

Lecture 3: 运算方法和运算电路

数电基础

二值运算

概念：二值运算

- 与或非运算：



- 异或运算：

- 符号为 **XOR** 或 **EOR**；
- 异或算符的值为真，仅当两个运算元中 **恰有一个的值为真，而另外一个的值为非真**；

异或
(XOR)



$$F = X\bar{Y} + \bar{X}Y \\ = X \oplus Y$$

X	Y	F
0	0	0
0	1	1
1	0	1
1	1	0

概念：组合逻辑电路与时序逻辑电路

- 概念：什么是组合逻辑电路

- 介绍：

- 组合逻辑电路是指其输出仅取决于当前输入信号的电路，而与输入信号的历史状态无关。换句话说，组合逻辑电路没有记忆功能，输出的变化仅由当前的输入决定；

- 特点：

- 无记忆功能**：电路的输出只与当前的输入信号有关，不依赖于电路过去的状态；
- 稳定输出**：在输入信号稳定后，电路的输出会迅速稳定在一个确定的值；
- 任意时刻的输出**：对于给定的输入值，电路的输出是唯一确定的，不受时间影响；

- 举例：

- 加法器；
- 编码器；
- 多路选择器；

- 概念：什么是时序逻辑电路

- 介绍：

- 时序逻辑电路是指其输出不仅取决于当前的输入信号，还取决于电路过去的状态，即电路具有记忆功能；
- 这种电路通常包含存储元件，如触发器或寄存器，用来存储信息；

- 特点：
 - 1. **有记忆功能**：电路的输出不仅取决于当前的输入，还取决于之前的状态；
 - 2. **状态变化**：输出的变化通常在特定的时钟信号或控制信号的边沿发生；
 - 3. **时间相关**：输出与输入信号的时序有关，电路的行为随时间变化；
- 举例：
 - 寄存器；

组合逻辑电路

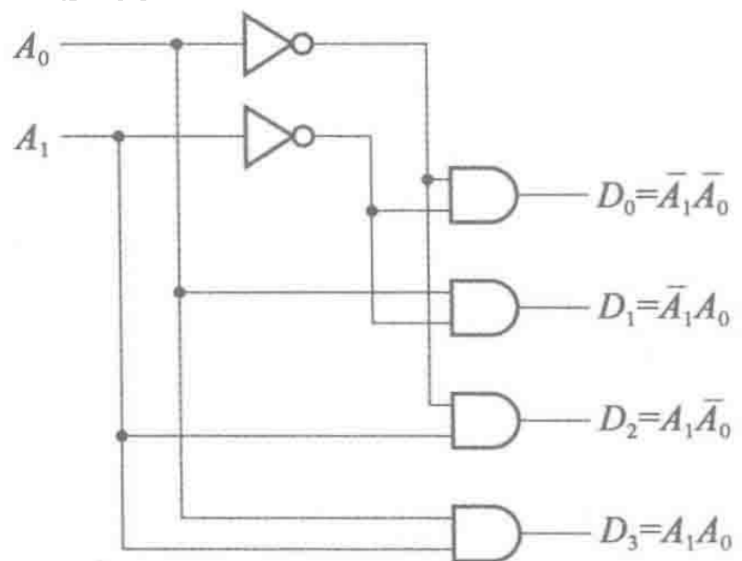
译码器与使能信号

概念：译码

- 概念：
 - 在数字计算机中，大量的离散信息都用二进制码表示，一种 n 位二进制码可以表示多达 2^n 个不同的编码信息。译码就是将一个 n 位的输入码转换成一个 m 位的输出码，并且有 $n \leq m \leq 2^n$ ，以保证每一个有效的输入码能产生唯一的输出码；
- 实现：
 - 译码通过译码器 (decoder) 来完成，译码器是一个组合电路，在它的输入端加载一个 n 位的二进制码，它就会在输出端产生一个 m 位的二进制码；
 - 译码器输入的某些组合可能没有，故在输出端也就得不到对应的 m 位码
- 举例：2-4译码器

A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

a)



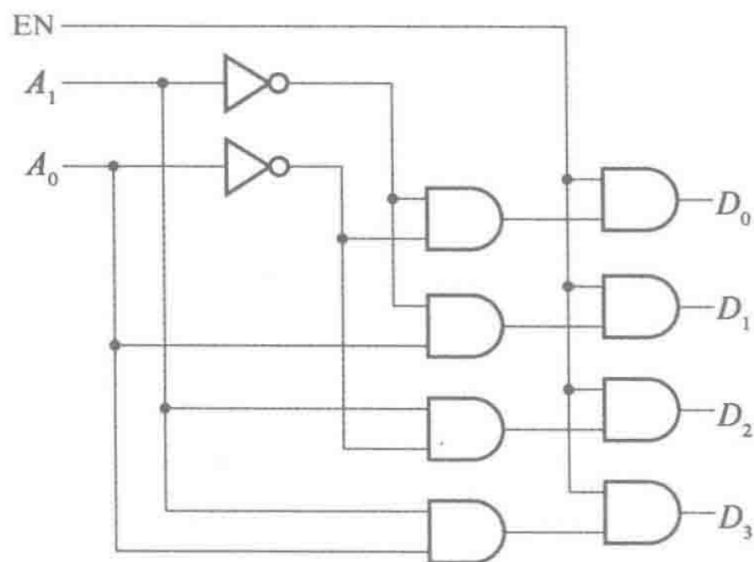
b)

概念：使能

- 概念：
 - 在数字电路中，使能信号常用于控制逻辑门的运行。逻辑门根据输入信号和使能信号的组合来确定输出；
 - 当使能信号为高电平时，逻辑门会根据输入信号执行相应的逻辑操作并生成输出；
 - 当使能信号为低电平时，逻辑门将忽略输入信号，并输出特定状态；
- 举例：带使能的2-4译码器

EN	A_1	A_0	D_0	D_1	D_2	D_3
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

a)



b)

多路选择器

概念：多路选择器

• 定义：

- 多路复用器是一个组合电路，它可以从多条输入中选择一个输入，并将信息直接传输到输出；
- 选择哪一条输入线由一组输入变量控制，它们被称为选择输入 (Selection Input)；

• 举例：2-1多路复用器

• 解释：

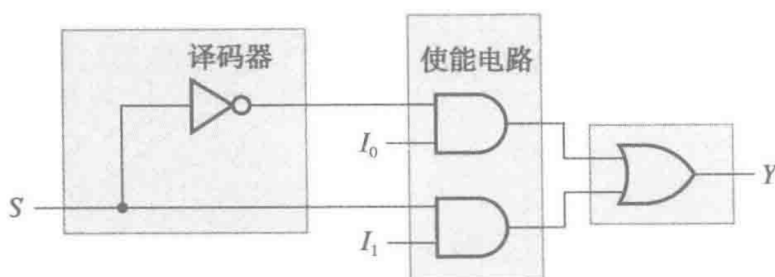
- 这个函数有两个信息输入： I_0 和 I_1 ，一个单独的选择输入 S ；
- 如果选择输入 $S=0$ ，多路复用器输出为 I_0 的值。如果选择输入 $S=1$ ，多路复用器输出 I_1 的值；

• 真值表：

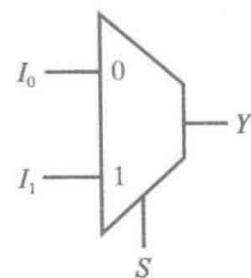
S	I_0	I_1	Y	S	I_0	I_1	Y
0	0	0	0	1	0	0	0
0	0	1	0	1	0	1	1
0	1	0	1	1	1	0	0
0	1	1	1	1	1	1	1

• 电路：

- S 会译码产生 0 和 1 两个信号；
- 使能电路通过进行 与运算，保留所需要的信号，进而产生结果 Y ；



a) 一位2-1多路复用器



b) 多路复用器常用符号

二进制加法器

概念：二进制加法器的实现

- 概念：加法器
 - 前提：
 - 设计一个电路实现两个一位二进制数相加，这个简单的加法包含 4 个可能的基本操作：
 - $0+0=0$, $0+1=1$, $1+0=1$, $1+1=10$;
 - 前三个运算产生的和只需要一位表示，但是当加数和被加数都等于 1 的时候，和就需要两位表示；
 - 因为如此，运算的结果需要两位表示：进位与和；
 - 半加器：
 - 由两位加法产生的进位将加到下一个高位的有效位中，实现两位相加的组合电路称为半加器 (halfadder)；
 - 全加器：
 - 实现三个位 (两个有效位和一个先前位产生的进位) 相加的电路称为全加器 (full adder)；
 - 关系：
 - 两个半加器可以用来实现一个全加器；
 - 半加器和全加器的命名就是基于这个原因；
 - 半加器：
 - 半加器是一个产生两位二进制数的和的算术电路，这个电路有两个输入和两个输出；
 - 输入：
 - 加数和被加数；
 - 输出：
 - 产生的和；
 - 进位；
 - 真值表与逻辑电路图：

输入		输出	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

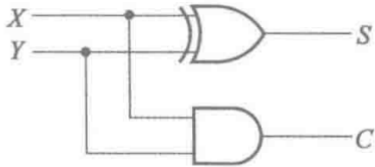
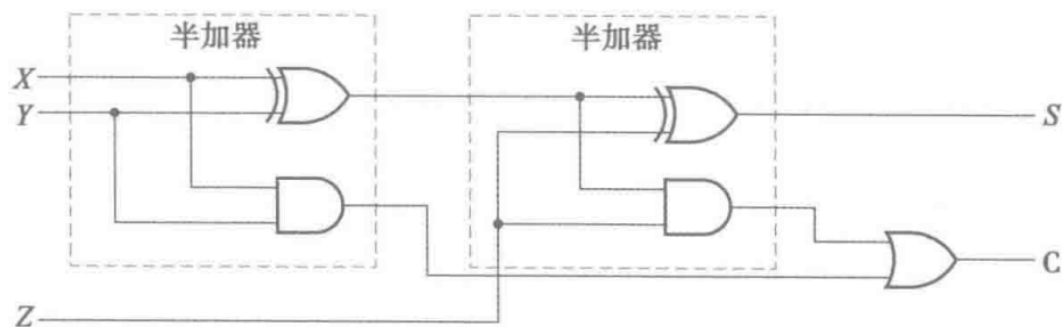


图 3-40 半加器的逻辑电路图

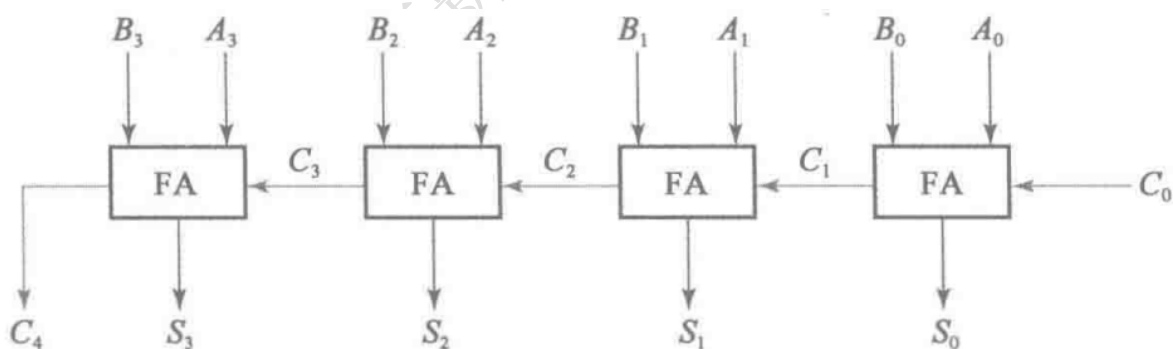
- 全加器：
 - 全加器是实现三位数相加的组合逻辑电路，除了三个输入，全加器还有两个输出；
 - 三位相加的和在 0~3 之间变化，而 2 和 3 需要两位二进制数表示；
 - 真值表和逻辑电路图：

输 入			输 出		输 入			输 出	
X	Y	Z	C	S	X	Y	Z	C	S
0	0	0	0	0	1	0	0	0	1
0	0	1	0	1	1	0	1	1	0
0	1	0	0	1	1	1	0	1	0
0	1	1	1	0	1	1	1	1	1



补充：二进制行波进位加法器

- 概念：
 - 一个并行加法器是一个仅采用组合逻辑计算出两个二进制数算术和的数字电路；
 - 并行加法器并行地连接 n 个 全加器，所有的输入位同时加载至 全加器，以产生和；
 - 并行加法器中的所有 全加器 用 级联 的方式连接在一起，一个全加器的进位输出连接到下一个全加器的进位输入；
 - 由于加法器最低有效位产生的进位 1 可能经过多个全加器传递到最高有效位，就好像一个小卵石丢入池塘激起的波浪一样，因此这种并行加法器又称为行波进位加法器；
- 图示：



时序电路

概念：什么是时序电路

- 概念：
 - 通常由组合电路和存储元件连接在一起组成，存储元件是用来存储二进制信息的电路，某一时刻存储在这些存储元件中的二进制信息称为该时刻时序电路的状态 (state)；
 - 时序电路通过输入从外界接收二进制信息，这些输入值和存储元件的当前状态一起决定了输出端的二进制值，同时也决定了存储元件的下一个状态；
 - 从下面的图示可以看出，时序电路的输出不仅是其输入的函数，而且还是存储元件当前状态的函数，存储元件的下一个状态也是电路输入以及当前状态的函数。因此，时序电路可以由输入、内部状态和输出的时间序列完全确定；

- 图示：

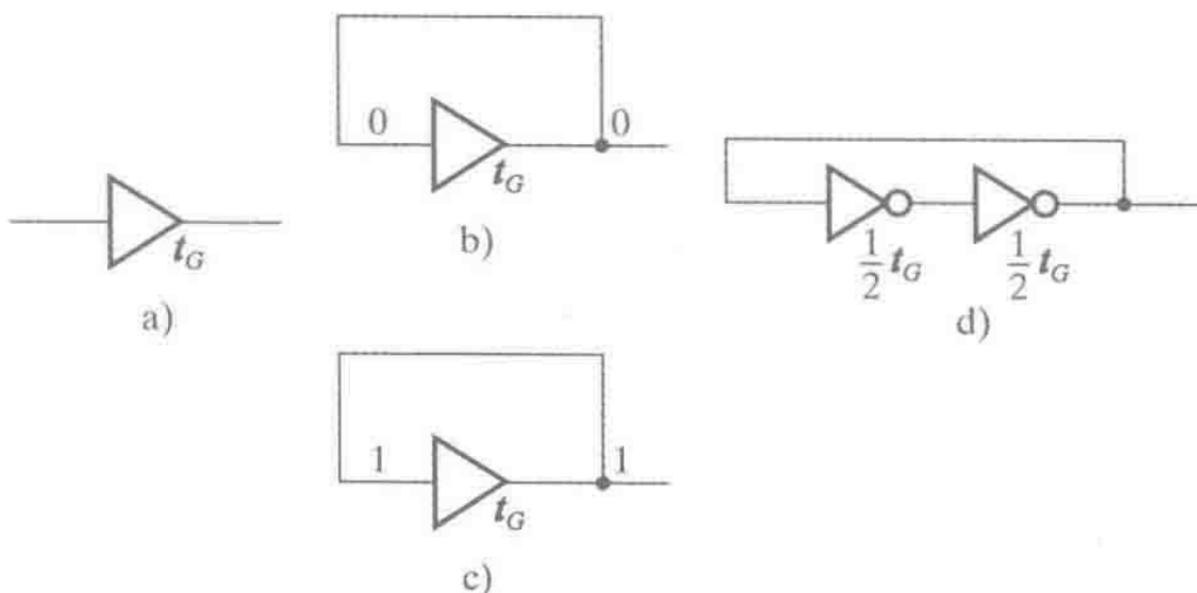


概念：缓冲器与信号存储

- 概念：

- 假设图 4-2a 中缓冲器的输出端连接到它自己的输入端，如图 4-2b 和图 4-2c 所示，再假设图 4-2b 中缓冲器的输入至少在 t (为缓冲器延迟) 时间内保持为 0，那么缓冲器在 $+t$ 时刻的输出为 0；
- 输入和输出之间的这种关系在所有时刻上保持不变，这样 0 就一直被存储下来；
- 可以将具有延送的逻辑连接成一个闭路圈来构造存储器。任何具有存储功能的环路必须像缓冲器一样，信号环绕环路一圈后没有改变；

- 图示：



概念：时钟信号、时钟发生器与触发器

- 概念：时钟

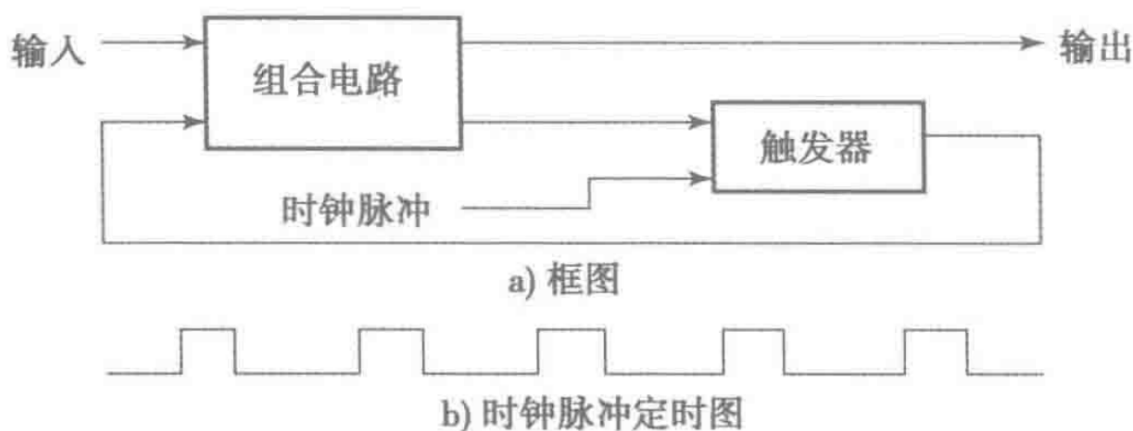
- 时钟发生器产生周期性的时钟脉冲 (clockpulse) 序列，脉冲分布在整个系统中，而同步存储元件只在每一个脉冲特定的时刻上受到影响；
- 实际上，时钟脉冲被用来与其他信号一起改变存储元件中的数据。存储元件的输出仅在出现时钟脉冲时发生变化；
- 把时钟脉冲作为存储元件输入信号的同步时序电路称为钟控时序电路，这种类型的电路在实际中广泛应用，因为不管电路延迟有多大的差别它们都能保证操作的正确性，同时设计也比较容易；

- 概念：触发器

- 介绍：

- 在最简单的钟控时序电路中使用的存储元件称为**触发器**；

- 接收输入信号同时，也从时钟中接收具有固定时间间隔的脉冲信号；
- 从一个状态到另一个状态的跳变只发生在由时钟脉冲所决定的固定时间间隔内，由此实现同步操作；
- 触发器有一个或两个输出端，其中一个存储正常值，另一个存储其反值；
- 图示：



基本运算部件

概念

- 组成：
 - 运算器由算术逻辑单元（ALU）、移位器、状态寄存器（PSW）和通用寄存器组等组成；
- 功能：
 - 基本功能包括加、减、乘、除四则运算；
 - 与、或、非等逻辑运算；
 - 移位、求补等操作；

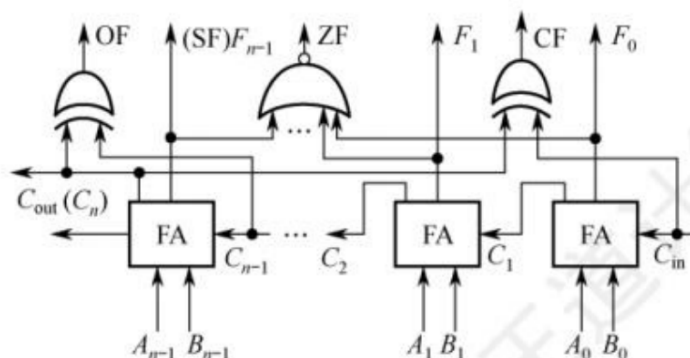
带标志加法器

概念

- 概念：
 - 无符号数加法器只能用于两个无符号数相加，不能进行有符号整数的加/减运算；
 - 为了能进行有符号整数的加/减运算，还需要在无符号数加法器的基础上，增加相应的逻辑门电路，使得加法器不仅能计算和/差，还要能生成相应的标志信息；
- 图示：



(a) 带标志加法器符号

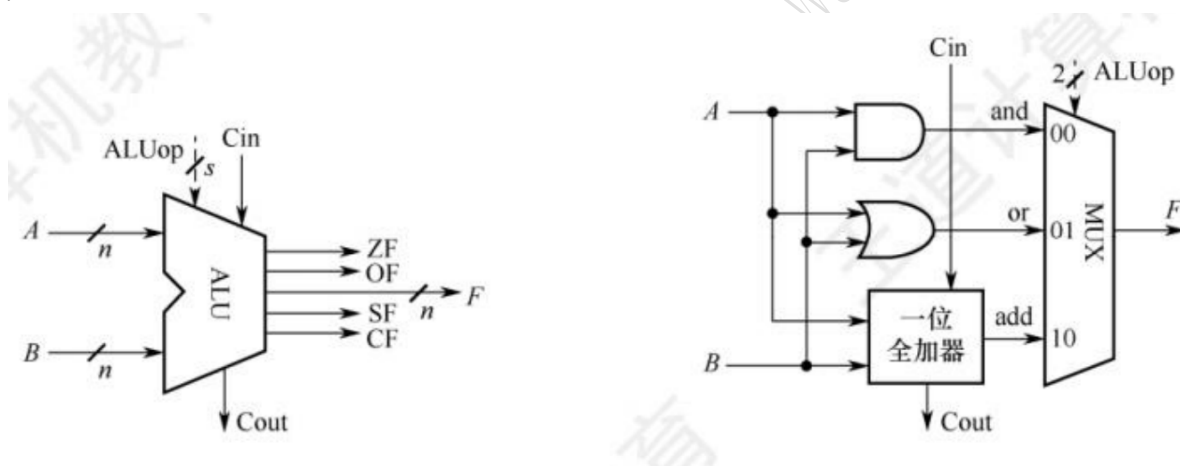


(b) 带标志加法器的逻辑电路

算术逻辑单元

概念

- 概念：
 - 算数逻辑单元，简称 **ALU**；
 - ALU** 能进行多种**算术运算和逻辑运算**；
 - 由于加、减、乘、除运算最终都能归结为加法运算，因此 **ALU** 的核心是带标志加法器；
- 解释：
 - A** 和 **B** 是两个 **n** 位**操作数**输入端；
 - C_{in} 是进位输入端；
 - ALU_{op} 是操作控制端 (发出控制信号)，用来决定 **ALU** 所执行的处理功能；
 - 例如， ALU_{op} 选择 **Add** 运算，**ALU** 就执行加法运算，输出的结果就是 A 加 B 之和；
- 注意：
 - ALU_{op} 的位数决定了操作的种类 -> 例如当位数为 **3** 时，ALU 最多只有 **8** 种操作；
- 图示：



定点数的移位运算

移位运算

- 概念：
 - 当计算机中没有 **乘/除** 法运算电路时，可以通过**加法和移位相结合的方法来实现乘/除法**运算；
- 左移：
 - 对于任意二进制整数，左移一位，若不产生溢出，相当于**乘以 2** (与十进制数的左移一位相当于乘以 **10** 类似)；
- 右移：
 - 对于任意二进制整数，右移一位，若不考虑因移出而舍去的末位尾数，相当于**除以 2**；
- 分类：
 - 根据操作数的类型不同，移位运算可以分为**逻辑移位**和**算术移位**；

逻辑移位

- 概念：
 - 不考虑符号位；
 - 逻辑移位将操作数视为 **无符号整数**；
- 规则：
 - 左移时，高位移出，低位补 **0**；
 - 右移时，低位移出，高位补 **0**；
 - 对于无符号整数的逻辑左移，若高位的 1 移出，则发生**溢出**；

算术移位

- 概念：
 - 算术移位需要**考虑符号位**的问题，即将操作数视为有符号整数；
 - 计算机中的有符号整数都是用补码表示的，因此对于有符号整数的移位操作应采用**补码算术移位方式**；
- 规则：
 - 左移时，高位移出，低位补 **0**。若移出的高位不同于移位后的符号位，即**左移前后的符号位不同**，则发生溢出；
 - 右移时，低位移出，**高位补符号位**，若低位的 **1** 移出，则影响精度；
- 举例：
 - 补码 **1001** 和 **0101** 左移时会发生溢出，右移时会丢失精度；
 - 补码 **1001** = 反码 **1000** = 原码 = **1111** = **-7**，右移后：**0010** -> 原码为：**0110**，错误数据；

定点数的加减运算

补码的加减运算

概念：补码加减运算

- 公式：

$$\begin{aligned}
 [A + B]_{\text{补}} &= [A]_{\text{补}} + [B]_{\text{补}} \pmod{2^{n+1}} \\
 [A - B]_{\text{补}} &= [A]_{\text{补}} + [-B]_{\text{补}} \pmod{2^{n+1}}
 \end{aligned}$$

- 特点：

1. 按二进制运算规则运算，**逢二进一**；
2. 若做加法，两个数的补码直接相加。若做减法，则将**被减数与减数的负数补码**相加；
3. 符号位与数值位一起参与运算，加、减运算结果的符号位也在运算中直接得出；
4. 最终运算结果的**高位丢弃**，保留 **n+1** 位，运算结果亦为补码；

- 补充：二进制加法运算

1. 计算时，先把两个二进制数对齐（如同十进制一样，从右向左）
2. **1+1** 为 **10**，此时向上一位进 **1,0** 写在本位（如同十进制）
3. 不全为 **1** 的两个数，直接相加得到结果即可；

- 补充：二进制减法运算

- $X - Y = X + \overline{Y} + 1$
- 即：将被减数逐位取反再加一，得到其负数；
- 补充：
 - 题目中进行补码运算时，可以不按照补码的规则计算。全部转换成真值之后、利用真值计算；

举例：补码计算

- 问题：
 - 已知 $A=15$ ， $B=24$ ，求 $[A + B]_{\text{补}}$ 和 $[A - B]_{\text{补}}$
- 解析：
 - $[A + B]_{\text{补}} = 00001111 + 00011000 = 00100111$ ，符号位为 0，对应真值为 + 39。
 - $[A - B]_{\text{补}} = [15]_{\text{补}} + [-24]_{\text{补}} = 00001111 + 11101000 = 11110111$ ，符号位为 1，对应真值为 -9。
 - 注： $[-24]_{\text{补}}$ ： $-24 = -(16 + 8) = 1, 0011000_{\text{原}} = 1, 1100111_{\text{反}} = 1, 1101000_{\text{补}}$

溢出判别方法

概念：溢出的产生

- 概念：
 - 仅当两个**符号相同**的数相加或两个**符号相异**的数相减才可能产生溢出；
 - 如两个正数相加，而结果的符号位却为 **1**（结果为负）；
 - 或一个负数减去一个正数，结果的符号位却为 **0**（结果为正）；
- 方法：
 - (1) 一位符号位；
 - (2) 双符号位；
 - (3) 采用一位符号位根据数值位的进位情况判断溢出；

概念：一位符号位

- 概念：
 - 由于减法运算在机器中是用加法器实现的，因此无论是加法还是减法，只要：
 1. **参加操作的两个数的符号相同**；
 2. **结果与原操作数的符号不同**；
 - 此时则表示结果溢出；
- 实现：
 - 设 A 的符号位 A_s ，B 的符号为 B_s ，运算结果的符号为 S_s ，则溢出逻辑表达式：
 - $$V = A_s B_s \overline{S_s} + \overline{A_s} \overline{B_s} S_s$$
 - 若 $V=0$ ，表示无溢出；若 $V=1$ ，表示有溢出；
- 图示：

★ 分析A+/-B下的OF标志

- 当进行A+B运算时, 若A、B不同符号, 比如A负B正, A正B负时不会发生溢出。溢出只会发生在AB同符号时, 相加超出了最大表示范围;
- 当A和B为正时: A15和B15为0, 如果F15为1, 则表示发生溢出。即: $\overline{A_{15}} \cdot \overline{B_{15}} \cdot F_{15}$
- 当A和B为负时: A15和B15为1, 如果F15为0, 则表示发生溢出。即: $A_{15} \cdot B_{15} \cdot \overline{F_{15}}$
- 只要两种情况当中出现一种就发生溢出, 因此使用或符号: $OF = \overline{A_{15}} \cdot \overline{B_{15}} \cdot F_{15} + A_{15} \cdot B_{15} \cdot \overline{F_{15}}$
- 当进行A-B运算时, 若A、B为两正或两负时, 不会发生溢出, 因为正-正不会得到绝对值更大的数, 同理负-负=负+正;
- 当A为正, B为负时: A15=0, B15=1, 因为正-负=正。如果结果F为负, 即F=1, 则表示发生溢出: $\overline{A_{15}} \cdot B_{15} \cdot F_{15}$
- 当A为负, B为正时: A15=1, B15=0, 因为负-正=负。如果结果F为正, 即F=0, 则表示发生溢出: $A_{15} \cdot \overline{B_{15}} \cdot \overline{F_{15}}$
- 只要两种情况当中出现一种就发生溢出, 因此使用或符号: $OF = \overline{A_{15}} \cdot B_{15} \cdot F_{15} + A_{15} \cdot \overline{B_{15}} \cdot \overline{F_{15}}$

概念: 二位符号位

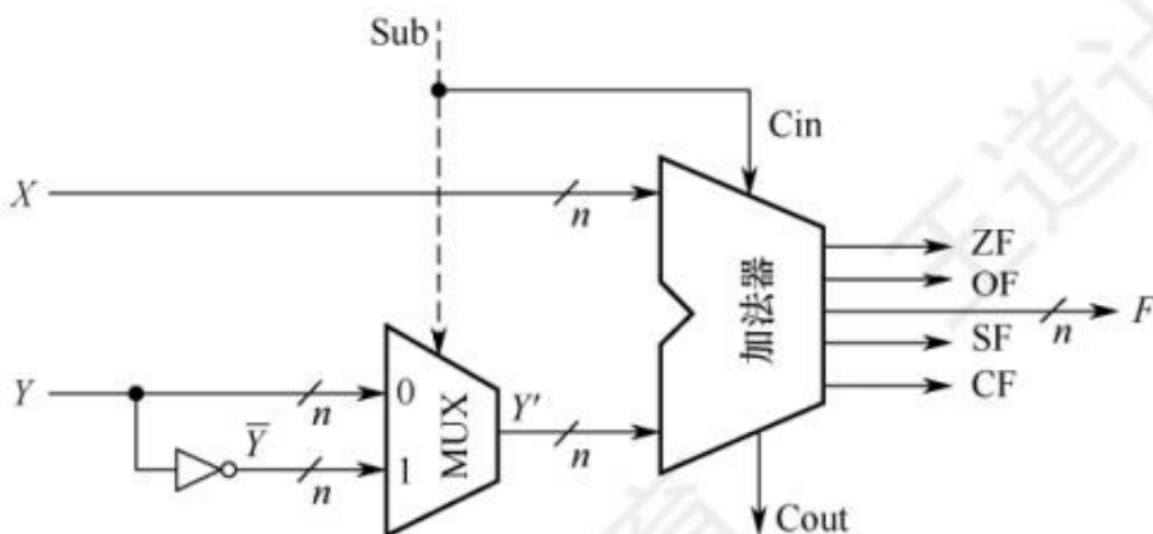
- 概念:
 - 双符号位法也称 模 4 补码;
 - 运算结果的两个符号位 S_{s1} 和 S_{s2} 相同, 表示未溢出;
 - 运算结果的两个符号位 S_{s1} 和 S_{s2} 不同, 表示溢出, 此时**最高位符号位代表真正的符号**;
- 公式:
 1. $S_{s1}S_{s2} = 00$: 表示结果为正数, 无溢出。
 2. $S_{s1}S_{s2} = 01$: 表示结果正溢出。
 3. $S_{s1}S_{s2} = 10$: 表示结果负溢出。
 4. $S_{s1}S_{s2} = 11$: 表示结果为负数, 无溢出。
- 判断: 一位符号位根据数值位的进位情况判断溢出
 - 若符号位 (最高位) 的进位 C_n 与最高数位 (次高位) 的进位 C_{n-1} 相同, 说明无溢出, 否则说明有溢出;
 - 溢出逻辑判断表达式 $V=0$, 表示无溢出; $V=1$, 表示有溢出;

加减运算电路

概念: 加减运算电路

- 概念:
 - 已知一个数的补码表示为 Y , 则这个数的负数的补码为 Y 取反+1;
 - 只要在原加法器的 Y 输入端加 n 个反向器以实现各位取反的功能, 然后加一个 2 选 1 多路选择器, 用一个控制端 Sub 来控制, 以选择是将 Y 输入加法器还是将 Y 取反 输入加法器, 并将 Sub 同时作为低位进位送到加法器 (做减法时实现末位加 1);
- 公式:
 - 当 $Sub=1$ 时 $\rightarrow$ 减法运算: $X + \overline{Y} + 1 = [x]_{补} + [-y]_{补}$
 - 当 $Sub=0$ 时 $\rightarrow$ 加法运算: $X + Y = [x]_{补} + [y]_{补}$
- 图示:

- 无符号整数相当于正整数的补码表示，因此下图电路同时也能实现无符号数的加/减运算；
- 有符号数 x 和 y ，图中 X 和 Y 分别是 x 和 y 的补码表示；
- 对于无符号数 x 和 y ，图中 X 和 Y 分别是 x 和 y 的二进制表示；
- 不论是补码减法还是无符号数减法，都是用被减数加上减数的负数的补码来实现的；



• 注意：

- 运算器本身无法识别所处理的二进制串是有符号数还是无符号数；
- 例如， $0-1=00\dots0+11\dots1=11\dots1$ ；
 - 若解释为有符号数，对应值为 -1 ，结果正确；
 - 若解释为无符号数，对应值为 $2^n - 1$ ，即 n 位无符号数的最大值，结果出错；

加减运算中的标志位

概念：ZF、OF、SF、CF

- **F**：
 - 表示结果；
- **ZF**：
 - **Z** $\rightarrow$ **zero**
 - 表示**零标志位**；
 - **ZF=1** 表示结果为 **0**，对于无符号数和有符号数的运算，**ZF** 都有意义；
- **OF**：
 - 表示**溢出标志位**；
 - 判断**有符号数运算是否溢出**，它是**符号位进位**与**最高数位进位**的异或结果；
 - 即 $OF = C_n \oplus C_{n-1}$
 - 对于无符号数运算，**OF** 没有意义，因为根据 **OF** 无法判断无符号数运算是否溢出；
 - 例如：无符号数加法 $010+011=101$ ，此时 **OF=1**，但结果未溢出；
- **SF**：
 - 表示**结果符号位**；
 - 即 **F** 的最高位。对于无符号数运算，**SF** 没有意义；
- **CF**：

- 表示**进/借位标志**;
- 表示**无符号数运算**时的进位/借位, 判断是否发生溢出;
- 加法时: $CF=1$ 表示结果溢出, 因此 CF 等于进位输出 C_{out}
- 减法时: $CF=1$ 表示有借位, 即不够减, 故 CF 等于进位输出 C_{out} 取反;
- 综合可得 $CF = Sub \oplus C_{out}$
 - 例如: 无符号数加法 $110+011$ 最高位产生进位, 无符号数减法 $000-111$ 最高位产生借位, 结果均发生溢出 (即 $CF=1$);
- 对于有符号数运算, CF 没有意义;

分析: 无符号数大小比较

- 概念:
 - 对于无符号数的运算, 零标志 ZF 、进/借位标志 CF 有意义;
- 实现:
 - 当两个无符号数相减时:
 - 若 $A=B$, 如 $A-B=011-011=000$, 此时结果为零: $ZF=1$, 无借位 $CF=0$;
 - 若 $A>B$, 如 $A-B=010-001=001$, 此时结果非零: $ZF=0$, 无借位 $CF=0$;
 - 若 $A<B$, 如 $A-B=000-001=(1)000-001=111$, 此时: $ZF=0$, 有借位 $CF=1$;
- 图示:

★ 总结: 无符号数加减法

- 设 x, y 是两个16位、补码表示的无符号整数

(1) 无符号数加法

- 当进行无符号加法时, $x+y$ 中都是无符号数, 因此其补码表示和原码表示一致;
- 当 $x+y$ 大于16位时, 发生溢出; (注意是无符号数加法, 不用考虑最高位是否变化)
- 因此 $CF=C_{out}$

(2) 无符号数减法

- 进行无符号减法时, 一定要注意: $x-y$ 中的 y 不能直接和 x 相减, 计算机执行的是 $x+(-y)$, 因此需先把 $(-y)$ 转化为对应的补码形式, 然后再和 x 相加;
- 当 $x>y$ 时, 因为 $x-y=0$, 此时最高位进位是1, 因为 $-y$ 是补码表示。当 $x>y$ 时, 把 $-y$ 转化为补码后、其十六进制数值反而是更大的, 比如: $3-2 = 3+(-2) = 0003H+FFFEH = 1,0001H$
- 由此可知, 在无符号减法时、当 $x>y$ 时, $x-y$ 反而产生了进位。如果是 $x<y$, 反而不产生进位。这一点和无符号加法相反;
- 反之, 如果 $x-y$ 时、 y 的数值更大, 发生溢出, 则此时 $x+(-y)$ 最高位进位为0, $CF=C_{out}$ 取反=1
- 因此 $CF=C_{out}$ 取反

分析: 有符号数大小比较

- 概念:
 - 对于有符号数的运算, 零标志 ZF 、溢出标志 OF 、符号标志 SF 有意义;
- 实现:
 - 若 $A = B$
 - 如 $[A]_{补} - [B]_{补} = 011 - 011 = [A]_{补} + [-B]_{补} = 011 + 101 = (1)000$

- 此时结果为: $ZF=1$
- 最高位进位与次高位进位的异或结果: $OF = C_3 \oplus C_2 = 0$;
- 结果的最高位 $SF=0$

• 结论:

- 减法时, 当 $ZF=1$ 时, 说明 $A=B$;

• 若 $A > B$

- 如 $[A]_{\text{补}} - [B]_{\text{补}} = 010 - 001 = 010 + 111 = (1)001$
 - 此时 $ZF=0, OF=0, SF=0$;
- 若 $[A]_{\text{补}} - [B]_{\text{补}} = 011 - 101 = 011 + 011 = 110$
 - 此时 $ZF=0, OF=1, SF=1$;

• 若 $A < B$

- 如 $[A]_{\text{补}} - [B]_{\text{补}} = 000 - 001 = 000 + 111 = 111$
 - 此时 $ZF=0, OF=0, SF=1$
- 如 $[A]_{\text{补}} - [B]_{\text{补}} = 101 - 011 = 101 + 101 = (1)010$
 - 此时 $ZF=0, OF=1, SF=0$

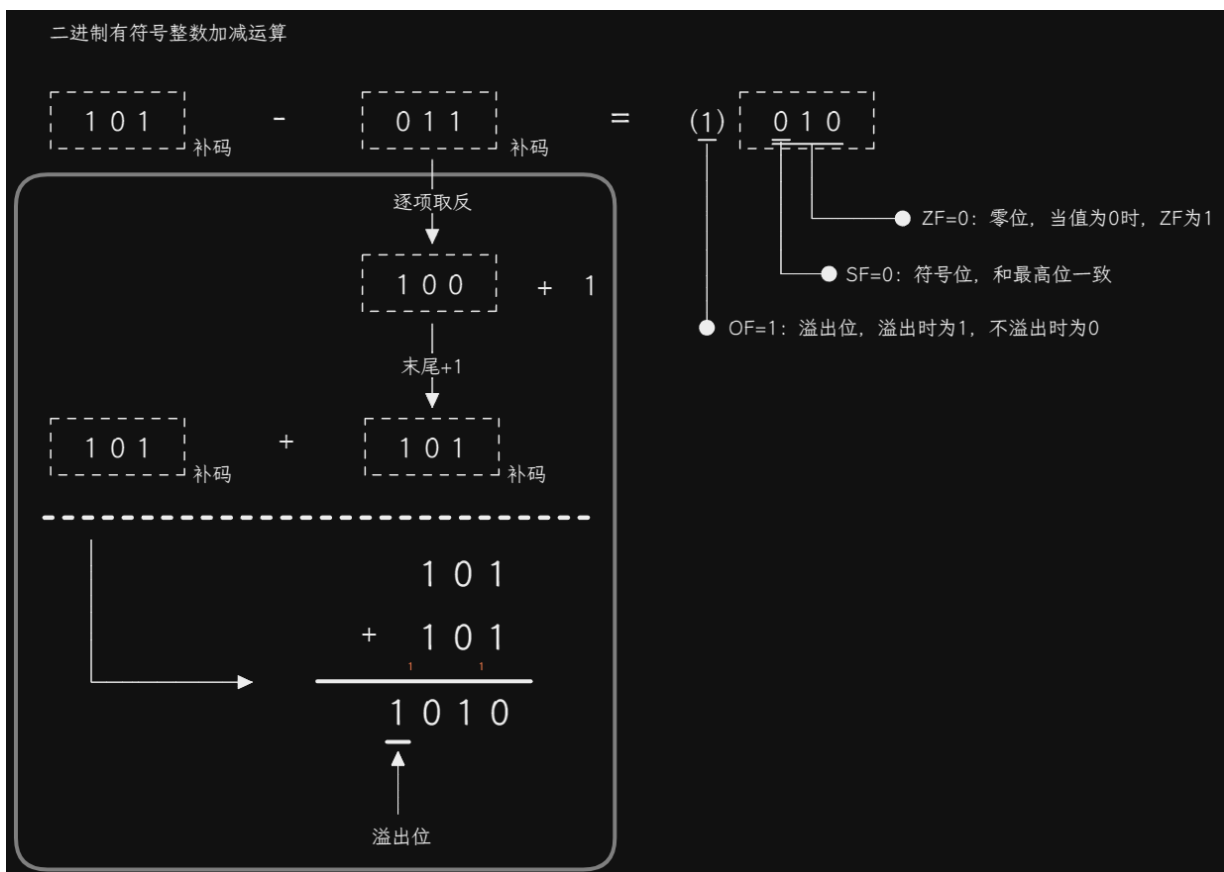
• 描述:

- 1. 当 $ZF=1$ 时, 说明 $A=B$;
- 2. 当 $ZF=0$ 且未发生溢出 ($OF=0$) 时, 若 $SF=0$, 则表示结果非负, 说明 $A>B$;
- 3. 当 $ZF=0$ 且发生溢出 ($OF=1$) 时, 若 $SF=1$, 则必然是**正数减去负数发生溢出**导致结果为负;

• 总结:



• 示例:



原码的加减运算

实现

- 概念:
 - 在原码加减运算中, 将符号位和数值位分开处理;
- 加法规则:
 - 遵循“**同号求和, 异号求差**”的原则, 先判断**两个操作数的符号位**;
 - 符号位相同, 则数值位相加, 结果符号位不变, 若最高数值位相加产生进位, 则发生溢出;
 - 符号位不同, 则做减法, 绝对值大的数减去绝对值小的数, 结果符号位与绝对值大的数相同;
- 减法规则:
 - 先将减数的 **符号取反**, 然后将被减数与符号取反后的减数按原码加法进行运算;

定点数的乘除运算

定点乘法运算

概念: 乘法运算原理 - 原码乘法运算

- 概念:
 - 原码的特点:
 - 符号位和数值位是分开的;
 - 方法:
 - 乘积的**符号位**:

- 由两个乘数的符号位**异或**得到;
- 乘积的**数值位**:
 - 是两个乘数的绝对值之积;
 - 计算起来和十进制类似, 每位进行相乘, 只不过进制是二进制;
- 实现:
 - 乘法运算可以用**加法和移位**实现;
 - 计算机内部采用补码来表示数值数据, 补码的乘法运算可以通过**加法和算术右移**来实现;
 - 编译器可将补码的乘法运算转换为一段循环代码, 其内容包含**比较、加法和算术右移**等指令, 进而实现乘法运算, 这属于软件实现乘法运算;
 - 右移 $\rightarrow$ 乘以 2^{-1}
- 图示:

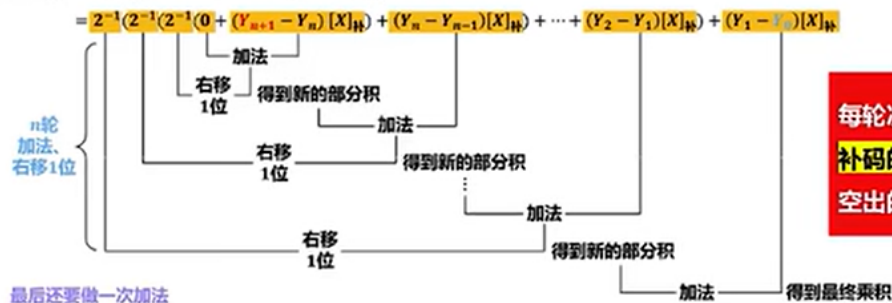
$$\begin{array}{r}
 0.1101 \\
 \times 0.1011 \\
 \hline
 1101 \text{ ----- } X \times y_4 \times 2^{-4} \quad X \times 1 \text{ 右移 4 位} \\
 1101 \text{ ----- } X \times y_3 \times 2^{-3} \quad X \times 1 \text{ 右移 3 位} \\
 0000 \text{ ----- } X \times y_2 \times 2^{-2} \quad X \times 0 \text{ 右移 2 位} \\
 1101 \text{ ----- } X \times y_1 \times 2^{-1} \quad X \times 1 \text{ 右移 1 位} \\
 \hline
 0.10001111
 \end{array}$$

被乘数 $X = 0.x_1x_2x_3x_4 = 0.1101$
乘数 $Y = 0.y_1y_2y_3y_4 = 0.1011$

Booth算法的推导和分析

被乘数 X $[X]_{\text{补}} = X_0.X_1X_2 \cdots X_n$ 乘数 Y $[Y]_{\text{补}} = Y_0.Y_1Y_2 \cdots Y_nY_{n+1}$ ($Y_{n+1} = 0$)

$$[X \times Y]_{\text{补}} = [X]_{\text{补}} \times 0.Y_1Y_2 \cdots Y_n - [X]_{\text{补}} \times Y_0$$



$Y_{i+1} - Y_i$	操作
0	部分积 + 0
1	部分积 + $[X]_{\text{补}}$
-1	部分积 - $[X]_{\text{补}}$, 相当于 部分积 + $[-X]_{\text{补}}$

分析: 三种乘法运算实现

- **真题:** 2020年
- 题目:
 - 针对以下三种情况:
 - a) 没有乘法指令;
 - b) 有使用 ALU 和位移器实现的乘法指令;
 - c) 有使用阵列乘法器实现的乘法指令;
 - 乘法在哪种情况下执行的时间最长? 在哪种情况下执行的时间最短? 说明理由
- 解析:

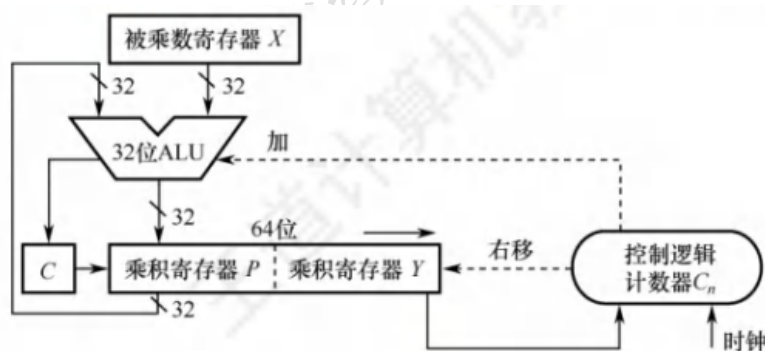
- a) 没有乘法指令 (软件实现)
 - 属于软件实现, 需要反复执行而对于每条指令, 都需要经历很多条指令, “取指、译码、取数、执行、保存结果”这一过程, 因此执行时间比较长;
- b) 有使用 ALU 和位移器实现的乘法指令 (通用硬件实现)
 - 只需用一条乘法指令即可实现乘法运算, 但乘法指令需要多个时钟周期才能完成;
- c) 使用阵列乘法器实现的乘法指令 (专用硬件实现)
 - 只需用一条乘法指令即可实现乘法运算, 且乘法指令可在一个时钟周期内完成;

概念: 乘法运算电路

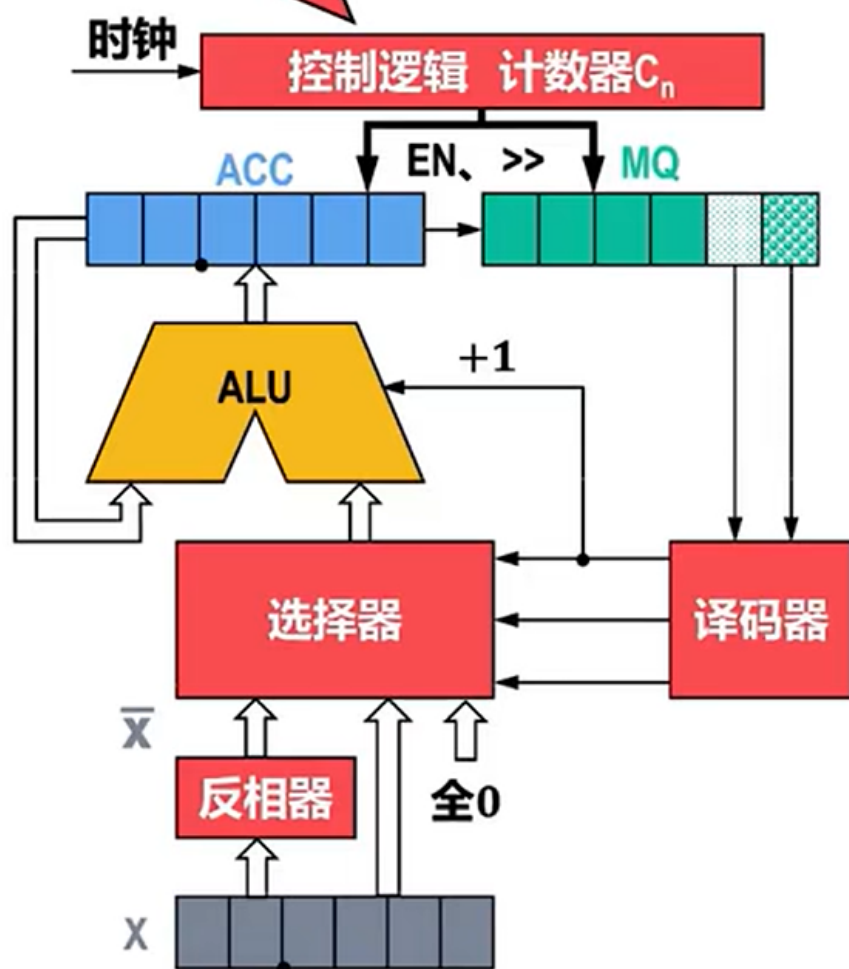
概念:

- C:
 - 部分积和被乘数 X 做无符号数加法时, 可能产生进位, 因此设置一个专门的进位位 C
- P:
 - 乘积寄存器 P 初始置 0
- C:
 - 计数器 C, 初值为 32, 每循环一次减 1
- ALU:
 - ALU 是乘法器的核心部件, 对乘积寄存器 P 和被乘数寄存器 X 的内容做“无符号加法”运算, 结果送回寄存器 P, 进位存放在 C 中;
 - 每次循环都对进位位 C、寄存器 P 和寄存器 Y 实现同步“逻辑右移”, 此时, 进位 C 移入寄存器 P 的最高位, 寄存器 P 的最低位移出;
 - 每次从寄存器 Y 移出的最低位都被送到控制逻辑, 以决定被乘数是否“加”到部分积上;

图示:



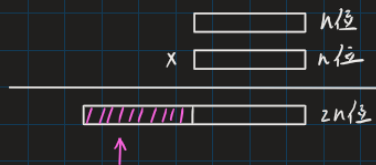
控制“加法、算术右移”操作
计数器 C_n 为“加法、算术右移”操作的次数



总结：乘除法运算溢出判断

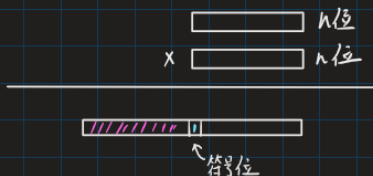
★ 总结：无符号/带符号数乘法的溢出判断

- 无符号时：当结果为 $2n$ 位乘积时，高 n 位为全0，不溢出，否则溢出；

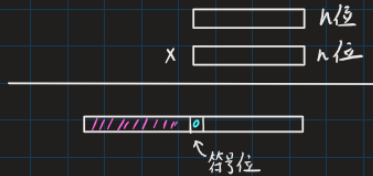


不发生溢出时，此部分进行无符号扩展，全为0；如果发生溢出，才会使得高 n 位不为0

- 有符号时：当结果为 $2n$ 位乘积时，高 $n+1$ 位为全0或全1，不溢出，否则溢出；



不发生溢出时，此部分进行符号扩展，全为1；
如果发生溢出，才会使得高 $n+1$ 位不全为1



不发生溢出时，此部分进行符号扩展，全为0；
如果发生溢出，才会使得高 $n+1$ 位不全为0

举例：当前字长为32位，因此两个32位的二进制数补码相乘，能得到的结果最多为 $32 \times 2 = 64$ 位，前32位为高 n 位，后32位为低 n 位。当结果的高33位不全为0或者不全为1，则此时发生了溢出，OF=1。

★ 总结：

- 乘法运算：乘2 = 左移一位；
- 除法运算：除2 = 右移一位；

★ 溢出处理与判断：

- 逻辑移动：
 - 左移空位：补0
 - 右移空位：补0
 - 左移时，若高位为1，溢出；
- 算数移动：
 - 左移空位：补0
 - 右移空位：补符号位
 - 左移时，若左移前后的符号位不同，溢出

除法运算

概念：除法运算原理

· 概念：

- n 位定点数的除法运算需统一为：
 - 一个 $2n$ 位的数除以一个 n 位的数，得到一个 n 位的商；
- 因此需要对被除数进行扩展；
- 对于定点正小数 (即原码小数)，只需在被除数低位添 n 个0即可对于定点正整数 (即无符号数)，只需在被除数高位添 n 个0即可。做整数除法时，若除数为0，则发生“除数为0”异常，此时需调出操作系统相应的异常处理程序进行处理。

· 过程：

- 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；
- 3. 用中间余数减除数够减则上商为1，不够减则上商为0；
- 4. 如此重复，直到商的位数满足要求为止；

· 图示：

$$\begin{array}{r}
 00111 \\
 0010 \overline{) 00001111} \\
 \underline{0010} \\
 00111 \\
 \underline{0010} \\
 0011 \\
 \underline{0010} \\
 0001
 \end{array}
 \begin{array}{l}
 \text{商} = 0111 = 7 \\
 \text{被除数 } X = 15 = 1111 \\
 \phantom{\text{被除数}} = 0000 \ 1111 \\
 \text{除数 } Y = 2 = 0010 \\
 \text{余数} = 0001 = 1
 \end{array}$$

方法：模二除法运算

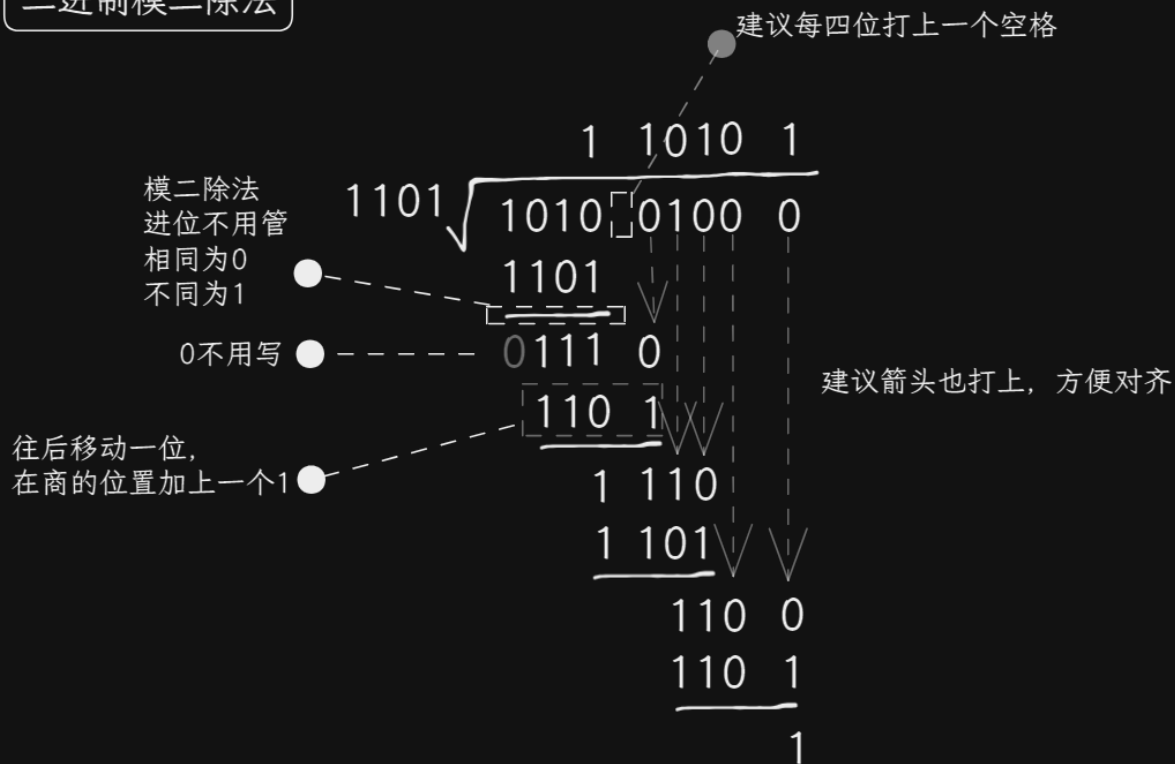
- 规则：
 - 不进位退位；
 - 相同的相减得 0；
 - 不同的相减得 1；

- 图示：

$$\begin{array}{r}
 110 \\
 1110 \overline{) 100101} \\
 \underline{1110} \\
 1110 \\
 \underline{1110} \\
 1
 \end{array}$$

- 举例：CRC校验

二进制模二除法



溢出判断

- 概念：
 - 当用 $2n$ 位除以 n 位的无符号数时，商的位数为 $n+1$
 - 若第一次试商时、试商的结果为 1，则表示结果溢出 $\rightarrow$ 即无法用 n 位表示商；
- 举例：
 - $1111\ 1111 / 1111 = 1\ 0001$

除法运算电路

- 概念：
 -
- 图示：

