Ch3: Arithmetic and Logic Operate and ALU

运算方法和运算部件

第一讲 定点数运算及其运算部件

第二讲 浮点数运算及其运算部件

第一讲：定点数运算及其运算部件

主 要 内 容

- C语言程序中涉及的运算
- 算术逻辑部件
- 定点数加减运算及基本运算部件ALU的设计
  - 补码加减运算
  - 原码加减运算
  - 移码加减运算
- 定点数乘法运算
- 定点数除法运算

数据的运算

- 高级语言程序中涉及的运算（以C语言为例）
  - 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
- 指令集中涉及到的运算
  - 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

为什么房子里要有厨房、卫生间、卧室？
CPU中需提供哪些运算部件？Why？

人的需要！程序的需要！

我们从程序对运算的需求开始分析。

浮点数有没有移位操作和扩展操作？为什么？

C语言程序中涉及的运算-算术及按位运算

- 算术运算（最基本的运算）
  - 无符号数、带符号整数、浮点数的+、-、*、/ 运算等
- 按位运算
  - 操作
    - 按位与：“&”
    - 按位或：“|”
    - 按位取反：“~”
    - 按位异或：“^”
  - 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状态/控制信息进行处理）

    举例：如何从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，得到结果为：0x2C00

C语言程序中涉及的运算-逻辑运算

- 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“AND”运算

      例如， if ((x>y) && (i<100)) then ……
    - “||”表示“OR”运算
    - “!”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k位:：x<<k;　　右移k位： x>>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（有效数据丢失）！

    问题：何时可能发生溢出？如何判断溢出？

    溢出判断：　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(C_n)不等于新的符号位(C_{n-1}), 即$C_n \oplus C_{n-1}=1$，则溢出。

    右移：低位移出，高位补符，可能发生有效数据丢失。

如何从16位数据y中提取高位字节？(y>>8) 再送入8位寄存器

某字长为8的机器中，x、y和z都是8位带符号整数，已知

$x=-81=1010\_1111_2$，则y=x/2=？（右移1位后$y=1101_0111_2=-41$）

z=2x=？（z=−162？左移1位后y=1010_1111<<1=0101_1110移出的位($C_n=1$)不等于新的符号位($C_{n-1}=0$)，溢出！

C语言程序中涉及的运算-位扩展和截断

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：长位数转短位数
    强行将高位丢弃，故可能发生“溢出”

例2:下面的i和j相等？No!

```
int i = 32768;
short si = (short) i;
int j = si;    // 16位转32位
```

$i = 32768_{10}$= 00 00 80 00H

$si = -32768_{10}$=8000H

i !=si的原因：对i截断时发生了“溢出”，即：32768截断为16位数时，因其超出16位能表示的最大有符号值，故不等！

$j = -32768_{10}$=FF FF 80 00H (16位有符号数扩展为32位有符号数)

例1（扩展操作）：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 = -32768;
unsigned short  usi = si;
int  i = si;
unsingned  ui = usi ;
```

$si = -32768_{10}$ = 80 00H(有符号整数)

$usi = 32768_{10}$ = 80 00H(16位无符号整数)

$i = -32768_{10}$ = FF FF 80 00H(有符号整数)

$ui = 32768_{10}$ = 00 00 80 00H(32位无符号整数)

如何实现高级语言源程序中的运算？

- 总结：C语言程序中的基本数据类型及其基本运算类型
  - 基本数据类型
    - 无符号数、带符号整数、浮点数、位串、字符（串）
  - 基本运算类型
    - 算术、按位、逻辑、移位、扩展和截断、匹配
- 计算机如何实现高级语言程序中的运算？
  - 将各类表达式编译（转换）为指令序列
  - 计算机直接执行指令来完成运算

例：C语言赋值语句“f = (g+h) – (i+j);”中变量i、j、f、g、h由编译器分别分配给MIPS寄存器$t0~$t4。寄存器$t0~$t7的编号对应8~15，上述程序段对应的MIPS机器代码和汇编表示（#后为注释）如下：

```
000000 01011 01100 01101 00000 100000   add $t5, $t3, $t4   # g+h
000000 01000 01001 01110 00000 100000   add $t6, $t0, $t1   # i+j
000000 01101 01110 01010 00000 100010   sub $t2, $t5, $t6   # f =(g+h)–(i+j)
```

需要提供哪些运算类指令才能支持高级语言需求呢？

n位整数加/减运算器

下面C程序段中x、y、z1和z2的机器数是什么?

```
int  x=9;        // 答:  x的机器数为[x]补;
int  y=-6;       // 答:  y的机器数为[y]补;
int  z1, z2;
z1=x+y;          // 答:  z1的机器数为[x+y]补;
z2=x-y;          // 答:  z2的机器数为[x-y]补
```

因此，计算机中需要有一个电路，能够实现以下功能:

已知 $[x]_{补}$ 和 $[y]_{补}$，计算$[x+y]_{补}$ 和 $[x-y]_{补}$。

根据补码定义，假定补码有n位，有如下公式:

- $[X]_{补}=(2^n+X) \bmod 2^n \quad (-2^n \le X < 2^n)$

  $[-y]_{补}=[\overline{y}]_{补}+1$

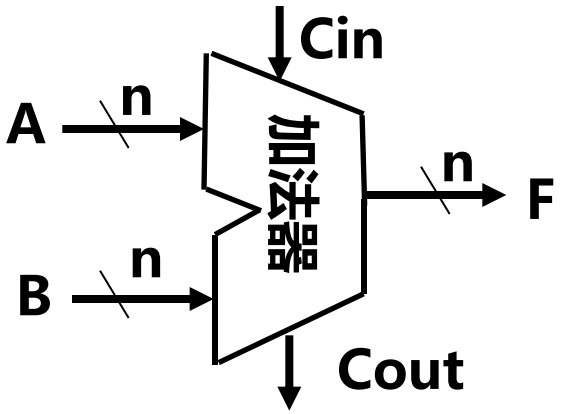
  $[x+y]_{补}=2^n+x+y=2^n+x+2^n+y=[x]_{补}+[y]_{补} \pmod{2^n}$

  $[x-y]_{补}=2^n+x-y=2^n+x+2^n-y=[x]_{补}+[-y]_{补} \pmod{2^n}$

整数加、减运算

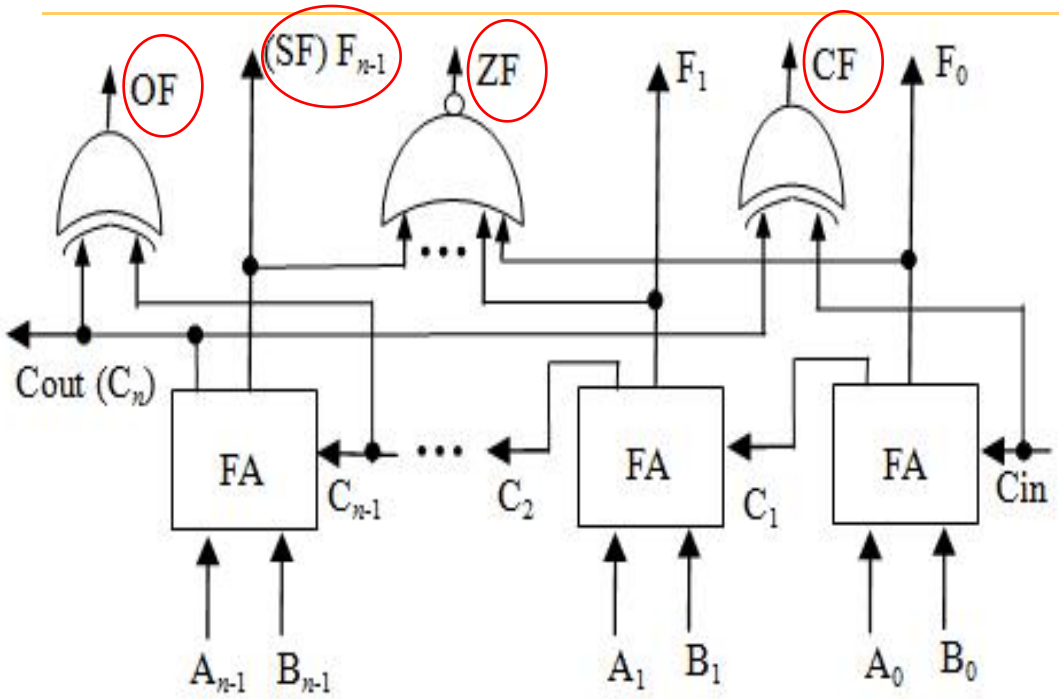
- C语言程序中的整数有
  - 带符号整数，如char、short、int、long型等
  - 无符号整数，如unsigned char、unsigned short、unsigned等，指针、地址等通常被说明为无符号整数
- 计算机中的加法器，因为只有n位，所以是一种模2^n运算系统！
- 下面的加法器无法直接用于两个n位带符号整数（补码）或无符号整数相加，因为无法直接判断出是否溢出

  例：n=4，A=1001，B=1100，则：F=0101，Cout=1



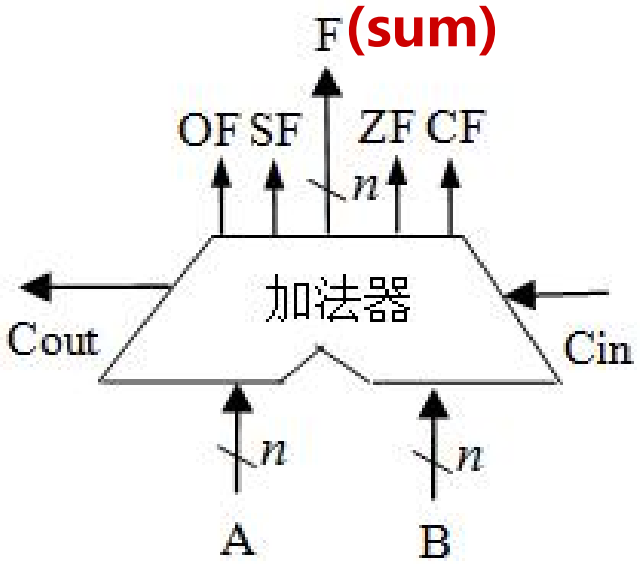
无符号整数和带符号整数的加、减运算电路需要完全一样，这个运算电路称为整数加减运算部件，需要基于带标志加法器实现

带标志加法器



带标志加法器的逻辑电路

==



带标志加法器符号

计算条件标志 (Flag)，其中C_n,C_{n-1}分别表示在第n,n-1位中进位来的值：

溢出标志OF为$C_n \oplus C_{n-1}$，或A与B'同号,但与Sum不同号，则1；否则0!

符号标志SF为F_{n-1},是F(即加法和的值sum)的最高位(符号位)

零标志ZF为1，当且仅当和的值F=0

进位/借位标志CF为Cout⊕Cin

4位整数加/减运算器

- 利用带标志加法器，可构造整数加/减运算器，进行有(或无)符号整数加(或减). 有符号补码加减运算公式如下，

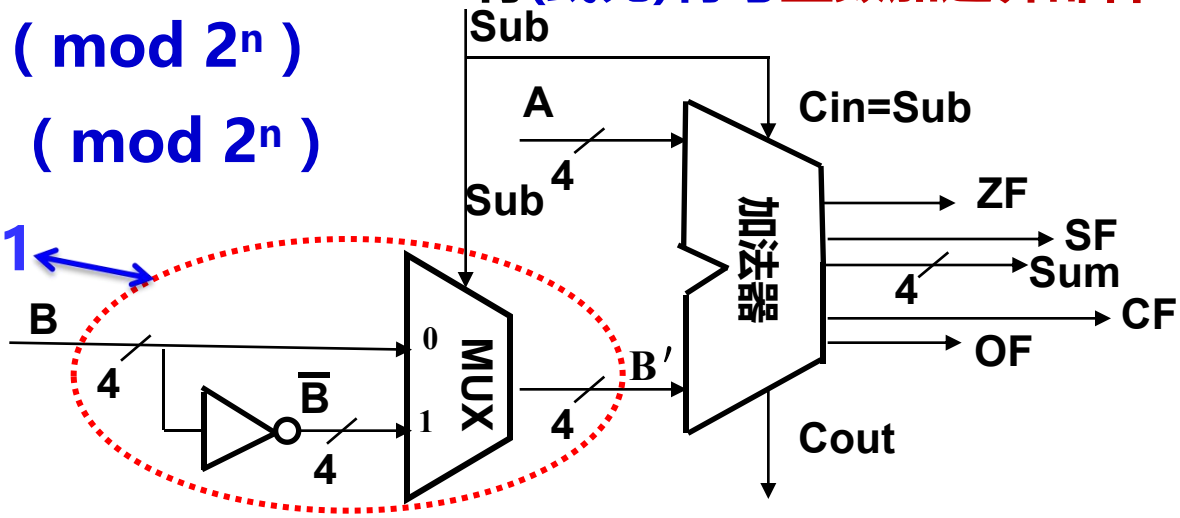
$[A+B]_{补} = [A]_{补} + [B]_{补} \quad (\bmod 2^n)$

$[A-B]_{补} = [A]_{补} + [-B]_{补} \quad (\bmod 2^n)$

如何求$[-B]_{补}=\overline{[B]}_{补}+1$

做减法时,Sub信号设为1;
做加法时,Sub信号设为0。

有(或无)符号整数加运算部件



进/借位CF=1表示无符号加法或减法溢出

溢出位OF=1表示带符号加法或减法溢出

带符号减溢出条件：(1) 最高位和次高位的进位不同,即 OF=1

或(2) 和的符号位和加数的符号位不同

ZF、OF、CF、SF这些条件标志在运算电路中产生，被分别记录到专门的寄存器中的一个标志位,通常称为程序/状态字寄存器或标志寄存器中,如 IA-32中的EFLAGS寄存器。

整数加法举例

进/借位CF=1表示无符号加法或减法溢出

溢出位OF=1表示带符号加法溢出

用该8位加法器计算107+46=?

分有符号无符号加法分别讨论

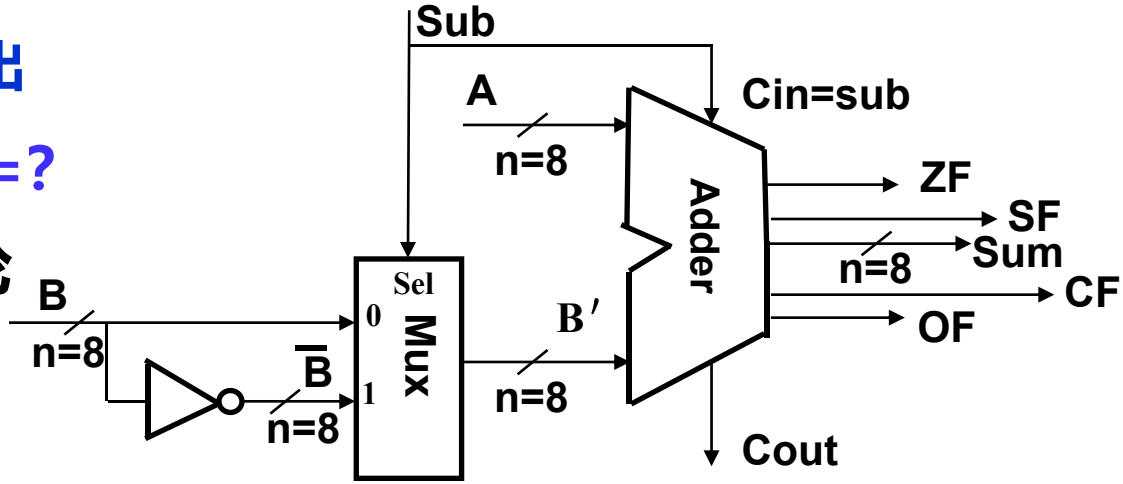
107_{10}= 0110 1011$_2$

46_{10} = 0010 1110$_2$

0 1001 1001

C_n=Cout位

整数加/减运算部件(8位加法器)



符号标志SF=1,两个正数相加，结果为负数，明显不对，溢出!

溢出标志OF=$C_n \oplus C_{n-1}$=1,故发生带符号加法溢出，sum= -103 这个结果错误!

零标志ZF=0

进/借位标志CF=Cout⊕ Cin=Cout⊕ sub=0⊕ 0=0,故无符号加未溢出（因为是做加法，所以这里sub信号=0）

如果是无符号加：因为CF=0，故未发生溢出，sum=153这个结果正确!

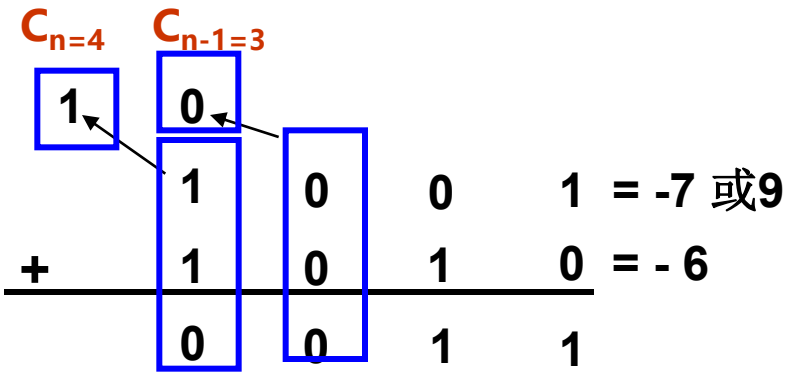
如果是有符号加：因为OF=1，故发生溢出，sum= -103 这个结果错误!

整数减法举例(用4位来表示每个数)

带符号减:(-7)-6=(-7)+(-6)=+3 X有溢出　　(-3) - 5=(- 3) +(-5) =(-8)√

无符号减: 9 - 6 = 3 √　　13 - 5 = 8 √

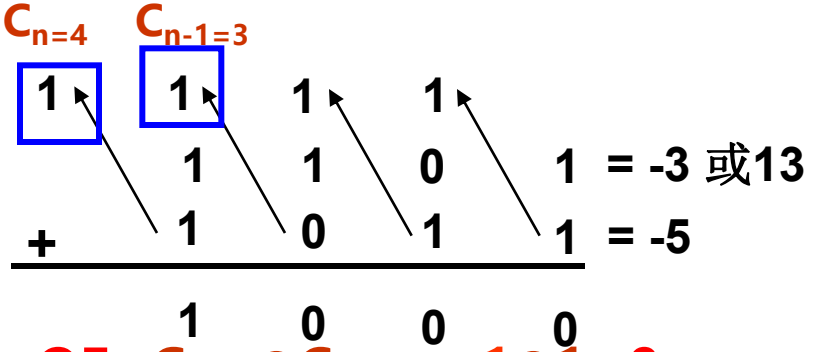
以上每个数用4位来表示,n=4, n-1=3



$OF=C_{n=4}\oplus C_{n-1=3}=1\oplus 0=1$

ZF=0、SF=0

借位CF=Cout⊕Cin=1⊕1=0



$OF=C_{n=4}\oplus C_{n-1=3}=1\oplus 1=0$

ZF=0、 SF=1、

进/借位CF=Cout⊕Cin=1⊕1=0

带符号减溢出条件： (1) 最高位和次高位的进位不同,即 OF=1

或(2) 和的符号位和加数的符号位不同

无符号减溢出条件：进/借位CF=1，即差为负数

做减法以比较大小，规则：

Unsigned: CF=0时，大于

Signed：OF=SF时，大于

验证 ⇒ 无符号减：9>6,故CF=0; 13>5,故CF=0

有符号减：-7<6，故OF≠SF

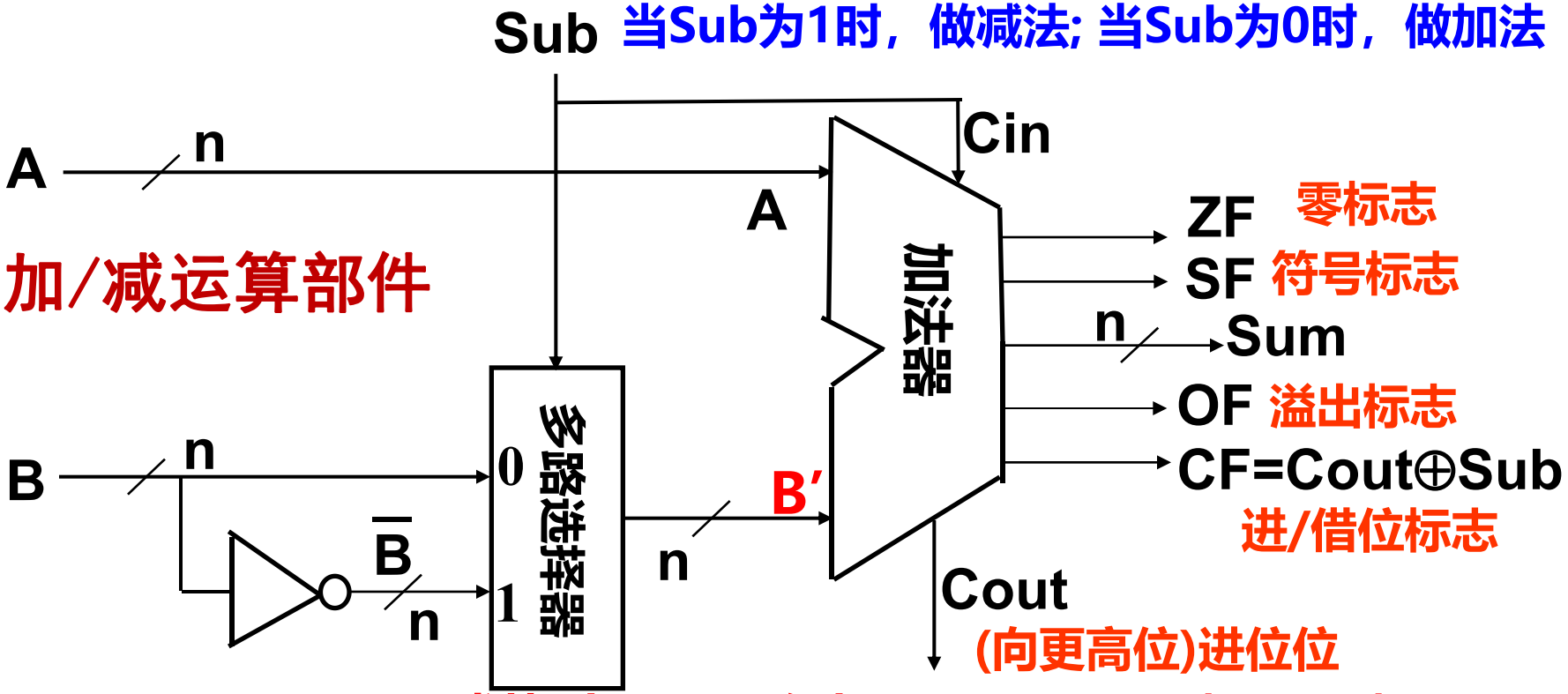
-3<5，故OF≠SF

(有或无符号)加法器所有运算电路的核心

重要认识1：计算机中所有加减运算都基于加法器实现！

重要认识2：加法器不知道所运算的是带符号数还是无符号数。

重要认识3：(n位)加法器不判定对错，总是取低n位作为结果，并生成标志信息。



OF：OF=$C_n \oplus C_{n-1}$；或若A与B'同号,但与Sum不同号，则1；否则0

SF：sum的符号位

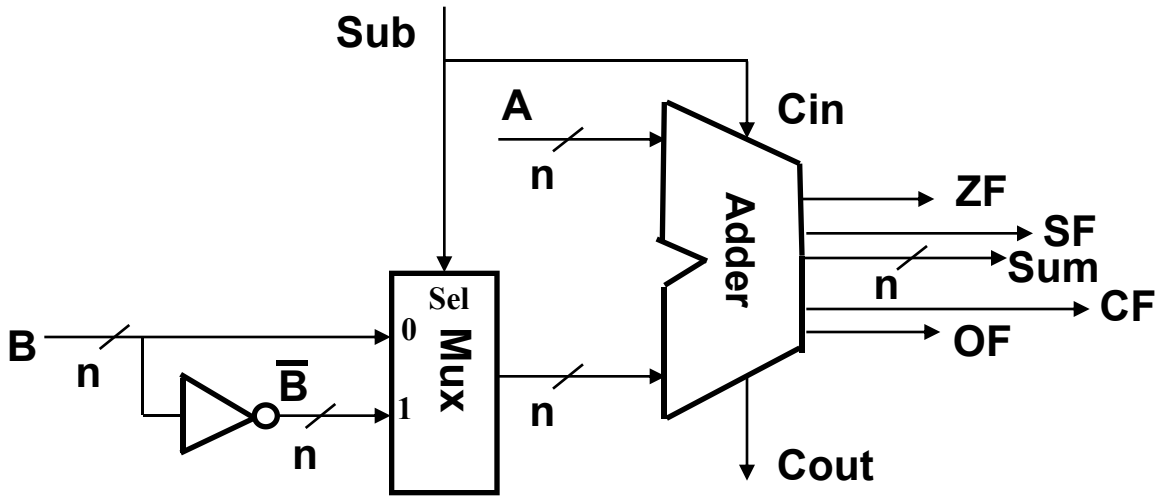
ZF：如Sum为0，则ZF=1；否则ZF=0。

CF：CF=Cout ⊕ sub; CF是进/借位, 不是低位向更高位的进位位Cout！！

整数加减运算及其部件

假定在字长n=8的机器上,用8位来保存以下变量,并且做以下运算:

```
unsigned int x=134;
unsigned int y=246;
int m=x;
int n=y;
unsigned int z1=x-y;
unsigned int z2=x+y;
int k1=m-n;
int k2=m+n;
```



x和m的机器数一样：1000 0110，y和n的机器数一样：1111 0110

z1和k1的机器数一样：1001 0000，CF=1，OF=0，SF=1无符号加/减溢出条件：进/借位CF=1

z1的值为144（=134-246+256，x-y<0），k1的值为-112。

z2和k2的机器数一样：0111 1100，CF=1，OF=1，SF=0无符号加/减溢出条件：进/借位CF=1

z2的值为124（=134+246-256，x+y>256）

k2的值为124（=134+246-256，m+n>128，即正溢出）

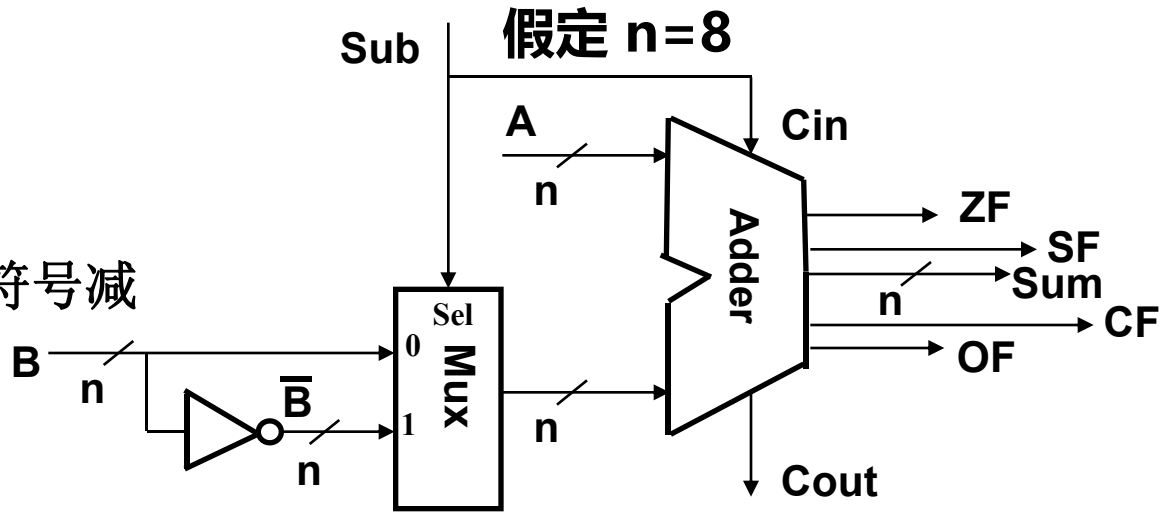
结果说明什么?

仅k1的值正确!

整数减法举例(1)

```
unsigned int x=134;
unsigned int y=246;
int m=x;
int n=y;
unsigned int z1=x-y; //无符号减
unsigned int z2=x+y;
int k1=m-n;
int k2=m+n;
```

无符号和带符号加减运算都用该部件执行



x和m的机器数一样：1000 0110，y和n的机器数一样：1111 0110

z1和k1的机器数一样：1001 0000，CF=1，OF=0，SF=1

(因为无符号加/减溢出条件:进/借位CF=1, 所以无符号减得到的z1的值溢出)

z1的值为144（=134-246+256，x-y<0），k1的值为-112。

无符号减公式：

$$result=\begin{cases} x-y & (x-y>0) \\ x-y+2^n & (x-y<0) \end{cases}$$ 溢出

带符号减公式：

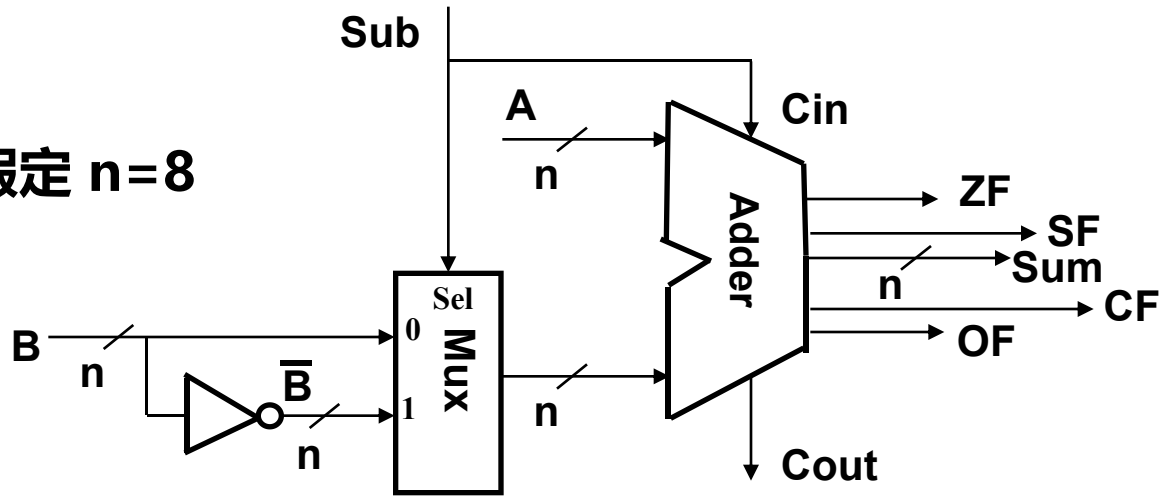
$$result=\begin{cases} x-y-2^n & (2^{n-1}\leqslant x-y) \quad \text{正溢出} \\ x-y & (-2^{n-1}\leqslant x-y<2^{n-1}) \quad \text{正常} \\ x-y+2^n & (x-y<-2^{n-1}) \quad \text{负溢出} \end{cases}$$

整数加法举例(2)

```
unsigned int x=134;
unsigned int y=246;
int m=x;
int n=y;
unsigned int z1=x-y;
unsigned int z2=x+y;
int k1=m-n;
int k2=m+n;
```

无符号和带符号加减运算都用该部件执行

假定 n=8



x和m的机器数一样：1000 0110，y和n的机器数一样：1111 0110

z2和k2的机器数一样：0111 1100，CF=1，OF=1，SF=0

z2的值为124（=134+246-256，x+y>256）

k2的值为124（=134+246-256，m+n>128，即正溢出）

无符号加公式：

$$\text{result}=\begin{cases} x+y & (x+y<2^n) \\ x+y-2^n & (2^n\leqslant x+y<2^{n+1}) \end{cases}$$ 溢出

带符号加公式：

$$\text{result}=\begin{cases} x+y-2^n & (2^{n-1}\leqslant x+y) \quad \text{正溢出} \\ x+y & (-2^{n-1}\leqslant x+y<2^{n-1}) \quad \text{正常} \\ x+y+2^n & (x+y<-2^{n-1}) \quad \text{负溢出} \end{cases}$$

无符号整数加法溢出判断程序

如何用程序判断一个无符号数相加没有发生溢出

$$\text{result}=\begin{cases} x+y & (x+y<2^n) \\ x+y-2^n & (2^n \leqslant x+y<2^{n+1}) \end{cases}$$ 溢出

发生溢出时，一定满足 (result<x) or (result<y)

否则，若x+y-2^n≥x，则 y≥2^n，这是不可能的！所以溢出！

```
/* Determine whether arguments can be added
without overflow */

int uadd_ok(unsigned x, unsigned y)
{
    unsigned sum = x+y;
    return sum >= x;
}
```

带符号整数加法溢出判断程序

如何用程序判断一个带符号整数相加没有发生溢出

$$\text{result}=\begin{cases} x+y-2^{n} & (2^{n-1}\leqslant x+y) & \text{正溢出} \\ x+y & (-2^{n-1}\leqslant x+y<2^{n-1}) & \text{正常} \\ x+y+2^{n} & (x+y<-2^{n-1}) & \text{负溢出} \end{cases}$$

```
/* Determine whether arguments can be added without
   overflow */

int tadd_ok(int x, int y) {
    int sum = x+y;
    int neg_over = x < 0 && y < 0 && sum >= 0;
    int pos_over = x >= 0 && y >= 0 && sum < 0;
    return !neg_over && !pos_over;
}
```

带符号整数减法溢出判断程序

以下程序检查带符号整数相减是否溢出有没有问题?

无符号整数减 $result=\begin{cases} x-y & (x-y>0) \\ x-y+2^n & (x-y<0) \end{cases}$ **溢出**

带符号整数减 $result=\begin{cases} x-y-2^n & (2^{n-1}\leqslant x-y) & \text{正溢出} \\ x-y & (-2^{n-1}\leqslant x-y<2^{n-1}) & \text{正常} \\ x-y+2^n & (x-y<-2^{n-1}) & \text{负溢出} \end{cases}$

```
/* Determine whether arguments can be subtracted
without overflow.WARNING: This code is buggy  */

int tsub_ok(int x, int y) {
      return tadd_ok(x, -y);
}
```

当x=0，y=0x80000000时，该函数判断错误

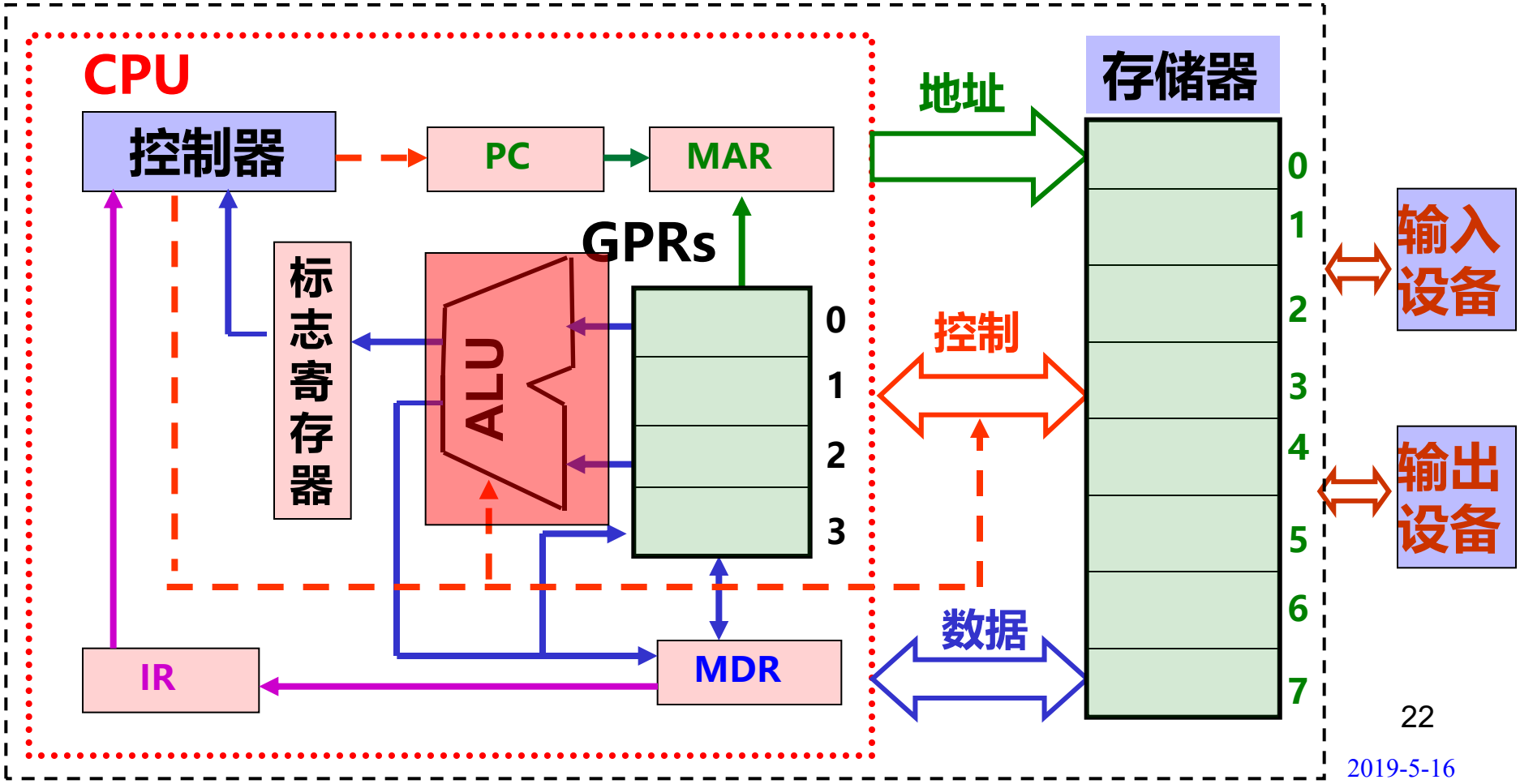
带符号(或无符号)减的溢出判断函数如何实现呢?

回顾：认识计算机中最基本的部件

CPU：中央处理器；PC：程序计数器；MAR：存储器地址寄存器

ALU：算术逻辑部件；IR：指令寄存器；MDR：存储器数据寄存器

GPRs：通用寄存器组（由若干通用寄存器组成）



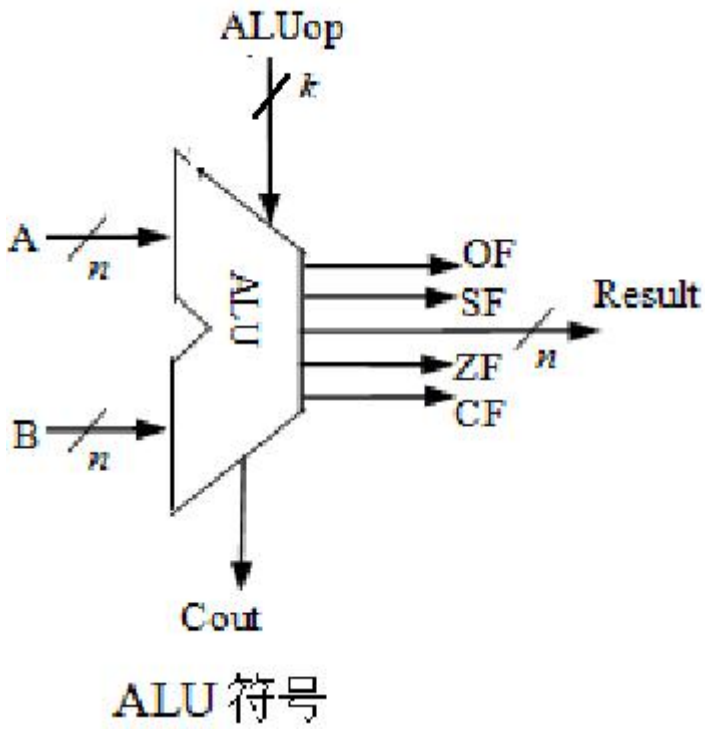
算术逻辑部件（ALU）

- 在整数加/减运算部件基础上，加上寄存器、移位器以及控制逻辑，就可实现ALU、乘/除运算以及浮点运算电路
- 进行基本算术运算与逻辑运算

  无符号整数加、减

  带符号整数加、减

  与、或、非、异或等逻辑运算



ALU 符号

- 有一个操作控制端（ALUop）来决定ALU能执行的处理功能，原来加法器中的Cin信号变成了ALU模块内部的信号。ALUop的位数k决定了操作的种类，当k=3时，ALU最多有2^3种操作。

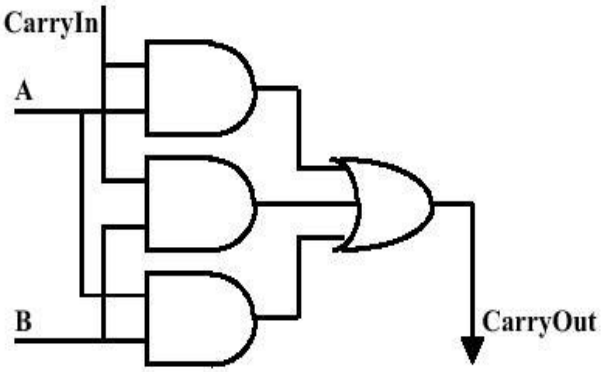
ALU的核心电路是带标志加法器，输出和/差等，还有标志信息

ALUop	Result	ALUop	Result	ALUop	Result	ALUop	Result
0 0 0	(有符号)A加B	0 1 0	A与B	1 0 0	A取反	1 1 0	(无符号)A加B
0 0 1	(有符号)A减B	0 1 1	A或B	1 0 1	A⊕B	1 1 1	(无符号)A减B

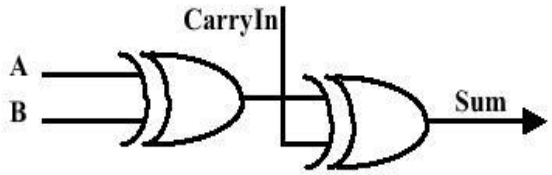
回顾：串行进位加法器

CarryOut 和 Sum 的逻辑图

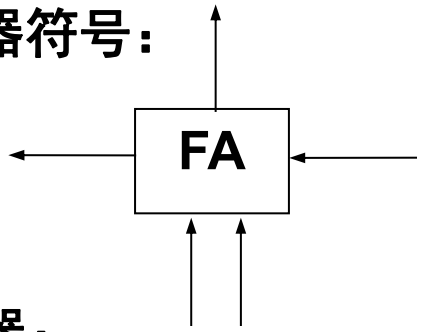
- CarryOut = B & CarryIn | A & CarryIn | A & B



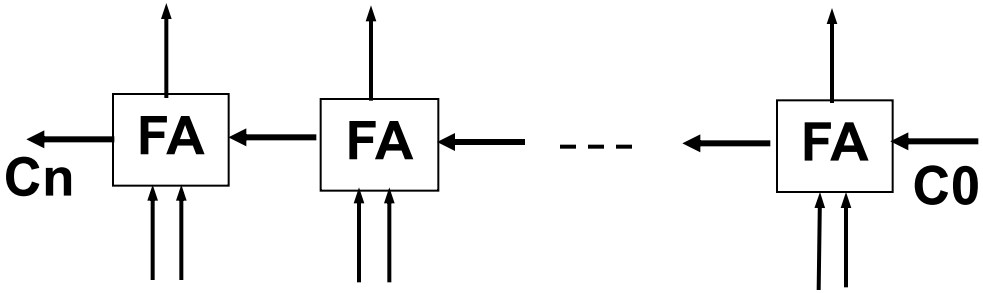
- Sum = A XOR B XOR CarryIn



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！

假定与/或门延迟为1ty，异或门3ty，则“和”与“进位”的延迟为多少？

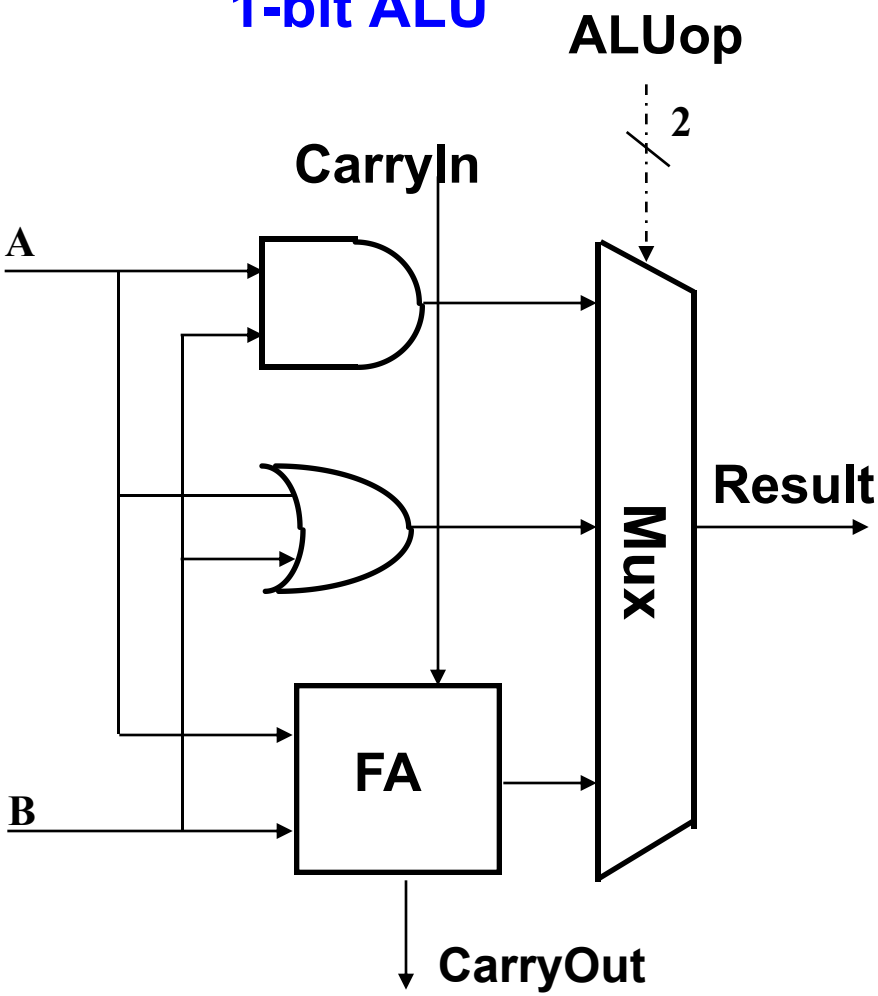
XOR:3ty延迟, 与或门各1ty延迟. Sum延迟为6ty；Carryout延迟为2ty

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=2(n-1) +3=2n+1级门延迟！

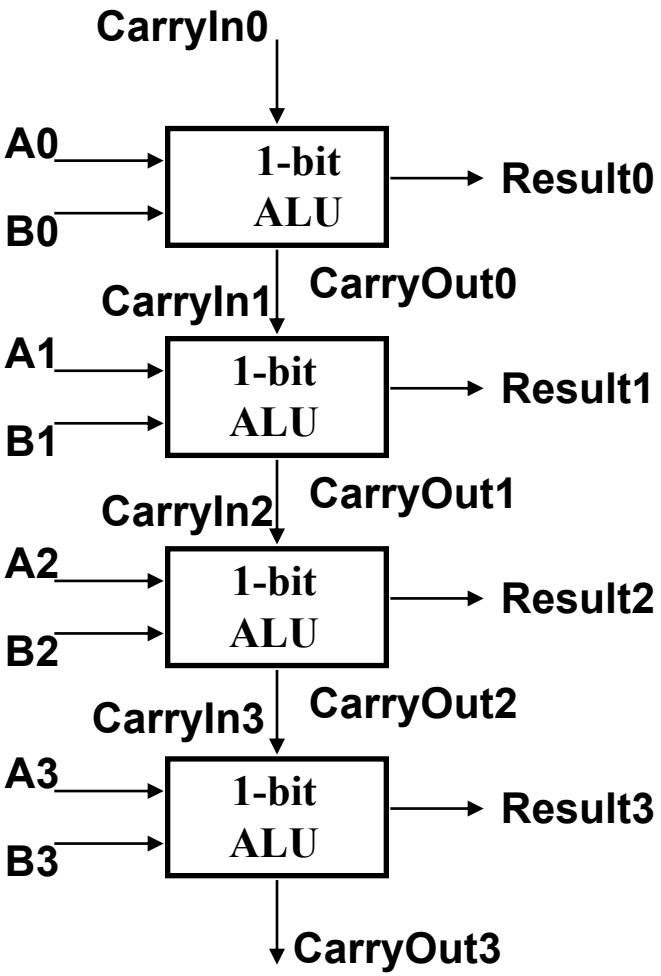
A 4-bit ALU

1-bit ALU



MUX是什么？

4-bit串行 ALU



关键路径延迟长，速度慢！

并行进位加法器(Carry Lookahead Adder,CLA,先行进位加法器)

- 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?

1) 定义进位生成/传递函数：进位生成函数 $G_i=A_iB_i$

进位传递函数$P_i=A_i+B_i$ （或 $P_i=A_i \oplus B_i$ ）

通常把实现上述逻辑的电路称为进位生成/传递部件

2)定义4位CLA部件逻辑:由全加逻辑方程：$S_i=A_i \oplus B_i \oplus C_i, C_{i+1}=G_i+P_iC_i, (i=0,\ldots n)$

设n=4(位),则：$C_1=G_0+P_0C_0$

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

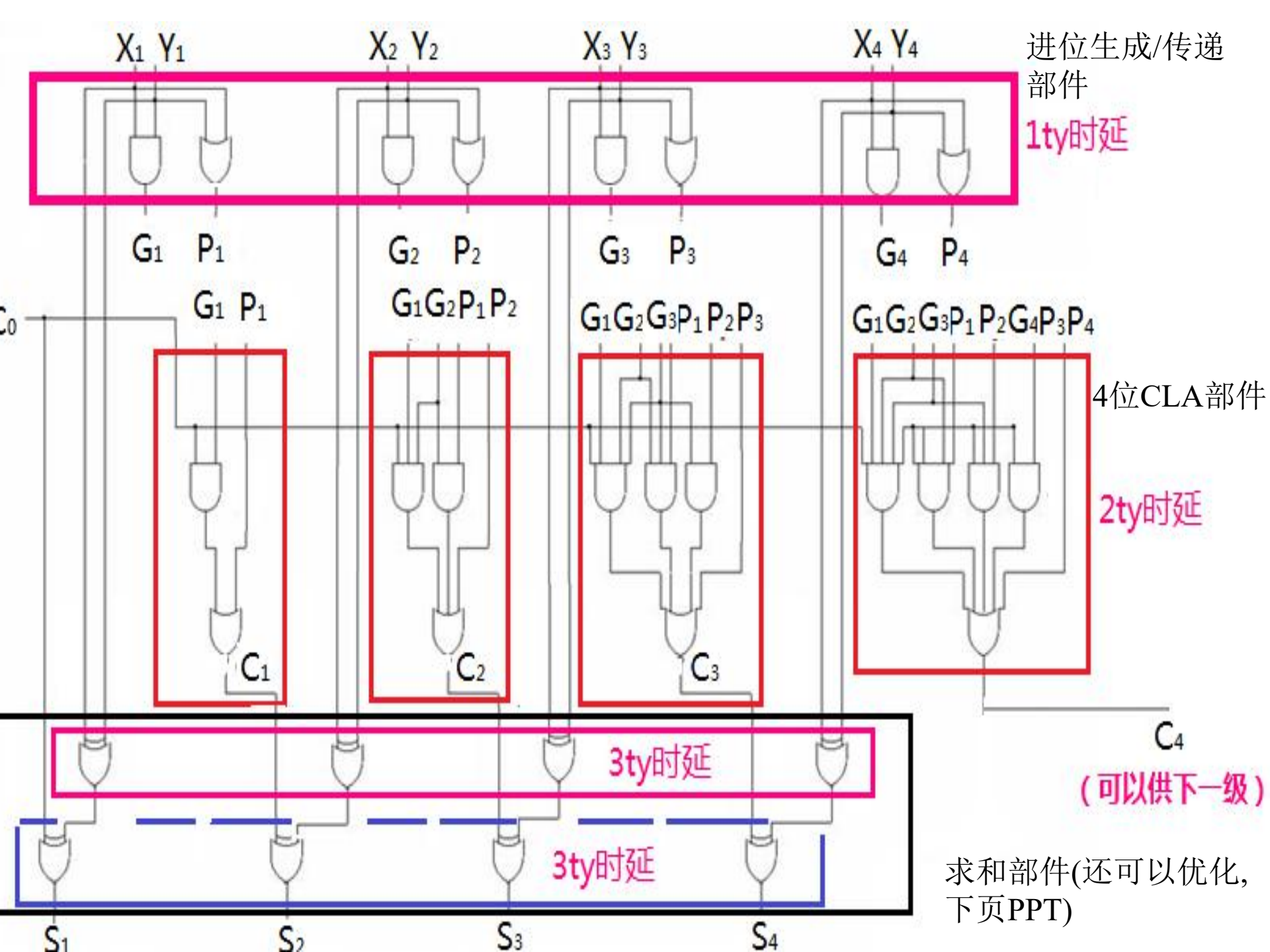
$C_3=G_2+P_2C_2=G_2+P_2G_1+P_2P_1G_0+P_2P_1P_0C_0$

$C_4=G_3+P_3C_3=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0+P_3P_2P_1P_0C_0$

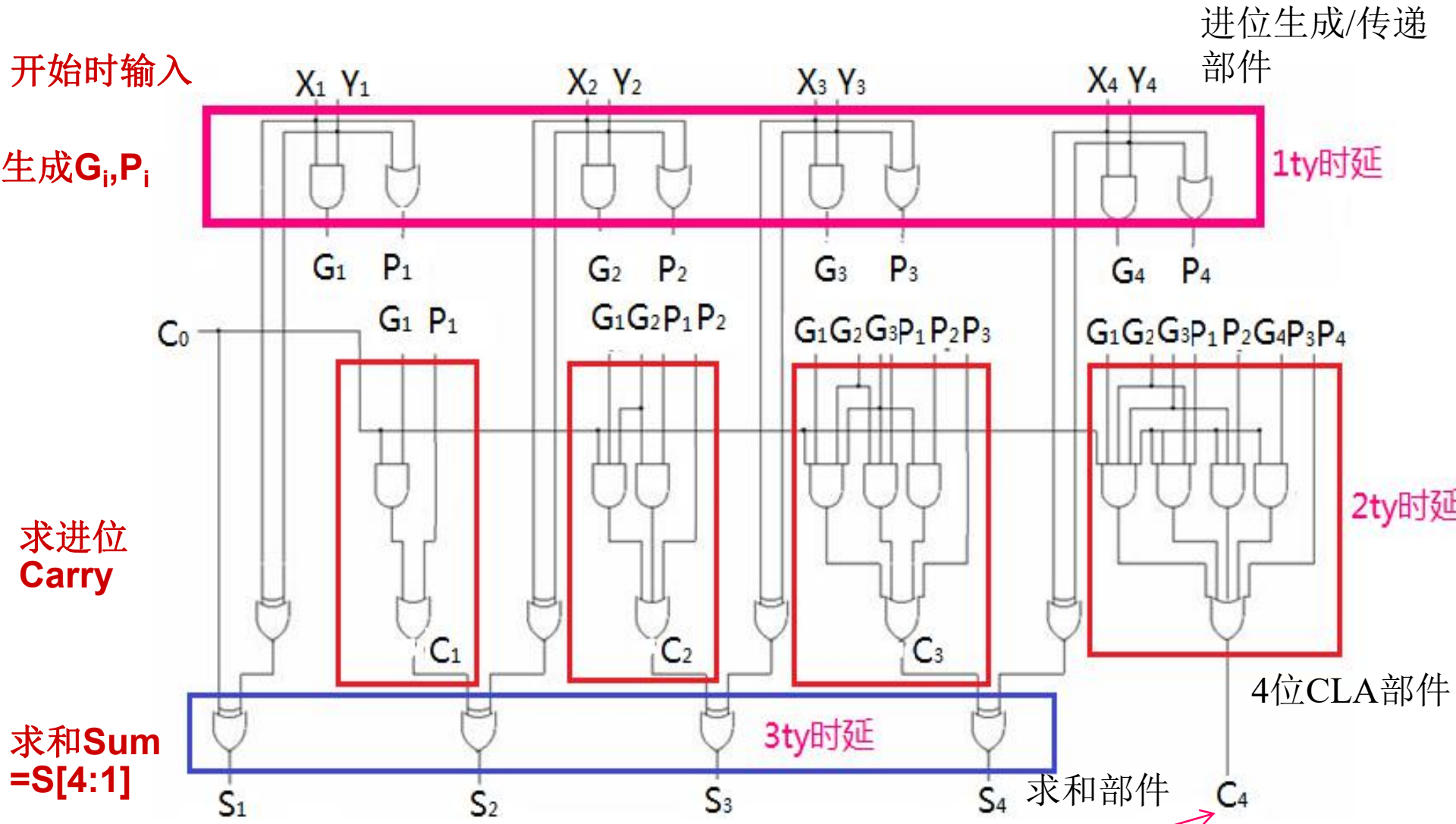
由以上4式可知:各进位之间无等待，相互独立并同时产生。

3)根据$S_i=A_i \oplus B_i \oplus C_i$ ，可并行求出各位和,实现这个求和的电路即求和部件

CLA加法器由“进位生成/传递部件”、“CLA部件”和“求和部件”构成。

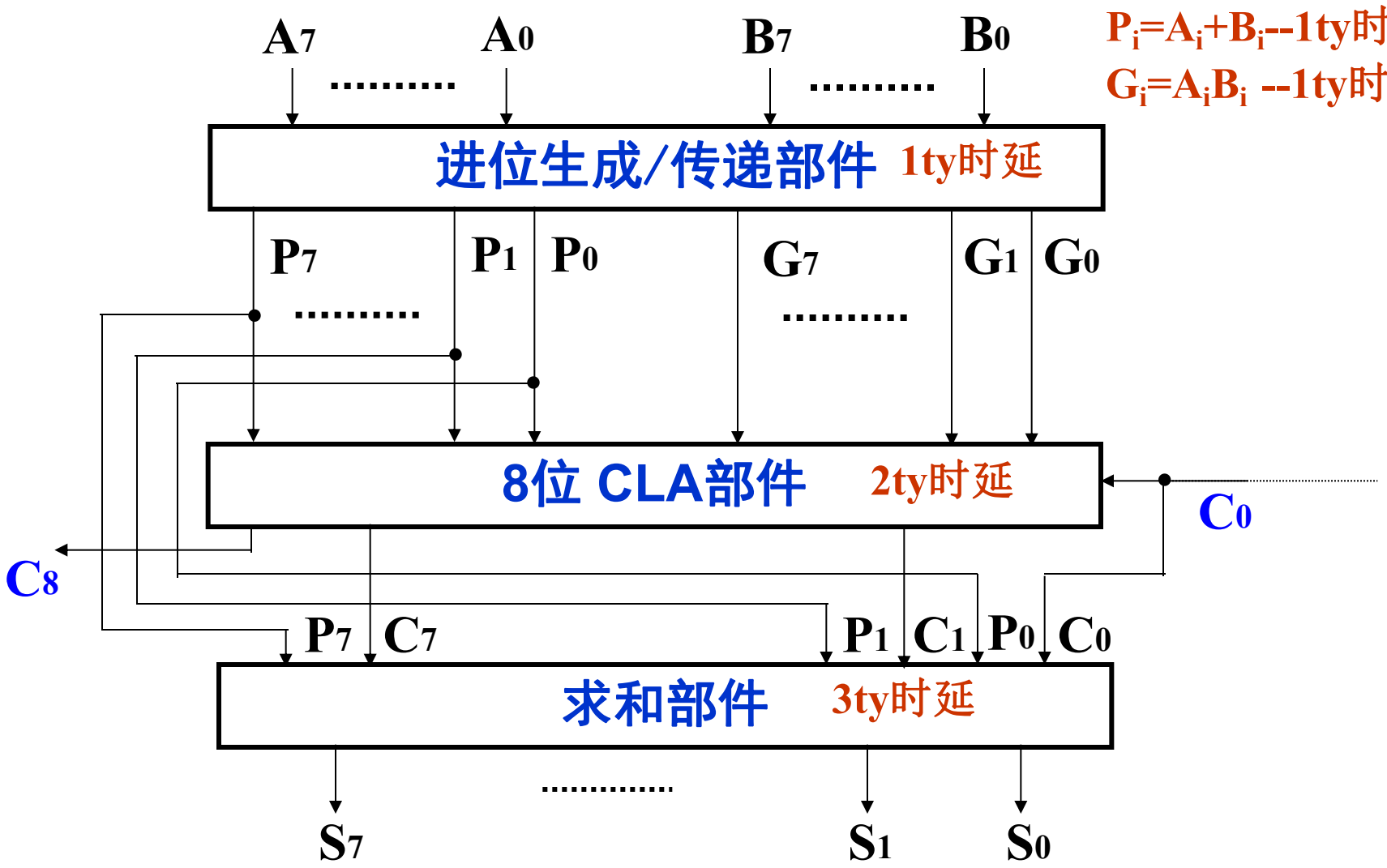

X1 Y1
X2 Y2
X3 Y3
X4 Y4
进位生成/传递部件
1ty时延
G1 P1
G2 P2
G3 P3
G4 P4
C0
G1 P1
G1G2P1P2
G1G2G3P1P2P3
G1G2G3P1P2G4P3P4
4位CLA部件
2ty时延
C1
C2
C3
C4
（可以供下一级）
3ty时延
3ty时延
求和部件(还可以优化，下页PPT)
S1
S2
S3
S4

4位全先行进位加法器(CLA)



求和S[4:1]的总延迟：1+2+3=6ty；进位C_4的延迟：1+2=3ty

8位全先行进位加法器



和的总延迟多少？进位C_8的延迟多少？

和的总延迟：1+2+3=6ty；进位C_8的延迟：1+2=3ty

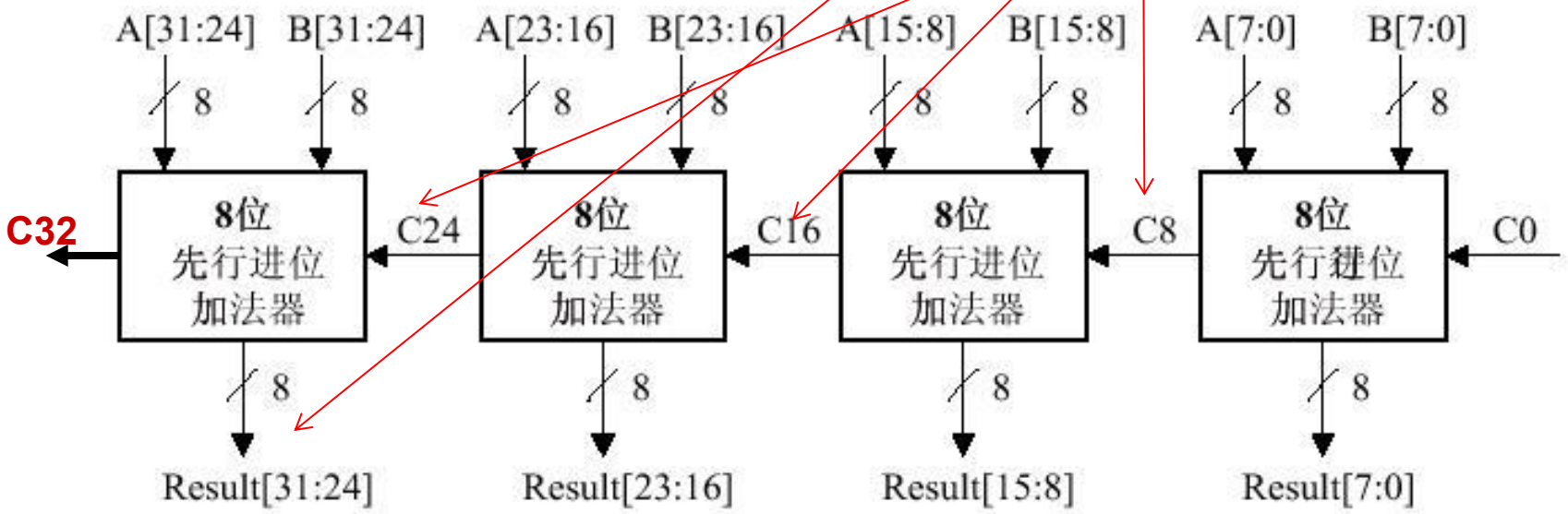
局部（单级）先行进位加法器

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？(3+2)+2+2+3=12ty



多级先行进位加法器

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：$C_1=G_0+P_0C_0$

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

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

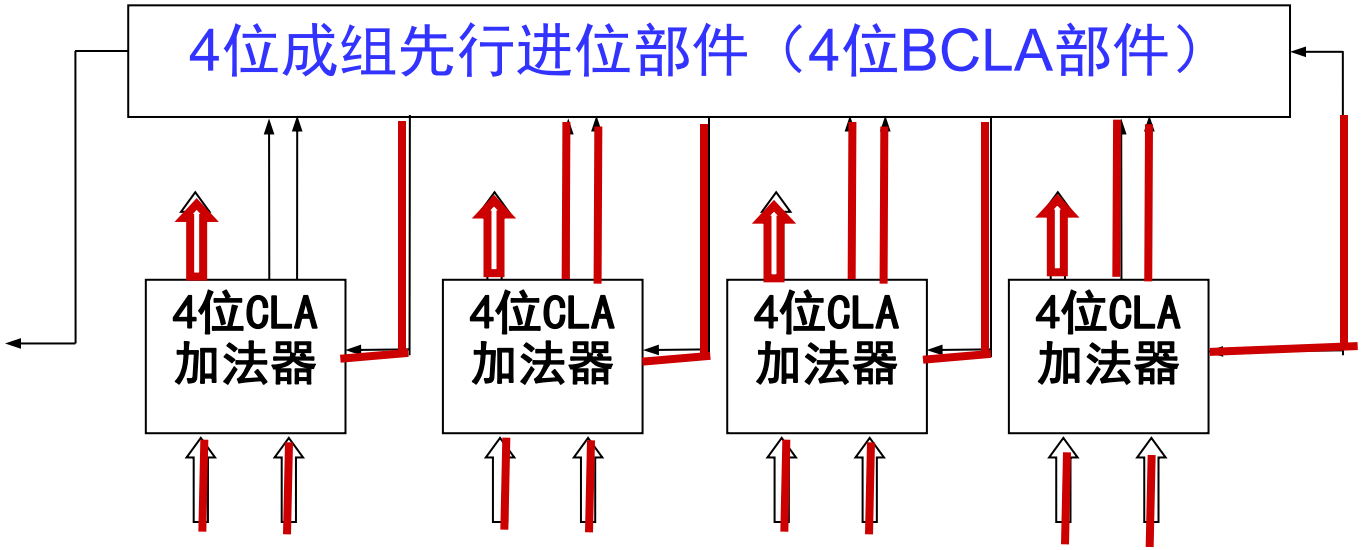
$G_3^*=G_3+P_3C_3=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0$

$P_3^*=P_3P_2P_1P_0$

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。

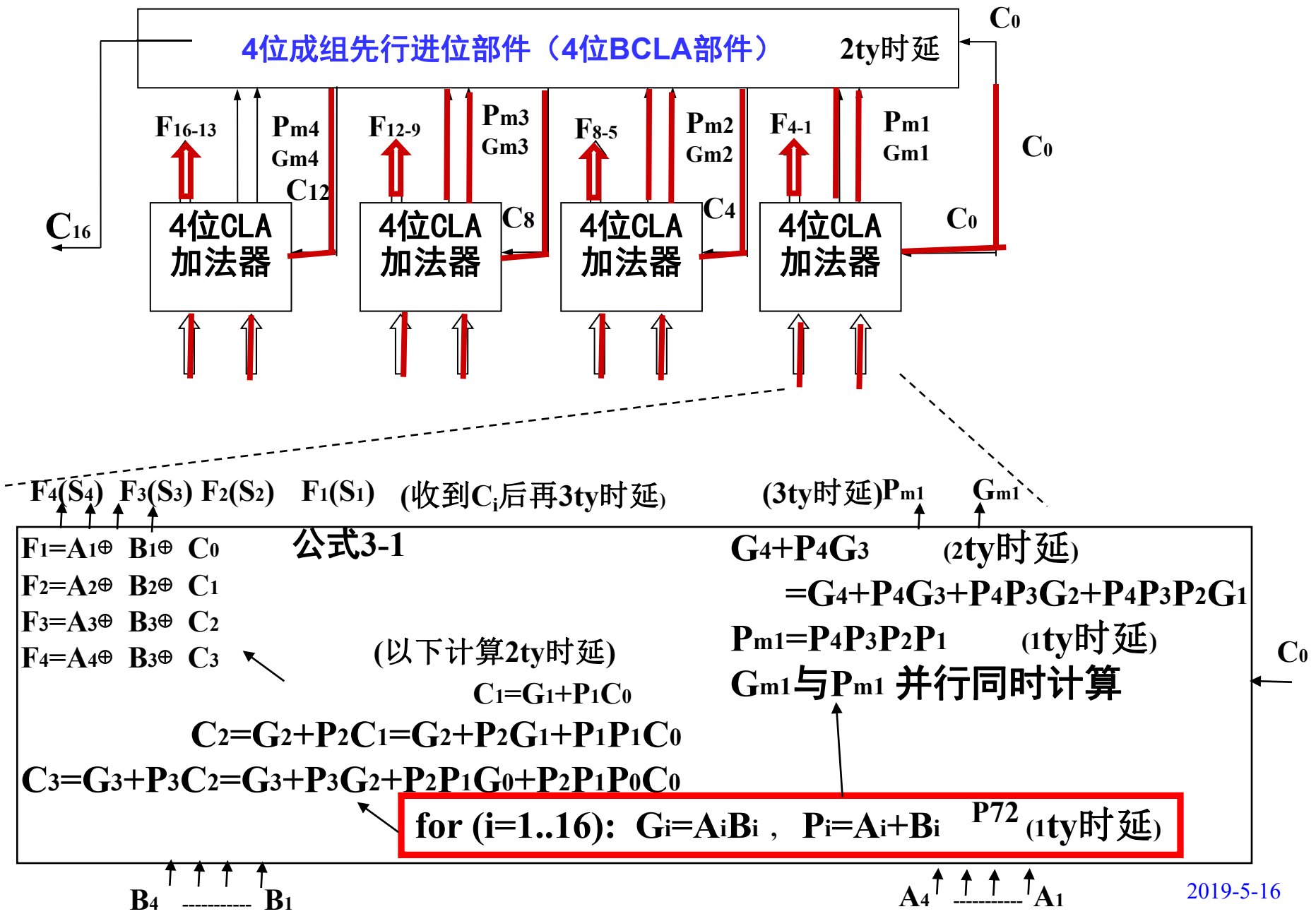
回顾：多级先行进位加法器

16位两级先行进位加法器



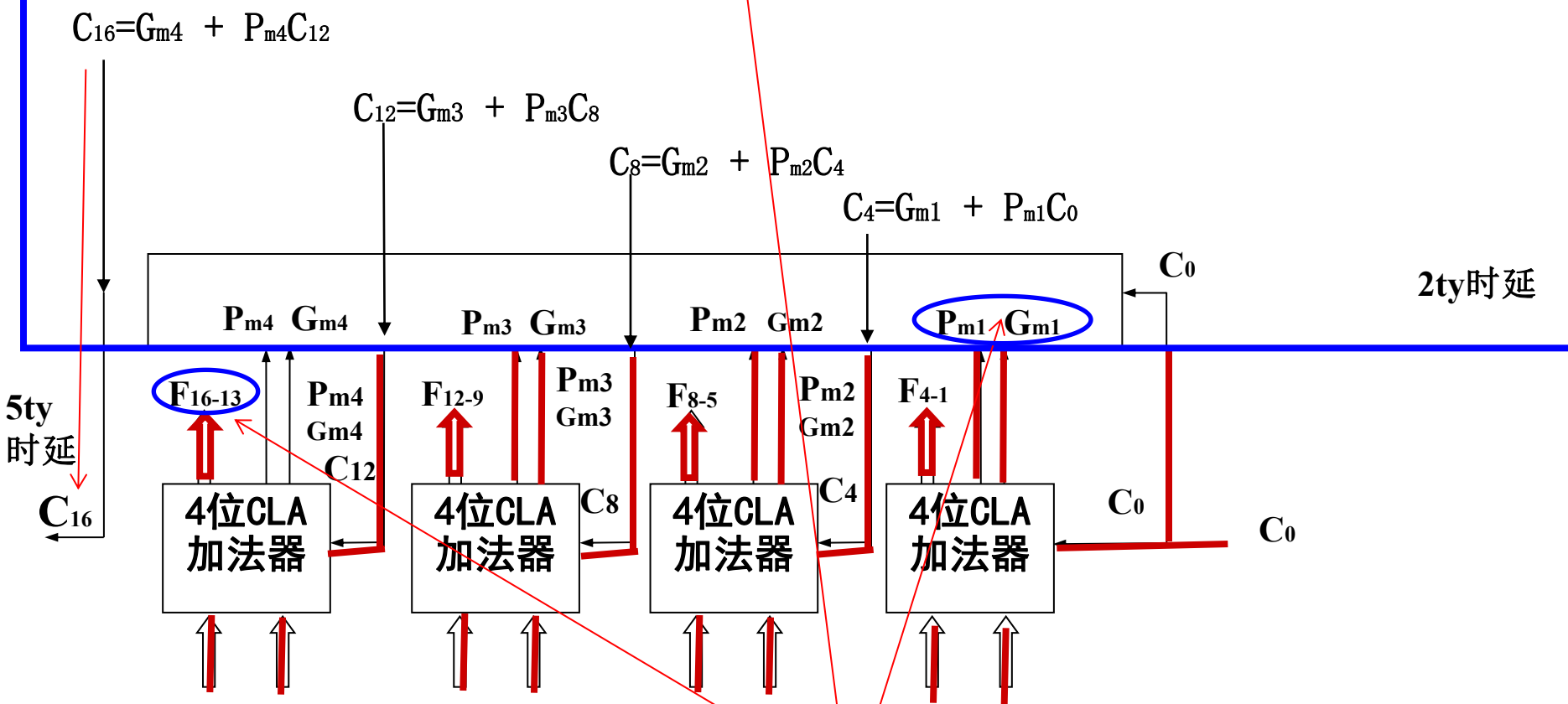
关键路径长度为多少？ 3+2+3 = 8ty

最终进位的延迟为多少？ 3+2=5ty


4位成组先行进位部件（4位BCLA部件）
2ty时延
C0
F16-13
Pm4
Gm4
C12
F12-9
Pm3
Gm3
C8
F8-5
Pm2
Gm2
C4
F4-1
Pm1
Gm1
C0
C16
4位CLA
加法器
4位CLA
加法器
4位CLA
加法器
4位CLA
加法器
C0
F4(S4) F3(S3) F2(S2) F1(S1) (收到Ci后再3ty时延)
(3ty时延)Pm1
Gm1
F1=A1⊕ B1⊕ C0
F2=A2⊕ B2⊕ C1
F3=A3⊕ B3⊕ C2
F4=A4⊕ B3⊕ C3
公式3-1
G4+P4G3 (2ty时延)
=G4+P4G3+P4P3G2+P4P3P2G1
Pm1=P4P3P2P1 (1ty时延)
Gm1与Pm1 并行同时计算
C0
(以下计算2ty时延)
C1=G1+P1C0
C2=G2+P2C1=G2+P2G1+P1P1C0
C3=G3+P3C2=G3+P3G2+P2P1G0+P2P1P0C0
for (i=1..16): Gi=AiBi , Pi=Ai+Bi P72 (1ty时延)
B4 ---------- B1
A4 ---------- A1

多级先行进位加法器（16位两级先行进位加法器）

4位成组先行进位部件(4位BCLA部件)同时得到下面四个进位 C_4, C_8, C_{12}, C_{16} . 2ty时延(见下页PPT)



关键路径长度为多少？ 3+2+3 = 8ty

最终进位的延迟为多少？ 3+2=5ty

4位成组先行进位部件（4位BCLA部件）

$C_{16} = G_{m4} + P_{m4}C_{12} = G_{m4} + P_{m4}(G_{m3} + P_{m3}G_{m2} + P_{m3}P_{m2}C_4) = G_{m4} + P_{m4}G_{m3} + P_{m4}P_{m3}G_{m2} + P_{m4}P_{m3}P_{m2}C_4$

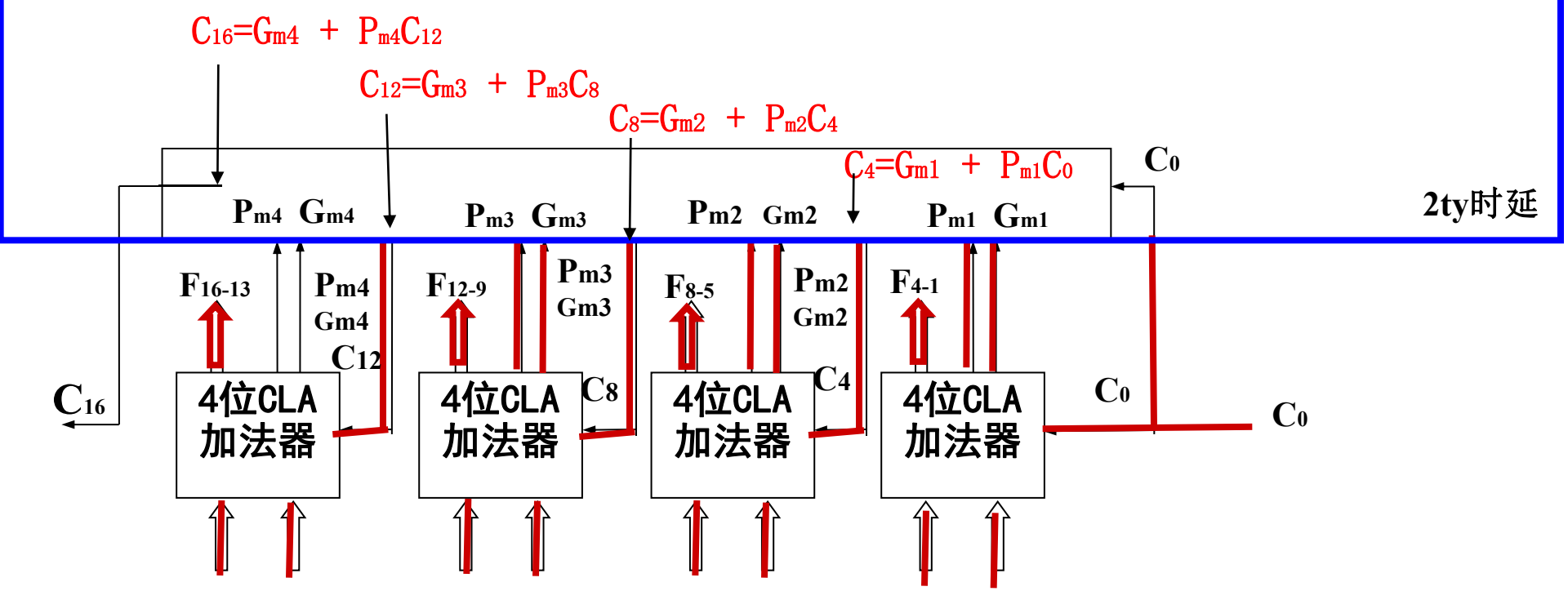
$= G_{m4}+P_{m4}G_{m3}+P_{m4}P_{m3}G_{m2}+P_{m4}P_{m3}P_{m2}(G_{m1}+P_{m1}C_0) = G_{m4} + P_{m4}G_{m3}+P_{m4}P_{m3}G_{m2} + P_{m4}P_{m3}P_{m2}G_{m1}+P_{m4}P_{m3}P_{m2}P_{m1}C_0$ （2ty时延）

$C_{12}=G_{m3} + P_{m3}C_8=G_{m3} + P_{m3}(G_{m2} + P_{m2}C_4)=G_{m3} + P_{m3}(G_{m2} + P_{m2}(G_{m1}+ P_{m1}C_0))$

$=G_{m3} + P_{m3}G_{m2} + P_{m3}P_{m2}G_{m1} + P_{m3}P_{m2}P_{m1}C_0$ （2ty时延）

$C_8=G_{m2} + P_{m2}C_4=G_{m2} + P_{m2}(G_{m1} + P_{m1}C_0)=G_{m2} + P_{m2}G_{m1} + P_{m2}P_{m1}C_0$ （2ty时延）

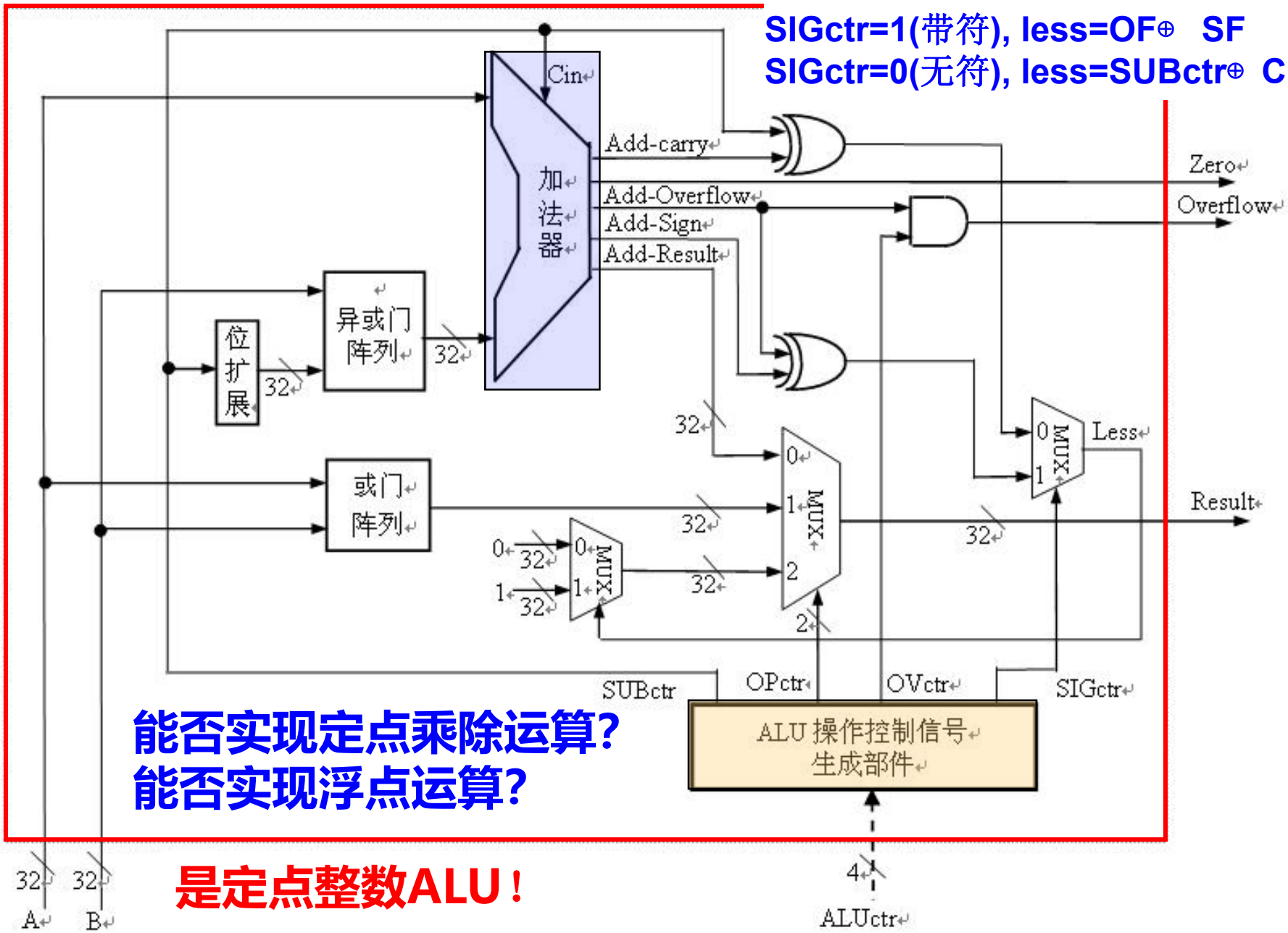
$C_4=G_{m1} + P_{m1}C_0$ （2ty时延）



关键路径长度为多少？ 3+2+3 = 8ty

最终进位的延迟为多少？ 3+2=5ty

例：实现某11条MIPS指令的ALU　P. 155 图5.13



无符号数的乘法运算

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

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

(小数点位置约定在最左边，不区分小数还是整数)

- 下面的例子中n=4,被乘数(Multiplicand),乘数(Multiplier)

被乘数 x = .1000　y_4的权重为2^{-4}

乘数 y　　* .1001 ($y=y_1y_2y_3y_4=1001$，..$y_3=0,y_4=1$)

$x\times y_4$ = 1000　y_4的权重为2^{-4},相乘得到的实际值为 $X\times y_4\times 2^{-4}$

$x\times y_3$ = 0000　　$X\times y_3\times 2^{-3}$

$x\times y_2$ = 0000　　$X\times y_2\times 2^{-2}$

$+x\times y_1$= 1000　　+　$X\times y_1\times 2^{-1}$

积)　0.1001000 即:$X\times Y=\sum_{i=1}^{4}(X\times y_i\times 2^{-i})$

整个运算过程中用到加法及左移两种操作，因而，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×yi，若yi = 0，则得0；若yi = 1，则得X
  - ② 把①求得的各项结果X× yi 逐次左移(手工法)，可表示为X× yi×2^{-i}
  - ③ 对②中结果求和，即 ∑ (X× yi×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×yi后，与前面所得结果累加，得到Pi，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×yi的开销。
  - ② 每次得X×yi后，不将它左移与前次部分积Pi相加，而将部分积Pi右移(机器法)后与X×yi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(机器法)，对为“0”的位只执行右移(机器法)，而不执行加法运算。

无符号乘法运算的算法推导-原码一位乘法

- 上述思想可写成如下数学推导过程-（以含有无符号小数为乘数）：

$$P_n = X \times Y = X \times (0.y_1 y_2 \ldots y_n)$$

$$= X \times y_1 \times 2^{-1} + X \times y_2 \times 2^{-2} + \ldots\ldots + X \times y_{n-1} \times 2^{-(n-1)} + X \times y_n \times 2^{-n}$$

(倒排序) $= 2^{-n} \times X \times y_n + 2^{-(n-1)} \times X \times y_{n-1} + \ldots\ldots + 2^{-2} \times X \times y_2 + 2^{-1} \times X \times y_1$

$= 2^{-1}(2^{-1}(2^{-1}\ldots 2^{-1}(2^{-1}(\underbrace{0}_{P_0} + X \times y_n) + X \times y_{n-1}) + \ldots\ldots + X \times y_2) + X \times y_1)$

（2^{-1} 共 n 个）

P_1

P_2

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

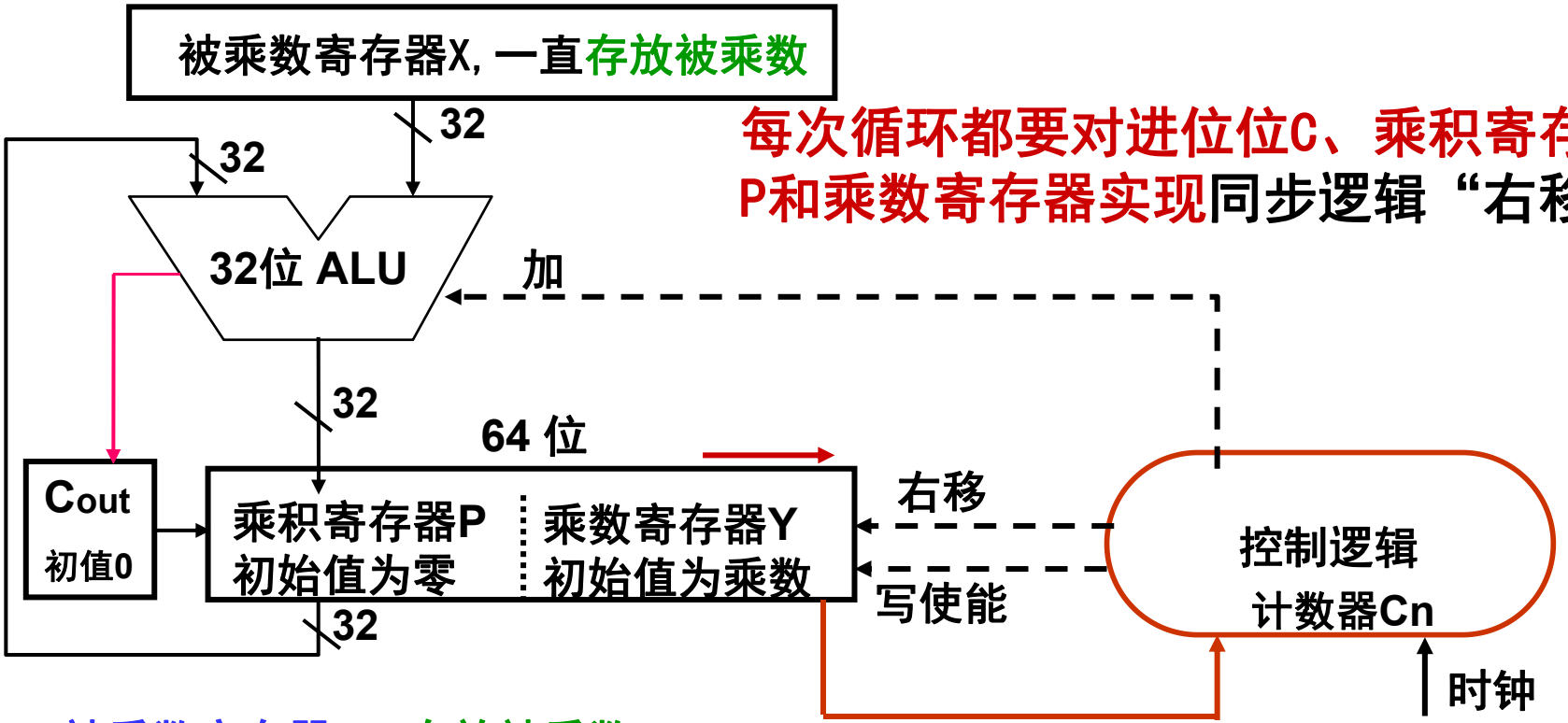
$P_2 = 2^{-1}(P_1 + X \times y_{n-1})$

- 该计算过程有明显的递归性质，设$P_0=0$，从乘数最低位y_n开始，经n次“判断–加法–右移”多次循环迭代，直到求出Pn为止。每步的乘积(作为部分积)为：
- $P_1=2^{-1}(P_0+X \times y_n)$ //y的最低位与x的所有位乘,再与0加后右移一位
- $P_2 = 2^{-1}(P_1 + X \times y_{n-1})$ //y的次低位与x的所有位乘,再与上一步得到的部分积加后右移一位
- …… ……

$P_n = 2^{-1}(P_{n-1} + X \times y_1) = X \times Y$ // 最后一步得到的是X×Y的最终结果

- 无符号乘法运算的递推公式为：$P_{i+1} = 2^{-1}(P_i + X \times y_{n-i})$ （i = 0, 1, 2, …, n-1）

32位乘法运算的硬件实现-原码一位乘法



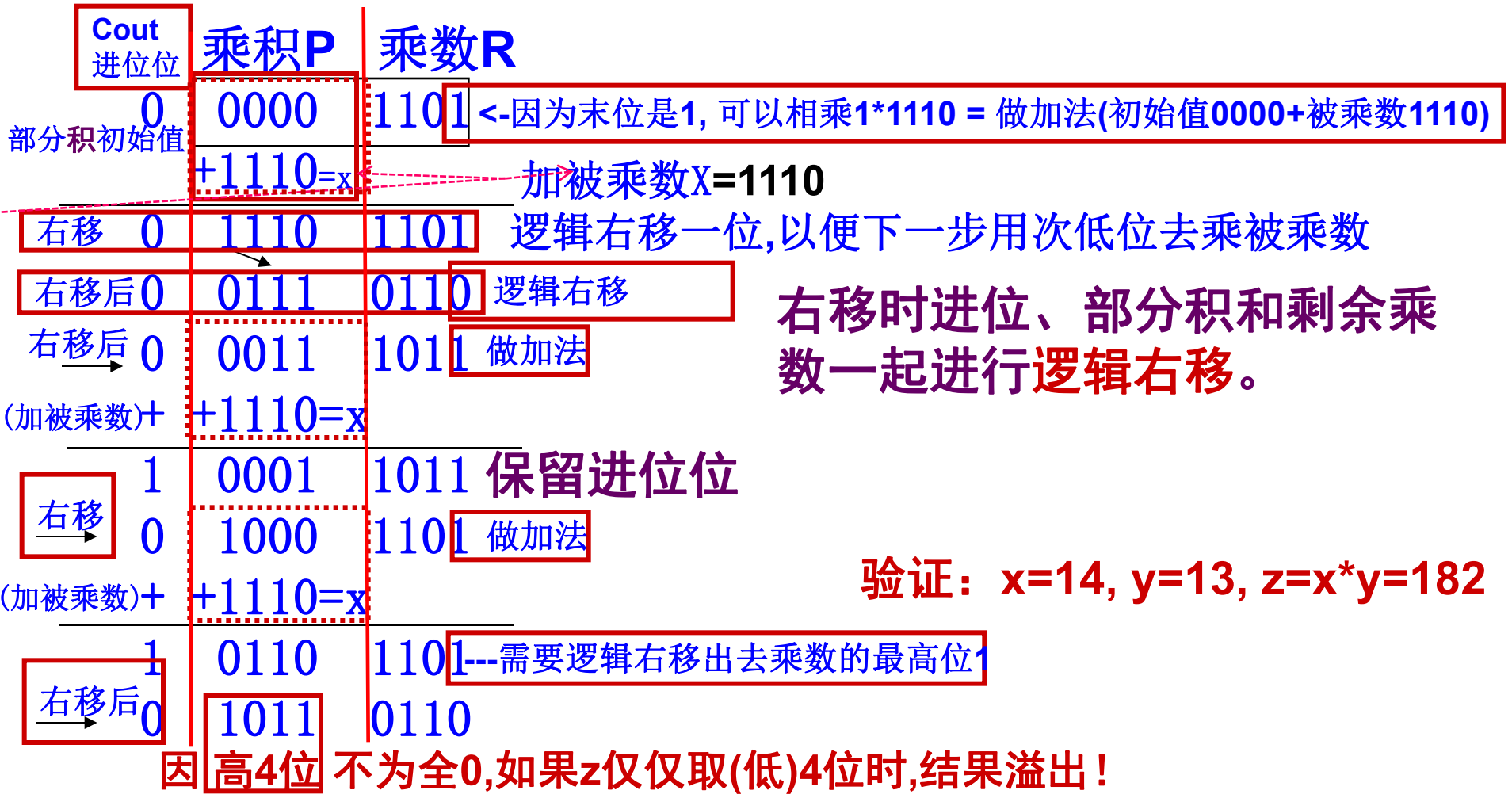
每次循环都要对进位位C、乘积寄存器P和乘数寄存器实现同步逻辑“右移”

- 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P0 = 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-原码一位乘法

举例说明：若需计算z=x*y; x、y和z都是unsigned类型。

设x=1110　y=1101　应用递推公式：$P_i=2^{-1}(x*y_i+P_{i-1})$



右移时进位、部分积和剩余乘数一起进行逻辑右移。

验证：x=14, y=13, z=x*y=182

用双倍字长的乘积寄存器,或两个单倍字长的寄存器来保留乘法的结果2n=8位

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：积符异或得到，数值用无符号乘法运算
- 例：设$[x]_{原}=0.1110$，$[y]_{原}=1.1101$，计算 $[x\times y]_{原}$
- 解：数值部分用无符号数乘法算法计算：$1110\times1101=1011\ 0110$
- 符号位：$0\oplus1=1$，所以：$[x\times y]_{原}=1.\ 10110110$
- 一位乘法：每次只取乘数的一位判断，需n次循环，速度慢。
- 两位乘法：每次取乘数两位判断(00, 01, 10, 11)，仅需n/2次循环，快一倍
- 两位乘法递推公式(浮点数尾
- 数相乘为例)：

设P_i为上次乘法结果,所取2位及对应的计算分别为:

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

y_i	y_{i-1}	T(当前值)	操　作	迭代公式
0	0	0	$0\rightarrow T$	$2^{-2}(P_i)$
0	0	1	$+X$　$0\rightarrow T$	$2^{-2}(P_i+X)$
0	1	0	$+X$　$0\rightarrow T$	$2^{-2}(P_i+X)$
0	1	1	$+2X$　$0\rightarrow T$	$2^{-2}(P_i+2X)$
1	0	0	$+2X$　$0\rightarrow T$	$2^{-2}(P_i+2X)$
1	0	1	$-X$　$1\rightarrow T$	$2^{-2}(P_i-X)$
1	1	0	$-X$　$1\rightarrow T$	$2^{-2}(P_i-X)$
1	1	1	$1\rightarrow T$	$2^{-2}(P_i)$

3X= 4X-X，本次-X，下次+4X!
部分积右移两位，相当于4X

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

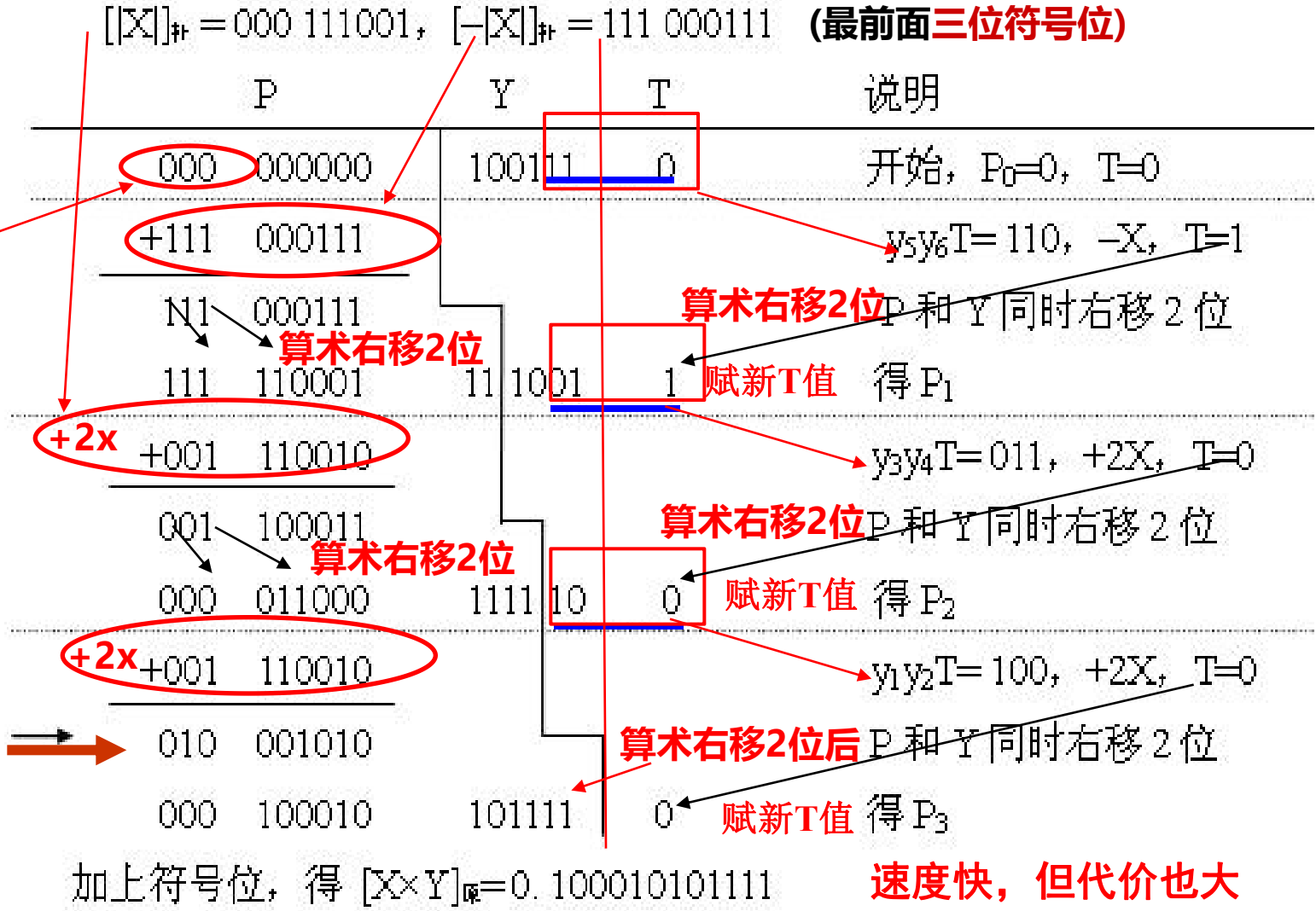
解： 先用无符号数乘法计算111001×100111，原码两位乘法过程如下：

采用补码算术右移，与一位乘法不同,为模8补码形式(三位符号位)

采用补码算术右移，与一位乘法不同

为模8补码形式(三位符号位)

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。



对带符号整数进行补码乘法运算-符号与数值统一处理

因$[X \times Y]_{补} \neq [X]_{补} \times [Y]_{补}$，故不能用无符号数乘法计算。如，若x=-5, 求x*x=?

设偶数位补码定点整数X,Y,$[X]_{补}=x_{n-1}x_{n-2}\cdots x_1x_0$，$[Y]_{补}=y_{n-1}y_{n-2}\cdots y_1y_0$，

用Booth's Algorithm求：$[X \times Y]_{补}$

基于以下补码性质：

$Y_{[n:0]}$的真值 $= -y_{n-1}\cdot 2^{n-1} + y_{n-2}\cdot 2^{n-2} + \cdots\cdots + y_1\cdot 2^1 + y_0\cdot 2^0$ 令$y_{-1}=0$，则：

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

因为 $y_{30}\cdot 2^{30} = y_{30}\cdot 2^{31} - y_{30}\cdot 2^{30}$

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

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

约定好小数点，乘32位的小数X = 乘X的数值部分再乘以(2^{-32})

$[X \times Y]_{补} = ((X^*(2^{-32}))\cdot(((y_{30}-y_{31})\cdot 2^{31} + (y_{29}-y_{30})\cdot 2^{30} + \cdots\cdots + (y_0-y_1)\cdot 2^1 + (y_{-1}-y_0)\cdot 2^0))$

$= (y_{30}-y_{31})X\cdot 2^{-1} + (y_{29}-y_{30})X\cdot 2^{-2} + \cdots\cdots + (y_0-y_1)X\cdot 2^{-31} + (y_{-1}-y_0)X\cdot 2^{-32}$

$= 2^{-1}(2^{-1}\cdots 2^{-1}(\underbrace{(2^{-1}(y_{-1}-y_0)X)}_{P_0} + (y_0-y_1)X) + \cdots + (y_{30}-y_{31})X)$ SKIP

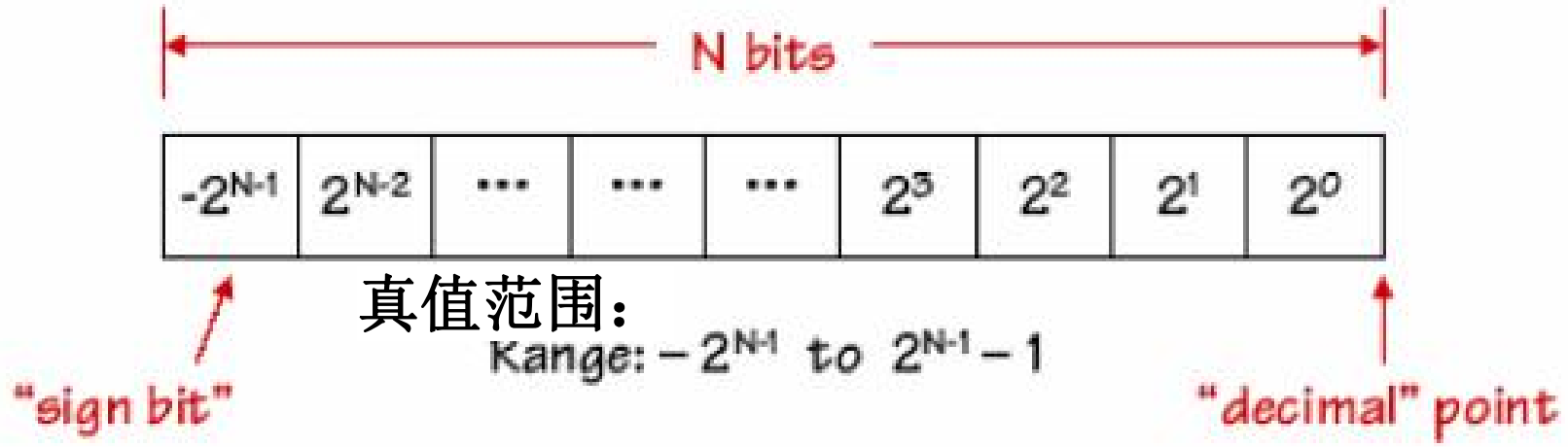
P_1

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

如何求补码的真值

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

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



8-bit 2's complement example:

$11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -128 + 64 + 16 + 4 + 2 = -42$

符号为0，则为正数，数值部分同

符号为1，则为负数，数值各位取反，末位加1

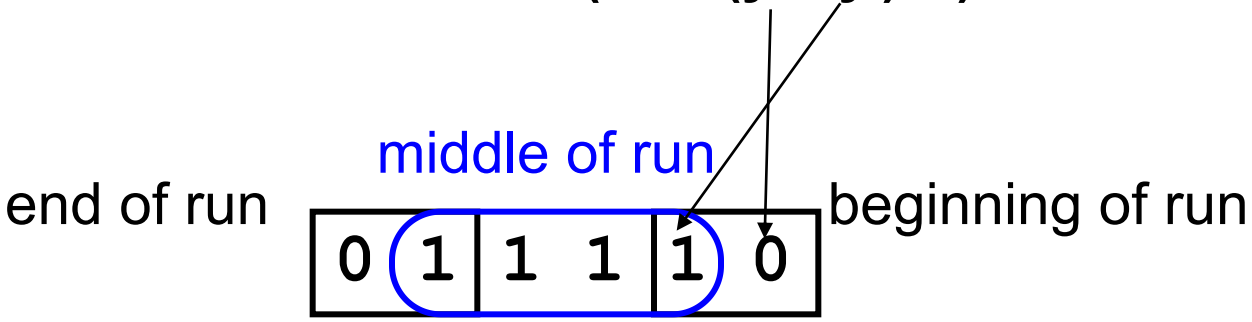
BACK

例如：补码“11010110”的真值为：-0101010=-(32+8+2)=-42

Booth's 算法实质

已经求得P_i, P=X*Y, 求 P_{i+1} 的部分积公式：

$$P_{i+1} = 2^{-1} (P_i + (y_{i-1} - y_i) X)$$



当前位y_i	右边位y_{i-1}	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。
- 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！

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

在“1串”中，第一个1时(10)做减法，最后一个1(01)做加法，其余情况只要右移1位

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

X=-3，Y=6，X×Y=-18，$[X\times Y]_{补}$应等于11101110或结果溢出

P	Y	y_{-1}	说明
0 0 0 0	0 1 1 0	0	设 $y_{-1}=0$，$[P_0]_{补}=0$
	→ 右移1位		$y_0\,y_{-1}=00$，P、Y 直接右移一位
0 0 0 0	0 0 1 1	0	减法 得$[P_1]_{补}$
+0 0 1 1			$y_1\,y_0=10$，$+[-X]_{补}$
0 0 1 1	→ 1		P、Y 同时右移一位
0 0 0 1	1 0 0 1	1	不加减 得$[P_2]_{补}$
	→ 1		$y_2\,y_1=11$，P、Y 直接右移一位
0 0 0 0	1 1 0 0	1	加法 得$[P_3]_{补}$
+1 1 0 1			$y_3\,y_2=01$，$+[X]_{补}$
1 1 0 1	→ 1		P、Y 同时右移一位
1 1 1 0	1 1 1 0	0	得$[P_4]_{补}$

如何判断结果是否溢出?

高4位是否全为符号位?不是,就溢出!

验证：当X×Y取8位时，结果 -0010010B=-18；取4位时，结果溢出

补码两位乘法

- 补码两位乘可用布斯算法推导如下：
  - $[P_{i+1}]_{补} = 2^{-1}\left([P_i]_{补} + (y_{i-1} - y_i)[X]_{补}\right)$
  - $[P_{i+2}]_{补} = 2^{-1}\left([P_{i+1}]_{补} + (y_i - y_{i+1})[X]_{补}\right)$

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

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

- 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	$2^{-2}[P_i]_{补}$

补码两位乘法举例

- 已知 $[X]_{补}$ = 1 101， $[Y]_{补}$ = 0 110，用补码两位乘法计算$[X \times Y]_{补}$。
- 解：$[-X]_{补}$= 0 011，用补码二位乘法计算$[X \times Y]_{补}$的过程如下。

P_n	P	Y	y_{-1}		说明
0	0 0 0 0	0 1 1 0	0		开始，设$y_{-1}=0$，$[P_0]_{补}=0$
+ 0	0 1 1 0				$y_1y_0y_{-1}$ =100，+2$[-X]_{补}$
0	0 1 1 0			→ 2	P和Y同时右移二位
0	0 0 0 1	1 0 0 1	1		得$[P_2]_{补}$
+ 1	1 0 1 0				$y_3y_2y_1$ = 011，+2$[X]_{补}$
1	1 0 1 1			→ 2	P和Y同时右移二位
1	1 1 1 0	1 1 1 0			得$[P_4]_{补}$

因此 [X × Y]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。

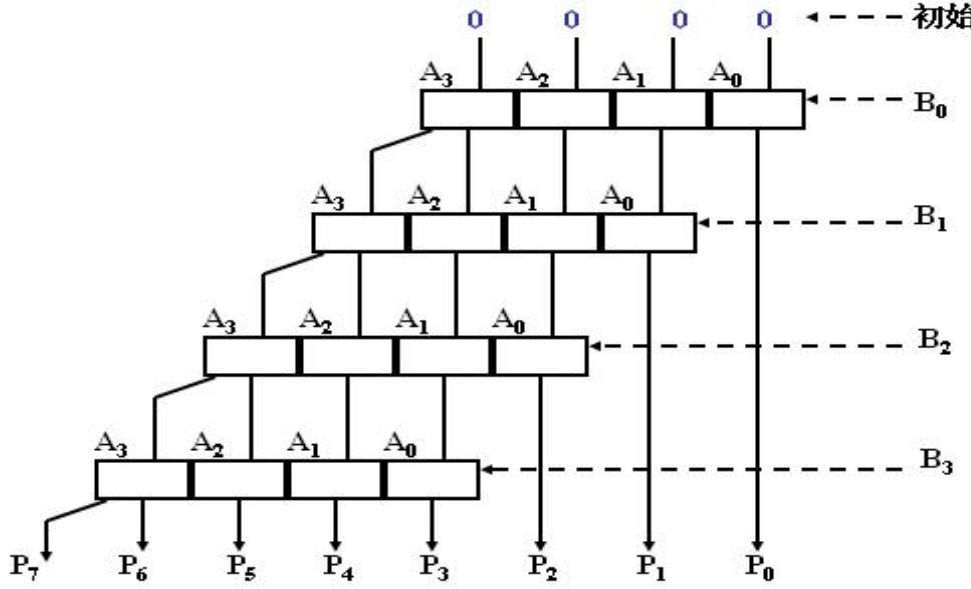
验证：-3 ×6=-18 （-10010B）

快速乘法器

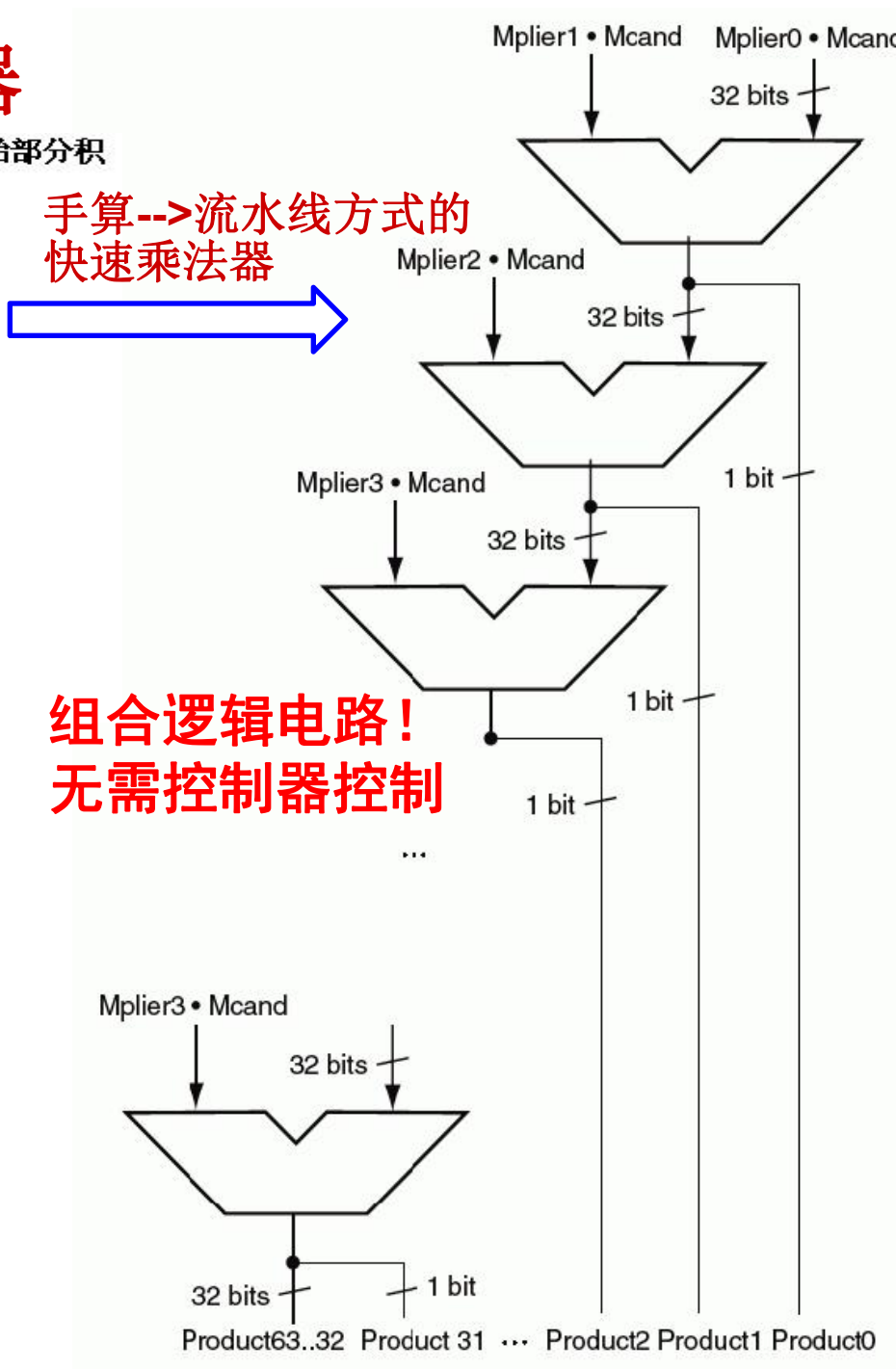
- 前面介绍的乘法部件的特点
  - 通过一个ALU多次做“加/减+右移”来实现
    - 一位乘法：约n次“加+右移”
    - 两位乘法：约n/2次“加+右移”

  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）

流水线方式的快速乘法器



手算-->流水线方式的快速乘法器

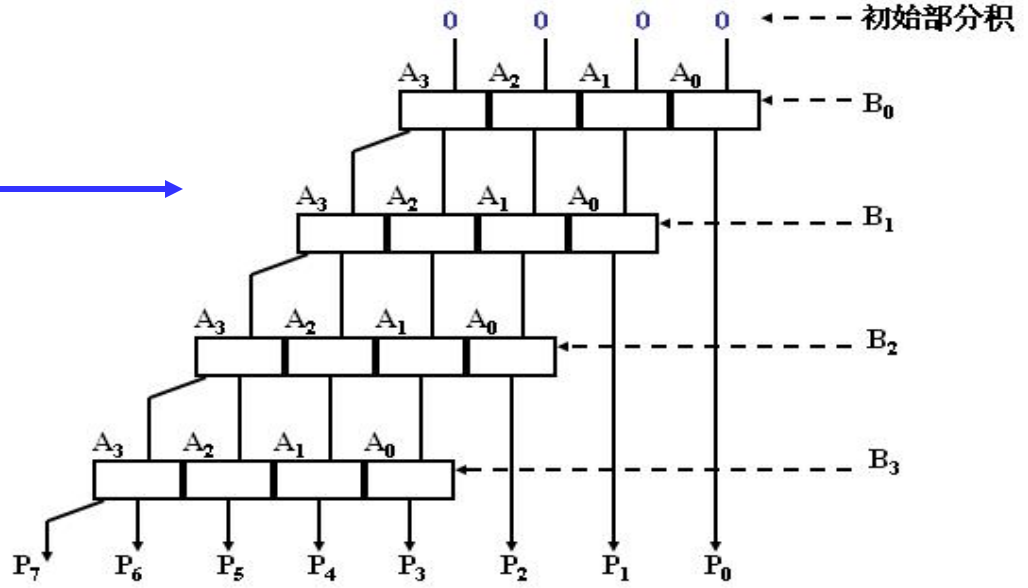


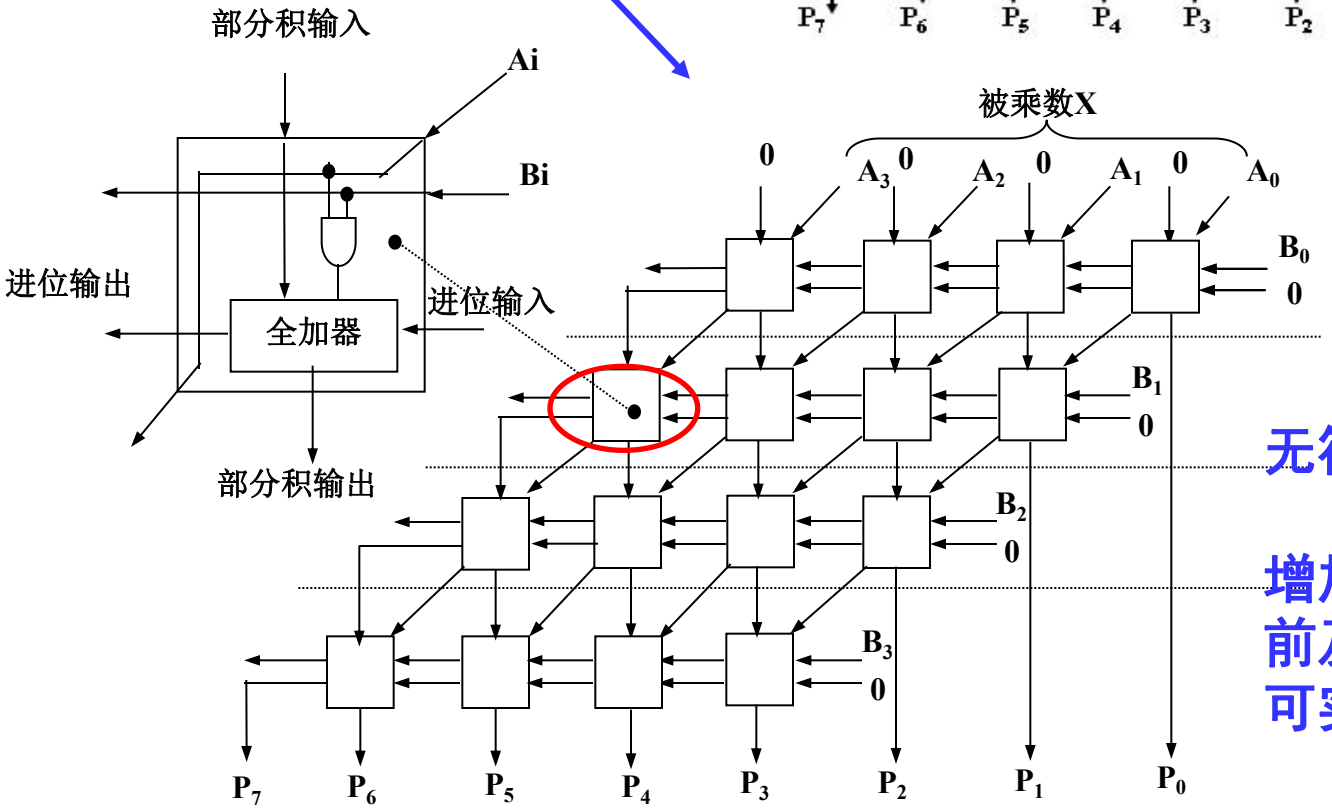
组合逻辑电路！
无需控制器控制

- 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

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

整数的乘运算

- 硬件可保留2n位乘积，故有些指令的乘积为2n位，可供软件使用
- 如果程序不采用防止溢出的措施，且编译器也不生成用于溢出处理的代码，就会发生一些由于整数溢出而带来的问题。
- 指令：分无符号数乘指令、带符号整数乘指令
- 乘法指令的操作数长度为n，而乘积长度为2n，例如：
  - IA-32中，若指令只给出一个操作数SRC，则另一个源操作数隐含在累加器AL/AX/EAX中，将SRC和累加器内容相乘，结果存放在AX（16位时）或DX-AX（32位时）或EDX-EAX（64位时）中。
  - 在MIPS处理器中，mult会将两个32位带符号整数相乘，得到的64位乘积置于两个32位内部寄存器Hi和Lo中，因此，可以根据Hi寄存器中的每一位是否等于Lo寄存器中的第一位来进行溢出判断。
- 乘法指令可生成溢出标志，编译器也可使用2n位乘积来判断是否溢出！

整数的乘运算只保留低n位结果时溢出判断

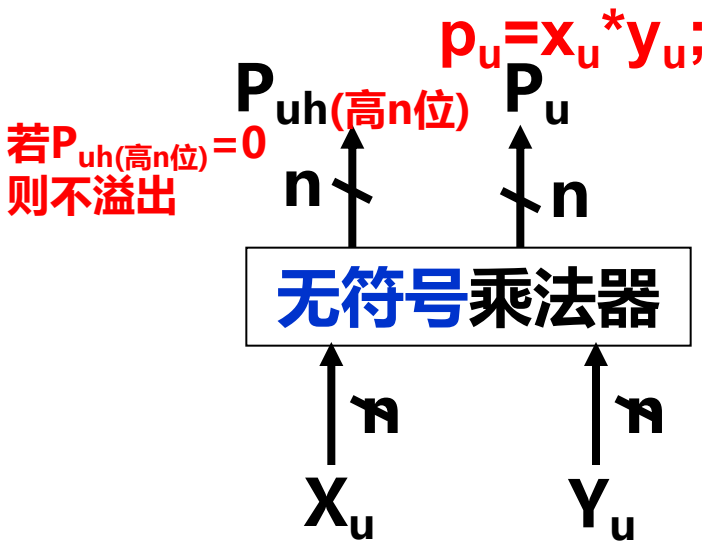
假定两个n位无符号整数x_u和y_u对应的机器数为X_u和Y_u，$p_u=x_u\times y_u$，p_u为(只)取$x_u\times y_u$ 后的低n位作为无符号整数，其对应的机器数为P_u

类似地,两个n位带符号整数x_s和y_s对应的机器数为X_s和Y_s，$p_s=x_s\times y_s$，p_s为n位带符号整数且对应的机器数为P_s。若$X_u=X_s$且$Y_u=Y_s$，则$P_u=P_s$

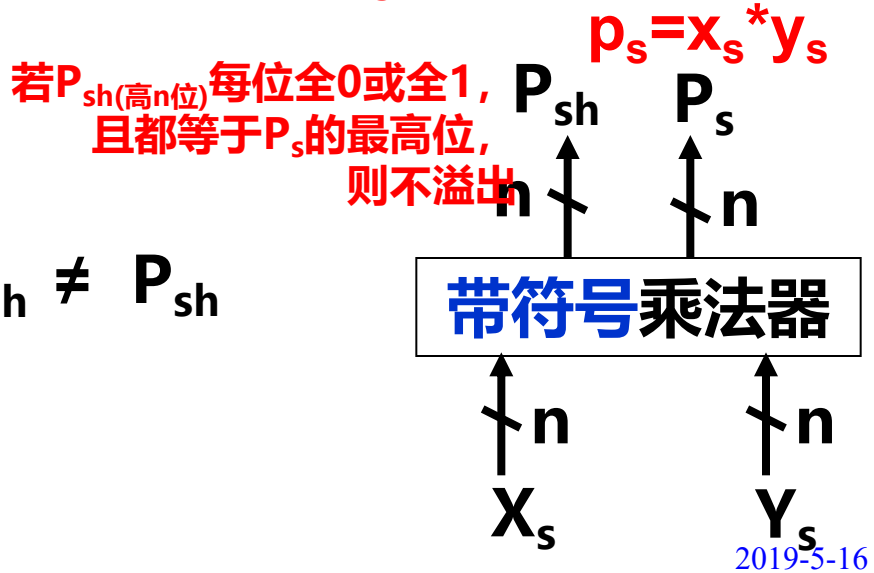
可用无符号乘来实现带符号乘，但高n位无法得到，故不能判断溢出

无符号乘：若$P_{uh(高n位)}=0$，则不溢出

带符号乘：若$P_{sh(高n位)}$每位全0或全1，且都等于P_s的最高位，则不溢出



$P_{uh} \neq P_{sh}$



Kahan累加算法

- The Father of Floating Point

jie@debian: ~/class/ch2/3

```
 1 #include <stdio.h>
 2
 3 void main()
 4 {
 5       float tem = 0.1f;
 6       float sum = tem;
 7       float c = 0;
 8       float y, t;
 9       int i;
10       for( i=1;i<4000000;i++)
11       {
12             y = tem -c;
13             t = sum +y;
14             c = (t-sum)-y;
15             sum = t;
16       }
17       printf("%f\n",
18 }
~
~
<ecision.c[1] [c] uni
```

jie@debian: ~/class/ch2/3

```
h2/3$ ./tt

h2/3$
```

jie@debian: ~/class/ch2/3

```
jie@debian:~/class/ch2/3$ ./precision
400000.000000
jie@debian:~/class/ch2/3$
```

整数的乘运算-举例

- 通常，高级语言中两个n位整数相乘得到的结果通常也是一个n位整数，也即结果只取2n位乘积中的低n位。
  - 例如，在C语言中，参加运算的两个操作数的类型和结果的类型必须一致，如果不一致则会先转换为一致的数据类型再进行计算。

```
int mul(int x, int y)
{
    int z=x*y;
    return z;
}
```

x*y 被转换为乘法指令，在乘法运算电路中得到的乘积是64位，但是，只取其低32位赋给z。

整数的乘运算-举例

在计算机内部，一定有$x^2 \geq 0$吗?

若x是带符号整数，则不一定!

如x是浮点数，则一定!

例如，当 n=4 时, $5^2=-7<0$!

```
0101
×  0101
-------
0101
+ 0101
-------
00011001
```

结果溢出

只取低4位，值为-111B=-7

```
int mul(int x, int y)
{
    int z=x*y;
    return z;
}
```

如何判断返回的z是正确值?

当 !x || z/x==y 为真时

什么情况下，乘积是正确的呢?

当 $-2^{n-1} \leq x*y < 2^{n-1}$ （不溢出）时

即：乘积的高n位为全0或全1，并等于低n位的最高位!

即：乘积的高n+1位为全0或全1

若x、y和z都改成unsigned类型，则判断方式为

乘积的高n位为全0，则不溢出

整数乘法溢出漏洞

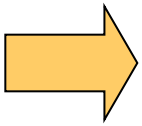
以下程序存在什么漏洞，引起该漏洞的原因是什么。

```
/* 复制数组到堆中，count为数组元素个数 */
int copy_array(int *array, int count) {
      int i;
      /* 在堆区申请一块内存 */
      int *myarray = (int *) malloc(count*sizeof(int));
      if (myarray == NULL)
            return -1;
      for (i = 0; i < count; i++)
            myarray[i] = array[i];
      return count;
}
```

2002年，Sun Microsystems公司的RPC XDR库带的xdr_array函数发生整数溢出漏洞，攻击者可利用该漏洞从远程或本地获取root权限。

攻击者可构造特殊参数来触发整数溢出，以一段预设信息覆盖一个已分配的堆缓冲区，造成远程服务器崩溃或者改变内存数据并执行任意代码。

当参数count很大时，则count*sizeof(int)会溢出。如count=$2^{30}+1$时，count*sizeof(int)=4。



堆(heap)中大量数据被破坏！

变量与常数之间的乘运算

- 整数乘法运算比移位和加法等运算所用时间长，通常一次乘法运算需要多个时钟周期，而一次移位、加法和减法等运算只要一个或更少的时钟周期，因此，**编译器在处理变量与常数相乘时，往往以移位、加法和减法的组合运算来代替乘法运算。**

  例如，对于表达式x*20，编译器可以利用$20=16+4=2^4+2^2$，将x*20转换为(x<<4)+(x<<2)，这样，一次乘法转换成了两次移位和一次加法。

- 不管是无符号数还是带符号整数的乘法，即使乘积溢出时，利用移位和加减运算组合的方式得到的结果都是和采用直接相乘的结果是一样的。

整数的乘运算后只保留低n位结果时溢出判断举例

- X和Y分别是n位的整数，X*Y得到2n位，其高n位可以用来判断溢出，规则：
  - 无符号：若高n位全0，则不溢出，否则溢出
  - 带符号：若高n位全0或全1且等于低n位的最高位，则不溢出

下表中n=4

运算	x	X	y	Y	x×y	X×Y	p	P	溢出否
无符号乘	6	0110	10	1010	60	0011 1100	12	1100	溢出
带符号乘	6	0110	-6	1010	-36	1101 1100	-4	1100	溢出
无符号乘	8	1000	2	0010	16	0001 0000	0	0000	溢出
带符号乘	-8	1000	2	0010	-16	1111 0000	0	0000	溢出
无符号乘	13	1101	14	1110	182	1011 0110	6	0110	溢出
带符号乘	-3	1101	-2	1110	6	0000 0110	6	0110	不溢出
无符号乘	2	0010	12	1100	24	0001 1000	8	1000	溢出
带符号乘	2	0010	-4	1100	-8	1111 1000	-8	1000	不溢出

高级语言中整数的乘运算

无符号：若$P_{uh(高n位)}=0$，则不溢出

带符号：若$P_{sh(高n位)}$每位全0或全1，且都等于P_s的最高位，则不溢出

n位的x,在计算机内x^2的结果保留低n位. 一定有$x^2 \geq 0$吗?

若x是整数,不一定!(可能溢出)

如x是浮点数，则一定!

例如，当 n=4位时, $5^2=-7<0$

```
0101
×  0101
------
0101
+ 0101
------
00011001
```

结果溢出

只取低4位1001，值为-7

```
int mul(int x, int y)
{
    int z=x*y;
    return z;
}
```

若x、y和z都改成unsigned，判断方式为乘积的高n位为全0,则不溢出

高级语言程序如何判断z是正确值？ 当 !x || z/x==y 为真时

编译器如何判断？ 当 $-2^{n-1} \leq x*y < 2^{n-1}$ 时,不溢出

即：乘积的高n位为全0或全1，并等于低n位的最高位,不溢出!

即：乘积的高n+1位为全0或全1,不溢出!

南大考试题

无符号乘(n位*n位)：若$P_{uh(高n位)}$=0，则不溢出

带符号乘(n位*n位)：若$P_{sh(高n位)}$每位全0或全1，且都等于P_s的最高位，则不溢出

在字长为32位的计算机上，某C函数原型声明为：

```
int imul_overflow(int x, int y);
```

该函数用于对两个int型变量x和y的乘积（也是int类型）判断是否溢出，若溢出则返回非0，否则返回0。请完成下列任务或回答下列问题。

（1）两个n位无符号数相乘、两个n位带符号整数相乘的溢出判断规则各是什么？（编译器如何判断溢出？）

（2）已知入口参数x、y所在存储单元的地址为R[ebp]+8、R[ebp]+12，返回值在EAX中，写出实现imul_overflow函数功能的AT&T汇编指令序列，并给出注解。（编译器中判断溢出的代码）

（3）使用64位整型（long long）变量来编写imul_overflow函数的C代码或描述实现思想。

南大考试题

无符号整数相乘：若乘积的高n位为非0，则溢出。

带符号整数相乘：若乘积高n位的每一位都相同，且都等于乘积低n位的符号，则不溢出，否则溢出。

实现该功能的汇编指令序列不唯一。

（例如，可利用imull指令生成的OF标志进行判断）

某实现方案下的汇编指令序列如下：

```
movl  8(%ebp), %eax    /* R[eax]=x
movl  12(%ebp), %edx   /* R[edx]=y
imull  %edx                    /*R[edx]=R[edx]:R[eax]=x*y
sarl   $31, %eax   /*算数右移 R[eax]=R[eax]>>31, 符号扩充31位
xorl  %edx, %eax   /*R[edx]与R[eax]按位异或，若为0，表示不溢
                   /*出。结果存在R[eax],
```

南大试题

将x*y的结果保存在long long型变量中，得到64位乘积，然后把64位乘积强制转换为32位，再符号扩展成64位，和原来真正的64位乘积相比，若不相等则溢出。

```
int imul_overflow(int x, int y)
{
        long long prod_64= (long long) x*y;
        return prod_64 != (int) prod_64;
}
```

关于乘运算的几个问题

- 假定CPU中没有乘法器，只有ALU、移位器以及与、或、非等逻辑电路，则其指令系统能提供乘法指令吗?
- 假定ISA中不包含乘法指令，但包含加、减、移位以及与、或、非三种逻辑运算指令，则基于该ISA的系统能执行以下程序吗?

```
int imul(int x, int y)  {
        return  x*y;
}
```

- 以下两个函数的机器级代码相同吗？ 返回的结果一定相同吗？什么情况下相同？返回的结果一定正确吗?

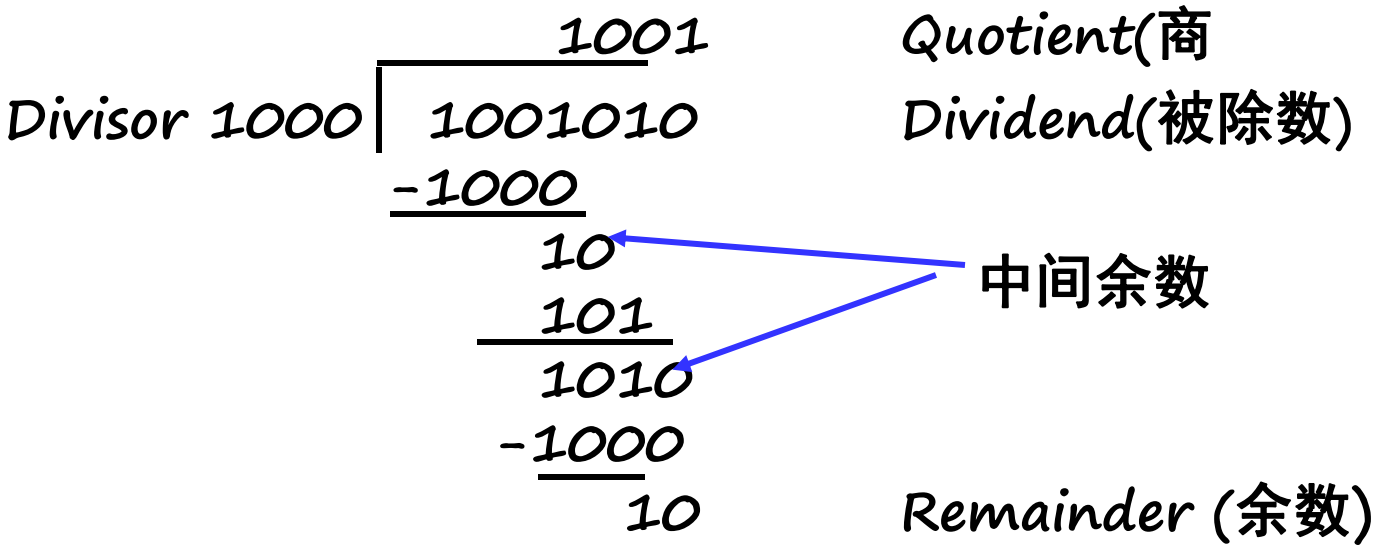
```
int imul(int x, int y)  {
        return  x*y;
}
unsigned imul(unsigned x, unsigned y)  {
        return  x*y;
}
```

①：用循环程序实现
②：直接用较慢指令实现
③：直接用较快指令实现

- 上述程序执行时间比较

①无乘法指令 > ②有用ALU实现乘法指令 > ③具有用乘法器实现乘法指令

除法（Divide）：Paper & Pencil(手算)



- 手算除法的基本要点

① 被除数与除数相减，够减则上商为1；不够减则上商为0。

② 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，够减则上商为1；不够减则上商为0。

③ 重复执行第②步，直到求得的商的位数足够为止。

定点数除法运算

- **除前预处理**

  ①若被除数=0且除数≠0，或定点整数除法时|被除数|<|除数|，则商为0，不再继续

  ②若被除数≠0、除数=0，则发生“除数为0”异常（浮点数时为∞）

  （若浮点除法被除数和除数都为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≠ 0，且商≠ 0时，才进一步进行除法运算。

- **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**

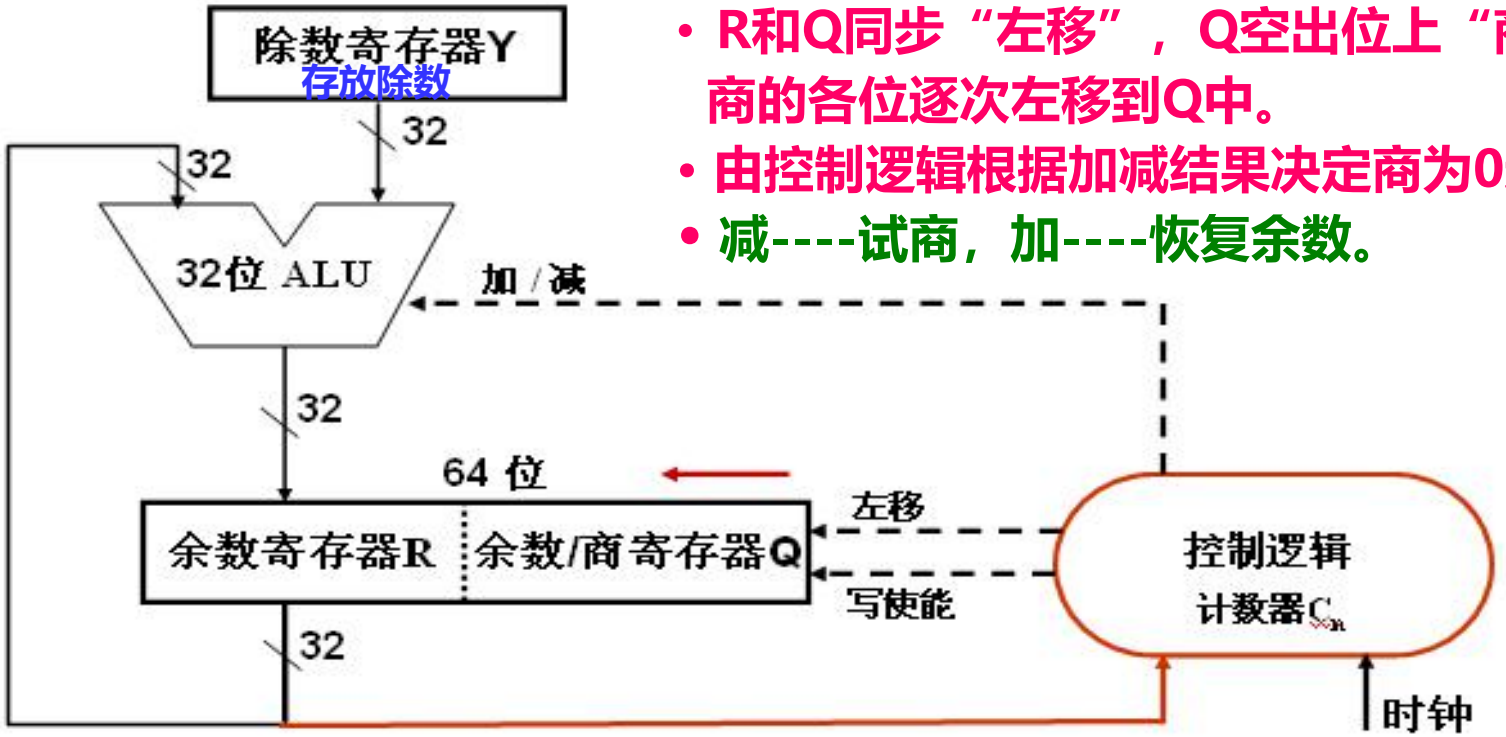
    够减,则上商1；不够减,则上商0（从msb→lsb得到各位商）
  - **基本操作为减（加）法和移位，故可与乘法合用同一套硬件**

两个n位数相除的情况-都统一为一个2n位数除以一个n位数:

(1)定点正整数（即无符号数 ）相除：在被除数的高位添n个0

(2)定点正小数（即原码小数）相除：在被除数的低位添加n个0

除法的硬件实现



- R和Q同步“左移”，Q空出位上“商”，商的各位逐次左移到Q中。
- 由控制逻辑根据加减结果决定商为0还是1
- 减----试商，加----恢复余数。

- 除数寄存器Y：存放除数。
- 余数寄存器R：初始时高位部分为高32位被除数；结束时是余数。
- 余数/商寄存器Q：初始时为低32位被除数；结束时是32位商。
- 存放循环次数的计数器Cn：初值32，每循环一次，减1；Cn=0时，除法结束
- ALU：除法核心部件。在控制逻辑控制下，对于寄存器R和Y的内容进行“加/减”运算，在“写使能”控制下运算结果被送回寄存器R。

第一次试商为1时的情况

两个n位数相除的情况-都统一为一个2n位数除以一个n位数

(1)定点正整数（即无符号数）相除：在被除数的高位添n个0

(2)定点正小数（即原码小数）相除：在被除数的低位添加n个0

两个n位数相除(转化为2n位除以n位),第一次试商为1，说明什么？商的第(n+1)位为1,商有n+1位数，因而溢出！例：1111 1111/1111 = 1 0001

若是2n位除以n位的无符号整数运算，则说明将会得到多于n位的商，因而结果“溢出”（即：无法用n位表示商）。

若是两个n位数相除(转化为2n位除以n位)，则第一位商为0，且肯定不会溢出,最大商为：（0000 1111）/ 0001=1111

若是浮点数中尾数原码小数运算，第一次试商为1，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。

例：0.11110000/0.1000=+1.1110

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

- 以定点小数原码除 法为例,基本公式：
- 设被除数 $[X]_{原} = x_f . x_1 x_2 \cdots x_n$

  除　数　$[Y]_{原} = y_f . y_1 y_2 \cdots y_n$

  若　$0 < X < Y$

  $[X \div Y]_{原} = (x_f \oplus y_f) . (X^* \div Y^*)$

  其中，　X^* 和Y^*分别是X和Y的绝对值

无符号整数除法举例-恢复余数法

验证：7 / 2 = 3 余 1

除数D=2=0010

–D = 1110

-D = 1110

		余数寄存器R	余数/商寄存器Q
	D: 0010	R: 0000	0111
Shl R	D: 0010	R: 0000	111□
R=R–D=R+(-D)	D: 0010	R: 1110	1110
+D,sl R,0(恢复余数)	D: 0010	R: 0001	110□
R=R–D=R+(-D)	D: 0010	R: 1111	1100
+D,sl R,0(恢复余数)	D: 0010	R: 0011	100□
R=R–D=R+(-D)	D: 0010	R: 0001	1001
sl R, 1	D: 0010	R: 0011	001□
R=R–D=R+(-D)	D: 0010	R: 0001	0011

7 / 2 的余数为1， 商为0011=3

两个n位无符号数相除，不会溢出,不必判断，故余数先左移

初始时中间余数为4个0,低4位为被除数;结束时低4位是4位商

开始余数先左移一位，然后减

每次上商为0后,即做加法来”恢复余数”。所以，称为“恢复余数法

	1110	1110 //+D,sl R,0三步曲
+	0010	<–加除数来恢复余数
	10000	1110
10	0001	110□sl R,左移一位后

不恢复余数除法(也称为加减交替法)

恢复余数法可进一步简化为**不恢复余数法，也叫“加减交替法”，也可在下一步运算时把当前多减的除数补回来** P89

不恢复余数法方法为 (设Y为除数，R_i为第i次中间余数)：

- 若余数$R_i>=0$,则商上“1”，左移1位，再减去除数， 得到新的余数$R_{i+1}=2R_i-Y$，然后再根据余数的新值给商赋新值。 即(“(余数为）正，(商为） 1，减(除数)”)
- 若$R_i<0$,则商上“0”，左移一位，加(除数)（实际是恢复余数），得到新的余数$R_{i+1}=2(R_i+Y)-Y=2R_i+Y$，然后再根据余数的新值给商赋新值。 (“(余数为）负，(商为）O，加(除数)”)

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。

原码小数不恢复余数法

已知:

$[X]_{原}$=0.1011

$[Y]_{原}$=0.1101

求 $[X/Y]_{原}$

$[Y]_{补}$ = 0.1101

$[-Y]_{补}$= 1. 0011

X / Y = 0.1101

原码除法执行的是绝对值相除，-Y用$+[-Y]_{补}$完成，+Y 用加Y的绝对值 (补码) 完成

余数寄存器R(初始放被除数)	余数 | 商	
0 1 0 1 1	0 0 0 0 □	开始情形
+) 1 0 0 1 1		-Y
1 1 1 1 0	0 0 0 0 0	<0, 商0
1 1 1 0 0	0 0 0 0 □	左移1位
+) 0 1 1 0 1		+Y
0 1 0 0 1	0 0 0 0 1	>0, 商1
1 0 0 10	0 0 0 1 □	左移1位
+) 1 0 0 1 1		-Y
0 0 1 0 1	0 0 0 1 1	>0, 商1
0 1 0 1 0	0 0 1 1 □	左移1位
+) 1 0 0 1 1		-Y
1 1 1 0 1	0 0 1 1 0	<0, 商0
1 1 0 1 0	0 1 1 0 □	左移1位
+) 0 1 1 01		+Y
0 0 1 1 1	0 1 1 0 1	>0, 商1

$[X / Y]_{原}$ 的余数为$0.0111*2^{-4}$

商的符号为正

商的最高位0说明没有溢出, $[X / Y]_{原}$ = 0.1101

原码小数除法举例

已知 $[X]_{原}$ = 0.1011

$[Y]_{原}$ = 1.1101

用恢复余数法计算$[X/Y]_{原}$

解：商的符号位：0 ⊕ 1 = 1

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。

$[|X|]_{补}$ = 0.1011

$[|Y|]_{补}$ = 0.1101

$[-|Y|]_{补}$ = 1.0011

思考：若实现无符号(整数)数相除，即1011除以1101，不同处？结果是什么？

被除数高位补0，1011除以1101，结果等于0

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		$R_3=2R_2-Y$
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，$[X/Y]_{原}$=1.1101 (最高位为符号位)，余数为 0.0111×2^{-4}。

小数在低位扩展0

判断溢出

若求$[Y/X]_{原}$结果如何？

原码除法举例

已知 $[X]_{原} = 0.1011$

$[Y]_{原} = 1.1101$

用加减交替法计算$[X/Y]_{原}$

解：$[|X|]_{补} = 0.1011$

$[|Y|]_{补} = 0.1101$

$[-|Y|]_{补} = 1.0011$

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		$R_3=2R_2-Y$
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 $q_0=1$

“加减交替法”的要点：

负、0、加

正、1、减

为

得到的结果与恢复余数法一样！

用被除数（中间余数）减除数试商时，怎样确定是否“够减”？

中间余数的符号！（正数-正数）

补码除法能否这样来判断呢？不能，因为符号可能不同！

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。

- 若是两个n位补码整数除法运算，则被除数进行符号扩展。
- 若被除数为2n位，除数为n位，则被除数无需扩展。

补码除法（自学，不作要求）

- 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。

上述判断规则归纳如下：

中间余数R的符号	除数Y的符号	同号：新中间余数=R–Y（同号为正商）		异号：新中间余数=R+Y（异号为负商）	
		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		

总结：余数变号不够减，不变号够减

实现补码除法的基本思想（自学，不作要求）

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。

（2）上商规则：

若余数符号不变，则够减，商1；否则不够减，商0 。

（3）修正规则：

若被除数与除数符号一致，则商为正。 此时，“够减，商1；不够减，商0，故上商规则正确，无需修正”

若被除数与除数符号不一致，则商为负。此时，“够减，商0；不够减，商1，故上商规则相反，需修正”

即：若商为负值，则需要“各位取反，末位加1”来得到真正的商

补码除法也有：恢复余数法和不恢复余数法

SKIP

补码恢复余数法（自学，不作要求）

- **两个n位带符号整数相除算法要点：**

(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 = 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!

补码：若原来为R = R–Y，则执行R = R+Y来恢复余数；

若原来是R = R+Y，则执行R = R–Y来恢复余数。

举例：7/3=?　(-7)/3=?（自学）

被除数=0000 0111　除数=0011

	A(余数)	Q(商)	M(除数)=0011	
	0000	0111		
←	0000	1110		
+	1101		减（同号）	
	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	1000	符同商1	
←	0001	0001		
+	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（异号）	
	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（自学）

- **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，不是新老余数之间！而是余数和除数之间

(3) 对于i =1到n，按以下规则求出n位商：

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

② 若Ri与Y异号，则q_{n-i}置0，Ri+1 =2Ri+[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”。

补码除法举例 -9/2

(自学)

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×商+余数= 被除数”进行验证，得

2×(–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	11011	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。

编译器遇到x/2时会如何做？ 右移一位吗！

整数的除运算-举例

代码段一：

```
int a = 0x80000000;
int b = a / -1;
printf("%d\n", b);
```

运行结果为-2147483648

用ObjDump工具在Linux下看代码段一的反汇编代码，得知除以 -1 被优化成取负指令neg(用零减去操作数,再存入该操作数)，故未发生除法溢出

代码段二：

```
int a = 0x80000000;
int b = -1;
int c = a / b;
printf("%d\n", c);
```

运行结果为“Floating point exception”，显然CPU检测到了异常

为什么显示是“浮点异常”呢？学完第6章应该能回答这个问题！

为什么两者结果不同！

整数变量与常数之间的除运算(1)

- 由于计算机中除法运算比较复杂，所以一次除法运算大致需要30个或更多个时钟周期，比乘法指令的时间还要长！
- 为了缩短除法运算的时间，编译器在处理一个变量与一个2的幂次形式的整数相除时，常采用右移运算来实现。
  - 无符号：逻辑右移
  - 带符号：算术右移
- 结果一定取整数
  - 能整除时，直接右移得到结果，移出的为全0

    例如，12/4=3：0000 1100>>2=0000 0011

    -12/4=-3：1111 0100 >>2=1111 1101
  - 不能整除时，右移移出的位中有非0，需要进行相应处理

整数变量与常数之间的除运算 (2)

- 不能整除时，采用朝零舍入，即截断方式
  - 无符号数、带符号正整数（地板）：移出的低位直接丢弃
  - 带符号负整数（天板）：加偏移量(2^k-1)，然后再右移k位，低位截断（这里K是右移位数）

举例：

无符号数 14/4=3：0000 1110>>2=0000 0011

带符号负整数 -14/4=-3

若直接截断，则 1111 0010 >>2=1111 1100=-4≠-3

应先纠偏，再右移: k=2, 故$(-14+2^2-1)/4=-3$

即： 1111 0010+0000 0011=1111 0101

1111 0101>>2=1111 1101=-3

-9/2: 10111+00001=11000

11000>>1=11100=-4

整数变量与常数之间的除运算-举例

- 假设*x*为一个int型变量，请给出一个用来计算*x*/32的值的函数div32。要求不能使用除法、乘法、模运算、比较运算、循环语句和条件语句，可以使用右移、加法以及任何按位运算。

解：若x为正数，则将x右移k位得到商；若x为负数，则x需要加一个偏移量(2^k-1)后再右移k位得到商。因为$32=2^5$，所以 $k=5$。$(2^k-1)= (2^5-1)=31$

即结果为: (x>=0 ? x : (x+31))>>5

但题目要求不能用比较和条件语句，因此要找一个计算偏移量b的方式

这里，x为正时b=0，x为负时b=31. 因此，可以从x的符号得到b

x>>31 得到的是32位符号，取出最低5位，就是偏移量b。

```
int div32(int x)
{  /* 根据x的符号得到偏移量b */
    int b=(x>>31) & 0x1F;
    return (x+b)>>5;
}
```

IA-32中的整数乘除指令

- 乘法指令：可给出一个、两个或三个操作数（生成OF和CF，不转异常处理）
  - 若给出一个操作数SRC，则另一个源操作数隐含在AL/AX/EAX中，将SRC和累加器内容相乘，结果存放在AX（16位）或DX-AX（32位）或EDX-EAX（64位）中。DX-AX表示32位乘积的高、低16位分别在DX和AX中。 n位× n位=2n位
  - 若指令中给出两个操作数DST和SRC，则将DST和SRC相乘，结果在DST中。n位× n位=n位
  - 若指令中给出三个操作数REG、SRC和IMM，则将SRC和立即数IMM相乘，结果在REG中。n位× n位=n位
- 除法指令：只明显指出除数（判定是否异常并在异常时转异常处理）
  - 若为8位，则16位被除数在AX寄存器中，商送回AL，余数在AH
  - 若为16位，则32位被除数在DX-AX寄存器中，商送回AX，余数在DX
  - 若为32位，则被除数在EDX-EAX寄存器中，商送EAX，余数在EDX

乘法和除法都区分带符号和无符号: MUL, IMUL
DIV，IDIV

MIPS中整数的乘、除运算处理

- 指令: mult ， multu；div ， divu
- MIPS中有一对32位寄存器Hi寄存器& Lo寄存器（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
  - 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指令不处理异常，由系统软件自行处理

问题：如何判断溢出?

第一讲小结

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)；原码小数；移码整数

- **定点运算：(ALU实现基本算术和逻辑运算，ALU+移位器实现其他运算)**

  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

第一讲小结

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，或用乘法实现

- **定点部件：ALU、GRS、MUX、Shifter、Q寄存器等，CU控制执行**

第二讲：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为例）
  - 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

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 $A=M_a \cdot 2^{Ea}$　$B=M_b \cdot 2^{Eb}$,则:

$A \pm B = (M_a \pm (M_b \cdot 2^{-(Ea-Eb)})) \cdot 2^{Ea}$　(假设Ea>=Eb, 对阶)

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

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

1.5+1.5=?

1.5-1.0=?

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 =〉+∞/-∞/溢出

SP中最大指数为127(不能128)　--P40

阶码下溢: 一个负指数比最小允许值还小 =〉+0/-0

非规格化SP最小指数为126　--P41

尾数溢出: 最高有效位有进位 =〉右规。尾数溢出，结果不一定溢出

非规格化尾数: 数值部分高位为0 =〉左规

运算过程中添加保护位

右规或对阶时，右段有效位丢失 =〉尾数舍入

IEEE建议为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 x ∞、 ∞/∞等
- 结果无效，如：源操作数是NaN、0/0、x/0、 ∞ /y 等

② 除以0（即：无穷大）

③ 数太大（阶码上溢）：对于SP，阶码 E >1111 1110 （阶大于127）

④ 数太小（阶码下溢）：对于SP，阶码 E < 0000 0001（阶小于-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

上述异常情况硬件可以捕捉到，可设定让硬件处理这些异常(这时候,称为硬件陷阱)，也可设定让软件处理。

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）。

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?

无法给出准确的答案!

Add/Sub:

	1.xxxxx	1.xxxxx	1.xxxx x	1.xxxxxxxx
+	1.xxxxx	0.001xxxxx	0.01xx xxx	-1.xxxxxxxx
	1x.xxxxy	1.xxxxxyyy	1x.xxxxyyy	0.0…0xxxx

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

Guard (保护位): 在significand右边的位

Round (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：若十进制数最终有效位数为 3，采用两位附加位（G、R）。

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！

$$
\begin{array}{r}
2.3400 \times 10^2 \\
+0.0253 \times 10^2 \\
\hline
2.3653 \times 10^2
\end{array}
$$

IEEE Standard: four rounding modes

1) 朝+∞方向舍入,always round towards plus infinity (round up)

2) 朝-∞方向舍入, always round towards minus infinity(round down)

3) 朝0方向舍入, round towards 0

4) (缺省)就近舍入：舍入为最近可表示的数，也称为就近舍入到偶数

round to nearest :

round digit < 1/2 then truncate (截断、丢弃)

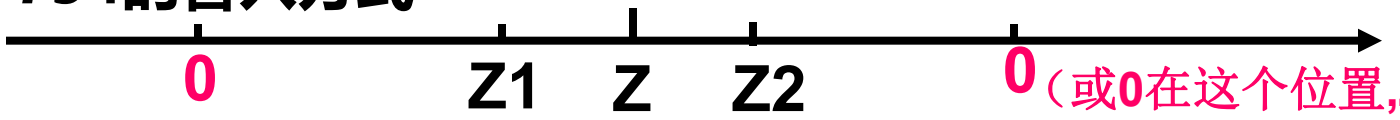
> 1/2 then round up (末位加1)

= 1/2 then round to nearest even digit(最近偶数)

可以证明默认方式得到的平均误差最小。

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的最近的可表示的左、右两个数）

(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

例如：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+∞方向舍入:舍入为Z2(正向舍入)

(3) 朝-∞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

BACK

浮点数加/减运算举例

- 十进制科学计数法的加法例子，加法计算后,小数点后保留三位

  $1.123 \times 10^5 + 2.560 \times 10^2$

  其计算过程为：

  $1.123 \times 10^5 + 2.560 \times 10^2 = 1.123 \times 10^5 + 0.002560 \times 10^5$

  $=(1.123 + 0.00256) \times 10^5 = 1.125656 \times 10^5$

  $=1.126 \times 10^5$

  “附加位”

- 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

浮点数加法运算举例

例：用二进制浮点数形式计算 0.5 +(– 0.4375) =?

解：$0.5 = 1.000 \times 2^{-1}$,

$0.4375 = 0.25+0.125+0.0625 = 0.0111B$

$-0.4375 = -1.110 \times 2^{-2}$

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

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

左　规：$0.001 \times 2^{-1} \rightarrow 1.000 \times 2^{-4}$

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?

答：因为即使是两个最大的尾数1.111+ 1.111相加，每个都小于2，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。

回顾：浮点数除0的问题

为什么整数除0会发生异常?

为什么浮点数除0不会出现异常?

浮点运算中，一个有限数除以0，结果为正(或负)无穷大

```
#include <conio.h>
#include <stdio.h>
int    main()
{
        int    a=1,b=0;
        printf( "Division    by    zero:%d\n ",a/b);
        getchar();
        return    0;
}
int    main()
{
        double    x=1.0,y=-1.0,z=0.0;
        printf( "division    by    zero:%f    %f\n ",x/z,y/z);
        getchar();
        return    0;
}
```

这是网上的一个帖子

问题一:为什么整除int型会产生错误? 是什么错误?

二:用double型的时候结果为1.#INF00和-1.#INF00，作何解释???

浮点数加减法基本要点

假定：Xm、Ym分别是X和Y的尾数， Xe和Ye 分别是X和Y的阶码

(1) 求阶差：Δe=Ye – Xe（若Ye > Xe，则结果的阶码为Ye)

(2) 对阶：将Xm右移Δe位，尾数变为 $Xm*2^{Xe-Ye}$（保留右移部分附加位)

(3) 尾数加减： $Xm*2^{Xe-Ye} \pm Ym$

(4) 规格化：

IEEE 754：$0.00\ldots0001 \times 2^{-126}$

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或阶码为00000000（-126，$0.f*2^{-126}$ 是非规格化数，见P41 表2.3续)

每次阶码减1后要判断阶码是否下溢（规格化时0<阶码e<255，非规格化可以表示数的 阶码e = 0。比最小可表示的阶码还要小时下溢)

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；阶码下溢到无法用非规格化数表示，（一般）置则结果为0，或有的机器进行异常处理。

如果尾数比规定位数长（有附加位），则需考虑舍入（有多种舍入方式)

若运算结果尾数是0，一般将阶码也置0。为什么?

答：尾数为0，结果一般应该为0（阶码和尾数为全0)

浮点数加/减运算–对阶

问题：如何对阶？通过计算$[\Delta E]_{补}$来判断两数的阶差：

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

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

4位移码,Ex=7, Ey=-7, Bias = $2^{(4-1)} = 2^{(3)} = 8$， $[Ex]_{移}=7+8=15=1111$

$[Ey]_{移}$ = （-7）+8=1，$[-[Ey]_{移}]_{补} = [-(1)]_{补}$=1111(4位表示)

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

$=1111+1111 \pmod{2^n} =1110$，ΔE<0,错

对IEEE754 SP格式来说，|ΔE|大于24时，结果就等于阶大的那个数（即小数被大数吃掉）,即 1.xx…x → 0.00…01xx…x(右移24位后，尾数变为0)

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$[Ex-Ey]_{补} \pmod{2^n}$, (n=8, 即 $\bmod 2^8 = 256$) 没有影响

$[\Delta E]_{补}=[Ex]_{移}+[-[Ey]_{移}]_{补} + 2^8 = [Ex]_{移} + (256-[Ey]_{移}) \pmod{256}$

$= [Ex]_{移}+[-[Ey]_{移}]_{补} \pmod{256}$

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

原码加/减运算

- 用于浮点数尾数运算 P77
- 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，加法不一定做加法，减法不一定做减加法！
- 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”。
  - （不论“加法”还是”减法”中的）求和时：数值位相加。若最高位产生进位，则结果溢出；无溢出，则 和的符号取被加数（被减数）的符号。
  - （不论“加法”还是”减法”中的）求差时：被加数（被减数）的数值部分加上加数（减数）的数值部分的补码。

    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/减运算举例

例1：已知 $[X]_{原}$ = 1.0011，$[Y]_{原}$ = 1.1010，要求计算$[X+Y]_{原}$ (原码加/减,书P77)

解：由原码加运算规则知：同号，则数值位加，有进位则溢出; 无溢出,则和的符号位取被加数符号。

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

和的符号位取被加数符号为：1， $[X+Y]_{原}$ = 1.1101

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

解：由原码减运算规则知：同号,则数值位相减(被减数的数值位加上减数位的数值位的补码)(求差,补码减法）.最高数值位有进位，结果为正；没有进位，结果为负，要对结果求补码，还原成绝对值形式的数值。

数值位差为:$[0011]_{原}-[1010]_{原}=0011+(-|[1010]_{原}|)_{补}=0011+0110=1001$

最高数值位没有进位,左边第一位符号位1表示这是真正结果的负数,求$(-1001)_{补}$才能还原为真正的数值表示, $(-1001)_{补}=(-|1001|)_{补}=0111$ (书P32-35)

差的符号位为被减数$[X]_{原}$的符号位取反，即：0，所以 $[X-Y]_{原}$ = 0.0111

思考题：如何设计一个基于加法器的原码加/减法器?

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) 如何表示浮点数? 用IEEE754标准！

(2) 判断阶码的大小方法：求$[\Delta E]_{补}$=?

(3) 对阶后尾数的隐含位处理：右移到数值部分，高位补0，保留移出低位部分

(4) 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

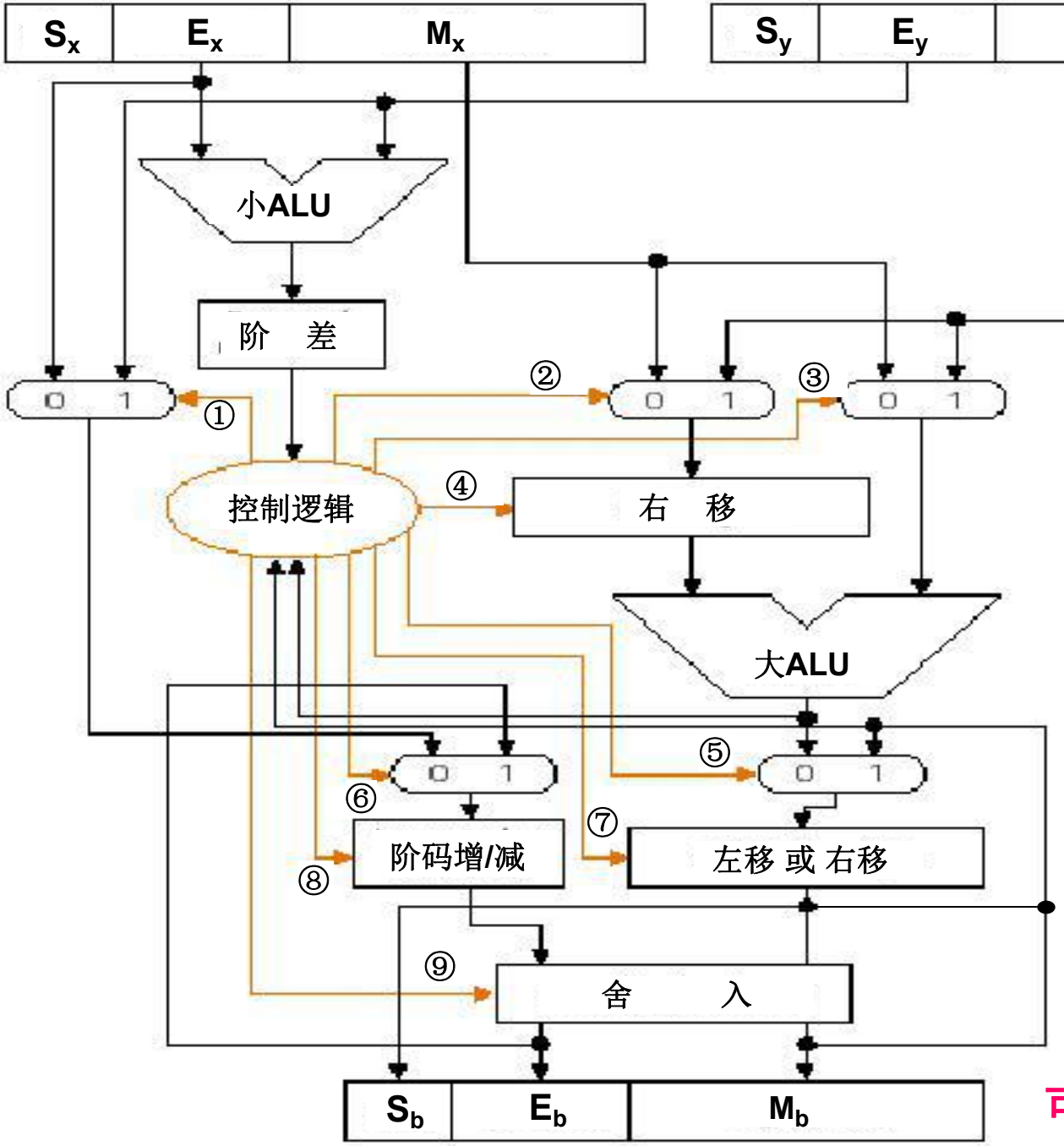
(5) 何时需要规格化，如何规格化?

±1x .xx……x 形式时，则右规: 尾数右移1位, 阶码加1

± 0.0…01x…x 形式时，则左规: 尾数左移k位, 阶码减k

(6) 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) 判断溢出方法：若最终阶码为全1，则上溢；若尾数为全0，则下溢



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！

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: $x=0.5=1/2=(0.100...0)_2=(1.00...0)_2x2^{-1}$

$y=-0.4325=(-0.01110...0)_2=(-1.110..0)_2x2^{-2}$

IEEE754$[x]_{浮}$=0(01111110)00...0 ; $[y]_{浮}$=1(01111101)110...0, (阶码在括号中)

对阶: $[\Delta E]_{补}=01111110+(-(01111101))_{补}=01111110+1000\ 0011$

$=0000\ 0001$, $\Delta E=1$

对y对阶: $[y]_{浮}$=1 (0111 1110) 1110...0(尾数的最高位加上隐藏位1)

用原码加法计算01.0000...0(x)+(10.1110...0)(y) x,y的最左边0,1都为符号位

尾数(原码)相加(x,y异符号,求减 P77): 1.0000...0-(0.1110...0) = 0.0010...0

左规: $+(0.00100...0)_2x2^{-1}=+(1.00...0)_2x2^{-4}$

(阶码减3==加了三次11111111)

$[x+y]_{浮\text{-}IEEE754}$= 0_0111_1011_00...0 (移码=127+(-4))

$x+y=(1.0)_2x2^{-4}=1/16=0.0625$

尾数加法器最多需要多少位? 1(符号位)+1(隐含位)+23+3(左移3次)=28位

浮点数乘/除法基本要点

- 浮点数乘法：$A*B=(M_a * M_b)\cdot 2^{Ea+Eb}$
- 浮点数除法：$A/B=(M_a / M_b)\cdot 2^{Ea-Eb}$

浮点数尾数采用原码乘/除运算

浮点数乘 / 除法步骤 (P35,指数的移码叫阶码(2进制表示), 阶指的是其真值)

设Xm、Ym分别是X和Y尾数原码， Xe和Ye 分别是X和Y阶的移码

(1) 求阶： Xe ± Ye ∓ 127

(2) 尾数相乘除： Xm */Ym （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，则需要将阶码也置0。

(7) 阶码溢出判断

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！

除法呢？左规次数不定！不需右规！

整数的除运算

- 对于带符号整数来说，n位整数除以n位整数，除$-2^{n-1}/-1=2^{n-1}$会发生溢出外，其余情况都不会发生溢出。Why?

  因为商的绝对值不可能比被除数的绝对值更大，因而不会发生溢出，也就不会像整数乘法运算那样发生整数溢出漏洞。

- 因为整数除法，其商也是整数，所以，在不能整除时需要进行舍入，通常按照朝0方向舍入，即正数商取比自身小的最接近整数（Floor，地板），负数商取比自身大的最接近整数（Ceiling，天板）。

  例如，7/2=?，-7/2=?

  7/2=3，-7/2=-3

- 整除0的结果可以用什么机器数表示?

  整除0的结果无法用一个机器数表示!

- 整数除法时，除数不能为0，否则会发生“异常”，此时，需要调出操作系统中的异常处理程序来处理。

求阶码的和、差

无符号数 10000001B=129；　有符号数补码10000001B=-128+1=-127

设E1和E2分别是两个指数，采用移码加减法计算指数阶码加减的结果：

- 阶码加法公式为：　$Eb \leftarrow Ex+Ey+129 \quad (\bmod 2^8)$

$$[(E1+E2)]_{移} = 127+(E1+E2)=(127+E1)+(127+E2)-127$$
$$= [E1]_{移} + [E2]_{移} - 127$$
$$= [E1]_{移} + [E2]_{移} + [-127]_{补} \quad //(-127=10000001)$$
$$= [E1]_{移} + [E2]_{移} + 10000001B(其无符号数为129)$$
$$= ([E1]_{移} + [E2]_{移} + 129) \quad (\bmod 2^8)$$

- 阶码减法公式为：　$Eb \leftarrow Ex+[-Ey]补+127 \quad (\bmod 2^8)$

$$[(E1-E2)]_{移} = (E1-E2)+127 =(127+E1)-(127+E2)+127$$
$$= [E1]_{移}-[E2]_{移} + 127$$
$$= [E1]_{移}+[-[E2]_{移}]_{补}+01111111B$$
$$= [E1]_{移}+[-[E2]_{移}]_{补}+127 \quad (\bmod 2^8)$$

移码计算举例

设E1和E2分别是两个指数，采用移码加减法计算指数阶码加减的结果：

- 阶码加法公式为： $Eb \leftarrow Ex+Ey+129 \ (\mathrm{mod}\ 2^8)$
- 阶码减法公式为： $Eb \leftarrow Ex+[-Ey]_{补}+127 \ (\mathrm{mod}\ 2^8)$

例：若两个阶码分别为10和-5，求10+(-5)和10-(-5)的移码。

解：$Ex = [10]_{移} = 127+10 = 137 = 1000\ 1001B$

$Ey = [-5]_{移} = 127+(-5) = 122 = 0111\ 1010B$

$[-Ey]_{补} = 1000\ 0110B$

① 求10+(-5)和10-(-5)的移码，将Ex和Ey代入上述公式，得：

$Eb = Ex+Ey+129 = 1000\ 1001 + 0111\ 1010 + 1000\ 0001$

$= 1000\ 0100B = 132 \ (\mathrm{mod}\ 2^8)$

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

② 求10-(-5)的移码 $Eb = Ex+[-Ey]_{补}+127$

$= 1000\ 1001 + 1000\ 0110 + 0111\ 1111$

$= 1000\ 1110B = 142 \ (\mathrm{mod}\ 2^8)$

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

浮点数运算中溢出判断

以下情况下，可能会导致阶码溢出

- **左规时,阶码需要减去 1**
  - **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。**
- **右规时,阶码需要加上1**
  - **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？ $+[-1]_{补} = +\ 11\ldots1$

阶码溢出举例

设Ex和Ey分别是两个操作数阶码的移码，Eb是结果阶码的移码表示

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Ex和Ey最高位皆1，而Eb最高位是0或Eb为全1，则阶码上溢**
  - **若Ex和Ey最高位皆0，而Eb最高位是1或Eb为全0，则阶码下溢**
- **除法运算求阶码的差时**
  - **若Ex的最高位是1，Ey的最高位是0，Eb的最高位是0或Eb为全1，则阶码上溢。**
  - **若Ex的最高位是0，Ey的最高位是1，Eb的最高位是1或Eb为全0，则阶码下溢。**

例：若Eb = 0000 0001，则左规一次后价码减去1，结果的阶码 Eb = ？

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

例：若Ex=1111 1110，Ey=1000 0000，则乘法运算时结果的阶码 Eb=?

解：Eb=Ex+Ey+129=1111 1110+1000 0000+1000 0001=1111 1111

阶码上溢!

BACK

C语言中的浮点数类型

- C语言中有float和double类型，分别对应IEEE 754单精度浮点数格式和双精度浮点数格式
- long double类型的长度和格式随编译器和处理器类型的不同而有所不同，IA-32中是80位扩展精度格式
- 从int转换为float时，不会发生溢出，但可能有数据被舍入
- 从int或 float转换为double时，因为double的有效位数更多，故能保留精确值
- 从double转换为float和int时，可能发生溢出，此外，由于有效位数变少，故可能被舍入
- 从float 或double转换为int时，因为int没有小数部分，所以数据可能会向0方向被截断

浮点数舍入举例

例：将同一实数分别赋值给单精度和双精度类型变量，然后打印输出。

```
#include <stdio.h>
main()
{
        float a;
        double b;
        a = 123456.789e4;
        b = 123456.789e4;
        printf( “%f/n%f/n” ,a,b);
}
```

运行结果如下：

```
        1234567936.000000
        1234567890.000000
```

同一个实数赋值给float型变量和double型变量，输出结果会有所不同

float情况下输出的结果会比原来的大，因为float可精确表示7个十进制有效数位，后面的数位是舍入后的结果，可能会更大，也可能更小

回顾：非规格化浮点数举例

当结果为 0.1×2^{-126} 时，是用非规格化数表示还是近似为0?

以下程序试图计算 $2^{-63}/2^{64}=2^{-127}$

```
#include <stdio.h>

main()
{  float x=1.08420217248550 4e-19;
   float y=1.84467440737095 5e+19;
   float z=x/y;
   printf("x=%f %x\n",x,x);
   printf("y=%f %x\n",y,y);
   printf("z=%f %x\n",z,z);
}
```

```
user@debian:~/Templates$ ./denom
x=0.000000 0
y=18446744073709551616.000000 0
z=0.000000 0
```

计算器

2^{-63}

2^{64}

讨论问题：

1. **计算器上算的准确吗?**
2. **为什么 x 输出为 0?**
3. **为什么 y 的输出发生变化?**
4. **为什么x、y、z用%x输出为0?**
5. **Z 输出为 0 说明了什么?**
6. **如下赋初值对否?**

   float x=0x40000000;

   float y=0x5f800000;

回顾：非规格化浮点数举例

```
#include <stdio.h>

main()
{  float x=0.000015258789062 5;
   float y=0.000030517578125;
   float z=x*x*x*y;
   float l=1.84467440737095516 16e+19;
   float m=z/l;
   printf("z=%e \n",z);
   printf("m=%1.38e \n",m);
   //printf("z=%f %x\n",z,z);
}
```

2^{-16}

2^{-15}

当结果为 $0.1x2^{-126}$ 时，用非规格化数表示，而不是近似表示成0！

z和m的输出结果说明了什么?

```
user@debian:~/Templates$ gcc -o denom denom.c
user@debian:~/Templates$ ./denom
z=1.084202e-19
m=5.87747175411143753984368268611122838909e-39
```

回顾：浮点数比较运算举例

- 对于以下给定的关系表达式，判断是否永真。

```
int x ;
float f ;
double d ;
```

Assume neither d nor f is NaN

自己写程序测试一下!

x == (int)(float) x　　否

x == (int)(double) x　　是

f == (float)(double) f　　是

d == (float) d　　否

f == -(-f);　　是

2/3 == 2/3.0　　否

d < 0.0　⇒　((d*2) < 0.0)　　是

d > f　⇒　-f > -d　　是

d * d >= 0.0　　是

x*x>=0　否

(d+f)-d == f　否

回顾：IEEE 754 的范围和精度

- 单精度浮点数（float型）的表示范围多大?

  最大的数据: $+1.11\ldots1 \times 2^{127}$

  约 $+3.4 \times 10^{38}$

  双精度浮点数（double型）呢?

  约 $+1.8 \times 10^{308}$

- 以下关系表达式是否永真?

```
if ( i == (int) ((float) i) ) {
      printf ( "true" );
}
```

Not always true!

How about double? True!

```
if ( f == (float) ((int) f) ) {
      printf ( "true" );
}
```

Not always true!

How about double? False!

- 浮点数加法结合律是否正确?

FALSE!

$x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, $z = 1.0$

$(x+y)+z = (-1.5\times10^{38}+1.5\times10^{38}) + 1.0 = 1.0$

$x+(y+z) = -1.5\times10^{38} + (1.5\times10^{38}+1.0) = 0.0$

举例：Ariana火箭爆炸

- 1996年6月4日，Ariana 5火箭初次航行，在发射仅仅37秒钟后，偏离了飞行路线，然后解体爆炸，火箭上载有价值5亿美元的通信卫星。
- 原因是**在将一个64位浮点数转换为16位带符号整数时，产生了溢出异常。**溢出的值是火箭的水平速率，这比原来的Ariana 4火箭所能达到的速率高出了5倍。在设计Ariana 4火箭软件时，设计者确认水平速率决不会超出一个16位的整数，但在设计Ariana 5时，他们没有重新检查这部分，而是直接使用了原来的设计。
- **在不同数据类型之间转换时，往往隐藏着一些不容易被察觉的错误**，这种错误有时会带来重大损失，因此，编程时要非常小心。

举例：爱国者导弹定位错误

- 1991年2月25日，海湾战争中，美国在沙特阿拉伯达摩地区设置的爱国者导弹拦截伊拉克的飞毛腿导弹失败，致使飞毛腿导弹击中了一个美军军营，杀死了美军28名士兵。其原因是由于爱国者导弹系统时钟内的一个软件错误造成的，引起这个软件错误的原因是浮点数的精度问题。
- 爱国者导弹系统中有一内置时钟，用计数器实现，每隔0.1秒计数一次。程序用0.1的一个24位定点二进制小数(常量x)来乘以计数值作为以秒为单位的时间. 这个常量x的二进制表示的机器数是多少呢?
- 0.1的二进制表示是一个无限循环序列：0.00011[0011]...，x=0.000 1100 1100 1100 1100 1100B。显然，x是0.1的近似表示，相差
- 0.1-x = 0.000 1100 1100 1100 1100 1100 [1100]... -

  0.000 1100 1100 1100 1100 1100B，即为(0.小数点后面跟24位)：

  =0.000 0000 0000 0000 0000 0000 1100 [1100]b(小数点后面跟24位)

  $=2^{-20}\times 0.1 \approx 9.54\times 10^{-8}$ (秒)　这就是机器值与真值之间的误差!

举例：爱国者导弹定位错误

已知在爱国者导弹准备拦截飞毛腿导弹之前，已经连续工作了100小时，飞毛腿的速度大约为2000米/秒，则由于时钟计算误差而导致的距离误差是多少?

100小时=计数了100(h)×60(m)×60(s)×10(次数/s)=36×10^5次，因而导弹的时钟已经偏差了((36×10^5)次)×((9.54×10^{-8})秒/次)≈ 0.343秒

因此，距离误差 = (2000米/秒)×(0.343秒) ≈ 687米　(少打那么远)

小故事：实际上，以色列已经发现了这个问题并于1991年2月11日知会了美国陆军及爱国者计划办公室（软件制造商）。以色列建议重启爱国者系统的电脑作为暂时解决方案，可是美国陆军不知道每次需要间隔多少时间重启系统一次。1991年2月16日，制造商向美国陆军提供了更新软件，但在飞毛腿导弹击中军营后的一天才运抵部队。

举例：爱国者导弹定位错误

- 若x用float型表示，则x的机器数是什么？0.1与x的偏差是多少？系统运行100小时后的时钟偏差是多少？在飞毛腿速度为2000米/秒的情况下，预测的距离偏差为多少？
  - 0.1= 0.0 0011[0011]B=+1.1 0011 0011 0011 0011 0011 00B×2^{-4}，故x的机器数为0 011 1101 1 100 1100 1100 1100 1100 1100
  - Float型仅24位有效位数，后面的有效位全被截断，故x与0.1之间的误差为：|x–0.1|=0.000 0000 0000 0000 0000 0000 0000 1100 [1100]...B。这个值等于$2^{-24}\times 0.1 \approx 5.96\times 10^{-9}$。100小时后时钟偏差$5.96\times 10^{-9}\times 36\times 10^{5} \approx 0.0215$秒。距离偏差$0.0215\times 2000 \approx 43$米。比爱国者导弹系统精确约16倍。

举例：爱国者导弹定位错误

- 若用32位二进制定点小数x=0.000 1100 1100 1100 1100 1100 1100 1101 B表示0.1，则误差比用float表示误差更大还是更小?
  - 当x=0.000 1100 1100 1100 1100 1100 1100 1101 B时，与0.1之间的误差约为：|x–0.1|=0.000 0000 0000 0000 0000 0000 0000 00 1100 [1100]...B。这个值等于$2^{-30}\times0.1 \approx 9.31\times10^{-11}$。100小时后时钟偏差$9.31\times10^{-11}\times36\times10^{5} \approx 0.000335$秒。预测的距离偏差仅为$0.000335\times2000 \approx 0.67$米。

举例：浮点数运算的精度问题

- 从上述结果可以看出：
  - 用32位定点小数表示0.1，比采用float精度高64倍
  - 用float表示在计算速度上更慢，必须先把计数值转换为IEEE 754格式浮点数，然后再对两个IEEE 754格式的数相乘，故采用float乘比直接将两个二进制数相乘要慢
- Ariana 5火箭和爱国者导弹的例子带来的启示
- 程序员应对底层机器级数据的表示和运算有深刻理解
- 计算机世界里，经常是“差之毫厘，失之千里”，需要细心再细心，精确再精确
- 不能遇到小数就用浮点数表示，有些情况下（如需要将一个整数变量乘以一个确定的小数常量），可先用一个确定的定点整数与整数变量相乘，然后再通过移位运算来确定小数点

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），编译器需小心处理
  - 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）

定点运算部件

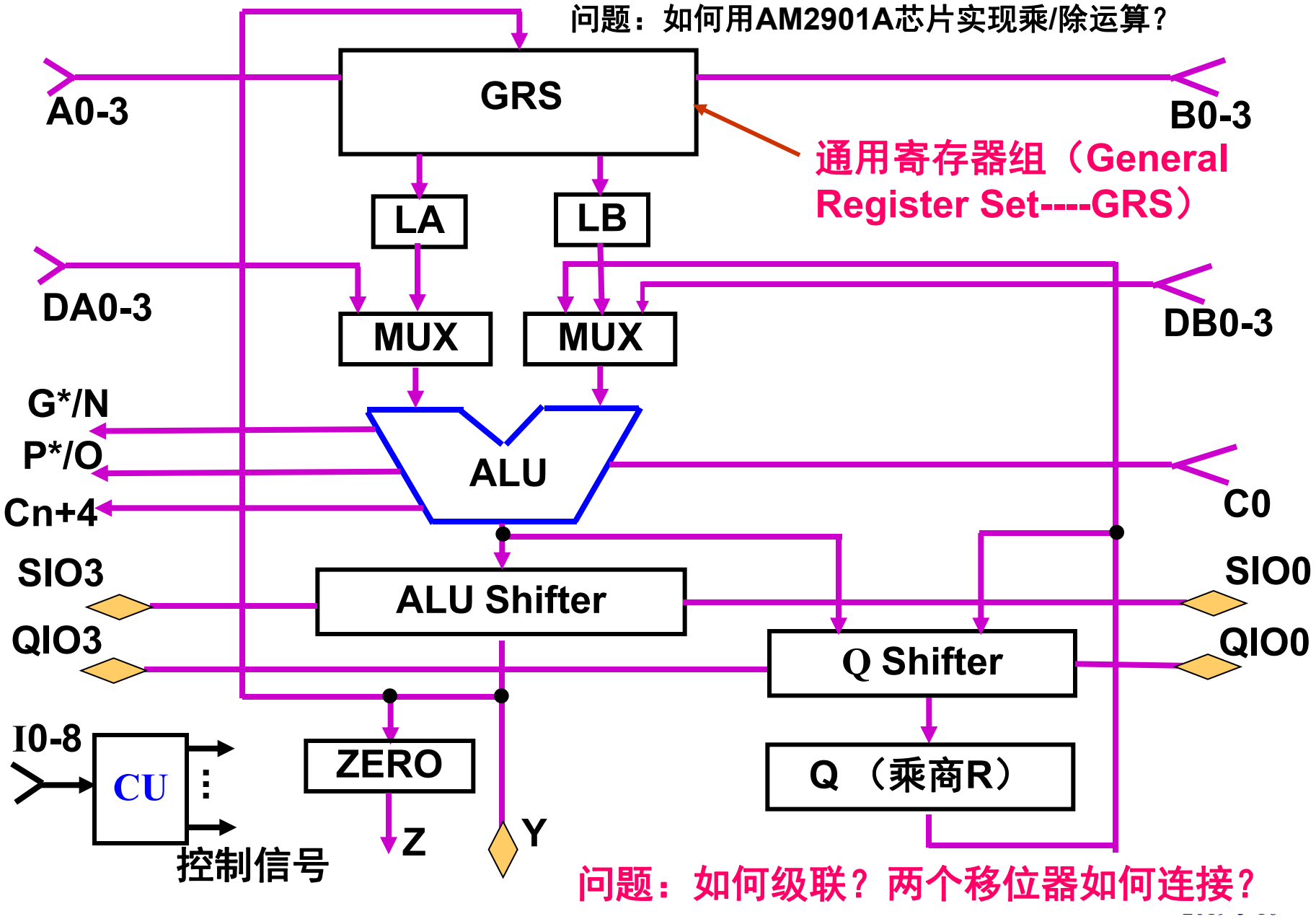
- 综合考虑各类定点运算算法后，发现：
  - 所有运算都可通过“加”和“移位”操作实现

    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”。

“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，只是强调的侧重不同

SKIP

定点运算器芯片举例-AM2901A芯片框图



AM2901A的功能和结构

- 核心是ALU(含移位器)，可实现A+B、A–B、B–A和与、或、异或等
  - 进位入C0、进位出Cn+4、组进位传递/溢出P*/O、组进位生成/符号G*/N
    - 串行级联时，C0和Cn+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

AM2901A的功能和结构

- **思考题：如何用AM2901A芯片实现乘/除运算?**

① R0被预置为0 / 被除数高位，Q寄存器被预置为乘数 / 被除数低位，R1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

第二讲小结

- 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）
    - 就近（中间值强迫为偶数）、+∞方向、-∞方向、0方向
    - 默认为“就近”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

作业

5、7（1）、7（2）、7（4）、11（2）(易错)、12（1）