中国科学院大学
University of Chinese Academy of Sciences

B0911006Y-01 2024-2025学年春季学期

计算机组成原理

Principles of Computer Organization

第 10 讲 计算机中数的运算IV

Booth乘法、快速乘法器、定点除法

主讲教师：石 侃

shikan@ict.ac.cn

2025年3月26日

④ Booth 算法递推公式

6.3

$$[x \cdot y]_补 = [x]_补[(y_1 - y_0) + (y_2 - y_1)2^{-1} + \cdots + (y_{n+1} - y_n)2^{-n}]$$

$[z_0]_补 = 0$

$[z_1]_补 = 2^{-1}\{(y_{n+1} - y_n)[x]_补 + [z_0]_补\}$ $\quad y_{n+1} = 0$

⋮

$[z_n]_补 = 2^{-1}\{(y_2 - y_1)[x]_补 + [z_{n-1}]_补\}$

$\boxed{-[x]_补 = +[-x]_补}$

$[x \cdot y]_补 = [z_n]_补 + (y_1 - y_0)[x]_补$ 最后一步不移位

如何实现

$y_{i+1} - y_i$ ？

y_i y_{i+1}	$y_{i+1} - y_i$	部分积的操作
0 0	0	$\rightarrow 1$
0 1	1	$+[x]_补 \rightarrow 1$
1 0	-1	$+[-x]_补 \rightarrow 1$
1 1	0	$\rightarrow 1$

Invented by Andrew Donald Booth in 1950 while doing research on crystallography at Birkbeck College in Bloomsbury, London

可能运算过程中部分积绝对值大于1，因此部分积和被乘数取双符号位

例6.23 已知 $x = +0.0011 \quad y = -0.1011$ 求 $[x\cdot y]_{补}$ 6.3

$[x]_{补} = 0.0011$

$[y]_{补} = 1.0101$

$[-x]_{补} = 1.1101$

乘数的符号位需参加运算

	部分积	乘数 y_i	y_{i+1}	
解：	00.0000	1.0101	0	
	+11.1101			$+[-x]_{补}$
	11.1101			
补码右移	11.1110	1 1010	1	$\rightarrow 1$（这里附加位参与右移！这里不是C_j）
	+00.0011			$+[x]_{补}$
	00.0001	1		
补码右移	00.0000	11 101	0	$\rightarrow 1$
	+11.1101			$+[-x]_{补}$
	11.1101	11		
补码右移	11.1110	111 10	1	$\rightarrow 1$
	+00.0011			$+[x]_{补}$
	00.0001	111		
补码右移	00.0000	1111 1	0	$\rightarrow 1$
	+11.1101			$+[-x]_{补}$
	11.1101	1111		最后一步不移位

$y_i \; y_{i+1}$	$y_{i+1}-y_i$	部分积的操作
0 0	0	$\rightarrow 1$
0 1	1	$+[x]_{补} \rightarrow 1$
1 0	-1	$+[-x]_{补} \rightarrow 1$
1 1	0	$\rightarrow 1$

$\therefore [x\cdot y]_{补}$

$=1.11011111$

y_i y_{i+1}	$y_{i+1}-y_i$	部分积的操作
0 0	0	$\rightarrow 1$
0 1	1	$+[x]_{补} \rightarrow 1$
1 0	-1	$+[-x]_{补} \rightarrow 1$
1 1	0	$\rightarrow 1$

例 6.21 已知 $[x]_{补}=0.1101$，$[y]_{补}=0.1011$，求 $[x \cdot y]_{补}$。

可能运算过程中部分积绝对值大于1，因此部分积和被乘数取双符号位

解：表 6.16 列出了例 6.21 的求解过程。

乘数的符号位需参加运算

表 6.16 例 6.21 求 $[x \cdot y]_{补}$ 的过程

部分积	乘数 y_n	附加位 y_{n+1}	说明
00.0000 + 11.0011	0101$\underline{1}$	$\underline{0}$	初值 $[z_0]_{补}=0$ $y_ny_{n+1}=10$，部分积加 $[-x]_{补}$
11.0011 11.1001 11.1100 + 00.1101	 1010$\underline{1}$ 1101$\underline{0}$	 $\underline{1}$ $\underline{1}$	 $\rightarrow 1$ 位，得 $[z_1]_{补}$ $y_ny_{n+1}=11$，部分积 $\rightarrow 1$ 位，得 $[z_2]_{补}$ $y_ny_{n+1}=01$，部分积加 $[x]_{补}$
00.1001 00.0100 + 11.0011	11 1110$\underline{1}$	 $\underline{0}$	 $\rightarrow 1$ 位，得 $[z_3]_{补}$ $y_ny_{n+1}=10$，部分积加 $[-x]_{补}$
11.0111 11.1011 + 00.1101	111 1111$\underline{0}$	 $\underline{1}$	 $\rightarrow 1$ 位，得 $[z_4]_{补}$ $y_ny_{n+1}=01$，部分积加 $[x]_{补}$
00.1000	1111		最后一步不移位，得 $[x \cdot y]_{补}$

故 $[x \cdot y]_{补}=0.10001111$

y_i y_{i+1}	$y_{i+1}-y_i$	部分积的操作
0 0	0	→1
0 1	1	$+[x]_补$ →1
1 0	-1	$+[-x]_补$ →1
1 1	0	→1

例 6.22 已知 $[x]_补=1.0101$，$[y]_补=1.0011$，求 $[x$

部分积	乘数 y_n	附加位 y_{n+1}	说明
00.0000 +00.1011	10011	0	 $y_n y_{n+1}=10$，部分积加 $[-x]_补$
00.1011 00.0101 00.0010 +11.0101	 11001 11100	 1 1	 →1 位，得 $[z_1]_补$ $y_n y_{n+1}=11$，部分积→1 位，得 $[z_2]_补$ $y_n y_{n+1}=01$，部分积加 $[x]_补$
11.0111 11.1011 11.1101 +00.1011	11 11110 11111	 0 0	 →1 位，得 $[z_3]_补$ $y_n y_{n+1}=00$，部分→1 位，得 $[z_4]_补$ $y_n y_{n+1}=10$，部分积加 $[-x]_补$
00.1000	1111		最后一步不移位，得 $[x\cdot y]_补$

故 $[x\cdot y]_补=0.10001111$

Long Table → Short Table

算的省心（符号任意）、算的快（某些情况仅做右移操作）

Booth布斯算法举例

【这里的P即前述的部分积z，另外这里y的脚标序号是递减的，前述是递增的】

已知$[X]_补$ = 1_0110_1101，$[Y]_补$ = 0_1111_1110，计算$[X\times Y]_补$

$[-X]_补$ = 0_1001_0011 X=-147，Y=254，X×Y=-37338，

$[X\times Y]_补$应等于1_0110_1110_0010_0110 **验证：$[Y]_补$中有连续1时，加速明显**

P	Y	y_{-1}	说明
00_0000_0000	0_1111_1110	0	y_{-1}为0，$[P_0]_补$为0
00_0000_0000	0 0111_1111	0	$y_0y_{-1}=00$, P Y右移一位
+ 00_1001_0011 00_1001_0011 00_0100_1001	10 011_1111	1	$y_1y_0=10$, $+[-x]_补$， P Y右移一位
00_0010_0100	110 01_1111	1	$y_2y_1=11$, P Y直接右移一位
00_0001_0010	0110 0_1111	1	$y_3y_2=11$, P Y直接右移一位
00_0000_1001	0_0110 0111	1	$y_4y_3=11$, P Y直接右移一位
00_0000_0100	10_0110 011	1	$y_5y_4=11$, P Y直接右移一位
00_0000_0010	010_0110 01	1	$y_6y_5=11$, P Y直接右移一位
00_0000_0001	0010_0110 0	1	$y_7y_6=11$, P Y直接右移一位
+ 11_0110_1101 11_0110_1110	0010_0110		$y_8y_7=01$, $+[x]_补$， 最后一步不移位

Booth算法的实质和优点

y_i y_{i+1}	$y_{i+1}-y_i$	部分积的操作
0 0	0	$\rightarrow 1$
0 1	1	$+[x]_{补} \rightarrow 1$
1 0	-1	$+[-x]_{补} \rightarrow 1$
1 1	0	$\rightarrow 1$

乘数的一轮移位操作：

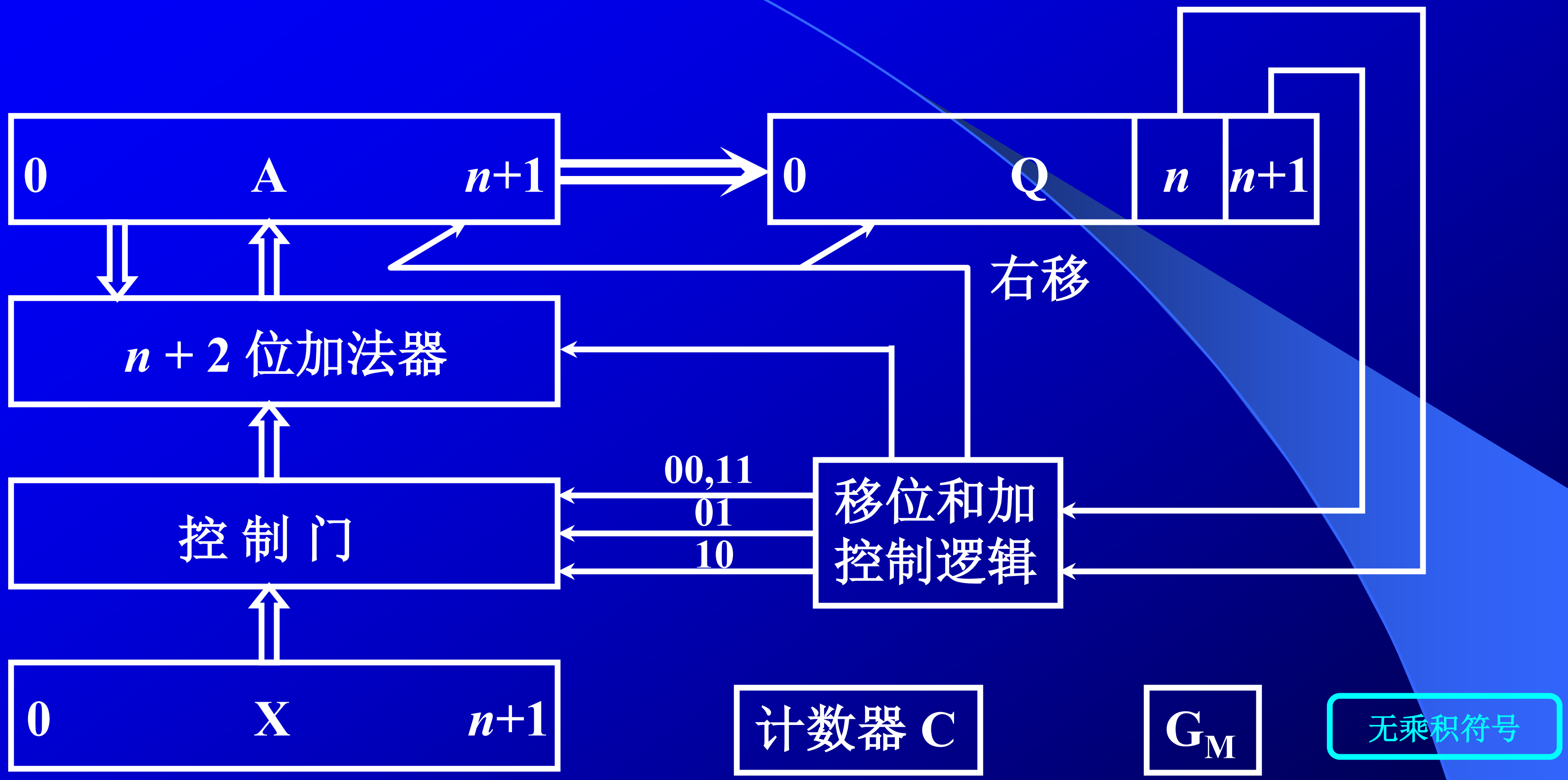
middle of run

end of run | 0 | 1 | 1 | 1 | 1 | 0 | beginning of run

当前位 y_i	右边位 y_{i+1}	除移位外的操作	Example
1	0	减被乘数	0001111000
1	1	加0 (不操作)	0001111000
0	1	加被乘数	0001111000
0	0	加0 (不操作)	0001111000

- 在“1串”中，第一个1时做减法，最后一个1做加法，其余情况只要移位
- 同前面算法一样，将乘积寄存器右移一位。（这里是算术右移）
- 假设一个8位乘数：0111_1110，它将产生6行非零的部分积。如果把该数字记成另一种形式，(1)000_00(-1)0（-1是负1），则只需相加两个部分积，可以大大减少非零行的数目，意味着相加次数的减少，从而加快了运算速度
- Booth算法可以减少部分积的数目，用来计算有符号乘法，提高乘法速度

(2) Booth 算法的硬件配置



A、X、Q均为$n+2$位寄存器(A与X为双符号位，Q为单符号位+附加位)

移位和加操作受乘数的末两位控制

补充知识：补码两位乘法

- 补码两位乘可用布斯算法推导如下：【这里的P即前述的部分积z，另外这里y的脚标序号是递减的，前述是递增的】
  - $[P_{i+1}]_补 = 2^{-1} ([P_i]_补 + (y_{i-1} - y_i) [X]_补)$
  - $[P_{i+2}]_补 = 2^{-1} ([P_{i+1}]_补 + (y_i - y_{i+1}) [X]_补)$

$$= 2^{-1} (2^{-1} ([P_i]_补 + (y_{i-1} - y_i) [X]_补) + (y_i - y_{i+1}) [X]_补)$$

$$= 2^{-2} ([P_i]_补 + (y_{i-1} + y_i - 2y_{i+1}) [X]_补)$$

- 开始置附加位y_{-1}为0，乘积寄存器最高位前面添加一位附加符号位0。
- 最终的乘积高位部分在乘积寄存器P中，低位部分在乘数寄存器Y中。
- 因为字长总是8的倍数，所以补码的位数n应该是偶数，因此，总循环次数为n/2。

y_{i+1}	y_i	y_{i-1}	操　作	迭代公式
0	0	0	0	$2^{-2}[P_i]_补$
0	0	1	$+[X]_补$	$2^{-2}\{[P_i]_补+[X]_补\}$
0	1	0	$+[X]_补$	$2^{-2}\{[P_i]_补+[X]_补\}$
0	1	1	$+2[X]_补$	$2^{-2}\{[P_i]_补+2[X]_补\}$
1	0	0	$+2[-X]_补$	$2^{-2}\{[P_i]_补+2[-X]_补\}$
1	0	1	$+[-X]_补$	$2^{-2}\{[P_i]_补+[-X]_补\}$
1	1	0	$+[-X]_补$	$2^{-2}\{[P_i]_补+[-X]_补\}$
1	1	1	0	$2^{-2}[P_i]_补$

小结：

1. 原码与补码的乘法，根本区别在于对符号位的处理
2. 补码（一位/两位）乘法：符号和数值一起运算，运算结果的符号是在数值部分的运算过程中自然形成
   ① 补码一位乘、乘数为负数时，校正法（加$[-x]_{补}$）
   ② 补码一位乘、乘数正负任意，比较法（**Booth算法**）
3. 注意区别$[-x^*]_{补}$和$[-x]_{补}$
   ① 原码两位乘法中使用$[-x^*]_{补}$
   ② 补码乘法中使用$[-x]_{补}$

y_i y_{i+1}	$y_{i+1}-y_i$	部分积的操作
0 0	0	→1
0 1	1	$+[x]_{补}$ →1
1 0	-1	$+[-x]_{补}$→1
1 1	0	→1

4. 由于不同的机器数运算规则不同，运算器硬件组成各不相同（包括寄存器位数、全加器输入端控制电路等）

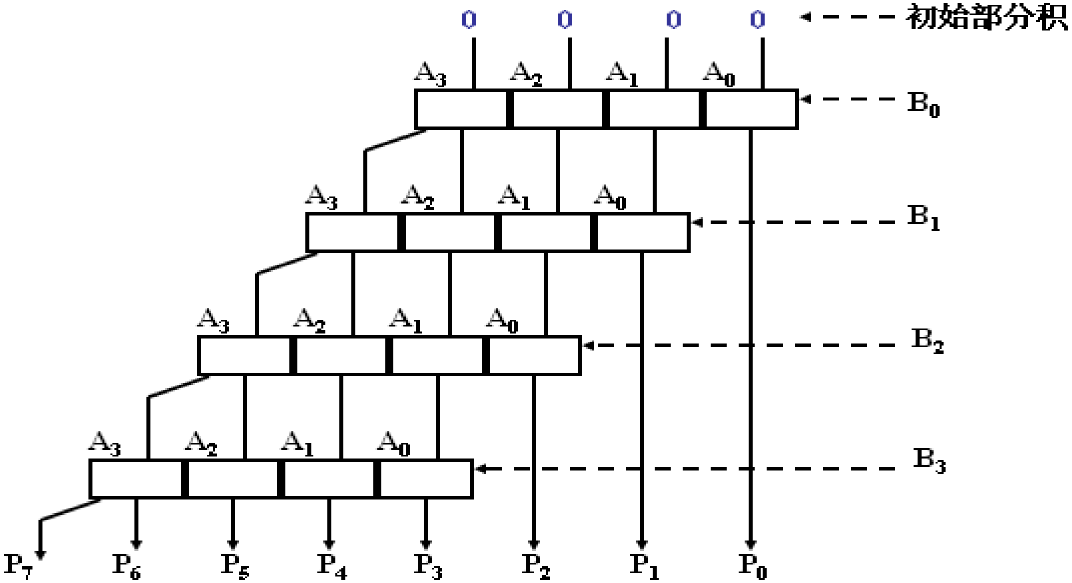
快速乘法器

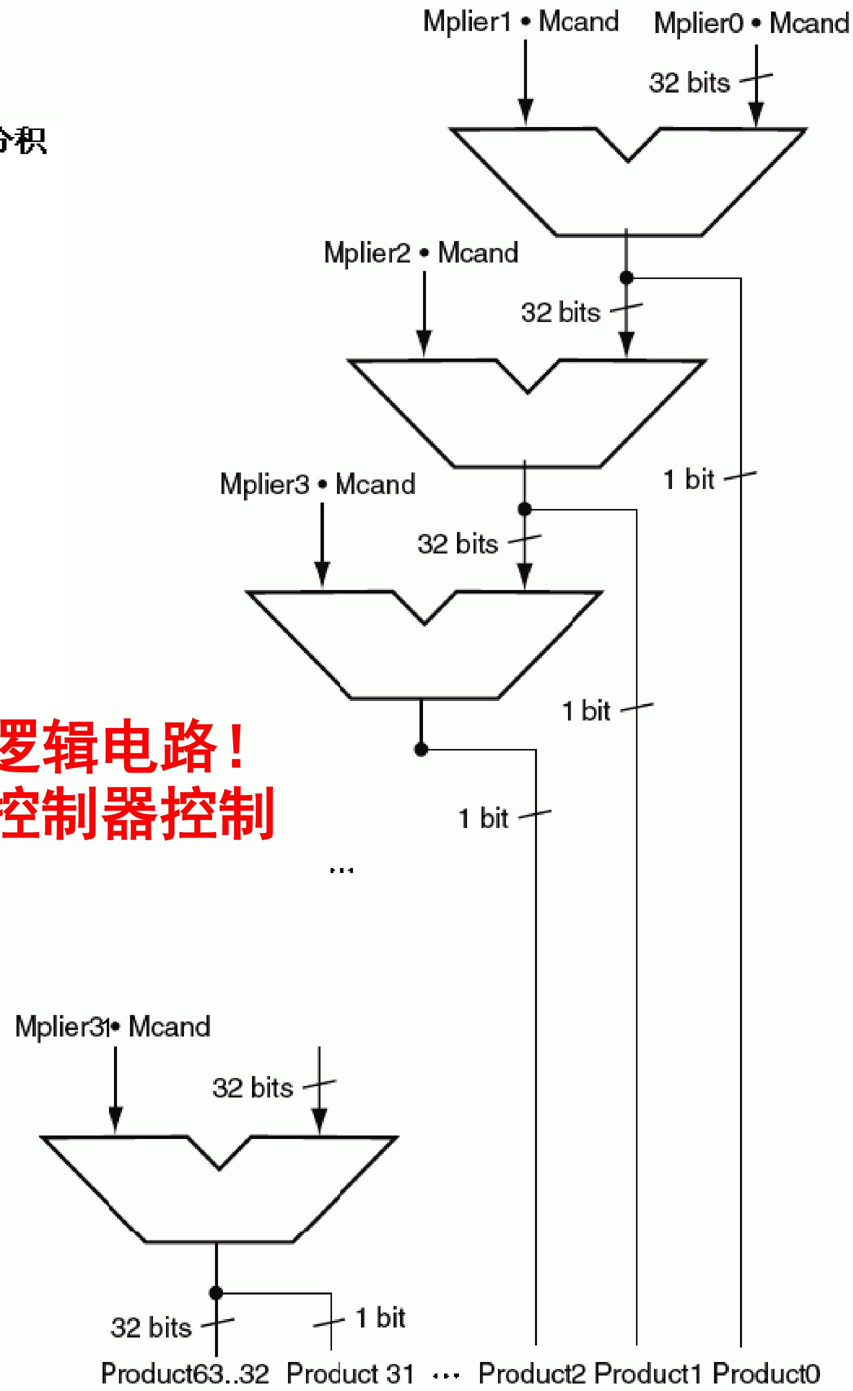
- 前面介绍的乘法部件的特点
  - 通过一个ALU多次做“加/减+右移”来实现
    - 一位乘法：约n次“加+右移”
    - 两位乘法：约n/2次“加+右移”

  所需时间随位数增多而加长，由时钟和控制电路控制
- 设计快速乘法部件的必要性
  - 大约1/3是乘法运算
- 快速乘法器的实现（由特定功能的组合逻辑单元构成）
  - 流水线方式/硬件叠加方式（如：阵列乘法器，附录6B）
    - 所有部分积并行相加，组织为二叉树结构，例如：16→8→4→2→结果
  - Booth算法(Andrew Donald Booth, 1950) + Wallace Tree(Chris Wallace, 1964)
    - 华莱士树（Wallace Tree）：硬件快速把n个数相加归约为2个数的相加，从而加速多个部分积相加速度【体系结构课程会继续介绍】

用“空间”换“时间”

流水线方式的快速乘法器



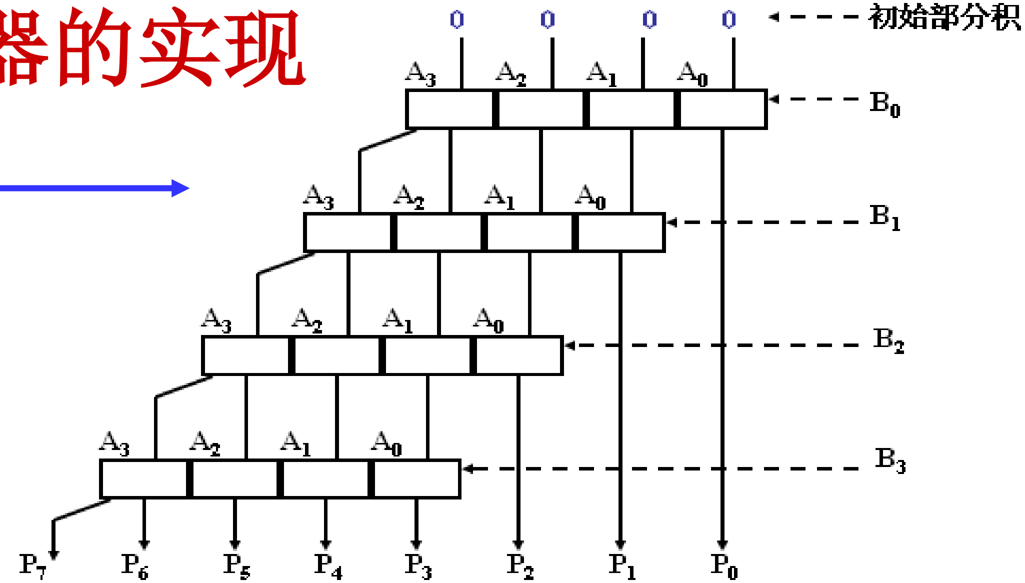


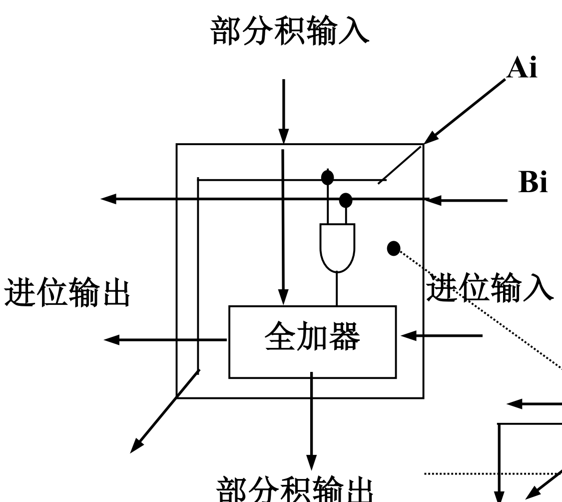
组合逻辑电路！
无需控制器控制

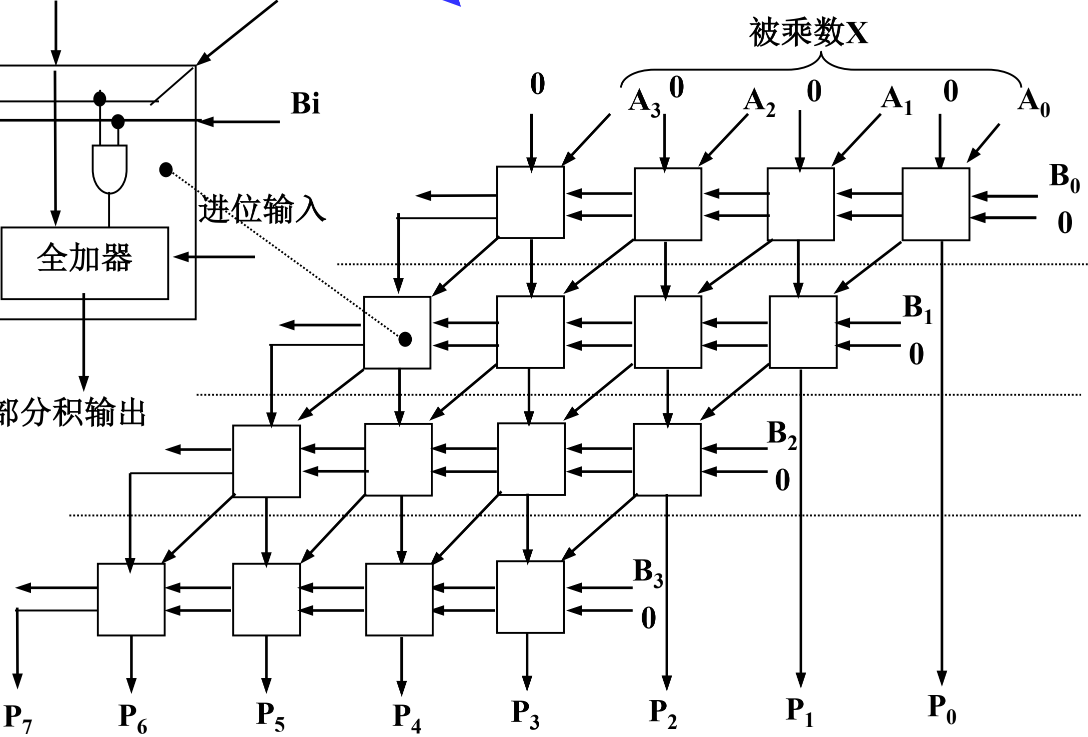
- 为乘数的每位提供一个n位加法器
- 每个加法器的两个输入端分别是：
  - 本次乘数对应的位与被乘数相与的结果（即：0或被乘数）
  - 上次部分积
- 每个加法器的输出分为两部分：
  - 和的最低有效位(LSB)作为本位乘积
  - 进位和高31位的和数组成一个32位数作为本次部分积

流水线式阵列乘法器的实现

- 手算乘法过程
- 阵列乘法器







速度仅取决于逻辑门和加法器的传输延迟

无符号阵列乘法器

增加符号处理电路、乘前及乘后求补电路，即可实现带符号数乘法器。

Google Tensor Processing Unit

(paper released on Apr 5, 2017)

In-Datacenter Performance Analysis of a Tensor Processing Unit

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon
Google, Inc., Mountain View, CA USA
jouppi@google.com

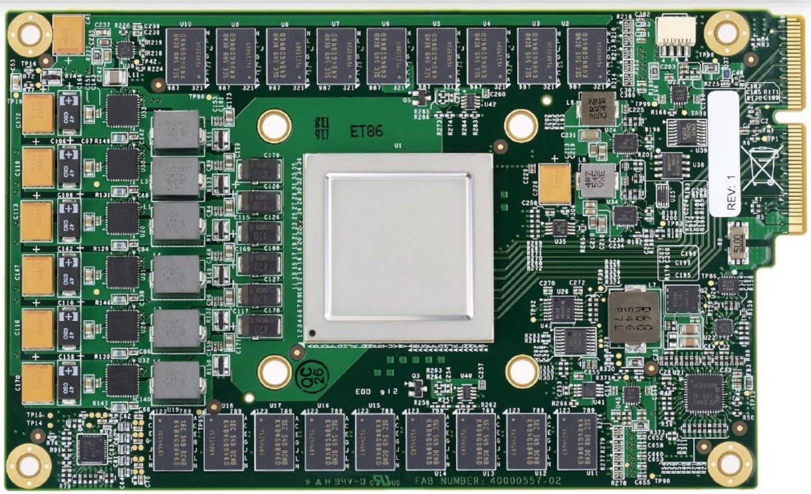


Figure 3. TPU Printed Circuit Board. It can be inserted in the slot for an SATA disk in a server, but the card uses PCIe Gen3 x16.

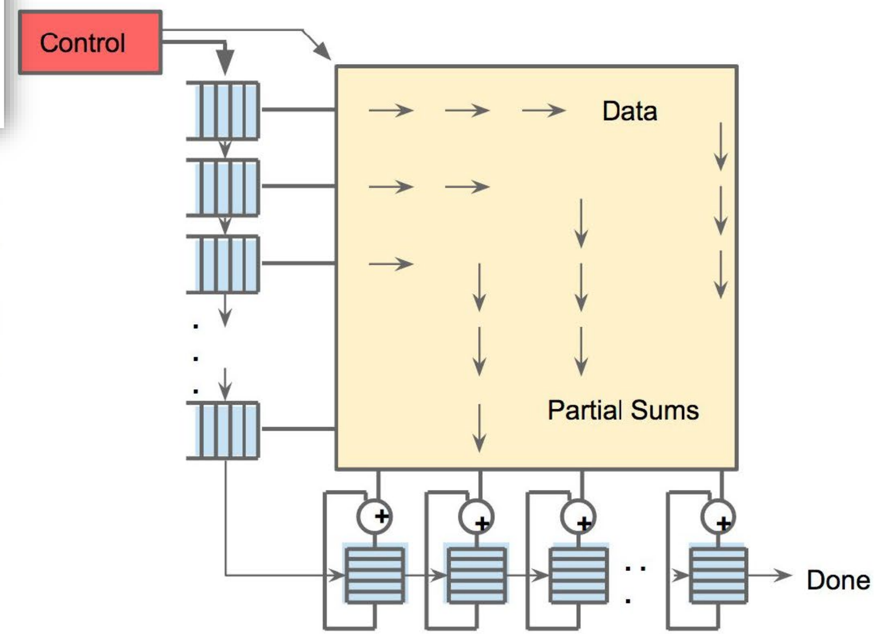


Figure 4. Systolic data flow of the Matrix Multiply Unit. Software has the illusion that each 256B input is read at once, and they instantly update one location of each of 256 accumulator RAMs.

Norman P. Jouppi et al., In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA 2017

乘法小结

- 整数乘法与小数乘法完全相同
  可用 逗号 代替小数点

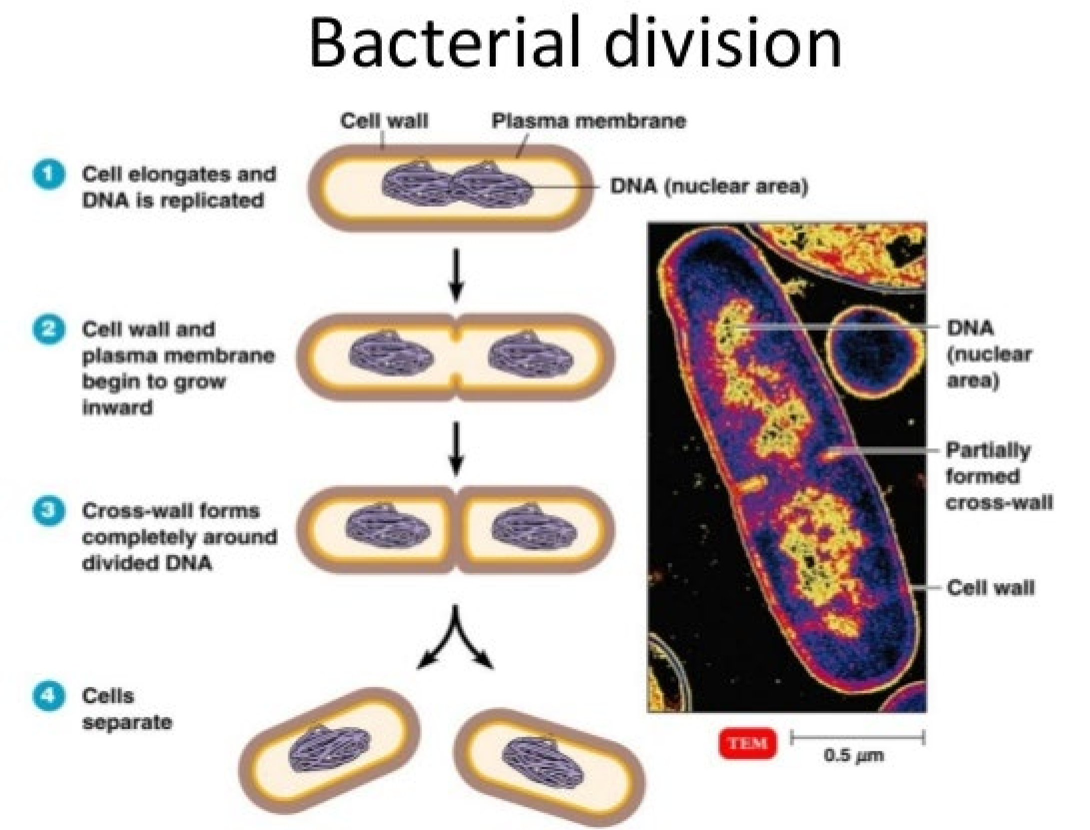
- 原码乘　符号位 单独处理
  补码乘　符号位 自然形成

- 原码乘去掉符号位运算　即为无符号数乘法

- 不同的乘法运算需有不同的硬件支持

Why are bacteria bad at math?

Because they multiply by dividing.



(a) A diagram of the sequence of cell division.

(b) A thin section of a cell of *Bacillus licheniformis* starting to divide.