

定点乘法器 设计

蒋小龙

2002. 12. 29

声 明

作此资料为本人个人行为，此资料版权为本人所有。

你可以任意使用，但你非经本人允许不得对此资料内容作任何修改。

你因使用此资料所带来任何收益，本人均不染指；因使用此资料所引起的任何不良后果，本人不承担任何形式的责任。

出版物引用，请注明！

蒋 小 龙

2002. 12. 29

目 录

声明	1
0、约定	5
1、无符号数一位乘法	7
2、符号数一位乘法	8
3、布思算法(Booth algorithm)	9
4、高基(High Radix)布思算法	10
5、迭代算法	14
6、乘法运算的实现——迭代	18
7、乘法运算的实现——阵列	20
8、乘加运算	24
9、设计示例1 —— 8位、迭代	26
1、实现方案1 —— 一位、无符号	26
2、实现方案2 —— 一位、布思	33
3、实现方案3 —— 二位	39
10、设计示例2 —— 16位、阵列	45
11、设计示例3 —— 32位、迭代、阵列	55
1、实现方案1 —— 乘、加一步走	56
2、实现方案2 —— 乘、加两步走	67
后记	77
个人介绍	79

0、约定

运算符:

+	对其两边的数据作加法操作;	A + B
-	从左边的数据中减去右边的数据;	A - B
-	对跟在其后的数据作取补操作,即用 0 减去跟在其后的数据;	- B
*	对其两边的数据作乘法操作;	A * B
&	对其两边的数据按位作与操作;	A & B
#	对其两边的数据按位作或操作;	A # B
@	对其两边的数据按位作异或操作;	A @ B
~	对跟在其后的数据作按位取反操作;	~ B
<<	以右边的数据为移位量将左边的数据左移;	A << B
\$	将其两边的数据按从左至右顺序拼接;	A \$ B

运算数据:

A	乘法操作数、位串, 位宽为 N;
B	乘法操作数、位串, 位宽为 M;
R	累加操作数、位串, 位宽为 Y;
D	乘法——累加运算结果、位串, 位宽为 (M+N)、或 Y, 取大者;
E	布思编码;
S	(加减) 运算结果, 结果宽度与二源操作数中位宽最大者同;
C	(无符号数运算) 进位;
X	(符号数运算) 扩展位;
T	考虑了进位 (C) 或扩展位 (X) 及结果 (S) 的结果, 即: $T = C \$ S$ 或 $T = X \$ S$;
$A_{[n]}$	数据 A 的第 n 位;
$A_{[m:n]}$	数据 A 的第 m 到 n 位;
$E_{\langle n \rangle}$	第 n 项编码项、编码值;
D_n	在状态 n 时, 数据 D 的值;

对运算符的约定只在文中论述的时候有效。

设计示例用 VerilogHDL 语言实现, 所有东西符合 VerilogHDL 语法。如有疑问, 请参阅 VerilogHDL 资料。

所有设计示例均通过仿真、综合。对不同的工具软件, 可能要作些许修改。

1、无符号数一位乘法

二进制乘法与十进制乘法的运算方法一样，只是此时应用二进制乘法规则：

$$0 * 0 = 0 ; 0 * 1 = 0 ; 1 * 0 = 0 ; 1 * 1 = 1$$

可以看到，二进制乘法规则正和逻辑与运算相同。

示例十进制乘法运算、二进制乘法运算如下：

十进制乘法运算：

$$\begin{array}{r}
 \text{被乘数} \quad 8765 \\
 \text{乘数} \quad \quad (*) 4321 \\
 \hline
 \quad \quad \quad 8765 \\
 \quad \quad 17530 \\
 \quad 26295 \\
 (+) 35060 \\
 \hline
 \text{积} \quad \quad 37873565
 \end{array}$$

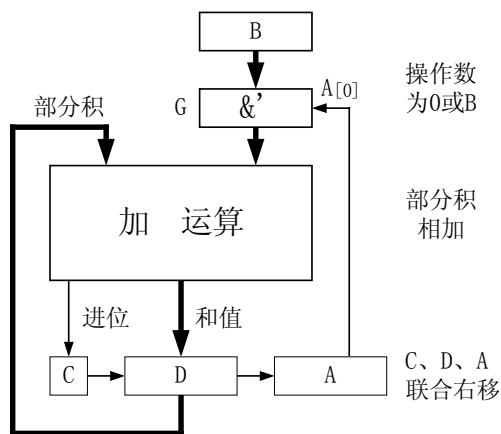
二进制乘法运算：

$$\begin{array}{r}
 \text{被乘数} \quad \quad \quad 1101 \\
 \text{乘数} \quad \quad \quad (*) 1011 \\
 \hline
 \quad \quad \quad 1101 \\
 \quad \quad 1101 \\
 \quad 0000 \\
 (+) 1101 \\
 \hline
 \text{积} \quad \quad 10001111
 \end{array}$$

对这种方式称之为手算，以区别于硬件实现。

手算时，将所有乘积项（被乘数与乘数中某一位的乘积）全都算出来，最后一并作加。这在硬件实现中称之为阵列实现，后面将有专门讨论。这种方式，速度快、资源耗费大，主要在时间要求较高的场合应用。

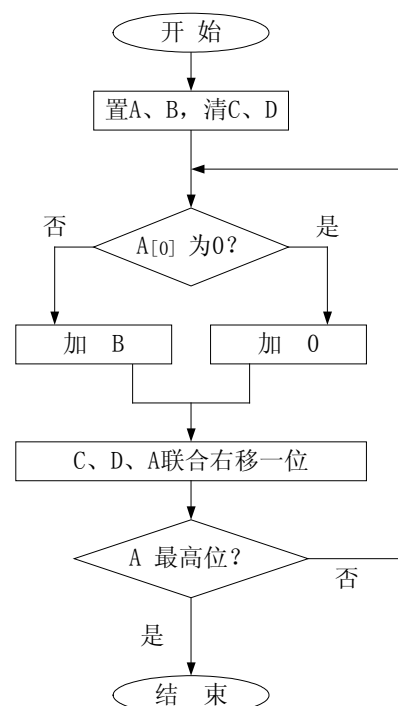
也可以每算出一乘积项，就加到乘积中，此时的积称作部分积。其原理如下左图，运算步骤如下右图。



手算时，乘积项与乘数相应位对齐（即将乘积项左移），加法运算宽度与被乘数位宽同。若硬件这样实现，则比较困难。

可以将部分积右移，将部分积高部分与乘积项相加，和值存于寄存器 D。考虑到二数相加，存在进位，用一寄存器保存。再将进位、和值、（前次）部分积低部分右移，得到本次运算的结果（部分积）。A 寄存器中已用过的位可以舍弃，将其右移。这样可使下次待用位始终处于最低位，简化了逻辑；其空出的高部分又正好可以存放部分积低部分。

这种实现方式称之为迭代，通常称移位——加方式。若乘数位宽为 N 位，它需要 2N 个时钟周期。



将上面的手算示例按硬件实现示例如下：

步 骤		C	D	A	说 明
置 初 值		0	0 0 0 0	<u>1</u> <u>0</u> <u>1</u> <u>1</u>	初始状态，置数 A、B，清 D
第 0 步	加操作			1 1 0 1	计算第 0 位；A _[0] 为 1，加 B
	和值	<u>0</u>	1 1 0 1		加法结果
	移 位		<u>0</u> 1 1 0	1 <u>1</u> <u>0</u> <u>1</u>	C、D、A 联合右移，得到本次部分积
第 1 步	加操作			1 1 0 1	计算第 1 位；A _[0] 为 1，加 B
	和值	<u>1</u>	0 0 1 1		加法结果
	移 位		<u>1</u> 0 0 1	1 1 <u>1</u> <u>0</u>	C、D、A 联合右移，得到本次部分积
第 2 步	加操作			0 0 0 0	计算第 2 位；A _[0] 为 0，加 0
	和值	<u>0</u>	1 0 0 1		加法结果
	移 位		<u>0</u> 1 0 0	1 1 1 <u>1</u>	C、D、A 联合右移，得到本次部分积
第 3 步	加操作			1 1 0 1	计算第 3 位；A _[0] 为 1，加 B
	和值	<u>1</u>	0 0 0 1		加法结果
	移 位		<u>1</u> 0 0 0	1 1 1 1	C、D、A 联合右移，得到本次部分积
结 束			1 0 0 0	1 1 1 1	乘积，D 中为高部分、A 中为低部分

其中的加法器设计可以参见本人大作《算术逻辑部件设计》。

2、符号数一位乘法

符号数相乘，可以将符号与绝对值分别处理，即：绝对值相乘，符号异或。

如果二数是用原码表示，本方法非常好，它的实现没有问题。如果二数是用反码表示，则只需将操作数按条件取反即行，实现也没有问题。可如果二数为补码表示，则需一取补逻辑，考虑时延、资源耗费，那就太差劲了。

可以考虑直接对符号数处理！

乘数 A 可表示为：

$$A = -A_{[N-1]} * 2^{N-1} + A_{[N-2]} * 2^{N-2} + \dots + A_{[1]} * 2^1 + A_{[0]} * 2^0 \quad (2-1)$$

则：

$$D = A * B \\ = (-A_{[N-1]} * 2^{N-1} + A_{[N-2]} * 2^{N-2} + \dots + A_{[1]} * 2^1 + A_{[0]} * 2^0) * B \quad (2-2)$$

$$= -A_{[N-1]} * B * 2^{N-1} + A_{[N-2]} * B * 2^{N-2} + \dots + A_{[1]} * B * 2^1 + A_{[0]} * B * 2^0 \quad (2-3)$$

可见，在对最高位（符号位）计算时，需作（相对于其它位是全作加的）特殊操作——减。据此思路，参照前面对无符号数乘法实现的讨论，符号数乘法也就不难实现。

由于在对最高位计算时，需作特殊操作处理，逻辑复杂也就不可避免。若能将这特殊操作免去可就好了！下节讨论的布斯算法即可实现此目标，它将最高位（符号位）作为数据位考虑。

3、布思算法(Booth algorithm)

对式(2-1):

$$\begin{aligned}
 A &= -A_{[N-1]} * 2^{N-1} + A_{[N-2]} * 2^{N-2} + \dots + A_{[1]} * 2^1 + A_{[0]} * 2^0 \\
 &= -A_{[N-1]} * 2^{N-1} + (2A_{[N-2]} - A_{[N-2]}) * 2^{N-2} + (2A_{[N-3]} - A_{[N-3]}) * 2^{N-3} + \dots + \\
 &\quad (2A_{[2]} - A_{[2]}) * 2^2 + (2A_{[1]} - A_{[1]}) * 2^1 + (2A_{[0]} - A_{[0]}) * 2^0 \tag{3-1}
 \end{aligned}$$

$$\begin{aligned}
 &= -A_{[N-1]} * 2^{N-1} + A_{[N-2]} * 2^{N-1} - A_{[N-2]} * 2^{N-2} + A_{[N-3]} * 2^{N-2} - A_{[N-3]} * 2^{N-3} + \dots + \\
 &\quad A_{[2]} * 2^3 - A_{[2]} * 2^2 + A_{[1]} * 2^2 - A_{[1]} * 2^1 + A_{[0]} * 2^1 - A_{[0]} * 2^0 \tag{3-2}
 \end{aligned}$$

$$\begin{aligned}
 &= (-A_{[N-1]} + A_{[N-2]}) * 2^{N-1} + (-A_{[N-2]} + A_{[N-3]}) * 2^{N-2} + \dots + \\
 &\quad (-A_{[2]} + A_{[1]}) * 2^2 + (-A_{[1]} + A_{[0]}) * 2^1 + (-A_{[0]} + 0) * 2^0 \tag{3-3}
 \end{aligned}$$

$$= \sum_{n=0, A_{[n-1]}=0}^{N-1} ((-A_{[n]} + A_{[n-1]}) * 2^n) \tag{3-4}$$

$$= \sum_{n=0}^{N-1} (E_{\langle n \rangle} * 2^n) \tag{3-5}$$

其中:

$$E_{\langle n \rangle} = -A_{[n]} + A_{[n-1]} \quad (0 \leq n \leq (N-1), A_{[-1]} = 0) \tag{3-6}$$

其值如下表所示:

$A_{[n]}$	$A_{[n-1]}$	$E_{\langle n \rangle}$
0	0	$-0 + 0 = 0$
0	1	$-0 + 1 = 1$
1	0	$-1 + 0 = -1$
1	1	$-1 + 1 = 0$

则:

$$\begin{aligned}
 D &= A * B \\
 &= \left(\sum_{n=0}^{N-1} (E_{\langle n \rangle} * 2^n) \right) * B \tag{3-7}
 \end{aligned}$$

$$= \sum_{n=0}^{N-1} ((E_{\langle n \rangle} * B) * 2^n) \tag{3-8}$$

这就是布思算法!

可见, 在这种处理方式中, 对最高位的处理并不像式(2-3)那样需作特殊要求。它——编码逻辑——考

考虑本位及相邻低位，确定操作为加或减、运算量为 0 或 B。总的编码项为 N 项，乘积项（被乘数与由乘数所得某一编码项值的乘积。此表达方式可能更具一般性！）为 N 项。

当然，在这种处理方式中，每一步都要编码，相对于式(2-3)有责众事实，典型的“侵犯公益”，并使（门）延迟增加，甚至会得不偿失（但特殊处理所带来的延迟也不低）。实际应用中，主要看中它“一视同仁”的“优秀品质”，及其高基运算能力。

对无符号数乘法：

乘数 A 可表示为

$$A = A_{[N-1]} * 2^{N-1} + A_{[N-2]} * 2^{N-2} + \dots + A_{[1]} * 2^1 + A_{[0]} * 2^0 \quad (3-9)$$

$$= \sum_{n=0}^{N-1} (A_{[n]} * 2^n) \quad (3-10)$$

则：

$$D = A * B$$

$$= \left(\sum_{n=0}^{N-1} (A_{[n]} * 2^n) \right) * B \quad (3-11)$$

$$= \sum_{n=0}^{N-1} ((A_{[n]} * B) * 2^n) \quad (3-12)$$

可见，式(3-8)、式(3-12)表达形式是相同的，在此又将无符号数与符号数的乘法运统一起来。

4、高基（High Radix）布思算法

上节中所谈论的布思算法以 2 为基。它形成 N 项编码项、乘积项，考虑本位及相邻低位，确定运算量是 0 或 B。还可进一步深入研究，以实现基 4 布思算法。

先考虑符号数相乘：

我们知道：以补码表示的二进制数据，扩展其最高位，并无影响。

乘数 A 位宽为 N，若 N 为奇数，将 A 作符号扩展为 A'，使其位宽为偶数。设定：经过处理以后，乘数 A' 宽度为 H，H 为偶数且不得小于 N。

则乘数 A' 可表示为：

$$A' = -A'_{[H-1]} * 2^{H-1} + A'_{[H-2]} * 2^{H-2} + \dots + A'_{[1]} * 2^1 + A'_{[0]} * 2^0 \quad (4-1)$$

$$= (-A'_{[H-1]} + A'_{[H-2]}) * 2^{H-1} + (-A'_{[H-2]} + A'_{[H-3]}) * 2^{H-2} + \dots +$$

$$(-A'_{[2]} + A'_{[1]}) * 2^2 + (-A'_{[1]} + A'_{[0]}) * 2^1 + (-A'_{[0]} + 0) * 2^0 \quad (4-2)$$

$$= (-2A'_{[H-1]} + A'_{[H-2]} + A'_{[H-3]}) * 2^{H-2} + (-2A'_{[H-3]} + A'_{[H-4]} + A'_{[H-5]}) * 2^{H-4} + \dots +$$

$$(-2A'_{[3]} + A'_{[2]} + A'_{[1]}) * 2^2 + (-2A'_{[1]} + A'_{[0]} + 0) * 2^0 \quad (4-3)$$

$$= \sum_{n=0, A'_{[n-1]}=0}^{(H/2)-1} ((-2A'_{[2n+1]} + A'_{[2n]} + A'_{[2n-1]}) * 2^{2n}) \quad (4-4)$$

$$= \sum_{n=0}^{(H/2)-1} (E_{\langle n \rangle} * 4^n) \quad (4-5)$$

其中:

$$E_{\langle n \rangle} = -2A'_{[2n+1]} + A'_{[2n]} + A'_{[2n-1]} \quad (0 \leq n \leq ((H/2)-1), A'_{[-1]} = 0) \quad (4-6)$$

其值如下表所示:

$A'_{[2n+1]}$	$A'_{[2n]}$	$A'_{[2n-1]}$	$E_{\langle n \rangle}$
0	0	0	$-0 + 0 + 0 = 0$
0	0	1	$-0 + 0 + 1 = 1$
0	1	0	$-0 + 1 + 0 = 1$
0	1	1	$-0 + 1 + 1 = 2$
1	0	0	$-2 + 0 + 0 = -2$
1	0	1	$-2 + 0 + 1 = -1$
1	1	0	$-2 + 1 + 0 = -1$
1	1	1	$-2 + 1 + 1 = -0$

则:

$$D = A * B = A' * B$$

$$= \left(\sum_{n=0}^{(H/2)-1} (E_{\langle n \rangle} * 4^n) \right) * B \quad (4-7)$$

$$= \sum_{n=0}^{(H/2)-1} ((E_{\langle n \rangle} * B) * 4^n) \quad (4-8)$$

此即(符号数)基4布思算法!也称为符号数二位乘法。

可以看到:基4布思编码一次考虑了三位:本位、相邻高位、相邻低位;处理了两位,确定运算量0、1B、2B,形成(H/2)项编码项、乘积项。对于2B的实现,只需要将B左移一位。因此,不管从那方面来说,基4算法方便又快捷。而基2算法一次只考虑了两位、处理了一位,形成N项编码项、乘积项,只是方便而已。

对于无符号数,将其高位作0扩展,并无影响。

乘数A位宽为N,若N为奇数,将A作0扩展为A',使其位宽为偶数。设定:经过处理以后,乘数A'宽度为H,H为偶数、($N \leq H \leq (N+1)$)。

则乘数A'可表示为:

$$A' = A'_{[H-1]} * 2^{H-1} + A'_{[H-2]} * 2^{H-2} + \dots + A'_{[1]} * 2^1 + A'_{[0]} * 2^0 \quad (4-9)$$

$$= (2A'_{[H-1]} + A'_{[H-2]}) * 2^{H-2} + (2A'_{[H-3]} + A'_{[H-4]}) * 2^{H-4} + \dots + (2A'_{[3]} + A'_{[2]}) * 2^2 + (2A'_{[1]} + A'_{[0]}) * 2^0 \quad (4-10)$$

$$= \sum_{n=0}^{(H/2)-1} ((2A'_{[2n+1]} + A'_{[2n]}) * 2^{2n}) \quad (4-11)$$

$$= \sum_{n=0}^{(H/2)-1} (E'_{\langle n \rangle} * 4^n) \quad (4-12)$$

其中:

$$E'_{\langle n \rangle} = 2A'_{[2n+1]} + A'_{[2n]} \quad (0 \leq n \leq (H/2)-1) \quad (4-13)$$

其值如下表所示:

$A'_{[2n+1]}$	$A'_{[2n]}$	$E'_{\langle n \rangle}$
0	0	$0 + 0 = 0$
0	1	$0 + 1 = 1$
1	0	$2 + 0 = 2$
1	1	$2 + 1 = 3$

可见, 处理中出现了3, 对3的处理非常困难。可以考虑将3视作加(4-1), 而4可以认为是向(相邻)高(二)位借位, 高(二)位处理时需考虑低位借位: 加之。即:

$$E''_{\langle n \rangle} = 2A'_{[2n+1]} + A'_{[2n]} + L_{\langle n-1 \rangle} \quad (0 \leq n \leq (H/2)-1, L_{\langle -1 \rangle} = 0) \quad (4-14)$$

综合考虑可得如下值表:

$A'_{[2n+1]}$	$A'_{[2n]}$	$L_{\langle n-1 \rangle}$	$E''_{\langle n \rangle}$	$E_{\langle n \rangle}$	$L_{\langle n \rangle}$
0	0	0	$0 + 0 + 0 = 0$	0	0
0	0	1	$0 + 0 + 1 = 1$	1	0
0	1	0	$0 + 1 + 0 = 1$	1	0
0	1	1	$0 + 1 + 1 = 2$	2	0
1	0	0	$2 + 0 + 0 = 2$	2	0
1	0	1	$2 + 1 + 0 = 3$	-1	1
1	1	0	$2 + 0 + 1 = 3$	-1	1
1	1	1	$2 + 1 + 1 = 4$	-0	1

由上值表可得出, 借位:

$$L_{\langle n \rangle} = A'_{[2n+1]} \& A'_{[2n]} \# A'_{[2n+1]} \& L_{\langle n-1 \rangle} \quad (0 \leq n \leq (H/2)-1, L_{\langle -1 \rangle} = 0) \quad (4-15)$$

$$= G_{\langle n \rangle} \# P_{\langle n \rangle} \& L_{\langle n-1 \rangle} \quad (0 \leq n \leq (H/2)-1, L_{\langle -1 \rangle} = 0) \quad (4-16)$$

其中:

$$G_{\langle n \rangle} = A'_{[2n+1]} \& A'_{[2n]} \quad (0 \leq n \leq (H/2)-1)$$

$$P_{\langle n \rangle} = A'_{[2n+1]} \quad (0 \leq n \leq (H/2)-1)$$

A' 值:

$$A' = \sum_{n=0}^{(H/2)-1} (E_{\langle n \rangle} * 4^n) \quad (4-17)$$

式中 $E_{\langle n \rangle}$ 值如上表所示。

则:

$$D = A * B = A' * B$$

$$= \left(\sum_{n=0}^{(H/2)-1} (E_{\langle n \rangle} * 4^n) \right) * B \quad (4-18)$$

$$= \sum_{n=0}^{(H/2)-1} ((E_{\langle n \rangle} * B) * 4^n) \quad (4-19)$$

式中 $E_{\langle n \rangle}$ 值如上表所示。

这就是（无符号数）基 4 布思算法！也称为无符号数二位乘法。

由式(4-16)可以非常明显地看出：借位传播。

如果用迭代方式，则用一位寄存器保存此借位，也非常简单、方便。但最高两位可能会出现借位，增加一次特殊运算是可能的，对此需特别注意。

在阵列乘法里，其解决则有点困难。对其具体实施，则会发现代价太大，失却布思算法快速、简单的本意。

无符号数二位乘法与符号数二位乘法，只是在布思编码上有区别，其余全都相同。可以考虑将无符号数作 0 扩展后视作符号数，这就非常简单，问题迎刃而解。并且，这种方法最为方便、省事。但扩展以后，执行的加法操作可能增加一次。其实，这两兄弟耗时（钟脉冲）没有差别，彼此彼此。

对于更高基的布思算法，无多大实用价值。例如：对于符号数基 8

$$E_{\langle n \rangle} = -4A'_{[3n+2]} + 2A'_{[3n+1]} + A'_{[3n]} + A'_{[3n-1]} \\ (0 \leq n \leq (H/3)-1), H \text{ 为 } 3 \text{ 的倍数且不得小于数据 } A \text{ 的宽度}, A'_{[-1]} = 0 \quad (4-20)$$

这里出现了 3。要实现可就难了！

符号数基 16 稍好一点，但不如就直接用基 4 算法来得方便、简单。

5、迭代算法

我们在前面的详细讨论了布思算法，现在讨论乘法运算的迭代算法。

先讨论**无符号数一位乘法**，参见式(3-10)：

$$A = A_{[(N-1):0]} = A_{N-1} = \sum_{n=0}^{N-1} (A_{[n]} * 2^n) \quad (5-1)$$

更一般的：

$$A_n = A_{[n:0]} = \sum_{r=0}^n (A_{[r]} * 2^r) \quad (0 \leq n \leq (N-1)) \quad (5-2)$$

$$A_n = A_{n-1} + (A_{[n]} * 2^n) \quad (0 \leq n \leq (N-1), A_{-1} = 0) \quad (5-3)$$

此处的 $A_{-1} = 0$ 是指迭代时 A 的初值为 0。

则：

$$D_n = A_n * B \quad (0 \leq n \leq (N-1)) \quad (5-4)$$

$$= (A_{n-1} + (A_{[n]} * 2^n)) * B \quad (0 \leq n \leq (N-1), A_{-1} = 0) \quad (5-5)$$

$$= (A_{n-1} * B) + ((A_{[n]} * 2^n) * B) \quad (0 \leq n \leq (N-1), A_{-1} = 0) \quad (5-6)$$

$$= (A_{n-1} * B) + ((A_{[n]} * B) * 2^n) \quad (0 \leq n \leq (N-1), A_{-1} = 0) \quad (5-7)$$

$$= D_{n-1} + ((A_{[n]} * B) * 2^n) \quad (0 \leq n \leq (N-1), D_{-1} = 0) \quad (5-8)$$

此处的 $D_{-1} = 0$ 是将 D 初始化为 0、或置初值为 0。

这就是无符号数一位乘法的迭代形式！

对此式的实现，考虑到 D_{n-1} 的最大有效宽度为 $(M+N-1)$ ，需用 $(M+N-1)$ 宽的加法器。应该考虑减少资源耗费！

二数相乘，若其一操作数为一位，则积宽定为另一操作数位宽；二（非一位）数相乘，其积宽为二乘法操作数位宽之和。（*如何证明！* 本资料直接使用）

参见式(5-4)、(5-2)，得：

$$D_n = A_n * B \quad (0 \leq n \leq (N-1)) \quad (5-9)$$

$$= A_{[n:0]} * B_{[(M-1):0]} \quad (0 \leq n \leq (N-1)) \quad (5-10)$$

$$= D_{[(M+n):0]} \quad (0 \leq n \leq (N-1)) \quad (5-11)$$

参见式(5-7)、(5-2)、(5-11)，得：

$$D_n = (A_{n-1} * B) + ((A_{[n]} * B) * 2^n) \quad (0 \leq n \leq (N-1), A_{-1} = 0) \quad (5-12)$$

$$= (A_{[(n-1):0]} * B_{[(M-1):0]}) + ((A_{[n]} * B) * 2^n) \quad (0 \leq n \leq (N-1), A_{[-1]} = 0) \quad (5-13)$$

$$= D_{[(M+n-1):0]} + ((A_{[n]} * B) * 2^n) \quad (0 \leq n \leq (N-1)) \quad (5-14)$$

所以:

$$D_n = D_{[(M+n):0]} = D_{[(M+n-1):0]} + ((A_{[n]} * B) * 2^n) \quad (0 \leq n \leq (N-1)) \quad (5-15)$$

注意: 此处 $D_{[(M+n-1):0]}$ 为前次运算结果, $D_{[(M+n):0]}$ 为本次运算结果。

因系无符号数, 可将 $D_{[(M+n-1):0]}$ 拆成两部分, 如下示:

$$D_{[(M+n):0]} = ((D_{[(M+n-1):n]} * 2^n) + D_{[(n-1):0]}) + ((A_{[n]} * B) * 2^n) \quad (5-16)$$

$$= ((D_{[(M+n-1):n]} + (A_{[n]} * B)) * 2^n) + D_{[(n-1):0]} \quad (5-17)$$

$$= ((D_{[(M+n-1):n]} + (A_{[n]} * B)) \ll n) + D_{[(n-1):0]} \quad (5-18)$$

$$= ((D_{[(M+n-1):n]} + (A_{[n]} * B)) \ll n) \# D_{[(n-1):0]} \quad (5-19)$$

$$= (D_{[(M+n-1):n]} + (A_{[n]} * B)) \$ D_{[(n-1):0]} \quad (5-20)$$

令

$$T_n = D_{[(M+n-1):n]} + (A_{[n]} * B) \quad (5-21)$$

式中 $D_{[(M+n-1):n]}$ 、 $(A_{[n]} * B)$ 均为 M 位, 其和值 S 为 M 位。考虑进位, T_n 为 $(M+1)$ 位。

所以:

$$D_{[(M+n):0]} = T_n \$ D_{[(n-1):0]} \quad (5-22)$$

$$= C \$ S_{[(M-1):0]} \$ D_{[(n-1):0]} \quad (5-23)$$

$$= (C \$ S_{[(M-1):1]}) \$ (S_{[0]} \$ D_{[(n-1):0]}) \quad (5-24)$$

$$= D_{[(M+n):(n+1)]} \$ D_{[n:0]} \quad (5-25)$$

以上各式皆限定: $0 \leq n \leq (N-1)$; D 初始化为 0; $D_{[-1:0]}$ 不存在, 其值为 0。前面各式中 $D_{[(n-1):0]}$ 均为前次计算结果低部分, $D_{[(M+n-1):n]}$ 均为前次计算结果高部分, 而式 (5-24) 中各部分则为本次运算结果。

这才实用! 它只需用 M 位宽的加法器。回顾第一节的无符号数一位乘法硬件实现, 其在这里得到了理论上的确认。

对于**符号数一位乘法**, 参见前面关于无符号数一位乘法迭代算法的推导。由式 (3-5)

$$A = A_{[(N-1):0]} = A_{N-1} = \sum_{n=0}^{N-1} (E_{\langle n \rangle} * 2^n) \quad (5-26)$$

其中:

$$E_{\langle n \rangle} = -A_{[n]} + A_{[n-1]} \quad (0 \leq n \leq (N-1), A_{[-1]} = 0) \quad (5-27)$$

可得:

$$A_n = A_{[n:0]} = \sum_{r=0}^n (E_{\langle r \rangle} * 2^r) \quad (0 \leq n \leq (N-1)) \quad (5-28)$$

$$A_n = A_{n-1} + (E_{\langle n \rangle} * 2^n) \quad (0 \leq n \leq (N-1), A_{-1} = 0) \quad (5-29)$$

$$D_n = A_n * B = D_{n-1} + ((E_{[n]} * B) * 2^n) \quad (0 \leq n \leq (N-1), D_{-1} = 0) \quad (5-30)$$

$$D_n = D_{[(M+n):0]} = D_{[(M+n-1):0]} + ((E_{\langle n \rangle} * B) * 2^n) \quad (0 \leq n \leq (N-1)) \quad (5-31)$$

此处的 $D_{-1} = 0$ 是将 D 初始化为 0、或置初值为 0。 $D_{[(M+n-1):0]}$ 为前次运算结果, $D_{[(M+n):0]}$ 为本次运算结果。

对式 (2-1)

$$A = -A_{[N-1]} * 2^{N-1} + A_{[N-2]} * 2^{N-2} + \dots + A_{[1]} * 2^1 + A_{[0]} * 2^0 \quad (5-32)$$

$$= (-A_{[N-1]} * 2^{N-1} + A_{[N-2]} * 2^{N-2} + \dots + A_{[v]} * 2^v) +$$

$$(A_{[v-1]} * 2^{v-1} + \dots + A_{[1]} * 2^1 + A_{[0]} * 2^0) \quad (5-33)$$

$$= A_{[(N-1):v]} * 2^v + A_{[(v-1):0]} \quad (5-34)$$

在此，将一符号数拆成两部分：符号数 $A_{[(N-1):v]}$ 、无符号数 $A_{[(v-1):0]}$ 。
以上各式限定： $0 \leq v \leq (N-1)$ ； $A_{[-1]}$ 不存在，其值为 0。

由此，对式(5-31)可作如下变换：

$$D_{[(M+n):0]} = ((D_{[(M+n-1):n]} * 2^n) + D_{[(n-1):0]}) + ((E_{[n]} * B) * 2^n) \quad (5-35)$$

$$= ((D_{[(M+n-1):n]} + (E_{[n]} * B)) * 2^n) + D_{[(n-1):0]} \quad (5-36)$$

$$= (D_{[(M+n-1):n]} + (E_{[n]} * B)) \$ D_{[(n-1):0]} \quad (5-36)$$

令

$$T_n = D_{[(M+n-1):n]} + (E_{[n]} * B) \quad (5-40)$$

其中的 $D_{[(M+n-1):n]}$ 、 $(E_{[n]} * B)$ 被解释为 M 位符号数，其和值 S 为 M 位；而 $D_{[(n-1):0]}$ 被解释为无符号数。考虑可能溢出，将 $D_{[(M+n-1):n]}$ 、 $(E_{[n]} * B)$ 作符号扩展一位。因此， T_n 为 $(M+1)$ 位。

所以：

$$D_{[(M+n):0]} = T_n \$ D_{[(n-1):0]} \quad (5-41)$$

$$= X \$ S_{[(M-1):0]} \$ D_{[(n-1):0]} \quad (5-42)$$

$$= (X \$ S_{[(M-1):1]}) \$ (S_{[0]} \$ D_{[(n-1):0]}) \quad (5-43)$$

$$= D_{[(M+n):(n+1)]} \$ D_{[n:0]} \quad (5-44)$$

以上各式皆限定： $0 \leq n \leq (N-1)$ ； D 初始化为 0； $D_{[-1:0]}$ 不存在，其值为 0。以上各式中 $D_{[(n-1):0]}$ 均为前次计算结果低部分， $D_{[(M+n-1):n]}$ 均为前次计算结果高部分，而式(5-44)中各部分则为本次运算结果。

它需用 M 位宽的加法器，扩展位另行处理。

对符号数二位乘法，参见式(4-8)

$$A' = A'_{[(H-1):0]} = \sum_{n=0}^{(H/2)-1} (E_{\langle n \rangle} * 4^n) \quad (5-45)$$

其中：

$$E_{\langle n \rangle} = -2A'_{[2n+1]} + A'_{[2n]} + A'_{[2n-1]} \quad (0 \leq n \leq ((H/2)-1), A'_{[-1]} = 0) \quad (5-46)$$

可得：

$$A'_n = A'_{[(2n+1):0]} = \sum_{r=0}^n (E_{\langle r \rangle} * 4^r) \quad (0 \leq n \leq ((H/2)-1)) \quad (5-47)$$

$$A'_n = A'_{n-1} + (E_{\langle n \rangle} * 4^n) \quad (0 \leq n \leq ((H/2)-1), A'_{-1} = 0) \quad (5-48)$$

$$D_n = A_n * B = A'_n * B \quad (5-49)$$

$$= (A'_{n-1} + (E_{\langle n \rangle} * 4^n)) * B \quad (0 \leq n \leq ((H/2)-1), A'_{-1} = 0) \quad (5-50)$$

$$= (A'_{n-1} * B) + ((E_{\langle n \rangle} * 4^n) * B) \quad (0 \leq n \leq ((H/2)-1), A'_{-1} = 0) \quad (5-51)$$

$$= (A'_{n-1} * B) + ((E_{<n>} * B) * 4^n) \quad (0 \leq n \leq ((H/2)-1), A'_{-1} = 0) \quad (5-52)$$

$$= D_{n-1} + ((E_{<n>} * B) * 4^n) \quad (0 \leq n \leq ((H/2)-1), D_{-1} = 0) \quad (5-53)$$

此处的 $D_{-1} = 0$ 是将 D 初始化为 0、或置初值为 0。

此即符号数乘法二位乘法的迭代形式！

参见式(5-49)、(5-47)，得：

$$D_n = A_n * B = A'_n * B \quad (0 \leq n \leq ((H/2)-1)) \quad (5-54)$$

$$= A'_{[(2n+1):0]} * B_{[(M-1):0]} \quad (0 \leq n \leq ((H/2)-1), A'_{[-1]} = 0) \quad (5-55)$$

$$= D_{[(M+2n+1):0]} \quad (0 \leq n \leq ((H/2)-1)) \quad (5-56)$$

参见式(5-52)、(5-47)、(5-56)，得：

$$D_n = (A'_{n-1} * B) + ((E_{<n>} * B) * 4^n) \quad (0 \leq n \leq ((H/2)-1), A'_{-1} = 0) \quad (5-57)$$

$$= (A'_{[(2n-1):0]} * B_{[(M-1):0]}) + ((E_{<n>} * B) * 4^n) \quad (0 \leq n \leq ((H/2)-1), A'_{[-1]} = 0) \quad (5-58)$$

$$= D_{[(M+2n-1):0]} + ((E_{<n>} * B) * 4^n) \quad (0 \leq n \leq ((H/2)-1)) \quad (5-59)$$

所以：

$$D_n = D_{[(M+2n+1):0]} = D_{[(M+2n-1):0]} + ((E_{<n>} * B) * 4^n) \quad (0 \leq n \leq ((H/2)-1)) \quad (5-60)$$

$D_{[(M+2n-1):0]}$ 为前次运算结果， $D_{[(M+2n+1):0]}$ 为本次运算结果。

因为符号数操作，对上式作如下变换：

$$D_{[(M+2n+1):0]} = ((D_{[(M+2n-1):2n]} * 4^n) + D_{[(2n-1):0]}) + ((E_{<n>} * B) * 4^n) \quad (5-61)$$

$$= ((D_{[(M+2n-1):2n]} + (E_{<n>} * B)) * 4^n) + D_{[(2n-1):0]} \quad (5-62)$$

$$= (D_{[(M+2n-1):2n]} + (E_{<n>} * B)) \$ D_{[(2n-1):0]} \quad (5-62)$$

令

$$T_n = D_{[(M+2n-1):2n]} + (E_{<n>} * B) \quad (5-66)$$

其中的 $D_{[(M+2n-1):2n]}$ 被解释为 M 位符号数； $(E_{<n>} * B)$ 被解释为符号数，因 $E_{<n>}$ 绝对值最大为 2，因此 $(E_{<n>} * B)$ 为 $(M+1)$ 位；而 $D_{[(2n-1):0]}$ 被解释为无符号数。二数—— $D_{[(M+2n-1):2n]}$ 、 $(E_{<n>} * B)$ ——相加，其和值 S 为 $(M+1)$ 位。考虑有可能溢出，将二数作符号扩展一位。因此， T_n 为 $(M+2)$ 位。

所以：

$$D_{[(M+2n+1):0]} = T_n \$ D_{[(2n-1):0]} \quad (5-67)$$

$$= X \$ S_{[M:0]} \$ D_{[(2n-1):0]} \quad (5-68)$$

$$= (X \$ S_{[M:2]}) \$ (S_{[1:0]} \$ D_{[(2n-1):0]}) \quad (5-69)$$

$$= D_{[(M+2n+1):(2n+2)]} \$ D_{[(2n+1):0]} \quad (5-70)$$

以上各式皆限定： $0 \leq n \leq (N-1)$ ； D 初始化为 0； $D_{[-1:0]}$ 不存在，其值为 0。以上各式中 $D_{[(M+2n-1):2n]}$ 均为前次计算结果低部分， $D_{[(2n-1):0]}$ 均为前次计算结果高部分，而式(5-70)中各部分则为本次运算结果。

它需用 $(M+1)$ 位宽的加法器，扩展位另行处理。

对**无符号数二位乘法**，除布思编码不同外，其与符号数二位乘法完全相同，在此也就不再赘述。

6、乘法运算的实现——迭代

无符号数一位乘法的实现在前面已有讨论，它要两步——加、移位——才能得到一次部分积。因此，若乘数位宽为 N ，则整个运算需用 $2N$ 个时钟周期。

参见式(5-20)、(5-23)、(5-24)、(5-25)：

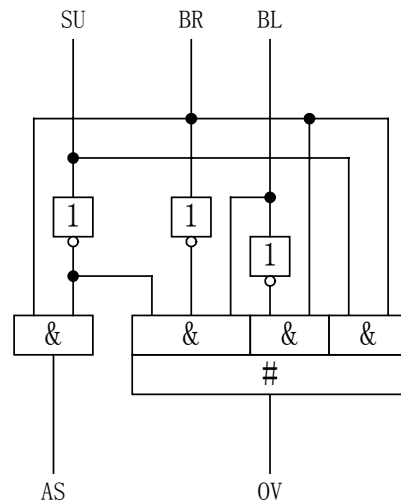
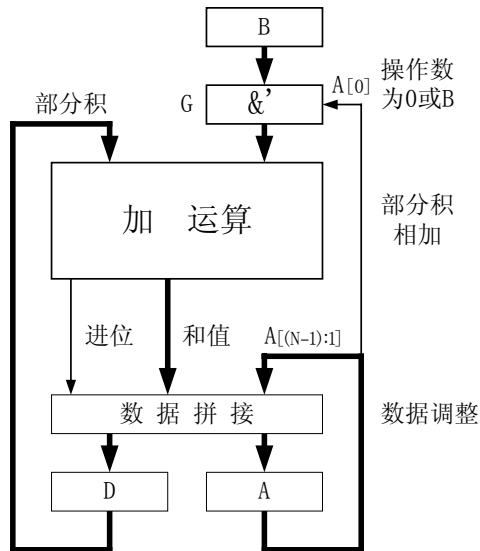
$$D_{[M+n]:0] = (D_{[M+n-1]:n] + (A_{[n]} * B)) \$ D_{[(n-1):0]} \quad (6-1)$$

$$= C \$ S_{[M-1]:0]} \$ D_{[(n-1):0]} \quad (6-2)$$

$$= (C \$ S_{[M-1]:1]} \$ (S_{[0]} \$ D_{[(n-1):0]}) \quad (6-3)$$

$$= D_{[M+n):(n+1]} \$ D_{[n:0]} \quad (6-4)$$

可以看出，可以将加法运算结果立即处理，从而省去独立的移位操作，实现一步得到部分积，则只需用 N 个周期即可完成运算。其原理如下左图。运算步骤与前述一样。操作时需注意：前述对寄存器 C 、 D 、 A 联合移位，而这里是将进位、和值、 A 寄存器联合移位。



对于符号数一位乘法，参照前面有关无符号数一位乘法实现。需要注意的是：布思编码、最低位扩展。布思编码确定操作是加或减、操作数为 0 或 B。

布思编码考虑本位、相邻低位。由于对 A 右移，使本位处于最低位，相邻低位已经移出，因此，需要扩展一位，以保存此相邻低位。此位初始值为 0。若有进位寄存器，可以借用。

这里也就不列出原理图、运算步骤了。

布思编码逻辑表达式如下：

$$AS = US \& BR \quad (6-5)$$

$$OV = US \& (BR \text{ @ } BL) \# (\sim US) \& BR \quad (6-6)$$

$$= US \& (\sim BR) \& BL \# BR \& (\sim BL) \# (\sim US) \& BR \quad (6-7)$$

以 BR 指示本位，以 BL 指示相邻低位。 AS 为 0 时加，为 1 时减； OV 为 0 时操作数为 0，为 1 时操作数为 B ； US 为 0 时作无符号数操作，为 1 时作符号数操作。其逻辑图如上右。

对于符号数二位乘法。参见式(5-62)、(5-68)、(5-69)、(5-70)、(4-6)：

$$D_{[M+2n+1]:0] = (D_{[M+2n-1]:n] + (E_{[n]} * B)) \$ D_{[(2n-1):0]} \quad (6-8)$$

$$= X \ \$ \ S_{[M:0]} \ \$ \ D_{[(2n-1):0]} \quad (6-9)$$

$$= (X \ \$ \ S_{[M:2]}) \ \$ \ (S_{[1:0]} \ \$ \ D_{[(2n-1):0]}) \quad (6-10)$$

$$= D_{[(M+2n+1):(2n+2)]} \ \$ \ D_{[(2n+1):0]} \quad (6-11)$$

$$E_{\langle n \rangle} = -2A'_{[2n+1]} + A'_{[2n]} + A'_{[2n-1]} \quad (6-12)$$

比照前面一位乘法的算法、实现，可知：1、每次移位 2 位；2、布思编码需确定运算为加、减；运算量为 0、B、2B；3、布思编码检测 A 寄存器最低 2 位及扩展位；4、符号数操作；5、若乘数为 N 位，则耗时为 (N/2) 或 ((N+1)/2) 个时钟周期。

在此，也就没有必要列出原理图、运算步骤，请诸位自己完成。

布思编码逻辑表达式其一如下：

$$AS = BH \quad (6-13)$$

$$OV = \sim ((\sim BH) \& (\sim BR) \& (\sim BL) \# BH \& BR \& BL) \quad (6-14)$$

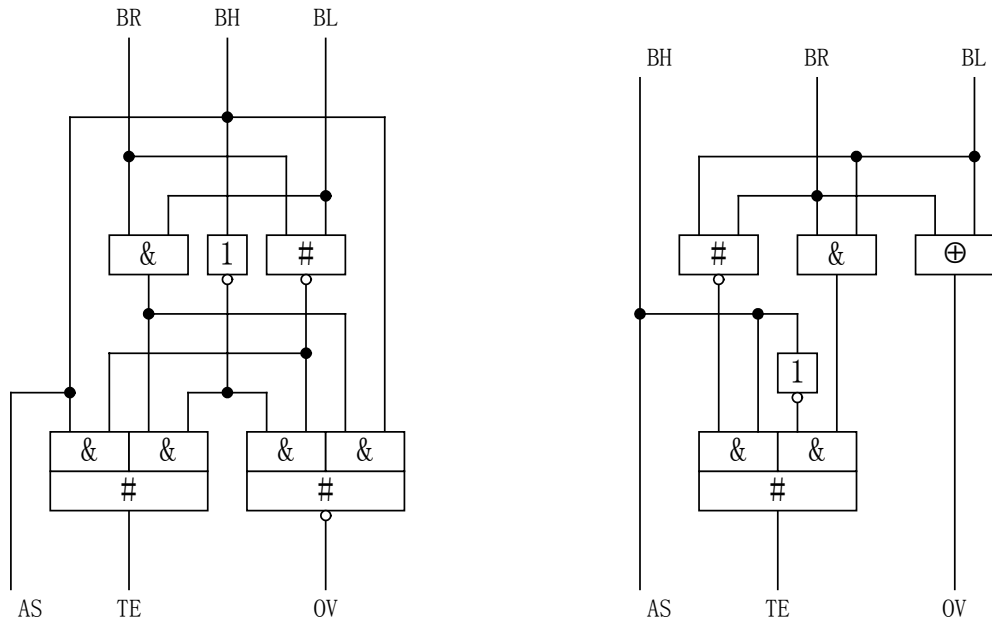
$$= \sim ((\sim BH) \& (\sim (BR \# BL)) \# BH \& (BR \& BL)) \quad (6-15)$$

$$= (\sim BH) \& (BR \# BL) \# BH \& (\sim (BR \& BL)) \quad (6-16)$$

$$TE = (\sim BH) \& BR \& BL \# BH \& (\sim BR) \& (\sim BL) \quad (6-17)$$

$$= (\sim BH) \& (BR \& BL) \# BH \& (\sim (BR \# BL)) \quad (6-18)$$

以 BH 指示相邻高位，以 BR 指示本位，以 BL 指示相邻低位。AS 为 0 时加，为 1 时减；OV 为 0 时操作数为 0，为 1 时操作数为 B 或 2B；TE 为 0 时操作数为 B，为 1 时操作数为 2B。其逻辑图如下左所示。



布思编码逻辑表达式另可为：

$$AS = BH \quad (6-13)$$

$$OV = (\sim BR) \& BL \# BR \& (\sim BL) \quad (6-15)$$

$$= BR \ @ \ BL \quad (6-16)$$

$$TE = (\sim BH) \& BR \& BL \# BH \& (\sim BR) \& (\sim BL) \quad (6-17)$$

$$= (\sim BH) \& (BR \& BL) \# BH \& (\sim (BR \# BL)) \quad (6-18)$$

以 BH 指示相邻高位，以 BR 指示本位，以 BL 指示相邻低位。AS 为 0 时加，为 1 时减；OV 为 0 时操作数为 0，为 1 时操作数为 B；TE 为 0 时操作数为 0，为 1 时操作数为 2B。其逻辑图如上右。实际应用时，此方案为好。

至于**无符号数二位乘法**，其与符号数二位乘法只在布思编码上有区别。

布思编码逻辑表达式可作如下确定：

$$AS = BH \& (\sim BR) \& BL \# BH \& BR \& (\sim BL) \# BH \& BR \& BL \quad (6-19)$$

$$= BH \& (BR \# BL) \quad (6-20)$$

还有一种等效方法：

见上表，会发现相邻乘积项错开二位。相邻高阶乘积项相对于本阶乘积项正好空出了低二位，本阶乘积项进位正好可利用此二位。

可第 $(H-1)/2$ 项乘积项进位怎样放？可以单独放置一行；有时，可以不考虑。

对于 $(-2 * B)$ 实现，如下：

$$-2 * B = 2 * (-B) \tag{7-1}$$

$$= (-B) \ll 1 \tag{7-2}$$

$$= ((\sim B) + 1) \ll 1 \tag{7-3}$$

$$= ((\sim B) \ll 1) + (1 \ll 1) \tag{7-4}$$

$$= ((\sim B) \$ 0) + 2 \tag{7-5}$$

$$= ((\sim B) \$ 0) + 1 + 1 \tag{7-6}$$

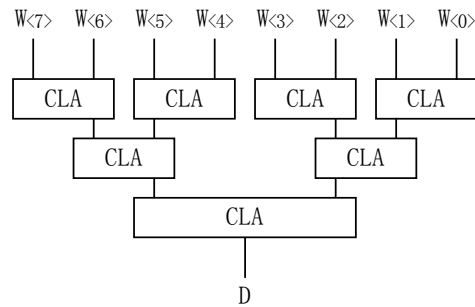
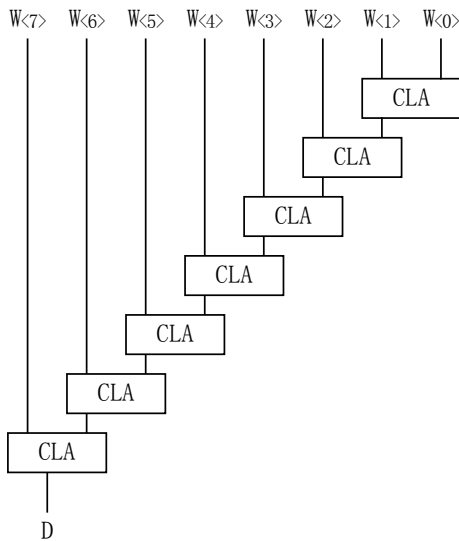
$$= ((\sim B) \$ 1) + 1 \tag{7-7}$$

$$= (\sim (B \$ 0)) + 1 = (\sim (B \ll 1)) + 1 \tag{7-8}$$

可以如式(7-4)所示，将 B 非值、进位全都左移一位；或如式(7-7)所示，将 B 非值左移一位，最低位填充 1，进位为 1；也可如式(7-8)所示，将 B 低位扩展一位（左移一位）取非，进位为 1。实现时，进位为 1 的方案为好。

乘积项阵列已得到解决！可还要将此阵列转换为最终结果，这就要用到加法器阵列。

很容易想到如下左图所示串行累加（假设有 8 项乘积项）。但问题是运算是串行的，延迟不会小（最快也是七级超前进位加法器）、速度也就不会高。



可以采用如上图所示加法器阵列，运算尽量并行，称作“华莱士树”。它的延迟为三级超前进位加法器延迟。虽如此，由于超前进位加法器延迟还比较大、速度也不高。

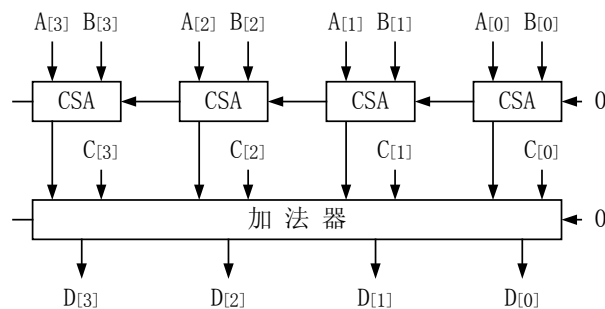
这两种方法中，所有的加法运算都存在进位传播，从而形成了较大的延迟。若能将进位传播解决，

那可就好了！

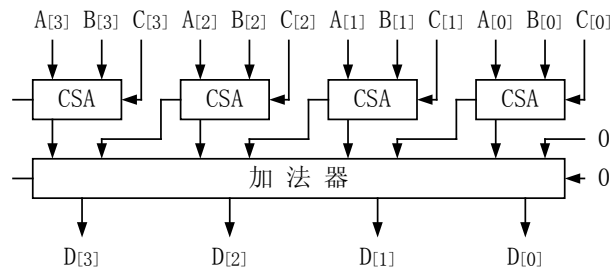
假设将三个 4 位数作加得一 4 位值，其中一方案如右所示：

上面一行为串行进位加法器，用以计算 A 与 B 的和值，其再与 C 相加得到最终结果。上面一行最大延迟为 4 级保留进位加法器延迟。

将其改造如后。它将原图中上一行进位通路打断，各保留进位加法器输出的进位直接输送到下一行加法器，保留进位加法器的 3 个输入端分别为 A、B、C 的相应位。



在这种方式中，源数据到下面一行加法器的延迟均为1级保留进位加法器延迟，明显比上一方案为佳。这也是最佳方案！



其实，对保留进位加法器，输入3个一位数据：A、B、Ci；输出2个一位数据：D、Co。其代数运算式如下：

$$Co * 2 + D = A + B + Ci \quad (7-9)$$

非常明显，保留进位加法器为一计数器——计算输入信号中“1”的个数，计数值由 Co、D 指示，且：Co 权值为2；A、B、Ci、D 权值为1。

其逻辑表达式如下：

$$D = A \oplus B \oplus Ci \quad (7-10)$$

$$Co = A \& B \# A \& Ci \# Ci \& A \quad (7-11)$$

若将保留进位加法器并成一行，则从三个输入端输入三个数据，就可在两个输出端得到两个数据。假设它们均按权值对齐，则：

$$Co_{[n+1]} \& D_{[n]} = A_{[n]} + B_{[n]} + Ci_{[n]} \quad (7-12)$$

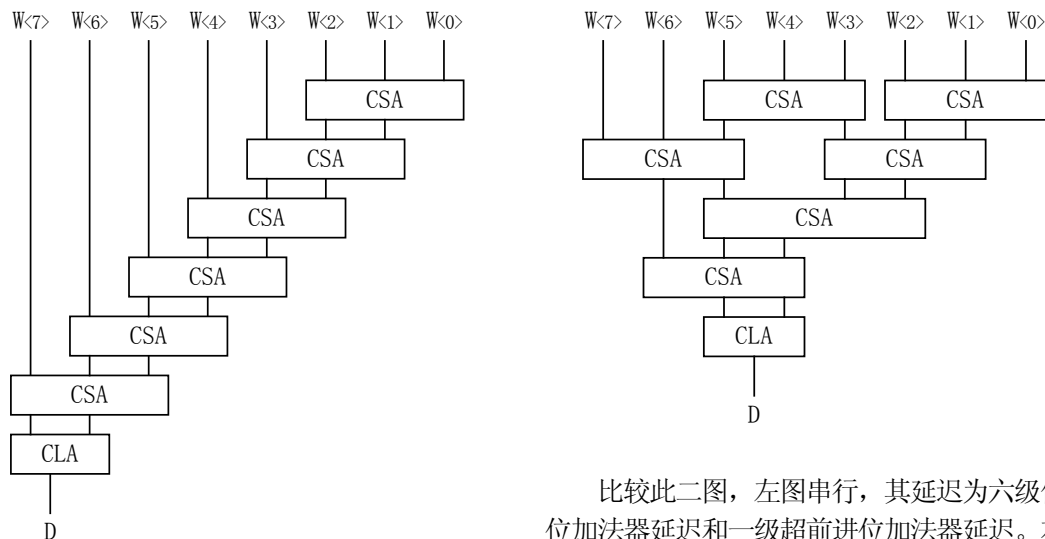
$$Co_{[n+1]} * 2 + D_{[n]} = A_{[n]} + B_{[n]} + Ci_{[n]} \quad (7-13)$$

可以看出，（结果）各位互不关联。且有：

$$Co + D = A + B + Ci \quad (7-14)$$

此时，得到的二值D，Co 是由三数相加而得，其与通常意义的和值、进位不同，称之为伪和、伪进位。

由上面的讨论，可将前面二方案中加法器阵列用保留进位加法器改造如下：



比较此二图，左图串行，其延迟为六级保留进位加法器延迟和一级超前进位加法器延迟。右图尽可能并行，存在四级保留进位加法器延迟和一级超前进位加法器延迟。左图比右图延迟大。右图亦称作“华莱士树”。

一般来说，超前进位加法器至少有六级门延迟，而保留进位加法器为二级门延迟。因此，不要从图中直观的得出，或从字面上简单地认为上右图延迟比前述右图方案延迟长。（其实，门级数并非如此算，只是提醒各位注意）。

前面保留进位加法器是作计数器用，将输入的三个数据转换为二个数据。也可以寻求其它计数方式。其中之一就是5-3计数器：

它有五个输入端：I0、I1、I2、I3、Ci；三个输出端：D、C、Co。将5-3编码器并成一行，即为5-3计数器；若将相邻低位之Co接入本位之Ci，则成为4-2压缩器。这样可以减少二个操作数。

5-3计数器代数运算式如下：

$$D + C * 2 + Co * 2 = I0 + I1 + I2 + I3 + Ci \quad (7-15)$$

即：I0、I1、I2、I3、Ci、D权值为1；C、Co权值为2。

其值表如下：

Ci	I0, I1, I2, I3 和值	Co	C	D
0	0	0	0	0
	1	0	0	1
	2	?	?	0
	3	?	?	1
	4	1	1	0
1	0	0	0	1
	1	?	?	0
	2	?	?	1
	3	1	1	0
	4	1	1	1

表中出现了“？”，表示其值待定，要求出现在同一行中的二“？”值不能相同。

对此“？”的确定有一定技巧：用作4-2压缩器时不能存在进位传播。

其中之一是：

$$D = I0 \oplus I1 \oplus I2 \oplus I3 \oplus Ci$$

$$C = (I0 \oplus I1 \oplus I2 \oplus I3) \& Ci \# (\sim (I0 \oplus I1 \oplus I2 \oplus I3)) \& (I0 \& I1 \# I2 \& I3)$$

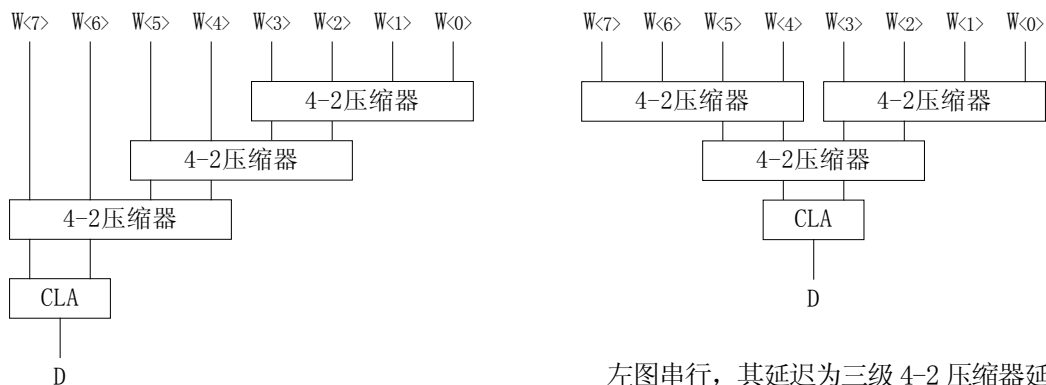
$$= (I0 \oplus I1 \oplus I2 \oplus I3) \& Ci \# (\sim ((I0 \oplus I1 \oplus I2 \oplus I3) \# (\sim (I0 \& I1 \# I2 \& I3))))$$

$$Co = (I0 \# I1) \& (I2 \# I3)$$

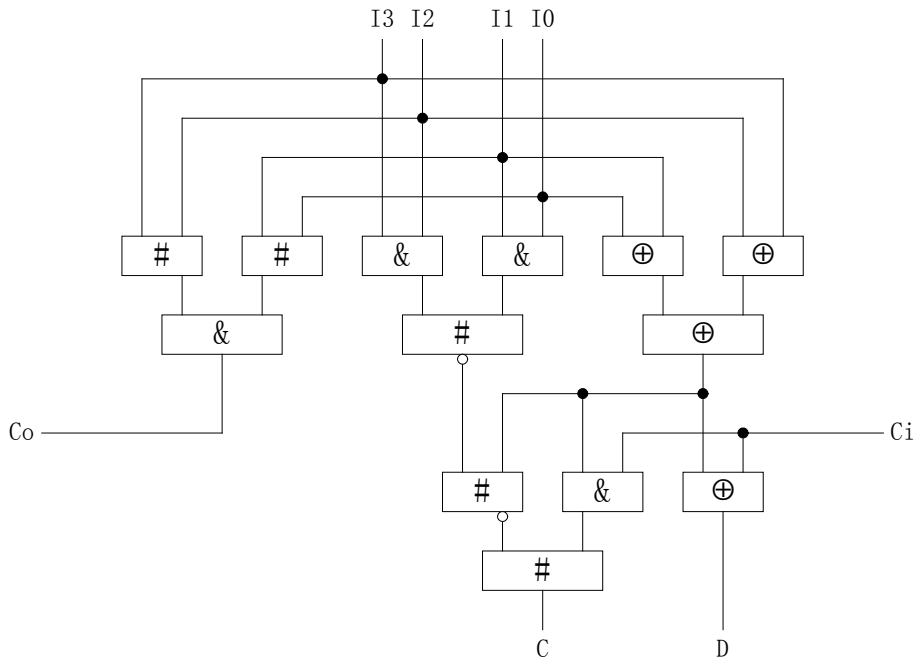
其逻辑图如下页所示。

由图中可以看出：用作4-2压缩器时，将相邻低位之Co接入本位之Ci，对延迟并无影响。因为Ci在被用到时，Co正好形成。

可以用4-2压缩器改造前面的方案，如下：



左图串行，其延迟为三级4-2压缩器延迟和一级超前进位加法器延迟。右图并行，其延迟为二级4-2压缩器延迟和一级超前进位加法器延迟。



从前面的讨论中可以得出：华莱士树速度最快。但对集成电路布图却是非常不友好，因为其极不规整。倒是前述列出的串行计数器阵列方式，对集成电路布图还非常友好。当然，速度快且适宜于集成电路布图的计数器阵列也还有其它的形式，这需要联系到后端一起考虑。在此，也就没有必要让我来“自曝其陋”了吧！

在乘法器中，并不需要完全使用超前进位！

8、乘加运算

在指令集中，常会出现乘加指令：

$$D = R + A * B$$

对其实现，若是迭代乘法，也就没有可说的；若是阵列实现，则，有必要说一下。

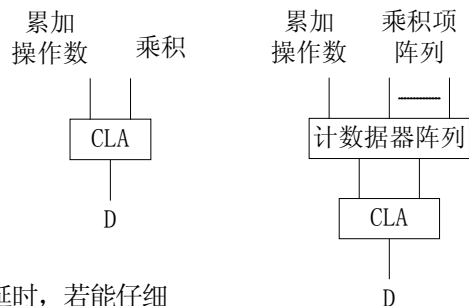
当然可以如右左图所示：先乘后加。

它的延迟是：乘法器延迟及加法器延迟，需两组超前进位链（若用的话）。

我们知道，乘法操作，由加法实现，其加就有多项乘积项。在此，我们可以将此累加项作为一乘积项，将其送入计数器阵列，就可以只用一组超前进位链。当然，这需要增加计数器。

相较于纯乘法器，此种方法的延时最多增加一级计数器延时，若能仔细安排计数器，甚至可以不增加延时。

有时可能会出现如下指令：



$$D = R - A * B$$

对其实现，其难点在于：减操作。可作如下变换：

$$D = R + (A * (-B))$$

我们当然可以先对 B 求补再作处理，但这样会增加延时。

其实，形成乘积项就需要求补运算，我们可以将在乘法中应作的取补取消，用原值的求补，就正好实现。实际上，

$$\begin{aligned} D &= R - A * B \\ &= R + ((-A) * B) \\ &= R + \left((-\sum_{n=0}^{(H/2)-1} (E_{\langle n \rangle} * 4^n)) * B \right) \end{aligned} \quad (9-1)$$

可以看到：对乘法运算求补，可以将布思编码取补，而这只需要将前述逻辑表达式中指示加减操作的信号取反即可。

在部分精简指令集计算机(RISC)中，出于性能的考虑，而又不致损失(整数)乘法精度，将一个完整的乘法运算用两条指令——取低部分积、取高部分积——实现。而乘法运算又要区分符号数与无符号数，则需要设置四条指令：符号数乘法取低部分积、符号数乘法取高部分积；无符号数乘法取低部分积、无符号数乘法取高部分积，以分别实现完整的符号数乘法与无符号数乘法。

因为对符号数扩展符号位、对无符号数作 0 扩展，其值不变，但却成为符号数。设此扩展位为 A_x ，对符号数为符号位、对无符号数为 0。则扩展后的数值为：

$$A = -A_x * 2^N + A_{[N-1]} * 2^{N-1} + A_{[N-2]} * 2^{N-2} + \dots + A_{[1]} * 2^1 + A_{[0]} * 2^0 \quad (9-2)$$

则，若有一位宽 M 不小于 N 的数据 B 与其相乘，则其结果低 N 位对于符号数与无符号数必定全部相同；若数据 B 位宽不大于 N，则其结果低 M 位对于符号数与无符号数也必定全部相同。除开此两种情况，则 A 或 B 扩展位会参与运算……

其证明就请各位自己动手（提示：要将二操作数中的每一位作运算）。

由于此特性，在设置指令时，就可以将符号数乘法取低部分积、无符号数乘法取低部分积这两条指令合并为一条指令：取乘积低部分。

在设计示例 2 中，用 Mul_DD_D2_A1_V2_T 对此作了验证。

此处再说明一点：对于此部分内容，若自成章节，当然很好，但蒋郎才疏，想不到合适的标题，只好缀于此，请见谅！

9、设计示例 1 —— 8 位、迭代

从本节开始，以三节篇幅举例说明乘法器的设计，并对前面文字叙述稍事补充。

本节示例：设计一乘法器，实现二 8 位数相乘。

1、实现方案 1 —— 一位、无符号

本方案只考虑无符号数乘法的实现，每次只处理一位。示例如下：

```
module Mul_DD_D1_A1_V1 (
    A, // source data
    B, // source data
    D, // result data
    Startup, // Load the data when it is '1', and Operate when it fall
    Busy, // Busy when operating and wating for save
    Over, // The operation is over and advise others device to save the result
    Accept, // The result was accepted
    Clock //
);

input [ 7:0] A,B;
output[15:0] D;
input      Startup;
output     Busy,Over;
input      Accept;
input      Clock;

reg [ 7:0] rta,rtb,rtd;
reg      rtc;

wire      Load, // load the source data to the temp register
          Operate, // calculate and shift
          Cal_Shift, // Calculate when it is '0', else Shift
          Loop_End, // the operation is over
          Save; // advise others device to save the result

// * * output

assign D = {rtd,rta};
```

```

assign Busy = Operate | Save & (~ Accept);
assign Over = Save;

// * * operate

always @(negedge Clock)
  if(Load)
    rtb <= B;

always @(negedge Clock)
  if(Load)
    {rtc, rtd, rta} <= {1'b0, A};
  else
    if(Operate)
      if(~ Cal_Shift) // step add
        {rtc, rtd, rta} <= ({(1'b0, rtd) + (rta[0] ? {1'b0, rtb} : {9{1'b0}})), rta};
      else // step shift
        {rtc, rtd, rta} <= {1'bx, rtc, rtd, rta[7:1]};

// * * state switch

reg [ 1:0] State;// 00: Idle; 01:Operate; 11:Operate; 10:Save

always @(negedge Clock)
  casex(State)
    2'b00: if(Startup)
      State <= 2'b01;
    else
      State <= 2'b00;
    2'b01: State <= 2'b11;
    2'b11: if(Loop_End)
      State <= 2'b10;
    2'b10: if(Accept)
      if(Startup)
        State <= 2'b01;
      else
        State <= 2'b00;
  endcase

assign Load = (State == 2'b00) & Startup | (State == 2'b10) & Accept & Startup;
assign Operate = (State == 2'b01) | (State == 2'b11);
assign Save = (State == 2'b10);

// * * Loop counter

```

```

reg    [ 3:0] Cc;

always @(negedge Clock)
    if(Operate)
        Cc <= Cc + 2'b01;
    else
        Cc <= 4'b0000;

assign Loop_End = & Cc;
assign Cal_Shift = Cc[0];

endmodule

```

此例并不难理解，说明也就免了吧！其测试模块如下：

```

module Mul_DD_D1_A1_V1_T ();

reg    [ 7:0] A,B;
wire   [15:0] D;
wire           Startup;
wire           Over, Busy;
wire           Accept;
reg           Clock;

reg    [ 7:0] C;

reg    [ 7:0] Ta, Tb;
reg    [15:0] TD;

reg    [15:0] RD;

wire           CR;

Mul_DD_D1_A1_V1 Mul_DD_D1_A1_V1_C (
    A,
    B,
    D,
    Startup,
    Busy,
    Over,
    Accept,
    Clock
);

```

```

always @(negedge Clock)
  if(Accept)
    RD <= D;

always @(negedge Clock)
  if(Accept)
    TD <= {1'b0, Ta} * {1'b0, Tb};

assign CR = (RD == TD);

always @(negedge Clock)
  if(Startup)
    {Ta, Tb} <= {A, B};

assign # 1 Startup = ((& (C[3:2] ^ C[1:0])) & (~ Busy));

assign # 1 Accept = Over & (& (C[7:6] ^ C[5:4]));

always @(negedge Clock)
  {A, B, C} <= $random;

initial
  begin
    Clock <= 1'b0;
    forever
      # 5 Clock <= ~ Clock;
  end

endmodule

```

此例并不难理解，没必要让我作出说明！只是 Startup、Accept 的形成有随机（周期）延迟，这是模拟实际系统中的调度处理；测试结果以 CR 显示出来，若其为低，则运算有误！

此例执行运算时，加法和移位各占一个周期，总共需 16 个周期。我们在前面已经说过，可以将加、移位在一个周期完成。示例如下：

```

module Mul_DD_D1_A1_V2 (
  A, // source data
  B, // source data
  D, // result data
  Active, // Operate when it is '1'
  Over, // The operation is over and advise others device to save the result
  Accept, // The result was accepted
  Clock //
);

```

```

input [ 7:0] A,B;
output[15:0] D;
input      Active;
output     Over;
input      Accept;
input      Clock;

reg [ 7:0] rta,rtb,rtd;

wire      Load,// load the source data to the temp register
          Operate,// calculate and shift
          Loop_End,// the operation is over
          Save;// advise others device to save the result

// * * output

assign D = {rtd,rta};

assign Over = Save;

// * * operate

always @(negedge Clock)
  if(Load)
    rtb <= B;

always @(negedge Clock)
  if(Load)
    {rtd,rta} <= {1'b0,A};
  else
    if(Operate)
      {rtd,rta} <= ({1'b0,rtd}+ ({9{rta[0]}} & {1'b0,rtb})),rta} >> 1;

// * * state switch

reg [ 1:0] State;// 00: Idle; 01:Operate; 11:Operate; 10:Save

always @(negedge Clock)
  if(Active)
    casex(State)
      2'b01: State <= 2'b11;
      2'b11: if(Loop_End)
              State <= 2'b10;
      2'b10: if(Accept)

```

```

                State <= 2'b00;
            default:State <= 2'b01;
        endcase
    else
        State <= 2'b00;

assign Load = Active & ((State == 2'b00) | (State == 2'b10) & Accept);
assign Operate = Active & (State == 2'b01) | (State == 2'b11);
assign Save = Active & (State == 2'b10);

// * * Loop counter

reg [ 2:0] Cc;

always @(negedge Clock)
    if(Operate)
        Cc <= Cc + 2'b01;
    else
        Cc <= 3'b000;

assign Loop_End = & Cc;

endmodule

```

同样，此例也不难理解，说明也就免了！其测试模块如下：

```

module Mul_DD_D1_A1_V2_T ();

reg [ 7:0] A,B;
wire [15:0] D;
reg Active;
wire Over;
wire Accept;
reg Clock;

reg [ 7:0] C;

reg [ 7:0] Ta,Tb;
reg [15:0] TD;

reg [15:0] RD;

wire CR;

```

```

Mu1_DD_D1_A1_V2 Mu1_DD_D1_A1_V2_C (
    A,
    B,
    D,
    Active,
    Over,
    Accept,
    Clock
);

always @(negedge Clock)
    if(Accept)
        {A,B} <= $random;

always @(negedge Clock)
    if(Accept)
        RD <= D;

always @(negedge Clock)
    if(Accept)
        TD <= {1'b0, Ta} * {1'b0, Tb};

assign CR = (RD == TD);

always @(negedge Clock)
    if(Active)
        {Ta, Tb} <= {A, B};

initial
    Active <= 1'b0;

always @(negedge Clock)
    if(~ Active)
        if(& (C[3:2] ^ C[1:0]))
            Active <= # 1 1'b1;
        else
            Active <= # 1 1'b0;
    else
        if(Accept)
            if(& (C[3:2] ^ C[1:0]))
                Active <= # 1 1'b1;
            else
                Active <= # 1 1'b0;

assign # 1 Accept = Over & (& (C[7:6] ^ C[5:4]));

always @(negedge Clock)
    C <= $random;

```

```

always
  begin
    # 5 Clock <= 1'b0;
    # 5 Clock <= 1'b1;
  end

endmodule

```

不用各位费心就可看出：此二例不同。最明显的是接口不同，而二者实现上的差别，也是因此而决定！若各位能看出更深层次的不同点，恭喜！

这最根本的不同点是设计思想不同！Mul_DD_D1_A1_V1 为自主运行，Mul_DD_D1_A1_V2 则是受控运行。

对 Mul_DD_D1_A1_V1 施加一启动信号，即可自动运行，事完报告。Mul_DD_D1_A1_V2 则需一直施加控制信号，不得中途撤消，否则前功尽弃。打个不大恰当的比方：员工 Mul_DD_D1_A1_V1 很自觉，主管下一任务，他会自行尽数完成，就连提醒都没有必要；员工 Mul_DD_D1_A1_V2 可就不自觉了，需有主管监督，否则会撻挑子。

可不要小看这二者之间的差别，效果（非结果）可就大不一样。在超标量处理器中，所有功能部件都必须自行控制，否则就无法实现。试想，某部件将控制器长久占用，流水线必然要停顿，任务如何分派？其它功能部件如何动作？

各位在仿真时可以看到，在结果被接收时，Load 信号出现一次，不管其后 Active 是否会继续。就请各位思考一下。

各位也可以按 Mul_DD_D1_A1_V1 接口实现将加法操作、移位操作在一个周期完成。

2、实现方案2 —— 一位、布思

此方案采用一位布思算法。

```

module Mul_DD_D1_A2_V1 (
  A, // source data
  B, // source data
  D, // result data
  Sign, // 0:unsign ; 1:sign
  Startup, // Load the data when it is '1', and Operate when it fall
  Busy, //
  Over, //
  Accept, // The result was accepted
  Clock //
);

input [ 7:0] A,B;
output [15:0] D;

```

```

input      Sign;
input      Startup;
output     Busy, Over;
input      Accept;
input      Clock;

reg  [ 8:0] rta, rtb, rtd;
reg      rtx;

wire      Load, // load the source data to the temp register
          Operate, // calculate and shift
          Loop_End, // the operation is over
          Save; // advise others device to save the result

wire      Be_AS, // add(0) or sub(1)
          Be_OV; // once is valid if it is '1' else zero

// * * output

assign D = {rtd, rta};

assign Busy = Operate | Save & (~ Accept);
assign Over = Save;

// * * operate

always @(negedge Clock)
  if(Load)
    {rtb} <= {(Sign & B[7]), B};

always @(negedge Clock)
  if(Load)
    {rtd, rta, rtx} <= {1'b0, {(Sign & A[7]), A}, 1'b0};
  else
    if(Operate)
      {rtd, rta, rtx} <= {{{rtd[7], rtd[7:0]} + {1'b0, (Be_AS & Be_OV)} +
                          (({9{Be_AS}} ^ rtb) & {9{Be_OV}})}, rta};

// * * booth encode
assign Be_AS = rta[0];
assign Be_OV = rta[0] ^ rtx;

// * * state switch

```

```

reg [ 1:0] State;

always @(negedge Clock)
  casex(State)
    2'b00: if(Startup)
      State <= 2'b01;
    else
      State <= 2'b00;
    2'b01: if(Loop_End)
      State <= 2'b11;
    2'b11: State <= 2'b10;
    2'b10: if(Accept)
      if(Startup)
        State <= 2'b01;
      else
        State <= 2'b00;
  endcase

assign Load = (State == 2'b00) & Startup | (State == 2'b10) & Accept & Startup;
assign Operate = (State == 2'b01) | (State == 2'b11);
assign Save = (State == 2'b10);

// * * Loop counter

reg [ 2:0] Cc;

always @(negedge Clock)
  if(Operate)
    Cc <= Cc + 2'b01;
  else
    Cc <= 3'b000;

assign Loop_End = & Cc;

endmodule

```

此例只处理符号数，它将输入的数据作符号扩展转换为符号数。由此，数据宽度为 9 位，布思编码为 9 项，运算也为 9 次。而计数器 Cc 只有 3 位，只能计数 8 次。在处理中应用了状态机中一个状态，从而实现 9 次计数。

可是又来疑问了：此例与前面各例相比，状态既没有多、也没有少，为何前面各例计数为 8，而此例却为 9。还麻烦各位分析一下。

其测试模块如下：

```

module Mul_DD_D1_A2_V1_T ();

reg    [ 7:0]  A,B;
reg          Sign;
wire   [15:0]  D;
wire          Startup;
wire          Over,Busy;
wire          Accept;
reg          Clock;

reg    [ 7:0]  C;

reg    [ 7:0]  Ta,Tb;
reg          Ts;
reg   [15:0]  TD;

reg   [15:0]  RD;

wire          CR;

Mul_DD_D1_A2_V1 Mul_DD_D1_A2_V1_C (
    A,
    B,
    D,
    Sign,
    Startup,
    Busy,
    Over,
    Accept,
    Clock
);

always @(negedge Clock)
    if(Accept)
        RD <= D;

always @(negedge Clock)
    if(Accept)
        TD <= {{8{(Ts & Ta[7])}}, Ta} * {{8{(Ts & Tb[7])}}, Tb};

assign CR = (RD == TD);

always @(negedge Clock)
    if(Startup)
        {Ta, Tb, Ts} <= {A, B, Sign};

assign # 1 Startup = ((& (C[3:2] ^ C[1:0])) & (~ Busy));

```

```

assign # 1 Accept = Over & (& (C[7:6] ^ C[5:4]));

always @(negedge Clock)
    {Sign,A,B,C} <= $random;

always
    begin
        # 5 Clock <= 1'b0;
        # 5 Clock <= 1'b1;
    end

endmodule

```

此例只处理符号数，需作 9 次计算，下例则是在处理时区分符号数与无符号数，只需作 8 次计算。

```

module Mul_DD_D1_A2_V2 (
    A, // source data
    B, // source data
    D, // result data
    Sign, // 0:unsign ; 1:sign
    Startup, // Load the data when it is '1', and Operate when it fall
    Busy, //
    Over, //
    Accept, // The result was accepted
    Clock //
);

input [ 7:0] A,B;
output [15:0] D;
input Sign;
input Startup;
output Busy, Over;
input Accept;
input Clock;

reg [ 7:0] rta, rtb, rtd;
reg rts, rtx;

wire Load, // load the source data to the temp register
Operate, // calculate and shift
Loop_End, // the operation is over
Save; // advice others device to save the result

wire Be_AS, // add(0) or sub(1)

```

```

        Be_OV;// once is valid if it is '1' else zero

// * * output

assign D = {rtd,rta};

assign Busy = Operate | Save & (~ Accept);
assign Over = Save;

// * * operate

always @(negedge Clock)
    if(Load)
        {rts,rtb} <= {Sign,B};

always @(negedge Clock)
    if(Load)
        {rtd,rta,rtx} <= {1'b0,A,1'b0};
    else
        if(Operate)
            {rtd,rta,rtx} <= {{{(rts & rtd[7]),rtd} + {1'b0,(Be_AS & Be_OV)} +
                (({9{Be_AS}} ^ {(rts & rtb[7]),rtb}) & {9{Be_OV}})},rta};

// * * booth encode
assign Be_AS = rta[0] & rts;
assign Be_OV = ((~ rts) & rta[0]) | (rts & (rta[0] ^ rtx));

// * * state switch

reg [ 1:0] State;

always @(negedge Clock)
    casex(State)
        2'b00: if(Startup)
                State <= 2'b01;
            else
                State <= 2'b00;
        2'b01: State <= 2'b11;
        2'b11: if(Loop_End)
                State <= 2'b10;
        2'b10: if(Accept)
                if(Startup)
                    State <= 2'b01;
            else

```

```

                State <= 2' b00;
        endcase

assign Load = (State == 2' b00) & Startup | (State == 2' b10) & Accept & Startup;
assign Operate = (State == 2' b01) | (State == 2' b11);
assign Save = (State == 2' b10);

// * * Loop counter

reg [ 2:0] Cc;

always @(negedge Clock)
    if(Operate)
        Cc <= Cc + 2' b01;
    else
        Cc <= 3' b000;

assign Loop_End = & Cc;

endmodule

```

分析也就自己动手吧！其测试模块可采用 Mul_DD_D1_A2_V1_T。当然，还需要修改被调用模块及其单元名。

各位也可以不用布思算法，按文中式(2-3)实现符号数相乘。

3、实现方案3 —— 二位

此方案采用二位布思算法。需要对8位输入数据作符号扩展，成为9位。作布思编码的数据应为偶数位，需要再扩展一位，而为10位。这样，布思编码有5项，需作5次计算。

```

module Mul_DD_D1_A3_V1 (
    A, // source data
    B, // source data
    D, // result data
    Sign, // 0:unsign ; 1:sign
    Startup, // Load the data when it is '1', and Operate when it fall
    Busy, //
    Over, //
    Accept, // The result was accepted
    Clock //
);

```

```

input [ 7:0] A,B;
output[15:0] D;
input      Sign;
input      Startup;
output     Busy, Over;
input      Accept;
input      Clock;

reg [ 9:0] rta;
reg [ 8:0] rtb;
reg [ 7:0] rtd;
reg      rtx;

wire      Load,// load the source data to the temp register
          Operate,// calculate and shift
          Loop_End,// the operation is over
          Save;// advice others device to save the result

wire [ 9:0] TSum;

// * * output

assign D = {rtd,rta};

assign Busy = Operate | Save & (~ Accept);
assign Over = Save;

// * * operate

always @(negedge Clock)
  if(Load)
    {rtb} <= {(Sign & B[7]),B};

always @(negedge Clock)
  if(Load)
    {rtd,rta,rtx} <= {1'b0, {(Sign & A[7]), (Sign & A[7]), A}, 1'b0};
  else
    if(Operate)
      {rtd,rta,rtx} <= {TSum,rta,1'b0} >> 2;

// * * Adder

wire [ 9:0] TO_BP;

```

```

wire          TO_XC;

assign TSum = {rtd[7],rtd[7],rtd} + TO_BP + {1'b0, TO_XC};

// * * booth encode & booth process

wire          Be_AS, // add(0) or sub(1)
              Be_OV, // once is valid if it is '1' else zero
              Be_TE; // twice is valid when it is '1' else zero

assign Be_AS = rta[1];
assign Be_OV = rta[0] ^ rtx;
assign Be_TE = ({rta[1:0], rtx} == 3'b011) | ({rta[1:0], rtx} == 3'b100);

assign TO_BP = (({rtb[8], rtb} ^ {10{Be_AS}}) & {10{Be_OV}}) |
               (({rtb, 1'b0} ^ {10{Be_AS}}) & {10{Be_TE}});
assign TO_XC = Be_AS & (Be_OV | Be_TE);

// * * state switch

reg  [ 1:0] State;

always @(negedge Clock)
  casex(State)
    2'b00: if(Startup)
            State <= 2'b01;
          else
            State <= 2'b00;
    2'b01: if(Loop_End)
            State <= 2'b11;
    2'b11: State <= 2'b10;
    2'b10: if(Accept)
            if(Startup)
              State <= 2'b01;
            else
              State <= 2'b00;
  endcase

assign Load = (State == 2'b00) & Startup | (State == 2'b10) & Accept & Startup;
assign Operate = (State == 2'b01) | (State == 2'b11);
assign Save = (State == 2'b10);

// * * Loop counter

reg  [ 1:0] Cc;

```

```

always @(negedge Clock)
  if(Operate)
    Cc <= Cc + 2'b01;
  else
    Cc <= 2'b00;

assign Loop_End = & Cc;

endmodule

```

本例为符号数二位布思算法。前面讨论中提到：无符号数二位布思算法。示例如下：

```

module Mul_DD_D1_A3_V2 (
  A, // source data
  B, // source data
  D, // result data
  Sign, // 0:unsign ; 1:sign
  Startup, // Load the data when it is '1', and Operate when it fall
  Busy, //
  Over, //
  Accept, // The result was accepted
  Clock //
);

input [ 7:0] A,B;
output [15:0] D;
input      Sign;
input      Startup;
output     Busy, Over;
input      Accept;
input      Clock;

reg [ 9:0] rta;
reg [ 8:0] rtb;
reg [ 8:0] rtd;
reg      rtx;
reg      rts;

wire      Load, // load the source data to the temp register
          Operate, // calculate and shift
          Loop_End, // the operation is over
          Save; // advice others device to save the result

```

```

wire [10:0] TSum;
wire      TBb;// borrow bit

// * * output

assign D = {rtd,rta};

assign Busy = Operate | Save & (~ Accept);
assign Over = Save;

// * * operate

always @(negedge Clock)
  if(Load)
    {rtb, rts} <= {(Sign & B[7]), B, Sign};

always @(negedge Clock)
  if(Load)
    {rtd, rta} <= {1'b0, {(Sign & A[7]), (Sign & A[7]), A}};
  else
    if(Operate)
      {rtd, rta} <= {TSum, rta[9:2]};

always @(negedge Clock)
  if(Load)
    rtx <= 1'b0;
  else
    if(Operate)
      rtx <= TBb;

// * * Adder

wire [ 9:0] TO_BP;
wire      TO_XC;

assign TSum = {rtd[8],rtd[8],rtd} + {TO_BP[9],TO_BP} + TO_XC;

// * * booth encode & booth process

wire      Be_AS,// add(0) or sub(1)
          Be_OV,// once is valid if it is '1' else zero
          Be_TE;// twice is valid when it is '1' else zero

assign Be_AS = rts ? rta[1] : (rta[1] & (rta[0] | rtx));
assign Be_OV = rta[0] ^ rtx;
assign Be_TE = ({rta[1:0],rtx} == 3'b011) | ({rta[1:0],rtx} == 3'b100);

```

```

assign TO_BP = (({rtb[8],rtb} ^ {10{Be_AS}}) & {10{Be_OV}}) |
               (({rtb,1'b0} ^ {10{Be_AS}}) & {10{Be_TE}});
assign TO_XC = Be_AS & (Be_OV | Be_TE);
assign TBb = rts ? rta[1] : Be_AS;

// * * state switch

reg [ 1:0] State;

always @(negedge Clock)
  casex(State)
    2'b00: if(Startup)
            State <= 2'b01;
          else
            State <= 2'b00;
    2'b01: if(Loop_End)
            State <= 2'b11;
    2'b11: State <= 2'b10;
    2'b10: if(Accept)
            if(Startup)
              State <= 2'b01;
            else
              State <= 2'b00;
  endcase

assign Load = (State == 2'b00) & Startup | (State == 2'b10) & Accept & Startup;
assign Operate = (State == 2'b01) | (State == 2'b11);
assign Save = (State == 2'b10);

// * * Loop counter

reg [ 1:0] Cc;

always @(negedge Clock)
  if(Operate)
    Cc <= Cc + 2'b01;
  else
    Cc <= 2'b00;

assign Loop_End = & Cc;

endmodule

```

分析就请自己动手吧！其测试模块可采用 Mu1_DD_D1_A2_V1_T。

10、设计示例 2 —— 16 位、阵列

要求实现二 16 位数相乘，此二数可分别解释为符号数或无符号数，其积值再与一 40 位累加数作累加或累减。如下：

$$D = A * B$$

$$D = C + A * B$$

$$D = C - A * B$$

在进入正题之前，先看看 4-2 压缩器的实现。不过，也不能急，还是先看看 5-3 计数器及 5-3 计数器行的实现。

5-3 计数器的实现如下：

```
module _53C_42C (
    I0, //
    I1, //
    I2, //
    I3, //
    Ci, //
    D, //
    C, //
    Co //
);

input      I0, I1, I2, I3, Ci;
output     D, C, Co;

assign D = I0 ^ I1 ^ I2 ^ I3 ^ Ci;
assign C = (I0 ^ I1 ^ I2 ^ I3) & Ci | (~ (I0 ^ I1 ^ I2 ^ I3)) & (I0 & I1 | I2 & I3);
assign Co = (I0 | I1) & (I2 | I3);

endmodule
```

5-3 计数器行的实现如下（输入数据 DW 位，位序与权值对齐）：

```
module _53C_L (
    I0, //
    I1, //
```

```

        I2, //
        I3, //
        Ci, //
        D, //
        C, //
        Co//
    );

parameter    DW = 8;

input  [(DW-1):0]  I0, I1, I2, I3, Ci;
output[(DW-1):0]  D;
output[DW:1]      C, Co;

assign D = I0 ^ I1 ^ I2 ^ I3 ^ Ci;
assign C = (I0 ^ I1 ^ I2 ^ I3) & Ci | (~ (I0 ^ I1 ^ I2 ^ I3)) & (I0 & I1 | I2 & I3);
assign Co = (I0 | I1) & (I2 | I3);

endmodule

```

对 4-2 压缩器，其输入数据 DW 位（位序与权值对齐），共产生 DW 位 Co，最高位即为用于输出之 Co。为避免混淆，称这 DW 位 Co 为 Cv。Cv 权值为 2，需左移一位，低位空出一位，正好用 Ci 填充。因此，Cv 为 (DW+1) 位，势必要求输入数据扩展一位而为 (DW+1) 位。既如此，D，C 即为 (DW+1) 位，C 权值为 2。实现如下：

```

module _42C_L (
    I0, //
    I1, //
    I2, //
    I3, //
    Ci, //
    D, //
    C, //
    Co//
);

parameter    DW = 8;

input  [(DW-1):0]  I0, I1, I2, I3;
input          Ci;
output[DW:0]      D;
output[(DW+1):1]  C;
output          Co;

wire  [(DW-1):0]  TXR, TAO, TOA;

assign TXR = I0 ^ I1 ^ I2 ^ I3;

```

```

assign TAO = (I0 & I1) | (I2 & I3);
assign TOA = (I0 | I1) & (I2 | I3);

assign D = {TXR[DW-1], TXR} ^ {TOA, Ci};
assign Co = TOA[DW-1];
assign C = ({TXR[(DW-1)], TXR} & {TOA, Ci}) | ((~ {TXR[(DW-1)], TXR}) & {TAO[(DW-1)], TAO});

endmodule

```

拖沓已久，该切入正题了！

按要求，定义接口为下：

```

module Mul_DD_D2_A1_V1 (
    AD, // A operand
    AS, // extend style for A
    BD, // B operand
    BS, // extend style for B
    ACC, // ACCumulate operand
    D, // result data
    CC // control code
);

input [15:0] AD, BD;
input AS, BS;
input [39:0] ACC;
output [39:0] D;
input [ 1:0] CC;

```

以 AS、BS 分别指示 AD、BD 为符号数或无符号数，ACC 为累加操作数，以 CC 第 1 位指示是否累加，以 CC 第 0 位指示累加还是累减。

```

module Mul_DD_D2_A1_V1 (
    AD, // A operand
    AS, // extend style for A
    BD, // B operand
    BS, // extend style for B
    ACC, // ACCumulate operand
    D, // result data
    CC // control code
);

input [15:0] AD, BD;
input AS, BS;

```

```

input [39:0] ACC;
output[39:0] D;
input [ 1:0] CC;

wire [39:0] ACE_T;

wire [39:0] TAR;// temp leading adder result

wire [39:0] TES,// temp counter array pseudo sum
          TEC;// temp counter array pseudo carry

// * * output

//assign D = TES + TEC;
assign D = TAR;

assign ACE_T = CC[1] ? ACC : 1'b0;

// * * Leading Carry Adder

wire [39:0] TAG,TAP;//g & p
wire [40:1] TAC;// carry
wire [11:0] TAG_S0,TAP_S0;
wire [ 3:0] TAG_S1,TAP_S1;
wire [12:1] TAC_S1;
wire [ 4:1] TAC_S2;

assign TAR = TAP ^ {TAC,1'b0};

assign TAG = TES & TEC;
assign TAP = TES ^ TEC;

Carry_Leading_4 CL_S0_P0 (TAG[ 3: 0],TAP[ 3: 0],1'b0 , TAC[ 4: 1],TAG_S0[0],TAP_S0[0]),
                  CL_S0_P1 (TAG[ 7: 4],TAP[ 7: 4],TAC_S1[1],TAC[ 8: 5],TAG_S0[1],TAP_S0[1]),
                  CL_S0_P2 (TAG[11: 8],TAP[11: 8],TAC_S1[2],TAC[12: 9],TAG_S0[2],TAP_S0[2]),
                  CL_S0_P3 (TAG[15:12],TAP[15:12],TAC_S1[3],TAC[16:13],TAG_S0[3],TAP_S0[3]),
                  CL_S0_P4 (TAG[19:16],TAP[19:16],TAC_S1[4],TAC[20:17],TAG_S0[4],TAP_S0[4]),
                  CL_S0_P5 (TAG[23:20],TAP[23:20],TAC_S1[5],TAC[24:21],TAG_S0[5],TAP_S0[5]),
                  CL_S0_P6 (TAG[27:24],TAP[27:24],TAC_S1[6],TAC[28:25],TAG_S0[6],TAP_S0[6]),
                  CL_S0_P7 (TAG[31:28],TAP[31:28],TAC_S1[7],TAC[32:29],TAG_S0[7],TAP_S0[7]),
                  CL_S0_P8 (TAG[35:32],TAP[35:32],TAC_S1[8],TAC[36:33],TAG_S0[8],TAP_S0[8]),
                  CL_S0_P9 (TAG[39:36],TAP[39:36],TAC_S1[9],TAC[40:37],TAG_S0[9],TAP_S0[9]);

Carry_Leading_4 CL_S1_P0 (TAG_S0[ 3:0],TAP_S0[ 3:0],1'b0 ,
                      TAC_S1[ 4:1],TAG_S1[0],TAP_S1[0]),

```

```

        CL_S1_P1 (TAG_S0[ 7:4], TAP_S0[ 7:4], TAC_S2[1],
                TAC_S1[ 8:5], TAG_S1[1], TAP_S1[1]),
        CL_S1_P2 (TAG_S0[11:8], TAP_S0[11:8], TAC_S2[2],
                TAC_S1[12:9], TAG_S1[2], TAP_S1[2]);

Carry_Leading_4 CL_S2_P0 (TAG_S1, TAP_S1, 1'b0, TAC_S2, ,);

// * * booth encode & booth process

wire [17: 0] TD_BP_I0;
wire [19: 2] TD_BP_I1;
wire [21: 4] TD_BP_I2;
wire [23: 6] TD_BP_I3;
wire [25: 8] TD_BP_I4;
wire [27:10] TD_BP_I5;
wire [29:12] TD_BP_I6;
wire [31:14] TD_BP_I7;
wire [33:16] TD_BP_I8;
wire [17:0] XC_BP;

Booth Booth_I0 ({AD[1:0], 1'b0}, CC[0], {(BS & BD[15]), BD}, TD_BP_I0, XC_BP[ 1: 0]),
        Booth_I1 (AD[ 3: 1], CC[0], {(BS & BD[15]), BD}, TD_BP_I1, XC_BP[ 3: 2]),
        Booth_I2 (AD[ 5: 3], CC[0], {(BS & BD[15]), BD}, TD_BP_I2, XC_BP[ 5: 4]),
        Booth_I3 (AD[ 7: 5], CC[0], {(BS & BD[15]), BD}, TD_BP_I3, XC_BP[ 7: 6]),
        Booth_I4 (AD[ 9: 7], CC[0], {(BS & BD[15]), BD}, TD_BP_I4, XC_BP[ 9: 8]),
        Booth_I5 (AD[11: 9], CC[0], {(BS & BD[15]), BD}, TD_BP_I5, XC_BP[11:10]),
        Booth_I6 (AD[13:11], CC[0], {(BS & BD[15]), BD}, TD_BP_I6, XC_BP[13:12]),
        Booth_I7 (AD[15:13], CC[0], {(BS & BD[15]), BD}, TD_BP_I7, XC_BP[15:14]),
        Booth_I8 ({(AS & AD[15]), (AS & AD[15]), AD[15]}, CC[0],
                {(BS & BD[15]), BD}, TD_BP_I8, XC_BP[17:16]);

defparam Booth_I0.DW = 16, Booth_I1.DW = 16, Booth_I2.DW = 16,
        Booth_I3.DW = 16, Booth_I4.DW = 16, Booth_I5.DW = 16,
        Booth_I6.DW = 16, Booth_I7.DW = 16, Booth_I8.DW = 16;

// * * counter array

wire [24:2] CAV_L1_L1_D;
wire [25:3] CAV_L1_L1_C;
wire [32:10] CAV_L1_L2_D;
wire [33:11] CAV_L1_L2_C;
wire [39:16] CAV_L1_L3_D;
wire [40:17] CAV_L1_L3_C;

wire [32:6] CAV_L2_L1_D;
wire [33:7] CAV_L2_L1_C;
wire [39:14] CAV_L2_L2_D;

```

```

wire [40:15] CAV_L2_L2_C;

wire [40:0] CAV_L3_L1_D;
wire [41:1] CAV_L3_L1_C;

// Level 1 Line 1
_42C_L _42C_L_L1_L1 (
    {{6{TD_BP_I0[17]}}, TD_BP_I0[17:2]},
    {{4{TD_BP_I1[19]}}, TD_BP_I1},
    {{2{TD_BP_I2[21]}}, TD_BP_I2, XC_BP[3:2]},
    {TD_BP_I3, XC_BP[5:4], 2'b00},
    1'b0,
    CAV_L1_L1_D,
    CAV_L1_L1_C,
);
defparam _42C_L_L1_L1.DW = 22;

// Level 1 Line 2
_42C_L _42C_L_L1_L2 (
    {{6{TD_BP_I4[25]}}, TD_BP_I4[25:10]},
    {{4{TD_BP_I5[27]}}, TD_BP_I5},
    {{2{TD_BP_I6[29]}}, TD_BP_I6, XC_BP[11:10]},
    {TD_BP_I7, XC_BP[13:12], 2'b00},
    1'b0,
    CAV_L1_L2_D,
    CAV_L1_L2_C,
);
defparam _42C_L_L1_L2.DW = 22;

// Level 1 Line 3
CSA_L CSA_L1_L3 (
    {{6{TD_BP_I8[33]}}, TD_BP_I8},
    {{22{1'b0}}, XC_BP[17:16]},
    ACE_T[39:16],
    CAV_L1_L3_D,
    CAV_L1_L3_C
);
defparam CSA_L1_L3.DW = 24;

// Level 2 Line 1
CSA_L CSA_L2_L1 (
    {{8{CAV_L1_L1_D[24]}}, CAV_L1_L1_D[24:6]},
    {{7{CAV_L1_L1_C[25]}}, CAV_L1_L1_C[25:6]},
    {CAV_L1_L2_D, TD_BP_I4[9:8], XC_BP[7:6]},
    CAV_L2_L1_D,
    CAV_L2_L1_C
);
defparam CSA_L2_L1.DW = 27;

```

```

// Level 2 Line 2
CSA_L CSA_L2_L2 (
    {{6{CAV_L1_L2_C[33]}}, CAV_L1_L2_C[33:14]},
    {CAV_L1_L3_D, XC_BP[15:14]},
    {CAV_L1_L3_C[39:17], 1'b0, ACE_T[15:14]},
    CAV_L2_L2_D,
    CAV_L2_L2_C
);
defparam CSA_L2_L2.DW = 26;

// Level 3 Line 1
_42C_L _42C_L_L3_L1 (
    {{7{CAV_L2_L1_D[32]}}, CAV_L2_L1_D, CAV_L1_L1_D[5:2], TD_BP_IO[1:0]},
    {{6{CAV_L2_L1_C[33]}}, CAV_L2_L1_C, 1'b0, CAV_L1_L1_C[5:3], 1'b0, XC_BP[1:0]},
    {CAV_L2_L2_D, CAV_L1_L2_C[13:11], 1'b0, XC_BP[9:8], 8'b0000_0000},
    {CAV_L2_L2_C[39:15], 1'b0, ACE_T[13:0]},
    1'b0,
    CAV_L3_L1_D,
    CAV_L3_L1_C,
);
defparam _42C_L_L3_L1.DW = 40;

assign TES = CAV_L3_L1_D;
assign TEC = {CAV_L3_L1_C, 1'b0};

endmodule

module Booth (
    Encode, //
    Negate, //
    Source, //
    Result, //
    Carry //
);

parameter DW = 8;

input [ 2:0] Encode;
input      Negate;
input [ DW:0] Source;
output[(DW+1):0] Result;
output[ 1:0] Carry;

// * * * * RTL Level * * * *

```

```

wire      Add_Sub, // add(0) or sub(1)
          Once_Valid, // once is valid if it is '1' else zero
          Twice_Enable; // twice is valid when it is '1' else zero

assign Add_Sub = Encode[2] ^ Negate;
assign Once_Valid = Encode[1] ^ Encode[0];
assign Twice_Enable = ((Encode == 3'b011) | (Encode == 3'b100));

assign Result = (((Source[DW], Source) ^ {(DW+2){Add_Sub}}) & {(DW+2){Once_Valid}} |
                (({Source, 1'b0} ^ {(DW+2){Add_Sub}}) & {(DW+2){Twice_Enable}}));
assign Carry = {1'b0, (Add_Sub & (Once_Valid | Twice_Enable))};

endmodule

```

```

module CSA_L (
    A, //
    B, //
    Ci, //
    D, //
    Co //
);

parameter    DW = 8;

input [(DW-1):0] A, B, Ci;
output [(DW-1):0] D;
output [DW:1] Co;

assign D = A ^ B ^ Ci;
assign Co = A & B | B & Ci | Ci & A;

endmodule

```

```

module Carry_Leading_4 (
    G, //
    P, //
    Ci, //
    C, //
    Gx, //
    Px //
);

input [ 3:0] G, P;

```

```

input      Ci;
output[ 4:1] C;
output     Gx,Px;

assign C[1] = G[0] | P[0] & Ci;
assign C[2] = G[1] | P[1] & G[0] | P[1] & P[0] & Ci;
assign C[3] = G[2] | P[2] & G[1] | P[2] & P[1] & G[0] | P[2] & P[1] & P[0] & Ci;
assign C[4] = Gx | Px & Ci;

assign Gx = G[3] | P[3] & G[2] | P[3] & P[2] & G[1] | P[3] & P[2] & P[1] & G[0];
assign Px = P[3] & P[2] & P[1] & P[0];

endmodule

```

其测试程序如下:

```

module Mul_DD_D2_A1_V1_T ();

reg  [15:0] AD,BD;
reg      AS,BS;
reg  [39:0] ACC;
wire [39:0] DR;
reg  [ 1:0] CC;

wire [39:0] TMD,TD;
reg      CR;

Mul_DD_D2_A1_V1 Mul_DD_D2_A1_V1_C (
    AD, // A operand
    AS, // extend style for A
    BD, // B operand
    BS, // extend style for B
    ACC, // Accumulator operand
    DR, // result data
    CC // control code
);

assign TMD = {{24{(AS & AD[15])}},AD} * {{24{(BS & BD[15])}},BD};
assign TD = (CC[1] ? ACC : 1'b0) + (CC[0] ? (- TMD) : TMD);

always
    # 10 CR = (DR == TD);

always
    # 10 {AD, BD, AS, BS, ACC, CC} = {$random, $random, $random, $random};

```

```
endmodule
```

此处加法器的 P 函数为： $P = A \wedge B$ ，而在《算术逻辑部件设计》一文中， $P = A \mid B$ 。这样做，是因为此处为纯算术运算。

此例为纯组合逻辑实现。为提高运算速度，各位也可将其改为流水线式。

前面说过：二数——位宽为 N 的数据 A、位宽为 M 的数据 B——相乘，乘积结果最低 N（或 M，取小者）位，不论是对符号数还是无符号数均相同。此即为测试程序：

```
module Mul_DD_D2_A1_V2_T ();

    reg    [15:0]  AD, BD;
    reg          AS, BS;
    reg    [39:0]  ACC;
    wire   [39:0]  DR_CP, DR_CC;
    reg     [ 1:0]  CC;

    reg          CR;

    Mul_DD_D2_A1_V1 Mul_DD_D2_A1_V1_CP (
        AD, // A operand
        AS, // extend style for A
        BD, // B operand
        BS, // extend style for B
        {40{1'b0}}, // Accumulator operand
        DR_CP, // result data
        2'b00 // control code
    );

    Mul_DD_D2_A1_V1 Mul_DD_D2_A1_V1_CC (
        AD, // A operand
        (~ AS), // extend style for A
        BD, // B operand
        (~ BS), // extend style for B
        {40{1'b0}}, // Accumulator operand
        DR_CC, // result data
        2'b00 // control code
    );

    always
        # 10 CR = (DR_CP[15:0] == DR_CC[15:0]);

    always
        # 10 {AD, BD, AS, BS} = {$random, $random, $random};
endmodule
```


当然，再设一加法器亦无不可，但就是太浪费了，有违节省资源之初衷。

再就是绕过布思编码及计数器阵列，将累加操作数及乘法结果直接送至加法器。此法对左图方案较为容易，但对右图方案可就有点难——主要在时序的处理上。

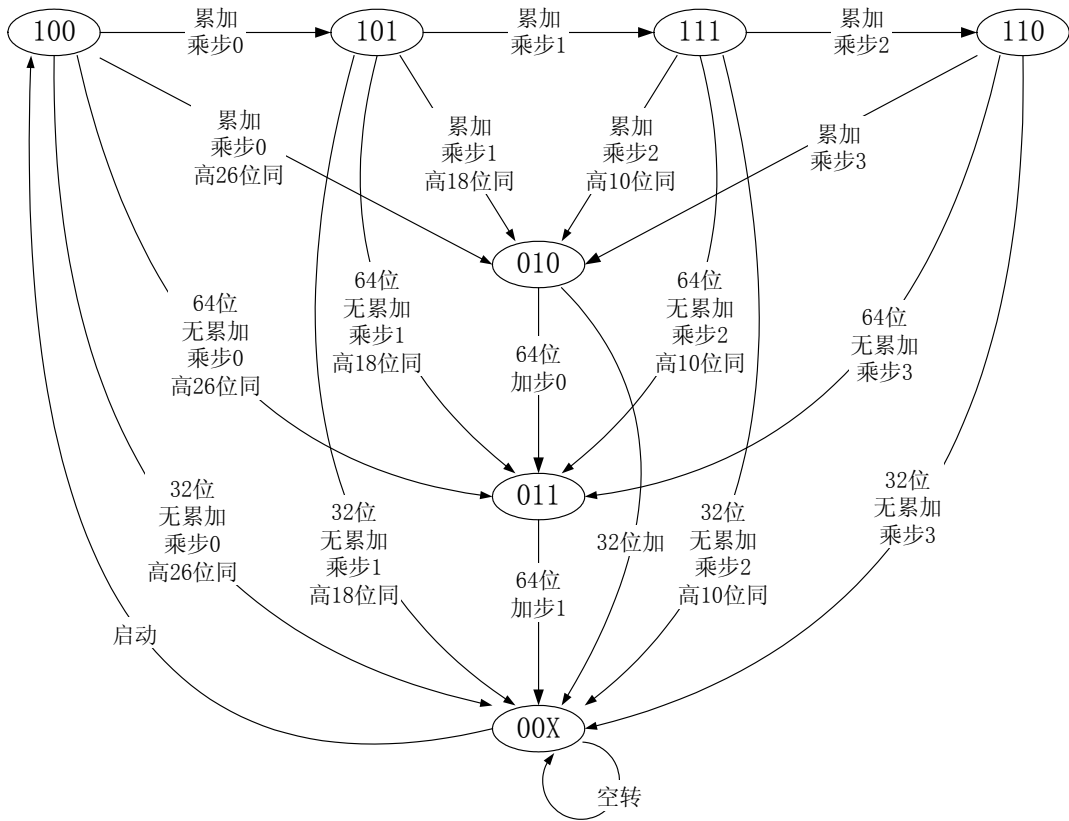
由于一数乘上1，积值为原数值。可以将累加操作视为1乘累加操作数，再累加前次运算结果。这就完全符合前面的乘法操作时序，处理起来可就非常简单、方便。

由于没有准备64位加法器，因此，对64位乘法及64位累加就需要用两个周期分两步才能实现64位加法。

对64位累加，在前一个周期累加低部分，后一个周期累加高部分。

对64位乘法，则是累加上0。而累加上低部分0的操作，则可以省掉。

按前述，作状态转换图如下：



在状态转换上，若下一状态为00X，则可以接收下一任务。但在状态图上没有表现。

1、 实现方案1 —— 乘、加一步走

此处实现前述左图方案。

```

module Mul_DD_D3_A1_V1 (
    A, // A operand
    B, // B operand
    D, // result data

```

```

    Sign,//
    OW,// 0: 32 bit operate ; 1: 64 bit operate
    AC,// Accumulate enable
    StartUp,//
    Busy,//
    ALL,// Load the Low part of the accumulate operand
    ALH,// Load the High part of the accumulate operand
    SRL,// Advice others device to save the result's Low part
    SRH,// Advice others device to save the result's High part
    ASI_I,// Input the annexation synchro information
    ASI_O_A,// for Loading accumulation operand
    ASI_O_S,// for saving result
    Clock//
);

parameter    ASI_W = 8;

input  [31:0]  A,B;
output [31:0]  D;
input        Sign,OW;
input        StartUp;
output       Busy;
input        AC;
output       ALH,ALL;
output       SRH,SRL;
input  [ (ASI_W-1):0]  ASI_I;
output [ (ASI_W-1):0]  ASI_O_A,ASI_O_S;
input        Clock;

reg        SRH,SRL;
reg  [ (ASI_W-1):0]  ASI_O_S;

wire        Same_up;
reg  [ 2:0]  State,Next;
wire        Load;

reg  [32:0]  AR,BR;
reg         ARx;
reg         OWR,ACR;
reg  [63:0]  DR;

reg  [31:0]  ACV;// accumulate value
reg         ACL;// carry when accumulating
wire [39:0]  TMR;// temp data of multiplication

reg  [ (ASI_W-1):0]  ASI_M[3'b111:3'b000];

```

```

// * * output

assign D = DR;

assign Busy = (! Next[2:1]);

assign ALL = (Next == 3'b010);
assign ALH = (Next == 3'b011) & ACR;

always @(negedge Clock)
  casex(State)
    3'b100:  {SRH, SRL} <= {1'b0, (Same_up & (~ ACR))};
    3'b101:  {SRH, SRL} <= {1'b0, (Same_up & (~ ACR))};
    3'b111:  {SRH, SRL} <= {1'b0, (Same_up & (~ ACR))};
    3'b110:  {SRH, SRL} <= {1'b0, (~ ACR)};
    3'b010:  {SRH, SRL} <= {1'b0, 1'b1};
    3'b011:  {SRH, SRL} <= {1'b1, 1'b0};
    default: {SRH, SRL} <= {1'b0, 1'b0};
  endcase

// * * The operation of the data register

always @(negedge Clock)
  casex(State)
    3'b100:  DR <= Same_up ? {{24{TMR[39]}}, TMR} : {TMR, {24{1'bx}}};
    3'b101:  DR <= Same_up ? {{16{TMR[39]}}, TMR, DR[31:24]} : {TMR, DR[31:24], {16{1'bx}}};
    3'b111:  DR <= Same_up ? {{8{TMR[39]}}, TMR, DR[31:16]} : {TMR, DR[31:16], {8{1'bx}}};
    3'b110:  DR <= {TMR, DR[31:8]};
    3'b010:  DR <= {DR[63:32], TMR[31:0]};
    3'b011:  DR <= TMR;
    default: DR <= {64{1'bx}};
  endcase

always @(negedge Clock)
  if(Load)
    AR <= {(Sign & A[31]), A};
  else
    if((Next == 3'b010) | (Next == 3'b011))
      AR <= {32{1'b0}};
    else
      AR <= {{8{AR[32]}}, AR[32:8]};

always @(negedge Clock)
  if(Load | ALL | ALH)
    BR <= {(Sign & B[31]), B};

```

```

always @(negedge Clock)
  if(Load)
    ARx <= 1'b0;
  else
    if(Next == 3'b010)
      ARx <= 1'b1;
    else
      if(Next == 3'b011)
        if(ACR)
          ARx <= 1'b1;
        else
          ARx <= 1'b0;
      else
        ARx <= AR[7];

always @(negedge Clock)
  if(Load)
    {OWR, ACR} <= {OW, AC};

// * * detect the up 26 bit value of the AR register
assign # 1 Same_up = (& AR[32:7]) | (! AR[32:7]);
//assign # 1 Same_up = (& AR[32:7]) | (! AR[32:8]);

// * * State Switch

always @(negedge Clock)
  if(Load)
    State <= 3'b100;
  else
    State <= Next;

always @(State or Same_up or ACR or OWR)
  casex(State)
    3'b100:
      if(Same_up)
        if(ACR)
          Next = 3'b010;
        else
          if(OWR)
            Next = 3'b011;
          else
            Next = 3'b00x;
      else
        Next = 3'b101;
    3'b101:

```

```

    if(Same_up)
        if(ACR)
            Next = 3'b010;
        else
            if(OWR)
                Next = 3'b011;
            else
                Next = 3'b00x;
        else
            Next = 3'b111;
3'b111:
    if(Same_up)
        if(ACR)
            Next = 3'b010;
        else
            if(OWR)
                Next = 3'b011;
            else
                Next = 3'b00x;
        else
            Next = 3'b110;
3'b110:
    if(ACR)
        Next = 3'b010;
    else
        if(OWR)
            Next = 3'b011;
        else
            Next = 3'b00x;
3'b010:
    if(OWR)
        Next = 3'b011;
    else
        Next = 3'b00x;
default:
    Next = 3'b00x;
endcase

assign Load = (! Next[2:1]) & StartUp;

// * * process synchro information

reg [2:0] ASI_DN;

always @(negedge Clock)
begin
    ASI_M[{Next[2:1], ((| Next[2:1]) & Next[0])}] <=

```

```

        ASI_M[{State[2:1], ((| State[2:1]) & State[0])}];
    if(Load)
        ASI_M[3'b100] <= ASI_I;
    end

assign ASI_O_A = ASI_M[{State[2:1], ((| State[2:1]) & State[0])}];

always @(negedge Clock)
    ASI_O_S <= ASI_M[{State[2:1], ((| State[2:1]) & State[0])}];

// * * get the accumulation operand
always @(State or DR)
    casex(State)
        3'b100: ACV = {32{1'b0}};
        3'b101: ACV = DR[63:32];
        3'b111: ACV = DR[63:32];
        3'b110: ACV = DR[63:32];
        3'b010: ACV = DR[31:0];
        3'b011: ACV = DR[63:32];
        3'b00x: ACV = {32{1'bx}};
    endcase

// * * multiply

wire [39:0] TAR;// temp leading adder result

wire [39:0] TES,// temp counter array pseudo sum
           TEC;// temp counter array pseudo carry

// * * output

assign TMR = TAR;

always @(negedge Clock)
    if(Load)
        ACL <= 1'b0;
    else
        ACL <= (BR[32] ^ ACV[31] ^ TAR[32]) & (State == 3'b010);

// * * Carry Leading Adder

wire [39:0] TAG,TAP;//g & p
wire [40:0] TAC;// carry

```

```

assign TAR = TAP ^ TAC;

assign TAG = TES & TEC;
assign TAP = TES ^ TEC;

Carry_Leading_40 Carry_40 (TAG, TAP, ACL, TAC[40:1]);

assign TAC[0] = ACL;

// * * booth encode & booth process

wire [33:0] TD_BP_I0;
wire [35:2] TD_BP_I1;
wire [37:4] TD_BP_I2;
wire [39:6] TD_BP_I3;
wire [41:8] TD_BP_I4;
wire [ 7:0] XC_BP;

Booth Booth_I0 ({AR[1:0], ARx}, 1'b0, BR, TD_BP_I0, XC_BP[1:0]),
    Booth_I1 (AR[3:1], 1'b0, BR, TD_BP_I1, XC_BP[3:2]),
    Booth_I2 (AR[5:3], 1'b0, BR, TD_BP_I2, XC_BP[5:4]),
    Booth_I3 (AR[7:5], 1'b0, BR, TD_BP_I3, XC_BP[7:6]),
    Booth_I4 ((State == 3'b110) ? AR[9:7] : 3'b000), 1'b0, BR, TD_BP_I4,);
defparam Booth_I0.DW = 32, Booth_I1.DW = 32, Booth_I2.DW = 32,
    Booth_I3.DW = 32, Booth_I4.DW = 32;

// * * counter array

wire [35:0] CAV_L1_L1_D;
wire [36:1] CAV_L1_L1_C;
wire [39:6] CAV_L1_L2_D;
wire [40:7] CAV_L1_L2_C;

wire [40:2] CAV_L2_D;
wire [41:3] CAV_L2_C;

// Level 1
CSA_L CSA_L_L1_L1 (
    {{4{ACV[31]}}, ACV},
    {{2{TD_BP_I0[33]}}, TD_BP_I0},
    {TD_BP_I1, XC_BP[1:0]},
    CAV_L1_L1_D,
    CAV_L1_L1_C
);
defparam CSA_L_L1_L1.DW = 36;

```

```

CSA_L CSA_L_L1_L2 (
    {{2{TD_BP_I2[37]}}, TD_BP_I2[37:6]},
    TD_BP_I3,
    {TD_BP_I4[39:8], XC_BP[7:6]},
    CAV_L1_L2_D,
    CAV_L1_L2_C
);
defparam CSA_L_L1_L2.DW = 34;

// Level 2
_42C_L _42C_L_L2 (
    {{4{CAV_L1_L1_D[35]}}, CAV_L1_L1_D[35:2]},
    {{3{CAV_L1_L1_C[36]}}, CAV_L1_L1_C[36:2]},
    {CAV_L1_L2_D[39:6], TD_BP_I2[5:4], XC_BP[3:2]},
    {CAV_L1_L2_C[39:7], 1'b0, XC_BP[5:4], 2'b00},
    1'b0,
    CAV_L2_D,
    CAV_L2_C,
);
defparam _42C_L_L2.DW = 38;

assign TES = {CAV_L2_D, CAV_L1_L1_D[1:0]};
assign TEC = {CAV_L2_C, 1'b0, CAV_L1_L1_C[1], 1'b0};

endmodule

module Carry_Leading_40 (
    G, //
    P, //
    Ci, //
    C //
);

input [39:0] G, P;
input      Ci;
output[40:1] C;

wire [11:0] G_S0, P_S0;
wire [ 3:0] G_S1, P_S1;
wire [40:1] C_S0;
wire [12:1] C_S1;
wire [ 4:1] C_S2;

assign C = C_S0;

Carry_Leading_4 CL_S0_P0 (G[ 3: 0], P[ 3: 0], Ci, C_S0[ 4: 1], G_S0[0], P_S0[0]),

```

```

        CL_S0_P1 (G[ 7: 4],P[ 7: 4],C_S1[1],C_S0[ 8: 5],G_S0[1],P_S0[1]),
        CL_S0_P2 (G[11: 8],P[11: 8],C_S1[2],C_S0[12: 9],G_S0[2],P_S0[2]),
        CL_S0_P3 (G[15:12],P[15:12],C_S1[3],C_S0[16:13],G_S0[3],P_S0[3]),
        CL_S0_P4 (G[19:16],P[19:16],C_S1[4],C_S0[20:17],G_S0[4],P_S0[4]),
        CL_S0_P5 (G[23:20],P[23:20],C_S1[5],C_S0[24:21],G_S0[5],P_S0[5]),
        CL_S0_P6 (G[27:24],P[27:24],C_S1[6],C_S0[28:25],G_S0[6],P_S0[6]),
        CL_S0_P7 (G[31:28],P[31:28],C_S1[7],C_S0[32:29],G_S0[7],P_S0[7]),
        CL_S0_P8 (G[35:32],P[35:32],C_S1[8],C_S0[36:33],G_S0[8],P_S0[8]),
        CL_S0_P9 (G[39:36],P[39:36],C_S1[9],C_S0[40:37],G_S0[9],P_S0[9]);

Carry_Leading_4 CL_S1_P0 (G_S0[ 3:0],P_S0[ 3:0],Ci      ,C_S1[ 4:1],G_S1[0],P_S1[0]),
        CL_S1_P1 (G_S0[ 7:4],P_S0[ 7:4],C_S2[1],C_S1[ 8:5],G_S1[1],P_S1[1]),
        CL_S1_P2 (G_S0[11:8],P_S0[11:8],C_S2[2],C_S1[12:9],G_S1[2],P_S1[2]);

Carry_Leading_4 CL_S2_P0 (G_S1,P_S1,Ci,C_S2[4:1],,);

endmodule

```

此处用到一种同步技术，即将同步信息与待处理数据一起送入，在器件内部，让此同步信息跟着数据一起走，这样同步就好处理了！

因何用此，因为 Startup 有可能先于 SRL、SRH 发出，而 AAL 与 SRL 可能同时发出，这就可能出现混乱，而此种同步技术则很容易处理。

因为 64 位是分作两次 32 位累加。由此涉及到高、低部分累加时之间进位。

各位可能想到，加法器中不就有进位，直接用不就得了吗，干嘛弄得如此复杂，浪费！

这是因为累加操作数为 BR、ACV，它们经过布思编码时都没有改变，但经过计数器阵列后则发生了变化。这时送入加法器的数据已不复为 BR、ACV，其进位也不复为 BR、ACV 之间的进位。因此，需由结果与 BR、ACV 反求进位！

其测试程序如下：

```

module Mul_DD_D3_A1_V1_T ();

    reg    [31:0]  A,B;
    reg          S;
    wire   [31:0]  D;

    reg          OW;
    wire         StartUp;
    wire         Busy;
    reg          AC;
    wire         ALH,ALL;
    wire         SRH,SRL;
    reg    [ 1:0] ASI_I;
    wire    [ 1:0] ASI_O_A,ASI_O_S;
    reg          Clock;

```

```

reg [ 129:0] RS_B[3:0],RS_AI;
wire [ 129:0] RS_A0,RS_CO;
reg [63:0] DR,TD;

reg [31:0] CC;

reg CR;

Mul_DD_D3_A1_V1 Mul_DD_D3_A1_V1_C (
    A, // A operand
    B, // B operand
    D, // result data
    S, //
    OW, // 0: 32 bit operate ; 1: 64 bit operate
    AC, // Accumulate enable
    StartUp, //
    Busy, //
    ALL, // Load the Low part of the accumulate operand
    ALH, // Load the High part of the accumulate operand
    SRL, // Advice others device to save the result's Low part
    SRH, // Advice others device to save the result's High part
    ASI_I, // Input the annexation synchro information
    ASI_O_A, // for Loading accumulation operand
    ASI_O_S, // for saving result
    Clock //
);
defparam Mul_DD_D3_A1_V1_C.ASI_W = 2;

always @(negedge Clock)
begin
    if(SRL)
        DR <= {{32{1'b0}},D};
    if(SRH)
        DR[63:32] <= D;
end

always @(negedge Clock)
if(SRL | SRH)
    TD <= ({{32{RS_CO[128] & RS_CO[31]}},RS_CO[31:0]} *
        {{32{RS_CO[128] & RS_CO[63]}},RS_CO[63:32]} + RS_CO[127:64]) &
        {{32{RS_CO[129]}}, {32{1'b1}}};

always @(negedge Clock)
    CR <= (! (TD ^ DR)) | SRH;

initial

```

```

ASI_I <= 0;

always @(negedge Clock)
  if(StartUp)
    # 1 ASI_I <= ASI_I + 2'b01;

always @(negedge Clock)
  RS_B[(StartUp ? ASI_I : ASI_O_A)] <= RS_AI;

assign RS_AO = RS_B[(StartUp ? ASI_I : ASI_O_A)];
assign RS_CO = RS_B[ASI_O_S];

always @(StartUp or OW or S or B or A or RS_AO or ALL or ALH)
  if(StartUp)
    RS_AI = {OW, S, {64{1'b0}}, B, A};
  else
    if(ALL)
      RS_AI = {RS_AO[129:128], {32{1'b0}}, B, RS_AO[63:0]};
    else
      if(ALH)
        RS_AI = {RS_AO[129:128], B, RS_AO[95:64], RS_AO[63:0]};
      else
        RS_AI = RS_AO;

assign # 1 StartUp = ((& (CC[3:2] ^ CC[1:0])) & (~ Busy));

// * * Generate operand
reg [31:0] ATD;

always @(negedge Clock)
  # 1 {A, ATD, B, S, OW, AC, CC} <= {$random, $random, $random, $random, $random};

always @(ATD or CC)
  if(CC[31:30] ^ CC[29:28] ^ CC[27:26] ^ CC[25:24])
    A = ({32{A[CC[22:18] ^ CC[17:13] ^ CC[12:8]]}} << (CC[22:18] ^ CC[17:13] ^ CC[12:8])) |
      (~ ({32{1'b1}} << (CC[22:18] ^ CC[17:13] ^ CC[12:8]))) & ATD;
  else
    A = ATD;

initial
  Mul_DD_D3_A1_V1_C.State <= 3'b00x;

always
  begin
    # 5 Clock <= 1'b1;
    # 5 Clock <= 1'b0;
  end

```

```

end

endmodule

```

2、实现方案2 —— 乘、加两步走

此处实现前述右图方案。

```

module Mul_DD_D3_A2_V1 (
    A, // A operand
    B, // B operand
    D, // result data
    Sign, //
    OW, // 0: 32 bit operate ; 1: 64 bit operate
    AC, // Accumulate enable
    StartUp, //
    Busy, //
    ALL, // Load the Low part of the accumulate operand
    ALH, // Load the High part of the accumulate operand
    SRL, // Advice others device to save the result's Low part
    SRH, // Advice others device to save the result's High part
    Clock //
);

input [31:0] A, B;
output [31:0] D;
input Sign, OW;
input StartUp;
output Busy;
input AC;
output ALH, ALL;
output SRH, SRL;
input Clock;

reg SRH, SRL;

wire Same_up;
reg [ 2:0] State, Next;
wire Load;

reg [32:0] AR, BR;
reg ARx;
reg OWR, ACR;

```

```

reg [68:0] DTRD,DTRC;// temp data result register
reg [ 1:0] CACC;// carry of accumulate

wire [44:0] TES, // temp counter array pseudo sum
          TEC; // temp counter array pseudo carry

reg [36:0] ACVD,ACVC;// accumulate value
reg        ACL;// carry when add
wire [31:0] TAR;// temp data of multiplication

// * * output

assign D = TAR;

assign Busy = (! Next[2:1]);

assign ALL = (Next == 3'b010);
assign ALH = (Next == 3'b011) & ACR;

always @(negedge Clock)
  casex(State)
    3'b100: {SRH, SRL} <= {1'b0, (Same_up & (~ ACR))};
    3'b101: {SRH, SRL} <= {1'b0, (Same_up & (~ ACR))};
    3'b111: {SRH, SRL} <= {1'b0, (Same_up & (~ ACR))};
    3'b110: {SRH, SRL} <= {1'b0, (~ ACR)};
    3'b010: {SRH, SRL} <= {1'b0, 1'b1};
    3'b011: {SRH, SRL} <= {1'b1, 1'b0};
    default: {SRH, SRL} <= {1'b0, 1'b0};
  endcase

// * * The operation of the data register

always @(negedge Clock)
  begin
    casex(State)
      3'b100: DTRD <= Same_up ? {{24{TES[44]}}, TES} : {TES, {24{1'bx}}};
      3'b101: DTRD <= Same_up ? {{16{TES[44]}}, TES, DTRD[31:24]} :
        {TES, DTRD[31:24], {16{1'bx}}};
      3'b111: DTRD <= Same_up ? {{8{TES[44]}}, TES, DTRD[31:16]} :
        {TES, DTRD[31:16], {8{1'bx}}};
      3'b110: DTRD <= {TES, DTRD[31:8]};
      3'b010: DTRD <= {DTRD[68:32], TES[31:0]};
      3'b011: DTRD <= {{37{1'bx}}, TES[31:0]};
      default: DTRD <= {69{1'bx}};
    endcase
  casex(State)

```

```

3'b100: DTRC <= Same_up ? {{24{TEC[44]}}, TEC} : {TEC, {24{1'bx}}};
3'b101: DTRC <= Same_up ? {{16{TEC[44]}}, TEC, DTRC[31:24]} :
        {TEC, DTRC[31:24], {16{1'bx}}};
3'b111: DTRC <= Same_up ? {{8{TEC[44]}}, TEC, DTRC[31:16]} :
        {TEC, DTRC[31:16], {8{1'bx}}};
3'b110: DTRC <= {TEC, DTRC[31:8]};
3'b010: DTRC <= {DTRC[68:32], TEC[31:0]};
3'b011: DTRC <= {{37{1'bx}}, TEC[31:0]};
default: DTRC <= {69{1'bx}};
endcase
end

always @(negedge Clock)
  CACC = (State == 3'b010) ? {(TEC[33] ^ (TES[32] & TEC[32])), (TES[32] ^ TEC[32])} : 2'b00;

always @(negedge Clock)
  if(Load)
    AR <= {(Sign & A[31]), A};
  else
    if((Next == 3'b010) | (Next == 3'b011))
      AR <= {32{1'b0}};
    else
      AR <= {{8{AR[32]}}, AR[32:8]};

always @(negedge Clock)
  if(Load)
    BR <= {(Sign & B[31]), B};
  else
    if(ALL | ALH)
      BR <= {1'b0, B};

always @(negedge Clock)
  if(Load)
    ARx <= 1'b0;
  else
    if(Next == 3'b010)
      ARx <= 1'b1;
    else
      if(Next == 3'b011)
        if(ACR)
          ARx <= 1'b1;
        else
          ARx <= 1'b0;
      else
        ARx <= AR[7];

always @(negedge Clock)

```

```

if(Load)
    {OWR, ACR} <= {OW, AC};

// * * detect the up 26 bit value of the AR register
assign Same_up = (& AR[32:7]) | (! AR[32:7]);

// * * State Switch

always @(negedge Clock)
    if(Load)
        State <= 3'b100;
    else
        State <= Next;

always @(State or Same_up or ACR or OWR)
    casex(State)
        3'b100:
            if(Same_up)
                if(ACR)
                    Next = 3'b010;
                else
                    if(OWR)
                        Next = 3'b011;
                    else
                        Next = 3'b00x;
            else
                Next = 3'b101;
        3'b101:
            if(Same_up)
                if(ACR)
                    Next = 3'b010;
                else
                    if(OWR)
                        Next = 3'b011;
                    else
                        Next = 3'b00x;
            else
                Next = 3'b111;
        3'b111:
            if(Same_up)
                if(ACR)
                    Next = 3'b010;
                else
                    if(OWR)
                        Next = 3'b011;
                    else

```

```

        Next = 3' b00x;
    else
        Next = 3' b110;
3' b110:
    if(ACR)
        Next = 3' b010;
    else
        if(OWR)
            Next = 3' b011;
        else
            Next = 3' b00x;
3' b010:
    if(OWR)
        Next = 3' b011;
    else
        Next = 3' b00x;
    default:
        Next = 3' b00x;
endcase

assign Load = (! Next[2:1]) & StartUp;

// * * get the accumulation operand
always @(State or DTRD or DTRC)
begin
    casex(State)
        3' b100:  ACVD = {37{1' b0}};
        3' b101:  ACVD = DTRD[68:32];
        3' b111:  ACVD = DTRD[68:32];
        3' b110:  ACVD = DTRD[68:32];
        3' b010:  ACVD = {5' b00000, DTRD[31:0]};
        3' b011:  ACVD = {5' b00000, DTRD[63:32]};
        3' b00x:  ACVD = {37{1' bx}};
    endcase
    casex(State)
        3' b100:  ACVC = {37{1' b0}};
        3' b101:  ACVC = DTRC[68:32];
        3' b111:  ACVC = DTRC[68:32];
        3' b110:  ACVC = DTRC[68:32];
        3' b010:  ACVC = {5' b00000, DTRC[31:0]};
        3' b011:  ACVC = {5' b00000, DTRC[63:32]};
        3' b00x:  ACVC = {37{1' bx}};
    endcase
end

// * * Carry Leading Adder

```

```

wire [31:0] TAG, TAP; //g & p
wire [32:0] TAC; // carry

always @(negedge Clock)
    ACL <= TAC[32] & (State == 3'b011);

assign TAR = TAP ^ TAC;

assign TAG = DTRD & DTRC;
assign TAP = DTRD ^ DTRC;

wire [32:1] TACt;

Carry_Leading_32 Carry_32 (TAG, TAP, ACL, TACt);

assign TAC = {TACt, ACL};

// * * multiply

// * * booth encode & booth process

wire [33:0] TD_BP_I0;
wire [35:2] TD_BP_I1;
wire [37:4] TD_BP_I2;
wire [39:6] TD_BP_I3;
wire [41:8] TD_BP_I4;
wire [ 7:0] XC_BP;

Booth Booth_I0 ({AR[1:0], ARx}, 1'b0, BR, TD_BP_I0, XC_BP[1:0]),
    Booth_I1 (AR[3:1], 1'b0, BR, TD_BP_I1, XC_BP[3:2]),
    Booth_I2 (AR[5:3], 1'b0, BR, TD_BP_I2, XC_BP[5:4]),
    Booth_I3 (AR[7:5], 1'b0, BR, TD_BP_I3, XC_BP[7:6]),
    Booth_I4 (((State == 3'b110) ? AR[9:7] : 3'b000), 1'b0, BR, TD_BP_I4,);
defparam Booth_I0.DW = 32, Booth_I1.DW = 32, Booth_I2.DW = 32,
    Booth_I3.DW = 32, Booth_I4.DW = 32;

// * * counter array

wire [40:2] CAV_L1_L1_D;
wire [41:3] CAV_L1_L1_C;
wire [41:6] CAV_L1_L2_D;
wire [42:7] CAV_L1_L2_C;

wire [43:0] CAV_L2_D;
wire [44:1] CAV_L2_C;

```

```

// Level 1 Line 1
_42C_L _42C_L_L1_L1 (
    {{6{TD_BP_IO[33]}}, TD_BP_IO[33:2]},
    {{4{TD_BP_I1[35]}}, TD_BP_I1},
    {{2{TD_BP_I2[37]}}, TD_BP_I2, XC_BP[3:2]},
    {TD_BP_I3, XC_BP[5:4], 2'b00},
    1'b0,
    CAV_L1_L1_D,
    CAV_L1_L1_C,
);
defparam _42C_L_L1_L1.DW = 38;

// Level 1 Line 2
CSA_L CSA_L1_L2 (
    {TD_BP_I4, XC_BP[7:6]},
    {{5{ACVD[36]}}, ACVD[36:6]},
    {{5{ACVC[36]}}, ACVC[36:6]},
    CAV_L1_L2_D,
    CAV_L1_L2_C
);
defparam CSA_L1_L2.DW = 36;

// Level 2
_42C_L _42C_L_L2 (
    {CAV_L1_L1_D[40], CAV_L1_L1_D[40], CAV_L1_L1_D[40:2], TD_BP_IO[1:0]},
    {CAV_L1_L1_C[41], CAV_L1_L1_C[41:3], 1'b0, ((State == 3'b011) ? CACC : XC_BP[1:0])},
    {CAV_L1_L2_D[41], CAV_L1_L2_D[41:6], ACVD[5:0]},
    {CAV_L1_L2_C[42:7], 1'b0, ACVC[5:0]},
    1'b0,
    CAV_L2_D,
    CAV_L2_C,
);
defparam _42C_L_L2.DW = 43;

assign TES = {CAV_L2_D[43], CAV_L2_D};
assign TEC = {CAV_L2_C, 1'b0};

endmodule

```

仿照前面的例子，请各位将模块 Carry_Leading_32 实现。

此例有两处需要处理进位：

第一处是在加累加操作数时：DTRD[31:0]、DTRC[31:0]加 BR，DTRD[63:32]、DTRC[63:32]加 BR（此时 AR 为 0，ARx 为 0 或 1），进位由 CACC 寄存器保存；

第二处是在将二数——DTRD、DTRC——相加得到最后结果时，进位由 ACL 寄存器保存；

而在（累加）处理时，将所有数据都处理成无符号数：在取累加操作数时，将累加操作数作 0 扩展；执行累加操作数时，只取前面乘法结果的最低 32 位，在高位填充 0；在将二数加为最终结果时，也只取最低 32 位，在高位填充 0。这样，就没有必要如上例所示由结果反求进位。

其测试模块如下：

```

module Mul_DD_D3_A2_V1_T ();

    reg    [31:0]  A,B;
    reg          S;
    wire   [31:0]  D;
    reg          OW;
    wire          StartUp;
    wire          Busy;
    reg          AC;
    wire          ALH, ALL;
    wire          SRH, SRL;
    reg          Clock;

    reg    [31:0]  CC;

    reg    [63:0]  DR, TD;
    reg          CR;

    reg    [ 129:0]  RS_B[3:0], RS_AI;
    wire   [ 129:0]  RS_A0, RS_C0;
    reg    [ 1:0]   ASI_I, ASI_S;

    Mul_DD_D3_A2_V1 Mul_DD_D3_A2_V1_C (
        A, // A operand
        B, // B operand
        D, // result data
        S, //
        OW, // 0: 32 bit operate ; 1: 64 bit operate
        AC, // Accumulate enable
        StartUp, //
        Busy, //
        ALL, // Load the Low part of the accumulate operand
        ALH, // Load the High part of the accumulate operand
        SRL, // Advice others device to save the result's Low part
        SRH, // Advice others device to save the result's High part
        Clock //
    );

    always @(negedge Clock)
        begin

```

```

    if(SRL)
        DR <= {{32{1'b0}},D};
    if(SRH)
        DR[63:32] <= D;
end

always @(negedge Clock)
    if(SRL | SRH)
        TD <= ({{32{RS_CO[128] & RS_CO[31]}},RS_CO[31:0]} *
            {{32{RS_CO[128] & RS_CO[63]}},RS_CO[63:32]} + RS_CO[127:64]) &
            {{32{RS_CO[129]}}, {32{1'b1}}});

always @(negedge Clock)
    CR <= (! (TD ^ DR)) | SRH;

always @(negedge Clock)
    if(StartUp)
        # 1 ASI_I <= ASI_I + 2'b01;

always @(negedge Clock)
    if(SRL)
        # 1 ASI_S <= ASI_S + 2'b01;

always @(negedge Clock)
    RS_B[(StartUp ? ASI_I : (ASI_I - 2'b01))] <= RS_AI;

assign RS_AO = RS_B[(StartUp ? ASI_I : (ASI_I - 2'b01)) & 4'h3];
assign RS_CO = RS_B[(ASI_S - {1'b0,SRH}) & 4'h3];

always @(StartUp or OW or S or B or A or RS_AO or ALL or ALH)
    if(StartUp)
        RS_AI = {OW, S, {64{1'b0}}, B, A};
    else
        if(ALL)
            RS_AI = {RS_AO[129:128], {32{1'b0}}, B, RS_AO[63:0]};
        else
            if(ALH)
                RS_AI = {RS_AO[129:128], B, RS_AO[95:64], RS_AO[63:0]};
            else
                RS_AI = RS_AO;

initial
    {ASI_I, ASI_S} <= 0;

assign # 1 StartUp = ((& (CC[3:2] ^ CC[1:0])) & (~ Busy));

```

```

// * * Generate operand
reg [31:0] ATD;

always @(negedge Clock)
  # 1 {A, ATD, B, S, OW, AC, CC} <= {$random, $random, $random, $random, $random};

always @(ATD or CC)
  if(CC[31:30] ^ CC[29:28] ^ CC[27:26] ^ CC[25:24])
    A = ({32{A[CC[22:18] ^ CC[17:13] ^ CC[12:8]]}} << (CC[22:18] ^ CC[17:13] ^ CC[12:8])) |
      (~ ({32{1'b1}} << (CC[22:18] ^ CC[17:13] ^ CC[12:8]))) & ATD;
  else
    A = ATD;

initial
  Mul_DD_D3_A2_V1_C.State <= 3'b00x;

always
  begin
    # 5 Clock <= 1'b1;
    # 5 Clock <= 1'b0;
  end

endmodule

```

此处对同步的处理采用队列方式。

后 记

有道是：自己懂与让别人懂是两回事。作此资料，我是深有感触，语言文字不佳的我，自是感受颇深。此资料历时年余。

我多次看到及收到寻求乘法器设计理论或示例。我以前曾作过《快速乘法器设计》一文。因涉及版权问题，且谋篇布局不甚合理，遂自去年9月提笔作此资料。

在今年3月份时，此稿尚未完成，但因特殊原因，发布过一次。明显有违治学严谨之道。

本资料原定还有：用保留进位加法器实现迭代乘法。但我既是无从证明也是无法验证其正确性，因此，我即确定其理念就不正确。在我与李亚民先生联系后，也确定其不正确，也就将其删除了（本部分内容在我以前发布的版本中出现）。

说到李亚民先生，也就有必要介绍一下他的一本书：《计算机组成与系统结构》，2000年4月出版。本书系统地介绍了计算机体系结构，是学习计算机体系结构的一本极好教材；对研究微处理器有很大的参考价值（本人在微处理器研究中的“流水线”及“超标量”这两方面的突破性进展即于此受益）。“用保留进位加法器实现迭代乘法”内容即取材于此。

本资料中4-2压缩器部分，我最早见于《MPP嵌入式计算机设计》，沈绪榜著，1999年8月出版。本书主要对适度并行处理计算机系统硬件部分的设计作了深入细致的讲解，对浮点运算器设计作了详细介绍。

对于布思算法，我见过的可就多了，但只有一本对我影响很大，因为只有他如我般（应该是：我如他般）推导此算法。但非常遗憾的是，我无法在此列出此书及其作者，因为我在离开绵阳时，将那本书组送了人。而我的习惯——不喜作笔记——也实在不好，也就真叫做个记不起来。在此向作者表示感谢，并深表歉意。

蒋 小 龙
2002. 12. 29

个 人 介 绍

蒋小龙乃身份证名，亦用名：蒋维隆

集成电路前端逻辑设计。设计过 16 位、32 位微处理器。研究超标量微处理器

联系：jiangweilong@cmmail.com

jiangweilong@china.com

