第 3 章　可编程器件及应用

3.1　可编程逻辑器件

人们一般使用的逻辑器件可分为两大类：固定逻辑器件(ASIC)和可编程逻辑器件(PLD)。

顾名思义，固定逻辑器件中的电路是固定不可更改的，它们被设定用来完成一种或一组功能，从出厂后就确定无法改变，用户根据需要可以选择不同功能的器件，用来完成自己的设计。对于固定逻辑器件，其电路和功能都是生产厂家设定的，根据器件复杂性的不同，生产厂家会花费数月到几年的时间来设计、验证直到生产这些芯片，而且芯片设计制造过程中需要投入大量的人力和物力成本，包括工程资源、昂贵的软件设计工具、用来制造芯片不同金属层的昂贵光刻掩膜组，以及初始原型器件的生产成本等，这些都导致芯片的设计生产是一个非常昂贵而且费时的过程。在这过程中如果出现设计失误，或者功能不能满足需要时，就需要重新设计和生产芯片，造成很大的损失。因此，厂家一般不会改变固定逻辑器件的逻辑电路，而普通使用者也很难去设计和生产自己需要的器件，这一矛盾导致可编程逻辑器件的产生。

可编程逻辑器件的出现解决了固定逻辑器件功能无法改变的问题，为使用者提供更加方便的设计方式。可编程逻辑器件的英文全称为 Programmable Logic Device，一般缩写为 PLD。它也是厂家提供的一种标准逻辑芯片，厂家也投入了巨大的成本来生产这些芯片，但是这些芯片的内部逻辑并没有被固定化，它可以支持用户修改内部逻辑，为用户提供多种逻辑能力、速度和电压特性，而且这种逻辑的修改是在用户现场完成，从而大大方便了逻辑设计。一般的 PLD 的集成度很高，足以满足设计一般的数字系统的需要，用户只要对器件编程来确定其逻辑功能，这样就可以由用户自行设计并把一个数字系统下载在一片 PLD 上，而不必去花昂贵的费用请芯片制造厂商设计和制作专用的集成电路芯片了。

可编程逻辑器件的生产厂家都会提供相应的开发软件，设计人员可利用价格低廉的软件工具快速开发、仿真和测试其设计，并可以很方便地将设计下载到器件中，在实际运行的电路中对设计进行测试。采用可编程器件的另一个主要优点就是在设计阶段中用户可根据需要修改电路，直到设计达到功能需求。

早期的可编程逻辑器件只有可编程只读存储器(PROM)、紫外线可擦除只读存储器(EPROM)和电可擦除只读存储器(EEPROM)3 种。由于结构的限制，它们只能完成简单的数字逻辑功能。后来，出现了结构上稍复杂的可编程芯片，即可编程逻辑器件，它能够完成各种数字逻辑功能。典型的 PLD 由一个“与”门和一个“或”门阵列组成，而任意一个组合逻辑都可以用“与—或”表达式来描述，所以，PLD 能以乘积和的形式完成大量的组合逻辑功能，例如，常见的 GAL。这些早期的 PLD 器件的一个共同特点是可以实现速度特性较好的逻辑功能，但过于简单的结构也使其只能实现规模较小的电路。为了弥补这一缺陷，20

世纪 80 年代中期 Altera 和 Xilinx 分别推出了类似于 PAL 结构的扩展型 CPLD 和与标准门阵列类似的 FPGA，它们都具有体系结构和逻辑单元灵活、集成度高以及适用范围宽等特点。这两种器件兼容了 PLD 和通用门阵列的优点，可实现较大规模的电路，编程也很灵活。

可编程逻辑器件按集成度划分可分为高低两类，早先出现的 PROM、PAL、可重复编程的 GAL 都属于低集成度芯片，可重构使用的逻辑门数量较少，也称为简单 PLD。现在大量使用的 CPLD、FPGA 器件是高集成度芯片，也称为复杂 PLD。如果按照内部结构划分，可分成乘积项结构器件和查找表结构器件两大类。其基本结构为“与—或”阵列的器件是乘积项结构器件，大部分简单 PLD 和 CPLD 都属于这个范畴。由简单的查找表组成可编程门，再构成阵列形式的器件是查找表结构器件，大多数 FPGA 属于此类器件。

数字电路实验平台上使用的是 Altera 公司的 CPLD EPM240，后面会详细介绍这款芯片。

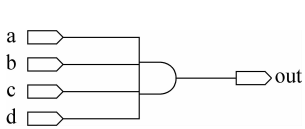
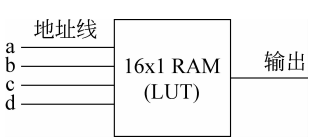
3.1.1 FPGA 工作原理及内部结构

一般的 FPGA 基于查找表结构，下面简单介绍查找表的原理。根据数字逻辑电路的基本知识可以知道，对于一个 n 输入的逻辑运算，不管是与或非运算还是异或运算，输出最多只可能存在 2^n 种结果。所以如果事先将所有的输出可能都存放于一个存储器中，根据输入的不同将相应的结果从存储器中取出作为输出，整个过程就像查找对数表一样，这样就相当于实现了与非门电路的功能。FPGA 的工作原理就是如此，它通过软件生成存储器的内容，然后将内容烧写到查找表存储器中，从而在相同的电路情况下实现不同的逻辑功能。

查找表(Look-Up-Table，LUT)本质上就是一个 RAM。目前 FPGA 中多使用 4 输入的 LUT，所以每一个 LUT 可以看成一个有 4 位地址线的 RAM。当用户通过原理图或 HDL 语言描述一个逻辑电路以后，FPGA 开发软件会自动计算逻辑电路的所有可能结果，并把真值表(即结果)事先写入 RAM，这样，每输入一组信号进行逻辑运算就等于输入一个地址进行查表，找出地址对应的内容，然后输出即可。

下面以一个 4 输入与门电路的例子来说明 LUT 实现逻辑功能的原理。表 3.1 是一个使用 LUT 实现 4 输入与门电路的真值表。

表 3.1　输入与门电路的真值表

<table>
<tr><td colspan="2">逻辑真值表
 
</td><td colspan="2">LUT
 
</td></tr>
<tr><td>输入</td><td>输出</td><td>RAM 地址</td><td>内容</td></tr>
<tr><td>0000</td><td>0</td><td>0000</td><td>0</td></tr>
<tr><td>0001</td><td>0</td><td>0001</td><td>0</td></tr>
<tr><td>⋮</td><td>⋮</td><td>⋮</td><td>⋮</td></tr>
<tr><td>1111</td><td>1</td><td>1111</td><td>1</td></tr>
</table>

从表中可以看到，LUT 具有和逻辑电路相同的功能。对于更加复杂的逻辑，在实际中会将很多个 LUT 连接起来，一些 LUT 的输出作为其他 LUT 的输入，这样就能实现更大规模的逻辑，由于基于 LUT 的 FPGA 具有很高的集成度，其器件密度从数万门到数千万门不等，可以完成极其复杂的时序逻辑与组合逻辑电路功能，所以适用于高速、高密度的高端数字逻辑电路设计领域。

目前流行的 FPGA 除了基于查找表技术之外还整合了一些常用的功能（如 RAM、时钟管理等）的硬核（ASIC 型）模块。如图 3.1 所示为 FPGA 的一般结构，主要由 6 部分组成，分别为可编程输入输出单元、可编程逻辑单元、嵌入式块 RAM、丰富的布线资源、底层嵌入功能单元和内嵌专用硬核等。

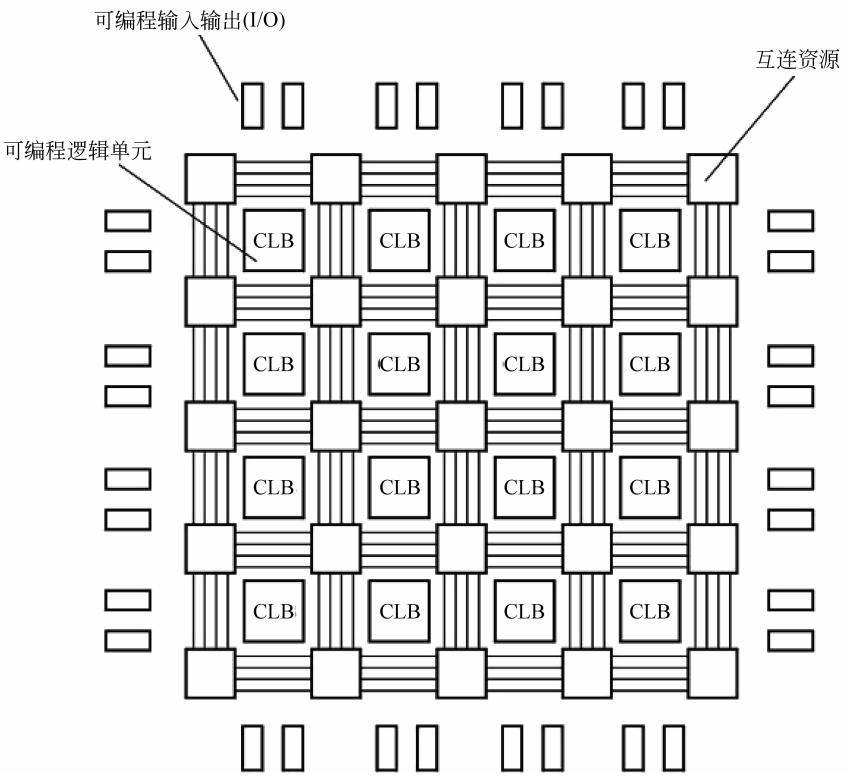


图 3.1　FPGA 芯片的内部结构

1. 可编程输入输出单元

可编程输入输出单元简称为 I/O 单元，是芯片与外界电路的接口部分，完成不同电气特性下对输入输出信号的驱动与匹配要求，其示意结构如图 3.2 所示。FPGA 内的 I/O 按组分类，每组都能够独立地支持不同的 I/O 标准。通过软件的灵活配置，可适配不同的电气标准与 I/O 物理特性，可以调整驱动电流的大小，可以改变上、下拉电阻。

外部输入信号可以通过 I/O 模块的存储单元输入到 FPGA 的内部，也可以直接输入 FPGA 内部。为了便于管理和适应多种电器标准，FPGA 的 I/O 单元被划分为若干个组（Bank），每个 Bank 的接口标准由其接口电压 V_{CCO} 决定，一个 Bank 只能有一种 V_{CCO}，但不同 Bank 的 V_{CCO} 可以不同。只有相同电气标准的端口才能连接在一起，V_{CCO} 电压相同是接

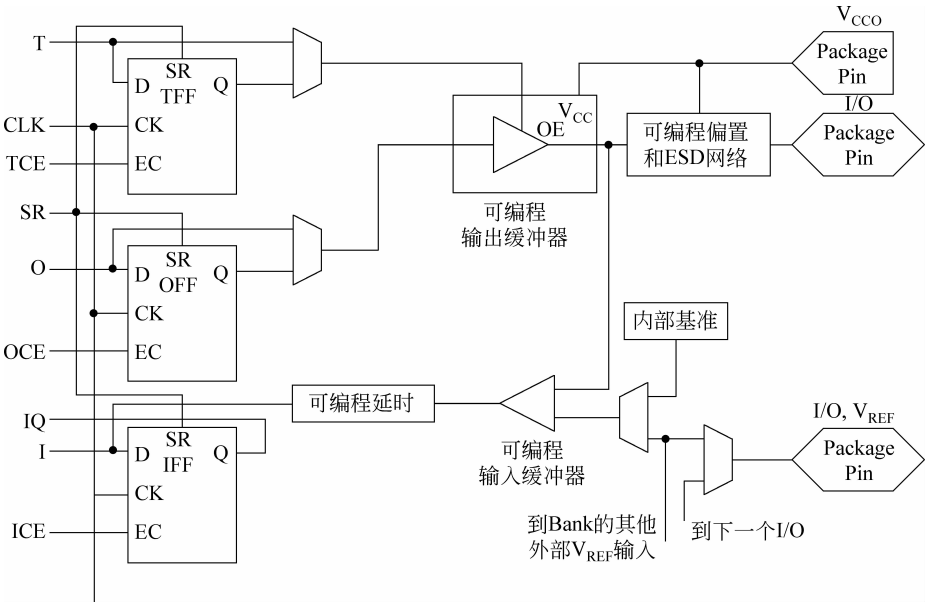


图 3.2　典型的 I/O 单元内部结构示意图

口标准的基本条件。

2. 可编程逻辑单元(CLB)

CLB 是 FPGA 内的基本逻辑单元,由它来完成主要逻辑功能。FPGA 的基本可编程逻辑单元是由查找表(LUT)和寄存器(Register)组成,由查找表来完成纯组合逻辑功能;而 FPGA 内部寄存器可配置为同步/异步复位和置位、时钟使能的触发器,也可以配置为锁存器,从而构成逻辑的时序部分。FPGA 一般依赖寄存器完成同步时序逻辑设计。一般来说,比较经典的基本可编程单元的配置是一个寄存器加一个查找表,但不同厂商的寄存器和查找表的内部结构有一定差异,而且寄存器和查找表的组合模式也不同。

如 Xilinx 公司的基本逻辑单位是 Slice,其内部结构如图 3.3 所示,一个 Slice 由两个 4 输入的查找表和两个寄存器组成,还有进位逻辑的两条快速进位链用于提高 CLB 模块的处理速度。

3. 嵌入式块 RAM

目前大多数 FPGA 都有内嵌的块 RAM,这大大拓展了 FPGA 的应用范围和灵活性。嵌入式块 RAM 可被配置为单端口 RAM、双端口 RAM、内容地址存储器(CAM)以及 FIFO 等常用存储结构。除了块 RAM,还可以将 FPGA 中的 LUT 灵活地配置成 RAM、ROM 和 FIFO 等结构。在实际应用中,芯片内部块 RAM 的数量也是选择芯片的一个重要因素。单片块 RAM 的容量为 18Kb,即位宽为 18b、深度为 1024,可以根据需要改变其位宽和深度。

4. 丰富的布线资源

布线资源连通 FPGA 内部所有单元,连线的长度和工艺决定着信号在连线上的驱动能力和传输速度。布线资源可划分为以下几类。

(1) 全局性的专用布线资源。用于完成器件内部的全局时钟和全局复位/置位的布线。

(2) 长线资源。用于完成器件 Bank 间的一些高速信号和一些第二全局时钟信号的

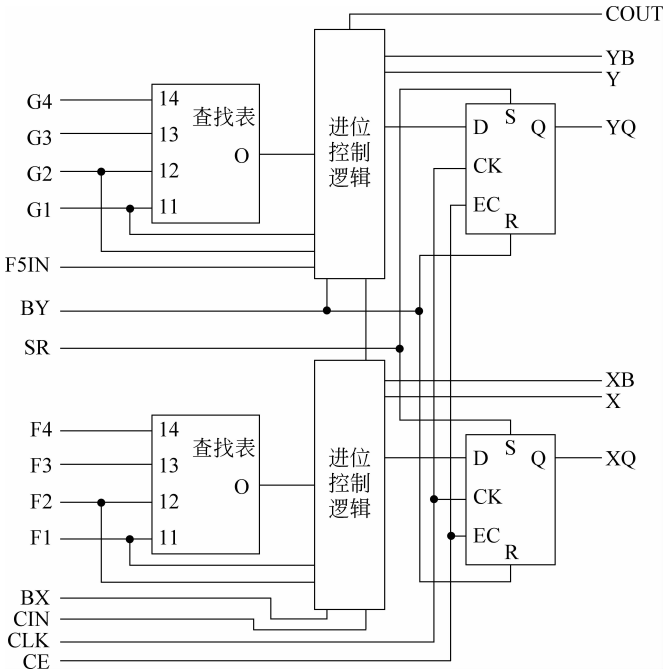


图 3.3　Slice 结构示意图

布线。

(3) 短线资源。用来完成基本逻辑单元间的逻辑互连与布线。

(4) 其他。在逻辑单元内部还有着各种布线资源和专用时钟、复位等控制信号线。

由于在设计过程中,往往由布局布线器自动根据输入的逻辑网表的拓扑结构和约束条件选择可用的布线资源连通所用的底层单元模块,所以常常忽略布线资源。其实布线资源的优化与使用和实现结果有直接关系。

5. 底层嵌入功能单元

底层嵌入功能单元是指通用程度较高的嵌入式功能模块,如锁相环(Phase Locked Loop,PLL)、DLL(Delay Locked Loop)、DSP(Digital Signal Processing)和 CPU 等。

6. 内嵌专用硬核

它与"底层嵌入单元"是有区别的,这里的硬核主要是指那些通用性相对较弱,不是所有 FPGA 器件都包含硬核。

3.1.2　CPLD 工作原理及内部结构

一般的 CPLD 都是采用基于乘积项(Product-Term)的 PLD 结构,如图 3.4 所示是一款 CPLD 的总体结构。

这种 PLD 可分为三块结构:宏单元(Macrocell)、可编程连线(PIA)和 I/O 控制块。宏单元是 PLD 的基本结构,由它来实现基本的逻辑功能。图 3.4 中 LAB 是多个宏单元的集合(因为宏单元较多,没有一一画出)。可编程连线负责信号传递,连接所有的宏单元。I/O 控制块负责输入输出的电气特性控制,例如可以设定集电极开路输出、三态输出等。INPUT/GLCK1、INPUT/GCLRn、INPUT/OE1、INPUT/OE2 是全局时钟,清零和输出使

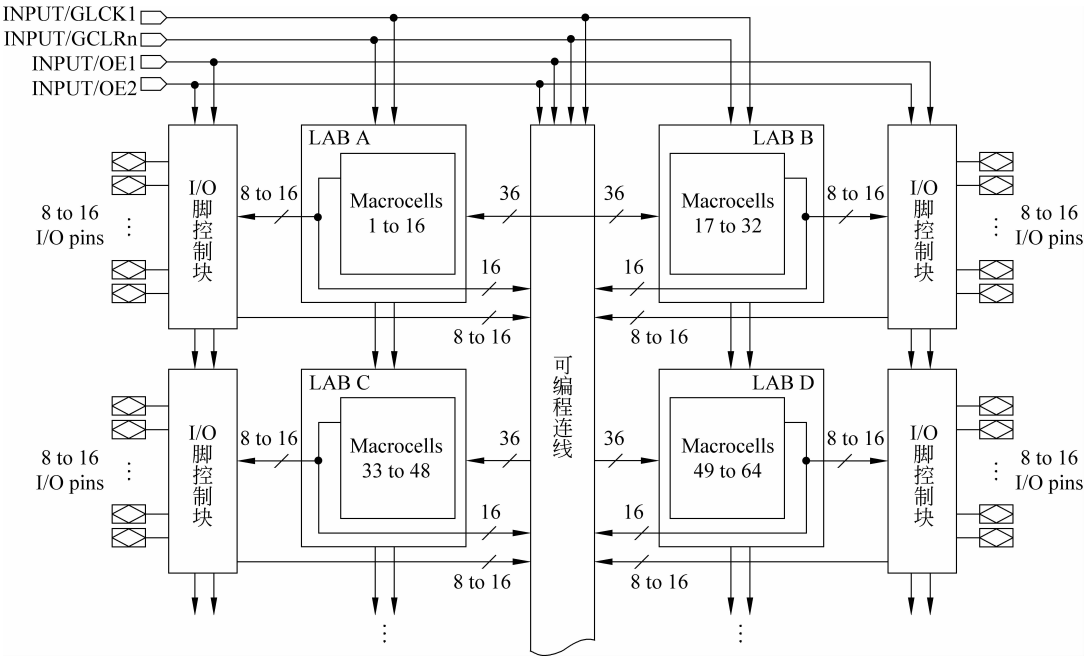


图 3.4　基于乘积项的结构

能信号，这几个信号有专用连线与 PLD 中每个宏单元相连，信号到每个宏单元的延时相同并且延时最短。

宏单元的具体结构如图 3.5 所示。

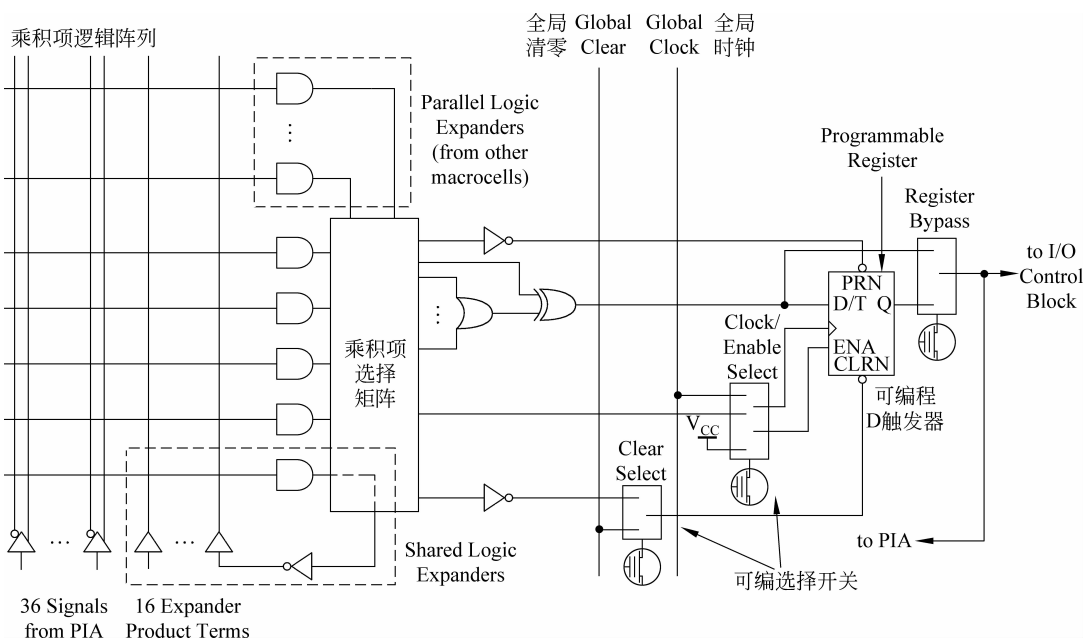


图 3.5　宏单元的具体结构

左侧是乘积项阵列，实际就是一个与或阵列，每一个交叉点都是一个可编程熔丝，如果导通就是实现“与”逻辑。后面的乘积项选择矩阵是一个“或”阵列。两者一起完成组合逻辑。图右侧是一个可编程 D 触发器，它的时钟，清零输入都可以编程选择，可以使用专用的全局清零和全局时钟，也可以使用内部逻辑（乘积项阵列）产生的时钟和清零。如果不需要触发器，也可以将此触发器旁路，信号直接输给 PIA 或输出到 I/O 脚。

下面以一个简单的电路为例，具体说明 PLD 是如何利用以上结构实现逻辑的，电路如图 3.6 所示。

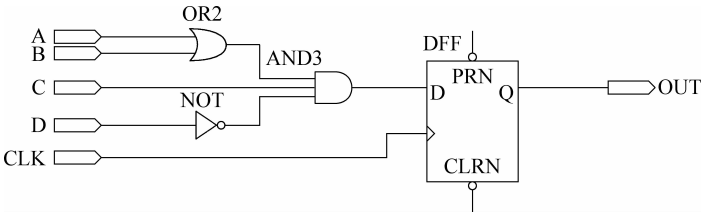


图 3.6　用 PLD 实现一个简单组合逻辑电路

假设组合逻辑的输出（AND3 的输出）为 f，则 $f=(A+B)*C*(!D)=A*C*!D+B*C*!D$。

PLD 将以下面的方式来实现组合逻辑 f，如图 3.7 所示。

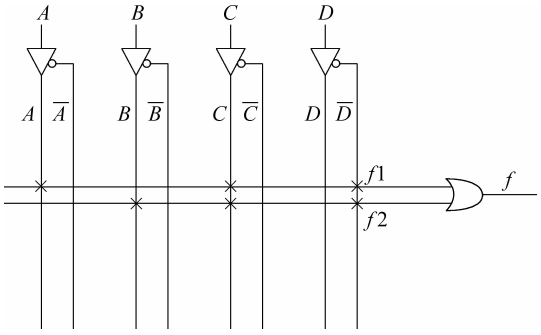


图 3.7　PLD 实现图 3.6 所示的组合逻辑电路功能

A、B、C、D 由 PLD 芯片的引脚输入后进入可编程连线阵列（PIA），在内部会产生 A、$\overline{A}$、B、$\overline{B}$、C、$\overline{C}$、D、$\overline{D}$ 8 个输出。图中每一个叉表示相连（可编程熔丝导通），所以得到：$f=f1+f2=(A*C*!\ D)+(B*C*!\ D)$。这样组合逻辑就实现了。图 3.6 中 D 触发器的实现比较简单，直接利用宏单元中的可编程 D 触发器来实现。时钟信号 CLK 由 I/O 脚输入后进入芯片内部的全局时钟专用通道，直接连接到可编程触发器的时钟端。可编程触发器的输出与 I/O 脚相连，把结果输出到芯片引脚。这样 PLD 就完成了图 3.6 所示电路的功能。以上这些步骤都是由软件自动完成的，不需要人为干预。

图 3.6 的电路是一个很简单的例子，只需要一个宏单元就可以完成。但对于一个复杂的电路，一个宏单元是不能实现的，这时就需要通过并联扩展项和共享扩展项将多个宏单元相连，宏单元的输出也可以连接到可编程连线阵列，再作为另一个宏单元的输入。这样 PLD 就可以实现更复杂逻辑。这种基于乘积项的 PLD 基本都是由 EEPROM 和 Flash 工艺制造的，一上电就可以工作，无须其他芯片配合。

3.1.3 EPM240 简介

数字逻辑实验平台上的可编程逻辑器件区只有一片可编程逻辑器件，是 Altera 公司的 CPLD，具体型号是 EPM240T100C5。首先先了解一下该器件的命名规则，这样就可以从器件的型号看出这是怎么样性能的一款器件，例如，EPM240T100C5 具体含义如下：EPM 代表器件类型为 MAX Ⅱ系列；240 代表逻辑单元(LE)的数量，即 240 个逻辑单元；T 代表封装形式为薄塑封四角扁平封装(Thin Quad Flat Package，TQFP)；100 代表引脚数为 100 脚；C 代表工作温度范围(确定其是工业级，军品级，还是商业级)；5 代表速度等级(数字越小速度越快)。

MAX Ⅱ系列是目前市场是非常成功的一代可编程器件，与上一代 MAX 产品相比，其成本降低了一半，功耗只有上一代的十分之一，同时保持 MAX 系列原有的瞬态启动、单芯片、非易失性和易用性。在消费类、通信、工业和计算机产品领域得到广大设计者的青睐，MAX Ⅱ系列器件的主要特点如下。

(1) 成本优化的架构。新型 MAX Ⅱ CPLD 架构包括基于 LUT 的 LAB 阵列、非易失性 Flash 存储模块和 JTAG 控制电路。

(2) 低功耗。MAX Ⅱ器件是动态功耗较低的 CPLD。

(3) 高性能。MAX Ⅱ器件支持高达 300MHz 的内部时钟，可为用户提供更高的系统级性能。

(4) 用户 Flash 存储器。MAX Ⅱ CPLD 内的用户 Flash 存储器是一个大小为 8Kb、用户可访问且可编程的 Flash 存储器块，可用于用户自己定义的数据。

(5) 实时在系统可编程能力(ISP)。MAX Ⅱ器件支持实时在系统可编程。

(6) 灵活的多电压内核。MAX Ⅱ架构支持 MultiVolt，允许器件在 1.8V、2.5V 或 3.3V 电压环境下工作。

(7) I/O 能力。MAX Ⅱ CPLD 支持多种 I/O 标准，最大容忍 5V 电压输入。

EPM240T100C5 这款器件有 240 个逻辑单元，等效宏单元 192 个，资源比较丰富，内有 8Kb Flash 的存储空间，具有最大 80 个用户 I/O，端对端最小延时为 5.5ns，足够课程实验使用。EPM240 的顶视图如图 3.8 所示。

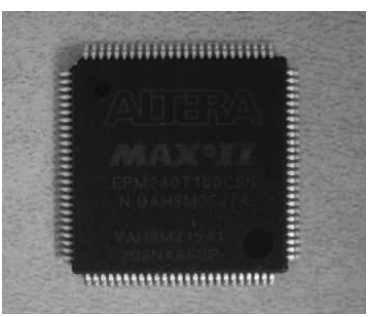


图 3.8 EPM240 的顶视图

该器件的左上角有一个圆点，表明引脚 1 的位置，如图 3.8 所示，然后按逆时针顺序引脚号依次递增。对于一般的贴片封装的器件来说，器件上都会有一个圆点(或者三角标识)标明引脚 1 的位置，然后再按逆时针顺序数引脚号。

3.2 硬件描述语言 VHDL 简介

硬件描述语言(Hardware Description Language，HDL)就是可以表述硬件的语言，它可以用文本语言的形式描述硬件电路的功能、信号连接关系以及时序关系等。HDL 的出现为

硬件设计提供了一种更加快捷精确的描述方法，成为连接硬件设计者和电子设计自动化(EDA)软件工具之间的桥梁。相对于以往的图形输入，硬件描述语言虽然没有那么直观，但是其描述能力更强，可描述的层次也更高，因而可以进行大规模的数字系统设计。

VHDL(Very-High-Speed Integrated Circuit Hardware Description Language)是本书采用的硬件设计语言。VHDL 主要用于描述数字系统的结构、行为、功能和接口，与其他硬件描述语言相比，VHDL 具有以下特点。

(1) 功能强大、设计灵活。

(2) 支持广泛、易于修改。

(3) 强大的系统硬件描述能力。

(4) 独立于器件的设计、与工艺无关。

(5) 很强的移植能力。

(6) 易于共享和复用。

VHDL 采用基于库(Library)的设计方法，可以建立各种可再次利用的模块。这些模块可以预先设计或使用以前设计中的存档模块，将这些模块存放到库中，就可以在以后的设计中进行复用，可以使设计成果在设计人员之间进行交流和共享，减少硬件电路设计。

本书中的 VHDL 部分简单地介绍了一些 VHDL 的基本语法，读者如果想要学习一些深入的内容，请参考其他一些专门的语言类书籍。

3.2.1 基本结构

一个完整的 VHDL 程序就是一个电路的抽象，它可以代表一个复杂的电路系统、一块电路板、一片芯片、一个电路模块或者一个门电路，这样的 VHDL 程序称为一个设计单元，它抽象了一个实际电路。一个 VHDL 程序的结构如图 3.9 所示。

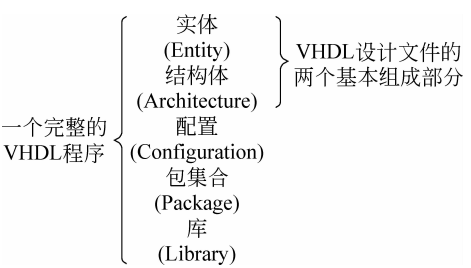


图 3.9 VHDL 程序组成

一个设计单元有以下几部分组成：实体(Entity)、结构体(Architecture)、配置(Configuration)、包集合(Package)、库(Library)。例 3.1 为一个 2 输入与门的 VHDL 程序。

【例 3.1】 一个 2 输入的与门的逻辑描述。

```
LIBRARY ieee;                          --库说明语句
USE ieee.std_logic_1164.ALL;           --程序包说明语句

ENTITY and2 IS                         ┐
    PORT(a,b : IN STD_LOGIC;           │
    y: OUT STD_LOGIC);                 ├实体
END and2;                              ┘

ARCHITECTURE and2x OF and2 IS          ┐
BEGIN                                  │
    y<=a AND b;                        ├结构体
END and2x;                             ┘
```

从上面的例子可以看出，实体部分描述了设计系统的外部接口信号（即输入输出信号），它从整体上表示了设计单元的外部特性；结构体则描述了设计单元的内部电路，它显示了设计单元的行为、数据的流程或组织结构形式；配置在例子中没有体现，它用于从库中选取所需元件安装到设计单元的实体中，来组成设计的不同规格的不同版本，使设计单元的功能发生变化；包集合存放了各设计模块能共享的数据类型、常数、子程序等；库中存放了已编译的实体、结构体、包集合和配置。

1. 实体

实体（Entity）是 VHDL 中描述元件外部特性的部分，即元件的输入和输出（I/O）的端口等信息，它并不描述元件的具体功能，在电路原理图上实体相当于元件符号。在层次化系统设计中，顶层的实体可以是整个系统模块或整个单元模块的输入、输出（I/O）描述；在一个器件级的设计中（底层），实体可以是一个元件或芯片的输入、输出（I/O）描述；实体在 VHDL 程序设计中描述一个元件或一个模块与设计系统的其余部分（其余元件、模块）之间的连接关系，可以看作是一个电路符号。如下面的例子就描述了一个二选一的多路选择器的实体部分，对应于元件符号如图 3.10 所示。

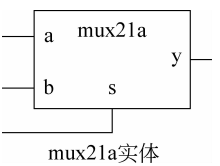


图 3.10　元件符号

```
ENTITY mux21a IS
PORT(a, b: IN BIT;
     s: IN BIT;
     y: OUT BIT);
END ENTITY mux21a;
```

一个实体由实体名、类属表、端口表、实体说明部分和实体语句部分组成，VHDL 中实体的语法格式如下：

```
ENTITY 实体名 IS
    [GENERIC(类属表);]
    [PORT(端口表);]
END [ENTITY] 实体名;
```

在这里 ENTITY、GENERIC、PORT、IS、BEGIN、END 是 VHDL 的关键字（保留字），实体名、端口名等均应为符合 VHDL 命名规则的标识符。实体中的每一个 I/O 信号称为端口，其功能对应于电路图符号的一个引脚。端口说明则是对一个实体的一组端口的定义，即对基本设计实体与外部接口的描述。端口是设计实体和外部环境动态通信的通道。

类属参数说明是可选部分，用来指定该设计单元的类属参数（如延时、功耗等），类属参数说明必须放在端口说明之前。参数在该模块被调用时从外部传入参数值，参数值为本实体所属的所有结构体使用，传入的类属参数作为常量使用，在使用时不能修改。类属特性的设计为多个高层设计实体使用同一个元件提供了静态参数上的区别。例如，一个通用的计数器元件，在高层使用时通过类属参数的指定，可以设定为 8b、16b 或者 24b 的计数器。

类属参数说明的语法格式如下：

```
GENERIC(端口名{,端口名}: [IN] 子类型 [:=初始值]
{;端口名{,端口名}: [IN] 子类型 [:=初始值]});
```

例如，指定上升沿时间为 3ns 的说明如下：

```
GENERIC(rising_edge: TIME: =3ns)
```

还可能会有实体说明部分,它是实体接口中的公共信息,放在端口说明之后,如定义一些子类型说明等。

端口为实体提供了与外部通信的通道,每个端口就相当于一个元件 I/O,对应于电路图符号的一个引脚。端口说明中的每一个 I/O 信号称为一个端口,一个端口就是一个数据对象,必须有端口名、端口模式和数据类型。

端口模式用来说明数据、信号传输通过该端口的方向。端口模式有以下几类。

(1) IN(输入)。定义的端口为单向只读模式,数据只能通过该端口被读入实体中,仅允许数据流进入端口,主要用于时钟输入、控制输入、单向数据输入。

(2) OUT(输出)。定义的端口为单向输出模式,数据只能通过该端口从实体向外流出,或者说可以将实体中的数据向此端口赋值。该模式通常用于终端计数一类的输出,不能用于反馈。

(3) INOUT(双向)。定义的端口为双向输入输出模式,允许数据流入或流出该实体。这个端口相当于一个输入端口和一个输出端口的组合。从端口内部看,可以对此端口进行赋值,也可以通过此端口读入外部的数据信息;从端口的外部看,信号既可以从此端口流出,也可以向此端口输入信号,如 RAM 的数据端口。该模式允许用于内部反馈。

(4) BUFFER(缓冲)。功能与 INOUT 类似,区别是当需要输入数据时,只允许回读内部信号,即允许反馈,该回读信号不是由外部输入,而是由内部产生。例如,计数器设计,可将计数器输出的计数信号回读,以作为下一计数值的初值。该模式允许数据流出该实体和作为内部反馈使用,但不允许作为双向端口使用。

如果端口模式没有指定,则该端口处于默认模式为 IN。

端口的数据类型原则上可以是任何标准的数据类型和用户自定义类型,这个根据需要自由选择,但是要注意当端口连接的时候数据类型的匹配。

图 3.11 是一个多端口的电路图。

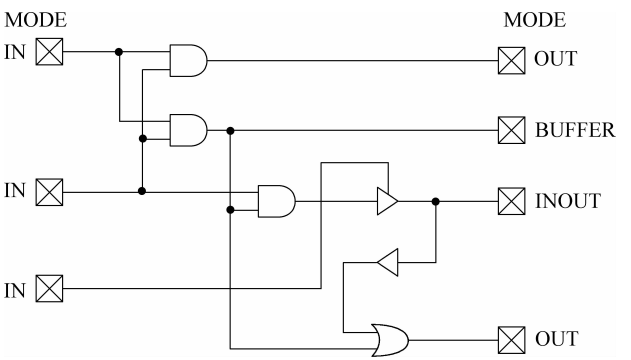


图 3.11 端口说明

这个电路图对应的实体有 3 个输入端口,两个输出端口,一个双向端口和一个缓冲端口,其对应的实体如下:

```
ENTITY ent IS
PORT(a, b, c: IN BIT;
```

```
    e, f  :OUT BIT;
       g    :BUFFER BIT;
    d : INOUT BIT);
END ENTITY ent;
```

2. 结构体

结构体(Architecture)是元件的功能部分,具体说明该实体的行为,定义该实体的功能,规定该实体的数据流程,指定实体中内部元件的连接关系。若把设计实体抽象为一个功能方块图,结构体则描述这个功能方块图的内部实现细节。对于一个实体来说,具体的电路实现可能有多种,因此一个电路系统程序设计的一个实体,可以对应多个结构体,以对应该元件的不同设计方案,通过配置来选择不同的实现结构体。当然,一个独立的 VHDL 文件一般由一个实体说明和一个结构体组成。例如,图 3.10 中的二选一多路选择器的实体,其可能的结构体图如图 3.12 所示。

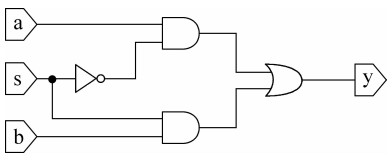


图 3.12　mux21a 结构体

这个图示所对应的结构体如下:

```
ARCHITECTURE one OF mux21a IS
    BEGIN
    y<=(band s) or (a and (not s));
END ARCHITECTURE one;
```

从上面的例子也可以看出,使用语言来描述这个元件比使用原理图要方便很多。

VHDL 中结构体语法结构如下:

```
ARCHITECTURE 结构体名 OF 实体名 IS
    [结构体说明部分];
BEGIN
    [并发处理语句];
END 结构体名;
```

结构体是对应于实体存在的,因此它必须有一个已经定义的实体,并且显式地表示出来。

结构体说明语句是对结构体的功能描述语句中将要用到的信号(SIGNAL)、数据类型(TYPE)、常数(CONSTANT)、函数(FUNCTION)和过程(PROCEDURE)等加以说明的语句。结构体中说明和定义的数据类型、常数、元件、函数和过程只能用于这个结构体中,若希望其能用于其他的实体或结构体中,则需要将其作为程序包来处理。

并发处理语句位于 BEGIN 和 END 之间,这些语句具体地描述了结构的行为。并发处理语句是功能描述的核心部分,也是变化最丰富的部分。并发处理语句可以使用赋值语句、进程语句、元件例化语句、块语句以及子程序等。需要注意的是,这些语句都是并发(同时)执行的,与排列顺序无关。

3. 库

包集合、实体、结构体编译之后可以存放在库(Library)里,通过其目录可查询和调用库里的内容。库的功能类似于操作系统中的目录,存放设计的数据,使设计者可以共享已经编译过的设计结果。库的引用说明总是放在设计实体的最前面,其语法结构如下:

```
LIBRARY 库名;
USE<库名>.<包名>.ALL;
```

引用后,在设计实体内的语句就可以使用库中的内容了,在 VHDL 语言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。VHDL 语言中的库分为设计库和资源库两类,如图 3.13 所示。

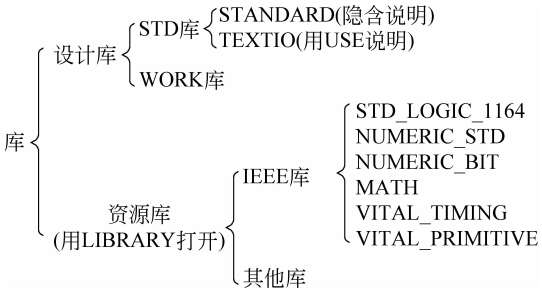


图 3.13　VHDL 库

1) 设计库

设计库是由 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。

(1) STD 库。VHDL 的标准库,预定义了 STANDARD 和 TEXTIO 两个包集合,这两个包集合是使用 VHDL 语言时最常用到的内容。STANDARD 的包集合不需要引用,其中所有预定义的数据类型和函数都可以使用,为所有设计单元所共享、隐含定义、默认和可见。若要使用 TEXTIO 包集合中的内容时,应先说明库和包集合名,然后才可使用该包集合中的内容。

(2) WORK 库。它是 VHDL 语言的工作库,用户在工程中已经设计好的,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。WORK 库对所有设计都是隐含可见的,因此在使用该库时无须进行任何说明。

也就是说,在每段 VHDL 程序中都隐含有下面的不可见的代码:

```
LIBRARY WORK;
LIBRARY STD;
USE STD.STANDARD.ALL;
```

2) 资源库

除了 STD 和 WORK 库以外,其他的库都为资源库,若想使用这些库中的内容都必须预先声明引用。资源库用来存放编译好的常规元件和标准模块,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,如 STD_LOGIC_1164、STD_LOGIC_ARITH(算术运算库)、STD_LOGIC_UNSIGNED 等。通常 VHDL 工具厂商与 EDA 工具专业公司也有自己的资源库,例如 ASIC 矢量库,为了进行门级仿真,各公司可提供面向 ASIC 的逻辑门库,在该库中存放着与逻辑门一一对应的实体。当然,用户也可以有自己定义的库,简称用户库,是由用户自己创建并定义的库。设计者可以把自己经常使用的非标准(一般是自己开发的)包集合和实体等汇集成在一起定义成一个库,作为对 VHDL 标准库的

补充。

常用的 IEEE 资源库包含的程序包如下：

```
STD_LOGIC_1164
NUMERIC_BIT
MATH_REAL
```

库中包集合的使用如下：

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

4. 结构体子结构

在规模较大的电路设计中，整个电路将被分成若干个相对独立的模块来描述。这样，一个结构体可以用几个子结构，即相对独立的几个模块来构成。VHDL 语言有以下 3 种形式的子结构描述语句：块 BLOCK 语句结构、进程 PROCESS 语句结构和子程序 SUBPROGRAMS 结构。

1）块 BLOCK

一般一个大规模的电路通常可以分成多个子模块，以便于设计和开发。同样，在 VHDL 程序中，结构体内部也可以由多个 BLOCK 块组成，每一个 BLOCK 块则对应一个子模块，其原理就如同电路图被分成若干个部分，每个子原理图对应于一个 VHDL 程序中的 BLOCK 块。

BLOCK 的语法格式如下：

```
块标号：BLOCK [(块保护条件)]
    [说明语句];
BEGIN
    [并发处理语句];
END BLOCK 标号名;
```

其中保护条件是可选项，它是一个布尔表达式，只有当其为真时，该块中的语句才被启动执行；否则，该块中的语句不被执行，如果有保护条件，则该条件应用圆括号括起来，放在 BLOCK 之后。BLOCK 语句中所描述的各个语句是可以并行执行的，它和书写顺序无关。下面是一个二选一选择器的 VHDL 实现，其中就是用了块语句。

```
ENTITY mux2_1 IS
    PORT(d0, d1, sel : IN STD_LOGIC;
         q           : OUT STD_LOGIC);
END mux2_1;
ARCHITECTURE amux OF mux2_1 IS
    SIGNAL tmp1,tmp2,tmp3 : STD_LOGIC;
BEGIN
    B1: BLOCK
    BEGIN
        tmp1<=d0 AND sel;
        tmp2<=d1 AND (not sel);
```

```
    END BLOCK B1;

    B2: BLOCK
    BEGIN
        tmp3<=tmp1 OR tmp2;
        q<=tmp3;
    END BLOCK B2;
END amux;
```

2）进程 PROCESS

PROCESS 语句是一种并发处理语句，在一个构造体中多个 PROCESS 语句可以同时并发运行。因此，PROCESS 语句是 VHDL 中描述硬件系统并发行为的最常用、最基本的语句。我们会在后面详细介绍其用法。

3）子程序

VHDL 的子程序有两种类型：过程（PROCEDURE）和函数（FUNCTION）。它们可以在结构体或包集合的任何位置被调用，而且可以反复调用。子程序返回后才能再被调用，不能递归调用。在调用时子程序首先要进行初始化，执行结束后子程序终止；再调用时要再进行初始化。因此，子程序内部的值不能保持。VHDL 的子程序具有可重载性，即允许有许多重名的子程序，但这些子程序的参数类型和返回数值类型是不同的。

过程（PROCEDURE）定义的语法结构如下：

```
PROCEDURE  过程名(参数 1; 参数 2; …)IS
    [定义语句];
BEGIN
    [顺序处理语句];
END 过程名;
```

调用过程语句的时候只要加上参数表就可以了，下面是一个求最大值的过程定义和调用实例：

```
PROCEDURE max(a, b: IN INTEGER;
              y: OUT INTEGER) IS
BEGIN
    IF (a<b) THEN
        y<=b;
    ELSE
        y<=a;
    END IF;
END max;
```

过程的调用：

```
max (x, y, maxout);
```

函数（FUNCTION）的用法同过程类似，但是函数比过程多了返回值，其语法结构如下：

```
FUNCTION  函数名(输入参数表)RETUEN 数据类型 IS
```

```
    [定义语句];
BEGIN
    [顺序处理语句];
    RETUEN [返回变量名];
END  [函数名];
```

函数定义的时候有返回的类型,函数体中也需要由 RETURN 语句来返回结果。

3.2.2 语言元素

1. 标识符

VHDL 语言是不区分大小写的,但所使用的名字,如信号名、实体名、结构体名、变量名、各种进程标记、块标记等这些标识符进行命名时,应遵守如下规则。

(1) 第一个字符必须是字母。

(2) 构成名字的字符只能用英文字母、数字和下划线。

(3) 不能连续使用下划线,标识符的最后一个字符也不能用下划线。

(4) VHDL 的保留字(关键字)不能用于标识符。

如下标识符是合法的:

```
tx_clk
three_state_Enable
sel7D
HIT_1124
```

如下标识符是非法的:

_tx_clk	标识符必须起始于字母
8B10B	第一个字符必须是字母
large# number	只能是字母、数字、下划线
link__bar	不能有连续两个下划线
select	关键字(保留字)不能用于标识符
rx_clk_	最后字符不能是下划线

在编写 VHDL 程序的时候,为了使程序结构清晰,具有更好的可读性,建议采用以下的书写规则。

(1) 对 VHDL 语言的保留字,习惯上用大写,其他应小写。但有一种情况需要注意,代表不定状态的 X 和高阻态的 Z 要求必须大写。

(2) 单词、信号名的含义要明确,以免造成混乱。

(3) 要求段落分明、含义确切,嵌套关系一目了然。

(4) 应辅以适量的程序注释。VHDL 语言中使用的注释符是"--",从注释符号"--"开始到该行末尾结束。所注释的文字不作为语句来处理,不产生硬件电路结构,不描述电路硬件行为。

2. 数据对象

在 VHDL 语言中,数据对象是可以赋予一个值的客体,即可以存储值的容器,常用的数据对象有 4 类:常量、变量、信号和文件。信号和变量可以连续地被赋值,而常量只能在它

被说明时赋值。文件内容是不可以通过赋值来更新的，文件可以作为参数向子程序传递，通过子程序对文件进行读和写操作，文件参数没有类型。

1）常量(CONSTANT)

常量是一个固定的值，在设计描述中不会变化，一般是全局量，通常是电路中不变的那些值，如电源、地等常数。在常量说明的过程中就对其赋予一个固定的值，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。常量说明的一般语法格式如下：

CONSTANT 常量名{，常量名}：数据类型：=表达式；

例如：

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

常量一旦被赋值就不能再改变。上面 VCC 被赋值为 5.0V，那么在所在的 VHDL 语言程序中 VCC 的值就固定为 5.0V，不像后面所提到的信号和变量那样，可以任意赋值不同的数值。另外，常量所赋的值应和定义的数据类型一致。

2）变量(VARIABLE)

变量是暂存数据的局部量，它在电路中没有实际对应的对象，只能在进程语句、函数语句和过程语句结构中使用。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。变量说明的语法格式如下：

VARIABLE 变量名{，变量名}：数据类型 [：=初始值]；

变量声明：

```
VARIABLE x, y: INTEGER;
VARIABLE count: INTEGER RANGE 0 TO 99:=0;
```

由于变量是一个局部量，它必须在进程或子程序的说明区域中加以说明。

变量赋值的时候表达式必须与目标变量具有相同的数据类型，赋值语法格式如下：

目标变量名：=表达式；

变量赋值是直接的、非预设的，它在某一时刻仅包含一个值。变量的赋值立即生效，不存在延时行为。变量常用在实现某种运算的赋值语句中。变量在赋值时不能产生附加延时。例如，tmp3 是变量，那么下式产生延时的方式是不合法的：

```
tmp3: =tmp1+tmp2 AFTER 10μs;
```

3）信号(SIGNAL)

信号是硬件设计语言特有的一种对象，是电子电路内部硬件之间相互的连线的抽象表示，除没有方向外，与“端口”概念相似。其实，端口说明中的量都是信号，由于是默认，因此不加 SIGNAL 标识。信号是全局量，通常在构造体、程序包和实体中说明，信号不能在进程中说明(但可以在进程中使用)。信号说明的语法格式如下：

SIGNAL 信号名{，信号名}：数据类型 [：=初始值]；

例如：

```
SIGNAL sysclk: STD_LOGIC;
```

在程序中，信号的赋值采用“<=”赋值符，而不是像变量赋值时用“:=”符，语法格式如下：

目标信号名<=表达式；

赋值语句中的表达式必须与目标信号具有相同的数据类型。信号一般包括 I/O 引脚信号以及 IC 内部缓冲信号，有硬件电路与之对应，因此信号之间的传递（赋值）有实际的附加延时。例如，s1 和 s2 都是信号，且 s2 的值经 10ns 延时以后才被赋值 s1。此时信号传送语句可书写如下：

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

可以想象出来，硬件中的连线总是同时工作的，对应于信号，它也是同时在各个模块中流动，这就是硬件电路的并发性。硬件设计语言 HDL 体现了实际电路中信号“同时”流动的这种基本特性。

4）信号和变量的区别

信号是硬件中连线的抽象描述，有实际的物理意义；变量在硬件中没有类似的对应关系，它们用于硬件特性的高层次建模所需的计算中，用作进程中暂存数据的单元。变量是一个局部量，只能用于进程或子程序中；信号是一个全局量，它可以用来进行进程之间的通信。

因为电流流动有时间延迟，信号赋值因此至少有 δ 延迟（如果不指定延迟，则延迟接近于 0），而变量因为不是实际物理量，因此变量的赋值就不允许有延迟。进程（PROCESS）语句只对信号敏感而不对变量敏感。信号和变量值的赋值不仅形式不同，而且其操作过程也不相同。在变量的赋值语句中，变量的赋值符为“:=”，该语句一旦被执行，其值立即被赋予变量。在执行下一条语句时，该变量的值就为上一句新赋的值。信号赋值语句采用“<=”赋值符，该语句即使被执行也不会使信号立即发生赋值。下一条语句执行时，仍使用原来的信号值。由于信号赋值语句是同时进行处理的，因此，实际赋值过程和赋值语句的处理是分开进行的。

信号赋值可以出现在进程中，也可以直接出现在结构体中，但它们的运行含义不同：前者属于顺序信号赋值，此时的赋值操作要视进程是否已被启动；后者属并行信号赋值，其赋值操作是各自独立并行发生的。

3. 数据类型

VHDL 语言是一种强数据类型语言，任一常量、信号、变量、函数和参数在声明时必须声明类型，使用时必须保持数据类型的一致性，不同类型之间的数据不能直接赋值，即使数据类型相同，而位长不同时也不能直接赋值，这种特性可以帮助设计者在设计前期发现错误。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必要手段。

1）标准数据类型

VHDL 提供了 10 种标准数据类型，如表 3.2 所示。

表 3.2　VHDL 的标准数据类型

数 据 类 型	含　义
整数	整数 32 位，取值范围：$-(2^{31}-1)\sim(2^{31}-1)$
实数	浮点数，取值范围：-1.0e+38～1.0e+38
位	逻辑 0 或 1
位矢量	位矢量，用双引号括起来的一组数据
布尔量	逻辑“真”或“假”，用 TRUE 和 FALSE 标记
字符	ASCII 字符
字符串	字符矢量
时间	时间单位 fs、ps、ns、μs、ms、sec、min、hr
自然数、正整数	整数的子集：自然数取值范围为 $0\sim(2^{31}-1)$；正整数是大于 0 的整数
错误等级	Note、warning、error、failure

上面的十种数据类型是 VHDL 标准的数据类型，在编程时可以直接引用。如果用户需使用其他的数据类型，则需要调用一些库，或者自定义一些需要的数据类型。

在 IEEE 库 STD_LOGIC_1164 程序包中有两个最常用的数据类型，分别是 STD_LOGIC 类型和 STD_LOGIC_VECTOR 类型。STD_LOGIC 类型的数据可以具有 9 种取值，其含义如下：

‘U’：初始值。

‘X’：不定态。

‘0’：强制 0。

‘1’：强制 1。

‘Z’：高阻态。

‘W’：弱信号不定态。

‘L’：弱信号 0。

‘H’：弱信号 1。

‘_’：不可能情况(可忽略值)。

其中，‘X’方便了系统仿真，‘Z’方便了双向总线的描述。

STD_LOGIC_VECTOR 是由 STD_LOGIC 构成的数组。定义如下：

```
TYPE STD_LOGIC_VECTOR IS array(natural range<>) OF STD_LOGIC;
```

在赋值的时候也需要保证相同数据类型，并且数据宽度相同。

2) 用户自定义数据类型

为了用户使用方便，VHDL 允许用户自己定义数据类型，其定义语法结构如下：

```
TYPE  数据类型名{,数据类型名}  IS  [数据类型定义];
```

常用的用户自定义数据类型主要包括枚举类型、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。

3）数据类型的转换

由于 VHDL 是一种强类型语言，不同类型的数据对象必须经过类型转换，才能相互操作。类型转换的方法有类型标记法和函数转换法。

(1) 类型标记法。

对相互间非常关联的数据类型(如整型、浮点型)，可进行直接类型转换，一般只适用于标量类型。使用方法就是直接在要转换的对象前加上类型名称，例如：

```
VARIABLE a, b: REAL;
VARIABLE c, d: INTEGER;
⋮
a:=real(c);
d:=integer(b);
```

(2) 函数转换法。

通过调用类型转换函数，使相互操作的数据对象的类型一致，从而完成相互操作，用户可以根据需要自己编写一些函数转换数据类型。VHDL 中也提供了多种转换函数，帮助转换一些常用类型的数据，以实现正确的赋值操作，在使用时必须首先打开库和相应的程序包。常用的类型转换函数如下。

① CONV_INTEGER ()。

将 STD_LOGIC_VECTOR 类型转换成 INTEGER 类型，存在于 STD_LOGIC_UNSIGNED。

② CONV_STD_LOGIC_VECTOR()。

将 INTEGER 类型、UNSIGNED 类型或 SIGNED 类型转换成 STD_LOGIC_VECTOR 类型，存在于 STD_LOGIC_ARITH。

③ TO_BIT ()。

将 STD_LOGIC 类型转换成 BIT 类型，存在于 STD_LOGIC_1164。

④ TO_BIT_VECTOR()。

将 STD_LOGIC_VECTOR 类型转换 BIT_VECTOR 类型，存在于 STD_LOGIC_1164。

⑤ TO_STD_LOGIC()。

将 BIT 类型转换成 STD_LOGIC 类型，存在于 STD_LOGIC_1164。

⑥ TO_STD_LOGIC_VECTOR()。

将 BIT_VECTOR 类型转换成 STD_LOGIC_VECTOR 类型，存在于 STD_LOGIC_1164。

4. 运算符与操作符

在 VHDL 语言中共有 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)。操作数的类型应该和操作符所要求的类型相一致。

1）逻辑运算符

在 VHDL 语言中逻辑运算符共有 6 种，包括一元逻辑运算符和二元逻辑运算符。一元逻辑运算符包括 NOT。二元逻辑运算符包括 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。必须注意，