Computer Organization and Design

中山大学

计算机学院

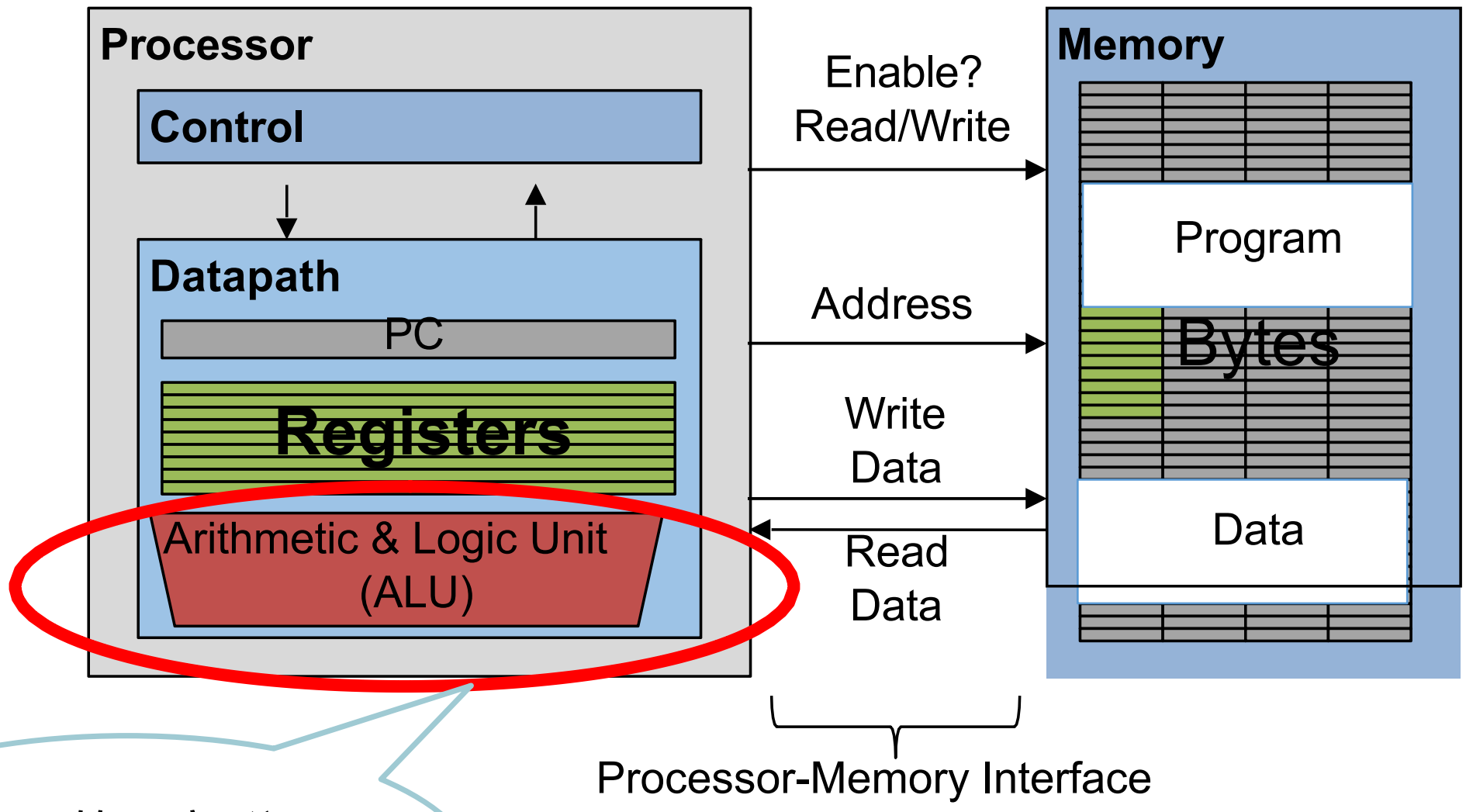
郭雪梅

Email:guoxuem@mail.sysu.edu.cn

Datapath Elements: Mux + ALU

* 设计过程与ALU设计
* The Design Process & ALU Design

Processor-ALU



处理器的一部分，
执行处理器的操作

Topics

- Arithmetic Circuits 运算电路
  - Adder 加法器
  - Subtractor 减法器

- Logic Circuits 逻辑电路

- Arithmetic & Logic Unit (ALU)
  - … 把算术运算和逻辑运算放在一起

- Reading: Ch. 3.1-3.5, Appendix C.5、C6

- Skim: Ch. 3.6-3.8

Type of Circuits

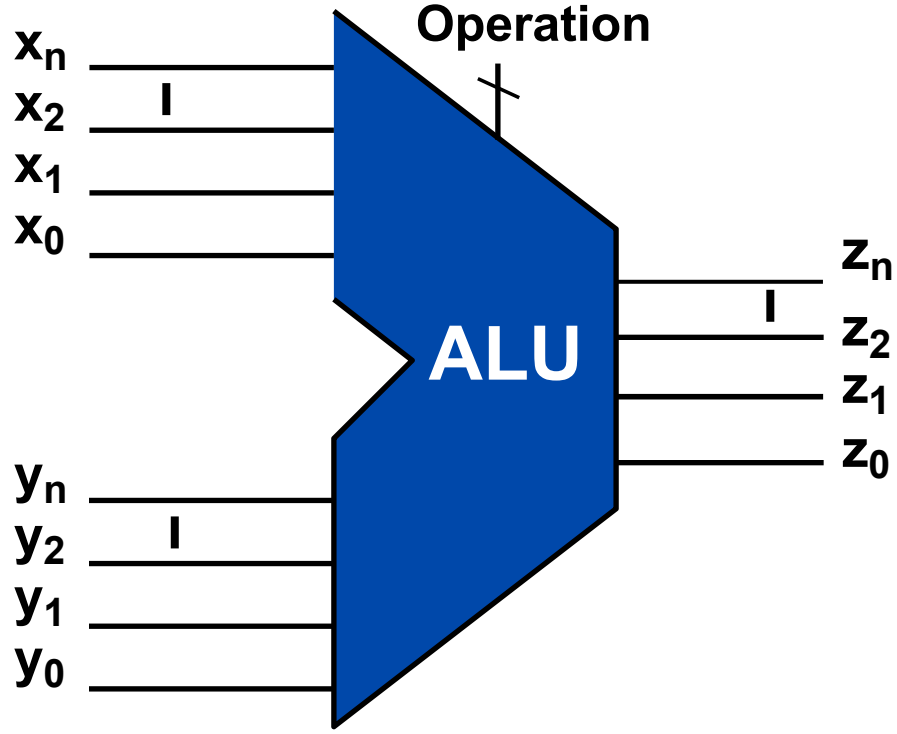
- *同步数字系统由两部分电路组成*:

  - **组合逻辑电路(CL)**
    - 输出只是输入的函数
    - E.g., 加法电路 add A, B (ALUs)

  - **时序逻辑电路(SL)**
    - 电路有 “记忆”， 能存储信息
    - “State Elements”状态元
    - 有记忆能力的寄存器(Registers)

Building It! The ALU

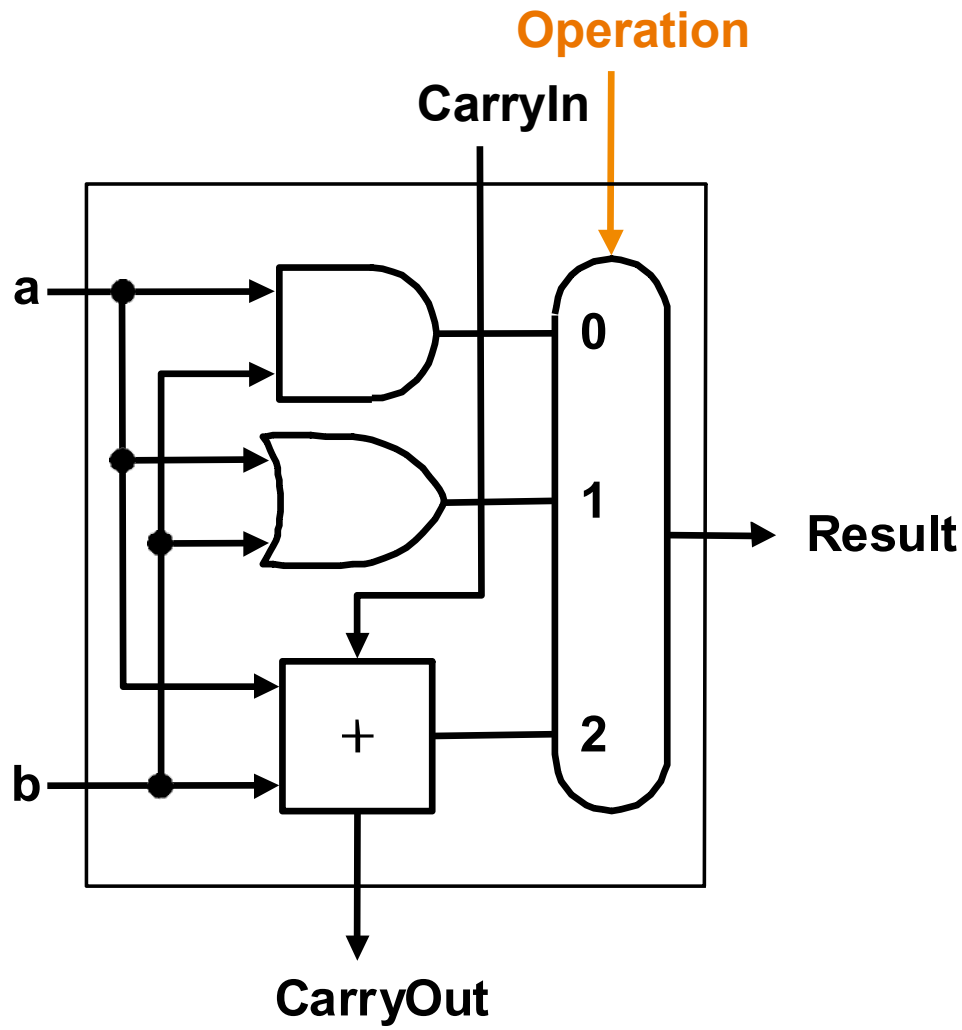
- 对两个 N-bit 进行操作的ALU
- 我们设计一个支持 MIPS and, andi, or, ori 的逻辑操作以及算术运算的ALU
- 先一位，再到32位



运算和逻辑单元

- 大多数处理器都包含一个称为“算术逻辑单元”（ALU）的特殊逻辑块
- 我们将向您展示一个简单的 ADD、SUB、按位与、按位或

先构建一位ALU



Operation=00,Result=a+b

Operation=01,Result=a-b

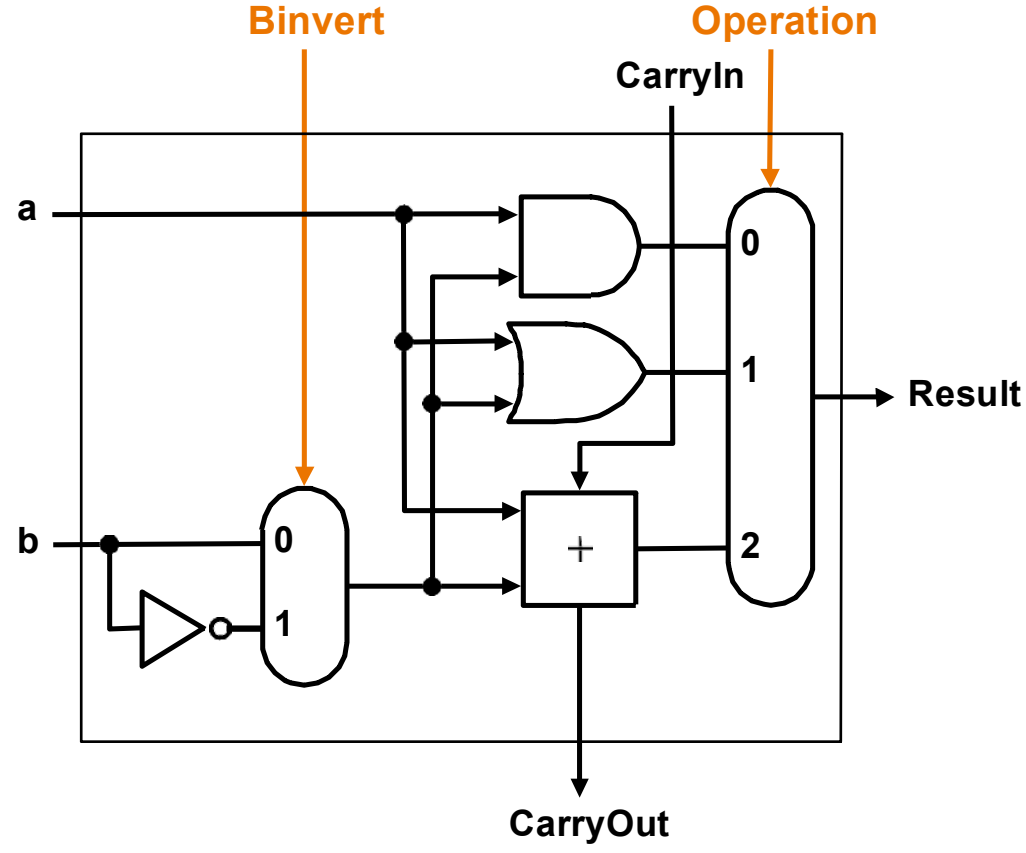
Operation=10,Result=a&b

Operation=00,Result=a or b

减法怎么设计?

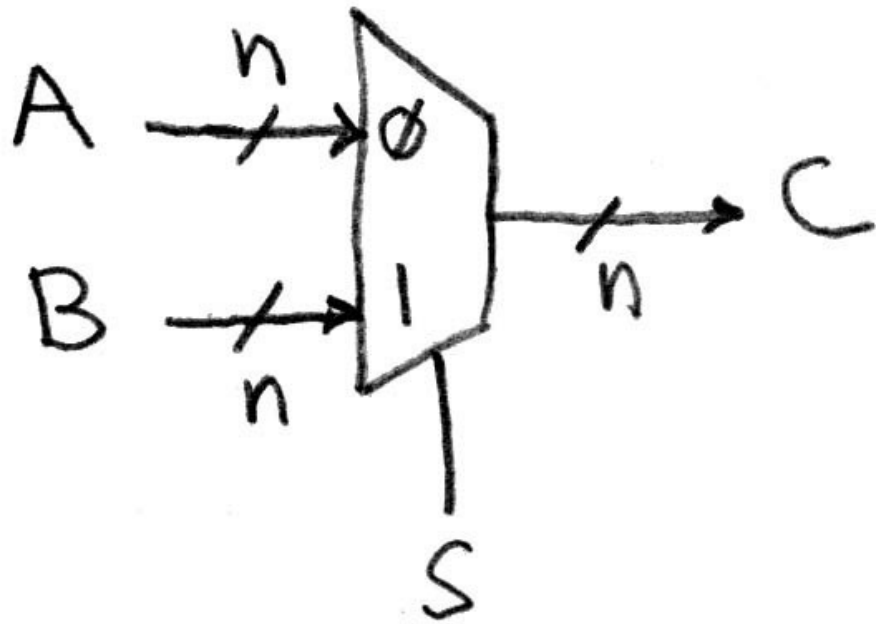
- 二进制补码方法：只需对b的补码求和， Binvert =1，a+（b反+ Binvert ）。
- 一个优雅的解决方案 - 在位 0 上使用进位 Carrylin

(减法,Binvert =1，加法, Binvert =0)：



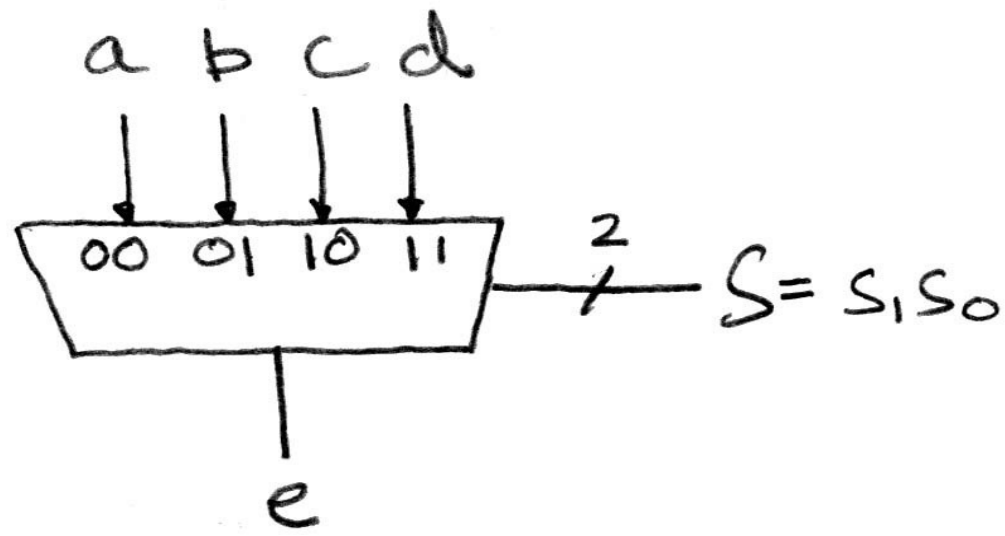
构建标准的函数单元

数据选择器 (“Mux”)
(here 2-to-1, n-bit-wide)



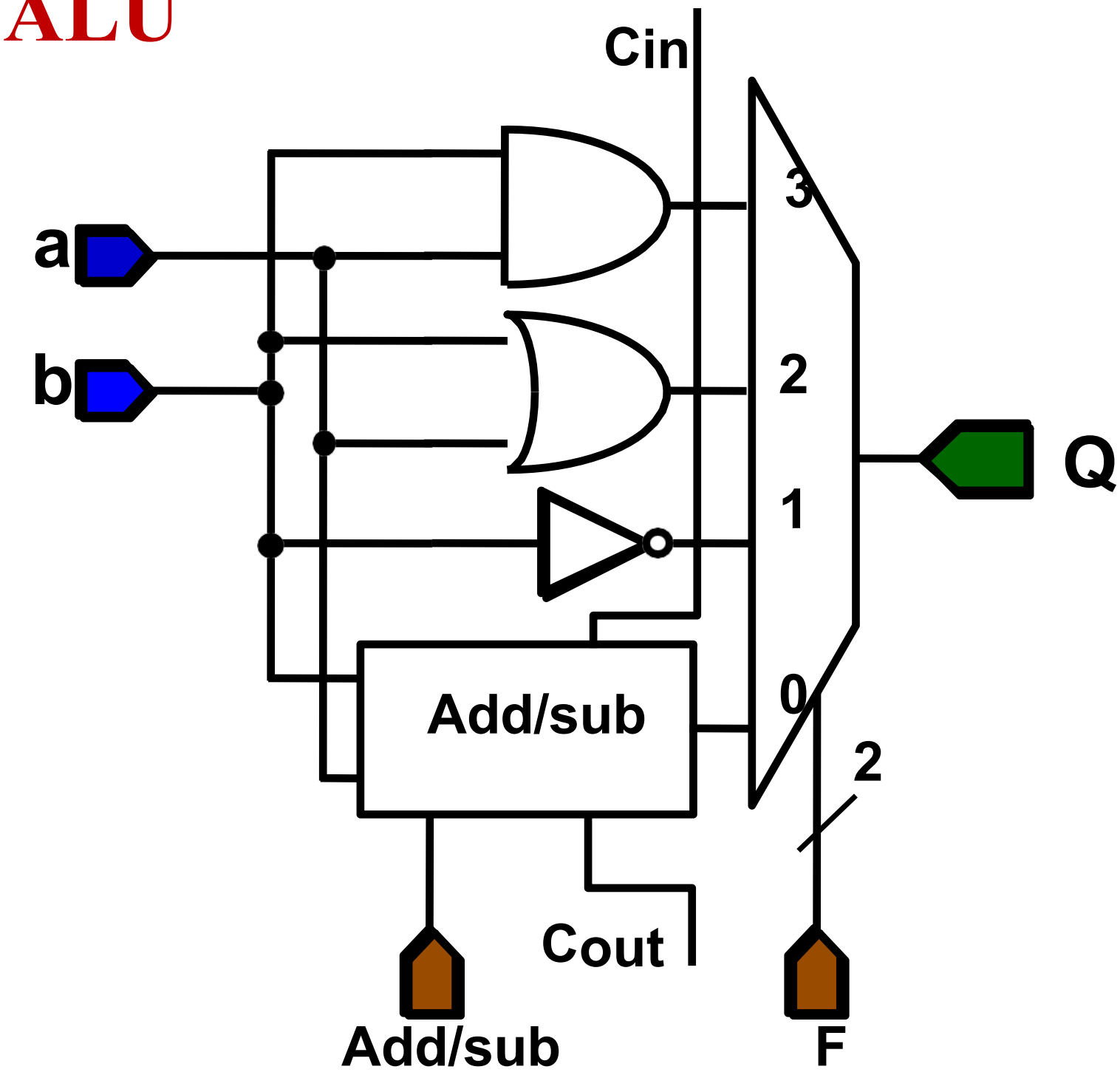
$$\overline{s}a + sb$$

4-to-1 multiplexer?



$$e = \overline{s_1 s_0} a + \overline{s_1} s_0 b + s_1 \overline{s_0} c + s_1 s_0 d$$

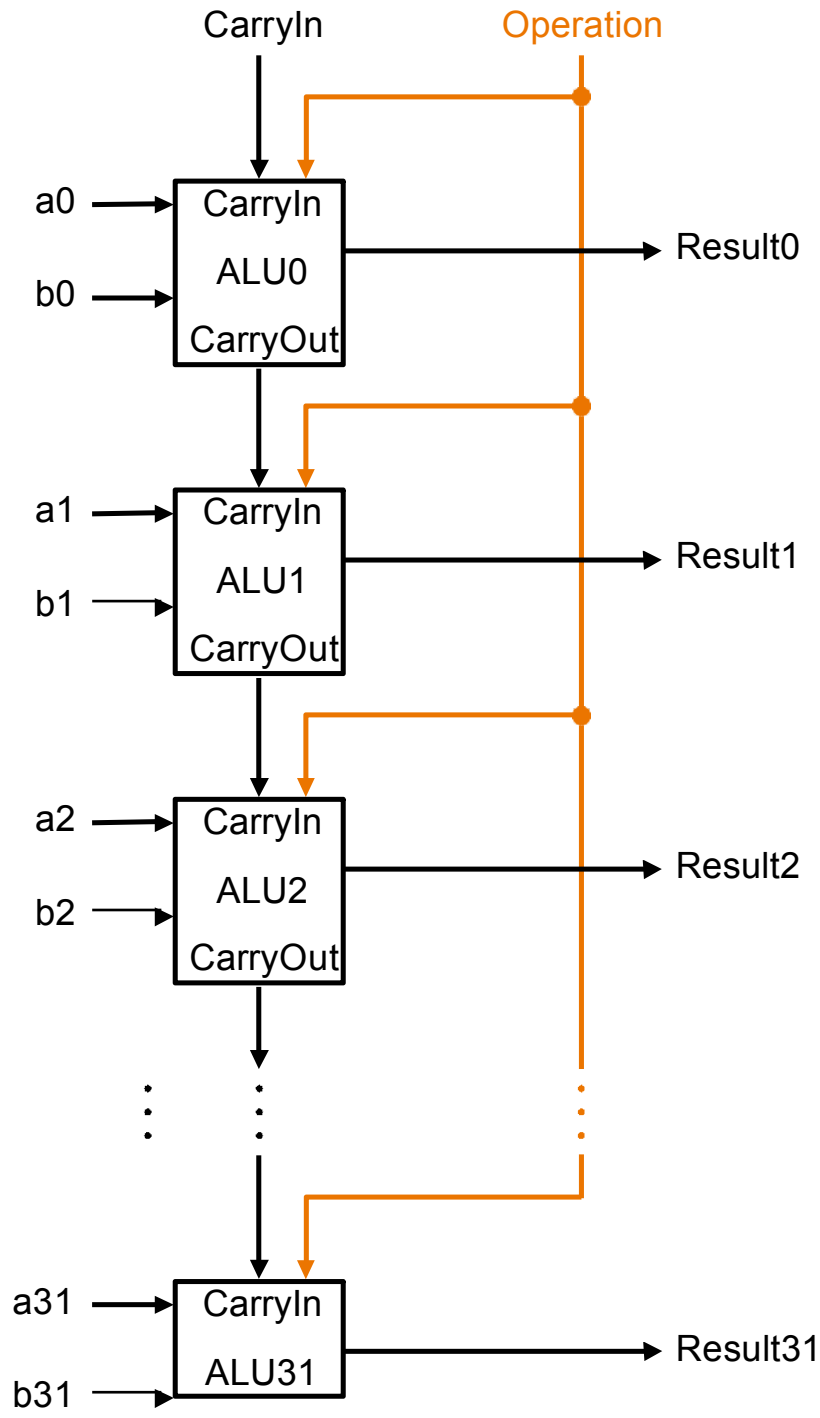
简单的 ALU



构建 32 bit ALU 的 add 运算

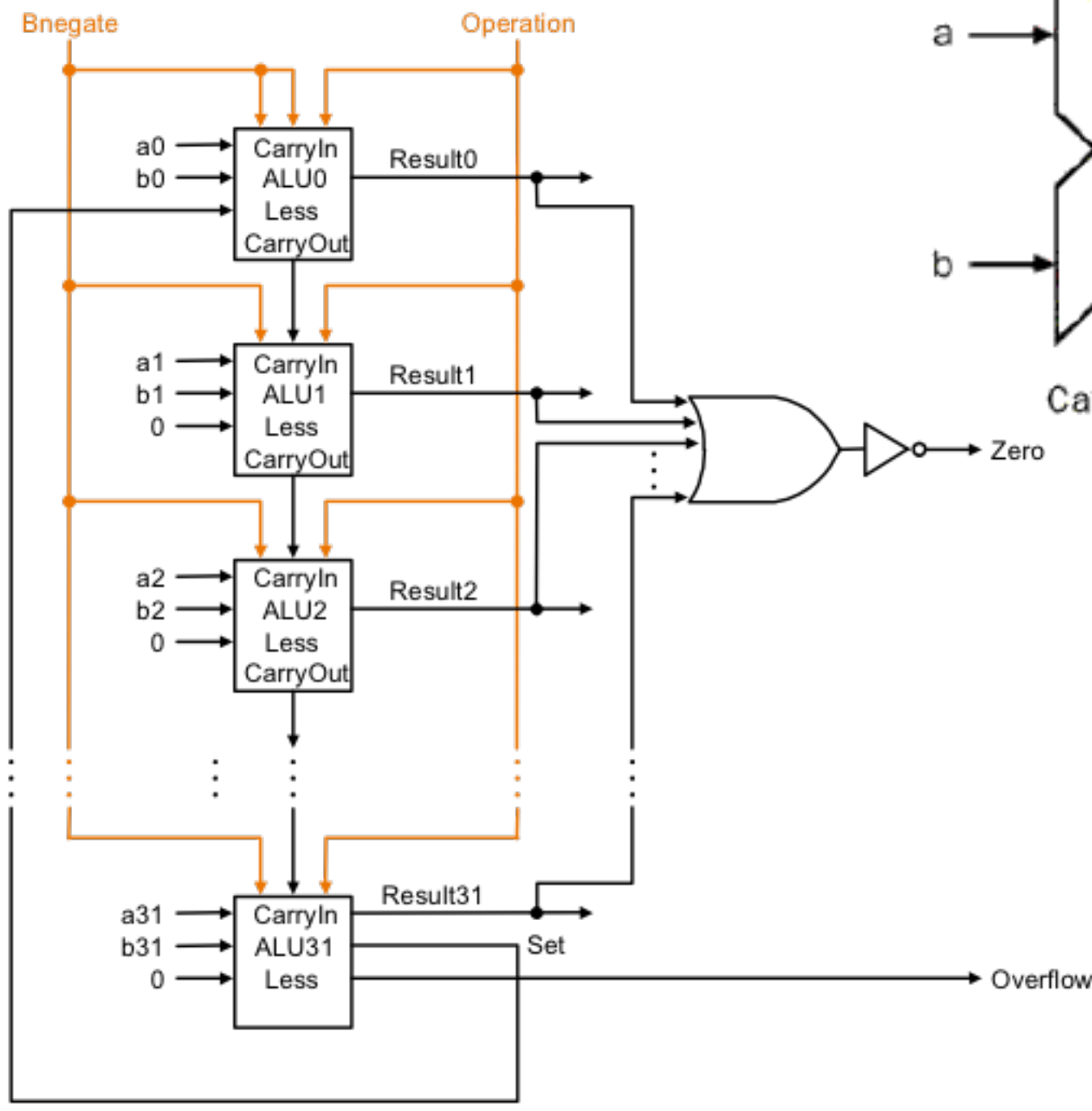
- 输入是并行的
- 进位是级联的
- 行波进位加法器
- 缓慢但简单
- 第一次进位 = 0

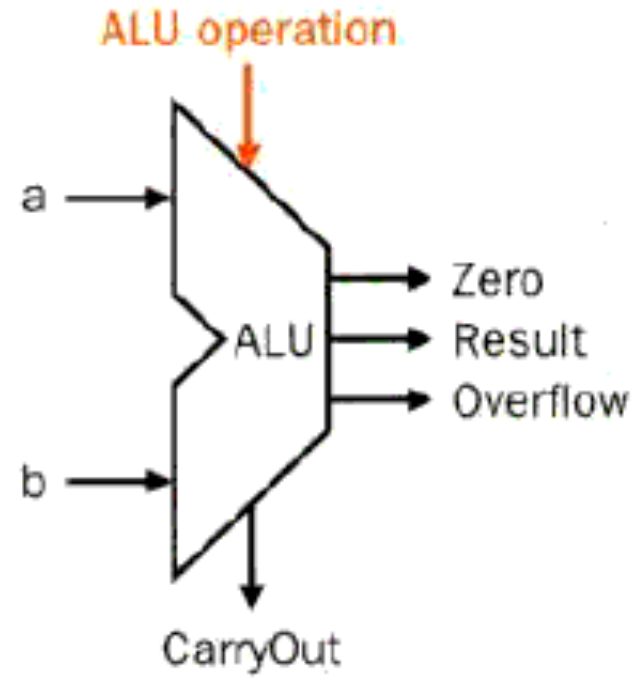
Forms a Ripple Carry Adder 行波进位加法器



Complete ALU

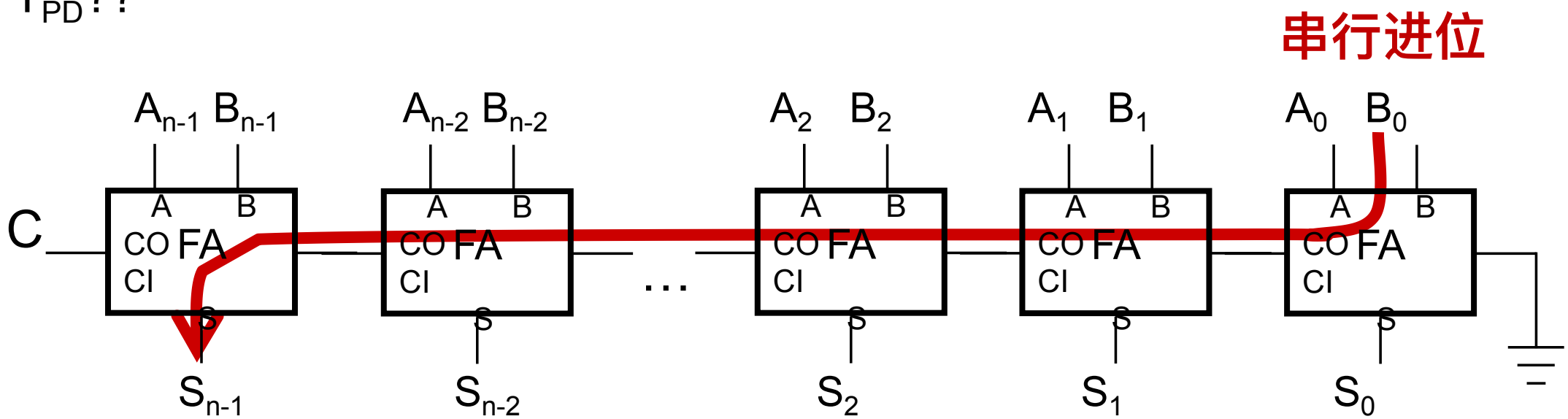
- ✱ 输入
  - A
  - B
- ✱ 控制线
  - Binvert
  - Operation
  - Carryin
- ✱ 输出
  - Result
  - Overflow
  - Zero





行波进位加法器（Ripple Carry Adder）

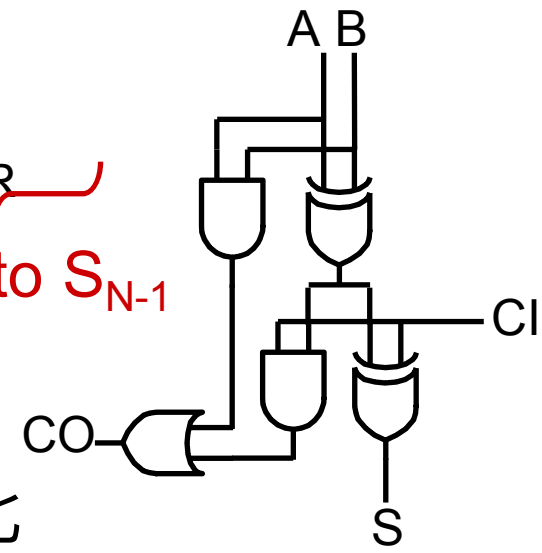
(What is T_{PD}??



最坏情况: 进位从最低位传到最高位LSB to MSB, e.g., 比如11…111 to 00…001.

$t_{PD} = (t_{PD,XOR} + t_{PD,AND} + t_{PD,OR}) + (N-2)*(t_{PD,OR} + t_{PD,AND}) + t_{PD,XOR}$

A,B to CO　　CI to CO　　CI_{N-1} to S_{N-1}



$\approx \Theta(N)$

$\Theta(N)$ is read “order N”加法器的延迟与操作数中的位数成正比

- 加位将从最低位(LSB)传播到最高位(MSB)
  - N位加法器在最坏情况下的延迟: 2N个门延迟

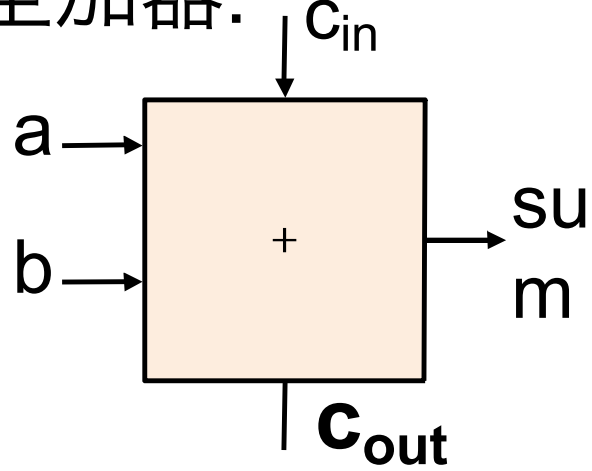
性能问题?

行波进位加法器- 太慢!

怎样解决?

先看一下进位公式:

- 1-bit 全加器:



a	b	Cin	Cout	SUM
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$C_{out} = (\bar{a}b + a\bar{b})C_{in} + ab = (ab + \bar{a}b + ab + a\bar{b})C_{in} + ab$$

$$= (b + a)C_{in} + ab$$

$$\text{Sum} = (a \oplus b) \oplus C_{in}$$

Carry-Lookahead Adder (CLA) 先行进位加法器

没有进位值,我们可以做什么?

$$c_{i+1} = b_i c_i + a_i c_i + a_i b_i$$

$$c_{i+1} = (a_i b_i) + (a_i + b_i) c_i$$

定义**G 为**产生进位，$G_i = a_i b_i$,定义**P 为**传播进位，

$$P_i = a_i + b_i$$

$$c_{i+1} = (a_i b_i) + (a_i + b_i) c_i$$

$$c_{i+1} = G_i + P_i c_i$$

$$c_{i+1} = G_i + P_i c_i$$

N位先行进位加法器

设置P和G两个先行进位输出端

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**

- c1 = g0 + (p0 * c0)
- c2 = g1 + p1*c1 = g1 + (p1 * g0) + (p1 * p0 * c0)
- c3 = g2 + p2*c2 = g2 + (p2 * g1) + (p2 * p1 * g0) + (p2 * p1 * p0 * c0)
- c4 = g3 + p3*c3 = g3 + (p3 * g2) + (p3 * p2 * g1)

  +(p3 * p2 * p1 * g0) + (p3 * p2 * p1 * p0 * c0)
- …. 第N位进位
- cN = gN-1 + pN-1*cN-1 = gN-1 + pN-1 * gN-2 + pN-1 * pN-2 * gN-3+…+ (pN-1 *…* p2 * p1 * p0 * c0)

cN只有3级逻辑，一级p/g，一级与，最后一级为或

并行加法器－－并行进位（或先行进位）

- 所有的函数可以表示成3级 3-level 逻辑.
- But:
  - 输入的门数量将急剧增加
- Target:

速度和尺寸之间的最佳选择

并行进位的特点

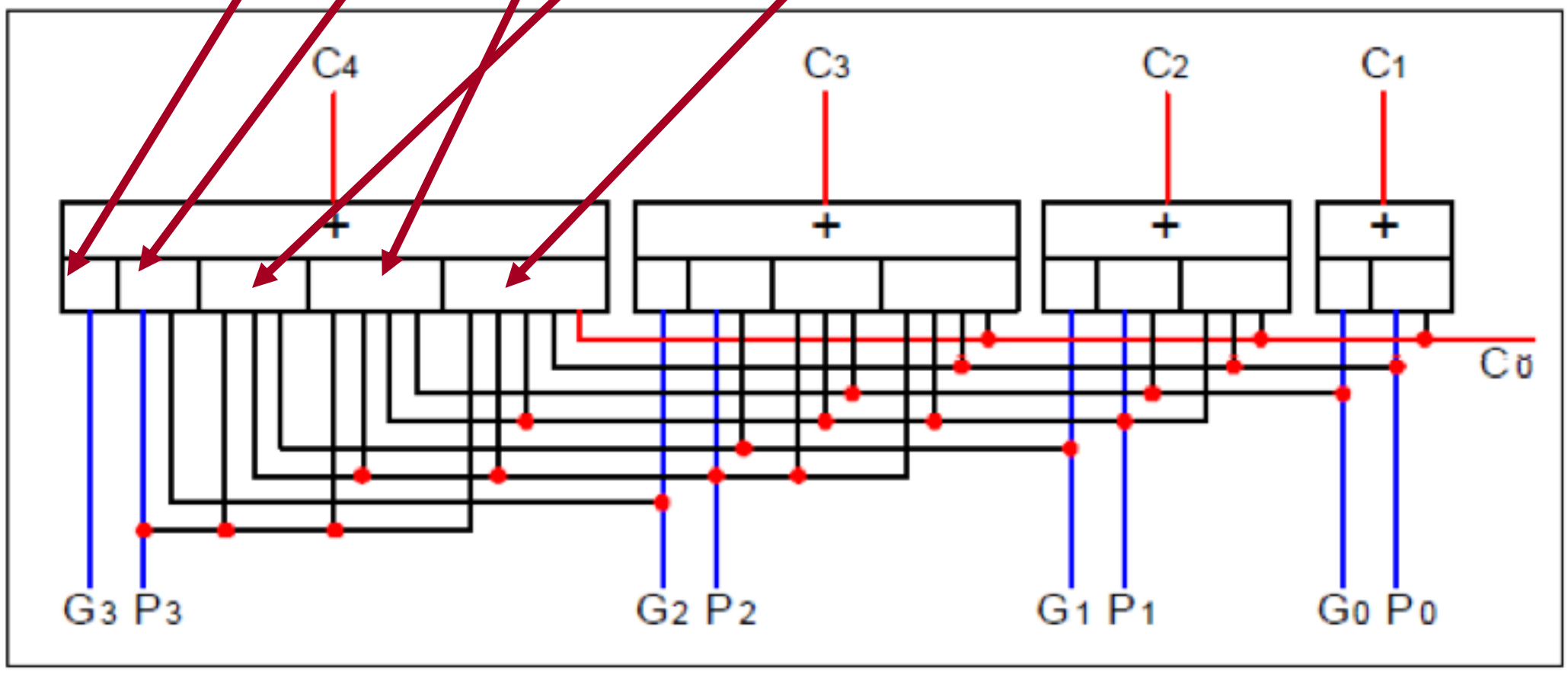
同时产生进位

加法延时缩短

实现相对复杂

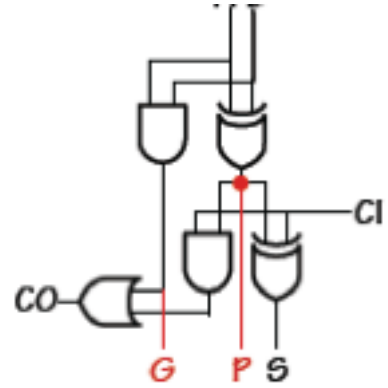
* 并行进位链

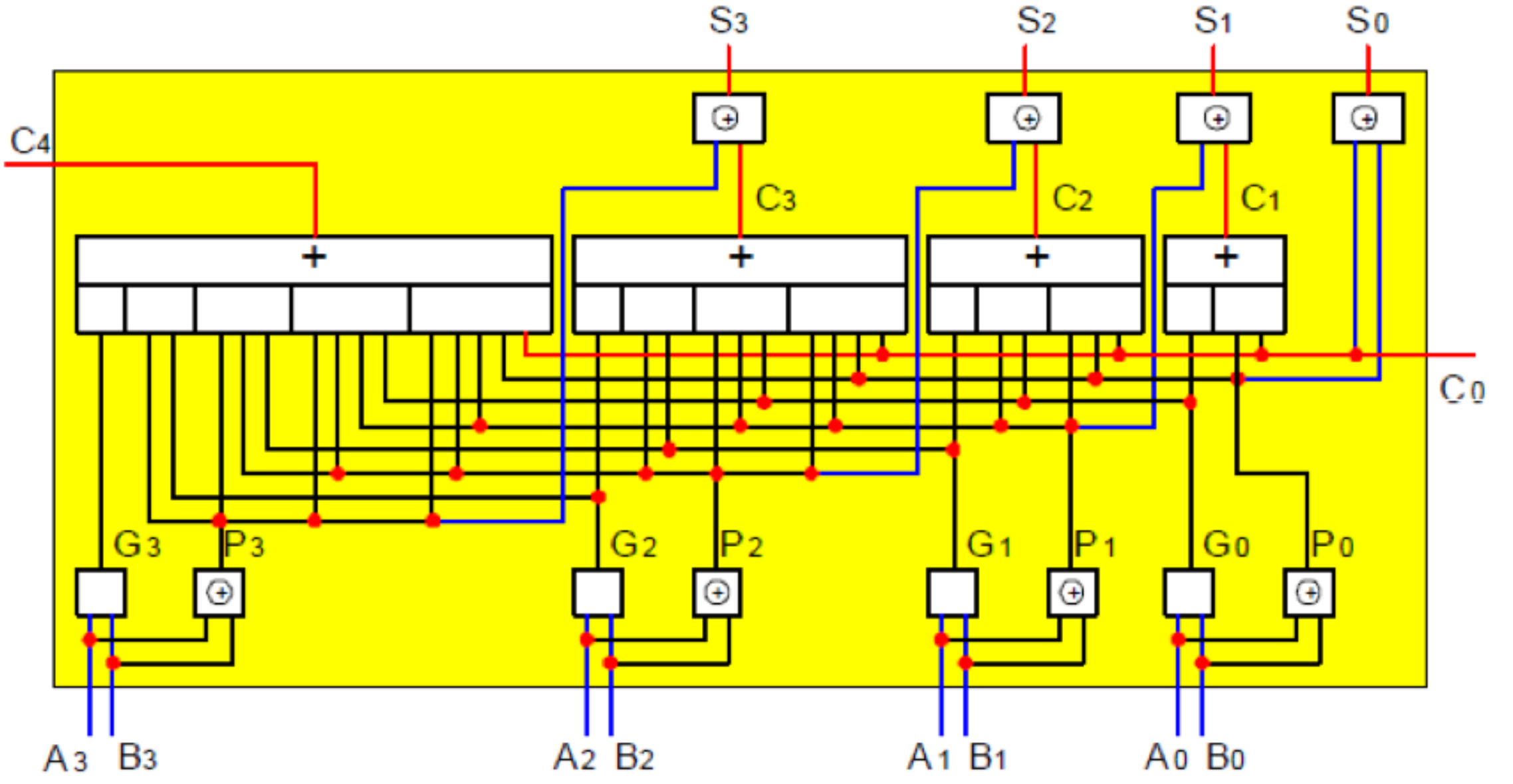
$$c4 = g3 + p3*c3 = g3 + (p3 * g2) + (p3 * p2 * g1) + (p3 * p2 * p1 * g0) + (p3 * p2 * p1 * p0 * c0)$$



设置P和G两个先行进位输出端

* 并行进位加法器





Si=Ci xor Pi

第i位产生的进位 **gi = Ai & Bi**

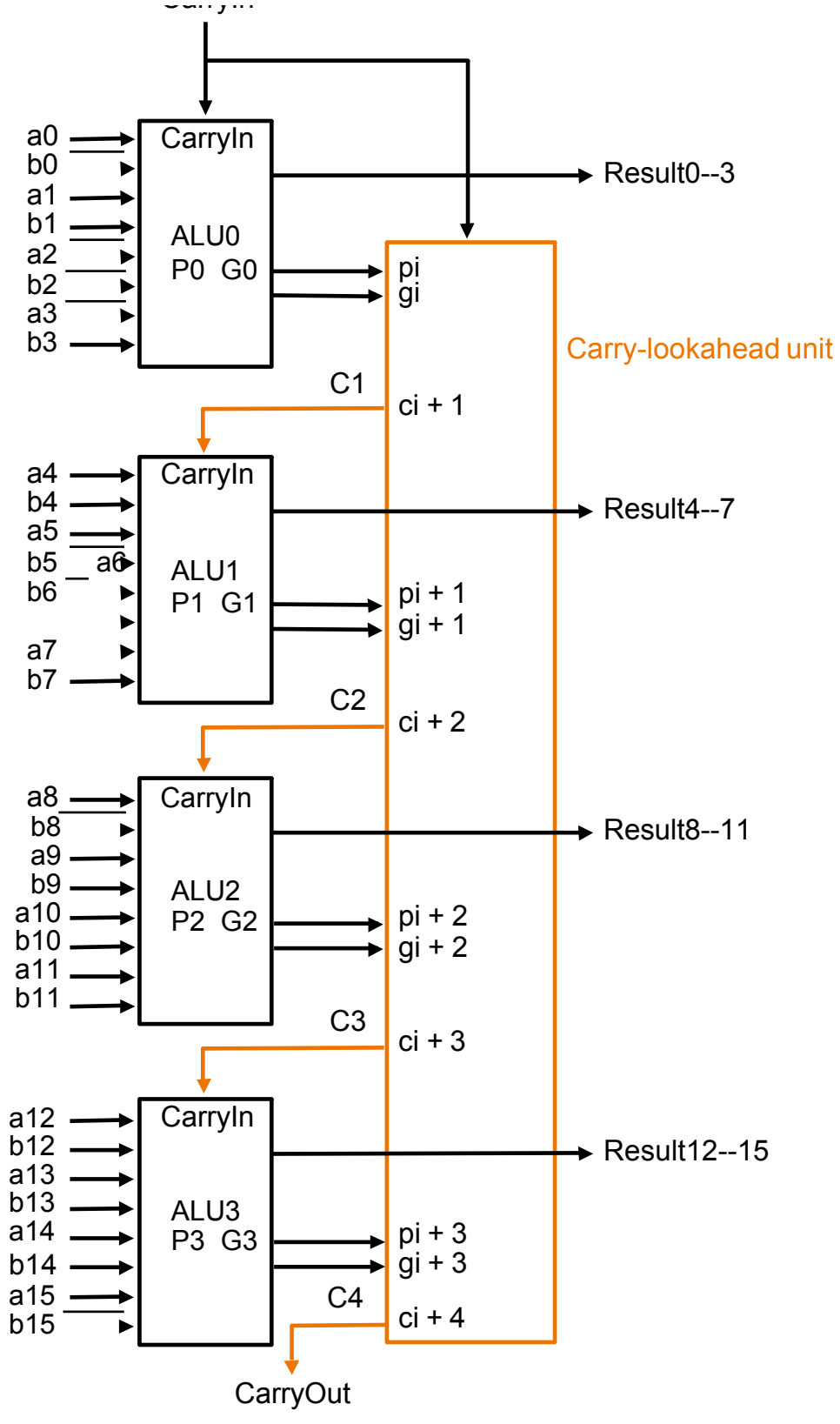
通过第i位的传播进位**(Propagate Carry) pi = Ai or Bi**

局部(级联)先行进位加法器

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

分组并行进位加法器（组内并行，组间传递）

用 CLA 原理构建一个大加法器



分组并行进位加法器（组内并行，组间串行）

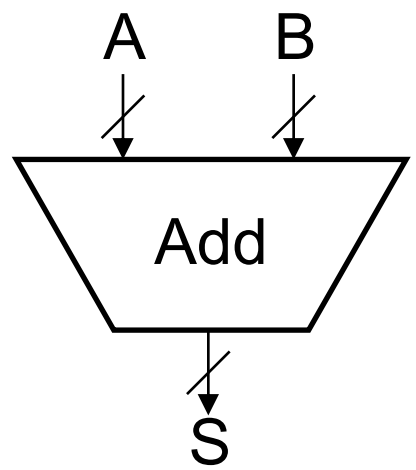
16 bit CLA 加法器太大，可以用4个4位CLA串连

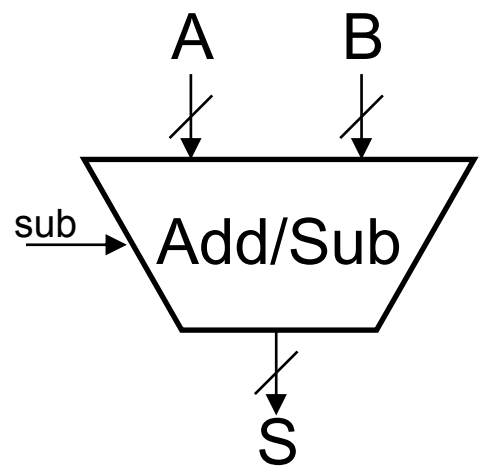
组成方法:

- 一组 4-bit CLA adders
- 把它们连在一起:
  - 在 CLA blocks 之间行波进位
  - Or, 建立另外一级 CLA

Adder Summary

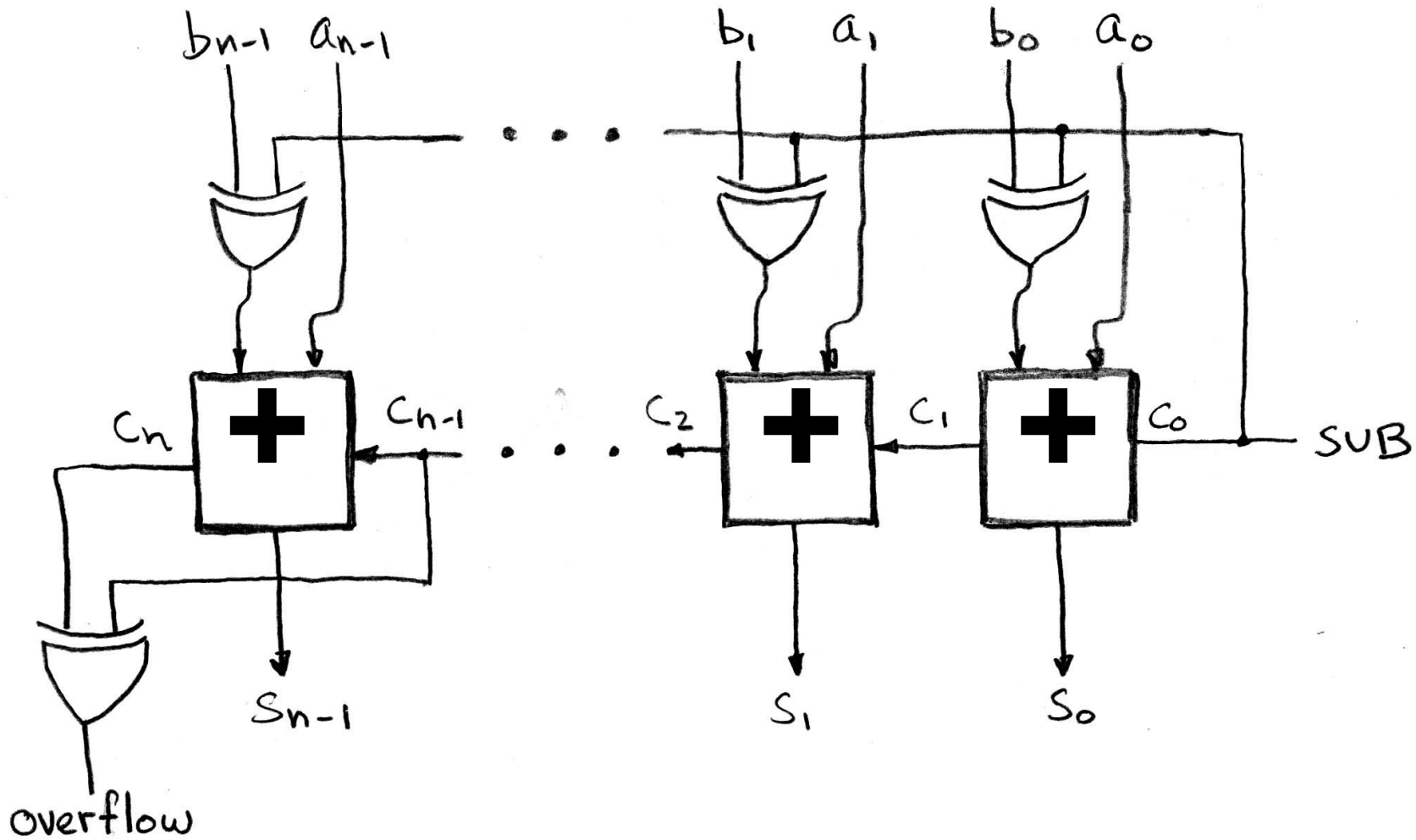
- 加法不仅很常见，而且往往是影响时间的最关键的操作之一
  - 因此，已经开发了广泛的加法器架构，允许设计人员在复杂性（在门的数量方面）与性能之间进行权衡。





加法器/减法器为一体：减法“求反加一”

x	y	XOR(x,y)
0	0	0
0	1	1
1	0	1
1	1	0



XOR 用作条件求反!

SUB=0 加法

SUB=1 减法

条件码

* 算术运算单元通常需要输出4位额外信息:

  - Z (zero):运算结果等于零标志
    - ➤ big NOR gate
  - N (negative): 运算结果小于零标志
    - ➤ S_{N-1}
  - C (carry): 运算结果有进位, e.g., “1 + (-1)”
    - ➤ Carry from last FA
  - V (overflow): 运算结果溢出标志
    - ➤ precisely: $V = A_{i-1}B_{i-1}\overline{N} + \overline{A}_{i-1}\overline{B}_{i-1}N$

    -or-

    $V = CO_{i-1} \oplus CI_{i-1}$

A 和 B 比较判断需要条件码:

符号数比较:

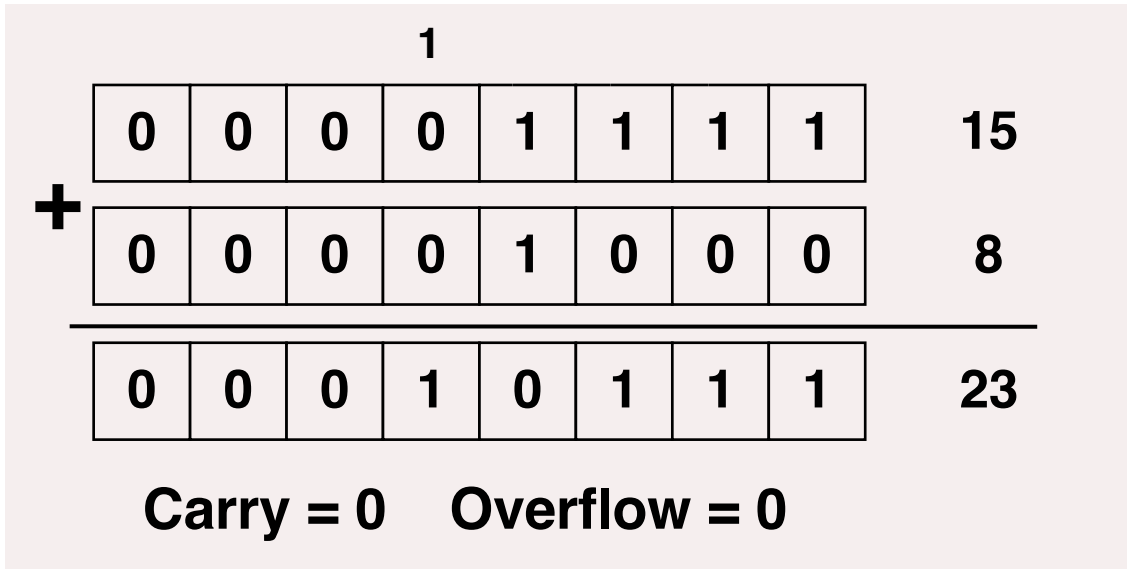
LT	**N⊕V**
LE	**Z+(N⊕V)**
EQ	**Z**
NE	**~Z**
GE	**~(N⊕V)**
GT	**~(Z+(N⊕V))**

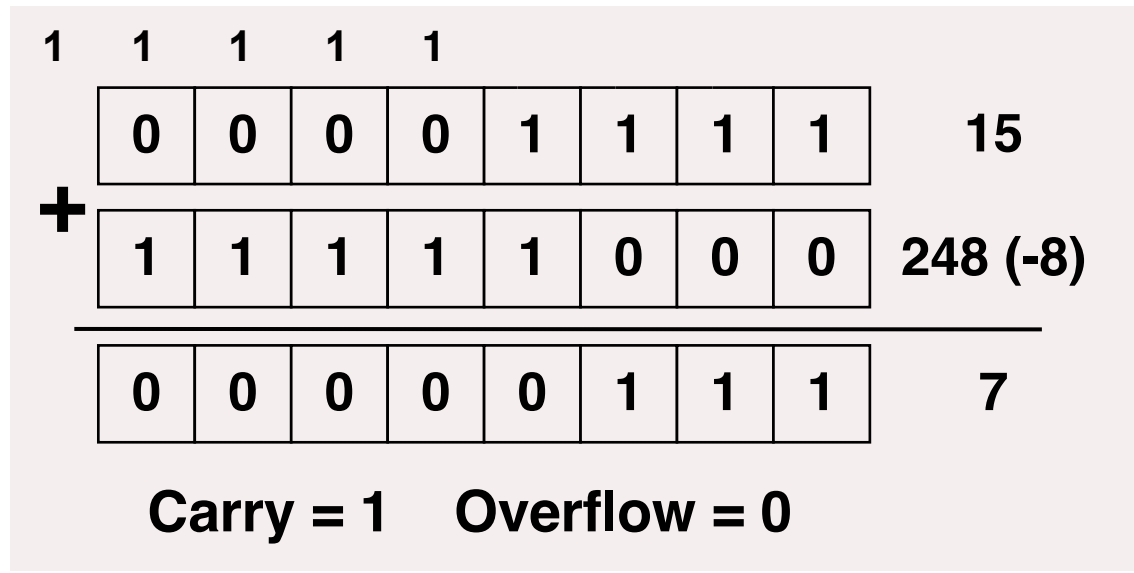
无符号数比较:

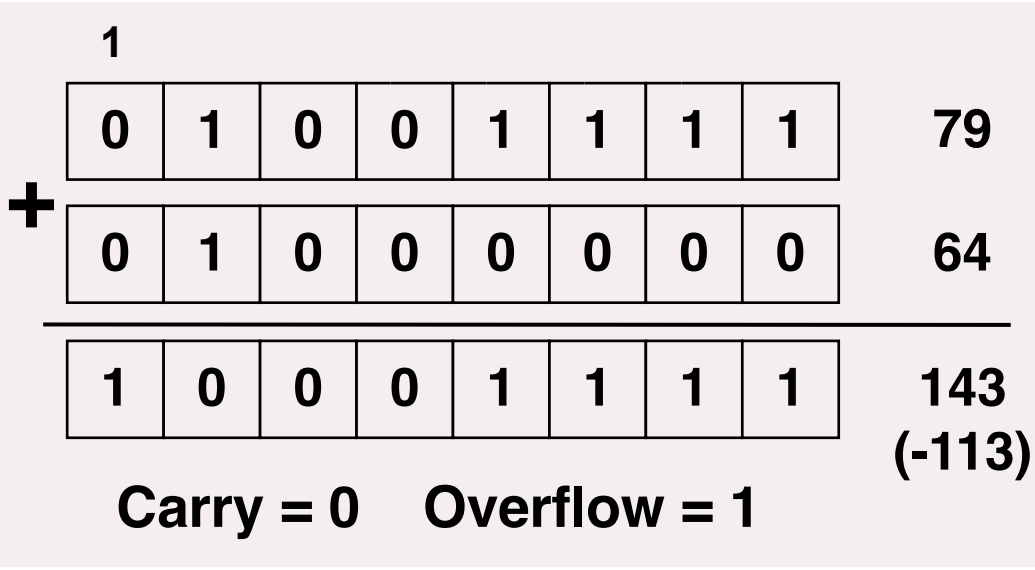
LTU	**C**
LEU	**C+Z**
GEU	**~C**
GTU	**~(C+Z)**

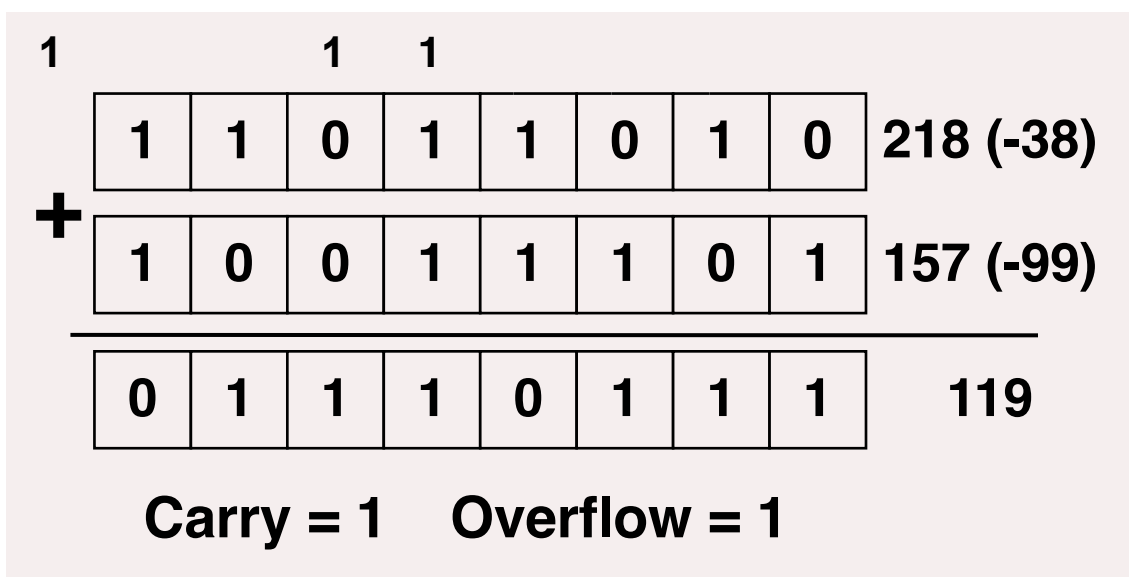
进位与溢出举例

- 可以有溢出，而没进位，反之亦然
- 如下四种可能的情况(Examples are 8-bit numbers)









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!

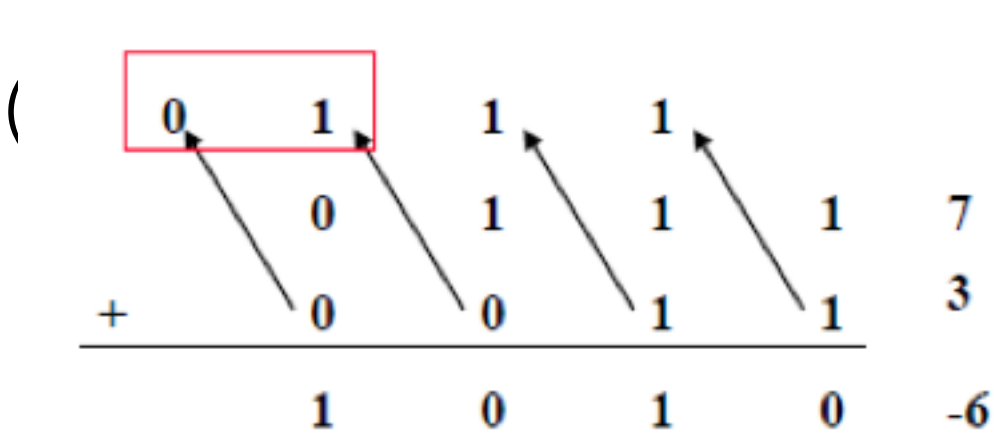
➤ 当进行下列加法时，出现溢出:

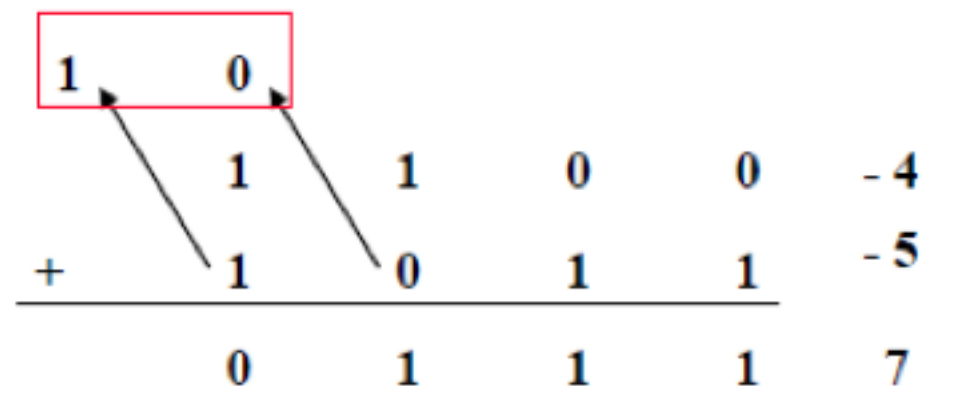
•两个正数相加，结果为负

•两个负数相加，结果为正

➤ 溢出可以用如下方法检测:

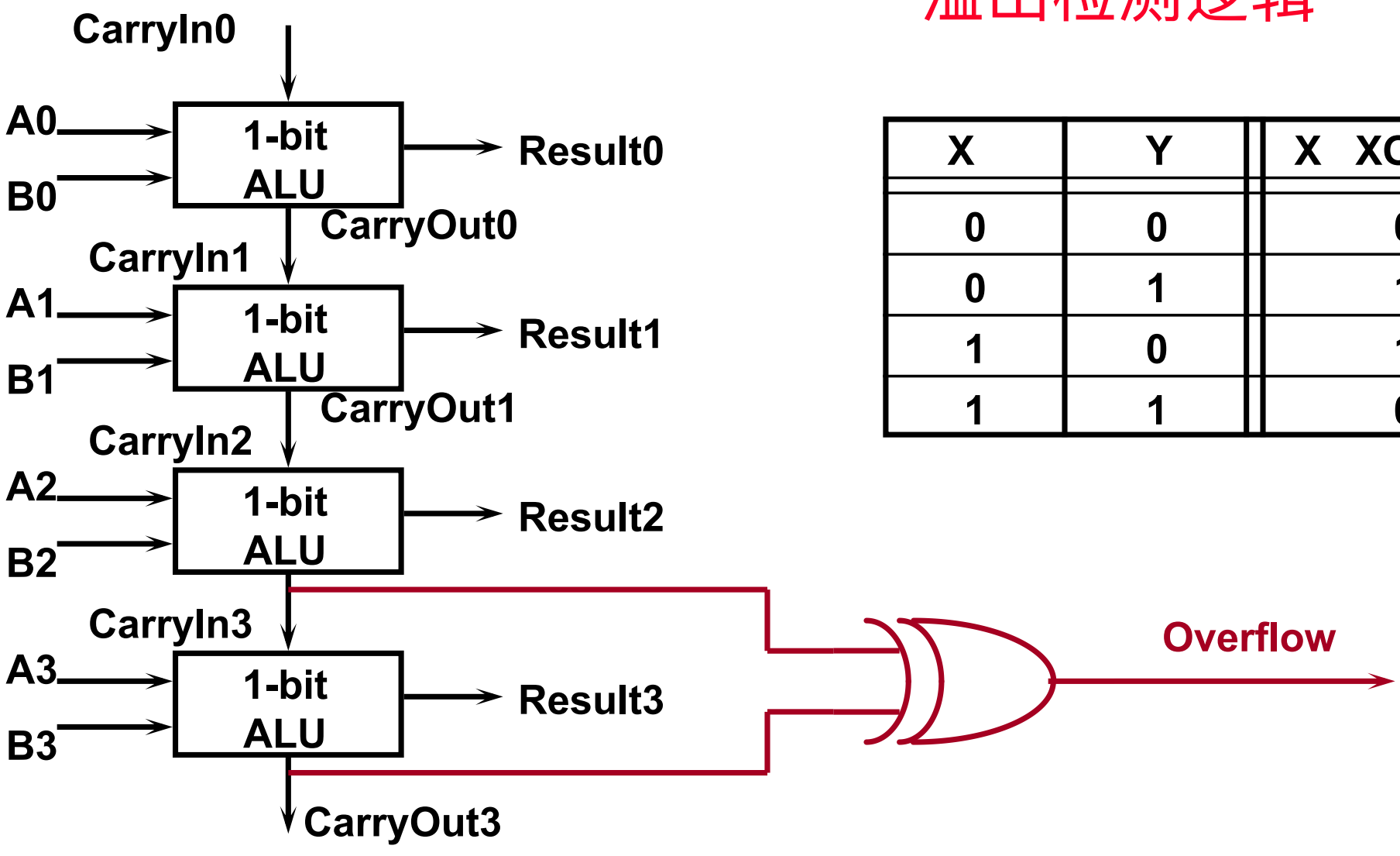
•输入到最大位的进位 != 从最大位输出的进位





Overflow Detection Logic

* Overflow = CarryIn[N-1] XOR CarryOut[N-1]

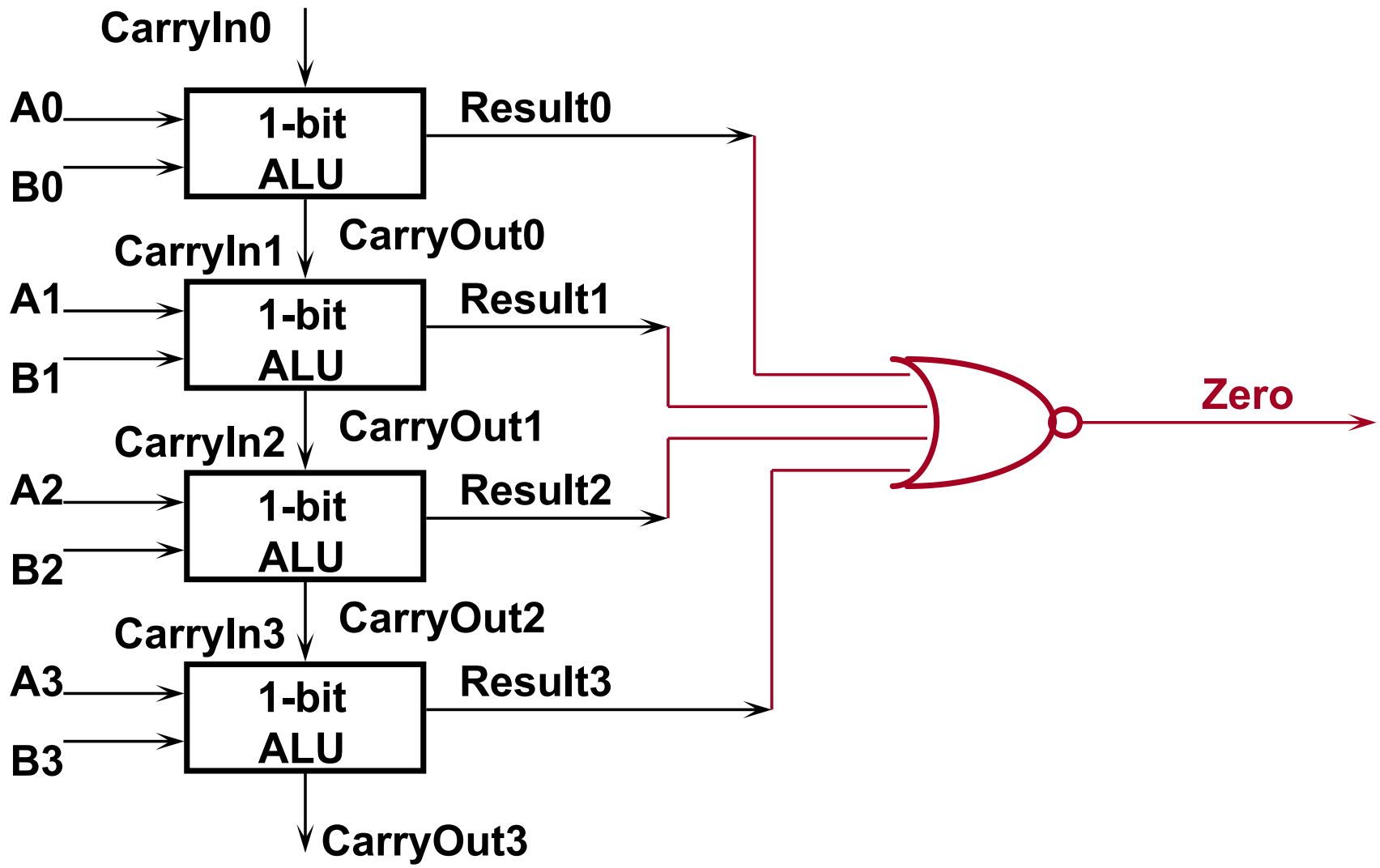


溢出检测逻辑

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

Zero Detection Logic

* 零检测用一个大的异或门(support conditional jump)



Verilog 编程

```
*    `timescale 1ns / 1ps
*    //////////////////////////////////////////////////////////////////////////////////
*    module addsub
*    #(parameter WIDTH=8)            //指定数据宽度参数，缺省值是8
*    (
*        input [(WIDTH-1):0] a,      // 缺省位数由参数WIDTH决定
*        input [(WIDTH-1):0] b,
*        input  sub,                 // =1为减法
*        output [(WIDTH-1):0] sum,
*        output cf,                  // 进位标志
*        output ovf,                   // 溢出标志
*        output sf,                  // 符号标志
*        output zf                   // 为0标志
*        );
*    wire [(WIDTH-1):0] subb,subb1;
*          wire cf2;    // 进位
*          assign subb1 = b ^ {WIDTH{sub}};  // 对于减法是取反
*          assign subb  =  subb1 + sub;   // 对于减法是加1，sub=1 减法)sub=0 加法）
*
*           assign {cf2,sum} = a + subb;
*           assign sf = sum[WIDTH-1];
*           assign zf = (sum == 0) ? 1 : 0 ;
*           assign cf = cf2 ^ sub;
*           assign ovf = (a[WIDTH-1] ^ sum[WIDTH-1]) & (subb[WIDTH-1] ^ sum[WIDTH-1]);

*    endmodule
```

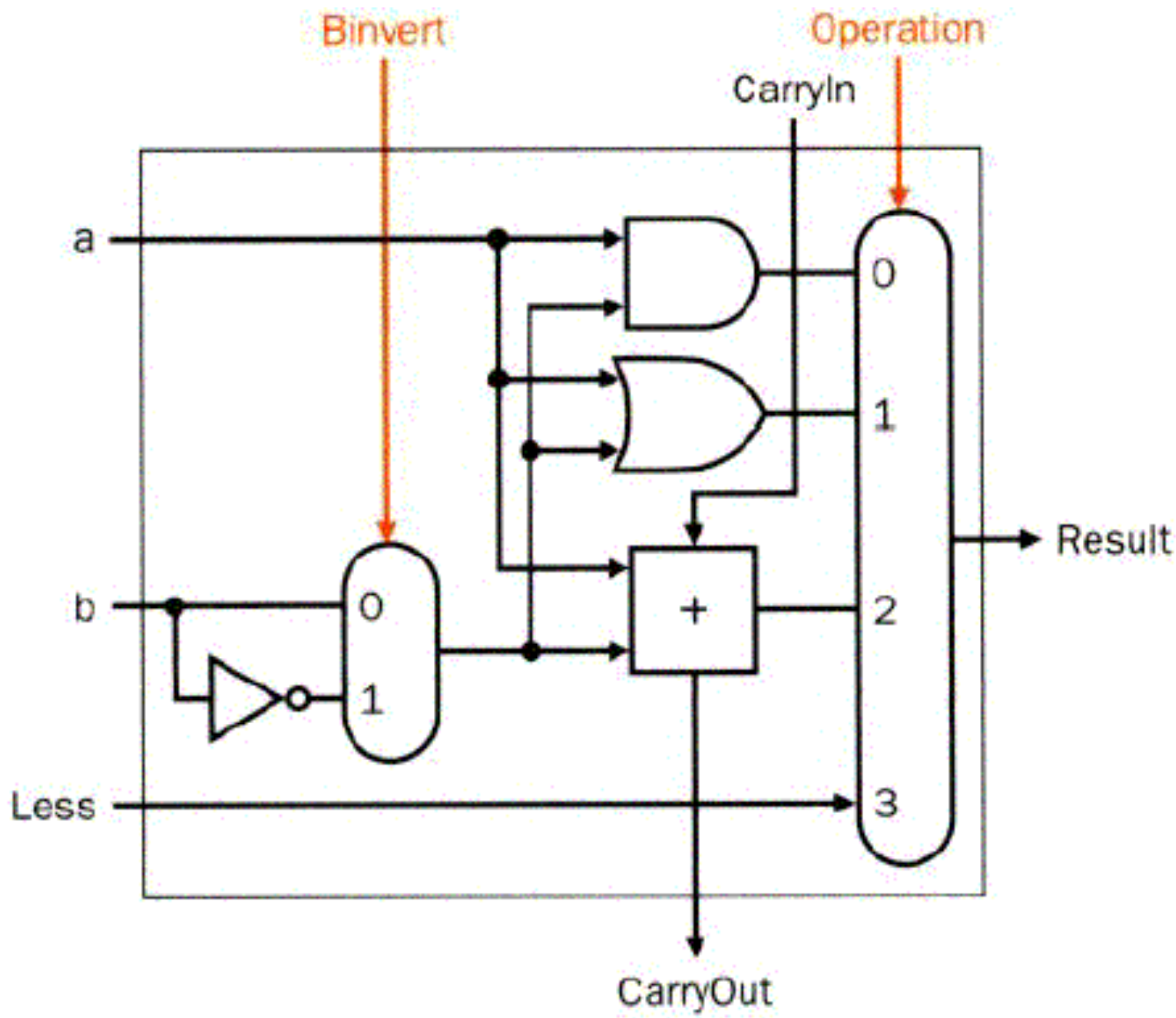
Set on Less Than （小于判断）

- Functions
  - AND
  - OR
  - Add
  - Subtract

- 少了比较: comparison 比较
  - Slt rd,rs,rt
  - If rs < rt, rd=1, else rd=0 $N \oplus V$
  - 除了最低位，所有位为零
  - rs - rt, 如果结果是负的，则 rs < rt
  - 使用符号位作为判断

Extended 1 bit ALU

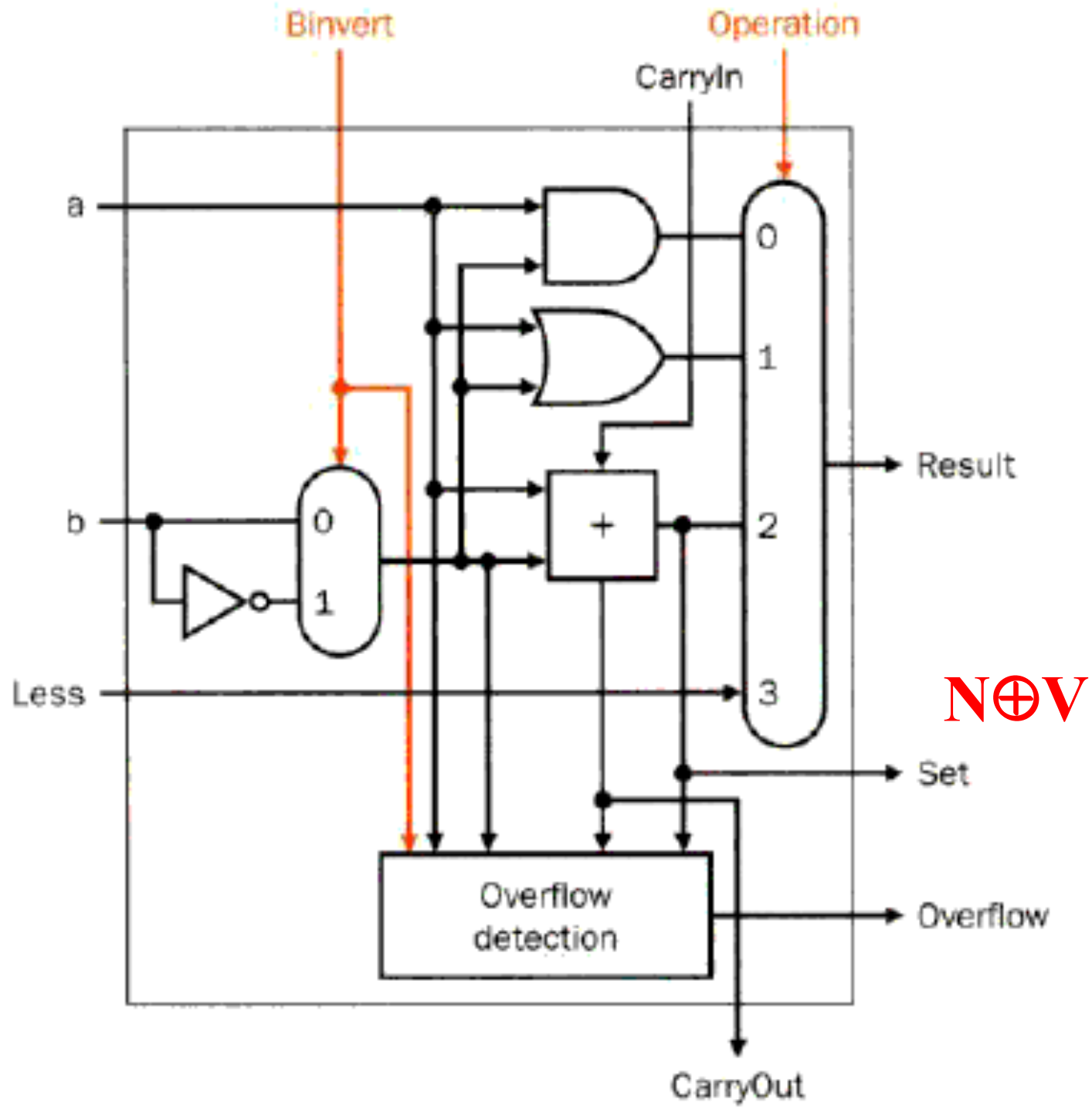
* ALU bit with input for Less data



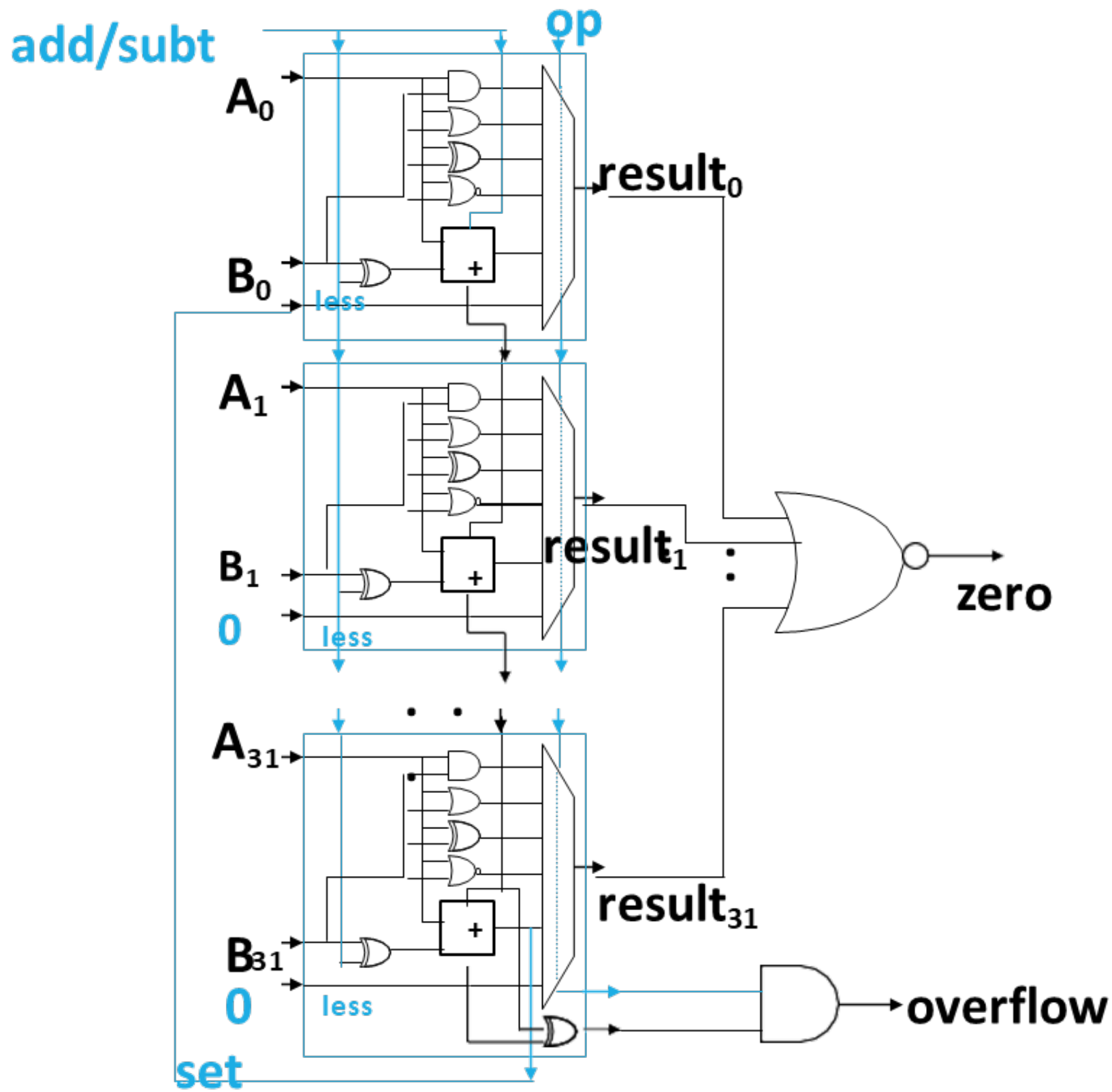
AND
OR
Add
Subtract
less

Most significant bit

- Set for comparison
- Overflow detect



ALU组合到一起


add/subt
op
A_0
B_0
less
+
$result_0$
A_1
B_1
0
less
+
$result_1$
A_{31}
B_{31}
0
less
+
$result_{31}$
set
zero
overflow

Shifter Design

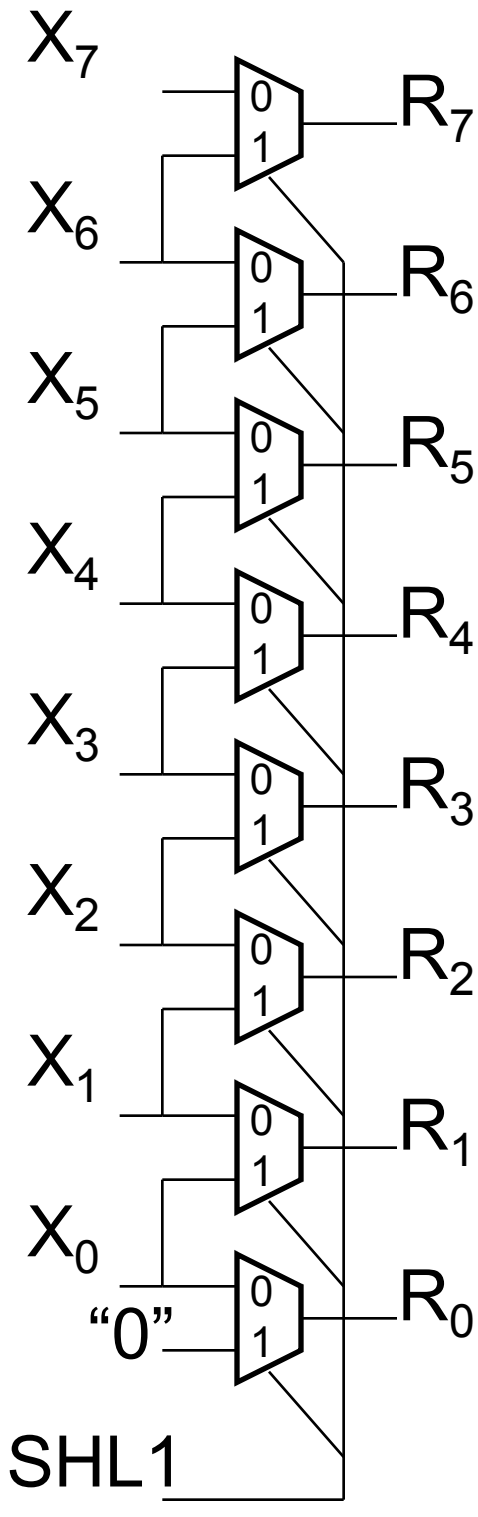
移位可以用于乘法或除法运算

- 左移N bits 相当于数字乘以 2^N
  - Ex: 00001 << 2 = 00100 $(1 \times 2^2 = 4)$
  - Ex: 11101 << 2 = 10100 $(-3 \times 2^2 = -12)$

- 算数右移 N 位 相当于数字除以 2^N
  - Ex: 01000 >>> 2 = 00010 $(8 \div 2^2 = 2)$
  - Ex: 10000 >>> 2 = 11100 $(-16 \div 2^2 = -4)$

Shifting Logic

- 移位是常用的操作
  - 作用于所有位
  - 用于对齐
  - 用于“捷径”算术运算
    - X << 1　等价于 2*X
    - X >> 1　等价于 X/2



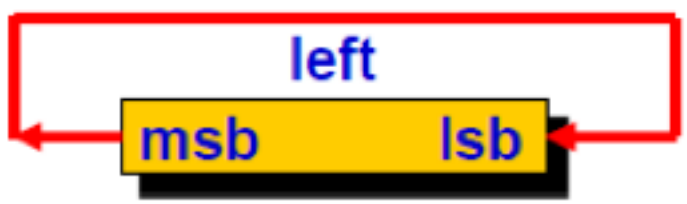
- For example:
  - $X = 20_{10} = 00010100_2$
  - 左移:
    - $(X << 1) = 00101000_2 = 40_{10}$
  - 右移:
    - $(X >> 1) = 00001010_2 = 1010$
  - 有符号数的算数右移:
    - $(-X >>> 1) = (11101100_2 >>> 1) = 11110110_2 = -10_{10}$

三种移位器

逻辑(*logical*) — 移入的数值总是 "0"

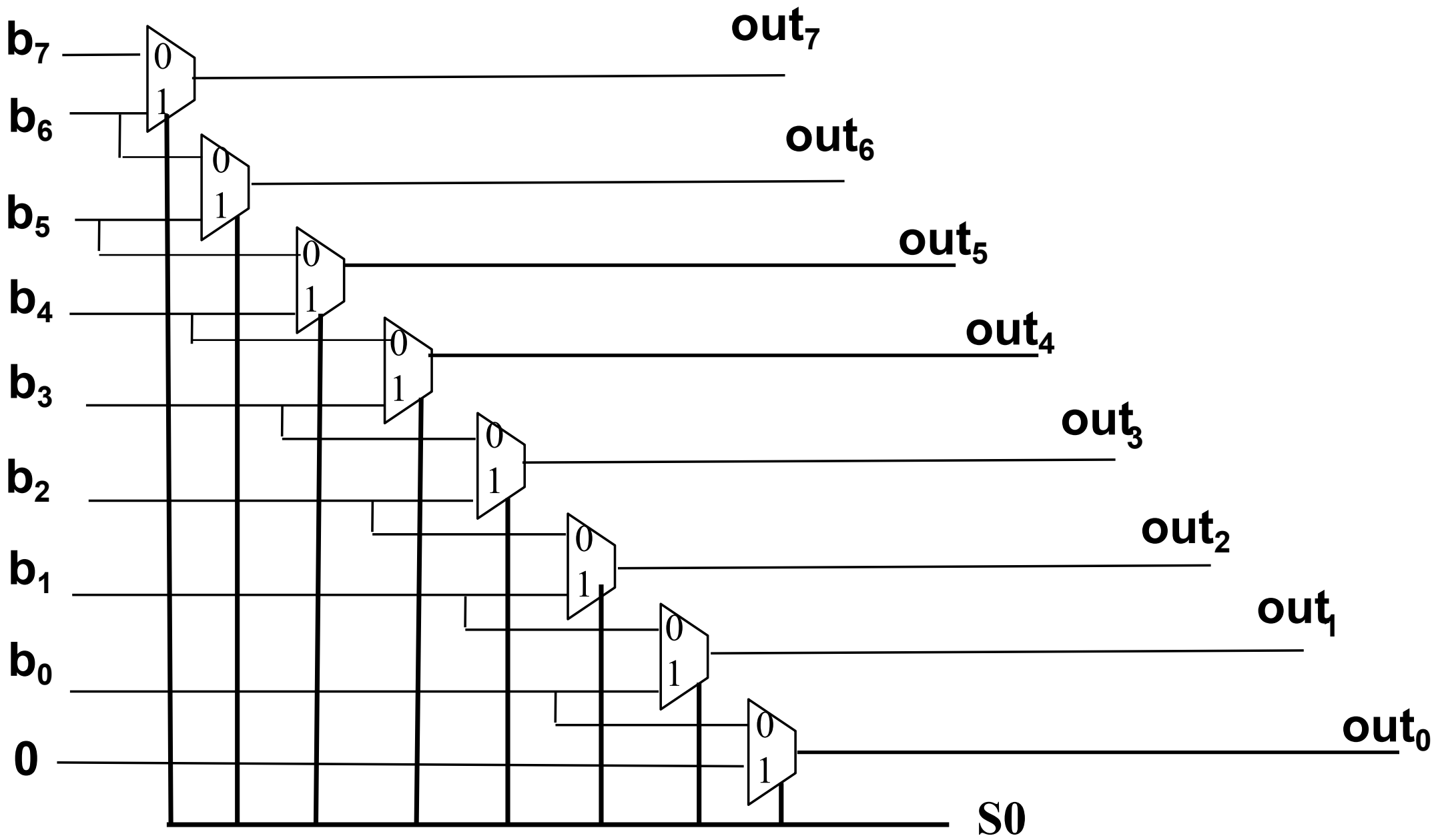
算术(*arithmetic*) — 在右移时，进行符号扩展

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



移位实现 Let’s simplify

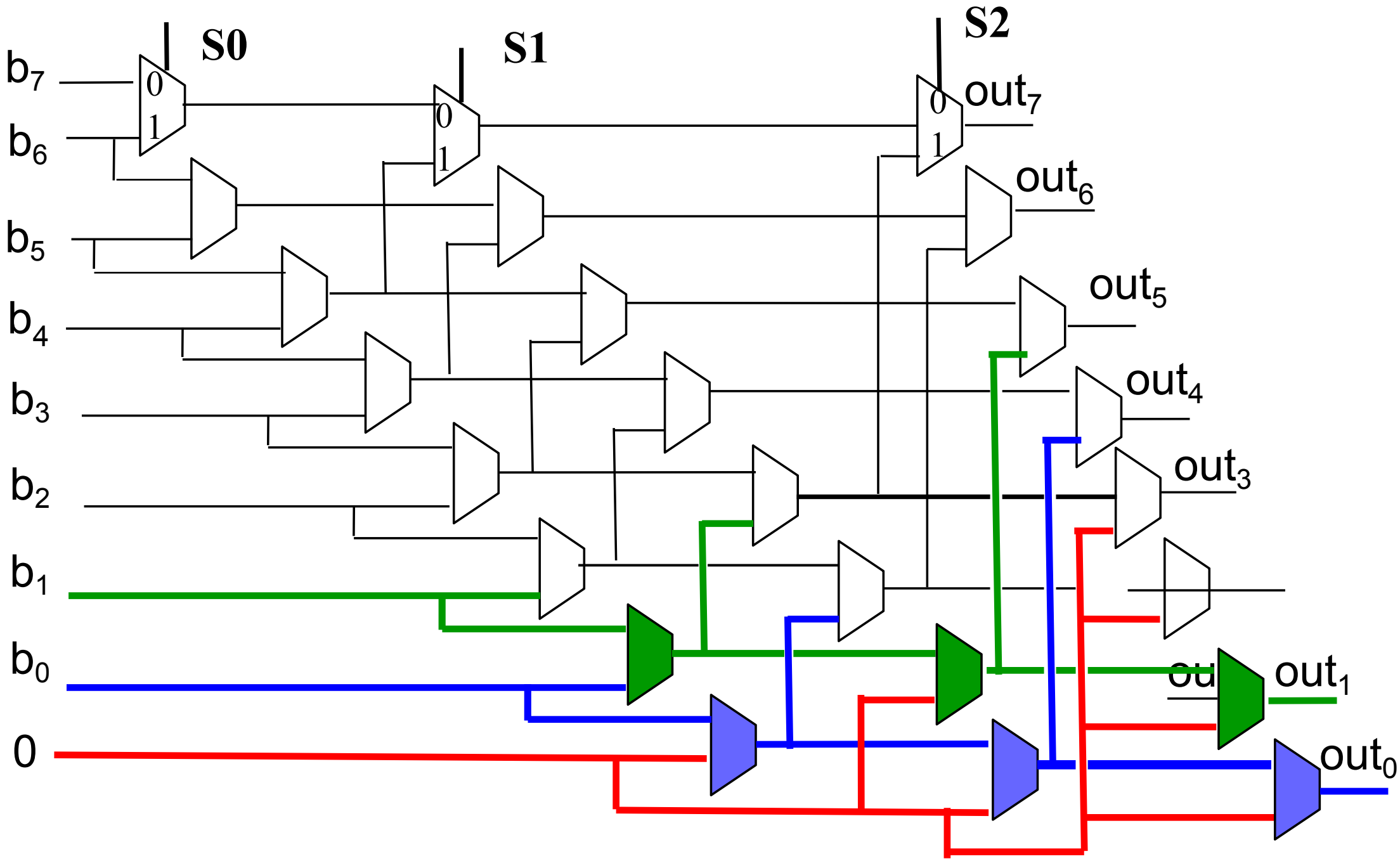
8 位数移位 1 位数（每位的选择器选择移位量）
S0=0,不移位，S0=1，移1位



Now shifted by 3-bit number

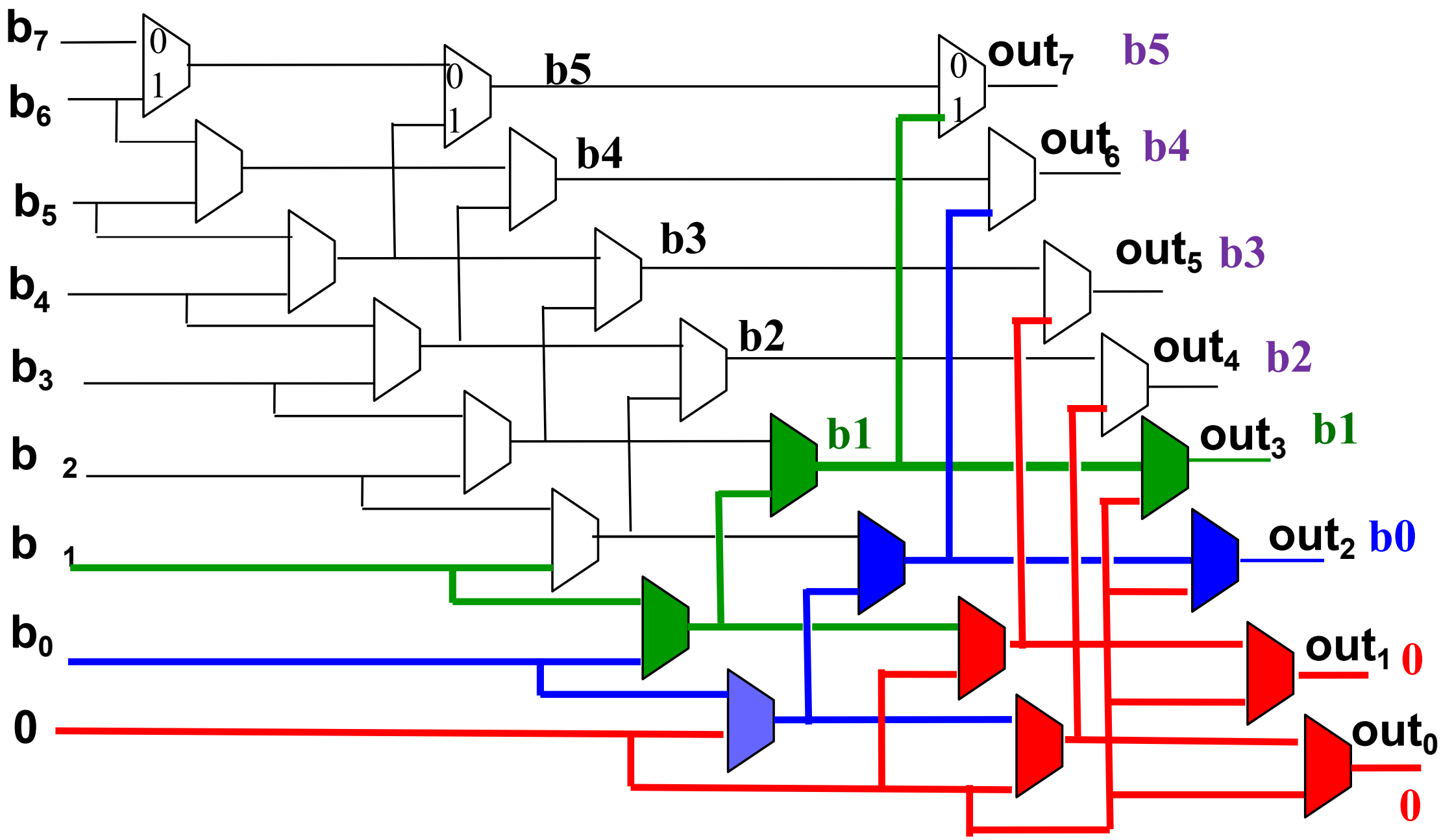
编码的移位控制线 S0-2

- Shifter in action: shift by 000 (all muxes have S=0)



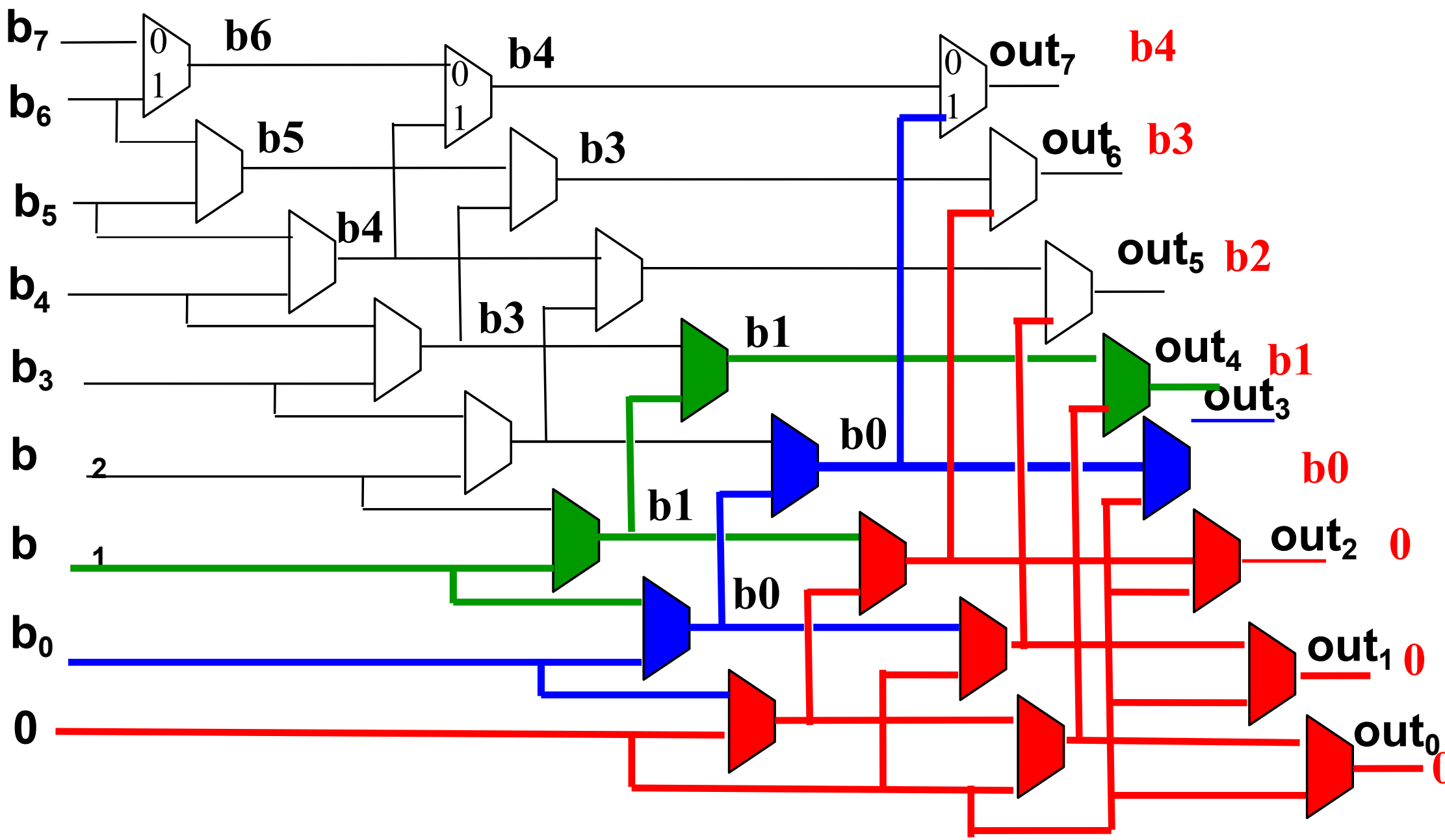
Now shifted by 3-bit number

- **Shifter in action: shift by 010**
  - **From L to R: S = 0, 1, 0, 左移2位**



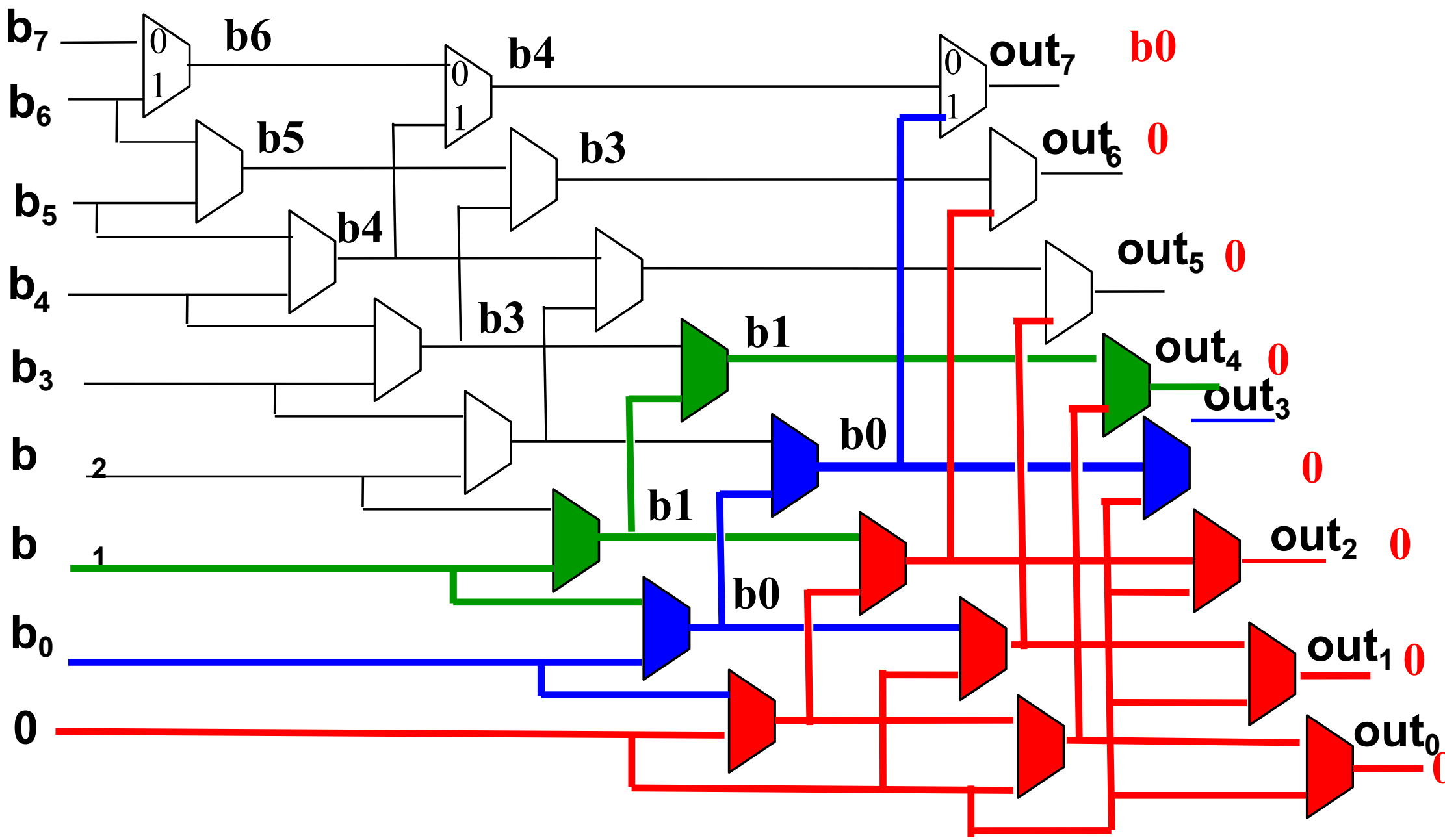
Now shifted by 3-bit number

- Shifter in action: shift by 011
  - From L to R: S= 1, 1, 0 (移位数倒着数011，移3位)



Now shifted by 3-bit number

- Shifter in action: shift by 111
  - From L to R: S= 1, 1, 1 (左移7位)



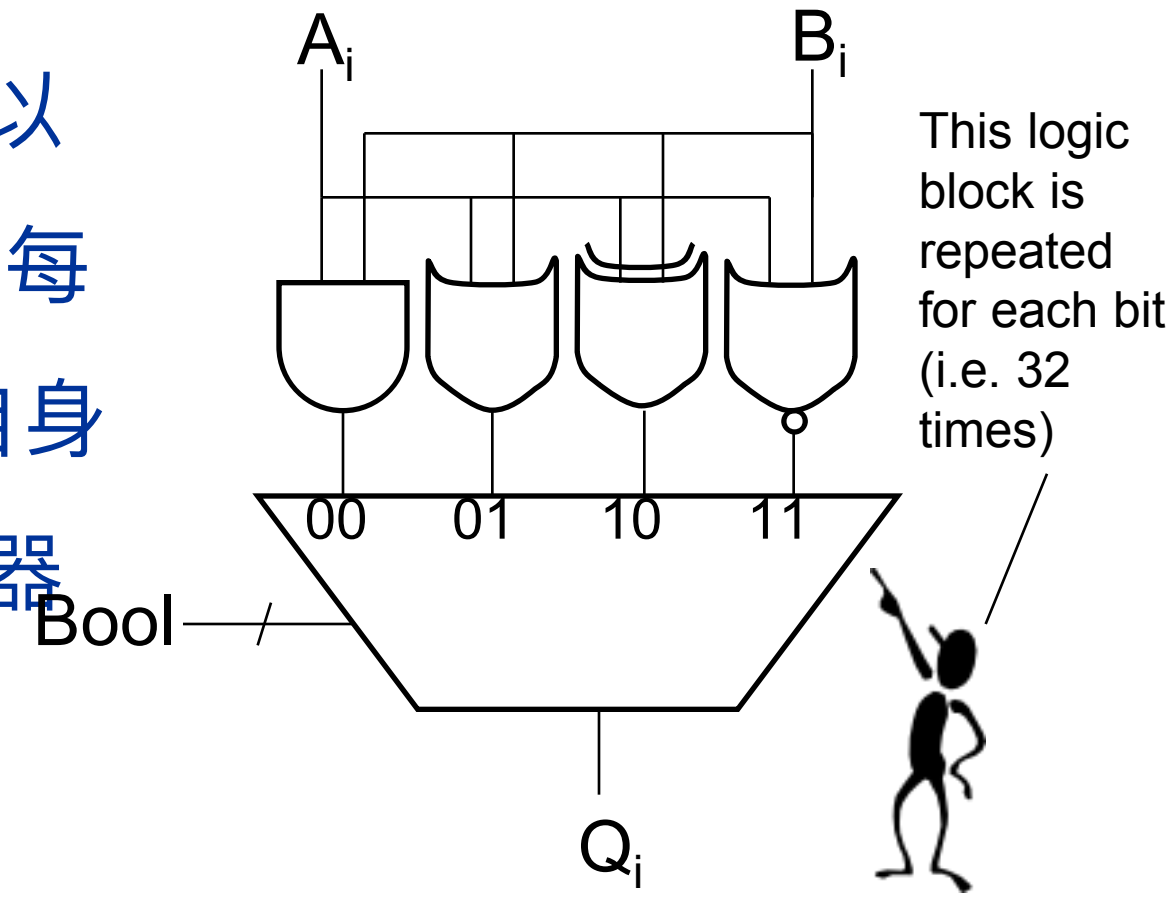
Boolean Operations 布尔运算

* 对位组执行逻辑运算也很有用?

- ANDing is useful for “masking” off groups of bits.
  - ex. 10101110 & 00001111 = 00001110 (mask selects last 4 bits)
- ANDing is also useful for “clearing” groups of bits.
  - ex. 10101110 & 00001111 = 00001110 (0’s clear first 4 bits)
- ORing is useful for “setting” groups of bits.
  - ex. 10101110 | 00001111 = 10101111 (1’s set last 4 bits)
- XORing is useful for “complementing” groups of bits.
  - ex. 10101110 ^ 00001111 = 10100001 (1’s invert last 4 bits)
- NORing is useful for.. uhm…
  - ex. 10101110 # 00001111 = 01010000 (0’s invert, 1’s clear)

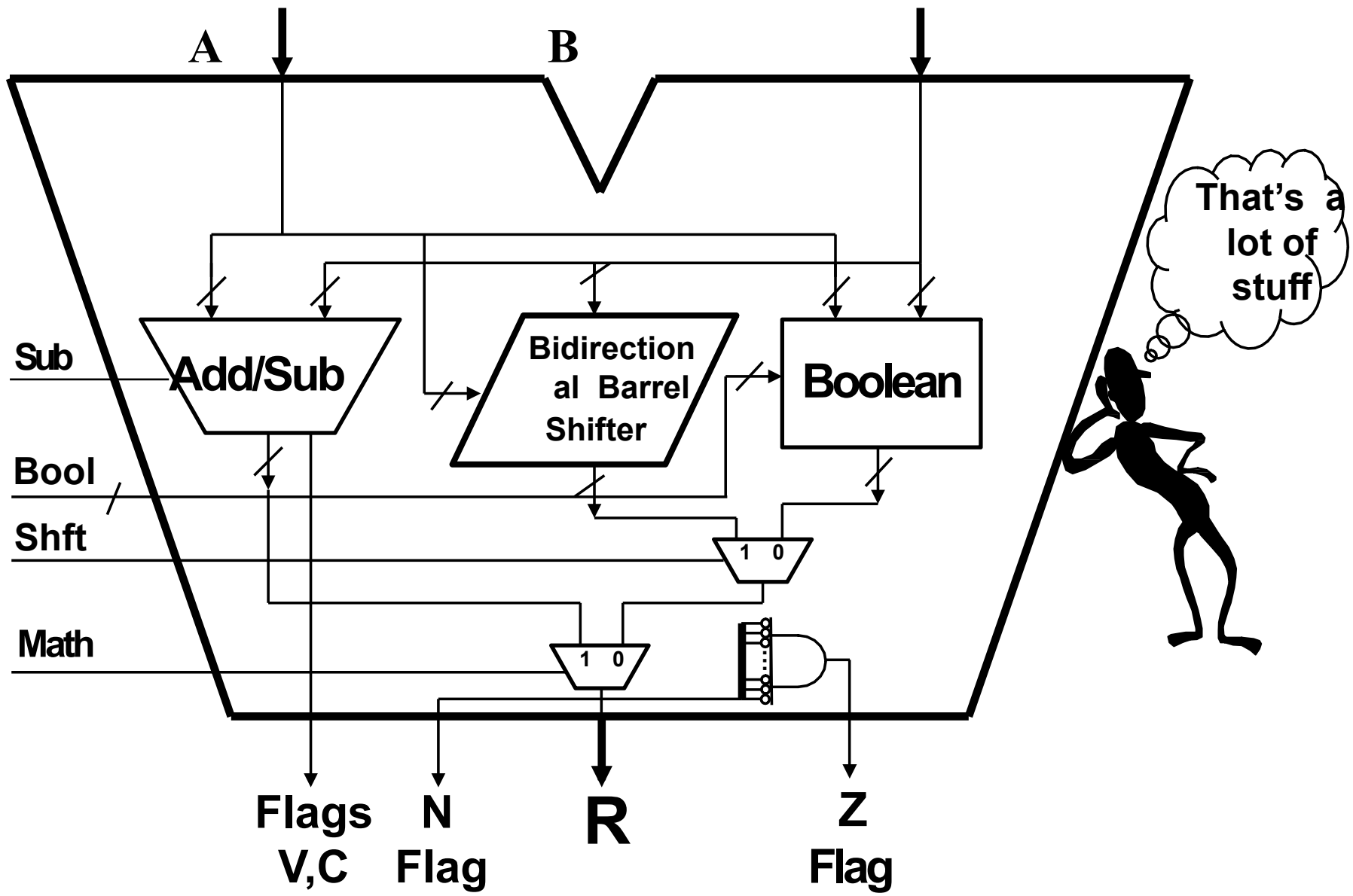
Boolean Unit

* 使用门电路和多路复用器来选择函数很简单.
  - 由于位之间没有互连，因此可以在每个位置简单地复制该单元.每位需要7个门电路，一个用于自身功能，大约3个用于4选一选择器



An ALU, at Last

我们给计算机的“数学中心”起了一个特殊的名字——算术逻辑单元。 对我们来说，它只是一个大盒子!



An ALU, at Last

我们给计算机的“数学中心”起了一个特殊的名字——算术逻辑单元。 对我们来说，它只是一个大盒子!