课程内容

1. 概述

计算机系统概述 ★☆☆
二进制编码 ★☆☆

2. 数字逻辑电路

数字逻辑基础 ★★☆
组合逻辑电路 ★★★
时序逻辑电路 ★★★

3. 计算机组成原理

运算方法和运算部件 ★★☆
指令系统 ★★★
中央处理器 ★★★

* 实验

Logisim实验与答疑 ★★☆

抽象层

程序（语言）

第5章 可编程逻辑器件

器件 → 电路 → 功能部件 → ISA及对应微架构

第2章 数字逻辑基础

第3章 组合逻辑电路

第4章 时序逻辑电路

第6章 运算方法和运算部件

第7章 指令系统

第8章 中央处理器

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~t6中，寄存器t3~t6的编号对应28~31，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

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！

- 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
- 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

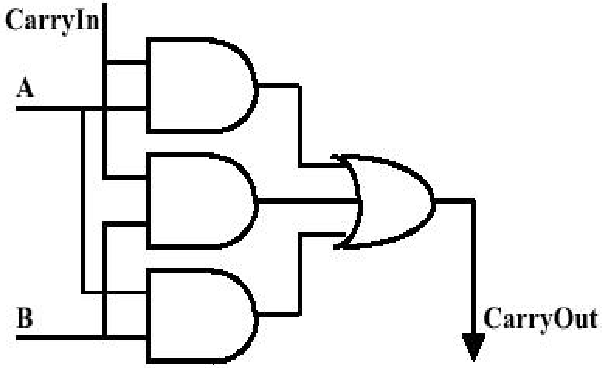
第一讲 基本运算部件

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）

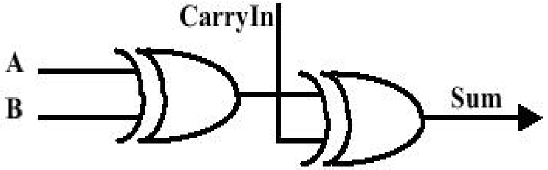
串行进位加法器

CarryOut 和 Sum 的逻辑图

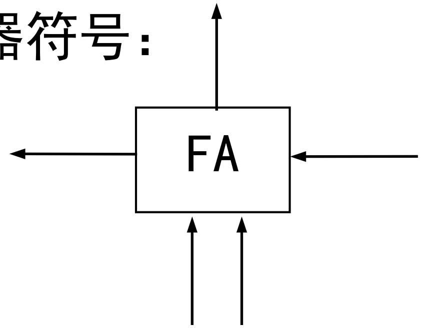
- CarryOut = B & CarryIn | A & CarryIn | A & B



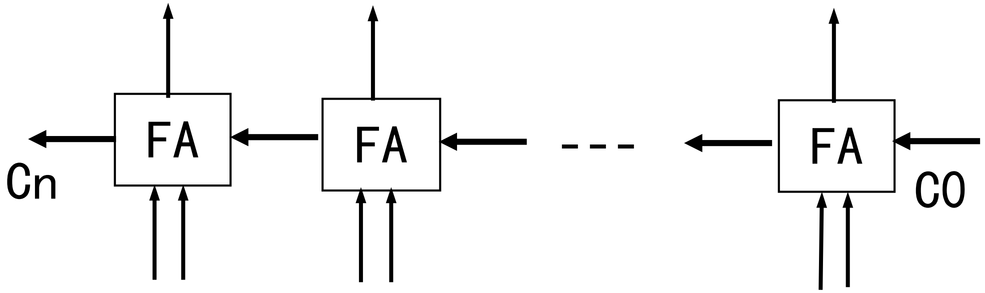
- Sum = A XOR B XOR CarryIn



全加器符号：



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！

假定与/或门延迟为1ty，异或门3ty，则“和”与“进位”的延迟为多少？

Sum延迟为6ty；Carryout延迟为2ty。

问题：n位串行加法器从C0到Cn的延迟时间为多少？ 2n级门延迟！

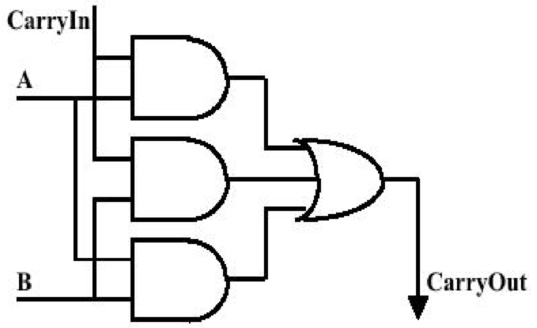
C0到最后一位和数的延迟时间为多少？

2n+1级门延迟！

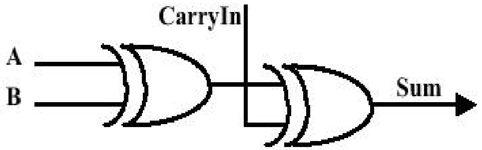
串行进位加法器

CarryOut 和 Sum 的逻辑图

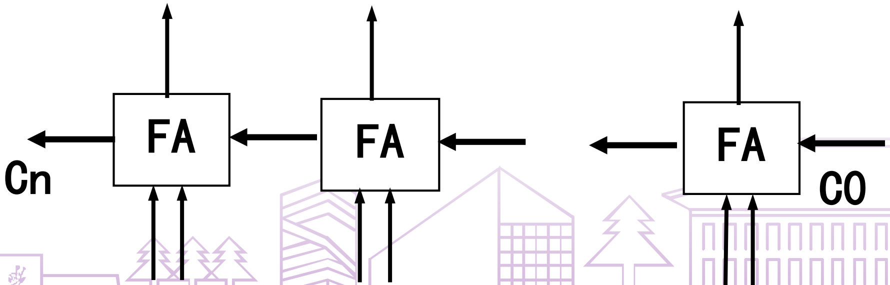
- CarryOut = B & CarryIn | A & CarryIn | A & B



- Sum = A XOR B XOR CarryIn



n位串行(行波)加法器:



4位串行进位加法器RCA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 RCA (
       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，现计算机采用一种先行进位/超前进位(Carry look ahead)方式。

- 如何产生先行进位？

  定义辅助函数：$G_i=X_iY_i$…进位生成函数

  $P_i=X_i+Y_i$…进位传递函数（或 $P_i=X_i\oplus Y_i$ ）

  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：$S_i= X_i\oplus Y_i\oplus C_{i-1}$ $C_i=G_i+P_iC_{i-1}$ $(i=1,\cdots n)$

  设n=4，则：$C_1=G_1+P_1C_0$

  $C_2=G_2+P_2C_1=G_2+P_2G_2+P_2P_1C_0$

  $C_3=G_3+P_3C_2=G_3+P_3G_2+P_3P_2G_0+P_3P_2P_1C_0$

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

  由上式可知：各进位之间无等待，可以独立并同时产生。

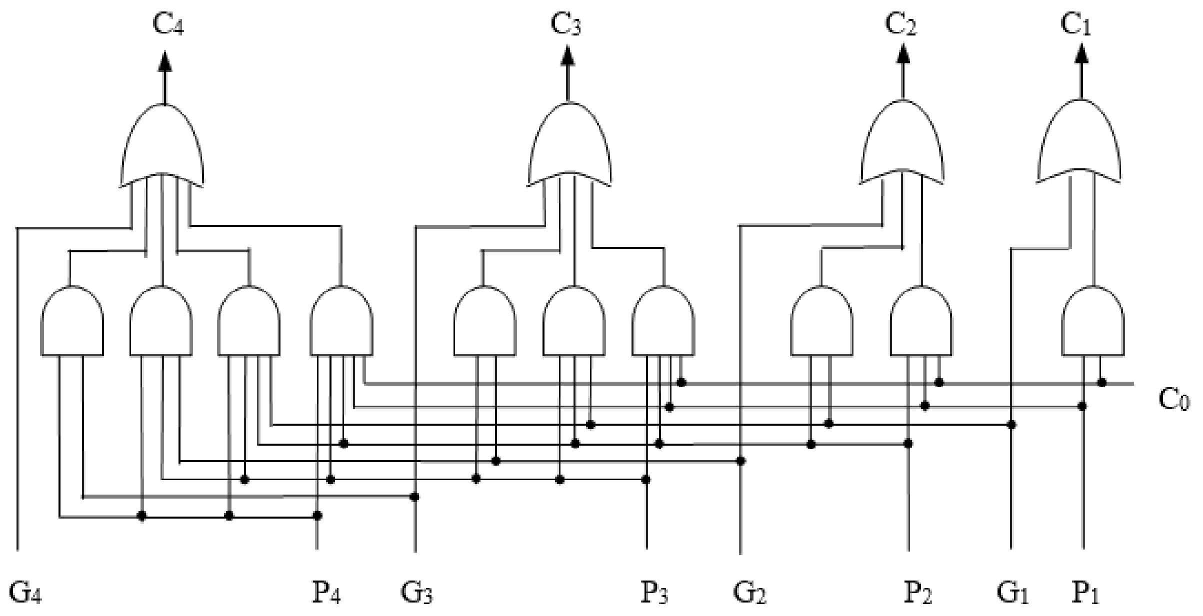
  通常把实现上述逻辑的电路称为4位CLU部件

第三章推导出的全加器逻辑函数

$F=A\oplus B\oplus \mathrm{Cin}$

$\mathrm{Cout}=A\cdot B+A\cdot \mathrm{Cin}+B\cdot \mathrm{Cin}$

并行进位加法器（CLA加法器）



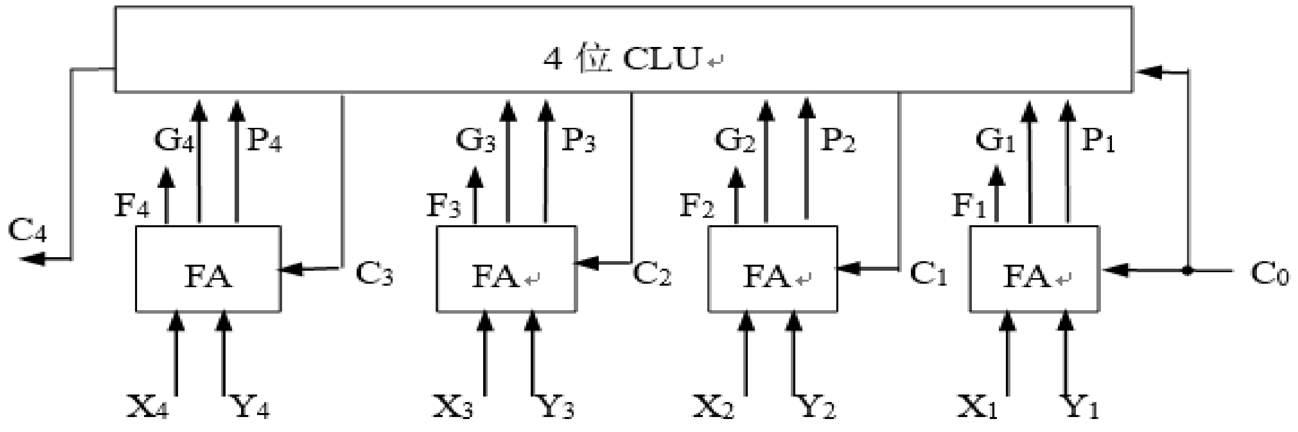
$C_1=G_1+P_1C_0$

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

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

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

4位CLU部件



$G_i=A_iB_i$

$P_i=A_i+B_i$（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）

```
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
```

$G_i=A_iB_i$

$P_i=A_i+B_i$ （或 $P_i=A_i\oplus B_i$ ）

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

```
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
```

$C_1=G_1+P_1C_0$

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

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

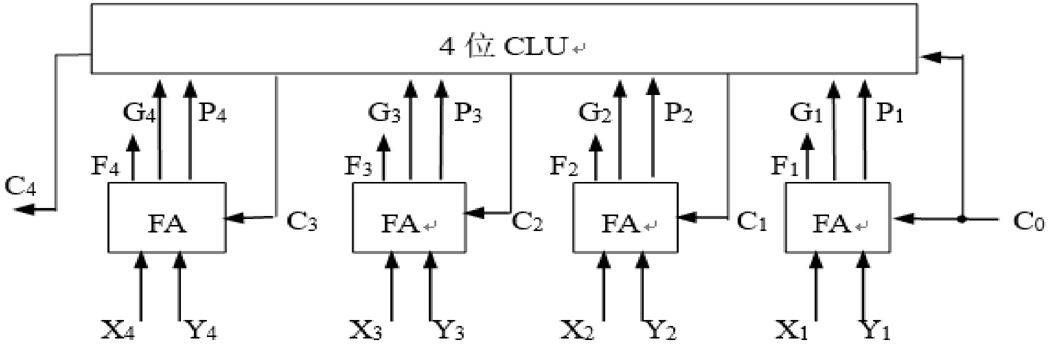
$C_4=G_4+P_4C_3=G_4+P_4G_3+P_4P_3G_2+P_4P_3P_2G_1+P_4P_3P_2P_1C_0$

```
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
```

局部（单级）先行进位加法器

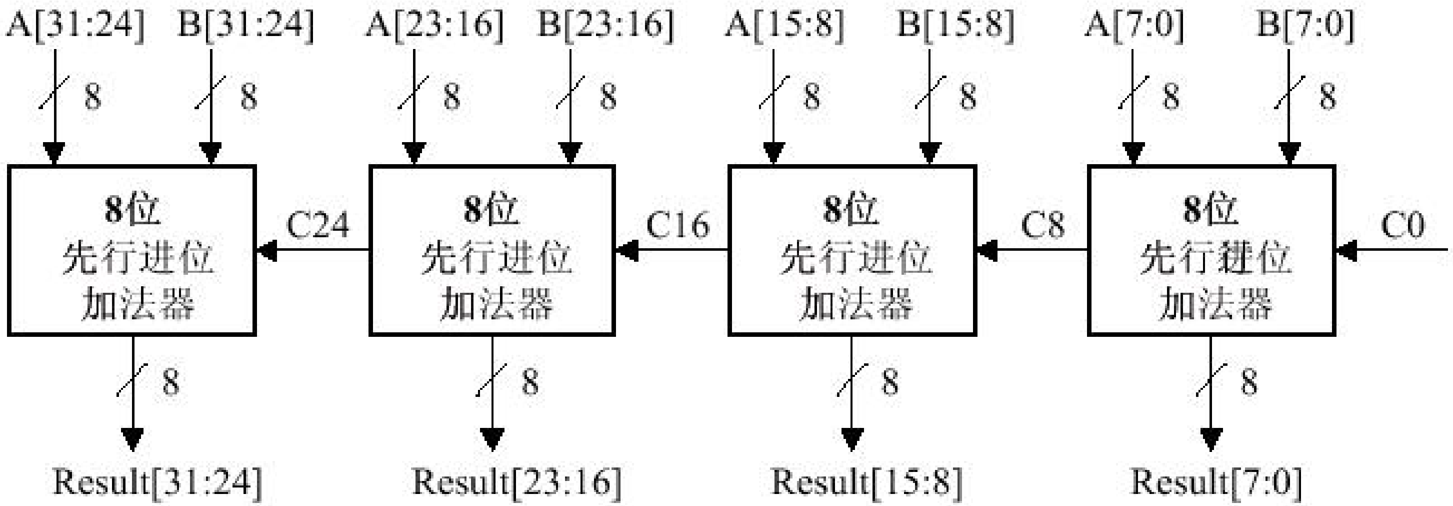
局部先行进位加法器(PartialLookahead Adder)

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？ 1(gp)+2(clu)+2(clu)+2(clu)+2(clu)+3(xor)=12ty



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： $C_1=G_1+P_1C_0$

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

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

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

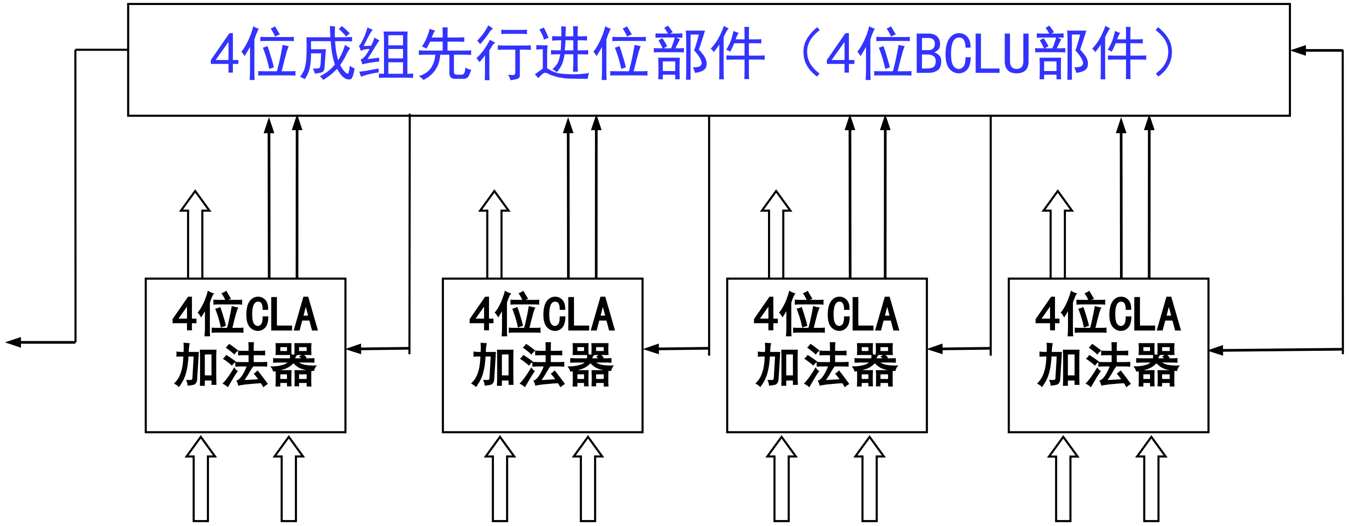
  $G_4^*=G_4+P_4G_3+P_4P_3G_2+P_4P_3P_2G_1$

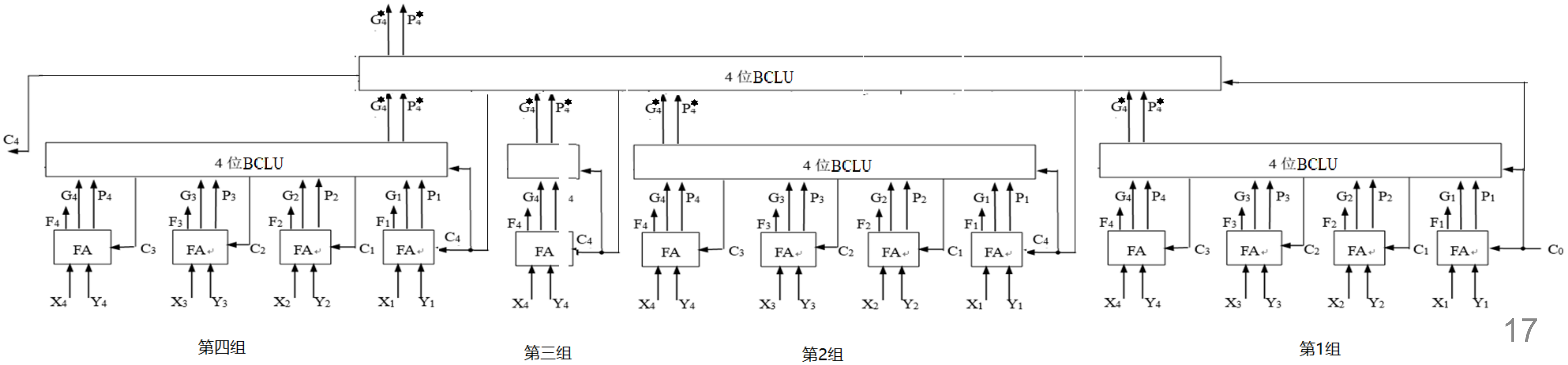
  $P_4^*=P_4P_3P_2P_1$

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

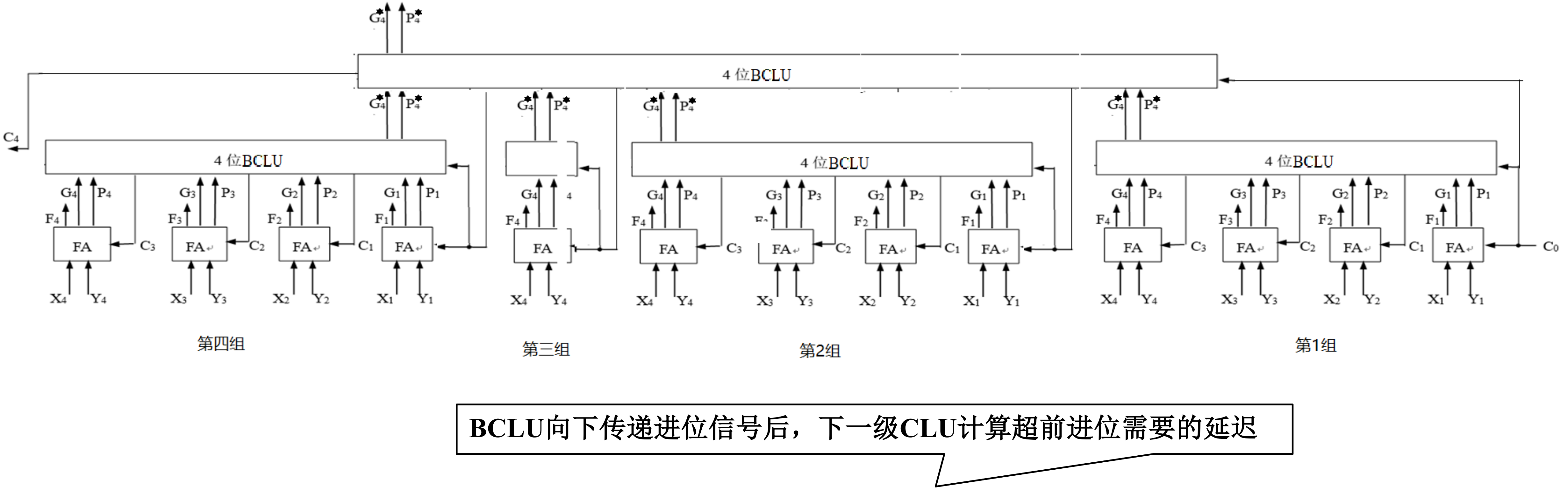
16位两级先行进位加法器





多级先行进位加法器

16位两级先行进位加法器



关键路径长度为多少？ 1(gp)+2(clu)+2(clu)+2(clu)+3(xor)=10ty

最终进位的延迟为多少？ 1(gp)+2(clu)+2(clu)=5ty

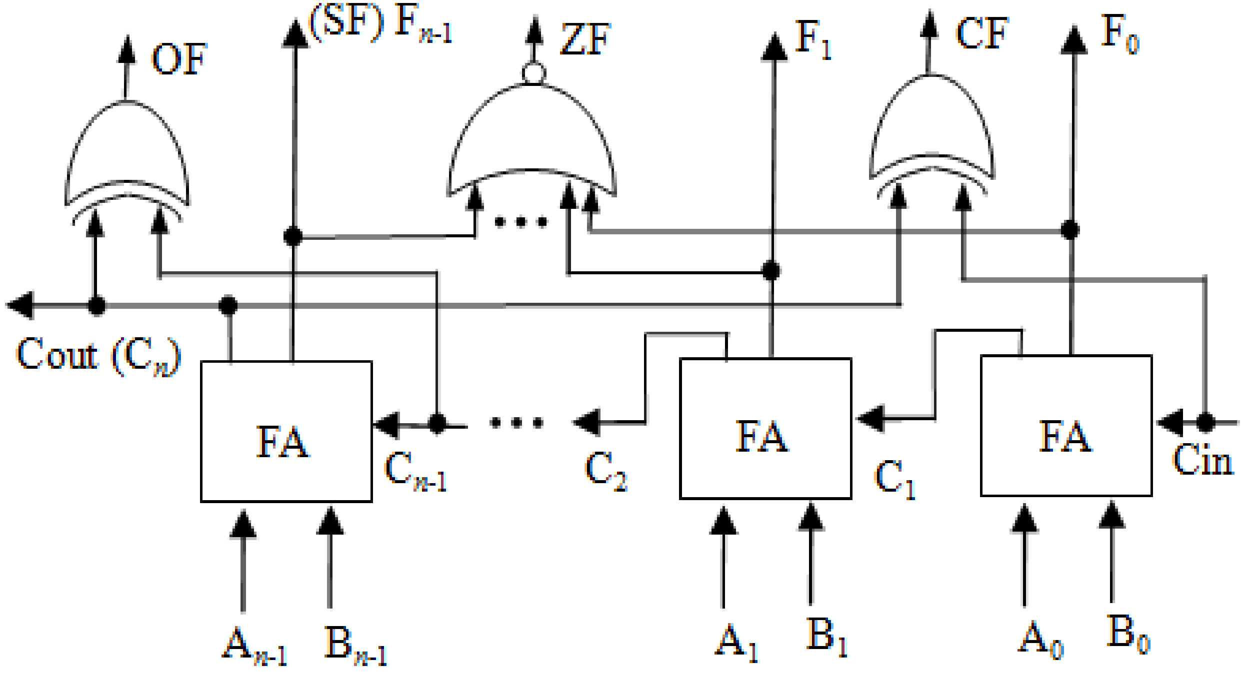
第一讲 基本运算部件

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）

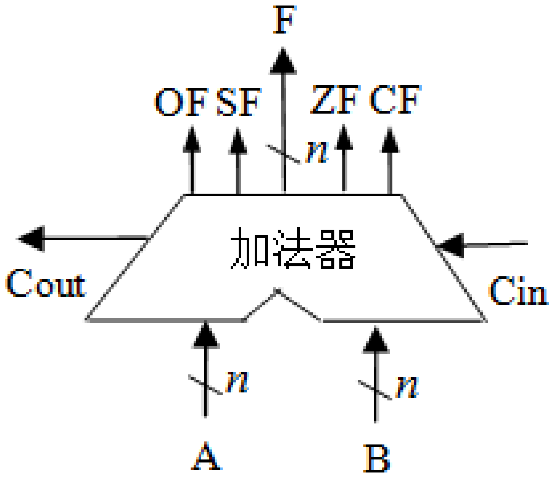
n位带标志加法器

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



带标志加法器的逻辑电路



带标志加法器符号

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

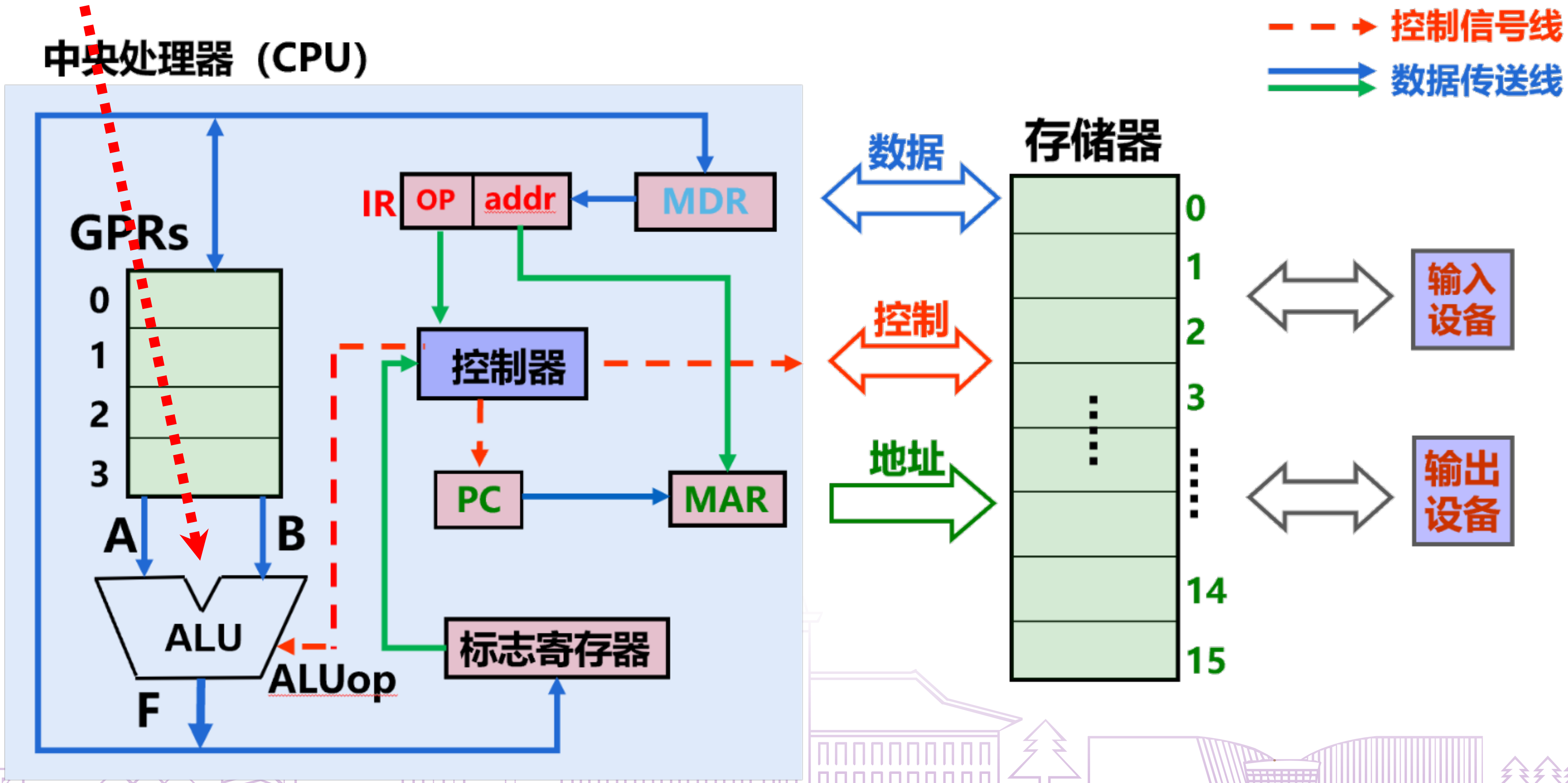
1. 高级语言程序中所涉及到的运算（以C语言为例）
   1. 整数算数运算、浮点数算数运算
   2. 按位、逻辑、移位、位扩展和位截断
2. 串行进位加法器
3. 并行进位加法器
   1. 全先行进位加法器
   2. 两级/多级先行进位加法器
4. 带标志加法器
5. 算数逻辑部件（ALU）

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

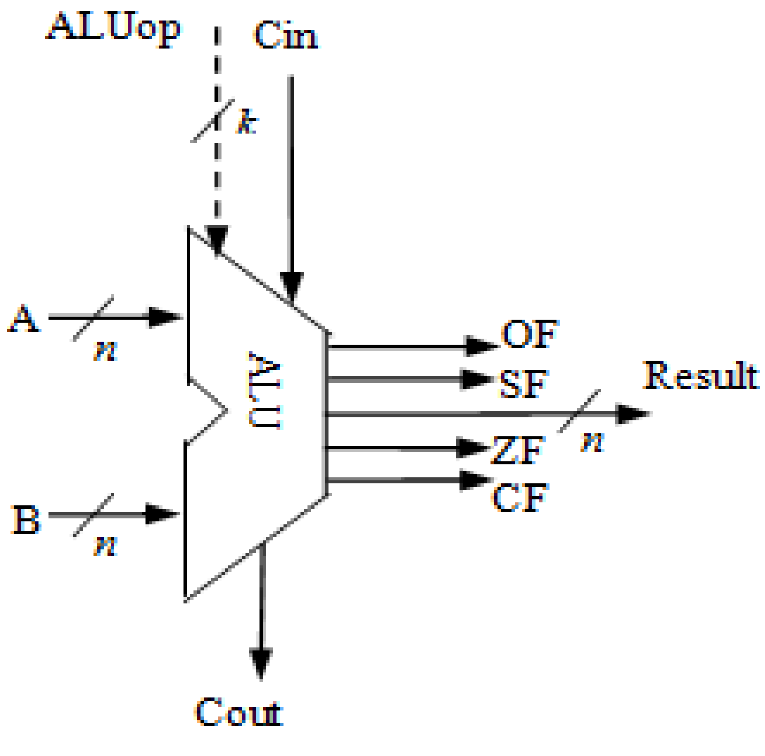
ALU：算术逻辑部件；IR：指令寄存器；MDR：存储器数据寄存器

GPRs：通用寄存器组（由若干通用寄存器组成）



算数逻辑部件（ALU）

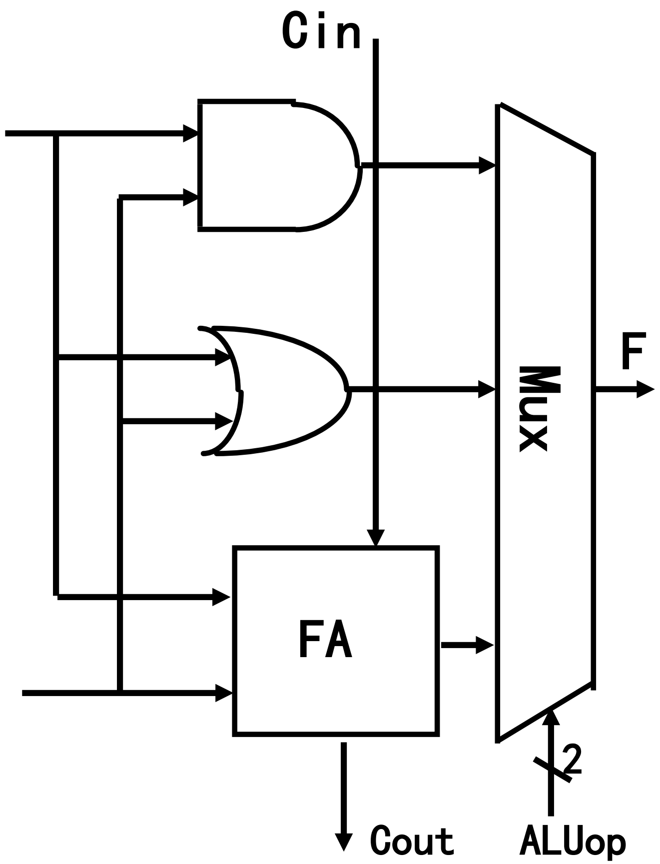
- 进行**基本**算术运算与逻辑运算
  - 无符号整数加、减
  - 带符号整数加、减
  - 与、或、非、异或等逻辑运算
- 核心电路是**整数加/减运算部件**
- 输出除**和/差等**，还有**标志信息**
- 有一个**操作控制端**（ALUop），用来决定ALU所执行的处理功能。ALUop的位数k决定了操作的种类--例如：当位数k为3时，ALU最多只有2^3=8种操作。



ALU 符号

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⊕B	1 1 1	未用

举例：1-bit ALU和4-bit ALU



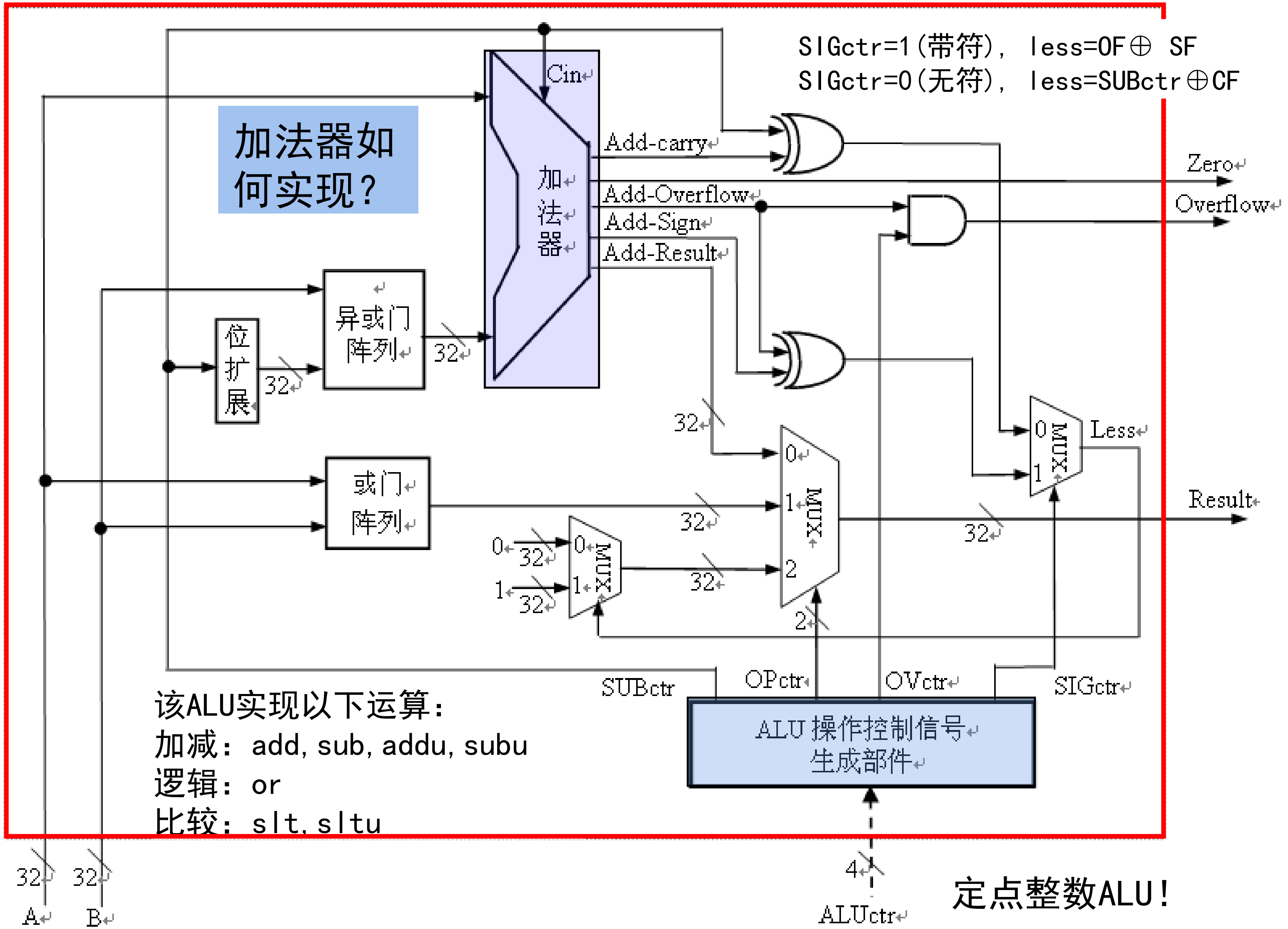
1-bit ALU

4位先行进位ALU的实现

```
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



该ALU实现以下运算：

加减：add, sub, addu, subu

逻辑：or

比较：slt, sltu

定点整数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;
```

补码的定义　　假定补码有n位，则：

$[X]_{补}=2^n+X \quad (-2^{n-1} \leqslant X < 2^{n-1}, \bmod 2^n)$

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

因此，计算机中需要有一个电路，能够实现以下功能：

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

根据补码定义，有如下公式：

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

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

n位整数加减运算器

- 补码加减运算公式

  $[A+B]_{补} = [A]_{补} + [B]_{补}$ (mod 2^n)

  $[A–B]_{补} = [A]_{补} + [–B]_{补}$ (mod 2^n)

  – 实现减法的关键工作在于：求$[–B]_{补}$

问题：如何求$[–B]_{补}$？

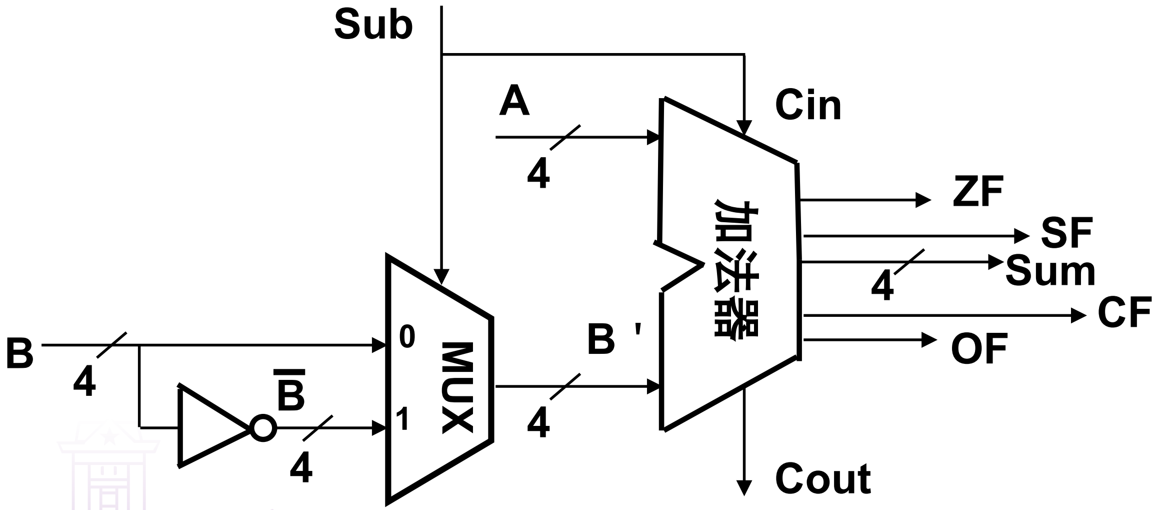
$[–B]_{补}=[\overline{B}]_{补}+1$

当Sub为1时，做减法
当Sub为0时，做加法

- 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



整数加/减运算部件

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

回顾 – 补码溢出判断

- 概念：在定点数运算中，运算结果超出了机器数的表示范围
- 判断方法：
  - 1. 人工用数值计算：和，如果不超出范围，就不会溢出。
  - 2. 人工用补码计算：如果参加运算的两个数符号相同，运算后符号改变，则发生了溢出

    （注意：符号相异的数相加不会产生溢出）
  - 3.人工用变形补码（模4补码）计算：

    00-无溢出　01-正溢出　10-负溢出　11-无溢出

    （注意：仅在加法器部分实现，在寄存器和存储器中仍然是普通补码表示）
  - 4. 若符号位进位与最高数值位进位相同，则无溢出

回顾 – 补码溢出判断

➢完备归纳证明：若符号位进位与最高数值位进位相同，则无溢出*

符号位进位 (=最高位进位)	最高数值位进位 (=次高位进位)	加数类型	溢出状态
0	0	正+正 正+负，负+正	0
0	1	正+正	1
1	0	负+负	1
1	1	负+负 正+负，负+正	0

整数加减法中的标志位

带符号数

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;

无符号数

进位/借位标志CF：

$CF=Cout \oplus Cin$

正数相加： C_n=0，C_{n-1}如果等于1，则结果为负，溢出，等于0，结果正常。

例如：0101(5)+0110(6)=1011(-5), 溢出； C_n=0，C_{n-1}= 1

0101(5)+0010(2)=0111(7), 正常； C_n=0，C_{n-1}= 0

异号相加： $C_n=C_{n-1}$，则OF一定为0，结果正常。

负数相加： C_n=1，C_{n-1}如果等于0，则结果为正，溢出，等于1，结果正常。

例如：1101(-3)+1110(-2)=1011(-5), 正常； C_n=1，C_{n-1}= 1

1101(-3)+1010(-6)=0111(7), 溢出； C_n=1，C_{n-1}= 0

加法指令：Cin=0，Cn=Cout，Cout=1，有进位。

减法指令：Cin=1，Cn=Cout，Cout=0，有借位。

例如：有符号 1101(-3)-1110(-2)= 1101(-3)+0001+1= 1111(-1),

无符号 1101(13)-1110(14)= 1101(13)+0001+1= 1111(15), 错 C_n=0，不够减

有符号 1101(-3)-1010(-6)= 1101(-3)+0101+1= 0011(3),

无符号 1101(13)-1010(10)= 1101(13)+0101+1= 0011(3), 对 C_n=1，够减

整数减法举例（注意标志位的使用）

-7- 6 = -7 + (-6) = +3 **X**
9 - 6 = 3 √

	1	0		
		1	0	0	1
+		1	0	1	0
		0	0	1	1

OF=1、ZF=0
SF=0、借位CF=0

-3 - 5 = - 3 + (- 5) = - 8 √
13 - 5 = 8 √

	1	1	1	1	
		1	1	0	1
+		1	0	1	1
		1	0	0	0

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

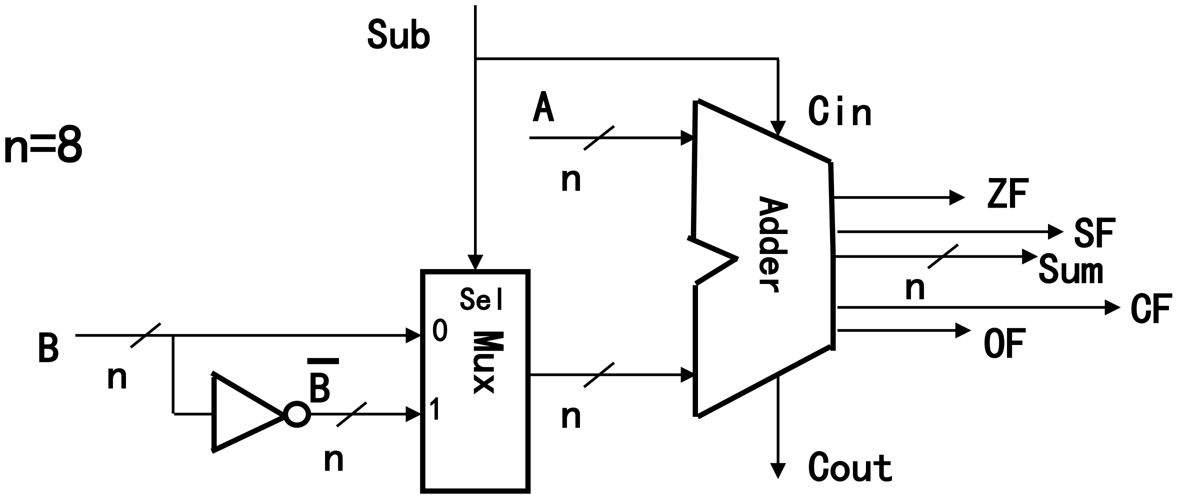
验证：-7<6，故OF≠SF
-3<5，故OF≠SF

整数减法举例（高级语言的对应）

```
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;
```

无符号和带符号加减运算都用该部件执行

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：

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

带符号减公式：

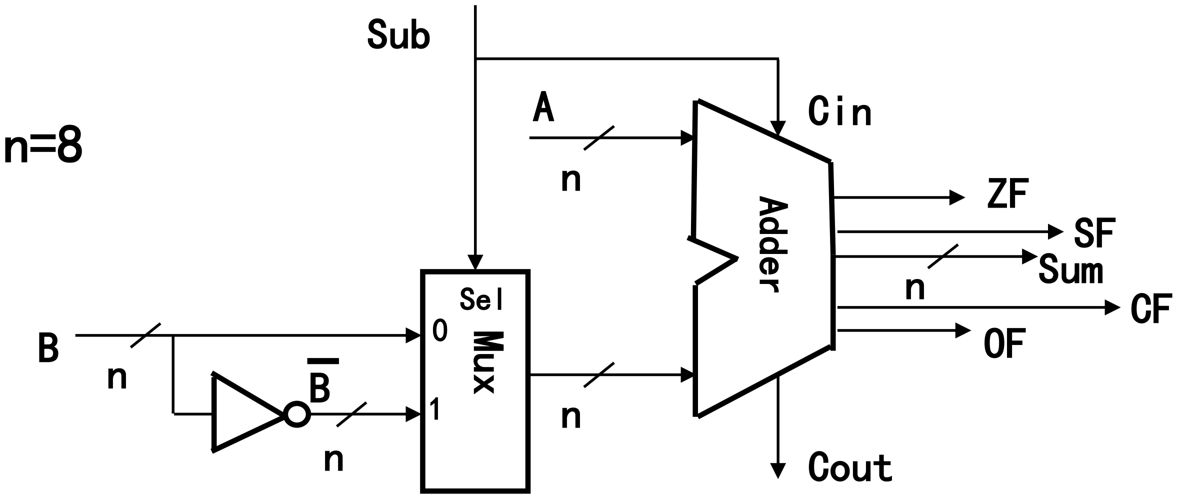
$$result=\begin{cases} x-y-2^n & (2^{n-1}\leqslant x-y) & \text{正溢出} \\ x-y & (-2^{n-1}\leqslant x-y<2^{n-1}) & \text{正常} \\ x-y+2^n & (x-y<-2^{n-1}) & \text{负溢出} \end{cases}$$

整数加法举例（高级语言的对应）

```
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;
```

无符号和带符号加减运算都用该部件执行

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：0111 1100，CF=1，OF=1，SF=0

z2的值为124（=134+246-256，x+y>256），k2的值为124(=134+246-256,负溢出)。

无符号加公式：

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

带符号加公式：

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

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）

$$[E1]_{移}+[E2]_{移}=2^{n-1}+E1+2^{n-1}+E2=2^n+E1+E2=[E1+E2]_{补} \quad (\mathrm{mod}\ 2^n)$$

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

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

$$=2^n+E1-E2=[E1-E2]_{补} \quad (\mathrm{mod}\ 2^n)$$

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$[E1]_{移}$和$[E2]_{移}$进行模2^n加，然后对结果的符号取反。

② 减法：先将减数$[E2]_{移}$求补（各位取反，末位加1），然后再与被减数 $[E1]_{移}$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：$[-7]_{移} = 0001$　　$[-6]_{移} = 0010$　　$[-3]_{移} = 0101$　　$[6]_{移} = 1110$

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

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

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

例2：　用四位移码计算“－7 －（－　6）”和“－3 －　5”的值。

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

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

$[-3]_{移} - [5]_{移} = 0101 + 0011 = 1000$，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：$[X]_{原}=x_0.x_1\ldots x_n$，$[Y]_{原}=y_0.y_1\ldots y_n$ ，求$[x\times Y]_{原}$

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

Paper and pencil example:

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

$n=4$

Multiplicand	1000	
Multiplier	x 1001	
	1000	$X\times y_4\times 2^{-4}$
	0000	$X\times y_3\times 2^{-3}$
	0000	$X\times y_2\times 2^{-2}$
	1000	$X\times y_1\times 2^{-1}$
Product(积)	0.1001000	

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×yi，若yi = 0，则得0；若yi = 1，则得X
  - 把①求得的各项结果X× yi 逐次左移，可表示为X× yi×2^{-i}
  - 对②中结果求和，即 ∑（X× yi×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×yi后，与前面所得结果累加，得到Pi，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×yi的开销。
  - 每次得X×yi后，不将它左移与前次部分积Pi相加，而将部分积Pi右移后与X×yi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：

$$X \times Y = X \times (0.y_1 y_2 \cdots y_n)$$

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

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

$$= \underbrace{2^{-1}(2^{-1}(2^{-1} \cdots 2^{-1}(2^{-1}}_{n个2^{-1}}(0 + X \times y_n) + X \times y_{n-1}) + \cdots\cdots + X \times y_2) + X \times y_1)$$

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：

$$P_1 = 2^{-1}(P_0 + X \times y_n)$$

$$P_2 = 2^{-1}(P_1 + X \times y_{n-1})$$

…… ……

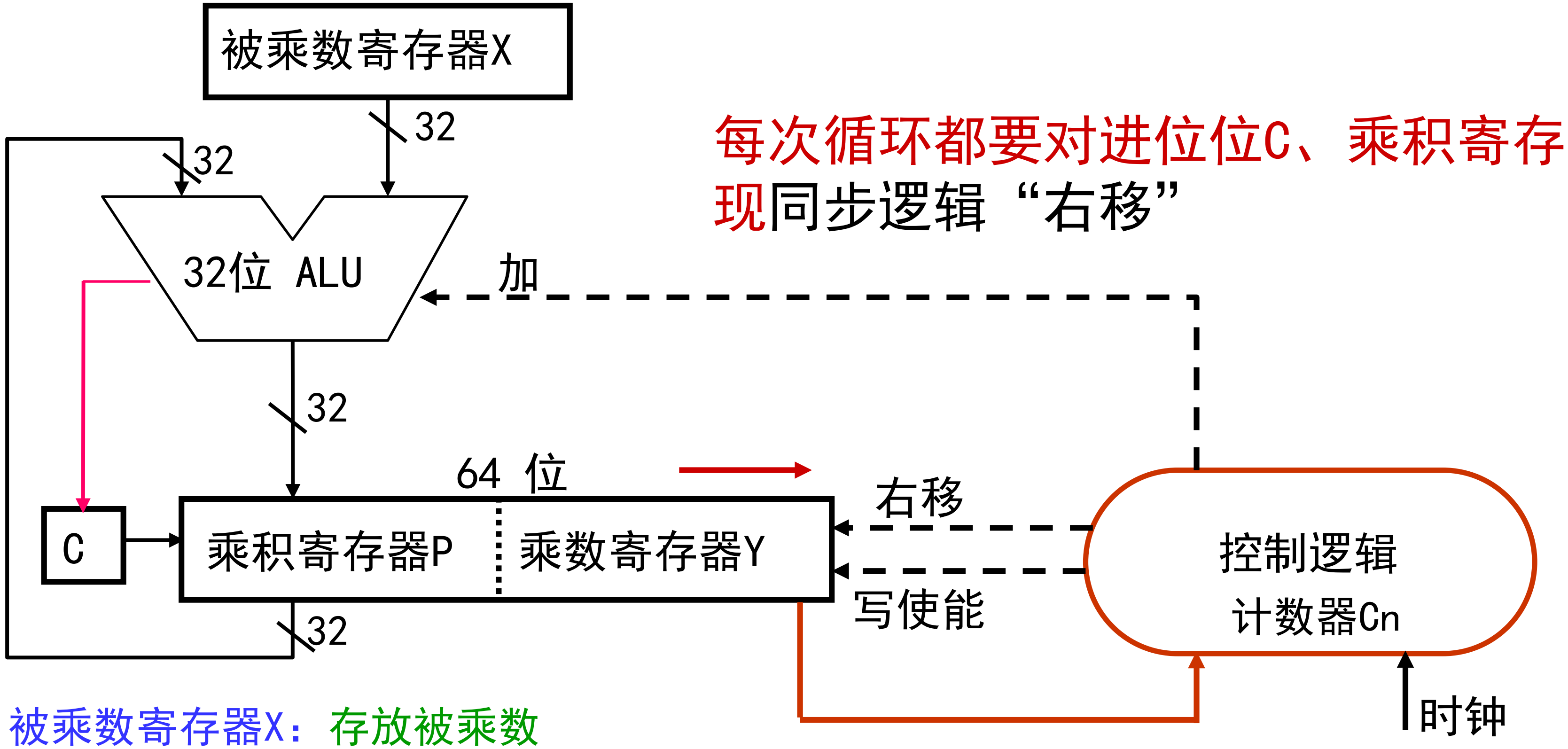
$$P_n = 2^{-1}(P_{n-1} + X \times y_1)$$

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Pn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



每次循环都要对进位位C、乘积寄存器P和乘数寄存器实现同步逻辑“右移”

- 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P0 = 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：积符异或得到，数值用无符号乘法运算

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

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

符号位：0 ⊕ 1=1，所以： $[x\times y]_原$=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：**

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

$=2^{-2}(Pi-X)+X$

触发器T用来记录下次是否要执行“+X”

“–X”运算用“$+[-X]_补$”实现！

3X时，本次-X，下次+X！

部分积右移两位，相当于4X

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

原码两位乘法举例

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

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

$[|X|]_补$ = 000 111001，$[-|X|]_补$ = 111 000111

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 $[X\times Y]_原$=0. 100010101111

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。

速度快，但代价也大

补码乘法运算

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

因为$[X\times Y]_补 \neq [X]_补\times[Y]_补$，故不能直接用无符号数乘法计算

基于以下补码性质：

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

令：$y_{-1}=0$，则：

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

$$\downarrow$$

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

$$\downarrow$$

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

$[X]_补$

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

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

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

Booth’s 算法实质

middle of run

end of run | 0 | 1 | 1 1 | 1 | 0 | beginning of run

• 当前位	右边位	操作	Example
1	0	减被乘数	0001111000
1	1	加0（不操作）	0001111000
0	1	加被乘数	0001111000
0	0	加0（不操作）	0001111000

• 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。

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

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

P	Y	y_{-1}	说明
0000	0110	0	设 $y_{-1}=0$，$[P_0]_补=0$
			$y_0\,y_{-1}=00$，P、Y 直接右移一位
0000	0011	0	得$[P_1]_补$
+0011			$y_1\,y_0=10$，$+[-X]_补$
0011			P、Y 同时右移一位
0001	1001	1	得$[P_2]_补$
			$y_2\,y_1=11$，P、Y 直接右移一位
0000	1100	1	得$[P_3]_补$
+1101			$y_3\,y_2=01$，$+[X]_补$
1101			P、Y 同时右移一位
1110	1110	0	得$[P_4]_补$

验证：当X×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 $[X]_{补}$ = 1 101， $[Y]_{补}$ = 0 110，用补码两位乘法计算$[X\times Y]_{补}$。
- 解：$[-X]_{补}$= 0 011，用补码二位乘法计算$[X\times Y]_{补}$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，$[P_0]_{补}$ = 0
+ 0 0 1 1 0			$y_1y_0y_{-1}$ =100，+2$[-X]_{补}$
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$[P_2]_{补}$
+ 1 1 0 1 0			$y_3y_2y_1$ = 011，+2$[X]_{补}$
1 1 0 1 1		$\rightarrow 2$	P和Y同时右移二位
1 1 1 1 0	1 1 1 0		得$[P_4]_{补}$

因此 [X × Y]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 ×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 |1001010               Dividend(被除数)
             -1000
              -------
                  10
                  101
                ---------
                  1010
                 -1000
                  ------
                    10              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≠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≠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≠ 0，且商≠ 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→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！

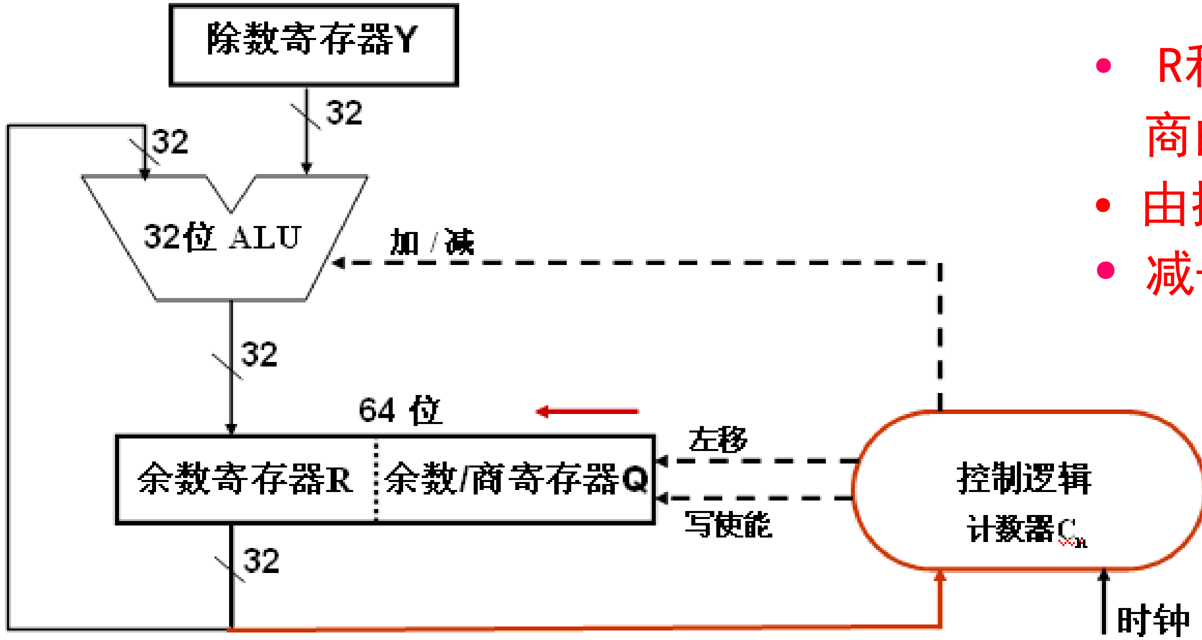
通常意义下，若是2n位除以n位的无符号整数运算，则说明将会得到n+1位的商，因而结果“溢出”（即：无法用n位表示商）。

例：1111 1111/1111 = 1 0001

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

最大商为:0000 1111/0001=1111

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Cn：存放循环次数。初值是32，每循环一次，Cn减1，当Cn=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: $\boxed{0001}$ 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
- 若R_i>=0,则商上“1”，不需恢复余数

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 = 1110

“不恢复余数法”例子

	D: 0010	R: 0000 0111
Shl R	D: 0010	R: 0000 1110
R = R–D	D: 0010	R: 1110 1110
sl R, 0	D: 0010	R: 1101 1100
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: $\boxed{0001}$ 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。

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*