位值数制

2024年11月2日 11:00

位值数制（加权数制）
place value system(weighted number system)

数制定义

在次中数制下，数用一串数字组成的序列表示，每一个数字在序列中有不同的位置，不同位置对应着不同的权值。其表示数值的基本逻辑为：

1.加权

这个数的数值等于表示之的数字序列中，每一个数字与其权值的加权和

2.小数点(radix point)

数字的绝对位置由其相对于小数点的距离决定

3.进制

不同位置对应的权值由幂次关系计算，而其基数(base)被称为进制(base)

4.数位

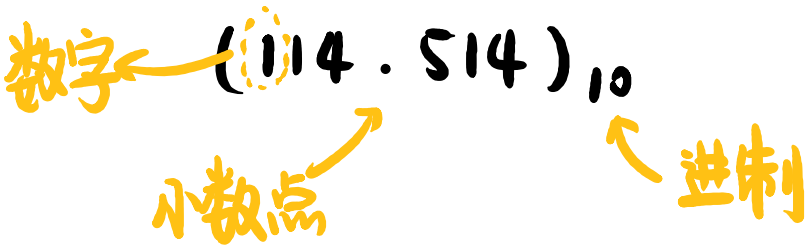
数字位数是进制之下含有的不同种类的数字的数量，一般与进制数量上相等

可以看出，决定加权数制下数值表示的因素有：

小数点位置；数字序列；数位；进制

由于一般而言，数位一般与进制相等，我们可以用进制来表示不同的位置数制

常用的数制有：十进制、二进制、八进制、十六进制；这里“十六”其实是十进制下的$(16)_{10}$
数制的记法：



可见，数字的表示可以看作由两个部分组成：数字序列 和 进制-数位

数制的转换

对于同一个数值而言，其可以采取不同的表示方法，对应不同的进制-数位组合

我们一般称某一种表示方法“采取了某种进制”，而不同的表示方式之间的转换也就是所谓的数位转换

数位转换方法的核心在于将加权和的形式写出，

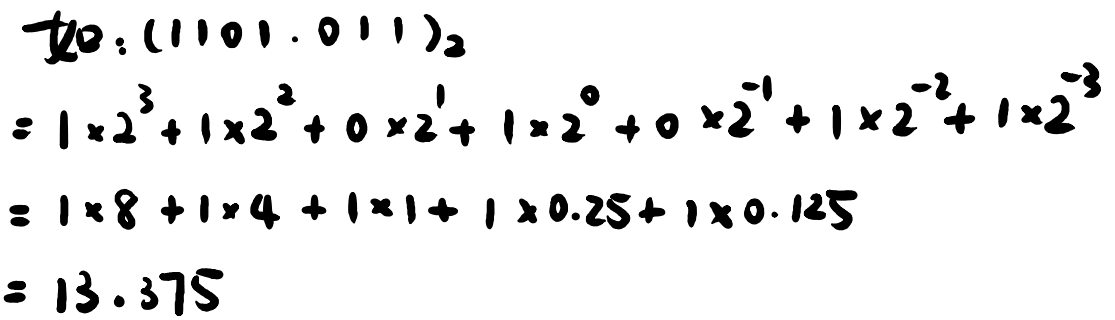
再利用乘/除与余数得知相应的数字序列；而且十进制作为一个必须的中间状态出现（除了一些特殊的互为整数幂次的进制，如2^3 = 8），其原因是我们只熟悉十进制下的幂次和乘法运算（九九乘法表）

十进制在数学上不是必须的中间步，但对于我们而言它是必须的一部

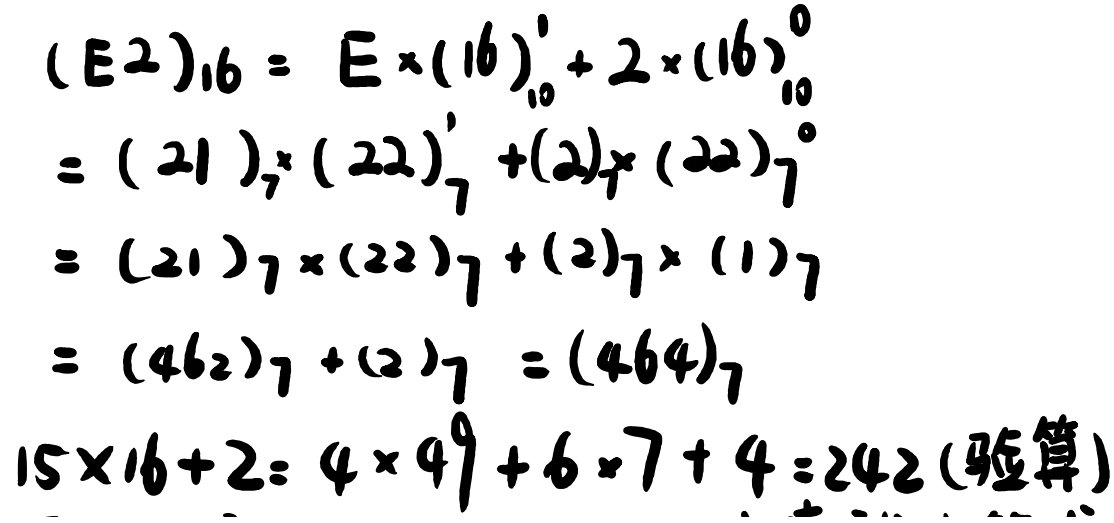
即：x进制->十进制->y进制

1.x进制转化为十进制

由于在十进制下，我们对幂次的计算非常熟悉，因此可以直接写出其在相乘后所表示的十进制数



他的实际逻辑是



$15\times16+2=4\times9^2+6\times7+4=242$（验算）

但 $(22)_7^3\cdot(65)_7\times(46)_7$ 这种值就比较难处理3，故我们一般仍采用十进制作为中间量

但要在其它进制下硬算也并非不可，其乘法的进位性质可以类比十进制，如：

$$(65)_7\times(46)_7=(6\times7^1+5)_{10}\times(4\times7^1+6)_{10}$$

$$=(4\times6)_{10}\times(7^2)_{10}+(5\times4+6\times6)_{10}\times(7^1)_{10}+(5\times6)_{10}\times(7^0)_{10}$$

$$=(33)_7\times(10)_7^2+(110)_7\times(10)_7^1+(42)_7\times(10)_7^0$$

$$=(3300)_7+(1100)_7+(42)_7=(4442)_7$$

2.十进制转化为y进制

我们将十进制下的某个数假设为以下形式：

小数点前：$(a_1)_{10}\times(n)_{10}^k+(a_2)_{10}\times(n)_{10}^{k-1}+\cdots+(a_{k+1})_{10}\times(n)_{10}^0$

小数点后：$(b_1)_{10}\times(n)_{10}^{-1}+(b_2)_{10}\times(n)_{10}^{-2}+\cdots+(b_{k'})_{10}\times(n)_{10}^{-k'}$

随后便可以对于小数点前、小数点后的部分分别运用除法、乘法进行处理（当然，也是因为我们熟悉十进制下的除法）；

对于小数点前，我们用带余数的除法，将表达式

除以（十进制下）其进制可以得到：

$$n \overline{)(a_1)\times(n)_{10}^{k}+(a_2)_{10}\times(n)_{10}^{k-1}+\cdots+(a_{k+1})\times(n)_{10}^{0}}$$

$$(a_1)\times(n)_{10}^{k-1}+(a_2)_{10}\times(n)_{10}^{k-2}+\cdots+(a_k)\times(n)_{10}^{0} \quad \cdots a_{k+1}$$

余数刚好为最右侧位上的数字

连续进行这样的短除法，直至除数为0.

$$n \overline{)(a_1)\times(n)_{10}^{k}+(a_2)_{10}\times(n)_{10}^{k-1}+\cdots+(a_{k+1})\times(n)_{10}^{0}}$$

$$n \overline{)\cdots\cdots} \quad \cdots a_{k+1}$$

$$n \overline{)\cdots\cdots} \quad \cdots a_k$$

$$n \overline{)\cdots\cdots} \quad \cdots a_{k-1}$$

$$\vdots$$

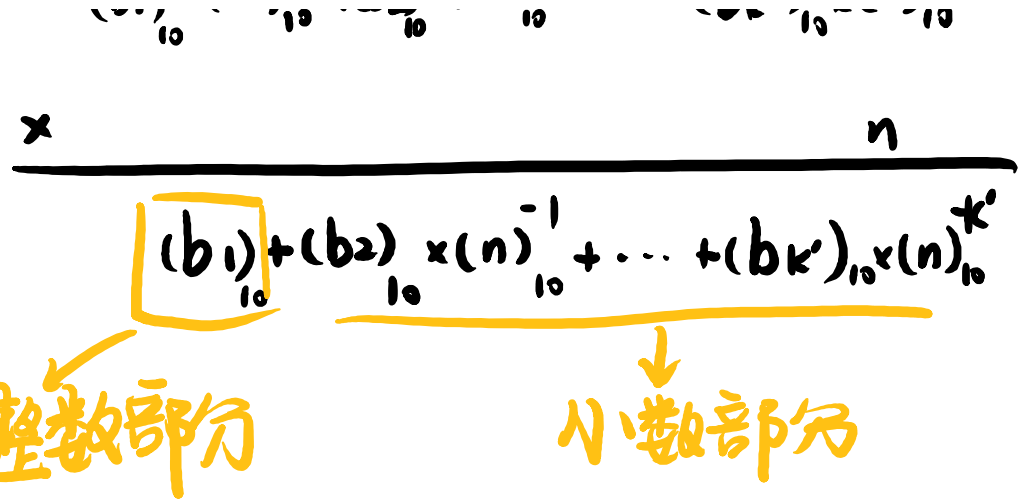
$$n \overline{)a_1} \quad \cdots a_1$$

$$0$$

然后将 $a_1 - a_{k+1}$ 转化为 n 进制下的数字后，从下往上连接，即可得到小数点前部分的 n 进制表示

对于小数点后部分的处理，可以采用相似的思想，利用乘法进行处理。

$$(b_1)_{10}\times(n)_{10}^{-1}+(b_2)_{10}\times(n)_{10}^{-2}+\cdots+(b_{k'})_{10}\times(n)_{10}^{-k'}$$



故只需不断取出乘后的整数部分，并转化为n进制下的数字，再从上到下连接，在序列前加上"0"(占位)与小数点即可

不同进制下的四则运算

加减法

相应数位上的数字按照数字的运算规则进行相加，存在进位和退位的情况

加法的逆元被定义为负数，相加得0的性质也是其最大的性质。一般而言负数用符号"-"联合其自身逆元对应的正数进行表示，如$(-32)_{10}$

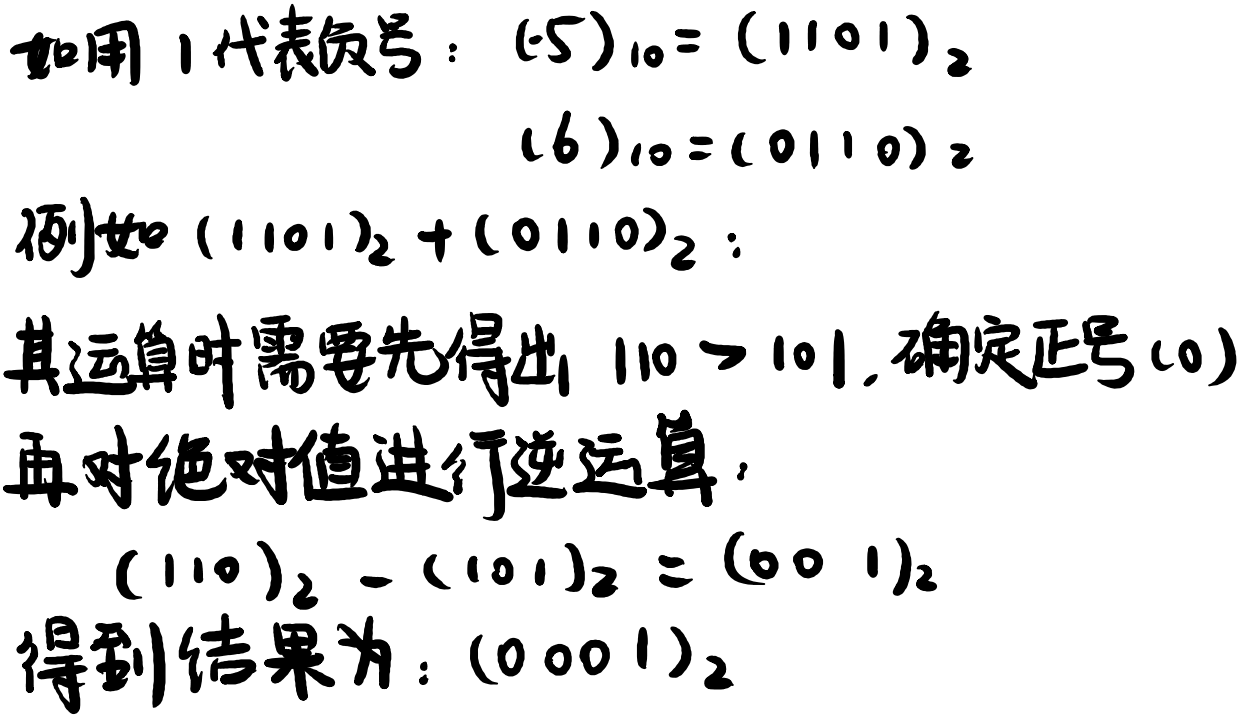
乘除法

将数位对应的位置值数相加，作为数字相乘后所对应的新的权重

位宽

尤其对于二进制数：储存之的媒介能够容纳的数字序列长度有限，这一长度成为位宽

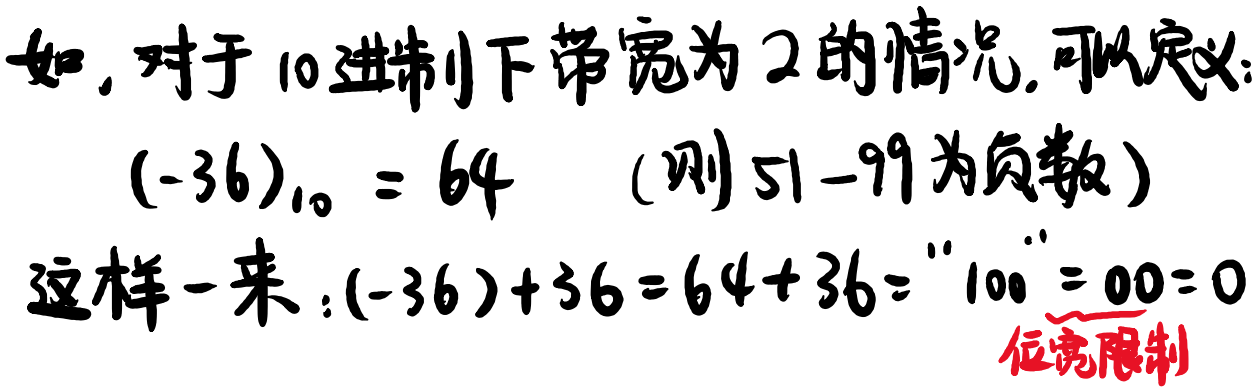
位宽的存在使得二进制中的负数表示可以采取多种形式。一种形式是采用一位数字作为**符号位**，其缺点在于，相应的运算需要 基于绝对值相对大小的、逻辑上的符号判断



为了简化运算，人们对于有限位宽的情况，采取了补码来表征符号。

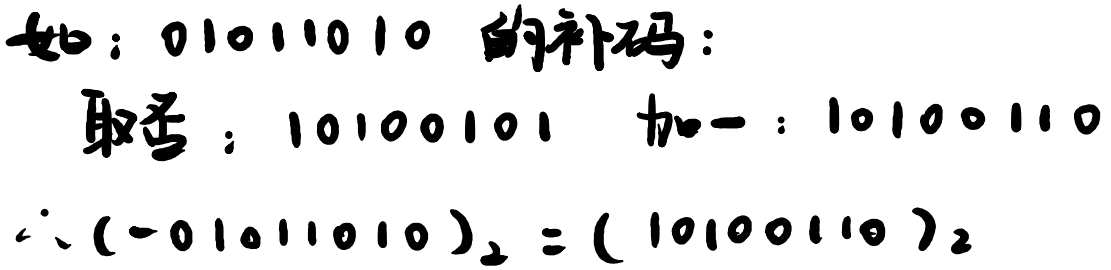
补码的原理是利用1000...0（位宽个0）在有限带宽之

下的非法性，将其等价为0后，以其作为加法的运算逆元从而创造出的一种表示方式。一般而言，其把能够表示的所有数字对半分为正数和负数，通常选择较大的一半表示负数，较小的一半表示正数



而对于2进制，在这种定义方法之下，将负数转化成等价的正数表示的方法更加简单：只需要将所有数位上的数字取否，然后再加1即可（易知其与逆元相加等于1000...00)

二进制的补码英文中称作(2's complement)



为了方便称呼，这种全部取反的操作得到的数字被称为“反码”，英文中称为(1's complement)

需要注意的是，补码本身存在着一定的缺陷。对于偶

数倍进制，存在着补码不存在的情况。如上面对于10进制2位宽的情况，50的补码是其自身，导致其实际上的含义难以区分。
在二进制中，这种情况为 "1000"(位宽4时) 这样的数，其补码为自身，但按照惯例（首位为1）将其看作负数(-8)

布尔代数与开关代数

2024年11月2日 17:03

布尔代数

代数结构定义

布尔代数是一种代数结构，其上有三种运算：

加法(+)、乘法(.)、取否(')

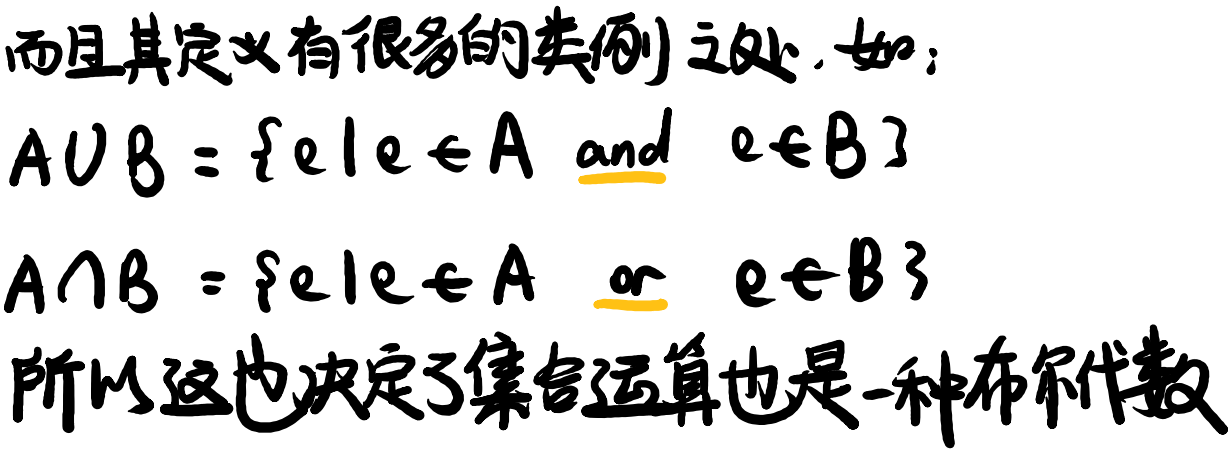
分别满足以下的运算法则：

1. **Closure**: For any $a, b \in B$, $a + b \in B$ and $a \cdot b \in B$.
2. **Commutative Laws**:
   - $a + b = b + a$
   - $a \cdot b = b \cdot a$
3. **Associative Laws**:
   - $a + (b + c) = (a + b) + c$
   - $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
4. **Distributive Laws**:
   - $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
   - $a + (b \cdot c) = (a + b) \cdot (a + c)$
5. **Identity Elements**:
   - There exists an element $0 \in B$ such that for every $a \in B$, $a + 0 = a$.
   - There exists an element $1 \in B$ such that for every $a \in B$, $a \cdot 1 = a$.
6. **Complement Laws**:
   - For each $a \in B$, there exists an element $a' \in B$ (called the complement of a) such that:
     - $a + a' = 1$
     - $a \cdot a' = 0$
7. **Idempotent Laws**:
   - $a + a = a$
   - $a \cdot a = a$
8. **Absorption Laws**:
   - $a + (a \cdot b) = a$
   - $a \cdot (a + b) = a$

第七和第八条实际上是定律，不是定义

需要注意的是，这里对于元素所属于的集合并没被有限制为Binary({0,1})的情况，存在|B|>=3的情况；所以“布尔代数”与后面提到的，也是在数字电路中用到的，“开关代数”，还是存在区别的。

理解布尔代数的一种更好的方法是将其与集合论对比，加法、乘法、取否恰好对应集合论里的并集(∪)、交集（∩）、补集（A^c），原因是其具有相同的运算律



运算性质

除了定义中规定的运算律，布尔代数还存在两个重要的定律可以帮助运算和化简：

DeMorgan's Law(德摩根定律)

(a+b)'=a'b'

(ab)'=a'+b'

Consensus Theorem(共识定理)

ab+ac+b'c=ab+bc

(a+b)(a+c)(b'+c)=(a+b)(b'+c)

Proof

AB+AC+B′C=AB+AC·1+B′C=AB+AC(B+B′)+B′C=AB+ABC+AB′C+B′C=AB(1+C)+(A+1)B′C=AB+B′C

(A+B)(A+C)(B'+C)=(A+B)(A+C+BB')(B'+C)=(A+B)(A+C(B+B')+BB')(B'+C)=(A+B)(A+B'C+B)(B'+C)

另外要指出的是，由于加法和乘法在定义上除了单位元(0,1)是不同的，其它的所有定义完全一致，故对于一个布尔代数中的结论，将其所有乘(.)和加(+)互换，将所有1和0互换，可以得到一个同样正确的结论（与原来的结论不同）。这一性质被称为“Duality(对偶性)”

开关代数(Switching algebra)

定义

定义在S={0,1}（布尔域）的布尔代数

开关函数(Switching function)

即定义在开关代数下的函数，为从布尔域到布尔域的映射；其存在两种表示形式，分别为解析式和真值表

逻辑表达式

即布尔代数表达式形式，如f(A,B,C)=AB+B'C

真值表(Truth table)

即开关函数的

表(look up table)形式，如：

A	B	B'	$f(A,B) = AB'$
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

香农展开定理

Shannon's expansion

f(x,y,z)=xf(1,y,z)+x'f(0,y,z)

f(x,y,z)=(x+f(0,y,z))(x'+f(1,y,z))

需要注意的是，香农展开定理对布尔代数结构中的函数普遍成立，并非只对交换代数下的函数成立；这里给出的是开关代数中的香农展开定理

(一个例子是，概率这个函数也符合香农展开定理，其展开出的函数恰好就是条件概率)

$$f(x,y,z) = a\,f(x=a,y,z) + \bar{a}\,f(x=\bar{a},y,z)$$

$$\Downarrow$$

$$\mathbb{P}(X,Y) = \mathbb{P}(x=a)\cdot\mathbb{P}(X,Y \mid x=a) + \mathbb{P}(x=\bar{a})\cdot\mathbb{P}(X,Y \mid x=\bar{a})$$

化简（卡诺图）

Minterms(Maxterms)

对于一个乘积而言（如ABC），只有当其含有的所有变量均为1式，此式子才为1，否则为0；现象上，其值与其变量中的最小值保持一致，故这一项也被称为最小项（Minterm)

相似的，一个和（如A+B+C）的值与变量中的最大值保持以制，故这一项被称为最大值(Maxterm)

由香农展开公式可知，所有的开关函数都可以表示为一系列最大项的和或者最小项的积。这种表示方式被称作正则表达式，分别为与或式(Sum of Products)和或与式(Product of Sums)

Sum of Products: f(A,B,C)=A'BC+AB'C
Product of Sums: f(A,B,C)=(A+B'+C)(A+B'+C')

可以将这些变量一次链接看作一个“数字序列”，规定变量本身为1，取否则为0，则可以用相应bit的二进制数来代表所有可能组成的积/和，其记法为：

$f(A,B,C)=A'BC+AB'C=m_3+m_5=\sum m(3,5)$

$f(A,B,C)=(A+B'+C)(A+B'+C')=M_5+M_4=\prod M(4,5)$

$A'BC\Rightarrow(011)_2\Rightarrow 3$　　$AB'C\Rightarrow(101)_2\Rightarrow 5$

$A+B'+C\Rightarrow(101)_2\Rightarrow 5$　　$A+B'+C'\Rightarrow(100)_2\Rightarrow 4$

其中，m意为minterm, M意为Maxterm

布尔表达式化简与卡诺图

对于一个布尔表达式，“化简”意味着将其表达式变换至项最少、每一项涉及尽量少变量的等价形式。

如：f(A,B,C)=ABC+ABC'

此处，根据(C+C')=1，可知：f(A,B,C)=AB(C+C')=AB

变换后的表达式只有一项，且只涉及了2个变量，所以是一个更“简单”的形式。可知其为最简

利用布尔表达式的运算性质进行化简主要运用：

(1)a+a'=1　a*a'=0

一般结合添项使用：如AB=AB(C+C'); AB=AB+C'C

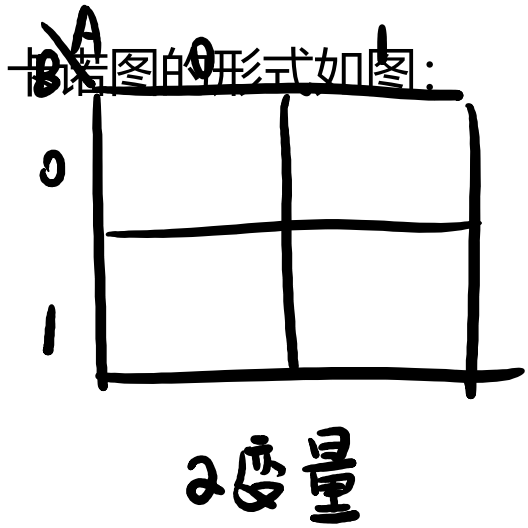
(2)Consensus Theorem & DeMorgan's law

(3)Idempotent Laws and Absorption Laws

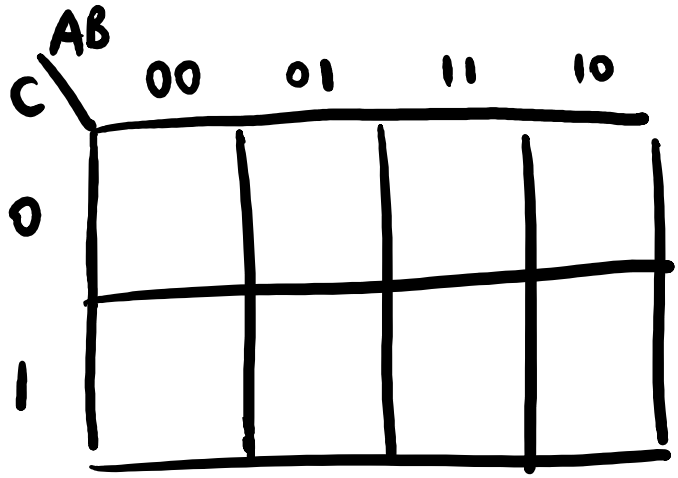
卡诺图(Kaunaugh's Map, K-Map)

实质上为利用Idempotent Laws 和 Complement Laws 对布尔表达式进行化简

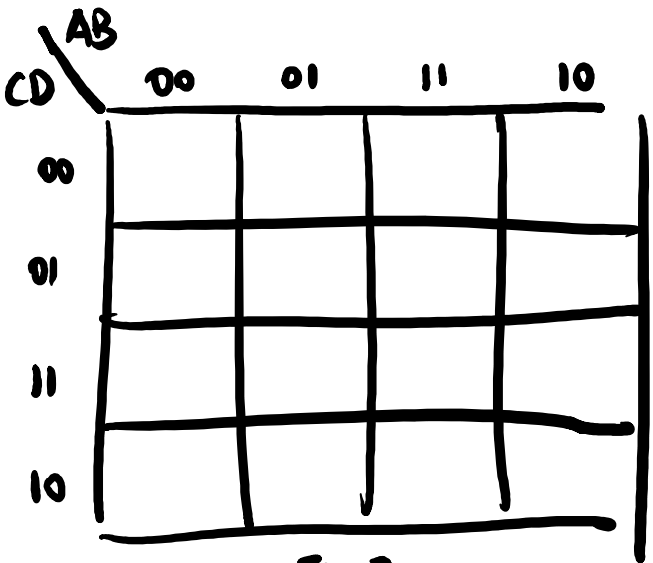
卡诺图的形式如图：



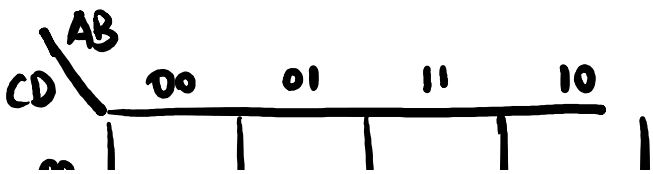
2变量

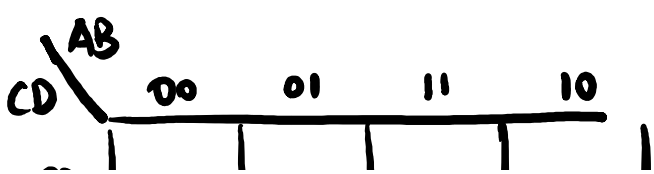


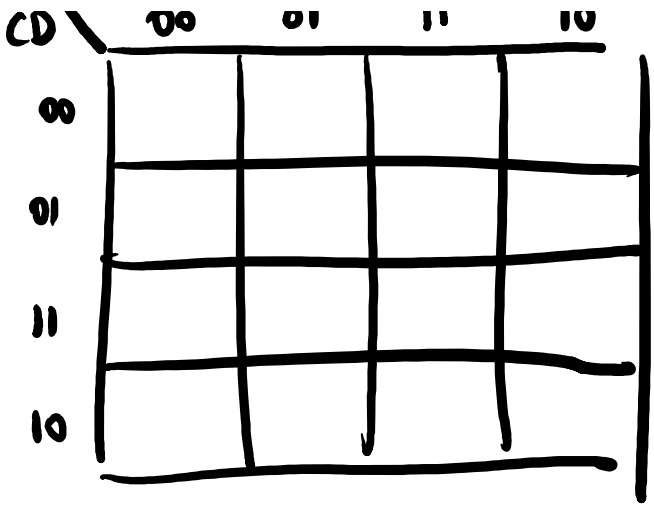
3变量



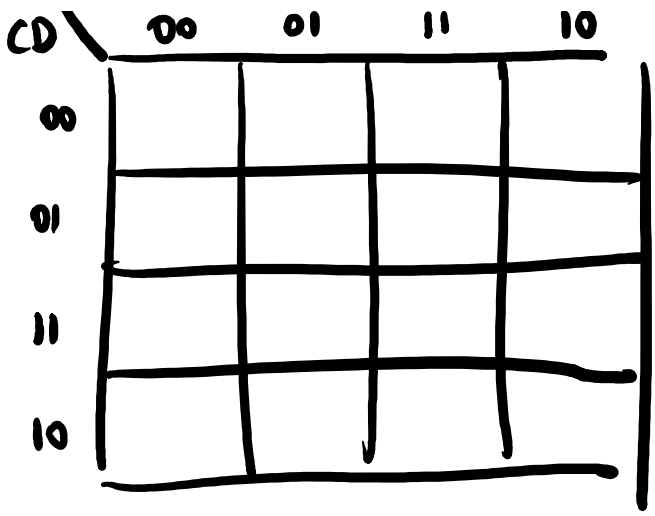
4变量





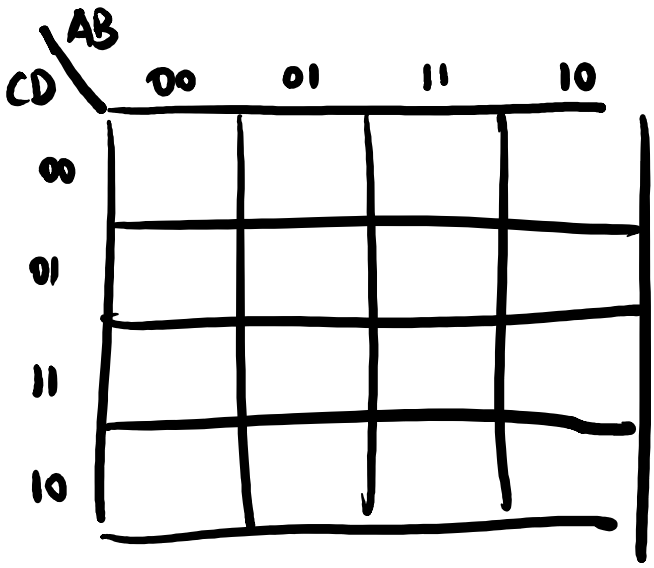


E=0

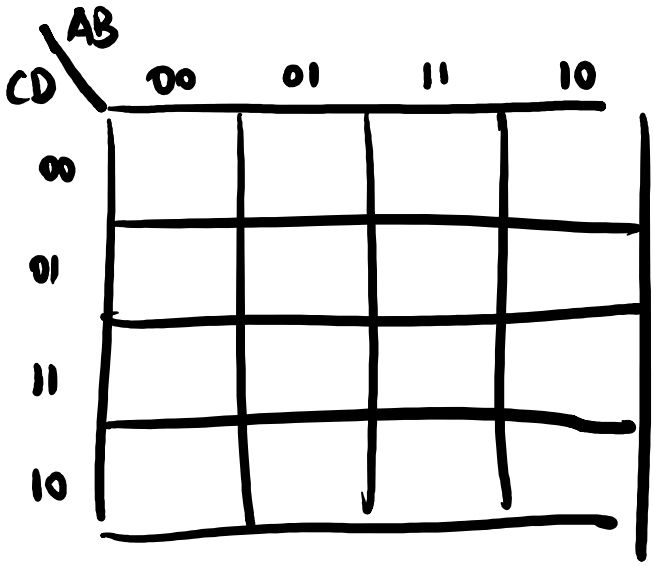


E=1

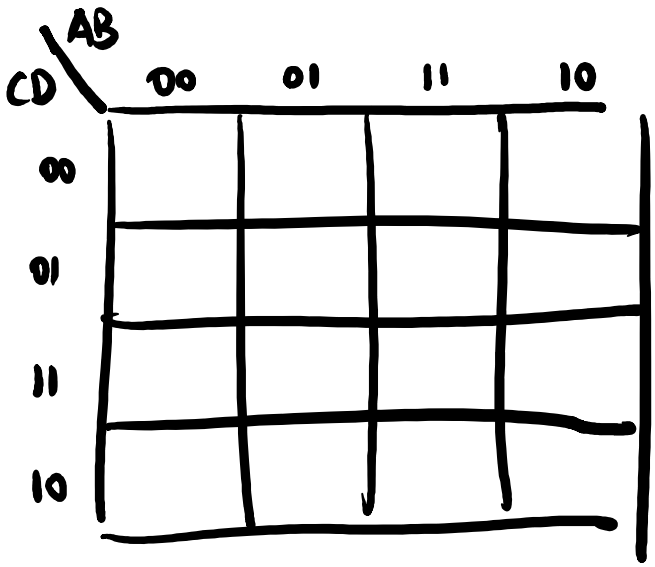
5变量



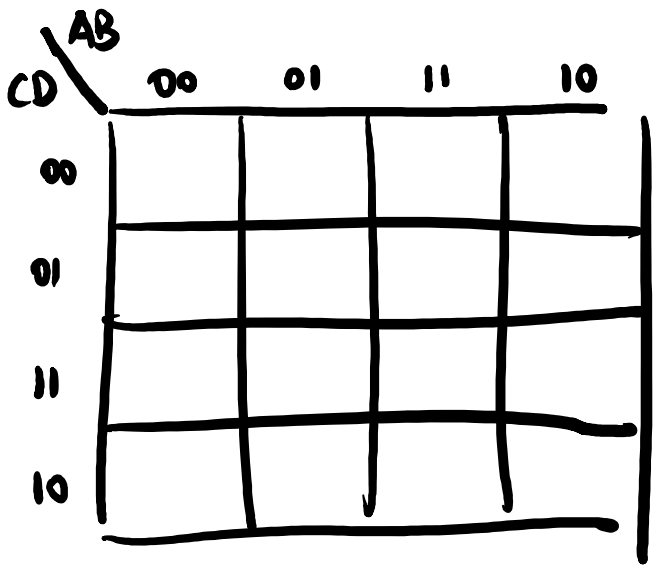
E=0

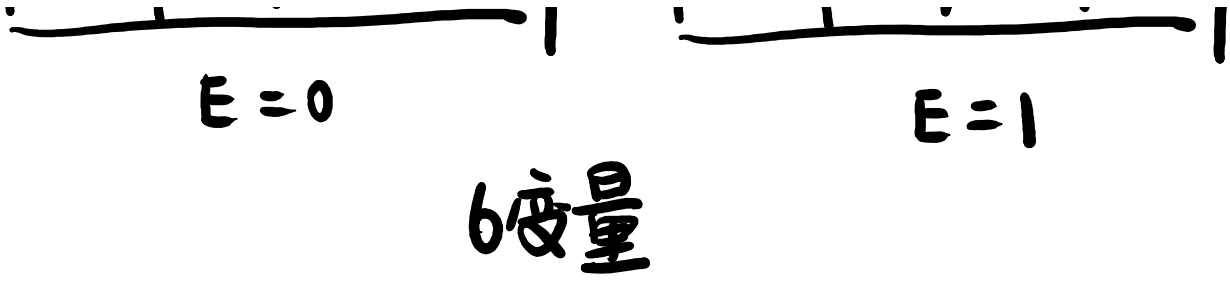


E=1



E=0

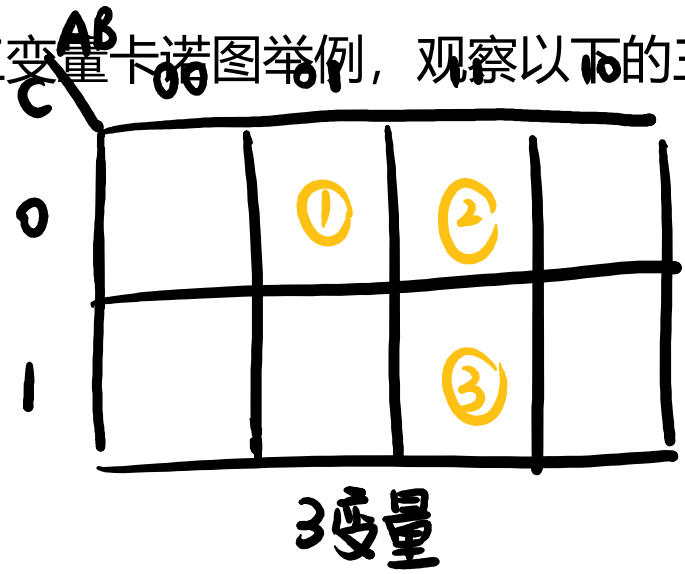




6变量

其设计核心在于把“位置相邻”构造成“代数互斥”(algebraic disjoint)，即任意相邻的两个组合中恰好有一对互为取否

拿下面的三变量卡诺图举例，观察以下的三个位置：



3变量

其代表的组合分别为：

A'BC'(A'+B+C'),ABC'(A+B+C'),ABC(A+B+C)

其中1和2在图上的位置是相连的，而比较其表达式，其区别在于A一项：A和A'，刚好是互斥的

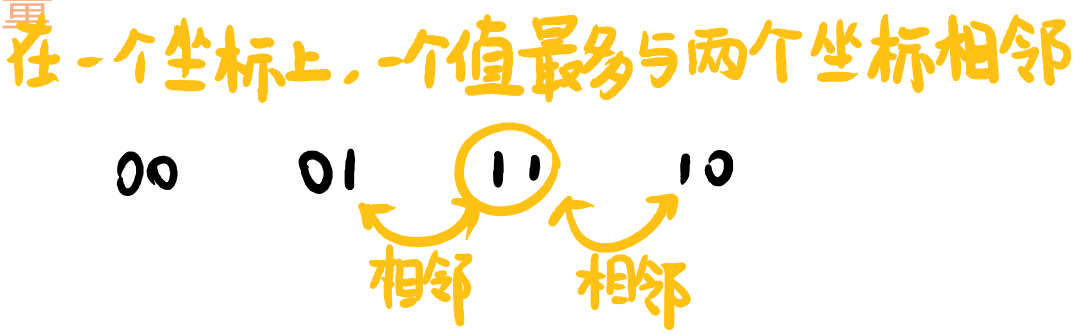
根据Complement Laws，互斥的情况是可以化简的：

A'BC'+ABC'=(A+A')BC'=BC'

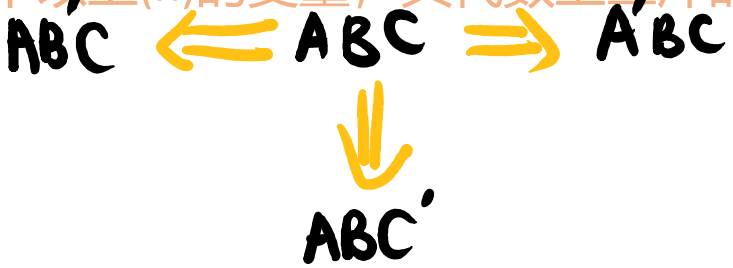
$(A'+B+C')(A+B+C')=A'A+(B+C')^2+(B+C')(A+A')=B+C'$

如果有多对互斥关系在一起，则又可以进行进一步的化简；而所有函数又可以根据卡诺图中完全列出的输入组合-输出的关系写成SOP或POS的形式，故可以据此得到函数的最简形式

需要注意的是，5个及以上的变量需要用多张平面图表示。这是因为若要保持卡诺图的互斥特性，一个坐标上最多表示2个变量



但对于三个以上(n)的变量，其代数上互斥的组合有n种：



同时注意，根据互斥的原则，左右两端、上下两端在位置上也是相接的。

每一种相接的组合叫做一个蕴涵项(Implicant)

如果一个蕴涵项没有被其它蕴涵项完全覆盖，那么其叫做一个

主要蕴涵项（素蕴涵）(Prime Implicant, 或者 Prime)
如果一个主要蕴涵项含有的某个/些组合，没有被任何其它主蕴涵项包括，则其称为基本素蕴涵(Essential Prime Implicant, 或者 Essential Prime)

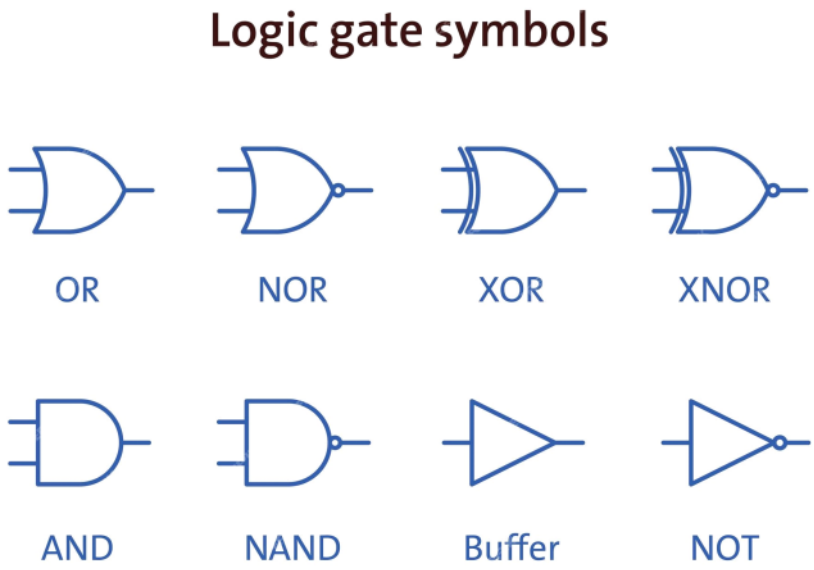
一个函数通过卡诺图能够化简出的形式即为其所有Essential Prime的和/积；化简的结果可能不止一种（有多种选取Essential Prime的方式）

补充

Variable: 函数变量
Operator：算子，开关函数中即+/./'
Literal: 字母，包括变量以及取否(如A,B,A',etc)
Expression：表达式

电路原理图

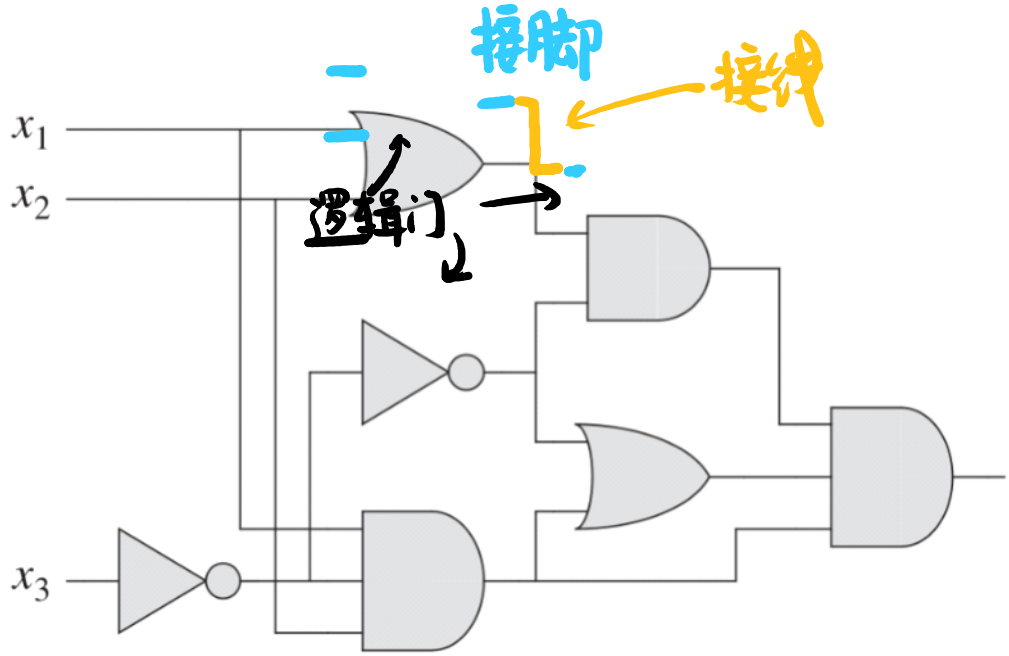
通过图的形式表现回路构造，也可以用来体现逻辑表达式
其基本组成部分为：
逻辑门(Logic Gates) 严格来说，逻辑门也由晶体管构成



接脚(pins)

接线(signal nets)

时钟信号(Clock Signal)

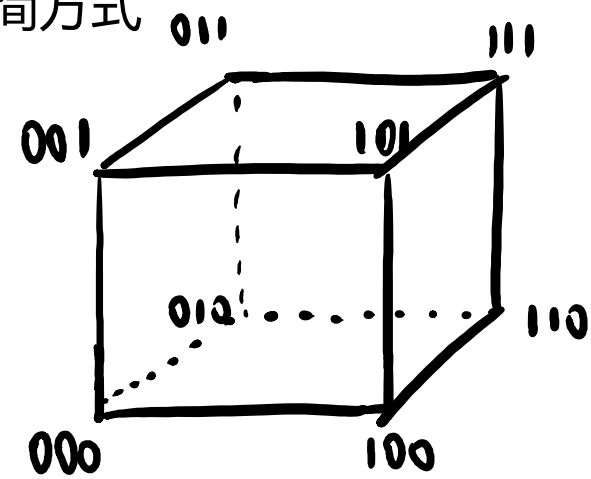


补充

严格来说，卡诺图并不是唯一的可以用来直观化简开关函数表

达式的方法；可以通过更高维的“位置相邻-代数邻近”构造出可视化的化简方式

如三维中：



Universal Set

布尔代数可以只用NAND 或者 NOR 写出

Functional Completeness of NAND and NOR:

- **Using NAND to Create NOT, AND, OR:**
  - **NOT:** $A' = A \uparrow A$
  - **AND:** $AB = (A \uparrow B)' = (A \uparrow B) \uparrow (A \uparrow B)$
  - **OR:** $A + B = (A' \uparrow B')$
- **Using NOR to Create NOT, OR, AND:**
  - **NOT:** $A' = A \downarrow A$
  - **OR:** $A + B = (A \downarrow B)' = (A \downarrow B) \downarrow (A \downarrow B)$
  - **AND:** $AB = (A' \downarrow B')$

思路是先写出’ 再用摩根定律写出AND和OR

这一性质应该叫functional completeness, 具有这一性质的算符集合被称为functionally complete set，如{NOT, AND}, {NAND}, {NOR}

但是在课件中这一点被称为Universal set, 一般而言这个名词被用于指代“全集”（论域），并非

functionally complete set 的概念

XOR和XNOR虽然不具有这样的完备性，但是由于其形式在很多应用中出现（如加法器），所以也单独作为一个逻辑门出现

Expressing XOR and XNOR Using AND, OR, NOT:

- **XOR:**
  - $A \oplus B = (A + B)(A' + B')' = AB' + A'B$
- **XNOR:**
  - $A \odot B = AB + A'B'$
  - $A \odot B = (A \oplus B)'$

补充：对于一个卡诺图而言，其能够给出的相对最简的表达是可能不止一种；基本质蕴项的意义在于找出那些在这些表达式之中都会出现的项，从而给整个回路的表达式一个“lower bound”；但是是否是“最简”还需要进一步对于卡诺图中给出的这些结果进行分析

组合逻辑电路

2024年11月4日 9:55

组合逻辑（combinational logic）

组合逻辑与时序逻辑相对，是数字电路中所有不含时间的回路以及其表示的回路的通称
“组合”意味着“输出”是且仅为输入的组合；与之相对的，在时序逻辑中，输出还与回路状态有关

由于代数上的逻辑部分可以用开关代数完全描述，这里不再赘述；本章重点介绍相应的回路

回路的层级

工程上的问题大多数有一个从简到繁的过程，这一过程的实现叫做创造，其对应的逆过程则为分析。一个工程系统可以拆解成若干的功能模块，每一个功能模块又可以拆解为一系列小部件，每一个小部件又有不同的元器件组成，每一个元器件又由不同的原材料，经过不同的工艺组合而成

数字电路也可以看作这样一个功能系统，其内部也存在着相应的层级。
(Device level)最基础的部分可以认为是晶体管、引脚、接线；晶体管是实现二值电路系统的基础，而有了引脚和接线一个电路系统才能形成回路、正常运转。

(Gate level)晶体管的组合可以构成逻辑门，从而开始与布尔代数进行对应，每一个逻辑门以二值性的电平为0和1，通过控制电压来实现相应的逻辑运算。如布尔代数中的加法(+)、乘法(.)便可以用AND Gate, OR Gate进行实现

(Module level)在数字电路的层级划分中，逻辑门作为最基本的构建单元，通过组合可以实现更为复杂的功能。逻辑门的组合可以进一步构成各种逻辑模块，这些模块是实现特定功能的核心组件。例如，加法器、多路复用器/解复用器、编码器/解码器等。

> 首先，加法器（Adder）是一种基础的算术逻辑单元，用于执行二进制数的加法运算。最简单的加法器是半加器（Half Adder），它能够实现两个输入位的相加。而全加器（Full Adder）则是在半加器的基础上增加了对进位的处理能力，因此可以实现多位二进制数的逐位相加。多个全加器可以串联组成一个更大的加法模块，用于更复杂的运算。
>
> 其次，多路复用器（Multiplexer, 简称Mux）和解复用器（DeMultiplexer, 简称DeMux）则负责在多个信号之间进行选择和分配。多路复用器的主要功能是将多个输入信号通过选择信号选择输出到一个输出端，而解复用器的功能则正好相反，它将一个输入信号通过选择信号分配到多个输出端。这些模块在通信和数据传输中起到了重要的作用。
>
> 编码器（Encoder）和解码器（Decoder）是用于数据编码和译码的模块。编码器将多路输入转化为较少的输出位，从而实现数据压缩，而解码器则是对编码数据进行还原，恢复其原始状态。编码器和解码器广泛用于数字通信中，以便实现数据的传输和恢复。

(Microarchitecture level)这些逻辑模块可以构成更为复杂的电路系统，这些系统既包括组合逻辑模块，也包括时序逻辑模块，例如计数器、寄存器、ALU（算术逻辑单元）等。

(System level)这些模块通过相互连接和交互，构建成功能完整的数字电路系统，例如计算机的中央处理器（CPU）或者控制系统中的嵌入式处理器。

每个层级的构建从基础的晶体管到逻辑门，再到组合逻辑和时序逻辑模块，最终组成系统。这种自下而上的结构使得数字电路的设计具有极强的模块化特点。模块化的设计使得各个功能单元的开发、测试和改进更加

便捷，工程师们可以专注于每一个单元模块的性能优化，最终将这些模块组合成一个高效、可靠的系统。这种设计思路从底层元件一直延续到系统级别，为现代工程提供了强大的创造和分析能力。

当然，网上或者教科书上也给出了不同的层级系统的分类，本文按照此处的理解进行组织
如数字电路设计的抽象层次

抽象层次	行为域	结构域	物理域
系统层	设计的功能和指标	处理器、存储器等	
算法层	算法	硬件模块框图	
寄存器传输层	传输方程、状态机等	寄存器、ALU等	宏单元
逻辑层	布尔逻辑函数	门级电路	标准单元库
电路层	网络方程	晶体管级电路	工艺层
版图层			版图

系统层：是整个设计过程的第一步，也是最关键的一步。主要进行电路功能的定义，各种电学参数（如工作频率、功耗、工作温度等）的确定。这一层的设计好坏直接决定了整个集成电路性能的好坏、价格的高低、市场的占有率，更决定了后续设计阶段的难易程度及效率。

算法层：主要进行算法设计以及描述。首先根据系统的功能要求，制定可以实现此功能的不同算法，分析和比较这些算法的优缺点，选定一种最适合的；其次，根据所选定的算法对整个系统进行功能划分及各个功能模块之间的数据流与控制流连接；此外，时钟方案等对系统性能起关键作用的部分也在算法层实现。由此可见，算法层的设计对于整个系统的性能起着至关重要的作用。

寄存器传输层：也即 RTL，主要将算法层用代码的形式进行抽象描述。在进行 RTL 层描述时，必须严格按照设计所使用的综合工具要求的规范进行，并尽可能地考虑延时、面积、测试等问题。

逻辑层：将系统再进一步细化，转换为用普通的逻辑门来实现。对于当前的设计方法来说，这一层次主要借助于综合工具来完成。

电路层：将逻辑层中的门电路用具体的晶体管、电容、电阻等基本电子元器件来表示，并将之间的互连关系呈现出来。

版图层：将各种电子元件用物理的方式呈现出来，根据所选工艺将这些元件转换成不同的几何图形，并相互连接。版图层的实现方式是系统最终的呈现方式，也是整个设计中最低的层次，并且仅仅是结构描述和物理特性表征。

行为域 主要关注系统的功能实现，对系统的输入/输出关系进行描述。
结构域 关注系统中每一抽象层次的实现方式，包含具体的逻辑和电路结构。输出关系进行描述。
物理域 则更加关注集成电路最终的呈现方式，以物理特性表征。

组合逻辑电路模块

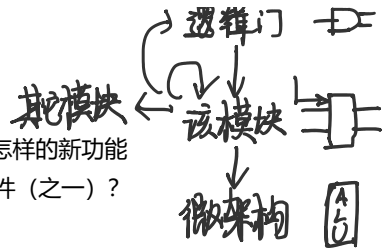
上面已经提到，逻辑门通过组合可以构成各种模块，进而实现一些基础的功能；而实现组合逻辑的基本模块中，有以下最基本的几种模块：

1. Adder(Half-Adder; Full-Adder)
2. Comparators
3. Encoder/Decoder
4. Multiplexer(MUX)/Demultiplexer(DeMUX)
5. Code Converter

接下来，本文会从以下的几个问题，去探究各种电路模块的性质：

1. 该模块如何用逻辑门实现?
2. 它本身的功能是什么?

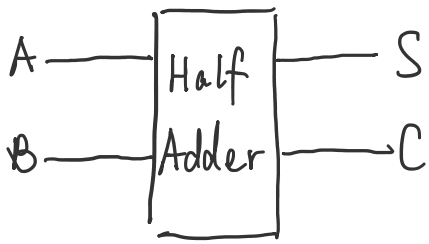
1. 该模块如何用逻辑门实现?
2. 它本身的功能是什么?
3. 为什么这一功能让它成为一个被广泛使用的基本模块?
4. 它与自身/其他模块/逻辑门进行简单的组合后能够产生怎样的新功能
5. 它最重要的应用有哪些?它是哪些微架构层次的核心器件（之一）?



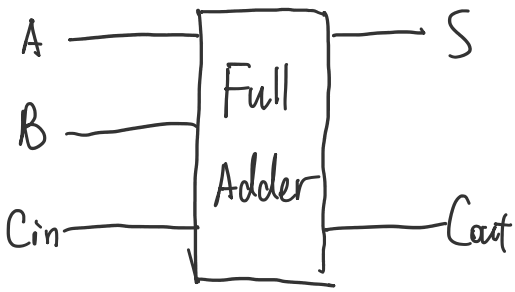
Adder(Half-Adder, Full-Adder)

顾名思义，加法器是一种用于执行二进制加法的基本数字模块。半加器有两个输入：A 和 B，输出包括和 (Sum) 和进位 (Carry)。全加器有三个输入：A、B 和前一位的进位 (Cin)，输出同样为和 (Sum) 和进位 (Cout)。加法器主要用于算术运算，是数字系统中执行加法操作的基础模块。

Input		Output	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



A	B	Cin	Sum	Cout
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

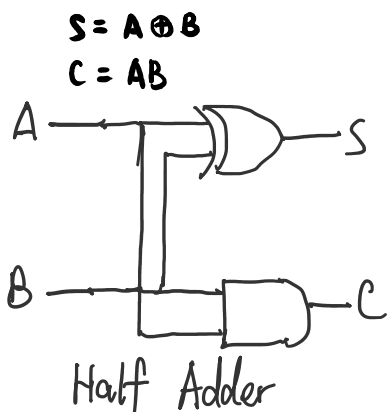


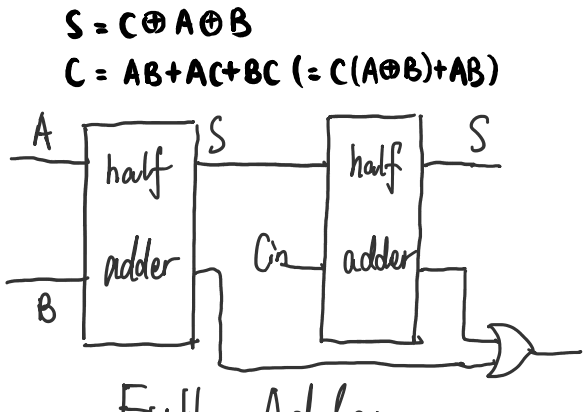
$Cout = AB + BC + AC$

1. 该模块如何用逻辑门实现?

半加器 (Half Adder): 半加器使用两个逻辑门实现其功能。XOR 门：计算两个输入位的和 (Sum)。
AND 门：计算进位 (Carry)。
输入的两个位通过 XOR 门得到“和”输出，而通过 AND 门得到“进位”输出。

全加器 (Full Adder): 全加器可以通过两个半加器和一个 OR 门实现。第一个半加器的 XOR 门用于处理两个输入位的和，AND 门用于生成第一个进位。
第二个半加器则将第一个半加器的和输出和进位输入再进行 XOR 运算，得到最终的和输出，同时通过 AND 门生成第二个进位。
两个进位通过 OR 门相加得到最终的进位输出。
这种逻辑门的组合使得全加器可以对三个位进行计算，从而实现多位二进制数的逐位加法。





2. 它本身的功能是什么?

半加器的功能是将两个单个位相加，产生一个和输出和一个进位输出。

全加器的功能是在半加器的基础上，增加了对前一位进位的处理能力。因此，全加器可以进行**两个多位二进制数的逐位加法运算。**

3. 为什么这一功能让它成为一个被广泛使用的基本模块?

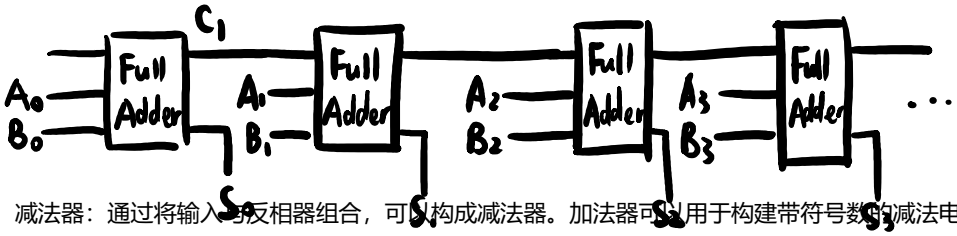
加法是计算机中**最基础的算术运算之一**，广泛应用于算术运算、数据处理、计时和控制等领域。

全加器的设计支持对多位二进制数进行加法计算，通过**串联多个全加器**可以实现**任意位数的二进制加法。**

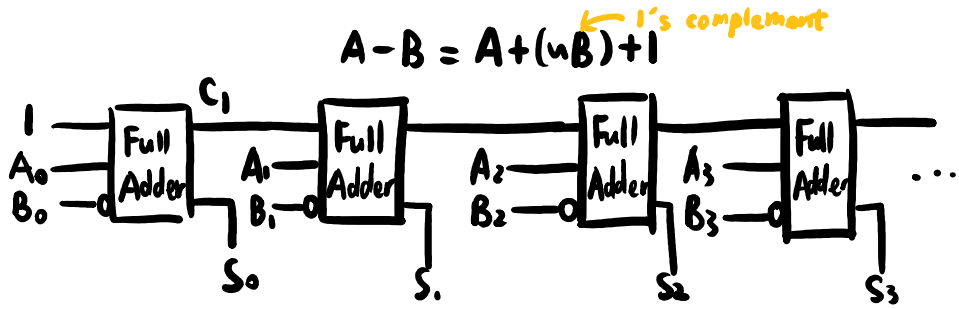
因此，加法器模块成为了许多数字电路中不可或缺的基础构件，尤其是在计算和逻辑运算中被大量使用。

4. 它与自身/其他模块/逻辑门进行简单的组合后能够产生怎样的新功能?

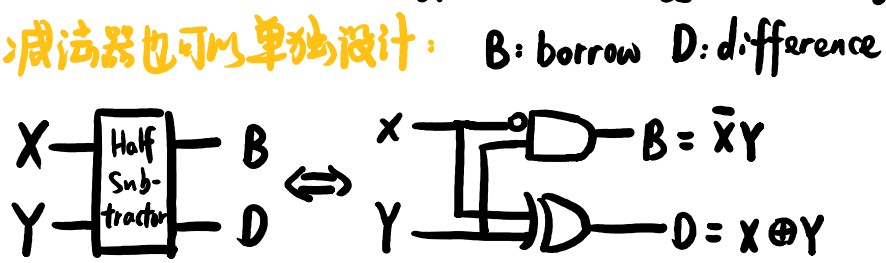
级联全加器：通过将多个全加器级联，构成多位加法器，实现多位二进制数的加法运算。这在算术逻辑单元 (ALU) 中非常常见。

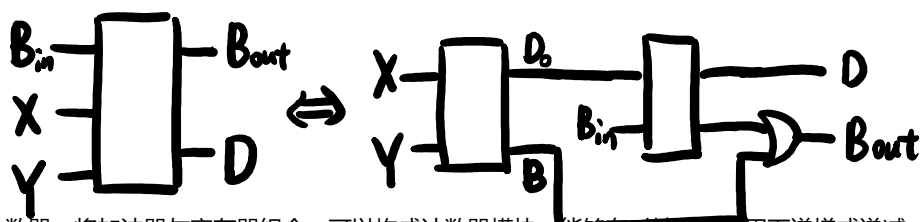


减法器：通过将输入与反相器组合，可以构成减法器。加法器可以用于构建带符号数的减法电路。



减法器也可以单独设计：





计数器：将加法器与寄存器组合，可以构成计数器模块，能够在时钟信号作用下递增或递减。

累加器：通过与存储单元组合，可以构成累加器，实现累积求和操作，这在数据处理和信号处理应用中非常常见。

5. 它最重要的应用有哪些?它是哪些微架构层次的核心器件（之一）?

累加器：通过与存储单元组合，可以构成累加器，实现累积求和操作，这在数据处理和信号处理应用中非常常见。

5. 它最重要的应用有哪些？它是哪些微架构层次的核心器件（之一）？

重要应用：加法器在计算机算术运算单元 (如 ALU)、数据路径、计时和控制系统中被广泛使用。算术逻辑单元 (ALU)：加法器是 ALU 的核心部分，负责处理加法、减法、移位等基本运算。

计数器：在时序电路中用作计时器或计数模块。

累加器：在信号处理中用于累加操作。

微架构层次中的核心器件：在微架构层次中，加法器是 ALU 的基础构件之一。ALU 是 CPU 的核心组成部分，用于执行算术和逻辑运算。

加法器同样用于寄存器传输层 (RTL) 的数据路径设计，用于数据的递增/递减和地址计算等操作。

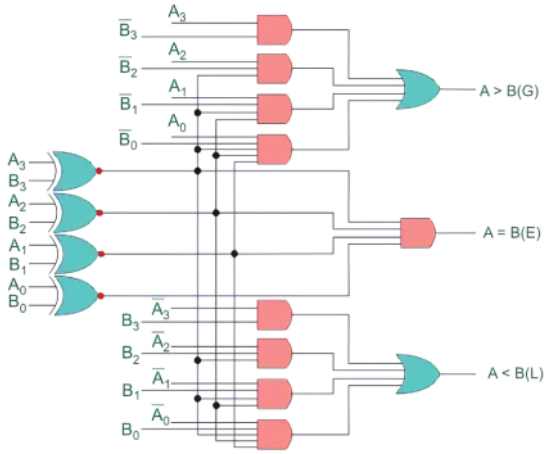
加法器模块因其高效的运算能力和模块化设计成为数字电路中不可或缺的基本构件，在各层次的电路设计中扮演重要角色。

等待补充笔记

Comparator（比较器）

比较器是一种用于比较两个二进制数的基本模块。它的输入通常为两个多位二进制数 (A 和 B)，输出则是表示它们之间关系的信号，例如 **A > B**、**A = B** 或 **A < B**。比较器的基本用途是判断两个数的大小关系，在决策控制和数据排序中广泛应用。

1. **该模块如何用逻辑门实现?**
   - 比较器可以通过逻辑门实现，例如使用 **XOR**、**AND** 和 **OR** 门等。对于单个位的比较，可以使用 XOR 门判断两个输入位是否相等。对于多位数的比较，通常采用级联结构，通过逐位比较和逻辑运算来生成整体结果。
   - 一个基本的**单位比较器**可以使用 XOR 门来判断两个位是否相等，然后使用 AND 和 OR 门结合多个单位比较器来实现多位比较器，最终得到整体大小关系。



$$A=B \Longleftrightarrow (\overline{A_0 \oplus B_0})(\overline{A_1 \oplus B_1})\cdots(\overline{A_n \oplus B_n})$$

$$A \quad B$$

$$> \Longleftrightarrow (A_n \bar{B_n})(A_{n-1}\overline{B_{n-1}} \;\&\; \overline{A_n \oplus B_n})\cdots(A_0\bar{B_0} + \sum_{i=1}^{n}(A_i \oplus B_i))$$

$$A<B \Longleftrightarrow (B_n \bar{A_n})(B_{n-1}\overline{A_{n-1}} \;\&\; \overline{B_n \oplus A_n})\cdots(B_0\bar{A_0} + \sum_{i=1}^{n}(B_i \oplus A_i))$$

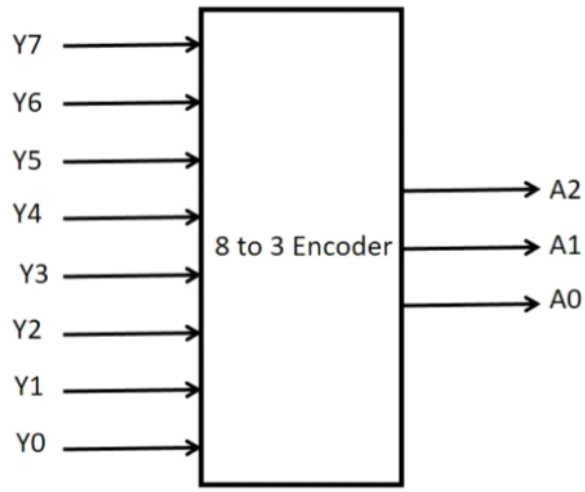
2. **它本身的功能是什么?**
   - 比较器的功能是比较两个输入数的大小关系，输出表示两个数是相等、大于还是小于。具体的输出信号有三种：**A > B**、**A = B**、**A < B**，可以同时或者单独输出这些比较结果。
3. **为什么这一功能让它成为一个被广泛使用的基本模块?**
   - 在许多数字系统中，需要根据数值的大小关系进行**判断和控制**，例如在**排序、选择、数据分配**等场景中。比较器提供了简单而高效的方法来实现这些判断功能，因而成为了数据处理和控制系统

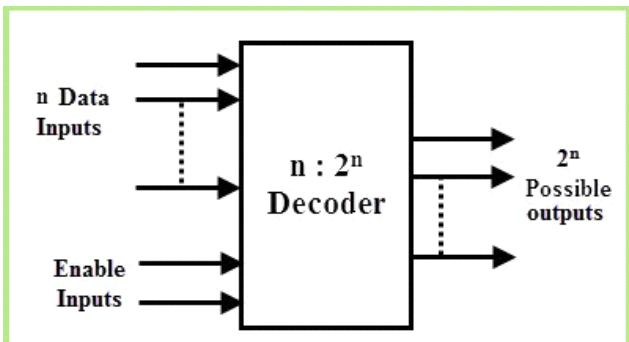
中常见的基本模块。

4. **它与自身/其他模块/逻辑门进行简单的组合后能够产生怎样的新功能?**
   - **与选择器（多路复用器）结合**：比较器的输出可以用来控制多路复用器，从而根据比较结果**选择**相应的数据或路径。
   - **与加法器和计数器组合**：在算法中比较器可以结合加法器和计数器进行条件性递增或递减，构成复杂的**控制逻辑。**
   - **构成排序网络**：多个比较器可以级联组成**排序**网络，用于对多个数进行排序操作，常用于数据排序和决策模块中。
5. **它最重要的应用有哪些？它是哪些微架构层次的核心器件（之一）？**
   - **重要应用**：比较器在控制系统、数据路径、排序电路、信号处理等领域广泛应用。例如，比较器在 CPU 中用于条件判断操作，在数据排序电路中用来确定数据的排列顺序。
   - **微架构层次中的核心器件**：在微架构层次中，比较器常用于控制单元和数据路径中的**条件判断和分支控制。**例如，比较器在条件判断逻辑和分支预测中起关键作用，是现代处理器中控制流和决策机制的基础组件之一。

Encoder/Decoder(编码器、解码器)

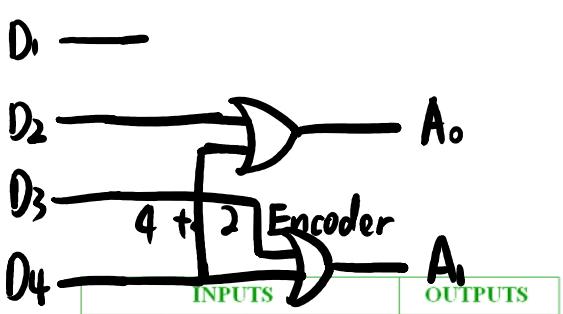
编码器 (Encoder) 和解码器 (Decoder) 是数字电路中常用的模块。编码器将多个输入转换为较少的输出位，通常用于**数据压缩或转换**；解码器则相反，将少量输入转换为多条输出线，通常用于数据的**解码和选择。**编码器和解码器在数据传输、地址选择和译码操作中有广泛应用。
*priority encoder：如果有多个输入同时激活，编号最大的线路有效



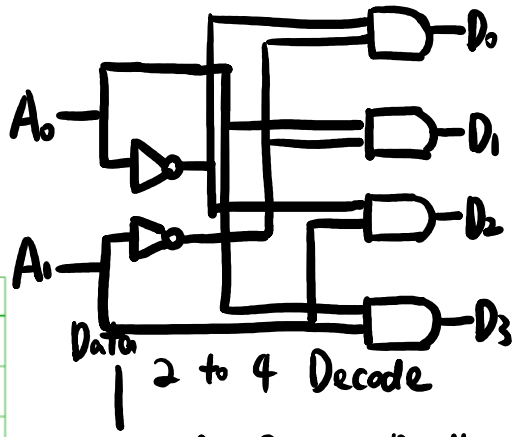


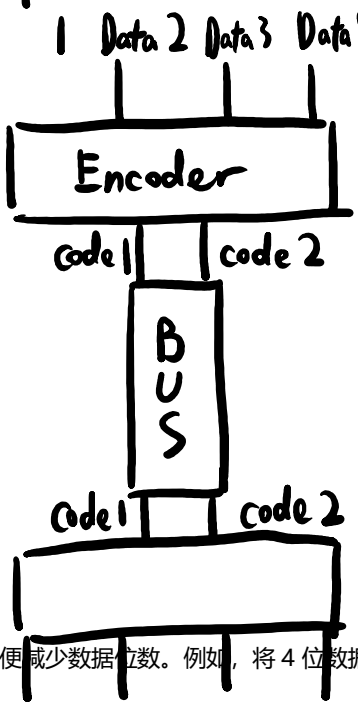
1. **该模块如何用逻辑门实现?**
   - **编码器 (Encoder)**: 编码器可以通过 OR 和 AND 门实现。例如，一个 4-to-2 编码器有 4 个输入 (D0, D1, D2, D3) 和 2 个输出 (Y1, Y0)，输出信号根据输入的二进制编码。例如，若 D2 为 1 且其他输入为 0，则输出为 Y1 = 1，Y0 = 0。
   - **解码器 (Decoder)**: 解码器通过 AND 和 NOT 门实现。例如，一个 2-to-4 解码器有 2 个输入 (A1, A0) 和 4 个输出 (Y0, Y1, Y2, Y3)。输入的二进制值将激活对应的输出。例如，当输入为 A1

= 0 和 A0 = 1 时，Y1 输出为 1，其他输出为 0。



INPUTS				OUTPUTS	
Y3	Y2	Y1	Y0	A1	A0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1





2 to 4 Decoder – Truth Table

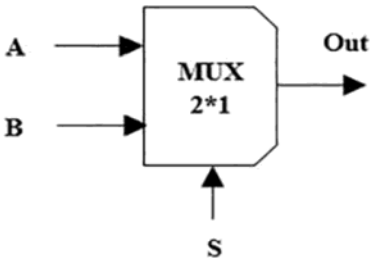
- 2 to 4 decoder

X1	X0	Y3	Y2	Y1	Y0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

2. **它本身的功能是什么?**
   - **编码器**的功能是将多个输入编码为较少的输出位，以便减少数据位数。例如，将 4 位数据输入压缩为 2 位输出。**用更少的编码信息传达并列的分支**
   - **解码器**的功能是将少数输入位解码为多个输出，用于激活特定的输出线，例如在地址译码中使用。**将传达的编码信息转化成对应的分支**
3. **为什么这一功能让它成为一个被广泛使用的基本模块?**
   - 编码器和解码器提供了有效的**信号转换**和**路由**方法。编码器通过减少数据位数提升数据传输效率，解码器则在地址选择和控制系统中用于快速定位和激活特定信号。它们的简便性和功能性使其成为数据传输和处理系统中的基本模块。
4. **它与自身/其他模块/逻辑门进行简单的组合后能够产生怎样的新功能?**
   - **与多路复用器 (MUX)/解复用器 (DeMUX) 组合**：编码器可以用来**选择多路复用器的输入**，而解码器则可以**与解复用器组合**，用于更复杂的信号分发和选择。
   - **构建地址译码器**：通过与存储器模块组合，解码器可以构建成**地址译码器**，用于内存和寄存器的地址访问。
   - **与计数器组合**：编码器可以配合计数器，实现顺序编码和控制。
5. **它最重要的应用有哪些? 它是哪些微架构层次的核心器件（之一）?**
   - **重要应用**：编码器用于数据压缩、信号编码和控制电路中；解码器广泛应用于地址译码、数据选择、指令解码和控制系统中。
   - **微架构层次中的核心器件**：在微架构层次中，解码器常用于 CPU 的控制单元和指令译码，负责将**操作码译码成控制信号。**编码器和解码器是数据路径和控制路径中的关键组件之一，帮助实现**数据选择**、**地址管理**和**信号控制**等功能。

MUX(Multiplexer)/DeMUX(Demultiplexer) (多路选择器/解复用器)

多路复用器 (Multiplexer, MUX) 和解复用器 (Demultiplexer, DeMUX) 是常用的数字模块。多路复用器的输入包括多个数据输入和选择信号，输出为单一的输出信号，选择信号决定哪个输入信号通过输出端输出。解复用器则相反，输入为单一数据输入，通过选择信号将数据分配到多个输出端。多路复用器主要用于信号选择和路由，解复用器则用于信号分配。



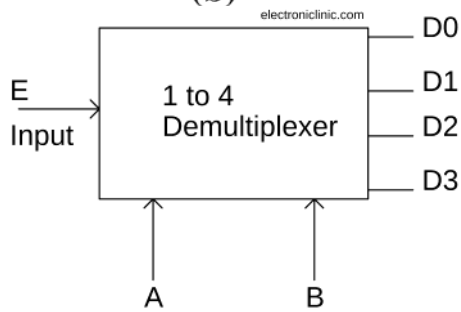
(a)

A	B	S	Out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(c)

S	Out
0	B
1	A

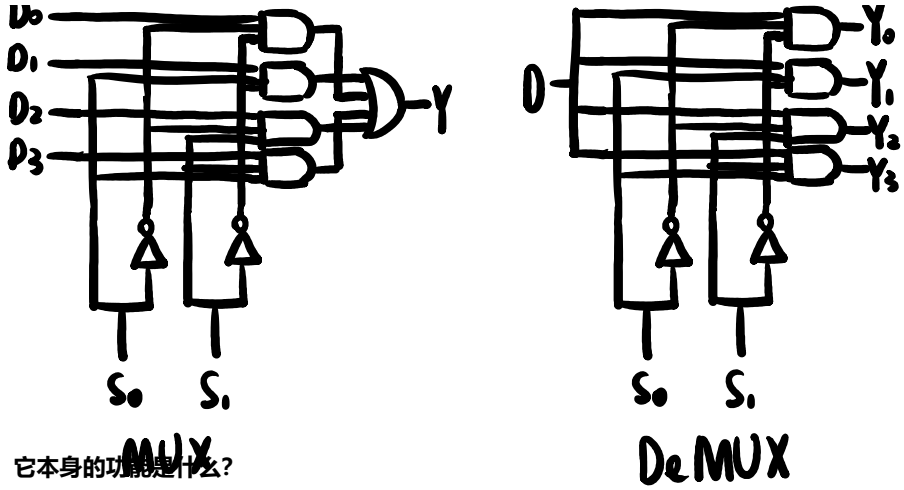
(b)



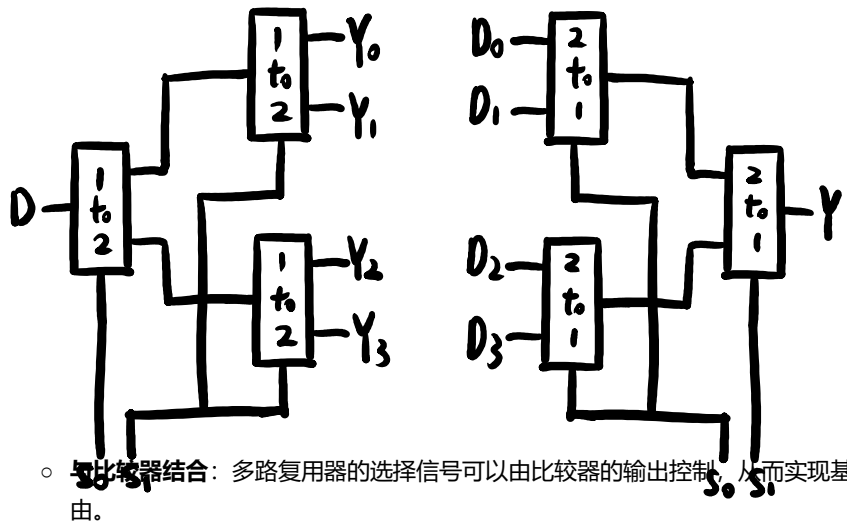
Data Input	Select Inputs			Outputs							
D	S_2	S_1	S_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
D	0	0	0	0	0	0	0	0	0	0	D
D	0	0	1	0	0	0	0	0	0	D	0
D	0	1	0	0	0	0	0	0	D	0	0
D	0	1	1	0	0	0	0	D	0	0	0
D	1	0	0	0	0	0	D	0	0	0	0
D	1	0	1	0	0	D	0	0	0	0	0
D	1	1	0	0	D	0	0	0	0	0	0
D	1	1	1	D	0	0	0	0	0	0	0

回答：

1. **该模块如何用逻辑门实现?**
   - **多路复用器 (MUX)**: 多路复用器可以通过 AND、OR 和 NOT 门来实现。例如，2-to-1 多路复用器有两个数据输入 (D0, D1) 和一个选择信号 (S)，输出为 Y=(¬S∧D0)∨(S∧D1)Y = (\neg S \land D0) \lor (S \land D1)Y=(¬S∧D0)∨(S∧D1)。
   - **解复用器 (DeMUX)**: 解复用器通过 AND 和 NOT 门实现。例如，1-to-2 解复用器有一个数据输入 (D) 和一个选择信号 (S)，两个输出分别为 Y0=¬S∧DY0 = \neg S \land DY0=¬S∧D 和 Y1=S∧DY1 = S \land DY1=S∧D。



2. **它本身的功能是什么?**
   - **多路复用器**的功能是选择一个输入信号并通过单一输出端输出，根据选择信号的不同可以实现**不同信号的路由。选择接受哪条路径上的信号**
   - **解复用器**的功能是将一个输入信号分配到多个输出端之一，这样可以实现**信号在多个输出端之间的分配。选择将信号分配到哪条路径上**
3. **为什么这一功能让它成为一个被广泛使用的基本模块?**
   - 多路复用器和解复用器提供了有效的信号选择和分配方法，在数据路由、通信系统、数据选择等领域广泛应用。其灵活性和简单的控制方式使得它们在各种数字电路中不可或缺，尤其是在需要高效数据传输和切换的应用中。
4. **它与自身/其他模块/逻辑门进行简单的组合后能够产生怎样的新功能?**
   - **与编码器/解码器组合**：多路复用器和解复用器可以与编码器和解码器组合，用于复杂的信号选择和数据分配场景。
   - **构建数据选择系统**：通过级联多个多路复用器可以实现更大规模的信号选择系统，例如 4-to-1 或 8-to-1 MUX。



   - **与比较器结合**：多路复用器的选择信号可以由比较器的输出控制，从而实现基于条件的信号路由。
5. **它最重要的应用有哪些?它是哪些微架构层次的核心器件（之一）?**
   - **重要应用**：多路复用器广泛应用于数据选择、通信系统、信号路由和处理器中的总线系统中。解复用器常用于数据分配、地址译码和信号分发等。
   - **微架构层次中的核心器件**：在微架构层次中，多路复用器是数据路径设计中的关键组件之一，用于控制不同数据源之间的数据流向。例如，在处理器中的 ALU 前，MUX 用于选择操作数；在存储器管理中，MUX 和 DeMUX 用于控制数据的读写路径。

*Encoder/Decoder与MUX/DeMUX的区别

Encoder/Decoder的传递的是信号本身（加密后的数据本身还是数据），而MUX/DeMUX是在不同信号中

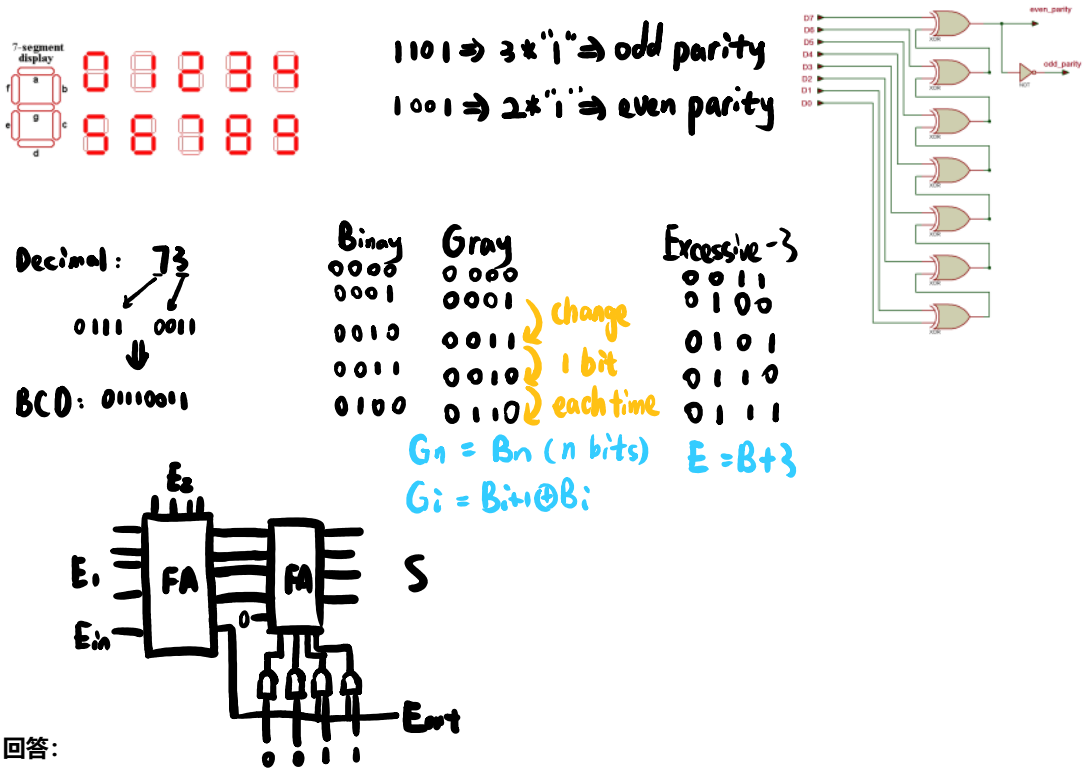
进行选择（只负责在多条数据通路上选择一条）

Code Converter（编码转换器）

代码转换器是一种用于在不同编码之间转换的数字电路模块。其输入为一种编码格式（如二进制、BCD 等），输出为另一种编码格式。代码转换器的基本用途是将数据从一种编码转换为另一种编码，以便适应不同的应用需求，例如在显示、数据处理和通信系统中使用。

常用的编码形式有：

Binary, Octal, Hexadecimal, Decimal; Gray code, Seven-segment display code, Binary-coded decimal, Excessive-3 code, Parity code, ASCLL, Unicode



回答：

1. **该模块如何用逻辑门实现？**
   - 代码转换器的实现方式取决于转换的编码类型。例如，**二进制到BCD**（二进制编码十进制）转换器可以通过组合多个逻辑门来实现输入到输出的映射关系。
   - 每种代码转换器的实现逻辑可能会有所不同。例如，**灰码到二进制**转换器可通过 XOR 门和 AND 门实现，逐位进行转换，确保转换结果符合灰码和二进制的规则。
2. **它本身的功能是什么？**
   - 代码转换器的功能是将输入数据从一种**编码格式**转换为另一种编码格式，以便在不同的系统或模块之间进行**数据传递和兼容**。例如，将二进制编码转换为 BCD 编码，可以将数据输出到七段显示器上，更**直观**地显示十进制数。
3. **为什么这一功能让它成为一个被广泛使用的基本模块？**
   - 许多数字系统和设备需要在不同编码格式之间转换以实现兼容性。代码转换器提供了高效的编码转换方法，使得数据在不同模块或设备间传输时**无需重新设计编码方式**。因此，它在数据处理、显示设备和通信系统中得到了广泛应用。
4. **它与自身/其他模块/逻辑门进行简单的组合后能够产生怎样的新功能？**
   - **与显示器模块组合**：代码转换器可以与七段显示器模块结合，将二进制或 BCD 数据转换为显示格式，用于直观地显示数值。
   - **与计数器组合**：在计数系统中，代码转换器可将二进制计数值转换为可读的十进制值，以便在**显示设备上输出。**
   - **与存储器和处理器组合**：在数据传输过程中，代码转换器可以与存储器或处理器组合，确保数据在不同编码格式之间**无误**传递。

5. **它最重要的应用有哪些？它是哪些微架构层次的核心器件（之一）？**
    - **重要应用**：代码转换器广泛应用于**显示设备**（如七段显示器）、数据通信、数据存储和数据处理系统中。例如，在数字钟表、计算器和电子仪表中，代码转换器用于将数据转换为可显示格式。
    - **微架构层次中的核心器件**：在微架构层次中，代码转换器可以作为数据路径中的辅助模块，确保数据以正确的编码格式传输和显示。例如，在显示系统中，代码转换器将 CPU 输出的二进制数据转换为 BCD 或其他编码格式，以便显示在屏幕或仪表上。

正负逻辑和混合逻辑

正负逻辑

在物理电路和理论电路中，逻辑赋值的方式并不唯一。例如，通常我们将 **0** 表示为低电平（low voltage），**1** 表示为高电平（high voltage），这被称为**正逻辑**；但我们同样可以选择**1** 表示为低电平，**0** 表示为高电平，这种情况被称为**负逻辑**。从逻辑赋值的对称性来看，这两种方式本质上是等价的。
在布尔代数中，**AND** 和 **OR** 运算具有对偶性，这一点可以通过以下公式看出：

- $A+B=(A'\cdot B')'$
- $A\cdot B=(A'+B')'$

这一性质在真值表中表现得更加直观。例如，在正逻辑中，如果我们将 0 表示为低电平、1 表示为高电平，

那么电路的逻辑行为可以用 A·B（AND）形式表示。但如果我们将 1 表示为低电平、0 表示为高电平（负逻辑），那么电路的逻辑行为可以用 A′+B′（OR）形式表示。两者在逻辑上是完全对称的，没有本质区别。

A	B	$A \cdot B$	A'	B'	$A' + B'$
0	0	0	1	1	1
0	1	0	1	0	1
1	0	0	0	1	1
1	1	1	0	0	0

这种对称性在更复杂的布尔表达式中也成立。例如：

- AB+CD（正逻辑形式）
- 等价于 [(A′+B′)(C′+D′)]′或 [(A′)′(B′)′+(C′)′(D′)′]（负逻辑形式）

从以上表达式可以看出，通过对逻辑取反并调整运算的组织方式，我们可以在负逻辑中实现与正逻辑完全相同的逻辑功能，输出的真值表完全一致。
总结来说，对于正逻辑中的运算 **AND, OR, NOT**，只需要将输入值取反，并将运算替换为 **OR, AND, NOT**（对偶运算），即可实现同样的逻辑功能。这一对偶性体现了布尔代数和逻辑电路的基本对称性，也是正逻辑与负逻辑等价的理论基础。

混合逻辑

在分析电路时，我们可以同时利用**正逻辑**和**负逻辑**来进行设计和优化。这两种逻辑之间是可以互相转换的，其核心方法是通过**取反并取共轭运算**，这本质上就是利用了**德摩根定理（De Morgan's Theorem）**的性质。

正负逻辑的转换及其实现
通过正负逻辑之间的转换，我们可以利用逻辑运算的对偶性来实现同等性质的运算。例如：

- **在正逻辑中**，加法（OR）直接用 OR 门实现，乘法直接用AND门来实现。
- **在负逻辑中**，加法可以通过对输入信号取反后使用 AND 门来实现，乘法可以通过对输入信号取反后使用 OR门来实现，而输出结果仍然是负逻辑的表达形式

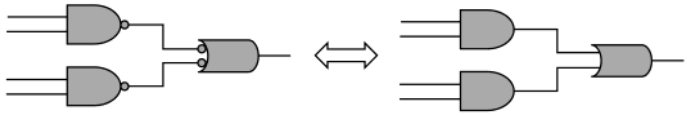
这说明，只要理解正负逻辑之间的转换关系，我们可以在两种逻辑形式之间灵活切换，并通过适当的运算组织达到相同的功能。
例如，对于某个电路的输出，其逻辑门的等效可以用正负逻辑之间的转换表示。这种等效转换才是真正体现了正负逻辑之间的对偶性。

混合逻辑的出现与应用

在实际的电路设计中，经常会遇到所谓的“**混合逻辑**”电路。虽然逻辑上可能同时包含正逻辑和负逻辑，但从分析角度出发，我们通常是以正逻辑的视角来理解和优化的。负逻辑在这种场景中，往往被视为一种为了利用逻辑门的等效性而出现的“**中间状态**”。

从逻辑的完备性上来说，加法（OR）、乘法（AND）和取反（NOT）已经足够描述任何布尔函数。但在实际硬件实现中，不同逻辑门的生产成本不同，因此合理利用正负逻辑之间的等效性，可以显著优化电路的成本。

利用正负逻辑的优化示例



比如，我们希望实现一个基础的二级逻辑 AB+CD。在正逻辑下，这通常需要用 AND 门和 OR 门来实现。但如果我们利用负逻辑的思想，可以发现：

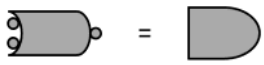
- A NAND B和 A'+B'是等价的。

这意味着，我们可以用一个 NAND 门（负逻辑的运算）来实现 OR 门的功能。其核心原理在于：

- 通过德摩根定理，路径中的两次取反会相互抵消，从而在正逻辑中实现等价运算。

这种方式不仅降低了逻辑门的种类需求，也减少了硬件成本。

所以对于这一个代换，逻辑门的等效：



才是真正的正负逻辑之间的转换

未完全等效的情况及其修正

然而，如果电路中的取反并未完全相消，那么逻辑门的等效性不能完全实现。此时，实际表达式中必须包含未抵消的取反操作。也就是说：

- 我们“读图”时，看到的是逻辑门的等效功能。
- 但从原理上讲，电路的实际表达式可能需要额外的修正以反映未完全等效的部分。

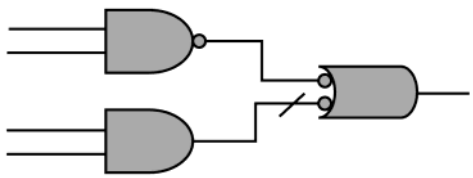
例如，当我们希望用 NAND 门实现 OR 功能时，可能得到如下形式的电路：

- **期望的形式**：逻辑完全等效，不需要额外的取反修正。
- **实际的形式**：由于取反未相消，需额外用斜杠（/）标注，提醒逻辑未完全等效。

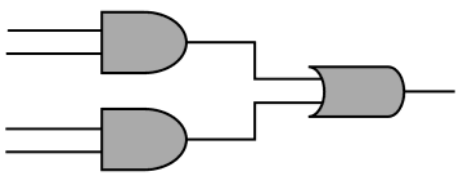
在这种情况下，正确的表达式需要结合取反操作进行修正，以准确反映电路功能。

就比如说，我们通过负逻辑的方式，希望用NAND来实现OR的功能，得到了这样的一幅图：

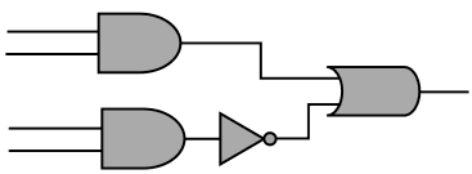
这里，斜杠/(slash)是用来标注取反没有相消的，相当于对“没有完全等效”起了警示作用



我们期望它可以呈现出这种形式：



但它实际上等价为这样的形式：



所以我们写表达式的时候，其实是先假设它“完全等效”为相应的逻辑门，然后再把一些事实性的、没有等效好的地方的表达式通过取反进行修正，得到回路所代表的表达式

在设计的时候，我们也可以先利用“成本低的”或者“手头有的”门，利用正负逻辑的思想等效转化为我们想要的门，然后通过引入NOT对其进行调整，使得我们能够恰好得到我们需要的表达式

设计中的策略

在电路设计中，为了降低成本或适应手头现有的器件资源，可以利用正负逻辑的思想进行优化：

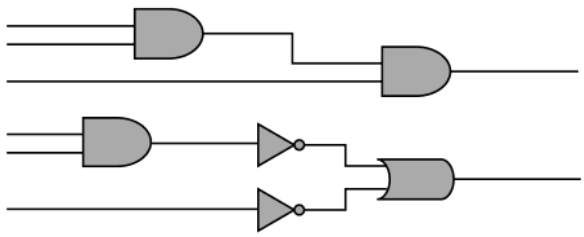
1. 使用负逻辑的门（如 NAND 门）等效实现正逻辑的功能。
2. 根据需要引入取反操作（NOT 门）对未完全等效的部分进行调整。
3. 最终实现所需的逻辑表达式，同时最小化硬件成本。

通过这种方法，我们能够灵活地利用正负逻辑之间的对偶性，在满足功能需求的同时，优化资源利用和实现效率。

这种基于正负逻辑转换的设计思路，是逻辑电路优化的核心，也是德摩根定理在实际硬件设计中的典型应用。

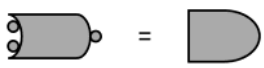
(下面是draft)
然而，即使是在同一回路中，我们也可以同时利用正逻辑和负逻辑
正负逻辑之间是可以转换的，方法就是，“取反并取共轭运算”
也就是说，逻辑之间的转换，**本质上是使用De Morgan Theorem**

转换之后，就可以利用运算的对偶性，实现同等性质的运算
比如我们想实现加法，我们可以利用取反将信号从正逻辑转化为负逻辑，然后用乘法对其进行实现（当然，输出的结果也在负逻辑之中）



（第二个输出的是反逻辑）

所以对于这一个代换，逻辑门的等效：



才是真正的正反逻辑之间的转换

在分析电路的时候，我们也会遇见所谓“混合逻辑”电路

但其实，我们一直是以正逻辑的视角去看的；在这个过程中，“负逻辑”被视为了“为了利用逻辑门等效而出现的一种中间状态”

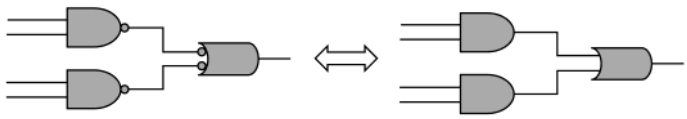
毕竟其实从逻辑的角度上而言，加法、乘法和取反已经完全够用了（完备性），为什么还要引入正负逻辑呢？就是因为不同的逻辑门在生产中有不同的成本，若能利用好逻辑门之间的等效就可以对成本进行优化

比如我们想实现一个基础的二级逻辑：先乘再加
我们当然可以

如果出现了连续的两次转换，那么信号经过传输后还是正逻辑，逻辑门等效便顺理成章了

如果这一“中间状态”没有很好被“矫正”回正逻辑，由于其和纯粹的逻辑门是不等效的，而是还有一项取反，则我们需要修正表达式

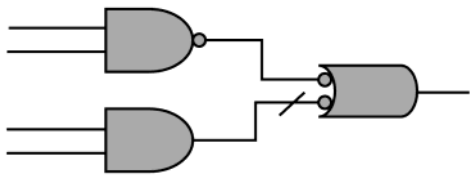
比如说，我们希望实现一个基本的二级逻辑：AB+CD
正常情况下，我们需要两种门
但是利用反逻辑的思想，我们发现：A NAND B 和 A' OR B' 是等价的
我们就可以用一个理应为“AND”的门，实现“OR”的功能



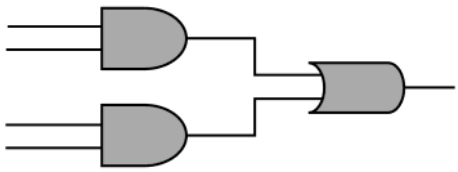
因为路径之中的两次取反相消了，我们相当于在一个正逻辑中，运用了AND和OR进行运算
这样，我们就利用了De Morgan Theorem所代表的正负逻辑转换，用“NAND”实现了“OR”

但如果这种取反并没有相消，我们就不能认为，这种逻辑门的“等效”是被完全实现的，其正确的表达式应该有一项取反；就是说，我们“读图”的时候，读的是等效后的逻辑门；但是原来的回路其实并没有实现真正的等效，所以为了获得原来的回路所实现的正确的表达式，我们需要通过取反对这种“彻底的”等效进行修正

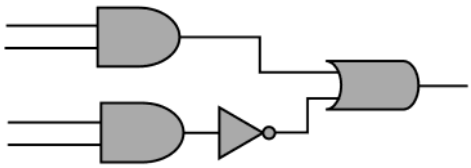
就比如说，我们通过负逻辑的方式，希望用NAND来实现OR的功能，得到了这样的一幅图：
这里，斜杠/(slash)是用来标注取反没有相消的，相当于对“没有完全等效”起了警示作用



我们期望它可以呈现出这种形式：



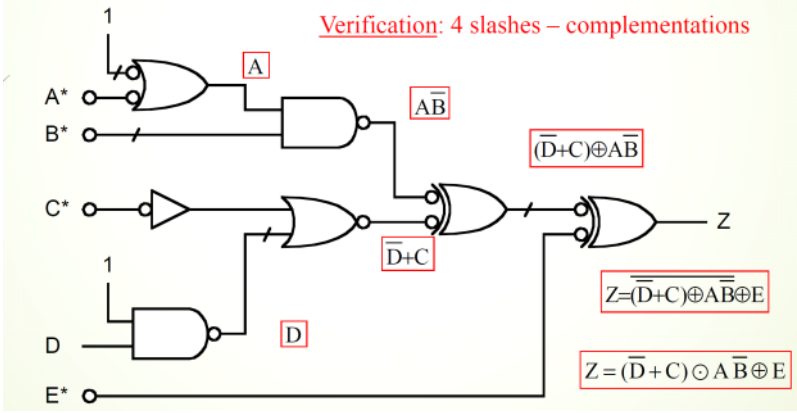
但它实际上等价为这样的形式：



所以我们写表达式的时候，其实是先假设它“完全等效”为相应的逻辑门，然后再把一些事实性的、没有等效好的地方的表达式通过取反进行修正，得到回路所代表的表达式

在设计的时候，我们也可以先利用“成本低的”或者“手头有的”门，利用正负逻辑的思想等效转化为我们想要的门，然后通过引入NOT对其进行调整，使得我们能够恰好得到我们需要的表达式

Example 3 (Cont'd)



时序电路

Wednesday, December 4, 2024 10:20 AM

时序含义

简而言之，时序代表“信息的保持”，是“某一时刻上，回路中信息被接收到的时刻构成的序列”

对于一个系统而言，其接受信息，处理信息，输出信息

组合逻辑的特点是，被接受的信息会立刻进入处理、输出的进程

而时序逻辑中，这个被接受的信息，不一定会被立刻使用；这一机制叫做“记忆”，也就是说，此刻所接受的信息不一定会被在此刻处理，此刻同时处理的信息不一定在此刻被输出；此时的信息可能被“保留”（“存储”），从而在另一个时刻（也就是“以后”的时刻，因为时间是单向的）被处理
广义地说，记忆便是：此时的信息在之后的时刻被处理
信息的“保留”是实现之的必要手段，因为在“之后的时刻”中，之前的信息应该仍然存在

如果我们想象用接受时的时刻赋予信息一个标签，对于组合逻辑而言，这一标签几乎是无意义的，因为回路中的所有信息的时间标签几乎可以认为是同一的，在运行时信息之间没有时间的差别；而对于时序逻辑而言，由于回路中存在“先前”时刻存储（“保留”）的信息，信息之间就有了时间上的差别，或者说时间上的“先后”，这便是所谓“时序”

时序还有一个特性是“串行阻断”，也就是说，对于一个**串行**的回路，“信息的记忆”这一性质会影响其它不涉及记忆部分的行为，使得回路作为一个整体产生时序：
对于记忆单元之前的回路部分，由于记忆意味着“保留先前的信息”，这也意味着此时的信息不能被接受，导致无论前面的部分在这段时间内怎么变化，其产生信息并没有被后续的回路接受，是“无效的”，而“有效的”只有在记忆单元开始接受新的信息的时刻所产生的信息
对于记忆单元之后的回路部分，由于记忆单元的存在，其行为会受到前一时刻记忆单元输出的信息所限制。也就是说，记忆单元输出的“先前时刻的信息”会作为此刻输入后续回路的唯一有效信息，进而影响后续回路的行为。由于记忆单元只在特定时刻更新输入并输出新信息，在其未更新时，后续回路所处理的仍然是记忆单元“记住的”先前信息。

这种特性意味着：

1. 后续回路的行为具有延迟性
   后续回路的状态变化总是滞后于当前时刻的输入，因为它依赖于记忆单元的输出，而记忆单元的输出是之前一个有效输入的结果。
2. 信息的“串行传递”特性
   记忆单元在每个更新周期只允许一次有效信息的传递，这会使得后续回路不能立即响应前段回路的所有变化，而是基于记忆单元的更新节奏进行处理。
3. 时序的传递性
   记忆单元在时间上的“阻断”作用会直接将时序的特性传递到后续回路。后续回路的逻辑处理将变成一个基于时间步进的序列化过程，而不再是纯粹的并行逻辑。

也就是说，整个回路因为“记忆单元”对于前后的组合逻辑部分的影响，可以视为只在特定的时刻工作。
这种“串行阻断”特性使得后续回路在工作时只能以固定的时间步长（由记忆单元的更新周期决定）逐步推进，无法实时响应当前的输入。这既为时序逻辑提供了有序性与控制力，也带来了延迟与同步的需求。在实际设计中，必须考虑这种延迟对整体系统性能的影响，并通过优化时钟周期、减少冗余逻辑等方式进行平衡。

时序的种类

上面提到，时序是“某一时刻上，回路中信息被接收到的时刻构成的序列”
一般而言，根据信息接受的机制不同，信息也能构成不同种类的序列；大体分为两种：同步逻辑(Synchronous sequential circuit)和异步逻辑(Asynchronous sequential circuit)

同步逻辑是一种“离散化的序列”，信息的接受与存储是按照预先规定周期进行的，只有在这个预先周期的特定阶段，信息才能被接受和存储
异步逻辑是一种“连续化的序列”，信息的接受与存储在特定条件满足时便可进行，没有固定的时间上的规定与限制，能够产生的“时间标签”是连续而多样的

同步逻辑可以理解为“离散化的时间序列”，即只有在预先规定的时刻（通常由时钟信号来触发）才会接收并存储信息。简而言之，系统中的所有触发器或存储元件会在同一拍（clock edge）或者同一时钟周期的某一特定阶段（例如上升沿、下降沿）进行数据的采样或更新。这就导致系统中信息的更新、流动都“跟着”时钟跑，时间被切割为一个一个的周期。
异步逻辑则是“连续化的时间序列”，不需要统一的全局时钟信号来控制信息的接收或存储，只要满足特定的逻辑条件或协议，数据就可以在任何时刻被采集、传播、存储。系统中没有一个明确的、统一的节拍，信息在不同模块或不同部分之间依赖彼此的握手信号、就绪信号或者其他逻辑关系来决定下一步何时发生。

从本质上说，二者的区别就是：
同步逻辑：信息处理“依赖一个统一的时钟”，在离散化的时刻发生。
异步逻辑：信息处理“只依赖必要的条件和局部控制”，在没有统一时钟的情况下也可以进行。

记忆元件

所谓“记忆元件”，就是能够在一段时间内保存（保持）二进制信息（通常为 0 或 1）的电路结构。这种“保持”的能力使时序逻辑具备了与组合逻辑不同的关键特征，即系统对“当前”与“过去”的信息进行区分，从而产生时序行为。

记忆元件基本的特性为“可读”和“可写”。严格而言，所有元件都是“可读可写”的；但是对于一些元件，其只能进行一次写入，或者其写入的条件物理性地非常苛刻，我们也认为其是“不可写”的。这样，“可读”被认为是最重要的特性。

以下内容简要介绍时序逻辑中用于“记忆”的核心元件——双稳态器件（如锁存器、门控锁存器、触发器）是如何在物理和逻辑层面实现的，以及它们各自的特性。

双稳态器件

要实现信息的存储，我们要实现以下两点：
(1):信息写入控制：我们要把控信息，使其只能在特定的时间内写入
(2):信息维持：在信息写入之后，我们要保证其在需要的时间内能够维持原状，否则便损失了相应的信息
而对于电子系统而言，信息最基本的储存单位是一个二进制的比特：0和1；“信息维持”在这一语境下，便要求我们的装置可以稳定地保持写入的信息不变
对于能够满足这一性质的器件，我们称之为“双稳态”器件；可以看出，双稳态器件是实现电子系统中的信息存储所必不可少的一环

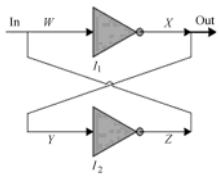
定义
“双稳态”指的是电路在物理上能够稳定地停留在两种不同的输出状态（往往对应数字电路的 0 和 1），而不自行跳变到其它状态。

为什么双稳态器件可以用来实现记忆?
因为它拥有两个稳定的平衡点，一旦电路进入其中一个状态，在没有外部干扰或改变条件的情况下，就会一直保持这个状态不变。这正是存储一位信息所需要的特性。

为什么记忆需要用双稳态器件?
若电路无法形成两个互不干扰的稳定状态，就会出现状态难以预测或自发摆动的问题，无法可靠地“记住”数据。双稳态的特性保证了数据的确定性和可控性。

种类
以下是不同类型的双稳态器件（Bistable Devices）的实现方式列表，根据其工作原理分类：

基于逻辑反馈的电子双稳态器件
- 锁存器（Latch）
  - SR锁存器（Set-Reset Latch）
  - D锁存器（Data Latch）
  - 门控锁存器（Gated Latch）
- 触发器（Flip-Flop）
  - D触发器
  - JK触发器
  - T触发器
  - SR触发器
- 静态随机存储器（SRAM）单元
  - 由交叉耦合反相器（Cross-Coupled Inverters）组成，用于存储单个比特信息。



基于逻辑门的锁存器是我们在数字电路这一语境下考虑的主要存储方式，但并不只有锁存器可以组成双稳态器件；以下是一些其它实现方式下的双稳态器件

基于物理特性的双稳态器件

- 电容存储器件
  - 动态随机存储器（DRAM）：通过电容充电和放电实现0和1，但需要周期性刷新。
  - 铁电存储器（FeRAM）：利用铁电材料的极化特性存储数据。
- 磁性存储器件
  - 磁芯存储器（Magnetic Core Memory）：利用磁场方向表示两个稳定状态。
  - 磁阻式随机存储器（MRAM）：通过磁性层的自旋方向存储数据。
- 电阻式存储器件
  - 阻变存储器（RRAM）：通过高电阻和低电阻状态实现双稳态。
  - 相变存储器（PCM）：利用材料相态（晶态与非晶态）转换存储数据。

基于光学或机械原理的双稳态器件

- 光学双稳态器件
  - 激光二极管中的双稳态：基于光学增益和反馈的稳定光学输出状态。
  - 光学存储介质（如CD、DVD）：利用反射率的高低存储数据。
- 机械双稳态器件
  - 机械开关：例如弹簧加载的开关（如普通的灯光开关），具有两个稳定的机械位置。
  - 微机电系统（MEMS）：利用机械结构实现双稳态。

基于量子特性的双稳态器件

- 量子比特（Qubit）
  - 利用量子态的叠加和纠缠，形成双稳态。
  - 例如超导量子比特和自旋量子比特。

生物学和化学中的双稳态现象

- 生物开关
  - 基因调控网络中的双稳态：例如开-关基因表达状态。
- 化学反应系统
  - 化学振荡反应中通过催化剂浓度维持的双稳态。

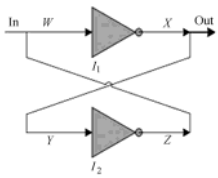
总结

双稳态器件的实现方式涵盖了广泛的领域，包括电子、光学、磁性、机械以及量子领域。虽然实现原理各异，但它们的共同特性是拥有两个稳定的状态，可以可靠地存储信息或保持状态。

锁存器 (Latch)

构造

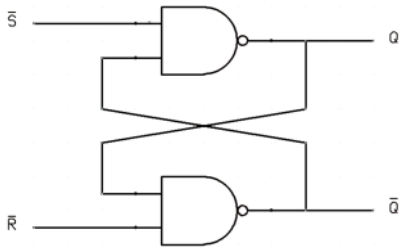
本质上由交叉耦合反相器组成

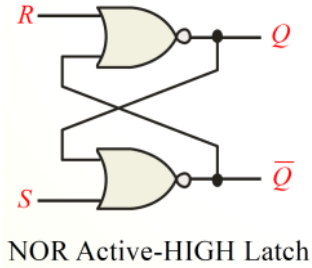


最基本的 SR 锁存器由两级交叉耦合的 NAND 或 NOR 门实现。它能够在输入满足某种条件时保持内部状态不变，构成一个简单的“双稳态结构”。这里的NAND和NOR相当于为NOT(反相器)的写入提供了一种更简单的方式；

其中NOR构建的锁存器被称为High activated的

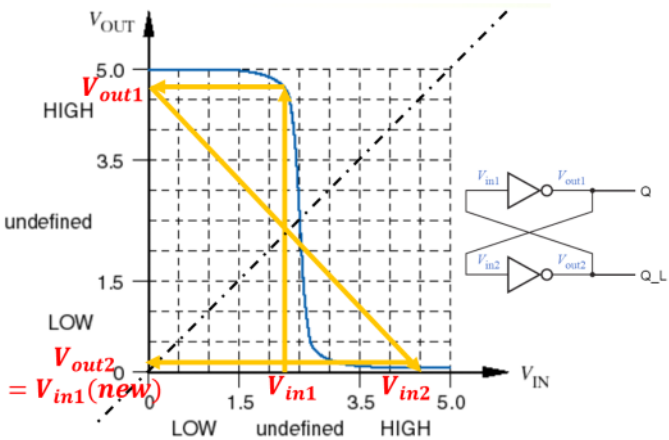
NAND构建的被称为为Low activated的





NOR Active-HIGH Latch

物理特性：为什么是双稳态?



在一个交叉耦合结构中，每个门的输出会通过反馈线作为另一个门的输入。由于逻辑门本身具有增益和阈值特性，在输出趋于高电平或低电平后，会“抵抗”小的扰动，因此形成两个稳定平衡点。当某一路输出被“置高”时，另一边被“置低”，并维持此状态直至外部信号强制其切换。

种类

1. SR 锁存器

SR 锁存器通过 S（Set）和 R（Reset）信号直接控制状态，是最基本的锁存器类型之一。

操作方式：

S = 1，R = 0：输出被置为 1（Q = 1）。

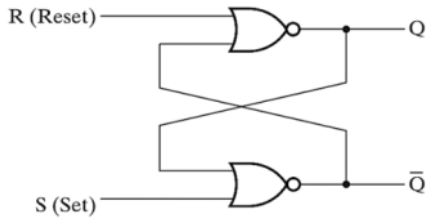
S = 0，R = 1：输出被置为 0（Q = 0）。

S = 0，R = 0：保持当前状态。

S = 1，R = 1：无效状态（不可用）。

特点：设计简单，但在 S 和 R 同时为 1 时会出现不确定的无效状态。

应用场景：用于简单的状态存储和控制电路中。



2. D 锁存器

D 锁存器通过单一的 D（Data）输入，配合控制信号（Enable），实现数据存储功能。

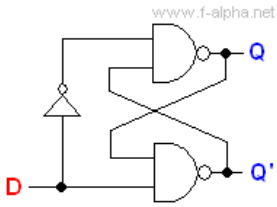
操作方式：

当 Enable = 1 时，输出 Q 跟随输入 D 的状态（Q = D）。

当 Enable = 0 时，输出保持当前状态。

特点：没有无效状态，输入和输出关系简单直接。

应用场景：用于数据寄存器和同步电路中，是存储单个数据位的理想选择。



3. T 锁存器

T 锁存器基于“Toggle（翻转）”操作，常用于计数器和状态机设计中。

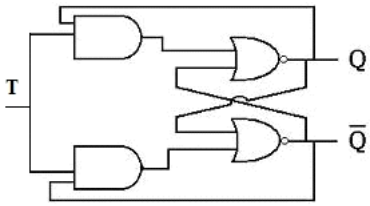
操作方式：

当 T = 0 时，输出保持不变。

当 T = 1 时，输出状态翻转（Q = NOT Q）。

特点：翻转功能内置，不需要额外的逻辑电路支持。

应用场景：适合构建二进制计数器和周期性翻转逻辑。



4. JK 锁存器

JK 锁存器是 SR 锁存器的扩展版本，解决了 SR 锁存器的无效状态问题，并增加了多功能性。（结构上有一点像 T+SR）

操作方式：

J = 0，K = 0：保持当前状态。

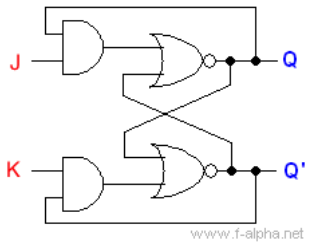
J = 1，K = 0：置位（Q = 1）。

J = 0，K = 1：复位（Q = 0）。

J = 1，K = 1：输出翻转（Q = NOT Q）。

特点：功能强大，可以替代其他类型的锁存器实现置位、复位、保持和翻转功能。

应用场景：用于计数器、复杂的状态机和其他多功能逻辑电路。



5. D' 锁存器

D' 锁存器是 D 锁存器的一个变体，专注于处理输入数据的补信号（D'）。

操作方式：

当 Enable = 1 时，输出为输入 D 的反相状态（Q = D'）。

当 Enable = 0 时，输出保持当前状态。

特点：用于需要对数据进行反相存储或差分信号处理的场景。

应用场景：差分信号处理、冗余设计或需要反相逻辑的系统。

(同时也存在T'锁存器，也是一个意思)

总结

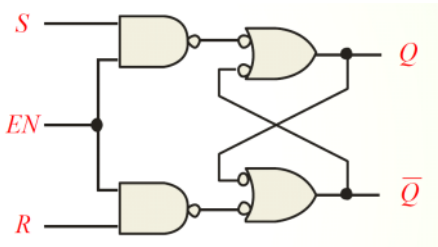
锁存器类型	输入信号	功能	主要应用场景
SR	S, R	置位、复位、保持状态	简单存储与控制电路
D	D, Enable	数据存储	数据寄存器、同步电路
T	T, Enable	状态翻转	二进制计数器、状态机
JK	J, K	置位、复位、翻转功能	计数器、复杂逻辑电路
D'	D', Enable	反相数据存储	差分信号处理、冗余设计

门控锁存器 (Gated Latch)

实现记忆元件所需的另一条件便是“我们要把控信息，使其只能在特定的时间内写入”，门控便是这一功能的体现

构造

在简单锁存器的基础上增加一个“门控”信号（通常称为 Enable），使锁存器只在该信号为有效时接收并更新输入；否则维持原有状态不变。



最简单的实现方式便是用一个AND门把输入信号和门控信号连接在一起

特性

1. 能够更好地配合同步电路使用，只有在合适的时间（门控信号有效时）才更新状态。
2. 依旧是电平敏感（Level-Sensitive）的：只要门控信号处于有效电平，一旦输入变化，锁存器可能更新多次。

触发器 (Flip-Flop)

通过对于控制clk作为门控信号的方式，使得Latch其只在时钟的改变边沿进行信息读取，从而使其工作的时间范围更为狭窄，从而增强同步时序逻辑中时刻的离散性

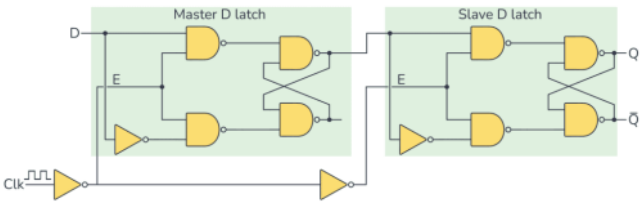
这一性质叫做"Edge-triggered"；历史上，Edge-triggered和Level-triggered的Gated Latch均被称作为Flip-flop，但现代的称呼一般还是将Edge-triggered特指为flip-flop，而level-triggered的被视作一种gated latch

构造

触发器通常由两个或更多级的锁存器级联，并在关键位置采用边沿触发机制（Edge-Triggered），使其只在时钟信号上升沿或下降沿的瞬间采样输入并更新输出，之后保持不变。

这一机制形成的结构基础为主从结构

主从结构（Master-Slave）是一种用于构建触发器（Flip-Flop）的设计方式，通过组合两个级联的锁存器（Latch）以及控制时钟信号的门控策略，实现对输入信号的边沿敏感采样，并保持稳定的输出状态。



（下面的D filp-flop也用了这张图）

主从结构通过以下方式实现更精准的同步时序控制：

1. 主锁存器（Master Latch）：

在时钟信号的一个状态（如高电平）下采样输入信号，并保持该值直到时钟状态改变。

2. 从锁存器（Slave Latch）：

在时钟信号的另一状态（如低电平）下读取主锁存器的输出，并将其值传递到触发器的输出端。

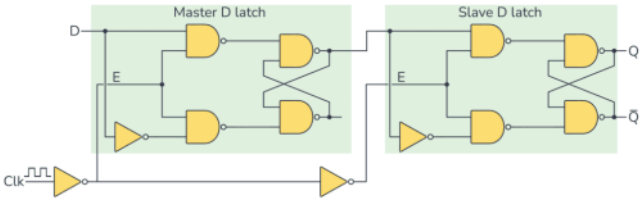
通过这种结构，主锁存器与从锁存器在不同的时钟阶段工作，确保输入信号只能在时钟沿（上升沿或下降沿）发生改变时影响输出。

特性

1. 边沿敏感（Edge-Sensitive）：相比门控锁存器，触发器只在时钟跳变的一瞬间更新数据，避免了电平敏感的多次更新问题。
2. 同步设计主力：绝大部分同步电路使用触发器来在时钟沿时完成数据采样和状态更新。

种类

1. D 触发器：最常见的类型，只有一个数据输入 D 和时钟输入 CLK，在时钟沿到来时将 D 的值锁存到输出。



2. T 触发器：输出会在时钟沿时根据 T 信号翻转或保持原状态，常用于计数器。
   （这里由于沿用历史的称呼，找不到“edge-triggered”的T、JK触发器的图片，故不提及）
3. JK 触发器：可实现多种功能模式（置位、复位、保持、翻转），设计上更灵活，但现代数字电路更普遍使用 D 触发器。

所有这些记忆元件本质上都是利用“双稳态”特性来存储信息：电路在 0 和 1 两个稳定态之间可控地切换，并在没有外部信号干扰时保持某一状态。物理实现通常依赖交叉耦合的逻辑门（如 NAND、NOR、Inverter）构成反馈回路，从而在电路层面形成“自保持”能力。通过进一步的门控或时钟信号控制，就可以使锁存器或触发器在更精确的时间点进行采样或更新，从而为时序逻辑提供严格的同步或异步的时序基础。

寄存器

由多个触发器串联而成

存储器件

分类

缓存（Cache Memory）
缓存是位于 **CPU** 和 **主存储器（RAM）** 之间的一种小型、高速的存储器。它通过存储 CPU 经常访问的数据，减少了 CPU 与主存储器之间的访问延迟，从而提高了整体处理速度。缓存通常由 **静态 RAM（SRAM）** 构成，其访问速度比主存储器中的 **动态 RAM（DRAM）** 快得多。缓存分为多个层次，包括 **L1 缓存**、**L2 缓存** 和 **L3 缓存**，分别位于 CPU 内部和 CPU 外部，用于存储最常用的数据和指令。

主存储器（Primary Memory/Main Memory/Memory）
主存储器是计算机系统中的主要存储单元，用于暂时存储正在执行的程序和数据。它直接与 CPU 相连，具有高速、低延迟的特点，通常包括易失性的 **RAM（随机存取存储器）** 和非易失性的 **ROM（只读存储器）**。主存储器为计算机提供高效的数据访问，使得程序能够快速读取和写入所需数据。

辅存储器（Secondary Memory/Internal Storage/Storage）
辅存储器是计算机中的长期数据存储设备，用于保存操作系统、应用程序和用户文件。它具有非易失性，数据在断电后仍能保留。常见的辅存储设备包括 **HDD（硬盘驱动器）** 和 **SSD（固态硬盘）**。辅存储器的容量远大于主存储器，但访问速度相对较慢，主要用于长期存储大量数据。

注意：

如果从体系结构和直接 CPU 访问的角度看，ROM 被视为主存储器，因为它在存储层级中发挥作用（例如，用于启动过程的固件存储）。
如果从非易失性和长期数据存储的角度看，ROM 更接近辅存储器，因为它在断电后可以保留关键数据。

对于其中的主存，基本上可以用下表表示：

```
┌─ 1. 随机存取存储器 (Random Access Memory, RAM)
│   ├─ 1.1 可读写存储器 (Read/Write Memory, Volatile)
│   │   ├─ 1.1.1 静态 RAM (SRAM)
│   │   └─ 1.1.2 动态 RAM (DRAM)
│   └─ 1.2 只读存储器 (Read Only Memory, ROM, Nonvolatile)
│       ├─ 1.2.1 掩模只读存储器 (Mask ROM)
│       ├─ 1.2.2 一次性可编程 (PROM)
│       ├─ 1.2.3 可擦除可编程 (EPROM)
│       ├─ 1.2.4 电可擦除可编程 (EEPROM)
│       └─ 1.2.5 闪存 (Flash ROM)
│
├─ 2. 串行存取存储器 (Serial Access Memory)
│   ├─ 2.1 移位寄存器 (Shift Registers)
│   │   ├─ 2.1.1 串入并出 (SIPO)
│   │   └─ 2.1.2 并入串出 (PISO)
```

```
│  └─ 2.2 队列 (Queues)
│     ├─ 2.2.1 先进先出 (FIFO)
│     └─ 2.2.2 后进先出 (LIFO)
│
└─ 3. 内容寻址存储器 (Content Addressable Memory, CAM)
```

这套层级结构以「**存取方式**」为最顶层划分依据，细分为“随机存取”、“串行存取”和“内容寻址”三大类。随后在“随机存取”分支根据「**是否易失**」来区分读写 RAM 和只读 ROM，并在 ROM 下细分了各种编程/擦写类型；在“串行存取”分支则按照移位寄存器和队列进行展开；最后单列“内容寻址存储器 (CAM)”以强调其独特的高速匹配功能。
在此基础上，进一步补充了“材料”与“原因”的信息，从硬件角度解释了每种存储器的物理结构如何决定其读写特性、易失性、以及成本与容量等方面的差异。通过保持这一树形结构和补充背后的硬件原理，可以更深入且一目了然地了解不同类型存储器的特征与应用场景。

对于每种器件而言，其特性和物理结构分别为：

1. 随机存取存储器 (Random Access Memory, RAM)
1.1 可读写存储器 (Read/Write Memory, Volatile)

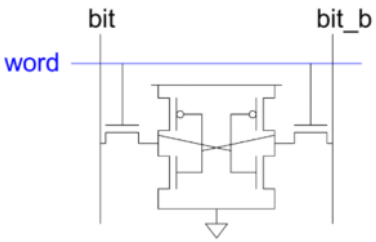
- **易失性（Volatile）**：断电后数据会丢失。
- **读取/写入**：可以随机访问任意地址下的数据，适合临时性、经常变动的数据存储。

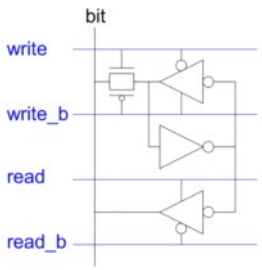
1.1.1 静态 RAM (SRAM)

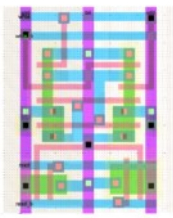
- **数据存储单元**：由触发器 (flip-flop) 构成。
- **刷新需求**：不需要周期性刷新，速度较快。
- **成本/容量**：成本高、集成度相对较低，通常用于 CPU Cache 等高速缓冲场景。

材料与结构

- **材料**：由 CMOS 触发器（通常 4~6 个晶体管组成一个存储单元）构成。
- **原因**：触发器可以在有电的情况下持续保持一个稳定的高或低电平，不会随时间自动衰减，因此无需刷新；但由于一个位需要多个晶体管，导致面积大、成本高、容量有限。





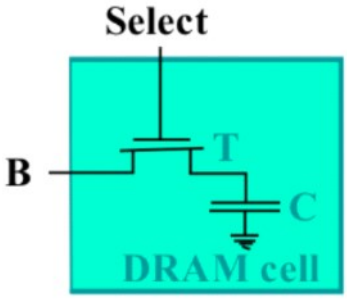


1.1.2 动态 RAM (DRAM)

- **数据存储单元**：使用电容存储电荷。
- **刷新需求**：需要定期刷新以保持数据。这个刷新的特性也让其比较慢。
- **成本/容量**：成本较低、可大规模扩充，是主存（如计算机内存）的主流选择。

材料与结构

- **材料**：每个存储单元由一个晶体管和一个电容组成。
- **原因**：利用电容存储电荷来表示 0 或 1，但电容会随时间漏电，需要定期刷新才能保持数据。优点是一个位只需极少的晶体管与电容，单位面积可容纳更多数据，造价更低，适合大容量应用。



1.2 只读存储器 (Read Only Memory, ROM, Nonvolatile)

- **非易失性（Nonvolatile）**：断电后能保持数据。
- **写入特性**：大多在出厂后或特定条件下才能编程或擦除，写入速度远慢于读取。

1.2.1 掩模只读存储器 (Mask ROM)

- **写入方式**：在芯片制造阶段就将数据固化，后期无法更改。
- **适用场景**：量产固件、数据完全固定的应用。

材料与结构

- **材料**：主要由硅晶体管和固定连接的金属线路构成。
- **原因**：在生产时直接将数据硬编码到晶体管连接中，无法后期修改，数据稳定且成本低，适合大批量生产的固定用途。

1.2.2 一次性可编程 (PROM)

- **写入方式**：可通过编程器写入一次，写后无法改写或擦除。
- **特点**：适合小批量或开发阶段需要一次性烧录的场合。

材料与结构

- **材料**：使用熔丝 (fuses) 或抗熔丝 (antifuses) 技术制成。
- **原因**：在编程时通过强电流熔断熔丝或改变抗熔丝导通性，一旦写入就不可逆，适合一次性永久存储。

1.2.3 可擦除可编程 (EPROM)

- **擦除方式**：紫外线照射可清除数据，再重新编程。
- **特点**：可多次擦写，但擦写步骤相对繁琐。

材料与结构

- **材料**：由浮栅晶体管 (floating gate transistors) 组成。
- **原因**：浮栅中存储的电子可以通过高压编程写入，紫外线则可将电子释放，从而实现擦除并再次写入。

1.2.4 电可擦除可编程 (EEPROM)

- **擦除/写入**：可通过电信号实现擦除与重写，无需紫外线。
- **应用**：适用于需要更新固件或小数据存储的场合；擦写次数有限但较便利。

材料与结构

- **材料**：同样使用浮栅晶体管，但支持电信号擦除。
- **原因**：通过在浮栅上施加一定的电压来增加或移除电子，无需紫外线即可完成擦写，使用更方便。

1.2.5 闪存 (Flash ROM)

- **擦写方式**：与 EEPROM 相似，但可块级（block）或页级（page）快速擦除重写。
- **应用**：广泛用于 U 盘、固态硬盘、移动设备等；读写速度更快、集成度更高。

材料与结构

- **材料**：高级浮栅晶体管阵列，支持块级操作。
- **原因**：相比传统 EEPROM，闪存可一次性擦除整块区域，提高了写入速度和效率，且单位成本更低，适合存储容量需求更大的应用。

2. 串行存取存储器 (Serial Access Memory)

2.1 移位寄存器 (Shift Registers)

- **存取方式**：数据通过时钟脉冲一个接一个地移位，若要访问特定位置的数据，必须先读出之前所有数据。
- **功能**：在数字系统中实现数据的串行/并行转换和缓冲。

2.1.1 串入并出 (SIPO)

- **数据输入**：以串行方式逐位输入。
- **数据输出**：可同时并行输出所有位。

2.1.2 并入串出 (PISO)

- **数据输入**：并行方式一次性将所有位输入。
- **数据输出**：通过时钟脉冲，一个比特一个比特地串行输出。

材料与结构

- **材料**：由一系列触发器 (flip-flops) 或动态逻辑电路构成。
- **原因**：通过级联的触发器使数据随时钟信号逐位移动或输出，决定了其顺序访问特性。

2.2 队列 (Queues)

- **队列机制**：数据按照特定“进出顺序”进行管理，常用于数据缓冲、流量控制等场景。

2.2.1 先进先出 (FIFO)

- **逻辑原则**：先进入的数据先输出，类似排队。
- **应用**：网络路由缓冲、流媒体播放缓冲等。

2.2.2 后进先出 (LIFO)

- **逻辑原则**：后进入的数据先输出，类似堆栈(stack)。
- **应用**：计算机执行栈、表达式求值等。

材料与结构

- **材料**：通常基于 SRAM 或逻辑链表结构来实现数据的存放次序。
- **原因**：通过进出指针（或栈顶指针）等硬件逻辑控制数据的读写顺序，实现 FIFO 或 LIFO 的特性。

3. 内容寻址存储器 (Content Addressable Memory, CAM)

- **存取方式**：根据数据内容（而非地址）进行查找，返回匹配结果所在位置。
- **优势**：能并行地高速匹配数据，适合需要超快速查找或比对的场合。
- **应用**：网络路由器中的地址表、CPU Cache 的标签匹配等。

材料与结构

- **材料**：主要由异或逻辑电路 (XOR logic circuits) 和并行比较器 (parallel comparators) 构成，通常基于 SRAM。
- **原因**：CAM 的核心逻辑可以同时对比所有存储单元中的数据，实现高效率的并行匹配，但硬件结构复杂、功耗高。

优势和劣势

一般而言，（优势）速度越高，（劣势）容量越小，功耗越高，成本越高
对于缓存、主存、存储而言，其这四个方面的关系如下：

存储类型	容量	速度	功耗	成本
缓存 (Cache)	小（几 KB 至几 MB）	非常快	较高	昂贵
主内存 (Main Memory)	中（几 GB 至几十 GB）	较快	较低	较便宜
存储 (Storage)	大（几十 GB 至 TB）	较慢	较低	便宜

下面以SRAM、DRAM、ROM、DHH和SSD为例，分析其在四个指标下的相对性能

1. SRAM (静态随机存取存储器)

标准 **SRAM**

容量 **小容量**：SRAM 单元较大，通常用于缓存（几 KB 到几 MB）。

速度 **非常快**：访问速度极快，适合用作 CPU 缓存。

功耗 **较高**：由于需要更多晶体管，功耗较大。

成本 **昂贵**：每位数据需要更多的晶体管，导致成本较高。

- **优点**：速度非常快，适用于 CPU 缓存（L1、L2、L3 缓存）。
- **缺点**：容量小，功耗高，成本贵，主要用于快速存取的小容量存储。

2. DRAM (动态随机存取存储器)

标准 **DRAM**

容量 **大容量**：容量通常从几 GB 到几十 GB。

速度 **较慢**：比 SRAM 慢，需要周期性刷新数据。

功耗 **较低**：与 SRAM 相比，功耗较低，但仍需刷新。

成本 **较便宜**：成本较低，适合大容量内存。

- **优点**：容量大，成本相对较低，广泛用于主内存（RAM）。
- **缺点**：速度较慢，需定期刷新数据，易失性。

3. ROM (只读存储器)

标准 **ROM**

容量 **小容量**：容量通常较小，用于存储固件。

速度　**较慢**：访问速度慢，写入速度比读取慢。

功耗　**低功耗**：数据在断电后仍能保存，几乎不消耗电力。

成本　**较低**：由于其简单的结构，成本较低。

- **优点**：非易失性，数据保存可靠，成本低。
- **缺点**：容量小，速度较慢，写入和擦除不方便，适用于存储固定的固件或程序。

4. SSD (固态硬盘)

标准　**SSD**

容量　**大容量**：现代 SSD 容量从几百 GB 到几 TB。

速度　**较快**：相较于传统 HDD，SSD 具有更快的读写速度。

功耗　**较低**：SSD 相比 HDD 功耗较低，没有机械部件。

成本　**较高**：虽然成本下降，但相对于 HDD，价格依然较高。

- **优点**：速度快、功耗低、没有机械部件、耐用，适合长期存储。
- **缺点**：相比于 HDD，容量和成本仍有一定劣势（尽管有显著的性价比提升）。

5. HDD (硬盘驱动器)

标准　**HDD**

容量　**非常大**：容量通常从几百 GB 到几 TB，适合大容量存储。

速度　**较慢**：读写速度比 SSD 慢，受到机械部件的限制。

功耗　**较高**：由于有机械部件，功耗较大。

成本　**便宜**：相对于 SSD，HDD 成本低，单位存储成本较便宜。

- **优点**：成本低，容量大，适合大数据存储。
- **缺点**：速度慢，易受物理损坏影响，功耗较高。

综合对比总结：

存储类型	容量	速度	功耗	成本
SRAM	小容量	非常快	较高	昂贵
DRAM	大容量	较慢	较低	较便宜
ROM	小容量	较慢	低功耗	较低
SSD	大容量	较快	较低	较高
HDD	非常大容量	较慢	较高	便宜

- **SRAM**: 用于缓存，速度最快，但容量小且昂贵。
- **DRAM**: 用于主内存，容量大，速度较慢，成本适中。
- **ROM**: 用于固件存储，容量小，速度较慢，但具备非易失性。
- **SSD**: 用作长期存储，速度较快，功耗低，但成本相对较高。
- **HDD**: 用于大容量数据存储，成本低，速度慢，功耗高。

同步逻辑

同步逻辑就是时间离散的含有记忆功能的电路

运行特性

前面提到，由于具有“串行阻断”的特性，同步逻辑整体的运转比较可控，并且具有较强的周期性

可以将其想象为：回路的“时间”和时钟信号所对应的物理时间周期具有一定的区别
一般而言，回路的“时间”处于“冻结”的状态；只有在时钟信号的特定阶段时，回路的时间才会被“激活”
在“时间激活”的阶段，回路运转，发生以下的变化：

1. 检测输入/传递输出
   回路中输出部分的寄存器将上一周期内存储在其内的信号传递给其下游；这一行为对于下游的回路而言相当于检测输入信号发生的改变
   需要注意的是，即使输入为信号的直接写入，很多设备的有效写入也是在相同的“激发阶段”下发生的
   同时，如果输出部分采用的是组合逻辑的话，其运行不依赖于激发时间（也就是其不会被“冻结”，其运算随物理时间持续地发生）
2. 处理状态，进行计算，组合性质输出

状态处理相关的回路根据自身所处的状态以及激发时间传入的信号，对于新状态进行计算，并完成状态的更新；同时对于整个回路而言给出对外的输出（这一部分本身是组合逻辑，不属于时序逻辑；但是由于其输入是时序逻辑，其实际产生变化依赖于这个时间周期）

这个是对于一个寄存器的上下游而言的；

优势与劣势

优点：

1. **易于设计**：同步时序电路采用时钟信号进行控制，设计时可以通过时钟周期来控制所有元件的状态变化，从而简化了电路设计。
2. **可预测性强**：由于所有的状态变化都受到时钟信号的控制，因此系统行为更加可预测，便于调试和测试。
3. **较少的时序问题**：由于时钟信号同步控制，避免了由于不同信号到达时间不一致导致的竞态条件和毛刺现象。

缺点：

1. **时钟频率限制**：同步电路的最大操作频率受限于时钟的周期，时钟频率过高可能会导致时延问题，从而影响电路的稳定性和可靠性。
2. **能耗较大**：由于每个时钟周期都会使得所有触发器同步工作，因此相比异步电路，同步电路的功耗较高，尤其是在时钟频率较高时。

异步逻辑

优点：

1. **速度优势**：在某些特定情况下，异步电路可以比同步电路运行得更快，因为它不依赖时钟信号进行控制，输入信号的变化可以直接影响电路状态。
2. **无时钟限制**：异步电路没有时钟频率的限制，因此在某些应用中，可能实现更高的速度和效率。

缺点：

1. **设计复杂**：由于没有时钟信号的统一控制，设计异步电路时必须仔细处理信号的变化和时序问题，避免出现竞态和不稳定的状态，这使得设计过程更加复杂。
2. **时序问题较多**：输入信号的变化随时可能影响到电路的内存元件，因此需要精确控制信号的传递时间和延迟，以防止毛刺或竞争条件的发生。
3. **难以调试**：由于异步电路的行为可能取决于输入信号的变化时刻，调试时可能较为困难，尤其是在复杂的电路中，难以预测和分析各个元件的状态变化。

同步电路适合于需要明确时序控制和高稳定性的应用，设计相对简单，但其速度和能效受限于时钟频率。
异步电路在特定情况下具有更高的运行速度，但其设计和调试更加复杂，容易出现时序问题，适合用于对时序要求更灵活、对速度有更高要求的应用。

有限状态机

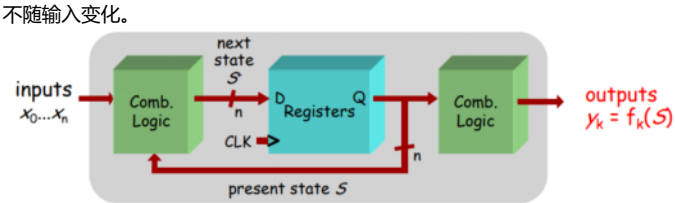
有限状态机（FSM，Finite State Machine）是一种特殊的同步逻辑架构（也就是说，其应该被理解为同步时序逻辑的实现方式其中的一种），用于描述系统在不同状态之间的转换与输出行为。它通过有限个状态和状态之间的转换规则来模拟一个具有内存和反馈机制的动态系统。在数字电路设计中，FSM通常用于控制逻辑的设计，广泛应用于处理器、通信协议、控制系统等多个领域。

结构

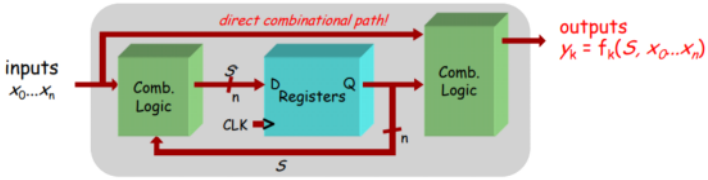
理论结构

有限状态机的结构可以划分为两种主要类型：**Moore机**和**Mealy机**。

1. **Moore机**：在Moore机中，输出仅由当前状态决定，不受输入信号的影响。也就是说，每个状态都有一个固定的输出，不随输入变化。



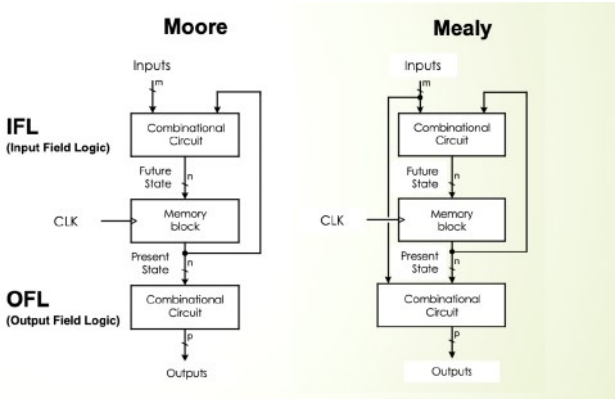
2. **Mealy机**：与Moore机不同，在Mealy机中，输出不仅依赖于当前状态，还依赖于当前的输入信号。换句话说，Mealy机的输出是当前状态和输入的组合函数。

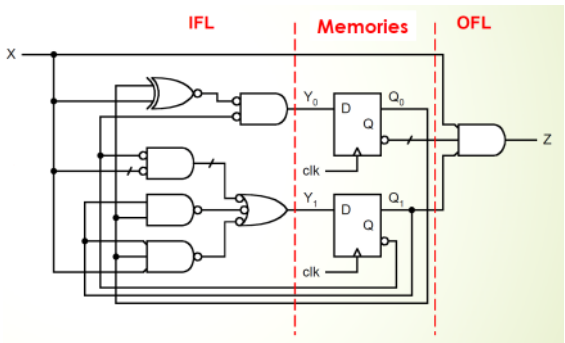


基本组成

有限状态机的基本组成部分包括：

- **输入字段逻辑（IFL，Input Field Logic）**：负责处理外部输入信号，并将这些信号提供给状态机。IFL会将输入信号处理后传递给内存块，以决定下一个状态的转移。
- **内存块（Memory Block）**：存储当前状态的信息。在每个时钟周期结束时，内存块会根据输入信号和当前状态的转换规则更新为下一个状态。内存块的输出即为当前状态。
- **输出字段逻辑（OFL，Output Field Logic）**：根据当前状态和（在Mealy机中）输入信号，决定输出值。OFL将内存块的状态信息与输入信号进行处理，生成最终输出。





状态的含义

状态是指**内存块在当前时刻所存储的信息**，也可以理解为上一时刻已经处理并（被内存块）储存的信息。状态的存在使得信息在有限状态机中可以得到存储和反馈，进而影响系统的行为和状态更新。状态的存储和反馈机制允许信息在电路中长期存在，使得同一个信息能够在不同时间点多次循环影响系统的运行。

在FSM架构下，状态不仅决定了系统的输出，还影响了状态的更新。具体来说，Moore机的输出仅与当前状态相关，输出是离散的；而Mealy机的输出不仅与当前状态相关，还与输入信号实时变化有关。

运转过程

有限状态机的运作过程可以分为以下三个阶段：

1. **时钟信号到达前**：在时钟信号到达之前，输入字段逻辑（IFL）会将外部输入信号处理完毕，并维持一段时间。这段时间至少应大于或等于**设定时间（set-up time）**，以保证输入信号被正确地传递到内存块并使其稳定。
2. **时钟信号激发时**：当时钟信号到来时，内存块（Memory Block）会根据输入信号和当前状态的转换规则，迅速更新状态。更新后的状态信息即成为下一时钟周期的输入，并会影响后续的输出。
3. **时钟信号后**：在时钟信号后，更新后的内存块会将当前状态信息传递给输出字段逻辑（OFL）。OFL会根据当前状态以及输入信号（在Mealy机中）生成输出结果，并将其反馈到系统的输出端。

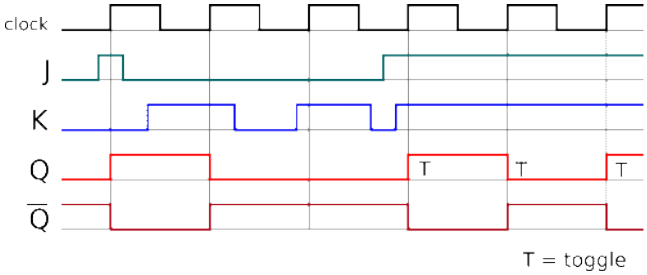
有限状态机（FSM）通过输入字段逻辑、内存块和输出字段逻辑的协同工作，能够实现复杂的状态转移和输出控制。Moore机和Mealy机的主要区别在于输出是否仅由当前状态决定。FSM的运转过程严格依赖于时钟信号，具有高度的时序性和可预测性，适用于需要状态控制和反馈的系统设计。

有限状态机（时序逻辑）图表

在有限状态机（FSM）的设计和分析过程中，时序逻辑常用多种图表来表示系统的行为、状态转移以及输入输出之间的关系。以下是四种广泛应用于FSM的图表：

1. 时序图（Timing Diagram）

时序图用于展示系统信号随时间变化的行为，特别是输入信号、输出信号和时钟信号的波形关系。它通常用于显示系统在不同时间点上各个信号的状态变化，可以帮助设计者理解信号之间的时序关系和同步特性。在FSM设计中，时序图帮助分析时钟信号如何驱动状态变化，并展示状态机如何响应输入信号。



2. 状态转移表（State Transition Table）

状态转移表是FSM的另一种常用表示方式。它以表格形式列出所有可能的状态和输入条件，并明确列出在这些条件下状态的转移规则。表中的每一行代表一个状态的输入和相应的状态转换。状态转移表为FSM的设计者提供了直观的状态变化信息，便于分析状态机在不同输入条件下的行为。

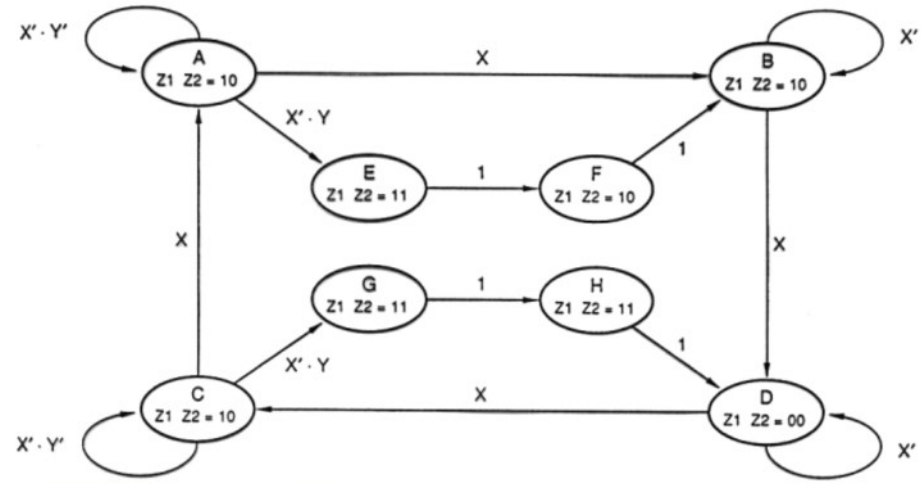
Current State　Input　Next State　State Update　Output

Q_1	Q_0	X	Y_1	Y_0	Q_1^+	Q_0^+	Z
0	0	0	1	0	1	0	0
0	0	1	1	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1
1	1	0	1	1	1	1	0
1	1	1	1	0	1	0	0

3. 状态图（State Diagram）

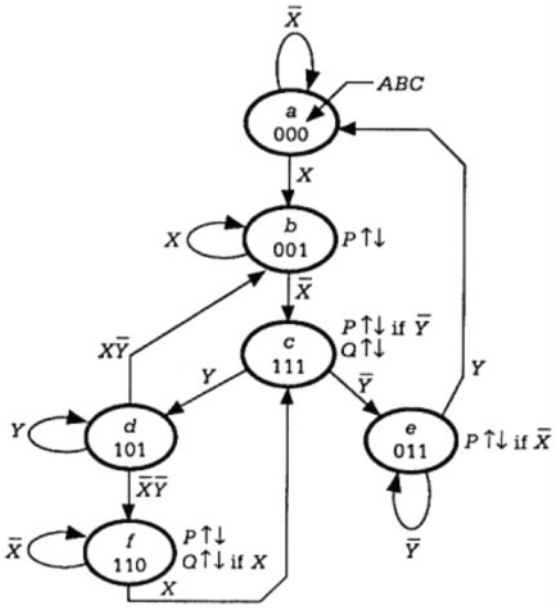
状态图是状态机的图形化表示，通常使用圆形表示状态，箭头表示状态之间的转换。每个箭头标注着导致状态变化的条件或输入信号。状态图清晰地展示了FSM的所有状态及其之间的转换关系，便于直观地理解系统在不同输入条件下的行为流程。它是理解FSM如何响应输入并做出状态转换的核心工具。

Moore Machine 的状态图：



圆圈上方为状态编号，下面为处于此状态下的输出；箭头上方的条件为状态按此箭头方式转移对应的输入条件；和正常的FSM逻辑一样，在CLK触发沿到来时，状态迅速发生转移

Mealy Machine 的状态图：



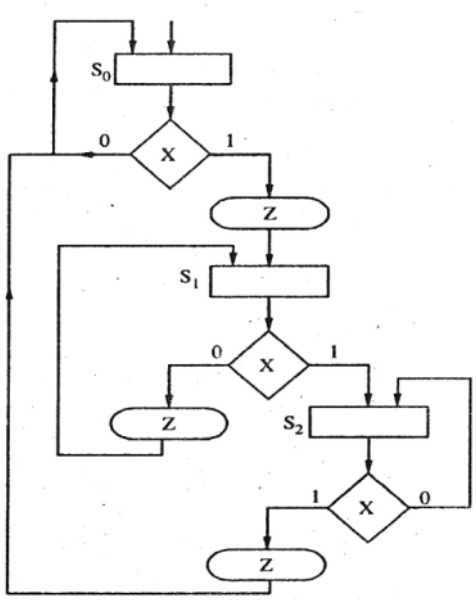
Mealy machine中，每一个状态下的输出还可能会有输入上的条件；如果没有条件则表示到了此状态便执行
这里一个尴尬的点是对于只与output有关而与state无关的量表示起来会比较麻烦

4. 算法状态机图（ASM图，Algorithmic State Machine Chart）

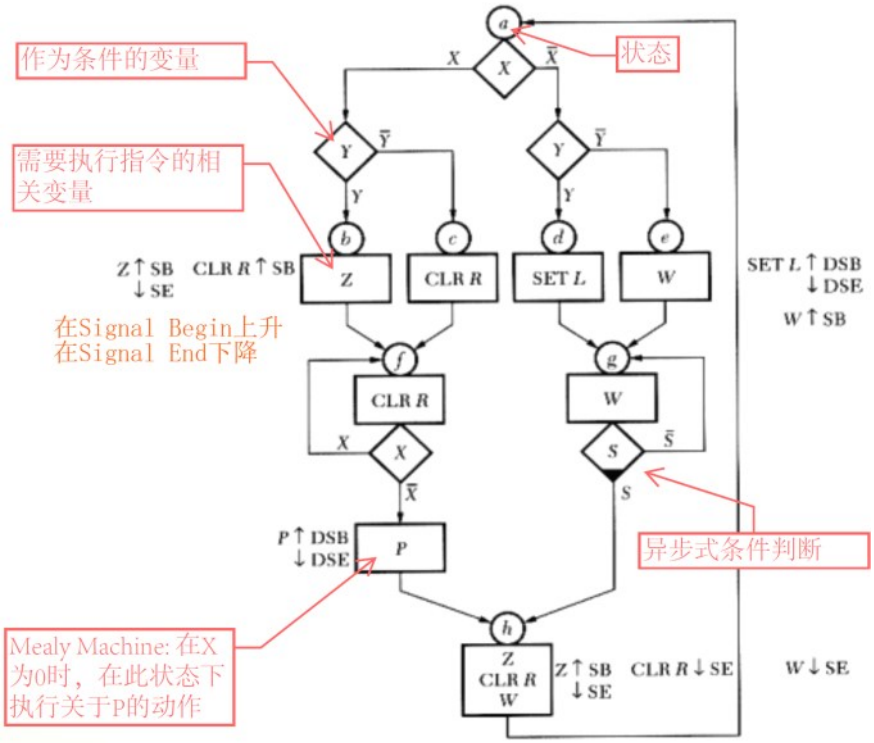
ASM图是一种扩展的状态图，结合了状态图和程序控制逻辑，算流程图的一种(Flow Chart)。它不仅表示状态和状态转移，还通过控制信号、决策和条件判断来描述系统的逻辑流程。ASM图通常包括三个主要部分：状态框（表示状态）、决策框（表示条件判断）、输出框（表示执行的操作）。ASM图常用于复杂的状态机设计，尤其是在需要详细描述控制算法和数据流的情况下。

课程中所使用的ASM图似乎与一般使用的ASM图不一样：一般而言，状态用矩形表示（这里用圆圈表示）；执行指令一般用胶囊形表示（这里用矩形表示）

一般用的：

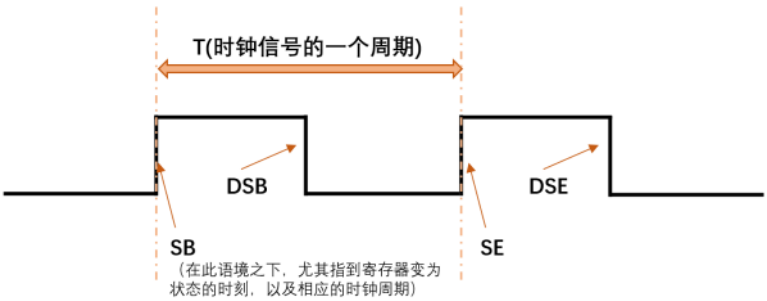


课程中用的：



其中的SB、SE、DSB、DSE遵循以下规律

（分别为signal begin, signal end, delayed signal begin, delayed signal end）



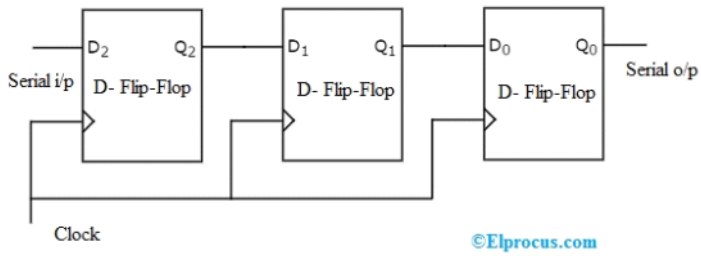
这些图表通过不同的方式帮助设计人员在理解和实现FSM时提供不同层次的抽象视角和信息。

时间分析（同步逻辑）

在同步逻辑电路设计中，时间分析是确保电路稳定、高效运行的关键步骤。通过对时序特性的深入理解和分析，可以优化电路性能，避免时序错误和亚稳态问题。以下将详细阐述同步时序逻辑中的时间分析，包括串联寄存器的时间行为、寄存器的时间需求、逻辑元件的时间特性、回路时间约束条件、关键路径分析、流水线设计以及时钟偏差的处理。

串联寄存器的时间行为

“连续性节点”（并不是一个专业术语，这里用作形象的表示，应该叫pipelining）



在同步时序电路中，寄存器通常以串联方式连接，这种连接方式被称为“连续性节点”。在连续性节点中，相邻寄存器之间的信息传递恰好相差一个时钟周期。这意味着，当一个寄存器在时钟上升沿触发后，其输出将在下一个时钟周期被下一个寄存器捕获和处理。

也就是说，从一个reg的输出到另一个reg的输入之中，一个周期发生一次且恰为一次的信息更新（在同一周期内，前一个reg的更新不会影响下一个reg的更新；前一个reg的更新在下一个周期才会影响下一个reg的更新，更新指 输入写入为输出）

理解这一寄存器时间行为至关重要！

这么做的好处

这种pipeline的方式其实也不是reg的内禀性质，而是我们为同步时序逻辑中reg工作方式选择的一种规范

采用连续性节点的主要好处在于使整个电路的行为规范且可控。具体体现为：

1. **时序一致性**：所有寄存器在同一时钟信号的驱动下同步工作，保证了数据传递的时序一致性。
2. **简化设计**：通过统一的时钟控制，减少了各寄存器之间时序协调的复杂性，设计更加简洁。
3. **提高可靠性**：规范的时序行为减少了竞争条件和毛刺现象的发生，提高了电路的稳定性和可靠性。

寄存器的时间需求

寄存器在时序逻辑中扮演着关键角色，其时间需求（需求也是一种比喻）主要分为输入侧和输出侧两个方面。

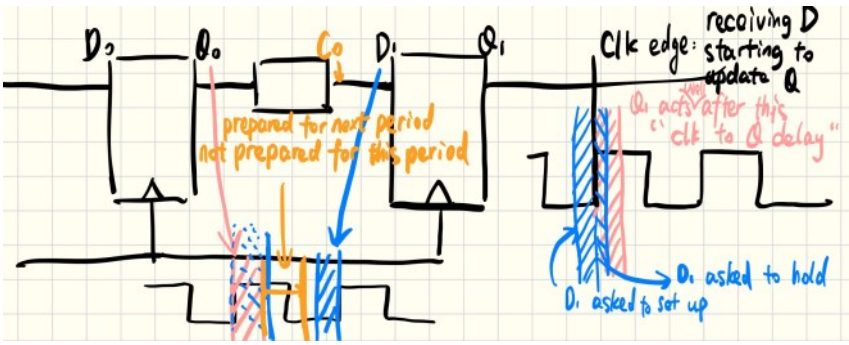
输入侧要求

1. **设定时间（Set-up Time，t_{su}）**：
   - 定义：在时钟信号的触发沿到达之前，输入数据必须在寄存器的输入端稳定并保持至少设定时间，以确保数据能够被正确地捕获。
   - 作用：避免亚稳态问题，确保寄存器在时钟触发时能够稳定地读取到正确的数据。
2. **保持时间（Hold Time，t_{Hold}）**：
   - 定义：在时钟信号的触发沿到达之后，输入数据需要继续保持稳定至少保持时间，以保证寄存器在时钟触发后仍能正确地保持数据。
   - 作用：防止在时钟触发后数据发生变化，导致寄存器捕获到错误的数据。

输出侧要求

寄存器的输出在触发时钟信号后，会经历一个从不稳定到稳定的过程，这一过程可以分为两个阶段：

1. **污染时间（Contamination Clock to Q time，T_{ccq}）**：
   - 定义：寄存器触发时，输出信号开始变化但尚未产生显著变化的时间段。
   - 特点：输出信号开始响应时钟变化，但尚未达到稳定状态。
2. **传播时间（Propagation Clock to Q time，T_{pcq}）**：
   - 定义：输出信号已经产生显著变化，但仍未完全稳定的时间段。
   - 特点：输出信号逐渐趋于稳定，完成从不稳定到稳定的过渡。



逻辑元件时间特性

不同逻辑元件在电路中的计算时间存在差异，这主要是由于以下原因：

1. **工艺差异**：不同类型的逻辑门（如与门、或门、非门等）在制造工艺上有所不同，导致其开关速度和延迟时间不同。
2. **结构复杂度**：逻辑元件内部的结构越复杂，其信号传播路径越长，导致计算时间越长。
3. **负载容量**：元件驱动的负载越大，其响应时间越长。

高电平（HL）与低电平（LH）的非对称性

在实际电路中，逻辑元件的高电平到低电平（HL）转换与低电平到高电平（LH）转换的时间往往是不对称的。这种非对称性主要是由于二极管的特性所导致：

- **二极管导通特性**：在不同电平转换过程中，二极管的导通和关断时间不同，导致信号在转换过程中的延迟不一致。

  （导通）当电信号需要通过二极管从低电平（逻辑0）转换到高电平（逻辑1）时，二极管需要一定的时间来完全导通。这一过程受限于二极管的导通电压（通常为0.7V对硅二极管而言）和电流流动的变化。这导致了从低电平到高电平的转换过程中可能出现较长的延迟。

（关断）当信号从高电平转变为低电平时，二极管需要从导通状态切换到关断状态。二极管在关断过程中会有一定的反向恢复时间，即电流停止流动并达到稳定状态，这个过程也需要时间。这种恢复时间往往导致高电平到低电平转换时出现一定的延迟。

- **电容充放电**：电容的充电和放电速率不同，也会引起高低电平转换时间的不对称。

回路时间约束条件

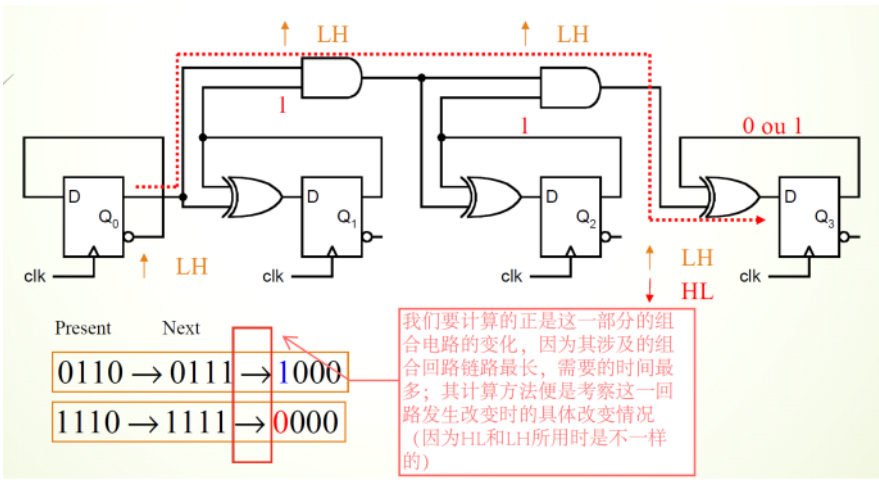
在同步时序电路中，为了确保数据能够在一个时钟周期内正确传递和稳定，必须满足一定的时间约束条件。

1. **主时钟周期约束**：
   T≥Tpcq+Tcomb+TsuT \geq T_{pcq} + T_{comb} + T_{su}
   $$T \geq T_{pcq} + T_{comb} + T_{su}$$
   - **解释**：时钟周期 TT 必须大于或等于寄存器的传播时钟到Q时间 TpcqT_{pcq}、组合逻辑的传播延迟 TcombT_{comb} 以及设定时间 TsuT_{su} 的总和。这样才能确保组合逻辑的计算能够在一个时钟周期内完成，并且结果能够稳定地被下一个寄存器捕获。
2. **保持时间约束**：
   THold≤Tccq+TcombT_{Hold} \leq T_{ccq} + T_{comb}
   $$T_{Hold} \leq T_{ccq} + T_{comb}$$
   - **解释**：保持时间 THoldT_{Hold} 必须小于或等于寄存器的污染时钟到Q时间 TccqT_{ccq} 加上组合逻辑的传播延迟 TcombT_{comb}。这样可以避免上一个寄存器在时钟触发后立即更新输出，干扰当前寄存器的数据保持。

关键路径分析

关键路径是指在时序分析中决定电路最大工作频率的最慢路径（也就是**决定主时钟周期约束的路径**）。在同步时序电路中，关键路径通常由组合逻辑部分的最长延迟决定。

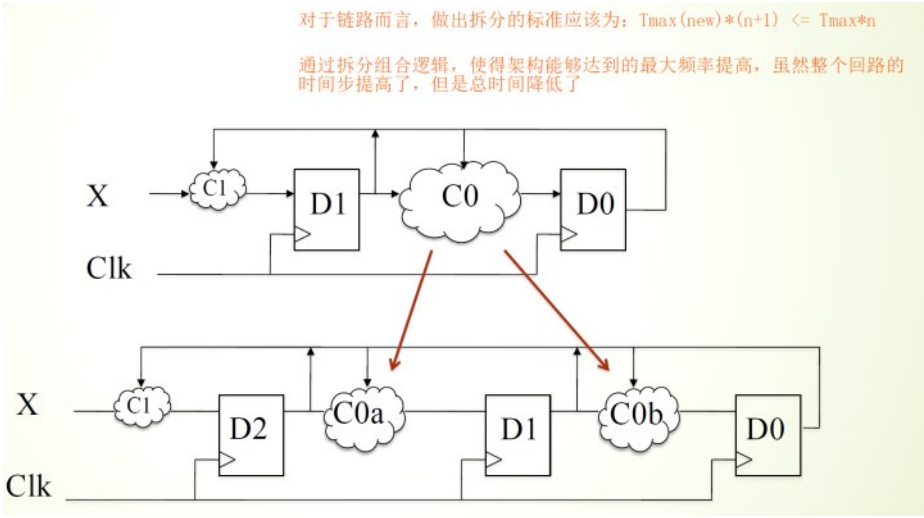
1. **确定关键路径**：
   - 根据主时钟周期约束，关键路径应是使 T_{pcq} + T_{comb} + T_{su} 最大的路径。
     $$T \geq T_{pcq} + T_{comb} + T_{su}$$
   - 通常，组合逻辑部分越长、越复杂，其传播延迟T_{comb} 越大，因此更有可能成为关键路径。
2. **关键路径的特性**：
   - **最长延迟**：关键路径上的组合逻辑延迟最长，限制了整个电路的时钟频率。
   - **复杂逻辑**：关键路径通常涉及多个逻辑门和复杂的逻辑运算，增加了信号传播的时间。



流水线设计

流水线(Pipeline)设计是一种通过将组合逻辑划分为多个阶段，并在各阶段之间添加寄存器，从而提高电路整体工作频率的方法。

1. **优化组合逻辑时间**：
   - 通过在长且复杂的组合逻辑链路中添加寄存器，将其分解为多个较短的组合逻辑段，每段的计算时间减少。
   - 这样可以降低每个时钟周期内组合逻辑的延迟 TcombT_{comb}，从而提高整体的时钟频率。
2. **影响**：
   - **提高频率**：减少组合逻辑延迟后，时钟周期可以缩短，提升电路的运行速度。
   - **增加寄存器数量**：为了实现流水线，需要增加额外的寄存器，增加了电路的面积和复杂度。
   - **延迟增加**：虽然每个时钟周期的延迟减少，但整体运算所需的时钟周期数增加，单次运算的总延迟可能会增加。



时钟偏差

时钟偏差指的是不同寄存器接收到时钟信号的时间存在微小差异，可能导致时序问题。

现象含义

时钟偏差可能导致某些寄存器在时钟触发时比其他寄存器稍早或稍晚触发，进而引起数据传输的不一致和时序错误。例如，偏差过大可能导致数据在设定时间 tsut_{su} 内未能稳定，或在保持时间 tHoldt_{Hold} 内被错误地更新。

修正后的时间约束

为了应对时钟偏差，需要对时间约束进行修正，确保即使存在时钟偏差，电路仍能正常工作。

1. **修正后的主时钟周期约束**：

   $T \geq Tpcq+Tcomb+Tsu+\Delta T$ \geq T_{pcq} + T_{comb} + T_{su} + \Delta t

   - **解释**：在原有的主时钟周期约束基础上，增加时钟偏差 Δt\Delta t 的影响，以确保即使存在偏差，组合逻辑计算仍能在一个时钟周期内完成。
2. **修正后的保持时间约束**：

   $THold \leq Tccq+Tcomb-\Delta tT_{Hold}$ \leq T_{ccq} + T_{comb} - \Delta t

   - **解释**：在保持时间约束中，考虑到时钟偏差会缩短数据稳定的时间，因此需要减去偏差 Δt\Delta t，确保数据在保持时间内不会被错误更新。

总结

时间分析在同步时序逻辑电路设计中至关重要，通过对寄存器的时间行为、时间需求、逻辑元件的时间特性、回路时间约束条件、关键路径分析、流水线设计以及时钟偏差的全面分析，可以有效优化电路性能，提升运行频率，并确保电路的稳定性和可靠性。理解并正确应用这些时间分析方法，是设计高效、稳定的同步时序电路的基础。