中，常常会碰到乘法和加法的位级架构设计，涉及到位并行、位串行和数位串行三种实现类型。

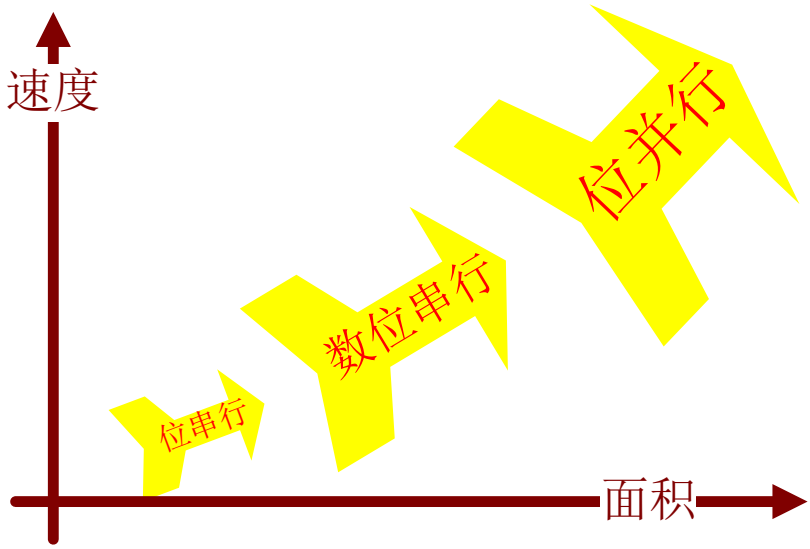


图 1　　位并行、数位串行 和 位串行 系统的速度和面积比较

位并行系统每个时钟周期处理输入样点的一个字，占用资源多但速度快；位串行系统每个时钟周期处理输入样点的一个位，速度慢但占用资源少；而数位串行居于两者之间，是一个折中方案。这么看来，掌握以上三种不同类型电路设计方法，就能根据实际情况给出有针对性的系统方案。

注意，位级架构设计是一个复杂的话题，目前还没统一设计方法——因此，这一章只讨论乘法器和滤波器的位级架构设计，而且主要关注 ① 位并行 和 ② 位串行 两者情况。其他系统的位级架构设计请调研相关的文献。

在本章的讨论中，我们不应满足于了解或记忆课本上给出的设计结果，而应该深入的考究为什么可以这么设计？为了弄懂“到底为什么可以这么设计”，我们的讨论将深入细节，也许你会觉得很琐碎，看不下去。。但请记住，这是位级的架构设计，而非算法级，所以不把琐碎的细节弄清楚就学不到东西。

讨论具体硬件设计方法之前，先进行一些约定：假设所有的有符号数都采用定点 2 的补码表示（2C），一个 W 位$[-1,1)$的数 A 可表示为

$$A = a_0.\underline{a}_1\underline{a}_2\cdots\underline{a}_{W-1} = -a_0 + \sum_{i=1}^{W-1}\underline{a}_i\cdot 2^{-i} \tag{1}$$

其中，$a_0 \in \{0,1\} \wedge \underline{a}_i|_{i=1,\ldots,W-1} \in \{0,1\}$，根据 2C 数定义，$a_0$称为符号位，显然公式(1)

定义了一个$[-1,1)$的小数，包括-1，但不包括+1（请记住这一点），对于公式(1)所表示的定点数记为 S1.W-1。一般的，定点数$SN.M$，其中 S 表示一位符号位，N 位整数位，M 位小数位，如公式(2)

$$a_N\cdots a_1a_0.\underline{a}_1\underline{a}_2\cdots\underline{a}_M=-a_N\cdot 2^N+\sum_{i=0}^{N-1}a_i\cdot 2^i+\sum_{i=0}^{M}\underline{a}_i\cdot 2^{-i} \tag{2}$$

对于无符号定点数，可以直接记为$N.M$，其中 N 位整数，M 位小数位，如公式(3)

$$a_{N-1}\cdots a_1a_0.\underline{a}_1\underline{a}_2\cdots\underline{a}_M=\sum_{i=0}^{N-1}a_i\cdot 2^i+\sum_{i=0}^{M}\underline{a}_i\cdot 2^{-i} \tag{3}$$

注意，对于具有负权值的系数，我们用带下划线的$\underline{a}_i$表示，整数部分的系数用正常的a_i表示，也可以认为$\underline{a}_i=a_{-i}$。

另：本章的最后一节 DA 算法中，还会涉及到偏移数的表示，具体在 DA 算法中讨论！

两个 W 位 2C 数相乘，精确结果为 2W-1 位 2C 数，如公式(4)

$$P=A\cdot B=\left(-a_0+\sum_{i=1}^{W-1}\underline{a}_i\cdot 2^{-i}\right)\cdot\left(-b_0+\sum_{i=1}^{W-1}\underline{b}_i\cdot 2^{-i}\right)=-p_0+\sum_{i=1}^{2(W-1)}\underline{p}_i\cdot 2^{-i} \tag{4}$$

这个不难理解，从

$$\left(\underline{a}_{W-1}\cdot 2^{-(W-1)}\right)\cdot\left(\underline{b}_{W-1}\cdot 2^{-(W-1)}\right)=\underline{a}_{W-1}\cdot\underline{b}_{W-1}\cdot 2^{-2(W-1)}$$

即可看出，P 的最小权值为$2^{-2(W-1)}$；又因为$\left(A,B\in[-1,1)\right)\wedge\left(A,B\text{不同时为}-1\right)$，所以$P\in[-1,1)$。在有限字长的计算中，可以对 P 的低位进行截断，最简单的方法就是直接舍弃（另一种方法是四舍五入）

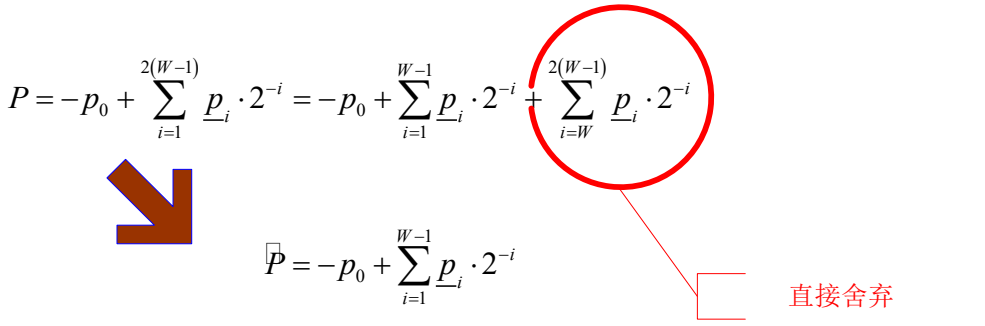


图 2　有限字长乘法中的截断，红圈部分为截断误差，$err\in\left(-2^{-W+1},2^{-W+1}\right)$

注意，A 和 B 不能同时为-1，否则 P=(-1)*(-1)=1，发生计算溢出。当然了，可以设置硬件检测电路，当发现 A=B=-1 时，用$0.\underbrace{11\cdots1}_{W-1\text{个}1}$作为 P 的近似值也是可以的。。。大致的设计思路如下图（未优化），

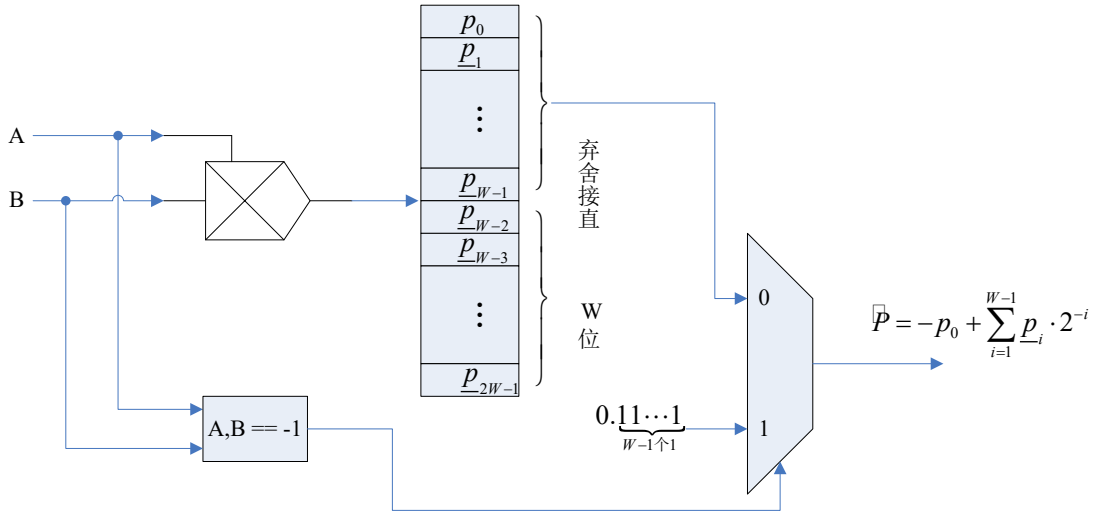


图 3　　改进的 2C 乘法模块

Horner 法则[1]，多项式求值快速算法，使用最少的乘法次数计算多项式的值，如公式(5)

$$P(x)=p_0+\sum_{i=1}^{N}p_i\cdot x^i \tag{5}$$

先看看“最笨的”计算方式的复杂度

$$\overbrace{p_0 \quad + \quad \underbrace{p_1\cdot x}_{1次乘法} \quad + \quad \underbrace{p_2\cdot x\cdot x}_{2次乘法} \quad + \quad \ldots \quad + \quad \underbrace{p_N\cdot x\cdot\ldots\cdot x}_{N次乘法}}^{N次加法} \tag{6}$$

一共需要$(N^2+N)/2$次乘法和N次加法。。

再来看看“中等笨”的计算方式，

$$\underbrace{p_0 \quad + \quad \underbrace{p_1\cdot x}_{1次乘法} \quad + \quad \underbrace{p_2\cdot x^2}_{1次乘法} \quad + \quad \underbrace{p_3\cdot x^3}_{1次乘法} \quad + \quad \ldots \quad + \quad \underbrace{p_{N-1}\cdot x^{N-1}}_{1次乘法} \quad + \quad \underbrace{p_N\cdot x^N}_{1次乘法}}_{N次加法}$$

$x \xrightarrow{x\cdot x} x^2 \xrightarrow{x^2\cdot x} x^3 \rightarrow \ldots \rightarrow x^{N-1} \xrightarrow{x^{N-1}\cdot x} x^N$（$x\cdot x$、$x^2\cdot x$、$x^{N-1}\cdot x$ 均为1次乘法）

(7)

这种方式“巧妙的”利用前一项计算后一项，即$x^i\cdot x\rightarrow x^{i+1}$，这样避免重复的乘法运算，总的复杂度为$2N-1$乘法，$N$次加法。。

最后，看看 Horner 计算方式，

$$\overbrace{p_0+\left(p_1+\left(p_2+\left(\ldots\left(p_{N-1}+p_N\cdot x\right)\cdot x\ldots\right)\cdot x\right)\cdot x\right)\cdot x}^{N次乘法} \tag{8}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{N次加法}$$

---

[1] 霍纳法则，可用于设计迭代乘法器

只需要N次乘法和N次加法。。比较说来，三种方法在加法次数上一样，但所需乘法次数 Horner 方式最少，实际上 Horner 方式所需乘法次数和加法次数恰好是多项式阶数大小。

下面跑跑题，来练习一下基于 Horner 法则的迭代乘法器设计。乘法算式如下

$$P = A \cdot B = A \cdot \left(-b_0 + \sum_{i=1}^{W-1} \underline{b}_i \cdot 2^{-i} \right) = -(A \cdot b_0) + \sum_{i=1}^{W-1} (A \cdot \underline{b}_i) \cdot 2^{-i} \tag{9}$$

对比公式(5)和(9)，相当于令

$$\begin{aligned} N &= W-1 \\ x &= 2^{-1} \\ p_0 &= -(A \cdot b_0) \\ \underline{p}_i &= A \cdot \underline{b}_i \end{aligned} \tag{10}$$

当然了，公式(5)中的乘法运算在公式(9)中变为取相反数或者右移操作。

思考题：请问，算术右移和逻辑右移的区别；2C 数的右移操作操作（乘 1/2）属于哪一种，具体硬件电路如何设计？

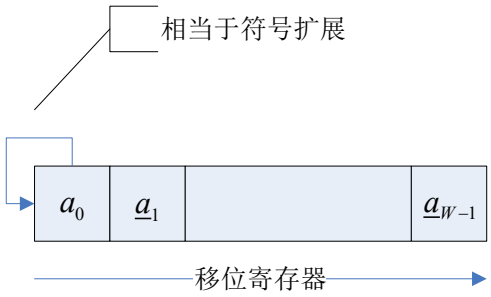


图 4　　2C 数的算术右移，等价于乘 1/2

下面根据公式(11)来设计一个迭代乘法器，

$$P = -A \cdot b_0 + \left(A \cdot \underline{b}_1 + \left(A \cdot \underline{b}_2 + \left(\ldots \left(A \cdot \underline{b}_{W-2} + A \cdot \underline{b}_{W-1} \cdot 2^{-1} \right) \cdot 2^{-1} \ldots \right) \cdot 2^{-1} \right) \cdot 2^{-1} \right) \cdot 2^{-1} \tag{11}$$

不要以为很容易设计，一上来很多同学会忽略甚至都没想到，中间结果溢出的情况。为了避免中间结果溢出，有两个方式：1）强制输入的操作数在$[-1/2, 1/2)$之间，2）符号扩展一位。

对于第 1 种方式，本质上就是用“不考虑溢出的 W+1 字长的乘法器来进行 W 字长乘法”，其中缘由大家自己思考。下面是用第 2 种方式来设计 Horner 迭代乘法器，

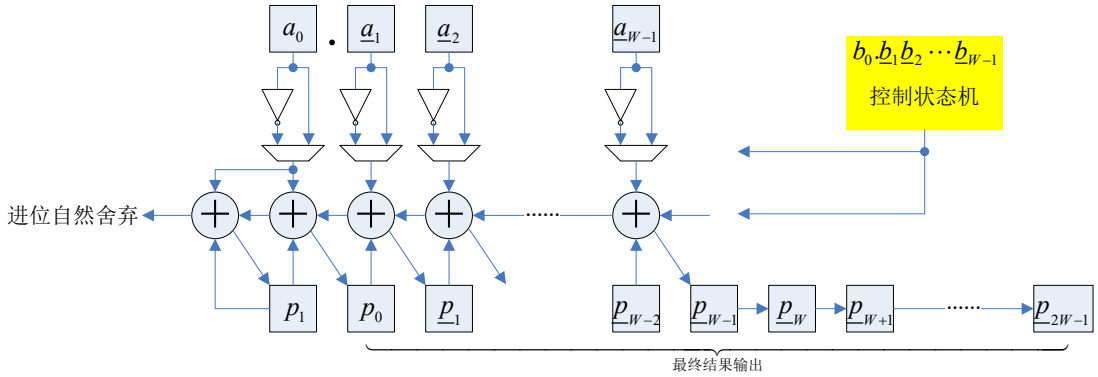


图 5　　Horner 迭代乘法器

并行乘法器设计

这一段我们讨论几种典型的并行乘法器（也成为阵列乘法器）设计，进行某种并行乘法器设计之前，首先要清楚乘法的内部细节，

$$
\begin{aligned}
A\cdot B &= \left(-a_3\cdot 2^3+a_2\cdot 2^2+a_1\cdot 2^1+a_0\cdot 2^0\right)\left(-b_3+b_2\cdot 2^2+b_1\cdot 2^1+b_0\cdot 2^0\right)\\
&= \left(-a_3+a_2\cdot 2^2+a_1\cdot 2^1+a_0\cdot 2^0\right)\cdot b_0\cdot 2^0\\
&+ \left(-a_3+a_2\cdot 2^2+a_1\cdot 2^1+a_0\cdot 2^0\right)\cdot b_1\cdot 2^1\\
&+ \left(-a_3+a_2\cdot 2^2+a_1\cdot 2^1+a_0\cdot 2^0\right)\cdot b_2\cdot 2^2\\
&- \left(-a_3+a_2\cdot 2^2+a_1\cdot 2^1+a_0\cdot 2^0\right)\cdot b_2\cdot 2^3\\
&= \left(-a_3b_0+a_2b_0\cdot 2^2+a_1b_0\cdot 2^1+a_0b_0\cdot 2^0\right)\cdot 2^0\\
&+ \left(-a_3b_1+a_2b_1\cdot 2^2+a_1b_1\cdot 2^1+a_0b_1\cdot 2^0\right)\cdot 2^1\\
&+ \left(-a_3b_2+a_2b_2\cdot 2^2+a_1b_2\cdot 2^1+a_0b_2\cdot 2^0\right)\cdot 2^2\\
&+ \left(-\bar{a}_3b_3+\bar{a}_2b_3\cdot 2^2+\bar{a}_1b_3\cdot 2^1+\bar{a}_0b_3\cdot 2^0+LSB'1\right)\cdot 2^3
\end{aligned}
\tag{12}
$$

将公式(12)写成表格形式如图 6，

				$-a_3$	a_2	a_1	a_0
×				$-b_3$	b_2	b_1	b_0
				$-a_3b_0$	a_2b_0	a_1b_0	a_0b_0
+			$-a_3b_1$	a_2b_1	a_1b_1	a_0b_1	
		$-a_3b_2$	a_2b_2	a_1b_2	a_0b_2		
	$-\bar{a}_3b_3$	$\bar{a}_2b_3$	$\bar{a}_1b_3$	$\bar{a}_0b_3$			
				b_3			
p_6	p_5	p_4	p_3	p_2	p_1	p_0	

图 6　　2C 乘法的内部细节

注意，，图 6 所表示的是 2C 乘法细节，所以$-a_3b_i$前面都带有负号，且第四行清楚的表示了

取反+$LSB'1$。

如何（直观地）设计图五的乘法硬件电路呢？首先要明白一点是：图 6 中上下两行相加，必须进行符号位对齐，也就是说

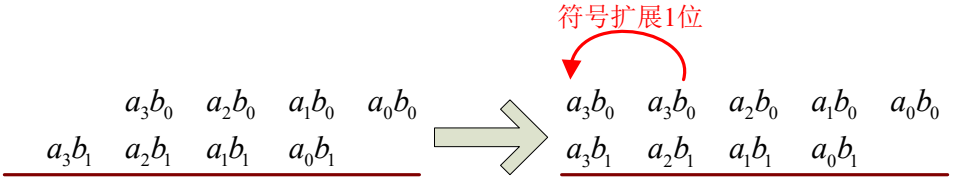


图 7　　2C 数相加之前要进行符号对齐，使用符号扩展一位即可解决

这样一来，图 6 的硬件过程变为

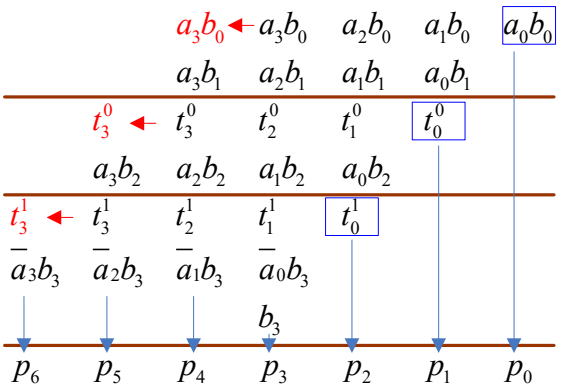


图 8　　不考虑内部溢出的 2C 乘法过程

图 8 所示的过程是不考虑内部溢出的，实际上如果不对输入进行限制，则最终计算结果可能出错：解决方法之一就是限制输入操作数取值范围为原来的 1/2，这样就不会发生溢出；另一种解决方法是多进行一位符号扩展。

本质上说，第一种解决方法就是用 W+1 字长乘法器（图 8）来完成 W 字长的操作数相乘。下面重点讨论一下第二种解决方法，

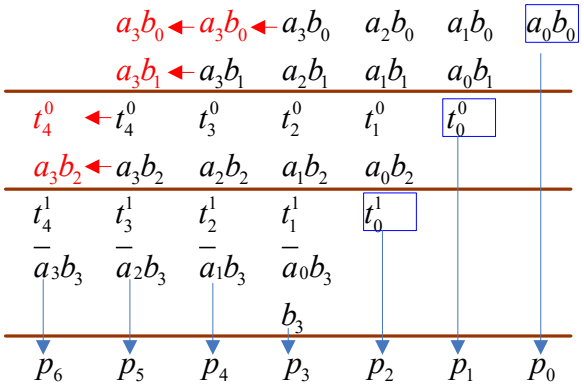


图 9　　考虑内部溢出的 2C 乘法过程，使用符号扩展来避免溢出

练习题：为了验证图 9 计算过程的正确性，可以设置溢出检测电路（图 75 左图），如果没检测到溢出错误，则图 9 计算过程是正确的。试用 verilog 代码来描述电路，并进行详尽的

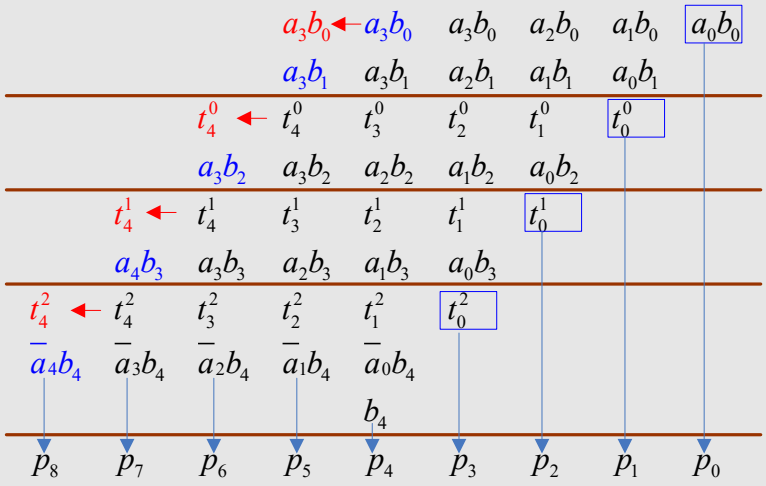
仿真！

更为“吝啬的”的警戒位设计方案

先来讨论原始警戒位设计方案，假设输入字长 W=4。因为设置一位警戒位，实际是操作数变为

$$\begin{cases} A = a_3 & a_3 & a_2 & a_1 & a_0 \\ B = b_3 & b_3 & b_2 & b_1 & b_0 \end{cases}$$

然后按图 8 的计算方式计算即可，如下图，

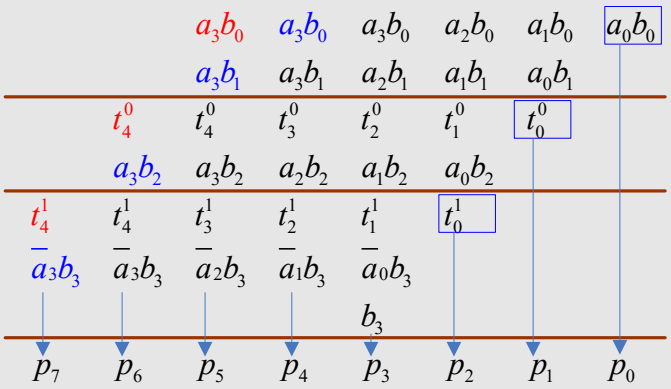


这就相当于，使用 W+1 位的不考虑内部溢出的乘法器来实现 W 位乘法，，计算结果如果正确的话，必有 $p_8 = p_7 = p_6$，其实是符号扩展造成的影响（可进行仿真验证），最终输出结果应为 $P = p_6p_5p_4p_3p_2p_1p_0$。

上面的计算方法是有冗余的，实际上只需在其中一个操作数上设立警戒位即可，比如说将操作数改为，

$$\begin{cases} A = a_3 & a_3 & a_2 & a_1 & a_0 \\ B = & b_3 & b_2 & b_1 & b_0 \end{cases}$$

只在 A 上设立警戒位，B 不设置，这样可以减少一次部分积累加，加快乘法速度，具体的计算过程如下图



类似，必有 $p_7 = p_6$，最终结果为 $P = p_6p_5p_4p_3p_2p_1p_0$。

上面我们介绍了若干种稍有不同的并行乘法器计算过程，下面以图 8 过程为例，给出具体的硬件电路，其他计算过程也可类似构造电路！

观察图 8 可知，乘法器说到底就是多次部分积累加运算，，如何来快速完成多次累加呢？

1） 最简单的，串行进位加法

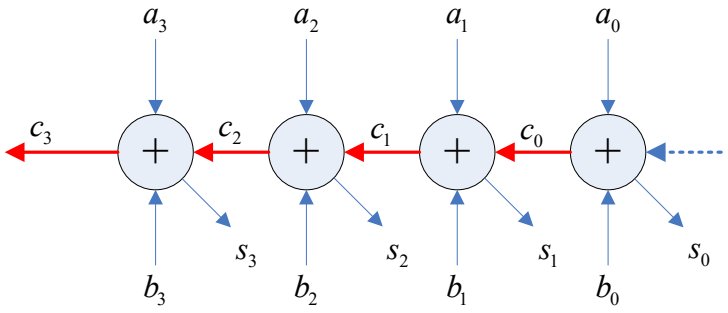


图 10　　串行进位加法器，红线为进位链

2） 速度更快，进位保留加法（CSA）

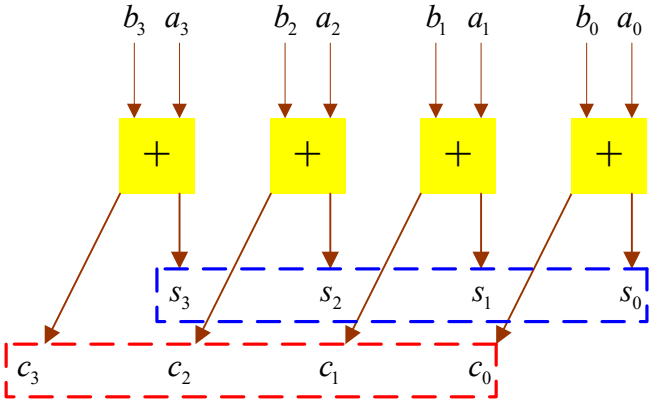


图 11　　进位保留加法器，没有进位链

进位保留加法是一种很巧妙的技术，不过有点让人莫名其妙，因为图 11 并没给出完整

的加法结果，而只是分别给出了“和 S”以及“进位 C”,,，要得到最终结果，还必须将 S 和 C 相加。？？？？很多人会有疑问，这种设计能带来啥好处？不急，其实进位保留加法优势在于多次累加，如图

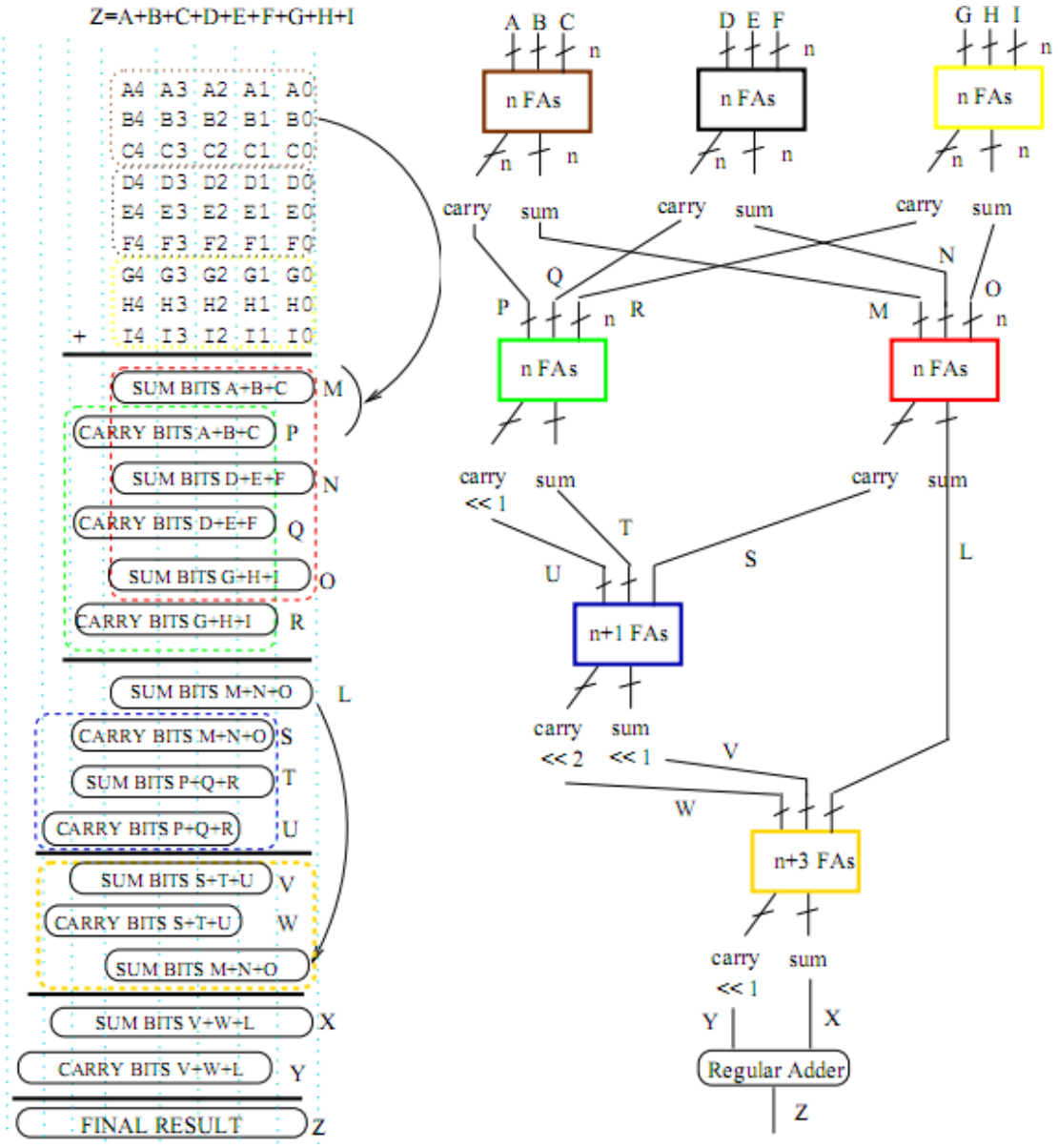


图 12　多次累加的进位保留加法运算，只在最后一级需要一个矢量合并器（常规加法器）

[点击查看详情]

以图 8 为例，采用不同的累加方式得到不同的阵列乘法器依赖图。采用串行进位的阵列乘法器依赖图如图 13，使用水平的割集流水线可将关键路径从 2W(+1*)缩短为 W，如图 14。

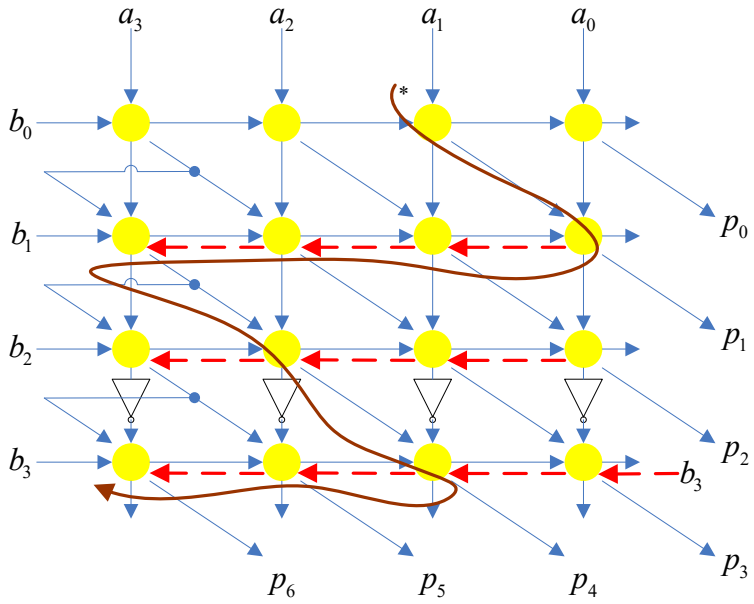


图 13　4x4 位串行进位阵列乘法器依赖图，褐色线为关键路径，2W(+1*)

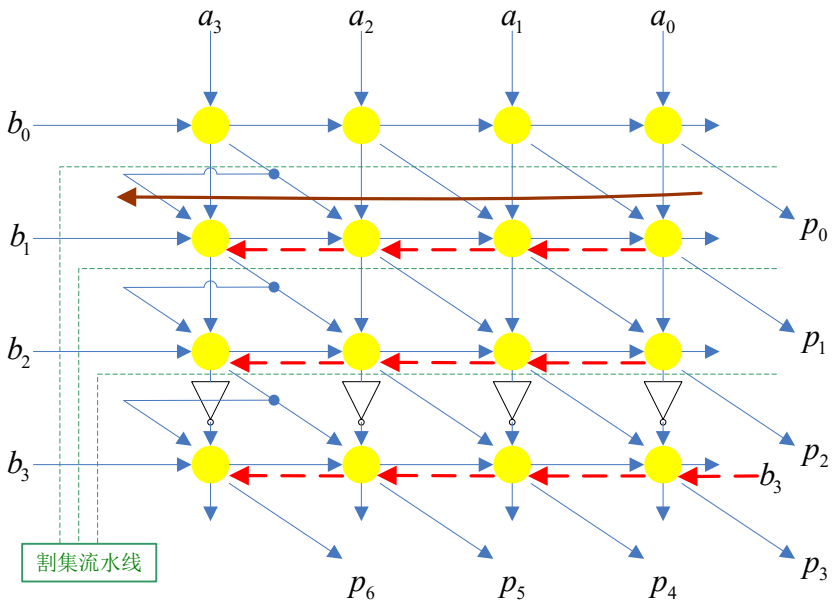


图 14　使用割集流水线斩断关键路径，新关键路径长 W

采用进位保留累加方式的阵列乘法器依赖图如图 15，撇开矢量合并器不提，关键路径长度为 W-1(+1*)，同样的可以使用割集流水线斩断关键路径，如图 16 使用了 2 维流水线，关键路径缩短为 1。。。。

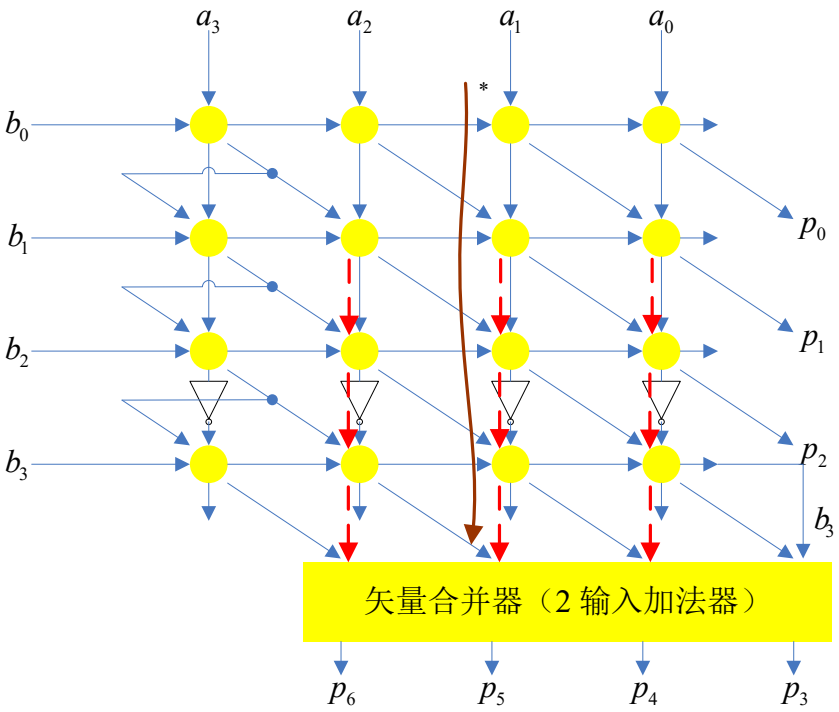


图 15　4x4 进位保留乘法阵列，不考虑矢量合并器的话，关键路径长为 W-1(+1*)

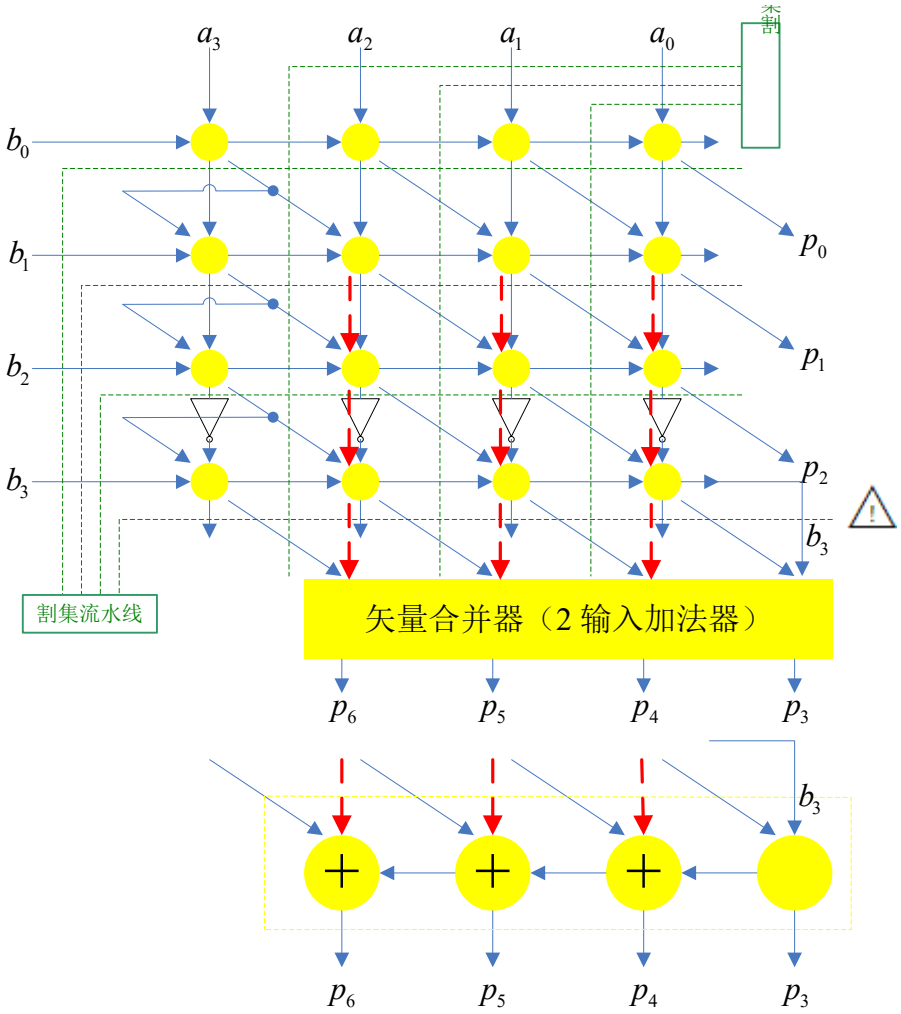
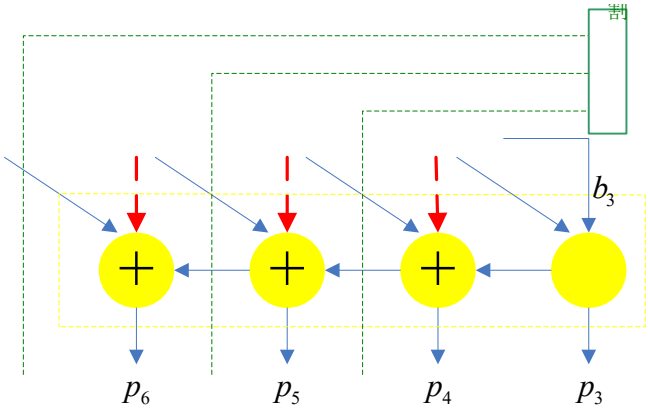


图 16　使用 2 维流水线的进位保留加法器，关键路径缩短为 1，，？？

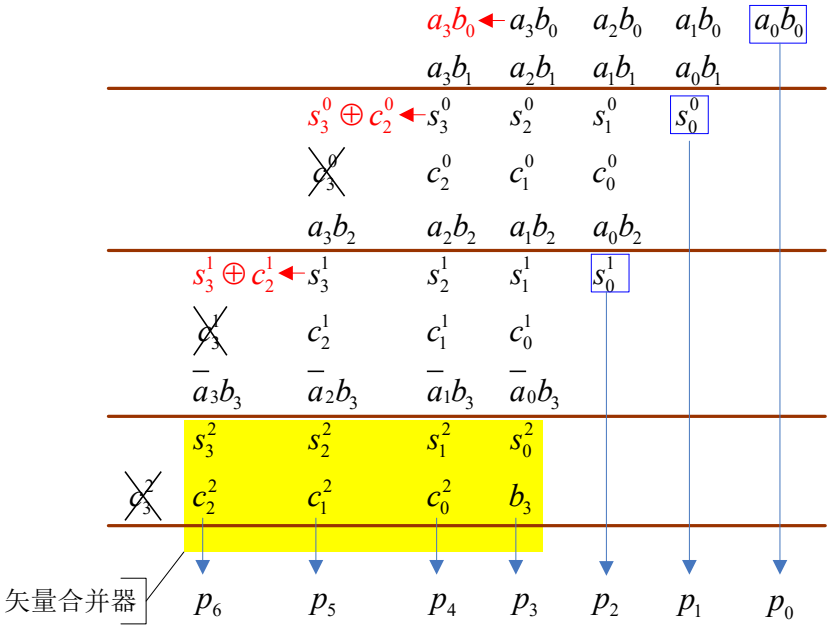
实际上，必须设计出关键路径为 1 的矢量合并器，才能使得图 16 运转起来！当然了，存在很多高速加法器设计方案，所以不要担心实现不了。

思考题：不要以为图 16 中竖直方向流水线技术可应用于矢量合并器，为什么呢？有没有办法在矢量合并器中使用竖直割集流水线呢（提示，请体会课本图 13-7 和图 13-9，其中提供了可用竖直割集流水线的矢量合并器）？



我的疑问？

图 15 和 图 16 给出了基于进位保留累加方式的阵列乘法器依赖图（来自于课本）及其流水线实现方案，，但是，进位保留加法器可能会在符号扩展的计算上出错！！



正如上图所示，除了第一次累加是正常符号扩展之外，第二次和第三次（除了矢量合并外所有后面的累加），其符号扩展应该是“和最高位 ⊕ 相应的进位位”作为符号扩展，而不是“和最高位”作为符号扩展位！

也许仿真验证一下更能知道哪个是对，哪个是错！（不是说课本上错误，只是可能作者没有给出所有细节，因此学习者应该自己把这些细节补上）

上面一段都是基于图 8 来进行设计，细心的同学会发现，每次累加都需要进行符号扩展来进行符号位对齐，，一个巧妙的想法是：能不能把具有负权值的项统一放到最后进行累加呢？

下面就动手来试试，看能否实现！从原始乘法公式入手，

$$\begin{aligned} A\cdot B &= \left(-a_3\cdot 2^3+\sum_{i=0}^{2}a_i\cdot 2^i\right)\left(-b_3\cdot 2^3+\sum_{i=0}^{2}b_i\cdot 2^i\right) \\ &= \left(b_3\cdot a_3\cdot 2^6\right)+ \\ &\quad \left(-a_3\cdot 2^3\cdot\sum_{i=0}^{2}b_i\cdot 2^i\right)+\left(-b_3\cdot 2^3\cdot\sum_{i=0}^{2}a_i\cdot 2^i\right)+ \\ &\quad \left(\left(\sum_{i=0}^{2}a_i\cdot 2^i\right)\left(\sum_{i=0}^{2}b_i\cdot 2^i\right)\right) \end{aligned} \tag{13}$$

公式(13)红色部分两项都是纯粹的负数，首先将其转化为常规补码形式，如下

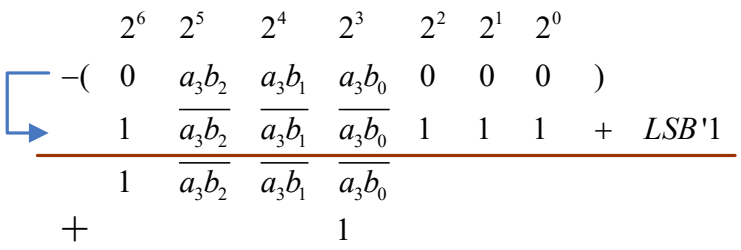


图 17

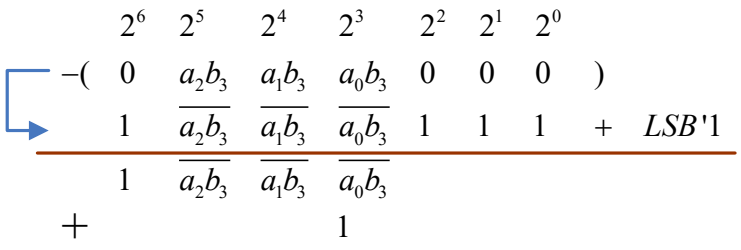


图 18

然后可将公式(13)的三个部分组合在一起，如下图 19，

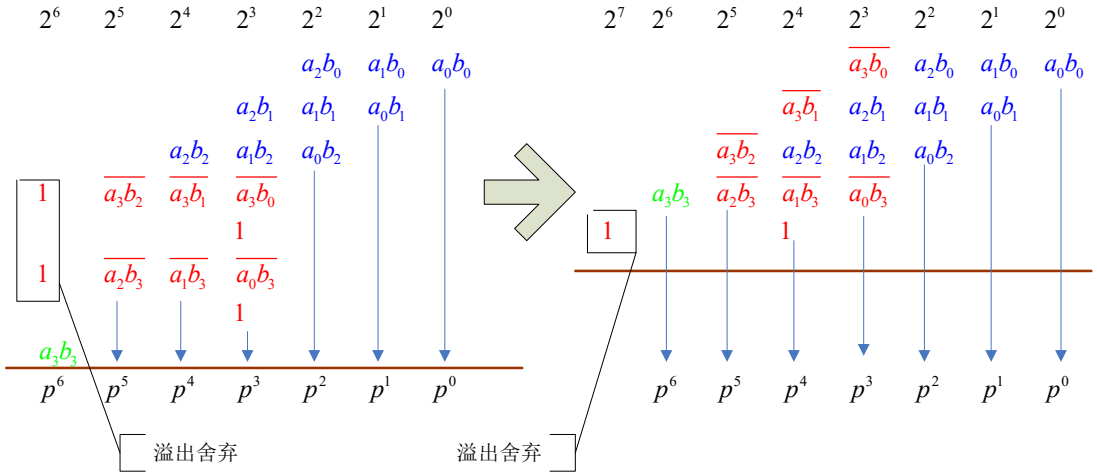


图 19　Baugh-Wooley 乘法器过程

经过巧妙的变形术，竟然能不用符号扩展就能实现乘法器!! 图 19 的 Baugh-Wooley 乘法器比图 8 的一般 2C 乘法器要节省资源（这是个非常有意思的变形）。

练习题：设计基于进位累加和进位保留的 Baugh-Wooley 乘法器，并仿真验证!

乘法器设计的方案很多，上面我们讨论了标准的乘法器设计，和，巧妙的 Baugh-Wooley 乘法器设计。不论怎么说，上面讨论的都是经典乘法器范畴，下面讨论如何来设计更为高速的乘法器。

课本上总结得很好：乘法运算设计两个部分，部分积的产生 和 部分积的累加。“对症下药”，

有两条提高乘法速度的途径：1）减少部分积的数目，2）加快部分积累加速度。。前面讨论的进位保留加法器就是 通过加快部分积累加速度来提高乘法器速度，，那么可不可以设计一种只进行少量部分积累加的乘法器呢？

高手们总是能给出满意的答案，基本思路：通过改造数的编码，从而产生更少的部分积，典型的有 Booth 编码和 CSD 编码，，这里先讨论 Booth 编码乘法器，本章后面还会讨论 CSD 编码乘法器。

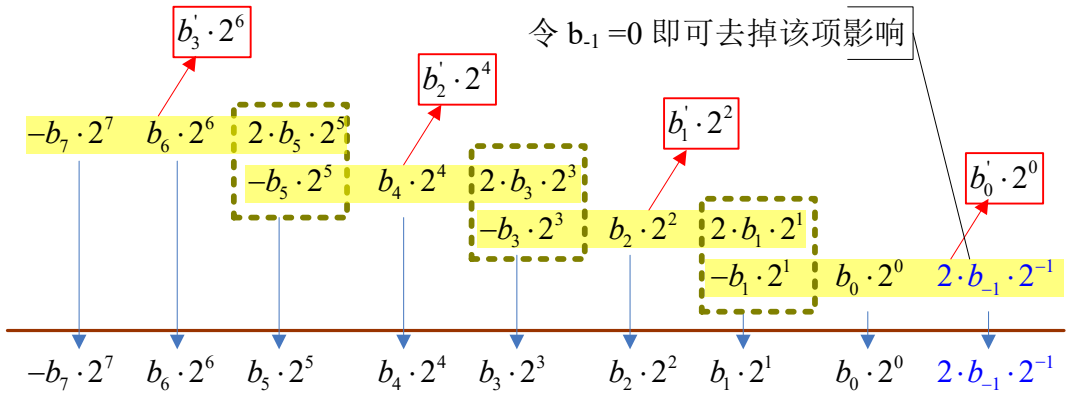


图 20　Booth 编码，其中$b_{-1}=0$

改进的 Booth 乘法器，对其中一个乘数进行重编码，如图 20，也就是说，将

$$b_7b_6b_5b_4b_3b_2b_1b_0 \rightarrow b_3'b_2'b_1'b_0' \tag{14}$$

编码公式如下

$$b_i' = -2 \cdot b_{2i+1} + b_{2i} + b_{2i-1} \tag{15}$$

其中，$b_i' \in \{-2,-1,0,1,2\}, i=0,1,2,3$，称为 5 级 Booth 重编码，容易证明，

$$\sum_{i=0}^{3} b_i' \cdot 2^{2i} = -b_7 \cdot 2^7 + \sum_{i=0}^{6} b_i \cdot 2^i \tag{16}$$

代码 1　Booth 编码的验证

```
> restart:
> b_-1 := 0;
                        b_-1 := 0
> f := sum(B_i·2^(2·i), i = 0..3);
                  f := B_0 + 4 B_1 + 16 B_2 + 64 B_3
> sort(subs(seq(B_i = -2·b_(2 i+1) + b_(2 i) + b_(2 i-1), i = 0..3), f))
          b_0 + 2 b_1 + 4 b_2 + 8 b_3 + 16 b_4 + 32 b_5 + 64 b_6 - 128 b_7
```

注意，（改进）BOOTH 乘法只对其中一个乘数进行重编码，比如 B，，那么 A 仍然保持 2C

编码，所以

$$\begin{aligned} A \cdot B &= A \cdot \sum_{i=0}^{3} b_i' \cdot 2^{2i} \\ &= A \cdot b_0' + A \cdot b_1' \cdot 2^2 + A \cdot b_2' \cdot 2^4 + A \cdot b_3' \cdot 2^6 \end{aligned} \tag{17}$$

原本应进行 7 次部分积累加，现在“最多”只需进行 3 次部分积累加。。其实，以上讨论还不足以显示出 BOOTH 乘法的威力，我们使用 matlab 来进行一些试编码一些例子，从中总结一下，

代码 2　改进 BOOTH 的非零比特数统计

```
function booth_demo()
clc
N   = 500;
b   = rand(N,64)>0.5;
M   = numel(b);
disp('number of non-zero bits in b');disp(sum(b(:))/M);
n   = 0;
for k = 1:N
    n   = n+sum(booth_improve(b(k,:))~=0);
end
disp('number of non-zero bits in B');disp(n/M);
disp('save')
disp((sum(b(:))-n)/sum(b(:)));

%b=[MSB... b7,b6,b5,b4,b3,b2,b1,b0,...LSB];
function B  = booth_improve(b)
b   = fliplr(b);
N   = length(b);
K   = ceil(N/2);
if mod(N,2)
    b   = [0,b,0];
else
    b   = [0,b];
end
B   = zeros(1,K);
for k = 1:K
    B(k)= -2*b(2*k+1)+b(2*k)+b(2*k-1);
end
B   = fliplr(B);
```

```
number of non-zero bits in b
    0.5022

number of non-zero bits in B
    0.3744

save
```

```
0.2546
```

代码 2，统计 64 位随机数的改进 BOOTH 编码的非零比特数，从运行结果可知，改进 BOOTH 编码的非零比特比未编码的 2C 数非零比特少 13.03%，也就是说可以节约 25.46%的计算功耗？

总结起来，（改进）BOOTH 编码不仅仅可以减少部分积数目（或者说部分积累加次数），而且还能带来功耗上的改善！

特别是，当 2C 码中出现大量连续的 0 或连续的 1 时，（改进）BOOTH 编码将给出更少的非零比特，从而大幅节约功耗！

代码 3　大量连 1 或连 0 的改进 BOOTH 编码

```
disp('booth_improve([1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1])');
disp(booth_improve([1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1]));
disp('booth_improve([1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 0 1 0 0 0 0 0])');
disp(booth_improve([1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 0 1 0 0 0 0 0]));

disp('booth_improve([1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1])');
disp(booth_improve([1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1]));
```

```
booth_improve([1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1])
     2     0    -2     2     0     0     0    -1     2     0     0    -1

booth_improve([1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 0 1 0 0 0 0 0])
     1     0     0     1     0    -2     0     1    -1    -2     0     0

% “最槽糕”的改进BOOTH编码,^_^bbb
booth_improve([1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1])
     1     1     1     1     1     1     1     1     1     1     1     1
```

练习题：请查资料，弄明白标准 BOOTH 编码算法，并给出标准 BOOTH 和改进 BOOTH 的区别！

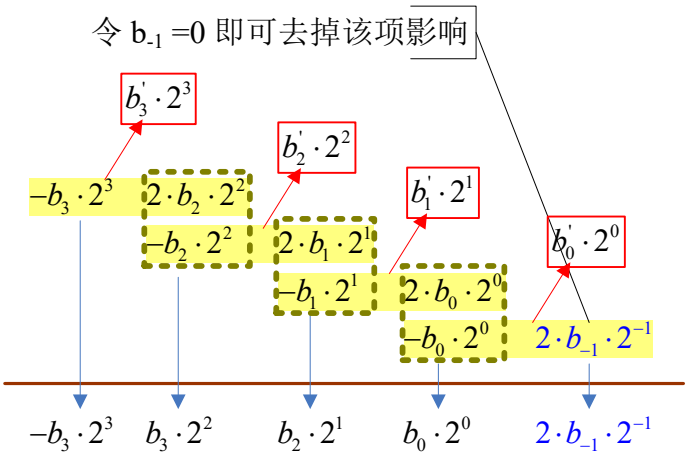


图 21　标准 BOOTH 编码？

关于乘法器的位级运算架构就讨论那么多，想学得更为深入，可以参考最新的相关文献！课本的 Section 13.3，介绍了数字滤波器中，将多个“乘法-加法”的部分积混合在一起计算的一种布局，合理的布局会带来布线复杂度的降低，从而提高系统稳定性和降低功耗节约面积等一堆好处!!

至于深入的布局知识，这里就不讨论了，下图是可用于混合布局的“乘法-加法”特例，

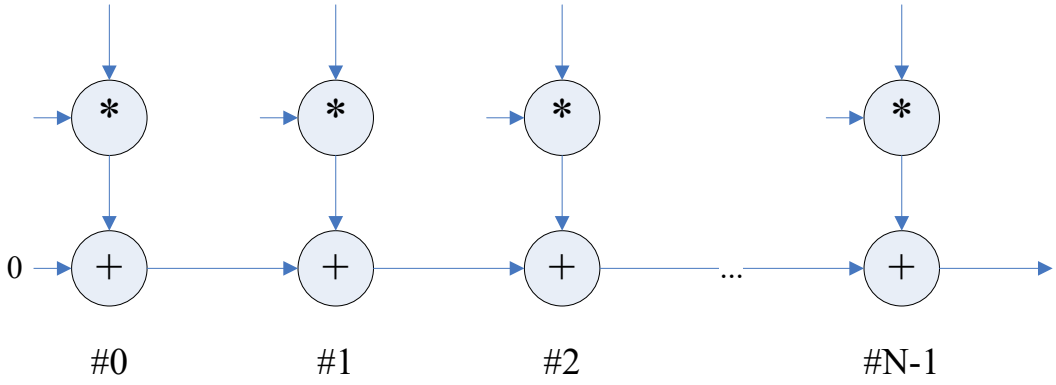


图 22　对各个“乘加”单元的部分积合理布局，混合在一起计算，以便降低布线复杂度

脉动位串行乘法器设计

下面进入本章另一个重点内容：**位串行架构设计，以乘法器和滤波器为例！**

相对于大规模的并行处理系统而言，位串行处理系统与其背道而驰，，但并不是说位串行处理系统一无是处，，在资源紧缺且通信带宽有限的情况下，位串行处理是个不错的选择。

课本上对乘法器给出两种截然不同的位级设计方法：1）脉动阵列，和 2）Horner 法则。我们从简单的脉动阵列法开始，脉动阵列在第七章中有过详细讨论，这里不再深入细节。

给出标准乘法器依赖图如下，

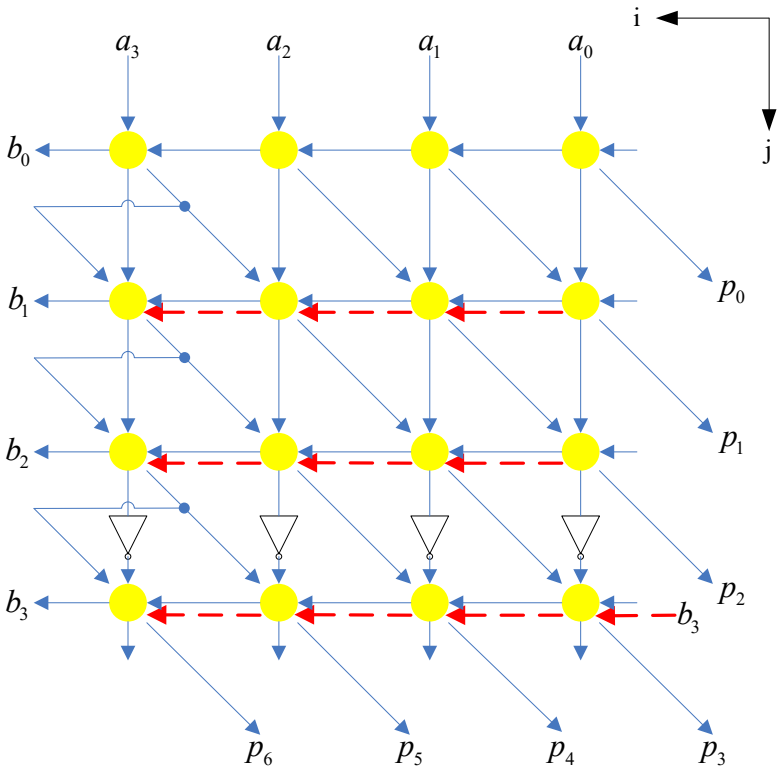


图 23　标准乘法器依赖图，对应图 8

看起来，图 23 的依赖图并不是非常的规则，不过这没关系，只要在具体硬件电路上稍微进行一下“手术”就能得到功能正确的位串行处理架构。

按课本顺序，先来试试例 13.4.1，依赖图如图 24，

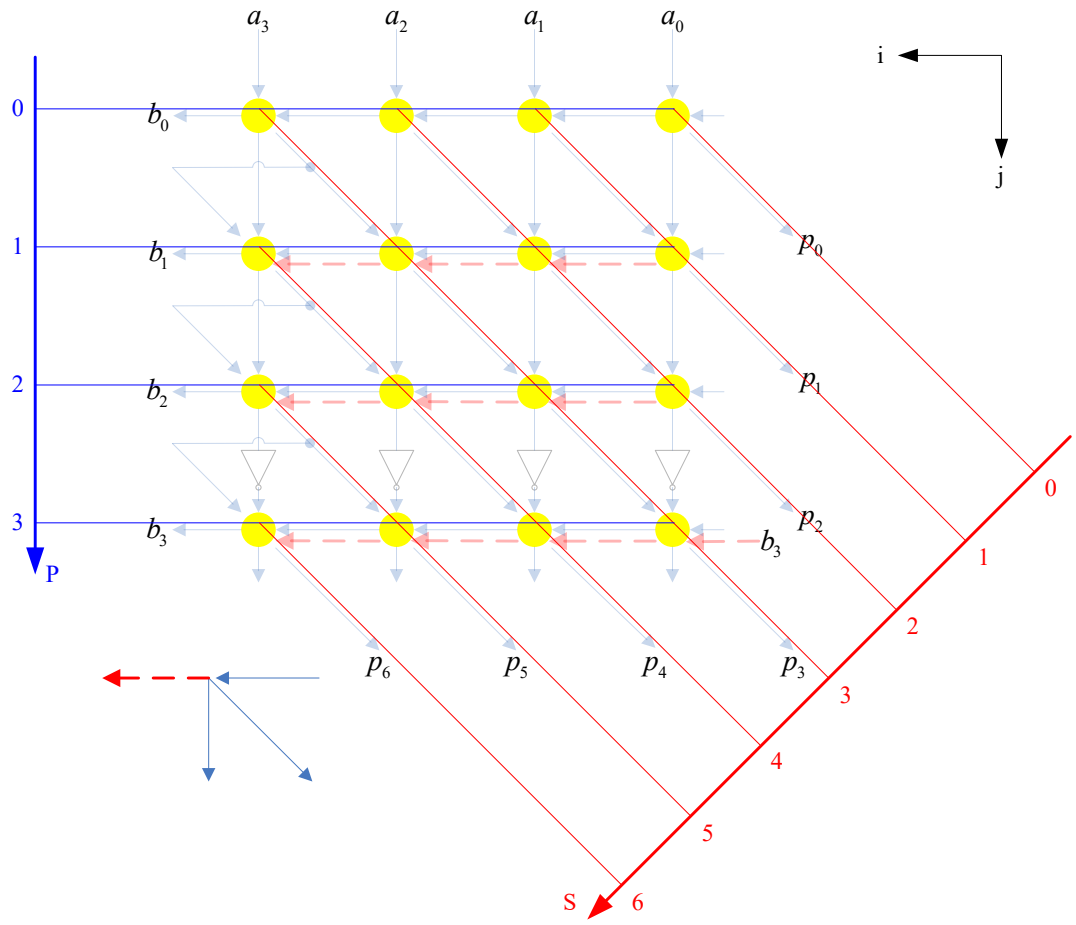


图 24　课本例 13.4.1 $S^T=[1,1]\wedge P^T=[0,1]$，试试看能得到什么硬件架构

根据给出的时间轴和处理器轴，计算边映射关系如下，

表格 1　图 24 脉动边映射表

e^T	$P^T e$	$S^T e$
$a\,(0,1)$	1	1
$b\,(1,0)$	0	1
$carry\,(1,0)$	0	1
$p\,(-1,1)$	1	0

根据边映射 表格 1，可画出脉动结构如

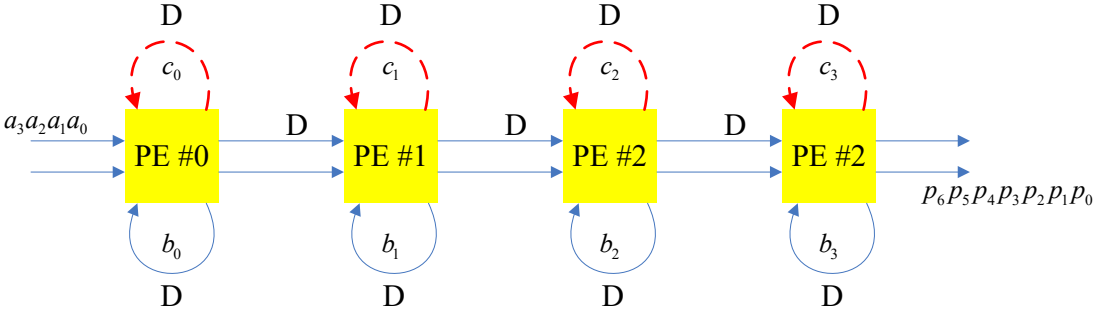


图 25　根据图 24 得出的脉动结构图

接下来的工作就很具有艺术性了，从图 25 设计出具体的硬件电路，，因为原始的依赖图有些些不规则，所以硬件设计中需要进行恰当的“手术”，

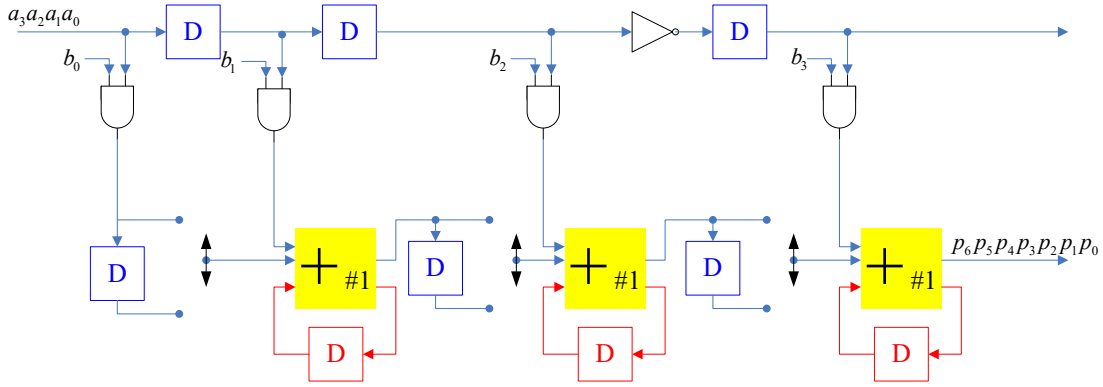


图 26　根据图 25 得出的硬件电路图

有兴趣者可以深入研究，通过编写代码观察仿真的时序，可帮助你理解如何构造硬件电路，同时也是对硬件设计能力的一种磨练。

对比图 26 和课本图 13-17，可以发现，这就是 Lyon's Multiplier，，虽然课本上是用 Horner 法则推导出 Lyon's Multiplier，但这里用脉动技术也能得到相同电路，很强大吧！

例 13.4.1 末尾，提出了一练习，要求设计具有位级流水的 Lyon 乘法器，，回头看图 26 可知，输出链路是一条组合逻辑路径，这也是系统的关键路径，如果能进一步将该输出链路用延时切断，那么就能实现位级流水的 Lyon，，为了达到这个目的，我对着图 24 的依赖图“苦思冥想”好久，发现可以试试这个情况

$$S^T = [1,2] \wedge P^T = [0,1] \tag{18}$$

如图 27　示

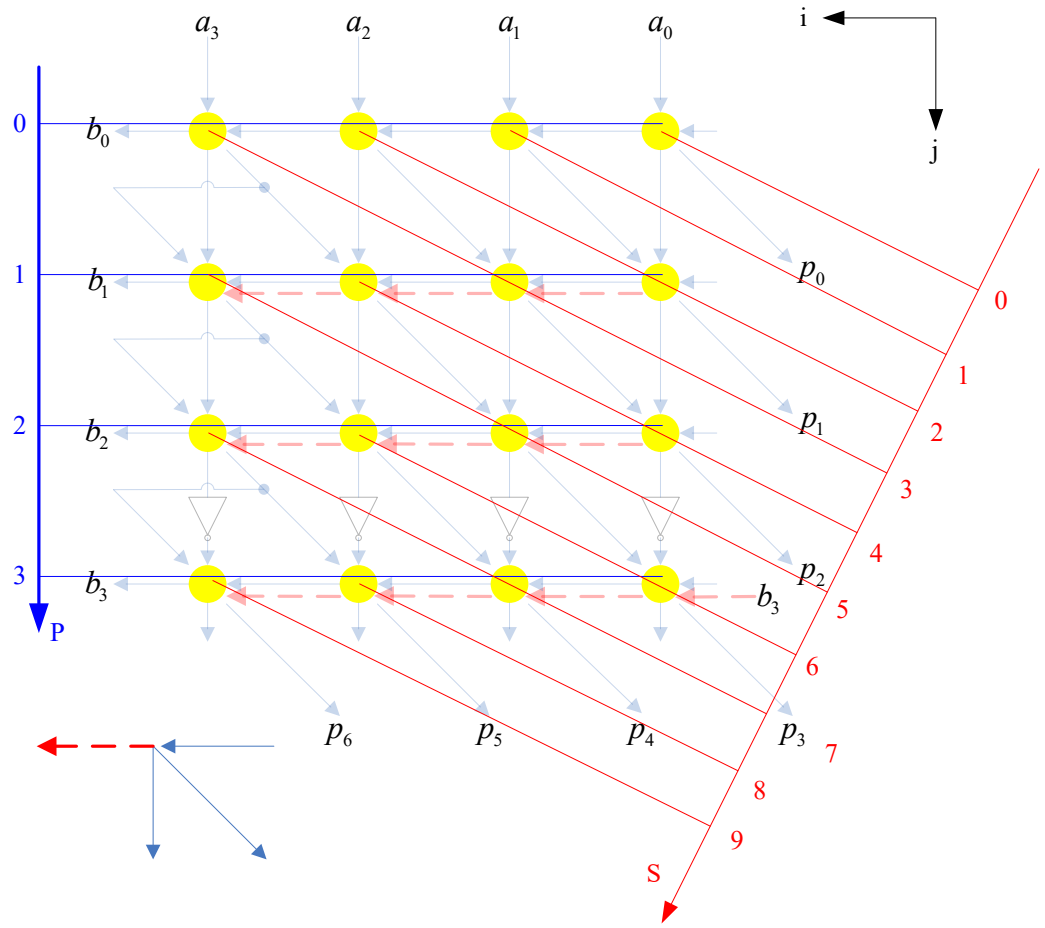


图 27　位级流水 Lyon 乘法器映射图

表格 2　课本习题 10 脉动边映射表

e^T	$P^T e$	$S^T e$
$a\,(0,1)$	1	2
$b\,(1,0)$	0	1
$carry\,(1,0)$	0	1
$p\,(-1,1)$	1	1

脉动结构图如下

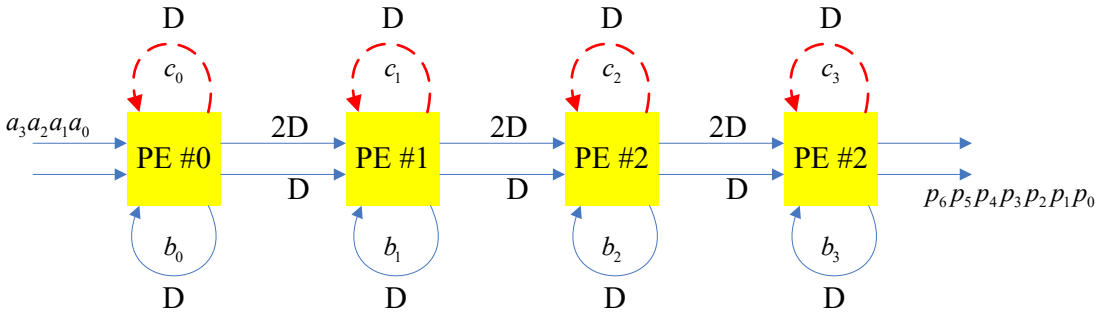


图 28　位级流水 Lyon 乘法器脉动结构

对比图 28 和 图 25，显然位级流水 Lyon 就是在图 25 的前馈割集上插入流水线的结果，应验了课本习题 10 的预言！图 28 硬件电路与图 26 相似，大家自己动手试试，（提示：割集流水线）！

再看下一个例子，
例 13.4.2，

$$P^T = [1,0] \wedge S^T = [0,1] \tag{19}$$

脉动映射如图 29，

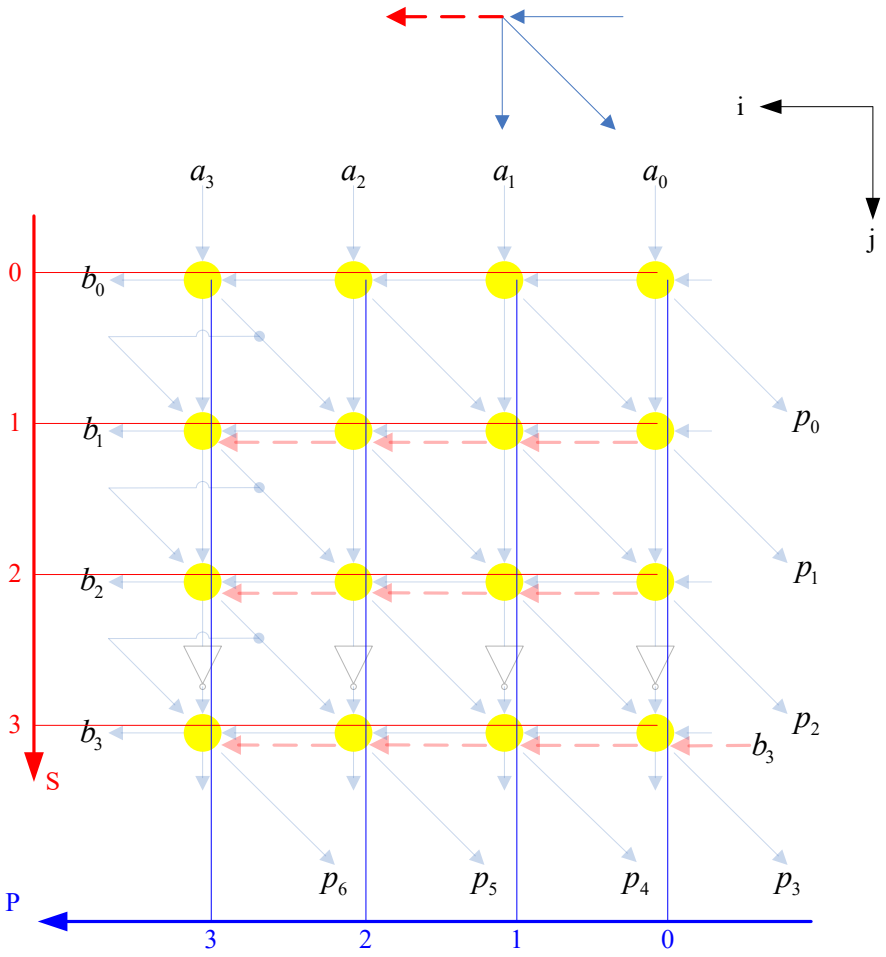


图 29

边映射表如下，

表格 3

e^T	$P^T e$	$S^T e$
$a\,(0,1)$	0	1
$b\,(1,0)$	1	0
$carry\,(1,0)$	1	0
$p\,(-1,1)$	-1	1

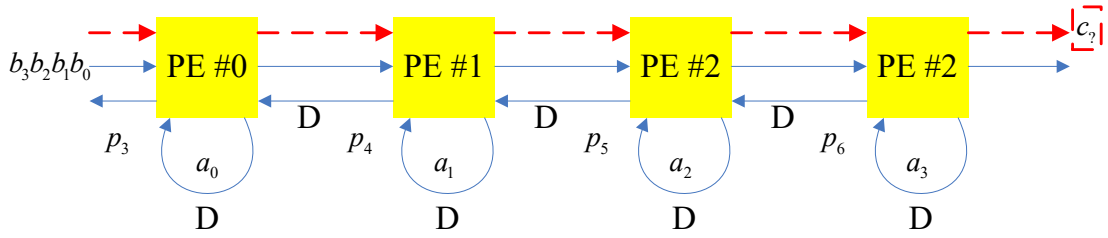


图 30

脉动结构如图 30,，结合图 30 和 图 29，构造具体硬件电路如课本图 13-18。

例 13.4.4 设计具有串行进位矢量合并的进位保留 Baugh-Wooley 乘法器，依赖图如图 31，

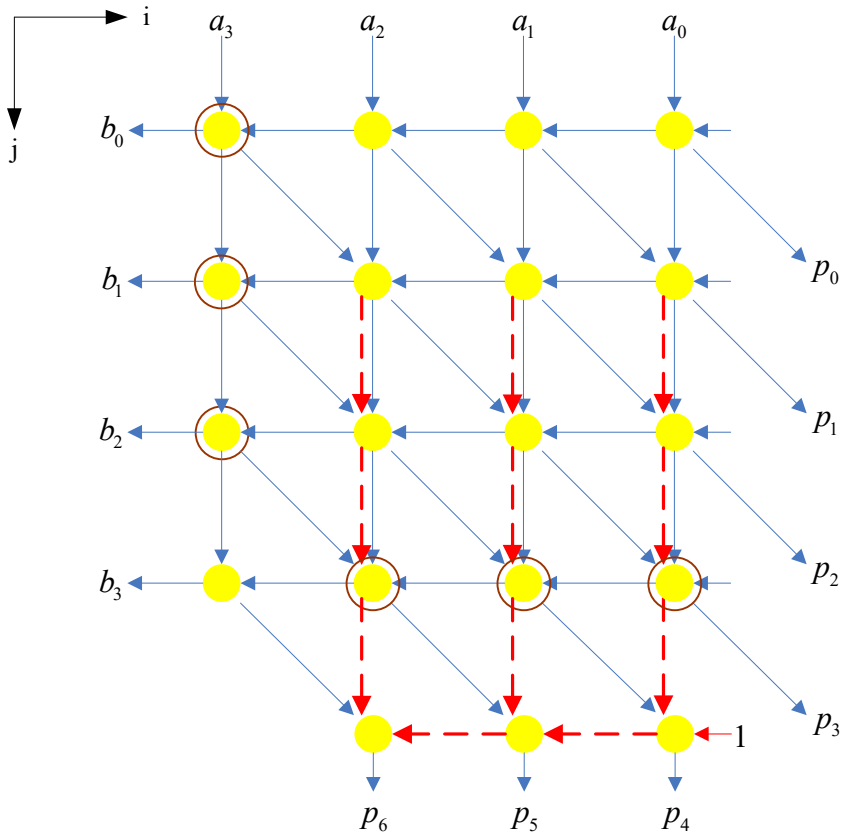


图 31　具有串行进位矢量合并的进位保留 Baugh-Wooley 乘法器依赖图

指定，

$$P^T = [1,0] \wedge S^T = [0,1] \tag{20}$$

则，脉动映射如

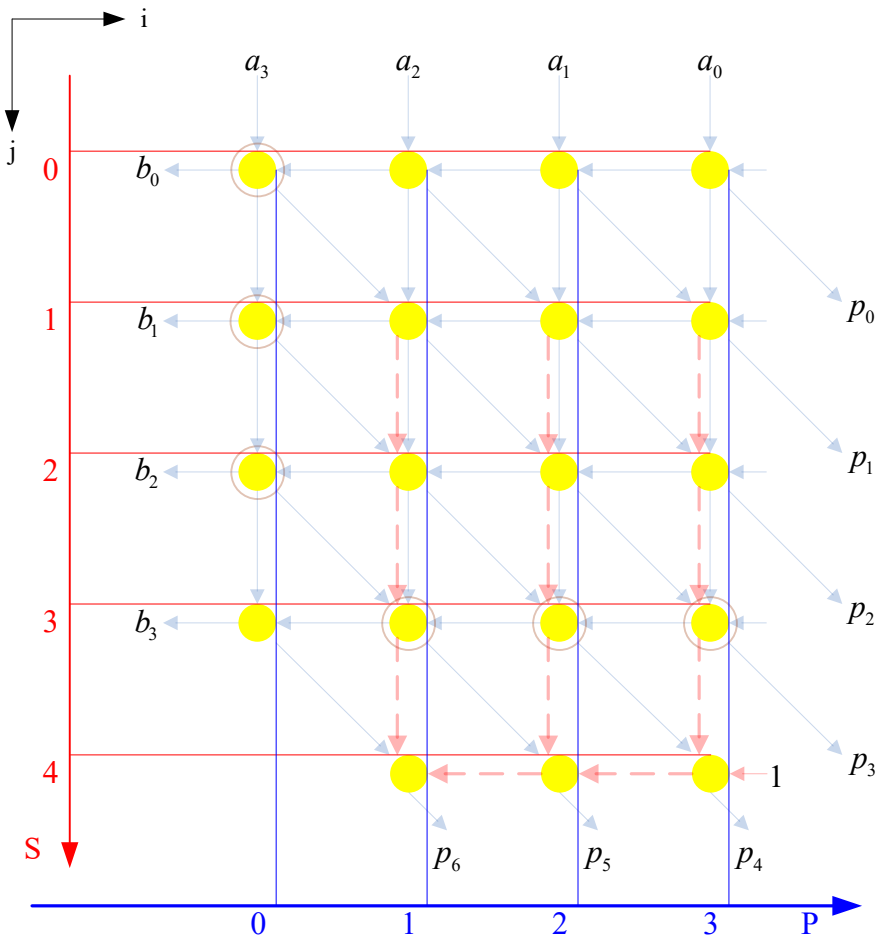
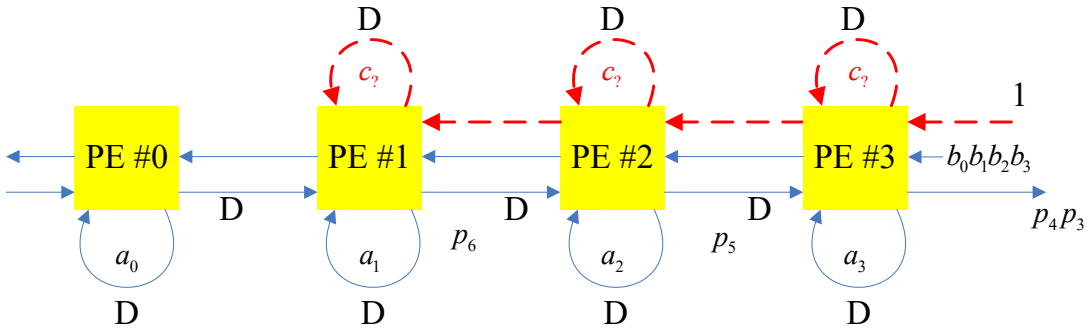


图 32　具有串行进位矢量合并的进位保留 Baugh-Wooley 乘法器脉动映射图

e^T	$P^T e$	$S^T e$
$a\,(0,1)$	0	1
$b\,(-1,0)$	-1	0
$carry\,(0,1)$ and $(-1,0)$	0 and -1	1 and 0
$p\,(1,1)$	1	1



总之总之，脉动设计是一项带有艺术性质的工作，，简单的脉动设计大部分同学都能掌握，但是对于那些不怎么规则，需要做些“手术”的，就不是人人都能轻松搞定的了。这里将课

本的例 13.4.3 和例 13.4.4 的后一部分留作练习，不要光是看课本上的设计结果，自己也动手试试，加上功能强大的仿真软件，我相信你能在曲折中弄懂如何进行复杂的脉动设计。通过对一个设计不断的调试和修改，你也就学会“如何来动手术了”!!
