

Computer Organization and Design

中山 大 学
计算机学院

郭雪梅

Email: guoxuem@mail.sysu.edu.cn

* 设计过程与ALU设计



Topics

*Brief overview of:

- integer multiplication 整数乘法
- integer division 整数除法
- floating-point numbers and operations 浮点数操作

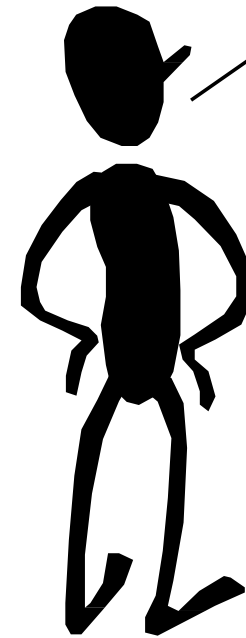
Reading: Study Chapter 3.1-3.4

Binary Multipliers

乘法的关键技巧是记住一个数字到数字的表格..... 其他一切都只是求和

×	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

×	0	1
0	0	0
1	0	1



二进制使乘法简单:

- 0 => place 0
- 1 => place a copy

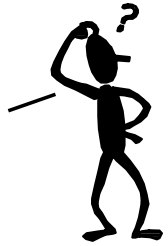
Key sub-parts:

- 放拷贝或不放
- 移位拷贝
- 最后求和

“Shift and Add” Multiplier

The “Binary”
Multiplication
Table

Hey, that
looks like
an AND
gate



X	0	1
0	0	0
1	0	1

二进制乘法是使用在小学学习的基本手算算法实现的。

$A_j B_i$ is a “partial product”

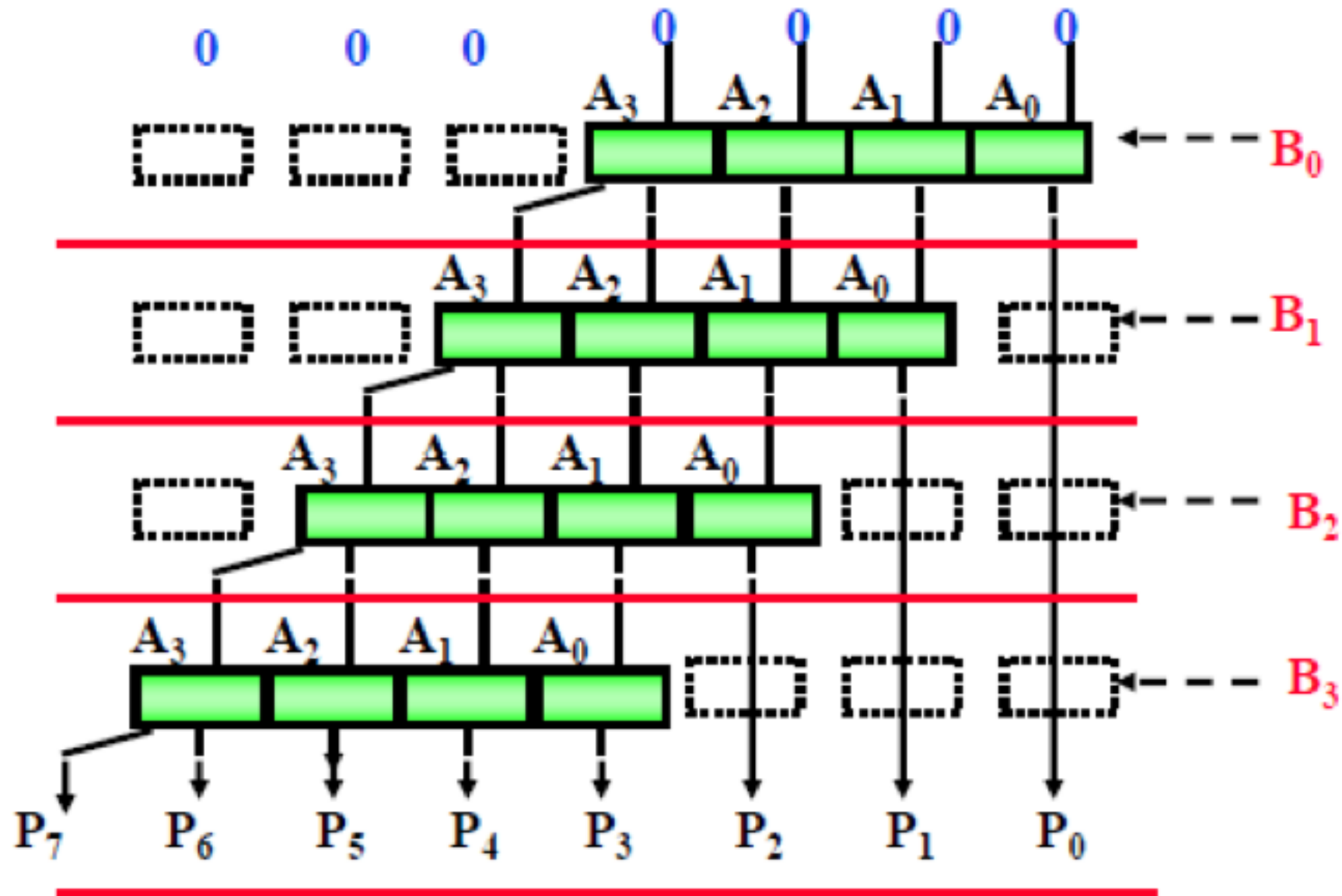
$$\begin{array}{r}
 \begin{array}{cccc}
 & A_3 & A_2 & A_1 & A_0 \\
 \times & B_3 & B_2 & B_1 & B_0 \\
 \hline
 & & & & A_3 B_0 & A_2 B_0 & A_1 B_0 & A_0 B_0 \\
 & & & & A_3 B_1 & A_2 B_1 & A_1 B_1 & A_0 B_1 \\
 & & & & A_3 B_2 & A_2 B_2 & A_1 B_2 & A_0 B_2 \\
 + & A_3 B_3 & A_2 B_3 & A_1 B_3 & A_0 B_3 & & &
 \end{array} \\
 \hline
 \end{array}$$

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： $0 \Rightarrow 0$ (0 x 被乘数)

$1 \Rightarrow$ 被乘数 (1 x 被乘数)

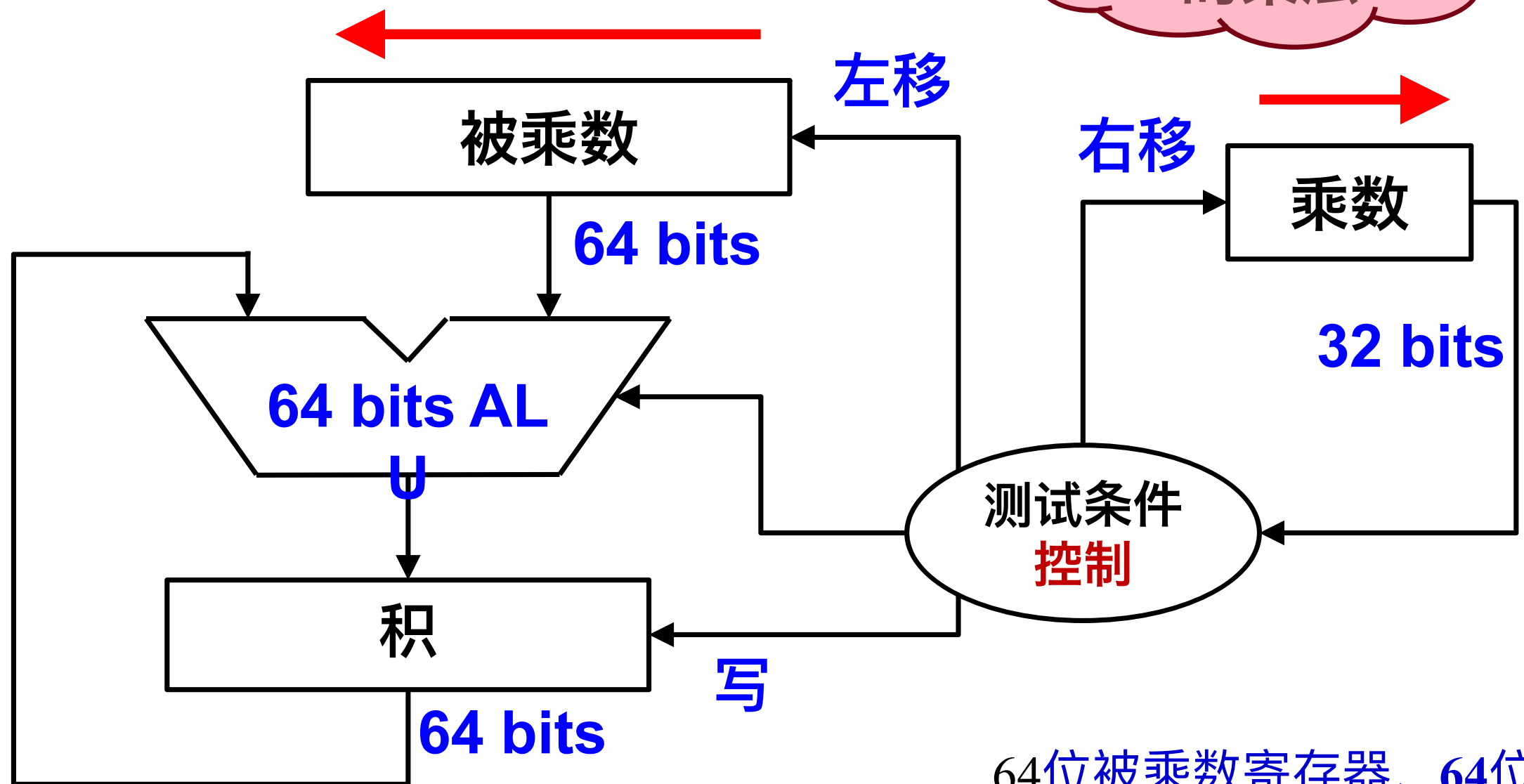
上述乘法器如何工作？



- °每级都对A进行左移 ($\times 2$)
- °使用B的下一位来判断是否加上左移后的被乘数 ($B_i=1$ 则加, 否则不加)
- °每级都累计 $2n$ 位的部分积

第一种无符号移位-加法乘法器V1。

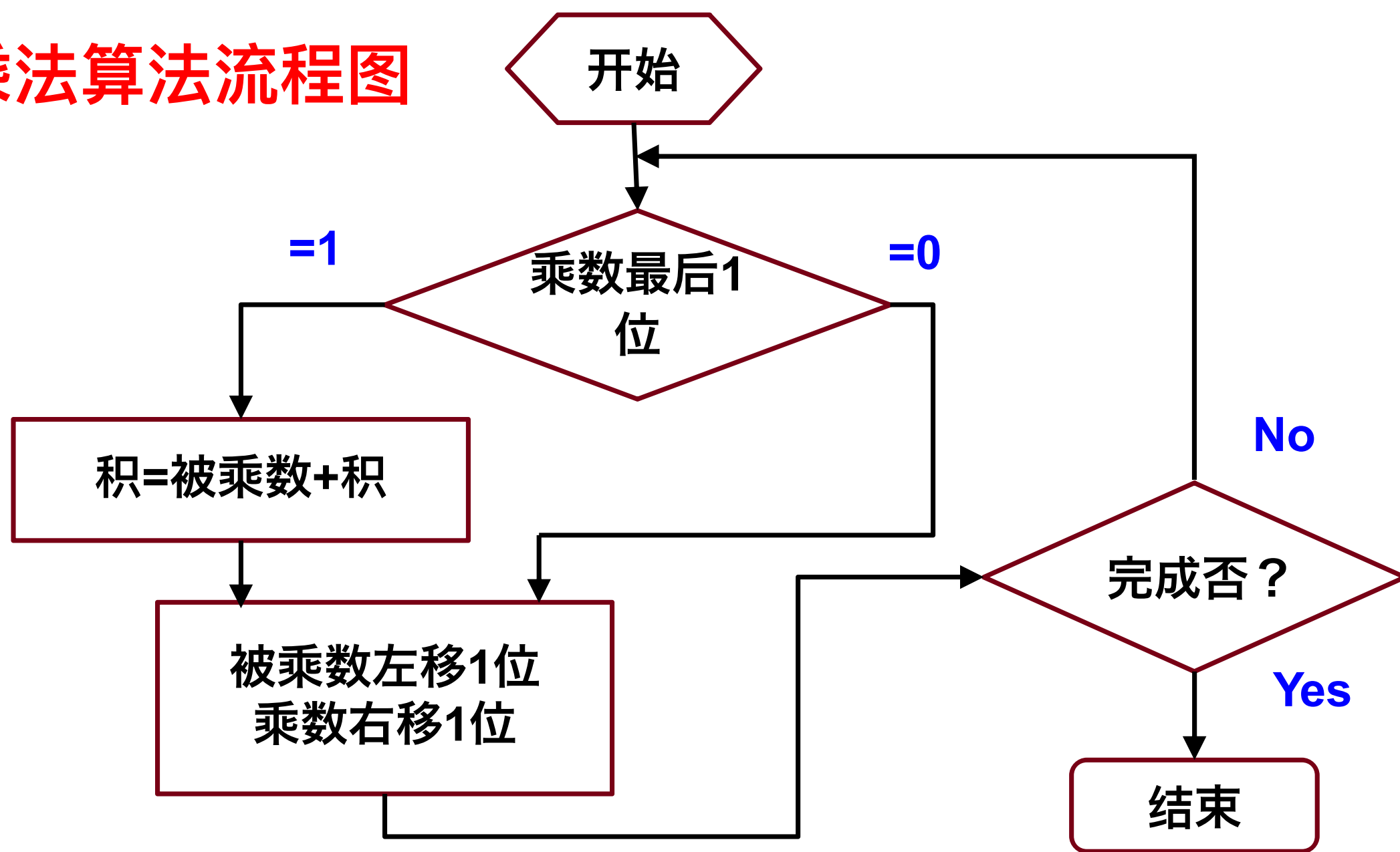
先了解正数的乘法



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

第一种乘法算法流程图



✱ 需要32次迭代 (加法移位比较)

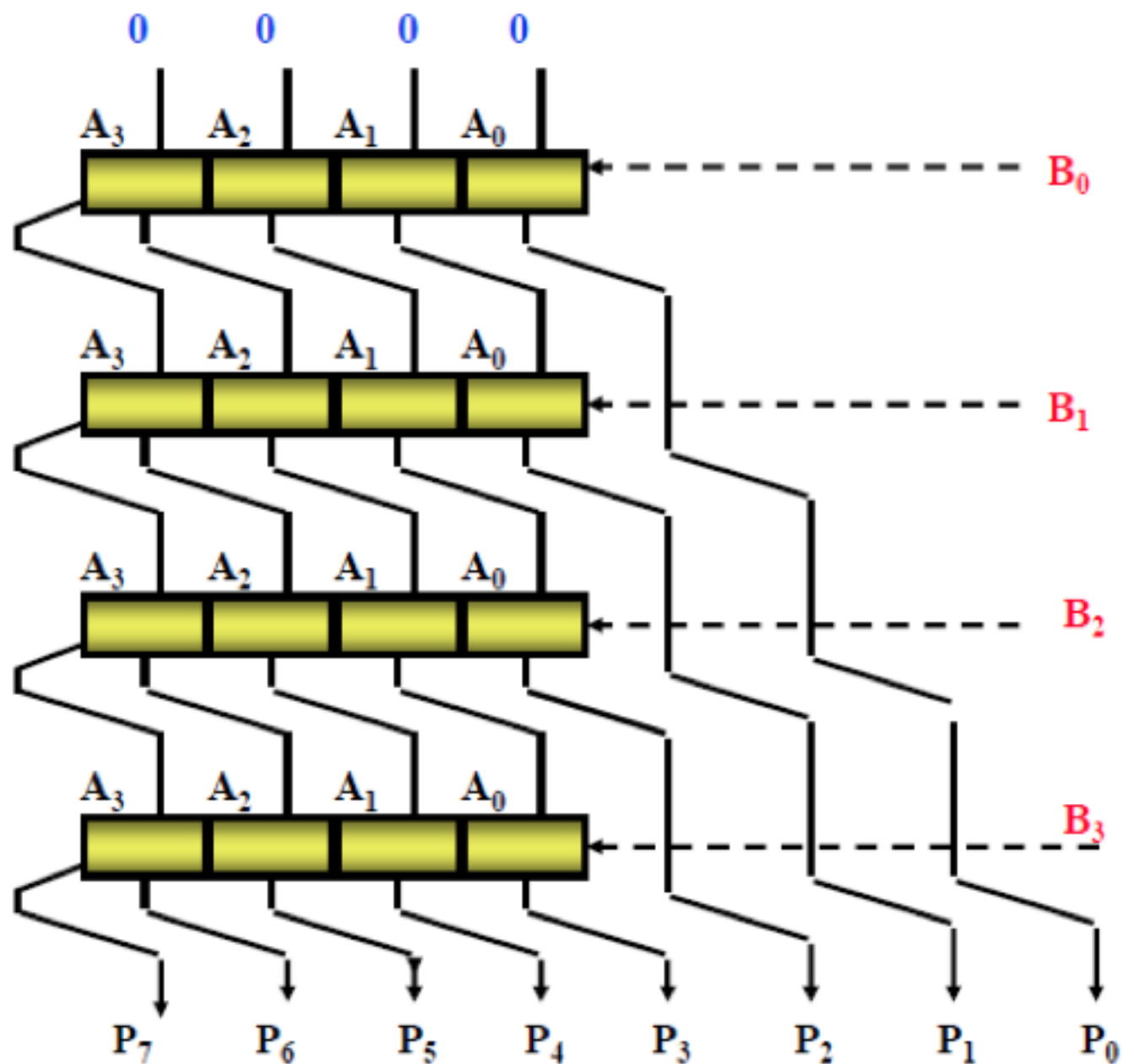
✱ 几乎用 100 时钟周期

✱ 非常大, 太慢!

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低位, 它就永远不变 !!!
- * 用乘积右移 替代 被乘数左移?

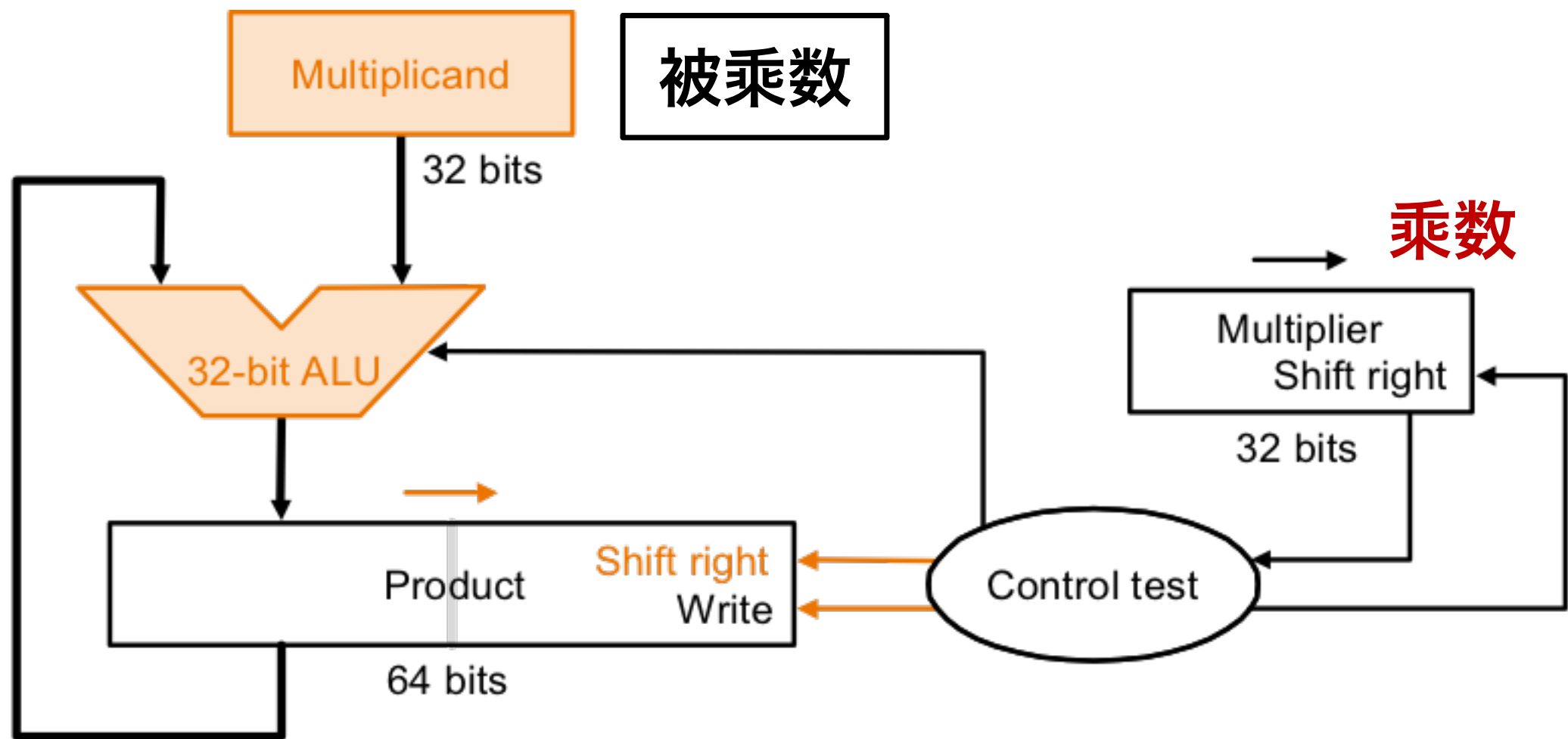
如何进一步改进？



被乘数固定不动，乘积右移 **ALU reduced to 32 bits!**

Multiplier V2-- Logic Diagram

- * 第二版本 V2 乘法器
- * 只有乘积寄存器**左边一半**有变化



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

Example for second version

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial	1011	0010	0000 0000
1	Test true shift right	1011 0101	0010	0010 0000 0001 0000
2	Test true shift right	0101 0010	0010	0011 0000 0001 1000
3	Test false shift right	0010 0001	0010	0001 1000 0000 1100
4	Test true shift right	0001 0000	0010	0010 1100 0001 0110

Multiplier V 3 (Final Version)

- ✱ 进一步优化
- ✱ 乘积寄存器的初始状态为 '0'
- ✱ 乘积寄存器的低 32 bits 只是简单地被移出去
- ✱ 改进:
把这个低32位用作乘数寄存器

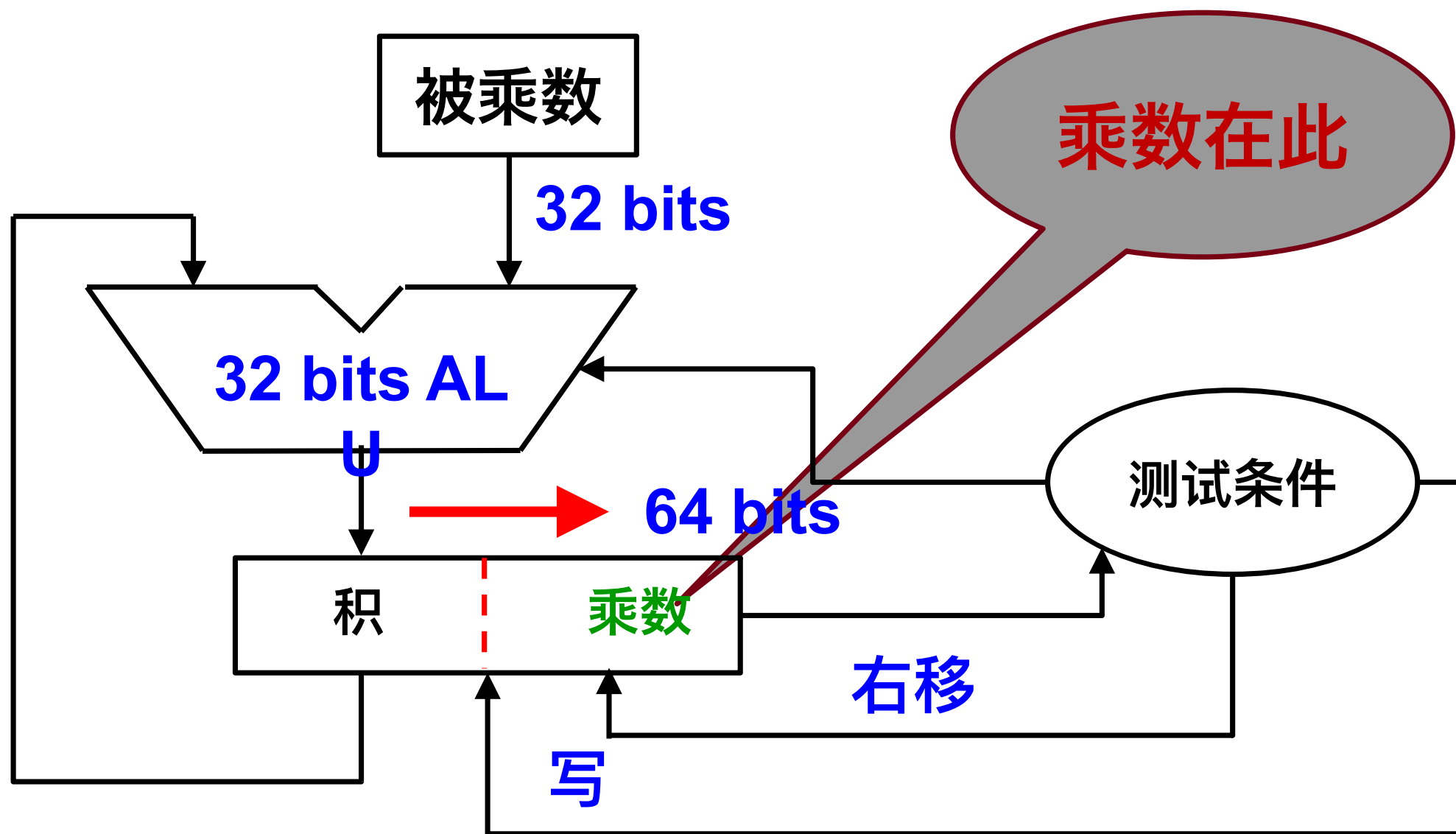
0	0	0	1	0	0	0	0		
0	0	0	1	1	0	0	0	0	
0	0	0	1	1	1	0	0	0	0
0	0	0	0	1	1	1	0	0	0
0	0	0	0	0	1	1	1	0	0

multiplier

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

改进的乘法实现硬件

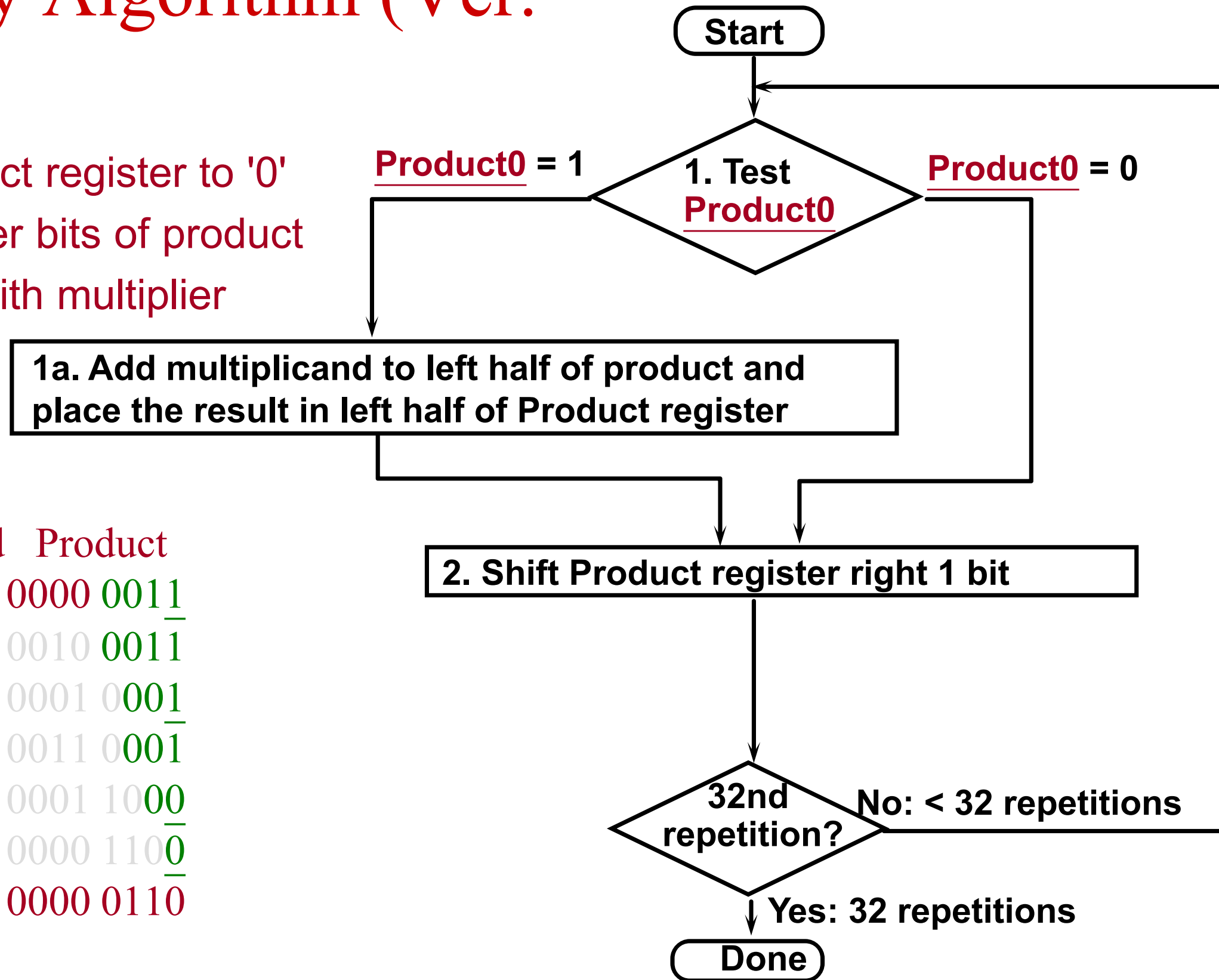
第三种乘法硬件



测试乘数最右边的位，当乘数为1时，将乘数和被乘数移位，同时将被乘数与积相加

Multiply Algorithm (Ver. 3)

- * Set product register to '0'
- * Load lower bits of product register with multiplier



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

Example with V3

- Multiplicand x multiplier: 0001 x 0111

Multiplicand:	0001		
Multiplier:×	0111		
	00000111		
1	00010111		
	00001011	1	#积(product) 初始化值 #加被乘数0001, 乘数=1 #乘积右移一位后
	0001		
2	00011011		
	00001101	1	#加被乘数0001, 乘数=1 #乘积右移一位后
	0001		
3	00011101		
	00001110	1	#加被乘数0001, 乘数=1 #乘积右移一位后
	0000		
4	00001110		
	00000111	0	#加被乘数0000, 乘数=0 #乘积右移一位后

Signed Multiplication

有符号数乘法如何?

✱简单地策略是 假设两个源操作数都是正数, 在运算结束时再对乘积进行修正 (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

✱比较合理的版本:

- 使用无符号乘法硬件
- 右移时, 扩展乘积的符号位
- 如果乘数是负数, 最后一步应该是减法

Signed Multiplication (Pencil & Paper)

* Case 1: Positive Multiplier

$$\begin{array}{r} \text{Multiplicand} \quad 1100_2 = -4 \\ \text{Multiplier} \quad \times \quad 0101_2 = +5 \\ \hline \end{array}$$

$$\begin{array}{r} \text{Sign-extension} \left\{ \begin{array}{l} \rightarrow 11111100 \\ \rightarrow 111100 \end{array} \right. \\ \hline \end{array}$$

$$\text{Product} \quad 11101100_2 = -20$$

如果乘数是负数,
最后一步应该是减法

* Case 2: Negative Multiplier

$$\begin{array}{r} \text{Multiplicand} \quad 1100_2 = -4 \\ \text{Multiplier} \quad \times \quad 1101_2 = -3 \\ \hline \end{array}$$

$$\begin{array}{r} \text{Sign-extension} \left\{ \begin{array}{l} \rightarrow 11111100 \\ \rightarrow 111100 \end{array} \right. \\ \hline \end{array}$$

$$0100 \quad (2\text{'s complement of } 1100)$$

$$\text{Product} \quad 00001100_2 = +12$$

$$(1100)_{\text{补码}}$$

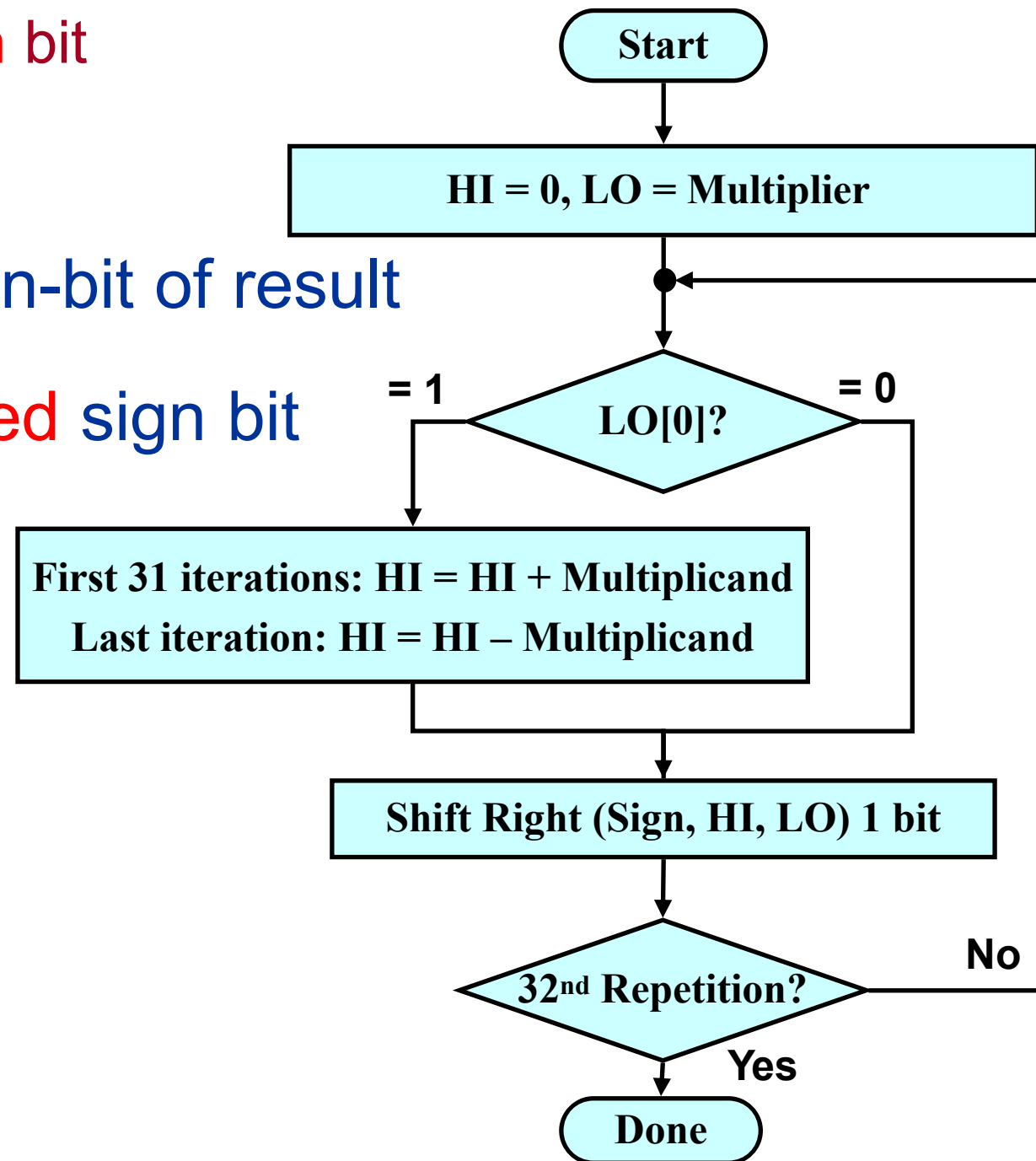
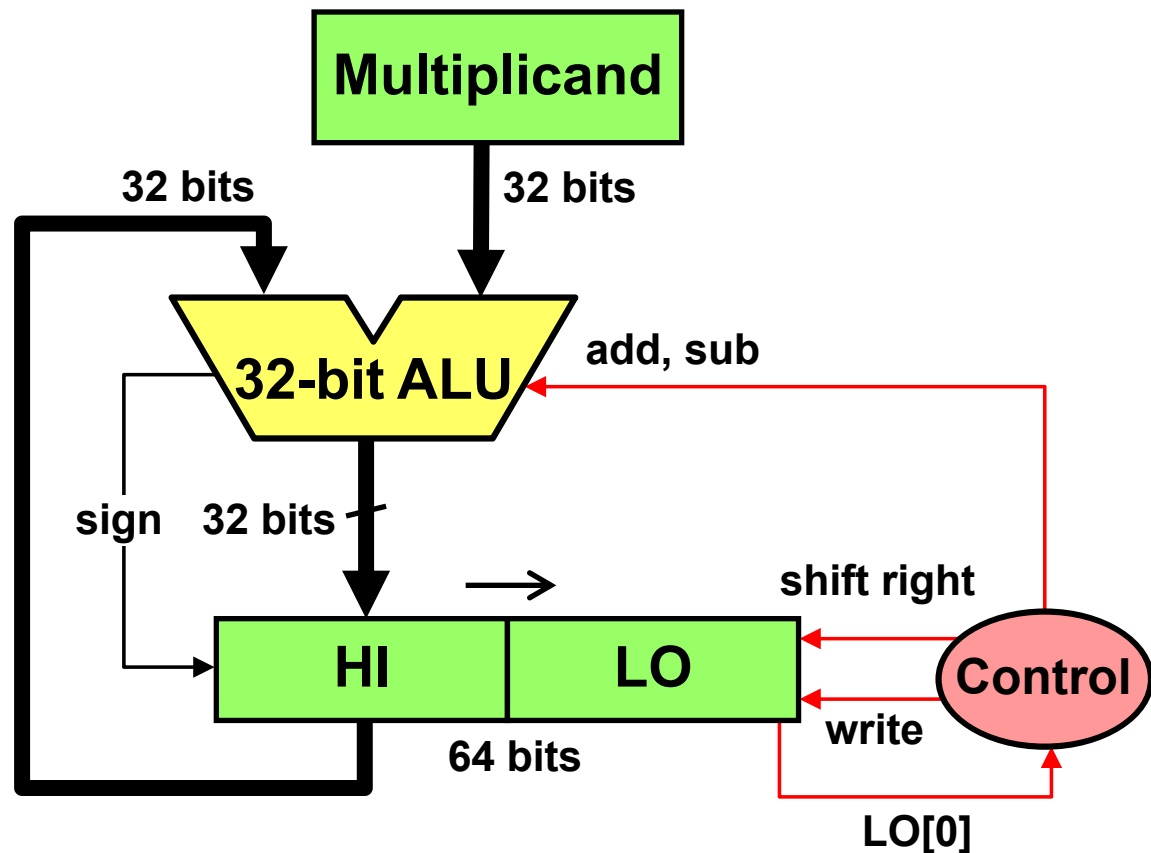
$$= 0100$$

减1100等于加0100

Sequential Signed Multiplier

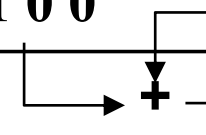
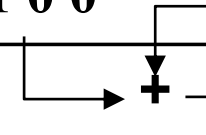
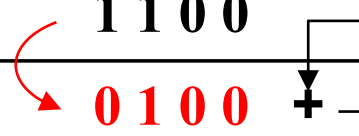
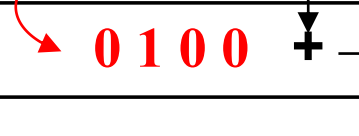
顺序有符号乘法器

- * ALU produces 32-bit result + Sign bit
- * Check for overflow
 - No overflow → Extend sign-bit of result
 - Overflow → Invert extended sign bit



Signed Multiplication Example

- * Consider: 1100_2 (-4) \times 1101_2 (-3), Product = 00001101_2
- * Check for overflow: No overflow \rightarrow Extend sign bit
- * Last iteration: add 2's complement of Multiplicand

Iteration		Multiplicand	Sign	Product = HI, LO
0	Initialize (HI = 0, LO = Multiplier)	1 1 0 0		0 0 0 0 1 1 0 1
1	LO[0] = 1 \Rightarrow ADD	 1 1 0 0	1	1 1 0 0 1 1 0 1
	Shift (Sign, HI, LO) right 1 bit	1 1 0 0		1 1 1 0 0 1 1 0
2	LO[0] = 0 \Rightarrow Do Nothing			
	Shift (Sign, HI, LO) right 1 bit	1 1 0 0		1 1 1 1 0 0 1 1
3	LO[0] = 1 \Rightarrow ADD	 1 1 0 0	1	1 0 1 1 0 0 1 1
	Shift (Sign, HI, LO) right 1 bit	 1 1 0 0 0 1 0 0		1 1 0 1 1 0 0 1
4	LO[0] = 1 \Rightarrow SUB (ADD 2's compl)	 0 1 0 0	0	0 0 0 1 1 0 0 1
	Shift (Sign, HI, LO) right 1 bit			0 0 0 0 1 1 0 0

Booth's Algorithm 改进

Booth's Algorithm Key Idea

看一下1的串：

$$2 \times 30 = 00010_2 \times 011110_2$$

$$30 = -2 + 32$$

$$011110 = -000010 + 100000$$

乘：

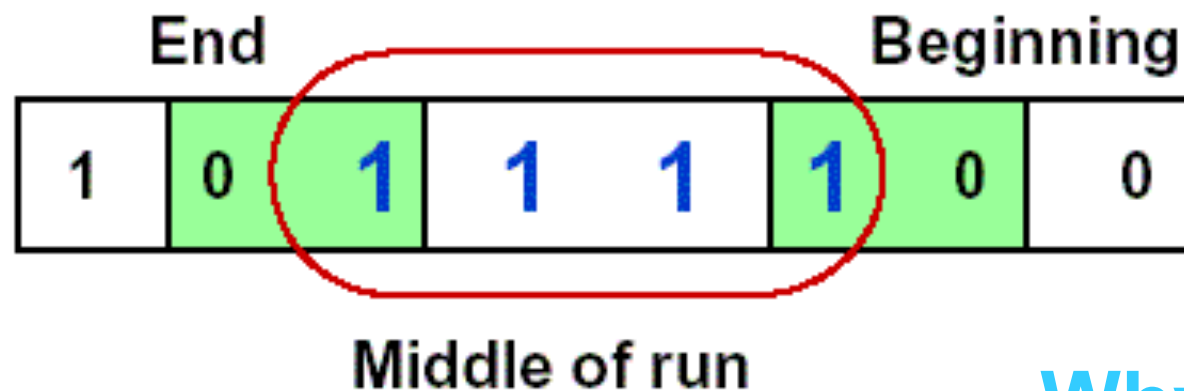
- 加 000010 **4次** (w/ shifts) (011110_2)
- 或 -
- 加 100000 **一次 减 000010 一次** (w/ shifts) ($-000010 + 100000$)

When is this faster?

Booth's Algorithm 改进

✱ Idea: 如果乘数有连续个 '1'

- 乘数第一个1做减法
- 对'1'移位
- 最后一个1做加法



乘数中间有'1'的串 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。

✱ Result:

- 可能更多移位, 少了加法
- 更快, 移位操作比加法速度快

Why it works? -1

$$\begin{array}{r} + 10000 \\ \hline 01111 \end{array}$$

早期, 由于移位比加法速度更快, 采用该算法主要为了速度

Booth's Algorithm rule

* Analysis of two consecutive bits

Current	last	Explanation	Example
1	0	Beginning	0000111 1 0000
1	1	middle of '1'	00001 11 10000
0	1	End	000 01 1110000
0	0	Middle of '0'	00 00 11110000

* Action

1 0	左半部分减被乘数
1 1	不做运算
0 1	左半部分加被乘数
0 0	不做运算

* Bit₋₁ = '0'

* 算术右移:

- 保持最高位不变
- 保持符号位不变!

乘数为负举例

* 2 * -3 = - 6

* 0010 * 1101 = 1111 1010

- 1 0 左半部分减被乘数
- 1 1 不做运算
- 0 1 左半部分加被乘数
- 0 0 不做运算

iteration	step	Multiplicand	product
0	Initial Values	0010	0000 1101 0
1	1.c:10→Prod=Prod-Mcand	0010	1110 1101 0
	2: shift right Product	0010	1111 0110 1
2	1.b:01→Prod=Prod+Mcand	0010	0001 0110 1
	2: shift right Product	0010	0000 1011 0
3	1.c:10→Prod=Prod-Mcand	0010	1110 1011 0
	2: shift right Product	0010	1111 0101 1
4	1.d: 11 → no operation	0010	1111 0101 1
	2: shift right Product	0010	1111 1010 1

并行乘法器

早期的计算机中采用串行的1位乘法方案，即多次执行“*加-移位*”操作来实现。这种方法硬件简单，但速度太低，不能满足科学技术对高速乘法所提出的要求。自从大规模集成电路问世以来，高速的单元阵列乘法器应运而生，出现了各种形式的流水式阵列乘法器，它们属于*并行乘法器*。

设两个不带符号的二进制整数

$$A = a_{m-1} \dots a_1 a_0$$

$$B = b_{n-1} \dots b_1 b_0$$

$$\text{则二进制乘积 } P = ab = (\sum a_i 2^i)(\sum b_j 2^j) = \sum \sum (a_i b_j) 2^{i+j}$$

Array Multiplier 阵列乘法器

❖ 基本思路

若将所有的部分积 $a_i b_j$ 都同时算出来，剩下的就是带进位位的加法求和。这种乘法器要实现 n 位* n 位时，需要 $n(n-1)$ 个全加器和 n 个“与”门。

▶ 以 4 位无符号数为例

■ 手工计算： $15 \times 11 = 165$

$$\begin{array}{r} \text{被乘数} \quad 1111 \\ \times \text{乘数} \quad 1011 \\ \hline 1111 \\ 11110 \\ 0000 \\ + 1111 \\ \hline 10100101 \\ \text{积} \end{array}$$

$$\text{二进制乘积 } P = ab = (\sum a_i 2^i)(\sum b_j 2^j) = \sum \sum (a_i b_j) 2^{i+j}$$

$$\text{其中 } C_{ij} = A_i B_j$$

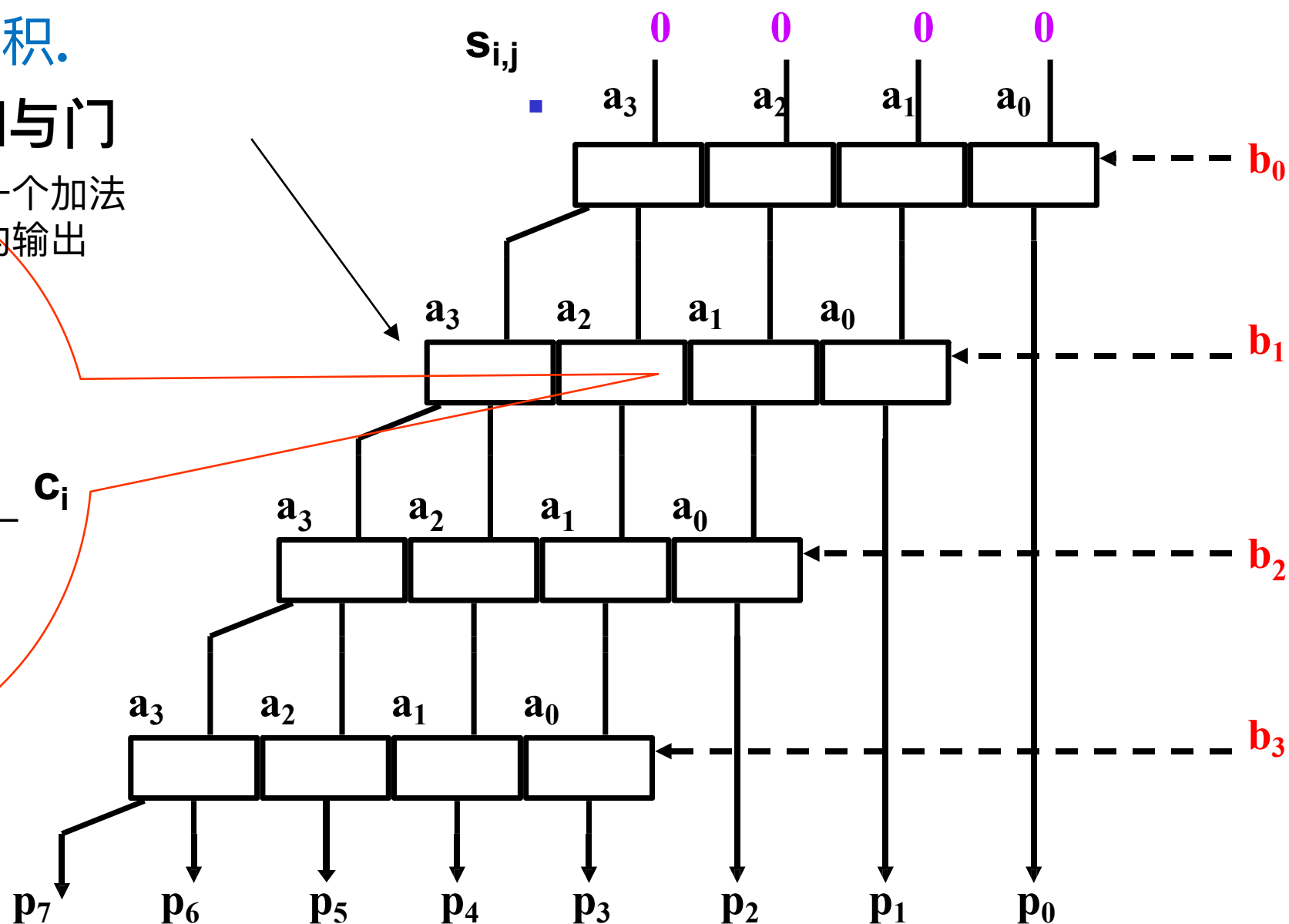
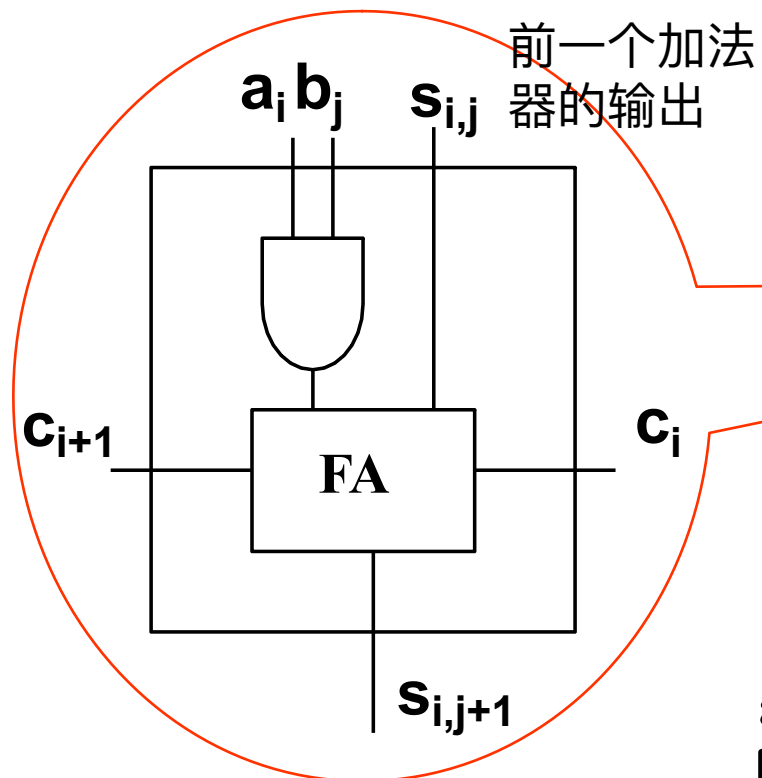
阵列乘法器

ALU电路设计

■ 阵列乘法器(并行乘法器)——方案

同时生成所有 n 个部分积。

每一行: n -bit 加法器和与门

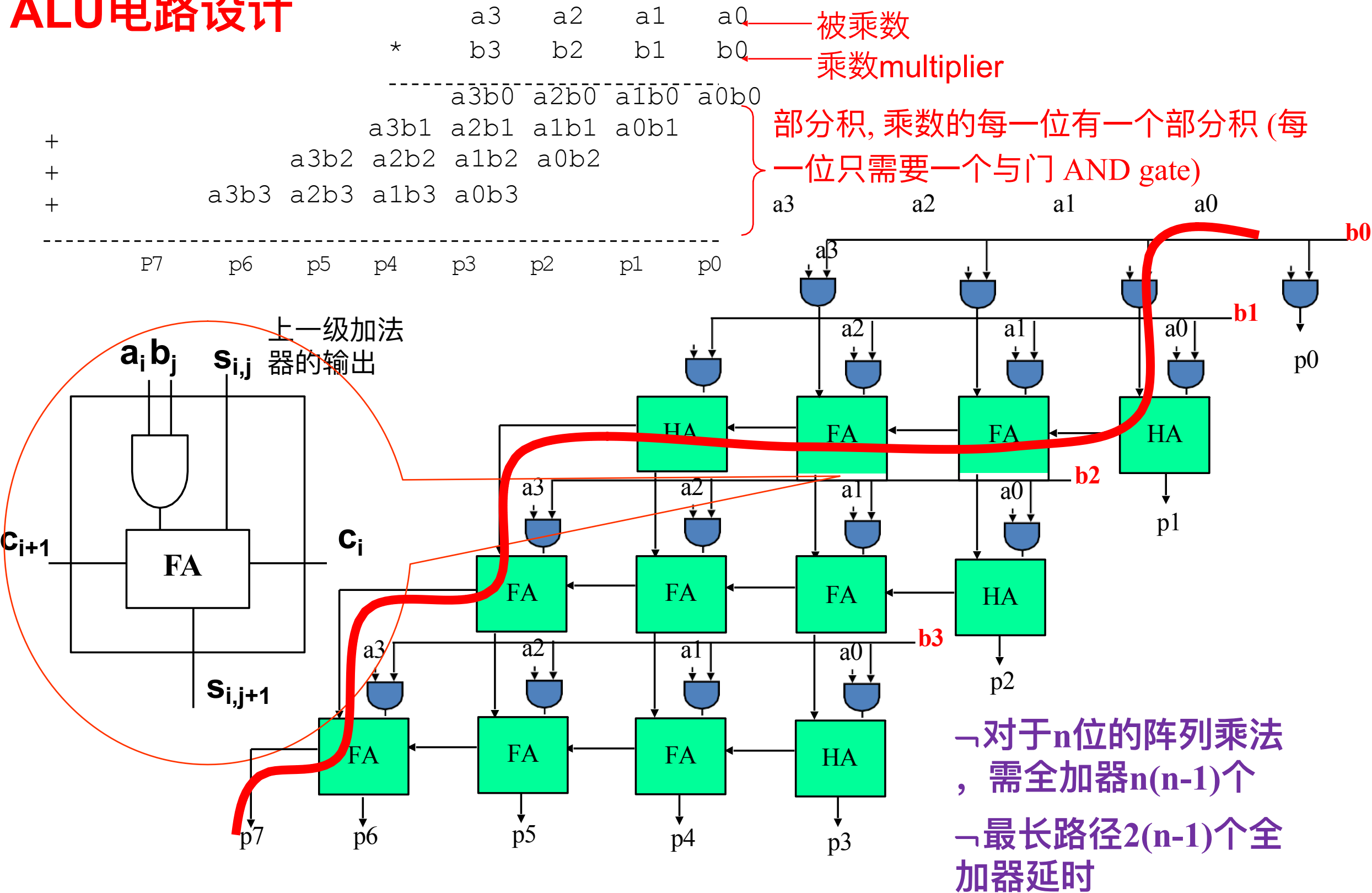


阵列乘法器

ALU电路设计

同时生成所有 n 个部分积。

每一行: n -bit 加法器和与门



阵列乘法 ALU电路设计

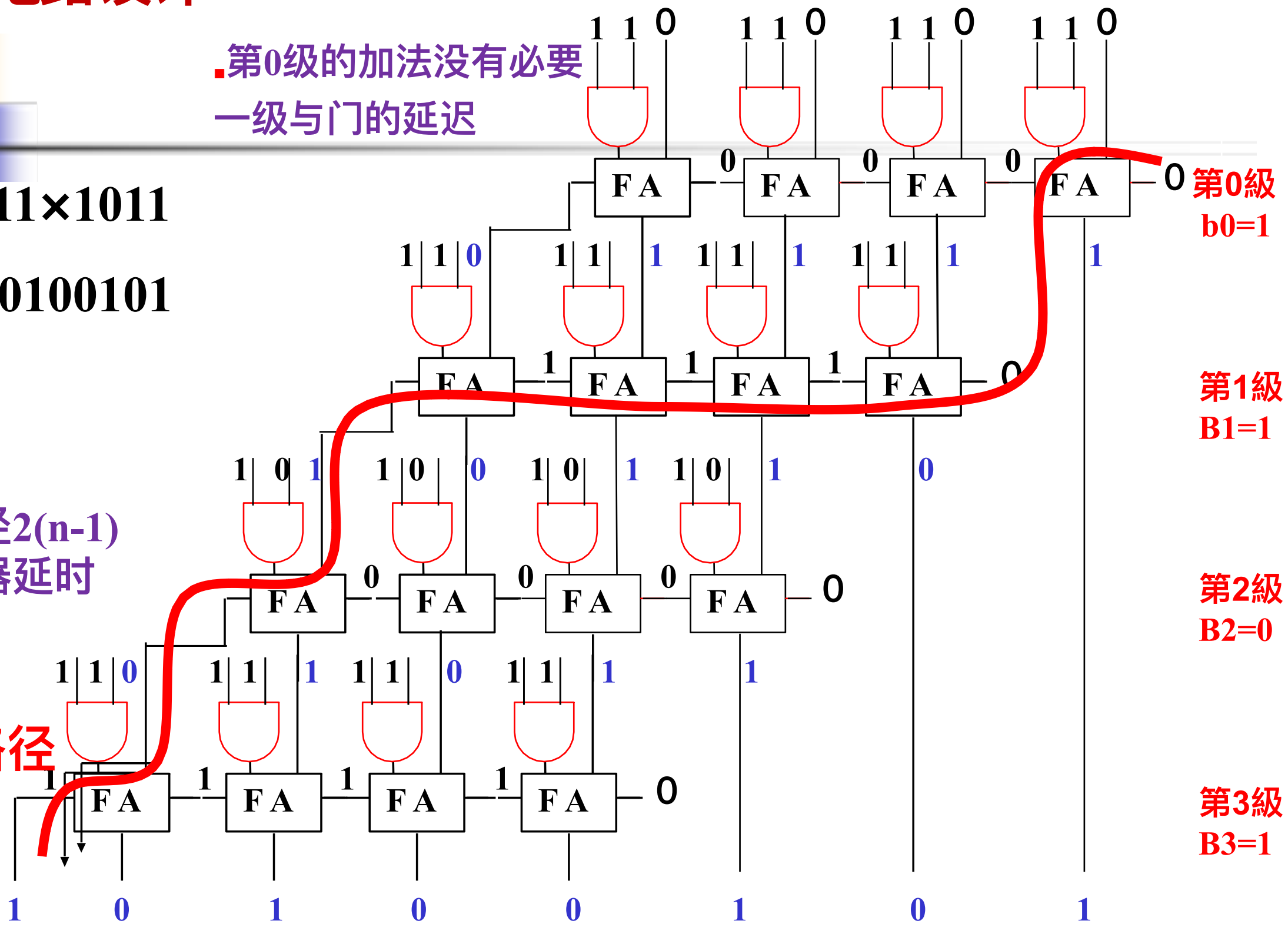
$P0=a0b0$

第0级的加法没有必要
一级与门的延迟

1111×1011
 $= 10100101$

最长路径 $2(n-1)$
个全加器延时

最长路径



阵列乘法存在的问题

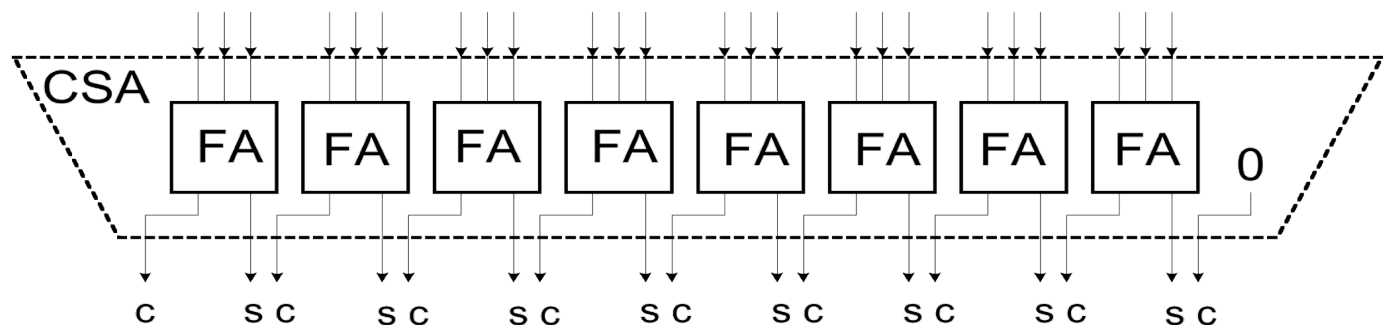
ALU电路设计

■ 存在的问题：

- 第0级的加法没有必要
- 每一级加法采用串行进位，速度受到影响

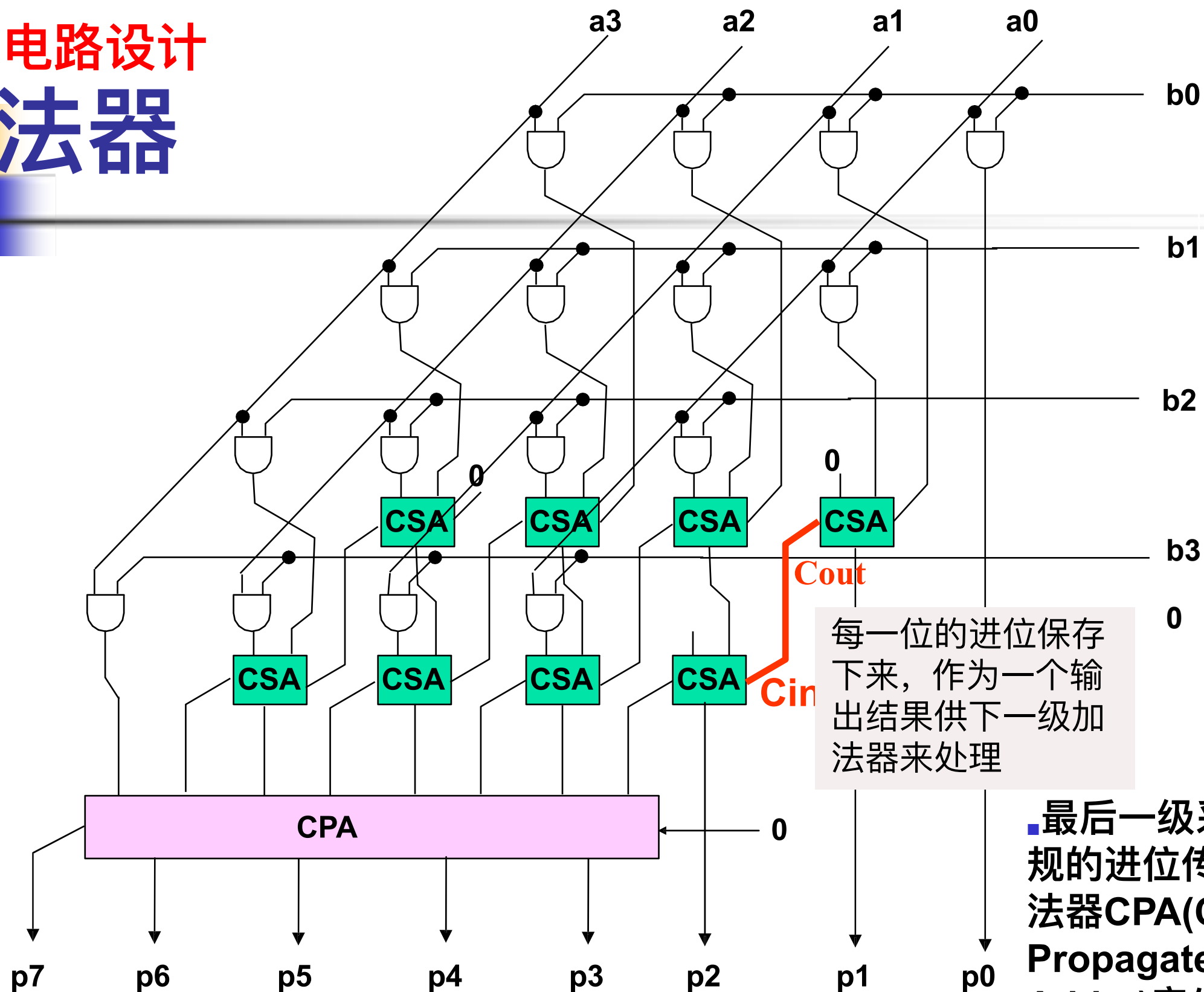
■ 改进措施：采用CSA——保存进位加法器(Carry Save Adder)

- CSA输出每一位相加的部分和，同时将每一位的进位保存下来，作为一个输出结果供下一级加法器来处理，而不是向同一级的下一位进位
- 由于CSA不存在同级每个位之间的串行进位，所以能以更快的速度得到进位和部分和



ALU电路设计

乘法器



每一位的进位保存下来，作为一个输出结果供下一级加法器来处理

■ 最后一级采用常规的进位传播加法器CPA(Carry Propagate Adder)产生最后的乘积结果

乘法

ALU电路设计

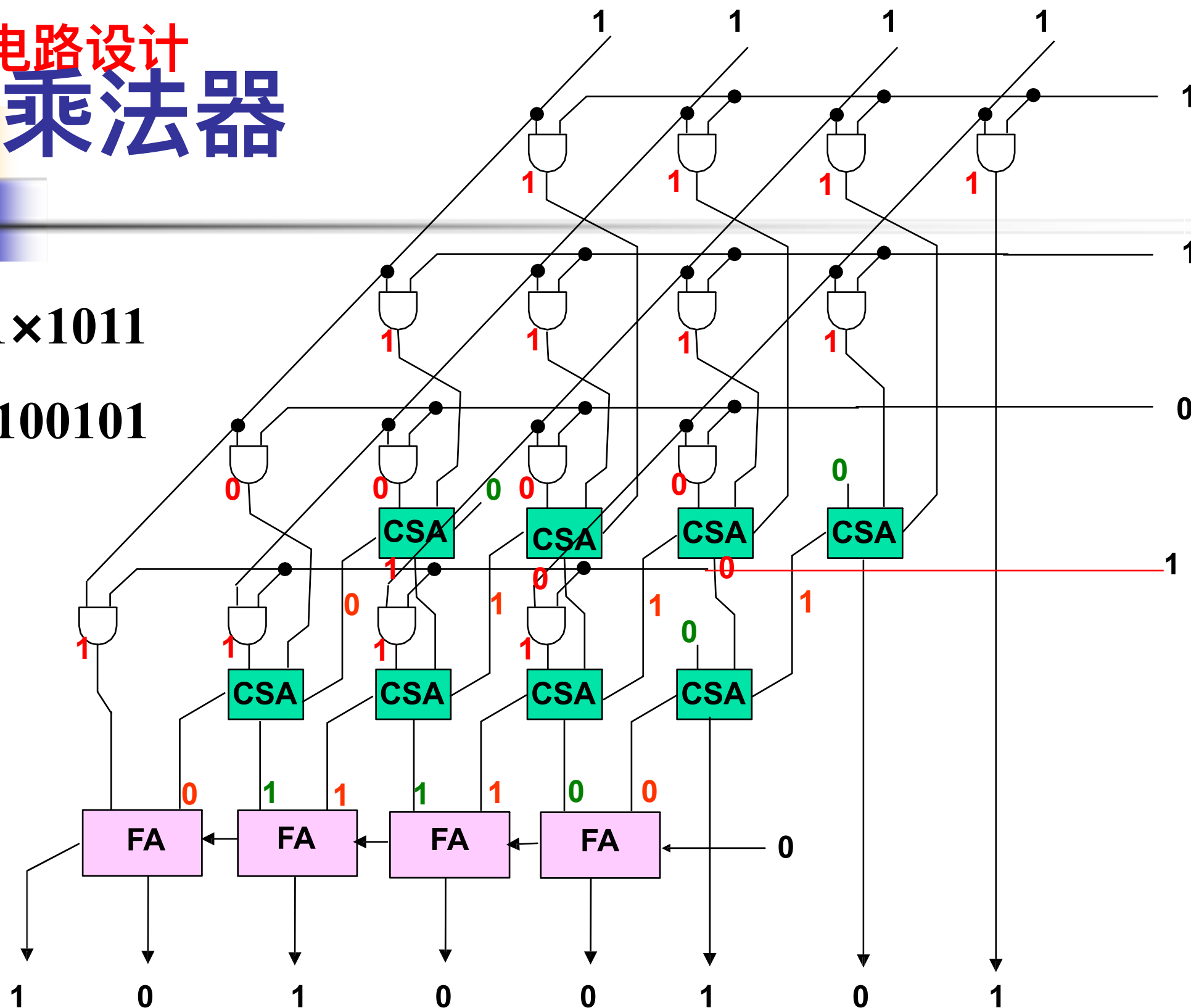
- 从结构和逻辑上看，1位CSA就是一个1位全加器
- 采用CSA构造阵列乘法器，前面几级采用不考虑进位的CSA，最后一级采用常规的进位传播加法器CPA(Carry Propagate Adder)产生最后的乘积结果
- CPA可以采用串行进位加法器，也可以采用先行进位加法器

ALU电路设计

乘法器

1111×1011

=10100101



Multiplication in MIPS

h
m
e
t
r
i
c
-
3
3

`mult $t1, $t2 # t1 * t2`

- * 没有目标寄存器：乘积可能是 $\sim 2^{64}$ ；需要两个特殊寄存器保存它
- * 3-step process:

`$t1`

01111111111111111111111111111111

`X $t2`

01000000000000000000000000000000

00011111111111111111111111111111 11000000000000000000000000000000

Hi

Lo

`mfhi $t3`

`$t3`

00011111111111111111111111111111

`mflo $t4`

`$t4`

11000000000000000000000000000000

Verilog 实现

//hi lo寄存器

```
reg [31:0] hi;
```

```
reg [31:0] lo;
```

```
wire Write_hi,Write_lo;
```

```
//always@(inst[31:26],inst[5:0])
```

```
always@(negedge clkin)
```

```
begin
```

```
  if(inst[31:26]==6'b0000000&&inst[5:0]==6'b011000)
```

```
    begin //mult
```

```
      {hi,lo}=RsData*RtData;
```

```
    end
```

```
    else if(inst[31:26]==6'b0000000&&inst[5:0]==6'b011010)
```

```
    begin //div
```

```
      lo=RsData/RtData;
```

```
      hi=RsData%RtData;
```

```
    end
```

```
    else if(inst[31:26]==6'b0000000&&inst[5:0]==6'b010001)
```

```
    begin //mthi
```

```
      RsData=hi;
```

```
    end
```

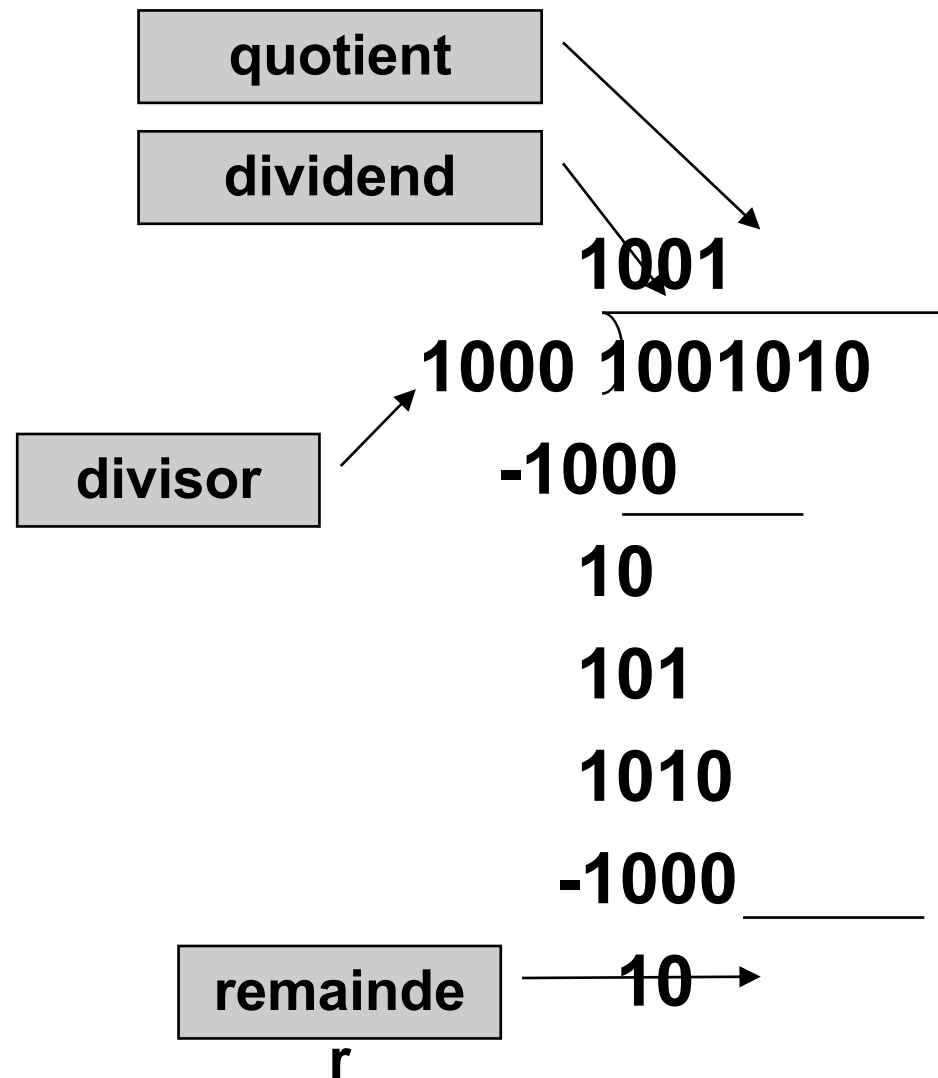
```
    else if(inst[31:26]==6'b0000000&&inst[5:0]==6'b010011)
```

```
    begin //mtlo
```

```
      RsData=lo;
```

```
    end
```

3.4 Division



n -bit 操作数产生 n -bit
商 quotient 和余数 remainder

※ 笔算除法分析：

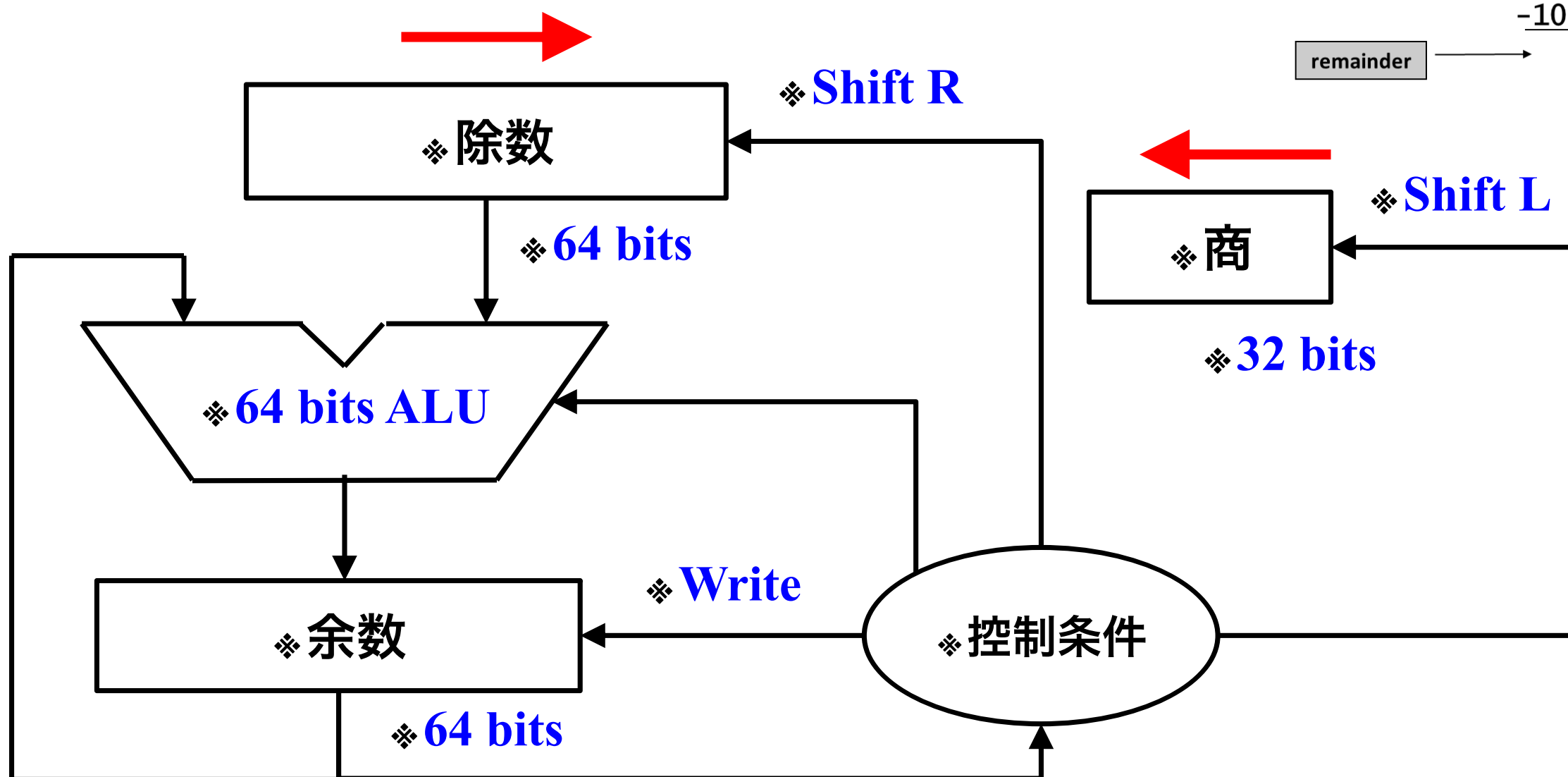
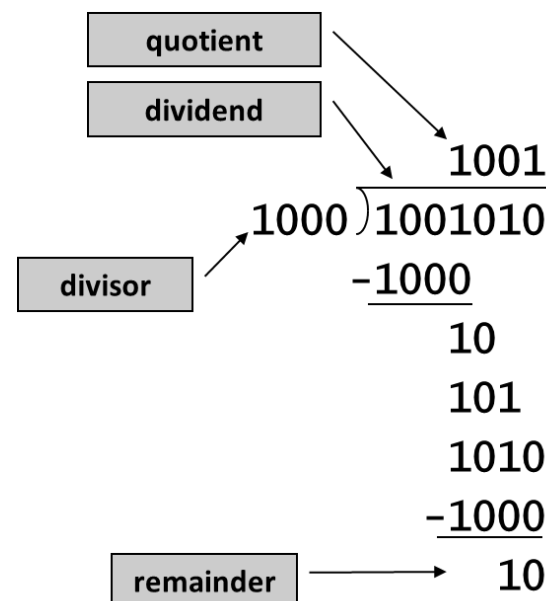
- ※ 1. 二进制除法实质是“作被除数（余数）和除数的减法，求新的余数”的过程；
- ※ 每次上商都是由心算来比较余数(被除数)和除数的大小，确定商为1还是0；
- ※ 每做一次减法，总是保持余数不动，低位补0，再减去右移后的除数；
- ※ 商符单独处理。

$$\text{被除数} = \text{商} \times \text{除数} + \text{余数}$$

※ 第一种除法算法

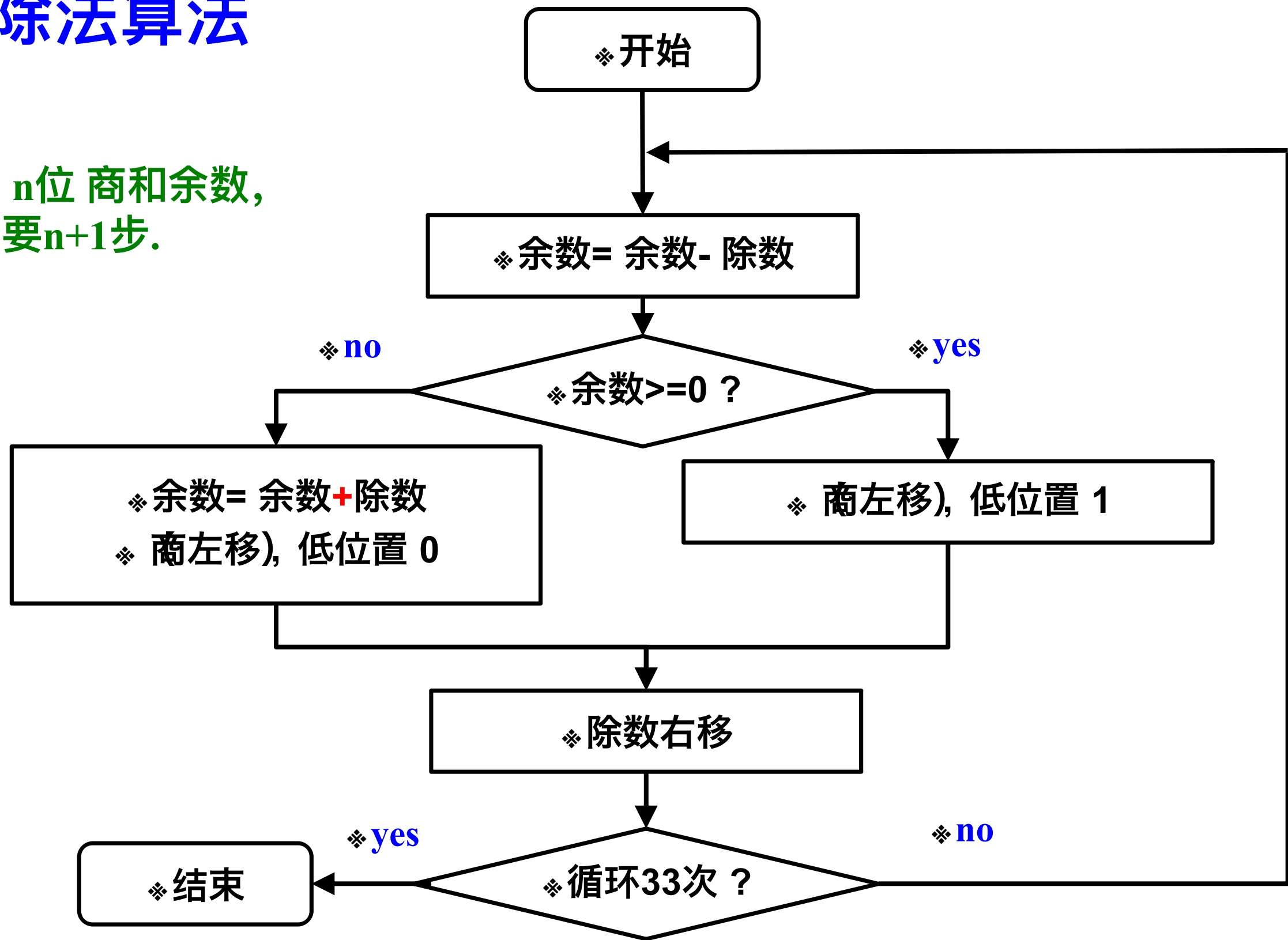
■ 32位二进制除法硬件电路

- 32位被除数放在64位余数寄存器的低32位中；
- 32位除数放在64位除数寄存器的高32位，低32位填0
- 32位商初始化为0



❖ 除法算法

对 n 位 商和余数,
需要 $n+1$ 步.



■ 计算： $7_{10} \div 2_{10} = 0111_2 \div 0010_2$

❖选代	❖步骤	❖商	❖除数	❖余数
❖0	❖初始化	❖0000	❖0010 0000	❖0000 0111
❖1	❖余数=余数-除数	❖0000	❖0010 0000	❖1110 0111
	❖余数=余数+除数 ❖商左移, LSB置0	❖0000	❖0010 0000	❖0000 0111
	❖右移除数	❖0000	❖0001 0000	❖0000 0111
❖2	❖余数=余数-除数	❖0000	❖0001 0000	❖1110 0111
	❖余数=余数+除数 ❖商左移, LSB置0	❖0000	❖0001 0000	❖0000 0111
	❖右移除数	❖0000	❖0000 1000	❖0000 0111

■ 计算： ✧ $7_{10} \div 2_{10} = 0111_2 \div 0010_2$ ✧ (续2)

✧选代	✧步骤	✧商	✧除数	✧余数
✧3	✧余数=余数-除数	✧00 00	✧0000 1000	✧ 1 110 0111
	✧余数=余数+除数 ✧商左移，LSB置0	✧0 000	✧0000 1000	✧0000 0111
	✧右移除数	✧0 000	✧0000 0100	✧0000 0111
✧4	✧余数=余数-除数	✧0 000	✧0000 0100	✧ 0 000 0011
	✧商左移，LSB置1	✧ 0001	✧0000 0100	✧0000 0011
	✧右移除数	✧ 0001	✧0000 0010	✧0000 0011

■ 计算: $\diamond 7_{10} \div 2_{10} = 0111_2 \div 0010_2 \diamond$ (续3)

※迭代	※步骤	※商	※除数	※余数
※5	※余数=余数-除数	※0001	※0000 0010	※0000 0001
	※商左移, LSB置1	※0011	※0000 0010	※0000 0001
	※右移除数	※0011	※0000 0001	※0000 0001

※补码恢复余数除法。

除数中 1/2的位数 总是为 0 => 64位加法器的1/2浪费 !!!

※是否可以用 余数左移 替代 除数右移 ?

※=> 变换次序 到 首先移位 然后再减, 可以减少一次迭代

ALU电路设计

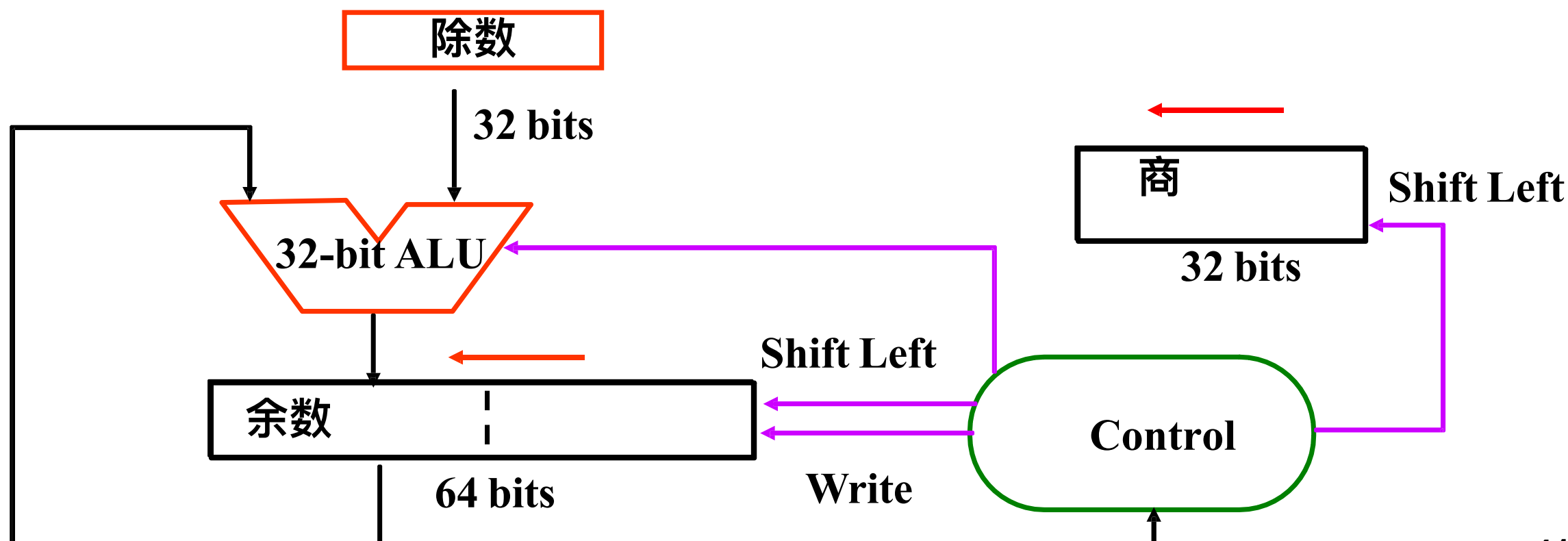
除法器

※除数和ALU的宽度减少一半

※=> 变换次序 到 首先移位 然后再减, 可以减少一次迭代

串行除法器算法2

- 将除数寄存器向右移位改为余数寄存器向左移位, 除数寄存器保持不变
- 除数寄存器为32位, ALU为32位



第二种除法算法 7/2

Remainder	Quotient	Divisor
0000 0111	0000	0010

	Q: 0000	D: 0010	R: 0000 0111
1 1: Shl R 余数左移	Q: 0000	D: 0010	R: 0000 1110
2: R = R-D 余数减去除数	Q: 0000	D: 0010	R: 1110 1110
3b: +D, sl Q, 0 余数=余数+除数, 商为0	Q: 0000	D: 0010	R: 0000 1110
2 1: Shl R 余数左移	Q: 0000	D: 0010	R: 0001 1100
2: R = R-D 余数=余数-除数	Q: 0000	D: 0010	R: 1111 1100
3b: +D, sl Q, 0	Q: 0000	D: 0010	R: 0001 1100
3 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 商左移, 商为1	Q: 0001	D: 0010	R: 0001 1000
4 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

n = 4 here

3b: 补码恢复余数

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

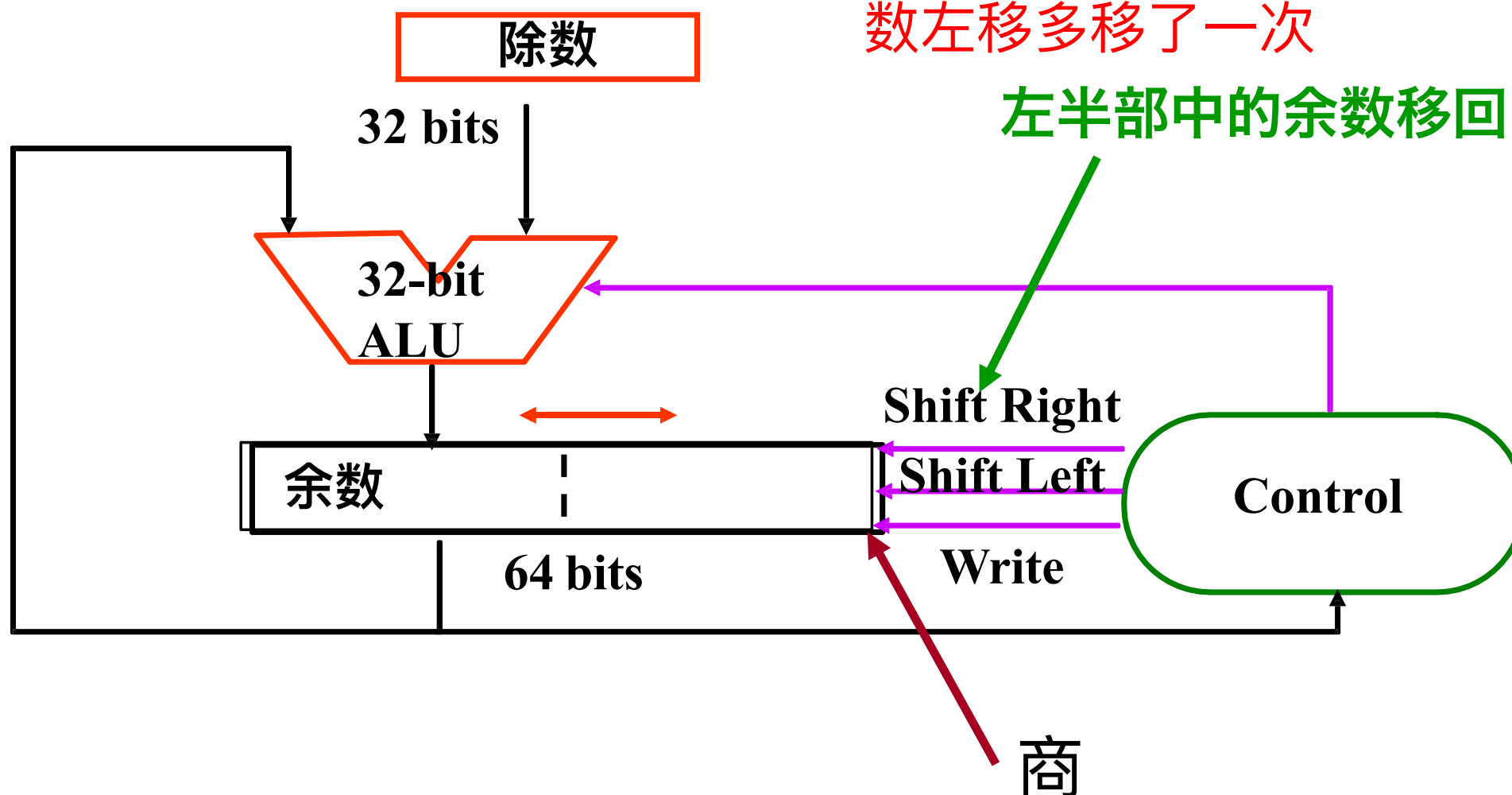
ALU电路设计

除法器

32位除数寄存器、32位ALU、64位余数寄存器(没有商寄存器), 余数寄存器左移时留出的空位正好可以存商。

■ 串行除法器算法3

两个寄存器联合在一起导致最后余数左移多移了一次

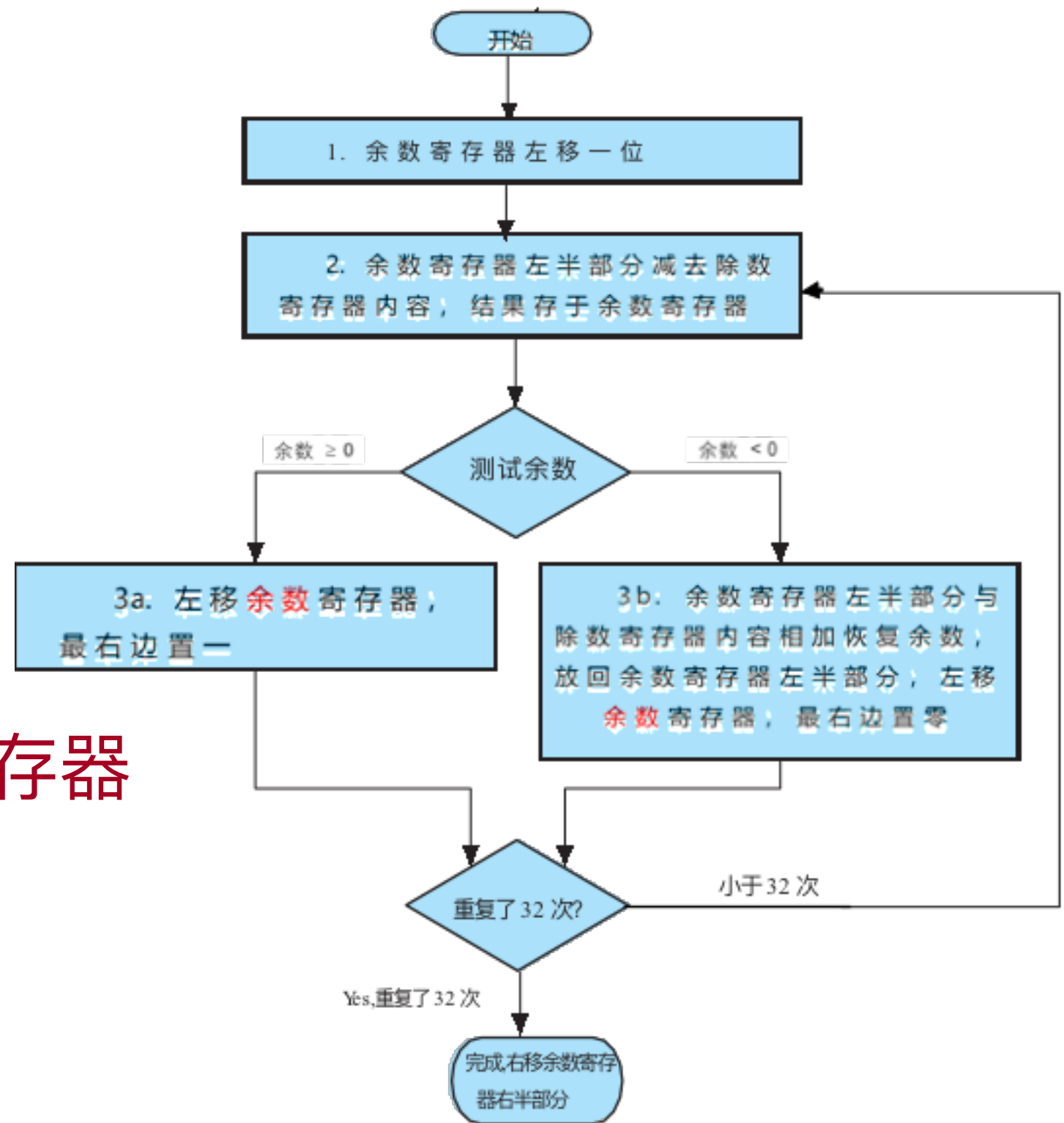


Algorithm V 3

✱ Much the same than the last one

✱ Except change of register usage

其他一样，只是商寄存器的使用变化到
余数寄存器低位



Example 7/2 for Division V3

* Well known numbers: 0000 0111/0010

iteration	step	Divisor	Remainder
0	Initial Values	0010	0000 0111
	Shift Rem left 1	0010	0000 1110
1	1.Rem=Rem-Div	0010	1110 1110
	2b: Rem<0 → +Div, sll R, R ₀ =0	0010	0001 1100
2	1.Rem=Rem-Div	0010	1111 0110
	2b: Rem<0 → +Div, sll R, R ₀ =0	0010	0011 1000
3	1.Rem=Rem-Div	0010	0001 1000
	2a: Rem>0 → sll R, R ₀ =1	0010	0011 0001
4	1.Rem=Rem-Div	0010	0001 0001
	2a: Rem>0 → sll R, R ₀ =1	0010	0010 0011
	Shift left half of Rem right 1 左半部中的余数移回		0001 0011

3.4.2 有符号除法

$$\text{被除数} = \text{除数} \times \text{商} + \text{余数}$$

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)

如果余数不与被除数符号一致, 会产生被除数和除数的符号不同, 商的绝对值得到不同的结果。

Signed division

* Keep the signs in mind for Dividend and Remainder

➤ $(+7) \div (+2) = +3$ Remainder = $+1$

$$7 = 3 \times 2 + (+1) = 6 + 1$$

➤ $(-7) \div (+2) = -3$ Remainder = -1

$$-7 = -3 \times 2 + (-1) = -6 - 1;$$

$$-7 = -4 \times 2 + (+1) = -8 + 1 \text{ 这个也满足公式)?}$$

(如果余数不与被除数符号一致, 结果的绝对值为4), 会产生商的绝对值会因为被除数和除数的符号不同而得到不同结果, **保持被除数和余数的符号必须相同可避免此**

➤ $(+7) \div (-2) = -3$ Remainder = $+1$,

$$7 = -3 \times (-2) + (+1) = 6 + 1$$

➤ $(-7) \div (-2) = +3$ Remainder = -1

➤ $-7 = 3 \times (-2) + (-1) = -6 - 1$

※ **被除数 = 商 * 除数 + 余数**

※ **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**

❖ MIPS中的除法

符号数除法指令

- **div \$s2, \$s3**
- ❖ **# Lo = \$s2 / %s3; 商**
- ❖ **# Hi = \$s2 % \$s3; 余数**

无符号数除法指令

- **divu \$s2, \$s3**
- ❖ **# Lo = \$s2 / %s3; 商**
- ❖ **# Hi = \$s2 % \$s3; 余数**

➤ 取除法运算结果指令

- ❖ **mflo \$s1 # \$s1 = Lo**
- ❖ **mfhi \$s1 # \$s1 = Hi**

**No overflow or
divide-by-0
checking**

**Software must
perform checks if
required**

❖ Faster Division

- ✱ 快速乘法算法不适用于除法，因为每步迭代都需要知道减法结果的符号
- ✱ 更快的除法器（例如 SRT 除法）每步生成多个商位
 - 基于被除数和余数的高位通过表查找来猜测商位
 - 后续步骤纠正错误猜测
 - 仍需多步完成
- ✱ 也有其它快速除法器
 - nonrestoring dividers 非恢复除法器), nonperforming dividers, ...

❖ 数的表示

N bits 我们可以表示数的范围?

Unsigned: $0 \rightarrow 2^n - 1$

Signed: $-2^{n-1} \rightarrow 2^{n-1} - 1$

What about:

非常大的数怎样表示?

9,349,787,762,244,859,087,678

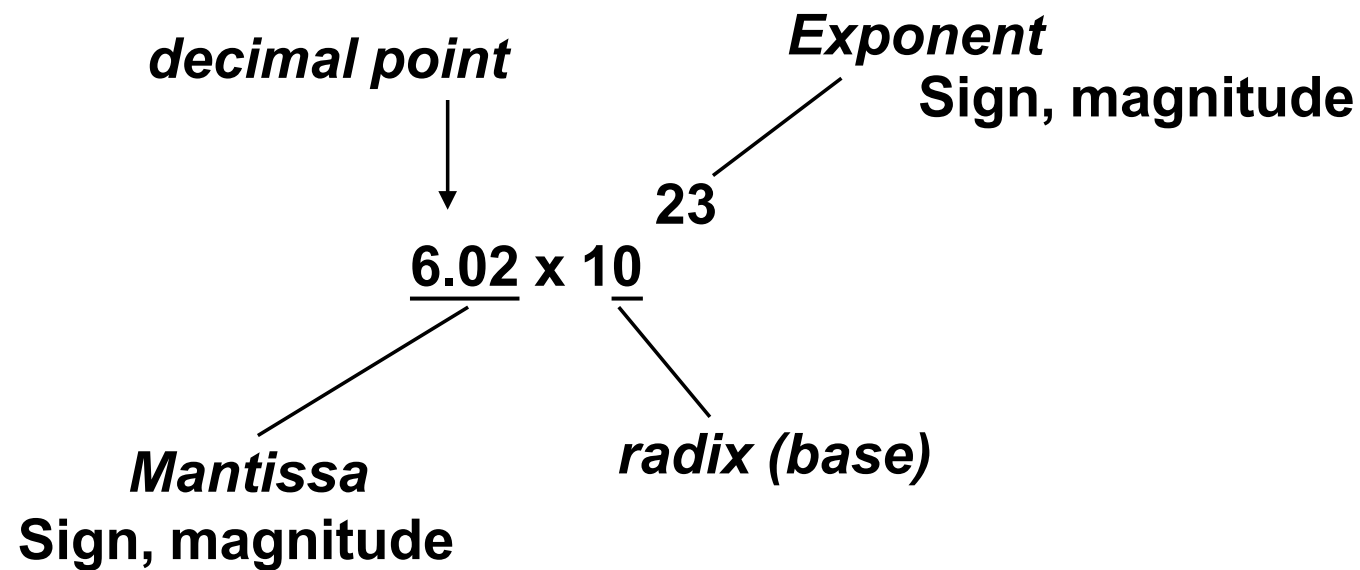
非常小的数怎样表示?

0.000000000000000000000000004691

无理数怎样表示?

$2/3$

❖ 科学记数法



- In Binary:

radix = 2

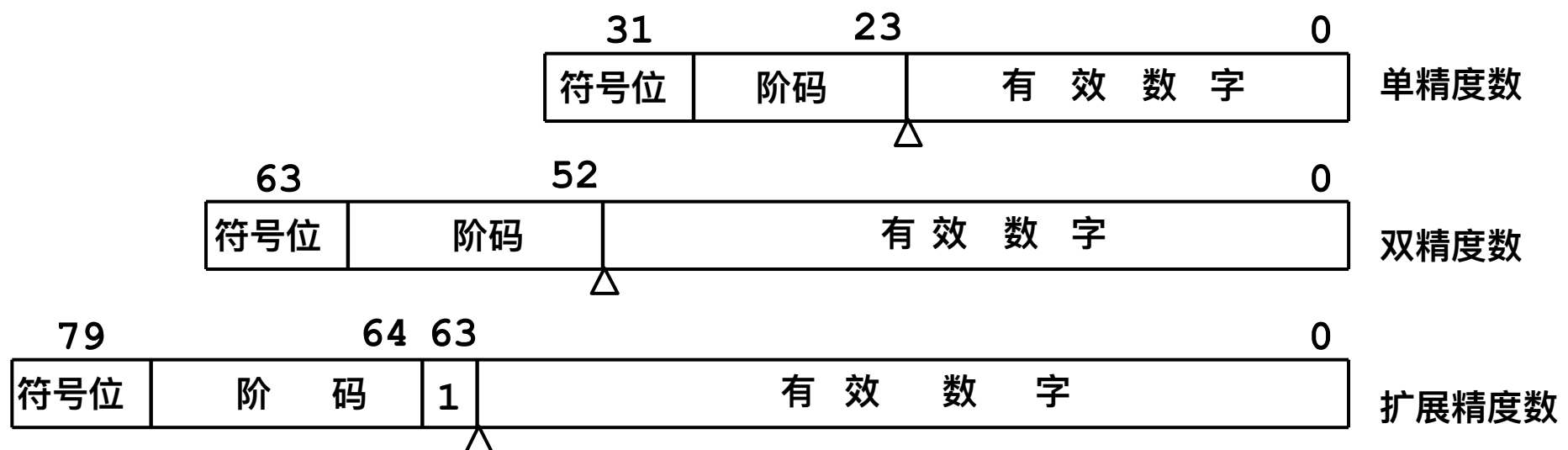
$$\text{value} = (-1)^s \times M \times 2^E$$



ALU电路设计

浮点运算器

IEEE 754标准浮点数格式



- 32位单精度——1位符号，8位指数，23位尾数
- 64位双精度——1位符号，11位指数，52位尾数
- 80位扩展精度——1位符号，15位指数，64位尾数
- 数阶码以一种偏置形式存放

3.5.1 浮点表示

■ 浮点数的一般表示形式：

- ※ S=0：表示正数
- ※ S=1：表示负数

※ $(-1)^S \times F \times 2^E$

※ 符号

※ 指数

※ 尾数

■ IEEE754浮点数的编码

S	阶码E	尾数M
---	-----	-----

※ IEEE754的3种浮点表示格式

类型	S	E	M	总位数	偏移值
单精度浮点数	1	8	23	32	127
双精度浮点数	1	11	52	64	1023
扩展精度浮点数	1	15	64	80	16383

3.5.1 浮点表示

■ IEEE754编码格式

单精度		双精度		表示的数
指数	尾数	指数	尾数	
0	000	0	000	0
0	非零	0	非零	\pm 非规格化数
1~254	任意	1~2046	任意	\pm 浮点数
255	0	2047	0	$\pm\infty$
255	非零	2047	非零	非数

非数NaN的表示

$\text{Sqrt}(-4.0) = ?$ $0/0 = ?$

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

非数NaN的计算机表示

指数 **Exponent** = 255

尾数 **Significand**: nonzero 非零

NaNs can help with debugging

Operations

$\text{sqrt}(-4.0) = \text{NaN}$

$0/0 = \text{NaN}$

$\text{op}(\text{NaN}, x) = \text{NaN}$

$+\infty + (-\infty) = \text{NaN}$

$+\infty - (+\infty) = \text{NaN}$

$\infty/\infty = \text{NaN}$

etc.

Representation for 0

How to represent 0?

exponent: all zeros

significand: all zeros

What about sign? Both cases valid.

+0: 0 00000000 000000000000000000000000

-0: 1 00000000 000000000000000000000000

$+\infty/-\infty$ 的表示

∞ : infinity

在浮点数中, 除数为0的结果是 $+/-\infty$, 不是溢出异常.

为什么要这样处理?

- 可以利用 $+\infty/-\infty$ 作比较。例如: $X/0 > Y$ 可作为有效比较

表示无穷大 $+\infty/-\infty$?

- 指数全是 1 (11111111B = 255)
- 尾数全是 0

$+\infty$: 0 11111111 000000000000000000000000

$-\infty$: 1 11111111 000000000000000000000000

Operations

$$5 / 0 = +\infty,$$

$$-5 / 0 = -\infty$$

$$5 + (+\infty) = +\infty,$$

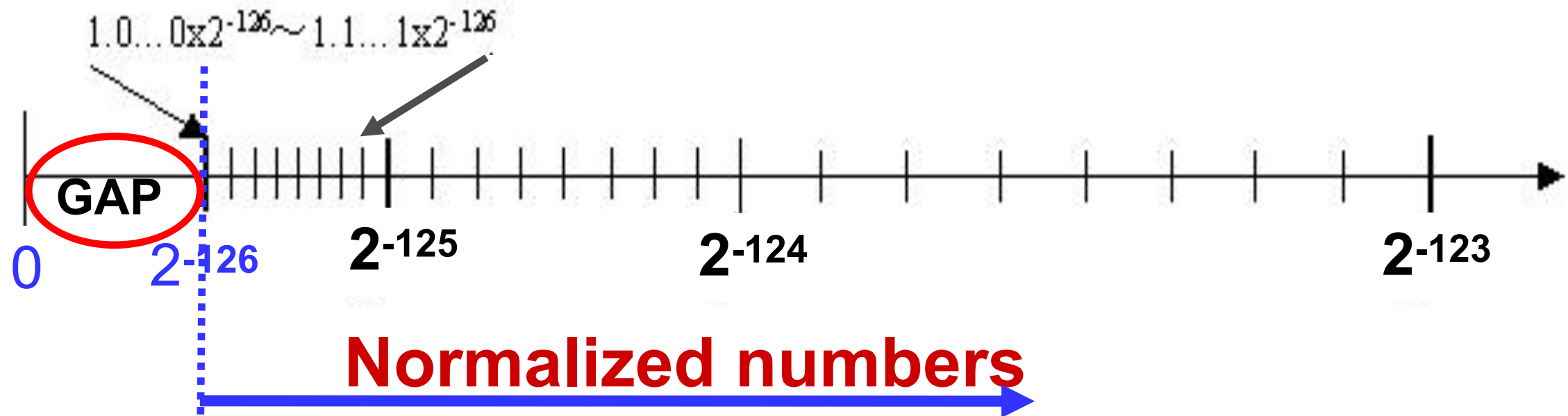
$$(+\infty) + (+\infty) = +\infty$$

$$5 - (+\infty) = -\infty,$$

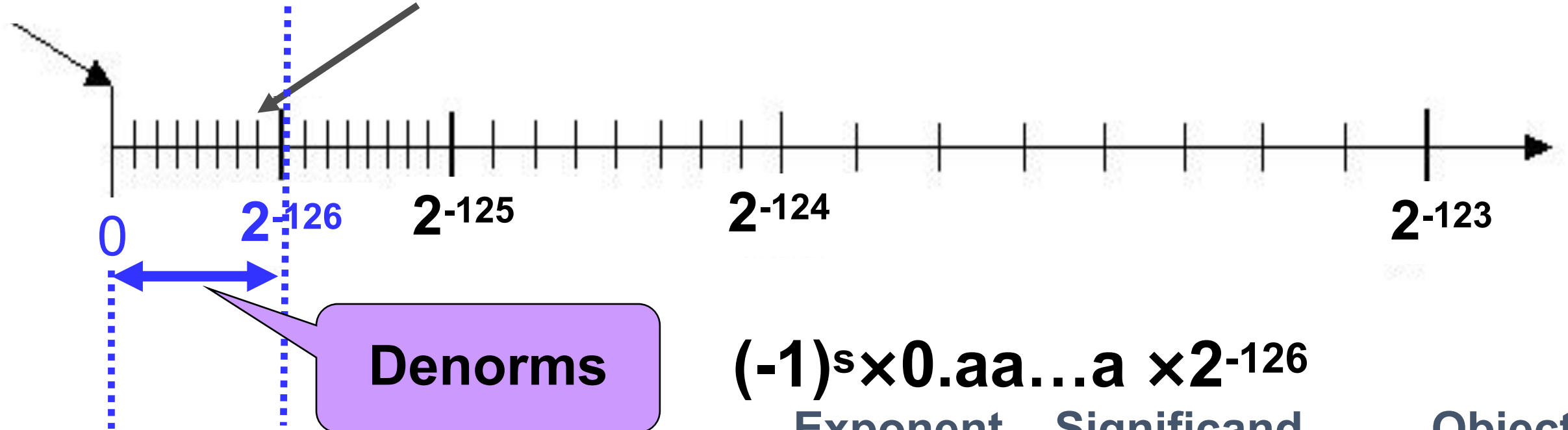
$$(-\infty) - (+\infty) = -\infty \quad \text{etc}$$

非规格化数的计算机浮点表示

$1.0...0 \times 2^{-126} \sim 1.1...1 \times 2^{-126}$



$0.0...0 \times 2^{-126} \sim 0.1...1 \times 2^{-126}$



$(-1)^s \times 0.aa...a \times 2^{-126}$

Exponent

Significand

Object

0

nonzero

单精度浮点数表示范围

✱ 指数在 00000000 和 11111111 之间

✱ 最小数 (规格化数)

- 指数: 00000001

⇒ 实际指数 = $1 - 127 = -126$

- 小数: 000...00 ⇒ 尾数 significand = 1.0

- $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$

✱ 最大数

- 指数: 11111110

⇒ 实际指数 = $254 - 127 = +127$

- 小数: 111...11 ⇒ 尾数 significand ≈ 2.0

- $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

双精度浮点数表示范围

✱ 指数在 0000...00 和 1111...11 之间

✱ 最小数 (规格化数)

- 指数: 000000000001

⇒ 实际指数 = $1 - 1023 = -1022$

- 小数: 000...00 ⇒ 尾数 significand = 1.0

- $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$

✱ 最大数

- 指数: 111111111110

⇒ 实际指数 = $2046 - 1023 = +1023$

- 小数: 111...11 ⇒ 尾数 significand ≈ 2.0

- $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

非规格化

指数0, 尾数000....1

$0x000000001 = 2^{-23} \times 2^{-126}$

浮点数精度

* 相对精度

- 所有小数位都是有效的
- 单精度: 大约 2^{-23}
 - ▶ 等价于 $23 \times \log_{10} 2 \approx 23 \times 0.3 \approx 6$ 小数点后面6位十进制数的精度
- 双精度: approx 2^{-52}
 - ▶ 等价于 $52 \times \log_{10} 2 \approx 52 \times 0.3 \approx 16$ 小数点后面16位十进制数的精度

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	\pm denormalized number
1–254	Anything	1–2046	Anything	\pm floating-point number
255	0	2047	0	\pm infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Floating-Point Example

* Represent -0.75

- $-0.75 = -0.11_2 = (-1)^1 \times 1.1_2 \times 2^{-1}$
- $S = 1$
- Fraction = $1000\dots00_2$
- Exponent = $-1 + \text{Bias}$
 - Single: $-1 + 127 = 126 = 01111110_2$
 - Double: $-1 + 1023 = 1022 = 011111111110_2$

* Single: $1011111101000\dots00$

* Double: $10111111111101000\dots00$

Limitations

- * **Overflow:**

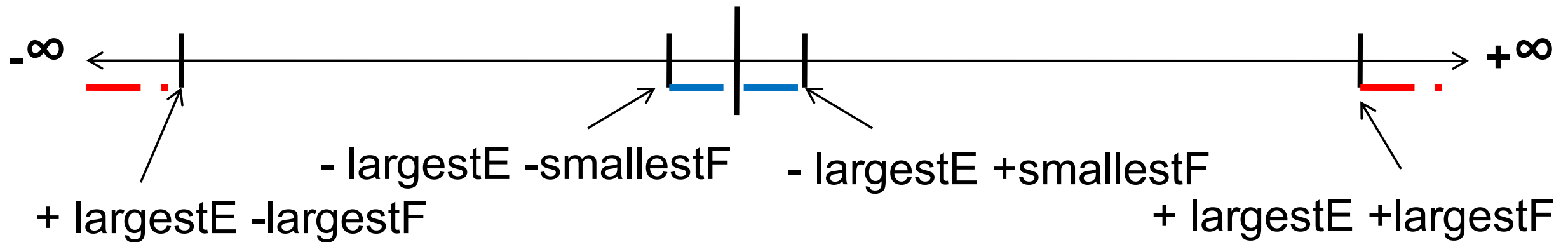
- The number is too big to be represented

- * **Underflow:**

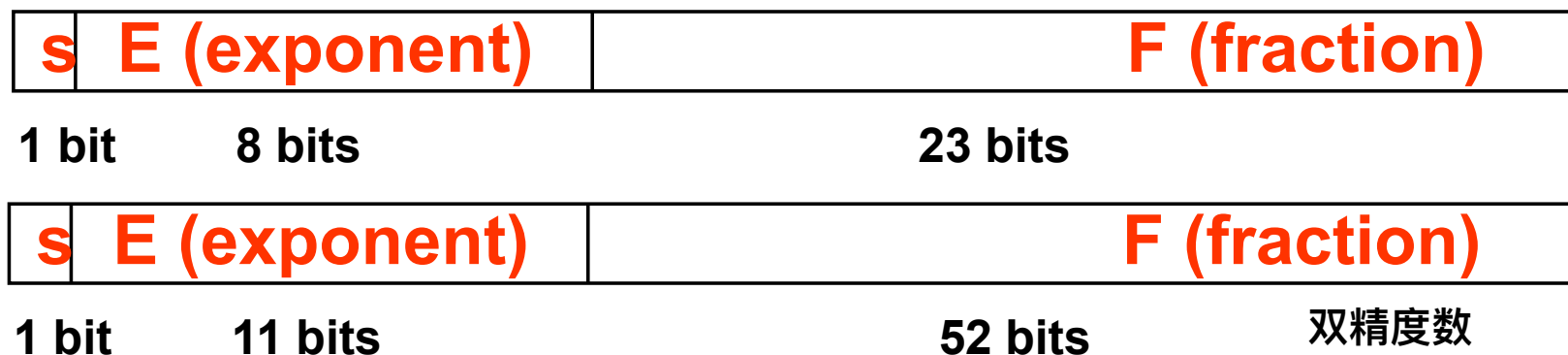
- The number is too small to be represented

浮点数中的异常事件

- **Overflow 上溢** (浮点数) 当正指数变得太大而超出指数字段表示范围时发生
- **Underflow 下溢** (浮点) 在负指数变得太大而超出指数字段表示范围时发生



- 减少下溢或上溢机会的一种方法是提供另一种具有更大指数字段的格式
- 双精度浮点数 – 用 MIPS 两个字表示



浮点数加减算法

- $(\pm F1 \times 2^{E1}) + (\pm F2 \times 2^{E2}) = \pm F3 \times 2^{E3}$

Step 0: 恢复 F1 和 F2 中的隐藏位

Step 1: **Align对阶** 使两数的指数相等，即对齐小数点，
原则：小指数向大指数看齐

如果 $E1 - E2$ 为正，F2 右移 $E1 - E2$ 的差值，否则右移F1

Step 2: 尾数相加减，F2 加 F1 得到F3

Step 3: 对结果进行规格化 F3表示成1.XXXXXX ...的形式

Step 4: 对舍入处理，**Round** F3

舍1入。移去的最高位为1, F3的最低位+1

如果+1后，F3又溢出，再次规格化

Step5: 检查是否溢出

下溢，机器0

上溢，置溢出标志

Step 6: 存储结果之前隐含F3的最高位

以十进制为例

* 两个数对齐

$$9.999 \cdot 10^1$$

$$0.01610 \cdot 10^1 \rightarrow 0.016 \cdot 10^1 \quad \text{Truncation}$$

* Addition

$$\begin{array}{r} 9.999 \cdot 10^1 \\ + 0.016 \cdot 10^1 \\ \hline 10.015 \cdot 10^1 \end{array}$$

* 规格化

$$1.0015 \cdot 10^2$$

* 四舍五入

$$1.002 \cdot 10^2$$

以二进制为例 $y=0.5+(-0.4375)$

* $0.5_{10} = 1.000_2 \times 2^{-1}$

* $-0.4375_2 = -1.110_2 \times 2^{-2}$

* Step1: 小指数向大指数看齐, 右移差值

$$-1.110_2 \times 2^{-2} \rightarrow -0.111_2 \times 2^{-1}$$

* Step2: 尾数相加

$$\begin{array}{r} 1.000_2 \times 2^{-1} \\ +) - 0.111_2 \times 2^{-1} \\ \hline 0.001_2 \times 2^{-1} \end{array}$$

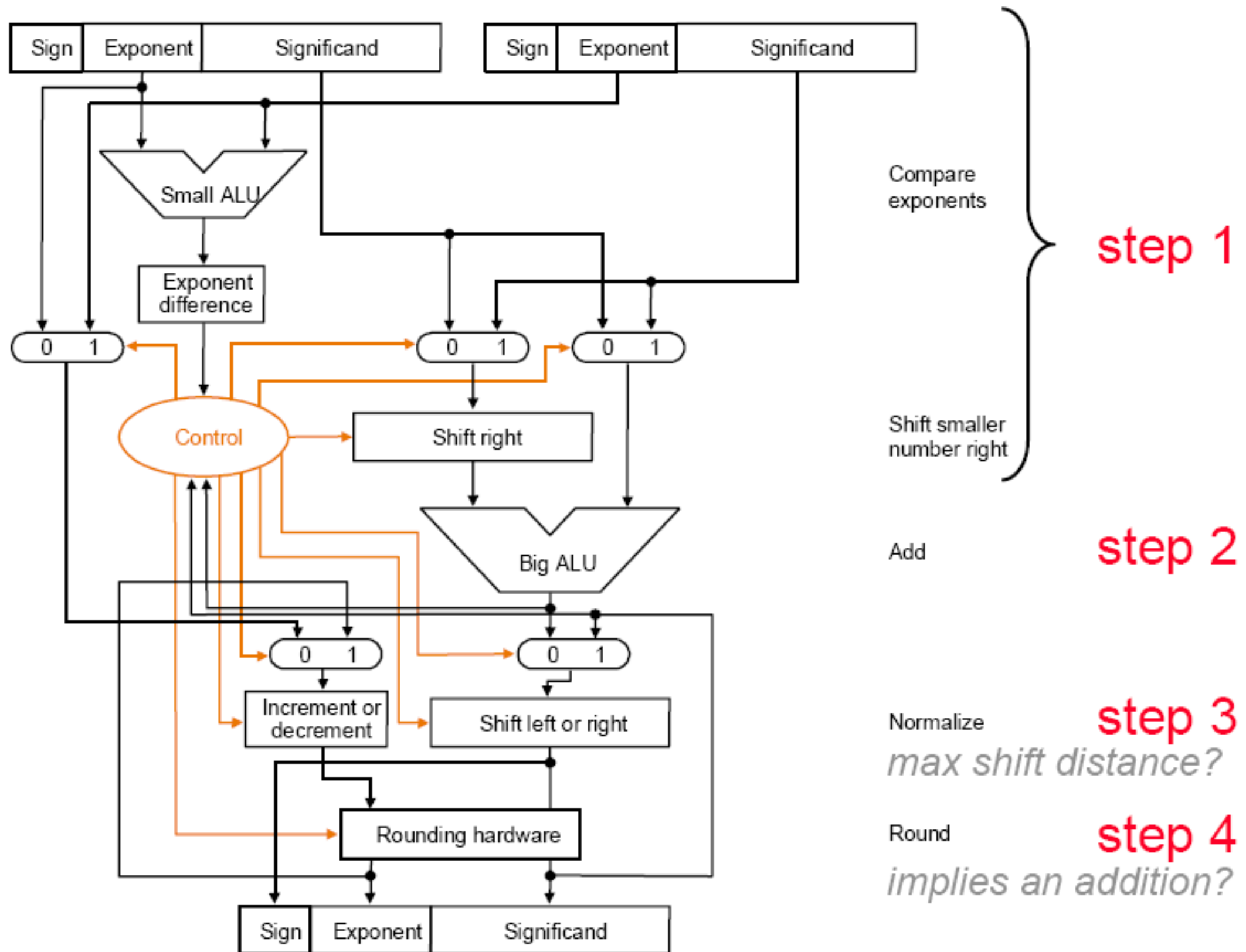
* Step3: 规格化结果, 检查是否溢出 overflow or underflow

* $0.001_2 \times 2^{-1} \rightarrow 0.010_2 \times 2^{-2} \rightarrow 0.100_2 \times 2^{-3} \rightarrow 1.000_2 \times 2^{-4}$

* Step4: 舍入处理结果

$$1.000_2 \times 2^{-4} = 0.0625_{10}$$

浮点数加法的硬件实现



浮点数乘法算法

$$(\pm F1 \times 2^{E1}) \times (\pm F2 \times 2^{E2}) = \pm F3 \times 2^{E3}$$

Step 0: 恢复F1 和 F2的隐含位

Step 1: 指数（包含偏移值）相加，再减去偏差值，

$$E1 + E2 - 127 = E3$$

确定乘积符号，由符号位决定

Step 2: 尾数相乘 $F1 * F2 = F3$

Step 3: 规格化 F3 (so it is in the form 1.XXXXXX ...)

- F3右移一位，E3递增1
- 检查是否溢出 overflow/underflow

Step 4: F3舍入，再次规格化 F3

Step 5: 存储结果前隐含1. xxx 中的1

举例

$$(\pm F1 \times 2^{E1}) \times (\pm F2 \times 2^{E2}) = \pm F3 \times 2^{E3} \quad E1 + E2 - 127 = E3$$

✱ 指数和尾数单独处理，尾数相乘，指数相加

$$(s1 \cdot 2^{e1}) \cdot (s2 \cdot 2^{e2}) = (s1 \cdot s2) \cdot 2^{e1+e2}$$

$$1 \ 10000010 \quad 000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 = -1 \times 2^3$$

$$0 \ 10000011 \quad 000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 = 1 \times 2^4$$

✱ 两个数的符号异号 \rightarrow product = 1 \rightarrow Sign=1

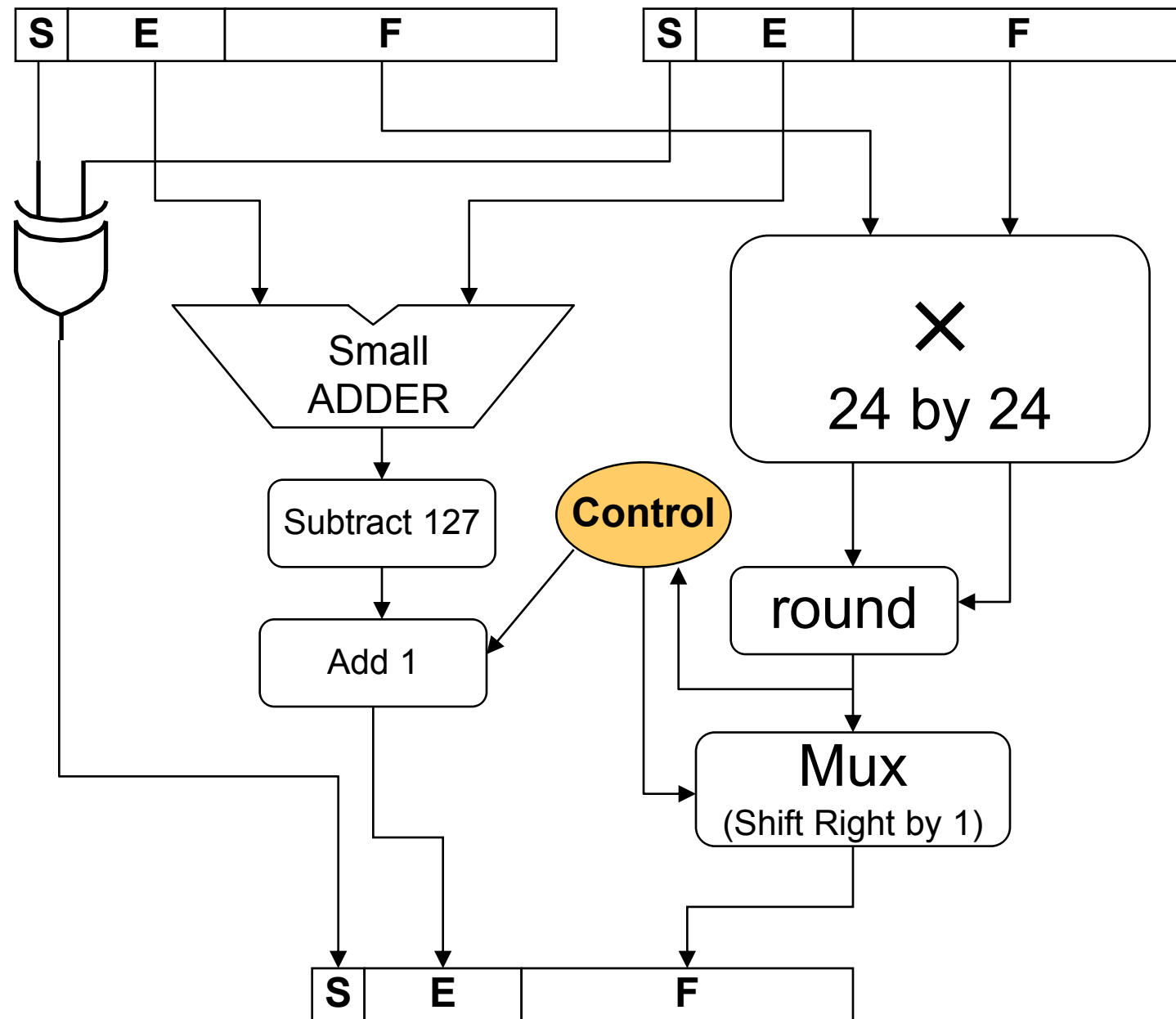
✱ 指数相加, bias = 127

$$\begin{array}{r} 10000010 \\ +10000011 \\ \hline 110000101 \end{array}$$

校正: $110000101 - 01111111 = 10000110 = 134 = 127 + 3 + 4$

• 结果: $1 \ 10000110 \ 000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 = -1 \times 2^7$

Floating-Point Multiplication



Step 1:

尾数相乘，指数相加

$$E_R = E_1 + E_2 - 127$$

Step 2:

M=尾数相乘结果，舍入处理

If $M \geq 2$, 右移 M 一位，
指数增一，如果指数超出范围则溢出，

乘完的结果舍入处理

浮点数乘法举例

* 4-digit 二进制数为例

- $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2} \quad (0.5 \times -0.4375)$

* 1. 指数相加

- Unbiased: $-1 + -2 = -3$

- Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$

* 2. 尾数相乘

- $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$

* 3. 规格化结果，检查是否溢出 over/underflow

- $1.110_2 \times 2^{-3}$ (no change) 没有溢出 over/underflow

* 4. 如果必要再次规格化和舍入

- $1.110_2 \times 2^{-3}$ (no change)

* 5. 确定符号 sign: +ve \times -ve \Rightarrow -ve

- $-1.110_2 \times 2^{-3} = -0.21875$

Division-- Brief

- * Subtraction of exponents
- * Division of the significants
- * Normalisation
- * Runding
- * Sign

Accurate Arithmetic

✱ IEEE754在中间计算时，右边总是多保留2位，第一位为“保护位”，第二位为“舍入位”。

✱ 带舍入保护位 Guard Digits

- $2.56_{\text{ten}} \times 10^0 + 2.34_{\text{ten}} \times 10^2$

$$\begin{array}{r} 2.3400_{\text{ten}} \\ + 0.0256_{\text{ten}} \\ \hline 2.3656_{\text{ten}} \end{array}$$

- 保留小数点后两位，保护位是5，入1.

- $\text{Sum} = 2.37_{\text{ten}} \times 10^2.$

✱ 不带保护位

- $\text{Sum} = 2.36_{\text{ten}} \times 10^2$

$$\begin{array}{r} 2.34_{\text{ten}} \\ + 0.02_{\text{ten}} \\ \hline 2.36_{\text{ten}} \end{array}$$

✱ **保护位(Guard Digits):** 在浮点数中间计算中, 在尾数中右边多保留两位, 第一位为保护位, 第二位为舍入位。

✱

✱ **舍入位(Round Digits):** 规格化后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。

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。

。

✱ **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。

为了更高的准确性, 当舍入数字为 $B/2$ 时, 需要一个粘位, 即如果有在舍入数字的尾部有任何1位丢失, 则置为1。

IEEE 754 round modes

*IEEE*标准:

四种舍入方式:

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:

舍入成 最接近的数 (省缺)

舍入数字 $< B/2$, 那么 截去 (B数值大小)

$> B/2$, 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

四舍五入方法中的偶数法则:标准的方法是**四舍六入,五看奇偶**.如果是5的话,就看5后面还有没有数,有数就入,如果5后面都是0了,则看前面一位,是偶数就舍,是奇数就入.

y	round down (towards $-\infty$)	round up (towards $+\infty$)	round towards zero	round away from zero	round to nearest
+23.67	+23	+24	+23	+24	+24
+23.50	+23	+24	+23	+24	+23 or +24
+23.35	+23	+24	+23	+24	+23
+23.00	+23	+23	+23	+23	+23
0	0	0	0	0	0
-23.00	-23	-23	-23	-23	-23
-23.35	-24	-23	-23	-24	-23
-23.50	-24	-23	-23	-24	-23 or -24
-23.67	-24	-23	-23	-24	-24

二进制数舍入偶数法则

二进制小数

如果是一半的话,就看后面还有没有数,有数就入,如果一半后面都是0了,则看前面一位,是0就舍,是1就入.

如果右边 = $10[0]_2$.看前面是0则舍,前面是1则入。

Examples, 保留小数点后面两位 ($1/2 = 100$)

Value	Binary	Rounded	Action	Rounded Value
$2 \frac{3}{32}$	10.00011_2	10.00	($< 1/2$: down)	2
$2 \frac{3}{16}$	10.00110_2	10.01	($> 1/2$: up)	$2 \frac{1}{4}$
$2 \frac{7}{8}$	10.11100_2	11.00	($1/2$: up)	3 一半前面是1则入
$2 \frac{5}{8}$	10.10100_2	10.10	($1/2$: down)	$2 \frac{1}{2}$ 一半前面是0则舍

For Example

- 手算 $3.984375 \times 10^{-1} + (3.4375 \times 10^{-1} + 1.771 \times 10^3)$, 假设有*
1位保护位, 1位舍入位和1位黏贴位, 并采用向最靠近的偶数舍入的模式, 给出所有步骤。(16位精度表示, 1位符号位, 指数5位, 尾数10位)
- 解:
- $3.984375 \times 10^{-1} = 1.1001100000 \times 2^{-2}$
- $3.4375 \times 10^{-1} = 1.0110000000 \times 2^{-2}$
- $1.771 \times 10^3 = 1771 = 1.1011101011 \times 2^{10}$
- shift binary point of smaller left 12 so exponents match
- (B) 0.0000000000 01 0110000000 Guard 0, Round 1, Sticky 1
- (C) 1.1011101011
- (B+C) = 1.1011101011
- A 0.0000000000 011001100000
- A+(B+C) + 1.1011101011 No round
- $A+(B+C) + 1.1011101011 \times 2^{10} = 0110101011101011 = 1771$

*** Example** Calculate the sum of A and B by hand, assuming A and B are stored in the modified 16-bit NVIDIA format. Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps.



a. $2.6125 \times 10^1 + 4.150390625 \times 10^{-1}$

b. $-4.484375 \times 10^1 + 1.3953125 \times 10^1$

* Solution:

a.

$$2.6125 \times 10^1 + 4.150390625 \times 10^{-1}$$

$$2.6125 \times 10^1 = 26.125 = 11010.001 = 1.1010001000 \times 2^4$$

$$4.150390625 \times 10^{-1} = .4150390625 = .011010100111$$

$$= 1.1010100111 \times 2^{-2}$$

Shift binary point 6 to the left to align exponents,

GR

$$1.1010001000 \ 00$$

$$+ .0000011010 \ 10 \ 0111 \text{ (Guard = 1, Round = 0, Sticky = 1)}$$

$$1.1010100010 \ 10 \ 1$$

In this case the extra bits (G,R,S) are more than half of the least significant bit (0).

Thus, the value is rounded up.

$$1.1010100011 \times 2^4 = 11010.100011 \times 2^0 = 26.546875 \\ = 2.6546875 \times 10^1$$

* Solution:

b.

$$-4.484375 \times 10^1 + 1.3953125 \times 10^1$$

$$-4.484375 \times 10^1 = -44.84375 = -1.0110011011 \times 2^5$$

$$1.1953125 \times 10^1 = 11.953125 = 1.0111111010 \times 2^3$$

Shift binary point 2 to the left and align exponents,

GR

$$\begin{array}{r} -1.0110011011 \ 00 \\ \end{array}$$

$$\begin{array}{r} 0.0101111110 \ 10 \text{ (Guard = 1, Round = 0, Sticky = 0)} \\ \hline \end{array}$$

$$\begin{array}{r} -1.0000011100 \ 10 \\ \end{array}$$

In this case, the Guard is 1 and the Round and Sticky bits are zero. This is the “exactly half” case—if the LSB was odd (1) we would add, but since it is even (0) we do nothing.

$$\begin{aligned} -1.0000011100 \times 2^5 &= -100000.11100 \times 2^0 = -32.875 \\ &= -3.2875 \times 10^1 \end{aligned}$$

*** Example** Calculate $A \times (B + C)$ by hand, assuming A, B, and C are stored in the modified 16-bit NVIDIA format described in the text. Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

a.

$$1.666015625 \times 10^0 \times (1.9760 \times 10^4 - 1.9744 \times 10^4)$$

b.

$$3.48 \times 10^2 \times (6.34765625 \times 10^{-2} - 4.052734375 \times 10^{-2})$$

* Solution:

a.

$$1.666015625 \times 10^0 \times (1.9760 \times 10^4 - 1.9744 \times 10^4)$$

$$(A) 1.666015625 \times 10^0 = 1.1010101010 \times 2^0$$

$$(B) 1.9760 \times 10^4 = 1.0011010011 \times 2^{14}$$

$$(C) -1.9744 \times 10^4 = -1.0011010010 \times 2^{14}$$

Exponents match, no shifting necessary

$$(B) 1.0011010011$$

$$(C) -1.0011010010$$

$$(B+C) 0.0000000001 \times 2^{14}$$

$$(B+C) 1.0000000000 \times 2^4$$

$$\text{Exp: } 0 + 4 = 4$$

Signs: both positive, result positive

* Solution:

a.

Mantissa:

$$\begin{array}{r} (A) \qquad \qquad \qquad 1.1010101010 \\ (B+C) \qquad \times 1.0000000000 \\ \hline \end{array}$$

11010101010

1.10101010100000000000

$$A \times (B+C) \quad 1.1010101010 \ 0000000000$$

Guard=0, Round=0, Sticky=0: No Round

$$A (B+C) \ 1.1010101010 \times 2^4$$

* Solution:

b.

$$3.48 \times 10^2 \times (6.34765625 \times 10^{-2} - 4.052734375 \times 10^{-2})$$

$$(A) 3.48 \times 10^2 = 1.0101110000 \times 2^8$$

$$(B) 6.34765625 \times 10^{-2} = 1.0000010000 \times 2^{-4}$$

$$(C) -4.052734375 \times 10^{-2} = 1.0100110000 \times 2^{-5}$$

Shift binary point of smaller left 1 so exponents match

$$(B) \quad 1.0000010000 \times 2^{-4}$$

$$(C) \quad -.1010011000 \ 0 \times 2^{-4}$$

$$(B+C) \quad .0101111000 \text{ Normalize, subtract 2 from exponent}$$

$$(B+C) \quad 1.0111100000 \times 2^{-6}$$

$$\text{Exp: } 8 - 6 = 2$$

Signs: both positive, result positive

* Solution:

b.

Mantissa:

(A)

(B + C)

$$\begin{array}{r} 1.0101110000 \\ \times 1.0111100000 \\ \hline \end{array}$$

$$\begin{array}{r} 10101110000 \\ 10101110000 \\ 10101110000 \\ 10101110000 \\ 10101110000 \\ 10101110000 \end{array}$$

$$A \times (B + C) \quad 1.1111111100 \ 100000000000$$

Guard=1, Round=0, Sticky=0: Round to even

$$A \times (B + C) \quad 1.1111111100 \times 2^2$$

是偶数就舍,是奇数就入

并行性和计算机算法：关联性

✱ if $x + (y + z) = (x + y) + z$.

- $x = -1.5_{\text{ten}} \times 10^{38}$, $y = 1.5_{\text{ten}} \times 10^{38}$, and $z = 1.0$
- $x + (y + z) = 0.0$
- $(x + y) + z = 1.0$ 因为计算机表示浮点数的位数是有限的。

浮点数异常

1.0 不等于 10×0.1 (Why?)

✱ ex. $1.0 \times 10.0 == 10.0$ but, $0.1 \times 10.0 != 1.0$

✱ 十进制的 $0.1 == 1/16 + 1/32 + 1/256 + 1/512 + 1/4096 + \dots ==$

$0.000110011001100110011\dots$

对比十进制的 $1/3$ 是循环小数 $0.333333\dots$

如果小数点后面位数有限, 那么 $3 \times 1/3 != 1$

*** Example $(A \times B) + (A \times C) = A \times (B + C)$?**

a.

$$1.666015625 \times 10^0 \times 1.9760 \times 10^4 - 1.666015625 \times 10^0 \times 1.9744 \times 10^4$$

b.

$$1.666015625 \times 10^0 \times (1.9760 \times 10^4 - 1.9744 \times 10^4)$$

$$\text{b. solution: } A (B+C) = 1.1010101010 \times 2^4 = 26.65625$$

* Solution: a.

Mantissa:

```
(A)      1.1010101010
(C)      × 1.0011010010
          -----
          11010101010
           11010101010
            11010101010
             11010101010
              11010101010
               11010101010
                -----
```

10.0000000111110111010 Normalize, add 1 to exponent

A×C 1.0000000011 11 101110100 Guard=1, Round=1, Sticky=1: Round

A×C -1.0000000100 × 2¹⁵

A×B 1.0000000101 × 2¹⁵

A×C -1.0000000100 × 2¹⁵

A×B+A×C .0000000001 × 2¹⁵

A × B + A × C 1.0000000000 × 2⁵

$$(A \times B) + (A \times C) = 1.0000000000 \times 2^5 = 32$$

MIPS里的浮点数指令FP

* FP 硬件是协处理器

- 扩展 ISA 的辅助处理器

* 独立的 FP 寄存器组

- 32个 单精度: $\$f0, \$f1, \dots \$f31$
- 配对组成双精度浮点数的寄存器: $\$f0/\$f1, \$f2/\$f3, \dots$
 - MIP 第 2 版 ISA 支持 32×64 位 FP reg
 - FP 指令只在浮点数寄存器组操作
- 程序通常不对 FP 数据执行整数操作，反之亦然
- 更多的寄存器对减少代码大小有冲击

* FP 内存取load 和内存写指令

- lwc1, ldc1, swc1, sdc1
 - e.g., ldc1 $\$f8, 32(\$sp)$

FP Instructions in MIPS

* Single-precision arithmetic

- add.s, sub.s, mul.s, div.s
 - e.g., add.s \$f0, \$f1, \$f6

* Double-precision arithmetic

- add.d, sub.d, mul.d, div.d
 - e.g., mul.d \$f4, \$f4, \$f6

* Single- and double-precision comparison

- c.xx.s, c.xx.d (xx is eq, lt, le, ...)
- Sets or clears FP condition-code bit
 - e.g. c.lt.s \$f3, \$f4

* Branch on FP condition code true or false

- bc1t, bc1f
 - e.g., bc1t TargetLabel

FP Example: °F to °C

* C code:

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

- fahr in **\$f12**, result in **\$f0**, literals in global memory space

* Compiled MIPS code:

```
f2c: lwc1  $f16, const5($gp)  
     lwc2  $f18, const9($gp)  
     div.s $f16, $f16, $f18  
     lwc1  $f18, const32($gp)  
     sub.s $f18, $f12, $f18  
     mul.s $f0,  $f16, $f18  
     jr   $ra
```

补充指令知识

- * 编译mips程序时，会将小变量放入sdata, sbss, scommon等段。这个大小由编译选项-G来决定。-G0则表示不使用小数据段。默认是-G8，表示小于或等于8 byte的数据将放入小变量区。

这样的话，在上电时给gp初始化一个值，那么，所有小变量区的变量就可以通过

lw r1, offset(\$gp)来访问了。

- * load immediate:

```
li register_destination, value
```
- * #load immediate value into destination register
- * 顾名思义，这里的 li 意为 load immediate

FP Example: Array Multiplication

* $X = X + Y \times Z$

- All 32×32 matrices, 64-bit double-precision elements

* C code:

```
void mm (double x[],  
         double y[], double z[]) {  
    int i, j, k;  
    for (i = 0; i != 32; i = i + 1)  
        for (j = 0; j != 32; j = j + 1)  
            for (k = 0; k != 32; k = k + 1)  
                x[i][j] = x[i][j]  
                    + y[i][k] * z[k][j];
```

- Addresses of x, y, z in \$a0, \$a1, \$a2, and
i, j, k in \$s0, \$s1, \$s2

FP Example: Array Multiplication

■ MIPS code:

```
li $t1, 32    # $t1 = 32 (row size/loop end)
li $s0, 0     # i = 0; initialize 1st for loop
L1: li $s1, 0  # j = 0; restart 2nd for loop
L2: li $s2, 0  # k = 0; restart 3rd for loop
    sll $t2, $s0, 5 # $t2 = i * 32 (size of row of x)
    addu $t2, $t2, $s1 # $t2 = i * size(row) + j
    sll $t2, $t2, 3 # $t2 = byte offset of [i][j]
    addu $t2, $a0, $t2 # $t2 = byte address of x[i][j]
    l.d $f4, 0($t2) # $f4 = 8 bytes of x[i][j]
L3: sll $t0, $s2, 5 # $t0 = k * 32 (size of row of z)
    addu $t0, $t0, $s1 # $t0 = k * size(row) + j
    sll $t0, $t0, 3 # $t0 = byte offset of [k][j]
    addu $t0, $a2, $t0 # $t0 = byte address of z[k][j]
    l.d $f16, 0($t0) # $f16 = 8 bytes of z[k][j]
...
```

FP Example: Array Multiplication

...

```
sll $t0, $s0, 5    # $t0 = i*32 (size of row of y)
addu $t0, $t0, $s2  # $t0 = i*size(row) + k
sll $t0, $t0, 3     # $t0 = byte offset of [i][k]
addu $t0, $a1, $t0  # $t0 = byte address of y[i][k]
l.d $f18, 0($t0)    # $f18 = 8 bytes of y[i][k]
```

```
mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
add.d $f4, $f4, $f16   # f4=x[i][j] + y[i][k]*z[k][j]
```

```
addiu $s2, $s2, 1     # $k = k + 1
bne $s2, $t1, L3      # if (k != 32) go to L3
s.d $f4, 0($t2)       # x[i][j] = $f4
```

```
addiu $s1, $s1, 1     # $j = j + 1
bne $s1, $t1, L2      # if (j != 32) go to L2
```

```
addiu $s0, $s0, 1     # $i = i + 1
bne $s0, $t1, L1      # if (i != 32) go to L1
```

3.7 Real Stuff: Floating Point in the x86

- * the x86 floating-point operands on the stack are 80 bits wide
- * The Intel Streaming SIMD Extension 2 (SSE2) Floating-Point Architecture
- * SSE 4.2 in core 2

总结

- ✱ 数字本身并没有内在含义：操作确定它们是 ASCII 字符，还是整数、浮点数。
- ✱ 除法和乘法可以使用相同的硬件：在 MIPS 处理器中的 Hi & Lo 寄存器
- ✱ 浮点数的处理基本上沿用科学计数法的手算方法，对尾数的大小的处理使用乘法和除法的整数算法

MIPS R2000 Organization

