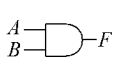
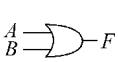
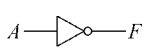
第3章 数字电路和逻辑电路基础

数字集成电路是计算机功能得以实现的重要物理基础。目前人们就像使用“积木”一样，用集成电路搭建成各种功能的计算机。本章主要介绍计算机中常用的逻辑部件和逻辑电路。

3.1 布尔代数

布尔代数有三种基本逻辑操作——“与(逻辑乘，符号·)”、“或(逻辑加，符号+)”，“非(求反，符号－或′)”。它们的逻辑含义和图形表示如表3.1所示。

表3.1 基本的逻辑函数

名 称	图形表示	函数表示	真值表
AND		$F=A\cdot B$ or $F=AB$	A B | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F=A+B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NOT		$F=\overline{A}$ or $F=A'$	A | F 0 | 1 1 | 0

任何其他的逻辑函数都可以通过这三类基本函数构成。布尔代数是以命题为对象，包含三个基本逻辑操作的完整的代数学，它可以对命题进行运算，其运算的基本依据是如表3.2所示的基本规则。

表 3.2　布尔运算的基本规则

表达式(与相关)	表达式(或相关)	基本定律
$A \cdot B = B \cdot A$	$A+B=B+A$	交换律
$A \cdot (B+C)=(A \cdot B)+(A \cdot C)$	$A+(B \cdot C)=(A+B) \cdot (A+C)$	分配率
$A \cdot A'=0$	$A+A'=1$	互补率
$A \cdot (A+B)=A$	$A+AB=A$	吸收率
$A \cdot (A'+B)=AB$	$A+A'B=A+B$	
$0 \cdot A=0$	$1+A=1$	0-1 律
$1 \cdot A=A$	$0+A=A$	
$A \cdot A=A$	$A+A=A$	等幂律
$A \cdot (B \cdot C)=(A \cdot B) \cdot C$	$A+(B+C)=(A+B)+C$	结合律
$(A \cdot B)'=A'+B'$	$(A+B)'=A' \cdot B'$	德·摩根律

3.2　逻辑函数的化简

将一个复杂的逻辑函数表示成一个简单、等价的逻辑函数，称为化简。化简能简化逻辑函数实现的电路。常用的化简方法有代数化简法和卡诺图化简法。

3.2.1　代数化简法

代数化简法是直接利用布尔代数运算规则进行化简的一种方法。

【例 1】　化简 $F=A'B'C'+A'B'C+A'BC+ABC'+ABC$

【解】

$$\begin{aligned} F &= A'B'C'+A'B'C+A'BC+ABC'+ABC \\ &= A'B'C'+A'B'C+A'BC+ABC+ABC+ABC' \\ &= (A'B'C'+A'B'C)+(A'BC+ABC)+(ABC+ABC') \\ &= A'B'(C'+C)+BC(A'+A)+AB(C'+C) \\ &= A'B'+BC+AB \end{aligned}$$

3.2.2　卡诺图化简法

卡诺图(Karnaugh Map)是 20 世纪 50 年代美国工程师卡诺(M. Karnaugh)提出来的，是逻辑函数的一种图形直观表示方法。

卡诺图中每一个方格里面是一个最小项。由全部变量或其反变量形成的逻辑乘积项称为最小项(Minterm)。例如，有 AB 两个变量，它的最小项有 $A'B'$，$A'B$，AB' 和 AB。图 3.1(a)中方格里的 00，01，10 和 11 对应着前面 4 个最小项。图 3.1(b)中每个方格里列出的是相应最小项的逻辑值，对应着图 3.1(c)中的 F 值。可以看出卡诺图与真值图之间存在一一对应关系。对于 n 个变量，则存在 2^n 个最小项，在卡诺图中也需要划分 2^n 个小方格。可以将 n 个变量分成一组，如图 3.2(a)所示，也可以将 n 个变量分成两组，如 $ABCD$ 可以分成 AB 和 CD 两组，ABC 可以分成 A 和 BC 两组，ABC 和 $ABCD$ 的卡诺图分别如图 3.2(b)和图 3.2(c)所示。$ABCD$ 4 个变量的卡诺图还可以简化为图 3.2(d)。

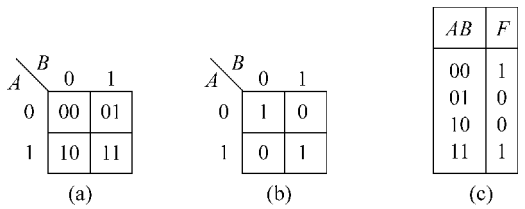


图 3.1 卡诺图与真值图的对应关系

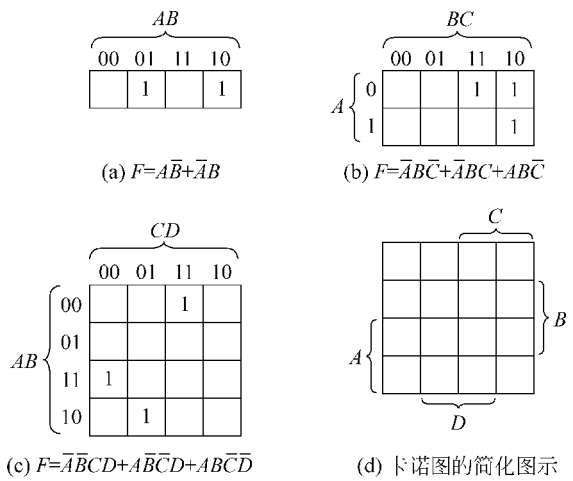


图 3.2 卡诺图

需要注意的是，相邻小格逻辑变量的海明距离必须为 1，也就是只允许一个逻辑变量是相反的，其他逻辑变量必须相同。下面通过实例来阐述化简过程。

【例 2】 化简 $F=A'C'D'+A'BC'+A'BD+B'C'D'+AC+BCD+AB'D'$。

【解】 首先将函数展开为最小项表达式。任一个函数都可以展开为若干个最小项之和。如 $ABC=ABCD+ABCD'$。然后将展开的结果在卡诺图中进行标识，如果存在某个最小项，则该最小项的方格里填 1，否则不填或填 0，如图 3.3 所示。

化简是通过相邻小块的合并来完成的。任意两个相邻小格构成的块叫一维块，相邻两个一维块构成二维块，同理可以进一步构成高维块。相邻块指的是海明距离为 1 的两个块，例如图 3.3(a)和图 3.3(b)中 $A'BC'D$ 和 $A'BCD$，$A'B'C'D$ 和 $AB'C'D$ 都是相邻块，相应的相邻块可以合并成 $A'BD$ 和 $B'C'D$。可见合并的过程就是变量减少的过程，也就是化简的过程。函数 F 的化简结果为 $F=A'C'D'+AB'D'+AC+A'BD$，如图 3.4 所示。可以看出相邻块的合并方案并不是唯一的，如函数 F 可以化简得到另外一种结果，如图 3.4(c)所示。$F=B'C'D'+A'BC'+AC+A'BD$，但这两种结果都是等价的。卡诺图的使用也有限制，当逻辑变量迅速增加后，卡诺图的使用就不方便了。

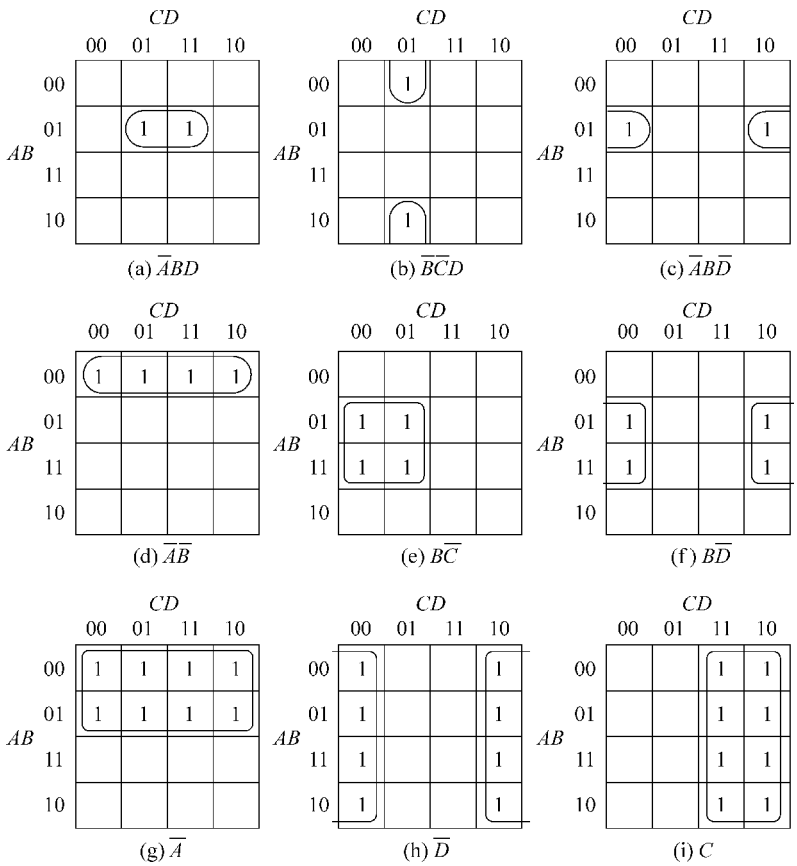


图 3.3　卡诺图的化简过程

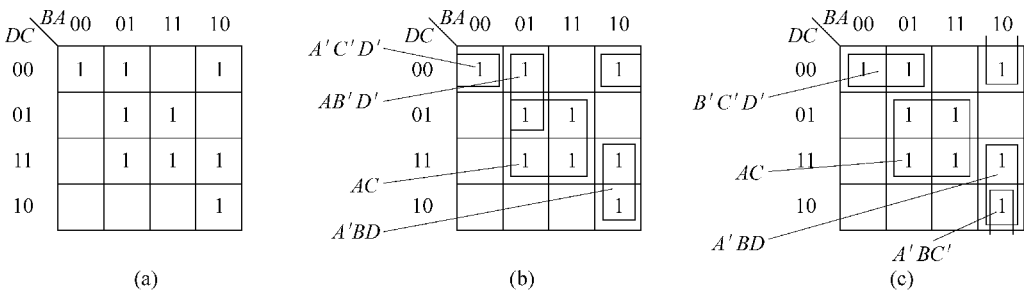


图 3.4　使用卡诺图化简逻辑函数

3.3　组合逻辑电路

如果逻辑电路的输出状态仅和当时的输入状态有关，而与过去的输入状态无关，这种逻辑电路称为组合逻辑电路。常见的组合逻辑电路有加法器，算术逻辑单元，译码器和数据选择器。

3.3.1 加法器

加法器是计算机中基本运算部件之一。不考虑进位输入的加法称为半加。半加的真值表如表 3.3 所示。其实现电路图如图 3.5 所示。逻辑表达式如下。

$$H_n = X_n \oplus Y_n$$

表 3.3 半加的真值表

X_n	Y_n	H_n
0	0	0
0	1	1
1	0	1
1	1	0

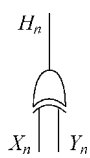


图 3.5 半加的实现电路图

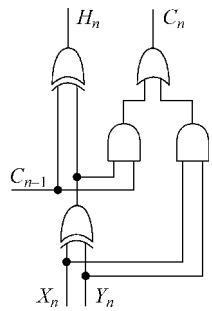


图 3.6 全加器的电路图

考虑前一位进位输入的加法称为全加，其真值表如表 3.4 所示。其实现电路图如图 3.6 所示。逻辑表达式如下。

$$H_n = X_n \oplus Y_n \oplus C_{n-1}$$
$$C_n = X_n Y_n + (X_n \oplus Y) C_{n-1}$$

表 3.4 全加的真值表

X_n	Y_n	C_{n-1}	H_n	C_n
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

一般来说，有 n 位的加法就需要 n 个全加器电路，同时要解决从一个单元到下一个单元的进位传送，这就是行波进位加法器。其原理图如图 3.7 所示。

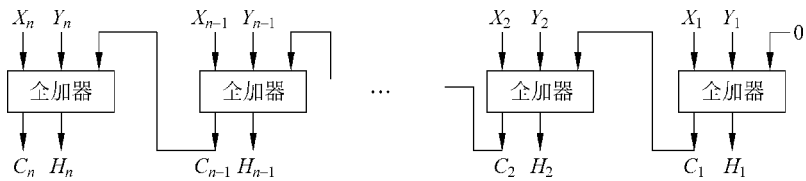


图 3.7　行波进位加法器

行波进位加法器需要经过一步一步传递，第 n 位的加法必须等到第 $n-1$ 位的进位到来后才能进行，加法时间与位数有关。所以必须改变进位的传递路径，才能提高加法器的工作速度，解决办法可以使用“超前进位形成电路”来同时形成各位进位，从而实现快速加法。这种方法被称为“超前进位加法器”。

超前进位产生电路是根据各位进位的形成条件来形成的。只要满足下列条件中的任一个，就可形成 C_1：①X_1 和 Y_1 均为 1；②X_1，Y_1 任一个为 1，且 C_0 为 1。由此可以写出 C_1 的表达式。

$$C_1 = X_1Y_1 + (X_1 + Y_1)C_0$$

只要满足下列条件中的任一个，就可形成 C_2：①X_2 和 Y_2 均为 1；②X_2，Y_2 任一个为 1，且 X_1 和 Y_1 均为 1；③X_2，Y_2 任一个为 1，同时 X_1 和 Y_1 任一个为 1，且 C_0 为 1。由此可以写出 C_2 的表达式。

$$C_2 = X_2Y_2 + (X_2 + Y_2)X_1Y_1 + (X_2 + Y_2)(X_1 + Y_1)C_0$$

同理可得

$$\begin{aligned}C_3 =& X_3Y_3 + (X_3 + Y_3)X_2Y_2 + (X_3 + Y_3)(X_2 + Y_2)X_1Y_1 \\ &+ (X_3 + Y_3)(X_2 + Y_2)(X_1 + Y_1)C_0\\ C_4 =& X_4Y_4 + (X_4 + Y_4)X_3Y_3 + (X_4 + Y_4)(X_3 + Y_3)X_2Y_2 \\ &+ (X_4 + Y_4)(X_3 + Y_3)(X_2 + Y_2)X_1Y_1 \\ &+ (X_4 + Y_4)(X_3 + Y_3)(X_2 + Y_2)(X_1 + Y_1)C_0\end{aligned}$$

下面引进进位传递函数和进位产生函数的概念，它们的定义为：

$$\begin{cases}P_i = X_i + Y_i \\ G_i = X_iY_i\end{cases}$$

其中，P_i 的意义是：当 X_i 和 Y_i 中有一个为 1 时，若有进位 C_{i-1} 输入，则进位会向 $i+1$ 位传递，这个进位可看成低位进位越过本位直接向高位传递的。而 G_i 的意义是：当 X_i，Y_i 均为 1 时，不管有无进位 C_{i-1} 输入，定会向 $i+1$ 位传递进位。

将 P_i 和 G_i 代入 $C_1 \sim C_4$ 的表达式，可得：

$$\begin{cases}C_1 = G_1 + P_1C_0 \\ C_2 = G_2 + P_2G_1 + P_2P_1C_0 = G_{1,2} + P_{1,2}C_0 \\ C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1C_0 = G_{1,3} + P_{1,3}C_0 \\ C_4 = G_4 + P_4G_3 + P_4P_3G_2 + P_4P_3P_2G_1 + P_4P_3P_2P_1C_0 = G_{1,4} + P_{1,4}C_0\end{cases}$$

其中，$P_{1,2} = P_2P_1$，同理 $G_{1,3}$，$P_{1,3}$，$G_{1,4}$，$P_{1,4}$ 等都能得到类似的表达式。更为普遍的，能够得到递推关系式：

$G_{i,k} = G_{j+1,k} + P_{j+1,k}G_{i,j}$，$P_{i,k} = P_{i,j}P_{j+1,k}(i > 0, i \leqslant j < k)$，且 $G_{i,i} = G_i$，$P_{i,i} = P_i$

根据上述的递推关系，可以设计超前进位加法器如图 3.8 所示。首先信号从顶部流到底部，计算 P 和 G 的不同值，然后根据 C_i 的计算式计算 C_i 的值。最后在树的底部送入 C_0，所有的进位都在顶部输出。这种方法比行波进位加法器的效率提高了很多倍，通过在空间上的小投入换来了速度上的巨大改善。

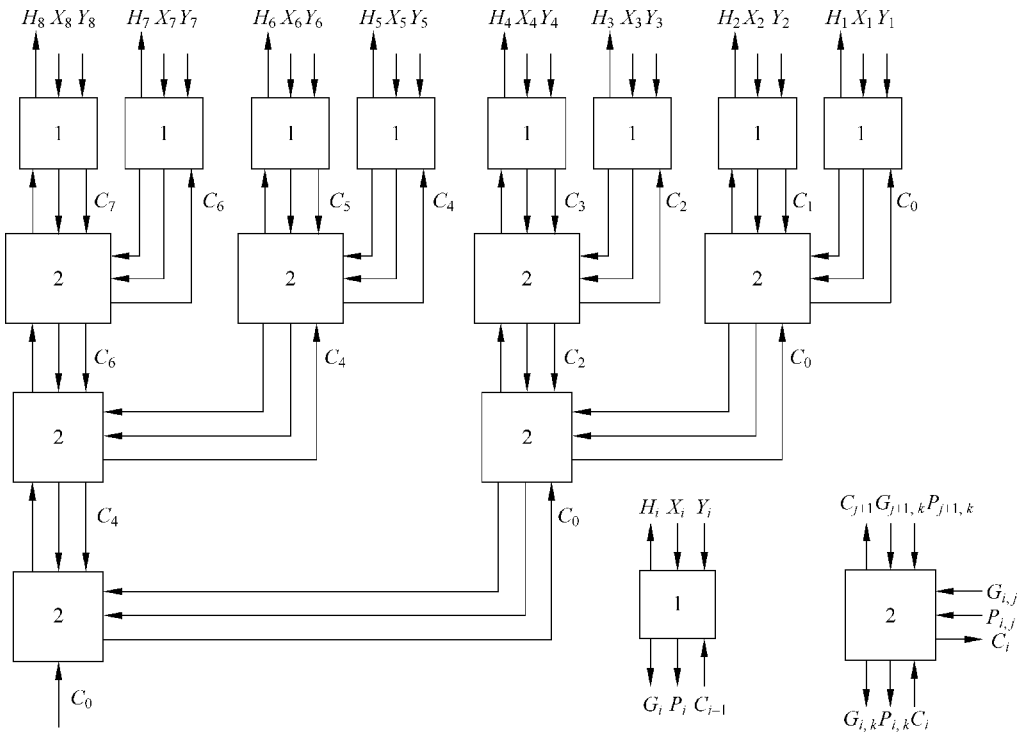


图 3.8　超前进位产生电路

3.3.2　译码器

二进制译码器的输入是一组二进制代码，输出的是一组高、低电平信号。

如图 3.9 所示的是 2 线-4 线二进制译码器的框图。输入的两位二进制代码 A_1A_0 共有 4 种状态，译码器将每个输入代码译成对应的输出线上的低电平信号，如表 3.5 所示。E'_n 为使能输入端，低电平有效。

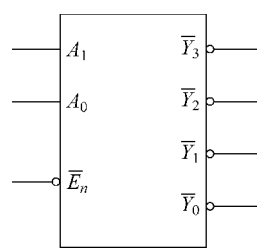


图 3.9　2 线-4 线译码器的框图

表 3.5　2 线-4 线译码器的真值表

输　入			输　出			
E'_n	A_1	A_0	Y'_3	Y'_2	Y'_1	Y'_0
1	×	×	1	1	1	1
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1

相应的函数表达式为：

$$\overline{Y}_3 = \overline{\overline{E}_n \cdot A_1 \cdot A_0}$$

$$\overline{Y}_2 = \overline{\overline{E}_n \cdot A_1 \cdot \overline{A}_0}$$

$$\overline{Y}_1 = \overline{\overline{E}_n \cdot \overline{A}_1 \cdot A_0}$$

$$\overline{Y}_0 = \overline{\overline{E}_n \cdot \overline{A}_1 \cdot \overline{A}_0}$$

其逻辑实现图如图 3.10 所示。

在实际应用中，译码器常用来进行地址的译码。其详细情况将在第 5 章中介绍。

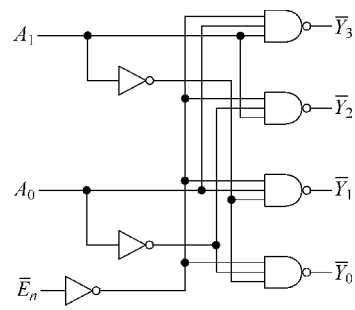


图 3.10　2 线-4 线二进制译码器的逻辑图

3.3.3　算术逻辑单元

算术逻辑单元简称为 ALU，是计算机的核心逻辑部件，能进行各种算术运算和逻辑运算。简单的 ALU 能够完成“与”，“或”，“非”等逻辑运算和“加”、“减”等算术运算。

图 3.11 中使用了一个两位的译码器，根据控制信号，F_0 和 F_1 产生 4 种不同的使能信号，同一时刻只能有一个输出为 1。

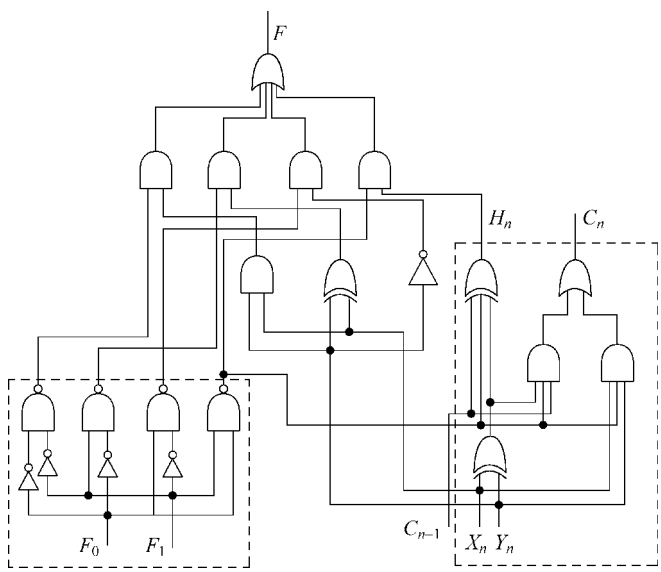


图 3.11　算术逻辑单元的电路实现图

其中左下角是一个 2-4 的译码器，右下角是一个全加器。根据 F_0 和 F_1 的输入，选择适当的计算电路得到最终的输出。

ALU 的功能框图如图 3.12 所示。

因为 ALU 是组合逻辑电路，因此实际应用 ALU 时，其输入端口 A 和 B 必须和锁存器相连，而且在运算过程中锁存器的内容是不变的。其输出也必须送到寄存器中保存，现有的芯片中将寄存器和 ALU 电路集成在一个芯片内，如 29C101，其框图如图 3.13 所示。

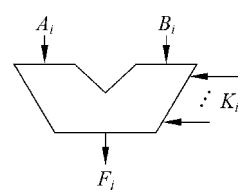


图 3.12　ALU 的框图

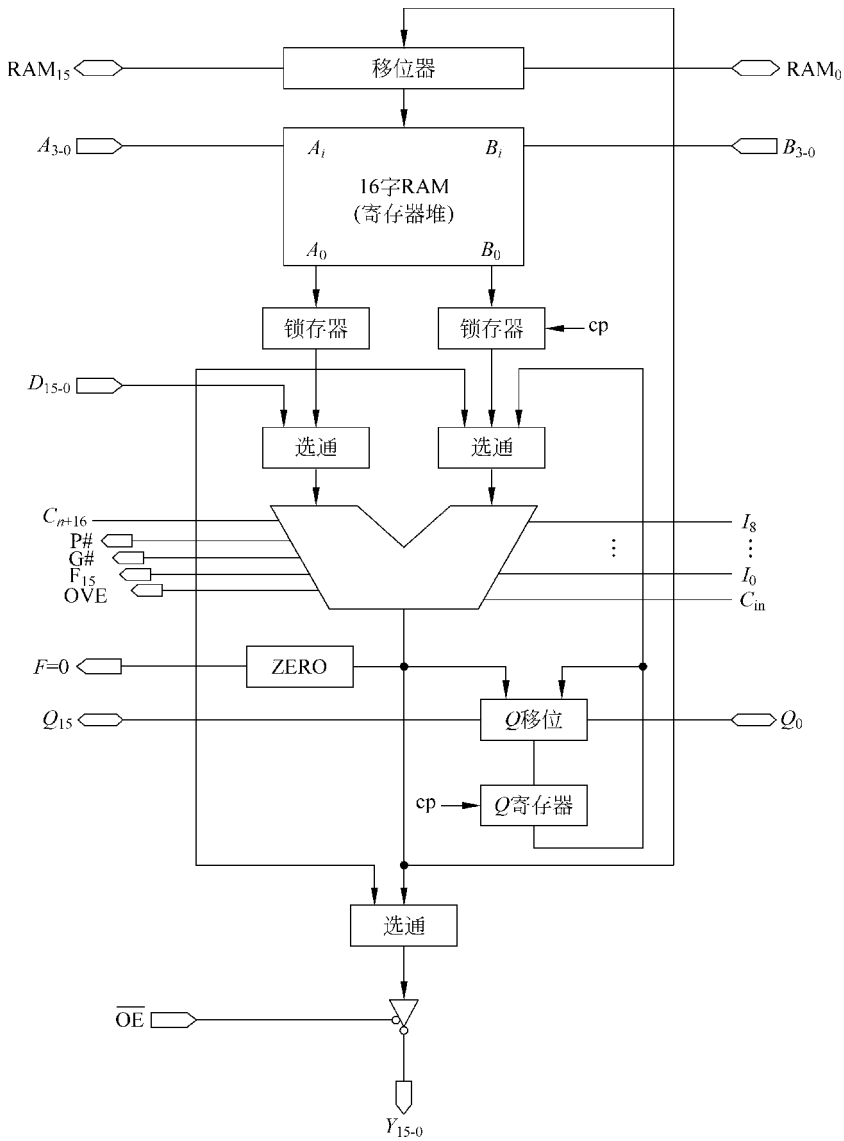


图 3.13　29C101 框图

该芯片的核心部件是一个容量为 16 字的双端口 RAM 和一个高速 ALU 电路。

RAM 可视为由 16 个寄存器组成的寄存器堆。只要给出 A_i 口或 B_i 口的 4 位地址，就可以从 A_0 出口或 B_0 出口读出对应于出口地址的存储单元内容。写入时，只能写入到由 B_i 口指定的那个单元内。参与操作的两个数分别由 RAM 的 A_0、B_0 出口输出至两个锁存器中。

ALU 则受 $I_8 \sim I_0$ 控制，I_1、I_0 控制 ALU 的数据源；I_5、I_4、I_3 控制 ALU 所能完成的三种算术运算和 5 种逻辑运算；$I_8 \sim I_6$ 用来控制 RAM 和 Q 移位器，决定是否移位以及 Y 口输出是来自 RAM 的 A 出口还是 ALU 的 F 出口。

ALU 的 C_{in} 为低位来的外来进位，C_{n+16} 为向高位的进位，可供 29C101 级联时用。ALU

结果为 0 时，$F=0$ 可直接输出，OVR 为溢出标记。而 $\overline{P}$、$\overline{Q}$ 可供先行进位方式时使用。ALU 的输出可直接通过移位器存入 RAM，也可通过选通门在$\overline{OE}$有效时，从 $Y_{15}\sim Y_0$ 输出。Q 寄存器主要为乘法和除法服务，$D_{15}\sim D_0$ 为 16 位立即数的输入口。

3.3.4　数据选择器

数据选择器是在地址选择信号的控制下，从多路输入数据中选择一路作为输出的电路，也叫多路开关，简称 MUX。4 选 1 数据选择器的真值表如表 3.6 所示，实现原理示意图如图 3.14 所示。$D_3\sim D_0$ 为 4 个数据输入端，F 为数据输出端，A_1、A_0 是地址选择输入端。当 A_1、A_0 为不同代码时，$D_3\sim D_0$ 中相应输入通道数据送至输出端 F。E'为使能端，低电平时数据选择器正常工作。

表 3.6　4 选 1 数据选择器的真值表

$\overline{E}$	A_1	A_0	D_0	D_1	D_2	D_3	F
0	0	0	0	×	×	×	0
0	0	0	1	×	×	×	1
0	0	1	×	0	×	×	0
0	0	1	×	1	×	×	1
0	1	0	×	×	0	×	0
0	1	0	×	×	1	×	1
0	1	1	×	×	×	0	0
0	1	1	×	×	×	1	1
1	×	×	×	×	×	×	×

函数表达式为：

$$F=\overline{E}\overline{A}_1\overline{A}_0D_0+\overline{E}\overline{A}_1A_0D_1+\overline{E}A_1\overline{A}_0D_2+\overline{E}A_1A_0D_3$$

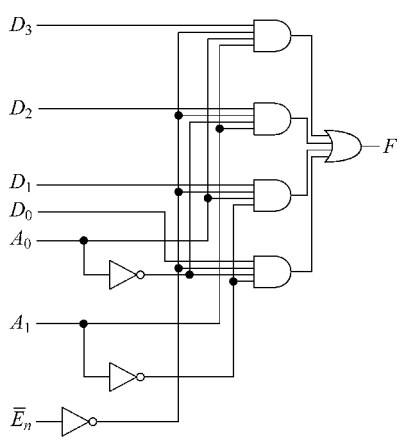


图 3.14　4 选 1 数据选择器的逻辑实现图

3.4 时序逻辑电路

组合逻辑电路的输出仅与门电路有关，与电路的原状态没有关系，也就是网络中没有记忆特性。时序逻辑电路的特点是任意时刻的输出不仅取决于该时刻的输入，而且和电路的原状态有关。时序逻辑电路逻辑功能上的特点决定了时序逻辑电路在结构上的特点是：①电路上包含存储元件，通常由触发器构成；②存储元件的输出和电路的输入之间存在反馈连接，其结构框图如图 3.15 所示。

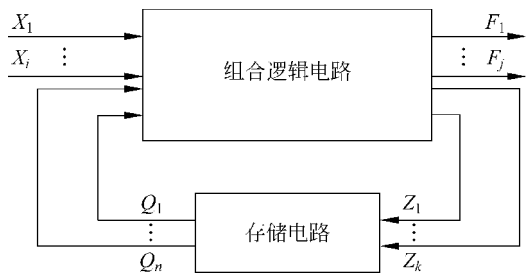


图 3.15 时序逻辑电路框图

从图 3.15 中可以看出，时序电路由组合逻辑电路和存储电路两部分组成。图中包含两组输入和两组输出，其中 $X(X_1,\cdots,X_i)$ 代表时序电路的外部输入；$Q(Q_1,\cdots,Q_n)$ 代表时序电路的内部输入，它是存储元件的输出反馈到组合逻辑电路的输入；$F(F_1,\cdots,F_j)$ 代表时序电路的外部输出，$Z(Z_1,\cdots,Z_k)$ 代表时序电路的内部输出。时序电路通过存储电路来“记忆”时序电路的过去状态，而时序电路的外部输出 F 和内部输出 Z 都和外部输入 X 和存储元件的“记忆”Q 有关。

3.4.1 触发器

触发器(Flip-flop)是能够存储一位二进制的逻辑电路，是时序逻辑电路的存储部件。触发器按照功能可分为 S-R 触发器、D 触发器以及 J-K 触发器等；按照触发方式的不同可以分为时钟控制主从触发和边沿触发等类型。本节只介绍触发器的外部特性，目的是供设计相关电路时正确使用。

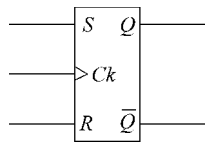
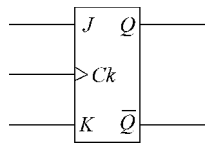
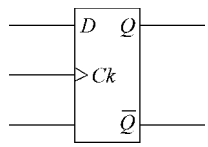
1. 同步 S-R 触发器

同步 S-R 触发器的逻辑符和特征表如表 3.7 所示。其中，R 为置 0 端，S 为置 1 端，均为高电平有效。只有当时钟脉冲到来时，SR 才能使触发器置 0 和置 1，SR 不允许同时为 1。Q 和 Q' 表示触发器的状态，也是它的输出端。

由表 3.7 的特征表，可以写出 S-R 触发器的次态方程为：

$$Q_{n+1} = S + \bar{R}Q_n$$

表 3.7　几种常用的触发器

名　称	图形表示	特征表
S-R		S R Q_{n+1} 0 0 Q_n 0 1 0 1 0 1 1 1 —
J-K		J K Q_{n+1} 0 0 Q_n 0 1 0 1 0 1 1 1 $\overline{Q_n}$
D		D Q_{n+1} 0 0 1 1

2. J-K 触发器

在同步 S-R 触发器中，必须限制输入 R 和 S 同时为 1 的情况，这给使用带来不便。为了从根本上消除这种情况，人们设计了 J-K 触发器。逻辑图和特征表如表 3.7 所示。当时钟脉冲未到的时候，触发器状态保持不变；当时钟脉冲到来时，如果 $J=0$，$K=0$，触发器保持原状态不变；如果 $J=0$，$K=1$，无论触发器的现态如何，其次态总是 0；如果 $J=1$，$K=0$，无论触发器的现态如何，其次态总是 1；如果 $J=1$，$K=1$，触发器必将翻转。

由表 3.7 的特征表，可以写出 J-K 触发器的次态方程为：

$$Q_{n+1} = J\overline{Q}_n + \overline{K}Q_n$$

3. D 触发器

D 触发器又叫 D 锁存器或延迟触发器，其逻辑符号和特征表如表 3.7 所示。D 触发器的逻辑功能比较简单，当输入 D 为 1 时，来一个时钟脉冲，触发器就被置为 1；当输入 D 为 0 时，触发器就被置为 0。其次态方程为：

$$Q_{n+1} = D$$

3.4.2　寄存器和移位寄存器

寄存器用来暂存参与运算的数据和运算结果。一个寄存器只能寄存一位二进制数，要存多位数时，就得用多个触发器。常用的有 4 位，8 位，16 位和 32 位寄存器。

寄存器常分为数码寄存器和移位寄存器两种，其区别在于是否有移位的功能。

1. 数码寄存器

数码寄存器用于寄存数据，不能进行数据移位。一般寄存器是借助时钟脉冲的作用把数据寄存在触发器内，寄存数据的位数与寄存器的个数是一致的，尤其是实现方阵字位数据的寄存时，需要使用触发器阵列构成寄存阵列。

图 3.16 是使用 4 个 D 触发器构成的寄存器的逻辑图。该寄存器有一个 Load 引脚，与 CLK 时钟信号相与之后输出到各个触发器的时钟输入端。当 Load＝0 时，输出端 $Q_0 \sim Q_3$ 与输入端 $D_0 \sim D_3$ 的状态无关，寄存器会保持原有的值不变，当 Load＝1 时，寄存器就会将输入端 $D_0 \sim D_3$ 并行送入 4 个 D 触发器的输出端 $Q_0 \sim Q_3$，新的数据被锁存到寄存器中。

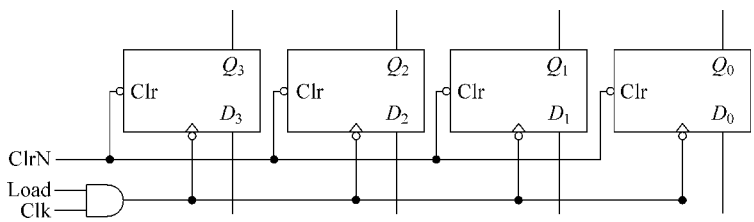


图 3.16　由 4 个 D 触发器构成的数码寄存器

2. 移位寄存器

移位寄存器是在数码寄存器的基础上发展起来的。除了具有寄存数据的功能以外，还具有移位功能。在移位操作时，要求每来一个时钟脉冲，寄存器中存放的数据就顺次左移或者右移一位。

图 3.17 是移位寄存器的逻辑图。当 Shift＝0 时，寄存器中的内容保持不变，当 Shift＝1 时，移位发生在时钟脉冲的上升沿处。设寄存的二进制数为“1011”，按移位脉冲的工作节拍从高位到低位依次串行送到 Serial Out(SO)端。工作之初先清零，首先 Serial In(SI)端输入 1，第一个移位脉冲的上升沿到来时使第一个触发器翻转，Q_3＝1，其他输出为 0，接着 SI 输入 0，第二个移位脉冲使得第一个和第二个触发器翻转，Q_3＝0，Q_2＝1，直到第四个脉冲到来时，$Q_3 \sim Q_0$ 4 个触发器的输出分别为 1101，这时可以从 4 个触发器的 Q 端得到并行的数码输出。

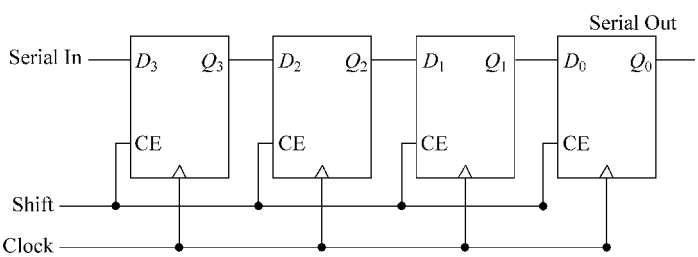


图 3.17　由 D 触发器构成的移位寄存器

3.4.3　计数器

计数功能就是累计输入脉冲的个数。实现计数功能的数字电路被称为计数器。计数器在循环中的状态个数叫做计数器的模。在计算机、数控设备及各种数字仪表中，计数器得到了广泛的应用。本节主要考虑同步计数器(Synchronous Counter)，同步计数器的特点是各触发器受同一时钟脉冲控制，同步更新状态。计数器可以使用二进制计数，也可以使用十进制计数，本节先讨论相对简单的二进制计数器。

二进制计数器可以使用 D 触发器来完成。使用 4 个 D 触发器可以计数 0000～1111 共 16 个状态，也就是该计数器的模为 16。逻辑图如图 3.18 所示。假定初始状态所有的触发器都是 0，当第一个时钟脉冲来的时候，计数器将变为 0001，再来一个脉冲，计数器将变成 0010，等到计数到了 1111 的时候再来一个脉冲，计数器将变成 0000。

经过分析，各个触发器的输出和输入之间的关系如下。

(1) 第一位输出 Q_0，每来一个计数脉冲都需要翻转一次，故 $D_0=Q_0'=Q_0\oplus 1$。

(2) 第二位输出 Q_1，在 $Q_0=1$ 时来一次翻转，故 $D_1=Q_1\oplus Q_0$。

(3) 第三位输出 Q_2，在 Q_0 和 Q_1 为 1 时来一次翻转，故 $D_2=Q_2\oplus Q_0\ Q_1$。

(4) 第四位输出 Q_3，在 Q_0、Q_1 和 Q_2 为 1 时来一次翻转，故 $D_3=Q_3\oplus Q_0\ Q_1\ Q_2$。

其逻辑图如图 3.18 所示。

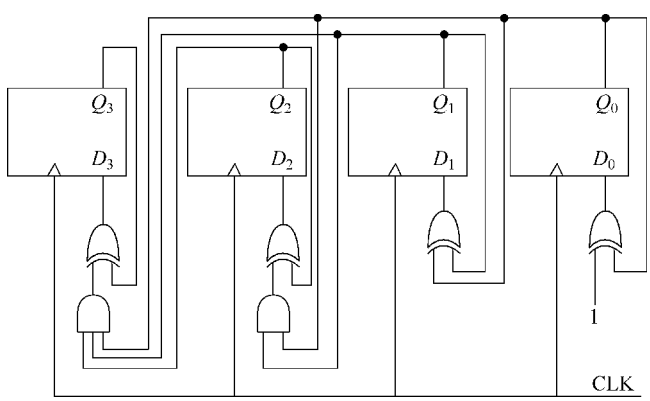


图 3.18　由 4 个 D 触发器构成的二进制计数器

十进制计数器在二进制的基础上进行改进，所不同的是当计数值编程 1001 时，计数器会变成 0000，而不是 1010。各个触发器的输出和输入之间的关系如下。

(1) 第一位输出 Q_0，每来一个计数脉冲都需要翻转一次，故 $D_0=Q_0'=Q_0\oplus 1$。

(2) 第二位输出 Q_1，在 $Q_0=1$ 时来一次翻转，而在 $Q_3=1$ 时不得翻转，故 $D_1=Q_0Q_3'\oplus Q_0$。

(3) 第三位输出 Q_2，在 Q_0 和 Q_1 为 1 时来一次翻转，故 $D_2=Q_2\oplus Q_0\ Q_1$。

(4) 第四位输出 Q_3，在 Q_0、Q_1 和 Q_2 为 1 时来一次翻转，并且在输出为 1001 的时候也需要翻转一次，故 $D_3=Q_3\oplus(Q_0\ Q_1\ Q_2+Q_0\ Q_3)$。

其逻辑图如图 3.19 所示。

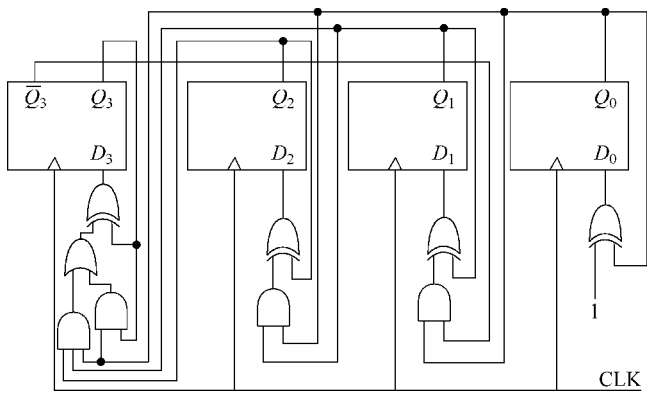


图 3.19 由 4 个 D 触发器构成的十进制计数器

3.5 阵列逻辑电路

阵列逻辑电路近年来得到了迅速的发展,“阵列”是指逻辑元件在硅芯片上以阵列形式排列,这种电路具有设计方便、芯片面积小、产品成品率高、用户自编程、减少了系统的硬件规模等优点。阵列逻辑电路主要有半导体存储器和可编程逻辑器件等。

半导体存储器是一种能存储大量的二值数据的半导体器件。半导体存储器的种类很多,并有不同的分类方法。根据半导体存储器的存取方式可分为两类,其中一类是只读存储器(Read Only Memory,ROM),另一类是随机存取储存器(Random Access Memory,RAM)。

只读存储器只能读取数据,而不能写入信息。ROM 的特点就是具有非易失性(Non-Volatile),即在一定条件下将二值数据写入 ROM 后,这些数据不会因为芯片掉电而损失,所以在计算机的 BIOS 中就使用 ROM 来存储相关函数。只读存储器按照编程方式可以划分为掩模式 ROM,PROM,EPROM,E^2PROM 和 Flash Memory 等。

随机存储器 RAM 可以根据需要随时向存储器里写入数据或者从中读取数据,根据所采用的存储单元工作原理的不同,可以将随机存储器分为静态随机存储器(Static RAM,SRAM)和动态随机存储器(Dynamic RAM,DRAM)。

随着电子制造工艺的发展,可编程逻辑器件(Programmable Logic Device,PLD)得到了前所未有的发展,集成度也从数百门的中小规模发展到数百万门的大规模和超大规模,应用也从原有的常规组合逻辑和时序逻辑发展到目前的系统级应用,例如,可以使用 PLD 设计一个较为复杂的 CPU。PLD 如同一张白纸或一堆积木,工程师可以通过传统的原理图输入法,或是硬件描述语言自由地设计一个数字系统。通过软件仿真,可以事先验证设计的正确性,在设计完成之后,可以通过重新编程使 PLD 执行不同的功能,而不需要重新布线,使用 PLD 来开发数字电路,可以大大缩短设计时间,减少 PCB 面积,提高系统的可靠性。PLD 主要包括可编程逻辑阵列(Programmable Logic Arrays,PLA),可编程阵列逻辑(Programmable Arrays Logic,PAL),复杂可编程逻辑器件(Complex Programmable Logic Devices,CPLD)和现场可编程门逻辑阵列(Field Programmable Gate Arrays,FPGA)等。

3.5.1　只读存储器

1. 掩模式 ROM

掩模式 ROM 所存储的数据是在 ROM 芯片制造过程中固化到芯片内部的，芯片出厂后是无法更改的。这种芯片具有结构简单，价格低廉和可靠性高等特点，适合批量生产。生产厂家可以根据用户要求生产专用的掩模式 ROM。

掩模式 ROM 一般包括地址译码矩阵，存储单元矩阵和输出缓冲区三个组成部分。

图 3.20 是二极管只读 ROM 的电路。该 ROM 有两个输入行和 4 个输出行。两个输入行对应 2^2 个输入模式，每个输入模式对应一条输出信号，分别为 W_3，W_2，W_1 和 W_0，这 4 路输出信号线称为字线。4 条字线中每次只会有一条被激活，这时输出线 F_3，F_2，F_1 和 F_0 上就会输出一组相应的 4 位编码，通常把这组编码称为“字”，字的字长等于输出行的条数，数据输出线也被称为位线。该存储器总共能输出 4 个字，也就是说，该 ROM 的存储容量总共为 $2^2 \times 4$b。位线输出端的缓冲器用来提高存储器的带负载能力和实现电平转换功能，便于直接与系统的数据总线相连。

当读出数据时，只要给定一个输入模式，即给出存储单元的地址时，相应的存储单元的数据就会出现在输出线上。如图 3.20 中若输入代码 $A_1A_0=11$ 时，字线 W_3 被激活，相应的位线 F_3，F_2 和 F_1 会导通，F_3，F_2 和 F_1 会输出 1，而 F_0 会输出 0。这表示地址单元 11 存储的数据为 1110。同样可以得到其他地址单元的存储数据。

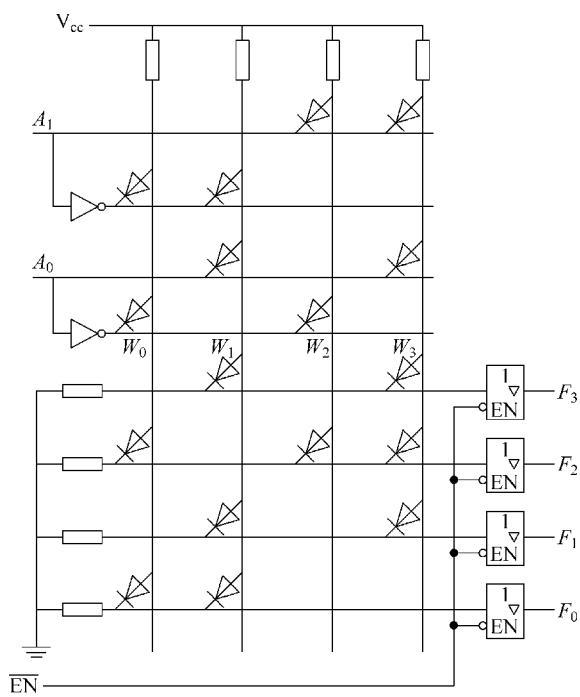


图 3.20　二极管只读 ROM 的电路

进一步分析可知，字线和输入行之间的关系有 $W_3=A_1'A_0'$，$W_2=A_1'A_0$，$W_1=A_1A_0'$，$W_0=A_1A_0$。位线与字线之间的关系有 $F_3=W_1+W_3$，$F_2=W_0+W_2+W_3$，$F_1=W_1+W_3$，$F_0=W_0+W_1'$。可以进一步将图 3.20 的电路化简，化简结果如图 3.21 所示。只读存储器的基本部分可以分为与门阵列和或门阵列，与门阵列实现对地址的译码过程，产生变量的全部最小项，而或门阵列完成最小项之间的或运算，从理论上来讲，利用 ROM 可以实现任何组合逻辑函数。

2. 可编程只读存储器

固定 ROM 出厂之后就不能再进行改动，如果用户需要根据自己的需要改动 ROM 中的内容，就需要 PROM。PROM 的整体结构与固定 ROM 相似，也是由地址译码器、存储矩阵和输出电路组成。所不同的是 PROM 在出厂前，在存储矩阵的所有交叉点都制作了存储单元，即相当于存储器的每个单元都存储 1。只是存储单元的结构与固定 ROM 的结构不同。

图 3.22 是一个熔丝型 PROM 的一个存储单元的原理图。其中熔丝使用低熔点合金丝或多晶硅导线制成，只要将该熔丝熔断，该单元就能存储 0。所以用户可以通过熔断相应位置的熔丝来对 PROM 进行编程，但是熔丝熔断之后不可还原，所以 PROM 只能写入一次。

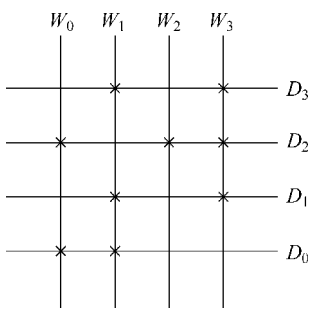


图 3.21 存储矩阵的或门阵列表示形式

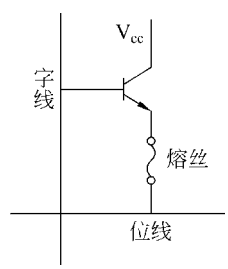


图 3.22 熔丝型 PROM 存储单元的原理图

3. 可擦除可编程只读存储器

可擦除可编程 ROM 可以分为两种，一种为 EPROM，一种为 E^2PROM。EPROM 是光可擦除 PROM，使用紫外线照射的方式来擦除 EPROM 中的数据，使其全部恢复到全 1 状态，然后使用专用的编程器进行重新编程，而 E^2PROM 则是使用电的方式进行擦除，其擦除时间比 EPROM 要短得多，使用起来更加方便，但容量小和速度慢是它的最大缺点。

4. 快闪存储器

Flash Memory 是近年来应用广泛的一种新型存储器，可归属于 E^2PROM。其特点是在不加电的情况下能长期保持存储的信息。它既有 ROM 的特点，又有高速的存取性能，擦除改写比较方便，功耗也比较低。一些新型的主板上也使用 Flash Memory 技术，使 BIOS 升级变得更为方便。

目前主要有基于 NOR 技术和 NAND 技术的 Flash Memory。NOR 技术闪速存储器是

最早出现的 Flash Memory，目前仍是多数供应商支持的技术架构，它源于传统的 EPROM 器件。它的主要特点如下。

(1) 与其他 Flash Memory 技术相比，具有可靠性高、随机读取速度快的优势。

(2) 在擦除和编程操作较少而直接执行代码的场合，尤其是代码(指令)存储的应用中广泛使用。

(3) 由于 NOR 技术 Flash Memory 的擦除和编程速度较慢，而块尺寸又较大，因此擦除和编程操作所花费的时间很长，在纯数据存储和文件存储的应用中，NOR 技术显得力不从心。

(4) 支持芯片内执行(eXecute In Place，XIP)，存储在 Flash Memory 芯片内的程序不用复制到 RAM 就可以直接运行。

而 NAND 技术 Flash Memory 具有以下特点。

(1) 以页为单位进行读和编程操作，一页为 256B 或 512B。以块为单位进行擦除操作，一块为 4KB、8KB 或 16KB。具有块编程和块擦除的功能，其块擦除时间是 2ms。而 NOR 技术的块擦除时间达到几百毫秒。

(2) 数据、地址采用同一总线，实现串行读取。随机读取速度慢且不能按字节随机编程。

(3) 芯片尺寸小，引脚少，是位成本(bit cost)最低的固态存储器，突破了每兆字节 1 元的价格限制。

(4) 芯片包含失效块，其数目最大可达到 3～35 块(取决于存储器密度)。失效块不会影响有效块的性能，但设计者需要将失效块在地址映射表中屏蔽起来。

基于 NAND 的存储器可以取代硬盘或其他块设备。

3.5.2　随机存取存储器

RAM 可以分为 SRAM 和 DRAM。SRAM 内部使用的是类似 *D* 触发器的电路，只要不对它断电，存放在里面的数据就可以长久保存，而且它的速度很快，所以 SRAM 广泛应用于 Cache 中。而 DRAM 则使用由晶体管和小电容组成的存储单元构成的阵列来存放数据，通过电容的充电和放电来存放 0 和 1。由于电容的放电现象，需要外部电路不断地进行刷新，所以 DRAM 比 SRAM 的外部接口要复杂，但是它的容量比较大，存储密度比较高，所以计算机的主存都是使用 DRAM 构成的。DRAM 的缺点就是速度要比 SRAM 慢很多。下面来介绍两者的原理。

图 3.23 给出存储一位的单个位元的典型 DRAM 结构。当数据读出或写入位元时，启动地址线。若一电压施加到地址线上，作为开关的晶体管被导通；若无电压施加到地址线上，晶体管开路(无电流通过)。

对于写操作，一个电压信号施加到位线：高电压代表 1，低电压代表 0。另一信号则施加到地址线，允许电荷传输到电容器。

对于读操作，当地址线被选中时，晶体管导通，存储在电容上的电荷被送出到位线和读出放大器。读出放大器将此电容电压与一参考值相比较来确定位元保存的是逻辑 1 还是逻辑 0。位元的读出放掉了电容上的电荷，必须重新存储才算完成本次读操作。

虽然 DRAM 位元能存储单一位值(0 或 1)，但它本质上是一个模拟设备。因为电容能存储一定范围内的任何电荷值。故必须使用一个门槛值来确定电荷位代表的是 1 还是 0。

相对而言，静态 RAM(Static RAM，SRAM)是一个数字设备，它使用与处理器相同的

逻辑元件。静态 RAM 采用传统的触发器、逻辑门配置来存储二进制值。只要电源不断，SRAM 将一直保持它所存储的数据。

图 3.24 给出单个位元的典型 SRAM 结构。4 个晶体管交叉连接组成一个有稳定逻辑状态的排列。逻辑状态 1 下，C1 点是高电平，C2 点是低电平，晶体管 T1 和 T4 断，T2 和 T3 通。逻辑状态 0 下，C1 点是低电平，C2 点是高电平，T1 和 T4 通，T2 和 T3 断。只要直流电源一直供电，两个状态都是稳定的；不同于 DRAM，这里不需要刷新。

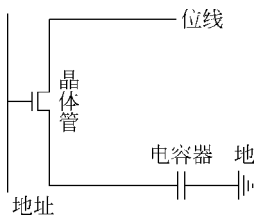


图 3.23 DRAM 位元

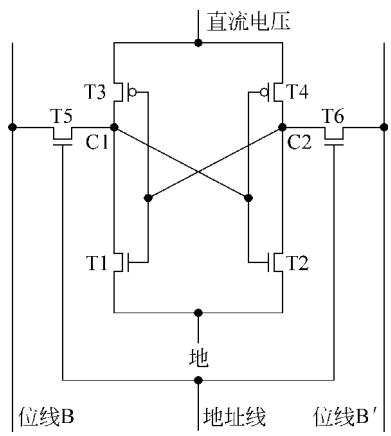


图 3.24 SRAM 位元

如同 DRAM 中一样，地址线用来控制开关的通断。这里，地址线控制两个晶体管（T5 和 T6），当信号施加到地址线上，两个晶体管导通允许读/写操作。写操作时，位值施加到 B 线，相反的值施加到 B′线，这强迫 4 个晶体管（T1、T2、T3、T4）进入一个相应的稳态。读操作时，位值由 B 线读出。

3.5.3 可编程逻辑器件

传统的 ROM 和 RAM 都属于纯硬件的范围，而现在硬件和软件之间的界限越来越模糊，硬件和软件的结合也越来越紧密了。随着系统集成度的不断提高，硬件设计人员也从更高层来思考硬件系统的设计。将软件思想引入到硬件设计的理念中，就产生了可编程逻辑器件。编程的意思是将事先设计好的数据通过某种方式装入 PLD 的可编程存储器中，以确定 PLD 各模块的逻辑功能以及各模块之间以及模块与 I/O 之间的连接方式，数据写入在编程过程中完成，数据写入后 PLD 就实现用户设定的功能，如果需要更改功能，用户需要重新进行编程。

本节以现在最流行的现场可编程逻辑器件 FPGA 为例来进行介绍，其他的 PLD 请参考相关书籍。相对于其他 PLD，FPGA 具有更高的密度、更快的工作速度和更大的编程灵活性，主要的生产厂家有 Xilinx 和 Altera 公司。

FPGA 主要有以下 4 个组成部分。

（1）可配置逻辑块（Configurable Logic Block，CLB）。

（2）可编程输入输出块（Input/Output Block，IOB）。

（3）可编程内部连线（Programmable Interconnect，PI）。

(4) 重构逻辑的程序存储器。

图 3.25 示出了它们之间的关系，下面对其中的每个部分进行展开。

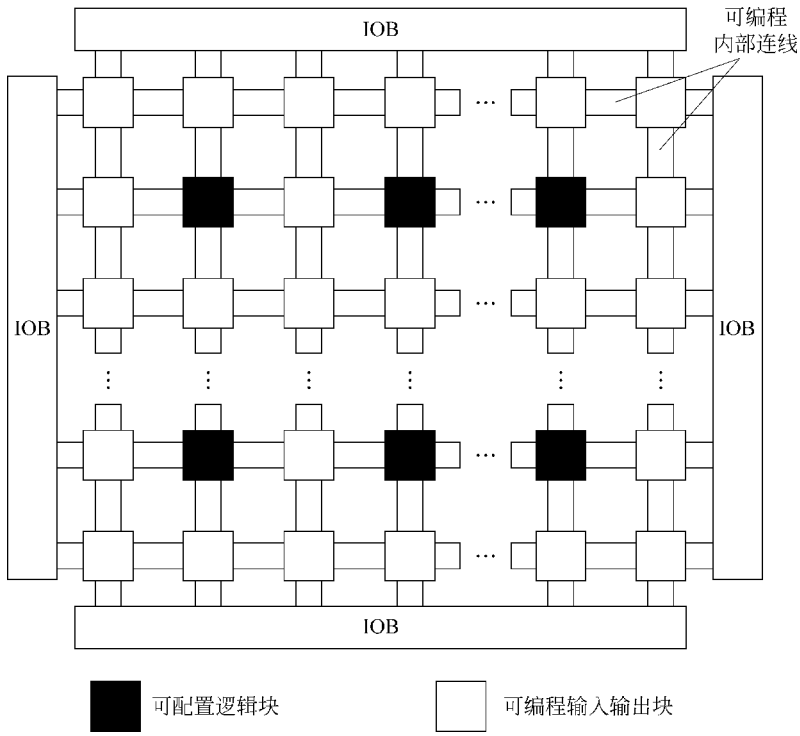


图 3.25　典型的 FPGA 的内部结构

1. 可配置逻辑块

CLB 实现了 FPGA 中的大部分逻辑，图 3.26 中示出了一个简单的 CLB 的结构。其内部由两个 4 输入的逻辑函数发生器、两个触发器和多个可编程数据选择器构成。逻辑函数发生器通过查表(Lookup Tables，LUT)实现组合逻辑，每个 LUT 就是一个 16×1 比特的 PROM，PROM 里存放待产生的函数的真值表信息。当输入 G_4，G_3，G_2 和 G_1 或者 F_4，F_3，F_2 和 F_1 时，LUT 就会从 PROM 中找出与输入信号对应的模式，然后选择相应的输出送到 G 或者 F。H 数据选择器根据 H_1 的输入选择 F 或者 G 作为其输出。X，Y 输出和触发器的输入都是根据可编程数据选择器决定的，数据选择器里的信息在 FPGA 进行配置的时候产生。

2. 可编程输入输出块

输入输出块是为芯片外部引脚或内部逻辑模块提供接口的电路，通过对 IOB 的编程，可以将引脚定义为输入、输出和双向 I/O 三种方式。

3. 可编程内部连线

内部连线是带有可编程节点和开关矩阵的内部金属线段，这些线段为内部各个 CLB、

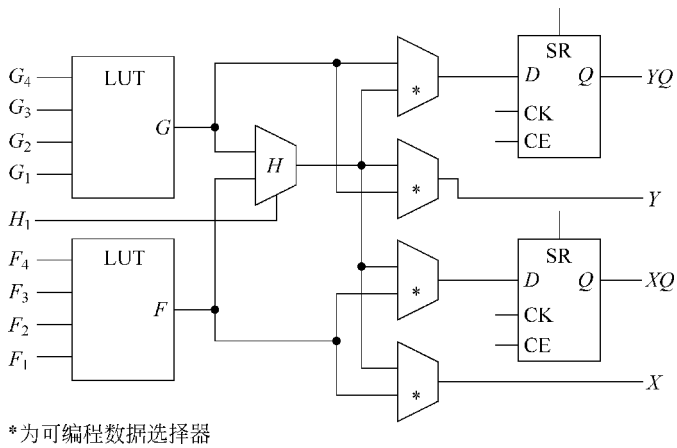


图 3.26 CLB 的内部结构

IOB 提供了连线资源，用来实现内部 CLB 之间、IOB 之间，以及 CLB 和 IOB 之间的相互连线，正是有了互联资源才可以实现极其复杂的逻辑电路。FPGA 中常用三种布线：单长线，双长线和长线，根据布线的距离和驱动能力进行选择。

4. 重构逻辑的程序存储器

FPGA 工作时，首先要将用户所需实现的逻辑以某种程序形式从片外读至 FPGA 重构逻辑的程序存储器内，该存储器的存储单元输出直接去控制指定的 CLB、IOB 等单元，从而使器件具有确定的功能。常将这一过程称为配置。重构逻辑的程序存储器是整个 FPGA 配置的关键部分，CLB 和 IOB 中的数据是通过存放在重构逻辑存储器中的数据来确定的。数据由开发系统产生并直接装至重构逻辑存储器内。开发系统能自动将用户逻辑划分为 CLB 和 IOB 的集合；自动确定每个宏单元块所要承担的逻辑功能；自动在逻辑块间进行互连设计；最终自动生成。这样，FPGA 就可以多次修改逻辑，且更为经济和快捷，是设计各种数字系统的一种比较好的选择。

图 3.27 是 20 万门的 FPGA 的实验开发板，可以通过高级编程软件进行系统设计和仿真，最后将数据下载配置到 FPGA 中，FPGA 则能按照设计给定的模式进行工作。

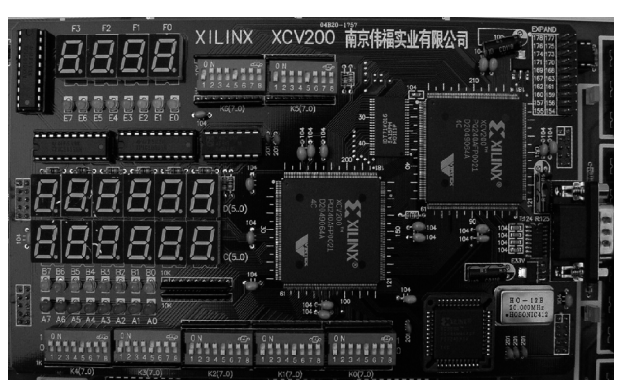


图 3.27 FPGA 实验开发板

3.6　VHDL 硬件描述语言

随着电子元件集成度的提高，对集成电路的设计技术要求越来越高。专注于元件级的系统设计技术已经不能满足大规模和超大规模系统的要求，硬件描述语言（Hardware Description Language，HDL）的重要性也显得更加重要。现在程序员通常使用 C 语言来编写程序，然后编译成机器语言在计算机上运行。硬件描述语言类似于 C 语言，允许在更高级别上进行系统设计和调试，调试无误后合成为一系列组件和互连，之后输出到 CPLD 或者 FPGA 硬件，这种合成可以通过相应的辅助设计工具来完成。这种设计方法能够极大提高数字系统的设计速度，减少设计周期，同时也能提高设计的正确性。

HDL 设计数字系统一般采用自上而下（Top Down）的分层设计方法。所谓自上而下的设计方法，就是从系统总体出发，自上而下地逐步将设计内容细化，最后完成系统硬件的整体设计。层次自上而下分别如下。

（1）第一层行为描述，就是对整个系统的数学模型进行描述。

（2）第二层 RTL 方式描述，即寄存器传输描述，也称为数据流描述。

（3）第三层逻辑综合，就是利用逻辑综合工具，将 RTL 方式描述的程序转换成用基本元件表示的文件（门级网表）。

HDL 到硬件实现通常需要先进行仿真设计和测试，然后下载到 FPGA 硬件之中，过程如图 3.28 所示。

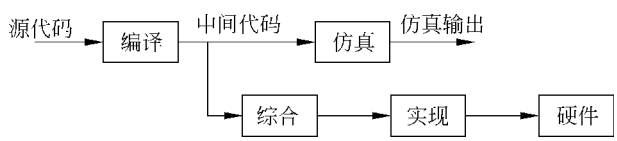


图 3.28　HDL 到硬件实现的各个阶段

目前有两种使用最广泛的 HDL，分别是 VHDL 和 Verilog HDL。

本节主要介绍 VHDL。VHDL 硬件描述语言的全称是 Very High Speed Integrated Circuit Hardware Description Language，即超高速集成电路硬件描述语言。由美国政府牵头，VHDL 工作小组于 1981 年 6 月成立，提出了一个满足电子设计各种要求的能够作为工业标准的 HDL。1983 年第三季度，由 IBM、TI 等公司签约，组成开发小组，工作任务是提出语言版本和开发软件环境。1986 年，IEEE 标准化组织开始工作，讨论 VHDL 标准，历时一年有余，于 1987 年，12 月通过标准审查，并宣布实施，即 IEEE STD 1076—1987。1993 年，VHDL 重新修订，形成新的标准即 IEEE STD 1076—1993。从此以后，美国国防部实施新的技术标准，要求电子系统开发商的合同文件一律采用 VHDL 文档。即第一个官方 VHDL 标准得到推广、实施和普及。VHDL 适用范围很广，可以设计由几个门电路构成的系统，也可以设计由多个复杂系统互联之后的更复杂的系统。

3.6.1　VHDL 语法

一个 VHDL 程序包含实体（Entity）、结构体（Architecture）、配置（Configuration）、包集合（Package）、库（Library）5 部分。实体用于描述所设计的系统的外部接口信号；构造体用

于描述系统内部的结构和行为；包集合存放各设计模块都能共享的数据类型、常数和子程序等；配置用于从库中选取所需单元来组成系统设计的不同版本；库中存放已经编译好的实体、构造体、包集合和配置。

1. VHDL 的基本单元

VHDL 的实体(Entity)是 VHDL 设计的基本单元。一个实体，简单的可以是一个与门，复杂点的可以是一个系统。但是，不管是简单的数字电路，还是复杂的数字电路，其基本构成是一致的，都由实体说明(Entity Declaration)和构造体(Architecture Body)两部分构成。如前所述，实体说明部分规定了设计单元的输入输出接口信号或引脚，而构造体部分定义了设计单元的具体构造和行为。例 3 示出了二选一电路的 VHDL 描述。由例 3 可以看出，实体说明是二选一器件外部引脚的定义；而构造体则描述了二选一器件的逻辑电路和逻辑关系。

【例 3】 二选一电路的 VHDL 描述

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
-- ENTITY DECLARATION
ENTITY mux IS
PORT (d0,d1,sel: IN STD_LOGIC;
         q: OUT STD_LOGIC);
END mux;
-- ARCHITECTURE BODY
ARCHITECTURE connect OF mux IS
SIGNAL tmp: STD_LOGIC;
BEGIN
    PROCESS(d0,d1,sel)
    BEGIN
    tmp <= (d0 AND sel) OR (d1 AND (NOT sel));
    q <= tmp;
    END PROCESS;
END connect;
```

1) 实体说明

任何一个基本设计单元的实体说明都具有如下的结构：

```
ENTITY 实体名 IS
  [类属参数说明];
  [端口说明];
END 实体名;
```

(1) 类属参数说明

类属参数说明必须放在端口说明之前，用于指定参数，具体书写范例如下：

```
GENERIC (m: TIME:= 1 ns)
```

该语句指定了 m 的值为 1，时间单位为 ns。

(2) 端口说明

端口说明是对基本设计实体与外部接口的描述，也可以说是对外部引脚信号的名称、数据类型和输入、输出方向的描述。其书写格式如下：

```
PORT(端口名{,端口名}: 方向　数据类型名;
    …
    端口名{,端口名}: 方向　数据类型名);
```

① 端口名。

端口名是赋予每个外部引脚的名称，通常用一个或几个英文字母，或者用英文字母加数字命名。例如例 3 中的外部引脚为 d0，d1，sel，q。

② 端口方向。

端口方向用来定义外部引脚的信号方向是输入还是输出。表示方向的说明符及其含义如表 3.8 所示。

表 3.8　端口方向说明

方 向 定 义	含　义
IN	输入
OUT	输出(构造体内部不能使用)
INOUT	双向
BUFFER	输出(构造体内部可再使用)

其中，BUFFER 和 OUT 的区别主要在于：如果构造体内部要使用该信号，那么锁存器的输出端必须说明为“BUFFER”，而不能用“OUT”说明。而且当一个构造体用“BUFFER”说明输出端口时，与其连接的另一个构造体的端口也要用“BUFFER”说明。

③ 数据类型。

在 VHDL 中有 10 种数据类型，但是在逻辑电路设计中只用到两种：STD_LOGIC 和 STD_LOGIC_VECTOR。当端口被说明为 STD_LOGIC 数据类型时，该端口的信号取值只可能是“1”或“0”。当端口被说明为 STD_LOGIC_VECTOR 数据类型时，该端口的取值可能是一组二进制位的值。

2) 构造体

构造体是一个基本设计单元，它具体地指明了该基本设计单元的行为、元件及内部的连接关系，也就是说它定义了设计单元具体的功能。构造体对其基本设计单元的逻辑关系可以用三种方式进行描述，即行为描述(基本设计单元的数学模型描述)、寄存器传输描述(数据流描述)和结构描述(逻辑元件连接描述)。不同的描述方式，只体现在描述语句上，而构造体的结构是完全一样的。由于构造体是对实体功能的具体描述，因此它一定要跟在实体的后面。

一个构造体的具体结构描述如下：

```
ARCHITECTURE 构造体名 OF 实体名 IS
 [定义语句] 内部信号,常数,数据类型,函数等的定义;
 BEGIN
  [并行处理语句];
 END 构造体名;
```

(1) 构造体名称的命名

构造体的名称是对本构造体的命名，它是该构造体的唯一名称。OF 后面紧跟的实体名表明了该构造体所对应的是哪一个实体，用 IS 来结束构造体的命名。

(2) 定义语句

定义语句位于 ARCHITECTURE 和 BEGIN 之间,用于对构造体内部所使用的信号、常数、数据类型和函数进行定义。例如:

```
ARCHITECTURE behav OF mux IS
 SIGNAL  nesl: STD_LOGIC;
    …
 BEGIN
    …
 END behav;
```

信号定义和端口说明的语句一样,应有信号名和数据类型的说明。因它是内部连接用的信号,故没有也不需有方向的说明。

(3) 并行处理语句

并行处理语句处于语句 BEGIN 和 END 之间,这些语句具体地描述了构造体的行为及其连接关系。在构造体中的语句都是可以并行执行的,也就是说,语句的执行不以书写的语句顺序为执行顺序。

2. VHDL 构造体的子结构

在规模较大的电路设计中,全部电路都用唯一的一个模块来描述是不方便的。为此,电路设计者总希望将整个电路分成若干个相对比较独立的模块来进行电路的描述。这样,一个构造体可以用若干个子结构,即相对比较独立的模块来构成。VHDL 可以有以下三种形式的子结构描述语句。

(1) BLOCK 语句结构

(2) PROCESS 语句结构

(3) SUBPROGRAM 结构

其中,PROCESS 语句结构最常用。下面对上述三种子结构逐一说明。

1) BLOCK 语句结构描述

人们在用计算机电路辅助设计工具输入电路原理图时,往往将一个大规模的电路原理图分割成多张子原理图,进行输入和存档。同样在 VHDL 中也不例外,电路的构造体对应整个电路原理图,而构造体可以由多个 BLOCK 块构成,每一个 BLOCK 块对应一张子原理图。这样电路原理图的分割关系和 VHDL 程序中用 BLOCK 分割构造体的关系是一一对应的。

在用其他高级语言编程时,总希望程序模块小一点,以利于编程和查错,也利于实现积木化结构。同理,在 VHDL 中采用 BLOCK 语言对编程、查错、仿真及再利用都会带来莫大的好处。

采用 BLOCK 语句描述局部电路的书写格式如下所示:

```
块结构名: BLOCK
  BEGIN
    …
  END  BLOCK 块结构名;
```

如果采用 BLOCK 语句来描述二选一电路,那么用 VHDL 就可以书写为:

【例 4】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
-- ENTITY DECLARATION
ENTITY mux IS
PORT (d0,d1,sel: IN STD_LOGIC;
              q: OUT STD_LOGIC);
END mux;

-- ARCHITECTURE BODY
ARCHITECTURE connect OF mux IS
SIGNAL tmp: STD_LOGIC;
BEGIN
      blk: BLOCK
      BEGIN
      tmp <= (d0 AND sel) OR (d1 AND (NOT sel));
      q <= tmp;
      END BLOCK blk;
END connect;
```

上述程序的构造体中只有一个 BLOCK 块，如果电路复杂时可以由几个 BLOCK 块组成。

2）PROCESS 语句结构描述

采用 PROCESS 语句描述电路结构的书写格式如下：

```
[进程名]  PROCESS(敏感信号 1,敏感信号 2,…)
            BEGIN
                …
          END  PROCESS;
```

进程名可以有，也可以省略。PROCESS 语句从 PROCESS 开始至 END PROCESS 结束。执行 PROCESS 语句时，通常带有若干个敏感信号量。这些信号量将在 PROCESS 结构的语句中被使用。用 PROCESS 语句结构描述的程序如下所示。

【例 5】

```
ENTITY  mux  IS
 PORT(d0,d1,sel: IN  STD_LOGIC;
              q: OUT  STD_LOGIC);
 END mux;
 ARCHITECTURE  connect  OF  mux  IS
 BEGIN
   PROCESS(d0,d1,sel)
   VARIABLE  tmp1,tmp2,tmp3: STD_LOGIC;
      BEGIN
      tmp1:= d0 AND sel;
      tmp2:= dl AND(NOT sel);
      tmp3:= tmp1 OR tmp2;
      q <= tmp3;
      END PROCESS;
END connect;
```

程序中，tmp1、tmp2 和 tmp3 是变量，变量只在进程内部定义，详见后面阐述。

3) SUBPROGRAM 语句结构描述

所谓 SUBPROGRAM(过程和函数)就是在主程序调用它以后能将处理结果返回主程序的程序模块，其含义和其他高级语言中的子程序概念相当。它可以反复调用，使用非常方便。子程序在调用时首先要进行初始化，执行结束后子程序就终止。再调用时要再进行初始化。因此子程序内部的值不能保持，子程序返回以后才能被再调用，它是一个非重入的程序。

在 VHDL 中 SUBPROGRAM 有以下两种类型。

(1) 过程语句

在 VHDL 中，过程语句的书写格式如下：

```
PROCEDURE 过程名(参数 1,参数 2,…) IS
    [定义语句];        (变量等定义)
    BEGIN
      [顺序处理语句];   (过程的语句)
    END 过程名;
```

在 PROCEDURE 结构中，参数可以是输入也可以是输出。也就是说，过程中的输入输出参数都应列在紧跟过程名的括号内。例如，在 VHDL 中，将位矢量转换为整数的程序可以由一个过程语句来实现。

```
PROCEDURE vector_to_Int
              (z: IN STD_LOGIC_VECTOR;
              x_flag: OUT BOOLEAN;
              q: INOUT INTEGER) IS
 BEGIN
 q:= 0;
 x_flag:= FALSE;
 FOR i IN z' RANGE LOOP
    q:= q * 2;
 IF(z(i) = 1) THEN
    q:= q + 1;
 ELSIF(z(i)/ = 0) THEN
    x_flag:= TRUE;
 END IF;
 END  LOOP;
END vector_to_int;
```

该过程调用后，如果 x_flag＝TRUE，则说明转换失败，不能得到正确的转换整数值。

在上例中，z 是输入，x_flag 是输出，q 为输入输出。在没有特别指定的情况下，“IN”作为常数；而“OUT”和“INOUT”则看做“变量”进行拷贝。当过程的语句执行结束以后，在过程内所传递的输出和输入输出参数值，将拷贝到调用者的信号或变量中。此时输入输出参数如没有特别指定则按变量对待，将值传递给变量。如果调用者需要将输出和输入输出作为信号使用，则在过程参数定义时要指明是信号。例如

```
PROCEDURE  shift(
                din: IN  STD_LOGIC_VECTOR;
                SIGNAL dout: OUT STD_LOGIC_VECTOR);
```

```
END shift;
```

前面已经提到，PROCESS 结构中的语句是顺序执行的，那么在过程结构中的语句也是顺序执行的。调用者在调用过程前应先将初始值传递给过程的输入参数。然后过程语句启动，按顺序自上至下执行过程结构中的语句，执行结束，将输出值复制到调用者的“OUT”和“INOUT”所定义的变量或信号中。

(2) 函数语句

在 VHDL 中，函数语句的书写格式如下：

```
FUNCTION  函数名(参数 1,参数 2,…)
        RETURN  数据类型名 IS
  [定义语句];
  BEGIN
    [顺序处理语句];
    RETURN [返回变量名];
END[函数名];
```

在 VHDL 中，FUNCTION 语句中括号内的所有参数都是输入参数或称输入信号。因此在括号内指定端口方向的“IN”可以省略。FUNCTION 的输入值由调用者复制到输入参数中，如果没有特别指定，在 FUNCTION 语句中按常数处理。

通常各种功能的 FUNCTION 语句的程序都被集中在包集合(Package)中。

上面详细叙述了过程、函数的结构和使用方法。为了能重复使用这些过程和函数，这些程序通常组织在包集合、库中。它们与包集合和库具有这样的关系：即多个过程和函数汇集在一起构成包集合(Package)，而几个包集合汇集在一起就形成一个库(Library)。有关包集合和库的详细内容将在下面介绍。但是，需要指出的是，不同公司发布的包集合和库的使用方法各不相同。

3. 包集合和库

除了实体和构造体之外，包集合和库是在 VHDL 中可以各自独立进行编译的设计单元。

1) 包集合

数据类型、常量与子程序可以在实体说明部分和结构体部分加以说明，而且实体说明部分所定义的类型、常量及子程序在相应的结构体中是可见的(可以被使用)。但是，在一个实体的说明部分与结构体部分中定义的数据类型、常量及子程序对于其他实体的说明部分与结构体部分是不可见的。为了使一组类型说明、常量说明和子程序说明对多个设计实体都成为可见的，VHDL 提供了包结构。包由包头和包体两部分组成。包头以保留字 Package 开头，包体则以 Package Body 识别。实际上，标准库中存在的也是一些程序包。

程序包的语法是：

```
Package <包名> is
   [外部子程序说明]
   [外部常量说明]
   [外部元件模板]
   [外部类型说明]
```

```
   [属性说明]
   [属性指定]
end [<包名>]
Package  body <包名> is
   [外部子程序体]
   [内部子程序说明]
   [内部子程序体]
   [内部常量说明]
   [内部类型说明]
end[<包名>]
```

【例 6】

```
PACKAGE  Logic  IS
TYPE  three_level_logic IS('0','1','Z');
CONSTANT unknown_value: three_level_logic: = '0';
FUNCTION invert(input: three_level_logic)
RETURN  three_level_logic;
END  logic;

PACKAGE BODY logic IS
FUNCTION  invert(input: three_level_logic)
RETURN three_level_logic IS
    BEGIN
      CASE input IS
        WHEN '0' = > RETURN '1';
        WHEN '1' = > RETURN '0';
        WHEN 'Z' = > RETURN 'Z';
    END CASE;
END invert;
END logic;
```

2) 库

库(Library)是经编译后的数据的集合,存放包集合定义、实体定义以及构造体定义。库的功能类似于 MS-DOS 操作系统中的目录,库中存放设计的数据。在 VHDL 中,库的说明总是放在设计单元的最前面,如:

```
Library 库名;
```

这样,在设计单元内的语句就可以使用库中的数据。因此,库的好处就在于使设计者可以共享已经编译过的设计结果。在 VHDL 中可以存在多个不同的库,但库与库之间是独立的,不能互相嵌套。

(1) 库的种类

VHDL 中的库分为设计库和资源库两类。

① 设计库。

所谓设计库,即由 VHDL 标准规定的,对所有项目是默认可见的,不需要用 Library、USE 子句声明的库。STD 库和 WORK 库是设计库。

STD:STD 库中包含预定义的程序包 STANDARD 和 TEXTIO,这两个程序包是使用

VHDL 时必须用到的工具。

WORK 库：是 VHDL 的工作库，用户在设计项目中已经设计好的，正在校验、未仿真的中间件等都放在工作库中。实际上 WORK 是一个临时的仓库，用来存放成品或半成品。

② 资源库。

除了 STD 和 WORK 库以外，其他的库都为资源库。资源库用来存放常规元件和标准模块，使用时要预先声明。在有些库中，存放的元件、函数都是 IEEE 标准化组织认可的，称为 IEEE 库。VHDL 工具厂商与 EDA 工具专业公司都有自己的资源库。

(2) 库的使用

所有被编译的元件都存在设计库(Library)中，程序包通常也存在设计库中，不同库中程序包的个数也不相同，一个程序包中可以包含若干个子程序，子程序中又包含函数、过程、元件等基础的设计单元。尽管每个文件中可以存放多个单元，但为了更便于组织和管理，通常在一个文件中只存放一个实体说明和结构体对。用 VHDL 写程序时必须首先指定库和包的名字。在每个设计项目的开头声明选用的库名，用 USE 语句声明选中的逻辑单元。一个设计单元可以访问多个设计库，有些库被 IEEE 标准化组织认可，被称为 IEEE 库。IEEE 库存放了 IEEE 标准 1076 中的标准设计单元。

库语句和 USE 语句的语法形式是：

```
Library <库名>
 Use <库名>.<包名>.all;
```

例如：

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

3.6.2　数据类型、运算操作符

VHDL 像其他高级语言一样，具有多种数据类型。对大多数数据类型的定义是一致的。

1. VHDL 的对象及其分类

在 VHDL 中凡是可以赋予一个值的对象就称为 object。object 主要包括以下三种：信号、变量、常数(signal、variable、constant)。在电路设计中，这三类 object 通常都具有一定的物理含义。例如，信号对应的代表物理设计中的某一条硬件连接线；常数对应的代表数字电路中的电源和地等；变量对应关系不太直接，通常只代表暂存某些值的载体。三类 object 的含义和说明场合如表 3.9 所示。

表 3.9　三类 object 含义和说明场合

类　别	含　义	说 明 场 合
信号	全局量	ARCHITECTURE、PACKAGE、ENTITY
变量	局部量	PROCESS、FUNCTION、PROCEDURE
常数	全局量	以上两种场合均可

1）常数

常数是一个固定的值。所谓常数说明就是对某一常数名赋予一个固定的值。通常赋值在程序开始前进行，该值的数据类型则在说明语句中指明。常数说明的一般格式如下：

```
CONSTANT  常数名: 数据类型:= 表达式;
```

例如：

```
CONSTANT VCC: REAL:= 5.0;
CONSTANT DELAY: TIME:= 100 ns;
```

常数一旦被赋值就不能再改变。上面 VCC 被赋值为 5.0 V，那么在所有的 VHDL 程序中 VCC 的值就固定为 5.0 V，不像后面所提到的信号和变量那样，可以任意代入不同的数值。另外，常数所赋的值应和定义的数据类型一致。

2）变量

变量只能在进程语句、函数语句和过程语句结构中使用，它是一个局部量。在仿真过程中，它不像信号那样，到了规定的仿真时间才进行赋值，变量的赋值是立即生效的。变量说明语句的格式为：

```
VARIABLE  变量名: 数据类型约束条件;
```

例如：

```
VARIABLE  x,y: INTEGER;
VARIABLE  count: INTEGER;
```

变量在赋值时不能产生附加延时。例如，tmp1，tmp2，tmp3 都是变量，那么下式产生延时的方式是不合法的：

```
tmp3:=  tmpl + tmp2  AFTER 10 us;
```

3）信号

信号是电路内部硬件连接的抽象。它除了没有数据流动方向说明以外，其他性质几乎和前面所述的“端口”概念一致。信号通常在构造体、包集合和实体中说明。信号说明语句格式为：

```
SIGNAL  信号名: 数据类型;
```

例如：

```
SIGNAL  sysclk: STD_LOGIC;
SIGNAL  ground: STD_LOGIC;
```

在程序中，信号值的代入采用“<=”代入符，而不是像变量赋值时用“:=”符。而且信号代入时可以附加延时。例如，s1 和 s2 都是信号，且 s2 的值经 10 ns 延时以后才被代入 s1。此时信号传送语句可书写为：

```
s1 <=  s2  AFTER  10 ns;
```

信号是一个全局量，它可以用来进行进程之间的通信。

2. VHDL 的数据类型

如前所述，在 VHDL 中信号、变量、常数都要指定数据类型。为此，VHDL 提供了多种标准的数据类型。另外，为使用户设计方便，还可以由用户自定义数据类型。这样使语言的描述能力及自由度更进一步提高，从而为系统高层次的仿真提供了必要手段。

同时，VHDL 数据类型的定义相当严格，不同类型之间的数据不能直接代入，而且，即使数据类型相同，而位长不同时也不能直接代入。这样，为了熟练地使用 VHDL 编写程序，必须准确理解各种数据类型的定义。

1) 标准的数据类型

标准的数据类型共有 10 种，如表 3.10 所示。

表 3.10　标准数据类型

数 据 类 型	含　义
整数	32 位整数
实数	浮点数
位	逻辑“0”或“1”
位矢量	一组位数据
布尔量	逻辑“假”和“真”
字符	ASCII 字符
时间	时间单位 fs，ps，ns 等
错误等级	NOTE，WARNING，ERROR，FAILURE
自然数，正整数	整数的子集
字符串	字符矢量

(1) 整数

整数(integer)与数学中的整数的定义相同。在 VHDL 中，整数的表示范围为 −2 147 483 647～2 147 483 647，即从 $-(2^{31}-1)$ 到 $(2^{31}-1)$。

(2) 实数

实数(real)的定义值范围为 −1.0E+38 ～+1.0E+38。实数有正负数，书写时一定要有小数点。例如：

−1.0，+2.5，−1.0E38

(3) 位

在数字系统中，信号值通常用一个位(bit)来表示。位值的表示方法是，用字符 0 或者 1 表示。位与整数中的 1 和 0 不同，0 和 1 仅表示一个位的两种取值。另外，位不能用来描述三态信号。

(4) 位矢量

位矢量(bit_vector)是用双引号括起来的一组位数据。例如：“001100”，X“00BB”。这里，位矢量最前面的 X 表示是十六进制。用位矢量数据表示总线状态最形象也最方便。

(5) 布尔量

一个布尔量(boolean)具有两种状态，“真”或者“假”。虽然布尔量也是二值枚举量，但它和位不同，布尔量没有数值的含义，也不能进行算术运算。它能进行关系运算。例如，它

可以在 IF 语句中被测试，测试结果产生一个布尔量真或者假。

(6) 字符

字符(character)也是一种数据类型，所定义的字符量通常用单引号括起来，如'A'。字符量中的字符可以是 a～z 中的任一个字母，0～9 中的任一个数以及空白或者特殊字符，如 $，@，%等。包集合 STANDARD 中给出了预定义的 128 个 ASCII 码字符类型，不能打印的用标识符给出。

(7) 字符串

字符串(string)是由双引号括起来的一个字符序列，它也称为字符矢量或字符串数组。例如：“ Integer　range”字符串常用于程序的提示和说明。

(8) 时间

时间(time)是一个物理量数据。完整的时间量数据应包含整数和单位两部分，而且整数和单位之间至少应留一个空格的位置。例如，55 sec，2 min 等。在包集合 STANDARD 中给出了时间的预定义，其单位为 fs，ps，ns，μs，ms，sec，min，hr。

(9) 错误等级

错误等级类型数据用来表征系统的状态，它共有 4 种：NOTE(注意)，WARNING(警告)，ERROR(出错)，FAILURE(失败)。在系统仿真过程中可以用这 4 种状态来提示系统当前的工作情况。

2) 用户定义的数据类型

用户自定义的数据类型包括枚举类型、整数类型、实数类型、数组类型、记录类型、存取类型、文件类型等。对于自定义的数据类型的一般书写格式是：

```
TYPE 数据类型名{,数据类型名} IS 数据类型定义;
```

(1) 枚举类型(enumerated type)

格式：

```
TYPE  数据类型名 IS(元素 1,元素 2,…);
```

(2) 整数类型、实数类型

格式：

```
TYPE  数据类型名  IS  数据类型约束范围
```

(3) 数组类型(array type)

格式：

```
TYPE 数据类型名 IS ARRAY (INTEGER  0  TO  9) OF  STD_LOGIC;
```

(4) 记录类型(record type)

格式：

```
TYPE 数据类型 IS  RECORD
    元素名: 数据类型名;
    元素名: 数据类型名;
END  RECORD;
```

(5) 存取类型

定义：存取类型(access type)也称为寻址类型，用来在对象之间建立联系，或者给新对象分配或释放存储空间，与高级语言中使用的指针类似。在 VHDL'93 语言标准 IEEE std_1076 的程序包 TEXTIO 中定义了一个存取类型。

格式：

```
TYPE (存取类型名)line IS ACCESS string;
```

表示类型为 line 的变量，它的值是指向字符值的指针。只有变量才能定义为存取类型。

(6) 文件类型(files type)

格式：

```
TYPE 文件类型名 IS  FILE;
```

3. VHDL 的运算操作符

在 VHDL 中共有 4 类操作符，可以分别进行逻辑运算(logical)、关系运算(relational)、算术运算(arithmetic)和并置运算(concatenation)。需要注意的是，被操作符所操作的对象是操作数，且操作数的类型应该和操作符所要求的类型相一致。另外，运算操作符是有优先级的，例如逻辑运算符 NOT，在所有操作符中其优先级最高。表 3.11 列出了所有操作符的优先次序。

表 3.11　操作符的优先级

优先级顺序	运算操作符类型	操　作　符	功　　能
低	逻辑运算符	AND	逻辑与
		OR	逻辑或
		NAND	逻辑与非
		NOR	逻辑或非
		XOR	逻辑异或
	关系运算符	=	等号
		/=	不等号
		<	小于
		>	大于
		<=	不大于
		>=	不小于
	加、减、并置运算符	+	加
		−	减
		&	并置
	正负运算符	+	正
		−	负
	乘除运算符	*	乘
		/	除
		MOD	求模
		REM	取余
	其他运算符	* *	指数
		ABS	取绝对值
高		NOT	取反

1）逻辑运算符

在 VHDL 中逻辑运算符共有 6 种，它们分别是：NOT，AND，OR，NAND，NOR，XOR。

这 6 种逻辑运算符可以对 STD_LOGIC 和 STD_LOGIC_VECTOR 等数据进行逻辑运算。当一个语句中存在两个以上的逻辑表达式时，在 C 语言中运算有自左至右的优先级顺序的规定，而在 VHDL 中，左右没有优先级差别。在所有逻辑运算符中 NOT 的优先级最高。

2）算术运算符

VHDL 有 10 种算术运算符，它们分别是：＋，－，＊，/，MOD，REM，＋（一元运算），－（一元运算），＊＊，ABS。

在算术运算中，对于一元运算的操作数（正、负）可以为任何数值类型（整数、实数、物理量）。加法和减法的操作数也和上面一样，具有相同的数据类型，而且参加加、减运算的操作数的类型也必须相同。乘除法的操作数可以同为整数和实数。物理量可以被整数或实数相乘或相除，其结果仍为一个物理量。物理量除以同一类型的物理量即可得到一个整数量。求模和取余的操作数必须是同一整数类型数据。一个指数的运算符的左操作数可以是任意整数或实数，而右操作数应为一整数。

3）关系运算符

VHDL 中有 6 种关系运算符，它们分别是：＝，/＝，＜，＜＝，＞，＞＝。

在关系运算符的左右两边是运算操作数，不同的关系运算符对两边的操作数的数据类型有不同的要求。其中，等号“＝”和不等号“/＝”可以适用所有类型的数据。其他关系运算符则可使用于整数（INTEGER）和实数（REAL）、位（STD_LOGIC）等枚举类型以及位矢量（STD_LOGIC_VECTOR）等数组类型的关系运算。

4）并置运算符

并置运算符“&”用于位的连接。例如，将 4 个位用并置运算符“&”连接起来就可以构成一个具有 4 位长度的位矢量。两个 4 位的位矢量用并置运算符“&”连接起来就可以构成 8 位长度的位矢量。

如 en 是 b(0)～b(3)的允许输出信号，tmp_b ＜＝b　AND (en & en & en & en)

这个语句表示 b 的 4 位位矢量由 en 进行选择得到一个 4 位位矢量的输出。

3.6.3　VHDL 实例

【例 7】 二进制无符号的乘法运算的 VHDL 代码

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity mult8X8 is
    Port (CLK,St: in std_logic;
        Mplier,Mcand : in std_logic_vector(7 downto 0);
        Done : out std_logic;
        Product: out std_logic_vector(15 downto 0));
end mult8X8;
```

```
architecture Behavioral of mult8X8 is
    signal State,NextState: integer range 0 to 3;
    signal count: std_logic_vector (2 downto 0):= "000"; -- 3 - bit counter
    signal A: std_logic_vector (8 downto 0);              -- accumulator
    signal B: std_logic_vector (7 downto 0);
    alias M: std_logic is B(0);                           -- M is bit 0 of B
    signal addout: std_logic_vector (8 downto 0);
    signal K,Load,Ad,Sh: std_logic;
    begin
        Product <= A(7 downto 0) & B;                     -- 16 - bit product is in A and B
        addout <= '0' & A(7 downto 0) + Mcand;          -- adder output is 9 bits including carry
        K <= '1' when count = 7 else '0';
        process (St,State,K,M)
        begin
            Load <= '0'; Sh <= '0'; Ad <= '0'; Done <= '0';   -- initial control signals are '0'
            case State is
            when 0 =>
                if St = '1' then Load <= '1'; NextState <= 1;
                else NextState <= 0; end if;
            when 1 =>
                if M = '1' then Ad <= '1'; NextState <= 2;
                else if K = '0' then Sh <= '1'; NextState <= 1;
                    else Sh <= '1'; NextState <= 3; end if;
                end if;
            when 2 =>
                if K = '0' then Sh <= '1'; NextState <= 1;
                else Sh <= '1'; NextState <= 3;   end if;
             when 3 =>
                Done <= '1'; NextState <= 0;
             end case;
    end process;

        process (clk)
        begin
            if clk'event and clk = '1' then -- update registers on rising edge of clk
                if load = '1' then
                    A <= "000000000";   Count <= "000";      -- clear A and counter
                    B <= Mplier;
                end if;                                      -- load multiplier
                if Ad = '1' then A <= addout; end if;
                if Sh = '1' then
                    A <= '0' & A(8 downto 1);
                    B <= A(0)& B(7 downto 1);                -- right shift A and B
                    count <= count + 1;                      -- increment counter
                        -- uses " + " operator from ieee_std_logic_unsigned package
                end if;
                State <= NextState;
            end if;
        end process;
end Behavioral;
```

该例子可以仿真时序电路,其仿真结果如图 3.29 所示。

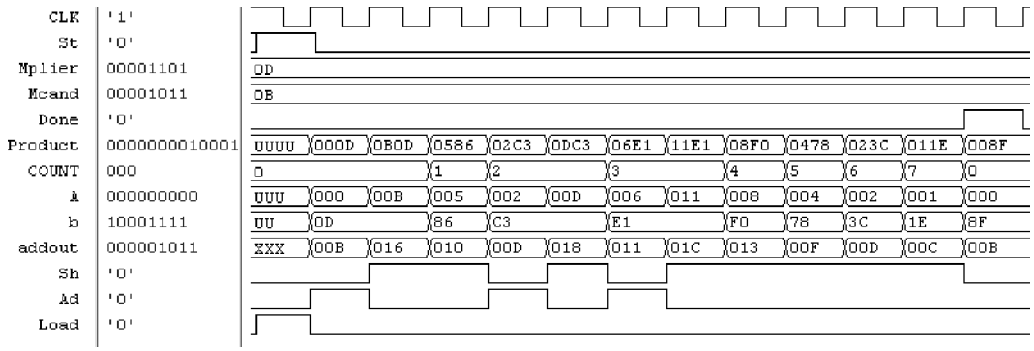


图 3.29 二进制无符号乘法的仿真结果

【例 8】 8 位 Hamming 编码和校验的代码

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

ENTITY hamming IS
PORT (choice     : IN   STD_LOGIC;
      data_in    : IN   STD_LOGIC_VECTOR(7 DOWNTO 0);
      data_ham   : OUT  STD_LOGIC_VECTOR(12 DOWNTO 0);
      data_check : IN   STD_LOGIC_VECTOR(12 DOWNTO 0);
               s : OUT  STD_LOGIC_VECTOR(4 DOWNTO 0)
      );
END hamming;

ARCHITECTURE behav OF hamming IS
BEGIN
 -- 编码过程
 PROCESS(choice,data_in)
 VARIABLE p:STD_LOGIC_VECTOR(4 DOWNTO 0);
 BEGIN
   IF choice = '0' THEN
     p(0)   := data_in(0) XOR data_in(1) XOR data_in(3) XOR data_in(4) XOR data_in(6);
     p(1)   := data_in(0) XOR data_in(2) XOR data_in(3) XOR data_in(5) XOR data_in(6);
     p(2)   := data_in(1) XOR data_in(2) XOR data_in(3) XOR data_in(7);
     p(3)   := data_in(4) XOR data_in(5) XOR data_in(6) XOR data_in(7);
     p(4)   := data_in(0) XOR data_in(1) XOR data_in(2) XOR data_in(3) XOR data_in(4) XOR data_
in(5) XOR data_in(6) XOR data_in(7) XOR p(0) XOR p(1) XOR p(2) XOR p(3);
      -- 按照位置关系,将编码后的校验位和数据位按照正确的位置关系输出
     data_ham(0) <= p(0);
     data_ham(1) <= p(1);
     data_ham(3) <= p(2);
     data_ham(7) <= p(3);
     data_ham(12)<= p(4);
     data_ham(2) <= data_in(0);
     data_ham(4) <= data_in(1);
     data_ham(5) <= data_in(2);
```

```
    data_ham(6) <= data_in(3);
    data_ham(8) <= data_in(4);
    data_ham(9) <= data_in(5);
    data_ham(10)<= data_in(6);
    data_ham(11)<= data_in(7);
  END IF;
END PROCESS;

-- 校验过程
PROCESS(choice,data_check)
VARIABLE p : STD_LOGIC_VECTOR(4 DOWNTO 0); -- 5 位校验位
VARIABLE d : STD_LOGIC_VECTOR(7 DOWNTO 0); -- 8 位数据位
BEGIN
  IF choice = '1' THEN
    -- 从接收到的 Hamming 码中按照位置关系分离出数据位和校验位
    p(0)   := data_check(0);
    p(1)   := data_check(1);
    p(2)   := data_check(3);
    p(3)   := data_check(7);
    p(4)   := data_check(12);
    d(0)   := data_check(2);
    d(1)   := data_check(4);
    d(2)   := data_check(5);
    d(3)   := data_check(6);
    d(4)   := data_check(8);
    d(5)   := data_check(9);
    d(6)   := data_check(10);
    d(7)   := data_check(11);
    -- 根据得到的校验位和数据位计算 s,判断哪位出错;
    s(0)   <= p(0) XOR d(0) XOR d(1) XOR d(3) XOR d(4) XOR d(6);
    s(1)   <= p(1) XOR d(0) XOR d(2) XOR d(3) XOR d(5) XOR d(6);
    s(2)   <= p(2) XOR d(1) XOR d(2) XOR d(3) XOR d(7);
    s(3)   <= p(3) XOR d(4) XOR d(5) XOR d(6) XOR d(7);
    s(4)   <= p(4) XOR d(0) XOR d(1) XOR d(2) XOR d(3) XOR d(4) XOR d(5) XOR d(6) XOR d(7) XOR
p(0) XOR p(1) XOR p(2) XOR p(3);
  END IF;
END PROCESS;
END behav;
```

该例子针对外界输入的 choice 执行不同的功能,其仿真结果如图 3.30 所示。

3.6.4　VHDL 工具介绍

VHDL 的编辑和仿真工具比较多,这里以 Quartus Ⅱ开发工具为例来介绍 VHDL 的编辑和仿真等过程。Quartus Ⅱ开发工具是美国 Altera 公司自行设计提供了完整的多平台设计环境,能满足各种 FPGA、CPLD 的设计需要,是片上可编程系统(SOPC)设计必需的综合性环境。它具有原理图输入和文本输入(采用硬件描述语言)两种输入手段,将设计电路图或电路描述程序变成基本的逻辑单元写入到可编程的芯片中(如 FPGA 芯片)。它是 EDA 设计中不可缺少的一种有用工具,目前在国内使用较为普遍。VHDL 源程序编辑和

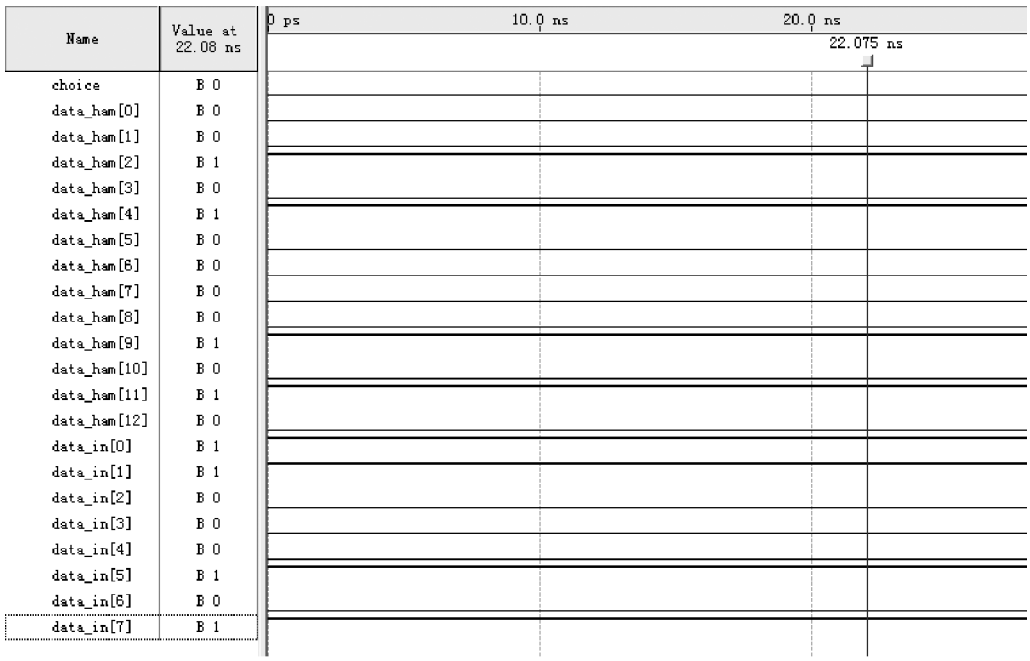


图 3.30　海明码编码的仿真结果

仿真输出界面如图 3.31 所示。综合和输出到硬件需要 CPLD 或者 FPGA 等硬件的支持，这里就不再叙述了。

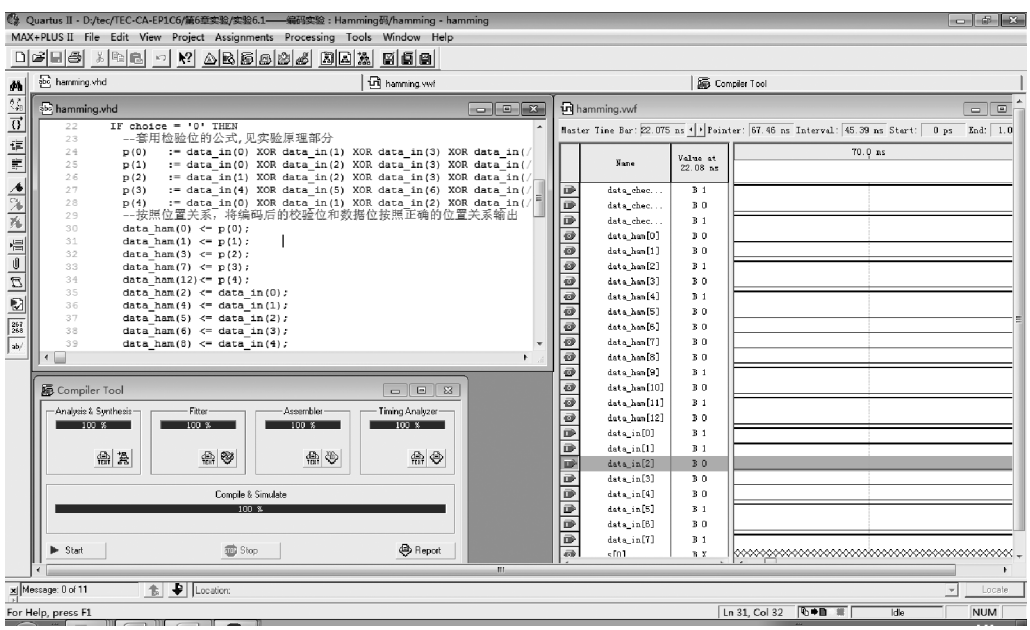


图 3.31　VHDL 源程序编辑和仿真输出界面

Quartus Ⅱ包括模块化的编译器。编译器包括的功能模块有分析/综合器(Analysis & Synthesis)、适配器(Fitter)、装配器(Assembler)、时序分析器(Timing Analyzer)、设计辅助

模块(Design Assistant)、EDA 网表文件生成器(EDA Netlist Writer)、编辑数据接口(Compiler Database Interface)等。可以通过选择 Start Compilation 来运行全局编译,也可以选择 Start 单独运行各个模块。

小结

本章主要介绍了数字电路和逻辑电路的基础,包含布尔函数、布尔函数的化简、各种基本的组合逻辑电路和时序逻辑电路。计算机的主要部件的操作是通过各种逻辑电路组合而成。

计算机的最重要的部件就是它的 ALU。加法器就是一个简单的 ALU,计算机通过加法器来完成一位和多位的加法运算。一位全加器可以由两个半加器构成,多位加法器则是由多个全加器组成的,每个全加器都把自己产生的进位传给它左边的那个全加器。

阵列逻辑电路近年来得到了迅速的发展,阵列逻辑电路主要有半导体存储器和可编程逻辑器件等。ROM,PROM,Flash Memory 和 RAM 等是典型的半导体存储器,在计算机中通过它们的组合构成计算机的存储系统。

可编程逻辑器件由于它的灵活性在很多领域得到了广泛的应用。而在这些可编程逻辑器件中,基于 FPGA 的设计方式成为主流。通过 HDL 描述和设计系统,最终下载到 FPGA 中,就可以让 FPGA 具有设计者指定的功能。FPGA 和 VHDL 的基本理论在本章进行了阐述。

习题

1. 用真值表证明 $X=(X \text{ AND } Y) \text{ OR } (X \text{ AND NOT } Y)$。
2. 先把下列函数化简,然后画出逻辑图。

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

(2) $F=AC+\overline{A}\overline{B}+\overline{B}\overline{C}\overline{D}+B\overline{C}E+\overline{C}DE$

3. 选用 VHDL 工具编写一个布思算法求解的程序。
4. 使用 D 触发器设计一个十二进制的计数器。
5. 解释下列概念:PROM,EPROM,E^2PROM,Flash memory,FPGA,SRAM 和 DRAM。