
Ch3: Arithmetic and Logic

Operate and ALU

运算方法和运算部件

第一讲 不同层次程序员看到的运算及**ALU**

第二讲 定点数运算及其运算部件

第三讲 浮点数运算及其运算部件

第一讲：不同层次程序员看到的运算及ALU

主 要 内 容

- ◆ C语言程序中涉及的运算
 - 整数算术运、浮点数算术运算
 - 按位、逻辑、移位、位扩展和位截断
- ◆ 指令集中与运算相关的指令（以MIPS为参考）
 - 涉及到的定点数运算
 - 算术运算
 - 带符号整数运算：取负 / 符号扩展 / 加 / 减 / 乘 / 除 / 算术移位
 - 无符号整数运算：0扩展 / 加 / 减 / 乘 / 除
 - 逻辑运算
 - 逻辑操作：与 / 或 / 非 / ...
 - 移位操作：逻辑左移 / 逻辑右移
 - 涉及到的浮点数运算：加、减、乘、除
- ◆ 基本运算部件ALU的设计

C语言程序中涉及的运算

- ◆ 算术运算（最基本的运算）

- 无符号数、带符号整数、浮点数的运算

- ◆ 按位运算

- 用途

- 对一个位串实现“掩码”（**mask**）操作或相应的其他处理
（主要用于对多媒体数据或控制信息进行处理）

- 操作

- 按位或：“|”
 - 按位与：“&”
 - 按位取反：“~”
 - 按位异或：“^”

问题：如何从一个**16位**采样数据**y**中提取高位字节，并使低字节为**0**？

可用“&”实现“掩码”操作：**y & 0xFF00**

例如，当**y=0x2C0B**时，通过掩码操作得到结果为：**0x2C00**

C语言程序中涉及的运算

◆ 逻辑运算

• 用途

- 用于关系表达式的运算

例如，`if (x>y and i<100) then .....`中的“and”运算

• 操作

- “||”表示“OR”运算
- “&&”表示“AND”运算

例如，`if ((x>y) && (i<100)) then .....`

- “!”表示“NOT”运算

• 与按位运算的差别

- 符号表示不同：& ~ && ; | ~ || ;
- 运算过程不同：按位 ~ 整体
- 结果类型不同：位串 ~ 逻辑值

C语言程序中涉及的运算

◆ 移位运算

• 用途

- 提取部分信息
- 扩大或缩小数值的2、4、8...倍

• 操作

- 左移:: $x \ll k$; 右移: $x \gg k$
- 不区分是逻辑移位还是算术移位, 由x的类型确定
- 无符号数: 逻辑左移、逻辑右移

高(低)位移出, 低(高)位补0

问题: 何时可能发生溢出? 如何判断溢出?

若高位移出的是1, 则左移时发生溢出

- 带符号整数: 算术左移、算术右移

左移: 高位移出, 低位补0。

溢出判断: 若移出的位不等于新的符号位, 则溢出。

右移: 低位移出, 高位补符, 可能发生数据丢失。

C语言程序中涉及的运算

◆位扩展和位截断运算

• 用途

- 在进行类型转换时，可能需要数据的扩展或截断

• 操作

- 没有专门的操作运算符，根据类型转换前后数据长短来确定是扩展还是截断
- “扩展”：短数转为长数；“截断”，长数转为短数
- 扩展
 - 无符号数：0扩展，即：前面补0
 - 带符号整数：符号扩展，即：前面补符号
- 截断
 - 强行将一个长数的高位丢弃，故可能会发生“溢出”

例1：在大端机上输出si, usi, i, ui的十进制和十六进制值是什么？

```
short si = -12345;
unsigned short usi = si;
int i = si;
unsigned int ui = usi ;
```

```
si = -12345    CF C7
usi = 53191    CF C7
i = -12345     FF FF CF C7
ui = 53191     00 00 CF C7
```

例2：在大端机上执行后，i和j是否相等？

```
int i = 53191;
short si = (short)i;
int j = si;
```

不相等！

```
i = 53191  00 00 CF C7
si = -12345  CF C7
j = -12345  FF FF CF C7
```

原因：对i截断时发生了“溢出”，即：53191截断为16位数时，无法正确表示！

MIPS定点算术运算指令

<i>Instruction</i>	<i>Example</i>	<i>Meaning</i>	<i>Comments</i>
add	add \$1,\$2,\$3	$\$1 = \$2 + \$3$	3 operands; exception possible
subtract	sub \$1,\$2,\$3	$\$1 = \$2 - \$3$	3 operands; exception possible
add immediate	addi \$1,\$2,100	$\$1 = \$2 + 100$	+ constant; exception possible
add unsigned	addu \$1,\$2,\$3	$\$1 = \$2 + \$3$	3 operands; no exceptions
subtract unsigned	subu \$1,\$2,\$3	$\$1 = \$2 - \$3$	3 operands; no exceptions
add imm. unsign.	addiu \$1,\$2,100	$\$1 = \$2 + 100$	+ constant; no exceptions
multiply	mult \$2,\$3	Hi, Lo = $\$2 \times \3	64-bit signed product
multiply unsigned	multu \$2,\$3	Hi, Lo = $\$2 \times \3	64-bit unsigned product
divide	div \$2,\$3	Lo = $\$2 \div \3 , Hi = $\$2 \bmod \3	Lo = quotient, Hi = remainder
divide unsigned	divu \$2,\$3	Lo = $\$2 \div \3 , Hi = $\$2 \bmod \3	Unsigned quotient & remainder

涉及到的操作数：32/16位 无符号数， 32/16位带符号数

涉及到的操作：加 / 减 / 乘 / 除（有符号 / 无符号）

MIPS 逻辑运算指令

and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$
or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \mid \$s3$
nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim (\$s2 \mid \$s3)$
and immediate	andi \$s1,\$s2,100	$\$s1 = \$s2 \& 100$
or immediate	ori \$s1,\$s2,100	$\$s1 = \$s2 \mid 100$
shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$
shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$

涉及到的操作数：**32/16位 逻辑数**

涉及到的操作：**按位与 / 按位或 / 按位或非 / 左移 / 右移**

MIPS定点比较和分支指令

branch on equal	beq \$s1,\$s2,25	if (\$s1 == \$s2) go to PC + 4 + 100
branch on not equal	bne \$s1,\$s2,25	if (\$s1 != \$s2) go to PC + 4 + 100
set on less than	slt \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0
set less than immediate	slti \$s1,\$s2,100	if (\$s2 < 100) \$s1 = 1; else \$s1 = 0
set less than unsign	sltu \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0
set less than immediate unsigned	sltiu \$s1,\$s2,100	if (\$s2 < 100) \$s1 = 1; else \$s1 = 0

涉及到的操作数：**32/16**位 无符号数， **32/16**位带符号数

涉及到的操作：大小比较和相等比较（有符号 / 无符号）

通过减法运算实现“比较”操作！

MIPS定点数据传送指令

load word	lw \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$
store word	sw \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$
load half unsigned	lhu \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$
store half	sh \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$
load byte unsigned	lbu \$s1,100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$
store byte	sb \$s1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$
load upper immediate	lui \$s1,100	$\$s1 = 100 * 2^{16}$

涉及到的操作数： **32/16**位带符号数（偏移量可以是负数）

涉及到的操作： 加 / 减 / 符号扩展 / 0扩展

MIPS中的浮点算术运算指令

FP add single	<code>add.s \$f2,\$f4,\$f6</code>	$\$f2 = \$f4 + \$f6$
FP subtract single	<code>sub.s \$f2,\$f4,\$f6</code>	$\$f2 = \$f4 - \$f6$
FP multiply single	<code>mul.s \$f2,\$f4,\$f6</code>	$\$f2 = \$f4 \times \$f6$
FP divide single	<code>div.s \$f2,\$f4,\$f6</code>	$\$f2 = \$f4 / \$f6$
FP add double	<code>add.d \$f2,\$f4,\$f6</code>	$\$f2 = \$f4 + \$f6$
FP subtract double	<code>sub.d \$f2,\$f4,\$f6</code>	$\$f2 = \$f4 - \$f6$
FP multiply double	<code>mul.d \$f2,\$f4,\$f6</code>	$\$f2 = \$f4 \times \$f6$
FP divide double	<code>div.d \$f2,\$f4,\$f6</code>	$\$f2 = \$f4 / \$f6$

MIPS提供专门的浮点数寄存器：

- **32个32位单精度浮点数寄存器**：**\$f0, \$f1,, \$f31**
- 连续两个寄存器（一偶一奇）存放一个双精度浮点数

涉及到的浮点操作数：**32位单精度 / 64位双精度浮点数**

涉及到的浮点操作：**加 / 减 / 乘 / 除**

MIPS中的浮点数传送指令

load word copr. 1	lwcl \$f1, 100(\$s2)	\$f1 = Memory[\$s2 + 100]
store word copr. 1	swcl \$f1, 100(\$s2)	Memory[\$s2 + 100] = \$f1

涉及到的浮点操作数： **32**位单精度浮点数

涉及到的浮点操作： 传送操作（与定点传送一样）

还涉及到定点操作： 加 / 减（用于地址运算）

例： 实现将两个浮点数从内存取出相加后再存回到内存的指令序列为：

```
lwcl  $f1, x($s1)
lwcl  $f2, y($s2)
add.s $f4, $f1, $f2
swlc  $f4, z($s3)
```

MIPS中的浮点数比较和分支指令

branch on FP true	bclt 25	if (cond == 1) go to PC + 4 + 100
branch on FP false	bclf 25	if (cond == 0) go to PC + 4 + 100
FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s \$f2,\$f4	if (\$f2 < \$f4) cond = 1; else cond = 0
FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d \$f2,\$f4	if (\$f2 < \$f4) cond = 1; else cond = 0

涉及到的浮点操作数： **32**位单精度浮点数 / **64**位双精度浮点数

涉及到的浮点操作： 比较操作（用 减法来实现比较）

还涉及到的定点操作： 加 / 减（用于地址运算）

有一个专门的浮点标志**cond**，无需在指令中明显给出**cond**

MIPS指令考察的结果

◆ 涉及到的操作数:

- 无符号整数、带符号整数
- 逻辑数
- 浮点数

◆ 涉及到的运算

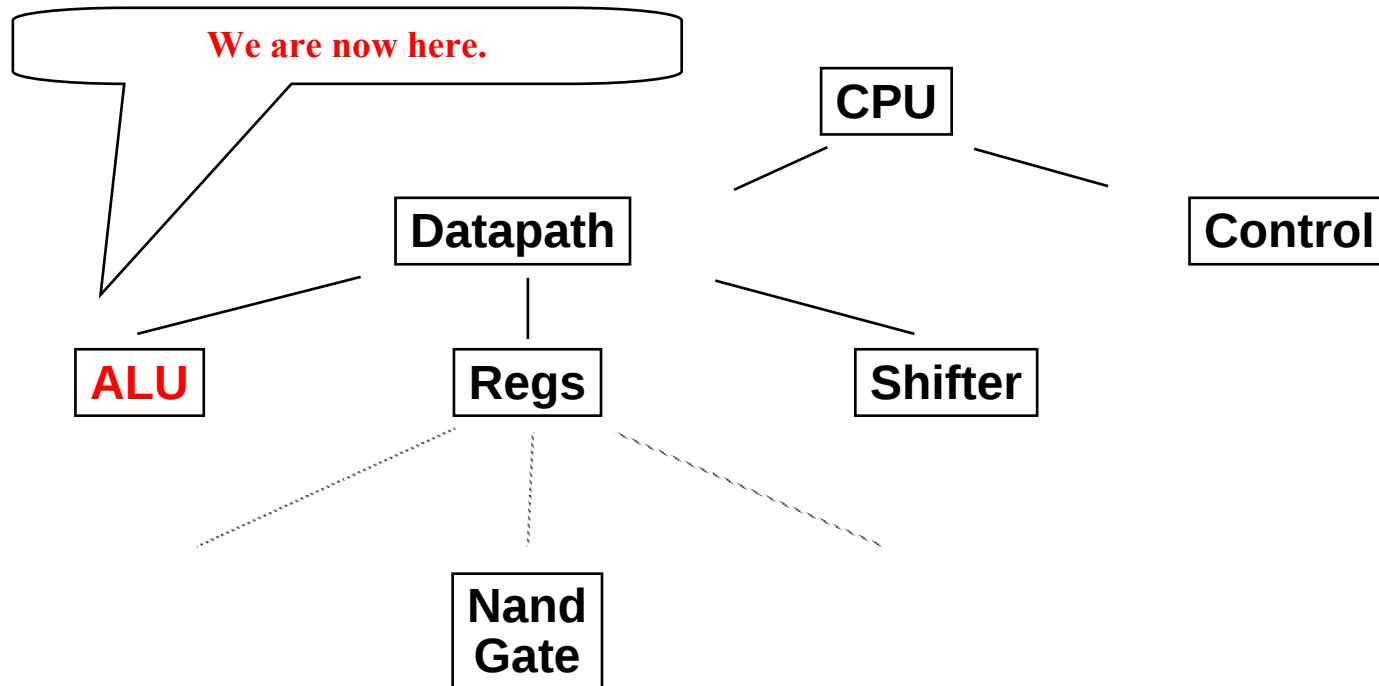
- 定点数运算
 - 带符号整数运算: 取负 / 符号扩展 / 加 / 减 / 乘 / 除 / 算术移位
 - 无符号整数运算: 0扩展 / 加 / 减 / 乘 / 除
- 逻辑运算
 - 逻辑操作: 与 / 或 / 非 / ...
 - 移位操作: 逻辑左移 / 逻辑右移
- 浮点数运算: 加、减、乘、除

实现MIPS定点运算指令的思路:

首先实现一个能进行基本算术运算（加/减）和基本逻辑运算（与/或/或非）、并能生成基本条件码（ZF/VF/CF/NF）的ALU，再由ALU和移位器实现乘除运算器。

ALU是运算部件的核心！ 以下介绍ALU的实现。

ALU的位置



ALU (Arithmetic logic unit) : 算术逻辑部件

功能：能进行基本算术、逻辑运算，并生成条件码

硬件功能描述 (Design as Representation)

(1) Functional Specification(功能说明)

"VHDL Behavior"

Inputs: 2 x 16 bit operands—A, B; 1 bit carry input—Cin.

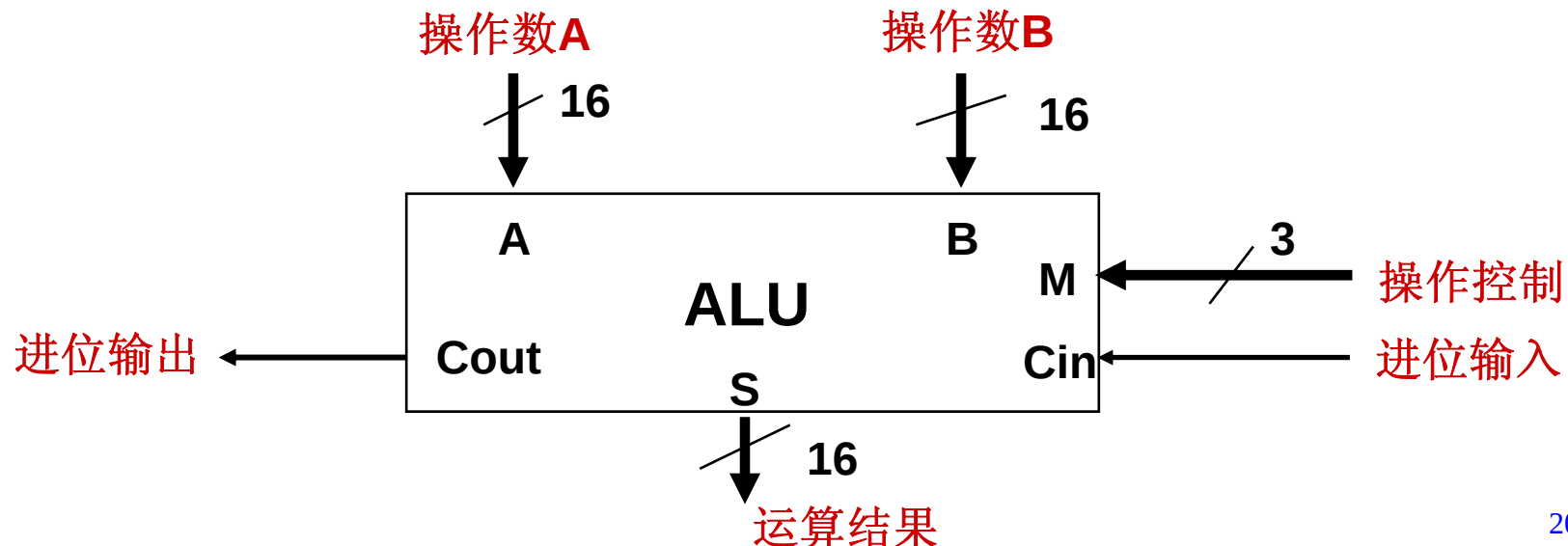
Outputs: 1 x 16 bit result—S; 1 bit carry output—Cout.

Operations: PASS, ADD ($A + B + C_{in}$), SUB ($A - B$),
AND, XOR, OR, COMPARE (equality)

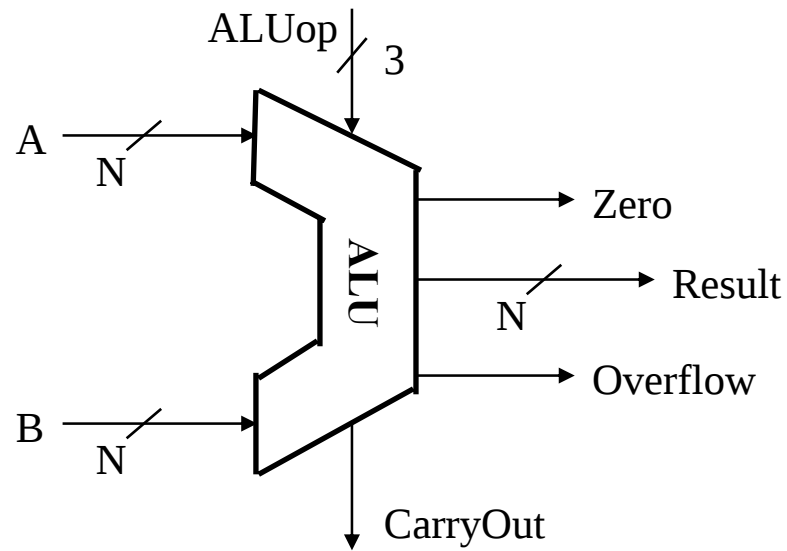
VHDL (Very-High-Speed Integrated Circuit Hardware Description Language) 1987 年底IEEE和美国国防部确定其为标准硬件描述语言

(2) Block Diagram/Schematic (框图/原理图表示)

"VHDL Entity"



ALU的功能说明



◆ ALU Control Lines (ALUOp)

- 000
- 001
- 010
- 110
- 111

Function

And

Or

Add

Subtract

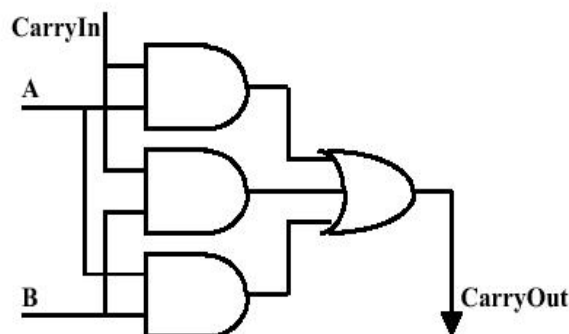
Set-on-less-than

ALU可进行基本的加/减算术运算、基本逻辑运算。其核心部件是加法器。
有关串行加法器和并行加法器的原理在数字逻辑电路课已讲过，在此仅简单回顾。

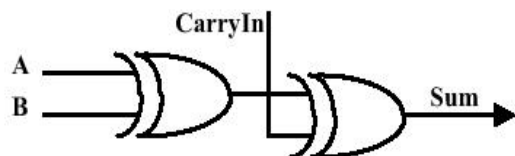
回顾：串行进位加法器

CarryOut 和 Sum 的逻辑图

° $\text{CarryOut} = B \& \text{CarryIn} \mid A \& \text{CarryIn} \mid A \& B$



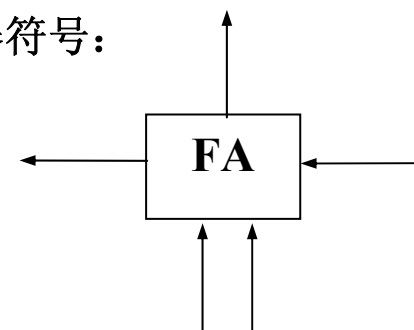
° $\text{Sum} = A \text{ XOR } B \text{ XOR } \text{CarryIn}$



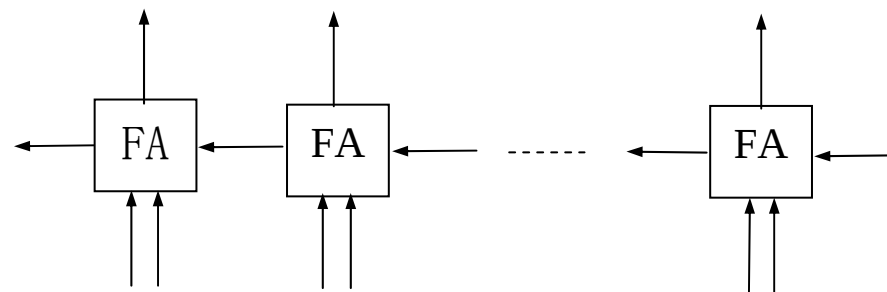
Sum延迟为6ty；进位Carryout延迟为 2 ty

(假定一个与门/或门延迟为1ty，异或门的延迟则为 3 ty)

全加器符号：



n位串行(行波)加法器：



串行加法器的缺点：

进位按串行方式传递，速度慢！

问题：n位串行加法器从C0到Cn的延迟时间为多少？ 2n级门延迟！

最后一位和数的延迟时间为多少？

2n+1级门延迟！

回顾：并行进位加法器

- 为什么用先行进位方式？

串行（行波）进位加法器采用串行逐级传递进位，电路的延迟与位数成正比关系。因此，现代计算机采用一种先行进位(Carry look ahead)方式。

- 如何产生先行进位？

定义两个辅助函数： $G_i = X_i Y_i \dots$ 进位生成 $P_i = X_i + Y_i \dots$ 进位传递（或 $P_i = X_i \oplus Y_i$ ）

通常把实现上述逻辑的电路称为进位生成/传递部件

全加逻辑方程： $S_i = P_i \oplus C_i$ $C_{i+1} = G_i + P_i C_i$ ($i=0,1,\dots,n$)

设 $n=4$,则： $C_1 = G_0 + P_0 C_0$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

$$C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$$

由上式可知:各进位之间无等待，相互独立并同时产生。

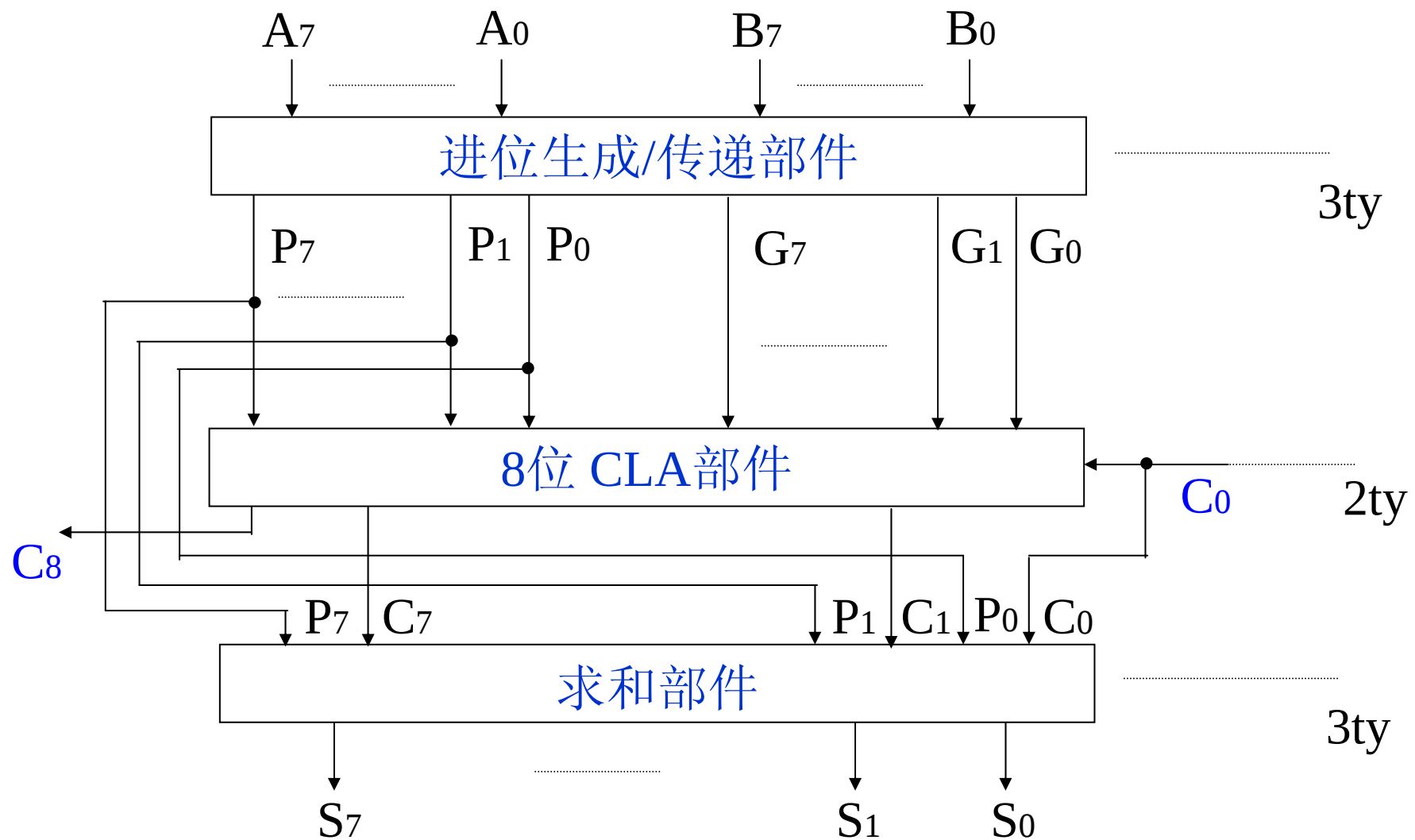
通常把实现上述逻辑的电路称为4位CLA部件

由此，根据 $S_i = P_i \oplus C_i$ ，可并行求出各位和。

通常把实现 $S_i = P_i \oplus C_i$ 的电路称为求和部件

CLA加法器由“进位生成/传递部件”、“CLA部件”和“求和部件”构成。

回顾： 8位全先行进位加法器



和的总延迟： $3+2+3=8ty$ ； 进位 C_8 的延迟： $3+2=5ty$

回顾：局部先行进位加法器

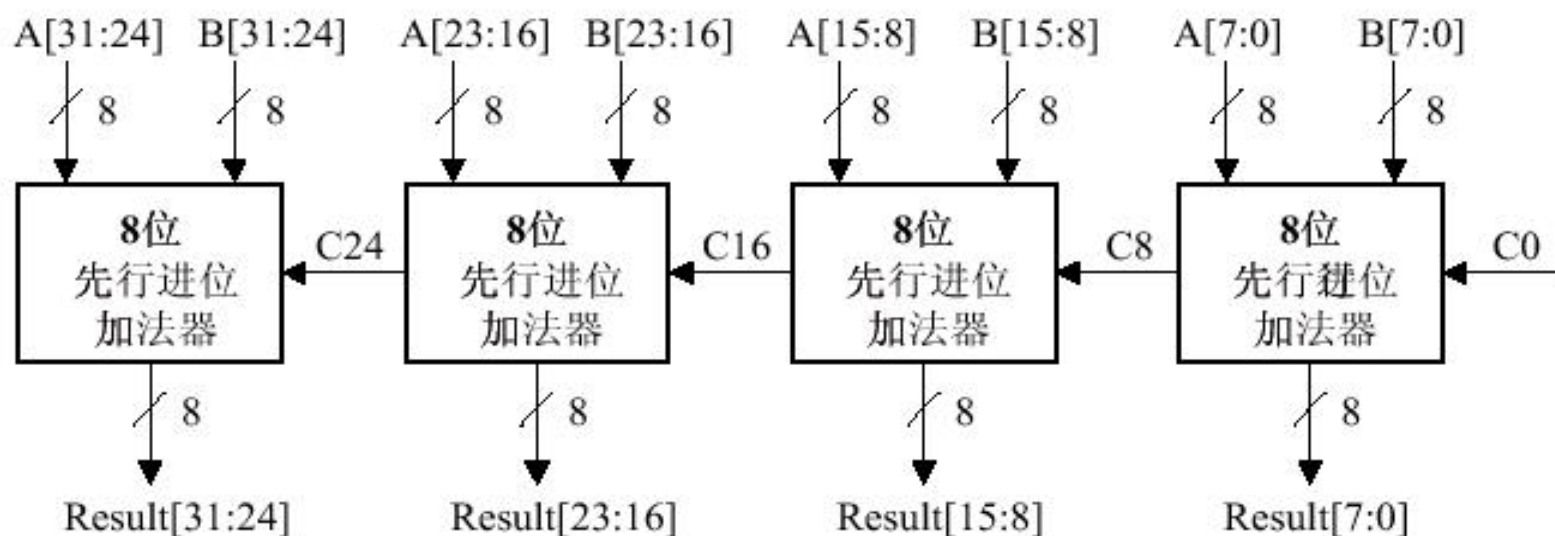
局部先行进位加法器 (Partial Carry Lookahead Adder)

或称 单级先行进位加法器

- 实现全先行进位加法器的成本太高
 - 想象 **Cin31** 的逻辑方程的长度
- 一般性经验：
 - 连接一些 **N** 位先行进位加法器，形成一个大加法器
 - 例如：连接 **4** 个 **8** 位进位先行加法器，形成 **1** 个 **32** 位局部先行进位加法器

问题：所有和数产生的延迟为多少？

$$5+2+2+5=14t_y$$



回顾：多级先行进位加法器

(3) 多级先行进位加法器

- 单级(局部)先行进位加法器的进位生成方式：
“组内并行、组间串行”
- 所以，单级先行进位加法器虽然比行波加法器延迟时间短，但高位组进位依赖低位组进位，故仍有较长的时间延迟
- 通过引入组进位生成/传递函数来实现“组内并行、组间并行”进位方式

设 $n=4$,则: $C_1=G_0+P_0C_0$

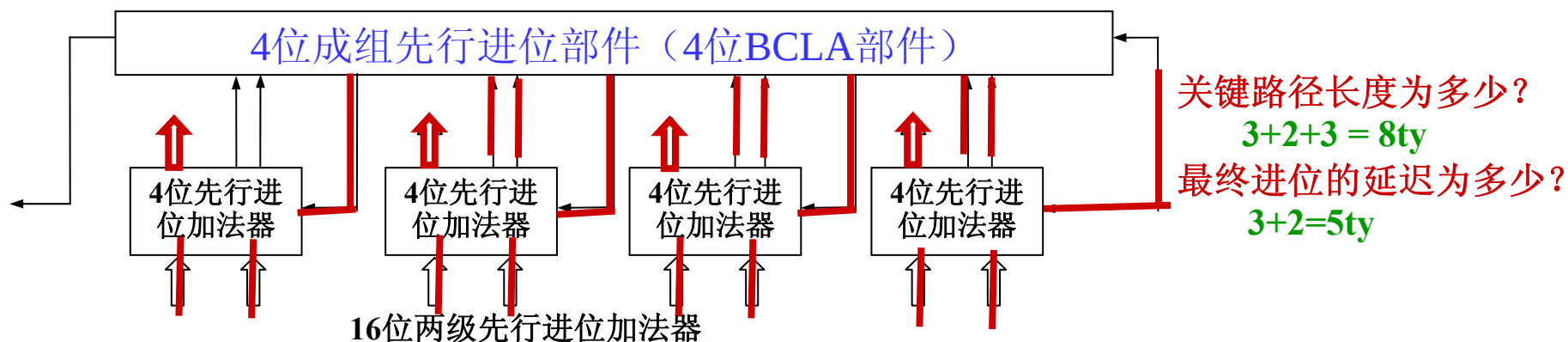
$$C_2=G_1+P_1C_1=G_1+P_1G_0+P_1P_0C_0$$

$$C_3=G_2+P_2C_2=G_2+P_2G_1+P_2P_1G_0+P_2P_1P_0C_0$$

$$G_3^*=G_3+P_3C_3=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0$$

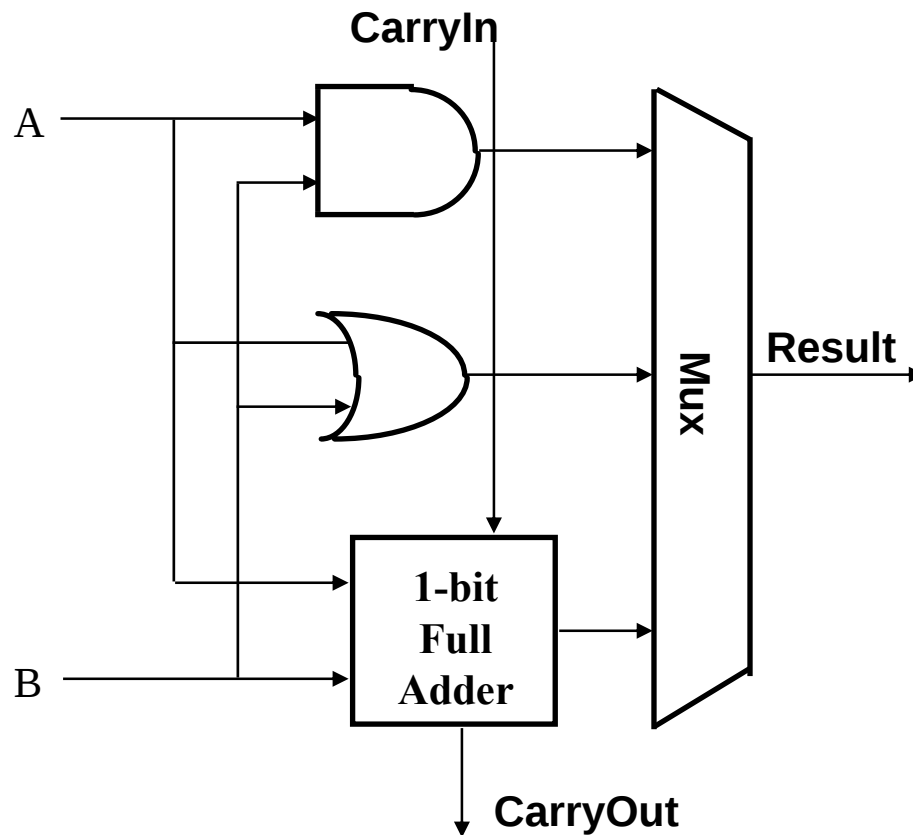
$$P_3^*=P_3P_2P_1P_0$$

所以 $C_4 = G_3^* + P_3^*C_0$ 。把实现上述逻辑的电路称为**4位BCLA部件**。



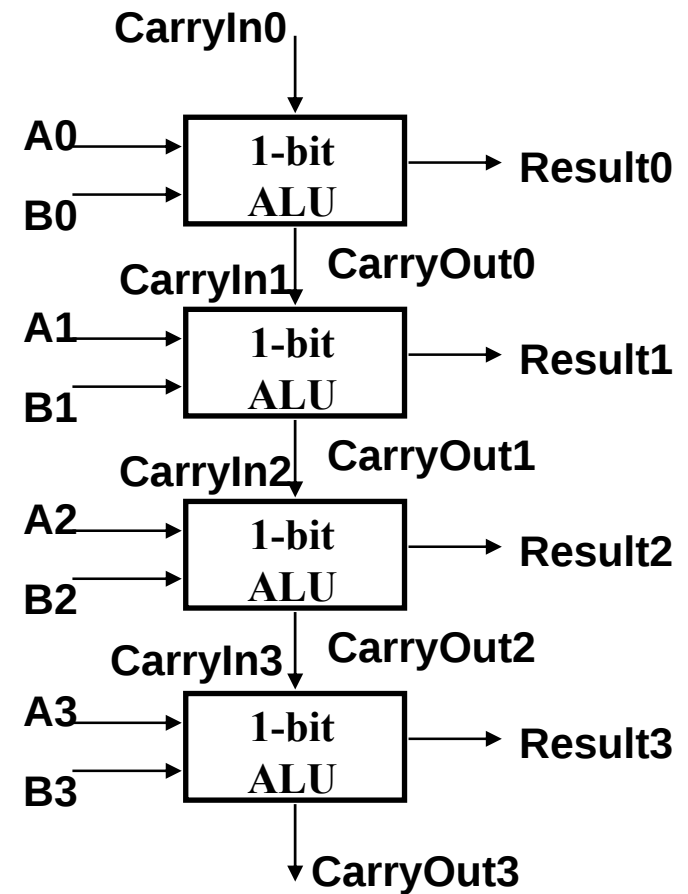
A 4-bit ALU

1-bit ALU



MUX是什么？（数字电路课学过）

4-bit串行 ALU



关键路径延迟长，速度慢！

先行进位ALU

◆ 先行进位ALU 芯片（SN74181）

- 四位ALU芯片，中规模集成电路。在先行进位加法器基础上附加部分线路，具有基本的算术运算和逻辑运算功能。
- SN74181的逻辑图和功能表
- SN74182是4位BCLA (成组先行进位)芯片。

◆ 多芯片级联构成先行进位ALU

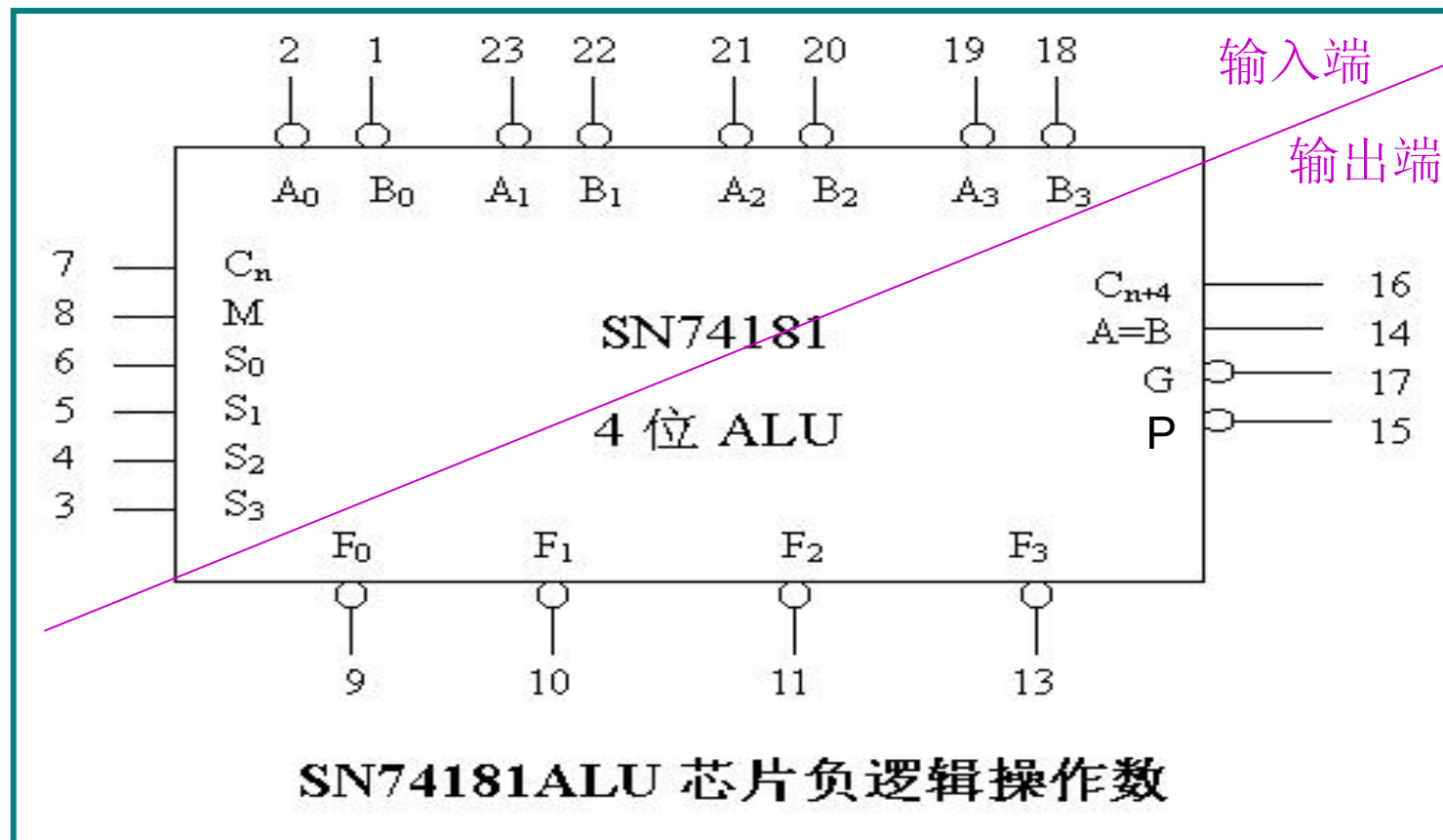
- 1个SN74181芯片直接构成一个4位全先行进位ALU
- 4个SN74181芯片串行构成一个16位单级先行进位ALU
- 4个SN74181芯片与1个SN74182芯片可构成16位两级先行进位ALU
- 16个SN74181芯片与5个SN74182芯片可构成64位先行进位ALU

ALU中的“加”运算电路相当于n档二进制加法算盘。

所有其他运算都以ALU 中“加”运算为基础！

SKIP

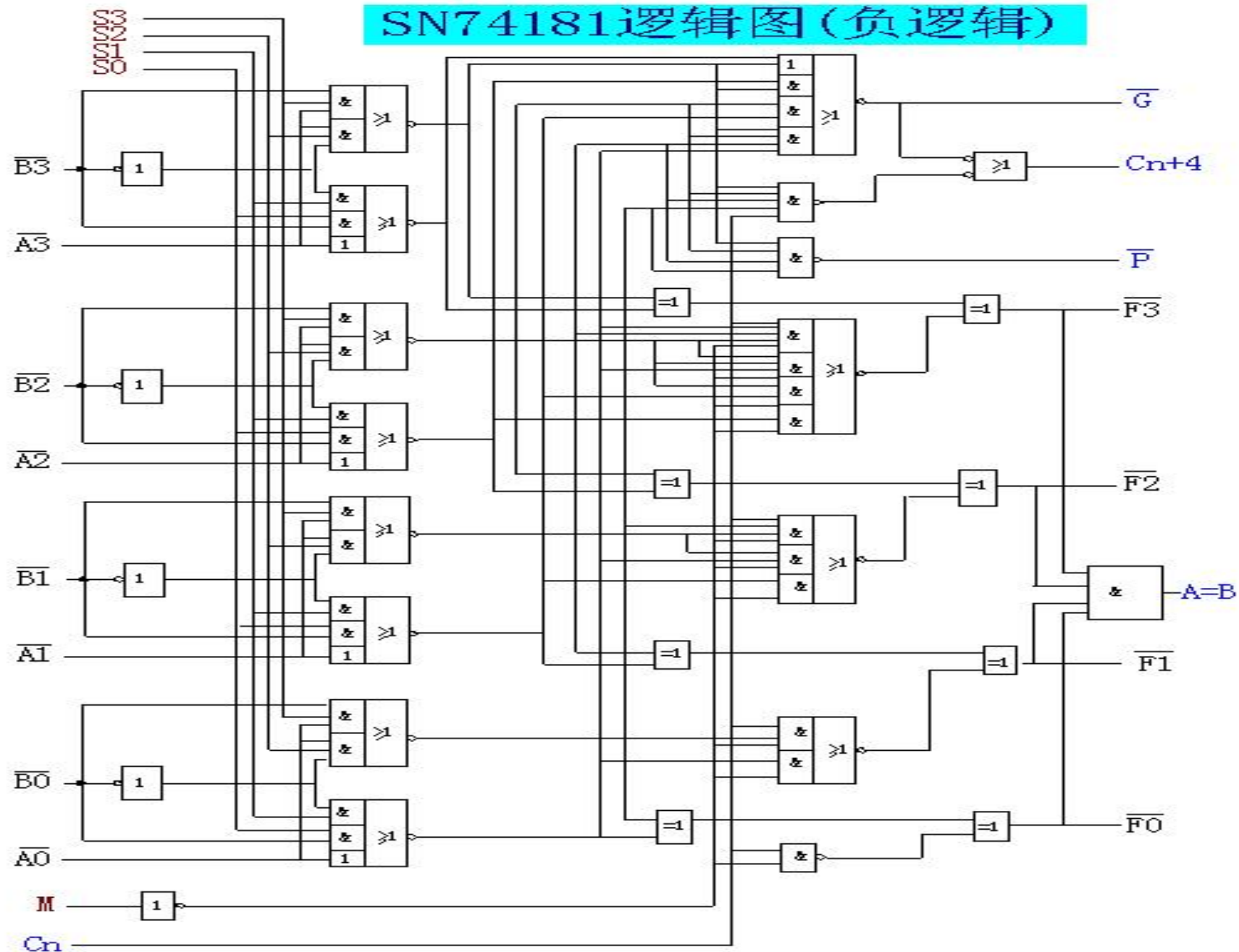
SN74181的引脚



输入端： A_i 和 B_i 分别为第1和2操作数， C_n 为低位进位， M 为功能选择线， S_i 为操作选择线，共4位，故最多有16种运算。

输出端： F_i 为运算结果， C_{n+4} 、 P 和 G 为进位，“ $A=B$ ”为相等标志

SN74181逻辑图(负逻辑)

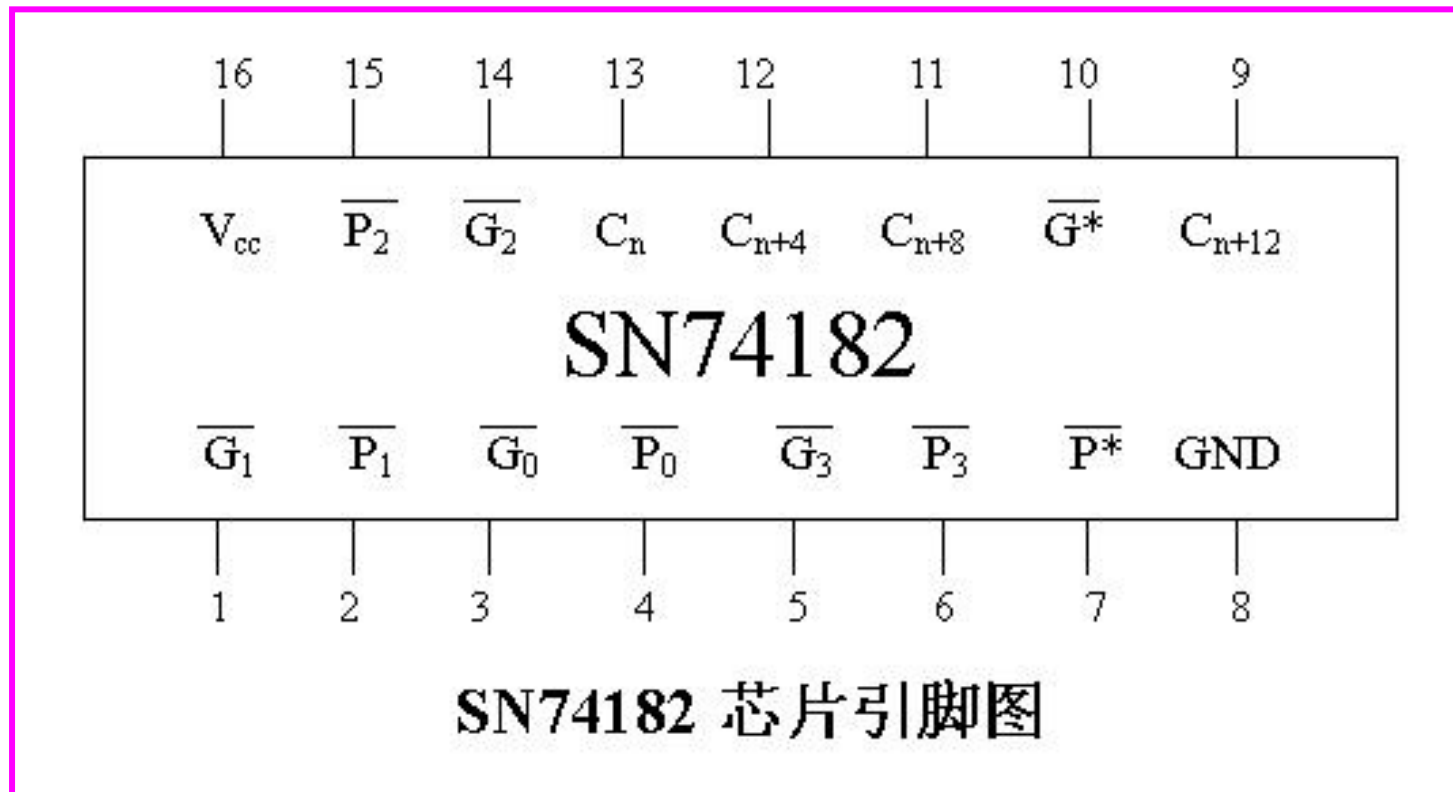


SN74181正逻辑功能表

S_3	S_2	S_1	S_0	M=H 逻辑运算	M=L 算术运算	
					$\overline{C_n}=1$	$\overline{C_n}=0$
L	L	L	L	$\overline{A}$	A	A+1
L	L	L	H	$\overline{A+B}$	A+B	(A+B)加 1
L	L	H	L	$\overline{A} \cdot B$	A+B	(A+B)加 1
L	L	H	H	“0”	减 1	“0”
L	H	L	L	$\overline{A \cdot B}$	A 加(A· $\overline{B}$)	A 加(A· $\overline{B}$) 加 1
L	H	L	H	$\overline{B}$	(A· $\overline{B}$)加(A+B)	(A· $\overline{B}$)加(A+B)加 1
L	H	H	L	$A \oplus B$	A 减 B 减 1	A 减 B
L	H	H	H	$A \cdot B$	(A· $\overline{B}$)减 1	A· $\overline{B}$
H	L	L	L	$\overline{A+B}$	A 加(A·B)	A 加(A·B) 加 1
H	L	L	H	$\overline{A \oplus B}$	A 加 B	A 加 B 加 1
H	L	H	L	B	(A·B)加(A+B)	(A·B)加(A+B) 加 1
H	L	H	H	$A \cdot B$	(A·B)减 1	A·B
H	H	L	L	“1”	A 加 A	A 加 A 加 1
H	H	L	H	$A+B$	A 加(A+B)	A 加(A+B) 加 1
H	H	H	L	A+B	A 加(A+B)	A 加(A+B) 加 1
H	H	H	H	A	A 减 1	A

[BACK](#)

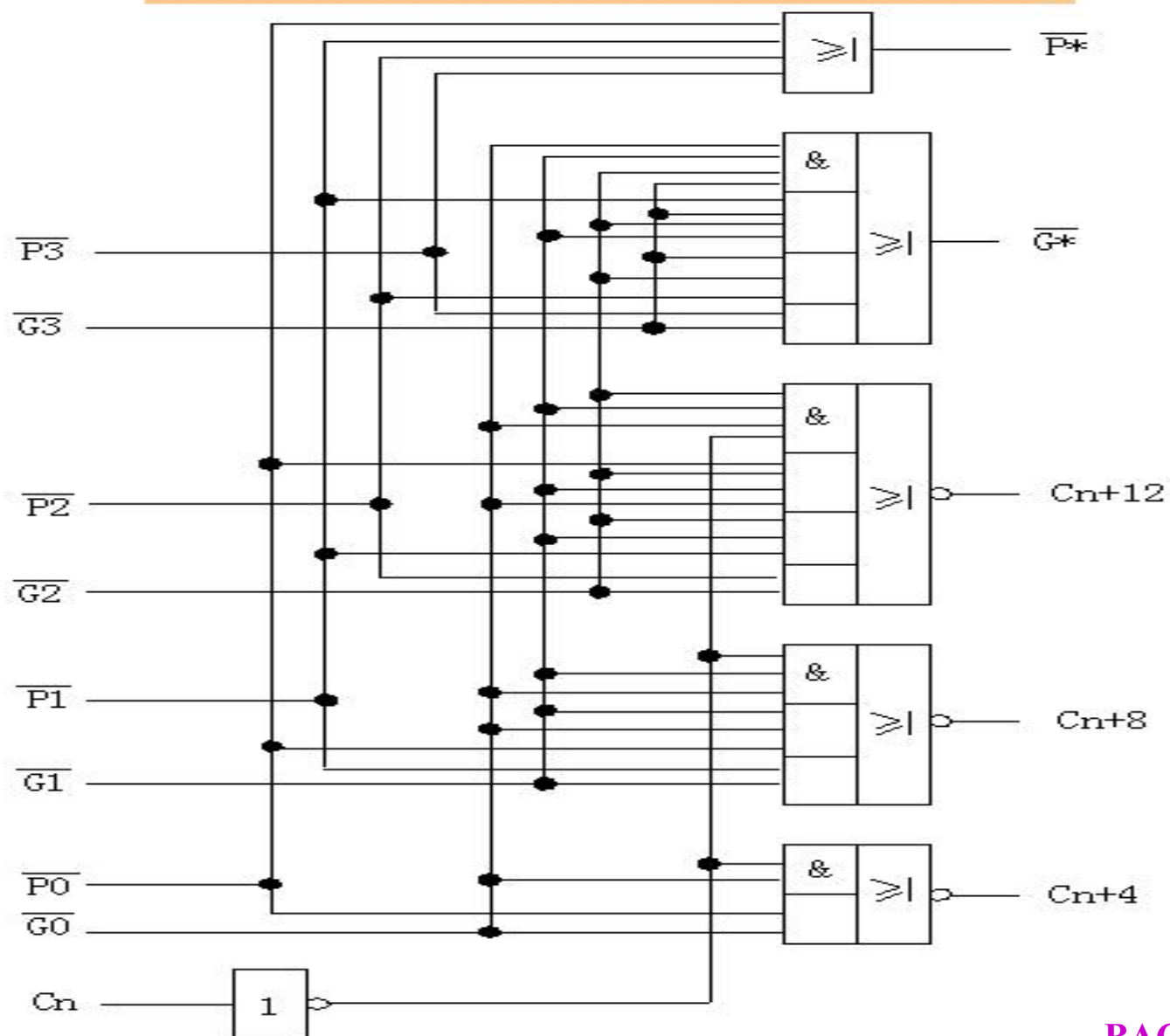
SN74182芯片的引脚



输入端： P_i 和 G_i 分别为第 i 组的组内进位传递函数和进位生成函数， C_n 为低位进位。

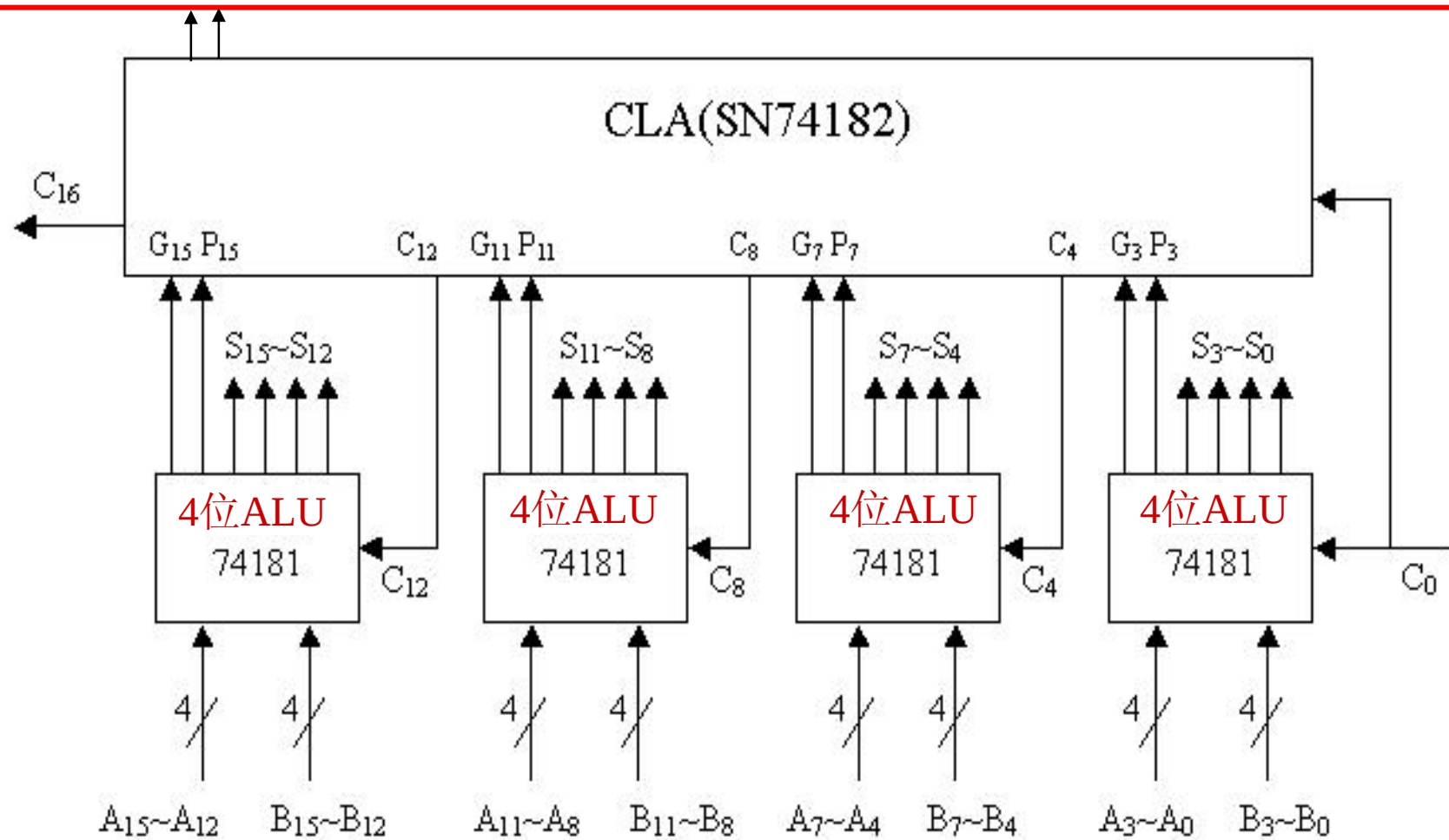
输出端： C_{n+4} 、 C_{n+8} 、 C_{n+12} 为相应组的组内进位， P^* 和 G^* 分别为整个大组的组进位传递函数和进位生成函数。

SN74182快速进位扩展器逻辑图



[BACK](#)

SN74181和SN74182组成16位先行进位ALU



16位两级先行进位ALU

[BACK](#)

第一讲小结

◆ C语言程序中涉及的运算

⌚ 整数算术运算、浮点数算术运算

⌚ 按位、逻辑、移位、位扩展和位截断

◆ 指令集中与运算相关的指令（以MIPS为参考）

• 涉及到的定点数运算

- 算术运算

- 带符号整数运算：取负 / 符号扩展 / 加 / 减 / 乘 / 除 / 算术移位
- 无符号整数运算：0扩展 / 加 / 减 / 乘 / 除

- 逻辑运算

- 逻辑操作：与 / 或 / 非 / ...
- 移位操作：逻辑左移 / 逻辑右移

• 涉及到的浮点数运算：加、减、乘、除

◆ 基本运算部件ALU的设计

- 全加器、串行加法器、先行进位加法器
- 串行ALU、先行进位ALU（单级/多级）

定点运算包括：

- 无符号数
 - 按位逻辑运算
 - 逻辑移位运算
 - 位扩展和截断运算
 - 加/减/乘/除运算
- 带符号整数
 - 算术移位运算
 - 扩展运算和截断运算
 - 补码加/减/乘/除运算

浮点运算包括：

- 原码加/减/乘/除运算
- 移码加/减运算

切记：

ALU中的“加”运算电路相当于n档二进制加法算盘。所有其他运算都以ALU“加”运算为基础！

是模运算系统！

第二讲：定点数运算及运算部件

主 要 内 容

- ◆ 加/减运算及其运算部件
 - 补码 / 原码 / 移码 加减运算
- ◆ 乘法运算及其运算部件
 - 原码 / 补码 乘法运算
 - 快速乘法器
- ◆ 除法运算及其运算部件
 - 原码 / 补码 除法运算
 - 快速除法器
- ◆ 定点运算器
- ◆ 十进制加减运算

注：无符号数的按位逻辑运算可用逻辑门电路实现；无符号数的逻辑移位运算可用专门的移位器或斜送结果等多种方式来实现；带符号数的移位运算、无符号数和带符号整数的位扩展运算和截断运算也可用简单电路很容易地实现。

补码加/减运算及其部件

◆ 补码加减运算公式

- $[X+Y]_{\text{补}} = [X]_{\text{补}} + [Y]_{\text{补}} \pmod{2^n}$
- $[X-Y]_{\text{补}} = [X]_{\text{补}} + [-Y]_{\text{补}} \pmod{2^n}$

问题：补码加减运算的用途是什么？

用于实现带符号整数的加减运算！

◆ 补码加减运算要点和运算部件

- 加、减法运算统一采用加法来处理
- 符号位(最高有效位**MSB**)和数值位一起参与运算
- 直接用**ALU**实现两个数的加运算（模运算系统）

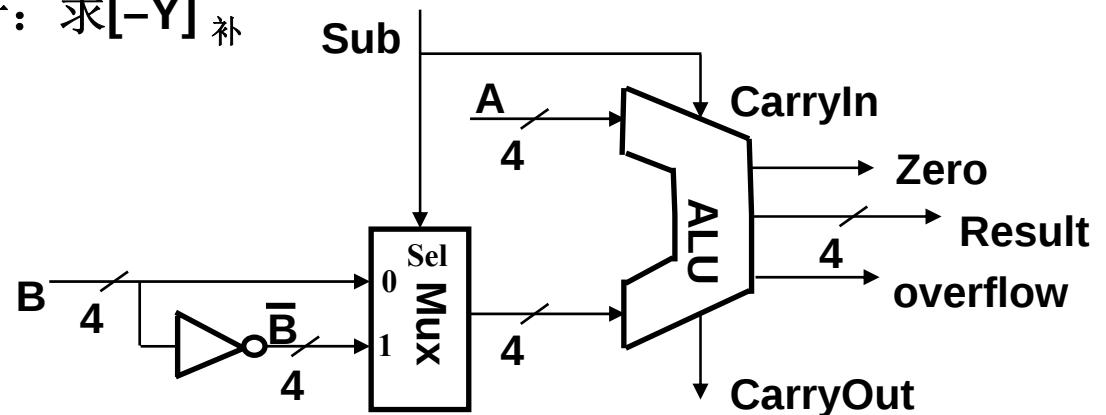
问题：模是多少？

- 运算结果的高位丢弃，保留低 n 位，相当于对和数取模 2^n
- 实现减法的主要工作在于：求 $[-Y]_{\text{补}}$

问题：如何求 $[-Y]_{\text{补}}$ ？

$$[-B]_{\text{补}} = \overline{B} + 1$$

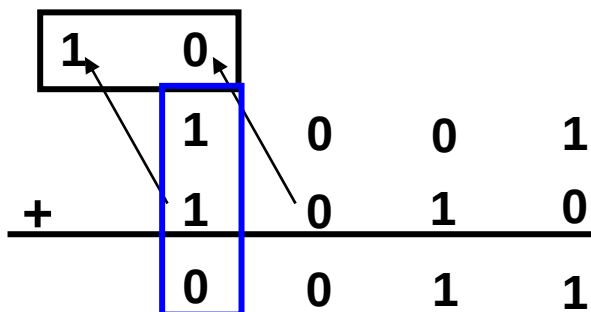
当控制端Sub为1时，做减法，实现 $A-B$
当控制端Sub为0时，做加法，实现 $A+B$



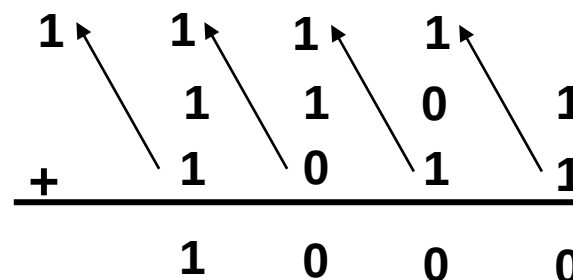
补码加减运算部件

补码加/减运算与“溢出”判断

Example1: $-7 - 6 = -7 + (-6) = +3$ **X**



$-3 - 5 = -3 + (-5) = -8$ **✓**



- 溢出现象:
- (1) 最高位和次高位的进位不同
 - (2) 和的符号位和加数的符号位不同

Example2: 用8位补码求 107 和 46的“和”

结果错误: $107 + 46 = -103$.

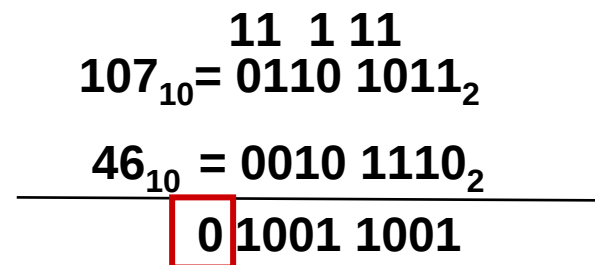
有两种“溢出”判断规则:

1. 和的符号位和加数的符号位不同
2. 最高位和次高位的进位不同

采用变形补码时的“溢出”判断条件:

结果的两个符号位不同

采用变形补码时, 可保留运算中间结果。
从乘除运算过程可看出这点!



溢出时, 符号位的进位是真正的符号: **+153**

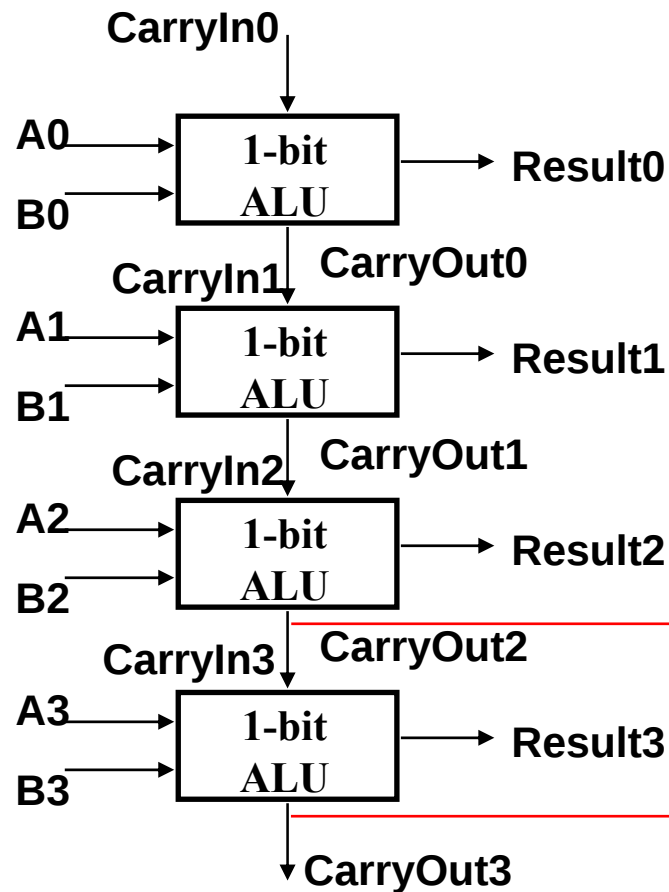
问题: 若采用变形补码则结果怎样? 有何好处?

结果的值为01 0011001, 左边第一位为真正的符号, 数值部分进到了右边符号位上。

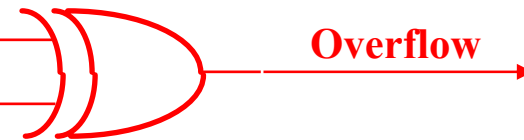
Overflow Detection Logic(溢出判断逻辑)

◆ Carry into MSB $\neq$ Carry out of MSB

- For a N-bit ALU: $\text{Overflow} = \text{CarryIn}[N-1] \text{ XOR } \text{CarryOut}[N-1]$

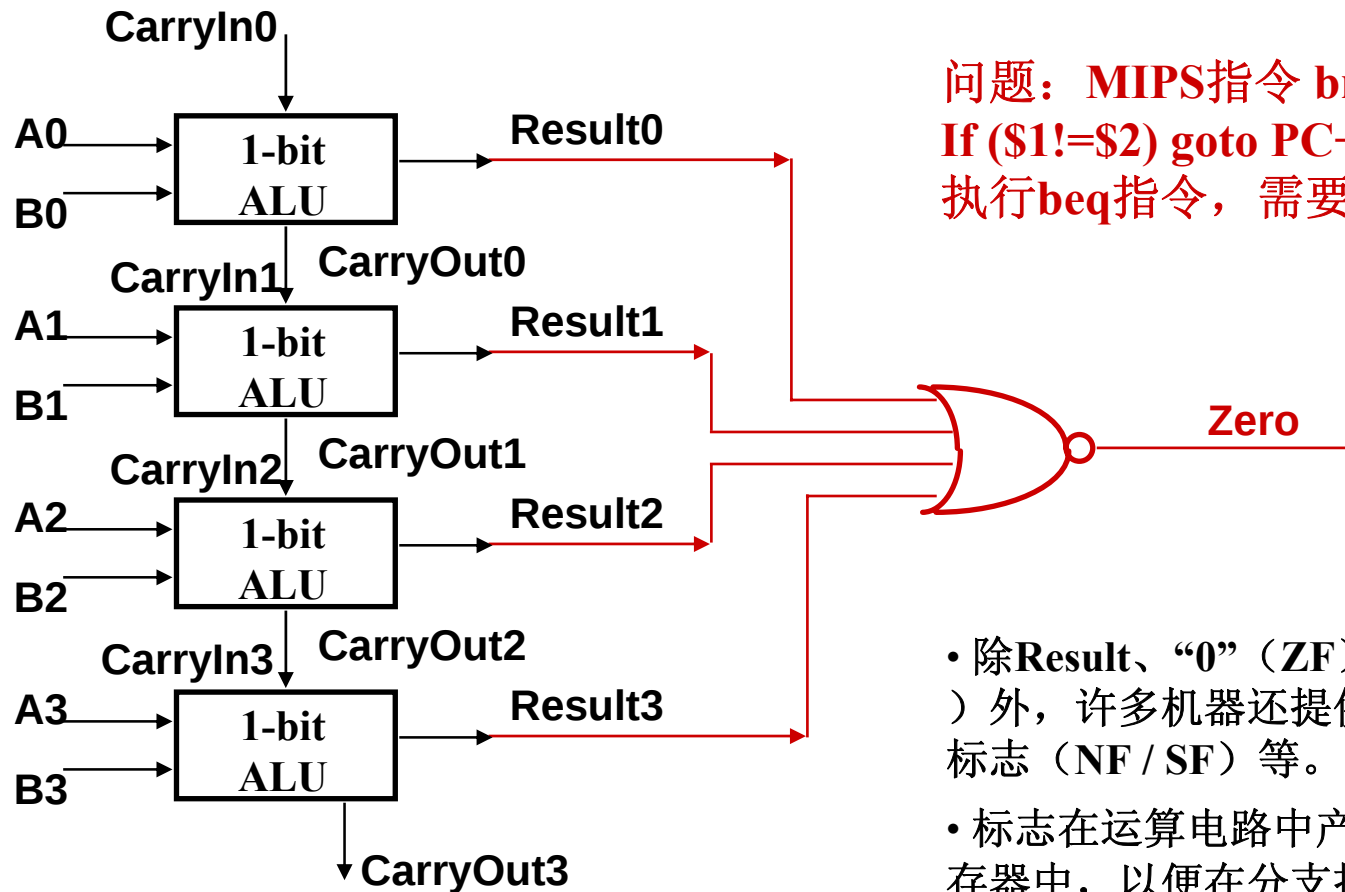


X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0



也可以用其他判断方法。

Zero Detection Logic(判0逻辑)



问题：MIPS指令 `bne $1,$2,25` 的含义为：
`If ($1!= $2) goto PC+4+100 else goto PC+4`
执行 `beq` 指令，需要判断什么标志？

- 除Result、“0”（ZF）、“Overflow”标志（OF）外，许多机器还提供进位标志（CF）、符号标志（NF / SF）等。
- 标志在运算电路中产生，被记录到专门的寄存器中，以便在分支指令中被用来作为条件。
- 存放标志的寄存器通常称为程序/状态字寄存器或标志寄存器。每个标志对应标志寄存器中的一个标志位。

原码加/减运算

- ◆ 用于浮点数尾数运算
- ◆ 符号位和数值部分分开处理
- ◆ 仅对数值部分进行加减运算，符号位起判断和控制作用
- ◆ 规则如下：
 - 比较两数符号，对加法实行“同号求和，异号求差”，对减法实行“异号求和，同号求差”。
 - 求和：数值位相加，若最高位产生进位，则结果溢出。和的符号取被加数（被减数）的符号。
 - 求差：被加数（被减数）加上加数（减数）的补码。分二种情况讨论：
 - a) 最高数值位产生进位，表明加法结果为正，所得数值位正确。
 - b) 最高数值位没有产生进位，表明加法结果为负，得到的是数值位的补码形式，需对结果求补，还原为绝对值形式的数值位。
 - 差的符号位：
 - a) 情况下，符号位取被加数（被减数）的符号
 - b) 情况下，符号位为被加数（被减数）的符号取反

原码加/减运算

例1：已知 $[X]_{\text{原}} = 1.0011$ ， $[Y]_{\text{原}} = 1.1010$ ，要求计算 $[X+Y]_{\text{原}}$

解：根据原码加减运算规则，知：两数同号，用加法求和，和的符号同被加数符号。

所以：和的数值位为： $0011 + 1010 = 1101$ （ALU中无符号数加）

和的符号位为：1

$[X+Y]_{\text{原}} = 1.1101$

求和：直接加，有进位则溢出，符号同被

例2：已知 $[X]_{\text{原}} = 1.0011$ ， $[Y]_{\text{原}} = 1.1010$ ，要求计算 $[X-Y]_{\text{原}}$

解：根据原码加减运算规则，知：两数同号，用减法求差（补码减法）

差的数值位为： $0011 + (1010)_{\text{补}} = 0011 + 0110 = 1001$

最高数值位没有产生进位，表明加法结果为负，需对1001求补，还

原为绝对值形式的数值位。即： $(1001)_{\text{补}} = 0111$

差的符号位为 $[X]$ 原的符号位取反，即：0

$[X-Y]_{\text{原}} = 0.0111$

求差：加补码，不会溢出，符号分情况

思考题：如何设计一个基于ALU的原码加/减法器？

移码加/减运算

- ◆ 用于浮点数阶码运算
- ◆ 符号位和数值部分可以一起处理
- ◆ 运算公式（假定在一个n位ALU中进行加法运算）

$$[E1]_{\text{移}} + [E2]_{\text{移}} = 2^{n-1} + E1 + 2^{n-1} + E2 = 2^n + E1 + E2 = [E1 + E2]_{\text{补}} \pmod{2^n}$$

$$\begin{aligned} [E1]_{\text{移}} - [E2]_{\text{移}} &= [E1]_{\text{移}} + [-[E2]_{\text{移}}]_{\text{补}} = 2^{n-1} + E1 + 2^n - [E2]_{\text{移}} = 2^{n-1} + E1 + 2^n - 2^{n-1} - E2 \\ &= 2^n + E1 - E2 = [E1 - E2]_{\text{补}} \pmod{2^n} \end{aligned}$$

结论：移码的和、差等于和、差的补码！

- ◆ 运算规则 补码和移码的关系：符号位相反、数值位相同！
 - ① 加法：直接将 $[E1]_{\text{移}}$ 和 $[E2]_{\text{移}}$ 进行模 2^n 相加，然后对结果的符号取反。
 - ② 减法：先将减数 $[E2]_{\text{移}}$ 求补（各位取反，末位加1），然后再与被减数 $[E1]_{\text{移}}$ 进行模 2^n 相加，最后对结果的符号取反。
 - ③ 溢出判断：进行模 2^n 相加时，如果两个加数的符号相同，并且与和数的符号也相同，则发生溢出。

思考题：如何设计一个基于ALU的移码加/减法器？

IEEE754 SP格式的偏置常数是127，这会不会影响阶码运算电路的复杂度？

对计算 $[E1-E2]_{\text{补}} \pmod{2^n}$ 没有影响，但 $[E1+E2]_{\text{移}}$ 和 $[E1-E2]_{\text{移}}$ 的计算变复杂！

$$[\Delta E]_{\text{补}} = 256 + E_x - E_y = 256 + 127 + E_x - (127 + E_y) = 256 + [E_x]_{\text{移}} - [E_y]_{\text{移}} = [E_x]_{\text{移}} + [-[E_y]_{\text{移}}]_{\text{补}} \pmod{256}$$

移码加/减运算

例1：用四位移码计算“-7+(-6)”和“-3+6”的值。

解：[-7]移 = 0001 [-6]移 = 0010 [-3]移 = 0101 [6]移 = 1110

[-7]移 + [-6]移 = 0001 + 0010 = 0011 (两个加数与结果符号都为0，溢出)

[-3]移 + [6]移 = 0101 + 1110 = 0011，符号取反后为 1011，其真值为+3

问题：[-7+(-6)]移=? [-3+(6)]移=? 

例2：用四位移码计算“-7-(-6)”和“-3-5”的值。

解：[-7]移 = 0001 [-6]移 = 0010 [-3]移 = 0101 [5]移 = 1101

[-7]移 - [-6]移 = 0001 + 1110 = 1111，符号取反后为 0111，其真值为-1。

[-3]移 - [5]移 = 0101 + 0011 = 1000，符号取反后为 0000，其真值为-8。

无符号数的乘法运算

◆ Paper and pencil example:

Multiplicand	1000
Multiplier (被乘数)	1001
(乘数)	1000
	0000
	0000
	1000
Product (积)	1001000

假定: $[X] = 0.x_1 \dots x_n$, $[Y] = 0.y_1 \dots y_n$,

则: $z_1 \dots z_{2n} = (0.x_1 \dots x_n) \times (0.y_1 \dots y_n)$

(小数点位置约定, 不区分小数还是整数)

$$X \times y_4 \times 2^{-4}$$

$$X \times y_3 \times 2^{-3}$$

$$X \times y_2 \times 2^{-2}$$

$$X \times y_1 \times 2^{-1}$$

$$X \times Y = \sum_{i=1}^4 (X \times y_i \times 2^{-i})$$

阵列乘法器

◆ 手工乘法的特点: 思考题: 若计算机完全模拟这样做, 则如何实现?

或

① 每步计算: $X \times y_i$, 若 $y_i = 0$, 则得0; 若 $y_i = 1$, 则得X

流水线乘法器

② 把①求得的各项结果 $X \times y_i$ 逐次左移, 可表示为 $X \times y_i \times 2^i$

③ 对②中求得的结果求和, 即 $\sum (X \times y_i \times 2^i)$, 这就是两个无符号数的乘积

◆ 计算机内部稍作以下改进:

① 每次得 $X \times y_i$ 后, 与前面所得结果累加, 得到 P_i , 称之为部分积。因为没有等到全部计算后一次求和, 所以减少了保存每次相乘结果 $X \times y_i$ 的开销。

② 每次得 $X \times y_i$ 后, 不将它左移与前次部分积 P_i 相加, 而将部分积 P_i 右移后与 $X \times y_i$ 相加。因为加法运算始终对部分积中高 n 位进行。故用 n 位加法器可实现二个 n 位数相乘。

③ 乘数中为“1”位执行加法和右移, 对为“0”位只执行右移, 而不执行加法运算。

整个运算过程中用到两种操作: 加法 + 左移 因而, 可用ALU和移位器来实现乘法运算

无符号乘法运算的算法推导

- ◆ 上述改进思想可写成如下数学推导过程：

$$\begin{aligned} X \times Y &= X \times (0.y_1 y_2 \dots y_n) \\ &= X \times y_1 \times 2^{-1} + X \times y_2 \times 2^{-2} + X \times y_3 \times 2^{-3} + \dots + X \times y_n \times 2^{-n} \\ &= \underbrace{2^{-1} (2^{-1} (2^{-1} \dots 2^{-1} (2^{-1} (0 + X \times y_n) + X \times y_{n-1}) + \dots + X \times y_2) + X \times y_1)}_{n \uparrow 2^{-1}} \end{aligned}$$

- ◆ 上述推导过程具有明显的递归性质，因此，无符号数乘法过程可归结为循环计算下列算式的过程：设 $P_0 = 0$ ，每步的乘积为：

$$P_1 = 2^{-1} (P_0 + X \times y_n)$$

$$P_2 = 2^{-1} (P_1 + X \times y_{n-1})$$

.....

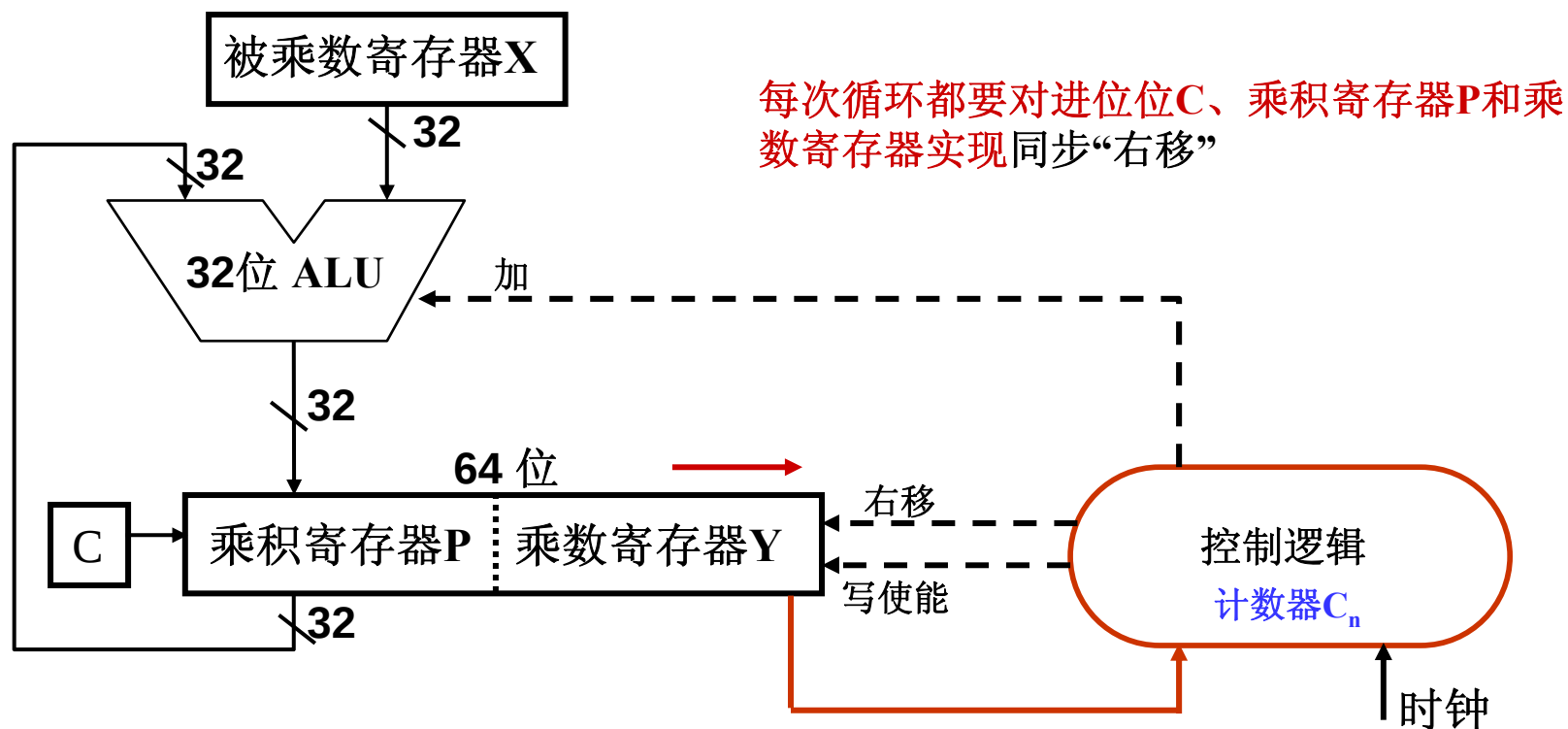
$$P_n = 2^{-1} (P_{n-1} + X \times y_1)$$

- ◆ 其递推公式为： $P_{i+1} = 2^{-1} (P_i + X \times y_{n-i}) \quad (i = 0, 1, 2, 3, \dots, n-1)$

- ◆ 最终乘积 $P_n = X \times Y$

迭代过程从乘数最低位 y_n 和 $P_0 = 0$ 开始，经 n 次“判断-加法-右移”循环，直到求出 P_n 为止。假定每次循环需要一个时钟周期，则 n 位乘法需要 n 个时钟周期完成。

32位乘法运算的硬件实现



- ◆ 被乘数寄存器X: 存放被乘数
- ◆ 乘积寄存器P: 开始时, 置初始部分积 $P_0 = 0$; 结束时, 存放的是64位乘积的高32位
- ◆ 乘数寄存器Y: 开始时, 置乘数; 结束时, 存放的是64位乘积的低32位
- ◆ 进位触发器C: 保存加法器的进位信号
- ◆ 循环次数计数器C_n: 存放循环次数。初值32, 每循环一次, C_n减1, C_n = 0时结束
- ◆ ALU: 乘法核心部件。在控制逻辑控制下, 对乘积寄存器P和被乘数寄存器X的内容进行“加”运算, 在“写使能”控制下运算结果被送回乘积寄存器P, 进位位存放在C中

Example: 无符号整数乘法运算

举例说明:

设A=1110 B=1101

应用递推公式: $P_i = 2^{-1}(Ab_i + P_{i-1})$

C	乘积P	乘数R
0	0000	1101
+	1110	
0	1110	1101
→ 0	0111	0110
→ 0	0011	1011
+	1110	
1	0001	1011
→ 0	1000	1101
+	1110	
1	0110	1101
→ 0	1011	0110

可用一个双倍字长的乘积寄存器; 也可用两个单倍字长的寄存器。

部分积初始为0。

保留进位位。

右移时进位、部分积和剩余乘数一起进行逻辑右移。

验证: $A=14, B=13, A \times B=182$

原码乘法算法

- ◆ 用于浮点数尾数乘运算
- ◆ 符号与数值分开处理：积符用两个符号异或得到，数值用无符号乘法运算

例：设 $[x]_{\text{原}}=0.1110$ ， $[y]_{\text{原}}=1.1101$ ，计算 $[X \times Y]_{\text{原}}$

解：数值部分用无符号数乘法算法计算： $1110 \times 1101 = 1011\ 0110$

符号位： $0 \oplus 1 = 1$ ，所以： $[X \times Y]_{\text{原}} = 1.10110110$

上述算法称为原码一位乘法，其实现思想为：

每次只取乘数中的一位进行判断，需 n 次循环，速度相对较慢。

原码两位乘法的思想：对乘数的每两位取值进行判断，每步求出对应两位的部分积。

- ◆ 原码两位乘法的操作的递推公式：

00 — $P_{i+1} = 2^{-2} P_i$

01 — $P_{i+1} = 2^{-2} (P_i + X)$

10 — $P_{i+1} = 2^{-2} (P_i + 2X)$

11 — $P_{i+1} = 2^{-2} (P_i + 3X) = 2^{-2} (P_i + 4X - X)$
 $= 2^{-2} (P_i - X) + X$

3X时，本次-X，下次+X！

y_{i-1}	y_i	T	操 作	迭 代 公 式
0	0	0	$0 \rightarrow T$	$2^{-2} (P_i)$
0	0	1	$+X \quad 0 \rightarrow T$	$2^{-2} (P_i + X)$
0	1	0	$+X \quad 0 \rightarrow T$	$2^{-2} (P_i + X)$
0	1	1	$+2X \quad 0 \rightarrow T$	$2^{-2} (P_i + 2X)$
1	0	0	$+2X \quad 0 \rightarrow T$	$2^{-2} (P_i + 2X)$
1	0	1	$-X \quad 1 \rightarrow T$	$2^{-2} (P_i - X)$
1	1	0	$-X \quad 1 \rightarrow T$	$2^{-2} (P_i - X)$
1	1	1	$1 \rightarrow T$	$2^{-2} (P_i)$

采用两位一乘，运算速度提高多少？

运算次数减少一半，速度提高一倍！

T触发器用来记录下次是否要执行“+X”

“-X”运算用“+[-X]_补”实现！

原码两位乘法举例

已知 $[X]_{\text{原}}=0.111001$, $[Y]_{\text{原}}=0.100111$, 用原码两位乘法计算 $[X \times Y]_{\text{原}}$

解: 先采用无符号数乘法计算 111001×100111 的乘积, 原码两位乘法过程如下:

$$[|X|]_{\text{补}} = 000\ 111001, [-|X|]_{\text{补}} = 111\ 000111$$

	P	Y	T	说明
采用补码右移	000 000000	100111	0	开始, $P_0=0$, $T=0$
数据为模8补码形式 (三位符号位), 为什么?	+111 000111			$y_5y_6T=110$, $-X$, $T=1$
	111 000111			P 和 Y 同时右移 2 位
	111 110001	11 1001	1	得 P_1
若采用模4补码, 则进行P和Y同时右移2位操作时, 按照补码右移规则, 得到的P3是负数, 显然, 两个正数相乘, 乘积不可能是负数	+001 110010			$y_3y_4T=011$, $+2X$, $T=0$
	001 100011			P 和 Y 同时右移 2 位
	000 011000	1111 10	0	得 P_2
	+001 110010			$y_1y_2T=100$, $+2X$, $T=0$
	010 001010			P 和 Y 同时右移 2 位
	000 100010	101111	0	得 P_3
加上符号位, 得 $[X \times Y]_{\text{原}}=0.100010101111$				

补码乘法运算

◆ 用于定点整数乘法运算 因为 $[A \times B]_{\text{补}} \neq [A]_{\text{补}} \times [B]_{\text{补}}$ ，故不能直接用无符号乘法计算。

◆ 符号与数值统一处理 **Booth's Algorithm**推导如下：

假定： $[A]_{\text{补}} = a_{n-1}a_{n-2}\cdots a_1a_0$ (a_{n-1} 为数符)

$[B]_{\text{补}} = b_{n-1}b_{n-2}\cdots b_1b_0$ (b_{n-1} 为数符)

求： $[AB]_{\text{补}} = ?$

基于以下补码性质：

令： $[A]_{\text{补}} = a_{n-1}a_{n-2}\cdots a_1a_0$ ，

则： $A = -a_{n-1} \cdot 2^{n-1} + a_{n-2} \cdot 2^{n-2} + \cdots + a_1 \cdot 2^1 + a_0 \cdot 2^0$

令： $a_{-1} = 0$ ，则：

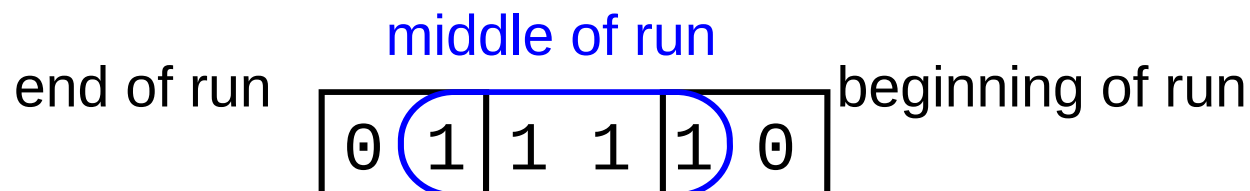
当 $n=32$ 时， $A = -a_{31} \cdot 2^{31} + a_{30} \cdot 2^{30} + \cdots + a_1 \cdot 2^1 + a_0 \cdot 2^0 + a_{-1} \cdot 2^0$

$$\downarrow$$
$$-a_{31} \cdot 2^{31} + (a_{30} \cdot 2^{31} - a_{30} \cdot 2^{30}) + \cdots + (a_0 \cdot 2^1 - a_0 \cdot 2^0) + a_{-1} \cdot 2^0$$

$$\downarrow$$
$$(a_{30} - a_{31}) \cdot 2^{31} + (a_{29} - a_{30}) \cdot 2^{30} + \cdots + (a_0 - a_1) \cdot 2^1 + (a_{-1} - a_0) \cdot 2^0$$

部分积公式： $P_i = 2^{-1}((a_{i-1} - a_i) \times B + P_{i-1})$

Booth's 算法实质



当前位	右边位	操作	Example
1	0	减被乘数	000111 <u>1</u> 000
1	1	加0(不操作)	00011 <u>1</u> 1000
0	1	加被乘数	00 <u>0</u> 1111000
0	0	加0(不操作)	0 <u>0</u> 1111000

- ◆ 最初提出这种想法是因为在Booth的机器上移位操作比加法更快！
- ◆ 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）

例：

Multiplicand	Product (2 x 7)	Multiplicand	Product (2 x -3)
0010	0000 0111 0	0010	0000 1101 0

Booths Example: 2 x 7

Operation	Multiplicand	Product Multiplier	next?
0. initial value	0010	0000 0111 0	10 -> sub
1a. $P = P - m$	1110	+ 1110 1110 0111 0	shift P (sign ext)
1b.	0010	1111 0011 1	11 -> nop, shift
2.	0010	1111 1001 1	11 -> nop, shift
3.	0010	1111 1100 1	01 -> add
4a.	0010	+ 0010 0001 1100 1	shift
4b.	0010	0000 1110 0	done

mythical bit

最后乘积

Booths Example: 2 x -3

mythical bit

Operation	Multiplicand	Product	next?
0. initial value	0010	0000 1101 0	10 -> sub
1a. $P = P - m$	1110	+ 1110 1110 1101 0	shift P (sign ext)
1b.	0010	1111 0110 1 + 0010	01 -> add
2a.		0001 0110 1	shift P
2b.	0010	0000 1011 0 + 1110	10 -> sub
3a.	0010	1110 1011 0	shift
3b.	0010	1111 0101 1	11 -> nop
4a		1111 0101 1	shift
4b.	0010	1111 1010 1	done

最后乘积

补码两位乘法

◆ 补码两位乘可用布斯算法推导如下：

$$\begin{aligned}
 \bullet [P_{i+1}]_{\text{补}} &= 2^{-1} ([P_i]_{\text{补}} + (y_{i-1} - y_i) [X]_{\text{补}}) \\
 \bullet [P_{i+2}]_{\text{补}} &= 2^{-1} ([P_{i+1}]_{\text{补}} + (y_i - y_{i+1}) [X]_{\text{补}}) \\
 &= 2^{-1} (2^{-1} ([P_i]_{\text{补}} + (y_{i-1} - y_i) [X]_{\text{补}}) + (y_i - y_{i+1}) [X]_{\text{补}}) \\
 &= 2^{-2} ([P_i]_{\text{补}} + (y_{i-1} + y_i - 2y_{i+1}) [X]_{\text{补}})
 \end{aligned}$$

◆ 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	$2^{-2}[P_i]_{\text{补}}$
0 0 1	$+ [X]_{\text{补}}$	$2^{-2}\{[P_i]_{\text{补}} + [X]_{\text{补}}\}$
0 1 0	$+ [X]_{\text{补}}$	$2^{-2}\{[P_i]_{\text{补}} + [X]_{\text{补}}\}$
0 1 1	$+ 2[X]_{\text{补}}$	$2^{-2}\{[P_i]_{\text{补}} + 2[X]_{\text{补}}\}$
1 0 0	$+ 2[-X]_{\text{补}}$	$2^{-2}\{[P_i]_{\text{补}} + 2[-X]_{\text{补}}\}$
1 0 1	$+ [-X]_{\text{补}}$	$2^{-2}\{[P_i]_{\text{补}} + [-X]_{\text{补}}\}$
1 1 0	$+ [-X]_{\text{补}}$	$2^{-2}\{[P_i]_{\text{补}} + [-X]_{\text{补}}\}$
1 1 1	0	$2^{-2}[P_i]_{\text{补}}$

补码两位乘法举例

◆ 已知 $[X]_{\text{补}} = 1\ 101$, $[Y]_{\text{补}} = 0\ 110$, 用补码两位乘法计算 $[X \times Y]_{\text{补}}$ 。

◆ 解: $[-X]_{\text{补}} = 0\ 011$, 用补码二位乘法计算 $[X \times Y]_{\text{补}}$ 的过程如下。

P_n	P	Y	y_{-1}	说明
0	0000	01 <u>10</u>	0	开始, 设 $y_{-1} = 0$, $[P_0]_{\text{补}} = 0$
+ 0	0110			$y_1y_0y_{-1} = 100$, $+2[-X]_{\text{补}}$
	00110			P 和 Y 同时右移二位
→ 0	0001	10 <u>01</u>	1	得 $[P_2]_{\text{补}}$
+ 1	1010			$y_3y_2y_1 = 011$, $+2[X]_{\text{补}}$
	11011			P 和 Y 同时右移二位
→ 1	1110	1110		得 $[P_4]_{\text{补}}$

因此 $[X \times Y]_{\text{补}} = 1110\ 1110$, 与一位补码乘法 (布斯乘法) 所得结果相同, 但循环次数减少了一半。

验证: $-3 \times 6 = -18$ ($-10010B$)

快速乘法器

- ◆ 前面介绍的乘法部件的特点:

- 通过一个**ALU**多次“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”

存在瓶颈: 所需时间随位数增多而加长

- ◆ 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

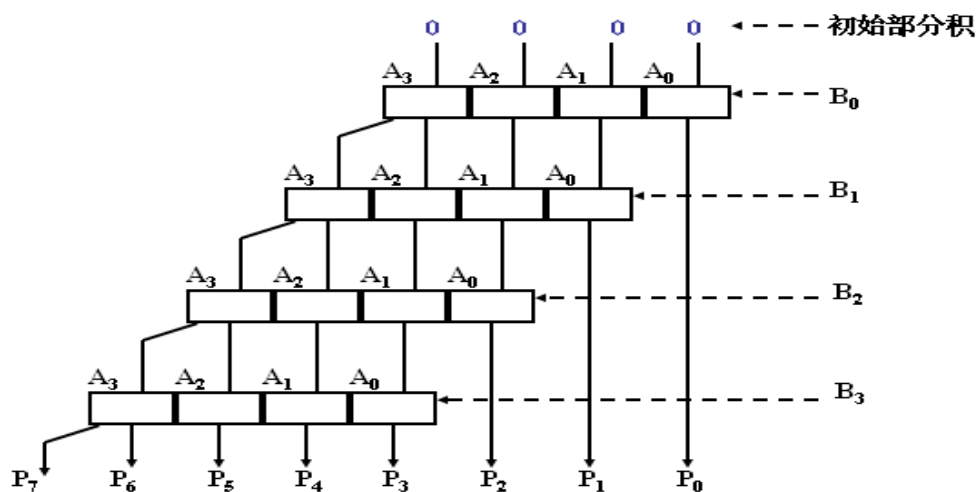
- 流水线方式
 - 硬件叠加方式 (如: 阵列乘法器)

- ◆ 阵列乘法器

- 用一个实现特定功能的组合逻辑单元构成一个阵列

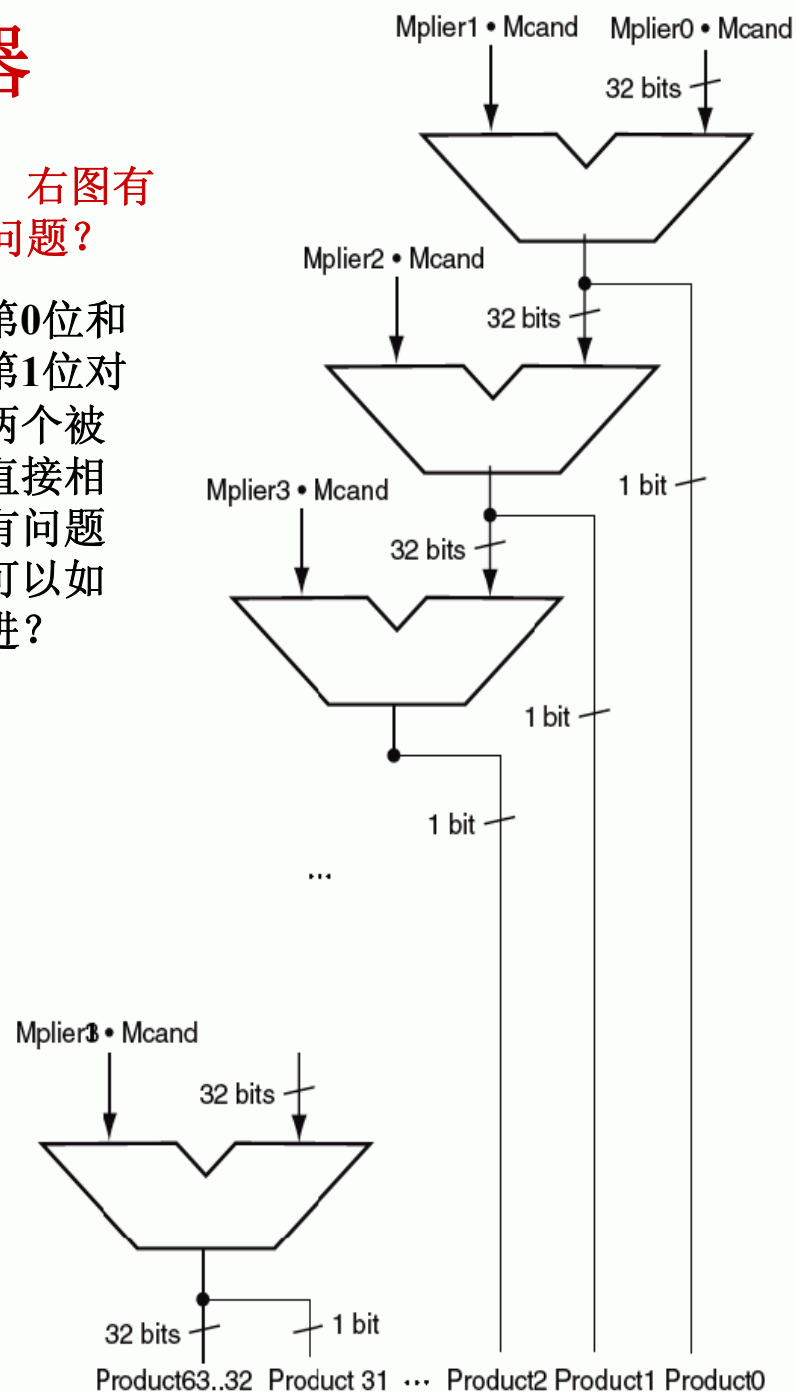
流水线方式的快速乘法器

- ◆ 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



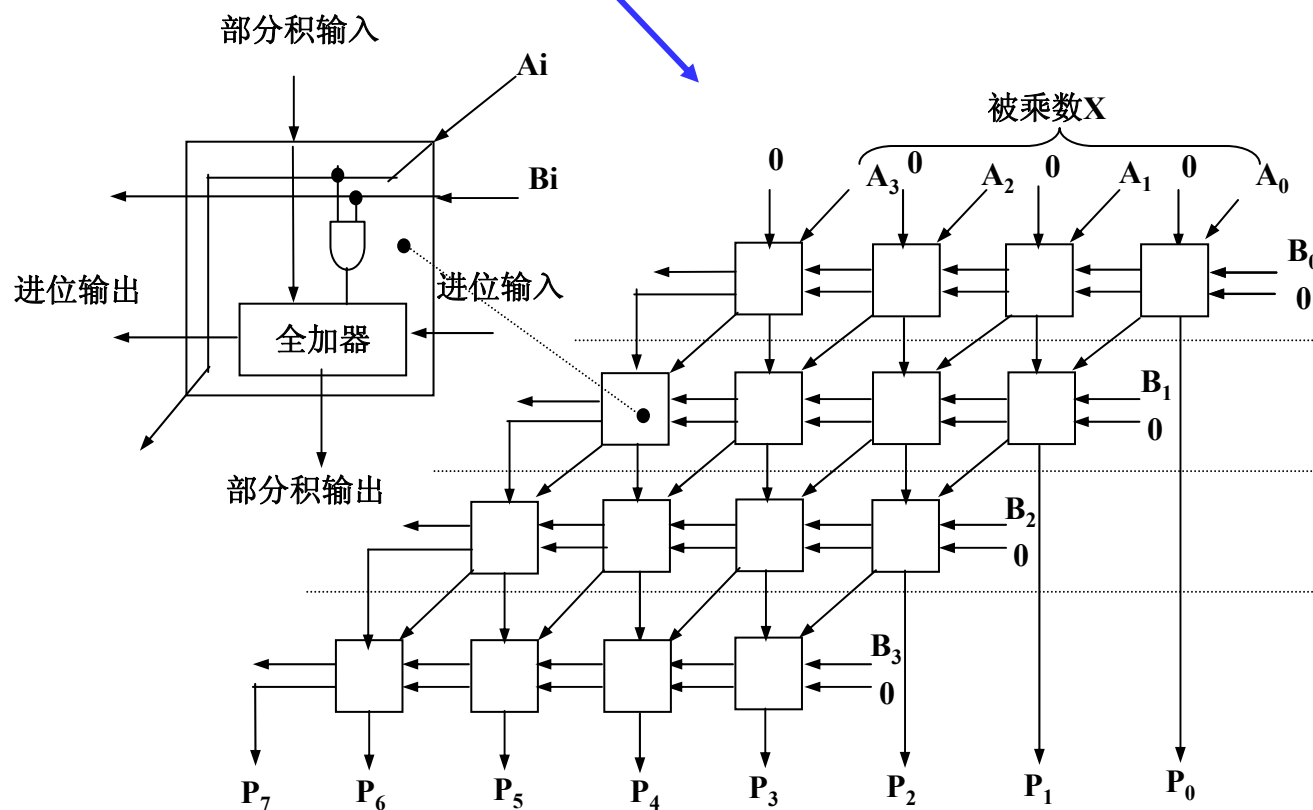
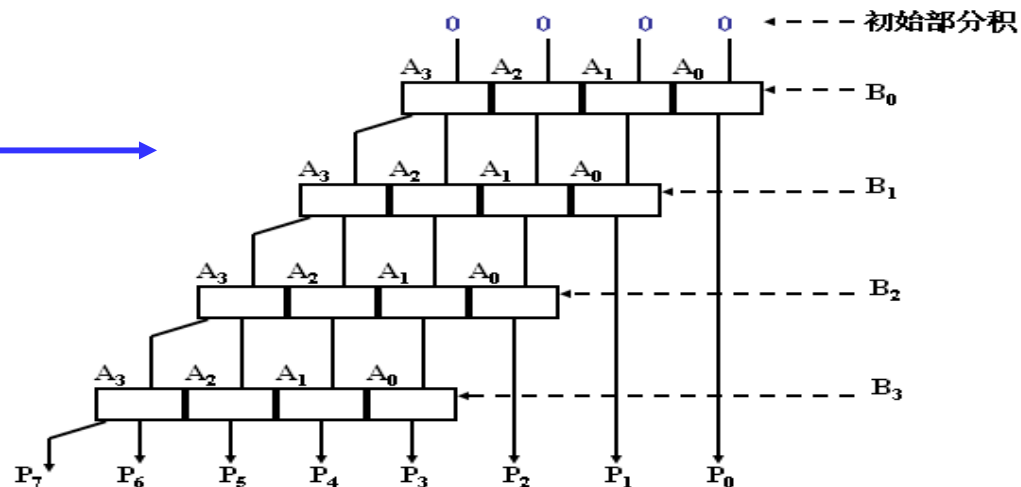
问题：右图有
没有问题？

乘数第0位和
乘数第1位对
应的两个被
乘数直接相
加，有问题
吗？可以如
何改进？



阵列乘法器的实现

- ◆ 手算乘法过程
- ◆ 阵列乘法器



乘法速度仅取决于逻辑门和加法器的传输延迟

在无符号数阵列乘法器的基础上，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。

定点除法运算

◆ 除前预处理

- ①若被除数为0、除数不为0，或 定点整数除法 $|被除数| < |除数|$ ，则商为0，不再继续执行
- ②若被除数不为0、除数为0，则发生“除数为0”异常
- ③若被除数和除数都为0，则有些机器产生一个不发信号的NaN，即“quiet NaN”

只有当被除数和除数都不为0，并且商也不可能为0时，才进一步进行除法运算。

◆ 手算除法的基本要点

- 被除数与除数相减，若够减，则上商为1；若不够减，则上商为0。
- 每次得到的差为中间余数，将除数右移后与上次的中间余数比较。用中间余数减除数，若够减，则上商为1；若不够减，则上商为0。
- 重复执行第②步，直到求得的商的位数足够为止。

◆ 计算机内部无符号数除法运算

- 与手算一样，通过被除数（中间余数）减除数来得到每一位商
够减上商1；不够减上商0
- 基本操作为减法（用加法实现）和移位，故可与乘法合用同一套硬件

Divide: Paper & Pencil

Divisor 1000

$$\begin{array}{r} 1001 \\ 1000 \overline{) 1001010} \\ \underline{-1000} \\ 10 \\ \underline{101} \\ \underline{1010} \\ \underline{-1000} \\ 10 \end{array}$$

Quotient(商)
Dividend(被除数)

中间余数

Remainder (余数)

◆ 手算除法的基本要点

- 被除数与除数相减，若够减，则上商为**1**；若不够减，则上商为**0**。
- 每次得到的差为中间余数，将除数右移后与上次的中间余数比较。用中间余数减除数，若够减，则上商为**1**；若不够减，则上商为**0**。
- 重复执行第②步，直到求得的商的位数足够为止。

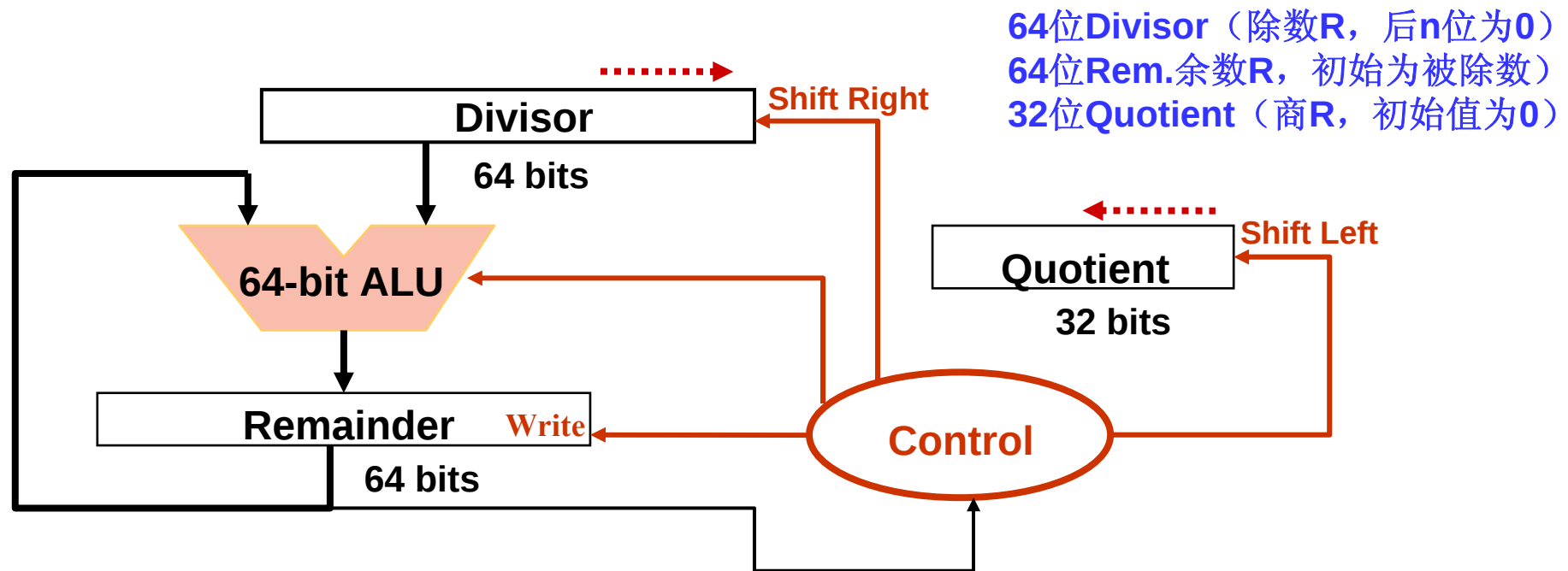
完全模拟手工的无符号数除法硬件实现

两个n位数相除的情况：

(1)两个n位定点正整数（即：两个n位无符号数）相除：在被除数的高位添n个0

(2)两个n位定点正小数（即两个作为浮点数尾数的n位原码小数）相除：在被除数的低位添加n个0

这样，就将所有情况都统一为：一个2n位数除以一个n位数 以下是64位数除以32位数的例子

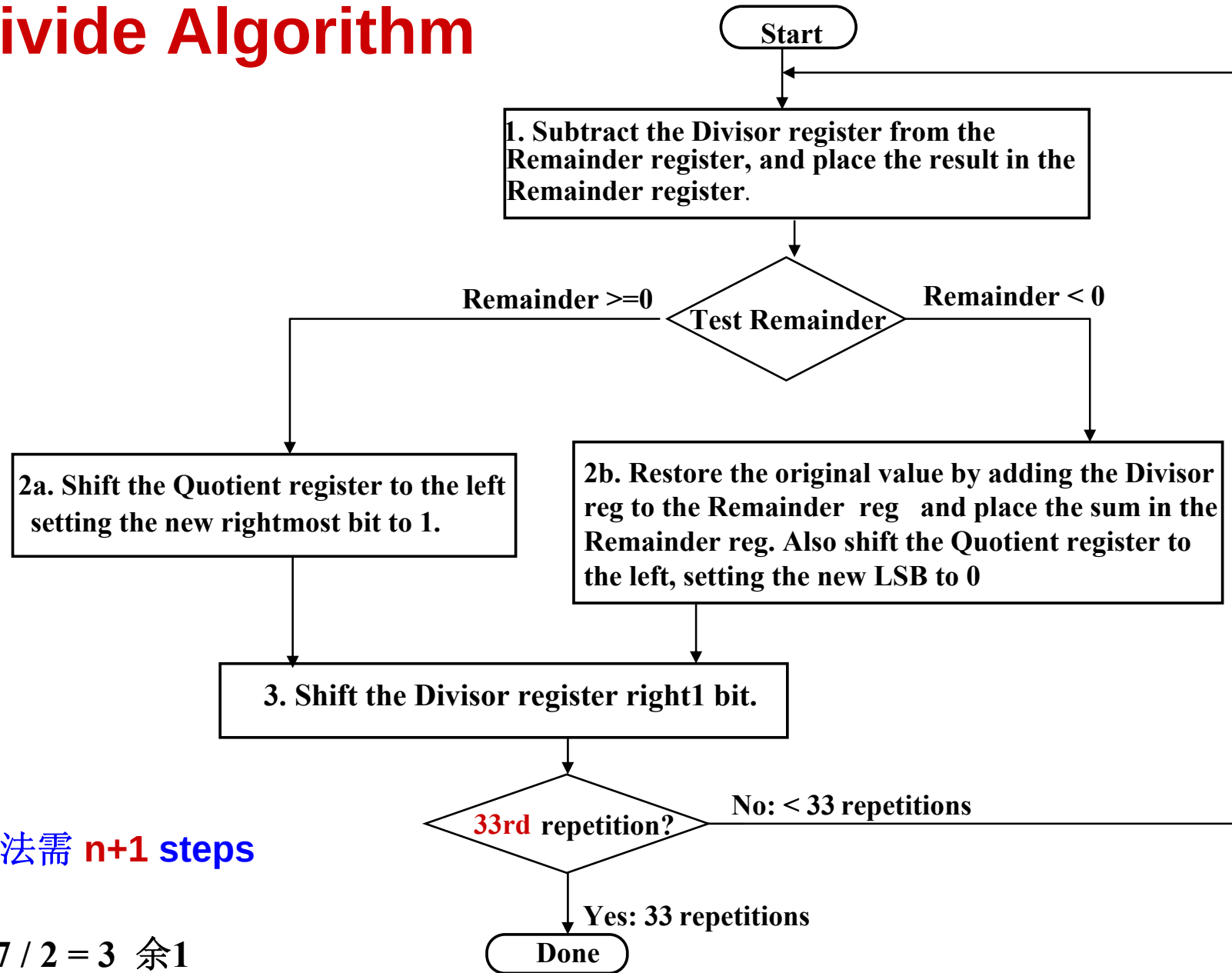


问题：第一次试商为1，说明什么？

若是无符号整数运算，则说明将会得到n+1位的商，因而结果“溢出”。但若是两个n位数相除，则肯定不会溢出，为什么？最大商为： $11...11/00...01=11...1$

若是浮点数中尾数原码小数运算，则说明尾数部分有“溢出”，可通过浮点数的“右规”消除“溢出”。所以，在浮点数运算器中，第一次得到的商“1”要保留。

手工Divide Algorithm



◆ **n**位除法需 **n+1** steps

举例: $7 / 2 = 3$ 余1

Divide Algorithm--example

	Q: 0000	D: 0010 0000	R: 0000 0111 -D = 1110 0000
1: R = R-D	Q: 0000	D: 0010 0000	R: <u>1110 0111</u>
2b: +D, sl Q, 0	Q: <u>000</u> 0	D: 0010 0000	R: <u>0000 0111</u>
3: Shr D	Q: 0000	D: <u>0001 0000</u>	R: 0000 0111 -D = 1111 0000
1: R = R-D	Q: 0000	D: 0001 0000	R: <u>1111 0111</u>
2b: +D, sl Q, 0	Q: <u>00</u> 00	D: 0001 0000	R: <u>0000 0111</u>
3: Shr D	Q: 0000	D: <u>0000 1000</u>	R: 0000 0111 -D = 1111 1000
1: R = R-D	Q: 0000	D: 0000 1000	R: <u>1111 1111</u>
2b: +D, sl Q, 0	Q: <u>00</u> 00	D: 0000 1000	R: <u>0000 0111</u>
3: Shr D	Q: 0000	D: <u>0000 0100</u>	R: 0000 0111 -D = 1111 1100
1: R = R-D	Q: 0000	D: 0000 0100	R: <u>0000 0011</u>
2a: sl Q, 1	Q: <u>000</u> 1	D: 0000 0100	R: 0000 0011
3: Shr D	Q: 0001	D: <u>0000 0010</u>	R: 0000 0011 -D = 1111 1110
1: R = R-D	Q: 0001	D: 0000 0010	R: <u>0000 0001</u>
2a: sl Q, 1	Q: <u>001</u> 1	D: 0000 0010	R: 0000 0001
3: Shr D	Q: 0011	D: <u>0000 0001</u>	R: 0000 0001

验证: $7 / 2 = 3$ 余 1

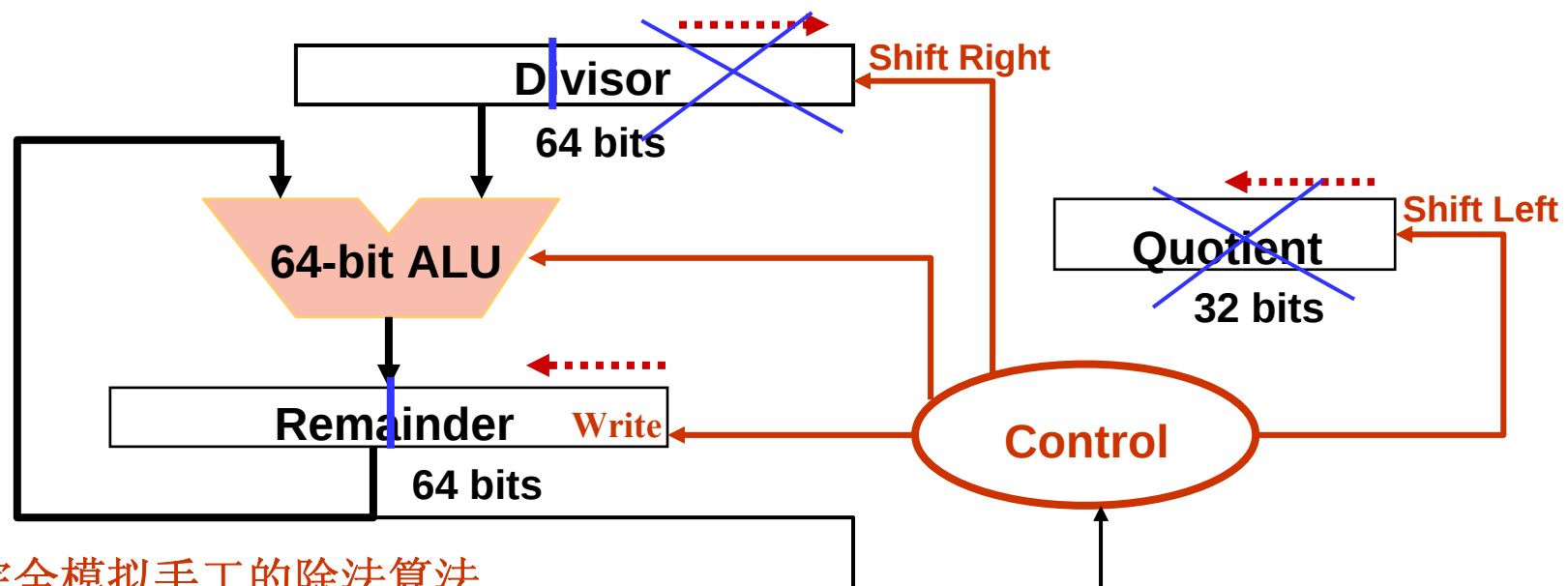
问题: 这里Q中第一个“0”说明什么?

不是真正的商, 而是表示没有“溢出”!

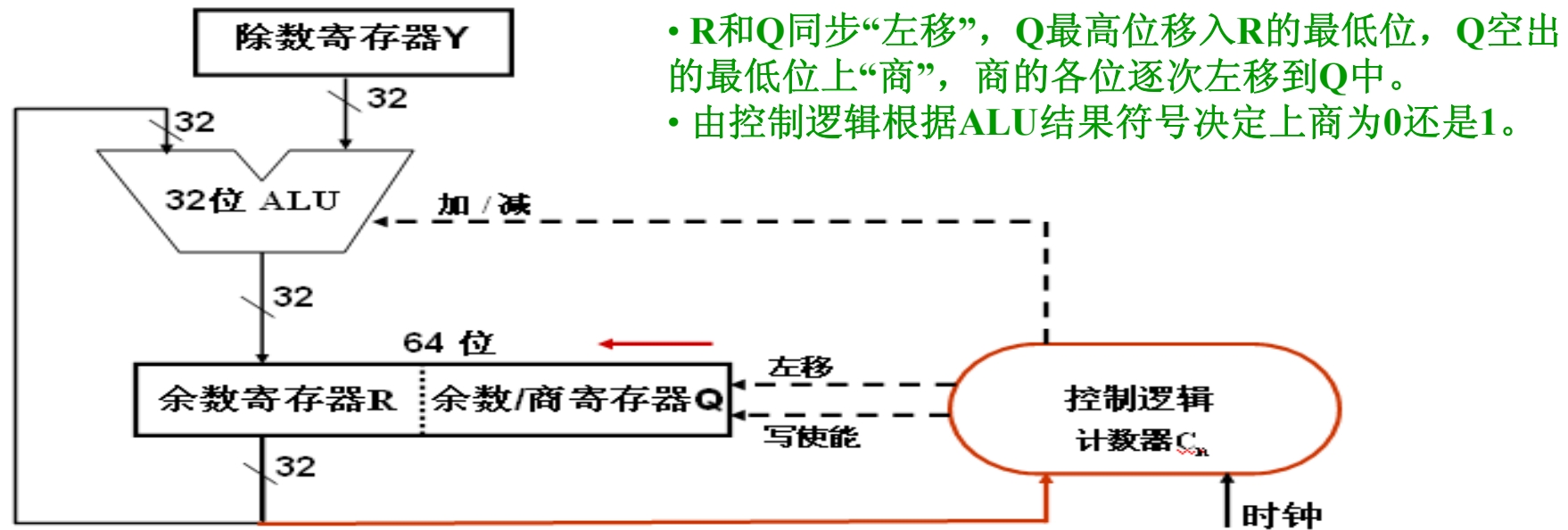
对手算除法算法的观察

- ◆ 1/2 bits in divisor (除数寄存器) always 0
=> 1/2 of 64-bit adder、1/2 of divisor is wasted
- ◆ 能否考虑只用32位除数寄存器和32位ALU?
- ◆ 可用余数左移代替除数右移
- ◆ 可使商通过和余数一起左移来取消商寄存器
 - 开始时先使余数左移一位 (问题: 为什么可这样做?)
 - 因为余数寄存器的左移实际上使左半部的余数和右半部的商同时左移, 所以循环里面只包含了两步。
 - 两个寄存器的结合以及循环内次序的调换使得余数被多左移了一次。所以最后一步余数寄存器的左半边的余数必须向右移一位。

因为结果肯定不“溢出”, 故第1次商肯定为0, 不用先做减法试商, 即: 将第一步换成先左移, 可减少一次迭代



无符号数除法算法的硬件实现



- R和Q同步“左移”，Q最高位移入R的最低位，Q空出的最低位上“商”，商的各位逐次左移到Q中。
- 由控制逻辑根据ALU结果符号决定上商为0还是1。

- ◆ 除数寄存器Y：存放除数。
- ◆ 余数寄存器R：置初始中间余数R0的高位部分为高32位被除数；结束时，存放的是余数。
- ◆ 余数/商寄存器Q：开始时，置初始中间余数R0的低位部分为低32位被除数；结束时，存放的是32位商。因为寄存器Q中存放的并不是商的全部位数，而是部分为被除数或中间余数，部分为商，只有到最后一步才是商的全部位数。
- ◆ 循环次数计数器Cn：存放循环次数。初值是32，每循环一次，Cn减1，当Cn = 0时，除法运算结束。
- ◆ ALU：除法核心部件。在控制逻辑控制下，对于余数寄存器R和除数寄存器Y的内容进行“加/减”运算，在“写使能”控制下运算结果被送回余数寄存器R。

Divide Algorithm example

Divisor	Remainder	
0010	0000	0111
	D: 0010	R: 0000 0111
Shl R	D: 0010	R: <u>0000</u> 1110
R = R-D	D: 0010	R: <u>1110</u> 1110
+D, sl R, 0	D: 0010	R: <u>0001</u> <u>1100</u>
R = R-D	D: 0010	R: <u>1111</u> 1100
+D, sl R, 0	D: 0010	R: <u>0011</u> <u>1000</u>
R = R-D	D: 0010	R: <u>0001</u> 1000
sl R, 1	D: 0010	R: <u>0011</u> <u>0001</u>
R = R-D	D: 0010	R: <u>0001</u> 0001
sl R, 1	D: 0010	R: <u>0010</u> <u>0011</u>
Shr R(rh)	D: 0010	R: <u>0001</u> 0011

从例子可看出：

每次上商为0时，需做加法，以“恢复余数”。所以，称为“恢复余数法”。

也可在下一步运算时把当前多减的除数补回来。这种方法称为“不恢复余数法”，又称“加减交替法”。

最后的余数需向右移一位。

验证：7 / 2 = 3 余 1

不恢复余数除法(加减交替法)

恢复余数法可进一步简化为“加减交替法”

根据恢复余数法(设B为除数, R_i 为第*i*次中间余数), 有:

- 若 $R_i < 0$, 则商上“0”, 做加法恢复余数, 即:

$$R_{i+1} = 2(R_i + 2^n|B|) - 2^n|B| = 2R_i + 2^n|B|$$

(由上式可知: “负, 0, 加”)

- 若 $R_i > 0$, 则商上“1”, 不需恢复余数, 即:

$$R_{i+1} = 2R_i - 2^n|B|$$

(由上式可知: “正, 1, 减”)

省去了恢复余数的过程

注意: 最后一次上商为“0”的话, 需要“纠余”处理, 即把试商时被减掉的除数加回去, 恢复真正的余数。

不恢复余数法也称为加减交替法

带符号数除法

◆ 原码除法

○ 商符和商值分开处理

- 商的数值部分由无符号数除法求得
- 商符由被除数和除数的符号确定：同号为 0，异号为 1

○ 余数的符号同被除数的符号

◆ 补码除法

○ 方法1：同原码除法一样，先转换为正数，先用无符号数除法，然后修正商和余数。

○ 方法2：直接用补码除法，符号和数值一起进行运算，商符直接在运算中产生。

对于两个 n 位补码整数的除法运算，被除数需要进行符号扩展。

若被除数为 $2n$ 位，除数为 n 位，则被除数无需扩展。

原码除法举例

已知 $[X]_{\text{原}} = 0.1011$

$[Y]_{\text{原}} = 1.1101$

用恢复余数法计算 $[X/Y]_{\text{原}}$

解：分符号位和数值位两部分进行。商的符号位： $0 \oplus 1 = 1$

商的数值位采用恢复余数法。

减法操作用补码加法实现，是否够减通过中间余数的符号来判断，所以中间余数要加一位符号位。

因此，需先计算出：

$[X]_{\text{补}} = 0.1011$

$[Y]_{\text{补}} = 0.1101$

$[-Y]_{\text{补}} = 1.0011$

因为是原码定点小数，所以在被除数低位扩展0

思考：若实现无符号数相除即：1011除以1101，则有何不同？结果是什么？

余数寄存器 R	余数/商寄存器 Q	说 明
01011	0000□	开始 $R_0 = X$
+10011		$R_1 = X - Y$
11110	00000	$R_1 < 0$, 则 $q_4 = 0$
+01101		恢复余数: $R_1 = R_1 + Y$
01011		得 R_1
10110	0000□	$2R_1$ (R 和 Q 同时左移, 空出一位商)
+10011		$R_2 = 2R_1 - Y$
01001	00001	$R_2 > 0$, 则 $q_3 = 1$
10010	0001□	$2R_2$ (R 和 Q 同时左移, 空出一位商)
+10011		$R_3 = 2R_2 - Y$
00101	00011	$R_3 > 0$, 则 $q_2 = 1$
01010	0011□	$2R_3$ (R 和 Q 同时左移, 空出一位商)
+10011		$R_4 = 2R_3 - Y$
11101	00110	$R_4 < 0$, 则 $q_1 = 0$
+01101		恢复余数: $R_4 = R_4 + Y$
01010	00110	得 R_4
10100	0110□	$2R_4$ (R 和 Q 同时左移, 空出一位商)
+10011		$R_5 = 2R_4 - Y$
00111	01101	$R_5 > 0$, 则 $q_0 = 1$

商的最高位为 0, 说明没有溢出, 商的数值部分为: 1101。

所以, $[X/Y]_{\text{原}} = 1.1101$ (最高位为符号位), 余数为 0.0111×2^4 。

在确认不会溢出时可省略

原码除法举例

已知 $[X]_{\text{原}} = 0.1011$

$[Y]_{\text{原}} = 1.1101$

用不恢复余数法计算 $[X/Y]_{\text{原}}$

解: $[[X]]_{\text{补}} = 0.1011$

$[[Y]]_{\text{补}} = 0.1101$

$[-Y]_{\text{补}} = 1.0011$

“加减交替法”的要点:

正、1、减

负、0、加

最后一步得到的结果与
恢复余数法一样!

余数寄存器 R	余数/商寄存器 Q	说 明
01011	0000□	开始 $R_0 = X$
+10011		$R_1 = X - Y$
11110	00000	$R_1 < 0$, 则 $q_4 = 0$, 没有溢出
11100	0000□	$2R_1$ (R 和 Q 同时左移, 空出一位商)
+01101		$R_2 = 2R_1 + Y$
01001	00001	$R_2 > 0$, 则 $q_3 = 1$
10010	0001□	$2R_2$ (R 和 Q 同时左移, 空出一位商)
+10011		$R_3 = 2R_2 - Y$
00101	00011	$R_3 > 0$, 则 $q_2 = 1$
01010	0011□	$2R_3$ (R 和 Q 同时左移, 空出一位商)
+10011		$R_4 = 2R_3 - Y$
11101	00110	$R_4 < 0$, 则 $q_1 = 0$
11010	0110□	$2R_4$ (R 和 Q 同时左移, 空出一位商)
+01101		$R_5 = 2R_4 + Y$
00111	01101	$R_5 > 0$, 则 $q_0 = 1$

问题: 对于原码小数和无符号整数的除法运算, 用被除数 (中间余数) 减除数进行试商时, 根据什么来确定是否“够减”?

中间余数的符号!

补码除法能否这样来判断呢?

实现补码除法的基本思想

◆ 补码除法判断是否“够减”的规则

- (1) 当被除数（或中间余数）与除数同号时，做减法，若新余数的符号与除数符号一致表示够减，否则为不够减；
- (2) 当被除数（或中间余数）与除数异号时，做加法，若得到的新余数的符号与除数符号一致表示不够减，否则为够减。

上述判断规则用表表示为：

中间 余数R	除数Y	新中间余数：R-Y（正商）		新中间余数：R+Y（负商）	
		0	1	0	1
0	0	够减	不够减		
0	1			够减	不够减
1	0			不够减	够减
1	1	不够减	够减		

从上表可得到补码除法的基本算法思想：

(1) 运算规则：

当被除数（或中间余数）与除数同号时，做减法；异号时，做加法。

SKIP

(2) 上商规则：

若新余数与原余数符号一致（即余数符号不变），则够减，商1；否则不够减，商0。

若商为正时：新余数与除数符号一致，够减，商1；否则不够减，商0。（原码）

若商为负时：新余数与除数符号不一致，够减，商0；否则不够减，商1。（反码）

补码除法也有：恢复余数法和不恢复余数法

补码恢复余数法

◆ 算法要点:

(1) 操作数的预置:

除数装入除数寄存器Y，被除数经符号扩展后装入余数寄存器R和余数/商寄存器Q。

(2) R和Q同步串行左移一位。

(3) 若R与Y同号，则 $R = R - Y$ ；否则 $R = R + Y$ ，并按以下规则确定商值 q_0 ：

① 若中间余数R、 $Q = 0$ 或R操作前后符号未变，表示够减，则 q_0 置1，转下一步；

② 若操作前后R的符号已变，表示不够减，则 q_0 置0，恢复R值后转下一步；

(4) 重复第(2)和第(3)步，直到取得n位商为止。

(5) 若被除数与除数同号，则Q中就是真正的商；否则，将Q求补后是真正的商。

（即：若商为负值，则需要“各位取反，末位加1”来得到真正的商）

(6) 余数在R中。

问题：如何恢复余数？通过“做加法”来恢复吗？

无符号数（或原码）除法通过“做加法”恢复余数，但补码不是！

若原来为 $R = R - Y$ ，则执行 $R = R + Y$ 来恢复余数，否则若原来是 $R = R + Y$ ，则执行 $R = R - Y$ 。

举例：7/3=? (-7)/3=?

被除数：0000 0111 除数 0011

	A	Q	M=0011
	0000	0111	
←	0000	1110	
+	1101		减
	1101	1110	
+	0011		恢复(加)商0
	0000	1110	
←	0001	1100	
+	1101		减
	1110	1100	
+	0011		恢复(加)商0
	0001	1100	
←	0011	1000	
+	1101		减
	0000	1001	符同商1
←	0001	0010	
+	1101		减
	1110	0010	
+	0011		恢复(加)商0
	0001	0010	

余:0001/商:0010

验证：7/3 = 2，余数为1

被除数：1111 1001 除数 0011

	A	Q	M=0011
	1111	1001	
←	1111	0010	
+	0011		加
	0010	0010	
+	1101		恢复(减)商0
	1111	0010	
←	1110	0100	
+	0011		加
	0001	0100	
+	1101		恢复(减)商0
	1110	0100	
←	1100	1000	
+	0011		加
	1111	1001	符同商1
←	1111	0010	
+	0011		加
	0010	0010	
+	1101		恢复(减)商0
	1111	0010	

商为负数，需求补：0010→1110

余:1111/商:1110

验证：-7/3 = -2，余数为-1

补码不恢复余数法

◆ 算法要点:

(1) 操作数的预置:

除数装入除数寄存器Y，被除数经符号扩展后装入余数寄存器R和余数/商寄存器Q。

(2) 根据以下规则求第一位商 q_n :

若被除数X与Y同号，则 $R1=X-Y$ ；否则 $R1=X+Y$ ，并按以下规则确定商值 q_n :

① 若新的中间余数R1与Y同号，则 q_n 置1，转下一步；

② 若新的中间余数R1与Y异号，则 q_n 置0，转下一步；

q_n 用来判断是否溢出，而不是真正的商。以下情况下会发生溢出:

若X与Y同号且上商 $q_n=1$ ，或者，若X与Y异号且上商 $q_n=0$ 。

判断是否同号与恢复余数法不同，不是新老余数！

是余数和除数之间

同号则够减，商1？

异号则不够减，商0？

错！参看前面的表

(3) 对于 $i=1$ 到 n ，按以下规则求出 n 位商:

① 若 R_i 与Y同号，则 q_{n-i} 置1， $R_{i+1}=2R_i-[Y]$ 补， $i=i+1$ ；

② 若 R_i 与Y异号，则 q_{n-i} 置0， $R_{i+1}=2R_i+[Y]$ 补， $i=i+1$ ；

(4) 商的修正: 最后一次Q寄存器左移一位，将最高位 q_n 移出，最低位置上商 q_0 。若被除数与除数同号，Q中就是真正的商；否则，将Q中商的末位加1。商已经是“反码”

(5) 余数的修正: 若余数符号同被除数符号，则不需修正，余数在R中；否则，按下列规则进行修正: 当被除数和除数符号相同时，最后余数加除数；否则，最后余数减除数。

补码不恢复余数法也有一个六字口诀“同、1、减；异、0、加”。其运算过程也呈加/减交替方式，因此也称为“加减交替法”。

举例：-9/2

将X=-9和Y=2分别表示成5位补码形式为：

[X]补 = 1 0111

[Y]补 = 0 0010

被除数进行符号扩展为：

[X]补=11111 10111

[-Y]补 = 1 1110

同、1、减
异、0、加

$X/Y = -0100B = -4$,

余数为 $-0001B = -1$

将各数代入公式：

“除数 $\times$ 商+余数=被除数”进行验证，得：

$2 \times (-4) + (-1) = -9$

余数寄存器 R	余数/商寄存器 Q	说 明
11111	10111	开始 $R_0 = [X]$
+00010		$R_1 = [X] + [Y]$
00001	10111	R_1 与 $[Y]$ 同号，则 $q_5 = 1$
00011	01111	$2R_1$ (R 和 Q 同时左移，空出一位上商 1)
+11110		$R_2 = 2R_1 + [-Y]$
00001	01111	R_2 与 $[Y]$ 同号，则 $q_4 = 1$
00010	11111	$2R_2$ (R 和 Q 同时左移，空出一位上商 1)
+11110		$R_3 = 2R_2 + [-Y]$
00000	11111	R_3 与 $[Y]$ 同号，则 $q_3 = 1$
00001	11111	$2R_3$ (R 和 Q 同时左移，空出一位上商 1)
+11110		$R_4 = 2R_3 + [-Y]$
11111	11111	R_4 与 $[Y]$ 异号，则 $q_2 = 0$
11111	11110	$2R_4$ (R 和 Q 同时左移，空出一位上商 0)
+00010		$R_5 = 2R_4 + [Y]$
00001	11110	R_5 与 $[Y]$ 同号，则 $q_1 = 1$
00011	11101	$2R_5$ (R 和 Q 同时左移，空出一位上商 1)
+11110		$R_6 = 2R_5 + [-Y]$
00001	11101	R_6 与 $[Y]$ 同号，则 $q_0 = 1$ ，Q 左移，空出一位上商 1
+11110	+ 1	商为负数，末位加 1；减除数以修正余数
11111	11100	

所以， $[X/Y]_补 = 11100$ 。余数为 11111。

快速除法器

问题：可以像乘法一样用32个Adder同时进行加/减运算来实现流水线方式的快速除法器吗？

不行！每次做加法还是减法，必须要知道上次余数的符号。

- ◆ 很难实现流水化
- ◆ 比快速乘法器更难实现。阵列除法器比阵列乘法器复杂

右图是实现两个正数按不恢复余数法进行相除的阵列除法器

第一次总是做减法，故 $P=1$ 使第一行做减法；

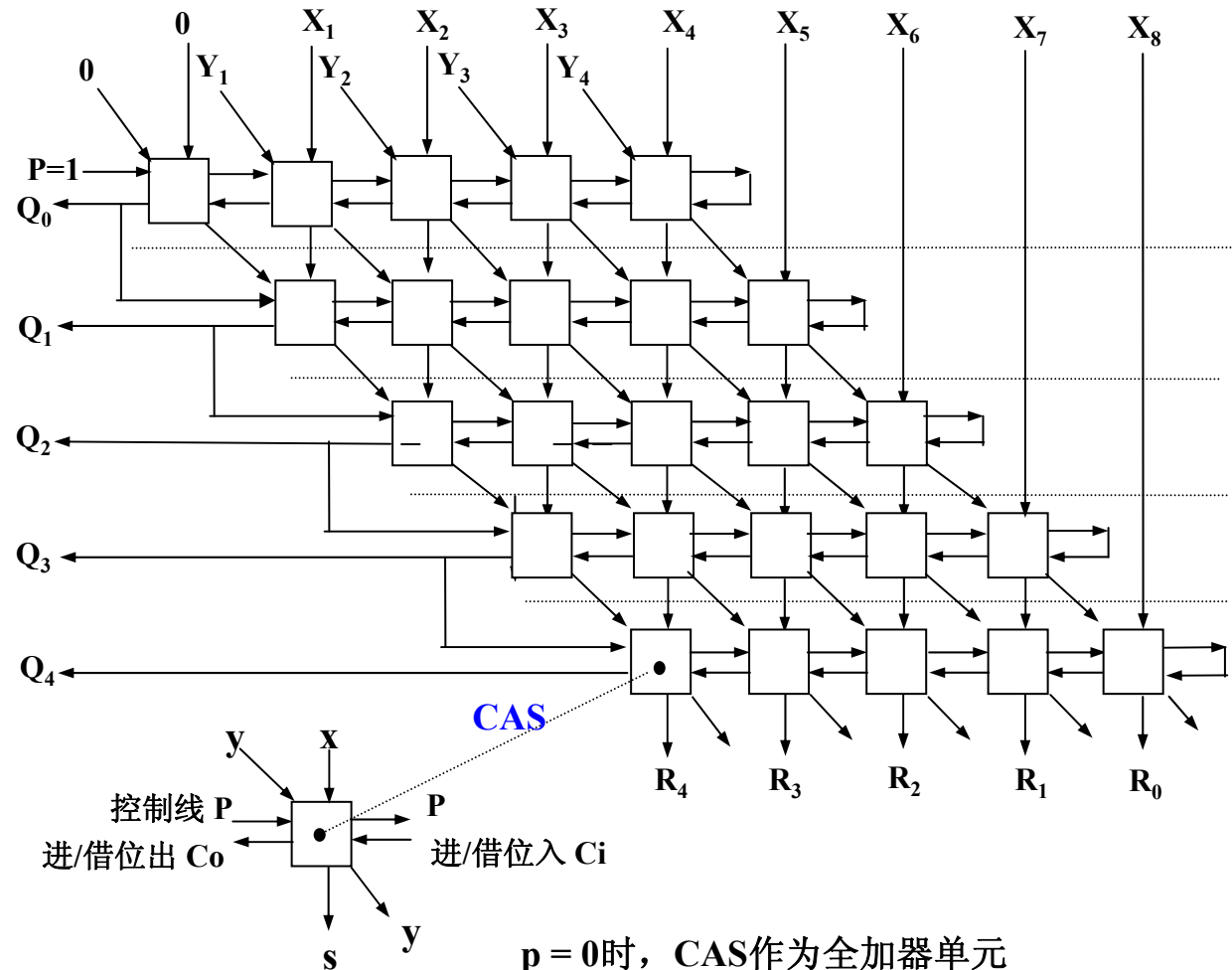
中间行最高位进位 C_0 确定商和下次做加/减，故 C_0 连到 Q_i 和下一行控制线 P

$$s = x \oplus (p \oplus y) \oplus c_i$$

$$C_0 = (x + C_i) (p \oplus y) + x C_i$$

$$p = 0 \text{ 时, } s = x \oplus y \oplus C_i, C_0 = xy + yC_i + xC_i$$

$$p = 1 \text{ 时, } s = x \oplus y \oplus C_i, C_0 = xy + yC_i + xC_i$$



$p = 0$ 时，CAS作为全加器单元

$p = 1$ 时，输入 y 被取反，CAS作为全减器单元

定点运算部件

◆ 综合考虑各类定点运算算法后，发现：

- 所有运算都可通过“加”和“移位”操作实现

以一个或多个**ALU**（或加法器）为核心，加上移位器和存放中间临时结果的若干寄存器，在相应控制逻辑的控制下，可以实现各种运算。

◆ 运算部件通常指**ALU**、移位器、寄存器组，加上用于数据选择的多路选择器和实现数据传送的总线等构成的一个运算数据通路。

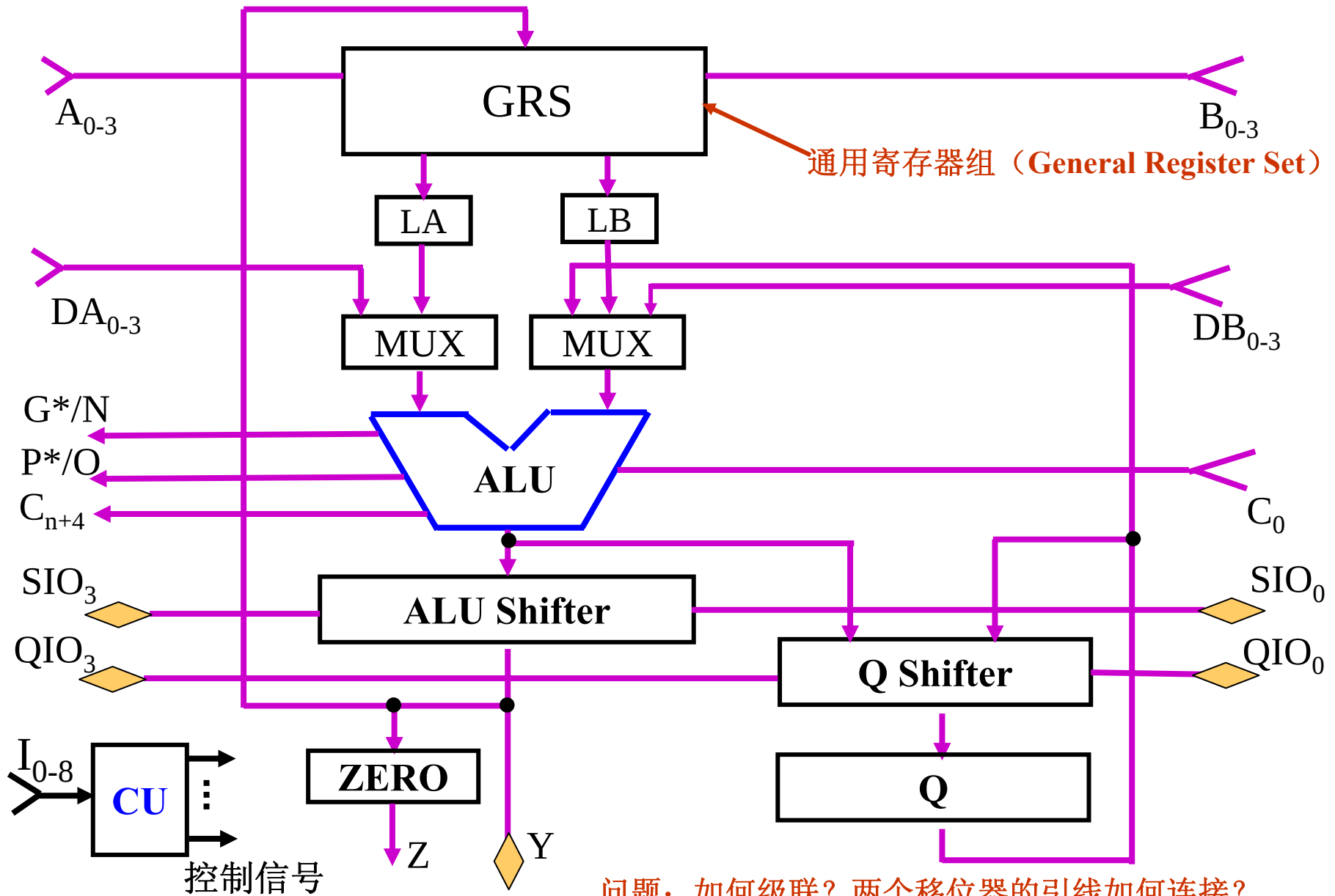
- 可用专门运算器芯片实现（如：**4位运算器芯片AM2901**）
- 可用若干芯片级联实现（如**4个AM2901构成16位运算器**）
- 现代计算机把运算数据通路和控制器都做在**CPU**中，为实现高级流水线，**CPU**中有多个运算部件，通常称为“功能部件”或“执行部件”。

注：

“运算器（Operate Unit）”、“运算部件（Operate Unit）”、“功能部件（Function Unit）”、“执行部件（Execution Unit）”和“数据通路（DataPath）”的含义基本上一样，只是强调的侧面不同。

SKIP

定点运算器芯片举例-AM2901A芯片框图



问题：如何级联？两个移位器的引线如何连接？

AM2901A的功能和结构

- ◆ 核心是4位ALU(含移位器), 可实现 $A+B$ 、 $A-B$ 、 $B-A$ 和“与”、“或”、“异或”等
 - 进位输入 C_0 、进位输出 C_{n+4} 、组进位传递/溢出 P^*/O 、组进位生成/符号 G^*/N
 - 串行级联时, C_0 和 C_{n+4} 用来串行进位传递
 - 多级级联时, 后两个信号用作组进位传递信号 P^* 和组进位生成信号 G^*
 - ALU的操作数来自主存或寄存器, B 输入端还可以是 Q 寄存器
- ◆ 4位双口GRS(16个), 一个写入口、两个读口 A 和 B
 - A_0-A_3 为读口 A 的编号, B_0-B_3 为写口或读口 B 的编号
 - A 和 B 口可同时读出, 分别 LA 和 LB 送到多路选择器MUX的输入端
- ◆ 一个 Q 寄存器和 Q 移位寄存器, 主要用于实现乘/除运算
 - 乘法的部分积和除法的中间余数都是双倍字长, 需放到两个单倍字长寄存器中, 并对其同时串行左移(除法)或右移(乘法)
 - Q 寄存器就是乘数寄存器或商寄存器。因此, 也被称为 Q 乘商寄存器
 - ALU移位器和 Q 移位器一起进行左移或右移, 移位后, ALU移位器内容送ALU继续进行下次运算, 而 Q 移位器内容送 Q 乘商寄存器。
- ◆ 将ALU的结果进行判“0”后可通过 Z 输出端将“零”标志信息输出,
- ◆ ALU、MUX、移位器等的控制信号来自CCU, 通过对指令操作码 I_0-I_8 译码, 得到控制信号
- ◆ 思考题: 如何用AM2901A芯片实现乘/除运算?
 - ① R_0 被预置为0/被除数高位, Q 寄存器被预置为0/被除数低位, R_1 被预置为被乘数/除数
 - ② 控制ALU执行“加”或“减”, 并使ALU移位器和 Q 移位器同时“右移”/“左移”
(移位后ALU移位器中是高位部分积/中间余数, Q 移位器中是低位部分积/中间余数)
 - ③ ALU移位器送ALU的 A 端, Q 移位器送 Q 寄存器后, 再送回 Q 移位器
 - ④ 反复执行第②、③两个步骤, 直到得到所有乘积位或商

BACK

MIPS中的乘、除运算处理

- ◆ MIPS中有一对32位寄存器Hi & Lo （相当于Q乘商寄存器）
- ◆ 乘法和除法运算的硬件相同：
 - 仅需做加、减和64位寄存器的左/右移位
 - Hi和Lo结合起来实现64位寄存器
 - 乘法 Hi中存放高32位积， Lo中存放低32位积
 - 除法： Hi中存放remainder， Lo中存放 quotient
- ◆ mflo指令用来把Lo中的32位数据取到通用寄存器
- ◆ 指令: mult , multu; div , divu
- ◆ 两种乘法指令都忽略overflow, 而由软件自行处理溢出
 - 软件通过mfhi指令取出Hi寄存器来判断是否溢出
 - 溢出判断规则： Hi中为以下数值时，不溢出，否则溢出
 - 无符号数乘指令（multu）时：全0
 - 带符号数乘（mult）时： Lo中的符号
- ◆ MIPS指令不处理“溢出”和“除数为0”，由软件自行处理

十进制数的加减运算

- ◆ 有的机器有十进制加减法指令，用于对**BCD**码进行加减运算。
所以这些机器中必须要有相应的十进制加减运算逻辑。
- ◆ 以**NBCD**码（**8421**码）为例，讨论十进制整数的加减运算。
- ◆ 一般规定数符在最高位 **1100**：正，**1101**：负
或 **0**：正， **1**：负

例如：+2039 **1100** 0010 0000 0011 1001

或 **0** 0010 0000 0011 1001

-1265 **1101** 0001 0010 0110 0101

1 0001 0010 0110 0101

十进制加法运算举例

例1 $25+31=56$

```
  0010 0101
+ 0011 0001
-----
  0101 0110
```

例2 $25+39=64$

```
  0010 0101
+ 0011 1001
-----
  0101 1110
    0110
-----
  0110 0100
```

$(1110)_2 > 9$
需“+6”校正

例3 $27+39=66$

```
  0010 0111
+ 0011 1001
-----
  0101 0000
    1 0110
-----
  0110 0110
```

低位有进位，则
进到高位，同时
该低位“+6”校正

- ◆ 结果 ≤ 9 时，不需校正；大于9或向高位有进位时，需“+6”校正
- ◆ 当最高位有进位时，发生溢出

问题：本位和在什么范围内需“+6”校正？

大于9的情况：1010, 1011, ..., 1111

有进位的情况：10000, 10001, 10010和10011(19)

最大为19: $2 \times 9 + 1 = 19$, 范围为10~19

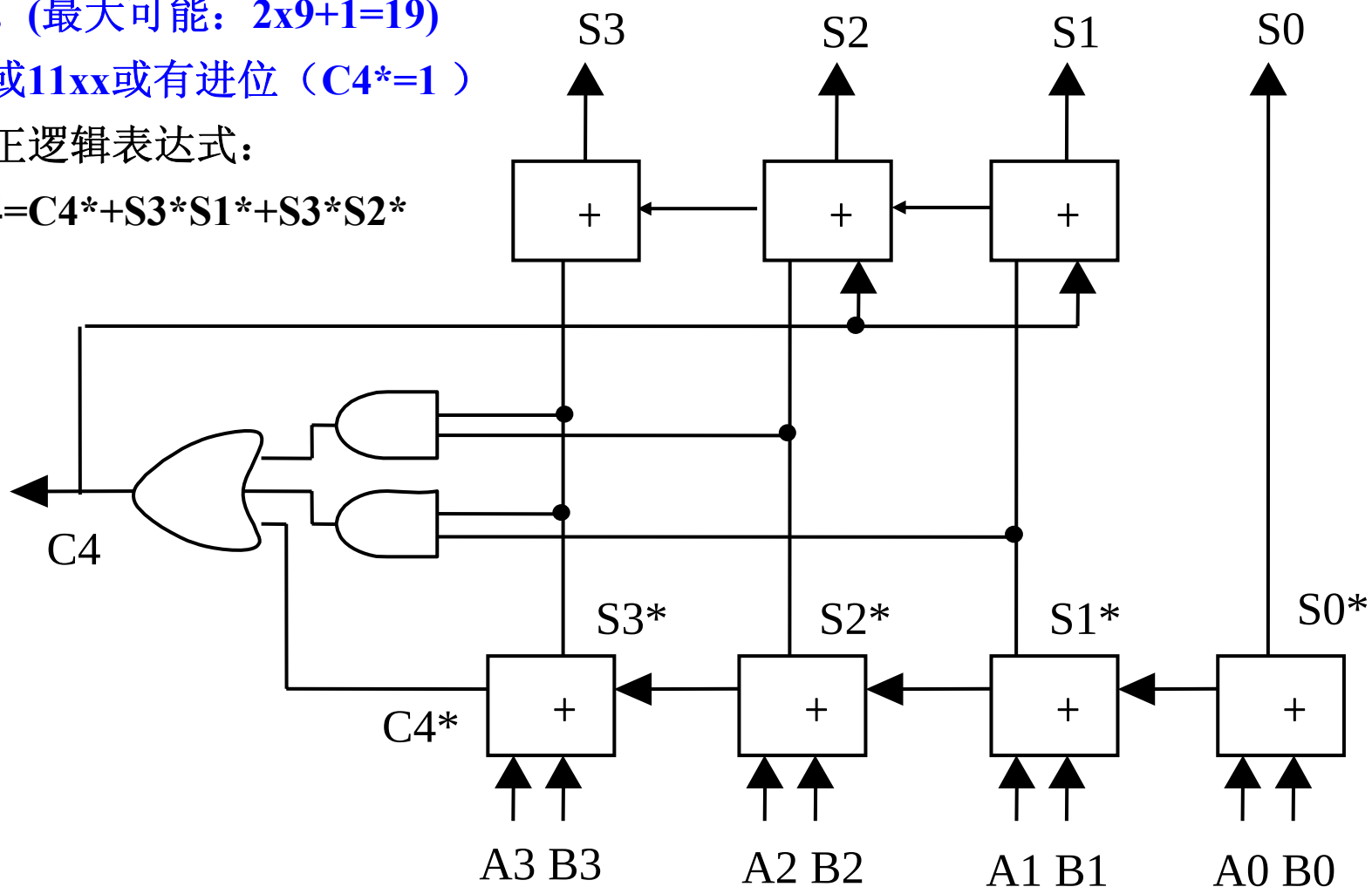
一位十进制加法器

当某位运算结果在10~19之间时，需进行校正。(最大可能： $2 \times 9 + 1 = 19$)

即：1x1x或11xx或有进位 ($C4^*=1$)

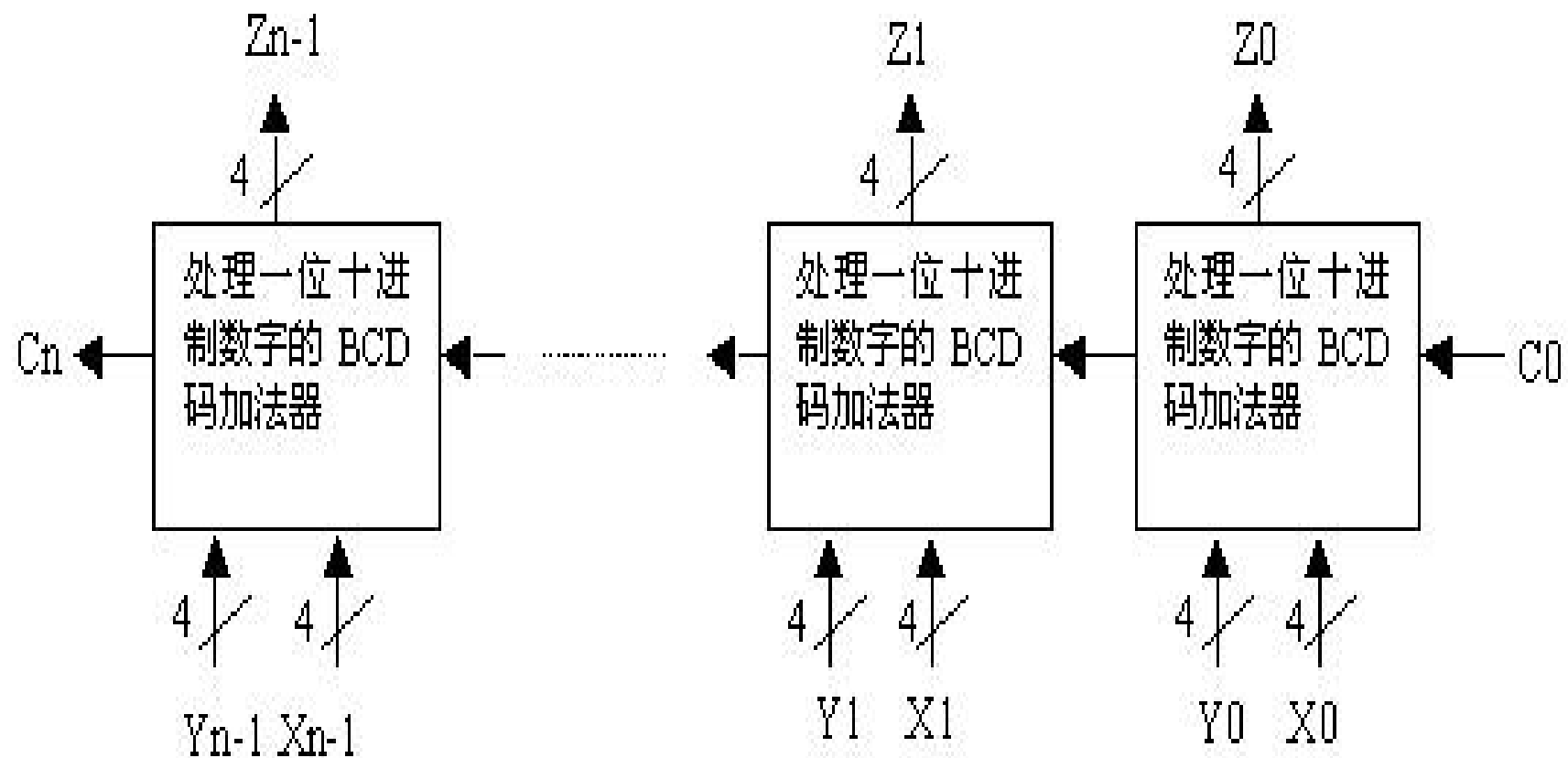
所以，校正逻辑表达式：

$$C4 = C4^* + S3^*S1^* + S3^*S2^*$$



n位十进制加法器

- ◆ n个一位十进制加法器=
一个n位十进制串行加法器



十进制减法运算

- ◆ 方法：
 - “加补码”： $N_1 - N_2 = N_1 + (10^n - N_2) \pmod{10^n}$
- ◆ 十进制数的补码求法：
 - 每位求反，末位加“1”
- ◆ 一位十进制数（**NBCD**码）求反的方法，有：
 - 对各二进位求反，再“+10”
 - 先“+6”，再各位求反
 - 直接用求反电路
- ◆ 只要在加法器基础上增加求补逻辑和最终结果的修正逻辑。

$$\begin{aligned}\text{例: } [7]_{\text{反}} &= \overline{0} \overline{1} \overline{1} \overline{1} + 1010 = 1000 + 1010 = 0010 = 2 \\ &= 0 \ 1 \ 1 \ 1 + 0110 = 1101 = 0010 = 2\end{aligned}$$

十进制减法运算举例

例1 $309-125=184$

加补码: $309+875=184$

例2 $125-309=-184$

$125+691=-184 \pmod{10^3}$

$$\begin{array}{r} 0011\ 0000\ 1001 \\ +1000\ 0111\ 0101 \\ \hline 1011\ 0111\ 1110 \\ 0110\quad\quad 0110 \\ \hline 10001\ 1000\ 0100 \end{array}$$

进位为1，表示
被减数大于减
数，结果为正

$$\begin{array}{r} 0001\ 0010\ 0101 \\ +0110\ 1001\ 0001 \\ \hline 0111\ 1011\ 0110 \\ 0110 \\ \hline 1000\ 0001\ 0110 \end{array}$$

无进位，表示差
值为负数，故应
将结果取补

取补

$$\downarrow$$
$$-0001\ 1000\ 0100$$

第二讲小结

- ◆ 逻辑运算、移位运算、扩展运算等电路简单，主要考虑算术运算
- ◆ 定点运算涉及的对象
无符号数；带符号整数(补码)；定点原码小数；定点移码整数
- ◆ 定点数运算：**ALU**实现基本算术和逻辑运算，**ALU+移位器**实现其他运算
加减运算：
 - 补码加/减：符号位和数值位一起运算，减法用加法实现。同号相加时可能溢出
 - 原码加/减：符号位和数值位分开运算，用于浮点数尾数加/减运算
 - 移码加减：移码的和、差等于和、差的补码，用于浮点数阶码加/减运算乘法运算：
 - 无符号数乘法：“加”+“右移”
 - 原码（一位/两位）乘法：符号和数值分开运算，数值部分用无符号数乘法实现，用于浮点数尾数乘法运算。
 - 补码（一位/两位）乘法：符号和数值一起运算，采用**Booth**算法。
 - 快速乘法器：流水化乘法器、阵列乘法器除法运算：
 - 无符号数除法：用“加/减”+“左移”，有恢复余数和不恢复余数两种。
 - 原码除法：符号和数值分开，数值部分用无符号数除法实现，用于浮点数尾数除法运算。
 - 补码除法：符号位和数值位一起。有恢复余数和不恢复余数两种。
 - 快速除法器：很难实现流水化除法器，可实现阵列除法器
- ◆ 定点运算部件：**ALU**、**GRS**、**MUX**、**Shifter**、**Q**寄存器等，在**CU**控制下执行
- ◆ 十进制数加、减运算及运算部件

第三讲：浮点数运算

主 要 内 容

- ◆ 指令集中与浮点运算相关的指令（以**MIPS**为参考）
 - 涉及到的操作数
 - 单精度浮点数
 - 双精度浮点数
 - 涉及到的运算
 - 算术运算： 加 / 减 / 乘 / 除
- ◆ 浮点数加减运算
- ◆ 浮点数乘除运算
- ◆ 浮点数运算的精度问题

MIPS浮点运算指令的总结

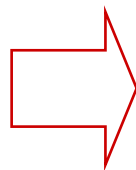
- ◆ 浮点操作数的表示
 - 32位单精度浮点数 / 64位双精度浮点数
- ◆ 浮点数的运算
 - 加法 / 减法 / 乘法 / 除法

问题：IA-32中浮点数寄存器是80位，这会给float和double类型变量的运算带来什么隐患？

例子：将以下程序编译为MIPS汇编语言

```
Float f2c (float fahr)
{
    return ((5.0 / 9.0) * (fahr-32.0));
}
```

假设变量fahr存放在\$**f12**中，
返回结果存放在\$**f0**中。
三个常数存放在通过\$**gp**能访问到的存储单元中。



假设不考虑信息在栈帧中的保存和恢复

```
f2c : lwcl    $f16, const5($gp)
      lwcl    $f18, const9($gp)
      div.s   $f16, $f16, $f18
      lwcl    $f18, const32($gp)
      sub.s   $f12, $f12, $f18
      mul.s   $f0, $f16, $f12
      jr      $ra
```

有关Floating-point number的问题

实现一套浮点数运算指令，要解决的问题有：

Issues:

- **Representation(表示):**
Normalized form (规格化形式) 和 Denormalized form
单精度格式 和 双精度格式
- **Range and Precision(表数范围和精度)**
- **Arithmetic (+, -, *, /)**
- **Rounding(舍入)**
- **Exceptions (e.g., divide by zero, overflow, underflow)**
(异常处理：如除数为0，上溢，下溢等)
- **Errors(误差)与精度控制**

浮点数运算及结果

设两个规格化浮点数分别为 $A = M_a \cdot 2^{E_a}$ $B = M_b \cdot 2^{E_b}$,则:

$$A \pm B = (M_a \pm M_b \cdot 2^{-(E_a - E_b)}) \cdot 2^{E_a} \quad (\text{假设 } E_a \geq E_b)$$

$$A * B = (M_a * M_b) \cdot 2^{E_a + E_b}$$

$$A / B = (M_a / M_b) \cdot 2^{E_a - E_b}$$

上述运算结果可能出现以下几种情况:

阶码上溢: 一个正指数超过了最大允许值 $\Rightarrow +\infty / -\infty$ / 溢出 SP最大允许的指数为多少? 127!

阶码下溢: 一个负指数超过了最小允许值 $\Rightarrow +0 / -0$ SP最小允许的指数为多少?

尾数上溢: 最高有效位有进位 $\Rightarrow$ 右规 -126!

非规格化尾数: 数值部分高位为0 $\Rightarrow$ 左规 尾数溢出, 不一定浮点数溢出, 即不一定会发生“溢出异常”

右规或对阶时, 右段有效位丢失 $\Rightarrow$ 尾数舍入 在运算过程中, 添加保护位

IEEE建议实现时为每种异常情况提供一个 自陷允许位。若某异常对应的位为1, 则发生相应异常时, 就调用一个特定的异常处理程序执行。

IEEE754标准规定的五种异常情况

① 无效操作（无意义）

- 操作中有一个数是非有限数，如：

加 / 减 ∞ 、 $0 \times \infty$ 、 ∞ / ∞ 等

- 结果无效，如：

源操作数是NaN、 $0/0$ 、 $x \text{ REM } 0$ 、 $\infty \text{ REM } y$ 等

② 除以0（即：无穷大）

③ 数太大（阶码上溢）：对于SP，指阶码 $E > 1111\ 1110$ （指数大于127）

④ 数太小（阶码下溢）：对于SP，指阶码 $E < 0000\ 0001$ （指数小于-126）

⑤ 结果不精确（舍入时引起），例如1/3，1/10等不能精确表示成浮点数

上述情况硬件可以捕捉到，因此这些异常可设定让硬件处理，也可设定让软件处理。

让硬件处理时，称为硬件陷阱。

注：硬件陷阱可以理解成：事先设定好是否要进行硬件处理（即挖一个陷阱），当出现相应的异常情况时，就由硬件自动地进行对应的异常处理（掉入陷阱）。

浮点数加/减运算

◆ 十进制科学计数法的加法例子

$$0.123 \times 10^5 + 0.456 \times 10^2$$

其计算过程为：

$$\begin{aligned} 0.123 \times 10^5 + 0.456 \times 10^2 &= 0.123 \times 10^5 + 0.000456 \times 10^5 \\ &= (0.123 + 0.000456) \times 10^5 = 0.123456 \times 10^5 \end{aligned}$$

因此，进行尾数加减运算前，必须“对阶”！最后还要考虑舍入
计算机内部的二进制运算也一样！

◆ “对阶”操作：目的是使两数阶码相等

- 小阶向大阶看齐，阶小的那个数的尾数右移，右移位数等于两个阶码差的绝对值
- IEEE754尾数右移时，要将隐含的“1”移到小数部分，空出位补0，移出的低位保留到特定的“附加位”上

通过计算 $[\Delta E]_{\text{补}}$ 来判断两数的阶差，

$$[\Delta E]_{\text{补}} = [E_x - E_y]_{\text{补}} = [E_x]_{\text{移}} + [-[E_y]_{\text{移}}]_{\text{补}} \pmod{2^n}$$

问题：在 ΔE 为何值时无法
根据 $[\Delta E]_{\text{补}}$ 来判断阶差？

浮点数加减法基本要点

浮点数加 / 减法步骤

(假定: X_m 、 Y_m 分别是 X 和 Y 的尾数, X_e 和 Y_e 分别是 X 和 Y 的阶码)

- (1) 求阶差: $\Delta e = Y_e - X_e$ (假定 $Y_e > X_e$, 则结果的阶码为 Y_e)
- (2) 对阶: 将 X_m 右移 Δe 位, 尾数变为 $X_m 2^{X_e - Y_e}$ (保留右移的尾数部分: 附加位)
- (3) 尾数加减: $X_m 2^{X_e - Y_e} \pm Y_m$
- (4) 规格化:
 - 当尾数高位为0, 则需左规: 尾数左移一次, 阶码减1, 直到MSB为1
 - 每次阶码减1后要判断阶码是否下溢 (比最小可表示的阶码还要小)
 - 当尾数最高位有进位, 需右规: 尾数右移一次, 阶码加1, 直到MSB为1
 - 每次阶码加1后要判断阶码是否上溢 (比最大可表示的阶码还要大)
 - 阶码溢出异常处理: 阶码上溢, 则结果溢出; 阶码下溢, 则结果为0
- (5) 如果尾数比规定位数长, 则需考虑舍入 (有多种舍入方式)
- (6) 若尾数是0, 则需要将指数也置0。为什么?

尾数为0说明结果应该为0, 即: 指数和尾数为全0。

浮点数加法运算举例

Example: 用二进制形式计算 $0.5 + (-0.4375) = ?$

解: $0.5 = 1.000 \times 2^{-1}$, $-0.4375 = -1.110 \times 2^{-2}$

对 阶: $-1.110 \times 2^{-2} \rightarrow -0.111 \times 2^{-1}$

加 减: $1.000 \times 2^{-1} + (-0.111 \times 2^{-1})$
 $= 0.001 \times 2^{-1}$

规格化: $0.001 \times 2^{-1} \rightarrow 1.000 \times 2^{-4}$

判溢出: 无

结果为: $1.000 \times 2^{-4} = 0.0001000 = 1/16 = 0.0625$

在计算机内部执行上述运算时, 必须解决哪些问题?

(1) 如何表示? 用IEEE754标准!

(2) 如何判断阶码的大小? 求 $[\Delta E]_{\text{补}} = ?$ $[\Delta E]_{\text{补}} = [E_x - E_y]_{\text{补}} = [E_x]_{\text{移}} + [-[E_y]_{\text{移}}]_{\text{补}} \pmod{2^n}$

(3) 对阶后尾数的隐含位如何处理? 右移到数值部分, 高位补0, 保留移出低位部分

(4) 如何进行尾数加减? 隐藏位还原后, 按原码进行加减运算, 附加位一起运算

(5) 何时需要规格化, 如何规格化? $\pm 1x.xx\dots x$ 形式时, 则右规: 尾数右移1位, 阶码加1
 $\pm 0.0\dots 01x\dots x$ 形式时, 则左规: 尾数左移k位, 阶码减k

(6) 如何舍入? 最终须把附加位去掉, 此时需考虑舍入 (IEEE754有四种舍入方式)

(7) 如何判断溢出? 若最终阶码大于127 (即: 阶码为全1) 则上溢; 若尾数为全0, 则下溢

浮点数加法运算举例

例子（同前） $x=0.5$ $y=-0.4375$ 求 $x+y=?$

解2: 假定用IEEE754标准单精度格式表示

$$x=0.5=1/2=(0.100...0)_2=(1.00...0)_2 \times 2^{-1}$$

$$y=-0.4325=(-0.01110...0)_2=(-1.110..0)_2 \times 2^{-2}$$

$$[x]_{\text{浮}}=0 \ 01111110, 00...0 \quad [y]_{\text{浮}}=1 \ 01111101, 110...0$$

$$\text{对阶: } [\Delta E]_{\text{补}}=0111 \ 1110 + 1000 \ 0011=0000 \ 0001 \quad \Delta E=1$$

$$[\Delta E]_{\text{补}}=256+E_x-E_y=256+127+E_x-(127+E_y)=256+[E_x]_{\text{移}}-[E_y]_{\text{移}}=[E_x]_{\text{移}}+[-[E_y]_{\text{移}}]_{\text{补}} \pmod{256}$$

故对 y 进行对阶 $[y]_{\text{浮}}=1 \ 0111 \ 1110 \ 1110...0$ (高位补隐藏位)

尾数相加: $01.0000...0+(10.1110...0)=00.00100...0$ (原码加法, 最左边一位为符号)

左规: $+(0.00100...0)_2 \times 2^{-1}=+(1.00...0)_2 \times 2^{-4}$ (阶码减3, 实际上是加了三次11111111)

$$[x+y]_{\text{浮}}=0 \ 0111 \ 1011 \ 00...0$$

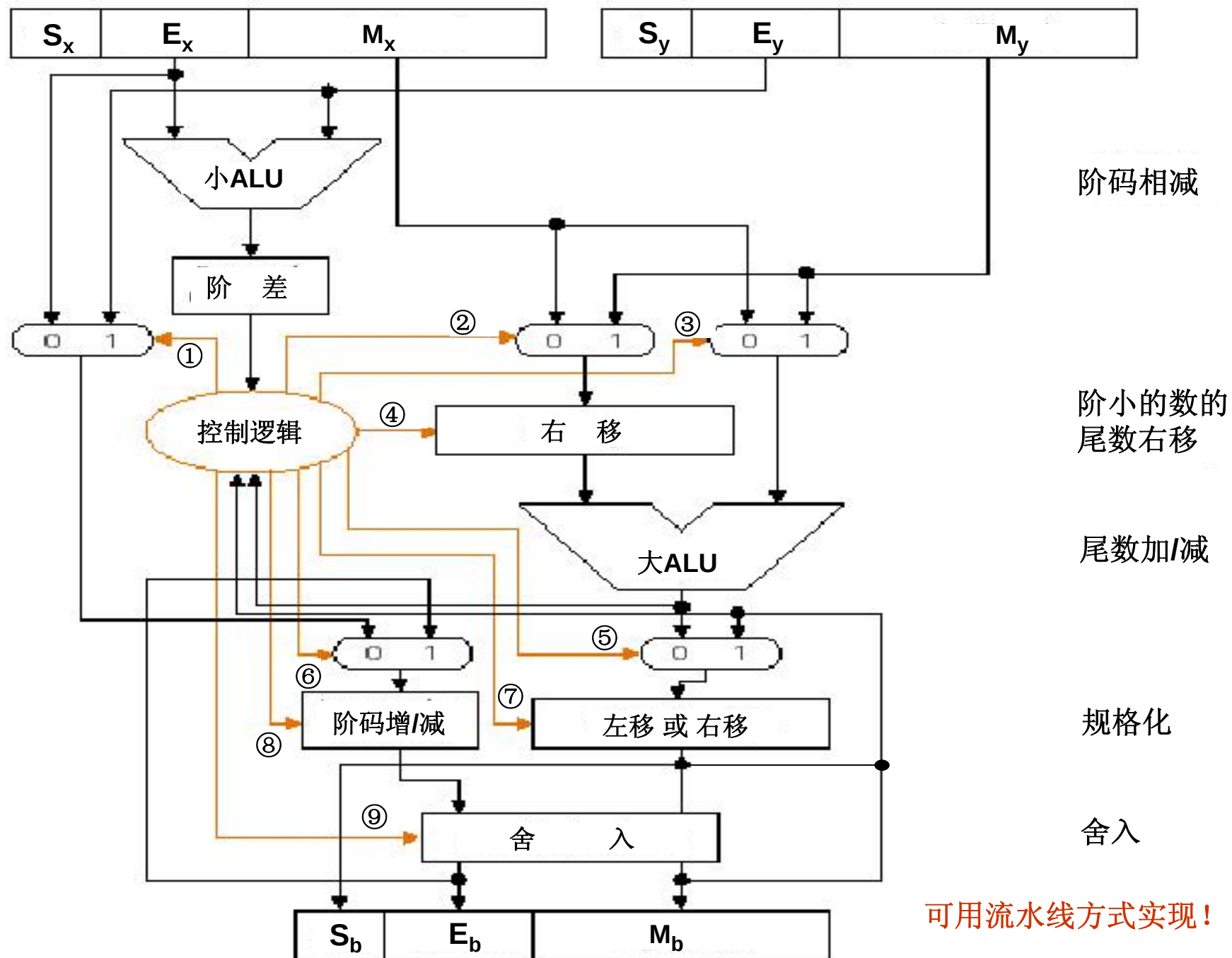
$$x+y=(1.0)_2 \times 2^{-4}=1/16=0.0625$$

即: $0111 \ 1110 \ (-1)$ 加3次 $[-1]_{\text{补}}$

$$\begin{aligned} & ((0111 \ 1110+11111111)+11111111)+11111111 \\ & =0111 \ 1011 \ (-4) \end{aligned}$$

问题: 为何加减运算右规时最多只需一次?

因为即使是两个最大的尾数相加, 得到的和的尾数也不会达到4, 故尾数的整数部分最多有两位, 保留一个隐含的“1”后, 最多只有一位被右移到小数部分。



可用流水线方式实现!

浮点数乘/除法基本要点

◆ 浮点数乘法: $A*B=(M_a * M_b) \cdot 2^{E_a+E_b}$

◆ 浮点数除法: $A/B=(M_a / M_b) \cdot 2^{E_a-E_b}$

浮点数尾数采用原码乘、除运算。

浮点数乘 / 除法步骤

(假定: X_m 、 Y_m 分别是X和Y的尾数, X_e 和 Y_e 分别是X和Y的阶码)

(1) 求阶: $X_e \pm Y_e + 127$

(2) 尾数相乘除: $X_m * / Y_m$ (两个形为1.xxx的数相乘/除)

(3) 两数符号相同, 结果为正; 两数符号相异, 结果为负;

如果需要规格化 (尾数形如: 1.xx...x), 则按 (4) 进行。

(4) 当尾数高位为0, 需左规: 尾数左移一次, 阶码减1 ($+[-1]_{补}$), 直到MSB为1。

每次阶码减1后要判断阶码是否下溢 (比最小可表示的阶码还要小)

当尾数最高位有进位, 需右规: 尾数右移一次, 阶码加1, 直到MSB为1。

每次阶码加1后要判断阶码是否上溢 (比最大可表示的阶码还要大)

(5) 如果尾数比规定的长, 则需考虑舍入。

(6) 若尾数是0, 则需要将指数也置0。

(7) 阶码溢出判断

问题1: 乘法运算结果最多左规几次? 最多右规几次? 不需左规! 最多右规1次!

问题2: 除法呢? 左规次数不定! 不需右规!

求阶码的和、差

◆ 假设**Ex**和**Ey**分别是两个操作数的阶码，**Eb**是结果的阶码，则：

- 阶码加法运算公式为： $Eb \leftarrow Ex + Ey + 129 \pmod{2^8}$

$$\begin{aligned}[E1 + E2]_{\text{移}} &= 127 + E1 + E2 = 127 + E1 + 127 + E2 - 127 \\ &= [E1]_{\text{移}} + [E2]_{\text{移}} - 127 \\ &= [E1]_{\text{移}} + [E2]_{\text{移}} + [-127]_{\text{补}} \\ &= [E1]_{\text{移}} + [E2]_{\text{移}} + 10000001B \pmod{2^8}\end{aligned}$$

- 阶码减法运算公式为： $Eb \leftarrow Ex + [-Ey]_{\text{补}} + 127 \pmod{2^8}$

$$\begin{aligned}[E1 - E2]_{\text{移}} &= 127 + E1 - E2 = 127 + E1 - (127 + E2) + 127 \\ &= [E1]_{\text{移}} - [E2]_{\text{移}} + 127 \\ &= [E1]_{\text{移}} + [-[E2]_{\text{移}}]_{\text{补}} + 01111111B \pmod{2^8}\end{aligned}$$

举例：若两个IEEE754操作数的阶码分别为10和-5，求10+(-5)和10-(-5)的移码。

解： $E_x = 127 + 10 = 137 = 1000\ 1001B$ ， $E_y = 127 + (-5) = 122 = 0111\ 1010B$ ， $[-E_y]_{\text{补}} = 1000\ 0110B$

将 E_x 和 E_y 代入上述公式，得：

$$E_b = E_x + E_y + 129 = 1000\ 1001 + 0111\ 1010 + 1000\ 0001 \pmod{2^8} = 1000\ 0100B = 132$$

其阶码的和为 $132 - 127 = 5$ ，正好等于 $10 + (-5) = 5$ 。

$$E_b = E_x + [-E_y]_{\text{补}} + 127 = 1000\ 1001 + 1000\ 0110 + 0111\ 1111 \pmod{2^8} = 1000\ 1110B = 142,$$

其阶码的差为 $142 - 127 = 15$ ，正好等于 $10 - (-5) = 15$ 。

[BACK](#)

Extra Bits(附加位)

"Floating Point numbers are like piles of sand; every time you move one you lose a little sand, but you pick up a little dirt."

“浮点数就像一堆沙，每动一次就会失去一点沙，并捡回一点脏”

有谁能解释上述这段话的含义？

如何才能使失去的“沙”和捡回的“脏”都尽量少呢？

加多少附加位才合适？

Addition:

	1.xxxxx	1.xxxxx	1.xxxxx	1.xxxxxxxxx
+	1.xxxxx	0.001xxxx	0.01xxxx	-1.xxxxxxxxx
	1x.xxxx y	1.xxxxx yy	1x.xxxx yy	0.0...0xxxx

IEEE754规定：中间结果须在右边加2个附加位（guard & round）

Guard bit(保护位)：在significand右边的位

Rounding bit(舍入位)：在保护位右边的位

附加位的作用：用以保护对阶时右移的位或运算的中间结果。

附加位的处理：①左规时被移到significand中；②作为舍入的依据。

[BACK](#)

Rounding Digits(舍入位)

举例: $B = 10, p = 3$

假定采用两位附加位

思考: 若没有舍入位, 采用就近舍入到偶数, 则结果是什么?

结果为2.36! 精度没有2.37高!

$$\begin{array}{|c|c|c|} \hline 0 & 2 & 2.34 \\ \hline \end{array} = 2.3400 * 10^2$$

$$\begin{array}{|c|c|c|} \hline 0 & 0 & 2.53 \\ \hline \end{array} = \underline{0.0253} * 10^2$$

$$\begin{array}{|c|c|c|} \hline 0 & 2 & 2.37 \\ \hline \end{array} = 2.3653 * 10^2$$

IEEE Standard:

four rounding modes (用图说明):

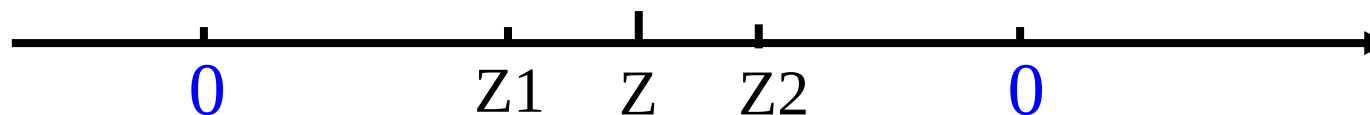


注: **ULP=units in the last place.**

可以证明默认方式得到的平均误差最小。

IEEE754的舍入方式的说明

IEEE754的舍入方式



(Z1和Z2分别是结果Z的最近可表示的左、右数)

(1) 就近舍入：舍入为最近可表示的数

(非中间值：0舍1入；

中间值：强迫结果为偶数-慢)

如：附加位为

01：舍

11：入

10：(强迫结果为偶数)

例：1.1101**11** ~ 1.1110; 1.1101**01** ~ 1.1101; 1.1101**10** ~ 1.1110; 1.1111**10** ~ 10.0000;

(2) 朝 $+\infty$ 方向舍入：舍入为Z2(正向舍入)

(3) 朝 $-\infty$ 方向舍入：舍入为Z1(负向舍入)

(4) 朝0方向舍入：截去。即，总是舍入成Z1与Z2中绝对值较小的那个(正数：取Z1；负数：取Z2)

IEEE754通过在舍入位后再引入“粘位**sticky bit**”来简化“中间值”舍入问题。

加减运算对阶时，较小数的尾数右移后，舍入位之后有非0数，则可设置**sticky bit**。

举例：1.24 $\times 10^4$ +5.09 $\times 10^1$ 分别采用一位、二位、三位附加位时，结果各是多少？（就近舍入到偶数）

尾数精确结果为1.24**509**，所以分别为：

1.24, **1.24**, **1.25**

[BACK](#)

溢出判断

◆ 可能会导致阶码溢出的情况:

问题: 如何减1? $+[-1]_{\text{补}}$

- 左规（阶码-1）、右规（阶码+1）时

- 左规（-1）时：先判断阶码是否为全0，若是，则直接置阶码下溢；否则，阶码减1后判断阶码是否为全0，若是，则阶码下溢。
- 右规（+1）时，先判断阶码是否为全1，若是，则直接置阶码上溢；否则，阶码加1后判断阶码是否为全1，若是，则阶码上溢。

- 乘法运算求阶码的和时

- 若 E_x 、 E_y 的最高位都是1，而 E_b 的最高位是0或 E_b 为全1，则阶码上溢。
- 若 E_x 、 E_y 的最高位都是0，而 E_b 的最高位是1或 E_b 为全0，则阶码下溢。

- 除法运算求阶码的差时

- 若 E_x 的最高位是1， E_y 的最高位是0， E_b 的最高位是0或 E_b 为全1，则阶码上溢。
- 若 E_x 的最高位是0， E_y 的最高位是1， E_b 的最高位是1或 E_b 为全0，则阶码下溢。

例：若 $E_b=0000\ 0001$ ，则左规一次后，结果的阶码 $E_b=?$

解： $E_b=E_b+[-1]_{\text{补}}=0000\ 0001+1111\ 1111=0000\ 0000$ 阶码下溢！

例：若 $E_x=1111\ 1110$ ， $E_y=1000\ 0000$ ，则乘法运算时，结果的阶码 $E_b=?$

解： $E_b=E_x+E_y+129=1111\ 1110+1000\ 0000+1000\ 0001=1111\ 1111$ 阶码上溢！

实例:PowerPC和80x86中的浮点部件

◆ PowerPC中的浮点运算

- 比MIPS多一条浮点指令：乘累加指令

- 将两个操作数相乘，再与另一个操作数相加，写到结果操作数
- 可以用一条乘累加指令代替两条MIPS浮点指令
- 可以为中间结果多保留几位，得到最后结果后，再考虑舍入，精度高
- 利用它来实现除法运算和平方根运算

- 浮点寄存器的数量多一倍（32xSPR, 32xDPR）

◆ 80x86中的浮点运算

- 采用寄存器堆栈结构：栈顶两个数作为操作数
- 寄存器堆栈的精度为80位 (MIPS和PowerPC可以32位或64位)
- 所有浮点运算都转换为80位扩展浮点数进行运算，写回存储器时，再转换位32位（float）或64位（double）。有时会发生很奇怪的问题。
- 浮点数数据存取指令自动完成转换
- 指令类型：存取、算术、比较、函数（正弦、余弦、对数等）

第三讲小结

- ◆ 浮点运算指令（以MIPS为参考）
- ◆ 浮点数的表示（IEEE754标准）
 - 单精度SP（float）和双精度DP（double）
 - 规格化数(SP): 阶码1~254, 尾数最高位隐含为1
 - 0(阶为全0, 尾为全0)
 - ∞ (阶为全1, 尾为全0)
 - NaN(阶为全0, 尾为非0)
 - 非规数(阶为全1, 尾为非0)
- ◆ 浮点数加减运算
 - 对阶、尾数加减、规格化（上溢/下溢处理）、舍入
- ◆ 浮点数乘除运算
 - 求阶、尾数乘除、规格化（上溢/下溢处理）、舍入
- ◆ 浮点数的精度问题
 - 中间结果加保护位、舍入位（和粘位）
 - 最终进行舍入（有四种舍入方式）
 - 最近（中间值强迫为偶数）、 $+\infty$ 方向、 $-\infty$ 方向、0方向
 - 默认为“最近”舍入方式

本章总结（1）

定点数运算：由**ALU** + 移位器实现各种定点运算

- ◆ 移位运算
 - 逻辑移位：对无符号数进行，左（右）边补**0**，低（高）位移出
 - 算术移位：对带符号整数进行，移位前后符号位不变，编码不同，方式不同。
 - 循环移位：最左（右）边位移到最低（高）位，其他位左（右）移一位。
- ◆ 扩展运算
 - 零扩展：对无符号整数进行高位补**0**
 - 符号扩展：对补码整数在高位直接补符
- ◆ 加减运算
 - 补码加/减运算：用于整数加/减运算。符号位和数值位一起运算，减法用加法实现。同号相加时，若结果的符号不同于加数的符号，则会发生溢出。
 - 原码加/减运算：用于浮点数尾数加/减运算。符号位和数值位分开运算，同号相加，异号相减；加法直接加；减法用加负数补码实现。
- ◆ 乘法运算：用加法和右移实现。
 - 补码乘法：用于整数乘法运算。符号位和数值位一起运算。采用**Booth**算法。
 - 原码乘法：用于浮点数尾数乘法运算。符号位和数值位分开运算。数值部分用无符号数乘法实现。
- ◆ 除法运算：用加/减法和左移实现。
 - 补码除法：用于整数除法运算。符号位和数值位一起运算。
 - 原码除法：用于浮点数尾数除法运算。符号位和数值位分开运算。数值部分用无符号数除法实现。

本章总结（2）

- ◆ 浮点数运算：由多个**ALU + 移位器**实现
 - 加减运算
 - 对阶、尾数相加减、规格化处理、舍入、判断溢出
 - 乘除运算
 - 尾数用定点原码乘/除运算实现，阶码用定点数加/减运算实现。
 - 溢出判断
 - 当结果发生阶码上溢时，结果发生溢出，发生阶码下溢时，结果为**0**。
 - 精确表示运算结果
 - 中间结果增设保护位、舍入位、粘位
 - 最终结果舍入方式：就近舍入 / 正向舍入 / 负向舍入 / 截去四种方式。
- ◆ **ALU的实现**
 - 算术逻辑单元**ALU**：实现基本的加减运算和逻辑运算。
 - 加法运算是所有定点和浮点运算（加/减/乘/除）的基础，加法速度至关重要
 - 进位方式是影响加法速度的重要因素
 - 并行进位方式能加快加法速度
 - 通过“进位生成”和“进位传递”函数来使各进位独立、并行产生