第3章 组合逻辑的分析与设计

3.1 逻辑代数基础

逻辑代数是布尔代数向电子工程领域延伸的结果，是布尔代数的一种特例。1847年，英国数学家乔治·布尔(G. Boole)提出了用数学分析方法表示命题陈述的逻辑结构，并成功地将形式逻辑归结为一种代数演算，从而诞生了有名的“布尔代数”。此后于1938年，克劳德·香农(C. E. Shannon)将布尔代数应用于电话继电器的开关电路，提出了“开关代数”。随着电子技术的发展，集成电路逻辑门已经取代了机械触点开关，故“开关代数”这个术语已很少再使用了。为了与“数字逻辑电路设计”这一术语相适应，人们更习惯于把开关代数叫作逻辑代数。

目前，逻辑代数已成为研究数字系统逻辑设计的基础理论。无论何种形式的数字系统，都是由一些基本的逻辑电路组成的。为了解决数字系统分析和设计中的各种具体问题，必须掌握逻辑代数这一重要的数学工具。

本节主要从实用的角度出发，介绍逻辑代数的基本概念、基本公式、基本定理和规则。


视频讲解

3.1.1 逻辑变量及基本逻辑运算

逻辑代数和普通代数一样，也是用字母表示变量的。但逻辑代数是一种二值代数系统，任何逻辑变量的取值只有两种可能性：0或1。在数字系统中，开关的接通与断开、电压的高低、信号的有无、晶体管的导通和截止等两种稳定的物理状态，均可用0和1这两个不同的逻辑值来表示。为了反映一个数字系统中各开关元件之间的关系，可用几种运算关系来描述。逻辑代数中定义了“与”“或”“非”三种基本运算。

1. “与”运算

逻辑问题中，如果决定某一事件发生的多个条件必须同时具备，事件才能发生，则这种因果关系称为“与”逻辑。逻辑代数中，“与”逻辑关系用“与”运算描述。“与”运算又称为逻辑乘(Logic Multiplication)，其运算符号为·，有时也用∧表示。两变量“与”运算关系可表示为

$$F = A \cdot B \quad \text{或者} \quad F = A \wedge B$$

读作“F 等于 A 与 B”。意思是：若 A，B 均为1，则 F 为1；否则，F 为0。

这个逻辑关系可用如图3-1所示的电路来描述，两个开关串联控制同一个灯，仅当两个开关同时闭合时，灯才能亮，否则灯熄灭。假定开关闭合状态用1表示，开关断开状态用0表示，灯亮用1表示，灯灭用0表示，则电路中灯 F 和开关 A、B 之间的关系可用表3-1

表示。

从表 3-1 可得“与”运算法则为

$$0 \cdot 0 = 0 \qquad 1 \cdot 0 = 0$$
$$0 \cdot 1 = 0 \qquad 1 \cdot 1 = 1$$

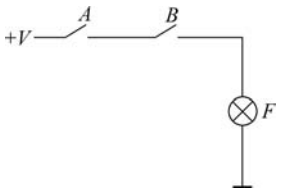


图 3-1　串联开关电路

表 3-1　“与”运算表

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

2. “或”运算

逻辑问题中，如果决定某一事件发生的多个条件中，只要有一个或一个以上条件成立，事件便可发生，则这种因果关系称为“或”逻辑。“或”逻辑关系用“或”运算描述。“或”运算又称逻辑加(Logic Addition)，其运算符号为＋，有时也用 ∨ 表示。两变量“或”运算的关系可表示为

$$F = A + B \quad \text{或者} \quad F = A \vee B$$

读作“F 等于 A 或 B”。意思是：A，B 中只要有一个为 1，则 F 为 1；仅当 A，B 均为 0 时，F 才为 0。这个逻辑关系同样可用图 3-2 所示的电路来描述。其灯 F 与开关 A、B 之间的关系可用表 3-2 表示。由表 3-2 可得“或”运算法则为

$$0+0=0 \qquad 1+0=1$$
$$0+1=1 \qquad 1+1=1$$

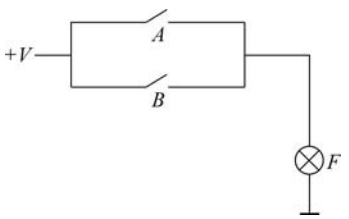


图 3-2　并联开关电路

表 3-2　“或”运算表

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

3. “非”运算

逻辑问题中，如果某一事件的发生取决于条件的否定，即事件与事件发生的条件之间构成矛盾，则这种因果关系称为“非”逻辑。逻辑代数中，“非”逻辑用“非”运算描述。“非”运算也叫求反运算或者逻辑否定(Logic Negation)。其运算符号为－。“非”运算的逻辑关系可表示为

$$F=\overline{A}$$

读作“F 等于 A 非”。意思是：若 A 为0,则 F 为1；反之,若 A 为1,则 F 为0。

这个逻辑关系也可用图3-3电路来描述,其灯 F 与开关 A 的关系可用表3-3表示。

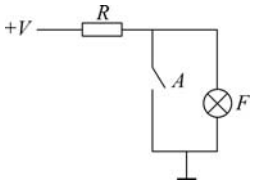


图3-3　开关与灯并联电路

表3-3　“非”运算表

A	F
0	1
1	0

由表3-3可得“非”运算法则为

$$\overline{0}=1 \qquad \overline{1}=0$$

视频讲解

3.1.2　逻辑代数的基本公式、定理与规则

1. 基本公式

1) 交换律

$$A+B=B+A$$
$$A\cdot B=B\cdot A$$

2) 结合律

$$(A+B)+C=A+(B+C)$$
$$(A\cdot B)\cdot C=A\cdot(B\cdot C)$$

3) 分配律

$$A\cdot(B+C)=A\cdot B+A\cdot C$$
$$A+B\cdot C=(A+B)(A+C)$$

4) 0-1律

$$\begin{cases}A+0=A\\A+1=1\end{cases}$$
$$\begin{cases}A\cdot 1=A\\A\cdot 0=0\end{cases}$$

5) 互补律

$$A+\overline{A}=1$$
$$A\cdot\overline{A}=0$$

6) 吸收律

$$\begin{cases}A+AB=A\\A+\overline{A}B=A+B\end{cases}$$
$$\begin{cases}A(A+B)=A\\A(\overline{A}+B)=AB\end{cases}$$

7）重叠律

$$A+A=A$$

$$A \cdot A=A$$

8）对合律

$$\overline{\overline{A}}=A$$

9）反演律

$$\overline{(A+B)}=\overline{A}\ \overline{B}$$

$$\overline{AB}=\overline{A}+\overline{B}$$

10）包含律

$$AB+\overline{A}C+BC=AB+\overline{A}C$$

$$(A+B)(\overline{A}+C)(B+C)=(A+B)(\overline{A}+C)$$

以上几对公式是逻辑代数的基本公式，其中1）～5）可以作为逻辑代数的公理，以此为基础可以推导出逻辑代数的其他公式。

2. 逻辑代数的主要定理


视频讲解

定理1 德·摩根(De Morgan)定理。

(1) $\overline{x_1+x_2+\cdots+x_n}=\overline{x}_1 \cdot \overline{x}_2 \cdot \cdots \cdot \overline{x}_n$

(2) $\overline{x_1 \cdot x_2 \cdot \cdots \cdot x_n}=\overline{x}_1+\overline{x}_2+\cdots+\overline{x}_n$

这就是说，n 个变量的“或”的“非”等于各变量的“非”的“与”；n 个变量的“与”的“非”等于各变量的“非”的“或”。这实际就是反演律的一般形式。

定理2 香农(Shannon)定理。

$$\overline{f(x_1,x_2,\cdots,x_n,0,1,+,\cdot)}=f(\overline{x}_1,\overline{x}_2,\cdots,\overline{x}_n,1,0,\cdot,+)$$

这就是说，任何函数的反函数(或称补函数)，可以通过对该函数的所有变量取反，并将常量1换为0，0换为1，运算符+换为·，·换为+而得到。

香农定理实际上是德·摩根定理的推广，它可用在任何复杂函数中。

例3-1 已知函数 $F=\overline{A}B+A\overline{B}(C+\overline{D})$，求其反函数 $\overline{F}$。

解： $\overline{F}=\overline{\overline{A}B+A\overline{B}(C+\overline{D})}=\overline{\overline{A}B} \cdot \overline{A\overline{B}(C+\overline{D})}$

$=(A+\overline{B}) \cdot (\overline{A\overline{B}}+\overline{C+\overline{D}})=(A+\overline{B})((\overline{A}+B)+\overline{C}D)=(A+\overline{B}) \cdot (\overline{A}+B+\overline{C}D)$

利用香农定理，可以直接写出

$$\overline{F}=(A+\overline{B}) \cdot (\overline{A}+B+\overline{C}D)$$

定理3 展开定理。

(1) $f(x_1,x_2,\cdots,x_i,\cdots,x_n)=x_i f(x_1,x_2,\cdots,1,\cdots,x_n)+\overline{x}_i f(x_1,x_2,\cdots,0,\cdots,x_n)$

(2) $f(x_1,x_2,\cdots,x_i,\cdots,x_n)=[x_i+f(x_1,x_2,\cdots,0,\cdots,x_n)] \cdot [\overline{x}_i+f(x_1,x_2,\cdots,1,\cdots,x_n)]$

这就是说，任何布尔函数都可以对它的某一变量 x_i 展开，或展开成如(1)所示的“与-或”形式，或展开成如(2)所示的“或-与”形式。

由展开定理可得下列两个推理：

推理 1

(1) $x_i f(x_1,x_2,\cdots,x_i,\cdots,x_n)=x_i f(x_1,x_2,\cdots,1,\cdots,x_n)$

(2) $x_i+f(x_1,x_2,\cdots,x_i,\cdots,x_n)=x_i+f(x_1,x_2,\cdots,0,\cdots,x_n)$

推理 2

(1) $\bar{x}_i f(x_1,x_2,\cdots,x_i,\cdots,x_n)=\bar{x}_i f(x_1,x_2,\cdots,0,\cdots,x_n)$

(2) $\bar{x}_i+f(x_1,x_2,\cdots,x_i,\cdots,x_n)=\bar{x}_i+f(x_1,x_2,\cdots,1,\cdots,x_n)$

下面举例说明展开定理的应用。

例 3-2 证明公式 $AB+\bar{A}C+BC=AB+\bar{A}C$(包含律)。

解：用展开定理，等号左边可展开成

$$\begin{aligned}左边&=A(1\cdot B+0\cdot C+BC)+\bar{A}(0\cdot B+1\cdot C+BC)\\&=A(B+BC)+\bar{A}(C+BC)=AB+\bar{A}C=右边\end{aligned}$$

证毕。

例 3-3 将函数 $F=\bar{A}\bar{B}+AC$ 表示成“或-与”形式。

解：由展开定理，可得

$$F=[A+(1\cdot\bar{B}+0\cdot C)]\cdot[\bar{A}+(0\cdot\bar{B}+1\cdot C)]=(A+\bar{B})(\bar{A}+C)$$

视频讲解

3. 逻辑代数的重要规则

逻辑代数有三个重要规则，现分别叙述如下。

1) 代入规则

任何一个含有变量 x 的等式，如果将所有出现 x 的位置，都代之以一个逻辑函数 F，则等式仍然成立。这个规则称为代入规则。

由于任何一个逻辑函数也和任何一个变量一样，只有 0 或 1 两种取值，显然，以上规则是成立的。

例如：已知等式 $\overline{A+B}=\bar{A}\cdot\bar{B}$，函数 $F=B+C$，若用 F 代入此等式中的 B，则有

$$\overline{A+(B+C)}=\bar{A}\cdot\overline{B+C}$$

$$\overline{A+B+C}=\bar{A}\cdot\bar{B}\cdot\bar{C}$$

据此类推可以证明 n 变量的德·摩根定理的成立。

2) 对偶规则

任何一个逻辑函数表达式 F，如果将表达式中所有的+改成·，·改成+，1 改成 0，0 改成 1，而变量保持不变，则可得到一个新的函数表达式 F_d，称 F_d 为 F 的对偶函数。若两个逻辑函数相等，则其对偶式也相等，这一规则称为对偶规则。例如，下列为几个原函数及其对偶函数：

$$F=\bar{A}B+A\bar{B}\bar{C}\qquad F_d=(\bar{A}+B)(A+\bar{B}+\bar{C})$$

$$F=A(\bar{B}+CD)+E\qquad F_d=[A+\bar{B}(C+D)]E$$

$$F=(A+0)(B+C\cdot 1)\qquad F_d=A\cdot 1+B(C+0)$$

$$F=A+\overline{B+\bar{C}+\overline{D+\bar{E}}}\qquad F_d=A\cdot\overline{B\cdot\bar{C}\cdot\overline{D\cdot\bar{E}}}$$

需要注意的是，在运用对偶规则求对偶函数时，必须按照先“与”后“或”的顺序，否则容易写错，例如 $F=\bar{A}B+A\bar{B}\bar{C}$，若求出对偶函数 $F_d=\bar{A}+B\cdot A+\bar{B}+\bar{C}$，这是错误的。因此，

要特别注意原来函数中的“与”项，当这些“与”项变为“或”项时，应加括号。

从上面这些例子可以看出，如果 F 的对偶函数为 F_d，则 F_d 的对偶函数就是 F。也就是说，F 和 F_d 互为对偶函数，即 $(F_d)_d=F$。

上面叙述的 10 个基本公式都是成对出现的，实际上是互为对偶的等式。由此证明一个规则：如果两个函数的表达式相等，则它们的对偶函数也相等，即如果函数 $F=G$，则其对偶函数 $F_d=G_d$。

3）反演规则

任何一个逻辑函数表达式 F，如果将表达式中的所有＋改成·，·改成＋，1 改成 0，0 改成 1，原变量改成反变量，反变量改成原变量，则可得函数 F 的反函数（或称补函数）$\overline{F}$。这个规则称为反演规则。

实际上，反演规则就是香农定理。运用反演规则可以很方便地求一个函数的补函数。例如，下面为几个原函数及其补函数：

$$F=\overline{A}B+A\overline{B}\overline{C} \qquad \overline{F}=(A+\overline{B})(\overline{A}+B+C)$$

$$F=A(\overline{B}+CD)+E \qquad \overline{F}=[\overline{A}+B(\overline{C}+\overline{D})]\overline{E}$$

$$F=(A+0)(B+C\cdot 1) \qquad \overline{F}=\overline{A}\cdot 1+\overline{B}(\overline{C}+0)$$

$$F=A+\overline{B+\overline{C}+\overline{D+\overline{E}}} \qquad \overline{F}=\overline{A}\cdot\overline{\overline{B}\cdot C\cdot\overline{\overline{D}\cdot E}}$$

与求对偶函数一样，求补函数需要注意的是，在运用反演规则时，必须按照先“与”后“或”的顺序进行变换。因此，特别要注意原来函数中的“与”项，当这些“与”项变换为“或”项时，应加括号。

把上述补函数的例子与前面对偶函数的例子对照一下，可以看出，补函数和对偶函数之间在形式上只差变量的“非”。因此，若已求得一函数的对偶函数，只要将所有变量取反便得该函数的补函数；反之亦然。

4. 几个常用公式

$$A\oplus B=A\overline{B}+\overline{A}B\ （异或）$$

$$A\odot B=AB+\overline{A}\overline{B}\ （同或）$$

$$A\oplus B=\overline{A\odot B}$$

$$A\oplus 1=\overline{A} \qquad A\oplus 0=A$$

$$A\odot 1=A \qquad A\odot 0=\overline{A}$$

$$AB+A\overline{B}=A \qquad (A+B)(A+\overline{B})=A$$

3.1.3 逻辑函数及其表达式


视频讲解

1. 逻辑函数的表示法

常用的逻辑函数表示方法有三种：逻辑表达式、真值表、卡诺图。

1）逻辑表达式

逻辑表达式是由逻辑变量和“与”“或”“非”三种运算符所构成的式子。例如：

$$F=f(A,B)=\overline{A}\cdot B+A\cdot\overline{B}$$

为一个由两个变量（A 和 B）进行逻辑运算构成的逻辑表达式，它描述了一个两变量的逻辑函数 F。函数 F 和变量 A，B 的关系是：当变量 A 和 B 取值不同时，函数 F 的值为 1；否

则,函数 F 的值为0。关于逻辑表达式的书写,为了简便起见,可按下述规则省略某些括号或运算符号:

① 进行"非"运算可不加括号,如 $\overline{A}$,$\overline{A+B}$ 等。

② "与"运算符一般可省略,如 $A \cdot B$ 可写成 AB。

③ 在一个表达式中,如果既有"与"运算又有"或"运算,则按先"与"后"或"的规则省去括号。如$(A \cdot B)+(C \cdot D)$可写为 $AB+CD$,但$(A+B) \cdot (C+D)$不能省略括号而写成 $A+B \cdot C+D$。

④ "与"运算和"或"运算均满足结合律,因此,$(A+B)+C$ 或者 $A+(B+C)$可用 $A+B+C$ 代替;$(AB)C$ 或者 $A(BC)$可用 ABC 代替。

2) 真值表

用真值表描述逻辑函数的方法是一种表格表示法。由于一个逻辑变量只有0和1两种可能的取值,故 n 个逻辑变量共有 2^n 种可能的取值组合。任何逻辑函数总是和若干逻辑变量相关,有限的变量个数使得变量取值组合的总数必然是有限的,从而,能够用穷举的方法来描述逻辑函数的功能。为了清晰,常用的方法是对一个函数求出所有输入变量取值下的函数值并用表格形式记录下来,这种表格称为真值表。换而言之,真值表是一种由逻辑变量的所有可能取值组合及其对应的逻辑函数值所构成的表格。

真值表由两部分组成,左边一栏列出变量的所有取值组合,为了不发生遗漏,通常各变量取值组合按二进制数码顺序给出;右边一栏为逻辑函数值。例如,函数 $F=A \cdot \overline{B}+\overline{A} \cdot C$ 的真值表如表3-4所示。事实上,前面介绍三种基本运算时所列出的表3-1、表3-2、表3-3分别是"与""或""非"三种逻辑运算的真值表。

表3-4 函数 $F=A \cdot \overline{B}+\overline{A} \cdot C$ 的真值表

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

真值表是一种十分有用的逻辑工具。在逻辑问题的分析和设计中,会经常用到这一工具。

3) 卡诺图

卡诺图是由表示逻辑变量的所有可能组合的小方格所构成的平面图,它是一种用图描述逻辑函数的方法,这种方法在逻辑函数化简中十分有用,将在后面结合函数化简问题进行详细介绍。

上述表示逻辑函数的三种方法各有特点,它们各适用于不同的场合。但针对某个具体问题而言,它们仅仅是同一问题的不同描述形式。在它们之间存在内在的联系,可以很方便地相互变换。

2. 逻辑函数的基本形式

根据展开定理,任何一个 n 变量函数总可以展开成"与-或"形式,或者"或-与"形式。其

中“与-或”形式又叫“积之和”形式，而“或-与”形式又叫“和之积”形式。

如 $F(A,B,C)=\overline{A}+B\overline{C}+A\overline{B}C$ 就称为“积之和”形式。

而 $F(A,B,C,D)=(A+B)(C+\overline{D})(\overline{A}+B+C)$ 称为“和之积”形式。

但 $F(A,B,C,D)=(AC+B)CD+(AC+B)\overline{D}$ 既不是“积之和”也不是“和之积”形式，但它可以转换成两种基本形式。

一个函数可以有多种表示形式，但有两种统一的标准形式。

1）标准积之和


视频讲解

所谓标准积，是指函数的积项包含了函数的全部变量，其中每个变量都以原变量或反变量的形式出现，且仅出现一次。标准积项通常称为最小项。

一个函数可以用最小项之和的形式来表示，称为函数的“标准积之和”形式。例如，一个三变量函数为

$$F(A,B,C)=\overline{A}B\overline{C}+\overline{A}BC+A\overline{B}\overline{C}+ABC$$

它由 4 个最小项组成，这是函数“标准积之和”的形式。

由最小项的定义可知，三个变量最多可组成 8 个最小项：$\overline{A}\overline{B}\overline{C}$，$\overline{A}\overline{B}C$，$\overline{A}B\overline{C}$，$\overline{A}BC$，$A\overline{B}\overline{C}$，$A\overline{B}C$，$AB\overline{C}$ 和 ABC。为了叙述和书写方便，通常用 m_i 表示最小项，其下标 i 是这样确定的：把最小项中的原变量记为 1，反变量记为 0，当变量顺序确定后，可以按顺序排列成一个二进制数。那么，与这个二进制数相对应的十进制数就是最小项的下标 i。表 3-5 列出了三个变量的全部最小项。

表 3-5　三变量的所有最小项和最大项

A	B	C	最小项 m_i	最大项 M_i
0	0	0	$m_0=\overline{A}\overline{B}\overline{C}$	$M_0=A+B+C$
0	0	1	$m_1=\overline{A}\overline{B}C$	$M_1=A+B+\overline{C}$
0	1	0	$m_2=\overline{A}B\overline{C}$	$M_2=A+\overline{B}+C$
0	1	1	$m_3=\overline{A}BC$	$M_3=A+\overline{B}+\overline{C}$
1	0	0	$m_4=A\overline{B}\overline{C}$	$M_4=\overline{A}+B+C$
1	0	1	$m_5=A\overline{B}C$	$M_5=\overline{A}+B+\overline{C}$
1	1	0	$m_6=AB\overline{C}$	$M_6=\overline{A}+\overline{B}+C$
1	1	1	$m_7=ABC$	$M_7=\overline{A}+\overline{B}+\overline{C}$

因此，上述函数 $F(A,B,C)$ 可以写成

$$\begin{aligned}F(A,B,C)&=\overline{A}B\overline{C}+\overline{A}BC+A\overline{B}\overline{C}+ABC\\&=m_2+m_3+m_4+m_7=\sum m(2,3,4,7)\end{aligned}$$

其中，符号 $\sum$ 表示各最小项求“或”，括号内的十进制数字表示各最小项的下标。

最小项有下列三个主要性质：

① 对于任意一个最小项，只有一组变量取值使其值为 1。

② 任意两个不同的最小项之积必为 0，即

$$m_i\cdot m_j=0\quad(i\neq j)$$

③ n 变量的所有 2^n 个最小项之和必为 1，即

$$\sum_{i=0}^{2^n-1} m_i = 1$$

用展开定理可以证明,任一个 n 变量的函数都有一个且仅有一个最小项表达式,即"标准积之和"形式。下面介绍求函数的"标准积之和"的两种常用的方法。

一种是代数演算法,即通过反复使用公式 $x+\overline{x}=1$ 和 $x(y+z)=xy+xz$ 而求得"标准积之和"的方法。例如,设 $F(A,B,C)=\overline{A}B+A\overline{B}\overline{C}+BC$,则得

$$\begin{aligned}F(A,B,C)&=\overline{A}B(C+\overline{C})+A\overline{B}\overline{C}+BC(A+\overline{A})\\&=\overline{A}BC+\overline{A}B\overline{C}+A\overline{B}\overline{C}+ABC+\overline{A}BC\\&=\overline{A}B\overline{C}+\overline{A}BC+A\overline{B}\overline{C}+ABC=m_2+m_3+m_4+m_7\\&=\sum m(2,3,4,7)\end{aligned}$$

另一种是列表法,即列出函数的真值表,使函数取值为 1 的那些最小项相加,就构成了函数的"标准积之和"的形式。例如,函数 $F(A,B,C)=\overline{A}B+A\overline{B}\overline{C}+BC$ 的真值表列于表 3-6,根据真值表可以很方便地写出函数的表达式为

$$F(A,B,C)=m_2+m_3+m_4+m_7=\sum m(2,3,4,7)$$

式中,m_2,m_3,m_4 和 m_7 对应于真值表中使函数取值为 1 的那些最小项。

表 3-6　函数的真值表与最小项

A	B	C	$F(A,B,C)$	最　小　项
0	0	0	0	—
0	0	1	0	—
0	1	0	1	$m_2=\overline{A}B\overline{C}$
0	1	1	1	$m_3=\overline{A}BC$
1	0	0	1	$m_4=A\overline{B}\overline{C}$
1	0	1	0	—
1	1	0	0	—
1	1	1	1	$m_7=ABC$

视频讲解

2) 标准和之积

所谓标准和是指函数的和项包含了全部变量,其中每个变量都以原变量或反变量的形式出现,且仅出现一次。标准和项通常又称最大项。

一个函数可以用最大项之积的形式表示,把这种形式称为函数的"标准和之积"式。例如,一个三变量函数为

$$F(A,B,C)=(A+B+C)(A+B+\overline{C})(\overline{A}+B+\overline{C})(\overline{A}+\overline{B}+C)$$

它由 4 个最大项组成,这就是函数的"标准和之积"形式。

同样,三个变量最多可组成 8 个最大项,如表 3-5 所示。通常,最大项用 M_i 来表示,其下标 i 是这样确定的:当最大项的各变量按一定次序排好后,把其中的原变量记为 0,反变量记为 1,便得到一个二进制数,与该二进制数相应的十进制数就是最大项的下标 i。

这样,上述函数 $F(A,B,C)$可以写成

$$\begin{aligned}F(A,B,C)&=(A+B+C)(A+B+\overline{C})(\overline{A}+B+\overline{C})(\overline{A}+\overline{B}+C)\\&=M_0M_1M_5M_6=\prod M(0,1,5,6)\end{aligned}$$

其中，符号 $\prod$ 表示各最大项相与，括号内的十进制数表示各最大项的下标。

最大项具有下列三个主要性质：

① 对于任意一个最大项，只有一组变量取值可使其值为 0。

② 任意两个不同的最大项之和必为 1，即

$$M_i + M_j = 1 \quad (i \neq j)$$

③ n 变量的所有 2^n 个最大项之积必为 0，即

$$\prod_{i=0}^{2^n-1} M_i = 0$$

同样地用展开定理可以证明，任何 n 变量的函数都有一个且仅有一个最大项表达式，即“标准和之积”形式。求函数的“标准和之积”的方法也有两种方法。

一是代数演算法，即通过反复地使用公式 $x \cdot \bar{x} = 0$ 和 $x + yz = (x + y)(x + z)$ 而求得“标准和之积”的方法。例如：

$$\begin{aligned}
F &= \bar{A}B + A\bar{B}\bar{C} + BC = (\bar{A}B + A)(\bar{A}B + \bar{B})(\bar{A}B + \bar{C}) + BC \\
&= (\bar{A} + A)(B + A)(\bar{A} + \bar{B})(B + \bar{B})(\bar{A} + \bar{C})(B + \bar{C}) + BC \\
&= (A + B)(\bar{A} + \bar{B})(\bar{A} + \bar{C})(B + \bar{C}) + BC \\
&= ((A + B)(\bar{A} + \bar{B})(\bar{A} + \bar{C})(B + \bar{C}) + B)((A + B)(\bar{A} + \bar{B})(\bar{A} + \bar{C})(B + \bar{C}) + C) \\
&= ((A + B + B)(\bar{A} + \bar{B} + B)(\bar{A} + \bar{C} + B)(B + \bar{C} + B)) \cdot \\
&\quad ((A + B + C)(\bar{A} + \bar{B} + C)(\bar{A} + \bar{C} + C)(B + \bar{C} + C)) \\
&= (A + B)(\bar{A} + B + \bar{C})(B + \bar{C})(A + B + C)(\bar{A} + \bar{B} + C)
\end{aligned}$$

由于表达式中第一项缺少变量 C，所以要加上 $C \cdot \bar{C}$；第三项缺少变量 A，所以要加 $A \cdot \bar{A}$，即

$$\begin{aligned}
F &= (A + B + C \cdot \bar{C})(\bar{A} + B + \bar{C})(A \cdot \bar{A} + B + \bar{C})(A + B + C)(\bar{A} + \bar{B} + C) \\
&= (A + B + C)(A + B + \bar{C})(\bar{A} + B + \bar{C})(\bar{A} + \bar{B} + C) \\
&= M_0 M_1 M_5 M_6 = \prod M(0,1,5,6)
\end{aligned}$$

可见，用这种方法是比较麻烦的。

二是采用列表法，即列出函数的真值表，那些使函数取值为 0 的最大项，就构成了函数的“标准和之积”形式。例如，上述函数的真值表见表 3-7，根据真值表可以很方便地写出表达式为

$$F(A,B,C) = M_0 M_1 M_5 M_6 = \prod M(0,1,5,6)$$

表 3-7　函数的真值表与最大项

A	B	C	$F(A,B,C)$	最　大　项
0	0	0	0	$M_0 = A + B + C$
0	0	1	0	$M_1 = A + B + \bar{C}$
0	1	0	1	—
0	1	1	1	—
1	0	0	1	—
1	0	1	0	$M_5 = \bar{A} + B + \bar{C}$
1	1	0	0	$M_6 = \bar{A} + \bar{B} + C$
1	1	1	1	—

比较表3-6和表3-7,可以得出两点结论:

① 同一个函数既可以表示成"标准积之和"的形式,又可表示成"标准和之积"的形式。对于本例,有

$$\begin{aligned}F(A,B,C)&=\overline{A}B+A\overline{B}\,\overline{C}+BC\\&=\sum m(2,3,4,7)\\&=\prod M(0,1,5,6)\end{aligned}$$

② 同一函数的最大项与最小项是互斥的,即如果真值表中的某一行作为函数的最小项,那么它就不可能是同一函数的最大项;反之亦然。一般有 $m_i=\overline{M}_i$。换句话说,一个布尔函数的最小项的集合与它的最大项的集合互为补集。因此,若已知一布尔函数的"标准积之和"形式,就可以很容易写出该函数的"标准和之积"形式。例如,已知函数的"标准积之和"的形式为

$$F(A,B,C)=\sum m(1,3,4,6,7)$$

则该函数的"标准和之积"的形式为

$$F(A,B,C)=\prod M(0,2,5)$$

3.2 逻辑函数的化简

同一个逻辑函数可以有多种表示形式。一种形式的函数表达式对应一种逻辑电路,尽管它们的形式不同,但其逻辑功能是相同的。函数表达式越简单,则逻辑电路越简单,因此如何使函数的表达式最简单,即函数的化简,成为逻辑设计的一个重要问题。

在各种不同形式的表达式中,"与-或"表达式是最基本的,其他形式都可由它变换而得。

例如

$$\begin{aligned}F(A,B,C)&=\overline{A}\,\overline{B}+AC && \text{"与-或"形式}\\&=(\overline{A}+C)(A+\overline{B}) && \text{"或-与"形式}\\&=\overline{\overline{A}\,\overline{B}\cdot\overline{AC}} && \text{"与-非"形式}\\&=\overline{\overline{A}+C}+\overline{A+\overline{B}} && \text{"或-非"形式}\\&=\overline{A\overline{C}+\overline{A}B} && \text{"与或非"形式}\end{aligned}$$

因此,下面将从"与或"表达式出发来讨论函数的化简方法。

什么是函数的最简"与或"式呢?一个最简"与-或"式应同时满足如下两个条件:

(1) 该式中的与项最少;

(2) 该式中每个与项的变量最少。

这样,用逻辑门来实现布尔函数时,所需的与门数目最少,而且每个与门的输入端数目也最少。

视频讲解

3.2.1 代数化简法

代数化简法,就是运用逻辑代数的基本公式、定理和规则来化简逻辑函数。这种方法没

有固定的步骤，主要是凭对逻辑代数的公式、定理和规则的熟练运用程度。但还是有一些适用于大多数情况的方法，下面介绍几种常用方法。

1. 并项法

利用公式 $AB+A\overline{B}=A$，将两个“与”项合并成一个“与”项，合并后消去一个变量。例如：

$$A\overline{B}C+A\overline{B}\,\overline{C}=A\overline{B}$$

$$AB\overline{C}+A\overline{B\overline{C}}=A$$

2. 吸收法

利用公式 $A+AB=A$，消去多余的项。例如：

$$\overline{B}+A\overline{B}D=\overline{B}$$

$$\overline{A}B+\overline{A}BCD(E+F)=\overline{A}B$$

3. 消去法

利用公式 $A+\overline{A}B=A+B$，消去多余变量。例如：

$$AB+\overline{A}C+\overline{B}C=AB+(\overline{A}+\overline{B})C=AB+\overline{AB}C=AB+C$$

$$\overline{A}+AC+\overline{C}D=\overline{A}+C+\overline{C}D=\overline{A}+C+D$$

4. 配项法

利用公式 $A\cdot 1=A$ 及 $A+\overline{A}=1$ 或 $A\cdot\overline{A}=0$ 或利用包含律加上一个冗余项，然后再利用并项、吸收、消去等方法进行化简。例如：

$$\begin{aligned}A\overline{B}+B\overline{C}+\overline{B}C+\overline{A}B&=A\overline{B}+B\overline{C}+(A+\overline{A})\overline{B}C+\overline{A}B(C+\overline{C})\\&=A\overline{B}+B\overline{C}+A\overline{B}C+\overline{A}\,\overline{B}C+\overline{A}BC+\overline{A}B\overline{C}\\&=A\overline{B}+B\overline{C}+\overline{A}C\end{aligned}$$

上面介绍的是几种常用的方法，举出的例子都比较简单。而实际中遇到的逻辑函数往往比较复杂，化简时应灵活使用所学的公理、定理及规则，综合运用各种方法。下面再举几个化简实例。


视频讲解

例 3-4 化简 $F=A\overline{C}+ABC+AC\overline{D}+CD$。

解：

$$\begin{aligned}F&=A\overline{C}+ABC+AC\overline{D}+CD\\&=A(\overline{C}+BC)+C(A\overline{D}+D)&&\text{（分配律）}\\&=A(\overline{C}+B)+C(A+D)&&\text{（消去法）}\\&=A\overline{C}+AB+AC+CD&&\text{（分配律）}\\&=A(\overline{C}+C)+AB+CD&&\text{（分配律）}\\&=A+AB+CD&&\text{（互补律）}\\&=A+CD&&\text{（吸收法）}\end{aligned}$$

例 3-5 化简 $F=AD+A\overline{D}+AB+\overline{A}C+BD+ACEF+\overline{B}EF+DEFG$。

解：

$$\begin{aligned}F&=AD+A\overline{D}+AB+\overline{A}C+BD+ACEF+\overline{B}EF+DEFG\\&=A+AB+\overline{A}C+BD+ACEF+\overline{B}EF+DEFG&&\text{（并项法）}\\&=A+\overline{A}C+BD+\overline{B}EF+DEFG&&\text{（吸收法）}\\&=A+C+BD+\overline{B}EF+DEFG&&\text{（消去法）}\\&=A+C+BD+\overline{B}EF&&\text{（包含律）}\end{aligned}$$

例 3-6 化简或与表达式 $F=(A+B)(A+\overline{B})(B+C)(B+C+D)$。

解：可以利用对偶规则，先求出 F 的对偶式：

$$F_d=AB+A\overline{B}+BC+BCD$$

然后利用与或式的化简方法化简得

$$F_d=A+BC$$

最后再对 F_d 求对偶式，则得

$$F=(F_d)_d=A(B+C)=AB+AC$$

从前面的介绍可以看出，代数化简法的优点是不受变量数目的约束，当对公理、定理和规则十分熟练时化简比较方便；其缺点是没有一定的规律和步骤，技巧性很强，而且在很多情况下难以判断化简结果是否最简。因此，这种方法有较大的局限性。

视频讲解

3.2.2 卡诺图化简法

卡诺图化简法是将逻辑函数用卡诺图来表示，在卡诺图上进行函数化简的方法。这是一种简单、直观的方法，并有一定的步骤可依，能确定所简化的函数是否为最简形式。在以后的章节里经常会用到它。

1. 卡诺图的构成

卡诺图是真值表的图形化。把真值表中的最小项重新排列成矩阵形式，并且使矩阵的横向和纵向的逻辑变量的取值按 Gray 码的顺序排列，这样构成的图形就是卡诺图。在卡诺图上任意两个相邻的最小项在图上是相邻的。图 3-4 表示了二变量、三变量和四变量的卡诺图的构成。

B \ A	0	1
0	0	2
1	1	3

(a) 二变量

C \ AB	00	01	11	10
0	0	2	6	4
1	1	3	7	5

(b) 三变量

CD \ AB	00	01	11	10
00	0	4	12	8
01	1	5	13	9
11	3	7	15	11
10	2	6	14	10

(c) 四变量

图 3-4 卡诺图的构成

从图 3-4 中可看出，不仅图上任意两个相邻的最小项是相邻的(只有一位变量值不一样)，而且图中同一行两端的最小项是相邻的，同一列两端的最小项也是相邻的。二变量的最小项有两个最小项与其相邻；三变量的最小项有三个最小项与其相邻；四变量的最小项有四个最小项与其相邻。

五变量的卡诺图可由两幅四变量卡诺图组成，如图 3-5 所示。两幅四变量卡诺图的同一位置的两个最小项也是相邻的。如最小项 3 与最小项 1、2、7、11 及 19 都是相邻的。

DE \ ABC	000	001	011	010
00	0	4	12	8
01	1	5	13	9
11	3	7	15	11
10	2	6	14	10

100	101	111	110
16	20	28	24
17	21	29	25
19	23	31	27
18	22	30	26

图 3-5 五变量卡诺图

六变量卡诺图可由 4 幅四变量卡诺图组成，如图 3-6 所示。在考虑相邻关系时，除要注意左右两幅四变量卡诺图的重合关系，还要注意上下两幅四变量卡诺图的重合关系。如最小项 3 除与最小项 1、2、7、11、19 相邻外，还与最小项 35 相邻。多于 6 个变量时，卡诺图显得很复杂，已失去了它的优越性，一般不采用。

AEF \ BCD	000	001	011	010	100	101	111	110
000	0	4	12	8	16	20	28	24
001	1	5	13	9	17	21	29	25
011	3	7	15	11	19	23	31	27
010	2	6	14	10	18	22	30	26
100	32	36	44	40	48	52	60	56
101	33	37	45	41	49	53	61	57
111	35	39	47	43	51	55	63	59
110	34	38	46	42	50	54	62	58

图 3-6　六变量卡诺图

2. 逻辑函数在卡诺图上的表示

从卡诺图的构成可看出，卡诺图只是把真值表图形化了，它们之间有一定的对应关系。如果逻辑函数是以真值表的形式或者以“标准积之和”的形式给出的，只要在卡诺图上找出那些与给定逻辑函数的最小项相对应的方格，并标以 1，就能得到该函数的卡诺图。例如，三变量函数

$$F(A,B,C)=\sum m(2,3,5,7)$$

其卡诺图如图 3-7 所示。

如果逻辑函数是“与或”表达式，则要将各“与项”分别标在卡诺图上。例如，给定函数的表达式为

$$F(A,B,C)=AB+A\overline{C}$$

只要在 AB 为 11 的列标以 1，并在 A 为 1、C 为 0 的对应方格中标上 1，便可得到如图 3-8 所示的卡诺图。

C \ AB	00	01	11	10
0	0	1	0	0
1	0	1	1	1

图 3-7　$F(A,B,C)=\sum m(2,3,5,7)$ 的卡诺图

C \ AB	00	01	11	10
0	0	0	1	1
1	0	0	1	0

图 3-8　$F(A,B,C)=AB+A\overline{C}$ 的卡诺图

视频讲解

3. 卡诺图的性质

卡诺图化简的原理是基于相邻最小项合并的性质的，也就是卡诺图的以下性质。

性质 1：卡诺图上任何两个（2^1 个）标 1 的相邻最小项，可合并为一项，并消去一个变量。

性质 2：卡诺图上任何四个（2^2 个）标 1 的相邻最小项，可合并为一项，并消去两个变量。

在四变量卡诺图中,4个标1的小方格相邻的典型情况如图3-9所示。

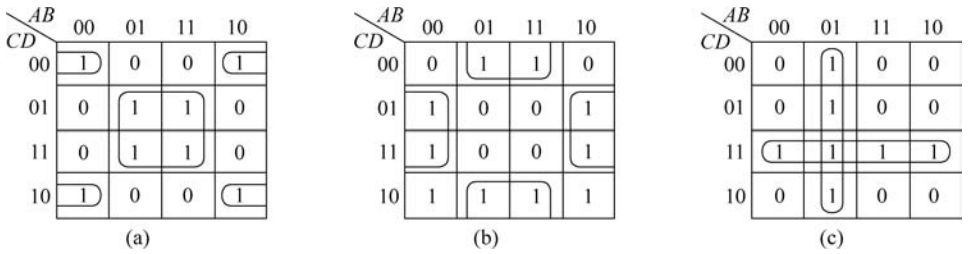


图3-9 4个相邻最小项合并的情况

在图3-9(a)中,可得函数表达式为

$$F(A,B,C,D)=\overline{B}\,\overline{D}+BD$$

在图3-9(b)中,可得函数表达式为

$$F(A,B,C,D)=B\overline{D}+\overline{B}D$$

在图3-9(c)中,可得函数表达式为

$$F(A,B,C,D)=\overline{A}B+CD$$

性质3:任何8个(2^3个)标1的相邻最小项,可合并为一项,并消去三个变量。

由上述性质可知,相邻最小项的数目必须为2^i个才能合并成一项,并消去i个变量。包含的最小项数目越多,消去的变量也越多,从而所得到的逻辑表达式就越简单。

视频讲解

4. 卡诺图化简的步骤

在化简中涉及的几个概念:

(1) 蕴涵项。在函数的任何与或表达式中,每个与项称为该函数的蕴涵项(Implicant)。显然,在函数的卡诺图中,任一标1的最小项以及由2^i个相邻最小项所形成的圈都是函数的蕴涵项。

(2) 质蕴涵项。如果函数的某一蕴涵项不是该函数中其他蕴涵项的一个子集,则此蕴涵项称为质蕴涵项(Prime Implicant)。从卡诺图上看,所谓质蕴涵项就是大得不能再大的圈。例如,在图3-10中,BD、$A\overline{C}\,\overline{D}$和$AB\overline{C}$都是质蕴涵项,而$BCD$、$\overline{A}B\overline{C}D$等都不是质蕴涵项。

(3) 必要质蕴涵项。如果函数的一个质蕴涵项,至少包含了一个其他任何质蕴涵项都不包含的标1最小项,则此质蕴涵项称为必要质蕴涵项(Essential Prime Implicant)。例如,在图3-10中,BD、$A\overline{C}\,\overline{D}$是必要质蕴涵项,而$AB\overline{C}$就不是必要质蕴涵项。

根据以上定义,可以给出用卡诺图化简布尔函数的基本步骤如下:

(1) 将逻辑函数正确地标到卡诺图上,并在图上找出所有质蕴涵项;

(2) 求出所有必要质蕴涵项;

(3) 求函数的最小覆盖(即函数的最简表达式)。

下面通过具体例子来说明上述步骤。

例3-7 用卡诺图法化简函数。

$$F(A,B,C,D)=\sum m(0,3,4,5,7,11,13,15)$$

解:

(1) 作F的卡诺图,并求得所有质蕴涵项为BD、CD、$\overline{A}\,\overline{C}\,\overline{D}$、$\overline{A}B\overline{C}$,如图3-11所示。

(2) 求出所有必要质蕴涵项为 CD、BD、$\overline{A}\overline{C}\overline{D}$。

(3) 由于必要质蕴涵项的集合已覆盖了函数的所有最小项，因此函数的最简与或式为

$$F(A, B, C, D) = BD + CD + \overline{A}\overline{C}\overline{D}$$

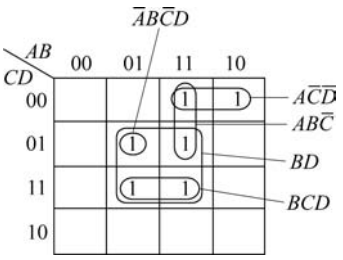


图 3-10　卡诺图中的质蕴涵项

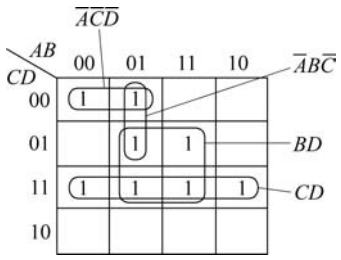


图 3-11　例 3-7 图

例 3-8　用卡诺图化简函数。

$$F(A, B, C, D) = \sum m(0, 1, 3, 4, 7, 12, 13, 15)$$

解：作 F 的卡诺图如图 3-12(a)所示。该函数有 8 个质蕴涵项，它们相互交连，找不出哪个是必要质蕴涵项，这种情况通常称为循环结构。对于这类循环结构，通常可选取一个最大的质蕴涵圈作为必要质蕴涵，以打破循环结构。本例中，由于各个质蕴涵圈的大小相等，故可任选一个质蕴涵作为必要质蕴涵项。图 3-12(b)为其中的一种解，可得 F 的最简表达式为

$$F(A, B, C, D) = \overline{A}\overline{B}\overline{C} + \overline{A}CD + ABD + B\overline{C}\overline{D}$$

同理，可得 F 的另一个最简表达式为

$$F(A, B, C, D) = \overline{A}\overline{B}D + BCD + AB\overline{C} + \overline{A}\overline{C}\overline{D}$$

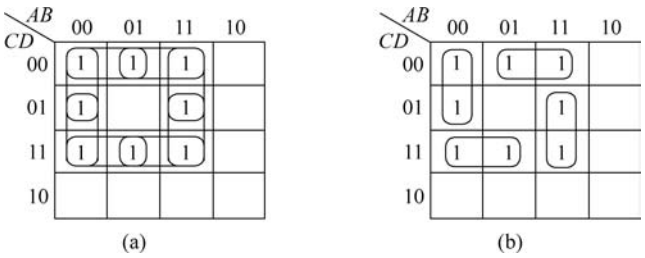


图 3-12　例 3-8 图

例 3-9　化简五变量函数。

$$F(A, B, C, D, E) = \sum m(2, 4, 5, 6, 7, 12, 13, 18, 20, 21, 22, 23, 24, 25, 28, 29)$$

解：五变量布尔函数可用两个四变量卡诺图来表示，相重叠的最小项是相邻的，可以合并。画出函数 F 的卡诺图如图 3-13 所示。重叠部分的质蕴涵为 $\overline{B}D\overline{E}$、$\overline{B}C$ 和 $C\overline{D}$；不重叠的质蕴涵为 $AB\overline{D}$。这几个都为必要质蕴涵，且覆盖了函数的全部标 1 最小项。因此，函数的最简表达式为

$$F(A,B,C,D,E) = (\overline{B}D\overline{E} + \overline{B}C + C\overline{D} + AB\overline{D})$$

由上述例子可见，用卡诺图化简布尔函数，简单明了，形象直观，容易掌握。

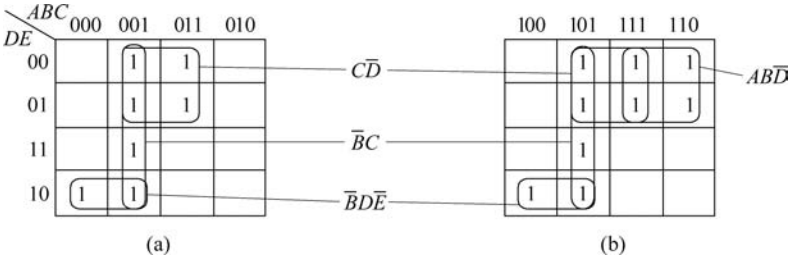


图 3-13　例 3-9 图

视频讲解

3.2.3　列表化简法

上面介绍的卡诺图化简法虽然简单、直观，但毕竟是手算的一种方法，不便于机器实现，并且当变量数大于 6 时，这种方法就失去它的优越性了。

现介绍一种更有规律、系统的方法，即列表化简法。该方法是由奎恩(W. V. Quine)和麦克拉斯基(E. J. McCluskey)于 1956 年研究发表的，故也称奎恩-麦克拉斯基法，简称 Q-M 法。这种方法的基本步骤与卡诺图类似，只是化简过程通过约定的表格进行。列表法可分成 4 个步骤：

(1) 将函数写成最小项形式，并用二进制数表示最小项，将所有的最小项列成表，按二进制数中 1 的个数分成组。

(2) 逐个最小项比较(组与组之间)，找出所有质蕴涵项。

(3) 列质蕴涵表，求必要质蕴涵项。

(4) 列简化的质蕴涵表，进一步求最小覆盖。

例 3-10　用列表法化简逻辑函数。

$$F(A, B, C, D) = \sum m(2, 4, 6, 8, 9, 10, 12, 13, 15)$$

解：

(1) 将函数的最小项用二进制表示并分组列表，如表 3-8 所示。

表 3-8　例 3-10 表

m_i	A	B	C	D		m_i	A	B	C	D		m_i	A	B	C	D	
2	0	0	1	0	√	2,6	0	—	1	0	P_2	8,9 12,13	1	—	0	—	P_1
4	0	1	0	0	√	2,10	—	0	1	0	P_3						
8	1	0	0	0	√	4,6	0	1	—	0	P_4						
6	0	1	1	0	√	4,12	—	1	0	0	P_5						
9	1	0	0	1	√	8,9	1	0	0	—	√						
10	1	0	1	0	√	8,10	1	0	—	0	P_6						
12	1	1	0	0	√	8,12	1	—	0	0	√						
13	1	1	0	1	√	9,13	1	—	0	1	√						
15	1	1	1	1	√	12,13	1	1	0	—	√						
						13,15	1	1	—	1	P_7						

(2) 在组与组之间逐个最小项进行比较，如相邻，则合并成一项，消去一个变量，用“—”表示，将已合并的蕴涵项标“√”，将得到的新蕴涵项列于表格的另一列。在新的蕴涵项表中

再进行同样的工作，得到第三列表，直至没有可合并的蕴涵项为止。

将没有标“√”的蕴涵项找出来标以 P_1，P_2，…这就是该函数所有的质蕴涵项。这个例子中所有的质蕴涵项为

$$P_1=A\overline{C},\ P_2=\overline{A}C\overline{D},\ P_3=\overline{B}C\overline{D},\ P_4=\overline{A}B\overline{D},\ P_5=B\overline{C}\overline{D},\ P_6=A\overline{B}\overline{D},\ P_7=ABD$$

(3) 列质蕴涵表，找必要质蕴涵。将上面求出的所有质蕴涵列成表，如表 3-9 所示。

表 3-9　质蕴涵表

P_j	m_i								
	m_2	m_4	m_6	m_8	m_9	m_{10}	m_{12}	m_{13}	m_{15}
√ P_1				×	⊗		×	×	
P_2	×		×						
P_3	×					×			
P_4		×	×						
P_5		×					×		
P_6				×		×			
√ P_7								×	⊗
覆盖情况				×	×		×	×	×

表 3-9 中横向表头中为函数包括的最小项，纵向表头中是求出的所有质蕴涵。对于每个质蕴涵，它包含的最小项处标×。从表的纵向看，如果一列中只有一个×，则将该×标成⊗。那么包含⊗的质蕴涵即为必要质蕴涵，该例中找到 P_1 和 P_7 两个必要质蕴涵，将 P_1 和 P_7 所包含的最小项列在覆盖情况栏内，从该栏可看到，所要化简的函数还有 m_2、m_4、m_6、m_{10} 没有被覆盖。

(4) 列简化质蕴涵表，求最小覆盖。

把已经决定的必要质蕴涵项及它所覆盖的最小项从质蕴涵表中去掉，得简化的质蕴涵表，如表 3-10 所示。

表 3-10　简化质蕴涵表

P_j	m_i			
	m_2	m_4	m_6	m_{10}
P_2	×		×	
P_3	×			×
P_4		×	×	
P_5		×		
P_6				×

在表 3-10 中，要取最少的质蕴涵(必要质蕴涵)而覆盖剩下的所有最小项。应该如何选择呢？下面介绍常用的两种方法。

① 行列消去法。

优势行规则：设质蕴涵 P_i 和 P_j 是简化质蕴涵表中的两行，其中 P_j 行中的×完全包含在 P_i 行中，则称 P_i 为优势行，P_j 为劣势行，记作 $P_i \supset P_j$，这时，在简化质蕴涵表中可以消去劣势行 P_j(这是因为选取了优势行 P_i 后，不仅可覆盖劣势行 P_j 所覆盖的最小项，而且还可覆盖其他最小项)。

例如表 3-11 为一简化质蕴涵表，表中 P_2 相对于 P_1 来说是劣势行，即 $P_1 \supset P_2$，可消去

P_2；同理，P_3 相对于 P_1 来说也是劣势行，即 $P_1 \supset P_3$，可消去 P_3，最后只剩下 P_1。

优势列规则：设最小项 m_i 和 m_j 是简化质蕴涵表中的两列，其中 m_j 列中的×完全包含在 m_i 列之中，则称 m_i 列为优势列，m_j 为劣势列，记作 $m_i \supset m_j$，这时，在简化质蕴涵表中可以消去优势列 m_i（这是因为选取了覆盖劣势列的质蕴涵后，一定能覆盖优势列，反之则不一定）。

例如，设表 3-12 为简化质蕴涵表，表中的 m_2 和 m_3 相对于 m_1 来说是劣势列，即 $m_2 \subset m_1$，$m_3 \subset m_1$，故可消去优势列 m_1。

表 3-11　简化质蕴涵表

P_i	m_i	
	m_1	m_3
P_1	×	×
P_2	×	
P_3		×

表 3-12　优势列举例

P_j	m_i		
	m_1	m_2	m_3
P_1	×	×	
P_2	×		×
P_3	×		

这种优势行规则和优势列规则可以反复交替使用，使简化的质蕴涵表进一步简化，以便求得所需的质蕴涵项。

对于表 3-10 的简化质蕴涵表，用行列消去法进行求最小覆盖，从表中可看出

$$P_4 \supset P_5, \quad P_3 \supset P_6 \quad \text{（优势行规则）}$$

故可消去 P_5，P_6。

又由于

$$m_4 \subset m_6, \quad m_{10} \subset m_2 \quad \text{（优势列规则）}$$

故可消去 m_2，m_6。最后，求得所需的质蕴涵为 P_3 和 P_4，它覆盖了简化质蕴涵表的所有最小项 m_2，m_4，m_6 和 m_{10}。

② 逻辑代数法。

所谓逻辑代表数法，就是从简化质蕴涵表列出逻辑表达式，从中选出最简的所需质蕴涵项，对于该例题，表 3-10 要覆盖所有最小项，其逻辑表达式为

$$\begin{aligned}&(P_2+P_3)(P_4+P_5)(P_2+P_4)(P_3+P_6)\\=&(P_3+P_2P_6)(P_4+P_2P_5)\\=&P_3P_4+P_2P_3P_5+P_2P_4P_5+P_2P_5P_6\end{aligned}$$

该式表明，要覆盖 m_2，m_4，m_6，m_{10}，有 4 个方案，即 P_3P_4，$P_2P_3P_5$，$P_2P_4P_5$，$P_2P_5P_6$，当然 P_3P_4 为最小覆盖，所以取 P_3P_4，这与行列消去法的结果是一样的。

所以例 3-10 的最后结果为

$$F(A, B, C, D)=P_1+P_7+P_3+P_4=A\overline{C}+ABD+\overline{B}C\overline{D}+\overline{A}B\overline{D}$$

视频讲解

3.2.4　逻辑函数化简中的两个实际问题

1. 包含无关最小项的逻辑函数化简

前面讨论的逻辑函数对于输入变量的任何一种取值组合都有确定的函数值与之对应。假定一个含 n 个变量的函数能用 k 个最小项之和表示，那么这 k 个最小项就给出了使函数值为 1 的 k 种输入变量取值组合，而剩下的 2^n-k 个最小项给出了使函数值为 0 的 2^n-k 种输入变量取值组合。这就表明了一个含 n 个变量的逻辑函数与 2^n 个最小项均有关。

但在实际问题中，往往存在这样两种情况：

(1) 由于某种特殊限制，使得输入变量的某些取值组合根本不会出现。

(2) 虽然每种输入组合都可能出现，但对其中的某些输入取值组合究竟使函数值为 1 还是为 0，对输出没有影响。

称这部分输入组合为无关最小项，简称无关项，用 d 表示。包含无关项的逻辑函数称不完全确定的逻辑函数。

因为无关最小项对应的输入变量取值组合根本不会出现，或者尽管可能出现，但相应的函数值是什么无关紧要。所以，在变量的这些取值下，函数可以任意取值 0 或 1。换而言之，无关最小项可以随意加到函数表达式中或不加到函数表达式中，而并不影响函数的实际逻辑功能。根据这一特点，在化简这类函数时，往往可以通过对无关最小项进行适当地取舍，从而使逻辑函数得到更好的化简。下面举例说明。

例 3-11 用卡诺图化简逻辑函数 $F(A, B, C, D)=\sum m(3, 4, 5, 10, 11, 12)+\sum d(0, 1, 2, 13, 14, 15)$。

解：做出该函数表达式的卡诺图，如图 3-14 所示。图中填 d 的小方格既可当成 1，也可当作 0，具体处理可根据哪样更有利于函数化简的原则来确定。

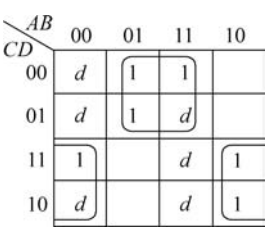


图 3-14 例 3-11 的图

画在圈内的 d 即作为 1 看待，圈外的作为 0 看待。结果得

$$F(A, B, C, D)=B\overline{C}+\overline{B}C$$

例 3-12 用列表法化简不完全确定的逻辑函数 $F(A, B, C, D)=\sum m(1, 3, 4, 5, 10, 11, 12)+\sum d(2, 13)$。

解：用列表法化简这种包含无关项的逻辑函数时，应注意两点：一是在列表求所有质蕴涵时，应该令 $d=1$，以尽量利用随意项进行合并；二是在列质蕴涵表时，应令 $d=0$，即随意项的覆盖问题可不必考虑，以有利于得到最简式。

(1) 求所有质蕴涵项。令所有无关项为 1，如表 3-13 所示。求得所有质蕴涵项为 $P_1=\overline{B}C$，$P_2=B\overline{C}$，$P_3=\overline{A}\overline{B}D$，$P_4=\overline{A}\overline{C}D$。

表 3-13 例 3-12 的表

m_i	A	B	C	D		m_i	A	B	C	D		m_i	A	B	C	D	
m_1	0	0	0	1	√	m_1, m_3	0	0	—	1	P_3	m_2, m_3 m_{10}, m_{11}	—	0	1	—	P_1
m_2	0	0	1	0	√	m_1, m_5	0	—	0	1	P_4	m_4, m_5 m_{12}, m_{13}	—	1	0	—	P_2
m_4	0	1	0	0	√	m_2, m_3	0	0	1	—	√						
m_3	0	0	1	1	√	m_2, m_{10}	—	0	1	0	√						
m_5	0	1	0	1	√	m_4, m_5	0	1	0	—	√						
m_{10}	1	0	1	0	√	m_4, m_{12}	—	1	0	0	√						
m_{12}	1	1	0	0	√	m_3, m_{11}	—	0	1	1	√						
m_{11}	1	0	1	1	√	m_5, m_{13}	—	1	0	1	√						
m_{13}	1	1	0	1	√	m_{10}, m_{11}	1	0	1	—	√						
						m_{12}, m_{13}	1	1	0	—	√						

(2) 求必要质蕴涵项。令所有无关项为0,即在质蕴涵表中不必列无关项。如表3-14所示,找到必要质蕴涵项 P_1,P_2。

表3-14 质蕴涵表

P_j	m_i						
	m_1	m_3	m_4	m_5	m_{10}	m_{11}	m_{12}
√ P_1		×			⊗	⊗	
√ P_2			⊗	×			⊗
P_3	×	×					
P_4	×	×		×			
覆盖情况		×	×	×	×	×	×

(3) 求函数的最小覆盖。

去掉质蕴涵表中必要质蕴涵及已被它覆盖的最小项后,只剩下最小项 m_1 未被覆盖,直接选取 P_3 或 P_4 可覆盖 m_1。因此函数的最简表达式为

$$F(A,B,C,D)=P_1+P_2+P_3=B\overline{C}+\overline{B}C+\overline{A}\overline{B}D$$

2. 多输出逻辑函数的化简

关于单个逻辑函数的化简问题,已经进行了系统讨论。但在实际问题中,大量存在着根据一组相同输入变量产生多个输出函数的情况。对于一个具有相同输入变量的多输出逻辑电路,如果只是孤立地将单个输出函数一一化简,然后直接拼在一起,通常并不能保证整个电路最简,因为各输出函数之间往往存在可共享的部分。这就要求在化简时把多个输出函数当作一个整体考虑,以整体最简为目标。下面,以"与-或"表达式为例来介绍多输出函数的化简。

衡量多输出函数最简的标准是:

(1) 所有逻辑表达式中包含的不同"与"项总数最少。

(2) 在满足上述条件的前提下,各不同"与"项中所含的变量总数最少。

多输出函数化简的关键是充分利用各函数之间可供共享的部分。例如,某逻辑电路有两个输出函数:

$$F_1(A,B,C)=A\overline{B}+A\overline{C}$$

$$F_2(A,B,C)=AB+BC$$

其对应的卡诺图如图3-15(a)和图3-15(b)所示。从卡诺图可以看出,就单个函数而言,F_1,F_2 均已达到最简。此时,两个函数表达式共含4个不同"与"项,4个不同"与"项所包含的变量总数为8个。

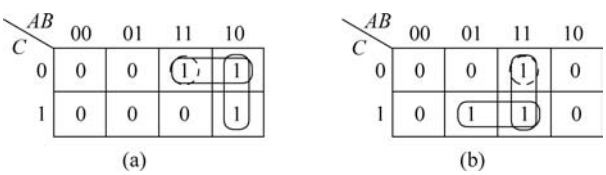


图3-15 函数 F_1,F_2 的卡诺图

假如用卡诺图化简上述函数,如图3-15中虚线所示,将最小项 m_6 单独圈出,则可得到函数表达式

$$F_1(A,B,C)=A\overline{B}+AB\overline{C}$$

$$F_2(A,B,C)=BC+AB\overline{C}$$

这样处理后，尽管从单个函数来看，上述两个表达式均未达到最简。但从整体来说，由于恰当地利用了两个函数的共享部分，使两个函数表达式中的不同“与”项总数由原来的 4 个减少为 3 个，各不同“与”项包含的变量总数由 8 个减少为 7 个，从而使整体得到了进一步简化。

这里，简单介绍一下多输出函数的卡诺图化简法。

用卡诺图化简多输出函数一般分为两步进行。首先按单个函数的化简方法用卡诺图对各函数逐个进行化简。其次在卡诺图上比较两个以上函数的相同 1 方格部分，看是否能够通过改变卡诺图的画法找出公共项。在进行后一步时要注意：第一，卡诺圈的变动必须在两个或多个卡诺图的相同 1 方格部分进行，只有这样，对应的项才能供两个或多个函数共享；第二，卡诺圈的变动必须以使整体得到进一步简化为原则。下面举例说明。

例 3-13 用卡诺图化简多输出函数。

$$F_1(A,B,C,D)=\sum m(2,3,5,7,8,9,10,11,13,15)$$

$$F_2(A,B,C,D)=\sum m(2,3,5,6,7,10,14,15)$$

$$F_3(A,B,C,D)=\sum m(6,7,8,9,13,14,15)$$

解：先画出三个函数的卡诺图(见图 3-16)，按单个函数的卡诺图化简，图中实线部分，得最简函数表达式

$$F_1(A,B,C,D)=BD+\overline{B}C+A\overline{B}$$

$$F_2(A,B,C,D)=C+\overline{A}BD$$

$$F_3(A,B,C,D)=BC+A\overline{B}\overline{C}+ABD$$

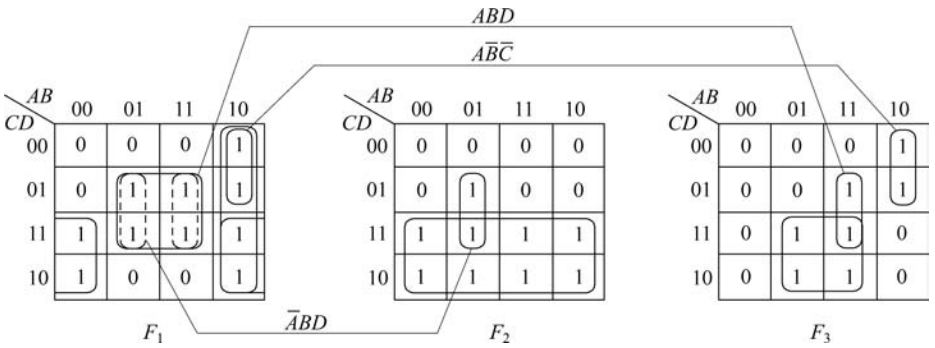


图 3-16 例 3-13 的卡诺图

以上三个函数表达式中共含 8 个不同“与”项，各“与”项包含的总变量数为 18 个。进一步观察三个卡诺图上相同的 1 方格，不难发现：F_2 中的“与”项 $\overline{A}BD$，F_3 中的“与”项 ABD 和 $A\overline{B}\overline{C}$ 对应的 1 方格均出现在 F_1 卡诺图上。如果用 $\overline{A}BD+ABD$ 代替 F_1 的“与”项 BD，则可在总的“与”项中减少一个独立的“与”项 BD。同样，F_1 中的“与”项 $A\overline{B}$ 可以由 F_3 中的“与”项 $A\overline{B}\overline{C}$ 代替。图 3-16 中的虚线部分，就是找出的函数之间的公共部分，这样就使函数从整体上得到进一步简化。经修改后的最简结果为

$$F_1(A,B,C,D)=\overline{A}BD+ABD+A\overline{B}\overline{C}+\overline{B}C$$

$$F_2(A,B,C,D)=C+\overline{A}BD$$

$$F_3(A,B,C,D)=BC+A\overline{B}\overline{C}+ABD$$

从单个函数看不一定是最简,但总体看该组表达式中只含6个不同“与”项,各不同“与”项所含的变量总数为14个,比按单个函数最简形式组成的总体更简单。

视频讲解

3.3 组合逻辑电路的分析

数字逻辑电路按其功能和结构的不同,可以分成两大类:一类叫作组合逻辑电路,简称组合电路;另一类叫作时序逻辑电路,简称时序电路。

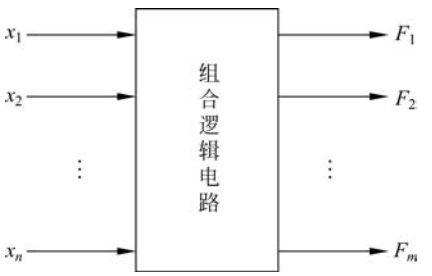


图3-17 组合逻辑电路框图

所谓组合电路是指电路在任何时刻产生的稳定输出值仅与该时刻的输入值有关,而与电路过去的输入值无关。组合逻辑电路可用如图3-17所示的框图来描述。图中,$x_1,x_2,\cdots,x_n$ 是电路的 n 个输入变量,$F_1,F_2,\cdots,F_m$ 是电路的 m 个输出信号。组合电路的输出与输入之间的逻辑关系可表示为

$$F_i=f_i(x_1,x_2,\cdots,x_n),\quad i=1,2,\cdots,m$$

从结构上看,组合电路具有以下两个特点。

(1) 电路仅由逻辑门电路构成,没有记忆能力。这是因为组合电路的输出仅与当时的输入有关,而与过去的状态无关,所以不需要任何记忆元件。

(2) 电路中不存在任何输出到输入的反馈回路。

对数字逻辑电路的研究包括两个方面,即电路的分析与设计。组合电路也不例外。对于一个给定的逻辑电路,找出其逻辑功能的过程称为分析;而对于已知的逻辑功能要求,确定用什么样的逻辑电路来实现的过程叫作设计,或称为逻辑综合。显然,分析和设计是两个相反的过程。本节和3.4节将分别讨论组合电路的分析和设计方法。

3.3.1 组合逻辑电路分析的一般方法

当对某一给定的逻辑电路进行研究时,经常需要推敲其设计思想,或要更换其中的某些部件,或要评价其经济技术指标。为此,实际应用中经常需要对给定的逻辑电路进行分析。对组合逻辑电路的分析就是根据给定的组合逻辑电路,写出输出函数表达式,并在必要时对其进行化简,以此确定输出和输入的逻辑关系,进而分析出电路实现的功能,并可对电路设计是否合理做出评定。

一般情况下,组合逻辑电路的分析可按下述步骤进行。

(1) 根据给定的逻辑电路图,写出输出函数表达式。首先,将全部逻辑门的输入和输出端都标以字母;然后从最靠近输入的一端开始,依次写出各个门的逻辑表达式,并将前级门的输出函数代入后一级门的输出函数表达式中,直至得到仅以输入变量表示的最终输出函数表达式。

(2) 对(1)中得到的输出函数表达式进行化简。开始得到的输出函数表达式往往不是

最简的，为了更清楚地分析逻辑电路的功能，应对其进行化简。化简方法可视具体情况，灵活运用前面所学的方法。化简结果可作为评价电路功能和经济技术指标的依据。

(3) 根据化简后的逻辑函数表达式列出真值表。真值表详尽地反映了输入输出的取值关系，可以直观地描述电路的逻辑功能。

(4) 逻辑功能描述。根据输出函数表达式和真值表，用文字概括地描述电路的功能，并对原电路的设计方案进行评定，必要时可提出改进意见和方案。

3.3.2 组合逻辑电路分析举例

例 3-14 分析如图 3-18(a)所示的组合逻辑电路。

解：(1) 根据给定的逻辑电路图，写出输出函数表达式。

由图 3-18(a)可知

$$P_1=\overline{ABC},\quad P_2=A\cdot P_1=A\cdot\overline{ABC},\quad P_3=B\cdot P_1=B\cdot\overline{ABC}$$

$$P_4=C\cdot P_1=C\cdot\overline{ABC},\quad F=\overline{P_2+P_3+P_4}=\overline{A\cdot\overline{ABC}+B\cdot\overline{ABC}+C\cdot\overline{ABC}}$$

(2) 对输出函数表达式进行化简。

对该输出函数表达式可用代数法化简如下：

$$F=\overline{A\cdot\overline{ABC}+B\cdot\overline{ABC}+C\cdot\overline{ABC}}=\overline{\overline{ABC}}(A+B+C)$$

$$=\overline{\overline{ABC}}+\overline{A+B+C}=ABC+\overline{A}\,\overline{B}\overline{C}$$

(3) 根据化简后的输出函数表达式列出真值表，如表 3-15 所示。

表 3-15 例 3-14 的真值表

A	B	C	F	A	B	C	F
0	0	0	1	1	0	0	0
0	0	1	0	1	0	1	0
0	1	0	0	1	1	0	0
0	1	1	0	1	1	1	1

(4) 进行逻辑功能描述。

由真值表可知，仅当输入 A、B、C 全为 0 或全为 1 时，输出 F 才为 1；否则 F 为 0。即当三个输入变量的值完全一致时，输出为 1，否则输出为 0。因此，通常称该电路为“一致性”检查电路。在某些可靠性要求较高的系统中，如控制系统，常常让几套相同的设备同时工作，该电路可对各设备的运行结果进行“一致性”检查，一旦运行结果不一致，便可报警，以确保系统的高可靠性。

由分析可知，电路的原设计方案并不是最佳的，根据化简后的表达式，可将其改为如图 3-18(b)所示的更简单、清晰的电路。

以上介绍分析步骤是就一般情况而言的，实际问题中可根据电路的复杂程度和具体要求进行适当取舍、灵活运用。

例 3-15 分析如图 3-19(a)所示的组合逻辑电路。

解：由图 3-19(a)可知

$$P_1=\overline{A}+\overline{B},P_2=\overline{\overline{A}+C},P_3=B\oplus C,P_4=B+C,P_5=\overline{P_1P_2}=\overline{(\overline{A}+\overline{B})(\overline{A}+C)}$$

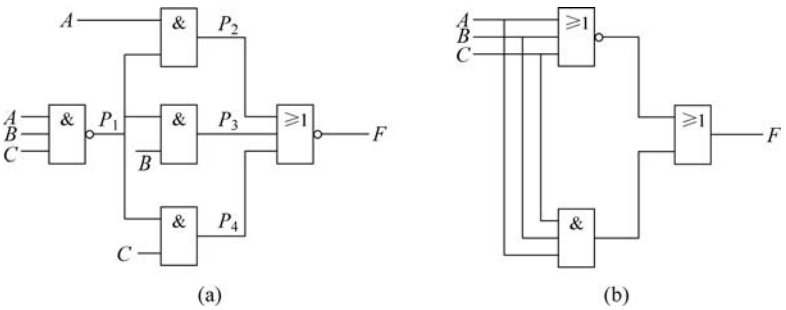


图 3-18　例 3-14 的逻辑电路图

$P_6=P_3P_4=(B\oplus C)(B+C)$，$F=P_5P_6=\overline{(\overline{A}+\overline{B})\overline{(\overline{A}+C)}}(B\oplus C)(B+C)$

用代数法对 F 化简如下：

$$F=\overline{(\overline{A}+\overline{B})\overline{(\overline{A}+C)}}(B\oplus C)(B+C)=(AB+\overline{A}+C)(B\oplus C)(B+C)$$
$$=(B+\overline{A}+C)(B+C)(B\oplus C)=(B+C)(B\oplus C)$$
$$=(B+C)(\overline{B}C+B\overline{C})=(\overline{B}C+B\overline{C})=B\oplus C$$

可见，该电路实现的是“异或”功能，显然，原电路的设计是不合理的，只需一个异或门便可实现其功能，如图 3-19(b)所示。

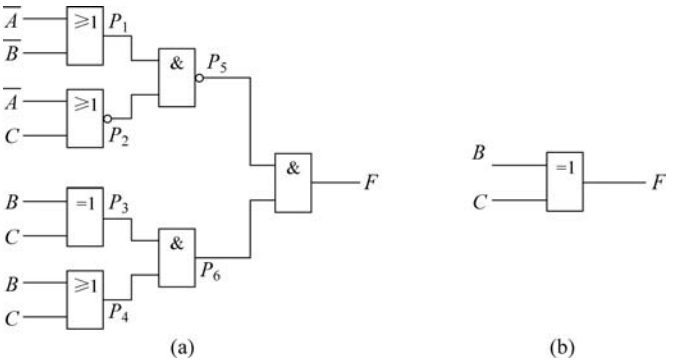


图 3-19　例 3-15 的逻辑电路图

例 3-16　分析如图 3-20 所示的组合电路。假定图中 M 为控制变量，输入变量 A，B，C，D 和输出变量 W，X，Y，Z 均表示一位二进制数，试说明在 $M=0$ 和 $M=1$ 时，电路分别实现什么功能。

解：根据图 3-20 可直接写出电路的输出函数表达式。

$$W=A,\quad X=A\oplus B$$
$$Y=\overline{\overline{M(X\oplus C)}\cdot\overline{\overline{M}(B\oplus C)}}=M(X\oplus C)+\overline{M}(B\oplus C)$$
$$=M(A\oplus B\oplus C)+\overline{M}(B\oplus C)$$
$$Z=\overline{\overline{M(X\oplus C\oplus D)}\cdot\overline{\overline{M}(C\oplus D)}}=M(X\oplus C\oplus D)+\overline{M}(C\oplus D)$$
$$=M(A\oplus B\oplus C\oplus D)+\overline{M}(C\oplus D)$$

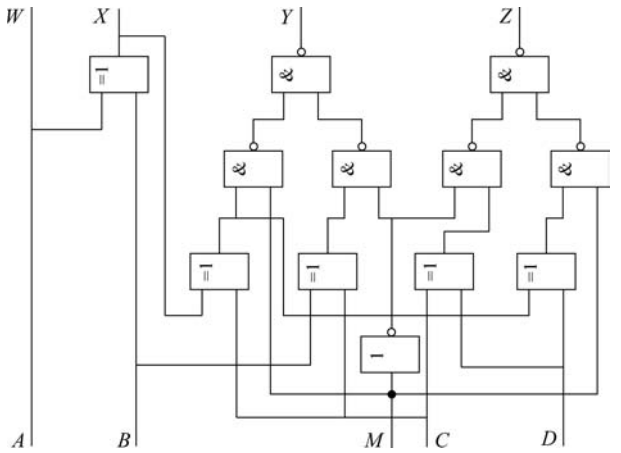


图 3-20　例 3-16 的逻辑电路图

由此可知，当 $M=0$ 时，输出函数表达式为

$$W=A \qquad X=A\oplus B$$
$$Y=B\oplus C \quad Z=C\oplus D$$

当 $M=1$ 时，输出函数表达式为

$$W=A \qquad X=A\oplus B$$
$$Y=A\oplus B\oplus C \quad Z=A\oplus B\oplus C\oplus D$$

至此，由 1.2.3 节可知，该电路实现的逻辑功能是 4 位二进制码和 Gray 码之间的相互转换。当 $M=0$ 时，将 4 位二进制码转换成 Gray 码；当 $M=1$ 时，将 4 位 Gray 码转换成相应的二进制码。

3.4　组合逻辑电路的设计

视频讲解

3.4.1　组合逻辑电路设计的一般方法

组合逻辑电路的设计是根据给定的逻辑功能要求，选用适当的门电路设计出实现该功能的逻辑电路的过程。可见，组合逻辑电路的设计是其分析的逆过程。

一般，组合逻辑电路的设计过程可分为以下 4 步。

(1) 根据给定的逻辑功能要求，进行逻辑约定并列出真值表。

给定的逻辑功能要求通常是用文字描述的，由此直接写出逻辑函数表达式是比较困难的，但可先据此列出真值表。对任何逻辑问题，只要能列出其真值表，就能设计出逻辑电路图。因此，列真值表是解决问题最关键的一步，如果真值表有错误，则以后的设计就前功尽弃。要列真值表，关键是正确而全面地理解问题的文字描述，找出问题中输入和输出之间的逻辑关系，并用适当的逻辑变量表示输入和输出，再对它们的取值含义做出约定，对每种可能的输入取值组合，求出对应的输出值即可。

(2) 根据真值表写出逻辑函数的“最小项之和”表达式。

由真值表可以很容易地写出逻辑函数的“最小项之和”表达式。但对于某些简单的逻辑

问题,也可以不列真值表,直接写出其逻辑函数表达式。

(3) 将逻辑函数的“最小项之和”表达式,化成最简“与-或”式,并进行适当变换。

由真值表得到的逻辑函数的“最小项之和”表达式并不一定是最简形式,只有将其化成最简形式,才能使设计出的逻辑电路最简单,器件最节省。对于一个具体的逻辑问题,在设计时,有时还要考虑使用的门电路的类型、数量、扇入系数、扇出系数、级数等因素。因此,有时还需将逻辑函数的最简“与-或”式变换成相应的形式,以满足具体实现的要求。

(4) 根据化简或变换后的逻辑函数表达式,画出逻辑电路图。

以上设计步骤是就一般情况而言的,在实际问题中,根据问题的难易程度和设计者的水平,有时可以跳过其中的某些步骤,设计中可视具体情况灵活应用。

例 3-17 用“与非”门设计一个四变量的“多数表决电路”。

解:(1) 根据逻辑功能要求建立真值表。

不难理解,“多数表决电路”就是按照少数服从多数的原则进行表决,以确定某项决议是否通过的一种电路。假设用 A、B、C、D 代表参与表决的 4 个逻辑变量,F 表示表决结果。并且约定,输入用 1 表示同意,0 表示反对;输出用 1 表示通过,0 表示被否决。按照这些约定,当输入变量 A、B、C、D 中有三个或三个以上取值为 1 时,函数 F 的值为 1,否则函数 F 的值为 0。据此可得真值表,如表 3-16 所示。

表 3-16 例 3-17 的真值表

A	B	C	D	F	A	B	C	D	F
0	0	0	0	0	1	0	0	0	0
0	0	0	1	0	1	0	0	1	0
0	0	1	0	0	1	0	1	0	0
0	0	1	1	0	1	0	1	1	1
0	1	0	0	0	1	1	0	0	0
0	1	0	1	0	1	1	0	1	1
0	1	1	0	0	1	1	1	0	1
0	1	1	1	1	1	1	1	1	1

(2) 根据真值表写出函数的“最小项之和”表达式。

由如表 3-16 所示的真值表,可直接写出函数 F 的“最小项之和”表达式,即

$$F(A,B,C,D)=\sum m(7,11,13,14,15)$$

(3) 化简函数表达式,并进行适当变换。

函数 F 的卡诺图如图 3-21(a)所示,F 的最简“与-或”式为

$$F(A,B,C,D)=ABC+ABD+ACD+BCD$$

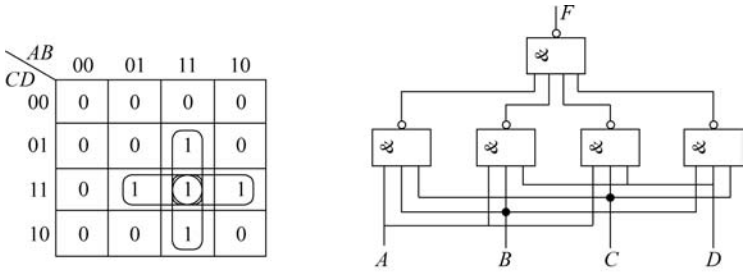
为达到用“与非”门实现的目的,需将“与-或”式转换成“与非-与非”式,即

$$F(A,B,C,D)=\overline{ABC+ABD+ACD+BCD}=\overline{\overline{ABC}\cdot\overline{ABD}\cdot\overline{ACD}\cdot\overline{BCD}}$$

(4) 画出逻辑电路图。

由函数的“与非-与非”式,可画出对应的逻辑电路图,如图 3-21(b)所示。

例 3-18 用“与非”门设计一个燃油锅炉自动报警器。要求燃油喷嘴在开启状态下,若锅炉水温或压力过高则发出报警信号。



(a) 卡诺图　　(b) 逻辑电路图

图 3-21　例 3-17 的卡诺图及逻辑电路图

解：(1) 根据功能要求进行逻辑约定并建立真值表。

将喷嘴开关、锅炉水温、压力分别用变量 A、B、C 表示：$A=1$ 表示喷嘴开关打开，$A=0$ 表示喷嘴开关关闭；B、C 为 1 表示温度、压力过高，为 0 表示温度、压力正常。报警信号作为输出变量用 F 表示，$F=0$ 正常，$F=1$ 报警。据此可列出真值表，如表 3-17 所示。

表 3-17　例 3-18 的真值表

A	B	C	F	A	B	C	F
0	0	0	0	1	0	0	0
0	0	1	0	1	0	1	1
0	1	0	0	1	1	0	1
0	1	1	0	1	1	1	1

(2) 根据真值表写出函数的“最小项之和”表达式。

由如表 3-17 所示的真值表，可直接写出函数 F 的“最小项之和”表达式，即

$$F(A,B,C)=\sum m(5,6,7)$$

(3) 化简函数表达式，并进行适当的变换。

函数 F 的卡诺图如图 3-22(a)所示，F 的最简“与-或”式为

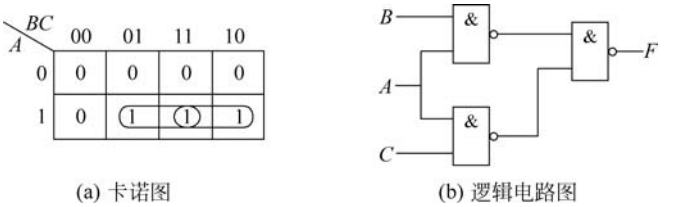
$$F(A,B,C)=AB+AC$$

为达到用“与非”门实现的目的，需将“与-或”式转换成“与非-与非”式，即

$$F(A,B,C)=\overline{\overline{AB+AC}}=\overline{\overline{AB}\cdot\overline{AC}}$$

(4) 画出逻辑电路图。

由函数的“与非-与非”式，可画出对应的逻辑电路，如图 3-22(b)所示。



(a) 卡诺图　　(b) 逻辑电路图

图 3-22　例 3-18 的卡诺图及逻辑电路图

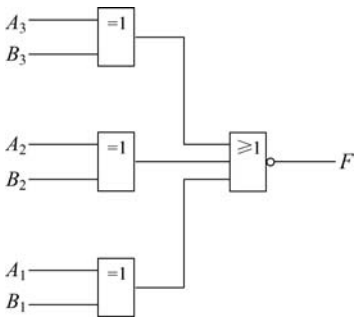


图 3-23　例 3-19 比较器的逻辑电路图

例 3-19　设计一个比较两个三位二进制数是否相等的数值比较器。

解：设待比较的两个三位二进制数分别为 $A=A_3A_2A_1$，$B=B_3B_2B_1$，电路的输出为 F。当 $A=B$ 时，F 为 1；否则 F 为 0。可以根据该约定列出真值表，进而完成全部设计，但该问题的逻辑关系比较清晰，只要通过简单的思考便可直接写出其输出表达式，省去不必要的设计步骤。

根据常识可知，要使 $A=B$，必须使 $A_3=B_3$、$A_2=B_2$、$A_1=B_1$，即要使 F 为 1，必须使 A 的每一位和 B 的每一位都对应相等，即同时为 0 或同时为 1。所以，F 和 A、B 的逻辑关系可用以下函数描述：

$$F=(\overline{A_3B_3}+A_3B_3)(\overline{A_2B_2}+A_2B_2)(\overline{A_1B_1}+A_1B_1)=\overline{A_3\oplus B_3\cdot A_2\oplus B_2\cdot A_1\oplus B_1}$$
$$=\overline{(A_3\oplus B_3)+(A_2\oplus B_2)+(A_1\oplus B_1)}$$

相应的电路图如图 3-23 所示。

例 3-20　设计一个判断献血者与受血者的血型是否相容的电路。血型相容规则如表 3-18 所示，表中√表示血型相容。

表 3-18　例 3-20 的血型相容规则

献血	受血			
	A	B	AB	O
A	√		√	
B		√	√	
AB			√	
O	√	√	√	√

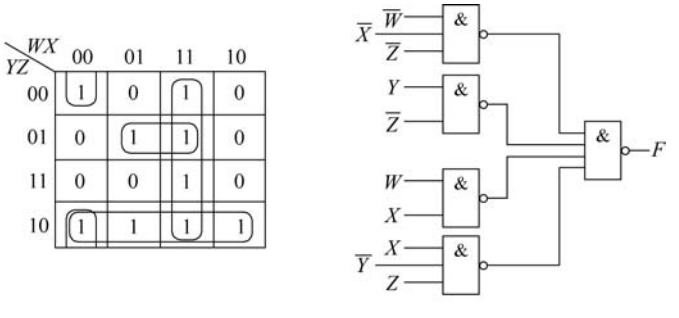
表 3-19　例 3-20 的血型编码

血型	献 WX	受 YZ
A	00	00
B	01	01
AB	10	10
O	11	11

解：根据题意，献血者和受血者的血型为电路的输入变量。4 种血型可用两个变量的 4 种编码表示。设变量 W、X 表示献血者的血型，Y、Z 表示受血者的血型，采用如表 3-19 所示的编码。电路的输出用 F 表示，当血型相容时 F 为 1；否则 F 为 0。根据血型相容规则，可直接得出函数的卡诺图，如图 3-24(a)所示。由卡诺图可得函数的最简“与或”式为

$$F=\overline{W}\overline{X}\overline{Z}+Y\overline{Z}+WX+X\overline{Y}Z$$

用“与非”门实现上述函数的逻辑电路如图 3-24(b)所示。



(a)　　(b)

图 3-24　例 3-20 的卡诺图和逻辑电路图

以上介绍了组合逻辑电路设计的一般方法。在实际设计中，往往还有许多要进一步考虑的问题，例如，何用特定的门电路实现；电路是单输出的，还是多输出的；有无无关项等。下面分别进行说明。


视频讲解

3.4.2 组合逻辑电路设计中应考虑的问题

1. 逻辑函数形式的变换

在实际设计中，有时按要求只能使用特定类型的门电路，为此就要将已获得的逻辑函数的最简“与-或”式变换成相应的形式，如“与非-与非”式“或非-或非”式“与或非”式等。这种函数形式的变换，可采用德·摩根定律和对偶规则来实现。下面举例说明。

1）逻辑函数的“与非”门实现

将最简“与-或”式变换为“与非-与非”式有两种方法：一种是对 F 两次求反，一次展开；另一种是对 $\overline{F}$ 三次求反，一次展开。

例 3-21 用“与非”门实现逻辑函数 $F=A\overline{B}+B\overline{C}+C\overline{D}+D\overline{A}$。

解：方法一：对 F 两次求反，一次展开可得

$$F=A\overline{B}+B\overline{C}+C\overline{D}+D\overline{A}=\overline{\overline{A\overline{B}}\cdot\overline{B\overline{C}}\cdot\overline{C\overline{D}}\cdot\overline{D\overline{A}}}$$

该式对应的逻辑电路如图 3-25(a)所示。

方法二：先求出 $\overline{F}$（用卡诺图法较简单），再对 $\overline{F}$ 三次求反，一次展开可得

$$\overline{F}=\overline{A\overline{B}+B\overline{C}+C\overline{D}+D\overline{A}}=\overline{A}\,\overline{B}\overline{C}\overline{D}+ABCD,\quad F=\overline{\overline{A}\,\overline{B}\overline{C}\overline{D}+ABCD}=\overline{\overline{A}\,\overline{B}\overline{C}\overline{D}}\cdot\overline{ABCD}$$

该式对应的逻辑电路如图 3-25(b)所示。

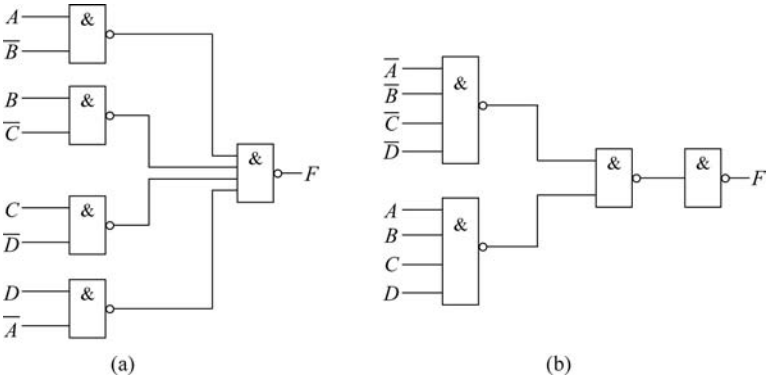


图 3-25 例 3-21“与非”门实现的逻辑电路图

可见，方法一得到的是两级电路，方法二得到的是三级电路，因此方法一所得的电路的传输速度较方法二快。当原函数比较简单时，方法一可节省门电路；当反函数比较简单时，方法二可节省门电路。另外，方法二要求“与非”门的输入端数量较多。

2）逻辑函数的“或非”门实现

将最简“与-或”式变换为“或非-或非”式，可以采用对 F 两次求对偶的方法。即先求 F 的对偶式 F_d，并将其化为最简“与非-与非”式，然后再求 F_d 的对偶式 $(F_d)_d$，则 $(F_d)_d$ 即是 F 的最简“或非-或非”式。

例 3-22 用“或非”门实现函数 $F=A\overline{B}+B\overline{C}+C\overline{A}$。

解：先求 F 的对偶式 F_d，并将其化成最简“与-或”式，即

$$F_d=(A+\overline{B})(B+\overline{C})(C+\overline{A})=\overline{A}\,\overline{B}\,\overline{C}+ABC$$

再将 F_d 的最简“与-或”式两次求反，一次展开变为“与非-与非”式，即

$$F_d=\overline{\overline{A}\,\overline{B}\,\overline{C}\cdot\overline{ABC}}$$

对 F_d 再求对偶，则得

$$F=(F_d)_d=\overline{\overline{A}+\overline{B}+\overline{C}+\overline{A+B+C}}$$

由该式可画出如图 3-26 所示的逻辑电路图。

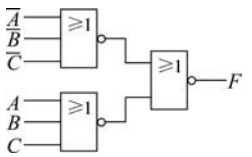


图 3-26 例 3-22“或非”门实现的逻辑图

另外，也可先求出 F 的最简“或-与”式，并将其两次取反，一次展开，即可得到最简“或非-或非”式。该方法请读者自己练习。

3）逻辑函数的“与或非”门实现

一般来说，使用“与或非”门的同时，也允许使用“非”门，所以在“与或非”式中同时也允许单独的“非”号存在。

将最简“与-或”式变换为“与或非”式也有两种方法：一种是对 F 两次求反，另一种是对 $\overline{F}$ 一次求反。

例 3-23 用“与或非”门实现函数 $F=A\overline{B}+B\overline{C}+C\overline{A}$。

解：方法一 对 F 两次求反，可得

$$F=\overline{\overline{A\overline{B}+B\overline{C}+C\overline{A}}}$$

该式对应的逻辑电路如图 3-27(a)所示。

方法二 先求 $\overline{F}$，再对 $\overline{F}$ 一次求反，可得

$$\overline{F}=\overline{A\overline{B}+B\overline{C}+C\overline{A}}=\overline{A}\,\overline{B}\,\overline{C}+ABC,\quad F=\overline{\overline{F}}=\overline{\overline{A}\,\overline{B}\,\overline{C}+ABC}$$

该式对应的逻辑电路如图 3-27(b)所示。

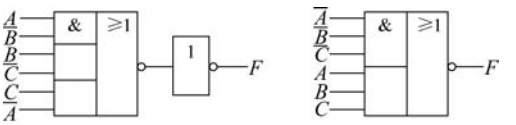


图 3-27 例 3-23“与或非”门实现的逻辑图

视频讲解

可见，方法二比方法一省一个非门，但方法二要求门电路的输入端较多。

2. 多输出组合逻辑电路的设计

就其一般步骤而言，多输出组合逻辑电路的设计与单输出组合逻辑电路的设计基本相同。关键在于设计时要把多个输出看成一个整体，进行逻辑化简时，不能孤立地以使每个函数化成最简为目标，应协调各个函数之间的关系，找出多个函数间可共享的部分，即公共项，力求使整体电路达到最简。

例 3-24 用“与非”门实现下列多输出函数：$F_1=\sum m(1,3,4,5,7)$，$F_2=\sum m(3,4,7)$。

解：如果把 F_1 和 F_2 看成两个孤立的函数，用卡诺图分别把 F_1 和 F_2 化简，则得

$$F_1=C+A\overline{B},\quad F_2=BC+A\overline{B}\,\overline{C}$$

按此结果，可画出如图 3-28(a)所示的逻辑电路图。

如果从“全局”出发，统一考虑 F_1 和 F_2 的各个与项，尽量使它们具有公共项，可将上式改为

$$F_1 = C + A\overline{B}\overline{C}, \quad F_2 = BC + A\overline{B}\overline{C}$$

按此式可画出如图 3-28(b)所示的逻辑电路图。比较两图可发现，图 3-28(b)比图 3-28(a)节省一个“与非”门，且少两条连接线。这就说明，尽管此时 F_1 已不是最简表达式，但由于它与 F_2 之间存在公共项，反而使整个电路变得更简单了。因此，多输出组合逻辑电路设计的关键是寻找公共项。

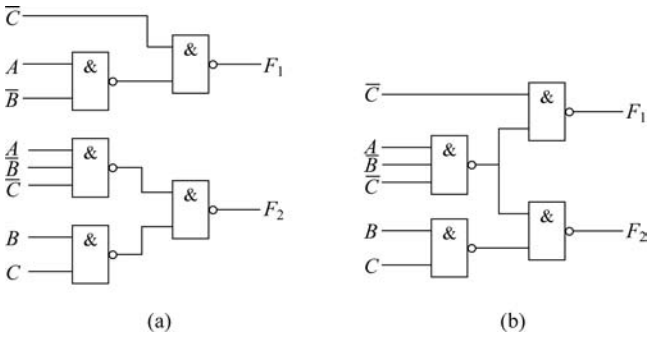


图 3-28　例 3-24 的逻辑电路图

3. 包含无关项的组合逻辑电路的设计

前面已经介绍过包含无关项的逻辑函数的化简问题，这里再举一个例子说明组合逻辑电路的设计中如何利用无关项，使设计出的电路更简单。

例 3-25　用“与非”门设计一个组合逻辑电路，用于判断 1 位余 3 码表示的十进制数是否为合数。

解：由题意可知，该电路的输入为 1 位余 3 码表示的十进制数，输出为对其值进行判断的结果。设输入十进制数的余 3 码用 A、B、C、D 表示，输出函数为 F，当输入变量的取值为合数(4,6,8,9)时输出 F 为 1；否则 F 为 0。因为按照余 3 码的编码规则，$ABCD$ 的取值组合不允许为 0000、0001、0010、1101、1110、1111，与其对应的最小项可作为无关项，对应这 6 组输入值，函数 F 的值可记为 d，表示函数 F 既可以当作 1 处理，也可以当作 0 处理。据此可得出该问题的真值表如表 3-20 所示。

表 3-20　例 3-25 的真值表

A	B	C	D	F	A	B	C	D	F
0	0	0	0	d	1	0	0	0	0
0	0	0	1	d	1	0	0	1	1
0	0	1	0	d	1	0	1	0	0
0	0	1	1	0	1	0	1	1	1
0	1	0	0	0	1	1	0	0	1
0	1	0	1	0	1	1	0	1	d
0	1	1	0	0	1	1	1	0	d
0	1	1	1	1	1	1	1	1	d

由真值表可直接写出函数的“最小项之和”表达式,即

$$F(A,B,C,D)=\sum m(7,9,11,12)+\sum d(0,1,2,13,14,15)$$

用卡诺图化简函数 F 时,若不考虑无关项(如图 3-29(a)所示),则化简后的逻辑表达式为

$$F(A,B,C,D)=A\bar{B}D+AB\bar{C}\bar{D}+\bar{A}BCD$$

如果化简时将无关项加以利用(如图 3-29(b)所示),则化简后的逻辑表达式为

$$F(A,B,C,D)=AB+AD+BCD$$

显然,后一个表达式比前一个更简单。考虑到要用“与非”门实现,故可将 F 的最简“与-或”式变换成“与非-与非”式。

$$F=\overline{\overline{AB+AD+BCD}}=\overline{\overline{AB}\cdot\overline{AD}\cdot\overline{BCD}}$$

该式对应的逻辑电路如图 3-30 所示。

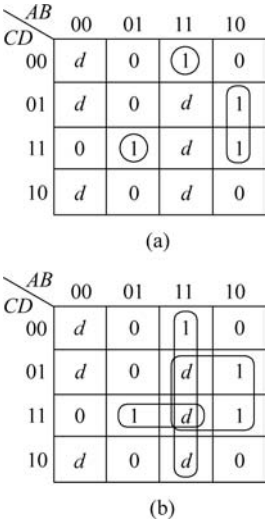


图 3-29　例 3-25 的卡诺图

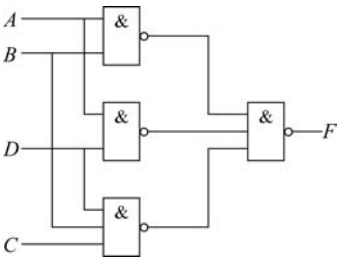


图 3-30　例 3-25 的逻辑电路图

4. 考虑级数的组合逻辑电路设计

前面所讨论的组合逻辑电路的设计都是以追求使电路最简为目标的,而从不考虑所设计的逻辑电路在速度上是否能满足应用要求,也没有考虑门电路的扇入或扇出系数是否超出了现有集成电路产品的技术指标。而实际应用中经常遇到这方面的问题,下面就讨论一下这两个问题。

当逻辑电路的级数增加时,输出相对于输入的传输延迟时间就增大,以致电路的工作速度变慢,甚至不能满足要求。此时,就要设法压缩电路的级数,使所设计的电路在满足速度要求的前提下最简单。

当电路所要求的门电路的扇入或扇出系数超出了现有器件的技术指标时,需要采用增加电路级数的办法来降低对门电路的扇入或扇出系数的要求,使所设计的电路在满足现有器件的扇入或扇出系数的前提下最简单。

然而,压缩级数和增加级数的设计思想是互斥的。一般来说,压缩逻辑电路的级数可以提高逻辑电路的速度,但却要求门电路的扇入或扇出系数增加;反之,增加电路的级数可以

降低对门电路的扇入或扇出系数的要求，但却会使电路的速度变慢。因此，在设计组合逻辑电路时，应全面考虑电路的级数问题。对单一要求的电路，可大胆地压缩级数或增加级数；对同时要满足两方面要求的电路，应采取折中的方案。

电路的级数反映在输出函数表达式中，就是与、或、非运算的层数。因此，压缩级数可以通过对输出函数求反或展开来获得。增加级数的设计一般用于多级译码电路的设计中。

例 3-26 用“与非”门、“与或非”门分别实现函数：$F=AB+\overline{A}C$。

解：对 F 两次求反可得其“与或非”形式，再进行一次展开，可得到其“与非-与非”形式。

$$F=\overline{\overline{AB+\overline{A}C}},\quad F=\overline{\overline{AB}\cdot\overline{\overline{A}C}}$$

上式对应的逻辑电路分别如图 3-31(a)、(b)所示。

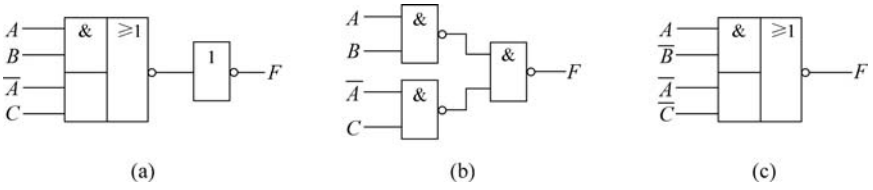


图 3-31 例 3-26 的逻辑电路图

如果先求出 $\overline{F}$ 的最简“与-或”式，再对 $\overline{F}$ 求反，可使 F 的级数减少。

$$\overline{F}=\overline{AB+\overline{A}C}=A\overline{B}+\overline{A}\,\overline{C},\quad F=\overline{A\overline{B}+\overline{A}\,\overline{C}}$$

上式对应的逻辑电路如图 3-31(c)所示。

假设“与非”门和“非”门的平均传输延迟时间为 t_{pd}，“与或非”门为 $1.5t_{pd}$，那么图 3-31(a)由一个“与或非”门和一个“非”门组成，传输延迟时间为 $2.5t_{pd}$；图 3-31(b)由两级“与非”门组成，传输延迟时间为 $2t_{pd}$；图 3-31(c)由一个“与或非”门组成，传输延迟时间为 $1.5t_{pd}$。

比较后发现，如图 3-31(c)所示的电路级数最少。这种先对 F 求反，并求出 $\overline{F}$ 的最简“与-或”式，再对 $\overline{F}$ 求反来压缩电路级数的方法，称为求反压缩法。但这种方法仅适用于反函数比较简单的情况。

3.5 VHDL 描述基础

前面学习的组合逻辑电路设计方法是基于门电路的经典手工设计方法，效率低下。随着可编程逻辑器件的广泛应用，数字系统的设计方法出现了极大的变革，用户已经能够像设计软件一样通过编写程序的方法设计硬件电路了，从而实现硬件电路设计的高度自动化，缩短硬件设计周期，减少硬件设计成本，这就是通常所说的电子设计自动化(Electronic Design Automation，EDA)。在这种设计方法中，设计者的工作仅限于用硬件描述语言对所设计电路的功能进行描述，在 EDA 软件的帮助下得到最终设计结果，尽管设计目标是硬件，但整个设计和修改过程如同完成软件设计一样方便和高效。本节学习这种设计方法中的硬件描述语言。

视频讲解

3.5.1 VHDL 概述

硬件描述语言(Hardware Description Language，HDL)是一种对硬件电路进行性能描

述和模拟的语言。相对于传统的原理图设计方法,硬件描述语言最大的优势在于可以借鉴高级语言程序设计的功能特性对硬件电路的行为和功能结构进行高度抽象化的描述。同时,硬件描述语言还可以对硬件电路的设计进行不同层次、不同领域的模拟验证和综合优化处理,从而实现硬件电路设计的高度自动化。常用的硬件描述语言有 ABEL、AHDL、Verilog HDL 和 VHDL 等,其中以 Verilog HDL 和 VHDL 最为流行,本书选择 IEEE 标准硬件描述语言(VHDL)对所学电路进行描述。由于本书不是专门介绍 VHDL 的书籍,限于篇幅,只介绍 VHDL 的基本知识,以该课程及相关后续课程够用为原则,关于 VHDL 更详尽的内容请参考相关书籍。根据笔者多年的教学经验,考虑了教学过程中学生的接受能力,在章节安排上,没有把对 VHDL 的介绍集中在一起进行,而是融入不同章节及相关的设计实例中进行介绍,这样既避免了集中学习语言的单调,又因与具体实例相结合而容易接受,同时也可以使读者在学习了相关电路基础之后较早地接触 VHDL 描述。另外,为便于设计时查阅,本书最后以附录的形式给出了 VHDL 的基本语句和相关设计实例。本节介绍 VHDL 的基础知识以及用 VHDL 描述组合逻辑电路的方法,时序逻辑电路的 VHDL 描述将在后面相关章节进行介绍。

1. VHDL 的产生与发展

自从 20 世纪 70 年代硬件描述语言产生以来,众多 EDA 公司和科研单位纷纷研制开发了适应自身 EDA 开发工具的硬件描述语言。这些硬件描述语言具有很大的差异,并且只能在自己公司的 EDA 工具上使用,这大大限制了硬件描述语言的使用。因此硬件电路设计人员需要一种强大的面向设计的多层次、多领域并得到广大 EDA 厂商认同的标准硬件描述语言。

20 世纪 70 年代末美国国防部提出了“超高速集成电路”(Very High Speed Integrated Circuits,VHSIC)计划,研究一种新的硬件描述语言是该项目的一个需要。1981 年新的语言研究完成,取这个计划名称的第一个字母 V,命名该硬件描述语言为 VHDL。

VHDL 原本只是美国国防部的一种标准,经过反复修改和扩充后于 1987 年被 IEEE 协会接受,成为硬件描述语言的标准,即 IEEE Std1076-1987,也就是通常所说的 VHDL-87。1993 年 IEEE 进一步对其进行修改,发布 IEEE Std1076-1993,也就是大家所说的 VHDL-93。VHDL 成为标准以后,很快在世界各地得到了广泛的应用,为电子设计自动化的普及和推广奠定了坚实的基础。目前标准的版本还在不断完善和更新中。

2. VHDL 的特点

VHDL 能够成为标准化的硬件描述语言并获得广泛应用,其自身必然具备很多其他硬件描述语言所不具备的优点。归纳起来,VHDL 主要有以下基本特点。

(1) 可以在各个不同的设计阶段对系统进行描述。使用 VHDL 可以在不同的设计阶段对系统进行比较抽象的性能描述,也可以进行比较具体的数据流描述和更加具体的逻辑结构描述。

(2) 支持层次化的设计方法。系统硬件的设计经常采用自顶向下的层次化设计方法,从系统的性能要求出发,将设计任务逐层分解、细化和实现。VHDL 的模块化结构,完全支持这种设计方法。

(3) 设计描述与器件无关。由于 VHDL 可以根据需要建立不同的单元库,因此 VHDL 硬件描述与具体的工艺技术和硬件结构无关,其硬件实现所选用的目标器件有广泛的选择

范围，其中包括各种系列的 CPLD、FPGA，以及各种门阵列器件等。设计者可以不懂硬件的结构，也不必首先考虑器件的选择，而是集中精力进行电路设计的优化，当硬件电路的设计描述完成以后，VHDL 允许采用多种不同的器件结构来实现。

(4) 程序易于共享和复用。大型硬件电路的设计不可能从门级电路开始一步步地进行设计，而应该由一些已有模块和一些新设计的模块组成。VHDL 采用基于库(library)的设计方法，在设计过程中，设计人员可以建立各种可以再次利用的模块，将这些模块存放在库中，以便在以后的设计中进行复用。

(5) 具有很强的移植能力。因为 VHDL 是一种标准语言，所以用 VHDL 描述的硬件电路可以被不同的工具所支持，可以从一个模拟工具移植到另一个模拟工具、从一个综合工具移植到另一个综合工具或者从一个工作平台移植到另一个工作平台上去执行。

3.5.2 VHDL 描述的基本结构

一个完整的 VHDL 描述通常包括库(library)、程序包(package)、实体(entity)、结构体(architecture)和配置(configuration)5 个部分。其中实体和结构体是必需的，而库、程序包和配置不是必需的，一般可以根据设计的需要来添加。下面以一个简单的二输入与非门的 VHDL 描述为例来说明实体和结构体的作用。

例 3-27 二输入与非门的 VHDL 描述。

```
ENTITY nand_2 IS                    -- 实体描述
   PORT(a,b:IN BIT;                 -- 输入信号
        f:OUT BIT);                 -- 输出信号
END nand_2;
ARCHITECTURE rtl OF nand_2 IS       -- 结构体描述
BEGIN
   f <= a NAND b;                   -- 电路功能描述
END rtl;
```

如上程序由实体和结构体两部分组成。实体部分用于描述电路的外部接口信号，如本例中的电路包含两个输入信号 a 和 b，一个输出信号 f。结构体对电路的内部结构和性能进行具体描述，如本例中通过语句 f <= a NAND b 将电路的功能具体描述为与非功能，即 $f=\overline{ab}$。

从本例还可以看出，每个 VHDL 语句都是以分号(;)结束的，注释以两个减号(--)开始。


视频讲解

1. 实体

实体类似于原理图中的一个部件符号，它并不描述设计的具体功能，只是定义该设计所需的全部输入/输出信号。实体的一般格式如下，[]中的内容是可选内容，为增加可读性建议不要省略。

```
ENTITY 实体名 IS
    PORT (信号名: 信号类别 信号类型;
                ⋮
          信号名: 信号类别 信号类型);
END [实体名];
```

实体描述主要由 PORT 部分组成。PORT 的中文是“端口”的意思,也就是说该部分用于说明电路或系统的对外连接。实体名和信号名都是用户自定义的标识符,相同类别和类型的信号可以用逗号分开,放在一个语句行中说明,如例 3-27 中的输入信号 a 和 b。

信号类别主要有以下 4 种。

(1) IN:进入实体的输入信号,注意不能给输入信号赋值。

(2) OUT:离开实体的输出信号,注意输出信号不能在内部反馈使用,即不能读取输出信号的数据。

(3) INOUT:双向信号,既可以进入实体,也可以离开实体。

(4) BUFFER:缓冲信号,也是实体的输出信号,但同时可以在实体内部反馈。BUFFER 是 INOUT 的子集,但不能由外部驱动。BUFFER 主要用于构成带反馈的逻辑电路,一般的组合逻辑电路是没有反馈的,但是后面几章要学习的时序电路是有反馈的。

信号类别可以用图 3-32 说明,图中的方框代表一个设计或模块。

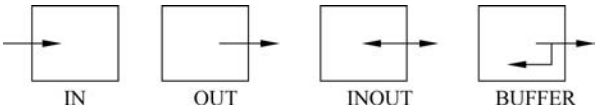


图 3-32 端口类别说明

信号类型可以是预定义的,也可以是用户自定义的。预定义类型是 VHDL 中最常用、最基本的数据类型,是一种在已有程序包中预先定义好的数据类型,包括 VHDL 标准包 STANDARD 中预定义的数据类型、IEEE 库相关程序包中预定义的数据类型和其他程序包中预定义的数据类型。用户可以通过 IEEE 和 STD 库调用标准程序包,使用这些数据类型。

VHDL 在 STD 库的标准包 STANDARD 中预定义的常用数据类型有如下几种。

(1) BIT:二进制位类型,信号的值只能是 0 或 1。

(2) BIT_VECTOR:二进制位矢量类型,是由多位 BIT 类型构成的二进制位向量。

(3) BOOLEAN:布尔类型,取值只能是 true 或 false。

(4) INTEGER:整型,32 位二进制数表示的整型,取值范围是 $2^{31}-1\sim-2^{31}$。

(5) CHARACTER:字符型,8 位二进制编码的 ASCII 字符。

以上这些数据类型的定义包含在 STD 库的标准包 STANDARD 中,用户可以直接使用,不需要另外声明。

在 IEEE 库的程序包 STD_LOGIC_1164 中,还定义了两个非常重要的数据类型,它们分别是标准逻辑位类型 STD_LOGIC 和标准逻辑位矢量类型 STD_LOGIC_VECTOR。在 VHDL 的设计中经常用到这两个数据类型。使用时可以通过下列语句打开 IEEE 库,调用 STD_LOGIC_1164 程序包。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
```

(1) STD_LOGIC 类型。STD_LOGIC 是用 IEEE 1164 标准定义的一种常用的 9 值逻辑位数据类型,是 BIT 数据类型的扩展。

STD_LOGIC 类型有 9 种逻辑值：U(未初始化的)、X(强未知的)、0(强 0)、1(强 1)、Z(高阻态)、W(弱未知的)、L(弱 0)、H(弱 1)、_(忽略)。

注意：STD_LOGIC 中的数据是用大写字母定义的，不能使用小写字母。

(2) STD_LOGIC_VECTOR 类型。STD_LOGIC_VECTOR 是基于 STD_LOGIC 数据类型的一维标准逻辑数组，常用于描述数字系统中的总线。数组中每个元素的数据类型都应是 STD_LOGIC 中定义的逻辑值。

VHDL 是强类型语言，不允许将一种信号类型赋值给另一种不同的信号类型。若要对不同类型的信号进行赋值，则需要进行强制类型转换，基本类型转换函数如表 3-21 所示。

表 3-21　常见数据类型转换函数表

所在程序包	函　数　名	功　　能
STD_LOGIC_1164	to_stdlogicvector(*a*)	由 BIT_VECTOR 转换为 STD_LOGIC_VECTOR
	to_bitvector(*a*)	由 STD_LOGIC_VECTOR 转换为 BIT_VECTOR
	to_stdlogic(*a*)	由 BIT 转换为 STD_LOGIC
	to_bit(*a*)	由 STD_LOGIC 转换为 BIT
STD_LOGIC_ARITH	conv_std_logic_vector(*a*,len)	由 INTEGER 转换为 STD_LOGIC_VECTOR。*a* 为整数，len 为位长
	conv_integer(*a*)	由 UNSIGNED、SIGNED 转换为 INTEGER
STD_LOGIC_UNSIGNED	conv_integer(*a*)	由 STD_LOGIC_VECTOR 转换为 INTEGER

下面给出一个使用预定义类型转换函数的示例，该示例将 BIT_VECTOR 类型的输入信号相与后经 TO_STDLOGICVECTOR 进行类型转换，赋值给 STD_LOGIC_VECTOR 类型的输出信号 *q*。

例 3-28　类型转换函数的示例。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY typeconvert IS
    PORT(a,b:IN BIT_VECTOR(3 DOWNTO 0);
         q:OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END typeconvert;
ARCHITECTURE rtl OF typeconvert IS
BEGIN
    q <= TO_STDLOGICVECTOR(a AND b);
END rtl;
```


视频讲解

2. 结构体

实体描述电路的对外接口，而结构体则用来描述电路的内部操作，即描述实体实现的功能。结构体由声明部分和功能描述部分组成，其一般格式如下：

```
ARCHITECTURE 结构体名 OF 实体名 IS
    [结构体声明部分]
BEGIN
    [功能描述部分]
END [结构体名];
```

注意：结构体名和实体名是相互联系的，“结构体名 OF 实体名”的含义是“实体的一个结构体为……”。对于一个设计实体，可以有几种不同的结构体实现，一个结构体名只表示实体的一种实现描述，还可以用另一个结构体名表示实体的另一种实现描述。

结构体声明部分位于 ARCHITECTURE 和 BEGIN 之间，用于定义结构体中用到的数据对象和子程序，并对所引用的元件加以说明，即对结构体的功能描述部分所用到的信号(SIGNAL)、类型(TYPE)、常数(CONSTANT)、元件(COMPONENT)、函数(FUNCTION)和过程(PROCEDURE)等加以说明和定义，但不能定义变量。

功能描述部分用于描述实体的逻辑行为。VHDL 允许采用三种描述方式，即数据流描述(Dataflow Description)、行为描述(Behavioral Description)和结构描述(Structural Description)。这三种描述方式从不同的角度对设计实体的行为和功能进行了描述，各有特点。VHDL 还允许采用混合描述方式。

1) 数据流描述

数据流描述也称为寄存器传输(Register Transfer Level，RTL)描述，是以类似于寄存器传输级的方式描述数据的传输和变换的，是对信号传输的数据流路径形式进行的描述，简单地说，数据流描述就是利用 VHDL 中的赋值符和逻辑运算符对电路进行的描述，如例 3-27 中对二输入与非门的描述，就用了信号赋值语句“f<=a NAND b;”，其中 NAND 为“与非”运算符。数据流描述以并行赋值语句为基础，当语句中的任一输入信号值发生变化时，赋值语句将被激活，使信息从所描述的结构中“流出”。

2) 行为描述

行为描述依据设计实体的功能或算法对结构体进行描述，不需要给出实现这些行为的硬件结构，只强调电路的行为和功能。行为描述主要用函数、过程和进程语句，以功能或算法的形式描述数据的转换和传送。下面给出二输入与非门的行为描述。

例 3-29 二输入与非门的 VHDL 行为描述法。

```
ENTITY nand_2 IS
   PORT(a,b:IN BIT;
        f:OUT BIT);
END nand_2;
ARCHITECTURE behavior OF nand_2 IS
BEGIN
    PROCESS(a,b)                                  -- 进程语句
        VARIABLE tmp: BIT_VECTOR(1 DOWNTO 0);  -- 变量定义
    BEGIN
        tmp: = a&b;          -- a 和 b 连接成位向量,再赋给 tmp,也可以用 tmp: = (a,b);
        CASE tmp IS          -- 用 CASE 语句描述电路的功能,类似真值表
            WHEN "00" = > f < = '1';
            WHEN "01" = > f < = '1';
            WHEN "10" = > f < = '1';
            WHEN "11" = > f < = '0';
        END CASE;
    END PROCESS;
END behavior;
```

从程序中的 CASE 语句可以看出，行为描述非常类似于前面学习的真值表描述，这里

只有当两个输入 a 和 b 都为 1 时，输出 f 才为 0，其他情况下输出 f 均为 1。


视频讲解

行为描述中一定要包含 PROCESS(进程)语句。PROCESS 语句是在结构体中描述特定电路功能的程序模块，它提供了一种用顺序语句描述电路逻辑功能的方法。所谓顺序语句是指执行顺序和书写顺序一致的语句，顺序语句一定要放在 PROCESS 语句中。一个结构体中可以有多个进程语句，这些进程语句并行执行，即执行顺序与书写的先后无关，只要触发条件满足就可以执行。PROCESS 语句的格式为

```
[进程标号:] PROCESS [(敏感信号列表)] [IS]
    [进程声明部分]
BEGIN
    顺序描述语句
END PROCESS [进程标号];
```

每个进程语句都可以有一个可选的进程标号，作为进程的名称。进程语句从 PROCESS 开始到 END PROCESS 结束。

敏感信号是触发进程执行的信号，当敏感信号列表中的某个信号发生变化时，立即启动 PROCESS 语句，按顺序执行进程中的语句，直到敏感信号稳定不变为止。如例 3-29 中的敏感信号为与非门的输入信号 a 和 b，只要 a 或 b 的电平发生变化，就触发 PROCESS 语句的执行并产生正确的输出信号 f，这与硬件电路的特征是相符的。

程序中的 tmp 是 PROCESS 语句中用到的变量(VARIABLE)。运算符 & 是连接运算符，可以将多个数据元素合并成一个新的一维数组(向量)。这里是将两个 BIT 类型的信号 a 和 b 连接成位宽为 2 的一维数组，再赋值给变量 tmp。也可以先将信号 a 和 b 放到一对括号里，组成向量再赋值给变量 tmp，即用语句“tmp:=(a,b);”实现同样的功能。注意单个取值的常量用单引号，而数组常量要用双引号。

在 VHDL 中变量为局部量，用来存储中间数据，以便实现程序的算法。变量只能在进程、函数和过程内部使用。变量的赋值立即生效，不存在任何延时。变量的定义格式如下：

```
VARIABLE 变量名: 数据类型 [:=初始值];
```

如

```
VARIABLE a:INTEGER:=0;
```

变量赋值语句的格式为

```
变量名:=表达式;
```

如

```
a:=b+c;
```

与变量容易混淆的是信号(SIGNAL)。信号是全局量，在实体说明、结构体描述和程序包说明中使用。SIGNAL 用于声明内部信号，而非外部信号(外部信号对应为 IN、OUT、INOUT、BUFFER，在 PORT 语句中声明)，在元件之间起互连作用，内部信号的值可以赋值给外部信号。例如下面的例 3-30 中的 out1 就是用 SIGNAL 定义的中间信号。信号的定义格式为

```
SIGNAL 信号名: 数据类型 [:=初始值];
```

如SIGNAL count: STD_LOGIC_VECTOR(3 DOWNTO 0) :="1000";表示信号count是4位标准逻辑矢量,从高位(第3位)到低位(第0位)的初始值为1000。也可以用关键字TO从低位到高位定义矢量信号,如"SIGNAL count: STD_LOGIC_VECTOR(0 TO 3) :="0001";"。

信号赋值语句的格式为

```
信号名<=表达式;
```

如

```
q<=count;
```

信号与变量的比较如下。

(1) 声明的形式与位置不同:信号声明用SIGNAL,变量声明用VARIABLE。信号声明在子程序、进程等的外部,而变量声明在子程序、进程等的内部。

(2) 赋值符号和赋值起作用的时刻不同:信号用<=赋值,而变量用:=赋值。在进程中,信号的赋值在进程结束时才起作用,而对变量的赋值是立刻起作用的。

下面举例说明信号和变量的区别。

例3-30 信号和变量的区别。

程序一:使用信号的情况。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY xor_sig IS
    PORT (a,b,c:IN STD_LOGIC;
          x,y:OUT STD_LOGIC);
END xor_sig;
ARCHITECTURE sig_arch OF xor_sig IS
    SIGNAL d: STD_LOGIC;
BEGIN
    sig:PROCESS(a,b,c)
    BEGIN
        d<=a;              --编译时将被忽略
        x<=c XOR d;
        d<=b;              --最后一次的赋值起作用
        y<=c XOR d;
    END PROCESS;
END sig_arch;
```

程序二:使用变量的情况。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY xor_var IS
    PORT (a,b,c:IN STD_LOGIC;
          x,y:OUT STD_LOGIC);
END xor_var;
ARCHITECTURE var_arch OF xor_var IS
BEGIN
```

```
    var:PROCESS(a,b,c)
        VARIABLE d: STD_LOGIC;
    BEGIN
        d := a;              -- 变量赋值,立刻起作用
        x <= c XOR d;
        d := b;
        y <= c XOR d;
    END PROCESS;
END var_arch;
```

程序一中两次对信号 d 进行了赋值，但由于对信号的赋值只有在进程结束时才起作用，因此，第一次的赋值将被忽略，第二次的赋值才真正起作用。程序实现的功能是：$x=c\oplus b$，$y=c\oplus b$。

程序二中因为对变量的赋值是立刻生效的，所以第一次赋值后 $d=a$，第二次赋值后 $d=b$。程序实现的功能是：$x=c\oplus a$，$y=c\oplus b$。

例 3-29 中的 CASE 语句是 VHDL 中的流程控制语句，属于顺序描述语句。CASE 语句用于两路或多路分支的判断结构，其判断条件为多值表达式，根据表达式的取值不同实现多路分支。CASE 语句的格式为

```
CASE 表达式 IS
    WHEN 值 1 => 顺序语句 1;
    WHEN 值 2 => 顺序语句 2;
        ⋮
    WHEN 值 k => 顺序语句 k;
    [WHEN OTHERS => 顺序语句 k+1; ]
END CASE
```

执行该语句时，先计算表达式的值，然后选择与表达式值相同的分支，去执行它所对应的顺序语句。当表达式的值与所有的值都不相同时，则执行 OTHERS 后面的顺序语句。

例 3-29 中，因为 a 和 b 是只有 0 和 1 两种取值的 BIT 类型，所以变量 tmp 只有 4 种取值组合，程序中给出了全部取值组合情况，因此就不需要 OTHERS 分支了。如果程序中将 a 和 b 定义为有 9 值逻辑的 STD_LOGIC 类型，则可将最后一个分支“WHEN "11"=>f<='0'”改为“WHEN OTHERS =>f<='0'”，或 4 个分支不变，最后再增加一个分支“WHEN OTHERS =>f<='X'”。

与 CASE 语句同属流程控制语句（通过条件控制来决定是否执行某一条或多条语句）的还有 IF 语句。IF 语句是一种条件语句，在 IF 语句中至少应有一个条件句，该条件句必须由 BOOLEAN 型表达式构成。IF 语句依据条件产生的判断结果 TRUE 或 FALSE，有选择地去执行指定的语句。利用 IF 语句可以实现两个或两个以上的条件分支判断，其格式有三种。

格式一：单选择控制。

```
IF 条件句 THEN
    顺序语句;
END IF;
```

当条件句的逻辑值为 TRUE 时，则执行 THEN 后面的顺序语句，否则结束该语句的执行。

格式二：二选择控制。

```
IF 条件句 THEN
    顺序语句;
ELSE
    顺序语句;
END IF;
```

当条件句的逻辑值为 TRUE 时,则执行 THEN 后面的顺序语句,否则执行 ELSE 后面的顺序语句。

格式三:多选择控制。

```
IF 条件句 THEN
    顺序语句;
ELSIF 条件句 THEN
    顺序语句;
    ⋮
ELSE
    顺序语句;
END IF;
```

当满足多个条件之一时,执行该条件 THEN 后面的顺序语句;如果所有条件都不满足,则执行 ELSE 后面的顺序语句。注意关键字 ELSIF 不要写成 ELSEIF。

注意:虽然 CASE 语句和 IF 语句同属流程控制语句,都能实现多分支的选择,但 CASE 语句的分支是无序的,所有表达式的值都是并行处理的,分支书写顺序可以随意。而 IF 语句的条件判断是有序的,先处理最起始优先的条件,后处理次优先条件,条件的优先顺序由书写顺序确定。

下面给出用 IF 语句实现的二输入与非门的 VHDL 描述。

```
ENTITY nand_2 IS
    PORT(a,b:IN BIT;
         f:OUT BIT);
END nand_2;
ARCHITECTURE behavior OF nand_2 IS
BEGIN
    PROCESS(a,b)
        VARIABLE tmp: BIT_VECTOR(1 DOWNTO 0);
    BEGIN
        tmp: = a&b;
        IF (tmp = "00") THEN f < = '1';
        ELSIF (tmp = "01") THEN f < = '1';
        ELSIF (tmp = "10") THEN f < = '1';
        ELSE f < = '0';
        END IF;
    END PROCESS;
END behavior;
```

视频讲解

3) 结构描述

结构描述是以元件(COMPONENT)为基础,通过描述元件和元件之间的连接关系,来反映整个系统的构成和性能的。例如,二输入与非门可以看成由一个二输入与门和一个非门构成的两级系统,下面给出二输入与非门的 VHDL 结构描述方法。

例 3-31 二输入与非门的 VHDL 结构描述法。

```
ENTITY nand_2 IS
    PORT(a,b:IN BIT;
         f:OUT BIT);
END nand_2;
ARCHITECTURE struct OF nand_2 IS
    COMPONENT inv                              -- 非门元件声明
      PORT(ain:IN BIT;
           fout:OUT BIT);
    END COMPONENT;
    COMPONENT and_2                            -- 二输入与门元件声明
      PORT(ain,bin:IN BIT;
           fout:OUT BIT);
    END COMPONENT;
    SIGNAL out1:BIT;                           -- 中间信号
BEGIN
    u1:and_2 PORT MAP(a,b,out1);               -- 二输入与门元件例化
    u2:inv PORT MAP(ain=>out1,fout=>f);        -- 非门元件例化
END struct;
```

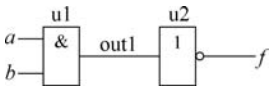


图 3-33 例 3-31 对应的逻辑电路图

在上面的描述过程中，需要用到已生成的模块，即 and_2(二输入与门)和 inv(非门)，在编译本例前需要先完成这两个模块的设计，请读者自己完成。COMPONENT 语句指定了已生成的模块，供结构体调用。用 PORT MAP 语句将生成的模块与所设计的各模块(u1、u2)联系起来，并定义相应的信号，以表示所设计各模块之间的连接关系。为便于理解，图 3-33 给出了该设计对应的逻辑电路图，其中中间信号 out1 为与门的输出。

COMPONENT 语句(元件声明语句)的格式如下：

```
COMPONENT 元件名
    PORT (信号名: 信号类别 信号类型;
          ⋮
          信号名: 信号类别 信号类型);
END COMPONENT;
```

可见，元件声明与实体定义基本类似，只要将实体定义中的 PORT 部分复制过来就可以了。

COMPONENT 语句只是对要调用的元件进行的声明，元件的具体调用是由元件例化语句实现的。元件例化语句说明该元件与当前设计实体是如何连接的。元件例化语句的格式如下：

```
标号: 元件名 PORT MAP(信号名 1,信号名 2,…,信号名 n);
```

或

```
标号: 元件名 PORT MAP(接口信号 1=>信号名 1,…,接口信号 n=>信号名 n);
```

标号是必不可少的，可以将标号理解为这个元件在电路中的名称。因为，同一个元件可

以在电路中多次使用,它们都有相同的元件名,但要有不同的标号,或者说在电路中具有不同的名称。

PORT MAP 部分说明本设计实体的信号与元件声明中信号的对应关系。第一种格式是通过书写顺序来表示它们之间的对应关系,如上例中的 u1。第二种格式是直接给出各信号的对应关系,“接口信号”表示元件中的信号,“信号名”则是加到当前实体的实际信号的名称,如上例中的 u2。

从上例可以看出,利用结构描述方式可以进行多层次的结构化设计,首先将一个大的设计划分成若干小的模块,逐一完成设计、编译、调试,然后再利用结构化描述将这些模块组装起来,形成更为复杂的设计,这样的描述,其结构非常清晰,并能做到与原理图中所画的器件一一对应。

3. 库和程序包

根据 VHDL 语法规则,在 VHDL 程序中使用的文字、数据对象、数据类型都需要预先定义。为了方便 VHDL 编程和提高编程效率,可以将预先定义好的数据类型、元件调用声明及一些常用子程序汇集在一起,形成程序包,供 VHDL 设计实体共享和调用,若干程序包则形成库。

1) VHDL 库

常用 VHDL 库有 IEEE 标准库、STD 库、WORK 库和用户自定义库。

IEEE 标准库包括 STD_LOGIC_1164 程序包和 STD_LOGIC_ARITH 程序包。其中 STD_LOGIC_ARITH 程序包是 Synopsys 公司加入 IEEE 标准库的程序包,其中包括 STD_LOGIC_SIGNED(有符号数)和 STD_LOGIC_UNSIGNED(无符号数)等程序包。STD_LOGIC_116 是最重要和最常用的程序包,大部分数字系统设计都是以此程序包设定的标准为基础的。

STD 库包含 STANDARD 和 TEXTIO 程序包,这两个程序包是文件输入/输出程序包,在 VHDL 的编译和综合过程中,系统都需要调用这两个程序包中的内容。用户在进行数字系统设计时,STANDARD 程序包可以直接调用,TEXTIO 程序包需要用 LIBRARY 语句和 USE 语句声明。

WORK 库是用户设计的现行工作库,用户在项目设计中将设计成功、正在验证、未仿真的中间件都堆放在 WORK 中。在 PC 或工作站上利用 VHDL 进行项目设计时,不允许在根目录下进行,必须在根目录下为设计建立一个工程目录(即文件夹),VHDL 综合器将此目录默认为 WORK 库。VHDL 标准规定 WORK 库总是可见的,不需要明确指定。

用户自定义库是用户根据需要将自主开发的程序包和实体汇集在一起形成的库,用户自定义库在使用时需要用 LIBRARY 语句和 USE 语句声明。

LIBRARY 语句和 USE 语句的格式如下:

```
LIBRARY 库名;
USE 库名.程序包名.项目名; 或   USE 库名.程序包名.ALL;
```

第一种 USE 语句用于打开指定库中特定程序包内所选定的项目,第二种 USE 语句用于打开指定库中特定程序包内的所有内容。

2) VHDL 程序包

在设计实体中声明的数据类型、子程序和数据对象对于其他设计实体是不可见的。为了使已声明的数据类型、子程序、元件等能被其他设计实体调用或共享,可以把它们汇集在程序包中。

程序包由包首和包体两部分组成，格式如下：

```
PACKAGE 程序包名 IS                                    -- 包首
    包首说明部分;
END [PACKAGE] [程序包名];
PACKAGE BODY 程序包名 IS                               -- 包体
    包首说明和实现部分;
END [PACKAGE BODY] [程序包名];
```

包首为程序包定义接口，声明程序包中的数据类型、常量、元件、函数、过程等。包体给出包中函数、过程等的具体实现。如果只是定义数据类型等内容，则可以没有包体，只在包首定义即可。

下面给出一个在当前 WORK 库中定义和使用程序包的例子。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
PACKAGE my_pkg IS
    COMPONENT nd2 IS
        PORT (a,b:IN STD_LOGIC;
              c:OUT STD_LOGIC);
    END COMPONENT;
    COMPONENT latch1 IS
        PORT (d,ena:IN STD_LOGIC;
              q:OUT STD_LOGIC);
    END COMPONENT;
    FUNCTION max(a,b:IN STD_LOGIC_VECTOR)
        RETURN STD_LOGIC_VECTOR;
END my_pkg;
PACKAGE BODY my_pkg IS
    FUNCTION max(a,b:IN STD_LOGIC_VECTOR)
        RETURN STD_LOGIC_VECTOR IS
    BEGIN
        IF (a>b)THEN RETURN a;
        ELSE RETURN b;
        END IF;
    END max;
END my_pkg;
```

程序包 my_pkg 的包头中包含一个二输入与非门 nd2 元件声明、一个一位锁存器 latch1 元件声明和一个求最大值函数 max 的声明。在 my_pkg 的包体部分给出了 max 函数的函数体声明。由于程序包也是用 VHDL 编写的，所以其源程序也需要以.vhd 类型保存。下面是引用程序包 my_pkg 的例子，该例中对 max 函数进行了调用，为了使用 my_pkg 程序包中声明的内容，在设计实体的开头需要用 USE 语句将其打开。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.my_pkg.ALL;
ENTITY ex_ my_pkg IS
  PORT (a,b:IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        f:OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
```

```
END ex_my_pkg;
ARCHITECTURE rtl OF ex_my_pkg IS
BEGIN
    f <= max(a,b);
END rtl;
```

4. 配置

一个实体可以用多个结构体描述,在具体综合时选择哪一个结构体,则由配置来确定。设计者可以用配置语句为实体选择不同的结构体。配置语句的格式为

```
CONFIGURATION 配置名 OF 实体名 IS
   [说明语句; ]
   FOR 选配的结构体名                          -- 注意此处无分号(;)
   END FOR;
END [CONFIGURATION];
```

下面给出一个2位相等比较器的设计实例,该例用了4种不同的描述方式实现,即有4个不同的结构体。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY equ2 IS
   PORT (a,b:IN STD_LOGIC_VECTOR(1 DOWNTO 0);
         equ:OUT STD_LOGIC);
END equ2;
-- 结构体一: 用结构描述方式(元件例化)实现
ARCHITECTURE netlist OF equ2 IS
   COMPONENT nor_2
      PORT(a,b:IN STD_LOGIC;
           c:OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT xor_2
      PORT(a,b:IN STD_LOGIC;
           C:OUT STD_LOGIC);
   END COMPONENT;
SIGNAL x:STD_LOGIC_VECTOR(1 DOWNTO 0);
BEGIN
   u1:xor_2 PORT MAP(a(0),b(0),x(0));
   u2:xor_2 PORT MAP(a(1),b(1),x(1));
   u3:nor_2 PORT MAP(x(0),x(1),equ);
END netlist;
-- 结构体二: 用行为描述方式实现,采用并行语句
ARCHITECTURE con_behavior OF equ2 IS
BEGIN
  equ <= '1' WHEN a = b ELSE
         '0';
END con_behavior;
-- 结构体三: 用行为描述方式实现,采用顺序语句
ARCHITECTURE seq_behavior OF equ2 IS
```

```
BEGIN
   PROCESS(a,b)
   BEGIN
      IF a = b THEN equ <= '1';
      ELSE equ <= '0';
      END IF;
   END PROCESS;
END seq_behavior;
-- 结构体四: 用数据流描述方式(布尔方程)实现
ARCHITECTURE equation OF equ2 IS
BEGIN
  equ <= (a(0) XOR b(0))AND (a(1) XOR b(1));
END equation;
```

本例中,实体 equ2 有 4 个结构体: netlist、con_behavior、seq_behavior、equation,要生成该相等比较器 aequb,实体究竟应该对应于哪一个结构体,可以通过配置语句灵活地选择,如选用结构体 netlist,则配置如下:

```
CONFIGURATION aequb OF equ2 IS
   FOR netlist
   END FOR;
END CONFIGURATION;
```

如选用结构体 equation,只需将配置修改为

```
CONFIGURATION aequb OF equ2 IS
   FOR equation
   END FOR;
END CONFIGURATION;
```

3.5.3 VHDL 的标识符和保留字

1. 标识符

标识符(identifier)是设计人员为书写程序所定义的一些词,用来表示常数、变量、信号、端口、子程序、实体和结构体等的名字。

VHDL 的标识符由英文字母 a～z、A～Z、数字 0～9 以及下画线(_)组成。使用时应注意以下几点:

(1) VHDL 不区分大小写;

(2) 标识符一定要以字母开头;

(3) 下画线不能连用,也不能放在结尾;

(4) VHDL 的保留字不能用作标识符。

如 a_h_1、show_new_state、COUNTER_A 等都是有效的标识符,而 a%h_1、show-new-state、COUNTER_、T_ _1、2nand 等都是非法标识符。另外,为便于阅读和理解,标识符应具有一定的含义,如 half_adder 表示半加器,nand_2 表示二输入与非门,max4_1 表示四选一的多路选择器等。虽然 VHDL 不区分大小写,但为了区别,本书中的保留字一般用大写,自定义标识符用小写。

2. 保留字

保留字又称关键字,是具有特殊含义的标识符,只能作为固定的用途,不能用作标识符。VHDL的保留字如下:ABS、ACCESS、AFTER、ALIAS、ALL、AND、ARCHITECTURE、ARRAY、ASSERT、ATTRIBUTE、BEGIN、BLOCK、BODY、BUFFER、BUS、CASE、COMPONENT、CONFIGURATION、CONSTANT、DISCONNECT、DOWNTO、ELSE、ELSIF、END、ENTITY、EXIT、FILE、FOR、FUNCTION、GENERATE、GENERIC、GROUP、GUARDED、IF、IMPURE、IN、INERTIAL、INOUT、IS、LABLE、LIBRARY、LINKAGE、LITERAL、LOOP、MAP、MOD、NAND、NEW、NEXT、NOR、NOT、NULL、OF、ON、OPEN、OR、OTHERS、OUT、PACKAGE、PORT、POSTPONED、PROCEDURE、PROCESS、PURE、RANGE、RECORD、REGISTER、REJECT、REM、REPORT、RETURN、ROL、ROR、SELECT、SEVERITY、SIGNAL、SHARED、SLA、SLL、SRA、SRL、SUBTYPE、THEN、TO、TRANSPORT、TYPE、UNAFFECTED、UNITS、UNTIL、USE、VARIABLE、WAIT、WHEN、WHILE、WITH、XNOR、XOR。

关于VHDL的基本知识本节就介绍到这里,其他相关知识后面会结合具体实例或相关类型电路的描述进行介绍。这里再强调一下VHDL是硬件描述语言,描述的是硬件电路,因此与普通的计算机软件设计语言有很大的区别。普通计算机语言是CPU按照时钟节拍,一条指令执行完后才能执行下一条指令(当然也有流水执行方式和并发执行方式),因此指令执行是有先后顺序的,即顺序执行,且每条指令的执行占用特定的时间。而与VHDL描述结果对应的是硬件电路,要遵循硬件电路的特点,语句的执行没有先后顺序,是并行执行的。当然,为了实现特定的算法和功能,VHDL中也有顺序语句,但顺序语句必须放在PROCESS语句中,且PROCESS语句本身也是并行语句。另外VHDL语句的执行不像普通软件那样每条语句占用一定的时间,而是遵循硬件电路自身的特点。再有,尽管VHDL的语句很多,但一般情况下,30%的基本语句就可以完成95%以上的硬件电路的设计。所以,读者在学习VHDL时,应该多用心钻研常用语句,深入理解这些语句的硬件含义。

3.6 组合逻辑电路设计举例

前面学习了组合逻辑电路设计的一般方法和步骤,本节再给出几种数字系统中常用的组合逻辑电路的设计方法,并给出其VHDL描述。

3.6.1 半加器和全加器的设计

加法运算是计算机中最基本的一种算术运算,算术运算中的许多其他运算,如减法、乘法和除法等都可以由加法运算来实现。实现二进制加法运算的逻辑电路,称为加法器。加法器根据其功能和运算对象的不同,又可分为半加器和全加器两种,下面分别讨论其设计方法和VHDL描述。

视频讲解

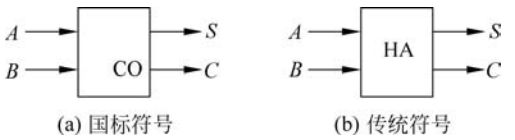
1. 半加器的设计

能完成两个一位二进制数的相加运算并求得“和”及“进位”的逻辑电路,称为半加器

(Half Adder，HA)，其逻辑符号如图 3-34 所示。其中，A 和 B 分别为两个一位二进制数的输入端；S 和 C 分别为相加后形成的“和”及“进位”输出端。半加器用来实现多位二进制加法中最低位相加运算的实现。

半加器的设计步骤如下：

(1) 根据功能要求列出真值表，如表 3-22 所示。



(a) 国标符号　　(b) 传统符号

图 3-34　半加器的逻辑符号

表 3-22　半加器真值表

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(2) 由真值表写出输出函数的“最小项之和”表达式。

$$S=\overline{A}B+A\overline{B},\quad C=AB$$

(3) 将输出函数化简为最简“与-或”式。S 和 C 已经是最简“与-或”式。

(4) 画出逻辑电路图。至此，半加器的设计基本完成。

另外，对 S 和 C 进行适当变换后，可有几种不同方案实现半加器。

方案一：假设输入端既可提供原变量又可提供反变量，可将 S 和 C 作以下变换。

$$S=\overline{A}B+A\overline{B}=\overline{\overline{A}B\cdot A\overline{B}},\quad C=\overline{\overline{AB}}$$

其对应的逻辑电路如图 3-35(a)所示。

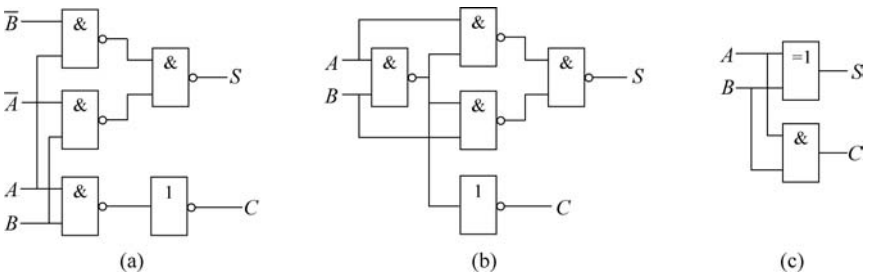


图 3-35　半加器的逻辑电路图

方案二：如果输入端只能提供原变量，而不能提供反变量，则可将 S 和 C 作以下变换。

$$S=\overline{A}B+A\overline{B}=\overline{\overline{AB}\cdot B}+A\cdot\overline{\overline{AB}}=\overline{\overline{\overline{AB}\cdot B}\cdot A\cdot\overline{\overline{AB}}},\quad C=\overline{\overline{AB}}$$

其对应的逻辑电路如图 3-35(b)所示。

方案三：由 S 的表达式可知，半加和运算实际上是进行的异或运算，所以可以用“异或”门来实现，逻辑电路如图 3-35(c)所示。

下面给出半加器的一种 VHDL 数据流描述(依据表达式 $S=\overline{AB}(A+B)$，$C=\overline{\overline{AB}}$)。

```
ENTITY half_adder IS
    PORT (a,b:IN BIT;                    -- 输入信号
```

```
        s,c:OUT BIT);                              -- 输出信号
END half_adder;
ARCHITECTURE dataflow_h_adder OF half_adder IS
    SIGNAL c0,d:BIT;                               -- 中间信号
BEGIN
    c0 <= a OR b;
    d <= a NAND b;
    c <= NOT d;
    s <= c0 AND d;
END dataflow_h_adder;
```

程序中主要用了并行赋值语句和逻辑运算符。所谓并行语句是指这些语句是同时执行的,与书写顺序无关,程序中的4条信号赋值语句可以以任意顺序书写,而不影响程序执行的结果。这与硬件电路中的信号可以经过不同的路径同时前进是一致的。

VHDL预定义了5种运算符,即算术运算符、逻辑运算符、关系运算符、符号运算符和移位运算符。表3-23列出了VHDL各种运算符的类型、符号、功能和操作数的数据类型。

表3-23 VHDL运算符

运算符的类型	符号	功　能	操作数的数据类型
算术运算符	+	加法运算	整数
	−	减法运算	整数
	*	乘法运算	整数和实数
	/	除法运算	整数和实数
	MOD	取模运算	整数
	REM	求余运算	整数
	* *	乘方运算	整数
	ABS	取绝对值	整数
	&	并置(连接)运算	一维数组
关系运算符	=	等于	任何数据类型
	/=	不等于	任何数据类型
	<	小于	枚举与整数,以及对应的一维数组
	>	大于	枚举与整数,以及对应的一维数组
	<=	小于或等于(也作信号赋值运算符)	枚举与整数,以及对应的一维数组
	>=	大于或等于	枚举与整数,以及对应的一维数组
逻辑运算符	AND	逻辑与运算	BIT、BOOLEAN、STD_LOGIC
	OR	逻辑或运算	BIT、BOOLEAN、STD_LOGIC
	NAND	逻辑与非运算	BIT、BOOLEAN、STD_LOGIC
	NOR	逻辑或非运算	BIT、BOOLEAN、STD_LOGIC
	XOR	逻辑异或运算	BIT、BOOLEAN、STD_LOGIC
	XNOR	逻辑异或非(同或)运算	BIT、BOOLEAN、STD_LOGIC
	NOT	逻辑非运算	BIT、BOOLEAN、STD_LOGIC
符号运算符	+	正号	整数
	−	负号	整数

续表

运算符的类型	符号	功　　能	操作数的数据类型
移位运算符	SLL	逻辑左移	BIT 或 BOOLEAN 型一维数组
	SRL	逻辑右移	BIT 或 BOOLEAN 型一维数组
	SLA	算术左移	BIT 或 BOOLEAN 型一维数组
	SRA	算术右移	BIT 或 BOOLEAN 型一维数组
	ROL	逻辑循环左移	BIT 或 BOOLEAN 型一维数组
	ROR	逻辑循环右移	BIT 或 BOOLEAN 型一维数组

由真值表推导出电路的输出表达式后，再用并行赋值语句建模编写 VHDL 源程序，这是半加器设计的一种方案，但不是最好的方式。用 VHDL 的行为描述方式，可以使源程序更加简洁明了。下面给出一种半加器的 VHDL 行为描述方式，该描述主要用一个“＋”运算符描述相加运算。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY half_adder_1 IS
  PORT (a,b:IN STD_LOGIC;                          --输入信号
        s,c:OUT STD_LOGIC);                        --输出信号
END half_adder_1;
ARCHITECTURE behavior_h_adder OF half_adder_1 IS
   SIGNAL temp:STD_LOGIC_VECTOR(1 DOWNTO 0);       --中间信号
BEGIN
    temp<=('0'&a)+('0'&b);
    c<=temp(1);
    s<=temp(0);
END behavior_h_adder;
```

在源程序中，用 2 位信号 temp 暂存加法运算的结果，用并接符号 & 将加数 a，b 扩展为 2 位操作数，确保赋值符号＜＝两边的数据类型和数据位数的一致性。

另外，因为＋运算在程序包 IEEE.STD_LOGIC_UNSIGNED.ALL 中定义，所以程序开头要用 USE IEEE.STD_LOGIC_UNSIGNED.ALL 打开该程序包。

2. 全加器的设计


视频讲解

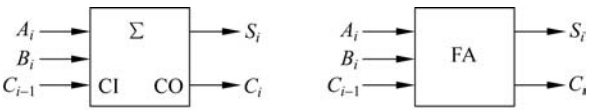
当多位二进制数相加时，高位的相加运算除了要将本位的加数和被加数相加以外，还要考虑低位是否有向该位的进位。因此，用半加器不能实现多位二进制加法运算。这时就需要一种能完成将两个 1 位二进制数相加，并考虑低位送来的进位，即相当于将 3 个 1 位二进制数相加的电路，该电路称为全加器(Full Adder，FA)，其逻辑符号如图 3-36 所示，其中 A_i 和 B_i 分别为两个 1 位二进制数的输入端，C_{i-1} 为低位来的进位输入端，S_i 和 C_i 分别为相加后的“和”及向高位的“进位”输出端。

全加器的设计过程如下：

(1) 根据功能要求列出真值表，如表 3-24 所示。

(2) 写出输出函数的“最小项之和”表达式。

$$S_i=\sum m(1,2,4,7),\quad C_i=\sum m(3,5,6,7)$$



(a) 国标符号　　(b) 传统符号

图 3-36　全加器的逻辑符号

表 3-24　全加器真值表

A_i	B_i	C_{i-1}	S_i	C_i
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

(3) 将输出函数化简成最简“与-或”式。

S_i 和 C_i 的卡诺图如图 3-37 所示。由卡诺图可知

$$S_i=\overline{A}_i\overline{B}_iC_{i-1}+\overline{A}_iB_i\overline{C}_{i-1}+A_i\overline{B}_i\overline{C}_{i-1}+A_iB_iC_{i-1},\quad C_i=A_iB_i+A_iC_{i-1}+B_iC_{i-1}$$

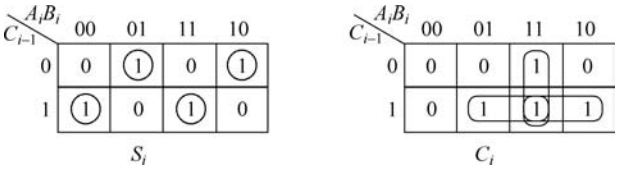


图 3-37　全加器的卡诺图

(4) 画出逻辑电路图。

根据选用门电路类型的不同,将 S_i 和 C_i 进行适当的变换,可用以下几种方案来实现全加器。

方案一:用“与非”门实现全加器,S_i 和 C_i 可变换成

$$S_i=\overline{\overline{A}_i\overline{B}_iC_{i-1}\cdot\overline{\overline{A}_iB_i\overline{C}_{i-1}}\cdot\overline{A_i\overline{B}_i\overline{C}_{i-1}}\cdot\overline{A_iB_iC_{i-1}}},\quad C_i=\overline{\overline{A_iB_i}\cdot\overline{A_iC_{i-1}}\cdot\overline{B_iC_{i-1}}}$$

其对应的逻辑电路如图 3-38(a)所示。

方案二:用半加器实现全加器,S_i 和 C_i 可变换成

$$\begin{aligned}S_i&=\overline{A}_i\overline{B}_iC_{i-1}+\overline{A}_iB_i\overline{C}_{i-1}+A_i\overline{B}_i\overline{C}_{i-1}+A_iB_iC_{i-1}\\&=(\overline{A}_i\overline{B}_i+A_iB_i)C_{i-1}+(A_i\overline{B}_i+\overline{A}_iB)\overline{C}_{i-1}\\&=\overline{(A_i\oplus B_i)}C_{i-1}+(A_i\oplus B_i)\overline{C}_{i-1}=A_i\oplus B_i\oplus C_{i-1}\\C_i&=\overline{A}_iB_iC_{i-1}+A_i\overline{B}_iC_{i-1}+A_iB_i\overline{C}_{i-1}+A_iB_iC_{i-1}\\&=A_iB_i(\overline{C}_{i-1}+C_{i-1})+(\overline{A}_iB_i+A_i\overline{B}_i)C_{i-1}=A_iB_i+(A_i\oplus B_i)C_{i-1}\end{aligned}$$

若用“异或”门构成的半加器实现全加器，则其逻辑电路如图 3-38(b)所示。

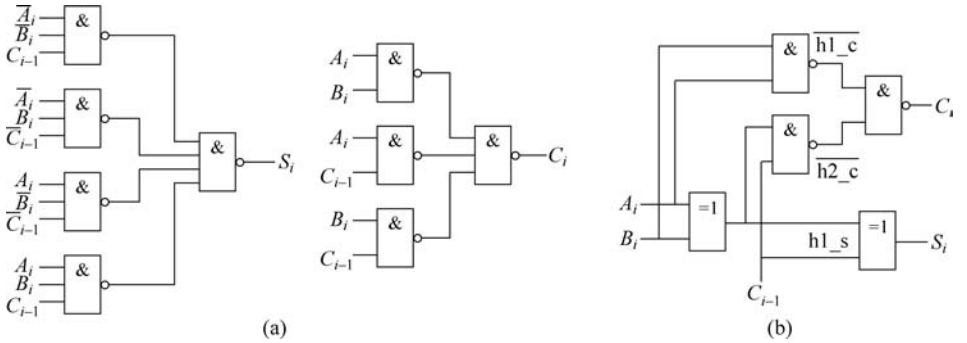


图 3-38　全加器的逻辑电路图

另外，还可用“或非”门、“与或非”门实现全加器。请读者作为练习自己完成。

下面给出依据上述表达式用两个半加器实现全加器的一种 VHDL 混合描述法。

```
ENTITY full_adder IS
    PORT (a,b,cin:IN BIT;                          -- 输入信号
          s,cout:OUT BIT);                         -- 输出信号
END full_adder;
ARCHITECTURE mix_f_adder OF full_adder IS
    COMPONENT half_adder                           -- 半加器部件声明
       PORT(a,b:IN BIT;
            s,c:OUT BIT);
    END COMPONENT;
SIGNAL h1_s,h1_c,h2_c:BIT;                         -- 内部信号
BEGIN
    h1:half_adder PORT MAP(a,b,h1_s,h1_c);         -- 半加器元件例化
    h2:half_adder PORT MAP(h1_s,cin,s,h2_c);
    cout <= h1_c OR h2_c;
END mix_f_adder;
```

混合描述就是在结构体中同时使用两种或两种以上的描述方式，它可以使描述简单灵活。在本程序的同一结构体中求和运算采用了结构描述方法，由两个半加器元件例化实现。而进位输出 cout 则采用了数据流描述方法。为方便理解，图 3-38(b)中标出了程序中的中间信号与半加器接口信号之间的对应关系。

全加器的 VHDL 描述同样也可以用类似半加器行为描述的方法用＋运算符进行行为描述，请读者自己完成。

3.6.2　BCD 码编码器和七段显示译码器的设计

日常生活中人们普遍使用十进制数据，而计算机只能识别二进制的 0 和 1。因此，如何将十进制数的二进制编码送入计算机中，经过处理后，又如何将二进制编码表示的十进制数以十进制的形式显示出来，这是相当重要的。这主要由 BCD 码编码器和 BCD 码七段显示译码器来实现，如图 3-39 所示。下面分别介绍它们的设计方法及其 VHDL 描述。

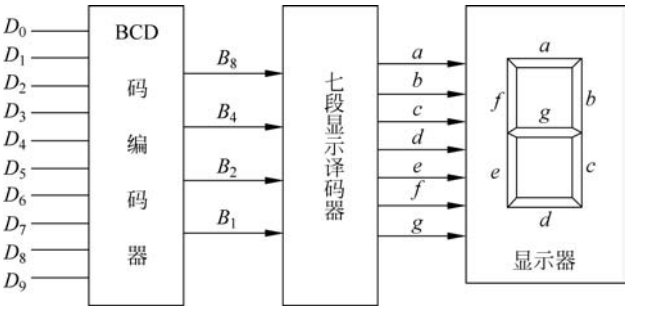


图 3-39　BCD 码编码器和七段显示译码器框图

视频讲解

1. BCD 码编码器的设计

BCD 码编码器的作用是将输入的十进制数以 BCD 码的形式输出，如图 3-39 所示。其中 D_0，D_1，…，D_9 为十进制数据输入端，分别代表 0，1，…，9。除 D_0 以外，当某个输入端为 1 时，表示以其对应的十进制数作为输入。显然，除 D_0 以外的 9 个输入端应是互斥的，任何时候只能有一个有效。当 10 个输入端都为 0 时，表示输入的是十进制数 0(当然也可以约定 D_0 为 1，其余为 0 时表示输入的是 0，之所以这样做是因为最终设计出的电路可以将 D_0 去掉，省去一个输入端)。B_8、B_4、B_2、B_1 为输入数字对应的 BCD 码输出端。

BCD 码编码器的设计过程如下。

(1) 列出真值表。根据以上对 BCD 码编码器功能的分析，可列出如表 3-25 所示的部分真值表(由于问题的特殊性不需要列出完整的真值表)。

表 3-25　BCD 码编码器真值表

数字	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	B_8	B_4	B_2	B_1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0	0	0	1
2	0	0	0	0	0	0	0	1	0	0	0	0	1	0
3	0	0	0	0	0	0	1	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0	0	0	0	1	0	0
5	0	0	0	0	1	0	0	0	0	0	0	1	0	1
6	0	0	0	1	0	0	0	0	0	0	0	1	1	0
7	0	0	1	0	0	0	0	0	0	0	0	1	1	1
8	0	1	0	0	0	0	0	0	0	0	1	0	0	0
9	1	0	0	0	0	0	0	0	0	0	1	0	0	1

(2) 写出函数表达式。根据真值表可直接写出输出函数表达式。

$$B_1 = D_1 + D_3 + D_5 + D_7 + D_9, \quad B_2 = D_2 + D_3 + D_6 + D_7$$

$$B_4 = D_4 + D_5 + D_6 + D_7, \quad B_8 = D_8 + D_9$$

(3) 表达式已为最简，考虑到多输出函数尽量使用公共项，可作以下变换：

$$B_1 = \overline{\overline{D_1 + D_9} \cdot \overline{D_3 + D_7} \cdot \overline{D_5 + D_7}}, \quad B_2 = \overline{\overline{D_3 + D_7} \cdot \overline{D_2 + D_6}}$$

$$B_4 = \overline{\overline{D_5 + D_7} \cdot \overline{D_4 + D_6}}, \quad B_8 = \overline{\overline{D_8 + D_9}}$$

(4) 画出逻辑电路图，如图 3-40 所示。

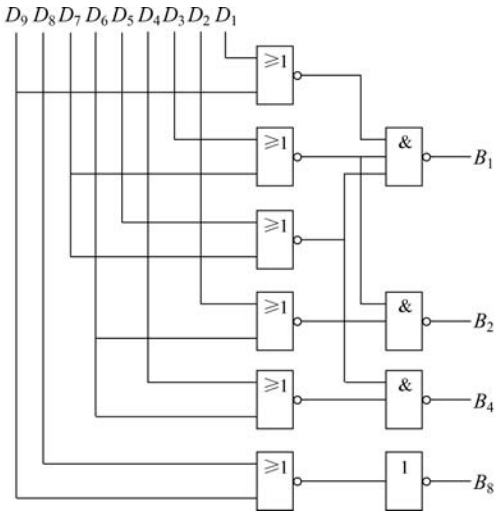


图 3-40　BCD 编码器逻辑电路图

下面给出 BCD 码编码器的一种 VHDL 行为描述。

```
LIBRARY IEEE;                                          -- 包含库
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY coder_bcd IS
  PORT(coder_in:IN STD_LOGIC_VECTOR(8 DOWNTO 0);       -- 编码输入向量
       coder_out:OUT STD_LOGIC_VECTOR(3 DOWNTO 0));    -- BCD 码输出向量
END coder_bcd;
ARCHITECTURE behavior_coder_bcd OF coder_bcd IS
BEGIN
  WITH coder_in SELECT
       coder_out <= "0000" WHEN "000000000",
                    "0001" WHEN "000000001",
                    "0010" WHEN "000000010",
                    "0011" WHEN "000000100",
                    "0100" WHEN "000001000",
                    "0101" WHEN "000010000",
                    "0110" WHEN "000100000",
                    "0111" WHEN "001000000",
                    "1000" WHEN "010000000",
                    "1001" WHEN "100000000",
                    "XXXX" WHEN OTHERS;                -- 输出不确定
END behavior_coder_bcd;
```

程序中使用的 WITH 语句称为选择信号赋值语句，该语句与 CASE 语句类似，以选择表达式的不同取值为选择条件，将多个表达式的值赋给赋值目标。注意：CASE 语句属于顺序描述语句，应放在 PROCESS 语句中，而 WITH 语句属于并行描述语句。WITH 语句

的格式为

```
WITH 选择表达式 SELECT
    赋值目标<=表达式1 WHEN 选择值1,
              表达式2 WHEN 选择值2,
                    ⋮
              表达式n WHEN 选择值n/OTHERS;
```

WITH语句对子句中的“选择值”进行选择,当子句中的“选择值”与“选择表达式”的值相同时,则将子句中的“表达式”的值赋给赋值目标。

使用WITH语句应注意:①不允许有选择值重叠现象;②不允许有选择值涵盖不全的情况,如果选择值不需要列全,则“选择值n”用OTHERS代替;③每个子句以“,”结束,最后一个子句以“;”结束。

与WITH语句同属并行信号赋值语句的还有条件信号赋值语句,它根据不同的赋值条件,选择表达式中的值赋给赋值目标。条件赋值语句的格式为

```
赋值目标<=表达式1 WHEN 赋值条件1 ELSE
          表达式2 WHEN 赋值条件2 ELSE
                    ⋮
          表达式n WHEN 赋值条件n ELSE
          表达式n+1;
```

条件赋值语句的功能与放在PROCESS中的IF语句(为顺序描述语句)相同,每个赋值条件是按书写先后顺序逐项测定的,一旦发现赋值条件为TRUE,立即将对应表达式的值赋给目标信号。而WITH语句中对各选择值的判断是同时进行的,与书写顺序无关。下面给出用条件赋值语句实现的二输入与非门的VHDL描述。

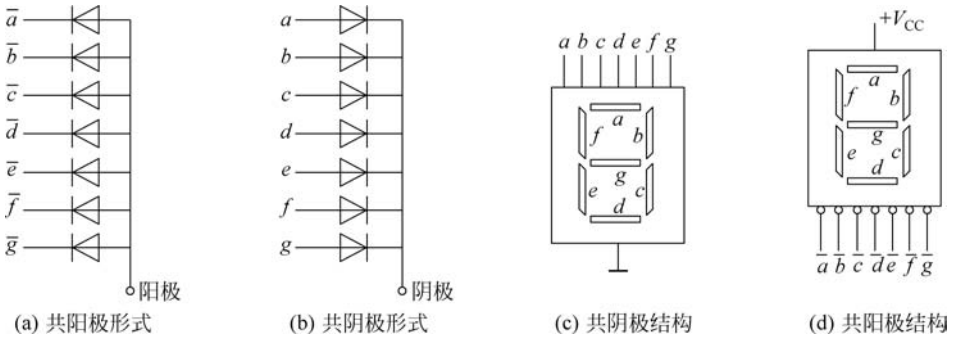
```
ENTITY nand_2 IS
   PORT(a,b:IN BIT;
        f:OUT BIT);
END nand_2;
ARCHITECTURE behavior OF nand_2 IS
   SIGNAL s:BIT_VECTOR(1 DOWNTO 0);
BEGIN
   s<=a&b;
   f<= '1' WHEN s<="00" ELSE
       '1' WHEN s<="01" ELSE
       '1' WHEN s<="10" ELSE
       '0';
END behavior;
```

视频讲解

2. BCD码七段显示译码器的设计

BCD码七段显示译码器的作用是将BCD码表示的十进制数转换成七段LED显示器的7个驱动输入端,如图3-39所示。其中B_8、B_4、B_2、B_1为BCD码输入端,输出$a\sim g$为七段LED显示器的7个驱动输入端。

七段 LED 显示器由 7 个条形发光二极管(LED)组成,不同段 LED 的亮、灭组合,即可显示不同的数字。LED 显示器有共阳极和共阴极连接两种形式,如图 3-41 所示。共阳极形式是将 7 个 LED 的阳极连在一起并接高电平,当需要某段 LED 点亮时,就让其阴极接低电平即可;否则接高电平。共阴极形式和共阳极形式相反,将 7 个 LED 的阴极连在一起并接低电平,当需要某段 LED 点亮时,就让其阳极接高电平即可;否则接低电平。



(a) 共阳极形式　(b) 共阴极形式　(c) 共阴极结构　(d) 共阳极结构

图 3-41　七段 LED 显示器

BCD 码七段显示译码器的设计过程如下:

(1) 列出真值表。假设采用共阴极连接形式,根据以上功能分析,可列出如表 3-26 所示的真值表。

表 3-26　BCD 码七段显示译码器真值表

数字	B_8	B_4	B_2	B_1	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	0	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
10	1	0	1	0	d	d	d	d	d	d	d
11	1	0	1	1	d	d	d	d	d	d	d
12	1	1	0	0	d	d	d	d	d	d	d
13	1	1	0	1	d	d	d	d	d	d	d
14	1	1	1	0	d	d	d	d	d	d	d
15	1	1	1	1	d	d	d	d	d	d	d

(2) 列出函数表达式并化简。因为输出函数共有7个,如果按多输出函数进行化简将十分复杂,因此,这里按7个单输出函数进行单独化简。这里,可由真值表直接做出卡诺图并进行化简。此处卡诺图从略,直接写出输出函数表达式,请读者自己画出卡诺图进行验证。

$$a=B_8+\overline{B}_4\overline{B}_1+B_2B_1+B_4B_1,\quad b=\overline{B}_4+B_2B_1+\overline{B}_2\overline{B}_1,\quad c=B_4+\overline{B}_2+B_1$$

$$d=\overline{B}_4\overline{B}_1+\overline{B}_4B_2+B_2\overline{B}_1+B_4\overline{B}_2B_1,\quad e=\overline{B}_4\overline{B}_1+B_2\overline{B}_1$$

$$f=B_8+\overline{B}_2\overline{B}_1+B_4\overline{B}_2+B_4\overline{B}_1,\quad g=B_8+B_2\overline{B}_1+\overline{B}_4B_2+B_4\overline{B}_2$$

(3) 画出逻辑电路图。如用“与非”门实现上述函数,并考虑公共项,其逻辑电路如图3-42所示。

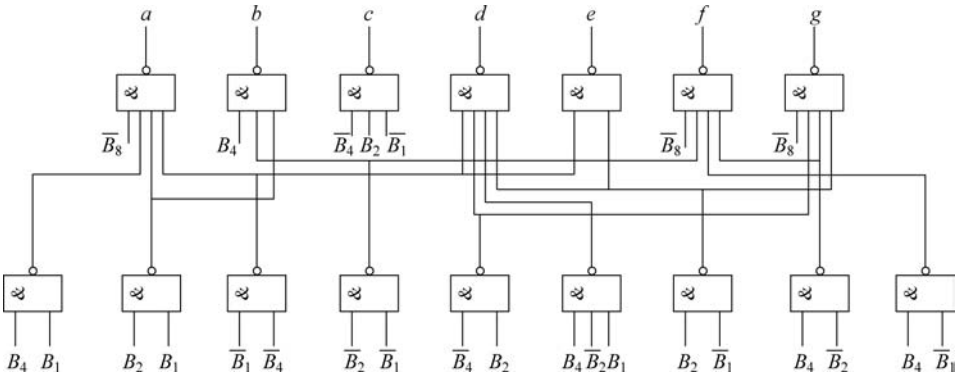


图3-42 BCD七段译码器逻辑电路图

下面给出一种BCD码七段显示译码器的VHDL行为描述。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY decoder_bcd_display IS
    PORT(decoder_in:IN STD_LOGIC_VECTOR(3 DOWNTO 0);          -- BCD码输入向量
         decoder_out:OUT STD_LOGIC_VECTOR(6 DOWNTO 0));       -- 七段码输出向量
END decoder_bcd_display;
ARCHITECTURE behavior_decoder_bcd_display OF decoder_bcd_display IS
BEGIN
    WITH decoder_in SELECT
        decoder_out <= "1111110" WHEN "0000",
                       "0110000" WHEN "0001",
                       "1101101" WHEN "0010",
                       "1111001" WHEN "0011",
                       "0110011" WHEN "0100",
                       "1011011" WHEN "0101",
                       "0011111" WHEN "0110",
                       "1110000" WHEN "0111",
                       "1111111" WHEN "1000",
                       "1110011" WHEN "1001",
```

```
                    "XXXXXXX" WHEN OTHERS;
END behavior_decoder_bcd_display;
```


视频讲解

3.6.3 代码转换器的设计

根据需要，在计算机及其他各种数字设备中，普遍采用多种不同类型的代码，这些代码在必要时需要相互转换。下面举例说明实现这些代码转换功能的代码转换器的设计方法，并给出其 VHDL 描述。

1. 8421BCD 码到余 3 码的代码转换器的设计

设输入的 8421BCD 码用 B_8、B_4、B_2、B_1 表示，输出的余 3 码用 E_4、E_3、E_2、E_1 表示。设计过程如下。

(1) 列出如表 3-27 所示的真值表，列表时不要忘记无关项 d。

表 3-27 8421BCD 码到余 3 码的代码转换器的真值表

数字	BCD 码				余 3 码			
	B_8	B_4	B_2	B_1	E_4	E_3	E_2	E_1
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0
10	1	0	1	0	d	d	d	d
11	1	0	1	1	d	d	d	d
12	1	1	0	0	d	d	d	d
13	1	1	0	1	d	d	d	d
14	1	1	1	0	d	d	d	d
15	1	1	1	1	d	d	d	d

(2) 写出函数表达式并化简。

由真值表画出卡诺图，如图 3-43 所示。由卡诺图化简可得

$$E_4 = B_8 + B_4B_2 + B_4B_1, \quad E_3 = \overline{B}_4B_2 + \overline{B}_4B_1 + B_4\overline{B}_2\overline{B}_1$$

$$E_2 = B_2B_1 + \overline{B}_2\overline{B}_1, \quad E_1 = \overline{B}_1$$

(3) 画出逻辑电路图。

采用“与非”门实现的逻辑电路图，如图 3-44 所示。

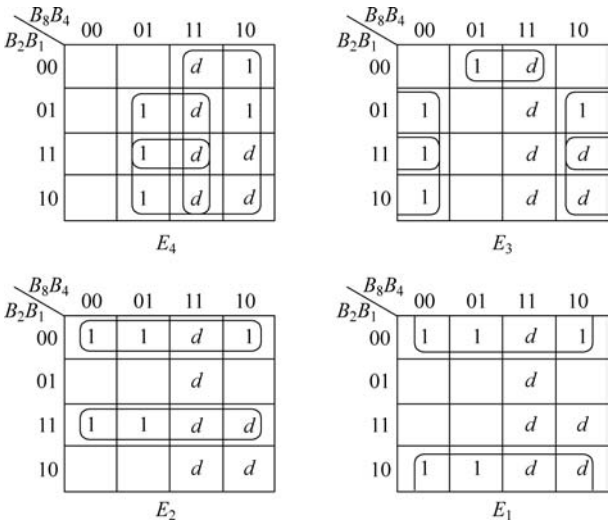


图 3-43　8421BCD 码到余 3 码的代码转换器的卡诺图

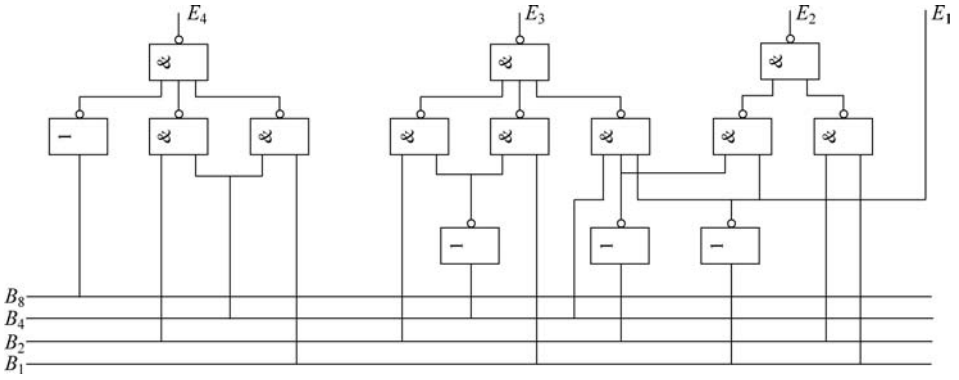


图 3-44　8421BCD 码到余 3 码的代码转换器的电路图

下面给出一种 8421BCD 码到余 3 码的代码转换器的 VHDL 行为描述。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY converter_bcd_remainder3 IS
    PORT(converter_in:IN STD_LOGIC_VECTOR(3 DOWNTO 0);          -- BCD 码输入向量
         converter_out:OUT STD_LOGIC_VECTOR(3 DOWNTO 0));      -- 余 3 码输出向量
END converter_bcd_remainder3;
ARCHITECTURE behavior_ converter_bcd_remainder3 OF converter_bcd_remainder3 IS
BEGIN
    WITH converter_in SELECT
         converter_out <= "0011" WHEN "0000",
                          "0100" WHEN "0001",
                          "0101" WHEN "0010",
                          "0110" WHEN "0011",
                          "0111" WHEN "0100",
                          "1000" WHEN "0101",
```

```
                    "1001" WHEN "0110",
                    "1010" WHEN "0111",
                    "1011" WHEN "1000",
                    "1100" WHEN "1001",
                    "XXXX" WHEN OTHERS;
END behavior_converter_bcd_remainder3;
```

反之，若输入为余 3 码，也可以转换成 BCD 码，请读者自行练习。应当注意 0000～0010 和 1101～1111 这 6 种码为余 3 码的非法码，应作为无关项处理。

2. 4 位二进制码到 Gray 码的代码转换器的设计

设输入的 4 位二进制码用 B_4、B_3、B_2、B_1 表示；输出的 Gray 码用 G_4、G_3、G_2、G_1 表示。由第 1 章中对 Gray 码的介绍可知，Gray 码与二进制码之间的关系为

$$G_4 = B_4, \quad G_3 = B_4 \oplus B_3$$

$$G_2 = B_3 \oplus B_2, \quad G_1 = B_2 \oplus B_1$$

据此，可选用“异或”门设计该电路，逻辑电路图如图 3-45 所示。

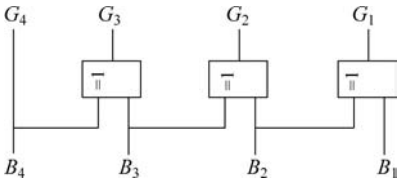


图 3-45　4 位二进制码到 Gray 码的代码转换器的逻辑电路图

下面给出一种 4 位二进制码到 Gray 码的代码转换器的 VHDL 数据流描述。

```
ENTITY binary_gray IS
    PORT(b1,b2,b3,b4:IN BIT;
         g1,g2,g3,g4:OUT BIT);
END binary_gray;
ARCHITECTURE rtl_ binary_gray OF binary_bray IS
BEGIN
   g4 <= b4;
   g3 <= b4 XOR b3;
   g2 <= b3 XOR b2;
   g1 <= b2 XOR b1;
END rtl_binary_gray;
```

3.7　组合逻辑电路中的竞争与险象

对于组合逻辑电路，前面只讨论了输入和输出在稳定状态时的关系，而从未考虑信号在传输中的时延问题。实际上，信号经过任何门电路和导线都存在一定的时间延迟，这就使得电路的输入达到稳定状态时，输出并不一定能立即达到稳定状态。例如，假定一个二输入“与非”门的延迟时间为 t_{pd}，当输入 B 为 1 时，若让 A 从 0 变到 1 再变回到 0，则输出将由 1 变到 0 再变到 1，如图 3-46 所示。可见，输入信号经过延迟时间 t_{pd} 后才传输到输出端，即输出对输入的响应滞后了 t_{pd} 的时间。

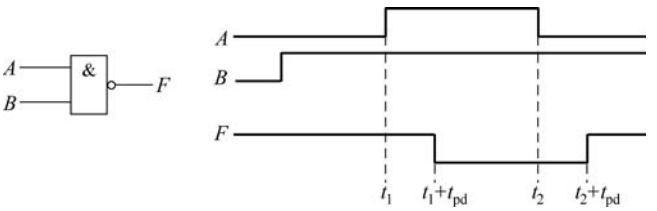


图 3-46 "与非"门延迟时间的影响

视频讲解

一般来说,延迟时间对数字系统是一个不利因素。如可使系统操作速度变慢、导致电路中信号的波形参数变坏,更为严重的是可能产生竞争冒险现象。本节将专门讨论后一个问题。

3.7.1 竞争与险象的产生

逻辑电路中的信号经过同一电路的不同路径所需的时间一般是不同的,这与信号经过的门的级数、具体逻辑门的时延大小、导线的长短有关。因此,输入信号经过不同的路径到达输出端的时间也就有先有后,这种现象称为竞争现象,简称竞争(race)。在逻辑电路中,可以把竞争现象广义地理解为多个信号到达某一点时由时差所引起的现象。

竞争现象在逻辑电路中是普遍存在的。如果电路中存在竞争现象,当输入信号变化时就有可能引起输出信号出现非预期的错误输出,这种现象称为险象(hazard)。但并不是所有的竞争都会产生错误输出,常把不会产生错误输出的竞争称为非临界竞争(noncritical race),而会导致错误输出的竞争称为临界竞争(critical race)。

组合逻辑电路中的险象是一种瞬态现象,它表现为在输出端产生不应有的尖峰脉冲,短暂地破坏正常逻辑关系,一旦时延结束,即可恢复正常的逻辑关系。下面举例说明这一现象。

图 3-47(a)是一个由"与非"门构成的组合逻辑电路,该电路有三个输入,一个输出,输出函数表达式为

$$F = AB + \overline{A}C$$

假设输入变量 B 和 C 均为 1,则上式可变为

$$F = A + \overline{A} = 1$$

即,无论此时 A 怎样变化,F 的值都应恒为 1。然而,这只是在理想状态下得出的结论。下面讨论当 $B=C=1$ 并且考虑延迟时间时,A 的变化会使电路产生怎样的输出响应。假设每个门的延迟时间均为 t_{pd},则图 3-47(b)可以说明输出对输入的响应关系。

从图 3-47(b)可以看出,当 A 由低变高后,经过一个 t_{pd} 时间,"非"门 G_1 的输出 d 由高变低,同时"与非"门 G_2 的输出 e 也由高变低,但要再过一个 t_{pd} 时间,"与非"门 G_3 的输出 g 才能由低变高。最后到达"与非"门 G_4 输入端的是由同一个信号 A 经不同路径传输而得到的两个信号 e 和 g,而 e 和 g 的变化方向相反,并有一个 t_{pd} 的时差。显然,这里(图中标 1 处)存在一次竞争现象,但因门 G_4 是一个"与非"门,e 和 g 的竞争结果使门 G_4 的输出保持高电平,没有出现尖峰脉冲,即无险象产生,所以这是一次非临界竞争。但当 A 由高变低时,情况就不同了。e 和 g 同样在门 G_4 上发生竞争,且 e 和 g 在一个 t_{pd} 的时间内同为高电平,所以输出 F 也必须会出现一个负跳变的尖峰脉冲(图中标 2 处),即这次竞争产生了险

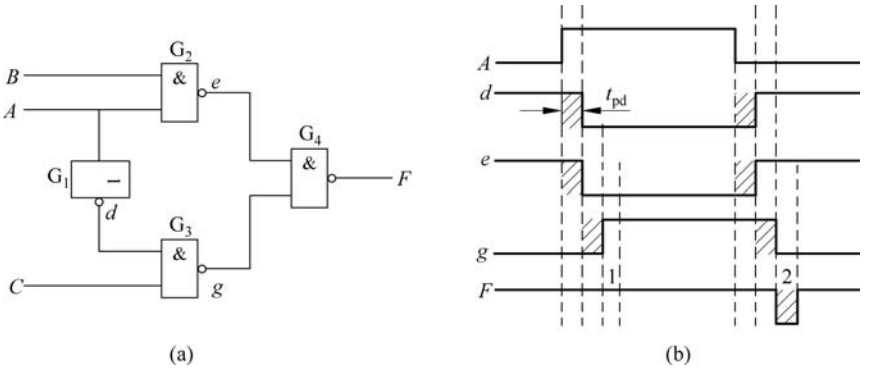


图 3-47　具有险象的逻辑电路及时间图

象，是一次临界竞争。

3.7.2　险象的分类

根据输入变化前后，输出是否应该相同可将组合电路中的险象分为静态险象(static hazard)和动态险象(dynamic hazard)两类。

如果在输入变化而输出不应发生变化的情况下，输出端却产生了短暂的错误输出，即产生了险象，则这种险象称为静态险象。如果在输入变化而输出应该变化的情况下，输出在变化过程中产生了短暂的错误输出，则这种险象称为动态险象。显然，如图 3-47 所示的险象属于静态险象。

险象还可按照错误输出的尖峰脉冲的极性分为 0 型险象和 1 型险象。若错误输出的尖峰脉冲为负脉冲，则称为 0 型险象；反之，若错误输出的尖峰脉冲为正脉冲，则称为 1 型险象。显然，图 3-47 中的险象属于 0 型险象。请读者自己分析，若将图 3-47 中的门 G_2、G_3、G_4 改为“或非”门，则会在何处产生哪种险象。显然，无论静态险象还是动态险象，都可分为 0 型险象和 1 型险象。图 3-48 示出了组合电路中 4 种险象类型的波形，图中以虚线为界表示输入变化前后的输出波形。

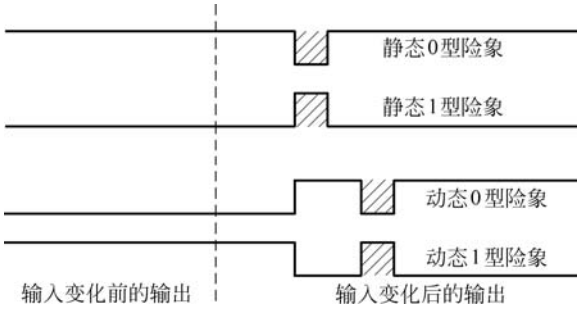


图 3-48　组合电路中的 4 种险象类型

值得指出的是，组合电路中的动态险象一般都是由静态险象引起的。因此，如果消除了电路中的静态险象则也就消除了动态险象。

3.7.3 险象的判断

判断组合逻辑电路中是否有可能产生险象的方法有两种,即代数法和卡诺图法。

由前面对竞争和险象的分析可知,当某变量 X 同时以原变量和反变量两种形式出现在函数中,并且在一定的条件下,可将函数表达式化简成 $X+\overline{X}$ 的形式时,则该函数表达式对应的电路在 X 发生变化时,就可能由于竞争的存在而产生险象(例如图 3-47 中的例子)。同样,若在一定的条件下,函数表达式可以化简成 $X\cdot\overline{X}$ 的形式,则相应的电路在 X 发生变化时也有可能因为竞争的存在而产生险象。

视频讲解

因此,组合逻辑电路中存在险象可能性的必要条件是:某变量 X 同时以原变量 X 和反变量 $\overline{X}$ 两种形式出现在函数中,并且在一定的条件下可以将函数表达式化简成 $X+\overline{X}$ 或 $X\cdot\overline{X}$ 的形式。因为 $X+\overline{X}=1$,$X\cdot\overline{X}=0$,所以若函数表达式可以化简成 $X+\overline{X}$ 的形式,则可能存在的险象为 0 型险象;若函数表达式可以化简成 $X\cdot\overline{X}$ 的形式,则可能存在的险象为 1 型险象。

1. 代数法

代数法是根据函数表达式的结构来判断是否有产生险象所必需的条件。具体方法是:首先,检查函数表达式中是否存在具备竞争条件的变量,即是否有某个变量 X 同时以原变量 X 和反变量 $\overline{X}$ 的形式在函数表达式中出现。若有,则消去函数表达式中的其他变量,方法是将这些变量的各种取值组合依次代入函数表达式中,从而使表达式中仅含被研究的变量 X。最后,再看函数表达式是否能化成 $X+\overline{X}$ 或 $X\cdot\overline{X}$ 的形式,若能,则对应的逻辑电路存在产生险象的可能性。下面举例说明。

例 3-32 判断函数表达式 $F=\overline{A}\,\overline{C}+\overline{A}B+AC$ 对应的逻辑电路是否可能产生险象。

解:由函数表达式可知,变量 A 和 C 具备竞争的条件,所以应对这两个变量进行分析。先考察变量 A,为此将 B 和 C 的各种取值组合分别代入函数表达式中,可得

$$BC=00\text{ 时},F=\overline{A};\ BC=01\text{ 时},F=A$$

$$BC=10\text{ 时},F=\overline{A};\ BC=11\text{ 时},F=A+\overline{A}$$

可见,$BC=11$ 时,变量 A 的变化可能使电路产生 0 型险象。

用同样的方法考查变量 C,可知变量 C 的变化不会使电路产生险象。

例 3-33 判断函数表达式 $F=(A+B)(\overline{A}+C)(\overline{B}+C)$ 对应的逻辑电路中是否存在产生险象的可能性。

解:由函数表达式可知,变量 A 和 B 具备竞争的条件。首先考察变量 A,为此将变量 B 和 C 的各种取值组合分别代入函数表达式中,可得

$$BC=00\text{ 时},F=A\cdot\overline{A};\ BC=01\text{ 时},F=A$$

$$BC=10\text{ 时},F=0;\ BC=11\text{ 时},F=1$$

可见,当 $BC=00$ 时,变量 A 的变化可能使电路产生 1 型险象。

用同样的方法考察 B,可知,当 $AC=00$ 时,变量 B 的变化也可能使电路产生 1 型险象。

2. 卡诺图法

判断险象的另一种方法是卡诺图法。当电路对应的逻辑函数已是“与-或”式时,卡诺图法比代数法更直观方便。具体方法是:首先画出函数的卡诺图,并画出和函数表达式中各“与”项对应的卡诺圈。然后观察卡诺圈,若发现某两个卡诺圈存在“相切”关系,即两个卡诺

圈之间存在不被同一个卡诺圈包含的相邻最小项，则该电路可能产生险象。下面举例说明。

例 3-34 判断函数 $F=\overline{A}D+\overline{A}C+AB\overline{C}$ 对应的逻辑电路是否可能产生险象。

解：首先，做出函数 F 的卡诺图，并画出各“与”项对应的卡诺圈，如图 3-49 所示。

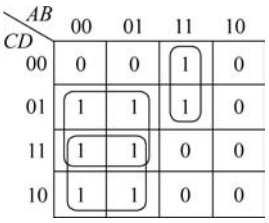


图 3-49 例 3-34 的卡诺图

观察卡诺圈可以发现，包含最小项 m_1、m_3、m_5、m_7 的卡诺圈和包含最小项 m_{12}、m_{13} 的卡诺圈之间存在相邻最小项 m_5 和 m_{13}，且 m_5 和 m_{13} 不被同一个卡诺圈所包含，所以这两个卡诺圈“相切”。这就说明相应的电路可能产生险象。

进一步用代数法验证，可以发现当 $BCD=101$ 时，函数表达式可化成 $F=A+\overline{A}$ 的形式，可见变量 A 的变化可能使电路产生 0 型险象。

3.7.4 险象的消除


视频讲解

为使所设计的电路能可靠地工作，设计者应设法消除或避免电路中可能出现的险象。下面介绍几种常用的方法。

1. 增加冗余项法

增加冗余项的方法，是通过在函数表达式中“加”多余的“与”项或“乘”多余的“或”项，使原函数不再可能在某种条件下化成 $X+\overline{X}$ 或 $X\cdot\overline{X}$ 的形式，从而将可能产生的险象消除。冗余项的具体选择方法可采用代数法或卡诺图法，下面举例说明。

例 3-35 用增加冗余项的方法消除如图 3-47(a)所示的电路中可能产生的险象。

解：如图 3-47(a)所示的电路对应的函数表达式为

$$F=AB+\overline{A}C$$

由前面的分析可知，当 $BC=11$ 时，输入变量 A 的变化使电路的输出产生 0 型险象，即在输出应该为 1 的情况下产生了一个瞬间的 0 信号。解决办法是在保证 $BC=11$ 时，使输出保持为 1。显然，若在表达式中包含“与”项 BC，即可达到目的。又由逻辑代数的基本公式(包含律)可知

$$AB+\overline{A}C+BC=AB+\overline{A}C$$

所以，BC 是上述函数的一个冗余项，将 BC 加入函数表达式中并不影响原函数的功能。增加了冗余项 BC 后的逻辑电路如图 3-50 所示，该电路不再有产生险象的可能性。

冗余项的选择也可以通过在函数的卡诺图上增加多余的卡诺圈来实现。具体方法是：若卡诺图上某两个卡诺圈“相切”，则用一个多余的卡诺圈将它们之间的相邻最小项圈起来，与该卡诺圈对应的“与”项就是要加入函数表达式中的冗余项。

例 3-36 某组合电路对应的函数表达式为 $F=\overline{A}C+B\overline{C}D+A\overline{B}\overline{C}$，试用增加冗余项的方法消除该电路中可能产生的险象。

解：首先，做出函数的卡诺图，如图 3-51 所示。该卡诺图中，卡诺圈①和②“相切”，其相邻的最小项为 m_7 和 m_5；卡诺圈②和③“相切”，其相邻的最小项为 m_9 和 m_{13}。可见，该电路可能由于竞争的存在而产生险象。为了消除险象，可在卡诺图上增加两个多余的卡诺圈，分别把最小项 m_5、m_7 和 m_9、m_{13} 圈起来，如图中虚线所示。由此得到函数表达式为

$$F=\overline{A}C+B\overline{C}D+A\overline{B}\overline{C}+\overline{A}BD+A\overline{C}D$$

式中,$\bar{A}BD$ 和 $A\bar{C}D$ 为冗余项。读者可用代数法验证,该函数表达式对应的逻辑电路不再存在险象。

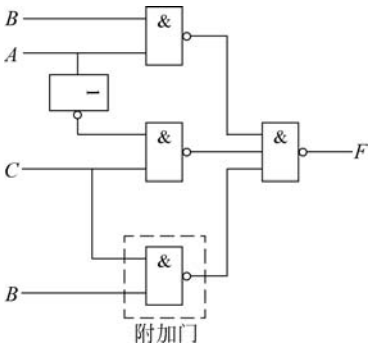


图 3-50　图 3-47(a)增加冗余项后的逻辑电路图

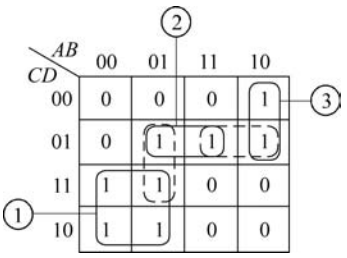


图 3-51　例 3-36 的卡诺图

应该指出,用增加冗余项消除险象的方法是以增加器件为代价的。增加冗余项后,函数表达式不再是最简式,相应的逻辑电路也就不再是最简电路。前面讨论逻辑电路的设计时,总是尽可能将函数化成最简,现在为了消除险象,又让函数表达式回到了非最简形式,然而两者都是必要的。前者是为了寻求最经济的方案,后者是为了使设计出来的电路能可靠地工作。经济和可靠都是衡量设计方案好坏的重要标志。因此,在设计逻辑电路时,通常在先不考虑险象的情况下求出最简电路,然后再判断是否存在险象并设法消除。

2. 增加惯性延时环节法

在实际电路中用来消除险象的另一种方法是在组合电路的输出端串接一个惯性延时环节。通常采用 RC 电路作为惯性延时环节,如图 3-52(a)所示。由电路知识可知,图中的 RC 电路实际上是一个低通滤波器。由于组合电路的输出信号的频率较低,而由于竞争引起的险象是一些频率较高的尖峰脉冲,因此,险象在通过 RC 电路后能基本被滤掉,保留下来的仅是一些幅度较小的毛刺,它们不再对电路的可靠性产生影响。图 3-52(b)示出了这种方法的效果。可以看出,输出的规律和力学系统中物体惯性运动的规律相似,所以称为"惯性"环节。

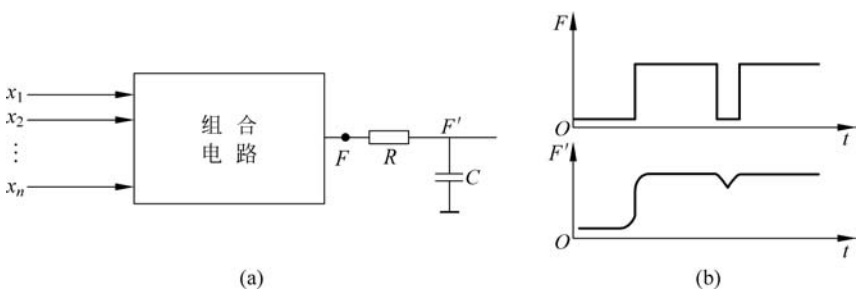


图 3-52　惯性延时环节

但要注意,采用这种方法必须选择适当的惯性环节的时间常数 τ($\tau=RC$),一般要求 τ 大于尖峰脉冲的宽度,以便能将尖峰脉冲"削平",但也不能太大,否则会使电路的正确输出信号产生不允许的畸变。

3. 选通法

增加冗余项和增加惯性延时环节均可以消除组合逻辑电路中的险象，但这两种方法的缺点是要增加额外的器件。对于组合逻辑电路中的险象除了可用以上两种方法消除以外，还可以采取另一种完全不同的方法，即避开险象而不是消除险象的方法。选通法就是基于这样一种思想的方法，它不需要增加任何器件，仅仅是利用选通脉冲的作用，从时间上加以控制，使险象脉冲无法输出。

由于组合电路中的险象总是发生在输入信号发生变化的过程中，而且险象总是以尖峰脉冲的形式输出的，因此，只要对输出波形从时间上加以选择和控制，利用选通脉冲选择输出波形的稳定部分，而有意避开可能出现的尖峰脉冲，便可获得正确的输出。

例如，如图 3-53 所示的电路的输出函数表示式为

$$F=\overline{\overline{A\cdot 1}\cdot\overline{\overline{A}\cdot 1}}=A+\overline{A}$$

可见，当 A 发生变化时，可能产生 0 型险象。

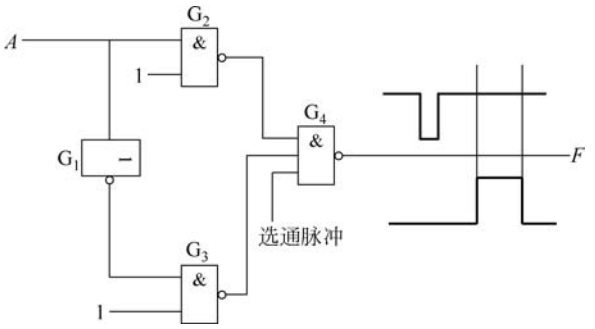


图 3-53　用选通法避开险象的原理图

为了避开险象，可采用选通脉冲对该电路的输出门加以控制。在选通脉冲到来之前，该输入线上为低电平，门 G_4 关闭，电路输出被封锁，使险象脉冲无法输出。当选通脉冲到来后，相应的输入线上变为高电平，门 G_4 开启，使电路送出稳定输出信号。

这种在时间上让信号有选择地通过的方法称为选通法。

3.8　小　　结

逻辑代数已成为研究数字系统逻辑设计的基础理论。无论何种形式的数字系统，都是由一些基本的逻辑电路所组成的。为了解决数字系统分析和设计中的各种具体问题，必须掌握逻辑代数这一重要数学工具。

同一个逻辑函数可以有多种表示形式。函数表达式越简单，则逻辑电路越简单。逻辑函数的化简有代数化简法、卡诺图化简法和列表化简法(Q-M 法)。

组合逻辑电路是由各种门电路组合而成的。它的特点是输出只与当时的输入状态有关，而与电路过去的输入状态无关。

对组合逻辑电路的研究包括分析和设计两个方面。分析是根据已知电路，找出其实现的功能；设计是根据已知的逻辑功能，求解出实现该功能的逻辑电路。分析和设计是一个

互为相反的过程。

组合逻辑电路的设计除按一般步骤进行外,还有几个要考虑的特殊问题:设计时采用的门电路的类型、多输出函数的电路设计、包含无关项的电路设计和考虑级数的电路设计。

逻辑设计可以用硬件描述语言来描述。VHDL是IEEE标准的硬件描述语言。一个完整的VHDL程序通常包含实体和结构体两部分,实体部分用于描述电路的对外接口信号,结构体描述电路的具体功能和结构。VHDL的结构体有4种描述方式,即数据流描述、行为描述、结构描述和混合描述方式。

竞争与险象是组合逻辑电路设计在最后阶段要考虑的一个重要问题。由于竞争的存在,可能会产生险象,险象会使电路工作的可靠性下降。必须对电路是否存在险象进行分析并设法消除险象,以使所设计的电路能可靠地工作。判断是否有险象存在可用代数法和卡诺图法。险象的消除可采用增加冗余项、增加惯性延时环节或采用选通信号等方法。

3.9 习题与思考题

1. 回答下列问题。

(1) 如果已知 $X+Y=X+Z$,那么 $Y=Z$ 正确吗?为什么?

(2) 如果已知 $XY=XZ$,那么 $Y=Z$ 正确吗?为什么?

(3) 如果已知 $X+Y=X+Z$,那么 $XY=XZ$,且 $Y=Z$ 正确吗?为什么?

(4) 如果已知 $X+Y=X\cdot Y$,那么 $X=Y$ 正确吗?为什么?

2. 用逻辑代数的公式、定理和规则将下列逻辑函数化简为最简"与或"表达式。

(1) $F=AB+\overline{A}\overline{B}C+BC$

(2) $F=A\overline{B}+B+BCD$

(3) $F=(A+B+C)(\overline{A}+B)(A+B+\overline{C})$

(4) $F=BC+D+\overline{D}(\overline{B}+\overline{C})(AC+B)$

3. 将下列函数转换为由"标准积之和"及"标准和之积"形式表示的函数。

(1) $F(A,B,C)=A\overline{B}C+\overline{A}B+AC+AB\overline{C}$

(2) $F(A,B,C)=\overline{A}+A\overline{C}+BC+A\overline{B}C$

(3) $F(A,B,C)=B(A+\overline{B}+\overline{C})(\overline{A}+\overline{C})C$

(4) $F(A,B,C)=ABC+\overline{\overline{A}\overline{B}\,\overline{C}}$

(5) $F(A,B,C)=(\overline{A}B+C)[(\overline{A}\overline{B}+B)C+A]$

4. 用卡诺图化简法求出下列逻辑函数的最简"与或"表达式和最简"或与"表达式。

(1) $F(A,B,C,D)=\overline{A}\overline{B}+\overline{A}\overline{C}D+AC+B\overline{C}$

(2) $F(A,B,C,D)=BC+D+\overline{D}(\overline{B}+\overline{C})(AD+B)$

(3) $F(A,B,C,D)=\prod M(2,4,6,10,11,12,13,14,15)$

(4) $F(A,B,C,D)=\sum m(0,2,7,13,15)+\sum d(1,3,4,5,6,8,10)$

5. 用卡诺图化简法求下列逻辑函数的最简"与-或"表达式。

(1) $F(A,B,C)=\sum m(0,1,2,4,5,7)$

(2) $F(A,B,C,D)=\sum m(0,1,2,3,4,6,7,8,9,11,15)$

(3) $F(A, B, C, D)=\sum m(3, 4, 5, 7, 9, 13, 14, 15)$

(4) $F(A, B, C, D)=\sum m(0, 1, 2, 5, 6, 7, 8, 9, 13, 14)$

(5) $F(A, B, C, D)=\sum m(0, 2, 3, 5, 7, 8, 10, 11)+\sum d(14, 15)$

(6) $F(A, B, C, D, E)=\sum m(0, 3, 4, 6, 7, 8, 15, 16, 17, 20, 22, 25, 27, 29, 30, 31)$

6. 用列表法化简下列逻辑函数。

(1) $F(A, B, C, D)=\sum m(0, 2, 3, 5, 7, 8, 10, 11, 13, 15)$

(2) $F(A, B, C, D)=\sum m(3, 5, 8, 9, 10, 12)+\sum d(0, 1, 2, 13)$

7. 用卡诺图化简下列多输出逻辑函数。

$F_1(A, B, C, D)=\sum m(0, 2, 4, 5, 7, 8, 10, 13, 15)$

$F_2(A, B, C, D)=\sum m(0, 2, 5, 7, 8, 10)$

$F_3(A, B, C, D)=\sum m(2, 4, 6, 13, 15)$

8. 分析如图 3-54 所示的组合逻辑电路，并画出其简化的逻辑电路图。

9. 分析如图 3-55 所示的组合逻辑电路，指出在哪些输入取值下，输出 F 的值为 1，并用“异或”门实现该电路的逻辑功能。

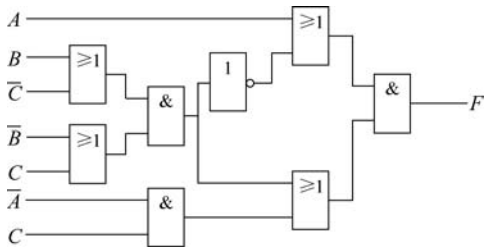


图 3-54　题 8 的图

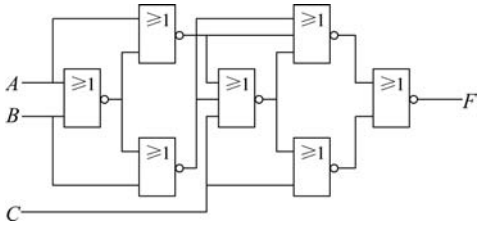


图 3-55　题 9 的图

10. 分析如图 3-56 所示的求补电路。要求写出输出函数表达式，列出真值表。

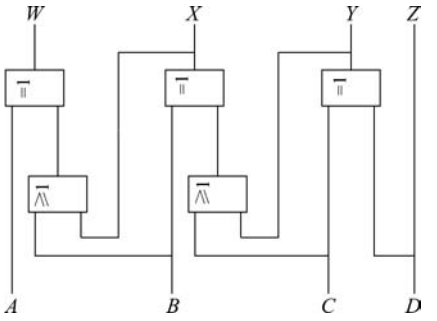


图 3-56　题 10 的图

11. 图 3-57 为两种十进制代码的转换器，输入为余 3 码，分析输出是什么代码。

12. 分析如图 3-58 所示的组合逻辑电路，假定输入是一位十进制数的 8421 码，试说明该电路的功能。

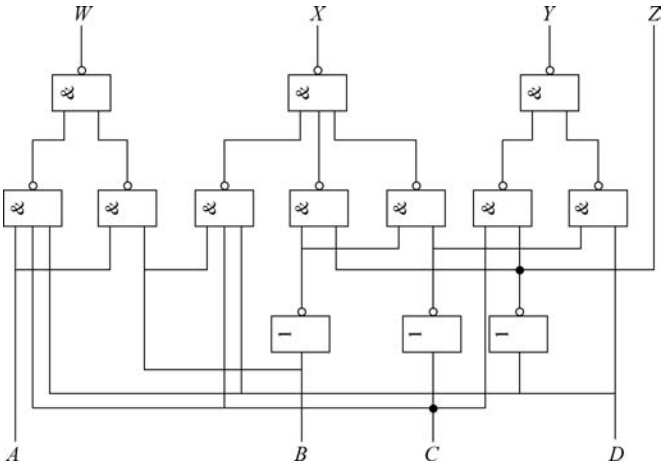


图 3-57　题 11 的图

13. 图 3-59 是一个受 M 控制的 4 位二进制自然码和 Gray 码相互转换的电路。$M=1$ 时,完成二进制自然码至 Gray 码的转换; 当 $M=0$ 时,完成相反的转换。请说明之。

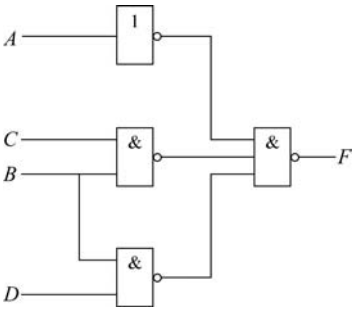


图 3-58　题 12 的图

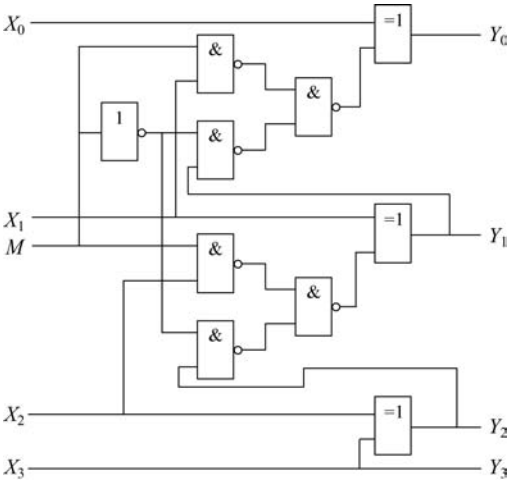


图 3-59　题 13 的图

14. 分析如图 3-60 所示的组合逻辑电路,回答以下问题:

(1) 假定电路的输入变量 A、B、C 和输出函数 F、G 均代表 1 位二进制数,请问该电路实现什么功能?

(2) 若将图中虚线框内的反向器去掉,即令 X 点和 Y 点直接相连,请问该电路实现什么功能?

(3) 若将图中虚线框内的反向器改为异或门,异或门的另一个输入端与输入控制变量 M 相连,请问该电路实现什么功能?

15. 如图 3-61 所示的组合电路中,A、B 为输入变量,S_3、S_2、S_1、S_0 为选择控制变量,F 为输出函数。试写出该电路在选择控制变量的控制下的输出函数表达式,并说明电路的功能。

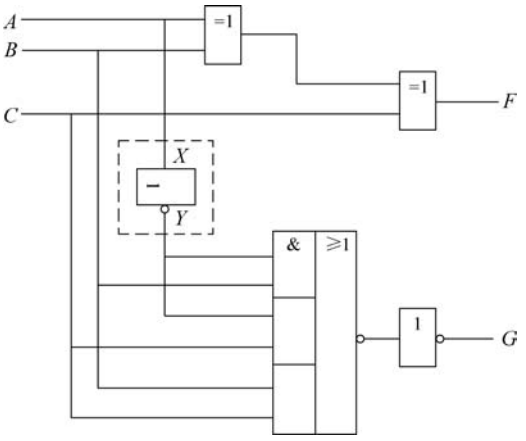


图 3-60　题 14 的图

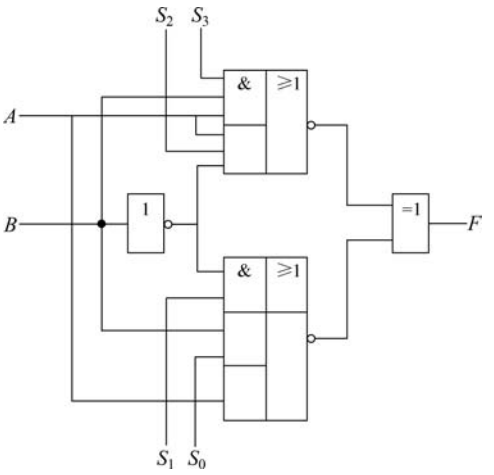


图 3-61　题 15 的图

16. 设 A、B、C 为某密码锁的三个按键，当单独按下 A 键时，锁既不打开也不报警；只有当同时按下 A、B、C 或者 A、B 或者 A、C 时，锁才能被打开；当按下不符合上述条件的按键时，将发出报警信号，试用“与非”门设计此密码锁的逻辑电路。

17. 一热水器如图 3-62 所示，图中虚线表示水位，A、B、C 电极被水浸没时会有信号输出。水面在 A，B 间时为正常状态，绿灯 G 亮；水面在 B，C 间或 A 以上时为异常状态，黄灯 Y 亮；水面在 C 以下时为危险状态，红灯 R 亮。试用“与非”门设计实现该功能的逻辑电路。

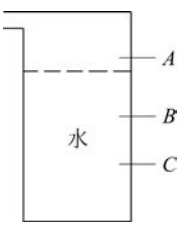


图 3-62　题 17 的图

18. 设输入既有原变量又有反变量，用“与非”门实现下列多输出函数电路。

$F_1(A, B, C, D) = \sum m(2, 4, 5, 6, 7, 10, 13, 14, 15)$

$F_2(A, B, C, D) = \sum m(2, 5, 8, 9, 10, 11, 12, 13, 14, 15)$

19. 设输入既有原变量又有反变量，试用“或非”门实现下列函数。

(1) $F(A, B, C, D) = \sum m(0, 1, 2, 4, 6, 10, 14, 15)$

(2) $F = \overline{\overline{A+B}+\overline{B+C}} \cdot \overline{A}\,\overline{B}$

20. 设计一个 2 位二进制数乘法器。该电路的输入接收两个 2 位二进制数 $A = A_2A_1$，$B = B_2B_1$，输出为 $A \times B$。

21. 设计一个 1 位二进制加/减法器，该电路在 M 的控制下进行加、减运算。当 $M = 0$ 时，实现全加器功能；当 $M = 1$ 时，实现全减器功能。

22. 用“与非”门设计下列代码转换电路。

(1) 余 3 码转换成 8421 码。

(2) 余 3 码转换成七段显示代码(用共阴极形式的 LED)。

23. 设计对 10 的补码产生器。对 10 的补码是指以 10 为模的补码。一个数的补数 $C = M - N$，此处模为 $M = 10$，N 为 0～9 这 10 个数符。对应关系如下：0 的补码为 0，1 的补码为 9，2 的补码为 8，……，9 的补码为 1。设计 $N \rightarrow C$ 的转换电路(不考虑符号)，N 和 C 均为 8421 码形式。

24. 奇偶校验电路是一种检查数据在传输中是否出现错误的电路。发送端不仅发出数据,还发出监督码(也称奇偶校验位),它们一起组成传输码。监督码的作用是使传输码中1的个数为奇数或者为偶数。在接收端有一奇偶校验电路,它的输入为发送端发出的传输码。接收端的奇偶校验电路输出为1,表明数据在传输中没有出现错误;若奇校验电路输出为0,则说明数据在传输中出现了错误。如图3-63所示,设发送端发送的数据是一位8421码,请用逻辑门电路设计图3-63中的奇偶发生器和奇偶校验器(假设采用奇校验)。

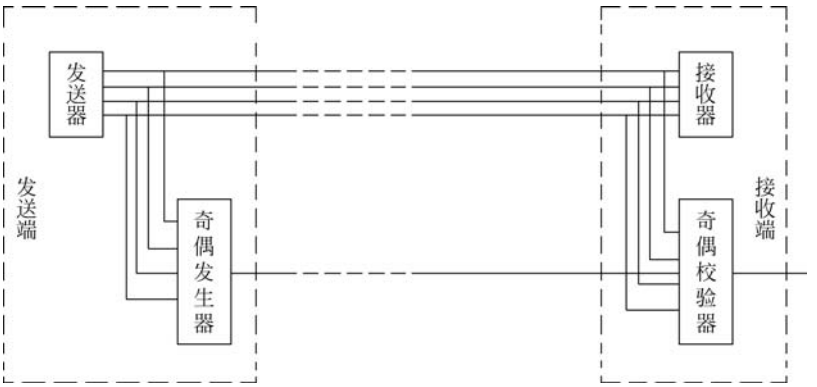


图3-63　题24的图

25. 用VHDL描述一个组合逻辑电路,该电路接收两个1位二进制数A和B,并比较其大小,当$A>B$时,输出1;否则输出0。

26. 用VHDL描述一个组合逻辑电路,该电路的输入为1位十进制数的8421码,当输入的十进制数字为素数时,输出为1;否则为0。

27. 用VHDL描述一个1位十进制数的数值范围指示器。电路的输入为一位十进制数的8421码,当输入的十进制数大于或等于5时,输出为1;否则为0。

28. 判断下列函数是否可能发生竞争?竞争结果是否会产生险象?在什么情况下产生险象?若可能产生险象,试用增加冗余项的方法消除。

(1) $F_1=AB+A\overline{C}+\overline{C}D$;(2) $F_2=AB+\overline{A}CD+BC$;(3) $F_3=(A+\overline{B})(\overline{A}+\overline{C})$。

29. 判断如图3-64所示的电路有无险象?若有,请说明出现险象的输入条件,经修改设计后画出无险象的电路图。

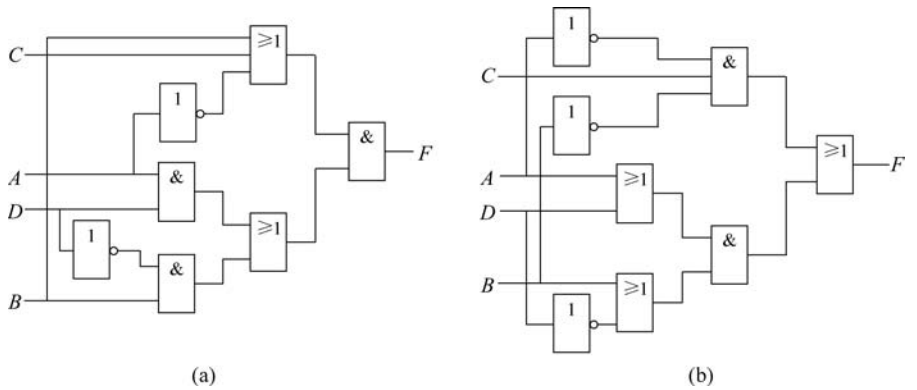


图3-64　题29的图

30. 阅读下面的 VHDL 程序,分析其实现的功能。

(1)
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY vote3 IS
   PORT(a,b,c:IN STD_LOGIC;
        f:OUT STD_LOGIC);
END vote3;
ARCHITECTURE behavor OF vote3 IS
   SIGNAL temp1,temp2,temp3:STD_LOGIC;
BEGIN
   temp1 <= NOT(a AND b);
   temp2 <= NOT(b AND c);
   temp3 <= NOT(a AND c);
   f <= NOT(temp1 AND temp2 AND temp3);
END behavor;
```

(2)
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY tri_state_gate IS
   PORT(din:IN STD_LOGIC;
        en:IN STD_LOGIC;
        dout:OUT STD_LOGIC);
END tri_state_gate;
ARCHITECTURE behavor OF tri_state_gate IS
BEGIN
   PROCESS(din,en)
   BEGIN
      IF (en = '1') THEN
         dout <= din;
      ELSE
         dout <= 'Z';
      END IF;
   END PROCESS;
END behavor;
```

(3)
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY single_dir_bus8 IS
   PORT(a:IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        en :IN STD_LOGIC;
        b:OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END single_dir_bus8;
ARCHITECTURE behavor OF single_dir_bus8 IS
BEGIN
   PROCESS(a,en)
   BEGIN
      IF (en = '1') THEN
         b <= a;
      ELSE
         b <= (OTHERS =>'Z'); -- 等价于 b <= "ZZZZZZZZ";
      END IF;
```

```
    END PROCESS;
END behavor;
```

(4)
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY bi_dir_bus8 IS
    PORT(ain:IN STD_LOGIC_VECTOR(7 DOWNTO 0);
         bin:IN STD_LOGIC_VECTOR(7 DOWNTO 0);
         en :IN STD_LOGIC;
         dir:IN STD_LOGIC;
         aout:OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
         bout:OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END bi_dir_bus8;
ARCHITECTURE behavor OF bi_dir_bus8 IS
BEGIN
    bout <= ain WHEN en = '1' AND dir = '1'
                ELSE (OTHERS => 'Z');
    aout <= bin WHEN en = '1' AND dir = '0'
                ELSE (OTHERS => 'Z');
END behavor;
```

(5)
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY comp_code IS
    PORT(din:IN STD_LOGIC_VECTOR(7 DOWNTO 0);
         dout:OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END comp_code;
ARCHITECTURE rtl OF comp_code IS
BEGIN
    dout <= NOT(din) + '1';
END rtl;
```