Computer Organization and Design

Ch3: Arithmetic for Computers

计算机算术运算

1. 定点数的表示和运算
2. 浮点数的表示和运算

第一讲：定点数的表示和运算

主　要　内　容

- 指令集中与定点运算相关的指令（ 以**MIPS**为参考 ）
  - 涉及到的操作数
    - 无符号整数
    - 带符号整数
    - 逻辑数
  - 涉及到的运算
    - 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
- 基本运算部件**ALU**的设计
- 利用**ALU**部件和移位器实现乘除运算

MIPS定点算术运算指令

Instruction	*Example*	*Meaning*	*Comments*
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 x $3	64-bit signed product
multiply unsigned	multu$2,$3	Hi, Lo = $2 x $3	64-bit unsigned product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
divide unsigned	divu $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $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	$s1 = ~ ($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 << 10
shift right logical	srl	$s1,$s2,10	$s1 = $s2 >> 10

涉及到的操作数：**32/16**位 逻辑数

涉及到的操作：与 / 或 / 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)	$s1 = Memory[$s2 + 100]
store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1
load half unsigned	lhu $s1,100($s2)	$s1 = Memory[$s2 + 100]
store half	sh $s1,100($s2)	Memory[$s2 + 100] = $s1
load byte unsigned	lbu $s1,100($s2)	$s1 = Memory[$s2 + 100]
store byte	sb $s1,100($s2)	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 无符号数装入时，进行的是**0**扩展

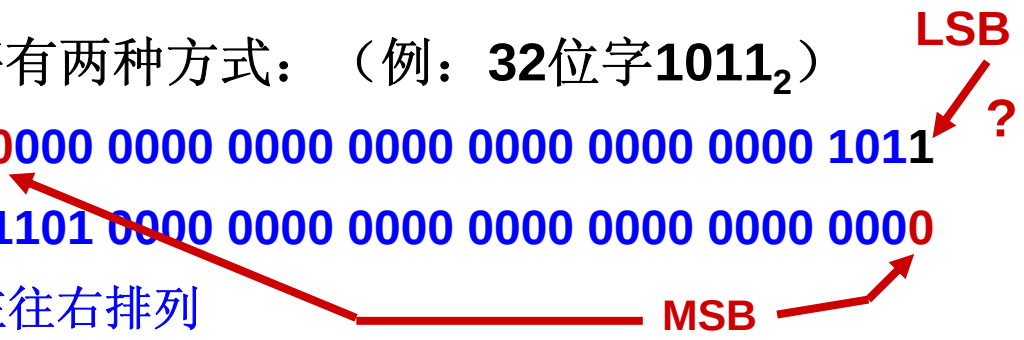
MIPS定点指令考察的结果

- 涉及到的操作数
  - 无符号整数
  - 带符号整数
  - 逻辑数
- 涉及到的运算
  - 算术运算
    - 带符号整数运算：取负 / 符号扩展 / 加 / 减 / 乘 / 除
    - 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

实现MIPS定点运算指令的思路：首先实现一个能进行基本算术运算（加/减）和基本逻辑运算（与/或/或非）、并能生成基本条件码（ZF/VF/SF/NF）的ALU，然后再由ALU和移位器实现乘除运算器。

回顾： Unsigned integer(无符号整数)

- 机器中字的位排列顺序有两种方式：（例：32位字1011_2）
  - 高到低位从左到右：0000 0000 0000 0000 0000 0000 0000 1011
  - 高到低位从右到左：1101 0000 0000 0000 0000 0000 0000 0000
  - MIPS采用高到低从左往右排列
  - Leftmost和rightmost这两个词有歧义，故用least significant bit, LSB来表示最低有效位，用MSB来表示最高有效位



- 若一个字为n位，则可表示的不同模式的字有2^n个
  - N=4时，16种模式为0000 ~ 1111
- 一般在全部是正数运算且不出现负值结果的场合下，可使用无符号数表示。例如地址运算
- 无符号数的各位编码中没有符号位
- 在字长相同的情况下，它的表示范围大于有符号数
- 无符号数总是整数。最大8位无符号整数是11111111B，其值为255

回顾： Signed integer（带符号整数）

- 必须让计算机能够处理正数**(positive)** 和负数**(negative)**，计算机用 **MSB**来表示数的符号
- 有三种表示方式
  - **Signed magnitude** （原码）

    用来表示浮点（实）数的尾数
  - **One's complement** （反码）

    现已不用
  - **Two's complement** （补码）

    50年代以来，所有计算机都用补码来表示定点（整）数

回顾： Sign and Magnitude （原码的表示）

Decimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Decimal	Binary
-0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111

- 容易被人理解，但是：
  - 0 的表示不唯一，不利于程序员编程.
  - 加、减不统一.
  - 需要额外对符号位进行处理，不利于硬件设计.
  - 特别当 a<b时，实现 a-b比较困难

因此，从 50年代开始，整数都采用补码来表示！

回顾： Two's Complement （补码的表示）

- 负数的补码表示
  - **Bitwise inverse and add 1**（各位取反，末位加**1**）
  - 负数的 **MSB**总是**"1" => sign bit**
- **Biggest 4-bit Binary Number: 7　　Smallest 4-bit Binary Number: - 8**

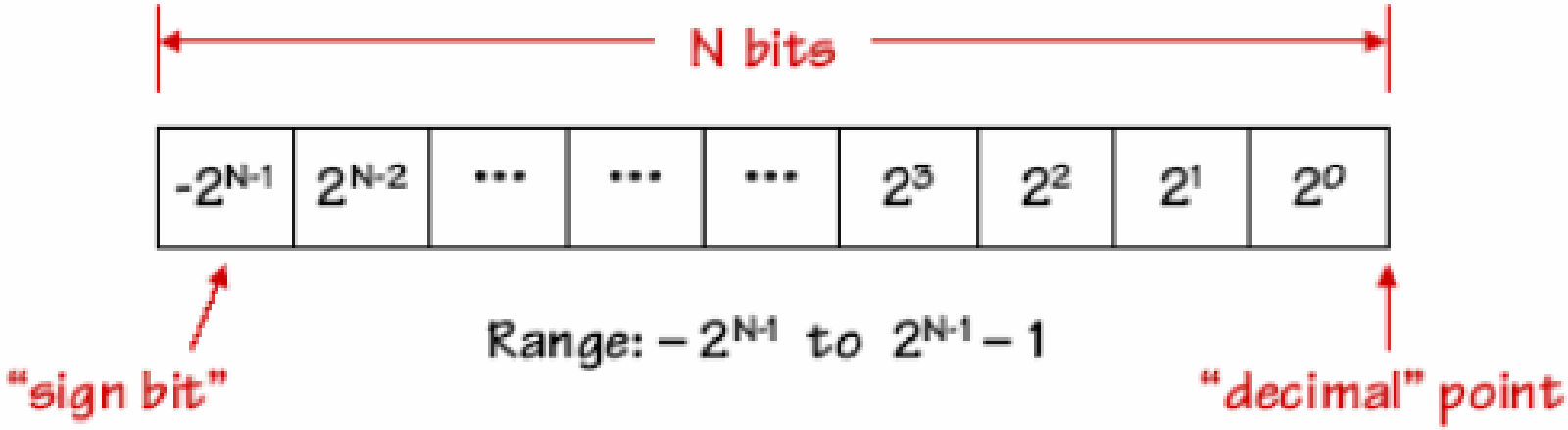
Decimal	Binary	Decimal	Bitwise Inverse	2's Complement
0	0000	-0	1111	0000
1	0001	-1	1110	1111
2	0010	-2	1101	1110
3	0011	-3	1100	1101
4	0100	-4	1011	1100
5	0101	-5	1010	1011
6	0110	-6	1001	1010
7	0111	-7	1000	1001
8	1000	-8	0111	1000

+0和-0表示唯一

值太大，用4位无法表示，"溢出"！ "Illegal"正数!

回顾：如何求补码的值

根据补码各位上的“权”，可以求出一个补码的值



8-bit 2's complement example:

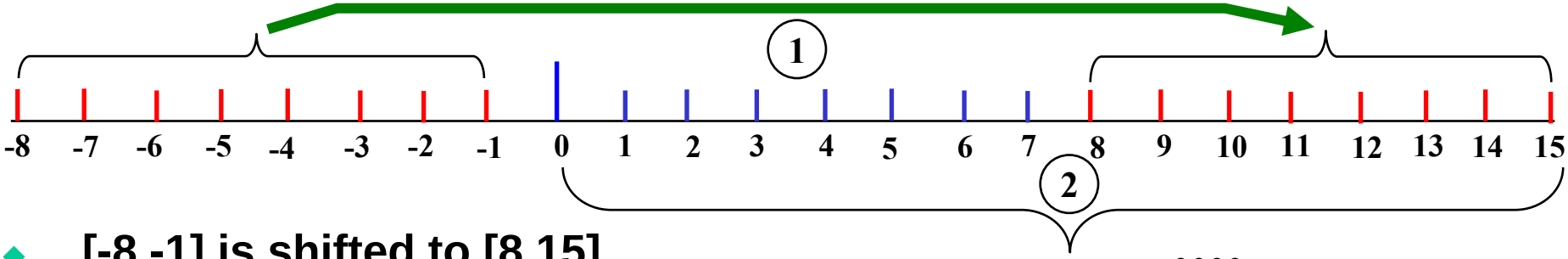
$11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -128 + 64 + 16 + 4 + 2 = -42$

当N=4时，范围为：$-2^3 \sim 2^3 - 1$ （即：-8 ~ +7）

当N=32时，范围为：$-2^{31} \sim 2^{31} - 1$

BACK to Booth Multiply

回顾：补码特性 - 模运算



- **[-8,-1] is shifted to [8,15].**
- **The modul here is 10000**
- 时钟是一种模-12系统

假定钟表时针指向10点，要将它拨向 6 点，则有两种拨法：

① 倒拨4格：10-4=6

② 顺拨8格：10+8=18≡6(mod 12)

模12系统中：　10-4 ≡ 10+8(mod 12)

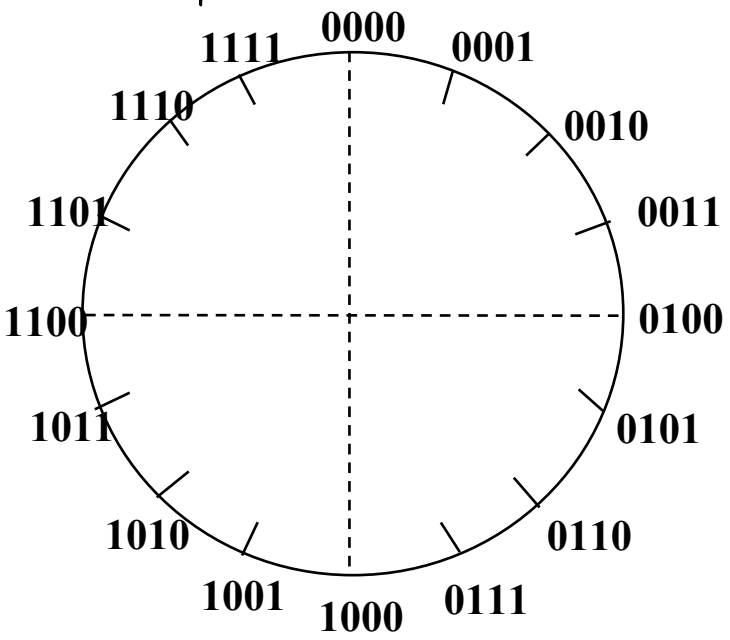
-4≡ 8 (mod 12)

则，称8是- 4对模12的补码。

同样有 -3 ≡ 9（mod 12）

-5 ≡ 7（mod 12）等

补码（modular运算）：+ and – 的统一



“对于某一确定的模，某数减去小于模的另一数，总可以用该数加上另一数的补码来代替”。

回顾：补码的特性

- 可用加法实现减法
- 加法运算时，符号位同数值位一样参加运算
  - 可以用无符号数加法器实现带符号数加法，只不过对**MSB**的解释不同
  - **MSB**向前面的进位就是（带符号数或无符号数）和的进位

例1：“钟表”模运算系统

10-4=10+(12-4)=10+8=6 （mod 12）

例2：“4位十进制数” 模运算系统（相当于只有四档的算盘）

$9828-1928=9828+(10^4-1928)$

$=9828+8072$

$=1\,7900$

$=7900$（mod 10^4）

取模的含义就是只留余数，高位的“1”被丢弃！

- 取负（**Negate**）：各位取反，末尾加**1**

  例：已知$[x]_{补}=1001\ 0010$，则：$[-x]_{补}=0110\ 1101+1=0110\ 1110$

- 符号扩展操作（**Sign extension**）：符号位前面添加若干符号位

  例：已知$[x]_{补}=1001\ 0010$，扩展8位后 $[x]_{补}=1111\ 1111\ 1001\ 0010$

带符号数和无符号数的比较

- 扩充操作有差别
  - **MIPS提供了两种加载指令**
    - 无符号数：**lbu $t0, 0($s0) ；$t0的高24位补0 （称为0扩展）**
    - 带符号数： **lb $t0, 0($s0) ；$t0的高24位补符号 （称为符号扩展）**
- 数的比较有差异
  - 无符号数：**MSB为1的数比MSB为0的数大**
  - 带符号数： **MSB为1的数比MSB为0的数小**
  - **MIPS中提供不同的比较指令，如：**
    - 无符号数：**sltu $t0, $s0, $s1**
    - 带符号数： **slt $t1, $s0, $s1**

  假定： **$s0=1111 1111 1111 1111 1111 1111 1111 1111**

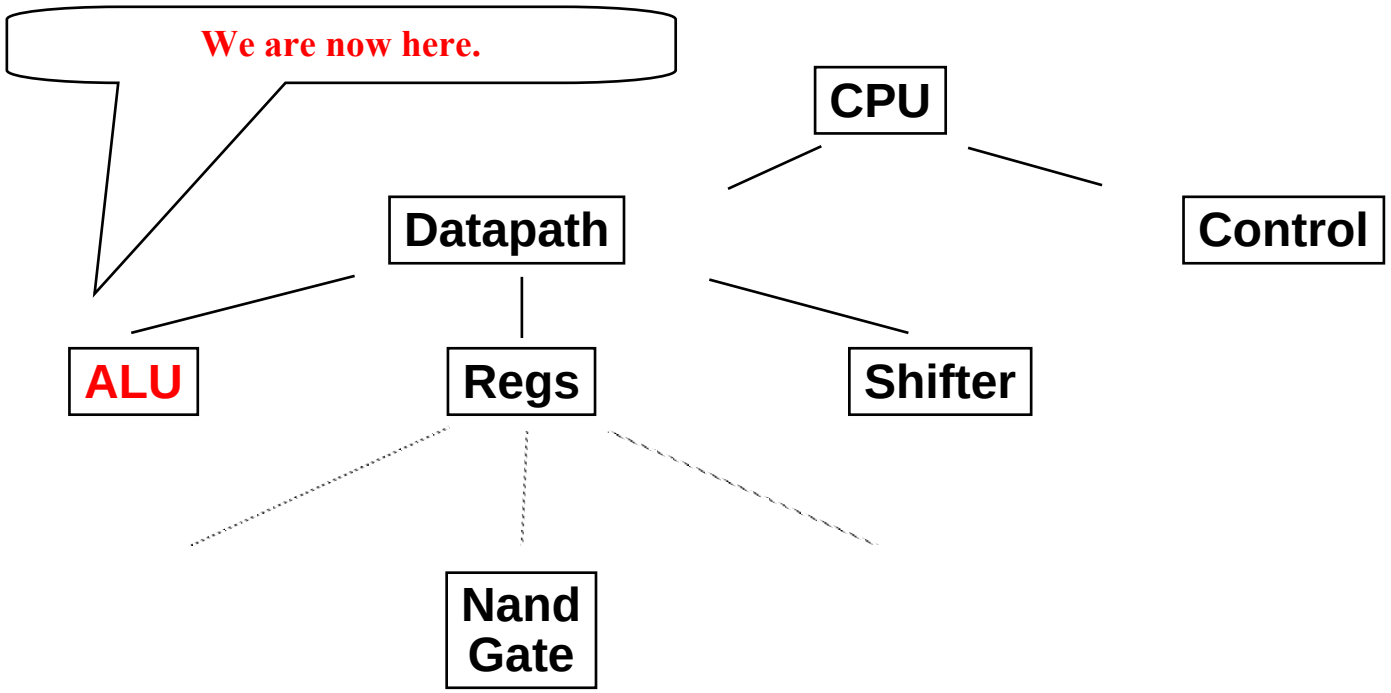
  $s1=0000 0000 0000 0000 0000 0000 0000 0001

  则：**$t0和$t1分别为多少？**

  答案：**$t0和$t1分别为0和1。**
- 溢出判断有差异（无符号数根据最高位是否有进位来判断溢出）
  - **MIPS规定：无符号数运算溢出时，不产生“溢出异常”**

C语言中，无符号数：unsigned int (short / long)；带符号数： int (short / long)

Design Process（ALU设计过程）



ALU（Arithmetic logic unit）：算术逻辑部件

功能：能进行基本算术、逻辑运算，并生成条件码

回顾： Design as Representation

(1) Functional Specification(功能说明) "VHDL Behavior"

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

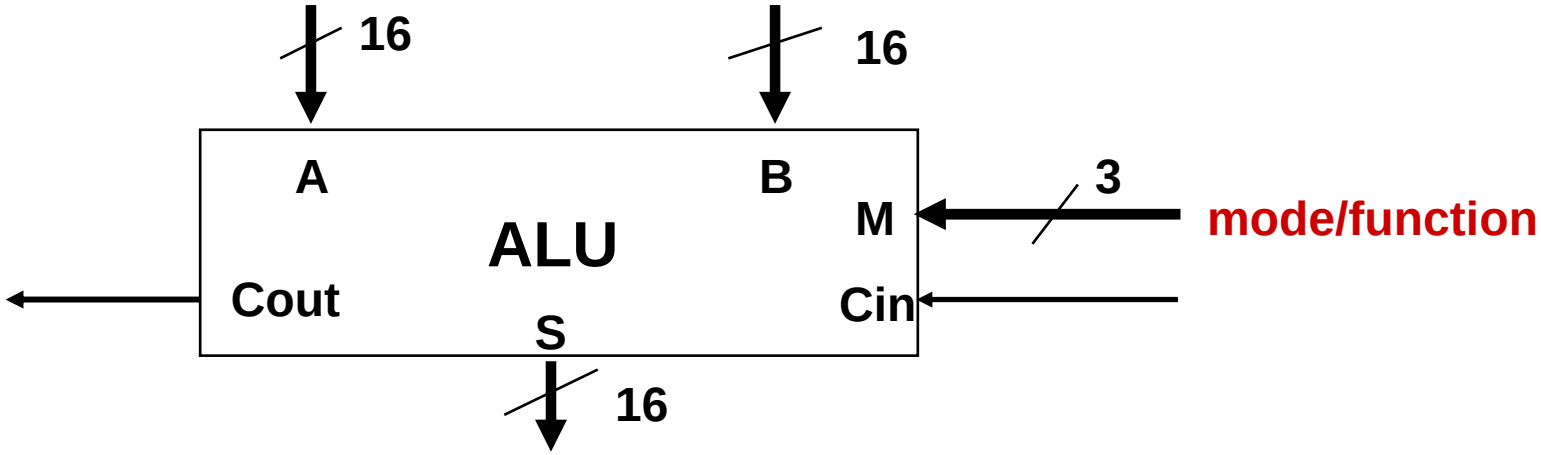
Outputs: 1 x 16 bit result－S; 1 bit carry output－Cout.

Operations: PASS, ADD (A plus B plus Cin), SUB (A minus B minus Cin), AND, XOR, OR, COMPARE (equality)

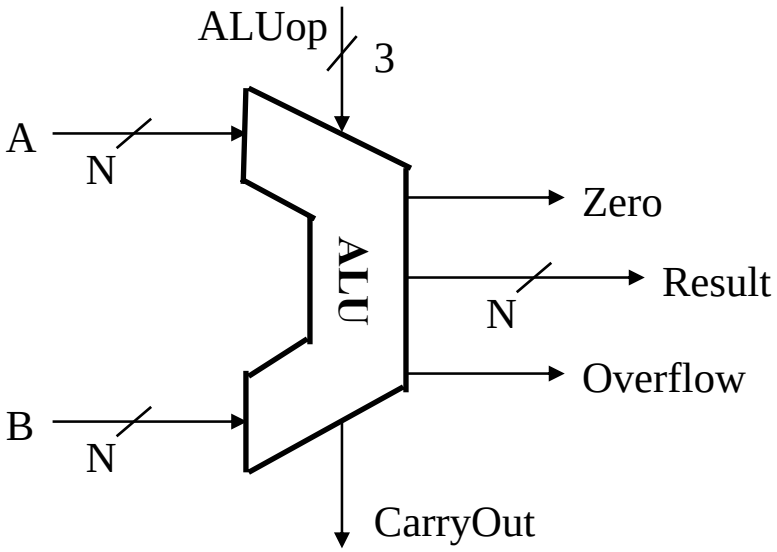
VHDL (Very-High-Speed Integrated Circuit Hardware Description Language) 1987 年底IEEE和美国国防部确定其为标准硬件描述语言

(2) Block Diagram（框图表示） "VHDL Entity"

Understand the data and control flows



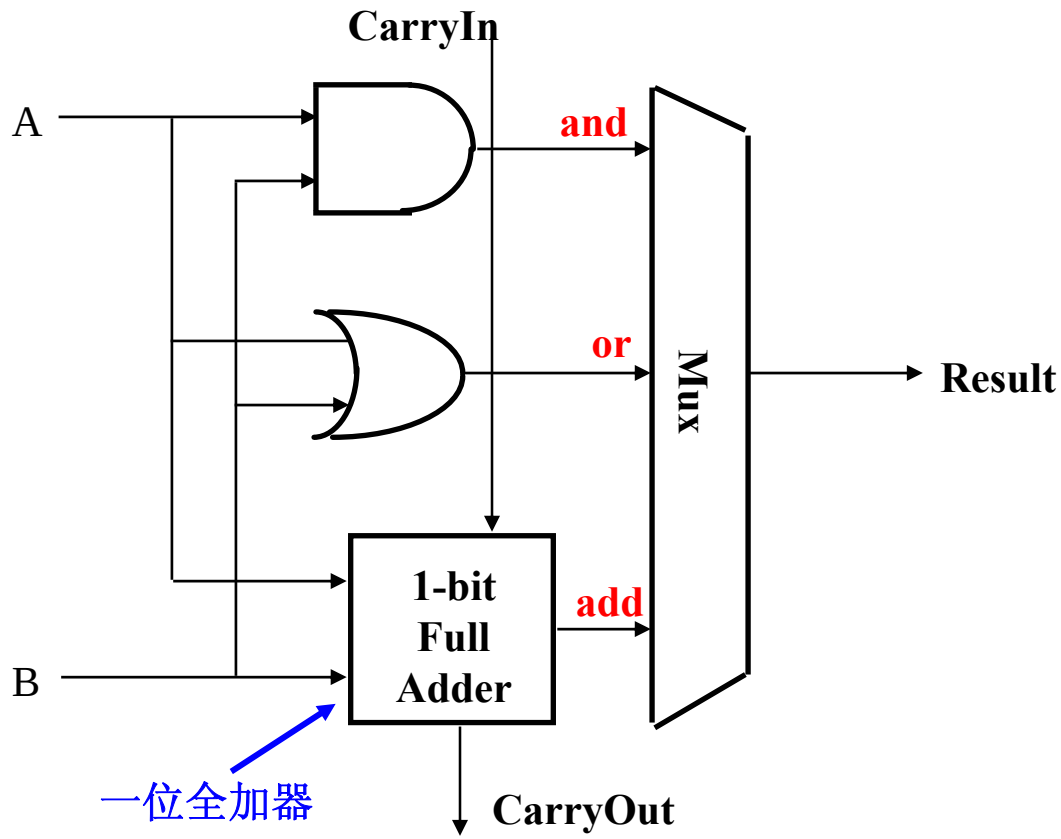
回顾： ALU的功能说明



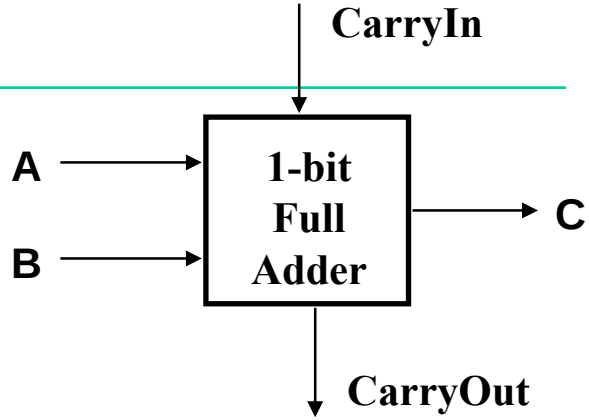
- **ALU Control Lines (ALUop)** — **Function**
  - **000** — **And**
  - **001** — **Or**
  - **010** — **Add**
  - **110** — **Subtract**
  - **111** — **Set-on-less-than**

回顾： A One Bit ALU

- This 1-bit ALU will perform AND, OR, and ADD



回顾： Full Adder(全加器)

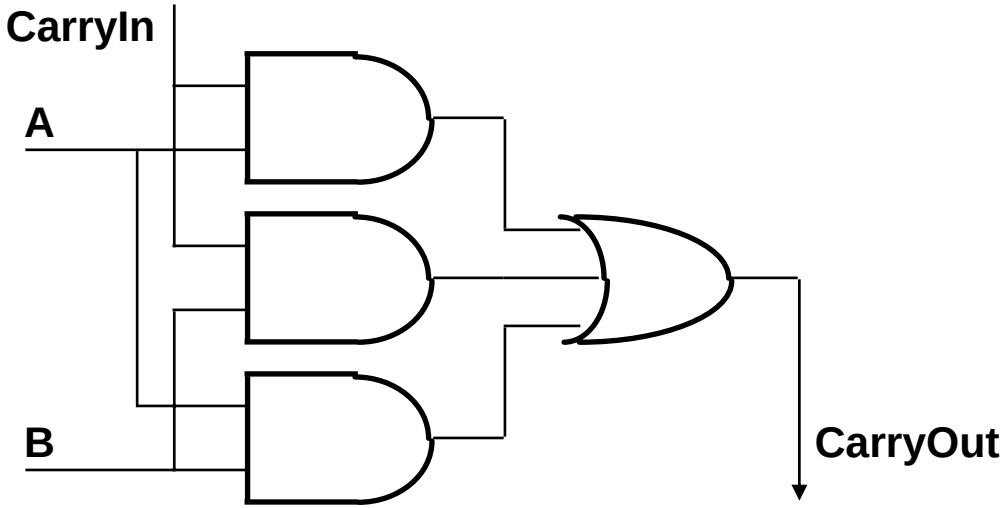


- This is also called a (3, 2) adder
- Half Adder (半加器) : No CarryIn nor CarryOut
- Truth Table:

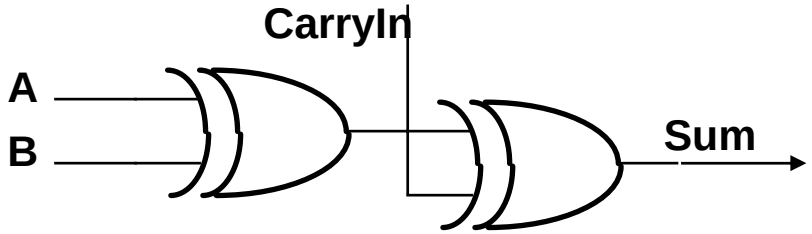
Inputs			Outputs		
A	B	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	0 + 0 + 0 = 00
0	0	1	0	1	0 + 0 + 1 = 01
0	1	0	0	1	0 + 1 + 0 = 01
0	1	1	1	0	0 + 1 + 1 = 10
1	0	0	0	1	1 + 0 + 0 = 01
1	0	1	1	0	1 + 0 + 1 = 10
1	1	0	1	0	1 + 1 + 0 = 10
1	1	1	1	1	1 + 1 + 1 = 11

回顾： CarryOut and Sum

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

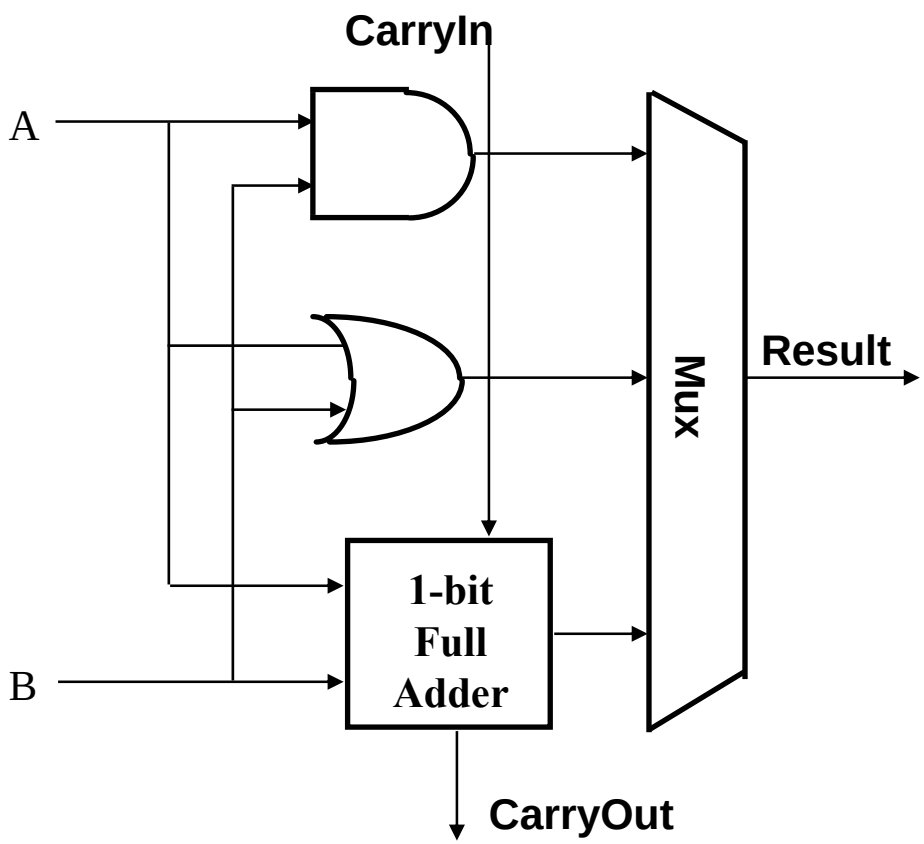


- Sum = A XOR B XOR CarryIn

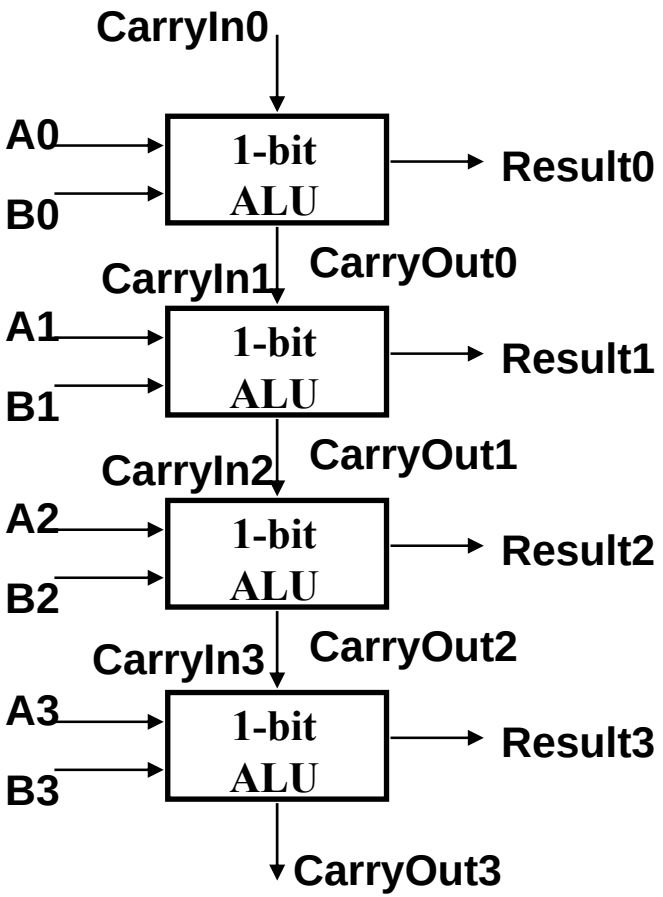


回顾： A 4-bit ALU

1-bit ALU



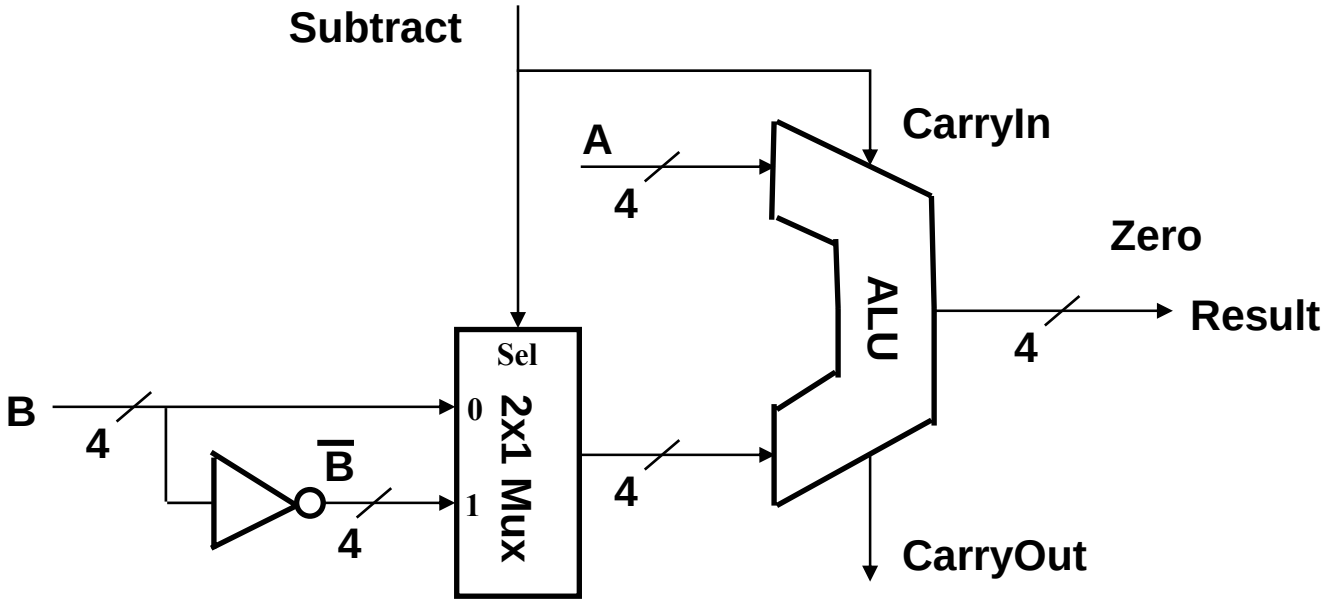
4-bit ALU



Mux是什么？（数字电路课学过）

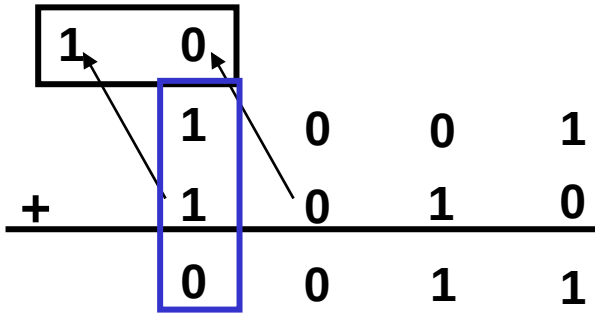
Subtraction?

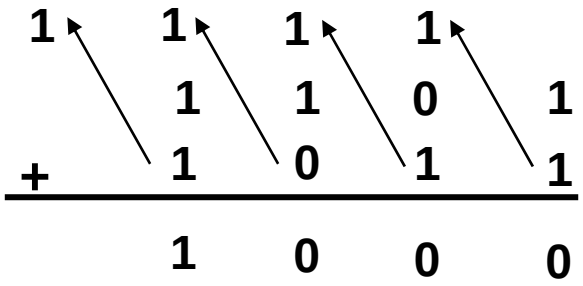
- Keep in mind the followings:
  - (A－B) is the that as: A＋(－B)
  - 2's Complement: inverse of every bit and add 1
- Bit-wise inverse of B is $\overline{B}$:
  - A ＋ $\overline{B}$ ＋ 1 = A ＋ ($\overline{B}$ ＋ 1) = A＋(－B) = A－B



补码加减运算与“溢出”判断

- **Examples1: -7 - 6 = -7 + (- 6) = +3** X　　**-3 - 5 = - 3 + (- 5) = - 8** V





溢出现象：(1) 最高位和次高位的进位不同

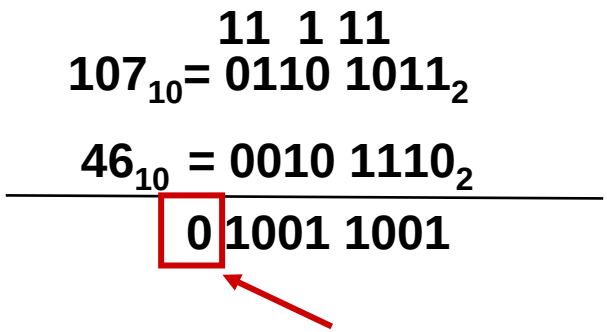
(2) 和的符号位和加数的符号位不同

- **Example2:** 用8位补码求 **107** 和 **46**的“和”

结果错误: **107 + 46 = -103.**

有两种“溢出”判断规则:

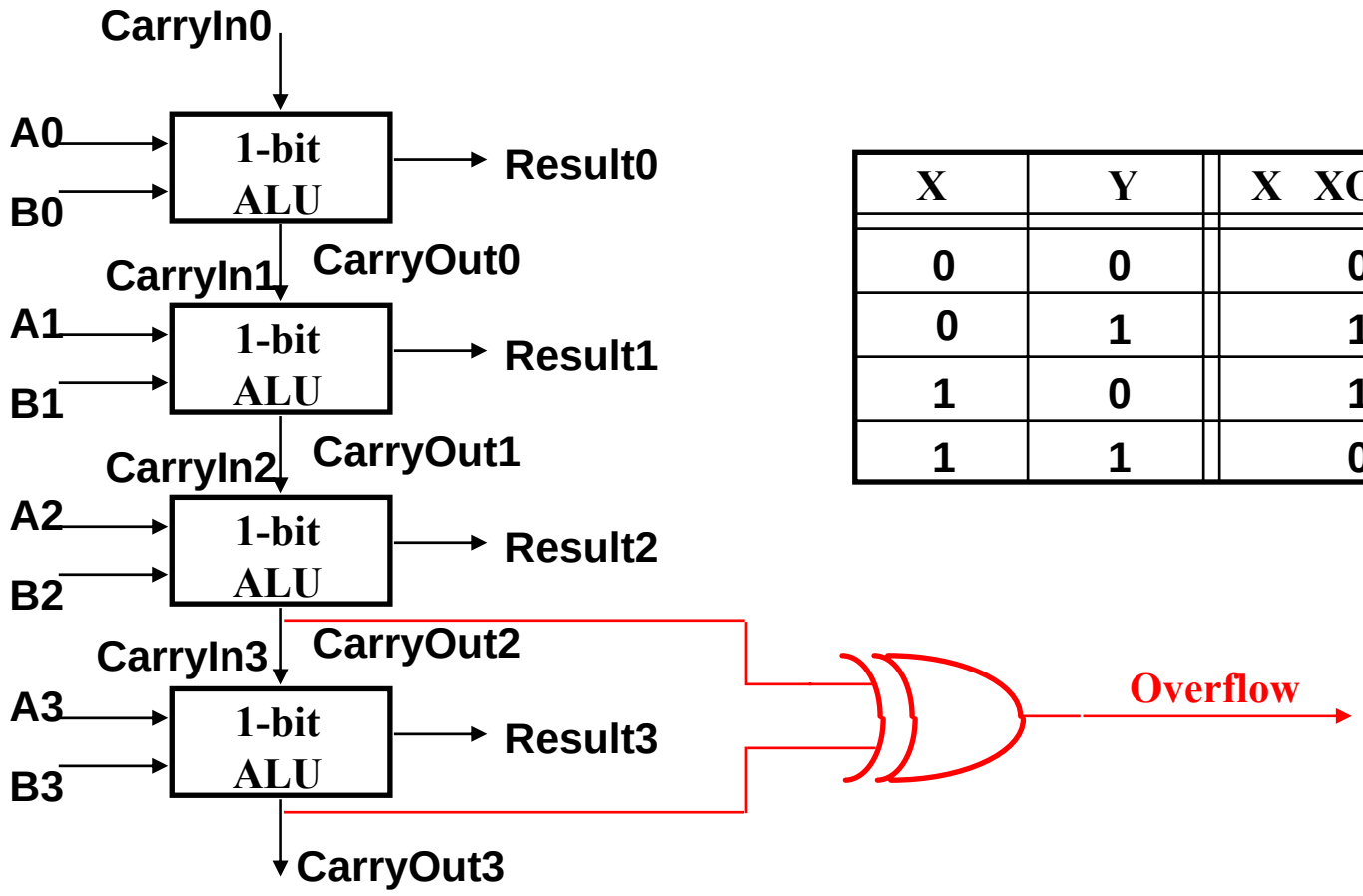
1. 和的符号位和加数的符号位不同
2. 最高位和次高位的进位不同



溢出时，符号位的进位是真正的符号：**+153**

Overflow Detection Logic(溢出判断逻辑)

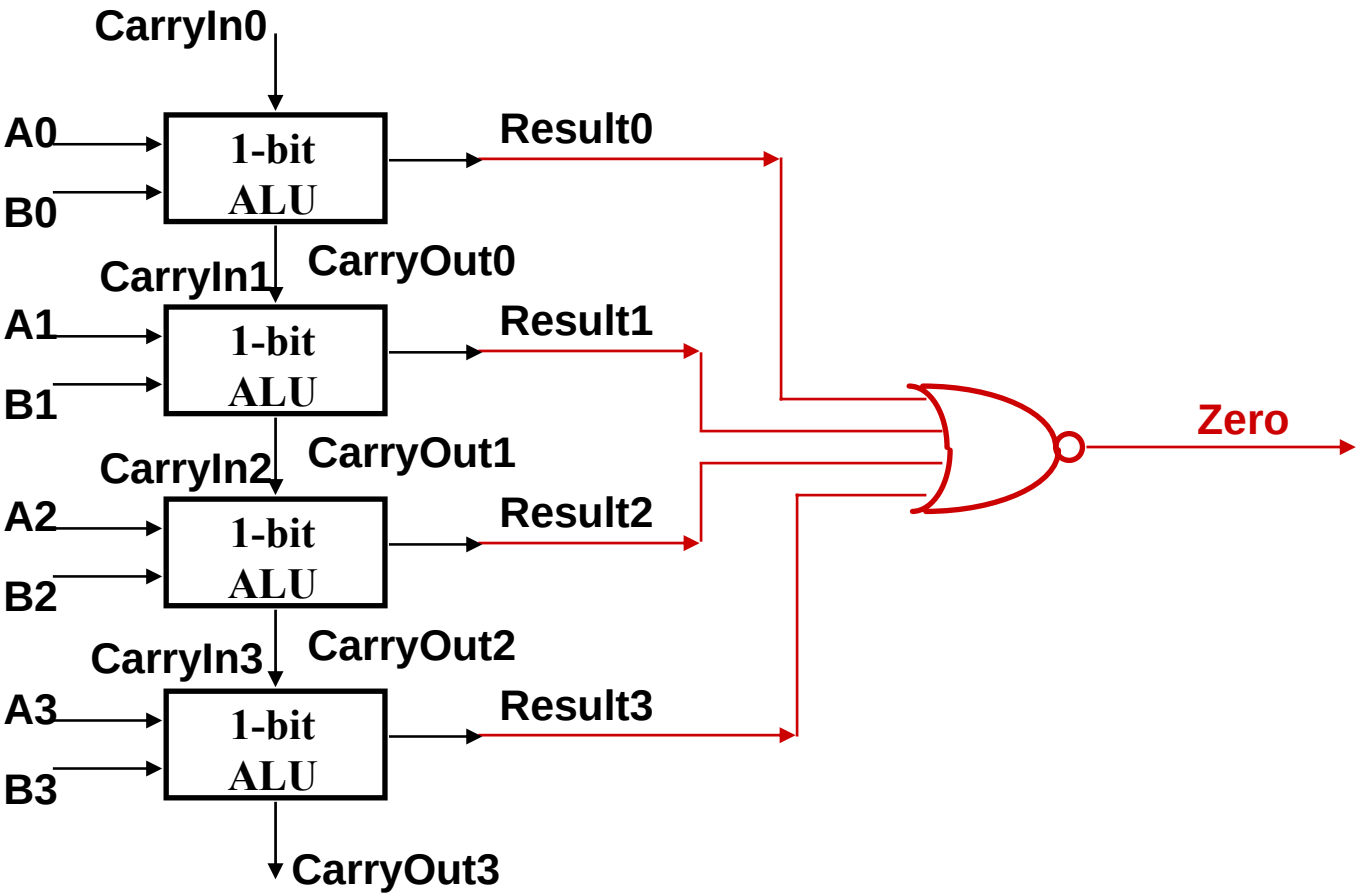
- Carry into MSB ! = Carry out of MSB
  - For a N-bit ALU: Overflow = CarryIn [N - 1] XOR CarryOut [N - 1]



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

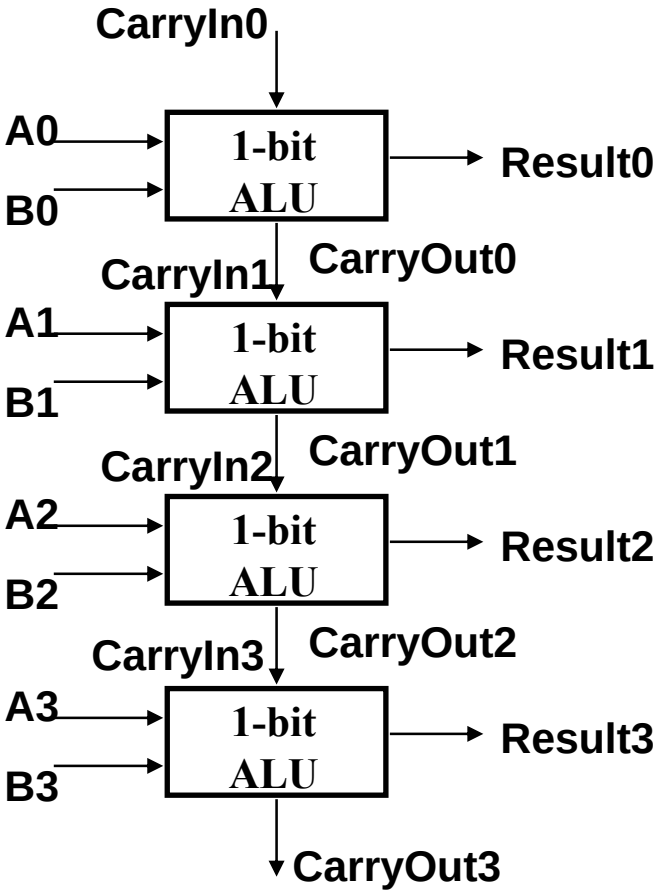
Zero Detection Logic(判0逻辑)

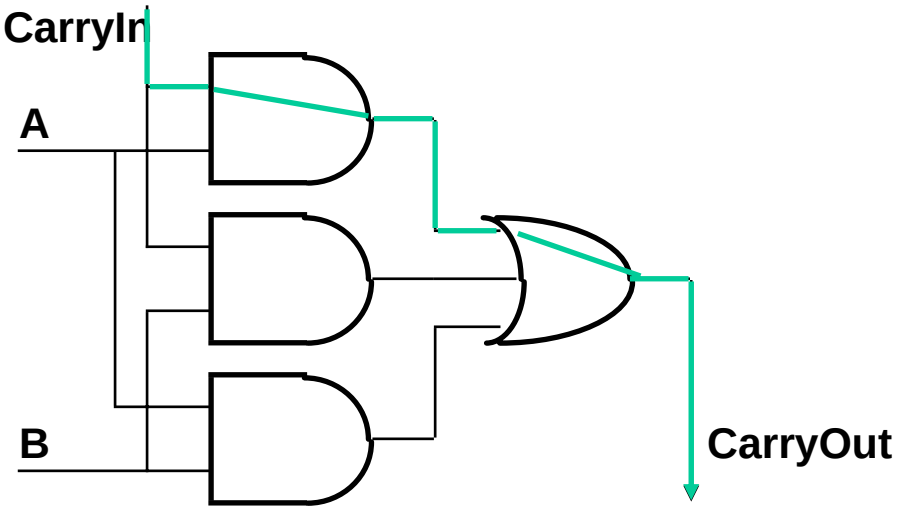
- Zero Detection Logic is just a one BIG NOR gate
  - Any non-zero input to the NOR gate will cause its output to be zero



回顾：Ripple Carry（行波进位的不足）

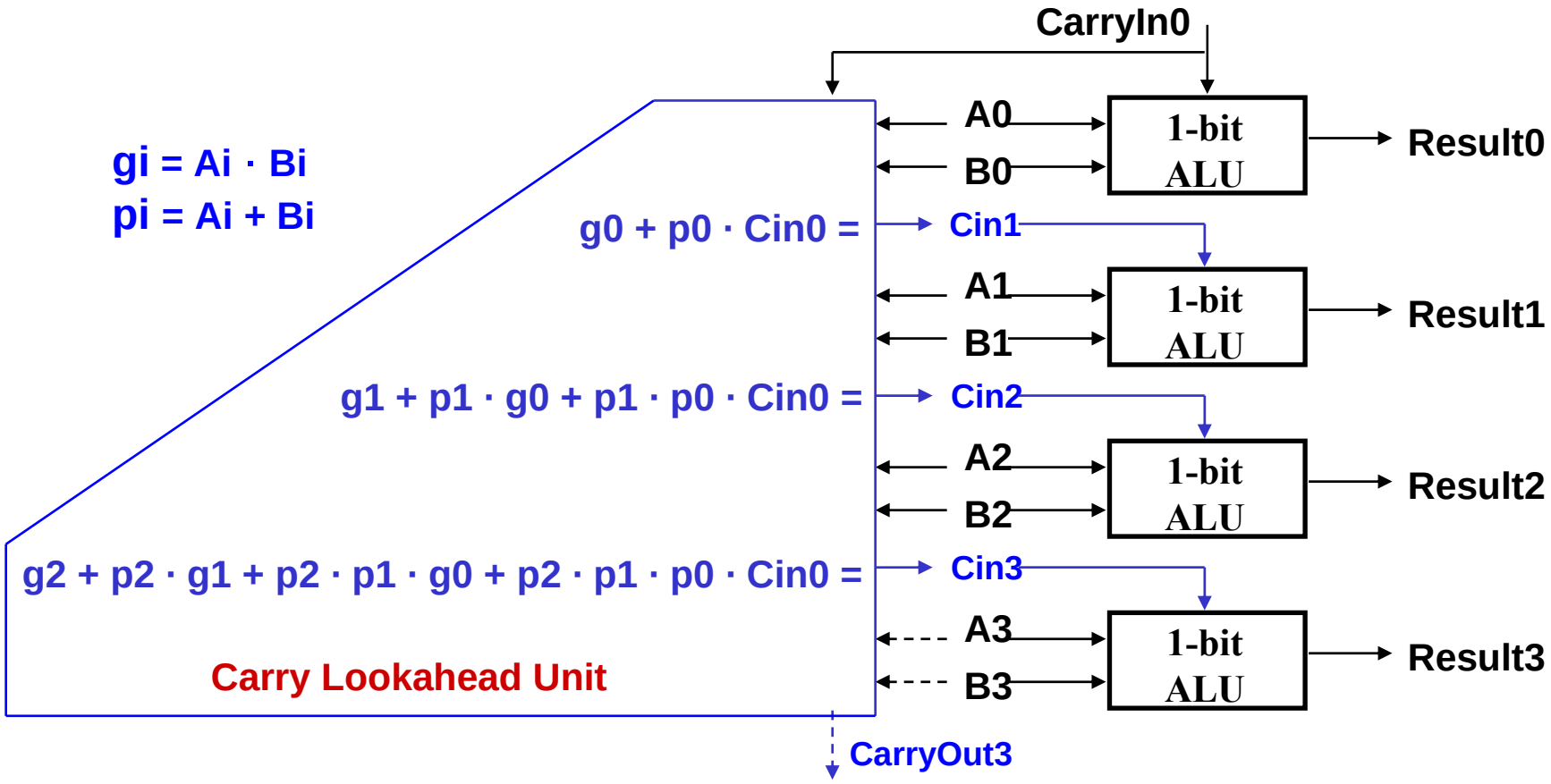
- **The adder we just built is called a "Ripple Carry Adder"(行波进位加法器)**
  - **The carry bit may have to propagate(传递) from LSB to MSB**
  - **Worst case delay for a N-bit adder: 2N-gate delay**





先行进位加法器：Carry Lookahead Adder

- $Cin_i = f(g_0,g_1,\ldots g_{i-1},p_0,p_1,\ldots p_{i-1}, Cin_0) = f(A_0,A_1,\ldots A_{i-1},B_0,B_1,\ldots B_{i-1}, Cin_0)$



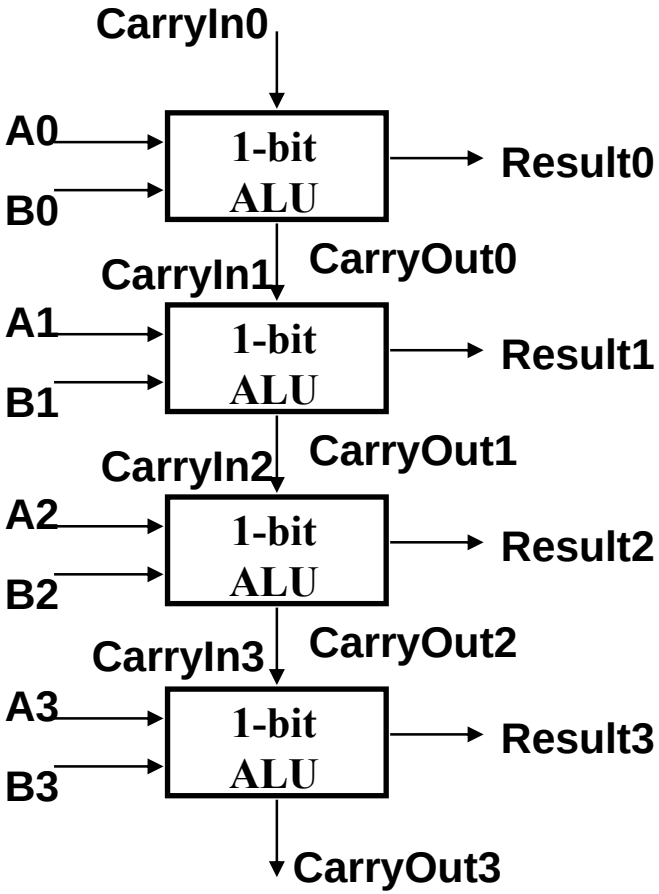
Calculation of Cini can be quickly started since it is based on all initial inputs.

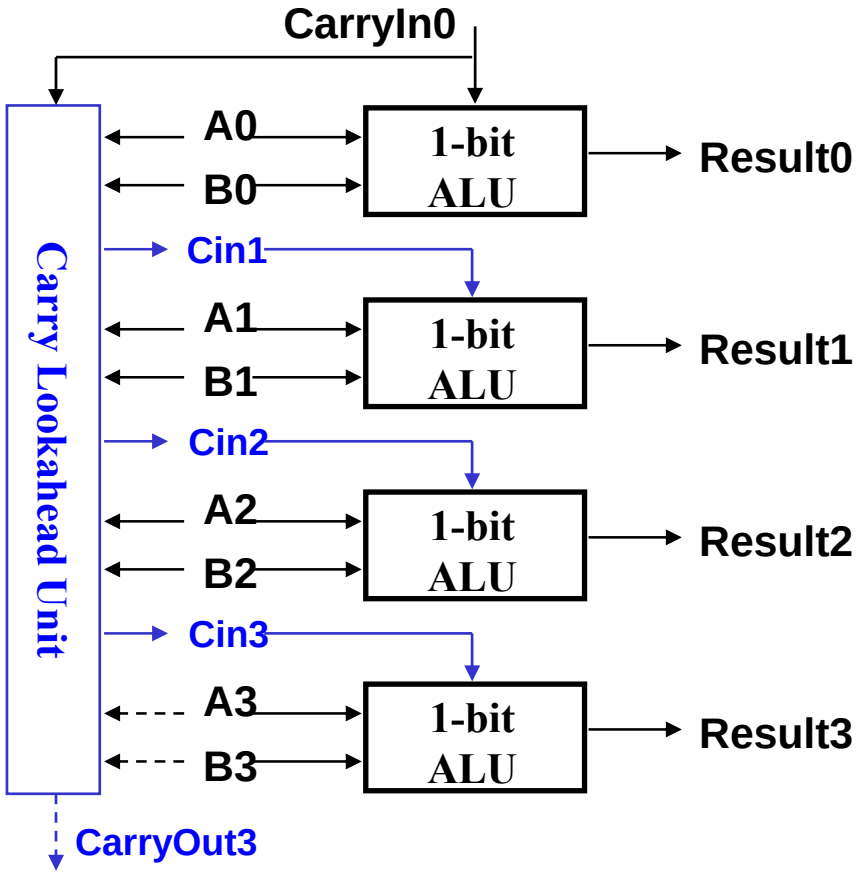
各Cini 的延迟： 2 levels of gates.

根据Si= Ai ⊕ Bi ⊕ Cini ， 可并行求出各位和，各位和的 延迟：3 gates

] 5 gates

Compare Ripple Carry and Carry Lookahead





2N-gate delay

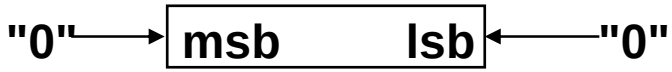
？ N-gate delay： 5 gates

N=16时，提高了 32 / 5 ≈ 6倍

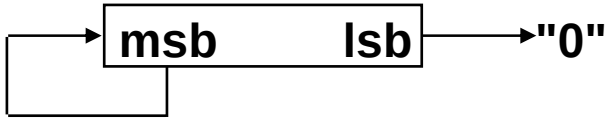
Shifter (移位)

Three different kinds:

***logical*-- value shifted in is always "0"**



***arithmetic*-- on right shifts, sign extend**



***rotating*-- shifted out bits are wrapped around (not in MIPS)**

MIPS指令中可以用**“shamt”**字段指定移多少位。

移位操作用**“移位器”**实现。

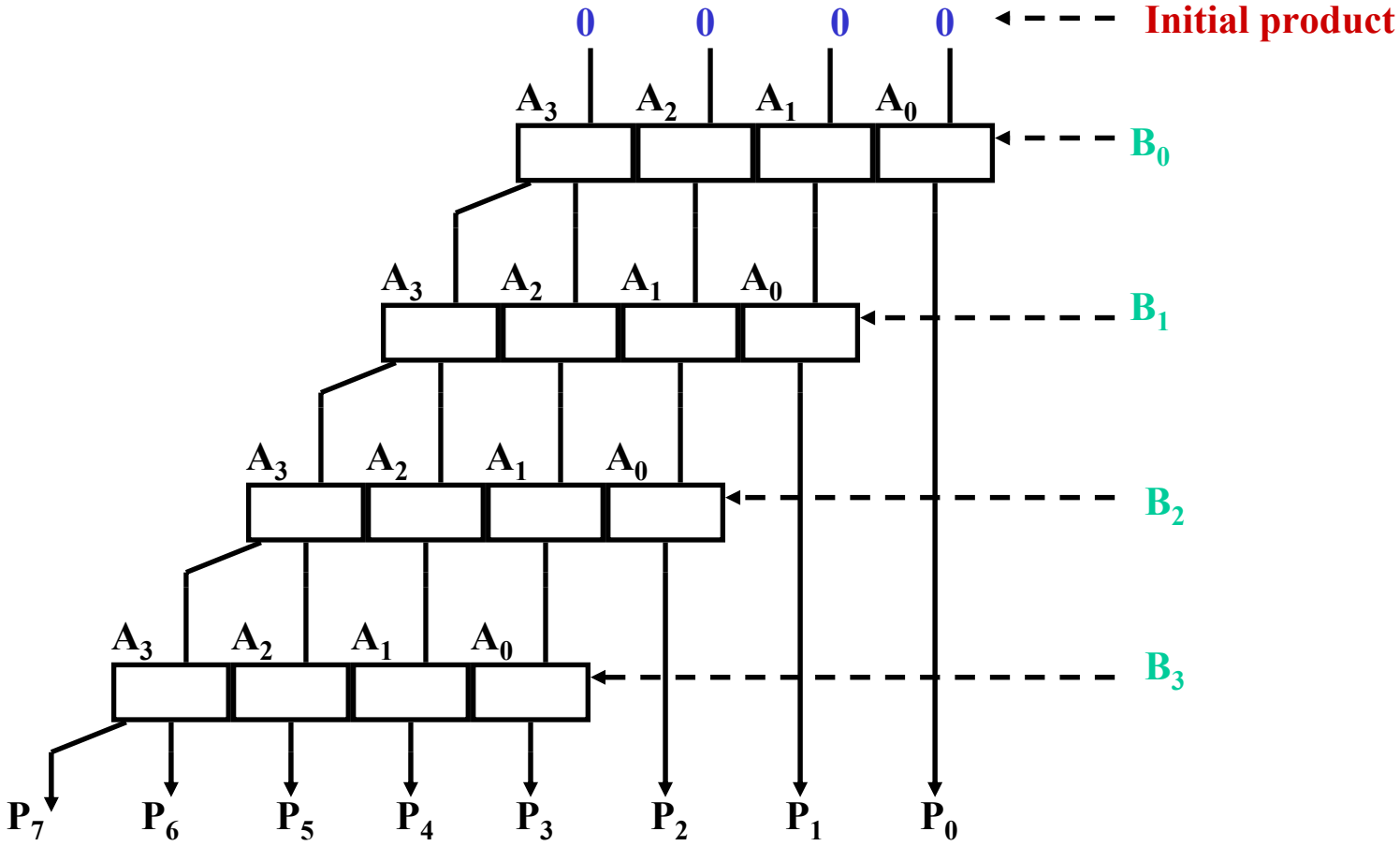
MULTIPLY (乘法)

- Paper and pencil example:

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

- m bits x n bits = m+n bit product
- 二进制乘法容易：每一步只有两种选择
  - 1 => place multiplicand (1 x multiplicand)
  - 0 => place 0 (0 x multiplicand)
- 3种乘法方法：逐步求精的过程（从模拟笔算方式开始逐步简化）

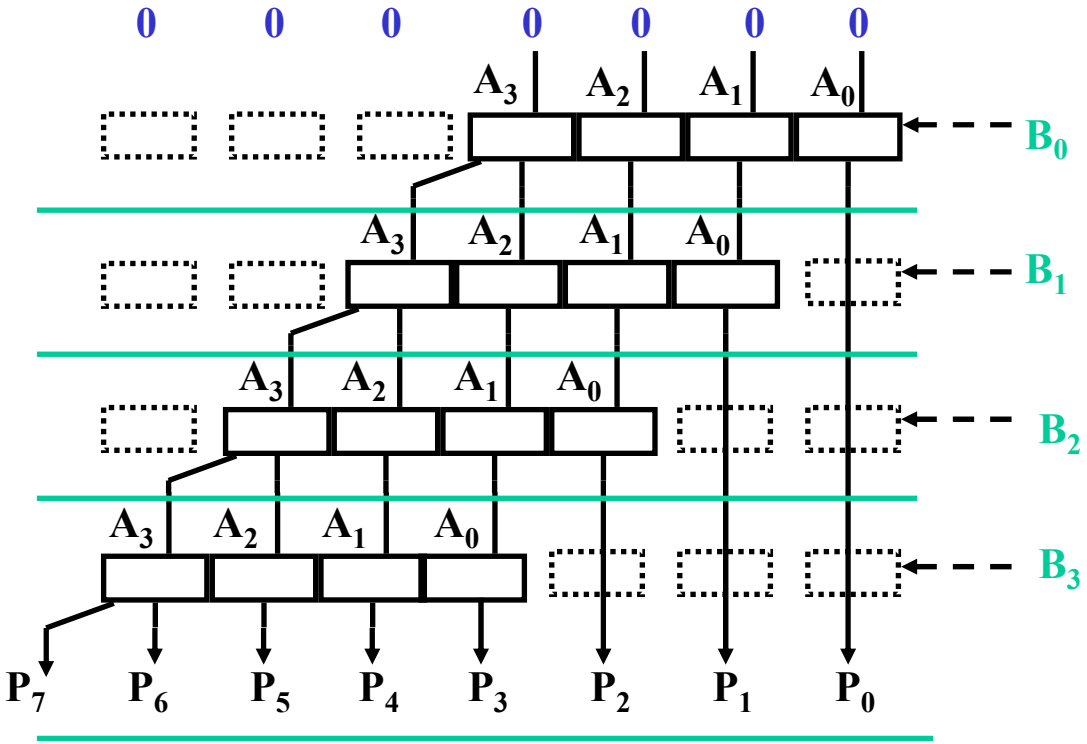
无符号整数乘法



◆**Stage i accumulates $A * 2^i$ if $B_i == 1$**

完全模拟手工乘法：每次被乘数和Bi相乘，然后左移一位后和上次部分积相加。

How does it work?



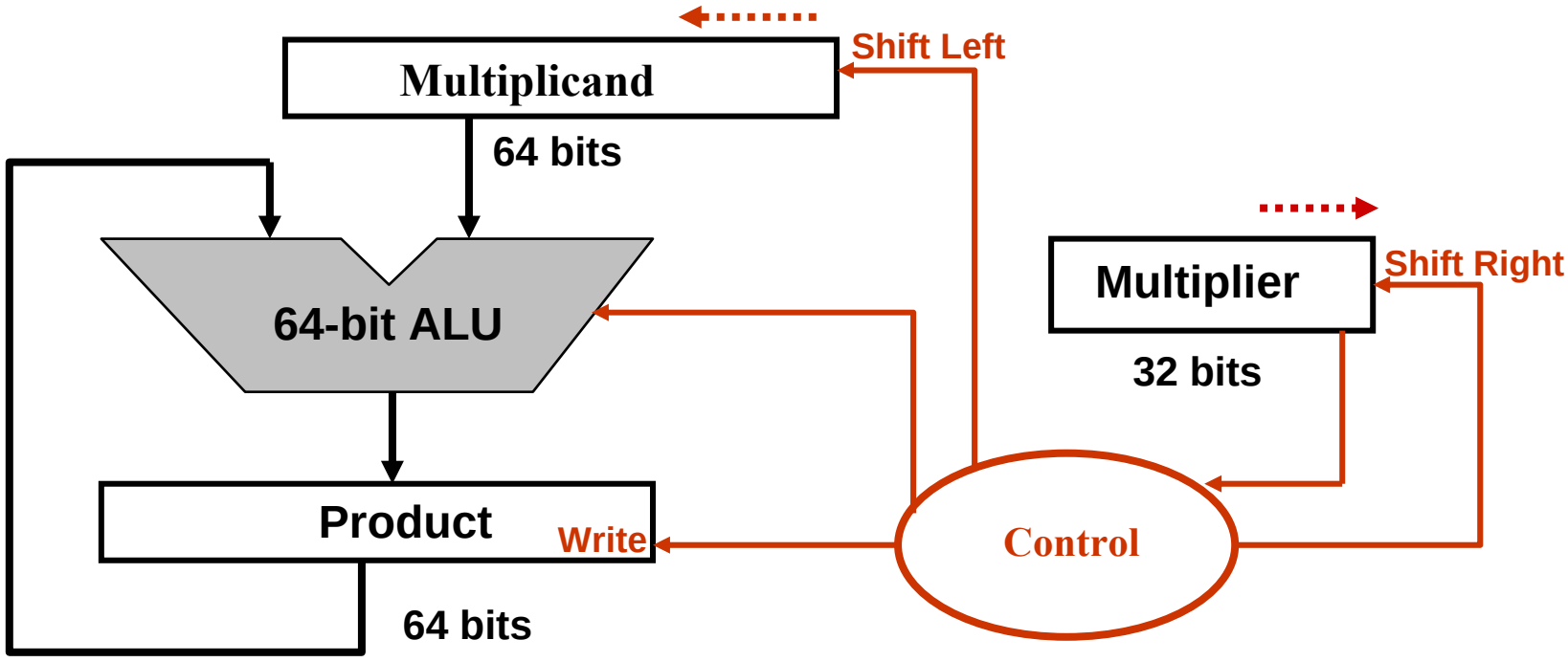
- 每一步将被乘数A左移一位 (x 2)
- 根据B的下一位是否为1确定是否加被乘数A
- 每一步的部分积的位数是2n

整个运算过程中用到两种操作：加法 + 左移

怎样用ALU和移位器来实现？

乘法算法1的硬件实现

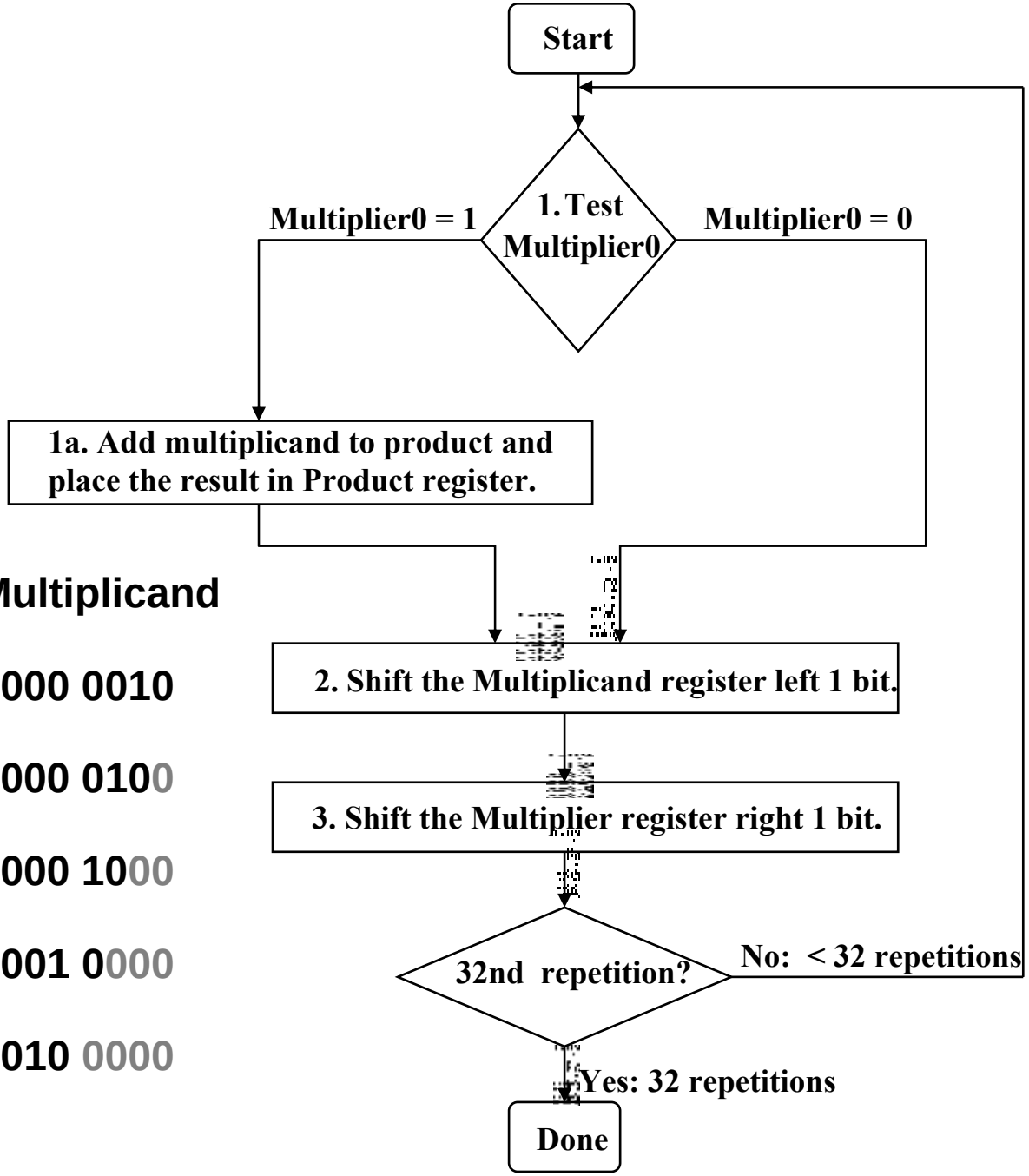
- 三个寄存器：64-bit Multiplicand, 64-bit Product, 32-bit multiplier
- 64-bit ALU



乘法算法1

2 x 3 = 6

Product	Multiplier	Multiplicand
0000 0000	0011	0000 0010
0000 0010	0001	0000 0100
0000 0110	0000	0000 1000
0000 0110	0000	0001 0000
0000 0110	0000	0010 0000

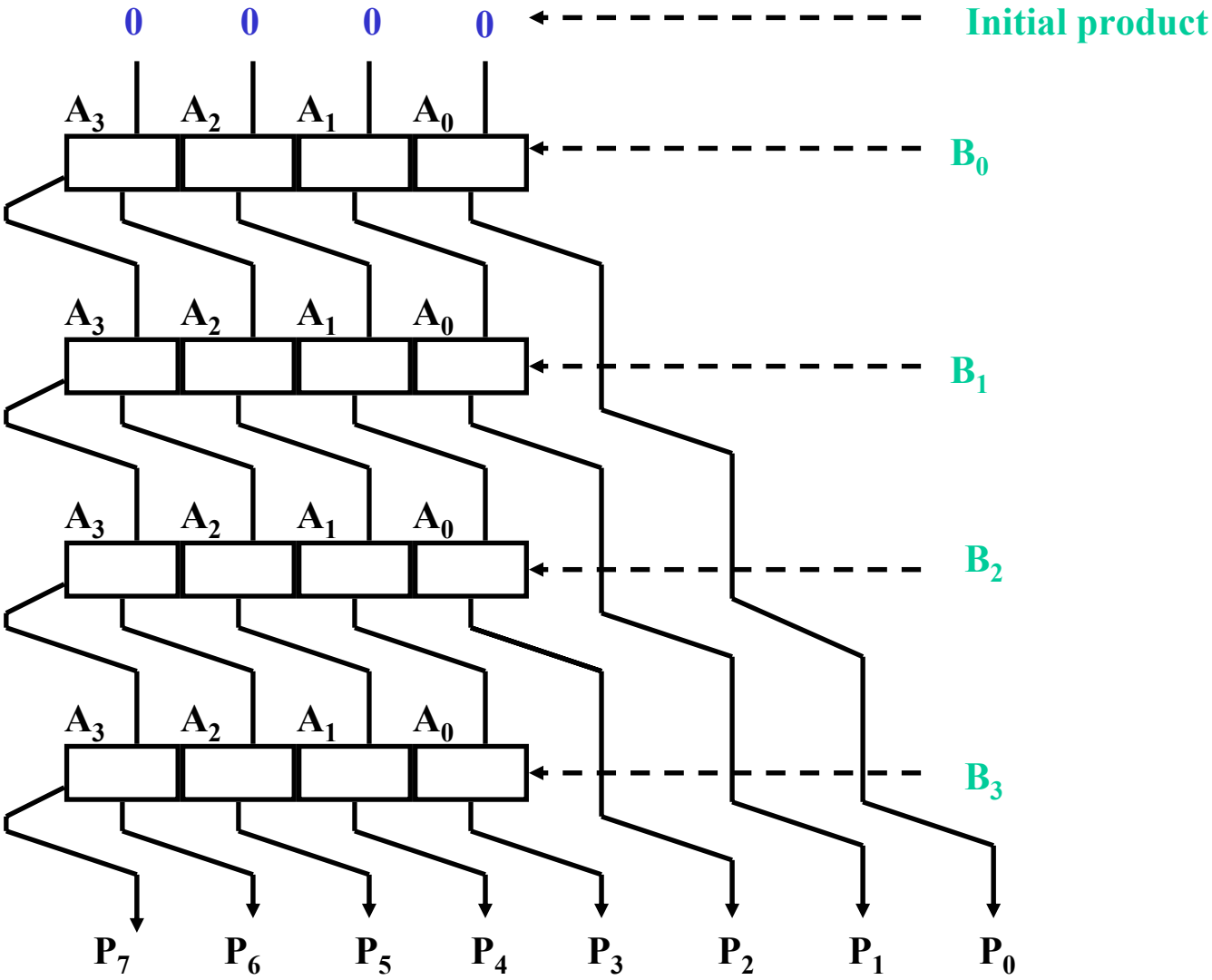


对乘法算法1的观察

- **1 clock cycle per step =>**
  - **～ 100 cycles & a 64-bits operate of each step.**
    - 程序中乘法和加法的比例： **1:5 to 1:100**
    - **Amdahl's Law:** 即使很少出现的慢速操作也会影响性能
- 被乘数的一半总是**0 => 64-bit adder is wasted**
- 被乘数左移时，右边空出位补**0 =>** 乘积后面几个最低有效位总是不变

  64位比**32**位要慢很多！！
- 能否考虑只用**32**位被乘数寄存器和**32**位**ALU?**
- 能否考虑将被乘数左移改为乘积右移而被乘数不动**?**

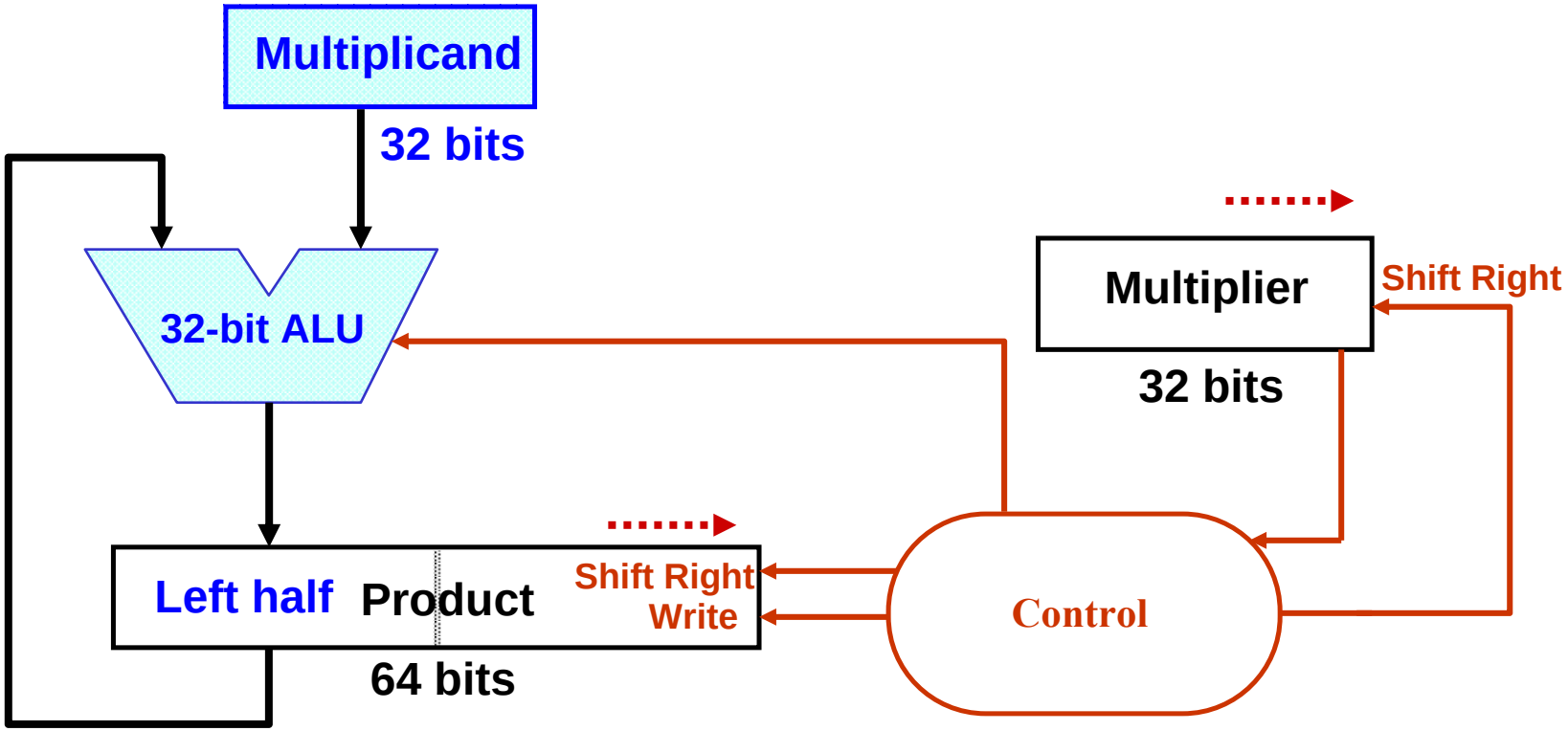
What's going on?



- Multiplicand stays still and product moves right

乘法算法2的硬件实现

- 32-bit Multiplicand reg, 64-bit Product reg, 32-bit Multiplier reg
- 32-bit ALU

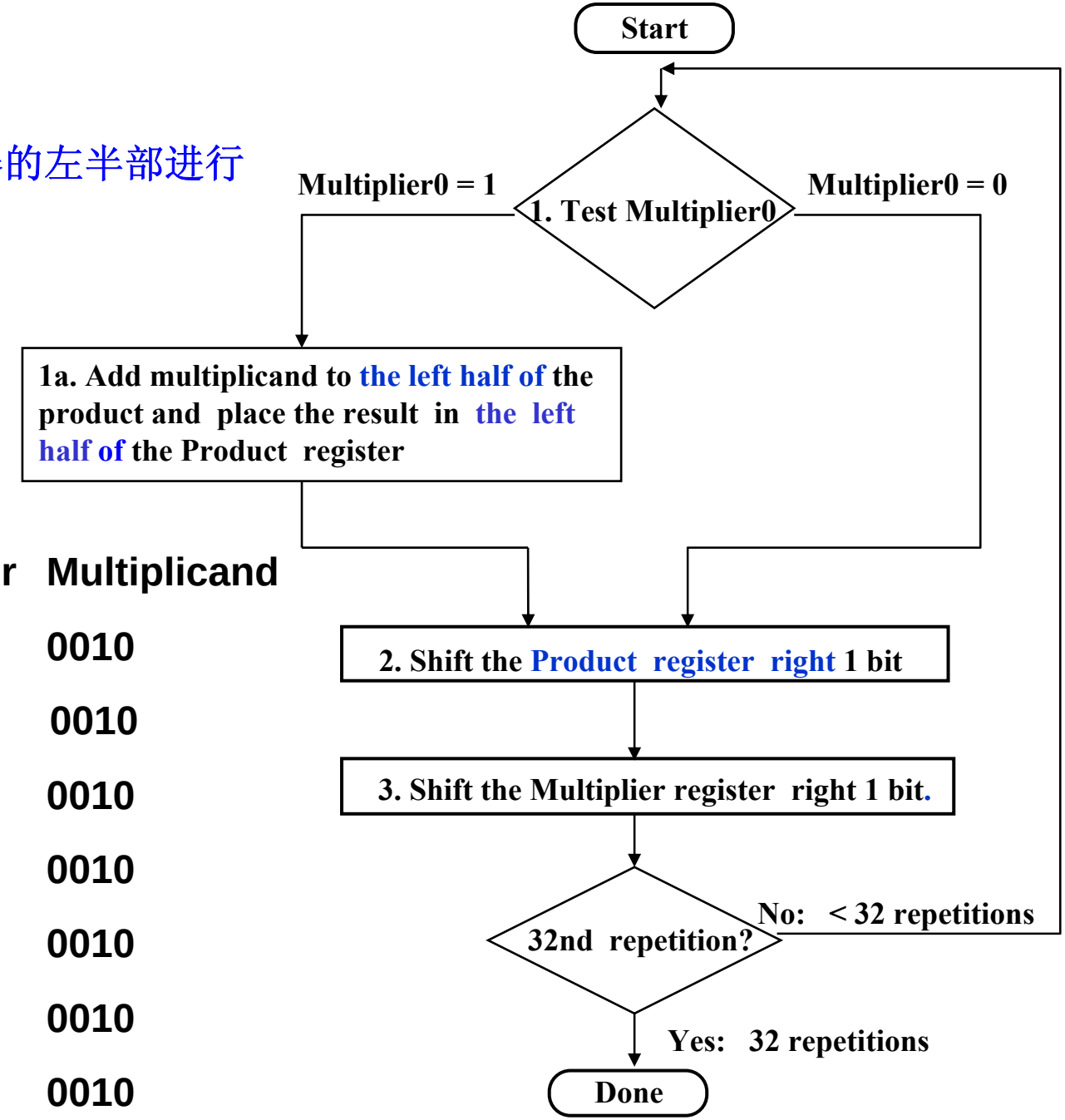


乘法算法 2

- 加法操作只在乘积寄存器的左半部进行
- 右移乘积寄存器

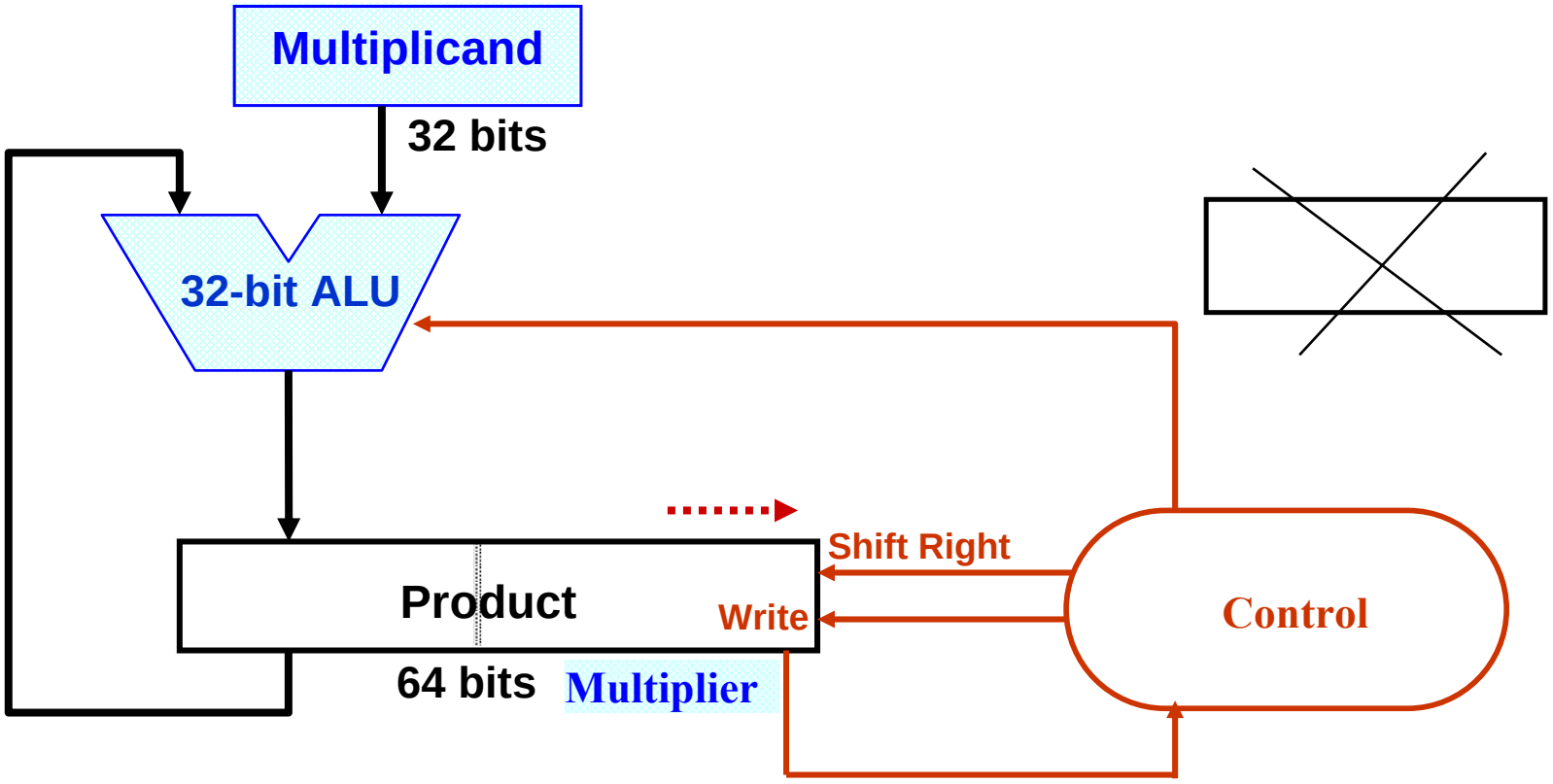
2 x 3 = 6

Product	Multiplier	Multiplicand
0000 0000	0011	0010
0010 0000	0011	0010
0001 0000	0001	0010
0011 0000	0001	0010
0001 1000	0000	0010
0000 1100	0000	0010
0000 0110	0000	0010



对乘法2硬件的改进 - 乘法3硬件实现

- 乘积寄存器有一半位数的值不变，空出来放乘数，这样省一个乘数寄存器
- **32-bit Multiplicand reg, 64-bit Product reg, 32 -bit ALU**



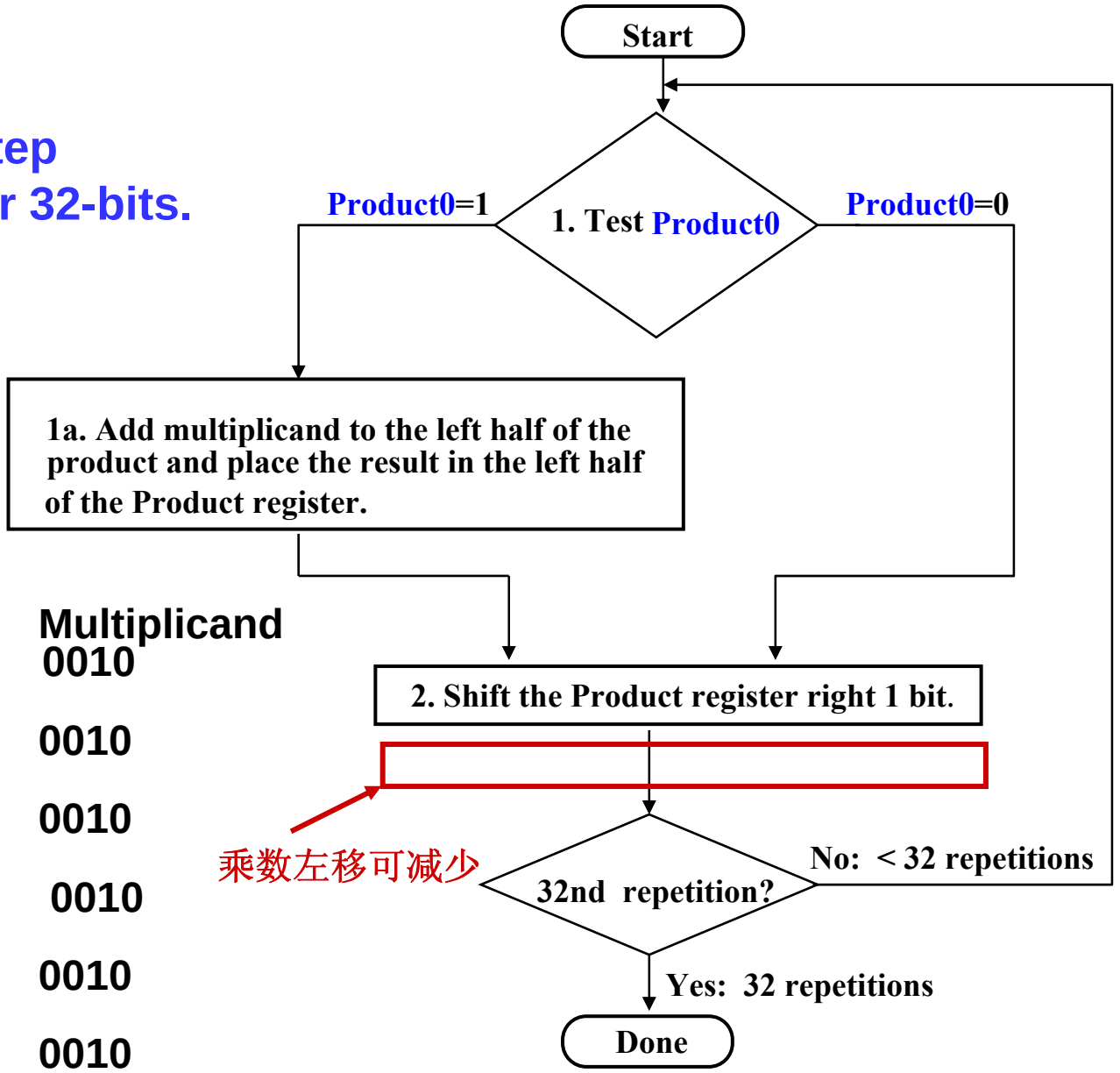
乘法算法 3

- 1 clock cycle per step
  => ～ 60 cycles for 32-bits.
- 每一步是32位运算
- 用一个ALU执行32次
- 递推公式：

$$Pi=2^{-1}(Abi+ Pi-1)$$

2 x 3 = 6

	Product | Multiplier	Multiplicand
◦	0000 0011	0010
◦	0010 0011	0010
◦	0001 0001	0010
◦	0011 0000	0010
◦	0001 1000	0010
◦	0000 1100	0010
◦	0000 0110	0010



乘数左移可减少

快速乘法器可以用32个Adder同时进行加法，并错开一位，得到最终乘积。

Example：无符号整数乘法运算

举例说明：

设A=1110　B=1101　应用递推公式：$P_i=2^{-1}(Ab_i+P_{i-1})$

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

✓可以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。

✓左移时进位、部分积和剩余乘数一起进行逻辑右移。

验证：**A=14, B=13, AB=182**

带符号数乘法

◆定点有符号数乘法

- 原码乘法
  - 将符号与数值分开处理
  - 积符用两个乘数的符号异或得到
  - 数值部分用无符号乘法运算
- 补码乘法

方法1：两数都变成正数，用无符号乘法运算，最后根据两个乘数是否异号确定是否对结果取负。

方法2：用Booth乘法，其速度更快。

(符号位和数值位可一起参加运算)

Booth's Algorithm推导

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

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

求：$[A\,B]_{补}=?$

基于以下补码性质：

大家记得"补码的值"的公式吗？

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

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

假设：　a-1 =0，则：

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

↓

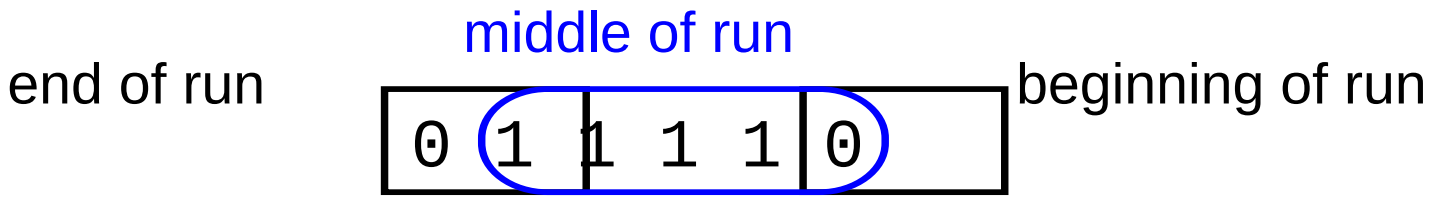
$-a_{31}\cdot 2^{31}+(a_{30}\cdot 2^{31}-a_{30}\cdot 2^{30})+\cdots\cdots+(a_0\cdot 2^1-a_0\cdot 2^0)+a_{-1}\cdot 2^0$

↓

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

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

Booth's Algorithm Insight



Current Bit	Bit to the Right	Operate	Example
1	0	减被乘数	000111**10**00
1	1	加0(不操作)	00011**11**000
0	1	加被乘数	00**01**111000
0	0	加0(不操作)	0**00**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。

同前面的算法一样，将乘积寄存器右移一位。（这里是算术右移）

例：

Multiplicand	Product (2 x 7)
0010	0000 0111 0

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

Booths Example: 2 x 7

mythical bit

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

最后乘积

Booths Example: 2 x –3

mythical bit

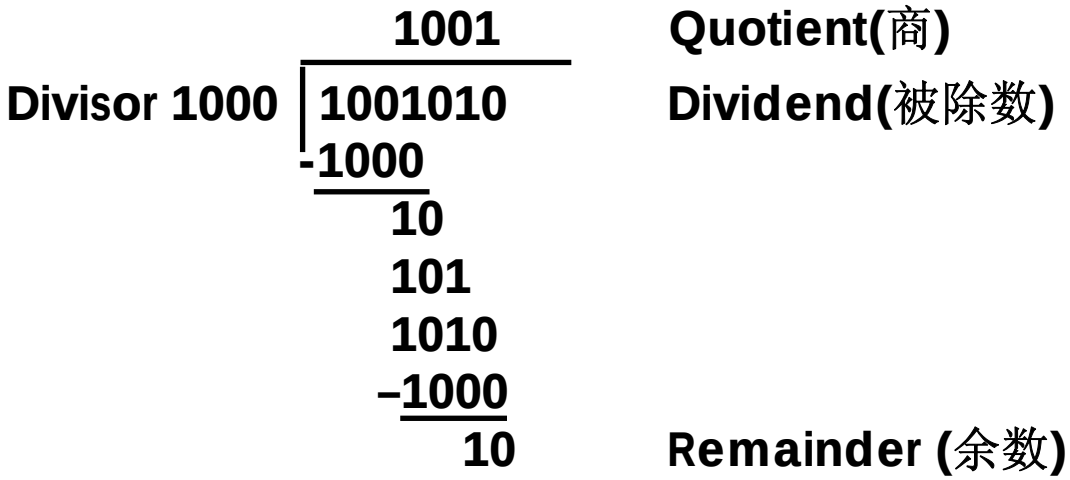
Operation	Multiplicand	Product	next?
0. initial value	0010	0000 1101 0	10 -> sub
1a. P = P - m	1110	+ 1110	
		1110 1101 0	shift P (sign ext)
1b.	0010	1111 0110 1	01 -> add
		+ 0010	
2a.		0001 0110 1	shift P
2b.	0010	0000 1011 0	10 -> sub
		+ 1110	
3a.	0010	1110 1011 0	shift
3b.	0010	1111 0101 1	11 ->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

最后乘积

MIPS中的乘法运算处理

- MIPS中有一对32-bit寄存器Hi & Lo用来存放64-bit乘积
  （有些机器中把这种寄存器称为Q乘商寄存器）
- mflo指令用来把Lo中的32位乘积取到通用寄存器
- Signed & Unsigned multiply指令: mult & multu
- 两种指令mult and multu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。
  - 全0 (无符号数乘multu)
  - LO中的符号 (带符号数乘mult)

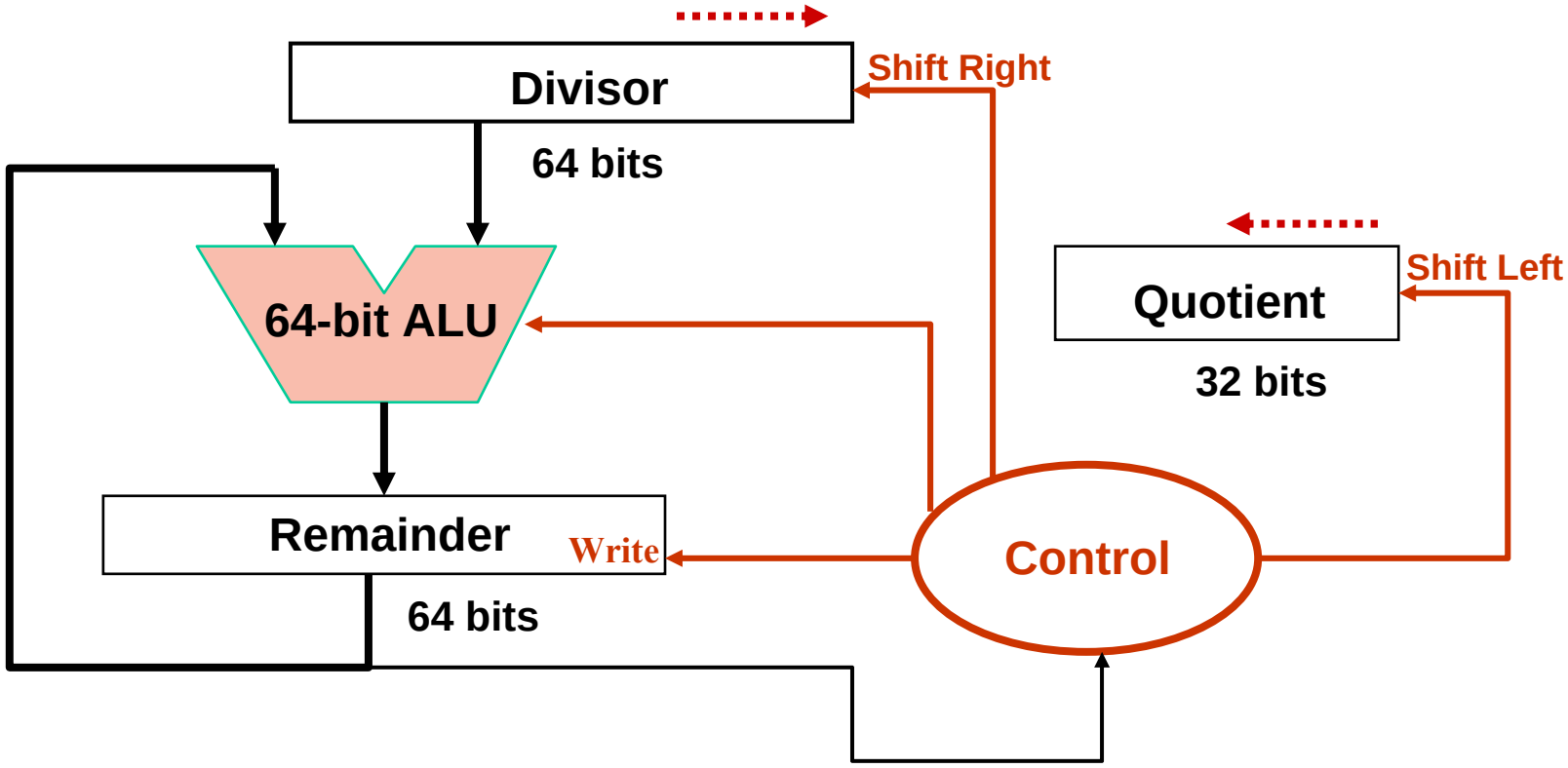
Divide: Paper & Pencil



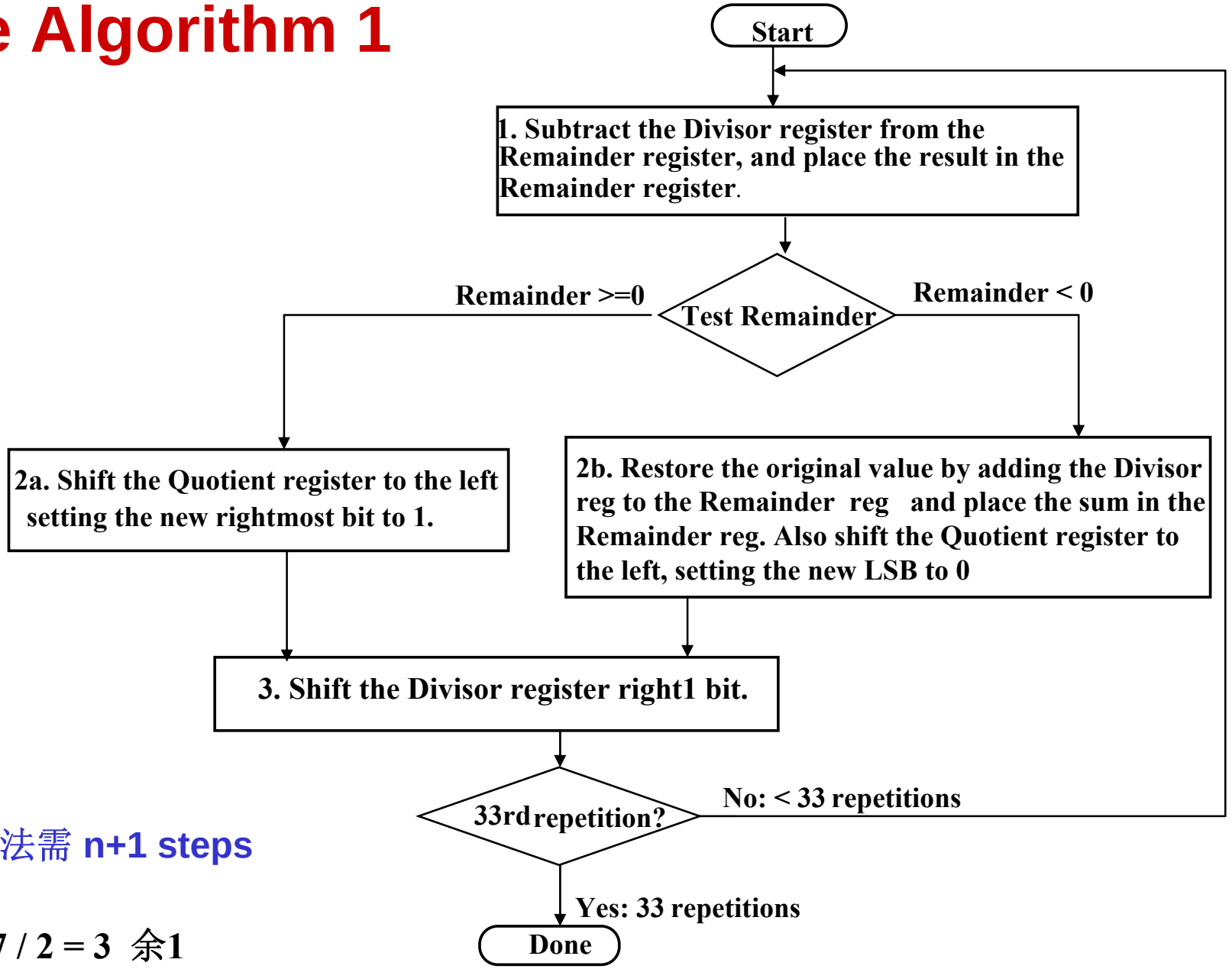
- 基本思想：通过减除数进行试商，够减上商1；不够减上商0
- **Dividend = Quotient x Divisor + Remainder**
- 基本操作为减法（用加法来实现）和移位，所以其硬件同乘法
- 3种除法方法：逐步求精的过程（从模拟笔算方式开始逐步简化）

除法算法1的硬件实现

- 三个寄存器：64-bit Divisor, 64-bit Remainder ,32-bit Quotient
- 64-bit ALU



Divide Algorithm 1



◆n位除法需 n+1 steps

举例：7 / 2 = 3 余1

Divide Algorithm Version 1--example

	Q: 0000	**D: 0010 0000**	**R: 0000 0111 –D = 1110 0000**
1: R = R–D	Q: 0000	D: 0010 0000	R: 1110 0111
2b: +D, sl Q, 0	Q: 0000	D: 0010 0000	R: 0000 0111
3: Shr D	Q: 0000	D: 0001 0000	R: 0000 0111 **–D = 1111 0000**
1: R = R–D	Q: 0000	D: 0001 0000	R: 1111 0111
2b: +D, sl Q, 0	Q: 0000	D: 0001 0000	R: 0000 0111
3: Shr D	Q: 0000	D: 0000 1000	R: 0000 0111 **–D = 1111 1000**
1: R = R–D	Q: 0000	D: 0000 1000	R: 1111 1111
2b: +D, sl Q, 0	Q: 0000	D: 0000 1000	R: 0000 0111
3: Shr D	Q: 0000	D: 0000 0100	R: 0000 0111 **–D = 1111 1100**
1: R = R–D	Q: 0000	D: 0000 0100	R: 0000 0011
2a: sl Q, 1	Q: 0001	D: 0000 0100	R: 0000 0011
3: Shr D	Q: 0001	D: 0000 0010	R: 0000 0011 **–D = 1111 1110**
1: R = R–D	Q: 0001	D: 0000 0010	R: 0000 0001
2a: sl Q, 1	Q: 0011	D: 0000 0010	R: 0000 0001
3: Shr D	Q: 0011	D: 0000 0001	R: 0000 0001

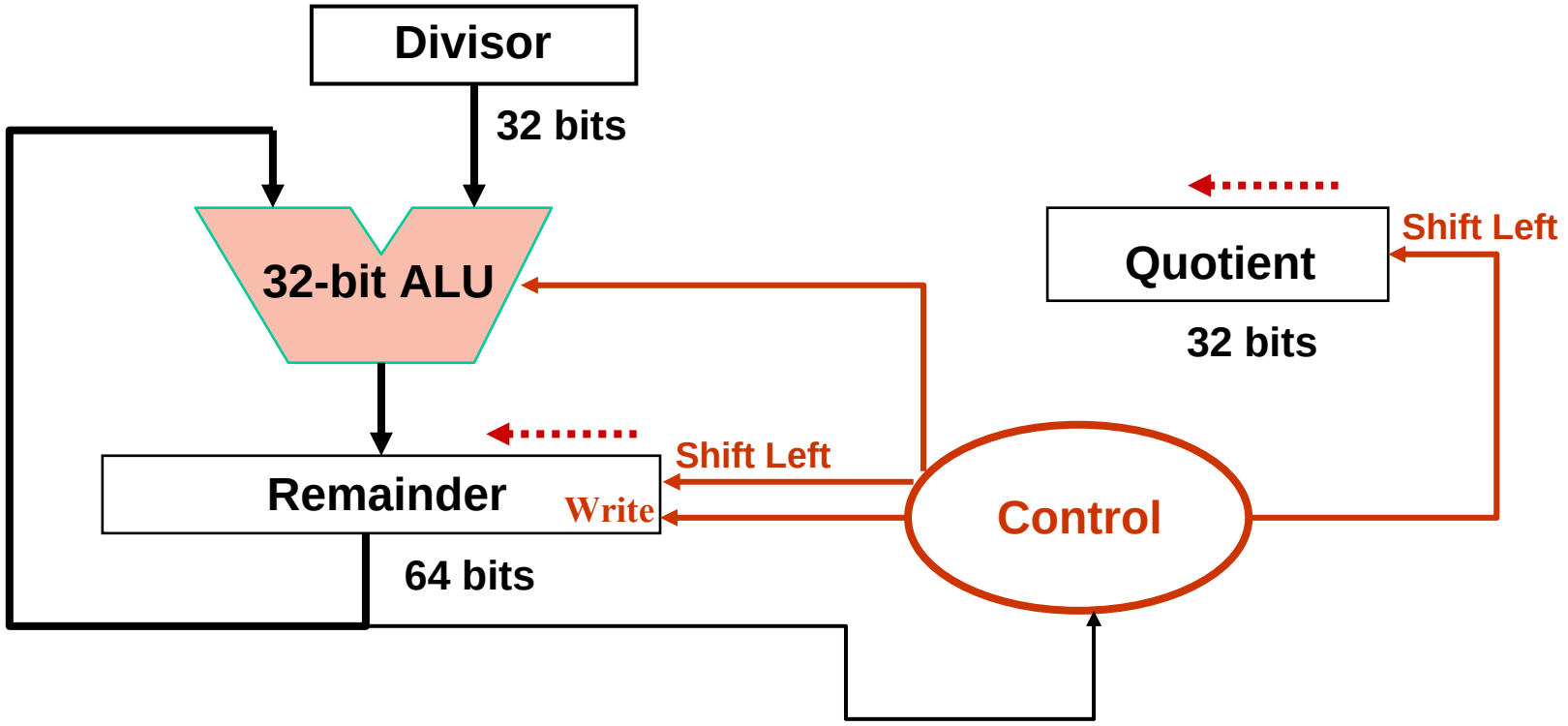
验证：7 / 2 =3 余 1

对除法算法 1的观察

- **1/2 bits in divisor always 0**
  => 1/2 of 64-bit adder is wasted
  => 1/2 of divisor is wasted
- 能否考虑只用32位除数寄存器和32位ALU?
- 可用余数左移代替除数右移
- 第一步试商是用来判断是否“溢出”的。若第一次上商为1，则会产生n+1位商，因而就产生“溢出”。程序中一般先判断是否溢出，只有在保证不溢出的情况下，才继续做除法。因此，除法过程的第一步不可能产生商1，因而可将第一步换成先移位再减，这样，可以减少一次迭代

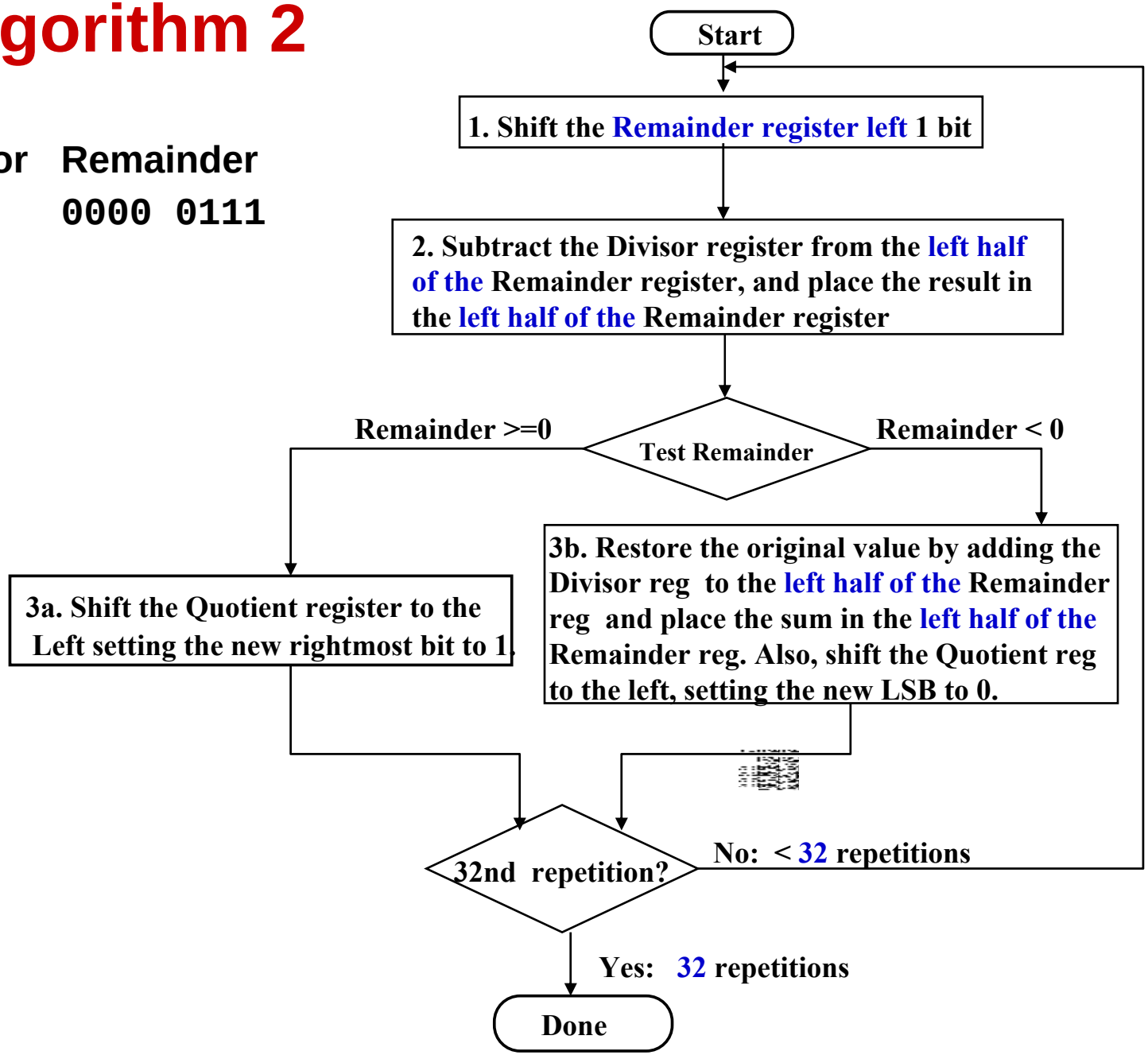
除法算法2的硬件实现

- reg: 32-bit Divisor, 64-bit Remainder, 32-bit Quotient
- 32 -bit ALU



Divide Algorithm 2

Quotient	Divisor	Remainder
0000	0010	0000 0111



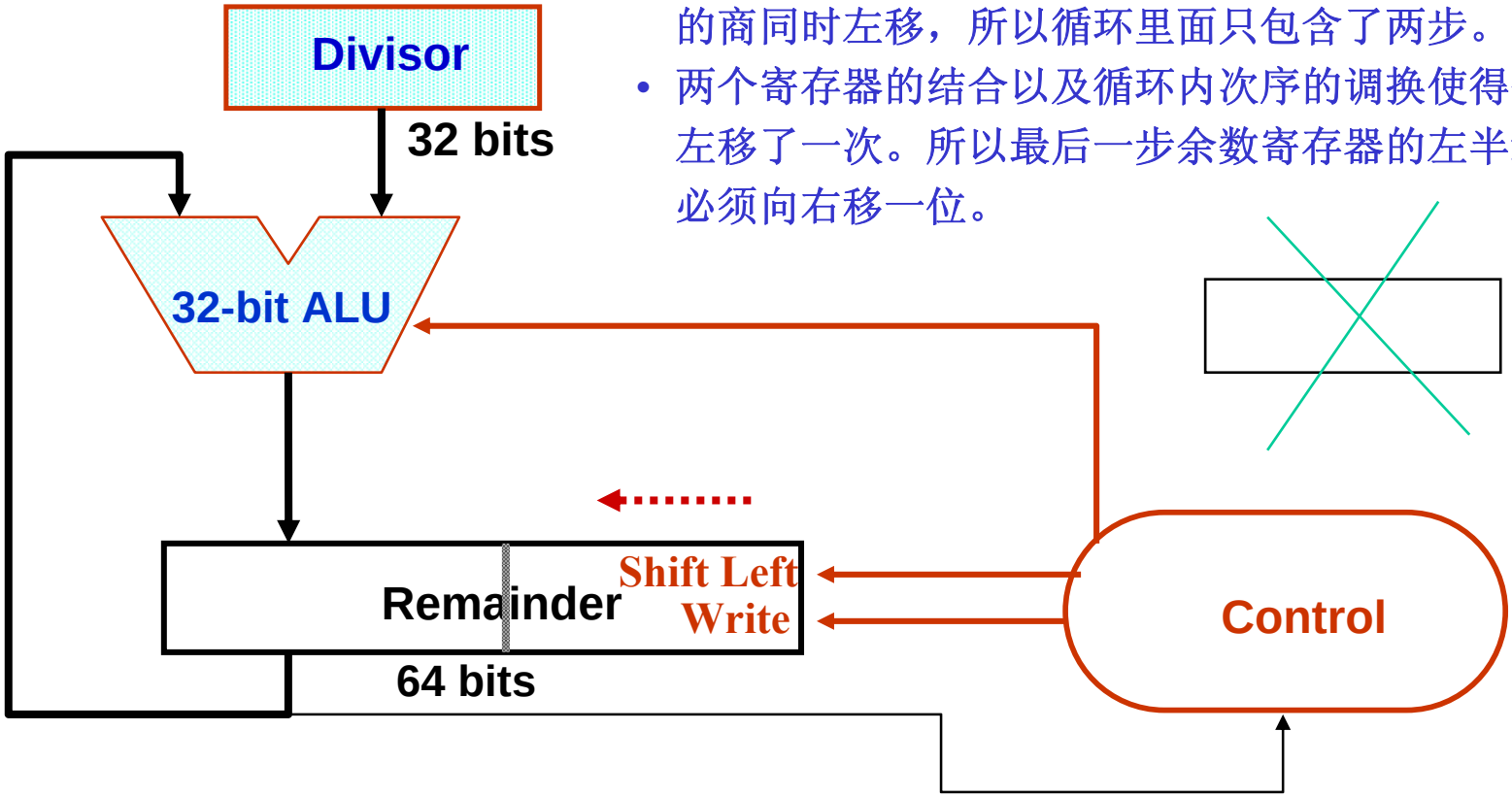
Divide Algorithm Version 2--example

	Q: 0000	D: 0010	R: 0000 0111
1: Shl R	Q: 0000	D: 0010	R: 0000 1110
2: R = R–D	Q: 0000	D: 0010	R: 1110 1110
3b: +D, sl Q, 0	Q: 0000	D: 0010	R: 0000 1110
1: Shl R	Q: 0000	D: 0010	R: 0001 1100
2: R = R–D	Q: 0000	D: 0010	R: 1111 1100
3b: +D, sl Q, 0	Q: 0000	D: 0010	R: 0001 1100
1: Shl R	Q: 0000	D: 0010	R: 0011 1000
2: R = R–D	Q: 0000	D: 0010	R: 0001 1000
3a: sl Q, 1	Q: 0001	D: 0010	R: 0001 1000
1: Shl R	Q: 0001	D: 0010	R: 0011 0000
2: R = R–D	Q: 0001	D: 0010	R: 0001 0000
3a: sl Q, 1	Q: 0011	D: 0010	R: 0001 0000

验证：**7 / 2 =3 余 1**

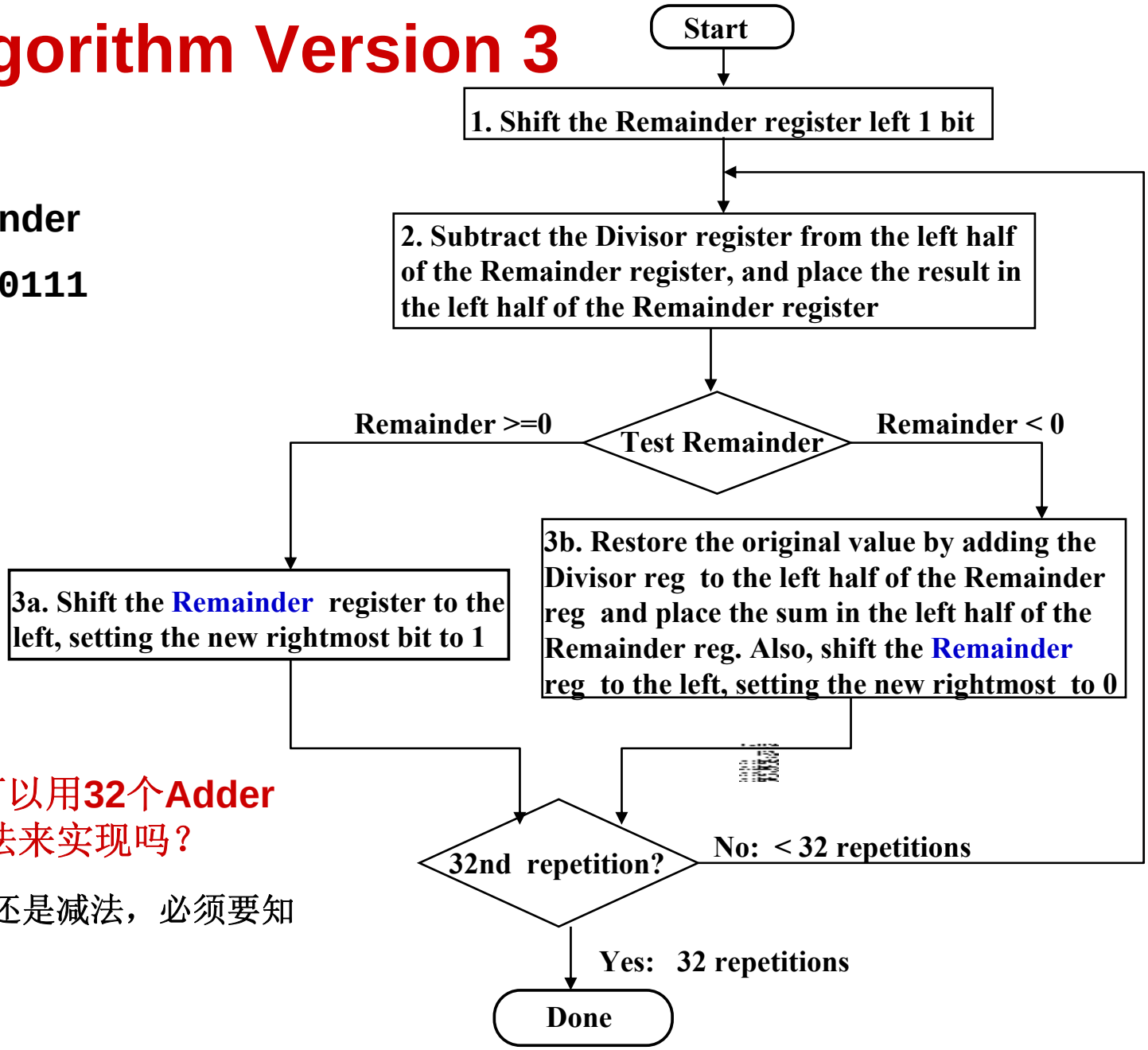
除法算法3的硬件实现

- 32-bit Divisor reg, 64-bit Remainder reg
- 32-bit ALU
- 可以使商通过和余数一起左移来取消商寄存器
  - 开始时先使余数左移一位
  - 因为余数寄存器的左移实际上使左半部的余数和右半部的商同时左移，所以循环里面只包含了两步。
  - 两个寄存器的结合以及循环内次序的调换使得余数被多左移了一次。所以最后一步余数寄存器的左半边的余数必须向右移一位。



Divide Algorithm Version 3

Divisor	Remainder
0010	0000 0111



快速除法器：可以用32个Adder同时进行加/减法来实现吗？

不行！每次做加法还是减法，必须要知道上次余数的符号。

Divide Algorithm Version 3--example

	D: 0010	R: 0000 0111
1: Shl R	D: 0010	R: 0000 1110
2: R = R–D	D: 0010	R: 1110 1110
3b: +D, sl R, 0	D: 0010	R: 0001 1100
2: R = R–D	D: 0010	R: 1111 1100
3b: +D, sl R, 0	D: 0010	R: 0011 1000
2: R = R–D	D: 0010	R: 0001 1000
3a: sl R, 1	D: 0010	R: 0011 0001
2: R = R–D	D: 0010	R: 0001 0001
3a: sl R, 1	D: 0010	R: 0010 0011
Shr R(rh)	D: 0010	R: 0001 0011

验证：7 / 2 =3 余 1

MIPS中的除法运算处理

- 与乘法运算的硬件相同：
  - 仅需做加减和63位寄存器的左/右移位。
  - Hi和Lo结合起来实现64位寄存器
- 有符号数除法最简便方法是转换为正数,然后必要时再对商和余数求补。（具体调整方法见书中3.5.2中说明）
  - 被除数和余数一定同号！
  - 如果被除数和除数符号不同，则商为负。
- **Instruction: Signed (div); unsigned (divu)**
- **Hi中存放remainder， Lo中存放 quotient**
- **MIPS指令不处理“溢出”和“除数为0”，由软件自行处理**

不恢复余数除法(加减交替法)

前面给出的除法算法要恢复余数，可以更进一步简化为“加减交替法”

根据恢复余数法(设B为除数，R_i为第i次中间余数)，有：

- 若$R_i<0$,则商上“ 0 ”，做加法恢复余数，即：

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

  (由上式可知：“负， 0 ，加”)

- 若$R_i>0$,则商上“1”，不需恢复余数，即：

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

  (由上式可知：“正， 1 ，减”)

省去了恢复余数的过程

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

不恢复余数法也称为加减交替法

设 $[x]_{原}$=1 0010 0110　$[y]_{原}$=0 0111　求 $[x/y]_{原}$,$[R]_{原}$

解：$[-|y|]_{补}$=1001　|x|=0010 0110

A	Q	上商	说明
0010	0110		
1001			试商，做减法
1011	0110	0	不够减，商0
0111			做加法，恢复余数
0010	0110		
0100	1100		左移一位
1001			试商，做减法
1101	1100	0	不够减，商0
0111			做加法，恢复余数
0100	1100		
1001	1000		左移一位
1001			试商，做减法
0010	1000	1	够减，商1
0101	0001		左移一位
1001			试商，做减法
1110	0001	0	不够减，商0
0111			做加法，恢复余数
0101	0001		
1010	0010		左移一位
1001			试商，做减法
0011	0010	1	够减，商1

例1：恢复余数法

一般第一次不判断溢出，即：不试商而直接左移，这样只要n次循环。
是否溢出由软件来判断

$[x/y]_{原}$=1 0101

$[R]_{原}$=1 0011

x/y=-5

余数R=-3

验证：

-38=(-5)x(+7)+(-3)

不恢复余数法

设 $[x]_{原}$=1 0010 0110　$[y]_{原}$=0 0111　求 $[x/y]_{原}$,$[R]_{原}$

解：$[-|y|]_{补}$=1001　|x|=0010 0110

A	Q	上商	说明
0010	0110		
1001			试商，做减法
1011	0110	0	负，0，加
0110	1100		左移一位
0111			做加法
1101	1100	0	负，0，加
1011	1000		左移一位
0111			做加法
0010	1000	1	正，1，减
0101	0001		左移一位
1001			做减法
1110	0001	0	负，0，加
1100	0010		左移一位
0111			做加法
0011	0010	1	正，1，减 (最后一步)

第一次总是做减法，上的商不是真正的商，只是用来判断是否溢出，故最终的商应是Q和Q_{-1}一起左移一位后在Q中的数。即：商的数值为0101。

运算结果同前面的恢复余数法

最后一次上商为“1”，故不需恢复余数。

带符号数除法

- 有符号数的除法
  - ✓原码除法
    - o 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 。
    - o 余数的符号同被除数的符号。
  - ✓补码除法
    - o 方法1：先转换为正数，用无符号数除法，然后修正商和余数。（参看书中3.5.2节内容说明）
    - o 方法2：直接用补码除法。（请参考相关教材）

补码定点除法算法1

算法要点：

商：

采用后面表中给定的上商规则，得到商的反码表示。

所以，要得到最终的补码表示，必须按下列规则进行补正：

“若结果符号为0，则不补正；若为1，则须在最末位加1”

余数：

最后要按下列规则进行补正：

“若余数符号同被除数符号，则不需补正；

若余数符号与被除数符号不同，则

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。”

(参看 徐福培等《计算机组成与结构》)

补码定点除法算法1

余数A	除数M	A-M		A+M	
		0	1	0	1
0	0	够减(商1)	不够减(商0)	两数同号，商为正	
1	1	不够减(商0)	够减(商1)		
0	1	两数异号，商为负 商取反码表示		够减(商0)	不够减(商1)
1	0			不够减(商1)	够减(商0)

- 根据不同的上商原则可得到不同的运算方法

上述上商规则，得到商的反码表示。
若采用“够减商1，不够减商0”的规则，则得到商的绝对值表示。
下列方法二即是。

补码定点除法算法2

算法要点

1.操作数的预置：除数装入M寄存器，被除数装入A,Q寄存器，被除数须以2n位补码形式表示（不足时补上高n位进行符号扩展，如：1001->11111001,0101->00000101）。

2.A,Q左移一位。

3.若A与M同号，则计算：A=A-M；否则，计算：A=A+M;

根据计算结果，按以下规则确定商值Q_0 ：

若A和Q中的中间余数=0或A操作前后符号未变,则Q_0置1,转下一步；

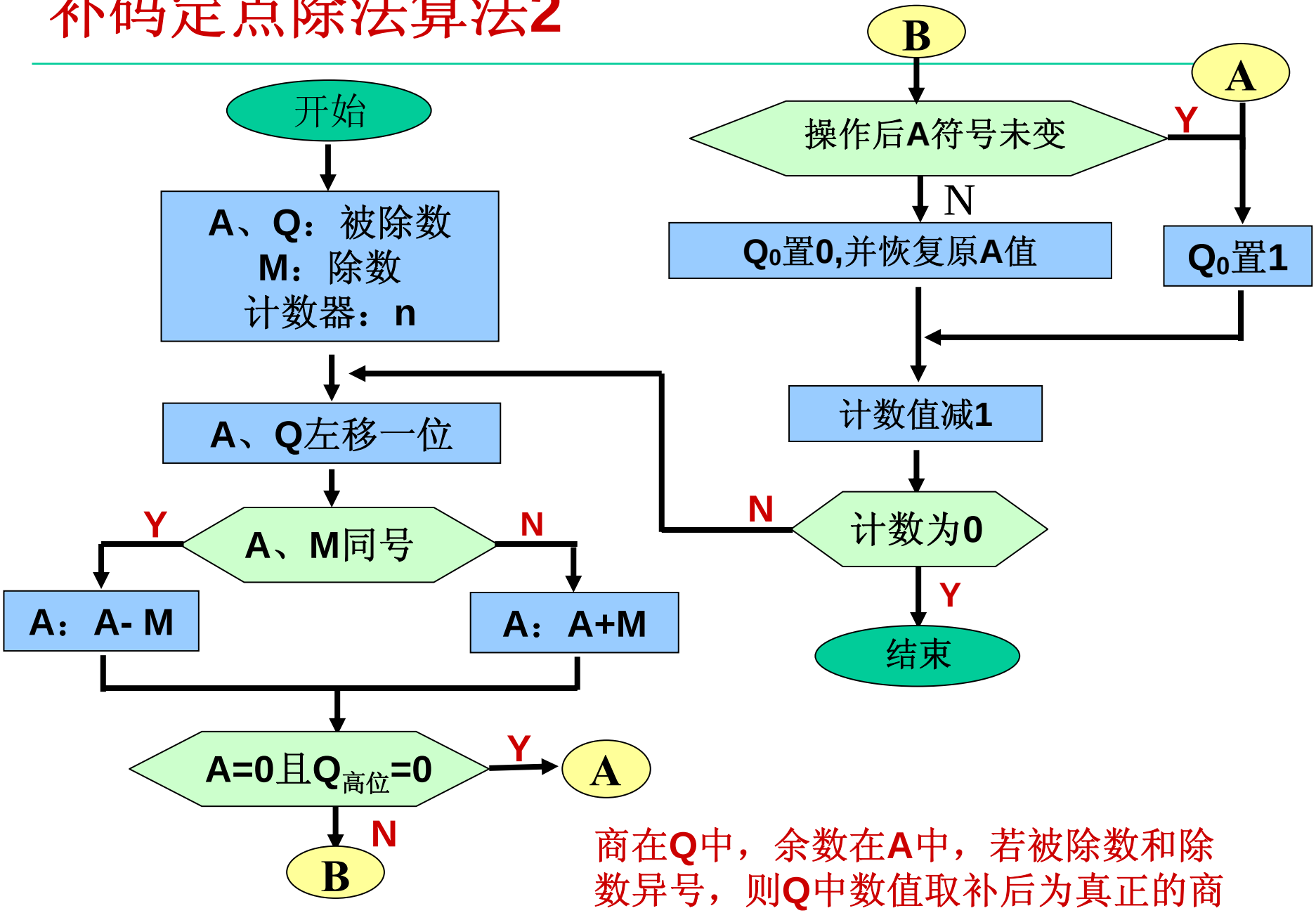
若A操作前后符号已变,则Q_0置0,恢复A值，转下一步；

4.重复2-3步，直到取得n位商为止。

5.余数在A中。若被除数与除数同号，则商在Q中；否则，Q中数值求补（Q取负）后是真正的商。

参看 《COMPUTER ORGANIZATION AND ARCHITECTURE Design for Performance》William Stallings

补码定点除法算法2



商在Q中，余数在A中，若被除数和除数异号，则Q中数值取补后为真正的商

举例：7/3=?　(-7)/3=?

被除数：0000 0111　除数 0011

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

余:0001/商:0010

被除数：1111 1001　除数 0011

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

余:1111/商:1110

第一讲总结

- 定点整数的二进制表示
  - 无符号数：正整数，用来表示地址等；带符号数：用补码表示
- 定点数的运算：**ALU**实现基本算术和逻辑运算，**ALU+**移位器实现其他运算
- 移位运算
  - 逻辑移位 、算术移位、循环移位
- 扩展运算
  - 零扩展 / 符号扩展
- 加/减运算
  - 补码加/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：符号位和数值位分开运算，同号相加，异号相减，大数减小数，结果取大数的符号。减法用加负数补码实现。（用于浮点数尾数加/减运算。自学 ）
- 乘法运算：用加法和右移实现。
  - 补码乘法：符号位和数值位一起运算。采用**Booth**算法。
  - 原码乘法：符号位和数值位分开运算。数值部分用无符号数乘法实现。（用于浮点数尾数乘法运算。）
- 除法运算：用加/减法和左移实现。
  - 补码除法：符号位和数值位一起运算。
  - 原码除法：符号位和数值位分开运算。数值部分用无符号数除法实现。（用于浮点数尾数除法运算。）

第二讲：浮点数的表示和运算

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
- 浮点数的表示（IEEE754标准）
- 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中的浮点算术运算指令

FP add single	add.s $f2,$f4,$f6	$f2 = $f4 + $f6
FP subtract single	sub.s $f2,$f4,$f6	$f2 = $f4 - $f6
FP multiply single	mul.s $f2,$f4,$f6	$f2 = $f4 × $f6
FP divide single	div.s $f2,$f4,$f6	$f2 = $f4 / $f6
FP add double	add.d $f2,$f4,$f6	$f2 = $f4 + $f6
FP subtract double	sub.d $f2,$f4,$f6	$f2 = $f4 - $f6
FP multiply double	mul.d $f2,$f4,$f6	$f2 = $f4 × $f6
FP divide double	div.d $f2,$f4,$f6	$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	lwc1 $f1,100($s2)	$f1 = Memory[$s2 + 100]
store word copr. 1	swc1 $f1,100($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，相加后再存回到内存。

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

MIPS中的浮点数比较和分支指令

branch on FP true	bc1t 25	if (cond == 1) go to PC + 4 + 100
branch on FP false	bc1f 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

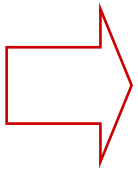
MIPS浮点运算指令的总结

- 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

例子：将以下程序编译为**MIPS**汇编语言

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

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



假设不考虑信息在栈帧中的保存和恢复

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

有关Floating-point number的问题

实现一套浮点数运算指令，要解决的问题有：

Issues:

- **Representation(表示)：**

  Normalized form (规格化形式) 和 Denormalized form

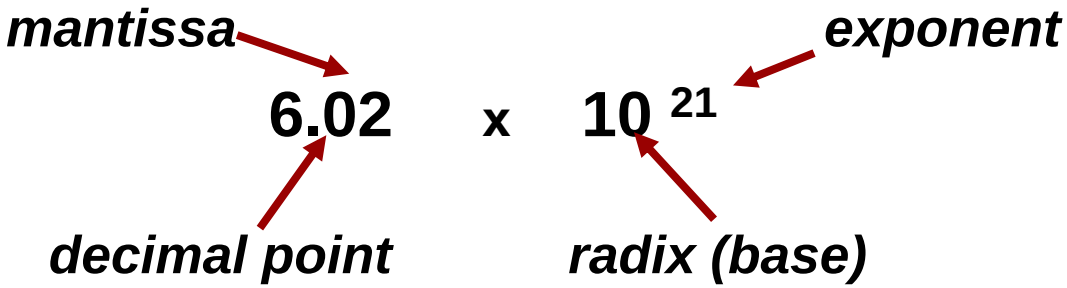
  单精度格式 和 双精度格式

- **Range and Precision(表数范围和精度)**
- **Arithmetic (+, -, *, /)**
- **Rounding(舍入)**
- **Exceptions (e.g., divide by zero, overflow, underflow)**

  (异常处理：如除数为0，上溢，下溢等)
- **Errors(误差)与精度控制**

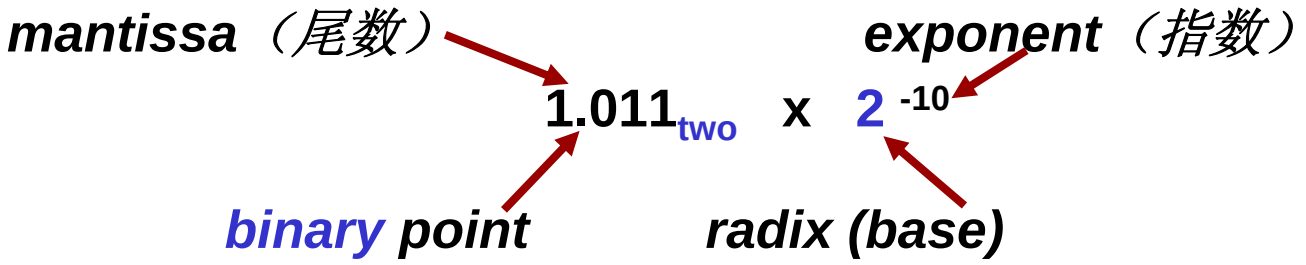
科学计数法与浮点数

Example:



- **Normalized form**（规格化形式）：小数点前只有一位非0数
- 同一个数有多种表示形式。例：对于数 **1/1,000,000,000**
  - **Normalized (唯一的规格化形式): 1.0×10^{-9}**
  - **Unnormalized（非规格化形式不唯一）: 0.1×10^{-8}, 10.0×10^{-10}**

Like Scientific Notation for Binary Numbers:



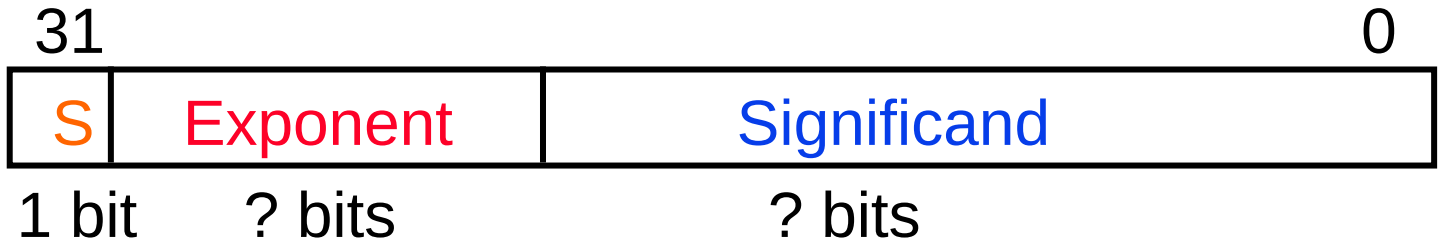
只要对尾数和指数分别用二进制编码，就可以表示一个浮点数（即：实数）

Floating Point （浮点数的表示）

- Normal format:

  $+/-1.xxxxxxxxxx_{two} \times 2^{Exponent}$

- 32-bit version



S represents Sign

Exponent用 excess (or biased) notation(移码/增码)来表示

Significand represents x's

(The base could be 2/ 4 / 8 / 16)

- Until about 1980, each manufacturer used different format => not compatible, 机器之间数据传送时，会带来麻烦

SKIP

Excess (biased) notion- 移码表示

- 什么是“excess (biased) notation-移码表示”？

  将每一个数值加上一个偏置常数（ Excess / bias）

- 一般来说，当编码位数为 n时， bias取 2^{n-1}

  Ex. n=4: $E_{biased} = E + 2^3$ （bias= $2^3 = 1000_2$ ）

  $-8\ (+8) \sim 0000_2$

  $-7\ (+8) \sim 0001_2$

  …

  $0\ \ (+8) \sim 1000_2$

  …

  $+7\ (+8) \sim 1111_2$

- 为什么要用移码来表示指数（阶码）？

  便于浮点数加减运算时的对阶操作

例：$1.01 \times 2^{-1} + 1.11 \times 2^3$

补码：1111< 0011 ?
(-1) (3)

简化比较

$1.01 \times 2^{-1+4} + 1.11 \times 2^{3+4}$

移码：0011< 0111
(3) (7)

Back to last

“Father” of the IEEE 754 standard

1970年代后期, **IEEE**成立委员会着手制定浮点数标准

1985年完成浮点数标准**IEEE754**的制定

现在所有计算机都采用**IEEE754**来表示浮点数

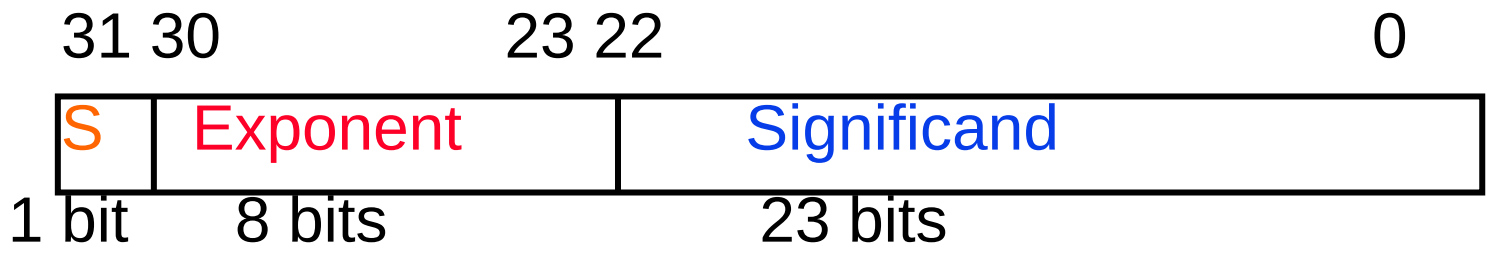
This standard was primarily the work of one person, UC Berkeley math professor William Kahan.

www.cs.berkeley.edu/~wkahan/ieee754status/754story.html

Prof. William Kahan

IEEE 754 Floating Point Standard

Single Precision： (Double Precision is similar)



- **Sign bit: 1 表示negative ; 0表示 positive**
- **Exponent（阶码 / 指数）：**

  全0和全1编码要用来表示特殊的值！

  - **SP规格化数阶码范围为0000 0001(-126) ~ 1111 1110(127)**
  - **bias为127 (single), 1023(double)** （为什么用**127**？）
- **Significand（尾数）：**
  - 规格化尾数最高位总是**1**，所以隐含表示，省**1**位
  - **1 + 23 bits （ single），1 + 52 bits （double）**

SP: $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

DP: $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-1023)}$

Ex: Converting Binary FP to Decimal

BEE00000H is the hex. Rep. Of an IEEE 754 SP FP number

1	011 1110	1110 0000 0000 0000 0000 0000

$$(-1)^{S} \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$$

- **Sign: 1 => negative**
- **Exponent:**
  - **$0111\ 1101_{two} = 125_{ten}$**
  - **Bias adjustment: 125 - 127 = -2**
- **Significand:**

  $1 + 1x2^{-1} + 1x2^{-2} + 0x2^{-3} + 0x2^{-4} + 0x2^{-5} + \ldots$

  $= 1 + 2^{-1} + 2^{-2} = 1 + 0.5 + 0.25 = 1.75$
- **Represents: $-1.75_{ten} x 2^{-2} = -0.4375$ $(= -4.375x10^{-1})$**

Ex: Converting Decimal to FP

-1.275×10^1

1. Denormalize: -12. 75
2. Convert integer part:

   $12 = 8 + 4 = 1100_2$
3. Convert fractional part:

   $.75 = .5 + .25 = .11_2$
4. Put parts together and normalize:

   $1100.11 = 1.10011 \times 2^3$
5. Convert exponent: $127 + 3 = 128 + 2 = 10000010_2$

1	100 0001 0	100 1100 0000 0000 0000 0000

The Hex rep. is C14C0000H

Normalized numbers（规格化数）

前面的定义都是针对规格化数（**normalized form**）

How about other patterns?

Exponent	Significand	Object
1-254	anything implicit leading 1	Norms
0	0	?
0	nonzero	?
255	0	?
255	nonzero	?

Representation for 0

How to represent 0?

exponent: all zeros

significand: all zeros

What about sign? Both cases valid.

+0: 0 00000000 00000000000000000000000

-0: 1 00000000 00000000000000000000000

Representation for +∞/-∞

- In FP, 除数为0的结果是 +/- infinity, 不是overflow.
- 为什么要这样处理?
  - 可以利用+∞/-∞作比较。 例如：X/0 > Y 可作为一个有效比较
- How to represent +∞/-∞?
  - Exponent : all ones (11111111B=255)
  - Significand: all zeros

  +∞ : 0 11111111 00000000000000000000000

  -∞ : 1 11111111 00000000000000000000000
- Operations

  5 / 0 = +∞, -5 / 0 = -∞

  5+(+∞)=+∞, (+∞)+(+∞)=+∞

  5 - (+∞)=-∞, (-∞) - (+∞)= -∞ etc

Representation for “Not a Number”

sqrt(-4.0)=? 0/0=?

- **Called Not a Number (NaN) - “非数”**

- **How to represent NaN**

  Exponent = 255

  Significand: nonzero

  NaNs can help with debugging

- **Operations**

  sqrt(-4.0)=NaN 0/0=NaN

  op(NaN,X) = Na +∞+(-∞)=NaN

  +∞-(+∞)=NaN ∞/∞=NaN

  etc.

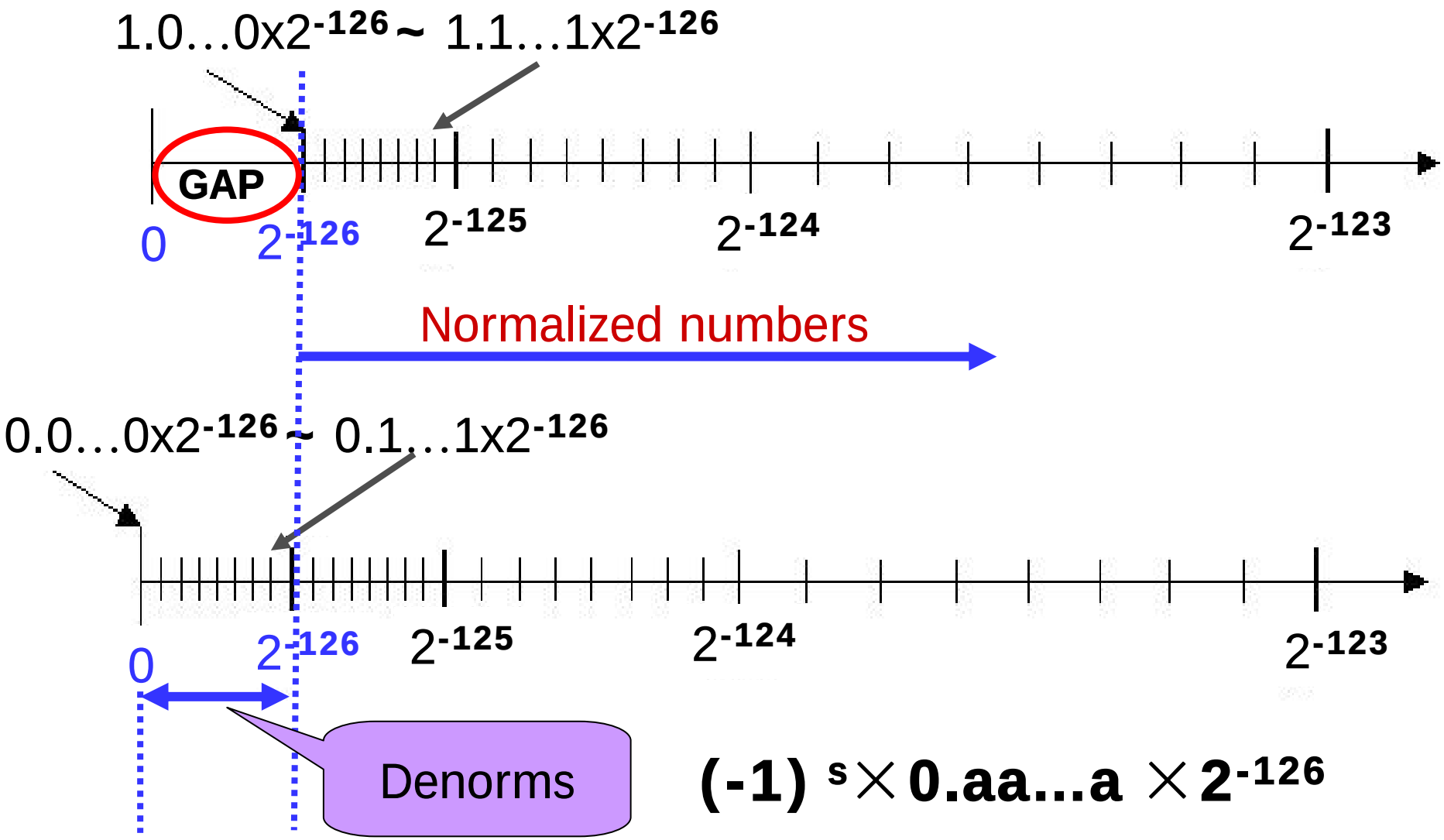
Representation for Denorms(非规格化数)

What have we defined so far? (Single Precision)

Exponent	Significand	Object
0	0	+/-0
0	nonzero	Denorms
1-254	anything implicit leading 1	Norms
255	0	+/- infinity
255	nonzero	NaN

Used to represent Denormalized numbers

Representation for Denorms



Questions about IEEE 754

- What's the range of representable values?

  The largest number for single: +1.11…1X2^{127} （约 +2.0X10^{38}）

  How about double?

- What about following type converting: not always true!

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

How about double? True!

How about double? Not always true!

- How about FP add associative? FALSE!

  x = – 1.5 x 10^{38}, y = 1.5 x 10^{38}, z = 1.0

  (x+y)+z = (–1.5x10^{38}+1.5x10^{38}) +1.0 = 1.0

  x+(y+z) = –1.5x10^{38}+ (1.5x10^{38}+1.0) = 0.0

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)

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

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

上述运算结果可能出现以下几种情况：　最大允许的指数为多少？ 127！

阶码上溢：一个正指数超过了最大允许值=〉+∞/-∞/溢出

阶码下溢：一个负指数超过了最小允许值=〉+0/-0

最小允许的指数为多少？ -126！

尾数上溢：最高有效位有进位=〉右规

非规格化尾数：数值部分高位为0=〉左规

右规或对阶时，右段有效位丢失=〉尾数舍入　在运算过程中，添加保护位

IEEE建议实现时为每种异常情况提供一个自陷允许位。当允许自陷处理程序的异常情况发生时，就调用一个用户自陷处理程序执行。

IEEE754标准规定的五种异常情况

① 无效操作

- 操作中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 REM 0、 ∞ REM y 等

② 除以0

③上溢（阶码上溢）：对于SP，则指阶码 E >1111 1110 （指数大于127）

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

⑤ 结果不精确（舍入时引起）

浮点数加减法基本要点

浮点数加 / 减法步骤

（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 (假定 Ye > Xe，则结果的阶码为Ye)

(2) 对阶：将Xm右移Δe位，即尾数变为： $Xm\ 2^{Xe-Ye}$

(3) 尾数加减： $Xm\ 2^{Xe-Ye} \pm Ym$ （保留右移的尾数部分：保护位）

如果需要规格化 (尾数形如: 1.xx…x)，则按 (4) 进行。

(4) 当尾数高位为0，需左规：尾数左移一次，阶码减1，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) 如果尾数比规定的长，则需考虑舍入（有多种舍入方式，后面将专门介绍）。

(6) 若尾数是0，则需要将指数也置0。为什么？

尾数为0说明结果应该为0，即：指数和尾数为全0。

浮点数加法运算举例

Example：用二进制形式计算 0.5 +(– 0.4375) =？

解1：$0.5=1.000\times2^{-1}$，$+(-0.4375)=-1.110\times2^{-2}$

Step1: -1.110×2^{-2} ➔ -0.111×2^{-1}

Step2: 1.000×2^{-1} $+(-0.111\times2^{-1})$
$= 0.001\times2^{-1}$

Step3: 0.001×2^{-1} ➔ 1.000×2^{-4}

Step4: None

结果为： **$1.000\times2^{-4} = 0.0001000=1/16= 0.0625$**

上述第二步浮点数的尾数如何相加减？采用补码加减运算吗？

不是，因为IEEE754标准的尾数采用原码表示，所以应采用原码加减运算。

如何进行原码加减运算？与补码加减运算有什么差别？（思考题）

浮点数加法运算举例

例子（同前）**x=0.5　y=-0.4375　求x+y=?**

解2：假定用IEEE754标准单精度格式表示

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

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

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

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

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

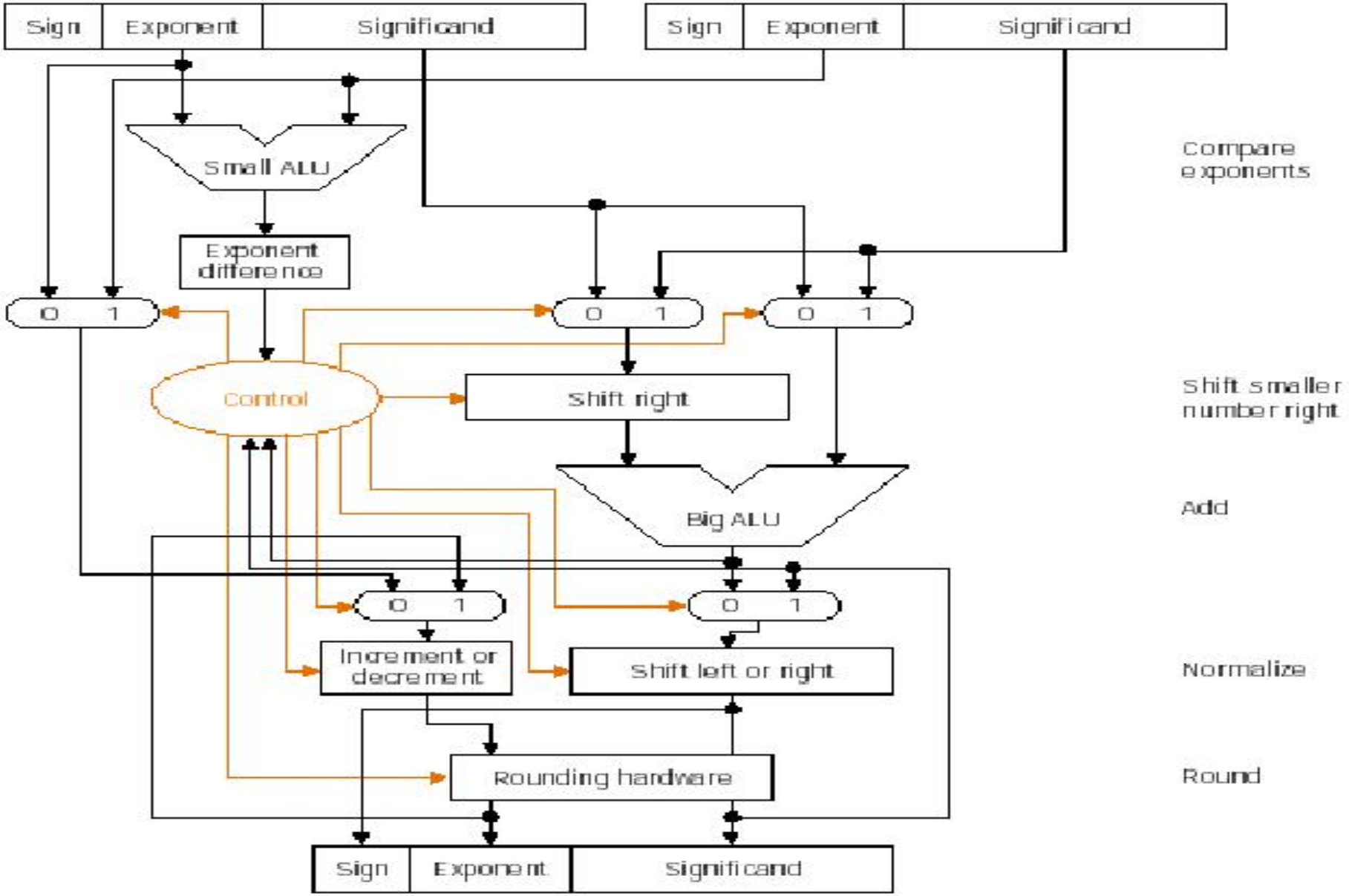
尾数相加：01.0000...0+(10.1110...0)=00.00100…0 (原码加法，最左边一位为符号)

左规：$+(0.00100…0)_2x2^{-1}=+(1.00…0)_2x2^{-4}$ (阶码减3)

$[x+y]_{浮}$=0 0111 1011 00…0

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

浮点加/减法器


Sign
Exponent
Significand
Sign
Exponent
Significand
Small ALU
Compare exponents
Exponent difference
0 1
0 1
0 1
Control
Shift right
Shift smaller number right
Big ALU
Add
0 1
0 1
Increment or decrement
Shift left or right
Normalize
Rounding hardware
Round
Sign
Exponent
Significand

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： $A/B=(M_a / M_b)\cdot 2^{Ea-Eb}$

浮点数乘 / 除法步骤

（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： $Xe \pm Ye \mp 127$

(2) 尾数相乘除： Xm */Ym （保留右边多出来的尾数部分：保护位）

(3) 两数符号相同，结果为正；两数符号相异，结果为负；

如果需要规格化 (尾数形如: 1.xx…x)，则按 (4) 进行。

(4) 当尾数高位为0，需左规：尾数左移一次，阶码减1，直到MSB为1。

每次阶码减1后要判断阶码是否下溢（比最小可表示的阶码还要小）

当尾数最高位有进位，需右规：尾数右移一次，阶码加1，直到MSB为1。

每次阶码加1后要判断阶码是否上溢（比最大可表示的阶码还要大）

(5) 如果尾数比规定的长，则需考虑舍入（有多种舍入方式，后面将专门介绍）。

(6) 若尾数是0，则需要将指数也置0。

上述第二步浮点数的尾数应采用原码乘除运算。

Extra Bits(附加位)

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

“浮点数就像一堆沙，每动一次就会失去一点沙，并捡回一点脏”

有谁能解释上述这段话的含义？

如何才能使失去的沙尽量少？

加多少附加位才合适？

Addition:

	1.xxxxx	**1.xxxxx**	**1.xxxxx**
+	**1.xxxxx**	**0.001xxxxx**	**0.01xxxxx**
	1x.xxxxy	**1.xxxxxyyy**	**1x.xxxxyyy**

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

Guard Digits(保护位): 在significand右边的位，用以保护对阶时右移的位，最终规格化时可能会左移到significand中。

Rounding Digits(舍入位)

举例： B = 10, p = 3

假定采用两位附加位

0	2	2.34	=	$2.3400 * 10^2$
0	0	2.56	=	$0.0256 * 10^2$
0	2	2.37	=	$2.3656 * 10^2$

在保护位右边有一位舍入位，规格化左移时可以根据其值进行舍入。对于加减运算，舍入位没有必要，但是，对于乘除运算，可能尾数最高位为0，需要左规，就用到舍入位了。

IEEE Standard:

four rounding modes: **round to nearest (default)**
round towards plus infinity (always round up)
round towards minus infinity(always round down)
round towards 0

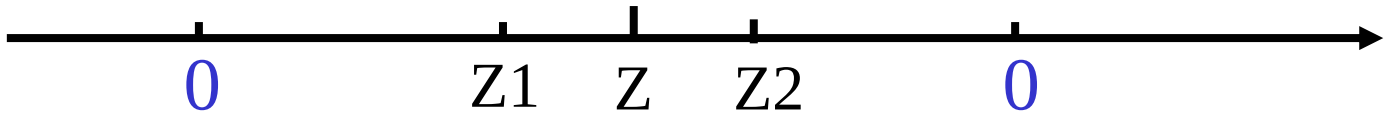
round to nearest:
round digit < B/2 then truncate(截取)
> B/2 then round up (add 1 to ULP)
= B/2 then round to nearest even digit

注：**ULP=units in the last place.**

可以证明默认方式得到的平均误差最小。

IEEE754的舍入方式的说明

IEEE754的舍入方式



(Z1和Z2分别是结果Z的最近可表示的左、右数)

(1) 就近舍入：舍入为最近可表示的数

(非中间值：0舍1入；

中间值：强迫结果为偶数-慢)

如：附加位为
01：舍
11：入
10：(强迫结果为偶数)

例：**1.110111 ~ 1.1110; 1.110101 ~ 1.1101; 1.110110 ~ 1.1110; 1.111110 ~ 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。即，总是舍入成Z1与Z2中绝对值较小的那个(正数：取Z1；负数：取Z2)

IEEE754通过在舍入位后再引入"粘位sticky bit"来简化"中间值"舍入问题。
加减运算对阶时，较小数的尾数右移后，舍入位之后有非0数，则可设置sticky bit。

实例:PowerPC和80x86中的浮点部件

◆PowerPC中的浮点运算

- 比MIPS多一条浮点指令：乘累加指令
  - 将两个操作数相乘，再与另一个操作数相加，写到结果操作数
  - 可以用一条乘累加指令代替两条MIPS浮点指令
  - 可以为中间结果多保留几位，得到最后结果后，再考虑舍入，精度高
  - 利用它来实现除法运算和平方根运算
- 浮点寄存器的数量多一倍（32xSPR, 32xDPR）

◆80x86中的浮点运算

- 采用寄存器堆栈结构：栈顶两个数作为操作数
- 寄存器堆栈的精度为80位(MIPS和PowerPC最多都是64位)
- 浮点数数据存取指令自动完成转换
- 指令类型：存取、算术、比较、函数（正弦、余弦、对数等）

本讲小结

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

本章总结（1）

数据信息有两大类：数值数据与非数值数据

- 数值数据：在数轴上有对应的点、能比较大小的数。
  - 二进制表示
    - 无符号数(unsigned int)：正整数，用来表示地址等
    - 带符号数：分定点数和浮点数两种
      - » 定点整数(short / int / long)：表示整数，用补码表示。
      - » 定点小数：表示浮点数中的尾数部分。
      - » 浮点数(float / double)：用来表示实数。现代计算机统一用IEEE754标准表示浮点数。尾数用原码定点小数表示，指数用移码定点整数表示。
  - 十进制表示（可直接用ASCII码表示）
    - 用二进制对十进制数进行编码，称为BCD码。一般用8421码表示。
- 非数值数据：在数轴上没有对应的点的数据。
  - 逻辑数(Bool)
  - 西文字符、汉字等 (char)

本章总结（2）

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，大数减小数，结果取大数的符号。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（3）

- 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
  - 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

3.2

3.5

3.7

3.9

3.10

3.12

3.27

3.30

3.35

3.37

3.43

3.44

附录：Units of a binary number

- ***bit***
  - **“on”/ “off ” state**
  - **“high” / “low” voltage**
- ***Byte***
  - 最小可寻址单位***(addressable* unit *)***
  - 可寻址意味着在存储部件中可访问到
- ***word***
  - 最常用的长度为**16, 32, or 64 bits.**
  - 在一个字编址（ **word-addressable**）系统中, 最小寻址单位为字

‘b’表示 bit, ‘B’表示 Byte

附录： Decimal / Binary（十 / 二进制数）

- **The decimal number 5836.47 in powers of 10:**

$$5 \times 10^3 + 8 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 + 4 \times 10^{-1} + 7 \times 10^{-2}$$

- **The binary number 11001 in powers of 2 :**

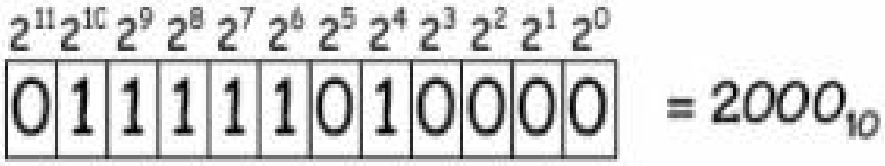
$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$= 16 + 8 + 0 + 0 + 1 = 25$$

- 用一个下标表示数的基（ **radix / base**）

$$11001_2 = 25_{10}$$

附录： Octal / Hexadecimal (八 / 十六进制数)



$$v = \sum_{i=0}^{n-1} 2^i b_i$$

2^3=8

03720

Octal - base 8

000 - 0
001 - 1
010 - 2
011 - 3
100 - 4
101 - 5
110 - 6
111 - 7

2^4=16

0x7d0

Hexadecimal - base 16

0000 - 0	1000 - 8
0001 - 1	1001 - 9
0010 - 2	1010 - a
0011 - 3	1011 - b
0100 - 4	1100 - c
0101 - 5	1101 - d
0110 - 6	1110 - e
0111 - 7	1111 - f

计算机用二进制表示所有信息！

为什么要引入 **8 / 16**进制？

8 / 16进制是二进制的简便表示。便于阅读和书写！

它们之间对应简单，转换容易。

在机器内部用二进制，在屏幕或其他外部设备上表示时，转换为**8/16**进制数，可缩短长度

附录： Conversions of numbers

(1) R进制数 => 十进制数

按“权”展开 **(a power of R)**

例1: $(10101.01)_2=1x2^4+1x\ 2^2+1x2^0+1x2^{-2}=(21.25)_{10}$

例2: $(307.6)_8=3x8^2+7x8^0+6x8^{-1}=(199.75)_{10}$

例1: $(3A.\ 1)_{16}=3x16^1+10x16^0+1x16^{-1}=(58.0625)_{10}$

(2)十进制数 => R进制数

整数部分和小数部分分别转换

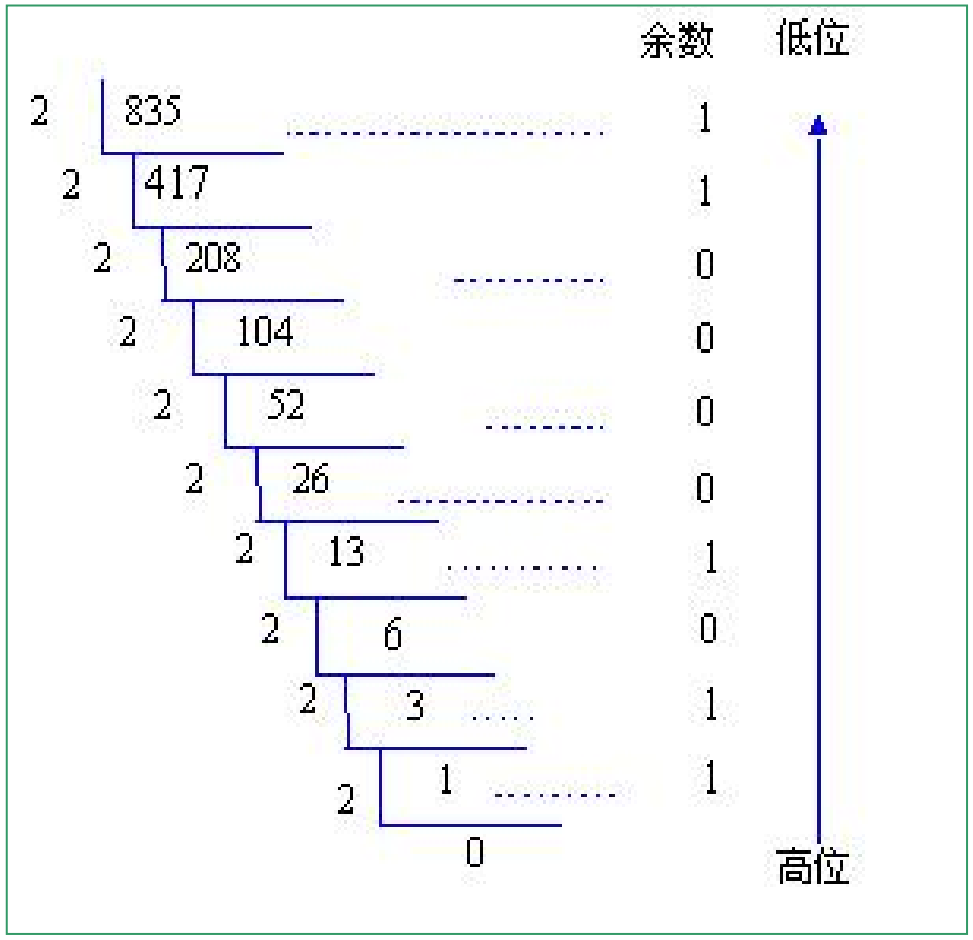
① 整数**(integral part)----**“除基取余，上右下左”

② 小数**(fractional part)----**“乘基取整，上左下右”

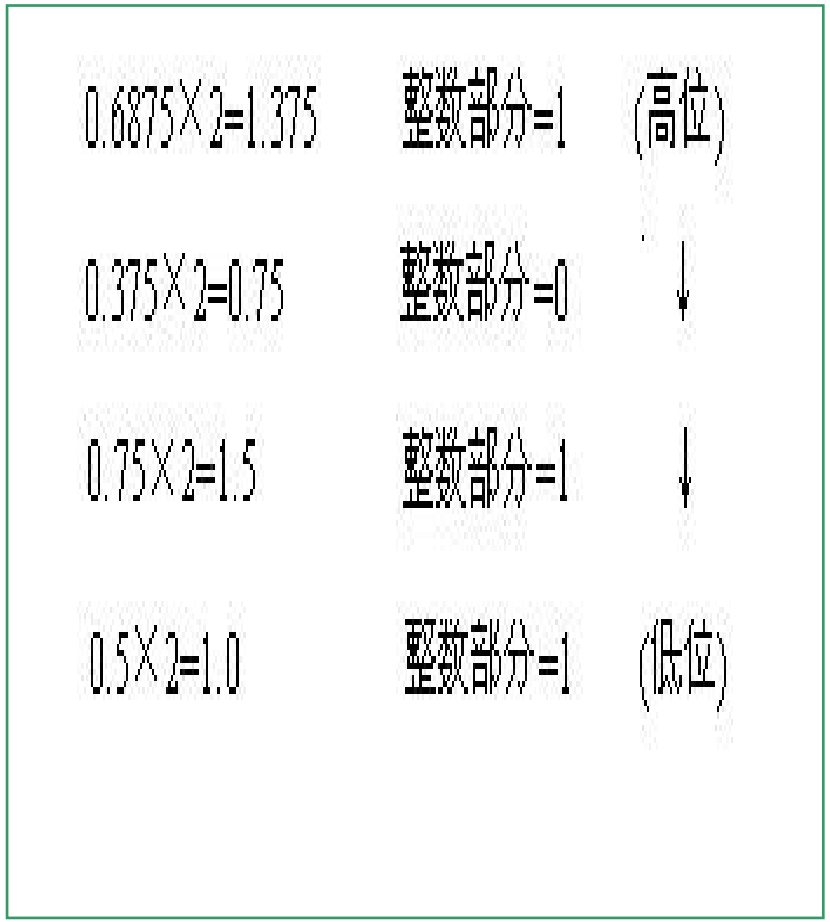
附录： Decimal to Binary Conversions

例1:$(835.6785)_{10}=(1101000011.1011)_2$

整数----“除基取余，上右下左”



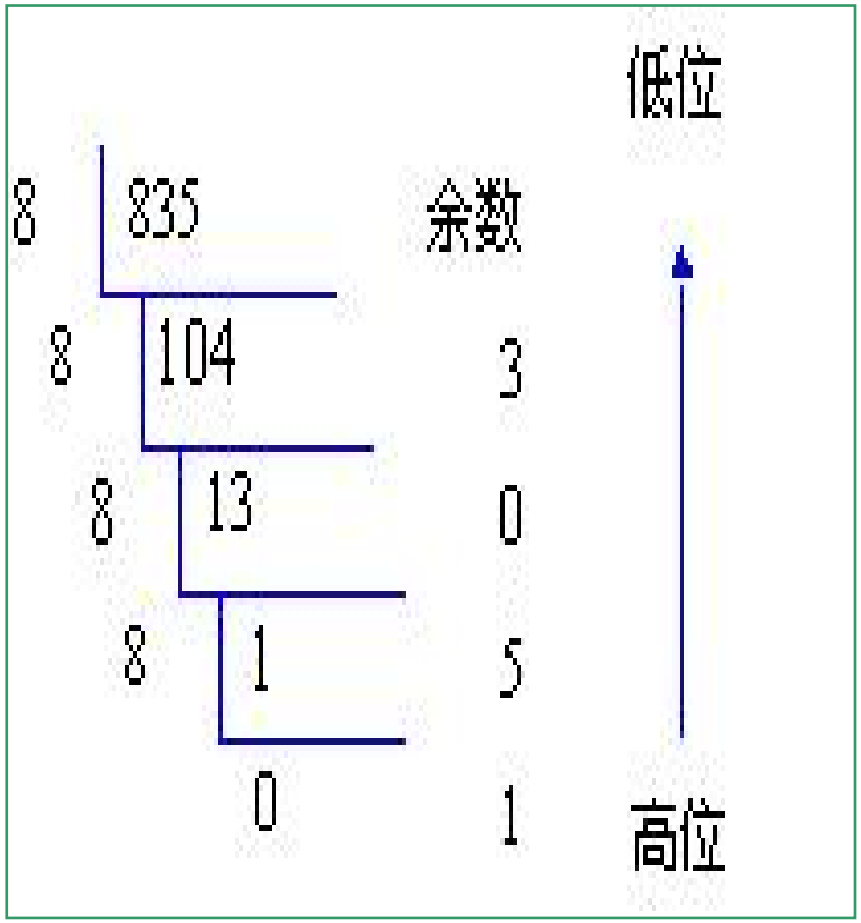
小数----“乘基取整，上左下右”



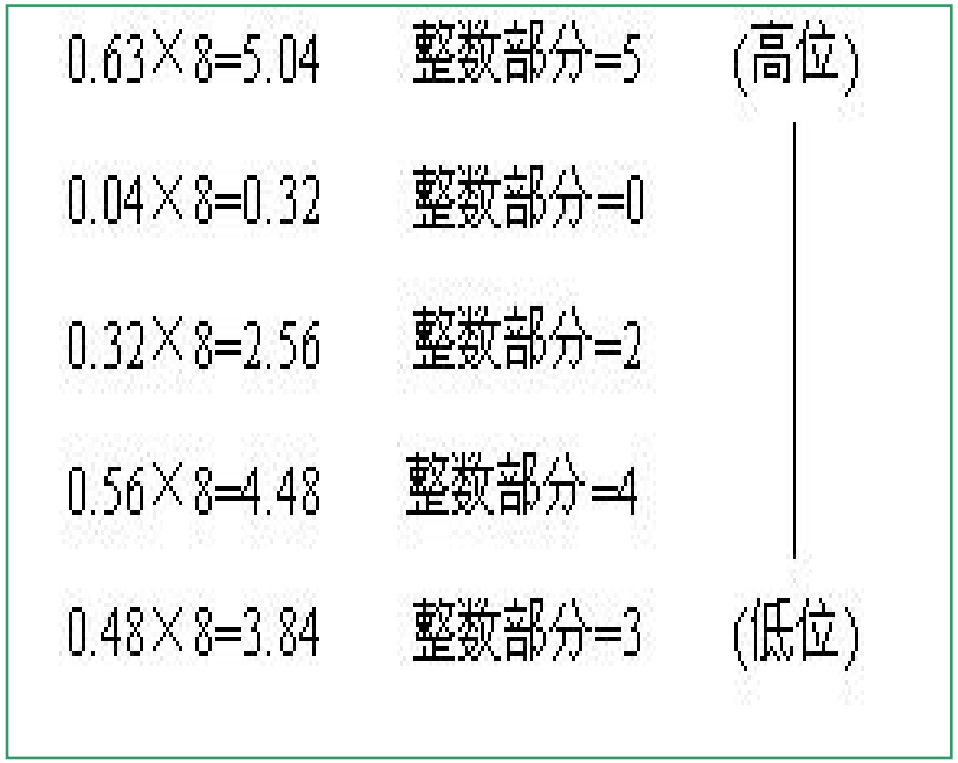
附录： Decimal to Binary Conversions

例2:$(835.63)_{10}=(1503.50243\ldots)_8$

整数----“除基取余，上右下左”　小数----“乘基取整，上左下右”



有可能乘积的小数部分总得不到0 ，此时得到一个近似值。



附录： Conversions of numbers

(3) 二/八/十六进制数的相互转换

① 八进制数转换成二进制数

$(13.724)_8=(\ 001\ 011\ .\ 111\ \ 010\ \ 100\ )_2=(1011.1110101)_2$

② 十六进制数转换成二进制数

$(2B.5E)_{16} = (00101011\ .\ 01011110)_2 = (101011.0101111)_2$

③ 二进制数转换成八进制数

$(0.10101)_2 = (\ 000\ .\ 101\ \ 010\ )_2 = (\ 0.52)_8$

④ 二进制数转换成十六进制数

$(11001.11)_2 = (\ 0001\ \ 1001\ .\ 1100\ )_2 = (\ 19.C\ )_{16}$

附录：十进制数的二进制编码（BCD）表示

- 数值数据（**numerical data**）的两种表示
  - **Binary (二进制数表示)**
    - **Fixed-point number (integer)**
    - **Floating-point number (real number)**
  - **Binary coded Decimal----BCD (二进制编码的十进制数表示)**
- 使用**BCD**码会耗费较多的设备量

附录： BCD码表示

- 编码思想

每个十进数位必须至少有4位二进制位来表示。而4位二进制位可以组合成16种状态，去掉10种状态后还有6种冗余状态。

- 编码方案

1. 十进制有权码
   - 指表示每个十进制数位的四个二进制数位（称为基2码）都有一个确定的权。8421码是最常用的十进制有权码。也称自然BCD（NBCD）码。
2. 十进制无权码
   - 指表示每个十进制数位的四个基2码没有确定的权。在无权码方案中，用的较多的是余3码和格雷码。
3. 其他编码方案 （5中取2码、独热码等）

附录： BCD码表示

- 一般计算机中有两种十进制数表示方式
  - 紧缩或紧凑**BCD (Packed BCD)**

    每四位表示一个十进制数位，如：

    $265=(0010\ 0110\ 0101)_{8421}$
  - 扩展**BCD (Unpacked BCD)**

    每八位表示一个十进制数位，高四位固定。如：

    $265=(xxxx0010\ xxxx0110\ xxxx0101)_{8421}$
- 符号位的表示：
  - **“+”：1100 ； “-”：1101**

附录：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

附录：十进制加法运算

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

附录：十进制加法运算要点

- 当运算结果的各位<=9时，不需校正
- 当某位>9时，该位需“+6”校正
- 当某位向高位有进位时，需将进位进到高位，同时该位“+6”校正（何时产生进位？）
- 当最高位有进位时，发生溢出

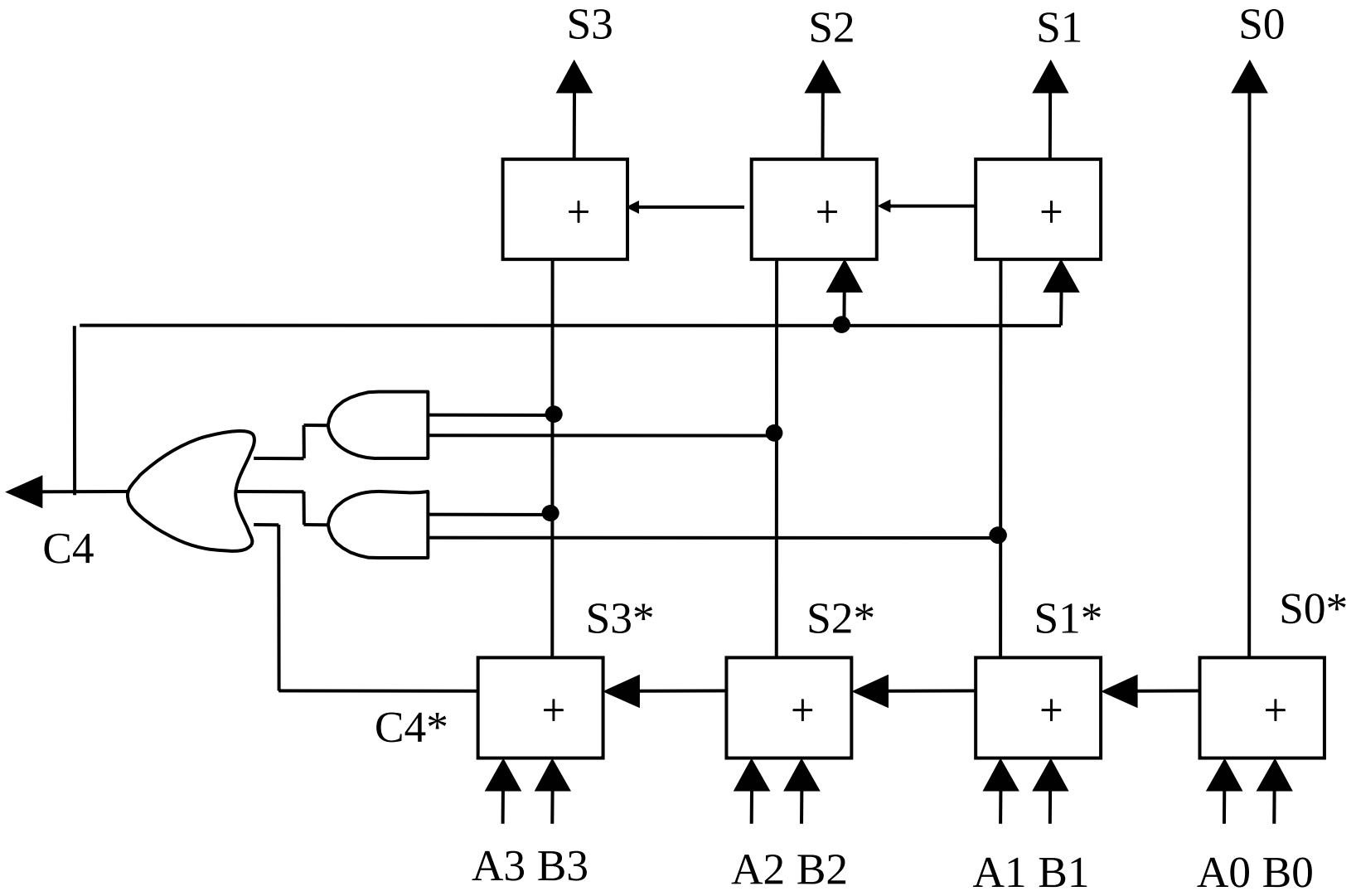
综上所述，当某位运算结果在10～19之间时，需进行校正。

(最大可能：2x9+1=19)

即：1x1x或11xx或有进位（ $C_4^*=1$ ）

校正逻辑表达式：$C_4=C_4^*+S_3^*S_1^*+S_3^*S_2^*$

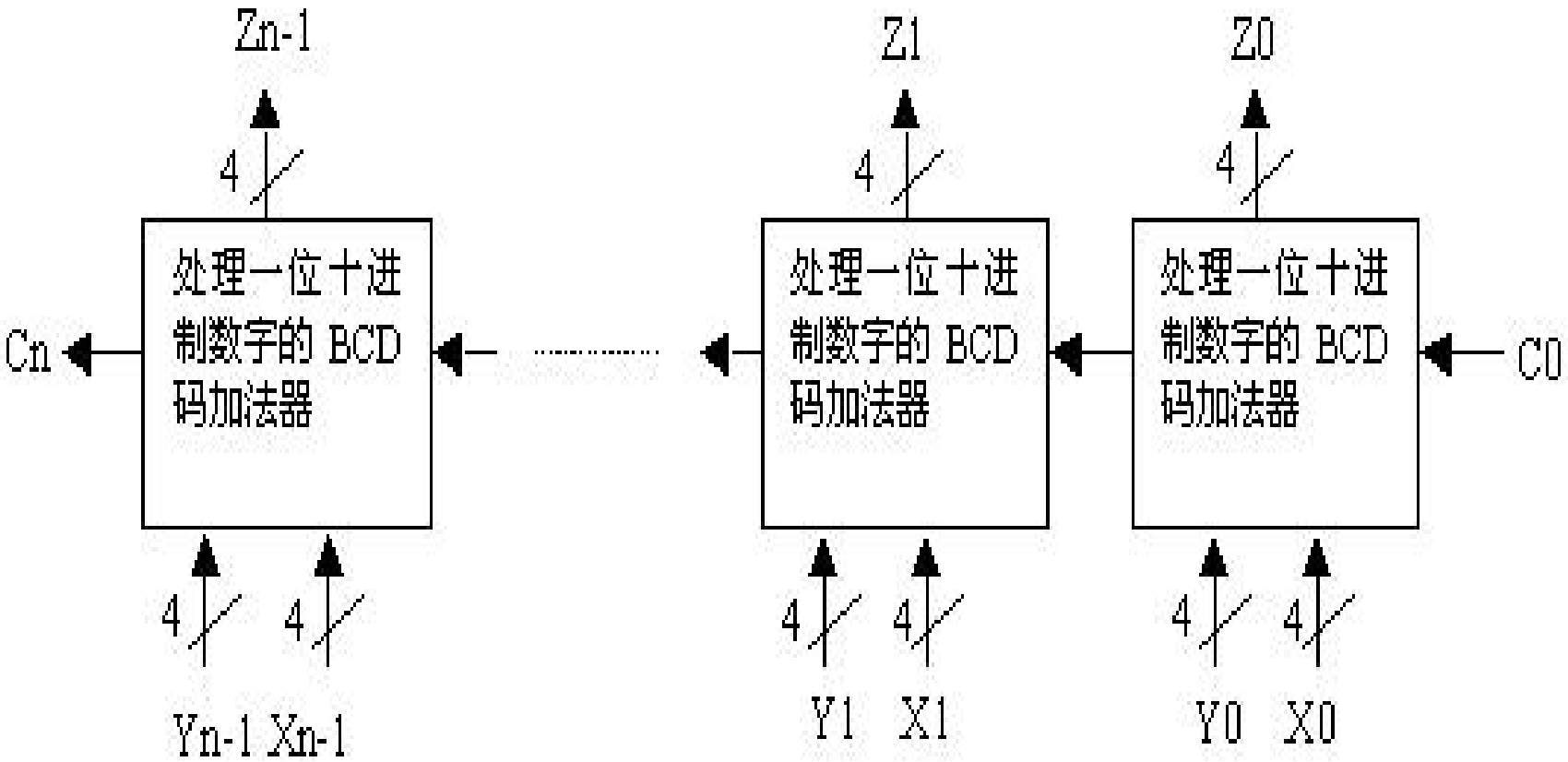
附录：一位十进制加法器



附录： n位十进制加法器

- n个一位十进制加法器=〉

一个n位十进制串行加法器



附录：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。

例：$[7]_{反}=\bar{0}\ \bar{1}\ \bar{1}\ \bar{1}+1010 = 1000+1010 = 0010 = 2$

$=0\ 1\ 1\ 1+0110 = 1101 = 0010 = 2$

附录：十进制减法运算举例

例1 309-125=184　　例2 125-309 =-184

加补码：309+875=184　　125+691=-184（mod 10^3）

```
  0011 0000 1001          0001 0010 0101
+ 1000 0111 0101        + 0110 1001 0001
-----------------       -----------------
  1011 0111 1110          0111 1011 0110
  0110      0110              0110
-----------------       -----------------
 10001 1000 0100          1000 0001 0110
```

进位为1，表示被减数大于减数，结果为正

无进位，表示差值为负数，故应将结果取补

取补

-0001 1000 0100