计算机组织结构

4 整数运算

刘博涵

2022年9月29日

南京大学
NANJING UNIVERSITY

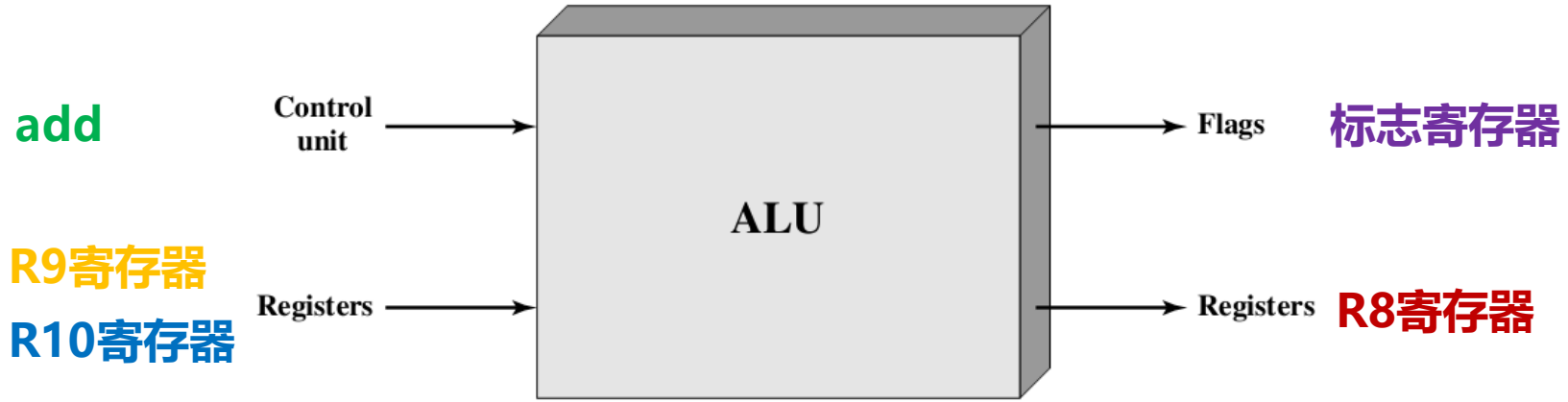
教材对应章节

第3章 运算方法和运算部件

第10章 计算机算术运算

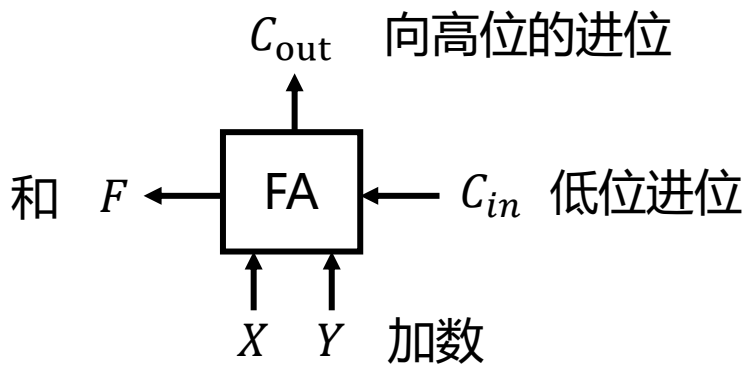
算术逻辑单元 (ALU)

- 算术逻辑单元 (ALU) 是计算机实际完成数据算术逻辑运算的部件
  - 数据由寄存器 (Registers) 提交给ALU，运算结果也存于寄存器
  - ALU可能根据运算结果设置一些标志 (Flags)，标志值也保存在处理器内的寄存器中
  - 控制器 (Control unit) 提供控制ALU操作和数据传入送出ALU的信号

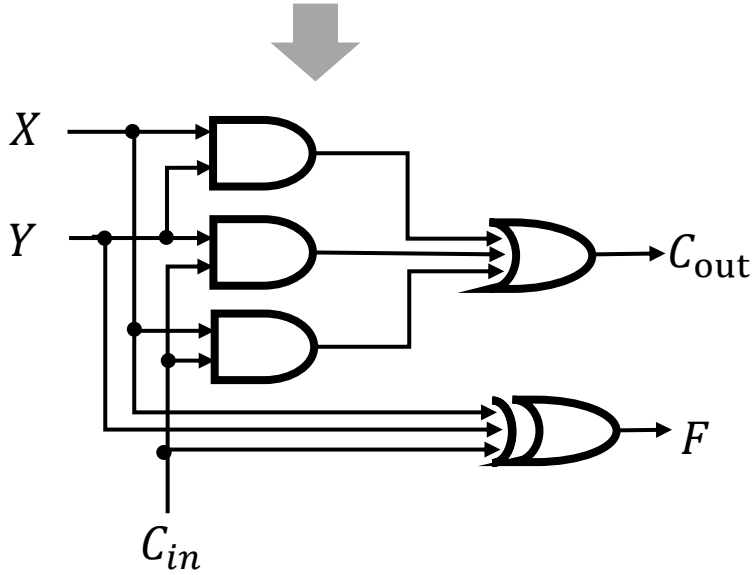


全加器

- 一位（1bit）加法：$X + Y$



X	Y	C_{in}	F	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

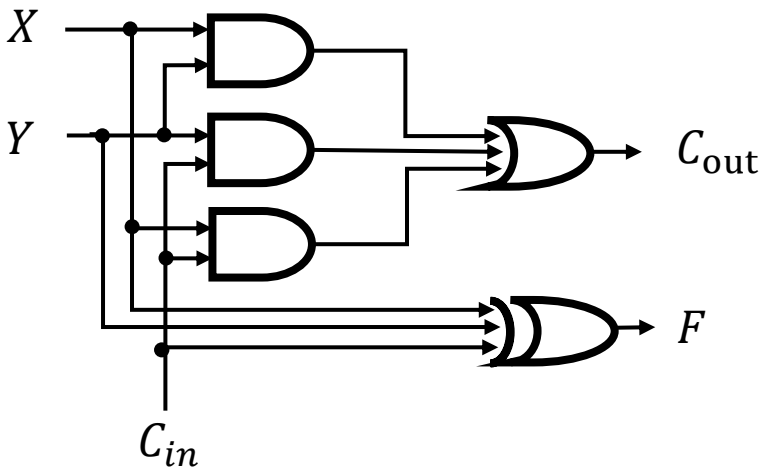


$\oplus$ → 异或
$+$ → 或
$\cdot$ → 与

$$F = X \oplus Y \oplus C_{in}$$

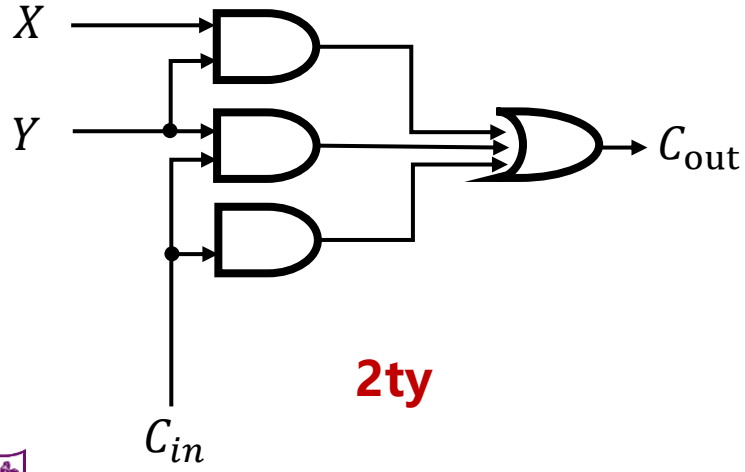
$$C_{\text{out}} = X \cdot Y + X \cdot C_{in} + Y \cdot C_{in}$$

全加器（一位加法器）

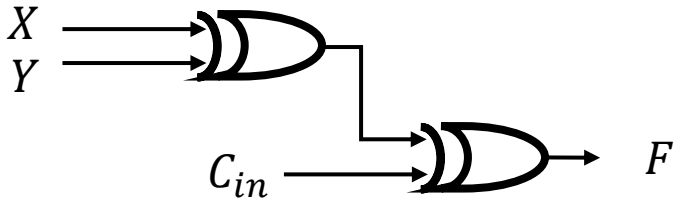


假设：
与门延迟：　1级门延迟 (1ty)
或门延迟：　1级门延迟 (1ty)
异或门延迟: 3级门延迟 (3ty)

与门和或门可以有多个输入端；异或门只能有2个输入端



2ty

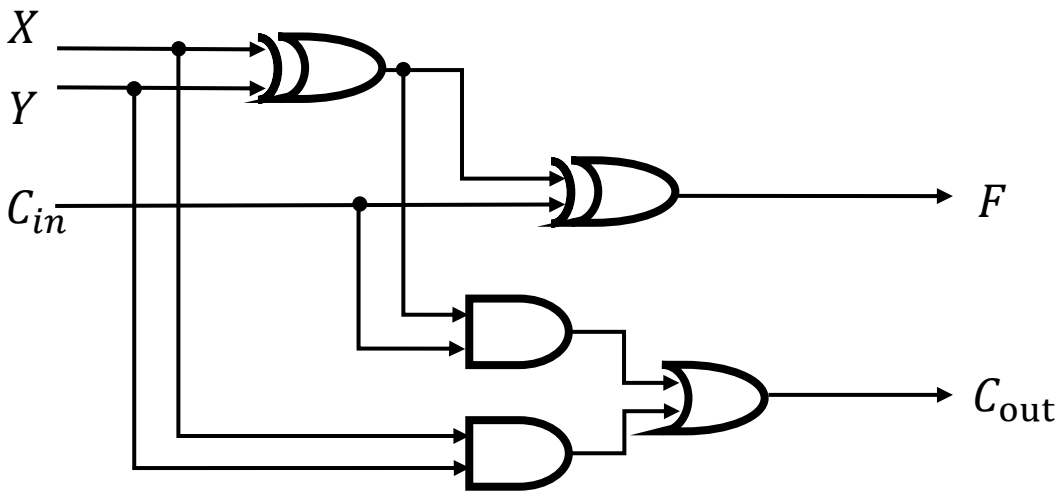


6ty

实际需要3个与门，1个或门，2个异或门

全加器（一位加法器）

$$C_{\text{out}} = X \cdot Y + X \cdot C_{in} + Y \cdot C_{in} \quad \rightarrow \quad C_{\text{out}} = X \cdot Y + (X + Y) \cdot C_{in} = X \cdot Y + (X \oplus Y) \cdot C_{in}$$



实际需要2个与门，1个或门，2个异或门

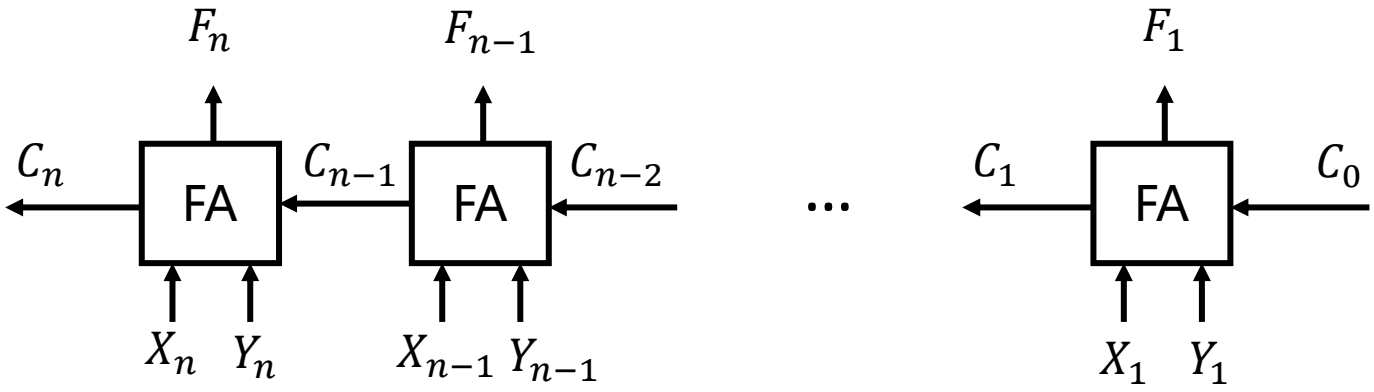
延迟?

串行进位（行波进位）加法器

- 1位（1bit）加法： $X_i + Y_i$
- 第 i 位加法：

$$F_i = X_i \oplus Y_i \oplus C_{i-1}$$

$$C_i = X_i C_{i-1} + Y_i C_{i-1} + X_i Y_i$$



- 缺点：延迟慢
  - C_n: $\boldsymbol{2n}\ ty$
  - F_n: $(\boldsymbol{2n+1})\ ty$

高位的运算必须**等待低位**的“进位输出信号”
能否**提前计算**出“进位输出信号”？

注意：这不是一个确定的公式

全先行进位加法器

$$C_i = X_i \cdot Y_i + (X_i + Y_i) \cdot C_{i-1}$$

$$C_1 = X_1 \cdot Y_1 + (X_1 + Y_1) \cdot C_0$$

$$C_2 = X_2 \cdot Y_2 + (X_2 + Y_2) \cdot X_1 \cdot Y_1 + (X_2 + Y_2) \cdot (X_1 + Y_1) \cdot C_0$$

$$C_3 = X_3 \cdot Y_3 + (X_3 + Y_3) \cdot X_2 \cdot Y_2 + (X_3 + Y_3) \cdot (X_2 + Y_2) \cdot X_1 \cdot Y_1 + (X_3 + Y_3) \cdot (X_2 + Y_2) \cdot (X_1 + Y_1) \cdot C_0$$

$$C_4 = \ldots$$

设：

- **生成（Generate）信号：** $G_i = X_i \cdot Y_i$
- **传播（Propagate）信号：** $P_i = X_i + Y_i$

则：

$$C_1 = G_1 + P_1 \cdot C_0$$

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

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

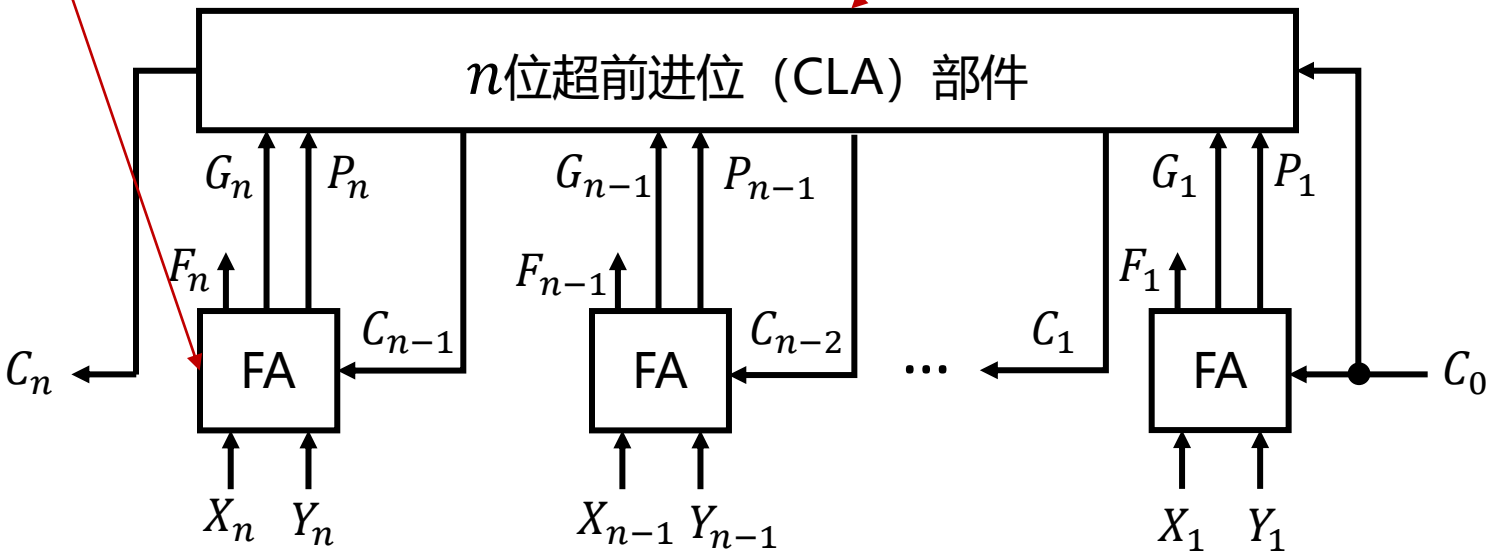
$$C_4 = \ldots$$

C_0，X_i，Y_i 都是已知输入

全先行进位加法器

每个FA包含：
1个或门，1个与门，2个异或门

C_n 的实现需要：
1个或门和n个与门



延迟：1ty + 2ty + 3ty = 6ty

延迟和加法器的位数无关

部分先行进位加法器

实现一个32位的加法器

◆采用行波进位（RCA）：

- **65级门延迟**

◆采用全先行进位（CLA）：

- **6级门延迟**
- **难以实现，C_{32}需要1个或门和32个与门，且最大需要33输入的与门和或门**

结合

◆ 部分先行进位加法器（串联4个8位CLA）

- **12级门延迟**

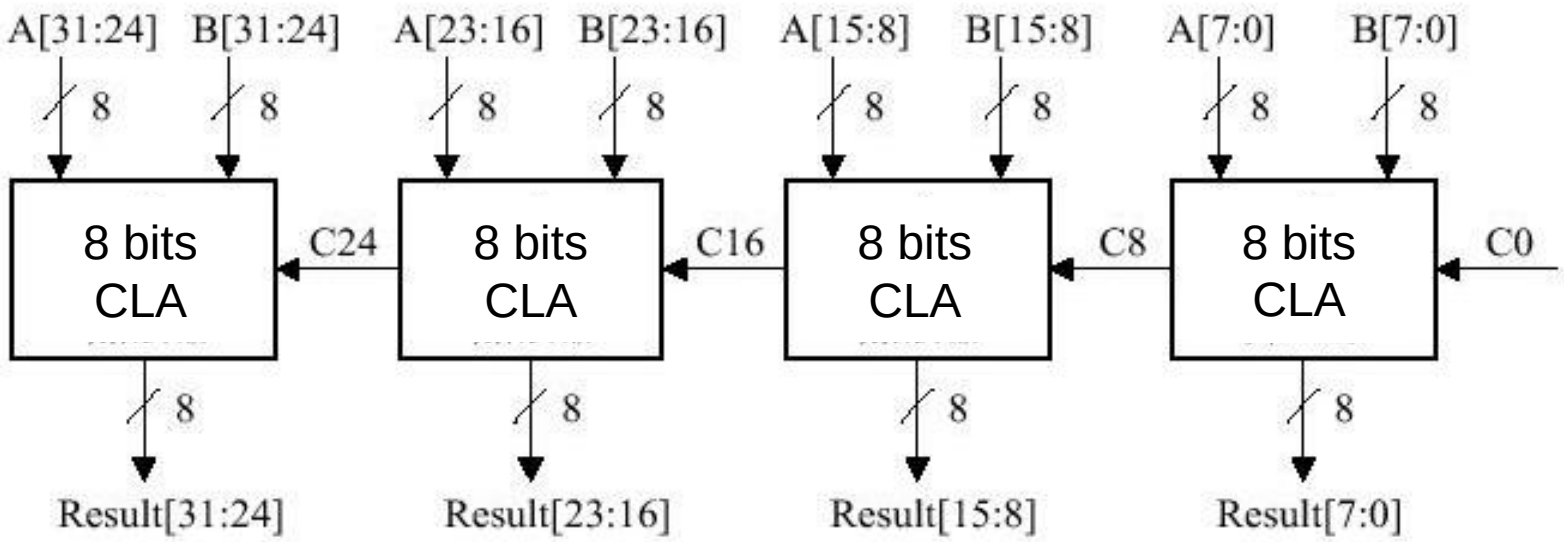
2021年国产第一台光刻机可以制作28nm芯片
2013年的iPhone5s使用28nm芯片，主频1.3GHz

加法器	28nm工艺（门延迟0.02ns）		7nm工艺（门延迟0.002ns）	
	最小延迟时间	最大时钟频率	最小延迟时间	最大时钟频率
32位RCA	1.3ns	0.77GHz	0.13ns	7.7GHz
32位CLA	0.12ns	8.33GHz	0.012ns	83.3GHz
4级8位CLA	0.24ns	4.17GHz	0.024ns	41.7GHz

实际上还存在寄存器保持时间、连线延迟等等

部分先行进位加法器

- **思路**
  - 采用**多个CLA**并将其**串联**，取得**计算时间**和**硬件复杂度**之间的**权衡**
- **例子**



延迟怎么算?

3ty + 2ty + 2ty + 5ty = 12ty

加法

- $[X+Y]_c = [X]_c + [Y]_c \ (MOD\ 2^n)$
- 溢出：

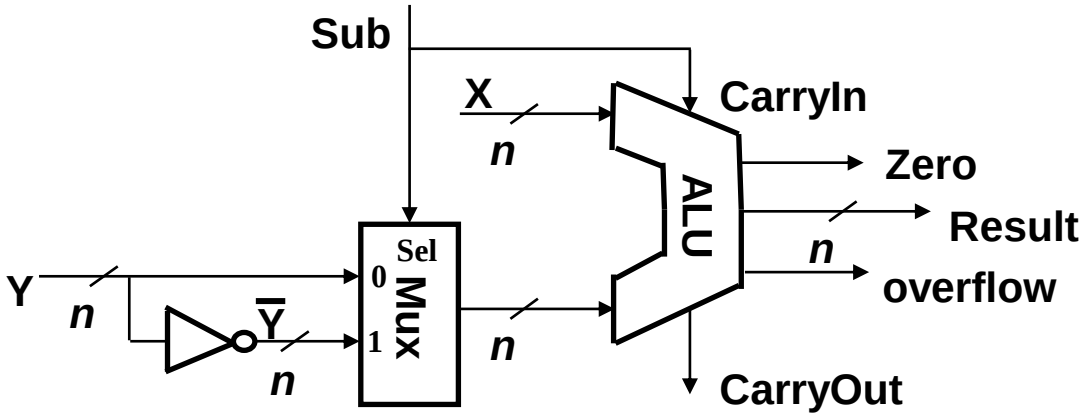
-7	1001		7	0111			
+ -6	+ 1010		+ 6	+ 0110			
-13	10011	3	13	1101	-3		

- $X_n = Y_n$且$F_n \neq X_n, Y_n$: $overflow = X_n Y_n \overline{F_n} + \overline{X_n}\ \overline{Y_n} F_n$
- $C_n \neq C_{n-1}$: $overflow = C_n \oplus C_{n-1}$

减法

- $[X-Y]_c = [X]_c + [-Y]_c \ (MOD\ 2^n)$
- 溢出：与加法相同

-7	1001	
- 6	+ 1010	
-13	10011	3



乘法

10进制计算机（ENIAC）

```
      2345
×     6789
-----------
     21105
    18760
   16415
  14070
-----------
  15920205
```

1位乘法
1位全加器
⋮
n位进位加法器

1位乘法 → 99乘法表 → 软件实现（存储）
99乘法表 → 硬件实现

2进制计算机（EDVAC）

```
      0111     被乘数
×     0110     乘数
-------------
      0000
     0111
    0111      部分积
   0000
-------------
  0101010     积
```

1位乘法 → 乘数位为0则为0
1位乘法 → 乘数位为1则为被乘数

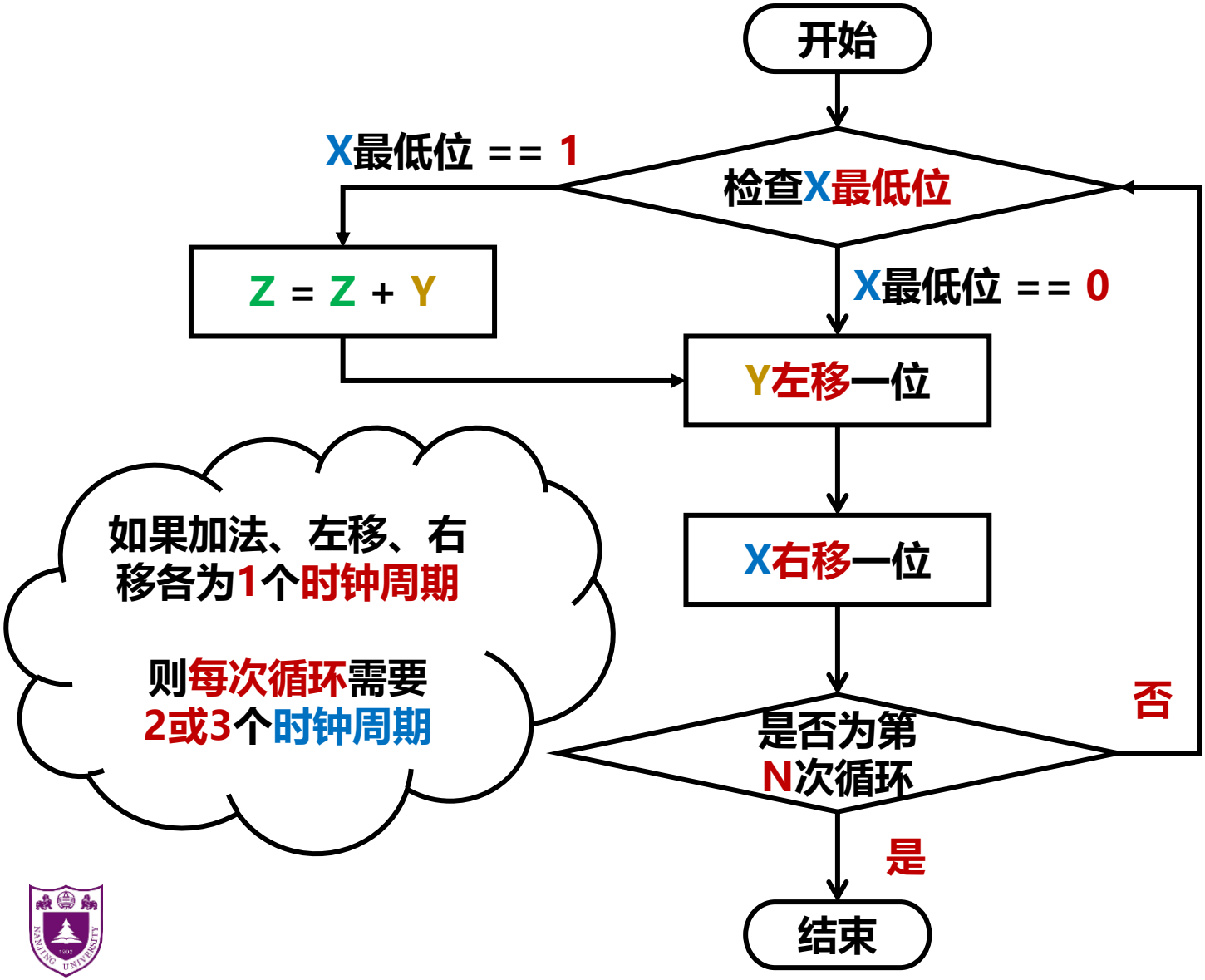
乘法运算过程

```
      0 1 1 1
×     0 1 1 0
-------------
      0 0 0 0
    0 1 1 1
  0 1 1 1
0 0 0 0
-------------
0 1 0 1 0 1 0
```

乘数	被乘数	部分积
0 1 1 0	0 1 1 1	0 0 0 0
0 1 1 **0**	0 1 1 1	0 0 0 0
0 1 **1** 0	0 1 1 1 **0**	**0 1 1 1** 0
0 **1** 1 0	0 1 1 1 **0 0**	**1 0 1 0 1** 0
0 1 1 0	0 1 1 1 **0 0 0**	**0 1 0 1 0 1** 0

	乘数右移一位	被乘数左移一位	中间结果直接与部分积累加
需要：	n位支持右移寄存器	2n位支持左移寄存器	2n位寄存器 2n位加法器

乘法流程图


N位乘法器
乘数寄存器：X
被乘数寄存器：Y
乘积寄存器：Z
开始
检查X最低位
X最低位 == 1
Z = Z + Y
X最低位 == 0
Y左移一位
X右移一位
是否为第
N次循环
否
是
结束
如果加法、左移、右
移各为1个时钟周期
则每次循环需要
2或3个时钟周期

乘法流程优化：加法和移位并行

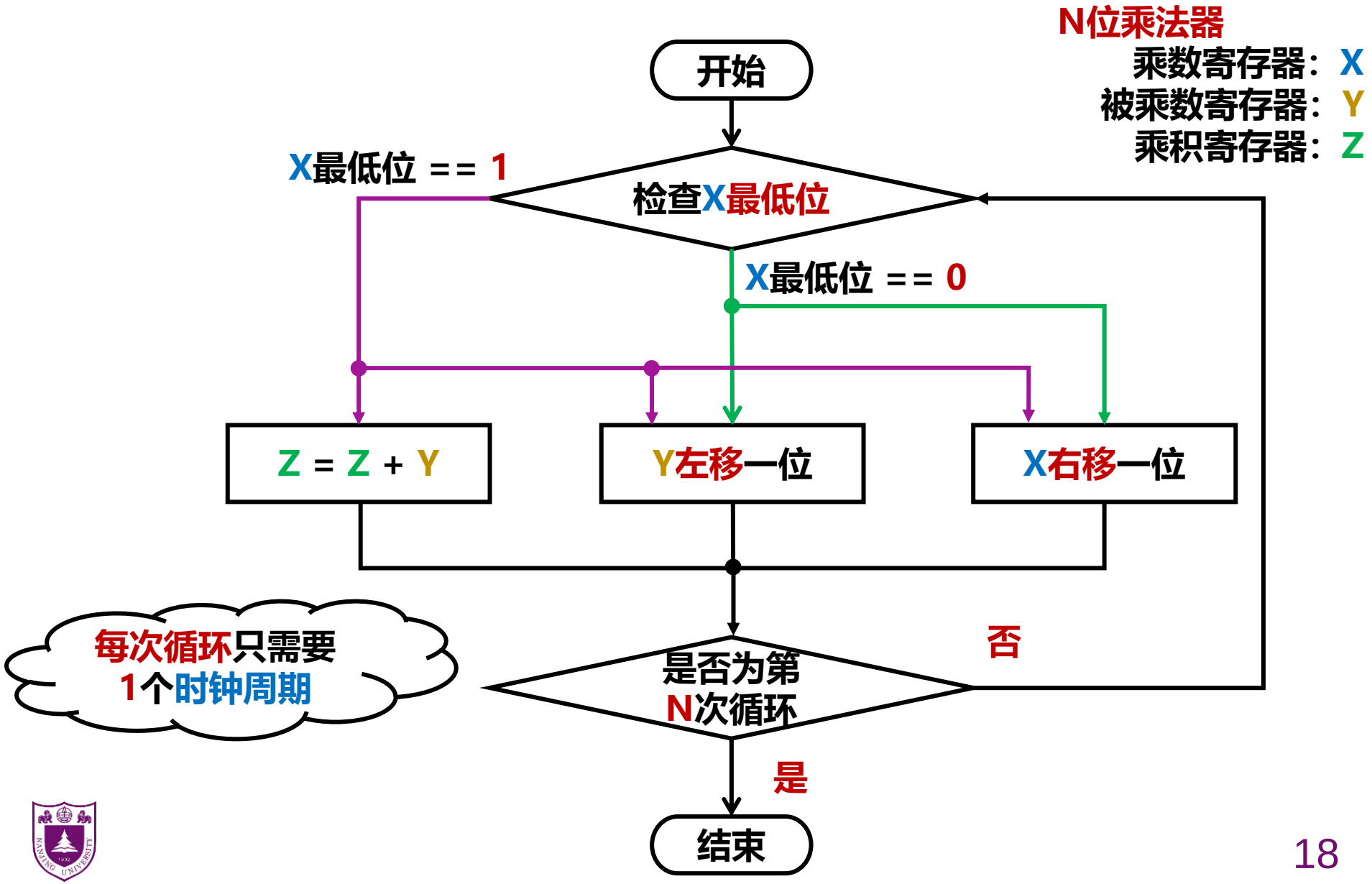
时钟上升沿到来之前

- 各寄存器中的数据被取出
- 寄存器中数据不会发生变化

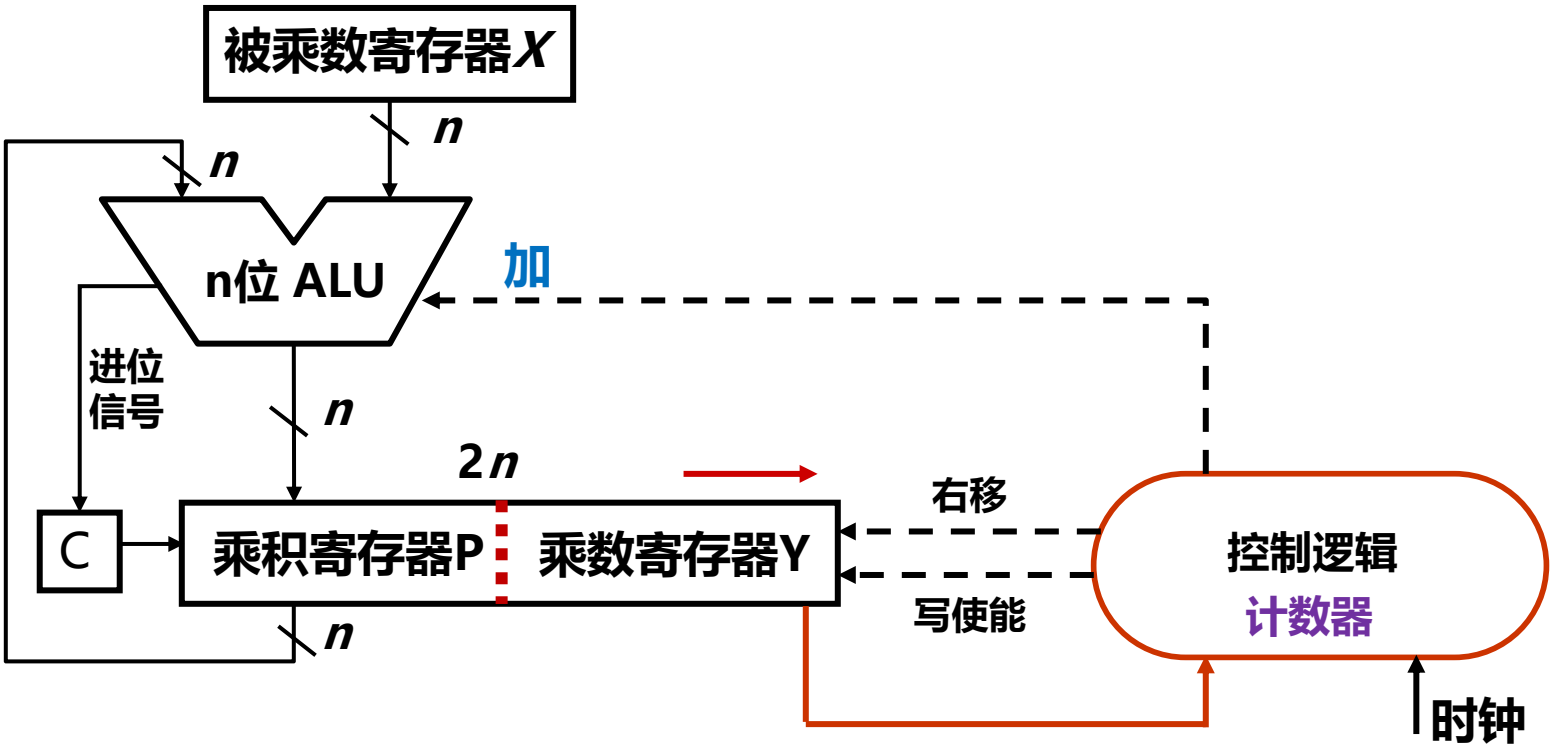
时钟上升沿到来时

- 寄存器根据输入发生变化
- 已经被取出的数据不会发生变化

乘法流程优化：加法和移位并行



乘法器优化：减少不必要的硬件



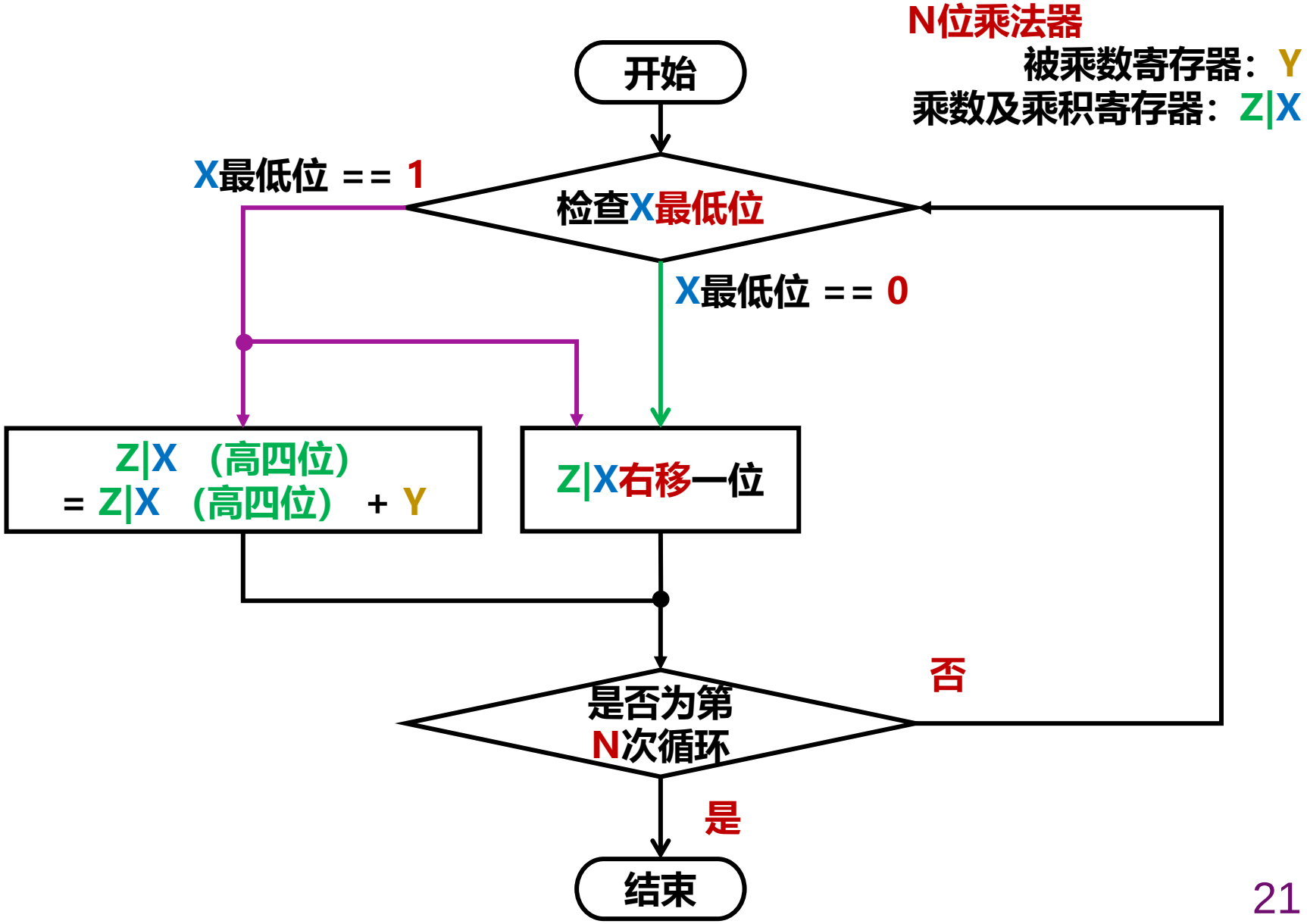
一个n位加法器
一个n位的寄存器：被乘数
一个2n位支持右移的寄存器：乘数和乘积共用

优化的乘法运算过程

```
      0 1 1 1
×     0 1 1 0
-------------
      0 0 0 0
    0 1 1 1
  0 1 1 1
0 0 0 0
-------------
0 1 0 1 0 1 0
```

	寄存器A		寄存器B
	部分积	乘数	被乘数
初始化	00000	0110	0111
A右移，首位补0	00000	0011	0111
A高4位加运算	01110	0011	0111
A右移，首位补0	00111	0001	0111
A高4位加运算	10101	0001	0111
A右移，首位补0	01010	0100	0111
A右移，首位补0	00101	0010	0111

优化的乘法流程图


N位乘法器
被乘数寄存器：Y
乘数及乘积寄存器：Z|X
开始
检查X最低位
X最低位 == 1
X最低位 == 0
Z|X （高四位）
= Z|X （高四位） + Y
Z|X右移一位
是否为第
N次循环
否
是
结束

乘法的问题

7	0111	-7	1001
× 6	× 0110	× -6	× 1010
42	01010100 **42**	42	10110100 **-38**

问题1： $[X \times Y]_c \neq [X]_c \times [Y]_c$

问题2：溢出

原码一位乘法

- 将被乘数和乘数由补码表示改为原码表示
- 符号位和数值位分开运算
- 将乘积结果由原码表示改为补码表示

	符号位	数值位
-7	1	111
× -6	⊕ 1	× 110
42	0	101010

补码一位乘法：布斯算法

$$[X \times Y]_c = [X \times y_n y_{n-1} \cdots y_2 y_1]_c$$

$$= [X \times (-y_n \times 2^{n-1} + y_{n-1} \times 2^{n-2} + \cdots + y_2 \times 2^1 + y_1 \times 2^0)]_c$$

$$= [X \times \begin{pmatrix} -y_n \times 2^{n-1} + y_{n-1} \times (2^{n-1} - 2^{n-2}) + \cdots \\ +y_2 \times (2^2 - 2^1) + y_1 \times (2^1 - 2^0) \end{pmatrix}]_c$$

$$= [X \times \begin{pmatrix} (y_{n-1} - y_n) \times 2^{n-1} + (y_{n-2} - y_{n-1}) \times 2^{n-2} + \cdots \\ +(y_1 - y_2) \times 2^1 + (y_0 - y_1) \times 2^0 \end{pmatrix}]_c \qquad y_0 = 0$$

$$= [2^n \times \sum_{i=0}^{n-1} (X \times (y_i - y_{i+1}) \times 2^{-(n-i)})]_c \qquad = [P_{i+1}]_c$$

$$[P_{i+1}]_c = [2^{-1} \times (P_i + X \times (y_i - y_{i+1}))]_c$$

补码一位乘法：布斯算法

$[X \times Y]_c = 2^n \times [P_n]_c$ **即约定部分积的小数点到最右侧**

$[P_{i+1}]_c = [2^{-1} \times (P_i + X \times (y_i - y_{i+1}))]_c$ **求得$[P_i]_c$后，根据两位即可求得$[P_{i+1}]_c$**

$y_{i+1}y_i = 01$ 则 $[P_{i+1}]_c = [2^{-1} \times (P_i + X)]_c$

$y_{i+1}y_i = 10$ 则 $[P_{i+1}]_c = [2^{-1} \times (P_i - X)]_c$

} 执行$[P_i]_c + [\pm X]_c$然后**右移一位**

$y_{i+1}y_i = 00$ 则 $[P_{i+1}]_c = [2^{-1} \times P_i]_c$

$y_{i+1}y_i = 11$ 则 $[P_{i+1}]_c = [2^{-1} \times P_i]_c$

} **右移一位**

运算步骤：

1. 增加 $y_0 = 0$
2. 根据 $y_{i+1}y_i$决定是否执行$[P_i]_c + [\pm X]_c$
3. 右移部分积
4. 重复步骤 2和步骤 3共 n 次，得到最终结果

布斯乘法运算过程

$$\begin{array}{r} -7 \\ \times\ -6 \\ \hline 42 \end{array}$$

$[X]_c = 1001$
$[-X]_c = 0111$
$[Y]_c = 1010$

i	$y_{i+1}y_i$	操作	寄存器A：部分积	寄存器A：乘数	寄存器B：被乘数
			0 0 0 0	1 0 1 0	1 0 0 1
0	0 0	右移	0 0 0 0 0	1 0 1	1 0 0 1
1	1 0	$[P_i]_c + [-X]_c$	0 1 1 1 0	1 0 1	1 0 0 1
		右移	0 0 1 1 1	0 1 0	1 0 0 1
2	0 1	$[P_i]_c + [X]_c$	1 1 0 0 1	0 1 0	1 0 0 1
		右移	0 1 1 0 0	1 0 1	1 0 0 1
3	1 0	$[P_i]_c + [-X]_c$	1 1 0 1 0	1 0 1	1 0 0 1
		右移	0 1 1 0 1	0 1 0	1 0 0 1

=106?

布斯乘法运算过程

$$\begin{array}{r} -7 \\ \times\ -6 \\ \hline 42 \end{array}$$

$[X]_c = 1001$
$[-X]_c = 0111$
$[Y]_c = 1010$

i	$y_{i+1}y_i$	操作	寄存器A：部分积	寄存器A：乘数	寄存器B：被乘数
			0 0 0 0	1 0 1 0	1 0 0 1
0	0 0	右移	0 0 0 0 0	1 0 1	1 0 0 1
1	1 0	$[P_i]_c + [-X]_c$	0 1 1 1 0	1 0 1	1 0 0 1
		右移	0 0 1 1 1	0 1 0	1 0 0 1
2	0 1	$[P_i]_c + [X]_c$	1 1 0 0 1	0 1 0	1 0 0 1
		右移	1 1 1 0 0	1 0 1	1 0 0 1
3	1 0	$[P_i]_c + [-X]_c$	0 1 0 1 0	1 0 1	1 0 0 1
		右移	0 0 1 0 1	0 1 0	1 0 0 1

=42

补的第n位 = 原第n位

乘法溢出

对于带符号整数：

- 当$-2^{n-1} \leq x \cdot y \leq 2^{n-1}-1$时**不溢出**
- **即当乘积的高n位全0或全1，并等于低n位的最高位时，不溢出**

对于无符号整数：

- 当$0 \leq x \cdot y \leq 2^{n}-1$时**不溢出**
- **当乘积的高n位全0，不溢出**

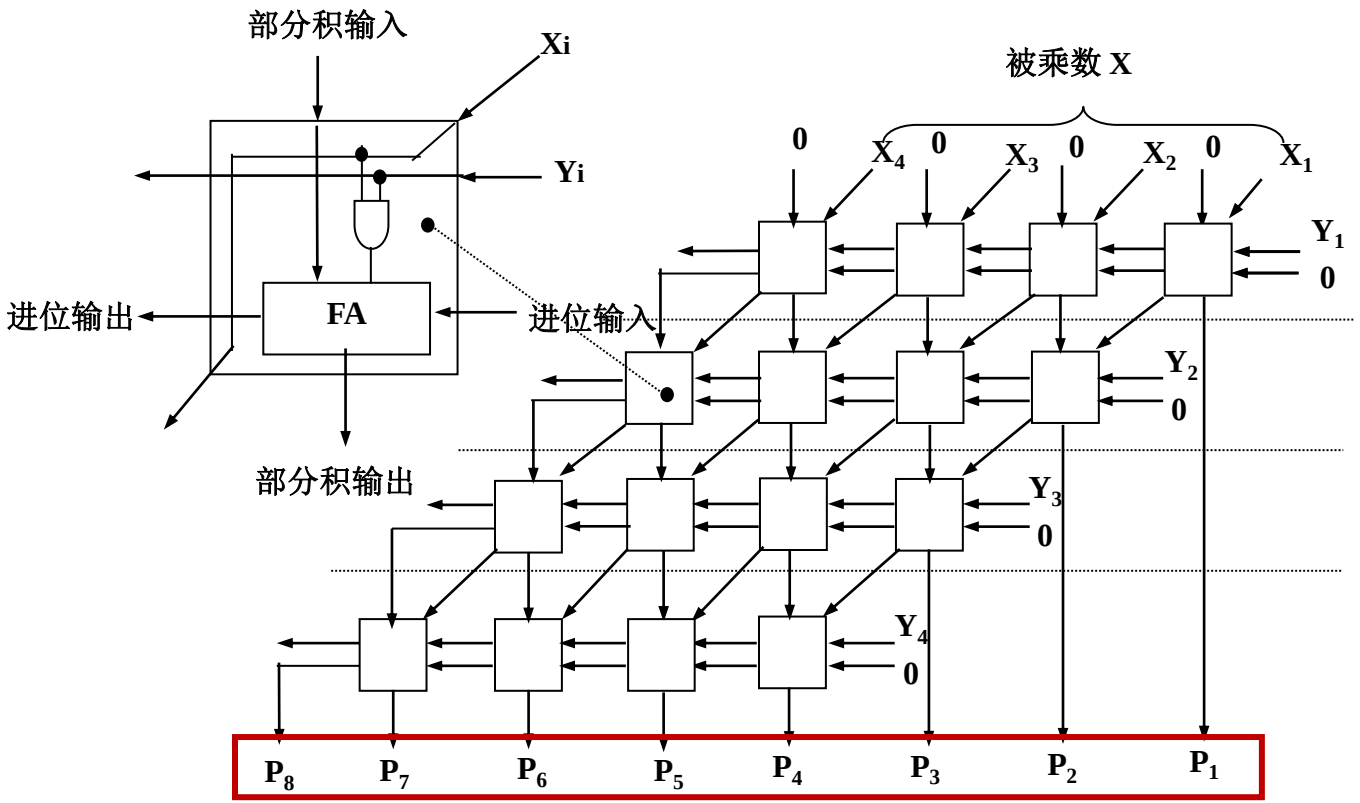
带符号整数和无符号整数的溢出判断不同 ⟶ 分无符号数乘指令和带符号数乘指令

硬件不判断溢出：寄存器会存2n位乘积

软件判断溢出：1）编译器判断溢出；2）程序员通过高级语言判断溢出

阵列乘法器

$$
\begin{array}{r}
0\,1\,1\,1 \\
\times\quad 0\,1\,1\,0 \\
\hline
0\,0\,0\,0 \\
0\,1\,1\,1 \\
0\,1\,1\,1 \\
0\,0\,0\,0 \\
\hline
0\,1\,0\,1\,0\,1\,0
\end{array}
$$



除法

不同情形的处理

- 若被除数为0，除数不为0：商为0

  例如：$0 \div 1 = 0$

- 若被除数不为0，除数为0：发生“除数为0”异常

  例如：$1 \div 0 = NaN$

- 若被除数、除数均为0：发生“除法错”异常

  例如：$0 \div 0 = NaN$

- 若被除数、除数均不为0：进行进一步除法运算

  例如：$1 \div 1 = ?$

除法

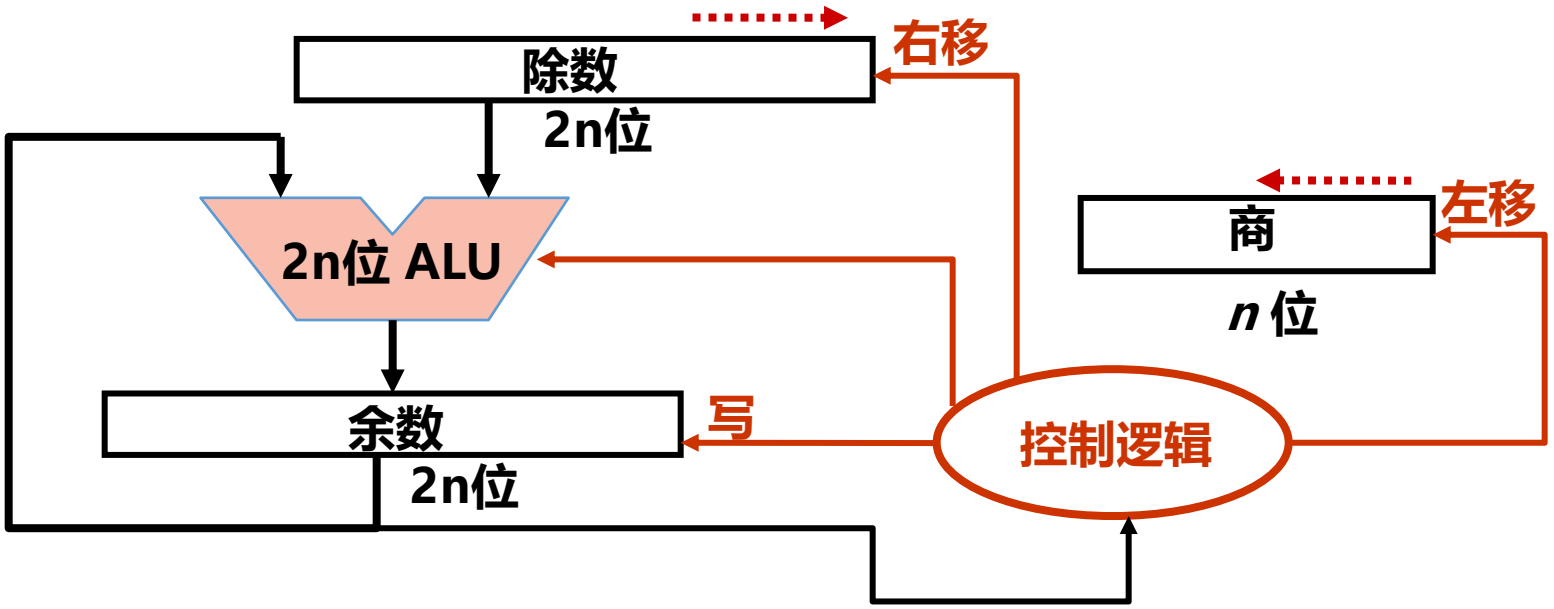
手工演算除法

- 在被除数的左侧补充符号位，将除数的最高位与被除数的次高位对齐
- 从被除数中减去除数，若够减，则上商为1；若不够减，则上商为0
- 右移除数，重复上述步骤

```
    2
3 ) 7
    6
    1
```

```
          0010
0011 ) 00000111
       0000
       ----
        0001
      - 0000
      ------
         0011
      -  0011
      -------
          0001
      -  0000
      -------
          0001
```

除法器



一个2n位加法器
一个2n位寄存器：被除数/余数
一个2n位支持右移的寄存器：除数
一个n位支持左移的寄存器：商

除法运算过程

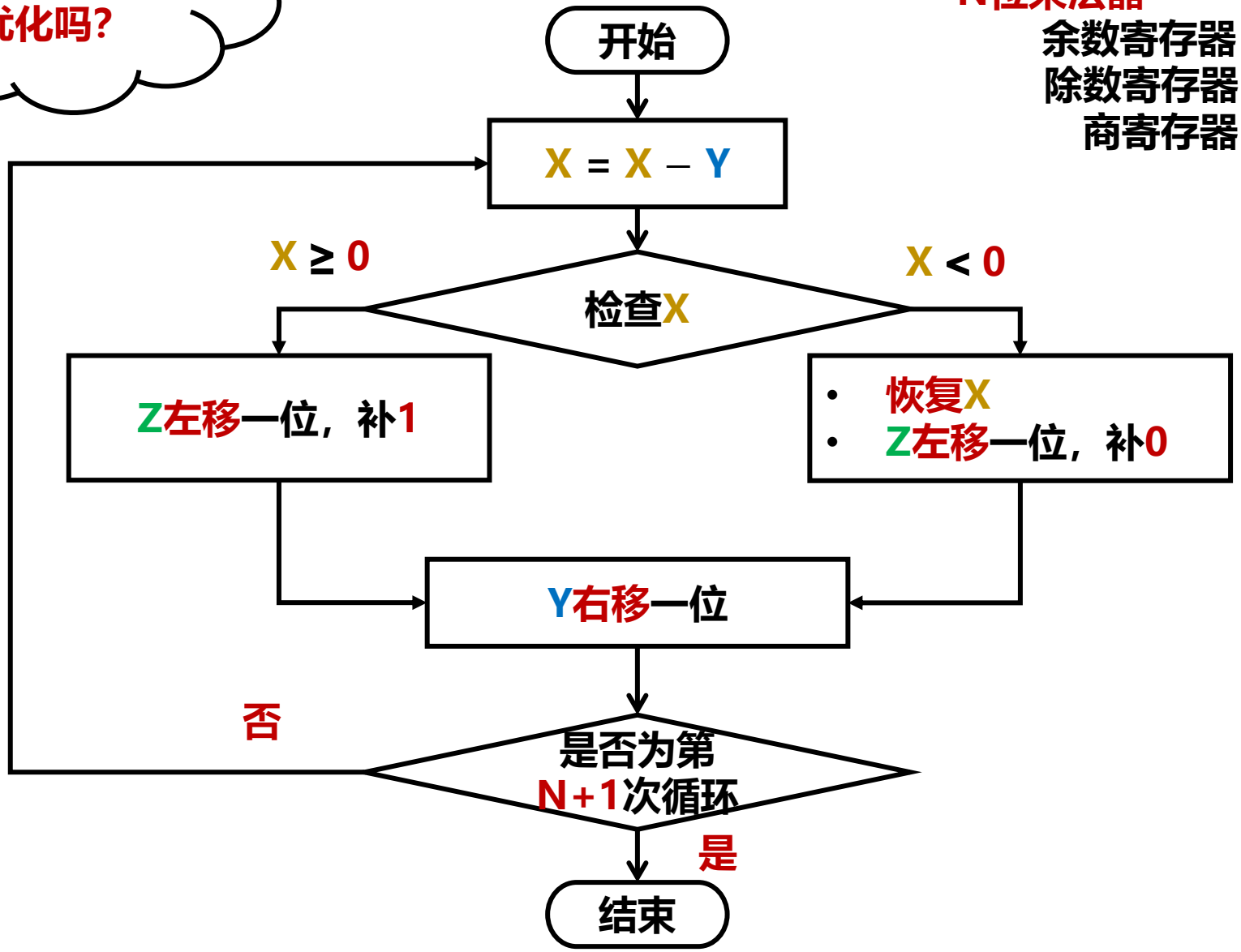
```
          0010
0011 ) 00000111
       0000
       ----
        0001
      - 0000
       ----
         0011
       - 0011
       ------
         0001
       - 0000
       ------
         0001
```

	寄存器X	寄存器Y	寄存器Z
	余数	除数	商
初始化	00000111	00110000	0000
余数<除数	00000111	00011000	0000
Y右移	00000111	00011000	0000
余数<除数	00000111	00001100	0000
Y右移	00000111	00001100	0000
余数>除数	00000111	00000110	0000
X-Y；Y右移；Z左移补1	00000001	00000110	0001
余数<除数	00000001	00000011	0001
Z左移补0	00000001	00000011	0010

除法流程图

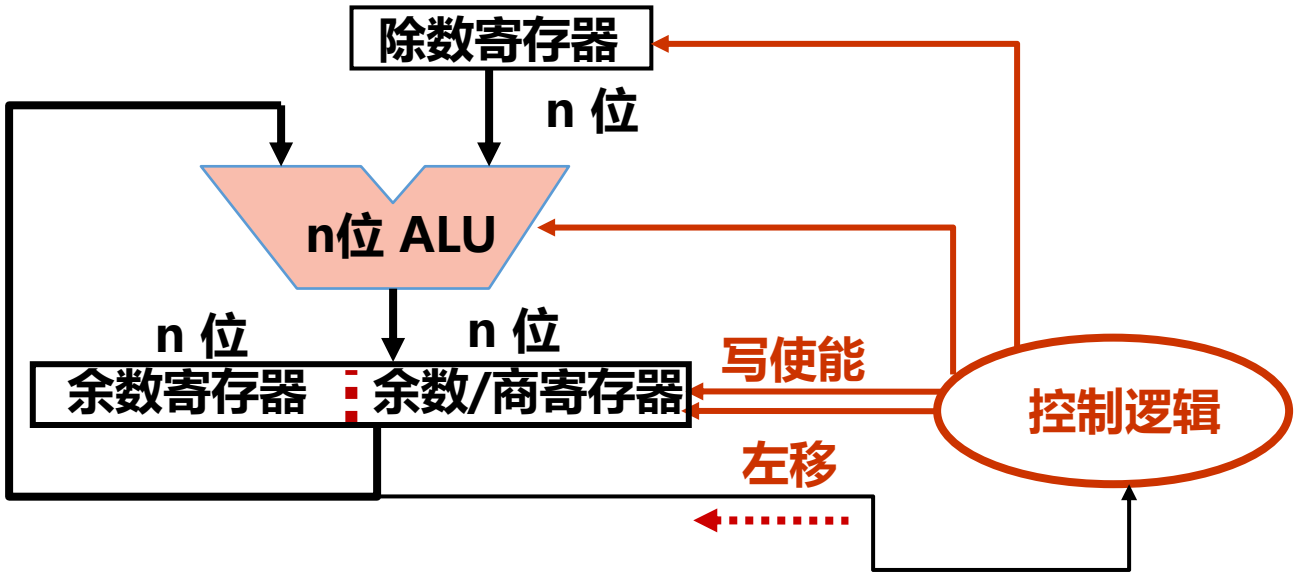
能像乘法一样优化吗?

N位乘法器
余数寄存器：X
除数寄存器：Y
商寄存器：Z



优化的除法器

- 余数和除数的减法运算中，实际上只有n位参与了运算
- 余数和除数寄存器中，至少有一个需要支持左移或右移
- 商寄存器必须支持左移，且只需要n位



一个n位加法器
一个n位支持左移的寄存器：被除数/余数
一个n位支持左移的寄存器：余数/商
一个n位寄存器：除数

异号的除法

```
      -2
   ┌────
3  │ -7
     -6
   ────
     -1
```

```
                1110
                 ↑
               0010
          ┌─────────
0011      │11111001
        + 0000
        ─────────
          111001
        + 0000
        ─────────
           11001
        +  0011
        ─────────
            1111
        +  0000
        ─────────
            1111
             ↓
           1111
```

异号，所以商需要取补码，余数本来就是补码

比较余数和除数

- 如何判断“**够减**”：余数是否足够“大”
  - 如果余数和除数的符号相同：**减法**
  - 如果余数和除数的符号不同：**加法**

中间余数 R	除数 Y	减法		加法	
		0	1	0	1
0	0	够	不够	----	----
0	1	----	----	够	不够
1	0	----	----	不够	够
1	1	不够	够	----	----

余数减除数后：1）绝对值变小；2）符号不能变

补码除法运算过程

```
          1110
           ↑
          0010
      ┌─────────
0011  │ 11111001
    + 0000
    ──────────
        111001
      + 0000
      ────────
         11001
       + 0011
       ──────
          1111
        + 0000
        ──────
          1111
           ↓
          1111
```

$-7 \div 3 = -2 \ldots -1$

初始化

X和Z左移，空1位

余数+除数 （变号）

恢复余数、补商

X和Z左移，空1位

余数+除数 （变号）

恢复余数、补商

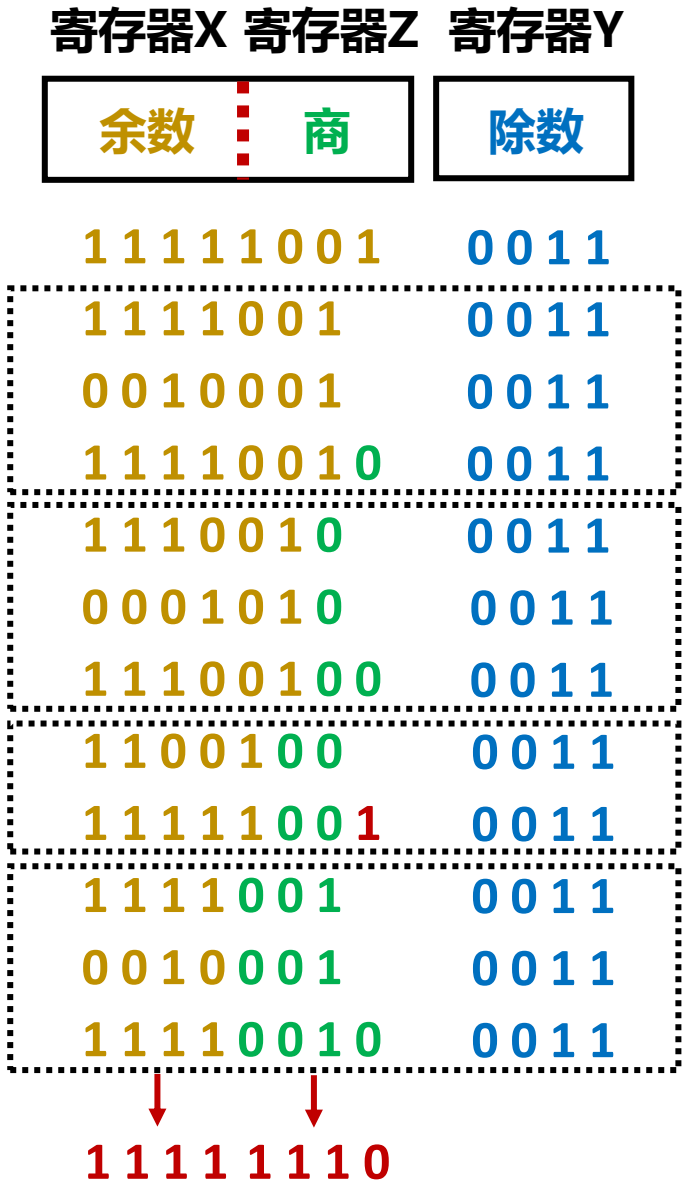
X和Z左移，空1位

余数+除数 （未变号）

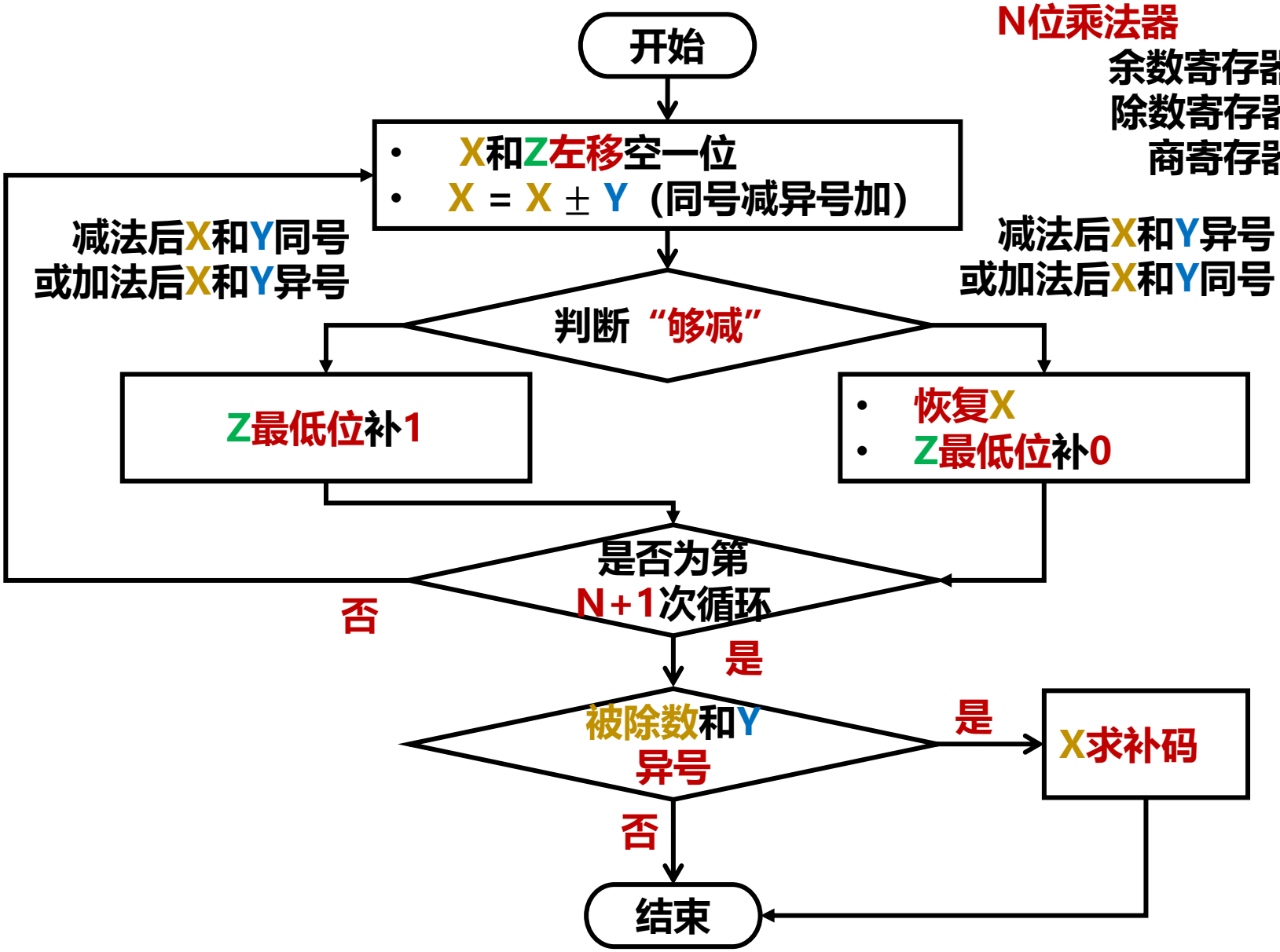
X和Z左移，空1位

余数+除数 （变号）

恢复余数、补商



补码除法流程图


N位乘法器
余数寄存器：X
除数寄存器：Y
商寄存器：Z
开始
X和Z左移空一位
X = X ± Y (同号减异号加)
减法后X和Y同号
或加法后X和Y异号
减法后X和Y异号
或加法后X和Y同号
判断“够减”
Z最低位补1
恢复X
Z最低位补0
是否为第
N+1次循环
否
是
被除数和Y
异号
是
X求补码
否
结束

不恢复余数除法

问题：恢复余数成本高

大致思路：不恢复余数

- **只考虑减法**
  - 如果**余数 R_i**足够大

$$R_{i+1} = 2R_i - Y$$

  - 如果**余数 R_i**不够大

$$R_{i+1} = 2(R_i + Y) - Y = 2R_i + Y$$

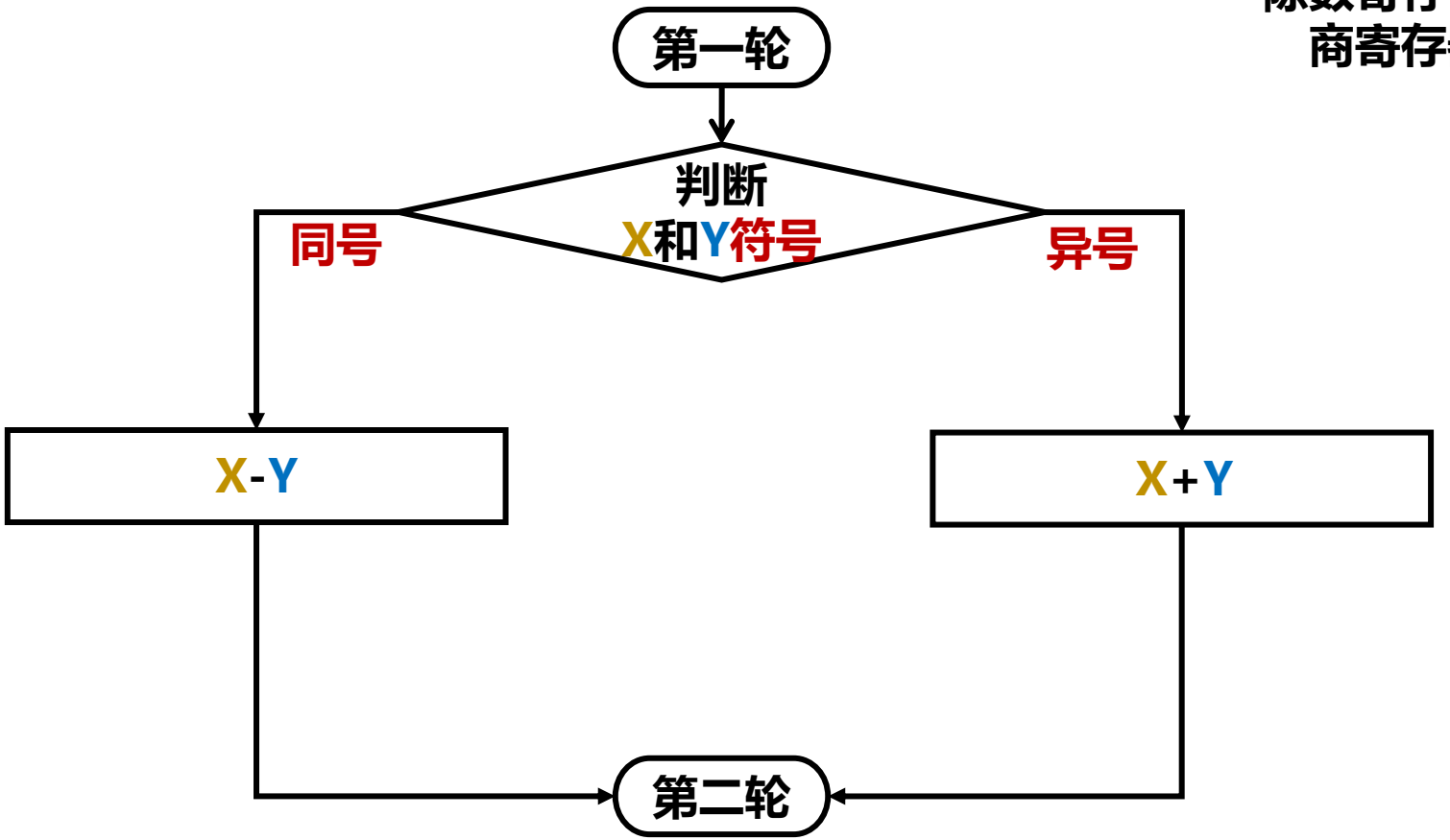
补码不恢复余数除法流程图 1

N位乘法器

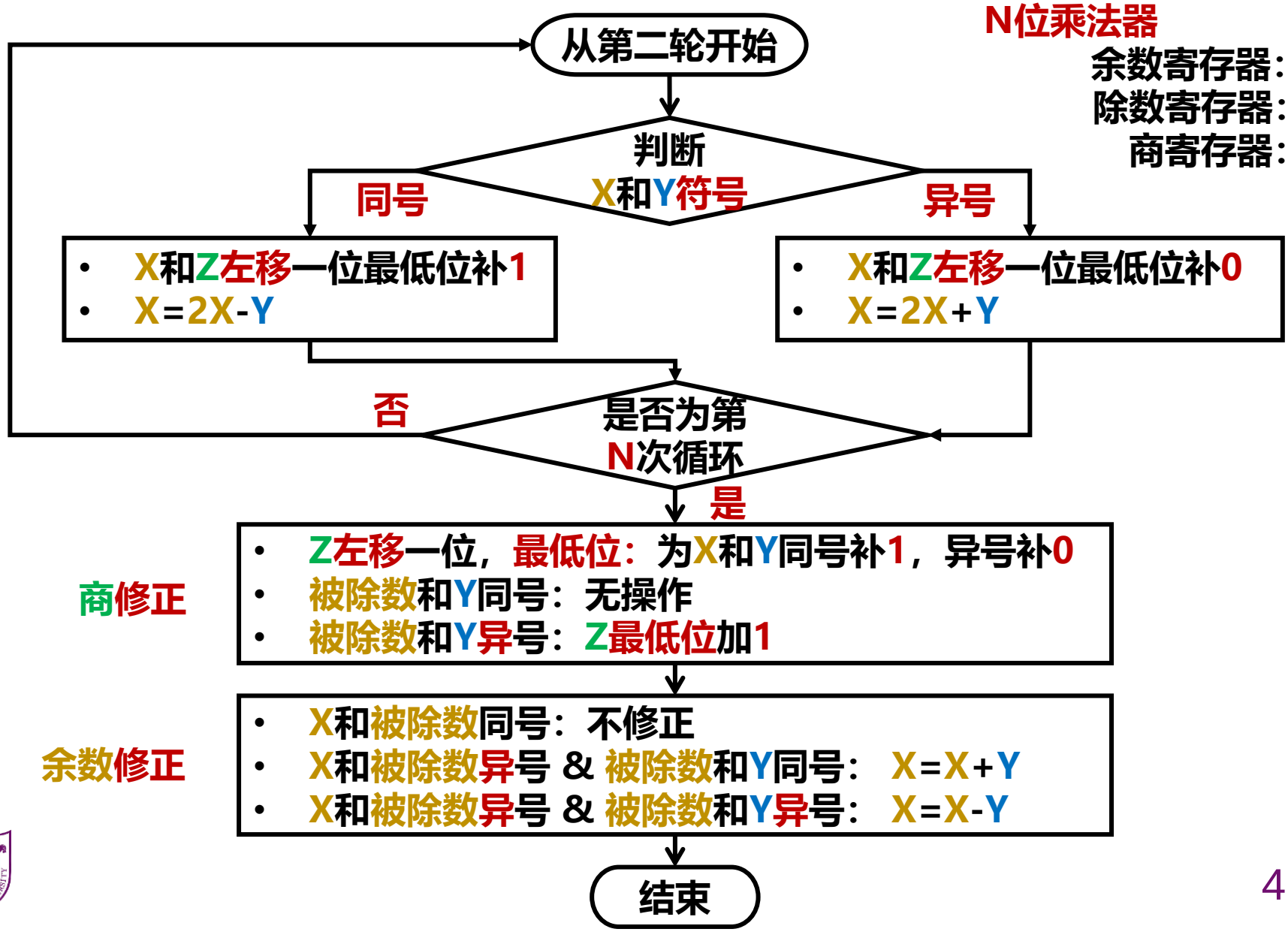
余数寄存器：X

除数寄存器：Y

商寄存器：Z



补码不恢复余数除法流程图 2



不恢复余数补码除法运算过程

```
          1110
           ↑
          0010
      ┌─────────
0011  ╱ 11111001
    + 0000
    ──────────
        111001
      + 0000
      ────────
         11001
       + 0011
       ──────
          1111
        + 0000
        ──────
          1111
           ↓
          1111
```

$-7 \div 3 = -2 \ldots -1$

	寄存器X 余数	寄存器Z 商	寄存器Y 除数
初始化	1 1 1 1 1 0 0 1		0 0 1 1
余数+除数（同号）	0 0 1 0 1 0 0 1		0 0 1 1
X和Z左移，末位补1	0 1 0 1 0 0 1 1		0 0 1 1
2余数-除数（同号）	0 0 1 0 0 0 1 1		0 0 1 1
X和Z左移，末位补1	0 1 0 0 0 1 1 1		0 0 1 1
2余数-除数（同号）	0 0 0 1 0 1 1 1		0 0 1 1
X和Z左移，末位补1	0 0 1 0 1 1 1 1		0 0 1 1
2余数-除数（异号）	1 1 1 1 1 1 1 1		0 0 1 1
X和Z左移，末位补0	1 1 1 1 1 1 1 0		0 0 1 1
2余数+除数（同号）	0 0 1 0 1 1 1 0		0 0 1 1
商修正、余数修正	1 1 1 1 1 1 1 0		0 0 1 1

余数和被除数异号 &被除数和除数异号： 余数-除数

Z左移一位补1； 被除数和除数异号： Z末位加1

其他

只有一种情况发生溢出：

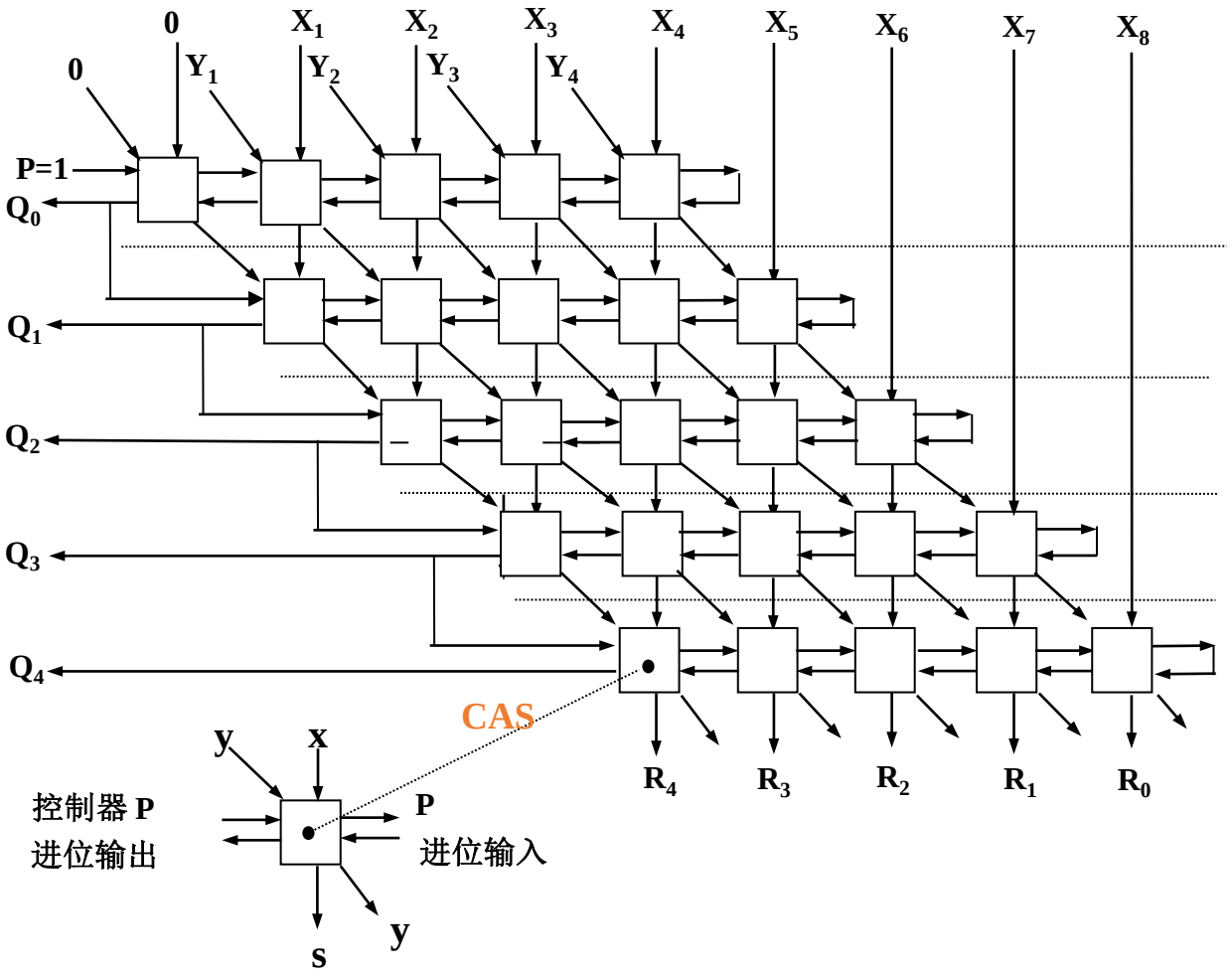
- 当$\frac{-2^{n-1}}{-1} = 2^{n-1}$时

编译器处理一个变量与2^n 相除时，一般采用右移运算实现：

- **无符号**：逻辑右移
- **带符号**：算术右移
- **能整除时**：直接右移得到结果，被移出的全为0
- **不能整除时**：被移出数存在非0，采取**朝零舍入**
  - **无符号**：直接右移得到结果，移出的低位直接舍弃
  - **带符号**：加偏移量$2^k - 1$，然后再移k位，低位截断

例：14/4=3 0000 1110 >> 0000 0011
-14/4=-3 1111 0010 + 0000 0011 = 1111 0101 >> 1111 1101

阵列除法器


0
X_1
X_2
X_3
X_4
X_5
X_6
X_7
X_8
0
Y_1
Y_2
Y_3
Y_4
P=1
Q_0
Q_1
Q_2
Q_3
Q_4
CAS
R_4
R_3
R_2
R_1
R_0
y
x
控制器 P
进位输出
P
进位输入
y
s

总结

- 算术逻辑单元（ALU）
  - 全加器
  - 串行进位加法器，全先行进位加法器，部分先行进位加法器
- 补码表示的整数运算
  - 加法：溢出判定
  - 减法：硬件实现
  - 乘法：布斯算法
  - 除法：恢复余数 / 不恢复余数

谢谢

bohanliu@nju.edu.cn

南京大學
NANJING UNIVERSITY