第四章　乘法器和除法器

1.乘法的运算过程

2.乘法器的实现

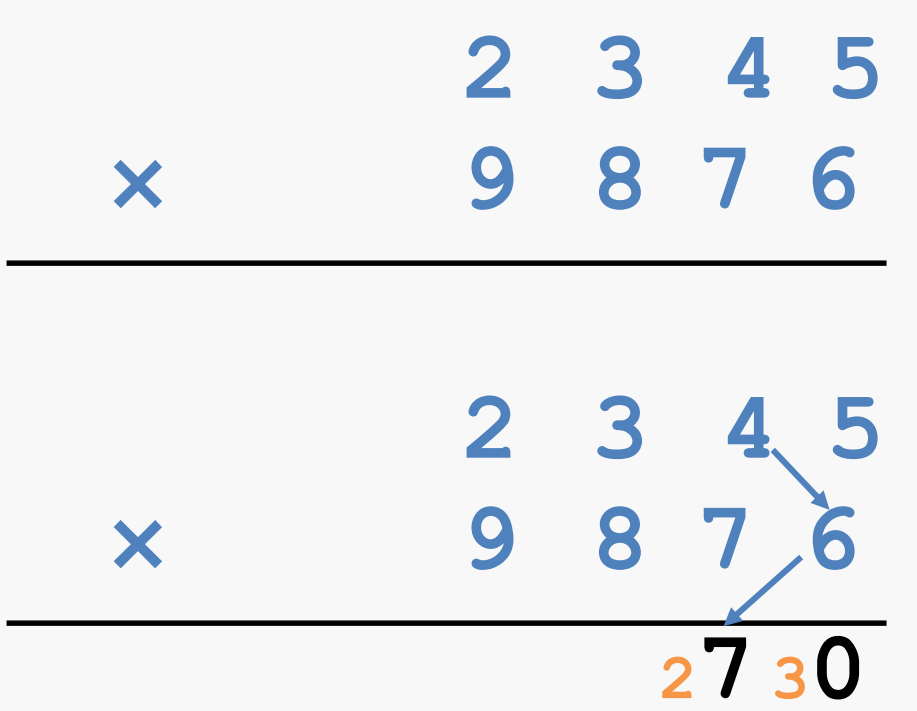
3.乘法器的优化

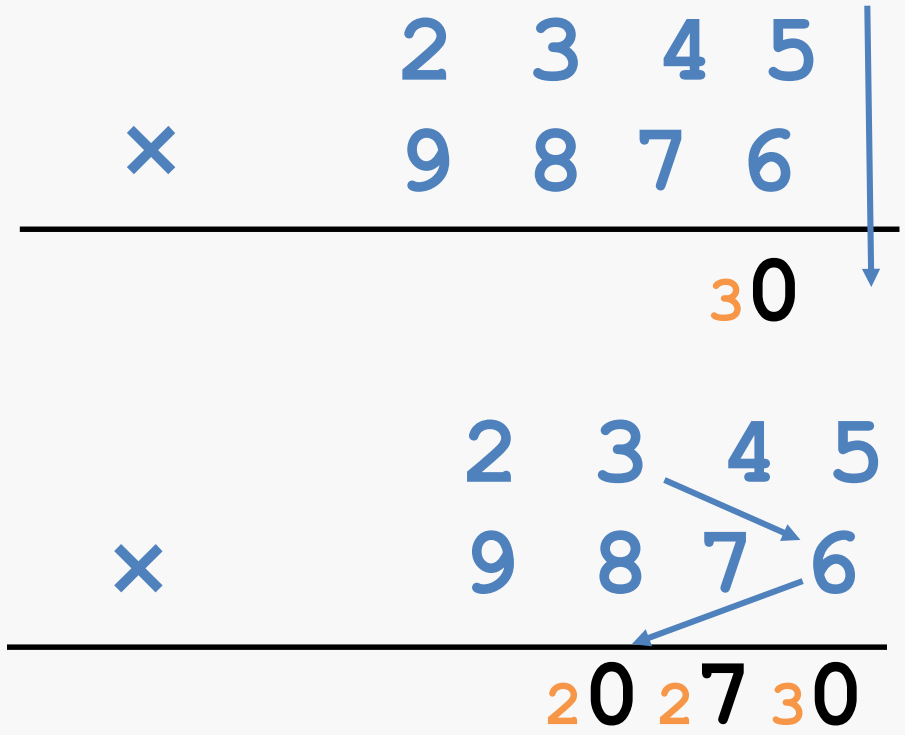
4.除法的运算过程

5.除法器的实现

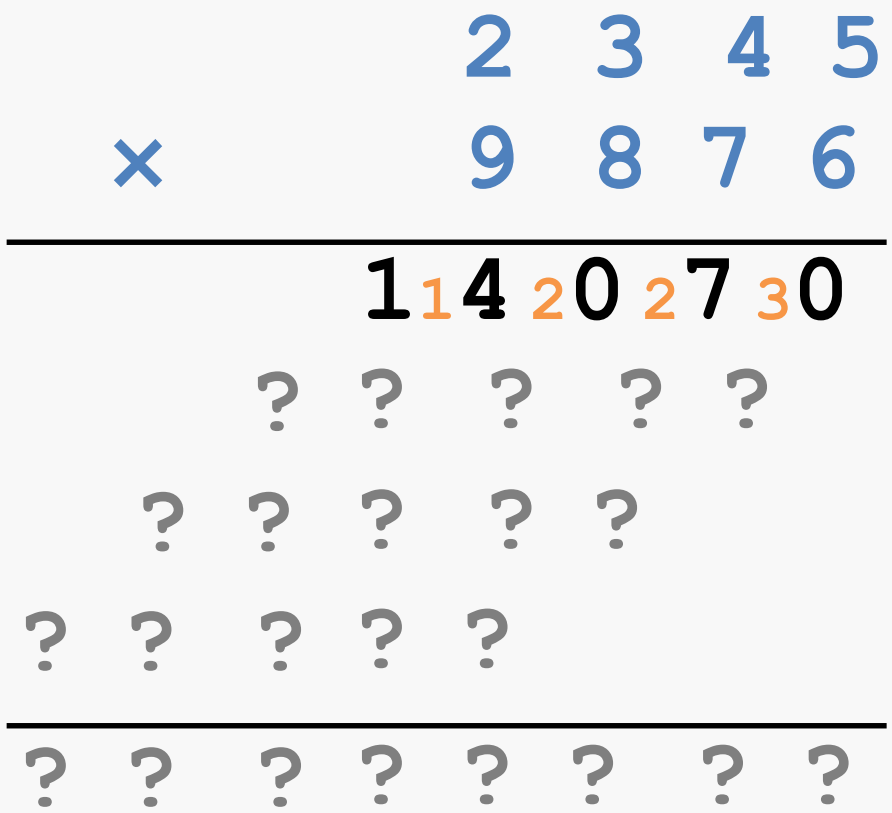
6.除法器的优化

手工进行乘法运算

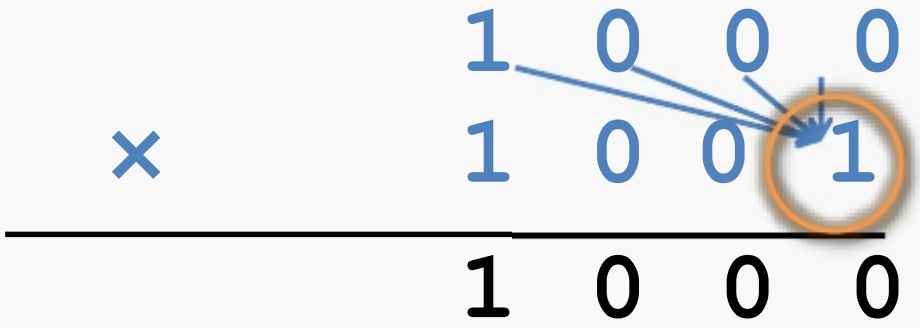

2 3 4 5
× 9 8 7 6
2 3 4 5
× 9 8 7 6
2 7 3 0
2 3 4 5
× 9 8 7 6
1 1 4 2 0 2 7 3 0


2 3 4 5
× 9 8 7 6
3 0
2 3 4 5
× 9 8 7 6
2 0 2 7 3 0

手工进行乘法运算

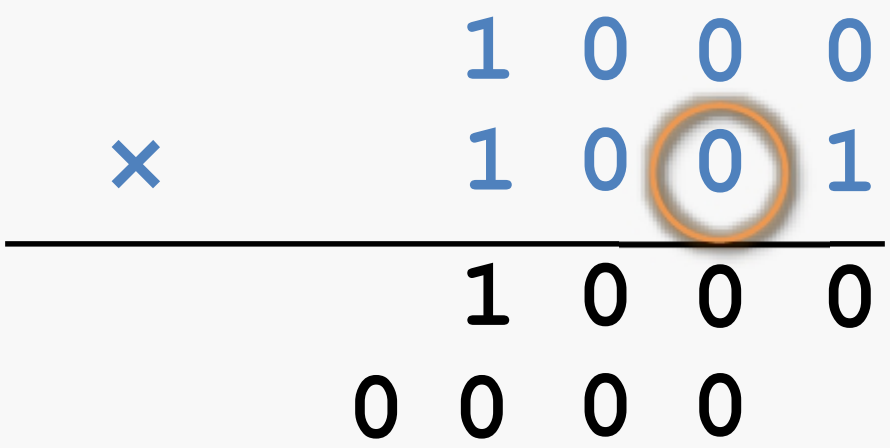


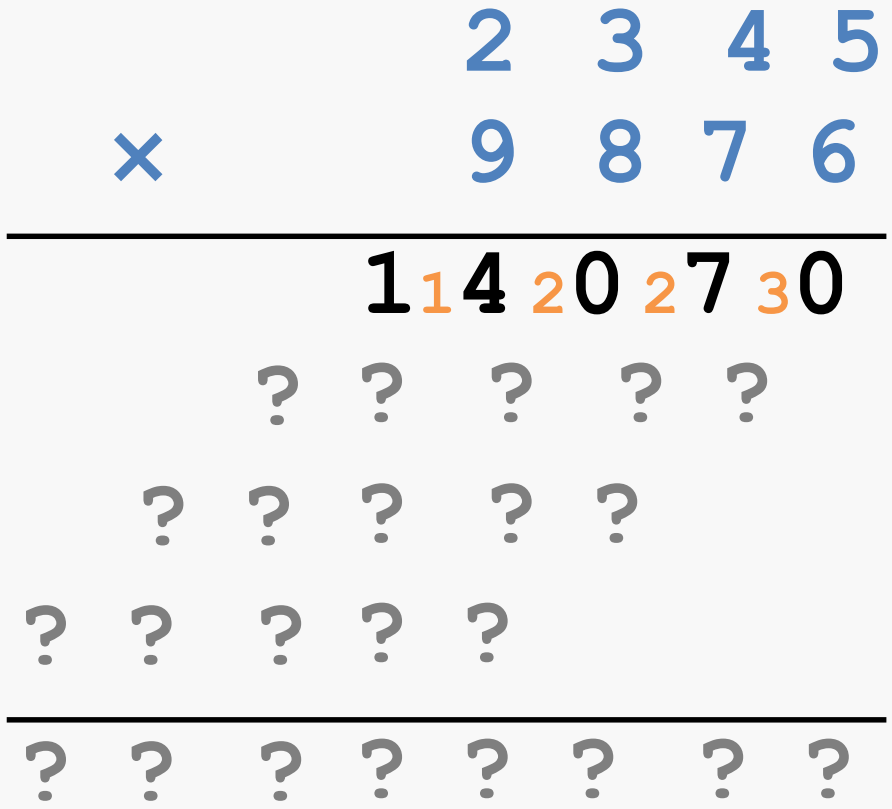
较为简单的数字



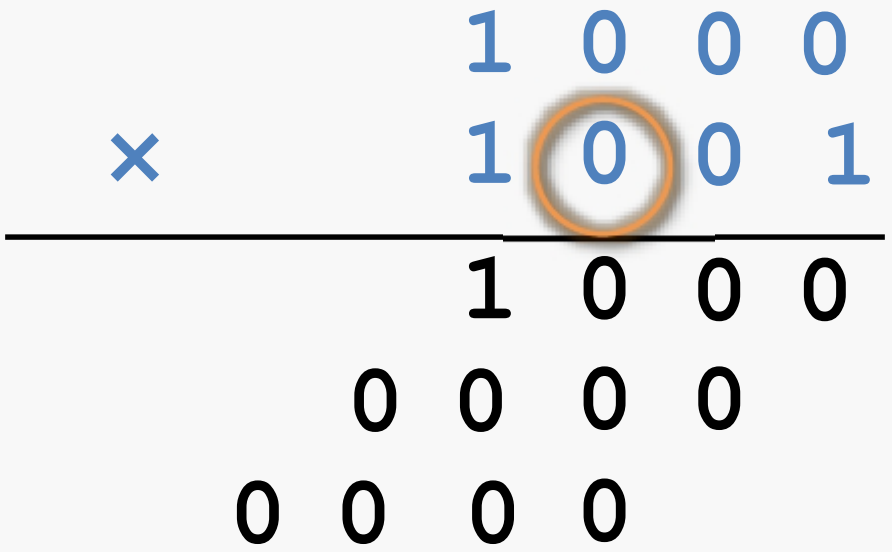
$$
\begin{array}{r}
2\ 3\ 4\ 5 \\
\times\quad 9\ 8\ 7\ 6 \\
\hline
1_1 4_2 0_2 7_3 0 \\
?\ ?\ ?\ ?\ ? \\
\ ?\ ?\ ?\ ?\ ? \\
?\ ?\ ?\ ?\ ? \\
\hline
?\ ?\ ?\ ?\ ?\ ?\ ?\ ?
\end{array}
$$

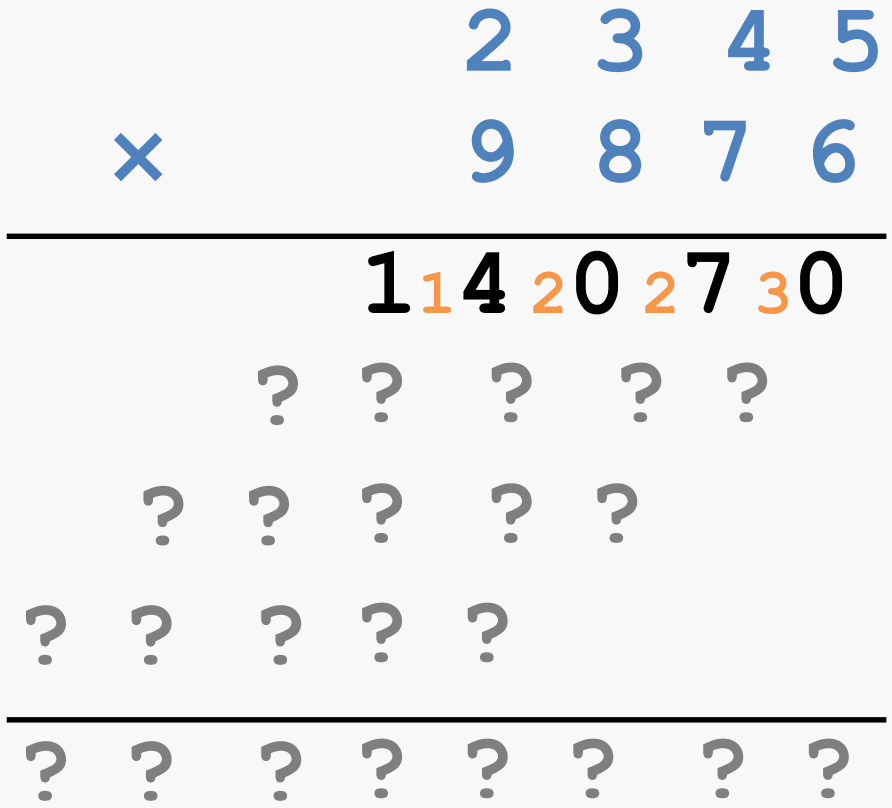
较为简单的数字



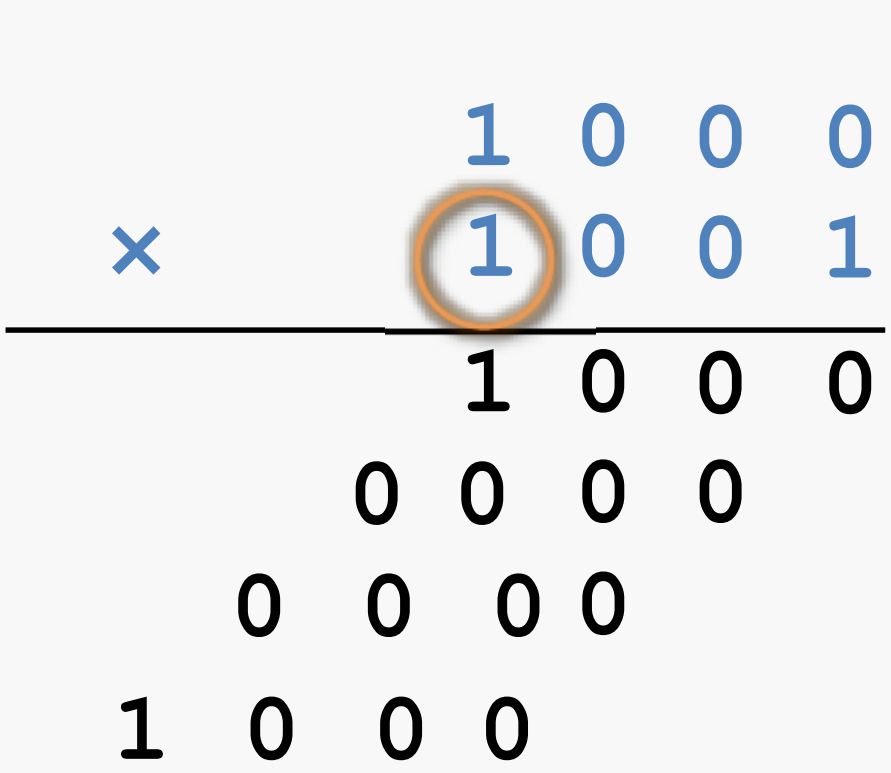


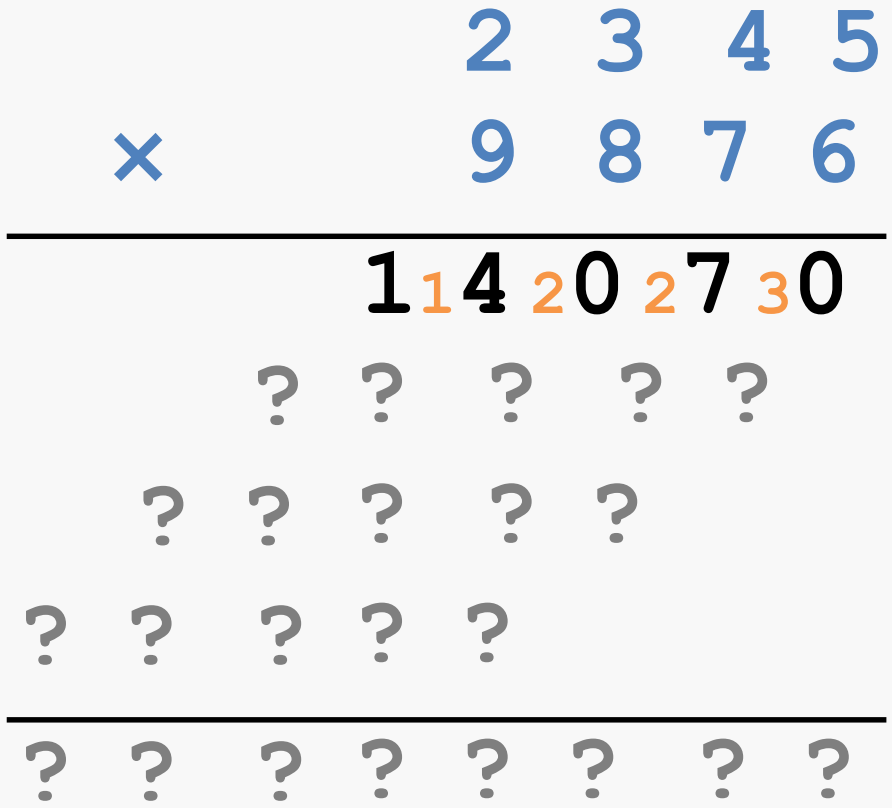
较为简单的数字



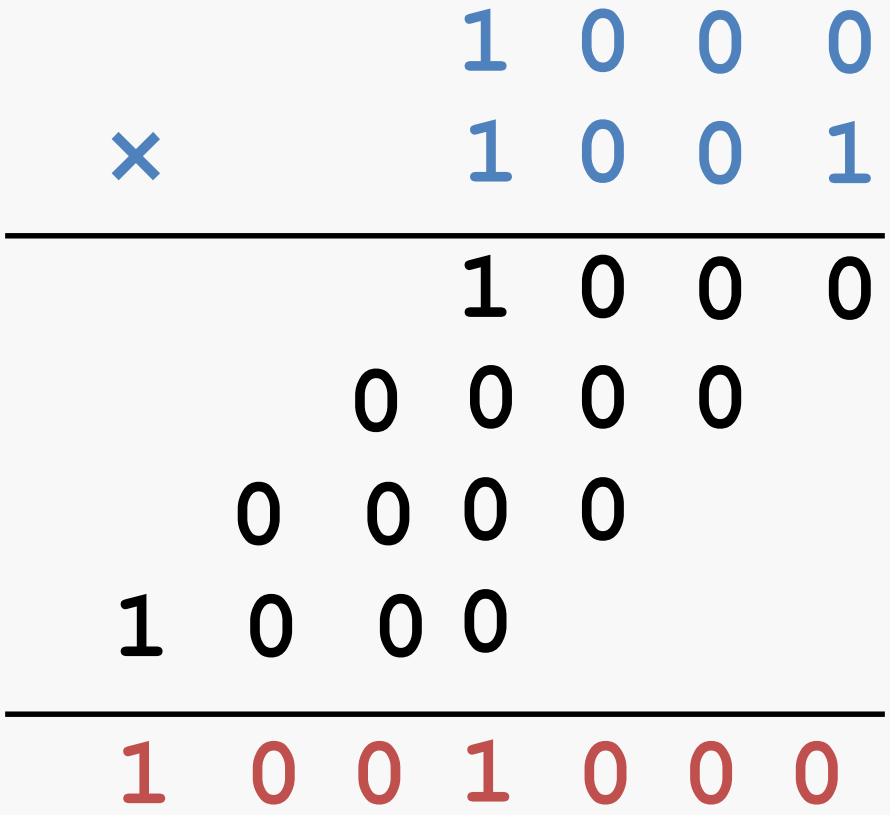


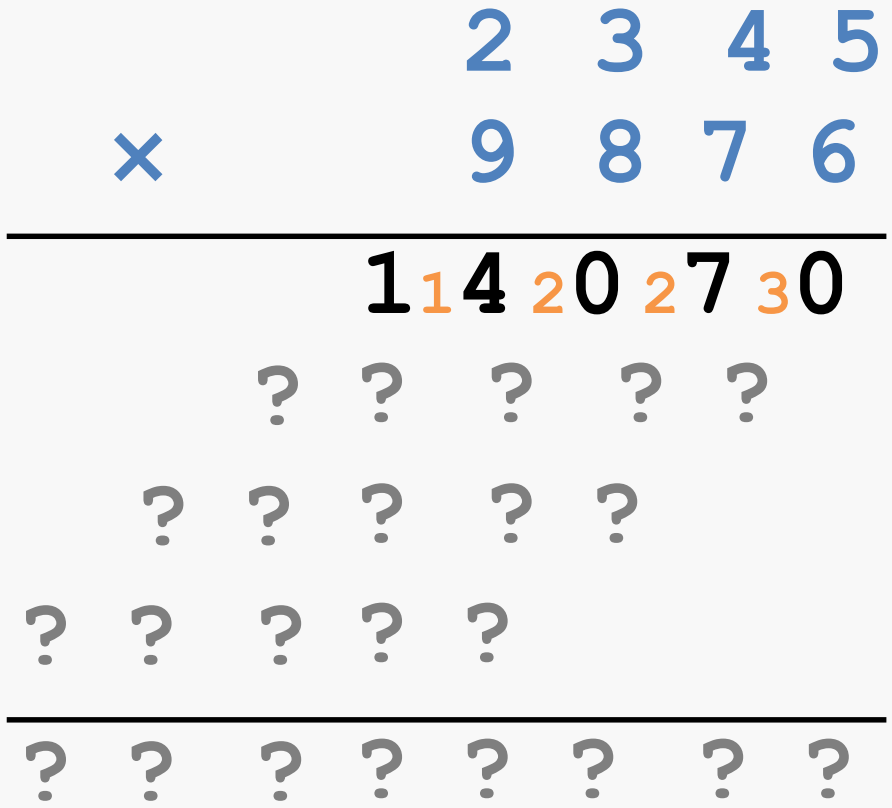
较为简单的数字



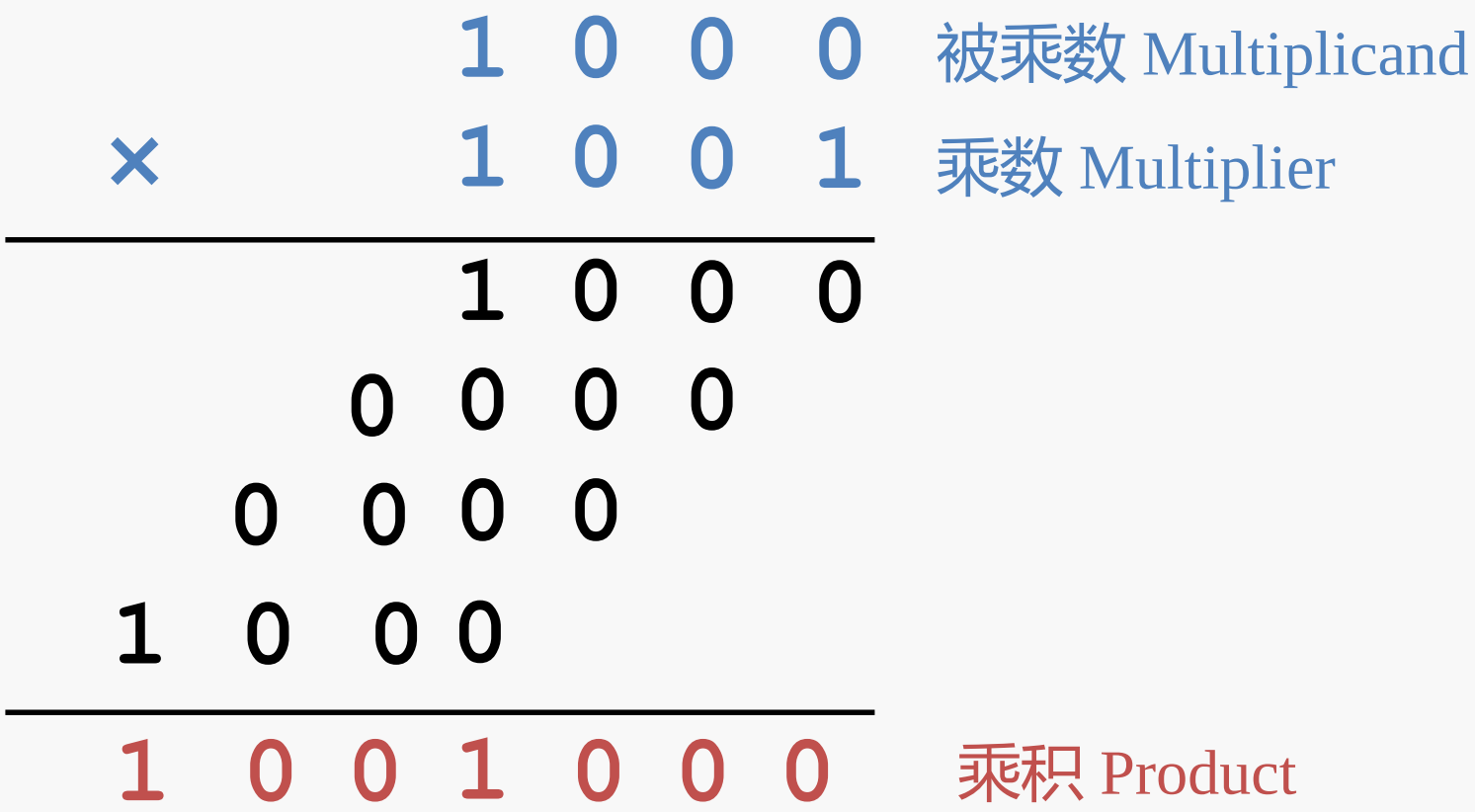


较为简单的数字





简化后的运算过程

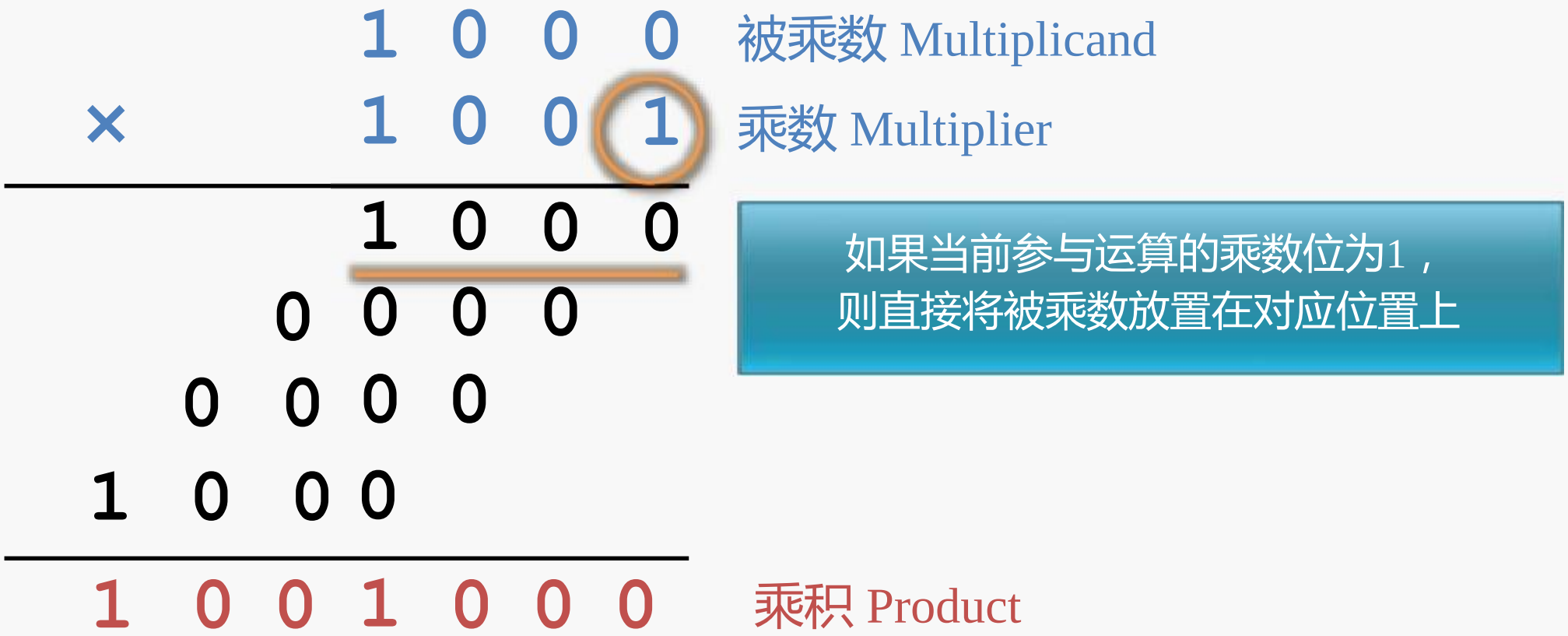


被乘数 Multiplicand

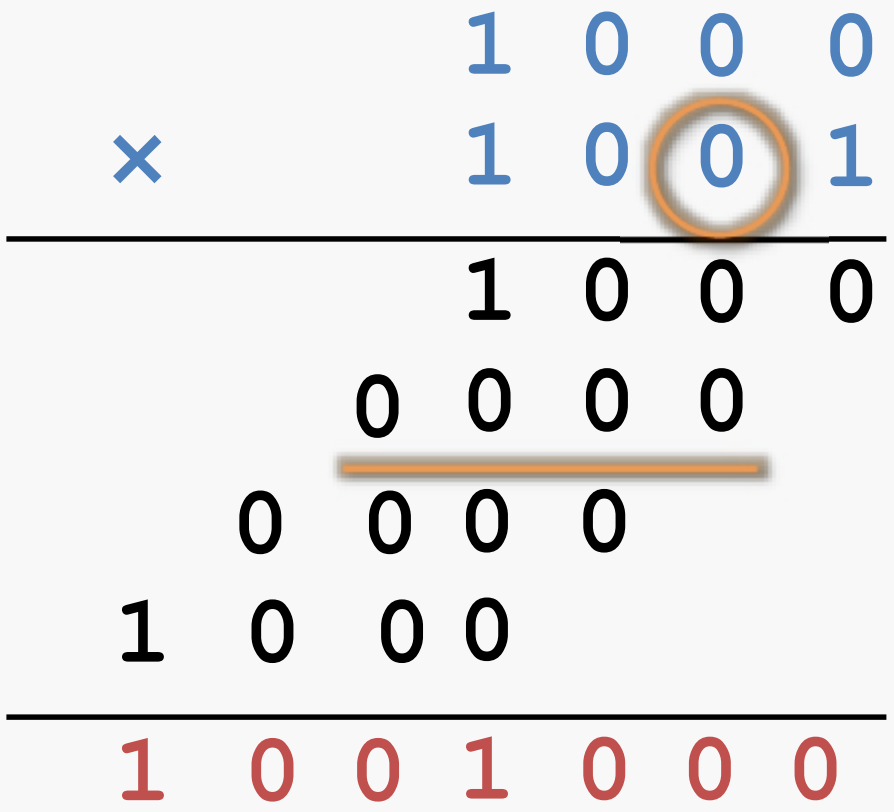
乘数 Multiplier

乘积 Product

简化后的运算过程


1 0 0 0
被乘数 Multiplicand
× 1 0 0 1
乘数 Multiplier
1 0 0 0
0 0 0 0
0 0 0 0
1 0 0 0
1 0 0 1 0 0 0
乘积 Product
如果当前参与运算的乘数位为1，
则直接将被乘数放置在对应位置上

简化后的运算过程



被乘数 Multiplicand

乘数 Multiplier

如果当前参与运算的乘数位为1，
则直接将被乘数放置在对应位置上

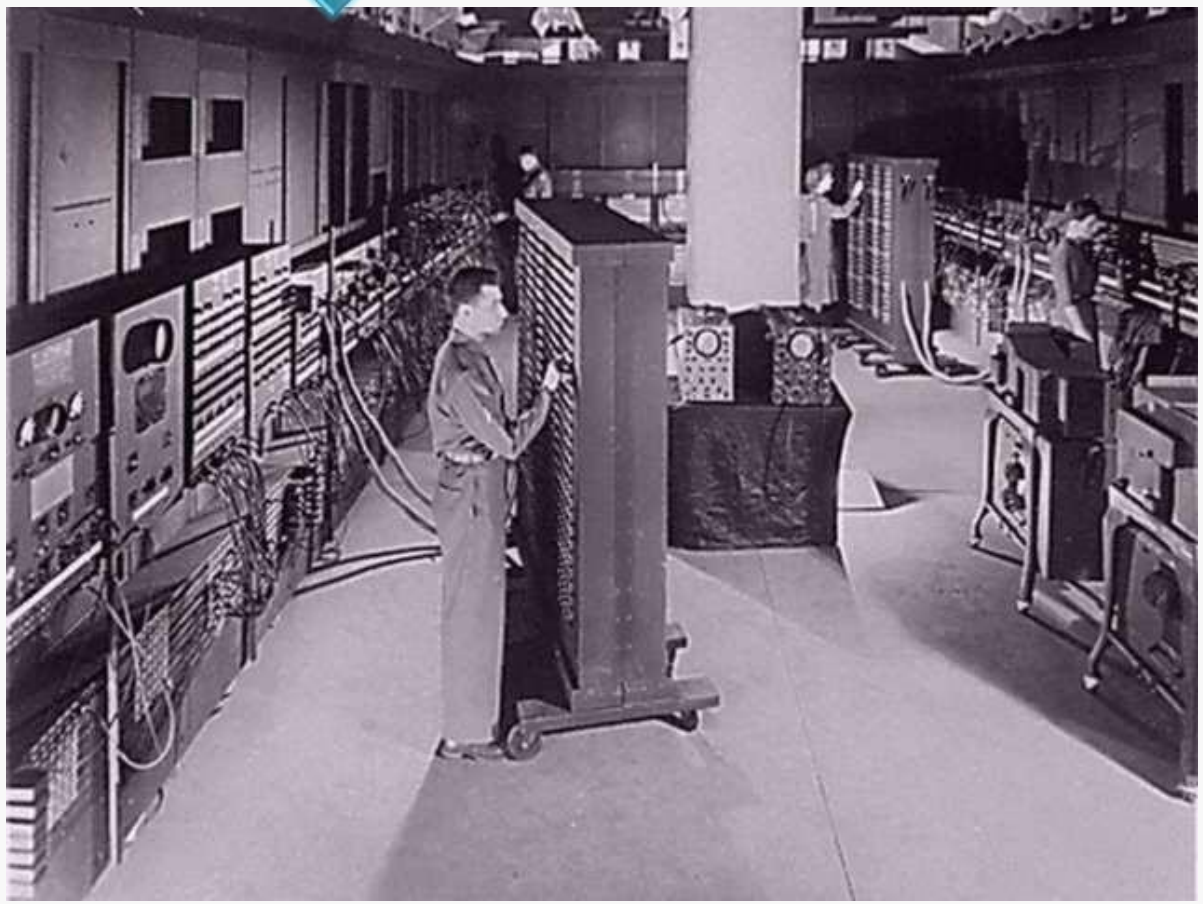
如果当前参与运算的乘数位为0，
则直接将“0”放置在对应位置上

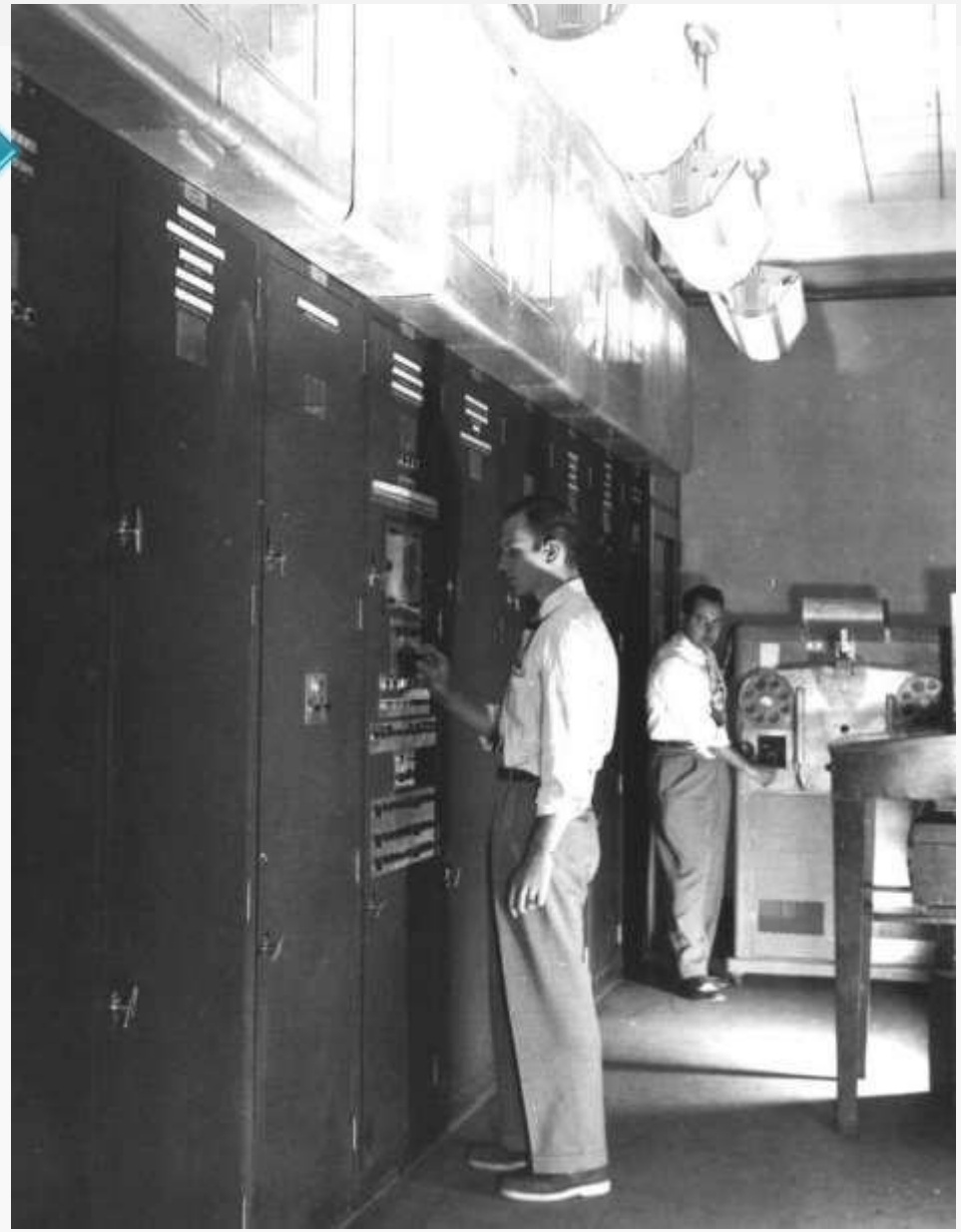
乘积 Product

十进制和二进制运算的选择

采用十进制的ENIAC

采用二进制的EDVAC





十进制和二进制运算的选择

关于EDVAC的
报告草案
1945

电子管是一种“全或无”设备（all-or-none），适合表示只有两个数值的系统，即二进制。

二进制可以大幅度地简化乘法和除法的运算过程。尤其是对于乘法，不再需要十进制乘法表，也不再需要两轮的加法。

必须要记住，十进制才是适合人使用的。因此，输入输出设备需要承担二进制和十进制之间的转换工作。

约翰·冯·诺依曼
John Von Neumann
1903~1957

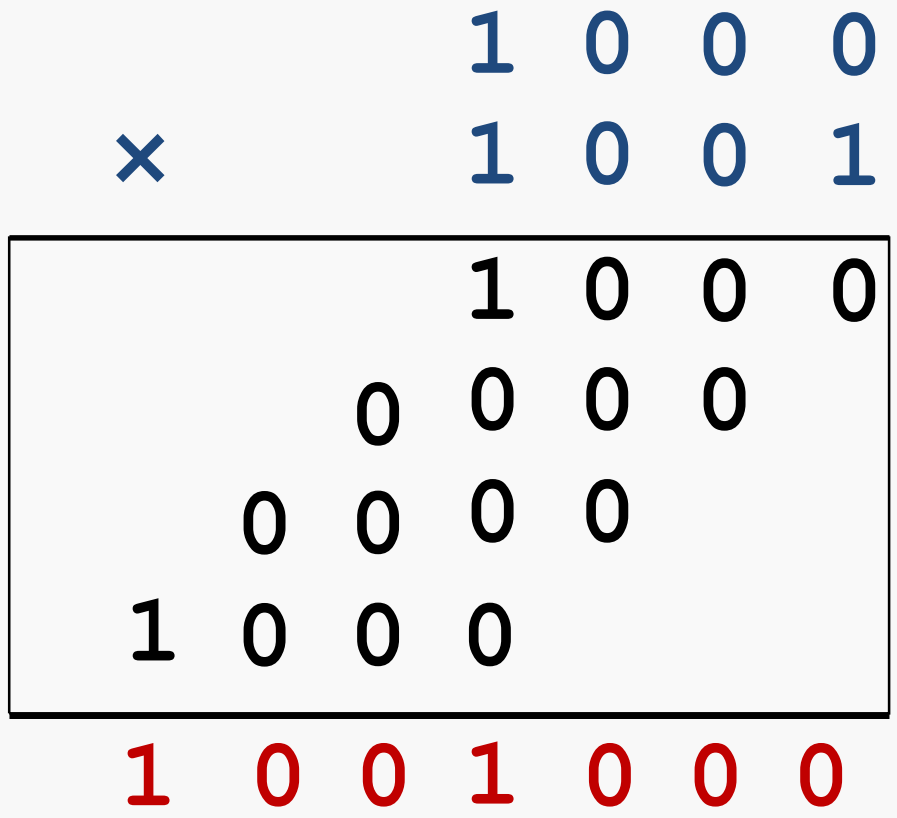
二进制乘法的运算过程


1 0 0 0 被乘数 Multiplicand
× 1 0 0 1 乘数 Multiplier
1 0 0 0
0 0 0 0
0 0 0 0
1 0 0 0
1 0 0 1 0 0 0 乘积 Product
如何面向硬件调整运算过程？

运算过程的进一步调整


1 0 0 0
被乘数 Multiplicand
× 1 0 0 1
乘数 Multiplier
1 0 0 0
0 0 0 0
0 0 0 0
1 0 0 0
1 0 0 1 0 0 0
乘积 Product

运算过程的进一步调整



被乘数 Multiplicand

乘数 Multiplier

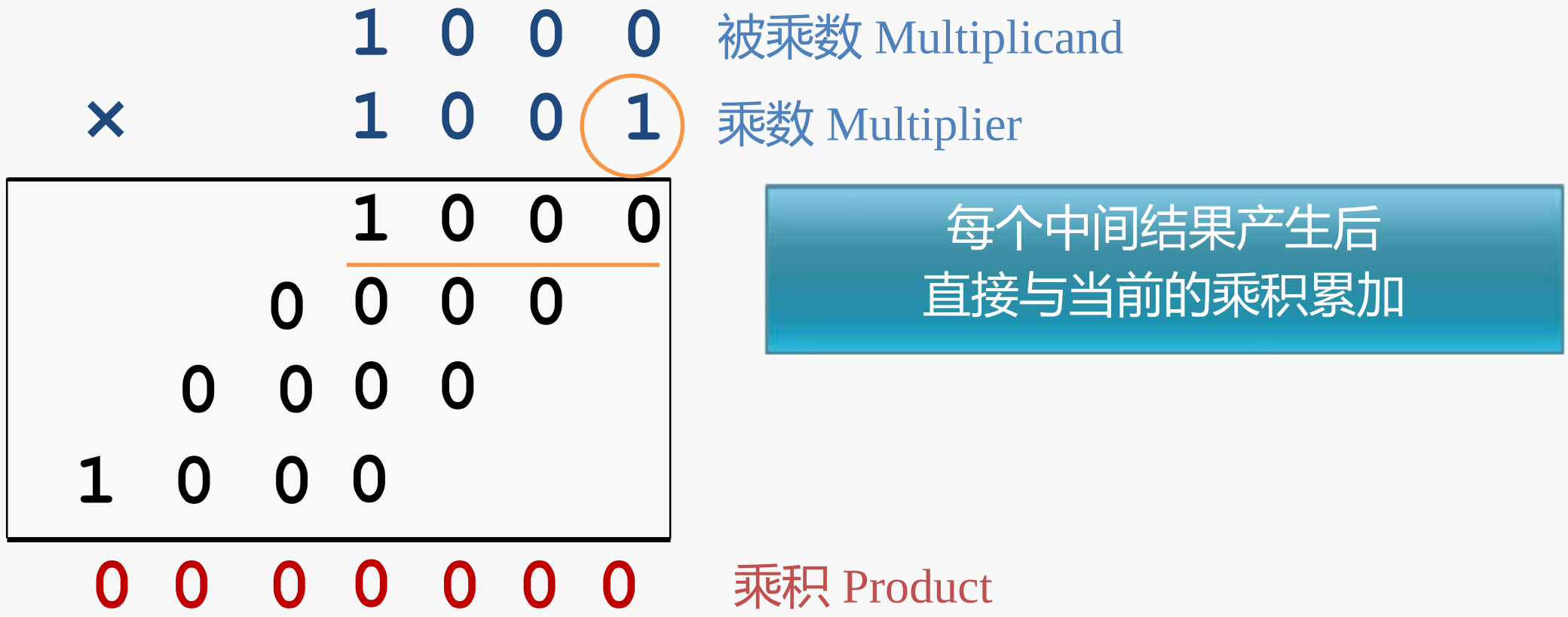
运算开始时，乘积记为“0”

乘积 Product

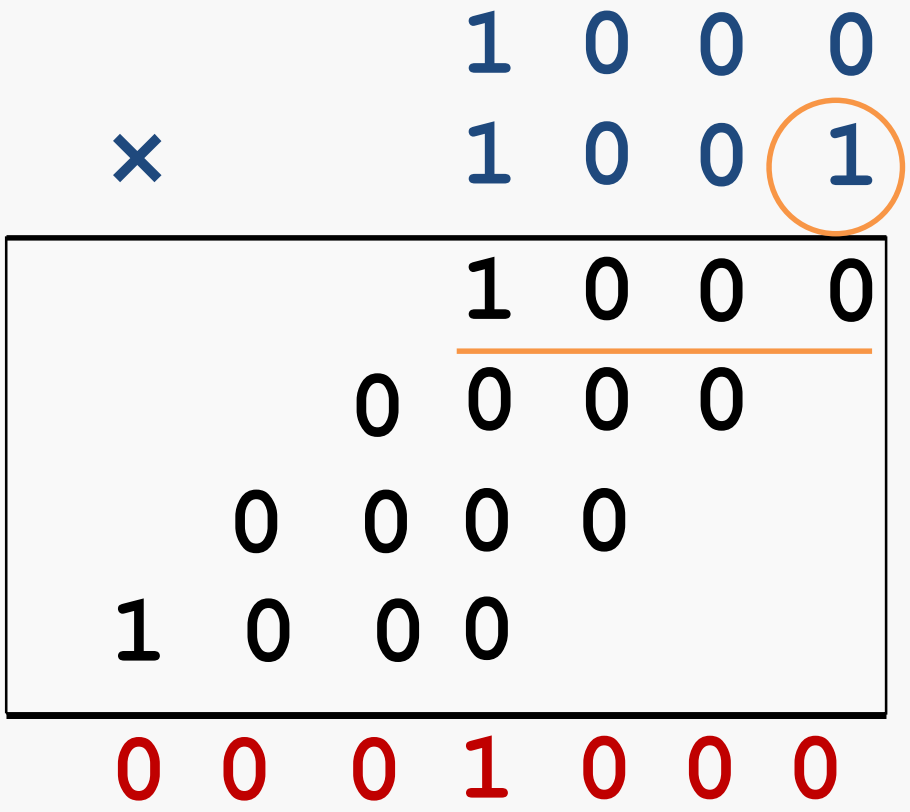
运算过程的进一步调整


1 0 0 0
被乘数 Multiplicand
× 1 0 0 1
乘数 Multiplier
1 0 0 0
0 0 0 0
0 0 0 0
1 0 0 0
0 0 0 0 0 0 0 0
乘积 Product
运算开始时，乘积记为“0”

运算过程的进一步调整


1 0 0 0
被乘数 Multiplicand
× 1 0 0 1
乘数 Multiplier
1 0 0 0
0 0 0 0
0 0 0 0
1 0 0 0
0 0 0 0 0 0 0 0
乘积 Product
每个中间结果产生后
直接与当前的乘积累加

运算过程的进一步调整



被乘数 Multiplicand

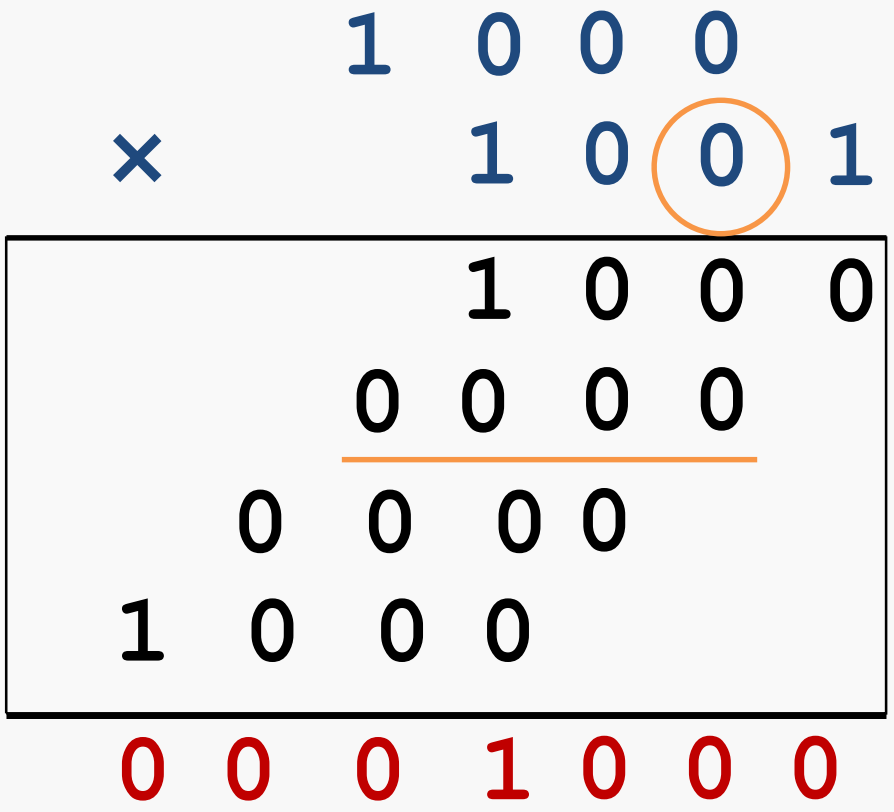
乘数 Multiplier

每个中间结果产生后
直接与当前的乘积累加

每产生一个中间结果
被乘数向左移动一位

乘积 Product

运算过程的进一步调整



被乘数 Multiplicand

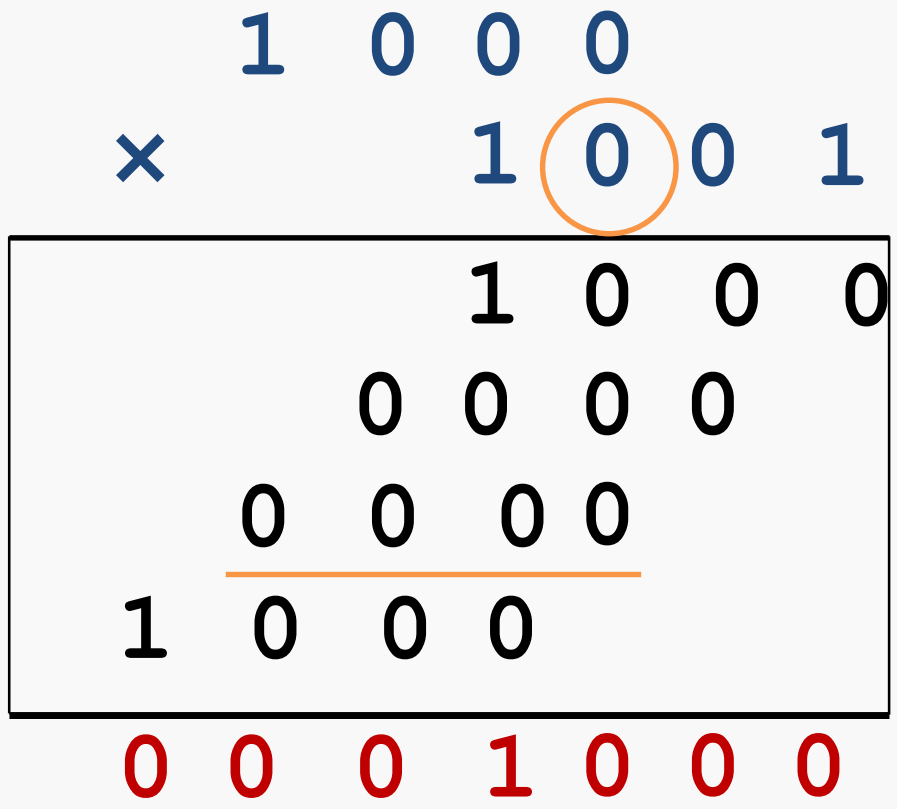
乘数 Multiplier

每个中间结果产生后
直接与当前的乘积累加

每产生一个中间结果
被乘数向左移动一位

乘积 Product

运算过程的进一步调整



被乘数 Multiplicand

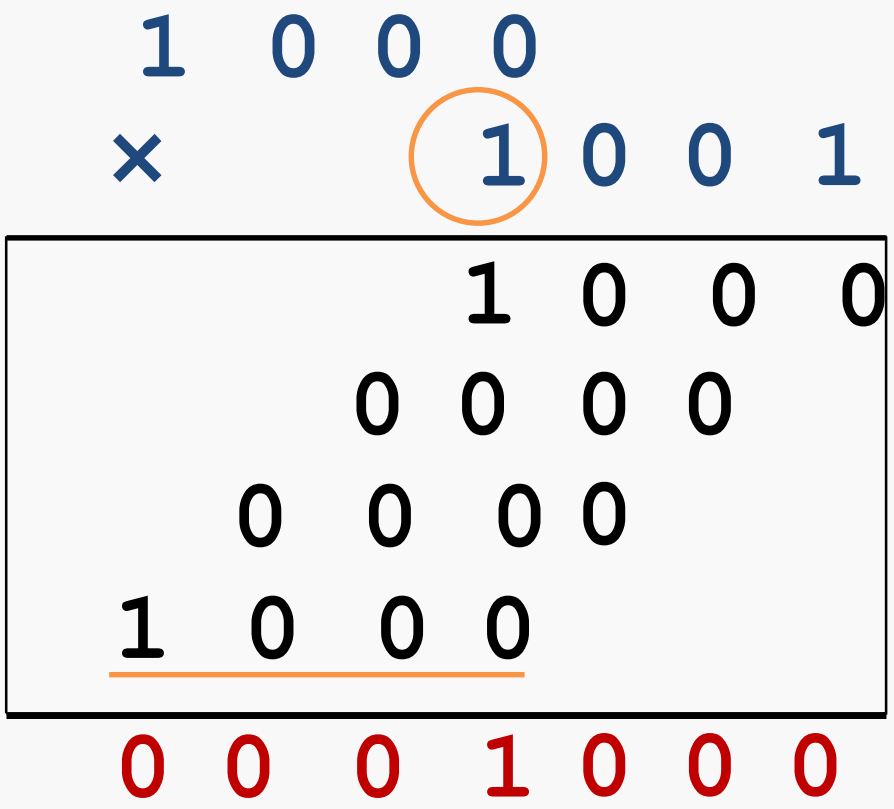
乘数 Multiplier

每个中间结果产生后
直接与当前的乘积累加

每产生一个中间结果
被乘数向左移动一位

乘积 Product

运算过程的进一步调整



被乘数 Multiplicand

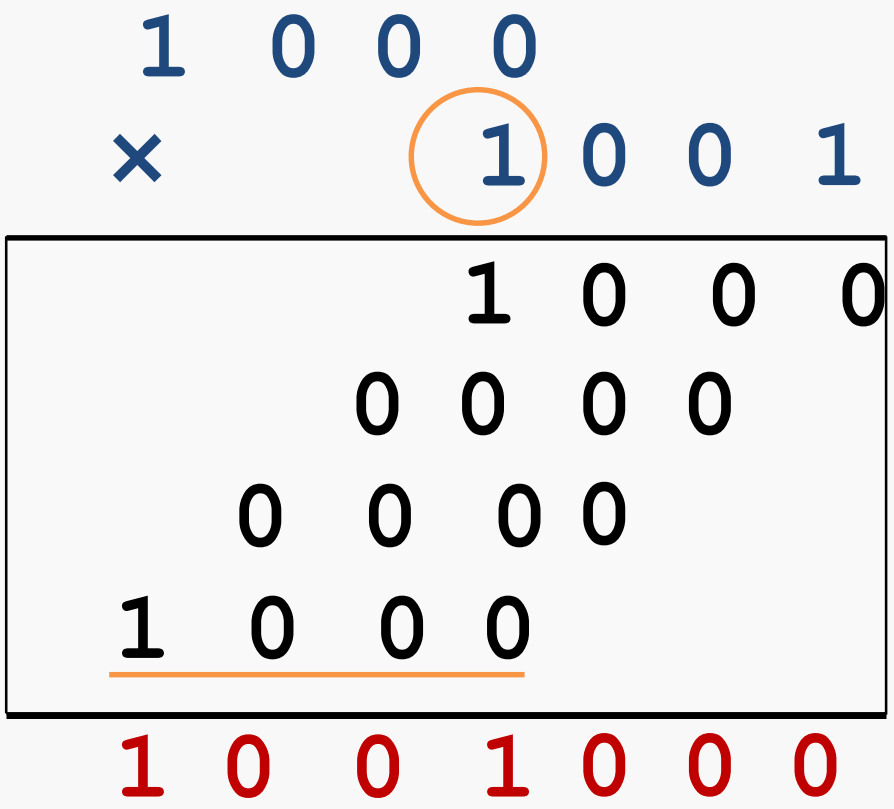
乘数 Multiplier

每个中间结果产生后
直接与当前的乘积累加

每产生一个中间结果
被乘数向左移动一位

乘积 Product

运算过程的进一步调整



被乘数 Multiplicand

乘数 Multiplier

每个中间结果产生后
直接与当前的乘积累加

每产生一个中间结果
被乘数向左移动一位

乘积 Product

运算过程的进一步调整

```
  1 0 0 0          被乘数 Multiplicand
× 1 0 0 1          乘数 Multiplier
```

```
1 0 0 1 0 0 0      乘积 Product
```

适合硬件实现的运算过程！

第四章　乘法器和除法器

1.乘法的运算过程

2.乘法器的实现

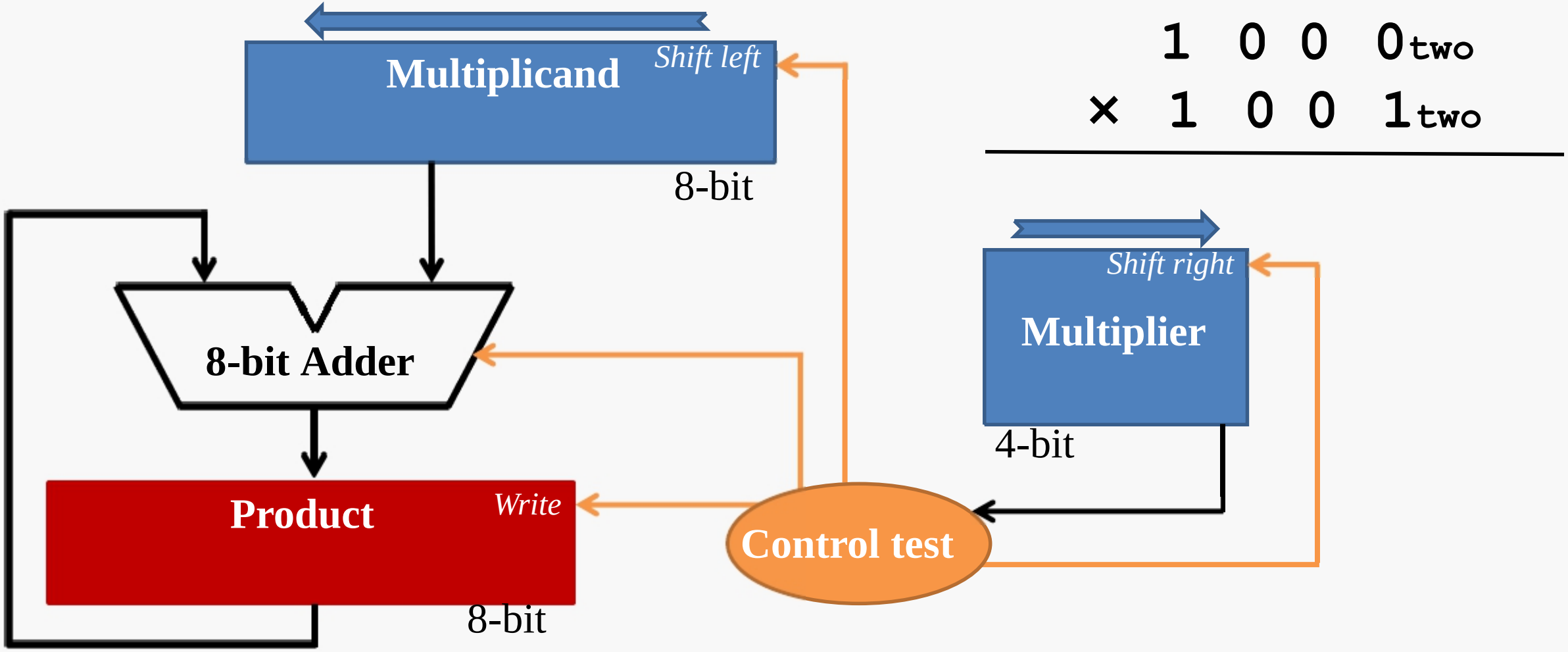
3.乘法器的优化

4.除法的运算过程

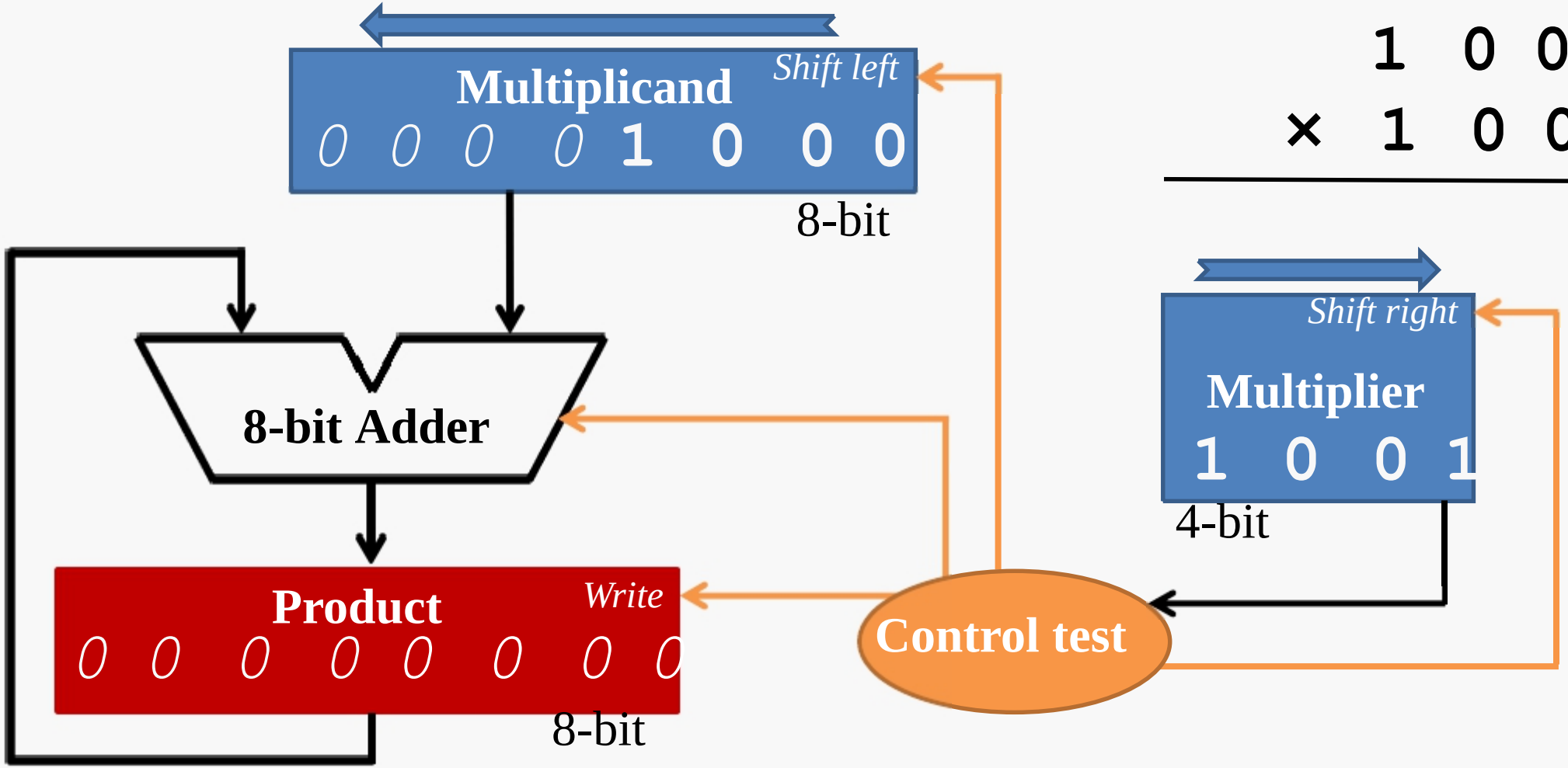
5.除法器的实现

6.除法器的优化

乘法器的实现结构

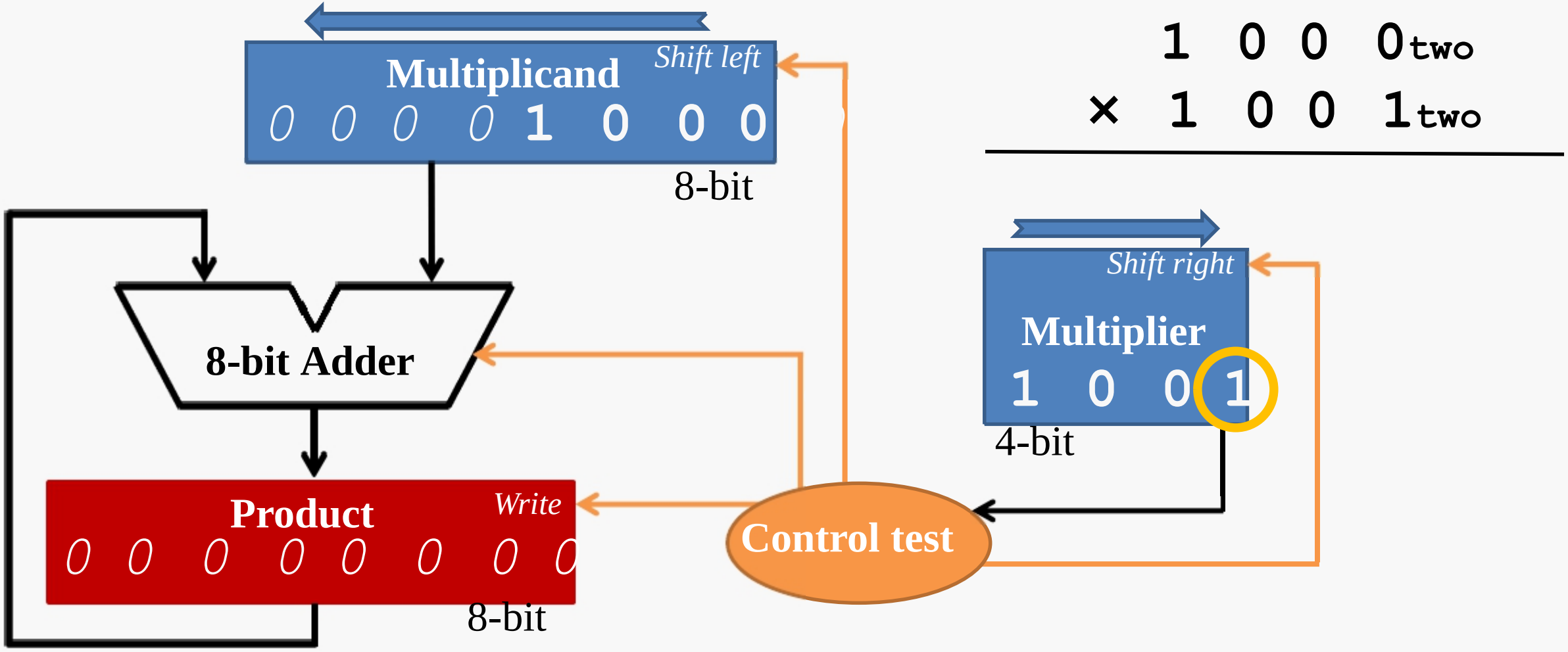

Multiplicand
Shift left
8-bit
8-bit Adder
Product
Write
8-bit
Control test
Shift right
Multiplier
4-bit
1 0 0 0two
× 1 0 0 1two

乘法器的工作过程（初始化）

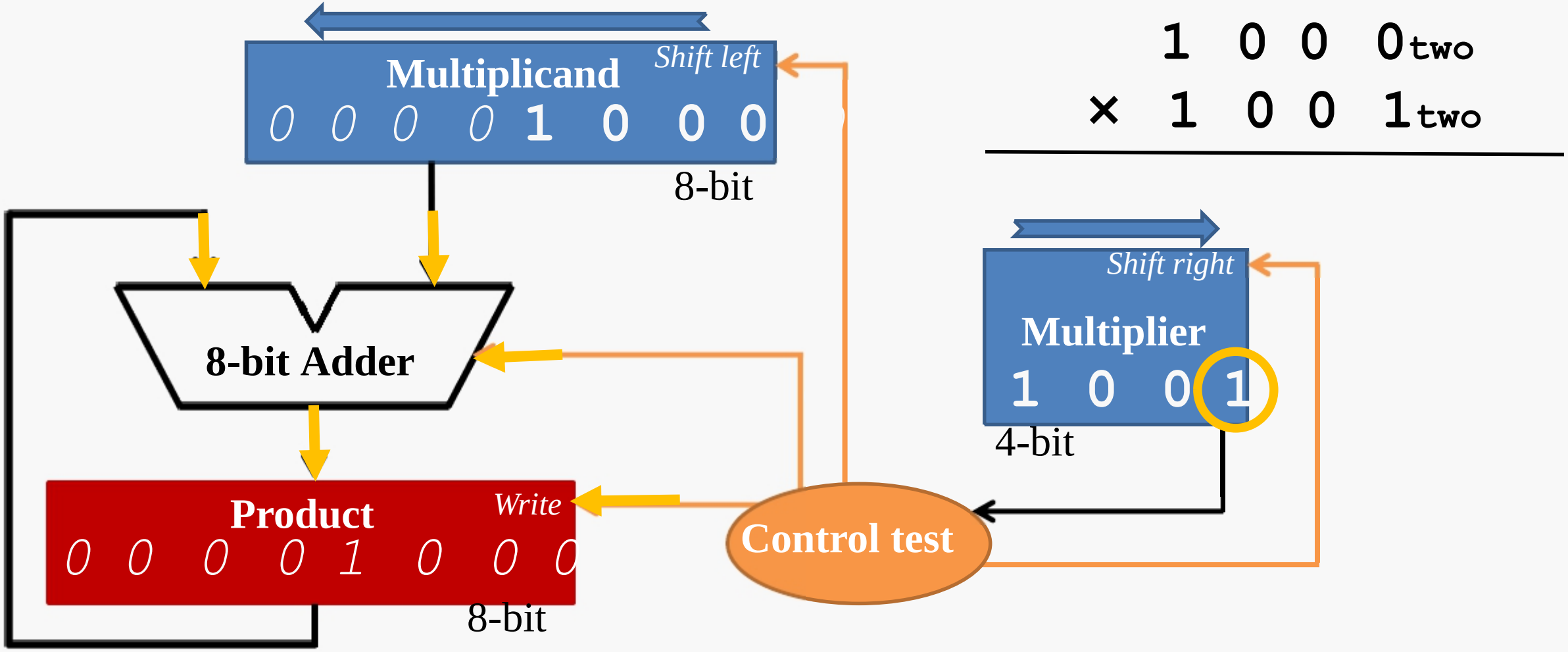


$$\begin{array}{r} 1\ 0\ 0\ 0_{two} \\ \times\ 1\ 0\ 0\ 1_{two} \\ \hline \end{array}$$

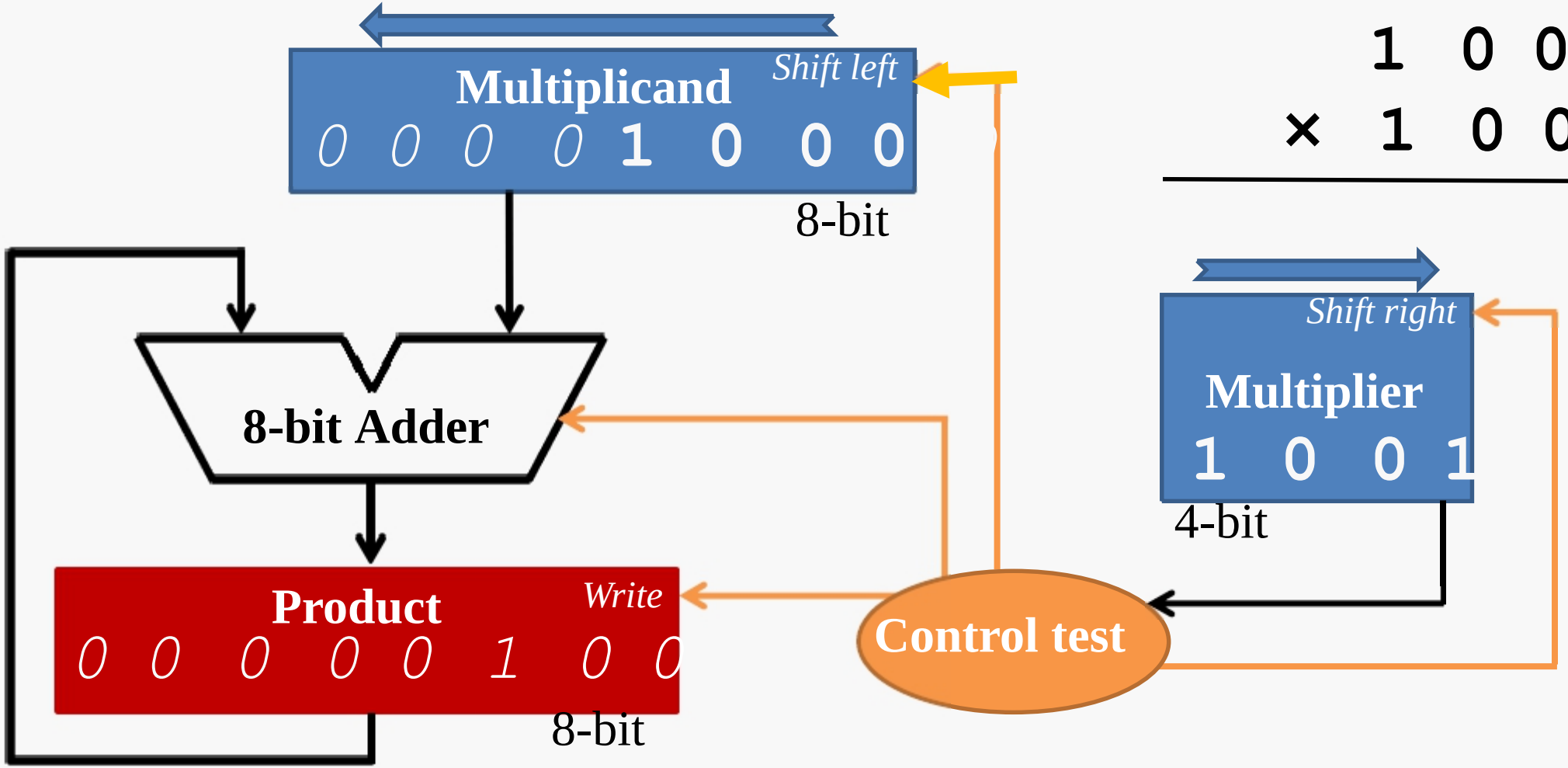
乘法器的工作过程（1）


Multiplicand
Shift left
0 0 0 0 1 0 0 0
8-bit
8-bit Adder
Product
Write
0 0 0 0 0 0 0 0
8-bit
Control test
Shift right
Multiplier
1 0 0 1
4-bit
1 0 0 0two
× 1 0 0 1two

乘法器的工作过程（1a）

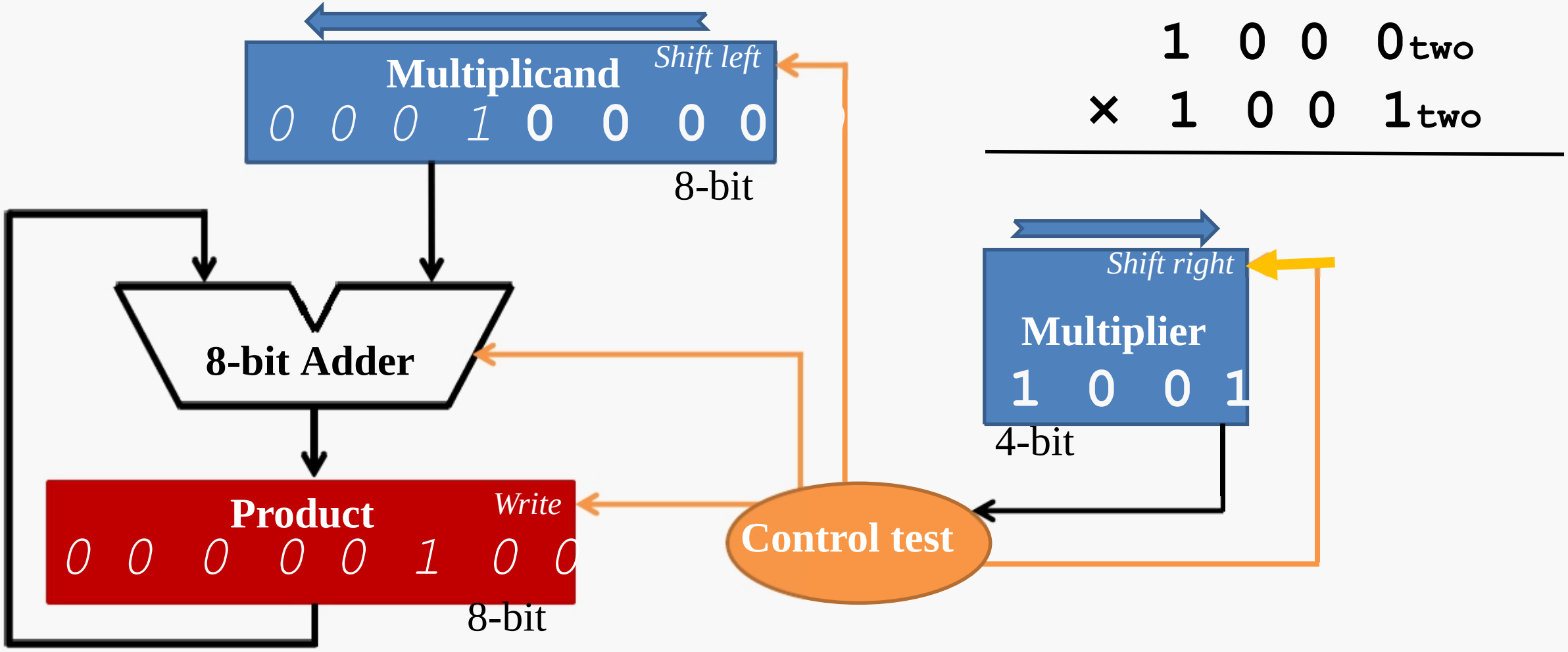

Multiplicand
Shift left
0 0 0 0 1 0 0 0
8-bit
8-bit Adder
Product
Write
0 0 0 0 1 0 0 0
8-bit
Control test
Shift right
Multiplier
1 0 0 1
4-bit
1 0 0 0two
× 1 0 0 1two

乘法器的工作过程（2）

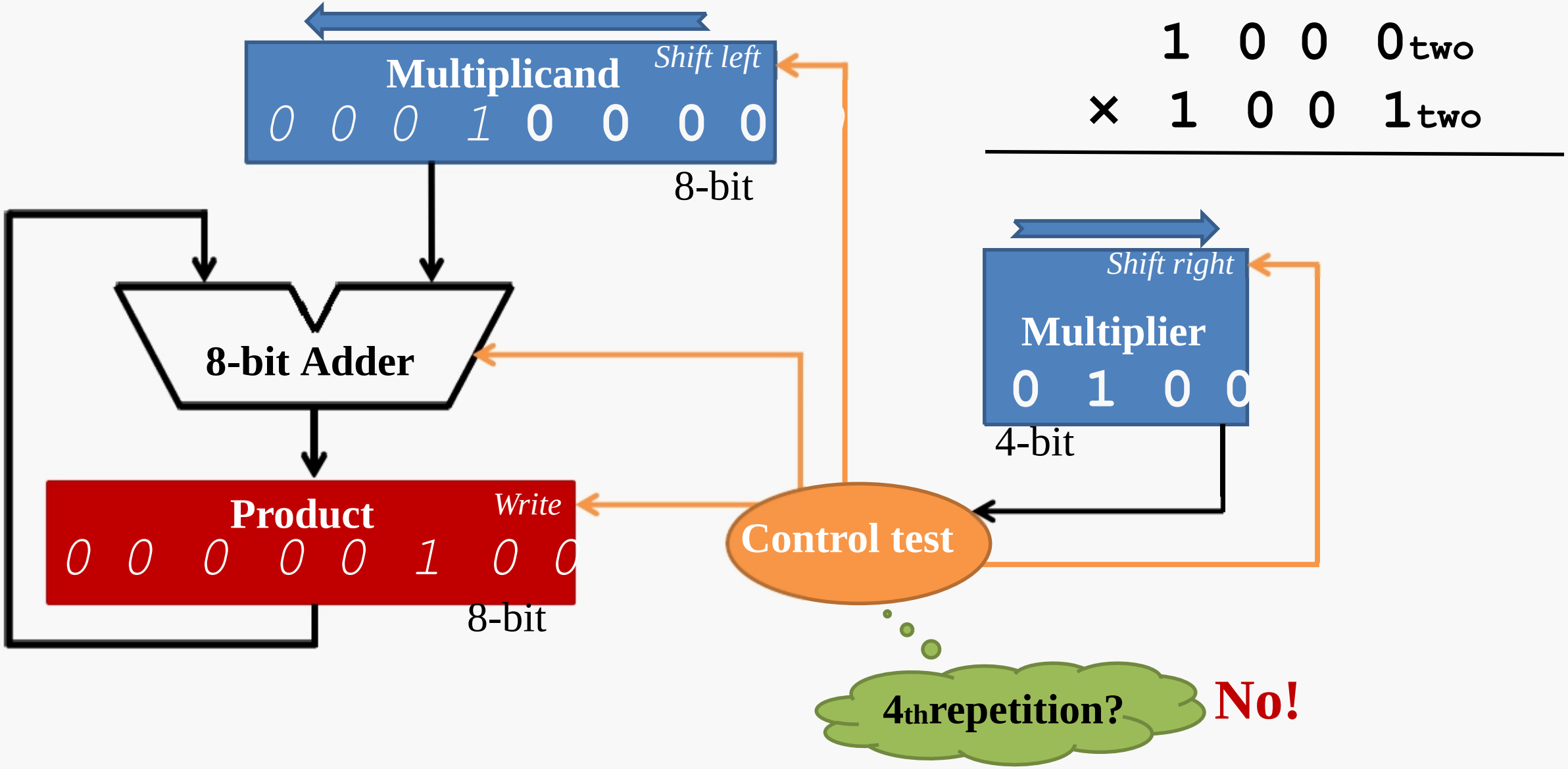


$$\begin{array}{r} 1\ 0\ 0\ 0_{two} \\ \times\ 1\ 0\ 0\ 1_{two} \\ \hline \end{array}$$

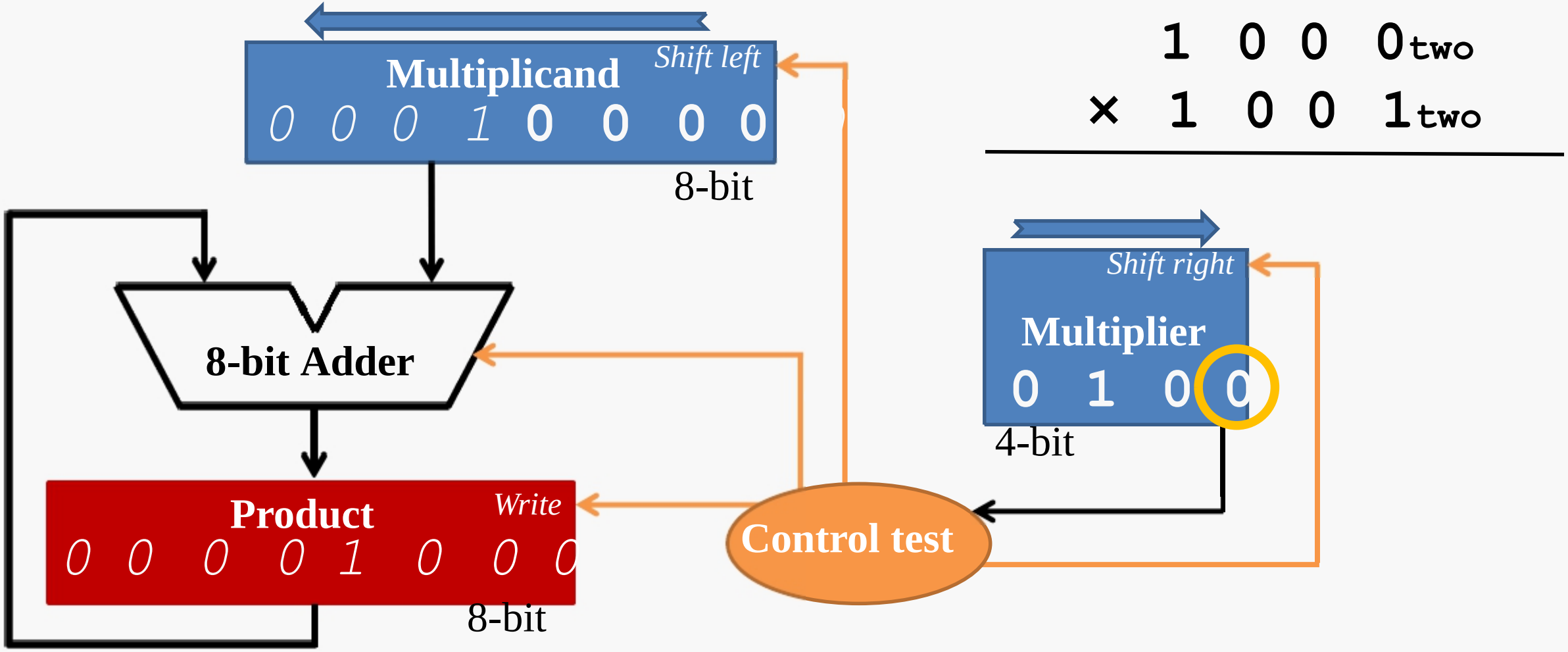
乘法器的工作过程（3）


Multiplicand
Shift left
0 0 0 1 0 0 0 0
8-bit
8-bit Adder
Product
Write
0 0 0 0 0 1 0 0
8-bit
Control test
Shift right
Multiplier
1 0 0 1
4-bit
1 0 0 0two
× 1 0 0 1two

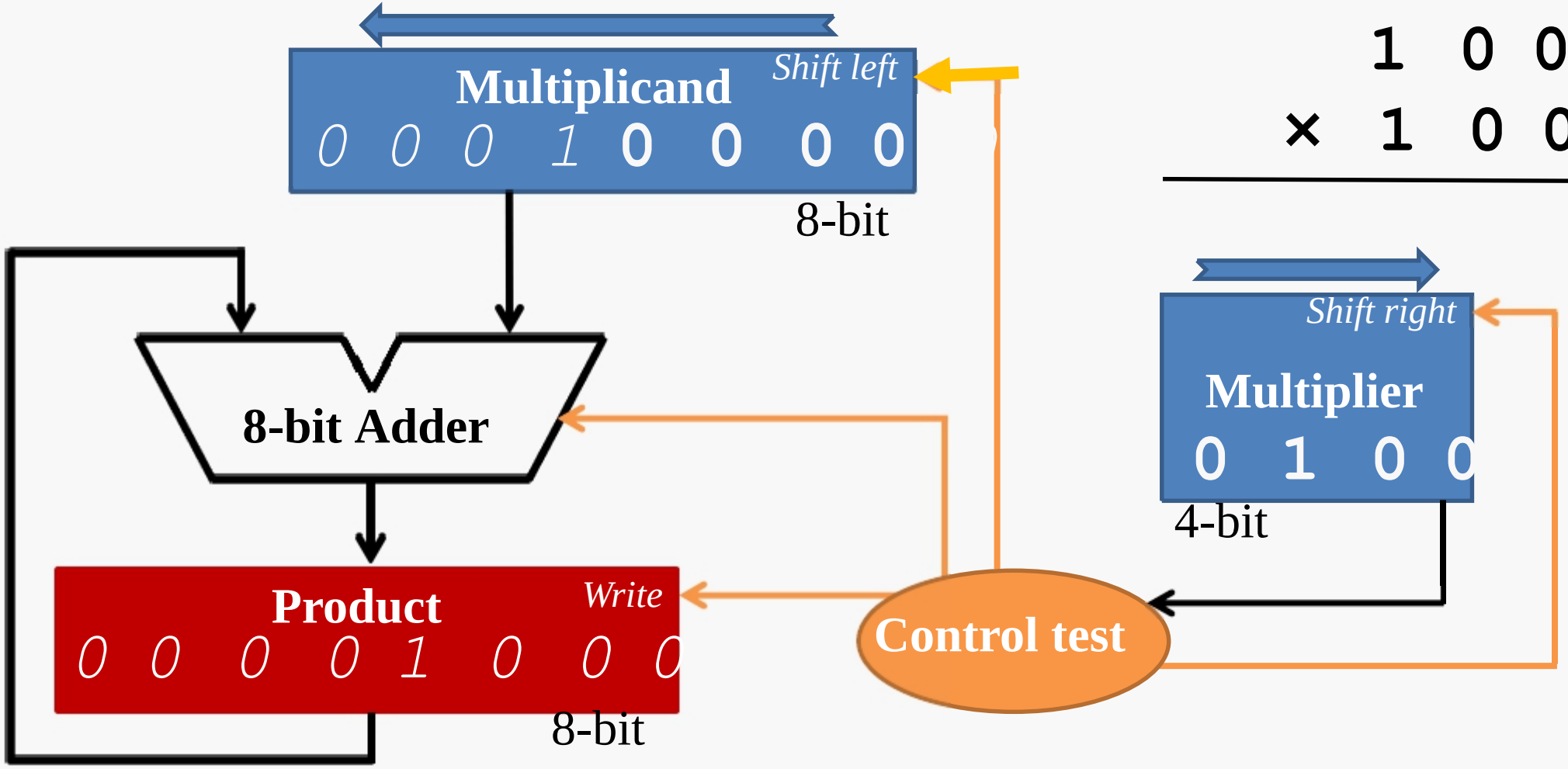
乘法器的工作过程（4）


1 0 0 0two
× 1 0 0 1two
Multiplicand
Shift left
0 0 0 1 0 0 0 0
8-bit
8-bit Adder
Shift right
Multiplier
0 1 0 0
4-bit
Product
Write
0 0 0 0 0 1 0 0
8-bit
Control test
4threpetition?
No!

乘法器的工作过程（1）　　第2轮

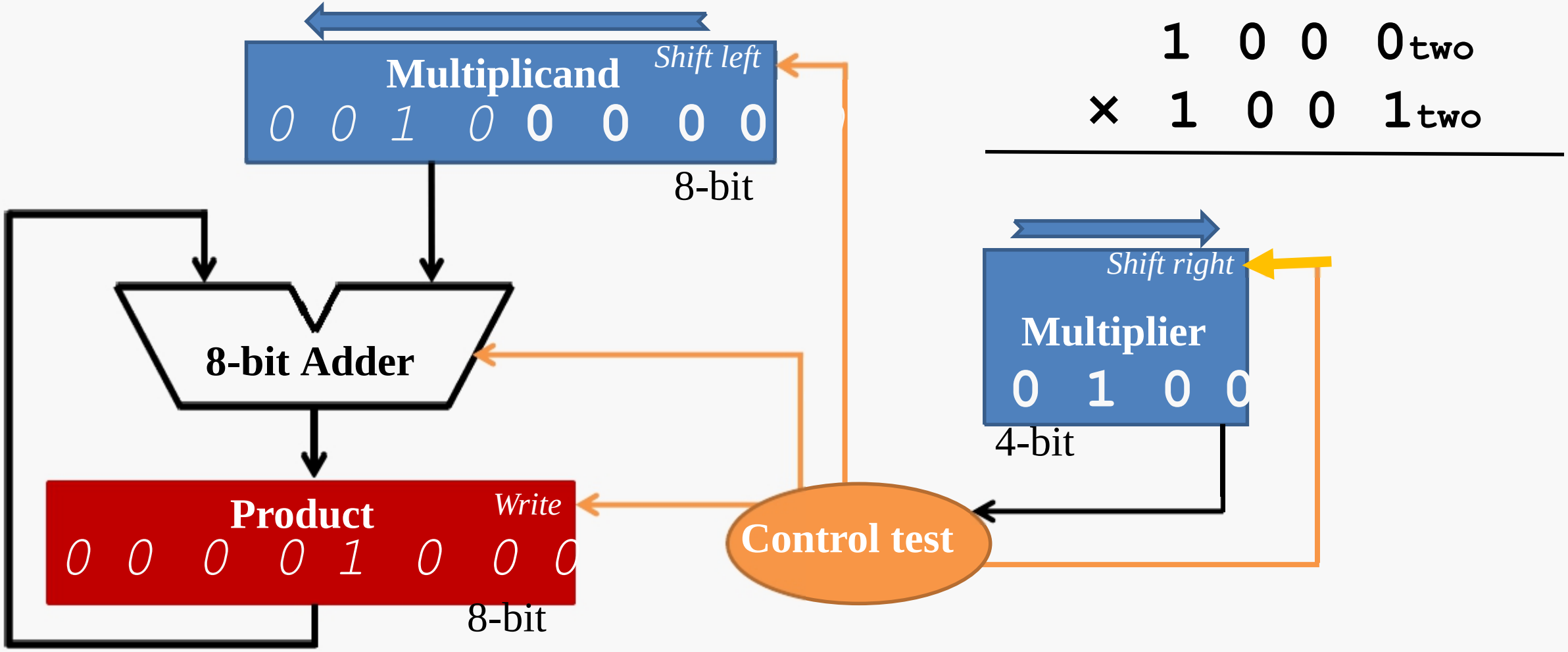


乘法器的工作过程（2）　　第2轮



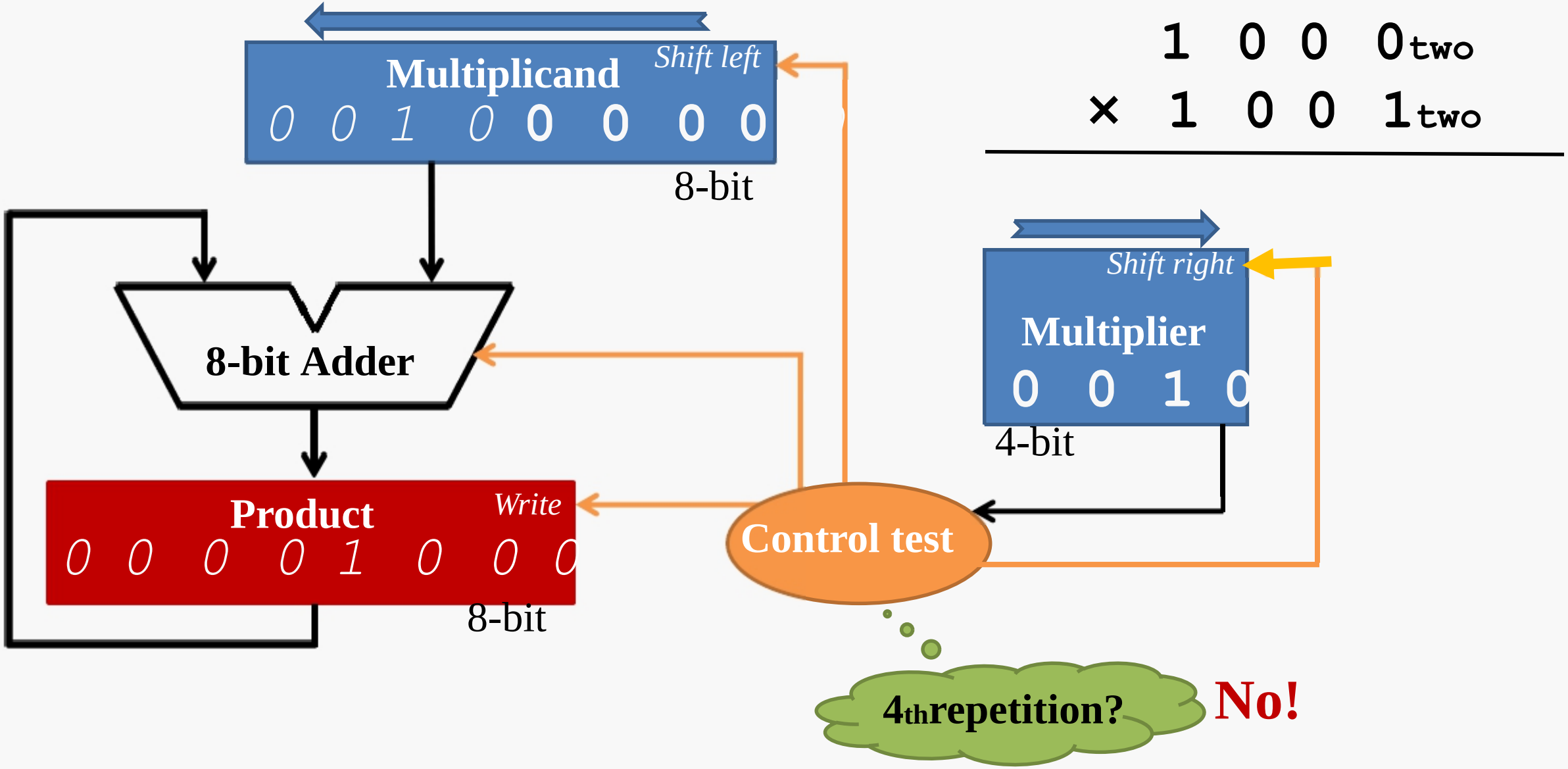
$$\begin{array}{r} 1\ 0\ 0\ 0_{two} \\ \times\ 1\ 0\ 0\ 1_{two} \\ \hline \end{array}$$

乘法器的工作过程（3）　　第2轮



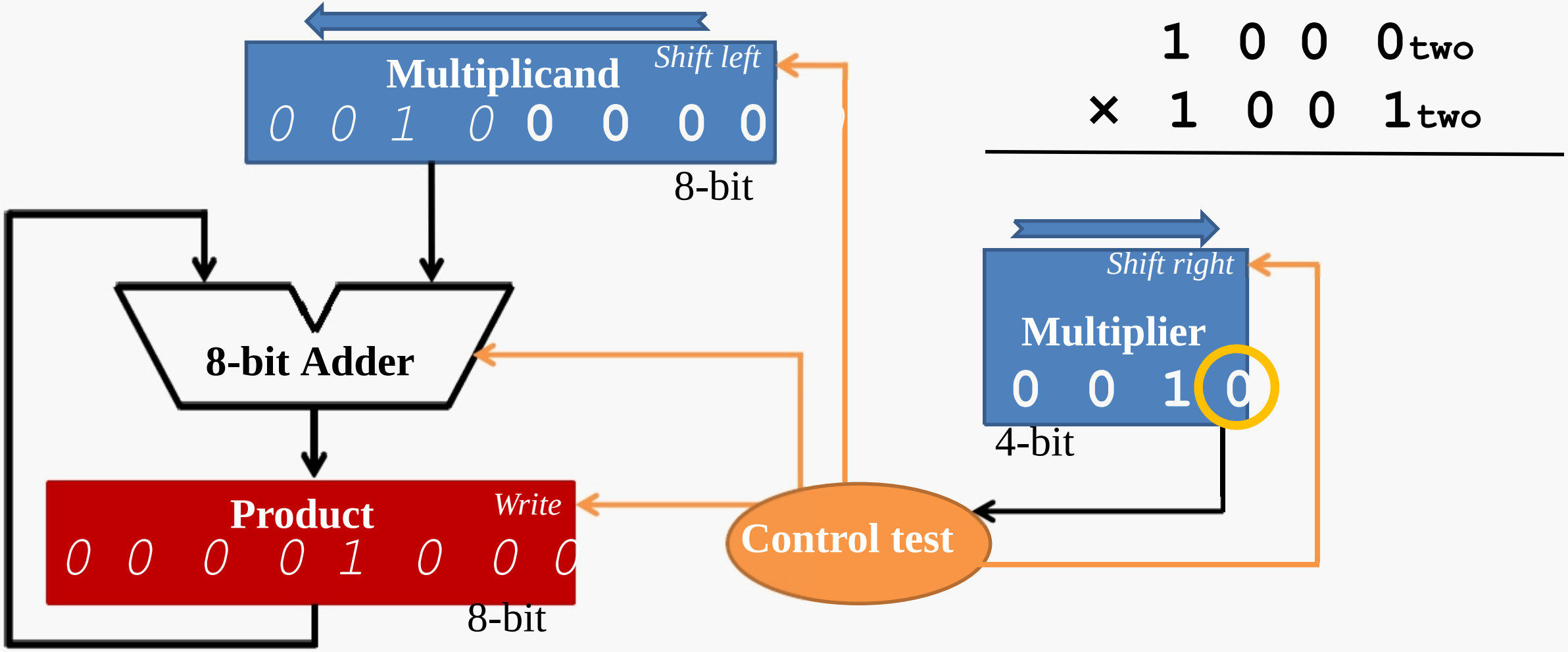
$$\begin{array}{r} 1\ 0\ 0\ 0_{two} \\ \times\ 1\ 0\ 0\ 1_{two} \\ \hline \end{array}$$

乘法器的工作过程（4）　　第2轮

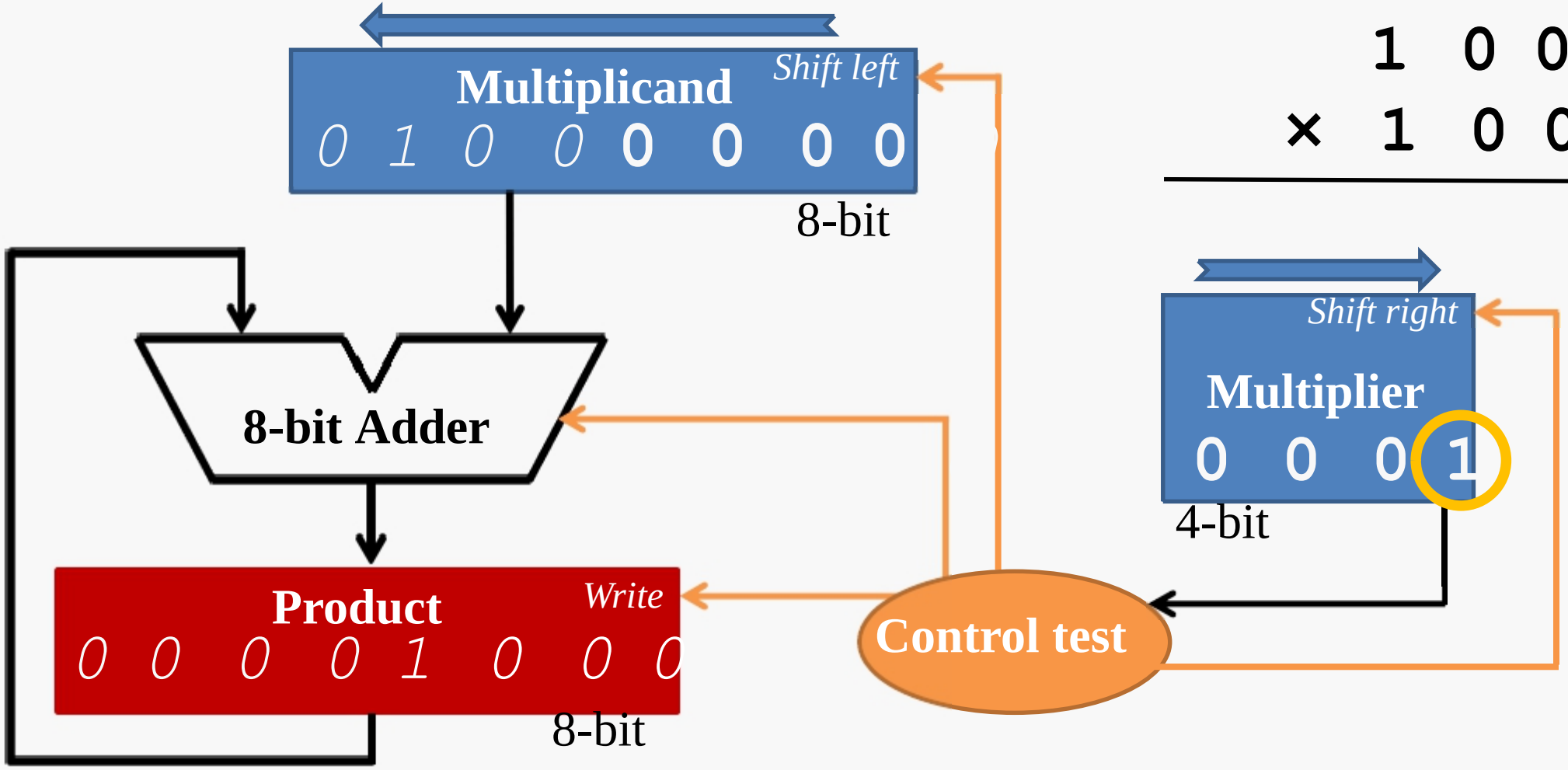


$$\begin{array}{r} 1\ 0\ 0\ 0_{two} \\ \times\ 1\ 0\ 0\ 1_{two} \\ \hline \end{array}$$

乘法器的工作过程（1）　　第3轮

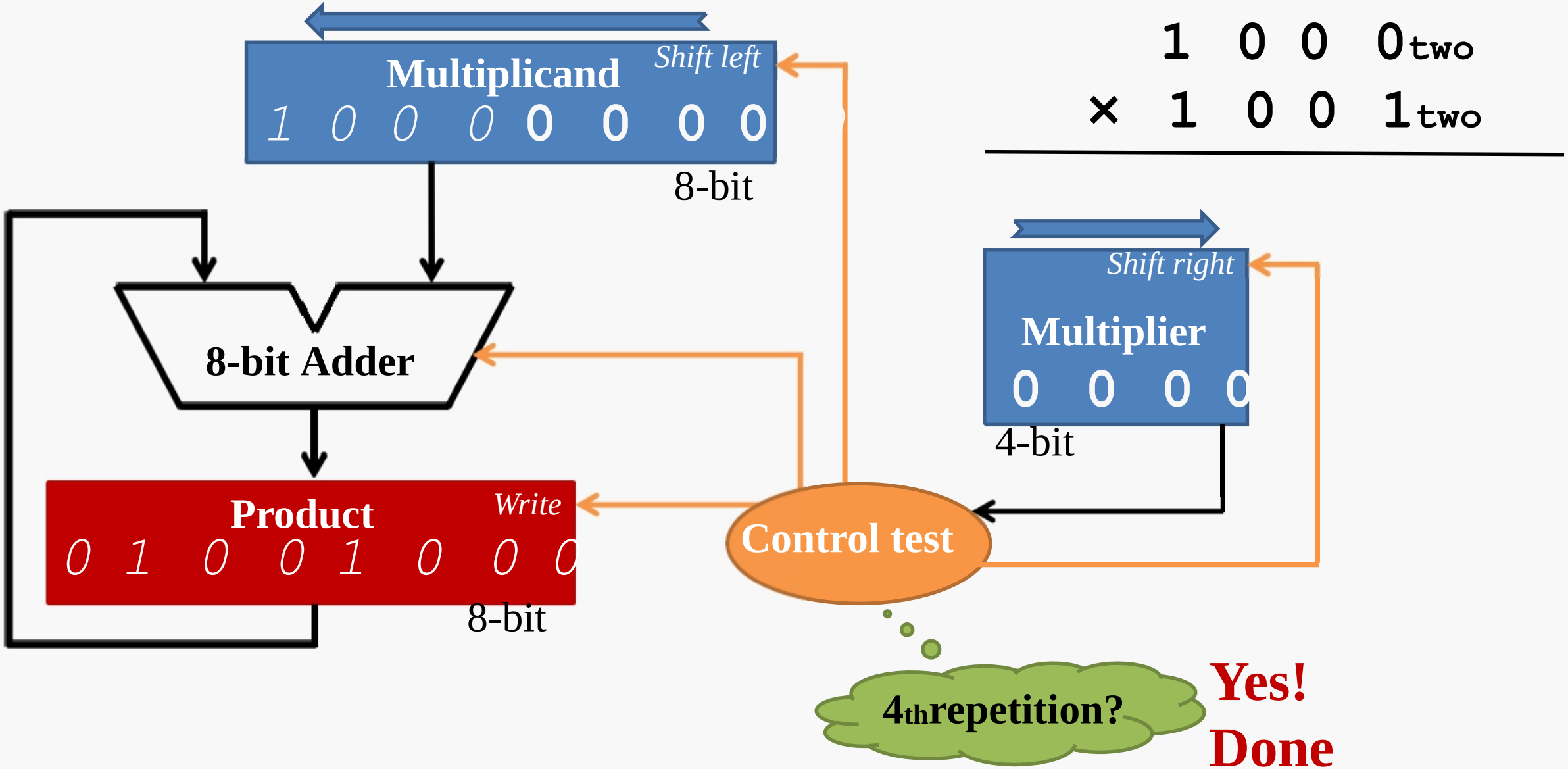


乘法器的工作过程（1）　　第4轮

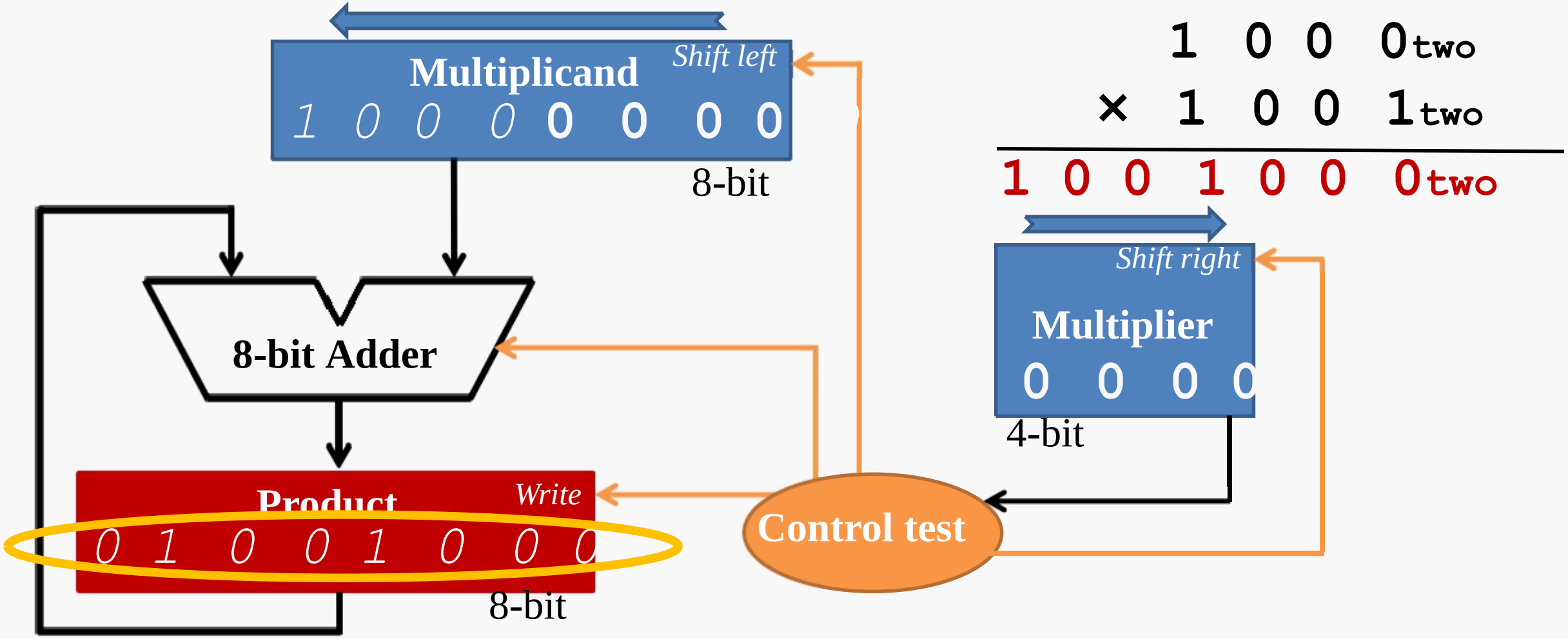


$$\begin{array}{r} 1\ 0\ 0\ 0_{two} \\ \times\ 1\ 0\ 0\ 1_{two} \\ \hline \end{array}$$

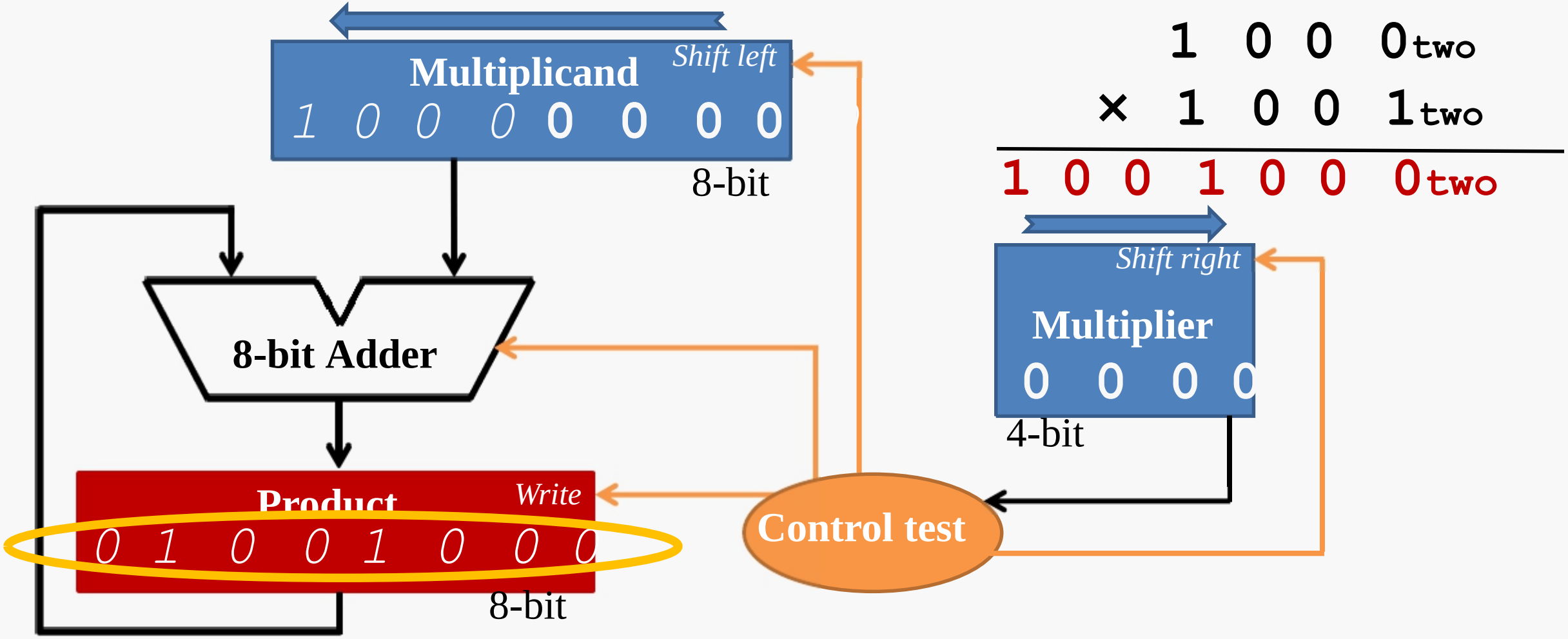
乘法器的工作过程（4）　　第4轮



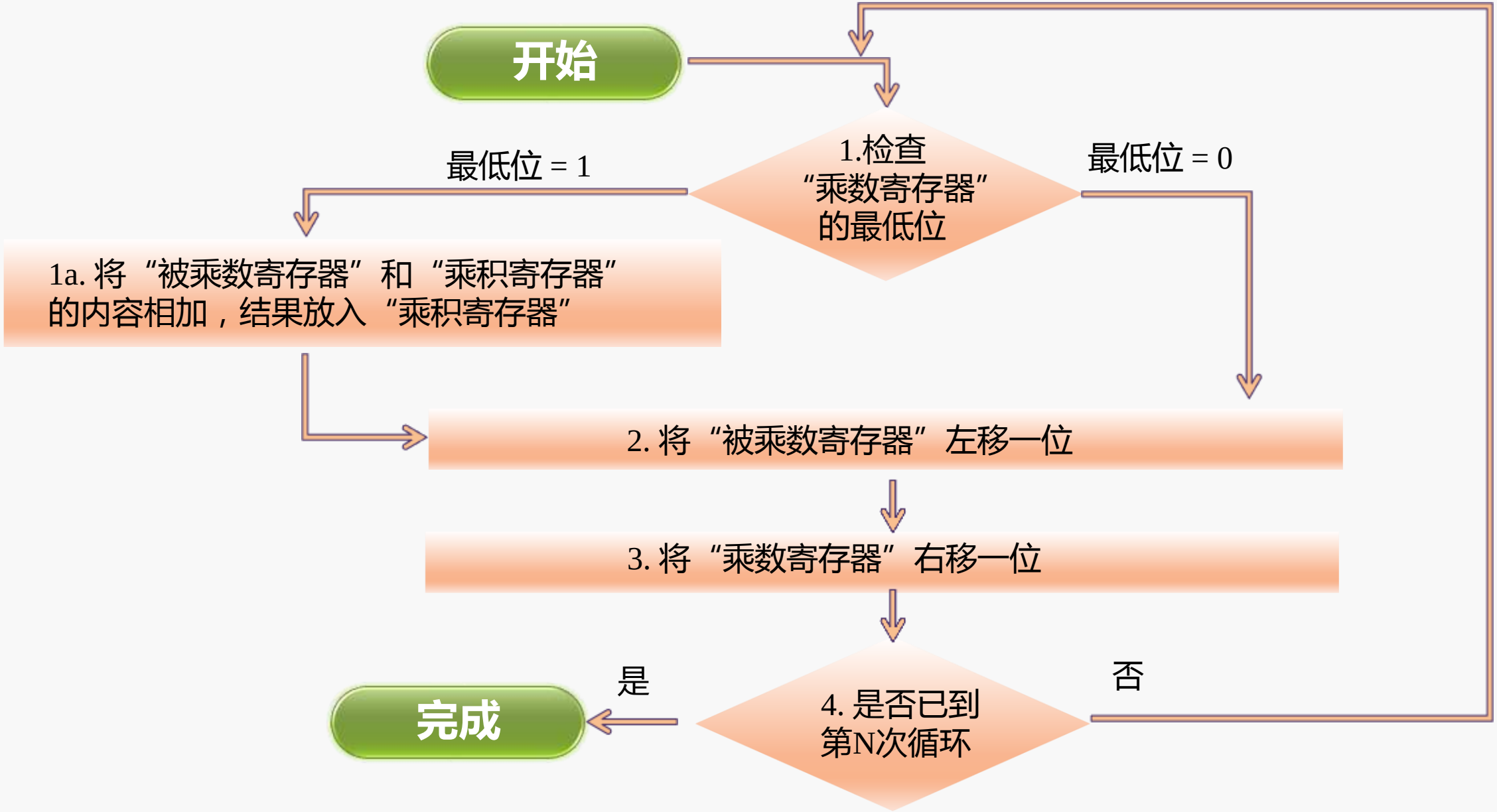
乘法器的运算结果


Multiplicand
Shift left
1 0 0 0 0 0 0 0
8-bit
8-bit Adder
Product
Write
0 1 0 0 1 0 0 0
8-bit
Control test
Shift right
Multiplier
0 0 0 0
4-bit
1 0 0 0two
× 1 0 0 1two
1 0 0 1 0 0 0two

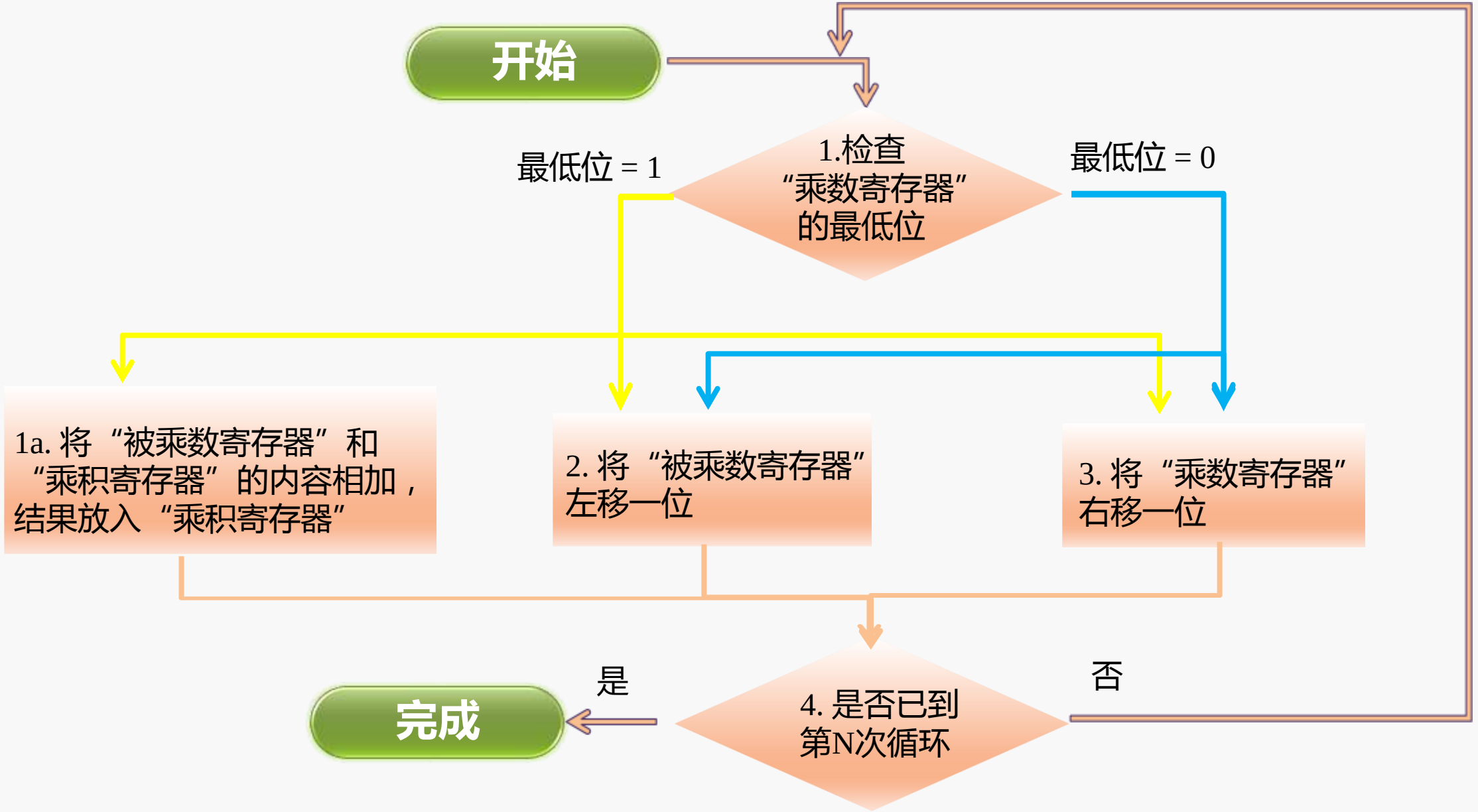
乘法器的运算结果


Multiplicand
Shift left
1 0 0 0 0 0 0 0
8-bit
8-bit Adder
Product
Write
0 1 0 0 1 0 0 0
8-bit
Control test
Shift right
Multiplier
0 0 0 0
4-bit
1 0 0 0two
× 1 0 0 1two
1 0 0 1 0 0 0two

N位乘法器的工作流程图



对比：N位乘法器的工作流程（优化后）



第四章　乘法器和除法器

1.乘法的运算过程

2.乘法器的实现

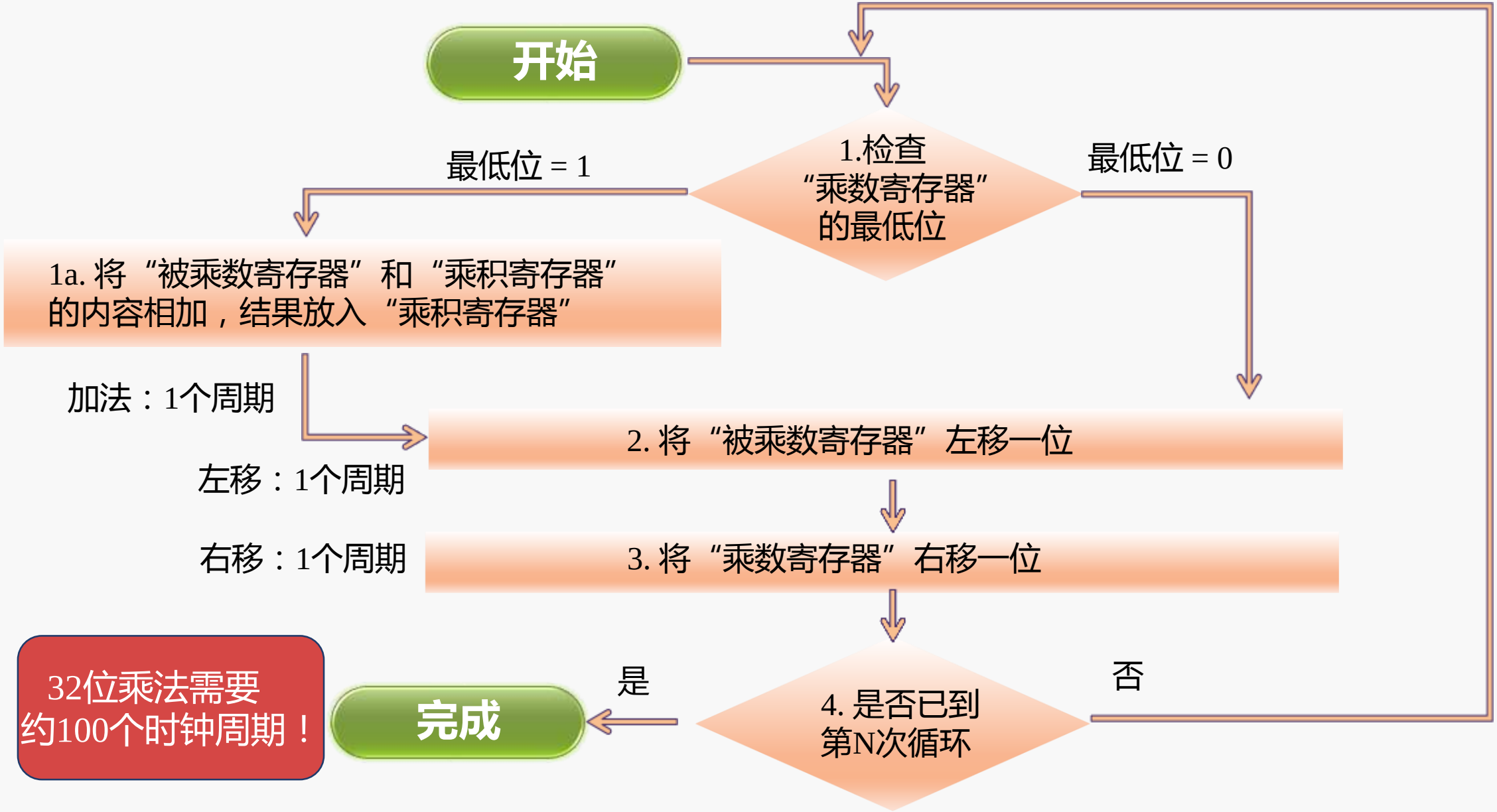
3.乘法器的优化

4.除法的运算过程

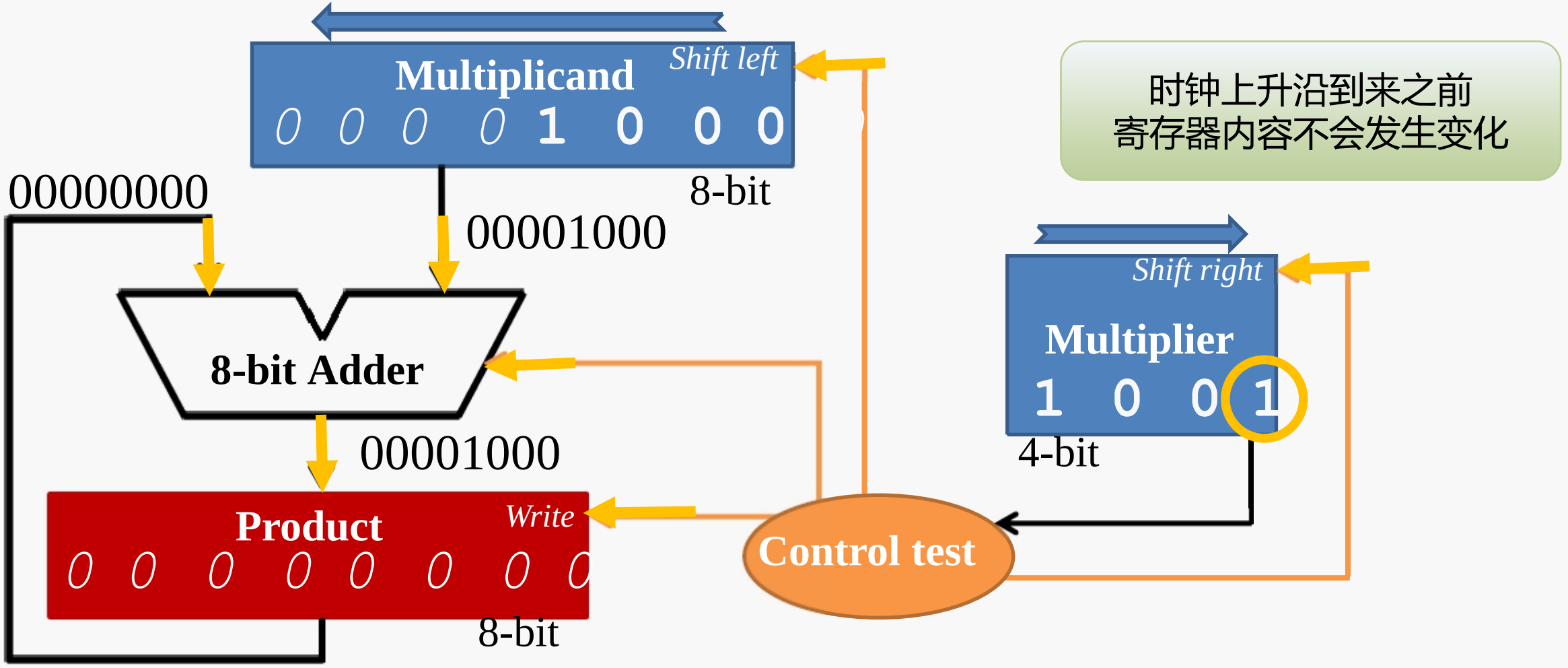
5.除法器的实现

6.除法器的优化

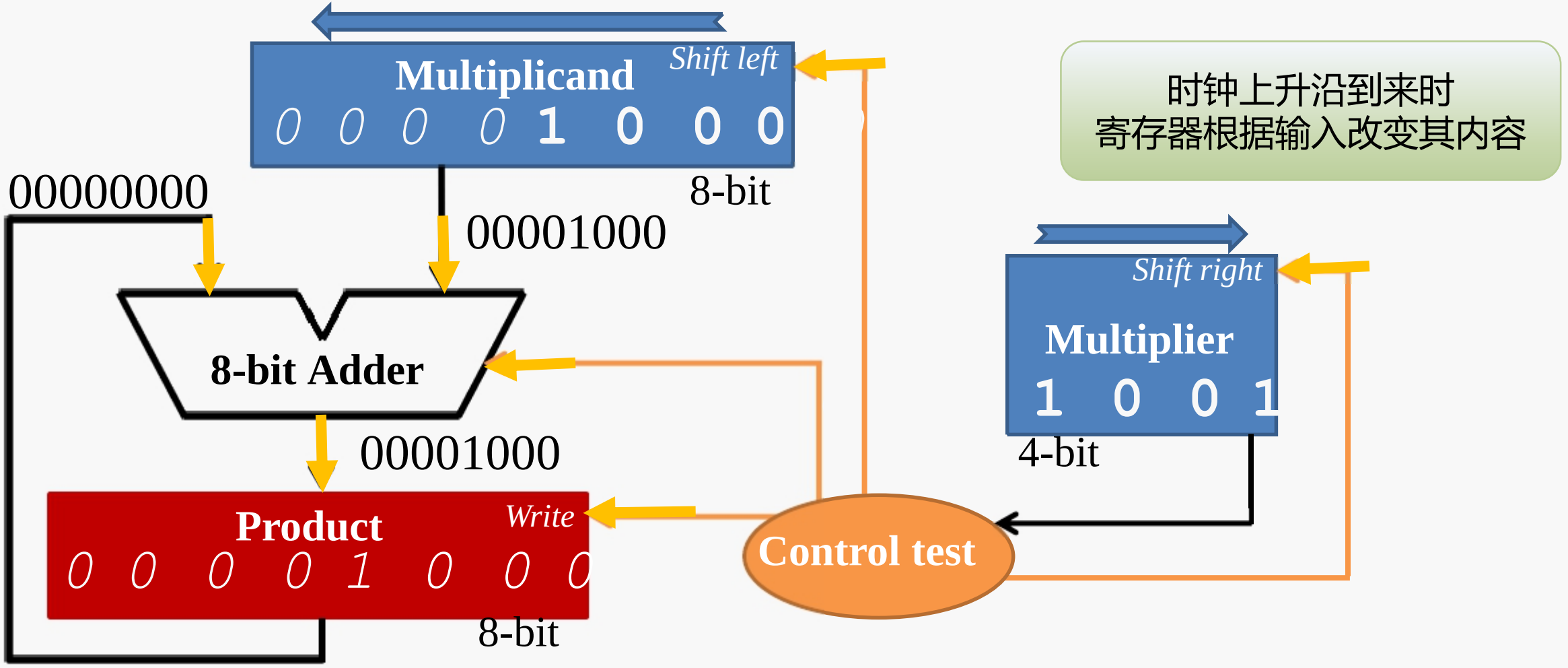
N位乘法器的工作流程图



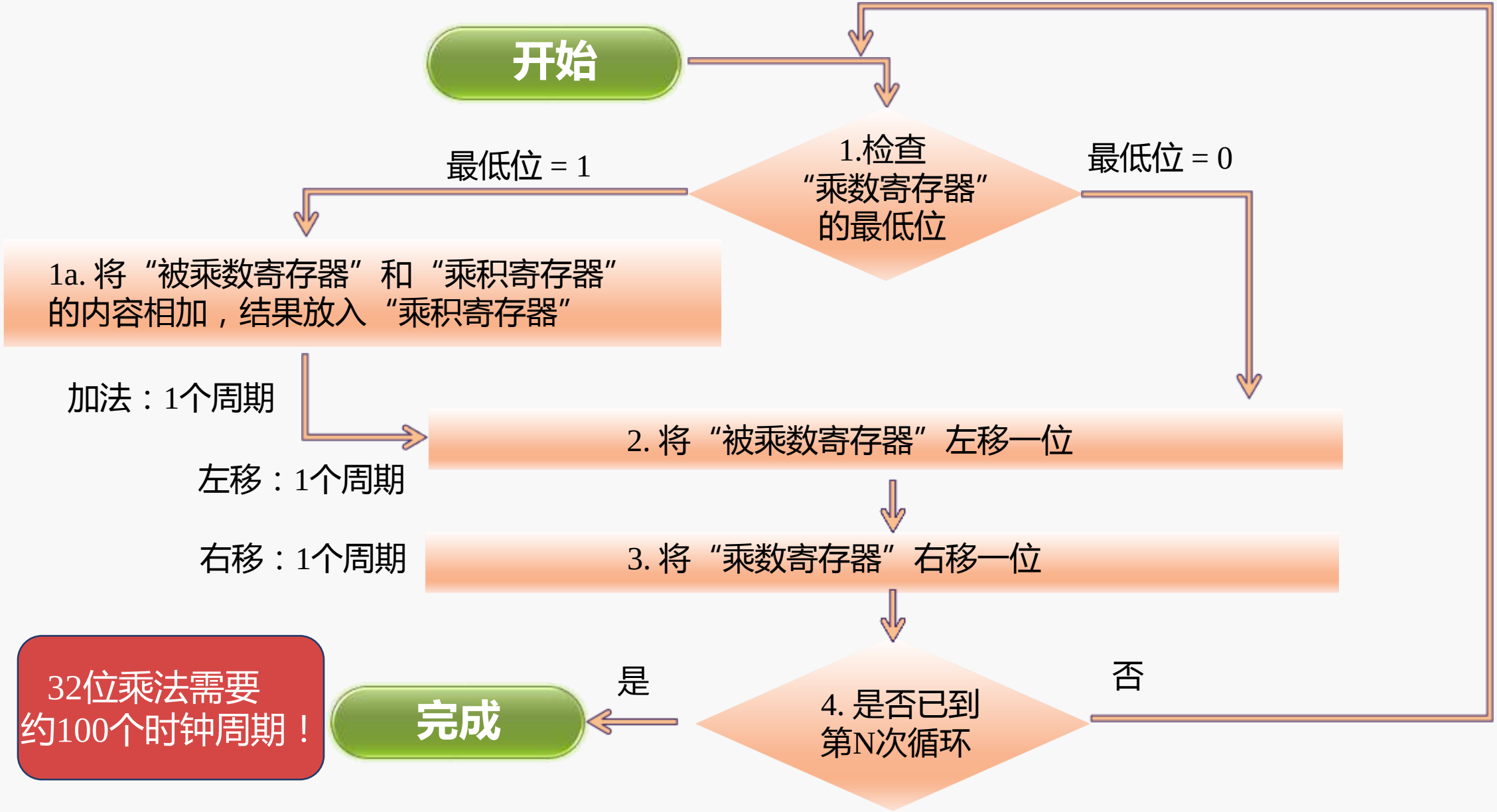
乘法器的优化1：加法移位并行



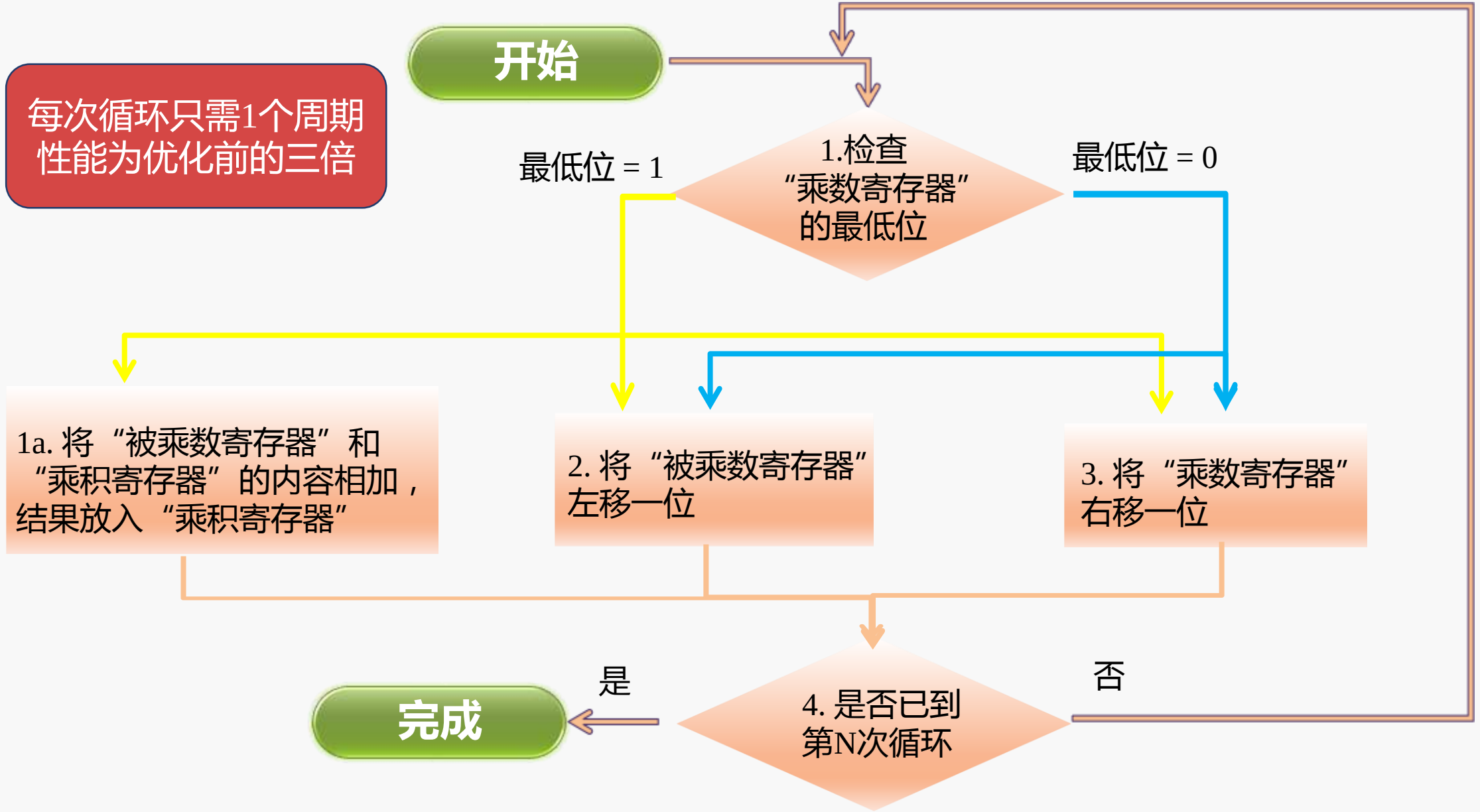
乘法器的优化1：加法移位并行



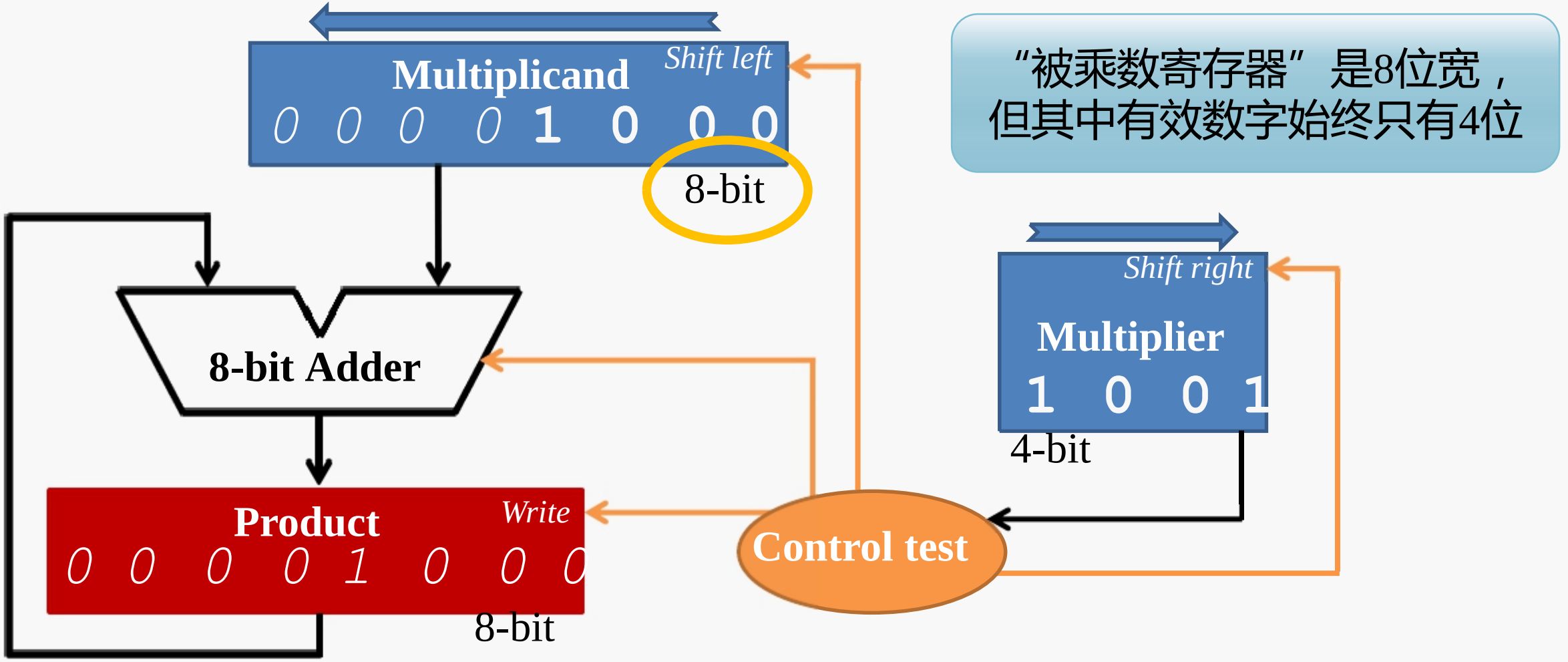
N位乘法器的工作流程图



N位乘法器的工作流程化

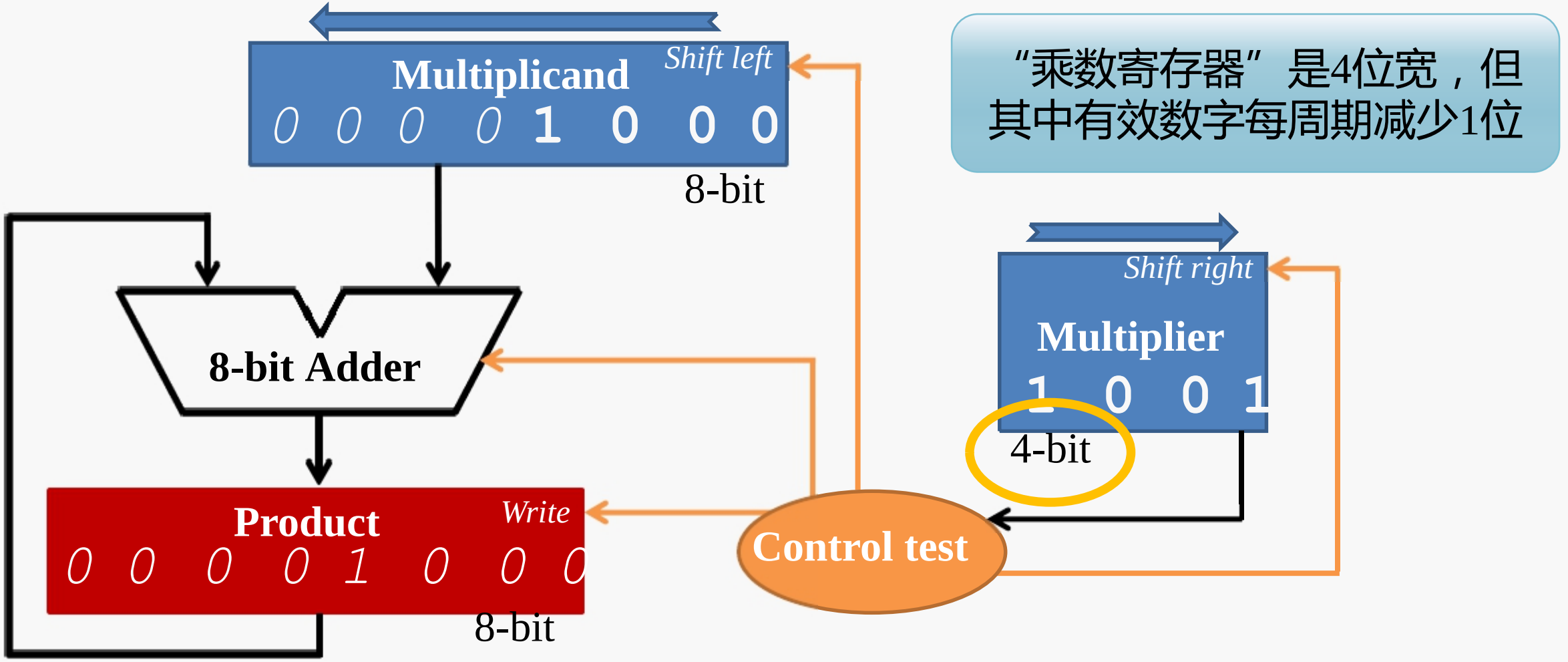


乘法器的优化2：减少不必要的硬件资源



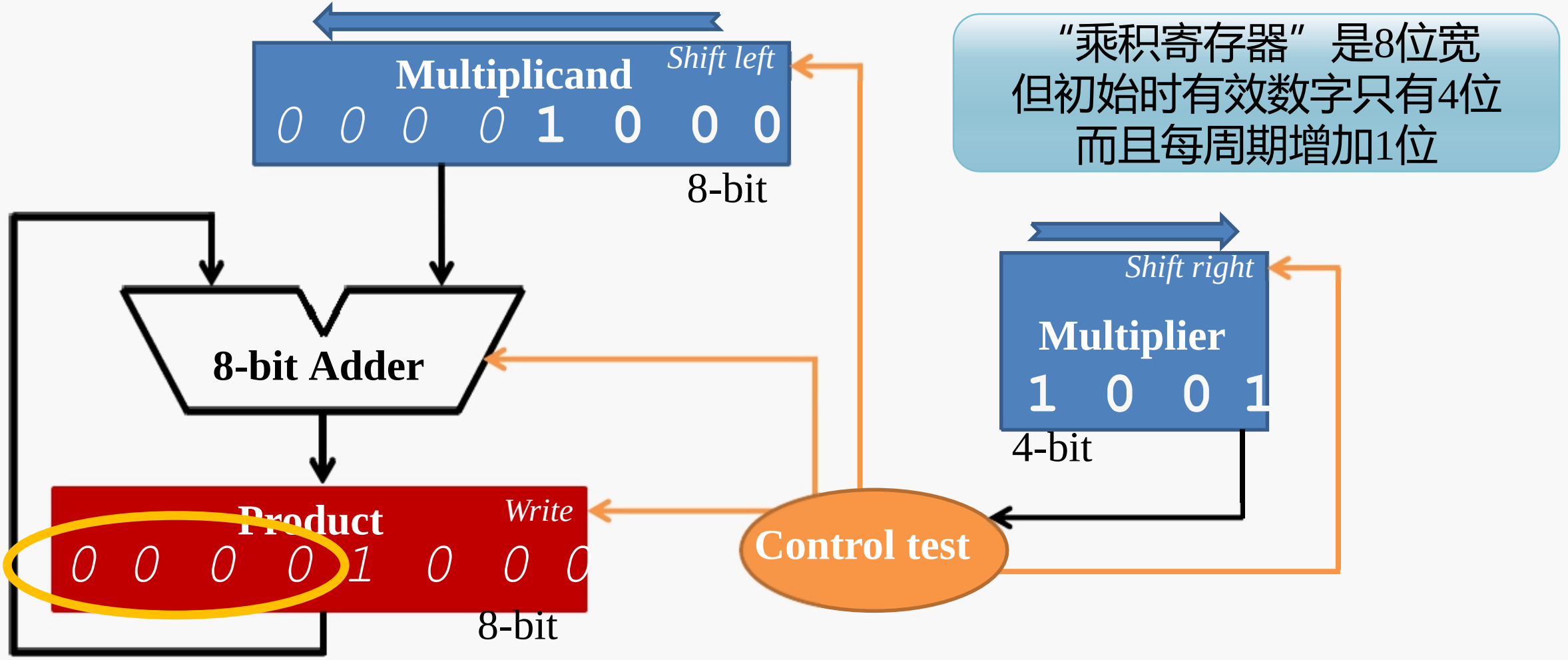
“被乘数寄存器”是8位宽，
但其中有效数字始终只有4位

乘法器的优化2：减少不必要的硬件资源

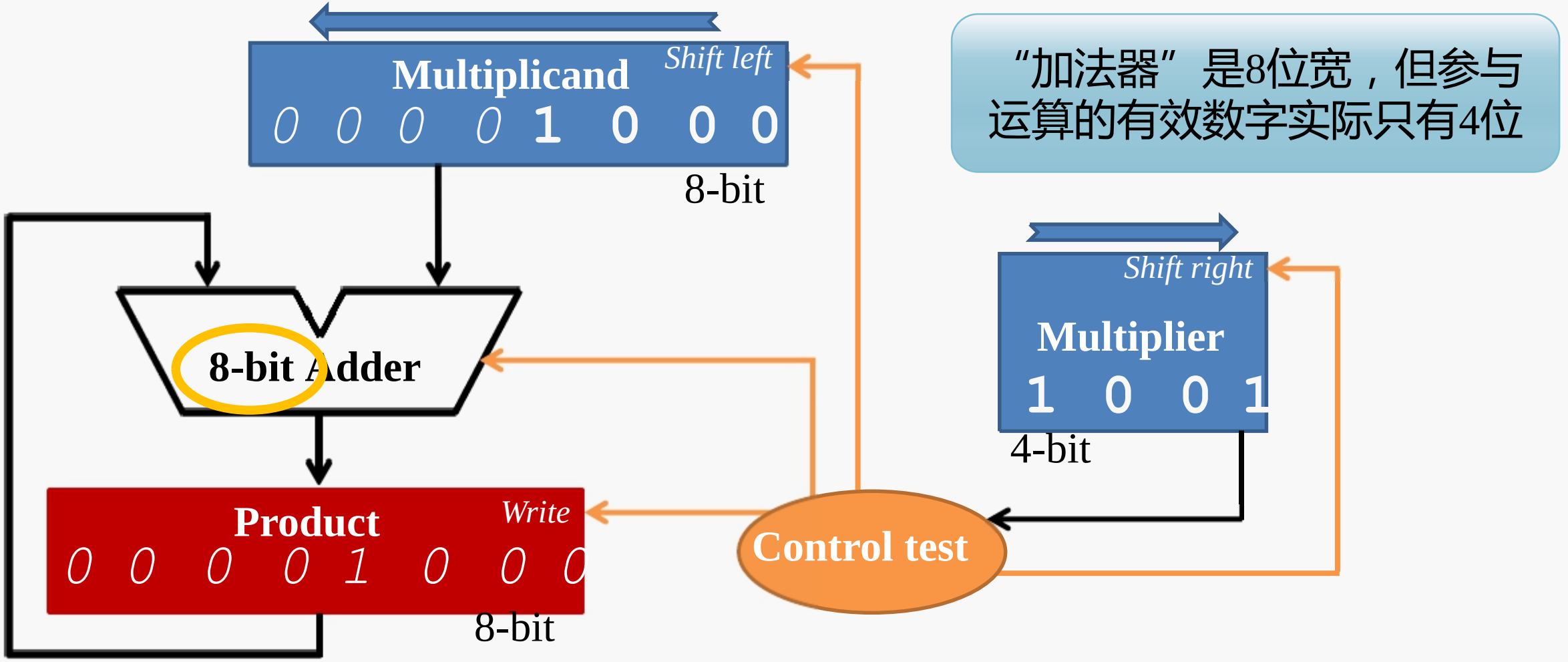


“乘数寄存器”是4位宽，但其中有效数字每周期减少1位

乘法器的优化2：减少不必要的硬件资源



乘法器的优化2：减少不必要的硬件资源



“加法器”是8位宽，但参与运算的有效数字实际只有4位

优化方案分析

“被乘数寄存器”8位宽带左移
但其中有效数字始终只有4位

“乘数寄存器”4位宽带右移
但其中有效数字每周期减少1位

“乘积寄存器”8位宽
但初始时有效数字只有4位
而且每周期增加1位

“加法器”8位宽，但参与运算
的有效数字实际只有4位

优化方案分析

“被乘数寄存器”缩减为4位而且取消左移功能

“乘数寄存器”4位宽带右移但其中有效数字每周期减少1位

“乘积寄存器”8位宽但初始时有效数字只有4位而且每周期增加1位

“加法器”8位宽，但参与运算的有效数字实际只有4位

优化方案分析

4-bit

8-bit

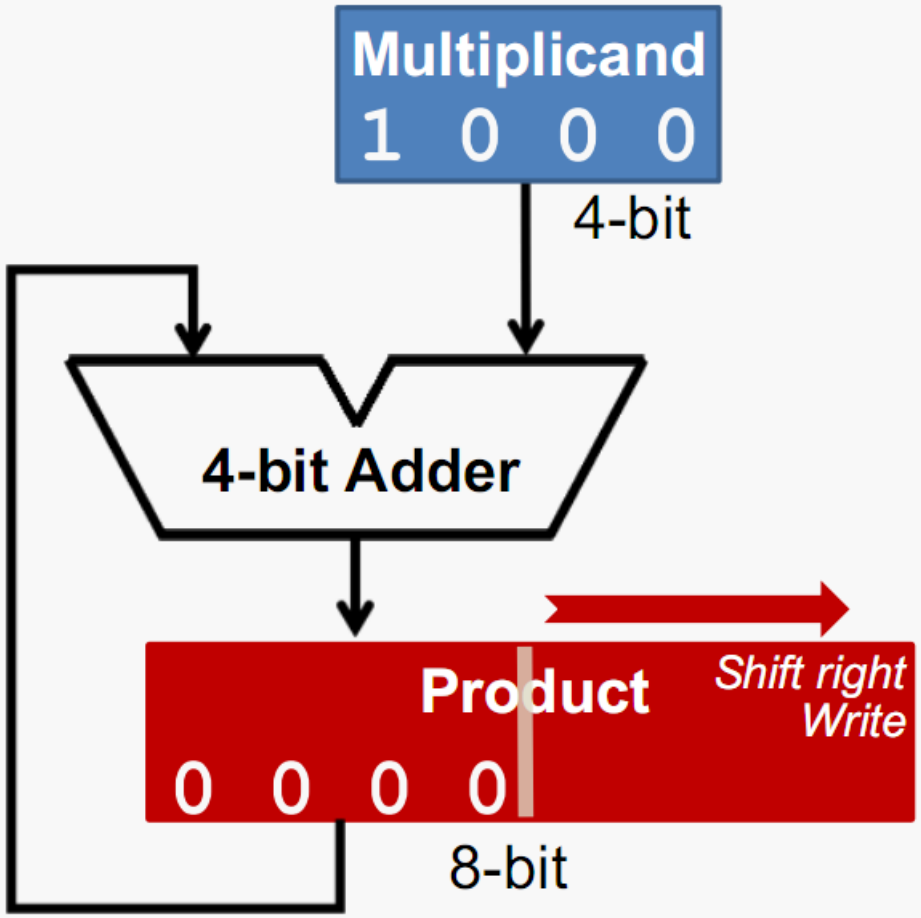
“被乘数寄存器”缩减为4位而且取消左移功能

“乘数寄存器”4位宽带右移但其中有效数字每周期减少1位

“乘积寄存器”增加右移功能乘积初始值置于其中高4位，随着运算过程不断右移

“加法器”8位宽，但参与运算的有效数字实际只有4位

优化方案分析



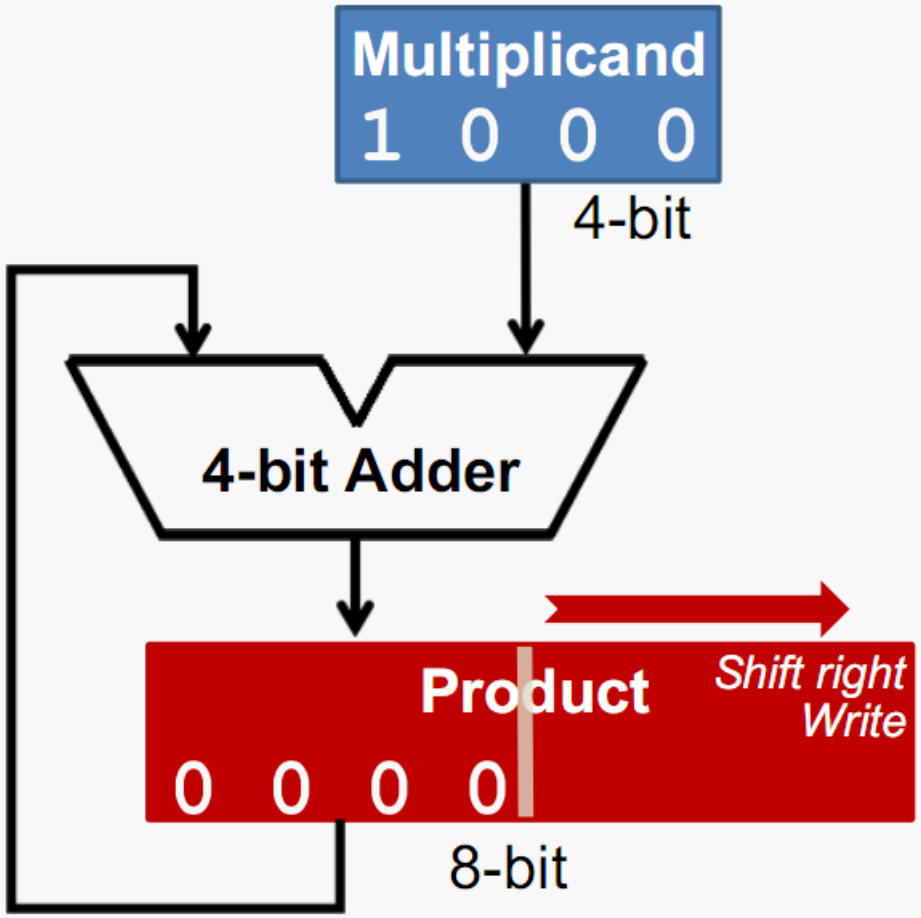
“被乘数寄存器”缩减为4位而且取消左移功能

“乘数寄存器”4位宽带右移但其中有效数字每周期减少1位

“乘积寄存器”增加右移功能乘积初始值置于其中高4位，随着运算过程不断右移

“加法器”缩减为4位宽，“乘积寄存器”只有高4位参与运算

优化方案分析



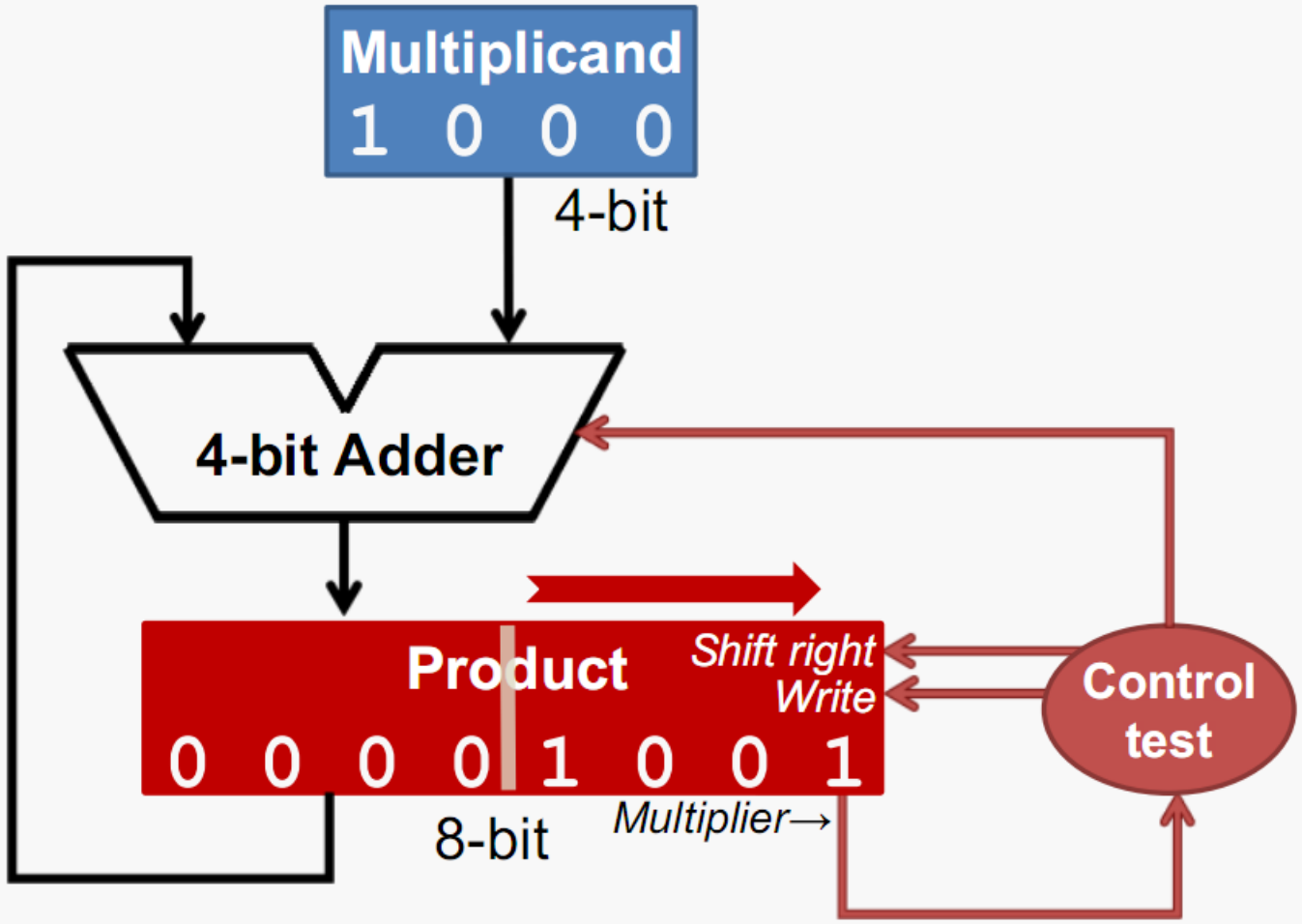
“被乘数寄存器”缩减为4位而且取消左移功能

取消“乘数寄存器”，乘数初始置于“乘积寄存器”低4位

“乘积寄存器”增加右移功能乘积初始值置于其中高4位，随着运算过程不断右移

“加法器”缩减为4位宽，“乘积寄存器”只有高4位参与运算

乘法器的优化2 ：减少不必要的硬件资源



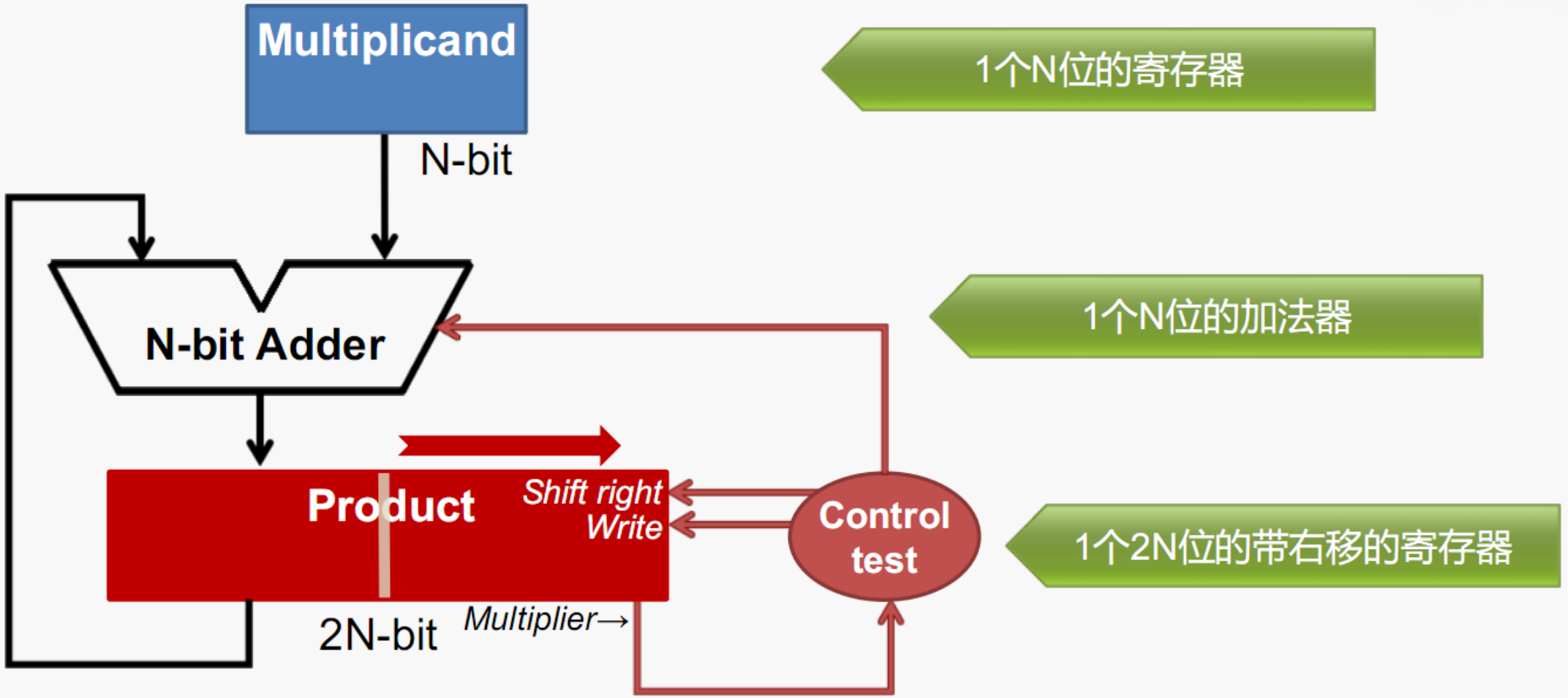
“被乘数寄存器”缩减为4位而且取消左移功能

取消“乘数寄存器”，乘数初始置于“乘积寄存器”低4位

“乘积寄存器”增加右移功能
乘积初始值置于其中高4位，随着运算过程不断右移

“加法器”缩减为4位宽，“乘积寄存器”只有高4位参与运算

N位乘法器的实现结构


Multiplicand
N-bit
N-bit Adder
Product
Shift right
Write
2N-bit
Multiplier→
Control
test
1个N位的寄存器
1个N位的加法器
1个2N位的带右移的寄存器

第四章　乘法器和除法器

1.乘法的运算过程

2.乘法器的实现

3.乘法器的优化

4.除法的运算过程

5.除法器的实现

6.除法器的优化

除法的运算过程（示例1）

除数 Divisor

$1\ 0\ 0\ 0_{ten}$

$$
\begin{array}{r|l}
 & 1\ 0\ 0\ 1_{ten} \\
\hline
1\ 0\ 0\ 1\ 0\ 1\ 0_{ten} & \\
-\ 1\ 0\ 0\ 0 & \\
\hline
1\ 0 & \\
1\ 0\ 1 & \\
1\ 0\ 1\ 0 & \\
-\ 1\ 0\ 0\ 0 & \\
\hline
1\ 0_{ten} &
\end{array}
$$

商 Quotient

被除数 Dividend

余数 Remainder

Dividend = Quotient × Divisor + Remainder

除法的运算过程（示例2）

	十进制	二进制	位宽
被除数	7	00000111	8-bit
注：可以视为初始时的余数			
除数	2	0010	4-bit
注：可视为不断右移，并和被除数相减			
余数	1	00000001	8-bit
注：可与被除数共享一个寄存器			
商	3	0011	4-bit
注：每个bit依次生成，可视为不断左移			

第四章　乘法器和除法器

1.乘法的运算过程

2.乘法器的实现

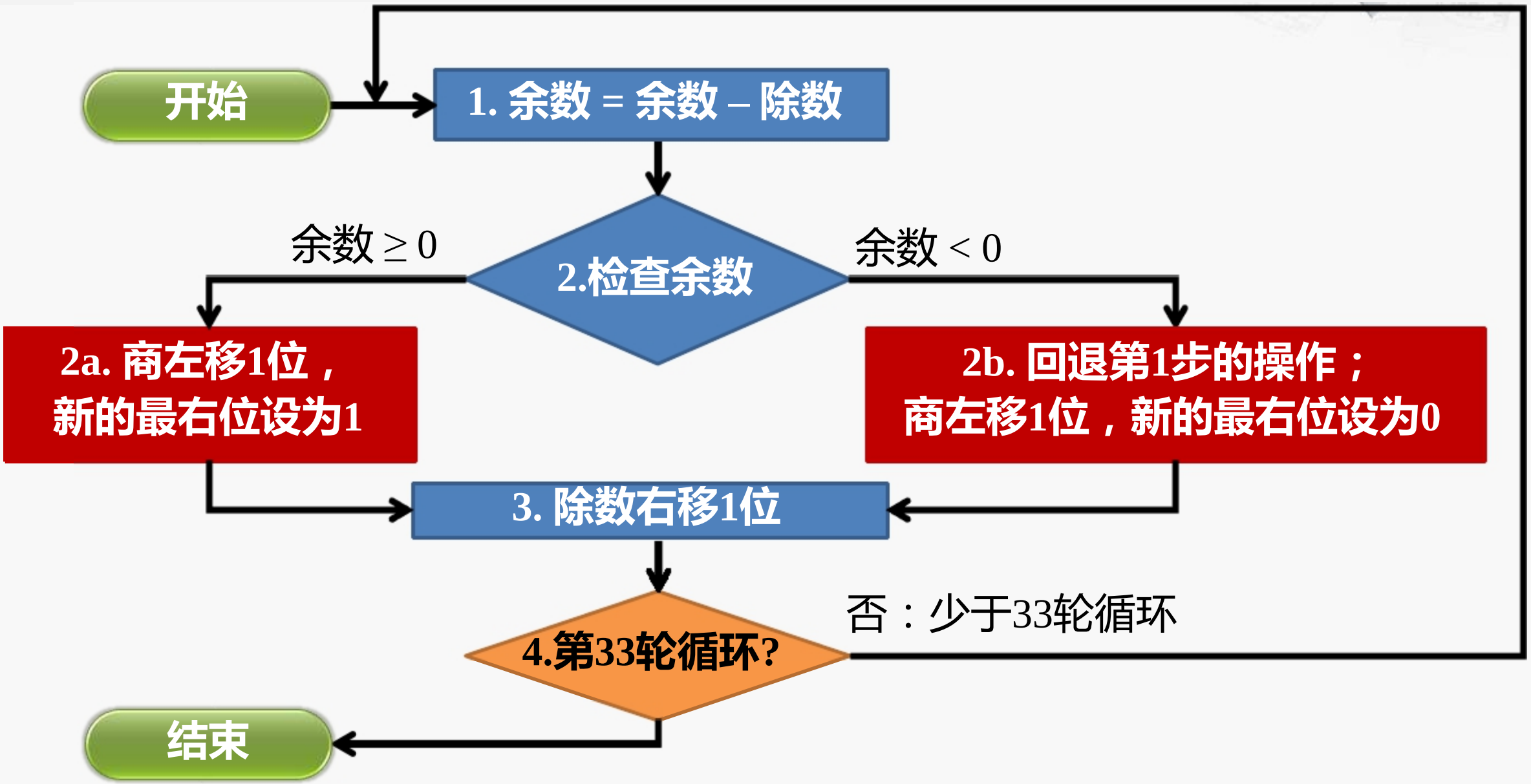
3.乘法器的优化

4.除法的运算过程

5.除法器的实现

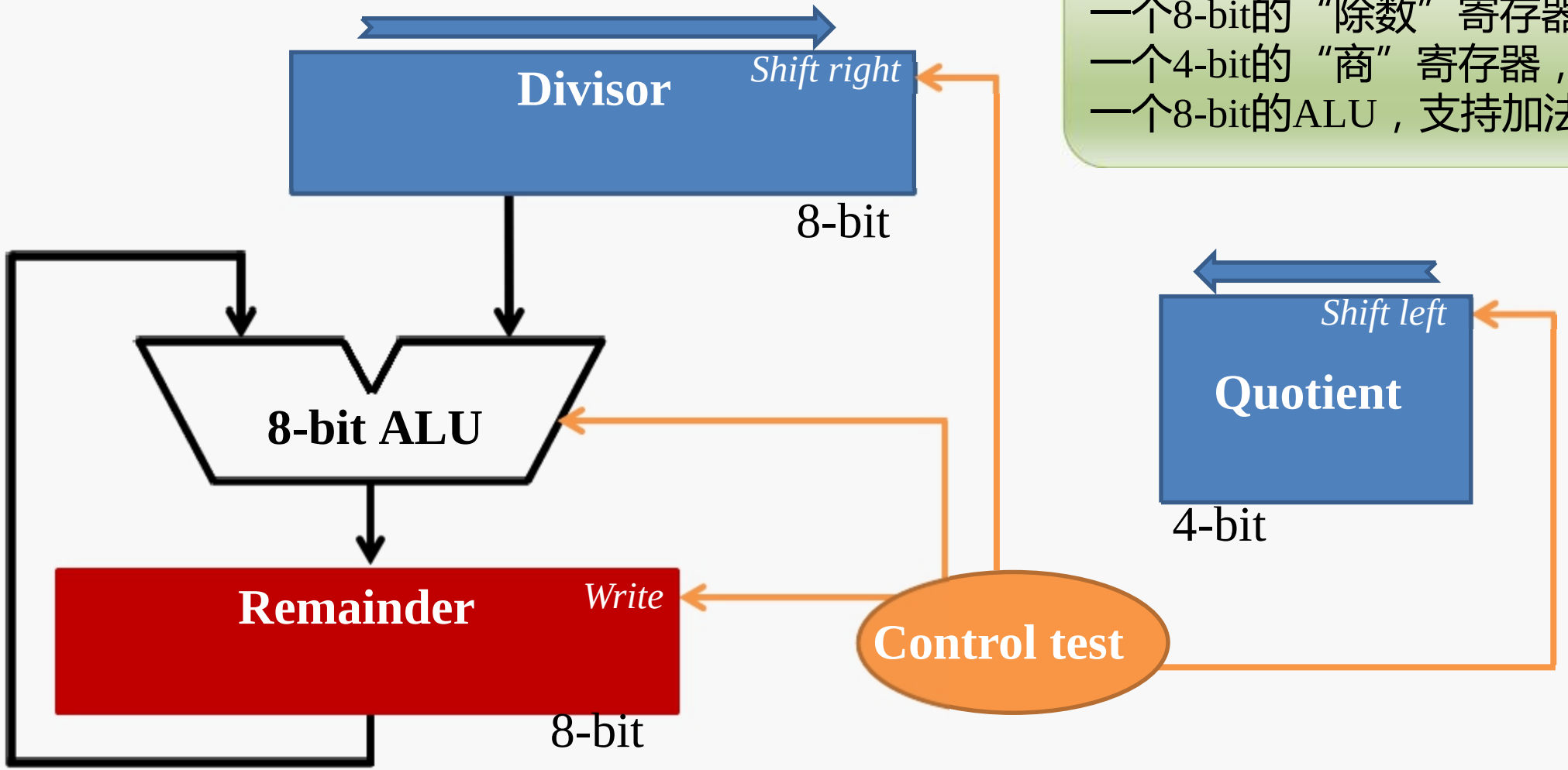
6.除法器的优化

32-bit 除法器的工作流程图



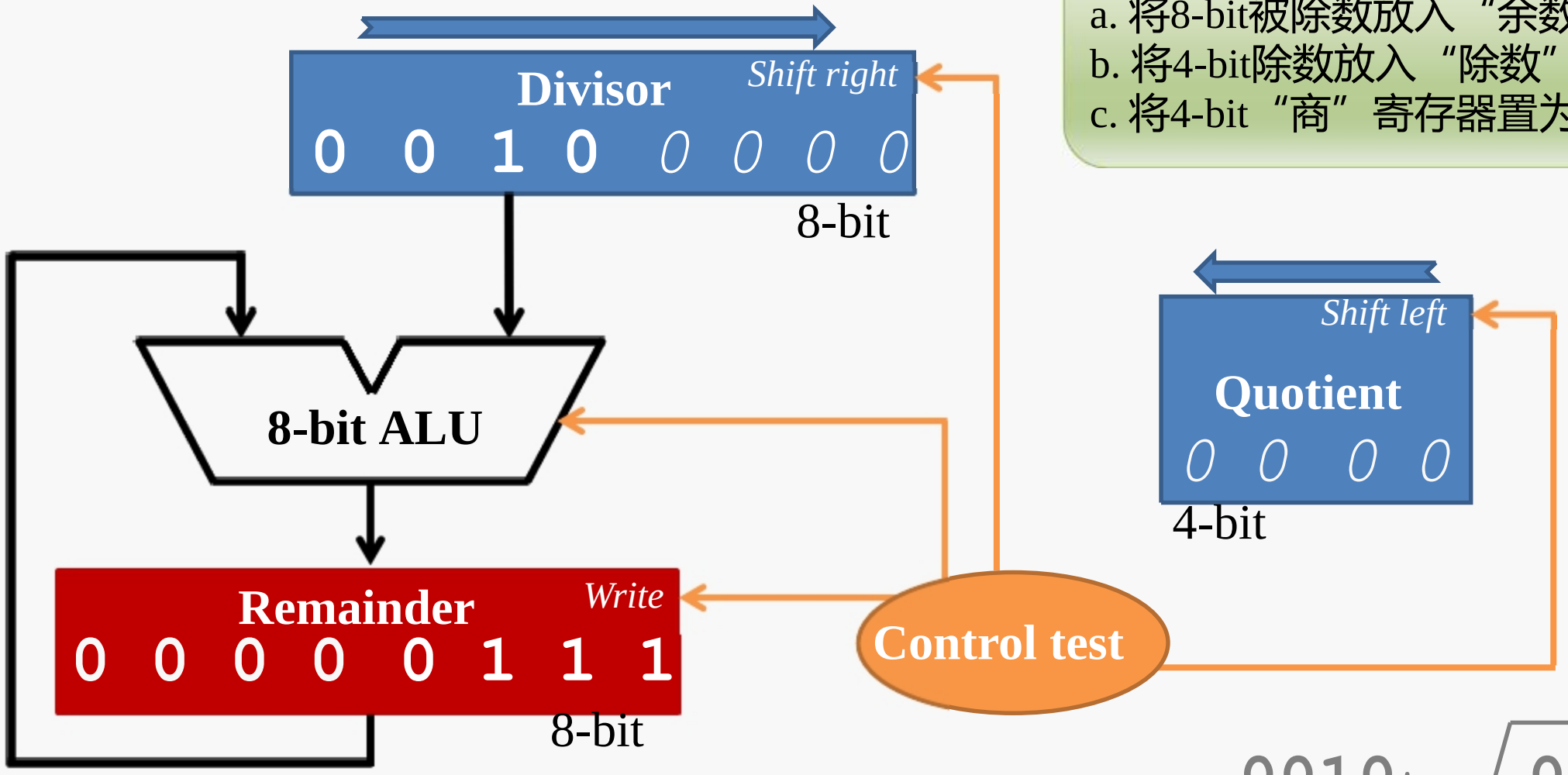
4-bit 除法器的实现示例

一个8-bit的“余数”寄存器
一个8-bit的“除数”寄存器，带右移功能
一个4-bit的“商”寄存器，带左移功能
一个8-bit的ALU，支持加法和减法运算



除法器的工作过程（0）

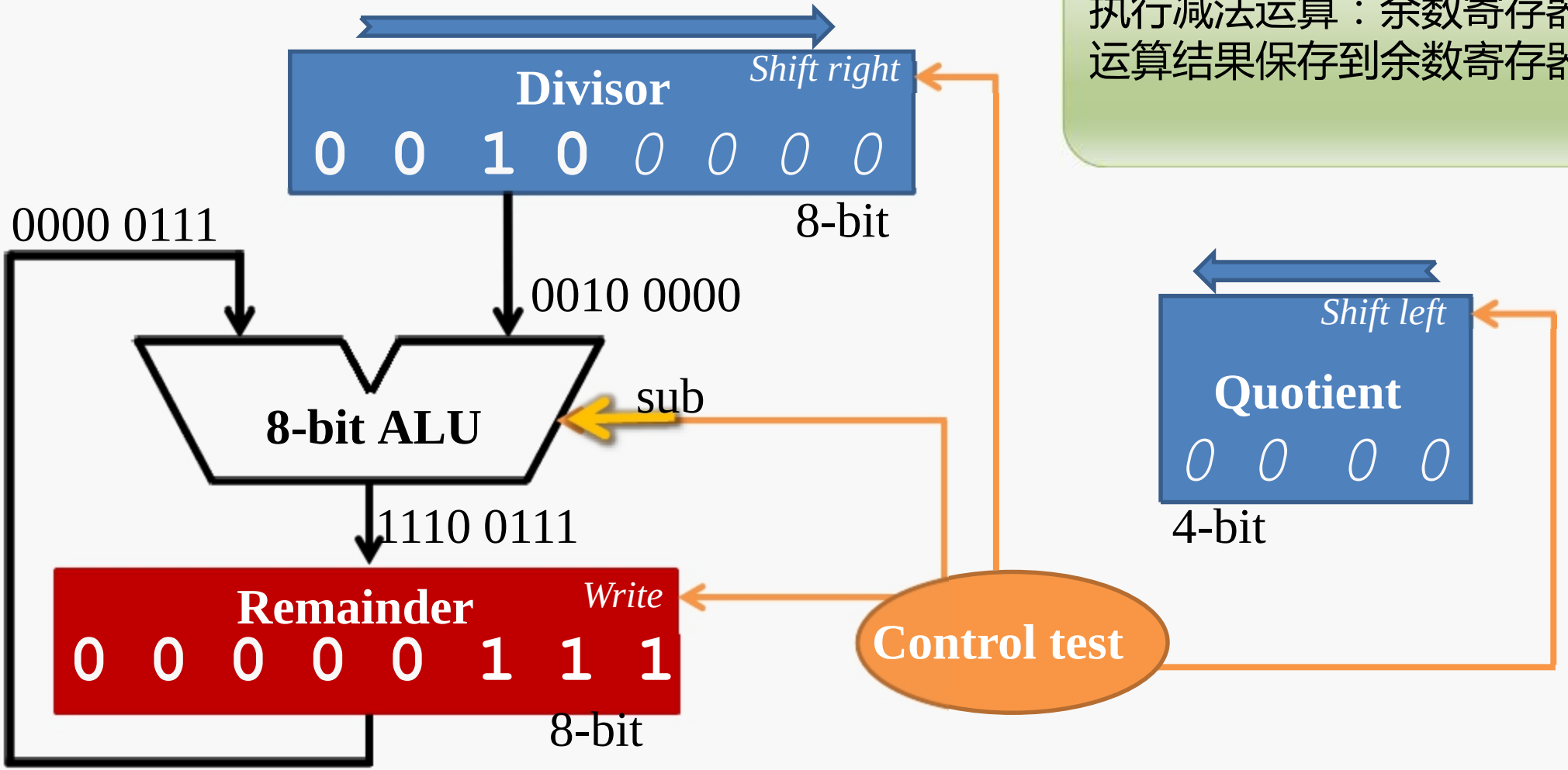
初始化：
a. 将8-bit被除数放入“余数”寄存器；
b. 将4-bit除数放入“除数”寄存器的高4-bit；
c. 将4-bit“商”寄存器置为零。



$0010_{two} \overline{)\ 00000111_{two}}$

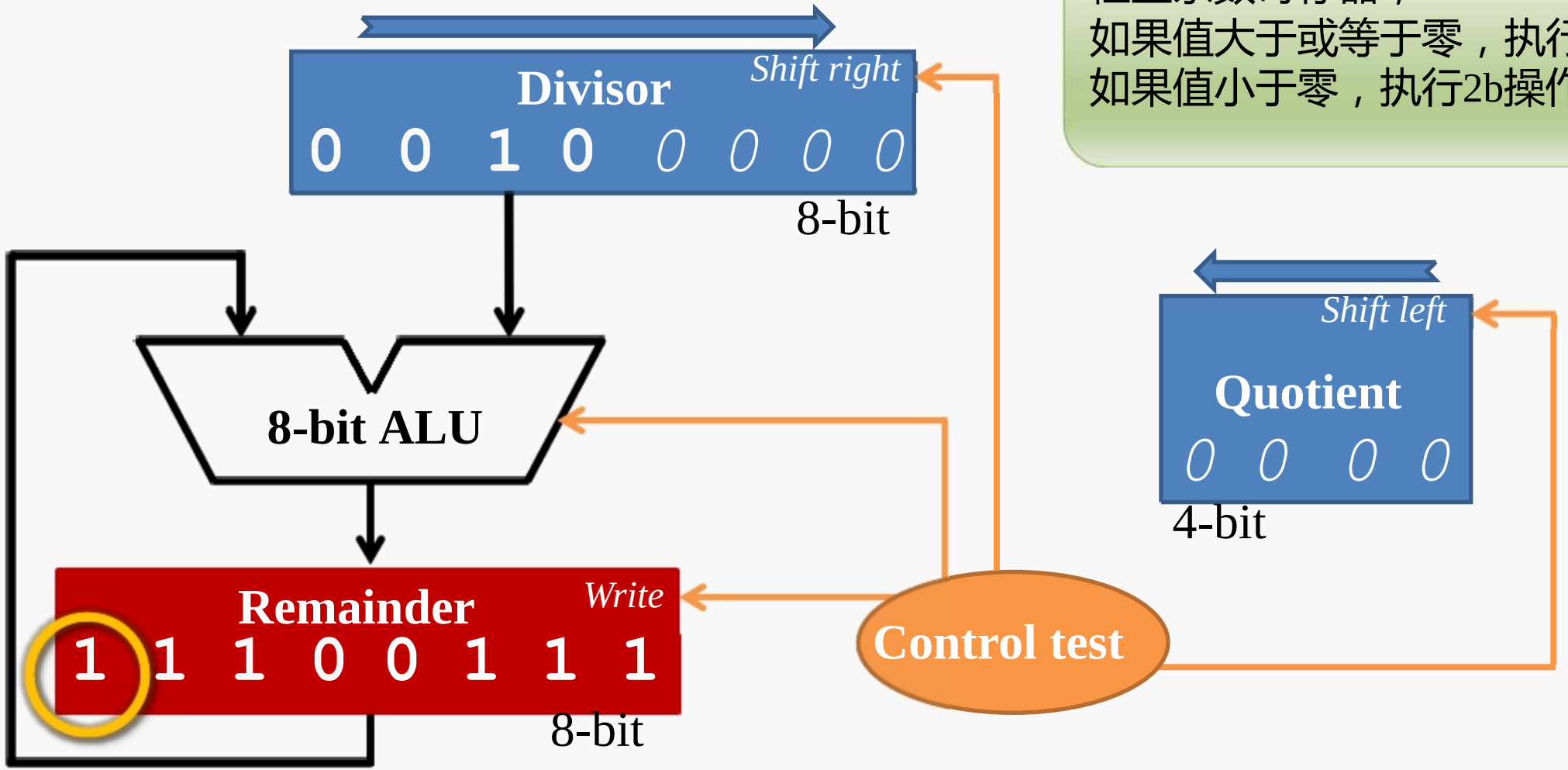
除法器的工作过程（1）

执行减法运算：余数寄存器 – 除数寄存器
运算结果保存到余数寄存器中



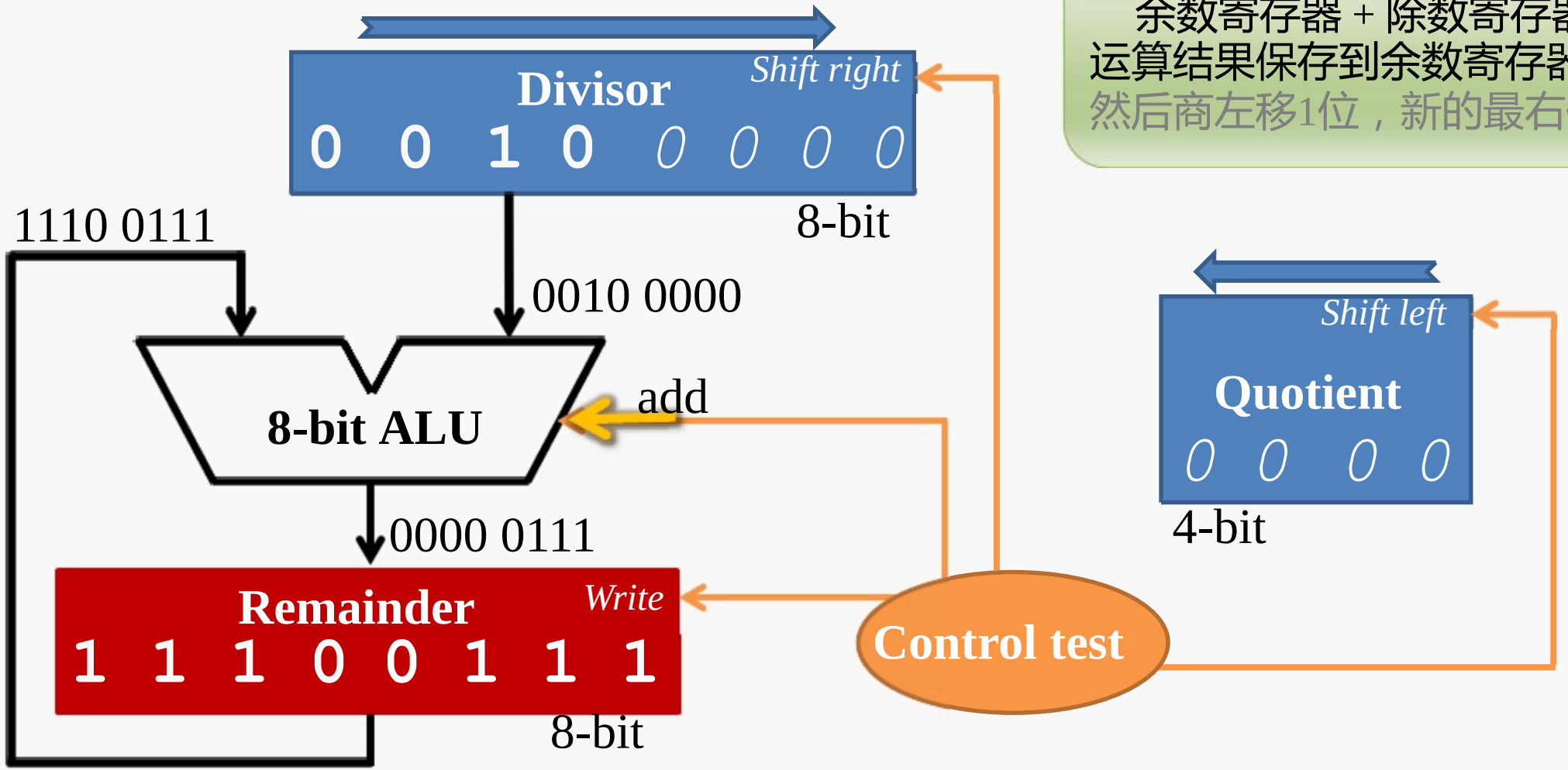
除法器的工作过程（2）

检查余数寄存器，
如果值大于或等于零，执行2a操作；
如果值小于零，执行2b操作。



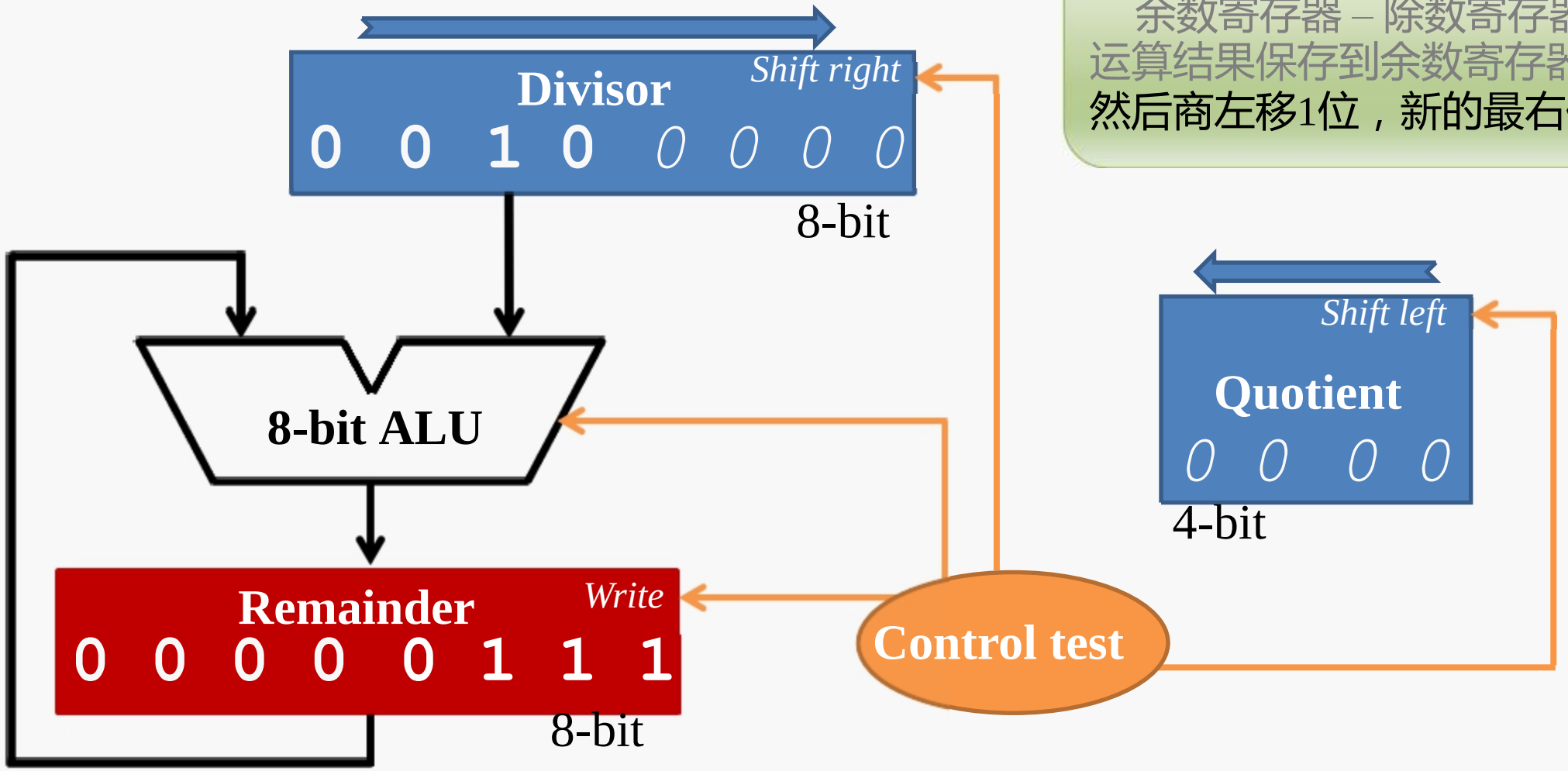
除法器的工作过程（2a）

首先回退第1步的操作，即执行加法运算：
余数寄存器 + 除数寄存器
运算结果保存到余数寄存器；
然后商左移1位，新的最右位设为0

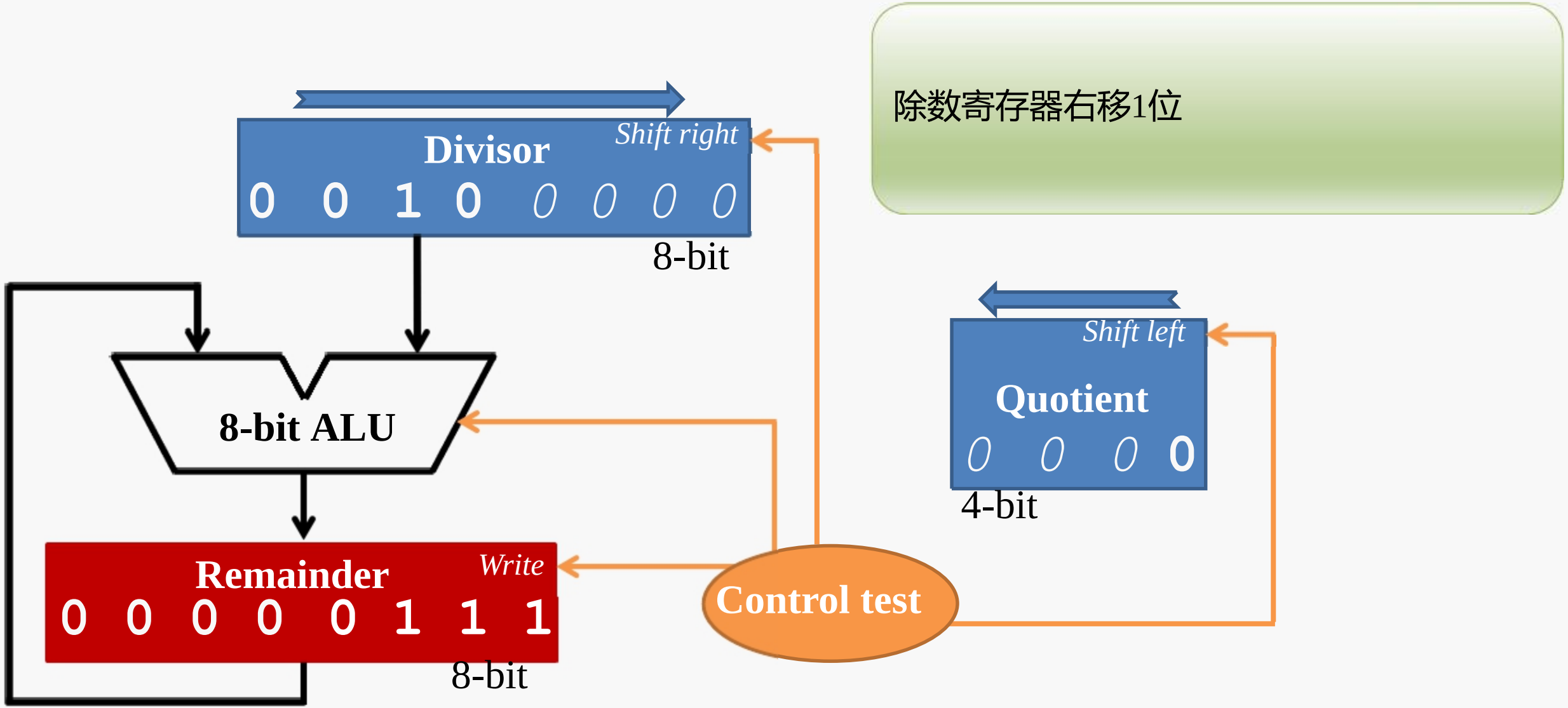


除法器的工作过程（2b）

首先回退第1步的操作，即执行加法运算：
余数寄存器 – 除数寄存器
运算结果保存到余数寄存器；
然后商左移1位，新的最右位设为0

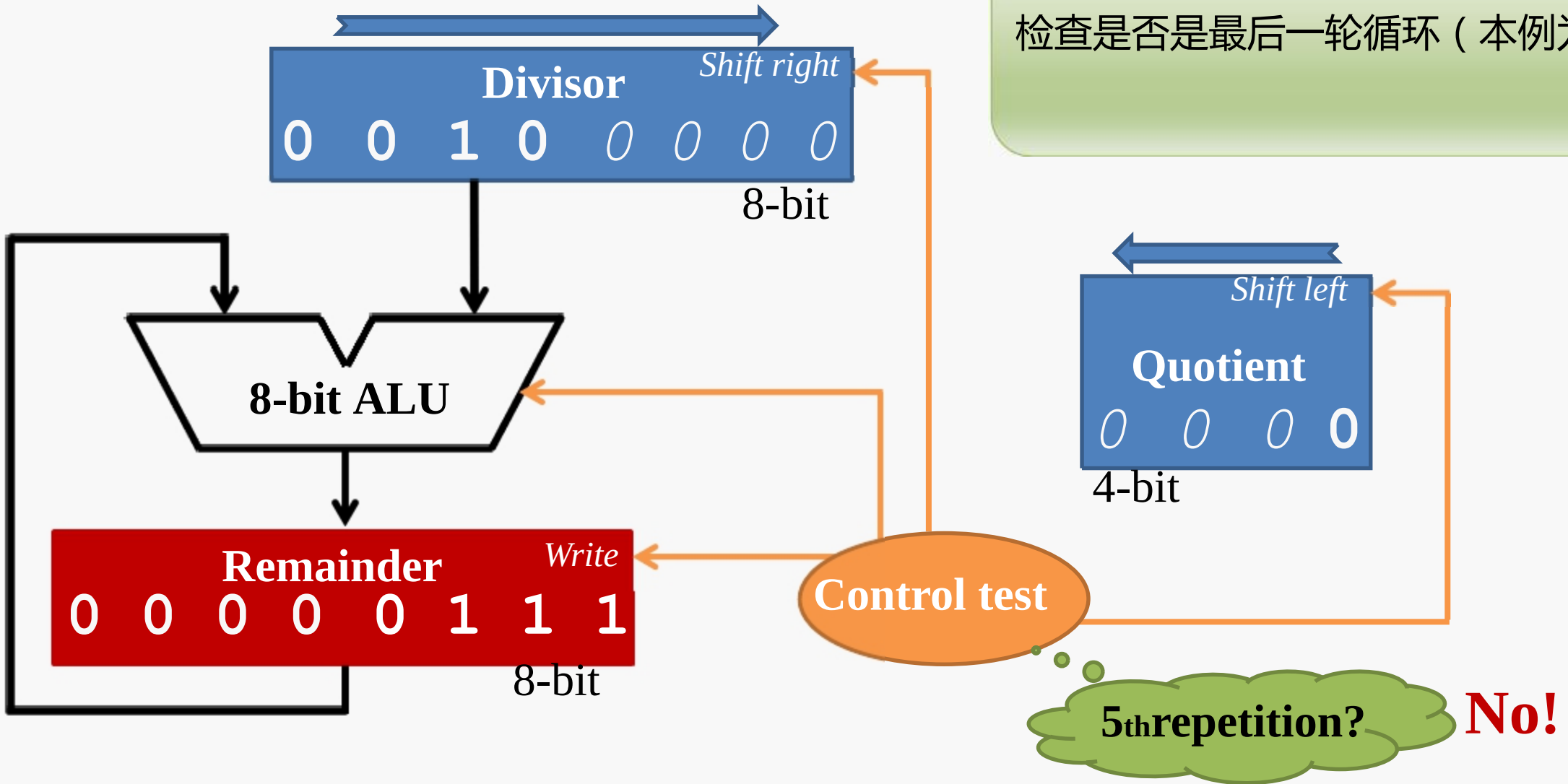


除法器的工作过程（3）


除数寄存器右移1位
Divisor
Shift right
0 0 1 0 0 0 0 0
8-bit
8-bit ALU
Shift left
Quotient
0 0 0 0
4-bit
Remainder
Write
0 0 0 0 0 1 1 1
8-bit
Control test

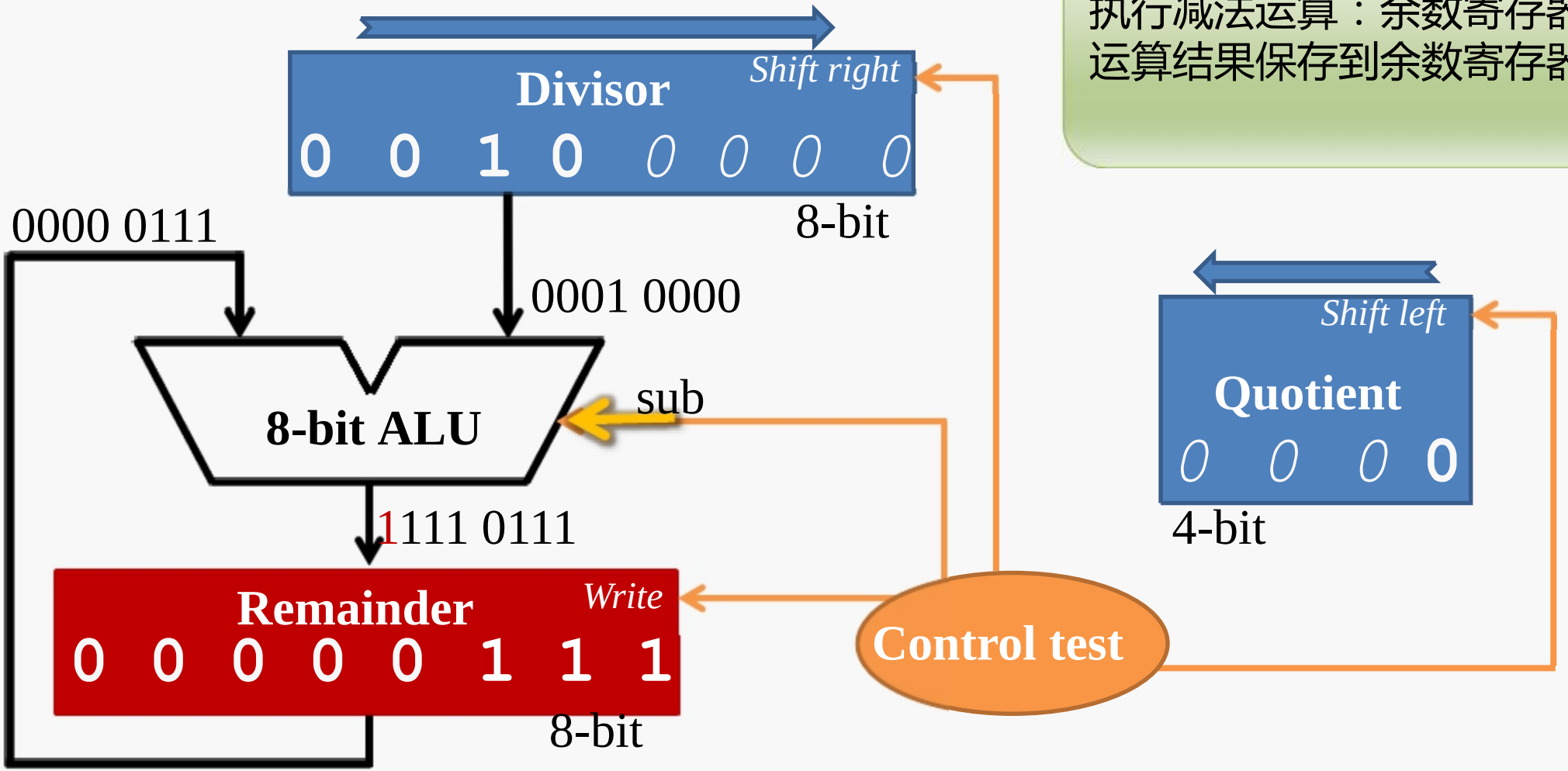
除法器的工作过程（4）

检查是否是最后一轮循环（本例为第五轮）



除法器的工作过程（1）第二轮

执行减法运算：余数寄存器 – 除数寄存器
运算结果保存到余数寄存器中

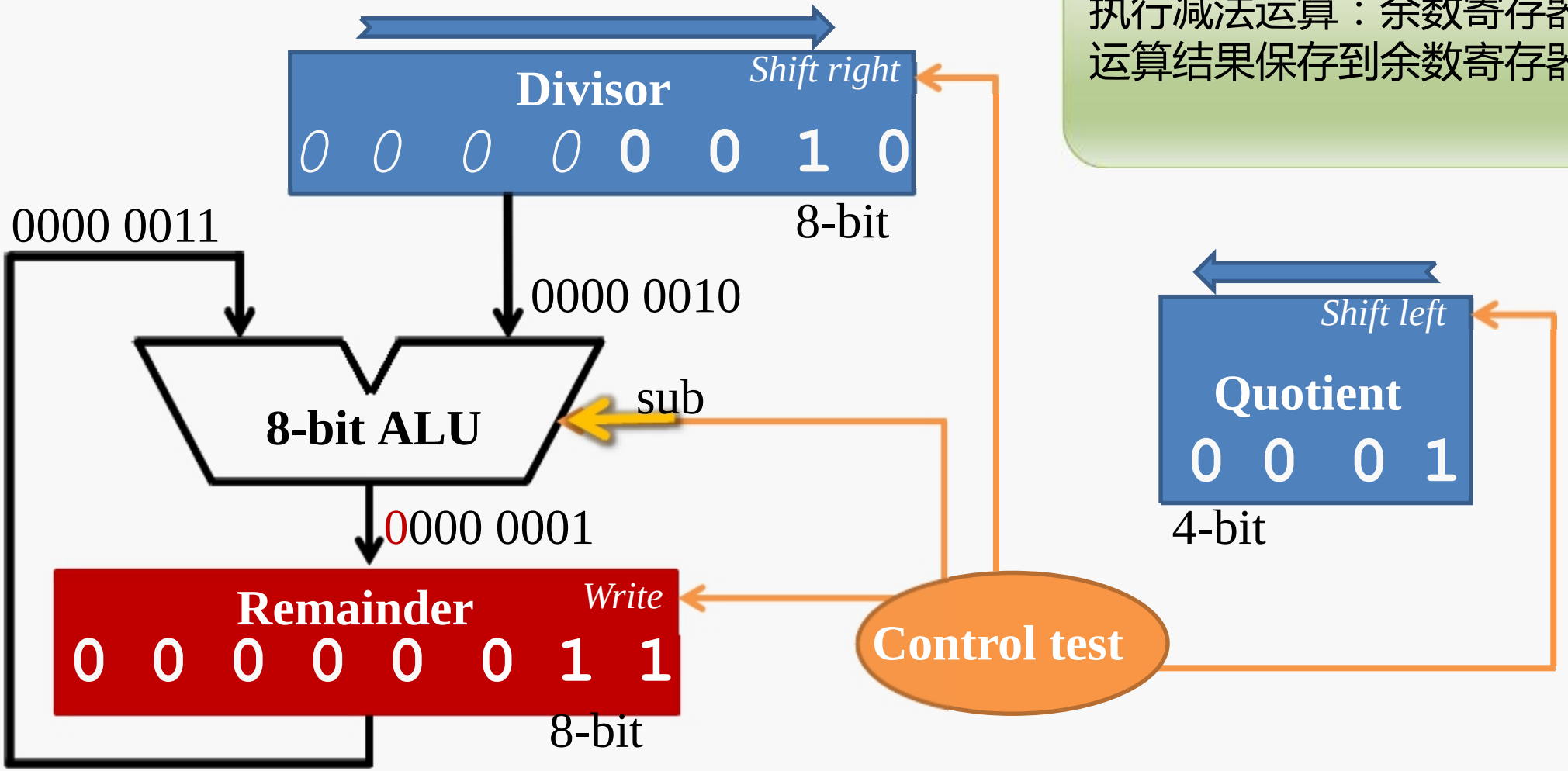


除法器的工作过程（第二轮 ~ 第四轮）

轮次	操作	商	除数	余数
二	1. 余数=余数-除数	0000	0001 0000	1111 0111
	2b. 余数=余数+除数，商左移补0	0000	0001 0000	0000 0111
	3. 除数右移	0000	0000 1000	0000 0111
三	1. 余数=余数-除数	0000	0000 1000	1111 1111
	2b. 余数=余数+除数，商左移补0	0000	0000 1000	0000 0111
	3. 除数右移	0000	0000 0100	0000 0111
四	1. 余数=余数-除数	0000	0000 0100	0000 0011
	2a. 商左移补1	0001	0000 0100	0000 0011
	3. 除数右移	0001	0000 0010	0000 0011

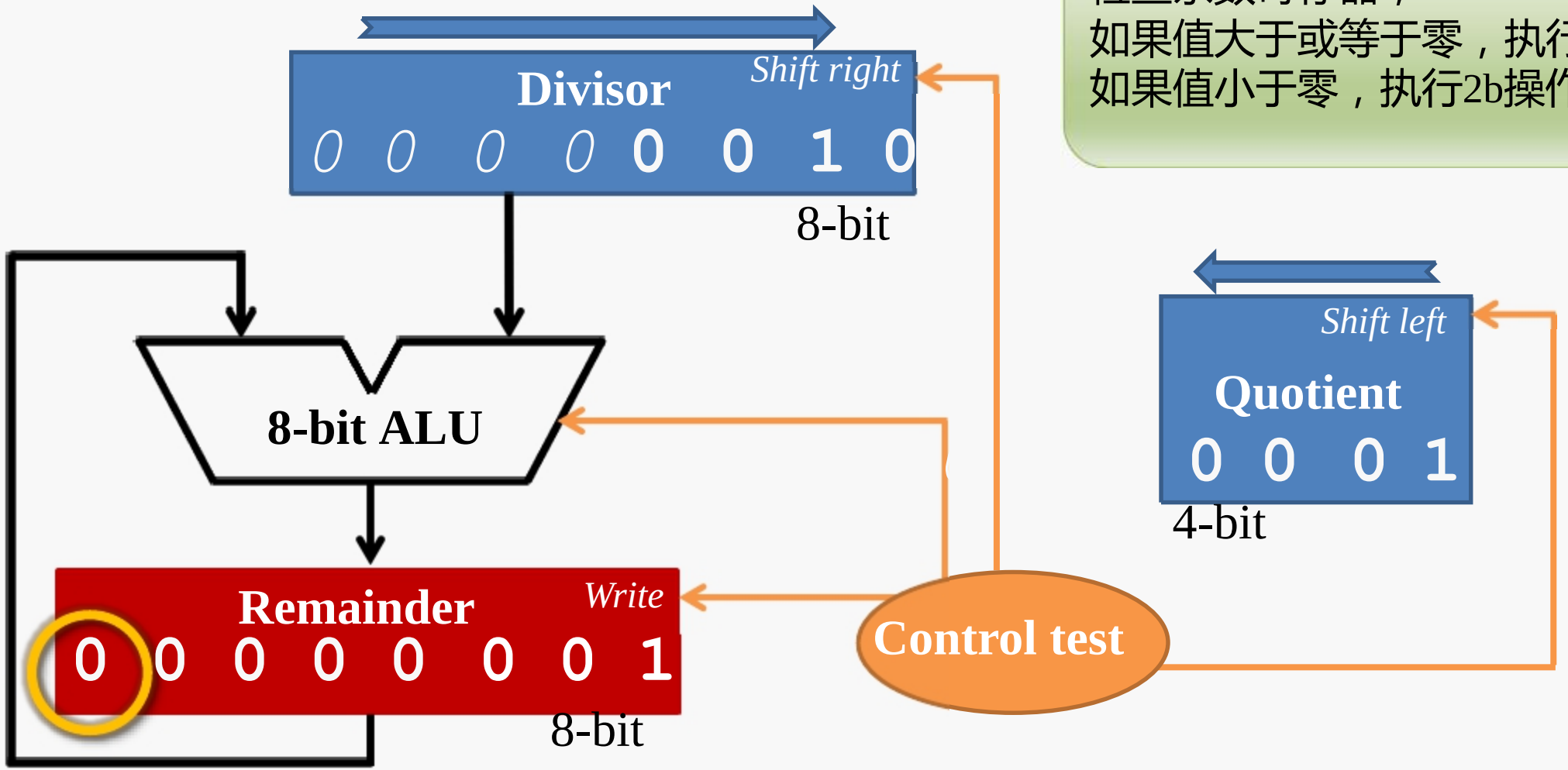
除法器的工作过程（1）第五轮

执行减法运算：余数寄存器 – 除数寄存器
运算结果保存到余数寄存器中



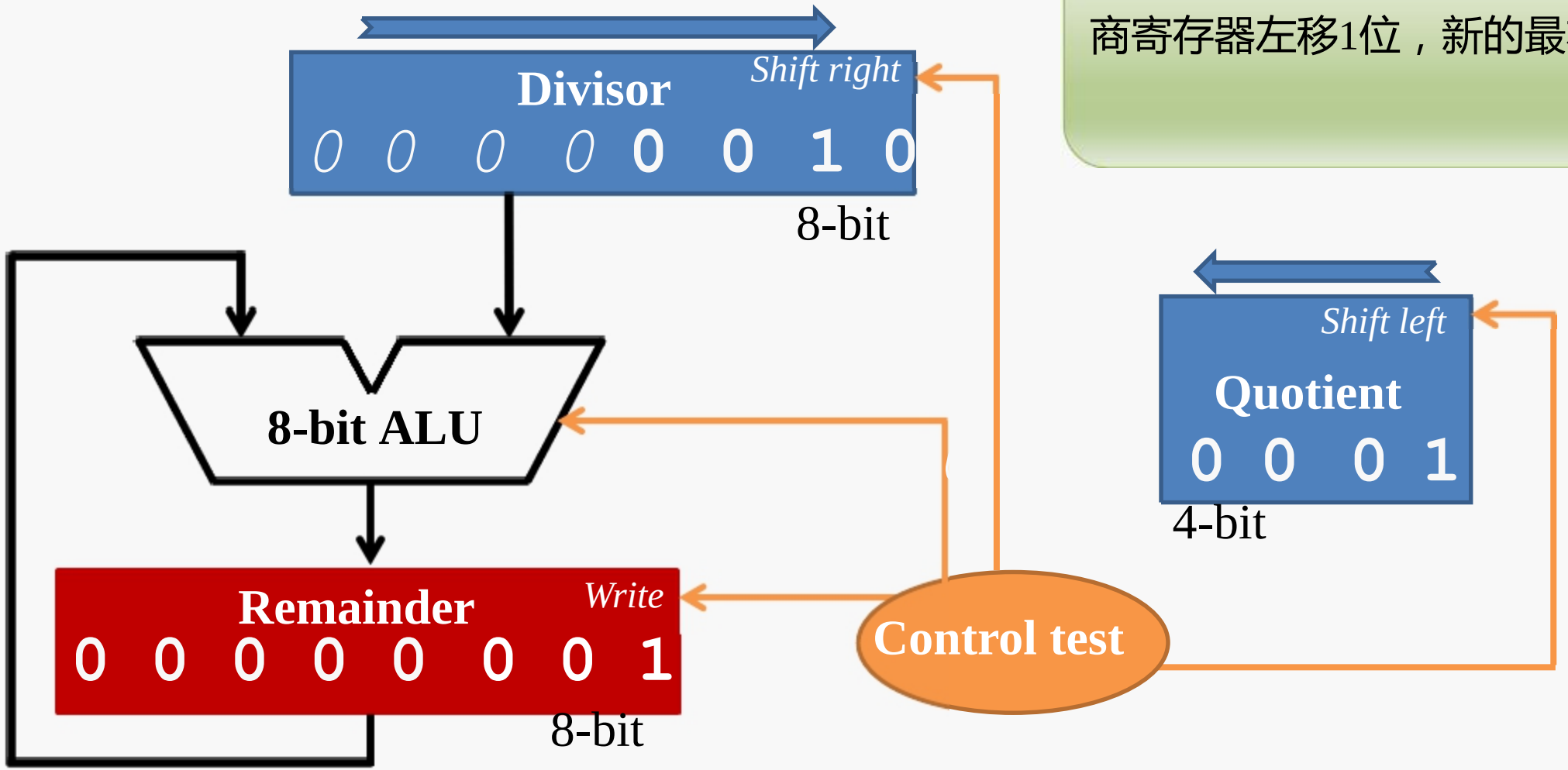
除法器的工作过程（2）第五轮

检查余数寄存器，
如果值大于或等于零，执行2a操作；
如果值小于零，执行2b操作。

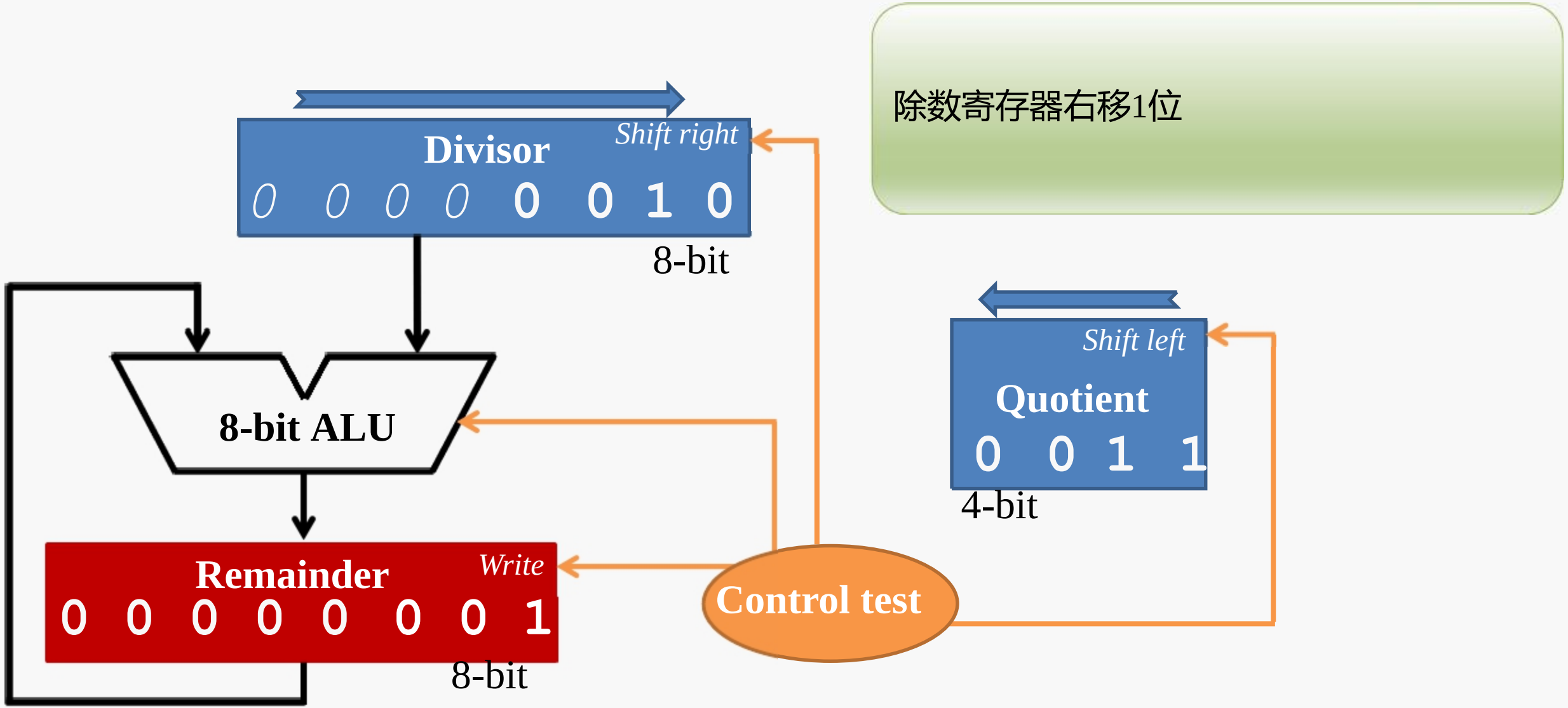


除法器的工作过程（2a）第五轮

商寄存器左移1位，新的最右位设为1

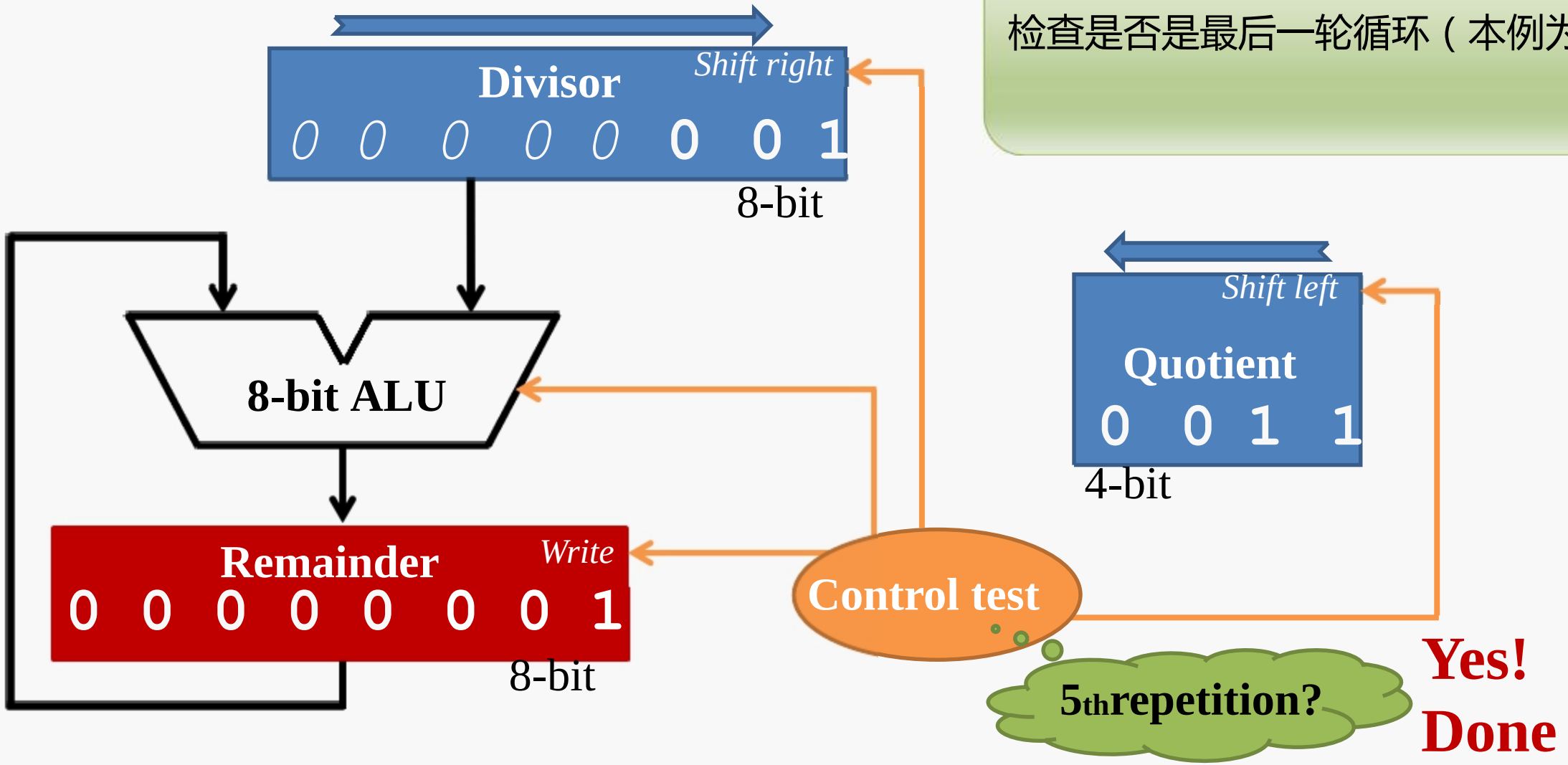


除法器的工作过程（3）第五轮

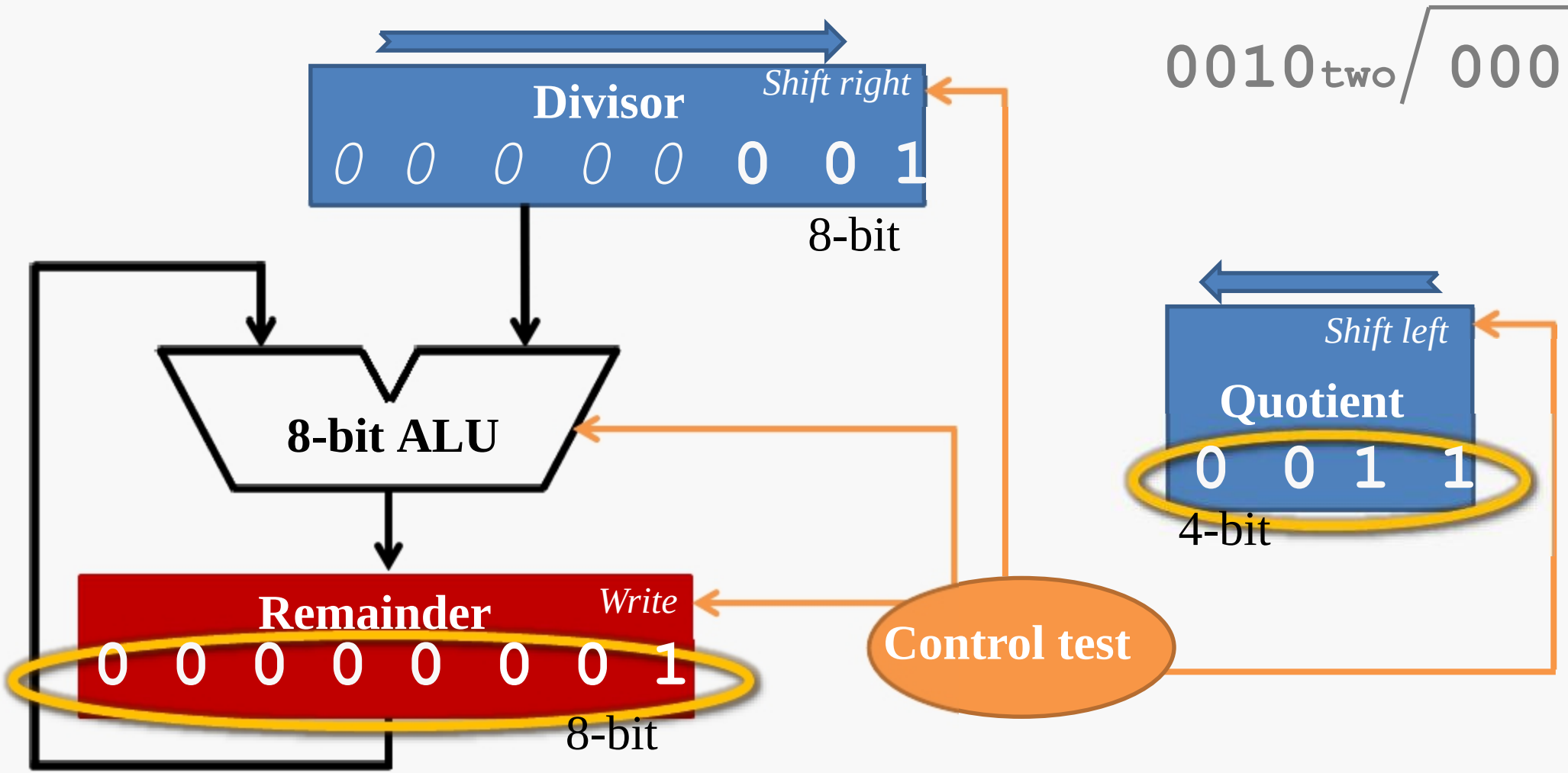


除法器的工作过程（4）第五轮

检查是否是最后一轮循环（本例为第五轮）



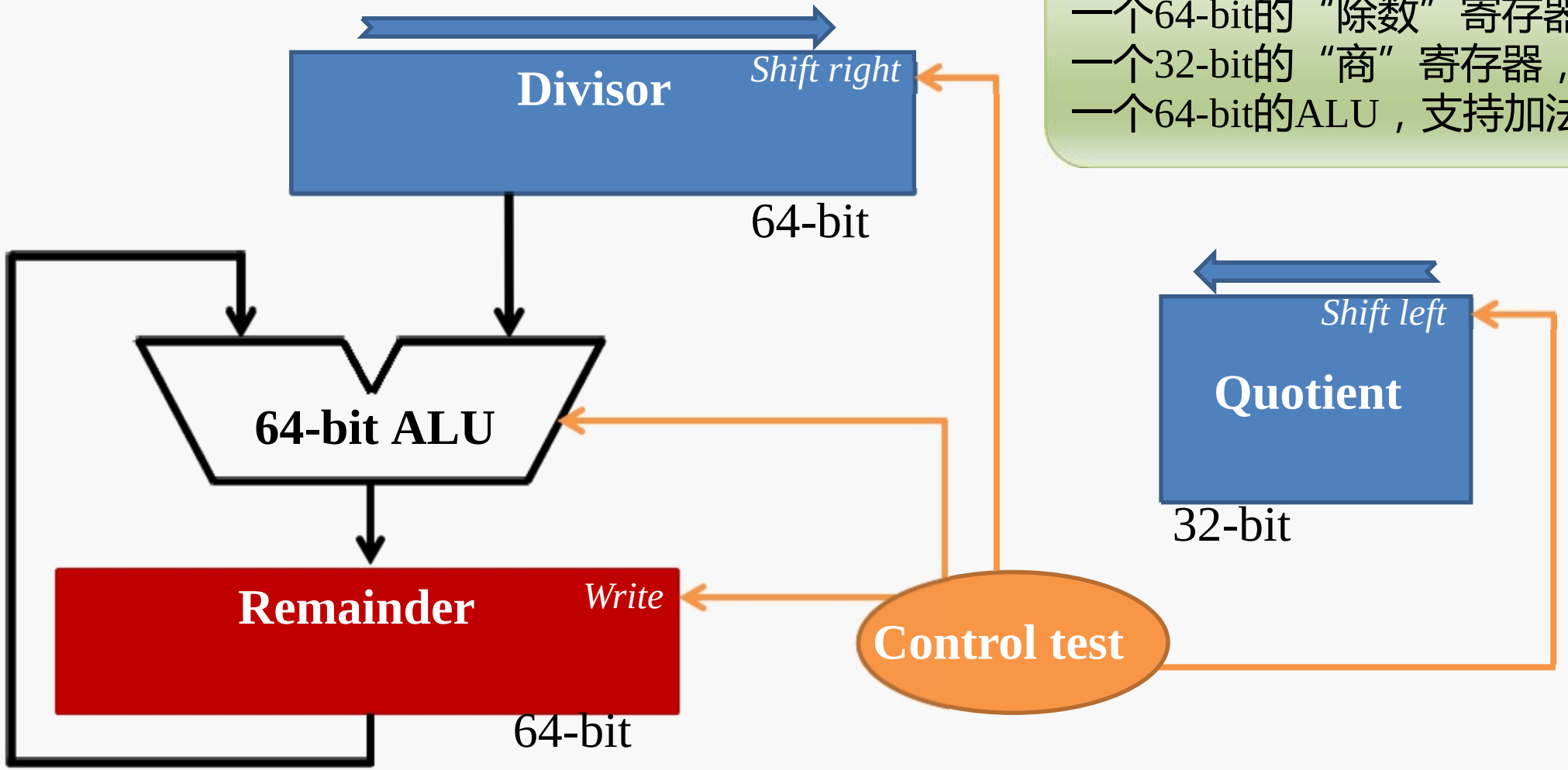
除法器的运算结果



$$0010_{two} \overline{)\ 00000111_{two}}$$

32-bit 除法器的实现

一个64-bit的“余数”寄存器
一个64-bit的“除数”寄存器，带右移功能
一个32-bit的“商”寄存器，带左移功能
一个64-bit的ALU，支持加法和减法运算



第四章　乘法器和除法器

1.乘法的运算过程

2.乘法器的实现

3.乘法器的优化

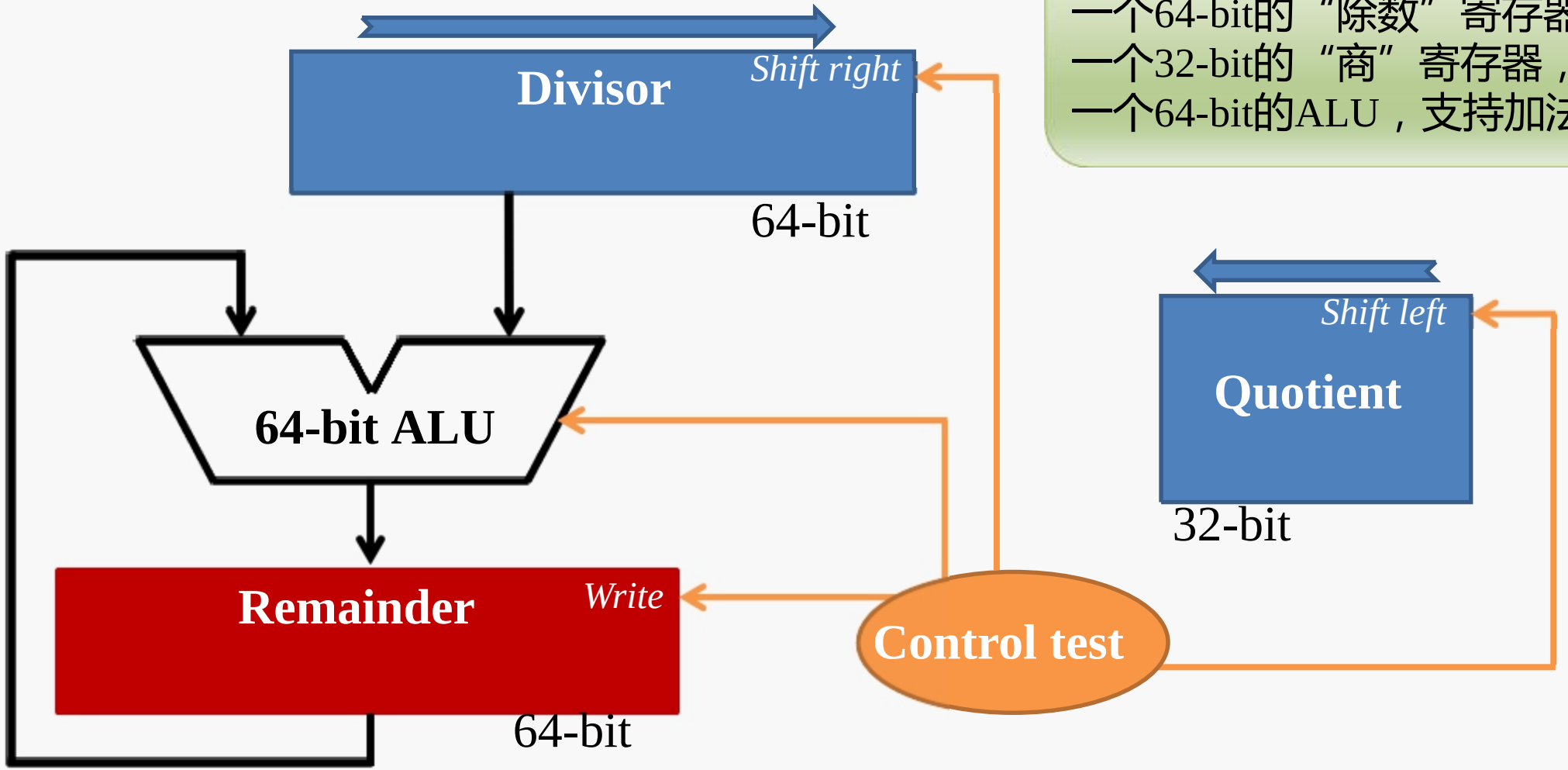
4.除法的运算过程

5.除法器的实现

6.除法器的优化

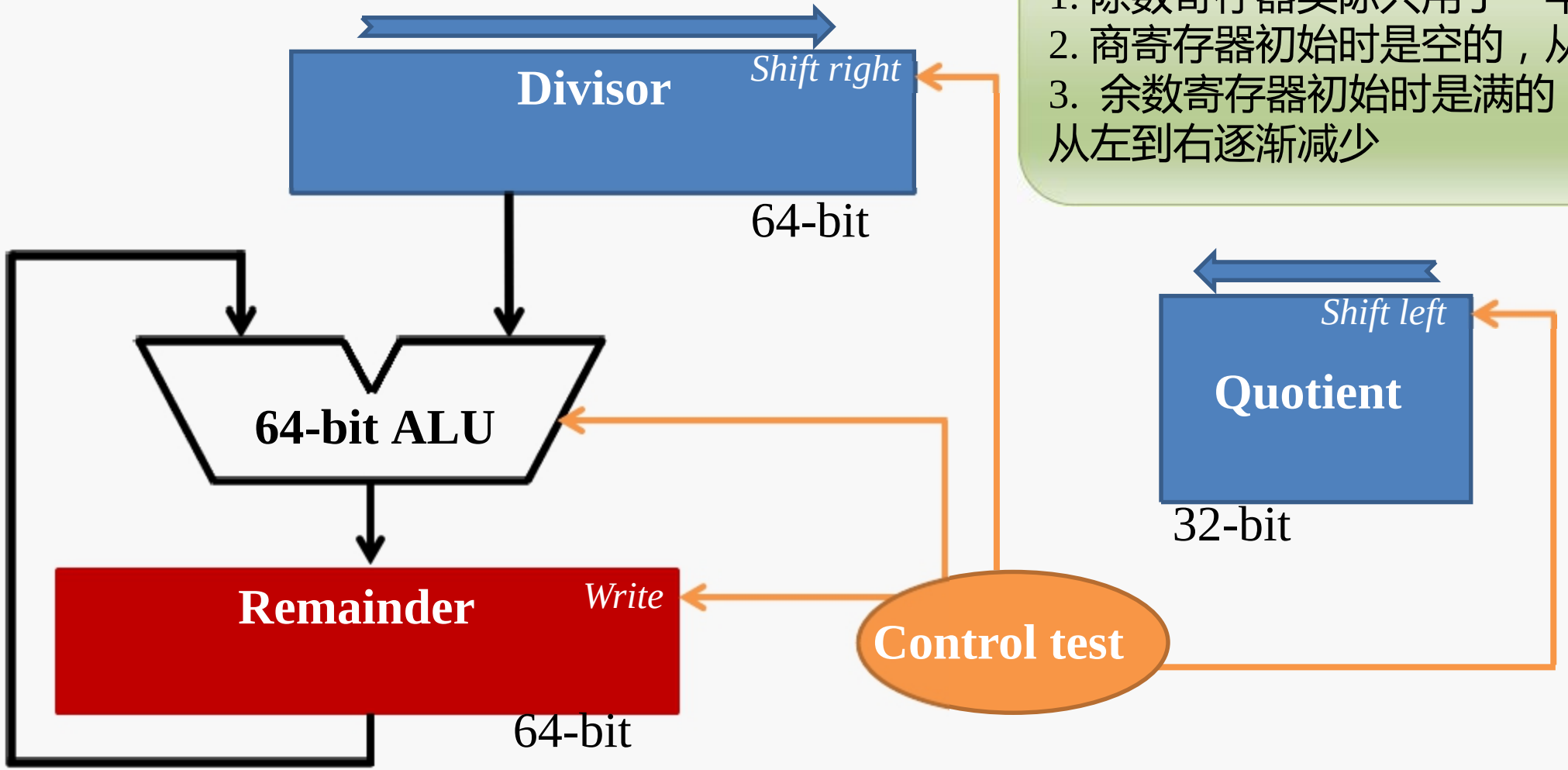
除法器的实现（第一版）

一个64-bit的“余数”寄存器
一个64-bit的“除数”寄存器，带右移功能
一个32-bit的“商”寄存器，带左移功能
一个64-bit的ALU，支持加法和减法运算

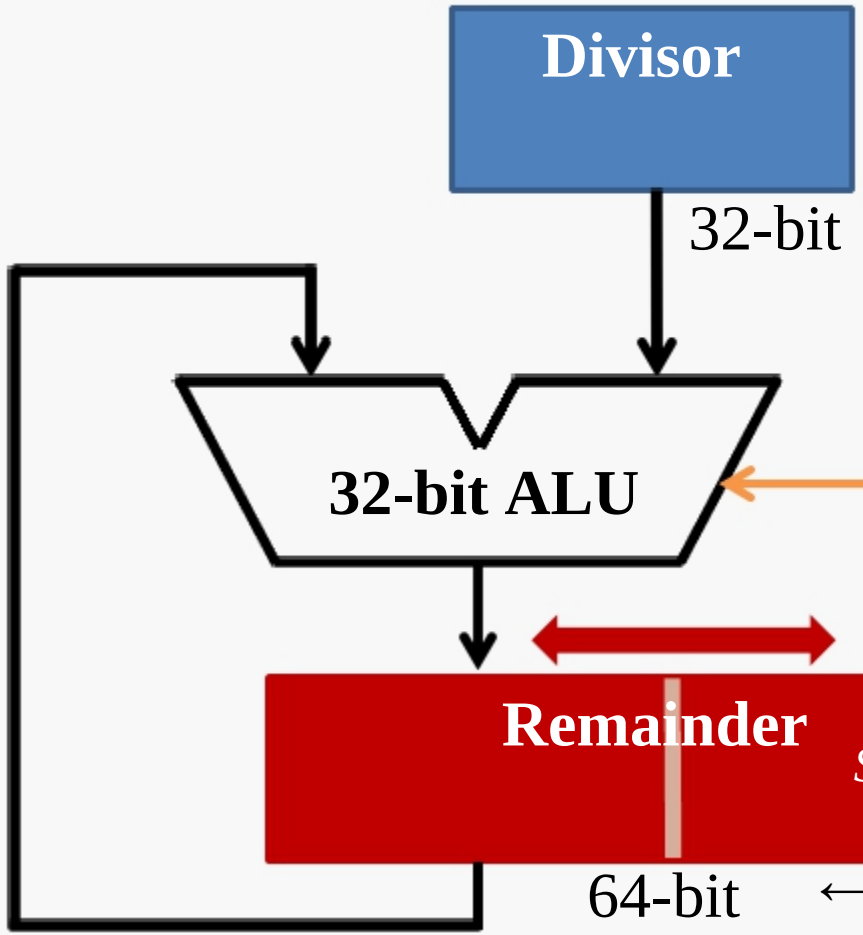


除法器的面积优化

从减小面积的角度进行分析：
1. 除数寄存器实际只用了一半
2. 商寄存器初始时是空的，从右到左逐位填满
3. 余数寄存器初始时是满的，有实际意义的位从左到右逐渐减少



除法器的实现（第二版）



优化方案
1. 除数寄存器缩小为32-bit，无需支持移位
2. 取消商寄存器
3. 64-bit ALU缩小为32-bit
4. 余数寄存器只有高32-bit参与加减法运算
5. 余数寄存器需支持左移和右移
6. 商从右端逐位移入余数寄存器
7. 运算结束时，商占据余数寄存器的低32-bit

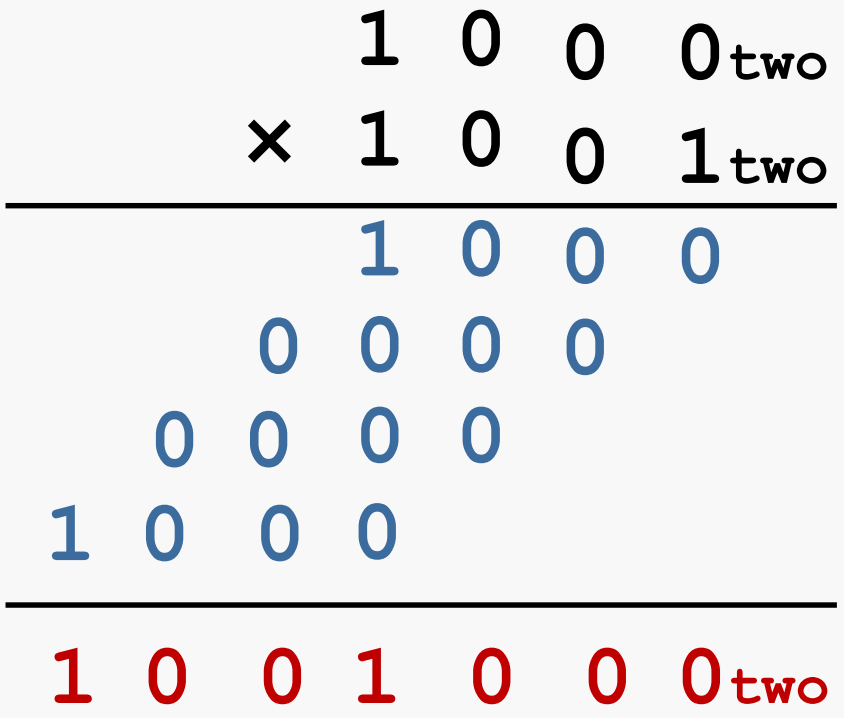
Control test

原先的结构：
一个64-bit的“余数”寄存器
一个64-bit的“除数”寄存器，带右移功能
一个32-bit的“商”寄存器，带左移功能
一个64-bit的ALU，支持加法和减法运算

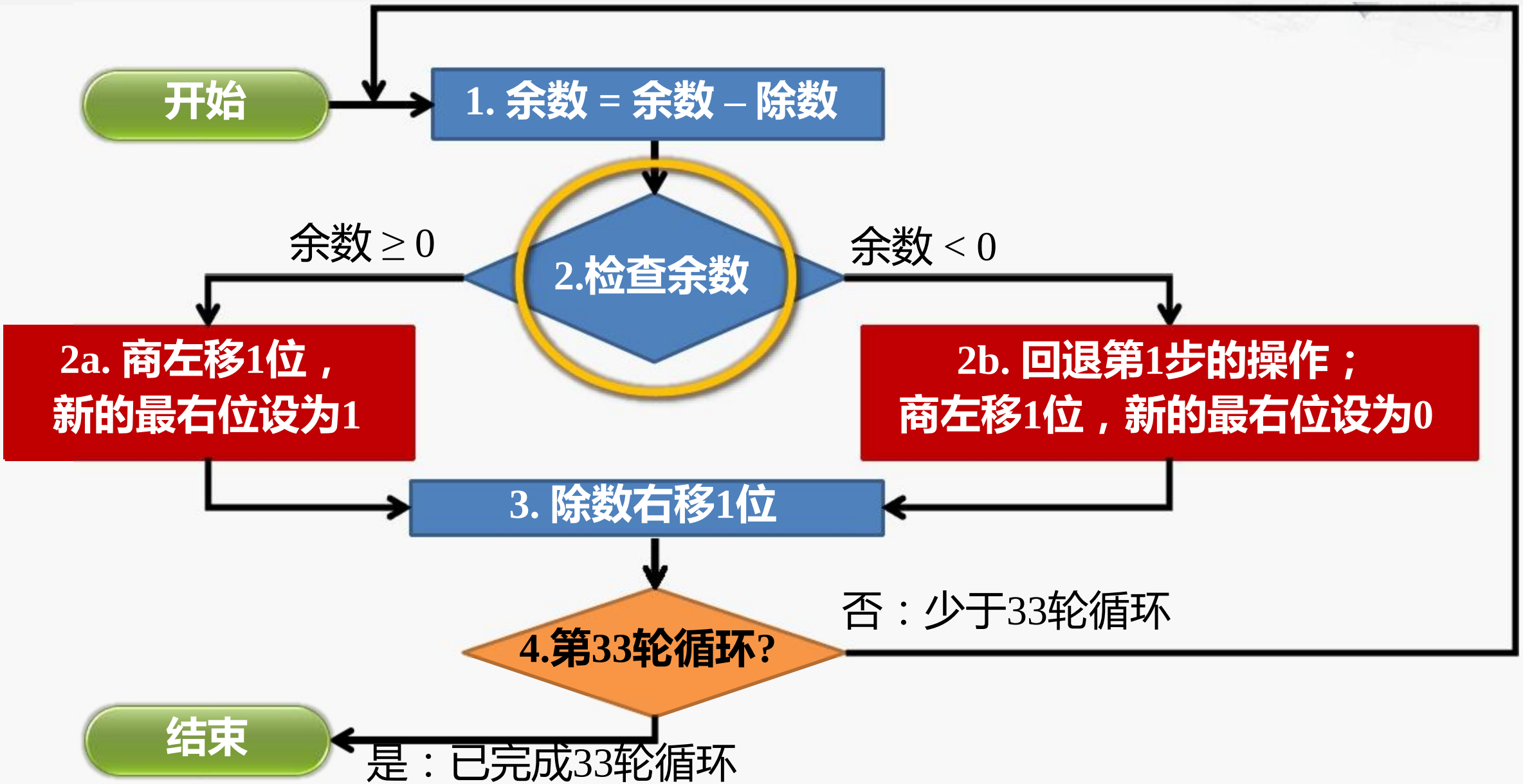
除法的性能优化分析

回顾：乘法的特点

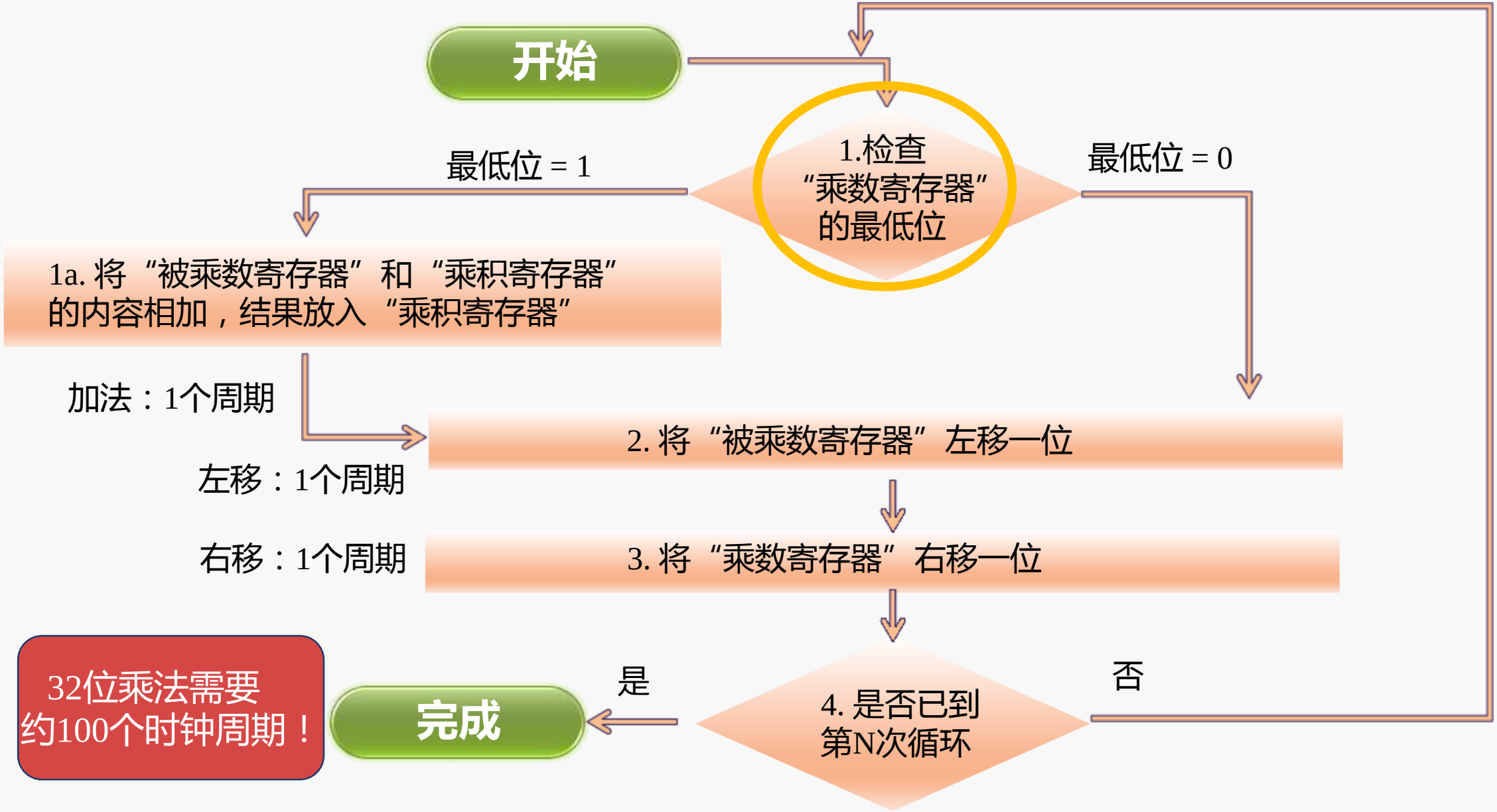
- 每个部分积都是独立的
- 可以并行计算各个部分积



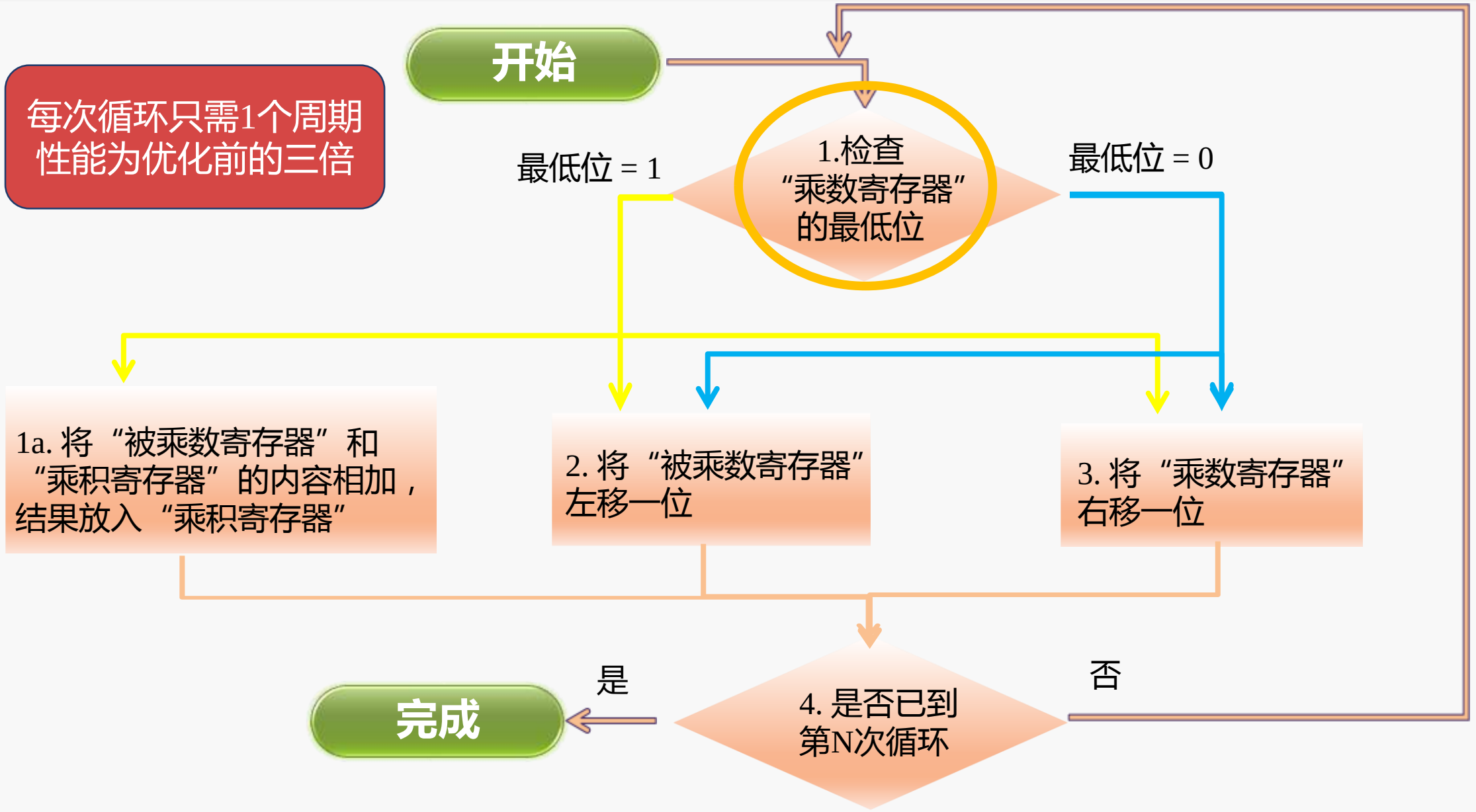
除法的性能优化分析


开始
1. 余数 = 余数 – 除数
2.检查余数
余数 ≥ 0
余数 < 0
2a. 商左移1位，
新的最右位设为1
2b. 回退第1步的操作；
商左移1位，新的最右位设为0
3. 除数右移1位
4.第33轮循环?
否：少于33轮循环
结束
是：已完成33轮循环

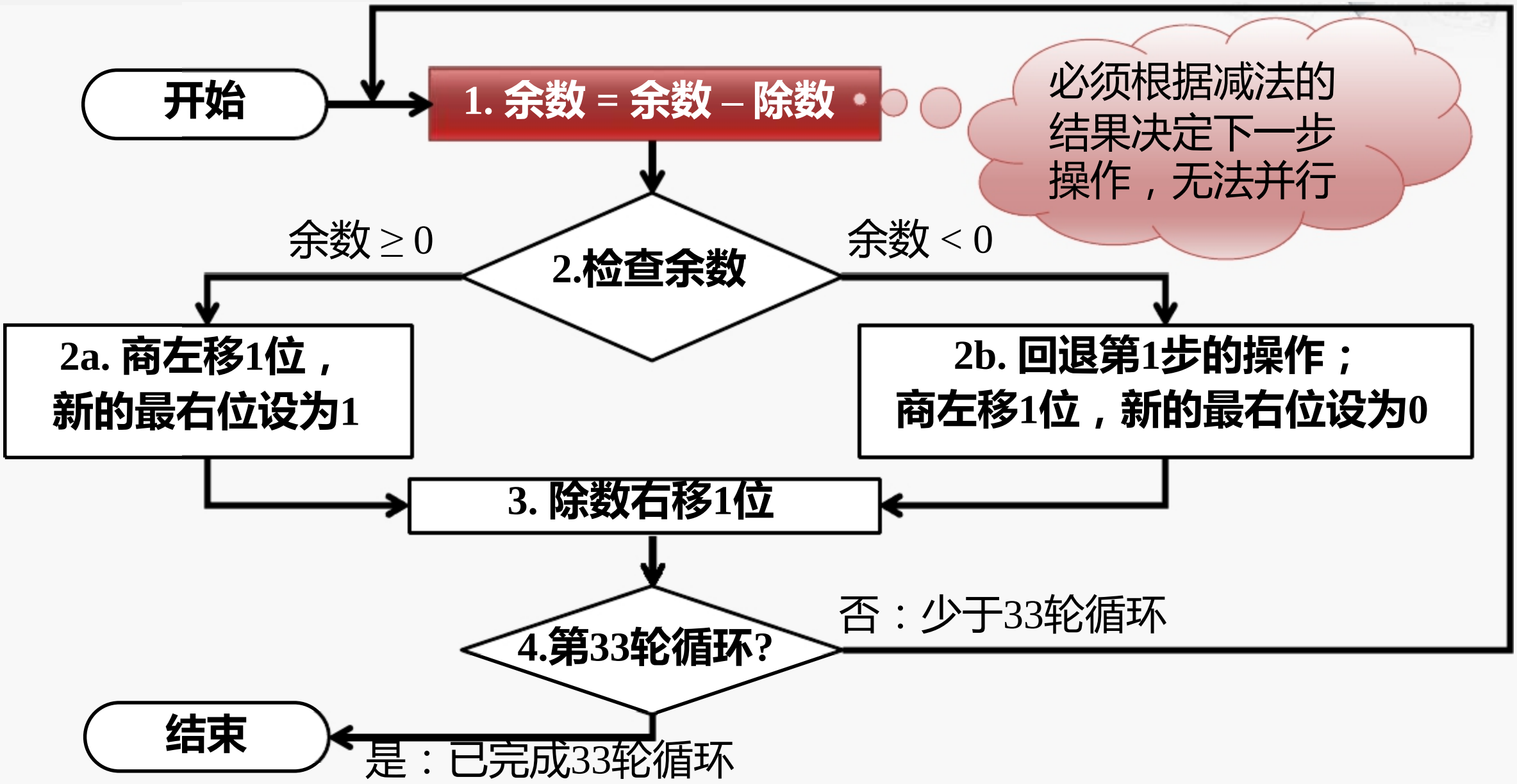
对比：N位乘法器的工作流程图



对比：N位乘法器的工作流程（优化后）



除法的性能优化分析


开始
1. 余数 = 余数 – 除数
必须根据减法的结果决定下一步操作，无法并行
2.检查余数
余数 ≥ 0
余数 < 0
2a. 商左移1位，新的最右位设为1
2b. 回退第1步的操作；商左移1位，新的最右位设为0
3. 除数右移1位
4.第33轮循环?
否：少于33轮循环
结束
是：已完成33轮循环

第五章　单周期处理器

1. 处理器设计的 主要步骤
2. 数据通路的建立
3.运算指令的 控制信号
4.访存指令的 控制信号
5.分支指令的 控制信号
6.控制信号的集成

处理器的设计步骤

① 分析指令系统，得出对数据通路的需求

② 为数据通路选择合适的组件

③ 连接组件建立数据通路

④ 分析每条指令的实现，以确定控制信号

⑤ 集成控制信号，形成完整的控制逻辑

MIPS指令系统的简化版本

无符号加法和减法

- addu rd,rs,rt
- subu rd,rs,rt

立即数的逻辑或

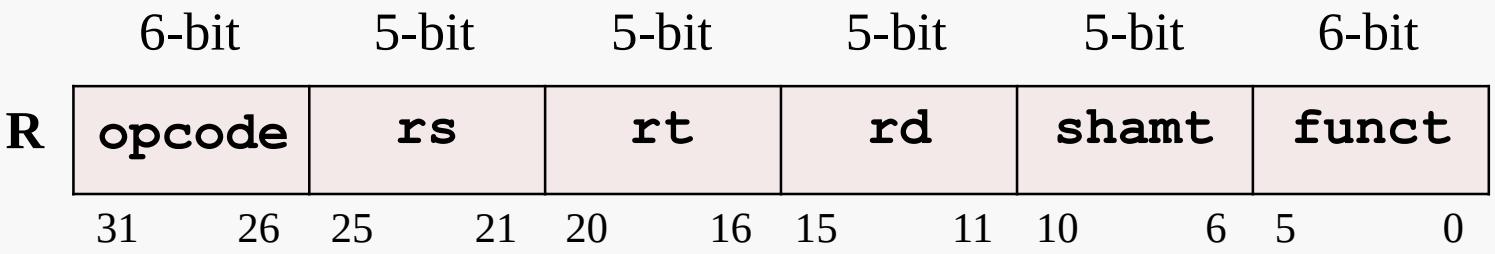
- ori rt,rs,imm16

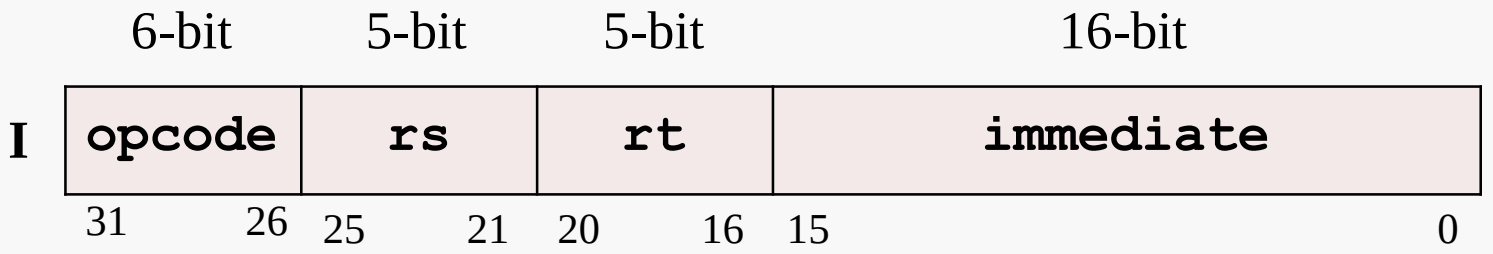
装载和存储一个字（32位）

- lw rt,imm16(rs)
- sw rt,imm16(rs)

条件分支

- beq rs,rt,imm16



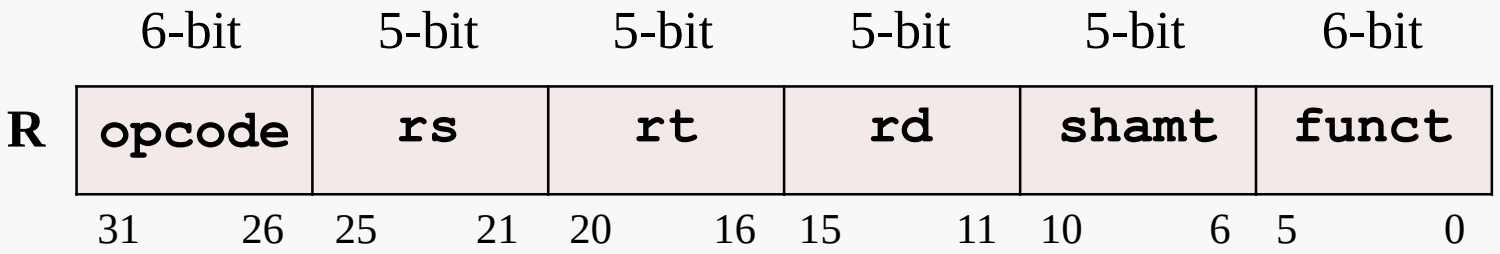


指令的含义

需求：存放指令的存储器，可读，地址和数据均为32位

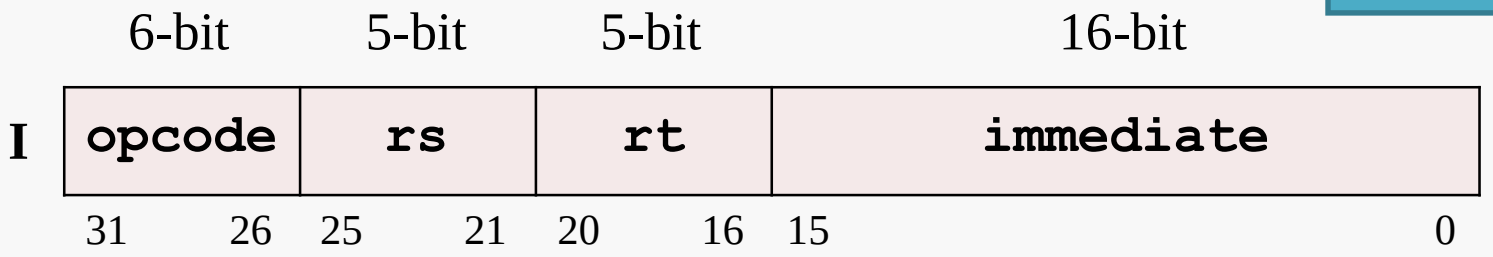
指令位域的分解

- R型指令：{op, rs, rt, rd, shamt, funct} ← MEM[PC]



- I型指令：{op, rs, rt, Imm16} ← MEM[PC]

需求：存放指令地址的32位寄存器



指令的含义

指令的操作

- ADDU　　R[rd]← R[rs] + R[rt]; PC← PC+4
- SUBU　　R[rd]← R[rs] – R[rt]; PC← PC+4

需求：改写一个寄存器的内容（rd或rt）

需求：一组存放数据的32位通用寄存器

需求：同时读取两个寄存器的内容（rs和rt）

- ORI　　R[rt]← R[rs] | zero_ext(Imm16); PC← PC+4

需求：运算的操作数可以是 寄存器 或者扩展后的立即数

需求：提供加、减、逻辑或 三种功能的运算器

需求：将16位立即数扩展到32位（零扩展）

指令的含义

指令的操作

- LOAD R[rt]← MEM[R[rs]+sign_ext(Imm16)]; PC← PC+4
- STORE MEM[R[rs]+sign_ext(Imm16)]← R[rt]; PC← PC+4

需求：存放数据的存储器，可读写，地址和数据均为32位

需求：将16位立即数扩展到32位（符号扩展）

- BEQ if (R[rs]==R[rt])
  then PC← PC+4 + (sign_ext(Imm16)||00)
  else PC← PC+4

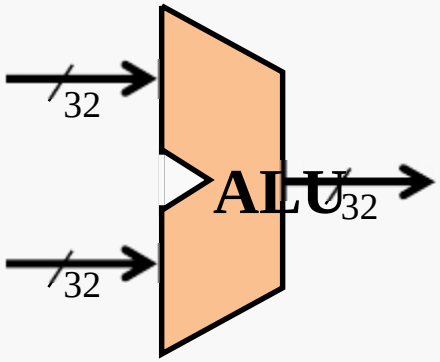
需求：比较两个数，判断是否相等

需求：PC寄存器支持两种自增方式，加4 或 加一个立即数

指令系统的需求

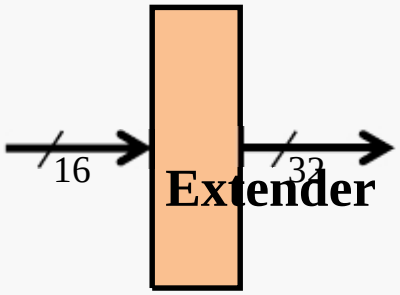
算术逻辑单元（ALU）

- 运算类型：加、减、或、比较相等
- 操作数：2个32位的数，来自寄存器 或 扩展后的立即数



立即数扩展部件

- 将一个16立即数扩展为32位数
- 扩展方式：零扩展、符号扩展



程序计数器（PC）

- 一个32位的寄存器
- 支持两种加法：加4 或 加一个立即数

指令系统的需求

寄存器堆

- 每个寄存器为32位宽，共32个
- 支持读操作：rs 和 rt
- 支持写操作：rt 或 rd
- 注：这称为“两读一写”的寄存器堆

存储器

- 一个只读的指令存储器，地址和数据均为32位
- 一个可读写的数据存储器，地址和数据均为32位
- 注：这两个存储器实际对应了CPU中的指令和数据高速缓存（Cache）

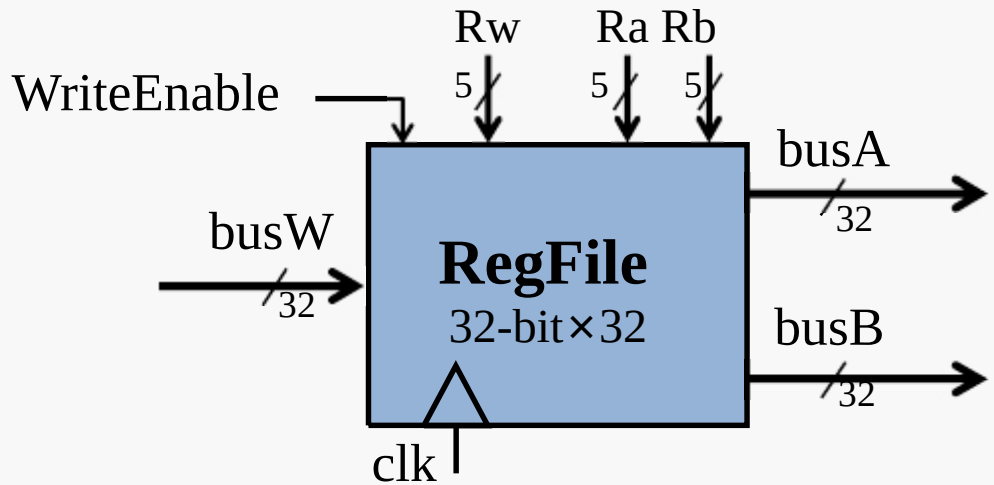
存储组件：寄存器堆

内部构成

- 32个32位的寄存器

数据接口信号

- busA，busB：两组32位的数据输出
- busW：一组32位的数据输入



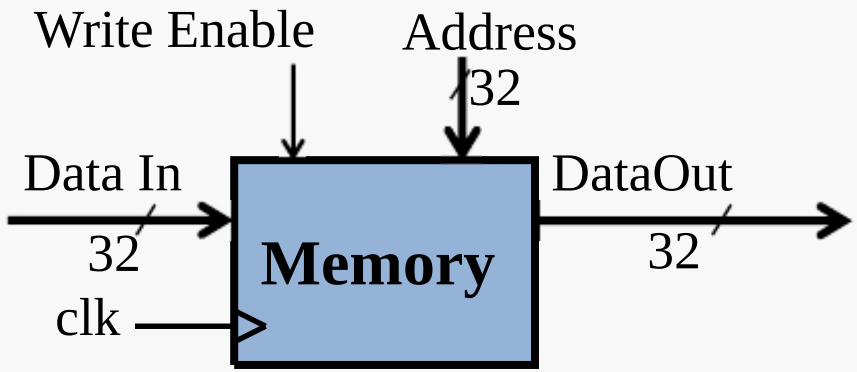
读写控制

- Ra(5位)：选中对应编号的寄存器，将其内容放到busA
- Rb(5位)：选中对应编号的寄存器，将其内容放到busB
- Rw(5位)：选中对应编号的寄存器，在时钟信号（clk）的上升沿，如果写使能信号有效（WriteEnable==1），将busW的内容存入该寄存器
- 注：寄存器堆的读操作不受时钟控制

存储组件：存储器

数据接口信号

- Data In：32位的数据输入信号
- Data Out：32位的数据输出信号



读写控制

- Address：32位的地址信号。该信号指定一个存储单元，将其内容送到数据输出信号
- Write Enable：写使能信号。在时钟信号（clk）的上升沿，如果写使能信号有效（为1），将数据输入信号的内容存入地址信号指定存储单元
- 注：存储器的读操作不受时钟控制

处理器的设计步骤

① 分析指令系统，得出对数据通路的需求 

② 为数据通路选择合适的组件 

③ 连接组件建立数据通路

④ 分析每条指令的实现，以确定控制信号

⑤ 集成控制信号，形成完整的控制逻辑

第五章　单周期处理器

- 1. 处理器设计的 主要步骤
- 2. 数据通路的建立
- 3.运算指令的 控制信号
- 4.访存指令的 控制信号
- 5.分支指令的 控制信号
- 6.控制信号的集成

建立数据通路的方法

基本原则

- 根据指令需求，连接组件，建立数据通路

指令的需求

- 所有指令的共同需求
- 不同指令的不同需求

建立数据通路的方法

基本原则

- 根据指令需求，连接组件，建立数据通路

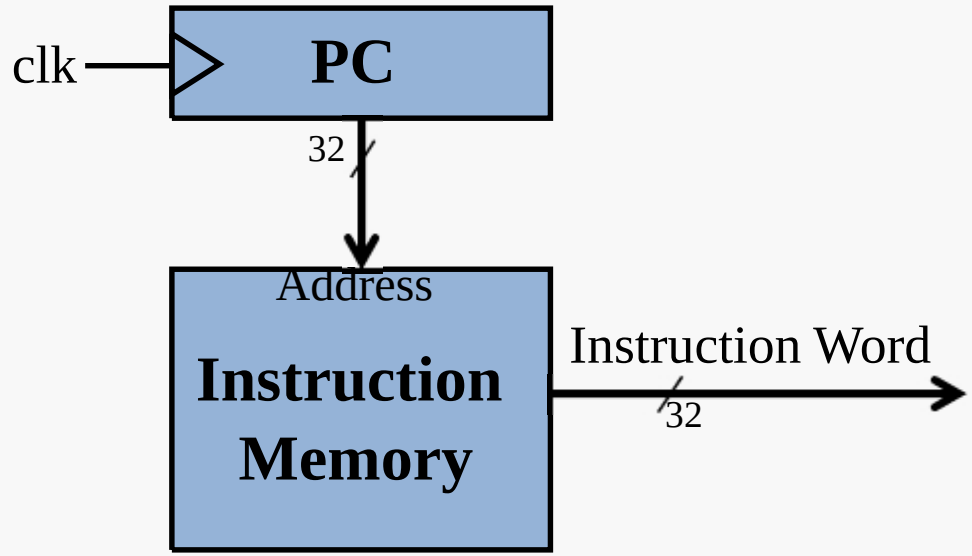
指令的需求

- **所有指令的共同需求**
- 不同指令的不同需求

所有指令的共同需求

取指令

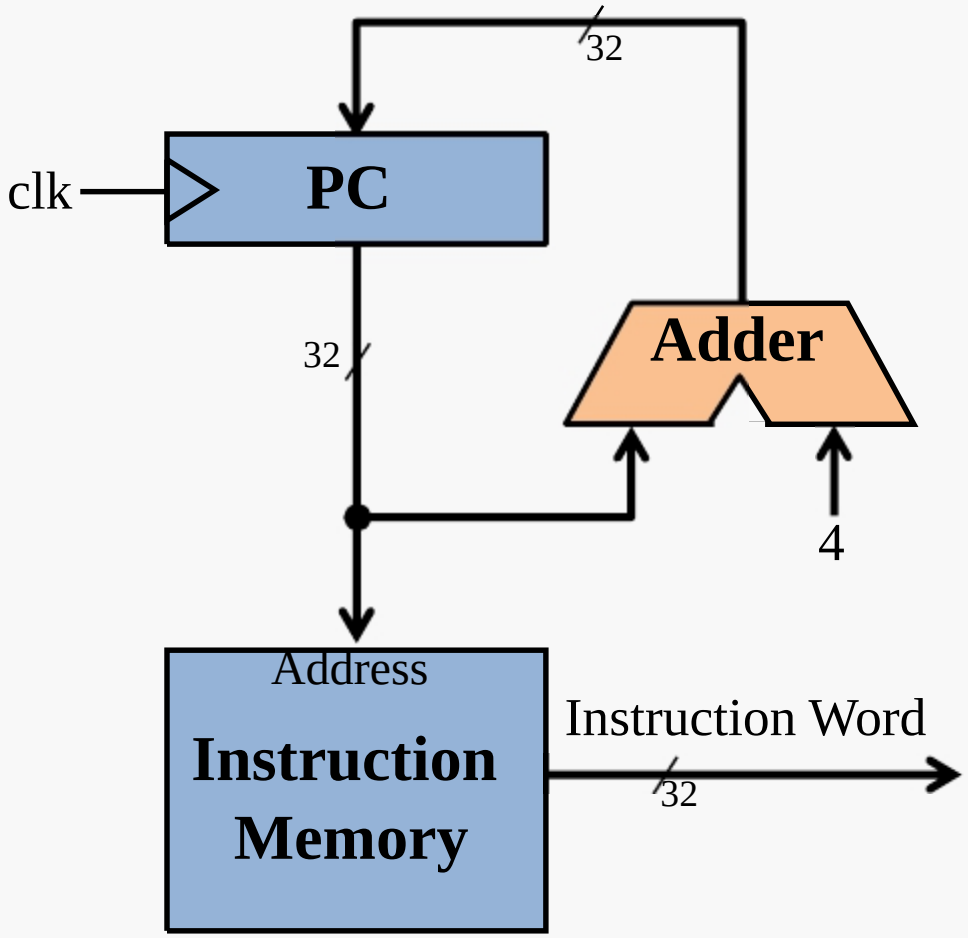
- 程序计数器（PC）的内容是指令的地址
- 用PC的内容作为地址，访问指令存储器获得指令编码



所有指令的共同需求

取指令

- 程序计数器（PC）的内容是指令的地址
- 用PC的内容作为地址，访问指令存储器获得指令编码



更新程序计数器（PC）

- 顺序执行时
  - PC ← PC + 4
- 发生分支时
  - PC ← 分支目标的地址

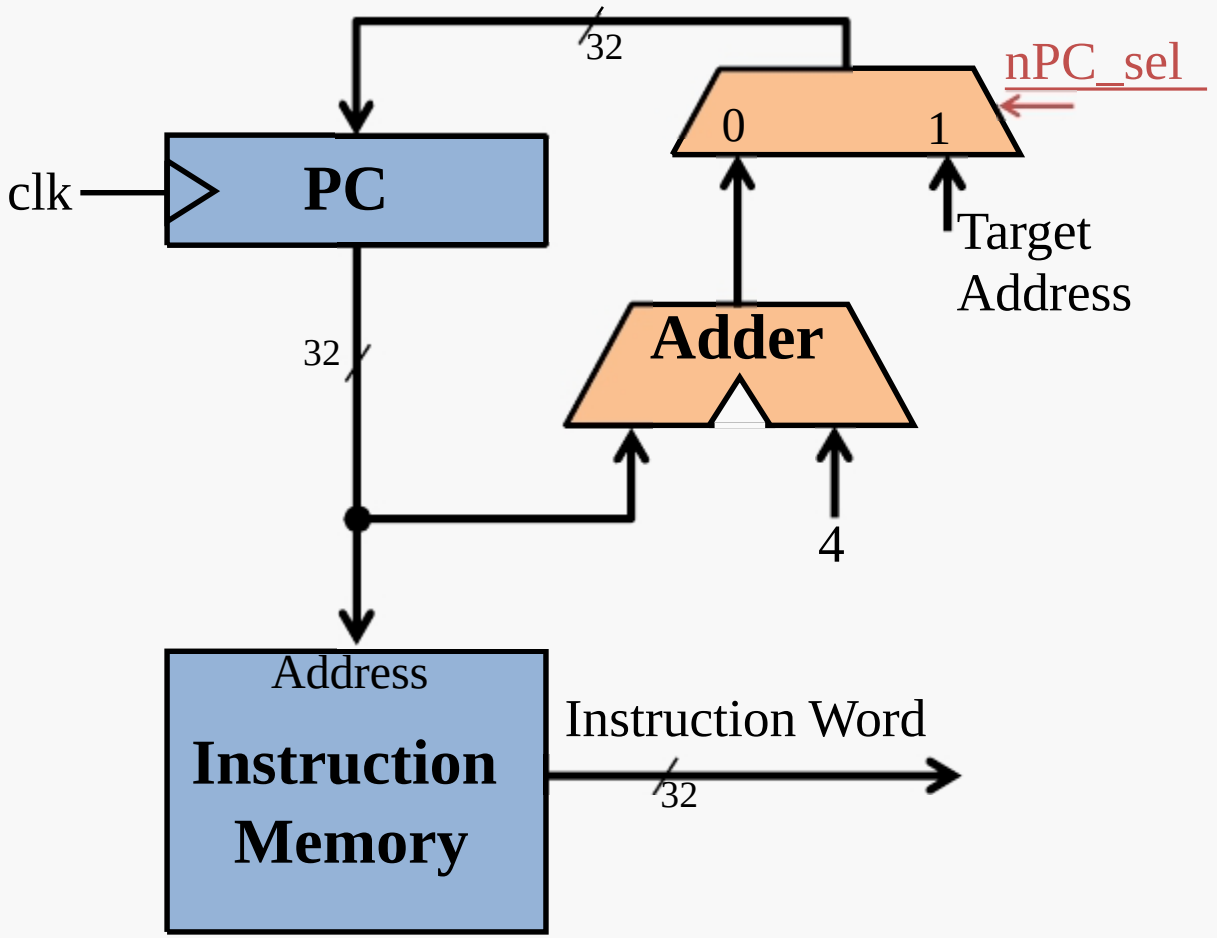
所有指令的共同需求

取指令

- 程序计数器（PC）的内容是指令的地址
- 用PC的内容作为地址，访问指令存储器获得指令编码

更新程序计数器（PC）

- 顺序执行时
  - PC ← PC + 4
- 发生分支时
  - PC ← 分支目标的地址



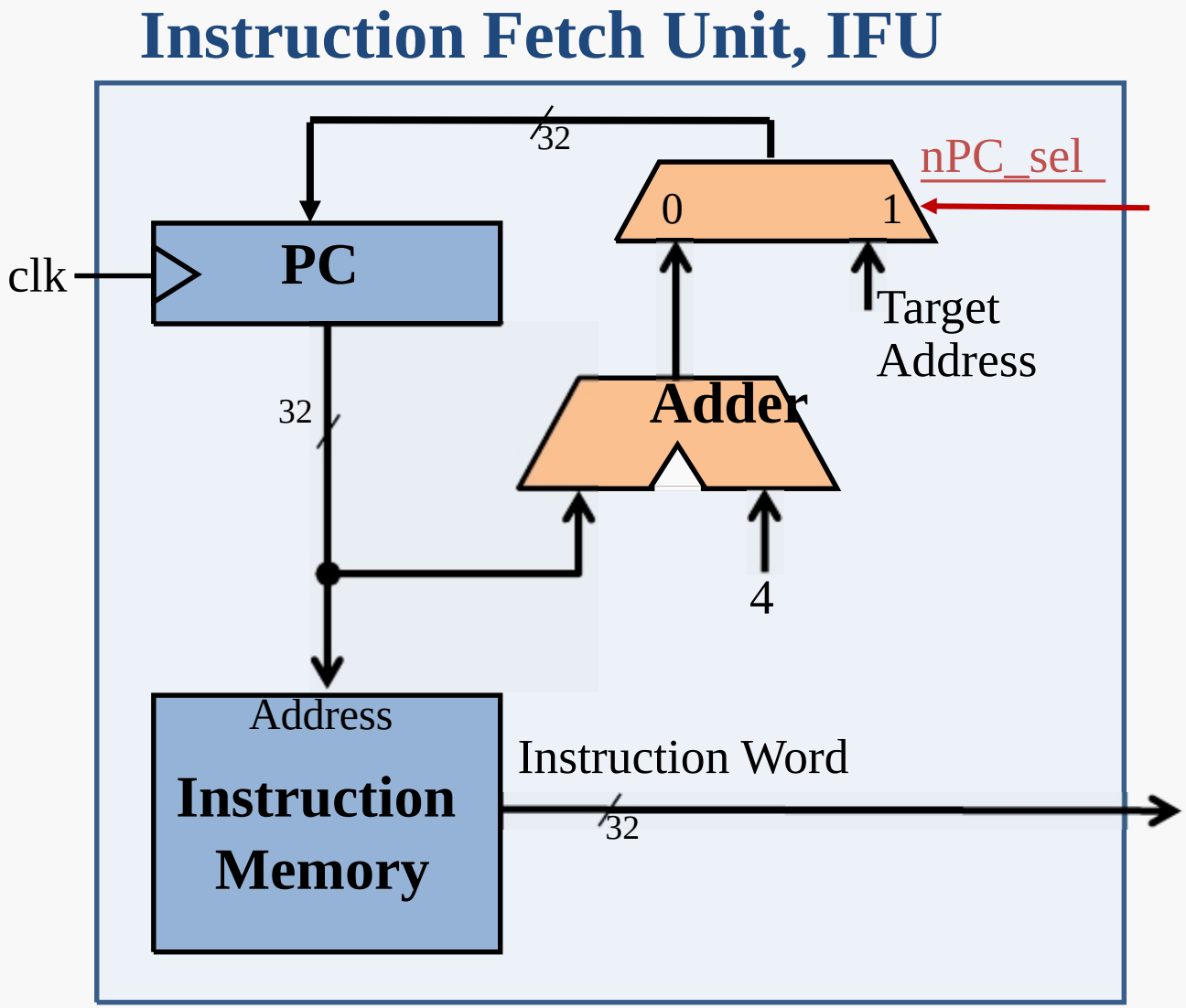
所有指令的共同需求

取指令

- 程序计数器（PC）的内容是指令的地址
- 用PC的内容作为地址，访问指令存储器获得指令编码

更新程序计数器（PC）

- 顺序执行时
  - PC ← PC + 4
- 发生分支时
  - PC ← 分支目标的地址



建立数据通路的方法

基本原则

- 根据指令需求，连接组件，建立数据通路

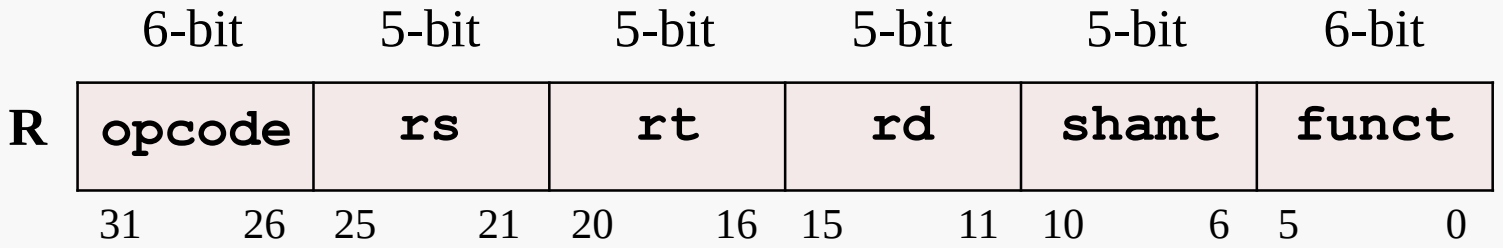
指令的需求

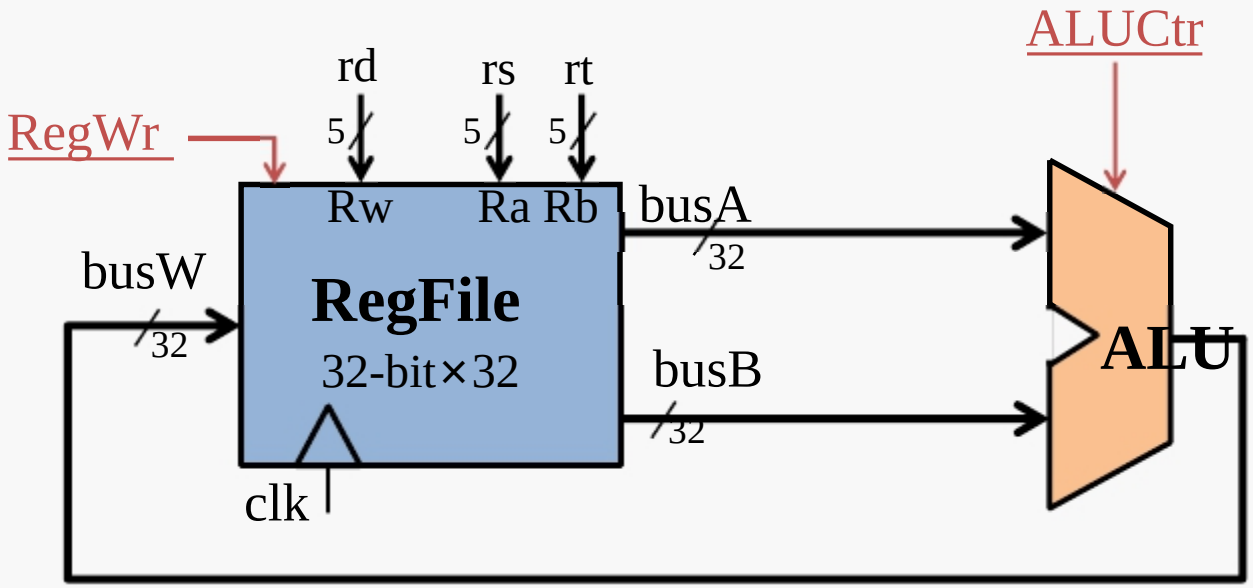
- 所有指令的共同需求
- **不同指令的不同需求**

加法和减法指令的需求

R[rd] = R[rs] op R[rt]

- addu rd,rs,rt　　subu rd,rs,rt



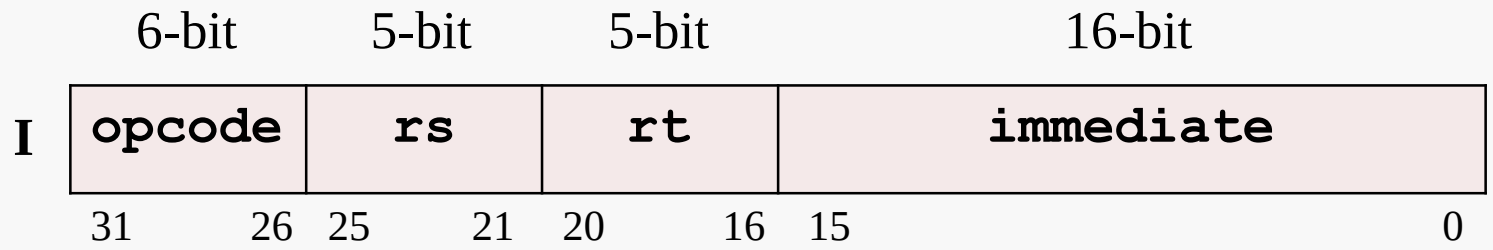


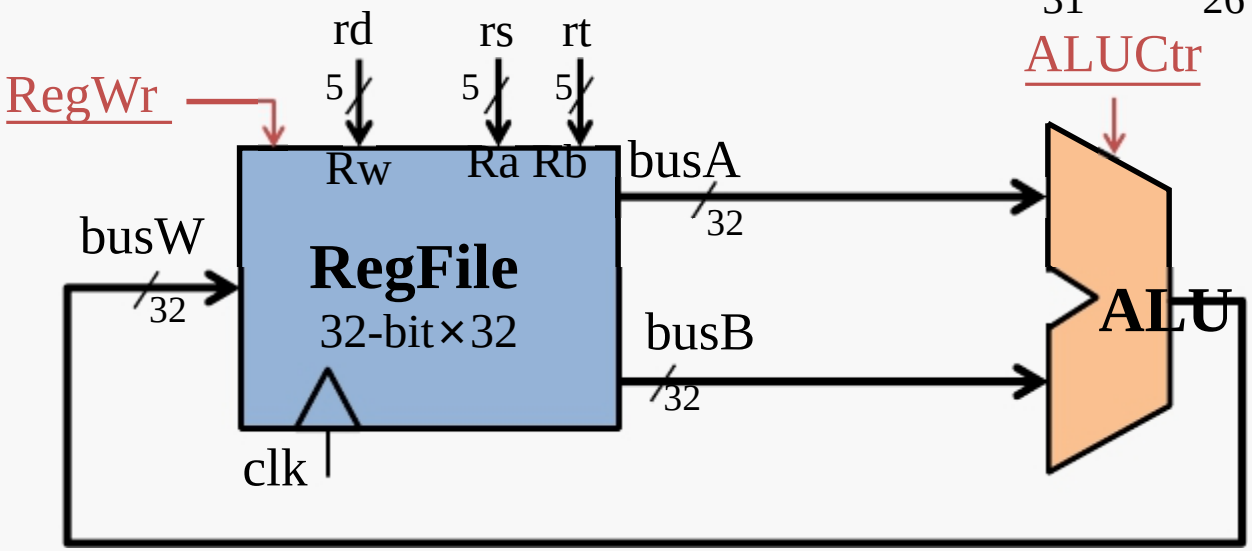
注：ALUCtr和RegWr是由指令译码生成的控制信号

逻辑运算指令的需求

```
R[rt] = R[rs] op ZeroExt[imm16]
```

- `ori rt,rs,imm16`





问题1：目的寄存器是rt而非rd

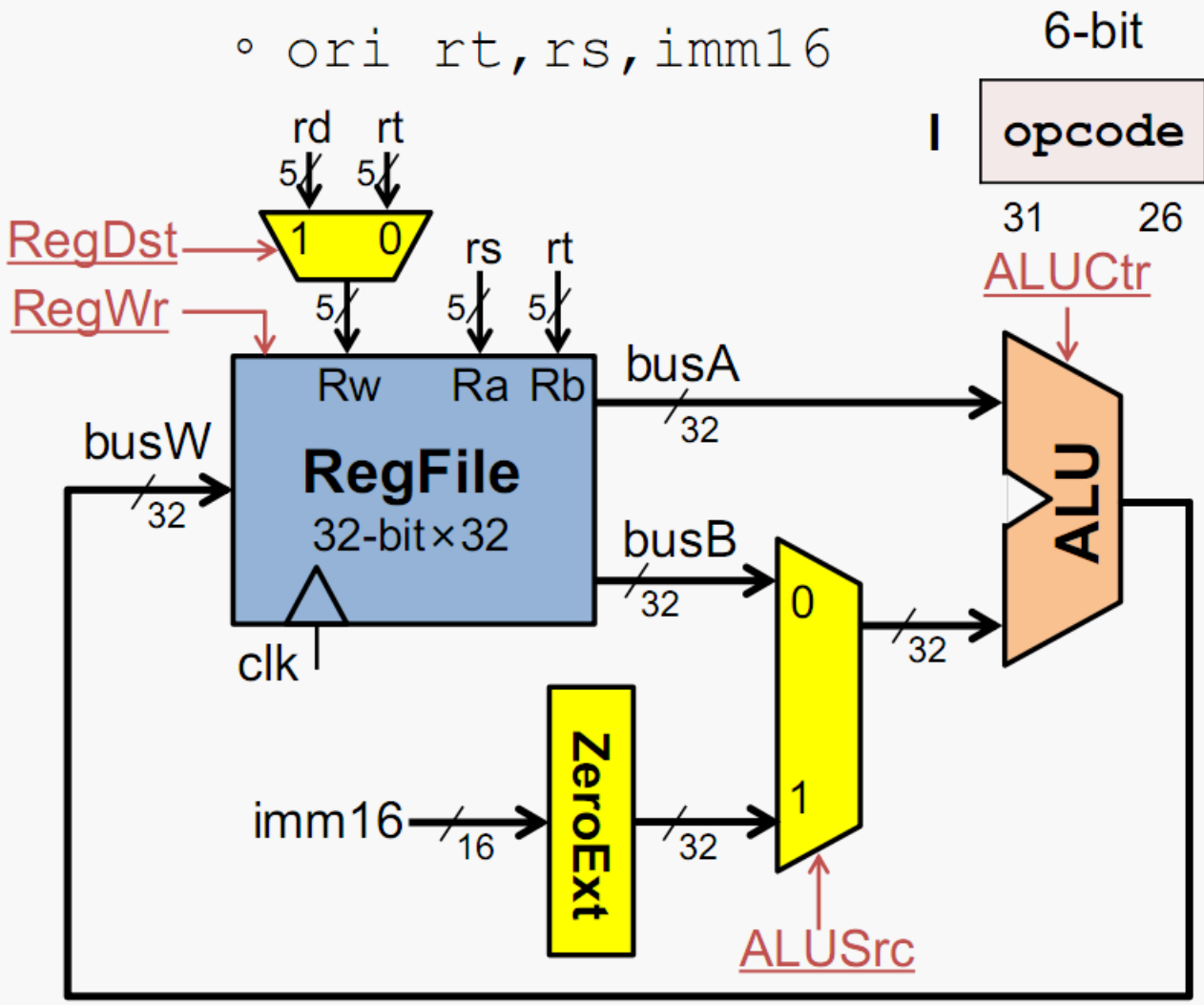
问题2：立即数是ALU的输入

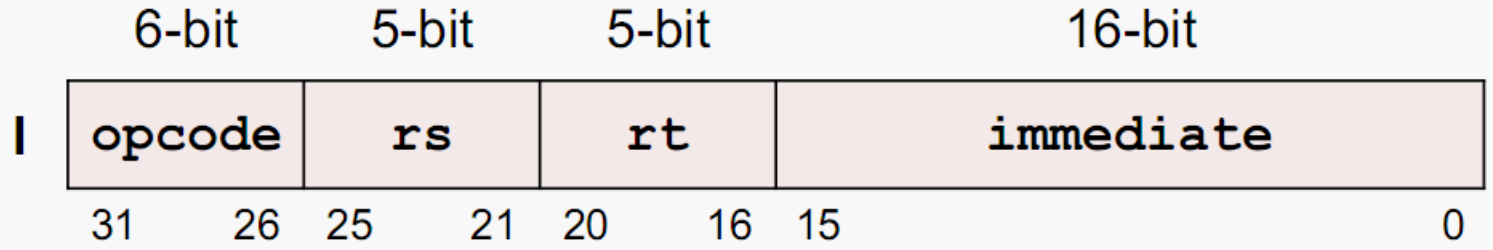
问题3：立即数只有16位

逻辑运算指令的需求

R[rt] = R[rs] op ZeroExt[imm16]

- ori rt,rs,imm16





问题1：目的寄存器是rt而非rd

问题2：立即数是ALU的输入

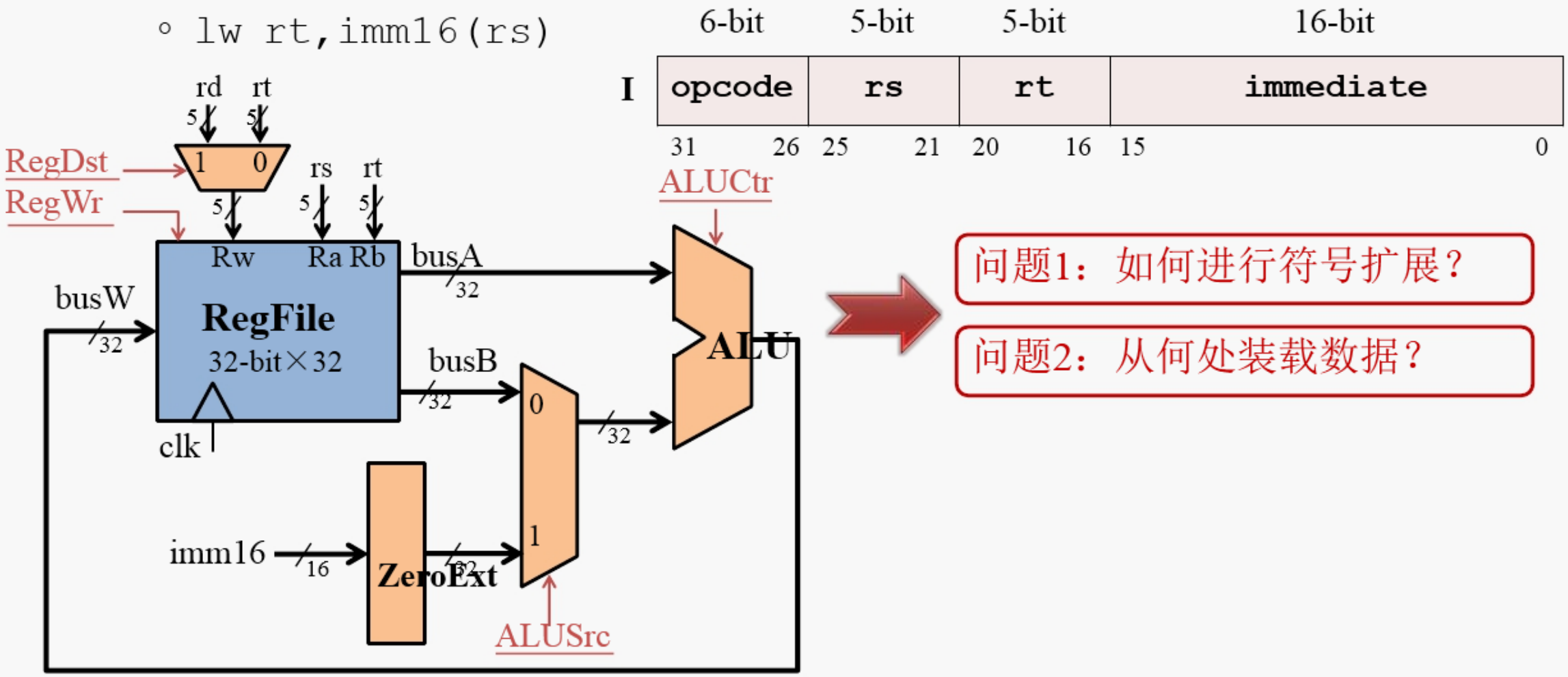
问题3：立即数只有16位

解决方案：增加两个多选器，增加一个零扩展部件

访存指令的需求（Load）

R[rt] = Mem[R[rs]+SignExt[imm16]]

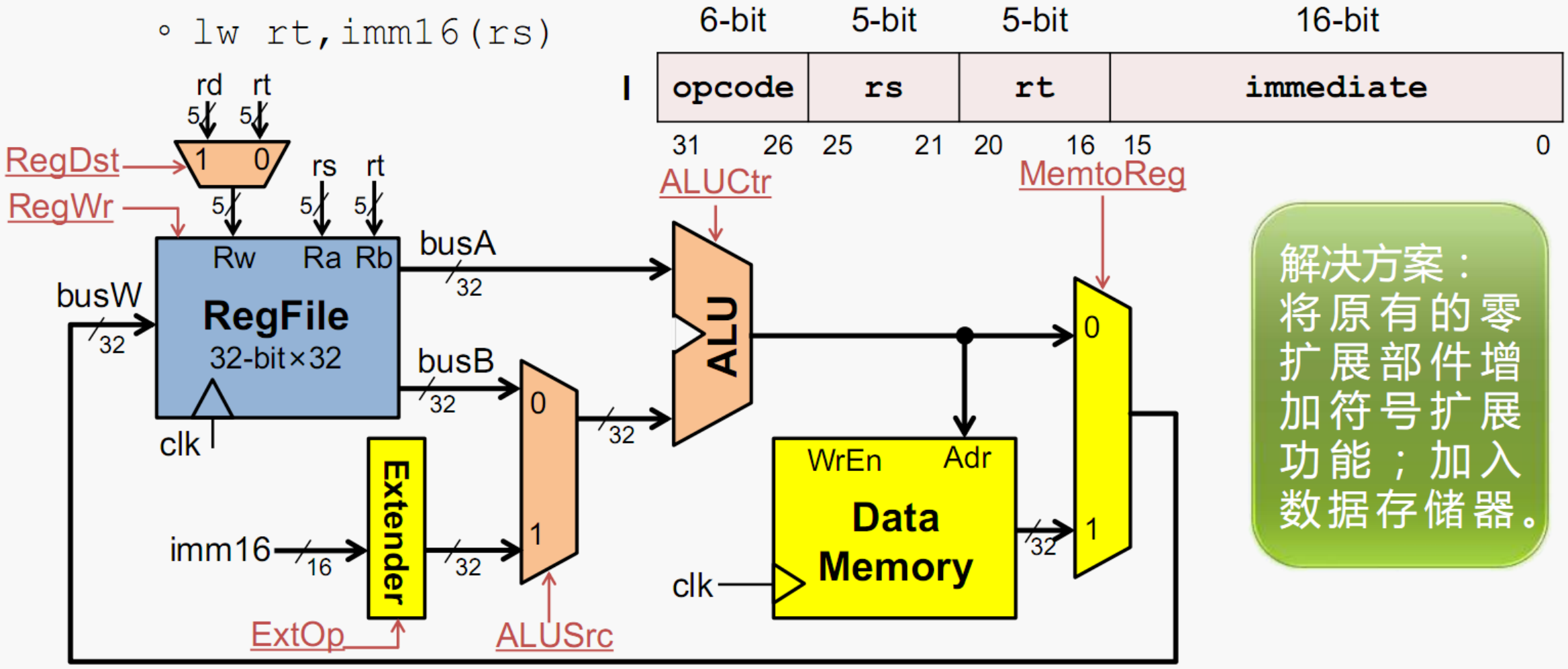
- lw rt,imm16(rs)



访存指令的需求（Load）

R[rt] = Mem[R[rs]+SignExt[imm16]]

- lw rt,imm16(rs)

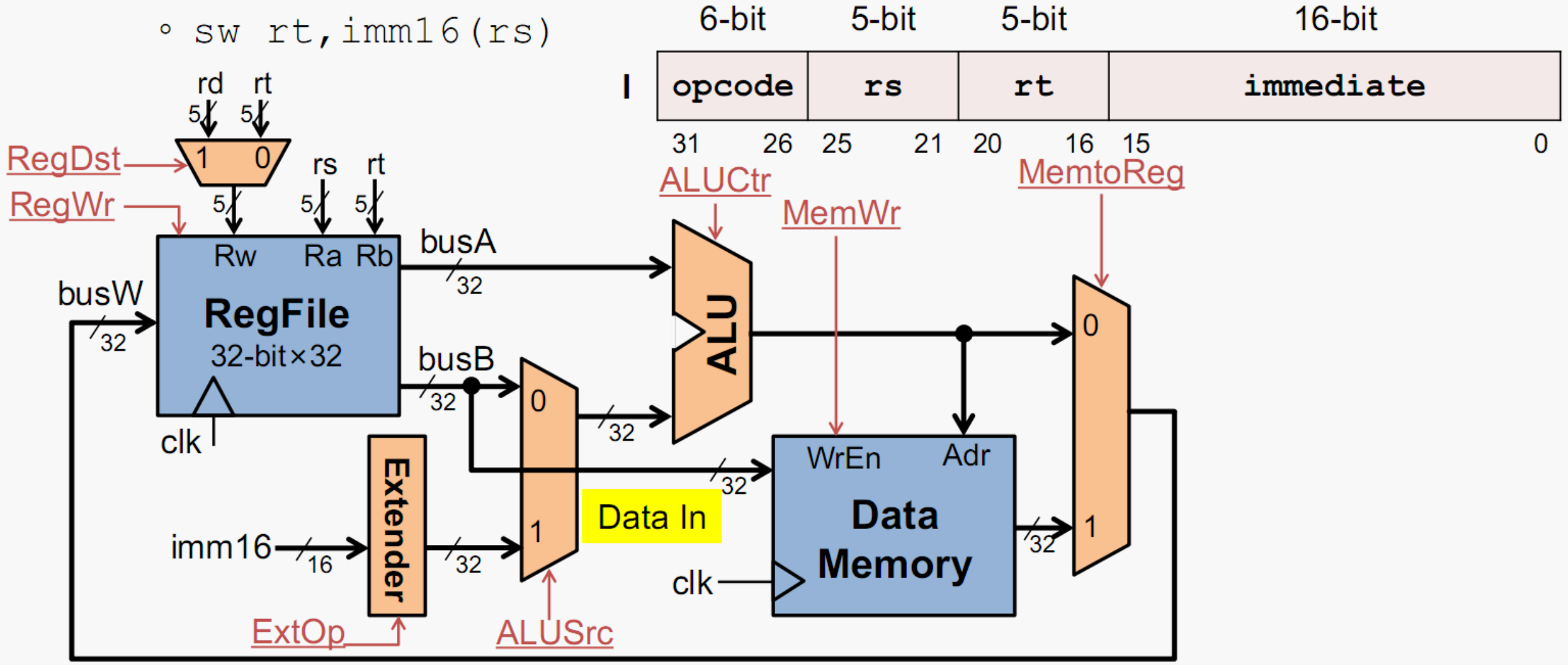


解决方案：将原有的零扩展部件增加符号扩展功能；加入数据存储器。

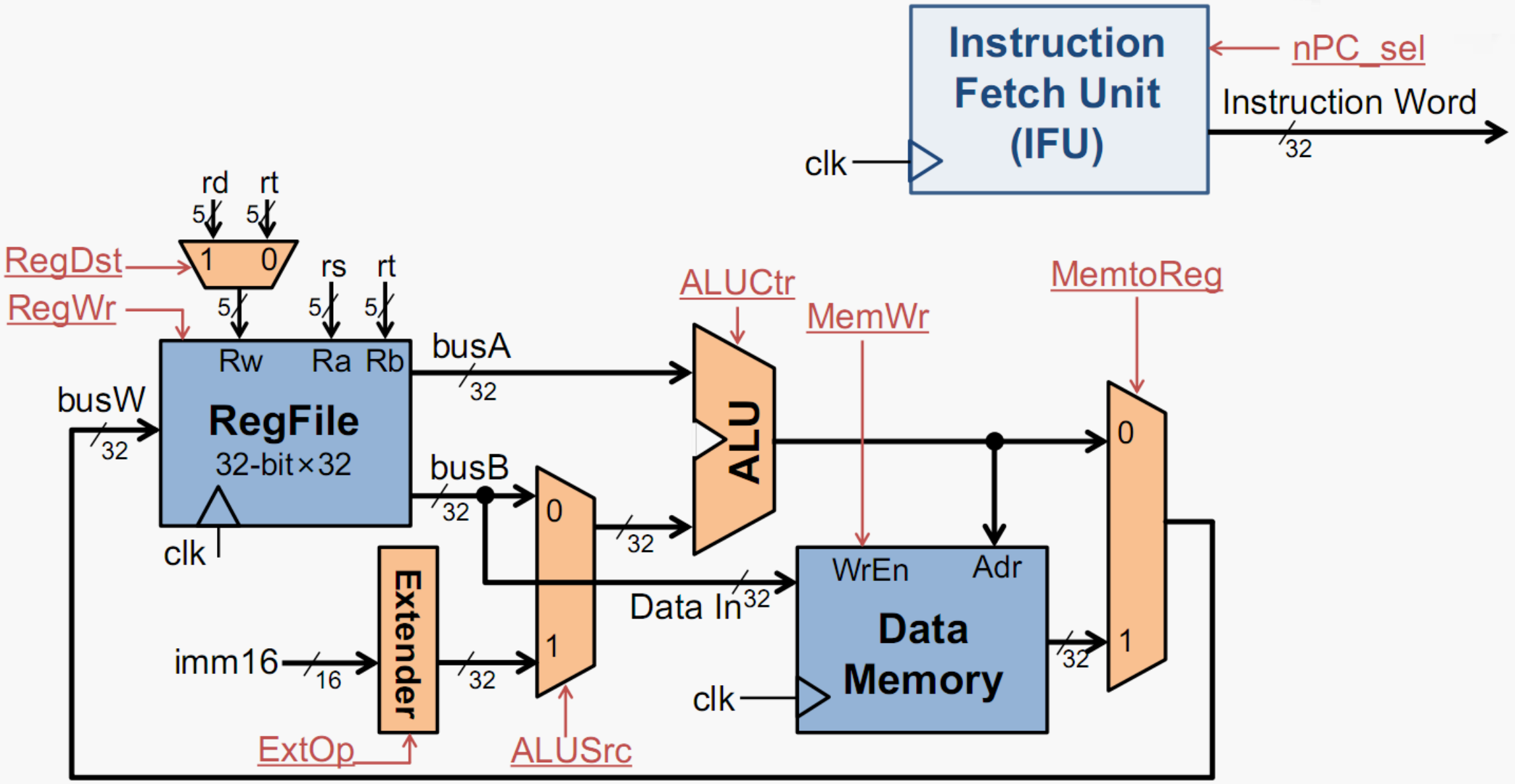
访存指令的需求（Store）

Mem[R[rs]+**SignExt**[imm16]] = R[**rt**]

- sw rt,imm16(rs)



数据通路初步完成


Instruction Fetch Unit (IFU)
nPC_sel
Instruction Word
32
clk
rd
rt
5
5
RegDst
1
0
RegWr
5
rs
rt
5
5
Rw
Ra
Rb
RegFile
32-bit×32
busW
32
clk
busA
32
busB
32
ALUCtr
ALU
MemWr
MemtoReg
0
1
32
Extender
imm16
16
32
ExtOp
ALUSrc
Data In
32
WrEn
Adr
Data Memory
clk
32
0
1

处理器的设计步骤

① 分析指令系统，得出对数据通路的需求 ✓

② 为数据通路选择合适的组件 ✓

③ 连接组件建立数据通路 ✓

④ 分析每条指令的实现，以确定控制信号

⑤ 集成控制信号，形成完整的控制逻辑

第五章　单周期处理器

1. 处理器设计的 主要步骤
2. 数据通路的建立
3. 运算指令的 控制信号
4. 访存指令的 控制信号
5. 分支指令的 控制信号
6. 控制信号的集成

不同维度的指令分类

	R型指令	I型指令	J型指令
运算指令	addu rd,rs,rt subu rd,rs,rt	ori rt,rs,*imm16*	
访存指令		lw rt,*imm16*(rs) sw rt,*imm16*(rs)	
分支指令		beq rs,rt,*imm16*	

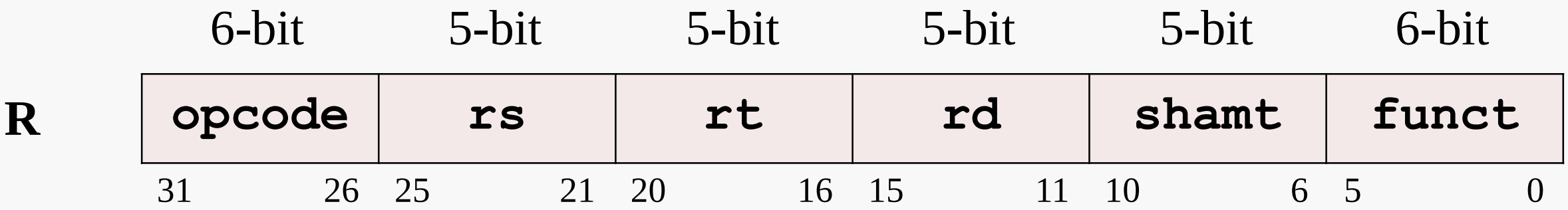
加法指令的操作步骤

```
addu rd, rs, rt
```

① MEM[PC] 从指令存储器中取回指令

② R[rd]=R[rs]+R[rt] 指令指定的操作

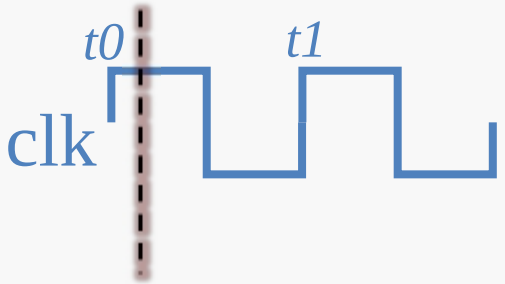
③ PC=PC + 4 计算下一条指令的地址

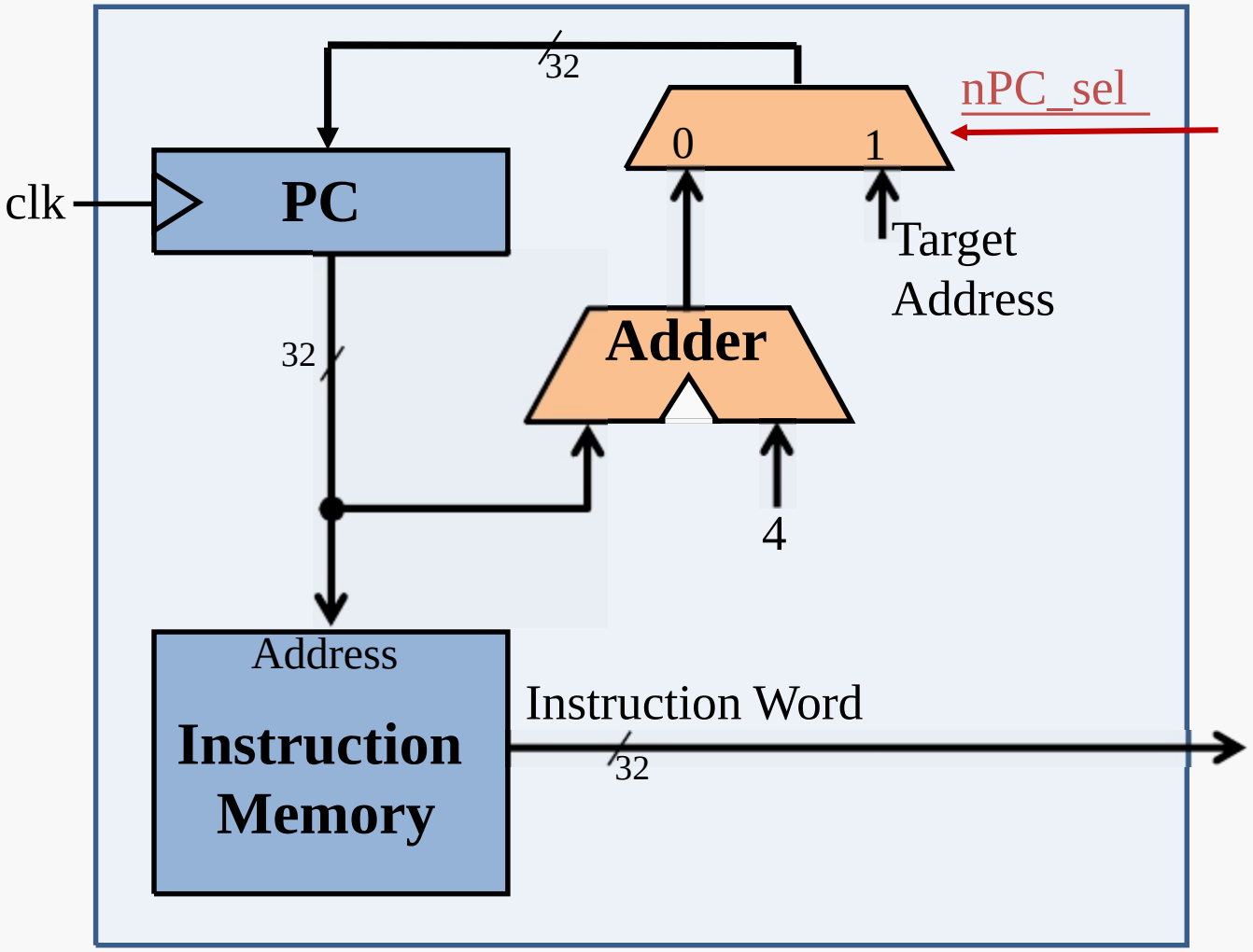


加法指令的操作步骤（1）

`Instruction = MEM[PC]` **Instruction Fetch Unit, IFU**

- 从指令存储器中取回指令
- 所有指令均有此步骤

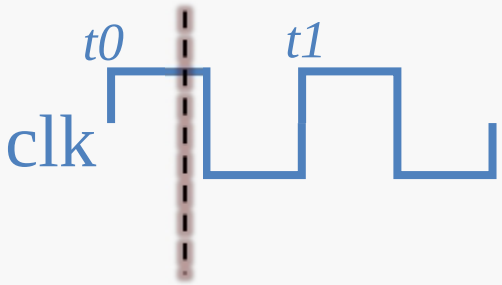


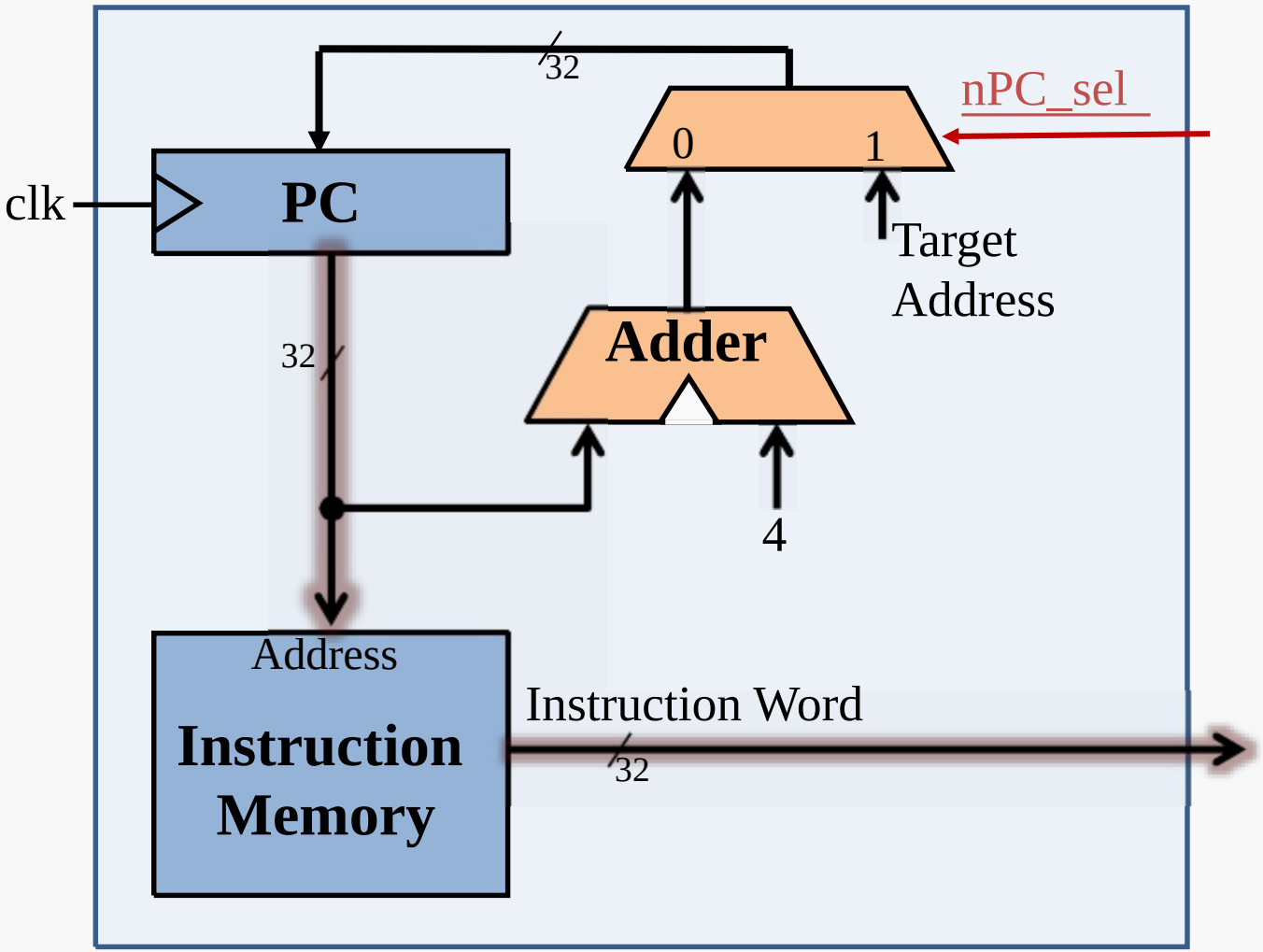


加法指令的操作步骤（1）

`Instruction = MEM[PC]` **Instruction Fetch Unit, IFU**

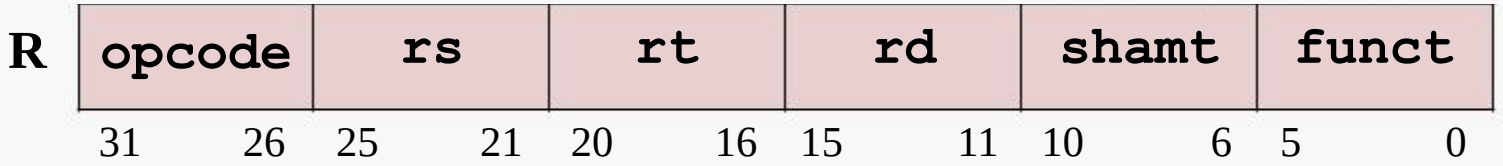
- 从指令存储器中取回指令
- 所有指令均有此步骤

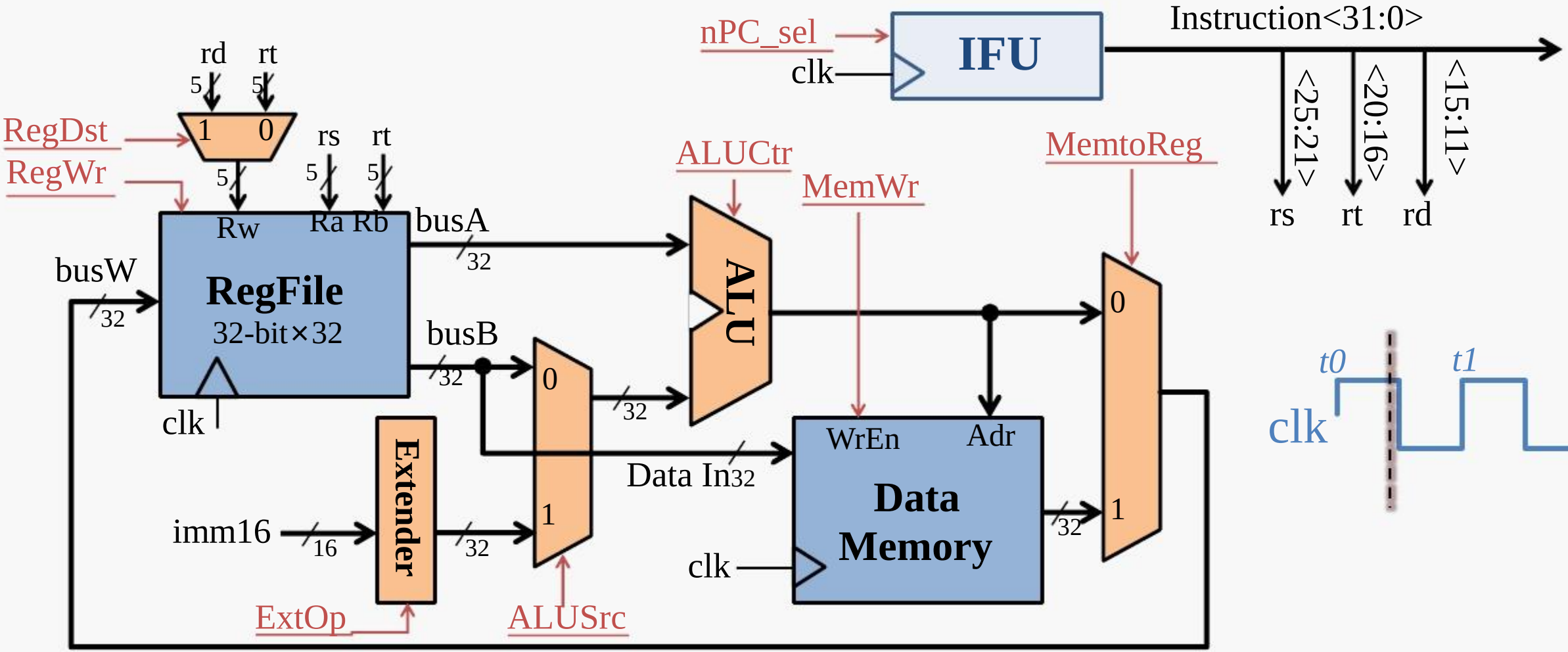




加法指令的操作步骤（2）

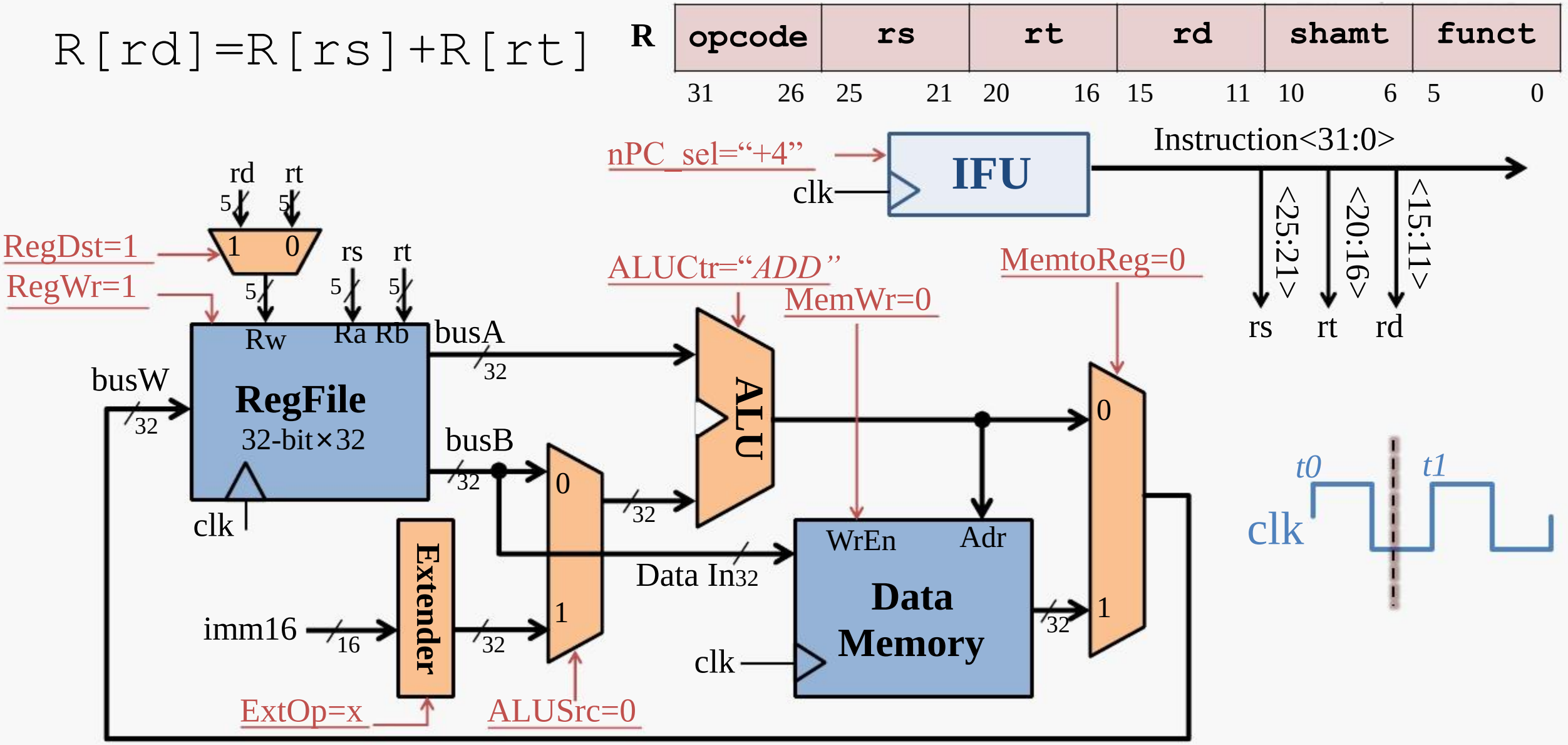
R[rd]=R[rs]+R[rt]





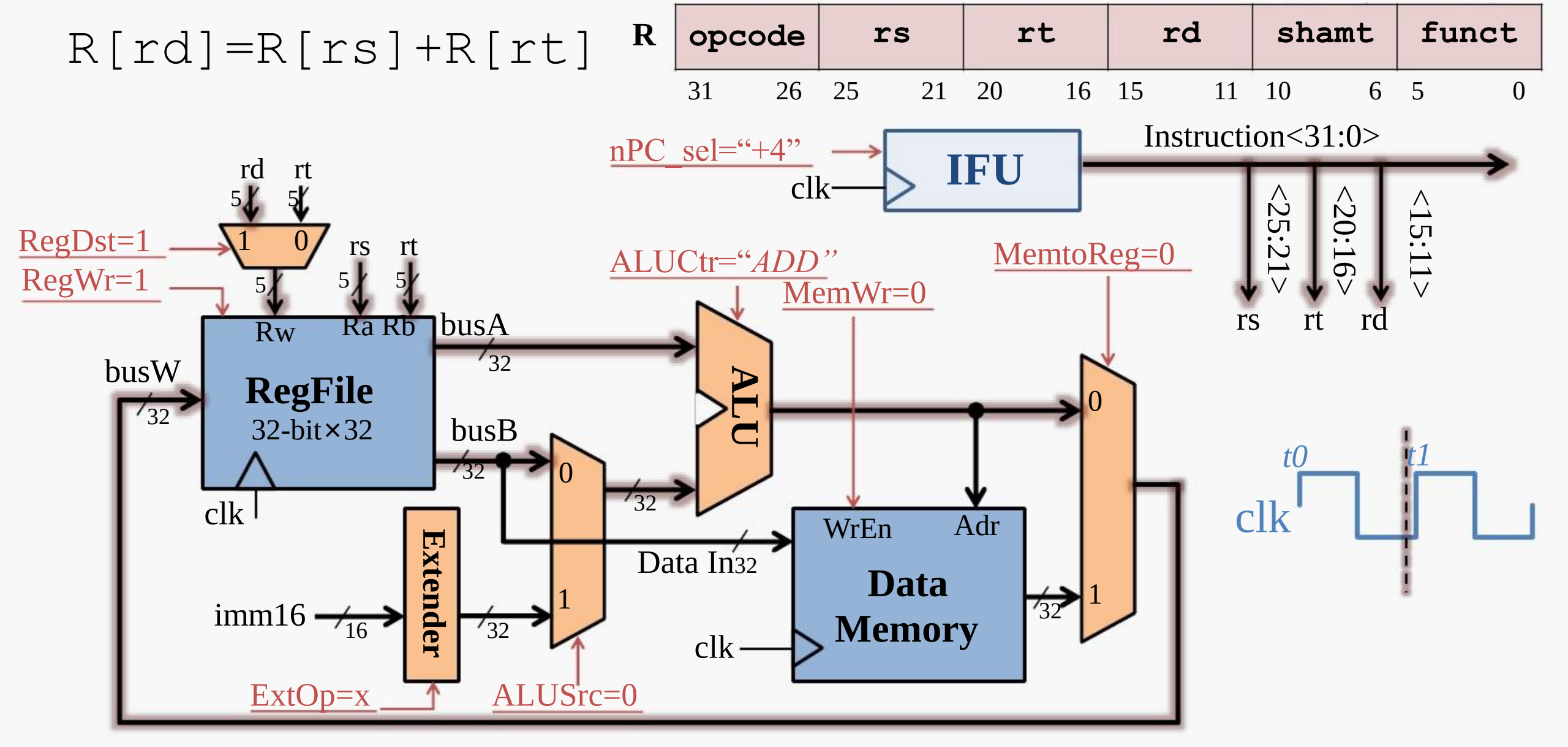
加法指令的操作步骤（2）

R[rd]=R[rs]+R[rt]



加法指令的操作步骤（2）

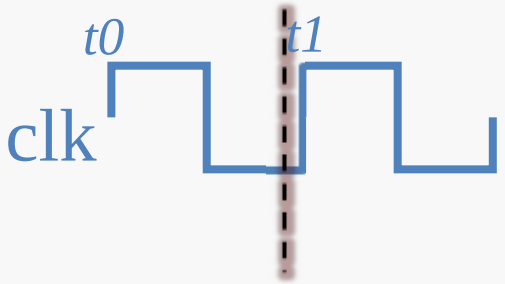
R[rd]=R[rs]+R[rt]

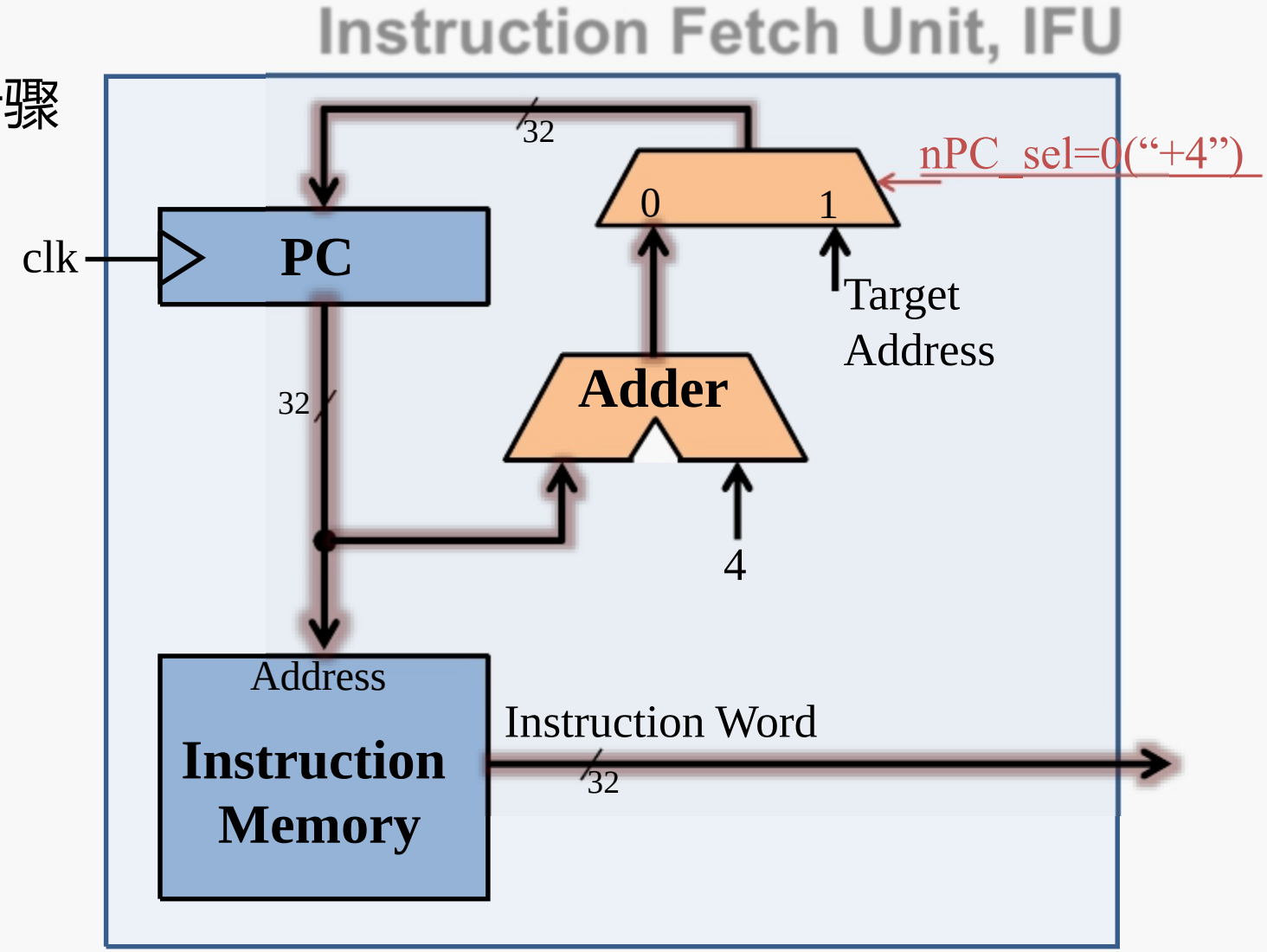


加法指令的操作步骤（3）

PC=PC + 4

- 除了分支指令，均有此步骤





不同维度的指令分类

运算指令	addu rd,rs,rt subu rd,rs,rt	ori rt,rs,*imm16*	
访存指令		lw rt,*imm16*(rs) sw rt,*imm16*(rs)	
分支指令		beq rs,rt,*imm16*	
	R型指令	I型指令	J型指令

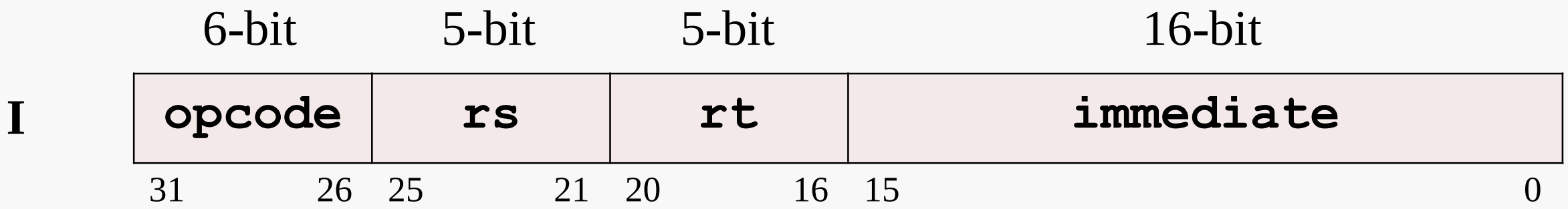
ori指令的操作步骤

```
ori  rt, rs,  imm16
```

① `MEM[PC]` 从指令存储器中取回指令

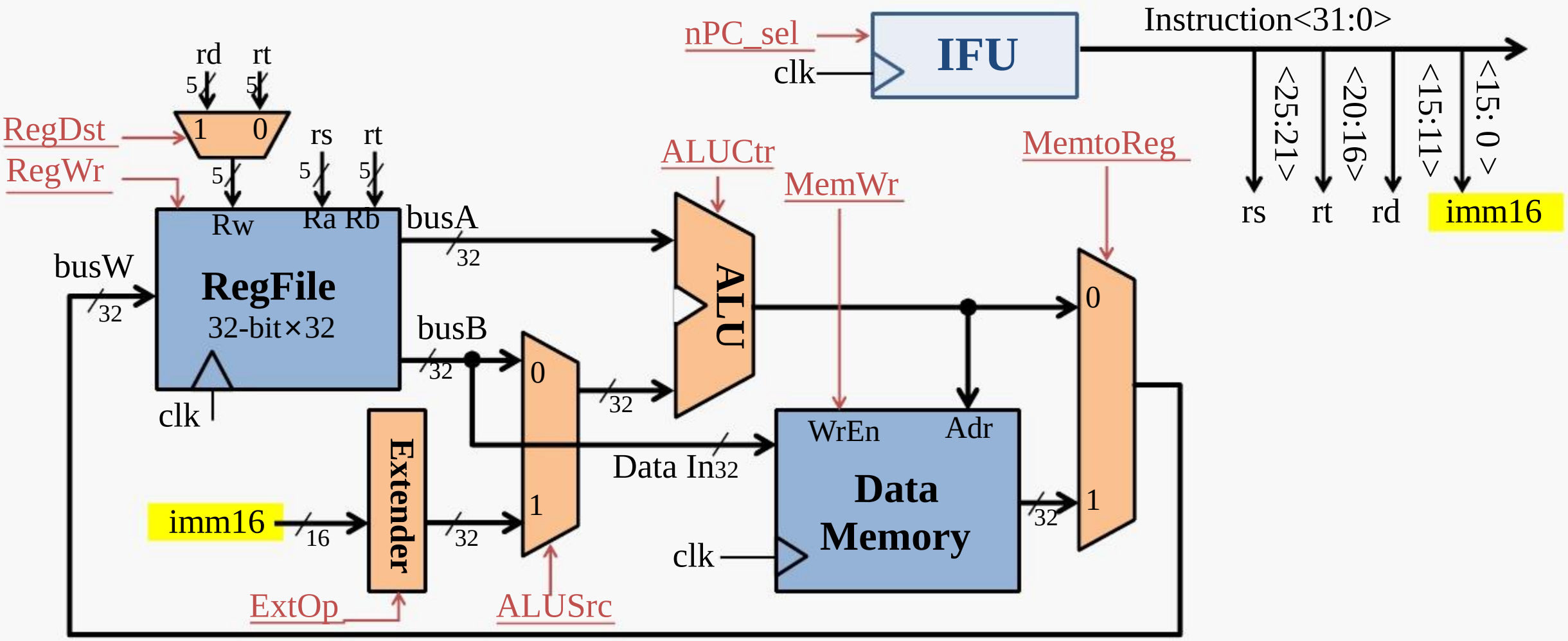
② `R[rt]=R[rs] | ZeroExt[imm16]` 指令指定的操作

③ `PC=PC + 4` 计算下一条指令的地址



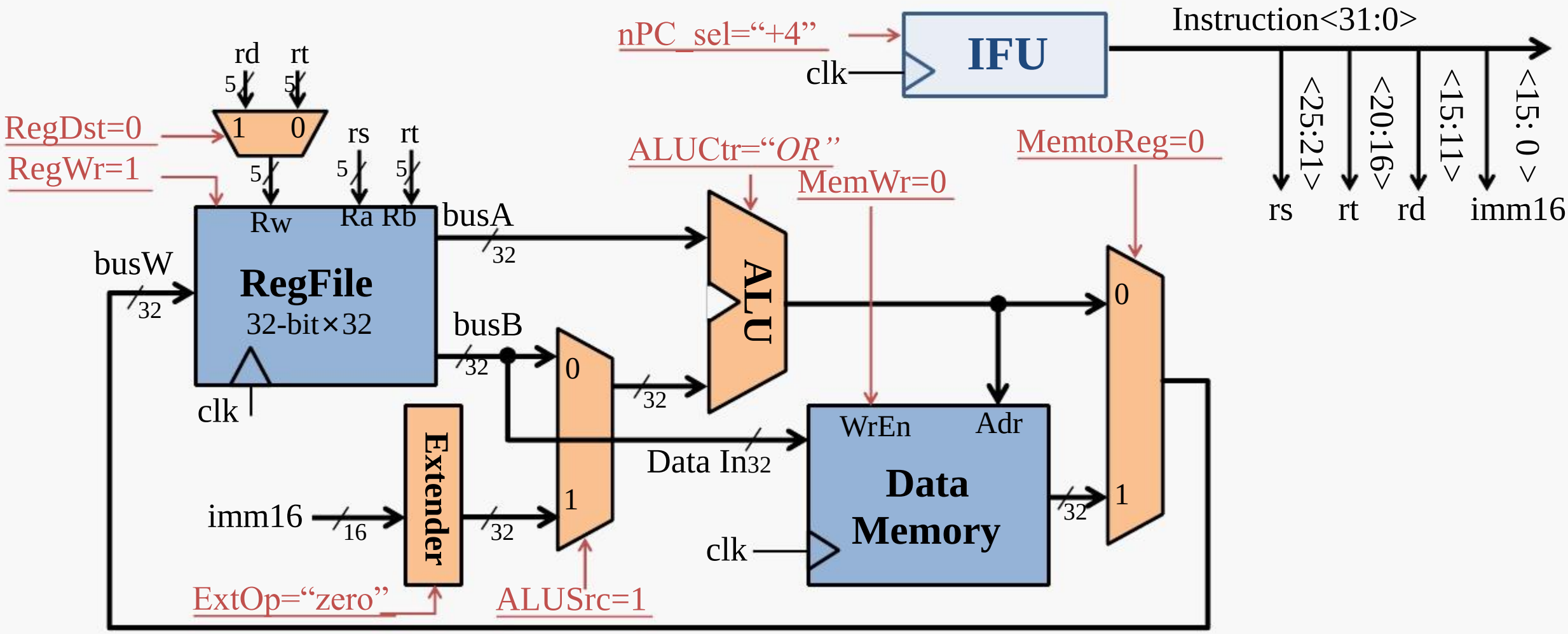
ori指令的操作步骤（2）

```
R[rt]= R[rs] | ZeroExt[imm16]
```



ori指令的操作步骤（2）

```
R[rt]= R[rs] | ZeroExt[imm16]
```



ori指令的操作步骤（2）

```
R[rt]= R[rs] | ZeroExt[imm16]
```