
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, 可能溢出!

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。可能溢出!

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

- 类型转换时可能需要数据扩展或截断

• 操作

- 没有专门操作运算符，根据类型转换前后数据长短确定是扩展还是截断
- 扩展：短转长

无符号数：0扩展，前面补0

带符号整数：符号扩展，前面补符

- 截断：长转短

强行将高位丢弃，故可能发生“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 ui = usi ;
```

```
si = -12345    CF C7
```

```
usi = 53191    CF C7
```

```
i = -12345     FF FF CF C7
```

```
ui = 53191     00 00 CF C7
```

例2：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，有效数据丢失，无法被正确表示！

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	$\$s1 = \$s2 \ \ \$s3$
nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim (\$s2 \ \ \$s3)$
and immediate	andi \$s1,\$s2,100	$\$s1 = \$s2 \ \& \ 100$
or immediate	ori \$s1,\$s2,100	$\$s1 = \$s2 \ \ 100$
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：将两个浮点数从内存取出相加后再存到内存的指令序列为：

lwcl \$f1, x(\$s1)

lwcl \$f2, y(\$s2)

add.s \$f4, \$f1, \$f2

swcl \$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

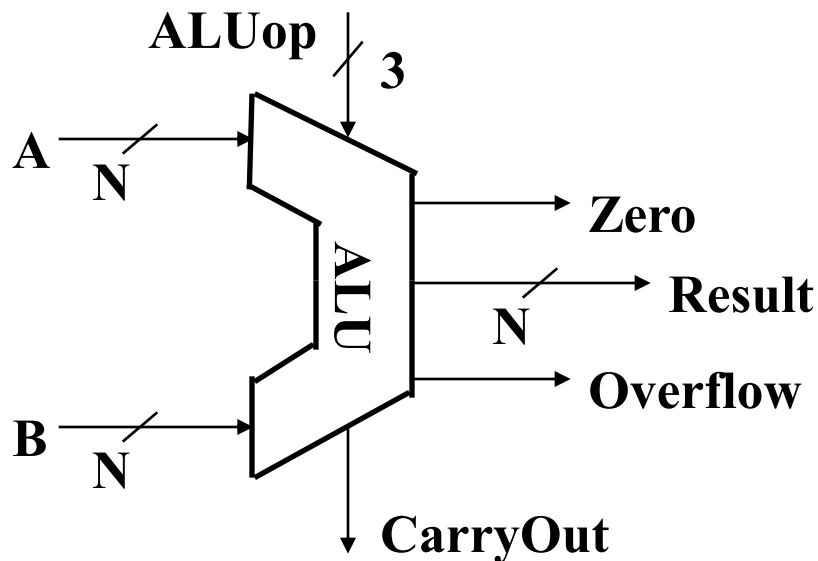
- 定点数运算
 - 带符号整数运算：取负 / 符号扩展 / 加 / 减 / 乘 / 除 / 算术移位
 - 无符号整数运算：0扩展 / 加 / 减 / 乘 / 除
- 逻辑运算
 - 逻辑操作：与 / 或 / 非 / ...
 - 移位操作：逻辑左移 / 逻辑右移
- 浮点数运算：加、减、乘、除

实现MIPS定点运算指令的思路：

先实现一个能进行基本算术运算（加/减）和基本逻辑运算、并生成基本条件码（ZF/OF/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的功能说明



**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。**

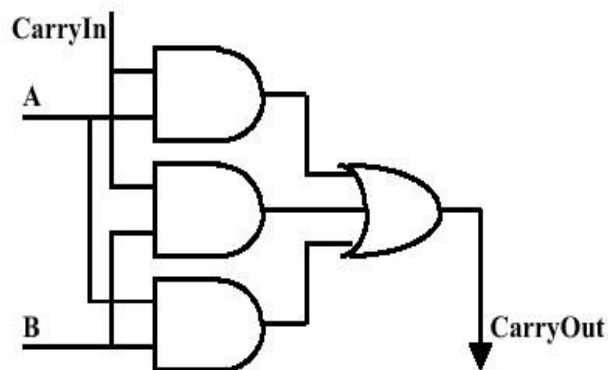
◆ ALU Control Lines (ALUop)	Function
• 000	And
• 001	Or
• 010	Add
• 110	Subtract
• 111	Set-on-less-than

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。

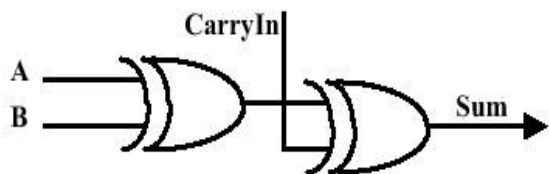
串行进位加法器

CarryOut 和 Sum 的逻辑图

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



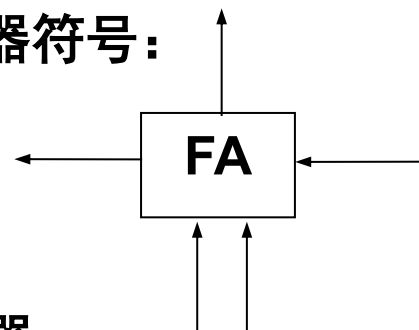
◦ $\text{Sum} = A \text{ XOR } B \text{ XOR } \text{CarryIn}$



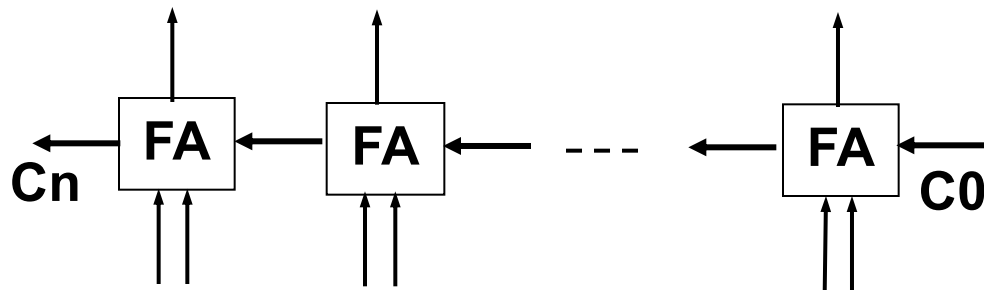
假定与/或门延迟为 $1t_y$ ，异或门 $3t_y$ ，
则“和”与“进位”的延迟为多少？

Sum延迟为 $6t_y$ ；Carryout延迟为 $2t_y$ 。

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 C_0 到 C_n 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！

并行进位加法器（CLA加法器）

◆ 为什么用先行进位方式？

串行进位加法器采用**串行逐级传递进位**，电路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？

定义辅助函数： $G_i = X_i Y_i$...进位生成函数

$P_i = X_i + Y_i$...进位传递函数（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： $S_i = P_i \oplus C_i$ $C_i = G_i + P_i C_{i-1}$ ($i=0, 1, \dots, n$)

设 $n=4$,则： $C_1 = G_1 + P_1 C_0$

$$C_2 = G_2 + P_2 C_1 = G_2 + P_2 G_1 + P_2 P_1 C_0$$

$$C_3 = G_3 + P_3 C_2 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 C_0$$

$$C_4 = G_4 + P_4 C_3 = G_4 + P_4 G_3 + P_4 P_3 G_2 + P_4 P_3 P_2 G_1 + P_4 P_3 P_2 P_1 C_0$$

由上式可知:各进位之间无等待，相互独立并同时产生。

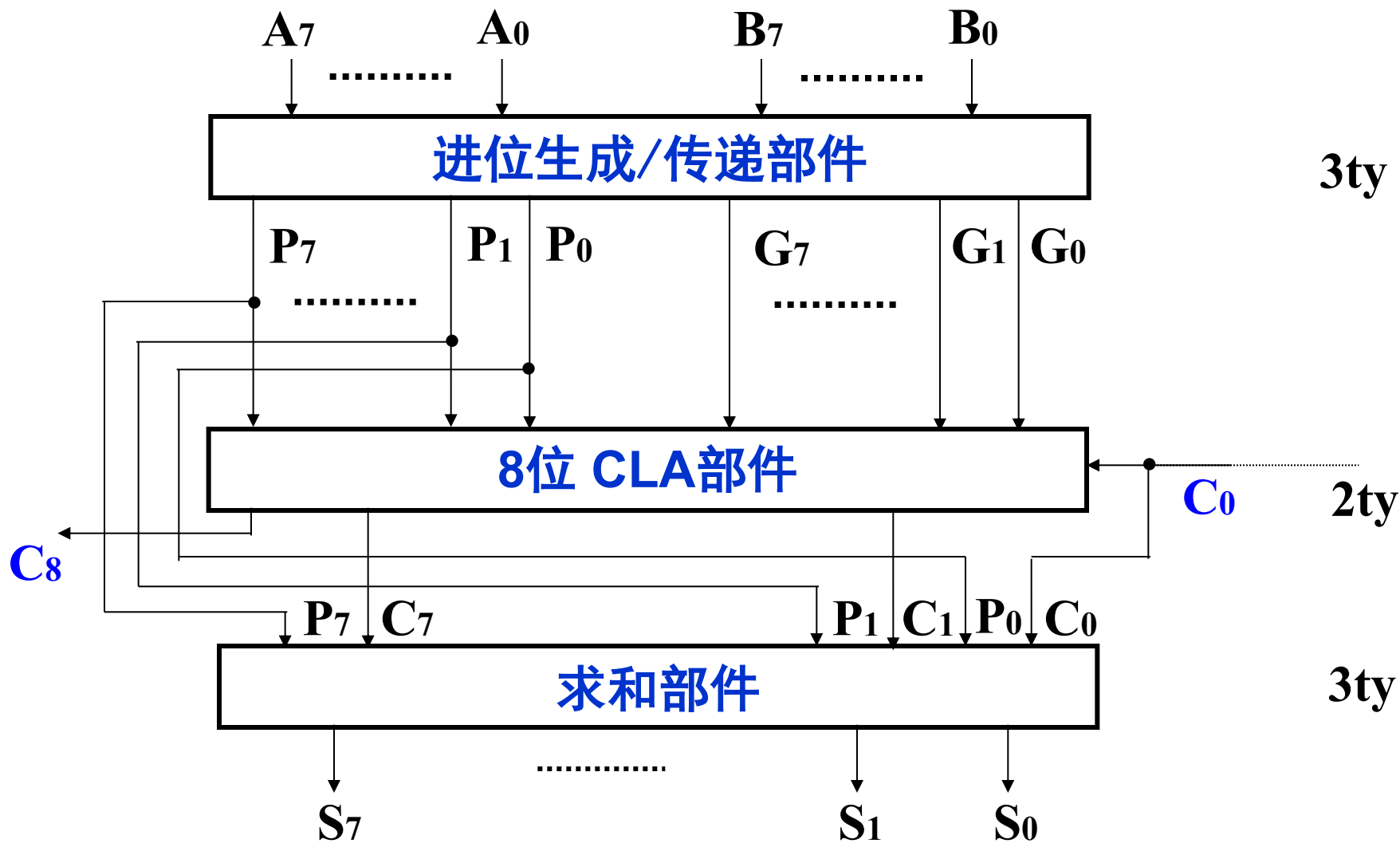
通常把实现上述逻辑的电路称为**4位CLA部件**

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**

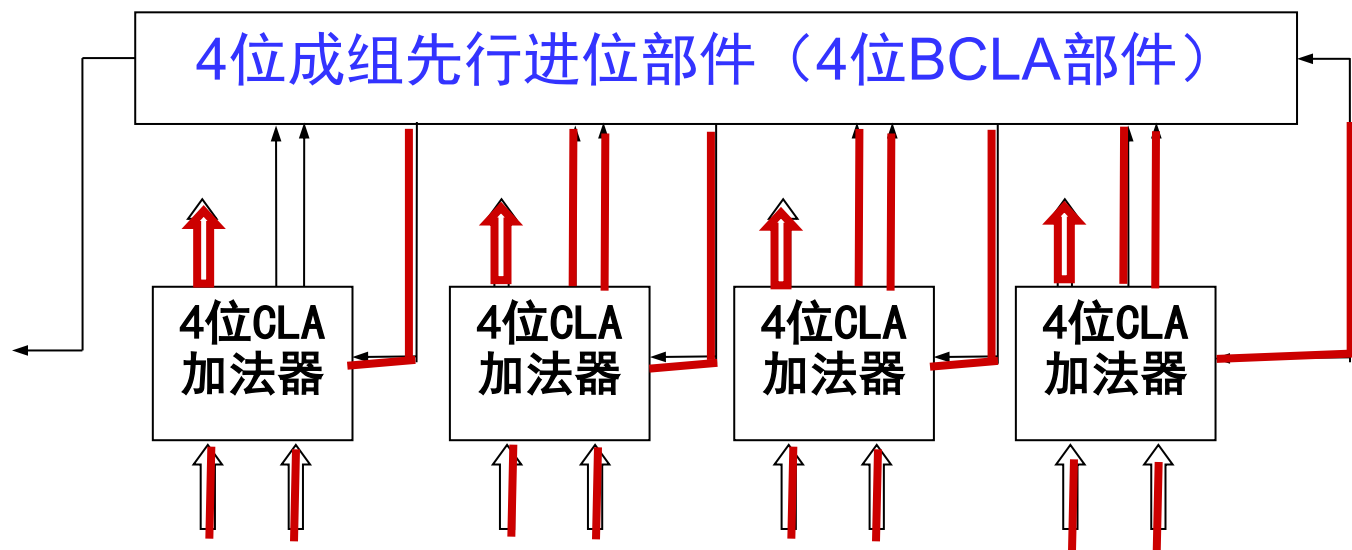
CLA加法器由“**进位生成/传递部件**”、“**CLA部件**”和“**求和部件**”构成。

8位全先行进位加法器



多级先行进位加法器

16位两级先行进位加法器



关键路径长度为多少？ $3+2+3 = 8t_y$

最终进位的延迟为多少？ $3+2=5t_y$

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

是一种模运算系统！

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（自学）
- ◆ 除法运算及其运算部件（自学）
 - 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；带符号数的算术移位运算、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

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

◆ 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）

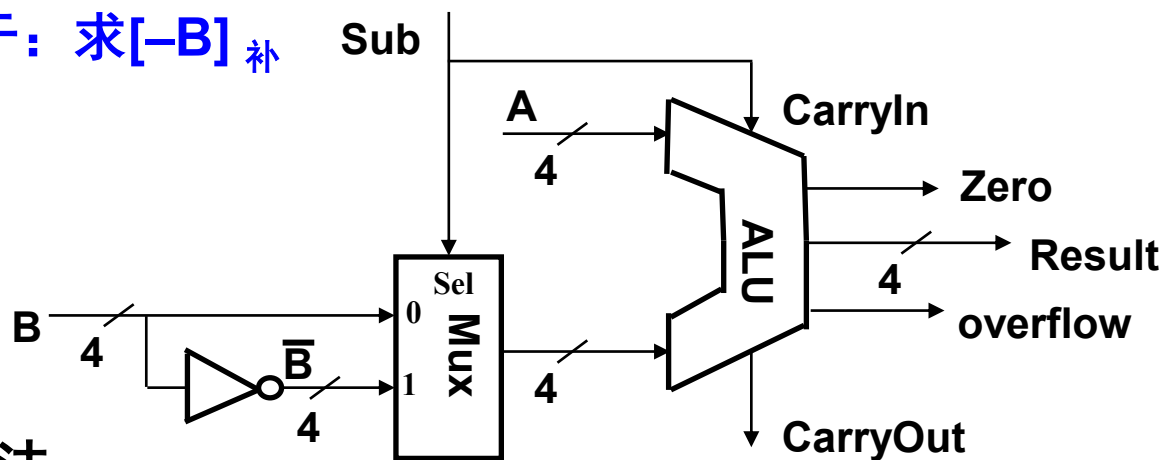
问题：模是多少？ 运算结果高位丢弃，保留低 n 位，相当于取模 2^n

- 实现减法的主要工作在于：求 $[-B]_{\text{补}}$

问题：如何求 $[-B]_{\text{补}}$ ？

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

当控制端Sub为1时，做减法
当控制端Sub为0时，做加法



补码加减运算部件

补码加/减运算与“溢出”判断

Ex1: $-7 - 6 = -7 + (-6) = +3$ **X**

	1	0			
	1	0	0	1	
+	1	0	1	0	
	0	0	1	1	

$-3 - 5 = -3 + (-5) = -8$ **✓**

	1		1		1		1		
	1	1	1	0	1				
+	1	0	1	1	1				
	1	0	0	0	0				

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

Ex2: 用8位补码计算 $107 + 46 = ?$

结果错误: $107 + 46 = -103$.

$107_{10} = 0110\ 1011_2$

$46_{10} = 0010\ 1110_2$

0	1001	1001
---	------	------

问题: 若采用变形补码结果怎样?

变形补码可保留运算中间结果。
从乘除运算过程可看出这点!

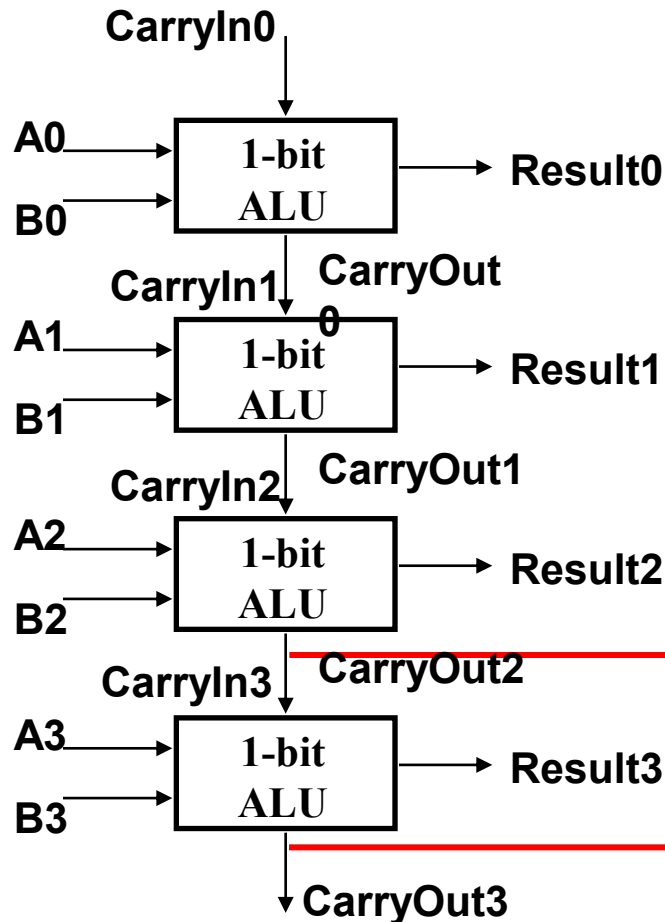
进位是真正的符号: $+153$

采用变形补码时“溢出”判断条件:
结果的两个符号位不同

Overflow Detection Logic(溢出判断逻辑)

◆ Carry into MSB != Carry out of MSB

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

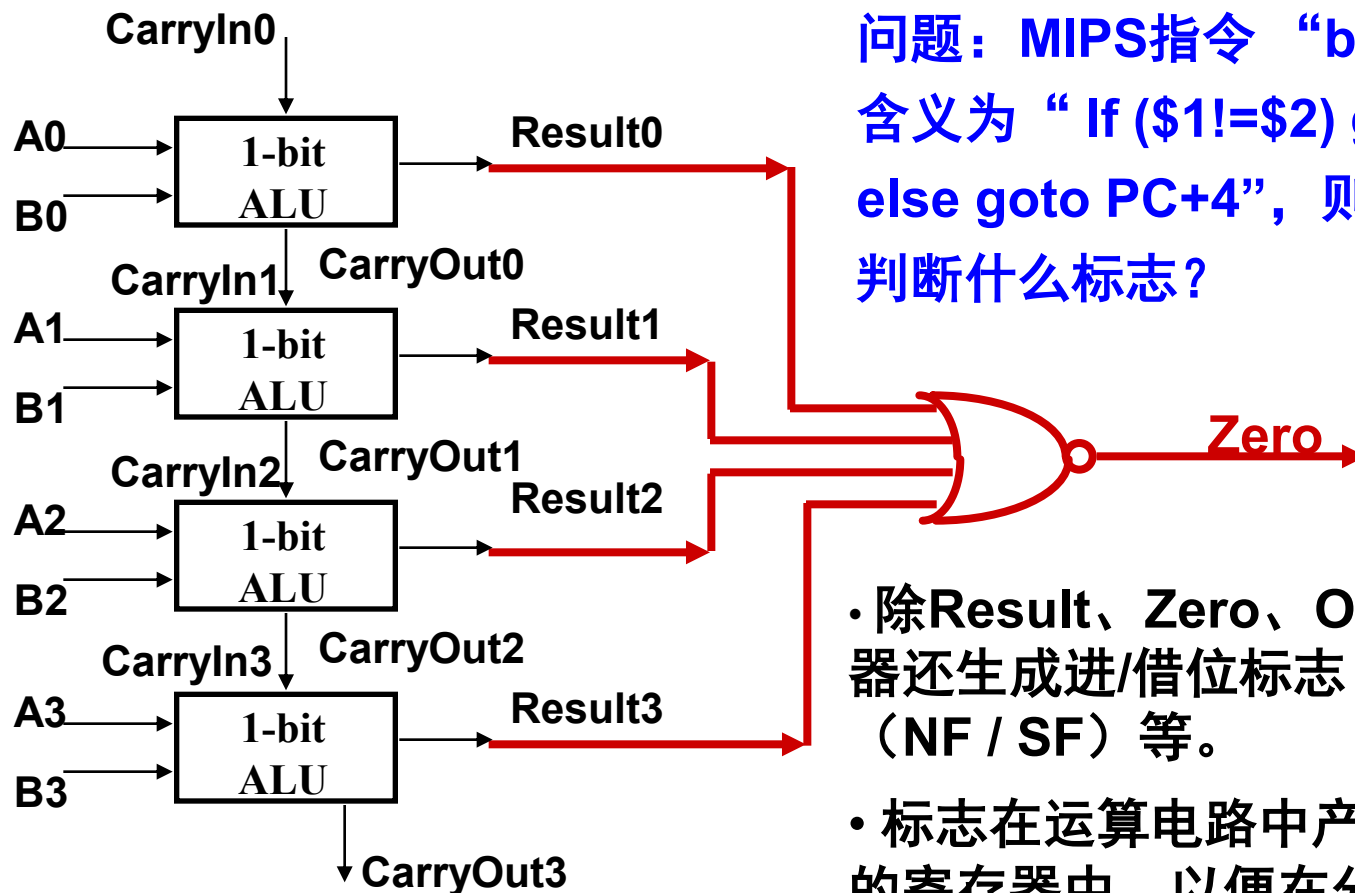


X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0



也可以用其他实现方式。

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”，则执行bne指令需判断什么标志？

- 除Result、Zero、Overflow外，许多机器还生成进/借位标志（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。
 - 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：由原码加减运算规则知：同号相加，则求和，和的符号同被加数符号。

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

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：由原码加减运算规则知：同号相减，则求差（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 $[X]_{\text{原}}$ 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 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。

移码加/减运算

例1：用四位移码计算 “ $-7 + (-6)$ ” 和 “ $-3 + 6$ ” 的值。

解： $[-7]_{\text{移}} = 0001$ $[-6]_{\text{移}} = 0010$ $[-3]_{\text{移}} = 0101$ $[6]_{\text{移}} = 1110$

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

$[-3]_{\text{移}} + [6]_{\text{移}} = 0101 + 1110 = 0011$ ，符号取反后为 1011，其真值为+3

问题： $[-7 + (-6)]_{\text{移}} = ?$ $[-3 + (6)]_{\text{移}} = ?$

例2：用四位移码计算 “ $-7 - (-6)$ ” 和 “ $-3 - 5$ ” 的值。

解： $[-7]_{\text{移}} = 0001$ $[-6]_{\text{移}} = 0010$ $[-3]_{\text{移}} = 0101$ $[5]_{\text{移}} = 1101$

$[-7]_{\text{移}} - [-6]_{\text{移}} = 0001 + 1110 = 1111$ ，符号取反后为 0111，其真值为-1。

$[-3]_{\text{移}} - [5]_{\text{移}} = 0101 + 0011 = 1000$ ，符号取反后为 0000，其真值为-8。

无符号数的乘法运算

假定： $[X]_{\text{原}} = x_0.x_1 \dots x_n$ ， $[Y]_{\text{原}} = y_0.y_1 \dots y_n$ ，求 $[x \times y]_{\text{原}}$

数值部分 $z_1 \dots z_{2n} = (0.x_1 \dots x_n) \times (0.y_1 \dots y_n)$

(小数点位置约定，不区分小数还是整数)

◆ Paper and pencil example:

Multiplicand 1000

Multiplier $\times$ 1001

$$\begin{array}{r} 1000 \\ 0000 \\ 0000 \\ 1000 \\ \hline \end{array}$$

Product (积) 0.1001000

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

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

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

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

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

整个运算过程中用到两种操作：加法 + 左移

因而，也可用ALU和移位器来实现乘法运算

无符号数的乘法运算

◆ 手工乘法的特点：

- ① 每步计算： $X \times y_i$ ，若 $y_i = 0$ ，则得0；若 $y_i = 1$ ，则得 X
- 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**。因为不等到最后一次求和，减少了保存各次相乘结果 $X \times y_i$ 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 对乘数中为“1”的位执行加法和右移，对为“0”的位只执行右移，而不执行加法运算。

无符号乘法运算的算法推导

- ◆ 上述思想可写成如下数学推导过程：

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

- ◆ 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：

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

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

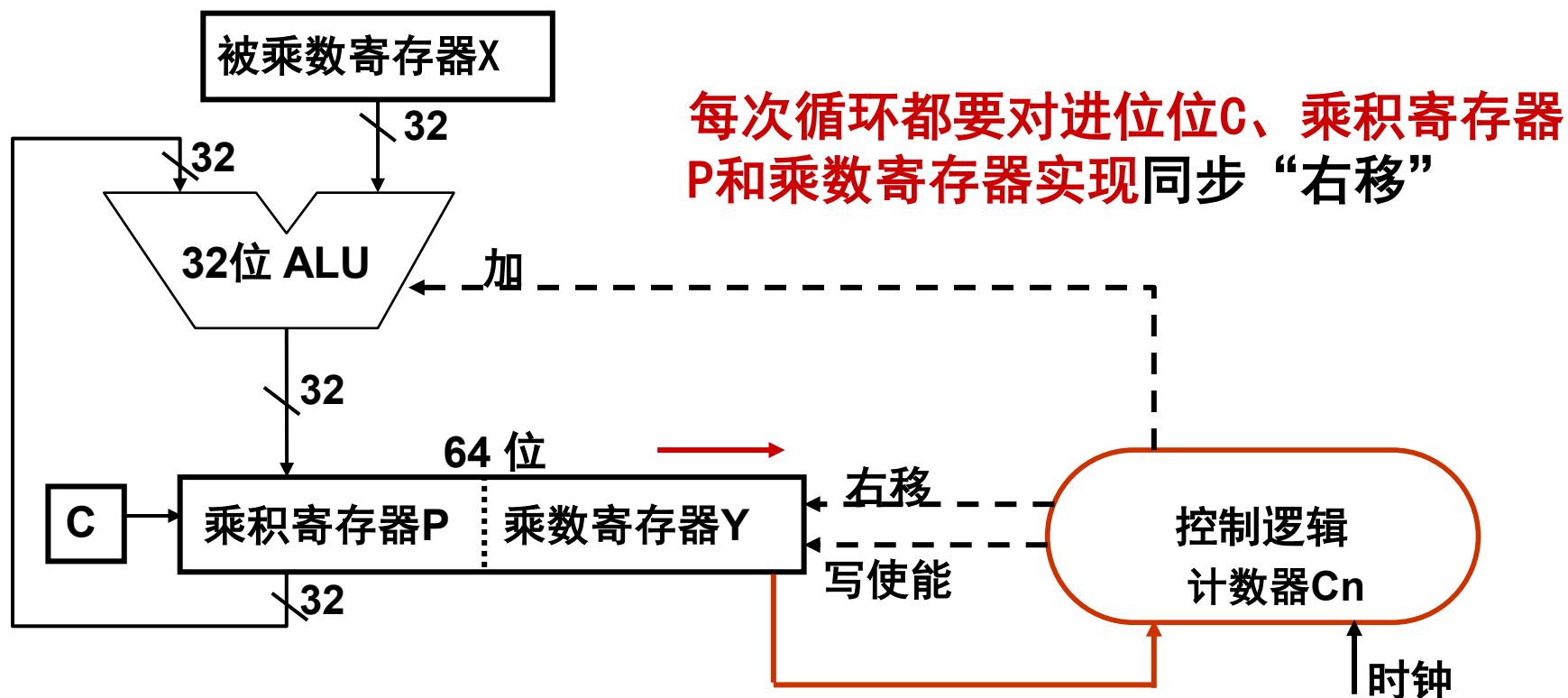
.....

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

- ◆ 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。

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Cn**：存放循环次数。初值32，每循环一次，Cn减1，Cn=0时结束
- ◆ **ALU**：乘法核心部件。在控制逻辑控制下，对P和X的内容“加”，在“写使能”控制下运算结果被送回P，进位位在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Y
	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 x 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：积符号或得到，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： $[X \times Y]_{\text{原}} = 1.10110110$

原码一位乘法：每次只取乘数的一位判断，需 n 次循环，速度慢。

原码两位乘法：每次取乘数两位判断，只需 $n/2$ 次循环，快一倍。

◆ 原码两位乘法递推公式：

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

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	$1 \rightarrow T$	$2^{-2} (P_i)$

T触发器用来记录下次是否要执行“+X”

“-X”运算用“+ $[-X]_{\text{补}}$ ”实现！

3X时，本次-X，下次+X！

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: 先用无符号数乘法计算 111001×100111 , 原码两位乘法过程如下:

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

采用补码右移

为模8补码形式
(三位符号位), 为什么?

若用模4补码, 则P和Y同时右移2位时, 得到的P3是负数, 这显然是错误的! 需要再增加一位符号。

P	Y	T	说明
000 000000	100111	0	开始, $P_0=0$, $T=0$
+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
+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
- ◆ 符号与数值统一处理

因为 $[X \times Y]_{\text{补}} \neq [X]_{\text{补}} \times [Y]_{\text{补}}$ ，故不能直接用无符号乘法计算。

Booth's Algorithm推导如下：

假定： $[X]_{\text{补}} = x_{n-1}x_{n-2}\cdots x_1x_0$ ， $[Y]_{\text{补}} = y_{n-1}y_{n-2}\cdots y_1y_0$ ，求： $[X \times Y]_{\text{补}} = ?$

基于以下补码性质：

令 $[Y]_{\text{补}} = y_{n-1}y_{n-2}\cdots y_1y_0$ ，则 $Y = -y_{n-1} \cdot 2^{n-1} + y_{n-2} \cdot 2^{n-2} + \cdots + y_1 \cdot 2^1 + y_0 \cdot 2^0$

令： $y_{-1} = 0$ ，则：

当 $n=32$ 时， $Y = -y_{31} \cdot 2^{31} + y_{30} \cdot 2^{30} + \cdots + y_1 \cdot 2^1 + y_0 \cdot 2^0 + y_{-1} \cdot 2^0$

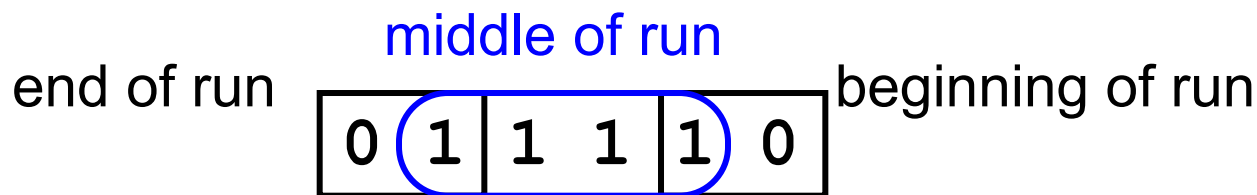
$$\downarrow$$
$$-y_{31} \cdot 2^{31} + (y_{30} \cdot 2^{31} - y_{30} \cdot 2^{30}) + \cdots + (y_0 \cdot 2^1 - y_0 \cdot 2^0) + y_{-1} \cdot 2^0$$

$$\downarrow$$
$$(y_{30} - y_{31}) \cdot 2^{31} + (y_{29} - y_{30}) \cdot 2^{30} + \cdots + (y_0 - y_1) \cdot 2^1 + (y_{-1} - y_0) \cdot 2^0$$

$$\begin{aligned} 2^{-32} [X \times Y]_{\text{补}} &= (y_{30} - y_{31})X \cdot 2^{-1} + (y_{29} - y_{30})X \cdot 2^{-2} + \cdots + (y_0 - y_1)X \cdot 2^{-31} + (y_{-1} - y_0)X \cdot 2^{-32} \\ &= 2^{-1}(2^{-1} \cdots (2^{-1}(y_{-1} - y_0)X) + (y_0 - y_1)X) + \cdots + (y_{30} - y_{31})X \end{aligned}$$

部分积公式： $P_i = 2^{-1}(P_{i-1} + (y_{i-1} - y_i)X)$

Booth's 算法实质



当前位	右边位	操作	Example
1	0	减被乘数	000111 <u>10</u> 00
1	1	加0 (不操作)	00011 <u>11</u> 000
0	1	加被乘数	00 <u>01</u> 111000
0	0	加0 (不操作)	0 <u>00</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）

布斯算法举例

已知 $[X]_{\text{补}} = 1\ 101$, $[Y]_{\text{补}} = 0\ 110$, 计算 $[X \times Y]_{\text{补}}$ $[-X]_{\text{补}} = 0011$

P	Y	y_{-1}	说明
0000	0110	0	设 $y_{-1} = 0$, $[P_0]_{\text{补}} = 0$
0000	0011	0	$y_0 y_{-1} = 00$, P、Y 直接右移一位 得 $[P_1]_{\text{补}}$
+0011	1001	1	$y_1 y_0 = 10$, $+[-X]_{\text{补}}$ P、Y 同时右移一位 得 $[P_2]_{\text{补}}$
0000	1100	1	$y_2 y_1 = 11$, P、Y 直接右移一位 得 $[P_3]_{\text{补}}$
+1101	1110	0	$y_3 y_2 = 01$, $+ [X]_{\text{补}}$ P、Y 同时右移一位 得 $[P_4]_{\text{补}}$
1101			
1110			

验证: $X = -3$, $Y = 6$, $X \times Y = -0010010B = -18$, 结果正确!

Booths Example: 2 x -3

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

mythical bit

最后乘积

MIPS中的乘、除运算处理

- ◆ 指令: `mult` , `multu`; `div` , `divu`
- ◆ 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`/`mfhi`指令用来把Lo/Hi中的32位数据取到通用寄存器
- ◆ 两种乘法指令都忽略overflow, 而由软件自行处理溢出 **问题: 如何判断溢出?**
 - 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“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

符号和数值部分分开处理！

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

问题：本位和在什么
范围内需“+6”校正
？

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

大于9：1010, 1011, ..., 1111

有进位：10000, 10001, 10010和10011

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

一位十进制加法器

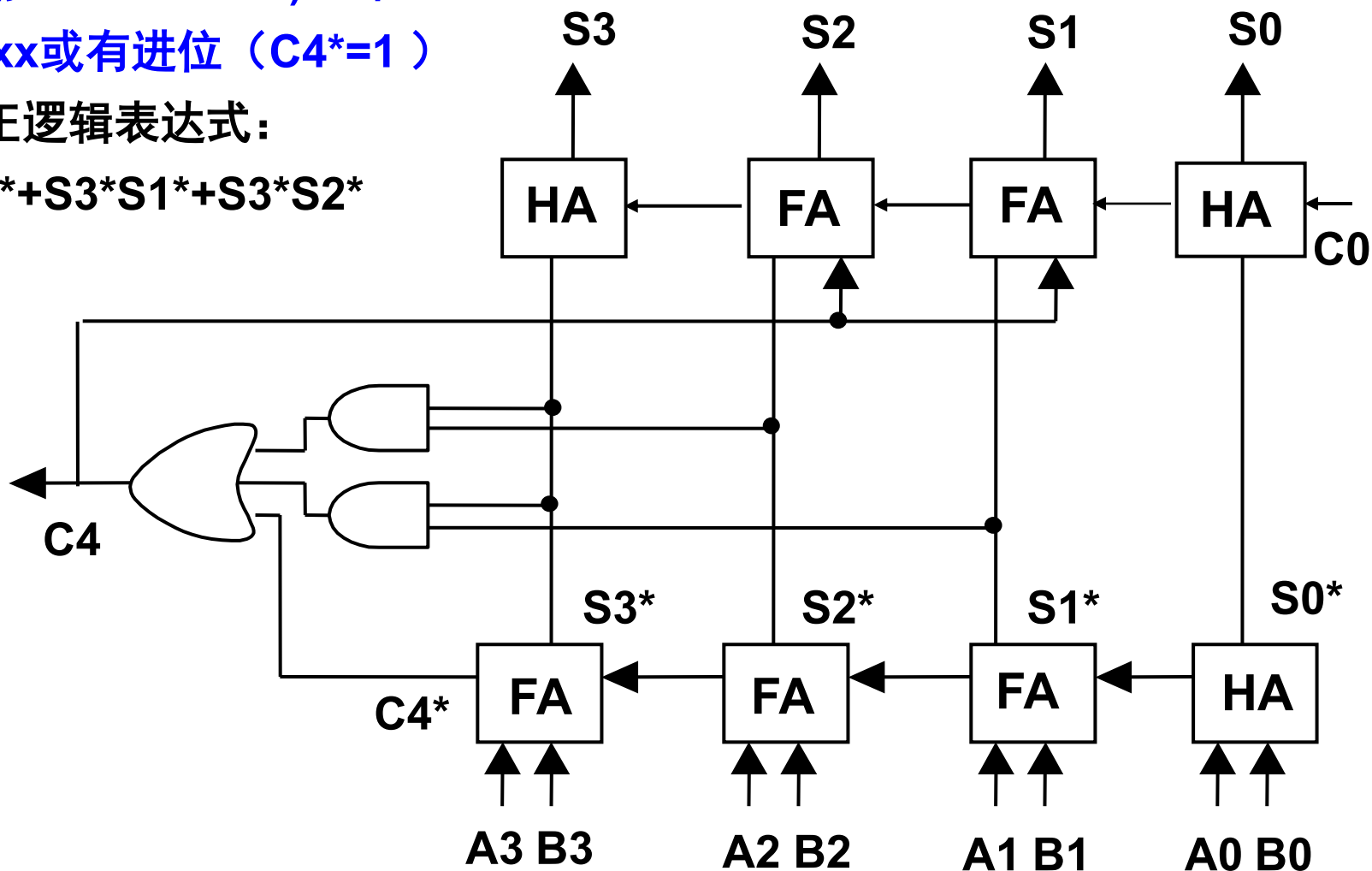
当结果在10~19之间时，需校正

。(最大可能： $2 \times 9 + 1 = 19$)。即：

1x1x或11xx或有进位 ($C4^*=1$)

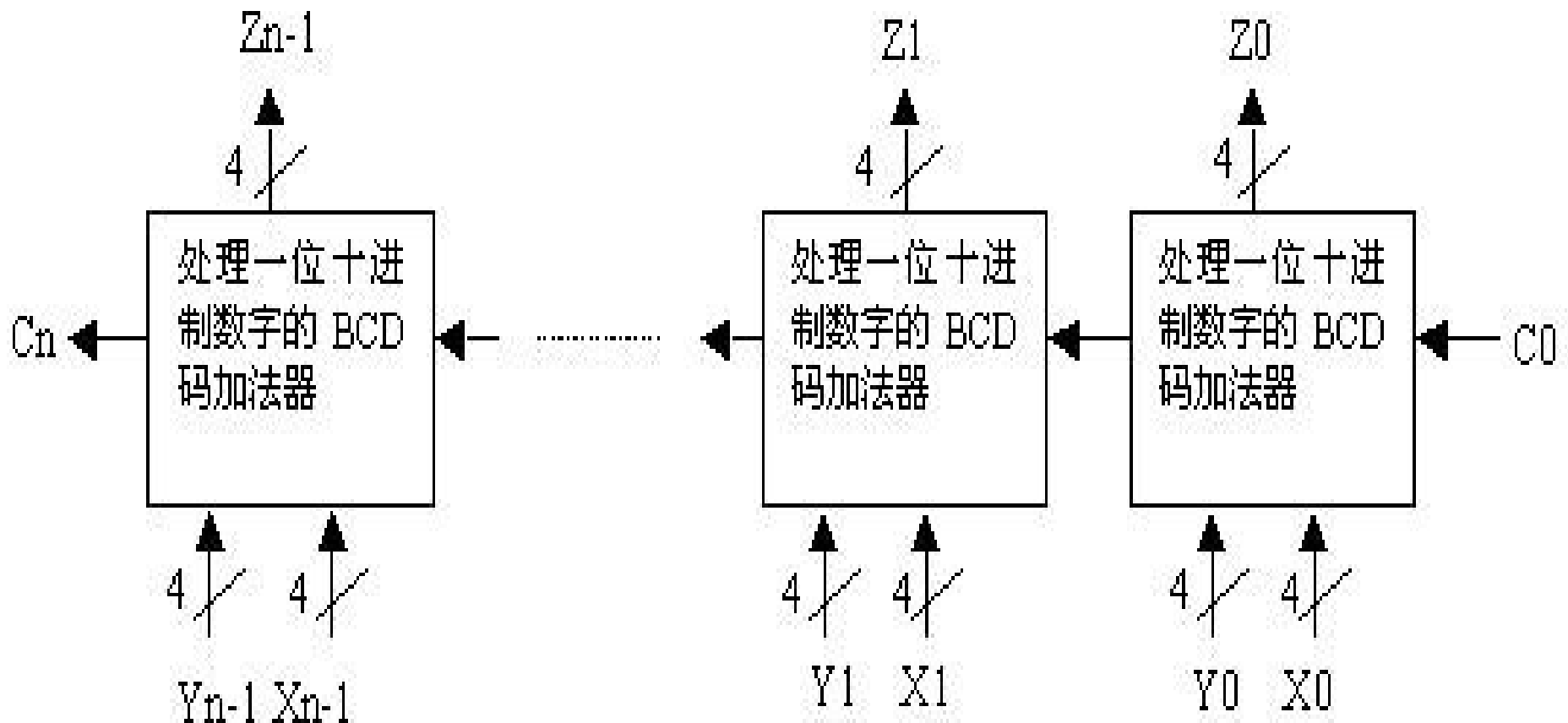
所以，校正逻辑表达式：

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



n位十进制加法器

- ◆ n个一位十进制加法器 $\Rightarrow$
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。

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

十进制减法运算举例

例1 $309-125=184$

加补码: $309+875=184$

例2 $125-309 = -184$

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

0011 0000 1001

+1000 0111 0101

1011 0111 1110

0110 0110

10001 1000 0100

进位为1，表示
被减数大于减数
，结果为正

0001 0010 0101

+0110 1001 0001

0111 1011 0110

0110

1000 0001 0110

无进位，表示差
值为负数，故应
将结果取补

取补

-0001 1000 0100

减法运算肯定不会溢出！

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));
}
```



```
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
```

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。

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:

SP最大指数为多少? 127!

阶码上溢: 一个正指数超过了最大允许值 $\Rightarrow +\infty/-\infty/\text{溢出}$

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

尾数溢出: 最高有效位有进位 $\Rightarrow$ 右规 **尾数溢出, 结果不一定溢出** **-126!**

非规格化尾数: 数值部分高位为0 $\Rightarrow$ 左规

右规或对阶时, 右段有效位丢失 $\Rightarrow$ 尾数舍入 **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 无效运算（无意义）

- 运算时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：事先设定好是否要进行硬件处理（即挖一个陷阱），当出现相应异常时，就由硬件自动进行相应的异常处理（掉入陷阱）。

浮点数加/减运算

◆ 十进制科学计数法的加法例子

$$0.123 \times 10^5 + 0.560 \times 10^2$$

其计算过程为：

$$\begin{aligned} 0.123 \times 10^5 + 0.560 \times 10^2 &= 0.123 \times 10^5 + 0.000560 \times 10^5 \\ &= (0.123 + 0.00056) \times 10^5 = 0.12356 \times 10^5 \\ &= 0.124 \times 10^5 \end{aligned}$$

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
- IEEE 754尾数右移时，要将隐含的“1”移到小数部分，高位补0，移出的低位保留到特定的“附加位”上

浮点数加/减运算-对阶

问题：如何对阶？

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

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

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

例如， $Ex=7$ ， $Ey=-7$ ，则 $[\Delta E]_{\text{补}}=1111+1111=1110$ ， $\Delta E < 0$ ，错！

问题：对IEEE754 SP格式来说， $|\Delta E|$ 大于多少时结果就等于阶大的那个数？ 24！

$1.xx...x \rightarrow 0.00...01xx...x$ (右移24位后，尾数变为0)

问题：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 $[Ex - Ey]_{\text{补}} \pmod{2^n}$ 没有影响

$$\begin{aligned} [\Delta E]_{\text{补}} &= 256 + Ex - Ey = 256 + 127 + Ex - (127 + Ey) \\ &= 256 + [Ex]_{\text{移}} - [Ey]_{\text{移}} = [Ex]_{\text{移}} + [-[Ey]_{\text{移}}]_{\text{补}} \pmod{256} \end{aligned}$$

但 $[Ex + Ey]_{\text{移}}$ 和 $[Ex - Ey]_{\text{移}}$ 的计算会变复杂！ 浮点乘除运算涉及之。

浮点数加减法基本要点

(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, 则需要将阶码也置0。为什么?

尾数为0说明结果应该为0, 即: 阶码和尾数为全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: $1.000 \times 2^{-4} = 0.0001000 = 1/16 = 0.0625$

问题: 为何IEEE 754 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。

IEEE 754 浮点数加法运算

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 $[\Delta E]_{\text{补}} = ?$

(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？

(6) 如何舍入？ $\pm 1x.xx\dots x$ 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

(7) 如何判断溢出？ 最终须把附加位去掉，此时需考虑舍入（IEEE754有四种舍入方式）

若最终阶码为全1，则上溢；若尾数为全0，则下溢

IEEE 754 浮点数加法运算举例

已知 $x=0.5$, $y=-0.4375$, 求 $x+y=?$ (用IEEE754标准单精度格式计算)

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

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

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

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

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: $01.0000...0 + (10.1110...0) = 00.00100...0$

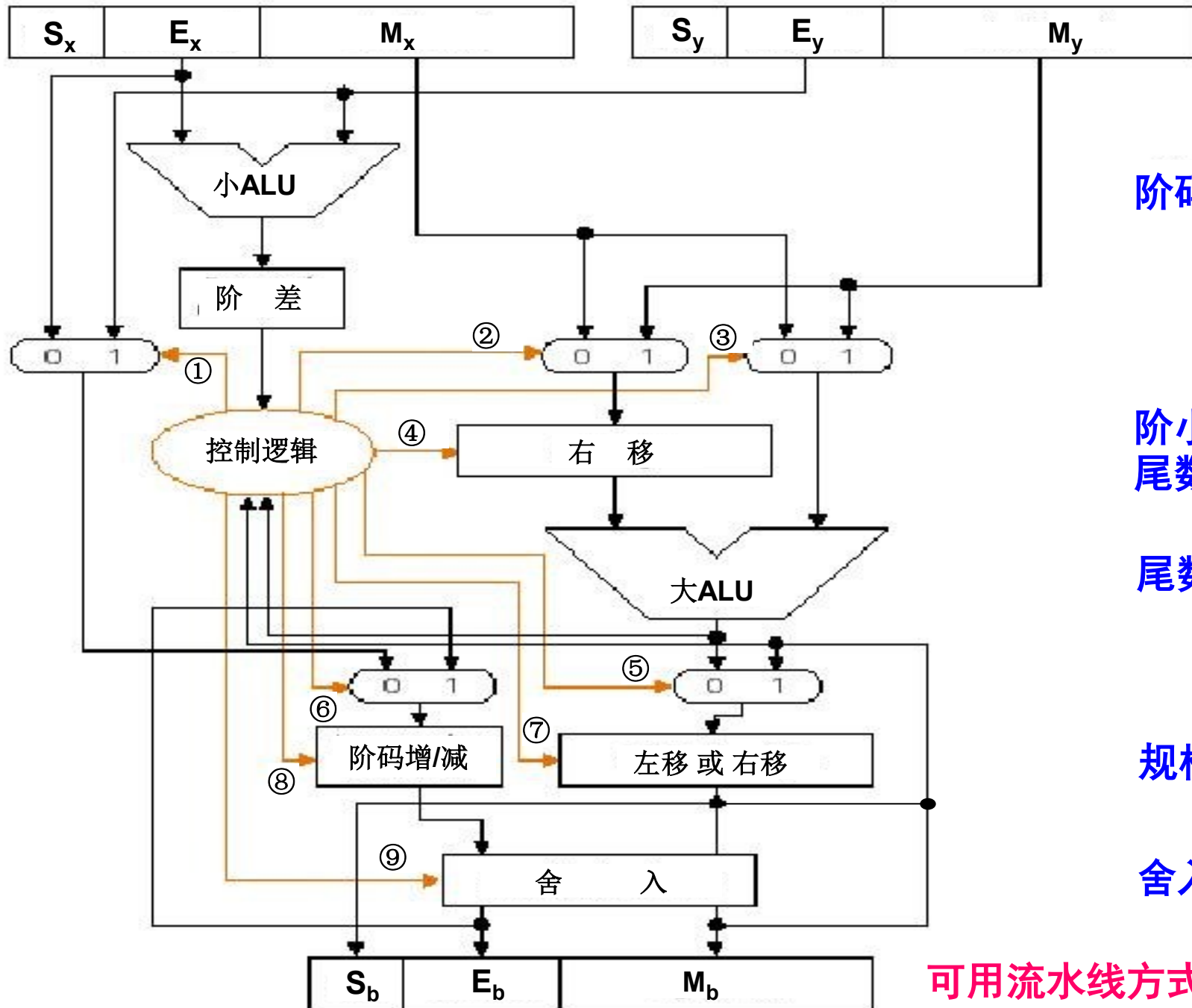
(原码加法, 最左边一位为符号位)

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)

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

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



阶码相减

阶小的数的
尾数右移

尾数加/减

规格化

舍入

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

(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**: $X_e \pm Y_e \mp 127$

(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;

(4) 当尾数高位为0, 需左规; 当尾数最高位有进位, 需右规。

(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?

问题2: 除法呢? 左规次数不定! 不需右规! 不需左规! 最多右规1次!

求阶码的和、差

设 E_x 和 E_y 分别是两个操作数的阶码， E_b 是结果的阶码，则：

- 阶码加法公式为： $E_b \leftarrow E_x + E_y + 129 \pmod{2^8}$

$$\begin{aligned}[E_1 + E_2]_{\text{移}} &= 127 + E_1 + E_2 = 127 + E_1 + 127 + E_2 - 127 \\ &= [E_1]_{\text{移}} + [E_2]_{\text{移}} - 127 \\ &= [E_1]_{\text{移}} + [E_2]_{\text{移}} + [-127]_{\text{补}} \\ &= [E_1]_{\text{移}} + [E_2]_{\text{移}} + 10000001B \pmod{2^8}\end{aligned}$$

- 阶码减法公式为： $E_b \leftarrow E_x + [-E_y]_{\text{补}} + 127 \pmod{2^8}$

$$\begin{aligned}[E_1 - E_2]_{\text{移}} &= 127 + E_1 - E_2 = 127 + E_1 - (127 + E_2) + 127 \\ &= [E_1]_{\text{移}} - [E_2]_{\text{移}} + 127 \\ &= [E_1]_{\text{移}} + [-[E_2]_{\text{移}}]_{\text{补}} + 01111111B \pmod{2^8}\end{aligned}$$

举例

设 E_x 和 E_y 分别是两个操作数的阶码， E_b 是结果的阶码

例：若两个阶码分别为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：

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

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

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

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’”

如何才能使失去的“沙”和捡回的“脏”都尽量少呢？在后面加附加位！
加多少附加位才合适？

Add/Sub: 无法给出准确的答案！

	1.xxxxx	1.xxxxx	1.xxxxx	1.xxxxxxxxx
+	1.xxxxx	0.001xxxx	0.01xxxx	-1.xxxxxxxxx
	<u>1x.xxxx</u>	<u>1.xxxxx</u>	<u>1x.xxxx</u>	<u>0.0...0xxxx</u>
	y	yyy	yyy	

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

[BACK](#)

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。

Rounding Digits(舍入位)

举例：十进制数，最终有效位数为 3，假定采用两位附加位。

问题：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！

$$\begin{array}{r} 2.34\textcolor{blue}{00} * 10^2 \\ 0.02\textcolor{blue}{53} * 10^2 \\ \hline 2.365\textcolor{red}{3} * 10^2 \end{array}$$

IEEE Standard: four rounding modes (用图说明)

round to nearest **(default)**

round towards plus infinity (always round up)

round towards minus infinity (always round down)

round towards 0

round to nearest: 简称为就近舍入到偶数

round digit < 1/2 then truncate (截取)

> 1/2 then round up (add 1 to ULP)

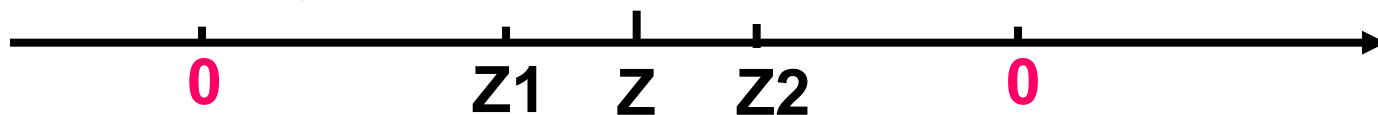
= 1/2 then round to nearest even digit

可以证明默认方式得到的平均误差最小。

注：ULP=units in
the last place.

IEEE 754的舍入方式的说明

IEEE 754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: $1.1101\mathbf{11} \rightarrow 1.1110$; $1.1101\mathbf{01} \rightarrow 1.1101$;
 $1.1101\mathbf{10} \rightarrow 1.1110$; $1.1111\mathbf{10} \rightarrow 10.0000$;

(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。正数: 取Z1; 负数: 取Z2

IEEE 754的舍入方式的说明

IEEE 754通过在舍入位后再引入“粘位sticky bit”增强精度
加减运算对阶过程中，若阶码较小的数的尾数右移时，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1.24509，所以分别为：

1.24, 1.24, 1.25

[BACK](#)

溢出判断

以下情况下，可能会导致阶码溢出

- 左规（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（阶码 + 1）时

- 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。

问题：机器内部如何减1？ $+[-1]_{\text{补}} = + 11\dots 1$

举例

以下情况下，可能会导致阶码溢出（续）

- 乘法运算求阶码的和时

- 若 E_x 和 E_y 最高位皆1，而 E_b 最高位是0或 E_b 为全1，则阶码上溢
- 若 E_x 和 E_y 最高位皆0，而 E_b 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，作为结果操作数
 - 可用一条乘累加指令代替两条MIPS浮点指令
 - 可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是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），有时会发生奇怪的现象
- 由浮点数访存指令自动完成转换
- 指令类型：访存、算术、比较、函数（正弦、余弦、对数等）

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

本章作业

◆习题3中的3、5、6、7（1-3）、9（1）

第1题需要掌握不需写作业提交；

其它写在A4纸上，要求书写工整、清楚，答题要完整！