



第6章 运算方法和运算部件

第一讲 基本运算部件

第二讲 定点数运算

第三讲 浮点数运算*

第一讲 基本运算部件

1. 高级语言程序中所涉及到的运算（以C语言为例）

1. 整数算数运算、浮点数算数运算
2. 按位、逻辑、移位、位扩展和位截断

2. 串行进位加法器

3. 并行进位加法器

1. 全先行进位加法器
2. 两级/多级先行进位加法器

4. 带标志加法器

5. 算数逻辑部件（ALU）

如何实现高级语言源程序中的运算？

- C语言程序中的基本数据类型及其基本运算类型
 - 基本数据类型 无符号数、带符号整数、浮点数、位串、字符（串）
 - 基本运算类型 算术、按位、逻辑、移位、扩展和截断、匹配
- 计算机如何实现高级语言程序中的运算？
 - 将各类表达式编译（转换）为指令序列
 - 计算机直接执行指令来完成运算

例：C语句 “ $f = (g+h) - (i+j);$ ” 中变量 i 、 j 、 g 、 h 由编译器分别分配在RISC-V寄存器 $t3 \sim t6$ 中，寄存器 $t3 \sim t6$ 的编号对应 $28 \sim 31$ ， f 分配在 $t0$ （编号5），则对应的RISC-V机器代码和汇编表示（#后为注释）如下：

```
0000000 11111 11110 000 00101 0110011  add t0, t5, t6  # g+h
0000000 11101 11100 000 00110 0110011  add t1, t3, t4  # i+j
0100000 00110 00101 000 00101 0110011  sub t0, t0, t1  # f=(g+h) - (i+j)

func7    rs2    rs1    func3 rd    opcode
```

需要提供哪些运算类指令才能支持高级语言需求呢？

数据的运算

- 高级语言程序中涉及的运算（以C语言为例）

整数算术运算、浮点数算术运算

按位、逻辑、移位、位扩展和位截断

逻辑运算、移位、扩展和截断等指令实现较容易，算术运算指令实现较难！

- 指令集中涉及的运算（如RISC-V指令系统提供的运算类指令）

□ 涉及的定点数运算

算术运算

- 带符号整数：取负 / 符号扩展 / 加 / 减 / 乘 / 除 / 算术移位
- 无符号整数：0扩展 / 加 / 减 / 乘 / 除

逻辑运算

- 逻辑操作：与 / 或 / 非 / ...
- 移位操作：逻辑左移 / 逻辑右移

□ 涉及的浮点数运算：加、减、乘、除

以下介绍基本运算部件：加法器（串行→并行）→ 带标志加法器 → ALU

第一讲 基本运算部件

1. 高级语言程序中所涉及到的运算（以C语言为例）

1. 整数算数运算、浮点数算数运算
2. 按位、逻辑、移位、位扩展和位截断

2. 串行进位加法器

3. 并行进位加法器

1. 全先行进位加法器
2. 两级/多级先行进位加法器

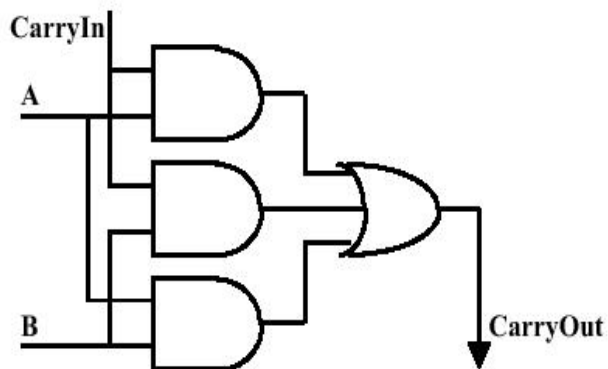
4. 带标志加法器

5. 算数逻辑部件（ALU）

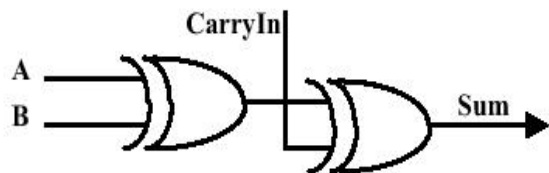
串行进位加法器

CarryOut 和 Sum 的逻辑图

° $\text{CarryOut} = B \& \text{CarryIn} \mid A \& \text{CarryIn} \mid A \& B$

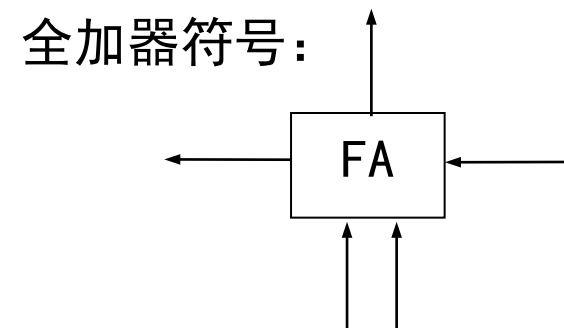


° $\text{Sum} = A \text{ XOR } B \text{ XOR } \text{CarryIn}$

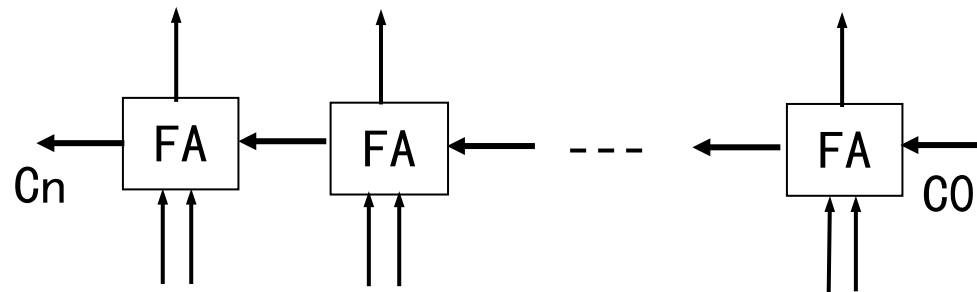


假定与/或门延迟为 $1t_y$ ，异或门 $3t_y$ ，则“和”与“进位”的延迟为多少？

Sum延迟为 $6t_y$ ；Carryout延迟为 $2t_y$ 。



n位串行(行波)加法器：



串行加法器的缺点：进位按串行方式传递，速度慢！

问题：n位串行加法器从 C_0 到 C_n 的延迟时间为多少？ $2n$ 级门延迟！

最后一位和数的延迟时间为多少？

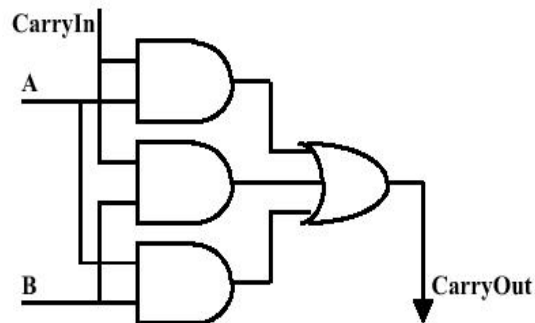
$2n+1$ 级门延迟！

$n > 2$

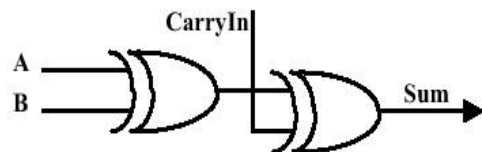
串行进位加法器

CarryOut 和 Sum 的逻辑图

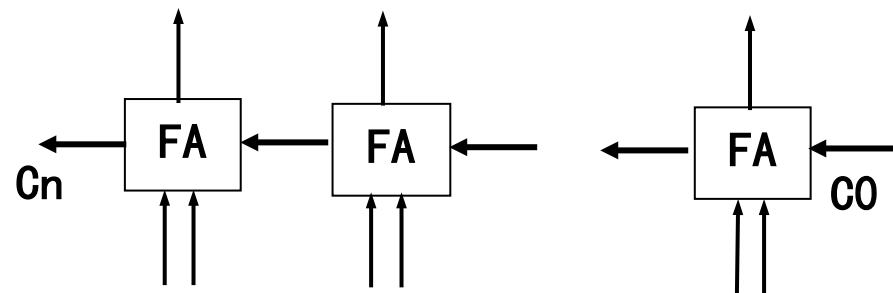
◦ $\text{CarryOut} = B \& \text{CarryIn} \mid A \& \text{CarryIn} \mid A \& B$



◦ $\text{Sum} = A \text{ XOR } B \text{ XOR } \text{CarryIn}$



n位串行(行波)加法器:



4位串行进位加法器CRA的实现

```
module FA (  
    input x, y, cin,  
    output f, cout  
);  
    assign f = x ^ y ^ cin;  
    assign cout = (x & y) | (x & cin) |  
    (y & cin);  
endmodule  
  
module CRA (  
    input [3:0] x, y,  
    input cin,  
    output [3:0] f,  
    output cout  
);  
    wire [4:0] c;  
    assign c[0] = cin;  
    FA fa0(x[0], y[0], c[0], f[0], c[1]);  
    FA fa1(x[1], y[1], c[1], f[1], c[2]);  
    FA fa2(x[2], y[2], c[2], f[2], c[3]);  
    FA fa3(x[3], y[3], c[3], f[3], c[4]);  
    assign cout = c[4];  
endmodule
```

第一讲 基本运算部件

1. 高级语言程序中所涉及到的运算（以C语言为例）
 1. 整数算数运算、浮点数算数运算
 2. 按位、逻辑、移位、位扩展和位截断
2. 串行进位加法器
3. 并行进位加法器
 1. 全先行进位加法器
 2. 两级/多级先行进位加法器
4. 带标志加法器
5. 算数逻辑部件（ALU）

并行进位加法器 (CLA加法器)

- 为什么用先行进位方式？

串行进位加法器采用串行逐级传递进位，电路延迟与位数成正比关系，**太慢了**。因此，现代计算机采用一种先行进位 (Carry look ahead) 方式。

- 如何产生先行进位？

定义辅助函数： $G_i = A_i B_i \cdots$ 进位生成函数

$P_i = A_i + B_i \cdots$ 进位传递函数 (或 $P_i = A_i \oplus B_i$)

通常把实现上述逻辑的电路称为进位生成/传递部件

全加逻辑方程： $S_i = A_i \oplus B_i \oplus C_{i-1}$ $C_i = G_i + P_i C_{i-1}$ ($i=1, \cdots, n$)

设 $n=4$, 则： $C_1 = G_0 + P_0 C_0$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

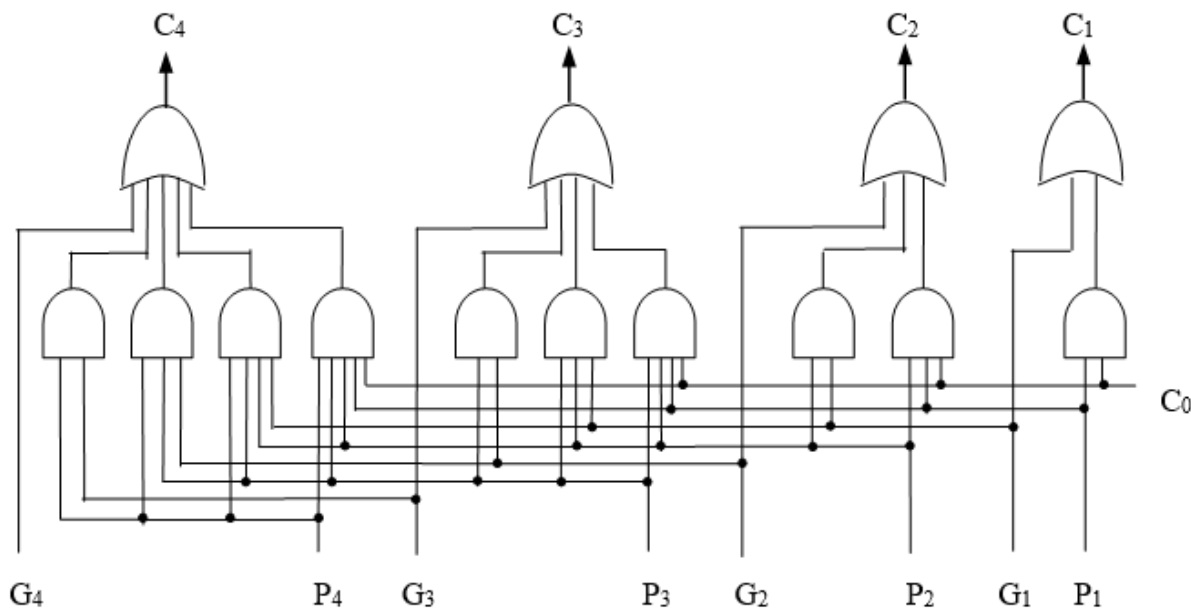
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

$$C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$$

由上式可知: 各进位之间无等待，可以独立并同时产生。

通常把实现上述逻辑的电路称为4位CLU部件

并行进位加法器 (CLA加法器)



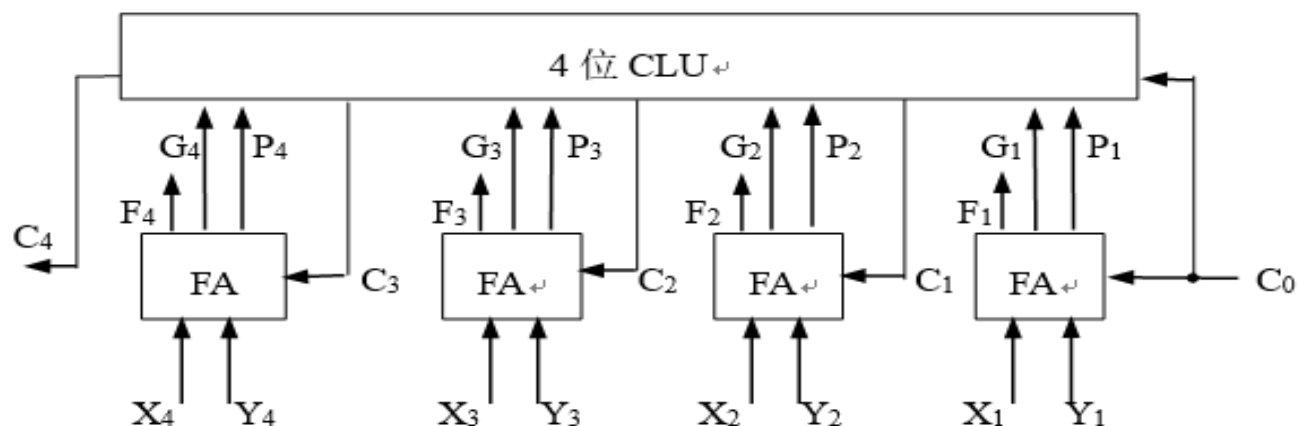
$$C_1 = G_1 + P_1 C_0$$

$$C_2 = G_2 + P_2 C_1 = G_2 + P_2 G_1 + P_2 P_1 C_0$$

$$C_3 = G_3 + P_3 C_2 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 C_0$$

$$C_4 = G_4 + P_4 C_3 = G_4 + P_4 G_3 + P_4 P_3 G_2 + P_4 P_3 P_2 G_1 + P_4 P_3 P_2 P_1 C_0$$

4位CLU部件



$$G_i = A_i B_i$$

$$P_i = A_i + B_i \quad (\text{或 } P_i = A_i \oplus B_i)$$

$$F_i = A_i \oplus B_i \oplus C_{i-1}$$

4位CLA加法器

并行进位加法器 (CLA加法器)

$$G_i = A_i B_i$$

$$P_i = A_i + B_i \quad (\text{或} \quad P_i = A_i \oplus B_i)$$

$$F_i = A_i \oplus B_i \oplus C_{i-1}$$

```
module FA_PG (  
    input x, y, cin,  
    output f, p, g );  
    assign f = x ^ y ^ cin;  
    assign p = x | y;  
    assign g = x & y;
```

```
endmodule
```

```
module CLU (  
    input [4:1] p, g,  
    input c0,  
    output [4:1] c );  
    assign c[1] = g[1] | (p[1] & c0);  
    assign c[2] = g[2] | (p[2] & g[1]) | (p[2] & p[1] & c0);
```

// 以下两个表达式使用了位拼接运算和归约运算

```
    assign c[3] = g[3] | (p[3] & g[2]) | (&{p[3:2], g[1]}) | (&{p[3:1], c0});  
    assign c[4] = g[4] | (p[4] & g[3]) | (&{p[4:3], g[2]}) | (&{p[4:2], g1}) | (&{p[4:1], c0});  
endmodule
```

```
module CLA (  
    input [3:0] x, y,  
    input cin,  
    output [3:0] f,  
    output cout );  
    wire [4:0] c;  
    wire [4:1] p, g;  
    assign c[0] = cin;  
    FA_PG fa0(x[0], y[0], c[0], f[0], p[1], g[1]);  
    FA_PG fa1(x[1], y[1], c[1], f[1], p[2], g[2]);  
    FA_PG fa2(x[2], y[2], c[2], f[2], p[3], g[3]);  
    FA_PG fa3(x[3], y[3], c[3], f[3], p[4], g[4]);  
    CLU clu(p, g, c[0], c[4:1]);  
    assign cout = c[4];  
endmodule
```

$$C_1 = G_1 + P_1 C_0$$

$$C_2 = G_2 + P_2 C_1 = G_2 + P_2 G_1 + P_2 P_1 C_0$$

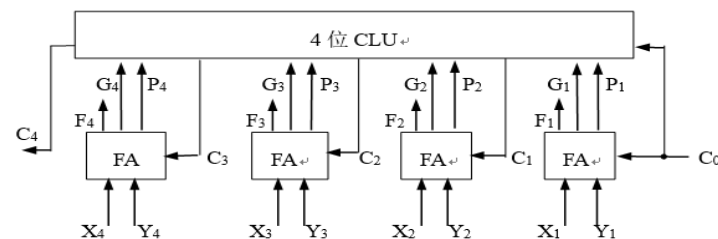
$$C_3 = G_3 + P_3 C_2 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 C_0$$

$$C_4 = G_4 + P_4 C_3 = G_4 + P_4 G_3 + P_4 P_3 G_2 + P_4 P_3 P_2 G_1 + P_4 P_3 P_2 P_1 C_0$$

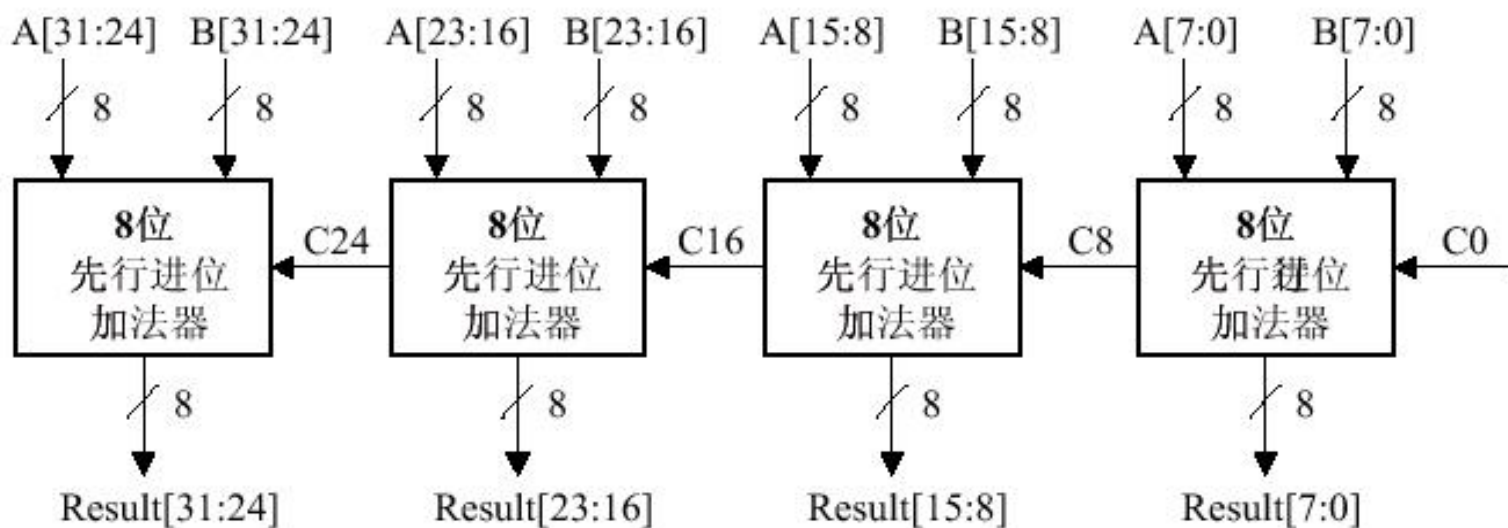
局部（单级）先行进位加法器

局部先行进位加法器 (Partial Carry Lookahead Adder)
或称 单级先行进位加法器

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



问题：所有和数产生的延迟为多少？ $3+2+2+5=12t_y$



多级先行进位加法器

- 单级(局部)先行进位加法器的进位生成方式：
“组内并行、组间串行”
- 所以，单级先行进位加法器虽然比行波加法器延迟时间短，但高位组进位依赖低位组进位，故仍有较长的时间延迟
- 通过引入组进位生成/传递函数实现“组内并行、组间并行”进位方式

设 $n=4$, 则: $C_1=G_1+P_1C_0$

$$C_2=G_2+P_2C_1=G_2+P_2G_1+P_2P_1C_0$$

$$C_3=G_3+P_3C_2=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1C_0$$

$$C_4=G_4+P_4C_3=G_4+P_4G_3+P_4P_3G_2+P_4P_3P_2G_1 + P_4P_3P_2P_1C_0$$

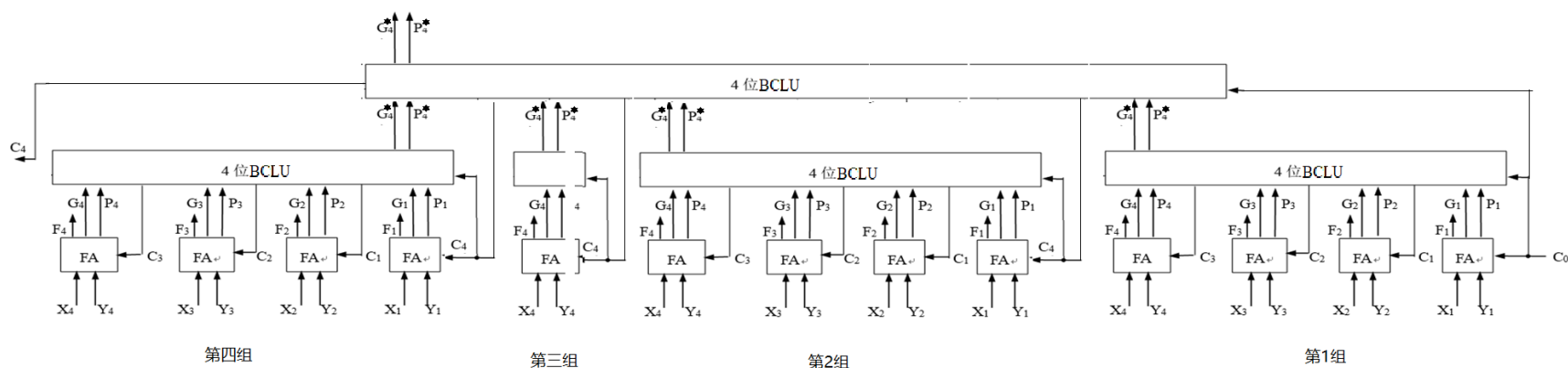
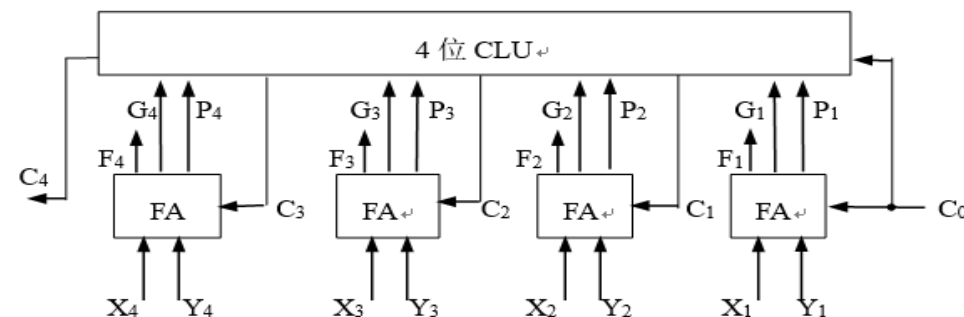
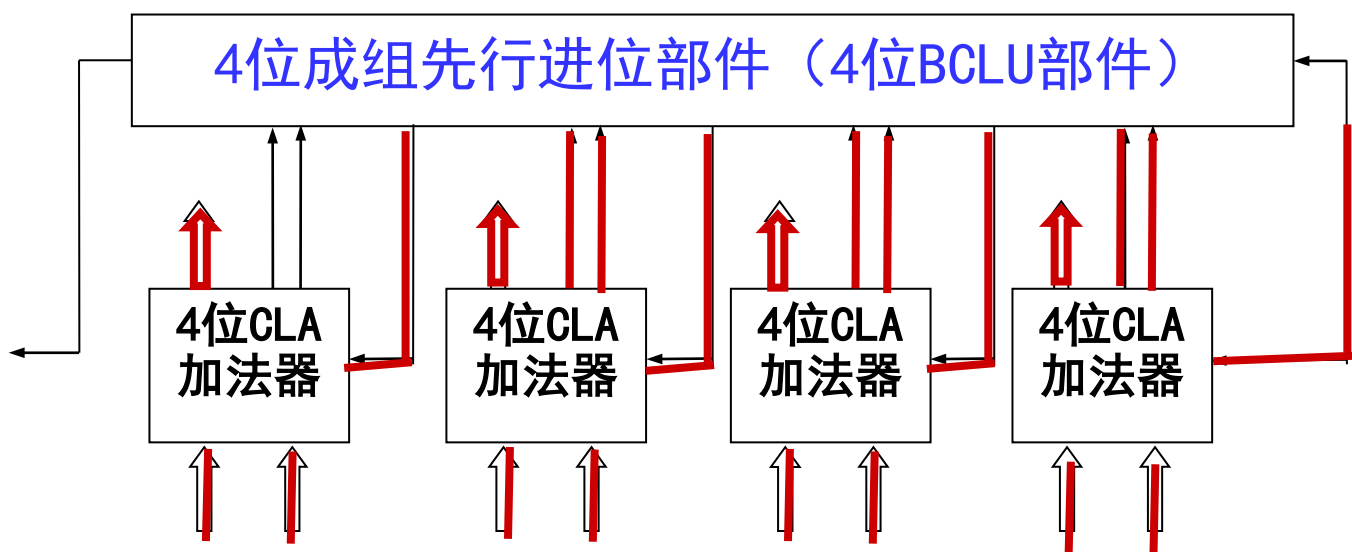
$$G_4^*=G_4+P_4C_3=G_4+P_4G_3+P_4P_3G_2+P_4P_3P_2G_1$$

$$P_4^*=P_4P_3P_2P_1$$

所以 $C_4 = G_4^*+P_4^*C_0$ 。把实现上述逻辑的电路称为4位BCLU (Block CLU) 部件。

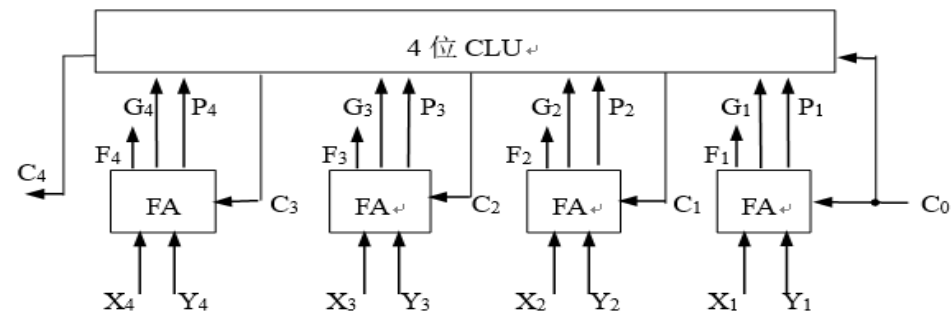
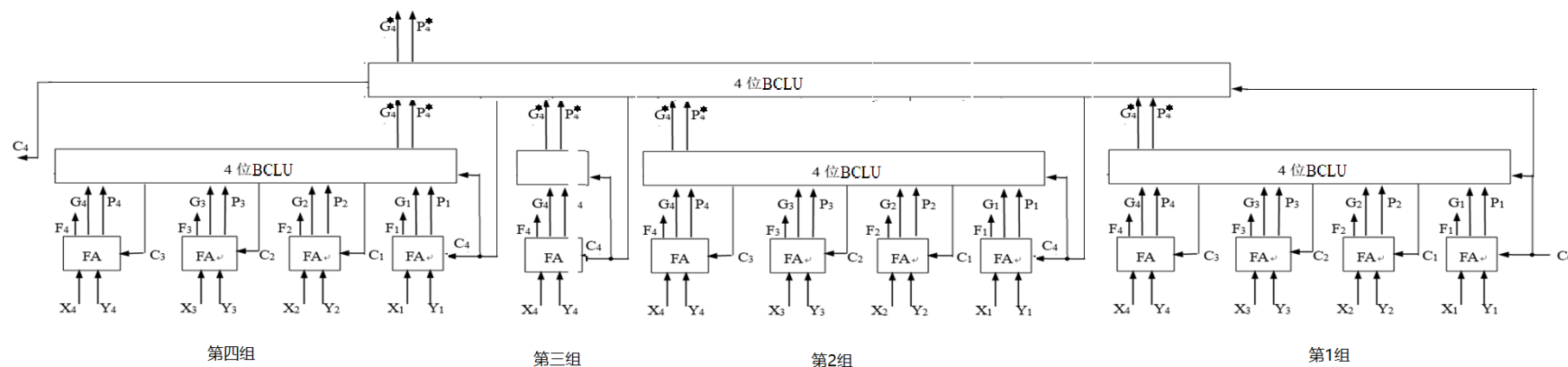
多级先行进位加法器

16位两级先行进位加法器



多级先行进位加法器

16位两级先行进位加法器



BCLU向下传递进位信号后，下一级CLU计算超前进位需要的延迟

$$1(\text{gp}) + 2(\text{clu}) + 2(\text{clu}) + 2(\text{clu}) + 3(\text{xor}) = 10t_y$$

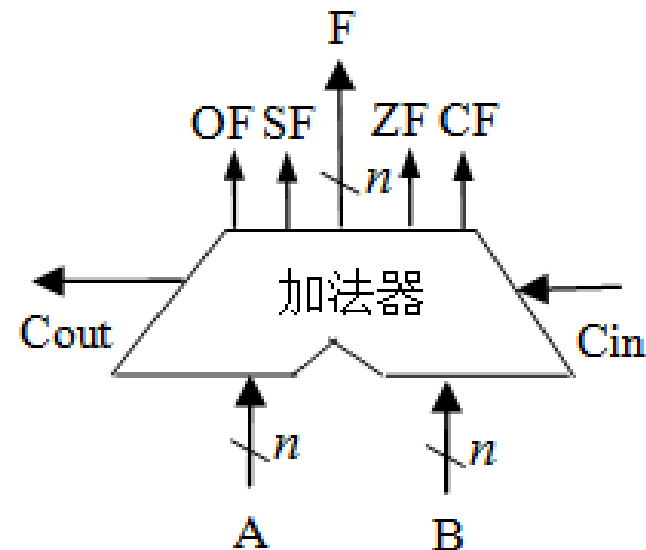
第一讲 基本运算部件

1. 高级语言程序中所涉及到的运算（以C语言为例）
 1. 整数算数运算、浮点数算数运算
 2. 按位、逻辑、移位、位扩展和位截断
2. 串行进位加法器
3. 并行进位加法器
 1. 全先行进位加法器
 2. 两级/多级先行进位加法器
4. 带标志加法器
5. 算数逻辑部件（ALU）

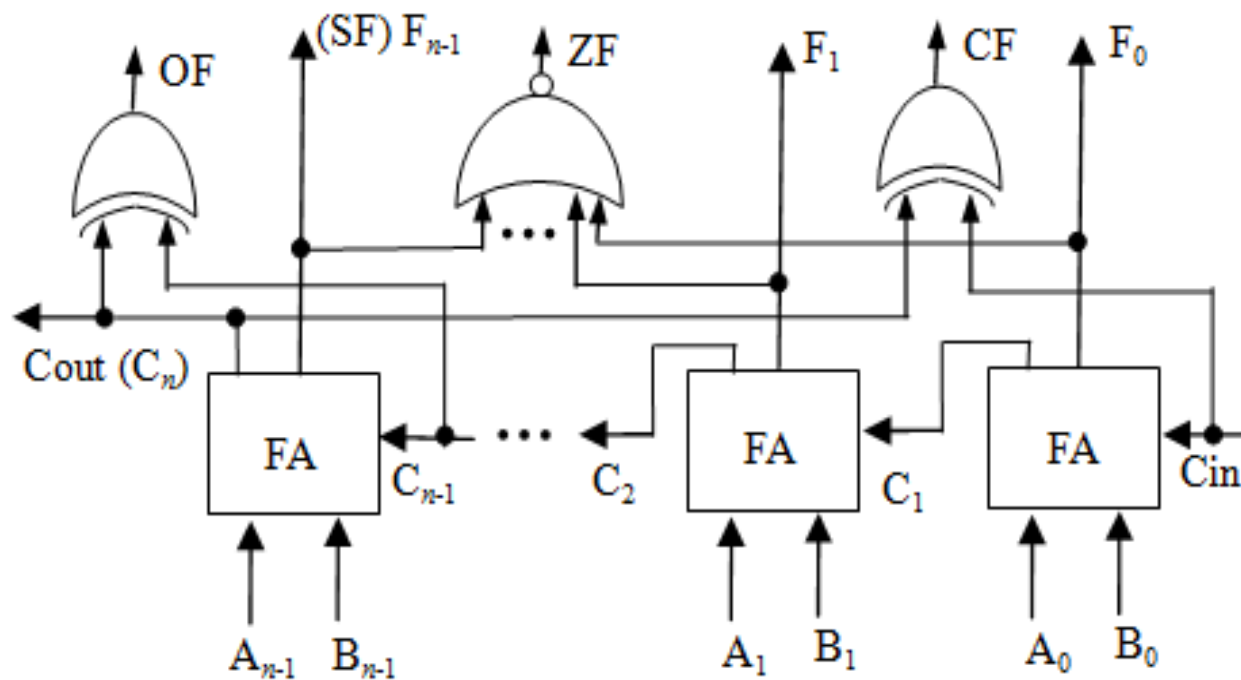
n位带标志加法器

需求：增加运算结果的标志信息（只针对加法）

- 判断是否溢出
 - 通盘考虑：n位带符号整数（补码）相加
- 比较大小
 - 通过（在加法器中）做减法来判断



带标志加法器符号



带标志加法器的逻辑电路

溢出标志OF：

$$OF = C_n \oplus C_{n-1}$$

符号标志SF：

$$SF = F_{n-1}$$

零标志ZF=1：

当且仅当F=0；

进位/借位标志CF：

$$CF = Cout \oplus Cin$$

第一讲 基本运算部件

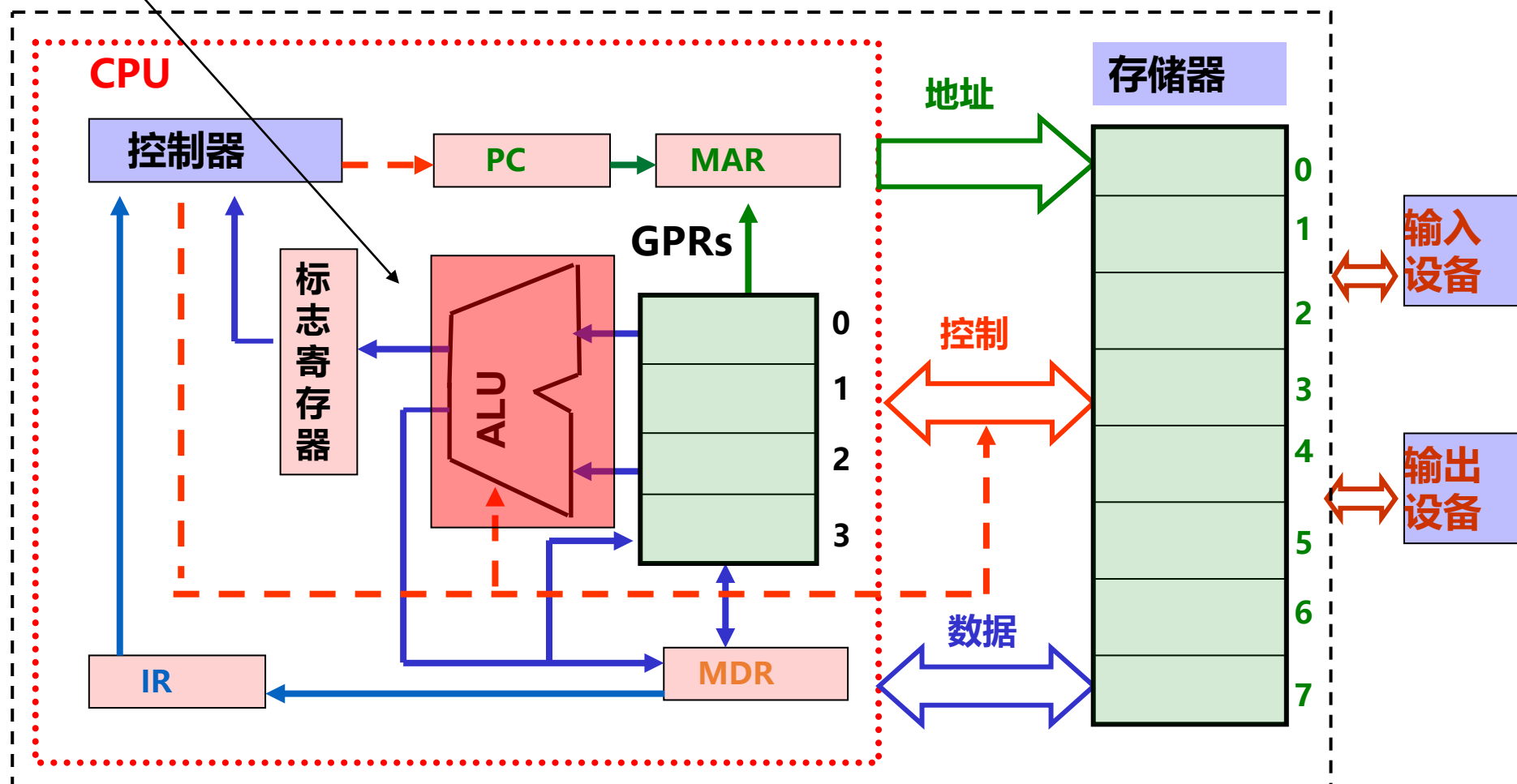
1. 高级语言程序中所涉及到的运算（以C语言为例）
 1. 整数算数运算、浮点数算数运算
 2. 按位、逻辑、移位、位扩展和位截断
2. 串行进位加法器
3. 并行进位加法器
 1. 全先行进位加法器
 2. 两级/多级先行进位加法器
4. 带标志加法器
5. 算数逻辑部件（ALU）

回顾：认识计算机中最基本部件

CPU：中央处理器；PC：程序计数器；MAR：存储器地址寄存器

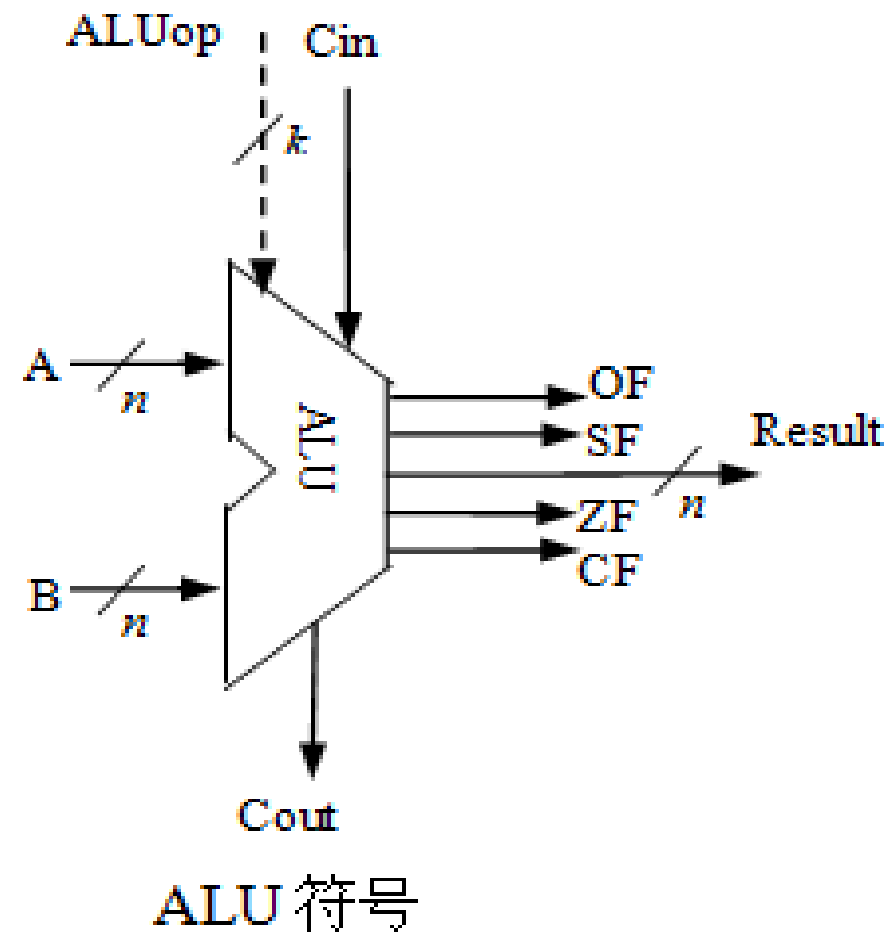
ALU：算术逻辑部件；IR：指令寄存器；MDR：存储器数据寄存器

GPRs：通用寄存器组（由若干通用寄存器组成）



算数逻辑部件 (ALU)

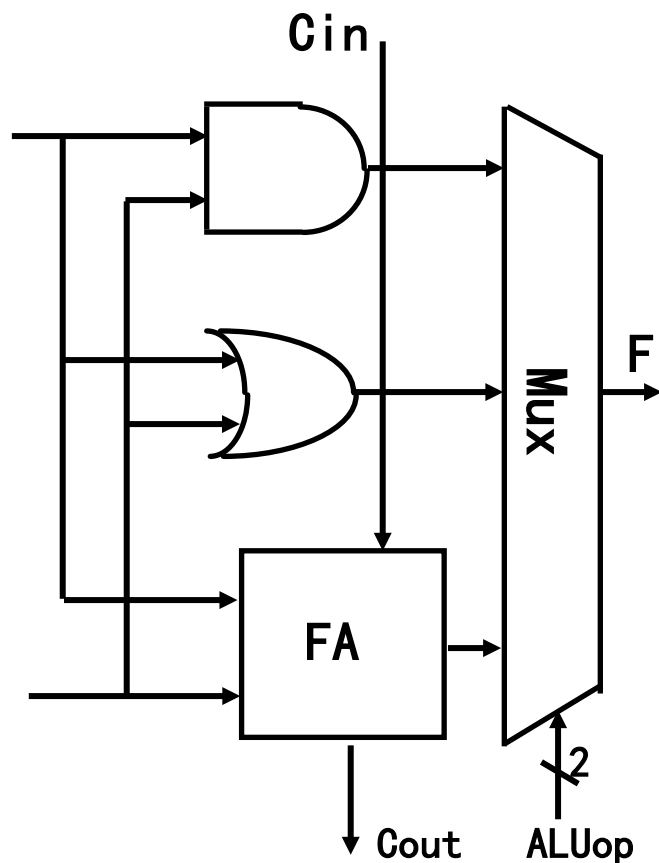
- 进行**基本**算术运算与逻辑运算
 - 无符号整数加、减
 - 带符号整数加、减
 - 与、或、非、异或等逻辑运算
- 核心电路是**整数加/减运算部件**
- 输出除**和/差等**，还有**标志信息**
- 有一个**操作控制端** (ALUop)，用来决定ALU所执行的
处理功能。ALUop的位数 k 决定了操作的种类--例如：
当位数 k 为3时，ALU最多只有 $2^3=8$ 种操作。



ALUop	Result	ALUop	Result	ALUop	Result	ALUop	Result
0 0 0	A加B	0 1 0	A与B	1 0 0	A取反	1 1 0	A
0 0 1	A减B	0 1 1	A或B	1 0 1	$A \oplus B$	1 1 1	未用

举例：1-bit ALU和4-bit ALU

4位先行进位ALU的实现

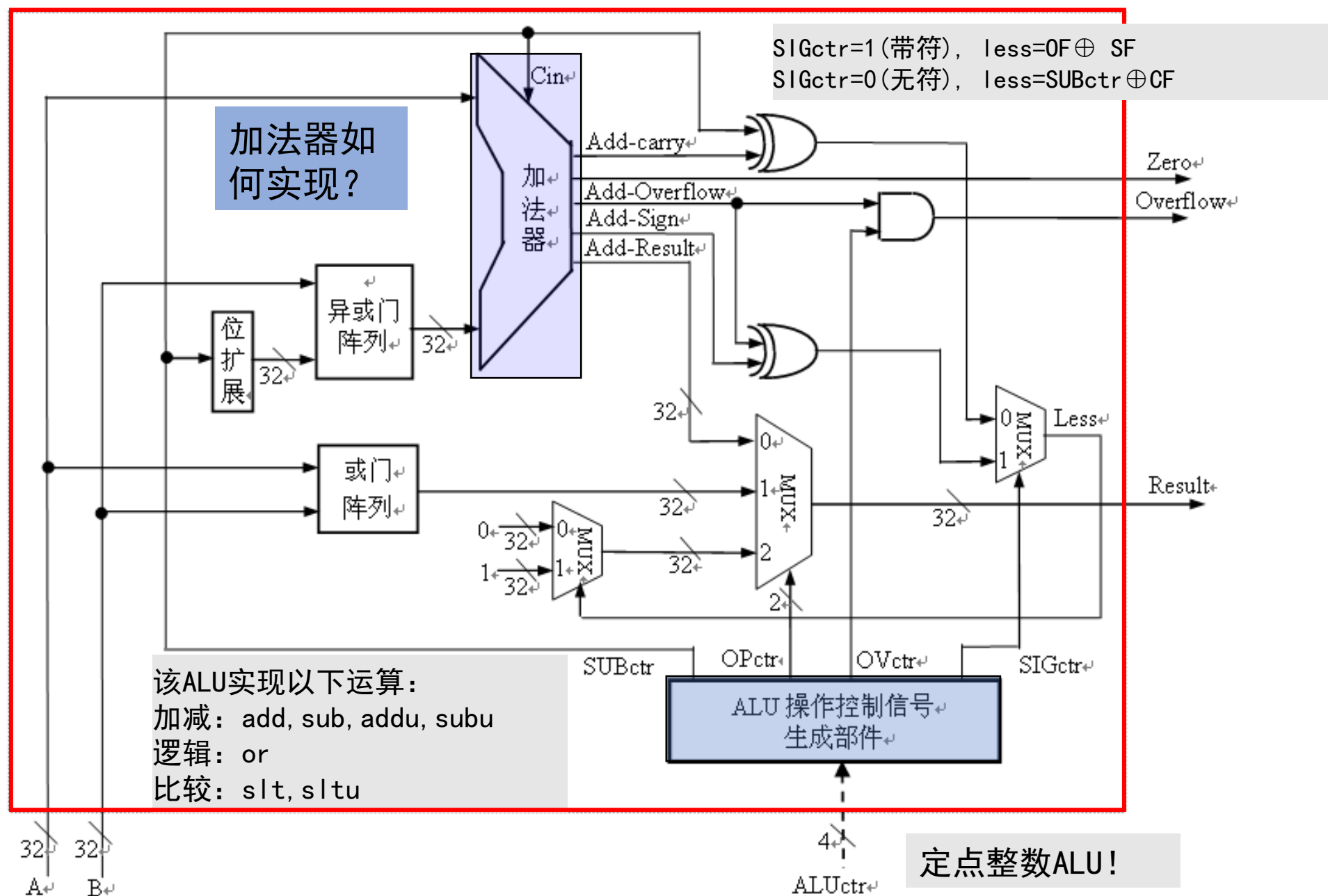


1-bit ALU

```
module ALU (  
    input [3:0] a, b,  
    input [1:0] aluop,  
    input cin,  
    output [3:0] f,  
    output OF, SF, ZF, CF,  
    output cout  
);  
  
    wire [3:0] sum;  
    CLA_FLAGS(a, b, cin, sum, OF, SF, ZF, CF,  
    cout);  
  
    always @(*) begin  
        case(aluop)  
            2' b00: f = a & b;  
            2' b01: f = a | b;  
            2' b10: f = sum;  
            default: f = 0;  
        endcase  
    end  
endmodule
```

实际的ALU中还包括减法、算术移位、逻辑移位等其他运算功能

例：实现某11条MIPS指令的ALU



第二讲 定点数运算

1. 定点数加减运算

补码加减运算 原码加减运算 移码加减运算

2. 定点数乘法运算

原码乘法运算 补码乘法运算 快速乘法器

3. 定点数除法运算

原码除法运算 补码除法运算

提醒：后续的运算设计的基本思路是基于前述的ALU设计基础上进行的。

n位整数加减运算器

先看一个C程序段：

```
int x=9, y=-6, z1, z2;
```

```
z1=x+y;
```

```
z2=x-y;
```

问题：上述程序段中，x和y的机器数是什么？z1和z2的机器数是什么？

回答：x的机器数为 $[x]_{\text{补}}$ ，y的机器数为 $[y]_{\text{补}}$ ；z1的机器数为 $[x+y]_{\text{补}}$ ；z2的机器数为 $[x-y]_{\text{补}}$ 。

因此，计算机中需要有一个电路，能够实现以下功能：

已知 $[x]_{\text{补}}$ 和 $[y]_{\text{补}}$ ，计算 $[x+y]_{\text{补}}$ 和 $[x-y]_{\text{补}}$ 。

根据补码定义，有如下公式：

$$[x+y]_{\text{补}} = 2^n + x + y = 2^n + x + 2^n + y = [x]_{\text{补}} + [y]_{\text{补}} \pmod{2^n}$$

$$[x-y]_{\text{补}} = 2^n + x - y = 2^n + x + 2^n - y = [x]_{\text{补}} + [-y]_{\text{补}} \pmod{2^n} \quad [-y]_{\text{补}} = \overline{[y]_{\text{补}}} + 1$$

补码的定义

假定补码有n位，则：

$$[X]_{\text{补}} = 2^n + X \quad (-2^n \leq X < 2^n, \text{ mod } 2^n)$$

n位整数加减运算器

- 补码加减运算公式

$$[A+B]_{\text{补}} = [A]_{\text{补}} + [B]_{\text{补}} \pmod{2^n}$$

$$[A-B]_{\text{补}} = [A]_{\text{补}} + [-B]_{\text{补}} \pmod{2^n}$$

— 实现减法的关键工作在于：求 $[-B]_{\text{补}}$

- 利用带标志加法器，可构造整数加/减运算器，进行以下运算：

无符号整数加、无符号整数减

带符号整数加、带符号整数减

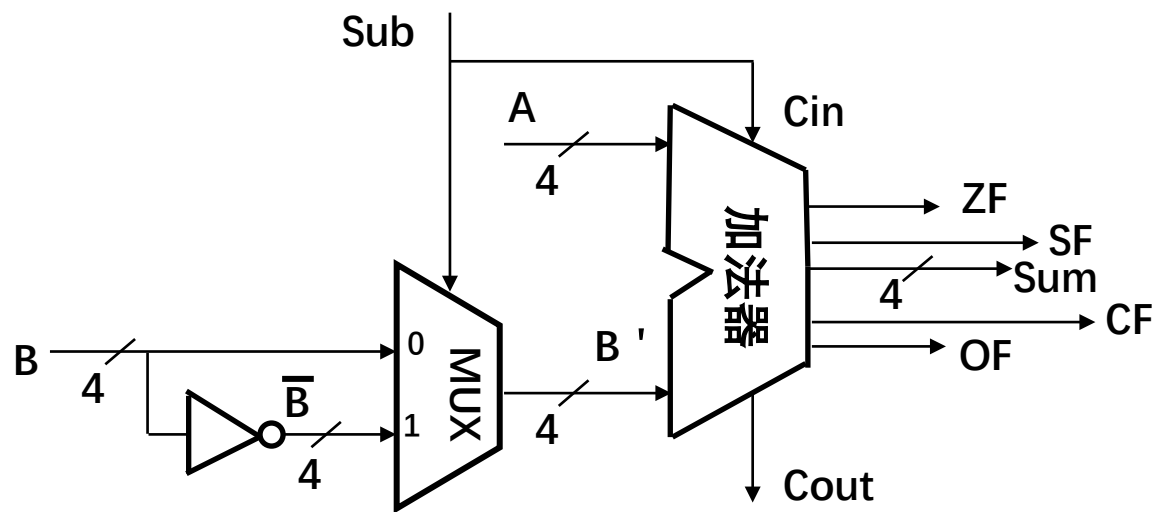
在整数加/减运算部件基础上，加上寄存器、移位器以及控制逻辑，就可实现
ALU、**乘/除**运算以及**浮点**运算电路

问题：如何求 $[-B]_{\text{补}}$ ？

$$[-B]_{\text{补}} = \overline{[B]_{\text{补}}} + 1$$

当Sub为1时，做减法

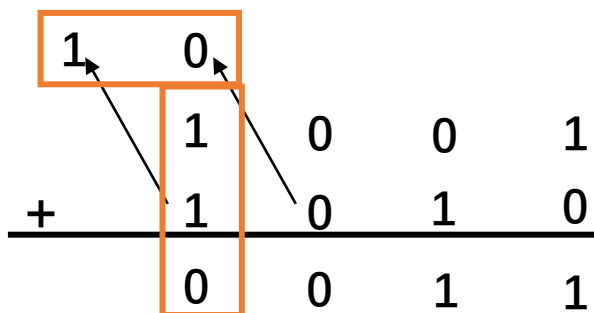
当Sub为0时，做加法



整数加/减运算部件

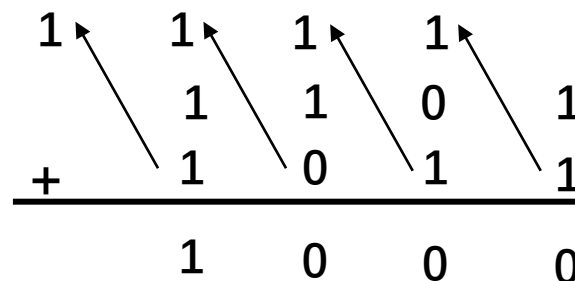
整数减法举例（注意标志位的使用）

$$\begin{array}{l} -7 - 6 = -7 + (-6) = +3 \quad \text{X} \\ 9 - 6 = 3 \quad \checkmark \end{array}$$



OF=1、ZF=0
SF=0、借位CF=0

$$\begin{array}{l} -3 - 5 = -3 + (-5) = -8 \quad \checkmark \\ 13 - 5 = 8 \quad \checkmark \end{array}$$



OF=0、ZF=0、
SF=1、借位CF=0

带符号溢出：

- (1) 最高位和次高位的进位不同,or
- (2) 和的符号位和加数的符号位不同

做减法以比较大小，规则：

Unsigned: CF=0时，大于

Signed: OF=SF时，大于

验证：9>6，故CF=0；13>5，故CF=0

验证：-7<6，故OF≠SF

-3<5，故OF≠SF

整数减法举例（高级语言的对应）

无符号和带符号加减运算都用该部件执行

```
unsigned int x=134;
```

```
unsigned int y=246;
```

```
int m=x;
```

```
int n=y;
```

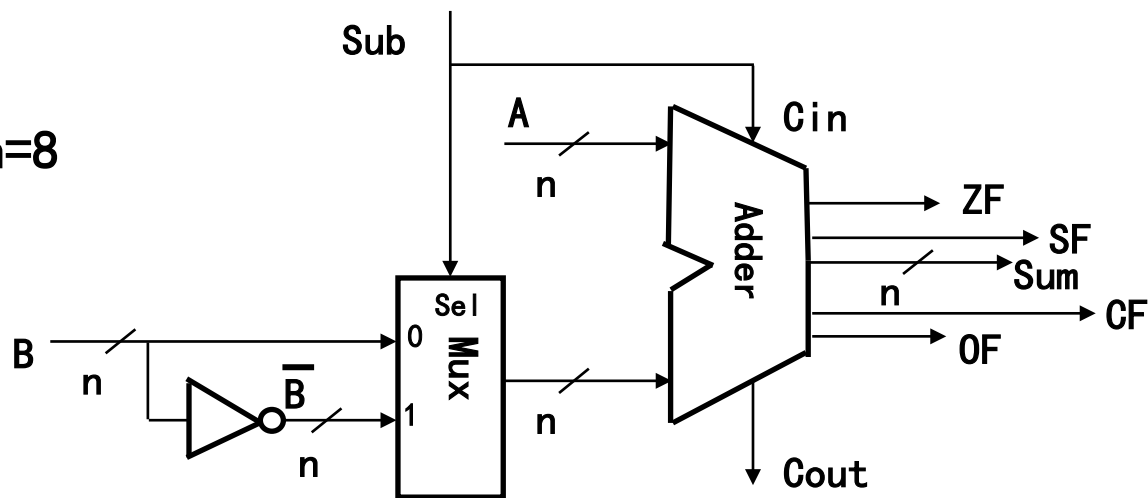
```
unsigned int z1=x-y;
```

```
unsigned int z2=x+y;
```

```
int k1=m-n;
```

```
int k2=m+n;
```

假定 $n=8$



x和m的机器数一样：1000 0110，y和n的机器数一样：1111 0110

z1和k1的机器数一样：1001 0000，CF=1，OF=0，SF=1

z1的值为144（ $=134-246+256$ ， $x-y<0$ ），k1的值为-112。

无符号减公式：

$$\text{result} = \begin{cases} x-y & (x-y > 0) \\ x-y+2^n & (x-y < 0) \end{cases}$$

带符号减公式：

$$\text{result} = \begin{cases} x-y-2^n & (2^{n-1} \leq x-y) & \text{正溢出} \\ x-y & (-2^{n-1} \leq x-y < 2^{n-1}) & \text{正常} \\ x-y+2^n & (x-y < -2^{n-1}) & \text{负溢出} \end{cases}$$

整数加法举例（高级语言的对应）

无符号和带符号加减运算都用该部件执行

```
unsigned int x=134;
```

```
unsigned int y=246;
```

```
int m=x;
```

```
int n=y;
```

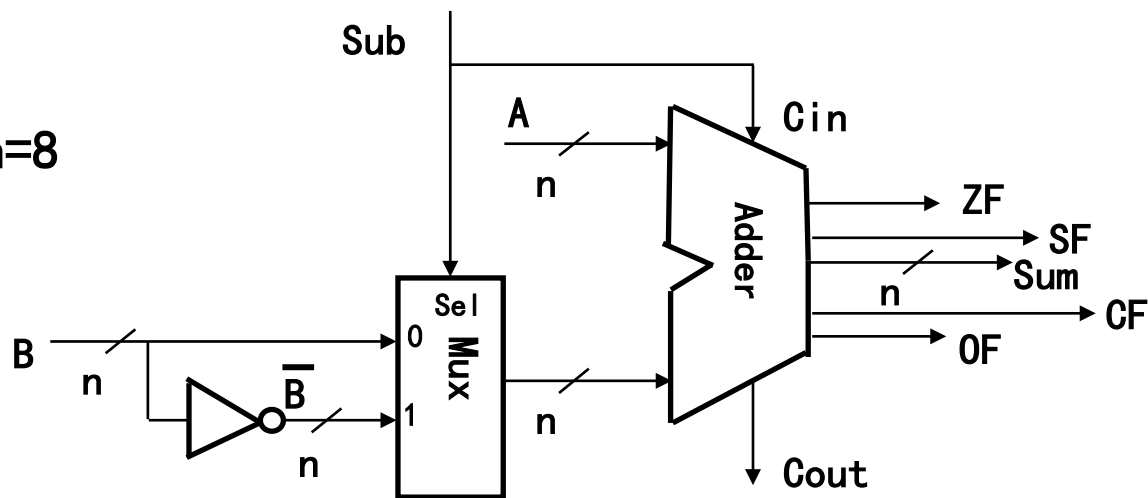
```
unsigned int z1=x-y;
```

```
unsigned int z2=x+y;
```

```
int k1=m-n;
```

```
int k2=m+n;
```

假定 $n=8$



x和m的机器数一样：1000 0110，y和n的机器数一样：1111 0110

z2和k2的机器数一样：1001 0000，CF=1，OF=0，SF=1

z2的值为124 ($=134-246-256$ ， $x+y>256$)，k1的值为124 ($=134+246-256$ ，正溢出)。

带符号加公式：

无符号加公式：

$$\text{result} = \begin{cases} x+y & (x+y < 2^n) \\ x+y-2^n & (2^n \leq x+y < 2^{n+1}) \end{cases}$$

$$\text{result} = \begin{cases} x+y-2^n & (2^{n-1} \leq x+y) & \text{正溢出} \\ x+y & (-2^{n-1} \leq x+y < 2^{n-1}) & \text{正常} \\ x+y+2^n & (x+y < -2^{n-1}) & \text{负溢出} \end{cases}$$

原码加减运算

用于浮点数尾数运算(注意：浮点数加减运算时，要先对齐阶码)

- 符号位和数值部分分开处理
- 仅对数值部分进行加减运算，符号位起判断和控制作用

规则如下：

- 比较两数符号，对加法实行“同号求和，异号求差”，对减法实行“异号求和，同号求差”。
- 求和：数值位相加，和的符号取被加数（被减数）的符号。若最高位产生进位，则结果溢出。
- 求差：被加数（被减数）加上加数（减数）的补码。
 - a) 最高数值位产生进位表明加法结果为正，所得数值位正确。
 - b) 最高数值位没产生进位表明加法结果为负，得到的是数值位的补码形式，需对结果求补，还原为绝对值形式的数值位。
- 差的符号位：
 - a) 情况下，符号位取被加数（被减数）的符号；
 - b) 情况下，符号位为被加数（被减数）的符号取反。

移码加减运算

- 用于浮点数阶码运算（符号位和数值部分可以一起处理）
- 运算公式（假定在一个n位ALU中进行加法运算）

$$[E1]_{\text{移}} + [E2]_{\text{移}} = 2^{n-1} + E1 + 2^{n-1} + E2 = 2^n + E1 + E2 = [E1 + E2]_{\text{补}} \pmod{2^n}$$

$$[E1]_{\text{移}} - [E2]_{\text{移}} = [E1]_{\text{移}} + [-[E2]_{\text{移}}]_{\text{补}} = 2^{n-1} + E1 + 2^n - [E2]_{\text{移}}$$

$$= 2^{n-1} + E1 + 2^n - 2^{n-1} - E2$$

$$= 2^n + E1 - E2 = [E1 - E2]_{\text{补}} \pmod{2^n}$$

结论：移码的和、差等于和、差的补码！（需要转换成移码）

- 运算规则
 - ① 加法：直接将 $[E1]_{\text{移}}$ 和 $[E2]_{\text{移}}$ 进行模 2^n 加，然后对结果的符号取反。
 - ② 减法：先将减数 $[E2]_{\text{移}}$ 求补（各位取反，末位加1），然后再与被减数 $[E1]_{\text{移}}$ 进行模 2^n 相加，最后对结果的符号取反。
 - ③ 溢出判断：进行模 2^n 相加时，如果两个加数的符号相同，并且与和数的符号也相同，则发生溢出。

移码加减运算

例1： 用四位移码计算 “ $-7+(-6)$ ” 和 “ $-3+6$ ” 的值。

解： $[-7]_{\text{移}} = 0001$ $[-6]_{\text{移}} = 0010$ $[-3]_{\text{移}} = 0101$ $[6]_{\text{移}} = 1110$

$[-7]_{\text{移}} + [-6]_{\text{移}} = 0001 + 0010 = 0011$ （两个加数与结果符号都为0，溢出）

$[-3]_{\text{移}} + [6]_{\text{移}} = 0101 + 1110 = 0011$ ， 符号取反后为 1011 ， 其真值为+3

问题： $[-7+(-6)]_{\text{移}}=?$ $[-3+(6)]_{\text{移}}=?$



例2： 用四位移码计算 “ $-7 - (-6)$ ” 和 “ $-3 - 5$ ” 的值。

解： $[-7]_{\text{移}} = 0001$ $[-6]_{\text{移}} = 0010$ $[-3]_{\text{移}} = 0101$ $[5]_{\text{移}} = 1101$

$[-7]_{\text{移}} - [-6]_{\text{移}} = 0001 + 1110 = 1111$ ， 符号取反后为 0111 ， 其真值为 -1 。

$[-3]_{\text{移}} - [5]_{\text{移}} = 0101 + 0011 = 1000$ ， 符号取反后为 0000 ， 其真值为 -8 。

第二讲 定点数运算

1. 定点数加减运算

补码加减运算 原码加减运算 移码加减运算

2. 定点数乘法运算

原码乘法运算 补码乘法运算 快速乘法器*

3. 定点数除法运算

原码除法运算 补码除法运算

提醒：后续的运算设计的基本思路是基于前述的ALU设计基础上进行的。

无符号数的乘法运算

假定： $[X]_{\text{原}} = x_0.x_1\dots x_n$ ， $[Y]_{\text{原}} = y_0.y_1\dots y_n$ ，求 $[x \times y]_{\text{原}}$

数值部分 $z_1\dots z_{2n} = (0.x_1\dots x_n) \times (0.y_1\dots y_n)$ (小数点位置约定，不区分小数还是整数)

Paper and pencil example:

Multiplicand	1000		
Multiplier	<u>x 1001</u>		
	1000	$X \times y_4 \times 2^{-4}$	$n=4$
	0000	$X \times y_3 \times 2^{-3}$	
	0000	$X \times y_2 \times 2^{-2}$	
	<u>1000</u>	$X \times y_1 \times 2^{-1}$	
Product (积)	0. 1001000		

$$X \times Y = \sum_{i=1}^4 (X \times y_i \times 2^{-i})$$

整个运算过程中用到两种操作：加法 + 左移

因而，可用ALU和移位器来实现乘法运算

无符号数的乘法运算

□手工乘法的特点：

- 每步计算： $X \times y_i$ ，若 $y_i = 0$ ，则得0；若 $y_i = 1$ ，则得 X
- 把①求得的各项结果 $X \times y_i$ 逐次左移，可表示为 $X \times y_i \times 2^{-i}$
- 对②中结果求和，即 $\sum (X \times y_i \times 2^{-i})$ ，这就是两个无符号数的乘积

□计算机内部稍作以下改进：（i从右n向左1排列）

- 每次得 $X \times y_i$ 后，与前面所得结果累加，得到 P_i ，称之为**部分积**。因为不用等到最后一次求和，减少了保存各次相乘结果 $X \times y_i$ 的开销。
- 每次得 $X \times y_i$ 后，不将它左移与前次部分积 P_i 相加，而将部分积 P_i 右移后与 $X \times y_i$ 相加。
- **因为加法运算始终对部分积中高n位进行，故用n位加法器可实现二个n位数相乘。**
- 对乘数中为“1”的位执行加法和右移，对为“0”的位只执行右移，而不执行加法运算。

无符号数的乘法运算

上述思想可写成如下数学推导过程：

$$\begin{aligned} X \times Y &= X \times (0.y_1 y_2 \cdots y_n) \\ &= X \times y_1 \times 2^{-1} + X \times y_2 \times 2^{-2} + \cdots + X \times y_{n-1} \times 2^{-(n-1)} + X \times y_n \times 2^{-n} \\ &= 2^{-n} \times X \times y_n + 2^{-(n-1)} \times X \times y_{n-1} + \cdots + 2^{-2} \times X \times y_2 + 2^{-1} \times X \times y_1 \\ &= 2^{-1} \underbrace{(2^{-1} (2^{-1} \cdots 2^{-1} (2^{-1} (0 + X \times y_n) + X \times y_{n-1}) + \cdots + X \times y_2) + X \times y_1)}_{n \uparrow 2^{-1}} \end{aligned}$$

上述推导过程具有明显的递归性质，因此，无符号数乘法过程可归结为循环计算下列算式的过程：
设 $P_0 = 0$ ，每步的乘积为：

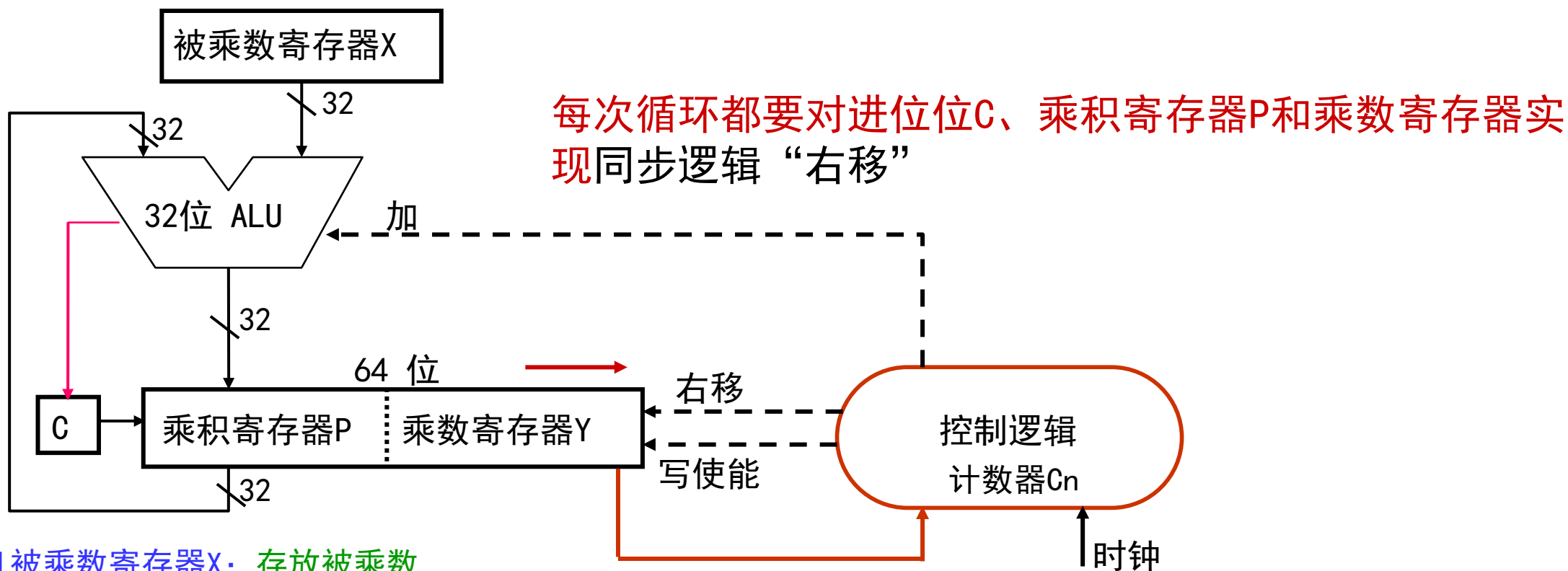
$$\begin{aligned} P_1 &= 2^{-1} (P_0 + X \times y_n) \\ P_2 &= 2^{-1} (P_1 + X \times y_{n-1}) \\ &\cdots \cdots \cdots \\ P_n &= 2^{-1} (P_{n-1} + X \times y_1) \end{aligned}$$

迭代过程从乘数最低位 y_n 和 $P_0=0$ 开始，经 n 次“判断 - 加法 - 右移”循环，直到求出 P_n 为止

其递推公式为： $P_{i+1} = 2^{-1} (P_i + X \times y_{n-i})$ ($i = 0, 1, 2, 3, \cdots, n-1$)

最终乘积 $P_n = X \times Y$

32位乘法的运算实现



- **被乘数寄存器X**：存放被乘数
- **乘积寄存器P**：开始置初始部分积 $P_0 = 0$ ；结束时，存放的是64位乘积的高32位
- **乘数寄存器Y**：开始时置乘数；结束时，存放的是64位乘积的低32位
- **进位触发器C**：保存加法器的进位信号
- **循环次数计数器Cn**：存放循环次数。初值32，每循环一次，Cn减1，Cn=0时结束
- **ALU**：乘法核心部件。在控制逻辑控制下，对P和X的内容“加”，在“写使能”控制下运算结果被送回P，进位位在C中

无符号整数乘法运算举例

举例说明：若需计算 $z=x*y$ ； x 、 y 和 z 都是unsigned类型。

设 $x=1110$ $y=1101$ 应用递推公式： $P_i=2^{-1}(x*y_i+ P_{i-1})$

	C	乘积P	乘数R
	0	0000	1101
	+	1110	
	0	1110	1101
→	0	0111	0110
→	0	0011	1011
	+	1110	
	1	0001	1011
→	0	1000	1101
	+	1110	
	1	0110	1101
→	0	1011	0110

可用一个双倍字长的乘积寄存器；也可用两个单倍字长的寄存器。

部分积初始为0。

保留进位位。

右移时进位、部分积和剩余乘数一起进行逻辑右移。

当 z 取4位时，结果发生溢出，因为高4位不为全0！

原码乘法运算

用于浮点数尾数乘运算

符号与数值分开处理：积符号或得到，数值用无符号乘法运算

例：设 $[x]_{\text{原}}=0.1110$ ， $[y]_{\text{原}}=1.1101$ ，计算 $[x \times y]_{\text{原}}$

解：数值部分用无符号数乘法算法计算： $1110 \times 1101 = 1011\ 0110$

符号位： $0 \oplus 1 = 1$ ，所以： $[x \times y]_{\text{原}} = 1.10110110$

一位乘法：每次只取乘数的一位判断，需 n 次循环，速度慢。

两位乘法：每次取乘数两位判断，只需 $n/2$ 次循环，快一倍。

◆ 两位乘法递推公式： 触发器 T 用来记录下次是否要执行“+X”

00: $P_{i+1} = 2^{-2}P_i$

01: $P_{i+1} = 2^{-2}(P_i + X)$

10: $P_{i+1} = 2^{-2}(P_i + 2X)$

11: $P_{i+1} = 2^{-2}(P_i + 3X) = 2^{-2}(P_i + 4X - X)$
 $= 2^{-2}(P_i - X) + X$

3X时，本次-X，下次+X！

部分积右移两位，相当于4X

“-X”运算用“+[-X]_补”实现！

y_{i-1}	y_i	T	操 作	迭 代 公 式
0	0	0	$0 \rightarrow T$	$2^{-2}(P_i)$
0	0	1	+X $0 \rightarrow T$	$2^{-2}(P_i + X)$
0	1	0	+X $0 \rightarrow T$	$2^{-2}(P_i + X)$
0	1	1	+2X $0 \rightarrow T$	$2^{-2}(P_i + 2X)$
1	0	0	+2X $0 \rightarrow T$	$2^{-2}(P_i + 2X)$
1	0	1	-X $1 \rightarrow T$	$2^{-2}(P_i - X)$
1	1	0	-X $1 \rightarrow T$	$2^{-2}(P_i - X)$
1	1	1	$1 \rightarrow T$	$2^{-2}(P_i)$

原码两位乘法举例

已知 $[X]_{\text{原}} = 0.111001$, $[Y]_{\text{原}} = 0.100111$, 用原码两位乘法计算 $[X \times Y]_{\text{原}}$

解: 先用无符号数乘法计算 111001×100111 , 原码两位乘法过程如下:

$$[|X|]_{\text{补}} = 000\ 111001, [-|X|]_{\text{补}} = 111\ 000111$$

采用补码算术右移, 与一位乘法不同, Why?

为模8补码形式(三位符号位), Why?

若用两位符号位, 则P和Y同时右移2位时, 得到的P3是负数, 这显然是错误的! 需要再增加一位符号。

P	Y	T	说明
000 000000	100111	0	开始, $P_0=0$, $T=0$
+111 000111			$y_5y_6T=110$, $-X$, $T=1$
111 000111			P 和 Y 同时右移 2 位
111 110001	11 1001	1	得 P_1
+001 110010			$y_3y_4T=011$, $+2X$, $T=0$
001 100011			P 和 Y 同时右移 2 位
000 011000	1111 10	0	得 P_2
+001 110010			$y_1y_2T=100$, $+2X$, $T=0$
010 001010			P 和 Y 同时右移 2 位
000 100010	101111	0	得 P_3

加上符号位, 得 $[X \times Y]_{\text{原}} = 0.100010101111$

速度快, 但代价也大

补码乘法运算

因为 $[X \times Y]_{\text{补}} \neq [X]_{\text{补}} \times [Y]_{\text{补}}$ ，故不能直接用无符号数乘法计算。
例如，若 $x = -5$ ，求 $x * x = ?$

用于对什么类型数据计算？

带符号整数！如 int 型

Booth's Algorithm 推导如下：

假定： $[X]_{\text{补}} = x_{n-1}x_{n-2}\dots x_1x_0$ ， $[Y]_{\text{补}} = y_{n-1}y_{n-2}\dots y_1y_0$ ，求： $[X \times Y]_{\text{补}} = ?$

基于以下补码性质：

$$Y = -y_{n-1} \cdot 2^{n-1} + y_{n-2} \cdot 2^{n-2} + \dots + y_1 \cdot 2^1 + y_0 \cdot 2^0$$

令： $y_{-1} = 0$ ，则：

$$\text{当 } n=32 \text{ 时, } Y = -y_{31} \cdot 2^{31} + y_{30} \cdot 2^{30} + \dots + y_1 \cdot 2^1 + y_0 \cdot 2^0 + y_{-1} \cdot 2^0$$

$$\downarrow$$
$$-y_{31} \cdot 2^{31} + (y_{30} \cdot 2^{31} - y_{30} \cdot 2^{30}) + \dots + (y_0 \cdot 2^1 - y_0 \cdot 2^0) + y_{-1} \cdot 2^0$$

$$\downarrow$$
$$(y_{30} - y_{31}) \cdot 2^{31} + (y_{29} - y_{30}) \cdot 2^{30} + \dots + (y_0 - y_1) \cdot 2^1 + (y_{-1} - y_0) \cdot 2^0$$

$$2^{-32} \cdot [X \times Y]_{\text{补}} = (y_{30} - y_{31}) X \cdot 2^{-1} + (y_{29} - y_{30}) X \cdot 2^{-2} + \dots + (y_0 - y_1) X \cdot 2^{-31} + (y_{-1} - y_0) X \cdot 2^{-32}$$

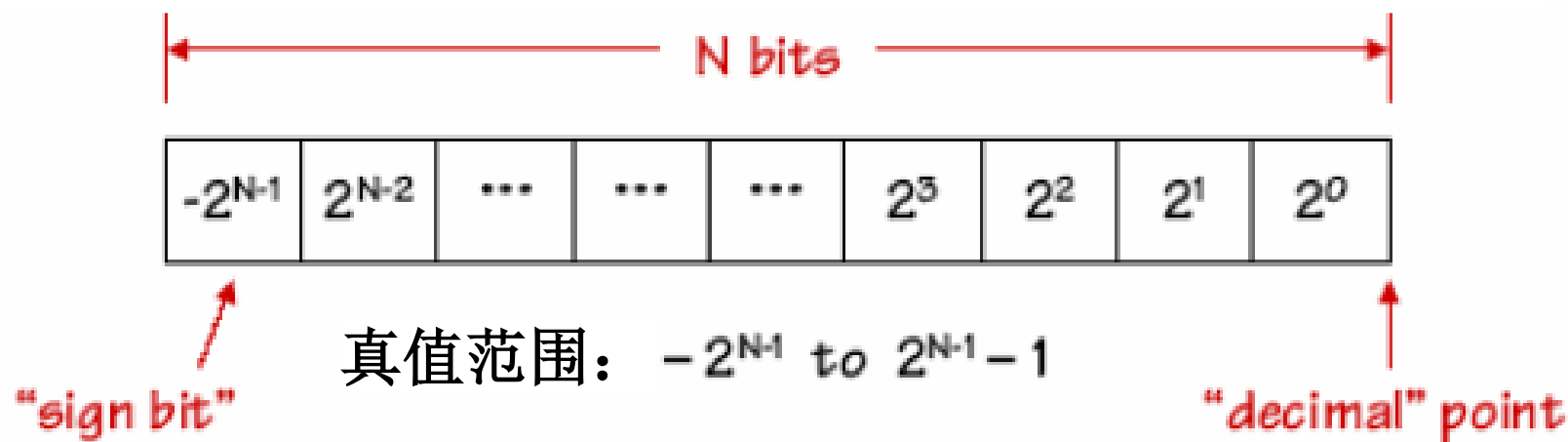
$$= 2^{-1} (2^{-1} \dots (2^{-1} (y_{-1} - y_0) X) + (y_0 - y_1) X) + \dots + (y_{30} - y_{31}) X$$

部分积公式： $P_i = 2^{-1} (P_{i-1} + (y_{i-1} - y_i) X)$ 符号与数值统一处理

回顾：如何求补码的真值

令： $[A]_{\text{补}} = a_{n-1}a_{n-2}\cdots a_1a_0$

则： $A = -a_{n-1} \cdot 2^{n-1} + a_{n-2} \cdot 2^{n-2} + \cdots + a_1 \cdot 2^1 + a_0 \cdot 2^0$



8-bit 2's complement example:

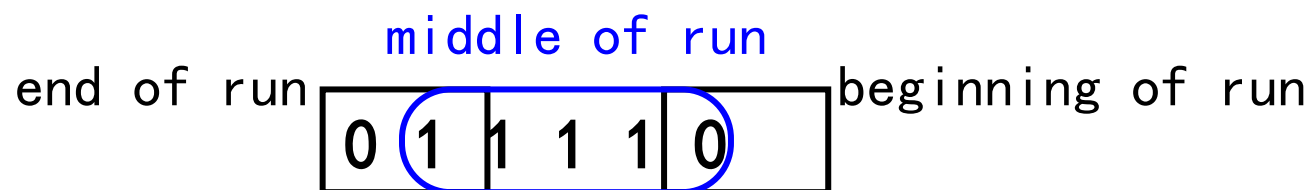
$$11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -128 + 64 + 16 + 4 + 2 = -42$$

符号为0，则为正数，数值部分相同

符号为1，则为负数，数值各位取反，末位加1

例如：补码“11010110”的真值为： $-01010110 = -(32+8+2) = -42$

Booth's 算法实质



- | • 当前位 | 右边位 | 操作 | Example |
|-------|-----|---------|---------------------|
| 1 | 0 | 减被乘数 | 000111 <u>10</u> 00 |
| 1 | 1 | 加0（不操作） | 00011 <u>11</u> 000 |
| 0 | 1 | 加被乘数 | 00 <u>01</u> 111000 |
| 0 | 0 | 加0（不操作） | 0 <u>00</u> 1111000 |
- 在“1串”中，第一个1时做减法，最后一个1做加法，其余情况只要移位。
 - 最初提出这种想法是因为在Booth的机器上移位操作比加法更快！

同前面算法一样，将乘积寄存器右移一位。（这里是算术右移）

Booth' s 算法举例

已知 $[X]_{\text{补}} = 1\ 101$, $[Y]_{\text{补}} = 0\ 110$, 计算 $[X \times Y]_{\text{补}}$ $[-X]_{\text{补}} = 0011$

$X=-3$, $Y=6$, $X \times Y=-18$, $[X \times Y]_{\text{补}}$ 应等于11101110或结果溢出

P	Y	y_{-1}	说明
0000	0110	0	设 $y_{-1} = 0$, $[P_0]_{\text{补}} = 0$
0000	0011	0	$y_0 y_{-1} = 00$, P、Y 直接右移一位 得 $[P_1]_{\text{补}}$
+0011	1001	1	$y_1 y_0 = 10$, $+[-X]_{\text{补}}$ P、Y 同时右移一位 得 $[P_2]_{\text{补}}$
0011	1100	1	$y_2 y_1 = 11$, P、Y 直接右移一位 得 $[P_3]_{\text{补}}$
0000	1110	0	$y_3 y_2 = 01$, $+ [X]_{\text{补}}$ P、Y 同时右移一位 得 $[P_4]_{\text{补}}$
+1101			
1101			
1110			

验证：当 $X \times Y$ 取8位时，结果 $-0010010B=-18$ ；取4位时，结果溢出

补码两位乘法（提速）

□ 补码两位乘可用布斯算法推导如下：

- $[P_{i+1}]_{补} = 2^{-1} ([P_i]_{补} + (y_{i-1} - y_i) [X]_{补})$
- $[P_{i+2}]_{补} = 2^{-1} ([P_{i+1}]_{补} + (y_i - y_{i+1}) [X]_{补})$
 $= 2^{-1} (2^{-1} ([P_i]_{补} + (y_{i-1} - y_i) [X]_{补}) + (y_i - y_{i+1}) [X]_{补})$
 $= 2^{-2} ([P_i]_{补} + (y_{i-1} + y_i - 2y_{i+1}) [X]_{补})$

- 开始置附加位 y_{-1} 为0，乘积寄存器最高位前面添加一位附加符号位0。
- 最终的乘积高位部分在乘积寄存器P中，低位部分在乘数寄存器Y中。
- 因为字长总是8的倍数，所以补码的位数 n 应该是偶数，因此，总循环次数为 $n/2$ 。

y_{i+1}	y_i	y_{i-1}	操 作	迭 代 公 式
0	0	0	0	$2^{-2}[P_i]_{补}$
0	0	1	$+ [X]_{补}$	$2^{-2}\{[P_i]_{补} + [X]_{补}\}$
0	1	0	$+ [X]_{补}$	$2^{-2}\{[P_i]_{补} + [X]_{补}\}$
0	1	1	$+ 2[X]_{补}$	$2^{-2}\{[P_i]_{补} + 2[X]_{补}\}$
1	0	0	$+ 2[-X]_{补}$	$2^{-2}\{[P_i]_{补} + 2[-X]_{补}\}$
1	0	1	$+ [-X]_{补}$	$2^{-2}\{[P_i]_{补} + [-X]_{补}\}$
1	1	0	$+ [-X]_{补}$	$2^{-2}\{[P_i]_{补} + [-X]_{补}\}$
1	1	1	0	$2^{-2}[P_i]_{补}$

补码两位乘法举例

□ 已知 $[X]_{\text{补}} = 1\ 101$, $[Y]_{\text{补}} = 0\ 110$, 用补码两位乘法计算 $[X \times Y]_{\text{补}}$ 。

□ 解: $[-X]_{\text{补}} = 0\ 011$, 用补码二位乘法计算 $[X \times Y]_{\text{补}}$ 的过程如下。

P_n	P	Y	y_{-1}	说明
0 0 0 0 0		0 1 1 0	0	开始, 设 $y_{-1} = 0$, $[P_0]_{\text{补}} = 0$
+ 0 0 1 1 0				$y_1 y_0 y_{-1} = 100$, $+2[-X]_{\text{补}}$
0 0 1 1 0			$\rightarrow 2$	P和Y同时右移二位
0 0 0 0 1		1 0 0 1	1	得 $[P_2]_{\text{补}}$
+ 1 1 0 1 0				$y_3 y_2 y_1 = 011$, $+2[X]_{\text{补}}$
1 1 0 1 1				P和Y同时右移二位
1 1 1 1 0		1 1 1 0	$\rightarrow 2$	得 $[P_4]_{\text{补}}$

因此 $[X \times Y]_{\text{补}} = 1110\ 1110$, 与一位补码乘法 (布斯乘法) 所得结果相同, 但循环次数减少了一半。

验证: $-3 \times 6 = -18$ (-10010B)

第二讲 定点数运算

1. 定点数加减运算

补码加减运算 原码加减运算 移码加减运算

2. 定点数乘法运算

原码乘法运算 补码乘法运算 快速乘法器*

3. 定点数除法运算

原码除法运算 补码除法运算

提醒：后续的运算设计的基本思路是基于前述的ALU设计基础上进行的。

除法（纸笔运算）

	1001	Quotient(商)
Divisor 1000	$\overline{)1001010}$	Dividend(被除数)
	$\begin{array}{r} -1000 \\ \hline 10 \\ 101 \\ \hline 1010 \\ -1000 \\ \hline 10 \end{array}$	
		Remainder (余数)

手算除法的基本要点

- ① 被除数与除数相减，够减则上商为1；不够减则上商为0。
- ② 每次得到的差为中间余数，将除数右移后与上次的中间余数比较。用中间余数减除数，够减则上商为1；不够减则上商为0。
- ③ 重复执行第②步，直到求得的商的位数足够为止。

定点除法运算

□ 除前预处理

①若被除数=0且除数 $\neq 0$ ，或定点整数除法时 $|被除数| < |除数|$ ，则商为0，不再继续

②若被除数 $\neq 0$ 、除数=0，则发生“除数为0”异常（浮点数时为 ∞ ）

（若浮点除法被除数和除数都为0，则有些机器产生一个不发信号的NaN，即“quiet NaN”

只有当被除数和除数都 $\neq 0$ ，且商 $\neq 0$ 时，才进一步进行除法运算。

□ 计算机内部无符号数除法运算

- 与手算一样，通过被除数（中间余数）减除数来得到每一位商
够减上商1；不够减上商0（从msb \rightarrow lsb得到各位商）
- 基本操作为减（加）法和移位，故可与乘法合用同一套硬件

两个n位数相除的情况：

(1) 定点正整数（即无符号数）相除：在被除数的高位添n个0

(2) 定点正小数（即原码小数）相除：在被除数的低位添加n个0

这样，就将所有情况都统一为：一个 $2n$ 位数除以一个 n 位数

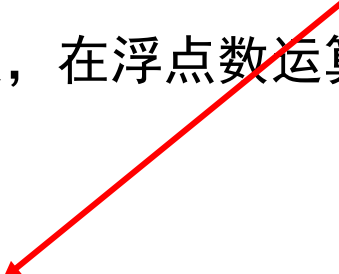
第一次试商为1时的情况

问题：第一次试商为1，说明什么？ 商有 $n+1$ 位数，因而溢出！

通常意义下，若是 $2n$ 位除以 n 位的无符号整数运算，则说明将会得到 $n+1$ 位的商，因而结果“溢出”（即：无法用 n 位表示商）。

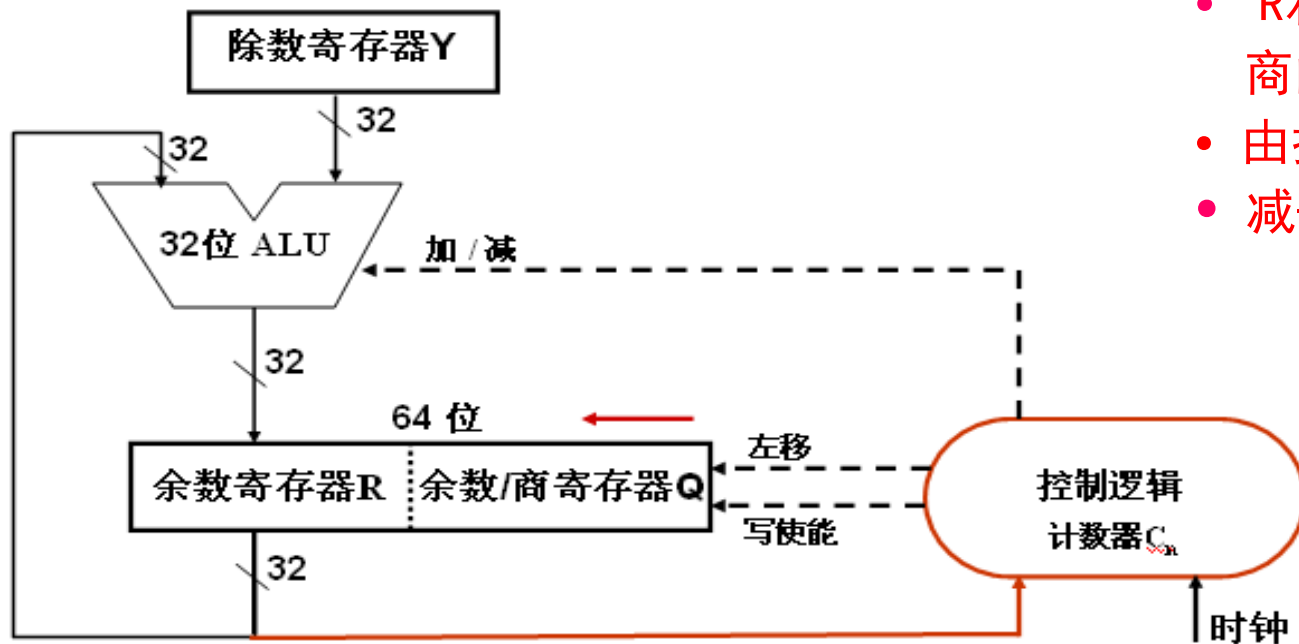
若是两个 n 位数相除，被除数高位扩展0，则第一位商为0，肯定不会溢出

若是浮点数中尾数原码小数运算，第一次试商为1，则说明尾数部分有“溢出”，可通过浮点数的“右规”消除“溢出”。所以，在浮点数运算器中，第一次得到的商“1”要保留。



例： $0.11110000/0.1000=+1.1110$

无符号数除法算法的硬件实现



- R和Q同步“左移”，Q空出位上“商”，商的各位逐次左移到Q中。
- 由控制逻辑根据加减结果决定商为0还是1
- 减----试商，加----恢复余数。

- ❑ 除数寄存器Y：存放除数。
- ❑ 余数寄存器R：初始时高位部分为高32位被除数；结束时是余数。
- ❑ 余数/商寄存器Q：初始时为低32位被除数；结束时是32位商。
- ❑ 循环次数计数器C_n：存放循环次数。初值是32，每循环一次，C_n减1，当C_n=0时，除法运算结束。
- ❑ ALU：除法核心部件。在控制逻辑控制下，对于寄存器R和Y的内容进行“加/减”运算，在“写使能”控制下运算结果被送回寄存器R。

除法算法举例

验证：7 / 2 = 3 余 1

R: 被除数（中间余数）； D: 除数

	D: 0010	R: 0000 0111
Shl R	D: 0010	R: 0000 1110
R = R-D	D: 0010	R: 1110 1110
+D, sl R, 0	D: 0010	R: 0001 1100
R = R-D	D: 0010	R: 1111 1100
+D, sl R, 0	D: 0010	R: 0011 1000
R = R-D	D: 0010	R: 0001 1000
sl R, 1	D: 0010	R: 0011 0001
R = R-D	D: 0010	R: 0001 0001
sl R, 1	D: 0010	R: 0010 0011
Shr R(rh)	D: 0010	R: 0001 0011

这里是两个n位无符号数相除，肯定不会溢出，故余数先左移而省略判断溢出过程。

从例子可看出：

每次上商为0时，需做加法以“恢复余数”。所以，称为“恢复余数法”。

也可在下一步运算时把当前多减的除数补回来。这种方法称为“不恢复余数法”，又称“加减交替法”。

（最后一轮）开始余数先左移了一位，故最后余数需向右移一位

不恢复余数除法（加减交替法）

恢复余数法可进一步简化为“加减交替法”

根据恢复余数法(设B为除数, R_i 为第*i*次中间余数), 有:

□ 若 $R_i < 0$, 则商上“0”, 把先做加法恢复余数再移位, 改为直接在下一步做加法, 即:

$$\square R_{i+1} = 2(R_i + 2^n |B|) - 2^n |B| = 2R_i + 2^n |B| \quad (\text{“负, 0, 加”})$$

□ 若 $R_i \geq 0$, 则商上“1”, 不需恢复余数, 即:

$$\square R_{i+1} = 2R_i - 2^n |B| \quad (\text{“正, 1, 减”})$$

省去了恢复余数的过程

□ 注意: 最后一次上商为“0”的话, 需要“纠余”处理, 即把试商时被减掉的除数加回去, 恢复真正的余数。

□ 不恢复余数法也称为加减交替法

不恢复余数除法（加减交替法）

验证：7 / 2 = 3 余 1 R: 被除数（中间余数）； D: 除数

	D: 0010	R: 0000 0111	
Shl R	D: 0010	R: 0000 1110	- D = 1110
R = R-D	D: 0010	R: 1110 1110	
sl R, 0	D: 0010	R: 1101 1100	“不恢复余数法” 例子
R = R+D	D: 0010	R: 1111 1100	
sl R, 0	D: 0010	R: 1111 1000	
R = R+D	D: 0010	R: 0001 1000	
sl R, 1	D: 0010	R: 0011 0001	
R = R-D	D: 0010	R: 0001 0001	
sl R, 1	D: 0010	R: 0010 0011	
Shr R(rh)	D: 0010	R: 0001 0011	

开始余数先左移了一位，故最后余数需
向右移一位

带符号数除法

- 原码除法
 - 商符和商值分开处理
 - 商的数值部分由无符号数除法求得
 - 商符由被除数和除数的符号确定：同号为0，异号为 1
 - 余数的符号同被除数的符号
- 补码除法
 - 方法1：同原码除法一样，先转换为正数（类似原码表示），先用无符号数除法，然后修正商和余数。
 - 方法2：直接用补码除法，符号和数值一起进行运算，商符直接在运算中产生。

若是两个 n 位补码整数除法运算，则被除数进行符号扩展。

若被除数为 $2n$ 位，除数为 n 位，则被除数无需扩展。

原码除法举例

已知 $[X]_{\text{原}} = 0.1011$

$[Y]_{\text{原}} = 1.1101$

用恢复余数法计算 $[X/Y]_{\text{原}}$

解：商的符号位： $0 \oplus 1 = 1$

减法操作用补码加法实现，是否够减通过中间余数的符号来判断，所以中间余数要加一位符号位。

$[|X|]_{\text{补}} = 0.1011$

$[|Y|]_{\text{补}} = 0.1101$

$[-|Y|]_{\text{补}} = 1.0011$

小数在低位扩展0

思考：若实现无符号数相除，即1011除以1101，则有何不同？结果是什么？

被除数高位补0，1011除以1101，结果等于0

余数寄存器 R	余数/商寄存器 Q	说 明
01011	0000□	开始 $R_0 = X$
+10011		$R_1 = X - Y$
11110	00000	$R_1 < 0$ ，则 $q_4 = 0$
+01101		恢复余数： $R_1 = R_1 + Y$
01011		得 R_1
10110	0000□	$2R_1$ (R 和 Q 同时左移，空出一位商)
+10011		$R_2 = 2R_1 - Y$
01001	00001	$R_2 > 0$ ，则 $q_3 = 1$
10010	0001□	$2R_2$ (R 和 Q 同时左移，空出一位商)
+10011		$R_3 = 2R_2 - Y$
00101	00011	$R_3 > 0$ ，则 $q_2 = 1$
01010	0011□	$2R_3$ (R 和 Q 同时左移，空出一位商)
+10011		$R_4 = 2R_3 - Y$
11101	00110	$R_4 < 0$ ，则 $q_1 = 0$
+01101		恢复余数： $R_4 = R_4 + Y$
01010	00110	得 R_4
10100	0110□	$2R_4$ (R 和 Q 同时左移，空出一位商)
+10011		$R_5 = 2R_4 - Y$
00111	01101	$R_5 > 0$ ，则 $q_0 = 1$

用于判断是否溢出

商的最高位为 0，说明没有溢出，商的数值部分为：1101。

所以， $[X/Y]_{\text{原}} = 1.1101$ (最高位为符号位)，余数为 0.0111×2^4 。

若求 $[Y/X]_{\text{原}}$
结果如何？

原码除法举例

已知 $[X]_{\text{原}} = 0.1011$

$[Y]_{\text{原}} = 1.1101$

用加减交替法计算 $[X/Y]_{\text{原}}$

解: $[|X|]_{\text{补}} = 0.1011$

$[|Y|]_{\text{补}} = 0.1101$

$[-Y]_{\text{补}} = 1.0011$

“加减交替法”的要点:

负、0、加

正、1、减

得到的结果与恢复余数法一样!

用被除数 (中间余数) 减除数试商时,
怎样确定是否“够减”?

中间余数的符号! (正数-正数)

余数寄存器 R	余数/商寄存器 Q	说 明
01011	0000 □	开始 $R_0 = X$
+10011		$R_1 = X - Y$
11110	0000 0	$R_1 < 0$, 则 $q_4 = 0$, 没有溢出
11100	0000 □	$2R_1$ (R 和 Q 同时左移, 空出一位商)
+01101		$R_2 = 2R_1 + Y$
01001	0000 1	$R_2 > 0$, 则 $q_3 = 1$
10010	0001 □	$2R_2$ (R 和 Q 同时左移, 空出一位商)
+10011		$R_3 = 2R_2 - Y$
00101	0001 1	$R_3 > 0$, 则 $q_2 = 1$
01010	0011 □	$2R_3$ (R 和 Q 同时左移, 空出一位商)
+10011		$R_4 = 2R_3 - Y$
11101	0011 0	$R_4 < 0$, 则 $q_1 = 0$
11010	0110 □	$2R_4$ (R 和 Q 同时左移, 空出一位商)
+01101		$R_5 = 2R_4 + Y$
00111	0110 1	$R_5 > 0$, 则 $q_0 = 1$

补码除法能否这样
来判断呢?

不能, 因为符号
可能不同!

补码除法

补码除法判断是否“够减”的规则

- (1) 当被除数（或中间余数）与除数同号时，做减法，若新余数的符号与除数符号一致表示够减，否则为不够减；
- (2) 当被除数（或中间余数）与除数异号时，做加法，若得到的新余数的符号与除数符号一致表示不够减，否则为够减。

上述判断规则归纳如下：

中间余数R 的符号	除数Y的 符号	同号：新中间余数= R-Y（同号为正商）		异号：新中间余数= R+Y（异号为负商）	
		0	1	0	1
0	0	够减	不够减	0100 (4)+1011(-5) =1111(-1)	
0	1	1011(-5)-1101(-3)=1110(-2)		够减	不够减
1	0	不够减	够减	不够减	够减
1	1			1011(-5) +0100 (4) =1111(-1)	

总结：余数变号不够减，不变号够减

实现补码除法的基本思想

从上表可得到补码除法的基本算法思想：

(1) 运算规则：

当被除数（或中间余数）与除数同号时，做减法；异号时，做加法。

(2) 上商规则：

若余数符号不变，则够减，商1；否则不够减，商0。

(3) 修正规则：

若被除数与除数符号一致，则商为正。此时，“够减，商1；不够减，商0，故上商规则正确，无需修正”

若被除数与除数符号不一致，则商为负。此时，“够减，商0；不够减，商1，故上商规则相反，需修正（末位加1）”

补码除法也有：恢复余数法和不恢复余数法

补码恢复余数除法

两个n位带符号整数相除算法要点：

(1) 操作数的预置：

除数装入除数寄存器Y，被除数经符号扩展后装入余数寄存器R和余数/商寄存器Q

(2) R和Q同步串行左移一位。

(3) 若R与Y同号，则 $R = R - Y$ ；否则 $R = R + Y$ ，并按以下规则确定商值 q_0 ：

① 若中间余数 $R = 0$ 或R操作前后符号未变，表示够减，则 q_0 置1，转下一步；

② 若操作前后R的符号已变，表示不够减，则 q_0 置0，恢复R值后转下一步；

(4) 重复第(2)和第(3)步，直到取得n位商为止。

(5) 若被除数与除数同号，则Q中就是真正的商；否则，将Q求补后是真正的商。

(即：若商为负值，则需要“各位取反，末位加1”来得到真正的商)

(6) 余数在R中。

问题：如何恢复余数？通过“做加法”来恢复吗？

无符号数（或原码）除法通过“做加法”恢复余数，但补码不是！

补码：若原来为 $R = R - Y$ ，则执行 $R = R + Y$ 来恢复余数；

若原来是 $R = R + Y$ ，则执行 $R = R - Y$ 来恢复余数。

举例：7/3 (-7) /3

被除数：0000 0111 除数 0011

A	Q	M=0011
0000	0111	
← 0000	1110	
+ 1101		减 (同号)
1101	1110	
+ 0011		恢复(加)商0
0000	1110	
← 0001	1100	
+ 1101		减
1110	1100	
+ 0011		恢复(加)商0
0001	1100	
← 0011	1000	
+ 1101		减
0000	1000	符同商1
← 0001	0001	
+ 1101		减
1110	0010	
+ 0011		恢复(加)商0
0001	0010	

余:0001/商:0010 验证：7/3 = 2, 余数为1

被除数：1111 1001 除数 0011

A	Q	M=0011
1111	1001	
← 1111	0010	
+ 0011		加 (异号)
0010	0010	
+ 1101		恢复(减)商0
1111	0010	
← 1110	0100	
+ 0011		加
0001	0100	
+ 1101		恢复(减)商0
1110	0100	
← 1100	1000	
+ 0011		加
1111	1001	符同商1
← 1111	0010	
+ 0011		加
0010	0010	
+ 1101		恢复(减)商0
1111	0010	

补码不恢复余数除法

补码不恢复余数法也有一个六字口诀“同、1、减；异、0、加”。

其运算过程也呈加/减交替方式，因此也称为“加减交替法”。

- 算法要点：

- (1) 操作数的预置：

- 除数装入除数寄存器Y，被除数经符号扩展后装入余数寄存器R和余数/商寄存器Q。

- (2) 根据以下规则求第一位商 q_n ：

- 若被除数X与Y同号，则 $R_1 = X - Y$ ；否则 $R_1 = X + Y$ ，并按以下规则确定商值 q_n ：

- ① 若新的中间余数 R_1 与Y同号，则 q_n 置1，转下一步；

- ② 若新的中间余数 R_1 与Y异号，则 q_n 置0，转下一步；

- q_n 用来判断是否溢出，而不是真正的商。以下情况下会发生溢出：

- 若X与Y同号且上商 $q_n=1$ ，或者，若X与Y异号且上商 $q_n = 0$ 。

- (3) 对于 $i = 1$ 到 n ，按以下规则求出 n 位商：

- ① 若 R_i 与Y同号，则 q_{n-i} 置1， $R_{i+1} = 2R_i - [Y]$ 补， $i = i + 1$ ；

- ② 若 R_i 与Y异号，则 q_{n-i} 置0， $R_{i+1} = 2R_i + [Y]$ 补， $i = i + 1$ ；

- (4) 商的修正：最后一次Q寄存器左移一位，将最高位 q_n 移出，最低位置上商 q_0 。若被除数与除数同号，Q中就是真正的商；否则，将Q中商的末位加1。

- (5) 余数的修正：若余数符号同被除数符号，则不需修正，余数在R中；否则，按下列规则进行修正：当被除数和除数符号相同时，最后余数加除数；否则，最后余数减除数。

举例： -9/2

将 $X=-9$ 和 $Y=2$ 分别表示成5位补码形式为：

$$[X]_{\text{补}} = 1\ 0111 \quad [Y]_{\text{补}} = 0\ 0010$$

被除数符号扩展为：

$$[X]_{\text{补}} = 11111\ 10111 \quad [-Y]_{\text{补}} = 1\ 1110$$

同、1、减
异、0、加

$$X/Y = -\ 0100B = -\ 4$$

$$\text{余数为 } -\ 0001B = -\ 1$$

将各数代入公式：

“除数 \times 商+余数= 被除数” 进行验证，得

$$2 \times (-4) + (-1) = -9$$

编译器遇到 $x/2$ 时会如何做？ 右移一位！

余数寄存器 R	余数/商寄存器 Q	说 明
11111	10111	开始 $R_0 = [X]$
+00010		$R_1 = [X] + [Y]$
00001	10111	R_1 与 $[Y]$ 同号，则 $q_5=1$
00011	01111	$2R_1$ (R 和 Q 同时左移，空出一位上商 1)
+11110		$R_2 = 2R_1 + [-Y]$
00001	01111	R_2 与 $[Y]$ 同号，则 $q_4=1$
00010	11111	$2R_2$ (R 和 Q 同时左移，空出一位上商 1)
+11110		$R_3 = 2R_2 + [-Y]$
00000	11111	R_3 与 $[Y]$ 同号，则 $q_3=1$
00001	11111	$2R_3$ (R 和 Q 同时左移，空出一位上商 1)
+11110		$R_4 = 2R_3 + [-Y]$
11111	11111	R_4 与 $[Y]$ 异号，则 $q_2=0$
11111	11110	$2R_4$ (R 和 Q 同时左移，空出一位上商 0)
+00010		$R_5 = 2R_4 + [Y]$
00001	11110	R_5 与 $[Y]$ 同号，则 $q_1=1$
00011	11101	$2R_5$ (R 和 Q 同时左移，空出一位上商 1)
+11110		$R_6 = 2R_5 + [-Y]$
00001	11101	R_6 与 $[Y]$ 同号，则 $q_0=1$ ，Q 左移，空出一位上商 1
+11110	+ 1	商为负数，末位加 1；减除数以修正余数
11111	11100	

所以， $[X/Y]_{\text{补}} = 11100$ 。 余数为 11111。

变量与常数之间的除运算

- 不能整除时，采用朝零舍入，即截断方式
 - 无符号数、带符号正整数（地板）：移出的低位直接丢弃
 - 带符号负整数（天板）：加偏移量(2^k-1)，然后再右移 k 位，低位截断（这里 k 是右移位数）

举例：

无符号数 $14/4=3$ ：0000 1110 $\gg 2$ = 0000 0011

带符号负整数 $-14/4=-3$

若直接截断，则 1111 0010 $\gg 2$ = 1111 1100 = -4 \neq -3

应先纠偏，再右移：k=2, 故 $(-14+2^2-1)/4=-3$

即：1111 0010 + 0000 0011 = 1111 0101

1111 0101 $\gg 2$ = 1111 1101 = -3

-9/2: 10111 + 00001 = 11000

11000 $\gg 1$ = 11100 = -4