



清华大学
Tsinghua University

第一单元 第四讲

算术运算及其电路实现

刘卫东

计算机科学与技术系

本讲提要



- ❖ 数据表示（复习）
- ❖ 运算器功能
- ❖ 用于实现运算功能的基础逻辑电路
- ❖ ALU设计
- ❖ 算术运算的实现

整数编码的定义



x 为真值 n 为整数的位数

$$[x]_{\text{原}} = \begin{cases} x & 2^n > x \geq 0 \\ 2^n - x & 0 \geq x > -2^n \end{cases}$$

$$[x]_{\text{补}} = \begin{cases} x & 2^n > x \geq 0 \\ 2^{n+1} + x & 0 \geq x \geq -2^n \pmod{2^n} \end{cases}$$

$$[x]_{\text{反}} = \begin{cases} x & 2^n > x \geq 0 \\ (2^{n+1} - 1) + x & 0 \geq x > -2^n \pmod{2^n} \end{cases}$$

补码的性质



补码与真值的对应

补码求真值

$$N = -b_{n-1} * 2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$$

真值求补码：

◆ 正数的补码是绝对值的原码

◆ 负数的补码是绝对值原码按位求反后，再在最低位加1

补码的加法运算

加法运算：符号位和数据位同样计算

$$[x+y]_{\text{补}} = [x]_{\text{补}} + [y]_{\text{补}}$$

补码的性质



⊕ $[x]_{\text{补}}$ 与 $[-x]_{\text{补}}$

▣ $[x]_{\text{补}}$ 连同符号位在内，逐位求反，再在最低位加1，即可得 $[-x]_{\text{补}}$

⊕ 补码减法

▣ $[x-y]_{\text{补}} = [x+(-y)]_{\text{补}} = [x]_{\text{补}} + [-y]_{\text{补}}$

⊕ 由 $[x]_{\text{补}}$ 求 $[x/2]_{\text{补}}$

原符号位不变，且符号位与数值位均右移一位

$$[X]_{\text{补}} = \underline{10010} \quad \text{则} \quad [X/2]_{\text{补}} = \underline{110010}$$

补码表示中的符号位扩展



不同位数的整数补码相加减时，

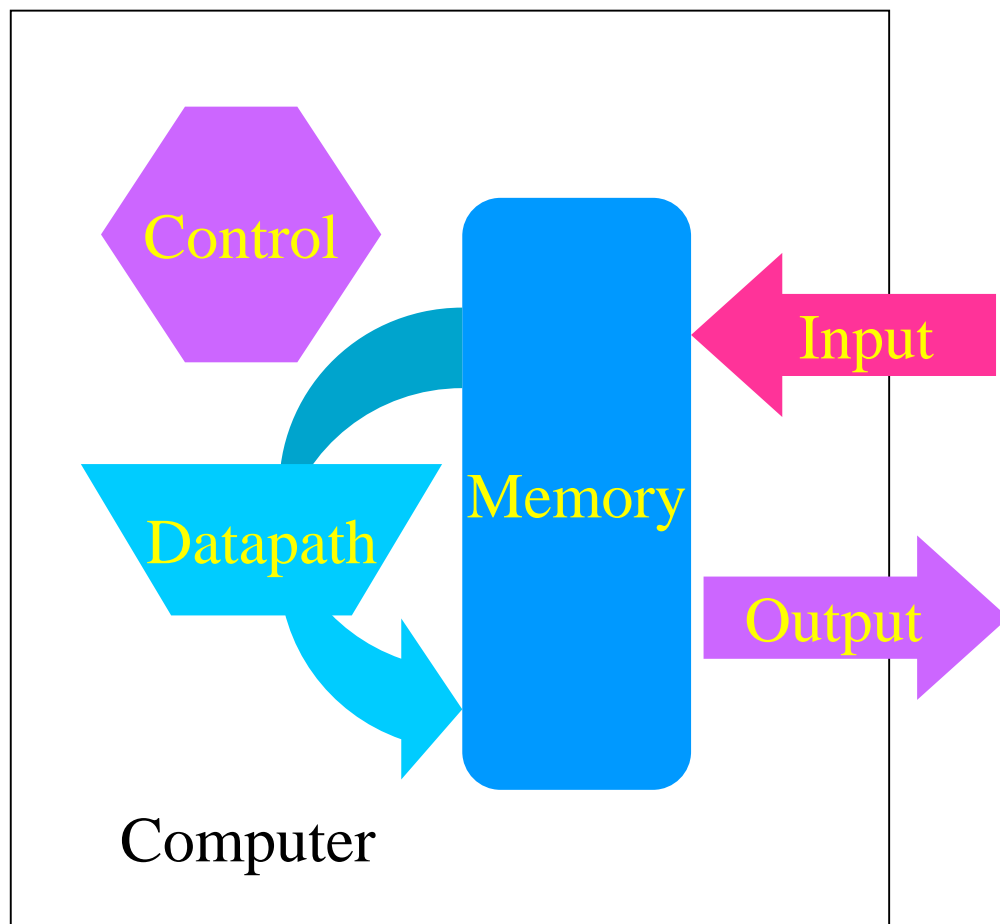
位数少的补码数的符号位向左扩展，
一直扩展到与另一数的符号位对齐。

$$\begin{array}{r} 0101010111000011 \\ + 11111111 \quad 10011100 \\ \hline 0101010101011111 \end{array}$$

$$\begin{array}{r} 0101010111000011 \\ + 00000000 \quad 00011100 \\ \hline 0101010111011111 \end{array}$$



计算机运行机制



✦ Datapath:
完成算术和
逻辑运算，
通常包括其
中的寄存器

运算器基本功能



- ✚ 完成算术、逻辑运算
 - ▣ $+$ 、 $-$ 、 \times 、 \div 、 \wedge 、 \vee 、 \neg
- ✚ 得到运算结果的状态
 - ▣ C、Z、V、S
- ✚ 取得操作数
 - ▣ 寄存器组、数据总线
- ✚ 输出、存放运算结果
 - ▣ 寄存器组、数据总线
- ✚ 暂存运算的中间结果
 - ▣ Q寄存器、移位寄存器
- ✚ 由控制器产生的控制信号驱动

运算器的基础逻辑电路



逻辑门电路

- 完成逻辑运算

加法器

- 完成加法运算

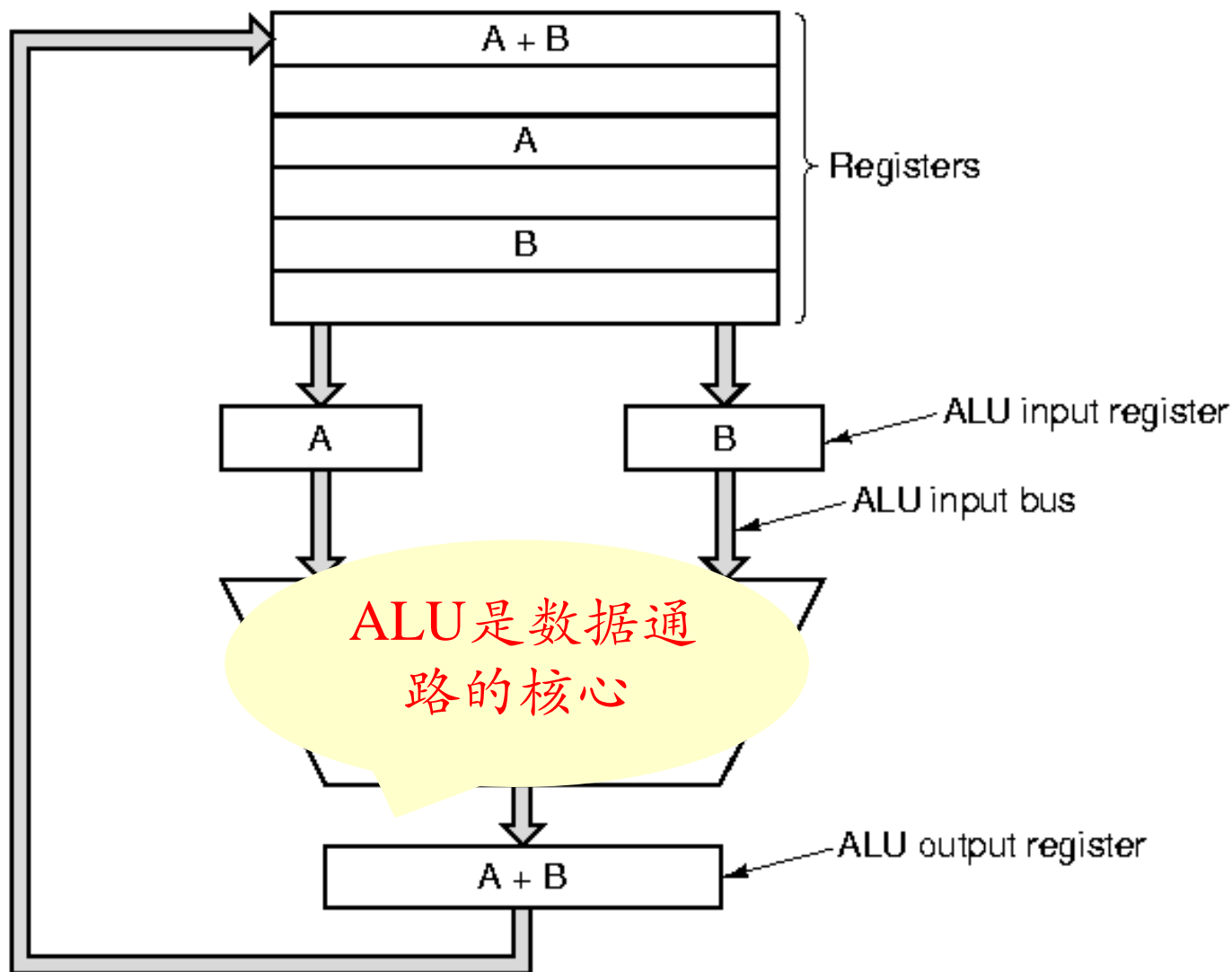
触发器

- 保存数据

多路选择器、移位器

- 选择、连通

数据通路 (Datapath)



ALU功能和设计

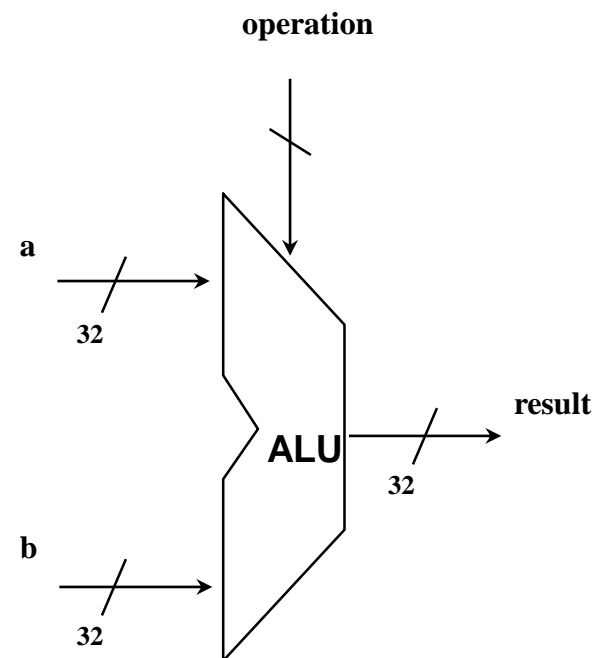


功能

- 对操作数A、B完成算术逻辑运算
- ADD、AND、OR

设计

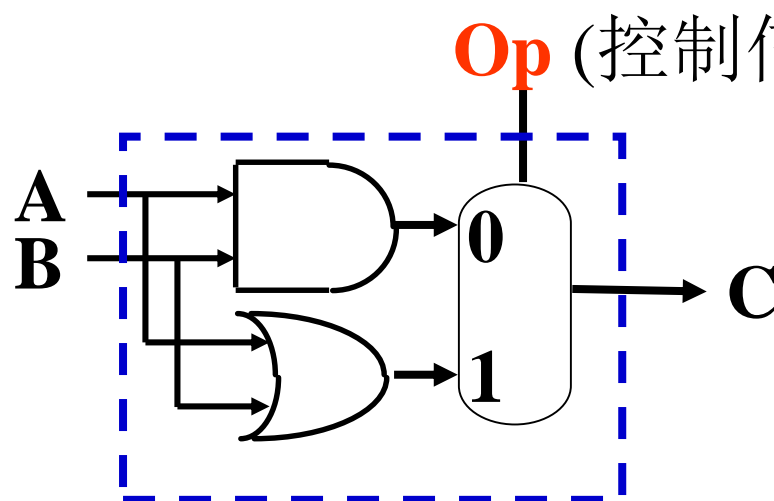
- 算术运算
 - 加法器
- 逻辑运算
 - 与门、或门



1位ALU逻辑运算实现



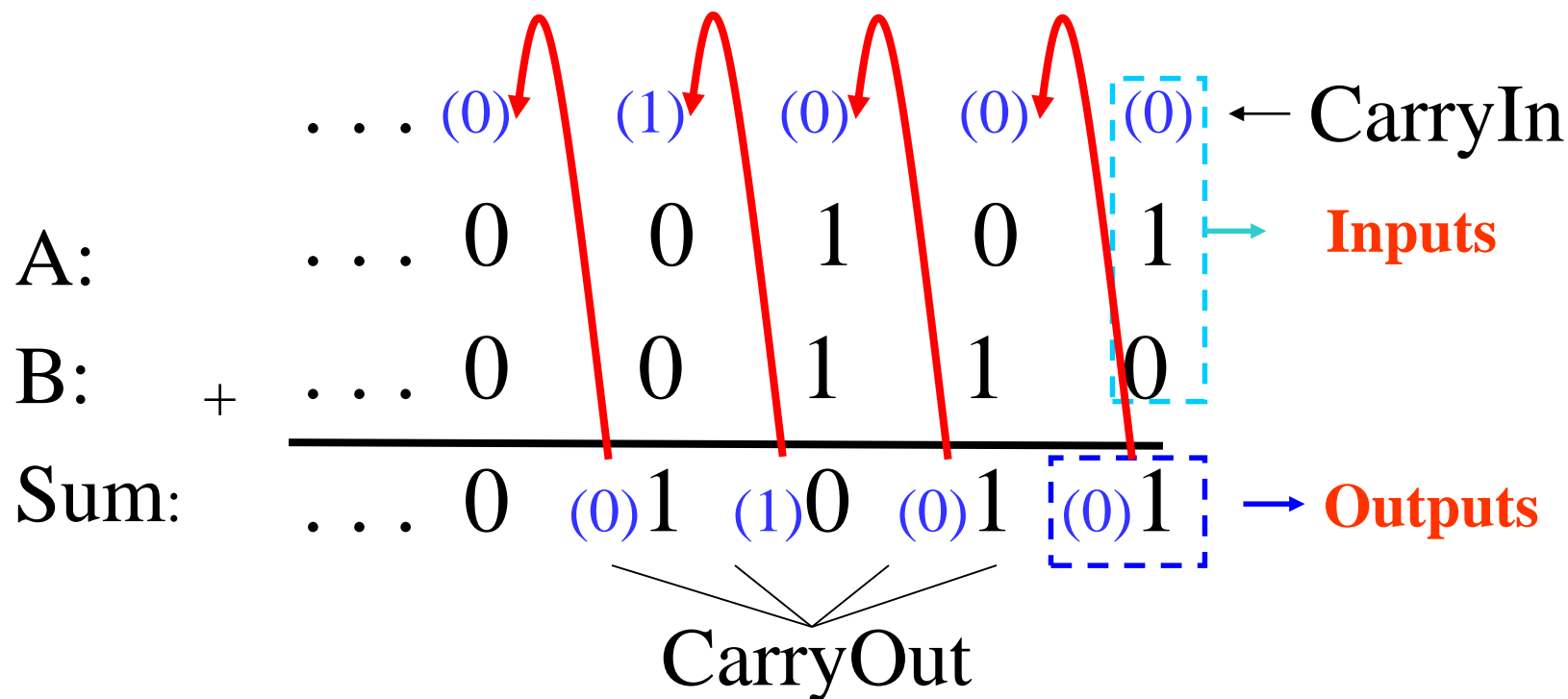
- ❖ 直接用逻辑门实现与和或的功能
- ❖ 多路选择器，通过OP控制信号输出结果



Op (控制信号) 功能表

Op	C
0	A and B
1	A or B

1位ALU加法运算实现



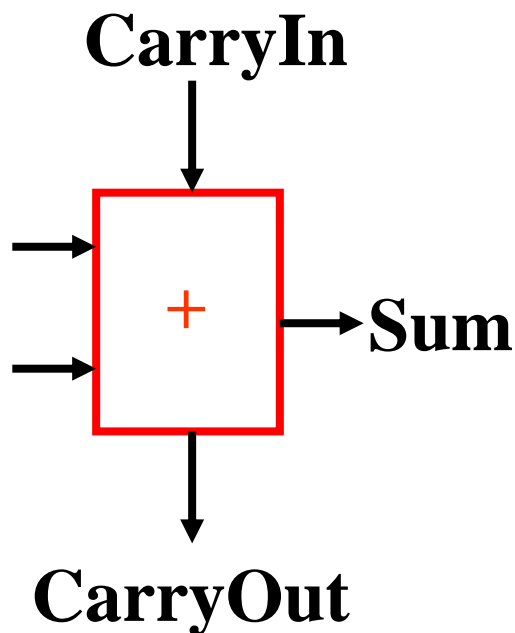
1位的加法:

- 3个输入信号: $A_i, B_i, \text{CarryIn}_i$
- 两个输出信号: $\text{Sum}_i, \text{CarryOut}_i$
- $(\text{CarryIn}_{i+1} = \text{CarryOut}_i)$

1位全加器的设计与实现



Symbol



真值表

A	B	CarryIn	CarryOut	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

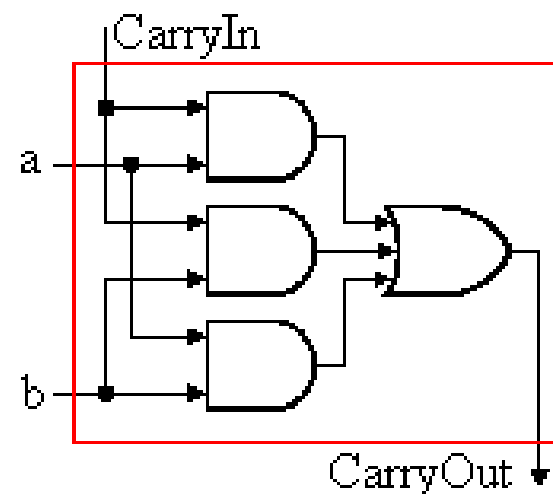
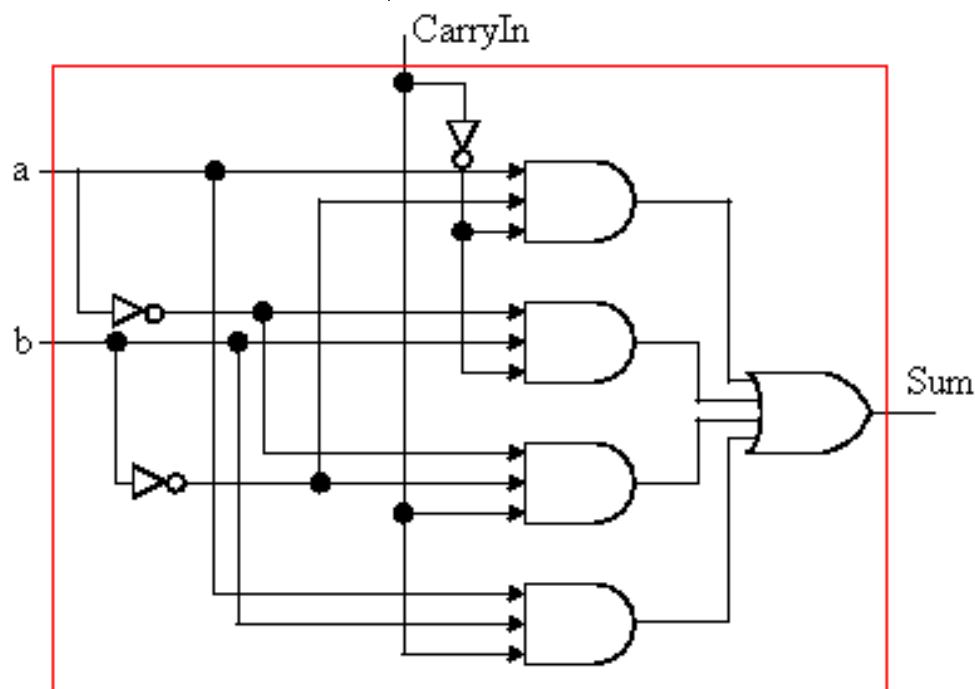
$$\text{CarryOut} = (\mathbf{A'} * B * \text{CarryIn}) + (A * \mathbf{B'} * \text{CarryIn}) + (A * B * \mathbf{\text{CarryIn'}}) + (A * B * \text{CarryIn}) = (B * \text{CarryIn}) + (A * \text{CarryIn}) + (A * B)$$

$$\text{Sum} = (\mathbf{A'} * \mathbf{B'} * \text{CarryIn}) + (\mathbf{A'} * B * \mathbf{\text{CarryIn'}}) + (A * \mathbf{B'} * \mathbf{\text{CarryIn'}}) + (A * B * \text{CarryIn})$$

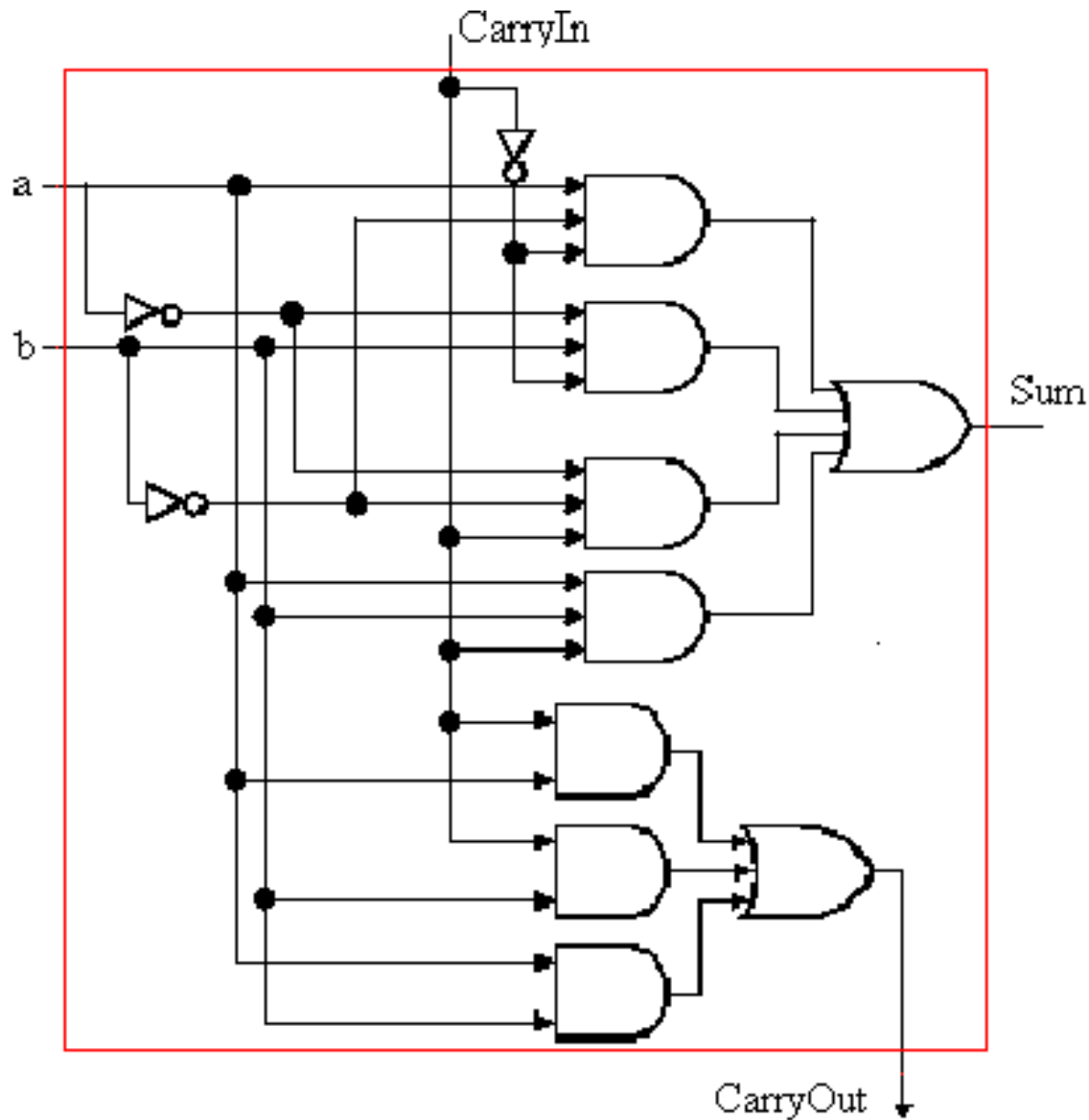
全加器设计与实现（续）



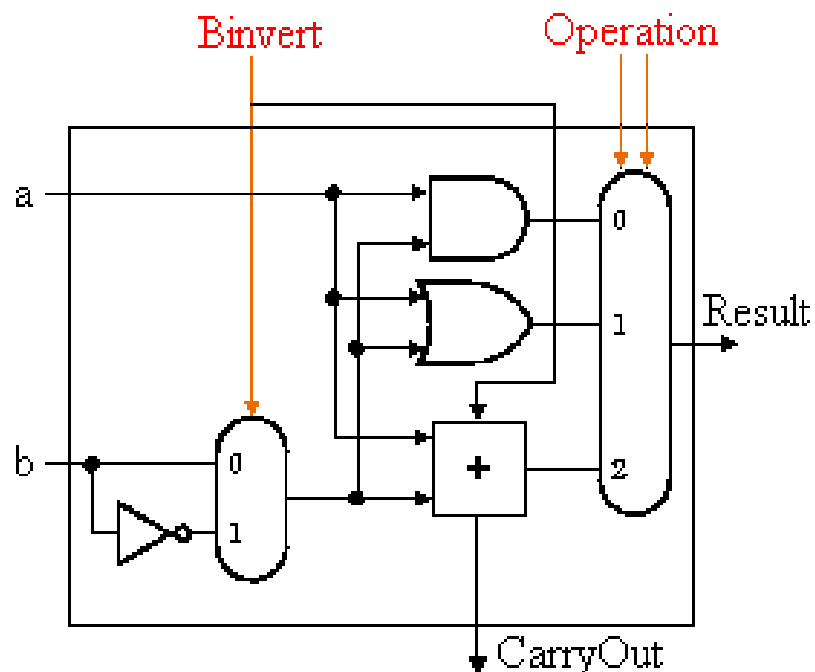
- 用逻辑门实现加法，求Sum
- 用逻辑门求 CarryOut
- 将所有相同的输入连接在一起



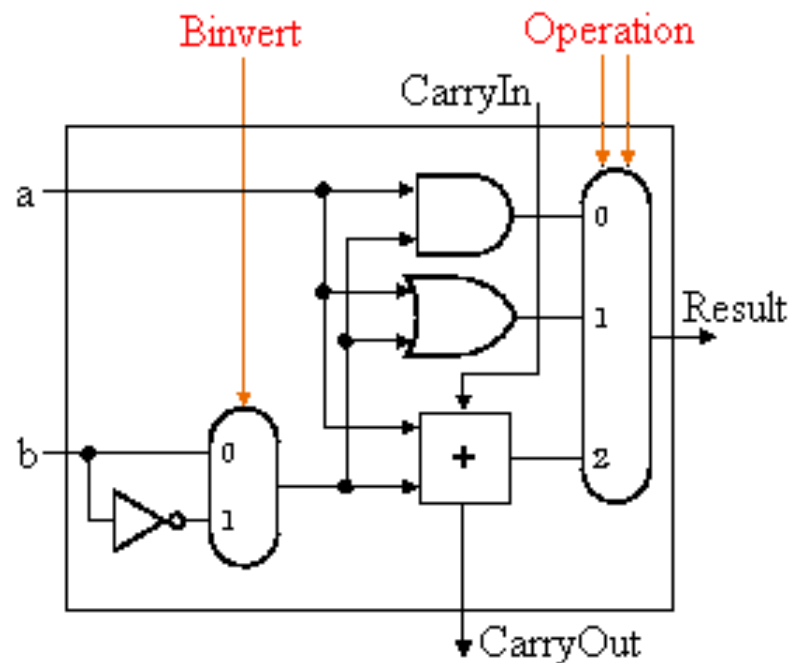
全加器



1位的ALU



最低位



其他位

1位ALU的设计过程



✚ 确定ALU的功能

▣ 与、或、加

✚ 确定ALU的输入参数

✚ 根据功能要求，得到真值表，并写出逻辑表达式

✚ 根据逻辑表达式实现逻辑电路

✚ 如何实现4位的ALU呢？

4位ALU实现方式



思路1:

- 同1位ALU设计，写真值表，逻辑表达式，通过逻辑电路实现。

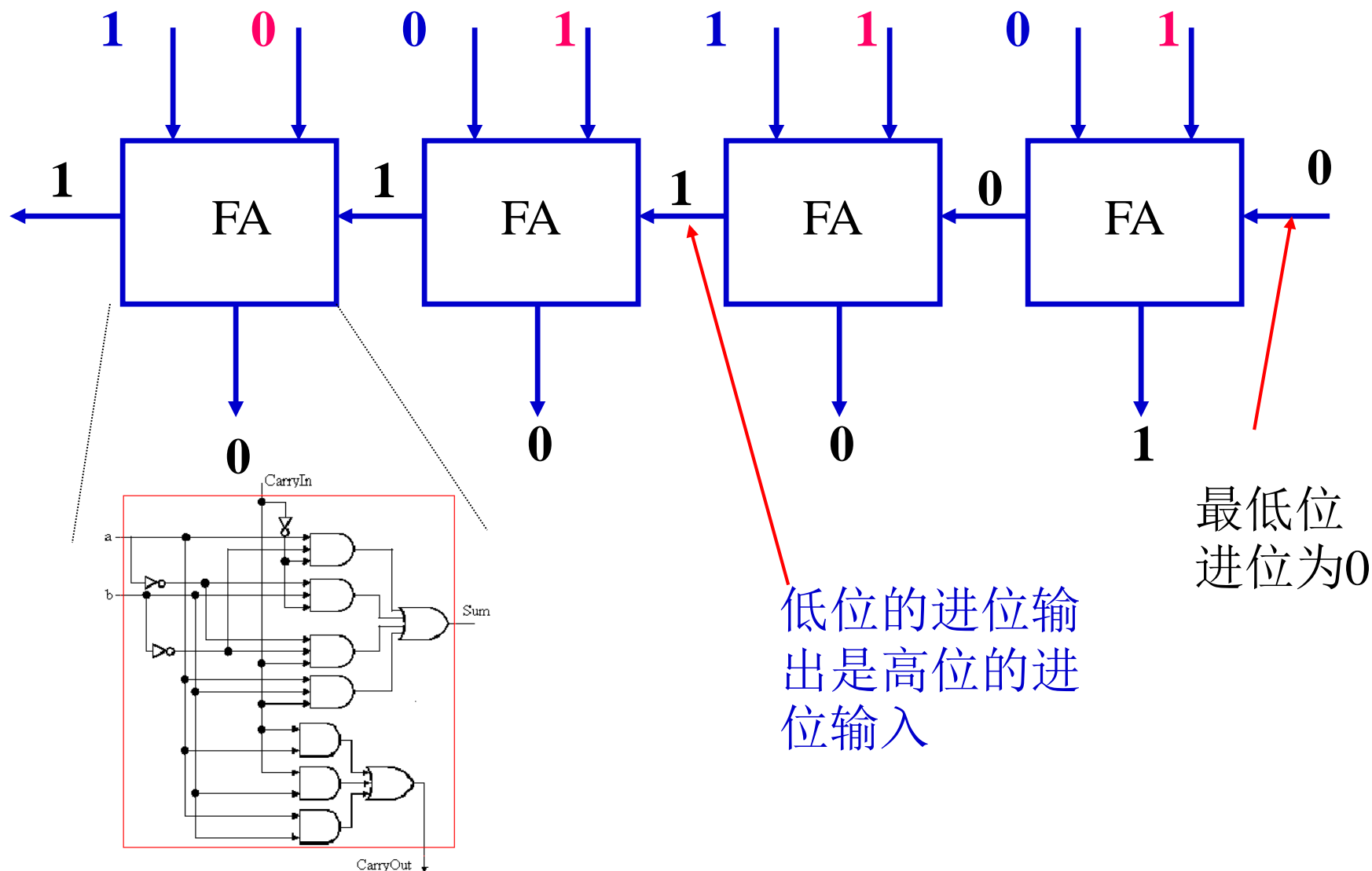
思路2:

- 用1位ALU串联起来，得到4位的ALU。

$$\begin{array}{rcccc} & 1 & 1 & 0 & 0 \\ & 1 & 0 & 1 & 0 \\ + & 0 & 1 & 1 & 1 \\ \hline 1 & 0 & 0 & 0 & 1 \end{array}$$



4位ALU设计





超前进位生成

如何能提前得到Cout?

显然:

1. 当 $a=b=0$ 时, $C_{out}=0$;
2. 当 $a=b=1$ 时, $C_{out}=1$;
3. 当 $a=1, b=0$ 或 $a=0, b=1$ 时, $C_{out}=C_{in}$.

由此可得:

$$C_1 = a_1b_1 + (a_1 + b_1)C_0;$$

$$C_2 = a_2b_2 + (a_2 + b_2)C_1;$$

$$C_3 = a_3b_3 + (a_3 + b_3)C_2;$$

$$C_4 = a_4b_4 + (a_4 + b_4)C_3.$$

定义:

$$P_i = a_i + b_i;$$

$$G_i = a_i b_i$$

通过单独的进位电路, 可以同时得到计算结果和进位

$$C_1 = G_1 + P_1 C_0$$

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

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

$$C_4 = \dots\dots$$



其它结果标志

$$Z=(F1 =0) \wedge (F2=0) \wedge (F3=0) \wedge (F4=0)$$

S = 最高位

$$OV=f1' * f2' * f_s + f1 * f2 * f_s'$$



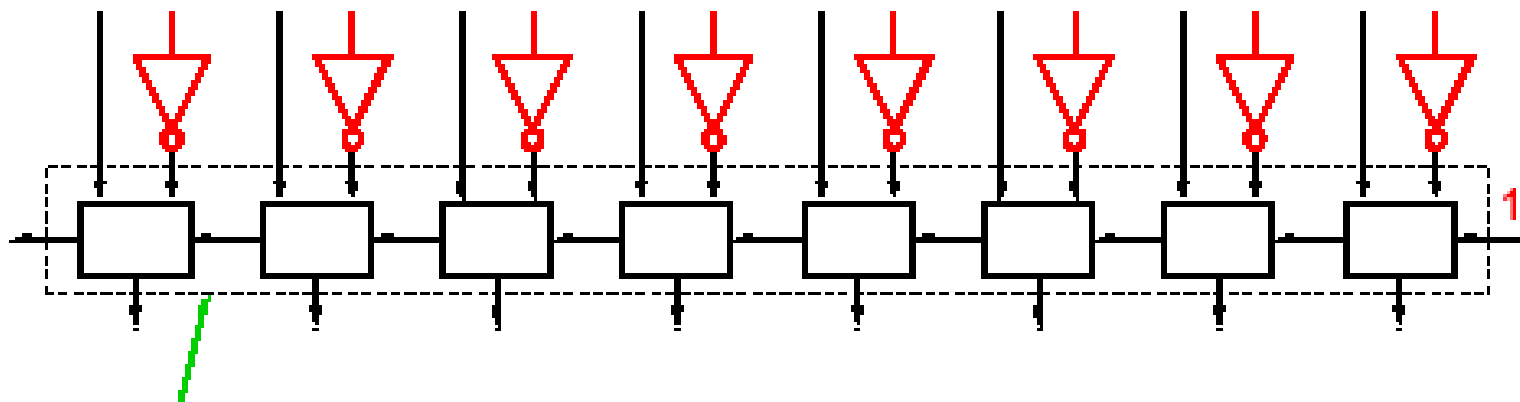
补码减法

根据算术运算规则：

$$[a-b]_{\text{补}} = [a]_{\text{补}} + [-b]_{\text{补}}$$

$[-b]$ 的补码为：将 $[b]_{\text{补}}$ 的各位求反，并加1。

由此，我们可以用加法器实现减法。



加法器

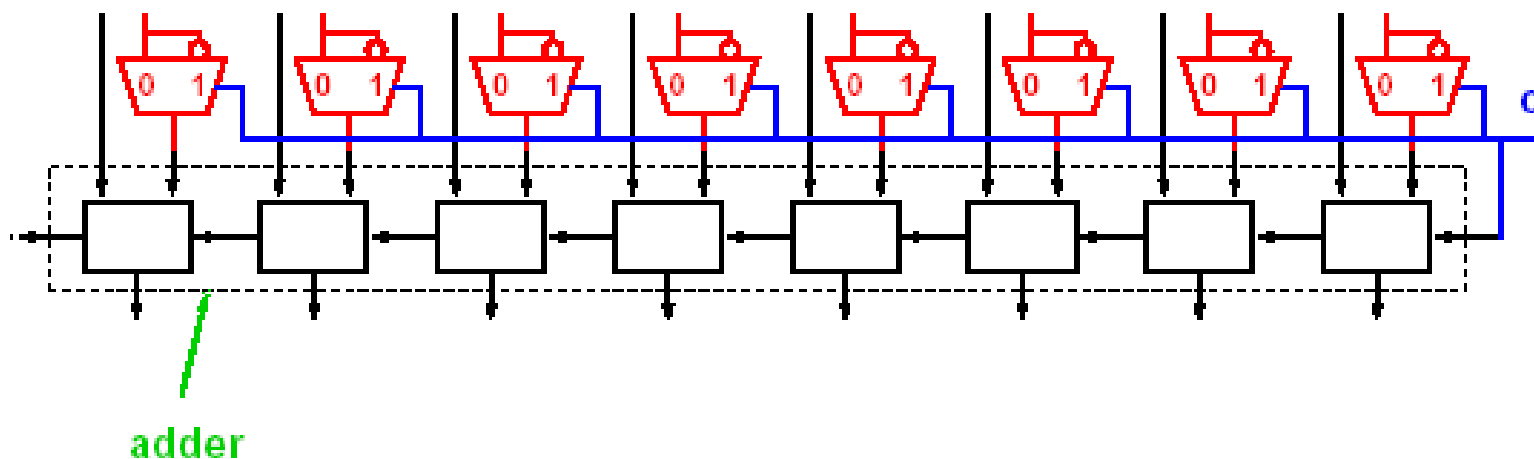
将加法和减法组合



给定控制命令 $C=0$ ，则ALU完成加法 $a+b$ ；

$C=1$ ，完成减法 $a-b$ 。

可以用选择器实现如下：



原码乘法



从一个简单的例子开始:

$$\begin{array}{r} 1000 \\ 1000 \\ 0000 \\ 0000 \\ 1000 \\ \hline 1001000 \end{array}$$

二进制乘法算法描述



✚ 基本算法：

- ✚ 若乘数的当前位 == 1， 将被乘数和部分积求和。
- ✚ 若乘数的当前位 == 0， 则跳过。
- ✚ 将部分积移位。

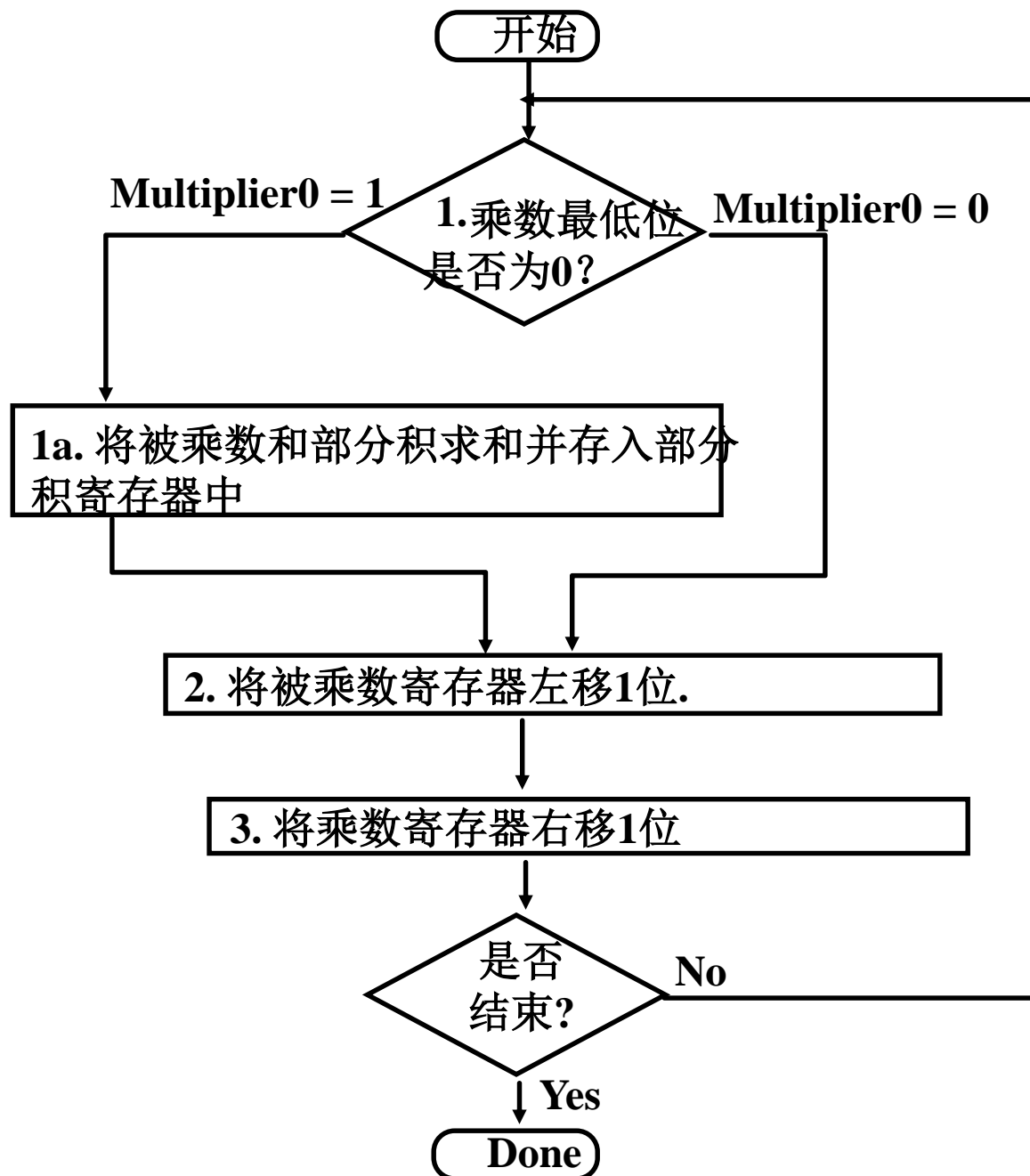
- ✚ 所有位都完成后， 部分积即为最终结果。

✚ N 位乘数 * M 位被乘数 $\Rightarrow N+M$ 位的积

✚ 乘法显然比加法更复杂...

- ✚ 但比10进制乘法要简单

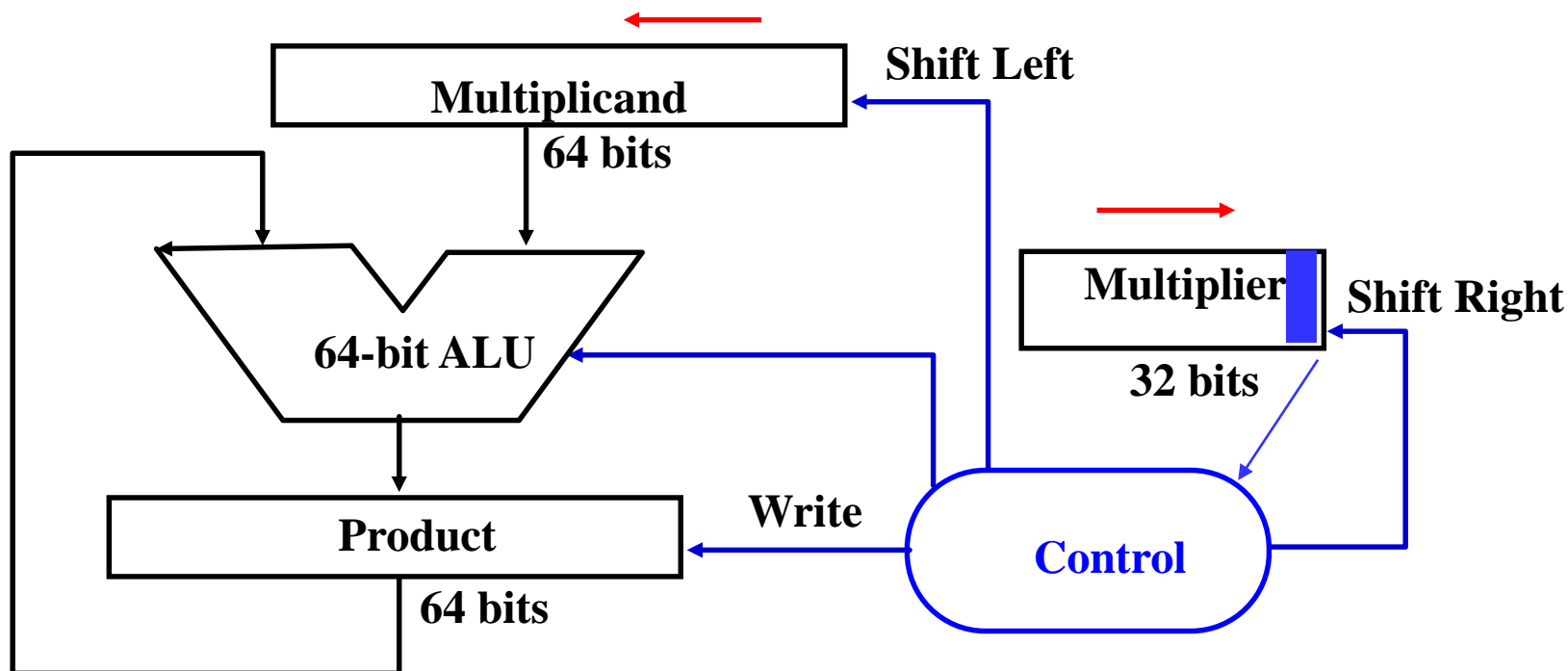
乘法算法 (一)





原码乘法的实现（一）

- 64-位被乘数寄存器, 64-位 ALU, 64-位部分积寄存器, 32-位乘数寄存器



Multiplier = datapath + control

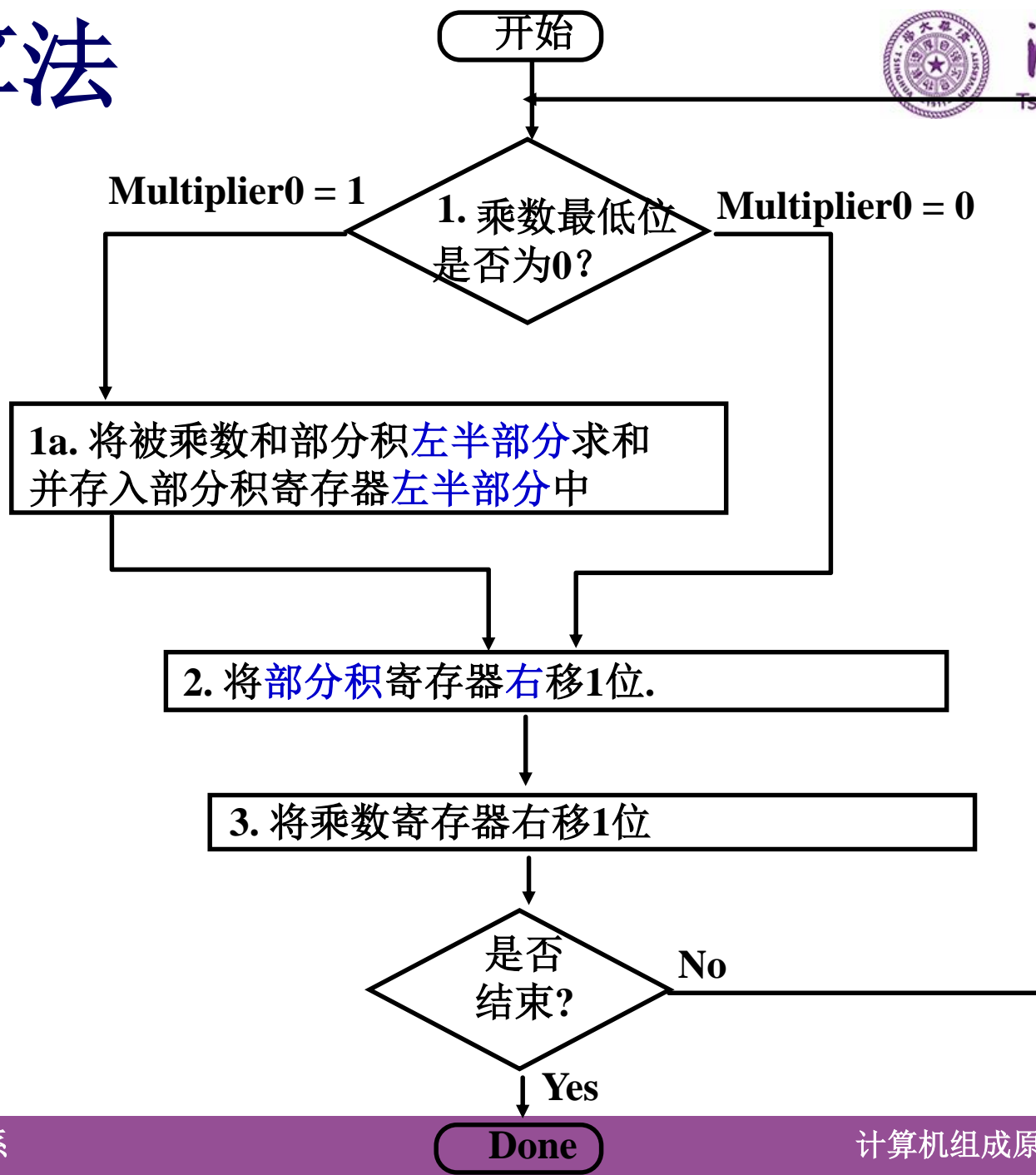
不足



❖ 实现（一）的不足：

- ❖ 被乘数的一半存储的只是0，浪费存储空间
- ❖ 每次加法实际上只有一半的位有效，浪费了计算能力

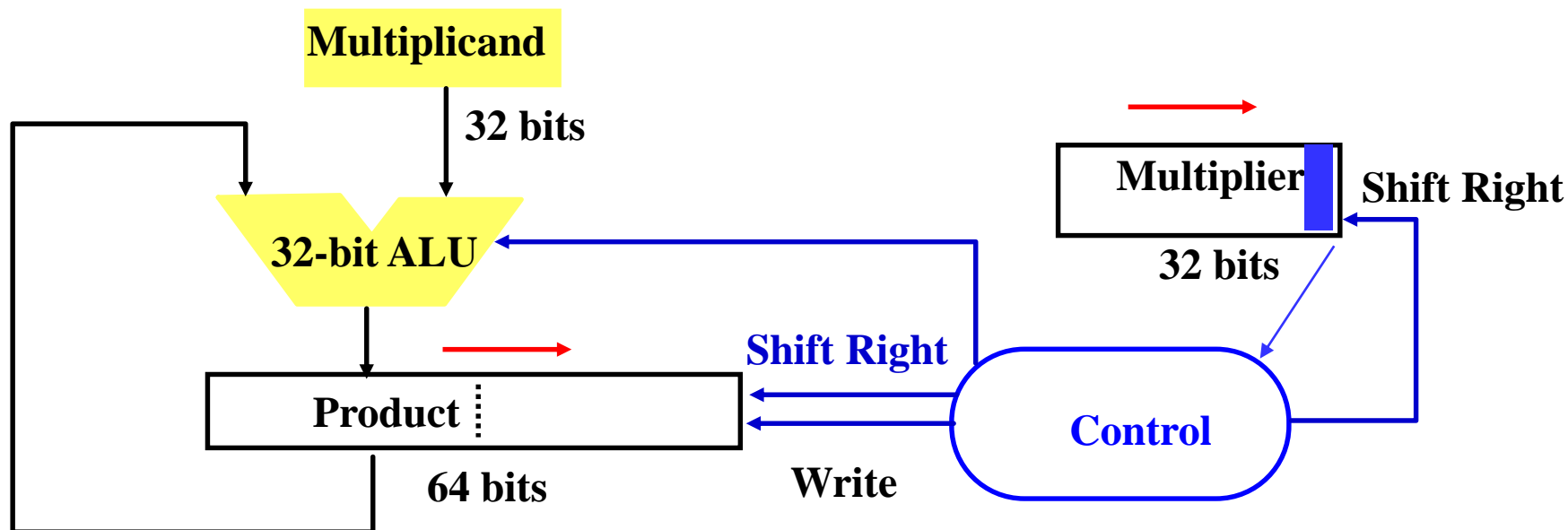
乘法算法 (二)



原码乘法的实现（二）



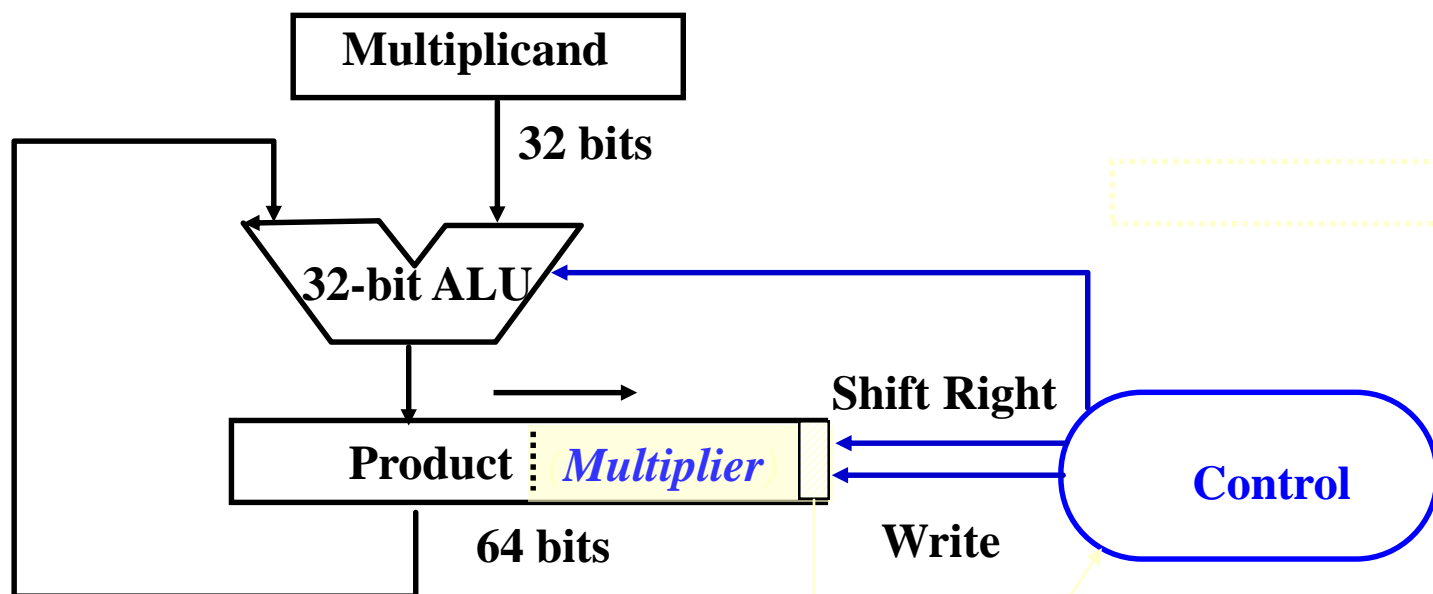
- 32-位被乘数寄存器，32-位 ALU，64-位部分积寄存器，32-位乘数寄存器



原码乘法的实现（三）



- 32-位被乘数寄存器，32-位ALU, 64-位部分积寄存器 (0-位乘数寄存器)





实现（三）的优点

- ❖ 实现（二）解决了对加法器位数的浪费
- ❖ 需要注意的是，乘数寄存器也存在浪费的情况：
 - ❑ 我们把已经完成的乘数位移出，移入的是0
 - ❑ 解决这个浪费，可以把乘数和部分积的低位结合起来。

补码乘法



❖ 方案一：

- ❑ 将补码转换为原码绝对值，进行原码的正数乘法
- ❑ 最后，根据以下原则得到符号位，并转换回补码表示：
 - ◆ 同号为正
 - ◆ 异号为负

❖ 方案二：补码直接乘

- ❑ 布斯算法

布斯算法



布斯算法原理：

- 虽然乘法是加法的重复，但也可以将它理解成加法和减法的组合。

例如：十进制乘法

- $6 \times 99 \dots$

- $6 \times 99 = 6 \times 100 - 6 \times 1 = 600 - 6 = 594$

布斯算法



二进制举例：

0111 x 0011 = ?

				0	1	1	1
				0	0	1	1
				<hr/>			
				0	1	1	1
			0	1	1	1	
		0	0	0	0		
	0	0	0	0			
	<hr/>						
0	0	1	0	1	0	1	

■ 我们也可以把它看成是下面计算过程：

■ $0 - 7*1 + 7*4 = 0 - 7 + 28 = 21$

补码乘法运算



$$[x]_{\text{补}} = \begin{cases} x & 2^n > x \geq 0 \\ 2^{n+1} + x & 0 \geq x \geq -2^n \pmod{2^{n+1}} \end{cases}$$

若 $[x]_{\text{补}} = x_{n-1}x_{n-2} \cdots x_1x_0$ ，则：

$$x = -2^{n-1}x_{n-1} + \sum_{i=0}^{n-2} x_i 2^i$$

同理： $[y]_{\text{补}} = y_{n-1}y_{n-2} \cdots y_1y_0$ ，则：

$$y = -2^{n-1}y_{n-1} + \sum_{i=0}^{n-2} y_i 2^i$$

补码乘法运算



$$\oplus [x*y]_{\text{补}} = [x]_{\text{补}} * (-2^{n-1} y_{n-1} + \sum_{i=0}^{n-2} y_i 2^i)$$

$$\oplus = [x]_{\text{补}} * [2^{n-1}(y_{n-2} - y_{n-1}) + 2^{n-2}(y_{n-3} - y_{n-2}) + \cdots + 2^0(y_{-1} - y_0)]$$

$$\oplus = [x]_{\text{补}} * \sum_{i=0}^{n-1} 2^i (y_{i-1} - y_i) \quad \text{其中: } y_{-1}=0$$

可直接用补码进行乘法运算

根据乘数相邻两位的不同组合，确定是 $+ [x]_{\text{补}}$ 或 $- [x]_{\text{补}}$

补码乘法运算



用 **Y 的值** 乘 $[X]_{\text{补}}$ ，达到 $[X]_{\text{补}}$ 乘 $[Y]_{\text{补}}$ ，
求出 $[X * Y]_{\text{补}}$ ，不必区分符号与数值位。
乘数最低一位之后要补初值为 0 的一位附加线
路，并且每次乘运算需要看附加位和最低位两
位取值的不同情况决定如何计算部分积，其规
则是：

0 0	+ 0
0 1	+ 被乘数
1 0	- 被乘数
1 1	+ 0

举例：2× (-5)



步骤	操作	部分积	附加位
0	初始值	0000011011	0
1	-X	1111011011	0
	右移	1111101101	1
2	右移	1111110110	1
3	+X	0000110110	1
	右移	0000011011	0
4	-X	1111011011	0
	右移	1111101101	1
5	右移	1111110110	1

$$(1111110110)_2 = -10$$

乘法运算：小结



- ✚ 与加法比较，需要使用更多的硬件来实现，也更复杂
- ✚ 若使用简单的方法来实现，则需要多个计算周期
- ✚ 仅仅介绍了乘法运算的一些“皮毛”：有许多提升和优化的空间

除法运算



- ❖ 在计算机内实现除运算时，存在与乘法运算类似的几个问题：加法器与寄存器的配合，被除数位数更长，商要一位一位地计算出来等。这可以用左移余数得到解决，且被除数的低位部分可以与最终的商合用同一个寄存器，余数与上商同时左移。
- ❖ 除法可以用原码或补码计算，都比较方便，也有一次求多位商的快速除法方案，还可以用快速乘法器完成快速除法运算。

原码一位除运算



$$[Y/X]_{\text{原}} = (X_S \oplus Y_S) (|Y| / |X|)$$

原码一位除是指用原码表示的数相除，求出原码表示的商。除操作的过程中，每次求出一位商。

恢复余数法：被除数-除数，若结果 ≥ 0 ，则上商1，移位；若结果 < 0 ，则商0，恢复余数后，再移位；

求下一位商；

但计算机内从来不用这种方法，而是直接用求得的负余数求下一位商。

原码一位除运算



$$[X / Y]_{\text{原}} = (X \oplus Y) (|X| / |Y|)$$

例如： $X = 0.1011$ $Y = -0.1101$

$$\begin{array}{r}
 0.1101 \\
 0.1101 \overline{) 0.10110} \\
 \underline{01101} \\
 10010 \\
 \underline{01101} \\
 10100 \\
 \underline{1101} \\
 0111
 \end{array}$$

被除数 (余数)	商	
00 1011	00000	初态
01 0110	000 0	第 1 次
01 0010	00 01	第 2 次
00 1010	0 011	第 3 次
01 0100	0110	第 4 次
00 0111	01101	第 5 次

X 和 Y 符号异或为负
最终商原码表示为：

1 1101

余数为： $0.0111 * 2^{-4}$

加减交替除法原理证明



1. 若第 $i-1$ 次求商，减运算的余数为 $+R_{i-1}$ ，商1，
余数左移1位得 $2R_{i-1}$ 。
2. 则下一步第 i 次求商 $R_i = 2R_{i-1} - Y$ 若 $R_i \geq 0$ ， ...
恢复余数为正且左移1位得 $2(R_i + Y)$ 若 $R_i < 0$ ，商 0
3. 则再下一步第 $i+1$ 次求商 $R_{i+1} = 2(R_i + Y) - Y$
 $= 2R_i + Y$

公式表明，若上次减运算结果为负，可直接左移，
本次用 $+Y$ 求余即可；减运算结果为正，用 $-Y$ 求余



被除数(余数)

商

```

      0 0 1 0 1 1
    +) 1 1 0 0 1 1
    -----
      1 1 1 1 1 0
      1 1 1 1 0 0
    +) 0 0 1 1 0 1
    -----
      0 0 1 0 0 1
      0 1 0 0 1 0
    +) 1 1 0 0 1 1
    -----
      0 0 0 1 0 1
      0 0 1 0 1 0
    +) 1 1 0 0 1 1
    -----
      1 1 1 1 0 1
      1 1 1 0 1 0
    +) 0 0 1 1 0 1
    -----
      0 0 0 1 1 1
  
```

```

      0 0 0 0 0
      0 0 0 0 0
      0 0 0 0 0
      0 0 0 0 1
      0 0 0 1 0
      0 0 1 1 1
      0 1 1 1 0
      0 1 1 0 1
      0 1 1 0 1
  
```

开始情形

-Y

<0, 商0

左移1位

+Y

>0, 商1

左移1位

-Y

>0, 商1

左移1位

-Y

<0, 商0

左移1位

+Y

>0, 商1

补码除法运算



补码除法与原码除法很类似

差别仅在于：

被除数与除数为补码表示，
直接用补码除，求出反码商，
再修正为近似的补码商。

实现中，求第一位商要判 2 数符号的同异，

同号，作减法运算，异号，则作加运算；

上商，余数与除数同号，商 1，作减求下位商，

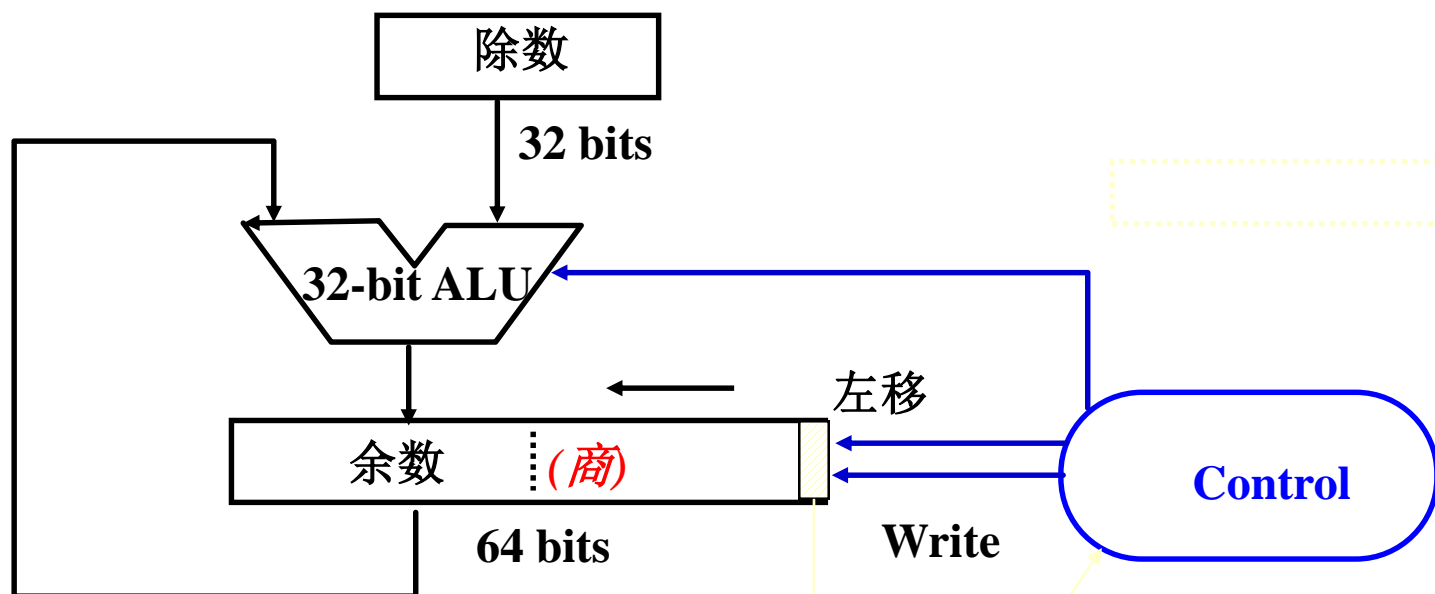
余数与除数异号，商 0，作加求下位商；

商的修正：多求一位后舍入，或最低位恒置 1

除法的实现



- 32-位除数寄存器, 32 -位ALU, 64-位
余数 (被除数) 寄存器



小结



- ✚ ALU的基本功能： 算术、 逻辑运算
- ✚ 1位ALU： 最基本的功能： 加法、 与、 或
- ✚ 位数扩展： 快速进位
- ✚ 功能扩展： 减法、 乘法、 除法

阅读与思考



阅读

- 教材第3章3.4、3.5、3.8节

思考

- ALU的最大延迟如何估算?
- 运算器其他功能如何实现?

书面作业和实验一

- 从网络学堂下载，其中，编程实现的作业在网络学堂提交，应包括源代码电子版和可执行文件以及你们自己的测试结果。其余作业提交手写版。
- 完成Project1。具体要求见《计算机硬件系统实验教程》第5.1节。