计算机组织与系统结构

设计过程与ALU设计

The Design Process & ALU Design

（第六讲）

程 旭

2020.11.19

设计过程

“设计即描绘 （To Design Is To Represent）”

设计活动 产生 有关一个对象的 描述/表达

- 传统的工匠并不区分 概念化（conceptualization） 和 工艺品（artifact）
- 由于 *复杂性*，这两者才开始分离
- 概念是用一种或多种 表示语言（*representation languages) 描述的*
- 概念化的过程 就是 设计

设计从需求分析开始

-- 功能需求: 将要完成的功能
-- 性能特征: 速度、功率、面积、成本, . . .

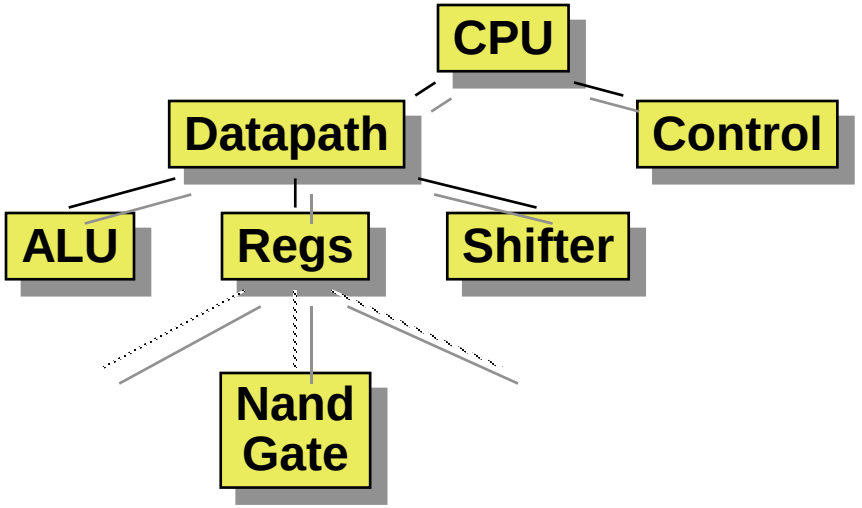
设计过程（续）

设计以组装结束 ***Design Finishes As Assembly***

-- 设计 分解 组成部件，以及如何组装部件

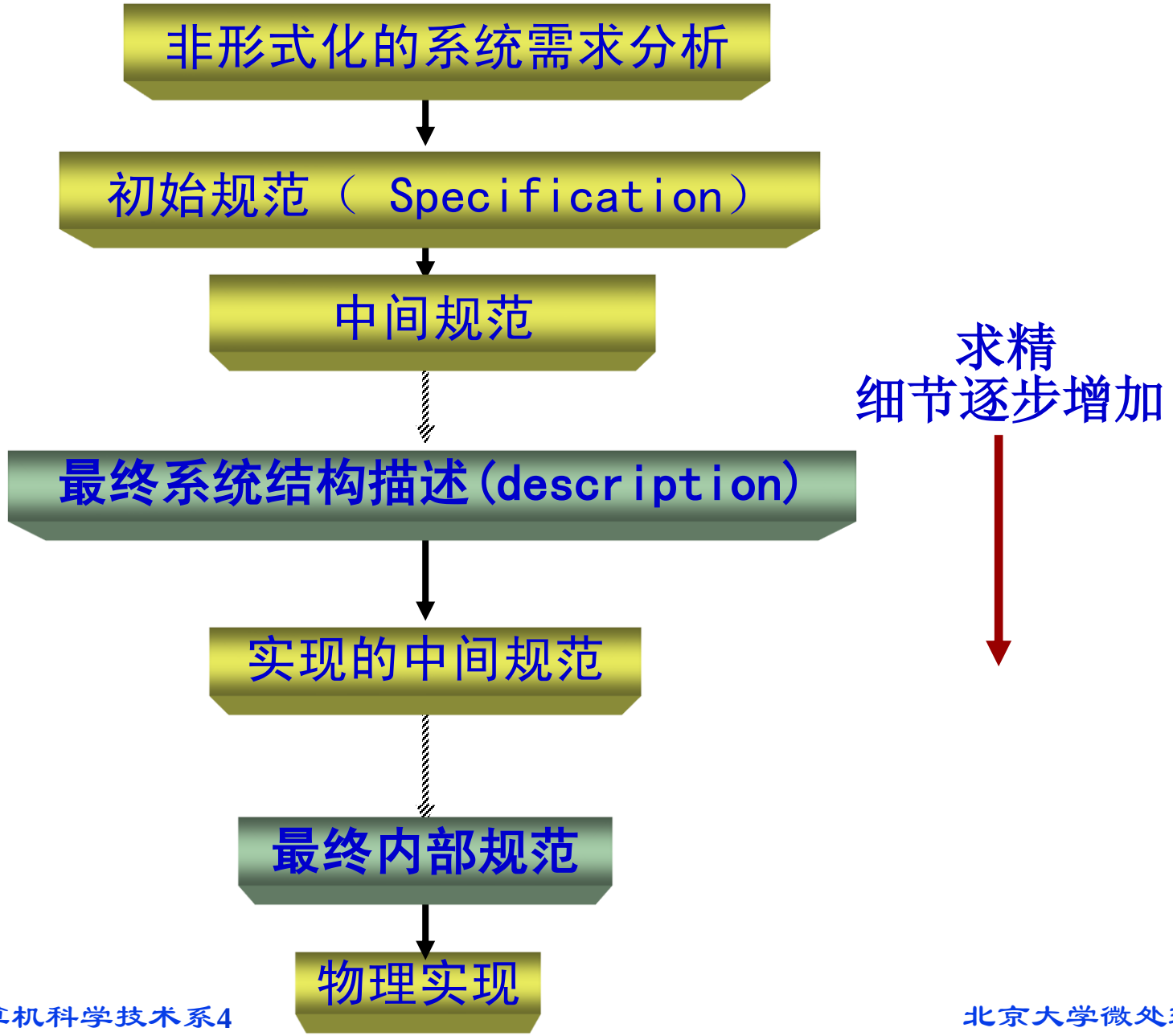
-- 自顶向下 将 复杂的功能（行为）
分解为
多个基本功能

-- 自底向上 将 多个基本 功能块
合成为
更复杂的组装体

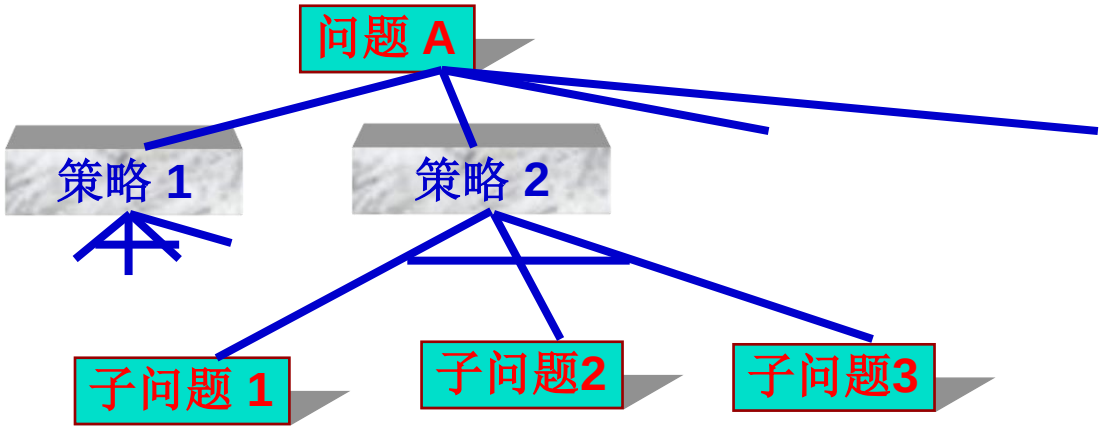


设计是一个“创造性的过程”，而不仅仅是使用某种简单的方法

设计求精


非形式化的系统需求分析
初始规范（ Specification）
中间规范
最终系统结构描述(description)
实现的中间规范
最终内部规范
物理实现
求精
细节逐步增加

设计如同搜寻



设计包含利用所学知识进行猜测和验证

-- 给定目标, 如何确定这些策略的优劣?
-- 给定可选的设计策略, 应该选择那种设计?
-- 给定部件和组装体的空间, 哪一部分能够产生最佳解决方案?

可行(好)抉择 vs. 最佳抉择

设计如同陈述（Representation）

(1) 功能描述

“VHDL行为”

输入： 2 x 16位操作数：A, B； 1 位进位输入：Cin.

输出： 1 x 16位结果：S； 1位进位输出：Co.

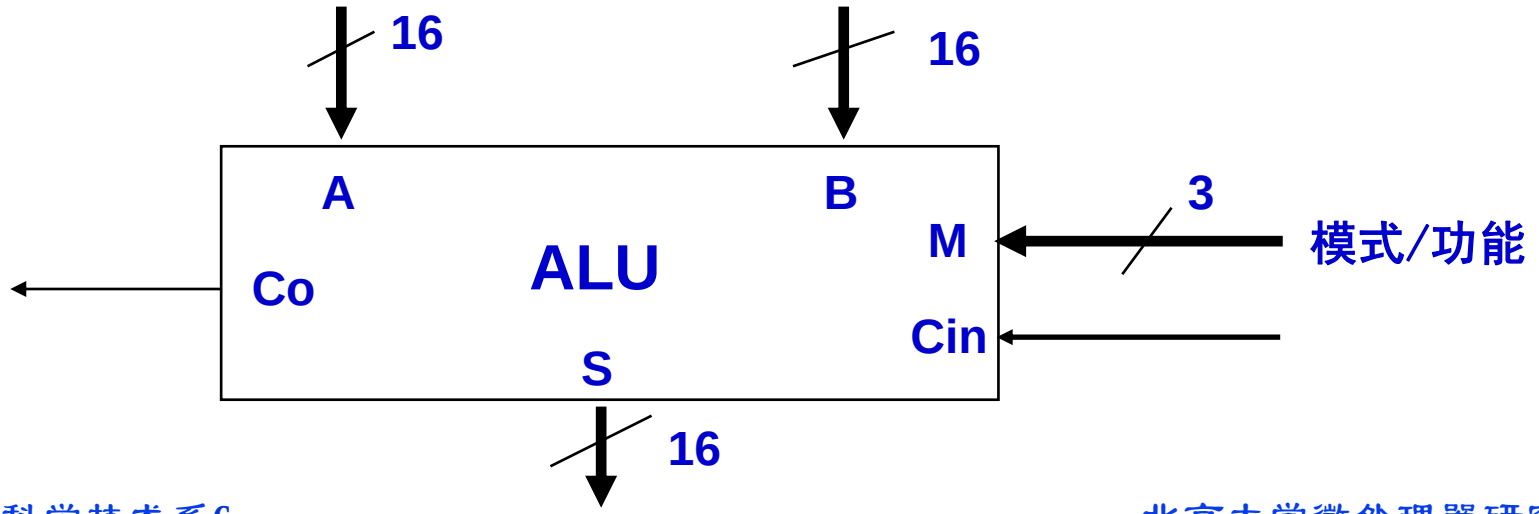
操作： PASS, ADD (A + B + Cin), SUB (A - B - Cin),
AND, XOR, OR, COMPARE (相等)

性能： 现在，还没有详细说明！

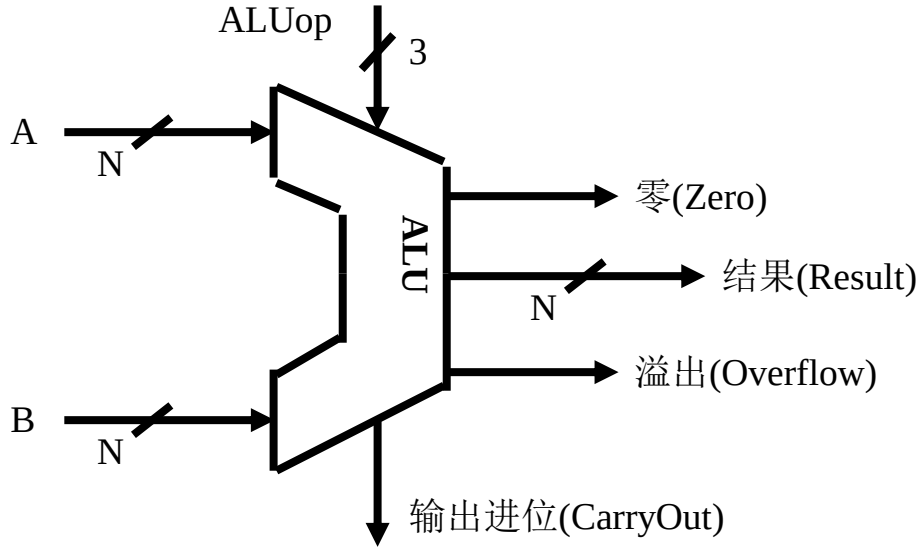
(2) 框图

“VHDL实体”

理解数据和控制流



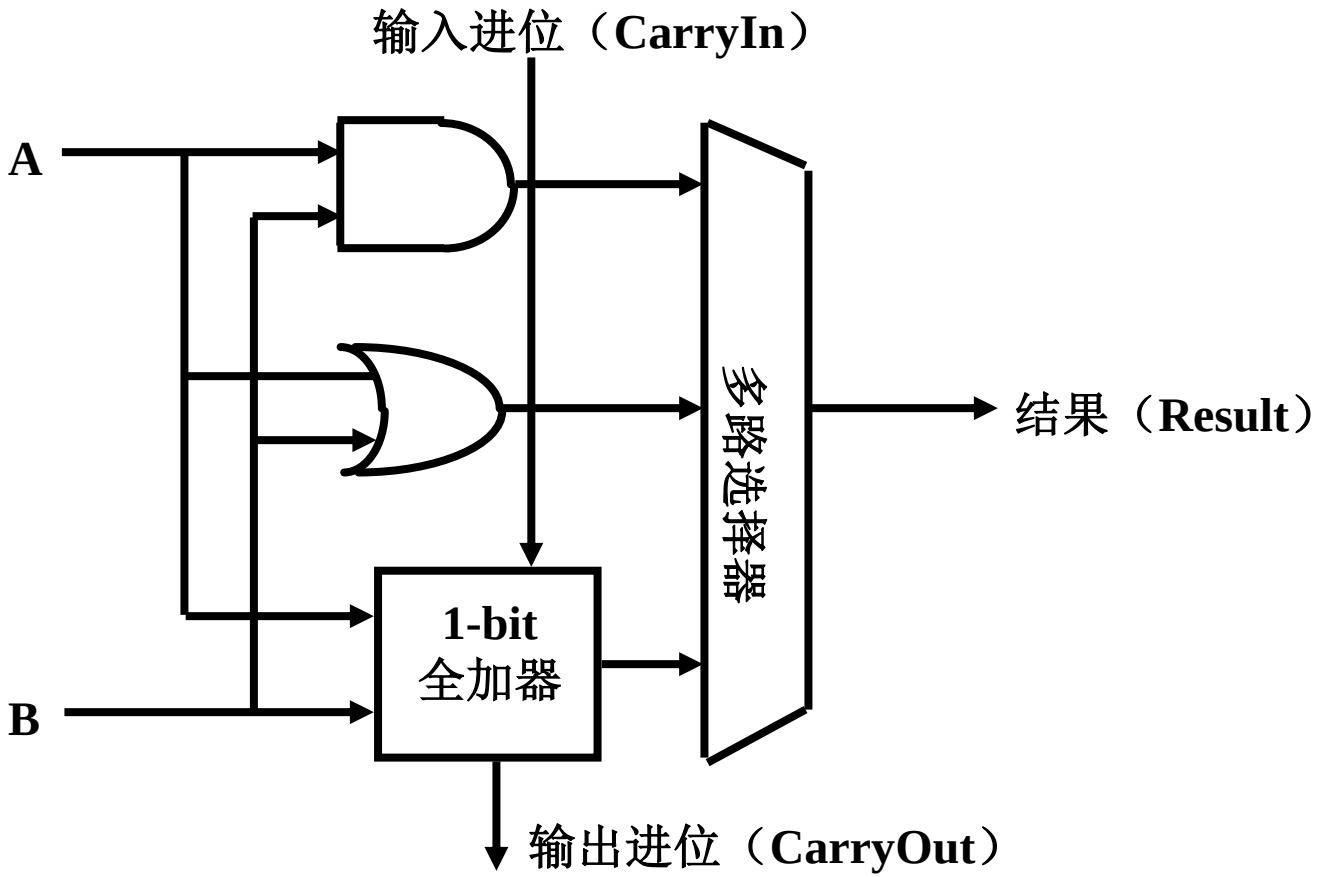
ALU的功能描述



ALU 控制线 (ALUop)	功能
000	And
001	Or
010	Add
110	Subtract
111	Set-on-less-than

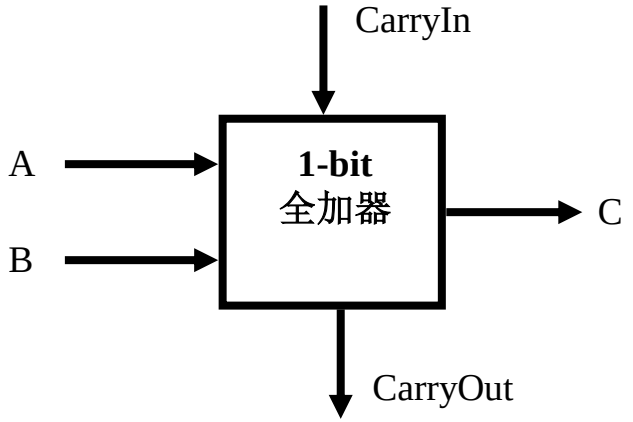
1位 ALU

- 该 1位 ALU将完成 AND、OR和 ADD



1位全加器

- 又称为 “ (3, 2) adder”
- 半加器: 没有CarryIn 和 CarryOut
- 真值表:



输入			输出		
A	B	CarryIn	CarryOut	Sum	注释
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

CarryOut的逻辑方程式

输入			输出		注释
A	B	CarryIn	CarryOut	Sum	
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

- CarryOut = (!A & B & CarryIn) | (A & !B & CarryIn) | (A & B & !CarryIn) | (A & B & CarryIn)
- CarryOut = B & CarryIn | A & CarryIn | A & B

Sum的逻辑方程式

输入			输出		注释
A	B	CarryIn	CarryOut	Sum	
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

- Sum = (!A & !B & CarryIn) | (!A & B & !CarryIn) | (A & !B & !CarryIn) | (A & B & CarryIn)

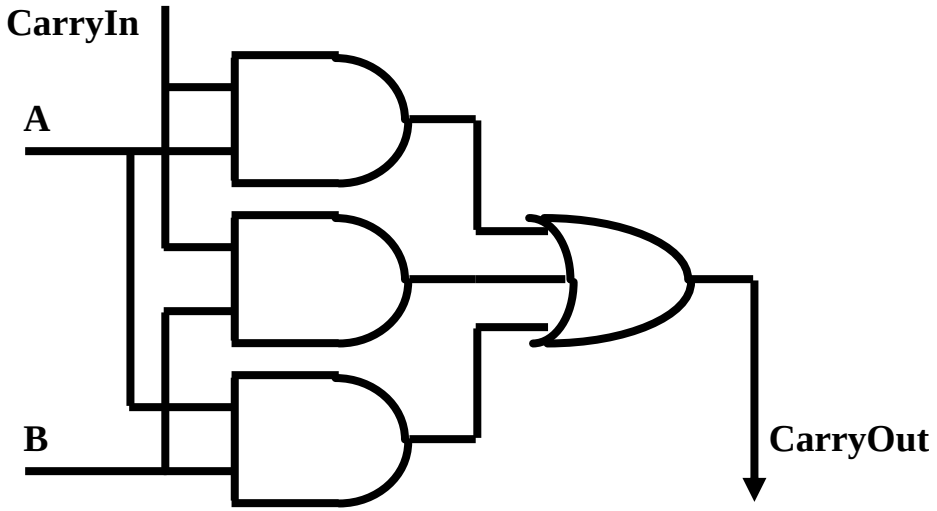
Sum的逻辑方程式（续）

- Sum = (!A & !B & CarryIn) | (!A & B & !CarryIn) | (A & !B & !CarryIn) | (A & B & CarryIn)
- Sum = A XOR B XOR CarryIn
- XOR的真值表:

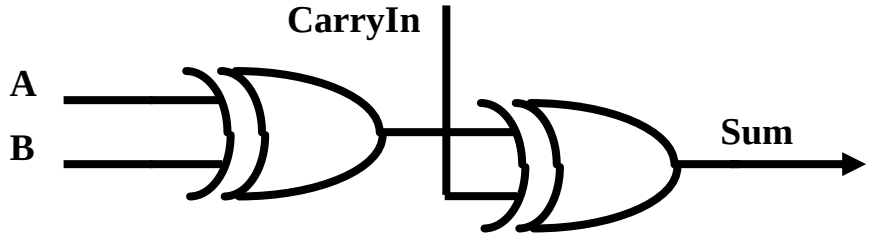
X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

CarryOut 和 Sum 的逻辑图

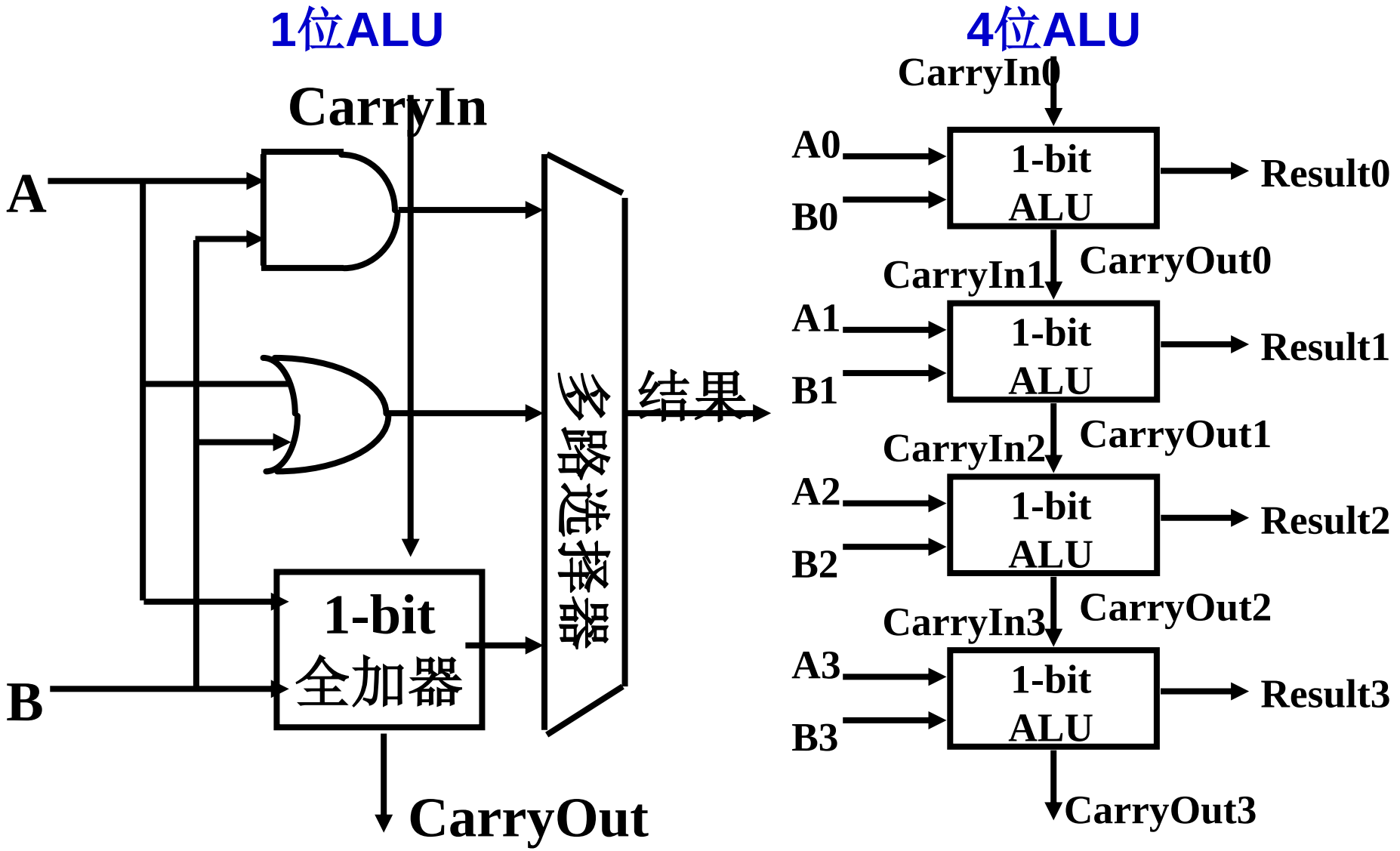
- CarryOut = B & CarryIn | A & CarryIn | A & B



- Sum = A XOR B XOR CarryIn



4位ALU


1位ALU
CarryIn
A
B
1-bit
全加器
多路选择器
结果
CarryOut
4位ALU
CarryIn0
A0
B0
1-bit
ALU
Result0
CarryOut0
CarryIn1
A1
B1
1-bit
ALU
Result1
CarryOut1
CarryIn2
A2
B2
1-bit
ALU
Result2
CarryOut2
CarryIn3
A3
B3
1-bit
ALU
Result3
CarryOut3

设计诀窍1：分治(divide & conquer)

- 将整个问题分解为较简单的小问题，在各个击破后，综合起来得到整个问题的解决方案。
- 例如：假设在设计我们的ALU之前，已经妥善处理了立即数问题
  - 10 个操作类型 (4 位)

00	add
01	addU
02	sub
03	subU
04	and
05	or
06	xor
07	nor
12	slt
13	sltU

逐步求精后的需求情况

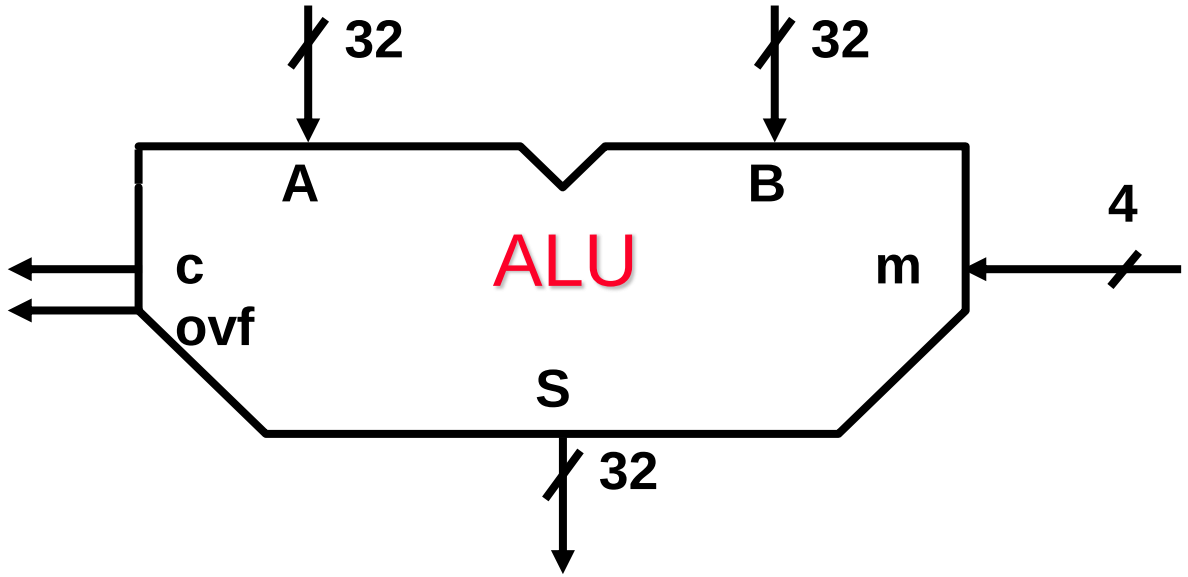
(1) 功能描述

输入: 2 x 32位操作数 A, B, 4位模式选择(mode)

输出: 32位结果S, 1位进位, 1位溢出

操作: add, addu, sub, subu, and, or, xor, nor, slt, sltU

(2) 框图(Block Diagram) (powerview symbol, VHDL entity)



行为表现：VHDL

VHSIC: Very High Speed Integrated Circuit

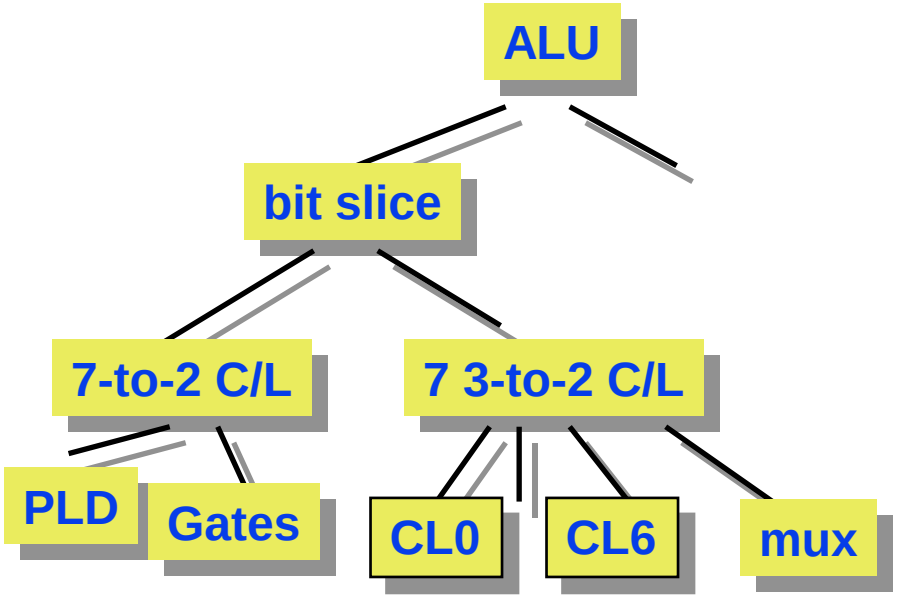
VHDL: VHSIC Hardware Description Language

```
Entity ALU is
   generic (c_delay: integer := 20 ns;
             S_delay: integer := 20 ns);

   port (  signal A, B:  in  vlbit_vector (0 to 31);
           signal    m:  in  vlbit_vector (0 to 3);
           signal    S: out  vlbit_vector (0 to 31);
           signal    c: out  vlbit;
           signal  ovf: out  vlbit)
    ...
end ALU;

    S <= A + B;
```

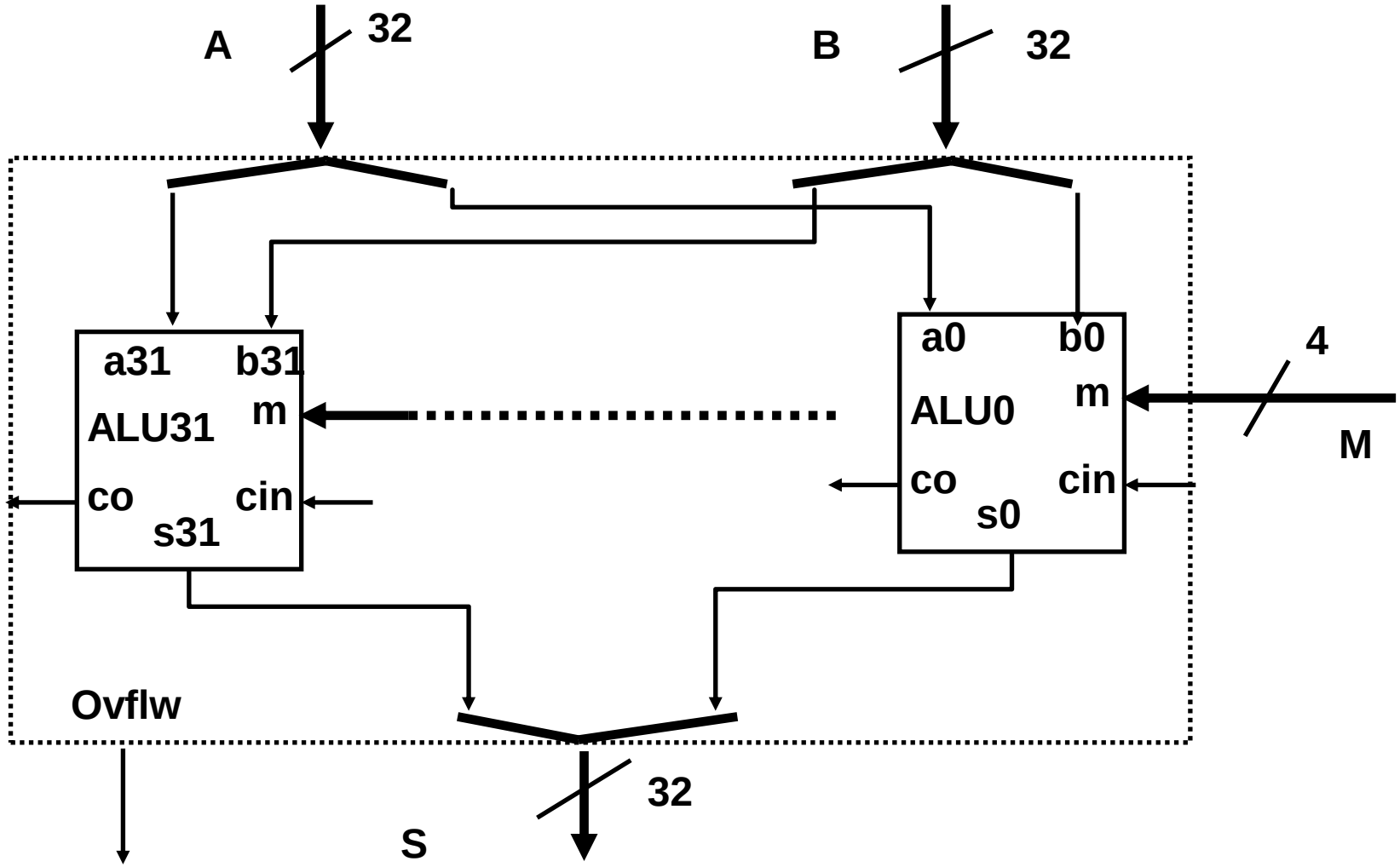
设计抉择



C/L:组合逻辑

- 简单位片（ bit-slice）
  - 大规模组合逻辑的问题
  - 许多小规模组合逻辑的问题
  - 划分成 2-step 的问题
- 具有先行进位的位片处理
-

细化的框图： 位片ALU


A
32
B
32
a31
b31
ALU31
m
co
cin
s31
a0
b0
ALU0
m
co
cin
s0
4
M
Ovflw
32
S

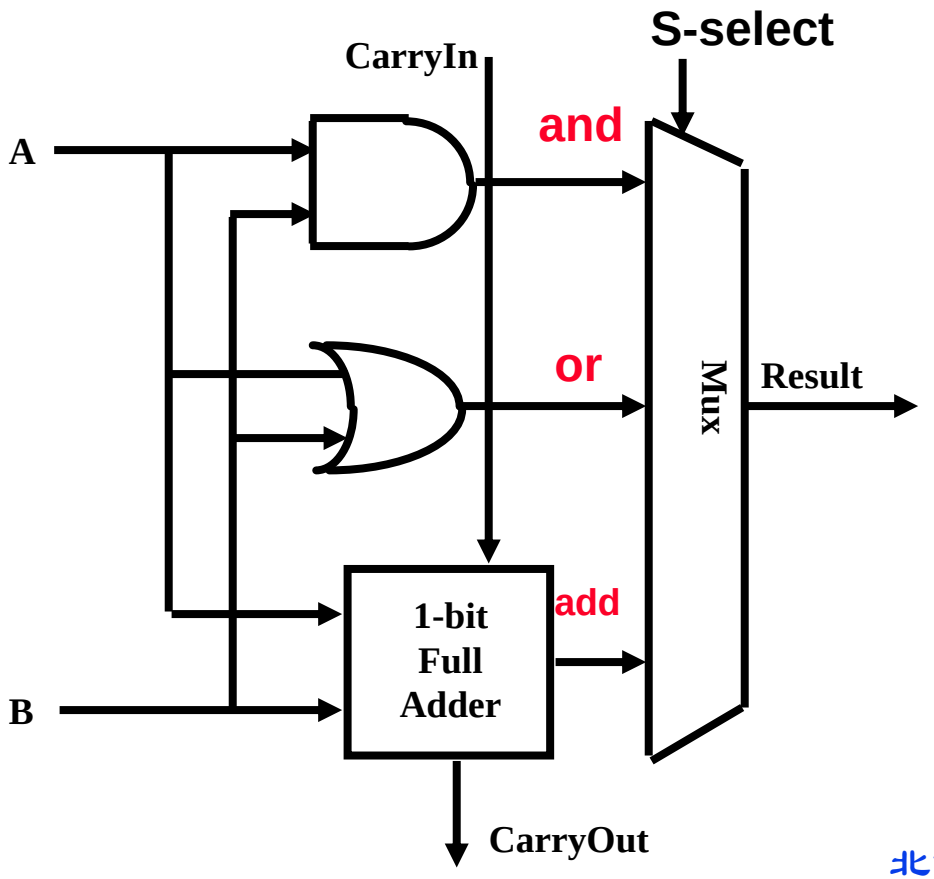
7-to-2 组合逻辑

- 设计起步 . . .

	Function	Inputs							Outputs		K-Map
		M0	M1	M2	M3	A	B	Cin	S	Cout	
0	add	0	0	0	0	0	0	0	0	0	
127											

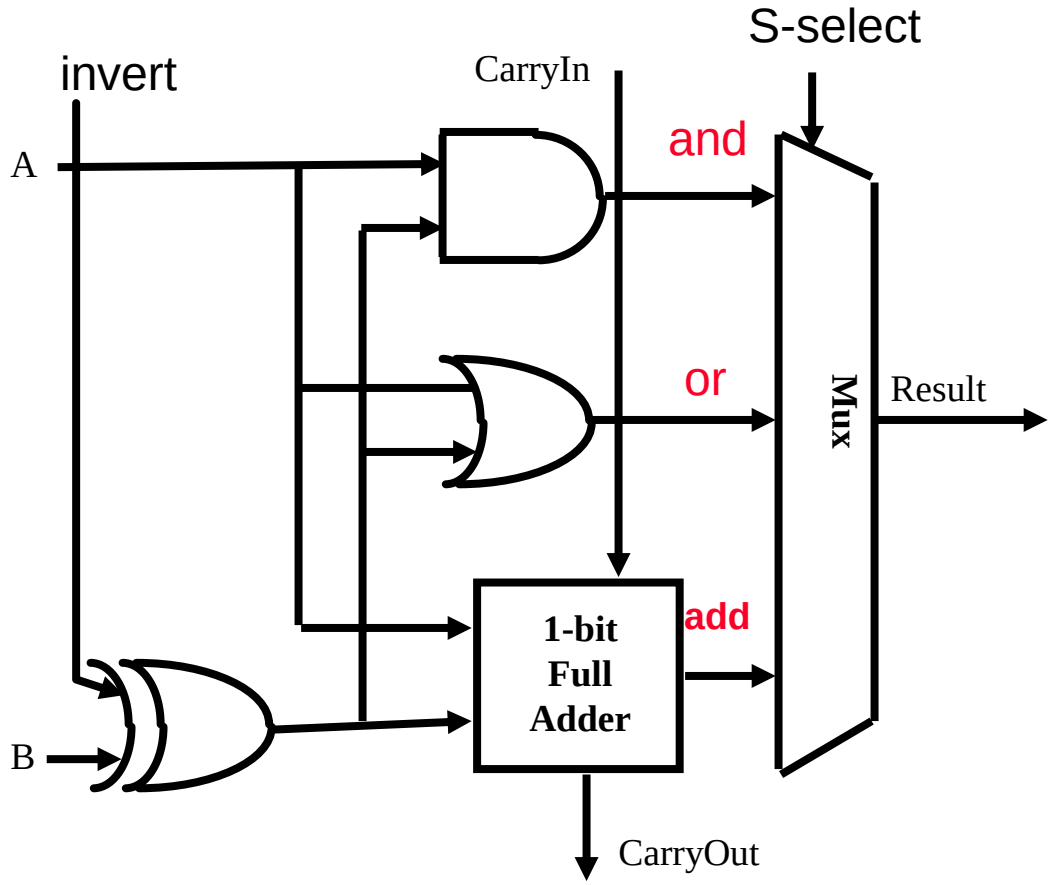
7个输入 + 一个MUX？

- **设计诀窍 2**：从你知道的 (或 可以想象的)局部方案开始设计，并努力将它们整合起来
- **设计诀窍 3**：先解决部分问题，再逐步扩充



其他操作

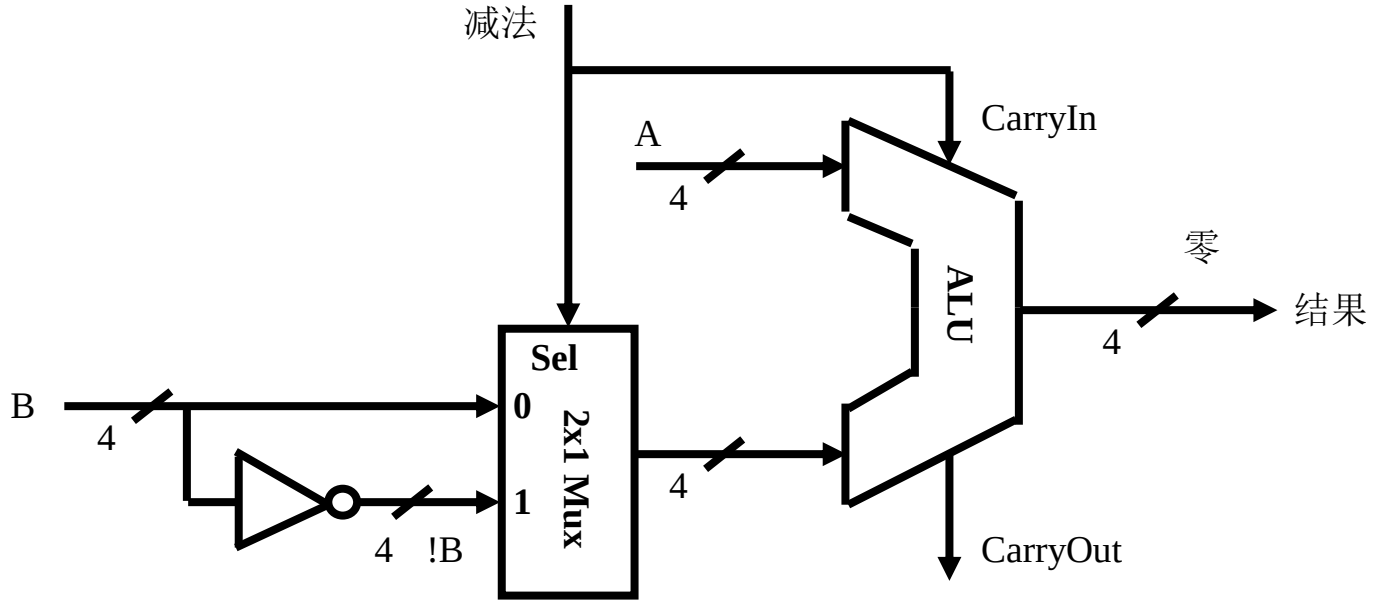
- A - B = A + (-B)
  - 通过 取反 再 加 1 获得负数的补码



Set-less-than? 课后思考

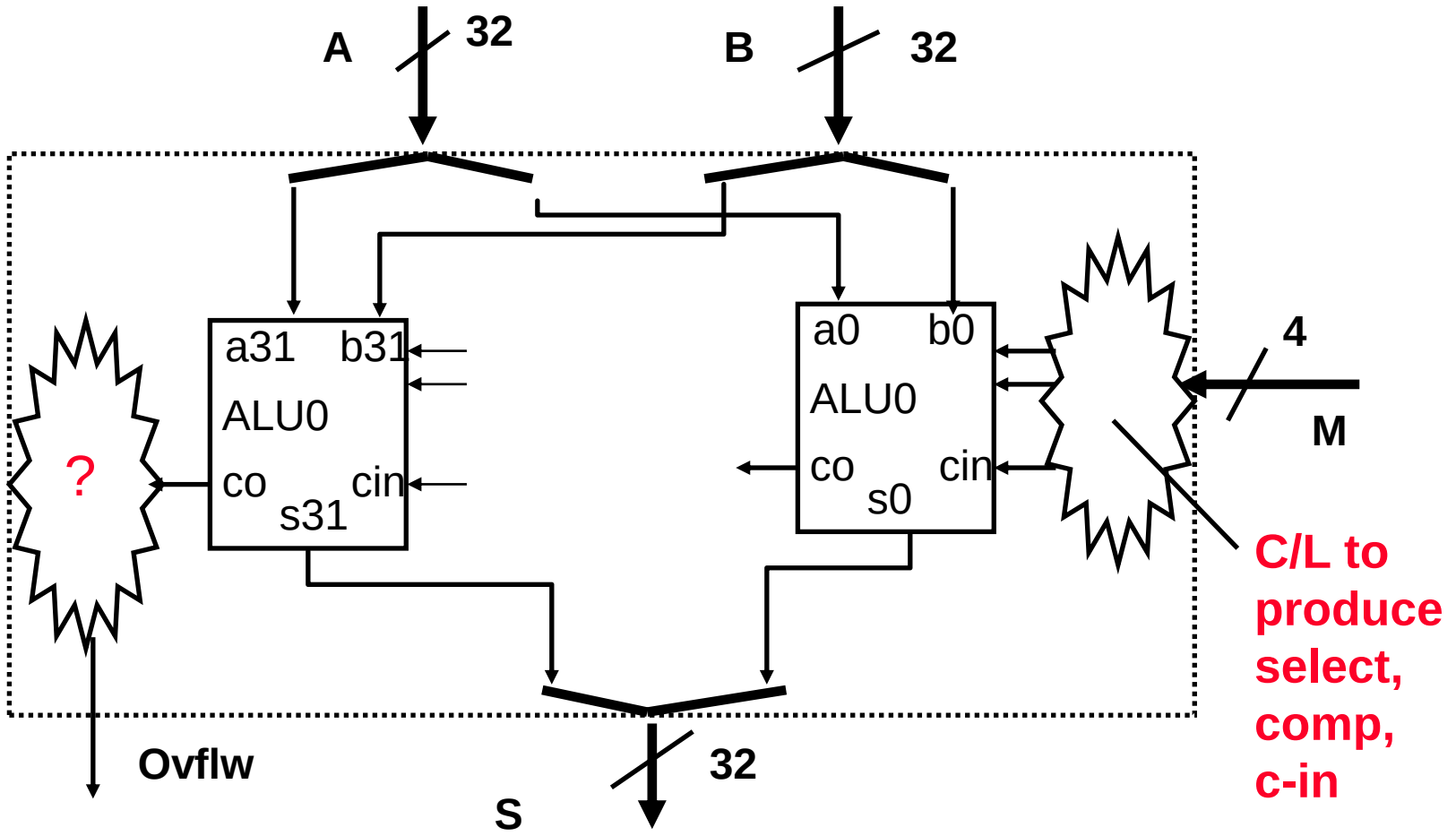
减法

- 记住:
  - (A - B) 可以看作: A + (-B)
  - 补码: 每位取反，末位加一
- 对B每位取反，即 !B:
  - A + !B + 1 = A + (!B + 1) = A + (-B) = A - B



进一步细化的框图

- LSB 和 MSB 需要进行特殊处理

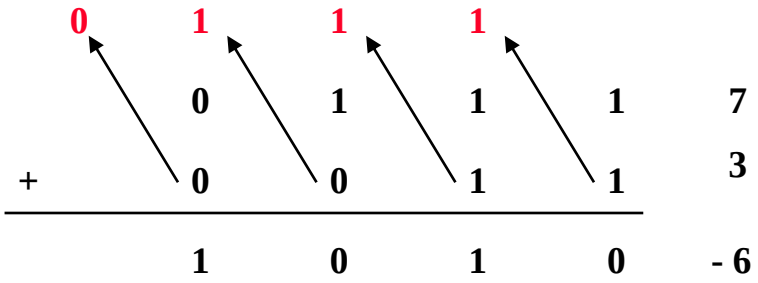


溢出

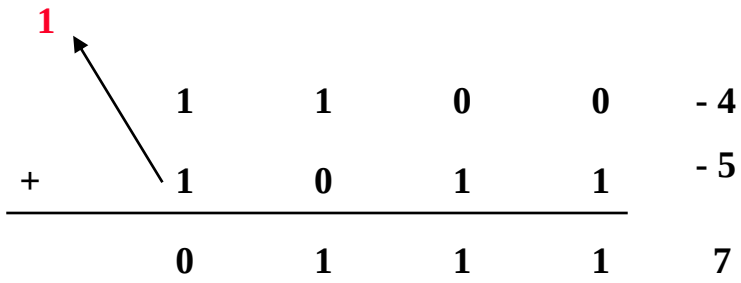
十进制	二进制
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

十进制	补码
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

◦ 示例: 7 + 3 = 10 但是 ...



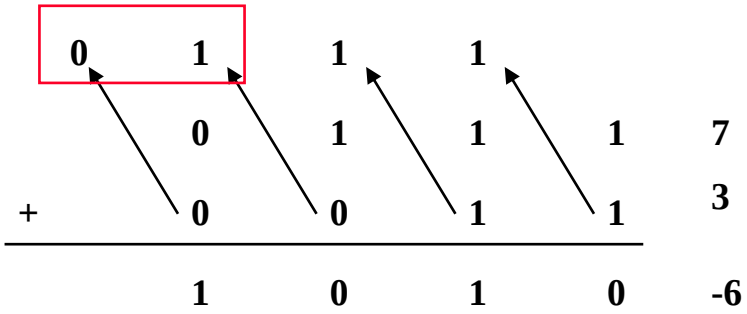
- 4 - 5 = - 9 但是 ...

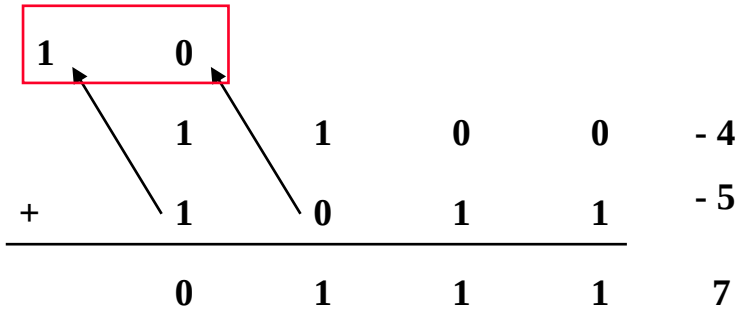


溢出检测

- 溢出：结果超出了正常的表示范围（太大 和 太小）
  - 例如：－ 8 ＜ ＝ 4位二进制数 <= 7
- 当对具有不同符号的操作数进行加法运算时，不会出现溢出！
- 当进行下列加法时，出现溢出：
  - 两个正数相加，结果为负
  - 两个负数相加，结果为正
- 溢出可以用如下方法检测：
  - 输入到最大位的进位 != 从最大位输出的进位

    （Carry into MSB ! = Carry out of MSB ）

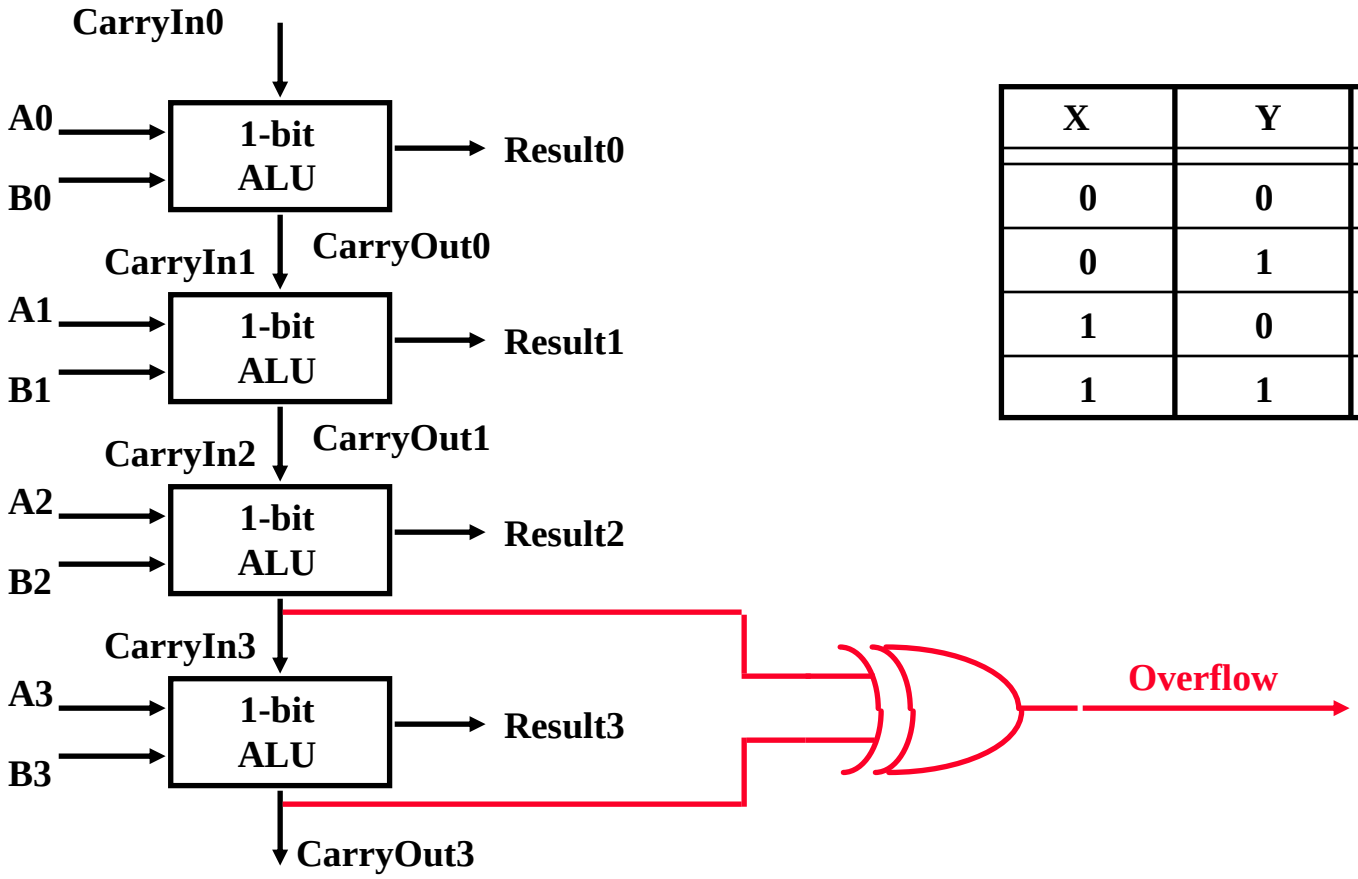




溢出检测逻辑

- 输入到最大位的进位 != 从最大位输出的进位

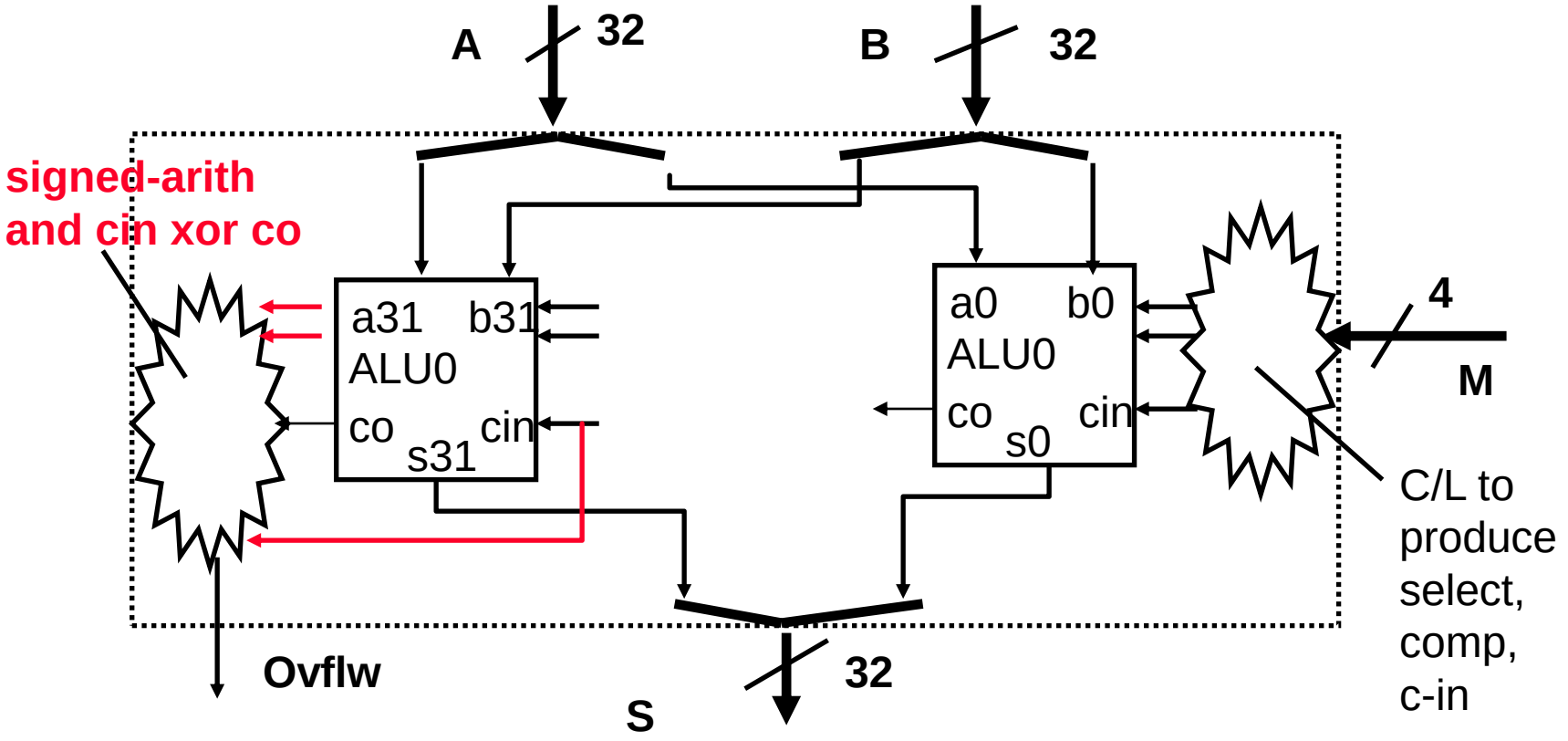
  对于N位ALU: 溢出 = CarryIn[N - 1] XOR CarryOut[N - 1]



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

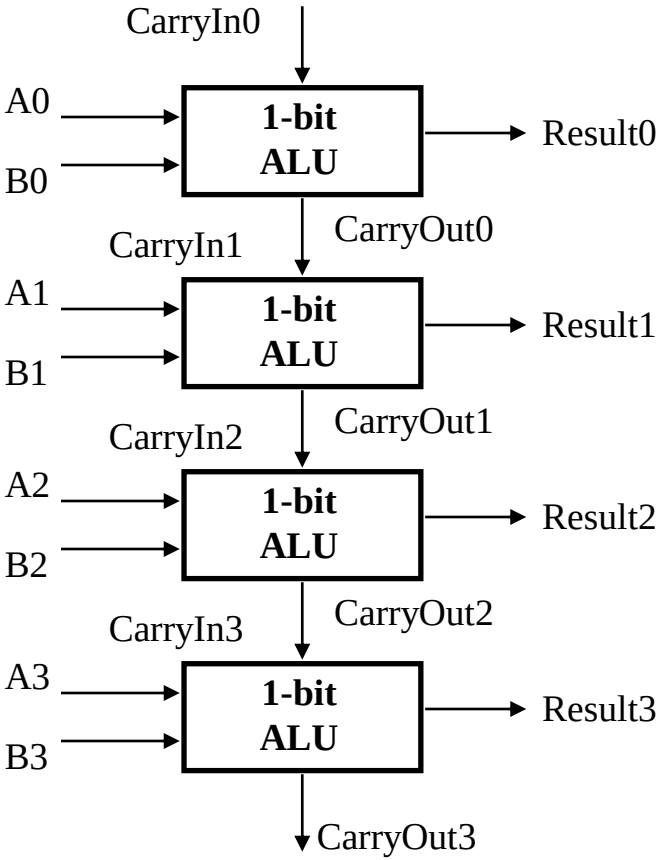
更加细化的框图

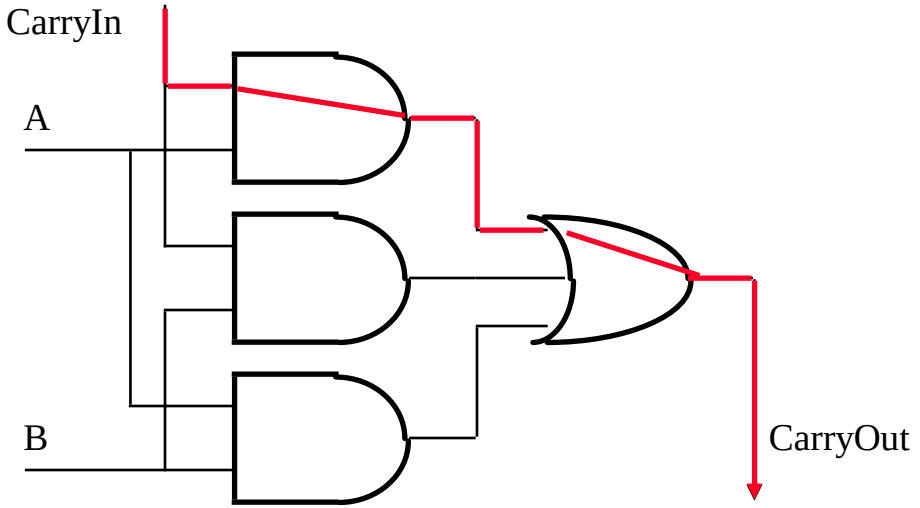
- LSB 和 MSB 需要进行特殊处理



行波进位的缺点

- 行波进位加法器（Ripple Carry Adder）
  - 加位将从最小位(LSB)传播到最大位(MSB)
  - N位加法器在最坏情况下的延迟: 2N个门延迟

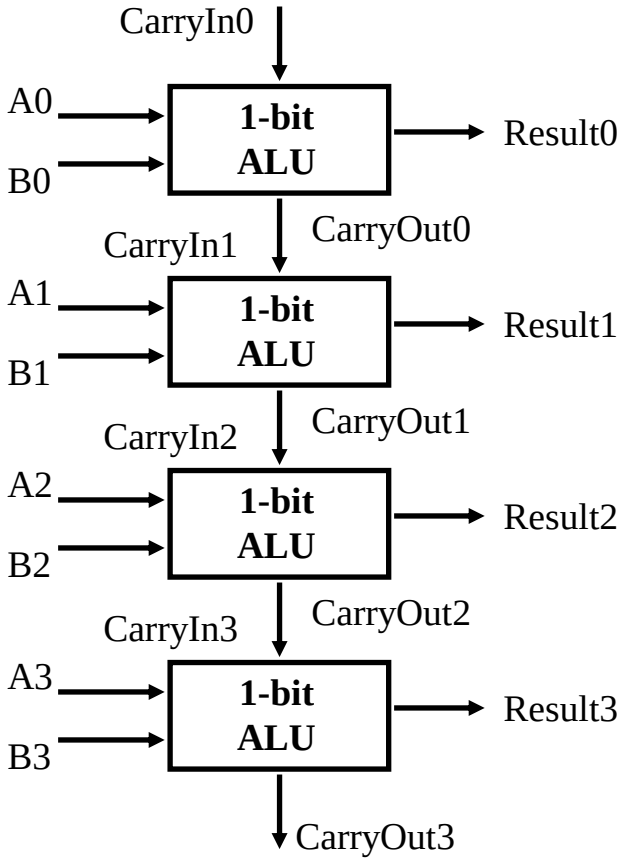




但是，性能如何呢？

- n位行波进位加法器的关键路径为 n×CP

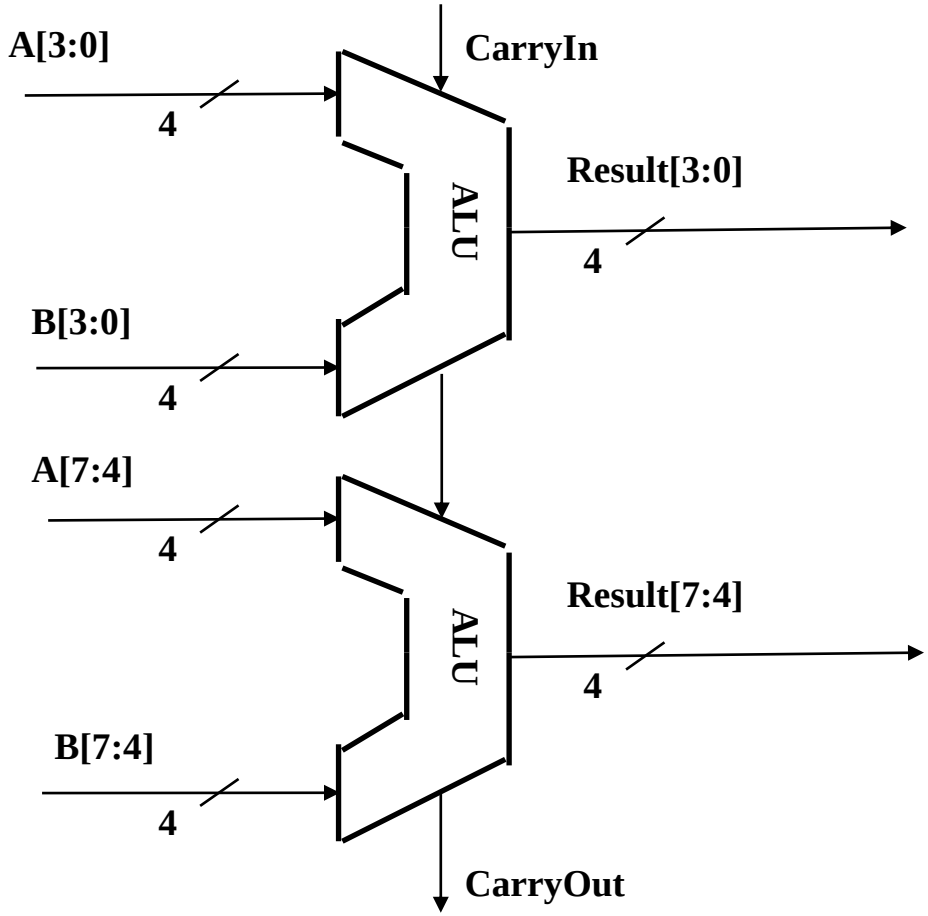
Carry Propagation



设计技巧: 减少关键路径上的硬件延迟

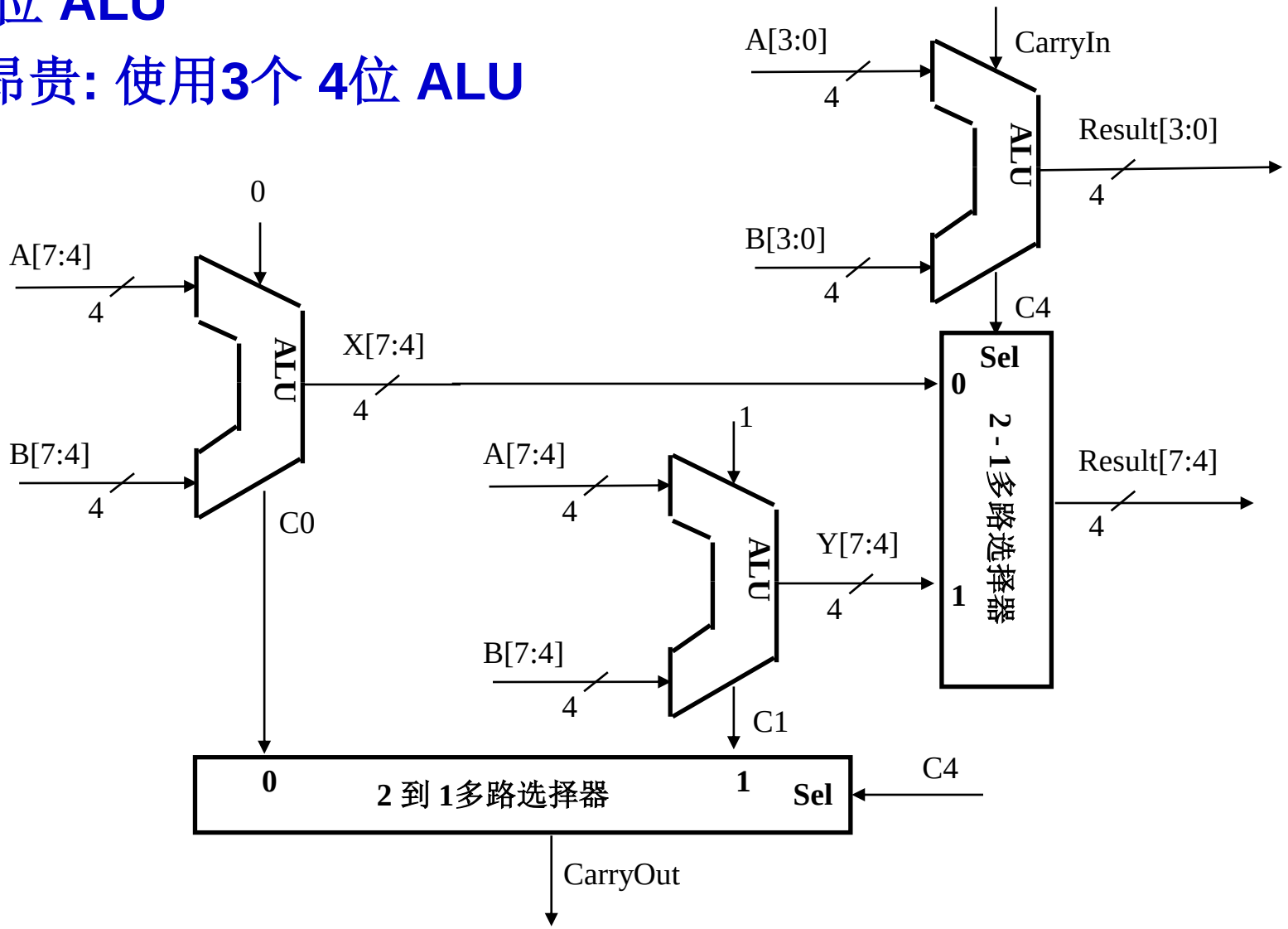
进位选择 (Carry Select Header)

- 设计一个 8位 ALU
  - 简单方案: 将两个4位ALU串联

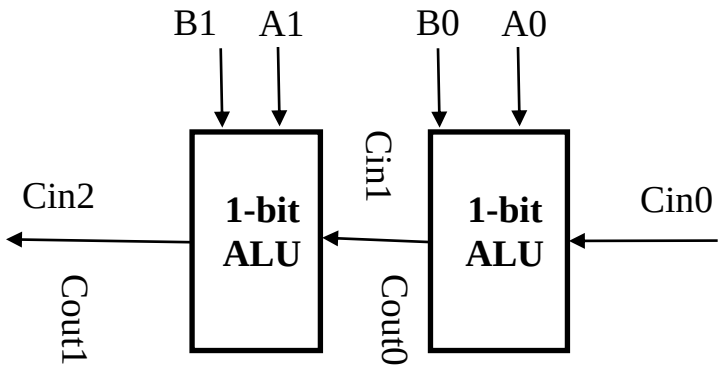


进位选择（续）

- 设计一个 8位 ALU
  - 快速、昂贵: 使用3个 4位 ALU



先行进位（Carry Lookahead）的理论基础



- 回忆: **CarryOut = (B & CarryIn) | (A & CarryIn) | (A & B)**
  - **Cin2 = Cout1 = (B1 & Cin1) | (A1 & Cin1) | (A1 & B1)**
  - **Cin1 = Cout0 = (B0 & Cin0) | (A0 & Cin0) | (A0 & B0)**
- **将Cin1 代入 Cin2:**
  - **Cin2 = (A1 & A0 & B0) | (A1 & A0 & Cin0) | (A1 & B0 & Cin0) | (B1 & A0 & B0) | (B1 & A0 & Cin0) | (B1 & A0 & Cin0) | (A1 & B1)**

Cin2 NOT depend of Cout0

定义两个新术语:

第i位产生的进位 **gi = Ai & Bi**

通过第i位的传播进位(Propagate Carry) **pi = Ai or Bi**

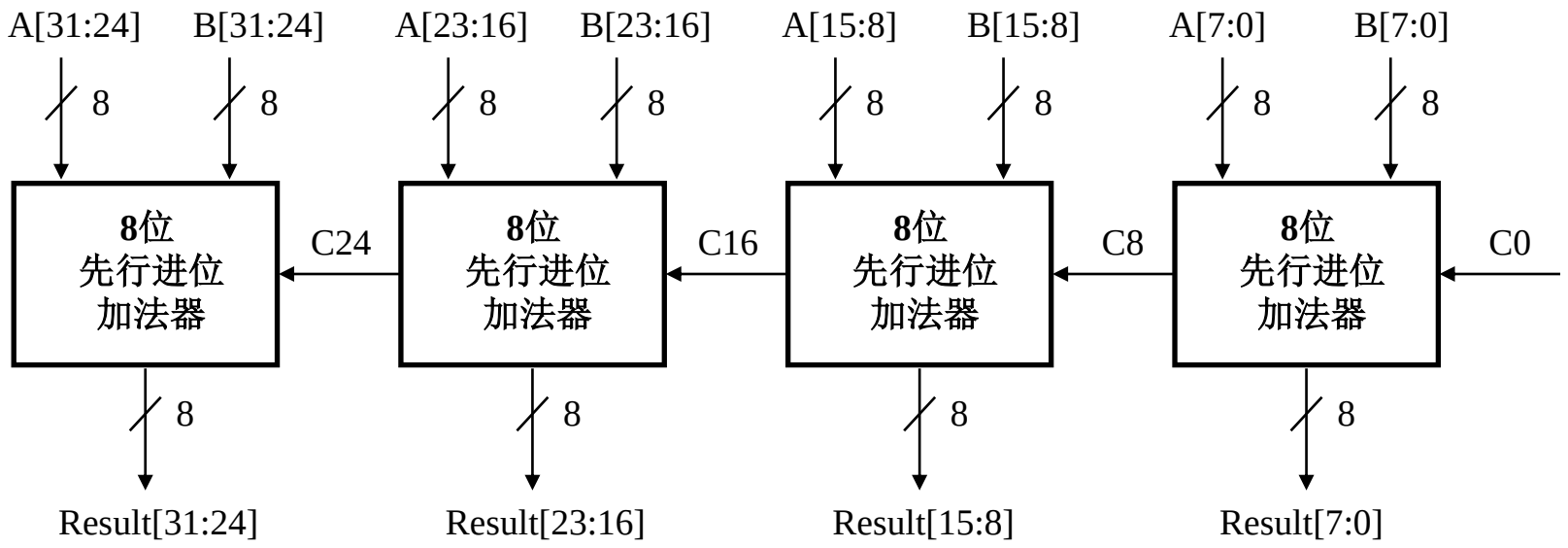
先行进位的理论基础（续）

- 使用新定义的两个术语:
  - 第i位产生的进位　　gi = Ai & Bi
  - 通过第i位的传播进位　　pi = Ai or Bi　　We can rewrite:
  - Cin1 = g0 | (p0 & Cin0)
  - Cin2 = g1 | (p1 & g0) | (p1 & p0 & Cin0)
  - Cin3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & Cin0)

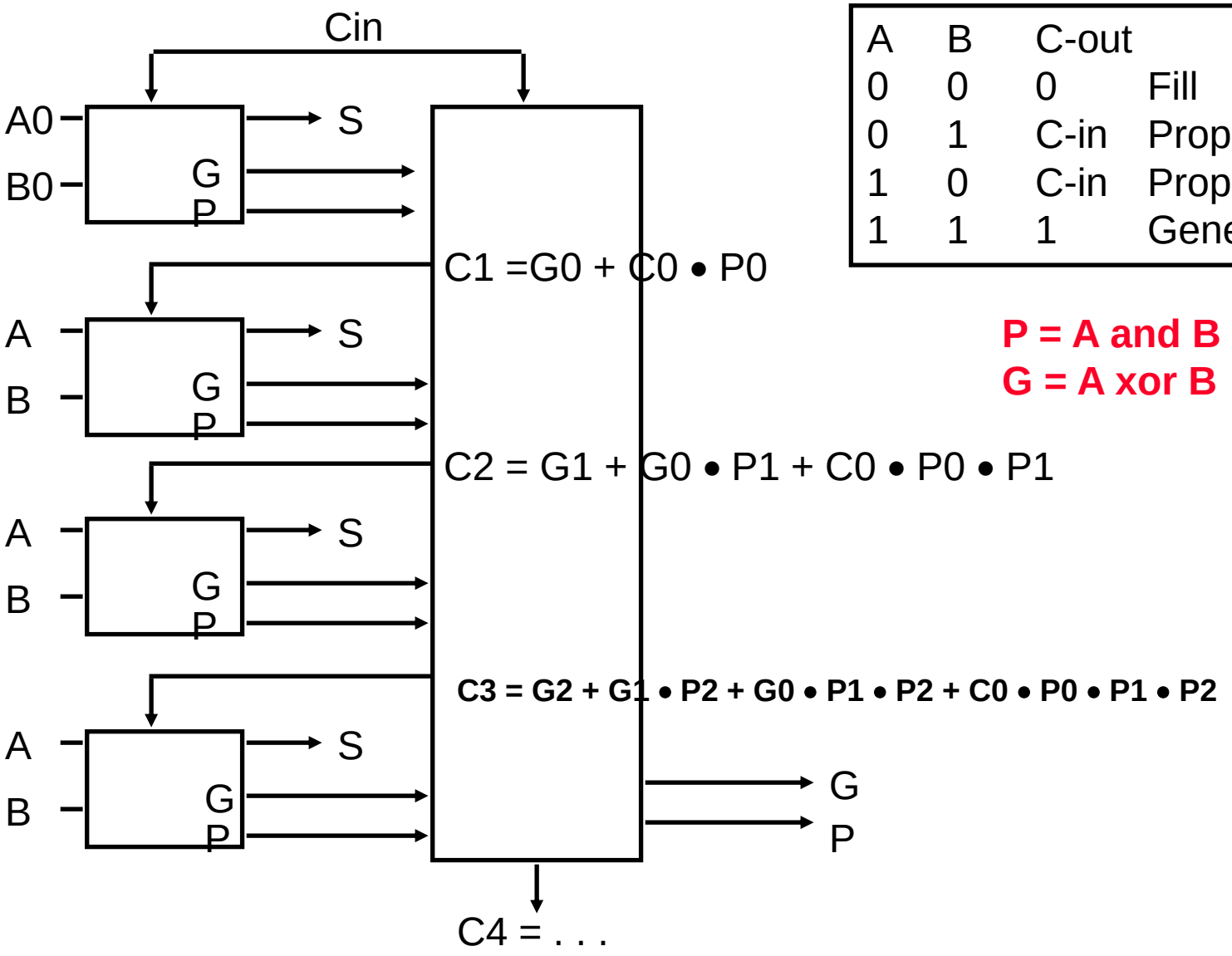
- 进入第3位的进位是1， 如果
  - 在第2位，产生了进位 (g2)
  - 或者 在第1位产生了进位 (g1)并且
    第2位传播了这个进位 (p2 & g1)
  - 或者 在第0位产生了进位 (g0)并且
    第1位和第2位都传播了这个进位 (p2 & p1 & g0)
  - 或者 在第0位有输入进位(Cin0) 并且
    第0位、第1位和第2位都传播了这个进位 (p2 & p1 & p0 & Cin0)

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

- 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



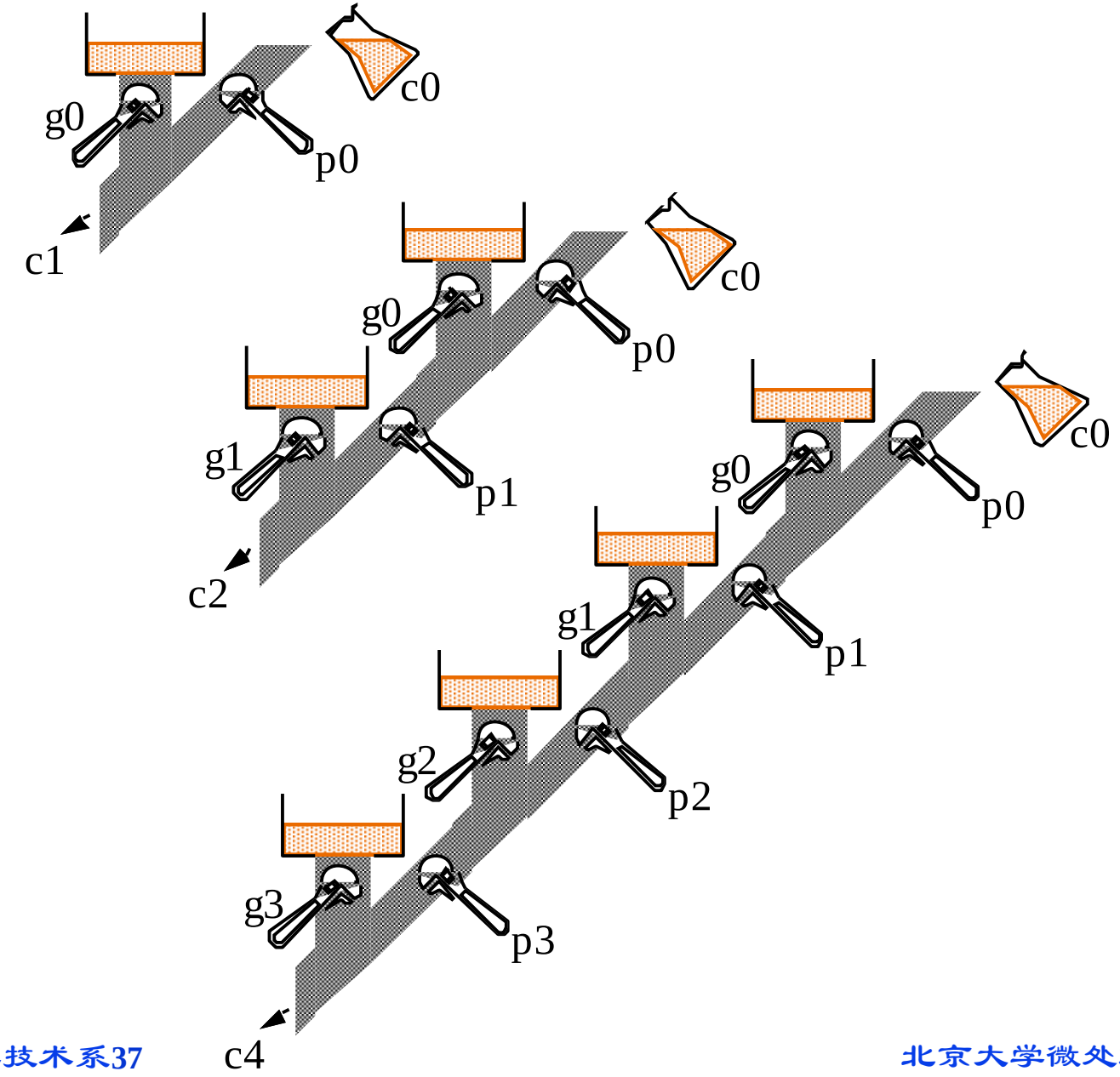
先行进位 Carry Look Ahead（设计技巧：窥探）



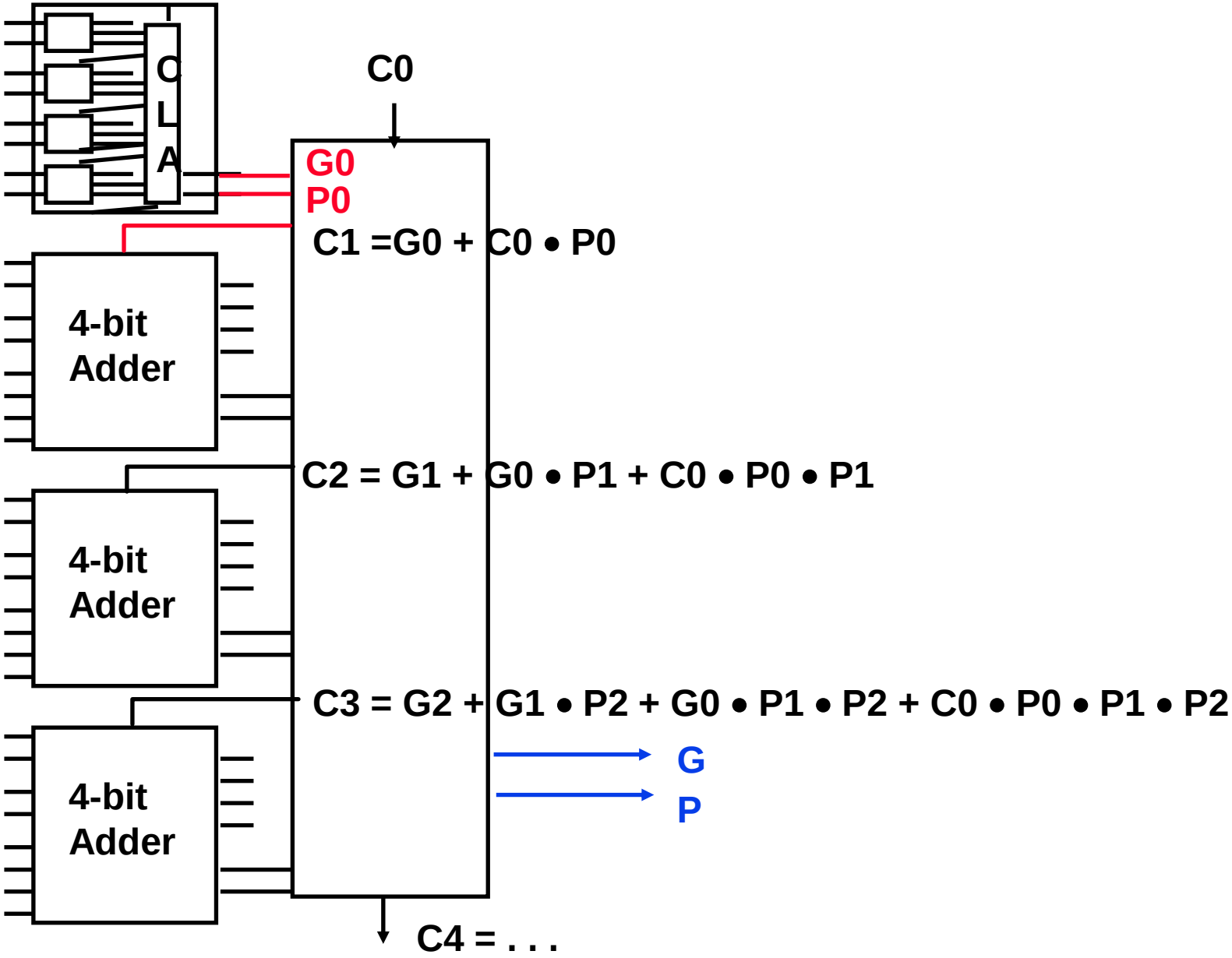
A	B	C-out	
0	0	0	Fill
0	1	C-in	Propagate
1	0	C-in	Propagate
1	1	1	Generate

P = A and B
G = A xor B

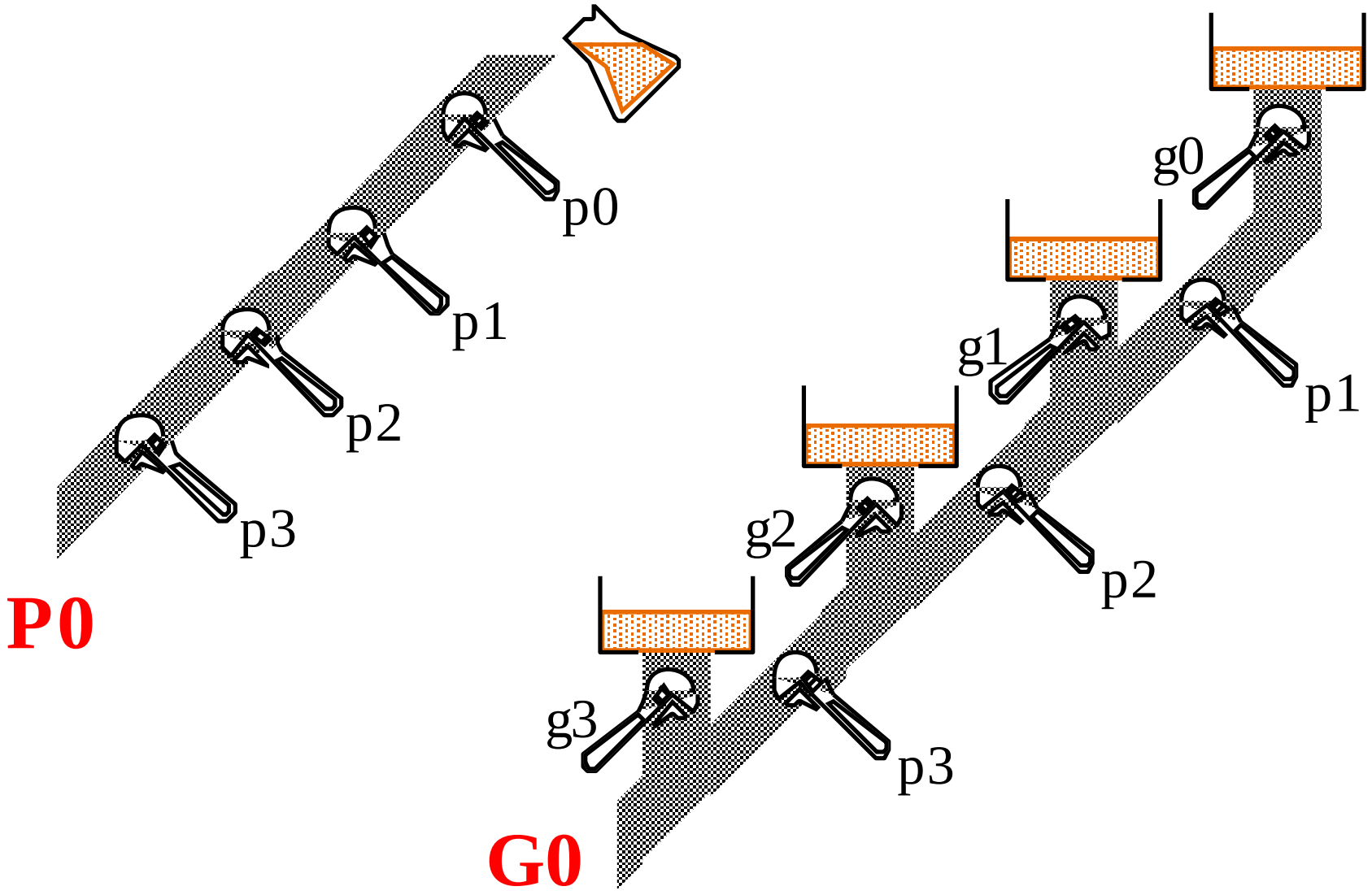
道管：先行进位的类比



层叠式先行进位（16位）：抽象

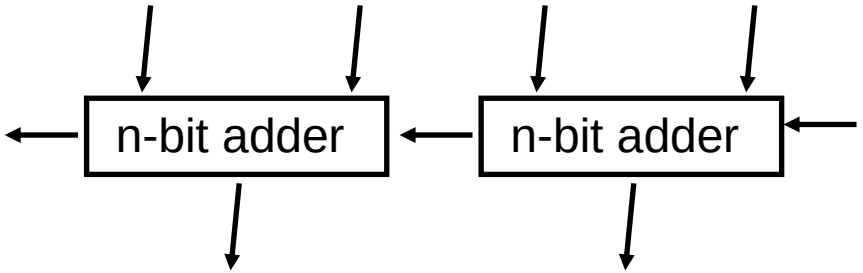


第二级进位、传播的管道类比

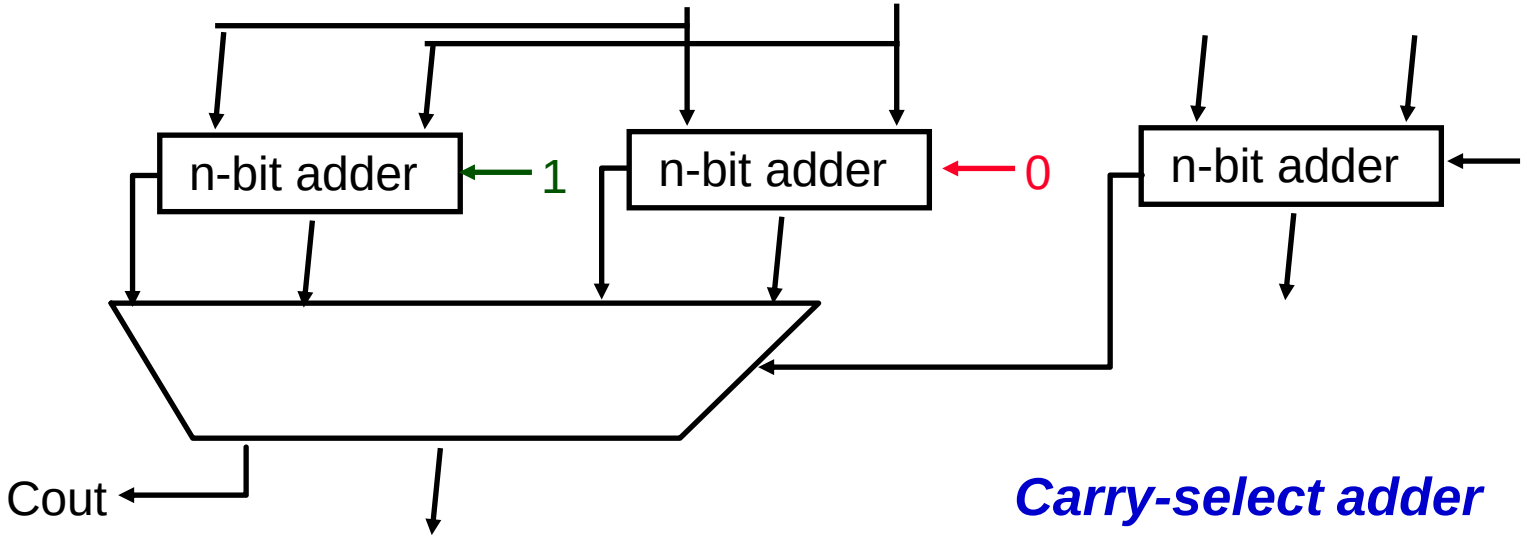


设计诀窍： 猜测

CP(2n) = 2*CP(n)



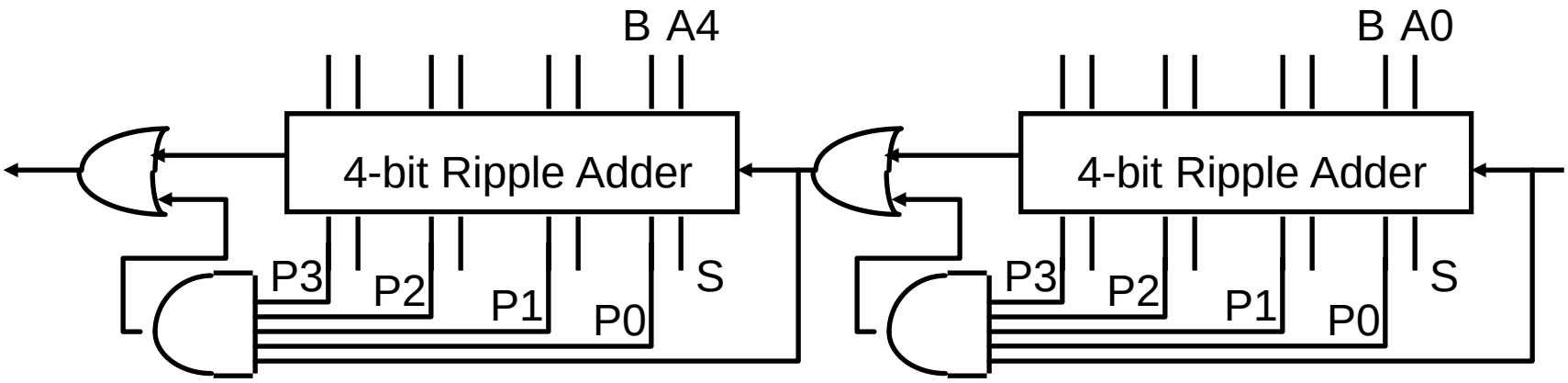
CP(2n) = CP(n) + CP(mux)



Carry-select adder

Assumes: mux is faster than adder

进位跳跃加法器 (Carry Skip Adder)：减少最坏情况的延迟



仅仅加速每个模块的最慢的情况

课后思考: 利用可变模块大小，来优化设计

设计过程的要素

- 分治 Divide and Conquer (e.g., ALU)
  - 针对较简单的部件，阐明解决方案.
  - 设计每个部件（子问题）
- 产生 并 测试 Generate and Test (e.g., ALU)
  - 给出一组积木，寻求如何将它们组装起来，满足需求
- 逐步求精 Successive Refinement (e.g., carry lookahead)
  - 解决“大多数”问题 (即, 忽视一些约束或特殊情况）, 检查并修改缺陷
- 阐明可供选择的高级方案 Formulate High-Level Alternatives (e.g., carry select)
  - 当追踪任何一种步骤时，都要最考虑多种策略
- 做已知如何做的事情 *Work on the Things you Know How to Do*
  - 在不断前进中，未知的事情将越来越明显。

设计过程小结

采用层次式设计处理复杂性

自顶向下 **vs.**自底向上 **vs.** 逐步求精

设计表达的重要性:

- 基本模块（**Block Diagrams**）
- 分解为位片（ **Bit Slices**）
- 真值表、 **K-Maps**
- 电路图
- 其他描述: 状态图、时序图, 寄存器传输, . . .

top down

bottom up

mux design meets at TT

优化标准:

门数 —— *面积*

[封装数] —— *面积*

管脚输出 —— *面积*

Logic Levels —— ***Delay*** ***Power***

Fan-in/Fan-out —— ***Delay*** ***Power***

Cost ***Design time***

ALU设计——乘法与移位

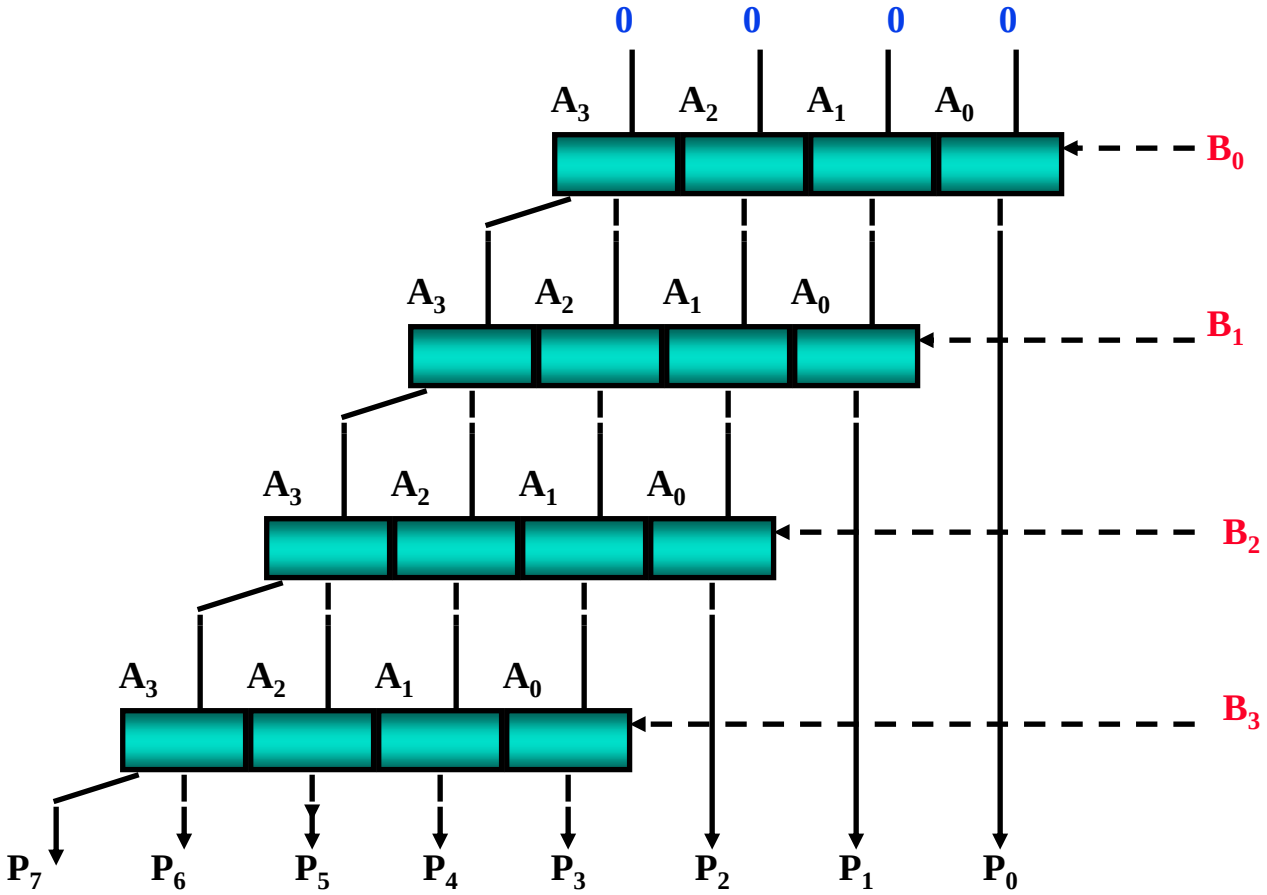
乘法（无符号数）

- 从纸和笔做乘法（无符号数）：

被乘数 Multiplicand		1000
乘数 Multiplier	x	1001
		1000
		0000
		0000
		1000
乘积 Product		1001000

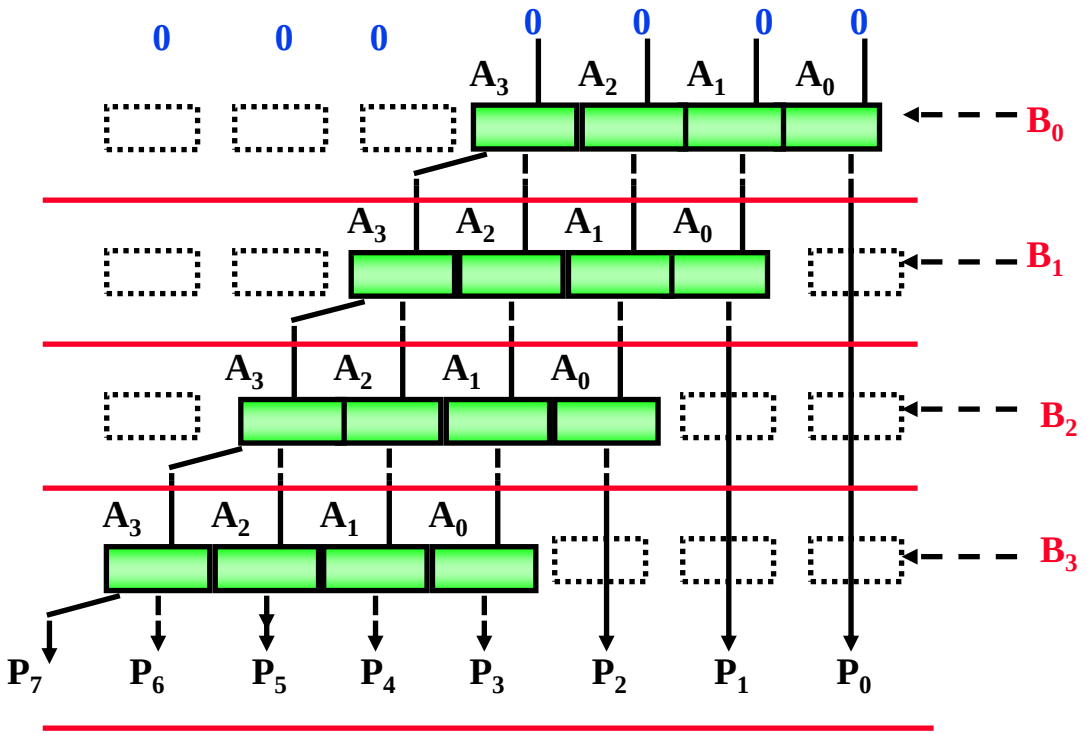
- 如果忽略符号位，m位 x n位 = m+n 位乘积
- 二进制，使得乘法更易实现：
  - 0 => 0 （ 0 x 被乘数）
  - 1 => 被乘数 （ 1 x 被乘数）
- 4 种 乘法硬件/算法
  - 循序渐进

无符号组合乘法器



- 如果$B_i == 1$，那么，第 i 级 累计 $A * 2^i$
- 问题: 32位乘法器需要多少硬件? 关键路径?

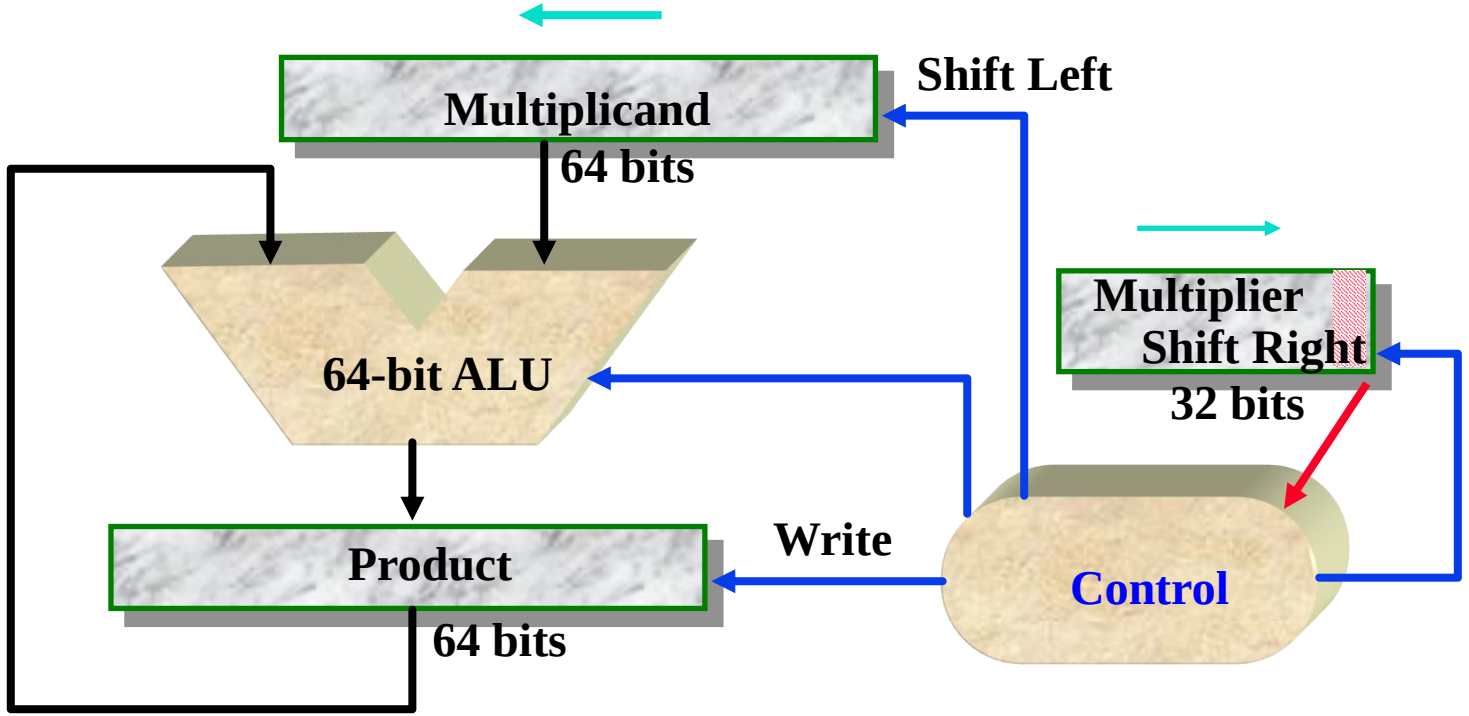
上述乘法器如何工作?



- 每级都对A进行左移 (x 2)
- 使用B的下一位来判断是否加上左移后的被乘数
- 每级都累计 2n 位的部分积

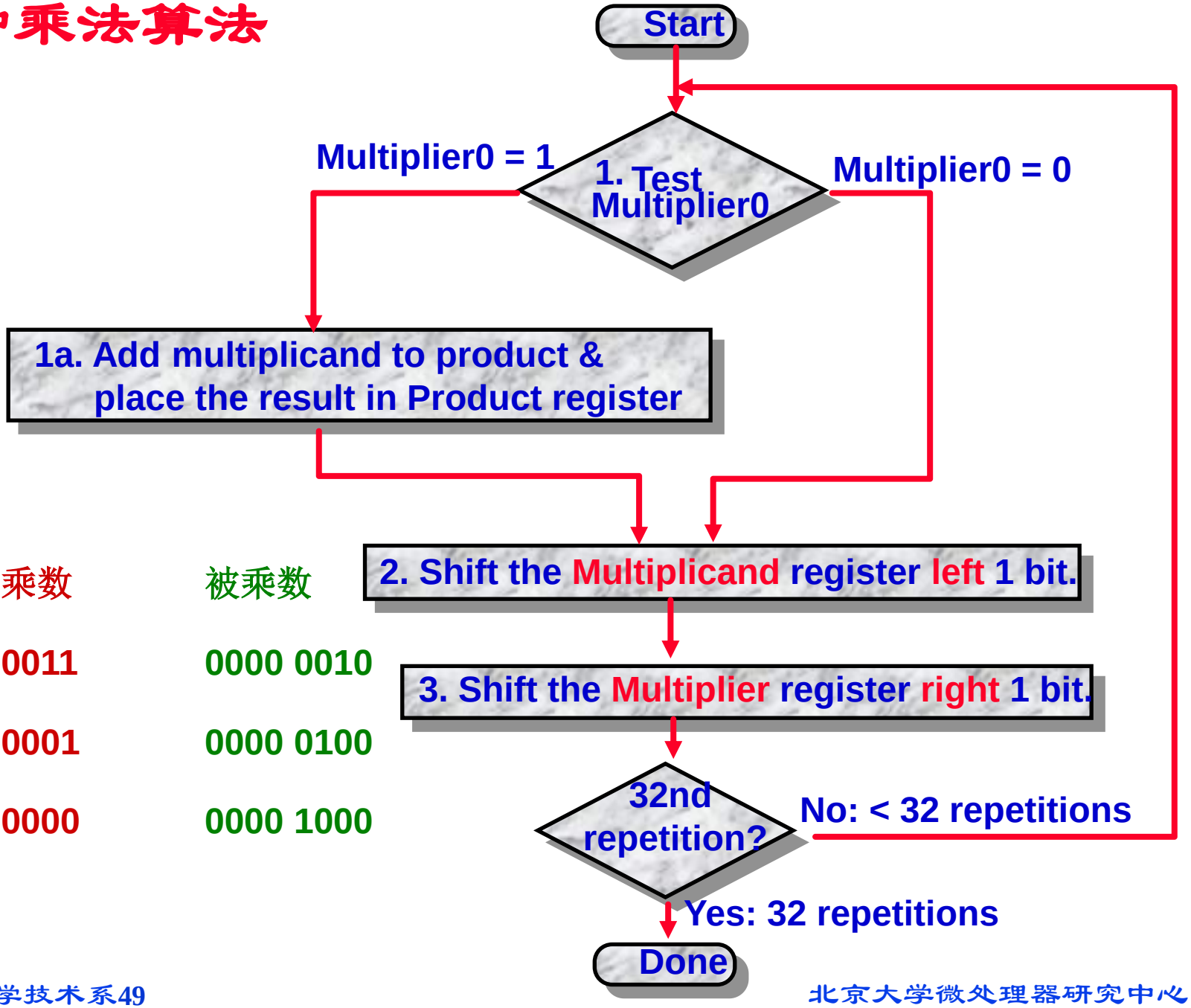
第一种无符号移位-加法乘法器

- 64位被乘数寄存器、64位ALU、64位乘积寄存器、32位乘数寄存器



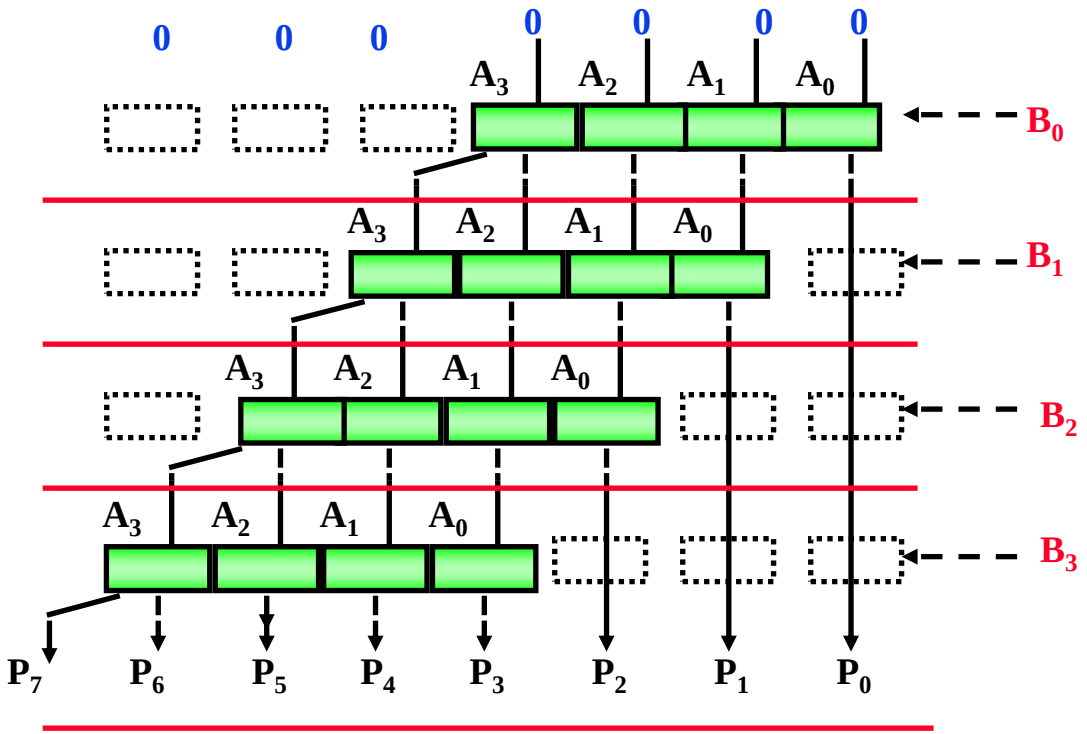
乘法器 = 数据通路 + 控制

第一种乘法算法



乘积	乘数	被乘数
0000 0000	0011	0000 0010
0000 0010	0001	0000 0100
0000 0110	0000	0000 1000
0000 0110		

上述乘法器如何工作?



- 每级都对A进行左移 (x 2)
- 使用B的下一位来判断是否加上左移后的被乘数
- 每级都累计 2n 位的部分积

第一种乘法的启示

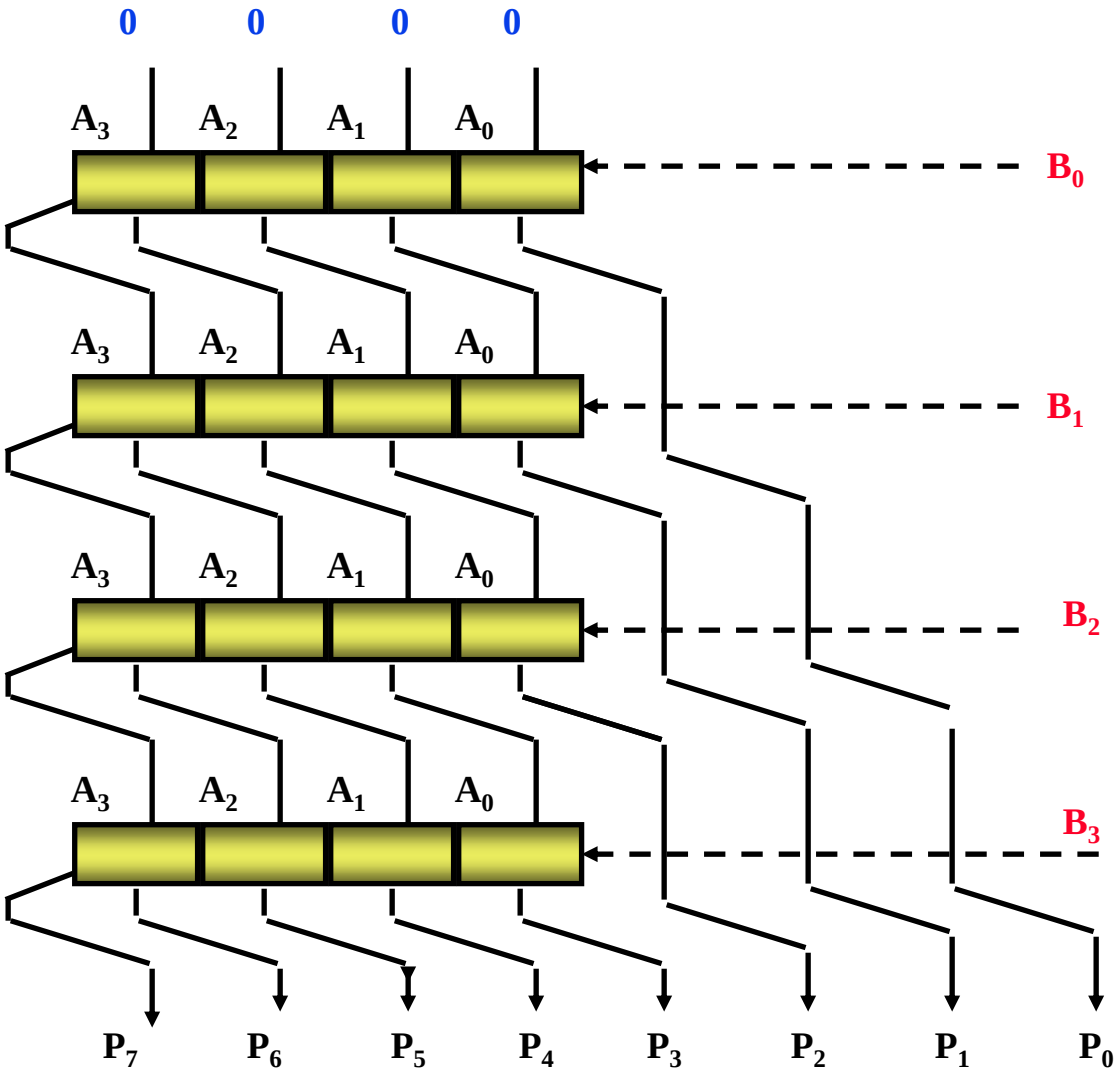
- 每个周期 1个时钟 => 每次乘法大约100（32×3）个时钟
  - 乘法与加法的出现频率比　5:1 ~ 100:1

- 64位被乘数中 1/2的位数　总是为　0
  => 使用64位加法器, 太浪费 !!!

- 在被乘数左移的过程中, 在左边插入 0
  => 一旦产生了乘积的最小位, 它就永远不变 !!!

- 用乘积右移　替代　被乘数左移?

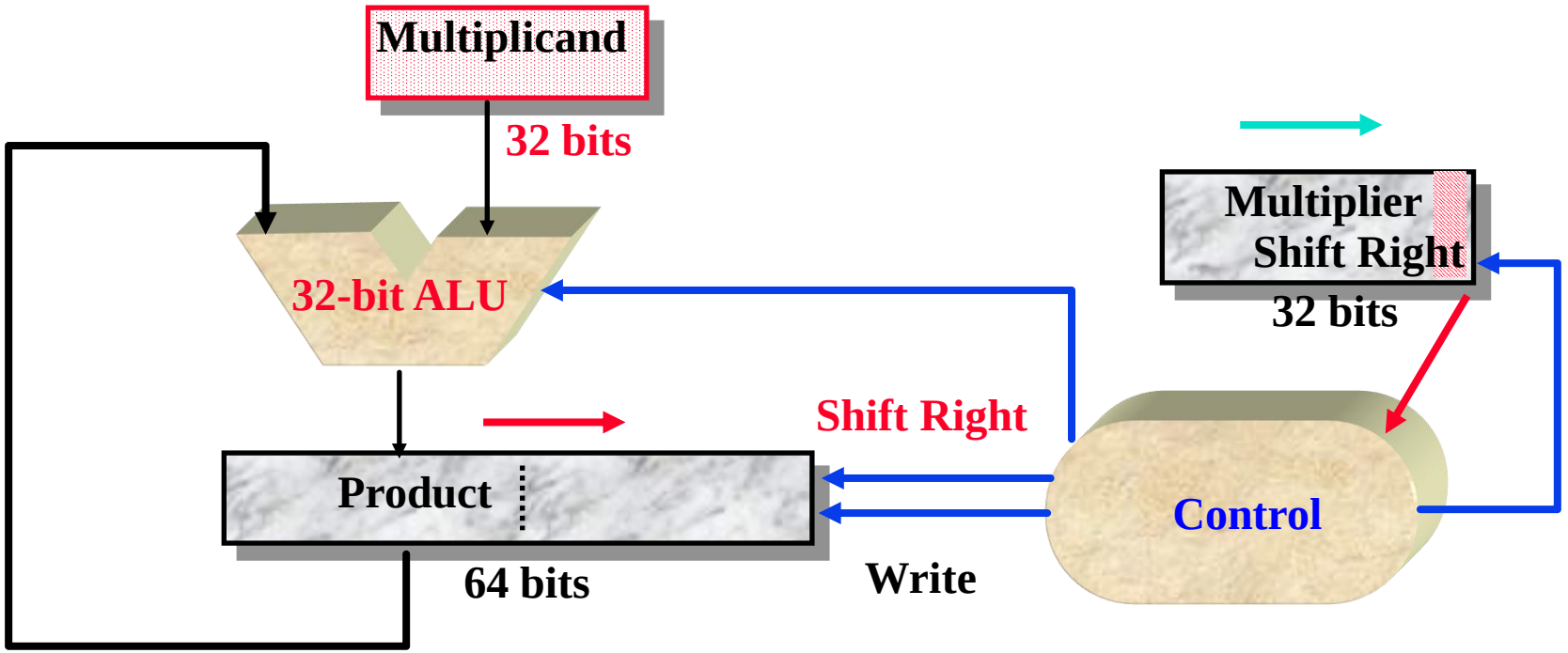
如何进一步改进？



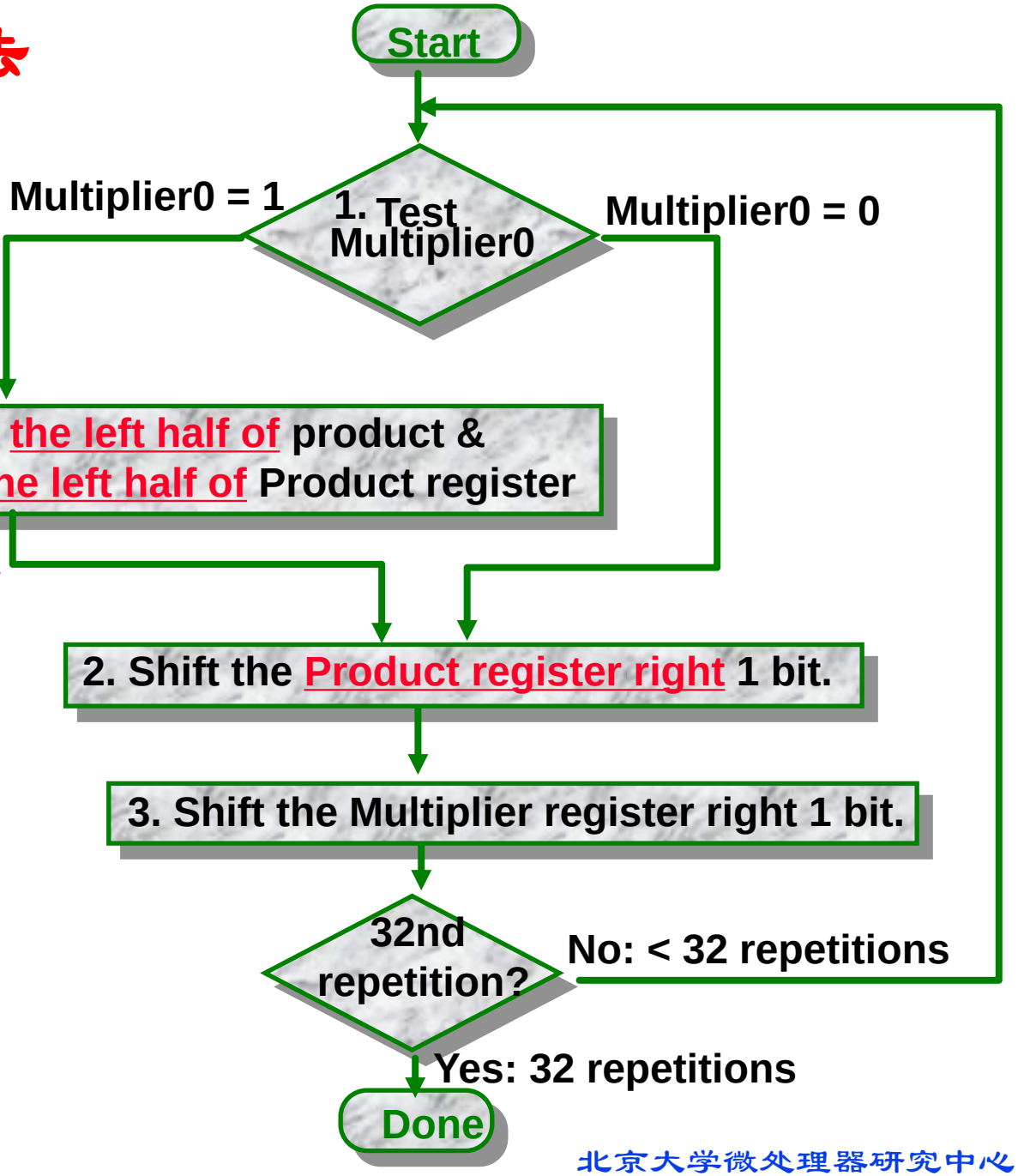
- 被乘数固定不动，乘积右移

第二种乘法硬件

- 32位被乘数寄存器、32位ALU、64位乘积寄存器、32位乘数寄存器

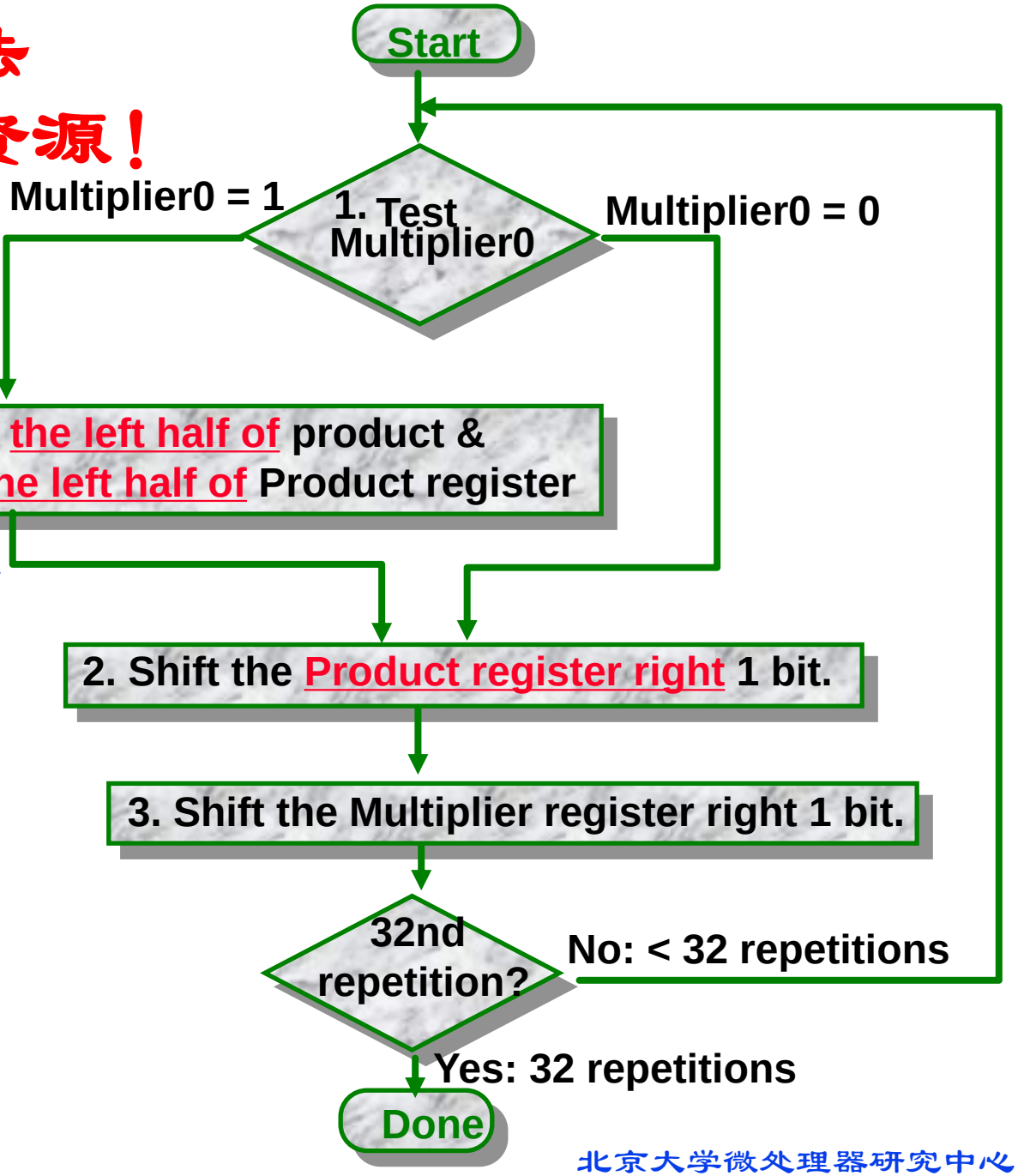


第二种乘法算法



	Product	Multiplier	Multiplicand
	0000 0000	0011	0010
1:	0010 0000	0011	0010
2:	0001 0000	0011	0010
3:	0001 0000	0001	0010
1:	0011 0000	0001	0010
2:	0001 1000	0001	0010
3:	0001 1000	0000	0010
1:	0001 1000	0000	0010
2:	0000 1100	0000	0010
3:	0000 1100	0000	0010
1:	0000 1100	0000	0010
2:	0000 0110	0000	0010
3:	0000 0110	0000	0010
	0000 0110	0000	0010

第二种乘法算法
仍然浪费很多资源！



	Product	Multiplier	Multiplicand
	0000	0000	0011	0010
1:	0010	0000	0011	0010
2:	0001	0000	0011	0010
3:	0001	0000	0001	0010
1:	0011	0000	0001	0010
2:	0001	1000	0001	0010
3:	0001	1000	0000	0010
1:	0001	1000	0000	0010
2:	0000	1100	0000	0010
3:	0000	1100	0000	0010
1:	0000	1100	0000	0010
2:	0000	0110	0000	0010
3:	0000	0110	0000	0010
	0000	0110	0000	0010

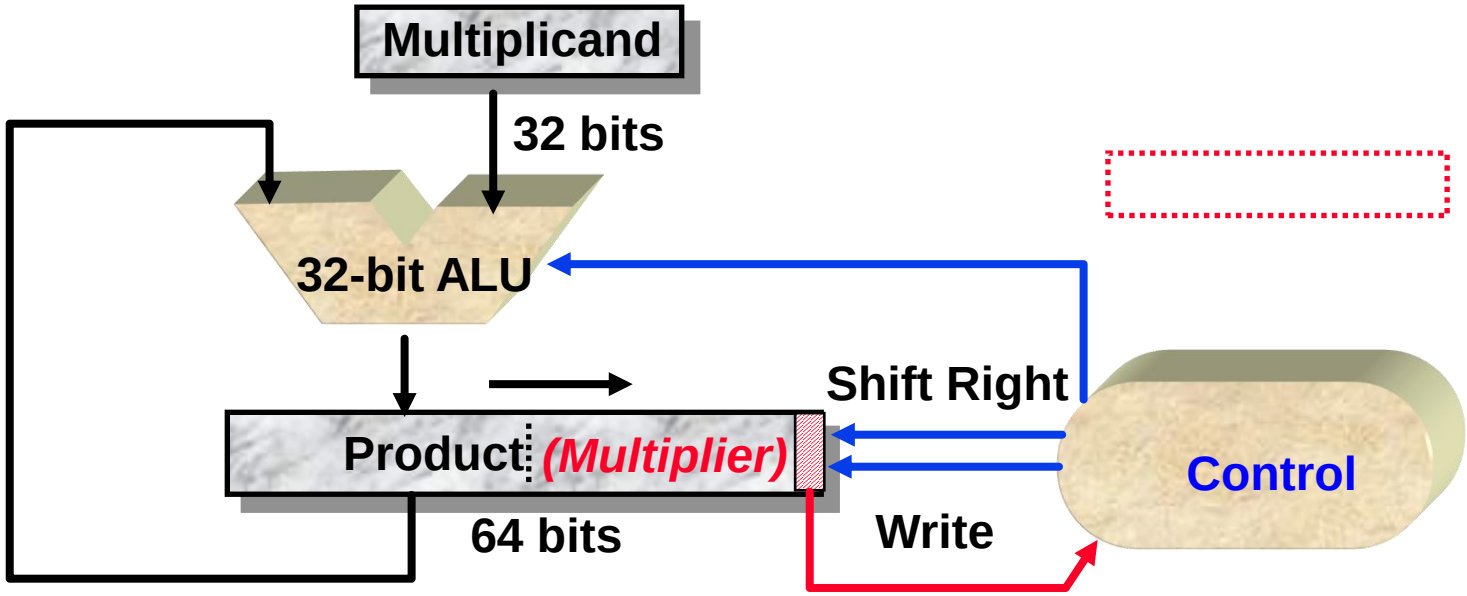
第二种乘法的启示

- 乘积寄存器浪费的空间 正好与 乘数中无用的空间 相同

=> 联合使用 乘数寄存器 和 乘积寄存器

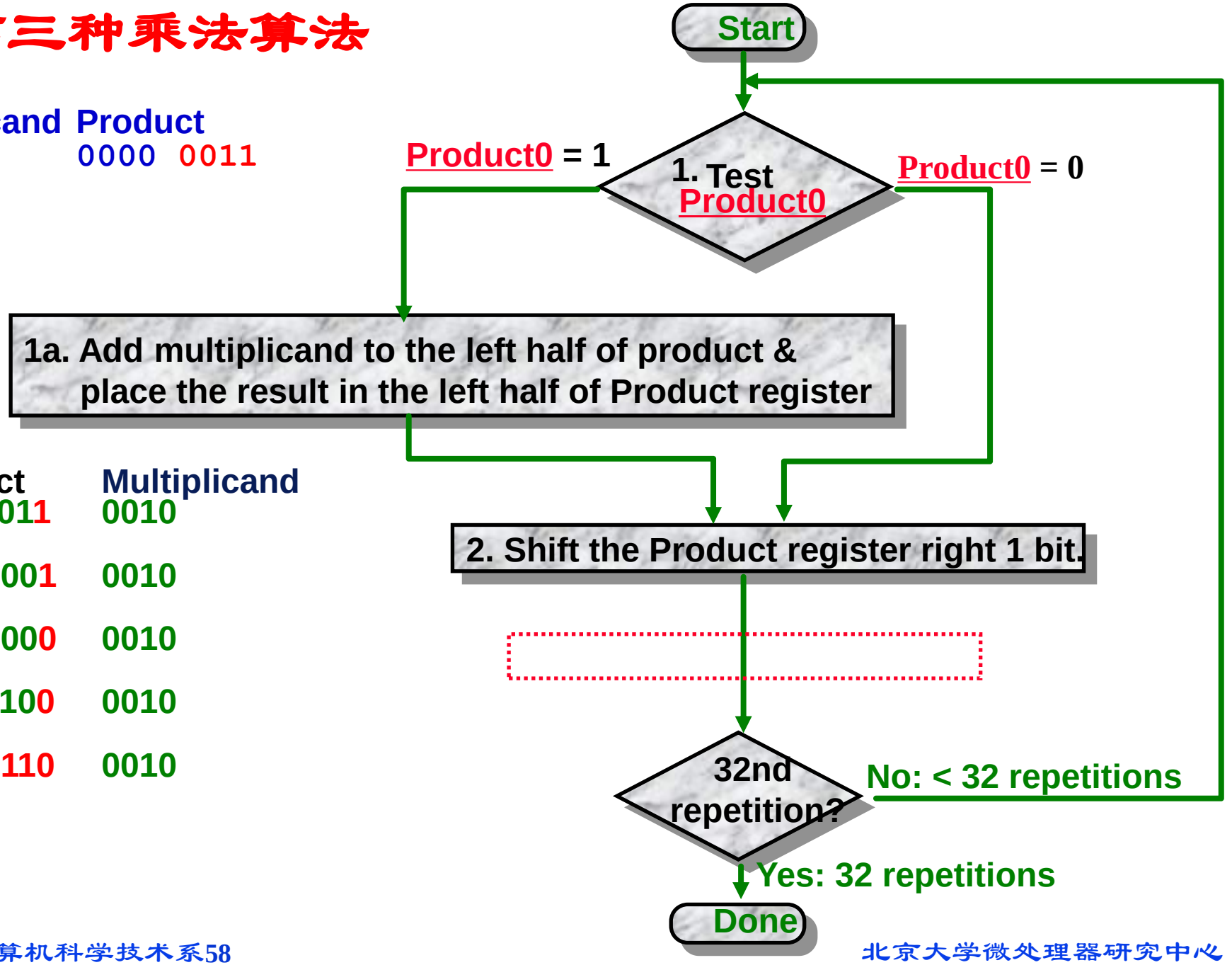
第三种乘法硬件

- 32位被乘数寄存器、32位ALU、64位乘积寄存器(没有乘数寄存器)



第三种乘法算法

Multiplicand Product
0010 0000 0011



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

第三种乘法的启示

- 由于乘数和乘积结合在一起，每位需要 2步
- MIPS中寄存器 Hi 和 Lo 是乘积的 左半部 和 右半部
- 提供了MIPS指令 MultU
- 如何进一步改进该算法的速度?
- 有符号数乘法如何?
  - 简单地策略是 假设两个源操作数都位正数，在运算结束时在对乘积进行修正（不算符号位，运算31步）
  - 使用补码
    - 需要对部分乘积进行符号扩展，在最后再进行减法
  - Booth算法是一种利用上述硬件进行有符号数乘法的 优秀算法
    - 可同时处理多位

Booth算法的动机

- 示例： 2 x 6 = 0010 x 0110:

```
          0010
   x      0110
  ------------
          0000     移位 (乘数运算位为 0)
   +      0010     加法 (乘数运算位为 1)
   +      0100     加法 (乘数运算位为 1)
   +      0000     移位 (乘数运算位为 0)
  ------------
      00001100
```

- 具有加法或减法的ALU可以采用多种方式计算相同的结果：

  6= -2 + 8 或者 0110 = -0010+ 1000

- 乘数中的 1串 可以用 看到第一个 1时，进行一次减法， 在处理完最后一个1后， 再加！

```
          0010
   x      0110
  ------------
          0000     移位 (乘数运算位为 0)
   -      0010     sub (乘数中的第一个 1)
   +      0000     shift (1串的中部)
   +      0010     add (最后1的前步)
  ------------
      00001100
```

Booth算法探奥

运行中部(middle of run)

运行尾部 (end of run)

运行首部 (beginning of run)

当前位	右边位	解释	示例
1	0	1串运行的首部	0001111000
1	1	1串运行的中部	0001111000
0	1	1串运行的尾部	0001111000
0	0	0串运行的中部	0001111000

早期, 由于移位比加法速度更快, 采用该算法主要为了速度

将乘数中间的1串 *用*

在第一次看见1时的一次初始减法 和 在最后一个1后的一次加法代替。

Booth算法

1. 依赖于当前和以前的位, 进行下述步骤之一:

   00: a. 0串的中部, 不进行任何运算操作.

   01: b. 1串的结束, 将被乘数加到乘积的左半部.

   10: c. 1串的首部, 从乘积的左半部减去被乘数.

   11: d. 1串的中部, 不进行任何运算操作.

2. 象第三种乘法算法一样, 将乘积算术右移1位.

被乘数	乘积 (2 x 3)	被乘数	乘积 (2 x -3)
0010	0000 0011 0	0010	0000 1101 0

Booth算法探奥

a_i	a_{i-1}	Operation
0	0	Do Nothing
0	1	Add b
1	0	Subtract b
1	1	Do Nothing

对于 $a_{i-1} - a_i$:

0: do nothing

+1: add b

- 1: subtract b

Booth算法探奥

$$(a_{-1}-a_0)\times b\times 2^0$$
$$+(a_0-a_1)\times b\times 2^1$$
$$+(a_1-a_2)\times b\times 2^2$$
$$+\ldots$$
$$+(a_{30}-a_{31})\times b\times 2^{31}$$

$$-a_i\times 2^i+a_i\times 2^{i+1}=(-a_i+2a_i)\times 2^i=(2a_i-a_i)\times 2^i=a_i\times 2^i$$

$$a_{-1}=0$$

$$b\times((a_{31}\times -2^{31})+(a_{30}\times 2^{30})+\ldots+(a_1\times 2^1)+(a_0\times 2^0)$$
$$=b\times a$$

Booths 示例 (2 x 7)

Operation	Multiplicand		Product	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

Booths 示例 (2 x -3)

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

三种移位器

逻辑(*logical*) — 移入的数值总是"0"

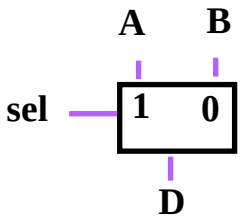
算术(*arithmetic*) — 在右移时，进行符号扩展

循环(*rotating*) — 移出的位又从另一端移入 (在MIPS中不支持)

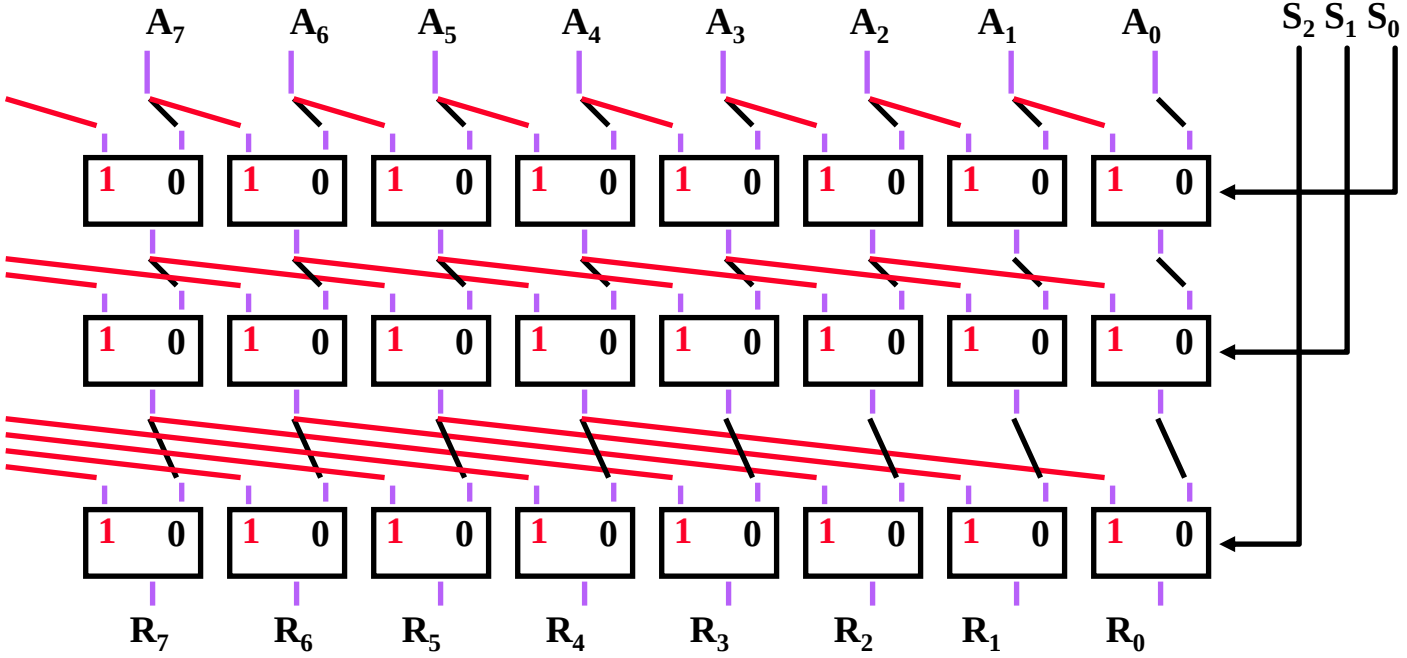
注：需要一位移动。特定指令可能要求进行0⇒32位移动！

使用多路选择器的组合移位器

基本单元



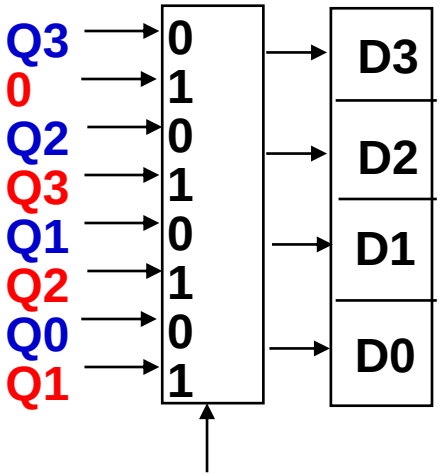
8位右移器



- MSBs的输入如何?
- 32位移位器需要多少级?
- 如果使用4-1多路选择器，结果如何?

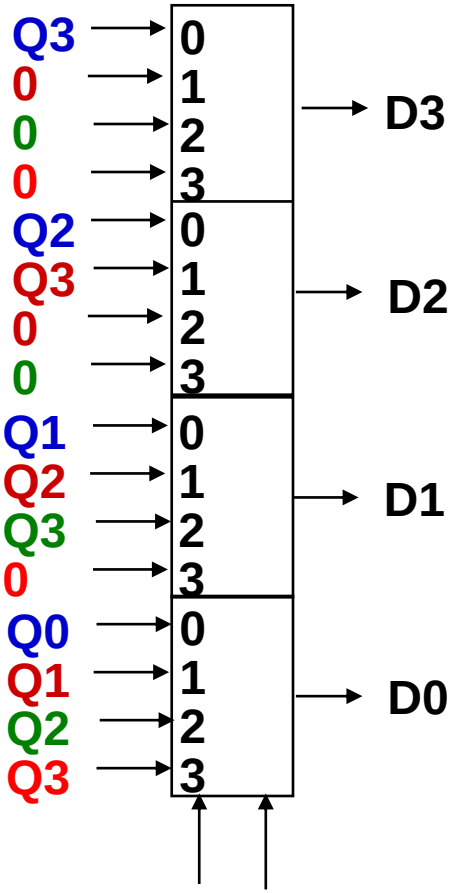
多路选择器/移位器

SHR:

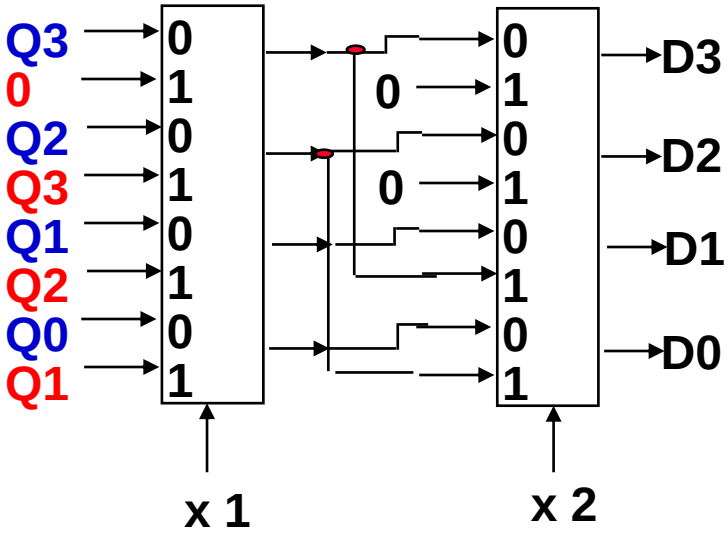


SHR/
don't
shift
(5 inputs)

SHR 0, 1, 2, 3 bits:



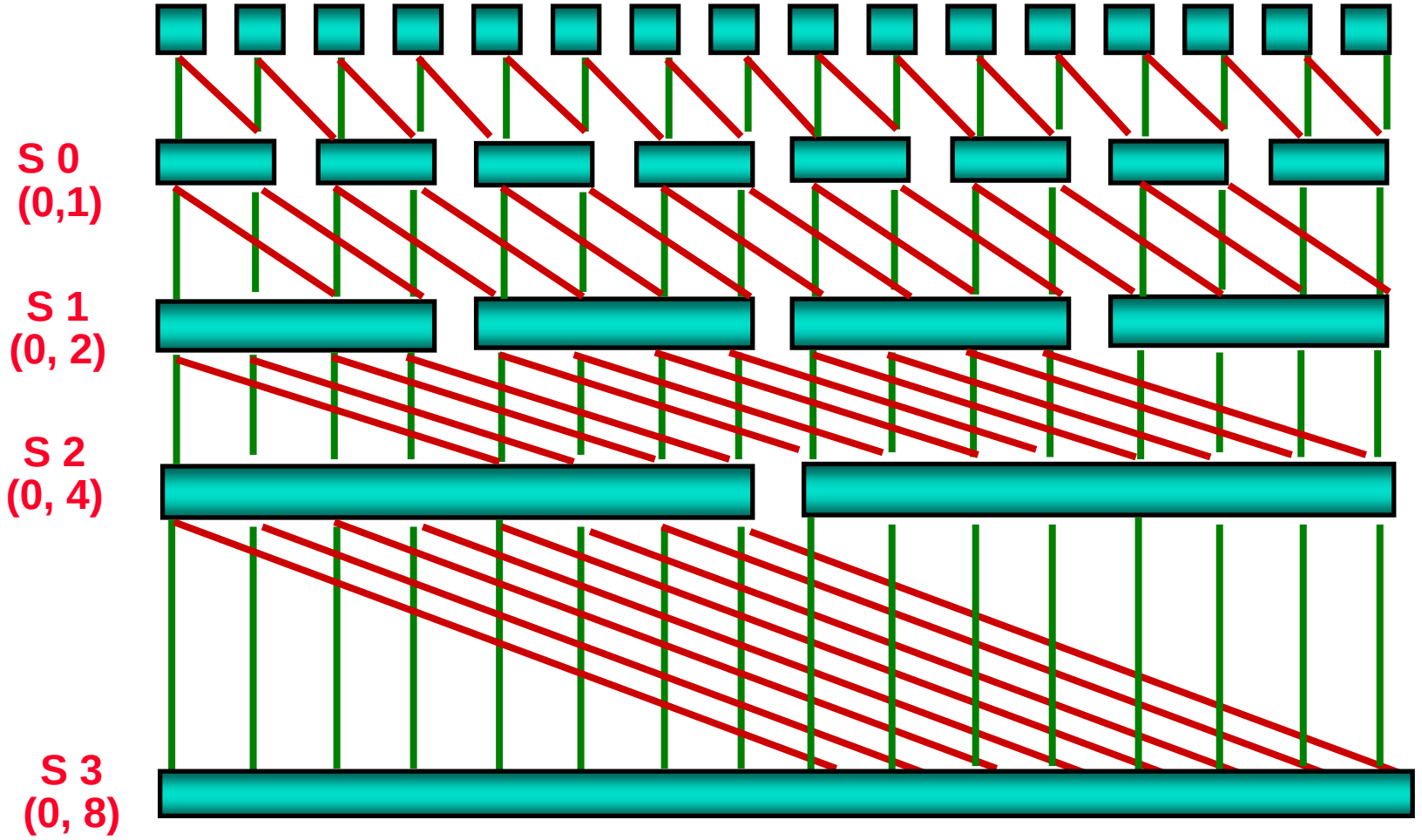
4 x 4:1 Mux
1 stage



8 x 2:1 Mux
2 stages

算术右移如何做?

通用右移策略（16位）



如果增加自右至左的连接，就可以支持循环移位

32 位移位器

使用上述配置对32位数据进行0~31位移位将导致:

如果使用2:1多路选择器，那么　需要5级多路选择器 (x1, x2, x4, x8, x16)

32位 x 5级 = 160个 2:1多路选择器!

如果使用2级 4:1多路选择器和1级2:1多路选择器, 那么 需要3级多路选择器

32 x 4:1 + 32 x 4:1 + 32 x 2:1

如果使用1级8:1多路选择器和1级4:1多路选择器, 那么　需要2级多路选择器

32 x 8:1 + 32 x 4:1

多位移位

复合策略，每次循环完成多位的多重控制循环

31位移位：

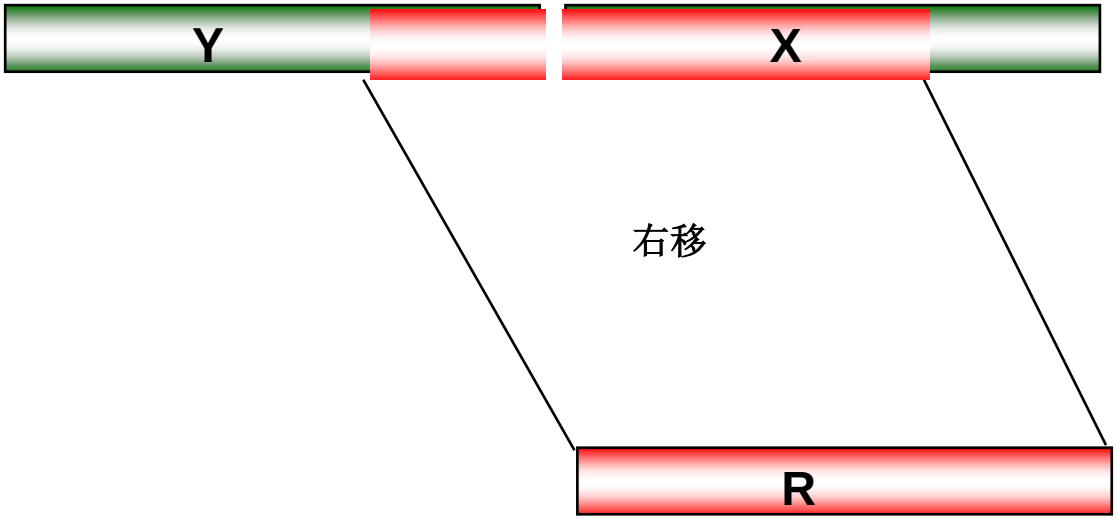
使用0、1位移位器	31次迭代
使用0、1、2、3位移位器	11次迭代
使用0、1、2、3、4、5、6、7位移位器	5次迭代
使用0⇒15位移位器	3次迭代

实际上，大多数移位都比较短（通常，实现0-3）

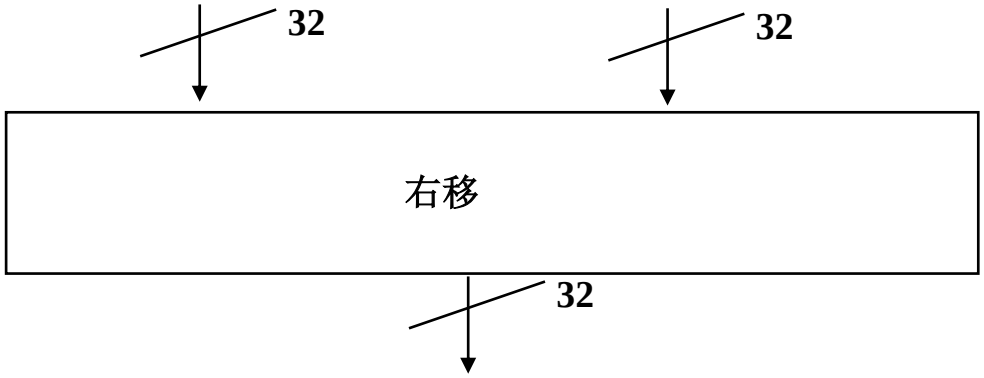
非常复杂： 目前，只能进行右移

漏斗移位器 (Funnel Shifter)

代替 从64位中抽取32位.

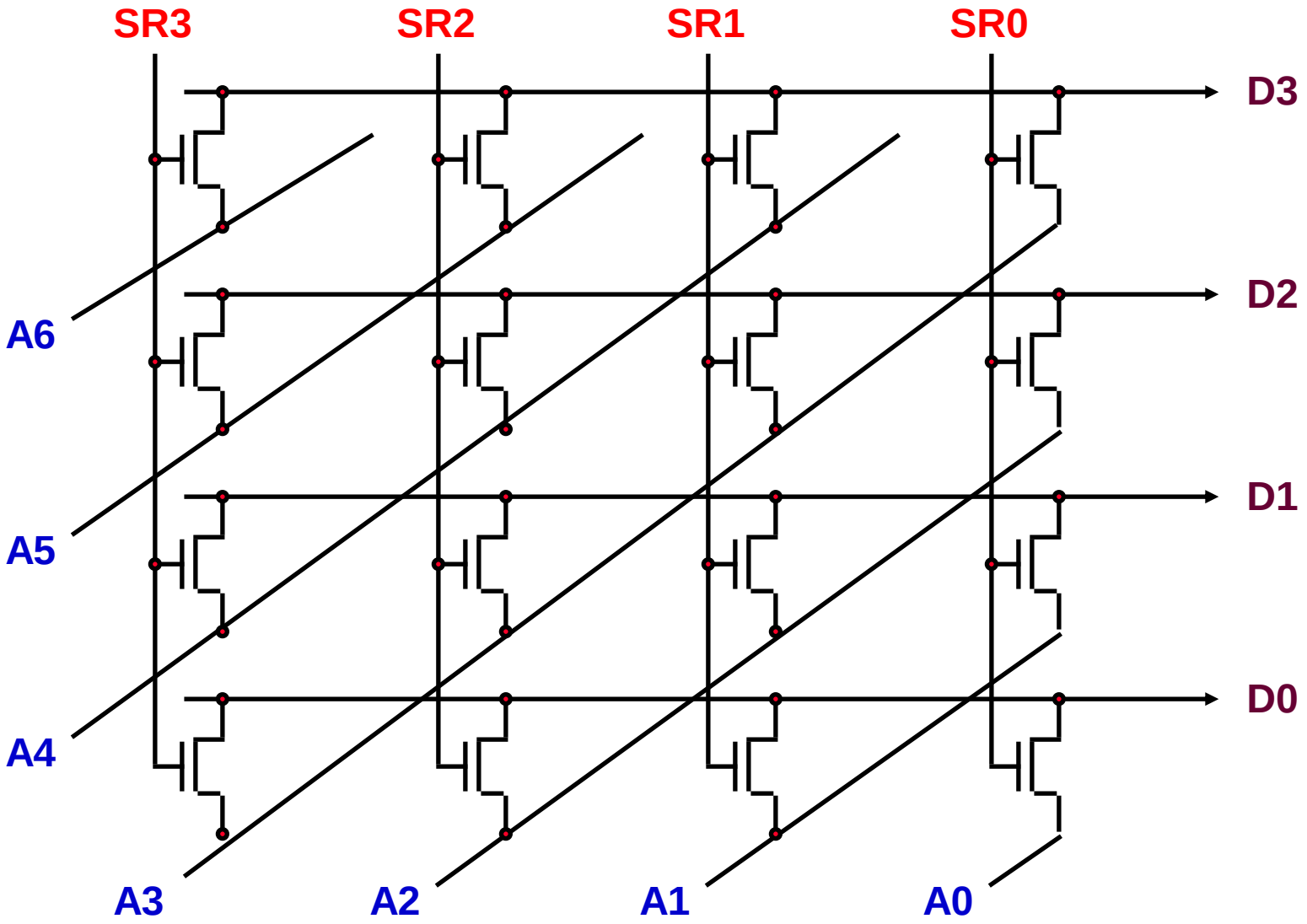


- A右移 i 位 （ sa = shift right amount ）
- 逻辑: Y = 0, X=A, sa=i
- 算术: Y = sign(A), X=A
- 循环: Y=A, X=A
- 左移: Y=A, X=0, sa= 32 - i



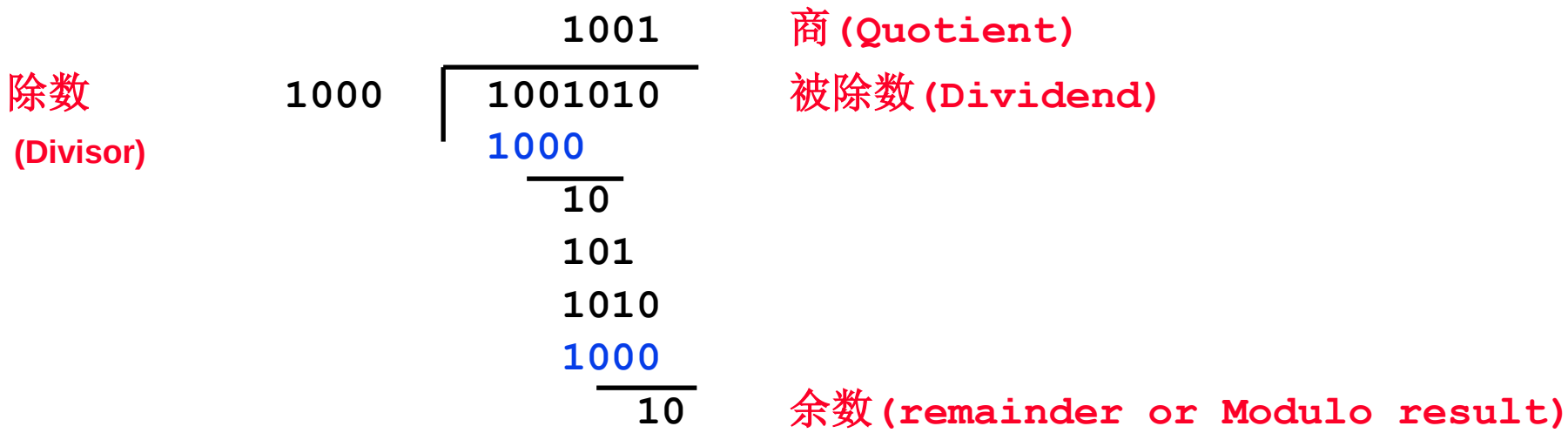
桶式移位器 (Barrel Shifter)

依赖于技术工艺的解决方案: 每个开关一个晶体管



除法、浮点数

除法：手算过程



查看可以减去多大的数，每步产生一位商

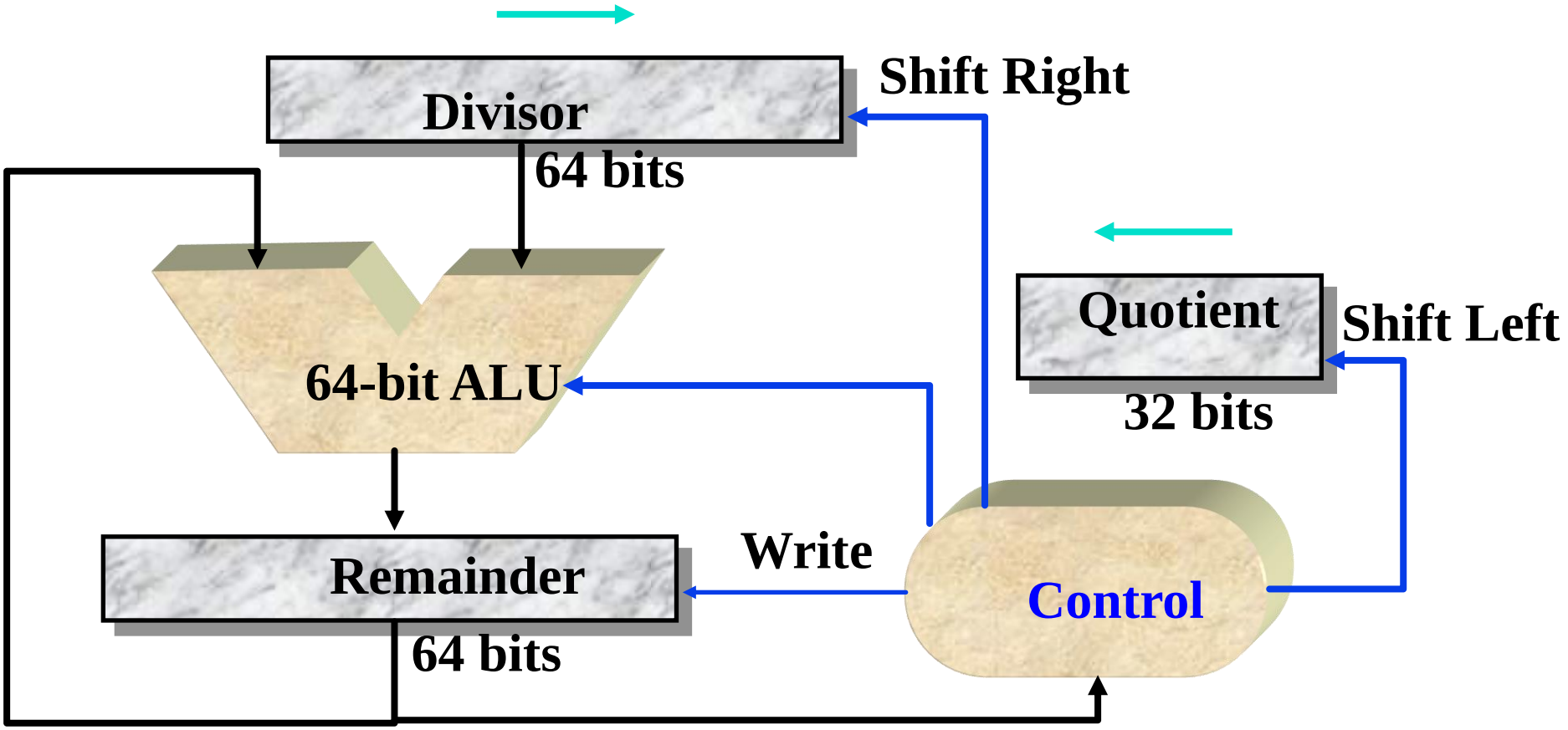
- 二进制 => 1 × 除数 or 0 × 除数

◦ 被除数 = 商 x 除数 + 余数
=> | 被除数 | = | 商 | + | 除数 |

◦ 循序渐进的三个除法算法

第一种除法算法

- 64位除数寄存器、64位ALU、64位余数寄存器、32位商寄存器



第一种除法算法 对 n位 商和余数，需要n+1步.

7/2

余数	商	除数
00000111	0000	00100000

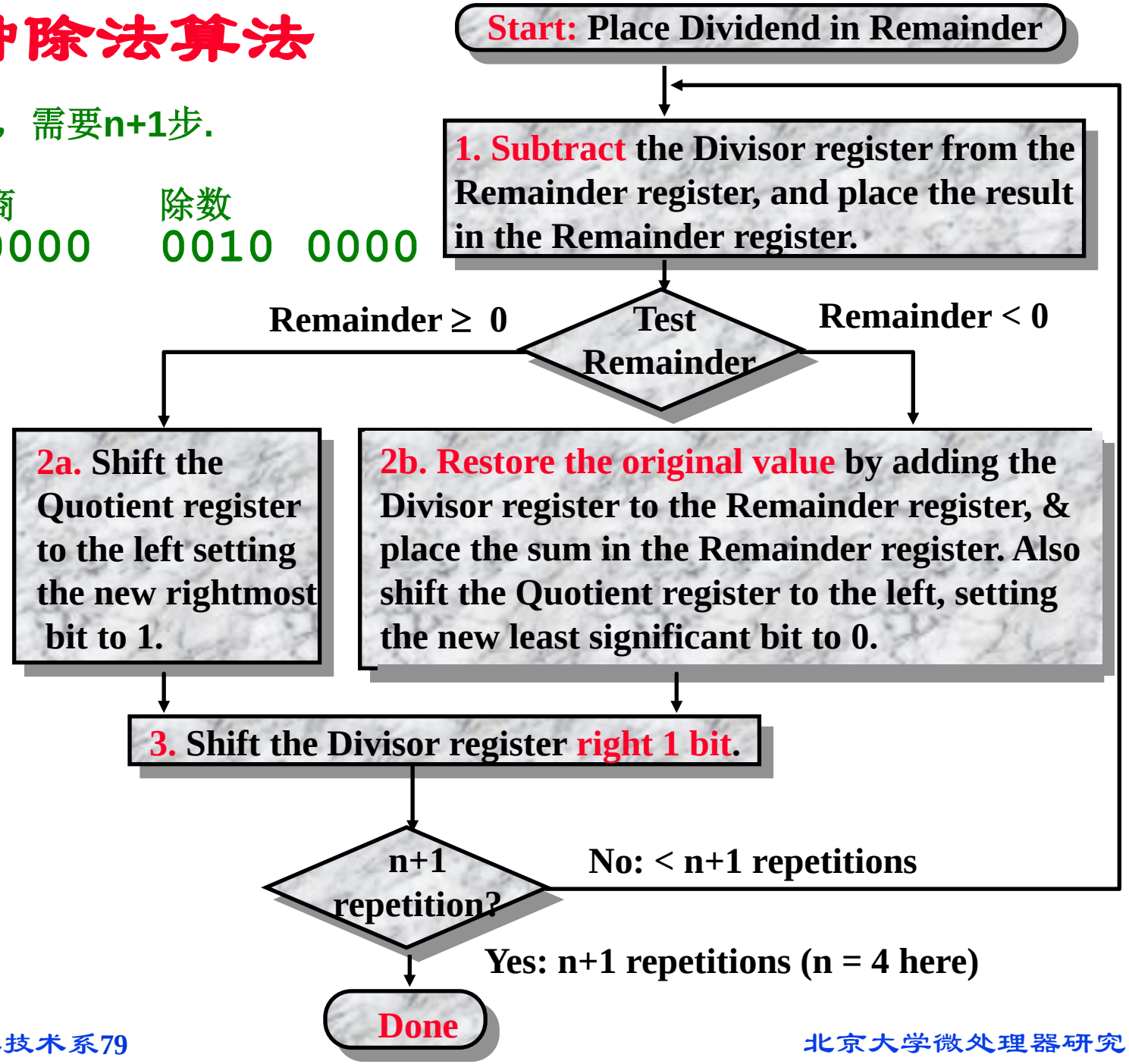
	Q: 0000	D: 0010 0000	R: 0000 0111 -D = 1110 0000
1: R = R-D	Q: 0000	D: 0010 0000	R: 1110 0111
2b: +D, sl Q, 0	Q: 0000	D: 0010 0000	R: 0000 0111
3: Shr D	Q: 0000	D: 0001 0000	R: 0000 0111 -D = 1111 0000
1: R = R-D	Q: 0000	D: 0001 0000	R: 1111 0111
2b: +D, sl Q, 0	Q: 0000	D: 0001 0000	R: 0000 0111
3: Shr 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: 0000	D: 0000 0010	R: 0000 0011 -D = 1111 1110
1: R = R-D	Q: 0000	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

第一种除法算法

- 对 n位 商和余数，需要n+1步.

余数 商 除数

```
0000 0111 0000   0010 0000
```



第一种除法的启示

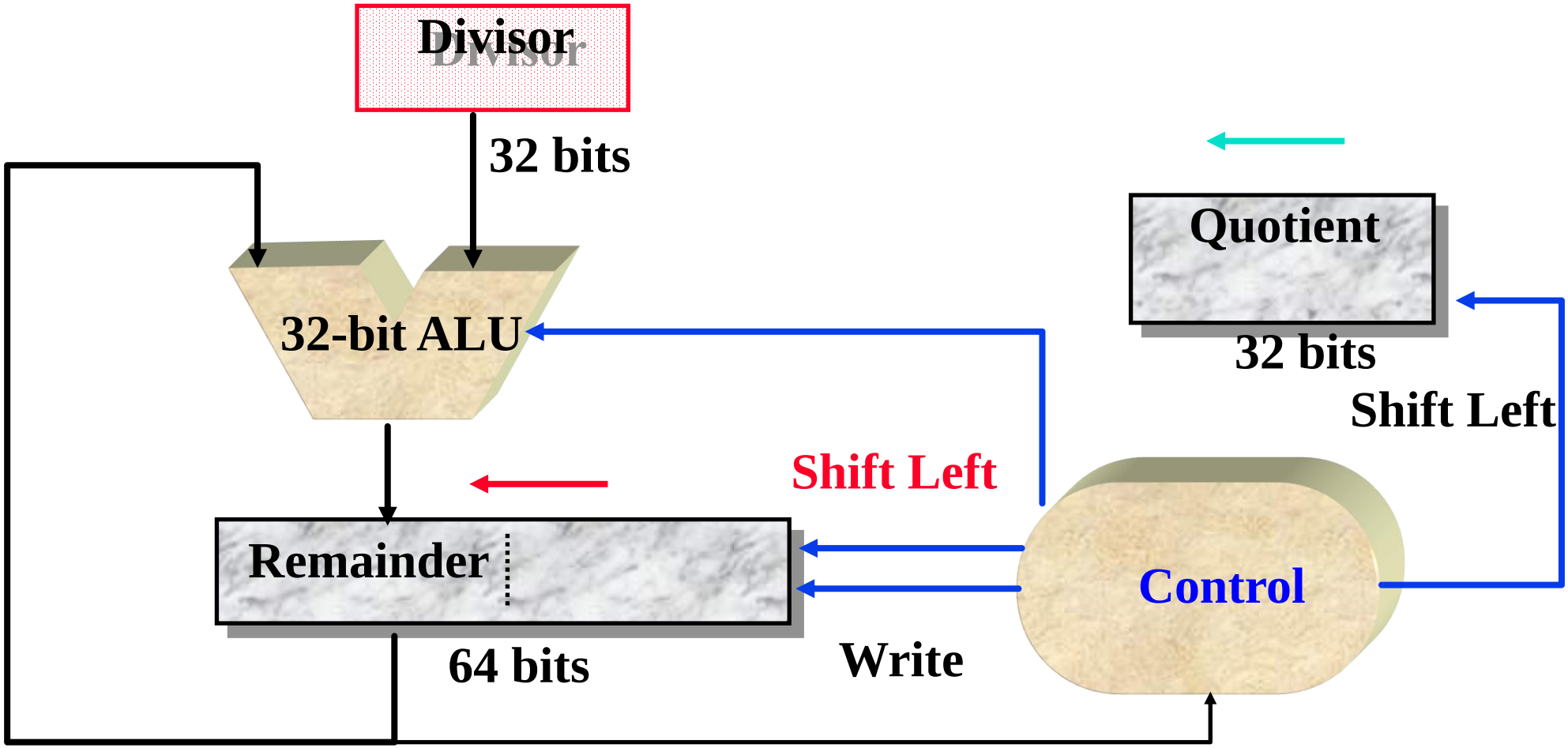
- 除数中 1/2的位数 总是为 0
  => 64位加法器的1/2浪费 !!!

  =>1/2的除数浪费 !!!
- 是否可以用 余数左移 替代 除数右移？
- 第一步不能在商中产生1(由于开始时余数的高32位为0，或者被除数太大)

  => 变换次序 到 首先移位 然后 再减, 可以减少一次迭代

第二种除法算法

- 32位除数寄存器、32位ALU、64位余数寄存器、32位商寄存器



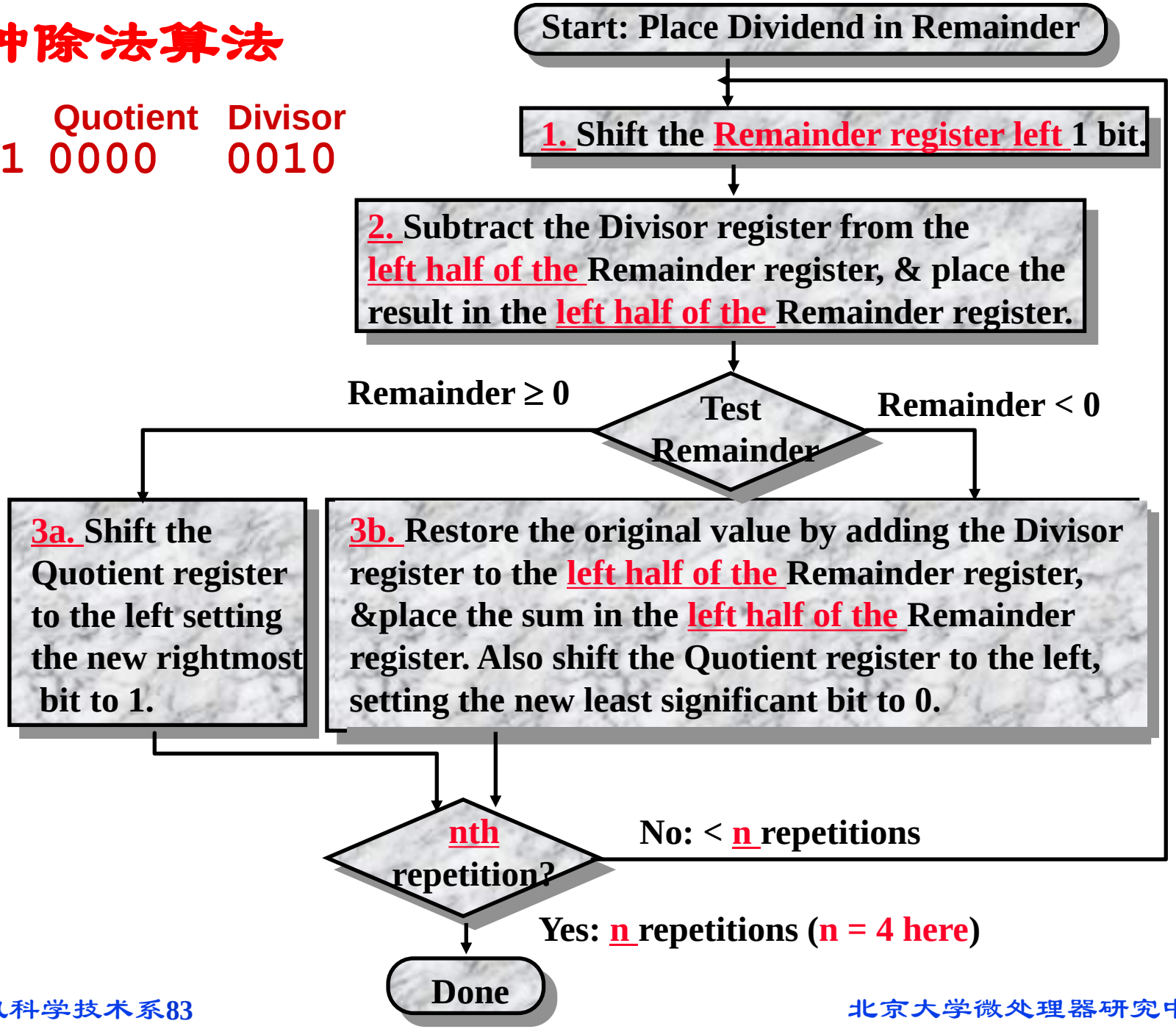
第二种除法算法

Remainder Quotient Divisor
0000 0111 0000 0010

	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: 0000	D: 0010	R: 0011 0000
2: R = R-D	Q: 0000	D: 0010	R: 0001 0000
3a: sl Q, 1	Q: 0011	D: 0010	R: 0001 0000

第二种除法算法

Remainder	Quotient	Divisor
0000 0111	0000	0010



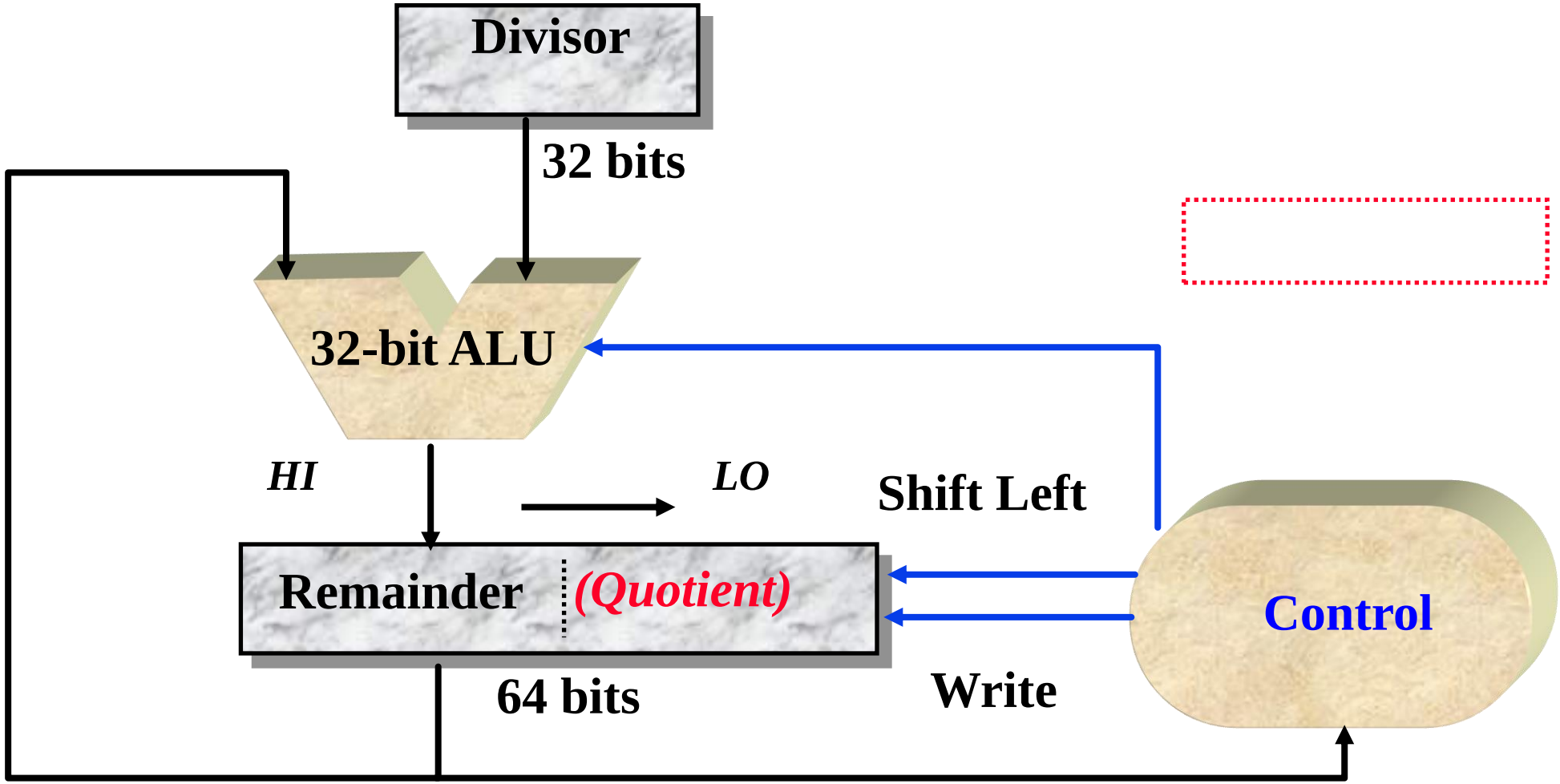
第二种除法的启示

- 通过在左移中,与余数合并, 取消商寄存器
  - 象前面一样, 以左移余数开始.
  - 其后, 由于余数寄存器的移动即可移动左半部的余数,又可移动右半部的商,因而每次循环只包括两步.
  - 将 两个寄存器联合在一起 和 循环内新的操作次序 的导致

    余数多左移一次
  - 因而, 最后一步 必须 将这个寄存器左半部中的余数 移回

第三种除法算法

- 32位除数寄存器、32位ALU、64位余数寄存器(没有商寄存器)



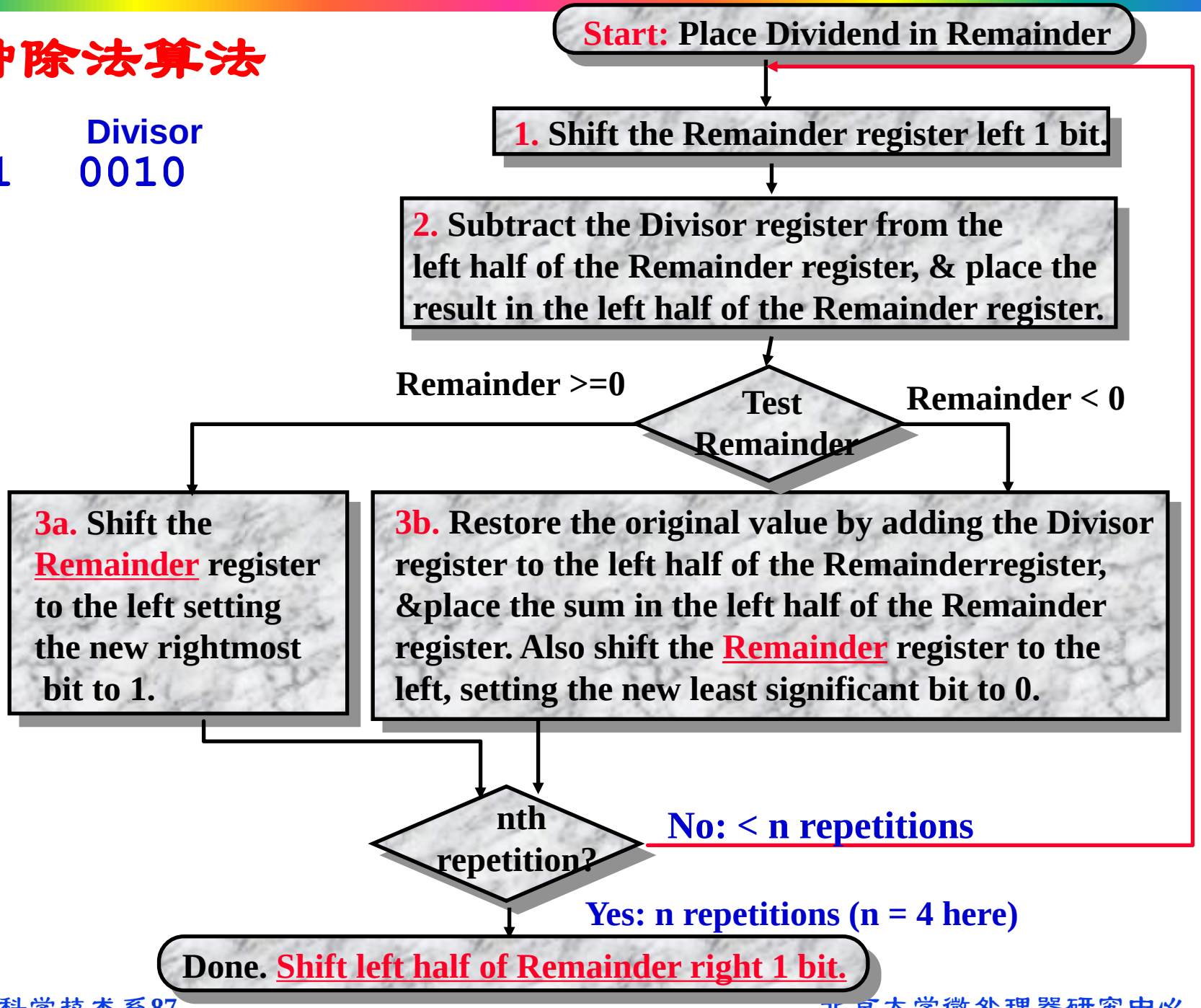
第三种除法算法

Remainder 0000 0111 Divisor 0010

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

第三种除法算法

Remainder Divisor
0000 0111 0010



第三种除法算法的启示

- 与乘法的硬件相同：只需要 ALU进行加法或减法, 并有一个可左移或右移的64位寄存器
- MIPS中的Hi 和 Lo 寄存器合并起来 作为乘法和除法的 64位寄存器
- 有符号除法: 最简单的方法是记住符号, 进行正数除法, 并根据需要对商和余数进行修正
  - 注: 被除数和余数的符号必须相同
  - 注: 如果除数和被除数的符号不同, 商为负
- 商有可能很大: 如果一个64位整数除以 1，那么商就为 64 位（称为“饱和”：saturation）

有符号除法的商和余数

被除数 = 商 × 除数 + 余数

（±7）÷（±2）？

（+7）÷（+2）：商为+3、余数为+1

⇒ 7 = 3 × 2 + 1

（-7）÷（+2）

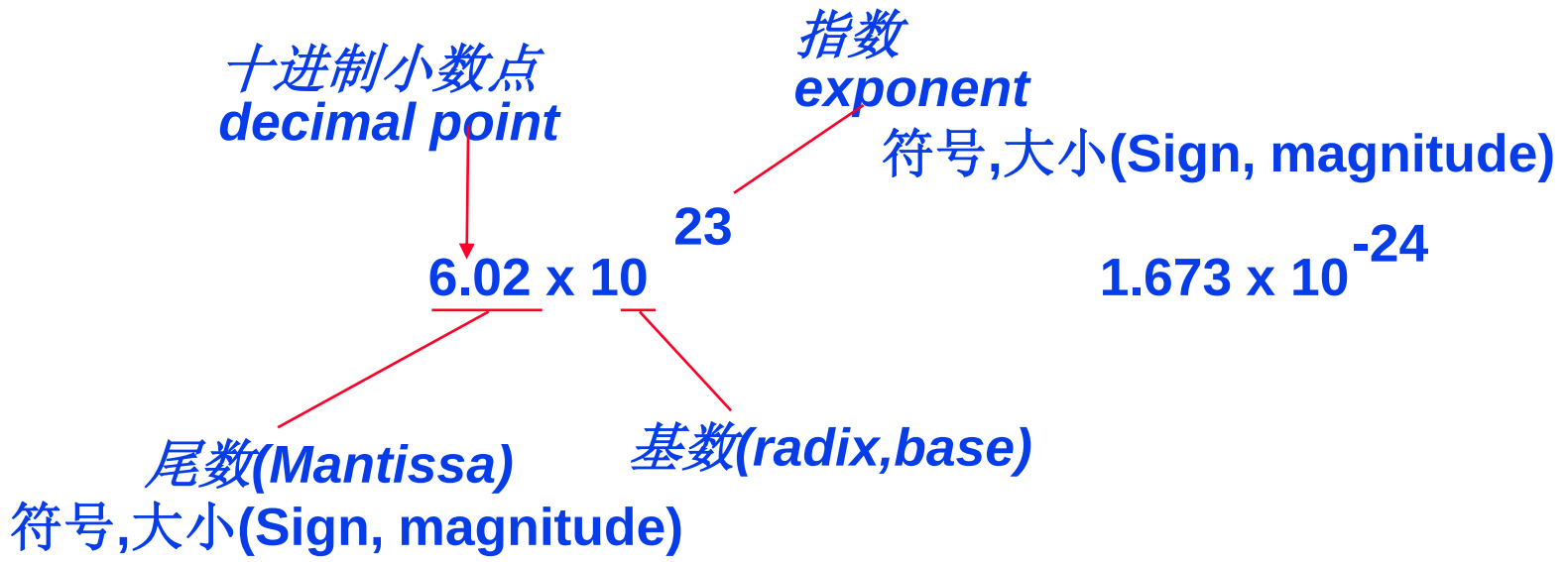
-7 =（-3）× 2 +（-1）　　商为多少？

-7 =（-4）× 2 +（+1）　　余数为多少？

如果操作数的符号不一致，商为负，

且被除数和余数的符号必须相同

科学计数法



问题:

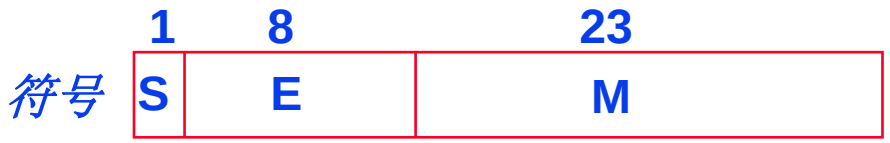
- 算术运算 (+, -, ×, ÷)
- 表示, 正规形式
- 范围和精度
- 舍入
- 意外事件 (例如, 除零、溢出、下溢)
- 出错
- 特性 (负数, 取反, if A ≠ B then A - B ≠ 0)

IEEE 浮点. $(\pm)1.M \times 2^{e-127}$

浮点运算

IEEE 754 标准 浮点数表示：

单精度



指数: 大于127的二进制整数

尾数: 符号 + 大小, 具有隐含整数位的规格化二进制数t: 1.M

真正的指数为：

$e = E - 127 = 10000001_2 - 127$

$0 < E < 255$

$N = (-1)^S 2^{E-127} (1.M)$

0 = 0 00000000 0 . . . 0　　-1.5 = 1 01111111 10 . . . 0

规格化数可以表示的数值的大小（Magnitude）范围：

$2^{-126} (1.0)$ ~ $2^{127} (2 - 2^{-23})$

大致为：

1.8×10^{-38} ~ 3.40×10^{38}

(对IEEE浮点数进行整数比较是有效的!)

浮点数表示

有理数(rational)

精度: 关于基数B的精度位数p ；　　指数的范围：偏移量 ~ （2^i - 1 - 偏移量）

（这里，假设尾数部分已经规格化了，即$|M| \geq 1/B$）

IEEE浮点标准中，最小和最大的指数域值 0 和 （2^i - 1）被保留 ，有特殊用途

假设： B = 3, p = 2

0	1	0 . 2 1

$$+(2\times 3^{-1}+1\times 3^{-2})\times 3^{1-bias}=\frac{4}{9}\times 3^{1-bias}$$

3^0到3^1之间的数值：

$3^1\times(1\times 3^{-1}+0\times 3^{-2})=1$

$3^1\times(1\times 3^{-1}+1\times 3^{-2})=1+1/3$

$3^1\times(1\times 3^{-1}+2\times 3^{-2})=1+2/3$

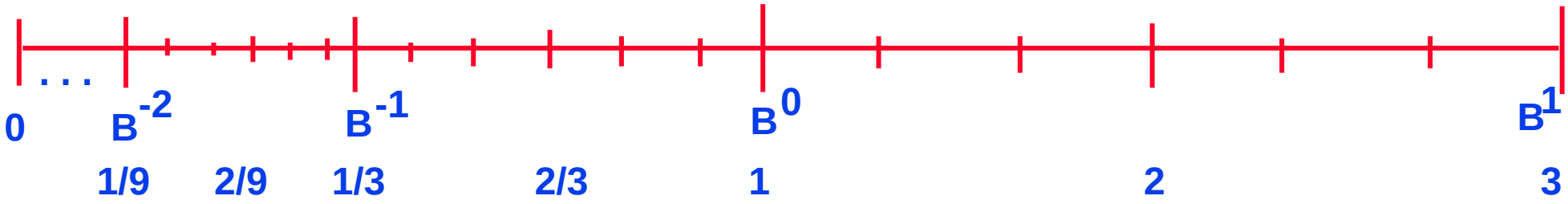
$3^1\times(2\times 3^{-1}+0\times 3^{-2})=2$

$3^1\times(2\times 3^{-1}+1\times 3^{-2})=2+1/3$

$3^1\times(2\times 3^{-1}+2\times 3^{-2})=2+2/3$

规格化浮点数的数值分布

假设： B = 3, p = 2



注意：每个$[B_i , B_{i+1})$区间长度都比前一个区间$[B_{i-1} , B_i)$的大, 但是每个区间内表示的数值的数目却相同： $(B-1)B^{p-1}$ - 对于每个指数数值非零的左部调整后的尾数大小的数目。 因而，对于每个连续区间，表示的数值的密度将减少B倍！

规格化数

规格化数新定义: $B>|M|\geq 1.0$

尾数向左调整 ⇒ 尾数尽可能大, 指数尽可能小

例如, $B = 2^4$, $p = 3$:

0	0110	0000 . 0110 1100

= 0.6C

非规格化(denormalized)

0	0101	0110 . 1100 0000

= 6.C0

规格化(normalized)

当 $B = 2$, 在进行向左调整后, 尾数的最大位总是为1. 因而就没有必要将这个“隐含”位存放在存储器中!

0	011	1.01

1

没有隐含位

0	011	.011

带隐含位⇒提高精度

在 FPU内部, 这个隐含位将被重新插入; 这是因为在进行浮点加法/减法之前, 需要先非规格化.

最小的规格化数: 0 0 . . . 0 “1” . 0 . . . 01 隐含位

0 0 . . . 0 1.0 . . . 00 无隐含位

必须与 0区别!

2^{-bias}

将浮点数转换成十进制数

0	0110 1000	101 0101 0100 0011 0100 0010

- Significand: $1 + (s1x2^{-1}) + (s2x2^{-2}) + \ldots$
  - $1+2^{-1}+2^{-3}+2^{-5}+2^{-7}+2^{-9}+2^{-14}+2^{-15}+2^{-17}+2^{-22}$

  $= 1+ 1/2 + 1/8 + 1/32 +1/128 + 1/512 + 1/16384 + 1/32768 + 1/131072 + 1/4194304$

  - Multiply fractions by 4194304 for sum

    $= 1.0 + (2097152 + 524288 + 131072 + 32768 + 8192 + 256 + 128 + 32 + 1)/4194304$

    $= 1.0 + (2793889)/4194304$

    $= 1.0 + 0.66612$

- Bits represent: $+1.66612_{ten} * 2^{-13} \sim +2.034 * 10^{-4}$

基本算术运算算法

浮点加法(或减法)的基本步骤:

(1) 计算 Ye - Xe

(2) 右移 Xm , 形成 Xm 2^{Xe-Ye}

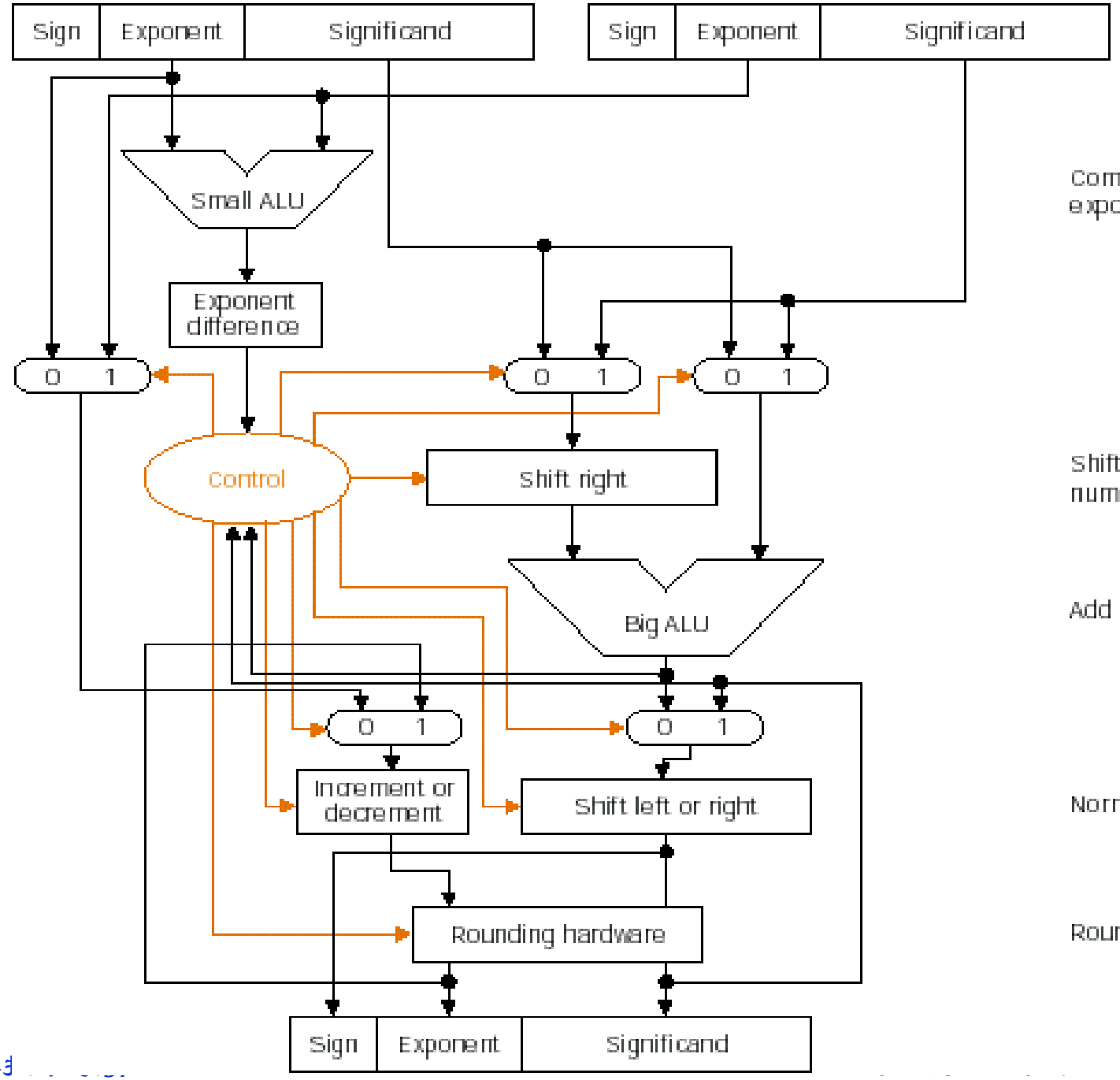
(3) 计算 Xm 2^{Xe-Ye} + Ym

如果结果需要规格化, 那么执行如下步骤:

(4) 左移结果, 减小结果的指数
右移结果, 增加结果的指数
重复上述步骤直到数据的最大位(MSB)为1 (注意在IEEE标准中的隐含位)

(5) 如果结果的尾数为 0, 需要特定步骤来将指数设置为0

浮点加法ALU的逻辑框图



附加位(Extra Bits)

"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."

需要多少附加位?

IEEE: 只要能够正确、全面地计算出结果.

加法:

	1.xxxxx	1.xxxxx	1.xxxxx
+	1.xxxxx	0.001xxxxx	0.01xxxxx
	1x.xxxxy	1.xxxxxyyy	1x.xxxxyyy
	后规格化 (post-normalization)	前规格化 (pre-normalization)	前和后 (pre and post)

- *保护位(Guard Digits)*: 在尾数大小中前p数位右边的数位, 用于防止丢失数位。在以后的规格化过程中，保护位可以左移到前p数位中。
- 加法: 移回最高进位（carry-out shifted in）
- 减法: 借数据位和保护位（borrow digit and guard）
- 乘法: 进位和保护位（carry and guard）， 除法需要保护位

数字舍入 (Rounding Digits)

规格化后的结果, 但是有效尾数大小的右边还有一些非零数字
⇒ 这个数据就需要舍入

例如, B = 10, p = 3:

	0 2 1.69	= 1.6900 * 10	2-bias
-	0 0 7.85	= - .0785 * 10	2-bias
	0 2 1.61	= 1.6115 * 10	2-bias

一个舍入数字必须被进位到保护位的右边，因而在规格化左移之后，根据舍入位的数值，可以对该结果进行舍入处理。

IEEE标准:

四种舍入方式:

- 舍入成 最接近的数（省缺）
- 舍入成 正无穷大
- 舍入成 负无穷大
- 舍入成 零

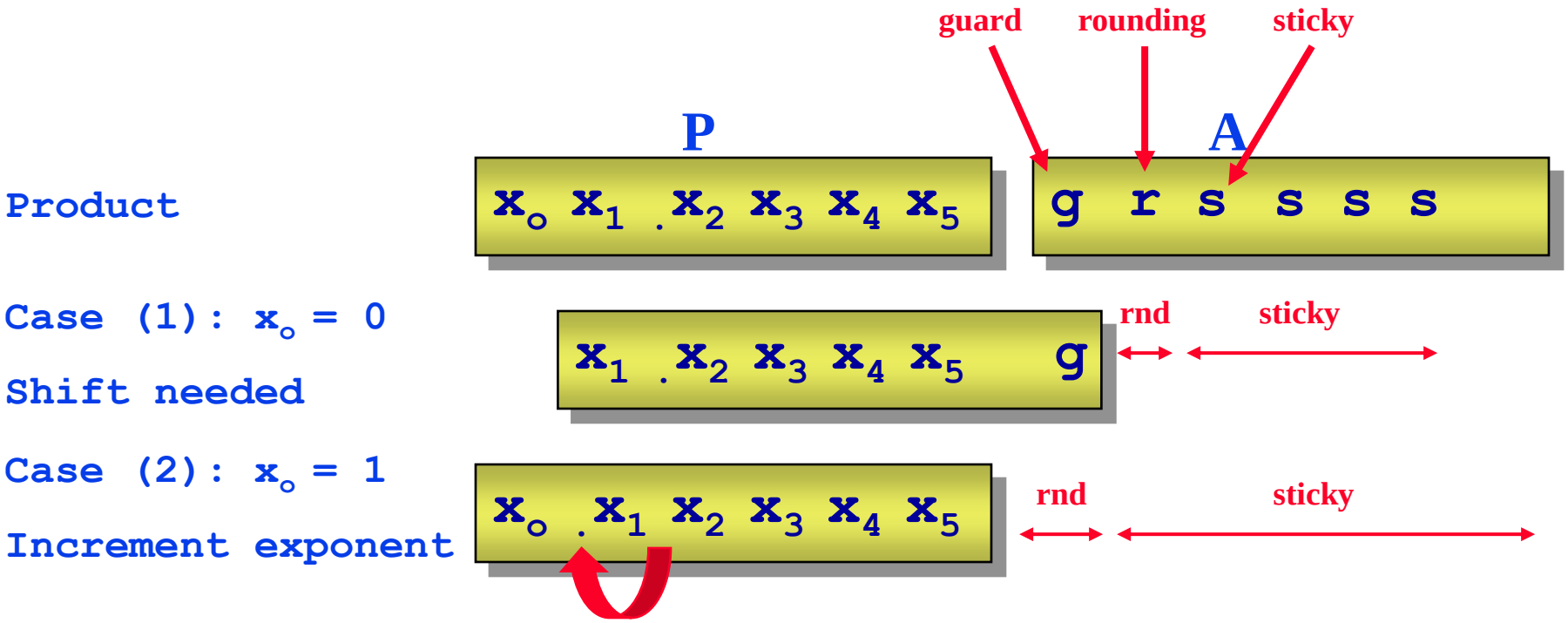
舍入成 最接近的数:

- 舍入数字 < B/2，那么 截去
- > B/2，那么 舍入
- = B/2，那么 舍入成最近的偶数数字

可以看出这一策略将由于引入舍入而产生的平均错误减至了最小

浮点乘法与舍入

A、B、P寄存器 p位宽度（本例中，p=6），2p位乘积存放在(P,A)中



- The high-order bit of P is 0. Shift P left 1 bit, shifting in the g bit from A. Shifting the rest of A is not necessary
- The high-order bit of P is 1. Set s := s ∪ r and r:=g, and add 1 to the exponent

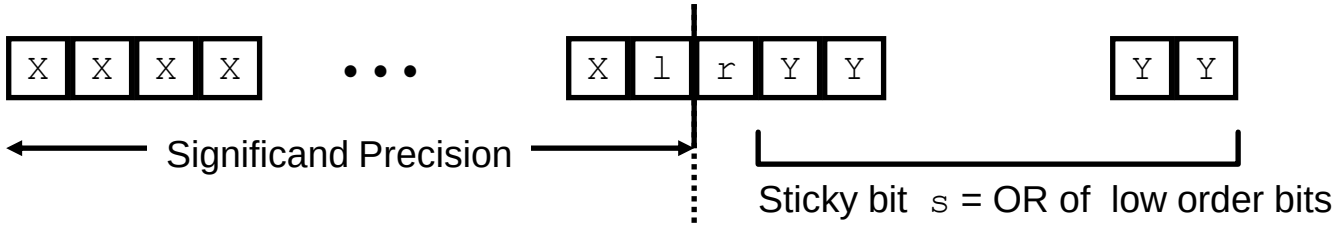
浮点操作

- **Conceptual View**
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into significiand
- **Actual Implementation**
  - Tricks to avoid computing all low order bits
- **Rounding Modes (illustrate with $ rounding)**

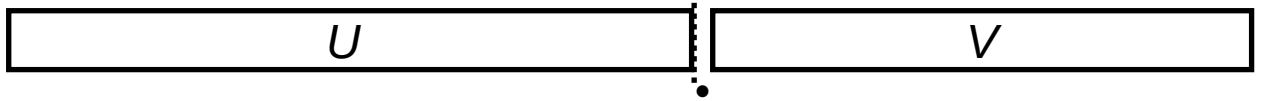
	$1.40	$1.60	$1.50	$2.50	–$1.50
• Zero	$1.00	$2.00	$1.00	$2.00	–$1.00
• $-\infty$	$1.00	$2.00	$1.00	$2.00	–$2.00
• $+\infty$	$1.00	$2.00	$2.00	$3.00	–$1.00
• Nearest Even (default)	$1.00	$2.00	$2.00	$2.00	–$2.00

实现舍入：负

Format of Exact Result Significand



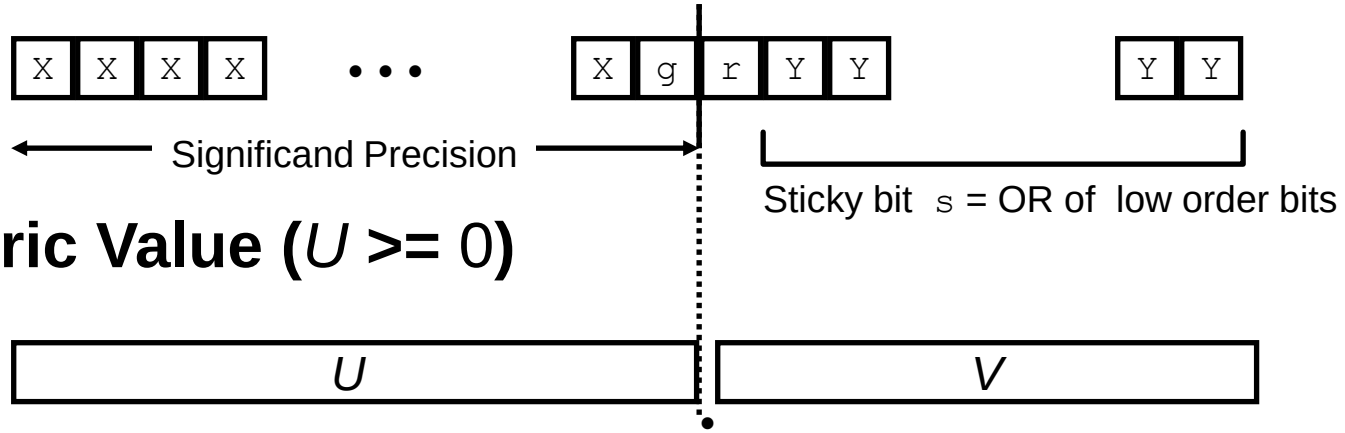
Numeric Value ($U < 0$)



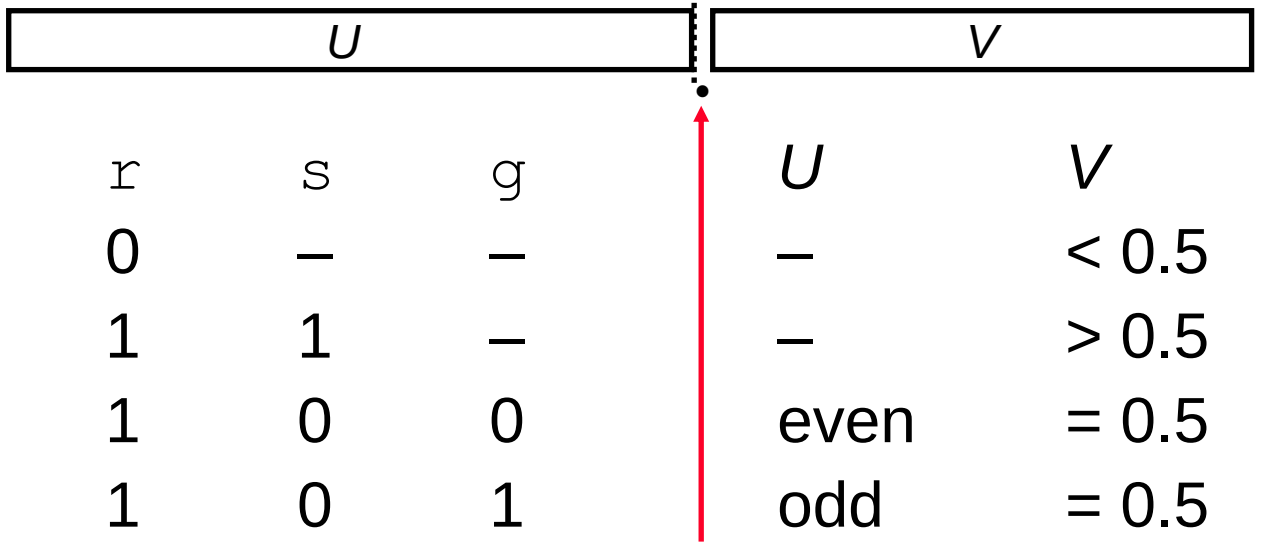
r	s	g	*U*	*V*
0	–	–	–	> –0.5
1	1	–	–	< –0.5
1	0	0	even	= –0.5
1	0	1	odd	= –0.5

实现舍入：非负

Format of Exact Result Significand

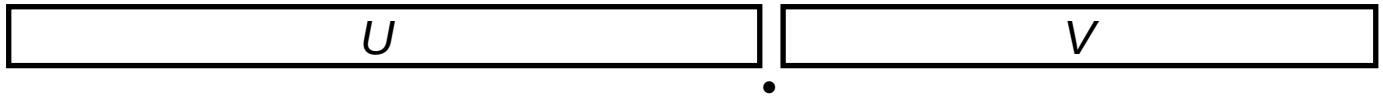


Numeric Value ($U >= 0$)



r	s	g	*U*	*V*
0	–	–	–	< 0.5
1	1	–	–	> 0.5
1	0	0	even	= 0.5
1	0	1	odd	= 0.5

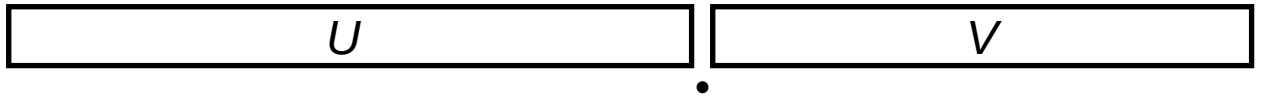
舍入规则：非负



Rounded Significand (*U* >= 0)

r	s	g	*U*	*V*	RT0	RT+Inf	RT–Inf	RTE
0	–	–	–	< 0.5	*U*	*U*	*U*	*U*
1	1	–	–	> 0.5	*U*+1	*U*+1	*U*+1	*U*+1
1	0	0	even	= 0.5	*U*	*U*+1	*U*	*U*
1	0	1	odd	= 0.5	*U*	*U*+1	*U*	*U*+1

舍入规则：负



Rounded Significand (*U* < 0)

r	s	g	*U*	*V*	RT0	RT+Inf	RT–Inf	RTE
0	–	–	–	> –0.5	*U*	*U*	*U*	*U*
1	1	–	–	< –0.5	*U*–1	*U*–1	*U*–1	*U*–1
1	0	0	even	= –0.5	*U*	*U*	*U*–1	*U*
1	0	1	odd	= –0.5	*U*	*U*	*U*–1	*U*–1

粘位 (Sticky Bit)

为进一步改进舍入处理的结果，在舍入数字右边的附加位

```
  d0 . d1 d2 d3 . . . dp-1  0  0  0
+  0 .  0  0  X . . .  X    X  X  S
                            X  X  S
```

粘位：如果有在舍入数字的尾部有任何1位丢失，则置为1

```
  d0 . d1 d2 d3 . . . dp-1  0  0  0
-  0 .  0  0  X . . .  X    X  X  0
                            X  X  0
```

```
  d0 . d1 d2 d3 . . . dp-1  0  0  0
-  0 .  0  0  X . . .  X    X  X  1
```

产生一个借位

舍入策略小结:

基数 2 将精度的不稳定性 减至最小

+ 、- 、× 、 ÷ 中的正常操作需要一个进位/借位，以及一个保护位

为了正确舍入，需要一个舍入数字

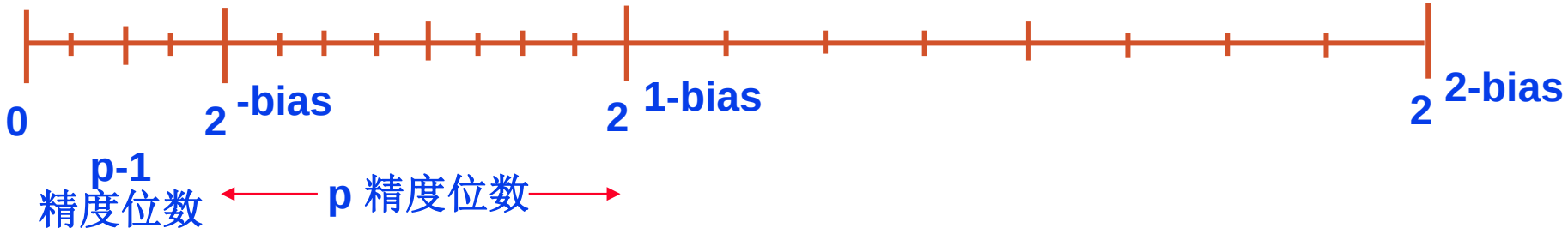
为了更高的准确性，当舍入数字为 B/2 时，需要一个粘位

如果假设数字是均匀分布的，那么舍入到最接近的数值的平均错误 = 0

非规格化数值

在0和下一个规格化数可表示的数值之间的间隙比任何其他相邻的两个可表示的数值之间的间隙要 大许多！

IEEE标准使用非规格化数值来填补这一间隙，
使得在接近0的数值之间的距离非常接近.



相同间距，减半了有效数值!

注：PDP-11, VAX不能表示低于正常的数值，这些机器用下溢成0来代替！

$$\text{Normalized Number} = (-1)^S * (1.M) * 2^{E-127}$$

$$\text{Denormalized Number} = (-1)^S * (0.M) * 2^{-126}$$

无穷大和非数 (NaN)

操作的结果溢出, 也就是说, 大于最大的可以表示的数值

注意：溢出与除零不同（产生不同的中断）！

±无穷大

这对于 对无穷大进行进一步运算是有意义的!
例如, **X/0 > Y** 也许是有效的比较

非数（**Not a number**）, 并不是无穷大 **(例如. sqrt(-4))**
无效操作中断（除非操作是 = 或 ≠）

NaN

NaN具有出传播性: f(NaN) = NaN

IEEE 754浮点数标准

Single Precision		Double Precision		Object represented
Exponent	Significand	Exponent	Significand	
0	0	0	0	0
0	nonzero	0	nonzero	±denormalized number
1-254	anything	1-2046	anything	±floating-point number
255	0	2047	0	±infinity
255	nonzero	2047	nonzero	NaN(Not a Number)

中断 (Exceptions)

无效操作:

操作的结果是 NaN （除了 = 或 ≠）

无穷大 ±无穷大; 0× 无穷大; 0÷0; 无穷大÷无穷大; x除以 y的余数，这里 y = 0;

sqrt(x)，这里 x < 0, x = ±无穷大

溢出:

操作的结果大于最大可以表示的数值

如果没有激活溢出中断，那么就变成 ± 无穷大。

除零:

x/0，这里 x = 0, ±不穷大;

如果没有激活除零中断，那么就变成±无穷大。

下溢:

小于规格化数的结果 或者 非零结果下溢成0

不精确数:

舍入后的结果，而不是实际结果 (舍入错误 ≠ 0)

IEEE 标准 ⇒ 指明省缺情况，并且允许用户来处理意外中断

这与完全放弃计算的一些做法形成鲜明对比！

Saving Bits

- Many applications in machine learning, graphics, signal processing can make do with lower precision
- IEEE “half-precision” or “FP16” uses 16 bits of storage
  - 1 sign bit
  - 5 exponent bits (exponent bias of 15)
  - 10 significand bits

BrainWave

- Microsoft “BrainWave” FPGA neural net computer uses proprietary 8-bit and 9-bit floating-point formats