目　录

第1章 结构化设计概念

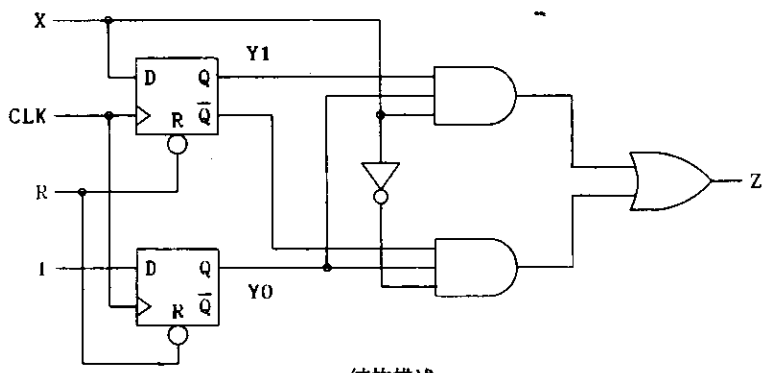
这一章给出与设计过程相关的基本定义。很有必要现在就来介绍它们，这对于解释其他概念是必需的。读者需要仔细研究这些定义才能理解后面将要介绍的内容。在后面的学习中再回过头来看看这一章也是很有益处的，因为这些定义的完整含义只有通过使用和实例才会变得更加明确。

1.1 抽象层次

本节介绍数字电路设计者所使用的抽象层次概念。抽象层次可以在以下两个域中表示：

- **结构域**：在结构域中，一个部件通过一些基本部件的互连来描述。
- **行为域**：在行为域中，一个部件通过定义它的输入/输出响应来描述。

图1-1给出了检测输入X的连续两个或更多1（或0）的逻辑电路的结构描述和行为描述。结构描述是指门和触发器等基本部件的互连，行为描述则是使用硬件描述语言（HDL）做文字性的表述。



结构描述

```
entity TWO_CONSECUTIVE is
  port(CLK,R,X: in BIT;Z: out BIT);
end TWO_CONSECUTIVE;

architecture DATAFLOW of TWO_CONSECUTIVE is
  signal Y1,Y0: BIT;
begin
  STATE: block((CLK = '1'and not CLK'STABLE) or R = '0')
  begin
    Y1 <= guarded '0' when R = '0' else X;
    Y0 <= guarded '0' when R = '0' else '1';
  end block STATE;
  Z <= Y0 and ((not Y1 and not X) or (Y1 and X));
end DATAFLOW;
```

行为描述

图1-1 一个采样电路的结构描述和行为描述

抽象层次定义如下：

抽象层次：一系列相关的表示层次，允许以不同的细节程度来描述一个系统。

一个典型的抽象层次如图1-2所示，层次中的第i层可以转换成第i+1层，细节程度一般随层次的下移而单调增加。

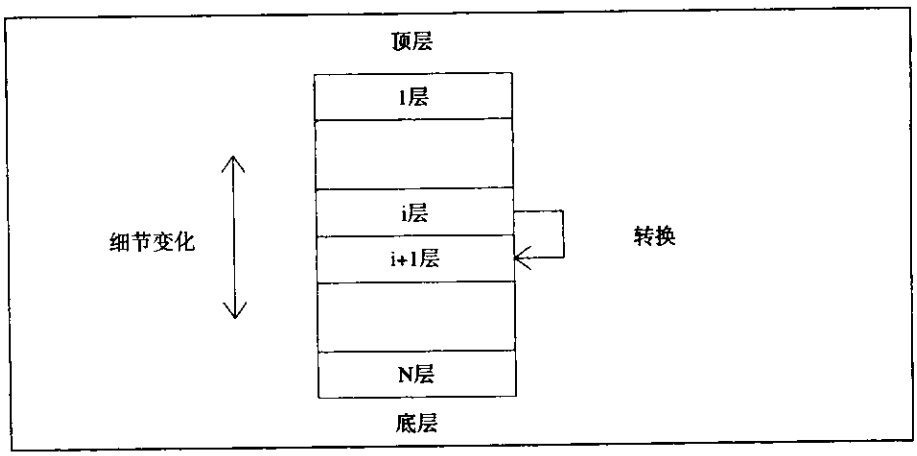


图1-2 抽象层次

本书使用的设计层次有6级：硅片级、电路级、门级、寄存器级、芯片级和系统级。表1-1举例说明了这种分层结构。硅片级是最低的层次，而系统级是最高的层次。一个设计可以用任何层次来表示。当设计从上而下进行时，该设计就逐步接近物理实现，在表示上就更少一些抽象。因此，表示一个设计所需的细节会随着它在层次中的下降而增加。在本书中，我们总是强调：在一个具体层次的设计中，保证它具有充分而不过多的细节是非常重要的。细节不充分会造成不精确的结果；相反，过多的细节则会使该层次的设计活动过多。

表1-1 设计的抽象层次

细节级别	行为域表示	结构域基本部件
系统	性能规格说明（自然语言，如英语）	计算机/磁盘/部件/雷达
芯片	算法	微处理器/RAM/ROM/串行端口/并行端口
寄存器	数据流	寄存器/ALU/计数器/多路复用器/ROM
门	布尔方程	与/或/异或/触发器
电路	微分方程	晶体管/电阻/电感/电容
版图/硅片	电子和空穴迁移方程	几何形状

表1-1显示了用结构化的基本部件及行为描述来表示每一层的这种层次结构的性质。结构化的基本部件通过互连可以在给定层次上形成结构化的模型。图1-3是各个层次上结构化的基本部件的例子，其行为表示则是在该层次上对设备的I/O响应的文本或图形描述。

在最低层，即硅片级，基本的单元是代表扩散区、多晶硅和硅片表面金属层的几何形状。这些图案的互连对设计者而言是对制造过程建立模型，这一级的行为描述则是描述在电子材料中电子和空穴迁移的物理公式。

在第2层，即电路级，其表示则是传统无源和有源电子电路元件的互连，这些电子电路元件包括电阻、电容、二极管和MOS晶体管等。元件的互连可以被用于以电压和电流的形式模

拟电路的行为，行为方面则通过微分方程的形式来表达。

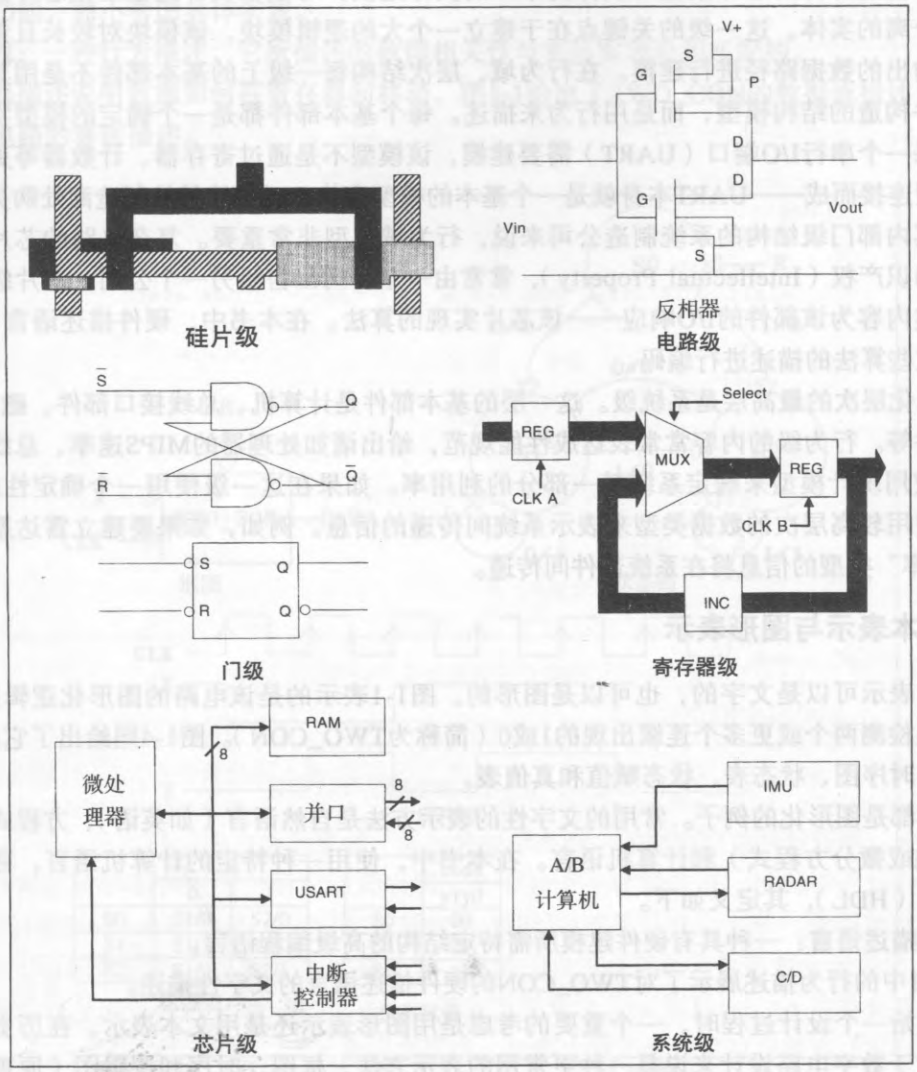


图1-3 结构域基本部件示例

第3层门级是传统数字器件的主要设计层次，其基本部件包括与/或/异或/反相操作门和不同类型的触发器。这些单元的互连构成组合和时序逻辑电路，布尔方程式定义了这一级的行为特征。

门级层次之上是寄存器级，这里的基本部件是寄存器、计数器、多路复用器和ALU。这些部件有时称为功能块，它们也对应着VLSI设计的宏，因此，寄存器级也称为功能级或宏级。虽然寄存器级的基本部件也可以用互连的门来表示，但是在进行这一层的设计时一般并不采用这种方法。寄存器级的基本部件采用真值表和状态表来表示，因此这两种形式可以用来进行这一层次的行为描述，并称之为*数据流*，也就是说它们反映在实际实现中数据的实际分布方式。在本书中，将会讲解这些数据流描述是如何在硬件描述语言（HDL）中实现的。

寄存器级之上是芯片级。在这一层次中，基本部件是微处理器、存储器、串口、并口和

中断控制器等部件。虽然芯片的边界一般就是模型的边界，但其他方案也是可能的。例如，组成一个功能部件的芯片集也可以作为单一的实体建模，可以在设计过程中把芯片的各部分建模为分离的实体。这一级的关键点在于建立一个大的逻辑模块，该模块对较长且集中的从输入到输出的数据路径进行建模。在行为域，层次结构每一级上的基本部件不是用其他更基本的部件构造的结构模型，而是用行为来描述。每个基本部件都是一个确定的模型实体。因此，如果一个串行I/O端口（UART）需要建模，该模型不是通过寄存器、计数器等更简单的功能模型连接而成——UART本身就是一个基本的模型实体。对于从其他制造商处购买芯片而不了解其内部门级结构的系统制造公司来说，行为域模型非常重要。复杂电路的芯片级模型被当作知识产权（Intellectual Property），常常由一个公司出售给另一个公司。芯片级模型的行为描述内容为该部件的I/O响应——该芯片实现的算法。在本书中，硬件描述语言（HDL）用来对这些算法的描述进行编码。

结构化层次的最高层是系统级。这一层的基本部件是计算机、总线接口部件、磁盘部件、雷达部件等。行为级的内容常常表达成性能规范，给出诸如处理器的MIPS速率、总线的带宽等，或使用统计模型来确定系统某一部分的利用率。如果在这一级使用一个确定性的模型，则它会使用较高层次的数据类型来表示系统间传递的信息。例如，如果要建立雷达系统的模型，“频率”类型的信息将在系统部件间传递。

1.2　文本表示与图形表示

设计表示可以是文字的，也可以是图形的。图1-1表示的是该电路的图形化逻辑原理图，其功能是检测两个或更多个连续出现的1或0（简称为TWO_CON）。图1-4则给出了它的框图、状态图、时序图、状态表、状态赋值和真值表。

这些都是图形化的例子。常用的文字性的表示方法是自然语言（如英语）、方程式（如布尔方程式或微分方程式）和计算机语言。在本书中，使用一种特定的计算机语言，称为*硬件描述语言*（HDL），其定义如下。

硬件描述语言：一种具有硬件建模所需特定结构的高级编程语言。

图1-1中的行为描述展示了对TWO_CON的硬件描述语言的文字性描述。

在开始一个设计过程时，一个重要的考虑是用图形表示还是用文本表示。在历史上，图形表示对于数字电路设计来说是一种更常用的表示方法，框图、时序和逻辑图（原理图）是基本的表示形式。然而，随着HDL的出现，文字性描述已经成为了主流。通过考查图1-1、图1-3、图1-4和表1-1，不难看出：结构描述基本上都是图形性的，而行为描述则是文字性的。这种分类方案的一些例外情况包括状态表、状态图和时序图，它们是图形性的，但是用来表示行为。至于应该使用图形还是文字这一基本问题，有人做了下述概括：*文字能更好地表达复杂行为，图形则更适合于阐明相互关系*。过度依赖于文字或图形都会造成观察上的损失，正如俗话说的：“只见树木，不见森林”。因此，在实际的设计系统中应权衡使用文字和图形——这也是本书所采用的方法。

1.3　行为描述的种类

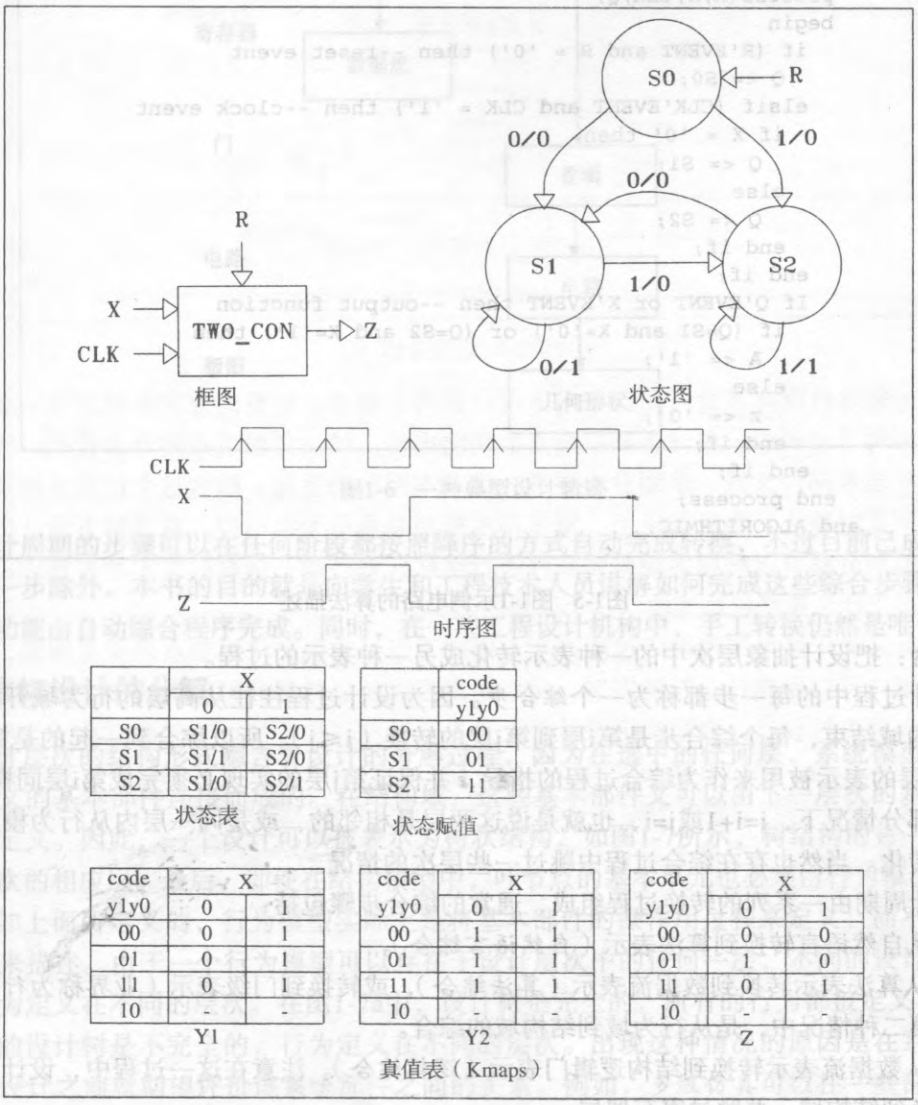
在HDL语言中的行为描述通常分为两类：*算法*和*数据流*。

算法：一种行为描述，其中定义I/O响应的过程没有隐含任何特定的物理实现。

因此，算法描述仅仅是一些写就的过程或程序，用于模拟器件的行为。以检验它是否执行正确的功能，而不考虑具体实现。

数据流：一种行为描述，它所描述的数据相关性与实际实现是相匹配的。

数据流描述说明数据如何在寄存器间移动。图1-1给出了TWO_CON的数据流描述，图1-5给出了该电路的算法描述。



	X	
	0	1
S0	S1/0	S2/0
S1	S1/1	S2/0
S2	S1/0	S2/1

状态表

	code
	y1y0
S0	00
S1	01
S2	11

状态赋值

code	X	
y1y0	0	1
00	0	1
01	0	1
11	0	1
10	-	-

Y1

code	X	
y1y0	0	1
00	1	1
01	1	1
11	1	1
10	-	-

Y2

code	X	
y1y0	0	1
00	0	0
01	1	0
11	0	1
10	-	-

Z

真值表（Kmaps）

图1-4 逻辑电路的图形表示

1.4 设计过程

我们将在本教材中描述一种结构化的设计过程，先从下面的一些定义开始。

设计：从系统的一种表示到另一种表示的一系列转化，直到最终的表示能被生产出来。

设计的途径涉及到综合（synthesis），它在字典上的定义为："把抽象的实体结合成单个或

统一的实体。”因此，综合就是把某些东西结合到一起，从我们的语境出发，可以有一个更特定的定义。

```
Architecture ALGORITHMIC of TWO_CONSECUTIVE is
  type STATE is (S0,S1,S2);
  signal Q: STATE := S0;
begin
  process(R,X,CLK,Q)
  begin
    if (R'EVENT and R = '0') then --reset event
      Q <= S0;
    elsif (CLK'EVENT and CLK = '1') then --clock event
      if X = '0' then
        Q <= S1;
      else
        Q <= S2;
      end if;
    end if;
    If Q'EVENT or X'EVENT then --output function
      if (Q=S1 and X='0') or (Q=S2 and X='1') then
        A <= '1';
      else
        z <= '0';
      end if;
    end if;
  end process;
and ALGORITHMIC;
```

图1-5 图1-1示例电路的算法描述

综合：把设计抽象层次中的一种表示转化成另一种表示的过程。

设计过程中的每一步都称为一个综合步。因为设计过程往往从高层的行为域开始，以低层的结构域结束，每个综合步是第i层到第j层的转换（$i \leqslant j$）。所以综合到一起的是第j层的表示，第i层的表示被用来作为综合过程的指导，并保证第j层的实现必须完成第i层同样的功能。在绝大部分情况下，j=i+1或j=i，也就是说这两层是相邻的，或是同一层内从行为模型到结构模型的转化。当然也存在综合过程中跳过一些层次的情况。

设计周期由一系列的转换过程组成，通常的综合步骤包括：

1) 从自然语言转换到算法表示（自然语言综合）。

2) 从算法表示转换到数据流表示（算法综合），或转换到门级表示（业界称为行为综合）。注意在第二种情况中，是从行为域到结构域的综合。

3) 从数据流表示转换到结构逻辑门表示（逻辑综合）。注意在这一过程中，设计者也从行为域转换到结构域，并跳过寄存器层。

4) 从逻辑门表示转换到版图表示（版图综合）。在这一综合步骤中，电路级被跳过，至此就完成了综合过程，因为有了版图信息就可以把芯片生产出来了。

完整的设计周期有时也称为设计综合。

图1-6给出了一个典型的经过各设计层次的设计轨迹。这个设计轨迹开始于行为域，并向下经过系统和芯片级再到寄存器级。在寄存器级，该级的行为化的数据流表示被转换成结构化的门级描述，门级描述又被转换到电路级，或直接到版图或硅片级。

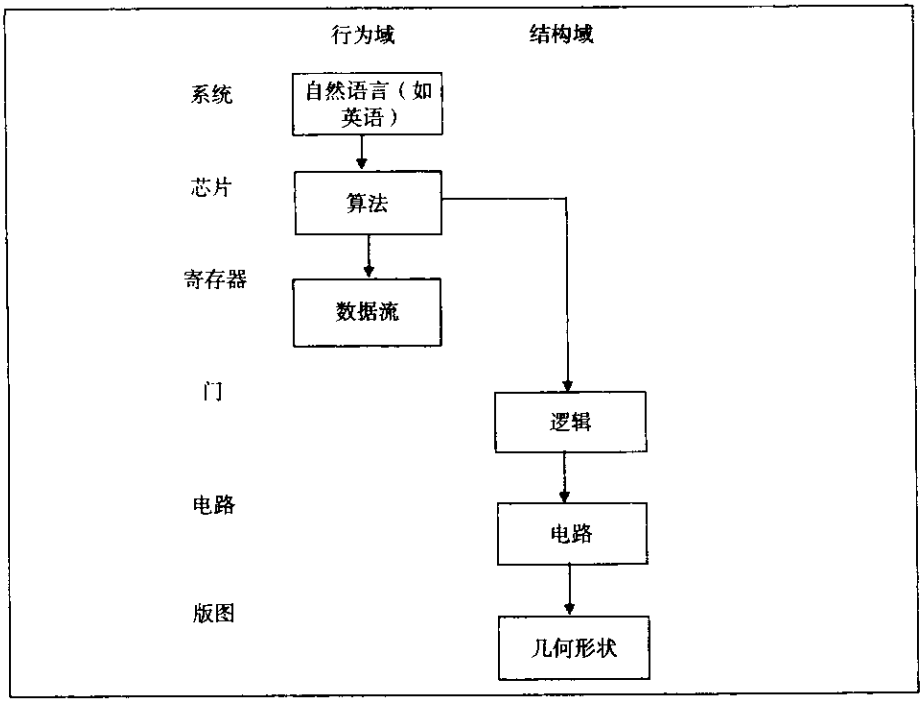


图1-6 一种典型设计轨迹

设计周期的步骤可以在任何阶段都按照降序的方式自动完成转换，不过目前已成为研究热点的第一步除外。本书的目的就是向学生和工程技术人员讲解如何完成这些综合步骤，因而了解哪些功能由自动综合程序完成。同时，在一些工程设计机构中，手工转换仍然是唯一的途径。

1.5 结构设计的分解

设计层次的结构形式隐含着设计的分解过程，因为在选中的任何层，系统模型都是由为该层定义的基本部件连接而成的。在结构域，这些基本部件又可以由下一层次的基本部件的连接来定义。因此，一个设计可以被表示为树状结构，如图1-7所示，树结构的每一层都对应抽象层次的相应层。最后，即使在结构模型中，叶节点的基本单元也必须由行为模型所表示。

正如上面所定义的，行为模型实际上是将基本部件的操作用过程来定义，而不是用另外的部件来描述。由于一个行为模型可以存在于设计层次中的任何一层，不同的设计部分也可以将行为定义在不同的层次。在图1-7a中，设计树是完全的，所有的行为都被定义在同一层。图1-7b的设计树是不完全的，行为定义在不同的层次。出现这种情况的原因是在系统部件尚未完全设计之前就期望评价该系统部件之间的关系。例如，多级仿真可以在一些部件已经设计到门级结构，而另一些部件只达到系统级时对系统进行评价。因此，在门级进行仿真时并不要求所有的系统部件都被定义在门级。系统测试时也可以采用对不同级别的部件进行行为仿真的方法。实际上，将一个大型系统的所有部件都安排在门级上进行仿真是不现实的。如果对它使用普通的门级仿真程序，则可能会需要数月的时间来完成该系统的仿真。人们常常在每次仿真时，针对不同的门级仿真部件，采用多级仿真的方法来代替一般的方法。其他部件使用系统级模型。由于系统级仿真的高效性，这种仿真所花的时间要少得多。进行几百次这种只需要几个小时的仿真就可以完成一般方法下需要数月时间才能完成的大型系统的仿真。

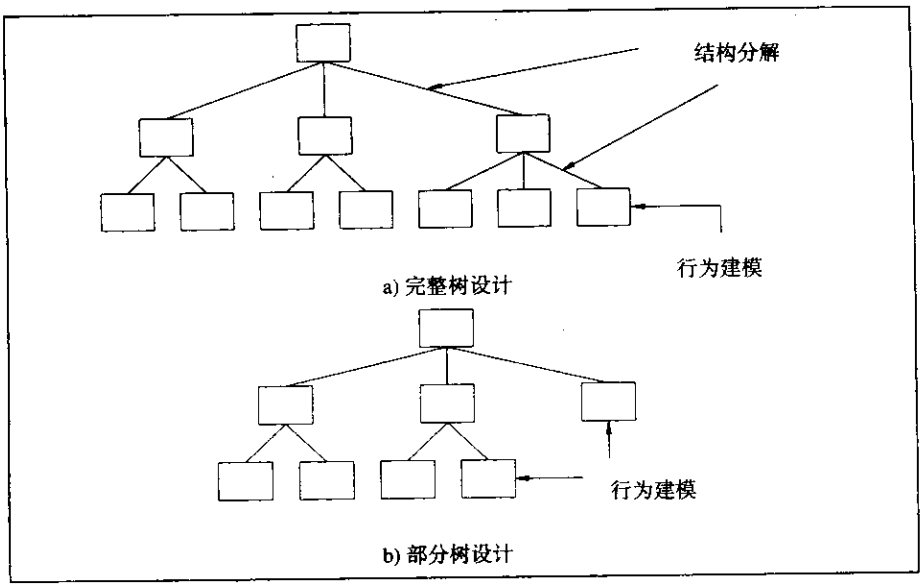


图1-7 结构分解

设计树包括自顶向下设计和自底向上设计这两个概念。这里的顶表示树的根，底表示树的叶。自顶向下设计中，设计者从根节点的功能出发，然后将根节点划分为较低一级的基本单元，这些基本部件又进一步划分为更低一级基本部件的互连。继续这个过程直到设计到达所有的叶节点时停止。在叶节点上，所使用的模型总是行为级模型。自顶向下设计的关键在于每一级的划分都可以根据客观限制条件进行优化，如开销、速度以及芯片面积等。划分过程并不受到“什么是可以达到的”这个问题的限制。

自底向上设计这个词在设计过程中有点用词不当，它仍然从根节点的定义出发，但划分的条件受到“可以做到什么”的限制。树中距离根部较远的部分可能会由于在其他工程中必须被使用而首先被设计。自顶向下设计看上去是一种理想的方法，它的缺点在于它所产生的部件并不是标准的，即它可能会增加设计的开销。而自底向上设计更加经济，但是它也同自顶向下设计一样会遇到客观行为限制。在大多数实际设计中，采用的是这两种方法的混合。

有关层次的最后一个概念是设计窗口，它指的是设计树中设计者所使用的层次的范围。VLSI芯片设计者的窗口包括硅片级、电路级、门级、寄存器级及芯片级。另一方面，计算机系统的设计者通常所关心的窗口只包括门、寄存器、芯片和系统级。因此，设计者所选择的窗口要适合自己的设计活动及工作所处的抽象级别，这样才可以只提供必需的信息，而不会卷入不必要的细节，用于设计的CAD系统应支持不同层次窗口之间的切换。

1.6 数字设计空间

在讨论自定向下设计时，曾指出每一步的划分都需要满足某些客观限制。这些限制条件是设计过程的关键因素。这些因素可以被看作是空间中的维数，数字设计空间中的一些有用的维包括速度、芯片面积和开销。图1-8给出了这样一个设计空间。不同的设计在空间中有不同的轨迹，设计者在各种设计中进行折衷。例如，如果设计者希望提高速度，则它可能要扩大芯片面积并且增加设计开销。图1-9给出了一个具体的例子。电路A和电路B实现同样的逻辑功能。电路A比电路B使用了更少的门（芯片面积），但速度要慢一些。电路B比电路A的速

度要快，但需要更多的门（芯片面积）。因此，设计者需要在速度和芯片面积之间进行折衷。

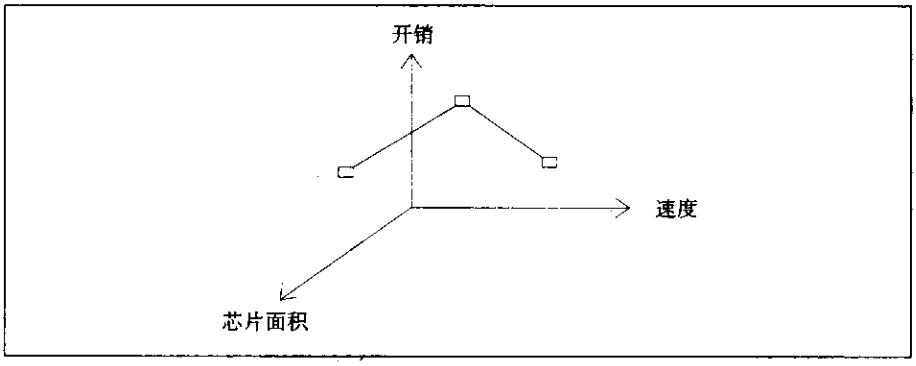


图1-8 一种典型设计空间

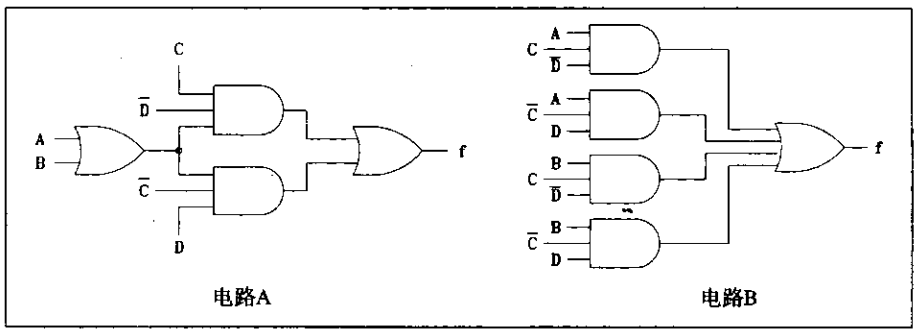


图1-9 一个设计空间折衷的例子

习题

1.1 画出一个由全加器互连而构成的16位行波进位加法器的结构模型，并用C、C++或JAVA语言写出该16位加法器的行为描述。

1.2 用AND、OR和INVERT基本部件画出全加器的门级图，参考有关电子学和VLSI设计的书，生成由晶体管、电阻、电容构成的全加器的CMOS电路，按如下内容比较这两种电路：

a) 统计这两种表示方法所需要的基本部件和连线的数量。

b) 在同一台计算机上用逻辑仿真程序和SPICE来仿真这两个电路，对这两种电路都使用所有可能的8种输入模式。比较它们的仿真时间。

1.3 研究数字系统在不同抽象层次的表示。使用Morris Mano所著的《计算机系统结构》（第2版，Prentice Hall 1982年出版）一书中给出的一台小型计算机的例子，记录该系统在以下层次上的描述。

a) 为该系统产生一页纸的系统级描述，要求没有空行，并且使用自然语言。

b) 用你最喜欢用的图形包软件画出该系统的芯片级框图，使用Mono机、RAM存储器及I/O逻辑部分作为基本单元。

c) 画出Mono机完整的寄存器级图。

d) 使用门和J-K触发器，画出程序计数器完整的门级图。

e) 从程序计数器选择一个门，画出它的CMOS电路结构。

f) 画出在上一步所选择的门的版图。

上面6步产生的表示中，哪些是结构描述？哪些是行为描述？

1.4 图1-10给出了一个简单的RC电路的结构模型。写出包括电流$i(t)$的一种行为描述，并绘出$i(t)$的坐标图。

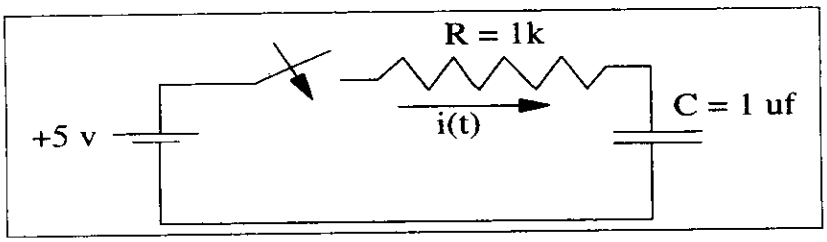


图1-10 RC电路

1.5 下面给出的是一个门级电路的行为描述：

$$F = AB \text{ or } \overline{CD} \text{ or } \overline{EF} \tag{1-1}$$

画出实现这个功能的两种不同的结构门级模型。

1.6 N. Wirth所著的《算法+数据结构=程序》（Prentice Hall 1976年出版）一书中对一个虚构的家族关系有如下描述：

我和一位寡妇（称为W）结了婚，她有一位成年的女儿（称为D）。我的父亲（F）经常到我们家做客，他爱上了我的继女并同她结了婚，于是我的父亲成为我的女婿，并且我的继女成为了我的母亲。几个月以后，我的妻子生了一个儿子（S1），他是我父亲的小舅子，也是我的舅舅。我父亲的妻子，即我的继女也有了一个儿子（S2）。问题：我是否为自己的外祖父？

画出这种关系的图形表示，节点代表人的名字，弧线表示人与人之间的关系。如"……的父亲"。哪种表示更容易理解？你能回答这一问题吗？

1.7 图1-11是Morns Mano的《计算机系统结构》（第2版，Prentice Hall 1982年出版）中一个计数器电路的逻辑图。用C或C++写出它的行为描述。哪一种描述使得电路的功能更易于理解？

1.8 图1-12给出了两个设备之间的系统接口：发送设备（设备1）和接收设备（设备2）。这两个设备之间的接口包括一个数据DATA线和3个控制信号READY、VAVID、ACCEPT。这两个异步设备间的通信协议如下：

a) 设备2对READY置位。

b) 设备1检测到READY变高后，将数据发送到DATA线上并将VALID置位。

c) 设备2检测到VALID变高后，接收数据并对RESET复位，对ACCEPT置位。

画出DATA、READY、VALID和ACCEPT的时序图，使用箭头表示信号间的触发关系。

1.9 给出如下的算法描述：

```
for I=1 to 3 loop
  A(I) = B(I) + C(I)
  D(I) = E(I) * A(I)
end for;
```

画出它的数据流程图，其中节点表示操作(+、*)，弧线表示输入或计算得到的值。

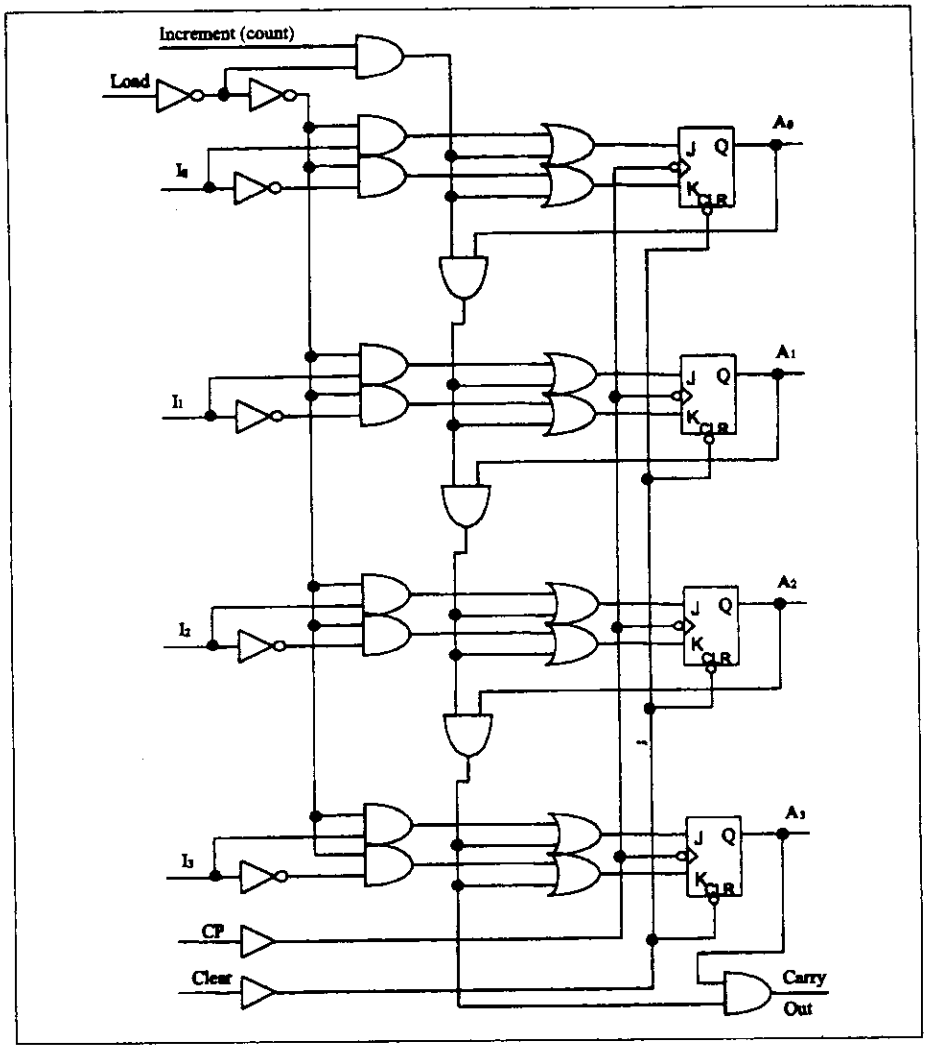


图1-11 计数器电路

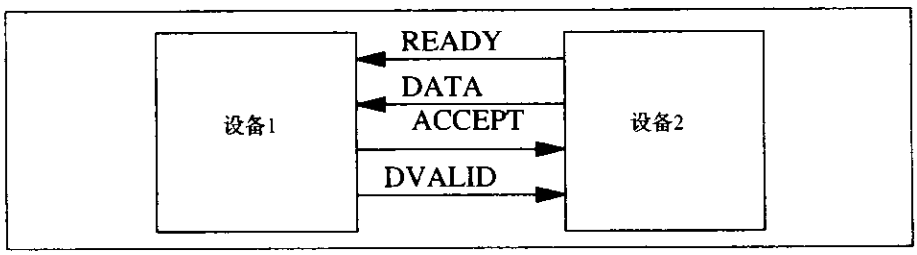


图1-12 接口协议

1.10 根据自己的经验，叙述一个你所熟悉的设计综合过程，识别出不同的综合步骤、抽象层次，并说明该表示属于行为域还是结构域。画出类似于图1-6的设计途径。

1.11 设计分解意味着使用哪种类型的VHDL模型？

1.12 比较自顶向下和自底而上两种设计方法在以下两方面的不同：a)开销；b)设计是否最优。

1.13 解释自顶向下和自底向上设计的区别。这两种设计的优、缺点各是什么？

1.14 在微处理器系统设计和用ASIC进行设计时，采用自顶向下或自底向上方法的特点是什么？为什么？

1.15 解释如何把自顶向下和自底向上的设计思想应用到软件设计中。

1.16 以16位加法器的设计为例来解释设计空间的折衷。

a) 设计一个16位行波进位加法器。

b) 设计一个16位先行进位加法器。

c) 比较这两种加法器实现所需门的数量及延时。

d) 给出这两种加法器实现所需的门数及延时的通用表达式，这个表达是加法器输入字宽n的一个函数。

第2章 设计工具

本章介绍设计过程中使用的CAD工具，这里所指的工具有特定的含义：

CAD工具：用来协助进行特定设计或使设计过程自动化的软件程序。

2.1 CAD工具分类

在过去的30年里，开发出了大量用于辅助设计的CAD工具。制造商对这些工具功能的分类标准并不统一。在本书中采用图2-1的分类方式，该分类方式将工具划分为类和子类，在图中还标记了工具所处的抽象层次。

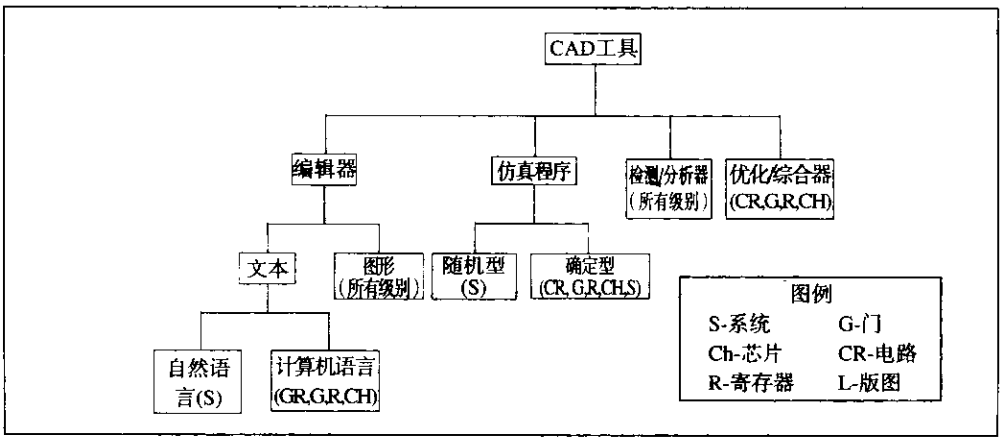


图2-1 CAD工具分类

2.1.1 编辑器

编辑器可以是图形的或文本的，文本编辑器可以用来编辑系统级规格说明的自然语言，或在电路级到芯片级编辑硬件描述语言。例如，SPICE就是一种专用于电路级描述的硬件描述语言。在本书中，VHDL可用于门级、寄存器级和芯片级的描述。

图形编辑器可以用于所有层次。在版图级，图形编辑器可通过创建几何图形的方式来表示硅片布局图。在所有的更高层次中，图形编辑器可用来创建框图或原理图。实际上这个过程有时称为“原理图获取”。原理图表示了一种结构模型。有两种解释方式：1) 网表，表示各个系统部件之间的连线。2) 仿真模型，用来确定系统对输入的响应。

2.1.2 仿真程序

仿真程序可以是随机的或确定的。随机型仿真在系统级执行。例如，可以测试某个功能部件忙的时间所占的百分比。确定型仿真可以在硅片级以上的各级进行，使用的单位，如电

压、位、标记，取决于抽象的级别。第3章讨论数据类型时，将进一步讨论此问题。

2.1.3 检查程序和分析程序

检查程序和分析程序可用于各级。在版图级，*设计规则检查程序*用来保证设计的电路版图能可靠实现。在其他层次，用规则检查程序来检查是否违反了连接规则或扇出规则。*时序分析程序*用来查找逻辑电路或系统中的最长路径。无论硬件描述语言用于哪一级（CR，G，R，CH中的）（见图2-1），分析程序都被用来检查结构和语义上的错误。

2.1.4 优化程序和综合程序

优化程序和综合程序将设计转化为另一种表示形式，转化后的形式被认为在某方面有所改进。例如，在门级可以使用最小化程序来产生简化的布尔表达式。在寄存器级则可用优化程序来决定控制序列和数据通路的最佳组合。正如前一章所述，在各个层次之中，各种形式的综合程序将设计向接近实现的低级层次的表现形式转化。

2.1.5 CAD系统

本节的重点在于对CAD工具进行分类。在大多数情况下，*CAD系统*由厂商开发，包含一系列工具。例如：可以把文本编辑器、原理图编辑器、规则检查程序、分析程序和仿真程序包含于一个系统中，现在详细介绍一些重要的工具。

2.2 原理图编辑器

在设计过程中，有很多时候会创建原理图，这时可以通过原理图编辑器来实现。

原理图编辑器：用来创建和显示一系列互连的图形标记的编辑器。

图形标记和结构基本单元一一对应，而互连的电路则对应于一个结构模型，这种对应关系常常很强，因此在创建图形标记电路的同时也创建了一个仿真的模型。

典型的原理图获取系统包含以下特征：

1) 基本单元符号库，每个基本单元都对应一个仿真模型。基本单元库可以是本地的（native），或与ASIC库元件的标准部件系列（如TTL、CMOS、ECL等）对应，或对应于综合工具的宏。本地基本单元是最基本的逻辑元件，如AND（与门）和OR（或门），一般内置于仿真程序中。标准部件系列是用本地基本单元表示的。图2-2给出了带时钟输入的RS触发器及其在Viewlogic公司的Workview系统中的本地基本单元表示的符号。大多数原理图获取系统允许用户向现存库中添加新的符号和仿真模型。在本书中则通过编写VHDL源代码来创建仿真模型。

2) 图形窗口系统用来创建图形标记之间的互连。窗口系统一般组织成树的形式（见图2-3）。从主菜单（树根）开始，随着更加详细的操作的进行，进入树的低层窗口之中。图2-4a给出了在Workview系统中创建一个元件的菜单命令序列（树的一条路径）。菜单标记序列为CREATE、COMPONENT。然后根据提示键入元件名，系统将在屏幕上选择的点处放上一个与元件名相对应的库元件的图形标记。图2-4b显示了创建两个元件的结果，分别为一个三输入与非门和一个D触发器。通过选择菜单命令CREATE、NET，在源引脚和目标引脚之间用连线来形成部件的互连。图2-4c显示了与非门的输出与D触发器输入之间的连线。对外部输入输

出的连线与之类似，见图2-4。通过选择菜单标记序列CREATE、LABEL，可进行信号标记。可用菜单命令pan使某个点成为图形的中心，或通过zoom命令来放大或缩小图片。图2-4d展示了pan和zoom命令之后的与非门与D触发器，它通过菜单命令序列VIEW、IN来实现。菜单命令同样允许移动和拷贝图形元件。

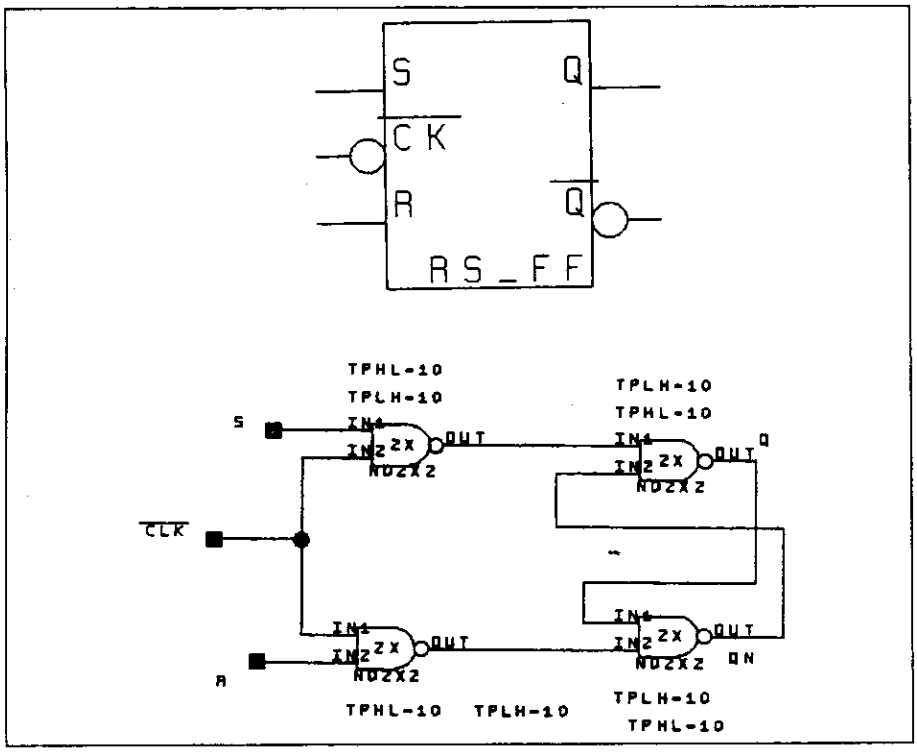


图2-2 RS触发器

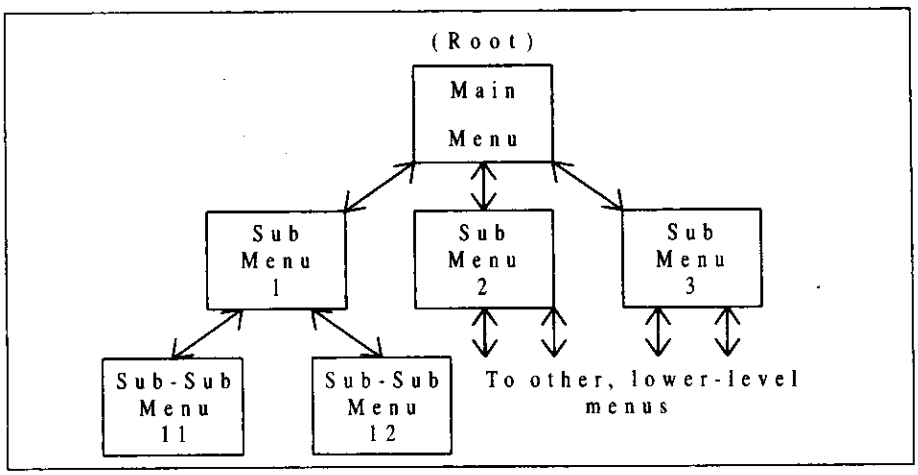
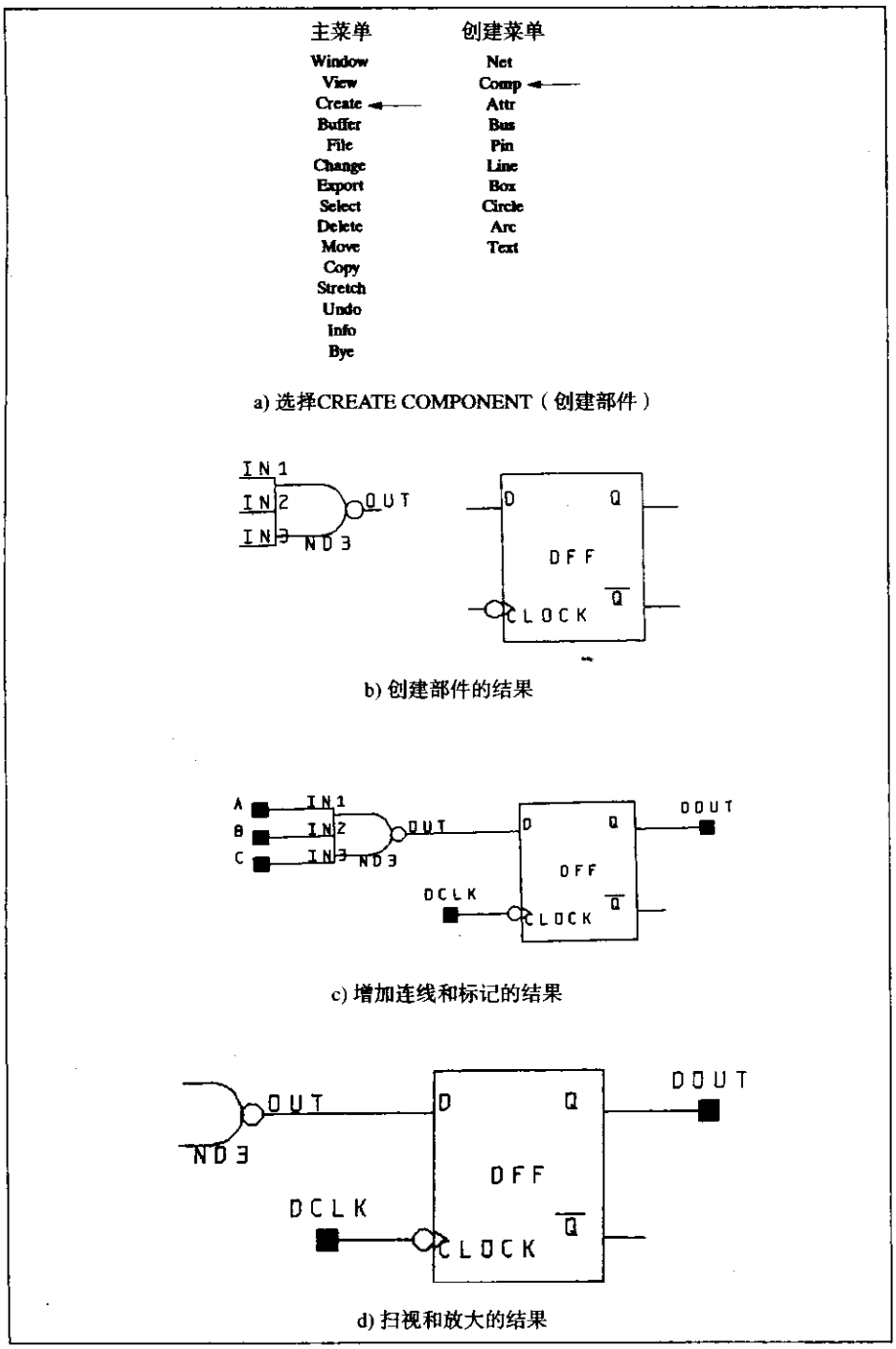


图2-3 菜单树的示例

3) 创建连线列表（Wirelist）的命令。连线列表有两种用途，首先可以用来创建实际的电路，同样也可以定义用于测试电路响应的仿真结构模型。这样，使用原理图获取工具之后一

般紧跟着对仿真程序的调用。



a) 选择CREATE COMPONENT（创建部件）

b) 创建部件的结果

c) 增加连线和标记的结果

d) 扫视和放大的结果

图2-4 用原理图编辑器进行的操作

2.3 仿真程序

仿真程序是开发数字系统的主要工具。

仿真程序：对系统输入激励的响应进行仿真的程序。

这里被建模的系统是数字逻辑单元的互连电路，我们的建模方法是将数字逻辑元件映射为一个或多个如下面定义的进程。

进程：对数字器件的功能和延时进行建模的计算实体。

图2-5给出了一个数字器件相应进程的图形表示，以及进程对应的VHDL源代码的例子。

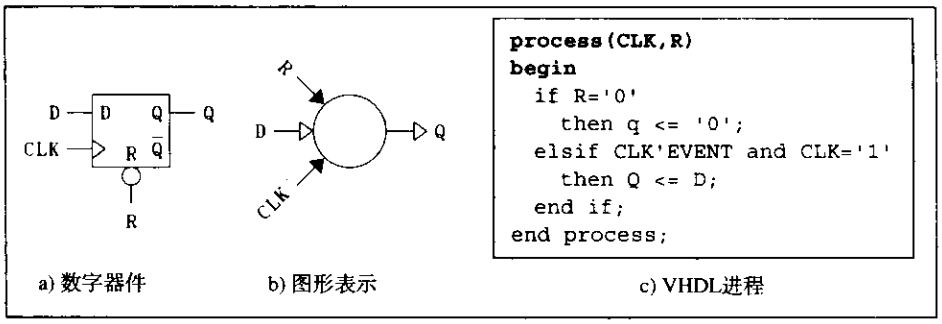


图2-5 器件与进程之间的对应关系

数字器件的互连可以通过进程的互连来仿真，见图2-6中所示。器件与进程的关系如下：

器件	进程
D1	P1
D2	P2, P3
D3, D4	P4

这说明：1) 有时单个的器件映射为单个的进程，2) 有时单个的器件映射为多个进程。3) 有时一系列器件映射为一个进程。数字器件网络的数字器件之间的连线在进程网络中用信号进行标记。连线与信号的对应关系如下：

连线	信号
A	SA
B	SB
C	SC
D	SD
E	SE
F	SF
G	SG
H	SH
I, J	*
*	SI, SJ

连线A到连线H对应于信号SA到SH。因为器件3和器件4对应于单个进程4，所以连线I和J没有对应的信号。信号SI和SJ没有对应的连线，这是因为器件2由进程2和3来代表。

器件在任意输入信号变化时计算新的输出，同样，进程在任意输入信号发生变化时被激活以计算新的输出。引起进程运行的信号称为触发信号（*triggering signal*）。在图形表示中，该类信号用实心箭头表示。图2-5b表示CLK和R是触发信号。有些信号不引起进程的执行，进

程仅对这些信号进行采样。这种信号用空心箭头表示。图2-5b中的信号D就是一个采样信号。

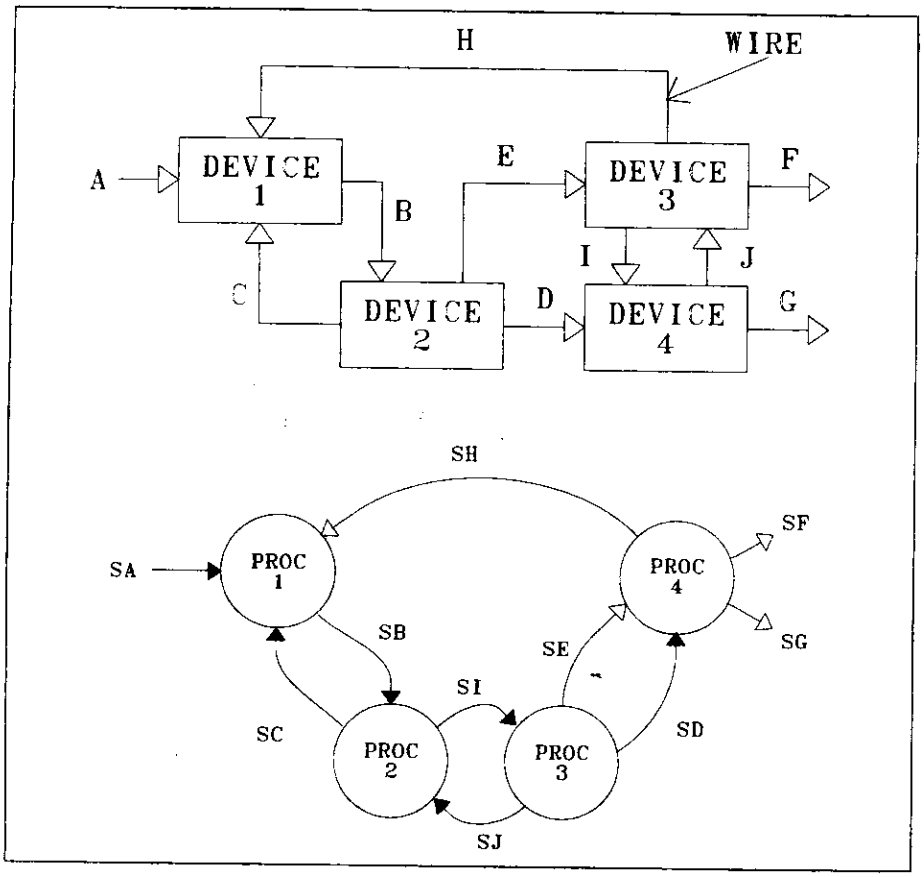


图2-6　用进程网络来对器件网络建模

在仿真的方法中，一个逻辑电路用一个进程网络模拟。对进程网络进行操作的仿真程序见图2-7。进程响应其输入上的信号事务（*signal transaction*）。这些信号事务保存在仿真程序的时间队列中。时间队列项采用形如（SN，V）的二元组表示，其中SN是信号名，V是信号在调度时的信号值。每个时间队列项称作一个信号事务。如果其中一个信号的值发生了变化，就称为发生了信号事件（*signal event*）。

上述例子的仿真过程如下：进程被编号为0到3。假设进程0代表原始输入（PI）。时间队列的初始项（A，1）表示进程0的输出信号A在时间0的输出应该为1。在这种情况下，一个信号跟踪程序确定信号A作为进程1的输入。假设信号A的值是新的（发生了信号事件），并且是触发输入，进程1将被激活。假设该进程计算出输出信号B的新值为1，由于内部传输延时了输出，信号B将在当前仿真时间延迟100ns之后变为逻辑1。时间队列中将插入新的一项。在时间0的所有变化都被处理之后，仿真时间向前前进到队列中下一项。队列的处理一直进行到时间100。这里项（B，1）被处理：信号B被置为逻辑1。同样，假设这是一个信号事件，信号跟踪程序确定在信号B上的事件将引发进程2和3的执行。进程2的执行产生信号C在50ns之后的新调度值，进程3的执行产生信号D在75ns之后的新调度值，假定信号C和D又引起进程网络中其他进程的执行，这里不再详述。仿真一直进行到时间队列为空或者达到某种外部控制时间

的限制。

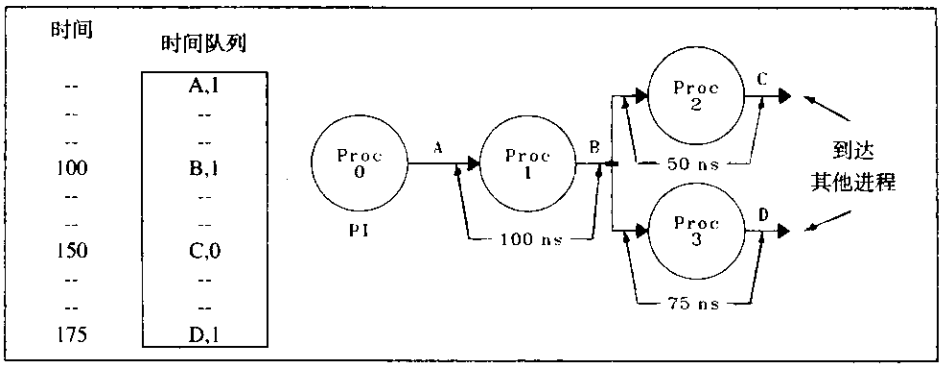


图2-7 仿真程序操作

2.3.1 仿真周期

一个模型的运行包括初始化阶段和对进程网络中进程的重复执行。每次重复称为一个*仿真周期*。在每个周期内，确定哪些信号发生了事件。如果在给定的信号上发生了信号事件，被该信号触发的进程将重新开始，并将作为该仿真周期的一部分而执行。因此，仿真周期包含以下各步：

1) 将仿真时间进行到时间队列的下一项，如果时间队列中没有新的项，则停止。

2) 对于所有在该时间内发生信号事件的信号，激活这些信号触发的进程。

3) 执行被触发的进程，如果有必要，调度时间队列的新项。转到第1步。

2.3.2 仿真程序组织

图2-8图示了仿真程序内核的组织结构，它由4部分组成：

1) 时间队列。

2) 时间队列处理器。

3) 信号跟踪器。

4) 进程执行器。

如前例所示，时间队列处理器完成对时间队列中项的插入和删除，并将信号事件提交信号跟踪器。信号跟踪器决定信号的扇出并且标记将要执行的进程。信号执行器控制所标志的进程的执行，将新的时间队列项传递给时间队列处理器，来进行新的调度。

2.3.3 语言调度机制

前面对仿真程序操作的讨论说明用来创建器件模型的语言需要一种调度机制。本书中使用VHDL，即VHSIC硬件描述语言。

在VHDL中，可以这样写：

```
X <= Y after 100 ns;
```

结果是X在当前时间的100ns（仿真时间）后，其值与Y相同，因而X的未来值已经安排好了。

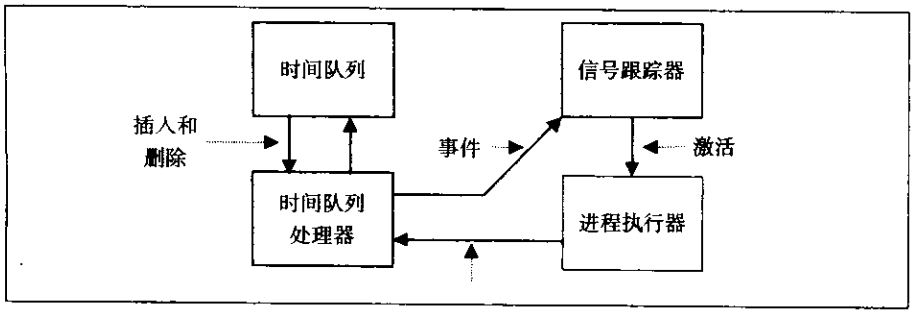


图2-8 仿真程序内核

2.3.4 仿真效率

当仿真大系统的模型时，仿真效率十分重要。对逻辑仿真，仿真效率（*simulation efficiency*）（E）定义为：

$$E = \frac{\text{实际逻辑时间}}{\text{主机 CPU 时间}}$$

实际逻辑时间指在真正的逻辑电路中，完成动作序列所需的时间。主机CPU时间指使用逻辑仿真程序在主机CPU上运行同一逻辑序列的时间。为了使这些数据更有意义，主机CPU的时间必须根据主机的速度进行归一。例如，仿真是运行于100MIP的机器上还是500MIP的机器上？

另一个衡量效率的方法是仿真程序单位时间内所能处理的事件个数。例如，门级仿真程序的效率通常以主机CPU每秒进行门计算的数目来计算。对于高层的模型，事件用高层活动的术语来定义。例如，在对微处理器进行建模时，可以用所建模的微处理器在主机CPU上每秒钟可以运行的时钟周期数来描述。另一种更高层次的衡量方法是测量被建模的微处理器在主机CPU1秒内执行的机器指令数，这种方法适用于在一种CPU上仿真另一种CPU的运行时的情况。

仿真效率由编程技术、所使用计算机的体系结构及建模的层次所决定。对于编程技术，这里指仿真程序是编译运行还是解释运行。如果是解释运行，系统的互连保存在表格里，各个器件的模型在运行时根据信号流的指示被调用。在这种情况下，仿真模型由解释器在运行时解释执行。如果是编译运行，则在仿真之前，整个模型被编译为机器代码，不需要运行时的解释。编译的仿真程序比解释执行的仿真程序要快，但是，如果模型编译成主机CPU的汇编语言，就会面临可移植性问题。然而，本书中讨论的VHDL仿真程序将仿真模型编译成C代码，这样就解决了可移植性问题。

前面对仿真程序的讨论都假设使用单CPU来运行仿真程序。最近，已经开发出使用并行构架的仿真程序引擎以加速门级仿真。例如，IBM的约克敦仿真程序引擎（Yorktown Simulation Engine，YSE）和Zycad的仿真加速器可以1000倍地加速门级仿真。

因而，编程技术和主机的结构可以极大地影响仿真效率。此外，影响仿真效率的另一个因素是仿真运行的抽象层次。例如，电路级仿真使用SPICE，尽管能够产生精确的结果，效率却不高。这是因为，在一个电路的仿真运行过程中，只有几百个电路节点可以被真实地仿

真。门级仿真的效率也一样不高。如果建立了精确的时序模型，据经验，仿真效率只是原来的10^{-7}。单位延时和零延时的仿真程序的在门级的效率要高一些。

芯片级仿真的效率可以达到门级仿真的100到10 000倍，建模时应该选择尽可能高的层次，这样既可以进行仿真，又可以得到足够的精度。

2.4 仿真系统

仿真程序只是*仿真系统*的一部分。图2-9表示了一个完全的仿真系统的例子。VHDL模型的源代码在文本编辑器里进行编辑，源代码提交给*分析程序*，由分析程序检查下述错误：

词法：检查源代码是否使用了正确的字符集。

语法：检查源代码是否遵守语法规则。例如，VHDL定义if 语句必须以end if结尾，否则将引起语法错误。

语义：任何语言对于语言中使用的各种结构都附有特定的含义，这种特定含义就称为*语义*。

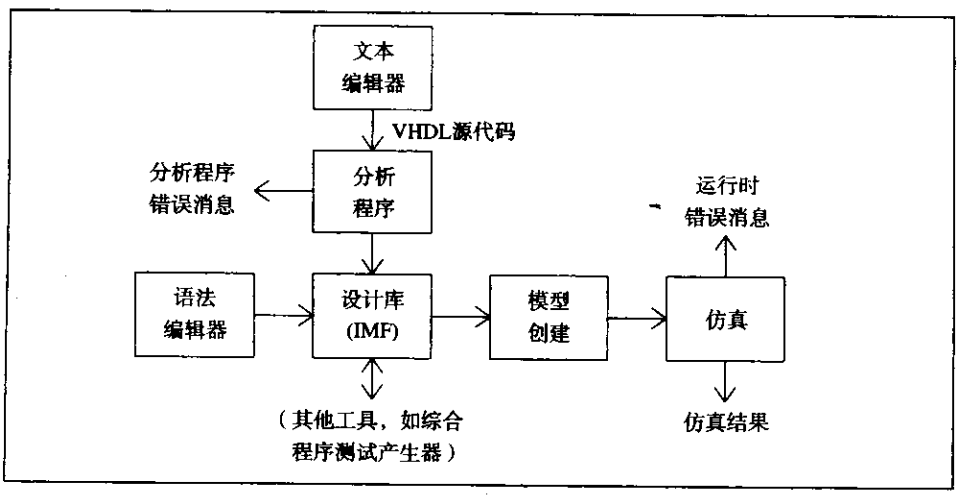


图2-9 仿真系统

作为例子，请看如下VHDL代码：

```
variable X: BIT ;
variable Y: INTEGER;            --X和Y具有不同的类型
---------------------------
---------------------------
X: = Y ;                        --语法正确，但语义错误
---------------------------
---------------------------
```

尽管语法是正确的，但因为X和Y是不同的数据类型，将Y的值赋给X将引起语义错误，它们的数据类型分别为BIT和INTEGER类型。由分析程序进行的语义检查是*静态的*，与仿真结果无关。

在模型的源代码通过分析之后，即被存放到*设计库*（*design library*）中，以备后继的仿真和其他设计活动使用。源代码采用一种*中间形式*（IMF）保存在设计库。IMF是一种提供对源代码的高效存储与访问的数据结构。

为准备实际的仿真，模型创建程序（*Model Builder*）将模型链接起来形成一个系统仿真模块。链接过程与软件系统进行的子程序链接相似。当模型从总体上做好了仿真的准备后，就称为细化求精。然后仿真程序对其进行仿真。这时可能发生运行时错误，称为动态语义错误。在下一节的模型调试中会介绍一个这样的例子。

图2-9给出了设计库中的另外两个接口。其中一个是原理图编辑器它可以产生针对VHDL的IMF代码，并保证词法、语法、语义的正确性。这样就可以直接插入到设计库中。通过另一个接口可以使其他工具（例如综合程序和产生测试的工具）也可以访问设计库。因为仿真程序、综合程序和产生测试的工具可能由不同的厂家开发，因而IMF格式的标准化很有必要。

2.5 仿真辅助工具

对于那些经常进行仿真的人来说，模型的创建、输入向量的产生、仿真模型的调试、仿真结果的解释是非常乏味的过程。因此，仿真公司开发了仿真辅助工具，使上述工作变得更容易。

2.5.1 模型准备

前面讨论的原理图编辑器对模型的创建大有帮助，它可以为数字系统创建结构模型。图2-4中就是这样的一个例子。一个结构模型由基本单元的互连组成，这些基本单元的行为级模型还需重新定义。行为级模型的创建可以使用一种叫做建模程序助理的交互工具而加以简化。这是一种可以让用户通过图形符号定义模型行为的工具。系统产生描述模型行为的VHDL源代码。图形符号类似于仿真程序那一节讨论的进程互连图（见图2-6），这里指的是进程模型图（PMG）。程序用PMG作为图形输入，交互地创建VHDL源代码。

系统的操作如下：

1) 进程模型图用图形输入，定义行为级模型的外壳。

2) 主要从进程基本部件库中选出可以构成特定应用的构件进程。

3) 需要一些文字输入，如指定延时和数据类型。

4) 结果就是模型的完整VHDL源代码描述。

在此期间可以查看代码，工具允许用户使用预定义的进程，或创建新的进程，并将其存入进程库中。

复杂器件的行为级模型的编码工作是一项劳动密集型工作。借助图形表示，菜单驱动的系统和可重用的代码，建模工具“建模程序助理”可以大大提高VHDL模型生成的效率，也使得行为级模型有良好的结构。本书第10章对此工具有详细的介绍。

2.5.2 模型测试向量的产生

要测试模型，就要产生输入测试向量。下面的VHDL代码检测第1章描述的连续‘0’、‘1’检测器模型。信号CLK、X和R是模型的三个输入。

```
CLK <='0','1'after 10 ns,'0'after 20 ns,
      '1'after 30 ns,'0'after 40ns,
      '1'after 50ns,'0'after 60 ns,
      '1'after 70 ns,'0'after 80 ns,
      '1'after 90 ns,'0'after 100 ns,
      '1'after 110 ns,'0'after 120 ns,
```

```
      '1'after 130 ns,'0'after 140 ns;
X <= '0','1'after 15 ns,'0' after 55 ns;
R <= '1','0' after 125 ns,'1' after 127 ns;
```

手工产生这些代码既乏味又容易出错。需要一个时序图处理器将模型输入的图形描述转化为相应的VHDL语句。第4章中在开发测试程序包时将介绍这种功能。

2.5.3 模型调试

与开发软件一样，在开发模型时，调试辅助工具很重要。调试器可以在分析和仿真时发现错误。

1. 文本调试

请看如下VHDL源代码。注意信号X的类型是INT_RANGE，包含从1到4的整数，这里5被赋值给X。

```
entity NUMS is
end NUMS;

architecture RINT of NUMS is
  type INT_RANGE is range 1 to 4;
signal X: INT_RANGE;
begin
  X <= 5;
end RINT;
```

在Synopsys仿真程序中对该文件进行分析，终端上将会打印如下出错信息：

```
  X <= 5;
       *
Warning: vhdlan,2 nums3.vhd(7) : Constraint error.
```

消息表明在源代码的第7行检测到了“constraint error”，而且，如果用户选择listing选项，将会产生如下文件：

```
-- Synopsys 1076 VHDL Analyzer Version 2.1c
--
--            Copyright (c) 1990 by Synopsys, Inc.
--                    ALL RIGHTS RESERVED
-- This program is proprietary and confidential information
-- of  Synopsys, Inc. and may be used and disclosed only as
-- authorized in a  license agreement  controlling such use
-- and disclosure.
--
--
-- Source File:  nums3.vhd
-- Thu Jan 30 12:55:17 1992

    1 entity NUMS is
    2 end NUMS;
    3 architecture RINT of NUMS is
    4  type INT_RANGE is range 1 to 4;
    5  signal X: INT_RANGE;
    6 begin
    7   X <= 5;
         *
Warning: vhdlan,2 nums3.vhd(7):
```

```
    Constraint error.
    8 end RINT;
    9
-- "nums3.vhd": errors: 0; warnings: 1.
```

为了说明在仿真期间调试运行时的错误，请看下例：

```
entity NUMS is
end NUMS;

architecture RINT of NUMS is
  type INT_RANGE is range 1 to 4;
  signal X: INT_RANGE:= 2;
  signal Y: INT_RANGE:= 3;
  signal Z: INT_RANGE;
begin
  Z <= X + Y;
end RINT;
```

当这段代码被执行时，5将被赋值给Z，同样超出了范围。在分析阶段发现不了这个错误，因为分析程序无法知道运算的结果。然而，在仿真过程中，Synopsys仿真程序检查出上述错误，并且打印出如下信息：

```
**Error: vhdlsim, 2:
    Constraint error.
      nums2.vhd(9) :
```

消息表明，仿真过程中，在文件“nums2.vhd”的第9行，执行语句Z<=X+Y时；发现了一个“constraint error”。

2. 图形调试器

图形调试器同样有用。它允许单步执行及设置断点，以便跟踪程序的运行。图2-10给出了的Synopsys调试器执行‘1’计数模型ALGORITMIC的一个窗口。该窗口和波形一起显示一起用来检验模型的正确性。

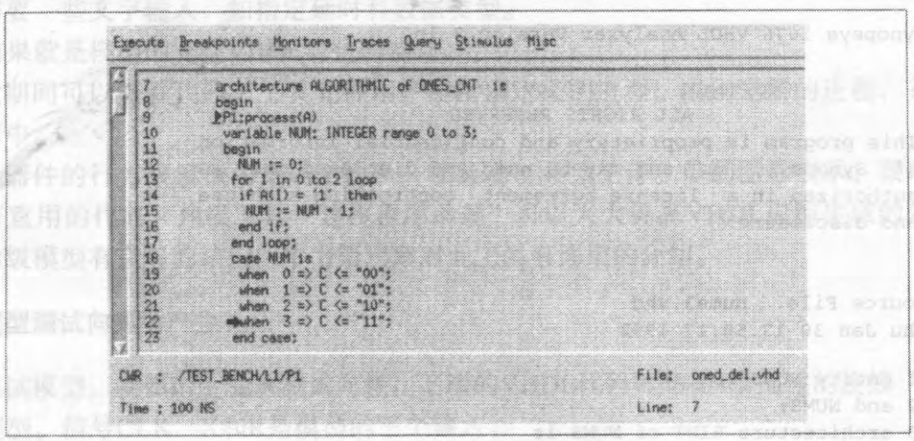


图2-10　图形调试器

2.5.4　解释结果

仿真系统的一个重要特征是解释结果的机制。图2-11给出了在Synopsys仿真程序中运行

"连续0、1检测器"的时序图。时序图是表示系统关键信号时序关系的最好方式。然而，对于复杂的模型和详细分析结果，文本输出更好。图2-12给出了Synopsys仿真程序用文本方式表示的仿真结果，与图2-11表示的结果相同。注意：文本输出详细显示了输出逻辑险态，在波形显示的全视图模式中用双线表示险态。

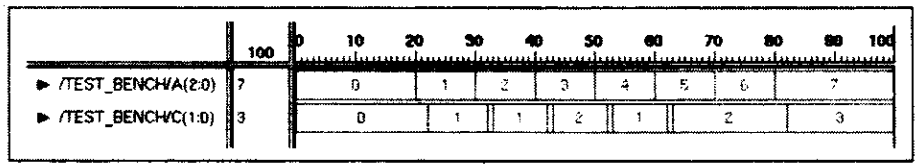


图2-11 波形显示

```
/
   1 NS
      SMON1:     ACTIVE /TEST_BENCH/C (value = X"0")
   10 NS
       SMON:     ACTIVE /TEST_BENCH/A (value = X"0")
   20 NS
       SMON:     ACTIVE /TEST_BENCH/A (value = X"1")
   22 NS
      SMON1:     ACTIVE /TEST_BENCH/C (value = X"1")
   30 NS
       SMON:     ACTIVE /TEST_BENCH/A (value = X"2")
   32 NS
      SMON1:     ACTIVE /TEST_BENCH/C (value = X"0")
   33 NS
      SMON1:     ACTIVE /TEST_BENCH/C (value = X"1")
   40 NS
       SMON:     ACTIVE /TEST_BENCH/A (value = X"3")
   42 NS
      SMON1:     ACTIVE /TEST_BENCH/C (value = X"3")
   43 NS
      SMON1:     ACTIVE /TEST_BENCH/C (value = X"2")
   50 NS
       SMON:     ACTIVE /TEST_BENCH/A (value = X"4")
   52 NS
      SMON1:     ACTIVE /TEST_BENCH/C (value = X"0")
   53 NS
      SMON1:     ACTIVE /TEST_BENCH/C (value = X"1")
   60 NS
       SMON:     ACTIVE /TEST_BENCH/A (value = X"5")
   62 NS
      SMON1:     ACTIVE /TEST_BENCH/C (value = X"3")
   63 NS
      SMON1:     ACTIVE /TEST_BENCH/C (value = X"2")
   70 NS
       SMON:     ACTIVE /TEST_BENCH/A (value = X"6")
   72 NS
      SMON1:     ACTIVE /TEST_BENCH/C (value = X"2")
   80 NS
       SMON:     ACTIVE /TEST_BENCH/A (value = X"7")
   82 NS
      SMON1:     ACTIVE /TEST_BENCH/C (value = X"3")
```

图2-12 结果的文本显示

2.6 仿真的应用

仿真主要用于两个应用领域：*系统验证和故障仿真*。在系统验证中，要验证系统符合某个规格说明。然而，在设计过程中的不同阶段，系统是用模型表示的。基于第1章介绍的设计周期，一个系统可以有芯片级、寄存器级、门级或电路级的模型。用户用仿真来验证这些模型是否与规格说明相符。随着设计层次的深入，模型与物理设计有了更紧密的关系，因而，模型的正确性就近似于系统的正确性。过去，通过做出硬件原型来完成验证，即仿真并不被作为一种完全的系统验证的方法。然而，现在的仿真工具使大系统的精确建模成为可能，因而可通过仿真，而不用通过做出硬件原型，就能够保证得到正确的系统。非常复杂的VLSI系统也能够保证第一次投片就成功，因为在仿真的整个过程中，检查已经保证了这一点。当然，对设计的全过程进行完全的仿真将非常费时和昂贵，这时仿真效率就变得非常重要，在仿真引擎中采用并行处理技术成为解决问题的关键。如果不使用这些仿真引擎，在传统的处理器上完全仿真复杂的VLSI系统，将耗时几个月乃至几年，因而并不经济。

关于通过仿真来验证系统的正确性有几个问题。第一个问题是，应该仿真多少种情况？第二个问题是，如何得到验证系统正确性的输入向量？对于第一个问题的简单答案是，使用所有输入向量的所有可能组合，对系统进行完全的测试。然而，对于复杂的系统，尤其是包括时序电路的系统，这样做并不实际。因而，设计者必须选择测试情况的一个子集进行仿真。

测试生成程序可以用来在门级仿真开发测试向量。对于高级的模型，设计者或模型开发人员通常需要手工创建测试程序。

另外一个问题是，随着层次的下降，模型越来越详细，如何比较设计的不同层次的输出就成为问题。一个相关的问题是*反向标注问题*，即如何将实际的版图信息反馈回仿真模型。本书的第7章给出了一个这方面的例子。

仿真的另一个应用是故障仿真。这里，故障被引入仿真模型，仿真系统运行以观察响应。它可以用来验证测试，即创建能够检测故障的测试。故障仿真同样可以用来创建*故障字典*，将输出信号状态（症状）与故障联系起来，以便于确定故障。

2.7 综合工具

综合工具可以用来将设计步骤自动化，或在设计过程中提供帮助。在第1章我们介绍了在行为域和结构域的不同抽象层次设计的表示。设计过程被定义为从高层次抽象开始，经过一系列转换，最后以能够实现的一种低级形式结束。设计过程的上述模型引出了如下定义：

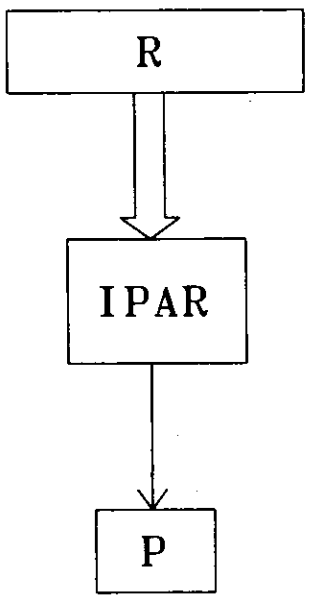
综合器：能够自动将一种设计表示形式向另一种设计表示形式转换的计算机程序，或协助进行手工转换的程序。

在第9章和第10章，将介绍如何用Synopsys的Design Compiler来综合ASIC及使用Xilinx的软件综合FPGA。这些工具主要工作于寄存器抽象级的这一层次。更高抽象层次的自动化综合正在研究之中，目前已经有*高层综合工具*在使用了。

综合工具可以将高层次的表示转化为低层次的表示，也可以从行为域转化为结构域，或同时进行。综合器进行的翻译工作和语言编译器进行的工作类似。例如，一个C编译器将高级语言编写的算法（C语言程序）编译成能够在特定机器上运行的机器代码。同样，*高层综合工*

具可以将高一级抽象的电路表示（如算法级）转化为低一级的表示（如门级），并可以用某种特定的技术实现（如CMOS）。事实上，一个将中级表示形式（一般是门级）转化为一种可以直接实现的硅版图的（结构域硅片级）的综合器称为**硅编译器**（*silicon compiler*）。

对设计者而言，在高抽象层次进行系统设计，再利用综合工具将设计转化为低层次的表示，与直接在低抽象层次来设计系统的情况进行比较，就类似于一个程序员用高级语言编程并用编译器将程序编译成机器代码和直接用机器代码进行编程时的情况。前一种情况的优点在于设计者可以将精力主要集中于系统级的问题上，而不必关心低级抽象时设计的细节问题。因为不必关心低级抽象层次的设计所面临的细节问题，在高级抽象层次上进行设计和编程将花费较少的时间，并且错误也少。但是，因为语言编译器或综合程序将不得不考虑所有转换中最为一般的情况，因而就不可能针对当前情况尽可能地优化。所以，编译过的程序和综合了的设计一般不是最优的。优化编译器或优化综合程序有时能够提供特定的优化机会，但并非对所有情况都奏效。一般情况下，由有经验的程序员编写的汇编程序，和由有经验的设计人员在尽可能低的抽象级别上做出的设计，比那些由高抽象层次进行的设计转化为的低层次设计的运行性能更好，而且开销更少。




```
--
-- This example illustrates VHDL
-- constructs that can be translated
-- into an iterative network.
--
entity IPAR is
  generic (PROP_DEL:time);
  port (
    R: in BIT_VECTOR (7 downto 0);
    P: out BIT);
end IPAR;
--
-- Model using a FOR loop
--
architecture LOOP4 of IPAR is
begin
  OPAR:process(R)
    variable X: BIT;
  begin
    X:='0';
    for I in 7 downto 0 loop
      X:=X xor R(I);
    end loop;
    P <= X after PROP_DEL;
  end process;
end LOOP4;
```

图2-13 奇校验模块的VHDL描述：电路的行为域描述，
该电路确定8位寄存器是否含有奇数个“1”

图2-13给出了一个检测如果寄存器R的“1”的个数为奇数则输出1的电路在寄存器传输级的VHDL描述。VHDL描述用FOR循环来计算结果。FOR循环可以直接转化为图2-14a所示的异或门的重复电路。因为对于VHDL，所有的FOR循环都是有效的，所以一个基本的综合程序可以用来转换成这种重复的电路。但是正如语言编译器可以进行某种优化一样，硬件综合

程序也可以寻求某种程度上的优化。优化一般在通用转化之后进行。在这个例子中，综合器应该能够判断图2-14b所示的异或门组成的树是否能够完成同样的功能，并且速度更快。而且可以知道第一个异或门可以利用布尔代数定理（X XOR 0 = X）而被消除。尽管这种改变对于异或门会起作用，但它并不意味着其他的互连可以用这种方法进行简化。优化的综合程序必须知道优化何时可行。同样，如果有三输入异或门，图2-14b中所示的设计将用更少的门并有更小的延时。这说明了优化应该是一个交互的过程，用户可以指定特定的参数，如可用的门的种类，门的级数的最大值，等等。

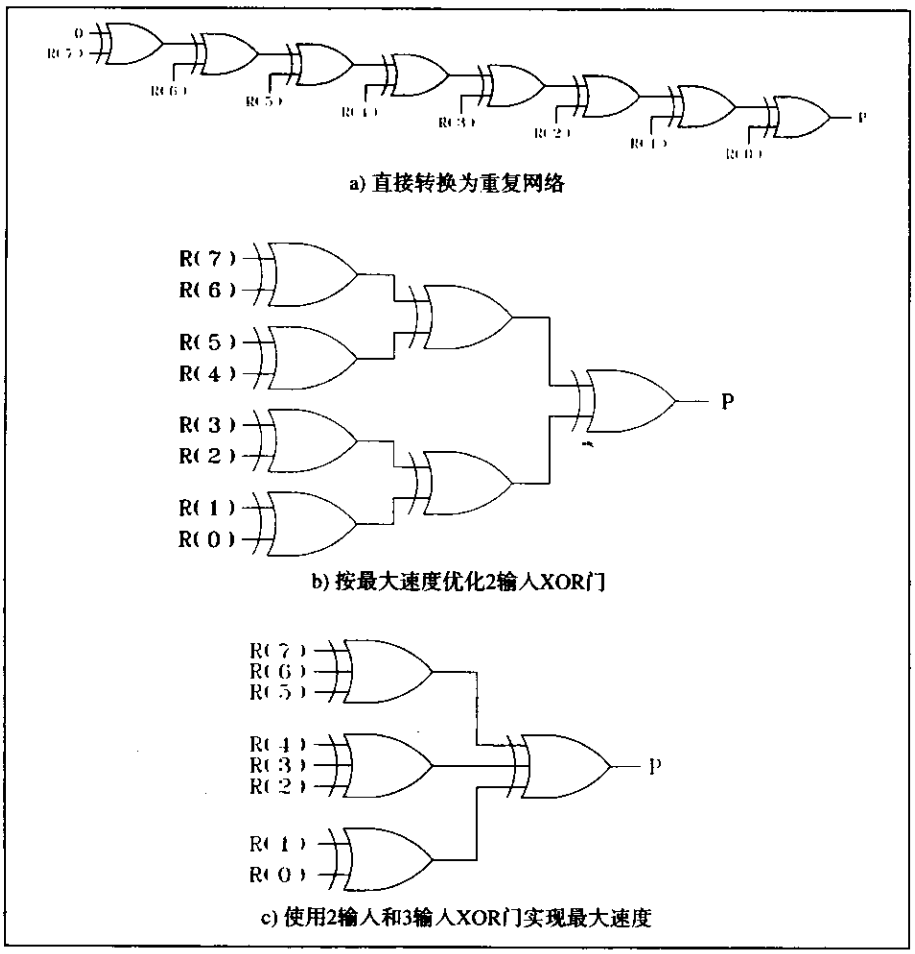


图2-14　VHDL中FOR循环的几种翻译：寄存器转换级（行为域）到门级（结构域）

这个讨论说明，转化实际上是一个两步的过程：通用转化，然后进行优化。显然，综合程序不可能了解所有的设计可能，因而，自动化的设计一般都不是最优的。在许多设计情况下，设计的优化并没有低的设计开销和高的设计可靠性来得重要。出于这个原因，许多公司在使用自动化设计工具的同时还需要进行人为干预。

一般来说，需要更多的受过高级培训的人员，来采用汇编语言编写程序，或在低的抽象层次上详细地进行设计，这将增加设计的开销。最好的设计方法通常由要求达到的批量来决定，小批量的产品可以在高的抽象级别上进行设计，自动转化为低抽象级别的表示以节省耗费。

大批量产品则应该在低抽象级别上进行设计，因为更高的设计开销在所要求的品质上得到补偿，而在生产中节省的开销（或更快速的电路）将弥补所增加的设计开销。对设计层次的选择也将影响到产品上市的时间。在高抽象级别进行设计，并将结果用程序进行转化能使设计在较短的时间内完成，在低抽象级别进行的设计将花费较多的时间，且生产成本可能会更高。

习题

2.1 找出与图2-1中的CAD工具分类相匹配的特定工具。根据自己的经验、文献检索和开发商的文档，识别出对应CAD工具分类中每个位置的特定工具。

2.2 用你最喜欢的图形编译器画出一个全加器的门级描述。为什么用这种图形表示？把你画出的图与一个数字设计捕获系统产生的图进行比较。

2.3 下面的VHDL描述中含有错误。使用VHDL软件分析并仿真这段代码。把错误按词法、语法和语义错误分类。哪种错误可以在分析和仿真阶段被发现？使用不同的VHDL系统是否得出相同的结果？

```
entity ERROR is
  port(X: in  BIT; C: in BIT; Z1: out INTEGER; Z2: out BIT);
end ERROR;
architecture HAS of ERROR is
  signal S^D: BIT;
begin
  Z1 <= X;
  process(X)
  begin
    if C= '1' then Z2 <= X;
  end process;
end HAS;
```

2.4 “事务”和“事件”的区别是什么？

2.5 在Synopsys仿真程序中，可以通过创建一个控制文件来监视所有的“活动”信号，这里“活动”的意思是什么？

2.6 观察图2-15所示的逻辑电路。对于电路中的每个门，它的输出上升延时（TLH）和输出下降延时（THL）已经给出，即电路的延时与输出信号为上升或下降有关。在第7章里将更深入地研究这种情况。假设电路的输入（A,B,C）的初始值为1。当t=5 ns时，B变为0。画出表示信号A、B、C、$\bar{B}$、A1、A2和R被更新的时间队列，并画出它们的时序图。

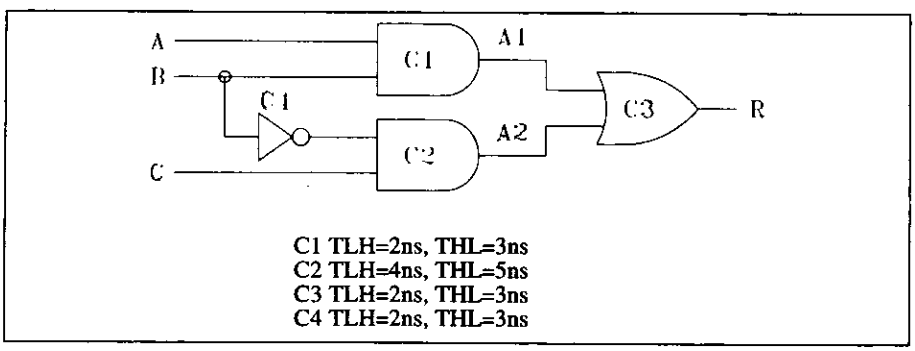
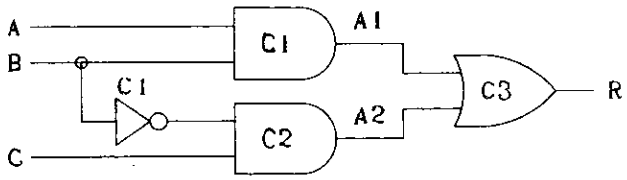


图2-15 具有时钟的电路

2.7 图2-16给出的电路与上一道题的电路相同，只有延时值不同。这个例子里的初始状态为A=B=“0”，C=“1”。图的底部给出了该电路的时间队列，画出包括所有事务和事件的时序图。



C1 TLH=4ns, THL=5ns
C2 TLH=1ns, THL=2ns
C3 TLH=1ns, THL=2ns
C4 TLH=2ns, THL=3ns

Sim Time	Queue Entry		
	Signal	Value	Time
0	A	1	10
0	B	1	10
0	C	0	10
10	A2	0	12
10	$\overline{B}$	0	13
10	A1	1	14
12	R	0	14
14	R	1	15

图2-16 时间队列问题

2.8 研究一个简单逻辑仿真程序的操作。例如，参考Alexander Miczo所著的《数字逻辑测试和仿真》(Harper & Row 1986年出版)。这个仿真程序用BASIC语言描写。研究它的程序清单，在个人电脑上运行该仿真程序并观察其操作。用这个仿真程序仿真几个简单的门级电路。简述其仿真周期的情况。

2.9 分析并仿真下面的振荡器的VHDL模型，用列表和波形输出观察响应的结果。

```
entity OSC is
end OSC;
architecture WAIT_DEL of OSC is
  signal CLOCK: BIT := '0';
begin
  process
    variable NUM: INTEGER := 0;
  begin
    while NUM < 10 loop
      CLOCK <= '1';
      wait for 75 ns;
      CLOCK <= '0';
      wait for 25 ns;
      NUM := NUM + 1;
```

```
    end loop;
    wait;
  end process;
end WAIT_DEL;
```

2.10 问题有两个部分：

a) 执行下面的WORKVIEW DIGITAL TUTORIALS部分。这些指导将使你熟悉WORKVIEW原理图获取工具和仿真系统的操作，1) Design Capture Tutorial；2) Simulation Tutorial。

b) 为John F. Wakerly所著的《数字设计原理及实践》(Prentice Hall 1990年出版）一书中的图5-15所示的上升沿触发的D触发器建立一个模型，图中的两个D符号为图5-12所示的锁存电路。对模型进行仿真，选择可以对模型进行完全测试的输入序列。

这个问题需要完成一份非正式报告，报告中对辅助说明和D触发器两项都至少要包括：

a) WORKVIEW输出的原理图，包括所有子图。

b) 仿真输出：

1) 使用表格输出；

2) 使用波形输出。

2.11 使用两种方法测试32位全加器模型，指出确定现象和随机现象的区别。

a) 使用WORKVIEW或类似的设计软件，产生一个16位行波进位加法器的门级原理图。

b) 编写一个可以产生所有可能输入的高级语言程序，如C或C++。把这些输入数据放入输入数据文件，对加法器进行仿真，记录仿真所需要的时间。

c) 采用随机数的方法，编写一个产生100个随机输入数据的高级语言程序。把这些输入数据作为输入数据文件对加法器进行仿真。记录仿真所需时间。

d) 检查这100个随机输入数据，是否还存在其他数据可以对加法器仿真？

e) 比较这两种方法所需的时间，以及它们对加法器测试的完全性。

2.12 本章中把仿真效率定义为实际逻辑时间与主机CPU时间之比。有很多操作系统具有测量CPU时间的机制，使用这些系统来测试以下仿真的效率：

a) 使用SPICE仿真电路级的全加器模型。

b) 使用门级仿真程序仿真门级的全加器模型。

c) 全加器的行为域模型可以用如VHDL的硬件描述语言编写，也可以用如PASCAL或C等高级语言编写。画出抽象级别与仿真效率的座标图。

2.13 仿真可以通过并行处理技术来进行加速。研究相关资料准备有关这种方法的报告。

2.14 在VLSI设计过程中，期望能够一次投片成功，根据这一需要，讨论仿真效率是如何限制芯片的复杂度的。

2.15 时序分析程序是用于找出电路最长延时路径的工具。研究时序分析程序的有关资料，写出一份比较时序分析程序与采用仿真来发现时序问题的报告。

第3章 VHDL的基本特征

本章介绍VHDL的基本特征，首先通过一个例子演示VHDL的基本特性，然后对语言各语句的语法和语义进行详细讨论。

VLSI时代需要结构化的设计过程，针对这种需要，人们在辅助设计领域进行了大量的工作，硬件描述语言就是这样的一个例子。事实上，使用这些语言并不是新事物。70年代就开始使用如CDL、ISP和AHDL这样的语言了。当然，它们主要的应用在于验证系统结构的正确性，并没有进行高精度模块设计的能力，这是因为它们的时序模型不够精确，或它们的语言构造隐含地与某种硬件结构相对应。更新的语言，如Verilog 和VHDL，有更加通用的时序模型，并且不针对某种特定的硬件。

硬件描述语言主要有两个应用：为一个设计建立文档和对设计进行建模。好的设计文档有助于保证设计的正确性，而且对设计的可移植性也很重要。

所有有用的硬件描述语言都由仿真程序来支持。这样就能够用硬件描述语言所写的模块来验证设计。建立一个复杂系统的原型耗费极大，硬件描述语言的发展目标则是用仿真进行验证的过程代替建立原型的过程。HDL的另一个重要应用在于逻辑综合，即从HDL模型直接产生门级电路。此外，HDL模型也用来产生测试。

在设计过程中使用硬件描述语言，大大改变了传统的设计方法。数字系统的设计人员往往拘泥于文字描述、框图、时序图和逻辑原理图来描述它们的设计。而那些进行仿真的设计人员则被称为“anathema”（“被咒逐的人”）。意思是：“只有那些不能设计的人才进行仿真。”情况现在已经大大不同了，设计人员在软件上的训练大大加强，他们在计算机上进行设计，设计以HDL源文件、原理图或两种方法相结合的方式输入，同时仿真已经成为被设计人员越来越多地用来验证设计正确性的手段，综合工具被用来将HDL模块转变为门级设计。因此，过去在学校中受过训练的设计人员已经不能够胜任这种新方法了，他们会认为使用硬件描述语言是不必要的开销。希望他们能够通过看到使用这种语言和相关工具进行设计产生的结果——更好的文档，没有错误，更快的生成设计——而改变他们的态度。

为了演示结构化设计的概念，必须使用一种硬件描述语言。直到目前，还没有一种标准的硬件描述语言能像标准的编程语言，如C、C++和JAVA一样。然而，在1983年，美国国防部资助开发了VHSIC硬件描述语言（即VHDL），其初始目的在于方便各个承包商交流超高速集成电路的设计，该语言的设计得到了计算机业界的支持，因而在硬件描述语言应该有哪些特征等方面综合了多家的观点。

1985年8月，该语言的7.2版发布，标志了语言设计的第一阶段的完成。7.2版是一个完整的语言，包含了用于结构化和行为建模的结构以及建立设计文档的方法。在7.2版发布之后，国防部又资助了VHDL语言的进一步开发，其目标是开发出改进的标准化的语言。修正阶段于1987年5月结束，并发布了语言的参考手册（LRM），提供给业界进行修改。1987年7月IEEE接受该版本的VHDL作为标准，并于1987年12月成为官方标准。从1988到1992年，VHDL用户建议对语言进行一些小的修改，并于1993年投票通过，本书大部分VHDL程序遵循

1987年版标准。

本书选用了VHDL语言来说明结构化设计的概念。之所以选择VHDL，不仅因为其流行性，而且因为作者认为VHDL的语言构件非常适合于设计的有效表示。

本章是对VHDL的简要介绍，随着讨论的深入，将在后继章节中进一步讨论VHDL的某些特性，作者发现这是一种非常有效的讲授计算机语言的方法，读者应该通过查阅手册以进一步了解语言的细节问题。

3.1 VHDL语言的基本结构

为介绍该语言，本节用一些简单的例子来说明VHDL语言的基本结构，该语言的语法在下一节详细说明。

3.1.1 设计实体

在VHDL中，特定的逻辑电路用设计实体来表示，被表示的逻辑电路可以像微处理器一样复杂，也可以像与门一样简单。设计实体按照如下顺序包括两种不同类型的描述：*接口描述*和一个或多个*结构体*。我们用一个例子进行说明：考虑一个对输入长度为3的向量中‘1’的个数进行计数的电路的接口描述：

```
entity ONES_CNT is
port (A: in BIT_VECTOR( 2 downto 0);
      C: out BIT_VECTOR( 1 downto 0));
end ONES_CNT;
```

可以看到，接口描述定义实体，并描述其输入输出。接口信号的描述包括信号的模式（即*输入*或*输出*）和信号的类型。在本例中，A是一个3位的输入数组，类型为BIT_VECTOR，下标的范围是（2，1，0）。C是2位的输出数组，类型为BIT_VECTOR，下标的范围是（1，0）。本章的后一部分将对数据类型的细节做进一步的介绍。

接口描述也记录了实体属性的文档信息。例如，我们重写上述的接口描述，并在注释里列出了真值表。图3-1说明了如何将真值表以注释的形式插入描述之中。任何以两个短划线开头的一行都被认为是注释。这只是可以加入接口描述的信息类型的一个途径，我们将在后续章节给出其他的例子。

```
entity ONES_CNT is
port (A: in BIT_VECTOR(2 downto 0);
      C: out BIT_VECTOR(1 downto 0));

------ Truth Table:
---
------------------------------
---|A2   A1   A0  |  C1   C0 |
------------------------------
--  |0    0    0   |  0    0  |
--  |0    0    1   |  0    1  |
--  |0    1    0   |  0    1  |
--  |0    1    1   |  1    0  |
--  |1    0    0   |  0    1  |
--  |1    0    1   |  1    0  |
--  |1    1    0   |  1    0  |
--  |1    1    1   |  1    1  |
end ONES_CNT;
```

图3-1 带注释的接口描述

3.1.2 结构体（构架）

接口描述只定义设计实体的输入和输出，结构体则定义了实体的行为模型或采用更加初级的组件来构成实体的结构化模型。

在设计过程的开始，设计者头脑中存在他们想要实现的算法。起初，他们想检验算法的正确性而不考虑具体的实现，因而实现的第一个构架应该是算法级的。图3-2就是‘1’计数器的一个这样的描述。

```
architecture ALGORITHMIC of ONES_CNT is
begin
  process(A)
    variable NUM: INTERGER range 0 to 3;
  begin
    NUM := 0;
    for I in 0 to 2 loop
      if A(I) = '1'  then
        NIM := NUM + 1;
      end if;
    end loop;
    case NUM is
      when 0 => C <= "00";
      when 1 => C <= "01";
      when 2 => C <= "10";
      when 3 => C <= "11";
    end case;
  end process;
end ALGORITHMIC;
```

图3-2 ‘1’计数器的算法描述

注意图3-2所示的结构体。循环按照从A（0）到A（3）的顺序扫描输入，每当发现某位为‘1’时，就递增变量NUM。根据NUM的最后值，由一个*case*语句选择传递给输出的位模式。

该行为级构架完整地描述了算法，但是它与实际硬件电路的对应性较差，如对于下面的问题：循环结构对应的是什么逻辑电路？综合的难易程度如何？整个电路的延时是多少？该构架并不能提供任何答案。但是，在设计的初级阶段，一般是不考虑这些细节问题的。

继续上面的例子，假设对‘1’计数器的设计进入了逻辑设计阶段。图3-3为两个输出C1和C0的卡诺图。由图可以得出C1和C0的布尔方程如下：

$$C1 = (A1)\,(A0) + (A2)\,(A0) + (A2)\,(A1)$$

$$C0 = (A2)\,(\overline{A1})\,(\overline{A0}) + (\overline{A2})\,(\overline{A1})\,(A0) + (A2)\,(A1)\,(A0) + (\overline{A2})\,(A1)\,(\overline{A0})$$

应该注意，C1是三个输入A2、A1、A0的多数函数（majority function）[MAJ3（A2、A1、A0）]，C0是三个输入的奇校验函数[OPAR3（A2，A1，A0）]。这时，设计者应该将算法级结构体改为图3-4中所示的数据流结构体。

在实现C（1）和C（0）的“积之和”等式的语句中，“与”项放入括号中。这是因为在VHDL中，“与”操作符和或操作符具有相同的优先级。

图3-4的两级逻辑机制实现了一个利用标准与门、或门和反相器的门结构。然而，既然C1能用MAJ3函数进行运算，C0能用OPAR3进行计算，一个更简单的宏级描述的构架如下：

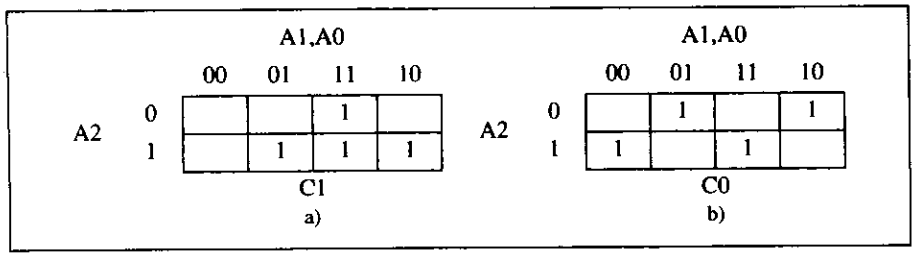


图3-3 ‘1’计数器的卡诺图

```
architecture DATA_FLOW of ONES_CNT is
  begin
    C(1) <= (A(1) and A(0)) or (A(2) and A(0))
         or (A(2) and A(1));
    C(0) <= (A(2) and not A(1) and not A(0))
         or (not A(2) and not A(1) and A(0))
         or (A(2) and A(1) and A(0))
         or (not A(2) and A(1) and not A(0));
  end DATA_FLOW;
```

图3-4 ‘1’计数器的数据流模型

```
architecture MACRO of ONES_CNT is

begin

  C(1) <= MAJ3(A);
  C(0) <= OPAR3(A);

end MACRO;
```

该结构体认为在硬件级存在MAJ和OPAR门。如果是VHDL描述的形式，就要求函数MAJ3和OPAR3必须预先声明和定义过了，并且是可见的。下一节将讲解如何实现这一点。

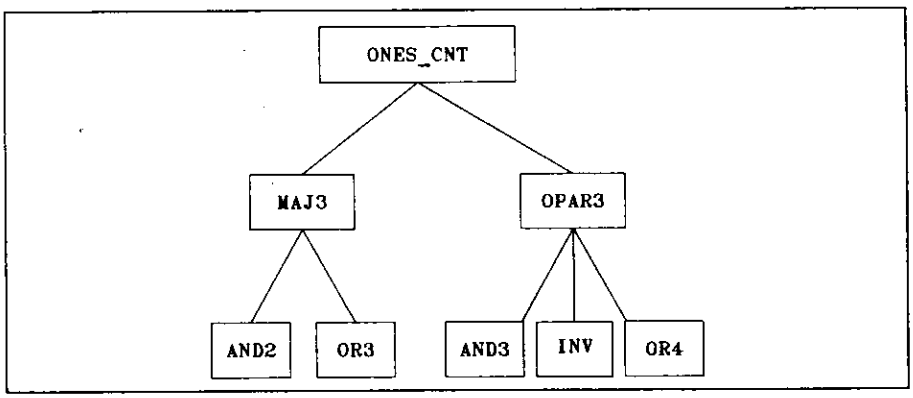


图3-5 ‘1’计数器的结构化设计层次

前述的‘1’计数器的三个结构体都是行为级的，因为它指定了输入输出的响应，而没有明确指出其内部结构。我们现在给出一个‘1’计数器的结构级描述。在这种方法里，采用图3-5所示的设计层次的形式，即，‘1’计数器先被分解为门MAJ3和OPAR3，然后这些门又被

分解成与门、或门和反相器。图3-6给出了门MAJ3、门AND2和门OR3的接口描述和构架。

```
use work.all;
entity MAJ3 is
  port (X: in BIT_VECTOR(2 downto 0); Z: out BIT);
end MAJ3;
architecture AND_OR of MAJ3 is
  component AND2C
    port (I1,I2: in BIT; O: out BIT);
  end component;
  component OR3C
    port (I1,I2,I3: in BIT; O: out BIT);
  end component;
  for all: AND2C use entity AND2(BEHAVIOR);
  for all: OR3C use entity OR3(BEHAVIOR);
  signal A1,A2,A3: BIT;
begin
  G1: AND2C
    port map (X(0),X(1),A1);
  G2: AND2C
    port map (X(0),X(2),A2);
  G3: AND2C
    port map (X(1),X(2),A3);
  G4: OR3C
    port map (A1,A2,A3,Z);
end AND_OR;
```

a) MAJ3描述

```
entity AND2 is
  port (I1,I2: in BIT; O: out BIT);
end AND2;
architecture BEHAVIOR of AND2 is
begin
  O <= I1 and I2;
end BEHAVIOR;
```

b) AND2描述

```
entity OR3 is
  port (I1,I2,I3: in BIT; O: out BIT);
end OR3;
architecture BEHAVIOR of OR3 is
begin
  O <= I1 or I2 or I3;
end BEHAVIOR;
```

c) OR3描述

图3-6 MAJ3的结构说明

我们将详细介绍实体MAJ3的结构体AND_OR。图3-7给出了该门结构的逻辑图。再看图3-6a，注意到在结构体AND_OR的声明部分，声明了两个元件（AND2C和OR3C）。它使用的端口说明和在实体定义时使用的端口声明（见图3-6b和图3-6c）是相同的。然后，声明了三个信号（A1、A2、A3）。三个与门的输出必须连到或门的输入，由于定义了信号，使得我们可以表示出这些连接。在关键词*begin*之后，实例化了四个元件：即创建了某个通用元件实体的特例。注意实例都用唯一的符号进行标记（即G1、G2、G3和G4），并加上了端口映射表。端

口映射确定了信号和元件声明的输入输出端口的一一对应关系。在本例中，映射是根据位置进行的。在另一种方法中，对应可以与位置无关。例如，标记为G1的实例端口映射表可以写成如下形式：

```
G1: AND2C
    port map (O => A1, I1 => X(0), I2 => X(1) );
```

在这种情况下，箭头把在元件声明时声明的形式信号与在结构级结构体中声明的实际信号对应起来。因为这种对应是与位置无关的，因而防止了错误，增强了可读性。

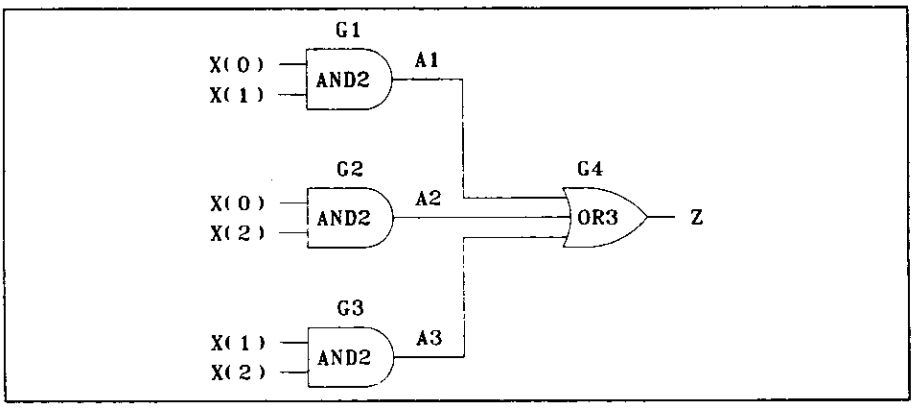


图3-7 多数函数的门级结构图

要完全定义实体MAJ3的操作，必须有在MAJ3的结构体内实例化的所有元件的接口描述和结构体。图3-6b和图3-6c给出了这些描述，OPAR3的结构也可同样定义。我们将它作为读者的一个练习（见习题3-2）。元件AND2C和OR3C的定义类似于没有芯片插入的连接器，它必须与库模型绑定（类似于插入芯片）。图3-6的下述语句完成该任务：

```
for all: AND2C use entity AND2(BEHAVIOR);
for all: OR3c use entity OR3(BEHAVIOR);
```

在MAJ3和OPAR3的元件定义完成后，结构级结构体的描述在图3-8中给出。

```
use work.all;
architecture STRUCTURAL of ONES_CNT is
  component MAJ3C
    port (X: in BIT_VECTOR(2 downto 0); Z: out BIT);
  end component;
  component OPAR3C
    port (X: in BIT_VECTOR(2 downto 0); Z: out BIT);
  end component;
  for all: MAJ3C use entity MAJ3(AND_OR);
  for all: OPAR3C use entity OPAR3(AND_OR);
begin
  COMPONENT_1: MAJ3C
    port map (A,C(1));
  COMPONENT_2: OPAR3C
    port map (A,C(0));
end STRUCTURAL;
```

图3-8 ‘1’计数器的结构级构架

3.1.3 模型测试

与其他形式的编程相同，在VHDL模型编写完之后，必须进行测试。可通过创建一个顶层模块*test bench*来实现。作为例子，图3-9给出了用来测试前面编写的ONES_CNT的test bench。注意TEST_BENCH的实体描述不包括端口语句，因为测试信号是在test bench内部产生的。在结构体ONES_CNT1中，对实体ONES_CNT的测试输入(A)和测试输出(C)被声明为信号。

```
entity TEST_BENCH is
end TEST_BENCH;

use WORK.all;
architecture ONES_CNT1 of TEST_BENCH is
  signal A: BIT_VECTOR(2 downto 0); ----Declare signals
  signal C: BIT_VECTOR(1 downto 0);
  component ONES_CNTA                ----Declare component
    port (A: in BIT_VECTOR(2 downto 0);
          C: out BIT_VECTOR(1 downto 0));
  end component;
 for L1: ONES_CNTA use entity ONES_CNT(ALGORITHMIC);
begin
  L1: ONES_CNTA
    port map(A, C);
  process
  begin
    A <= "000" after 1 ns,
         "001" after 2 ns,
         "010" after 3 ns,
         "011" after 4 ns,
         "100" after 5 ns,
         "101" after 6 ns,
         "110" after 7 ns,
         "111" after 8 ns;
    wait;
  end process;
end ONES_CNT1;
```

图3-9 实体ONES_CNT的test bench

接下来是元件ONES_CNTA的元件声明，然后将其与设计库中的实体ONES_CNT、结构体ALGORITHMIC绑定。在关键字*begin*之后，元件ONES_CNTA被实例化，其端口信号用端口映射语句映射到A和C。最后，进程语句包含一系列测试向量以驱动实体ONES_CNT。进程只在仿真开始的时候运行一次，然后遇到wait语句而被挂起。所有8个测试向量都被安排在纳秒间隔内出现。

图3-10给出了该test bench仿真的结果。右边的列指定某信号改变值的时间（单位为纳秒）。每行的剩余部分指明当时改变信号的新值。列中的破折号表示该信号在当时没有改变。输入A处于中间一列，对应的C的响应位于右边一列。在C上的响应延时为delta延时，例如：在2ns时A被赋予值“001”，C上的响应（“01”）在时间2ns+1 delta时出现。目前，我们把delta看作单位延时。在第4章，将更加精确地定义delta延时。

3.1.4 块语句

VHDL描述的一个基本元素为*block*（块）。一个块是一个限定的文本区域，包括一个声明

部分和一个可执行部分，因而结构体本身就是一个块。同时，在一个结构体内部，还可以存在内部的块。考虑图3-11所示的例子，块A和块B嵌套在构架的内部。在通常情况下，任何层次的嵌套都是可能的，例如，块A和块B可以进一步分解成子块。采用这种结构主要有两个原因：首先，它支持设计分解的自然形式，其次，对应于每个块可以有一个防护（guarded）条件。当防护条件为真时，块内某些类型的语句能够被执行。图3-12给出了一个防护块的例子。在块头“block（CON=‘1’）”中，“CON=‘1’”是一个防护表达式。防护隐含地被定义为布尔型。任何防护语句将在如下情况下被执行：1) 防护为真且防护语句右边的信号改变了。2) 防护由假变成真。当防护为假时，左边的信号保持原来的值。对于图3-12的例子，当CON为‘1’时，输出O1随着输入I1的变化而改变；当CON从‘1’变为‘0’时，O1将保持上次的值。注意，将O2的信号赋值代入语句与防护语句无关，因而只要I2变化就被激活，与防护的状态无关。

```
VHDL Report Generator

TIME    |-----------------SIGNAL NAMES-------------------|
(ns)    |        A(2 DOWNTO 0)              C(1 DOWNTO 0)
  0     |             "000"                      "00"
  2     |             "001"                      ----
 +1     |             -----                      "01"
  3     |             "010"                      ----
  4     |             "011"                      ----
 +1     |             -----                      "10"
  5     |             "100"                      ----
 +1     |             -----                      "01"
  6     |             "101"                      ----
 +1     |             -----                      "10"
  7     |             "110"                      ----
  8     |             "111"                      ----
 +1     |             -----                      "11"
```

图3-10 实体ONES_CNT的仿真结果

防护块对于寄存器基本元素的建模很有用。在第4章将进一步介绍。

```
architecture  BLOCK_STRUCTURED of SYSTEM is
--------------      Outer Block Declaration Section
--------------
--------------
begin
------------        Outer Block Executable Statements
------------
 A: block
    --------        Inner Block A Declaration Section
    --------
 begin
    --------        Inner Block A Executable Statements
    --------
 end block A;
 B: block
    --------        Inner Block B Declaration Section
    --------
 begin
    --------        Inner Block B Executable Statements
    --------
 end block B;
end  BLOCK_STRUCTURED;
```

图3-11 嵌套块的例子

```
entity GUARD_EXAMP is
    port(I1,I2,CON: in BIT; O1,O2: out BIT);
  end GUARD_EXAMP;

  architecture DF of GUARD_EXAMP is
    begin
     B:block(CON = '1')
      begin
       O1 <= guarded I1;
       O2 <= I2 ;
      end block B;
  end DF;
```

图3-12 防护块的例子

3.1.5 进程

VHDL的另一个主要建模元素为进程。图3-2给出了ONES_CNTS电路的行为级描述进程的例子。注意，该例中的进程以关键字*process*（A）开头。该例中的向量A构成了该进程的敏感信号列表。每当敏感信号表的一个信号发生变化，就会激活进程，进程块中的语句开始执行。进程块的典型应用是在某抽象级别上实现算法，即图3-2的情况。

后续章节将讨论进程结构的含义，将会发现它代表着对数字电路中的并发行为进行建模的基本方法。

3.2 词法描述

一个VHDL文本描述包括一个或多个设计文件。每个设计文件必须使用标准7位ASCII字符编码集中的字符。如果文件使用了文字处理工具包，如WORD，必须选中ASCII选项以保证产生一个纯ASCII字符代码。由这种字符处理文件生成的文件一般都包含一些非ASCII的控制字符，用来控制文件的格式。非标准字符代码将引起编译错误，因而建议使用能够产生纯ASCII文件的编辑器（如DOS编辑器或Windows的记事本）。

3.2.1 字符集

在1987版的VHDL中，只有下述字符允许在设计文件中使用。不可打印的字符，如回车，以该字符的ASCII码表示，用十六进制表示并用圆括号括起来。如回车符是（0D）=00001101。

1) 大写字母：A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

2) 数字：0 1 2 3 4 5 6 7 8 9

3) 特殊字符：" # & ' ( ) * + , - . / : ; < = > _ |

4) 空格符：（20）

5) 格式控制符：

a) 回车（0D）

b) 换行（0A）

c) 换页（0C）

d) 水平制表符（09）

e) 垂直制表符（0B）

6) 小写字母：a b c d e f g h i j k l m n o p q r s t u v w x y z

7) 其他特殊字符：！ $ % @ ? [\ ] ^ ` { } ~

图3-13给出了特殊字符的定义。

```
" quotation mark    # sharp, pound sign      & ampersand
' apostrophe        ( left parenthesis       ) right parenthesis
* star, multiply    + plus                   , comma
- hyphen, minus     . dot,period,point       / slash, divide
: colon             ; semi-colon             < less than
= equal             > greater than           _ underline
| vertical bar      ! exclamation point      $ dollar
% percent           @ commercial at          [ left square bracket
\ back-slash        ] right square bracket   ^ circumflex
` grave accent      { left brace             } right brace
~ tilde
```

图3-13 特殊字符

3.2.2 词法元素

词法元素指不可以拆分为其他更小元素的字符串，它是基本元素。设计文件是由词法元素的序列和分隔符组成的。词法元素的类型有：

1) 分界符。

2) 标识符。

3) 注释。

4) 字符文字。

5) 字符串文字。

6) 位串文字。

7) 抽象文字。

在相邻的词法元素之间，第一个词法元素之前，某行或某文件的最后一个词法元素之后，可以有任意数量的分隔符。在某些情况下，当两个词法元素连写会被当作一个词法元素时，两个词法元素之间必须加分隔符。

分隔符是：

1) 空格符。

2) 格式控制符。

3) 行末符。

语言并未指定用什么来表示行末符，这与实现有关。但行末符必须用格式控制符来定义。

3.2.3 分界符

分界符是用来分隔词法元素的，有特殊的语义。如下字符被归类为分界符。

```
& ' ( ) * + , - . / : ; < = > |
```

所有的VHDL语句都由分号（；）终止。其他字符的含义必要时在后续章节中加以介绍。

复合分界符指有特殊意义的连续的两个分界符。合法的复合分界符为：

```
=>   **   :=   /=   >=   <=   <>   --
```

复合分界符的用法由上下文来确定。例如，复合分界符<= 在某个语义环境里表示“小于等于”，而在另外的语义环境中则表示“对信号赋值”。

在多数情况下，分界符和复合分界符的使用省去了在词法元素中间插入特殊分隔符的必要。例如，（A * B）和（A*B）是一样的。

3.2.4 标识符

标识符是对象名或是保留字。保留字是在语言中有特殊地位的标识符，保留字不可以由用户显式地声明为标识符，图3-14给出了1987标准VHDL中定义的保留字。

abs	disconnect	label	package	units
access	downto	library	port	until
after		linkage	procedure	use
alias	else	loop	process	
all	elsif			variable
and	end	map	range	
architecture	entity	mod	record	wait
array	exit		register	when
assert		nand	rem	while
attribute	file	new	report	with
	for	next	return	
begin	function	nor		xor
block		not	select	
body	generate	null	severity	
buffer	generic		signal	
bus	guarded	of	subtype	
		on		
case	if	open	then	
component	in	or	to	
configuration	inout	others	transport	
constant	is	out	type	

图3-14 1987标准VHDL中的保留字

用户声明的标识符用来命名变量、块、进程等等。标识符是任何由字母打头的字母、数字和用下划线连接的字符串的组合。不可以有两个相连的下划线。最后的字符必须是字母或数字。下述为合法的用户定义的标识符。

```
COUNT          cout          c_out          AB2_5c
VHSIC          X1            FFT            Decoder
A_B_C          xyZ           h333           STORE_NEXT_ITEM
```

下述用户定义的标识符是非法的。

```
2CA            My-name       H$B            LOOP           _ABC
A__B           Decoder_      alpha 2        END            AB AC
N#3            BEGIN         MAP            REGISTER
```

所有用户定义的标识符必须互相区别，并与保留字相区别。VHDL是大小写无关的，用户定义的两个标识符，如果区别仅在大小写字母上，那么会被认为是同一个标识符。例如

STATE=state，ABC=AbC和A23_D=a23_d。下划线在标识符中很重要。本书中约定，用户定义的标识符用大写字母表示（COLOR、TRISTATE、GREEN、S1，等等），保留字用小写的斜体表示（例如*type*），内置的数据类型仅仅将首字母大写，这样做是为了提高可读性。

3.2.5 注释

注释是用分界符“--”开头，并在行尾结束的词法元素。显然，注释一定是一行之中最后的词法元素。注释可以开始于一行的任何一处。它可以跟在一行之中的合法词法元素之后，也可以是该行唯一的词法元素。注释不影响编译器和仿真程序，只是为了增加程序的可读性。注释可以包括合法字符，包括除了行末标识符以外的所有特殊字符。一些例子如下：

```
-- This line consists of only a comment.

C:=A*B; -- This line contains a VHDL statement followed by a comment.
        -- Long comments can be continued on the next line. The statement
        -- contains three user-defined identifiers (A B C) and three
        -- delimiters (:= * ;).

   ---------- Extra hyphens can enhance readability.
```

3.2.6 字符文字

字符文字指在两个撇号之间插入的一个字符，用来指定用于标量对象初始化的常量值，下面是一些例子：

```
'A'    ' '    '$'    '''    'b'    '.'    '1'
```

3.2.7 字符串文字

字符串文字是在两个引号之间插入一串可显示字符而得到的词法元素。串中可以不包含任何字符。如果串中包含一个引号，则使用两个相连的引号表示。注意，该约定排除了将两个字符串文字写在同一行，而中间没有分隔符进行分隔的情况。一个字符串文字的长度指的是串中字符的个数（将双引号记为一个字符）。下面是一些例子：

```
"A simple string literal."  -- length=24
""                          -- length=0
"A"                         -- length=1,  This is different from 'A'.
""""                        -- length=1, a single quotation character.
"Special characters, $, #, and | are allowed."
```

字符串文字必须写在一行中，因为它是词法元素。长度超过一行的串必须用相连接的短的字符串来表示。连接符是“&”字符。

```
"This is a long string literal that will not fit on one"&
"line which requires using the concatenation operator."
```

3.2.8 位串文字

位串文字是引号中间的数字串后面紧跟着一个基标志符的词法元素。基标志符是如下字母之一：B（二进制），O（八进制），或X（16进制）。数字串的数字必须和基相符。例如，B（0，1），O（0~7），X（0~9，A~F）。无论基标志符如何，都认为位串文字的值和位串的值相

等。例如，X“A”的值即为“1010”。在任何情况下，位串文字只是一个位串，VHDL也仅将它看作是一个位串，其数字值的定义取决于用户。例如，如果“1010”装入寄存器，在认为寄存器保存正值的情况下，用户可以将其解释为（+10）。或者在认为寄存器保存的为补码的情况下，将其解释为（-6）。VHDL语言一般将该情况解释为位串“1010”。位串文字用于指定二进制寄存器的内容的初始状态。位串文字也可被直接指定为位串，而不使用任何基标志符。

下划线字符可以用来增加位串文字的可读性，而不影响位串的值。位串文字的长度即位串中位的个数，与基标志符无关。下划线不影响位串文字的长度。然而，在未指定字符文字的基标志符时，不允许使用下划线。下面是一些例子：

```
B"11011110"    -- Length 8
"11011110"     -- Length 8, no base specifier
B"1101_1110"   -- Also length 8 and equivalent to the previous two examples.
X"DE"          -- Also length 8 and equivalent to the previous examples.
X"27"          -- Length 8, equivalent to B"0010_0111".
O"742"         -- Length 9, equivalent to B"111_100_010".
"1101_1110"    -- ILLEGAL SYNTAX, underscore is not a valid element value
               -- of type Bit.
```

3.2.9 抽象文字

抽象文字是指具有数值的词法元素。有两类抽象文字：实型和整型。实型文字有一个小数点，整型文字没有。有两种类型的抽象文字：十进制文字和基数文字。

3.2.10 十进制文字

十进制文字是在许多计算机高级语言中使用的由标准十进制表示的抽象文字。与位串文字相似，下划线可以用来提高十进制文字的可读性，而不影响文字的值。指数部分用字母E打头，只有实型文字才允许有负的指数。十进制文字的第一个字符必须是一个十进制数字，下面是一些整型十进制文字的例子：

```
5          85_257    3E3          0              015              23e0
```

下面是一些实型十进制文字的例子：

```
2.5        0.0       0.25         256.857        25.834_233       0.000_000_25
2.0e5      2.0E-5    2.0E+5       15.67e25       1.94E-15         03.14159
```

在抽象文字中不允许空格的存在。下述为非法的十进制文字：

```
2e-2       2.0e0     2,354        2.5 e-2         2.5e -2         .25
```

3.2.11 基数文字

基数文字是包括一个在2和16之间的基标志符的抽象文字。基标志符通常写为基10。在基数文字中，基数部分在基大于10的时候用A~F表示10~15。大小写字母意义相同。基数文字按如下顺序包括一个基标志，一串尾数位，及可选择的指数位。尾数位必须包含在‘#’号之间。在基数文字中间，不能够有空格。

```
-- Integer-based literals with value 4095
```

```
16#FfF#   2#1111_1111_1111#  10#4095#   8#7777#   4#333333#
-- Integer-based literals with value 480
16#1E#E1  2#1_1110#e4  4#132#E2  16#1e0#  2#0001_1110_0000#
-- Real-based literals with value 4095.0
16#0.fff#e3  2#1.1111_1111_111#E11  4#3.33333#e5
```

3.3 VHDL源文件

VHDL源文件是由一串由分隔符和分界符分隔的词法元素组成。任何词法元素都不可以分作两行。词法元素（分界符、标识符、注释、字符文字、字符串文字、位串文字和抽象文字）组成了可以在源文件中出现的合法的基本结构。分隔符和格式控制符定义词法元素的边界和分行。

在讲解VHDL语法时，提供了一些实例。下一节中定义的许多数据类型的例子在后续章节中将被多次用来定义各种各样的变量和信号。

3.4 数据类型

数据类型是已命名的一组值。对象的数据类型定义了该对象可以具有的值和对该对象可以进行的运算的限制。在VHDL中，由数据类型添加的限制是被强制执行的。VHDL是一种类型很强的语言。例如，整型的数据对象必须有整型的值（…-2，-1，0，1，2，…）。在该数据对象上进行的操作必须是那些限定于整型操作的运算，如加法、乘法等等。

子类型是带有限制的类型。当一个值是某个类型的合法值并且满足限定条件时，该值则属于该类型的子类型。例如，自然数类型是整型的子类型，其值限制为自然数（0，1，2，…）。子类型的基类是指创建该子类型的类型。例如，自然数子类型的基类是整型。一个类型是它本身的子类型，称为无限制子类型。一个类型也是它本身的基类。

3.4.1 数据类型分类

图3-15表示了VHDL语言完整的数据类型分类。标量类型的值由单个的值组成。例子包括整型和自然数类型。复合类型的值是包含多个元素的复合值，例如数组和记录。存取类型是能对其他数据类型进行访问的数据类型。文件类型提供对文件的存取。枚举类型和整型被归类为离散类型。整型、实型和物理类型被归类为数值类型。

1. 标量类型
   枚举类型-离散类型
   整数类型-离散类型、数值类型
   物理类型-数值类型
   浮点（或实数）类型-数值类型
2. 复合类型
   数组-所有元素具有相同类型
   记录-元素的类型可能不同
3. 存取类型
4. 文件类型

图3-15 数据类型的分类

3.4.2 标量数据类型

标量数据类型由简单、单一的值组成。一个整型值可以为15。标量类型与复合类型相反，如向量可以有一个复合的值（15，25，30）。标量类型包括枚举数据类型、数值数据类型（包括整型和实型）和物理数据类型。

1. 枚举类型

枚举类型是一种标量类型，定义为将一系列的值以一定的顺序列出。该序列中的元素可以为标识符或字符文字。图3-16给出了VHDL语言预定义的枚举数据类型。

```
type Bit is ('0', '1');
type Boolean is (FALSE, TRUE);
type Severity_Level is (NOTE, WARNING, ERROR, FAILURE);
type Character is (
     NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
     BS,  HT,  LF,  VT,  FF,  CR,  SO,  SI,
     DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
     CAN, EM,  SUB, ESC, FSP, GSP, RSP, USP,
     ' '  '!'  '"'  '#'  '$', '%', '&', '''
     '(', ')', '*', '+', ',', '-', '.', '/',
     '0', '1', '2', '3', '4', '5', '6', '7',
     '8', '9', ':', ';', '<', '=', '>', '?',
     '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
     'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
     'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
     'X', 'Y', 'Z', '[', '\', ']', '^', '_',
     '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
     'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
     'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
     'x', 'y', 'z', '{', '|', '}', '~', DEL);
```

图3-16 预定义枚举数据类型

类型Bit的两个值分别为字符文字“0”和“1”。VHDL语言定义了内置的逻辑操作符*not*、*and*、*or*、*nand*、*nor*和*xor*，它们可用于类型为Bit的变量和信号。

内置类型Boolean有值FALSE和TRUE。Boolean类型的变量和信号用来控制代码段的条件执行。内置的操作符*not*、*or*、*and*、*nand*、*nor*和*xor*同样也适用于Boolean类型的变量和信号。

内置字符类型（Character）定义了标准的128字符的ASCII字符集。该类型用于字符的操作。内置数据类型Severity_level将在后面与*assert*语句同时讨论。

用户定义的枚举数据类型实例如下：

```
type COLOR is (RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET);
type Bit is ('0', '1');
type TRISTATE is ('Z', '0', '1');
type STATE is (S0, S1, S2, S3, S4);
type STD_ULOGIC is ('U','X','0','1','Z','W','L','H','-');
```

类型STD_ULOGIC是IEEE标准1164定义的9值类型，对于综合非常重要。该类型的子类型STD_LOGIC和一个相关的判决函数是实际用于信号和变量定义的类型。详细细节后面还有介绍。

数据类型的值从左到右按升序排列，中间用逗号分隔，并且用圆括号括起来。如同前面

所指出的，所有的VHDL声明由分号结束。

如果同一标识符或字符文字被声明为多个枚举类型，则称为重载（*overload*）。例如，字符文字‘0’和‘1’如果同时被定义为BIT和TRISTATE类型，则为重载。

既然数据类型的值认为是升序排列，对于排列中每个元素赋予一个位置编号是很方便的。位置标号从最左边的元素开始标记为0，向右逐个递增1。类型COLOR的位置编号如下所示：

```
RED     ORANGE    YELLOW    GREEN     BLUE      INDIGO    VIOLET
 0        1         2         3         4         5         6
```

既然位置编号定义了值的数字顺序，类似“GREEN<BLUE”的表达式的结果为真。正如下面描述的，属性（attribute）提供了一个访问位置编号的机制。

1) **枚举数据类型的预定义属性**。预定义属性是各种类型的基本结构带有的一个值、函数、类型或范围。枚举数据类型带有一些预定义的属性。大部分属性都和隐含的位置编号有关系。

属性'*pos*是预定义函数，它确定列表里的特定值的位置编号。属性'val是根据位置标号得出列表中相应的值的预定义函数。属性名用单引号和类型名相连（有时也叫撇号），参见这些例子。

```
COLOR'pos(GREEN) = 3                COLOR'val(3) = GREEN
STATE'pos(S2) = 2                   STATE'val(0) = S0
TRISTATE'pos('Z') = 0               TRISTATE'val(1) = '0'
Bit'pos('0') = 0                    Bit'val(0) = '0'
```

对于内置Character类型，位置编号就是它的ASCII代码，参见这些例子。

```
Character'pos(NUL) = 0              (00 in hexadecimal)
Character'pos(CR)  = 13             (0D in hexadecimal)
Character'pos('0') = 48             (30 in hexadecimal)
```

属性'*left*和'*right*是枚举数据类型的预定义值，分别定义类型列表中左边和右边的元素。参见下面的这些例子。

```
COLOR'left = RED                    COLOR'right = VIOLET
STATE'left = S0                     STATE'right = S4
TRISTATE'left = 'Z'                 TRISTATE'right = '1'
Bit'left = '0'                      Bit'right = '1'
```

属性'*high*和'*low*是枚举数据类型的预定义值，分别定义类型列表中值的上限和下限。因为与枚举数据类型有关的值只有位置编号，因而属性'*low*和属性'*left*相等，属性'*high*和属性'*right*相等。然而，后边将会见到，对于其他的数据类型或枚举类型的子类型，这两对值并不能保持相等。下面是这些值的例子。

```
COLOR'low = RED                     COLOR'high = VIOLET
STATE'low = S0                      STATE'high = S4
TRISTATE'low = 'Z'                  TRISTATE'high = '1'
Bit'low = '0'                       Bit'high = '1'
```

属性'*leftof*是枚举类型的预定义函数，根据给出的列表中的值，给出其在列表中相邻的左边的一个值。同样，属性'*rightof*是枚举类型的预定义函数，根据给出的列表中的值，给出其列表中相邻的右边的值。参见例子：

```
COLOR'leftof(GREEN) = YELLOW        COLOR'rightof(BLUE) = INDIGO
STATE'leftof(S2) = S1               STATE'rightof(S3) = S4
TRISTATE'leftof('0') = 'Z'          TRISTATE'rightof('0') = '1'
```

属性'*pred*是枚举类型的预定义函数，根据给出的列表中的值，给出其列表中相邻的前一个值。同样，属性'*succ*是枚举类型的预定义函数，根据给出的列表中的值，给出其列表中相邻的后一个值。属性'*pred*和'*succ*指的是数字值，而非相对位置。属性'*pred*隐含“小于”的含义，'*succ*隐含“大于”的含义。见例子：

```
COLOR'pred(GREEN) = YELLOW          COLOR'succ(BLUE) = INDIGO
STATE'pred(S2) = S1                 STATE'succ(S3) = S4
TRISTATE'pred('0') = 'Z'            TRISTATE'succ('0') = '1'
```

对于枚举数据类型，'*pred*和'*leftof*相等，'*succ*和'*rightof*相等。然而，对于其他的数据类型，并不存在该相等关系。

在信号和变量初始化时，列表的顺序同样起到了作用。如果声明的变量或信号没有给出初始值，缺省的初始值是列表中最左边的元素。

子类型声明不定义新的数据类型，而仅仅限制该子类型的对象可以进行的运算的值。见例子：

```
subtype LONGWAVE is COLOR range COLOR'left to YELLOW;
subtype ODDSTATE is STATE range S1 to S3;
subtype SHORTWAVE is COLOR range COLOR'right downto GREEN;
```

声明的信号子类型LONGWAVE和SHORTWAVE都以类型COLOR为基类。有同样基类的信号被认为是同一类型。包含LOMGWAVE和SHORTWAVE子类型的信号被认为有基类COLOR。缺省的初始化按照子类型进行。子类型LONGWAVE的信号的缺省初始化值为RED，子类型SHORTWAVE的信号的缺省初始化值为VIOLET。类型SHORTWAVE的一些属性如下：

```
SHORTWAVE'right = GREEN
SHORTWAVE'high = VIOLET
SHORTWAVE'low = GREEN
SHORTWAVE'val(1) = ORANGE
SHORTWAVE'pos(RED) = 0
```

仔细研究这些属性，属性SHORTWAVE' right，SHORTWAVE' left，SHORTWAVE' low，SHORTWAVE' high指的是子类型的范围，然而属性SHORTWAVE' val和SHORTWAVE' pos参考的是基类型COLOR。位置编号通常参考基类型。

2) 枚举数据类型的用户定义属性。用户可以自己定义属性。在大多数情况下，用户定义的属性不影响仿真。用户定义的属性通常用来提供设计文档或者为其他设计工具提供信息。例如，用如下代码，设计者可以指定综合工具必须使用单热编码来实现状态机。综合工具可以从用户定义的STATE_ASSIGNMENT属性得到状态赋值信息。

```
type STATE_TYPE is (S1, S2, S3, S4, S5, S6, S7);
attribute STATE_ASSIGNMENT: String;
attribute STATE_ASSIGNMENT of STATE_TYPE: type is
"0000001 0000010 0000100 0001000 0010000 0100000 1000000 ";
```

定义用户自定义属性的通常格式如下：

```
attribute ATTRIBUTE_NAME: ATTRIBUTE_SUBTYPE;
```

关键字*attribute*后跟着用户选择的属性名。ATTRIBUTE_NAME属性可以是任何先前声明过的不是存取类型或文件类型的子类型（或类型）。在上一个例子中，STATE_ASSIGNMENT是属性名，而String（内置数据类型）是属性类型。

属性可以用下述形式和一个实体相连：

```
attribute ATTRIBUTE_NAME of ENTITY_NAME: entity_class is expression;
```

关键字*attribute*后跟着先前声明的属性名。关键字of分隔开属性名和带有该属性的实体名。冒号分隔开实体名和实体的类。在STATE_ASSIGNMENT的例子中，属性STATE_ASSIGNMENT和数据类型STATE_TYPE相连。实体类指示类型，属性可以是下列实体类：*entity*、*procedure*、*type*、*signal*、*label*、*architecture*、*function*、*subtype*、*variable*、*configuration*、*package*、*constant*和*component*。最后，关键词*is*将实体类型和计算ATTRIBUTE_SUBTYPE的表达式分隔开来。

在本例中，STATE_ASSIGNMENT是属性名，STATE_TYPE是实体名，实体类是type。

2. 数值数据类型——整型和实型

VHDL语言包括内置的数值数据类型Integer和Real，以及相应的操作，即加、减、乘和除。

图3-17给出了预定义的Integer和Real数据类型和子类型。Integer和Real数据类型的范围和实现有关。通常由目标机器的寄存器大小来决定。语言要求范围包括-214783647到+214783647。某些机器可以有扩展的范围。用户可以通过预定义的属性Integer' high和Integer' low来得到该范围。Positive和Natural是Integer类型的预定义子类型，使用在很多应用中。这些子类型的定义使用了预定义属性Integer' high。因而，该子类型可以用于任何机器，而与实现的Integer类型范围无关。这表明使用属性来进行声明比其他可能的方法更通用。现在，考虑如下用户自定义数值类型，（当考虑综合的模型时，将会见到使用受限的Integer类型而不使用Integer类型的重要性）。

```
type Integer is range _____________;
  -- Range is implementation dependent.
type Real is range _____________;
  -- Range is implementation dependent.
subtype Natural is Integer range 0 to Integer'high;
subtype Positive is Integer range 1 to Integer'high;
```

图3-17 预定义的数值数据类型

下面的例子是用户定义的受限数值数据类型的例子。

```
type COUNTER is range 0 to 100;
subtype LOW_RANGE is COUNTER range 0 to 50;
subtype MID_RANGE is COUNTER range 25 to 75;
subtype HIGH_RANGE is COUNTER range 50 to 100;
type REG is range 0 to 100;
type INDEXA is range 0 to 7;
type INDEXD is range 7 downto 0;
type PROBABILITY is range 0.0 to 1.0;
type ANGLE is range -90.0 to 90.0;
type TESTSCORE is range 100.0 downto 0.0;
```

VHDL分析程序通过寻找小数点来分辨Integer和Real类型。前面的例子只定义了7种不同的数据类型：COUNTER、REG、INDEXA、INDEXD、PROBABILITY、ANGLE和TESTSCORE。COUNTER和REG尽管范围相同，仍是不同的数据类型。不同数据类型的变量不可以混在一个表达式里。例如，如果变量A的类型是COUNTER，而变量B的类型为REG，

那么下面这样的写法就是错的：

```
A := A + B
```

数值数据类型的范围可以是升序或降序。例如，INDEXA的范围是升序的，而INDEXD的范围是降序的。然而，某些预定义的数据类型属性（如'left与'low）对于枚举数据类型是无法区别的，但对于降序的数值数据类型，则不是这种情况。下面的例子展示相似的几个概念，读者应仔细研究这些例子。

```
INDEXD'left = 7      INDEXD'low = 0
INDEXD'right = 0     INDEXD'high = 7
INDEXA'left = 0      INDEXA'low = 0
INDEXA'right = 7     INDEXA'high = 7
INDEXD'pred(6) = 5   INDEXD'leftof(6) = 7
INDEXD'succ(6) = 7   INDEXD'rightof(6) = 5
INDEXA'pred(6) = 5   INDEXA'leftof(6) = 5
INDEXA'succ(6) = 7   INDEXA'rightof(6) = 7
```

应该说明的是，对于升序的数据类型，（包括所有的枚举数据类型），属性*'left*和*'low*是相等的，属性*'right*和*'high*是相等的，属性*'pred*和*'leftof*是相等的，属性*'succ*和*'rightof*是相等的。同样，对于降序的数据类型，属性*'left*和*'high*是相等的，属性*'right*和*'low*是相等的，属性*'pred*和*'rightof*是相等的，属性*'succ*和*'leftof*是相等的。

3. 物理数据类型

*物理数据类型*是标量数值数据类型，带有相关单位的系统，用来表示物理测量的实体对象。如时间、长度、电压、电流等等。唯一预定义的物理类型是时间，其定义如下：

```
type Time is range _____________________________
  units
    fs;             -- femptosecond
    ps = 1000fs;    -- picosecond
    ns = 1000ps;    -- nanosecond
    us = 1000ns;    -- microsecond
    ms = 1000us;    -- millisecond
    sec= 1000ms;    -- second
    min= 60 sec;    -- minute
    hr = 60 min;    -- hour
  end units;
```

其范围与实现无关。给出的第一个单位（fs）称为基本单位，所有其他的单位必须是基本单位的整数倍，并且在‘type’那一行定义的范围之内。而且，任何类型为Time的变量，其实际计算出的时间值，必须是基本单位的整数倍，实际上，基本单位是一个不可继续分割的单元。基本单位不可能有小数部分。例如，2.5ps 是合法的，因为2.5ps=2500fs，而2.250326ps不合法，因为它在转化为fs时，不是整数（2.250326ps=2250.326fs）。

实现将限制物理类型的范围，任何实现必须声明任何范围含在-2，147，483，647到+2，147，483，647的范围之间的任何物理类型。如果满足了这个限制，任何用户定义的数据类型至少有三个单位是其次小单位的1000倍。下面是用户定义物理数据类型的例子。

```
-- Electrical Resistance
  type RESISTANCE is range 0 to Integer'high
    units
      NOHM;                  -- Base unit is nano-ohm.
      UOHM   = 1000 NOHM;    -- Microohm
      MOHM   = 1000 UOHM;    -- Milliohm
```

```
        OHM    = 1000 MOHM;     -- Ohm
        KOHM   = 1000 OHM;      -- Kilohm
        MEGOHM= 1000 KOHM;      -- Megohm
      end units;
  -- Electrical Power
    type POWER is range 0 to 1e9
      units pW;                 -- Pico-watt
            nw = 1000 pW;       -- Nano-watt
            uW = 1000 nW;       -- Micro-watt
      end units;
  -- Frequency
    type FREQUENCY is range 0 to 1E9
      units Hz;                 -- Hertz
            KHz = 1000 Hz;      -- Kilohertz
            MHz = 1000 KHz;     -- Megahertz
      end units;
```

在上面的例子中，物理数据类型RESISTANCE不能够工作于上述最小范围之下，因为一个可用的范围应该包括欧姆、千欧和兆欧。

3.4.3 复合数据类型

复合数据类型有复合的值。例如，一维整数数组的值可以为（15，35，2，956）。这个复合值由几个组件组成，表示一个数组，其中各个值之间有一定的关系。

1. 数组

数组是相同类型的数据集合在一起而形成的复合数据类型。数组元素是同类的。例如，有两种预定义的数组数据类型。数据类型String是字符元素的数组。使用它来产生用于打印或由其他程序使用的文本。数据类型Bit_Vector是位的数组。用来描述存在寄存器中的数据或通过并行总线传输的数据。定义为：

```
type String is array (Positive range <>) of Character;
type Bit_Vector is array (Natural range <>) of Bit;
```

注意 下标范围放在保留字*array*后的括号里。数据类型String的下标范围是数据类型Positive。括号<>表示没有范围限制。在这种情况下，用户在声明String类型的对象时，必须指定下标范围，对该范围的限制是下标范围必须是正整数，它的灵活性相当大。同样，类型Bit_Vector的下标范围也没有限制（除了范围必须是自然数）。类型Bit_Vector的对象可以有下标0，但类型String不可以有下标0。

一些用户定义的数组数据类型如下：

```
type STD_LOGIC_VECTOR is array (Natural range <>) of STD_LOGIC;
type REGISTER_32_Bit is array (31 downto 0) of Bit;
     -- Descending index range.
type COUNTER_16_Bit is array (0 to 15) of Bit;
     -- Ascending index range.
subtype BITVECT3 is Bit_Vector(0 to 2);
type COLOR_COUNT is array (COLOR range <>) of Natural;
type ARRAY_OF_COLORS is array (Natural range <>) of COLOR;
type ROM_TYPE is array(Natural range <>)of REGISTER_32_Bit;
type ARRAY_2D is array (Natural range <>, Natural range <>) of Bit;
```

类型STD_LOGIC_VECTOR是无限制数据类型，当使用IEEE 9值逻辑系统时用来表示寄存器和总线，如用于综合的模块中。数据类型REGISTER_32_BIT是一维32位的受限数组，下

标从31到0（从左到右）。例如，如果REGISTER_32_BIT类型的一个数据项的值为：

B"1111_1101_0101_1110_0000_0101_1100_0010"

则位31（最左边的位）的值为‘1’，位25的值为‘0’，位2的值是‘0’，位0（最右边的位）的值是‘0’，类型RECISTER_32_BIT的下标范围是降序的。

同样，类型COUNTER_16_BIT是受限的16位一维数组，位的位置从左到右为0到15，是升序的。如果类型COUNTER_16_BIT的对象的值为：

B"1011_0010_1011_0010"

则位0（最左边的位）值为‘1’，位4值为‘0’，位15值为‘0’（最右边的位）。

因为BIT_VECTOR是无限制数据类型，可以用不同的下标范围定义子类型。BITVECT3是类型BIT_VECTOR的子类型，其升序的下标范围从0到2。

数据类型COLOR_COUNT是自然数的数组。下标类型是无范围限制的类型COLOR。当声明类型为COLOR_COUNT的对象时，可以由用户随意选择范围。假设某个对象的范围选择为从RED到YELLOW，则该数组包括3个自然数，下标顺序是RED、ORANGE、YELLOW。例如，数组的值可以是（24，35，0），在这种情况下，下标为RED的元素是24，下标为ORANGE的元素是35，下标为YELLOW的元素是0。

同样，数组类型ARRAY_OF_COLOR是COLOR的数组。下标范围无限制，但必须是自然数。假设下标范围声明为3到5，如果数组的值是（BLUE, ORANGE, VIOLET），则下标为3的元素是BLUE，下标为4的元素是ORANGE，下标为5的元素是VIOLET。

有两种方法来声明二维数组，数组类型ROM_TYPE是数组的数组。每个元素都是REGISTER_32_BIT类型，换言之，ROM_TYPE是32位字的一维数组。一个类型是ROM_TYPE的结构的典型值如下，其中包含4个字：

```
(X"2F3C_5456", X"FF22_A5B9", X"9900_AD51", X"FFFF_FFFF").
```

换言之，数组中每个元素都是32位的大小。

相反，ARRAY_2D是每个元素都是类型BIT的二维数组。如果声明一个具有5行10列的对象，则该结构的一个典型值为：

```
(1, 0, 1, 1, 0, 0, 0, 1, 1, 1,
 0, 0, 1, 1, 1, 0, 1, 1, 0, 1,
 1, 1, 1, 0, 1, 0, 1, 0, 0, 1,
 1, 1, 1, 1, 0, 0, 0, 1, 1, 0,
 0, 0, 0, 0, 1, 1, 1, 0, 0, 1)
```

在这种情况下，数组的每个元素是单个位，关于数组元素的处理在实体声明部分讨论。

2. 数组的属性

数组有许多有用的内置属性。属性A的范围是返回名为A的一维数组的下标范围的函数。这使得可以创建一个进程，用以检查数组的元素而无需知道数组的实际范围。例如，下面的进程计算类型为Bit_Vector的一维数组DBUS中‘1’的个数。范围可以是升序，也可以是降序。该进程工作时不必考虑下标范围内元素的个数。

```
PROCESS_RANGE: process (DBUS)
    variable COUNT3: Integer := 0;
  begin
    COUNT3 := 0;
```

```
    for I in DBUS'range loop
      if DBUS(I) = '1' then
        COUNT3 := COUNT3 + 1;
      end if;
    end loop;
  end process;
```

属性A的上界是返回数组A下标范围内最大值的函数。属性A的下界是返回数组A下标范围内最小值的函数。同样，A的右界和A的左界分别返回A下标范围之内数组最右和最左之值的函数。A的长度是返回数组A下标范围之中的元素个数的函数。例如，如果DBUS声明如下：

```
signal DBUS: Bit_Vector (15 downto 0);
```

则数组属性有下列值：

```
DBUS'right = 0
DBUS'left = 15
DBUS'high = 15
DBUS'low = 0
DBUS'length = 16
```

下面的进程利用上述的两个属性来计算信号DBUS中'1'的个数。无论其下标值范围如何，该进程对于任何类型为Bit_Vector的信号都有效，范围可以是升序，也可以是降序的。

```
PROCESS_LENGTH: process (DBUS)
    variable COUNT2: Integer := 0;
  begin
    COUNT2 := 0;
    for I in 0 to DBUS'length-1 loop
      if DBUS(DBUS'low+I) = '1' then
        COUNT2 := COUNT2+1;
      end if;
    end loop;
  end process;
```

所有的数组属性都可适用于多维数组。多维数组属性的通用形式是A'attribute(n)，其中，属性用于第n个下标元素。

3. 记录

记录类型是元素类型不同的复合数据类型。即，元素可以有不同的类型。与数组类型类似，元素形成一个有序的列表。例如，考虑如下用来保存日期的记录类型。

```
type MONTH_NAME is (JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY,
AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER);

type DATE is record
   DAY  : Integer range 1 to 31;
   MONTH: MONTH_NAME;
   YEAR : Integer range 0 to 3000;
end record;
```

类型DATE的一个典型值为：

```
(16, AUGUST, 1943)
```

其中DAY=16，MONTH=AUGUST，YEAR=1943。

3.4.4 存取类型

存取类型是在动态存储应用中用来表示存储空间的分配与回收的类型，如链表和树。本

书对存取类型不作讨论。

3.4.5 文件类型

文件类型是在主机系统环境中，用来定义表示文件内容的对象的类型。可以用文件来保存VHDL程序需要的数据。VHDL模型的测试向量一般存在文件里。后面章节将对这些类型进行讨论。

3.4.6 类型标记

有时候在VHDL中，当重载发生时，正确判断程序员究竟指示哪种类型是不可能的。例如，考虑如下情况：

```
type TRISTATE is ('Z', '0', '1');
type MVL is ('0', '1', 'Z');
```

表达式（'Z'<'0'）是真还是假？因为字符'Z'和'0'被重载，仿真程序无法判别是使用类型TRISTATE（该情况下，表达式为真）还是使用类型MVL（该情况下，表达式为假），可以通过写类型标记来解决上述问题，如下所示：

```
(MVL'('Z') < MVL'('0'))
```

如果根据上下文无法明显得出类型，使用类型名后跟撇号（在这里叫作上下文中的类型标记），其值用括号括起来。因而在上述表达式中就不存在二义性了。

3.5 数据对象

数据对象是可以保存所声明类型的值的容器（container）。

3.5.1 对象的分类

表3-1给出了数据对象的三个类。

表3-1 数据对象的分类

类	说 明
常量	值在编译时被指定，不可由VHDL语句改变的数据对象
变量	当前值可以由VHDL语句改变的数据对象
信号	具有时间量纲的数据对象。VHDL语句可以将将来的值赋给信号而不影响当前值。使用波形（waveform），可以赋一串将来的值。当前的值不可改变

VHDL中的变量与其他高级编程语言中的变量类似，如C、C++和Java，变量只可以在进程或子程序内部定义。进程中定义的变量对于该进程是局部的。在进程内部，变量是静态的，它在仿真开始时进行初始化。然而，在子程序内部定义的变量是动态的，该值在一次调用到另一次调用之间并不保持原有的值，子程序中的变量在每次调用时重新初始化。

信号用来表示电路中实际物理数据线上的数据值。在实际系统中，例如电压信号，值不可能立即发生改变。因而，为了正确表示物理特性，VHDL信号的值也不立即发生改变。因此，赋给信号的新值直到将来的某个时刻才能够生效。这个信号改变值的时刻可以显式地定义，如果没有对信号改变指定时间，缺省的时间是一个叫做*delta time*的无限小的时间。一个重要的概念是，VHDL信号预定的改变决不在当时发生，而是一直延时到将来的某个时刻。

记住这个原则能够帮助理解信号的特性并减少错误。可以使用类似于波形的结构给VHDL信号赋予一串值。

3.5.2 数据对象的声明

数据对象必须在赋值语句中被引用之前定义。本节中声明对象例子使用前几节中定义的类型。这些声明的类型在后面的小节中多次用来书写各种各样的语句。

1. 常量的声明

通常，常量的值必须在声明时确定。使用本章前几节中定义的数据类型的常量值并把它们指定如下：

```
constant ALPHA_LEVEL: PROBABILITY := 0.75;
constant INITIAL_STATE: STATE := S0;
constant MID_COLOR: COLOR := GREEN;
constant HIGH_IMPEDANCE: TRISTATE := 'Z';
constant END_MARKER: Character := DEL;
constant PI: REAL := 3.14159;
constant C1K: Integer := 2 ** 10;
  -- Operator ** is exponentiation.
constant MESSAGE1: STRING := "Happy Birthday";
```

注意 常量名与类型名通过“:”分隔开。类型名与常量值通过分界符“:=”分开。通常，VHDL语句用“;”结束。

2. 变量的声明

变量名必须在用于表达式之前进行声明。变量只可在进程或子程序内部进行声明。变量声明通常指明其类型。变量的初始值显式地在声明中给出。如果对变量没有指定初始值，缺省值将是声明类型范围之中最左边的那个值。下面的例子是声明标量变量的例子：

```
variable STAR_COLOR, HAT_COLOR: COLOR;
variable BETA_LEVEL: PROBABILITY := 0.0;
variable CURRENT_STATE: STATE := INITIAL_STATE;
variable A, B, C, D: Bit;
variable R1, R2, R3, R4: REG;
variable A_COUNT, B_COUNT: COUNTER := 10;
variable TRANSMIT_FREQUENCY: FREQUENCY;
variable EVENT_Timer: Time := 1 ns;
variable MIDDLE_INITIAL: Character;
variable X: Integer := 0;
variable RADIUS: Real := 0.0;
```

为了声明同一类型的多个变量，用分界符“,”来分隔变量名，用“:”号将变量名列表与类型名分隔开，如果指定了初始值，用“:=”将指定值与类型名分开。

变量STAR_COLOR的缺省初始值为：

```
COLOR'left = RED
```

同样，A、R1、TRANSMIT_FREQUENCY和MIDDLE_INITIAL的缺省初始值分别为‘0’、0、0 Hz和NUL。

可以通过显式地声明初始值来取代缺省的初始值。例如，变量BETA_LEVEL的初始值显式地声明为0，尽管缺省值同样是0。CURRENT_STATE的初始值显式指定为常量INITIAL_STATE即S0，A_COUNT的初始值被显式地声明为10，ENENT_Timer的初始值声明为1 ns。

下面是一些声明数组变量的例子：

```
variable REG1: Bit_Vector (15 downto 0) := X"F5A2";
variable REG2: COUNTER_16_Bit := B"1111_0101_1010_0010";
variable COLOR_STRENGTH: COLOR_COUNT(ORANGE to INDIGO) :=
  (GREEN=> 15, BLUE | YELLOW => 20, others => 10);
variable MAP_COLOR:
  ARRAY_OF_COLORS(5 to 7) := (VIOLET, ORANGE, BLUE);
variable ROM_A: ROM_TYPE(0 to 7) :=
       (0=> X"FFFF_FFFF",
        5=> X"2222_CCCC",
  others=> X"0000_0000");
variable MATRIX_3X4: ARRAY_2D(0 to 2, 0 to 3) :=
     ( ('0','1','0','0'),
       ('1','1','0','0'),
       ('1','0','1','0'));
```

当变量的数据类型无下标范围限制时，该变量实际的下标范围用圆括号括起来，并直接放在变量类型名的后面。例如，变量REG1是元素类型为BIT的数组类型，且为降序。变量REG2是元素类型为BIT的数组类型，为升序，范围是0到15，该范围是在受限数据类型COUNTER_16_BIT的声明中预先指定的。REG1和REG2都通过将初始值置于“:=”之后进行初始化。例如，初始化值为REG1(15)='1'，REG1(0)='0'，REG2(0)='1'和REG2(15)='0'。

变量COLOR_STRENGTH是元素类型为自然数，下标类型为COLOR的数组。同样，因为类型COLOR_COUNT下标范围无限制，COLOR_STRENGH的实际范围通过类型名后紧跟的括号来指定。COLOR_STRENGTH通过命名相关列表（named associated list）来初始化。初始值以任意顺序列表，用“=>”与将要赋的值分隔开。赋值用“,”分隔。将同一值赋给多个元素，可以通过用分界符“|”分隔各下标名来实现。例如，下标为BLUE和YELLOW的元素都被赋于值20，保留字*others*用来指所有未赋值的元素的下标值。在本例中，下标为ORANGE和INDIGO的元素通过*others*子句赋予值10，最后，初始值的整个列都括在括号之内。变量MAP_COLOR是元素类型为COLOR、下标类型为Natural的数组类型。初始值用位置相关列表（*positional association list*）来指定。这种方法中，初始值按照位置顺序从左至右排列，且用括号括起来。例如，MAP_COLOR（5）的初始值是VIOLET。

一个集合是用来创建复合数据类型的值的一个元素相关列，它可以是命名相关列表或位置相关列表。在前面的例子中，变量ROM_A和COLOR_STRENGTH用命名相关列表的集合进行初始化。变量MATRIX_3X4和MAP_COLOR使用位置相关列表的集合进行初始化。

访问数组元素是通过将下标名置于括号中紧跟在数组名之后实现的。使用这种约定，一些数组元素的初始值是：

```
MAP_COLOR(5) = VIOLET
MAP_COLOR(6) = ORANGE
MAP_COLOR(7) = BLUE
COLOR_STRENGTH(YELLOW) = 20
COLOR_STRENGTH(ORANGE) = 10
REG2(0) = '1'
REG1(15) = '1'
ROM_A(0) = X"FFFF_FFFF"
MATRIX_3X4(0,0) = '0'
MATRIX_3X4(1,0) = '1'
MATRIX_3X4(2,3) = '0'
```

记录声明的例子包括：

```
constant BIRTH_DATE: DATE := (16, AUGUST, 1943);
```

```
variable CURRENT_DATE: DATE := (MONTH => JANUARY, DAY => 1, YEAR => 2000);
```

常量BIRTH_DATE使用位置相关列表来定义初始值。变量CURRENT_DATE使用命名相关列表来定义初始值。

3. 信号的声明

信号声明与变量声明相似。信号不可以在进程或子程序内定义。信号可以在实体声明中或在构架声明的声明区域中作为端口声明，信号声明的例子包括：

```
signal X1, X2, X3, X4, X5: Bit;
signal SR1, SR2, SR3, SR4: REG;
signal DOWN_COUNT: COUNTER := COUNTER'right;
signal ROM_B: ROM_TYPE(0 to 3) :=
  (X"FFFF_FFFF", X"2222_CCCC", X"A03B_0020", X"ABCC_506C");
signal DATE_REGISTER: DATE := (1, JANUARY, 1900);
```

X1的初始值缺省时为'0'。同样，SR1的初始值缺省时为'0'，然而DOWN_COUNT的初始值是100。ROM_B(0)的初始值是X"FFFF_FFFF"。

所有的端口必须是信号。

3.6 语句

本节介绍各种语句。VHDL有两种语句。顺序语句在进程和子程序中用来描述算法。除非遇到分支语句，它们是顺序执行的，这与其他过程性语句类似，如C。并发语句用在结构体中对信号建模。不像典型的编程语言，这些语句并不是按照编写的顺序执行。相反，并发语句只有在信号改变时才被触发。而且，所有的并发语句在仿真开始的时候执行一次。并发语句在结构体中的顺序并不重要。有些语句的类型既是顺序的，又是并发的。图3-18给出了语句的分类。

顺序语句	并发语句
Assertion	Assertion
Signal assignment	Signal assignment
Procedure call	Procedure call
Variable assignment	Process
If...then...else	Block
Case	Component instantiation
Loop	Generate
Wait	
Null	
Next	
Exit	
Return	

图3-18 顺序语句和并发语句

3.6.1 赋值语句

变量和信号的值可使用赋值语句来改变。

1. 变量赋值语句

变量赋值语句使得变量的当前值由一个表达式所确定的新值代替。变量立即接受新值。

复合分界符“:=”用来定义变量赋值。变量赋值语句的一些简单例子如下，这里使用了前节定义的一些变量。

```
STAR_COLOR := GREEN;
HAT_COLOR := MID_COLOR;                -- Assigns value GREEN to HAT_COLOR.
BETA_LEVEL := 0.5;
MAP_COLOR(5) := YELLOW;
COLOR_STRENGTH(BLUE) := 100;
ROM_A(6) := X"2AA6_344F";
ROM_A(5) := ROM_A(0);
MATRIX_3X4(1,2) := '1';
MATRIX_3X4(2,2) := MATRIX_3X4(0,1);
CURRENT_STATE := S4;
A := '1';
R1 := REG'(50);                        -- Need type mark because R1 is type
                                       -- REG not type Integer.
MIDDLE_INITIAL := Character'val(68);   -- Assigns value 'D'
                                       -- to MIDDLE_INITIAL
CURRENT_DATE := BIRTH_DATE;
CURRENT_DATE := (10, JANUARY, 1992);
CURRENT_DATE.YEAR := 1993;             -- Note that the "period" is used to
                                       -- separate the variable name from the
                                       -- element name.
Z := CURRENT_DATE.DAY;
R1 := CURRENT_DATE.DAY;                -- ILLEGAL because R1 is not type
                                       -- Integer.
```

2. 信号赋值语句

信号赋值语句给信号赋予新值，信号将在将来的某个时刻接受该值。信号赋值语句不改变信号的当前值。如果没有指定时间，缺省值是称作delta time的无穷小的时间。为了区分信号赋值和变量赋值语句，使用“<=”分界符，而非“:=”。下面是一些信号赋值语句的例子。

```
X1 <= '1' after 10 ns;
SR1 <= 5 after 5 ns;
X2 <= '0' after 10 ns, '1' after 20 ns, '0' after 30 ns;
X5 <= '1';
```

图3-19给出了上面的信号赋值语句产生的时序，假设所有的语句都在时间t执行。X1在时间t的当前值假设为‘0’，SR1在时间t的值假设为10，X2在时间t的值假设为‘1’，X5的值假设为‘0’。第一个语句引起信号X1在t+10ns时变为‘1’。同样，第二个语句使SR2在时间t+5ns从10变到5。给信号X2赋值的语句的类型是波形（waveform），引起信号X2将来的三次变化。因为没有显式地指定信号X5改变的时间，改变将在时间t+delta时发生。

给信号赋值的基类型必须与声明的信号的基类型相同。例如，信号赋值语句为：

```
DOWN_COUNT <= 50.5 after 5 ns;
```

出错，因为DOWN_COUNT的基类型是Integer，而值50.5是实型。

3. 信号驱动器

如果在一个进程中有一个或多个信号赋值语句给信号X安排将来某个时刻的值，VHDL仿真程序就为信号X创建一个与该进程相关的单值保持器，称作信号驱动器。驱动器维护一个为信号X赋值的调度值的有序列表。每个所调度信号的赋值称作一个事务（*transaction*）。由事务赋值给信号X的一个新值可以与信号的当前值相同，也可以不同。如果信号由于一个事务而产生了信号变化，则称发生了一个事件（*event*）。在两个事务之间，驱动器保持信号的值，

VHDL仿真程序从驱动器中读出当前信号的值。

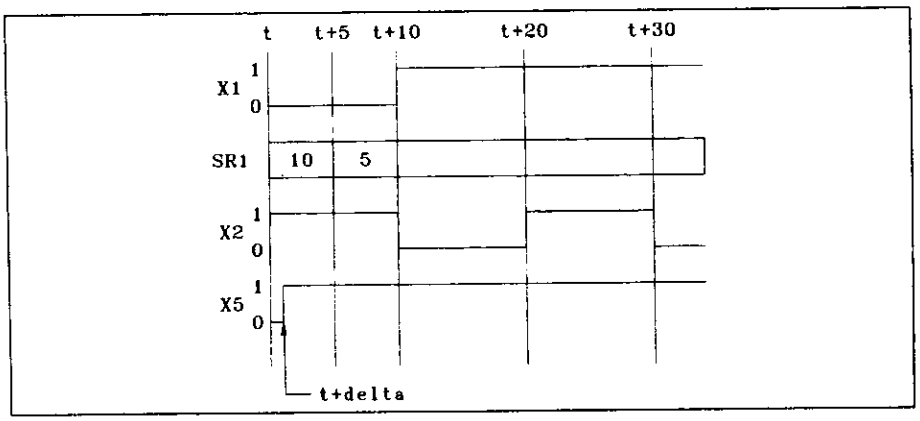


图3-19 选择信号赋值语句的时序图

如果多个进程都包括对同一个信号的信号赋值语句，VHDL仿真程序就针对每个这样的进程，为信号X创建一个单独的驱动器。因而，在特定的时间可能会有多个驱动器，对应于每个不同的进程，来安排信号X的值。很明显，如果同时有多个驱动器对信号X赋予不同的值，则必须存在一种方法来决定信号X的正确值。该问题是通过对信号X指定一个*判决函数*，来对所有两个可能同时给信号X代入的值进行判决，得出一个判决值。信号X则称为*判决信号*。判决函数可以和数据类型或信号相关联。下面这些例子说明了如何指定一个判决信号。

```
signal X1: WIRED_OR Bit;          --Example 1
subtype RESOLVED_Bit is WIRED_OR Bit;
signal X2: RESOLVED_BIT;
signal Y1: RESOLVED STD_ULOGIC; --Example 2
subtype STD_LOGIC is RESOLVED STD_ULOGIC;
signal Y2: STD_LOGIC;
```

第一个例子的第一个语句直接声明信号X1是类型BIT的判决信号，其判决函数是WIRED_OR。判决函数名和类型名之间用空格符来分隔，第二个语句将判决函数与数据子类型RESOLVED_Bit关联，该子类型的基类型是Bit。声明的类型为子类型RESOLVED_Bit的信号都将自动带有判决函数WIRED_OR。第三个语句声明信号X2有子类型RESOLVED_BIT。

第二个例子重复了IEEE标准逻辑中的进程。首先，信号Y1是直接声明的类型为STD_ULOGIC的判决信号，其判决函数为RESOLVED。然后判决函数RESOLNED与类型STD_ULOGIC相关联，从而形成子类型STD_LOGIC，信号Y2被声明为具有子类型STD_LOGIC，而且，它因而继承了判决函数RESOLVED。在综合的模型中，大部分标量信号和变量被声明为具有子类型STD_LOGIC。

当信号X的不同信号驱动器在一个特定的时间给信号X指定值时，仿真程序执行判决函数为信号X确定正确的值。编写判决函数的要求在第5章加以介绍。

4. 信号的属性

VHDL语言为信号预定义了几种属性。S'*active*是布尔函数。若S是标量信号，如果当前的仿真周期中在指定的信号上有一个事务，则S'*active*返回TRUE。若S是一个复合信号，如果在当前的仿真周期中S的任何一个标量成员上有一个事务，则S'*active*返回TRUE。同样，S'*event*

也是一个布尔函数。若S是标量信号，如果当前仿真周期之中S信号上有一个事件，则S'*event*返回TRUE。若S是复合信号，则在当前仿真周期之内，S的任一个标量成员上有一个事件，则S'*event*返回TRUE。

```
X2'active        -- An expression that is TRUE when a transaction occurs on
                 -- signal X2 during the current simulation cycle.
X3'event         -- An expression that is TRUE when an event occurs on
                 -- signal X3 during the current simulation cycle.
```

S'*stable(n)*定义了一个布尔信号，在时间t，如果信号S在前n个单位时间内没有事件，则为TRUE。同样，S'*quiet(n)*定义了一个布尔信号，在时间t，如果信号S在前n个单位时间内没有事务，则为TRUE。

```
X1'stable(5 ns)  -- An expression that is TRUE when there have been no
                 -- events on signal X1 during the past 5 ns.
X4'quiet(5 ns)   -- An expression that is TRUE when there have been no
                 -- transactions on signal X4 during the past 5 ns.
X2'stable        -- Either expression is TRUE unless there
X2'stable(0 ns)  -- is an event on X2 during the current simulation cycle.
X3'quiet         -- Either expression is TRUE unless there
X3'quiet(0 ns)   -- is a transaction on signal X3 during the current
                 -- simulation cycle.
```

注意 两个属性'*stable*和'*quiet*的缺省时间是0ns。

S'*transaction*定义了类型BIT的信号，若在仿真周期中信号S有事务发生时则翻转，信号S'*transaction*上的一个事件表明在信号S上有一个事务。

S'*last_active*是一个函数，它返回从信号S的上一次事务之后到当前时刻所经过的时间，同样，S'*last_event*是一个函数，它返回从信号S的上一次事件之后所经过的时间。

```
X1'last_event    -- An expression that has a value equal to the amount of
                 -- time that has elapsed since the last event on signal X1.
X2'last_active   -- An expression that has a value equal to the amount of
                 -- time that has elapsed since the last transaction
                 -- on signal X2.
```

S'*last_value*是一个函数，它返回在上一次信号改变之前信号S的值。

```
X4'last_value    -- An expression that has a value equal to the value of
                 -- signal X4 immediately before the last event on
                 -- signal X4.
```

S'*delayed*定义了一个信号，该信号等价于给定信号延迟一个指定时间后的信号。

```
X1'delayed(5 ns) -- An expression that defines the current value of a
                 -- signal that is equal to signal X1 delayed by 5 ns.
```

图3-20给出了类型为BIT的信号X1的时序图和该信号的一些属性。信号X1在时间5、10、20和30ns上有事件。信号X1'*delayed*(8ns)也属于类型BIT，它只是信号X1在时间上提前了8ns的副本。因而信号X1'*delayed*(8ns)在时间13、18、28和38ns时有事件发生。信号X1'*stable*(8ns)是布尔类型，其初始值为TRUE。该信号在时间t=5时因为X1的事件而变为FALSE。在t=18ns时因为信号已经稳定了连续8ns（即在这之前的8ns无事件发生）而从FALSE变为TRUE。信号X1'*stable*(8ns)在t=20ns时再次变为FALSE，是因为那时的信号X1上有事件。在t=28ns，信号X1又保持稳定了8ns，因而信号X1'*stable*(8ns)又变为TRUE。读者可

以验证在信号X1'*stable*(8ns)上剩余的事件。信号X1'*stable*(8ns)在时间18，20，28，30和38ns上都有事件。

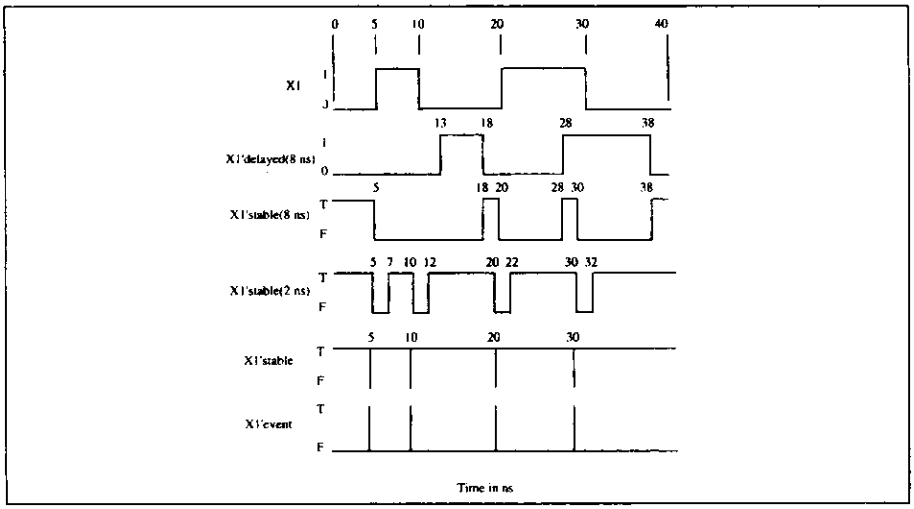


图3-20 信号X1的时序图及其属性

同样，信号X1'*stable*(2ns)初始化为TRUE并在t=5ns时因为信号X1上的事件而变为FALSE。在t=7ns时，信号X1'*stable*(2ns)从FALSE变为TRUE，因为信号X1已经连续稳定了2ns。读者可以验证X1'*stable*(2ns)上的所有事件。信号X1'*stable*在大部分时间里都为TRUE。只在时间5、10、20和30ns对应于信号X1上的事件而变为FALSE。

属性X1'*event*与图3-20中的其他属性不同，因为它不是一个信号。属性X1'*event*是一个函数属性，如果在当前的仿真周期中信号X1上有事件，则计算得到TRUE。注意：X1'*event*和X1'*stable*在所有的时间上都有互补值。

下面的表达式说明了信号属性的各个方面。注意：既然X1'*stable*和X1'*delayed*是信号，它们同样也具有属性。

```
X1'delayed(8 ns)             -- this expression has value '1' at t=15ns
X1'stable(8 ns)              -- this expression has value FALSE at t=15ns
X1'stable(2 ns)              -- this expression has value TRUE at t=15ns
X1'event                     -- this expression has value FALSE at t=15ns
X1'last_event                -- this expression has value 5ns at t=15ns
X1'last_value                -- this expression has value '1' at t=15ns
X1'stable'last_event         -- Value is 5ns at t=15ns
X1'stable(2 ns)'last_event   -- Value is 3ns at t=15ns
X1'delayed(8 ns)'event       -- Has value TRUE only at times
                             -- 13,18,28, and 38 ns
```

3.6.2 操作符和表达式

VHDL语言有大量的内置操作符。图3-21给出了VHDL操作符和它们的优先级关系。每行的操作符集合有相同的优先级。任一行的操作符集合的优先级高于下面行中的所有操作符，低于上面行中的所有操作符。例如，关系操作符的优先级高于逻辑操作符，低于加法操作符。

1. 逻辑操作符

逻辑操作符定义了类型为BIT或布尔型的变量和信号的逻辑运算。内置逻辑操作符不可用

于任何用户定义的数据类型。逻辑操作符也可以用于元素类型是BIT或布尔类型的数组，在这种情况下，逻辑运算施加于数组中的每一个成员。注意：所有的逻辑操作符有相同的优先级。如果改变运算次序，必须使用括号。例如，信号赋值语句：

Miscellaneous Operators	** abs not
Multiplying Operators	* / mod rem
Signing Operators	+ -
Adding Operators	+ - &
Relational Operators	= /= < <= > >=
Logical Operators	and or nand nor xor

Operators on the same line have equal precedence. For example, all logical operators have equal precedence. All operators on a given line have higher precedence than all operators below that line. Miscellaneous operators have the highest priority and logical operators have the lowest priority.

图3-21 VHDL操作符的优先级关系

```
X5 <= X1 and X2 or X3 and X4;
```

非法，会产生分析程序错误，表达式：

```
X5 <= (X1 and X2) or (X3 and X4);
```

是合法的，图3-22给出了对应的电路。

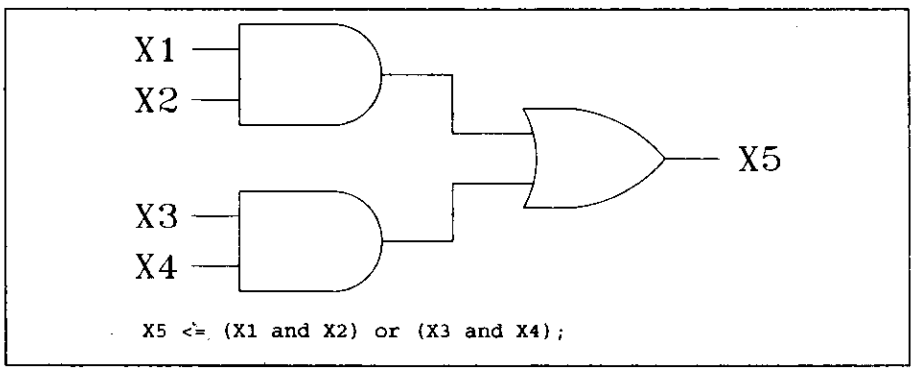


图3-22 VHDL语句的逻辑电路

逻辑操作符*and*、*or*和*xor*是可结合的，例如：

```
(X1 and X2) and X3 = X1 and (X2 and X3)
```

因而，下面这种不带括号的信号赋值：

```
X5 <= X1 and X2 and X3
```

是合法的，它正确实现了3输入与门的功能。然而*nand*和*nor*操作符是不可结合的。因此，3输入NAND操作不可以写成：

```
X5 <= X1 nand X2 nand X3.    -- Erroneous format.
```

上述的表达式将产生分析程序错误。事实上，尽管表达式：

```
(X1 nand X2) nand x3
```

是合法表达式，它并不能正确表示一个3输入NAND门，因为：

```
(X1 nand X2) nand X3 = (X1 and X2) or (not X3)
```

并非是要求的结果。一个3输入NAND门的正确表示如下：

```
X5 <= not (X1 and X2 and X3).
```

2. 关系操作符

关系操作符的定义见表3-2。

表3-2 关系操作符

符 号	操 作
=	等于
/=	不等于
>	大于
>=	大于或等于
<	小于
<=	小于或等于

关系操作符用来在表达式中比较两个对象的值。任何包括关系操作符的表达式的值都是布尔类型的。关系操作符可以用于任何内置或用户定义的标量数据对象。进行比较的对象必须有相同的基类型。对于枚举数据类型，比较的是值的位置序号。下面的例子包括一些典型的关系运算表达式，涉及本章前面定义的常量、变量和信号。

```
BETA_LEVEL := 0.5;                      -- Variable assignment statement.
ALPHA_LEVEL := 0.75;
(BETA_LEVEL < ALPHA_LEVEL)              -- Expression that is TRUE.
STAR_COLOR := GREEN; HAT_COLOR := VIOLET;
(STAR_COLOR > HAT_COLOR)                -- Expression that is FALSE.
R1 := REG'(3);
(COLOR'val(R1) = STAR_COLOR)            -- Expression that is TRUE.
CURRENT_STATE := S3;
(CURRENT_STATE /= INITIAL_STATE)        -- Expression that is TRUE.
(CURRENT_STATE > INITIAL_STATE)         -- Expression that is TRUE.
(STATE'right < CURRENT_STATE)           -- Expression that is FALSE.
(STATE'left = INITIAL_STATE)            -- Expression that is TRUE.
MIDDLE_INITIAL := 'G';
(MIDDLE_INITIAL >= '4')                 -- Expression that is TRUE.
(MIDDLE_INITIAL <= 'g')                 -- Expression that is TRUE.
(Character'pos(MIDDLE_INITIAL) <= 40)   -- Expression that is FALSE.
SR1 <= 10;                              -- Signal assignment statement.
SR2 <= 25;                              -- Signal assignment statement.
DOWN_COUNT <= 50;                       -- Signal assignment statement.

(DOWN_COUNT <= 50)                      -- Expression that is TRUE.
B1 <= (DOWN_COUNT <= 50);               -- Assigns the value TRUE to a
                                        -- signal, B1, of type Boolean.
(SR2 <= SR1)                            -- Expression that is FALSE.
(DOWN_COUNT <= SR2)                     -- Illegal expression because objects
                                        -- have different base types.
```

注意到符号“<=”在语言中有两个合法的意思：可以表示“小于或等于”，也可以代表信号赋值，需要根据符号的上下文来确定其意义。

3. 加操作符

加操作符的定义见表3-3。

表3-3 加操作符

符 号	操 作
+	同一基类型的数值操作数的算术加法
-	同一基类型的数值操作数的算术减法
&	同一基类型的一维数组的合并运算

如果未被重载，则算术加法和减法只适用于类型Integer和Real。数据类型严格强制执行这些运算。试图加减两个不同基类型的操作数将引起错误——即使两个基类型的范围是相同的。见下面的例子：

```
BETA_LEVEL := 0.2;                          -- Variable assignment statement.
SR1 <= 10;                                  -- Register assignment statement.
SR2 <= 25;                                  -- Register assignment statement.
DOWN_COUNT <= 50;                           -- Register assignment statement.
SR3 <= SR2 - SR1;                           -- SR3 is assigned the value 15.
BETA_LEVEL := BETA_LEVEL + BETA_LEVEL;      -- BETA_LEVEL is assigned
                                            -- the value 0.4
DOWN_COUNT <= DOWN_COUNT - SR1;             -- Illegal expression because the
                                            -- operands do not have the same
                                            -- base type.
```

合并两个相同基类型的一维数组产生一个新的一维数组，结果的长度是两个操作数的长度之和。新数组的元素包括左边操作数的元素（以从左到右的顺序），后面跟着右边操作数的元素（也是从左到右）。左边界是左边操作数的左边界，除非左操作数为空数组，在此情况下，合并的结果就是右操作数。方向是左边操作数的方向，除非左操作数是空数组，在此情况下，方向是右操作数的方向。见下面的例子：

```
variable FIRST_NAME: STRING (1 to 4);
variable MIDDLE_INITIAL: Character;
variable LAST_NAME: STRING (7 downto 1);
variable NAME: STRING(1 to 15);
FIRST_NAME:= "JOHN";
MIDDLE_INITIAL := 'Q';
LAST_NAME:= "CITIZEN";
NAME:= FIRST_NAME & ' ' & MIDDLE_INITIAL & ". " & LAST_NAME;
        -- The value of NAME is "JOHN Q. CITIZEN".
```

注意 当对象的类型是数组元素类型时，该操作数是合法的，见上例，如果第二操作数的类型是STRING，则合并操作字母'Q'是合法的。

4. 符号操作符

符号操作符是一元操作符+（保持符号不变）和-（改变符号）。它们只对数值操作数有效，因为优先级的原因，这些操作符不可以直接跟在乘或各种混合操作符之后，例如A/-B与A**-B是非法表达式，必须使用括号。表达式A/(-B)和A**(-B)是所要进行操作的合法形式。

```
type SMALL_Integer is Integer range -100 to 100;
variable A: SMALL_Integer := 10;
signal B, C: SMALL_Integer;
```

```
C <= -A;    -- Signal C has value -10.
A := -B;    -- Variable A has value +100.
```

5. 乘法操作符

表3-4说明了乘法操作符。

表3-4 乘法操作符

符 号	操 作
*	Integer类型和Real类型的算术乘法
/	Integer类型和Real类型的算术除法
Mod	Integer类型的取模运算
Rem	Integer类型的取余运算

整数的除法和余数用下述语句定义：

$$A=(A/B)*B+(A \text{ rem } B)$$

其中（A rem B）的符号与A的符号相同，绝对值小于B的绝对值。图3-23给出了典型数值的整数除法的结果。

```
 A     B   A/B   A rem B
 11    4    2       3
-11    4   -2      -3
 11   -4   -2       3
-11   -4    2      -3
```

图3-23 整数除法的结果

取模操作类似（A mod B），结果的符号与B的符号相同，其绝对值小于B的绝对值。而且，对于某些整数N，结果必须满足：

$$A=B*N+(A \bmod B)$$

```
 A     B    A mod B
 11    4       3
-11    4       1
 11   -4      -1
-11   -4      -3
```

图3-24 取模运算的结果

图3-24给出了典型数值整数取模运算的结果。其他例子包括：

```
variable A, B, C, D, E: Integer;
A := 5; B := 10; C := 15;  -- Initialize variables.
D := A * (-B) / 15;        -- Variable D has value -3.
E := (A *(-B)) rem 15;     -- Variable E has value -5.
E := (A * (-B)) mod 15;    -- Variable E has value 10.
```

任何物理数据类型的对象都可以被Integer类型或Real类型的实体乘或除。结果与第一个对象的类型相同。两个相同物理数据类型的对象也可以相除，结果为通用整数（*universal*

integer），即一种不受实现限制的整数。

```
variable RES1, RES2, RES3: RESISTANCE := 1 KOHM;
variable T1, T2: Time := 10 ns;
variable Z1, Z2: Integer := 10;
RES1 := Z1 * RES2;          -- RES1 has value 10 kohm.
T1 := 2.5 * T2;             -- T1 has value 25.0 ns.
Z1 := RES1/RES2;            -- Z1 has value 10.
T2 := T2/10;                -- T2 has value 1 ns.
Z1 := RES1/Z2;              -- Invalid statement. RES1 is type RESISTANCE,
                            -- Z1 is type Integer.
RES1 := RES1 * RES2;        -- Invalid statement.
                            -- Units do not match.
```

6. 杂类操作符

杂类操作符在所有操作符中优先级最高，见表3-5。

表3-5 杂类操作符

符 号	操 作
**	幂运算。左边操作数的类型必须为Integer或Real，右边操作数的类型必须为Integer。结果的类型与左边操作数的类型相同
abs	一元操作符，计算数值数据类型的绝对值
not	Bit或布尔类型的逻辑NOT操作符

如果左边操作数的类型是Real，指数为负的求幂运算是不允许的。这种操作的一些例子如下：

```
variable A, B, C: Integer := 5;
variable D: REAL := 2.5;
signal X1, X2, X3, X4: Bit := '1';
C := 2 ** A;                -- C is assigned the value 32.
D := 2.5 ** 2;              -- D is assigned the value 6.25
D := D ** (-1);             -- D is assigned the value 0.16
C := A ** (-2);             -- Invalid expression.
C := A ** (D);              -- Invalid expression.
C := abs (A - (3 * B));     -- C is assigned the value 10
X4 <= X1 and not X2;        -- X4 is assigned the value '0'.
```

3.6.3 顺序控制语句

本节介绍的控制语句用于在进程或子程序中说明算法，它们按照出现的顺序执行。

1. *wait*语句

*wait*语句的完整形式如下：

```
wait on X,Y until Z=0 for 100 ns;
```

该语句引起进程的执行被挂起，最多为100ns。如果100ns之前在X或Y上发生了事件，条件"Z=0"就被计算。如果当X或Y上的事件发生时"Z=0"为TRUE，那么进程将重新开始，否则，进程继续挂起。实际上，进程在100ns过后，或在Z=0条件下X或Y上发生事件时，无论哪个先来，都将重新开始。*wait*语句的其他有效形式为：

```
wait for 100 ns;            -- Always resume after 100 ns.
wait on A,B,C;              -- Resume when an event occurs on signal
                            -- A, B, or C.
```

```
wait until Z=0;              -- Resume when the expression "Z=0" becomes TRUE
                             -- due to an event on signal Z.
wait on A,B,C for 100 ns;    -- Wait for a maximum of 100 ns.
                             -- Resume earlier than 100 ns if there is an
                             -- event on A,B, or C.
wait on X,Y until Z=0;       -- Wait for an event on X or Y with Z=0.
wait until A=B for 100 ns;   -- Wait for the expression "A=B" to become TRUE
                             -- as the result of an event on A or B.
                             -- Maximum wait is 100 ns.
wait;                        -- wait forever, i.e., permanently suspend
                             -- the process.
```

2. *if*语句

*if*语句的格式如下：

```
if CONDITION1 then
  -- sequence_of_statements_1
elsif CONDITION2 then
  -- sequence_of_statements_2
     .
  -- any number of elsif clauses
     .
else
  -- last_sequence_of_statements
end if;
```

条件按照自上到下的顺序进行计算。第一个计算为TRUE的条件导致对应的{语句序列}被执行，如果所有条件的计算都为FALSE，则执行else对应的语句序列。注意，即使有多个条件为真，也至多只会有一个{语句序列}被执行。这是一个例子：

```
if A < 0 then
  LEVEL := 1;
elsif A > 1000 then
  LEVEL := 3;
else
  LEVEL :=2;
end if;
```

3. *case*语句

*case*语句的格式如下：

```
case EXPRESSION is
  when CHOICES1 => --sequence_of_statements_1
  when CHOICES2 => --sequence_of_statements_2
     .
     .
     .
  when others => --last_sequence_of_statements
end case;
```

要求表达式的值的所有可能选择只能在选择集中出现一次。选择的集合必须相斥，而且所有选择集的并集必须等于表达式的所有可能值的集合。表达式必须是离散的类型或一维的字符数组类型。每个选项必须和表达式是同样的类型，选项可以是一个列表，例：

```
when 0|1|2 => (AB <= "LOW");
```

关键字*others*用来包含表达式的所有没有包含在其他选项里的可能情况。这是一个方便的方法，保证表达式所有可能的值都被覆盖。

如果A和B是Integer类型，范围为（-10到+10），X是STRING类型，考虑这个例子：

```
case A+B is
  when 0          => X <= "ZERO";
  when (1 to 20)=> X <= "Positive";
  when others     => X <= "NEGATIVE";
end case;
```

4. *loop*语句

有3种类型的*loop*（循环）语句：

```
-- FOR loop
for NAME in RANGE loop
  -- sequence_of_statements
end loop;

-- WHILE loop
while CONDITION loop
  -- sequence_of_statements
end loop;

-- Simple loop
loop
  -- sequence_of_statements
end loop;
```

在*for*循环中，NAME必须和RANGE值有同样的基类型。循环被执行多次，每次NAME都从RANGE中取得一个新值。

在*while*循环中，在循环语句执行之前，先计算CONDITION。因而，如果CONDITION的初始值为FALSE，那么语句序列永远不会被执行。只要CONDITION的条件为TRUE，则语句序列将一直连续执行。如果CONDITION总为TRUE，则可能导致无限循环。

在*simple*循环中，语句序列被连续执行。为了防止无限循环，语句序列中必须包含一个*wait*语句或*exit*语句，见下面的这些例子：

```
while A<B loop
  -- sequence_of_statements
  A := A + 1;
end loop;
--
for I in 1 to 10 loop
  A(I) := A(I) + 1;
end loop;
--
loop
  compute (x);
  exit when x < 10;-- Exit statement is defined below.
end loop;
```

5. *next*语句

如果特定的条件为TRUE，*next*语句就用来中止当前的循环迭代。当迭代被中止后，语句序列以下一个迭代值再次开始。下面是格式：

```
next [loop_label] [when CONDITION];
```

如果loop_label存在，当CONDITION为TRUE时，带有给定标号的循环的当前迭代被中止；如果没有loop_label，则当CONDITION条件为TRUE时，最里面的循环被中止；如果CONDITION被忽略，则适当的循环迭代始终被终止。

6. *exit*语句

*exit*语句与*next*语句相似，不同之处在于*exit*使整个循环语句终止，*exit*语句的格式为：

```
exit [loop_label] [when CONDITION];
```

如果loop_label存在，当CONDITION为TRUE时，带有给定标号的循环的当前迭代被中止，即使迭代范围内的所有值没有都被处理；如果没有loop_label，则当CONDITION条件为TRUE时，最里面的循环被中止，即使迭代范围内的所有值没有都被处理；如果CONDITION被忽略，则适当的循环始终被终止。

7. *null*语句

*null*语句不做任何事。这在*case*语句中很有用，例如，如果对某个选项没有什么动作。因为必须覆盖所有的选项，因而在这种情况下，强制使用*null*语句。

3.6.4 结构体声明和并发语句

并发语句用于在结构体内描述信号的行为。并发语句的放置见如下说明：

```
architecture ARCHITECTURE_NAME of ENTITY_NAME is
   -- Architecture declaration section
   -- Signals are declared here.
   -- Variables may NOT be declared here.
begin
   --
   -- Concurrent statements that describe signal behavior
   --
end ARCHITECTURE_NAME;
```

本节讨论进程语句、断言语句和信号赋值语句。其他并发语句在需要时再进行介绍。

1. 进程语句

作为例子使用的进程（*process*）语句是并发语句。它们是结构体中使用的基本语句类型。实际上，其他并发语句都可以用等价的进程语句表示。讨论各种并发语句时，将利用这种等价性来帮助描述其他并发语句。进程语句有两种形式。一种形式中包括了敏感信号列表。以前的例子中大多采用这种形式。

```
LABEL:process (SENSITIVITY_SIGNAL_LIST)
        -- constant_declarations
        -- variable_declarations
        -- subprogram declarations
        -- signal declarations are NOT permitted here
        -- other, more specialized, items may also be declared here
      begin
        -- sequential_statements
      end process LABEL;
```

例如，图3-2的'1'计数器的结构体ALGORITHMIC包括一个并发进程语句。注意：标号是可选的。在本例中，进程语句没有标号。敏感信号列表包括一个信号A，该进程在仿真开始时执行一次，然后，只有在信号A上有事件发生时，才会被执行。通常，结构体的每个进程都在仿真开始时执行一次，随后，只有当敏感信号列表中的信号改变值（即在列表中一个或多个信号上有事件发生）时才被执行。注意，可以在进程内部声明常量和变量，但不可以声明信号。进程内部的语句必须是顺序语句。每次进程被激活时，执行顺序语句，这一点类似于C或C++那样的过程性语言。

进程内部声明的变量是静态（*static*）的，只在仿真开始的时候进行一次初始化，并且在

进程激活之间保持其不变。

进程语句的另一种形式中没有敏感信号的列表：

```
LABEL: process
         -- constant_declarations
         -- variable_declarations
         -- subprogram declarations
         -- signals may NOT be declared here.
         -- other, more specialized, items may also be declared here
       begin
         -- sequential_statements
       end process LABEL;
```

这种进程在仿真开始时执行一次，并且一直执行，直到遇到*wait*语句。当最后一个顺序语句执行之后，进程自动从第一个顺序语句处开始执行。因而，至少应该存在一个*wait*语句以防止无限循环的出现。带有敏感信号列表的进程与有*wait*语句作为进程中的最后一个语句的无敏感信号列表的进程等价。例如，进程：

```
S_LIST: process (S1, S2)
          -- constant_declarations
          -- variable_declarations
        begin
          -- sequential_statements
        end process S_LIST;
```

与如下进程等价：

```
NO_LIST: process
           -- constant_declarations   -- Same as in S_LIST
           -- variable_declarations   -- Same as in S_LIST
         begin
           -- sequential_statements   -- Same as in S_LIST
           wait on S1, S2
         end process NO_LIST;
```

假设除了最后一个*wait*语句，所有的声明和顺序语句都是相同的。在这两种情况下，进程在仿真开始时执行一次，随后，当信号S1和S2上发生事件时，进程将被执行。

很明显，没有敏感信号列表的进程必须有*wait*语句，而有敏感信号列表的进程不可以有*wait*语句。第一个要求防止了无限循环，第二个要求防止了相冲突的进程激活。

2. 并发断言语句

断言（*assert*）语句可以是顺序的或并发的。下面是并发断言语句的格式：

```
LABEL: assert Boolean_EXPRESSION
       report "Message_string"
       severity SEVERITY_LEVEL;
```

当Boolean_EXPRESSION中任何信号上有事件发生时，则计算Boolean_EXPRESSION。如果表达式计算结果为FALSE，Message_string写入标准输出设备。SEVERITY_LEVEL是标准VHDL语言的预定义枚举数据类型。缺省级别是WARNING。SEVERITY_LEVEL的解释与实现有关，在有些实现中，如果SEVERITY_LEVEL太高则会中止仿真。见图3-16中严重级别的定义，下面是例子：

```
LABEL: assert (A or B) = C
       report "WARNING: C is NOT equal to (A or B)"
       severity WARNING;
```

假设A，B，C是类型为Bit的信号。在正常的操作环境下，C应该等于（A or B）。我们检测违反此种情况的情况。断言语句在仿真开始时被激活。随后，任何时候（A，B，C）三个信号之中有事件发生时，就激活断言语句。一旦激活，第一行的条件被计算出，该条件是"(A or B)=C"，当条件为FALSE，即（A or B）与C不相等时，将在标准输出设备上输出警告信息。在大多数计算机系统中，输出可以重定向到文件里。

这个并发断言语句与下述进程语句等价。

```
LABEL:process (A, B, C)
      begin
        assert (A or B) = C
        report "WARNING: C is NOT equal to (A or B)"
        severity WARNING;
      end process LABEL;
```

3. 并发信号赋值语句

信号赋值语句可以是顺序的或并发的。如果是顺序的，则只有在算法到达该语句时才能执行。当它们是并发时，只要赋值语句右边的任何信号上有事件发生则被执行。例如，并发信号赋值语句：

```
LABEL:C <= A or B;
```

在仿真开始的时候被激活一次，然后，当信号A或B上发生事件时，该并发信号赋值语句等价于下列进程：

```
LABEL:process (A,B)
      begin
        C <= A or B;
      end process LABEL;
```

并发信号赋值语句也可以是条件的。条件并发信号赋值语句的格式如下：

```
LABEL:SIGNAL_NAME <= [transport]
      WAVEFORM1 when CONDITION1 else
      WAVEFORM2 when CONDITION2 else
                .
                .
                .
      WAVEFORMn when CONDITIONn else
      WAVEFORMq;
```

该并发语句在仿真开始时被激活，随后，只要WAVEFORM或CONDITION中任意信号上有事件发生，则被激活。如果CONDITION1为TRUE，WAVEFORM1就赋给SIGNAL_NAME，如果CONDITION1为FALSE而CONDITION2为TRUE，WAVEFORM2就赋给SIGNAL_NAME，只有当从CONDITION1到CONDITION n都是FALSE时，WAVEFORM q被赋给信号SIGNAL_NAME。只有WAVEFORM中的某一个被赋给信号，即使偶然有多个CONDITION是TRUE。第一个为TRUE 的CONDITION的那个WAVEFORM被赋给信号，一个例子如下：

```
LL1: S <= A or B when XX=1 else
         A and B when XX=2 else
         A xor B;
```

在这个例子中，假设A、B和S是类型为Bit的信号，XX是类型为Integer的信号。在仿真开始的时候语句被执行一次，随后在有事件发生于信号A、B和XX上时，语句被激活。该并发

信号赋值语句等价于下列进程：

```
LL1: process (A,B,XX) begin
       if XX=1 then S <= A or B;
       elsif XX=2 then S <= A and B;
       else S <= A xor B;
       end if;
     end process LL1;
```

选择并发信号赋值语句的格式是：

```
LABEL: with EXPRESSION select
       SIGNAL_NAME <= [transport]
       WAVEFORM1 when CHOICES1,
       WAVEFORM2 when CHOICES2,
                 .
                 .
                 .
       WAVEFORMn when CHOICES,
       WAVEFORMq when others;
```

EPRESSION的每一个可能值必须被包含于CHOICES之一。CHOICES的集合必须相斥，且其并集必须等于EPRESSION的所有值。注意：*others*是一个关键字，指包括EPRESSION的可能值却未包含在CHOICES之中的值。这是一种保证所有可能值都被包含的简便方法。在仿真开始时该并发语句被激活，随后在EXPRESSION或WAVEFORMS中任何信号上有事件发生时则被激活，例如：

```
LL2: with (S1 + S2) select
       C <= A after 5 ns when 0,
            B after 10 ns when 1 to Integer'high,
            D after 15 ns when others;
```

在t=0时被激活，而且只要信号S1，S2，A，B或D上有事件发生，则被激活。假设所有的信号类型都是Integer。这个选择并发信号赋值语句与下述进程语句等价：

```
LL2: process (S1, S2, A, B, D)
     begin
       case (S1 + S2) is
         when 0 => C <= A after 5 ns;
         when 1 to Integer'high => C <= B after 10 ns;
         when others => C <= D after 15 ns;
       end case;
     end process LL2;
```

3.6.5 子程序

在任何计算机语言中，通过编写函数和过程增强了内置于语言的基本操作。在VHDL中，函数和过程归在被称作子程序的基本编程结构之下。函数计算值并返回给调用表达式。函数不改变参数的值，只可在表达式内调用。过程是顺序或并发的语句。不给调用程序返回值，但可以修改它的参数。下面的结构体说明函数在何处被声明以及在何处被调用：

```
architecture WITH_SUBPROGRAMS of ENTITY_NAME is
  -- Subprograms may be declared here
begin
  -- Invoke subprograms here
end WITH_SUBPROGRAMS;
```

在结构体中的声明区域声明的子程序，只在该结构体内部是可见的。子程序也可在包集合中声明，这样通过use语句可以使子程序对于任何结构体都是可见的（参见本章中对包集合的讨论可知更多细节）。子程序也可以在进程、块或其他子程序的声明域进行声明。子程序只有在其声明的结构体内部才是直接可见的。（参见本章对可见性的讨论可知更多细节）。

后面将解释子程序是如何声明以及调用的。

1. 函数（function）

函数在VHDL中通过指定下列项进行声明：

1) 函数名。

2) 输入参数（如果有的话）。

3) 返回值和函数的类型。

4) 函数本身需要的任何声明。

5) 计算返回值的算法。

函数声明通用形式如下：

```
function FUNCTION_NAME (FORMAL_PARAMETER_DECLARATIONS)
  return RETURN_TYPE is
    -- constant and variable declarations
    -- no signal declarations are allowed here
begin
  -- sequential statements
  --
  return (RETURN_VALUE);
end FUNCTION_NAME;
```

函数的形式参数必须是*in*模式的。因而，不必指出函数形式参数的模式。仅允许对象类是常量和信号，缺省的对象类是常量。既然形式参数总是in模式，函数对调用环境就没有副作用。函数为调用表达式返回值，调用环境中的参数没有被修改。

变量和常量可以在函数的声明域进行声明。函数中的变量是动态的。每次函数调用变量都被初始化。在函数的调用之间，变量值并不被保存。因为函数说明算法，函数体中的所有语句必须是顺序语句。既然要立即给调用表达式返回值，函数体中就不可以有*wait*语句，该限制同样适用于任何被该函数调用的函数或过程。

在本章的开始处，对实体ONES_CNT给出的结构体MACRO使用了函数MAJ3(X)和OPAR3(X)，为了使用这些函数，它们必须如下声明：

```
function MAJ3(X: Bit_Vector(0 to 2)) return Bit is
  begin
    return (X(0) and X(1)) or (X(0) and X(2)) or (X(1) and X(2));
  end MAJ3;
```

参数X具有缺省的常量类。注意返回值的类型是Bit。返回值可以由一个表达式指定，正如上面的例子那样，或者通过执行一串语句指定。正如所说的，可以进行本地声明，但对这个简单的例子不需要如此。函数被用作表达式的一部分，因而在ONES_CNT的结构体中，MAJ3函数调用如下进行：

```
C(1) <= MAJ3(A);
```

图3-25给出了将Integer转化成长度为N的Bit_Vector函数的用法。它是在实体PULSE_GEN的结构体ALG中声明的。函数将整数INPUT和位置个数N作为参数传递。因为对于参数

INPUT和N没有指定类，故都被假定为常量。假设N足够大，产生的位向量可以保存转化后的值。函数实现了使用除以2的不断减少幂的连续整数除法的经典转换算法。在每一步，结果的位由整数除法的商来确定。整数除法的余数做为下一轮除法的被除数。

```
entity PULSE_GEN is
  generic(N: INTEGER; PER: TIME);
  port(START:in BIT; PGOUT:out BIT_VECTOR(N-1 downto 0);
       SYNC:inout BIT);
end PULSE_GEN;
architecture ALG of PULSE_GEN is
  function INT_TO_BIN (INPUT : INTEGER;N : POSITIVE)
    return BIT_VECTOR is
    variable FOUT: BIT_VECTOR(0 to N-1);
    variable TEMP_A: INTEGER:= 0;
    variable TEMP_B: INTEGER:= 0;
  begin                -- Begin function code.
    TEMP_A := INPUT;
    for I in N-1 downto 0 loop
      TEMP_B := TEMP_A/(2**I);
      TEMP_A := TEMP_A rem (2**I);
      if (TEMP_B = 1) then
        FOUT(N-1-I) := '1';
      else
        FOUT(N-1-I) := '0';
      end if;
    end loop;
    return FOUT;
  end INT_TO_BIN;
begin                  -- Begin architecture body.
  process(START,SYNC)
    variable CNT: INTEGER:= 0;
  begin                -- Begin process
    if START'EVENT and START='1' then
      CNT := 2**N-1;
    end if;
    PGOUT <=  INT_TO_BIN(CNT,N) after PER;
    if CNT /= -1 and START ='1' then
      SYNC <= not SYNC after PER;
      CNT := CNT -1;
    end if;
  end process;
end ALG;
```

图3-25 将类型Integer转化为类型Bit_Vector的函数

实体PULSE_GEN的目的是产生所有长度为N的二进制模式。在PULSE_GEN的结构体ALG中，函数INT_BIN用来将CNT的值转化为长度为N的位向量。CNT循环从0到2**N-1，因而，产生了所有的位向量，每个位向量保持一个周期PER。每次位向量改变时，一个同步信号就发生翻转。

函数中声明的参数叫做形参（*formal*），在本例中，函数INT_TO_BIN的形参为INPUT和N，形式参数只在函数声明内部可见。它们被用来说明函数的算法。当函数被调用时，调用语句中的参数叫做实参（*actual*）。在本例中，调用语句中的实参是CNT和N。该例使用位置相关列表将实参与形参相关联，即CNT与INPUT关联，N与N关联。因而在结构体中，变量CNT从Integer转化为Bit_Vector的形式。

2. 子过程（procedure）

子过程是可以修改一个或多个输入参数的子程序。子过程通过说明以下几点而被声明：

1) 子过程名。

2) 输入与输出参数（如果有的话）。

3) 过程本身需要的任何声明。

4) 算法。

下面是子过程声明的框架：

```
procedure PROCEDURE_NAME (FORMAL_PARAMETER_DECLARATIONS)
   -- Procedure declaration part
   -- Variables and constants may be declared
   -- Signals may NOT be declared here
begin
   -- Sequential statements
end PROCEDURE_NAME;
```

形式参数的模式可以为*in*、*out*和*inout*，如果未指明模式，则认为都是模式*in*，允许的对象类是常量、变量和信号。如果模式为*in*且没有指定对象类，则认为是常量，如果模式为*out*或*inout*，则缺省的对象类为变量。

在声明域中可以声明变量和常量，但不可以声明信号。子过程中的变量是动态的，每次调用时都初始化，而且在调用之间并不保持值。子过程体中*begin*关键字之后使用顺序语句来说明子过程执行的算法。与函数不同，子过程中可以包含*wait*语句。然而，如果子过程被函数调用，则其中不能包括*wait*语句。子过程并不对调用表达式返回值，而是修改其一个或多个形式参数。这引起与形式参数对应的调用环境中的实际参数被修改。通常，如果形式参数为信号类，则在当前的仿真周期，信号并不更新，而是被安排在将来的某一个仿真周期进行更新。

图3-26给出了执行位向量加法的子过程ADD的声明。形参A、B和CIN是*in*模式，因而它们只可被子过程读。子过程不能对*in*模式的形式参数赋予新值。因为没有给A、B和CIN指明其对象类，它们被认为是常量。形参SUM和COUT是*out*模式，只可被子过程赋值。子过程不可读出*out*模式形参的当前值。因为没有指定对象类，所以SUM和COUT有缺省类变量。注意：给SUM和COUT赋予新值的语句使用“:=”操作符，给SUM和COUT赋予的新值在当前仿真周期立即生效。声明了几个内部子过程变量，它们只有在子过程内部才可见。这种变量是动态的，每次子过程调用都重新初始化。因为没有指定初始值，类型为Bit的初始值是'0'，Bit_Vector的缺省值为全零向量。加运算的实现对于任何长度，无论范围是降序还是升序都可用。这是通过使用类型为Bit_Vector(A'LENGTH-1 downto 0) 的内部变量并在循环范围说明中使用SUMV'high来实现的。假设最低位在右边，通过循环来实现全加等式。重复应用循环可以实现多位的加法。

通常，子过程可以在结构体内作为并发语句调用或在进程、函数或其他子过程内作为顺序语句调用。然而，子过程ADD只可用在进程、函数或其他子过程内作为顺序语句调用，这是因为输出SUM和COUT是变量类型，因而，它们必须与变量类的实参相对应，因为结构体声明域中不可定义变量，故子过程不可在结构体内作为并发语句被调用。

与函数类似，当一个子过程被调用时，调用环境之中的实参和形参使用位置相关列表或命名相关列表相关联。形参及其相应之实参必须有相同的基类型。下一节讨论对象类的匹配

问题。

```
procedure ADD(A,B: in BIT_VECTOR; CIN: in BIT;
              SUM: out BIT_VECTOR; COUT: out BIT) is
  variable SUMV,AV,BV: BIT_VECTOR(A'LENGTH-1 downto 0);
  variable CARRY: BIT;
begin
  AV := A;
  BV := B;
  CARRY := CIN;
  for I in 0 to SUMV'HIGH loop
    SUMV(I) := AV(I) xor BV(I) xor CARRY;
    CARRY := (AV(I) and BV(I)) or (AV(I) and CARRY)
             or (BV(i) and CARRY);
  end loop;
  COUT := CARRY;
  SUM := SUMV;
end ADD;
```

图3-26 实现位向量相加的子过程

3. 子程序中形参与实参的对象类匹配

表3-6给出了在子程序调用时要求的对象类之间的匹配关系，一个常量形参可以与实参表达式相匹配是一种有趣的情况。当形参类为常量时，预示子程序不可对其进行修改。当子程序被调用时，参数值取自表达式的值。在其基本形式中，表达式可以为信号或变量，因而当形参类为常量时，这些对象亦可与形参匹配。见MAJ3函数的例子，对于参数X未指定对象类，因而，缺省情况下，对象类为常量，在ONES_CNT的构架MACRO中，使用的实参为信号A，是一个合法表达式，因而与常量形参X相匹配。

表3-6 形参和实参的对象类匹配

形 参	实 参
信号	信号
变量	变量
常量	表达式

对于图3-26中的子过程ADD，形参A、B与CIN的缺省类为常量。这样，两个形参都可与调用语句中的实参表达式相匹配。在下面的调用中，进程P1用子过程ADD从两个可能的和之中选择一个赋给信号R5。注意两个子过程调用都将形参A、B和CIN（*in*模式，对象类为常量）与信号匹配。因为信号名是合法表达式，因而这也是合法的。每个子过程调用将形参SUM和COUT（*out*模式，对象类为变量）与实参变量R6和COUT_TEMP匹配，这是必要的，因为变量类的形参必须与变量类的实参相匹配。进程结尾处的信号赋值语句将要求的值赋与R5和COUT。

```
architecture PROCEDURE_INVOCATION of PROCEDURE_TEST is
  -- Procedure declaration goes here.
  signal R1,R2,R3,R4,R5: Bit_Vector (7 downto 0);
  signal CIN: Bit := '0';
  signal COUT: Bit;
  signal S1: Boolean;
begin
```

```
  -- Other concurrent statements.
P1:  process (S1)
     variable R6: Bit_Vector (7 downto 0);
     variable COUT_TEMP: Bit;
   begin
     if S1 then
       ADD(R1, R2, CIN, R6, COUT_TEMP);
     else
       ADD(R3, R4, CIN, R6, COUT_TEMP);
     end if;
     R5 <= R6;
     COUT <= COUT_TEMP;
   end process P1;
   -- Other concurrent statements.
end PROCEDURE_INVOCATION;
```

3.7 VHDL的高级特征

本节讨论VHDL语言的高级特征。

3.7.1 重载

在VHDL中，可以通过重新声明改变文字、操作符名、函数和过程的含义。这种处理过程叫做重载。见下面的信号赋值语句：

```
F <= (A and B) or (C and D);
```

假设A，B，C，D和F原来声明为类型Bit，也就是说，使用二值逻辑。后来发现使用MVL4四值逻辑对于同一逻辑方程似乎更加合理，包含值'X'，'0'，'1'和'Z'，于是可以重新声明A，B，C，D和F为类型MVL4。注意，这样做就重载了文字值'0'和'1'，因为这两个文字是类型Bit的成员。然而如果声明了F，A，B，C，D为类型MVL4，内置的AND和OR操作符就不可再使用，因为它们是为类型Bit和Boolean定义的。可以定义两个函数：MVL4_AND（X；Y）和MVL4_OR（X；Y），以进行指定的操作，这要求用下述函数调用符号来重写等式：

```
F <= MVL4_OR(MVL4_AND(A,B), MVL4_AND(C,D));
```

用这种方法存在一些问题。对于复杂的布尔表达式，这种形式难以阅读，以这种方式写布尔等式容易引起错误。更好的方法是重载操作符*and*和*or*，可以如图3-27中所示，重新定义一个新的*and*操作符。

```
function "and" (L, R: MVL4) return MVL4 is
  -- Declare a two-dimensional table type.
  type MVL4_TABLE is array (MVL4, MVL4) of MVL4;
  -- truth table for "and" function
  constant table_AND: MVL4_TABLE :=
--  ------------------------------
--  |  X    0    1    Z  |
--  ------------------------------
    (('X', '0', 'X', 'X'),  --  | X |
     ('0', '0', '0', '0'),  --  | 0 |
     ('X', '0', '1', 'X'),  --  | 1 |
     ('X', '0', 'X', 'X')); --  | Z |
begin
  return table_AND(L, R);
end "and";
```

图3-27 重载*and*操作符

在函数的声明区域，声明了一个两维数组类型。该类型的数组元素的类型是MVL4，两列数据下标的类型也是MVL4。*and*操作符的操作表以常量形式进行声明。函数仅返回从表中得到的值。根据该函数的声明，可以写出如下表达式：

```
F <= A and B;
```

假设F、A和B是类型MVL4，如果出于某种原因想使用函数调用符号，同一表达式可以写为：

```
F <= "and"(A,B);
```

立刻会产生这样的问题：内置的*and*操作符对于类型为Bit和Boolean的对象是否仍可用？答案是肯定的。对于同名的运算，VHDL分析程序通过检查参数与结果的外形（profile）来判断究竟是使用哪个操作符。结果的外形指明返回结果的类型。使用这些外形，分析程序可以分辨不同的操作符。例如，内置*and*运算的两个参数类型是类型BIT或Boolean，对于MVL4的*and*操作符，两个操作数的类型是MVL4。

子程序名也可以重载。例如，考虑图3-28中所示函数，将MVL4_VECTOR转换成整数。函数先检查位的位置不等于'X'或'Z'，假设该向量由'0'和'1'组成，对于每个发现的'1'，函数加以适当的权。注意：该函数假设该字的最小位转换之后的下标最小。

```
function INTVAL (VAL: MVL4_VECTOR)
    return INTEGER is
  variable SUM: INTEGER:= 0;
begin
  for  N in VAL'LOW to VAL'HIGH loop
    assert not(VAL(N) = 'X' or VAL(N) = 'Z')
    report "INTVAL inputs not 0 or 1"
    severity WARNING;
    if  VAL(N) = '1' then
      SUM := SUM + (2**N);
    end if;
  end loop;
  return SUM;
end INTVAL;
```

图3-28 将类型MVL4_VECTOR转化为类型Integer的函数声明

假设声明了图3-29所示的另一个整数转换函数，用来将类型为Bit_Vector的参数转化为整数，因为两个函数名相同，由调用参数的类型来决定调用的是哪一个函数。

```
function INTVAL(VAL:BIT_VECTOR) return INTEGER is
  variable SUM: INTEGER:=0;
begin
  for N in VAL'LOW to VAL'HIGH loop
    if VAL(N)='1' then
      SUM := SUM + (2 ** N);
    end if;
  end loop;
  return SUM;
end INTVAL;
```

图3-29 将类型BIT_VECTOR转换成类型Integer的函数的声明

这里又产生了关于重载的另一个问题，如果两个子程序声明了同样的名字、同样的参数和结果外形，则哪个是可见的？结果是最近声明的那个隐藏了前面的一个。

重载的能力可以通过用IEEE包集合（下一节进行详细讨论）来重载操作符进行说明。首先，类型STD_LOGIC和STD_LOGIC_VECTOR重载了所有的布尔操作符。其次，算术操作符也可以被重载。在图3-30中，实体的三个输入是STD_LOGIC_VECTOR类型。然而，因为使用了包集合STD_LOGIC_SIGND，该包集合中重载的“+”操作符将输入解释为有符号数，进行补码算术运算，且返回类型为STD_LOGIC的值。如果使用的是包集合STD_LOGIC_UNSIGNED，则该包中重载的“+”操作符进行的是无符号数加法。

```
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_SIGNED.all;.
entity ADD_OVERLOAD is
  port ( A, B, C: in  STD_LOGIC_VECTOR (7 downto 0);
         SUM: out STD_LOGIC_VECTOR (7 downto 0));
end ADD_OVERLOAD;
architecture PACK_SIGNED of ADD_OVERLOAD is
begin
  SUM <=  A + B + C; --This is now 2's-
                     --complement addition
end PACK_SIGNED;
```

图3-30 用有符号的包集合重载ADD

总而言之，重载提供了改变一个名字的含义而无需修改调用它的VHDL代码的方式。

重载子程序名必须或者有不同数目的变量，或者其形参之中至少有一个有不同的数据类型。例如，对于一个D触发器，可以有三个重载版本。如下所示的三个子过程调用，假设形参的类型为Bit。

```
DFF (CLK, D, Q);
DFF (CLK, D, Q, QBAR);
DFF (CLK, D, CLEAR, PRESET, Q);
```

然而，不可以有如下所示的两个对D触发器的重载版本，见下面的两个过程调用，所有形参的类型都是Bit类型。

```
DFF (CLK, D, CLEAR, Q, QBAR);
DFF (CLK, D, PRESET, Q, QBAR);
```

前面的重载版本DFF过程为无效，因为两个形参的个数相同，类型也相同。

3.7.2 包

如果每次使用时都必须声明，则十分麻烦。VHDL使用包集合来保存常用的声明，而且每个包集合都有包集合名。可以通过访问包集合名使得包集合中的声明可见。首先，必须声明包，图3-31给出了一个这种声明的例子。注意：在HANDY包集合的声明中，声明了子类型和一个函数的接口。函数的代码在包体中给出。如果包集合不包含子程序，则不需要包体。

根据包集合HANDY的定义，假设一个实体LOGSYS想访问该包，在实体LOGSYS的接口描述之前使用*use*子句来实现：

```
use work.HANDY.all;
entity LOGSYS is
```

```
  port(X: in BITVECT3;Y: out BITVECT2);
end LOGSYS;
```

```
package HANDY is
  subtype BITVECT3 is BIT_VECTOR(0 to 2);
  subtype BITVECT2 is BIT_VECTOR(0 to 1);
  function MAJ3(X: BIT_VECTOR(0 to 2)) return BIT;

------ Other Declarations----------------------------

end HANDY;
package body HANDY is
  function MAJ3(X: BIT_VECTOR(0 to 2))
    return BIT is
  begin
    return (X(0) and X(1)) or (X(0) and X(2))
           or (X(1) and X(2));
  end MAJ3;

------ Other Subprogram Declarations -----------------

end HANDY;
```

图3-31 包集合的定义

于是，HANDY中包含的所有声明在实体LOGSYS中是可见的，包括其任何结构体。下一节将详细讨论可见性。

*use*语句的通用形式为：

```
use LIBRARY_NAME.PACKAGE_NAME.ELEMENT_NAME;
```

其中LIBRARY_NAME是包含该包集合的库的名字，PACKAGE_NAME是包集合名，ELEMENT_NAME是包集合中特定一项的名字，例如：

```
use MY_LIBRARY.MY_PACKAGE.DFF; -- Get one component.
use MY_LIBRARY.MY_PACKAGE.all; -- Get everything in package.
```

在HANDY例子中，关键字*work*指与当前工程相连的缺省库，库中包括所有的包集合、过程和组件，它们作为当前工程的一部分已经分析过了。

包集合是很有用的语言特征。设计组可以使用包类型声明和与系统相关的子程序的标准包。见第5章中的说明，我们讨论系统建模时，这些声明和子程序中的代码数量很重要。包机制使得建模者免于重复输入这些代码。如果在一个设计组中不同的设计者能共享，也可保证设计的一致性。

VHDL语言定义了包集合STANDARD，可以被所有的实体使用。和其他东西一起，该包集合包含了类型Bit、Bit_Vector、Boolean、Integer、Real、Character、String和Time类型，还有子类型Positive和Natural的定义。

IEEE已经开发了标准1164作为标准的9值逻辑系统，包括针对逻辑综合的特殊应用。IEEE委员会开发了两个包集合 1) STD_LOGIC_1164，定义了基本的数值系统和相关函数（由供应商使用）。2) NUMERIC_STD，提供重载算术操作符和其他用于综合的操作符。供应商也开发了他们自己版本的这个包。特别是Synopsys开发了：

1) STD_LOGIC_ARITH，定义了两种新的标准逻辑向量类型：SIGNED和UNSIGNED，

以及相关重载的算术和关系操作符。

2) STD_LOGIC_UNSIGNED，类型为STD_LOGIC_VECTOR，为无符号数的对象提供了重载的算术和关系操作符。

3) STD_LOGIC_SIGNED，类型为STD_LOGIC_VECTOR，为符号数的对象提供了重载的算术和关系操作符。

上面所有的三个包都允许操作数类型为Integer。Synopsys的包集合STD_LOGIC_MISC包括布尔内积操作符。例如，and内积操作符计算标准逻辑向量所有位的与而得到一个标量结果。Synopsys包集合STD_LOGIC_TEXTIO对类型为STD_LOGIC和STD_LOGIC_VECTOR的对象提供文本I/O。

IEEE也开发了包MATH_REAL和MATH_COMPLEX，包括操作符和支持计算算术表达式的函数，例如MATH_REAL就包括计算超越函数的代码。

3.7.3 可见性

在介绍VHDL语言时，我们给出了大量关于VHDL的类型、对象和子程序声明的例子，但并未对其在哪里声明给出明确的定义，下面就此进行讨论。

域：一个逻辑上连续的，有界的文本部分。

声明域：一个域，在其中可用一个名字无二义地对应于所声明的项。

当一个项在声明域中的某一点被声明之后，它的名字直到声明域的末尾都是可见的。它的实际效果是在使用一个项之前必须先定义。通常名字只有在其定义的域之内是可见的。即，它们是直接可见的（*directly visible*）。然而，正如我们所见，通过库和*use*子句，也可以使名字在其声明域之外可见。第二种类型的可见性叫做选择可见性（*visiblity by selection*）。

现在列出用来建立声明域的下列基本结构。这样可将它们分作两类：通用声明域（允许很大范围的声明）和专门声明域（只允许声明类型的一个限制的集合）。

创建一个通用声明域的结构如下：

1) 一个实体声明和一个相连的结构体。

2) 一个进程。

3) 一个块。

4) 一个子程序。

5) 一个包集合。

图3-32给出一个例子，在五个不同的声明域中用名字X声明信号和变量，这五个声明域为：包集合SIG、实体Y和它的结构体Z、函数R、嵌套块B，以及构架Z中的一个进程。因为每个声明对于给定的声明域都是局部的，因而这些声明可以毫无冲突地进行。在每种情况中，名字X直到在声明X的声明域的末尾都是直接可见的。

在包集合SIG中声明的信号X(X=1)在整个包集合SIG中是直接可见的。在实体Y中声明的信号X(X=2)在实体Y中和其所有结构体之中是直接可见的。因而，Z2被赋值2。函数R中的Return X语句返回X=3的值，该值对于函数R是直接可见的，因而Z3被赋值3。Z5的值是5，这是因为它被赋值X(X=5)，该值对于进程P1是直接可见的。

在这三种情况之中，名字X通过在其声明域外部的选择而成为可见的：

1) 包集合名通过语句“use work.SIG”变为可见（下一节将有介绍）。然后，选择名字

SIG.X来指示包SIG中X的值，该信号（X=1）的值被赋给信号Z1。

```
package SIG is
  signal X: INTEGER:= 1;
end SIG;

use work.SIG.all;
entity Y is
  signal X: INTEGER:= 2;
end Y;

architecture Z of Y is
  signal Z1,Z2,Z3,Z4,Z5: INTEGER:= 0;
  function R return INTEGER is
    variable X: INTEGER := 3;
  begin
    return X; -- Returns value of 3.
  end R;
begin
  B: block
    signal X: INTEGER := 4;
    signal Z6: INTEGER := 0;
  begin
    Z6 <= X + Y.X;  -- Z6 = 6
  end block B;

P1: process
    variable X: INTEGER :=5;
  begin
    Z5 <= X;          -- Z5=5
    wait;
  end process;

  Z1 <= work.SIG.X; -- Z1=1
  Z2 <= X;          -- Z2=2
  Z3 <= R;          -- Z3=3
  Z4 <= B.X;        -- Z4=4
end Z;
```

图3-32 声明域

2) 选择名字B.X，用来访问块B之中X的值，该信号（X=4）的值被赋予Z4。

3) 在块B内部，选择名字Y.X，用来访问实体Y中定义的X的值。该信号的值（X=2）与X在块B之中直接可见的局部值（X=4）相加，结果6被赋给Z6。

注意 在每种情况下，所选择的信号名采用形式P.S，其中P是代表名字声明出现结构的前缀，S是后缀，是声明的名字。名字选择可以嵌套无限层。例如，形如P1.P2.P3.S的选择名代表一个在P3中声明的信号S，而P3嵌套于P2，P2又嵌套于P1之中。

如果对实体Y的结构体Z进行仿真，可以看到信号Z_i被赋予值i。

下面的基本结构创建专门的声明域：

1) 记录类型：记录域的名字是隐含声明的。

2) 循环：循环指针控制变量是隐含声明的。

3) 元件声明：元件名、端口及类属名是隐含声明的。

可以通过在用户选项文件中加入这样的语句，从而将分析过的文件加入LIB2：

```
WORK  > LIB2
LIB2: /VTVHDL/USERS/JRA/LIB2
```

于是主机库LIB2将接收分析的结果。如果后来又想使LIB1变为work库，可以修改用户选项文件，或者可以通过暂时调用分析使LIB1成为WORK库。

```
VHDLAN -W LIB1  MODEL.VHD
```

只要该分析一结束，LIB2又会成为WORK库。

3.7.5 配置

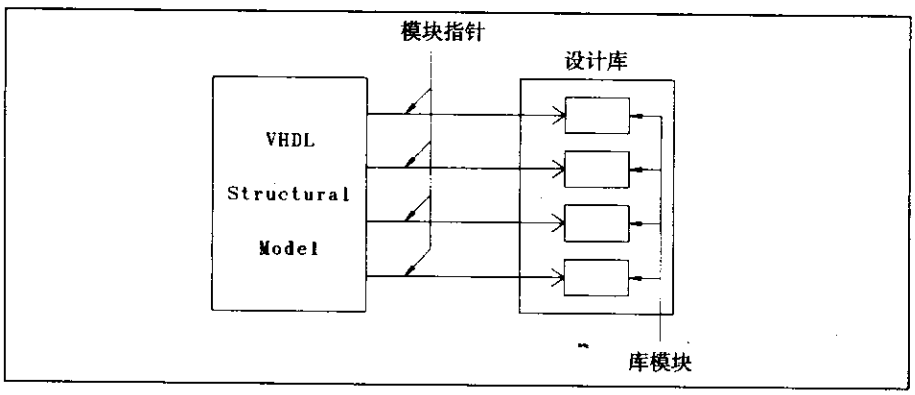


图3-33　指向库模块的指针

VHDL的结构化结构体通过构架声明域中声明组件的实例化而构成。在一个结构模型被仿真之前，每个实例化的元件与一个库模块绑定在一起，图3-33给出了实现过程。一个VHDL结构模块包括一组指向到前一节所讨论的设计库中模块的指针。一种确定指针的方法是将配置说明语句直接置于结构模块之中。图3-34给出了第1章中介绍的TWO_CONSECUTIVE实体的结构化模块的例子。注意注释为“pointer”的每个配置说明语句具有下面这种形式：

```
for INSTANTIATED_COMPONENT use LIBRARY_COMPONENT
```

对于D触发器的情况，INSTANTIATED_COMPONENTS是“all：EDGE_TRIGGERED_D”。项all指明所有实例化的EDGE_TRIGGERED_D组件映射入同一模型，当然也可以分别映射D触发器，如：

```
for C1: EDGE_TRIGGERED_D
  use entity EDGE_TRIG_D_A(BEHAVIOR);
for C2: EDGE_TRIGGERED_D
  use entity EDGE_TRIG_D_B(BEHAVIOR);
```

可以将每个D触发器映射入不同的库模型。

在这个例子中，库WORK包含该模型，正如所讨论的，库名WORK总是可见的。语句“use work.all”使得该模型的实体名为可见的。还有一种方法，完全不用“use work.all”这样的语句，而使用元件模型的选择名，例如。D触发器的组件说明如下：

```
for all: EDGE_TRIGGERED_D
  use entity work.EDGE_TRIG_D(BEHAVIOR); --model pointer
```

```
entity TWO_CONSECUTIVE is
  port(CLK,R,X: in BIT; Z: out BIT);
end TWO_CONSECUTIVE;
use work.all;
architecture STRUCTURAL of TWO_CONSECUTIVE is
  signal Y0,Y1,A0,A1: BIT := '0';
  signal NY0,NX: BIT :='1';
  signal ONE: BIT :='1';
  component EDGE_TRIGGERED_D
    port(CLK,D,NCLR: in BIT;
         Q,QN: out BIT);
  end component;
  for all: EDGE_TRIGGERED_D
    use entity EDGE_TRIG_D(BEHAVIOR); --model pointer
  component INVG
    port(I: in BIT;O: out BIT);
  end component;
  for all: INVG
    use entity INV(BEHAVIOR); --model pointer
  component AND3G
    port(I1,I2,I3: in BIT;O: out BIT);
  end component;
  for all: AND3G
    use entity AND3(BEHAVIOR); --model pointer
  component OR2G
    port(I1,I2: in BIT;O: out BIT);
  end component;
  for all: OR2G
    use entity OR2(BEHAVIOR); --model pointer
begin
  C1: EDGE_TRIGGERED_D
    port map(CLK,X,R,Y0,NY0);
  C2: EDGE_TRIGGERED_D
    port map(CLK,ONE,R,Y1,open);
  C3: INVG
    port map(X,NX);
  C4: AND3G
    port map(X,Y0,Y1,A0);
  C5: AND3G
    port map(NY0,Y1,NX,A1);
  C6: OR2G
    port map(A0,A1,Z);
end STRUCTURAL;
```

图3-34 实体TWO_CONSECUTIVE的配置说明

因为库名WORK总是可见的，这样也奏效。

绑定元件的另一种可能的方法是使用配置声明。在这种方法中，结构化结构体中的元件实例是无约束的，元件说明语句被集中置于一个独立的可分析单元之中，称作配置声明。例如，假设从构架实体TWO_CONSECOTIUE的STRUCTURAL中移出所有的配置说明语句（注释为“pointer”的），见图3-34。元件说明可集中置于配置声明之中，见图3-35。注意配置声明的名字是PARTS，是对实体TWO_CONSECUTIUE的配置声明。在配置声明之中，外层的*for*语句给出了配置的结构体名，内层的*for*语句是实例化元件的配置说明。

配置说明是一级库单元。然而，必须在对它们进行配置的结构体之后进行分析。而且，在模型被仿真的时候，配置声明与测试模块的测试程序包中被测试的元件进行绑定。见图3-36。

当想对同一结构化结构体通过将元件绑定到不同的库元件上进行多次不同的仿真时，配

置说明很有用。在这种情况下，可以在分析结构化结构体一次之后为它的结构体创建任何数量的配置，而不需要再次分析结构模型本身。这样可以减少错误和分析时间，提高了模块的重用性。在这种情况下，未绑定的结构体对应于空着芯片插座的印制电路板。在第7章和第11章，将讨论配置声明的其他有用特征。

```
configuration PARTS of TWO_CONSECUTIVE is
  for STRUCTURAL
    for all: EDGE_TRIGGERED_D
        use entity work.EDGE_TRIG_D(BEHAVIOR);
    end for;
    for all: INVG
        use entity work.INV(BEHAVIOR);
    end for;
    for all: AND3G
        use entity work.AND3(BEHAVIOR);
    end for;
    for all: OR2G
        use entity work.OR2(BEHAVIOR);
    end for;
  end for;
end PARTS;
```

图3-35 实体TWO_CONSECUTIVE的配置声明

```
entity TB is
  end TB;
use WORK.all;
architecture TCTEMPT of TB is
  signal X,R,CLK,Z: BIT;
  component TWIR
    port (CLK,R,X: in BIT;
          Z: out BIT);
    end component;
    for C1: TWIR use configuration work.PARTS;
begin
  C1: TWIR
    port map(CLK,R,X,Z);
  CLK <='0',            '1'after  10 ns,'0'after  20 ns,
       '1'after  30 ns,'0'after  40 ns,'1'after  50 ns,
       '0'after  60 ns,'1'after  70 ns,'0'after  80 ns,
       '1'after  90 ns,'0'after 100 ns,'1'after 110 ns,
       '0'after 120 ns,'1'after 130 ns,'0'after 140 ns;
  X <= '0','1' after  15 ns,'0' after 55 ns;
  R <= '1','0' after 125 ns,'1' after 127 ns;
end TCTEMPT;
```

图3-36 对配置进行测试

3.7.6 文件I/O

在VHDL模型开发中，有一个重要要求是能够使用测试向量来驱动一个模型。在对测试程序包的讨论中，我们展示了如何通过使用波形元素和信号赋值语句来进行程度有限的测试。这种方法对于大的测试向量集而言并不实际。还有一种相关问题是想要在仿真开始时从外部文件初始化存储器，最后可以将格式化的、可读的仿真结果写入外部文件。为了解决这些问

题，VHDL提供了文件I/O的能力，即在仿真期间读写外部文件的能力。所有的文件都是顺序的，可以看作数据流。有两种文件可以进行读写：格式化文件的和文本文件。格式化的文件必须由VHDL仿真程序来写，文件格式与主机有关，无法在主机环境中以人可理解的方式读出。文本文件是可读的，可使用文本编辑器在主机环境中创建。文件模式可为in或out，但不可为inout，因而，在某次仿真中写入的文件，不可以在同一仿真中读出。

1. 格式化I/O

格式化I/O需要两个声明：文件类型声明和使用先前声明的文件类型的文件声明。例如，表示用于测试模型的位向量流的文件类型可以声明为：

```
type INP_COMB is file of Bit_Vector;
```

该文件类型的基类是Bit_Vector。一般地，除了存取类型和其他文件类型，文件类型的基类可以为任何类型。根据这个声明，可以这样声明输出文件：

```
file OUTVECT: INP_COMB is out "TEST.VECT"
```

在这个声明之中，OUTVECT是文件逻辑名，而“TEST.VECT”是主机文件名，主机文件名是一个字符串，而且，必须用双引号引起来。只要文件声明为out模式，则隐含定义了一个WRITE过程，例如，对于上面的文件，相应的WRITE过程是：

```
WRITE(OUTVECT: out INP_COMB; V: in Bit_Vector);
```

其中V是在WRITE过程被调用的声明域中声明的一个变量。无论何时，只要执行WRITE过程，V的值就被添加到文件OUTVECT。

同样，可以这样定义输入文件：

```
file INVECT: INP_COMB is in "TEST.VECT"
```

对于所有的输出文件，隐含地定义了READ过程和ENDFILE函数。对于上面声明的文件，声明是：

```
READ(INVECT,V,LENGTH)
ENDFILE(INVECT)
```

每次对READ过程的调用，都从文件INVECT的头部中读出一个位向量并赋值给变量V，顺序文件中的下一个位向量则移至文件头部。对于每次读，自然数LENGTH返回位向量的长度。只有在文件类型的基类为无限制数组时，该过程参数才存在。ENDFILE函数返回类型为Boolean，如果读到文件末尾，则返回值为TRUE，否则返回FALSE。

为了说明格式化文件I/O的使用，参见图3-37。VHDL代码包括两个实体OBVS和IBVS，实体OBVS将一串向量写入叫做TEST.VECT的文件之中。测试向量由实体PULSE_GEN产生，我们在函数一节曾讨论过。回想对于给定的值N，它产生的序列包括所有2*N种可能长度为N的位向量。每输出一个新的位向量，输出SYNC就发生翻转，每次SYNC改变，过程WRITE_VECT就执行将位向量写入文件的操作。

实体IVBS将测试向量读回。当输入PLAY到达‘1’，过程READ_VECT的第一个wait语句完成时，就进入循环，每循环一次读出一个向量并将其值赋予变量V，V赋值给BVOUT。接下来等待长度为PER的周期。这样，BVOUT在每个PER时间单位改变一次，循环一直进行，直到到达文件尾。整个测试向量因此被重放。

实体OBVS向文件TEST_VECT中写入，实体IBVS从文件中读出。要使它正常工作，必须

满足两个条件：

```
entity OBVS is
  generic(N:INTEGER;PER: TIME);
    port(GEN: in BIT);
end OBVS;
use work.all;
architecture FIO of OBVS is
  type INP_COMB is file of BIT_VECTOR;
  file OUTVECT: INP_COMB is out "TEST.VECT";
  signal VECTORS: BIT_VECTOR(N-1 downto 0);
  signal SYNC: BIT;
  component PG
    generic(N: INTEGER; PER: TIME);
    port(START: in BIT;
         PGOUT: out BIT_VECTOR(N-1 downto 0);
         SYNC: inout BIT);
  end component;
  for C1: PG  use entity work.PULSE_GEN(ALG);
begin
  C1: PG
    generic map(N => N, PER => PER)
    port map(START => GEN, PGOUT => VECTORS, SYNC => SYNC);
  WRITE_VECT: process(SYNC)
    variable V: BIT_VECTOR(N-1 downto 0);
  begin
    V := VECTORS;
    WRITE(OUTVECT,V);
  end process WRITE_VECT;
end FIO;
entity IBVS is
  generic(N:INTEGER;PER: TIME);
  port(PLAY: in BIT; BVOUT: out BIT_VECTOR(N-1 downto 0));
end IBVS;
architecture FIO of IBVS is
  type INP_COMB is file of BIT_VECTOR;
  file INVECT: INP_COMB is in "TEST.VECT";
begin
  READ_VECT: process
    variable LENGTH: NATURAL := N;
    variable V: BIT_VECTOR(LENGTH-1 downto 0);
  begin
    wait on PLAY until PLAY = '1';
    loop
      exit when ENDFILE(INVECT);
      READ(INVECT,V,LENGTH);
      BVOUT <= V;
      wait for PER;
    end loop;
  end process READ_VECT;
end FIO;
```

图3-37 从格式化文件中读写

1) 输出文件（OUTVECT）和输入文件（INVECT）实际上是同一个主机文件“TEST.VECT”，必须在不同的声明域进行声明。

2) 在某次仿真中，不可试图对同一主机文件同时进行读写。因而，在先前的例子中，实

体OBVS和IBVS不可在同一仿真中执行。

这些条件保证文件在给字仿真中，文件没有处在inout模式。

2. 文本I/O

正如所讨论的，格式化的文件是不可为人所理解的。而且，在VHDL仿真以外，在主机操作系统环境之中创建这种文件也不容易。在许多情况中，可能需要在仿真过程中读取通过文本编辑器创建的文件或汇编器输出的目标代码。VHDL针对这些应用，提供了文本I/O。

文本I/O的基本声明在包含有库STD的包集合TEXTIO之内。正如上面所讨论的，库名STD总是可见的，但是TEXTIO名字或内容必须通过use子句来变得可见。TEXTIO中的前两个声明如下：

```
type LINE is access STRING;
type TEXT is file of STRING;
```

类型LINE被声明为存取类型STRING，存取类型用于指向存储位置的变量。这些指针在仿真过程中创建和释放。因而，空间并非永远分配给这些指针。(存取类型的细节可以在更高级的关于VHDL的书中找到。) 类型TEXT是基类型为STRING的文件类型。类型STRING用来在主机文件系统之中表示各种各样的输入数据。

根据这些定义，可以声明一个文本文件。例如，在从外部生成的文件中读取位向量的情况下，可以这样声明：

```
file INVECT: TEXT is "TVECT.TEXT";
```

因而，逻辑名为INVECT的文本文件映射到名为TVECT.TEXT的主机文件。

VHDL文本文件中的数据是用行组织的，每行元素的数量各异。从文本文件中读取数据时，首先执行READLINE命令读取整个一行，紧跟着用若干READ命令以读出该行中的单个元素。这些函数在包集合TEXTIO之中声明。要举例说明它的应用，考虑如何从上述定义的文本文件之中读出位向量，在本例中，可以这样调用读过程：

```
READLINE(INVECT,VLINE);
```

其中VLINE是类型LINE的变量，然后是一系列的下述调用

```
READ(VLINE,V);
```

其中V是类型Bit_Vector的变量，该命令的每次调用从VLINE中读出一个位向量，并将其复制到变量V中。

为了说明整个过程，我们以用文本编辑器创建的文本文件“TVECT.TEXT”的读过程为例，该文件内容如下：

```
000
001
010
011
100
101
110
111
000
```

图3-38给出了读该文件的VHDL代码。对于这个例子，类属N被置为3，在仿真过程中，当运行至'1'时，进入while循环。每次循环读出一行，并且该行唯一的位向量被取出并输出给

BVOUT。在这个例子中，只调用了一次 READ，这是因为每行只有一个位向量。如果在每行有多个位向量，则READ过程将被调用多次。当到达文件末尾时，循环退出。进程返回到进程顶部的wait语句，并挂起，直到信号PLAY上发生其他事件。

```
entity TBVS is
  generic(N:INTEGER;PER: TIME);
  port(PLAY: in BIT;
       BVOUT: out BIT_VECTOR(N-1 downto 0));
end TBVS;
use STD.TEXTIO.all;
architecture TIO of TBVS is
begin
  process
    variable VLINE: LINE;
    variable V:BIT_VECTOR(N-1 downto 0);
    file INVECT: TEXT is "TVECT.TEXT";
  begin
    wait on PLAY until PLAY = '1';
    while not(ENDFILE(INVECT)) loop
      READLINE(INVECT,VLINE);
      READ(VLINE,V);
      BVOUT <= V;
      wait for PER;
    end loop;
  end process;
end TIO;
```

图3-38 用来读取类型TEXT的输入文件的VHDL代码

对文本文件的写通过使用WRITE命令装配行来完成，然后用WRITELINE命令将整个行写入文件。这里不说明这个进程，而把它作为本章后的练习。

注意 在这个例子中，READ函数执行从类型TEXT到类型Bit_Vecter的转换。对目标类型限制如下：

```
Boolean          Integer
Bit              Real
Bit_Vector       String
Character        Time
```

```
package RFP is
  type POWER is range 0 to 1E9
    units pW;
      nw = 1000 pW;
      uW = 1000 nw;
    end units;
  type  FREQUENCY is range 0 to 1E9
    units Hz;
      KHz = 1000 Hz;
      MHz = 1000 KHz;
    end units;
  type RF_SIGNAL is
  record
    STRENGTH: POWER;
    FREQ: FREQUENCY;
  end record;
end RFP;
```

图3-39 包括类型声明的包集合

尽管容易理解，注意除了Time，其他物理类型都没有包括在内，也未包括用户定义的枚举类型。为了说明如何克服这种类型限制，考虑用VHDL对雷达信号进行建模的情况。VHDL代码见图3-39，在包RFP中先声明了类型POWER和FREQUENCY。然后，声明了记录类型RF_SIGNAL，字段为STRENGTH和FREQ，类型分别为POWER和FREQUENCY，假设创建一个文本文件SIQNAL.IN它包括：

```
10 26
30 25
27 24
14 27
```

该文件每行包括两个用空格分隔开的整数值，代表类型RF_SIGNAL的一个记录值。第一个整数表示信号强度，单位是皮瓦；第二个整数表示信号频率，单位是兆赫。图3-40的实体TB当信号RSIG为'1'时读取文件。对于每次循环，READLINE命令从文件中读取一行，包括两个整数，然后使用READ命令将两个整数读出，并与适当的物理类型基本单位（pW或MHz）相乘，将整数转化成适当的类型。结果值写入信号RFSIG的相应字段。图3-41之中显示了在Synopsys仿真程序中得到的实体TB的仿真结果。

```
entity TB is
end TB;
use work.RFP.all;
use STD.TEXTIO.all;
architecture TFILE5 of TB is
  signal RFSIG: RF_SIGNAL;
  signal RSIG: BIT;
begin
  process
    variable TEMP_STRENGTH: INTEGER:= 0;
    variable TEMP_FREQ: INTEGER:= 0;
    variable L: LINE;
    file INFILE: TEXT is in "SIGNAL.IN";
  begin
    wait on RSIG until RSIG = '1';
    while not(ENDFILE(INFILE)) loop
      READLINE(INFILE,L);
      READ(L,TEMP_STRENGTH);
      RFSIG.STRENGTH <= TEMP_STRENGTH*1 pW;
      READ(L,TEMP_FREQ);
      RFSIG.FREQ  <= TEMP_FREQ*1 MHz;
      wait for 10 ns;
    end loop;
  end process;
  RSIG <= '1';
end TFILE5;
```

图3-40 用TEXTIO读取雷达信号

3.8 VHDL的形式特征

本章通过选择一些例子，介绍VHDL语言的主要特征。该语言的形式化定义在《*IEEE*标准*VHDL*语言参考手册》（*IEEE standard VHDL Language Reperence Manual*）中给出，提供了该语言的语法和语义。这两个项是这样定义的。

```
/
  0 NS
    SMON1:    ACTIVE /TB/RSIG (value = '1')
    SMON:     ACTIVE /TB/RFSIG (value = (STRENGTH => 10 PW,
              FREQ => 26000000 HZ))
  10 NS
    SMON:     ACTIVE /TB/RFSIG (value = (STRENGTH => 30 PW,
              FREQ => 25000000 HZ))
  20 NS
    SMON:     ACTIVE /TB/RFSIG (value = (STRENGTH => 27 PW,
              FREQ => 24000000 HZ))
  30 NS
    SMON:     ACTIVE /TB/RFSIG (value = (STRENGTH => 14 PW,
              FREQ => 27000000 HZ))
```

图3-41　读取雷达信号的仿真结果

语法：规定语言语句形式的一组规则。

语义：语言基本结构的含义。

VHDL语言的语法用产生规则给出。例如，进程语句的产生规则见图3-42，第一条规则用关键字（begin, end）和其他组成部分，如可选的标号、声明域和语句域，给出进程语句的通用结构。该规则同时给出这些部分相互关联的方式。下面的两条规则说明了在语句部分和声明域中语言元素是如何表示的。这些语言元素有它们自己的产生规则。这种语法定义从特殊到一般，一直继续到基本语言元素，到那时则再无产生规则了。

```
process_statement ::=
  [process_label:]
    process[(sensitivity_list)]
      process_declarative_part
    begin
      process_statement_part
    end process[process_label];

process_statement_part ::=
  {sequential_statement}

sequential_statement ::=
    wait_statement
  | assertion_statement
  | signal_assignment_statement
  | variable_assignment_statement
  | procedure_call_statement
  | if_statement
  | case_statement
  | loop_statement
  | next_statement
  | exit_statement
  | return_statement
  | null_statement

process_declarative_part ::=
  {process_declarative_item}

process_declarative_item ::=
    subprogram_declaration
  | subprogram_body
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | variable_declaration
  | file_declaration
  | allias_declaration
  | attribute_declaration
  | attribute_specification
  | use_clause
```

图3-42　进程语句产生规则

语言参考手册也包含语言的语义定义。例如，对于进程语句，其语义规则为：

一个进程的执行包括对其中语句序列的重复执行。在进程语句之中的最后一个语句执行之后，执行将立即从语句序列的第一个语句开始。

这个规则显式地说明了进程执行是重复的并且进程是循环的。注意：语法规则决无此意。

随着读者对VHDL语言了解的加深，遇到语言细节问题参考《语言参考手册》（*Language Reference Manual*）十分必要。然而，该文档很抽象，因而必须花力气去理解。

3.9 VHDL 93

从1987年对VHDL的最初标准化之后，IEEE接受并审查了对语言进行修改的建议。1993年，IEEE发布了新的标准1076-1993并入了这些修改。改变不是很大，不用它们可以同样有效地进行建模。尽管如此，我们在这里对它们进行一下总结，从而使读者可以充分利用该语言的功能。但是这里必须提醒读者，从1999年起，一些VHDL供货商已经考虑了1993年版标准之中的所有变化。这里给出最为重要的改变的例子，用户可参考1993年版语言参考手册以求细节。亦可参考本书末尾提供的参考文献。注意，几乎所有VHDL1987代码在VHDL 1993标准下仍然有效。

3.9.1 词汇字符集

VHDL 87使用ASCII字符集，是ISO字符集256个字符中的前128个。VHDL 93使用全部的ISO字符集。包括用于其他语言的附加符号。如：德语中的元者变音。

3.9.2 语法变化

1987标准之中相对死板的规则已被放松，例如，标识符：

```
\7404INV\
```

现在是合法的。1993标准的标识符（叫做扩展标识符）必须使用反斜杠加以分隔。它们可以用数字开头，而且对大小写敏感。因而，\7404inv\与上述扩展标识符是不同的。扩展标识符可以表示保留字，如\entity\是合法标识符。扩展标识符一般与任何短标识符不同，如\XYZ\与XYZ不同。

在VHDL 87中，实体和元件声明的开头与结束语法不同：

```
entity X is
 --------
 --------
end X;
component X
 --------
 --------
end component;
```

现在元件可以同样写为：

```
component X is
 --------
 --------
end X;
```

VHDL 93中元件声明的语法为：

```
component identifier [is]
    [local_generic_clause]
    [local_port_clause]
end component [component_simple_name];
```

与此一致的规则也适用于实体、结构体、包声明、包体、配置、元件、块、进程、记录、if和case语句和子程序。这使得用户可以写出与语法形式更加一致的代码。

3.9.3 进程和信号定时及新的信号属性

1) **延期进程**（postponed process）。在第4章图4-7及其附近的讨论中，我们讨论了如下这种情况：其中一些基于delta延时的仿真周期在仿真时间前进之前发生，直到这些delta周期过去，模型的状态才能够确定，从建模者的观点出发，其他监测这些模块的模块将会产生不正确的响应。VHDL 93引入了延期进程的概念，此类进程直到时间点的最后一个delta周期才被激活。考虑下列代码：

```
A  <= '0', '1' after 1 ns;
B  <= not A;
assert (B = not A)
  report "B is equal to A!"
```

在稳定的状态条件下，不可能发现错误。当A变化时，直到delta之后B才取得A的非的值。因此，在一个delta周期之中A与B相等，断言语句检测出一个错误。然而，假设与进程等价的断言语句的代码为：

```
postponed assert (B = not A)
  report "B is equal to A!"
```

使用这种结构，断言只有在B改变之后才会执行，因而不会报告出错。这种延期表示可用于进程语句、并发信号赋值语句、过程调用和断言语句。

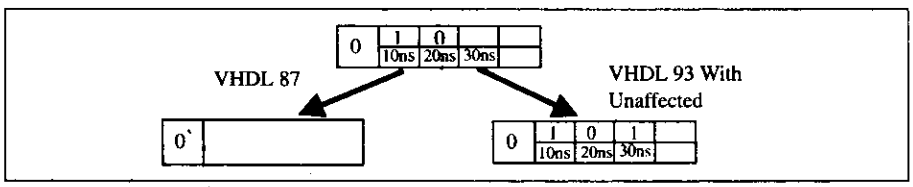


图3-43 VHDL93中的保留事务

2) **惯性延时控制**（inertial delay control）。现在可以直接控制惯性延时，而不受传输延时影响，考虑下面的代码：

```
A <= reject 2 ns inertial B after 6 ns;
A <= reject 2 ns B after 6 ns;
```

上述的每个语句都隐含了2ns的惯性延时和6ns的传输延时。惯性延时仍为缺省的；这样，关键字inertial的使用是可选的。在VHDL87中，必须通过编写下述代码以达到同样的效果。

```
C <= B after 2 ns;-- inertial delay and prop delay of 2ns.
A <= transport C after 4 ns; --- pure propagation delay of
                             --- 4 ns.
```

3) **不受影响的关键字**（unaffected keyword）。考虑下面的并发信号赋值语句：

```
A <= '1' after 10 ns when CON = '1' else A;
```

该语句的一个自然的解释是当CON='1'时，A在10ns之后的值为'1'ns。当CON='0'时，A无变化，这个解释并非是严格正确的。A的当前值并不改变。1) A上将会进行事务，因而A'*transaction*将发生切换，且A'*quiet*将会是FALSE。2) 作为波形更新规则的结果，A上所有其他未解决的事务将如图3-43之中左边分支所示的那样被删除。在VHDL 93中，条件信号赋值语句可以写为：

```
A <= '1' after 10 ns when CON = '1' else unaffected;
```

通过使用关键字*unaffected*，A上的旧事务在CON='0'时被保存，如图3-43中右边分支所示，而且A'*transaction*不触发且A'*quiet*为TRUE。

条件信号语句的另一个改变是其他分支变为可选，因而：

```
A <= '1' after 10 ns when CON = '1';
```

等价于在*else*子句中使用*unaffected*关键字。关键字*unaffected*同样适用于选择信号赋值语句。

4) **信号属性'*driving*和'*driving_value***。在VHDL 87中，模式为out的接口信号是不可读的，可以使用模式*inout*。但是更多的建模者倾向于将这种模式保留给真正的双向数据。考虑如图3-44所示的VHDL 93代码。

本实体SOUT的驱动器的值使用属性'*driving_value*读回。另一个属性'*driving*的类型为Boolean，如果驱动被连接则为TRUE，否则为FALSE。

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity DRIVING is
  port(SOUT: out STD_LOGIC);
end DRIVING;
architecture ALG of DRIVING is
  signal INT_SOUT: STD_LOGIC;
begin
  SOUT <= '0','1' after 10 ns,'0' after 20 ns,
          '1' after 30 ns;
  INT_SOUT <= SOUT'driving_value;
end ALG;
```

图3-44 属性'DRIVING_VALUE

3.9.4 新操作符

1) **移位和循环操作符**。操作符形式如下：

```
array_object operation N
```

其中数组对象是一维数组，元素的类型是Bit或Boolean。N是指定运算进行次数的整数。这些运算是：a) 逻辑左移（sll）—左边的输入是输入数组元素类型的'LEFT；b) 逻辑右移（sll）—右边的输入是输入数组元素类型的'LEFT；c) 算术左移 (sla) —最右边的值保留，所有位移入左边的下一个位置（保持符号不变）；d) 算术右移 (sla) —最左边的值保留，所有的位移入右边的下一个位置（符号保持不变）；e) 逻辑左循环（rol）；f) 逻辑右循环（ror）。包集合STD_LOGIC_SIGNED和STD_LOGIC_UNSIGNED包含用于类型STD_LOGIC_VECTOR

的移位操作符。包集合STD_LOGIC_ARITH包括用于SIGNED、UNSIGNED和一维STD_LOGIC数组的移位操作符。

2) xnor操作符。用于类型Bit、Boolean和这些类型的一维数组的内置操作符。所有这些逻辑操作符具有相同的优先级。像*nand*和*nor*一样，*xnor*操作符是不可结合的。在包集合STD_LOGIC_1164中，类型STD_LOGIC和STD_LOGIC_VECTOR重载该操作符。

3.9.5 结构化模型的改进

对结构化建模的过程有三个主要的改进：

1) 直接实例化：VHDL 87中，实例化元件涉及元件声明、元件实例化及元件配置说明。在VHDL 93中，可以像图3-45所示那样直接进行实例化，从而大大减少了代码。

2) 输入表达式：在VHDL 93中，不允许在输入端口上使用表达式，注意图3-45中输入端口上值'1'的应用。

3) 递增绑定：VHDL 93允许一个配置体覆盖前面的配置说明，在图3-45中，该特性被用来给NAND门反标一个更加精确的延时值。

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity NAND2 is
  generic(DELAY: TIME);
  port(A,B: in STD_LOGIC; C: out STD_LOGIC);
end NAND2;
architecture DF of NAND2 is
begin
  C <= not(A and B) after DELAY;
end DF;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity INVERTER is
  port(I: in STD_LOGIC; O: out STD_LOGIC);
end INVERTER;
architecture STRUCTURE of INVERTER is
begin
  C1: entity work.NAND2(DF)    --- direct instantiation
    generic map(DELAY => 1.0 ns)  -- expressions
    port map(A => I, B => '1',C =>O); --on input ports
end STRUCTURE;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
configuration NEW_TIMING of INVERTER is
  for STRUCTURE
    for C1: entity work.NAND2(DF)--incremental binding
      generic map(DELAY => 1.23 ns);
    end for;
  end for;
end NEW_TIMING;
```

图3-45　VHDL 93中新的结构化建模特征

3.9.6 共享变量

在VHDL标准化过程的早期，有些用户要求将共享变量加入到语言机制之中。这对于系统

级仿真是非常有效的。然而反对意见是硬件的对应性较弱。另外一个关键是：应该在语言之中加入何种存取控制机制来支持共享变量？VHDL 93包括了共享变量，但没有内置的存取控制机制（目前似乎更倾向于使用监视器），因而用户必须保证不会同时有两个或更多的代理对共享变量进行访问。图3-46说明了一种情况，其中两个进程P1和P2共享对共享变量LATEST的存取。LATEST的值取决于PROC1或PROC2中哪一个进程最近对它进行存取。注意到如果START1和START2同时进行正向转换，则无法预料到LATEST终了时为何值。然而如果使用并行处理器执行该模型，如果一个处理器在变量存取时干扰了另一个，则结果可能不是这两者之中的一个。因而必须存在一种存储控制机制。

```
entity SHARED_VAR is
  port(START1, START2: in BIT);
end SHARED_VAR;

architecture ALG of SHARED_VAR is
  type PROC_NUM is (PROC1,PROC2);
  shared variable LATEST: PROC_NUM;
begin
  P1: process
  begin
    wait until START1 = '1';
    LATEST := PROC1;
  end process;
  P2: process
  begin
    wait until START2 = '1';
    LATEST := PROC2;
  end process;
end ALG;
```

图3-46 共享变量

3.9.7 改进的报告能力

VHDL 87中的报告机制是人为的，使用断言语句可以写为：

```
assert FALSE
report "Executing Processor 1"
severity note;
```

另外一个问题是REPORT子句只接受字符串参数。如果想在消息之中包括整数变量I，建模者必须写一个类型转换函数，将I变成等价的字符串，在VHDL93可以这样写：

```
report  "Executing processor"&Integer'image(I);
```

它将I的值包含在消息输出之中。关键字*report*是新的报告命令，'*image*属性执行类型转换的功能。

3.9.8 通用编程特征

1) **非纯函数**（Impure function）。这是带有副作用的函数。非纯函数不是单纯地从调用接口进行通信，它可以读或修改外部数据。使用这种类型的函数，可以用外部的种子编程实现

随机数产生器或C的GET_CHAR过程，使得可以从外部文件读入字符并将字符指针移向下一个要读的字符。图3-47给出了一个非纯函数如何读取并递增一个外部信号。在VHDL 87之中，这是不允许的。

```
entity IMPURE_F is
end IMPURE_F;

architecture ALG of IMPURE_F is
  signal INT,INT_INC: INTEGER := 0;
  impure function INC_AND_UPDATE
    return INTEGER is
  begin
    INT <= INT + 1;
    return INT;
  end INC_AND_UPDATE;
begin
  INT_INC <= INC_AND_UPDATE;
end ALG;
```

图3-47　非纯函数的使用

2) **外部接口**（foreign interface）。VHDL 93包集合Standard中，属性FOREIGN定义为：

```
attribute FOREIGN is STRING;
```

假设想调用用户接口输入，从而在仿真中控制模型，可以声明这样的函数：

```
function USER_INP(INP: in  COMMAND) return Integer;
```

可以用声明与一个系统程序相关联：

```
attribute FOREIGN of USER_INP: function is "VHDL_USER_IN";
```

其中VHDL_USER_IN是用C和X windows编写的本地系统软件。其他对象也可以是FOREIGN。用其他编程语言或HDL规范实现的实体也可以使用同样的机制。

3.9.9　文件I/O

在VHDL 93中，文件可定义为参数传递给进程的第四个对象类。文件是无模式的。可以对一个文件进行读写。在新的包集合Standard中，打开文件的类型定义为：

```
type FILE_OPEN_KINK is (READ_MODE, WRITE_MODE, APPEND_MODE);
```

这样，读的文本文件可以声明为：

```
file IN_MEM: TEXT open READ_OPEN is "MEM_CONTENTS"
```

然后，允许写同一文件的另一个声明是可能的：

```
file OUT_MEM: TEXT open WRITE_OPEN is "MEM_CONTENTS"
```

APPEND_MODE是写模式，该模式下信息被加入现有文件的尾部。可隐式声明函数FILE_OPEN和FILE_CLOSE，文件INPUT（键盘）和OUTPUT（显示屏）则声明为打开。

3.9.10　组

在VHDL 87中，可以使用属性来注释描述，也可以用工具如综合工具提取该注释。然而，

这种注释只可用于单个项。VHDL 93引入了组的概念，其中许多相关项可以用一个值来注释。一般的应用是在信号路径上对时间约束的注释，假设BEGIN_PT和END_PT是两个信号对象：

```
group BEGIN_TO_END: PATH(BEGIN_PT,END_PT);
attribute PROPAGATION_DELAY: TIME;
attribute PROPAGATION_DELAY of BEGIN_TO_END group is 100 ns;
```

这个VHDL 93代码建立在信号BEGIN_PT和END_PT之间的受限延时上（注意这对仿真没有影响）。

3.9.11 位串文字的扩展

如果表示十进制数字35，可以用三种位串文字的形式表示：

B"100011"
O"43"
X"23"

在VHDL 87中，这些只能够赋给类型为Bit_Vecter的对象。在VHDL 93中，只要该数值系统与二进制、八进制、十六进制数值系统兼容，就可以同任何数值系统一起使用。例如，一个STD_LOGIC_VECTOR (7 downto 0)可以被初始化为O“43”但不可以为O“4Z”。

3.9.12 对标准包的增加与修改

在VHDL 93中，对VHDL 87包集合STANDARD进行了如下修改：

1) 增加

a) 类型：FILE_OPEN_KIND和FILE_OPEN_STATUS。

b) 子类型：DELAY_LENGTH。

c) 所有标准语言操作符的接口定义，如AND，*。

d) 属性FOREIGN。

2) 改变

ASCII字符集被ISO字符集取代。

3.10 小结

本章的目的在于让读者对VHDL有一个基本的了解。这里的介绍很不完全，尤其对于那些在后续章节中要使用的模型。其他语言特征将在建模过程中说明。这些模型将给出完整的示例，其中语言的全部有用特征被放在一起。通过本章的学习，读者将了解足够的VHDL知识，从而可以使用该语言并且能够理解模型的结构。

习题

3.1 建立三输入AND门的VHDL模型。使用下面的测试程序来进行仿真：

```
entity TB is
end TB;
architecture AND3T of  TB is
  signal X,Y,Z,O: Bit;
  component AND3
    port(I1,I2,I3: in Bit; O: out Bit);
```

```
  end component;
  for C1: AND3 use entity work.AND3(DATA_FLOW);
begin
  C1: AND3
    port map(X,Y,Z,O);
  X <= '1' after 1 ns, '0' after 2 ns, '1' after 3 ns,
       '0' after 4 ns, '1' after 5 ns, '0' after 6 ns,
       '1' after 7 ns;
  Y <= '0' after 1 ns, '1' after 2 ns, '1' after 3 ns,
       '0' after 4 ns, '0' after 5 ns, '1' after 6 ns,
       '1' after 7 ns;
  Z <= '0' after 1 ns, '0' after 2 ns, '0' after 3 ns,
       '1' after 4 ns, '1' after 5 ns, '1' after 6 ns,
       '1' after 7 ns;
end AND3T;
```

3.5

3.6

3.2 为本章计数器实例中的实体OPAR3开发一个VHDL描述。要求OPAR3的构架是纯结构化的。可以使用习题3.1完成的三输入AND门。为OPAR3实体产生一个测试程序并进行仿真。

3.3 考虑下面所示组合逻辑电路的真值表，其逻辑框图见图3-48：

X2	X1	X0	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	1
1	1	1	0

3.7

电路延时为DEL，

a) 开发一个有效的算法级VHDL描述。

b) 开发一个有效的数据流VHDL描述。

3.8

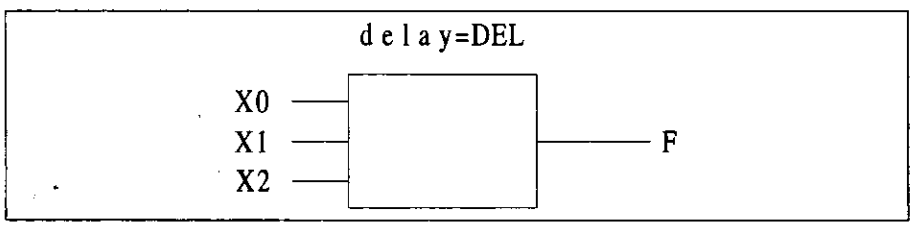


图3-48　习题3.3的电路框图

3.4 考虑下面的VHDL代码：

```
entity X is
  port(A: in Bit_Vector(7 downto 0);
       Z: out Bit_Vector(7 downto 0));
end X;
architecture ALG of X is
begin
  process(A)
    variable ZV: Bit_Vector(7 downto 0);
  begin
    for I in 7 downto 0 loop
      ZV(7-I) := A(I);
```

3.9

3.10

```
    end loop;
    Z <= ZV;
  end process;
end ALG;
```

a) 如果A从全0变为10101111，Z的输出是多少?

b) 用一句话描述实体X的功能。

3.5 写出一个位触发器的实体及算法级结构体，该触发器改变一个8位字的位的顺序，即把10101111变为11110101。

3.6 Fibonacci数的序列为0，1，1，2，3，5，8，13，21，等等。该序列可以被定义如下：

```
F0 = 0
F1 = 1
Fi = F(i-1) + F(i-2)   i > 1
```

除了序列中的前两个数，序列中的第i项为第i-1和i-2项之和。假设器件由一个START命令激活，开始计算序列中的前20项（F0到F19），并在60ns之后输出所有20个结果。为该器件建立一个精确的VHDL算法级描述。

3.7 设计一个5选2代码检测器，器件接受五位并行字，如果所有的代码字正好只有两个1则输出一个逻辑1，否则输出逻辑0。执行下面步骤：

a) 为检测器产生一个VHDL实体声明。

b) 为检测器产生一个算法级行为域结构体。

c) 为检测器产生一个数据流行为域结构体。

d) 为检测器产生一个结构化结构体。

e) 使用VHDL仿真系统对这三个结构体进行仿真，以验证每个模型的正确性。

f) 为结果产生一个总结报告。

3.8 设计一个并行偶校验器，其输入为四位编码。如果所有代码位校验和为偶，即代码字有偶数个1，则校验器输出逻辑1。执行下面的步骤为电路建模：

a) 为检测器产生一个VHDL实体声明。

b) 为检测器产生一个算法级行为域结构体。

c) 为检测器产生一个数据流行为域结构体。

d) 为检测器产生一个结构化结构体。

e) 仿真这三个实体以验证每个模型的正确性。

准备一个报告，报告中包括：

a) 每个模型创建过程描述。

b) 列出VHDL源代码。

c) 带注释的仿真结果。

3.9 设计者希望在没有详细的设计定义的情况下建立一个局域网协议模型。为信号接口建模需要使用那种定义？为什么？

3.10 回答下面有关词法元素的问题：

a) 说出分隔符和定界符之间的区别。

b) 画出表示两个类及两种抽象字文字类型之间关系的图，给出这些类和类型的组合实例。

c) 解释‘A’和“A”之间的区别。给出与每两个值相适应的类型声明实例。

d) 解释标识符和保留字的区别。

3.11 下面的位串文字哪种是正确的？对于每个有效的位串文字，判定其长度及其在VHDL语言中所表示的值。

a) B “1011_0101_1001”

b) B “0001_1011”

c) B “1001_0111”

d) B “01010111”

e) X “B5_CD”

f) X “3HA4”

g) O “237”

h) O “233_814”

i) O “0011_7322”

j) “1011”

k) “1011_0001”

3.12 下面哪些是正确的用户定义的标识符？对于每个无效的标识符，陈述其错误的理由。

a) Help

b) 2nd_item

c) Case

d) small_device

e) This_lable_might_be_too_long_what_do_you_think_about_the_possibility:

f) BODY

g) register

h) REPORT

i) branch

j) _name_

3.13 VHDL语言要求所有用户定义的标识符必须是不同的，下面的标识符对中哪些是不同的？

a) MY_NAME，MYNAME

b) cat，CAT

c) Dog，dOg

d) cow，cows

e) two，too

3.14 下面哪些是有效的抽象文字？对于每个有效的抽象文字，给出它的不带指数的十进制值，并说明该文字是否是十进制数、整数、实数或基。

a) 2e5

b) 5#224_33#

c) 2,534,215

d) 2_534_215

e) 2.5e2

f) 12#A_B#

g) 16e-2

h) 16.0e-0.5

i) .5

j) 2#1011_0001#e10

3.15 比较并对照内置的整数、自然数和正数数据类型。

3.16 对下面的数据类型定义特定的类型声明。

a) 四值逻辑系统MVL4，具有‘0’、‘1’、‘X’、‘Z’四个值。值‘0’和‘1’表示一般的逻辑意义。值‘X’表示逻辑值未知（可能是0或1）。值‘Z’表示为高阻状态。任何非初始化的数据值都是‘X’。

b) 枚举数据类型DAY_OF_WEEK。

c) 具有整数值1到120的数据类型ATOMIC_NUMBER。

d) 具有实数值$0.00到$1000.00的数据类型COST。

e) 降序数据类型DEC_16，整数值的范围从15到0。

f) 16位降序下标寄存器数据类型REGISTER_16_BIT_DESCENDING，其下标值为上面声明的类型DEC_16，以及类型Bit的组件值。

g) 二维表TRISTATE_TABLE_2D，具有下标值和表项，类型均为TRISTATE（参阅文中的枚举类型）。

h) 记录数据类型PERSONNEL，包括姓（LAST）（最多20位字符）、名（FIRST）（最多20位字符）、姓名缩写（MID），以及社会安全序号（SOC_SEC）。

3.17 对于习题3.16的数据类型，写出下列常数的声明。

a) 具有类型MVL4的值‘X’的常数DON’T_CARE。

b) 具有类型DAY_OF_WEEK的值WEDNESDAY的常数MID_WEEK。

c) 具有类型ATOMIC_NUMBER的值1的常数HYDROGEN。

d) 由REGISTER_16_BIT_DESCENDING类型表示的（-1）的2的补码MINUS_1。

e) 类型TRISTATE_TABLE_2D的常数TRISTATE_AND，它定义了两个TRISTATE数据对象的逻辑“与”操作。假设其值为1，即门的高阻输入被作为逻辑1。

f) 属于PERSONAEL类型的常量MY_PERSONNEL_RECORD，定义你自己的数据。

3.18 对于习题3.16，写出下列变量和信号的声明：

a) 类型为MVL4的信号RESET。

b) 变量CURRENT_DAY，它的值等于本星期的当前天。

c) 变量CHICKEN_PRICE，它的值等于现在的鸡肉价钱。

d) 信号REGISTER_1、REGISTER_2、REGISTER_3和REGISTER_4、它们以降序存放16位数据值。

3.19 对于习题3.18声明的变量和信号以及习题3.17声明的常量，写出下面的变量和信号赋值语句：

a) 把变量CURRENT_DAY赋值为MID_WEEK的变量赋值语句。

b) 每次执行时把CHICKEN_PRICE增加百分之一的变量赋值语句。

c) 使用REGISTER_4的位对REGISTER_2的位选择求补的信号赋值语句。REGISTER_4的位若为1，则REGISTER_2对应的位求补，其他的位保持不变，如：

```
REGISTER_2 initial value:    1011 0101 0001 1100
REGISTER_4 value:            1111 0000 1101 0101
                             -------------------
REGISTER_2 final value:      0100 0101 1100 1001
```

3.20 对于习题3.17声明的常量TRISTATE_AND，假设V1、V2和V3都是TRISTATE类型的信号。V1和V2是AND门的输入，V3为输出，门延时为10ns。

a) 写出当任一输入变化时，门输出更新的进程。

b) 写出随机2输入AND门输出更新的子程序，输入和输出信号都是TAISTATE类型，写出实现给定门调用信号V1、V2和V3的调用语句。

c) 对于在a和b中产生的模型，回答下面问题：

1) 对比这两个模型。

2) 讨论这两个模型的优、缺点。

3) 讨论这两个模型的应用。

3.21 根据下面的修改，重复习题3.20过程：对三态门增加ENABLE输入，当ENABLE为“0”时，输出为“Z”。当ENABLE为“1”时，输出为输入数据“与”的结果，与TTL电路类似，ENABLE的值为“Z”时作为“1”处理。

3.22 考虑图3-16和图3-17预定义的数据类型，为下面的表达式找出合适的值：

a) Character'pos('A')
b) SEVERITY_LEVEL'pos(WARNING)
c) Character'pos(STX) > SEVERITY_LEVEL'pos(ERROR)
d) Character'val(65)
e) SEVERITY_LEVEL'val(2)
f) Character('A') > Character('a')
g) SEVERITY_LEVEL'(NOTE) < SEVERITY_LEVEL'(FAILURE)
h) Boolean'high
i) Natural'low
j) Positive'left
k) Bit'right
l) Character'pred('A')
m) Character'pred('+') = Character'succ(')')

3.23 考虑下面的类型和信号声明：

```
type ANIMAL is (LION, DOG, CAT, MOUSE, HORSE, FOX, COW);
type ANIMAL_VECTOR is array (ANIMAL range <>) of Natural;
signal A1: ANIMAL := HORSE;
signal A2: ANIMAL;
signal BV1: ANIMAL_VECTOR (FOX downto CAT):=(1, 10, 2, 6);
```

指出下面的表达式哪些是有效的？对于有效的表达式，决定其在t=0时刻的值。

a) A2

b) BV1 (MOUSE)

c) MOUSE<LION

d) ANIMAL'val(BV1(FOX))

e) ANIMAL'pos(CAT)

3.24 考虑下列标准VHDL的类型和信号声明：

```
type COURSE is (MATH, SCIENCE, SOCIAL_STUDY, LANGUAGE);
type COURSE_VECTOR is array (Natural range <>) of COURSE;
```

```
signal A1: COURSE := SCIENCE;
signal A2: COURSE;
signal BV1: COURSE_VECTOR(10 downto 8):=(MATH, LANGUAGE,
  MATH);
```

指出上面的表达式哪些是有效的？对于有效的表达式，指出t=0时刻的值。

a) A2

b) BV1 (10)

c) SCIENCE>SOCIAL_STUDY

d) COURSE'val(1)

e) COURSE'pos(BV1(8))<SCIENCE

3.25 展示如何声明一种任意长度的数组类型，其中数组的元素为五值逻辑类型的值U，0，1，X，Z。

3.26 给出下面的声明：

```
Signal X: Bit_Vector(0 to 3);
```

找出：

a) X'RANGE

b) X'LEFT

c) X'RIGHT

d) X'HIGH

e) X'LOW

3.27 图3-49给出了字符的三维立方体，给出声明并初始化该立方体值的变量所需要的声明。

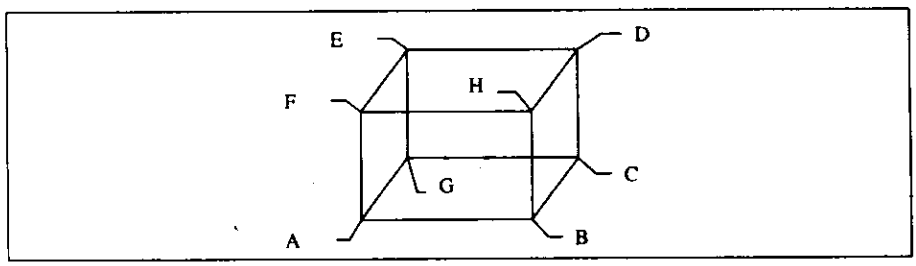


图3-49 习题3.27的三维字符立方体

3.28 图3-50的二维数组包括了整数类型元素。给出声明并初始化该数组值的变量所需要的声明。

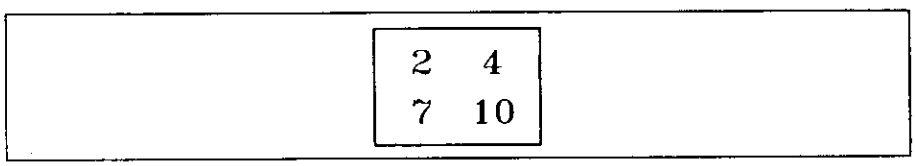


图3-50 习题3.28的二维数组

3.29 如果没有显式地初始化一个VHDL信号或变量，它的缺省值是多少？

3.30 回答下面有关Bit和Boolean数据类型的问题：

a) 解释Bit和Boolean类型的区别。

b) 对于逻辑操作应使用哪种类型？

c) 关系操作的结果为哪种类型？

d) if语句测试的表达式为哪种类型？

3.31 考虑下面三输入NOR门的VHDL描述：

```
entity NOR3 is
  generic (DEL: time);
  port (X1, X2, X3: in Bit; Y: out Bit);
end NOR3;
--
architecture GATE of NOR3 is
begin
  -- Assignment statements
end GATE;
```

下面哪一个赋值语句用于前面的模型中时可以实现三输入NOR门？可能不只一条语句有效。

a) Y<= X1 nor X2 nor X3

b) Y<= (X1 nor X2) nor X3

c) Y<= (X1 or X2) nor X3

d) Y<= not (X1 or X2 or X3)

e) Y<= not (X1 nor X2) nor X3

3.32 写出对位向量统计‘1’的个数的VHDL过程，该过程必须是通用的，即它必须接受任意长度的输入参数。

3.33 X的类型为Bit_vector(0到3)，为X写出一个加1函数。

3.34 下面的问题主要针对进程和wait语句：

a) 对于下面的进程，解释wait语句在仿真时的作用。

```
PROC1: process
  X <= X + 1;  --- X is type integer
  wait on R,Q until (U = '1') for 200 ns;
  X <= X - 1;
end process PROC1;
```

b) 对于下面的进程，解释当仿真开始时会发生什么事情，接着又会如何？

```
process
begin
  X  <= X + 1;
  wait on Y;
end process;
```

c) 对于下面的进程，仿真时会发生什么事情？

```
process
begin
  X <= X + 1;
end process;
```

3.35 写出与下面并发信号赋值语句等价的进程语句：

```
with Z select
```

```
A <= transport
  X and Y when 0,
  X or Y  when 1,
  not X   when others;
```

3.36 在下面的VHDL描述中，用圆圈指明哪一条信号赋值语句是不正确的：

```
entity EX1 is
end EX1;
architecture P7 of EX1 is
  signal A1: Bit;
begin
  B1: block
    signal X: Bit;
  begin
    X <= B2.Y;
    B2: block
      signal Y: Bit;
    begin
      Y <= B1.X;
      A1 <= B3.Z;
    end block B2;
  end block B1;
  B3: block
    signal Z: Bit;
  begin
    Z <= A1;
    Z <= B1.B2.Y;
    A1 <= Z or A1;
  end block B3;
end P7;
```

3.37 考虑如下VHDL代码：

```
package D is
  type D1 is array(0 to 2) of Integer;
  type D3 is array(0 to 1,0 to 1,0 to 2) of Integer;
end D;
use work.D.all;

entity IT is
  port(Q: in Bit; XOUT,YOUT: out Integer);
end IT;

architecture DO of IT is
begin
  process(Q)
    variable A: D3 :=
      (((16,8,7),(3,11,5)),((15,6,9),(17,9,4)));
    variable X: Integer;
    variable Y: Integer;
  begin
    X := 0;
    Y := 0;
    for k in 0 to 2 loop
      X := X + A(0,0,K) + A(1,1,K);
      Y := Y + A(0,1,K) + A(1,1,K);
    end loop;
    XOUT <= X;
    YOUT <= Y;
  end process;
end DO;
```

回答以下问题：

a) 画出由变量A表示的数据结构框架。

b) 假设该进程只执行一次。

1) 用文字和图表来解释该进程执行了哪种计算。

2) 给出XOUT和YOUT的结果值。

3.38 对于下面给出的真值表：

X2	X1	X0	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

产生一个隐含ROM实现的VHDL描述。

3.39 一个工程师希望为Motorola 68000微处理器系统建立VHDL模型，她希望仿真实际微处理代码的执行，并用汇编语言来编写68000程序，并且该程序可以经常改变。VHDL语言的哪些特征可以用于解决这种仿真情况？

3.40 根据下面的代码解释模型产生的结果。

```
arch X of Y is
begin
  for I in 1 to N generate
    S(I) <= V(I) nand W(I) after DEL;
  end generate;
end X;
```

3.41 给出电压、电流、功耗以及电阻的物理类型定义。并声明一个用于表达电压源的记录类型，即用一个字段表示电压源的电压输出，另一个字段表示自身的电阻，使用这些类型定义来计算电压源两端外部电阻的电流和功耗。

3.42 实现并应用下面的包集合：

a) 给出三态系统0，1，Z的枚举类型定义，其中Z为高阻状态。

b) 使用重载来定义系统的AND、OR和NOT操作，给出实现这些操作的电路。

c) 把a和b的结果放入一个包集合中。

d) 使用c中产生的包集合仿真下面的逻辑等式操作：E=(AB'+C'D)'。其中'表示取反。

3.43 为AND、OR和NOT门建立模型，并将其放入名为GATES的库中，在一个实现习题3.42逻辑等式的结构化构架中使用这些模型。仿真此构架。

3.44 建立一个实现习题3.42逻辑等式的非绑定结构化构架。用一个配置声明把这个构架绑定到GATES库中的模型（习题3.43），并仿真这个系统。

3.45 考虑下面的VHDL代码：

```
entity BLOCKS is
end BLOCKS;
architecture TEST of BLOCKS is
begin
  L1: block
    signal A: Integer := 10;
```

```
    signal B: Integer := 4;
  begin
    L2: block
      signal B: Integer := 5;
    begin
      A <= B after 5 ns;
      B <= L1.B after 10 ns;
    end block;
    B <= A after 15 ns;
  end block;
end TEST;
```

a) 解释上述块集合中信号的可见性情况。

b) 这段代码仿真时，作为时间的函数，所有信号的值是多少？

3.46 考虑下面来自Thomas Gray的著名诗歌“Elegy In A Country Church Yard”的几句话：

```
The curfew tolls the knell of parting day,
The lowing herd winds slowly o're the leigh,
The plowman homeward plods his weary way,
And leaves the world to darkness and to me.
```

本题的目的是使用文件和文本I/O。

a) 写出由独立单词构造成行的VHDL模型，使用文件I/O把每行写到一个文件中，再通过文件I/O读回，进行逐词比较以确保数据读写的正确性。使用一个断言语句输出成功完成测试的信息。同时使用仿真程序提供的信号和变量监视功能来证明这些单词被正确读回。

b) 使用一个文本编辑器把这首诗编辑到一个文件中，并使用文本I/O把该文件读入VHDL模型。用信号和变量监视功能来证明这些单词被正确读入。

第4章 基本的VHDL建模方法

本章讲述基本的VHDL建模方法，首先介绍VHDL语言如何支持传播延时及并发的建模，接着将详细讨论VHDL时序模型，然后讨论如何用VHDL为组合逻辑和时序逻辑建模，再给出数字逻辑设计中常用的基本单元的模型，最后解释如何使用测试包来验证模型的有效性。

4.1 用VHDL为延时建模

在数字逻辑中，延时是一个非常重要的方面。所以，任何硬件描述语言都必须包括一个或多个延时模型。这一节里描述了VHDL语言的延时特性。

4.1.1 传播延时

电子信号必须遵从基本物理定律，逻辑值‘0’和‘1’通常在实际电路中用不同的电压值来表示。例如，TTL电路中常用+5V电压表示逻辑‘1’，用0V表示逻辑‘0’。由于电路中电容的存在，电路里节点的电压不能瞬间变化。于是，在实际电路中，门的输入变化与门的输出变化之间总是存在着一定的延时。VHDL语言中使用“<=”表示在传输延时之后信号发生的变化。

	Initial	t1	t1+2	t1+4	t1+6
X	1	4	5	5	3
Y	2	2	2	3	2
AS	2	2	8	10	15
Z	0	3	2	2	2
BS	2	2	5	10	12

	Initial	t1	t1+2	t1+4	t1+6
X	1	4	5	5	3
Y	2	2	2	3	2
AV	2	8	10	15	6
Z	0	3	2	2	2
BV	2	11	12	17	8

图4-1 比较瞬时赋值和延时赋值的结果

在较高级别的模型中，用传统方式描述的算法中仍需要使用类似C语言或C++语言中变量赋值的概念。本文中，赋予变量一个新值是瞬时发生的。VHDL语言使用符号“:=”来表示瞬时变量赋值。下面的例子说明了这两个概念：

```
(1)    AS <= X*Y after 2 ns;---- delayed signal assignment
(2)    BS <= AS+Z after 2 ns;--- delayed signal assignment
```

这两条语句是信号赋值语句，每个信号赋值的传播延时是2ns，瞬时变量赋值语句如下：

```
(3)    AV := X*Y;     ---- instantaneous variable assignment
(4)    BV := AV+Z;    ---- instantaneous variable assignment
```

尽管已经有简单明了的概念，传播延时仍会造成意想不到的结果。为了说明瞬时信号赋值与延时信号赋值的不同，假设语句1) 到语句4) 按书写顺序执行（参看图4-2）。再假设进程所包含的语句在时间t=t1和t1之后的2 ns时执行。注意到信号赋值语句的延时也是2 ns。同样，假设两种情况的初态完全一样，如图4-2左边的空格所示。现在考虑整数输入X、Y和Z的波形：

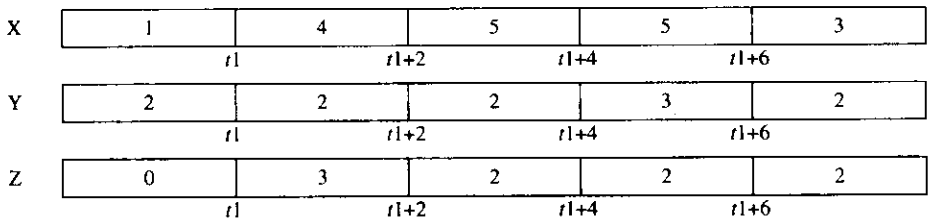


图4-1给出了瞬时信号赋值与延时信号赋值的结果。在t=t1时刻，AV和BV分别赋予新值8和11。由于在语句执行时变量的值被更新，AV在执行语句3) 时首先被赋值为8。语句4) 在计算BV时使用了AV的值，得出BV的值为11。BV在t=t1时刻瞬时得到新值。同样，在t=t1时刻，语句1) 为信号AS计算的新值也为8。但是，由于传播延时，AS的值到t=t1+2时才有效。于是当语句2) 在t=t1时刻被执行时，AS的值仍为2。所以，BS的值是5。类似地，BS的值要到t=t1+2时刻才有效。

在t=t1+2时刻，信号AV和BV的值被分别更新为8和5。然后，每条语句仍按书写顺序执行。各变量的新值可以瞬时得到，语句4) 在计算BV的值时使用了语句3) 所计算出的AV的值。语句1) 为AV计算出的新值为10，但AS的值要到t1+4时才有效。语句2) 使用AS的当前值8来计算BS的值，而BS仍然要到t1+4时才能得到新值。读者可以同样得出t1+4、t1+6时刻AV、BV、AS和BS的值。

由于传播延时，X的变化要在t1+4时刻才能作用到BS。而在瞬时传播中，X的变化在t1时刻马上可以作用到BV。更重要的是，这两个例子中，BV和BS的最终结果是不同的。可以明显看出，BS并不仅仅是BV延时后的值。传播延时改变了BS 相对于BV的真正值。读者应该仔细学习这个例子，从中理解模型中传播延时的重要作用。

虽然变量赋值语句适用于算法描述，但不能体现传播延时，所以在进程、函数和子过程中它的使用受到语法的限制。因此，变量的声明只能是在进程或子程序的内部。在结构体或块中，不允许在声明区域里出现变量声明。

所有的实体端口都对应于实际电路中的信号，所有的结构体都表示实体的行为，因此，所有的端口和结构体中声明的对象都表示一定的信号。内部信号可以在结构体或块中的声明区域内声明，而不是在进程或子程序中声明。信号赋值语句可以出现在任何地方。注意，图4-2满足上面讨论的所有限制。

在第3章曾经指出，进程中的语句总是按照书写顺序执行。按书写顺序执行的语句称为顺序语句。在进程PROP_DELAY中的信号赋值语句和进程INSTANEOUS中的变量赋值语句就是顺序语句。

进程INSTANEOUS中语句的顺序不可以改变。如果语句4) 在语句3) 之前执行，结果是不一样的。进程PROP_DELAY中的两个信号赋值语句，由于总是使用右边信号的当前值，而且左边信号的新值要在2 ns之后才有效，所以语句1) 和语句2) 可以按任意顺序执行而不改变结果。可以在不同的处理器上同时执行这两条语句以提高效率，条件是控制信号变化发生在2 ns

延时之后。读者可以改变它们的顺序来验证结果。

```
entity STATEMENTS is
  port(X,Y,Z: in INTEGER; -- Note: Entity ports are
       B: out INTEGER);   -- always signals.
end STATEMENTS;

architecture  PROP_DELAY of STATEMENTS is
  signal AS: INTEGER;
begin
 process (X,Y,Z)
 begin
   AS <= X*Y after 2 ns;    --Statement (1)
   B  <= AS+Z after 2 ns;   --Statement (2)
 end process;
end PROP_DEL;

architecture INSTANTANEOUS of STATEMENTS is
begin
 process(X,Y,Z)
   variable AV,BV: INTEGER;
 begin
   AV := X*Y;  --Statement (3)
   BV := AV+Z; --Statement (4)
   B <= BV;
 end process;
end INSTANTANEOUS;
```

图4-2　说明瞬时赋值和延时赋值的代码

可以同时执行的语句称为并发语句。因此，结构体内的语句1) 和语句2) 可以当作是并发语句。仿真时不需要放置在同一个进程内，两种情况的仿真结果是一样的。这种情形的出现是因为信号更新的延时。在下一节里可以看到VHDL仿真程序是如何使用延时来实现并发的。

4.1.2　延时和并发

由于逻辑信号的流向是并行的，VHDL模型逻辑电路的VHDL模型中必须包含并发执行支持机制。图4-3通过三个逻辑块说明这一概念。如果假设输入1和输入2同时有效，则逻辑块1和逻辑块2同时被激活。逻辑块3会在逻辑块1（Z1）或者逻辑块2（Z2）的输出有效时被激活。当信号传播通过逻辑块3时，新变化的信号又可以通过块1和块2传播。即信号流可以同时在这三个块中流动。

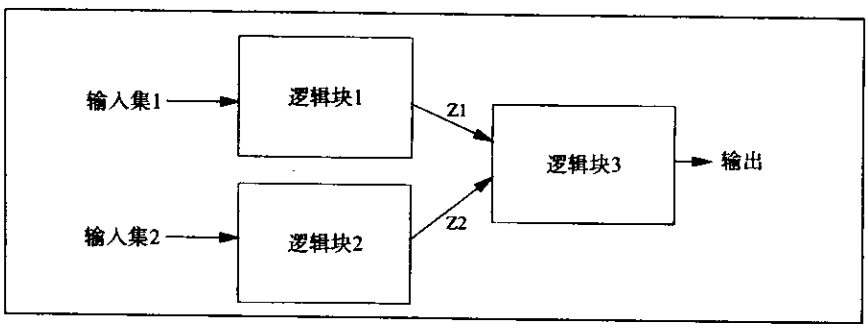


图4-3　把逻辑块映射到进程

```
LOGIC_BLOCK1: process(X1,X2,X3)
  variable YINT: BIT;
begin
  YINT := X1 and X2;
  Z1 <= YINT or X3 after 30ns;
end process LOGIC_BLOCK1;
```

图4-4 进程示例

硬件描述语言必须具有对这种同时性建模的机制。在VHDL语言中使用进程结构来完成这个功能。每个进程表示一个逻辑块，所有的进程并行执行（当然，如果仿真程序在单处理机上进行仿真，它们实际上是以一定顺序执行的，只不过从仿真的角度上可以把它们的执行看作是并发的）。在VHDL语言机制里，当一个进程敏感信号表（sensitivity list）中任一信号发生变化时，进程就会被执行。观察图4-3，假设我们知道每个块的功能，就可以为每个功能块建立VHDL进程。每个进程的敏感信号表一般包括了该逻辑块的输入信号集。具体一点说，逻辑块1可以用图4-4中的进程来表示。

注意 在图中，进程体首先包括一个声明部分，局部变量YINT在此声明。从关键字*begin*到*end*的执行部分包括变量赋值，用于计算变量（YINT）的中间结果，此后的信号赋值语句完成块中信号传播延时和函数值的计算。

进程也可以用来与信号赋值语句相互作用。考虑下面两个信号赋值语句：

```
AS <= X*Y;    ---- Statement S1
BS <= AS+Z;   ---- Statement S2
```

语句是按文字顺序书写的，所以也称按书写顺序执行。即顺序执行。

然而，这两个语句可以用另一种方式表示。S1和S2可以看作是并发的信号赋值语句。在这种情况下，它们相当于一个进程。S1的感应表包括X和Y，S2的感应表包括AS和Z。在这种解释下，语句S1在X或Y的值发生变化时执行。语句S2在AS或Z的值发生变化时执行。例如，并发信号赋值语句：

```
AS <= X * Y;   -- Concurrent signal assignment statement
```

等价于如下*process*语句：

```
process (X,Y)  -- Equivalent process statement
begin
  AS <= X * Y;
end process;
```

因为没有给出具体的传播时间，时间传播的缺省值为*delta*延时。它的值比0大，但比所有显式给出的特定时间值要小。

现在考虑S1和S2作为并发进程时的执行情况。假设X和Z都在t1时刻发生变化。则S1和S2都被激活。但是在这个例子中，S2使用的AS值仍为t1时刻AS的旧值，而不是语句S1计算出的AS的新值。这是因为S1在t1时刻计算的AS的新值要在t1+delta时才有效。当这个新值有效后，S2在t1+delta时刻被再次激活，根据AS的新值为BS，计算得出一个新值。

在图4-5里，使用前一节的例子来解释这个概念。X和Z的值都在t1时刻改变。由于t1时刻X发生变化，语句S1在t1时刻执行。它为信号AS计算出一个新值8，但AS的新值在t1+delta时

才会有效。同样，由于Z的变化也发生在t1时刻，语句S2也在t1时刻执行。这时由于AS的有效值仍为2，Z的值为3，所以它为BS计算得到的值为5。BS的新值在t1+delta时有效。

	Initial	t1	t1+delta	t1+2*delta
X	1	4	4	4
Y	2	2	2	2
AS	2	2	8	8
Z	0	3	3	3
BS	2	2	5	11

图4-5 并发信号赋值

在t1+delta时刻，AS和BS根据t1时刻的计算重新赋值（一个为8，另一个为5）。而AS的值在此刻发生了变化，语句S2在t1+delta时刻再次执行。它为BS计算出的新值为11，这个值在t1+2*delta时刻有效。X在t1时刻的变化在t1+delta时刻作用到了AS，在t1+2*delta时刻作用到了BS。因此，由于内部的delta延时，并发赋值与有限传播延时所引起的行为类似。

4.1.3 VHDL中的顺序语句和并发语句

通常人们无法从文本中推断它隐含的是顺序执行还是并发执行，而只能根据专门的注解或该语言的语法来判定执行类型。在VHDL语言中用语法来判别执行的类型。直接跟在结构体声明之后的语句可以被并发执行。而在进程或子程序内的语句被顺序执行。

图4-6表明了这一概念，实体STATEMENT的输入是X、Y和Z。在结构体CONCURRENT中，AS和BS声明为信号；AS是内部信号，BS是端口信号。这两个信号的赋值语句在结构体的代码段中。它们的书写顺序可以任意，而执行结果是一样的。

在结构SEQUENTIAL里，变量赋值语句S3和S4在有两个局部变量声明的进程里。因为这些语句在同一进程内，所以它们将按出现的顺序执行。这里S3和S4的顺序相当重要。计算的结果BV赋值到一个信号，所以可以在外部端口看见它的值。要注意，信号赋值语句既可以是顺序的，也可以是并发的。

顺序执行和并发执行的概念不仅仅适用于信号赋值语句，而且适用于其他一些情况。例如，因为进程语句紧跟在一个结构体内，所以它是并发语句。如果在同一个结构体里还有其他的进程；它们也不一定会按书写顺序执行，而是仅当敏感列表中的信号发生事件时才被执行。然而，与信号赋值语句不同的是，进程语句只能被并发执行。图3-18给出了一个完整的顺序语句和并发语句列表。

4.1.4 VHDL仿真程序中时间延时的实现

在VHDL中，时间被分为两种。考虑下面的两条信号赋值语句：

```
(1)     Y <= X;                 --delta delay
(2)     Y <= X after 10 ns;     --standard time unit delay.
```

在语句(1) 里，在delta延时后Y被赋值为X。Delta延时是一段比0大又比任何标准时间单位小的时间段。Y的值不是马上被更新，而是在delta延时后才发生变化。在当前时刻使用Y值的其他语句，即便它是在语句(1) 之后，使用的仍是Y的旧值。这是一个非常重要的概念，也是VHDL语言的核心，而对VHDL的新使用者来说又是常常会产生误解的地方。

```
entity STATEMENTS is
  port(X,Y,Z: in INTEGER; -- Note: entity ports are
       BS: out INTEGER);  -- always signals.
end STATEMENTS;

architecture CONCURRENT of STATEMENTS is
  signal AS: INTEGER;
begin
  AS <= X*Y;    --Statement S1
  BS <= AS+Z;   --Statement S2
end PROP_DEL;

architecture SEQUENTIAL of STATEMENTS is
begin
 process(X,Y,Z)
   variable AV,BV: INTEGER;
 begin
   AV := X*Y;  --Statement S3
   BV := AV+Z; --Statement S4
   BS <= BV;
 end process;
end INSTANTANEOUS;
```

图4-6 并发语句和顺序语句的VHDL代码

语句(2) 里使用的是一个标准的时间延时单位。例如，Y在10 ns后被赋值为X。在仿真期间，标准时间单位的流逝被称做仿真时间。注意到由于delta延时的定义，即使是delta延时的累计和也不会引起仿真时间的推进。

VHDL仿真程序使用这种时间延时机制来表明仿真周期的作用。在第2章，我们已经介绍了一种简单的标准时间单位累加的仿真周期。在这里，给出一个既可以允许delta延时又可以允许标准时间单位延时的仿真周期。一个周期内的处理步骤如下：

1) 如果时间队列为空，则停止；否则，把仿真时间推进到队列的下一个入口，执行步骤2。

2) 不推进仿真时间，而是开始一个新的仿真周期。将时间队列中当前仿真周期发生事件的表项删除，更新受到这些表项影响的所有信号值。激活被更新的信号所触发的所有进程。

3) 执行被激活的进程（没有执行顺序的限制）并重新安排（可能的）新的时间队列。某些时间队列表项可能包括delta延时。

4) 如果步骤3的调度中有由含delta延时引发的新的信号事件，转到步骤2；否则，进入步骤1。

注意 在同一仿真时间上会有多个仿真周期。这些仿真周期执行在同一个仿真时间的不同的delta时刻。因此，delta延时的精确含义为在没有推进仿真时间的情况下跳到下一个仿真周期。例如，语句(1) 被执行，Y的值在下一个仿真周期才有效。但是，如果同一个仿真时刻有需要使用Y值的其他语句，即使它在语句(1) 之后执行，它所使用的也只能是Y的旧值。

在语句(2) 里，Y的新值在当前时刻的10 ns后有效。这种情况下，很明显在10 ns之前的语句只能使用Y的旧值。而在语句(1) 中表现得没有这么明显。两者的区别仅在于Y的新值被使用前的时间长短不同。对于语句(1)，Y的新值在delta时间后有效；而对于语句(2)，Y的新值在10 ns后有效。

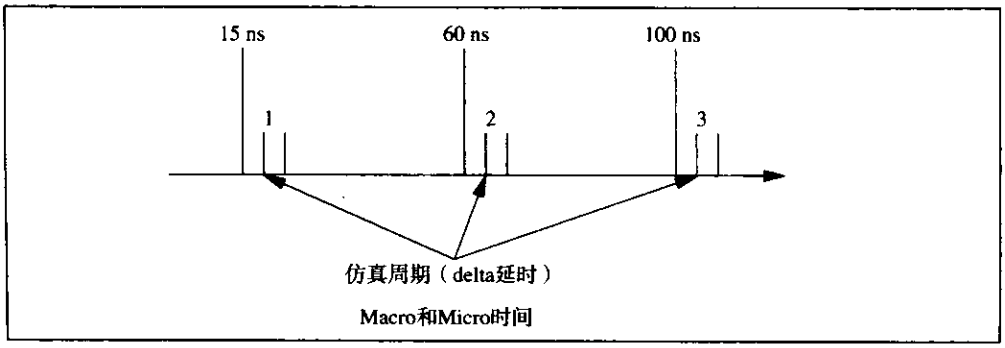


图4-7　微时间和宏时间

从另一个角度看delta周期，可以用它来表示微（*micro*）时间。标准时间单位表示为宏（*macro*）时间，macro时间可以包含任意多个micro时间。图4-7表明了仿真时的一种可能情况。在节点1，仿真时间推进了15 ns。在那个macro时间点上，有两个仿真周期，它们需要delta延时但不推进仿真时间。下一个仿真周期发生在macro时间为60的时刻。这种情况在点2（60 ns）和点3（100 ns）上重复。

这两种测量时间的方法构成了三种类型：

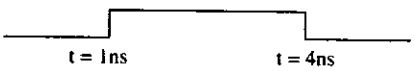
1) Delta延时：纯粹的模型功能验证。

2) 标准时间单位延时：用于验证系统定时的有效性。

3) 混合方法：这种模型包括以上两种延时。Delta延时用来形成延时不十分明显的事件的顺序。标准时间单位延时用于表示穿过逻辑块模型的明显的延时。混合模型也用于研究系统时钟。

为了说明这两种时钟模型的区别，考虑图4-8中实体BUFF的四个结构。在结构ONE里，输入X的值无延时地复制到变量Y1，Z在1 ns之后赋予X的值。整个结构ONE的传播延时为1 ns。在结构TWO，信号Y2在delta延时后赋值为X，输出值Z经过另一个delta延时赋予Y2的值。整个结构TWO的传播延时为2*delta。在结构THREE，Y3在delta延时后赋值为X，Z在1 ns延时后复制为变量Y3的值。整个结构中的延时是1 ns还是1 ns+ delta取决于是否把delta看作被1 ns所吸收，这个问题的考虑比较复杂。结构FOUR的所有延时为2 ns。

图4-9显示了每个结构对如下X脉冲的响应：



结构ONE、TWO、THREE、FOUR的输出分别为Z1、Z2、Z3和Z4。注意：1 delta的延时表示为+1；2 delta的延时表示为+2，并且时间推进和delta周期推进在同一个显示点显示。表述如下：

```
(ns)
 N       end of simulation cycle at time =  N ns.
 +1      end of simulation cycle at time =  N ns.
         plus 1 delta delay
 +2      end of simulation cycle at time =  N ns.
         plus 2 delta delays
 N+1     end of simulation cycle at time = N+1 ns.
```

对于进程中的信号赋值语句，要特别注意理解delta延时的意义。例如，考虑图4-10中实

体BUFF的结构体FIVE。如果X在t1时刻变化，它的值在t1+delta时刻复制到信号Y5，第二条语句使用t1时刻Y5的值，即在t1+delta时刻计算出的Z的值使用的是Y5的旧值。由于进程中的所有语句都是在同一时间执行，这两条信号赋值语句也是同时执行的。Y5的新值只有到X的另一事件发生时才会被复制到Z。图4-11表明了结构FIVE中信号Y5和Z5对应于脉冲X的响应。

```
entity BUFF is
  port(X: in BIT; Z: out BIT);
end BUFF;

architecture  ONE of BUFF is
begin
  process(X)
    variable Y1: BIT;
  begin
    Y1 := X;
    Z <= Y1 after 1 ns;
  end process;
end ONE;

architecture TWO of BUFF is
  signal Y2: BIT;
begin
  Y2 <= X ;
  Z <= Y2 ;
end TWO;

architecture THREE of BUFF is
  signal Y3: BIT;
begin
  Y3 <= X;
  Z <= Y3 after 1 ns;
end THREE;

architecture FOUR of BUFF is
  signal Y4: BIT;
begin
  Y4 <= X after 1 ns;
  Z <= Y4 after 1 ns;
end FOUR;
```

图4-8 四种不同的延时

TIME	SIGNAL NAMES							
(NS)	X	Z1	Y2	Z2	Y3	Z3	Y4	Z4
0	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
+1	---	---	'0'	'0'	'0'	---	---	---
1	'1'	'0'	---	---	---	'0'	'0'	'0'
+1	---	---	'1'	---	'1'	---	---	---
+2	---	---	---	'1'	---	---	---	---
2	---	'1'	---	---	---	'1'	'1'	---
3	---	---	---	---	---	---	---	'1'
4	'0'	---	---	---	---	---	---	---
+1	---	---	'0'	---	'0'	---	---	---
+2	---	---	---	'0'	---	---	---	---
5	---	'0'	---	---	---	'0'	'0'	---
6	---	---	---	---	---	---	---	'0'

图4-9 四个结构体的响应

```
architecture FIVE of BUFF is
  signal Y5: BIT;
begin
  process(X)
  begin
    Y5 <= X;
    Z  <= Y5;
  end process;
end FIVE;
architecture FIVE_A of BUFF is
  signal Y5: BIT;
begin
  process(X,Y5)
  begin
    Y5 <= X;
    Z  <= Y5;
  end process;
end FIVE_A;
```

图4-10 delta延时的作用

如果模型中Y5的值将在另一个delta延时后复制到Z，则Y5可以加入到结构FIVE_A的敏感表中。现在，Y5将在一个delta延时后变化，即在t1+delta时刻为Y5产生了一个新的事件。这一新事件在t1+delta时刻再次触发进程执行，于是Z将在t1+2*delta时刻更新为Y5的新值。从效果上看，在两个delta延时之后，Z将获得X的值。图4-11中的信号Y5A和Z5A表明了结构FIVE_A对脉冲X的响应。

Time	-------signal names------				
(ns)	X	Y5	Z5	Y5A	Z5A
0	'0'	'0'	'0'	'0'	'0'
+1	---	'0'	'0'	'0'	'0'
1	'1'	---	---	---	---
+1	---	'1'	'0'	'1'	'0'
+2	---	---	---	'1'	'1'
2	---	---	---	---	---
3	---	---	---	---	---
4	'0'	---	---	---	---
+1	---	'0'	'1'	'0'	'1'
+2	---	---	---	'0'	'0'
5	---	---	---	---	---
6	---	---	---	---	---

图4-11 两种结构体的响应

不用进程而使用结构体FIVE_B中的并发语句可以得到同样的结果。

```
architecture FIVE_B of BUFF is
  signal Y5: BIT;
begin
  Y5 <= X;
  Z  <= Y5;
end FIVE_B;
```

在此结构中，当X发生变化时，第一条语句被执行，使得Y5在一个delta延时后被赋予X的值。当Y5发生变化，第二条语句被执行，使得Z在两个delta延时之后获得X的值。

4.1.5 信号传播的惯性延时和传输延时

在VHDL语言里，信号赋值语句有两种类型的延时，一种是惯性延时，另一种是传输延时。下面是这两种延时的例子：

```
Z  <=  I  after 10 ns;           ----inertial delay
Z  <=  transport I after 10 ns; ----transport delay
```

第一个赋值语句表明是惯性延时，即信号传播发生的充要条件是输入I的电压保持10 ns不变——在*after*之后定义的时间。因此，I变化后的新值至少要保持10 ns才会对Z产生影响。第二个例子表明是传输延时，I的所有变化都会对Z产生影响，而不管I变化后的新值保持多久。

惯性延时机制过滤掉那些变化很快的输入，它模拟信号变化时电容的作用。在电路级，节点电压代表逻辑信号值。由于电容的存在，节点电压不可能瞬时变化。当电流增加或减少时，节点电压缓慢变化。如果该节点电压作为一个晶体管的驱动电压，则只有到达一个阈值电压后该晶体管才会发生变化。换句话说，电路具有惯性效应。节点电压的变化与晶体管状态的变化在时间上的滞后属于惯性延时。在VHDL中，惯性延时用一个不含关键字transport的信号赋值语句来表示。惯性延时在为实际电路建模时被普遍使用。由于传输延时对实际电路的表示并不精确，它通常应用于更高的抽象级上，例如，为测试而定义的模型输入。

4.2 VHDL调度算法

本节介绍惯性延时和传输延时的实现。为了讨论它们的实现，需要介绍一些概念。首先介绍的概念是事务（*transaction*）和波形（*waveform*）。图4-12阐明了这两个概念。信号Z由进程A和进程B两个进程所驱动。每个驱动Z的进程产生一个驱动信号，比如进程A中的DaZ和进程B中的DbZ。F的作用是由DaZ的值和DbZ的值计算Z的值。每次为驱动信号的赋值都表示一个事务的发生。事务在《*VHDL语言参考手册*》（Language Reference Manual，LRM）的定义如下：

事务：包括值和时间的对。值部分表示驱动器未来的值；时间部分表示值部分何时变成驱动器的当前值。

给出这个定义后，LRM定义了波形：

波形：一系列的事务。每个事务表示信号的驱动器的未来值。一个波形中的事务按时间排序，所以若一个事务的值先于另一个事务值，则波形中该事务也先于另一个事务出现。

图4-12是一系列的事务，如信号Z的驱动器DaZ的波形。最后，需要从LRM中引入更深入的概念：

驱动器的当前值

1) 存在一个时间部分不大于当前仿真时间的事务。

2) 驱动器的当前值为该事务的值部分。

3) 如果随仿真时间推进，当前仿真时间达到下一事务时间部分的值，则第一个事务从输出波形中删除，下一事务成为驱动器的当前值。

在图4-12中，驱动器DaZ的当前值用CV标记。图4-13表示仿真时间推进对驱动器当前值的作用。当simtime=15时，驱动器的当前值为最左的事务（10，0），它的时间部分为10。下一个事务为（22，1），它的时间部分为22。当simtime推进到22时，事务（22，1）决定了驱动信号（1）的当前值，事务（10，0）在波形中被删除。

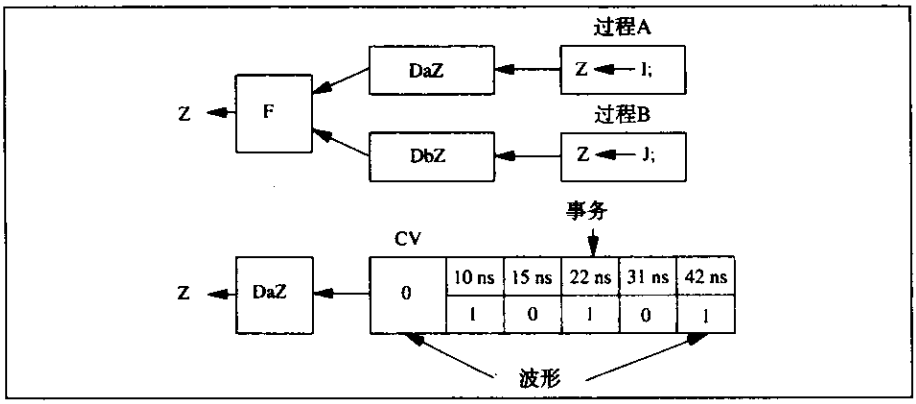


图4-12 事务和波形

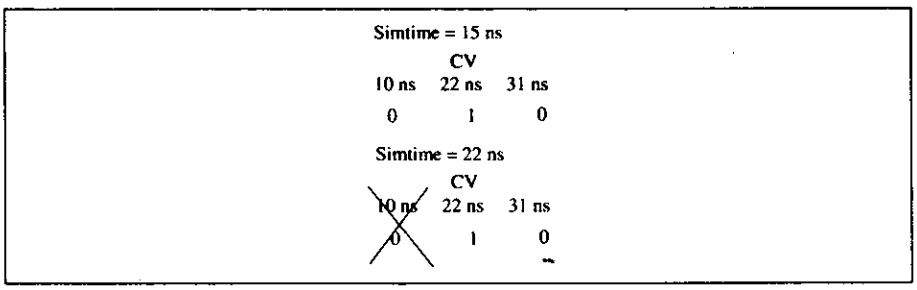


图4-13 驱动器的当前值

4.2.1 波形更新

当一条信号赋值语句执行后，该信号驱动器的波形被马上更新。信号赋值语句执行时该信号本身的值并不发生改变。只是该信号驱动器的值发生了变化。VHDL 语言参考手册（LRM）定义了如下的波形更新步骤。这个更新算法在每次信号赋值语句执行时被执行。如果在仿真周期期间同一个进程有多于一条信号赋值语句，则更新算法被多次执行。

波形更新算法

1) 从输出波形中删除在新事务最早发生时间之前的或同时发生的所有旧事务。

2) 新事务按事务发生的顺序附加到输出波形上。

如果对应的信号赋值语句中没有出现保留字transport，则波形中的第一个延时被看作惯性延时，输出波形做如下修改：

3) 标记所有新事务。

4) 在某个标记事务的前一个旧事务中，如果它的值部分与标记事务的值部分相同，则对它进行标记。

5) 决定驱动器当前值的事务被标记。

6) 将所有未标记事务（都是旧事务）从输出波形中删除。

图4-14a中，信号赋值语句的延时为10 ns。I的脉冲持续时间为5 ns。假设I的当前值为'0'。事件1将在t=0 ns时刻发生，I从'0'变为'1'。它导致事务（10 ns，1）附加到（1）行所表示的Z的波形上。驱动器的当前值和新事务都被作了标记，而且都被保留到波形上。事件2将在t=5

ns发生，使得I变低，并且将事务（15 ns，0）附加到（2）行所表示的波形上。如果是传输延时，所有的事务都保留在（2）行上。否则，标记进程将移出中间事务（10 ns，1），如（2）行的下一行所示。Z的波形的结果值为'0'，即I的输入脉冲被抑制了。

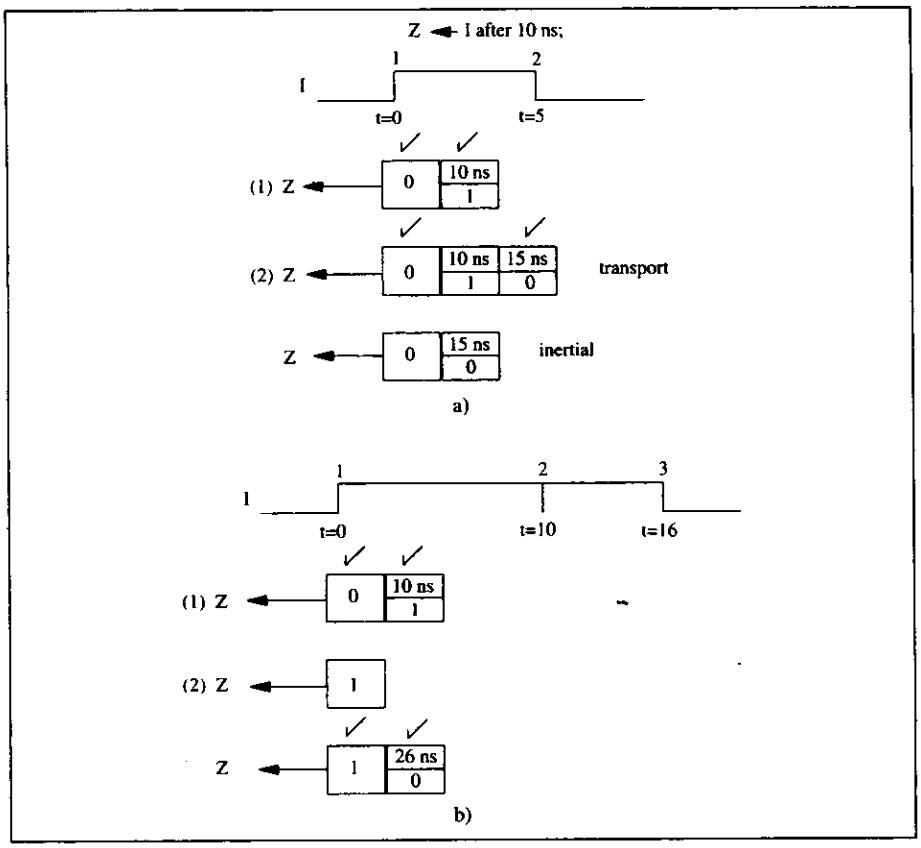


图4-14 波形更新

图4-14b中，I的脉冲持续时间为16 ns。事件1仍在t=0 ns时刻发生，使得I变高，事务（10 ns，1）附加到（1）行所表示的波形上。驱动器的当前值和新事务都被作了标记，而且都被保留到波形上。事件2将在t=10 ns发生，仿真时间随之推进，并且（2）行上驱动器的当前值变为'1'。在t=16时刻发生的事件3，使得I变低，事务（26 ns，0）附加到（2）行的下一行上。这两个事务都被标记并保留到波形上。由于仿真时间的推进，而且驱动器的当前值在新事务出现之前已经形成，I的输入脉冲传递到输出Z上。

4.2.2 副作用

尽管惯性延时机制对一般的逻辑建模非常有效，但它有时会产生不好的副作用。例如，在仿真初始使用如下的进程来调度信号的变化：

```
process
begin
  Z <= '1' after 50 ns;
  Z <= '0' after 100 ns;
  wait;
end process;
```

这一进程只在仿真最开始的时候执行一次，并且是顺序执行。假设Z的初始值为'0'，则惯性延时规则消除了由第一个信号赋值语句所定义的事务（50 ns，1）。为了解决这个问题，必须在语句中加入关键字transport：

```
process
begin
   Z <= transport '1' after 50 ns;
   Z <= transport '0' after 100 ns;
   wait;
end process;
```

或者为如下形式：

```
process
begin
   Z <=  '1' after 50 ns,
         '0' after 100 ns;
   wait;
end process;
```

最后一条语句的语义表示第一个赋值是惯性延时，第二个赋值是传输延时。所以，两个信号变化都被确定了时间。

4.3 组合逻辑和时序逻辑的建模

VHDL可以提供为组合逻辑和时序逻辑建模时所需的语言结构。图4-15a是一个组合逻辑电路，它的输出Z是输入X的函数，并且延时DEL后输出Z响应输入X的变化。如果用逻辑门的互连结构为其建模，它的结构图里没有环，即不存在反馈。该组合电路可由如下的VHDL进程描述：

```
COMBINATIONAL: process(X)
   --- declare process variables
   variable ZVAR:BIT;
begin
   ---  represent circuit functionality
   ---  compute ZVAR
   Z <= ZVAR after DEL; ---model circuit delay
end process;
```

为组合逻辑电路建模的步骤如下：

1) 当进程中信号X的敏感表发生变化时，进程将被执行，即敏感表的表项等价于电路的输入。

2) 该进程所表示的电路功能是计算变量ZVAR的值。

3) 电路延时用延时DEL后信号ZVAR的赋值表示。

所以，进程输出Z是输入X的一个函数，它经过了DEL延时，能有效地为组合电路建模。

时序逻辑有反馈，在图4-15b中同样由输入X驱动，但产生了两个输出：1) 电路的输出Z；2) 电路的状态变量Y，它作为反馈信号成为电路的附加输入。这种反馈机制致使电路可以表现出时序行为。下面所示的进程SEQUENTIAL是为时序行为建立的模型。状态变量存储在信号Y内，进程执行时Y被修改。同时，Y也被列入进程的敏感表中，当Y发生变化时进程被触发。这样，就构成了时序电路中的反馈模型。这种反馈显示出时序电路的记忆能力。由于所使用抽象级别的不同，有时候反馈机制很可能在模型中并没有显示出来。例如，为寄存器建立的行为模型，它与由互连门所组成的结构模型不同，行为模型中的反馈并不是那么明显。以后将会讨论在更高的抽象级上

为时序行为建模的机制。

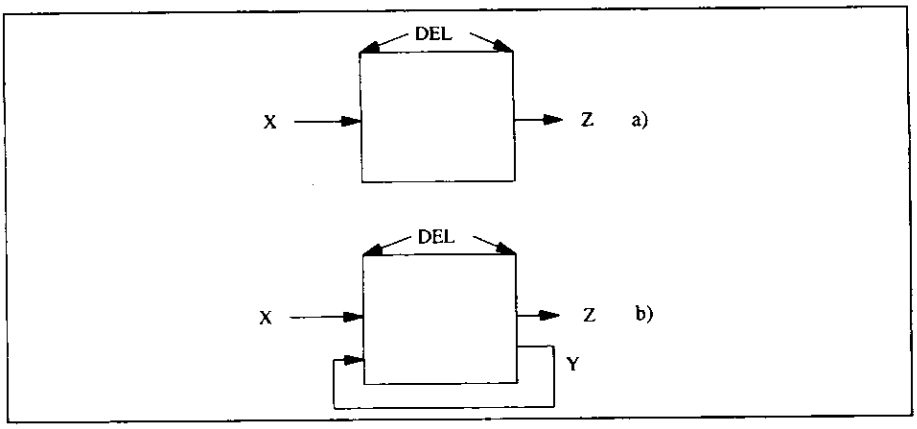


图4-15 基本的组合逻辑和时序逻辑

```
SEQUENTIAL: process(X,Y)
  --- declare process variables
  variable YVAR,ZVAR:BIT;
begin
  ---  represent circuit functionality
  ---  compute YVAR and ZVAR
  Y <= YVAR after DEL; ---state variable delay
  Z <= ZVAR after DEL; ---output delay
end process;
```

通过这个例子可以从中得出某些结论。图4-16a用VHDL描述了三输入电路的主要功能。其中，只用一条信号赋值语句就实现了电路行为，这条语句等价于一个进程。图4-16b的模型开环使用，它的行为是一个组合逻辑电路的行为。图4-16c的输出Z连接到了输入X（2）。这种反馈连接将模型变为时序电路，可以把它称做“一致（consensus）”元素，输出Z等于X（0）和X（1）合成的最终结果。

4.4 逻辑基本部件

表1-1列出了不同抽象级别上常用的结构域的基本部件。第9章将可以看到这些基本部件在ASIC库中的独特功能。本节会介绍一些基本部件（或称为原语）的VHDL模型。通过介绍这些模型，解释了为组合逻辑和时序逻辑建模时所使用的一些其他方法。图4-17给出了按组合逻辑和时序逻辑分类的基本部件集。本章中列出了它们当中的大多数模型。另外一些分散在文中的其他部分或者作为课后习题。我们只给出了这些基本部件的简单形式，使用通用的时间表示延时，实际上器件的时序模型也同样简单。模型里只允许使用类型Bit和Bit_Vector表示信号和数据。第5章里将使用到多值逻辑。第7章还会为模型引入更复杂的时钟。

本章在建模时使用两种基本方法——算法和数据流，它们的结构名分别为ALG和DF。

4.4.1 组合逻辑基本部件

现在介绍组合逻辑基本部件的模型。

1. 门

门是最基本的组合逻辑部件。图4-18是二输入AND门的模型。模型计算两个输入的

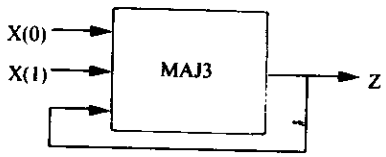
AND逻辑，并且经过通用延时DEL将结果赋值到输出O。其他门原语的模型作为读者的课后练习。

```
entity MAJ3 is
   generic(DEL: TIME);
   port(X: in BIT_VECTOR(0 to 2); Z: out BIT);
end MAJ3;

architecture DATAFLOW of MAJ3 is
begin
   Z <= (X(0) and X(1)) or (X(1) and X(2))
     or (X(0) and X(2)) after DEL;
```

a) VHDL描述

b) 组合电路



c) 时序电路

图4-16 一个主体/一致元素

组合电路基本部件	时序电路基本部件
门	触发器
缓冲器	寄存器
加法器	锁存
乘法器	计数器
解码器	RAMs
译码器	
比较器	
移位器	
算术逻辑部件	
通用计数器	
ROMs	
PLAs	

图4-17 设计基本部件

2. 缓冲器

当使能输入E为‘1’时，缓冲器的输入值I复制到输出值O。如果E的输入是‘0’，缓冲器的输出O为一个高阻状态。图4-19是缓冲器的模型。在这个例子中，输出的高阻状态用逻辑‘1’表示。因此，这个缓冲器的输出可以直接通过一个线与总线与另一个缓冲器的输出相连。

对于很多种情况来说，这是一种既足够精确的模型。注意，对缓冲器输出使能的时机使用了不同的传播延时来协调。在第5章采用了多值逻辑来更精确地为总线建模。

```
entity AND2 is
  generic(DEL: TIME);
  port(I1,I2: in BIT; O: out BIT);
end AND2;
architecture DF of AND2 is
begin
  O <= I1 and I2 after DEL;
end DF;
```

图4-18 AND门原语

```
entity BUF is
  generic(DATA_DEL,Z_DEL: TIME);
  port(I,EN: in BIT; O: out BIT);
end BUF;
architecture ALG of BUF is
begin
  process(I,EN)
  begin
    if EN = '1' then
      O <= I after DATA_DEL;
    else
      O <=  '1' after Z_DEL;
    end if;
  end process;
end ALG;
```

图4-19 缓冲器原语

3. 加法器

加法器在逻辑电路中非常重要。加法器中更基本的原语是全加器。全加器对两个数据位和一个进位相加，计算出结果及下一个进位。图4-20是一个全加器的数据流模型。注意：和的输出是3个输入数据的奇数函数。下一个进位的产生由3个输入位的多数函数所决定。这两个功能在模型中有着不同的延时。

```
entity FULL_ADDER is
  generic(SUM_DEL,CARRY_DEL:TIME);
  port(A,B,CI: in BIT; SUM,COUT: out BIT);
end FULL_ADDER;

architecture DF of FULL_ADDER is
begin
  SUM <= A xor B xor CI after SUM_DEL;
  COUT <= (A and B) or (A and CI) or (B and CI)
          after CARRY_DEL;
end DF;
```

图4-20 全加器基本部件

多位加法器由全加器层叠构成。图4-21是用全加器层叠所实现的4位加法器。

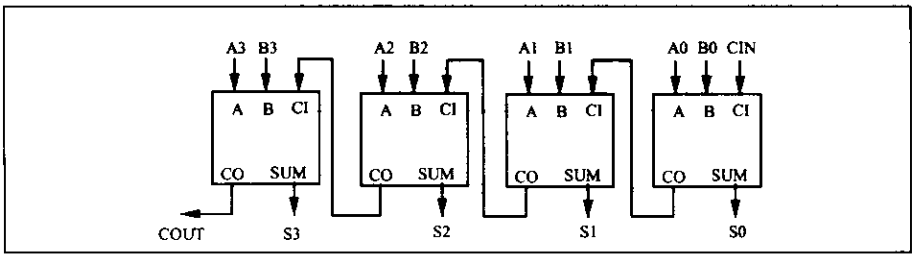


图4-21 4位加发器

4. 多路器

该部件是对一定数量的数据源进行选择时所必需的元件。通常数据源的个数是2^N，我们将这2^N选1的多路器。同时，把长度为B位的向量复用，B的范围从1到W，W是系统实现时数据字的大小。图4-22是4个数据源的4位多路器。注意这个模型中使用了一个条件赋值语句。

```
entity FOUR_TO_1_MUX is
  generic(DEL: TIME);
  port(IN0,IN1,IN2,IN3: in BIT_VECTOR(3 downto 0);
       SEL: in BIT_VECTOR(1 downto 0);
       O: out BIT_VECTOR(3 downto 0));
end FOUR_TO_1_MUX;

architecture DF of FOUR_TO_1_MUX is
begin
  O <= IN0 after DEL when SEL = "00" else
       IN1 after DEL when SEL = "01" else
       IN2 after DEL when SEL = "10" else
       IN3 after DEL;
end DF;
```

图4-22 多路器原语

5. 解码器

解码器将N位的输入变为2^N位的输出。它的输出i等价于N位输入代码表示的十进制值。从这个观点来看，解码器的功能实际上是完成二进制和十进制的转换。解码器的对位模式的转换对应于地址转换为单线选择信号。它也可以用来解释计算机指令操作码的位模式，以选择执行相应指令。图4-23给出了一个2到4的解码器模型。

6. 编码器

编码器的功能与解码器的功能相反，将2^N位的输入变为N位的输出。N位码表示最高的输入优先级。因此，为输入建立优先级别是必要的，例如，输入0具有最高的优先级，2^N-1具有最低的优先级。图4-24给出了4位到2位的编码器。同时，模型中的条件信号赋值语句可以精确实现优先级。注意，如果I（3）是唯一活跃的输入，或根本没有输入，解码器的输出将为“11”。

7. 移位器

移位操作是位向量的一种重要操作。在一些特定的情况下，向右移动一位表示被2除；向左移动一位表示乘以2。从纯粹的逻辑观点来看，移位对应着一种位模式的改变。图4-25是用

VHDL实现的移位器，SR='1'且SL='0'时向右移动一位，SR='0'且SL='1'时向左移动一位。当SR和SL取其他的值的组合时，输入直接传到输出而不发生移位。相应的IL和IR用来表示向左向右移动后的值。注意，为移位操作建模时使用了串联操作。

```
entity TWO_TO_4_DEC is
  generic(DEL: TIME);
  port(I: in  BIT_VECTOR(1 downto 0);
       O: out BIT_VECTOR(3 downto 0));
end TWO_TO_4_DEC;

architecture ALG of TWO_TO_4_DEC is
begin
  process(I)
  begin
    case I is
      when "00" => O<= "0001" after DEL;
      when "01" => O<= "0010" after DEL;
      when "10" => O<= "0100" after DEL;
      when "11" => O<= "1000" after DEL;
    end case;
  end process;
end ALG;
```

图4-23 解码器原语

```
entity FOUR_TO_2_ENC is
  generic(DEL: TIME);
  port(I: in  BIT_VECTOR(3 downto 0);
       O: out BIT_VECTOR(1 downto 0));
  end FOUR_TO_2_ENC;

architecture DF of FOUR_TO_2_ENC is
begin
  O <= "00" after DEL when I(0) = '1' else
       "01" after DEL when I(1) = '1' else
       "10" after DEL when I(2) = '1' else
       "11" after DEL;
end DF;
```

图4-24 编码器原语

8. 数据操作包

为了实现余下的基本部件模型，需要定义一些基本的函数和子函数。图4-26是包括了这些函数的包。所有的函数都针对不限制长度的位向量进行操作。每种操作都假设这些位向量代表的是一个二进制数，最低的数据位在最右端。注意：每种情况都产生一个内部变量，该变量具有降序的范围，它用于存储参数值。因此，由于最低的数据位在右，调用的参数既可以是升序又可以是降序。对这些特定函数或子函数的进一步说明如下：

1) ADD。用for循环语句实现全加器的逻辑等式。循环体的个数等于加数或被加数的长度。这里假设所有的操作数有相同的长度。

2) INC和DEC。也是使用一个for语句，INC（DEC）的功能是从最低位起，将所有为1(0)的取反，直到出现第一个0（1）。出现的这个0（1）也同样取反，剩下的位不变。

```
entity SHIFTER is
  generic(DEL: TIME);
  port(DATA_IN: in BIT_VECTOR(3 downto 0);
       SR,SL: in BIT; IL,IR: in BIT;
       DATA_OUT: out BIT_VECTOR(3 downto 0));
end SHIFTER;

architecture  ALG of SHIFTER is
begin
  process(SR,SL,DATA_IN,IL,IR)
    variable CON: BIT_VECTOR(0 to 1);
  begin
    CON := SR&SL;
    case CON is
      when "00" => DATA_OUT <= DATA_IN after DEL;
      when "01" => DATA_OUT <= DATA_IN(2 downto 0) & IL
                   after DEL;
      when "10" => DATA_OUT <= IR & DATA_IN(3 downto 1)
                   after DEL;
      when "11" => DATA_OUT <= DATA_IN after DEL;
    end case;
  end process;
end ALG;
```

图4-25　移位器原语

3) INTVAL。使用一条for循环语句，对不同幂次的二进制位求和，它可以将一个位向量变为一个整数。

9. ALU

算术逻辑单元（ALU）是在计算机系统中进行算术操作和逻辑操作的组合电路。图4-27是一个简单的ALU。它有3个数据输入端：A和B是位向量，CI是进位。有两个数据输出：F是一个位向量的输出，COUT是输出进位。功能选择输入FSEL控制ALU的功能。这个例子中，FSEL是一个2位向量，ALU有4种功能：

1) F = A。

2) F = not（A）。

3) F = A+B，+表示两个操作数的加。和中包括了进位CI，同时产生一个输出操作。

4) F = A and B。

这个简单的ALU具有基本的逻辑和算术运算能力。更复杂的操作可以通过ALU重复传递数据和用寄存器存储中间结果来实现。在ALU设计中有几个有意思的问题：1) 操作集的功能是否完备，例如，是否任意一个逻辑或算术功能都可以通过ALU的反复应用来实现？2) 逻辑功能的运算能力和执行速度，以及所需的基本部件（如寄存器和乘法器）的个数之间如何进行折衷？这些问题留作本章的作业。最后，简单的ALU经常层叠组成计算更长字节的ALU。图4-28给出了由4位ALU组成的16位ALU。注意，控制位C0和C1来自功能选择输入。

10. PLA

通常的VLSI技术中，组合逻辑常用一个可编程逻辑阵列（PLA）来实现。图4-29是一个含4种功能（Z1，Z2，Z3，Z4）以及3个变量（X1，X2，X3）的PLA。尽管阵列可以实现4输入变量不同的逻辑函数，4个输出函数将产生的所有结果限制在4个。这与3位地址位4位字宽的ROM相比较的话。ROM可以实现具有相同3输入所有4种函数。通常在同样的输入个数和输

出个数情况下，PLA的开销要小于ROM。但是PLA的灵活性要差一些。

```
package PRIMS is
  procedure ADD(A,B: in BIT_VECTOR; CIN: in BIT;
                SUM: out BIT_VECTOR; COUT: out BIT);
  function INC(X : BIT_VECTOR) return BIT_VECTOR;
  function DEC(X : BIT_VECTOR) return BIT_VECTOR;
  function INTVAL(VAL : BIT_VECTOR) return INTEGER;
end PRIMS;
package body PRIMS is
  procedure ADD(A,B: in BIT_VECTOR; CIN: in BIT;
                SUM: out BIT_VECTOR; COUT: out BIT) is
    variable SUMV,AV,BV: BIT_VECTOR(A'LENGTH-1 downto 0);
    variable CARRY: BIT;
  begin
    AV := A;
    BV := B;
    CARRY := CIN;
    for I in 0 to SUMV'HIGH loop
      SUMV(I) := AV(I) xor BV(I) xor CARRY;
      CARRY := (AV(I) and BV(I)) or (AV(I) and CARRY)
               or (BV(i) and CARRY);
    end loop;
    COUT := CARRY;
    SUM := SUMV;
  end ADD;

  function INC(X : BIT_VECTOR) return BIT_VECTOR is
    variable XV: BIT_VECTOR(X'LENGTH-1 downto 0);
  begin
    XV := X;
    for I in 0 to XV'HIGH loop
      if XV(I) = '0' then
        XV(I) := '1';
        exit;
      else XV(I) := '0';
      end if;
    end loop;
    return XV;
  end INC;
function DEC(X : BIT_VECTOR) return BIT_VECTOR is
    variable XV: BIT_VECTOR(X'LENGTH-1 downto 0);
  begin
    XV := X;
    for I in 0 to XV'HIGH loop
      if XV(I) = '1' then
        XV(I) := '0';
        exit;
      else XV(I) := '1';
      end if;
    end loop;
    return XV;
  end DEC;

  function INTVAL ( VAL: BIT_VECTOR) return INTEGER is
    variable VALV: BIT_VECTOR(VAL'LENGTH - 1 downto 0);
    variable SUM: INTEGER := 0;
  begin
    VALV := VAL;
    for N in VALV'LOW to VALV'HIGH loop
      if VALV(N) = '1' then
        SUM := SUM + (2**N);
      end if;
    end loop;
    return SUM;
  end INTVAL;
end PRIMS;
```

图4-26 数据操作包

图4-29中的PLA分为2个平面，AND和OR平面。AND平面通过对所有的位进行“与”操作产生结果。OR平面对所有的位进行“或”操作生成结果。PLA的基本电路实现是NOR-NOR逻辑。通过在输出加入反相器产生NOR-OR逻辑，在此基础上又可以产生AND-OR逻辑。根据De Morgan定律，NOR可以变为AND逻辑。

```
use work.PRIMS.all;
entity ALU is
  generic(DEL: TIME);
  port(A,B: in BIT_VECTOR(3 downto 0); CI: in BIT;
       FSEL: in BIT_VECTOR(1 downto 0);
       F: out BIT_VECTOR(3 downto 0); COUT: out BIT);
end ALU;
architecture ALG of ALU is
begin
  process(A,B,CI,FSEL)
    variable  FV: BIT_VECTOR(3 downto 0);
    variable COUTV: BIT;
  begin
    case FSEL is
      when "00" => F <= A after DEL;
      when "01" => F <= not(A) after DEL;
      when "10" => ADD(A,B,CI,FV,COUTV);
                   F <= FV after DEL;
                   COUT <= COUTV after DEL;
      when "11" => F <= A and B after DEL;
    end case;
  end process;
end ALG;
```

图4-27　ALU基本部件

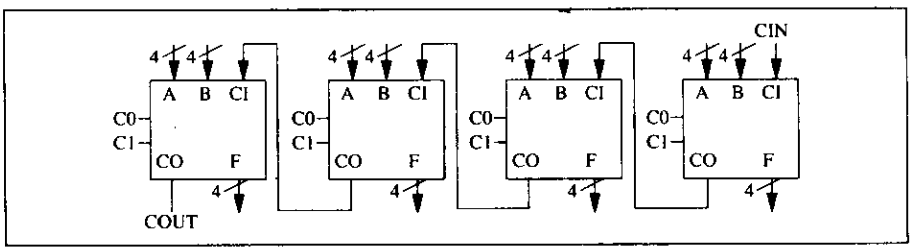


图4-28　16位ALU

$$\bar{A} \text{ and } B \text{ and } \bar{C} = \text{not}(A \text{ or } \bar{B} \text{ or } C)$$

为了产生如$\bar{A}B\bar{C}$的项，必须把输入A、$\bar{B}$和C连接到AND门的平面。水平的每一条线，如R1，都对应其产生的项。AND平面的输出直接连在OR平面的输入上，它用来产生逻辑OR结果项的一个子集。通过在合适的电路连线上加入晶体管开关来建立期望的连接。这种特定的PLA编程可以通过加入开关实现以下逻辑功能。

$$R1 = X1$$
$$R2 = \overline{X2}\,\overline{X3}$$
$$R3 = \overline{X1}\,\overline{X2}X3$$
$$R4 = \overline{X1}X2\overline{X3}$$

$$Z1 = R1 = X1$$
$$Z2 = R1 + R3 = X1 + (\overline{X1}\,\overline{X2}X3)$$
$$Z3 = R2 = \overline{X2}\,\overline{X3}$$
$$Z4 = R3 + R4 = (\overline{X1}\,\overline{X2}X3) + (\overline{X1}X2\overline{X3})$$

因为X2和X3作为第一级NOR门的输入，所以R2对应项$\overline{X2}\ \overline{X3}$。总之，如果变量X在一个乘积项中，则把$\bar{X}$连接到开关；如果用到$\bar{X}$，则把X连到开关上。读者可以验证其他的乘积项。

图4-29的PLA中，AND平面有24个可能的连接点，OR平面有16个。

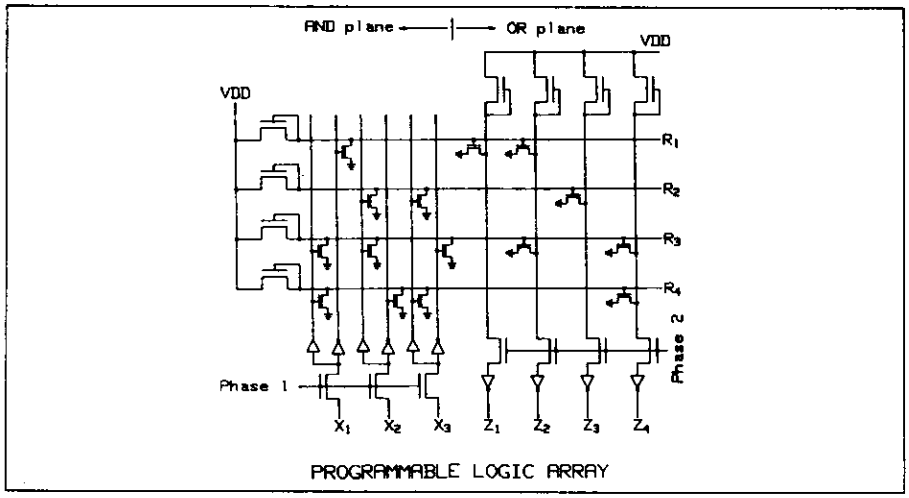


图4-29 可编程逻辑阵列

可以在开关级为PLA建模，但是它需要使用第7章讨论的多值逻辑系统，而且这一级模型的细节程度对于大多数应用来说都是不必要的。因此我们产生行为域模型，它通过扫描两个连接矩阵及AND和OR平面的输入值来计算PLA的输出。一个连接矩阵是1和0的数组，其中1代表开关连接，0代表开关断开。图4-30给出了PLA的行为模型。该模型被分为两个进程：AND_PLANE和OR_PLANE。信号(R)把 AND平面的输出和OR平面的输入连接在一起。

在进程AND_PLANE中，连接矩阵存放在一个三维数组中。第一维输入选择AND平面计算（Ri）的函数，第二位输入选择输入文字（Xj），第三维输入表示开关输入是TRUE还是FALSE（不能同时既是TRUE又是FALSE）。

在这个进程里，用两级嵌套循环对这个矩阵进行扫描。外部的循环索引扫描变量i；内部的循环索引扫描变量j。如果AND_PL(I,J,1)为‘1’，第I个乘积项中的变量J为“正”文字，则对于第I个乘积项的等式需要用到$\bar{J}$。如果AND_PL(I,J,2)为‘1’，第I个乘积项中的变量J为“反”文字，则对于第I个乘积项的等式需要用到J。注意：不能同时包含“正”文字和“反”文字；有一个断言可检测这一点。循环结束之后，结果通过NOR函数取反。注意由于NOR操作不可以关联，所以它不能在循环中直接使用。

在OR_PLANE进程中，连接矩阵是一个二维数组。当且仅当乘积项J包含在输出函数I中时，OR_PLANE(I,J)=‘1’。OR函数也使用两级嵌套循环来实现。从AND平面到OR平面的延时用两个类属延时AND_DEL和OR_DEL来建模。

图4-17的组合逻辑电路列中还剩下两个基本部件，通用计数器和比较器。在第3章已经讲述了如何用‘1’计数器实现一个通用计数器。为比较器建模将作为本章的课后作业。

4.4.2 时序逻辑基本部件

下面介绍时序逻辑基本部件的建模。

1. 触发器

触发器是最基本的时序逻辑部件。图4-31是一个JK触发器的算法模型。注意，直接置位

或重新置位会破坏触发器的时序行为。如果R=S=1，不会产生任何操作，这是一种不合法的情况，输出通常是不确定的。因此，不作任何操作的情况与规范一致。在第7章将讨论其他的针对一些意外情况的处理方法。

```
entity PLA is
  generic(AND_DEL,OR_DEL: TIME);
  port(X: in BIT_VECTOR(1 to 3);
       Z: out BIT_VECTOR(1 to 4));
end PLA;
architecture CONNECTION_MATRIX of PLA is
  signal R: BIT_VECTOR(1 to 4);
begin
 AND_PLANE: process(X)
   variable RV: BIT_VECTOR(1 to 4);
   type AND_ARRAY is array( 1 to 4, 1 to 3, 1 to 2 ) of BIT;
   variable AND_PL: AND_ARRAY :=
     ((('0','1'),('0','0'),('0','0')),
      (('0','0'),('1','0'),('1','0')),
      (('1','0'),('1','0'),('0','1')),
      (('1','0'),('0','1'),('1','0')));
 begin
   for I in 1 to 4 loop
     RV(I) := '0';
     for J in 1 to 3 loop
     assert not(AND_PL(I,J,1) = '1' and AND_PL(I,J,2) = '1')
       report "Error in AND plane wiring";
       if AND_PL(I,J,1) = '1' then
         RV(I) := RV(I) or X(J);
       end if;
       if AND_PL(I,J,2) = '1' then RV(I) := RV(I) or not X(J);
       end if;
     end loop;
     R(I) <= not RV(I) after AND_DEL;
   end loop;
 end process AND_PLANE;
 OR_PLANE: process(R)
   variable ZV: BIT_VECTOR(1 to 4);
   type OR_ARRAY is array(1 to 4,1 to 4) of BIT;
   variable OR_PLANE: OR_ARRAY :=
     (('1','0','0','0'),('1','0','1','0'),
      ('0','1','0','0'),('0','0','1','1'));
 begin
   for I in 1 to 4 loop
     ZV(I) := '0';
     for J in 1 to 4 loop
       if OR_PLANE(I,J) = '1' then ZV(I) := ZV(I) or R(J);
       end if;
     end loop;
     Z(I) <= ZV(I) after OR_DEL;
   end loop;
 end process OR_PLANE;
end CONNECTION_MATRIX;
```

图4-30　PLA模型

其他基本触发器的模型都非常简单。

2. 寄存器

```
entity JKFF is
  generic(SRDEL,CLKDEL: TIME);
  port(S,R,J,K,CLK: in BIT; Q,QN: inout BIT);
end JKFF;

architecture ALG of JKFF is
begin
  process(CLK,S,R)
  begin
    if S = '1' and R = '0' then
      Q <= '1' after SRDEL;
      QN <= '0' after SRDEL;
    elsif S = '0' and R = '1' then
      Q  <= '0' after SRDEL;
      QN <= '1' after SRDEL;
    elsif CLK'EVENT and CLK = '1' and S='0' and R='0' then
      if J = '1' and K = '0' then
        Q <= '1' after CLKDEL;
        QN <= '0' after CLKDEL;
      elsif J = '0' and K ='1'  then
        Q <= '0' after CLKDEL;
        QN <= '1' after CLKDEL;
      elsif J= '1' and K= '1'  then
        Q <= not Q after CLKDEL;
        QN <= not QN after CLKDEL;
      end if;
    end if;
  end process;
end ALG;
```

图4-31 JK触发器模型

使用寄存器是为了存储数据。尽管寄存器的结构模型可以用触发器和门来建立，在这里我们也定义了寄存器的行为模型。图4-32是一个寄存器模型。注意：寄存器是完全同步的，即复位和载入功能都是同步的，但复位优先执行。其他模型中的复位则可能与时钟异步。寄存器使用全面的同步定时可以避免定时错误。注意：寄存器是受控的与时钟即只有在LOAD='1'并且是时钟上升沿时才会写入寄存器。同样注意，这里要使用一个防护块（在下一节讨论）。有了控制信号（LOAD）的寄存器可以更容易被控制单元发出的控制信号所控制。控制单元在第6章和第8章详细讨论。

```
entity REG is
  generic(DEL: TIME);
  port(RESET,LOAD,CLK: in BIT;
       DATA_IN: in BIT_VECTOR(3 downto 0);
       Q: inout BIT_VECTOR(3 downto 0));
end REG;

architecture DF of REG is
begin
  REG: block(not CLK'STABLE and CLK ='1')
  begin
    Q <= guarded "0000" after DEL when RESET ='1' else
         DATA_IN after DEL when LOAD ='1' else
         Q;
  end block REG;
end DF;
```

图4-32 寄存器模型

3. 锁存

上面的有时序作用的寄存器模型在时钟正的跳变时对它的输入进行采样并立即更新输出。显然，同样可以定义在时钟下降沿触发的寄存器。锁存是类似寄存器的另一种具有存储功能的元件，当时钟为高电平时它的输出跟随输入，而当时钟变低时保存输入值。因此，当时钟为高时，锁存相当于一个逻辑缓冲，在下一个时钟，它的行为则表现为一个记忆元件。因为锁存的输出可以发生得更早，所以与边缘触发的记忆元件相比，它减少了传输延时。

图4-33是一个锁存元件的VHDL描述。当块LATCH的防护（CLK）变为‘1’时，D的值在LATCH_DEL之后变为LOUT，在CLK保持为‘1’的情况下，D的任何后续变化都被传播到LOUT。而在CLK为低电平时，防护条件无效，信号LOUT保持为在CLK=‘1’时最后写入的值。图4-34为锁存器的定时关系，LATCH_DEL=10 ns，输入D从全0变为全1。

需要指出的是：其他一些书或资料的作者会把寄存器和锁存这两个名词混为一谈。而在我们看来，这两个名词有着不同的行为，因此，按此处的定义使用。

```
LATCH is
  generic(LATCH_DEL:TIME);
  port(D: in BIT_VECTOR(7 downto 0);
       CLK: in BIT; LOUT: out BIT_VECTOR(7 downto 0));
end LATCH;

architecture DFLOW of LATCH is
begin
  LATCH: block(CLK = '1')
  begin
    LOUT <= guarded D after LATCH_DEL;
  end block LATCH;
end DFLOW;
```

图4-33　锁存原语

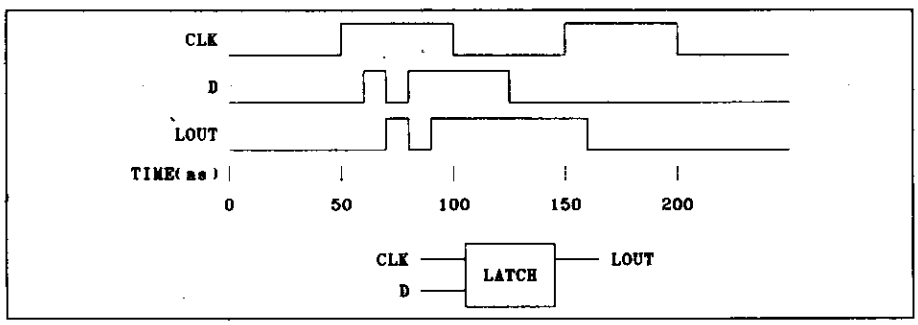


图4-34　锁存时序

4. 移位寄存器

移位寄存器对数据操作非常重要。图4-35是这种基本部件的模型。它既可以载入（LOAD=‘1’），又可以右移（SR=‘1’，且SL=‘0’）和左移（SR=‘0’，且SL=‘1’）。IR和IL分别是右移和左移的串行输入。载入优先于移位操作。

5. 计数器基本部件

计数是数字系统中的基本操作。图4-36描述了一个既可以复位（RESET=‘1’），载入

(LOAD= '1'), 又可以由时钟上升沿用UP信号控制增加或减少的计数器。优先级顺序是1)重置; 2) 载入; 3) 计数。计数使用了PACKAGE PRIMS中的INC和DEC函数。

```
entity SHIFTREG is
  generic(DEL: TIME);
  port(DATA_IN: in BIT_VECTOR(3 downto 0);
       CLK,LOAD,SR,SL: in BIT; IL,IR: in BIT;
       Q: inout BIT_VECTOR(3 downto 0));
end SHIFTREG;

architecture  DF of SHIFTREG is
begin
  SH:block(not CLK'STABLE and CLK ='1')
  begin
    Q <= guarded DATA_IN after DEL when LOAD= '1' else
    Q(2 downto 0) & IL after DEL when SL='1' and SR='0' else
    IR & Q(3 downto 1) after DEL when SL='0' and SR='1' else
    Q;
  end block SH;
end DF;
```

图4-35 移位寄存器原语

```
entity COUNTER is
  generic(DEL:TIME);
  port(RESET,LOAD,COUNT,UP,CLK: in BIT;
       DATA_IN: in BIT_VECTOR(3 downto 0);
       CNT: inout BIT_VECTOR(3 downto 0));
end COUNTER;
use work.PRIMS.all;
architecture ALG of COUNTER is
begin
  process(CLK)
  begin
    if CLK = '1' then
      if RESET = '1' then
        CNT <= "0000" after DEL;
      elsif LOAD ='1' then
        CNT <= DATA_IN after DEL;
      elsif COUNT ='1' then
        if UP = '1' then
          CNT <= INC(CNT) after DEL;
        else
          CNT <= DEC(CNT) after DEL;
        end if;
      end if;
    end if;
  end process;
end ALG;
```

图4-36 计数器原语

6. RAM部件

RAM部件是一种重要的时序基本部件。它是由解码函数选择的一组寄存器。因此，它可以使用已经定义的基本部件来构造。但是，为RAM单独设计一个行为模型将更加有效。

图4-37是RAM原语模型。模型是一个被NCS（片选信号）、RD（读信号）、WR（写信号）、DATA（数据输入）及ADDRESS（地址输入）的变化激活的一个简单进程。进程内部有一个对NCS变低的测试。如果NCS变低，将测试RD='1'和WR='1'，如果这两个信号中有一个为高电平，就执行进程中的操作。如果NCS为高电平，"1111"状态将被传到DATA总线上。注意到存储器访问时所使用到的类型变化：位数组BIT_VECTOR类型的信号ADDRESS变为在RAM中的INTEGER类型的数组索引。还需注意的是，使用了进程中的变量为RAM数组建模。同样可以用结构体中的信号为其建模。

这种RAM的简单模型对许多应用而言已经够用。在第5章会讨论其他一些方法。

```
use work.PRIMS.all;
entity RAM is
  generic (RDEL,DISDEL: TIME);
  port (DATA: inout BIT_VECTOR(3 downto 0);
        ADDRESS: in BIT_VECTOR(4 downto 0);
        RD,WR,NCS: in BIT);
end RAM;

architecture SIMPLE of RAM is
  type MEMORY is array(0 to 31) of BIT_VECTOR(3 downto 0);
begin
  process(RD,WR,NCS,ADDRESS,DATA)
    variable MEM: MEMORY;
  begin
    if NCS='0' then
      if RD='1' then
        DATA <= MEM(INTVAL(ADDRESS)) after RDEL;
      elsif WR='1'then
        MEM(INTVAL(ADDRESS)) := DATA;
      end if;
    else
      DATA <= "1111" after DISDEL;
    end if;
  end process;
end SIMPLE;
```

图4-37 RAM基本部件

7. 振荡器基本部件

在许多时序系统中，必须有一个振荡器来产生系统时钟。图4-38是一个时钟产生器模型，其中的振荡器是用反馈延时机制（进程输出的结果CLOCK是另一进敏感表中的一个信号）实现的。当RUN信号变高时，第一个时钟的间隔被确定，信号CLKE被置为'1'。之后，每一次CLOCK变高，下一个时钟间隔都被确定。当RUN信号变低时，CLKE被复位，当前振荡周期结束，时钟就会停止。注意，进程的结果CLOCK是从端口信号CLK得到的，这是因为CLOCK是另一进程的输入，所以它不能作为模型的输出端口信号。

振荡器也可以用*wait*语句来建模。图4-39就是这样一个例子。当RUN='1'时，CLOCK在HI_TIME时为'1'而在LOW_TIME时为'0'。HI_TIME和LOW_TIME的总和就是振荡器的一个周期；它们的相对数量就是振荡器的"duty cycle"。注意，这种模型只能在RUN的初始值为'0'的系统中使用。如果RUN的初始值为'1'，由于没有触发事件，振荡器将不会启动。

```
entity CLOCK_GENERATOR
  generic(PER: TIME);
  port(RUN: in BIT; CLK: out BIT);
end CLOCK_GENERATOR;

architecture ALG of CLOCK_GENERATOR is
  signal CLOCK: BIT;
begin
  process (RUN,CLOCK)
    variable CLKE: BIT := '0';
  begin
    if RUN='1' and not RUN'STABLE then
      CLKE := '1';
      CLOCK <= transport '0' after PER/2;
      CLOCK <= transport '1' after PER;
    end if;
    if RUN='0' and not RUN'STABLE then
      CLKE := '0';
    end if;
    if CLOCK='1' and not CLOCK'STABLE  and CLKE = '1'then
      CLOCK <= transport '0' after PER/2;
      CLOCK <= transport '1' after PER;
    end if;
    CLK <= CLOCK;
  end process;
end ALG;
```

图4-38 反馈晶体振荡器

```
entity COSC is
  generic(HI_TIME,LO_TIME: TIME);
  port(RUN: in BIT; CLOCK: out BIT := '0');
end COSC;
architecture ALG of COSC is
begin
  process
  begin
    wait until RUN ='1';
    while RUN = '1'  loop
      CLOCK <= '1';
      wait for HI_TIME;
      CLOCK <= '0';
      wait for LO_TIME;
    end loop;
  end process;
end ALG;
```

图4-39 *Wait*语句晶体振荡器

看起来进程的第一条语句“wait until RUN= ‘1’ ; ”是冗余的。但是，如果没有这条语句，仿真从RUN= ‘0’ 开始，将进入一个初始化循环，导致仿真时间不被推进。

4.4.3 模型测试：测试程序开发

任何VHDL模型的结果（大于两行）都必须在完全仿真后才可以得出是否正确的结论。模型用测试程序包进行测试。第3章列出了一些简单的测试程序包，这里讨论它们的形成。

图4-40是一个测试程序包模型。被测试模型（MUT）从激励产生器（Stimulus Generator）接收测试向量，激励产生器也同样提供预期（GOLD）输出。比较器比较MUT响应和理想输出，并得出GO/NO GO信号。一个测试包有两种反馈。第一种是模型状态可以反馈到激励产生器，它用来模拟这两种元素的交互。第二种是从比较器到激励产生器的反馈，以允许适应性测试，即一个测试的结果决定下一步进行的测试。激励产生器的功能是模拟被测模型所代表系统的环境的作用。

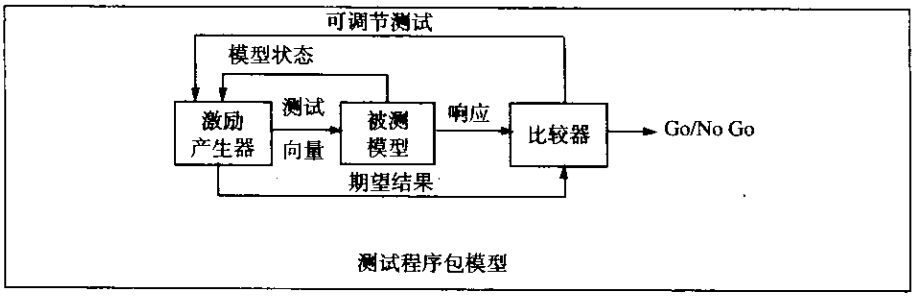


图4-40　测试程序包的组织

激励产生器的设计是测试程序开发中的主要部分。实现激励产生器的方法主要有5种。

(1) **使用信号赋值语句**

图3-9所示的ONES_CNT测试程序和图3-36所示的TWOS_CONSECUTIVE测试程序使用了信号赋值语句。在ONES_CNT测试程序中的信号赋值语句相当简单，而在TWOS_CONSECUTIVE测试程序中，信号赋值语句就比较复杂了。下面的3种方式可以缓和复杂信号赋值语句的使用。

(2) **使用文本I/O**

在很多情况下，文本I/O是产生模型输入和提供正确输出结果的最有效方法。例如，下面的测试文本可以用来测试加1计数器：

```
000 00
001 01
010 01
011 10
100 01
101 10
110 10
111 11
```

左边的三位向量是输入A；右边的两位向量是正确输出C，即理想值。图4-41是使用这个文本I/O文件的测试程序包。当信号PLAY从0变为1时，开始读取文件。每个纳秒读取一行。每一行读到的第一个向量（V1）赋值给输入A。每一行读到的第二个向量（V2）在assert语句中与输出C比较结果。

(3) **使用输入产生器基本部件**

此方法中，使用产生输入向量序列的基本部件作为组件来构建结构化模型。

图4-42是一个信号赋值语句集合，它们产生的测试向量作为JK触发器的输入。这个模型表现了测试时序模型的两个基本要求：1) 时钟的产生（信号CLK）；2) 电路输入（信号S、R、

J和K）组合的产生。如果不使用图4-42中产生CLK的复杂信号赋值语句，时钟的产生还可以采用振荡器基本部件，如图4-39所示的wait语句。为了产生输入组合，可以使用图4-43所示的模型。对于给定的整数值N，这一模型产生从0到2^N-1的整数序列，并将所有的整数转换为等价的N位二进制数。二进制输出随着周期PER的变化而变化。这种脉冲产生器对测试组合逻辑基本部件非常有效，实际中也可以用来测试本章中定义的基本部件。

```
entity TEST_BENCH
is end TEST_BENCH;

use STD.TEXTIO.all;
use WORK.all;
architecture ONES_CNT1 of TEST_BENCH is
  signal PLAY: BIT;
  signal A: BIT_VECTOR(2 downto 0);
  signal C: BIT_VECTOR(1 downto 0);
  component ONES_CNTA
    port (PLAY: in BIT; A: in BIT_VECTOR(2 downto 0);
          C: out BIT_VECTOR(1 downto 0));
  end component;
  for L1: ONES_CNTA use entity ONES_CNT(ALGORITHMIC);
begin
  L1: ONES_CNTA
    port map(PLAY, A, C);
  process
    variable VLINE: LINE;
    variable V1: BIT_VECTOR(2 downto 0);
    variable V2: BIT_VECTOR(1 downto 0);
    file INVECT: TEXT is "TVECT.TEXT";
  begin
    PLAY<= '0', '1' after 3 ns;
    wait on PLAY until PLAY = '1';
    while not(ENDFILE(INVECT)) loop
      READLINE(INVECT, VLINE);
      READ(VLINE, V1);
      READ(VLINE, V2);
      A<= V1;
      wait for 1 ns;
      assert (V2 = C)
        report "WARNING: C is NOT equal to (V2)"
          severity WARNING;
    end loop;
  end process;
end ONES_CNT1;
```

图4-41 文本I/O的测试程序包

图4-44是使用输入产生器基本部件的测试包，它用来测试JK触发器。JK触发器的时钟输入（CLK）由振荡器的输出（OSC）产生。输入S、R、J和K由四位PG的输出产生。振荡器和输入组合产生器有相同的周期，但是IGC信号在25 ns后产生，于是它的输出在时钟间隔的中间产生。

(4) 图形产生

市场上有一种工具，它可以用波形图表示输入，将其自动转换为VHDL代码的测试包。图4-45是一个用Synapticad的TestBencher Pro产生的波形。每个信号产生的VHDL描述如下：

```
S <= '1' after 10 ns, '0' after 220 ns, '1' after 920 ns,
     '0' after 1120 ns;
R <= '1' after 230 ns, '0' after 420 ns, '1' after 1320 ns,
     '0'after 1420 ns;
J <= '1' after 500 ns, '0' after 600 ns, '1' after 700 ns;
K <= '1' after 600 ns;
CLK <='1' after 50 ns,  '0' after 100 ns, '1' after 150 ns,
      '0' after 200 ns, '1' after 250 ns, '0' after 300 ns,
      '1' after 350 ns, '0' after 400 ns, '1' after 450 ns,
      '0' after 500 ns, '1' after 550 ns, '0' after 600 ns,
      '1' after 650 ns, '0' after 700 ns, '1' after 750 ns,
      '0' after 800 ns, '1' after 850 ns, '0' after 900 ns,
      '1' after 950 ns, '0' after 1000 ns,'1' after 1050 ns,
      '0' after 1100 ns,'1' after 1150 ns,'0' after 1200 ns,
      '1' after 1250 ns,'0' after 1300 ns,'1' after 1350 ns,
      '0' after 1400 ns,'1' after 1450 ns,'0' after 1500 ns,
      '1' after 1550 ns,'0' after 1600 ns,'1' after 1650 ns,
      '0' after 1700 ns;
```

图4-42 使用信号赋值语句为JK触发器产生测试向量

```
entity PULSE_GEN is
  generic(N: INTEGER; PER: TIME);
  port(START: in BIT; PGOUT: out BIT_VECTOR(N-1 downto 0));
end PULSE_GEN;
architecture ALG of PULSE_GEN is
  function INT_TO_BIN (INPUT : INTEGER;N : POSITIVE)
    return BIT_VECTOR is
    variable FOUT: BIT_VECTOR(0 to N-1);
    variable TEMP_A: INTEGER:= 0;
    variable TEMP_B: INTEGER:= 0;
  begin   -- Beginning of function body.
    TEMP_A := INPUT;
    for I in N-1 downto 0 loop
      TEMP_B := TEMP_A/(2**I);
      TEMP_A := TEMP_A rem (2**I);
      if (TEMP_B = 1) then
        FOUT(N-1-I) := '1'; else
        FOUT(N-1-I) := '0';
      end if;
    end loop;
    return FOUT;
  end INT_TO_BIN;
begin    -- Beginning of architecture body.
  process(START)
  begin
    for I in 0 to 2**N-1 loop
      PGOUT <= transport INT_TO_BIN(I,N) after I*PER;
    end loop;
  end process;
end ALG;
```

图4-43 输入组合产生器

```
process
  begin
    CLK <= '0';
    wait for 0 ns;
```

```
    while true loop
      CLK <= '1';
      wait for 10 ns;
      CLK <= '0';
      wait for 10 ns;
    end loop;
end process;
```

```
entity TEST_BENCH is
end TEST_BENCH;

use work.all;
architecture JKFF_TEST of TEST_BENCH is
  signal RUN, CLOCK, START, S, R, J, K, CLK, Q, QN: in BIT;
  signal PGOUT: BIT_VECTOR(3 downto 0);
------------ OSCILLATOR-------------------------
  component OSCILLATOR
    generic(HI_TIME, LO_TIME: TIME);
    port(RUN: in BIT; CLOCK: out BIT:= '0');
  end component;
------------ INPUT COMBINATION GENERATOR --------
  component ICG
    generic(N: INTEGER; PER: TIME);
    port(START: IN BIT; PGOUT: out BIT_VECTOR(N-1 downto 0));
  end component;
------------ JKFF -------------------------------
  component JK
    generic(SRDEL, CLKDEL: TIME);
    port(S, R, J, K: in BIT; CLK: in BIT; Q, QN: inout BIT);
  end component;
  for T1: OSCILLATOR use entity COSC(ALG);
  for T2: ICG use entity PULSE_GEN(ALG);
  for T3: JK use entity JKFF(ALG);
  signal PG: BIT_VECTOR(3 downto 0);
  signal OSC: BIT;
begin
  T1: OSCILLATOR
    generic map(25 ns, 25 ns)
    port map(RUN, OSC);
  T2: ICG
    generic map(4, 50 ns)
    port map(START, PG);
  T3: JK
    generic map( 12 ns, 15 ns)
    port map(PG(3), PG(0), PG(2), PG(1), OSC, Q, QN);
  process
  begin
    RUN<='0', '1' after 50 ns, '0' after 9000 ns;
    START<= '1', '0' after 25 ns;
    wait;
  end process;
end JKFF_TEST;
```

图4-44 使用输入产生器基本部件的测试程序包

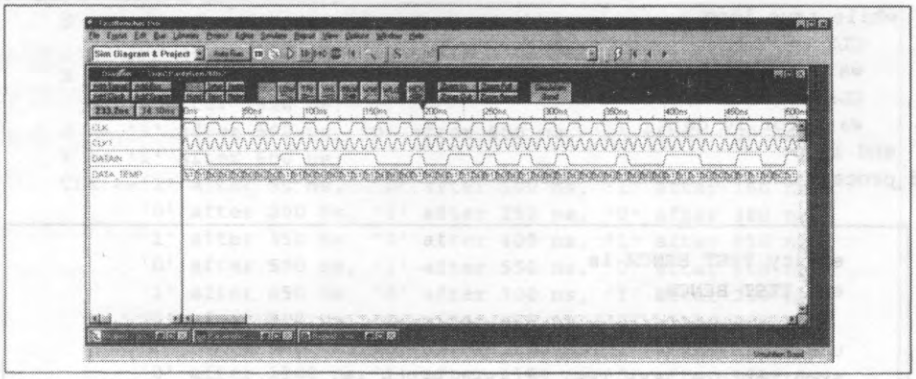


图4-45　基于图形产生输入

(5) 传感器建模

有些情况下，激励产生器是一个传感器模型，这个传感器作为物理系统的输入。在第9章和第11章将讨论这种情况。

习题

4.1　为图4-2的代码产生一个完整的VHDL描述。

a) 对描述进行仿真，输入变化的时间间隔为2ns，验证图4-1结果的正确性。

b) 改变语句(1)和(2)的顺序,并验证其结果不会发生改变。

c) 改变语句(3)和(4)的顺序，并证明其结果会发生改变。

4.2　解释delta时间的概念，并给出一个实例来说明这个概念。

4.3　给出两条语句

```
X1 <= Y after 2 ns;
X2 <= transport Y after 2 ns.
```

Y的波形如图4-46所示，画出X1、X2的响应波形图。

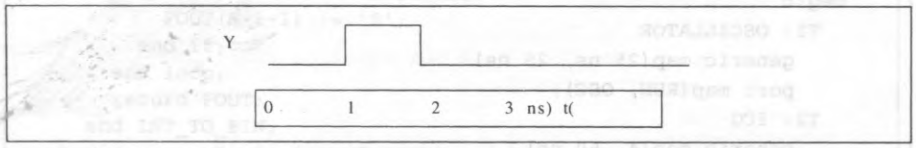


图4-46　信号Y的波形

4.4　考虑下面两条信号赋值语句：

```
Y <= transport X after 2 ns;
Z <= X after 2 ns;
```

若X的波形如图4-47，画出Y和Z的波形。

4.5　假设仿真下面的VHDL程序，

```
--
entity EX2 is
end EX2;
--
architecture TEST of EX2 is
```

```
  signal A,B,C,D: BIT_VECTOR (7 downto 0) := "11001100";
  procedure X(signal Y: in BIT_VECTOR (7 downto 0);
              signal Z: out BIT_VECTOR (7 downto 0)) is
  variable TEM: BIT_VECTOR (7 downto 0);
  begin
    for I in 0 to 7 loop
      TEM(I) := Y(7-I);
    end loop;
    Z <= TEM after 1 ns;
  end X;
begin
  S1: X(C,D);
  S2: C <= A and B after 2 ns;
  S3: A <= "10110110" after 5 ns, "10001100" after 10 ns;
  S4: B <= "11110000" after 5 ns, "00001111" after 10 ns;
end TEST;
```

回答下列问题：

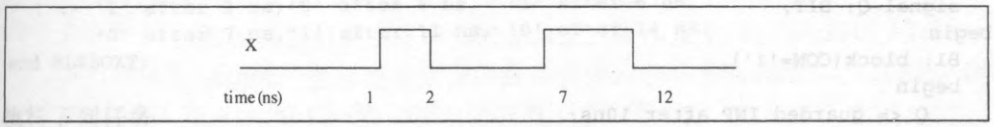


图4-47 信号X的波形

a) 对于每个标号（S1，S2，S3和S4），列出相关语句被执行的所有时间（单位为ns），假设当t=0时仿真开始；

b) 信号C的事件在哪些时候发生？

c) 画出C的波形图，列出在所有时间步上的值。

d) 信号D的最终值是多少？何时D变成最终值？解释你是如何得到答案的。

e) 简述过程X的功能。

4.6 假设仿真下面的VHDL程序。

```
--
entity FINAL is
end FINAL;
--
architecture TEST of FINAL is
  signal A,B,C,D: BIT_VECTOR (7 downto 0) := "11001100";
  procedure X(signal Y: in BIT_VECTOR (7 downto 0);
              signal Z: out BIT_VECTOR (7 downto 0)) is
    variable TEM: BIT_VECTOR (7 downto 0);
  begin
    for I in 6 downto 0 loop
      TEM(I) := Y(I+1);
    end loop;
    TEM(7) := Y(7);
    Z <= TEM after 2 ns;
  end X;
begin
  S1: C <= A xor B after 3 ns;
  S2: X(C,D);
  S3: A <= "10110110" after 7 ns, "10001100" after 15 ns;
  S4: B <= "10110110" after 7 ns, "00001111" after 15 ns;
end TEST;
```

回答下列问题：

a) 对于每个标号（S1，S2，S3和S4），列出相关语句被执行的所有时间（单位为ns），假设当t=0时仿真开始；

b) 信号C的事件在哪些时候发生？

c) 画出C的波形图，列出在所有时间步上的值。

d) 信号D的最终值是多少？何时D变成最终值？解释你是如何得到答案的。

e) 简述过程X的功能。

4.7 已经给出了实体QUESTON以及它的测试程序TB，现在分析并仿真测试程序，画出CON、INP、Q和QOUT的时序图，标出时间刻度。

```
entity QUESTION is
  port(INP: in BIT; CON: in BIT; QOUT: out BIT);
end QUESTION;

architecture ANSWER of QUESTION is
  signal Q: BIT;
begin
  B1: block(CON='1')
  begin
    Q <= guarded INP after 10ns;
    QOUT <= Q after 5 ns;
  end block;
end ANSWER;

entity TB is
end TB;

use WORK.all;
architecture QT of TB is
  signal INP,CON,QOUT: BIT;
  component QUEST
    port(INP: in BIT; CON: in BIT; QOUT: out BIT);
  end component;
  for C1: QUEST use entity QUESTION(ANSWER);
begin
  C1: QUEST
    port map(INP,CON,QOUT);
  process
  begin
    INP <= '1' after 5 ns;
    CON <= '1' after 10ns, '0' after 30 ns;
    wait;
  end process;
end QT;
```

4.8 下面是一个黑盒（BLKBOX）实体的VHDL描述：

```
entity BLKBOX is
  generic(T: TIME);
  port(I: in BIT; Z: out BIT);
end BLKBOX;

architecture INSIDE of BLKBOX is
  signal Y1,Y2: BIT;
begin
  Y1 <=  I after T;
```

```
    Y2 <=  transport I after T;
    Z  <=  Y1 xor Y2;
end INSIDE;

entity TB is
end TB;

use work.all;
architecture BLKBOXT of TB is
   signal I,Z: BIT;
   component BLKBOX
      generic(T: TIME);
      port(I: in BIT; Z: out BIT);
   end component;
   for C1: BLKBOX use entity work.BLKBOX(INSIDE);
begin
   C1: BLKBOX
      generic map(2 ns)
      port map(I,Z);
   I <= '1' after 3 ns,'0' after 4 ns,  '1' after 6 ns,
        '0' after 7 ns,'1' after 11 ns, '0' after 14 ns;
end BLKBOXT;
```

执行下列任务

a) 对于给出的测试程序TB，画出I、Y1、Y2和Z的时序图。

b) 用自然语言描述实体BLKBOX的功能。

4.9 给出信号赋值语句：

```
C <= A or B after 2 ns;
```

以及波形图（见图4-48），指出事务并说明波形更新算法如何对输入起作用。

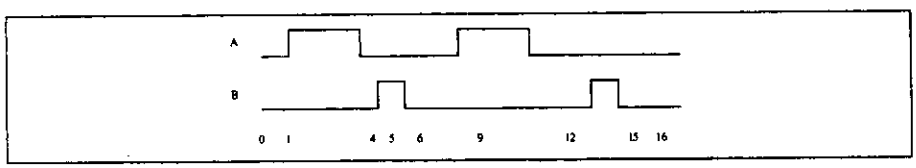


图4-48 信号A和B的波形

4.10 构架Loops（实体INT_SEQ中）的两个循环体似乎都能输出从0到9的整数，但是，其中的一个循环体并不发生作用。不发生作用的循环体是哪一个？通过对描述进行仿真来验证你的答案。

```
entity INT_SEQ is
   port(OUT1: out INTEGER; OUT2: out INTEGER);
end INT_SEQ;
architecture LOOPS of INT_SEQ is
   signal START: BIT;
begin
   START <= '1';
   process(START)
   begin
      for I in 0 to 9 loop
         OUT1 <= I after I*(1 ns);
      end loop;
   end process;
```

```
    process(START)
    begin
      for I in 0 to 9 loop
        OUT2 <= transport I after I*(1 ns);
      end loop;
    end process;
end LOOPS;
```

4.11　仿真择多（majority）/合意（consesnus）元素并验证把输出反馈为某个输入时将产生合意行为。

4.12　用两种方法为4位二进制比较器产生VHDL描述

a) 使用4个相同的比较器单元层叠而成；

b) 并行实现。

4.13　设计一个具有使能输入的2×1向量多路器（MUX），每个数据输入和输出都是8位向量，该部件通常是组合逻辑。如果使能输入为逻辑0，则输出向量是“11111111”。如果使能输入是逻辑1，则输出由选择输入和数据输入共同决定，这里有3个延时：

a) DATA_DELAY是一个数据输入变化后到输出发生变化之间的延时。

b) SELECT_DELAY是选择输入变化后到输出发生变化之间的延时。

c) ENABLE_DELAY是使能输入变化后到输出发生变化之间的延时。

描述应包括一个实体描述和一个算法级行为域结构描述。

4.14　对图4-30的PLA模型进行如下修改：

a) 修改PLA模型使之实现一个不同的逻辑功能。

b) 使该PLA模型更通用，使之可以实现对任意位输入和输出的逻辑功能。

4.15　为7400系列MSI部件建立行为域模型，时间延时用类属表示，为下面给出的模块建模；

a) BCD到7段译码器（SN 7446）

b) 4位二进制快速进位全加器（SN 7483）

c) 同步十进制乘法器（SN 74167）

d) 16位多端口三态输出寄存器文件（SN 74172）

e) 算术逻辑单元（SN 74181）

f) 向前进位产生器（SN 74182）

g) 先入先出存储器（SN 74S225）

4.16　下面给出了T触发器的真值表，其中有优先复位输入：

R	CLOCK	T	Q(t)	Q(t+1)
0	↑	1	1	0
0	↑	1	0	1
0	↑	0	1	1
0	↑	0	0	0
1	X	X	X	0

给出该触发器接口描述（实体）和行为域构架，假定该模型具有delta延时时钟。

4.17　下面给出了JK触发器的真值表，为该触发器建立一个VHDL行为域描述。用类属表示所有延时。

J	K	CLOCK	Q(t)	Q(t+1)
X	X	0	X	Q(t)
X	X	1	X	Q(t)
0	0	↑	X	Q(t)
1	0	↑	X	1
0	1	↑	X	0
1	1	↑	X	not Q(t)

注：X表示任意项，它的值可以为0也可以为1。

4.18 对于图4-49的设计树，使用VHDL设计一个3级二进制计数电路，该设计将分为两个阶段，阶段1使用设计树1构造电路，阶段2将使用设计树2。在这两个阶段都将使用一个晶体振荡器，它的实体声明如下：

```
entity OSC is
   generic(HITIME,LOWTIME:TIME);
   port(RUN: in BIT; CLK: out BIT);
end OSC;
```

HITIME和LOWTIME分别表示晶体振荡器在高电平和低电平保持时间，当RUN="1"时，晶体振荡器开始工作；当RUN="0"时则停止。

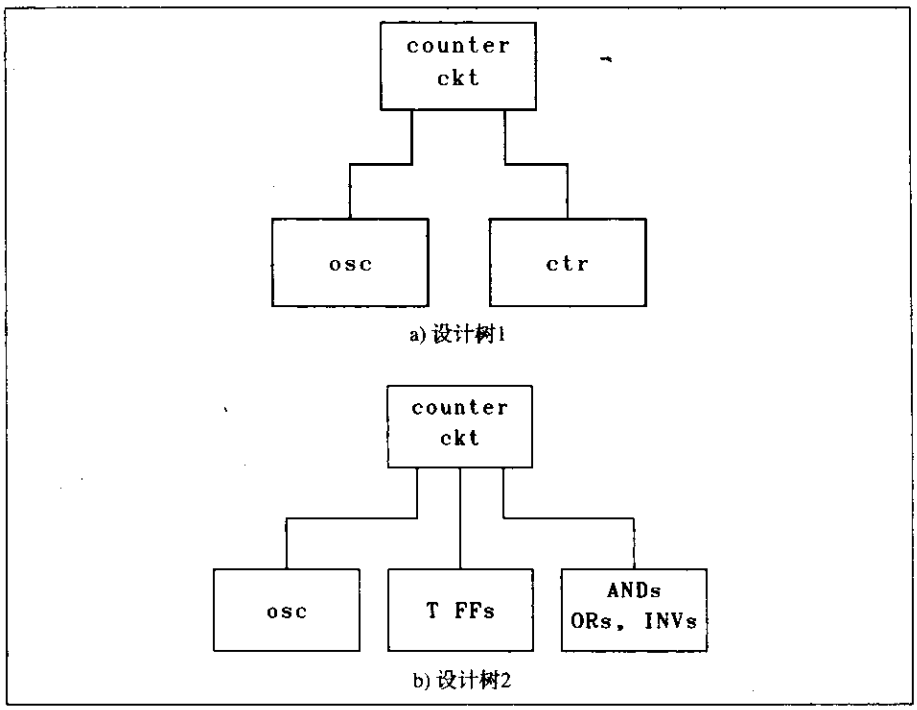


图4-49 设计树

设计阶段1

使用设计树1。为计数器产生一个算法级行为域模型，它的实体声明如下：

```
entity UP_CNT is
   generic((R_DEL,CLK_DEL: TIME);
```

```
  port(RESET,COUNT,CLK: in BIT;
       CNT: inout BIT_VECTOR(2 downto 0));
end UP_CNT;
```

RDEL和CLK_DEL表示计数器复位和时钟延时，RESET='1'表示计数器复位。当COUNT='1'且RESET='0'时，计数器正向计数，在模型实现中使用一个加1递增函数，为模型产生测试程序并进行仿真，观察它的响应。

设计阶段2：

使用设计树2，该计数器使用了T触发器及必要的AND、OR门和反相器，阶段2的设计必须满足如下要求：

a) 该计数器必须是一个3级同步计数器（不是行波进位计数器）。

b) 必须使用图所示的层次：即晶体振荡器、T触发器及其他门都是行为模型。

c) 晶体振荡器的输出为25%的工作循环（duty cycle），10MHz方波。

d) T触发器如图4-50所示，当T='1'且发生时钟正向跳变时触发器翻转，当T='0'时触发器保持前一状态，R为优先清除，当R='1'时，触发器被复位，它独立于时钟。T触发器的时序规范为复位延时=10ns，时钟延时=15ns。

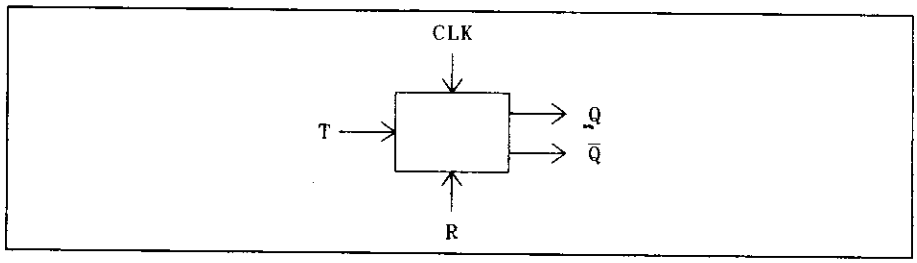


图4-50　T触发器

e) 所有的门延时均为5ns。

f) 对于计数器的每一个模型，使用类属使得该模型更通用。T触发器和2输入AND门的实体声明可以如下：

```
entity TFF is
  generic(RDEL,CLOCKDEL: TIME);
  port(R,T,CLK: in BIT; Q: out BIT);
end TFF;
entity AND2 is
  generic(DEL: TIME);
  port(I1,I2: in BIT; O: out BIT);
and AND2;
```

g) 为计数器系统开发一个结构模型，包括晶振，把这一构架插入在第1阶段开发的测试程序中并进行仿真。

准备一份包含如下内容的报告：

1) 关于每个模型是如何开发的描述。

2) VHDL模型和测试程序清单。

3) 注释后的仿真结果。

4.19　使用图4-51的JK触发器框图作为基本触发器类型，重新对习题4.18的计数器进行结构化设计，JK触发器的时序规范如下：

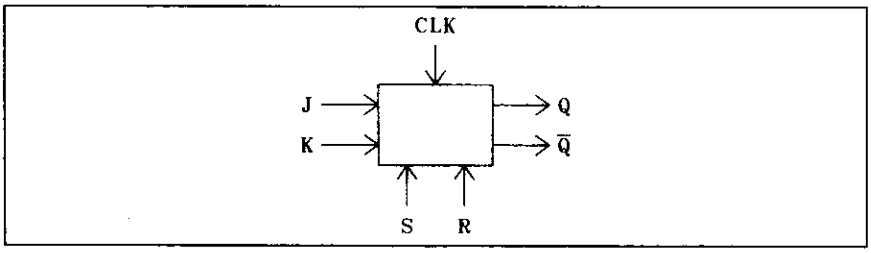


图4-51 JK触发器

参 数	输 入	输 出	说 明
TPLH	S	Q	16 ns
TPHL	S	$\bar{Q}$	25 ns
TPLH	R	$\bar{Q}$	16 ns
TPHL	R	Q	25 ns
TPLH	CLK	Q或$\bar{Q}$	16 ns
TPHL	CLK	Q或$\bar{Q}$	25 ns

逻辑门的时序规范如下：

```
TPLH   5 ns
TPHL   3 ns
```

反相器的时序规范如下：

```
TPLH   3 ns
TPHL   2 ns
```

解释在仿真用这些触发器建立的计数器时所观察到的时钟异常。

4.20 使用图4-52的D触发器框图作为基本触发器类型，重做习题4.19。D触发器的时序规范如下：

参 数	输 入	输 出	说 明
TPLH	PRESET	Q	16 ns
TPHL	PRESET	Q	25 ns
TPLH	CLEAR	Q	16 ns
TPHL	CLEAR	Q	25 ns
TPLH	CLK	Q或$\bar{Q}$	16 ns
TPHL	CLK	Q或$\bar{Q}$	25 ns

假设触发器为下降沿触发。

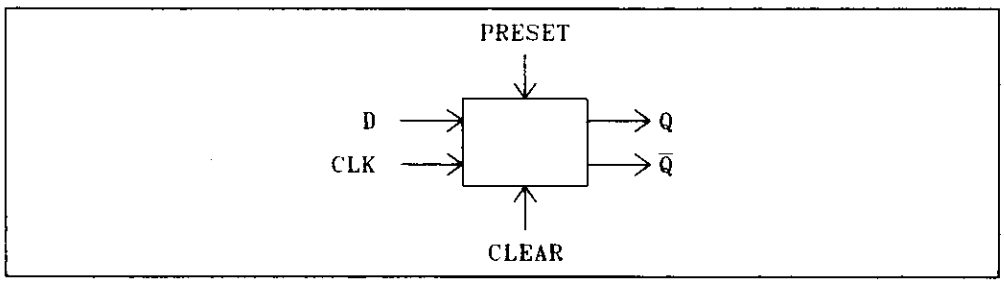


图4-52 D触发器

4.21　使用本章所涉及的基本单元实现图4-53所示的系统。系统模型必须满足以下要求：

a) 数据单元必须只使用一个四功能ALU，但其他基本单元没有限制，数据单元必须能够执行下面两种运算：

```
Z = X xor Y --- calculation 1
Z = 2*(X-Y) --- calculation 2
```

x和y来自测试程序的外部数据端口，结果Z可以保存在数据单元寄存器。假定为2的补码运算。

b) 对于每个计算，控制字序列都存放在控制存储器中（使用RAM基本部件），控制逻辑必须取出适当的控制序列，完成运算1或运算2的执行。

c) 控制逻辑可以与测试程序交互以控制开始、停止和数据载入。

d) 时钟使用晶体振荡器基本部件设计。

e) 让设计能使两种运算在最少的时间内完成，同一个设计必须用于这两种运算。

f) 用适当数量的测试实例对这两种运算进行仿真。

g) 准备一个报告总结所得的结果。

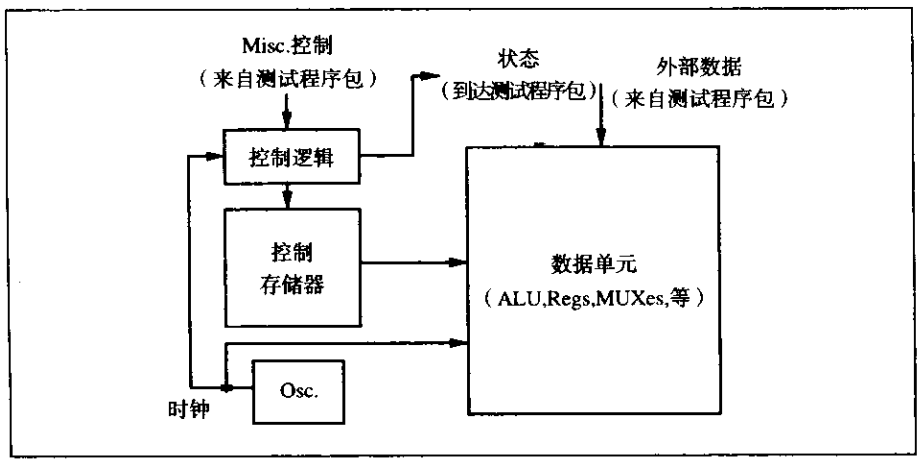


图4-53　系统框图

4.22　设计一个比本章中给出的ALU具有更强功能的ALU。重复习题4.21，讨论新的ALU在执行时间、需要使用的多路器和寄存器数量等方面的效果。

第5章　算法级设计

本章先从结构化设计开始讨论。图5-1给出的是最高一级的结构化设计块图。设计使用的是自然语言，如英语。自然语言描述常伴随着框图或者时钟图。过程的第一步将自然语言描述转换为如下三种形式：1) 真值表，2) 状态图，3) 算法模型。这三种形式都可以表示系统行为。真值表和状态图都是基本的图形表示，算法模型则采用了文本形式。本书使用VHDL语言来描述算法模型。这三种形式也都可以用来解释进程，本章选择的是算法模型。真值表和状态图在第8章详细讨论。为了使用真值表或状态图，必须从文字描述中判断真正的系统功能。而在使用算法模型时不需要这种判断。这种模型只需组织VHDL语言来匹配那些用文字描述的语句。

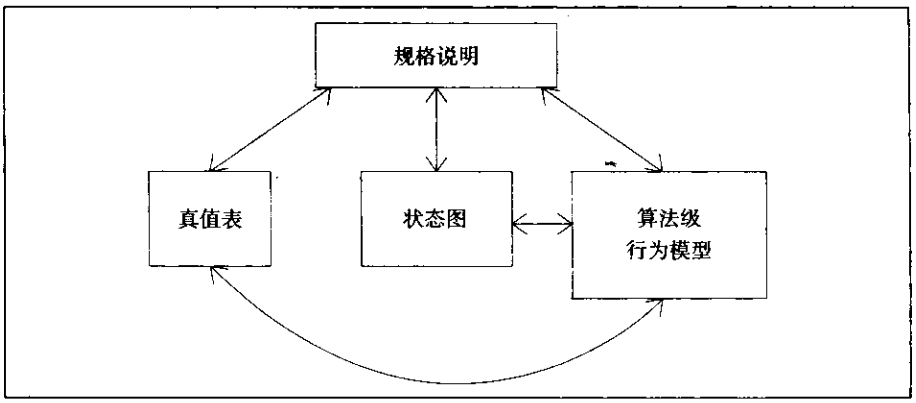


图5-1　顶级结构化设计块图

图5-1表明了真值表、状态图和算法模型的双向联系。即使首先建立了真值表或状态图，也需要为了证明它的正确性而建立一个行为模型，并对它进行模拟。相反，如果首先建立的是算法模型，之后也有可能使用真值表和状态图来表示它。因此，建立一个可以在不同表示形式之间相互转换的机制就比较重要了。

从某种程度上说，图5-1仅仅是“一个故事的开始”，在设计综合之前必须还有其他的步骤。在第8、9、10和11章，将会讨论该过程的其他方面。

5.1　行为域的一般算法模型

本节讨论如何从自然语言中建立算法模型。其步骤如下：

1) 根据需要把语句组匹映射VHDL进程。

2) 为每个进程指定一张活动表。

3) 用VHDL代码实现每个活动。

为了完成步骤1，引入图5-2的VHDL行为模型的图形表示，这个图形称为进程模型图（Process Model Graph，PMG）。

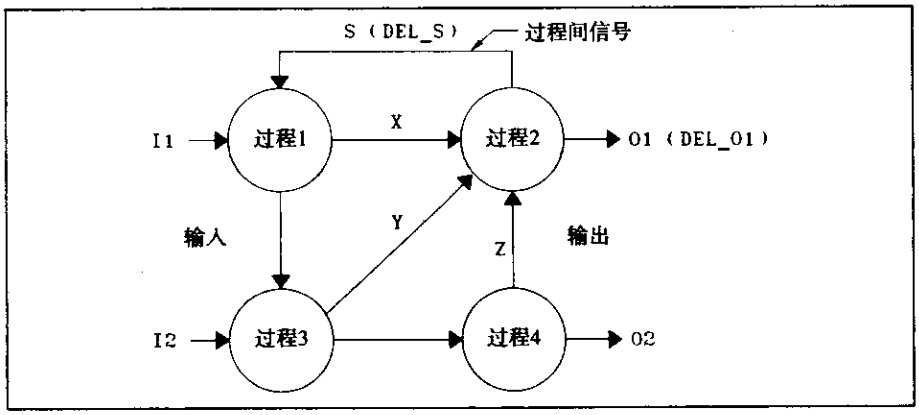


图5-2　典型的进程模型图（PMG）

5.1.1　进程模型图

VHDL中的结构体用来定义器件的行为。每个结构体都是并发执行进程的集合。进程既可以是一个进程块，也可以是不同形式的信号赋值语句。可以用进程模型图（PMG）给出行为级结构体的图形表示。图5-2就是典型行为模型的图示。图中的节点是模型的进程集合。图的边表示进程节点间的信号传输。每条边都标记有标志符，它的形式为S（DEL_S），其中S是信号名称，DEL_S是进程传输延时。例如，图中节点2的行为描述如下：

```
PROC_2: process(X,Y,Z)
  variable SVL,OVL: BIT;
begin
  --------  Function of the process: compute SVL and OVL
  S <= SVL after DEL_S;
  O1 <= OVL after DEL_O1;
end PROC_2;
```

DEL_S是驱动信号S的信号赋值语句的延时，DEL_O1是驱动信号O1的信号赋值语句的延时。

进程模型图表明了模型的一种划分。这种划分既可以是物理划分又可以是功能划分。物理划分时，边上的延时值表示特定块的传播延时，块表示实际逻辑门或触发器。在某些情况下，特别是设计的早期，一个PMG可以表示模型纯功能的划分。图5-3就是这样一个例子，它是一个处理器的进程模块图。节点分别是取部件、执行部件，等等。它们所表示的功能在硬件上没有特定的块与之对应。实际的某些物理实现中，两个功能节点可以共享一个逻辑块。例如，取部件和执行部件访问同一个“真正”的存储器。

在产生进程模块图时，要有许多情况要处理。图5-4列出了其中的一些。

1) 图5-4(a)表示两个信号到达同一个节点。实心箭头表示在进程敏感信号表中的信号。该信号产生一个事务时，进程被激活。空心箭头表示的信号被进程采样但并不激活进程执行。

2) 图5-4(b)的边上含有信号名（S），但没有给出延时值。这表示驱动信号赋值的延时时间是delta。

3) 图5-4(c)是一个单向信号流，该信号由两个进程产生，这里隐含了总线决断功能。

4) 图5-4(d)是两个进程间的双向信号流。它也隐含了总线决断功能。

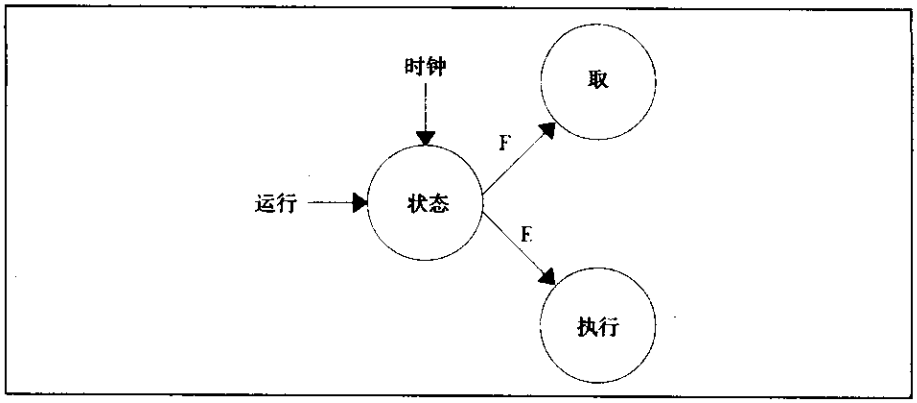


图5-3 处理器的PMG

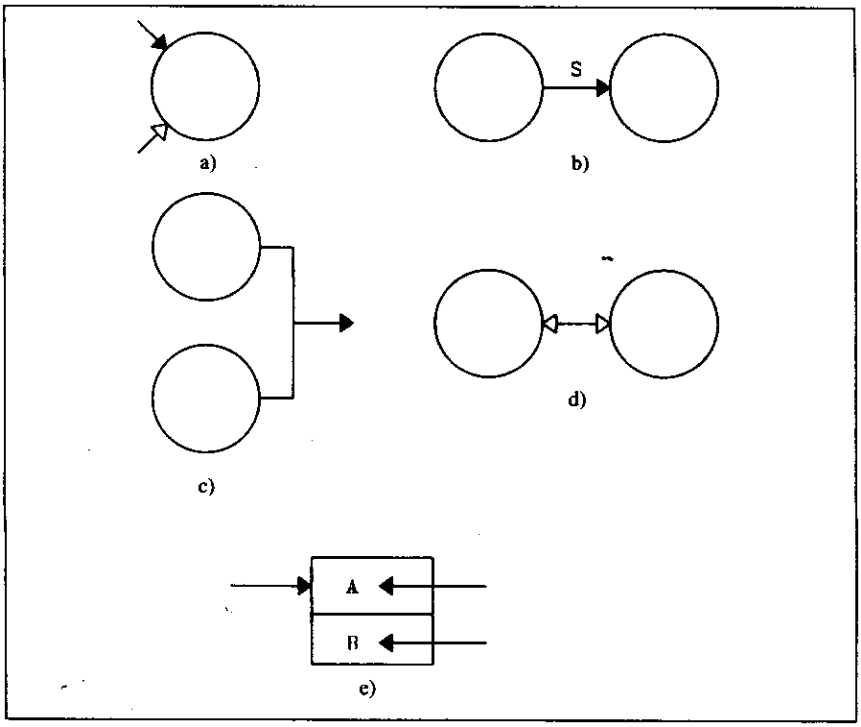


图5-4 PMG的构造

5) 一些模型使用了防卫块作为模型的主要部分。但是，防卫块是有特殊触发（防卫信号）的赋值语句的集合。这些赋值语句仅仅是进程的集合。接触块外部的边界信号称为防卫信号。另一些边指向块的内部进程，它们是这些进程的输入。

总之，进程结构图可以清晰表示行为域中算法级模型的结构。如果该图表示的是模型的物理划分，它可以清晰表明传播延时和系统结构。使用进程结构图后可以产生更易于理解的全面而又简单的模型。下面将看到，它们是算法模型建模的一个很好的开端。

5.1.2 并行到串行转换器的算法模型

本例中的算法模型针对并行到串行转换器（PARSER）。图5-5是转换器的框图。系统的自

然语言描述如下：

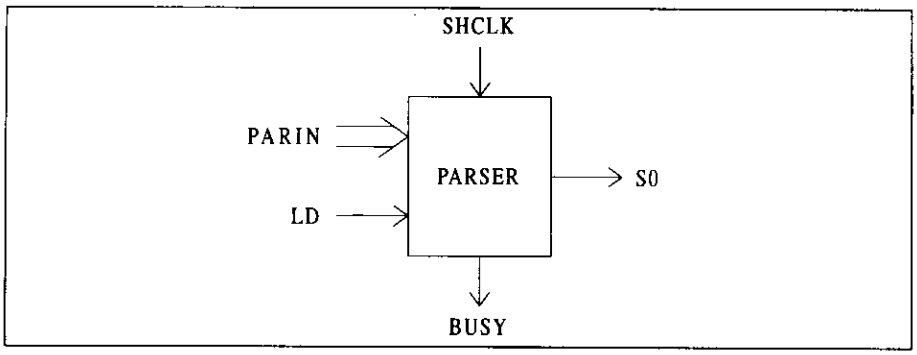


图5-5　并行到串行转换器的框图

8位并行字（PARIN）在控制信号LD由0变为1时载入转换器中。此时的状态信号BUSY被置为高电平。数据串行移出，其速率由输入移位时钟SHCLK控制。移位发生在时钟上升沿。BUSY在移出完成前一直保持高电平。当BUSY为高电平时，不能接收下一个输入。

前面曾经指出，算法模型的建立过程就是将自然语言的句子映射到VHDL的进程。通常有不止一种方法。本例中最直观的选择是使用两个进程：LOAD和SHIFT。自然语言需求到过程的对应关系如下：

1) 进程LOAD。a) 8位并行字（PARIN）在控制信号LD由0变为1时载入转换器中。此时的状态信号BUSY被置为高电平。b) BUSY在移出完成前一直保持高电平。当BUSY为高电平时，不能接收下一个输入。

2) 进程SHIFT。a) 数据串行移出，其速率由输入移位时钟SHCLK控制。移位发生在时钟上升沿。b) BUSY在移出完成前一直保持高电平。

注意到“BUSY在移出完成前一直保持高电平”的要求被两个进程共享。这是因为进程SHIFT将检测移位是否完成，并把该条件中用信号以通知LOAD进程，Load进程再重置BUSY信号。

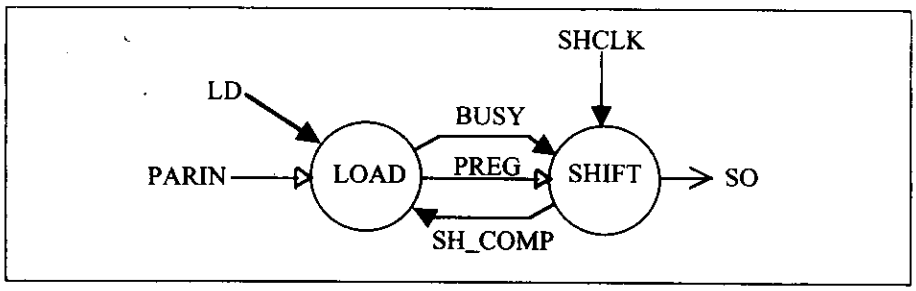


图5-6　并行到串行转换器的PMG

图5-6是算法模型的进程模块图。进程LOAD被信号LD触发后将输入向量PARIN载入到内部寄存器（PREG）中，并把信号BUSY置为高电平。BUSY的变高将触发进程SHIFT。进程SHIFT再将寄存器PREG中的数据读到串行寄存器中，并对数据移位，将其输出到串行线路SO。当移位完成后，进程SHIFT将信号SH_COMP置为‘1’，它将触发进程LOAD，使其将BUSY信号复位。模型的外壳定义如下：

```
entity PAR_TO_SER is
  port(LD,SHCLK: in BIT; PARIN: in BIT_VECTOR(0 to 7);
       BUSY: inout BIT; SO: out BIT);
end PAR_TO_SER;

architecture TWO_PROC of PAR_TO_SER is
  signal SH_COMP: BIT :='0';
  signal PREG: BIT_VECTOR(0 to 7);
begin
  LOAD:process(LD,SH_COMP)
  begin
    ---- Activities:
      ----1)Register Load
      ----2)Busy Set
      ----3)Busy Reset
  end process LOAD;

  SHIFT:process(BUSY,SHCLK)
  begin
    ----Activities:
      ----1)Shift Initialize
      ----2)Shift
      ----3)Shift Complete
  end process SHIFT;
end TWO_PROC;
```

进程模块图暗含了接口定义和进程的框架。以及进程间的信号。进程框架明确了每个进程的敏感信号表中的信号。基于自然语言规范和进程的映射，可以用描述了进程功能的活动列表为进程框架作注释。要进一步开发模型，可为每个活动编写VHDL代码。下面就是编写的结果。每个活动的代码包括两个部分：1) 控制语句，本例中是if语句，用于检查事件是否发生或条件是否满足。2) 数据操作：检测到事件发生或条件满足时被执行。

```
entity PAR_TO_SER is
  port(LD,SHCLK: in BIT; PARIN: in BIT_VECTOR(0 to 7);
       BUSY: inout BIT := '0'; SO: out BIT);
end PAR_TO_SER;

architecture TWO_PROC of PAR_TO_SER is
  signal SH_COMP: BIT :='0';
  signal PREG: BIT_VECTOR(0 to 7);
begin
  LOAD:process(LD,SH_COMP)
  begin
    ---- Activities:
    if LD'EVENT and LD='1'and BUSY='0' then
      ----1)Register Load
      PREG <=  PARIN;
      ----2)Busy Set
      BUSY <= '1';
    end if;
      if SH_COMP'EVENT and SH_COMP='1' then
        ----3)Busy Reset
        BUSY <= '0';
      end if;
  end process LOAD;
  SHIFT:process(BUSY,SHCLK)
    variable COUNT: INTEGER;
    variable OREG: BIT_VECTOR(0 to 7);
  begin
```

```
      ----Activities:
      if BUSY'EVENT and BUSY = '1' then
         ----1)Shift Initialize
         COUNT := 7;
            OREG := PREG;
         SH_COMP <= '0';
      end if;
      if SHCLK'EVENT and SHCLK= '1'and BUSY='1' then
         ----2)Shift
         SO<=OREG(COUNT);
         COUNT := COUNT - 1;
         ----3)Shift Complete
         if COUNT < 0 then
            SH_COMP <= '1';
         end if;
      end if;
   end process SHIFT;
end TWO_PROC;
```

5.1.3 带定时的算法模型

上面的例子中并没有包括对定时关系的建模。本节中将讨论如何对器件的输入/输出定时进行建模。举一个简单的例子，为一个8位缓冲寄存器建模。图5-7是这个缓冲寄存器的框图和它的定时规范。对寄存器的描述如下：

寄存器在选通信号（STRB）的上升沿装入，假设启用输出缓冲，缓冲的输出将在t_{SD}纳秒后发生改变。寄存器缓冲的使能条件是输入DS1和DS2的“与”操作结果。使能条件的任何变化将在t_{ED}纳秒后改变输出的结果。

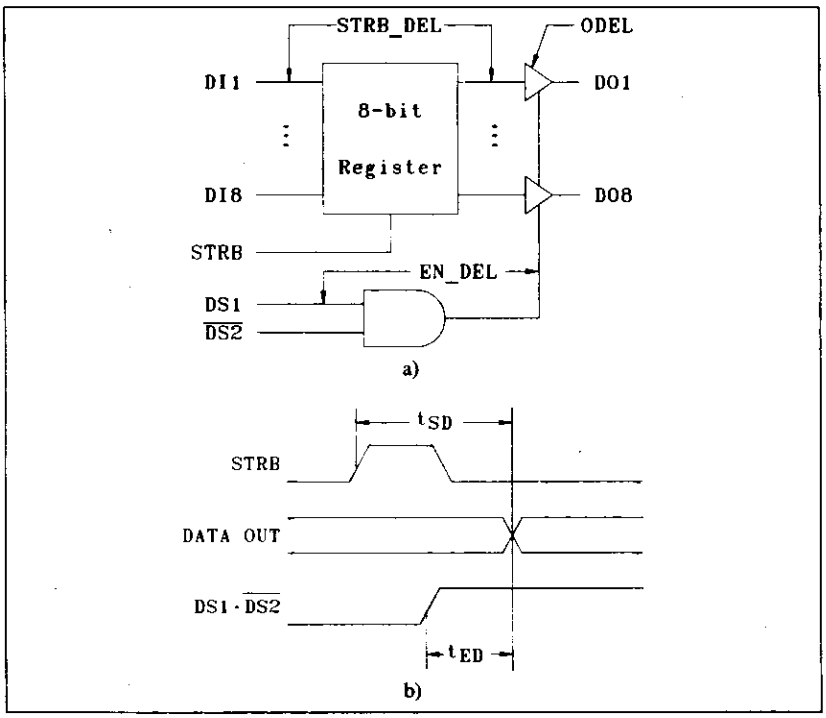


图5-7 缓冲寄存器的时序定义和框图

在分析这个说明时，注意到从输入到输出有两个信号通路：1) 从寄存器到输出的数据通路。2) 从使能输入到输出的控制通路。每一条通路都必须用一个独立的信号来表示。实际上，可以表述为如下的规则：

规则1：在包含时序的算法模型中，独立的输入/输出信号路径必须用分离的VHDL信号表示。并行路径中的信号可以被同时激活。

根据这条规则，可以确保所有路径上可能的并发信号行为均已建立了模型。

为了对缓冲寄存器模型的两条路径建立模型，假设t_{SD}=STRB_DEL+ODEL，而且t_{ED}=EN_DEL+ODEL。STRB_DEL表示从选通信号的上升沿到数据到达寄存器输出端口的延时时间。EN_DEL表示数据从输入到输出缓冲的延时时间；ODEL表示通过输出缓冲的延时。

F面将会使用三个进程，它们的自然语言描述如下：

1) PREG。寄存器在选通信号（STRB）的上升沿载入数据。

2) ENABLE。寄存器缓冲的使能条件是新DS1和DS2的“与”操作结果。

3) OUTPUT。a) 假设输出缓冲被使能，缓冲的输出将在t_{SD}纳秒后改变；b) 使能条件的任何变化都会导致输出在t_{ED}纳秒后发生改变。

图5-8是该寄存器的进程模块图。进程PREG接收信号和输入数据，在延时一段时间后输出数据。进程ENABLE产生使能信号（ENBLD）的一个延时值。进程OUTPUT接收信号REG和ENBLD，并产生输出DO。图中还包括这三个进程的延时（STRB_DEL，EN_DEL，ODEL）。因此，根据规则1，两个分离的数据路径在模型中分别表示为信号REG和ENBLD。

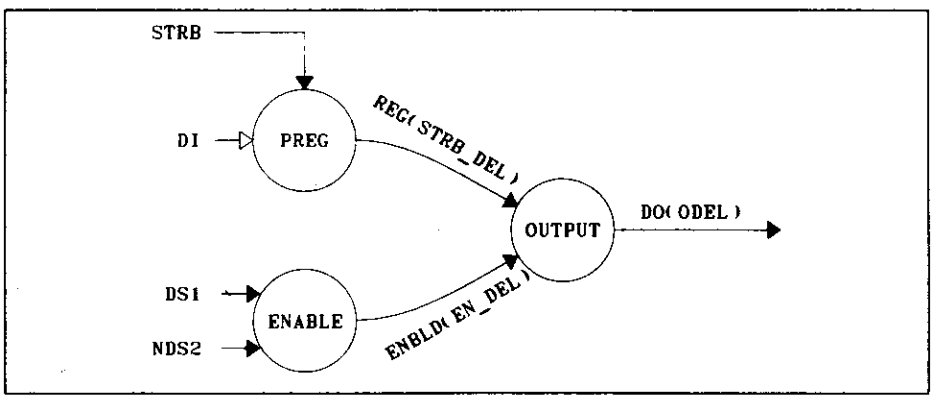


图5-8 缓冲寄存器的PMG

图5-9是三个进程的完整VHDL描述。在接口描述中，模型的三个延时被定义为generics。它允许这个模型是可能通用。当模型在一个互连系统描述里被实例化时，必须规定类属参数的值。以上的讨论中，图5-9的VHDL模型具有自解释能力。然而，在输出进程里，由于没有使用多值逻辑，输出的高阻状态表示为一个全‘1’向量。

其他的建模配置也是可能的：例如，进程PREG和ENABLE可以结合成一个进程，该进程具有这两个进程的功能：

```
FRONT_END: process(STRB,DS1,NDS2)
begin
  if STRB'EVENT and STRB = '1' then
```

```
    REG <=DI after STRB_DEL; --activity 1
  end if;
  if DS1'EVENT or NDS2'EVENT then
    ENBLD <= DS1 and  not NDS2 after EN_DEL;  --activity 2
  end if;
end process FRONT_END;
```

```
entity BUFF_REG is
     generic(STRB_DEL,EN_DEL,ODEL: TIME);
     port(DI:  in BIT_VECTOR(1 to 8);
       STRB: in BIT;DS1:  in BIT;
       NDS2: in BIT;
       DO: out BIT_VECTOR(1 to 8));
  end BUFF_REG;
  --
  architecture THREE_PROC of BUFF_REG is
     signal REG: BIT_VECTOR(1 to 8);
     signal ENBLD: BIT;
  begin
  PREG: process(STRB)
     begin
        if (STRB = '1') then
           REG <=DI after STRB_DEL;
        end if;
  end process PREG;
  --
  ENABLE: process(DS1,NDS2)
     begin
        ENBLD <= DS1 and  not NDS2 after EN_DEL;
  end process ENABLE;
  --
  OUTPUT: process(REG,ENBLD)
     begin
        if (ENBLD = '1')
        then
           DO <= REG after ODEL;
        else
           DO <= "11111111" after ODEL;
        end if;
  end process OUTPUT;
  end THREE_PROC;
```

图5-9 缓冲寄存器的算法级VHDL描述

这样就把两个具有单独行为的进程合成为一个进程的两个行为。然而，信号REG和ENBLD在这两种方法里的控制机制完全相同。这里有一个基本的问题，算法描述中如何对使用进程的多少进行折衷？该问题的答案取决于以下因素：

1) 信号的数量。使用多个进程会导致描述中有更多的信号。更多的信号又需要有更多的进程队列，这样会影响仿真的效率。

2) 单个进程的复杂度。如果单个进程非常复杂，进程的代码可能无法说明输入变化的所有可能的组合。这种情况会限制模型的实际应用。使用更简单的模型可以避免这种问题。

3) 映射的难易程度。使用了更多的进程更易于在需求说明与进程之间建立自然的映射；例如，使用PREG、ENABLE及OUTPUT进程是对需求定义的一种很自然的映射。

第3章已经说明，单个信号赋值语句是一个进程；因此，同样的模型结构可以被更加紧凑

地表示。使用这种方法的另一种寄存器模型的结构如下：

```
architecture DATA_FLOW of BUFF_REG is
begin
  B:block(STRB = '1'and not STRB'STABLE)
    signal REG: BIT_VECTOR(1 to 8);
    signal ENBLD: BIT;
  begin
    REG <= guarded DI after STRB_DEL;          ---process REG
    ENBLD <= DS1 and not NDS2 after EN_DEL;  ---proc ENABLE
    DO <=REG after ODEL when ENBLD = '1'      ---proc OUTPUT
         else "11111111" after ODEL;
  end block B;
end DATA_FLOW;
```

这里使用了防卫信号strobe从0变到1的防卫块。三个进程PREG、ENABLE、OUTPUT都由一条赋值语句实现。对应进程REG的语句被防卫，它只能在防卫条件为TRUE时才被执行。这种进程更加精确，但模型的结构不是十分明显。对于更复杂的模型，使用进程来实现一个复杂的I/O公式会更加有效。从结构体的名字可以看出，模型的这种简洁形式是一种数据流描述。下一章里会更加详细地讨论这种描述。

5.1.4 定时检查

第4章曾经讨论过，信号赋值语句的内部惯性延时会过滤掉信号中持续时间少于after语句所定义的最小延时值的脉冲。于是，对于寄存器模型，为信号REG、ENBLD及DO所置的新值必须至少保持在对应的STRB_DEL、ENBLD及ODEL的延时值之后，新值才会起作用。但是，设计者的意图并不意味着短的信号脉冲会被删除，实际上它表示该信号不发生变化。

在打印表示短脉冲被删除的提醒信息将很有用处，因为这种情况往往是一种可能的建模错误。此外，复杂定时约束的定义通常需要检验。例如，考虑刚刚建模的缓冲寄存器。寄存器的一个基本要求是数据输入在一定的时钟周期里保持稳定，然后才会被写入寄存器。这一要求被称为“建立时间（setup time）”，它定义数据与时钟的对应关系。另一个要求也可以陈述为数据在时钟跳变后必须保持一段最小的时间（即“保持时间（hold time）”）。VHDL允许使用断言（assertion）语句来检测时钟约束是否满足：

```
assert      Boolean Expression
  report      Error Message;
```

执行断言时检测布尔表达式，值为TRUE表示正常状态，FALSE表示错误状态，并打印出错误信息。断言可以检测任何实际问题，但是在本应用里只用来检测信号的时序。对于定时断言，Boolean表达式的内容是信号属性。两个非常有用的信号属性是X'stable(T)及X'delayed(T)。第3章曾指出，当且仅当信号X在T时间单元里保持稳定时X'stable(T)为TRUE。X'delayed(T)表示X在T时间单元前的值。

下面解释在定时模型里如何使用断言。图5-10是图5-7所示寄存器模型典型输入的定义。该定义为：

1) DI必须在STRB上升前保持SUT纳秒（建立时间）。

2) DI必须在STRB上升后保持HT纳秒（保持时间）。

3) STRB在高电平或低电平必须至少保持MPW纳秒（最小脉冲宽度）。

下面的断言检测建立时间的定义：

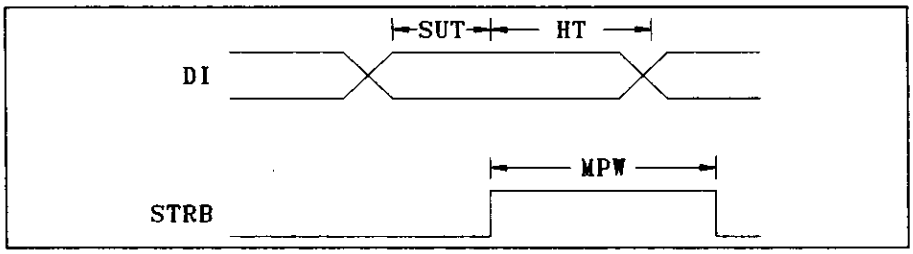


图5-10 缓冲寄存器的输入定义

```
assert not (not STRB'stable and (STRB = '1')
            and not DI'stable(SUT))
  report  "Setup Time Failure";
```

如果选通信号变高，两个数据输入并没有预先保持SUT ns，将报告建立时间错误。使用德·摩根定律，可将上述断言语句转换为更简单的形式：

```
assert STRB'stable or (STRB = '0') or DI'stable(SUT)
  report  "Setup Time Failure";
```

同样，使用德·摩根定律也可以写出保持时间测试语句的简单形式：

```
assert  STRB'delayed(HT)'stable or (STRB'delayed(HT) = '0')
        or DI'stable(HT)
  report "Hold Time Failure";
```

检测最小脉冲宽度的断言语句为：

```
assert  STRB'stable or (STRB = '1')
        or STRB'delayed'stable(MPW)
  report "Minimum pulse width failure";
```

该表达式的最后一项用STRB'delayed取代了STRB'stable，这是因为我们期望检测STRB在变化之前是否保持了一个delta周期。而检测它变化之后的保持时间是毫无意义的。

时序断言也可以通过如S'last_event及来自标准包中的函数NOW的属性产生。我们把它留作读者的练习。

断言可以插入到接口描述或设计实体的结构体中。如果把断言插入到结构体中，它被定义为设计实体的一种特殊实现。如果插入到接口描述中，它可以用来检测设计实体所有结构体的输入输出信号的时序。下面是缓冲寄存器模型接口描述的断言语句，它定义在模型的声明区域中。

```
entity BUFF_REG is
  generic(STRB_DEL,EN_DEL,ODEL,SUT,HT,MPW: TIME);
          port(DI:  in BIT_VECTOR(1 to 8);
               STRB: in BIT;DS1:  in BIT;
               NDS2: in BIT;
               DO: out BIT_VECTOR(1 to 8));
begin
  assert STRB'stable or (STRB = '0') or DI'stable(SUT)
    report  "Setup Time Failure";
  assert STRB'delayed(HT)'stable
         or (STRB'delayed(HT) = '0')
         or DI'stable(HT)
  report "Hold Time Failure";
  assert STRB'stable or (STRB = '1')
```

```
        or STRB'delayed'stable(MPW)
    report "Minimum pulse width failure";
end BUFF_REG;
```

开始时曾指出我们的目标是完成自然语言到VHDL进程的映射。这对于断言所代表的定时需求同样正确，因为在仿真期间每一个断言都由一个“被动的（passive）”进程所表示，即表示一个监测信号但不改变信号的进程。

5.2 系统互连的表示

VHDL中使用代表系统实体的结构体来实现系统组件的互连。考虑图5-11所示的简单系统X：

系统X包括两个元件，实体A和实体B，它们的接口描述如下：

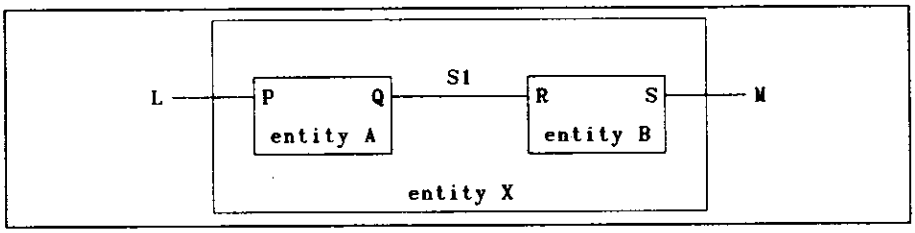


图5-11 系统互连模型

```
entity A is
  port(P: in BIT; Q: out BIT);
end A;

entity B is
  port(R: in BIT; S: out BIT);
end B;
```

系统X的互连结构定义为如下的实体声明和相应结构体：

```
entity X  is
  port(L: in BIT; M: out BIT);
end X;

architecture STRUCTURAL of X is
  signal S1: BIT; -- Internal interconnection signal declared
  component A     -- Component declared
    port(P: in BIT; Q out BIT);
  end component;
  component B     -- Component declared
    port(R: in BIT; S out BIT);
  end component;
begin
  A1: A port map(L,S1);  ----Component instantiation
  B1: B port map(S1,M);  ----Component instantiation
end STRUCTURAL;
```

系统组件的互连由元件实例化语句中的端口映射来定义。端口映射定义了连接到各元件端口的系统信号。上面所举的简单例子里，信号S1的定义是件A和件B的连接，这是因为它既是元件A的输出又是元件B的输入。

在上面的例子里，当元件实例化时，通过实例化语句中的端口映射项与元件声明里的端

口项的位置匹配来指定互连。它也可以用命名的关联来指定。例如，可以把实例化语句写为：

```
A1: A port map(Q => S1, P => L);
B1: B port map(S => M , R => S1);
```

其中，语句的位置并不重要，命名的关联确定了对应关系。

5.2.1 综合性算法建模实例

我们设计了一个多模块系统来对算法建模进行总结，它进一步说明了规格说明的解释和到进程的映射。

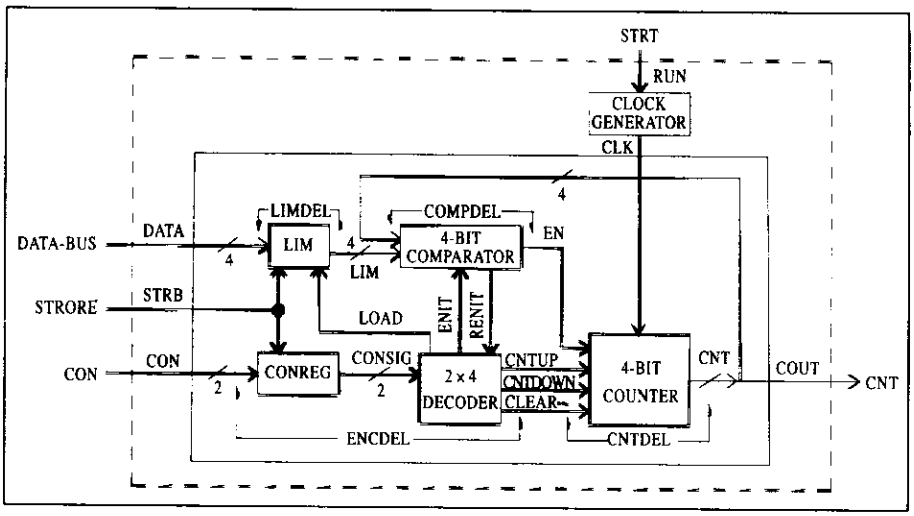


图5-12　具有两个模块系统的框图

图5-12是一个双模块系统的框图，它包括时钟产生器和一个4位控制计数器。时钟产生器产生占空比为50%的时钟，周期为PER ns。计数器包括五个逻辑块：1) 2位控制寄存器（CONREG）；2) 2到4解码器；3) 4位限制寄存器（LIM）；4) 4位比较器；5) 4位计数器。计数器模块控制输出（CON），该信号存储在CONREG中，然后解码并执行如下四条命令：

1)（CON =00）。清空计数器。

2)（CON =01）。从输入DATA线上载入LIM（limit）寄存器。

3)（CON =10）。加计数，直到COUNTER等于LIM寄存器中的值。

4)（CON =11）。减计数，直到COUNTER等于LIM寄存器中的值。

当计数器得到计数命令且使能信号变为高电平（EN = 1）时，它工作在递增或递减的模式，直到达到限制值。当计数命令为解码时，ENIT信号发送到解码逻辑的比较器。比较器在ENIT的上升沿把信号EN初始化为1。然后，EN受到比较器的控制。

框图里有四个传输延时：1) ENCDEL——通过控制寄存器（CONREG）和解码器的延时之和；2) LIMDEL——通过限制寄存器的延时；3) COMDEL——通过比较器的延时；4) CNTDEL——通过计数器的延时。接口时序功能如下所述。CONREG在STRB上升沿载入，LIM在STRB下降沿载入。信号CON和DATA都需要有一段建立时间，对应于它们各自的STRB跳变为SUT纳秒。信号STRB本身也需要有MPW的最小脉冲宽度。

图5-13是控制计数器的进程模型图。控制寄存器和解码器的功能表示为单个节点。只是

因为时序定义给出了两个块的总延时，而不是单个延时。因此，该节点的模型是单个VHDL进程。

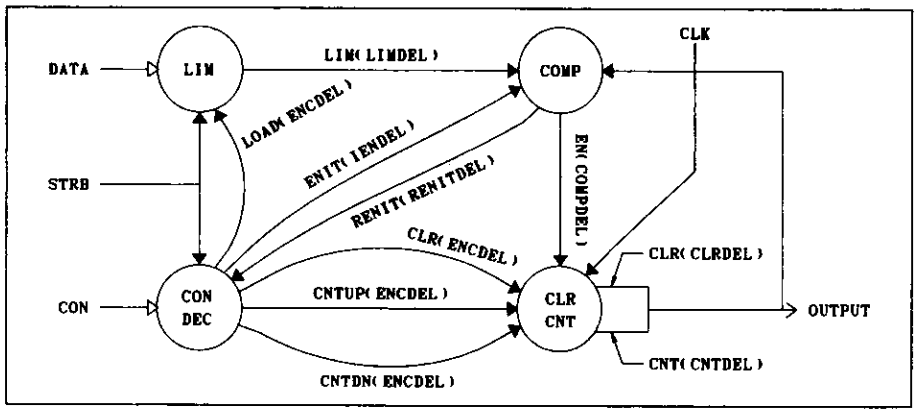


图5-13 控制计数器的进程模型图

图5-14和图5-15是该系统的VHDL模型。其中有两个实体——一个是时钟产生器（图5-14），另一个是4位控制计数器（图5-15）。

时钟产生器的振荡器用一个带有反馈延时机制的进程实现。该进程产生的信号CLOCK在进程敏感表中。实体的输出是CLK，它等于一个delta延时后CLOCK的值。由于VHDL不允许模型的输出信号在实体中被访问，所以才使用了这种缓冲机制。另外，时钟产生器只有在输入信号RUN由0变为1时才会产生时钟，而且在RUN下降后被抑制。

```
entity CLOCK_GENERATOR is
   generic (PER: TIME);
   port(RUN: in BIT;CLK: out BIT);
end CLOCK_GENERATOR;
--
architecture FEEDBACK of CLOCK_GENERATOR is
   signal CLOCK: BIT;
begin
  process (RUN,CLOCK)
    variable CLKE: BIT := '0';
  begin
    if RUN'EVENT then
      if RUN = '1' then
        CLKE := '1';
        CLOCK <= transport '0' after PER/2;
        CLOCK <= transport '1' after PER;
      else
        CLKE := '0';
      end if;
    end if;
    if(CLOCK'EVENT and CLOCK = '1' and CLKE = '1' ) then
      CLOCK <= transport '0' after PER/2;
      CLOCK <= transport '1' after PER;
    end if;
    CLK <= CLOCK;
  end process;
end FEEDBACK;
```

图5-14 时钟产生器的算法描述

控制计数器的实体描述（图5-15）首先定义了模型的时间延时类属。在关键字begin和end之间有一个时序断言。构架包括四个进程：DECODE，LOAD_LIMIT，CTR，以及LIMIT_CHK。进程DECODE将2位输入CON解码为4位信号CONSIG。它同时产生信号ENIT，该信号用于初始化比较器的使能端。进程LOAD_LIMIT当STRB下降且解码器输出为1时载入LIM寄存器。进程CTR在时钟使能信号（CLKE）置位时响应它的时钟输入（CLK）。CLKE在EN的上升沿被置位，比较器的使能信号在它变低时被复位。进程LIMIT_CHK在接收到信号ENIT的时候把使能信号EN置为'1'。它响应信号RENIT，RENIT在进程DECODE里被用来重置信号ENIT。进程LIMIT_CHK使用ENIT的下降沿为RENIT重新赋值。使能信号EN在计数达到限制值时被重置。

```
entity CONTROLLED_CTR is
  generic(SUT,MPW,ENCDEL,ENITDEL,RENITDEL,CLRDEL,
          CNTDEL,LIMDEL,COMPDEL: TIME);
  port(CLK,STRB: in BIT;
       CON:  in BIT_VECTOR(0 to 1);
       DATA:  in BIT_VECTOR(0 to 3);
       COUT: out BIT_VECTOR(0 to 3));
begin
  assert STRB'STABLE or STRB = '1' or DATA'STABLE(SUT)
    report "Set up time failure on DATA input"
    severity NOTE;
  assert STRB'STABLE or STRB = '0' or CON'STABLE(SUT)
    report "Set up time failure on CON input"
    severity NOTE;
 assert STRB'STABLE or STRB = '1' or STRB'DELAYED'STABLE(MPW)
    report "Pulse width failure on STRB"
    severity NOTE;
end CONTROLLED_CTR;
use work.counter_pac.all;
architecture PROCESS_IMPL of CONTROLLED_CTR is
  signal ENIT,RENIT: BIT;
  signal EN: BIT;
  signal CONSIG,LIM: BIT_VECTOR(0 to 3);
  signal CNT: BIT_VECTOR(0 to 3);
begin
  --
DECODE:process(STRB,RENIT)
  variable CONREG: BIT_VECTOR(0 to 1) := "00";
 begin
  if (STRB = '1') then
    CONREG := CON;
    case CONREG is
    -- Signal CLEAR is CONSIG(0).
      when "00" => CONSIG <= "1000" after ENCDEL;
    -- Signal LOAD is CONSIG(1).
      when "01" => CONSIG <= "0100" after ENCDEL;
    -- Signal CNTUP is CONSIG(2).
      when "10" => CONSIG <= "0010" after ENCDEL;
                   ENIT <= '1' after ENITDEL;
    -- Signal CNTDOWN is CONSIG(3).
      when "11" => CONSIG <= "0001" after ENCDEL;
                   ENIT <= '1' after ENITDEL;
    end case;
  end if;
  if RENIT'EVENT and RENIT = '1' then
     ENIT <= '0' after ENITDEL;
  end if;
 end process DECODE;
```

图5-15 控制计数器的行为域算法描述

```
LOAD_LIMIT:process(STRB)
 begin
  if(CONSIG(1)='1' and STRB'EVENT and STRB ='0') then
    LIM <= DATA after LIMDEL;
  end if;
 end process LOAD_LIMIT;
--
CTR: process(CONSIG(0),EN,CLK)
   variable CNTE: BIT:= '0';
 begin
  if (CONSIG(0)='1' and CONSIG(0)'EVENT) then
    CNT <= "0000" after CLRDEL;
  end if;
  if EN'EVENT then
    if EN = '1' then
      CNTE := '1';
    else
      CNTE := '0';
    end if;
  end if;
  if (CLK'EVENT and CLK =  '1' and CNTE = '1') then
    if (CONSIG(2)='1') then
      CNT <= INC(CNT);
    elsif (CONSIG(3)='1') then
      CNT <= DEC(CNT);
    end if;
  end if;
 end process CTR;
--
LIMIT_CHK: process(CNT,ENIT)
 begin
  if ENIT'EVENT then
    if ENIT = '1' then
      EN <= '1' after COMPDEL;
      RENIT <= '1' after RENITDEL;
    else
      RENIT <= '0' after RENITDEL;
    end if;
  elsif ((EN = '1') and (CNT = LIM)) then
    EN <= '0' after COMPDEL;
  end if;
end process LIMIT_CHK;
COUT <= CNT;
end PROCESS_IMPL;
```

图5-15 控制计数器的行为域算法描述（续）

图5-14和图5-15的描述把时钟产生器和控制计数器定义为一般实体。为了实现图5-12的完整系统，必须定义这两个设计实体的连接。图5-16完成了连接的定义。首先，定义一个实体来表示整个系统。在图5-12里这个实体位于虚线框内。端口语句为实体定义了穿过系统边界的信号。信号的名称在系统边界和系统内部可能不同。模型的结构体是结构化的，首先声明了两个元件和一个信号。在关键字begin和end之间，两个元件被实例化；即使用了类属映射和端口映射将类属参数及连接信号赋值到正确的输入。

因此，我们已经有了一个完整的系统模型。在下一节，将讨论系统模型的一些细节问题。

```
use work.all;
entity COUNT_SYS is
  port(STRT,STROBE: in BIT;
       CON: in BIT_VECTOR(0 to 1);
       DATA_BUS: in BIT_VECTOR(0 to 3);
       CNT: out BIT_VECTOR(0 to 3));
end COUNT_SYS;
--
architecture TWO_COMPONENT of COUNT_SYS is
  signal CLK: BIT;
  component CLOCK_GEN
    generic(PER: TIME);
    port (RUN: in BIT; CLK: out BIT);
  end component;
  component CON_CTR
    generic(SUT,MPW,ENCDEL,ENITDEL,RENITDEL,
            CLRDEL,CNTDEL,LIMDEL,COMPDEL:TIME);
    port (CLK,STRB: in BIT;
          CON: in BIT_VECTOR(0 to 1);
          DATA: in BIT_VECTOR(0 to 3);
          COUT: out BIT_VECTOR(0 to 3));
  end component;
  for CLKGEN: CLOCK_GEN use entity CLOCK_GENERATOR(FEEDBACK);
  for CTR: CON_CTR use entity CONTROLLED_CTR (PROCESS_IMPL);
begin
  CLKGEN: CLOCK_GEN
    generic map(100 ns)
    port map(STRT,CLK);
  CTR: CON_CTR
    generic map(20 ns, 30 ns, 25 ns, 11 ns, 11 ns, 10 ns,
                15 ns,12 ns,10 ns)
    port map(CLK,STROBE,CON,DATA_BUS,CNT);
end TWO_COMPONENT;
```

图5-16　计数器系统的结构化构造体

5.3　系统算法建模

本节讨论系统的算法建模。主要包括以下方面：

1) 多值逻辑系统。

2) 多路复用。

3) 互连模块通信协议。

5.3.1　多值逻辑系统

通常意义上，模型主要是为二值逻辑而建。然而，二值逻辑并不能完全表示出实际的逻辑门所表现的行为。系统建模时反映了这种情况。例如，考虑图5-17中的系统总线。总线的驱动器是DR1和DR2。对于给定的总线驱动器，它本身的值D_i在E_i= ‘1’ 的条件下传送到总线上。首先，假设所有的总线驱动器都是关闭的。这时，所有的驱动器都处于高阻状态；实际上，整个总线“飘浮”在高阻状态。设计中的高阻状态用“Z”表示。第二，假设DR1和DR2都被使能(E1=E2= ‘1’)，而且DR1= ‘0’、DR2= ‘1’。这些恐怕并不是“正常的条件”，但是既然该模型的主要应用是为了检测错误，这种情况当然也需要考虑。总线的值将是哪一

个？根据不同的驱动器电路设计以及采取方法的不同，可以认为是'0'而不是'1'，或者结论相反。但是普遍认为它的输出是未知的，设计时将这种未知状态用“X”表示。

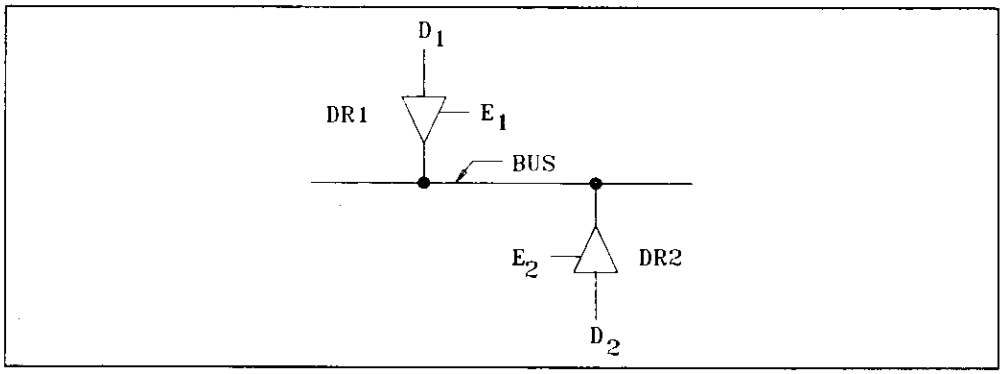


图5-17 系统总线

因此，为系统建立的算法模型使用了四值逻辑系统。它包括值'0'、'1'、“Z”及“X”。这四种值在此抽象级上对于很多种模型的表示是足够的。后面将会把集合扩展到七级，然后使用九级来精化模型，它可以更精确地为具体的电路效应建模。

1. 逻辑包SYSTEM_4

现在给出一个四值逻辑系统的包。本节将讨论这个包的重要特征。完整的SYSTEM_4包将在CD-ROM中给出。首先定义基本标量和向量类型：

```
type MVL4 is ('X','0','1','Z');
type MVL4_VECTOR is array(NATURAL range <>) of MVL4;
```

注意 在定义的MVL4中“X”是类型最左边的值，所以这种类型的所有信号和变量都会被初始化为省缺的未知状态。

四值系统的逻辑和判决函数存储在表中。这使得函数易于理解，更重要的是，函数的实现可以有效地仿真运行。为了实现这些表，需要如下类型定义：

```
  -- one dimensional array
type MVL4_TAB1D is array (MVL4) of MVL4;
  -- two dimensional array
type MVL4_TABLE is array (MVL4, MVL4) of MVL4;
  --three dimensional array
type MVL4_TAB3D is array (MVL4, MVL4, MVL4) of MVL4;
```

这种数组的索引值是类型MVL4。

2. 判决函数

第3章已经指出，当信号被不止一个进程驱动时，需要使用一个判决函数。这种函数是VHDL中多路复用的一种基本方法。一个判决函数接收由驱动信号组成的输入数组，产生一个标量输出代表判决值。这里判决函数的描述用二维表实现，它代表一对驱动信号。驱动数组被扫描，数组中每个相邻的驱动信号由表函数进行比较。

现在考虑六种判决函数的真值表：

1) WiredAnd函数：'0'占支配地位，及两个'Z'的判决结果为'Z'，模拟一条飘浮总线。

```
-- truth table for "WiredAnd" function
constant table_WIREDAND: MVL4_TABLE :=
```

```
--  ------------------------------
--  |  X    0    1    Z  |
--  ------------------------------
    (('X', '0', 'X', 'X'),  --  | X |
     ('0', '0', '0', '0'),  --  | 0 |
     ('X', '0', '1', '1'),  --  | 1 |
     ('X', '0', '1', 'Z')); --  | Z |
```

2) WiredOr函数：'1'占支配地位，两个'Z'的判决结果为'Z'，模拟一条飘浮总线。

```
-- truth table for "WiredOr" function
constant table_WIREDOR: MVL4_TABLE :=
--  ------------------------------
--  |  X    0    1    Z  |
--  ------------------------------
    (('X', 'X', '1', 'X'),  --  | X |
     ('X', '0', '1', '0'),  --  | 0 |
     ('1', '1', '1', '1'),  --  | 1 |
     ('X', '0', '1', 'Z')); --  | Z |
```

3) WiredPullup函数：'0'占支配地位，两个'Z'的判决结果为'1'，模拟一条带有上拉电阻的总线。

```
-- truth table for "WiredPullUp" function
constant table_WIREDPULLUP: MVL4_TABLE :=
--  ------------------------------
--  |  X    0    1    Z  |
--  ------------------------------
    (('X', '0', 'X', 'X'),  --  | X |
     ('0', '0', '0', '0'),  --  | 0 |
     ('X', '0', '1', '1'),  --  | 1 |
     ('X', '0', '1', '1')); --  | Z |
```

4) WiredPulldown函数：'1'占支配地位，两个'Z'的判决结果为'0'，模拟一条带有下拉电阻的总线。

```
-- truth table for "WiredPullDown" function
constant table_WIREDPULLDOWN: MVL4_TABLE :=
--  ------------------------------
--  |  X    0    1    Z  |
--  ------------------------------
    (('X', 'X', '1', 'X'),  --  | X |
     ('X', '0', '1', '0'),  --  | 0 |
     ('1', '1', '1', '1'),  --  | 1 |
     ('X', '0', '1', '0')); --  | Z |
```

5) WiredX函数：检测到一个'0'和一个'1'，结果为X。两个Z的判决结果为'Z'，模拟一条飘浮总线。

```
-- truth table for "WiredX" function
constant table_WIREDX: MVL4_TABLE :=
--  ------------------------------
--  |  X    0    1    Z  |
--  ------------------------------
    (('X', 'X', 'X', 'X'),  --  | X |
     ('X', '0', 'X', '0'),  --  | 0 |
     ('X', 'X', '1', '1'),  --  | 1 |
     ('X', '0', '1', 'Z')); --  | Z |
```

6) WiredOne函数：假设只有一个驱动器是活跃的，它等于'0'或'1'。两个'Z'的判决结果还为'Z'，模拟一条飘浮总线。

```
-- truth table for "WiredOne" function
-- this truth table allows only one driver to be active
constant table_WIREDONE: MVL4_TABLE :=
--  ------------------------------
--  |  X    0    1    Z  |
--  ------------------------------
    (('X', 'X', 'X', 'X'),  --  | X |
     ('X', 'X', 'X', '0'),  --  | 0 |
     ('X', 'X', 'X', '1'),  --  | 1 |
     ('X', '0', '1', 'Z')); --  | Z |
```

这些表都可以在函数体内进行访问，例如，函数WiredX的实现如下：

```
function WiredX (V: MVL4_VECTOR) return MVL4 is
  variable result: MVL4 := 'Z';
begin
  for i in V'range loop
    result := table_WIREDX(result, V(i));
    exit when result = 'X';
  end loop;
  return result;
end WiredX;
```

注意 这里没有驱动器的优先级，假设同一时间比较两个驱动器与比较所有几个驱动器是等价的。上面语句的第一段表示table_WIREDX定义了一个交换函数；第二部分表示表函数必须是一个关联函数。

其他函数的实现与此类似，可参见CD-ROM。函数WiredOne是一个例外。尽管它可以通过访问真值表实现，但这个表并不真正需要，该函数可以如下实现：

```
function WiredOne (V: MVL4_VECTOR) return MVL4 is
  variable result: MVL4 := 'Z';
  variable got_one: BOOLEAN := FALSE;
begin
  for i in V'range loop
    next when V(i) = 'Z';
    if got_one then
      assert false
      report "Multiple contributors to WiredSingle node."
      severity WARNING;
      result := 'X';
      return result;
    end if;
    got_one := TRUE;
    result := V(i);
  end loop;
  return result;
end WiredOne;
```

给出如下函数的定义：

```
function WiredPullUp (V: MVL4_VECTOR) return MVL4;
function WiredPullDown (V: MVL4_VECTOR) return MVL4;
function WiredX (V: MVL4_VECTOR) return MVL4;
function WiredOne (V: MVL4_VECTOR) return MVL4;
function WiredAnd (V: MVL4_VECTOR) return MVL4;
function WiredOr (V: MVL4_VECTOR) return MVL4;
```

就可以为如下wired标量信号定义子类型：

```
subtype DotPU is WiredPullUp MVL4;
subtype DotPD is WiredPullDown MVL4;
subtype DotX is WiredX MVL4;
subtype Dot1 is WiredOne MVL4;
subtype DotAnd is WiredAnd MVL4;
subtype DotOr is WiredOr MVL4;
```

信号可以被声明为这些类型，它将自动继承相应总线的判决功能。最后定义被判决总线的类型，这些类型同样具有相关联总线的判决功能。

```
type BusPU is array (Natural range <>) of DotPU;
type BusPD is array (Natural range <>) of DotPD;
```

```
type BusX is array (Natural range <>) of DotX;
type Bus1 is array (Natural range <>) of Dot1;
type BusAnd is array (Natural range <>) of DotAnd;
type BusOr is array (Natural range <>) of DotOr;
```

3. 感应及驱动函数

我们已经定义了有线信号类型，它们将用于表示互连设备的总线信号。然而，由于逻辑操作必须在四值系统里重新定义，使用类型MVL4及MVL4_VECTOR来表示内部信号会更加有效。这样，只需要重新定义一次即可。包集合SYSTEM_4包括类型转换函数，它完成总线类型和MVL4_VOECTOR类型的相互转换。例如，感应函数从类型BusX转换为类型MVL4_VECTOR如下：

```
function Sense (V: BusX; vZ: MVL4) return MVL4_VECTOR is
   alias Value: BusX (V'length-1 downto 0) is V;
   variable Result: MVL4_VECTOR (V'length-1 downto 0);
begin
   for i in Value'range loop
      if ( Value(i) = 'Z' ) then
         Result(i) := vZ;
      else
        Result(i) := Value(i);
      end if;
   end loop;
   return Result;
end Sense;
```

函数输入vZ允许用户选择在输入为'Z'时的函数值，该值依赖于实现技术。对于TTL电路，可以把状态'Z'看作是状态'1'。

```
function Drive (V: MVL4_VECTOR) return BusX is
begin
  return BusX(V);
end Drive;
```

这个函数使用了紧密关联类型的概念来完成转换。由于DotX是MVL4的子类型，因而没有必要在类型MVL4和DotX之间转换。但是，类型DotX并不是MVL4_VECTOR的子类型，因此这两者之间需要转换。

驱动和感应函数对应于硬件上的驱动和接收线路。包集合SYSTEM_4具有以上所述的不同有线信号类型的感应及驱动函数。

4. 具有更多值的逻辑系统

在四值逻辑系统MVL4里，使用值'X'的目的是为了表示总线竞争。再考虑图5-18，如果总线驱动器1的输出为'0'，而且驱动器2的输出为'1'，总线上的值则是未知的'X'。之所以使用这个值，是因为我们无法知道这两个输出孰强孰弱，也不知道最后取定哪一个值。使用'X'来表示这种情况比较悲观。很多情况下，逻辑0和逻辑1的阻抗不同，如果出现了两种值，总会取其中的一个。考虑图5-18的例子，强信号'1'的源阻抗为100欧姆，弱信号'0'的源阻抗为1000欧姆。如果有这两个信号，总线的结果电压VB是4.54伏特，接收电路把它当作逻辑'1'。由于逻辑系统中总线的问题相当普遍，能否表示信号强度也显得比较重要。通过消除未知信号值（'X'）可以产生更精确的仿真。

一个基本问题是，这里究竟需要使用多少个强度值？在使用SPICE建立电路级模型时，可能使用无限个强度。对于数字建模，显然不需要这么多。我们首先关注具有两个强度级别的

系统：强和弱。在七值逻辑系统MVL7里使用了这些强度级，该系统由ZYCAD（现在的Synopsys）演化而成，并逐渐成为IEEE标准的候送者。

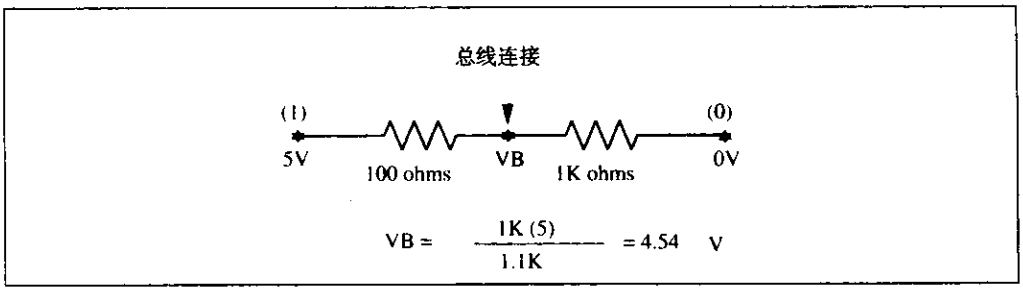


图5-18 强1与弱0

5. MVL7系统

系统MVL7是一个具有七种值的逻辑系统，它把三种状态与两个强度组合起来。三种状态分别为0、1和X，它们被用来表示系统的内部信息。具有强度的值组合状态表示设备接口的信息。MVL7使用两种强度：强和弱。MVL7系统的基本类型定义如下：

```
type MVL7 is ('X', -- strong X (strong unknown)
              '0', -- strong 0 (strong low)
              '1', -- strong 1 (strong high)
              'Z', -- tri-state X (high impedance)
              'W', -- weak X (weak unknown)
              'L', -- weak 0 (weak low)
              'H');-- weak 1 (weak high)
```

三种状态和两种强度组合为6种值。但是MVL7也包含有第三种强度Z。Z的源阻抗要远远大于弱强度。如果采用严格的强度和状态的概念，它增加了三种值：Z0、Z1和ZX。但是这个阻抗强到可以使该状态很难被识别出，因此认为Z0=Z1=ZX=Z。这样就形成了一个七值系统。

6. IEEE 9值系统

IEEE增加了另外两个值，'U'和'-'。值'U'表示没有初始化的值，它表示设备打开电源后的状态。值'-'表示传统的不考虑值，它在综合时辅助组合逻辑的最小化。九值逻辑系统成为基本IEEE 1164标准，被压缩在包STD_LOGIC_1164中。

```
type STD_ULOGIC is ('U','X','0','1','Z','W','L','H','-');
type STD_ULOGIC_VECTOR is array(NATURAL range <>)
     of STD_ULOGIC;
type STDLOGIC_TABLE is array (STD_ULOGIC, STD_ULOGIC)
     of STD_ULOGIC;
```

ULOGIC中的'U'表示未判决。判决函数的定义如下：

常数RESOLUTION_TABLE: STDLOGIC_TABLE := (

```
constant RESOLUTION_TABLE: STDLOGIC_TABLE := (
-------------------------------------------------------------
--|  U    X    0    1    Z    W    L    H    -        |    |
-------------------------------------------------------------
  ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), --  | U  |
  ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), --  | X  |
  ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), --  | 0  |
  ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), --  | 1  |
  ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), --  | Z  |
```

```
( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), --  | W |
( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), --  | L |
( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), --  | H |
( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  --  | - |
                                                    );
```

该表封装了值的相对强度，即强信号支配弱信号。如果R是由这个表实现的函数，则：

0 R H = H R 0 = 0 R L = L R 0 = 0
1 R L = L R 1 = 1 R H = H R 1 = 1

X同样也表示值为0或1；W表示值为L或H。因此，X和W都是竞争总线时产生的未知值：

0 R 1 = 1 R 0 = X
L R H = H R L = W

X也可由错误条件产生，如触发器进入未知状态。确定值和未知值里应用了强度：

L R X = X，1 R W = 1.

Z表示高阻状态。未初始化状态U在类型值的最左端，它是省缺的初始值。可以测试逻辑系统是否正确启动，在仿真过程中U状态将会变为0或1。值'-'表示综合阶段的不考虑值，它在仿真时当做状态X考虑。

判决函数如下：

```
function RESOLVED (S: STD_ULOGIC_VECTOR)
         return STD_ULOGIC is
  variable RESULT : STD_ULOGIC := 'Z'
begin
  if (S'LENGTH = 1) then
    return S(S'LOW);
  else
    for I in S'RANGE loop
      RESULT := RESOLUTION_TABLE(RESULT, S(I));
    end loop;
  end if;
  return RESULT;
end RESOLVED;
```

给出了判决函数RESOLVED之后，可以作如下声明：

```
subtype STD_LOGIC is RESOLVED STD_ULOGIC;
type STD_LOGIC_VECTOR is array(NATURAL range <>)
   of STD_LOGIC;
```

这些子类型和类型在大多数组织里都被使用，特别是为综合建模时。但是，对已判决的和未判决的类型，布尔操作是重载的。STD_LOGIC_MISC包里还有SENSE和DRIVE函数，它们完成对类型STD_ULOGIC_VECTOR和STD_LOGIC_VECTOR的相互转换，这种转换可能在一些不常出现的情况下被用到。

5.3.2 综合性的系统实例

现在举一个使用了SYSTEM_4的综合实例来说明在系统模型中多值逻辑的使用。

图5-19是一个包括缓冲寄存器、RAM及ADD和STORE单元的三模块系统。系统的行为序列如下：

1) 当STRB变高时，数据（DI）写入缓冲寄存器，可用信号（DAV）把数据置为'1'。

2) ADD和STORE单元通过将使能信号EN置为'1'来响应事件DAV='1'。它从缓冲寄存器

中读出数据到数据总线DATA_BUS上并重置DAV。

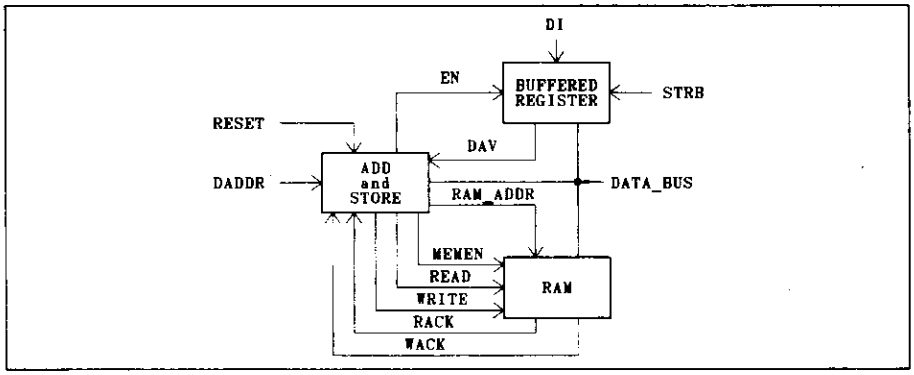


图5-19 三模块系统

3) ADD和STORE单元从数据总线DATA_BUS上读取数据并把它存到内部寄存器中。

4) ADD和STORE单元判断出一个读信号（READ）后根据地址RAM_ADDR（实现先前被置为DADDR的值）完成一个RAM读操作。RACK='1'表示读操作的完成并把信号READ置为低。信号READ和RACK通常被看作是握手信号。

5) ADD和STORE单元把总线上读的存储器数据复制下来，与内部寄存器中的数据相加，并把结果返回到内部寄存器中。

6) ADD和STORE单元进行一个RAM写操作，该操作把写信号WRITE变高并将内部寄存器的值写到RAM_ADDR指定的RAM地址单元里。WACK='1'表示写过程已经完成并且信号WRITE被置为低。WRITE和WACK信号也是一对握手信号。

7) ADD和STORE单元返回到初始状态，等待下一个从缓冲寄存器来的数据可用信号(DAV)。

由于数据总线被这3个模块分别驱动，因此它需要一个判决函数。我们将使用WiredOne函数，该函数需要使用BUS1总线类型。

图5-20、图5-21及图5-22分别是缓冲寄存器、RAM及ADD_STORE的算法模型。寄存器模型是自解释的；但是需要注意到函数DRIVE被用于将内部类型MVL4_VECTOR转换为BUS1类型。RAM模型也是顺序的。它响应信号CS、RD或WRITE来完成读或写操作。对于WRITE，函数SENSE把类型BUS1转换为MVL4_VECTOR。这里错误地把所有的'Z'变为'1'。READ使用了函数DRIVE把类型MVL4_VECTOR转换为BUS1类型。应答信号RACK及WACK在ACK_PW期间一直是高电平。这里使用了一个集合把存储器初始化为全'0'。

ADD_STORE是一个单元，由它来控制活动序列。它通过一个控制步骤序列，执行一个特定的活动集合。图5-22是ADD & STORE的算法模型。控制步骤被标注为CS0到CS6。每一步由wait语句所结束的VHDL语句序列组成。Wait语句有如下格式：

```
wait on signal until signal = value;
wait for time;
```

wait语句导致一个进程被挂起（参见第3章）。对于第一种格式，当wait语句里的信号取到合适值的时候，进程恢复执行。这种格式的wait语句可以用于检测握手信号的模型，该信号可以用来推进控制单元的下一个状态。ADD & STORE模型有三个地方使用了这种格式的wait

语句，它们检测三个信号的发生：DAV、RACK及WACK。第二种格式的wait语句用于在下一步开始前具有混合延时时间CLK_PER的模型中。顾名思义，这段时间对应于控制单元推进的时钟周期。这个延时明显长于由控制步骤触发的器件在下一个步骤发生之前的响应时间。第二种格式的模型模拟了一种开放端口通信机制。

```
use work.SYSTEM_4.all;
entity BUFF_REG is
  generic(STRB_DEL,DAV_DEL,ODEL: TIME);
  port(DI:  in MVL4_VECTOR(7 downto 0);
       STRB,EN: in MVL4;
       DAV: out MVL4;
       DO: out BUS1(7 downto 0):="ZZZZZZZZ");
end BUFF_REG;
--
architecture TWO_PROC of BUFF_REG is
  signal REG: MVL4_VECTOR(7 downto 0);
begin
  FRONT_END: process(STRB,EN)
  begin
    if STRB'EVENT and STRB = '1' then
      REG <=DI after STRB_DEL;
      DAV <= '1' after DAV_DEL;
    end if;
    if EN'EVENT and EN='1' then
      DAV <= '0' after DAV_DEL;
    end if;
  end process FRONT_END;
  --
  OUTPUT: process(REG,EN)
  begin
    if (EN = '1') then
      DO <= DRIVE(REG) after ODEL; else
      DO <= "ZZZZZZZZ" after ODEL;
    end if;
  end process OUTPUT;
end TWO_PROC;
```

图5-20　缓冲寄存器模型

需要指出的是，这3种模型的输出都与信号DATA_BUS相连，它们的初始状态都是'ZZZZZZZZ'。而类型为MVL4的信号的省缺初始化值是'X'，所以信号DATA_BUS的所有驱动器在没有其他定义的情况下接收到的初始值为'XXXXXXXX'。为了让总线可以正常工作，所有的驱动器都要求在非激活时的值必须是'Z'。因此，它们必须初始化为这个值。此外，当特定输出与总线信号的连接中断时，该驱动器必须返回到全'Z'状态。这三种模型都必须满足这一要求。

1. 控制步骤的硬件解释

在ADD & STORE的算法模型里，使用了包括wait语句的控制序列。我们的目标是生成对应于实际硬件的算法描述。因此，需要弄清楚控制序列的硬件解释。为了达到这个目的，首先要实现ADD_STORE的控制单元/数据单元部分（control unit/data unit partition）。图5-23表示了该部分。控制单元部分包括一个控制单元和一个时钟产生器。数据单元部分包括ADU及加法部件。

```
use work.SYSTEM_4.all;
--
entity RAM is
  generic(RDEL,DISDEL,ACK_DEL,ACK_PW: TIME);
   port(DATA: inout BUS1(7 downto 0):="ZZZZZZZZ";
        ADDR: in MVL4_VECTOR(4 downto 0);
        RD,WRITE,CS: in MVL4;
        RACK,WACK: out MVL4);
end RAM;
--
architecture SIMPLE of RAM   is
begin
 MEM: process (CS,RD,WRITE)
  type MEMORY is array(0 to 31) of MVL4_VECTOR(7 downto 0);
  variable MEM: MEMORY:= (others => (others  => '0'));
 begin
   if CS = '1' then
     if RD = '1' then
       DATA <= DRIVE(MEM(INTVAL(ADDR))) after RDEL;
       RACK <= '1' after ACK_DEL,
               '0' after ACK_DEL + ACK_PW;
     elsif WRITE = '1' then
       MEM (INTVAL(ADDR)):= SENSE(DATA,'1');
       WACK <= '1' after ACK_DEL, '0' after ACK_DEL+ACK_PW;
     end if;
   else
     DATA <= "ZZZZZZZZ"  after DISDEL;
   end if;
 end process MEM;
end SIMPLE;
```

图5-21 简单RAM模型

首先考虑时钟产生器和ADU的算法模型，以理解CU部件如何与它们交互（图5-24和图5-25）。时钟产生器模型由RESET下降沿启动。它把变量CLKE置为'1'并产生第一个时钟周期。然后，时钟产生器自动触发产生周期，直到信号RESET变为高电平，此时CLKE被置为'0'。当前时钟周期完成，时钟产生器在RESET变低之前始终保持为高电平。

图5-25所示的加法部件（ADU）响应信号CS2、CS3、CS4和CS5的变化，操作分别为：1) 寄存器载入数据；2) 执行加法；3) 输出存储器地址；4) 数据值输出到数据总线。

图5-26是控制部件结构化模型的示意图。CU接收输入RESET、CLK、DAV、RACK及WACK。CA1、CS2、CS3及CS5分别连到EN、READ及WRITE的接口控制信号。每个控制状态（CSi）匹配一个触发器。所有的触发器由信号CLK定时，所以控制部件是同步的。控制部件的行为类似于一个移位寄存器，它在CS0到CS6之间传递'1'。然而，从一种状态到另一种状态有三种推进方式：

1) 自动推进。控制状态CSi总是推进到状态(CSi+1)。例如，CS1是触发器C2的D输入端。CS1只持续一个时钟周期。假设外部行为由可以在一个时钟周期内完成的控制状态定时。这种情况下，CS1='1'导致EN='1'，它把数据发送到连接ADD和STORE单元中内部寄存器输入端的数据总线上。在CS2上升时，数据写入寄存器。这种数据传输操作必须在一个时钟周期内完成。可以通过对时钟频率的选择确保有足够的时间来完成该操作。

2) 握手推进。CS0和CS1的推进就属于这种推进。控制单元保持在CS0直到DAV='1'。在

下一时钟周期推进到CS1。该事件的发生对信号DAV的时钟加以限制。该信号只有在控制单元推进到下一状态CS1时才可以释放当前值。这可以通过建立控制单元和设备之间的握手协议来完成。这种情况下，设备被控制的部分是缓冲寄存器。协议过程如下：缓冲寄存器置位DAV='1'，保持此值直到控制单元推进到CS1并置EN='1'。当EN='1'时，缓冲寄存器释放DAV的值使之为'0'。

```
use work.SYSTEM_4.all;
entity ADD_STORE is
  generic(CON_DEL, DO_DEL, MA_DEL, DIS_DEL, CLK_PER: TIME);
  port(RESET,DAV,RACK,WACK: in MVL4;
       MEMEN: out MVL4;
       EN,READ,WRITE: inout MVL4;
       DATA: inout BUS1(7 downto 0):="ZZZZZZZZ";
       DADDR: in MVL4_VECTOR(4 downto 0);
       MADDR: out MVL4_VECTOR(4 downto 0));
end ADD_STORE;
architecture ALG of ADD_STORE is
begin
 CON: process
   variable DATA_REG: MVL4_VECTOR(7 downto 0);
 begin
   if RESET = '1' then
     DATA <= "ZZZZZZZZ"after DIS_DEL; --CS0
   end if;
   wait on DAV until DAV = '1';
-----------------------------------------
     EN <= '1' after CON_DEL;       --CS1
   wait for CLK_PER;
-----------------------------------------
     EN <= '0' after CON_DEL;
     DATA_REG := SENSE(DATA,'1');  --CS2
   wait for CLK_PER;
-----------------------------------------
     MADDR <=  DADDR after MA_DEL;
     MEMEN <= '1' after CON_DEL;   --CS3
     READ <= '1' after CON_DEL;
   wait on RACK until RACK ='1';
-----------------------------------------
     DATA_REG := ADD8(SENSE(DATA,'1'),DATA_REG);
     READ <= '0'after CON_DEL;
     MEMEN <= '0'after CON_DEL;    --CS4
   wait for CLK_PER;
-----------------------------------------
     DATA <= DRIVE(DATA_REG) after DO_DEL;
     WRITE <= '1'after CON_DEL;
     MEMEN <= '1'after CON_DEL;    --CS5
   wait on WACK until WACK ='1';
-----------------------------------------
     WRITE <= '0'after CON_DEL;
     MEMEN <= '0'after CON_DEL;    --CS6
     DATA <= "ZZZZZZZZ" after DIS_DEL;
   wait for CLK_PER;
 end process CON;
end ALG;
```

图5-22　Add & Store模型

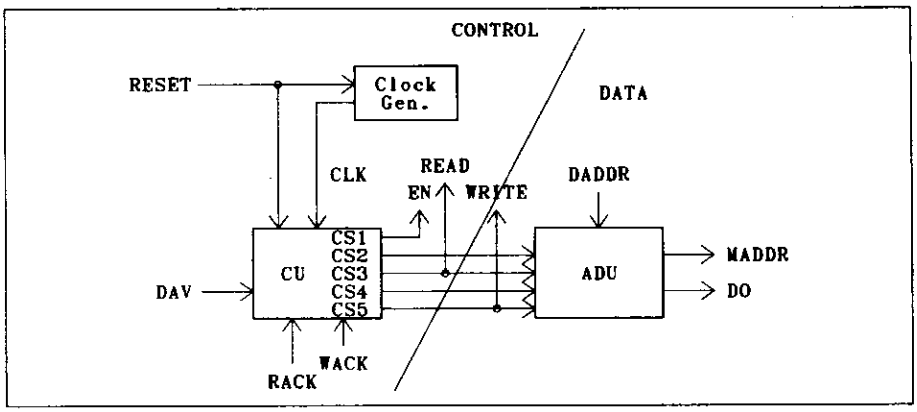


图5-23 ADD & STORE的控制/数据划分

```
use work.SYSTEM_4.all;
entity CLOCK_GENERATOR is
  generic(PER: TIME);
  port(RESET: in MVL4; CLK: out MVL4);
end CLOCK_GENERATOR;

architecture IMPL_1 of CLOCK_GENERATOR is
  signal CLOCK: MVL4;
begin
  process (RESET,CLOCK)
    variable CLKE: BIT := '0';
  begin
    if RESET='0' and not RESET'STABLE then
      CLKE := '1';
      CLOCK <= transport '0' after PER/2;
      CLOCK <= transport '1' after PER;
    end if;
    if RESET='1' and not RESET'STABLE then
      CLKE := '0';
    end if;
    if CLOCK='1' and not CLOCK'STABLE and CLKE = '1'then
      CLOCK <= transport '0' after PER/2;
      CLOCK <= transport '1' after PER;
    end if;
      CLK <= CLOCK;
  end process;
end IMPL_1;
```

图5-24 时钟产生器模型

3) 异步推进。从CS3到CS4属于这种推进。CS3变为1导致存储器进行读操作。RAM的RACK信号表示读操作完成。但是RACK保持为高电平的时间要少于一个时钟周期。因此，RACK从0到1的变化把RK触发器置为1状态。在下一个CLK的上升沿，CS4被置位，它使用异步输入R把触发器C10复位。因此，这种机制提供了一种根据期望把控制单元的异步响应推进到下一状态的方法。

2. 使用wait语句的算法模型的精确性

从ADD & STORE的算法模型里可以发现两个明显的不足：

1) 算法模型（图5-22）只在控制步骤CS0里采样复位信号，而在控制部件的结构化模型里

（图5-26），该单元在任何时候都可以响应复位信号。

```
use work.SYSTEM_4.all;
entity ADU is
  generic(DO_DEL,MA_DEL: TIME);
  port(CS2,CS3,CS4,CS5: MVL4;
       DATA: inout BUS1(7 downto 0) := "ZZZZZZZZ";
       DADDR: in MVL4_VECTOR(4 downto 0);
       MADDR: out MVL4_VECTOR(4 downto 0));
end ADU;
architecture BEHAVIOR of ADU is
begin
 DU: process(CS2,CS3,CS4,CS5)
   variable DATA_REG : MVL4_VECTOR(7 downto 0);
 begin
   if CS2'EVENT or CS4'EVENT then
     if CS2 = '1' then
       DATA_REG:= SENSE(DATA,'1');
     end if;
     if CS4 = '1' then
       DATA_REG:= ADD8(SENSE(DATA,'1'),DATA_REG);
     end if;
   end if;
   if CS3'EVENT and CS3 = '1' then
     MADDR <= DADDR after MA_DEL;
   end if;
   if CS5'EVENT then
     if CS5 = '1' then
       DATA <= DRIVE(DATA_REG) after DO_DEL;
     else
       DATA <= "ZZZZZZZZ";
     end if;
   end if;
 end process DU;
end BEHAVIOR;
```

图5-25　ADD部件模型

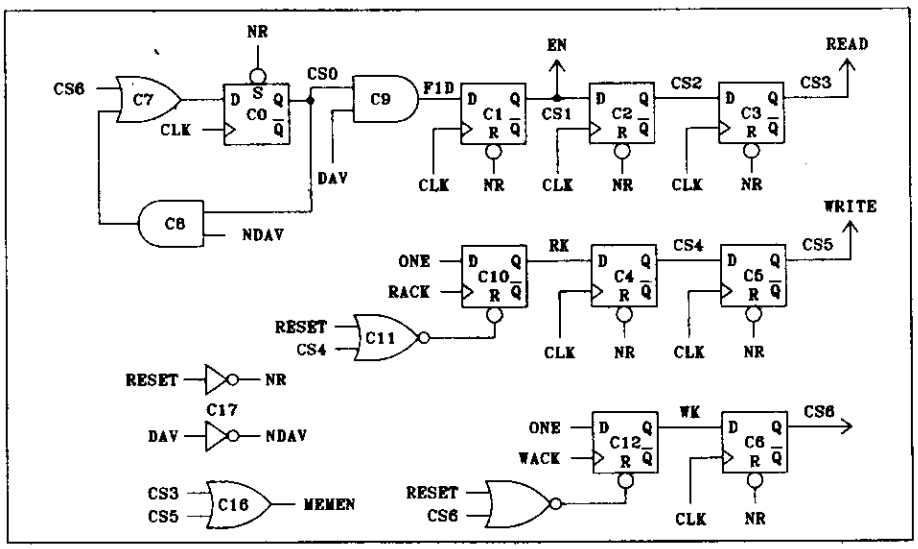


图5-26　控制部件结构化模型

2) ADD & STORE的控制序列是一个简单的循环。当控制序列里包括状态条件分支时，并不能判断是否可以使用这种方法。

3. 复位

一种解决复位问题的方法是并不把它当成一个问题。它把带有wait语句的算法模型看作是一种高级模型，而同时还包含具体的复位逻辑则是一种较低级别的设计活动，最好是在门级完成。然而，它意味着算法级和门级模型的响应不同，这是我们所不需要的。

图5-27和图5-28是一种较好的方法。实体WAIT_STEPS里的结构ONE是允许重置为任何状态的控制单元模型。它通过在循环里插入控制序列完成重置功能。控制序列的每一步都使用了wait语句和next语句来检测重置的发生。重置发生时，程序转到循环标记，也就是循环体的开始。由于循环条件现在是FALSE（因为R='1'），循环将被终止。而循环语句是进程的最后一条语句，进程将从头开始执行，即从关键字begin后的第一条wait语句处开始执行。程序将在进程初始处停止，直到重置信号变低。所以，从仿真一开始，模型在结构ONE里等待RUN='1'。在RUN变为'1'后，单元循环经过状态0、1、2，直到R='1'时退出循环，然后等待R再次变为'0'。开始时的wait语句是为了避免当R='1'时进入无限循环过程，无限循环会导致仿真时间无法向前推进。

4. 控制序列分支

似乎控制序列的实现可以通过把每个状态用一个带有标签的简单循环表示，然后使用next语句测试条件进入到正确的状态。但是，VHDL中next语句的语义并不允许这样做。next语句只能强制分支进入闭合循环，而不允许进入任意循环。这种限制在结构化设计里是必须的。

控制序列分支的实现可以通过把每个简单（非条件）序列合成为一个进程，然后在进程的最后使用if和case语句决定下一个即将活跃的简单序列。图5-27和图5-28表明了这一点。在实体WAIT_STEPS的进程体TWO中，当RUN变为1时，执行简单序列0、1。然后，如果X='1'，执行序列2、3；如果X='0'，执行序列4、5。这两种情况都返回到序列0、1，触发信号的变化则会激活进程。由于信号SINT由三个进程驱动，所以这里需要一个判决函数。INTRES包提供了这个函数。最后，三个进程的不同位置都有语句SINT <= null;。它表示驱动进程与判决函数分离，以达到SINT驱动信号的时分多路复用。下一节详细讨论这种情况。

总而言之，在综合意义上，对于大多数控制序列是简单的无分支序列的系统，从结构的角度上，可以认为对其使用wait语句的算法模型是有用的。 对于带有分支的更复杂的情况，使用第8章里讨论的状态机模型则更加合适。

5.3.3 时分多路复用

在前一节，使用了总线判决函数来实现分时共享总线。这种方法通过确保总线信号的所有驱动器不活跃时保持在'Z'状态来实现时分多路复用。现在考虑另一种时分多路复用。图5-29使用了两个防护块实现时分多路复用。

在信号PHASE_ONE='1'时，块PH_ONE将信号Z赋值为'0'。在信号PH_TWO ='1'时，块PH_TWO将信号Z赋值为'1'。PH_ONE和PH_TWO的时钟脉冲不可能同时达到'1'。（见图5-30）

信号Z被声明为类型DOTX，因此，它继承了WiredX的判决功能。假设建模者的意图是：
1) 当PHASE_ONE变高时由PH_ONE控制Z，并且在PHASE_TWO变高时由PH_TWO控制Z；
2) 当PH_ONE和PH_TWO都变低时Z保持当前值。但是，图5-30展示了真正的情况。如果Z开

始于'X'状态，它的值将在PHASE_ONE='1'时变为0。在信号PH_TWO ='1'时，期望Z的值变为'1'，而实际上Z返回到值'X'。这是因为块PH_ONE的驱动器包括了一个'0'，而且块PH_TWO的驱动器包括了一个'1'。总线判决函数WiredX判决的结果值是'X'。一个总线判决函数是它自身驱动的静态函数，它不考虑驱动器的值保持了多久。本例中，我们期望驱动器的值经常发生变化以控制信号Z的值。可以调整构架GUARDED_BLOCK的代码来解决这个问题，所以当一个特定块的防护信号变为0时，它的驱动器赋值为'Z'。

```
package INTRES is
  type INTARRAY is array(NATURAL range <>) of INTEGER;
  function INTBUS(S: INTARRAY) return INTEGER;
  subtype RINTEGER is INTBUS INTEGER;
end INTRES;

package body INTRES is
  function INTBUS(S: INTARRAY) return INTEGER is
    begin
      for I in S'RANGE loop
      return S(I);
    end loop;
  end INTBUS;
end INTRES;

use work.INTRES.all;
use work.SYSTEM_4.all;
entity  WAIT_STEPS is
  generic(CLK_DEL,DIS_DEL: TIME);
  port(RUN,X,R: in DOT1 := '0'; S: out RINTEGER);
end WAIT_STEPS;

architecture ONE of WAIT_STEPS is
begin
  process
  begin
    wait until R = '0' and RUN = '1';
LOOP_START: while R = '0'and RUN = '1' loop
       S <= 0;
       next LOOP_START when R = '1';     ---Step 0
    wait until R = '1' for CLK_DEL;
    ----------------------------
       S <= 1;
       next LOOP_START when R = '1';     ---Step 1
    wait until R = '1' for CLK_DEL;
    ----------------------------
       S <= 2;
       next LOOP_START when R = '1';     ---Step 2
    wait until R = '1' for CLK_DEL;
    end loop;
  end process;
end ONE;
```

图5-27　复位（重置）

```
if guard'event and guard = '0' then
  Z <= 'Z';
```

它允许非'Z'的值通过。但是，这里还有一个没有解决的问题：两个防护信号都变低时会

发生什么情况？把两个防护信号都赋值为'Z'，信号Z的结果也是'Z'。因此，这种方法并不会让Z保持在上一个值。实际上，不可能写出包括“过去”概念的判决函数。判决函数只能用来表示组合逻辑电路。

```
architecture TWO of WAIT_STEPS is
 signal TRIGGERB,TRIGGERBA,TRIGGERC,TRIGGERCA: DOT1 := '0';
 signal SINT: RINTEGER register;
begin
A: process
  begin
    SINT <= null;
   wait on RUN,TRIGGERBA,TRIGGERCA until RUN = '1';
    SINT <= 0;                              ---Step 0
   wait for CLK_DEL;
    SINT <= 1;                              ---Step 1
   wait for CLK_DEL;
    SINT <= null;
    if X = '1' then
      TRIGGERB <= not(TRIGGERB);
    else
      TRIGGERC <= not(TRIGGERC);
    end if;
  end process A;
B: process
  begin
    SINT <= null;
   wait on TRIGGERB;
    SINT <= 2;                              ---Step 2
   wait for CLK_DEL;
    SINT <= 3;                              ---Step 3
   wait for CLK_DEL;
    SINT <= null;
    TRIGGERBA <= not(TRIGGERBA);
  end process B;
C: process
  begin
    SINT <= null;
   wait on TRIGGERC;
    SINT <= 4;                              ---Step 4
   wait for CLK_DEL;
    SINT <= 5;                              ---Step 5
   wait for CLK_DEL;
    SINT <= null;
    TRIGGERCA <= not(TRIGGERCA);
  end process C;
  S <= SINT;
end TWO;
```

图5-28 分支

为了解决这种包括防护信号赋值语句的总线判决，VHDL为防护赋值的信号提供了一种特殊的机制。如果建模者希望采用时分多路复用，该信号可以设计“寄存器”或者是“总线”。例如，信号ZINT在图5-31中被设计为一个寄存器，而在图5-32里被设计为一条总线。如果防护信号被设计成一个寄存器或者一条总线，而且如果防护语句驱动该信号以使得块防护变为FALSE，则该语句的驱动器假设连接中止并被总线判决函数所忽略。防护为TRUE的块的值到

达总线判决函数。对于所有块的防护都被关闭的情况，如果ZINT是一个寄存器，它将保持上一个值。如果ZINT是一条总线，它将赋值为总线判决函数的缺省值。对于包SYSTEM_4里的判决函数，当函数没有活跃输入时返回的缺省值为'Z'。信号ZINT在一个delta延时后得到了期望值'Z'。

```
use work.SYSTEM_4.all;
entity TIME_MUX is
  generic(DEL1,DEL2: TIME);
  port(PHASE_ONE,PHASE_TWO: in MVL4;
       Z: out DOTX := '0');
end TIME_MUX;
architecture  GUARDED_BLOCK0 of TIME_MUX is
begin
  PH_ONE: block(PHASE_ONE = '1')
  begin
    Z <= guarded '0' after DEL1;
  end block PH_ONE;
  PH_TWO: block(PHASE_TWO = '1')
  begin
    Z <= guarded '1' after DEL2;
  end block PH_TWO;
end GUARDED_BLOCK0;
```

图5-29　具有两个防护块的时分多路复用器

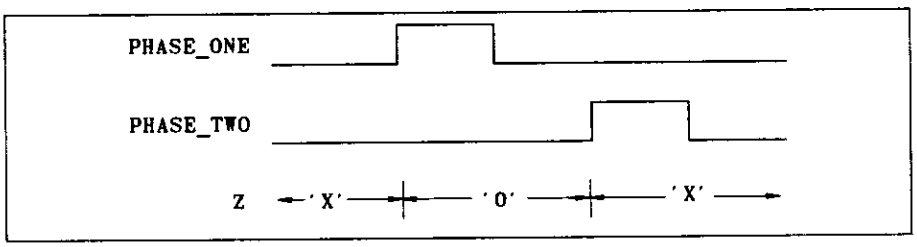


图5-30　两个防护块的时钟

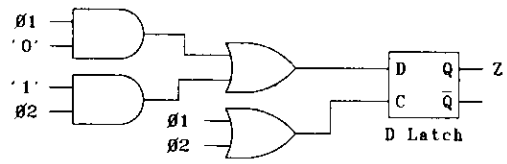
```
architecture  GUARDED_BLOCK1 of TIME_MUX is
  signal ZINT: DOTX register;
begin
  PH_ONE: block(PHASE_ONE = '1')
  begin
    ZINT <= guarded '0' after DEL1;
  end block PH_ONE;
  PH_TWO: block(PHASE_TWO = '1')
  begin
    ZINT <= guarded '1' after DEL2;
  end block PH_TWO;
  Z <= ZINT;
end GUARDED_BLOCK1;
```

图5-31　作为寄存器的ZINT

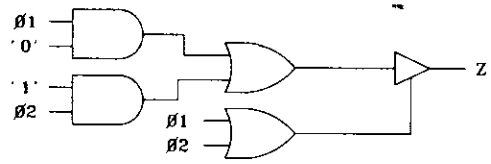
从硬件角度看，若ZINT是一个寄存器，它隐含表示一个多路复用的时钟控制寄存器。如果ZINT是一条总线，它隐含表示一条时钟控制总线。图5-33是这两种情况的等效电路。

```
architecture  GUARDED_BLOCK2 of TIME_MUX is
  signal ZINT: DOTX bus;
begin
  PH_ONE: block(PHASE_ONE = '1')
  begin
    ZINT <= guarded '0' after DEL1;
  end block PH_ONE;
  PH_TWO: block(PHASE_TWO = '1')
  begin
    ZINT <= guarded '1' after DEL2;
  end block PH_TWO;
  Z <= ZINT;
end GUARDED_BLOCK2;
```

图5-32 作为总线的ZINT



a) 多路复用的时钟控制寄存器



b) 多路复用的时钟控制总线

图5-33 两个防护块的等价逻辑

没有防护块的进程里也存在同样的情况。图5-34显示同时驱动信号Z的两个进程有类似结果。Z是两个驱动器的静态函数，它不考虑驱动器的当前值是什么时候产生的。

```
architecture PROC of TIME_MUX is
begin
  PH_ONE: process(PHASE_ONE)
  begin
    if PHASE_ONE = '1' then
      Z <= '0' after DEL1;
    end if;
  end process PH_ONE;
  PH_TWO: process(PHASE_TWO)
  begin
    if PHASE_TWO = '1' then
      Z <= '1' after DEL2;
    end if;
  end process PH_TWO;
end PROC;
```

图5-34 驱动信号Z的两个进程

这个问题的解决类似于防护块。图5-35解释了该问题的解决方法。内部信号也被设计为一个寄存器。在这两个进程中，无论是PHASE_ONE还是PHASE_TWO变为0，ZINT都赋值为'null'，该值是驱动器中止连接到判决函数的结果。同样，如果PHASE_ONE和PHASE_TWO都不同时为'1'，只有活跃（非null）的驱动器的值被通过。由于ZINT被设计为一个寄存器，如果两个驱动器都为null，ZINT将保持在最近的非null值。如果ZINT是一条总线，ZINT在两个驱动器都为null时的值为'Z'。

```
architecture PROC_NULL of TIME_MUX is
  signal ZINT: DOTX register;
begin
  PH_ONE: process(PHASE_ONE)
  begin
    if PHASE_ONE = '1' then
      ZINT <= '0' after DEL1;
    else
      ZINT <= null;
    end if;
  end process PH_ONE;
  PH_TWO: process(PHASE_TWO)
  begin
    if PHASE_TWO = '1' then
      ZINT <= '1' after DEL2;
    else
      ZINT <= null;
    end if;
  end process PH_TWO;
  Z <= ZINT;
end PROC_NULL;
```

图5-35　两个进程驱动单个信号的方法

由于我们希望在算法抽象级给出信号判决的某些语义，所以需要注意这种带有判决函数的时分多路复用。在较低的抽象级设计时分多路复用时，只需设计出完成该功能的实际电路，然后产生一个真实的时分多路复用的VHDL模型。但是，在算法级就使模型具有时分复用的能力比较方便，它不需要显式的硬件多路器。另一方面，如果打算综合含有时分复用的算法描述，综合过程必须能够识别出这种功能。

使用一条为驱动器赋值的IF语句，会使描述变得混乱而冗长。图5-36使用了判决函数分配的方法。它为每个进程使用了独立的信号，然后复用它们（用非SIGNAL'quiet）。这里需要假设两个信号不同时激活。

读者也许认为时分多路复用的问题在VHDL模型里很少出现，但是实际情况并非如此。考虑图5-37所示的一个处理器的算法模型框架。

同样假设E和F是不连续的脉冲，PC是一个未定义信号。这里无法写出一个可以让PC保持在上一个值的判决函数；必须使用null驱动器或'quiet多路器。

综合工具不能够处理高级别的多路器，它无法生成判定信号是否“quiet”的电路。尽管如此，在算法级具有时分复用的方法仍然是非常方便的，它不需要显式的硬件多路器。

```
entity TIME_MUX is
  generic(DEL1,DEL2: TIME);
  port(PHASE_ONE,PHASE_TWO: in MVL4;
       Z: buffer MVL4);
end TIME_MUX;

architecture QUIET_MUX of TIME_MUX is
  signal  PH1,PH2,Z1,Z2: MVL4;
begin
  PH_ONE: process(PHASE_ONE)
  begin
    if PHASE_ONE = '1' then
      Z1 <= '0' after DEL1;
    end if;
  end process PH_ONE;
  PH_TWO: process(PHASE_TWO)
  begin
    if PHASE_TWO = '1' then
      Z2 <= '1' after DEL2;
    end if;
  end process PH_TWO;
  Z <= Z1 when not Z1'quiet else
       Z2 when not Z2'quiet else
       Z;
end QUIET_MUX;
```

图5-36 'Quiet多路器

```
-----Computer Model

FETCH: process
   wait until (F = '1');
   -----
   -----
   ----- Fetch  instruction
   -----
   PC <= INC(PC); -- Increment the Program counter
end process FETCH;
EXECUTE: process
 wait until (E = '1');
 ------
 ------
 case IR(15 downto 12) is
 -----
 -----
 ---- Jump instruction
 when "0011" => PC < IR(11 downto 0);
 ----
 ----
 end process EXECUTE;
```

图5-37 处理器的算法模型框架

习题

5.1 列出在设计过程中算法级VHDL模型的不同使用方法，在每项的下面，用2到3句话予以说明。

5.2 图5-38是具有两个模块的数字系统的框图。该系统的基本功能相当于半个UART，即它并发接收一个4位长的字，并将其串行移出，移出的速率由并行到串行转换器外部的晶体振

荡器控制。当RUN=1时，CLKGEN产生一个周期为50 ns的时钟(CLK)。当RUN=0时，CLK的输出为逻辑0电平，并串转换器的功能如下：当移出信号SHOUT由0变为1时，并行数据被载入到内部寄存器SR，并且BUSY被置为高电平，在BUSY变高后第一个CLK由0变为1时，数据被传输到线SO，并在后面的3个时钟脉冲内完成全部数据的移出。四位移出完成之后，BUSY被重置。假设当BUSY为1时不会载入并行数据，用VHDL为系统建立完整的模型。

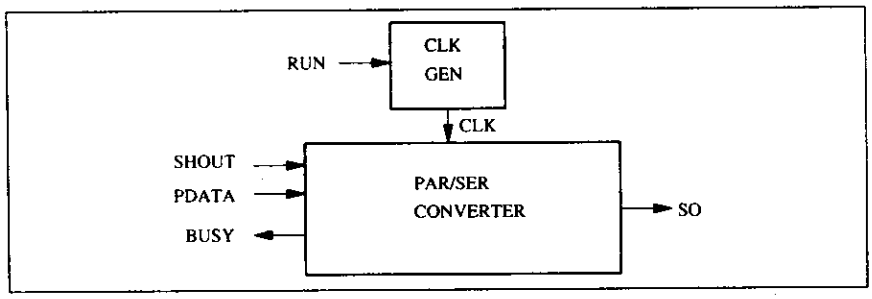


图5-38　并行到串行转换器

5.3　写出一段检测信号X变高和信号Y变低之间的时间间隔是否小于T的VHDL代码。

5.4　图5-39给出了一些互连模块的逻辑结构，对于每个块都给出了一个传输延时值，但没有给出它的功能描述。写出该模型的VHDL描述框架。描述中应包括一个独立的实体。假设W、X和T都是实体接口信号，Q、Y、Z、R和S都是内部信号。假设任一块的输入变化都会导致所有块的输出在一定的传输延时之后发生变化。根据上述信息包含尽可能多的细节。注意：题中的“块”指的是一个逻辑物理块，并不一定是一个VHDL块。

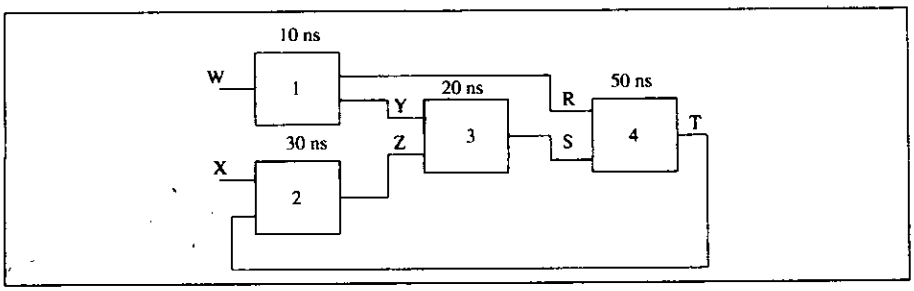


图5-39　互连块

5.5　设计一个具有使能输入的2 × 4解码器，使用SYSTEM_4包中定义的四值逻辑系统MVL4，输入和输出既可以是向量也可以是标量，当使能输入为‘0’时，所有的输出均为‘z’。当使能输入为‘1’时，只有一个输出为‘1’，其他输出均为‘0’，当数据输入或使能输入的值为‘Z’或‘X’时，输出值任意。请说明在这种情况下如何来决定输出值。该单元自然应该为组合逻辑，包括下面两个类属延时：

1) DATA_EDLAY。当输出已经被使能时,一个输入数据变化后导致输出变化的延时。

2) ENABLE_DELAY。使能输入变化时导致输出变化间的延时。

设计中应包括一个实体和一个算法级行为域的构架描述。

5.6　设计一个具有使能输入的2 × 1向量多路器(MUX)，每个数据输入和输出都是8位向量，使

用包SYSTEM_4中的四值逻辑系统。该单元应为组合电路，当使能输入为逻辑0时，输出向量为“ZZZZZZZZ”；当使能输入为逻辑1时，输出由选择输入和数据输入所决定，设计中包括下面3类延时：

1) DATA_DELAY。当输出被使能时，任一数据输入变化导致数据输出变化的延时。

2) SELECT_DELAY。当输出被使能时，任一选择输入变化导致数据输出变化的延时。

3) ENABLE_DELAY。使能输入变化时导致输出变化间的延时。

设计应包括一个实体描述和一个算法级行为域的构架描述。

5.7 设计一个4位升/降计数器，计数器有4个控制输入：UP、DOWN、LOAD和ENABLE。所有的计数器状态变化都与输入CLK的上升沿同步，异步输入CLEAR使计数器的所有位同时变为ZERO状态而独立于CLK，计数器有四位输入DATA_IN和四位输出DATA_OUT，当ENABLE=1时，DATA_OUT等于计数器中存储的4位值；当ENABLE=0时，DATA_OUT为“ZZZZ”。ENABLE信号独立于时钟工作，若UP=1，则计数器在CLK的每个上升沿递增。若DOWN=1，则计数器在CLK的每个上升沿递减。若LOAD=1，则计数器在CLK的上升沿从DATA_IN载入并行数据。只有当计数器递增达到‘1111’或递减达到‘0000’时，输出C为逻辑‘1’。该计数器为模16计数。若某一时刻UP、DOWN和LOAD中不只一个信号活跃，则结果为未定义。可以假设有函数INC4和函数DEC4存在，它们执行4位递增和4位递减操作。参见图4-26a和图4-26b的包PRIMS。

a) 画出UP/DOWN计数器的进程模型图。使用几个简单进程的设计所给分数将高于采用少量复杂进程的设计。

b) 基于a)所构造的进程模型图，写出计数器的算法级VHDL描述。

5.8 为8214优先级中断控制单元建立模型，要求该模型满足以下条件：

a) 模型必须只有一个构架，它属于芯片级行为域模型。

b) 通过分析来说明如何为模型设计路径延时。

c) 画出模型的进程模型图。

d) 包括建立和保持时间冲突的检测。

e) 对模型进行彻底的测试。

f) 写出一份精心准备的报告，阐述建模方法并解释得出的结果。

5.9 为16位并行错误检测校正电路SN74ALS617建立模型，建模过程包括以下方面：

a) 应为单构架芯片级行为域模型。

b) 分析说明如何为模型产生路径延时，该延时应从数据页中的I/O延时数据推论得出。

c) 画出模型的进程模型图。

d) 检测建立和保持时间的冲突，使用手册（右手栏）所给的建立和保持时间定义。

e) 完全测试模型，包括如下步骤：

1) 测试结构或模型中的EDAC，它包括一个RAM和一个Micro_shell，对于RAM，可以从一个简单的RAM模型开始，通过修改该模型使之能够检测：一位错误、双位错误、三位错误、全1错误和全0错误。给定RAM的一些控制输入，便之能根据命令查出这些错误。Micro_shell是一个用适当逻辑来控制EDAC和RAM的简单模型。

2) 使用EDAC“产生检测字”的功能来模拟写入存储器的过程（规范中的表1），在几种错误条件和无错情况下，使用“读、标记和纠错功能”（规范中的表4）读回

存储器中的数据。

f) 写出一份报告，阐述建模方法并解释得出的结果，解释结果时，对打印输出进行注释，从而可以更明显地显示是如何建模的。

5.10 写出一个线与的总线判决函数的代码，使之可以用于MVL('0', '1', '2')类型的信号。

5.11 假设信号类型为MVL('0', '1', '2')，写出一个线或的总线判决函数。

5.12 下面给出了一个类型BIT信号的判决函数，解释该函数的工作过程，并说明实现该函数的物理电路是什么？

```
function RES_FUNC(signal X: BIT_VECTOR) return BIT is
begin
   for i in X'range loop
      if X(i) = '0' then
         return '0';
      end if;
   end loop;
end RES_FUNC;
```

5.13 为具有上拉的四值逻辑系统MVL4的Wired_X判决函数产生一个双参数真值表。

5.14 下面是线下拉的总线函数的真值表和函数声明：

```
constant table_WIREDPULLDOWN: MVL4_TABLE :=
--  ---------------------------------
--  |  X    0    1    Z  |
--  ---------------------------------
    (('X', 'X', '1', 'X'),  --  | X |
     ('X', '0', '1', '0'),  --  | 0 |
     ('1', '1', '1', '1'),  --  | 1 |
     ('X', '0', '1', '0')); --  | Z |
function WiredPullDown (V: MVL4_VECTOR) return MVL4 is
   variable result: MVL4 := 'Z';
begin
   for i in V'range loop
      result := table_WIREDPULLDOWN(result, V(i));
      exit when result = '1';
   end loop;
   return result;
end WiredPullDown;
```

该函数实现了什么电路特性？画出该总线的电路图。

5.15 下面给出了一个具有两个进程的模型的部分VHDL描述。进程ONE和进程TWO被两个互相排斥的脉冲P1和P2激活，这两个进程都载入寄存器X。希望模型能够实现时分复用，即进程ONE在P1为'1'时可以控制X，进程TWO当P2变为'1'时可以控制X。

a) 给出的代码是否正确？如果不正确，为什么？

b) 如果对a的回答为否，则修改代码，使之能够工作。

```
type MVL is ('0','1',Z);
signal X: F MVL;  ----- F a previously declared bus
                  ----- resolution function.

ONE: process(P1)
begin
  --------
  --------
  X <= Y after 100 ns;
end process ONE;
```

```
TWO: process(P2)
begin
  -------
  -------
  X <= Z after 100 ns;
end process TWO;
```

5.16 设计一个具有二个二进制输入（X和Y）和一个二进制输出（Z）的器件，当输入X由0变为1，输出Z为1并保持到Y发生变化，当x=1时y的任何变化都会导致输出发生变化，如果输入X和Y同时发生变化，则输出未定义，图5-40为该器件的时序。

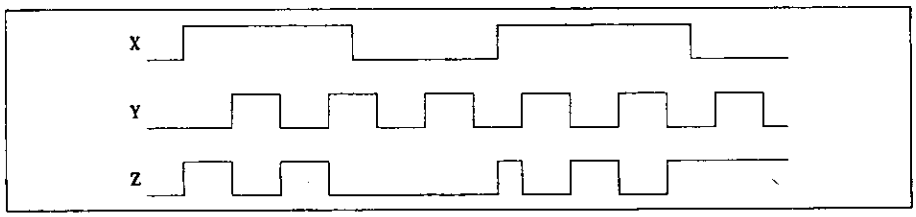


图5-40 器件时序定义

根据上面的设计规范，一位学生写出了如下的算法级描述：

```
entity DEVICE is
  port (X,Y: in VLBIT; Z: inout VLBIT);
end DEVICE;

architecture ALG of DEVICE is
begin
 Px: process (X)
 begin
  if X = '1' then
    Z <= '1';
  end if;
 end process;
 Py: process (Y)
 begin
   if X = '1' then
     Z <= not Z;
   end if;
 end process;
end ALG;
```

a) 上面的代码含有一个基本错误，说明这个错误。

b) 设计一个能正确工作的构架，用独立的进程执行上面两个进程体的功能。可以增加一些进程和额外的信号或变量。

5.17 VHDL模型的信号R表示一个寄存器硬件，设计期望R可以由两个不同的进程驱动。并且R总保持最后一个赋给它的值，用VHDL写出R的模型。

5.18 为图5-41所示数据传输控制单元写出一个算法级描述。当R='1'时单元无任何动作。R变低则单元进入等待输入数据可用（IDAV）信号上升的状态。当IDAV上升时单元进入读取输入数据（IDAT）并传输到输出端QDAT的状态。同样它也将输出数据可用（ODAV）置为'1'。输出数据和ODAV保持一个CLK_PER的时间，然后单元返回到等待IDAV再次上升的状态.算法要求使用Wait语句来控制异步和同步推进。

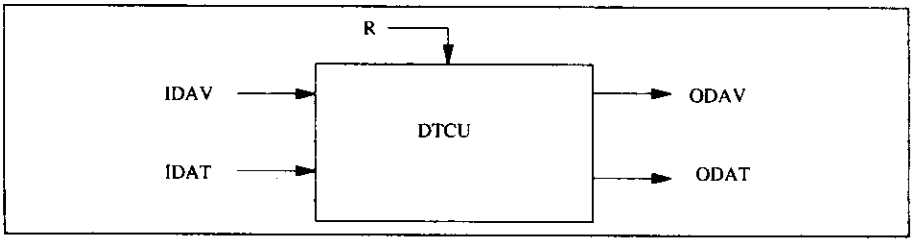


图5-41　数据传输控制单元

5.19　设计一个可变延时单元，在流水线计算机系统中，数据以流的形式前进。当数据流来自不同的源时，有时需要延时一个流以使得它在合适的时间到流水线输入端。此问题的模拟方法不可靠，使用移位寄存器则只适用于小的延时。图5-42给出了一个较好的方法，它使用双端口存储器延时并行数据流。在一个双端口存储器中，可以在读写地址不重叠时，同时对存储器进行读和写。它也给出了存储器的读/写控制器。系统的工作过程如下：一个RESET信号初始化设备。当START信号到达一个时钟周期，写地址(WADDR)初始化为0，读地址(RADDR)被初始化为DELAY(希望数据延时的时钟数)。在下一个时钟周期，第一个数据字进入存储器端口。每个时钟周期里数据被写入并且WAPDR递增。但是，对于第一个延时周期，RADDR递减但没有发生读。当RADDR变为0后读操作开始，此后，每个时钟RADDR递增。因此，在启动延时后，在DELAY个时钟周期之后，DATA_OUT将等于DATA_IN。当STOP变为1时，系统完成保存在存储器中的数据传输。该系统完全是一个同步系统，即控制单元和双端口存储器受同一个时钟触发。双端口存储器可以被看作是一个时钟激发的寄存器阵列。可以假设数据源也由同一个时钟触发，并且START和STOP也随时钟而变化。

a) 为控制器和双端口存储器建立一个算法级描述。控制器描述可以使用WHILE循环和WAIT语句。控制器应能把数据延时0~15个时钟脉冲。双端口存储器至少应具有支持这个延时周期数的最小单元数。数据字为4位。

b) 用命令文件模拟算法描述，此命令文件为重复几次0到15二进制序列的数据流。模拟不同的延时值。

c) 使用门和触发器把控制寄存器综合为门级描述。对于控制单元用与/或测试WADDR和RADDR的一部分，产生一个门级控制寄存器接口的算法级VHDL模型。使用a中产生的双端口存储器模型。

d) 用b中的方法模拟这个混合级设计。

e) 提交一份报告，描述并评价你的设计。

5.20　本题的目的在于理解如何为共用一个数据总线的多模块系统建立模型。图5-43给出了包括一个缓冲寄存器RAM、一个ADD和STORE单元的三模块系统。系统功能如下：

a) 当STRB变高时，数据（DI）进入缓冲寄存器，并且数据可用信号（DAV）变为'1'。

b) ADD和STORE通过置EN='1'来响应DAV='1'。它把数据从缓冲寄存器放到DATA-BUS上并重置DAV。

c) ADD和STORE读取来自DATA_BUS上的数据，把它与输入值NUM相加，并把结果存储在一个内部保持寄存器中。

d) ADD和STORE初始化一个RAM写操作，该操作把内部保持寄存器的值写入由输入DADDR指定的RAM单元中。

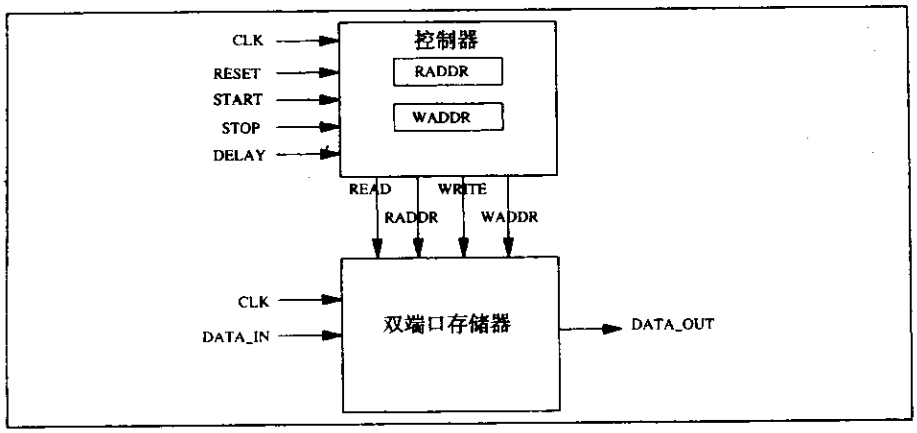


图5-42 变量延时

e) ADD和STORE返回到它的初始状态；等待来自缓冲寄存器的下一次数据可用信号(DAV)。

可以使用文中的缓冲寄存器模型，RAM的模型与文中给出的非常相似，为ADD和STORE单元创建构架。对整个系统进行仿真验证其正确性。

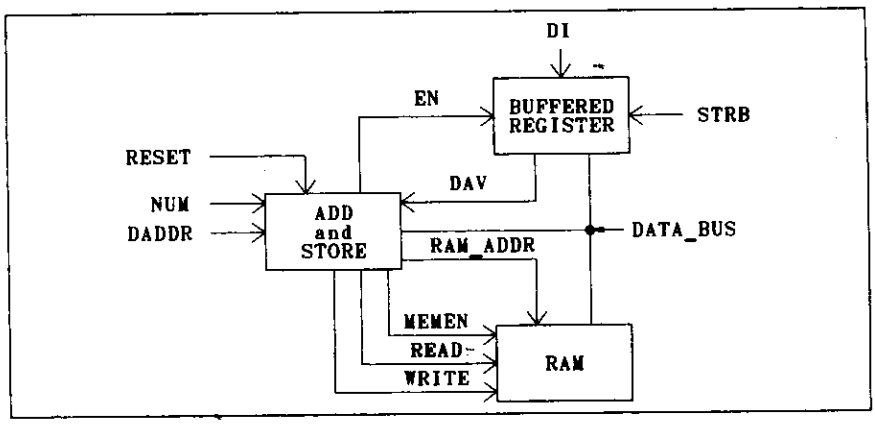


图5-43 Add和Store

5.21 用VHDL为DMA控制器接口建模，该接口在Morris Mano所著的《计算机系统构架》(第2版，Prentice Hall 1982年出版)一书中的第11章讨论。全部这些讨论对本题很有用，但应特别对如图11-18所示的系统建模为：

a) 为DMA控制器和RAM建立完整模型。

b) 为微处理器和外围设备建立部分模型，部分微处理器模型仅需包含建立微处理器、总线和DMA控制器之间的接口模型。外围设备模型只需与DMA控制器交互，能够发送和接收数据块。

c) 模型应包括实际时序，时序具体如何由你自己决定，一个例外是：RAM存储器模型单元必须包括地址和数据的建立时间及片选和读写控制的检测，所有时序均要作好记录。

d) 总线使用四态逻辑(X，0，1，Z)，因此使用SYSTEM-4包。

e) 通过仿真验证模型的正确性，除了验证一般操作，还需仿真以下情况：1) 一个设备

出错导致总线竞争。2) RAM输入的错误时序导致错误结果。

f) 准备一份报告总结结果。

5.22 图5-44是一个零活跃输入脉冲。它的定义指出：为了激活它所驱动的设备，它必须具有最小宽度为MPW ns的低脉冲。任何比它更窄的零脉冲都必须报告。写出检测和报告错误的VHDL代码。

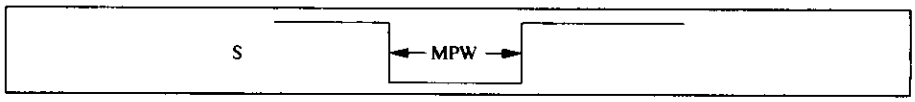


图5-44 零活跃输入脉冲

5.23 用自然语言解释下面代码的作用：

```
entity DAVCON is
  generic(CLK_PER:TIME);
  port(R,IDAV: in BIT;
       IDAT: in BIT_VECTOR(7 downto 0);
       ODAV: out BIT;
       ODAT: out BIT_VECTOR(7 downto 0));
end DAVCON;

architecture WAIT_LOOP of DAVCON is
begin
  process
  begin
    WLOOP: while R = '0' loop
      ODAV <= '0';
      wait until  IDAV = '1' and IDAV'EVENT;
      ODAT <= IDAT; ODAV <= '1';
      wait for CLK_PER;
    end loop;
  end process;
end WAIT_LOOP;
```

5.24 把5.3.2节描述的系统模型转换为IEEE九值逻辑。仿真并检测结果的正确性。

第6章　寄存器级设计

本章讨论在寄存器抽象级进行设计。在这一级，将用VHDL数据流来进行行为描述。

6.1　从算法到数据流描述的转换

在前一章，讲述了适合于系统级建模的VHDL算法模型。其中存在两个基本问题：1) 数据流描述与算法描述的区别何在？2) 什么样的设计活动可以在寄存器级进行，而无法在芯片级进行？我们先解答第1个问题。虽然算法级描述隐含了寄存器传输过程，但是数据流描述可以显式地表示它。特别是对于数据流描述:

1) 声明了表示真实电路中数据流动和连接的信号。

2) 在数据流描述的语句和寄存器结构模型之间有着直接映射关系。

3) 因为标识了寄存器级元件，而且确定了其连接关系，因而隐含了布局。

4) 标识了多路器和总线。

5) 寄存器的时钟机制是确定的。

6) 抽象数据类型通常用类型来表示，如BIT、BIT_VECTOR、MVL4、MVL4_VECTOR、STD_LOGIC和STD_LOGIC_VECTOR。它可以进行状态赋值，可以标识寄存器并指定其长度。

对于第2个问题，因为已经显式地表示了寄存器的互连，所以可以用数据流来研究。

1) 寄存器级元件的时序关系。

2) 资源分配。

3) 调度。

4) 微码控制单元设计。

5) 总线设计。

本章讲解第1项和第4项，第5项为作业，第2项和第3项将在第12章中涉及。

转换示例

考虑如下从算法级描述到数据流描述转换的例子：

一个系统由2个8位寄存器R1和R2和一个加法器组成。2位命令输入COM用来指定如下四种命令：

1) COM= “00”。载入R1。

2) COM= “01”。载入R2。

3) COM= “10”。R2与R1相加。

4) COM= “11”。从R2中减去R1。

图6-1给出了该寄存器系统的算法描述。尽管指明发生了什么，却无法说明是怎样发生的。

图6-2给出了该寄存器的详细数据流描述。注意到：1) 定义了信号以说明互连关系；2) 寄存器、多路器、译码器和数据操作——加、递增和反相——被严格确定；3) 信号赋值表明了数据是如何流动的。这时候，可以很容易画出一张该系统的原理图。

```
use work.funcs.all;
entity REG_SYS is
  port(C: in BIT; COM: in BIT_VECTOR(0 to 1);
       INP: in BIT_VECTOR(0 to 7));
end REG_SYS;
architecture ALG of REG_SYS is
  signal R1,R2: BIT_VECTOR(0 to 7);
begin
  process(C)
  begin
    if C='1' then
      case COM is
        when "00" => R1 <= INP;
        when "01" => R2 <= INP;
        when "10" => R1 <= ADD8(R1,R2);
        when "11" => R1 <= ADD8(R1,INC8(not(R2)));
      end case;
    end if;
  end process;
end ALG;
```

图6-1 寄存器系统的算法模型

```
architecture DF1 of REG_SYS is
  signal MUX_R1,R1,R2,R2C,R2TC,MUX_ADD,SUM:
         BIT_VECTOR(0 to 7);
  signal D00,D01,D10,D11,R1E: BIT;
begin
  D00 <= not COM(0) and not COM(1);
  D01 <= not COM(0) and COM(1);     ---Command Decoder
  D10 <= COM(0) and not COM(1);
  D11 <= COM(0) and COM(1);
  MUX_R1  <= SUM when D00 = '0' else  --Register 1 Mux INP;
      R1E <= D00 or D10 or D11;       --Register 1
R1_REG: block(R1E = '1' and C='1' and not C'STABLE)
  begin
    R1 <= guarded MUX_R1;
  end block R1_REG;
R2_REG: block(D01 = '1' and C='1' and not C'STABLE)
  begin
    R2 <= guarded INP;              ---Register 2
  end block R2_REG;
  R2C <= not R2;                    ---Complement
  R2TC <= INC8(R2C);                ---Increment
  MUX_ADD <= R2TC when D11 = '1' else
             R2;                    ---Adder Mux
  SUM <= ADD8(R1,MUX_ADD);          ---Adder
end DF1;
```

图6-2 详细数据流描述

也可以采用其他的数据流描述。图6-3给出了一个简要的数据流描述实现，它使用了两个防护信号赋值语句。尽管这个表示很简单，但它对于互连关系的表示能力还是比较弱，并且不支持以下将讨论的一些设计活动。

对图6-2中的详细数据流描述的一种改进方法是进行本地译码（图6-3中简要描述确实是这

样做的），如果采用本地译码技术，则不需要中心译码器，译码在寄存器或多路器上进行，这里需要进行控制。在这个简单的寄存器系统中，本地译码技术比较先进。图6-4给出了使用本地译码的详细数据流描述。图6-5给出了本地译码的一个原理图。

```
architecture DF2 of REG_SYS is
  signal R1,R2: BIT_VECTOR(0 to 7);
begin
R1_REG: block((COM(0) or not COM(1))='1' and C='1'
              and not C'STABLE)
  begin
    R1 <= guarded
       ADD8(R1,R2) when (COM(0) and not COM(1)) = '1' else
  ADD8(R1,INC8(not(R2))) when (COM(0) and COM(1)) ='1' else
       INP;
  end block R1_REG;
R2_REG: block(( not COM(0) and COM(1)) = '1' and C='1'
              and not C'STABLE)
  begin
    R2 <= guarded INP;
  end block R2_REG;
end DF2;
```

图6-3 寄存器系统的精确数据流描述

```
architecture DF3 of REG_SYS is
  signal MUX_R1,R1,R2,R2C,R2TC,MUX_ADD,SUM:
         BIT_VECTOR(0 to 7);
  signal R1E,R2E: BIT;
begin
  MUX_R1 <= SUM when COM(0) = '1' else  ---Register 1 Mux
            INP;
  R1E  <= COM(0) or not COM(1);          ---Register 1
R1_REG: block(R1E = '1' and C='1' and not C'STABLE)
  begin
    R1 <= guarded MUX_R1;
  end block R1_REG;
  R2E <= not R1E;
R2_REG: block(R2E = '1' and C='1' and not C'STABLE)
  begin
    R2 <= guarded INP;                   ---Register 2
  end block R2_REG;
  R2C <= not R2;                         ---Complement
  R2TC <= INC8(R2C);                     ---Increment
  MUX_ADD <= R2TC when COM(1) = '1' else
             R2;                         ---Adder Mux
  SUM <= ADD8(R1,MUX_ADD);               ---Adder
end DF3;
```

图6-4 本地译码技术的详细数据流描述

通常，中心译码有一定的优点。首先，它提供一种功能分解的更自然的形式，可以更容易地自动进行综合；其次，中心译码器的存在使得增加新功能会更加容易。然而，对于完全定制的设计，本地译码可以使逻辑简化。

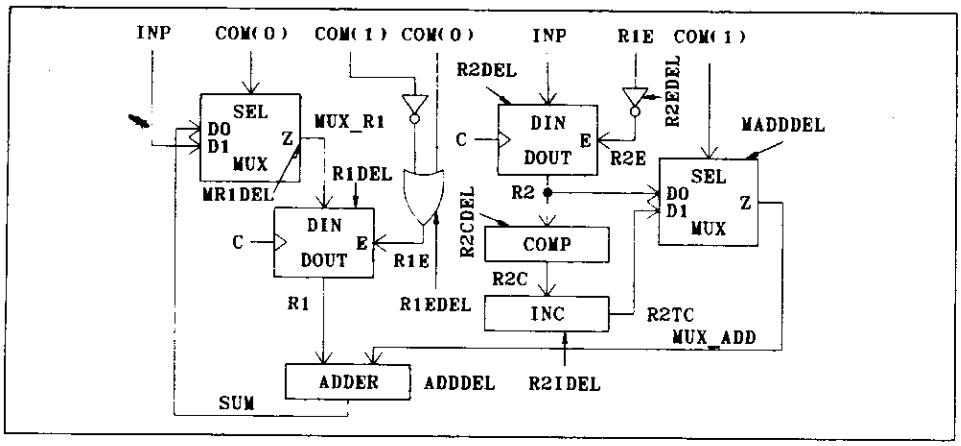


图6-5　带本地译码技术的详细数据流原理图

6.2　定时分析

现在讲解如何使用寄存器级模型的数据流描述进行时序分析。这里，将使用结构DF3并加入时序。图6-6给出了如何将时序加入模型之中。注意，Generics（类属）用来表示延时。每个寄存器元件的类属延时在图6-5中列在了元件的旁边。

在对寄存器级模型进行时序分析时，我们发现寄存器系统中隐含着两个时序问题：1) 命令输入端COM建立时间的要求与时钟C上升沿的关系；2) 时钟C的最小时钟周期要求。图6-7说明了这两个要求。

下面详细讨论这两个要求。

1) 命令输入端COM的建立时间与C的上升沿（COMSUTC）的关系。当命令输入COM的值改变时，在C的上升沿之前必须有足够的时间，以保证寄存器的输入稳定下来，可以将这些要求列出如下：

a) Register 1 使能端（R1E）稳定：COMSUTC >R1EDEL。

b) Register 2 使能端（R2E）稳定：COMSUTC >R1EDE + R2EDEL。

c) Register 1 数据输入端（MUX_R1）稳定：

COMSUTC >MADDEL+ADDDEL+MR1DEL

其中，要求c是最苛刻的要求，因此，将它视为COM的建立时间与C上升沿的关系要求。

2) 时钟C的最小时钟周期（CPER）要求。如果在某个周期数据存入寄存器R2（COM="00"），随后在下一个时钟周期是减法（COM="11"），时钟周期必须满足如下要求：

CPER>R2DEL+R2CDEL+R2IDEL+MADDDEL+ADDDEL+MR1DEL

虽然这里面对的是特定的电路，但这两个要求通常也是系统的典型要求。

在使用用于建模的带有时序的寄存器系统模型时，满足COM和C的要求是很重要的。为了保证这一点，在接口描述之中加入了“断言”（assertion），断言包含在一个由时钟C的变化而触发的进程内部。断言对于COM进行检查，它比较简单，无需太多的讨论。最小时钟周期测量相继的两个时钟正周期的时间间隔并与定义（规范）相比较，它使用了包集合Standard的函数NOW，该函数调用时返回当前的仿真时间。

```
use work.funcs.all;
entity REG_SYS_T is
  generic(MR1DEL,R1EDEL,R1DEL,R2EDEL,R2DEL,
          R2CDEL,R2IDEL,MADDDEL,ADDDEL: TIME);
  port(C: in BIT; COM: BIT_VECTOR(0 to 1);
       INP: in BIT_VECTOR(0 to 7));
begin
  process(C)
    variable COLDT,CNEWT: TIME:= 0 ns;
  begin
    assert not(C'EVENT and C = '1'
      and not COM'STABLE(MADDDEL+ADDDEL+MR1DEL))
      report "COM Set Up Time Failure" severity WARNING;
    if C'EVENT and C = '1' then
    CNEWT := NOW;-- NOW returns the current simulation time.
      assert (CNEWT - COLDT) > (R2DEL + R2CDEL + R2IDEL
                              + MADDDEL + ADDDEL + MR1DEL)
        report "Clock Period Too Short" severity WARNING;
    end if;
    COLDT := CNEWT;
  end process;
end REG_SYS_T;
architecture DF3T of REG_SYS_T is
  signal MUX_R1,R1,R2,R2C,R2TC,MUX_ADD,SUM:
         BIT_VECTOR(0 to 7);
  signal R1E,R2E: BIT;
begin
  MUX_R1 <= SUM after MR1DEL when COM(0) = '1' else
            INP after MR1DEL;
  R1E <= COM(0) or not COM(1) after R1EDEL; ---Register 1
R1_REG: block(R1E = '1' and C='1' and not C'STABLE)
  begin
    R1 <= guarded MUX_R1 after R1DEL;
  end block R1_REG;
  R2E <= not R1E after R2EDEL;
R2_REG: block(R2E = '1' and C='1' and not C'STABLE)
  begin
    R2 <= guarded INP after R2DEL;              ---Register 2
  end block R2_REG;
  R2C <= not R2 after R2CDEL;                   ---Complement
  R2TC <= INC8(R2C) after R2IDEL;               ---Increment
  MUX_ADD <= R2TC after MADDDEL when COM(1) = '1' else
             R2 after MADDDEL;                  ---Adder Mux
  SUM <= ADD8(R1,MUX_ADD) after ADDDEL;         ---Adder
end DF3T;
```

图6-6 有时序关系的寄存器系统

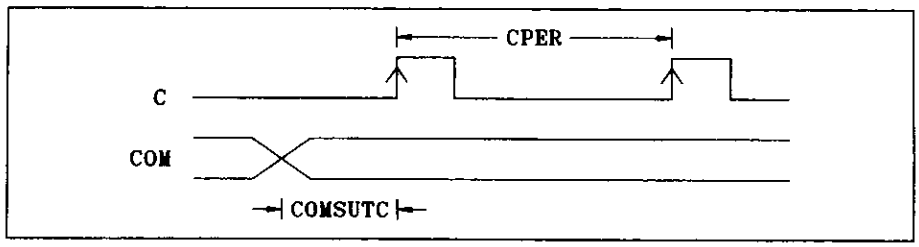


图6-7 寄存器系统时序说明

6.3 控制单元设计

寄存器级设计的一个主要设计活动是控制单元的设计，因为在这一级，所有的控制信号都是显式表示的。器件被分为两个主要部分，见图6-8。数据单元包括存储操作数和结果寄存器、操作和处理数据的组合逻辑单元。控制单元产生一系列的控制信号，以控制数据单元中数据的传输。控制单元需要数据单元提供状态信息，在控制单元中控制条件分支操作的执行。数据单元也根据数据、状态和控制信号产生器件输出信号。

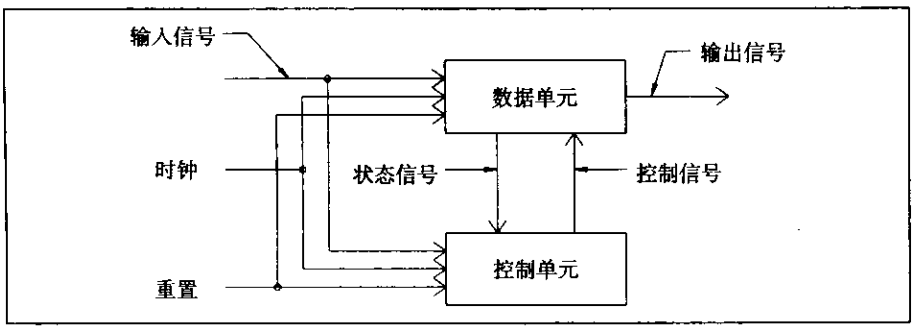


图6-8 寄存器器件划分为基本的单元

6.3.1 控制单元的类型

控制单元可以用微码或标准门（硬连线）来实现。硬连线控制单元必须对每个单元器件进行定制设计，而且可以有多种形式。图6-9给出了一个有限状态机控制器的Huffman模型，块MEM包括所有的控制触发器，块CL（组合逻辑）包括控制器的所有逻辑门。控制信号的当前值通常由来自数据单元的状态信息和控制触发器的当前状态决定（Mealy型）或仅由控制触发器的当前状态决定（Moore型）。控制触发器的下一个状态由当前状态及数据单元的状态信息决定。

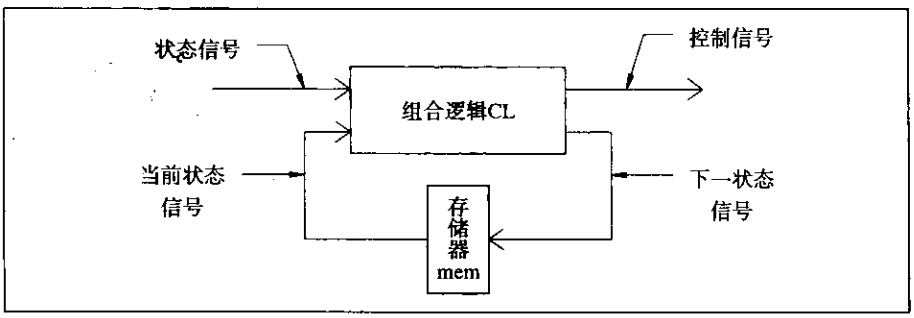


图6-9 硬连线控制器的Huffman模型

在第8章中介绍的控制器设计中，单热（one_hot）状态赋值的方法是Huffman模型的特殊情况：在控制器中，每个分立的触发器对应于一个状态。这种特殊情况具有容易自动操作的特点。尽管使用的寄存器比需要的多，但CL往往没那么复杂了，有可能降低整体复杂性。

硬连线控制器比微码控制器速度要快。主要的缺点是其更大的复杂性，更大的开销，以及必须进行定制设计。

微码控制器每一拍只需要从ROM中读出所有控制信号的值，见图6-10。在每一拍，所有控制信号的当前值从ROM中读出，存入存储器指令寄存器（MIR）。控制信号的电平保持在MIR的输出上，直到从ROM之中读出下一个值。正确的控制信号序列通过在ROM的输入端产生正确的地址序列实现。这种类型的控制器当地址序列相对简单时，工作得最好。通常，地址生成器（AG）电路是带有并行载入功能的计数器，当控制序列想要跳转到新的节点去执行时，则并行载入新地址。一般情况下，控制器顺序转入下一条序列地址以取得下一串控制信号值，偶尔需要并行载入一个新地址。地址产生器（AG）电路可以是一个不太复杂的电路，本章后面的小节将讨论针对地址产生器的其他折衷方法。

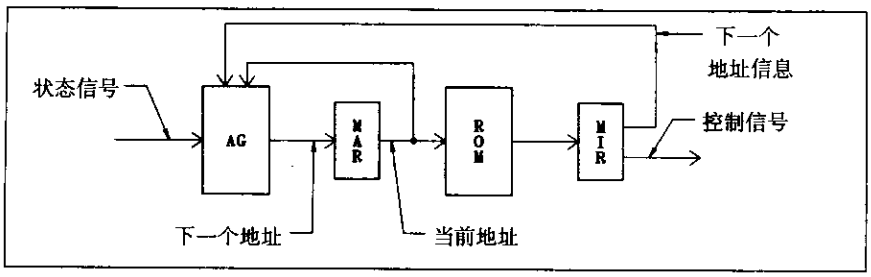


图6-10 微码控制单元的框图

微码控制器的主要优点是：

1) 对于一系列器件可以开发出一个标准设计。不同之处在于存储在ROM中控制信号的值。因而，通过改变ROM，可以得到不同的控制器。可以大批量生产标准板，在收到定货时，可以将相应的ROM安装在板上提供给用户。

2) 改变设计更容易。只需改变ROM程序就可以了。控制器硬件不必改变。因而，相对于硬连线控制器，改变设计所费费用较小。

3) 设计时间与设计开销大大减少，因为硬件已经设计和调试过，控制器设计精简为对ROM进行编程，从而可以有效使用现有的软件工具，如编译器、汇编器、仿真程序和调试器。通常，在设计过程中极少发生错误，出现的错误通过修改程序也极易解决，而不用修改硬件连线。

主要缺点是：

1) 与硬连线设计相比，考虑到ROM的读时间，操作速度较慢。

2) 对于小器件，微码设计将更加昂贵。因为至少需要包含一个ROM和两个寄存器。然而，对于非常大型的设计，微码控制器将更加便宜，因为采用硬连线设计需要大量的门和触发器。

3) 使用标准的控制器限制设计者只能使用控制器内置的功能。功能越多则越昂贵。即使他们不使用控制器的全部功能，但每个人必须承担更高的费用。每个设计者必须注意到控制器带来的限制。这些限制包括可以产生的控制信号的数量，可以处理的状态信号，以及由地址产生逻辑所带来的局限。控制器的设计是在灵活性和耗费之间进行折衷的结果。

我们通过讨论一个小的RISC机器的设计，来说明这些基本概念。

6.4 终极RISC机

RISC表示精简指令集计算机，与复杂指令集CISC不同的是，它的指令集极其简单。RISC

机器用VLSI实现效率极高，可以使其以很高的速度来执行其指令集，从而抵消了其指令集较小的缺点。既然决定设计一个小指令集的计算机，可以考虑一下由Mavaddat和Parham提出的终极RISC构架（URISC），URISC之所以是终极RISC机器，因为它只有一条指令，然而，两人声称该机是通用计算机，任何复杂的操作都可以使用这条指令编程而得到。这种简单性使得可以用最少的细节讨论许多设计问题。

6.4.1 单条URISC指令

URISC的指令是SUBTRACT AND BRANCH ON NEGATIVE指令。因为只有一条指令，因而没有必要定义操作码来指定指令的操作。指令格式只需标识出操作数，当结果为负时标识出分支地址即可。URISC的单一指令格式如下：

第一操作数的地址　　第二操作数的地址　　分支地址

指令的执行情况如下：

1) 从第二操作数中减去第一操作数，结果存在第二操作数的位置处。

2) 如果减法的结果为负，执行跳转到目标地址处，否则执行下一条指令。

3) 跳转到位置0的指令导致停机。

这样，所有的指令都是3字节指令，所有的指令都需3个存储器周期。因而，只有在访问高速存储器时URISC的运行才会很快。对于存储器的速度要求后面还有讨论。

对于此种指令，可以使用汇编语言助记符。

L：F，S，T

其中：

L是标号，标记该指令存储的地址。

F是第一操作数的符号名。

S是第二操作数的符号名。

T是分支地址的符号名。

假设存在如下形式的汇编语言伪操作：

L：WORD C

它使常量C的值存入位置L。

作为使用URISC的一个例子，请参考图6-11中所示的程序，该程序完成操作Z=(X+Y)/2。注意，(X+Y)除以2是通过连续的减法实现的。程序结束时，跳转至STOP（位置0），从而引起停机。后面将讨论如何实现的问题。

6.4.2 URISC的体系结构

本节描述URISC机器的体系结构。图6-12给出了URISC的数据单元。指令计数器（PC）保存正在执行的指令的下一个字的地址。存储器数据寄存器（MDR）和存储器地址寄存器（MAR）作为与程序存储器的接口。寄存器R保存减法操作前的第一操作数，减法操作由一个补码加法器来执行。加法器的一个输入值在BUS_A上，另一个输入或是R寄存器的按位求补结果（如果COMP是‘1’），或是常量“00000000”（如果COMP是‘0’）。加法器的输出一般连到BUS_B，COMP=‘1’且Cin=‘1’时，加法器执行补码减法，从BUS_A上减去寄存器R的值。COMP=‘0’且Cin=‘1’时，加法器给BUS_A的值加1，这种功能用来递增PC。COMP=‘0’且

Cin='0'时，BUS_A上的值直接传输到BUS_B上，因为BUS_A与常量0相加，这种配置用来将PC中的数据传送到MDR或MAR。加法器包括两个状态输出Z和N，分别指明结果为0或为负。如果Zin与Nin有效，这些状态值载入相应的状态触发器。

Memory	Symbolic	Instruction	Comments
0	STOP:	WORD 0	;Halt
1	READ_Y:	Y,TEMP1,NEXT1	;TEMP1<=(-Y)
2	NEXT1:	TEMP1,X,NEXT2	;X<=Y+X
3	NEXT2:	Z,Z,TEST	;Z<=0
4	TEST:	X,TEMP2,POSITIVE	;TEMP2<=-(X+Y)
5	NEGATIVE:	TWO,TEMP2,STOP	;TEMP2<=-(X+Y)-2
6	COUNT_NEG:	ONE,Z,NEXT3	;Z<=Z-1
7	NEXT3:	TWO,TEMP3,NEGATIVE	;Go to NEGATIVE
8	POSITIVE:	TWO,X,STOP	;X<=(X+Y)-2
9	COUNT_POS:	MONE,Z,NEXT4	;Z<=Z+1
10	NEXT4:	TWO,TEMP4,POSITIVE	;Go to POSITIVE
	TEMP1:	WORD 0	
	TEMP2:	WORD 0	
	TEMP3:	WORD 0	
	TEMP4:	WORD 0	
	X:	WORD 0	
	Z:	WORD 0	
	ONE:	WORD 1	
	MONE:	WORD-1	
	TWO:	WORD 2	

图6-11 计算(X+Y)/2的URISC程序

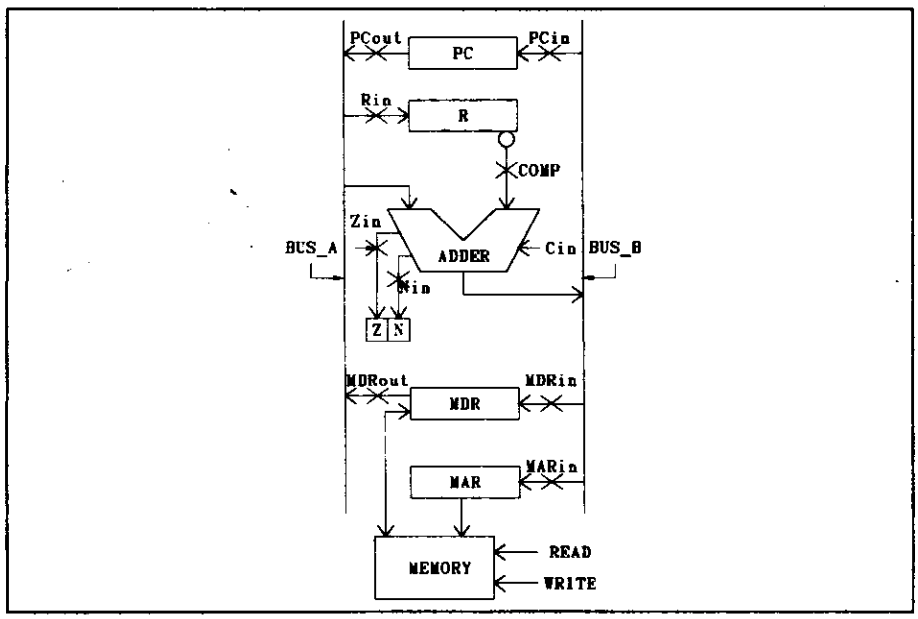


图6-12 URISC机器的数据单元

框图中的控制点由X标记。每个控制点上都标记着一个信号名。带有out后缀的控制点

（如PCout）作为将数据输入BUS_A的使能信号。当该信号有效时，指定的寄存器将值送入BUS_A。例如，当PCout有效时，PC寄存器的内容被置于BUS_A上。

带有后缀in的控制点控制将数据载入指定的寄存器。例如，当MARin有效时，BUS_B上的数据载入MAR。当Zin有效时，加法器的Z状态输出被载入Z状态触发器。

程序计数器通过使控制信号PCout、Cin与PCin有效来实现递增。PCout将PC与BUS_A相连。如果COMP无效，加法器的第二个输入是"00000000"，当Cin=1时，PC+1被送入BUS_B，当Pcin有效时，在时钟周期结束时PC+1将被载入PC。

通过将第一操作数载入R寄存器，第二操作数载入MDR寄存器，实现了从第二操作数中减去第一操作数。然后，当MDRout、Cin、COMP和MDRin有效时，执行补码减法并将结果存入MDR寄存器。

PC中的地址可以通过仅使PCout和MAR有效而移入MAR，PC中的地址与0相加，因而并不改变。

6.4.3 URISC的控制

为了控制URISC，数据单元中不同的控制点必须在适当的时刻有效。这通过控制单元来实现。图6-13给出了URISC的数据单元、控制单元和存储单元的框图。URISC的控制单元以系统时钟做为输入，用来控制信号的改变。控制单元的输出是图6-12中所示的控制信号（PCout、COMP、MDRin等），另外，控制单元还产生两个内部使用的特殊的控制信号ZEND和NNEND，下一节将解释其原因。

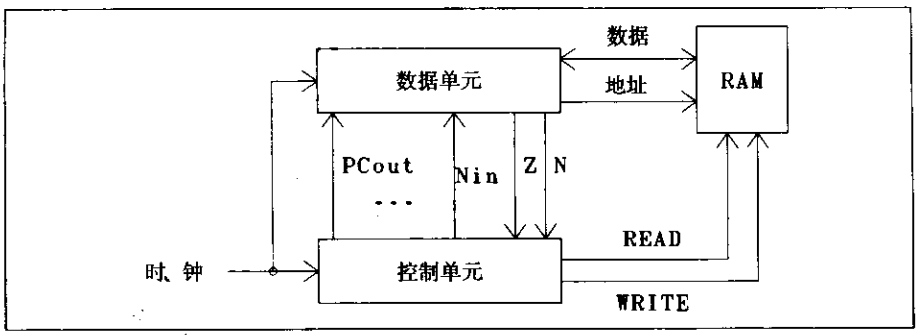


图6-13　URISC机器的框图

1. URISC指令周期控制序列

在每个时钟周期，控制单元必须输出每个控制信号的状态。图6-14给出了一个控制序列，控制从存储器中取一条指令并且执行。每一拍列出了必须有效的控制信号。未列出的信号无效。假设进入控制序列时，PC寄存器保存下一条执行的指令的第一个字的地址。再假设每条指令占据存储器相邻的三个位置，即第一操作数的地址、第二操作数的地址和分支地址（当从第二操作数减去第一操作数的结果为负时）。在控制序列的末尾，下一个执行指令的第一个字将存于PC寄存器中。通过重复这个序列，可以执行程序，这些步骤解释如下：

1）节拍0到节拍2：将第一操作数读入R寄存器。节拍0通过使PCout与MARin有效，将指令地址载入MAR。因为COMP与Cin无效，加法器将给PC地址加0。因为Zin有效，如果PC寄存器是0，则Z触发器置0。传输结束后，因为READ有效，第一操作数的地址将载入MDR。如

果PC地址为0，因为ZEND有效，控制序列将跳回节拍0。结果是，控制序列中的节拍0将不确定地执行，从而导致URISC机器的动态停机。假设PC地址不是0，在节拍1，MDRout与MARin有效，第一操作数的地址从MDR移入MAR。在控制节拍的结尾，因为READ有效，第一操作数被载入MDR寄存器，在控制节拍2，第一操作数将从MDR移至R。

```
0. PCout,Zin,MARin,READ,ZEND
1. MDRout,MARin,READ
2. MDRout,Rin
3. PCout,Cin,PCin,MARin,READ
4. MDRout,MARin,READ
5. MDRout,COMP,Cin,Nin,MDRin,WRITE
6. PCout,Cin,PCin,MARin,READ
7. PCout,Cin,PCin,NNEND
8. MDRout,PCin
```

图6-14 URISC指令执行的控制序列

2) 节拍3到节拍5：读入第二操作数，从中减去第一操作数，结果置于MDR，如果结果为负，则置位N触发器。在控制节拍3，PC的地址递增，以指向当前指令的第二个字，其中保存了第二操作数的地址。递增后的地址被置于PC和MAR。通过使PCout、Cin、MARin和PCin有效，PC+1送入PC和MAR，在控制节拍之结尾，第二操作数的地址从存储器读入MDR。控制节拍4将第二操作数地址从MDR移入MAR。在控制节拍4的结尾，第二操作数的值从存储器读入MDR寄存器。在控制节拍5，通过使MDRout、Cin、COMP和MDRin有效，使第一操作数从第二操作数中减去，结果置于MDR。同时，如果结果为负且Nin有效，则N触发器被置位，完成这一切之后，在控制节拍5的结尾，结果写入第二操作数的内存地址，因为MAR未改变。

3) 第6拍：将分支地址读入MDR，其中Cin、PCout和PCin有效，PC递增以保存指令的第三个字的地址。同时第三个字的地址亦载入MAR。在控制节拍6的结尾，分支地址从存储器读入MDR。

4) 第7拍：递增程序计数器指向存储器中顺序的下一条指令。在NNEND有效的情况下，如果N在节拍5未被置位则控制序列转至节拍0，这只有当结果非负时才会发生。因此，一个非负结果导致序列中的下一条指令被执行，负的结果将导致控制转至第8拍。

5) 第8拍：将分支地址复制到程序计数器。当减法的结果为负时，下一条指令取自分支地址，而非下一条顺序指令。当第8拍结束时，通常控制又转入节拍0。

2. URISC时序

通过研究URISC指令执行的控制序列，我们注意到两个相连的动作几乎在所有的控制节拍里都会发生。例如，在节拍0，PC寄存器的内容必须移入MAR寄存器，然后进行存储器读。另外，注意到在每种情况下，在URISC数据单元内部有一个操作并且随后是存储器操作。因此采用了两相时钟的方案，第一相完成数据单元内部的数据传输，第二相完成存储器操作，这似乎比较合理。图6-15提出了URISC的一种时序。在控制节拍的开始处，PH1的下降沿更新控制信号，并使其保持一个控制节拍。PH2的下降沿引起URISC数据单元内部的一次数据传输。在控制节拍结尾处PH1的下降沿引起存储器操作，并且更新下一个控制节拍的控制信号。图6-16给出了前3个控制节拍的时序操作。

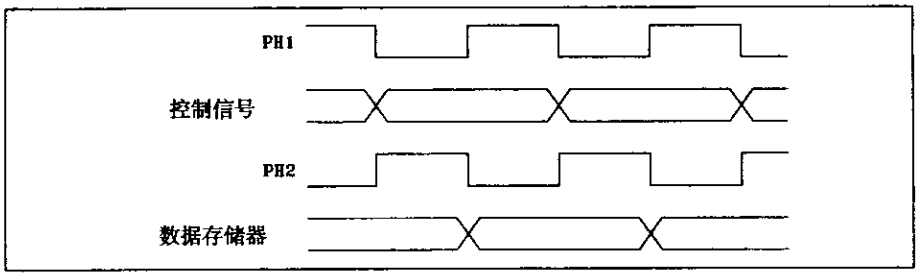


图6-15 URISC处理器的两相时钟

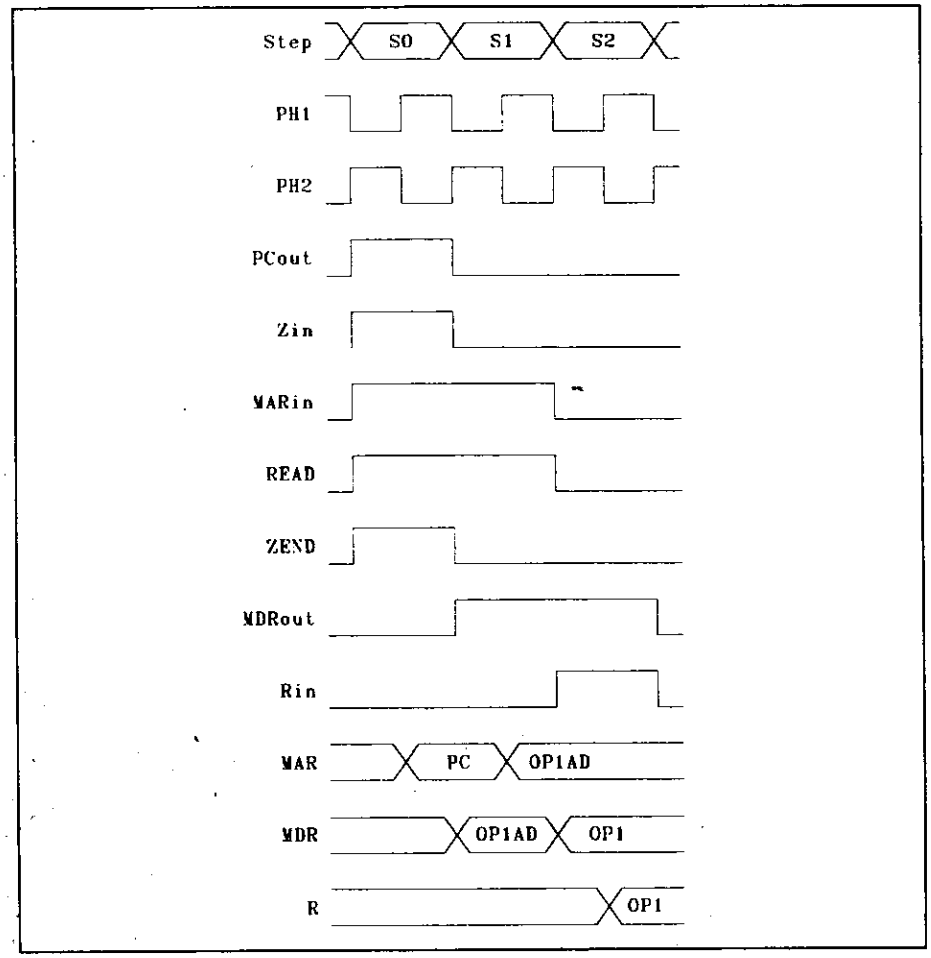


图6-16 节拍S0、S1和S2的控制信号

6.4.4 URISC系统

图6-17给出了使用URISC处理器的计算机系统的框图。包括一个RAM存储器模块和一个输入PORT模块，RUN输入用来启动或停止处理器。

信号DATA和ADDRESS是外部数据总线和地址总线。信号RDMR是使数据从RAM传入处理器的使能信号，信号WRITE是使数据从处理器传入RAM的使能信号。在基本URISC中加入

信号DAV，当输入PORT接收到新的数据项，要求处理器进行响应时，由输入PORT使该信号有效。只有在检测到RDIO信号时端口才将数据送入总线。

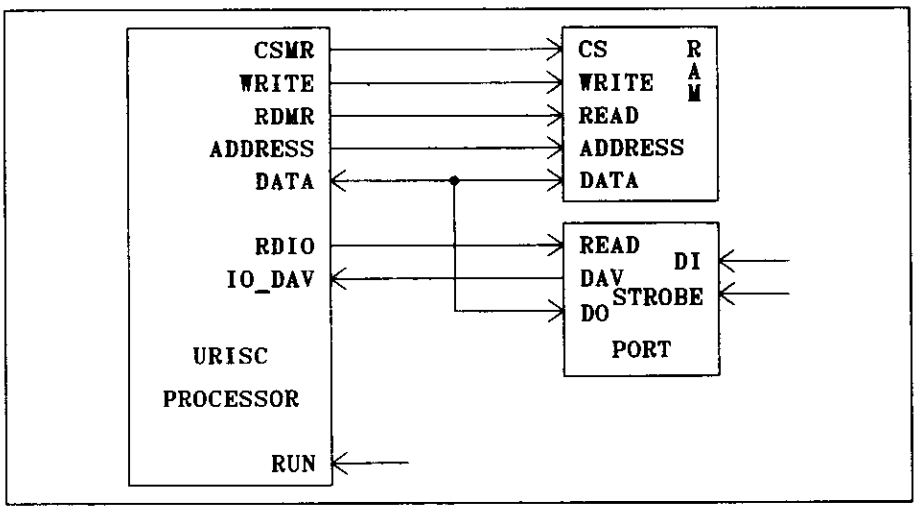


图6-17 URISC系统示例

6.4.5 在寄存器级的URISC设计

图6-18给出了URISC的寄存器级VHDL描述的框架。正如第5章所讨论的，CD ROM中的包集合SYSTEM_4包含了对一个四值逻辑系统MVL4（'X'，'0'，'1'，'Z'）的声明和函数，其中'X'解释为未知信号，'Z'解释为三态信号中的高阻态。

实体声明中的大部分类属声明是URISC中函数的时序延时，这里不再说明。ENABLE_DELAY是使能数据进入总线的延时，变量PER是时钟周期，DATA是接到RAM和输入PORT的外部数据总线，ADDRESS是连接到RAM的外部地址总线。类型WORD是8位向量，基类是MVL4。其他端口声明与图6-17系统框图中所示的一致。

结构声明包含了一系列URISC处理器内部使用的信号，其中一些，如COMP和ZEND，读者已经熟悉了。其他的将在后面定义。结构包括与URISC处理器中的每个寄存器和每个组合逻辑单元一一对应的块（block），以及与控制单元对应的块。一些块也将在后面定义。加法器块包含了对计算两个向量的进位加法的包函数的引用（reference）。

每个寄存器都是使用下降沿触发器设计而成的下降沿触发的寄存器。例如，图6-19给出了PC寄存器的完整设计。它被描述为“防护”（Guard）块，其“防护”条件是当PCin有效时被激活。PH2时钟从1变到0，当“防护”条件为真时，BUS_B被载入PC。无论“防护”条件是什么，如果PCout有效，PC值送入BUS_A。如果PCout无效，PC寄存器的输出被置为高阻态。其他寄存器有类似的说明，它们的设计作为练习留给读者（见习题6.13）。这里没有给出生成PH1与PH2的内部晶振，该晶振在RUN为高时启动。

6.4.6 URISC处理器的微码控制器

由于URISC的机器指令执行的分支点有两处，而且分支点都是在第0拍，所以微码控制器的地址产生逻辑实际上很简单，我们将使用带有并行载入全0功能的模9计数器。如果两者之

中任一分支情况为真，则并行载入全0。除非分支条件为真，否则计数器将从0计数到8，然后再回到0。如果分支条件为真，计数器将被载入全0，从而形成到节拍0的转移。图6-20给出了控制器的VHDL描述。这个描述可以直接转换成图6-10的硬件电路。

```
-- The entity declaration of the URISC processor
use work.all;use work.SYSTEM_4.all;
entity URISC is
  generic(ENABLE_DEL,DISBL_DEL,REG_DEL,ADD_DEL,PER,
          COUNT_DEL, ROM_DEL,OR_DEL,AND_DEL,INV_DEL,
          MUX_DEL: TIME);
  port(DATA: inout WORD:="ZZZZZZZZ";
       ADDRESS: inout WORD:="00000001";
       RUN,IO_DAV: in BIT; RDIO,WRITE: inout BIT;
       RDMR,CSMR: out BIT);
end URISC;
-- The architecture of the URISC -----------------------
use work.all;use work.SYSTEM_4.all;
architecture BEHAVIORAL of URISC is
  signal PC1,R,R_NOT,BUS_A,BUS_B,MDR1,MDR: WORD;
  signal PC: WORD:="00000001";
  signal Z,ZERO,ZIN,N,PH1,PH2,R_IN,MDR_IN,N_IN,MAR_IN,
         C_IN,CLK,CLEAR: BIT;
  signal COMP,MDR_OUT,PC_IN,PC_OUT,ZEND,NNEND,READ: BIT;
  signal C: BIT_VECTOR(3 downto 0);
begin
  PC_REG : block (PCIN and PH2='0'and not PH2'stable)
---Insert Code for Program Counter Register ----------
  end block;
  R_REG : block (R_IN='1'and PH2='0'and not PH2'STABLE)
---Insert Code for Register R ------------------------
  end block;
  -------- The Adder --------------------------------
  BUS_B <=  ADD(BUS_A,R_NOT,C_IN) after ADD_DEL;
  N_REG : block(N_IN= '1' and PH2='0'and not PH2'STABLE)
---Insert Code for Register N-------------------------
  end block;
  Z_REG : block(Z_IN= '1' and PH2='0'and not PH2'STABLE)
---Insert Code for Z Register ------------------------
  end block;
  MDR_REG : block((MDRIN and PH2='0'and not PH2'STABLE)
            or (READ and PH1='0' and not PH1'STABLE))
---Insert Code for Memory Data Register --------------
  end block;
  MAR : block (MAR_IN='1' and PH2='0' and not PH2'STABLE)
---Insert Code for Memory Address Register MAR -------
  end block;
  process
---Insert Code for Internal Two Phase Clocks ---------
  end process;
  -------- The Control Unit ------------------------
---Insert Code for the Control Unit
end BEHAVIORAL;
```

图6-18 URISC处理器的寄存器级描述的框架

PH1时钟从1变到0时，计数器COUNTER被载入。信号CLEAR定义了节拍0要求跳转的分支条件。条件Z和ZEND指明在PC寄存器中检测到了地址0，COUNTER将被不确定地置0，以

实现处理器的动态停机。条件（not N and NNEND）指明了减法的结果非负，而且控制器处在节拍7的末尾。要求的动作是返回节拍0，并且顺序执行存储器中的下一条指令。如果N为真，则执行节拍8，将分支地址载入PC引起向程序存储器中新地址的跳转。因为要开始执行下一条指令，COUNTER在第8拍结束时亦被清0，对于所有其他的情况，COUNTER递增，从而引起下一个控制节拍的执行。

```
---------The Program Counter Register -------------
-- PC is a negative edge triggered register--------
PC_REG : block (PCIN and PH2='0'and not PH2'stable)
begin
  PC1 <= guarded BUS_B after REG_DEL;
  BUS_A <= PC1 after ENABLE_DEL when PC_OUT='1' else
           "ZZZZZZZZ" after DISBL_DEL;
end block PC_REG;
```

图6-19 PC寄存器的设计

COUNTER的输出作为ROM的地址输入，无论何时COUNTER的值改变，将从ROM中读出一系列新的控制信号。在本设计中，控制信号可以直接在ROM的输出端使用，而不用载入MIR寄存器。这只有在ROM的驱动电路足够处理控制信号的负载时才可行。在大多数情况下，需要一个MIR或缓冲器（见习题6.17），该设计作为练习留给读者。

为了减少ROM字的位数，注意控制信号并非完全独立，例如，通过研究控制序列，可以注意到Zin和ZEND总是同时有效。因此，不是用ROM中独立的两个位来表示这两个控制信号，而是将Zin置为与ZEND相等。其他依赖关系在控制单元的LOGIC部分指出。

6.4.7 URISC处理器的硬连线控制器

硬连线单元的设计与微码单元的设计相比，结构性比较差，因为每个硬连线单元都必须定制设计。图6-21给出了控制硬件的寄存器级VHDL描述。一个更为结构化的描述在每个控制节拍都使用一个触发器（见习题6.18）。系统的设计方法参见第8章。

习题

6.1 学习第12章讨论的调度和分配问题，在调度和分配过程中的哪一步使用的是寄存器级描述而不是算法级描述。

6.2 假设一个VHDL数据流描述在一个防护块中包括如下两个并发语句：

```
B:block(not CLK'STABLE and CLK = '1')
begin
   X <= guarded ADD(Y,Z);
   R <= guarded ADD(Q,S);
end block B;
```

在VHDL模型里，这两条语句并发执行，但是对于实际硬件，可以并发执行也可以顺序执行。如果顺序执行，则可以共享一个加法器。假设信号Y、Z、D和S是数据输入，信号X和R是时钟触发寄存器，画出这两种方法的硬件框图。图中应包括多路器、加法器和寄存器。根据所需要的硬件和时钟周期比较并发和顺序两种实现方法。我们完成了哪些寄存器级设计活动？

```
---------The COUNTER -----------------------------
-- The Counter has a synchronous CLEAR ------------
COUNTER: block (PH1= '0' and not PH1'STABLE )
begin
  C <= guarded "0000" after COUNT_DEL when CLEAR='1' else
               INC_COUNTER(C) after COUNT_DEL;
end block COUNTER;
-------- The microinstructions ROM  ---------------------
ROM: process(C)
  type SQ_ARRAY is array(0 to 8,0 to 8) of BIT;
  constant MEM : SQ_ARRAY:=
-- 0       1    2    3     4     5    6    7     8   COLUMN
MDR_OUT,MAR_IN,N_IN,R_IN,PC_IN,ZEND,C_IN,WRITE,NNEND,micins
(('0',    '1', '0', '0',  '0',  '1', '0', '0',  '0'), --0
 ('1',    '1', '0', '0',  '0',  '0', '0', '0',  '0'), --1
 ('1',    '0', '0', '1',  '0',  '0', '0', '0',  '0'), --2
 ('0',    '1', '0', '0',  '1',  '0', '1', '0',  '0'), --3
 ('1',    '1', '0', '0',  '0',  '0', '0', '0',  '0'), --4
 ('1',    '0', '1', '0',  '0',  '0', '1', '1',  '0'), --5
 ('0',    '1', '0', '0',  '1',  '0', '1', '0',  '0'), --6
 ('0',    '0', '0', '0',  '1',  '0', '1', '0',  '1'), --7
 ('1',    '0', '0', '0',  '1',  '0', '0', '0',  '0'));--8
begin
  MDR_OUT  <=   MEM(INTVAL(C),0) after ROM_DEL;
  MAR_IN   <=   MEM(INTVAL(C),1) after ROM_DEL;
  N_IN     <=   MEM(INTVAL(C),2) after ROM_DEL;
  R_IN     <=   MEM(INTVAL(C),3) after ROM_DEL;
  PC_IN    <=   MEM(INTVAL(C),4) after ROM_DEL;
  ZEND     <=   MEM(INTVAL(C),5) after ROM_DEL;
  C_IN     <=   MEM(INTVAL(C),6) after ROM_DEL;
  WRITE    <=   MEM(INTVAL(C),7) after ROM_DEL;
  NNEND    <=   MEM(INTVAL(C),8) after ROM_DEL;
end process ROM;
LOGIC: block
begin
  ZIN      <=  ZEND;
  ZERO     <=  NOR_BITS(BUS_B) after OR_DEL;
  CLEAR <= (Z and ZEND) or (not N and NNEND) or
           (C = "1000") after AND_DEL+OR_DEL;
  PC_OUT   <=  not MDR_OUT after INV_DEL;
  READ     <= MAR_IN;
  COMP     <= N_IN;
  MDR_IN   <= N_IN;
  RDMR <= READ and not MVL4toBIT(ADDRESS(7)) after AND_DEL;
  RDIO <= READ and MVL4toBIT(ADDRESS(7)) after AND_DEL;
  CSMR     <= not MVL4toBIT(ADDRESS(7)) after INV_DEL;
end block LOGIC;
```

图6-20 URISC的微码控制单元

6.3 一个数据流VHDL描述包括下面的并发语句：

```
C <= A;
D <= B;
```

假设A、B、C和D都是寄存器，它们之间的传输通过一条或多条数据总线完成，一个并发操作需要两条总线；一个顺序操作只需一条总线。在TTL数据手册中找出实现8位宽带总线的芯片，比较这两种情况下所需的硬件和传输时间。我们完成了哪些寄存器级

设计活动?

```
-------- The COUNTER ------------------------------
-- The Counter has a synchronous CLEAR ------------
COUNTER: block (PH1= '0' and not PH1'STABLE )
  begin
    C <= guarded "0000" after COUNT_DEL
                 when (CLEAR='1'or C="1000") else
         INC_COUNTER(C) after COUNT_DEL;
  end block COUNTER;
  --Hard Wired Control Unit
  --Decoder
  --First Stage Decoding
  ST0 <= not C(2) and not C(1) and not C(0) after AND_DEL;
  ST1 <= not C(2) and not C(1) and C(0) after AND_DEL;
  ST2 <= not C(2) and C(1) and not C(0) after AND_DEL;
  ST3 <= not C(2) and C(1) and  C(0) after AND_DEL;
  ST4 <= C(2) and not C(1) and not C(0) after AND_DEL;
  ST5 <= C(2) and not C(1) and C(0) after AND_DEL;
  ST6 <= C(2) and C(1) and not C(0) after AND_DEL;
  ST7 <= C(2) and C(1) and C(0) after AND_DEL;
  --Second Stage Decoding
  ST07 <= ST0 or ST7 after OR_DEL;
  ST25 <= ST2 or ST5 after OR_DEL;
  ST36 <= ST3 or ST6 after OR_DEL;
  ST57 <= ST5 or ST7 after OR_DEL;
  ST78 <= ST7 or C(3) after OR_DEL;
  --Control Signals
  PC_OUT <= (ST07 or ST36) and not C(3)
            after (OR_DEL + AND_DEL);
  C_IN <= ST36 or ST57 after OR_DEL;
  PC_IN <= ST36 or ST78 after OR_DEL;
  MAR_IN <= not(ST25 or ST78) after (OR_DEL + INV_DEL);
  MDR_OUT <=not PC_OUT after INV_DEL;
  READ <= MAR_IN; COMP <= ST5; N_IN <= ST5; MDR_IN <= ST5;
  WRITE <= ST5; R_IN <= ST2; ZIN <= ST0; ZEND <=ST0;
  NNEND <= ST7;
  --Register and Counter Controls
  ZERO <= NOR_BITS(BUS_B) after OR_DEL;
  CLEAR <= (Z and ZEND) or (not N and NNEND)
           after AND_DEL+OR_DEL;
  RDMR <= READ and not MVL4toBIT(ADDRESS(7)) after AND_DEL;
  RDIO <= READ and MVL4toBIT(ADDRESS(7)) after AND_DEL;
  CSMR <= not MVL4toBIT(ADDRESS(7)) after INV_DEL;
```

图6-21 URISC硬连线控制单元的VHDL描述

6.4 此问题需要考虑总线时序的某些基本元素。图6-22给出了一个总线的框图及相关控制逻辑。选择信号SEL控制总线上数据DA或DB的进入，当CLK上升时若R_EN为1，则数据从总线上复制下来，信号SEL和R_EN由触发器（由CLK定时）产生，它们轮流驱动组合逻辑块（CLs）。画出具有CLK、SEL、R_EN和BUS的总线时序图。什么是最关键的时序要求？包括寄存器接收数据的建立时间。

6.5 为上题描述的总线系统产生一个VHDL模型，包括总线时序冲突的断言检测，通过仿真验证模型。

6.6 本题使用图6-6所用时序关系的寄存器系统，对于所选技术，如TTL、ECL或CMOS，根

据相应器件的数据手册决定系统延时类属的实际值。首先使用能满足时序约束的输入值来仿真系统，然后使用与时序约束冲突的输入数据。

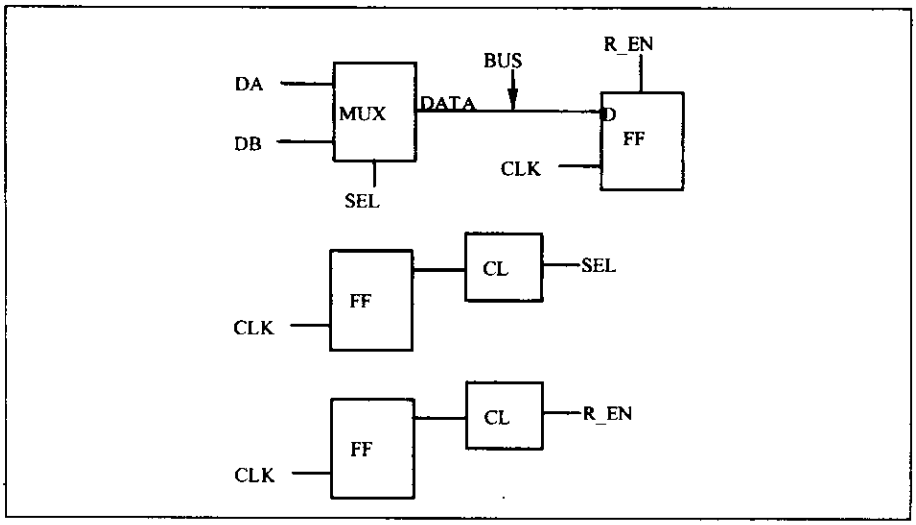


图6-22　时序总线框图

6.7　使用第4章定义的ALU基本部件（原语）重新设计寄存器系统（时序关系如图6-6）。

6.8　编写用URISC机实现两个无符号二进制数相乘的程序。

6.9　URISC机是否通用还存在争议，即给出足够指令它能否实现任何运算。URISC是否能实现布尔逻辑操作？通过分析来支持你的答案。

6.10　为URISC机画出从步骤0到8的两个完整的时序图，图6-16给出了从步骤0到2的示例图。一个时序图说明减法结果为负的情况，另一个时序图则说明结果为非负的情况。

6.11　决定URISC机的最大允许时钟频率。

a) 为URISC处理机的延时类属与最大允许时钟频率建立一个表达式，分别对微程序控制单元和硬连线控制单元进行分析。

b) 对于所选择的技术如TTL、ECL或CMOS，决定延时的实际值，并计算两种情况下最大允许时钟频率。

c) 对模型仿真验证结果。

6.12　列出微程序控制单元和硬件控制单元的优缺点。

6.13　完成图6-18给出的URISC处理器中以下部件的VHDL规范。

a) 寄存器R

b) 寄存器N

c) 寄存器Z

d) 存储数据寄存器

e) 存储器地址寄存器

f) 两相时钟

6.14　使用习题6.13的结果，完整地测试URISC处理器模块，如图6-18给出的框架。

6.15　设计并测试图6-17给出的URISC系统的RAM和STROBE模块。

6.16 把URISC系统部件(图6-17)集成为一个完全可操作的URISC系统的VHDL模型。使用习题6.14的URISC处理器模块和习题6.15的RAM及STORBE模块，使用图6-11所示的程序来仿真系统。

6.17 在图6-20的微代码控制单元中，控制信号由ROM直接输出，这只有在ROM输出可以驱动所有控制信号负载时才可能。通常ROM的输出只能驱动一个门载入。因此，在ROM输出和电路控制点之间需要有缓冲寄存器。对微码控制单元修改VHDL规范模型，加入存储指令寄存器(MIR)作为ROM输出的缓冲寄存器。讨论它与定时及系统时钟的关系。

6.18 修改URISC(图6-21)的硬连线控制单元，为每个控制步加入一个触发器，说明这种控制器的设计程序为什么比图6-21的设计过程更具有结构化的特点。

第7章 门级和ASIC库建模

本章讨论如何在门级对设计进行精确建模，以及在该抽象层次进行的设计活动。精确门级描述必须对时序进行精确建模，必须考虑各种电路的输出阻抗，必须检查错误条件。这种模型对于学习逻辑电路中的险态和竞争问题非常有用。对大型门级模型的仿真是计算密集型的，因此，模型必须高效。VHDL语言的配置管理结构为创建和维护与门级建模相关的设计数据库提供了一种十分有效的方法。

ASIC的设计过程需要ASIC库中单元的精确模型。这对于正确地对单元延时建模尤其重要。本章讨论解决这个问题的各种方法。

7.1 精确门级建模

本节讨论一个精确的门级模型的组成。我们用一个包括2输入或门的例子来介绍许多需要的特征。图7-1给出了2输入或门的三个构架，任何有VHDL基础知识的人都可以完成它。构架DELTA_DEL实现门的功能，指定delta延时。Delta作为单位延时。这种类型的模型只有在验证逻辑电路的功能时有用。构架FIXED_DEL对器件的传输延时建模，但并非通用模型。要改变延时，必须改变构架代码中的延时值。尽管这在门数少的情况下很灵活，但对于大型门级模型，如上千门，该方法是不实际的。构架GNR_DEL通过使用门的类属延时解决了这个问题。类属延时在接口描述中声明，在门初始化时其值被确实。

```
entity OR2 is
  port(I1,I2: in BIT; O: out BIT);
end OR2;

architecture DELTA_DEL of OR2 is
begin
  O <= I1 or I2;
end DELTA_DEL;

architecture FIXED_DEL of OR2 is
begin
  O <= I1 or I2 after 3 ns;
end FIXED_DEL;

entity OR2G is
  generic(DEL: TIME);
  port(I1,I2: in BIT; O: out BIT);
end OR2G;

architecture GNR_DEL of OR2G is
begin
  O <= I1 or I2 after DEL;
end GNR_DEL;
```

图7-1 三个基本OR门模型

在许多例子中，设计者为器件模型保留三套延时数据：最小、最大和典型值。对于单纯的功能验证，也可以是0，即delta延时。如果有这样一个能返回这四个值中的一个的函数就好了。而且，在某些建模情况之下，希望使这种选择是全局的，即所有器件模型使用同样的延时模式。图7-2给出了它的实现。在包集合TIMING_CONTROL中，声明了类型TIMING，用来确定四种延时模式。然后，声明了常量TIMING_SEL，其缺省值为典型（TYP），包集合中的函数T_CHOICE根据TIMING_SEL的值选择正确的延时模式。使用了函数T_CHOJCE的或门模型在图7-2中的包集合代码的后面。包集合的内容通过语句use work.TIMING_CONTROL.all对于所有的门模型都是可见的。要改变全局延时选项，必须将常量TIMING_SEL赋为新值，并重新分析该包集合。当TIMING_SEL的值改变后，所有的模型都使用新的延时值。

```
package TIMING_CONTROL is
  type TIMING is (MIN,MAX,TYP,DELTA);
  constant TIMING_SEL: TIMING := TYP;
  function T_CHOICE(TIMING_SEL: TIMING; TMIN,TMAX,TTYP: TIME)
    return TIME;
end TIMING_CONTROL;
package body TIMING_CONTROL is
  function T_CHOICE(TIMING_SEL: TIMING; TMIN,TMAX,TTYP: TIME)
    return TIME is
  begin
    case TIMING_SEL is
      when DELTA => return 0 ns;
      when TYP => return TTYP;
      when MAX => return TMAX;
      when MIN => return TMIN;
    end case;
  end T_CHOICE;
end TIMING_CONTROL;
use work.TIMING_CONTROL.all;
entity OR2_TV is
  generic(TMIN,TMAX,TTYP: TIME);
  port(I1,I2: in BIT; O: out BIT);
end OR2_TV;

architecture VAR_T of OR2_TV is
begin
  O <= I1 or I2 after T_CHOICE(TIMING_SEL,TMIN,TMAX,TTYP);
end VAR_T;
```

图7-2 时序控制包集合

在许多建模情况下，针对不同类型的门甚至单个的门可能会有不同的延时值，以后，在讨论门级建模的配置声明时，需要解决这个问题。

7.1.1 不对称定时

前面的延时建模方法对于学习基本的时序很有用；然而，从制造商的数据手册中会发现，实际的门延时是用不同的方式指出的。图7-3给出了一个通用的不对称（*asymmetric*）时序模型。该模型具有输入I和输出O，和两个延时参数：TPLHio和TPHLio，分别定义为：

TPLHio。输出O响应输入I之改变，从低（'0'）到高（'1'）改变所需的时间。

TPHLio。输出O响应输入I之改变，从高（'1'）到低（'0'）改变所需的时间。

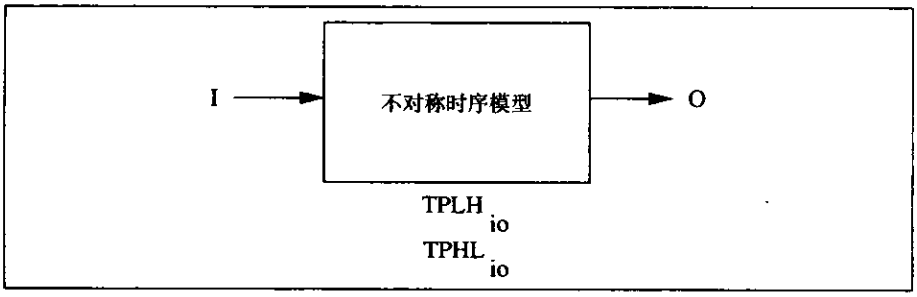


图7-3 不对称时序模型

注意对TPLHio，输出O响应输入I的变化，从低到高转换。输入I上的变化可以或者从低到高（非反相门）或者从高到低（反相门）。一个器件的TPLHio和TPHLio不同主要是因为当驱动到高或驱动到低时，输出源阻抗不同。

图7-4是2输入或门的模型，其中考虑到了TPLH和TPHL可能不同。该模型使用两个内部变量OR_NEW和OR_OLD来暂时保存函数当前和以前的过程调用的状态。这两个变量进行比较以确定是否安排一个输出值（考虑到模块的效率），如果安排了，使用哪个延时（TPLH和TPHL）。

```
entity OR2GV is
  generic(TPLH,TPHL: TIME);
  port(I1,I2: in BIT; O: out BIT);
end OR2GV;

architecture VAR_DEL of OR2GV is
begin
  process(I1,I2)
    variable OR_NEW,OR_OLD:BIT;
  begin
    OR_NEW := I1 or I2;
    if OR_NEW = '1' and OR_OLD = '0' then
       O <= OR_NEW after TPLH;
    elsif OR_NEW = '0' and OR_OLD = '1' then
       O <= OR_NEW after TPHL;
    end if;
    OR_OLD := OR_NEW;
  end process;
end VAR_DEL;
```

图7-4 可变延时的OR门模型

延时函数可以直接实现延时。图7-5给出实体OR2GV的构架FUNC_DEL（原来在图7-4中建模）。该构架调用包集合DELAY中的函数VER_DEL以确定是使用TPLH或是使用TPHL（见图7-7）。构架FUNC_DEL是一个简明模型。然而，它比构架VAR_DEL低效，因为只要输入变化，它就会安排一个输出事务，即使新的输出与旧的相同。构架VAR_DEL只有在旧的和新的输出值不同时，才会安排一个输出事务。

7.1.2 负载敏感延时建模

在实际电路中，电路的延时是电路负载的强函数（strong function）。在测量该负载并相应

调整电路延时模型方面已经做了大量的努力。本节介绍一些对负载敏感延时模型建模的方法。

```
use work.DELAY.all;
architecture FUNC_DEL of OR2GV is
begin
  process(I1,I2)
    variable OR_NEW:BIT;
  begin
    OR_NEW := I1 or I2;
    O <= OR_NEW after VAR_DEL(TPLH,TPHL,OR_NEW);
  end process;
end FUNC_DEL;
```

图7-5 使用函数来选择延时

1. 扇出敏感延时

对于印制电路板上的TTL电路，由于输入电容引起的延时是最重要的因素。通常，由金属线所引起的延时可以忽略。在本章的后面，将讨论根据给定电路的扇出如何反向标注输出延时。然而，这里提供一种不同的方法，门模型允许每个门实例在仿真的开始测量出自己的扇出。图7-6a给出了这种方法。在其一般输入（I1，I2）和输出（O）之外，模型有负载端口I1L、I2L和OL用来在门模型之中传递负载信息。I1L和I2L实际上是输出，OL实际上是输入。图7-6b给出了负载信息是如何在门之间进行传递的。驱动门的输入OL连到被驱动门的输出IL。每个驱动门在IL端口上输出一个整数值（如，1）。这些IL信号在求和交点处进行求和，OL从求和交点处读出，用于延时方程。

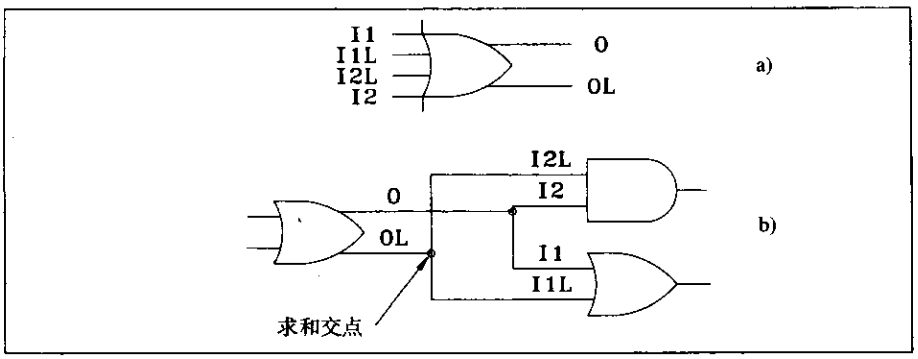


图7-6 扇出敏感门延时

为了进行求和，我们使用作用于整数信号值的特殊的判决函数（Resolution Function）。图7-7a中的包集合DELAY包括函数S_FANOUT。声明了的整数子类型FINT，其判决函数为S_FANOUT。图7-7b给出了一个2输入或门的门模型，但没有给出定义门的互连的结构模型。注意到在接口描述中，I1L、I2L和DL都是子类型FINT，而且声明了两个类属：DEL_VMIN和I_LOAD。在结构体中，InputLoad Process的进程。只在仿真开始时运行一次，它将I_LOAD的值赋给I1L和I2L，用来计算DEL_UNIT*OL的积，将结果赋值给DELAY，即仿真时门延时的值。OL是子类型FINT的判决（resolved）信号，因为判决函数的动作，OL等于门负载。类属DEL_VNST和I_LOAD的缺省值是1ns和1，当然，当门被初始化时，它们将被赋

与其他的值。

```
package DELAY is
  type INT_VECT is array (NATURAL range <> ) of INTEGER;
  function S_FANOUT(S: INT_VECT) return INTEGER;
  subtype FINT is S_FANOUT INTEGER;
  function VAR_DEL(TPLH,TPHL: TIME; FNEW: BIT) return TIME;
end DELAY;
package body DELAY is
  function S_FANOUT(S: INT_VECT) return INTEGER is
    variable SUM: INTEGER := 0;
  begin
    for I in S'RANGE loop
      SUM := SUM + 1;
    end loop;
    return SUM;
  end S_FANOUT;
  function VAR_DEL(TPLH,TPHL: TIME; FNEW: BIT)
    return TIME is
  begin
    if FNEW = '0' then
      return TPHL;
    else
      return TPLH;
    end if;
  end VAR_DEL;
end DELAY;
use work.DELAY.all;
```

a)

```
entity ORF is
  generic(DEL_UNIT: TIME := 1 ns;I_LOAD: INTEGER := 1);
  port(I1,I2: in BIT; I1L,I2L: out FINT;
       O: out BIT; OL: in FINT);
end ORF;
architecture FANOUT of ORF is
 signal DELAY: TIME:= 0 ns;
begin
  process      ----Input Load Process
  begin
    I1L <= I_LOAD;
    I2L <= I_LOAD;
    DELAY <= DEL_UNIT*OL;
    wait;
  end process;
  O <= I1 or I2 after DELAY;
end FANOUT;
```

b)

图7-7　扇出敏感延时模型

2. 输入延时模型

图7-4中的构架VAR_DEL是一个“孤立”门的不错的模型。然而，它并未考虑两个因素，1) 互连引进的延时；2) 由当前建模的门驱动的负载引起的延时。我们首先考虑互连效果。图7-8a显示了由AND门通过金属或多晶硅驱动的OR门。注意这里引起了两种延时：1) 延时DEL1，是从AND门的输出到信号扇出点的延时；2) 延时DEL2，是从扇出点到OR门的输入的

的延迟。一般的方法将（DEL1+DEL2）的和作为OR门的输入I2的单个输入延时。如果该方法用于系统中所有的门，则能对整个系统的互连延时的效果进行正确的建模。图7-8b给出OR门的输入延时的框图。每个输入都赋与了一个延时值。OR函数本身被认为是无延时的。门传输延时由输出延时建模。

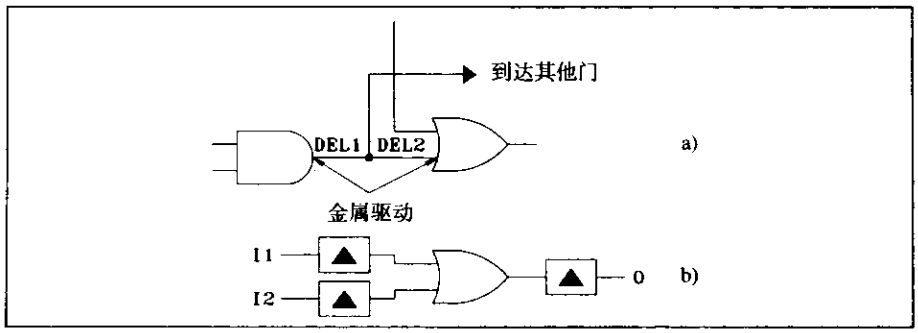


图7-8 输入延时建模

图7-9给出了输入延时模型的VHDL代码。注意图中第一个进程对输入延时建模。对每个输入i，都有类属TPLHi和TPHLi。这些类属用来在实体声明时确定模型输入延时。这些值产生于一个反标过程中，该过程在特定的门布局之下测量信号通路的长度，将该长度乘以金属或多晶硅材料的长度因子pf/unit，并将驱动门的输入电容加入其中，得到的电容值和建模电路的特定因子ns/pf相乘。这样对门的第i个输入计算出的值成为在模型中赋给TPLHi和TPHLi的值。

3. 输出延时模型

基于输出延时的模型也被普便应用。在EIA时序和环境规范中使用输出负载信息的延时方程被描述为：

$$T'_{pd} = (T_{pd} + l_{delay}(C_{load})) * K_v * K_t$$

其中：

T_{pd}是用在模型输出赋值中的被计算的传输延时，单位是ns。

T_{pd}是厂商提供的传输延时，单位是ns。

C_{load}是反标负载值（计算出的容性负载），单位是pf。

l_{delay}是该技术的电容与时间延时的函数，包括C_{ref}——厂商对端口类型的指定负载。

K_v是电压减载因子

K_t是温度减载因子

如果对l_{delay}使用线性模型，则：

$$l_{delay}(C_{load}) = (C_{load} - C_{ref}) * K_c$$

其中：

C_{load}是被输出驱动的总负载，单位为pf。

C_{ref}是数据手册提供的负载值，单位为pf。

K_c是直线电容负载因子，单位为time/pf。

```
entity OR2GI is
  generic(TPLH1,TPHL1,TPLH2,TPHL2,TPLHO,TPHLO: TIME);
  port(I1,I2: in BIT; O: out BIT);
end OR2GI;

architecture VAR_DEL of OR2GI is
  signal I1_D,I2_D: BIT;
begin
  process(I1,I2)
  begin
    if I1'EVENT then
      if I1 = '0' then
        I1_D <= I1 after TPHL1;
      else
        I1_D <= I1 after TPLH1;
      end if;
    end if;
    if I2'EVENT then
      if I2 = '0' then
        I2_D <= I2 after TPHL2;
      else
        I2_D <= I2 after TPLH2;
      end if;
    end if;
  end process;
  process(I1_D,I2_D)
    variable OR_NEW,OR_OLD:BIT;
  begin
    OR_NEW := I1_D or I2_D;
    if OR_NEW = '1' and OR_OLD = '0' then
      O <= OR_NEW after TPLHO;
    elsif OR_NEW = '0' and OR_OLD = '1' then
      O <= OR_NEW after TPHLO;
    end if;
    OR_OLD := OR_NEW;
  end process;
end VAR_DEL;
```

图7-9 输入延时模型的VHDL代码

注意到作为反标的结果，l_{delay}的值可能为负；然而，T_{pd}的结果值被限制为非负。C_{load}的值可以在反标过程中测量，由于金属或多晶硅及每个驱动门的输入电容的影响，结果可能是各种效果之和。

7.1.3 ASIC单元延时建模

图7-10给出了用于ASIC单元建模的延时模型，这是在“*Synopsys Library Compiler Reference Manual*”中给出的，是前面讨论过的输出模型的改进。计算出的延时（D_{TOTAL}）是门的输入引脚和下一个门的输入引脚之间的延时，D_{TOTAL}由下式给出：

$$D_{TOTAL} = D_I + D_S + D_T + D_C$$

其中：

D_I——门的固有延时，与任何特定的实例无关。

D_S——由输入信号的上升时间（ramp time）引起的坡度延时（Slope delay）。

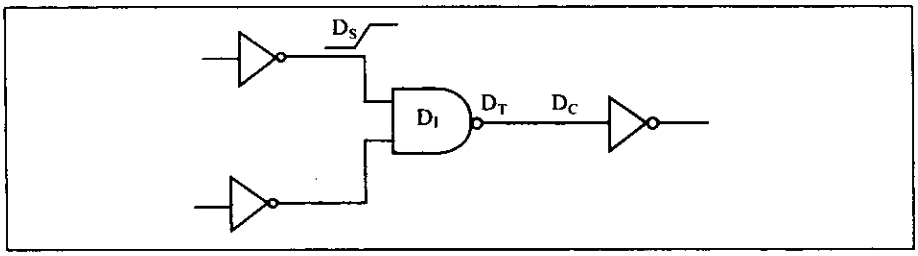


图7-10 单元延时模型

D_T——由输出引脚的负载引起的转换延时。

D_C——输入引脚的连接介质延时（线延时）。

1) **固有延时（D_I）**。一个电路元件的固有延时（D_I）是总延时中与电路元件的用法无关的延时。这部分是从电路元件的输出引脚到输入引脚的固定延时（或零负载时的延迟）。固有延时的常量值以浮点数保存在负载引脚的时序组（timing group）之中，它有两个属性：

```
intrinsic_rise : float;
intrinsic_fall : float;
```

2) **坡度延时（D_S）**。坡度延时是由输入信号的坡度引起的单元延时。正常情况下，库单元用一个指定的上升时间（一般为1.5ns）来确定从输入引脚到输出引脚的延时。然而在一些工艺技术中，该延时是上升时间的强函数。上升时间是由坡度敏感因子表示的。该因子说明了输入电压开始上升，但未达到通道开始导通的门限水平时的时间。在时序组（timing group）中定义该驱动引脚的属性为：

```
slope_rise
slope_fall
```

根据输入变化的方向，这些属性之一与输入上升时间相乘以得到D_S的值。

3) **转换延时（D_T）**。转换延时是使输出引脚改变状态所需之时间。是电路的源电阻（R_{DRIVER}）和负载电容的函数。负载电容是被驱动门的输入电容（C_{PINS}）与互连电容（C_{WIRE}）的和。这样，转换延时如下给出：

$$D_T = R_{DRIVER} * (C_{WIRE} + C_{PINS})$$

源电阻保存在驱动引脚时序组之中，作为浮点数使用下面两个属性。

```
rise_resistance : float ;
fall_resistance : float ;
```

C_{PINS}可以由被驱动门的已知输入电容测量出来。

该模型被Synopsys直接用于综合之后的延时计算。因而没有制造信息要反标。这样，必须估计C_{WIRE}，线长度根据电路中实际的扇出引脚个数和wire_load组之中fanout_length的规范进行计算。估计值再乘电容因子，该因子在wire_load组中定义为电容。如果在运行时没有指定wire_load组，则C_{WIRE}的值为0。

4) **连接延时（D_C）**。即使在输出的驱动门已经切换之后，新值经过互连到达门的输入也需要时间。互连可以用分布式RC电路建模。图7-11给出了用于Synopsys系统中分布式RC电路的三种情况下的类属：

最好的情况下，互连延时为：

$$D_C(best) = R_{WIRE}(C_{WIRE} + C_{PIN}) = 0$$

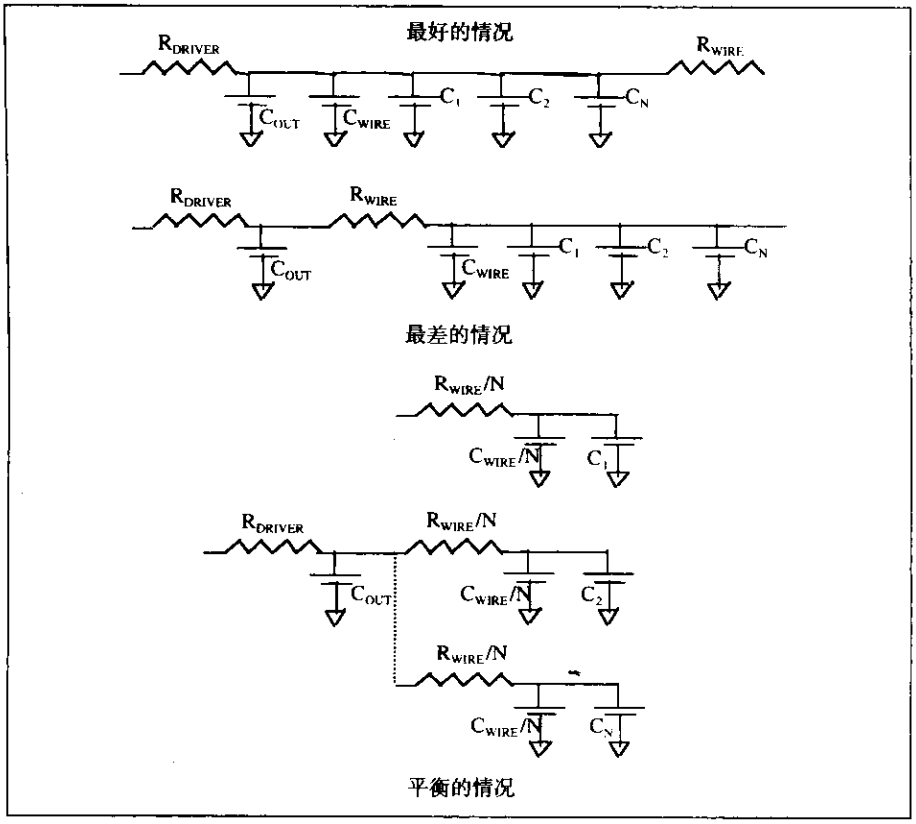


图7-11 Synopsys互连延时模型

对于这种情况，R_{WIRE}被认为是“线的尾部端点”，因为它与具有高输入阻抗的门串联，所以不影响RC延时。

最坏的情况下，互连延时为：

$$D_C(worst) = R_{WIRE}(C_{WIRE} + C_{PIN})$$

因为这时R_{WIRE}是“线的头部端点”，所以其全部影响都考虑了。

在平衡的情况下，电线电阻的由互连线的分开的相等分支产生。

$$D_C(balance) = R_{WIRE}/N(C_{WIRE}/N + C_{PIN})$$

R_{WIRE}是电路中的线电阻，由wire_load模型确定。线长度由全局估计函数计算，其参数是被估算的电路的扇出引脚数。该估计值除以电阻因子，该因子在wire_load group中定义为电阻。

例

在ASIC中，一个反相器驱动扇出数为4的电路。每个门输入具有1.5的输入电容，线电容是4.5。电路中线电阻是可忽略的。反相器的输入I由图7-12中所示的波形驱动。

下述数据是为反相器定义的。

Intrinsic rise : 0.5
Intrinsic fall : 0.4
Rise resistance : 0.15
Fall resistance : 0.07
Slope sensitivity factor : 0.03

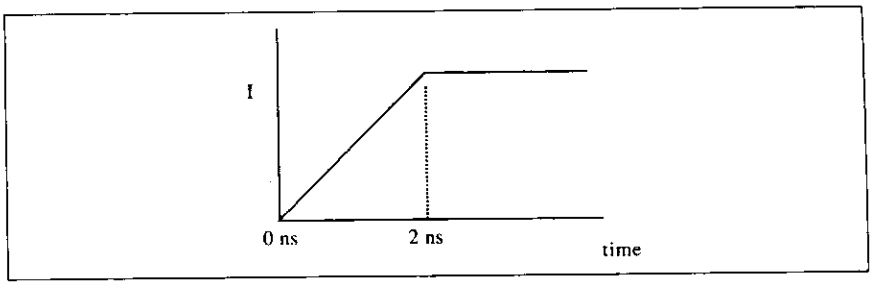


图7-12 反相器的输入波形

计算从反相器的输入到被驱动门的输入的总延时。

D_T=0.4ns, D_S=(0.03)(2ns)=0.06ns, D_T=0.07(4.5+4(1.5))=0.735ns, D_C=0所以D_{TOTAL}=1.195ns。

库元件的延时数据用工艺库的表示方法进行表示。图7-13给出了库LSI 10K中反相器的一个表示。延时数据给出的是引脚Y（输出）相对于引脚A的延时。首先，输出引脚的函数以A'后跟延时数据的形式给出。对于这个反相器，没有提供坡度敏感因子。假设存在一个特定因子，而且坡度敏感延时很小，包含在固有延时之中。

```
cell(IVDA) {
  area : 1 ;
  pin(A) {
    direction : input ;
    capacitance : 1.3 ;
  }
  pin(Y) {
    direction : output ;
    function : "A'" ;
    timing() {
      intrinsic_rise : 0.49 ;
      intrinsic_fall : 0.37 ;
      rise_resistance : 0.1438 ;
      fall_resistance : 0.0623 ;
      related_pin : "A" ;
    }
    internal_power() {
      related_pin : "A" ;
      rise_power(output_by_cap_and_trans) {
        values("2.5, 3.7, 4.3") ;
      }
      fall_power(output_by_cap_and_trans) {
        values("2.6, 3.8, 4.4") ;
      }
    }
  }
}
```

图7-13 LSI 10K反相器的工艺库表示方法

7.1.4 延时的反向标注

在为ASIC售出而进行布局后仿真时（见图9-29），使用真实的延时很重要。ASIC 生产厂商有工具可以测量线长，从而计算真实的互连延时。这些延时必须插入ASIC的门级仿真模型。这种延时的一种事实上的标准，称作标准延时格式（SDF）。首先由Cadence针对其工具开发出来，后来为其他厂商所采用。一个标准化的版本由1995 Open Verilog International（OVI）发布，版本3.0中的SDF定义引用SDF的下述特征：

“标准延时格式（SDF）文件保存由EDA工具产生的定时数据，可以在设计过程的任何阶段使用。SDF的文件用与工具无关的方式表示，包括：

- 延时：模块路径，器件，互连和端口。
- 时序检查：建立、保持、恢复、移动、扭曲、宽度、周期及无变化。
- 时序限制：路径、扭曲、和及差。
- 时序环境：将要进行定时的环境。
- 增量和绝对延时。
- 条件和无条件的模块路径延时和时序检查。
- 设计/实例专用或类型/库专用数据。
- 缩放比例、环境和技术参数。

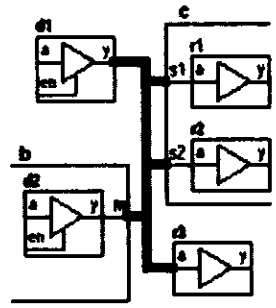
在整个设计过程中，可以使用几种不同的SDF文件。这些文件中的一些可以包括预布局延时信息。其他的可以包含路径限制和布局后延时数据。”

图7-14给出了D触发器的SDF延时，首先，给出一个“CELL”定义。一个CELL可以是低级元件，和这里情况一样；或者可以是该设计的某个区域。这里，CELL是一个D触发器的实例top/b/c（层次名）。它的延时是ABSOLUTE（绝对）值。当反标时，可以替换原来模型中的值。也可以使用INCREMENTAL延时，在这种情况下，反标时，它们与原来的模型值相加。IOPATH确定从输入（时钟上升沿）到一个输出（q）的延时。输入上升沿时钟是一个条件时钟输入。SDF也包含一个更加通用的形式COND(x)，当x为真时对一条路径的说明产生效果。延时值包含两组最小、典型和最大值的数组。这些值和用户控制的TIMESCALE值（如100ps）相乘，以得到实际的延时值。第一组三个值是对应输出上升的，它们应包括固有、坡度和转换延时的效果。第二组的值是针对输出下降的。PORT命令表示了CLR输入上的输入延时；同样，针对上升和下降也有分别的值。这是表示互连延时的一种方法，和图7-9中的模型一起使用。

```
(CELL
  (CELLTYPE "DFF")
  (INSTANCE top/b/c)
  (DELAY
    (ABSOLUTE
      (IOPATH (posedge clk) q (2:3:4) (5:6:7))
      (PORT clr (2:3:4) (5:6:7))
    )
  )
  (TIMINGCHECK
    (SETUPHOLD d (posedge clk) (3:4:5) (-1:-1:-1))
    (WIDTH clk (4.4:7.5:11.3))
  )
)
```

图7-14　D触发器的SDF定时

作为选择，点到点之间的互连延时可以用图7-15中的INTERCONNECT结构来表示。注意它涉及了三个层次而且延时与上升和下降无关。列出的延时值与TIMESCALE相乘得到实际的延时值。这个用这种方式测量的延时值对应于ASIC延时模型的连接延时。有关SDF的更多信息，读者请参考OVI网站。



```
ABSOLUTE
     (INTERCONNECT d1.y c.r1.a (0.01:0.02:0.03))
     (INTERCONNECT d1.y c.r2.a (0.03:0.04:0.05))
     (INTERCONNECT d1.y r3.a (0.05:0.06:0.07))
     (INTERCONNECT b.d2.y c.r1.a (0.04:0.05:0.06))
     (INTERCONNECT b.d2.y c.r2.a (0.02:0.03:0.04))
     (INTERCONNECT b.d2.y r3.a (0.02:0.03:0.04))
   )
)
```

图7-15 点到点互连延时的定义

7.1.5 VITAL：库元素的 VHDL模型的生成标准

由于Cadence Corporation开发了SDF和Verilog，它们领导开发ASIC库元素的Verilog模型是很自然的。这些模型的实现在它们的仿真程序中运行速度极快，这样一个综合后的ASIC的Verilog描述是很高效的表示。工具厂商在开发ASIC库元素的高效VHDL模型上较慢。这样，在许多情况下，可以发现这么一种有趣的现象，要综合的ASIC的行为级模型是用VHDL写的，这是因为其强大的建模特征。但是综合后电路的结构模型却是用Verilog生成的，以便提供高效的综合后仿真。

IEEE对这种情况的反映是成立VHDL Initiative Towards ASIC Library(VITAL)委员会。根据VITAL LRM："VITAL建立的目的可以用一句话总结：利用现存的模型开发方法，加速用VHDL对成品质量的ASIC单元仿真库的开发。……VITAL利用下述模型开发方法达到目的：a) 标准延时格式；b) VHDL技术组织Std_Timing包中的特定元素，和由Ryan & Ryan提供的专业化时序和行为技术；c) 多种ASIC库和用SDF延时实现的工具。VITAL使用了很多关于原语和用Verilog描述的时序模型。特别地，Verilog对表示真值/状态表的支持和它的进行引脚到引脚的延时选择的机制非常有用。"这个努力的结果是一套标准化的VHDL ASIC单元模型产生技术，它们由两个IEEE包支持：VITAL_timing和VITAL_primitive。

在厂商的系统中，这些模型的产生叫做库编译，我们用*Synopsys Library Compiler*中的例子来进行说明。图7-16、图7-17和图7-18给出了4输入ONAND门的库源代码以及编译得到的

VITAL模型。

```
Combinational Cell ONAND
/* Library Source Code */
cell(ONAND)
 {area : 9.0000 ;
  pin(Y) {direction : output ;
          capacitance : 0.5000 ;
          max_fanout : 15.0000 ;
          function : "!((A + B + C) D)" ;
          timing() {intrinsic_rise : 0.1234 ;
                    rise_resistance : 0.1500 ;
                    intrinsic_fall : 0.4500 ;
                    fall_resistance : 0.1600 ;
                    related_pin : "A" ;}
          timing() {intrinsic_rise : 0.1234 ;
                    rise_resistance : 0.1500 ;
                    intrinsic_fall : 0.4500 ;
                    fall_resistance : 0.1600 ;
                    related_pin : "B" ;}
          timing() {intrinsic_rise : 0.1234 ;
                    rise_resistance : 0.1500 ;
                    intrinsic_fall : 0.4500 ;
                    fall_resistance : 0.1600 ;
                    related_pin : "C" ;}
          timing() {intrinsic_rise : 0.1800 ;
                    rise_resistance : 0.0500 ;
                    intrinsic_fall : 0.5300 ;
                    fall_resistance : 0.0200 ;
                    related_pin : "D" ;}
         }
  pin(A) {direction : input ;
          capacitance : 1.0000 ;/
          fanout_load : 1.0000 ;}
  pin(B) {direction : input ;
          capacitance : 1.0000 ;
          fanout_load : 1.0000 ;}
  pin(C) {direction : input ;
          capacitance : 1.0000 ;
          fanout_load : 1.0000 ;}
  pin(D) {direction : input ;
          capacitance : 1.0000 ;
          fanout_load : 1.0000 ;}
 }
```

图7-16 组合单元ONAND的VITAL源代码。

注意在类属声明部分有两套延时：传输延时（如tpd_A_Y）和输入延时（如tipd_A）。这样，这个模型使用一个输入延时模型表示互连延时。传输延时表示固有的延时，其中包括坡度敏感延时。这些延时都具有缺省值，可以在反标时改变。转换延时的影响可以包括在反标的输入延时之中。

可声明内部信号以表示输入延时模型的连线，如A_ipd就是这样的内部信号。

结构体的代码段被分成三个主要部分：

1) 输入路径延时。函数VitalWireDelay 根据新的信号值、旧的信号值和线延时的值（如：

tipd_A）来计算正确的延时，并将新值在计算出的延时之后赋给线信号（如A_ipd）。该延时是传递的。

```
/* VITAL Model Output Code for Cell ONAND */
----- CELL ONAND -----
library IEEE;
use IEEE.STD_LOGIC_1164.all;
library IEEE;
use IEEE.VITAL_Timing.all;
-- entity declaration --
entity ONAND is
  generic(
    TimingChecksOn: Boolean := True;
    InstancePath: STRING := "*";
    Xon: Boolean := False;
    MsgOn: Boolean := True;
    tpd_A_Y    : VitalDelayType01:=(0.123 ns, .450 ns);
    tpd_B_Y    : VitalDelayType01 := (0.123 ns, 0.450 ns);
    tpd_C_Y    : VitalDelayType01 := (0.123 ns, 0.450 ns);
    tpd_D_Y    : VitalDelayType01 := (0.180 ns, 0.530 ns);
    tipd_A     : VitalDelayType01 := (0.000 ns, 0.000 ns);
    tipd_B     : VitalDelayType01 := (0.000 ns, 0.000 ns);
    tipd_C     : VitalDelayType01 := (0.000 ns, 0.000 ns);
    tipd_D     : VitalDelayType01 := (0.000 ns,0.000 ns));
  port(
    Y                              : out   STD_LOGIC;
    A                              : in    STD_LOGIC;
    B                              : in    STD_LOGIC;
    C                              : in    STD_LOGIC;
    D                              : in    STD_LOGIC);
  attribute VITAL_LEVEL0 of ONAND : entity is TRUE;
end ONAND;
```

图7-17 单元DNAND的VITAL模型实体的输出代码

2) 功能部分。ONAND门的功能是用STD_LOGIC的一般布尔操作符计算的。然而，对于更加复杂的模型，如触发器，可以使用来自VITAL_Primitive的原语（如状态表）。这些原语都针对高性能进行过优化。

3) 路径延时部分。根据输入信号改变和在干扰时进行调整所需的时间，选择输出延时。

为了完全理解这个功能是如何实现的，请读者参考VITAL_timing。

7.2 检错

我们希望对正确的和有故障的行为都进行建模。当对有故障的行为进行建模时，当发现错误时提示建模者非常有用。前面讨论了用于此目的的断言语句。这里讨论一下在门级模型用断言语句来建立检错机制。我们集中检查三种不同类型的错误条件：1) 给定输入的不正确的值；2) 错误的输入组合；3) 信号的错误时序。对三种不同类型出错条件的检查见图7-19所示。

1) **给定输入的错误值**。考虑输入信号值是多值逻辑类型的情况，如MVL4。然后对于某些输入，当‘0’和‘1’出现则正常，但对于‘X’和‘Z’则不正常并指示出错条件。在图7-19所示的RS触发器模型中，在R和S上出现的‘X’和‘Z’将被探测到并打印出错信息。

```
library IEEE;
use IEEE.VITAL_Primitives.all;
library LIBVUOF;
use LIBVUOF.VTABLES.all;
architecture VITAL of ONAND is
  attribute VITAL_LEVEL1 of VITAL : architecture is TRUE;
  SIGNAL A_ipd  : STD_ULOGIC := 'X';
  SIGNAL B_ipd  : STD_ULOGIC := 'X';
  SIGNAL C_ipd  : STD_ULOGIC := 'X';
  SIGNAL D_ipd  : STD_ULOGIC := 'X';
begin
--  INPUT PATH DELAYS
  WireDelay : block
  begin
    VitalWireDelay (A_ipd, A, tipd_A);
    VitalWireDelay (B_ipd, B, tipd_B);
    VitalWireDelay (C_ipd, C, tipd_C);
    VitalWireDelay (D_ipd, D, tipd_D);
  end block;
--  BEHAVIOR SECTION
  VITALBehavior : process (A_ipd, B_ipd, C_ipd, D_ipd)
   -- functionality results
    VARIABLE Results: STD_LOGIC_VECTOR(1 to 1):=(others => 'X');
    ALIAS Y_zd : STD_LOGIC is Results(1);
   -- output glitch detection variables
    VARIABLE Y_GlitchData : VitalGlitchDataType;
  begin
   --  Functionality Section
    Y_zd := (NOT ((D_ipd) AND ((B_ipd) OR (A_ipd) OR C_ipd))));
   --  Path Delay Section
    VitalPathDelay01 (
      OutSignal => Y,
      GlitchData => Y_GlitchData,
      OutSignalName => "Y",
      OutTemp => Y_zd,
      Paths => (0 => (A_ipd'last_event, tpd_A_Y, TRUE),
                1 => (B_ipd'last_event, tpd_B_Y, TRUE),
                2 => (C_ipd'last_event, tpd_C_Y, TRUE),
                3 => (D_ipd'last_event, tpd_D_Y, TRUE)),
      Mode => OnDetect,
      Xon => Xon,
      MsgOn => MsgOn,
      MsgSeverity => WARNING);
  end process;
end VITAL;
configuration CFG_ONAND_VITAL of ONAND is
  for VITAL
  end for;
end CFG_ONAND_VITAL;
```

图7-18 单元DNAND的VITAL模型输出代码中的构架

2) **错误的输入组合**。在某些情况下，有些输入组合将永远不会出现。如果出现，将会报告出错状况。对于RS触发器，这样一个条件是R=S=‘1’。图7-19给出的RS触发器模型是用两个交叉连接的NOR门实现的。当R=S=‘1’时，每个输出都被强制为0，这与实际电路中的情况相符。如果输入从R=S=‘1’切换到R=S=‘0’，触发器的最终值将取决于触发器内部的相对路径延时。在写该模型时，必须考虑如果输入同时从‘1’转到‘0’时终态如何。在模

型之中，我们选择对触发器复位。注意到布尔变量BOTH用来指出前一个输入情况是R=S='1'。在实际电路中一个输入可能会超前或滞后另一个。而且当R=S='0'且Q=QN='0'时，触发器处于不稳定状态。噪声和门延时的一点微小差别都可将触发器变成这样或那样一种状态。然而，在这种类型的模型之中，不可能考虑到低层次的效果。可以用其他方法来检测R=S='1'的出现，打印出错信息，不改变触发器的状态或将触发器的状态置为X，这将得到简单一点的模型，但不够精确。

```
use work.SYSTEM_4.all;
entity FFRS is
  generic(FFDEL,SPIKE_WIDTH: TIME);
  port(R,S: in MVL4; Q,QN: out MVL4);
end FFRS;
architecture BEHAV of FFRS is
begin
  process(R,S)
    variable R_LAST_EVT, S_LAST_EVT: TIME := 0 ns;
    variable BOTH: BOOLEAN:= FALSE;
  begin
  -----------Check for X's and Z's on inputs
   assert not(R='X') report "X  on R" severity WARNING;
   assert not(R='Z') report "Z  on R" severity WARNING;
   assert not(S='X') report "X  on S" severity WARNING;
   assert not(S='Z') report "Z  on S" severity WARNING;
   -------Spike Detection
   assert (NOW = 0 NS) or ((NOW - R_LAST_EVT) > SPIKE_WIDTH)
     report "Spike On R" severity WARNING;
   R_LAST_EVT := NOW;
   assert (NOW = 0 NS) or ((NOW - S_LAST_EVT) > SPIKE_WIDTH)
     report "Spike On S" severity WARNING;
   S_LAST_EVT := NOW;
if R = '0' and S ='1' then
      Q <= '1' after FFDEL;
      QN <= '0' after FFDEL;
     end if;
    if R = '1' and S='0' then
      Q <= '0' after FFDEL;
      QN <= '1' after FFDEL;
    end if;
    assert not(R = '1' and S = '1') --Check for R and S
      report "R and S Both 1"       --Both  '1'
      severity WARNING
    if R = '1' and S = '1' then
      Q <= '0' after FFDEL;
      QN <= '0' after FFDEL;
      BOTH := TRUE;
    end if;
    if R /= '1' and S /= '1' and BOTH then  --Previous
      Q <= '0' after FFDEL;      -- inputs were R=S='1'
     QN <= '1' after FFDEL;
     BOTH := FALSE;
    end if;
  end process;
end BEHAV;
```

图7-19 RS触发器模型

3) **不正确的信号时序**。在RS触发器模型的例子中，我们关心输入信号上尖锋的检测。这是由驱动电路中的险态引起的，在被驱动的电路中发现它们是一个有效的办法。检测通过保存R和S的“上一次事件的时间”来进行。只要建模触发器的过程被调用，函数NOW测量当前仿真时间，从这些时间之中减去得到的结果值与尖锋宽度定义进行比较来看是否出错。注意到触发器对于R和S上的尖锋无论多窄都会动作。另一个方法是当尖锋信号出现时保持触发器不变。

7.3 门级建模的多值逻辑

一个基本问题是，应该有多少个强度值？当在电路级用SPICE建模时，无限个强度值是可能的。对于数字建模，这显然没有必要。在第5章，我们讨论了具有两种强度的系统：强和弱。这些强度用于HVLT和IEEE9值系统。在要求对门级和低层次器件进行更加精确的建模时，将会发现需要更多的值。

7.3.1 MOS设计的附加值

MVL7系统的第7个值（Z）是通过将Z0、Z1和ZX叠加而得。其基本原理是阻抗极高因而基本的状态并不重要。对于门级建模，该假设正确。然而，现在越来越多的建模在开关级进行，该级是电路和门级的混合级。这一级的建模由于MOS技术用于VLSI电路设计而变得必要。在开关级建模时，通过增强对信号源阻抗的控制，来加强建模原语的基本逻辑。参考图7-20中的情况，其中给出了由传输门驱动的NMOS反相器。当CON=‘1’时，门是开的，点I驱动点Q，然而，如果CON变为‘0’，点Q保持相对较长的一段时间为1，大多数情况下为毫秒级。这是因为信号Q的源阻抗极高。如果使用MVL7，则认为Q的值为‘Z’，但是它的状态怎样？存储电荷是MOS电路的主要存储机制，如果想进行开关级建模，则必须对之进行建模。

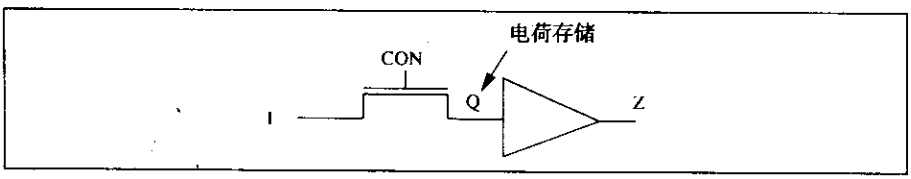


图7-20　MOS电路中的电荷存储

Z0	Z1	ZX
R0	R1	RX
F0	F1	FX

图7-21　九值系统

解决方式是用三个高阻态：Z0、Z1和ZX。结果得到一个根据三个强度（strong、weak和high impendeace）得出的九值系统和三个状态（0，1，x）。图7-21提供了该系统的一种可能的值标记方法，图7-22给出了九值系统的值格子（value lattice）。信号判决（R）对应于格子当中的最小上限（lub），例如(R0) R (R1)=RX，为了与专业术语保持一致，我们用项

forcing(F)代表强，resistive(R)代表weak。

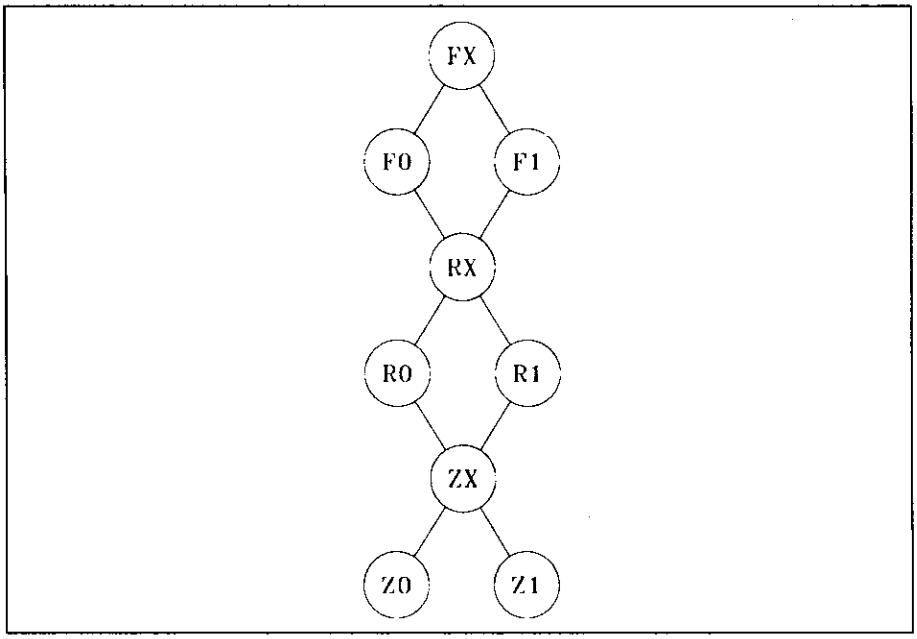


图7-22 九值系统的值网格

7.3.2 通用的状态/强度模型

当想对更多的开关级效果进行建模时，需要更多的强度。我们讨论状态/强度模型的一个通用形式。包含状态X、0和1，以及任意数量的强度，用自然数N表示。0是最大强度，然后强度值N单调递减。系统中的每个值是2元组，形式为（S_i, N），其中S_i为X、0或1，N是强度值。系统的值网格在图7-23中给出。注意格子之中最大的值是（X,0），它是（1,0）和（0,0）的最小上限，即将最强0和最强1混合而得到。格子中最小的元素是（X,inf），其强度最弱，即N=inf。

这个结构是由Smith和Acosta在MCC建模MOS开关电路时应用的。图7-24给出了基本的情况。信号通过一系列MOS开关，而无再生放大。每经过一个开关则信号减弱一点，从而需要大量的信号强度。

在MCC系统中，信号强度数量的最大可选值为255。选择的值由通过电路的最长路径决定。MCC的开关模型是双向的，其信号流建模为从源到漏或从漏到源。该电路模型是在节点处互连的开关。MCC实现了开关模型和后面由于性能原因作为本地系统原语而进行讨论的节点模型，他们同样也开发了等价的VHDL代码，见图7-25、图7-26和图7-27，图7-25给出了包BIT_TYPES，由开关和节点模型使用。每个开关模型接收源和漏的信号，由二元组组成，其值包括一个类型为BIT_CODE的状态和一个类型为WEAKNESS_VALUE的强度。节点模型接收输入，类型为BIT_CODE_VECTOR和WEAKNESS_VALVE_VECTOR。枚举类型SWITCH_STATE用来代表开关的状态。函数LATTICE_LUB和NEW_NODE_WINS同样为节点模型所用。图中只给出了其接口，如果理解了值格子，则其操作是等价的。

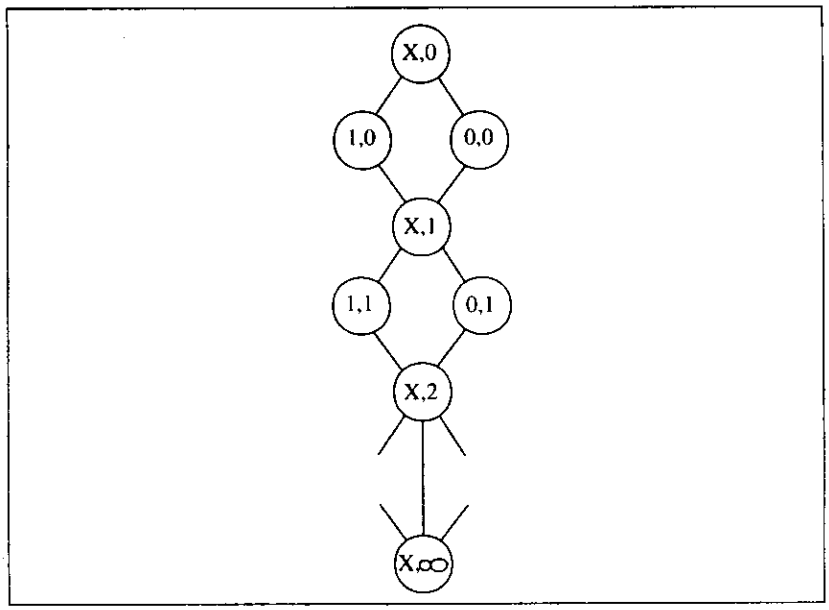


图7-23 强度/状态系统的通用值格子

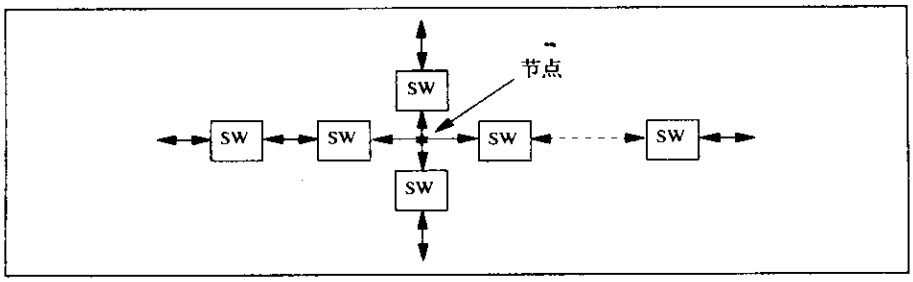


图7-24 MOS开关电路

```
package BIT_TYPES is
  type BIT_CODE is ('0', '1', 'X', 'Z');
  type BIT_CODE_VECTOR is array (Natural range <>)
       of BIT_CODE;
  type WEAKNESS_VALUE is range 0 to 255;
  type WEAKNESS_VALUE_VECTOR is array (Natural range <>)
       of WEAKNESS_VALUE;
  type SWITCH_STATE is (mos_turn_off, mos_turn_on,
                        mos_turn_x);
  procedure LATTICE_LUB(NEW_VALUE: inout BIT_CODE;
                        NEW_WEAKNESS: inout WEAKNESS_VALUE;
                        VALUE: in BIT_CODE;
                        WEAKNESS: in WEAKNESS_VALUE);
  function NEW_NODE_WINS(NEW_VALUE: BIT_CODE;
                         NEW_WEAKNESS: WEAKNESS_VALUE;
                         OLD_VALUE: BIT_CODE;
                         OLD_WEAKNESS: WEAKNESS_VALUE)
    return Boolean;
end BIT_TYPES;
```

图7-25 开关电路的包集合

图7-26给出了模型NSWITCH。该模型响应GATE输入上的变化，可适当设置其状态。Delta时间之后，当信号STATE改变时，可采取适当的动作。

```
use work.BIT_TYPES.all;
entity NSWITCH is
  generic (CHANNEL_DEGRADATION: WEAKNESS_VALUE := 1);
  port (GATE: in BIT_CODE;
     SOURCE, DRAIN: inout BIT_CODE;
     SOURCE_WEAKNESS, DRAIN_WEAKNESS: inout WEAKNESS_VALUE);
end NSWITCH;

architecture ARCH of NSWITCH is
  signal STATE: SWITCH_STATE;
begin
  process (GATE, SOURCE, DRAIN, STATE)
  begin
    if (GATE'event) then
      case (GATE) is
        when '0'    => STATE <= mos_turn_off;
        when '1'    => STATE <= mos_turn_on;
        when others => STATE <= mos_turn_x;
      end case;
    end if;
    case (STATE) is
      when mos_turn_off =>  -- infinite weakness
        DRAIN           <= 'X';
        DRAIN_WEAKNESS  <= WEAKNESS_VALUE'high;
        SOURCE          <= 'X';
        SOURCE_WEAKNESS <= WEAKNESS_VALUE'high;
      when mos_turn_on  =>
        DRAIN      <= SOURCE;
    DRAIN_WEAKNESS <= SOURCE_WEAKNESS + CHANNEL_DEGRADATION;
        SOURCE     <= DRAIN;
    SOURCE_WEAKNESS <= DRAIN_WEAKNESS + CHANNEL_DEGRADATION;
      when mos_turn_x   =>
        DRAIN      <= 'X';
    DRAIN_WEAKNESS <= SOURCE_WEAKNESS + CHANNEL_DEGRADATION;
        SOURCE     <= 'X';
    SOURCE_WEAKNESS <= DRAIN_WEAKNESS + CHANNEL_DEGRADATION;
    end case;
  end process;
end ARCH;
```

图7-26 开关模型

1) 状态：MOS_TURN_OFF：源和漏接收值‘X’，其弱度（weakness）被设为最大(255)。

2) 状态：MOS_TURN_ON：源和漏互相接受其值，而且其弱度（weakness）随开关的通道衰减而增加（缺省值为1）。

3) 状态：MOS_TURN_X：源和漏接值‘X’，其弱度（weakness）随开关的通道衰减增加。

图7-27给出节点模型。节点是当开关的输出（源或漏）连在一起形成的。实体IDEAL_NODE接收由连到节点作为输入的开关的输出值和强度组成的向量，然后：

1) 确定所有输入的最小上限。

```
use work.BIT_TYPES.all;
entity IDEAL_NODE is
  port (STATES: inout BIT_CODE_VECTOR;
        STRENGTHS: inout WEAKNESS_VALUE_VECTOR);
end IDEAL_NODE;

architecture ARCH of IDEAL_NODE is
begin
  process (STATES, STRENGTHS)
    variable NODE_VALUE, RESULT_VALUE: BIT_CODE;
    variable NODE_WEAKNESS, RESULT_WEAKNESS: WEAKNESS_VALUE;
  begin
    RESULT_VALUE := 'X'; -- infinite weakness
    RESULT_WEAKNESS := WEAKNESS_VALUE'high;
    for i in STATES'range loop
      LATTICE_LUB(RESULT_VALUE, RESULT_WEAKNESS, STATES(i),
                  STRENGTHS(i));
    end loop;
    if (NEW_NODE_WINS(RESULT_VALUE, RESULT_WEAKNESS,
        NODE_VALUE, NODE_WEAKNESS)) then
      for i in STATES'range loop
        if (NEW_NODE_WINS(RESULT_VALUE, RESULT_WEAKNESS,
            STATES(i), STRENGTHS(i))) then
          STATES(i)    <= RESULT_VALUE;
          STRENGTHS(i) <= RESULT_WEAKNESS;
        end if;
      end loop;
    end if;
    NODE_VALUE := RESULT_VALUE;
    NODE_WEAKNESS := RESULT_WEAKNESS;
  end process;
end ARCH;
```

图7-27　节点模型

2) 如果节点的新值比节点的旧值强度大，所有比新的节点值弱的输入值都置为与新值相等。

使用这种方法，MCC对大约有1000个开关的电路进行了建模。如果在电路级使用SPICE对这样的电路进行建模，则不太切合实际。

7.3.3 区间逻辑

另一种开发高值逻辑系统（higher-valued logic system）的有效方法是基于区间逻辑的概念。这种方法中，由一个基本逻辑集合开始，根据状态和强度，以某种特定方式排序。其他逻辑值由基本值集合的区间确定。作为例子，考虑一个用于初始HILO系统的值系统。基本值集合包括值{ '1', 'H', 'Z', 'L', '0' }，其值与MVL7中定义的意义相同。基本值集合被'Z'分成high_value子集和low_value子集，这样，从最左边的值'1'开始，从值'H'到值'Z'递减强度，没有状态信息。当在low子集中向右边移动时，强度从值'L'增到'0'。对这个基本值集合，可以定义如下数量的区间：4个长度为2的区间，3个长度为3的区间，两个长度为4的区间，和一个长度为5的区间。图7-28给出了这些区间的定义和其对应值符号。

该系统的类型定义应该是：

```
type HILO_BIT is(  'X',
```

```
    'U','D'
  'T','W','B'
 'P','R','F','N'
'1','H','Z','L','0');
```

区间长度	区间	符号
2	1H	P
2	HZ	R
2	ZL	F
2	L0	N
3	1HZ	T
3	HZL	W
3	ZL0	B
4	1HZL	U
4	HZL0	D
5	1HZL0	X

图7-28 HILO逻辑区间

类型声明的三角形形状暗示了值形成的方式。

总的来说，我们从一个包含强度的状态信息的基本集合开始，形成代表值的范围的区间。当范围的长度递增时，则信号保持该值的不确定性增加。最大的范围为值X，表示了最大的不确定性，即信号可以和任何基本值相等。建立这样一个值的目的在于，与一个值比较少的系统相比，允许判决函数产生更加确定的判决值，图7-29给出了HILO判决函数。

区间逻辑的概念可以通用化，考虑有基本值集合{V_1,V_2,…V_n}的系统，它可以产生长度为i的 n-i+1个区间。而且，系统逻辑值的总数为：

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

对于系统HILO，该等式等于15。

7.3.4 Vantage系统

现在考虑由Vantage Analysis开发的区间逻辑系统，使用下述基本值集合：

{F0, R0, W0, Z0, D, Z1, W1, R1, F1}

基本值集合有四种强度：F（Forcing）、R（resistive）、W（weak）和Z（high impedance）。值D代表完全没有连接的节点，其中没有存储电荷。根据这个基本值集合，可以产生45值的系统。在该Vantage系统加入值U，代表有未初始化的信号。这样，我们得到46个值。图7-30给出了这个系统的值的定义。选择值符号以表示区间的端点。如果跨越了0/1边界，则在尾部附加一个X，这样，值FZX代表区间形式F0（forcing 0）到Z1（high impedance 1）。

这种类型的系统其基本优点是当判决信号时，判决的结果可以比值较少的系统更加确定一点。考虑图7-31a中所示的电路，两个总线驱动器的输出连接在一起。图7-31a中给出了使用MVL7的情况。上面的驱动使其使能端连接于X。对于一般的CMOS和TTL技术（强度参数

=X01，正如下面所解释的），输出将是X，下面的驱动器的输出为0，X与0进行判决，结果为X。然而，既然两个缓冲器的输入都为0，这是个保守的结果：无论上面的缓冲器是否使能，判决值应为‘0’。图7-31b给出了使用46值系统的同样情况。这里，上面缓冲器的输出将为FZX，意味着输出或者为不同强度的0，或者为高阻态1，或者为不相连。高阻态的1说明在输出上先前保存的值可能为Z1。在这种情况下，判决函数将会逻辑地判决FZX和F0，得到F0，这是正确的且比较确定的结果。Vantage系统的判决函数是一张46×46表，在David Coelho的《VHDL手册》一书中给出。

	X	1	0	U	D	P	N	T	B	W	H	L	R	F	Z
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
1	X	1	X	1	X	1	X	1	X	1	1	1	1	1	1
0	X	X	0	X	0	X	0	X	0	0	0	0	0	0	0
U	X	1	X	U	X	U	X	U	X	U	U	U	U	U	U
D	X	X	0	X	D	X	D	X	D	D	D	D	D	D	D
P	X	1	X	U	X	P	X	P	X	U	P	U	P	U	P
N	X	X	0	X	D	X	N	X	N	D	D	N	D	N	N
T	X	1	X	U	X	P	X	T	X	U	P	U	T	U	T
B	X	X	0	X	D	X	N	X	B	D	D	N	D	B	B
W	X	1	0	U	D	U	D	U	D	W	W	W	W	W	W
H	X	1	0	U	D	P	D	P	D	W	H	W	B	W	B
L	X	1	0	U	D	U	N	U	N	W	W	L	W	L	L
R	X	1	0	U	D	P	D	T	D	W	H	W	R	W	R
F	X	1	0	U	D	U	N	U	B	W	W	L	W	F	F
Z	X	1	0	U	D	P	N	T	B	W	H	L	R	F	Z

图7-29　HILO判决函数

7.3.5　多值门级模型

在许多情况下，设计者想仿真他们设计的门级电路的结构模型。这一节介绍使用IEEE逻辑系统的门级模型。我们使用OR门作为例子。图7-32给出了OR函数的真值表。它来自STD_LOGIC_1164包集合，注意对这个函数，A OR 1=1 OR A=A OR H=H OR A=1，所有A的类型都是STD_LOGIC_1164，即1和H是等价的和支配的。

图7-33给出了OR门实体的VHDL描述，来自Synopsys包集合std_logic_entities。注意有四个类属：1) N，它是门输入的个数，即是一个通用OR模型；2) TLH；3) THL；4) 强度，它确定门输出值。

强度参数定义如下：

```
type STRENGTH is (strn_X01, strn_X0H, strn_XL1, strn_X0Z,
                  strn_XZ1, strn_WLH, strn_WLZ, strn_WZH,
                  strn_W0H, strn_WL1);
```

强度参数不可与数值系统的基本强度即weak和strong相混淆。而且这是一个类属强度参数，

用来确定采用不同技术的门输出的强度，每个强度参数是一个形式为ABC的三字符的标号，其中：

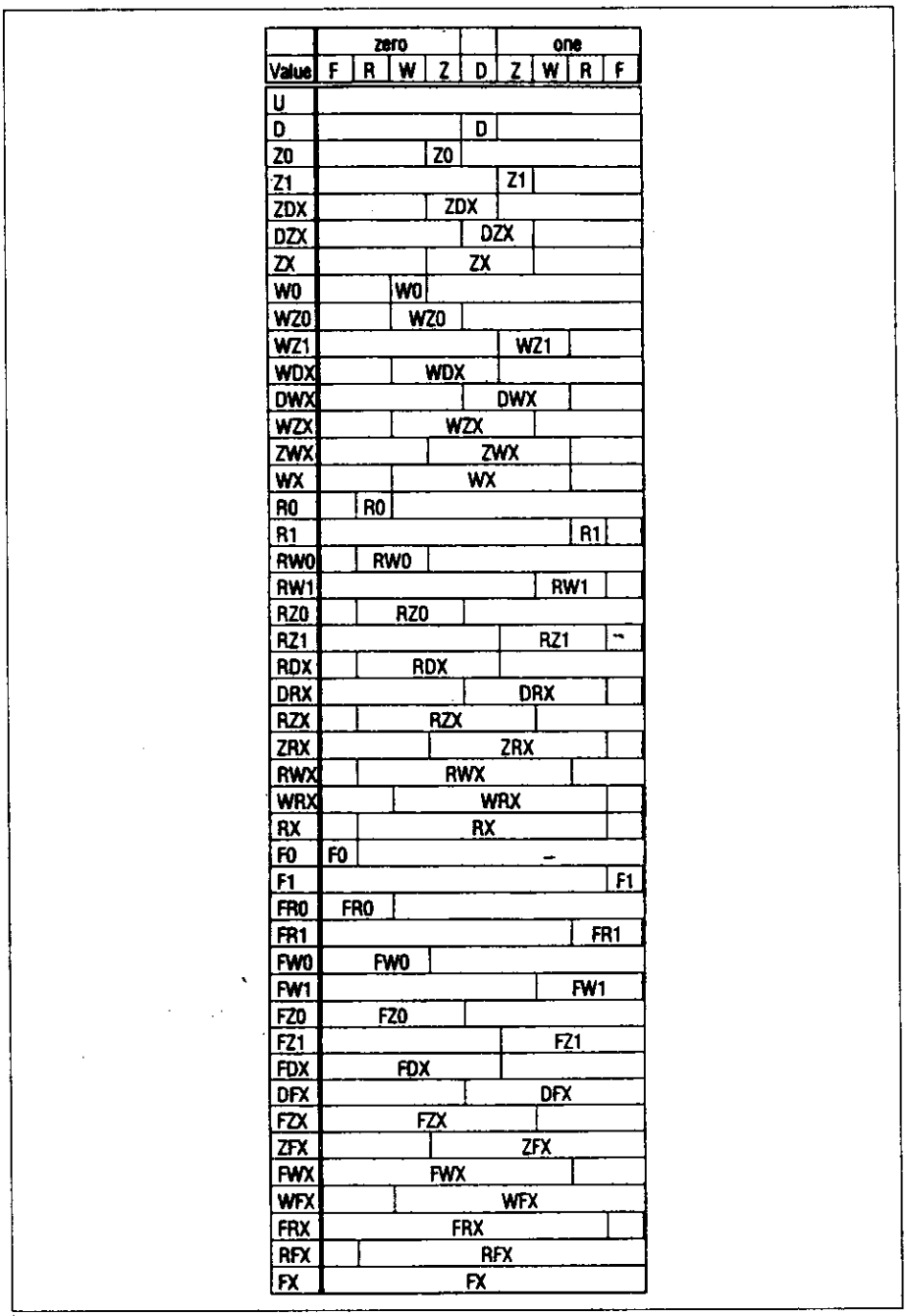


图7-30 Vantage 46值系统

A：代表输出值（X或W）为未知或Z。

B：代表输出值（0或L或Z）为低电平输出。

C：代表输出值（1或H或Z）为高电平输出。

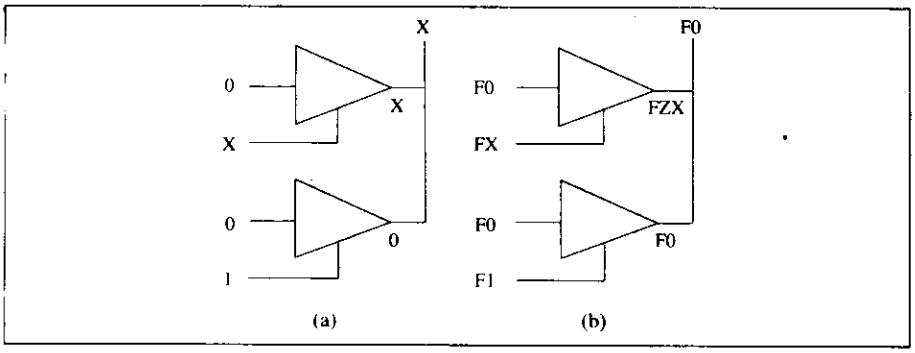


图7-31 Vantage逻辑系统的优点

```
-- truth table for "or" function
constant OR_TABLE : STDLOGIC_TABLE := (
----------------------------------------------------
--|  U    X    0    1    Z    W    L    H    -     |  |
----------------------------------------------------
  ( 'U', 'U', 'U', '1', 'U', 'U', 'U', '1', 'U' ),  -- | U |
  ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ),  -- | X |
  ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | 0 |
  ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ),  -- | 1 |
  ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ),  -- | Z |
  ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ),  -- | W |
  ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | L |
  ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ),  -- | H |
  ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ));  -- | - |
```

图7-32 OR函数的真值表

这样，强度参数WLZ表示：1) 未知的输出值将表示为W；2) 低电平的输出值则表示为L；3) 高电平的输出值则表示为Z（高阻状态）。表7-1给出一些Synopsys定义的强度参数和技术之间的对应关系。强度参数和图7-34的表同时使用，对于给定的内部STD_LOGIC类型，产生正确的输出强度。由内部模型产生的输出值和输出强度参数结合，以产生最后的输出。例如，对于强度参数WLZ。1) 内部模型值X、Z和W映射至W；2) 内部模型值D或L映射至L；3) 内部模型值1和H映射至Z。对于OR门模型，缺省强度为X01，它是标准的CMOS或TTL。

表7-1 强度参数技术

参 数	典型技术
X01	一般的CMOS或TTL（缺省）
X0H	NMOS
XL1	PMOS
X0Z	集电极开路/漏极开路
XZ1	CMOS源上拉电阻开路
WLH	总线保持器，阻塞锁存器
WLZ	总线保持器，弱上拉
WZH	二选一总线保持器
WHO	ECL线与
WL1	ECL线或

```
------------------------------------------------------------
--Primitive name: ORGATE
--Purpose: An OR gate for multiple value logic STD_LOGIC,
-- N inputs, 1 output.
--(see package IEEE.STD_LOGIC_1164 for truth table)
------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_MISC.all;
entity ORGATE is
  generic (N:   Positive := 2;-- number of inputs
           tLH: Time := 0 ns; -- rise inertial delay
           tHL: Time := 0 ns; -- fall inertial delay
           STRN: STRENGTH := STRN_X01);-- output strength
  port (INPUT: in STD_LOGIC_VECTOR (1 to N);-- inputs
        OUTPUT: out STD_LOGIC);  -- output
end ORGATE;
architecture A of ORGATE is
  signal CURRENTSTATE: STD_LOGIC := 'U';
  subtype TWOBIT is STD_LOGIC_VECTOR (0 to 1);
begin
  P: process
    variable NEXTSTATE: STD_LOGIC;
    variable DELTA: Time;
    variable NEXT_ASSIGN_VAL: STD_LOGIC;
  begin
  -- evaluate logical function
    NEXTSTATE := '0';
    for i in INPUT'range loop
      NEXTSTATE := INPUT(i) or NEXTSTATE;
      exit when NEXTSTATE = '1';
    end loop;
    NEXTSTATE := STRENGTH_MAP(NEXTSTATE, STRN);
    if (NEXTSTATE /= NEXT_ASSIGN_VAL) then
    -- compute delay
      case TWOBIT'(CURRENTSTATE & NEXTSTATE) is
        when "UU"|"UX"|"UZ"|"UW"|"U-"|"XU"|"XX"|"XZ"|"XW"|
             "X-"|"ZU"|"ZX"|"ZZ"|"ZW"|"Z-"|"WU"|"WX"|"WZ"|
             "WW"|"W-"|"-U"|"-X"|"-Z"|"-W"|"--"|"00"|"0L"|
             "LL"|"L0"|"11"|"1H"|"HH"|"H1"
          => DELTA := 0 ns;
        when "U1"|"UH"|"X1"|"XH"|"Z1"|"ZH"|"W1"|"WH"|"-1"|
             "-H"|"0U"|"0X"|"01"|"0Z"|"0W"|"0H"|"0-"|"LU"|
             "LX"|"L1"|"LZ"|"LW"|"LH"|"L-"
          => DELTA := tLH;
        when others => DELTA := tHL;
      end case;
    -- assign new value after internal delay
      CURRENTSTATE <= NEXTSTATE after DELTA;
      OUTPUT <= NEXTSTATE after DELTA;
      NEXT_ASSIGN_VAL := NEXTSTATE;
    end if;
  -- wait for signal changes
    wait on INPUT;
  end process P;
end A;
```

图7-33 OR门模型

构架开始的时候计算逻辑函数，这是通过一个for循环每次对两个输入进行“或”运算来实现的，它使用了图7-32中定义的表。这个for循环的方法使模型对于任意的N都可以工作。然而，该运算必须是可结合的。下一步，使用SRENGTH_MAP来确定输出值。这样，通过检查来确定OR函数的值从上次改变之后是否又发生了改变。即NEXTSTATE与NEXT_ASSIGN_

VAL进行比较。如果有不同，将会安排新的值。case语句用来确定delta的值，即被使用的延时值。通过比较NEXTSTATE和CURRENTSTATE来实现。强度改变仅在0ns内进行，即delta延时，第一条case语句说明了这种情况。下面两个语句说明了上升和下降的情况，其强度与状态都发生了改变。下一步CURRENTSTATE的值和输出的值被最终安排在delta时间单元之后改变。NEXT_ASSIGN_VAL与NEXTSTATE置为相等以便在下一次门级模型被调用时使用。然后进程开始等待输出变化。

```
-- truth table for output strength --> STD_ULOGIC lookup
constant TBL_STRN_STD_ULOGIC: STRN_STD_ULOGIC_TABLE :=
----------------------------------------------------------
--| X01 X0H XL1 X0Z XZ1 WLH WLZ WZH W0H WL1 | strn/output|
----------------------------------------------------------
  (('U','U','U','U','U','U','U','U','U','U'), -- | U |
   ('X','X','X','X','X','W','W','W','W','W'), -- | X |
   ('0','0','L','0','Z','L','L','Z','0','L'), -- | 0 |
   ('1','H','1','Z','1','H','Z','H','H','1'), -- | 1 |
   ('X','X','X','X','X','W','W','W','W','W'), -- | Z |
   ('X','X','X','X','X','W','W','W','W','W'), -- | W |
   ('0','0','L','0','Z','L','L','Z','0','L'), -- | L |
   ('1','H','1','Z','1','H','Z','H','H','1'), -- | H |
   ('X','X','X','X','X','W','W','W','W','W'));-- | - |
function STRENGTH_MAP(INPUT: STD_ULOGIC; STRN: STRENGTH)
  return STD_LOGIC is
begin
  return TBL_STRN_STD_ULOGIC(INPUT, STRN);
end STRENGTH_MAP;
```

图7-34　输出强度表和强度函数

为了理解这个模型，注意到NEXTSTATE、NEXT_ASSIGN_VAL和CURRENTSTATE的区别很重要。变量NEXTSTATE是输入在当前仿真时间的“或”运算的结果。变量NEXT_ASSIGN_VAL是在上一次输入变化时的输入“或”运算的结果。信号CURRENTSTATE是在上一次调度的输出改变发生时OR函数的值。通常NEXT_ASSIGN_VAL和CURRENTSTATE根据这种延时状态更新机制的不同而具有不同的值。

7.3.6　精确延时建模

早些时候为了对不同的延时建模，讨论了TPLH和TPHL的定义，这取决于输入是低到高还是高到低。在OR门模型中，如果信号改变只涉及强度，则建模为零延时。其他改变使用了TPLH和TPHL。在某些情况下，当从高阻态强度开始转换或转换到高阻态强度时，提供一套独立的延时是很重要的，因为对于MOS电路，转换到高阻态或从高阻态与正常的传输延时相比，将耗费相当长的时间。

7.4　门级模型的配置声明

在第3章讨论了配置说明的使用和配置声明，以及把它们作为将结构模型中的组件绑定到行为级模型上的方法。正如那时指出的，这些构架的目的在于将组件声明与库中存在的模型绑定在一起。在这一节，将讨论配置声明对于门级建模的其他用途。

先复习一下，在配置声明中，我们指定在一个独立的可分析的实体中，所有的组件绑定

于一个结构化构架。这样做的主要原因是：

1) 绑定可以推迟或做为一个设计过程进行修改。

2) 如果绑定发生改变，要重新分析配置声明，而结构化构架则不用重新进行分析。这种对结构化构架的配置声明比构架本身的代码要少得多。对于一个实例化上千个门的构架实体，更是这种情况。这样，当只需重新分析配置声明而不是整个结构化构架时，分析时间将大大减少。

3) 配置声明是进行名字映射的自然方法，使得在同一结构化构架中可以使用各种组件库模型。

4) 配置声明同样也是了解结构化模型中组件细节的自然方法，可以通过修改这种信息来学习设计的本质。

例如，配置声明是处理反向标注问题的一个好方法。对一个门阵列的实现收集到的延时值可以插入被建模电路的配置声明中。

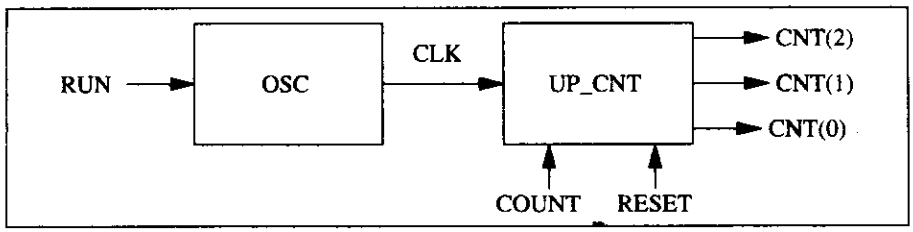


图7-35 计数器系统

我们通过图7-35中的计数器系统来说明配置声明的使用。晶振组件当RUN='1'时输出周期性的时钟信号（CLK）。如果组件UP_CNT的输入COUNT和RESET分别是'1'和'0'，UP_CNT模块进行模8计数，即是一个除8计数器。如果RESET是'1'，时钟活动被忽略，重置计数器。在图7-36中，UP_CNT模块可以更进一步结构化地分解为三个T触发器和一个AND门。

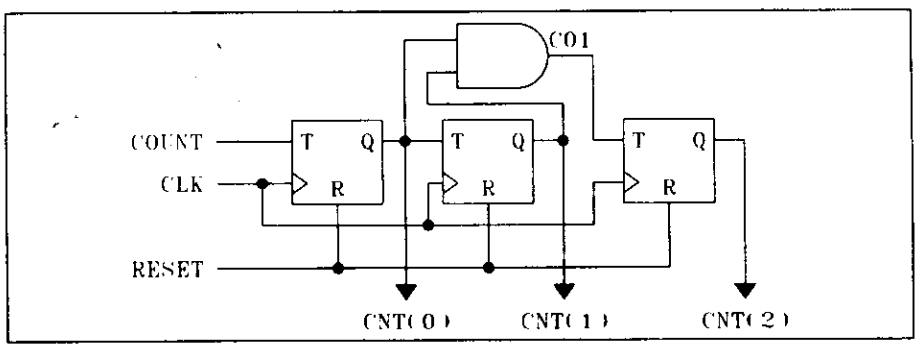


图7-36 正向计数器的结构模型

计数器系统的设计可以用两种不同的设计树表示。图7-37a给出了当UP_CNT被进行行为级建模时的设计树，图7-37b给出了当UP_CNT被进行结构级建模时的设计树。注意在每个框图中，表示叶组件的方框内部都有个方框。内部的方框表示组件可以插入的连接器。在组件插入连接器插槽之前，该槽为非绑定的。

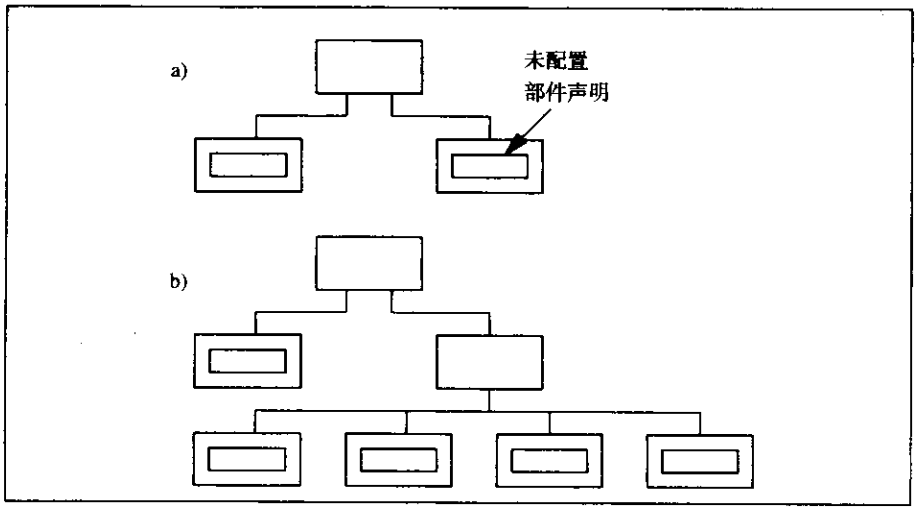


图7-37 计数器系统设计树

图7-38给出了计数器系统的VHDL结构化构架，即晶振和正向（UP）计数器。对应于图7-35中的原理图。注意即使组件OSC和UP_CNT已经声明了，它们仍是非绑定的。正如已经说明的，这些组件声明代表了将会使用到的组件连接器。组件实例如C0：OSC port map（RUN=>RUN，CLOCK=>CLK），指定组件的个数和互连连线。图7-39给出了正向计数器的一个未绑定的结构化构架，由T触发器和与门实现，对应于图7-36中所示的电路。

```
entity CTR is
  port(RESET,RUN,COUNT: in BIT;
       CNT: inout BIT_VECTOR(2 downto 0));
end CTR;

architecture STRUCTB of CTR is
  signal CLK: BIT;
  component OSC
    port(RUN: in BIT; CLOCK: out BIT := '0');
  end component;
  component UP_CNT
    port(RESET,COUNT,CLK: in BIT;
         CNT: inout BIT_VECTOR(2 downto 0));
  end component;
begin
  C0: OSC
    port map(RUN => RUN,CLOCK => CLK);
  C1: UP_CNT
    port map(RESET => RESET, COUNT=> COUNT,
             CLK => CLK, CNT => CNT);
end STRUCTB;
```

图7-38 计数器系统的未绑定结构化构架

图7-40给出了实体CTR的配置ALL_BEHAV，配置体是嵌套的。在for STRUCTB结构中是配置语句，它对应于实体CTR的构架STRUCTB，即组件规范语句针对构架STRUCTB的组件C0：OSC和C1：UP_CNT。包括在这些组件规范之中的是类属映像，用来指定两个组件的实际时序。注意ALL_BEHAV配置了图7-37中所示的设计树。

```
entity UP_CNT is
  generic(R_DEL,CLK_DEL:TIME);
  port(RESET,COUNT,CLK: in BIT;
       CNT: inout BIT_VECTOR(2 downto 0));
end UP_CNT;
architecture STRUCT of UP_CNT is
  signal C01: BIT;
  component T
    port(R,T,CLK: in BIT; Q: out BIT);
  end component;
  component AND2G
    port(I1,I2: in BIT; O: out BIT);
  end component;
begin
  C1: T
    port map(R => RESET,T => COUNT,CLK => CLK,Q => CNT(0));
  C2: T
    port map(R => RESET,T => CNT(0),CLK => CLK,Q => CNT(1));
  C3: T
    port map(R => RESET,T => C01, CLK => CLK, Q => CNT(2));
  C4: AND2G
    port map(I1 => CNT(0), I2 => CNT(1), O => C01);
end STRUCT;
```

图7-39 正向计数器系统的未绑定结构化构架

```
configuration ALL_BEHAV of CTR is
  for STRUCTB
    for C0: OSC use entity work.COSC(WAIT_DEL)
      generic map(60 ns, 40 ns);
    end for;
    for C1: UP_CNT use entity work.UP_CNT(ALG)
      generic map(10 ns, 15 ns);
    end for;
  end for;
end ALL_BEHAV;
```

图7-40 计数器系统的行为级配置

图7-41给出了实体CTR的配置PART_STRUCT，在这种情况下，一个结构化构架配置给UP_CNT，并在嵌套的内层进行配置。

7.4.1 缺省配置

在缺省配置中，系统查找名字与声明的组件名相匹配的实体。这时不需要绑定语句。

图7-42给出了缓冲器的算法级和结构级实现。图7-43给出测试该缓冲器进程的测试程序包，该测试程序包的缺省绑定包含在配置DEFAULT1中。在测试程序包中，它将实体BUFF最近分析过的版本绑定给组件BUFF。如果结构化实体是最近分析过的，也会在结构级构架中将最近分析过的实体INV绑定到组件INV。换言之，缺省绑定顺着层次向下扩展。这样，同样的测试程序包可以用来测试行为级和结构级模型。

尽管缺省配置提供了一种简单，有力的方法来指定库的绑定，但是如果忘记了针对实体的哪个构架是最近分析过的，就产生了小问题。如果好几个人共享一个组件库，这就是一个特别问题了。其他人可能分析了不同的构架而未通知你，这种事情有可能导致使用不正确缺

省配置的测试程序包，从而产生无法重复的结果。

```
configuration PART_STRUCT of CTR is
  for STRUCTB
    for C0: OSC use entity work.COSC(WAIT_DEL)
      generic map(60 ns, 40 ns);
    end for;
    for C1: UP_CNT use entity work.UP_CNT(STRUCT)
      generic map(10 ns, 15 ns);
        for STRUCT
          for all: T use entity work.TFF(ALG)
            generic map(10 ns, 15 ns);
          end for;
          for C4: AND2G use entity work.AND2(BEH)
            generic map(5 ns);
          end for;
        end for;
    end for;
  end for;
end PART_STRUCT;
```

图7-41　计数器系统的部分结构级配置

```
entity BUFF is
  port( I: in BIT; O: out BIT);
end BUFF;
architecture ALG of BUFF is
begin
  O <= I;
end ALG;
```

a) 行为级缓冲器

```
entity INV is
  port(I: in BIT; O: out BIT);
end INV;
architecture ALG of INV is
begin
  O <= not I;
end ALG;

entity BUFF is
  port( I: in BIT; O: out BIT);
end BUFF;

architecture STRUCTURE of BUFF is
  component INV
    port(I: in BIT; O: out BIT);
  end component;
  signal S: BIT;
begin
  C1: INV
    port map(I,S);
  C2 : INV
    port map(S,O);
end STRUCTURE;
```

b) 结构级缓冲器

图7-42　行为级和结构级缓冲器的缺省配置

```
entity TB is
end TB;

architecture BUF_TEST of TB is
  signal I,O: BIT;
  component BUFF
    port( I: in BIT; O: out BIT);
  end component;
begin
  C1: BUFF
    port map(I,O);
  I <= '0', '1' after 10 ns, '0' after 20 ns;
end BUF_TEST;
configuration DEFAULT_1 of TB is
  for BUF_TEST
    for C1:BUFF
    end for;
  end for;
end DEFAULT_1;
```

图7-43 测试程序包的缺省配置

7.4.2 配置和组件库

在使用组件库时，配置很重要。不同的仿真程序厂商有标准组件模块库。设计组在设计进展过程中同样也开发自己的库。这些库包括类属门和寄存器元件。这些门和触发器的接口描述一般是通用的；这样，配置体必须将结构级构架的描述映射到库接口描述之上。这就是前面提到的名字映射函数。

作为例子，考虑图1-1中所示的TWO_CONSECUTIVE电路的结构化模型。图7-44给出了电路的结构化模型，在这个例子中，使用了来自两个库的组件，Synopsys IEEE库包括由该公司开发的模块及WORK库。在TWO_CONSECUTIVE的构架模型中，声明并实例化了四个组件：EDGE_TRIGGERED_D、INVG、AND3G和OR2G，但未对其绑定。图7-45给出了采用的库模块的接口描述。有DFFREG、ANDGATE和INVGATE的描述。ORGATE的描述与ANDGATE的描述类似，这里没有给出。除了DFFRGE模块，其他接口描述取自SYNOPSYS IEEE库的包集合STD_LOGIC_COMPONENTS。该模块被修改以产生反相和非反相的状态输出。复位输出是低有效，每个模块具有类属THL和TLH以及强度。DFFREG、ANDGATE和ORGATE有类属N，对于DFFREG，指的是数据宽度；对于门模块，说明输入的个数。注意，所有的类属有其缺省值，缺省值在未指定其他值时使用。

图7-46给出了实体TWO_CONSECUTIVE的构架STRUCTURAL的两个配置。配置PARTS1将构架组件与前面讨论的四个组件绑定在一起。组件DFFREG来自库WORK，其他三个组件来自库synopsys IEEE库的包集合STD_LOGIC_ENTITIES。配置PARTS1使用时序的缺省值，其TLH与THL是0。这样，这个配置可用于仿真delta延时，配置PARTS2指定一套TLH，THL的值，这样，可用来对特定延时情况进行建模，配置PARTS1和PARTS1都使用缺省强度值，即其输出强度为X01，适合于标准TTL电路和CMOS电路。

7.5 对竞争和险态建模

本节讲解配置声明在对竞争和险态建模时的用途。作为用配置声明来建模竞争的例子，

我们再考虑图7-36的正向计数器。设计者对于该设计可能会问的一些基本问题是：

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_entities.all;
use work.all;
entity TWO_CONSECUTIVE is
  port(CLK, R: in STD_LOGIC; X: in STD_LOGIC;
       Z: out STD_LOGIC);
end TWO_CONSECUTIVE;

architecture STRUCTURAL of TWO_CONSECUTIVE is
  signal Y0, Y1, A0, A1: STD_LOGIC:= '0';
  signal NY0, NX: STD_LOGIC:= '1';
  signal ONE: STD_LOGIC:= '1';
  component EDGE_TRIGGERED_D
    port(CLK, D, NCLR: in STD_LOGIC; Q, QN: out STD_LOGIC);
  end component;
---------------- edge triggered register -------------
  component INVG
    port(I: in STD_LOGIC; O: out STD_LOGIC);
  end component;
---------------------- inverter ----------------------
  component AND3G
    port(I1, I2, I3: in STD_LOGIC; O: out STD_LOGIC);
  end component;
---------------------- ANDGATE -----------------------
  component OR2G
    port(I1, I2: in STD_LOGIC; O: out STD_LOGIC);
  end component;
------------------------ ORGATE ----------------------
begin
  C1: EDGE_TRIGGERED_D
    port map(CLK, X, R, Y0, NY0);
  C2: EDGE_TRIGGERED_D
    port map(CLK, ONE, R, Y1, open);
  C3: INVG
    port map(X, NX);
  C4: AND3G
    port map(X, Y0, Y1, A0);
  C5: AND3G
    port map(NY0, Y1, NX, A1);
  C6: OR2G
    port map(A0, A1, Z);
end STRUCTURAL;
```

图7-44　TWO_CONSECUTIVE电路的结构化模型

1) 对于一组延时，可使计数器正确工作的最大时钟频率为多少？

2) 什么是关键信号路径？

这实际上是图7-47中一个更为一般的问题中的一个特例。这里，信号IN在时钟的上升沿进入FF1触发器。FF1的输出驱动一个组合逻辑网络，该网络输出OUT同样在时钟CLK的上升沿由FF2取样。该电路要正确工作，时钟的周期（PER）必须大于数据进入触发器后通过FF1的延时（CLKDEL）和其穿过组合逻辑电路的延时（CLDEL）之和。实际的情况比这严重，必须考虑到FF2的建立时间，但是目前我们将忽略这个问题。

```
entity DFFREG is
  generic (N: Positive :=1;      -- N bit register
     tLH: Time :=  0 ns;         -- rise inertial delay
     tHL: Time :=  0 ns;         -- fall inertial delay
     strn: STRENGTH := strn_X01;-- output strength
     tHOLD : Time := 0 ns;       -- Hold time
     tSetUp : Time := 0 ns;      -- Setup time
     tPwHighMin : Time := 0 ns; -- min pulse width(high)
     tPwLowMin : Time := 0 ns); -- min pulse width(low)
  port (Data: in STD_LOGIC_VECTOR (N-1 downto 0);
     Clock,                      -- clock input
     Reset: in STD_LOGIC;        -- reset input
     Output: out STD_LOGIC_VECTOR (N-1 downto 0);
     NOutput: out STD_LOGIC_VECTOR (N-1 downto 0));
end DFFREG;
entity ANDGATE is
  generic (N:   Positive := 2;-- number of inputs
    tLH: Time := 0 ns;         -- rise inertial delay
    tHL: Time := 0 ns;         -- fall inertial delay
    strn: STRENGTH := strn_X01);-- output strength
  port (Input: in STD_LOGIC_VECTOR (1 to N); -- inputs
    Output: out STD_LOGIC);  -- output
end ANDGATE;
entity INVGATE is
  generic (tLH: Time := 0 ns; -- rise inertial delay
    tHL: Time := 0 ns;         -- fall inertial delay
    strn: STRENGTH := strn_X01);  -- output strength
  port (Input: in STD_LOGIC;       -- input
    Output: out STD_LOGIC);        -- output
end INVGATE;
```

图7-45 模块接口描述

在图7-36中的计数器例子中，两个高位计数器触发器是FF1和FF2。组合逻辑是单个AND门。该电路中将会出现的一个竞争状态如下：假设计数器处于状态CNT（2）CNT（1）CNT（0）=010。这种状态下，AND门的输出（信号C01）是‘0’。如果那时输入时钟信号，计数器在CLKDEL后将会改变为值011。接下来，AND门的输出C01将在门延时（DEL）之后变为“1”。在这种情况下，CLKDEL+CLDEL必须小于HI_TIME+LO_TIME，即晶振周期。图7-48给出了CTR的一个配置FAST，它满足了这个条件。正如图7-49所示，在C01上的信号改变在时钟上升沿前1ns到达。然而对于配置TOOFAST（图7-48），不满足该条件，信号C01改变为‘1’迟了1ns。作为结果，计数器如下计数：000，001，010，011，000，001等等，而非正常的8状态计数序列。电路配置的使用使得学习延时问题更加容易。

配置声明中，门级模型的精确时序也允许对时序险态的真实建模。图7-50给出了一个存在险态的电路。A、B和C是电路的输入，产生触发器的复位信号R。信号R的逻辑功能的卡诺图同样在图中给出。考虑这种情况，其中输入组合ABC初始化为111但切换到101，即B转换到低电平。对卡诺图的研究表明输入组合111和101产生输出1，但是当切换的时候，特定的门和反相器立刻使输出转到0。

为了研究延时的哪种组合产生险态，创建该电路的一个非绑定结构模型是很有用的方法，然后用各种包含不同延时组的配置来进行试验，以确定在哪种情况之下，险态引起了毛刺。图7-51给出了电路的一个结构模型。注意：所有组件都被声明为非绑定。

```
configuration PARTS1 of TWO_CONSECUTIVE is
  for STRUCTURAL
    for all: EDGE_TRIGGERED_D use entity work.DFFREG(A)
      port map(Clock => CLK, Data(0) => D, Reset => NCLR,
                        Output(0) => Q, Noutput(0)=> QN);
    end for;
    for all: INVG use entity INVGATE(A)
      port map(Input => I, Output => O);
    end for;
    for all: AND3G use entity ANDGATE(A)
      generic map(N => 3)
      port map(Input(1) => I1, Input(2) => I2,
               Input(3) => I3, Output => O);
    end for;
    for all: OR2G use entity ORGATE(A)
      generic map(N => 2)
      port map(Input(1) => I1, Input(2) => I2, Output => O);
    end for;
  end for;
end PARTS1;

configuration PARTS2 of TWO_CONSECUTIVE is
  for STRUCTURAL
    for all: EDGE_TRIGGERED_D use entity work.DFFREG(A)
      generic map(TLH=> 5 ns, THL=> 6 ns)
      port map(Clock => CLK, Data(0) => D, Reset => NCLR,
                        Output(0) => Q, Noutput(0)=> QN);
    end for;
    for all: INVG use entity INVGATE(A)
      generic map(TLH=> 1 ns, THL=> 2 ns)
      port map(Input => I, Output => O);
    end for;
    for all: AND3G use entity ANDGATE(A)
      generic map(N => 3, TLH=> 3 ns, THL=> 4 ns)
      port map(Input(1) => I1, Input(2) => I2,
               Input(3) => I3, Output => O);
    end for;
    for all: OR2G use entity ORGATE(A)
      generic map(N => 2, TLH=> 2 ns, THL=> 3 ns)
      port map(Input(1) => I1, Input(2) => I2, Output => O);
    end for;
  end for;
end PARTS2;
```

图7-46 绑定到部件库的两个配置

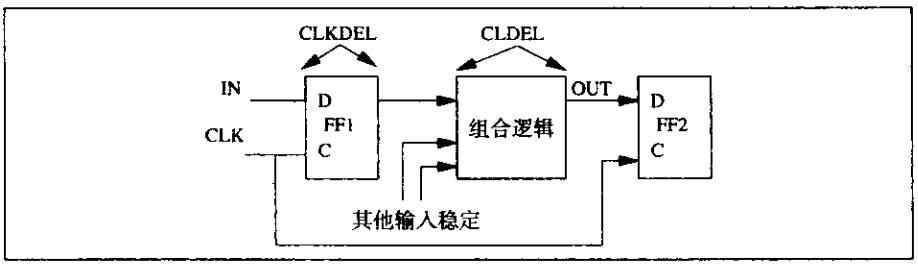


图7-47 一个基本时序问题

图7-52给出电路的一个配置，它将会产生一个短时脉冲信号干扰。在这个配置中，信号B

的数据通路通过反相器C4和AND门C2，它比上面经过C1的路径要长。图7-53给出了模型当输入组合从ABC=111改变为ABC=101时的时间响应，即输入B从1转换到0。在下面的AND门的输出信号A2转换到高电平3ns之后，上面的AND门的输出信号A1切换为低电平，在信号R上产生了3ns的0电平的干扰电压，会引起触发器Q复位。

```
configuration FAST of CTR is
  for STRUCTB
    for C0: OSC use entity work.COSC(WAIT_DEL)
      generic map(11 ns, 10 ns);
    end for;
    for C1: UP_CNT use entity work.UP_CNT(STRUCT)
      generic map(10 ns, 15 ns);
      for STRUCT
        for all: T use entity work.TFF(ALG)
          generic map(10 ns, 15 ns);
        end for;
        for C4: AND2G use entity work.AND2(BEH)
          generic map(5 ns);
        end for;
      end for;
    end for;
  end for;
end FAST;

configuration TOO_FAST of CTR is
  for STRUCTB
    for C0: OSC use entity work.COSC(WAIT_DEL)
      generic map(10 ns, 9 ns);
    end for;
    for C1: UP_CNT use entity work.UP_CNT(STRUCT)
      generic map(10 ns, 15 ns);
      for STRUCT
        for all: T use entity work.TFF(ALG)
          generic map(10 ns, 15 ns);
        end for;
        for C4: AND2G use entity work.AND2(BEH)
          generic map(5 ns);
        end for;
      end for;
    end for;
  end for;
end TOO_FAST;
```

图7-48 用于为竞争条件建模的配置

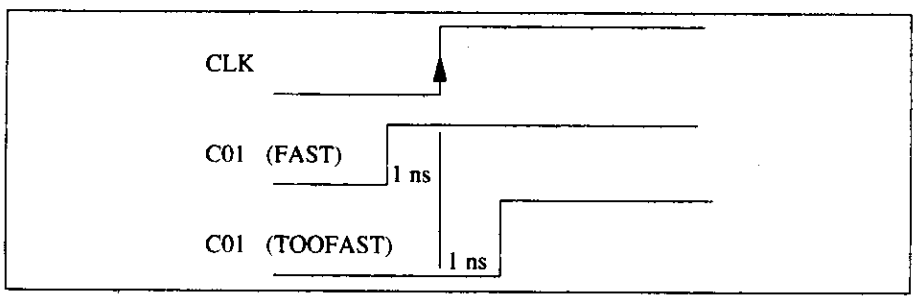


图7-49 计数器竞争条件

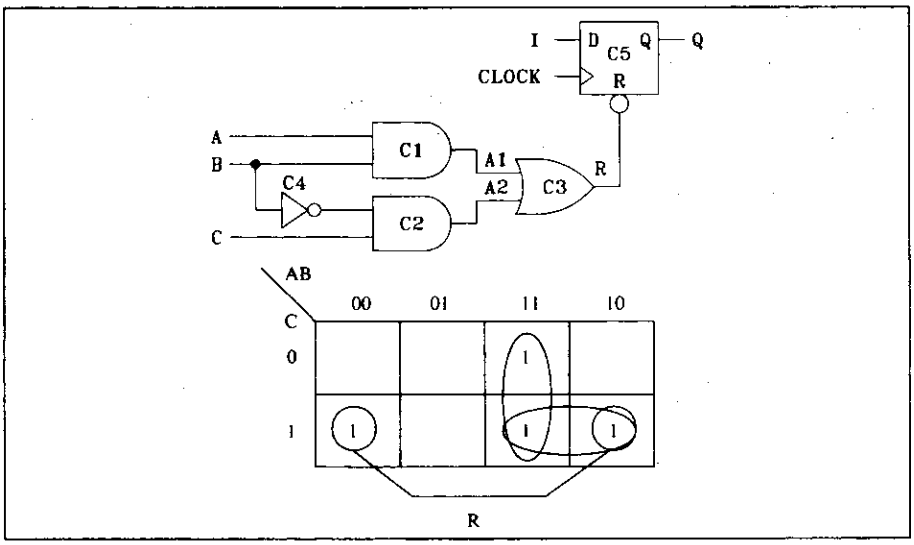


图7-50 险态电路

```
library SYNOPSYS;
use SYNOPSYS.TYPES.all;
entity RGLITCH is
  port(A,B,C,I,CLK: in MVL7; Q: out MVL7);
end RGLITCH;
architecture STRUCT of RGLITCH is
  signal A1,A2,NB,R : MVL7;
  component DFF
    port(R,CLK,D: in MVL7; Q: out MVL7);
  end component;
  component AND2
    port(I1,I2: in MVL7; O: out MVL7);
  end component;
  component OR2
    port(I1,I2: in MVL7; O: out MVL7);
  end component;
  component INV
    port(I: in MVL7; O: out MVL7);
  end component;
begin
  C1: AND2
    port map(I1 => A, I2  => B, O => A1);
  C2: AND2
    port map(I1 => NB, I2 => C, O => A2);
  C3: OR2
    port map(I1 => A1, I2 => A2, O => R);
  C4: INV
    port map(I => C, O => NB);
  C5: DFF
    port map(R => R, CLK => CLK, D => I, Q => Q);
end STRUCT;
```

图7-51 非绑定结构化构架

刚才分析的险态叫逻辑险态。可以通过给R的布尔表达式添加项A C来去除险态。当B从1变到0，这个项会使OR门的输出保持为逻辑1，如图7-50卡诺图中用虚线框所示。见可以看出，险态通常可以通过在其表达式中包括所有函数主项而消除。

```
configuration GLTCH of RGLITCH is
  for STRUCT
    for all: DFF use entity WORK.DFFREG(A)
      generic map(TLH => 5 ns, THL => 6 ns)
      port map(CLOCK => CLK, DATA(0) => D, RESET => R,
               OUTPUT(0) => Q);
    end for;
    for all: INV use entity SYNOPSYS.INVGATE(A)
      generic map(TLH => 2 ns, THL => 3 ns)
      port map(INPUT => I, OUTPUT => O);
    end for;
    for C1 : AND2 use entity SYNOPSYS.ANDGATE(A)
      generic map(N => 2, TLH => 2 ns, THL => 3 ns)
      port map(INPUT(1) => I1, INPUT(2) => I2, OUTPUT => O);
    end for;
    for C2 : AND2 use entity SYNOPSYS.ANDGATE(A)
      generic map(N => 2, TLH => 4 ns, THL => 5 ns)
      port map(INPUT(1) => I1. INPUT(2) => I2, OUTPUT => O);
    end for;
    for all: OR2 use entity SYNOPSYS.ORGATE(A)
      generic map(N => 2, TLH => 2 ns, THL => 3 ns)
      port map(INPUT(1) => I1, INPUT(2) => I2, OUTPUT => O);
    end for;
  end for;
end GLTCH;
```

图7-52 电路的可产生干扰配置

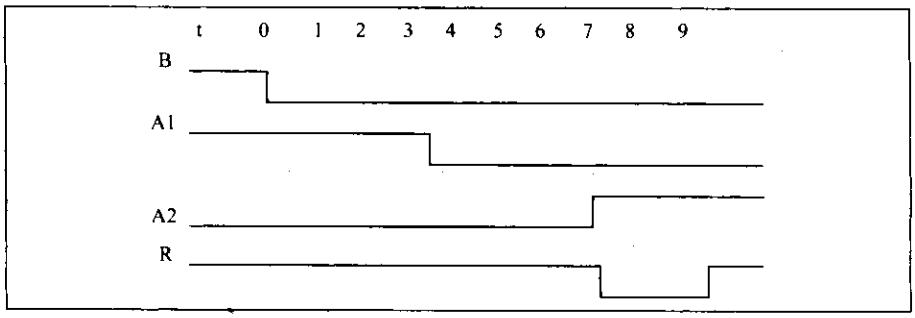


图7-53 逻辑险态的作用

另一种类型的险态是功能险态。这里，当从一种输出组合切换到另一种时，也会导致瞬间的不正确输出。然而，向布尔表达式中加入项不能改正这种类型的险态。信号R的功能险态是当输入组合ABC=001变到组合ABC=110时产生的。当信号A和B通过AND门C1的路径较信号C通过AND门C2的路径长时，电路对这种故障很敏感。图7-54给出了引起这种情况的一个配置。图7-55给出了电路响应。在这种情况下，A1的上升落后于A2的下降，在R上引起尖锋信号，使触发器复位。

有趣的是，这两种险态由两种完全不同的延时情况引起。正如所见的，使用不同的配置体是使一个电路感知不同险态的一种非常有效的办法。

7.6 延时控制的方法

本节介绍一些进行延时控制的不同方法。考虑图7-56中所示的电路，三种可能的延时建

模的情况如下：

```
configuration GLTCH2 of RGLITCH is
  for STRUCT
    for all: DFF use entity WORK.DFFREG(A)
      generic map(TLH => 5 ns, THL => 6 ns)
      port map(CLOCK => CLK, DATA(0) => D, RESET => R,
               OUTPUT(0) => Q);
    end for;
    for all: INV use entity SYNOPSYS.INVGATE(A)
      generic map(TLH => 2 ns, THL => 3 ns)
      port map(INPUT => I, OUTPUT => O);
    end for;
    for C1 : AND2 use entity SYNOPSYS.ANDGATE(A)
      generic map(N => 2, TLH => 4 ns, THL => 5 ns)
      port map(INPUT(1) => I1,INPUT(2) => I2,OUTPUT => O);
    end for;
    for C2 : AND2 use entity SYNOPSYS.ANDGATE(A)
      generic map(N => 2, TLH => 1 ns, THL => 2 ns)
      port map(INPUT(1) => I1,INPUT(2) => I2, OUTPUT => O);
    end for;
    for all: OR2 use entity SYNOPSYS.ORGATE(A)
      generic map(N => 2, TLH => 1 ns, THL => 2 ns)
      port map(INPUT(1) => I1,INPUT(2) => I2, OUTPUT => O);
    end for;
  end for;
end GLTCH2;
```

图7-54 另一个产生干扰的配置

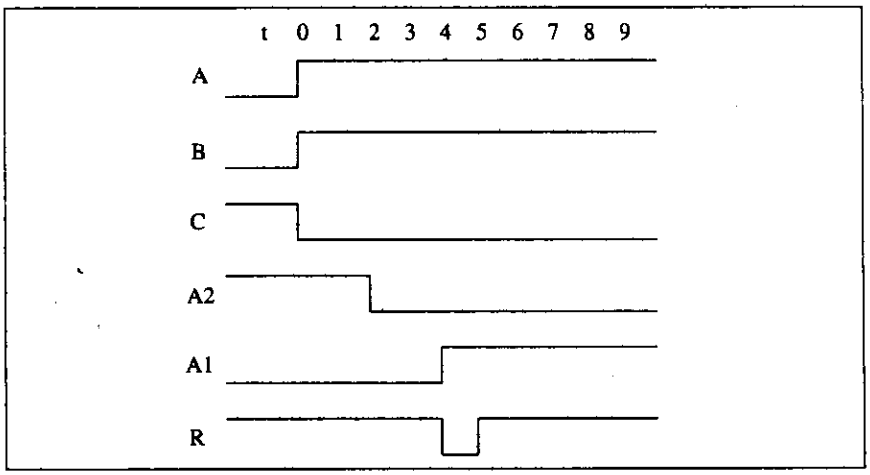


图7-55 功能险态的作用

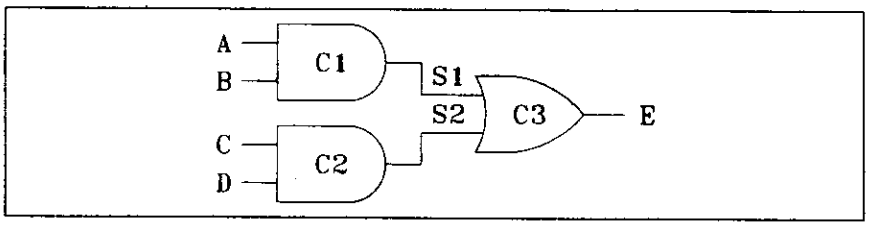


图7-56 有三个门的电路

1) 所有的门有相同的延时。

2) 所有特定类型的门有相同的延时。

3) 每个门可能有不同的延时。

我们使用图7-57的VHDL代码，用有三个门的电路来讲解这三种情况。门模型AND2G和OR2G使用的是基本门模型，每个有类属延时参数DEL。实体THREE_GATES的构架AND_OR给出了一个电路的结构模型，其中AND2和OR2的组件声明是非绑定的。前面列出的三个延时映射通过构架AND配置ONE、TWO和THREE这三个配置来实现。在配置ONE中，实体THREE的类属GATE_DEL映射到每个门的类属DEL，这样，所有的门都有相同的2ns延时。在配置TWO中，所有的AND门都提供了同样的延时（2ns），所有的OR门都有同样的延时（3ns）。在配置THREE中，对AND门C1赋值2ns延时，对AND门C2是3ns延时，对OR门C3是4ns延时。

应该注意，用配置来进行这种类型映射是不必要的。它们能够通过使用结构化构架中的配置说明来实现。然而，配置声明的使用是一种有序的方法，以进行不同的映射，改变延时特性并不需要重分析构架。

习题

7.1 为2输入NAND门SN7400产生一个VHDL模型，使用类属表示TPHL和TPLH，使用典型值仿真这个门。

7.2 为2输入NAND门建立VHDL模型，由于金属线不同，该与非门具有不对称的输入延时，假设这个NAND门被用于图7-58所示的门级电路，并假设金属线长度与图中线段距离成正比，参阅有关门阵列系列的资料，得出延时值与金属线长度，使用这些值对电路进行仿真。内部门延时的值也可以从门阵列中得出。

7.3 本题的目的是为了熟悉EIA时序和环境规范，假设一个1延时值的线性模型，使用实验室测试及厂商提供的数据为SN7400 2输入NAND门的Tpd建立一个表达式。如果可能，考虑电源电压和温度的影响。

7.4 建立具有以下特征的NAND门模型：

a) 通用N输入模型，N是实体描述中的一个类属。

b) 一个受扇出影响的延时函数，当初始化为一个特定电路时，每个门都可计算出自己的扇出。使用该模型对图7-25的逻辑电路图进行建模。

7.5 使用本章中给出的时序控制包，使用delta延时，最小、典型和最大时序为图7-59的门级电路建模。

7.6 图7-5给出的电路具有一个险态，它的输出反馈到一个触发器的复位输入端。本例中使用的触发器是一个库触发器。建立一个自己的触发器模型，该模型应能够检测自身的置位和复位，并仿真险态出现的情况。

7.7 逻辑门一般需要使用双值系统，但实际电路模型的效果需要使用多值逻辑。列出需要使用特殊值的情况，并给出这些值的一般符号表示。

7.8 为三态输出SN74LS240缓冲器建立一个VHDL模型。引入时序以对模型的数据延时和使能延时精确建模，包括从高阻状态开始的切换及到高阻状态的切换。

7.9 为使用图7-60九值逻辑系统（不是IEEE值系统）的NMOS转换器建模并进行仿真。

```
entity  OR2G is
  generic(DEL: TIME);
  port(I1,I2: in BIT; O: out BIT);
end OR2G;
architecture GNR_DEL of OR2G is
begin
  O <= I1 or I2 after DEL;
end GNR_DEL;

entity  AND2G is
  generic(DEL: TIME);
  port(I1,I2: in BIT; O: out BIT);
end AND2G;
architecture GNR_DEL of AND2G is
begin
  O <= I1 and I2 after DEL;
end GNR_DEL;

entity THREE_GATES is
  generic(GATE_DEL: TIME:= 2 ns);
  port(A,B,C,D: in BIT; E: out BIT);
end THREE_GATES;
architecture AND_OR of THREE_GATES is
  signal S1,S2: BIT;
  component  OR2
    port(I1,I2: in BIT; O: out BIT);
  end component;
  component  AND2
    port(I1,I2: in BIT; O: out BIT);
  end component;
begin
  C1: AND2
    port map(I1 => A, I2 => B, O => S1);
  C2: AND2
    port map(I1 => C, I2 => D, O => S2);
  C3: OR2
    port map(I1 => S1, I2 => S2, O => E);
end AND_OR;

configuration ONE of THREE_GATES is
  for AND_OR
    for all: AND2 use entity work.AND2G(GNR_DEL)
      generic map(DEL => GATE_DEL);
    end for;
    for all: OR2 use entity  work.OR2G(GNR_DEL)
      generic map(DEL => GATE_DEL);
    end for;
  end for;
end ONE;
```

图7-57 延时映射

7.10 使用IEEE九值系统实现一个NAND门模型，对于图7-60给出的RS触发器，假设两个门的输出初始值都为U，而且两个输入R和S 初始为0且同时变为1，在下面情况下仿真该系统。

a) 两个门具有相同的延时。

b) 两个门具有不同的延时，比较结果。

7.11 使用更多的逻辑值被认为会产生较少不利的仿真结果，即更少的x值（未知值）。设计并

7.12

仿真一个验证此说法的实验电路。

```
configuration TWO of THREE_GATES is
  for AND_OR
    for all: AND2 use entity work.AND2G(GNR_DEL)
      generic map(DEL => 2 ns);
    end for;
    for all: OR2 use entity  work.OR2G(GNR_DEL)
      generic map(DEL => 3 ns);
    end for;
  end for;
end TWO;

configuration THREE of THREE_GATES is
  for AND_OR
    for C1: AND2 use entity work.AND2G(GNR_DEL)
      generic map(DEL => 2 ns);
    end for;
    for C2: AND2 use entity work.AND2G(GNR_DEL)
      generic map(DEL => 3 ns);
    end for;
    for C3: OR2 use entity  work.OR2G(GNR_DEL)
      generic map(DEL => 4 ns);
    end for;
  end for;
end THREE;
```

图7-57 延时映射（续）

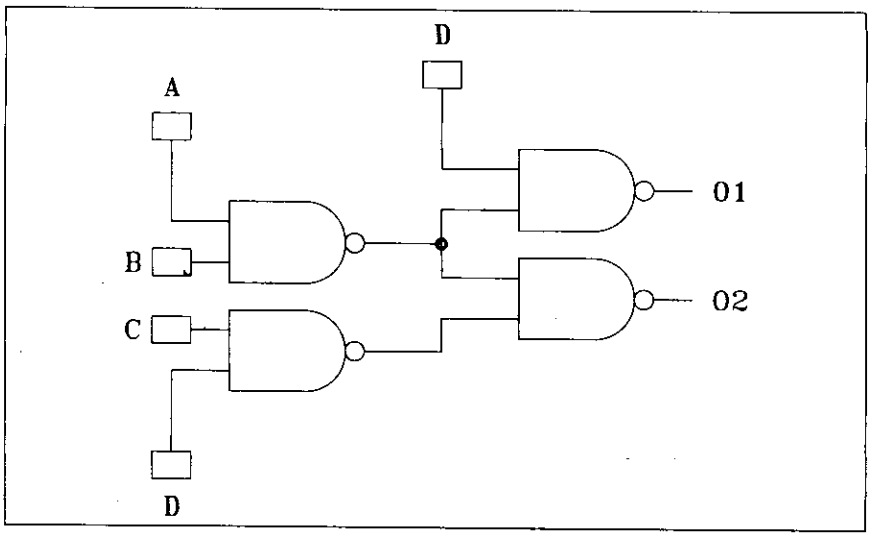


图7-58 金属线

7.12 多值逻辑可以用于发现时序电路中存在的不稳定性，考虑图7-61中简单的D触发器电路，假设所有元件都为delta延时，初始时D=C=Q=1。

a) 当t=0时把C变为X，然后在t=10时C变为0，对电路进行仿真，验证结果Q=X。（这个仿真过程被称为Eichelbenger方法，可参阅有关开关理论和逻辑设计的书进一步研究此问题。）

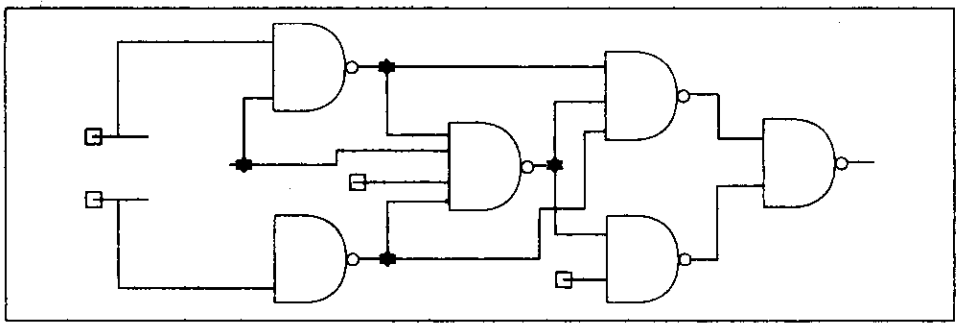


图7-59　电路扇出

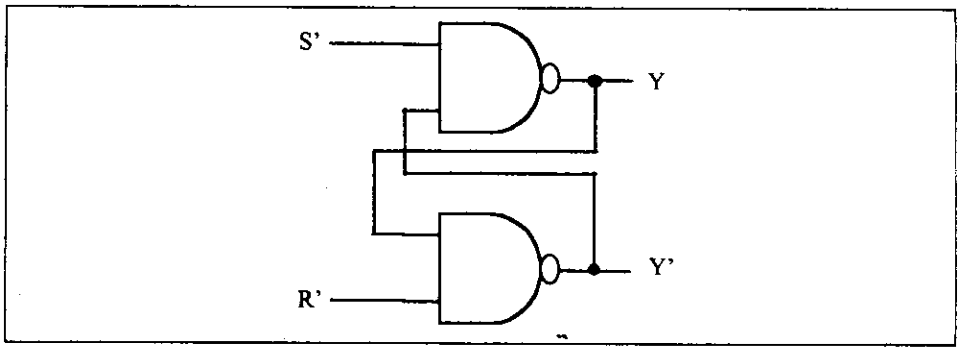


图7-60　RS触发器

b) 为电路延时赋值，输出将首先消除了X，但接着又会产生X。

c) 用D、C和Q为q产生布尔表达式及卡诺图，通过在表达式中增加合适的项来消除卡诺图中的所有静态险态，重复a证明电路是稳定的而且与延时无关。

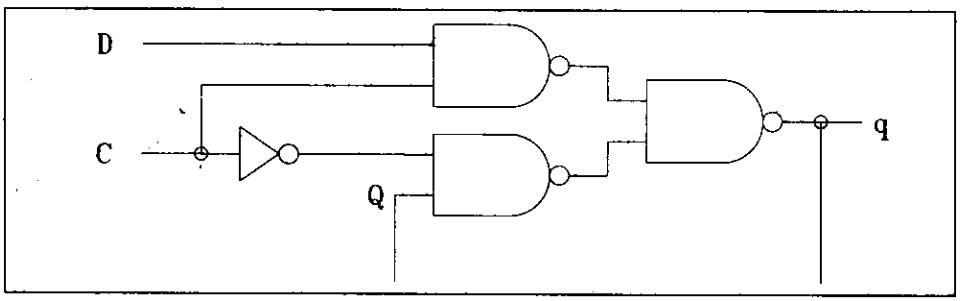


图7-61　D触发器

7.13　使用抽象代数学的方法证明WiredX判决函数是关联的并且是可交换的。

7.14　下面是系统HILO的值系统；

```
          X
        U   D
      T   W   B
    P   R   F   N
1     H     Z     L     0
```

给出下面值的含义：1，H，Z，L，0，F，U。

7.15 假设一个12值系统的产生过程如下：

a) 三个状态：0、1和x，这些状态课本中已定义。

b) 四个强度值：F、R、Z和U。F、R和Z在课本中已经定义，U为一个未知强度。

因此，这里有一个未知状态（X）和一个未知强度（U）。

a) 给出系统的12个值。

b) 把系统划分为状态强度或区间系统。

c) 考虑图7-62给出的高阻缓冲器，假设在它的输出有电容性存储，数据输入、输出和使能的初始状态已经给出。若使能由F0变为FX，则输出值变为什么？解释答案。

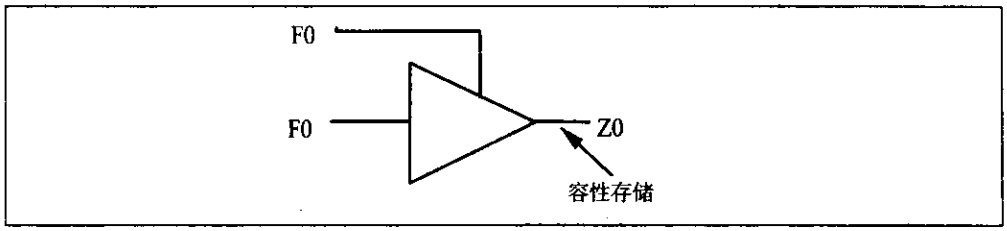


图7-62 高阻缓冲器

7.16 当互连的VHDL模型使用不同值系统时会遇到一些困难。在MVL和IEEE九值系统间建立映射，使用抽象代数中的原理讨论映射的代数属性。

7.17 对于使用状态强度模型的多值系统，图7-63输出的结果是什么？

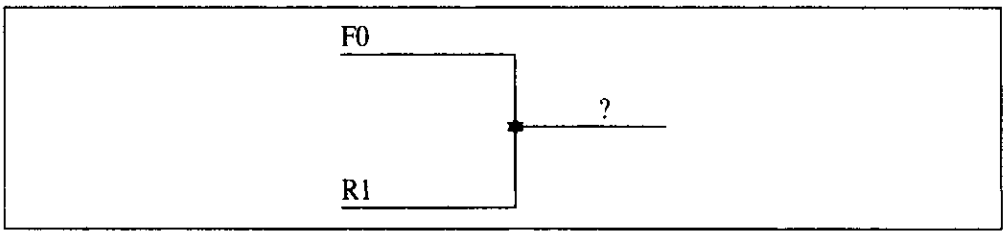


图7-63 总线判决

7.18 为第4章的PLA（图4-39）的单个行为模型产生一个开关级的结构模型。

7.19 对MOS开关使用Smith和Acosta模型，仿真一个小型MOS电路并评价结果。

7.20 为图7-64给出的非绑定电路建立一个结构化模型，使用配置体把电路绑定到一个模型库，库中的所有模型都具有输入过载检测，使用卡诺图分析判断电路的静态险态或功能险态以及哪些输入组合可以产生这些险态。使用配置体中的延时，使电路达到不同的险态条件。仿真该电路来验证所有的险态均可以被检测。

7.21 本题是对习题4.19的重复，习题4.19中的计数器由晶体振荡器、JK触发器和一个AND门产生，延时由晶体振荡器高电平和低电平、门和触发器的TPLH和TPHL所定义。为计数器模型产生一个非绑定的结构化模型，然后产生配置体对这些延时赋值。对于每个触发器和门的TPLH和TPLH值,决定晶体振荡器的最大可能频率。关键路径是什么？用关键路径延时表示最大振荡器频率，仿真并验证你的结果。

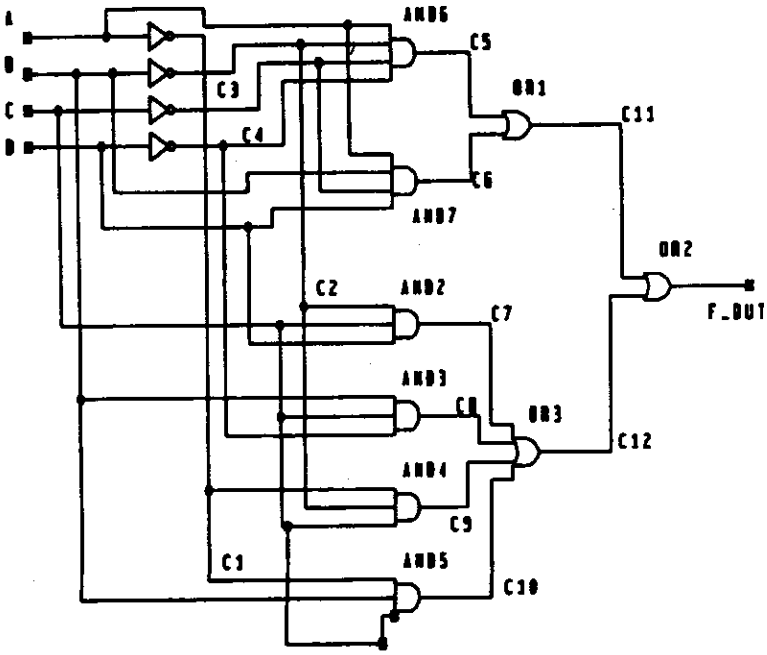

A
B
C
D
C3
C4
AND6
C5
OR1
C11
C6
AND7
OR2
C2
AND2
C7
F_OUT
AND3
C8
OR3
C12
AND4
C9
C1
AND5
C10
解释险态的四输入变量电路
SUN_OF_PRODUCT (-SOP) 格式

图7-64 险态电路

第8章　基于HDL的设计技术

本章将讨论组合逻辑和时序逻辑的结构化设计方法。我们将结合HDL（硬件描述语言）、卡诺图和状态表，讨论一种从算法描述到门级实现的、设计组合逻辑和时序逻辑的方法。本章包括了一个与微程序控制单元设计结合的方法。

8.1　组合逻辑电路的设计

本节讨论组合逻辑电路的设计问题。在本节的方法被应用之前，设计者必须首先决定在系统设计中使用的是组合逻辑。该决定必须基于部分设计者对设计规格说明和需求分析的判断。例如，考虑图8-1中所示的器件框图。

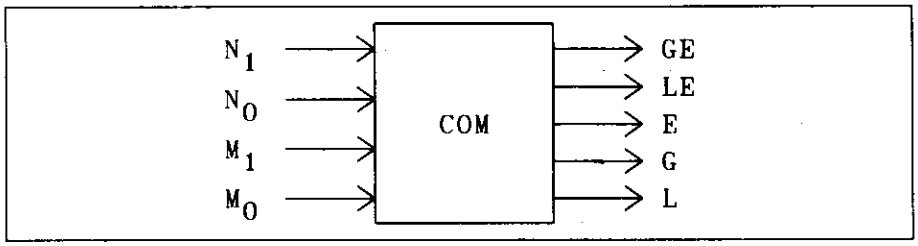


图8-1　二进制比较器框图

例8.1　二进制比较器(COM)的规格说明。

这是用来比较两个二进制数值的大小，并判断哪个更大一些的器件。输入是2位二进制数值，表示为$N=N_1N_0$和$M=M_1M_0$，输出为二进制信号GE、LE、E、G和L，定义见表8-1中。

表8-1　二进制输出信号

信　号	说　明
GE	二进制输出，当N大于等于M时为真
LE	二进制输出，当N小于等于M时为真
E	二进制输出，当N等于M时为真
G	二进制输出，当N严格大于M时为真
L	二进制输出，当N严格小于M时为真

从规格问题说明和框图，很容易生成器件COM的实体声明，见图8-2。类属变量D用来指定器件从输入到输出的时间延迟。有四个输入端口（N1、N0、M1和M0）分别表示二进制输入N_1、N_0、M_1和M_0。有五个输出端口（GE、LE、E、G和L）表示器件COM的五个二进制输出。

读了该说明之后，很清楚看出系列输出是完全独立的。因此，可以使用组合逻辑来实现该器件。如果决定使用组合逻辑实现，那么可采用本节讨论的方法。一个电路存在组合逻辑的实现并不影响对同一电路也可能存在时序逻辑的实现。实际上，如果输入保存在寄存器中，

则器件就可用时序电路来实现的，本章后面将进行介绍。

```
--
-- Device to compare two binary inputs.
--
entity COM is
  generic (D:time);
  port (N1, N0, M1, M0: in BIT;
        GE, LE, E, G, L: out BIT);
end COM;
```

图8-2 器件COM的实体规格说明

8.1.1 算法级的组合逻辑设计

如果允许，设计组合逻辑的第一步是创建一张真值表。真值表列出对应于输入的每种组合的输出值。本问题的真值表见图8-3。

N_1	N_0	M_1	M_0	GE	LE	E	G	L
0	0	0	0	1	1	1	0	0
0	0	0	1	0	1	0	0	1
0	0	1	0	0	1	0	0	1
0	0	1	1	0	1	0	0	1
0	1	0	0	1	0	0	1	0
0	1	0	1	1	1	1	0	0
0	1	1	0	0	1	0	0	1
0	1	1	1	0	1	0	0	1
1	0	0	0	1	0	0	1	0
1	0	0	1	1	0	0	1	0
1	0	1	0	1	1	1	0	0
1	0	1	1	0	1	0	0	1
1	1	0	0	1	0	0	1	0
1	1	0	1	1	0	0	1	0
1	1	1	0	1	0	0	1	0
1	1	1	1	1	1	1	0	0

图8-3 2位比较器的真值表

在这一节，我们介绍在行为域算法级的两种对真值表进行建模的方法。ARRAY模型可以直接映射为一种ROM实现。CASE模型可以直接映射为多路器的实现。通过使用优化过程，可以设计出几种不同的多路器的实现。

1. ARRAY模型

该方法使用多维数组来表示真值表，见图8-4。程序包TRUTH4×5用来定义数据。常量NUM_OUT PUTS是输出个数，对于器件COM为5。COM的输出是GE、LE、E、G和L。常量NUM _INPUTS是输入个数，对于器件COM为4，即输入是N_1、N_0、M_1、M_0。常量NUM_ROWS是真值表的行数，通过2^{num_inputs}进行计算。对于器件COM，这里的值是16。对真值表的每一行，5个输出可以表示为位的数组，称之为WORD。例如行0的WORD在真值表中是（1，1，1，0，0）。数组的元素通过下标4到0进行引用。整个真值表（TRUTH）是WORD的数组（类型MEM）。注意，这个数组的下标值是十进制数0到15。整个数组在TRUTH程序

单的查找表。首先，使用函数INTVAL将输入值的数组转化成十进制的下标（INDEX）。这个十进制下标用来从常量TRUTH中选出一个WORD，该WORD中含有想要的5个输出（WOUT）。这五个输出在延时D之后被赋予想要的值。

ARRAY模型可以综合为ROM实现，ROM的内容由数组常量指定。器件COM的输入被连入ROM的地址输入。ROM的数据输出连至器件的输出。例如，实现COM的ROM将需要16个5位字（16×5 ROM）。见图8-5中该实现的图形表示。正如第10章中所讲述的，自动化综合工具通常不实现全ROM形式，但确定推断的逻辑电路，并实现其最小的版本。

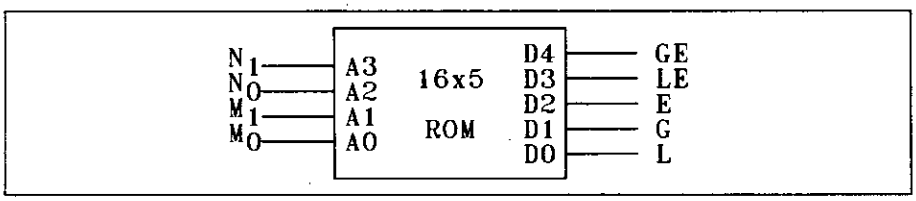


图8-5　直接将ARRAY模型转化成ROM

2. CASE模型

在该模型中，真值表的每一行对应于CASE语句的一个选项。图8-6给出了器件COM的VHDL CASE模型。

该构架被命名为MUX是因为其结构与硬件多路器极其相似。构架包含一个进程，该进程在器件的任何一个输入改变时被激活。该进程包括一个大的CASE语句，对于16种可能的输入组合的任何一个，将逻辑值赋给每一个输出。这些值直接产生于问题的规格说明。例如，如果$N_1N_0M_1M_0$=0101即N=01和M=01，则两个数相等。因此GE=LE=E=1且L=G=0。同样，如果$N_1N_0M_1M_0$=0111，即N=01和M=11，则N<M。因此L=LE=1且GE=E=G=0。对于输入变量的每种组合，输出都直接由规格说明导出。

该模型可以直接转换成硬件多路器。图8-7给出了输出信号GE的一个多路器实现。器件输入N_1、N_0、M_1、M_0直接连入MUX的地址输入，N_1连入高端地址输入，M_0连入低端地址输入。连入数据输入D*i*的逻辑常量是CASE *i*的GE值。例如，输入D5=1，因为对于CASE $N_1N_0M_1M_0$=0101，GE=1。输出信号LE、E、G和L可以用其他4个多路器来实现。

3. 算法级的优化过程

前一节的转换方式是产生多路器实现的最直接方式。然而，还有其他更有效的多路器实现。这些可以通过对VHDL模型进行转换，并将得到的模型转换为硬件来实现。这种转换过程称作优化步骤。这些步骤需要一些智能决定，因而需要一些形式化的人工智能技术使该过程自动化。本节将探讨应用于MUX类型实现的优化过程。

一种相对直接的优化方法是通过任意选择一个输入变量，将其从多路器的地址集合中删除。为了说明该算法，我们随意选择变量M0，将其从地址集合中删除，也可以选择任何其他的变量。图8-8给出实施了优化之后的新的VHDL描述。

转换过程如下。在图8-6中，考虑“0000”和“0001”两种情况。在优化的描述中用“000”来代表有这两种情况，基于变量N1、N0和M1，见图8-8。变量的信号赋值如下实现，比较每个输出变量（GE、LE、E、G和L）原来的VHDL描述中需要的信号赋值与输入信号M0的值，将存在四种可能性：

```
-- Standard model for combinational logic,using CASE statement.
-- This model can be directly translated
-- into a MUX implementation.
architecture MUX of COM is
begin
  process(N1,N0,M1,M0)
  begin
    case N1&N0&M1&M0 is
      when "0000" => GE <= '1' after D; LE <= '1' after D;
        E <= '1' after D; G <= '0' after D; L <= '0' after D;
      when "0001" => GE <= '0' after D; LE <= '1' after D;
        E <= '0' after D; G <= '0' after D; L <= '1' after D;
      when "0010" => GE <= '0' after D; LE <= '1' after D;
        E <= '0' after D; G <= '0' after D; L <= '1' after D;
      when "0011" => GE <= '0' after D; LE <= '1' after D;
        E <= '0' after D; G <= '0' after D; L <= '1' after D;
      when "0100" => GE <= '1' after D; LE <= '0' after D;
        E <= '0' after D; G <= '1' after D; L <= '0' after D;
      when "0101" => GE <= '1' after D; LE <= '1' after D;
        E <= '1' after D; G <= '0' after D; L <= '0' after D;
      when "0110" => GE <= '0' after D; LE <= '1' after D;
        E <= '0' after D; G <= '0' after D; L <= '1' after D;
      when "0111" => GE <= '0' after D; LE <= '1' after D;
        E <= '0' after D; G <= '0' after D; L <= '1' after D;
      when "1000" => GE <= '1' after D; LE <= '0' after D;
        E <= '0' after D; G <= '1' after D; L <= '0' after D;
      when "1001" => GE <= '1' after D; LE <= '0' after D;
        E <= '0' after D; G <= '1' after D; L <= '0' after D;
      when "1010" => GE <= '1' after D; LE <= '1' after D;
        E <= '1' after D; G <= '0' after D; L <= '0' after D;
      when "1011" => GE <= '0' after D; LE <= '1' after D;
        E <= '0' after D; G <= '0' after D; L <= '1' after D;
      when "1100" => GE <= '1' after D; LE <= '0' after D;
        E <= '0' after D; G <= '1' after D; L <= '0' after D;
      when "1101" => GE <= '1' after D; LE <= '0' after D;
        E <= '0' after D; G <= '1' after D; L <= '0' after D;
      when "1110" => GE <= '1' after D; LE <= '0' after D;
        E <= '0' after D; G <= '1' after D; L <= '0' after D;
      when "1111" => GE <= '1' after D; LE <= '1' after D;
        E <= '1' after D; G <= '0' after D; L <= '0' after D;
    end case;
  end process;
end MUX;
```

图8-6 使用CASE方法得到器件COM的VHDL模型

1) 如果两种情况中输出信号与变量M0相等，则将输出信号赋为M0。

2) 如果两种情况中，输出信号是M0的非，则输出信号赋值为“not M0”。

3) 如果在两种情况中，输出信号是逻辑‘0’，则输出信号赋为‘0’。

4) 如果在两种情况中，输出信号是逻辑‘1’，则输出信号赋为‘1’。

在原先的VHDL描述中，对于情况“0000”（其中M0=0）时变量G被赋予‘1’，对于情况“0001”（其中M0=1）时变量G被赋予‘0’。因而，对于优化描述中的情况“000”，GE赋为“not M0”。同样，因为LE在原来VHDL描述的“0000”与“0001”两种情况之下都是‘1’，在优化描述中，对于情况“000”时赋值LE=‘1’。因为两种情况下G=‘0’，在情况“000”中赋值G=‘0’。最后，由于在情况“0000”（M0=0）时，L=‘0’，在情况“0001”（M0=‘1’）

时，L=‘1’，在优化的VHDL描述的情况“000”下，赋值L=M0。通过检查所有这些只在变量M0中不同的情况对，可以容易得出图8-8中的精简的描述。

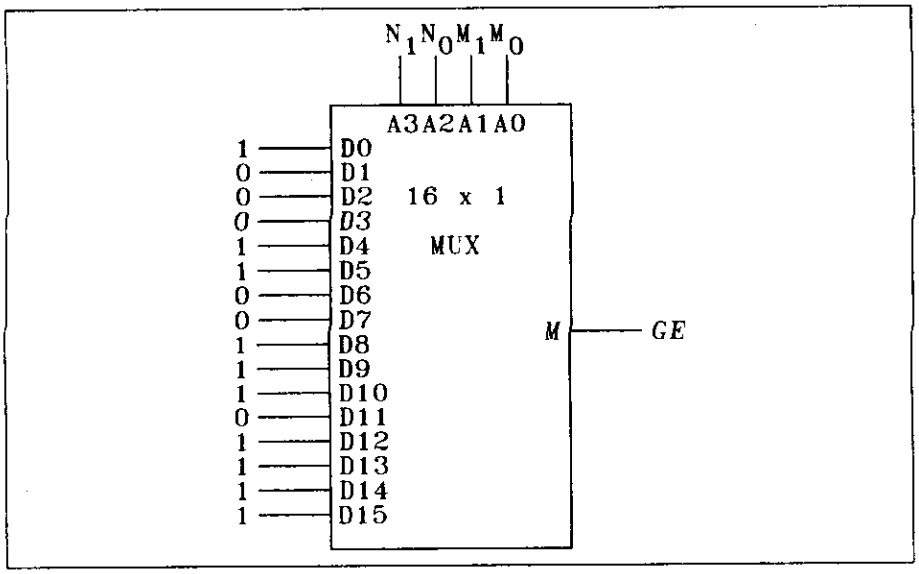


图8-7 直接将CASE模型转换成MUX实现

```
-- Improved CASE description. This model directly
-- corresponds to an 8x1 MUX implementation.
architecture MUX3 of COM is
begin
 process (N1, N0, M1, M0)
 begin
  case N1&N0&M1 is
   when "000" => GE <= not M0 after D; LE <= '1' after D;
    E <= not M0 after D; G <= '0' after D; L <= M0 after D;
   when "001" => GE <= '0' after D; LE <= '1' after D;
    E <= '0' after D; G <= '0' after D; L <= '1' after D;
   when "010" => GE <= '1' after D; LE <= M0 after D;
    E <= M0 after D; G <= not M0 after D; L <= '0' after D;
   when "011" => GE <= '0' after D; LE <= '1' after D;
    E <= '0' after D; G <= '0' after D; L <= '1' after D;
   when "100" => GE <= '1' after D; LE <= '0' after D;
    E <= '0' after D; G <= '1' after D; L <= '0' after D;
   when "101" => GE <= not M0 after D; LE <= '1' after D;
    E <= not M0 after D; G <= '0' after D; L <= M0 after D;
   when "110" => GE <= '1' after D; LE <= '0' after D;
    E <= '0' after D; G <= '1' after D; L <= '0' after D;
   when "111" => GE <= '1' after D; LE <= M0 after D;
    E <= M0 after D; G <= not M0 after D; L <= '0' after D;
  end case;
 end process;
end MUX3;
```

图8-8 器件COM的改进CASE型的VHDL描述

优化的VHDL描述可以直接转换成多路器实现。图8-9给出了输出GE的改进的多路器实现。

这种转换将CASE变量（N1、N0和M1）映射为多开关的地址输入，用信号赋值语句定义多路器的数据输入。例如，“not M0”被连入数据输入D0，因为对case“000”的信号赋值语句定义为“not M0”。同样，D6连入逻辑‘1’，因为case“110”的信号赋值语句为常量1。对LE、E、G和L存在类似的实现。

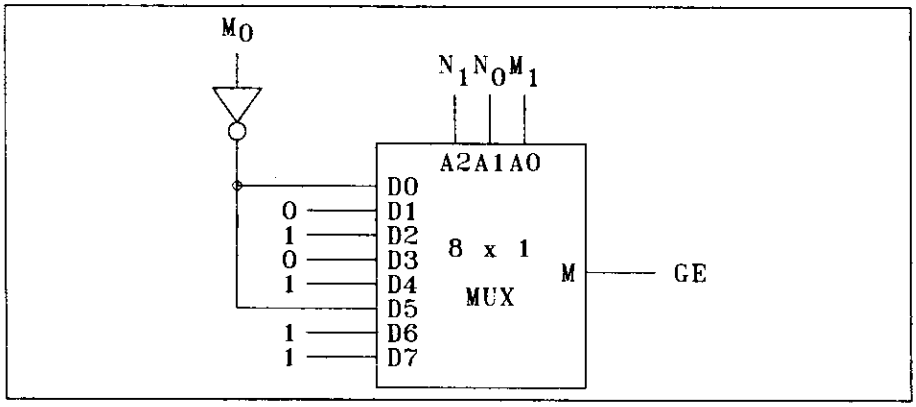


图8-9 直接将改进CASE模型转换成MUX实现

因为算法是通过VHDL描述来定义的，可以写一个程序来检查VHDL描述中的CASE型语句，以产生器件的优化的VHDL描述，从而自动实现该算法。

8.1.2 行为域的组合逻辑数据流模型设计

正如在图1-6的设计方法流程图中指出，设计者可以将算法级的行为模型转换为数据流行为模型。这种转换一般都带有优化的过程。设计者经常使用卡诺图的这种图形表示，来优化变量数目较少（3~8）的设计。使用编程方法，如Quine-McCluskey过程这样的一种文本表示，来优化具有中等数量变量（7~20）的设计。转换也使用智能分析来标明变量之间的关系，这样可以使结果简化。

1. 分解

设计分解通常用来将原先的器件分解成复杂性较小的组件。为了说明设计分解，考虑例8.1中的比较器。注意到5个输出并非相互独立。仔细研究5个输出，我们得出输出E、G和L可以从输出GE和LE得到如下：

```
E = GE and LE
G = GE and not LE
L = LE and not GE
```

因此，只需找到GE和LE的显式实现即可。

2. 数据流描述的综合

通过使用积和标准范式或和积标准范式，可以从真值表得到组合逻辑的数据流描述，该描述可以直接转换为门级结构域的描述。然而，这种实现非常低效。卡诺图（K-map）通常用来优化变量少于6个的情况。为了说明这个过程，我们使用卡诺图来优化设备COM的数据流实现。

图8-10给出了GE（Z_0）和LE（Z_1）优化后的两级积之和的实现。图8-11给出了优化后的

两级和之积的实现。很明显，和之积较积之和使用了较少的门和较少的门输入。因此，选择POS（和之积）形式进行进一步开发。

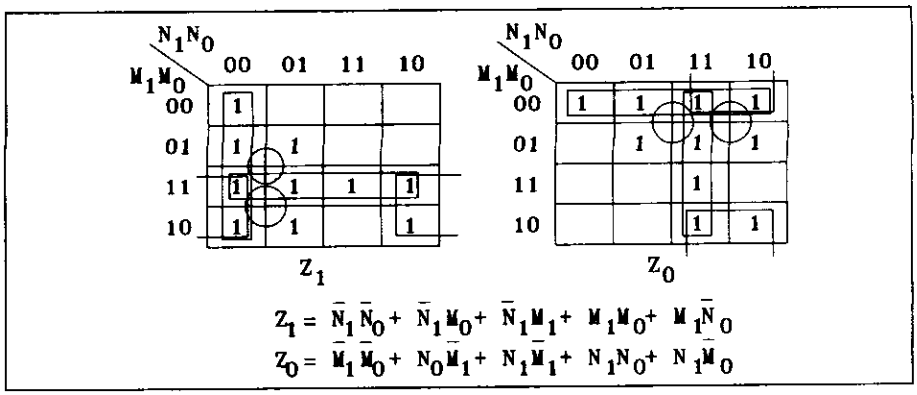


图8-10　GE（Z_0）和LE（Z_1）的优化两级积之和的实现

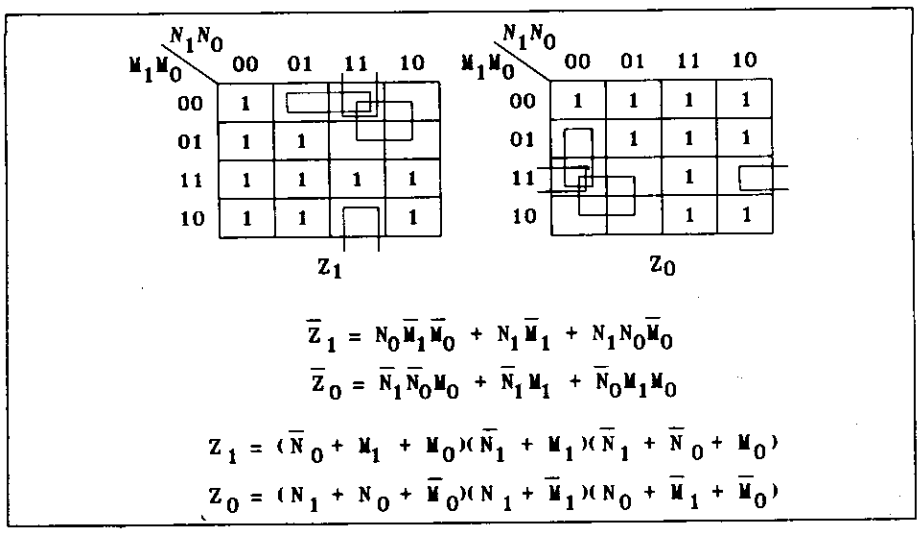


图8-11　GE（Z_0）和LE（Z_1）的优化两级和之积的实现

VHDL优化的数据流模型直接根据逻辑方程对每个二进制变量使用信号赋值语句，见图8-12。（见第10章对自动综合技术的进一步讨论。）

许多其他的数据流模型可以直接来自SOP（积之和）或POS（和之积）模型。例如，POS模型可以根据如下规则转化成一个NOR模型：

1) 通过在每个带括号的OR项前加入‘not’将OR项转化成NOR项。

2) 用or操作符替换and操作符，并在整个表达式之前插入not。

图8-13给出了使用这些转换规则转换图8-12中例子的结果。通过仿真可以验证其功能的完整性。

这里讨论的优化技术已经被大多数的工业综合工具实现了，我们将在第10章中讨论。

8.1.3　门级结构域组合逻辑电路的综合

这一节讨论组合逻辑的数据流描述到门级结构化描述的转换。这个过程很直接，而且可

以高度自动化。

```
-- Device to compare two binary inputs.
--
entity COM is
  generic (D:time);
  port (N1, N0, M1, M0: in BIT;
        GE, LE, E, G, L: out BIT);
end COM;
--
-- Optimum two-level product of sums data flow model.
--
architecture POSDF of COM is
  signal Z1,Z0: BIT;
begin
  Z1 <= (not N0 or M1 or M0) and (not N1 or M1) and
        (not N1 or not N0 or M0);
  Z0 <= (N1 or N0 or not M0) and (N1 or not M1) and
        (N0 or not M1 or not M0);
  LE <= Z1 after D;
  GE <= Z0 after D;
   E <= Z1 and Z0 after D;
   G <= Z0 and not Z1 after D;
   L <= Z1 and not Z0 after D;
end POSDF;
```

图8-12 器件COM的数据流模型的VHDL描述

```
-- Device to compare two binary inputs.
--
entity COM is
  generic (D:time);
  port (N1, N0, M1, M0: in BIT;
        GE, LE, E, G, L: out BIT);
end COM;
--
-- Optimum two-level product of sums data flow model.
-- Converted to two-level-nor data flow model.
--
architecture NORDF of COM is
begin
  process (N1, N0, M1, M0)
    variable Z1,Z0: BIT;
  begin
    Z1 := not(not(not N0 or M1 or M0) or not(not N1 or M1)
          or not(not N1 or not N0 or M0));
    Z0 := not(not(N1 or N0 or not M0) or not(N1 or not M1)
          or not(N0 or not M1 or not M0));
    LE <= Z1 after D;
    GE <= Z0 after D;
    E  <= not(not Z1 or not Z0) after D;
    G  <= not(not Z0 or Z1) after D;
    L  <= not(not Z1 or Z0) after D;
  end process;
end NORDF;
```

图8-13 器件COM的数据流模型的另一种VHDL描述

通常，一个给定的数据流模型可以转换成为几种不同的门级结构域模型。例如，基于积之和模型的数据流模型，可以以一种直接的方式转换为两级AND-OR电路、两级NAND-NAND电路、两级OR-NAND电路或任意数目的多级电路。同样，基于和之积模型的数据流模型，可以直接转换为两级OR-AND电路、两级NOR-NOR电路、两级AND-NOR电路（有时叫作AND-OR-INVERT）或任意数目的多级电路。

为了说明这个通用过程，我们将在图8-12中把比较器的和之积模型转换为几个门级结构模型。和之积模型可以转换为对偶电路或双电路。

根据图8-12中给出的VHDL描述，我们将每个and操作符替换成一个2输入与门，每个or操作符替换成一个2输入或门。结果见图8-14中所示。

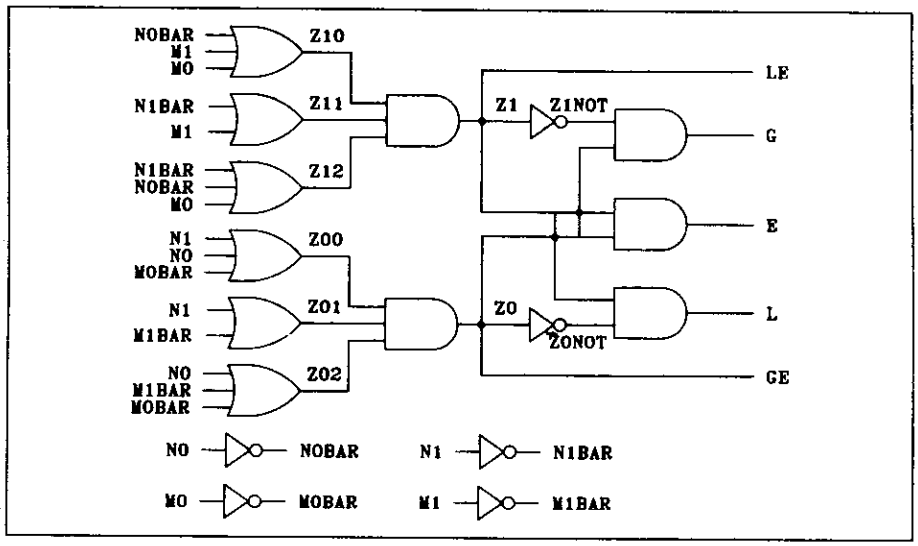


图8-14　COM模型到门级的直接转换，产生两级OR-AND实现

为验证图8-14中电路的功能，将创建一个VHDL结构模型进行仿真。图8-15的模型是用来仿真验证门级设计的。门实体的规格说明见图8-16。仿真也可以用来验证算法级模型的时序假设。注意：实体COM的时序信息转移到门实体（AND、OR等等）。在这一级上的仿真提供了更精确的时序信息。结构仿真的总计延时可以逆向标注到算法模型，以便在那一个抽象级别上提供精确的时序。这样，那个算法模型即可作为其他系统的组件，以其相对的简单性替代更加复杂的结构级模型。如果使用带有更加详细的模型精确信息的高级模型，则可以仿真大型的复杂系统；若使用需要更多仿真时间的更加细节的模型，则不可能对这些大系统进行仿真。

注意到两级OR-AND实现可以通过将每个门用一个大小合适的NOR门代替来实现，这样就可以有另一种实现。这作为本章的一道习题留给读者。

8.1.4　组合逻辑电路的设计活动小结

图8-17的框图小结了创建和之积模型的组合逻辑的设计活动，并且展示了在设计过程中VHDL模型的使用。创建积之和模型有相似的设计活动。见习题8.6。

根据器件的文字描述可以建立一张真值表，这是图形化表示。真值表用于算法级VHDL模型，在行为域中进行验证。上面已经介绍了两个这样的模型。ARRAY模型可以直接转换成

ROM实现，CASE模型可以直接转换成MUX实现。MUX实现可以通过优化步骤进行优化。

```
use work.all;
-- Device to compare two binary inputs.
entity COM is
  port (N1, N0, M1, M0: in BIT;
        GE, LE, E, G, L: out BIT);
end COM;
-- Two level OR-AND implementation derived from POS data
-- flow model
architecture TWO_LEVEL_OR_AND of COM is
  signal Z10,Z11,Z12,Z00,Z01,Z02: BIT;
  signal N0BAR,N1BAR,M0BAR,M1BAR: BIT;
  signal Z0,Z1,Z0NOT,Z1NOT: BIT;
  component NOT2G
    generic (D: TIME);
    port (I: in BIT; O: out BIT);
  end component;
  for all: NOT2G use entity NOT2(BEHAVIOR);
  component AND2G
    generic (D: TIME);
    port (I1, I2: in BIT; O: out BIT);
  end component;
  for all: AND2G use entity AND2(BEHAVIOR);
  component AND3G
    generic (D: TIME);
    port(I1,I2,I3: in BIT; O: out BIT);
  end component;
  for all: AND3G use entity AND3(BEHAVIOR);
  component OR2G
    generic (D: TIME);
    port(I1,I2: in BIT; O: out BIT);
  end component;
  for all: OR2G use entity OR2(BEHAVIOR);
  component OR3G
    generic (D: TIME);
    port (I1,I2,I3: in BIT; O: out BIT);
  end component;
  for all: OR3G use entity OR3(BEHAVIOR);
  component WIREG
    port (I: in BIT; O: out BIT);
    end component;
  for all: WIREG use entity WIRE(BEHAVIOR);
begin
 C1: NOT2G
     generic map (2 ns)
     port map (N0, N0BAR);
 C2: NOT2G
     generic map (2 ns)
     port map (N1, N1BAR);
```

图8-15 COM使用两级OR-AND形式的VHDL结构模型

算法级行为模型借助于卡诺图转换为数据流级行为模型。也可以使用其他的方法，如Quine-McCluskey。数据流模型在进一步处理之前通过仿真进行验证。

数据流模型以一种直接的方式转换成门级结构化模型。门级结构化模型进行仿真以验证其在该级的功能操作。从门级结构模型得到的时序信息可以逆向标注到算法和数据流模型，以提供更加精确的高层次时序仿真。

```
    C3: NOT2G
        generic map (2 ns)
        port map (M0, M0BAR);
    C4: NOT2G
        generic map (2 ns)
        port map (M1, M1BAR);
    C5: OR3G
        generic map (2 ns)
        port map (N0BAR, M1, M0, Z10);
    C6: OR2G
        generic map (2 ns)
        port map (N1BAR, M1, Z11);
    C7: OR3G
        generic map (2 ns)
        port map (N1BAR, N0BAR, M0, Z12);
    C8: AND3G
        generic map (2 ns)
        port map (Z10, Z11, Z12, Z1);
    C9: OR3G
        generic map (2 ns)
        port map (N1, N0, M0BAR, Z00);
    C10:OR2G
        generic map (2 ns)
        port map (N1, M1BAR, Z01);
    C11:OR3G
        generic map (2 ns)
        port map (N0, M1BAR, M0BAR, Z02);
    C12:AND3G
        generic map (2 ns)
        port map (Z00, Z01, Z02, Z0);
    C13:NOT2G
        generic map (2 ns)
        port map (Z1, Z1NOT);
    C14:NOT2G
        generic map (2 ns)
        port map (Z0, Z0NOT);
    C15:AND2G
        generic map (2 ns)
        port map (Z0, Z1, E);
    C16:AND2G
        generic map (2 ns)
        port map (Z0, Z1NOT, G);
    C17:AND2G
        generic map (2 ns)
        port map (Z1, Z0NOT, L);
    C18:WIREG
        port map (Z0, GE);
    C19: WIREG
        port map (Z1, LE);
end TWO_LEVEL_OR_AND;
```

图8-15 COM使用两级OR-AND形式的VHDL结构模型（续）

8.2 时序逻辑电路的设计

这一节讨论将硬件描述语言和状态表结合起来设计时序逻辑电路。设计方法包含了几个表示层次。

```
entity NOT2 is
  generic (D: TIME);
  port(I: in BIT; O: out BIT);
end NOT2;
--
architecture  BEHAVIOR of NOT2 is
begin
  O <= not I after D;
end BEHAVIOR;
--
entity AND2 is
  generic (D: TIME);
  port (I1,I2: in BIT; O: out BIT);
end AND2;
--
architecture BEHAVIOR of AND2 is
begin
  O <= I1 and I2 after D;
end BEHAVIOR;
--
entity AND3 is
  generic (D: TIME);
  port(I1,I2,I3: in BIT; O: out BIT);
end AND3;
--
architecture BEHAVIOR of AND3 is
begin
  O <= I1 and I2 and I3 after D;
end BEHAVIOR;
--
entity OR2 is
  generic (D: TIME);
  port(I1,I2: in BIT; O: out BIT);
end OR2;
--
architecture BEHAVIOR of OR2 is
begin
  O <= I1 or I2 after D;
end BEHAVIOR;
--
entity OR3 is
  generic (D: TIME);
  port (I1,I2,I3: in BIT; O: out BIT);
end OR3;
--
architecture BEHAVIOR of OR3 is
begin
  O <= I1 or I2 or I3 after D;
end BEHAVIOR;
```

图8-16 门实体的规格说明

我们从一个示例器件的文字描述开始。

例8-2设计一个串并转换器。图8-18给出了框图。输入CLK是系统中控制所有操作的时钟。复位信号R是同步的。如果在任何时钟周期末尾R=1，器件进入复位状态。输入A在输入D上串行数据到达之前的一个时钟周期有效。器件收集串行数据的4位，并将其并行输出在Z上，Z是一个4位向量。在并行数据处于输出Z上的那个时钟周期，信号DONE有效。输出Z和DONE

必须在一个完整的时钟周期内保持有效。DONE提示目标器件数据在Z上。图8-19给出了其时序，在并行数据在Z上的那个时钟周期，器件从连线A上收到另一个脉冲，指示新的数据将在下一个时钟周期到达D。如果这样，则必须准备接收数据。如果不是这样，器件在送完并行数据之后进入复位状态，等待新数据到来。

```
entity NOR2 is
  generic (D: TIME);
  port (I1,I2: in BIT; O: out BIT);
end NOR2;
--
architecture BEHAVIOR of NOR2 is
begin
  O <= I1 nor I2 after D;
end BEHAVIOR;
--
entity NOR3 is
  generic (D: TIME);
  port(I1,I2,I3: in BIT; O: out BIT);
end NOR3;
--
architecture BEHAVIOR of NOR3 is
begin
  O <= (I1 or I2) nor I3 after D;
end BEHAVIOR;
--
entity WIRE is
  port (I: in BIT; O: out BIT);
end WIRE;
--
architecture BEHAVIOR of WIRE is
begin
  O <= I;
end BEHAVIOR;
```

图8-16　门实体的规格说明（续）

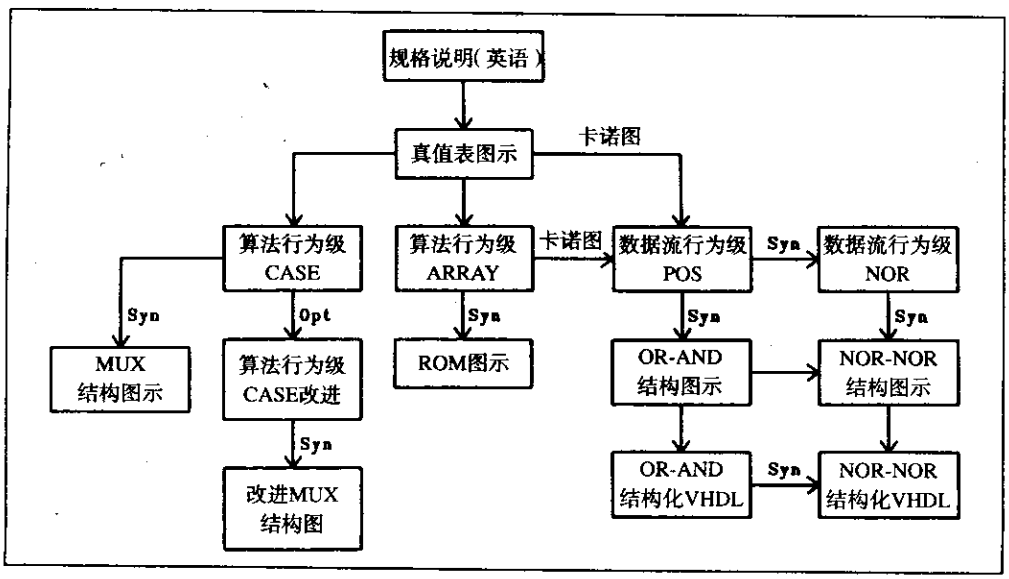


图8-17　POS组合逻辑的设计活动小结

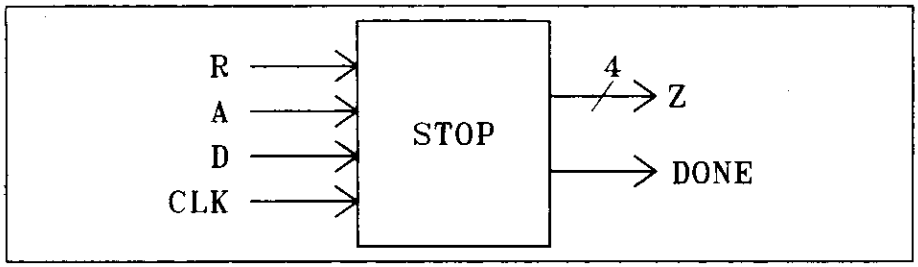


图8-18 串并转换器的框图

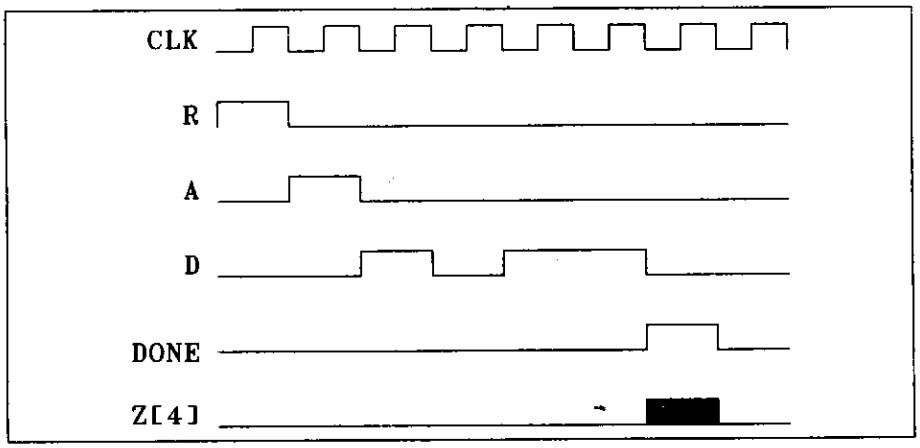


图8-19 串并转换器的时序图

8.2.1 Moore型或Mealy型的选择

设计有限状态机（FSM）的第一步是决定是设计Moore型还是设计Mealy型电路。读者应该已经很熟悉FSM的概念和Moore与Mealy器件的特性。因而这里只是简单回顾一下与建立状态表相关的一些概念。通过定义，Moore机输出只依赖于机器的当前状态，与输入信号值无关。Mealy机输出依赖于机器的状态和输入的值。从功能上说，根据定义可以建立Moore机或Mealy机，它们的主要区别在于输出的时序。要考虑三种实际效果：

1) 在Moore机中，输出在时钟的活动沿到达后的几个门延迟之后即得到输出值，并且在该时钟周期的剩余时间内保持不变，即使输入在该时钟周期内发生改变，输出值也保持不变。然而，因为输出与当前的输入无关，当前输入产生的任何效果将延迟到下一个时钟周期。Moore机的一个优点是将输入和输出分隔开。

2) 在Mealy机中，因为输出是输入的函数，如果输入改变，输出可以在一个时钟周期的中间发生改变。这使Mealy机比起Moore机来，对输入变化的响应要早一个时钟周期，但也使输出随着假输入的变化而变化。输入线上的噪声也会传到输出。

3) Moore机比对应的Mealy机可能需要更多的状态。

一个规格说明可能要求设计为Moore机或Mealy机或两者都需要。该选择常常取决于设计者的判断。

对于例8.2中的串并转换器，输出必须在最后一次输入之后的时钟周期内出现。因为上一次的输入已经得不到了，因而输出不可以依赖输入。而且，因为输出在整个时钟周期被指定

为常量，所以不可使用Mealy机，因而，必须设计一个Moore机。

8.2.2 状态表的建立

在选择了Moore机做目标器件后，就开始建立状态表。状态表的建立需要应用设计者的全部创造才能。对于大多数规格说明，总存在多个满足要求的状态表。尽管对于状态表的建立没有形式化的算法，但如果遵循结构化的方法，设计成功的可能性很大。

设计者通常用两种工具来简化状态表的建立过程：状态图和转换表。两种方法都将介绍。状态图提供了一种图形化的、易理解的对FSM的操作的描述，但限制于相对较小的器件。当问题太复杂而无法创建状态图时，应使用转换表。

8.2.3 创建状态图

因为串并转换器是相对较小的器件，使用状态图较为合适。建立状态图从一个容易用文字描述的状态开始。如果有复位状态，这往往是开始的好地方。我们建议为创建的每个状态写一个详细的文字说明，以使随后的设计便于参考，并为设计提供文档。过程总是迭代的。状态描述可以随设计过程而加以改进。然而，如果需要考虑新的情况，则希望尽可能修改先前所写的状态描述，而不是创建一个全新的状态。

对于串并转换器，我们从复位状态开始，用标号S0来表示这个状态，并使用如下的描述：

1) **状态S0，复位状态**。在任何时钟周期的末尾，如果输入R=1，无论其他输入值为多少，器件进入该状态，并一直处在该状态，直到线A变为逻辑1。当在状态S0时，DONE=0，表示线Z上的数据应被目标忽略，这意味着在输出Z上可以是任何信号。

下一步决定器件在S0状态下，对于输入上存在的各种情况应该如何响应。器件处在状态S0时，如果R=1，无论其他输入的值为多少，器件仍将处在状态S0。当R=0且A=0时，器件也处在状态S0。器件处在状态S0的完整条件是：

$$R+\bar{R}\,\bar{A}=R+\bar{A}$$

在某个时钟周期，如果R=0且A=1，器件必须准备好在下一个时钟周期在线D上接收数据。这意味着必须在这个时钟周期的末尾进入新的状态。如果用S1表示这个新的状态，这个新状态的描述如下：

2) **状态S1**。当R=0且A=1时，器件从状态S0进入状态S1。当器件处于状态S1时，第一个数据值出现在线D上，必须在本时钟周期的末尾保存，为下次的输出做准备。在状态S1，输出DONE=0，且Z是不确定的。

在研究状态描述时，画一张状态图，用图形表示器件操作是一种方便快捷的方法。状态图是有向图，其中的每个节点表示器件的一个状态。节点标号写在节点符号之内。对于Moore机，输出也可以出现在节点符号内部。如果器件从状态X转换到状态Y，则从节点X到节点Y之间有一个有向弧。每个弧上都有标号，以说明引起状态转换的输入条件。该条件与时钟的上升沿（或下降沿）同步。如果在写状态描述的同时画出状态图，就会对设计过程有帮助。串并转换器的部分状态图见图8-20。

在状态S0的节点处，记号0表示输出DONE=0，且输出Z未确定。在状态S1，输出也是0，从节点S0到节点S1的弧标记为$\bar{R}$ & A，指明在任何时钟周期的结尾，如果R=0且A=1，器件将从状态S0转换到状态S1。从节点S0到节点S0的弧标记为R+$\bar{A}$，指明在任何时钟周期的末尾，

如果R=1或A=0，器件仍保持状态S0。

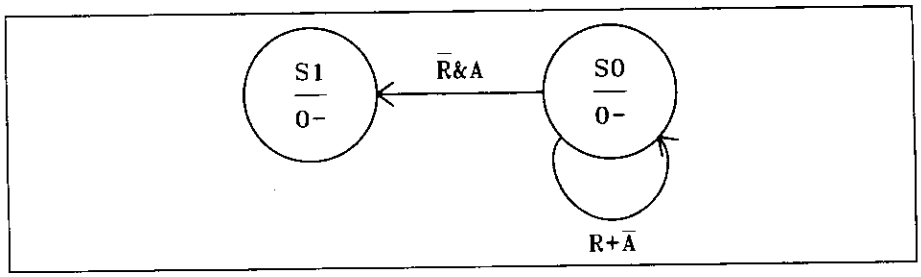


图8-20 串并转换器的第一个部分状态图

必须分析每个新状态，以确定对于所有可能的输入条件所需的状态转换。通过在现有节点之间添加弧或者根据需要添加新节点，可将这些状态转换加入状态图。该过程继续到所有节点都被分析完为止。这个过程有可能因状态图中状态的数量有限而终止。这即是有限状态机（FSM）名字的由来。也可能写出一个需要用无限状态数量来解决的问题的规格说明。判定何时会发生这种情况是个很难的问题，本书不作讨论。

对串并转换器的状态S1的分析如下。当器件处于状态S1时，如果R=1，电路复位至状态S0。这通过在状态图上加入从S1到S0的一道弧，标记为R来表明。如果R=0，器件进入新的状态，读入线D上4个输入值中的第2个值。新状态S2描述如下：

3) **状态S2**。当R=0时，状态S1转换至状态S2。当在状态S2时，第二个数据值在线D上，且必须在该时钟周期的末尾保存。在状态S2，输出DONE=0，而Z不确定。

因而，为状态S2在状态图上添加了一个新的节点。从S1到S2的弧标记为$\bar{R}$，表示状态转换。现在的状态图见图8-21。

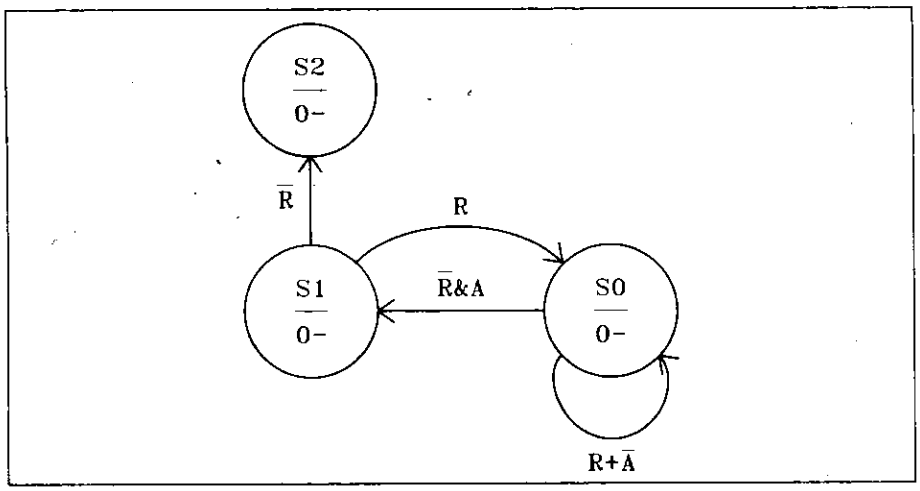


图8-21 串并转换的第2个部分状态图

图8-22给出了遵循规定的方法完成的器件STOP的最终状态图。其他状态的文字描述如下。

4) **状态S3**。当R=0时，从状态S2进入状态S3。当器件处于状态S3时，第三个输入出现在线D上，而且必须在该时钟周期结束时保存，输出DONE=0，Z不确定。

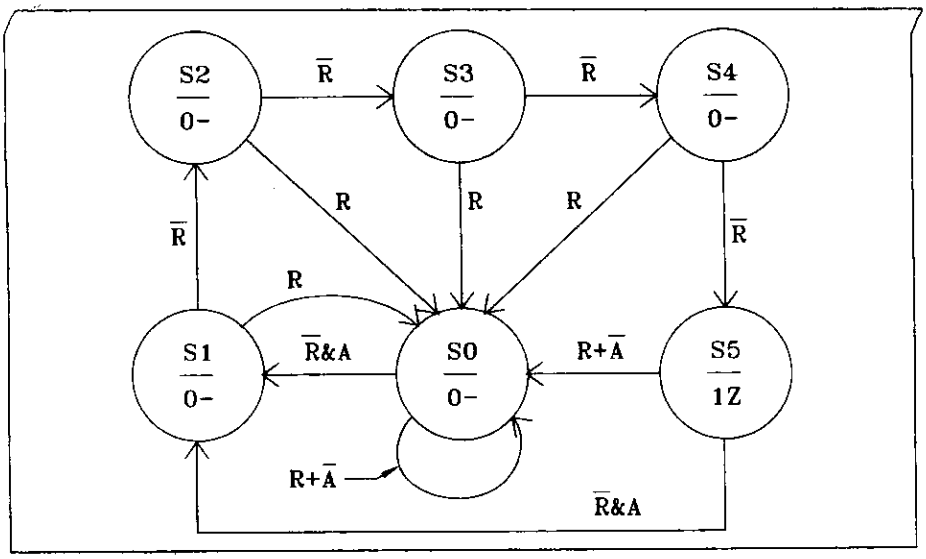


图8-22　串并转换器的最终状态图

5) **状态S4**。当R=0时，从状态S3进入状态S4。当器件处于状态S4时，第四个输入出现在线D上，而且必须在时钟周期结束时保存，输出DONE=0，Z不确定。

6) **状态S5**。当R=0时，从状态S4进入状态S5。在状态S5中，器件必须并行输出4个输入的数，并使DONE信号有效，输出DONE=1，输出Z等于四个输入位。

注意到当R=0且A=1时，引起从状态S5到状态S1的转换。输入A=1意味着器件必须准备好下一个时钟周期在线D上串行接受下一组4位。这时候，我们注意到可以再次使用状态S1，而不用创建一个新的状态。这样就封闭了状态图并完成了设计。既然再次使用了状态S1，就必须修改状态1原来的文字描述以反映出新的用途。当新的用法与原用法完全兼容时，可以再次使用那个状态。修改后状态S1的描述如下：

7) **修改后的状态S1**。当R=0且A=1时，从状态S0和S5进入状态S1。当器件处于状态S1时，第一个数据值出现在线D上，必须为下次的输出在本时钟周期的末尾保存该数据值。在状态S1，输出DONE=0，且Z是不确定的。

也应该修改状态S0以包括从S5状态的转化，这留给读者作为练习。

8.2.4　转换表

状态图是帮助理解状态之间关系的一种方法。提供输入序列，很容易找到状态转换的序列。然而，对于有许多状态的大电路，状态图会变得凌乱，极难画出可用的形式。这时，普遍使用一种称作转换表的文本描述方式。创建转换表的方法与状态图相同。转换列在一张表中，它不是采用在状态图中画箭头以表示状态转换。图8-23给出了串并转换器的状态转换表。

互斥原则。互斥原则可以在设计过程中用于帮助检查状态图的错误。离开任何节点的弧上的逻辑表达式必须成对互斥。即没有在离开同一节点不同弧上的两个表达式同时为真的情况。如果两个这样的表达式同时都是逻辑1，机器将试图进入两个不同的次状态——很明显，这不可能发生。

互斥测试的是离开同一节点的不同弧上的两个表达式的逻辑AND必须为逻辑0，例如，对

于节点S0：

当前状态	转换表达式	嵌套状态	数据传输	输出
S0	$R+\overline{A}$	S0	无	DONE=0, Z 不确定
S0	$\overline{R}$ & A	S1		
S1	$\overline{R}$	S2	存储位1	DONE=0, Z 不确定
S1	R	S0		
S2	$\overline{R}$	S3	存储位2	DONE=0, Z 不确定
S2	R	S0		
S3	$\overline{R}$	S4	存储位3	DONE=0, Z 不确定
S3	R	S0		
S4	$\overline{R}$	S5	存储位4	DONE=0, Z 不确定
S4	R	S0		
S5	$\overline{R}$ & A	S1	无	DONE=1, Z = 并行数据输出
S5	$R+\overline{A}$	S0		

图8-23 串并转换器的转换表

$$(\overline{R}A)(R+A) = (\overline{R}AR) + (\overline{R}A\overline{A}) = 0 + 0 = 0$$

因此，两个逻辑表达式是互斥的。

在创建过程中用这个原则来检查状态图。很明显，转换表也必须满足这个原则。某一个状态的不同表达式必须两两互斥。

8.2.5 创建状态机的VHDL模型

状态图和转换表都可以用来创建VHDL模型。这个模型可以用来验证电路的功能，因而，能够在进行更低一级的设计之前发现任何逻辑错误。

模型中，假设系统被分为数据单元、控制单元和输出单元，见图8-24。相同的时钟同步着控制单元和数据单元的操作。在Moore机中，输出是控制状态和数据单元中存储的值的函数。对于Mealy机，输出也可以依赖于输入，如虚线所示。

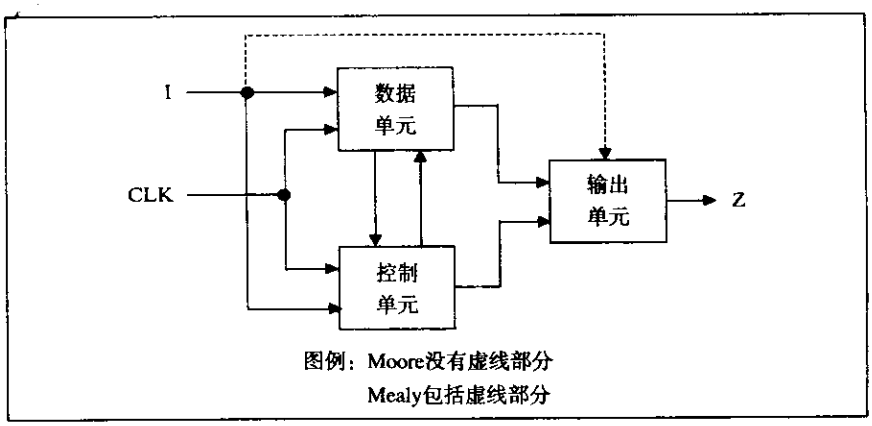


图8-24 状态机模型的框图

从状态图或转换表容易构建出图8-25中所示的VHDL模型。实体声明STOP声明了输入和输出。输入R、A、D和CLK是BIT类型的；输出Z是下标从3到0的4位向量。输出DONE是BIT类型。

在构架（FSM_RTL）中，枚举类型STATE_TYPE定义了状态的集合。类型列表中的元素是直接从状态图或转换表中取出的状态名。信号STATE保存了器件的当前状态值。

```
-- Serial to Parallel Converter
entity STOP is
  port (R, A, D, CLK: in BIT;
        Z: out BIT_VECTOR(3 downto 0);
        DONE: out BIT);
end STOP;

-- State Machine Description
-- for Serial to Parallel Converter (STOP)
architecture FSM_RTL of STOP is
  type STATE_TYPE is (S0, S1, S2, S3, S4, S5);
  signal STATE: STATE_TYPE;
  signal SHIFT_REG: BIT_VECTOR (3 downto 0);
begin
-- Process to update state at end of each clock period.
STATE: process (CLK)
  begin
    if CLK='1' then
      case STATE is
        when S0 =>
          -- Data Section
          -- Control Section
            if R='1' or A='0' then
              STATE <= S0;
            elsif R='0' and A='1' then
              STATE <= S1;
            end if;
        when S1 =>
          -- Data Section
          -- Shift in the first bit
            SHIFT_REG <= D & SHIFT_REG(3 downto 1);
          -- Control Section
            if R='0' then
              STATE <= S2;
            elsif R='1' then
              STATE <= S0;
            end if;
        when S2 =>
          -- Data Section
          -- Shift in the second bit
            SHIFT_REG <= D & SHIFT_REG(3 downto 1);
          -- Control Section
             if R='0' then
               STATE <= S3;
             elsif R='1' then
               STATE <= S0;
             end if;
```

图8-25 串并转换器的VHDL模型

```
-- Continuation of architecture FSM_RTL of STOP
--
        when S3 =>
          -- Data Section
          -- Shift in the third bit
            SHIFT_REG <= D & SHIFT_REG(3 downto 1);
          -- Control Section
            if R='0' then
              STATE <= S4;
            elsif R='1' then
              STATE <= S0;
            end if;
        when S4 =>
          -- Data Section
          -- Shift in the fourth bit
            SHIFT_REG <= D & SHIFT_REG(3 downto 1);
          -- Control Section
            if R='0' then
              STATE <= S5;
            elsif R='1' then
              STATE <= S0;
            end if;
        when S5 =>
          -- Data Section
          -- Control Section
            if R='0' and A='1' then
              STATE <= S1;
            elsif R='1' or A='0' then
              STATE <= S0;
            end if;
      end case;
    end if;
  end process STATE;
--
-- Output process
--
  OUTPUT: process (STATE)
  begin
    case STATE is
      when S0 to S4 =>
        DONE <= '0';
      when S5 =>
        DONE <= '1';
        Z <= SHIFT_REG;
    end case;
  end process OUTPUT;
end FSM_RTL;
```

图8-25 串并转换器的VHDL模型（续）

这时必须确定如何保存串行出现在线D上的位。可以用许多不同的方法实现，各种方法将导致不同的硬件配置，其耗费也不同。如何对位的保存进行建模对于综合后电路的复杂性或耗费有明显的影响。在这个例子中，用移位寄存器来保存位。移位寄存器被声明为BIT_VECTOR类型的信号，它与最终输出Z有相同的范围。

构架包括两个进程，进程STATE在每个时钟周期的结尾处当CLK从逻辑0跃变为逻辑1时，更新系统的状态。因而，输入CLK必须在进程的活动列表之中。“if CLK= ‘1’ then”语句检查CLK的上升沿。如果检测到上升沿，状态被更新。新状态在依赖于当前状态的CASE语句中

进行计算。CASE语句的每个选项对应当前状态的一个可能值。例如，CASE语句中选项S0对应状态图中的状态S0。状态处理包括两个部分：数据部分和控制部分。每个状态的文字语句定义了数据部分，状态图中离开状态的箭头定义控制部分。对于状态S0，没有数据部分的操作。状态图隐含在条件R=1或A=0时，下一个状态应该为S0。VHDL模型通过语句“if R=1 or A=0 then STATE<=S0”完成这个动作。状态图同样隐含如果当前状态是S0，条件为R=0且A=1，下一个状态将为S1。VHDL模型使用语句“elsif R=‘0’ and A=‘1’ then STATE<=S1”以引起想要的状态转换。

CASE语句的剩余选项定义了其他的状态转换，唯一的附加效果是在某些状态如S1中，数据传输必须与状态转换同时进行。例如，语句“SHIFT_REG<=D & SHIFT_REG （3 downto 1）”加入到状态S1的数据部分以定义对位的存储。‘&’符号表示并置，该语句实现了寄存器SHIFT_REG的右移。

可以从图8-23的转换表中得到同样的信息。从每个状态出发的转换的集合定义了那个状态的case语句的控制部分。状态的文字描述定义了数据部分。读者应检查转换表以验证可以利用该表中的信息建立VHDL的case语句。

进程OUTPUT定义了器件输出信号线上的逻辑值。因为采用的是Moore机，活动列表中的唯一信号是STATE。对于Mealy机，活动列表应包括状态信号和输入信号。作为结果，Moore机的输出只有在状态改变时才改变。反之，Mealy机的输出在状态或输入改变时都改变。因为是Moore机，输出可以使用“case STATE”结构来确定。输出来自状态图（或转换表）。例如，从状态图可得，在状态S5输出的DONE是逻辑1。VHDL程序使用信号赋值语句“DONE<=1”使输出改变。而且，输出Z只在状态S5中定义。在VHDL的case语句中，Z只在状态S5才被赋值。由于VHDL信号的语义，Z在其他状态时值保持不变，因此Z保持其值直到机器再次进入状态S5。根据问题的规格说明，Z在其他时候是不确定的。仿真正确地满足了这些规格说明。然而，在任何自动化综合过程中，低层次描述的仿真必须与高层次仿真的行为匹配。因而，所有低层次仿真必须在对Z赋值之间保存其值。结果，综合器也许会创建一个锁存器来保存Z。第10章对综合的效果有详细的讨论。

为了利用Z在除了S5之外的状态不确定这个事实，应使用多值逻辑。见第5.3.1节中对IEEE 9值逻辑系统的描述，其中包括对不确定状态的处理机制。

在状态表的VHDL模型中使用聚合

现在讲解一种从状态机的状态表中直接创建VHDL模型的方法。该方法由Ken Bakalar在Compass Design Systems中提出，使用记录来表示状态转换，使用聚合直接表示典型格式的状态表。

方法是：

1) 用枚举类型来表示状态。

2) 将状态表编码为实现次态和输出函数的数组聚合。

3) 根据要求存取数组聚合，对有限状态机的操作进行建模。

为了说明这个过程，我们将使用在第1章讲述的一个状态机的例子，系统检测在其输入之上是否出现两个或多个相连的1或0。图1-4给出了状态表。注意它隐含为Mealy机。

状态表的一个算法模型见图8-26。注意枚举类型STATE有三个值：S0、S1和S2，且机器的状态保存在信号FSM_STATE之中，其类型是STATE。定义了记录类型TRANSITION，它的

两个字段是OUTPUT和NEXT_STATE。下面是TRANSITION记录的两个二维数组的类型声明（类型TRANSITION_MATRIX）。注意到数组的两个下标的类型是STATE和BIT。状态表通过定义TRANSITION_MATRIX类型的常量，并用聚合将其初始化来实现。注意到第一个类型STATE的下标从数组中选择一行，第二个类型为BIT的下标从该行之中选取一个元素。每个数组元素是TRANSITION类型的记录聚合。

FSM进行的活动在一个进程内部发生。信号R、X、CLK和FSM_STATE触发那个进程。进程内部有三个活动：复位、时钟事件和输出函数。复位覆盖了时钟事件。时钟事件和输出函数活动存取数组并选择合适的记录字段。输出函数在机器输入和状态发生变化时被触发。

```
entity TWO_CONSECUTIVE is
  port(CLK,R,X: in BIT; Z: out BIT);
end TWO_CONSECUTIVE;
--
architecture FSM of TWO_CONSECUTIVE is
  type STATE is (S0,S1,S2);
  signal FSM_STATE: STATE := S0;
  type TRANSITION is record
    OUTPUT: BIT;
    NEXT_STATE: STATE;
  end record;
  type TRANSITION_MATRIX is array(STATE,BIT) of TRANSITION;
  constant STATE_TRANS: TRANSITION_MATRIX :=
    (S0 => ('0' => ('0',S1), '1' => ('0',S2)),
     S1 => ('0' => ('1',S1), '1' => ('0',S2)),
     S2 => ('0' => ('0',S1), '1' => ('1',S2)));
begin
  process(R,X,CLK,FSM_STATE)
  begin
    if R = '0' then -- Reset
      FSM_STATE <= S0;
    elsif CLK'EVENT and CLK ='1' then -- Clock event
      FSM_STATE <= STATE_TRANS(FSM_STATE,X).NEXT_STATE;
    end if;
    if FSM_STATE'EVENT or X'EVENT then -- Output Function
      Z <= STATE_TRANS(FSM_STATE,X).OUTPUT;
    end if;
  end process;
end FSM;
```

图8-26 使用记录和聚合的状态表的算法模型

使用聚合表示表格是一种实用高效的方法。数据组织成表格形式，便可以根据图形表格输入，自动产生算法模型。

8.2.6 VHDL状态机模型的综合

有限状态机的模型可以用一种相当直接的方式进行综合。模型具有图8.24所示的通用形式。我们从讨论控制部分的综合开始。

通过将每个状态赋值给一个控制触发器，综合程序通过检查下一个状态进程中case语句的每个when子句中的“if… then”语句可以确定一个完全的控制单元设计。图8-27所示的控制器可以用这样的综合程序得到。

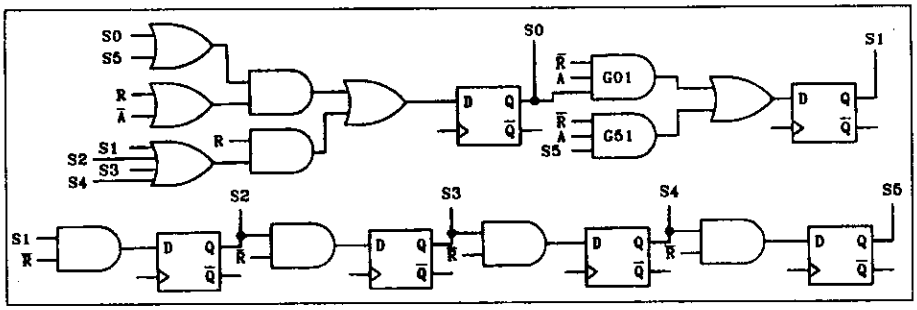


图8-27 从串并转换器的VHDL描述得到的综合控制电路

输入D到每个触发器的逻辑方程通过扫描VHDL case语句并创建要转换到该状态的全部条件的列表来实现。图8-28给出了串并转换器的列表。

到达状态	来自状态	条件	到达状态	来自状态	条件
S0	S0	R + $\overline{A}$	S1	S0	$\overline{R}$ & A
S0	S1	R	S1	S5	$\overline{R}$ & A
S0	S2	R	S2	S1	$\overline{R}$
S0	S3	R	S3	S2	$\overline{R}$
S0	S4	R	S4	S3	$\overline{R}$
S0	S5	R + $\overline{A}$	S5	S4	$\overline{R}$

图8-28 串并转换器的控制部分的综合列表

图8-27的控制电路直接来自图8-28中的数据。例如，考虑输入D到触发器S1的驱动逻辑。三输入与门G01导致S0到S1的转换条件$\overline{R}$A列在图8-28中右边列的顶端。同样，在条件$\overline{R}$A下需要从S_5转换到S_1导致三输入与门G51。选择一个或门来计算这两个函数的逻辑或，且或门的输出连接到触发器S1的输入D上。所有控制触发器的输出，除了可以用在逻辑表达式中作为控制触发器的D输入外，还可以作为数据单元和输出单元的输入。

这种类型的控制单元设计（每个状态一个触发器）叫做“单热”控制单元，因为每次只有一个触发器被置位。在第10章，会看到这对于FPGA的综合是一个有利方法。

数据单元可以根据下一个状态进程中的case语句以类似的过程进行综合。首先，综合程序扫描case语句，列表中包括所有的数据传输及需要的条件。图8-29给出了串并转换器的列表。

数据传输	状态	条件
SHIFT_REG <= D & SHIFT_REG(3downto 1)	S1	1
SHIFT_REG <= D & SHIFT_REG(3downto 1)	S2	1
SHIFT_REG <= D & SHIFT_REG(3downto 1)	S3	1
SHIFT_REG <= D & SHIFT_REG(3downto 1)	S4	1

图8-29 串并转换器数据单元的综合列表

每个数据传输的控制信号通过将每个状态和其对应条件进行逻辑“与”运算而得到（在这里，所有的条件都是1），然后，将结果进行逻辑或运算。根据该表，下述控制表达式能够

从移位操作中得到：

$$SHIFT = (S1)(1) + (S2)(1) + (S3)(1) + (S4)(1) = S1 + S2 + S3 + S4$$

图8-30给出了数据单元的设计。

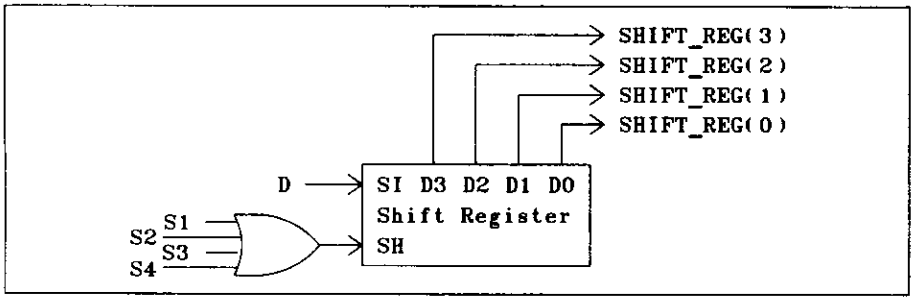


图8-30 串并转换器的数据单元

VHDL代码不包括如何设计移位寄存器的任何信息。移位寄存器的设计是另外一个工程。通常它已经被设计好，而综合程序只需简单地将其从库中取出。

输出单元的设计可以通过扫描输出进程从case语句中创建一个逻辑方程。对于串并转换器，DONE的逻辑方程是：

$$DONE = S5$$

Z的设计较为复杂一些，因为原先的说明只要求Z在状态S5有一个确定的值。然而，VHDL信号的语义指明其值将无限期地保持在Z线上，直到被改变为止。正如前面所提到的，综合工具将Z实现为一个锁存器（见第10章），该锁存器在状态S5用移位寄存器的值更新。这样，行为仿真和综合电路仿真将会一致。

8.3 微程序控制单元的设计

本节给出微程序控制单元的系统设计过程。硬件描述语言与其他设计方法相结合，提供了一个多层次设计的整体方法。

8.3.1 控制器和器件的接口

图8-31给出了典型器件的控制单元和数据单元的接口。控制单元产生s级控制信号，标记为S0到S(s-1)，控制数据单元中数据的传输。数据单元送出c个条件信号给控制单元，标记为C0到C(c-1)。控制单元根据这些条件选择合适的控制步骤序列来执行。选择的序列确定了接下来的数据传输。控制器和数据单元都必须由同一时钟信号定时。时钟可以是多相的，使得在一个主时钟周期之内可以执行几个相连的步骤。

8.3.2 硬连线和微程序控制单元的比较

硬连线控制单元是包含触发器和逻辑门的定制电路。控制器确定了将被执行的一系列控制节步骤。图8-32是控制单元的一部分，说明如果条件A为真时，如何实现从第5步（S5=1）到第6步（S6=1）的条件控制转移。信号A代表从控制单元出来的控制信号和其他控制触发器

所产生的信号的组合。S5和S6信号直接被数据单元用作控制信号。在硬连线控制单元中，每个控制信号都是利用控制单元信号和数据单元的状态信号来个别地导出。

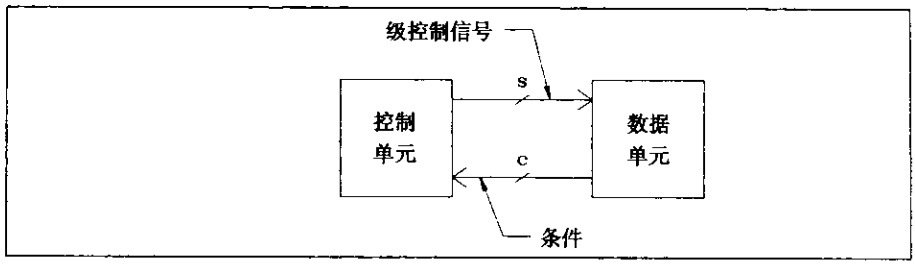


图8-31　控制单元和数据单元的接口

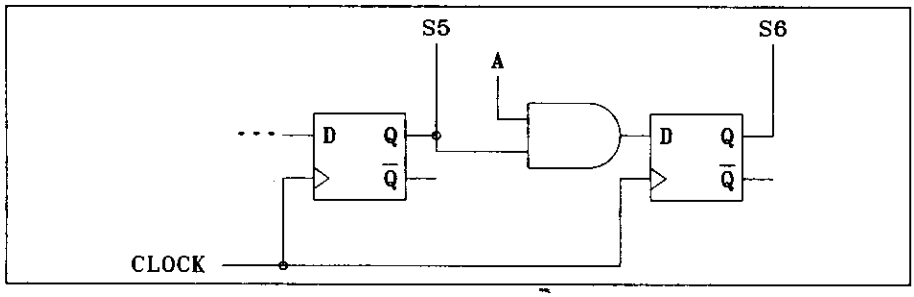


图8-32　硬连线控制单元的典型控制序列

在微程序控制单元中，所有控制信号的值从ROM中一个适当的地址中取出。ROM中每个地址中的内容叫做一个控制字（control word）。在每个时钟周期，适当级别的控制信号从ROM中读出，而不是由逻辑电路产生。理论上这是一个很简单的概念。图8-33给出了微程序控制单元的一般框图，包括一个a位的地址寄存器（MAR），w位指令寄存器（MIR），容量为$2^a w$位字长的ROM，以及地址产生逻辑（AGL）。在任何特定的时间内，地址产生逻辑计算出指向下一个要读的控制信号的地址。计算下一个地址所需的信息同样也从ROM中读出。地址产生逻辑同样使用从数据单元来的状态信号以帮助确定下一地址。在一个时间步内，ROM地址的产生比较困难。地址产生逻辑随设计不同而差别很大，这是控制器约束的主要来源。

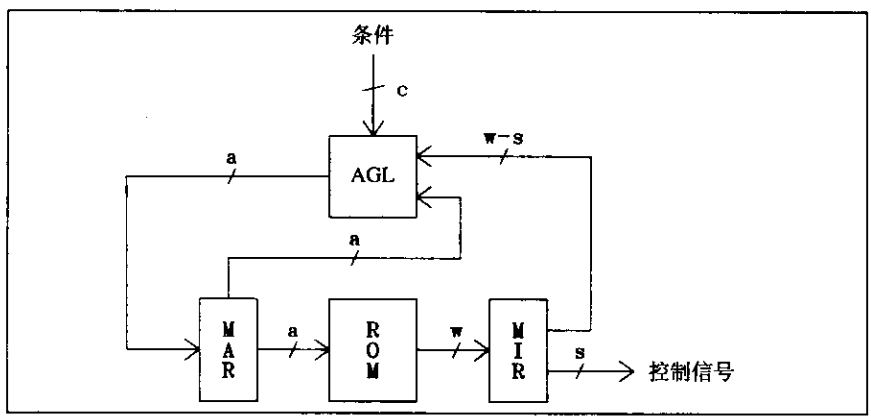


图8-33　微程序控制单元的框图

控制单元和数据单元的时序信号必须严格协调。如图8-33所示，从控制ROM中读出的一组控制信号分两步进行。ROM地址必须载入MAR，控制信号值载入MIR。这两步操作不可以并行进行，因为MIR中的数据是用来计算下一地址的。控制单元必须连续进行两次数据传输，以准备在数据单元中进行一组数据传输。图8-34给出了怎样将控制单元操作和数据单元操作重叠。在系统时钟的上升沿，控制信号的新值载入MIR。在系统时钟下降沿，下一个地址载入MAR。如果数据单元电路的延时足够小，可以保证控制信号有足够时间穿过数据单元的逻辑，并满足数据寄存器的所有建立时间，当然也有可能在数据单元内进行数据传输。在系统时钟的下一个上升沿，第二组数据传输在数据单元中完成，同时新的控制信号值载入MIR。通过这种安排，在每个主时钟周期内，数据单元中将进行两套相连的操作。

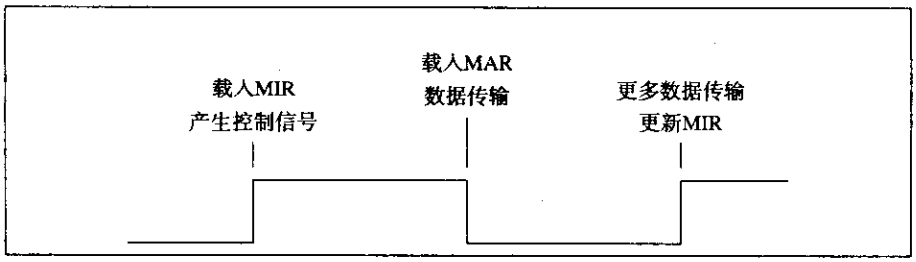


图8-34 微程序控制单元的时序

可能有人会认为，如果ROM输出驱动电路能提供足够的电力适应数据单元和地址产生逻辑（AGL）产生的负载时，则并不需要MIR。乍一看，似乎可以得出时钟周期可以快上将近两倍的结论。然而，实际情况并非如此，时钟周期必须满足下述限制：

$$\text{PER\_without\_MIR} > \text{MAR\_delay} + \text{ROM\_delay} + \text{AGL\_delay} + \text{MAR\_setup\_time}$$

在原先的设计中，包括MIR：

$$\frac{\text{PER}}{2} > \text{MAR\_delay} + \text{ROM\_delay} + \text{MIR\_setup\_time}$$

和

$$\frac{\text{PER}}{2} > \text{MIR\_delay} + \text{AGL\_delay} + \text{MAR\_setup\_time}$$

因为条件是在低时钟和高时钟之间划分的，因而，整个时钟周期必须满足：

$$\text{(PER\_with\_MIR)} > \text{MAR\_delay} + \text{ROM\_delay} + \text{MIR\_setup\_time} + \text{MIR\_delay} + \text{AGL\_delay} + \text{MAR\_setup\_time}$$

很明显，两个时钟周期之间的差是：

$$\text{(PER\_with\_MIR)} - \text{(PER\_without\_MIR)} = \text{MIR\_setup\_time} + \text{MIR\_delay}$$

这显然和总时钟周期的1/2不相等。实际上，去掉MIR，时钟周期可能会缩短10%到20%。

然而，因为许多ROM芯片为低功率驱动，至多能够应付一般的负载，因而必须要有输出寄存器缓冲，这样时钟周期只增加10%到20%。而且，如果去掉MIR，数据单元延迟只留有小的余地，这是因为ROM延迟插进了控制信号路径。如果有MIR，控制信号流从MIR直接流入数据单元，而不需经过ROM。没有MIR，控制信号路径的延时为：

CONTROL_SIGNAL_path_delay=MAR_delay+ROM_delay+DATA_UNIT_delay

有MIR，控制信号路径延时是：

CONTROL_SIGNAL_path_delay=MIR_delay+DATA_UNIT_delay

假设MAR和MIR有相近的延时，主要的差别是ROM延时，ROM延迟和门或寄存器延时相比一般要大得多。

因为去掉MIR的增益不大，却引入了几个问题，所以我们的介绍之中仍加入MIR。微程序控制单元的速度慢于硬连线控制单元的主要原因是ROM的存在。假设数据单元中控制路径的延时与地址产生逻辑的延时相当，总时钟周期必须被伸长大约ROM延时那么多的数量。如果去除MAR，可进行类似的分析，见习题8.35。

8.3.3 基本微程序控制单元

本节描述一个初级的微程序控制单元，叫做基本微程序控制单元（BMCU）。我们将使用一个简单的称之为两路分支的地址产生过程。每个控制字（CW）包括64位，如图8-35中所示。因而w=64。

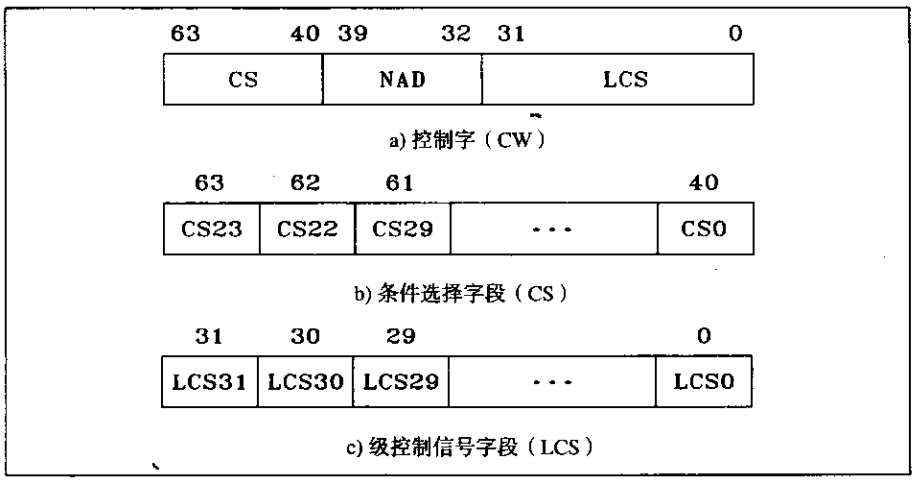


图8-35　BMCU中控制字的组织

条件选择（CS）字段（CW（63:40））包含用来选择输入状态信号的信息，以计算下一条指令地址（见图8-35b）。在BMCU中，我们使用“单热”编码来选择状态信号。当选择了条件Ci后，位CSi将是逻辑‘1’，CS字段的所有其他位都将是逻辑‘0’。这是因为CS字段在控制字中的位置，CSi选择位在位置CW（40+i），其中I的范围是从23到0。在本设计中，最多的状态信号数为24(c=24)。如果Ci=1且Ci被选择（CSi=1），则下一条地址（NAD）字段（CW（39:32））是要取的下一条控制字的地址。如果Ci被选择且Ci=0，当前存储器地址（MAR的内容）递增，以计算下一条地址。

分级控制信号（LCS）字段（CW（31:0））分别包含控制信号LCS31到LCS0的值，如图8-35c所示。因而，在本设计中，分级控制信号的最大数量是32（s=32）。控制信号LCSi的值与CW(i)的值相等。因为i的范围是从31到0。因为ROM之中的所有地址都必须直接存取于NAD字段，而NAD字段有8位（a=8），则最大ROM为256个字（2^8）。

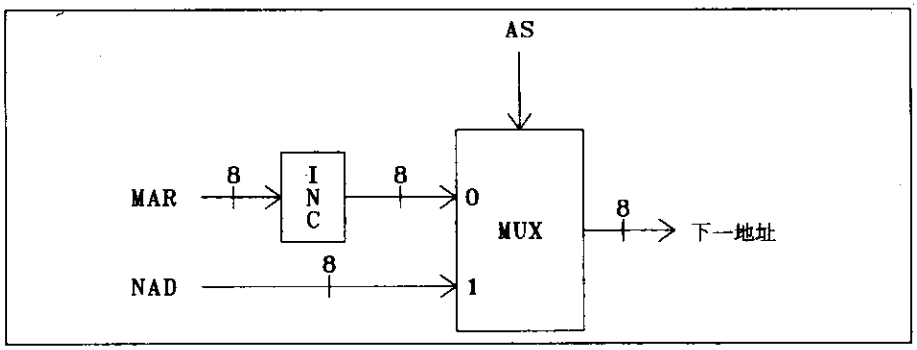


图8-36 BMCU的地址产生逻辑

地址产生逻辑可以用图8-36所示的向量MUX来设计。如果选择的状态信号为TRUE，地址选择信号（AS）必须为逻辑‘1’。AS控制信号的逻辑方程是：

$$AS = (CS23)(C23) + (CS22)(C22) + ... + (CS0)(C0)$$

或

$$AS = (CW63)(C23) + (CW62)(C22) + ... + (CS(40))(C0)$$

因而，如果选择条件为真，NAD的内容作为下一条地址；如果选择条件为假，MAR+1作为下一条地址。

8.3.4 BMCU的算法级模型

现在可以开发出BMCU的用于高级仿真的算法级模型。根据前面的描述，很容易得到图8-37中的进程模型图。注意已经加入了RESET信号以初始化BMCU。

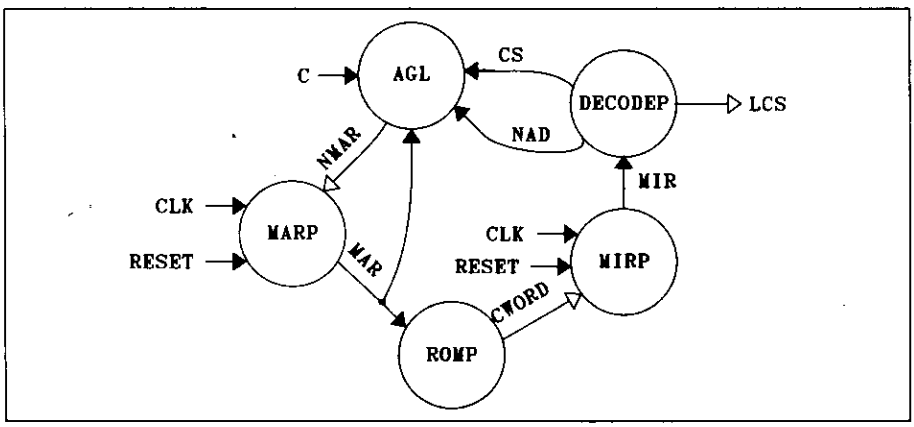


图8-37 BMCU的进程模型图

进程MARP代表MAR。当RESET活动时，MAR将被重置为地址X“0000”。在系统时钟（CLK）的下降沿，新地址（NMAR）将被载入MAR。进程ROMP代表控制ROM。当新地址载入MAR之后，新的一套控制信号（CWORD）将从控制ROM中读出。在系统时钟的上升沿，CWORD将由进程MIRP传输进MIR。当RESET活动时，MIR会被初始化为零向量。进程

DECODEP简单地将CWORD分成其组成部分（CS、NAD和LCS）。进程AGL计算下一条地址，从那里根据上一节介绍的规则，读入下一个控制字。如果选择的条件位为TRUE，则下一条地址为NAD。否则为MAR+1。信号NMAR代表计算出的地址。在系统时钟的下降沿，新地址载入MAR，整个序列又重新开始。

根据进程模型图，用第5章中介绍的方法可以很容易得到图8-38a和图8-38b中所示的VHDL算法级模型。注意，加入了每个组件的类属延迟以表示高级时序信息。

```
-- Basic MicroCoded Control Unit
entity BMCU is
  generic (AGL_DELAY, MAR_DELAY, ROM_DELAY, MIR_DELAY: TIME);
  port (C: in BIT_VECTOR (23 downto 0)--Cond. from data unit.
          :=B"0000_0000_0000_0000_0000_0000";
        CLK,RESET: in BIT:='0';  -- System clock and reset.
        LCS: out BIT_VECTOR (31 downto 0));--Level Control
end BMCU;                              --Signals to data unit.
--
-- Algorithmic level VHDL description of BMCU.
use work.BMCU_FUNCTIONS.all;
architecture ALGORITHMIC of BMCU is
  -- MAR is the Memory Address Register for the ROM.
  -- MIR is the Memory Instruction Register for the ROM.
  signal MAR,NMAR: BIT_VECTOR (7 downto 0);-- NMAR is nxt val
  signal CWORD: BIT_VECTOR (63 downto 0);-- Control word
  signal MIR: BIT_VECTOR (63 downto 0);
  signal NAD: BIT_VECTOR (7 downto 0);-- Branch address
  signal CS: BIT_VECTOR (23 downto 0);-- Condition Select
begin
  -- Memory Address Register process.
  MARP: process (CLK, RESET)
  begin
    if RESET='1' then
      MAR <= B"0000_0000" after MAR_DELAY;
    elsif CLK'event and CLK='0' then
      MAR <= NMAR after MAR_DELAY;
    end if;
  end process MARP;
  -- Memory Instruction Register Process
  MIRP: process (CLK, RESET)
  begin
    if RESET='1' then MIR <=
B"00000000_00000000_00000000_00000000_00000000_00000000_00000000_00000000"
      after MIR_DELAY;
    elsif CLK'event and CLK='1' then
      MIR <= CWORD after MIR_DELAY;
    end if;
  end process MIRP;
-- Model code continued in next figure.
```

图8-38a BMCU的算法级VHDL描述

程序包BMCU_FUNCTION包含将BIT_VECTORS转换成INTEGER类型和计算MAR+1的函数声明，见图8-39。

注意到函数是完全通用的，可用于任何大小和下标范围的向量。比较属性VEC'right和VEC'left判别下标范围是升序还是降序。两种情况应分别对待。

计算完MAR+1，注意到当一个二进制向量被递增时，最低位往往需要求反。而且，如果在向量的低位一侧有一串‘1’，则所有的‘1’也求反。从右向左扫描得到的第一个‘0’同样被求反。剩余的位保存不变。变量CARRY用来决定何时停止对位取反。只要扫描过的所有

位与CARRY的“与”结果是TRUE，则扫描只找到1；当扫描过的所有位与CARRY的“与”结果变为FALSE时，则表示至少找到了1个‘0’。CARRY初始化为1，以强制第1位求反。算法用下面的两个例子说明。

```
 0101 0111        1010 1010
+_______1        +_______1
 0101 1000        1010 1011
```

```
-- ROM process
  ROMP: process (MAR)
    type MEM_TYPE is array (0 to 255
      of BIT_VECTOR (63 downto 0);
    constant MEM: MEM_TYPE:=
  --|<-------- CS ---------->|<-NAD->|<------------ LCS ----------->|
  (0=>B"0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000",
   1=>B"0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000",
   2=>B"0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000",
      others => (others => '0'));
  begin
    CWORD <= MEM(BIT_VECTOR_TO_INT(MAR)) after ROM_DELAY;
  end process ROMP;
  -- Decode the MIR to obtain NAD,CS, and LCS.
  DECODEP: process (MIR)
  begin
    NAD <= MIR(39 downto 32);
    LCS <= MIR(31 downto 0);
    CS  <= MIR(63 downto 40);
  end process DECODEP;
  -- Address generation process.
  AGL: process (MAR, NAD, C, CS)
    variable AS: BIT;  -- AS is TRUE if selected condition is TRUE.
  begin
AS:=(CS(23) and C(23)) or (CS(22) and C(22)) or (CS(21) and C(21))
 or (CS(20) and C(20)) or (CS(19) and C(19)) or (CS(18) and C(18))
 or (CS(17) and C(17)) or (CS(16) and C(16)) or (CS(15) and C(15))
 or (CS(14) and C(14)) or (CS(13) and C(13)) or (CS(12) and C(12))
 or (CS(11) and C(11)) or (CS(10) and C(10)) or (CS(9) and C(9))
 or (CS(8) and C(8))   or (CS(7) and C(7))   or (CS(6) and C(6))
 or (CS(5) and C(5))   or (CS(4) and C(4))   or (CS(3) and C(3))
 or (CS(2) and C(2))   or (CS(1) and C(1))   or (CS(0) and C(0));
    case AS is
      when '0' => NMAR <= INC(MAR) after AGL_DELAY;
      when '1' => NMAR <= NAD after AGL_DELAY;
    end case;
  end process AGL;
end ALGORITHMIC;
```

图8-38b BMCU的算法级VHDL描述

8.3.5 状态机微程序控制器的设计

通常可以直接从器件的算法级描述直接设计微程序控制器。为了说明这种方法，我们将开发一种系统化方法，设计本章早些时候介绍的用VHDL建模方式表示的状态机。

根据状态机的寄存器传输方式的算法级描述来设计微程序控制器，包括下面这些步骤：

(1) 过程1：状态机微程序控制器的设计。

1) 通过扫描VHDL描述中的语句，建立一张表，表示所有状态的转换以及确定每一个转换的条件。

2) 根据第1步建立一个条件列表。在将ROM地址赋给状态之后，其中的某些条件可能不

再需要。

```
-- Package of functions for BMCU.
--
package BMCU_FUNCTIONS is
  function BIT_VECTOR_TO_INT(VEC: BIT_VECTOR) return INTEGER;
  function INC(A: BIT_VECTOR) return BIT_VECTOR;
end BMCU_FUNCTIONS;
--
package body BMCU_FUNCTIONS is
  -- Convert a BIT_VECTOR to INTEGER.
  function BIT_VECTOR_TO_INT(VEC: BIT_VECTOR)return INTEGER is
    variable SUM, WT: INTEGER:=0;
  begin
    if VEC'right <= VEC'left then
      for N in VEC'right to VEC'left loop
        if VEC(N)='1' then
          SUM := SUM + (2 ** WT);
        end if;
        WT := WT+1;
      end loop;
    else
      for N in VEC'right downto VEC'left loop
        if VEC(N)='1' then
          SUM := SUM + (2 ** WT);
        end if;
        WT := WT+1;
      end loop;
    end if;
    return SUM;
  end BIT_VECTOR_TO_INT;
-- Increment a BIT_VECTOR.
  function INC(A: BIT_VECTOR) return BIT_VECTOR is
    variable CARRY: BIT:='1';
    variable RESULT: BIT_VECTOR (A'range);
  begin
    if A'right <= A'left then
      for N in A'right to A'left loop
        if CARRY = '1' then
          RESULT(N) := not A(N);
        else
          RESULT(N) := A(N);
        end if;
        CARRY := CARRY and A(N);
      end loop;
    else
      for N in A'right downto A'left loop
        if CARRY = '1' then
          RESULT(N) := not A(N);
        else
          RESULT(N) := A(N);
        end if;
        CARRY := CARRY and A(N);
      end loop;
     end if;
     return RESULT;
  end INC;
end BMCU_FUNCTIONS;
```

图8-39 BMCU算法级模型的函数包

3) 标明复位状态。通常由规格说明提供复位状态。如果规格说明中没有指明，任意选一个复位状态。

4) 对每个状态赋予一个或多个ROM地址。ROM地址的赋值将在很大程度上影响控制器的性能和耗费。没有已知的有效的优化算法来优化地址赋值。最优赋值在于找出状态图中的最长路径，这是一个复杂的图论问题，超出了本书的范围。

5) 从第2步创建的条件列表中删除冗余的信号，并随意将状态信号赋给状态输出。这个赋值不会影响控制器的性能和耗费。尽管每个条件有可能在许多不同的分支情况下使用，但每个条件只可被通过一次。

6) 创建每个状态的传输列表和所要的输出列表。在表中包括传输和输出所需的条件。

7) 根据第6步创建的传输和输出列表，产生定时传输和输出控制信号列表。通常，对每个传输条件对和每个输出条件对都需要一个独立的信号。如果控制信号之间的关系能够被一个优化过程识别，也有简化的可能。可随意地将控制信号赋给控制输出。

8) 画出表示条件和控制信号产生的框图，包括复位和时钟信号。

9) 根据步骤1~8中创建的列表信息确定ROM程序，该过程与汇编程序执行的过程类似。

作为例子，考虑8.2节的串并转换器（STOP）。正如在第1步所确定的，扫描图8-25a和8-25b中的VHDL代码的语句，创建出图8-40中的表格，目前先忽略列“类型”，在第4步将会补上。

PS	NS	条件	类型
S0	S0	$R+\overline{A}$	B
	S1	$\overline{R}A$	I
S1	S2	$\overline{R}$	I
	S0	R	B
S2	S3	$\overline{R}$	I
	S0	R	B
S3	S4	$\overline{R}$	I
	S0	R	B
S4	S5	$\overline{R}$	I
	S0	R	B
S5	S1	$\overline{R}A$	B
	S0	$R+\overline{A}$	B

图8-40 执行算法的第1步得到的转换表

在第2步创建的条件列表是：

$$R+\overline{A},\overline{R}A,\overline{R},R$$

对于第3步，复位状态S0在问题的规格说明中定义。

尽管第4步中地址的优化赋值是个难题，下面的简单方法将在一个较短的时间得到一个合理的赋值方案。注意，这个过程与汇编程序第一遍中的地址赋值很相似。实际上，对控制器的ROM编程一般是使用类似于汇编程序的工具来实现的。

(2) 过程2：分配ROM地址

1) 用变量P表示当前状态，将P 初始化为复位状态。用LC代表位置计数器，包含下一次得

到的存储器地址。初始化LC=0。

2) 将状态P赋给地址LC，如果所有的状态都赋给了存储器地址，则停止，否则转第3步。

3) 确定实现状态P所需ROM地址的个数，设NSP是状态P之后的状态的数目。

a) 如果NSP=1，令LC=LC+1，只需要一个ROM地址。如果下一个状态仍没有被赋值，令P等于下一个状态，用I标记下一个状态用于增量，转向第2步。如果下一个状态已经分配给一个ROM地址，则标记下一个状态为B（用于分支），随意选择一个未赋值的状态P转向第2步。

b) 如果NSP>1，且至少P以后的一个状态还没有分配ROM地址，随意令Q等于一个没有分配地址的次态。这里就是可以引入优化的地方。优化算法可能对Q进行优化选择。如后面将要给出的，状态P需要NSP-1存储空间。因而，用I标记Q，用B标识P的其他后继状态，令LC=LC+NSP-1，P=Q，转到第2步。

c) 如果NSP > 1，且P的所有状态分配了ROM地址，实现状态P需要NSP个存储空间。用B标记所有的后继状态，令LC=LC+NSP，任意选择未赋值的状态赋给P，转到第2步。

在第1步创建的表中填入标记B或I很方便。对于作为例子的问题，过程2的执行创建了图8-40中的“类型”列。图8-41给出了由过程2将ROM地址赋给状态的结果。

状态	地址
S0	0
S1	1
S2	2
S3	3
S4	4
S5	5

图8-41　过程1的第4步，ROM地址赋值给状态

因为状态5需要两个存储器空间，所以下一个可得到的地址为7。

根据图8-41给出的地址赋值和图8-40中的信息，与汇编程序的第二遍相似，可以轻松建立ROM程序。在那样做之前，要定义需要的条件和控制信号。

在过程1的第5步，必须从数据单元传递到控制单元的条件信号对应于图8-40中标记为B的行。图8-42给出了将条件随意赋值给控制单元的条件输入的结果。注意到不需要条件$\overline{R}$，因为所有需要条件$\overline{R}$的后继状态都标记为I，表示转换通过递增MAR而完成。

输入	条件
C23	R
C22	$R + \overline{A}$
C21	$\overline{R}A$

图8-42　在过程1第5步进行的条件赋值

在过程1的第6步，再次扫描VHDL语句创建转换和输出的列表。至于为什么第1步和第6步的列表不可以在同一遍扫描之内完成，无原因可言。我们将其分成两步，只是为了清晰起

见。图8-43给出执行过程1的第7步得到的结果。注意，条件为1，表示无条件传输或输出。在这个例子中，所有的传输和输出都是无条件的。

状态	传输	条件	完成	条件	Z	条件
S0	-	-	0	1	-	-
S1	Shift SR	1	0	1	-	-
S2	Shift SR	1	0	1	-	-
S3	Shift SR	1	0	1	-	-
S4	Shift SR	1	0	1	-	-
S5	-	-	1	1	SR	1

图8-43 在过程1的第6步产生的传输和输出列表

图8-44显示了必须从控制器传到数据单元的控制信号列表。因为只有两个不同的输出状态，输出可以用一个叫作DONE_CONTROL的控制信号来控制。只需要一个转换，由控制信号SHIFT指示。图8-44给出了控制信号到控制单元输出端口的任意赋值。

输出	信号	传输/输出
LCS31	SHIFT	SR ← D & SR[3:1]
LCS30	DONE_CONTROL	DONE=1, Z=SR

图8-44 在过程1的第7步产生的控制信号列表

现在已经标记了控制器和数据单元的所有信号，我们可以建立数据单元的框图（第8步）。图8-45给出了STOP器件的框图。注意所有未使用的输入连接到常量逻辑‘0’（用符号名F代表）。使用过程1产生的信息直接填入控制ROM的内容（第9步）。图8-46显示了ROM的最终内容。

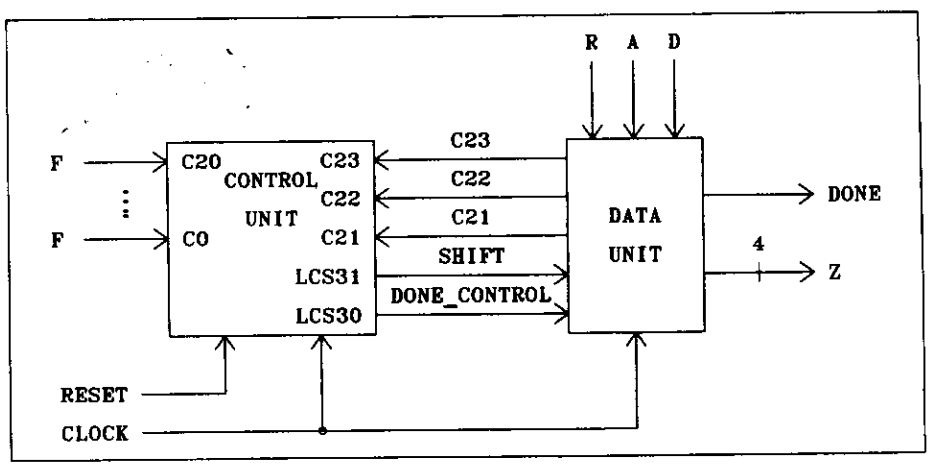


图8-45 显示模块间信号的STOP器件的框图

ROM地址0处的内容取自表中的信息。根据图8-40，分支操作的条件（下一个状态标号为

B）是C22(R + $\bar{A}$)，因而，CS字段中列CS22有单个的1。因为当条件C22为TRUE时，转化到状态S0，分支NAD字段的地址为S0（赋值为地址0）。SHIFT（LC31）中的0意味着在该状态不执行移位操作。DONE_CONTROL（LCS30）中的0表示DONE=0，而且Z无定义。当C22是FALSE时，ROM地址递增，且器件进入在地址1处的状态S1，地址1~4处的内容可同样确定。

		CS				NAD	LCS			
状态	加	23	22	21	20...0	7...0	31	30	29...0	
		63	62	61	60...40	39...32	31	30	29...0	CWORD位
S0	0	0	1	0	0...0	00000000	0	0	----	
S1	1	1	0	0	0...0	00000000	1	0	----	
S2	2	1	0	0	0...0	00000000	1	0	----	
S3	3	1	0	0	0...0	00000000	1	0	----	
S4	4	1	0	0	0...0	00000000	1	0	----	
S5	5	0	0	1	0...0	00000001	0	1	----	
S5	6	0	1	0	0...0	00000000	0	1	----	

图8-46　STOP器件的控制ROM的ROM内容

对状态S5的控制信号编程更加复杂，因为该分支不是内置于控制器的两路分支。有两个直接转移，而不是一个直接转移和一个递增转移。当有多于一个的直接转移时，控制器必须通过一次检查一个条件以执行它们。这需要多个ROM地址。在当前情况下，在ROM地址5处，如果条件C21(R$\bar{A}$)是TRUE，器件转移到地址1（状态S1），如NAD字段中所指出的。如果C21为FALSE，ROM转移到下一个地址6，其中检查条件C22(R + $\bar{A}$)，如果C22为TRUE，执行到地址0（状态S0）的跳转。实际上，状态5的执行伸长到2个微指令周期。SHIFT（LCS31）是0，因为这个周期不进行移位操作。DONE_CONTROL（LCS30）为1，指示在该状态，结果在Z输出上。状态5伸长到2个微指令，相对于问题的规格说明，改动了时序。参见8.3.6节设计的局限性中对此问题的讨论及本章结尾的问题列表。

确定了ROM的内容，则完成了该控制器的设计。既然控制器的硬件设计已经完成，唯一的设计任务就是编程ROM芯片，并将其插入控制器板。不需要增加多少工作，VHDL语言可以用来在ROM编程之前验证设计。在BMCU的算法模型中，插入对ROM内容的声明并将构架名由ALGORITHMIC改为STOP。实体声明没有改变。图8-47显示了得到的控制单元的构架。

还要设计数据单元。这是一个通用设计过程，与控制器是微程序的还是硬连线的无关。图8-48给出了数据单元的数据流模型。

图8-49给出了用来验证设计操作的正确性的测试程序包。控制单元和数据单元表示为测试程序包中的组件。组件根据图8-45中所示的框图进行互连。因为对一个模块的所有输入必须连接，未使用的对控制器的条件输入都连入常量逻辑‘0’（信号F）。未使用的输出则未连。这与实际硬件设计中的良好设计习惯一致。注意，其中包含了典型延时以仿真实际硬件延时。

8.3.6　微程序控制单元的普遍性和局限性

微程序控制单元的一个主要优点是同一块控制器板或控制器芯片可以用于许多设计。本节使用BMCU做为例子，讨论应用的局限性。

```
-- Algorithmic level VHDL description of BMCU.
use work.BMCU_FUNCTIONS.all;
architecture STOP of BMCU is
  -- MAR is the Memory Address Register for the ROM.
  -- MIR is the Memory Instruction Register for the ROM.
  signal MAR, NMAR: BIT_VECTOR (7 downto 0);-- NMAR is next value.
  signal CWORD: BIT_VECTOR (63 downto 0);-- Control word from ROM.
  signal MIR: BIT_VECTOR (63 downto 0);--Word read from ROM.
  signal NAD: BIT_VECTOR (7 downto 0); -- Branch address.
  signal CS: BIT_VECTOR (23 downto 0); -- Condition Select Field
begin
  -- Memory Address Register process.
  MARP: process (CLK, RESET)
  begin
    if RESET='1' then
      MAR <= B"0000_0000" after MAR_DELAY;
    elsif CLK'event and CLK='0' then
      MAR <= NMAR after MAR_DELAY;
    end if;
  end process MARP;
  -- Memory Instruction Register Process
  MIRP: process (CLK, RESET)
  begin
    if RESET='1' then MIR <=
     B"00000000_00000000_00000000_00000000_00000000_00000000_00000000_00000000"
      after MIR_DELAY;
    elsif CLK'event and CLK='1' then
      MIR <= CWORD after MIR_DELAY;
    end if;
  end process MIRP;
  -- ROM process
  ROMP: process (MAR)
    type MEM_TYPE is array (0 to 255) of BIT_VECTOR (63 downto 0);
    constant MEM: MEM_TYPE:=
  -----|<------- CS -------->|<-NAD->|<---------- LCS ---------->|
   (0=>B"01000000_00000000_00000000_00000000_00000000_00000000_00000000_00000000",
    1=>B"10000000_00000000_00000000_00000000_10000000_00000000_00000000_00000000",
    2=>B"10000000_00000000_00000000_00000000_10000000_00000000_00000000_00000000",
    3=>B"10000000_00000000_00000000_00000000_10000000_00000000_00000000_00000000",
    4=>B"10000000_00000000_00000000_00000000_10000000_00000000_00000000_00000000",
    5=>B"00100000_00000000_00000000_00000001_01000000_00000000_00000000_00000000",
    6=>B"01000000_00000000_00000000_00000000_01000000_00000000_0000_000_00000000",
      others => (others => '0'));
  begin
    CWORD <= MEM(BIT_VECTOR_TO_INT(MAR)) after ROM_DELAY;
  end process ROMP;
-- Code is continued on next page.
```

图8-47 STOP控制单元的VHDL模型

BMCU限于检查数据单元的24个条件，因为器件只有24个条件输入。由于总共只有32个控制输出，控制器被限制在控制数据单元中的32个独立操作。因为NAD字段有8位，因而ROM程序最多只有256个控制字。提出所有这些限制是因为控制字中各个字段的大小固定。同样，任何标准的微程序控制器都受其控制字中各字段的固定大小的限制。

也许微程序设计最严重的限制在于产生下一条地址的方法所带来的限制。通过选择一个条件输入，BMCU使用一种非常初级的两路分支策略：如果条件为TRUE，则执行条件跳转到

ROM中的任何一个位置处，如果条件为FALSE，则转到ROM中下一条顺序地址处。尽管通过执行多个两路分支指令，可以执行三路分支或m路分支，但这对控制器时序有负面影响。例如，执行三路分支涉及两个条件和一个缺省递增，需要2个完整的微指令周期。通常，执行m路分支需要$m-1$个微指令周期，因而，选择下一个地址耗费的时间取决于分支操作的复杂性。实际上，某些微指令周期将被拉长以提供计算下一条地址的时间。这种微指令周期的扩展也许将会在器件原有的输入和输出上引起时序问题，并且需要更多的执行时间。

```
  -- Decode the MIR to obtain NAD and S.
    DECODEP: process (MIR)
    begin
      NAD <= MIR(39 downto 32);
      LCS <= MIR(31 downto 0);
      CS  <= MIR(63 downto 40);
    end process DECODEP;
    -- Address generation process.
    AGL: process (MAR, NAD, C, CS)
      variable AS: BIT;  -- Selected condition is TRUE.
    begin
      AS := (CS(23) and C(23)) or (CS(22) and C(22)) or
            (CS(21) and C(21)) or (CS(20) and C(20)) or
            (CS(19) and C(19)) or (CS(18) and C(18)) or
            (CS(17) and C(17)) or (CS(16) and C(16)) or
            (CS(15) and C(15)) or (CS(14) and C(14)) or
            (CS(13) and C(13)) or (CS(12) and C(12)) or
            (CS(11) and C(11)) or (CS(10) and C(10)) or
            (CS( 9) and C( 9)) or (CS( 8) and C( 8)) or
            (CS( 7) and C( 7)) or (CS( 6) and C( 6)) or
            (CS( 5) and C( 5)) or (CS( 4) and C( 4)) or
            (CS( 3) and C( 3)) or (CS( 2) and C( 2)) or
            (CS( 1) and C( 1)) or (CS( 0) and C( 0));
      case AS is
        when '0' => NMAR <= INC(MAR) after AGL_DELAY;
        when '1' => NMAR <= NAD after AGL_DELAY;
      end case;
    end process AGL;
  end STOP;
```

图8-47　STOP控制单元的VHDL模型（续）

对于STOP器件，在状态5需要2个微指令周期来执行分支意味着，如果在第二个微指令周期A上出现脉冲，该脉冲信号将会被错过。因为那时控制器并没有查看A信号。通常，这种问题的解决只有通过在器件之间使用耗时的“握手”协议来实现。注意，在设计这两个器件时，控制器和数据单元之间没有这个问题。然而，如果要求当前器件与其他器件之间存在固定的时序关系，根据原来的规格说明，则常常不可能实现原来的输入和输出信号。例如，STOP器件指定A脉冲只需保持一个时钟周期，而且在下一个时钟周期数据就会到来，这时使用BMCU做为微控制器以实现输入信号之间的复杂的固定时序关系几乎是不可能的。实际上，任何微程序控制器的寻址策略将在实现的系统上加上时序限制。

8.3.7　其他的状态选择方法

在BMCU中，CS字段中的每个位表示一个条件。这种方法叫作线性选择，或“单热”。有

大量的编码用来进行条件选择。主要的折衷在于译码的耗费与控制字中需要的位数之间。对于我们的选择，无译码耗费，但是条件的数量限制为控制字中控制选择字段中位的个数。这是选择的一个极端。

```
-- Data Unit for Serial to Parallel Converter
--
entity DATASTOP is
  generic (SHIFT_DELAY, GATE_DELAY: TIME);
  port (R, A, D, CLK, SHIFT, DONE_CONTROL: in BIT;
        Z: out BIT_VECTOR(3 downto 0);
        DONE, C23, C22, C21: out BIT);
end DATASTOP;
--
architecture DATAFLOW of DATASTOP is
  signal SHIFT_REG: BIT_VECTOR (3 downto 0);
begin
  --
  -- Shift register
  --
  SHIFT_REG <= D & SHIFT_REG(3 downto 1) after SHIFT_DELAY
    when CLK'event and CLK='1' and SHIFT='1'
    else SHIFT_REG;
  --
  -- Condition signals needed in the control unit
  --
  C23 <= R;
  C22 <= R or not A after GATE_DELAY;
  C21 <= not R and A after GATE_DELAY;
  --
  -- Output signals
  --
  DONE <= DONE_CONTROL;
  Z <= SHIFT_REG;
end DATAFLOW;
```

图8-48 STOP数据单元的VHDL模型

在选择的另一极端，CS字段可以完全译码，BMCU的CS字段的16位可用16位输入、65 536输出译码电路进行译码。这代表可用16位数据字段选择的条件的最大数目，也代表译码的最大耗费，因为一个这样的译码器将是十分昂贵的。从另一方面看，通过使用4位的CS字段和4×16的译码器来产生条件选择信号，BMCU的控制字可以从64位减少到53位。ROM空间的节省抵消了译码器的耗费，因为在这种情况下，4×16译码器只需要16个4输入门和4个反相器。

在两个极端之间有许多选择。例如一个k-out-of-n编码特别容易译码并比线性选择编码提供更多的条件。在k-out-of-n编码中，每个条件的编码包含k个‘1’和$n-k$个‘0’。很明显，线性选择编码是1-out-of-n编码。我们将使用6位的CS字段来说明这些概念。线性选择方法不需要译码逻辑，但只能容纳6个独立的条件。全译码允许2^6=64个条件。需要64个5输入门和6个反相器进行译码。对于2-out-of-6编码，有15种方法来指定2个‘1’和4个‘0’。这取决于从6个项中取出2个项的个数。每个选择信号需要一个两输入与门。因而，总耗费是15个2输入与门，可以得到15个状态。3-out-of 6编码需要20个3输入与门，提供了20个状态信号。图8-50总结了至今讨论的选择范围。许多其他编码具有不同的特性，对这些问题的讨论已经超出

了本书范围。

```
use work.all;
entity TEST_BENCH is
end TEST_BENCH;
--
architecture BMCU_TEST of TEST_BENCH is
  signal R,A,D,CLK,INIT,RESET: BIT;
  signal C23, C22, C21: BIT;
  signal SHIFT, DONE_CONTROL: BIT;
  signal DONE: BIT;
  signal Z: BIT_VECTOR (3 downto 0);
  signal X: BIT_VECTOR (3 downto 1);
  signal F: BIT; -- Constant false signal.
  --
  component DATA_UNIT
     generic (SHIFT_DELAY, GATE_DELAY: TIME);
     port (R, A, D, CLK, SHIFT, DONE_CONTROL:  in BIT;
           Z: out BIT_VECTOR(3 downto 0);
           DONE, C23, C22, C21: out BIT);
  end component;
  --
  component MICRO_CONTROL_UNIT
   generic (AGL_DELAY, MAR_DELAY, ROM_DELAY, MIR_DELAY,
            MIR_SETUP, MAR_SETUP: TIME);
   port (C: in BIT_VECTOR (23 downto 0); -- Conditions.
         CLK, RESET: in BIT;  -- System clock and reset.
         LCS: out BIT_VECTOR (31 downto 0));-- Level CS.
  end component;
  --
  -- Component configuration statememts.
for L1: DATA_UNIT  use entity DATASTOP(DATAFLOW);
  for L2: MICRO_CONTROL_UNIT use entity BMCU(STOP);
begin
  L1: DATA_UNIT
    generic map (20 ns, 10 ns)
    port map (R, A, D, CLK, SHIFT, DONE_CONTROL, Z,
              DONE, C23, C22, C21);
  L2: MICRO_CONTROL_UNIT
    generic map (50 ns, 20 ns, 50 ns, 20 ns, 5 ns, 5 ns)
    port map (C(23) => C23, C(22) => C22, C(21) => C21,
          C( 0) => F, C( 1) => F, C( 2) => F, C( 3) => F,
          C( 4) => F, C( 5) => F, C( 6) => F, C( 7) => F,
          C( 8) => F, C( 9) => F, C(10) => F, C(11) => F,
          C(12) => F, C(13) => F, C(14) => F, C(15) => F,
          C(16) => F, C(17) => F, C(18) => F, C(19) => F,
          C(20) => F,
          RESET => RESET, CLK => CLK,
          LCS(31) => SHIFT, LCS(30) => DONE_CONTROL);
  -- Code continued on next page.
```

图8-49 STOP系统的VHDL模型

8.3.8 其他分支方法

有很多修改地址产生逻辑的方法，从而可减少加给设计的限制。主要的折衷在于：1) 附加的分支能力引起控制字增加位，从而增加耗费；2) 对地址产生逻辑附加的硬件耗费。

图8-51说明了一个有四种不同分支方法的微程序控制器的控制字。2位AT字段选择4个地址生成类型。6位的CS字段将被全译码以选择64个条件输入之一。标记为NAD1、NAD2和NAD3的3个8位字段用来计算目标地址。ROM地址是16位地址，所以ROM程序可以是64K字大小。32位的LCS字段包括32级控制信号，可用于在控制单元之中控制数据传输。

```
  -- Clock process
    process
    begin
      CLK <= '0';
      wait for 200 ns;
      CLK <= '1';
      wait for 200 ns;
    end process;
    --
    -- Process to provide inputs.
    F <= '0';
    RESET <= '1', '0' after 30 ns;
    process (X)
    begin
      R <= X(3);
      A <= X(2);
      D <= X(1);
    end process;
    --
    -- Assignment of values to input X
    X <= "100" after    50 ns, "010" after  650 ns,
         "001" after  1050 ns, "000" after  1450 ns,
         "001" after  1850 ns, "001" after  2250 ns,
         "000" after  2650 ns, "000" after  3050 ns,
         "010" after  3450 ns, "000" after  3850 ns,
         "001" after  4250 ns, "001" after  4650 ns,
         "000" after  5050 ns, "010" after  5450 ns,
         "001" after  5850 ns, "001" after  6250 ns,
         "011" after  6650 ns, "010" after  7050 ns,
         "010" after  7450 ns, "000" after  7850 ns,
         "101" after  8250 ns, "001" after  8650 ns,
         "001" after  9050 ns, "000" after  9450 ns,
         "001" after  9850 ns, "000" after 10250 ns,
         "001" after 10650 ns, "000" after 11050 ns,
         "000" after 11450 ns;
end BMCU_TEST;
```

图8-49 STOP系统的VHDL模型（续）

编码	最大条件数	耗费
线性	6	无
2-out-of-6	15	15个二输入AND门
3-out-of-6	20	20个三输入AND门
完全	64	64个五输入AND门

图8-50 6位字段译码的部分选择范围

当AT=00时，控制器无条件跳转到字段NAD1和NAD2并置而得到的16位ROM地址处。这

种寻址模式没有使用CS和NAD3字段。图8-52a说明了无条件分支操作的格式。

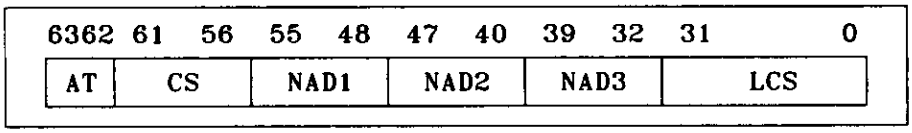


图8-51　灵活的分支微控制器的控制字

当AT=01时，控制器执行与BMCU相同的两路分支。CS字段被全译码以选择64个条件输入之中的一个。如果选择的输入条件为TRUE，控制器转移到由字段NAD1和NAD2并置而得到的16位地址处。如果选择的条件是FALSE，则控制器转到地址MAR+1。图8-52b说明了两路分支的格式。

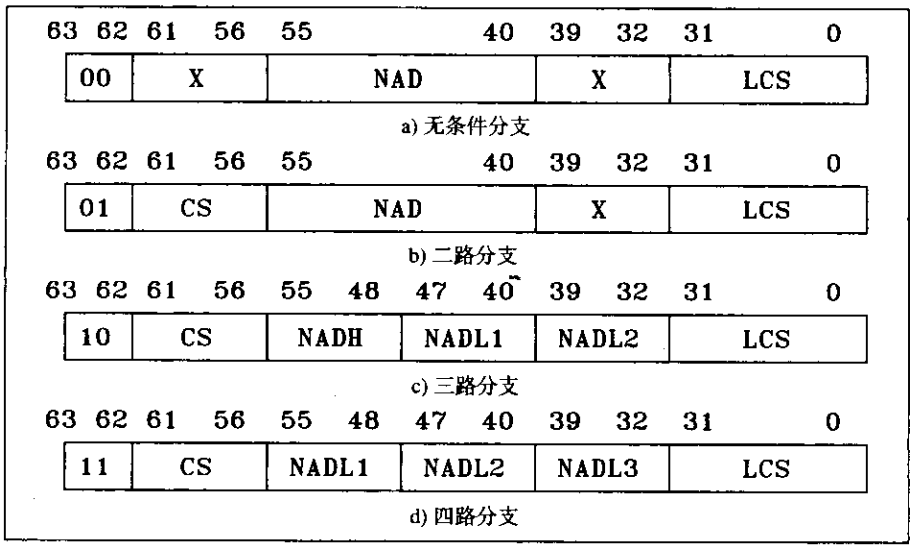


图8-52　扩展分支方法的控制字格式

当AT=10时，控制器执行3路分支。使用者必须选择两个相连的条件输入，两个条件中的前一个表示偶数（例如，10和11）。CS字段被全译码以得到由CDL表示的两位相连数值的低位。控制器然后对CS字段的低位求反，将结果全译码以选择两个相连数值的高位；用CDH表示。当然，用户必须保证这两个条件不可能同时为TRUE。如果CDL为TRUE，控制器转移到由NAD1和NAD2并置得到的地址处。如果CDH为TRUE，控制器转移到由NAD1和NAD3并置的地址处。如果CDL和CDH都为FALSE。控制器转移到MAR+1。图8-52c说明了三路分支格式。注意两个分支地址具有相同的高8位（用NADH表示）。如果CDL为TRUE，低位分支地址则为NADL1，若CDH为TRUE，则为NADL2。选择了这种方法，就不必在控制字中加入更多的位。如果在一个控制字中包含2个完整的16位地址，则控制字就会增加到72位。减少控制字的大小以增加分支地址的限制为代价。

如果AT=11，控制器执行四路分支。用户必须选择3个连续条件输入，且其第一个是4的偶倍数。即条件的最右边两位必须都是逻辑0。例如，用户可以选择条件24、25和26，其二进制表示分别为011000，011001和011010。低位由编程者在CS字段编码。控制器然后对CS字段

进行全译码以得到CD1（在本例中，CD1=24），对低位求反，再全译码得到CD2（CD2=25）。最后，对右边的第二位求反再全译码得到CD3（CD3=26）。用户必须保证这三个状态信号互斥。如果CD1=1，下一个地址并置MAR的高半字与NAD1。如果CD2=1，下一个地址并置MAR的高半字与NAD2。如果CD3=1，下一个地址并置MAR的高半字与NAD3。如果CD1=CD2=CD3=0，下一个地址为MAR+1。注意该方法限制了所有的三个分支地址都与当前地址在同一个256字的页内。图8-25d说明了这种四路分支的格式。所有三个直接分支地址必须共享高半字。可接受这些限制，以保证控制字的长度为64位。要在控制字中有三个完整的分支地址，需要多加24位（总共88位）。

使用这种组合地址产生策略可以满足大部分的应用，而不需要利用扩展时钟周期以容纳更多的分支。当然，五路或更多的分支不可能直接实现，而且，对于三路和四路分支必须满足一些限制，但对于大多数应用这不难满足。

要在分支中附加更多的灵活性，其耗费就是更加复杂的地址产生逻辑（见习题8.28）。

习题

8.1 通过从消多路器地址集合中除下列变量从图8-6的基本CASE语句模型中为器件COM，产生一个改进的CASE风格的VHDL描述。仿真每个模型。

a) N1

b) N0

c) M1

8.2 考虑习题8.1产生的改进的CASE风格的VHDL描述，把每个模型翻译为一个改进的MUX实现。

8.3 使用本章给出的规则，把下面为POS模型产生的每条VHDL语句翻译为一条适合NOR模型的VHDL语句：

a) X <= (not A or B or not C) and (not B or E or not C or not E) and (A or E or F)

b) Y <= (not A or not C) and (B or C) and (A or not B or not C)

c) Z <= A and (not B or C or D) and not E

d) P <= not A and not B

8.4 考虑图8-13的VHDL数据流模型：

a) 产生把图8-13的VHDL数据流模型自动翻译为只用NOR门实现的自动翻译规则。计划把每个赋值语句翻译成一个二级（或少于二级）NOR门配置。（可以使用NOT门对输入变量取反，但并不把它作为一级。）

b) 使用上面产生的规则，把图8-13的数据流模型翻译成NOR和NOT门组成的电路。

c) 为上述转换产生的门级电路建立一个结构VHDL模型。仿真该模型，验证它的精确程度。

8.5 考虑习题8.3产生的VHDL语句，使用习题8.4a产生的规则，把每一条VHDL语句翻译为一个二级NOR门电路。（可以使用NOR门把输入变量取反,但不作为一级）。

8.6 对图8-17组合逻辑的设计活动进行小结，着重使用POS标准格式和NOR门，画出组合逻辑设计活动的一个相近的图形表示，要求强调SOP标准格式和NAND门。

8.7 写出图8-10给出的COM器件优化SOP表示的VHDL模型。对模型仿真，验证它的精确程度。

8.8 为积之和（sum of products）VHDL模型转换到NAND VHDL模型产生一组规则。提示：应用教材中把积之和的VHDL模型翻译为NOR VHDL模型规则的对偶性原理。

8.9 根据习题8.8产生的规则，通过对习题8.7产生的SOP VHDL模型应用这些规则，为COM器件创建一个NAND VHDL模型。仿真模型以便验证它的精确性。

8.10 考虑习题8.9为COM器件创建的NAND VHDL数据流模型，执行下面任务：

a) 产生把NAND VHDL数据流模型自动翻译为用NAND门实现的规则，每条赋值语句翻译为二级（或更少）NAND门配置。（可以使用NOT门对输入变量求反，但不作为一级。）

b) 使用a产生的规则，把COM器件的NAND VHDL数据流模型翻译为NAND门和NOT门构成的电路。

c) 为上一步翻译获得的门级电路产生一个结构VHDL模型。仿真此模型，验证它的精确程度。

8.11 设计一个计算两个三位二进制正数乘积的乘法器硬件电路（M）。图8-53是器件M的框图，A_2、A_1、A_0表示一个3位数，B_2、B_1、B_0表示另一个3位数，$M_5M_4M_3M_2M_1M_0$表示6位乘积。如：若A_2，A_1，A_0=110，B_2，B_1，B_0=011，则$M_5M_4M_3M_2M_1M_0$=010010。使用图8-17所示的组合设计方法图的路径，从自然语言规格说明开始，直到ROM实现结束。

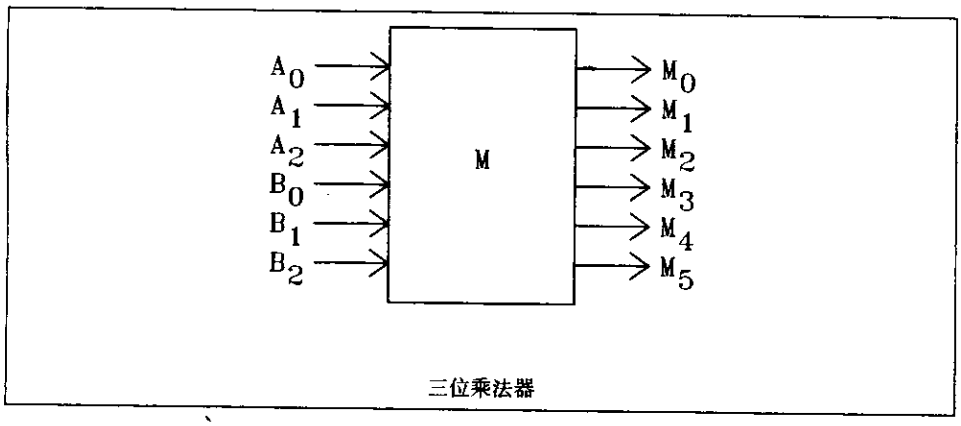


图8-53 三位二进制乘法器

a) 记录路径中从一种表达方式到另一种表达式的翻译过程。

b) 完全仿真路径中所有的VHDL表示。

c) 画出带有全部命名信号的最终实现的框图。

d) 若可以使用硬件仿真程序，则完全仿真最终的硬件实现。

8.12 重复习题8.11对3位乘法器的设计，使用图8-17中组合设计方法图所示的路径，从自然语言规格说明开始，到标准MUX结构表示结束。

8.13 重复习题8.11对3位乘法器的设计，使用图8-17中混合设计方法图所示的路径，从自然语言规格说明开始，到改进MUX结构表示结束。

8.14 电子传感器的传感输出从一个值变为更高或更低值的时候，瞬间电流变化很大。例如，一个4位标准二进制BCD码输出的电子传感器，输出值由7（0111）变为8（1000）时，4位输出信号都必须改变。由于4位信号无法同时改变，实际输出序列为：

0111
0110
0100
0000
1000

被测物理量（如温度）将在7和8之间停留相当长的时间，实际上，温度值在7和8之间会改变多次，因为温度中途徘徊于7和8之间。在这种情况下，采样传感器输出的计算机可能读出的值为0~15的不规则序列，而实际的值在7和8之间。使用格雷码可以防止这种瞬变。下面的图给出了十进制格雷码与二进制BCD码的比较：

十进制数字	格 雷 码	BCD码
0	0000	0000
1	0001	0001
2	0011	0010
3	0010	0011
4	0110	0100
5	1110	0101
6	1010	0110
7	1011	0111
8	1001	1000
9	1000	1001

在格雷码中，用1011表示7，1001表示8。因为只有左边第3位不同，所以采样传感器输出的计算机读出的值可能为7或8，而不是BCD码中无规则的结果。但是，信号采样后可能需要把格雷码转换为BCD码以便于计算机的内部处理。设计一个把格雷码采样信号翻译为BCD码的硬件电路。设备的输入为4位格雷码，输出为4位等价的BCD码，非有效输入组合（0100,0101,0111, 1100,1101,1111）将作为不考虑的情况。使用图8-11中的设计方法图中的下面路径之一：

a) 从自然语言规格说明开始到ROM图形表示结束。
b) 从自然语言规格说明开始到标准MUX结构图表示结束。
c) 从自然语言规格说明开始到改进MUX结构图表示结束。
d) 从自然语言规格说明开始到OR-AND结构图表示结果。
e) 从自然语言规格说明开始到OR-AND结构VHDL表示结束。
f) 从自然语言规格说明开始到NOR-OR结构图表示结束。
g) 从自然语言规格说明开始到NOR-OR结构VHDL表示结束。
h) 从自然语言规格说明开始到AND-OR结构图表示结束，对图8-11中所示的结构图使用对偶概念。
i) 从自然语言规格说明开始到AND-OR结构VHDL表示结束，对图8-11中所示的结构图使用对偶概念。
j) 从自然语言规格说明开始到NAND-NAND结构图表示结束。对图8-11中所示的结构图使用对偶概念。使用习题8.8和8.10的结论。
k) 从自然语言规格说明开始到NAND-NAND结构VHDL表示结束。对图8-11中所示的结构图使用对偶概念。

对于各种情况，执行下列步骤：

a) 记录路径中从一种表示翻译为另一种表示的过程。

b) 完全仿真路径中的所有VHDL表示。

c) 画出带有全部命名信号的最终实现的原理图。

d) 如果可以使用硬件仿真程序，则完全仿真最终的硬件实现。

8.15 对于习题8.14的格雷码,设计一个无效代码检测器，检测器的输出（E）当且仅当输入信号为无效输入码（0100,0101,0111,1100,1101,1111）之一时输出值为逻辑1。使用习题8.14中的设计路径之一执行同样的设计步骤。

8.16 设计一个与计算机接口的等值测试器。测试器从计算机接收4位数据序列并判断最后三个数据项是否相等。数据在4位数据总线上传输。测试器和计算机都是同步设备，但它们并不工作在同一个时钟下。不管计算机比测试器快还是慢，测试器都可以正确地独立工作。图8-54给出了等值测试器的接口和时序。当测试器完成当前数据处理并准备好处理下一数据项时，信号READY被置位。信号READY保持到计算机对DAV线置位并同时把4位数据放在DATA线上。当DAV线置位后，测试器至少等待一个完整的时钟周期后把4位数据发送到一个内部寄存器。在传递数据的后一个时钟周期里，测试器使READY线变低以通知计算机该数据已成功传输。计算机在发现READY线变低前一直保持DATA线活动，然后它变低DAV线并把数据从DATA线上移走，同时READY线变低。若最后三个数据项都相等时，输出EQ被置为1。当计算机发现READY线变低时，它知道EQ上有了期望的结果。它保存EQ的值，然后使DAV线变低。当测试器发现DAV线变低时，它要移出EQ的值并把READY线重新置为1。但是为了安全起见，它将在改变EQ并对READY线再次置位之前把EQ线的活动状态保持一个完整的时钟周期。使用本书中描述的设计方法：

a) 创建状态表。

b) 建立算法级VHDL模型来验证状态表。

c) 仿真算法级模型，使用可以完全验证功能规格说明的输入序列。

d) 把VHDL模型（手工）转换为硬件控制单元和数据单元。完整记录转换过程以及结果硬件电路。

e) 讨论对硬件自动翻译的可能性。若可以使用一种自动综合器，用它来综合算法级模型并与手工仿真进行比较。

f) 若可以使用一个硬件仿真程序，仿真控制和数据单元并与算法及仿真产生的输出进行比较。

8.17 把习题8.16描述的等值检测器综合为一个微程序控制单元。执行下面的设计步骤：

a) 使用本章描述的综合过程导出等值检测器的BMCU的ROM程序。

b) 画出等值检测器的框图，图中应有微程序控制单元和数据单元块。

c) 为微程序控制单元和数据单元开发一个类似于图8-47a和图8-47b给出的系统级的VHDL模型。

d) 仿真系统级VHDL模型，使用可以完整验证功能规格说明的输入序列。

e) 把系统级VHDL模型手工翻译为硬件。记录翻译过程并讨论自动翻译的可能性，若可以使用一种自动综合器，则用它进行综合。

f) 若可以使用一个硬件仿真程序，仿真上一步产生的硬件，使用验证系统级VHDL模型时的同一输入序列进行验证，并比较输出结果。记录并解释它们的不同之处。若输出有差别，它是由违反规格说明造成的还是由不同的表示产生的?

8.18 修改图8-25a中用于器件STOP的VHDL代码，把输出Z的未说明行为包括进去。

使用CD-ROM上的程序包MVL4。参考第5章有关MVL4的信息。

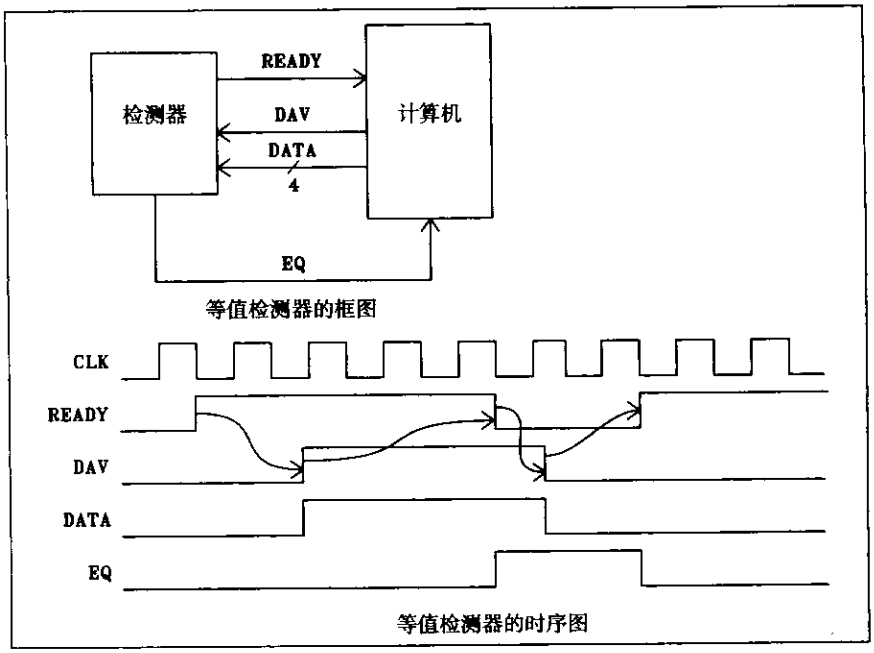


图8-54 等值检测器的接口和时序

8.19 使用状态表方法，设计把4位格雷码转换为BCD码的时序电路。对于格雷码的描述可以在习题8.14找出。输入输出由同一个系统时钟定时。假设器件接收和格雷码的最左位相符的一个START脉冲，格雷码剩下的位一个时钟周期接收一位（从左至右）。在4位格雷码接收结束时，设备并行输出相应的4位BCD码和一个DAV信号。下一次输出可以在最后一位输入后或更晚的时钟周期里开始。器件具有一个RESET输入，它把设备初始化到正确的开始状态。图8-55给出了框图和采样时钟图。

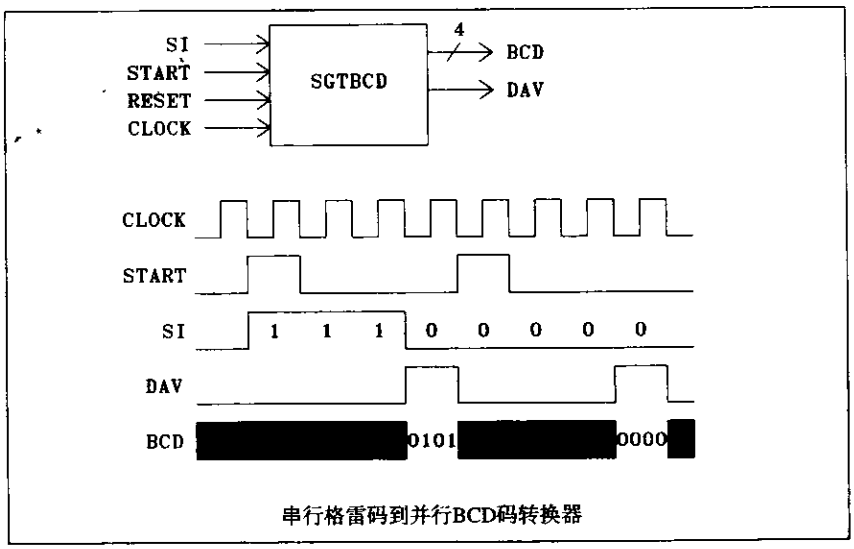


图8-55 格雷码到BCD码的转换器

8.20 设计一个十进制格雷码计数器，对习题8.14所示的格雷码序列计数。计数器从9到0推进。用一个系统时钟使得计数器同步。

8.21 使用状态表方法，设计一个检测连续4个1输入序列的Mealy序列检测器。检测器有一个二进制输入X和二进制输出Z。当且仅当最后4个输入都为逻辑1时信号Z的输出为逻辑1。这里给出了一个输入序列的例子：

X 01011111110111010111110
Z 00000011110000000000010

注意　输出1产生在第4个连续1输入的同时。

8.22 使用状态表方法，设计一个具有唯一二进制输入X和二进制输出Z的Mealy序列检测器。当且仅当最后4个输入为1100或1001时Z的输出为逻辑1。下面是采样的输入输出序列：

注意　输出1在被检测序列的第4位同时产出。

X 11001001101100110000
Z 00011001000000110010

8.23 使用状态表方法，设计一个具有唯一二进制输入X和二进制输出Z的Moore序列检测器。当且仅当最后4个输入为1101或0110时Z的输出为逻辑1。下面是采样的输入输出序列：

X 11011011010001101000-
Z 00001011011000000110O

注意　输出1在被检测序列后延迟一个时钟周期产生。

8.24 4位数据字以每块2到15个字的大小在一个4位数据总线上传输。每个数据字DAT（3到0）由三位信息位和一个校验位组成。校验位DAT（3）确保每个4位字中具有偶数个1。每块中最后一个字为校验字，字中的位确保块中的每一列中有偶数个1。下面的例子解释了这种格式：

	pddd
data word0	0011
data word1	1001
data word2	1111
data word3	0000
parity word	0101

字0的数据为011，第1位（校验位）为0是因为数据中的1有偶数个。在字1里的数据为001，校验位为1是因为数据中的1为奇数个。字4为校验字。注意到每列中所有的1都为偶数个。

当接收到一个数据块时，校验信息用来检测和纠正传输错误。若没有错误发生，每个接收字的奇偶校验均为偶数，每一列的奇偶校验也为偶数。若接收块中有1位发生错误，由于只有一行和一列的奇偶校验为奇数，因而可以检测并纠正这个错误。处于奇数奇偶校验行和奇数奇偶校验到交叉处的位可以被纠正。例如，假设前述数据块数据字2中到1的位在接收时发生错误，如下表所示。每一行右端的二进制变量当奇偶校验为偶数时它的值为0，当奇偶校验为奇数时它的值为1。类似地，每一列最下面的二进制变量值当列中的个数为奇数时它的值为1，当列中1的个数为偶数时它的值为0。这些计算变量的值称为接收数据的校正值。

由于具有奇数奇偶性的唯一行是数据字2，而且具有奇数奇偶性的唯一列是第1列，因此通过对字2和计算后的列奇偶校验执行按位XOR操作可以实现纠错：

		综合列
received word0	0011	0
received word1	1001	0
received word2	1101	1
received word3	0000	0
received word4	0101	0
syndrome row	0010	

很明显，任何1位错误都可以被校正，因此能够相互区别开来，进一步，若接收数据恰有两位发生错误，则有两行为奇校验或两列为奇校验，或者两种情况同时出现。举出说明这些情况的例子。因此，可以从所有区别出所有2位错误。但是，并非所有2位错误都可以彼此区别。你能找出两个2位错误导致同一校正表的例子吗？若多于两位接收时出错，有时可以检测出，有时则检测不出，这取决于有多少位出现错误以及错误位的位置。找出产生相同校正表的两个3位错误，找出产生的与无错时相同的校正表的一个4位错误。

错误字2	1101
计算后的列校验（综合行）	0010
校正后的字	1111

设计一个块奇偶校验分析程序。图8-56a给出了块奇偶校验分析程序框图。DAT为4位输入向量。SOB是持续一个时钟周期的块级的开始信号。EOB是持续一个时钟周期的块级结束信号。所有输入均由系统时钟同步。如果发生了一位错误，则四位输出向量EWRD指出出错的字。若字2有错，则EWRD=0010。EDAT为接收到的出错字。CDAT为纠错后的四位向量。SE为表示只有一处发生错误的信号。若检测到多个错误，则ME=1。当多于一个字为奇校验或多于一列为奇校验时，表明检测到多个错误。DAV=1表示其他的输出包含有效数据。除了DAV=1时的单时钟周期，所有输出都是未指定的。图8-56中的时序图表示一个三字序列的时间关系。时钟序列的三个点表示需要未知长度的时钟周期来计算输出信息。在DAV脉冲的一个时钟周期后，电路返回到等待另一个SOB脉冲的状态。内部寄存器应对另一次计算而初始化。SOB脉冲可以出现在每个DAV=1之后的下一个时钟周期内。

a) 为奇偶块校验分析程序创建算法级VHDL模型。

b) 仿真算法级模型。

c) 把VHDL模型（手工）转换为硬连线控制单元和数据单元。完整记录转换过程和得到的硬件电路。

d) 讨论自动转换硬件的可能性。

e) 若有可用的硬件仿真程序，仿真控制和数据单元并与算法级仿真的结果进行比较。

8.25 设计习题8.24描述的奇偶块校验分析程序的微程序控制单元。使用下面的设计步骤：

a) 使用本章描述的综合过程导出奇偶块校验分析程序的BMCU的ROM程序。

b) 画出具有微程序控制单元和数据单元块的奇偶块校验分析程序框图。

c) 为微程序控制单元和数据单元创建一个类似图8-47a和图8-47b的系统级VHDL模型。

d) 仿真系统级VHDL模型，使用可以完全验证功能规格说明的具有不同长度的输入序列选择。

e) 把系统级VHDL模型翻译为硬件。记录翻译过程并讨论自动翻译的可能性。若有可用的自动综合工具，则进行综合。

f) 若有可用的硬件仿真程序，仿真上一步产生的硬件。使用在系统级VHDL模型验证中

用到的输入序列并比较输出结果。记录并解释结果中的不同。若输出有很多差异，则它们是违反规格说明造成的还是由不同表示引起的？

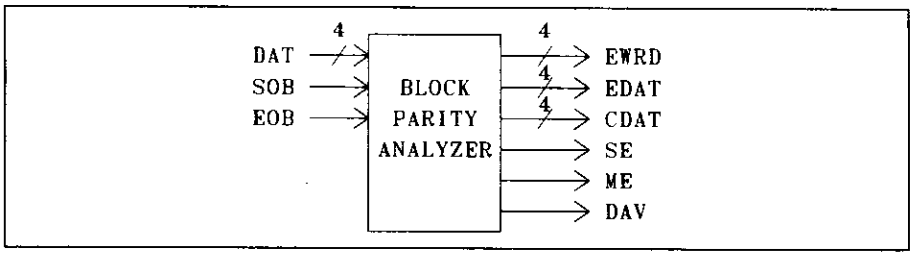


图8-56a　块奇偶校验分析器的框图

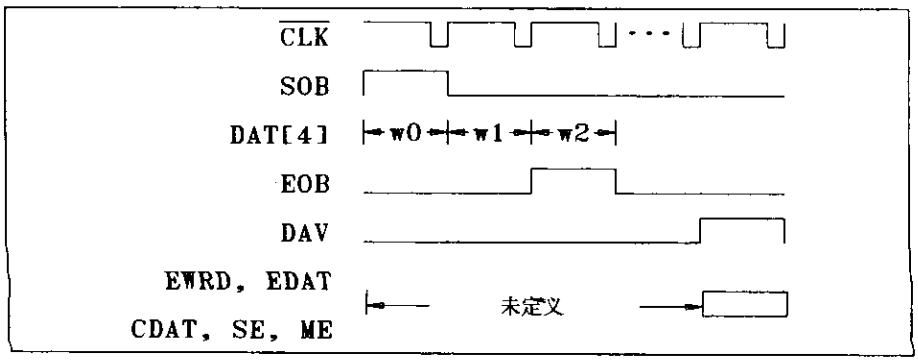


图8-56b　块奇偶校验分析器的时序示例

8.26　考虑图8-46 STOP器件状态5的实现。由于地址产生逻辑受限于二路分支，因此需要两个时钟周期。讨论这种状态5的实现对输入和输出时序的影响。

a) 例如，假设R或A在对应ROM最后一项的第二个分支操作一半时发生变化，这将会导致什么问题？

b) 在哪种环境中“展宽”微操作时钟周期会导致输出时序出现问题？

c) “展宽”时钟周期还会导致什么问题？

d) 对于c中的问题，给出针对STOP器件的可能解决方案。

e) 对于c中的问题，给出可以普遍应用的可能解决方案。

8.27　假设微控制器具有一个8位条件选择字段，列出一个类似图8-50的表，给出可能的条件数，以及对所有可能的*k*-out-of-*n*编码和完全译码字段电路的开销。5-out-of-8编码是否比开销更少的其他*k*-out-of-*n*编码更好？

8.28　对于图8-51和图8-52中用四分支法（4BMCU）定义的微控制器，完成下述任务：

a) 画出类似图8-33中BMCU的控制器框图。

b) 为4BMCU创建一个算法级VHDL描述，它可作为设计这种类型的控制单元模板。提示：图8-38和图8-39的BMCU可以通用。

c) 为4BMCU设计地址产生逻辑。

8.29　使用4BMCU来替换BMCU，决定本书中描述的STOP器件中微控制器的ROM内容。

8.30　用4BMCU替换BMCU，重做习题8.17。

8.31 用4BMCU替换BMCU，重做习题8.25。

8.32 下面是Moore有限状态机的状态表，它执行对输入X的串行奇偶校验，表的下面是VHDL算法级模型的部分代码。通过下列方式完成这些代码（a）填充表示状态表的数组的聚合体。（b）写出模型的可执行语句。假设R=“1”时状态机被异步重置为状态SO。

	输 入		输 出
当前状态	X=0	X=1	Z
S0	S0	S1	0
S1	S1	S0	1
	下一状态		

```
entity MOORE is
  port(CLK,R,X: in BIT; Z: out BIT);
end MOORE;

architecture ALG of MOORE is
  type STATE is (S0,S1);
  signal FSM_STATE: STATE := S0;
  type TRANSITION is record
    OUTPUT: BIT;
    NEXT_STATE: STATE;
  end record;
  type TRANSITION_MATRIX is array(STATE,BIT)
       of TRANSITION;
  constant STATE_TRANS: TRANSITION_MATRIX :=

  --Insert state table here.

begin

  --Insert code here.

end ALG;
```

8.33 图8-26是Mealy有限状态机的简要算法描述格式。尽管Moore有限状态机可以使用相同的存储格式，但是可以修改此格式，通过使用独立数组来存储下一状态和输出值，使数据结构更加有效。说明如何用更有效的数据结构实现下面的Moore机。

	X		
当前状态	0	1	Z
A	B	D	0
B	C	B	0
C	B	A	0
D	B	C	0
	下一状态		

8.34 实现Mealy有限状态机的算法模型，状态机的状态表如下：

	输 入			
当前状态	0	1	2	3
S1	S2,1	S4,0	S2,1	S3,1
S2	S1,0	S3,1	S3,1	S3,0
S3	S1,1	S3,1	S2,0	S5,0

（续）

	输　入			
当前状态	0	1	2	3
S4	S2,1	S1,0	S2,1	S2,1
S5	S2,1	S1,0	S2,1	S3,1
	下一状态，输出			

模型中应包括：1) 表示状态的枚举数据类型；2) 表示状态转换和输出信息的记录类型；3) 对表进行编码的数组聚合体。下面给出了描述中使用的程序包以及有限状态机的实体声明，它可以用于创建构架。

```
package INT_4 is
  type INT4 is range 0 to 3;
end INT_4;
use work.INT_4.INT4;
entity  FSM is
  port(CLK,R: in BIT; X: in INT4;
       Z: out BIT);
end FSM;
```

仿真一个最少漫游序列的应用，即：使所有状态均被至少访问一次的最小长度输入序列。

8.35　考虑消除图8-33所示的微程序控制单元中的MAR。进行时序分析并讨论这样做的优缺点，采用8.3.2节中对MIR的类似做法。

第9章 ASIC及ASIC设计过程

本章介绍什么是专用集成电路（Application Specific Integrated Circuit，ASIC）以及实现ASIC的不同技术。然后，给出一个ASIC设计过程的一般描述，重点是详细介绍建模和综合。我们介绍如何使用Synopsys的综合工具，结合标准单元库来设计ASIC，以及如何应用Xilinx的工具使用现场可编程门阵列（FPGA）实现ASIC。

9.1 什么是ASIC

在20世纪60年代和70年代，设计是通过标准元件进行的，其定义如下：

标准元件：大量生产的适合大范围应用的元件。

在这些年代，标准元件的典型代表是7400 TTL系列元件，该系列首先由德州仪器公司开发，随后其他厂商亦进行了开发。该系列首先包含小规模集成电路（SSI）元件，例如：门和触发器。很快，中规模集成电路（MSI）元件又加入了该TTL系列，如译码器、多路器和寄存器。SSI和MSI芯片置于印制电路板上，而且通过导电金属互连。在70年代早期，集成电路技术的发展使得可以开发大规模集成电路。一些公司，如Intel和Motorola，开发了单片微处理器和单片RAM和ROM存储器。其他的支持芯片，如并行端口、串行接口（UART）和中断控制器也很快开发出来。可用于微型计算机设计的LSI新系列也被开发出来。注意，采用芯片系列的设计是自底向上设计方法的最有力的例子（见第1章的讨论）。

然而，这里仍存在很多问题。在许多应用中，微处理器速度显得太慢，所以，随机逻辑中的很多部分仍然必须用TTL元件来完成。而且，印制电路的互连方式开销较大，焊接的连线又引起了可靠性问题。主要的半导体厂商都具有将这种随机逻辑集成到芯片内部的技术，但小批量生产的价格过高又成为抑制因素。工业上的两个发展改变了这种情况。首先，开发出了硅铸造（silicon foundries）方法，对于以标准形式提供的集成电路设计，可以进行小批量生产，并且价格可以接受。第二，开发出了门阵列，正如我们即将讨论的，门阵列包含成行的预制晶体管。用户可以将代表金属互连的文件送入门阵列制造处（另一个硅铸造），添加其金属互连以完成芯片。然而，在晶体管一级进行设计是很乏味的。这样，公司开始开发预先布局的单元库，对应于SSI和MSI组件。这允许设计者能够在更高抽象级别之上开发设计并提交硅铸造的文件。最后，为了支持这个过程，还开发出了许多CAD工具。原理图获取加快了结构化模型的输入速度。在80年代晚期，开发出了硬件描述语言，如VHDL和Verilog，提供设计的行为级输入方式。高速仿真程序和仿真引擎使得设计模型验证速度很快。自动化的检查和分析帮助在设计过程中可以发现并改正错误。最后，综合工具自动将行为级的HDL模型转化为结构级的逻辑模型。

所有这些开发都支持面向小市场的专用集成电路的设计。

专用集成电路（ASIC）：为某个专门应用制造的集成电路，一般生产量较小。

这个定义与技术无关，这样，正如我们所见，实现ASIC可使用一系列技术：PLD、门阵列、FPGA、标准单元和定制。然而，与计算机领域的其他类属定义，如RAM一样，ASIC所

代表的是一个更为专业化的意思。PLD和FPGA被称作可编程逻辑，而ASIC则指标准单元和门阵列，其芯片实际上是制造而不是在用户端进行编程。这样，工程师可以说“我不确定是使用FPGA还是ASIC”。定制的方法专指标准部件的设计。

专用ASIC和专用可编程逻辑的应用范围相对狭小，如数字信号处理、马达控制和网络布线。通用计算机的随机逻辑可以用标准单元技术置于一块芯片上。而且ASIC也不总是少量出售。上述的计算机芯片不可能小批量发售。而且，为专门用途开发的DSP芯片是通用的并且做为标准元件出售。ASIC（和可编程逻辑）的精华在于该设计方法使得设计可以处于用户的控制之下。

最近将ASIC推向一个新高度的是“智能产权”（Intellectual Property，IP）概念的出现。IP指一个公司出售给另一个公司的设计。设计用HDL模型表示，包括测试程序包，还包括综合的限制或控制综合的脚本文件。IP可以是RAM存储器或微处理器。代表了ASIC库开发的一个新水平，它对产品设计将会有深刻的影响。

ASIC和可编程逻辑的出现给系统设计人员提供了三个设计范例，可在以后加入系统设计之中。

1) 冯·诺依曼机器构架：CPU、存储器和I/O。设计者将系统需求转化为该机器的编程代码。

2) ASIC：针对要求高速的固定逻辑。

3) FPGA：针对可重新配置的逻辑，允许功能的快速转变。

在一个设计之中使用这三个范例的混合是一个令人激动的前景。一个IP处理器可以放在和ASIC相同的芯片上，这个芯片和FPGA芯片可以在同一基底之上结合，从而形成多芯片模块（MCM）。

9.2 ASIC电路技术

现在几乎所有的ASIC和可编程逻辑芯片都使用CMOS技术。

图9-1给出了一个CMOS反相器。当IN为低时NMOS晶体管（下拉并接地的那一个元件）是关的，PMOS晶体管（上拉并接入V_{cc}的那个元件）是开的，它们使得OUT=+5。当IN为高时，NMOS晶体管是开的，PMOS晶体管是关的，因为从V_{cc}到地阻抗很大，使得OUT=GND。静态的直流功率很小。

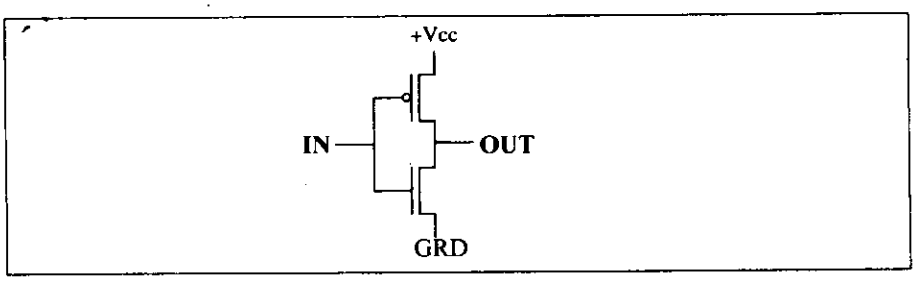


图9-1　CMOS反相器

其他的CMOS电路，如图9-2中所示的NOR门，静态功耗也低，这是因为上拉电路和下拉电路中总有一个是关闭的。然而，当电路打开或关闭时，有很短的一段时间上拉电路和下拉电路都开的，并且从V_{cc}到地之间产生一段电流尖峰。动态（AC）功耗由下述等式给出：

$$P_{AC} = CV^2_{CC}f \tag{9-1}$$

其中C是负载电容，V_{CC}是源电压，f是开关频率。尽管CMOS的AC功率高于CMOS的DC功率，但是相对于其他逻辑系列，它的功率仍然很低。

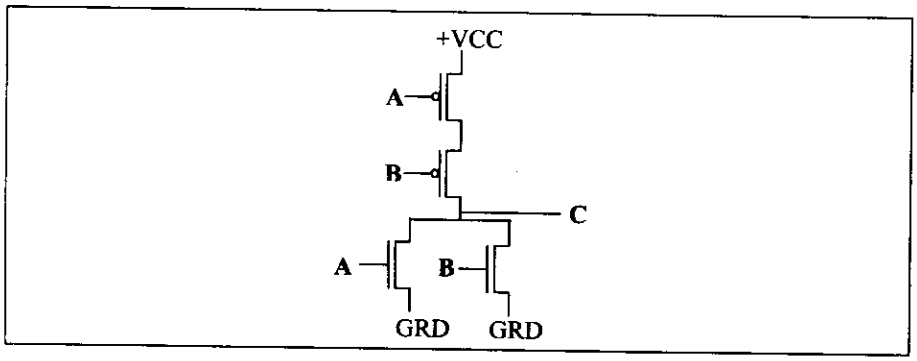


图9-2 CMOS的NOR门

CMOS技术的发展已经将CMOS电路的特征尺寸减小到0.15μm。小的特征尺寸和低功耗的结合，允许了高速、低功耗、高集成度的复杂电路的制造，从而成为VLSI技术的基础。

CMOS开关

我们考虑的系统不只是使用了反相器和门，还包括简单的晶体管开关（也称作通路晶体管pass transistors）。考虑这些开关在传递数据值时的效率是很重要的。

对于图9-3中所示的NMOS开关，如果V_{IN}=5V，输出将为弱1（IEEE逻辑系统中的H）。如果V_{IN}=GND，输出为强0，对于PMOS器件，5V的输入产生强1输出，而GND输入产生弱0（L）输出。这样，单独使用任何一种开关都会产生信号衰减，尤其当开关级联的时候。

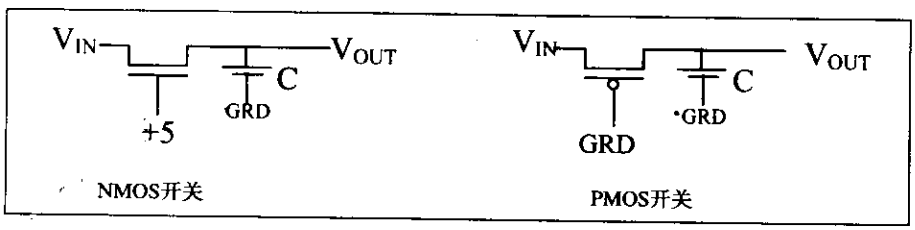


图9-3 CMOS开关

图9-4给出了一个改正这个缺陷的“好”开关。因为并行排列，并且两个开关都是打开的，NMOS开关传递“好”0，PMOS传递“好”1。然而，对每个开关需要两个晶体管，正如在FPGA中使用的那样，互连结构需要上千个开关，这时高价格将成为限制因素。

9.3 ASIC的类型

本节介绍进行ASIC和可编程逻辑设计的不同方法：PLD、FPGA、门阵列、标准单元和定制。

9.3.1 可编程逻辑器件

可编程逻辑器件（PLD）具有图9-5中所示的形式。输入变量，如I1~I8及它们的非，以列

的形式纵贯芯片。这些垂直列可以和水平列相连，形成输入变量和它们的“非”的“与”项的子集。八个这样的“与”项作为一个或门的输入。有九个或门，每个或门有八个“与”项输入。或门的输出连到D触发器的输出或直接连到输出引脚之上。反馈直接来自或门或者来自触发器的输出。所有的输出都被缓存。复杂PLD（CPLD）包括几个用开关互连的PLD。

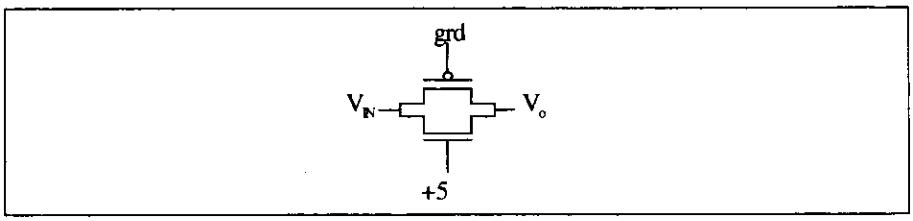


图9-4 好的CMOS开关

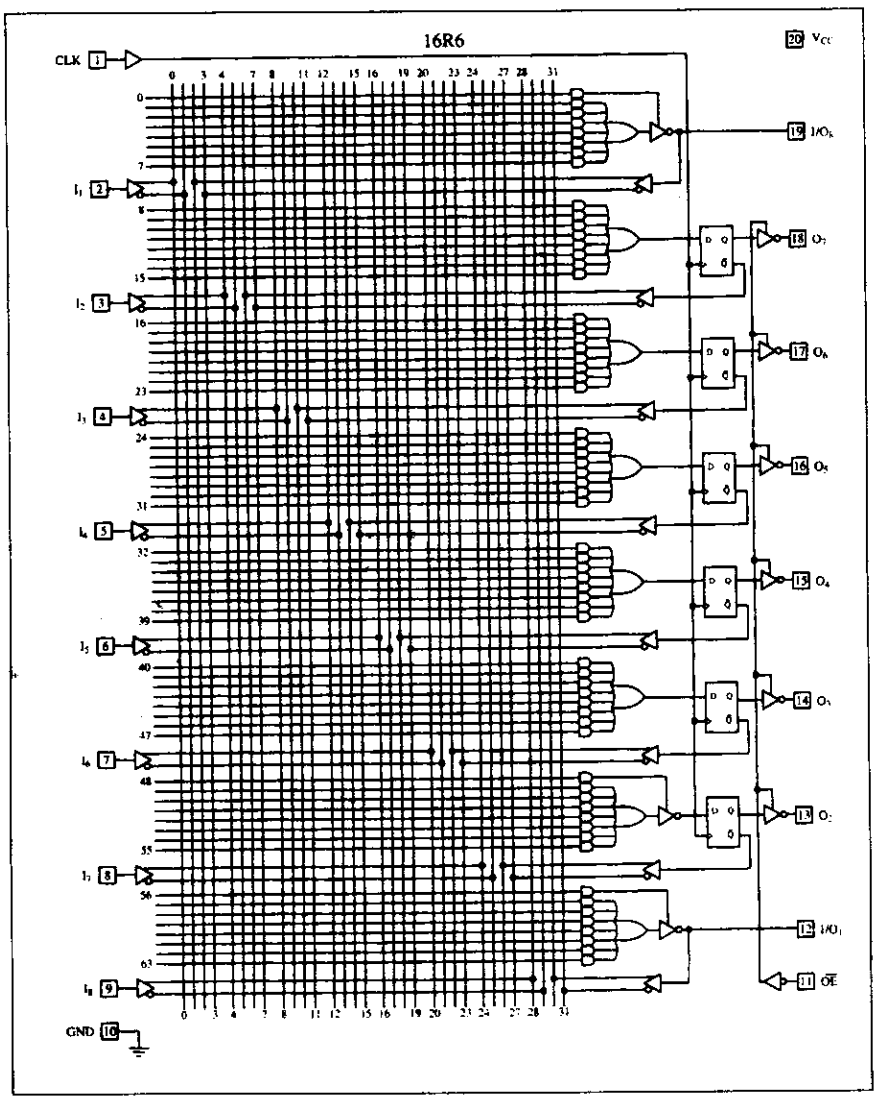


图9-5 可编程逻辑器件

PLD在实现使用D触发器的复杂组合逻辑函数的逻辑时尤为高效，如状态机。然而，许多系统具有触发器对门的比率较高的逻辑，对于这些系统，现场可编程门阵列（FPGA）则更加合适。

9.3.2 现场可编程门阵列

现场编程门阵列（Field Programmable Gate Array，FPGA）是掩模编程门阵列（Mask Programmable Array，MPGA）的扩展。通常称之为门阵列。门阵列由NMOS和PMOS单元列及在单元内连接这些晶体管的门和单元之间的布线信号组成。互连是在“硅铸造”中使用掩模来沉积金属层互连而实现的。一个FPGA也包含单元阵列，但其单元更加复杂，例如，查找表或多路器。这些元件互连在一起，在用户那里都是要编程的，从而提供一种实现逻辑的快速而经济的方法。

图9-6给出了一个以2D单元阵列组织的FPGA的主要元件：1) 可配置逻辑块的短形阵列CLB可实现多种逻辑功能。2) 用在单元之间布线信号的连线线段。3) Xbar开关，用来连接水平线和垂直线。4) 输入/输出（I/O）块，用来在芯片输入、输出引脚调节信号。所有这些资源都是可编程的并可能在用户处进行重新编程。

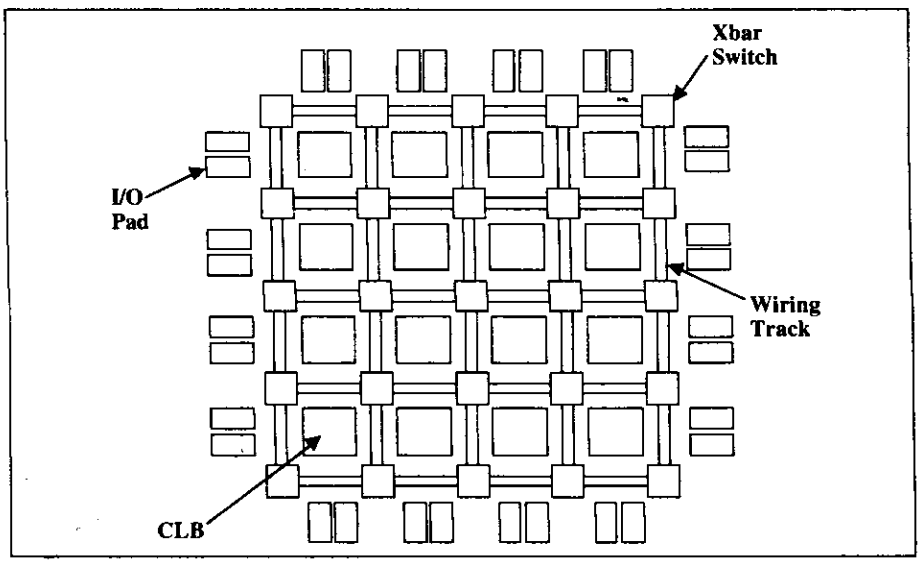


图9-6 FPGA单元的2D阵列

图9-7示例了模块构架的“海”。用硅表面之上的三层金属进行布线。该构架用在Actel公司的FPGA上。单元被紧密封装，并很大程度上使用了本地布线。

1. 可配置逻辑块（CLB）

选择逻辑块的一个主要考虑是其通用性。假设想实现表9-1中的逻辑函数。

表9-1 三输入逻辑函数

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

（续）

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

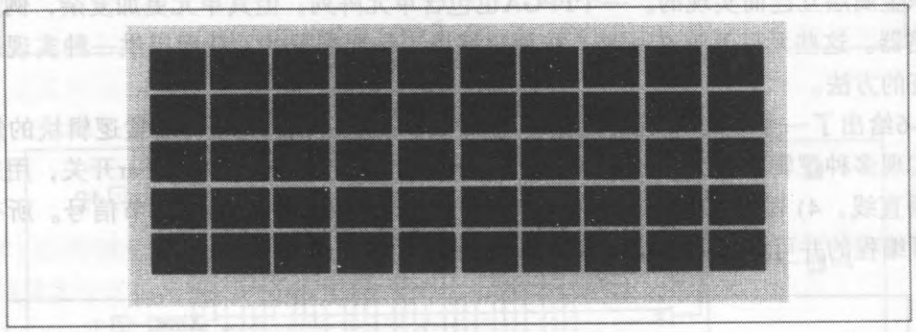


图9-7　模块结构的“海”

图9-8给出了使用通用逻辑单元实现该函数的两种可能方法。

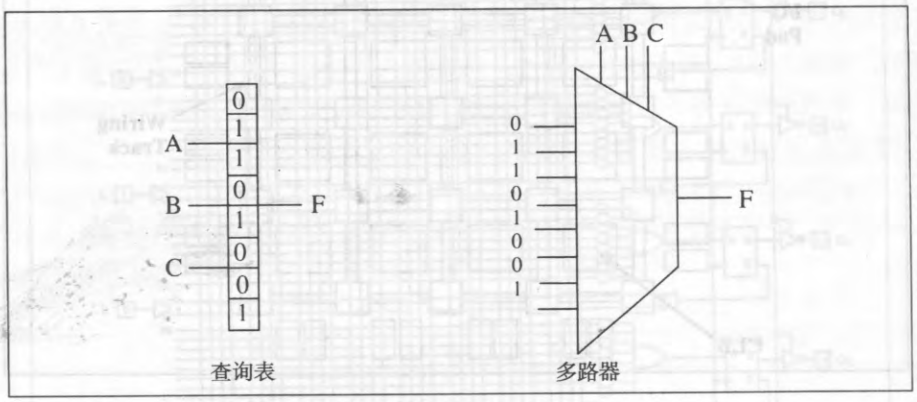


图9-8　通用逻辑单元

左边是一个查找表（LUT）实现。A、B和C是地址输入。一个特定的地址组合选择对应于该输入组合的相应输出。查找表是用RAM实现的。图9-8中的LUT表可以用8 × 1的RAM实现。右边是多路器实现，F的值连到8路多路器的数据输出。A、B、C连入多路器的选择输入。通常，一个n输入LUT可以实现n个变量的任何函数。对于常量输入，MUX保持不变。然而，如果第n+1个变量和它的非可以得到，则带有n个选择输入的多路器可以实现n+1个变量的任意函数。见8.1.1节MUX设计的详细例子。

一个基本问题是：n的理想值是多少？一个趋势是尽可能大，希望用一个单元覆盖复杂的操作。然而，经验表明，对于查找表，n=4是最优的，而对于MUX，则n=3时最优。对于更大

的值，LUT的许多输入没有被使用，而且浪费了大量的逻辑。由Aigotronix 实现的一个较早的FPGA使用了n=2的查找表，即可以产生两个变量的所有16个函数。然而，对于这样小的单元，尽管单元逻辑的利用率高，但即使是实现简单的函数也必须将许多单元连接起来。互连所需的多余空间和更多的延时反而不令人满意。

尽管可以使用带反馈的组合逻辑来实现时序元件，但这也是一种笨拙的方法。这样，可配置逻辑块（CLB）通常由触发器存储器组成，也包括用于单元内部数据布线的多路器。图9-9中给出了一个这样的组织。在这种情况下，CLB用来实现JK触发器，A和B分别是输入J和K。通过将其LUT的控制输出置为1，触发器从最左边的MUX反馈回LUT的输入C′。最右下位置的多路器将触发器的复位端（R）与单元输入C相连。与之相似，单元输出X通过右上的那个多路器连接到触发器的输出。

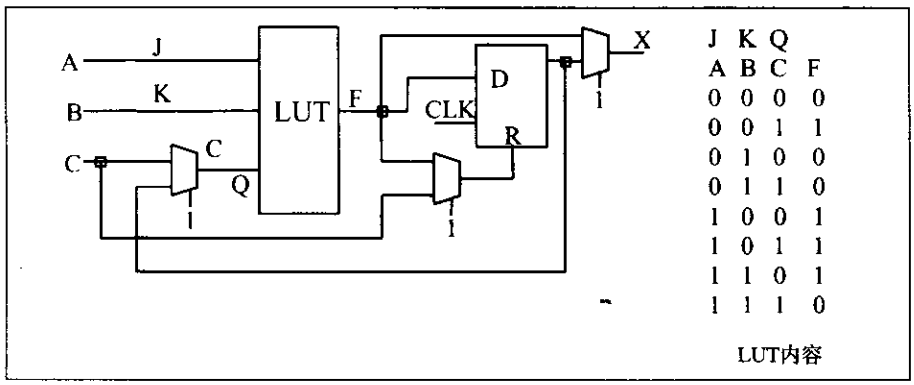


图9-9 带触发器存储的CLB

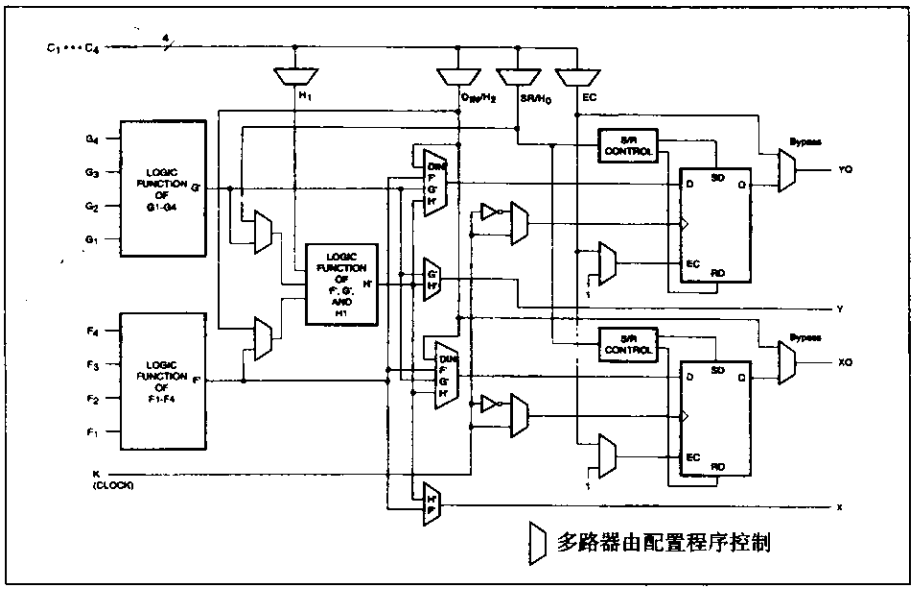


图9-10 Xilinx 4000系列的CLB

图9-10给出了Xilinx 4000系列FPGA的CLB图。它包括两个4输入LUT和一个3输入LUT，

可以独立编程或组合起来编程。它还包括多个多路器用于数据选择、内部单元布线及两个触发器存储元件。4000系列CLB可以实现：

- 任何两个4变量函数和任意3变量函数——都包括不同的输入变量。
- 任何5变量函数。
- 任何4变量函数，同时有一些6变量函数。
- 一些9变量的函数。

两个触发器或存储元件：

- 可以配置为锁存器或边缘触发器。
- 有一个共同时钟CLOCK和时钟使能（CE），对于每个存储元件，该公共时钟都是可反相的。
- 为全局复位控制，由编程来决定是置位还是复位以响应它。（图中并未标出全局复位。）

注意　有多个多路器控制LUT和触发器的输入，用来选择单元输出并将数据布线穿过单元。

图9-11给出了Actel SX系列的FPGA逻辑单元。R单元是时序存储单元。D触发器可以用硬连线时钟（hardwired clock）或布线时钟（routed clock）作为时钟输入，而且时钟的极性是可选择的。触发器有一个异步复位和置位。可以由最近的邻居或布线数据预置。C单元是组合逻辑单元。可以实现所有的3变量函数（注意反相器的存在）以及一些4变量和5变量函数。

2. 互连

开关技术。互连问题本质上包括将两个导体连一起。主要使用两种方法：多次编程和一次性编程。

图9-12给出了可编程互连节点（Programmable Interconnection Point，PIP）。方框表示一个用来控制开关状态的RAM位置。因为RAM的内容可以很快改变，开关同样可快速重新编程。EPROM和EEPROM技术也可用在可重复编程的开关上，但它们重新编程的时间较长，使用EPROM的芯片必须先从板上取下来，擦除后重新编程。EEPROM的开关可以在原位置进行编程，但重新编程时间与PIP相比较长。使用SRAM技术的FPGA可以在毫秒之内重新编程。然而，从前面的讨论中，我们能够知道CMOS开关衰减信号的强度。这种衰减可以用RC电路的形式进行建模，使信号增加负载和延时，如图9-13所示。信号衰减可以通过在不同的点缓存信号来消除，但这将顺次增加延时。

一种重要的互连开关类型是如图9-14中所示的“逆熔丝”。两个导体用一段表现为高阻抗的绝缘材料分开。然而，如果在绝缘体上加上足够大的电压，该绝缘体将被击穿，电流流过，于是在两个导体之间形成一个永久的低阻抗连接。此过程是不可逆转的，对于逆熔丝，有两种主要的商业实现，Plice逆熔丝是在多晶硅和n+扩散区之间的一层绝缘体，Vialink逆熔丝是两层金属之间的无定形硅。

当选择一种开关技术的时候，对各技术的下述方面进行比较是很重要的：

1) 开关的R和C值，串联的三个开关的延时模型见图9-13。

2) 可重复编程性：如果想很容易改变设计或改变逻辑函数，该特点很重要。

3) 易失性：当掉电时会发生什么?功能恢复是否容易?

4) 开关的面积：更多的开关使布线更容易；少的开关使布线困难，甚至在某些情况下不可能进行布线。

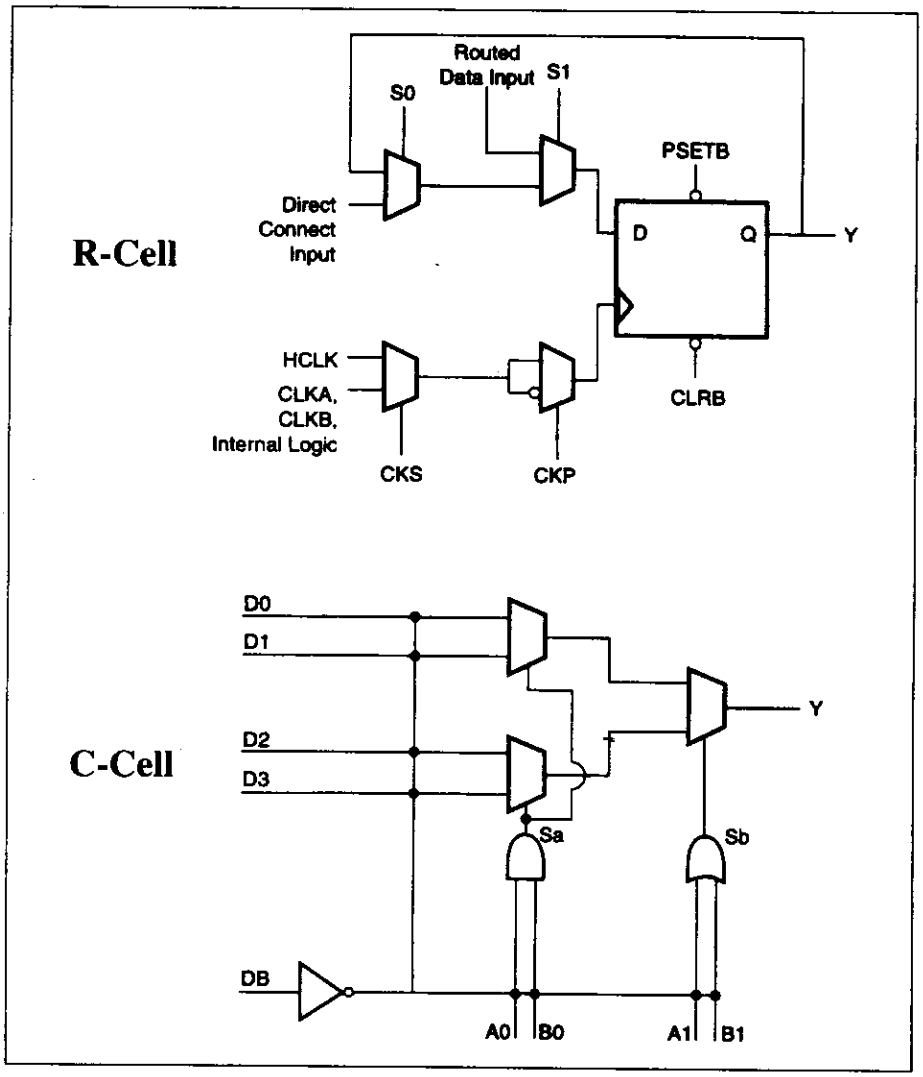


图9-11 Actel单元

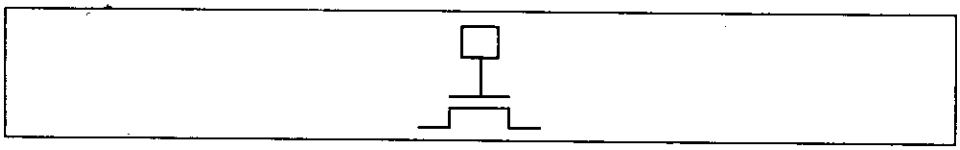


图9-12 可编程互连节点（PIP）

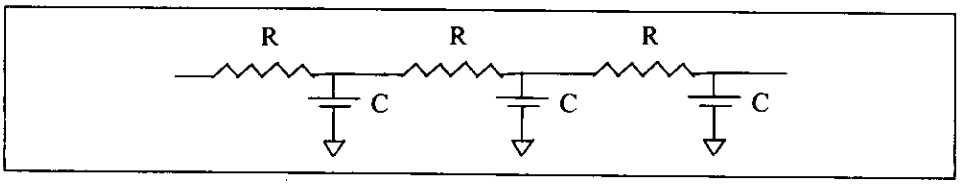


图9-13 开关延时模型

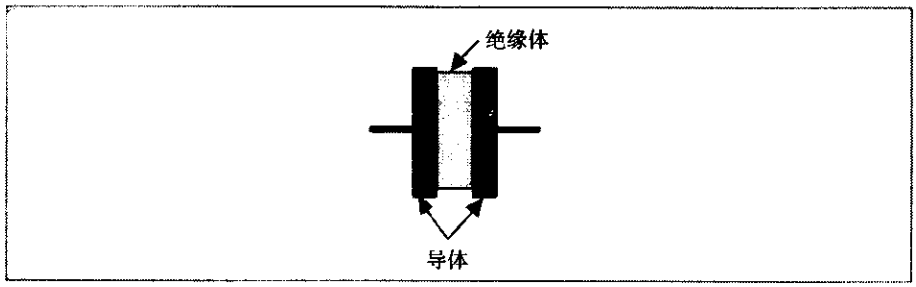


图9-14 逆熔丝

表9-2 编程技术的特性

编程技术	易失性	可重复编程特性	芯片面积	R(ohm)	C(ff)
SRAM	是	在电路中	大	1~2K	10~20
Plice逆熔丝	否	否	小的逆熔丝大的Large prog. trans.	300~500	3~5
ViaLink逆熔丝	否	否	小的逆熔丝大的prog. trans.	50~80	1.3ff
EPROM	否	在电路外	小	2~4K	10~20ff
EEPROM	否	在电路中	2×EPROM	2~4K	10~20ff

表9-2中对不同的技术进行了比较。两个最普遍的选择是SRAM和逆熔丝。SRAM是易失性的，但很容易重新编程，因为RC时间常数相对较高，引起相对慢的互连。逆熔丝是非易失性的，不可被重新编程，但在所有的选择中，该技术的RC常数最小，表现为它们支持的时钟频率最高。Actel使用Plice逆熔丝。然而Actel 逻辑单元比Xilinx逻辑单元小，这样，对于Xilinx需要更多的布线。于是，时钟频率的优势并不明显。当前人们所使用的系统里，在易失性很危险的系统应用中，Actel FPGA比较先进。Xilinx FPGA是可重新配置系统的明确选择。

互连结构。图9-6给出了以单元的2D阵列组织的FPGA的互连结构，这种阵列的互连机制包括使用布线通道和Xbar开关来连接CLB。我们将介绍Xilinx 4000系列FPGA的互连机制。图9-15给出了到布线通道的连接是如何实现的。

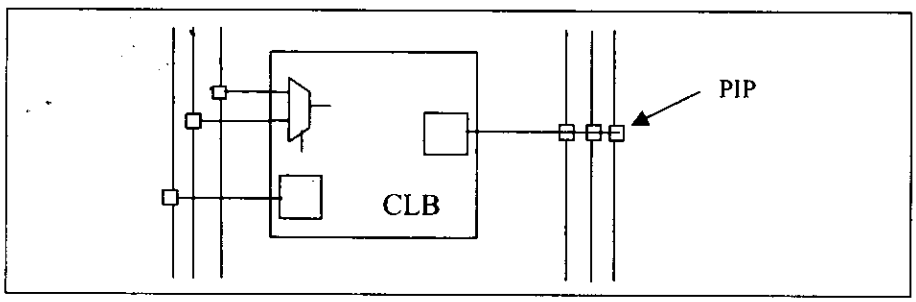


图9-15 CLB的布线通道连接

PSP（见图9-12）用来将CLB的输入和输出与线相连。在某些情况下，CLB内部的多路器从线输入中选择一些接入。图9-15只给出了到垂直线的连接。实际上，Xilinx的 CLB在四侧都有输出并可与水平线建立连接。

图9-16说明了Xbar是如何构建的。左边是Xbar开关节点。开关中的每条虚线代表一个PIP。右边是4×4的交叉开关（crossbar）。每个水平或垂直输入可以直接穿过，或者向左、向右转

向进行连接。一个输入同样也可连入多个输出线之上。开关是双向的。开关硬件很昂贵。一个N×N的交叉开关需要6N个晶体管。

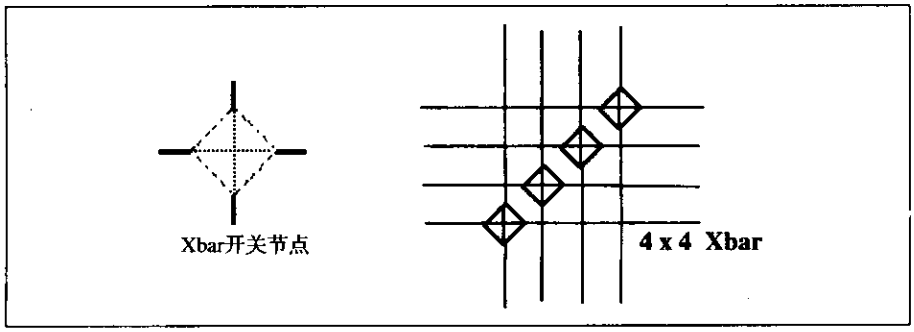


图9-16 Xbar

这样就能够从一个CLB连接到另一个CLB。然而，还必须考虑开关延时。在图9-6中，如果右下脚的CLB连到右上脚的CLB，将会遇到7个PIP延时，5个延时通过crossbar开关PIP，两个PIP延时分别对应于CLB的输出和输入连接。图9-6中的简单FPGA是CLB的4×4阵列。Xilinx XC 4010XL是CLB的20×20阵列，因而其布线延时将会更加糟糕。为了解决这个问题，Xilinx已经开发了在布线通道内线类型的层次：

1) 单长度线在其传输方向上通过每个开关矩阵。(图9-6中包含了这种类型的线)。这些线不适合于长连接，但对于短的局部连接效果极好。

2) 双长度线在进入一个开关矩阵之前要经过两个CLB。

3) 四长度线在进入一个开关矩阵之前要经过四个CLB，这样双长度线和四长度线与单长度线相比，在连接两个相隔较远的CLB时延时更短。

4) 直接连线——北、南、东和西——与最近的相邻CLB相连。这是一种不经过开关的互连，所以是最快的局部互连。

5) 长线贯穿整个阵列的长度或宽度，用于高扇出、时间关键网络或分布距离较大的电路。

FPGA布线程序的一个功能是为一个信号传送选择最好的布线类型。

图9-7给出了Actel构架为“模块海”的模块构架。这里没有展示它的互连结构。在Actel SX系列的FPGA中，互连与单元层分开并通过三层金属进行处理。图9-17给出了这种分层结构。逆熔丝在金属层2和3之间。该结构没有专门的布线通道，但使用了芯片的整个表面面积。

图9-18给出了该布线方法的细节。R和C单元组成簇(cluster)(用阴影区域表示)。互连类型的层次划分为：

1) 直接互连：无开关连接，用于簇内布线。

2) 快速互连：启用相邻簇的水平和垂直布线。这种连接方式引起一个逆熔丝延时。

3) 分段布线：规定了簇之间不同的通道长度。延时一般是两个逆熔丝和五个逆熔丝中的最大值。

4) 高驱动布线：这个结构提供三个时钟电路。第一个时钟HCLK在每个R单元中从中央的HCLK缓冲器硬连线连接到时钟选择MUX。其他两个时钟(CLKA、CLKB)是全局时钟，可以来自外部引脚或阵列中的内部逻辑。在每个R单元中可以进行选择。

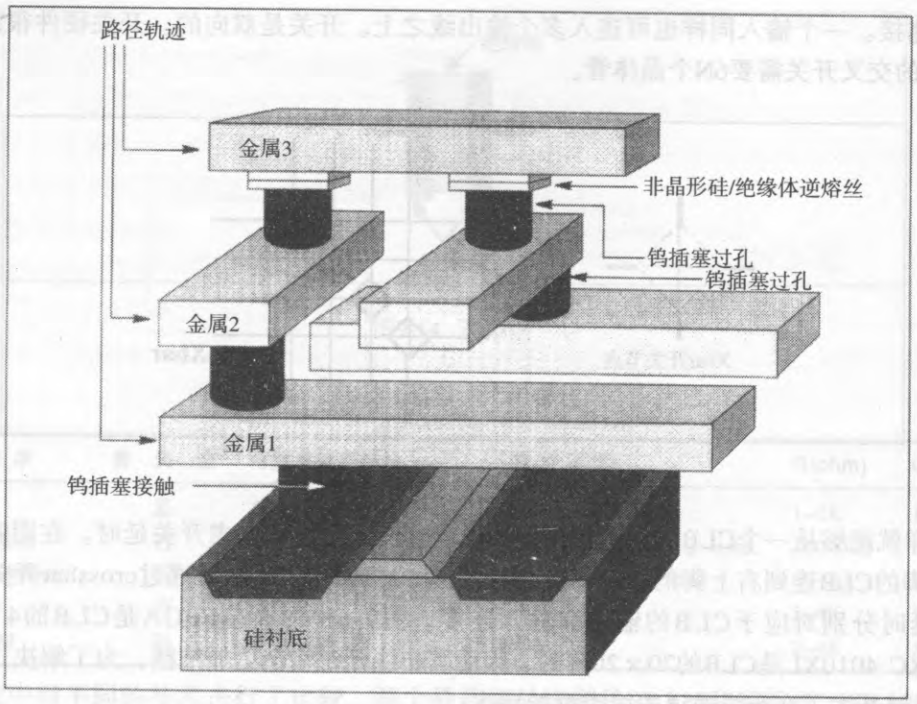


图9-17 Actel多层互连

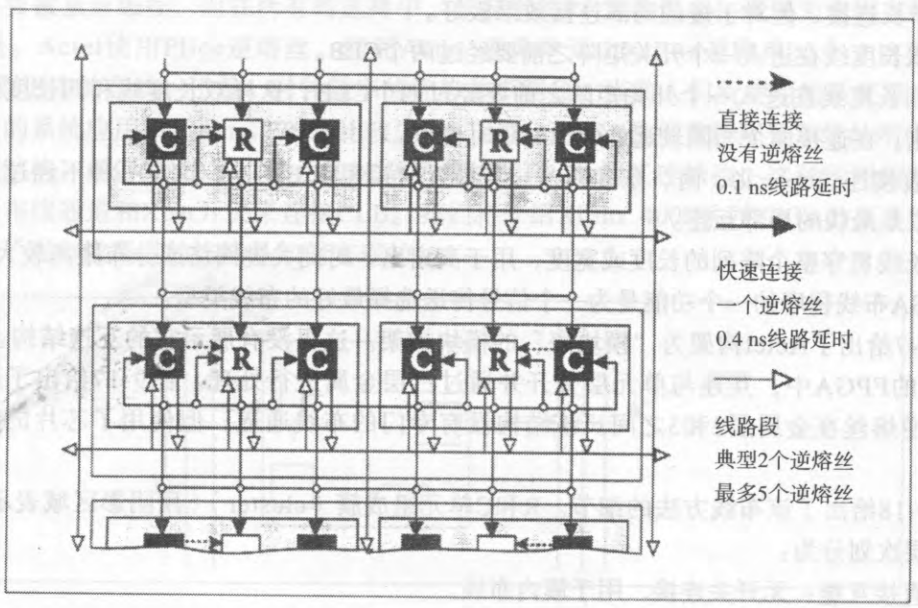


图9-18 ACTEL布线层次

3. 输入/输出块（IOB）

FPGA的外围是输入/输出块（IOB），图9-19给出了Xilinx 4000系列FPAG的IOB，它们具有如下特征：

1) 输入（经缓冲）可直接输入或经由时钟触发器输入。可在f/f输入中插入延时以增加建立时间（setup time），确保其保持时间（hold time）为正。

2) 输入可送入“快速捕获”（fast capture）锁存器，然后再输入时钟触发器。

3) 输出（经缓冲）可直接输出或由时钟触发器输出。

4) 输出扭曲率可控，可以是TTL或CMOS三态缓冲。当许多输出同时进行切换时，扭曲率控制用来使噪声耦合最小。

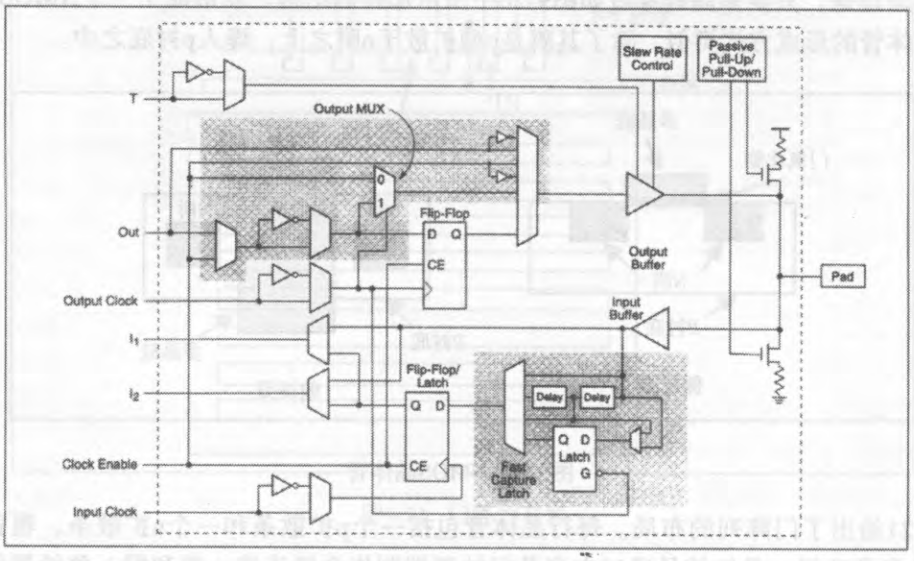


图9-19 Xilinx 4000系列的I/O块

Actel的FPGA同样也包括IOB，可以配置为输入、输出、三态输出或双向引脚。它不包含触发器存储，但可将一般的R单元用于此目的。

4. 专门逻辑

某些类型的逻辑不可能用FPGA的CLB实现，或者实现效率极低，例如：1) 总线三态系统。2) 译码器。3) 晶振。4) 快速加法器逻辑。5) RAM单元。Xilinx 4000系列的FPGA有用来实现线与和多路总线的专门电路。在芯片的边缘是4个可编程译码器。在板上的晶振提供8MHz、500KHz、16KHz、490Hz和15Hz的时钟信号。RAM可利用CLB中的两个4输出查找表作为两个16×1的RAM存储以高效实现RAM。每个CLB都有专门的快速加法器逻辑。

9.3.3 门阵列

门阵列芯片包括预定制的相连的PMOS和NMOS晶体管行。用户创建一个指定金属互连路径文件。门阵列设计可以借助CAD工具将原理图或HDL模型映射为晶体管配置而完成。晶体管中的水平布线通过两种方法实现：1) 晶体管行之间的水平通道之间。2) 使用未被使用的晶体管路径（门海）。现在第二种方法用得比较多，因为它使得门密度高于任何通道布线方法得到的门密度。布线层次方法包括：1) 单层金属；2) 单层金属和触点；3) 双层金属、触点和过孔；4) 三层金属、过孔和触点。设计者创建的文件被送给门阵列制造者（硅铸造），在那里创建一套掩模，用来沉积金属通道。因为有掩模的创建过程，门阵列有时叫作掩模可编程门阵列（MPGA），以与FPGA相区别。这样的MPGA不是用户可编程的，并非属于先前所定义的可编程逻辑范畴，但却是实际的ASIC。MPGA出现于FPGA之前，事实上，FPGA技术源自MPGA技术。MPGA的门密度和时钟速度都比FPGA快，而且是非易失性的。对MPGA设计的

修改非常昂贵和耗时，因为进行新的制造之前必须生成新的掩模。

为了理解门阵列是如何制作的，首先必须理解基本的MOS门制造过程。图9-20给出了NMOS晶体管的布局。两个高浓度的n阱埋入p衬底以形成晶体管的源和漏。在源和漏之间的硅表面沉积了一层氧化物。在氧化物之上沉积了一层多晶硅，形成门连接。这样，晶体管制造的基本原理是，只要多晶硅穿过如图9-20中所示那样的结构，就形成了一个NMOS晶体管。PMOS 晶体管的形成方式类似，除了其阱是p型扩散于n阱之上，埋入p衬底之中。

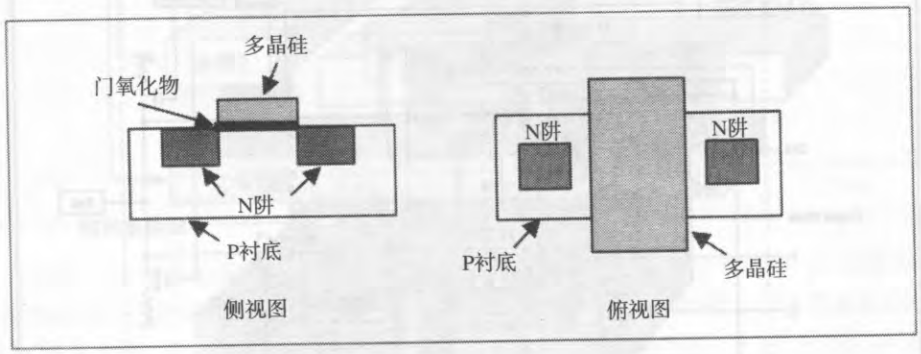


图9-20　NMOS晶体管

图9-21给出了门阵列的布局。每行晶体管包括一个p扩散条和一个n扩散条，覆盖在这些条上的是多晶硅门。晶体管是通过在多晶门的两端制作金属连接（源和漏）并给器件加电实现的。图9-1中反相器的布局见图9-21。

图9-21中所示反向器的线路示意了单元内连线。早期的门阵列使用晶体管行之间的水平通道来实现不同单元间的布线。垂直布线在另一个布线平面上进行。最近的门阵列使用未用的晶体管行来进行水平布线，而并无独立的水平布线通道。与通道布线门阵列相比，晶体管排列得更加紧密。使用这种方法的门阵列被称作门海。门阵列的一个缺点是晶体管都是一个尺寸，因而不可能立即得到更快的门。后面将看到用标准单元进行设计将涉及该问题。

9.3.4　标准单元

目前，大部分ASIC是使用库中的标准单元设计的。生产的芯片一般叫做基于单元的集成电路（Cell-based Integrated Circuits，CBIC）。在设计者一级，库包括不同复杂性的逻辑元件：SSI逻辑、MSI逻辑、数据通道模块、存储器和系统级模块。库包含每个逻辑单元的完整布局。使用者只需与逻辑块描述打交道即可，而不必关心电路布局的细节。

图9-22给出了标准单元布局的例子。所有扩散、接触点、过孔和多晶通道及金属通道都已完全确定。当该单元用于设计时，单元布局电子“粘贴”到芯片布局之上的单元行上。

图9-23给出标准单元互连的布局。这些单元（灰方框）按行进行构造。它们都是等高的，但宽度是根据复杂性变化的。多层金属（这里是两层）用来进行水平垂直布线。这些层次在单元上进行布线，这样就有整个的X-Y平面可用。然而，随着电路的密度增加（而特征尺寸减小），可能会需要三、四、五层的金属互连。为了建造芯片，必须为集成电路制造一整套的掩模。所有的层次都是在硅铸造厂完成的，而对于门阵列，只需要金属层的掩模，并且在硅铸造时只产生金属通道。

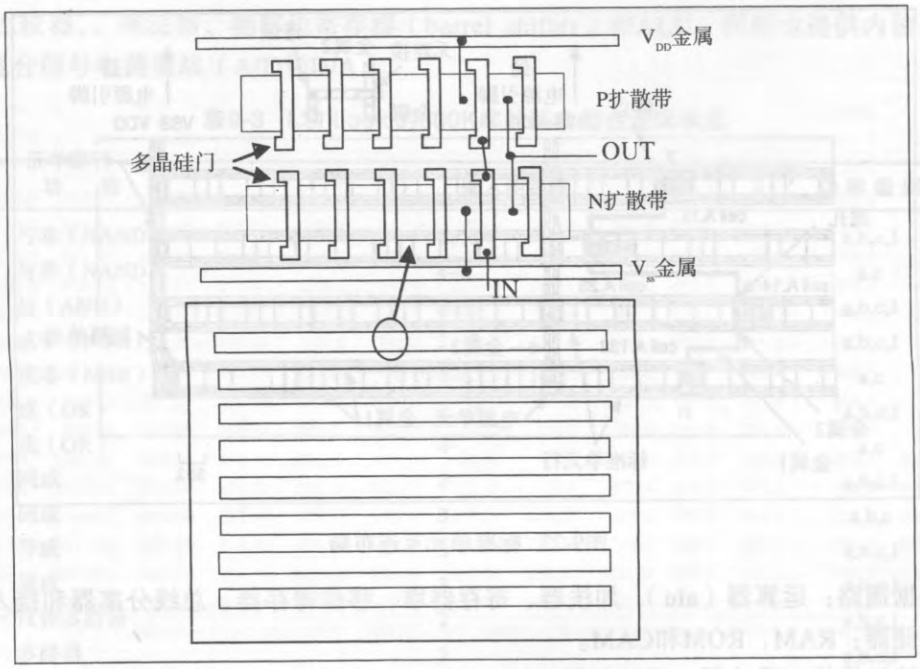


图9-21 门阵列布局

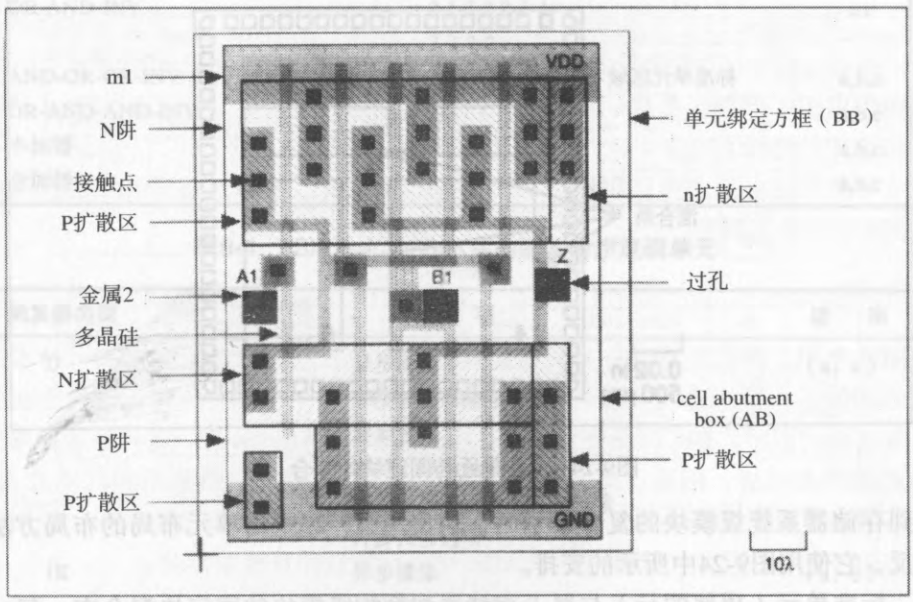


图9-22 标准单元布局

然而，逻辑设计者脱离了这些细节，只需关心逻辑表示，使用原理图获取或HDL作为综合工具的输入，设计者说明使用哪个库元件以及它们如何互连。库元件的类型以复杂性递增为序：

1) SSI逻辑：与非门、或非门、异或门、aoi、oai、反相器、缓冲器和寄存器。

2) MSI逻辑：译码器、编码器、奇偶校验树、加法器和比较器。

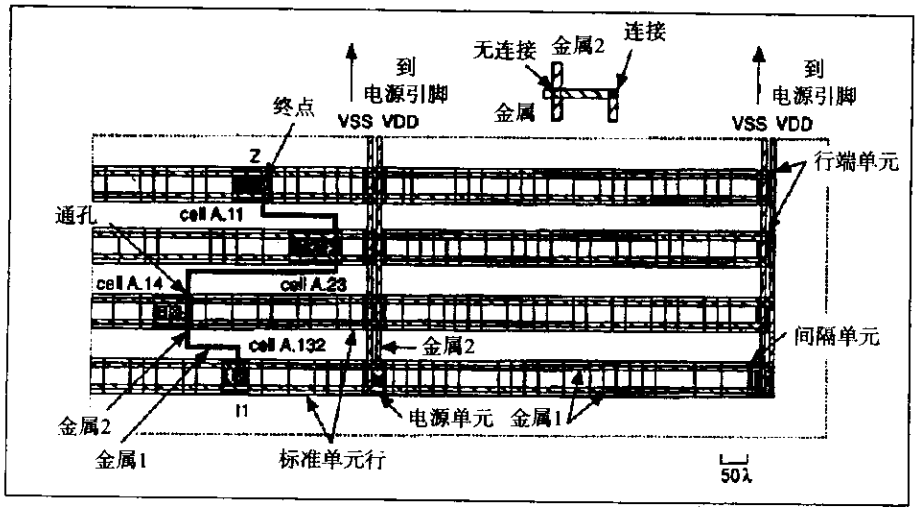


图9-23 标准单元互连布局

3) 数据通路：运算器（alu）、加法器、寄存器堆、移位寄存器、总线分离器和接入器。

4) 存储器：RAM、ROM和CAM。

5) 系统级模块：乘法器、微控制器、URAT和RISC内核。

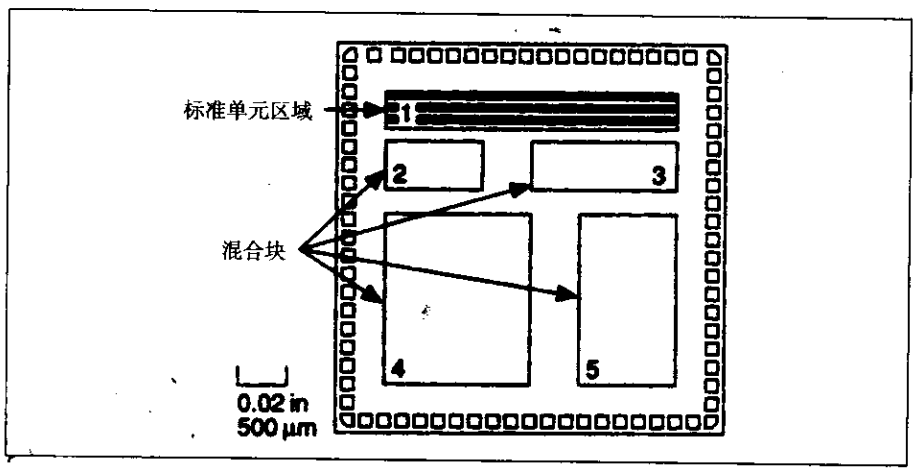


图9-24 标准单元与固定块的组合

当达到存储器系统级模块的复杂级别时，块使用图9-23中的单元布局的布局方法将不再有效。相反，它使用图9-24中所示的安排。

这里，标准单元（顶部的行）与属于存储器和系统级模块的固定块混合在一起。它们也称作内核（core），如果从其他公司买进，通常称作知识产权（IP）。

作为ASIC库的例子，我们考虑LSI Logic的500K库。表9-3给出了组合逻辑单元。通常有四个功率级别：a) 低(l)；b) 正常及高（a，b，c），其中a最低，c是最高的。对于像AND-OR-INV这样的电路，数字序列2-2-2表示输入阶段有三个2输入与门。表9-4给出触发器单元及它们的选项。这些都是基本单元。LSI提供CAD软件，可以根据这些基本单元建立宏，宏的例子包括：加法器（许多种）、计数器、大的多路器、递增器和递减器、译码器、移位寄存器、

FIFO、比较器，、乘法器、桶移位寄存器（barrel shifter）和ALU，同样也提供内核，如微处理器和混合信号电路模块（A/D和D/A）。

表9-3 LSI Logic的500K库的基本组合逻辑单元

功 能	输入的数目	功率级别
与非（NAND）	2，3	a,b,c,l
与非（NAND）	4~8	a,c
与（AND）	2，3	a,b,c,l
或非（NOR）	2，3	a,b,c,l
或非（NOR）	4~8	a,c
或（OR）	2，3	a,b,c,l
或（OR）	4	a,c
同或	2	a,b,c,l
同或	3	a,b,c
异或	2	a,b,c,l
异或	3	a,b,c,l
反相多路器	2	a,b,c,l
多路器	2	a,b,c,l
多路器	3，4，6，8	a,c
AND-OR-INV	2-1，2-2，2-1-1，2-2-2	a,c
OR-AND-INV	2-1,2-2,2-1-1 2-2-2-2	a,c
AND-OR-OR-INV	2-2	a,b,c
OR-AND-AND-INV	2-2	a,b,c
半加器	2	a,b,c
全加器	3	a,b,c

表9-4 LSI Logic的500K库的基本时序逻辑单元

触发器类型	选 项	功 率
D	异步清除	（a，c）
	异步置位和清除	
	异步置位	
	时钟下降沿触发	
	时钟下降沿触发和异步清除	
	许多扫描选项	
JK	异步清除	（a，c）
	异步置位和清除	
	许多扫描选项	

除了了解一个单元的功能，还需知道它的延时。要计算一个单元的延时，需要知道该单元的布线负载。然而如果该芯片还没有完成布局，那么不可能测出金属通道的长度。图9-25中包含了示例的连线负载表。对于给定的以mm^2为单位估计的布线面积和门扇出，该表给出了互连的容性负载。容性负载以标准负载的形式给出，即反相器的输入电容。如0.032pf。图9-26给出了一个D触发器的数据手册，为了计算器件的传输延时，使用下述等式：

$$C_T = C_{wireload} + C_{inpload} \tag{9-2}$$

尺寸 In mm2	扇出 1	2	3	4	5	6	7	8	16	32	64	Slope[2]
0.5x0.5	0.593	1.186	1.779	2.372	2.965	3.558	4.151	4.744	9.488	18.976	37.952	0.593
1.0x1.0	0.649	1.298	1.947	2.596	3.245	3.894	4.543	5.192	10.384	20.768	41.536	0.649
2.0x2.0	0.763	1.526	2.289	3.052	3.815	4.578	5.341	6.104	12.208	24.416	48.832	0.763
3.0x3.0	0.877	1.754	2.631	3.508	4.385	5.262	6.139	7.016	14.032	28.064	56.128	0.877
4.0x4.0	0.992	1.984	2.976	3.968	4.960	5.952	6.944	7.936	15.872	31.744	63.488	0.992
5.0x5.0	1.106	2.212	3.318	4.424	5.530	6.636	7.742	8.848	17.696	35.392	70.784	1.106
6.0x6.0	1.220	2.440	3.660	4.880	6.100	7.320	8.540	9.760	19.520	39.040	78.080	1.220
7.0x7.0	1.334	2.668	4.002	5.336	6.670	8.004	9.338	10.672	21.344	42.688	85.376	1.334
8.0x8.0	1.448	2.896	4.344	5.792	7.240	8.688	10.136	11.584	23.168	46.336	92.672	1.448
9.0x9.0	1.562	3.124	4.686	6.248	7.810	9.372	10.934	12.496	24.992	49.984	99.968	1.562
10.0x10.0	1.676	3.352	5.028	6.704	8.380	10.056	11.732	13.408	26.816	53.632	107.264	1.676
12.0x12.0	1.904	3.808	5.712	7.616	9.520	11.424	13.328	15.232	30.464	60.928	121.856	1.904
14.0x14.0	2.133	4.266	6.399	8.532	10.665	12.798	14.931	17.064	34.128	68.256	136.512	2.133
16.0x16.0	2.361	4.722	7.083	9.444	11.805	14.166	16.527	18.888	37.776	75.552	151.104	2.361
18.0x18.0	2.590	5.180	7.770	10.360	12.950	15.540	18.130	20.720	41.440	82.880	165.760	2.590
20.0x20.0	2.817	5.634	8.451	11.268	14.085	16.902	19.719	22.536	45.072	90.144	180.288	2.817

1. Wire length in mm = (wiresls * number_of_fanouts) + wiresic. For LCB500K: wiresic = 0; wiresls = 0.026X + 0.122
2. Intercept = 0.

图9-25 2层和3层金属的等价标准负载矩阵

其中$C_{wireload}$根据图9-25中的连线负载（wireload）表中得出，$C_{inpload}$根据驱动门提供的负载计算出来。例如，图9-26中D触发器的输入D提供了0.6的标准负载。一旦计算出C_T，立即用作查找延时表的索引。例如，如果C_T=10，则从图9-26的表中查出，版本fdlqa的tpLH是0.94 ns（0.87和1.02之间的内插）。

在芯片布局完成之后，可以测量金属通道的实际长度，实际互连延时回馈给使用标准延时格式（SDF）的仿真模型。

9.3.5 全定制芯片

全定制芯片中，在针对特定技术建立的设计规则下，设计者对于电路的设计有完全的控制权，如线的间隔。每个晶体管都可独立确定大小以求最优性能。然而，这种方法的设计效率较低，有这样的数字，如6.17晶体管/天/人（transistor/day/person）。这可以通过电子剪切或粘贴以前的设计来改进。但是，随着特征尺寸的减小，这种直接的重新利用不总是可能的。这样，定制设计需要大量的人力。一般它只用于标准部件设计，如微处理器。该领域的一个例外是混合信号设计。使用于通信电路的ASIC可以定制设计其模拟部分。

9.3.6 ASIC和FPGA的相对成本

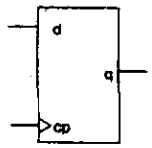
设计者不能只关心电路性能，还得考虑成本。表9-5给出MPGA和标准单元设计步骤的总结并与FPGA进行对比。FPGA不需要芯片制造和测试阶段，因为FPGA制造商已经完成了这两步。这样FPGA的Non-Recurring-Engineering（NRE）成本与MPGA或标准单元相比都要小得多。在MPGA和标准单元之间，标准单元的NRE高一点，因为标准单元的CAD过程复杂一点。必须做出更多的掩模且硅表面必须做更多层。Michael John Sebastian Smith所著的《专用

集成电路》(Addison Wesley于1997年出版)一书中的一些数字说明了这个例子。假设要实现10 000门电路，设计的典型NRE为：

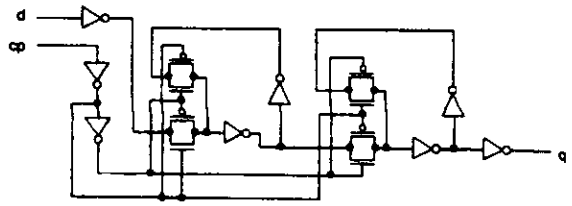
Name: fd1q Description: D Flip-Flop

Coding Syntax: u(q)=fd1q*(d,cp)

Logic Symbol:



Schematics:



Truth Table:

d	cp	q
0	↑	0
1	↑	1

Loading Characteristics:

Values stated in standard loads.

Version	d	cp
fd1qa	0.6	0.6
fd1qc	0.6	0.6

AC Characteristics:

VDD = 3.3 V, Junction Temperature = 25 degrees C, Standard Loading = Electrical Load + Wire Load, Process = Nominal. Values stated in nanoseconds. Ramp time = 0.2 ns (measured from 10% to 90% VDD).

Version	Cell Units	Delay Path	Output	Standard Loads				
				2	4	8	12	16
fd1qa	16	cp_to_q	tpLH	0.66	0.73	0.87	1.02	1.17
			tpHL	0.72	0.77	0.85	0.93	1.01
fd1qc	18	cp_to_q	tpLH	0.69	0.72	0.77	0.81	0.86
			tpHL	0.73	0.76	0.80	0.83	0.86

Timing Constraints:

Version	Parameters	d0	d1	Version	Parameters	d0	d1
fd1qa	d_setup	0.35	0.46	fd1qc	d_setup	0.35	0.46
	d_hold	-0.06	0.04		d_hold	-0.06	0.04
	cp_pw0	0.39	0.39		cp_pw0	0.39	0.39
	cp_pw1	0.23	0.23		cp_pw1	0.23	0.23

图9-26 D触发器数据手册

1) FPGA - $21,800
2) MPGA - $86,000
3) CBIC - $146,000

另一方面，单个元件的成本显示出相反的趋势。

1) FPGA - $39
2) MPGA - $10
3) CBIC - $8

三种方法成本的不同源自对单元的不同利用率。FPGA的利用率最低，MPGA的利用率次之，CBIC的利用率最高。这样，实现10 000门的电路，FPGA需要相对大的硅片尺寸（die size），而MPGA需要的硅片面积要小得多，但仍比CBIC大。大的硅片面积意味着每片圆片上只能获得有更少的硅片，从而导致高成本。

表9-5　MPGA和标准单元与FPGA的设计步骤比较

MPGA和标准单元	FPGA
系统设计	系统设计
逻辑设计	逻辑设计
布局和布线	布局和布线
时序仿真	时序仿真
测试样板产生	
掩模制造	
圆片制造	下载/编程
封装	
测试	
系统整合	系统整合

总的元件成本由下式给出：

$$\text{总的元件成本} = \text{NRE} + \text{成本} / \text{元件} \times \text{销售量} \tag{9-3}$$

图9-27说明了这种关系。对于小于11 000的小批量，FPGA成本低，在11 000到40 000元件之间，MPGA耗费少，在40 000个元件左右，标准单元CBIC的成本最低，这样，生产量是决定使用哪种方法的关键因素。

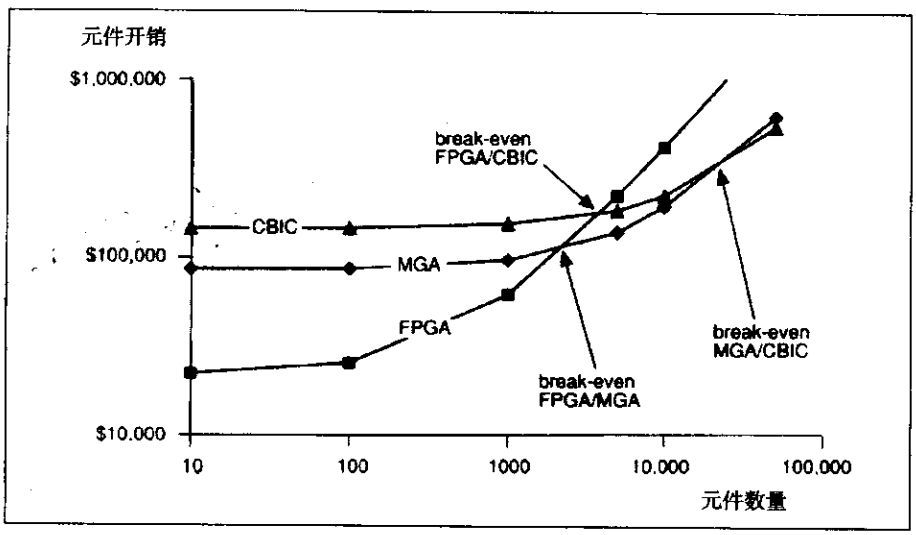


图9-27　元件总成本

另一个重要的考虑是迎合产品上市的时间。图9-28给出了一种情况，其中产品发布推后了一个季度，销售结果损失了$35 000。就投放市场的时间而言，三种方法由短到长排序如下：

1) FPGA

2) MPGA

3) CBIC

这种情况对于选择哪种方法有很深的影响。原来，FPGA被认为是严格的原型工具。然而，它们在领域之内的性能被证明是极优，而它们对于小的中等大小量的产品也是常用的选择，这样可以减少其上市时间。

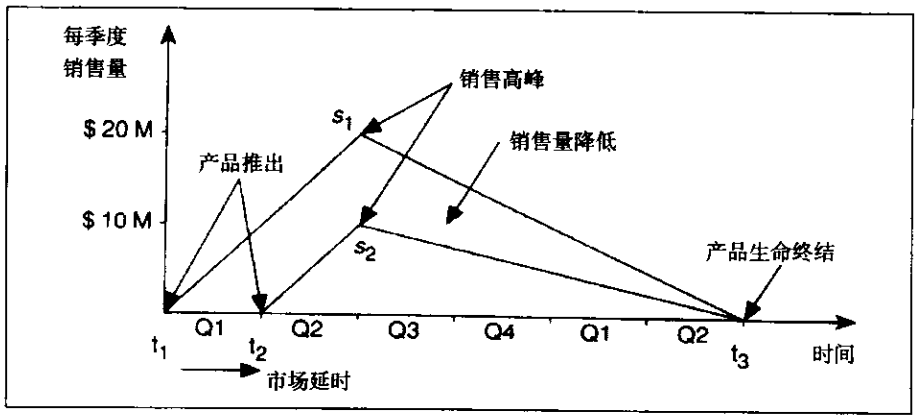


图9-28 上市时间对利润的影响

9.4 ASIC设计过程

本节介绍ASIC设计过程，侧重于HDL建模和综合方面。该过程可用于所有类型的ASIC和可编程逻辑。我们通过介绍Synopsys工具如何对标准单元ASIC进行综合和Xilinx工具如何综合FPGA来说明这个过程。

图9-29给出了ASIC的设计流程图，主要步骤是：

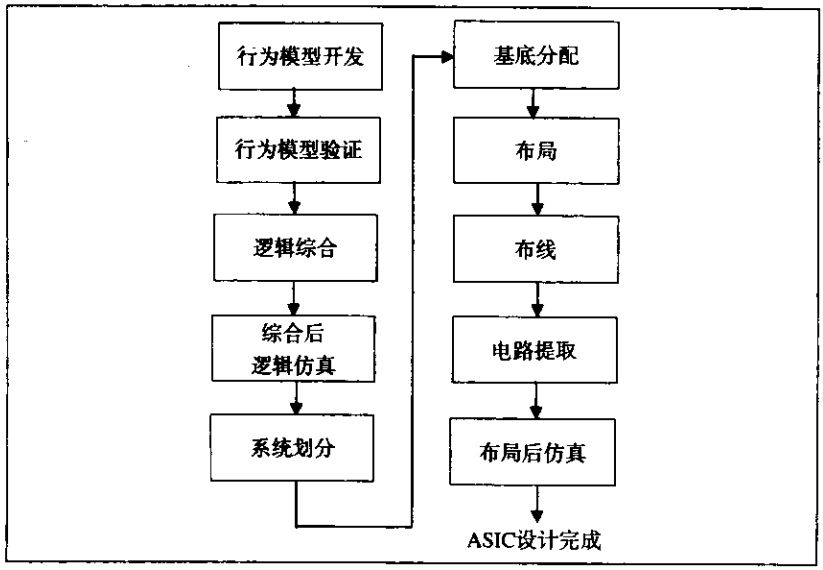


图9-29 ASIC设计过程

1) **行为模型开发**。在这一步，需求说明中的要求被转化为VHDL行为级模型。作为该过程的一部分，系统被分割成一组ASIC。第11章对分割过程进行了详细讲解。在进行功能分解时，块的大小受限于现存工具可以高效综合的门的个数。要使用自动化的综合工具，行为级模型必须以一种工具可接受的风格开发。我们将在第10章讨论综合风格，在系统被分为一些块之后，对每个块重复ASIC设计步骤。

2) **行为模型验证**。通常开发测试程序包来验证该行为模型，这是一个复杂的过程，因为测试程序包必须反映需求说明中的要求。一个主要问题是对模型进行穷尽的测试。例如，对一个时序电路模型，几乎不可能使用所有可能的输入序列。我们将在第10章详细讨论这个问题。测试程序包用来仿真该ASIC的行为模型。然后检查仿真结果，确定需求说明中的算法是否正确实现。在设计过程的这一阶段也可以检查时序关系。

3) **综合**。使用逻辑综合工具将行为级描述转化为结构化的门级电路。门级电布路由ASIC库中的基本单元组成。这个综合过程可能需要多次迭代。因为综合过程受延时、面积和功耗的限制。

4) **综合后逻辑仿真**。这一步仿真综合后的电路。a. 验证综合转换是否正确。对于成熟的工具，这一般不成问题，但工具中存在的不可避免的错误也许未被发现，尤其是当用户使用以前未使用过的编程风格时。b. 检查综合电路的时序。综合后电路的时序来自ASIC库和结构互连，而不是来自行为模型。这样，必须检查电路以纠正时序、输出延时、输出险态、寄存器建立时间等等。

5) **系统分割**。在开发行为级模型时，系统被分割成ASIC尺寸的块。然而，产生的门级电路可能相当大，需要在ASIC之内进行再分割。例如，在准备基底分配时，综合后的电路可能需要分割成功能块。这样的分割也许需要重复综合后逻辑仿真过程。

6) **基底分配**。这一步在芯片上安排网表块。图9-30给出了三个块（A、B和C）在芯片上的安排情况。

7) **布局**。在这一步，标准单元被布局在每个块中，以实现该块的功能。图9-30给出了块B内部的单元布局。

8) **布线**。这一步在单元和块之间进行布线连接。图9-30给出了块B中两个单元之间的连接。

9) **电路提取**。确定互连的阻抗和电容。这个信息反馈回门级模型，替代由连线负载模型产生的信息。标准延时格式（SDF）用于此处。

10) **布局后仿真**。在这一步，带有从布局布线得到的精确时序信息反标到门级电路重新进行仿真，以检查电路时序，并对电路功能进行最后检查。这些仿真的成功完成叫做ASIC sell off（开卖），现在，设计可以提供给硅铸造了。

9.4.1 标准单元ASIC综合

本节介绍综合标准单元ASIC的基本步骤。这里的讨论仅限于Synopsys，但其他厂商的工作步骤类似。我们在所有的例子中使用 LSI 10K库。其他库具有类似的特点。

对Synopsys系统，综合电路时一般经过下述步骤：

1) 分析。

2) 阐述。

3) 编译。

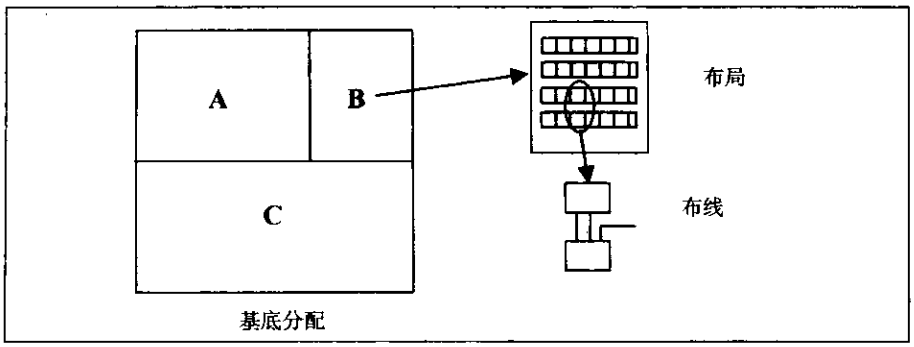


图9-30 基底分配、布局和布线

4) 报告。

5) 保存。

- **分析**。分析源VHDL文件，以便与VHDL可综合子集一致。如果使用了综合工具不支持的结构，即使是用常规的VHDL工具分析成功的模型，在这一步也可能分析不成功，如果分析成功，源VHDL文件被转化成一个中间格式。
- **阐述**。使用在分析步骤产生的中间格式，用类属组件创建设计。该设计是与技术无关的，即没有与任何特定的ASIC库绑定。
- **编译**。这一步将与技术无关的设计转化成基于库的设计。编译经常受要求的面积、延时或综合后的设计的功耗的限制。这一步产生的是用ASIC库宏表示的门级电路。
- **报告**。这个命令报告综合后电路的特征，如单元面积和关键路径的延时。这个命令有许多选项，如果想理解综合后的电路特性，就需要巧妙地使用该命令。
- **保存**。该命令保存综合的结果。这里使用了两种重要的文件格式。*.db格式使用工具的内部数据结构。编辑器工具无法读出这个文件。该文件可在后来重新加载，并进行新的编译。以*.vhd格式保存的文件产生电路的VHDL结构模型，可用于综合后电路的仿真，以验证其功能。该文件可用编辑器工具读出。

1. 综合工具使用策略

综合工具有两点比较重要。首先，文档没有完全定义工具的操作，这与VHDL仿真程序相反，其预期结果在VHDL语言参考手册中有定义。工具厂商意识到许多收入来自教授公司如何使用其工具上。这样，并非所有有用的技巧都在文档中指出。必须通过对工具进行实验以了解工具的操作。对于仿真程序也是这种情况，但综合工具要比仿真工具难用得多，这就导致了第二点，综合工具的操作很复杂。设计要受到许多方面的限制。编译之后，有许多报告选项可用来描绘它。这里阐明对工具进行实验的必要性。要从实验中学习，则过程必须是可重复的。当用菜单来控制综合工具时，记住一个给定实验的命令序列并不容易，使之重复则更难。为了处理这个问题，必须使用脚本来控制工具。

图9-31给出了一个计数器电路的VHDL描述。图9-32是一个简单脚本，对电路进行分析、阐述并根据时钟限制进行编译。然后文件保存为*.db和*.vhd格式，最后，生成报告。

图9-33a和图9-33b给出了示例报告。图9-33a是时序报告，在脚本文件（图9-32）中，产生了周期为10ns的时钟，这就给设计添加了时序限制，因为下一个状态逻辑的最长的延时（叫做关键路径延时）必须在一个时钟周期内完成。时序报告表明两个寄存器之间的关键路径

从COUNT_reg<0> CKK引脚（CP）开始，终止在COUNT_reg<3>的TI输入之上。图中列举了路径中的逻辑元件。总的路径延时是4.57ns。COUNT_reg<3>（FJK2S）使用的库单元的建立时间为1.80ns。这样，数据必须在8.2ns之内到达。路径有3.63ns的余地。这个路径是关键路径，因为所有其他与时钟有关的路径其剩余都比这个好。这样，该电路将以这种时钟周期动作。本章的后面将更加详细地讨论时序问题。

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.finc.all;
entity SM_COUNT is
  port(CLK,CON,RESET: in std_logic;
       COUNT: inout std_logic_vector(3 downto 0));
end SM_COUNT;

architecture ALG of SM_COUNT is
  begin
  process(CLK,CON,RESET)
    begin
      if RESET = '1' then
        COUNT <= "0000";
      elsif CLK'EVENT and CLK='1' then
        if CON = '1' then
          COUNT <= INC(COUNT);
      end if;
      end if;
  end process;
end ALG;
```

图9-31　计数器电路的VHDL描述

```
analyze -format vhdl sm_count.vhd
elaborate SM_COUNT -architecture ALG
create_clock  -period 10 -waveform {0 5} CLK
compile
write -format vhdl -output SM_COUNT_1.VHD
Write -output SM_COUNT_1.db
report_timing > SM_COUNT_1.rep
report_area  >> SM_COUNT_1.rep
```

图9-32　电路的综合脚本

图9-33b给出了面积报告，提供了每种类型单元的使用数量，并用等价的2输入NAND门给出了综合电路的面积，组合逻辑（14）和非组合逻辑（52）的面积。如果指定为连线负载模型，应该加入互连面积的估计值。在本例，没有指定连线负载类型。组合时序和面积报告提供信息以允许在这两个重要的设计参数之间进行折衷。

2. 最小延时电路

本节讨论如何综合具有最小延时的电路。这是在延时和面积之间所取的折衷。图9-34给出了理想的延时面积曲线，这是因为其延时是面积的单调递减函数。综合过程可以沿着这条曲线直到较平坦的区域开始。该区域的开始点即为最优设计，这一点之后，增加面积将不会使延时减少多少。

图9-35给出了一个更为实际的延时曲线。该曲线存在一个局部最小值和一个全局最小值。

这决定于开始的设计点和综合算法所采用的轨道，设计可能终止于局部最小点而无法达到全局最小点。我们通过例子说明这种情况，请看下面的命令脚本。

```
****************************************
Report : timing
         -path full
         -delay max
         -max_paths 1
Design : SM_COUNT
Version: 1998.02
Date   : Wed May 27 15:49:52 1998
****************************************

Operating Conditions:
Wire Loading Model Mode: top

 Startpoint: COUNT_reg<0>
             (rising edge-triggered flip-flop clocked by CLK)
 Endpoint: COUNT_reg<3>
             (rising edge-triggered flip-flop clocked by CLK)
 Path Group: CLK
 Path Type: max

 Point                                    Incr       Path
-----------------------------------------------------------
 clock CLK (rise edge)                    0.00       0.00
 clock network delay (ideal)              0.00       0.00
 COUNT_reg<0>/CP (FJK2S)                  0.00       0.00 r
 COUNT_reg<0>/Q (FJK2S)                   2.08       2.08 r
 U57/Z (ND2)                              0.30       2.38 f
 U55/Z (NR2)                              1.07       3.45 r
 U54/Z (EO)                               1.13       4.57 f
 COUNT_reg<3>/TI (FJK2S)                  0.00       4.57 f
 data arrival time                                   4.57

 clock CLK (rise edge)                   10.00      10.00
 clock network delay (ideal)              0.00      10.00
 COUNT_reg<3>/CP (FJK2S)                  0.00      10.00 r
 library setup time                      -1.80       8.20
 data required time                                  8.20
-----------------------------------------------------------
 data required time                                  8.20
 data arrival time                                  -4.57
-----------------------------------------------------------
 slack (MET)                                         3.63
```

图9-33a 时序报告

```
/*script 1*/
analyze -format vhdl sm_count.vhd
elaborate SM_COUNT -architecture ALG
create_clock  -period 10 -waveform {0 5} CLK
compile
```

命令为：

create_clock -period 10 -waveform {0 5} CLK

确定了周期为10ns，50%占空比的时钟，数据通路应终止于带时钟输入的触发器的限制。然后，编译电路。后续的时序报告或重要关键路径命令表示出关键路径是：

from COUNT_reg<0>/CP -to COUNT_reg<3>/TI with a delay of 4.57ns

```
****************************************
Report : area
Design : SM_COUNT

Version: 1998.02
Date   : Wed May 27 15:49:53 1998
****************************************

Library(s) Used:

  lsi_10k (File:
/software/synopsys/1998.02/libraries/syn/lsi_10k.db)

Number of ports:                  7
Number of nets:                  19
Number of cells:                 12
Number of references:             5
Combinational area:          14.000000
Noncombinational area:       52.000000
Net Interconnect area:       undefined
 (No wire load specified)
Total cell area:             66.000000
Total area:                  undefined
```

图9-33b　面积报告

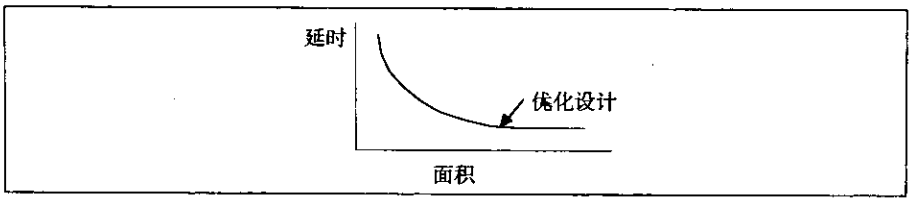


图9-34　理想的“延时—面积”曲线

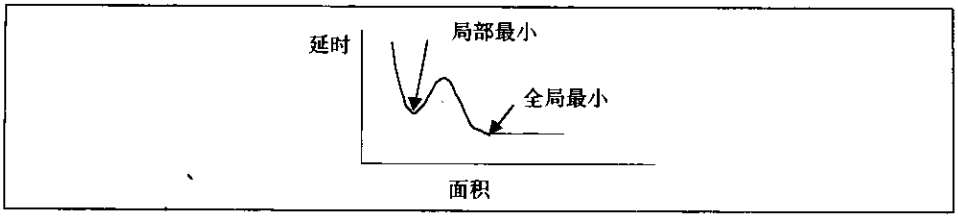


图9-35　实际的“延时—面积”曲线

可以用这个命令来限制这个路径：

set_max_delay 3 -from “COUNT_reg<0>/CP” -to “COUNT_reg<3>/TI”

该命令可以用来优化路径。重编译产生了另外一个电路，其关键路径为3.04ns，再次加上3ns限制，进行编译。该过程继续，一直到延时不再减少。表9-6总结了这种情况。

表9-6　情况1

操　作	延　时	面　积
执行脚本1	4.45	66
关键路径限制为3ns	3.04	67
关键路径限制为3ns	3.04	65

下面的脚本2应用了其他限制，其中开始点不同：

```
/*script 2*/
analyze -format vhdl sm_count.vhd
elaborate SM_COUNT -architecture ALG
create_clock  -period 10 -waveform {0 5} CLK
set_max_delay 0 -to all_outputs() + all_registers(-data_pins)
```

注意到命令：

set_max_delay 0 -to all_outputs( )+all_registers (-data_pins)

指定了对于所有终止于寄存器或输出的路径，最大延时为0。这个命令指导综合工具尽最大的努力以使电路上每条路径的延时最小。脚本2创建了一个最大延时为3.6ns和面积为69的电路。从该电路开始，在关键路径上附加3ns的限制，创建了一个延时为3.11ns和面积为68的电路，如表9-7中所示，附加的综合产生的变化不大。

表9-7 情况2

操 作	延 时	面 积
执行脚本2	3.60	69
关键路径限制为3ns	3.11	68
关键路径限制为3ns	3.11	69
关键路径限制为3ns	3.11	68

因为情况2创建的电路面积比情况1得到的大，可知情况2引起的优化算法陷于局部最小值之中，而情况1创建初始设计点，从中可以得出全局最优解。在设计空间中有许多路线，则由其命令序列创建的起始点和轨道要求，可确定得到的是哪类最小值。注意，使用脚本很重要，可以重复进行实验并进行比较。

3. 最优定时同步电路

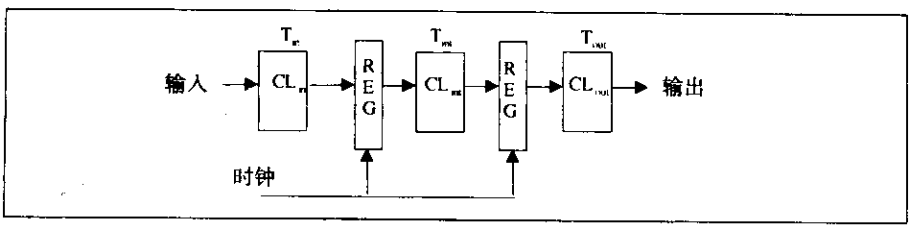


图9-36 同步系统

图9-36给出了同步系统的框图。假设希望系统在时钟频率为f_{clk}，时钟周期$T_{clk} = 1/f_{clk}$之下运行，这样显然：

$$T_{int} > T_{clk} \tag{9-4}$$

对于适当的操作。注意到T_{int}包括寄存器时钟延时和寄存器建立时间。我们将用这个定义来定义最优定时同步电路（OTSC），然而，在进行这样的定义时，也同样考虑到图9-36中的电路必须能够与其他性质相近的电路进行接口。这样，我们加上这样一条要求：

$$T_{in} < \frac{T_{clk}}{2} \text{ 和 } T_{out} < \frac{T_{clk}}{2} \tag{9-5}$$

如果满足条件(9-4)和(9-5)，就假设带时钟输入的寄存器之间的总延时比T_{clk}小，即使寄存器处在不同的电路中。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity ADD is
  port(CLK,RESET: in std_logic;
       A: in signed(1 downto 0);
       C: buffer signed(1 downto 0));
end ADD;

architecture ALG of ADD is
begin
  process(CLK)
  begin
    if RESET = '1' then
      C <= "00";
    elsif CLK'EVENT and CLK = '1' then
      C  <= A + C;
    end if;
  end process;
end ALG;
```

图9-37 寄存加法器的VHDL描述

尽管不必总满足条件(9-5)，在设计中包括它将是很好的设计实践，这样可以保证该新组件在所有应用之中都可使用。如果我们用一种综合工具来达到该目的，就必须根据要求认真进行设计。图9-37给出了两位寄存加法器的VHDL描述，可以用如下脚本对其进行综合：

```
analyze -format vhdl add.vhd
elaborate ADD -architecture ALG
create_clock  -period 10 -waveform {0 5} CLK
compile
```

然后对关键内部路径应用这些限制（CINTP），表9-8给出了这种情况，图9-38、图9-39和图9-40给出了产生的电路。注意，实验2改进了T_{int}的值，但以T_{in}的损失为代价。实验3减少了T_{in}并创建了时钟周期为4ns的OTSC电路。

表9-8 带寄存的加法器的综合情况

实 验	限 制	关键输入路径 (CINTP)	T_{int} (max)	关键输入路径 (CINPP)	T_{int} (max)	面 积	电 路
1	T_c=10ns	Creg<1>/CP To Creg<1>/D	4.27 ns	A To Creg<1>/D	2.25 ns	31	Add_int_ scr.db 加法器a
2	CINTP < 3ns	Creg<0>/CP To Creg<0>/D	2.75 ns	A To Creg<1>/TE	3.57 ns	35	Addb.db 加法器b
3	CINTP<2.5ns	Creg<0>/CP To C<0>	1.63 ns	A To Creg<0>	1.99 ns	36	Addc.db 加法器c

这样，与T_{int}一样，对Tin和Tout进行限制也很重要。我们通过对电路应用输入延时del_{in}和输出延时del_{out}来实现。延时被置在以时钟触发寄存器为终点（输入）或起点（输出）的路径之上。然后优化输入和输出路径，包括与时钟周期T_{clk}有关的那些延时，图9-41说明了这个方法。

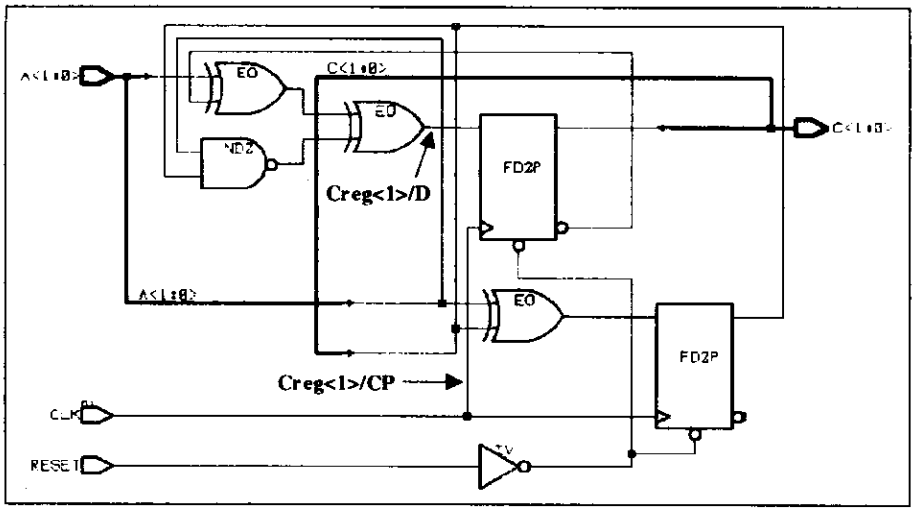


图9-38 加法器A

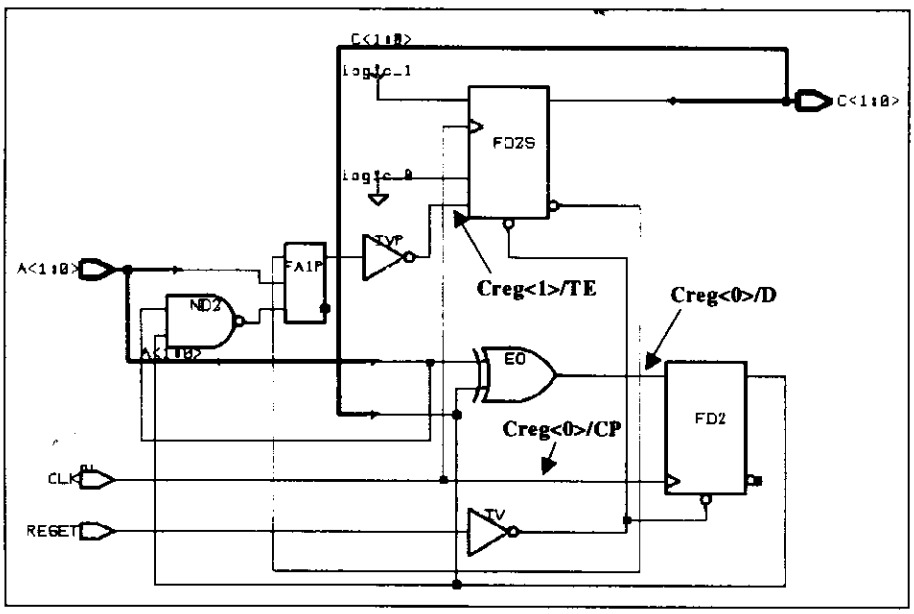


图9-39 加法器B

通常，工具将要达到$T_{in} = T_{clk} - del_{in}$和$T_{out} = T_{clk} - del_{out}$。对于我们的情况，为了达到$T_{in} < T_{clk} / 2$与$T_{out} < T_{clk} / 2$，令$del_{in} = del_{out} = T_{clk} / 2$。图9-42给出了达到这个目的的脚本。图9-43给出了综合后的电路。

图9-44中给出了报告文件（add.rpt）。报告按重要性的顺序列出了路径的端点，即最差的路径最先给出（其剩余时间值为–0.53），表示该路径长了0.53ns；其他路径根据其slack值按顺

序出现。前面的5个路径是输入路径，在时序元件FD2的输入D终止。（在本报告中，这一点不明显，其他报告提供了路径的端点，提供了该信息。）通过正在使用的库，可以知道单元的建立时间是0.85ns。对T_{clk} = 5ns，需要的数据到达时间（路径所需）是4.15ns。第一条实际通路为4.68ns，这样产生了-0.53ns的剩余。下面的四条路径同样也是输入路径，终止于时序元件FD2的输入TE之上，其建立时间为1.25ns，数据到达时间为3.75ns。第10条路径是一条输出路径，要求数据在时钟上升沿之后2.5ns到达。第11条路径是一条寄存器间的内部路径，在输入TE端中止，所要求的数据到达时间为3.75ns。最后给出的三条路径是和路径10类似的输出路径。要看到路径的两个端点，可以将report_timing命令-path参数改成short。要看全部路径，其中给出了经过的全部单元，可将-path选项改作full，-max_paths选项限制了输出路径的个数。-nworst选项控制在一特定终点有多少条终结的通路可以显示。在我们的例子中，-nworst=40。这里只显示了14条路径。然而注意到在C_reg<1>/D和C_reg<0>/TE各有五条路径终结，这样此引脚超出了最大值，因而就可以定下结论，如果nworst足够大，则还可以显示更多的26条其他路径。增加nworst使我们可以看到更多的路径类型，它以对某一相同节点的重复通路作为代价。

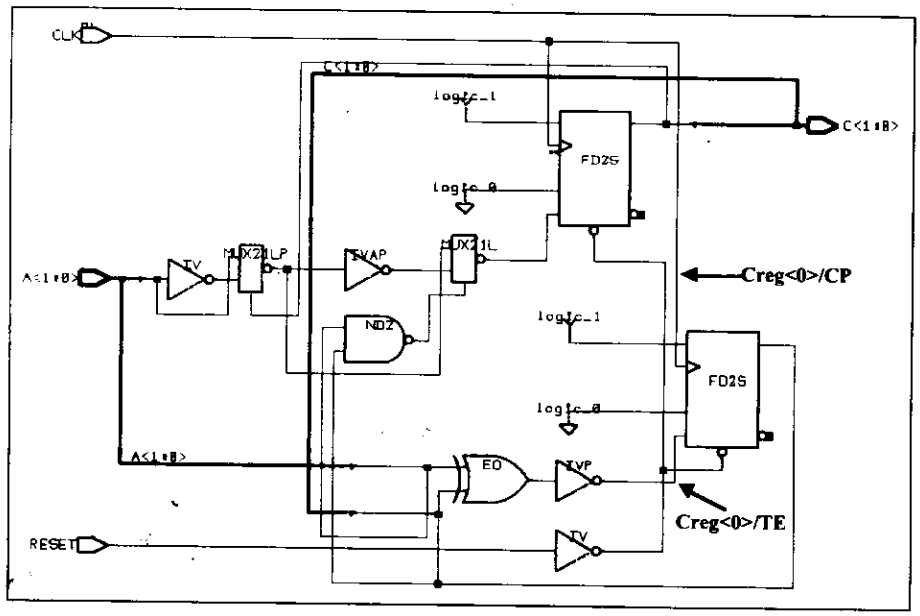


图9-40　加法器C

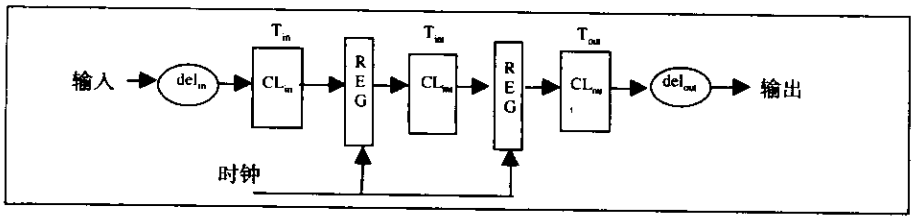


图9-41　输入输出延时优化

结果表明对于T_{CLK} = 5ns，无法得到OTSC，这是因为输入延时太大，然而，如果将T_{CLK}改为5.53ns，则可以满足这个要求。另外一种方法是，可限制从输入A到C_reg<1>/D的延时，然

后再综合一次。

```
analyze -format vhdl add.vhd
elaborate ADD -architecture ALG
create_clock  -period  5 -waveform {0 2.5} CLK
set_input_delay -clock CLK -max -rise 2.5 "A"
set_input_delay -clock CLK -max -fall 2.5 "A"
set_output_delay -clock CLK  -max -rise 2.5 "C"
set_output_delay -clock CLK  -max -fall 2.5 "C"
compile
report_timing -path end -delay max -max_paths 40 -nworst 5 >
add.rpt
```

图9-42 OTSC脚本

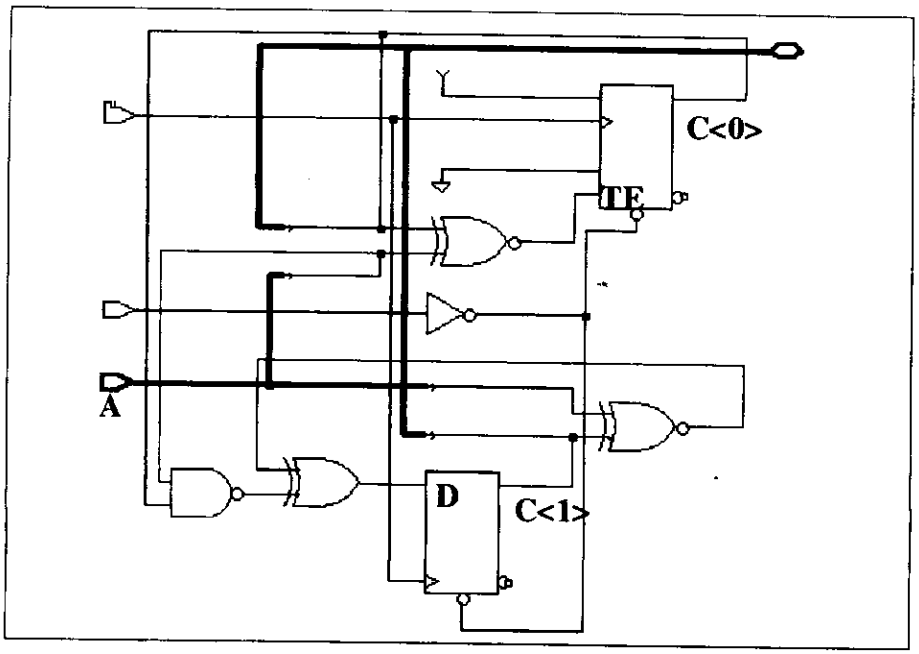


图9-43 OTSC电路

上述的情况建议使用迭代脚本来开发OTSC，该脚本的高级描述为：

1) 用如图9-42所示的脚本类型首次综合电路。

2) 确定有最差剩余时间的关键路径的T_{in}、T_{inp}和T_{out}。

3) 用r倍（T_{in}、T_{int}或T_{out}）作为该关键路径的限制，其中0<r<1。

4) 重新综合电路。

5) 如果电路是OTSC，则停止；否则，返回第2步。

4. 综合后模型的验证

综合后的模型可用手工检查及仿真来进行验证。

手工检查尽管只对最小的模块较为实际，但对于理解综合转化是如何实现的及基本的ASIC单元是如何操作的，会大有帮助。这样，从教育的观点来看，能检查简单的综合电路和完善电路功能是一种很重要的能力。图9-37中给出了时钟触发的两位加法器的VHDL描述。图

9-39给出了从VHDL描述（Adder B）中综合得到的电路。

```
****************************************
Report : timing
         -path end
         -delay max
         -nworst 5
         -max_paths 40
Design : ADD
Version: 1998.02
Date   : Wed May 27 08:03:21 1998
****************************************
Operating Conditions:
Wire Loading Model Mode: top

Endpoint              Path Delay      Path Required      Slack
------------------------------------------------------------
C_reg<1>/D (FD2)        4.68 f             4.15          -0.53
C_reg<1>/D (FD2)        4.68 f             4.15          -0.53
C_reg<1>/D (FD2)        4.50 r             4.15          -0.35
C_reg<1>/D (FD2)        4.50 r             4.15          -0.35
C_reg<1>/D (FD2)        4.50 f             4.15          -0.35
C_reg<0>/TE (FD2S)      3.69 f             3.75           0.06
C_reg<0>/TE (FD2S)      3.69 f             3.75           0.06
C_reg<0>/TE (FD2S)      3.58 r             3.75           0.17
C_reg<0>/TE (FD2S)      3.58 r             3.75           0.17
C<0> (inout)            1.63 r             2.50           0.87
C_reg<0>/TE (FD2S)      2.82 f             3.75           0.93
C<0> (inout)            1.53 f             2.50           0.97
C<1> (inout)            1.48 r             2.50           1.02
C<1> (inout)            1.47 f             2.50           1.03
```

图9-44　路径报告

根据库文件lsi_10k.lib，可以识别元件的功能，如：1) ND2——两输入NAND；2) EO——两输入异或；3) IV，INV——反相器；4) FD2S，FD2——带时钟的触发器；5) FA1P——全加器。ND2、EO、IV和IVP的功能如上。FD2的功能与普通的带时钟D触发器一样。这里需要知道FD2S和FAIP的引脚和逻辑方程。单元引脚可以在Design Analyzer中通过使用View Style命令打开这一层来确定。逻辑方程来自.lib文件。

图9-45给出了FA1P和FD2S的引脚。现在确定电路的功能。从图9-39中：

$$C<0>\ (D) = A<0>\ \text{XOR}\ C<0>$$

对于和的低位，上式是正确的。根据*.lib文件，FA1P（全加器）的和的输出（S）组件是：

$$S = CI\ A'\ B' + CI'\ A\ B' + CI'\ A'\ B + CIAB$$

然后，如果使CAR=A<0>C<0>，并且注意到全加器的输入是A=A<1>，B=CAR',且CI=C<1>'，那么：

$$S = C<1>'\ A<1>'CAR + C<1>\ A<1>\ CAR + C<1>\ A<1>'\ CAR' + C<1>'\ A<1>\ CAR'$$

即是 C<1>、A<1>和位置0的进位，CAR的异或。

在单元FD2S，下一个状态 Q的表达式如下：

$$D\ (TE)' + (TI)\ (TE)$$

其中TI=0，且D=1与(TE)'相等。然后，假设全加器的输出经过一个反相器，C<1>的下一个状态将取自上面给出的S值，这样C<1>是和的高位的正确值。

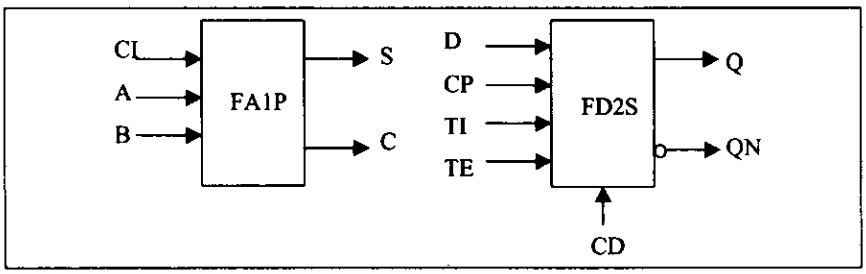


图9-45 FA1P和FD2S的引脚输出

9.4.2 综合后仿真

在综合结束之后，可对得到的结构模型进行仿真，以验证其正确性。这样做有两个原因，首先，不可完全信任综合工具进行的转化。它们的正确性由用户反馈来建立。第二，必须验证综合后电路的时序。综合后电路的时序来自目标ASIC库，而非原来作为综合工具的输入的模型。综合工具忽略输入模型的时序。不仅是通路延时不同，因为它是门级模型，综合的模型将会出现类似险态这样的时序现象，这些现象在原来的模型之中并未给出。当将综合后的模型与原来的模型进行比较时，必须考虑这些差异。

假设我们打算综合图9-31中所示的计数器模型。在综合之前，用测试程序包来测试模块(图9-46)，在计数器启用的情况下（CON=1），使之在100ns的时钟周期下运行。在这种情况下，因为模型是delta延时行为级模型，计数器输出在时钟上升沿到来后delta延时发生改变。

假设该模型已经综合过，并且综合后的模型由同一测试程序包测试，就是说，计数器在CON=1时进行运行。现在，即使时钟在100ns间隔进行开关，由于综合后电路的门延时，计数器的输出（COUNT）改变落后。不只这样，电路还出现了险态（正常）。图9-47给出了综合后电路输出从3转化到4的情况。理想情况下，电路切换速度应为400ns（400 000ps），但直到1.45ns（1450ps）之后才发生变化，并在1470ps之后稳定下来之前出现了20ps的险态。该情况说明模型需要验证。可以通过比较综合后电路和行为级模型的仿真输出来验证综合后模型。因而，比较两个模型时必须考虑延时，即，在与行为级模型的输出比较时，必须确定结构模型的输出已经稳定。进行比较的最佳时机是时钟上升沿，那时行为级模型尚未改变（delta延时之后它将会改变），而且综合后的模型在前一次时钟转换之后已经稳定下来。

也许在仿真综合后模型时，想使用同一个测试程序包（见图9-46）。行为级源代码的文件名为sm_count.vhd。该模型的实体名为SM_COUNT。当保存综合后的VHDL门级模型时，应该使用一个不同的UNIX文件名，如SM_COUNT.vhd，所以并没有覆盖源文件。综合工具使用SM_COUNT作为门级模型的实体名。注意在测试程序包文件中，缺省绑定为SM_COUNT。该绑定会使用最近分析过的实体SM_COUNT。这样，可以用测试程序包来测试行为级模型或综合模型。

对行为级仿真进行分析，使用这样两个命令：

```
vhdlan -t ps -xsim sm_count.vhd
vhdlan -t ps  -xsim tb_count.vhd
```

```
Library IEEE;
use IEEE.std_logic_1164.all;
entity TB_COUNT is
end;
architecture TESTBENCH of TB_COUNT is
  signal CLK,CON,RESET : std_logic := '0';
  signal COUNT: std_logic_vector(3 downto 0);
  component SM_COUNT
    port(CLK,CON,RESET: in std_logic;
         COUNT: inout std_logic_vector(3 downto 0));
  end component;
begin
  UUT :SM_COUNT
    Port Map (CLK,  CON,  RESET,  COUNT);
SignalSource : process
  begin
    CON <= '0', '1' after 60 ns;
    RESET <= '1', '0' after 40 ns;
    CLK <= '1', '0' after 50 ns, '1' after 100 ns,
           '0' after 150 ns,'1' after 200 ns,
           '0' after 250 ns, '1' after 300 ns,
           '0' after 350 ns, '1' after 400 ns,
           '0' after 450 ns, '1' after 500 ns,
           '0' after 550 ns,'1' after 600 ns,
           '0' after 650 ns, '1' after 700 ns,
           '0' after 750 ns, '1' after 800 ns,
           '0' after 850 ns, '1' after 900 ns,
           '0' after 950 ns, '1' after 1000 ns,
           '0' after 1050 ns, '1' after 1100 ns,
           '0' after 1150 ns,'1' after 1200 ns,
           '0' after 1250 ns, '1' after 1300 ns,
           '0' after 1350 ns, '1' after 1400 ns,
           '0' after 1450 ns, '1' after 1500 ns,
           '0' after 1550 ns, '1' after 1600 ns,
           '0' after 1650 ns, '1' after 1700 ns,
           '0' after 1750 ns, '1' after 1800 ns,
           '0' after 1850 ns, '1' after 1900 ns,
           '0' after 1950 ns, '1' after 2000 ns,
           '0' after 2050 ns, '1' after 2100 ns,
           '0' after 2150 ns;
    wait;
  end process;

end TESTBENCH;
configuration CFG_TB_COUNT of TB_COUNT is
  for TESTBENCH
    for UUT : SM_COUNT
    end for;
  end for;
end;
```

图9-46 计数器测试程序包

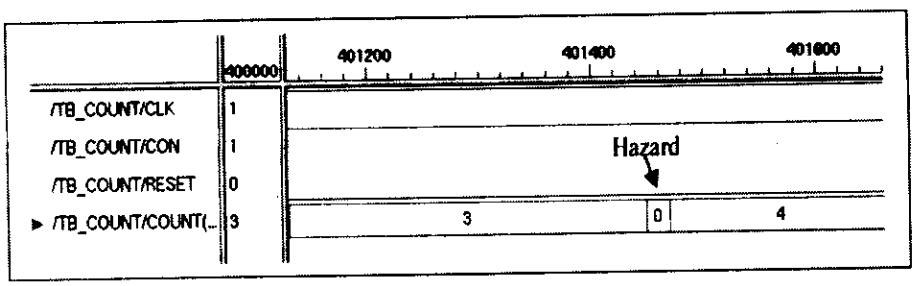


图9-47 综合后的电路响应

行为级模型与测试程序包绑定在一起。

对门级综合电路的仿真，使用这样两个命令：

```
vhdlan -t ps -xsim SM_COUNT.vhd
vhdlan -t ps  -xsim tb_count.vhd
```

则结构模型与测试程序包绑定在一起。使用的时标为ps（皮秒），因而能够发现险态。

下面是关于测试程序包如何被绑定的另一种方法的介绍。在配置声明语句中，绑定是通过缺省绑定实现的。该语句对无论是原来的行为级模型还是由综合工具产生的结构模型都使用其缺省的绑定。综合的构架中不包含配置说明。当然，组件声明使用的名字与库元件相同。例如，用ND2代表两输入NAND门。缺省绑定把这些组件声明名和单元仿真库中的模块名联系在一起。在顶层和底层都通过寻找与组件声明中相同的实体名的模块来进行绑定。

9.5 FPGA综合

本节介绍FPGA综合。我们用Xilinx的软件和XC4010XL SRAM可编程FPGA芯片进行示例说明。FPGA设计与图9-29中ASIC设计的过程有些区别。图9-48给出了FPGA设计过程。

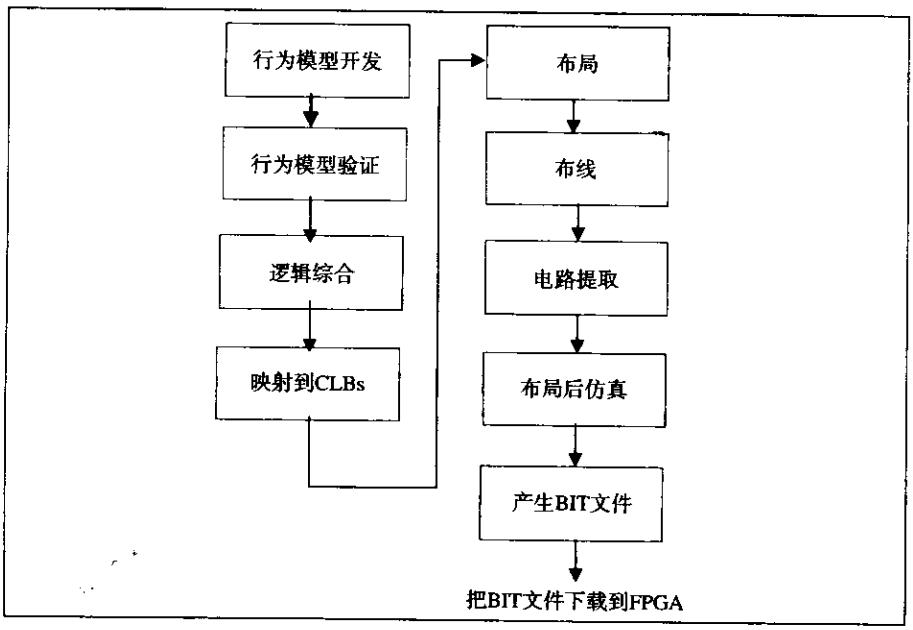


图9-48 FPGA设计过程

行为级模型的设计和图9-29中ASIC的设计过程一样。下一步是逻辑综合，行为级描述被转化成门级模型。在这种情况下，门级描述没有延时，最终的延时来自FPGA电路。综合由Xilinx开发的、包含在Xilinx工具包之中的FPGA Express程序进行。FPGA Express产生一个*.xnf文件，送入Design Manager程序，由此程序完成设计工作的余下部分。

图9-49给出了Design Manager执行的设计流程。

1) **转换**。该过程调用程序NGDBUILD，将由FPGA Express产生的*.xnf文件转换为本地类属数据库（Native Generic Database，NGD）文件。在这一步，当将*.xnf的基元映射到Xilinx库基元时，要使用一些映射技术。

2) **映射**。映射将CLB、IOB或其他Xilinx资源分配给设计中的逻辑元件。

3) **布局和布线**：PAR程序对设计映射的CLB和IOB自动进行优化布局和布线，该过程可以服从于用户指定的时序限制。首先进行布局，以尽可能减少布线的长度，如果布线失败则可能需要迭代。在FPGA中布线比在ASIC中复杂，因为布线资源是固定的。布局布线的结果是一个*.ncd文件，即本地电路描述（Native Circuit Description）文件。

4) **定时**。由Trace程序在已布局和布线的设计中进行时序分析。Trace是静态时序分析程序，确定设计中的路径延时，并检查是否违反了设计上所加的时序限制，生成报告，如果必要，该报告将用来修改设计。

5) **配置**。该功能是由BitGen程序实现的。它根据一个完全布线的.ncd文件创建配置位流（BIT文件）。BIT文件下载到FPGA的控制存储器单元，或创建在一个操作环境中用来完成初始化的PROM。

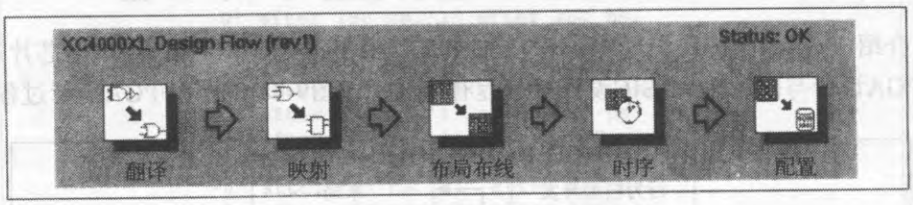


图9-49　Design Manager设计流程

9.5.1　FPGA示例

现在给出FPGA设计示例。使用的Xilinx芯片是XC4010XL，插在版本1.2的板子XESS XS40上，加上了一个XTEND板。两种板的混合提供多种输入和输出器件以驱动芯片，比如微控制器，RAM和立体声编码解码电路。

图9-50的例子包括到时钟触发器的输入逻辑，时钟触发器之间的逻辑，时钟触发器和输出及三态输出电路之间的逻辑。图9-51给出了XESS/XTEND板提供的通用I/O资源：1) 输入：三个按钮，八个dip开关，一个晶振。2) 输出：三个七段显示和八个LED。该接口电路的一个模型必须作为顶层结构模型，图9-52给出了顶层结构模块的框图。

TOPLEVEL模块实例化了三个组件：1) TIMECKT——要设计的模块。2) CLOCKGEN——该模型从12MHz时钟分频产生合适的时钟频率。3) DISPLAY——调整TIMECKT的输出，驱动七段和LED显示。DISPLAY本身是结构化模型，包括十六进制到七段的转换器（HEX2SEV）和7位、8位反相器（INV7和INV8），我们将其他模块的实现作为本章的习题。

对一个设计进行布局和布线，设计者必须准备一个用户限制文件（User Constraints File, UCF）。图9-53给出这样的一个文件作为例子。文件的主要部分确定了信号名到引脚名的映射。然而，文件的底部是时序限制部分。

1) 第一个TIMESPEC命令确定了带时钟的触发器的延时不超过10ns的限制。FFS代表通用信号组触发器。

2) 第二个TIMESPEC命令确定了输入PADS和带时钟的触发器之间的延时不应该超过4ns，PADS代表通用信号组PADS。

3) TNM命令定义了一个信号组CKTOUT，并且将信号F置于该组之中。

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity TIME_TEST is
Port( A,B,C,EN,CLK,RESET: in STD_LOGIC; F: out STD_LOGIC);
end TIME_TEST;

architecture ALG of TIME_TEST is
  signal  FF1, FF2: STD_LOGIC;
begin
 P1:process(RESET,CLK)
  begin
    if RESET = '1' then
      FF1 <=  '0';
    elsif ( CLK'EVENT and CLK = '1')
      FF1 <= A and B;
    end if;
  end process;
 P2: process(RESET,CLK)
  begin
    if RESET = '1' then
      FF2<=  '0';
    elsif ( CLK'EVENT and CLK = '1')
      FF2 <= FF1 and C;
    end if;
  end process;
  F <= FF2 and FF1 when EN = '1' else
       'Z' when others;
end ALG;
```

图9-50 模块（TIMECKT.VHD）示例

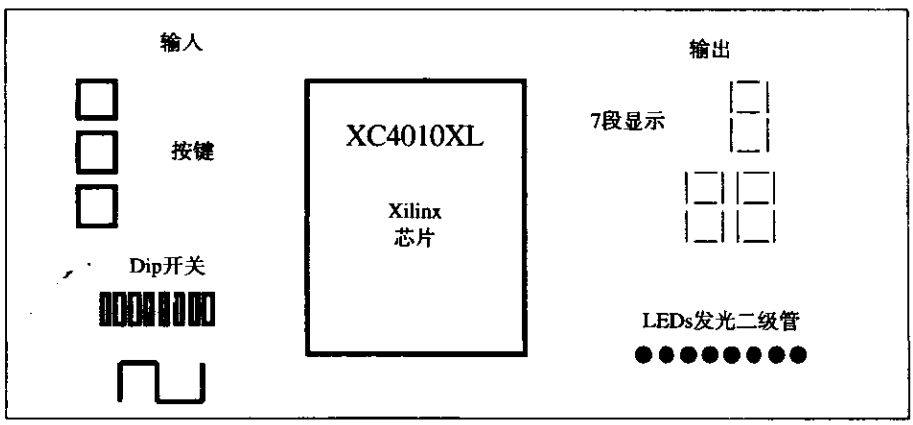


图9-51 FPGA I/O接口

4) 第三个TIMESPEC命令确定了FFS和信号组（CKTOUT）之间的延时不可超过17ns。

图9-54给出了示例电路部分FPGA的布局。左上脚的CLB实现了进程P1和进程P2的所有功能（见图9-50）。它们利用了两个4输入LUT和两个触发器。中间的CLB使用了一个4输入LUT来实现FF1和FF2的“与”操作。右下角的CLB使用一个4输入LUT反相输入信号EN，在信号集中的右下角，三态缓冲器用来实现模块中的条件赋值语句。第4个CLB（未画出）用来反相复位按钮的信号，并将其与启动电路相连。

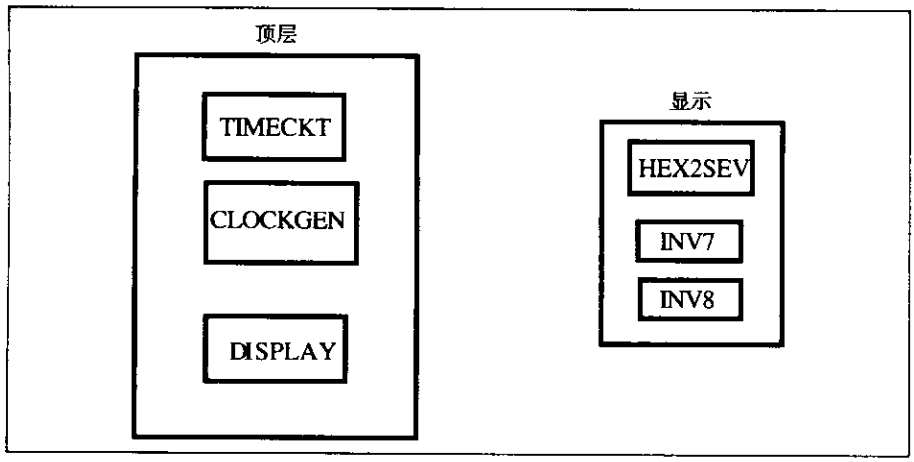


图9-52 顶层结构模块

表9-9给出了布局和布线之后FPGA的阵列反馈的值。带*的限制并没有被满足。

表9-9 定时的结果

限 制	要 求	实际情况	逻辑层次
FFS to FFS	10ns	4.01ns	2
PADS to FFS *	4ns	4.725ns	2
FFS to F *	17ns	22.2ns	4

9.5.2 与ASIC设计的比较

图9-55显示了TIMECKT.VHD的ASIC设计。该设计利用了FD2S库元件的TI和TE输入来执行对这个元件的“与”操作。这样就没有输入逻辑，没有时序元件之间的逻辑。F的输出延时是3.18ns。FD2S元件的时钟延时是1.42ns，建立时间为1.25ns，其最小时钟周期为2.67ns，最大时钟周期为374.5MHz。Xilinx FPGA设计的TRACE程序预测了最大时钟周期为4.01ns。最大时钟频率为249MHz。然而，LSI 10K库是成熟的技术，而XC4010A相对较新。来自LSI 500K库的一个可比较的时序元件其时钟延时为0.58ns，建立时间为0.61ns，对于这个例子中的设计，产生的最大时钟周期为1.19ns，时钟频率为840MHz。ASIC库设计的面积为25个等效2输入NAND门。Xilinx映射过程的结果给出了等效门数为39。注意，Xilinx使用的两个4输入LUT效率很低，一个用来实现2输入函数，一个用来实现反相器。

习题

9.1 解释标准部件设计和ASIC设计如何对应第1章讲述的自顶向下和自底向上设计。

9.2 查阅供应商的资料和贸易杂志，分别找出一种商业产品的例子，它属于：1) 标准部件设计。2) ASIC设计。3) 标准部件和ASIC设计混合。

9.3 给出5种可以购买到的大型内核的例子（IP）。

9.4 列出4个提供VHDL仿真和综合能力的CAD工具生产商。

9.5 给出一个混合了冯·诺依曼机构架、ASIC及FPGA的产品实例。

```
# TIMECKT Configuration File
NET XS40_CLK    LOC=P13;
# TOP SEVEN SEGMENT CONNECTIONS(ACTIVE-LOW)
NET TOP_7S<6>  LOC=P25;
NET TOP_7S<5>  LOC=P26;
NET TOP_7S<4>  LOC=P24;
NET TOP_7S<3>  LOC=P20;
NET TOP_7S<2>  LOC=P23;
NET TOP_7S<1>  LOC=P18;
NET TOP_7S<0>  LOC=P19;
# Dip Switch Connections
NET DS<3>     LOC=P6;
NET DS<2>     LOC=P9;
NET DS<1>     LOC=P8;
NET DS<0>     LOC=P7;
#PUSHBUTTON SWITCHES (ACTIVE-LOW)
NET RESET_BUT  LOC=P37;
# LEFT SEVEN SEGMENT CONNECTIONS (ACTIVE-LOW)
NET LEFT_7S<6> LOC=P3;
NET LEFT_7S<5> LOC=P4;
NET LEFT_7S<4> LOC=P5;
NET LEFT_7S<3> LOC=P78;
NET LEFT_7S<2> LOC=P79;
NET LEFT_7S<1> LOC=P82;
NET LEFT_7S<0> LOC=P83;
# RIGHT SEVEN SEGMENT CONNECTIONS (ACTIVE-LOW)
NET RIGHT_7S<6>LOC=P59;
NET RIGHT_7S<5>LOC=P57;
NET RIGHT_7S<4>LOC=P51;
NET RIGHT_7S<3>LOC=P56;
NET RIGHT_7S<2>LOC=P50;
NET RIGHT_7S<1>LOC=P58;
NET RIGHT_7S<0>LOC=P60;
# LED CONNECTIONS (ACTIVE-LOW)
NET LED<7>      LOC=P10;
NET LED<6>      LOC=P80;
NET LED<5>      LOC=P81;
NET LED<4>      LOC=P35;
NET LED<3>      LOC=P38;
NET LED<2>      LOC=P39;
NET LED<1>      LOC=P40;
NET LED<0>      LOC=P41;
#Timing constraints
TIMESPEC TS01=FROM FFS TO FFS 10;
TIMESPEC TS02=FROM PADS TO FFS 4;
NET F TNM=CKTOUT;
TIMESPEC TS03=FROM FFS TO CKTOUT 17
```

图9-53 UCF文件

9.6 画出CMOS三输入NAND门的晶体管级电路图。

9.7 CMOS反相器的典型静态功耗是多少？当时钟周期为10ns，V_{cc}为4.75V，负载电容为100pf时，它的动态功耗是多少？若要求节约CMOS微处理器在非计算的周期时的功耗，可以采取什么办法？

9.8 为什么CMOS逻辑电路具有低的静态功耗？

9.9 为什么使用门构造复杂芯片时低功耗非常重要？

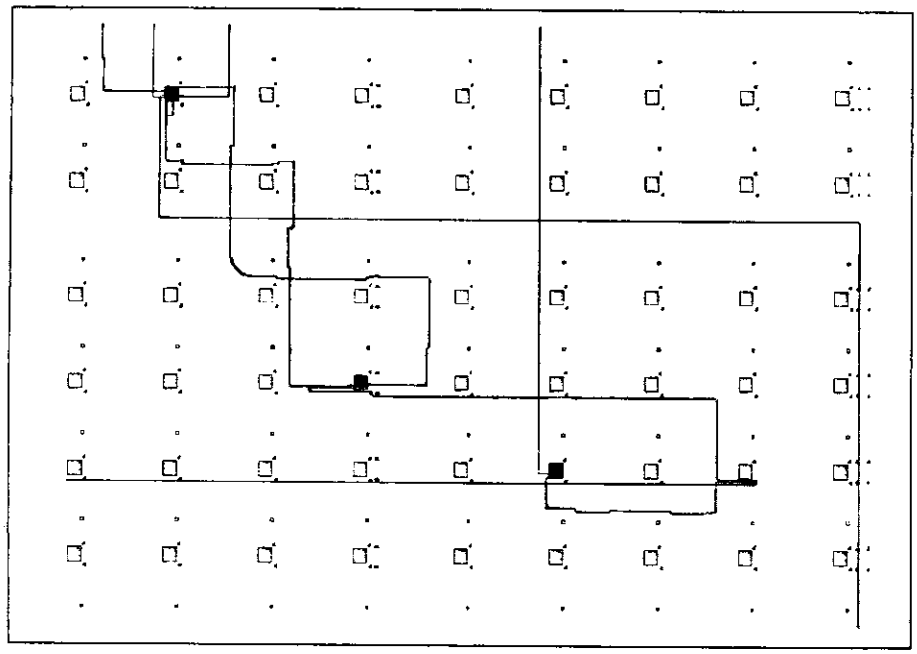


图9-54 作为示例部分的FPGA布局

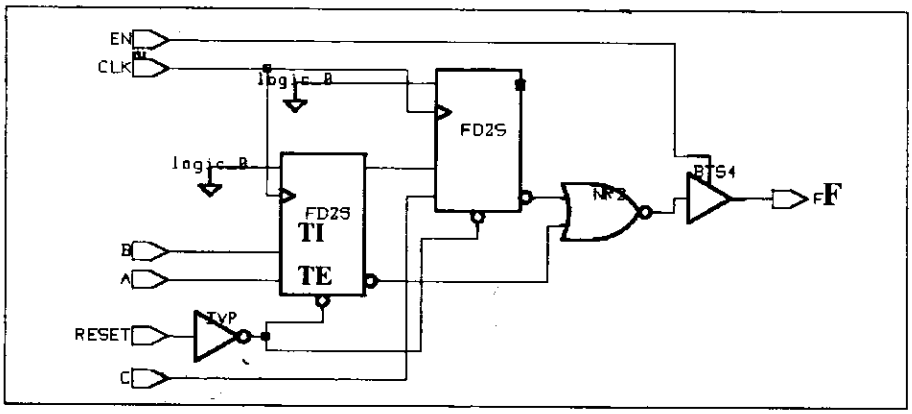


图9-55 ASIC设计

9.10 当今ASIC CMOS实现时的最小特征尺寸是多少？FPGA CMOS实现呢？

9.11 在PLD上实现图8-22中给出了状态图的状态机。对其进行仿真，估计所需PLD资源的百分比。

9.12 实现三输入的多数函数（参见第3章的1'计数器实例），使用：1) 一个8选1多路复用器；2) 一个4选1多路复用器及必需的反相器；3) 查找表。比较这三种电路的硬件复杂度。

9.13 图9-56a是“相等时翻转”电路的框图。该电路是一个时钟驱动的时序电路。当X1等于X2时，输出Z翻转。当X1与X2不相等时，输出Z不发生变化。检测器具有一个高度活跃的异步重置信号R，它使得输出Z变为‘0’，图9-56 b给出了一个FPGA的CLB单元。单元中的表可以实现任意三变量函数。说明CLB如何实现检测器。假设：1) A是表的最高地址位，C是最低地址位；2) 当多路器控制位为‘0’则选择上面的输入；3) 时钟

（CLK）为到单元的硬连线，输出Z应等于电路的当前状态。

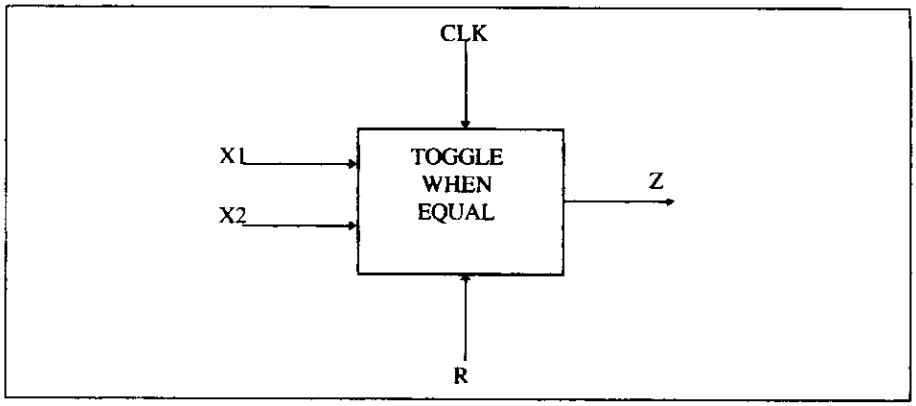


图9-56 a “相等时翻转”检测器

9.14 本题使用习题9.13的图。但图9-56 a是“最后一致检测器”的框图，该电路是一个时序电路，输出Z等于X1和X2相等的最后值，所以，若最后一次X1=X2=‘1’，则Z=‘1’。若最后一次X1=X2=‘0’，则Z=‘0’。当X1不同于X2时，Z不发生改变。检测器有一个非常活跃的异步重置R，它使得输出Z变为‘0’。图9-56 b给出了一个FPGA的CLB单元，该表可实现任意三变量函数。说明如何用该CLB实现“最后一致检测器”，假设：1) A是表的最高地址位，C是最低地址位。2) 当多路器控制位为‘0’时，选择上面的输入。3) 时钟（CLK）为到单元的硬连线。

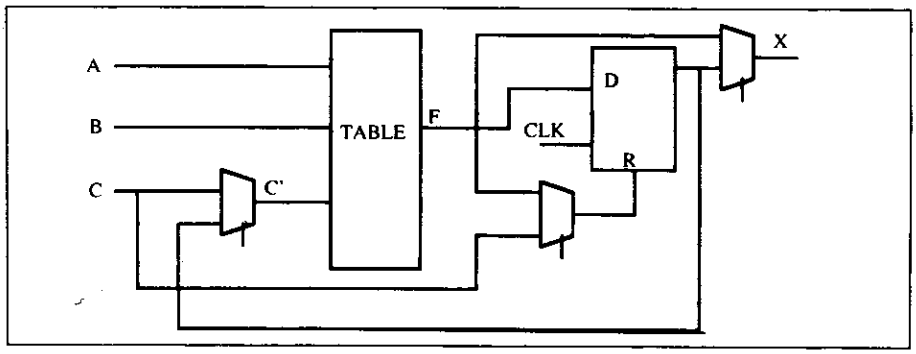


图9-56 b FPGA CLB

9.15 用图9-14的开关延时模型决定延时。假设输入为一个5V、10ns的脉冲，使用SPICE决定下面两种情况的延时：a) SRAM互连。b) 过孔连接反熔丝，并比较结果。

9.16 给出为什么会选择使用熔丝连接的FPGA而不用Xilinx的SRAM FPGA的两个理由。

9.17 本题针对课本中描述的XC4010XL FPGA。按照速度给下面的互连类型排队（1表示最高，4为最低）：

四方形长度连线_____

单长度连线_____

直接互连_____

双长度连线_____

9.18 假设一个CMOS芯片设计者在实现芯片时有以下选择：FPGA、门阵列（MPGA）和标准单元（CBIC），这里不考虑性能，对下面类型的设计那种选择的开销最小？（使用课本中的数据）：

a) 一个芯片上有85个门，生产300块芯片？_____

b) 一个芯片上有1500个门，生产9000块芯片？_____

c) 一个芯片上有5000个门，生产200000块芯片？_____

9.19 从门密度和时钟频率两个方面，比较当前门阵列和标准单元技术。

9.20 在标准单元库中通常有低功耗门和高功耗门两种，为什么会有用高功耗门替换低功耗门的情况？

9.21 用集成来说明为什么图9-28的销售损失确实是3500万美元。

9.22 ASIC设计过程的后续步骤是电路提取。该步骤的结果是什么？

9.23 使用Synopsys设计分析程序和LSI 10k库，根据图9-32的脚本对图9-31的计数器描述进行综合。综合时要把时钟周期变为5ns。把得到的有关面积和时间的结果与课本中的进行比较，会得出什么结论？

9.24 对图9-37描述的寄存器加法器，确定它在最高频率时的OTSC。

9.25 在Xilinx PFGA上实现计数器（图9-31）和寄存器加法器（图9-37）。对它所能达到的时钟频率与使用Synopsys及LSI 10K库得到的时钟频率进行比较。

9.26 设计并综合出一个三角波产生器芯片（见图9-57）。

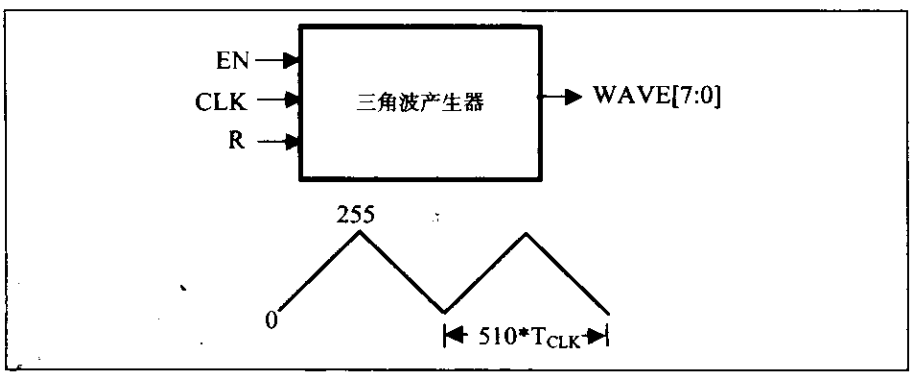


图9-57 三角波产生器

规格说明：设计一个实现三角波产生器功能的芯片。芯片输出有8位精度。当使能端EN=1，输出将由0依斜线升到255，然后再返回到0，并重复这个过程。上升、下降周期为$510*T_{CLK}$，其中T_{CLK}为时钟周期。当复位信号为低时（R='0'），计数器复位并被禁止工作。

a) 为芯片产生一个VHDL行为域模型，使用IEEE_STD逻辑包，包括具有重载+和-的那些包。重载+和-可以用于升降计数的递增和递减。

b) 用VHDL系统来验证此模型。

c) 用Xilinx对模型进行综合，产生两个版本：

1) 慢速时钟版本——用1.4Hz时钟驱动，并用LED观察响应。

2) 快速时钟版本——用12MHz时钟驱动，并用示波器观察波形范围的高阶段的响应，

对于这个版本，使用Xilinx的时序分析工具来确定最大可能的时钟频率。

d) 在Synopsys上综合模型。完成时序分析来决定电路可以运行的最大时钟频率，报告中应具有电路原理图的拷贝，标出原理图中的关键路径，与上面的C.2比较定时分析。哪种电路运行得更快？这是预料中的吗？为什么？与Synopsis一同使用的ASIC库为LSI 10K库，这是旧的CMOS技术，而XC4010XL则是新的FPGA芯片。因此，这种比较并不公平。LSI 500K库中的门延时是相应的LSI 10K库中门延时的五分之一。假设你使用的是LSI 500K库，重新进行比较。

9.27 使用Synopsys系统为习题9.26的模型产生一个OTSC。

9.28 给出了以下触发器数据：

建立时间	5ns
保持时间	2ns
tplh（时钟同步时）	12ns
tplh（时钟同步时）	14ns

图9-58的CL_DELAY为35ns，它与信号变化的方向无关，使电路正常工作的CLOCK信号的最大可能频率是多少？

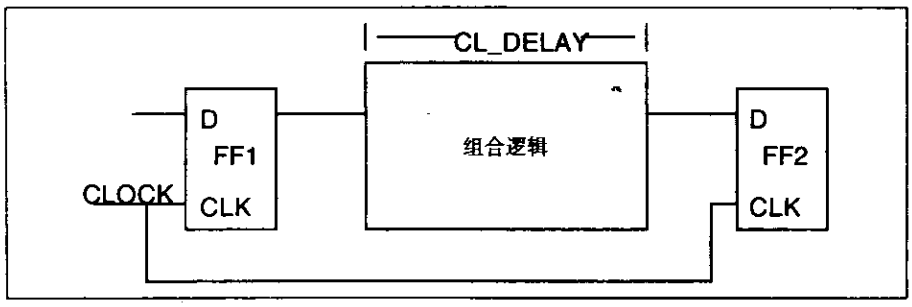


图9-58 习题9.28的框图

第10章 综合建模

本章给出了适合于综合的建模风格，并讨论支持特定建模风格式的综合工具的命令。对于讲述的每一种模型，都给出了用Synopsy或Xilinx软件综合得到的结果电路。

10.1 行为模型的产生过程

图9-29给出了ASIC设计过程的步骤。初始步骤是行为模型的产生和确认，然后直接进入综合。这种最初的行为模型需要为以后对其综合而采用合适的格式。本章中会详细讨论这一过程。

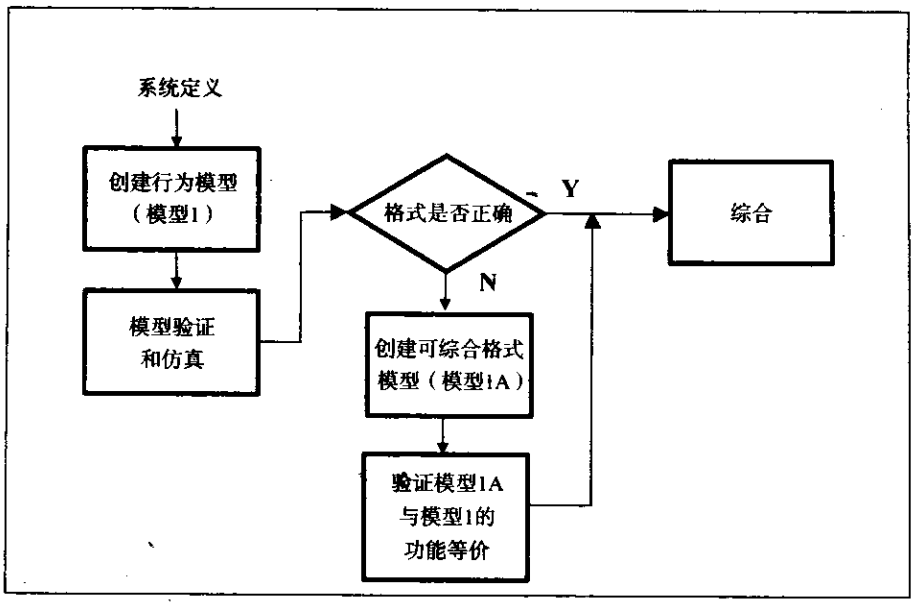


图10-1 设计流程中的建模和仿真

图10-1是一种常用的设计流程图。在许多例子中实际都使用了如下两种模型：

1) 模型1。模型1是含有规格说明的行为模型。这种模型通常称为“可执行说明书”，这是因为它直接对规格进行解释。本章后续部分将详细讨论该模型。读者也可以通过阅读第11章进一步了解此模型的产生。需要指出的，是不能认为综合一种特定的模型就可以代表整个系统。因此，模型1通常只是系统级语义的一个组成部分。第11章也讨论了如何产生这类语义。确认模型的仿真结果被反馈给规格制定者，他回答下面的问题：这就是你的意思吗？如果答案为“是”，则直接进入综合阶段。但在开始综合之前，还必须检查行为模型的风格是否符合综合工具的要求。本章将会看到，并非所有VHDL语言构件都可以被综合——只有那些具有硬件解释的部分才可以被综合。

2) 模型1A。很多例子里的初级行为模型并没有采用正确的格式，这就需要产生另外一种

模型，我们把它称为模型1A。它不仅期望和模型1具有同样的行为，而且期望具有适合使用综合工具的编码格式。模型1和模型1A也带来了新的问题，因为它需要额外的仿真来说明这两个模型具有等价的行为。问题是：为什么它们不能是同一个模型？答案是：产生模型1的目的是为了确认规格说明的正确性。很多例子里，例如DSP，使用一种已被证明正确的计算模型来说明规格。因此，模型制定者试图直接使用VHDL高级构件来实现这种计算模型，而这种构件并不一定可以用硬件说明。也许他还使用了高级数据类型的构件，如与计算模型联系紧密的实型数据结构。他主要关注语言的仿真语义。而在综合的时候，设计者需要考虑语言的综合语义及硬件实现。因此许多系统很难对这两种目标使用同一种模型。

这一问题已经由于“行为编译器”的产生而有所缓解，行为编译器可以将某些高级描述翻译成寄存器传输级的描述，尽管它们还不能转换所有的模型。

10.1.1 初始行为模型的创建

仿真过程的第一步是产生规格说明的行为模型。规格说明可以用自然语言、框图、状态图和时序图表示。对这一步骤的自动化完成已经有了一定的研究。Virginia Tech的研究者开发了一个进程，它可以将计算机系统的英语说明首先翻译成概念图，然后转换为VHDL模型片段。之所以只能翻译为模型片段，是因为英语语法和语义的不确定性。为了弥补这一不足，需要有一种面向英语的说明语言。在工业上，从说明中产生行为模型完全是人工的；但是我们认为基于图形的工具将会越来越多地应用到模型创建这一领域。目前有两种图形工具：“应用域”和“语言域”工具。

10.1.2 应用域工具

在应用域工具里，图的表示主要取决于应用领域。例如，状态图可以用于强控制模型。图10-2是一个模型的顶级状态图，用于产生红外搜索跟踪系统的二维像素帧。状态图允许并发状态，并可以仿真并发控制行为。图10-2中状态FUNC和CLOCK就是并发的。同样，状态嵌套也是被允许的，这样就可以避免“状态爆炸”的出现。例如，状态IDLE_CLK、ST_CLK及EN_CLK嵌套进了状态RUN_CLK。名字前有@标志的状态表示高级状态，它表示含有一个下级状态图。例如，INIT、RUN_0及RUN_1都是含有下级状态图的高级状态。状态变换由事件或时间流逝所触发。除了状态图以外，还需要填充一段代码模板来定义状态变换发生时的函数调用。

图10-3是函数DO_ANG_CALK的模板，当INIT状态被调用时它也被调用。

状态图模型可以在图形级别上进行仿真。可以使用代码产生界面来形成C或综合化的VHDL、Verilog语言代码。

数据流模型同样可以被图形化。图10-4是可综合的合成孔径雷达的模型，它是在Cadence的信号处理工作台（SPW）上产生的。该模型可以在图形级别上进行仿真，并且输出的FFT可以用来验证模型的正确性。代码产生接口可以产生等价的VHDL行为模型。图10-4中的复杂信号产生数据文件可以用来作为输出以测试低级别的模型。

图10-5是从雷达接受的下转信号，图10-6是FFT的输出信号。它具有一个单一频率的最大响应。FFT信号归一化为采样频率，但是有一个关联读出给出了作为规定频率合理范围的峰值频率。

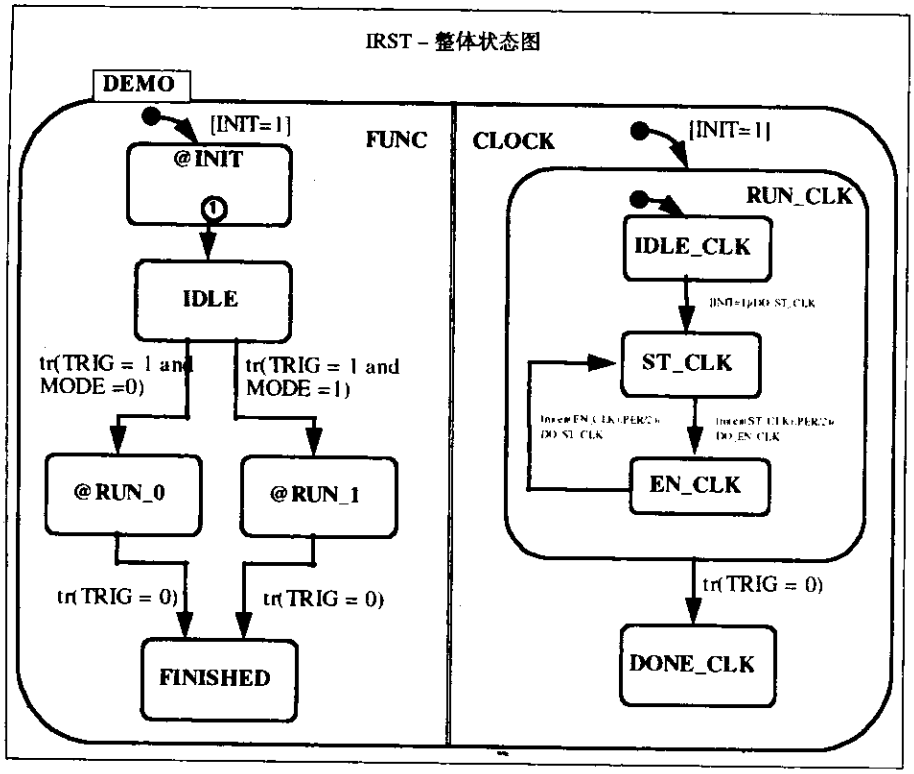


图10-2 IRST的整体状态图

```
ACTION DICTIONARY
Project: IRST

DO_ANG_CALC
Defined in chart: INIT
Definition: TAR_XCOORD:=XCOORD;
            TAR_YCOORD:=YCOORD;
            if(MODE_MOTION=1)then
               RAD_ANGLE:=0;
            end if;
            if(MODE_MOTION=2)then
               RAD_ANGLE:=(90*3.14)/180;
            end if;
            if(MODE_MOTION=3)then
               RAD_ANGLE:=(ANGLE*3.14)/180;
            end if;
            R_VECTOR:=VELOCITY;
```

图10-3 表示VHDL代码模板的例子

SPW是特定于DSP的。更一般的数据流系统可以用活动图建模并用Ilogix Express VHDL实现。图形表示，如流程图、框图，都可以从高层中获取，并作为可视化HDL来表示这些系统。

图形系统的优点在于模型可以建立在高级别上，特别是数字信号处理（DSP）模型使用已被证明了的数学系统。因此建模过程不需要硬件描述语言的细节知识，模型创建者可以直

接在应用域建模，在模型验证正确以后，自动产生C、VHDL或Verilog代码。

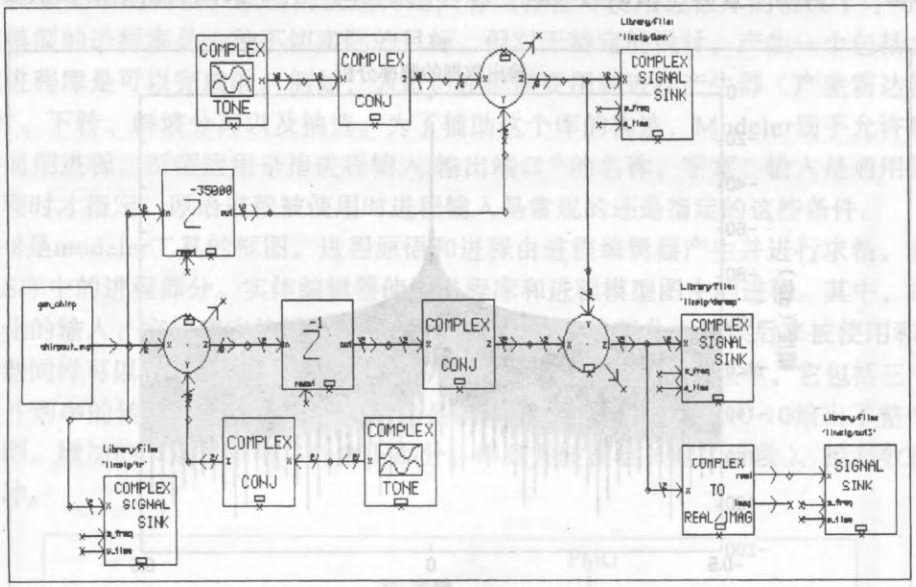


图10-4 一个可综合过孔雷达的数据流模型

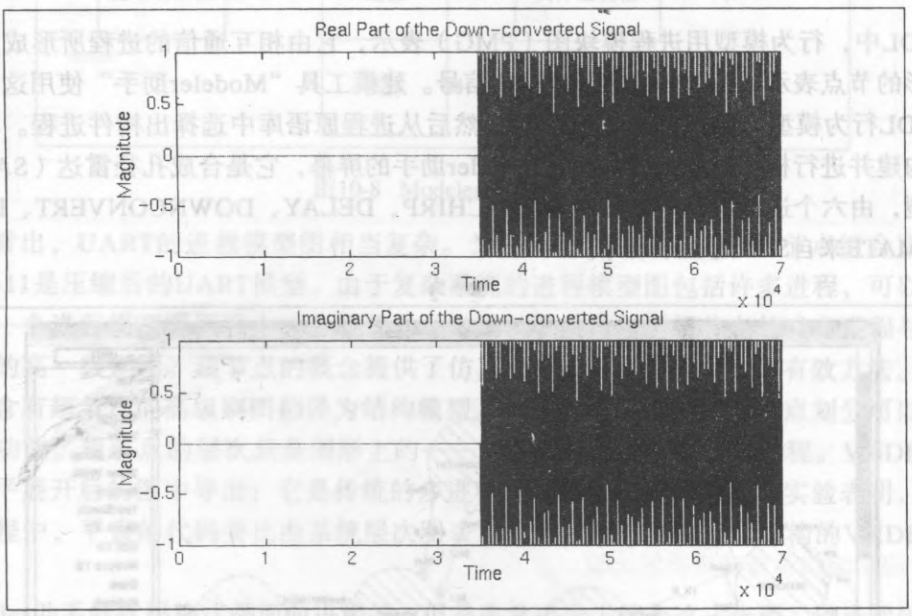


图10-5 接收的下转信号

10.1.3 语言域建模

同样可以在语言域建立模型。该域被CAD工程师和设计者所使用，他们需要对模型开发进行控制。这些人必须具备有关目标语言的知识。他们可以使用标准编辑工具生成目标语言代码的源文件。模型创建者可以使用“上下文相关编辑器”来检查VHDL语法。大多数情况

都是使用这种直接编辑的方法，它的缺点在于复杂模型的代码量太大而且很容易产生错误。因此，期望可以使用一种能产生模型的图形表示语言域工具，以便控制模型的复杂度。

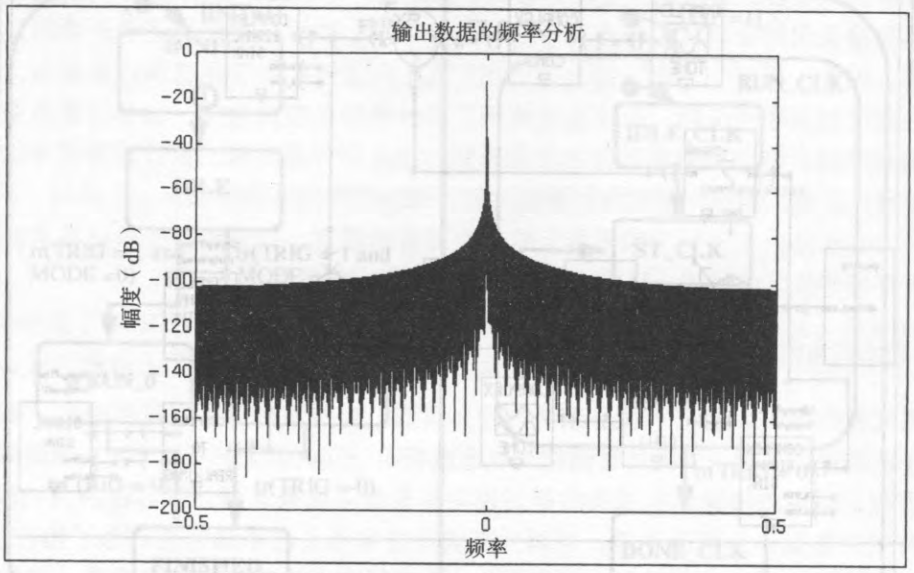


图10-6　坡度分立后信号的FFT

VHDL中，行为模型用进程模块图（PMG）表示，它由相互通信的进程所形成的网络构成。图形的节点表示进程，连线表示VHDL信号。建模工具“Modeler助手”使用这种表示来产生VHDL行为模型。用户首先进入PMG，然后从进程原语库中选择出构件进程。模型可以被快速构建并进行构件纠错。图10-7是Modeler助手的屏幕，它是合成孔径雷达（SAR）传感器的模型，由六个进程组成，进程原语GENCHIRP、DELAY、DOWNCONVERT、DERAMP及DECIMATE来自SAR进程原语库。

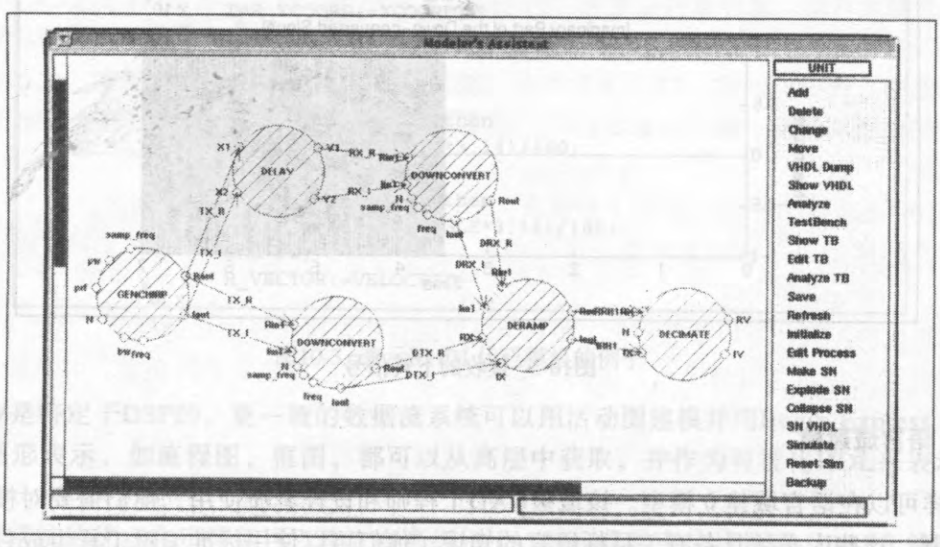


图10-7　SAR传感器模型

该系统提供了进程级的代码重用，在传统VHDL软件中并没有这种兼容性。模型原语可以有比整个模型更细的粒度。模型的快速构建只有当存在可使用进程库的情况下才可能实现。产生所有模型的进程库是一种不切实际的目标，但对于特定的设计，产生一个包括大多数模型功能的进程库是可以完成的。例如，为雷达建模需要用到进程产生器（产生雷达发出的信号）、延时、下转、斜坡分离以及抽选。为了辅助这个库的构造，Modeler助手允许建立进程原语，即通用进程。所谓运用是指进程输入/输出端口[⊖]的名称、字宽、输入是通用的，在使用原语进程时才指定。原语进程被使用时进程输入是常规的还是指定的这些条件。

图10-8是modeler工具的框图。进程原语和进程由进程编辑器产生并进行求精，它们存储在MODAS库中的进程部分。实体编辑器使用进程库和进程模型图中的进程。其中，进程模型图是图形化的输入，它构建实体模型并将其存储在库的实体部分中以供后来被使用和被优化。实体编辑器同样可以产生VHDL代码。图10-9显示的是实体编辑器的菜单，它包括三个进程模型，完成并到串的转换。并到串的转换电路是UART的输出部分。图10-10给出了整个串口的进程模型图。增加的节点表示串口的余下部分：串到并转换器（输入阶段）、控制数据到数据总线的缓冲。

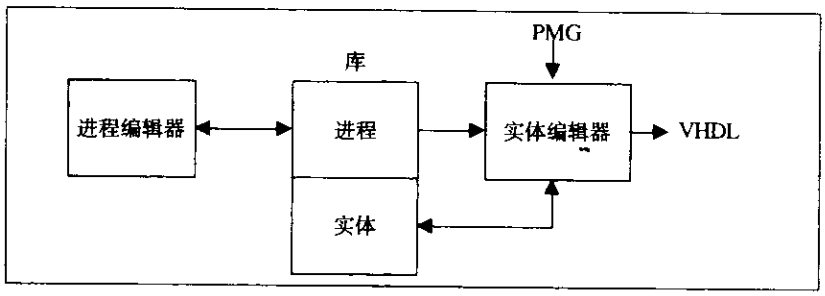


图10-8 Modeler助手框图

可以看出，UART的进程模型图相当复杂。为了简化复杂度，可以将节点组合成为超节点。图10-11是压缩后的UART模型。由于复杂系统的进程模型图包括许多进程，可以使用这种机制将一个进程模型图压缩为一个超节点，它可以形成只包括超节点的或由进程与超节点混合而成的高一级别图。超节点的概念提供了仿真VHDL语言行为层次的有效方法。程序也可以将只含有超节点的高级别图翻译为结构模型。因此，通过不同的超节点划分可以研究不同部分的功能。超节点的层次只是图形上的——VHDL中进程不能嵌套进程。VHDL代码从超节点水平展开后的图中导出；它是传统的多进程、单构架的行为模型。实验表明，在仿真和综合过程中，平直的代码要比由系统层次图表示映射过来的具有层次结构的VHDL模型更加有效。

Modeler助手的代码格式是面向进程的，但是非常适合于综合这一步骤，该步骤也是面向进程的。

10.1.4 建模及模型效率

当用工具产生一个模型而非手工建模时，知道模型产生过程的效率是非常重要的。在图10-2 IRST模型中（仅是顶级状态图）的复杂数据是：1) 状态图：43个节点和56条边。2) 模板：

⊖ 因为进程是源自模型的进程模型图的模块，所以为进程定义输入和输出端口。当产生模型VHDL时这些端口就会消失。

文本520行。3) VHDL（自动产生的）：948行。4) 编程时间：大约一个月，这是由于建模人员学习了Ilogix软件及IRST规格说明。如果假设之前建模人员已经具备了两者的知识，这种复杂度的模型的创建时间不会超过一个星期。这个例子里，编码的工作量比较小，但模型检测在高级别上要快得多。

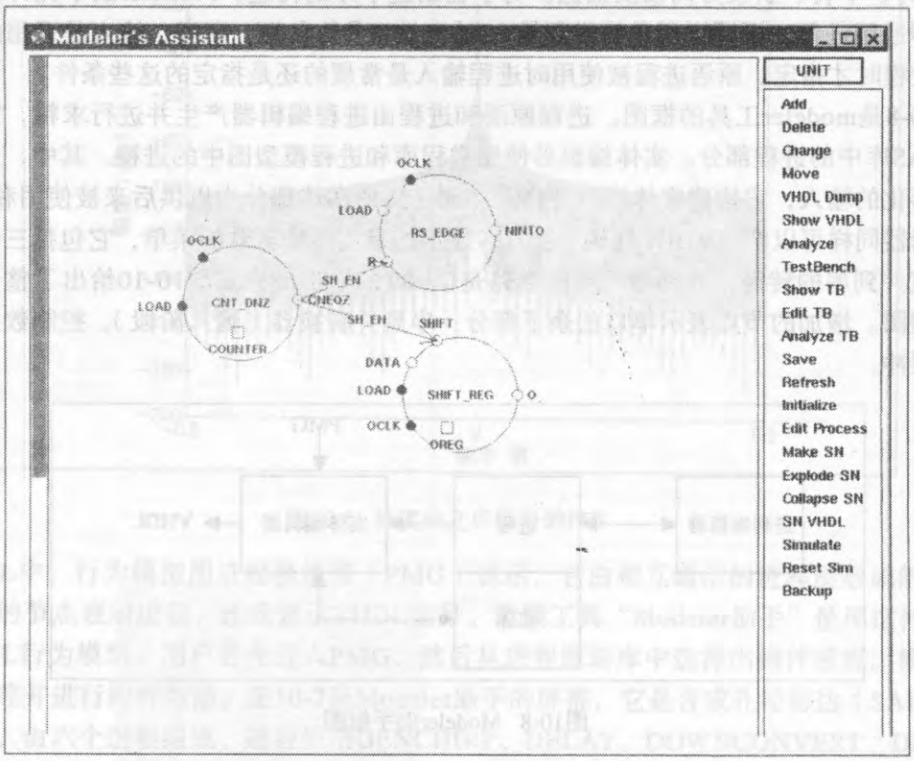


图10-9　Modeler助手的进程模型图

对于高级别建模工具产生的代码，代码效率的考虑有两个方面：1) *仿真效率*，即代码仿真的速度。2) *综合效率*，即综合的单位数是多少？Virginia Tech的研究表明手工建模产生代码的仿真效率是高级工具生成代码仿真效率的三倍，而且综合时减少了20%的单元。比较的结果依赖于所使用的工具及进行手工编程的人员。其中的关键在于监视高级工具产生的代码的效率。

10.1.5　应用域和语言域建模的比较

应用域模型由设计者完成，他不考虑具体的语言细节，设计者熟悉示意性的图形表示。应用域中的图形表示（如状态图）是一种被设计者所熟悉的语法。该域中底层的数学知识已经被熟知并被很好地定义了。他们可以在该领域中快速验证模型。然而，应用域模型的进化比手工编码的效率要低。而且，由于模型由系统的数学知识产生，例如DSP，它们需要使用不能被直接综合的语言构件。然而，通过工具中提供的代码接口来产生VHDL代码可以缓解这一问题，该代码可以具有不同的格式以供综合工具使用。

另外一种办法是提供覆盖不同应用域的工具。DSP域可以被高级建模工具覆盖，这是由

于很多的ASIC设计都可以进行信号处理。然而，大多数建模使用的工具并不能在高抽象级别上提供很好的支持。这个问题已经引起了注意。一些系统已经综合了控制流和数据流建模的能力。例如，Ilogix也可以通过活动图为数据流系统建模。高层设计工具的可视化VHDL也可以用框图、流程图、状态图和状态表为控制流和数据流行为建模。

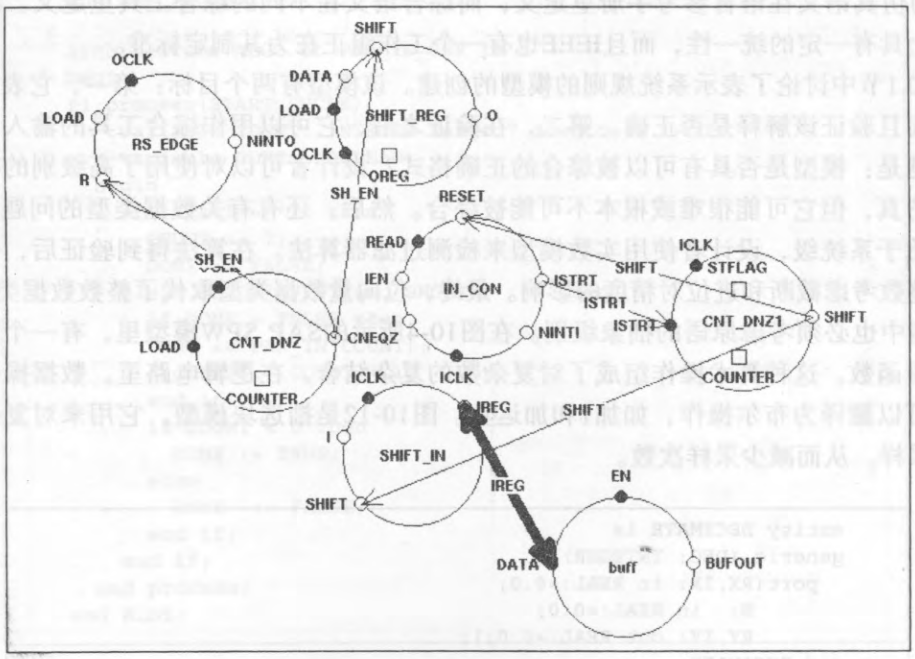


图10-10 完整UART的PMG

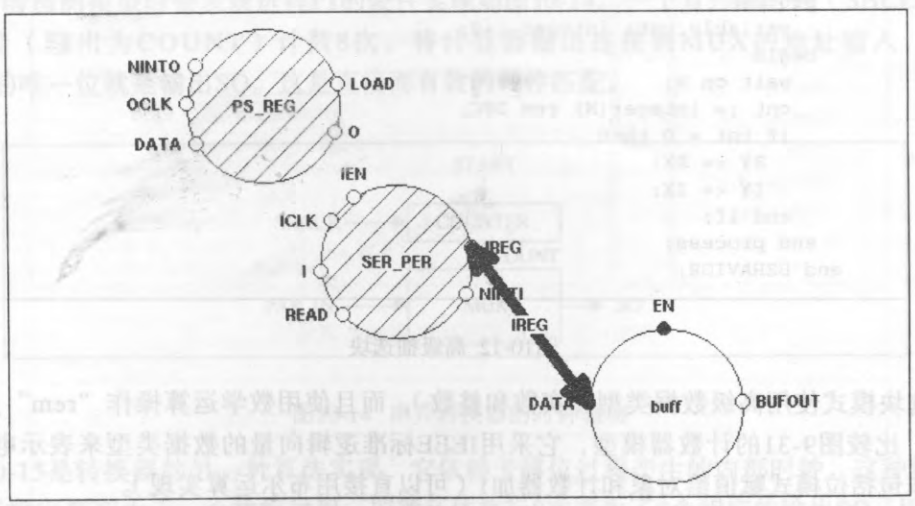


图10-11 UART模型的超节点

对于语言域建模，设计者必须熟悉语言构件，如VHDL进程。CAD工程师和设计者接受硬件描述语言的训练后可以在语言域建模。在这一领域中模型的进化需要更多的时间，但由于语言域模型直接使用语言，该域中的模型构件在仿真和综合时比应用域模型更加有效。模

型的格式是人为控制的，即设计者可以根据他期望使用的综合工具来选择确定模型的格式。

10.2 仿真和综合的语义

本节讨论VHDL综合建模。关键在于VHDL有一种仿真语义集合和另外一种综合语义集合。VHDL的仿真语义在语言参考手册里定义，而综合语义在不同的综合工具里定义。综合语义在工业上具有一定的统一性，而且IEEE也有一个工作组正在为其制定标准。

在10.1节中讨论了表示系统规则的模型的创建。该模型有两个目标：第一，它表示规范的解释，而且验证该解释是否正确。第二，在验证之后，它可以用作综合工具的输入。这里的关键问题是：模型是否具有可以被综合的正确格式？设计者可以对使用了高级别的模型进行很好的仿真，但它可能很难或根本不可能被综合。然后，还有有关数据类型的问题。DSP的应用开始于系统级，设计者使用实数模型来检测过滤器算法。在算法得到验证后，使用固定长度的整数考虑截断和进位对精度的影响。最终，位向量数据类型取代了整数数据类型。

模型中也必须考虑原语的抽象级别。在图10-4所示的SAP SPW模型里，有一个复杂结合器的原语函数。这种算术操作组成了对复杂数的复杂结合。在逻辑电路里，数据操作是布尔或至少可以翻译为布尔操作，如加1和加运算。图10-12是抽选块模型，它用来对复杂实数数据重新采样，从而减少采样次数。

```
entity DECIMATE is
generic (DEC: INTEGER);
  port(RX,IX: in REAL:=0.0;
       N:  in REAL:=0.0;
       RY,IY: out REAL:=0.0;);
end DECIMATE;
architecture BEHAVIOR of DECIMATE is
begin
  process
    variable cnt: integer :=0;
  begin
    wait on N;
    cnt := integer(N) rem DEC;
    if cnt = 0 then
      RY <= RX;
      IY <= IX;
    end if;
  end process;
end BEHAVIOR;
```

图10-12 高级抽选块

抽选块模式使用高级数据类型（实数和整数），而且使用数学运算操作“rem”达到期望的结果。比较图9-31的计数器模型，它采用IEEE标准逻辑向量的数据类型来表示电路状态。数据操作包括位模式赋值给对象和计数器加1（可以直接用布尔运算实现）。

下面考虑另一个建模风格起影响作用的例子。图10-13是时钟控制的并串转换器模型。

当接收到START信号时，COUNT初始化为7，信号DONE置为FALSE。当SHCLK上升时，如果DONE为FALSE，选择并行输入向量的一位，将其传输到输出SO。然后COUNT减1。如果COUNT变为负，DONE置为TRUE。其中COUNT是具有限制范围的整数。如果使用的是整数类型，综合工具将在实现计数器时使用不必要大的整数。

```
library IEEE;
use IEEE.std_logic_1164.all;
entity PAR_TO_SER is
  port(START,SHCLK: in STD_LOGIC;
       PAR_IN: in STD_LOGIC_VECTOR(7 downto 0);
       SO: out STD_LOGIC);
end PAR_TO_SER;

architecture ALG1 of PAR_TO_SER is
begin
P1:process(START,SHCLK)
    variable COUNT: INTEGER range 7 downto -1 := 0;
    variable DONE: BOOLEAN;
  begin
    if  START = '1' then
      COUNT := 7;
      DONE := FALSE;
    elsif SHCLK'EVENT and SHCLK = '1' then
      if DONE = FALSE then
        SO <= PAR_IN(COUNT);
        COUNT := COUNT - 1;
      end if;
      if COUNT < 0 then
        DONE := TRUE;
      else
        DONE  := FALSE;
      end if;
    end if;
  end process;
end ALG1;
```

图10-13 时钟驱动的串并转换器

设计者检测模型时会发现进程P1的硬件实现如图10-14。一个有外部时钟（SHCLK）驱动的计数器（输出为COUNT）计数8次。将计数器输出连接到MUX的地址输入，寄存器PAR_IN的唯一位就是输出SO。这是直接而有效的硬件匹配。

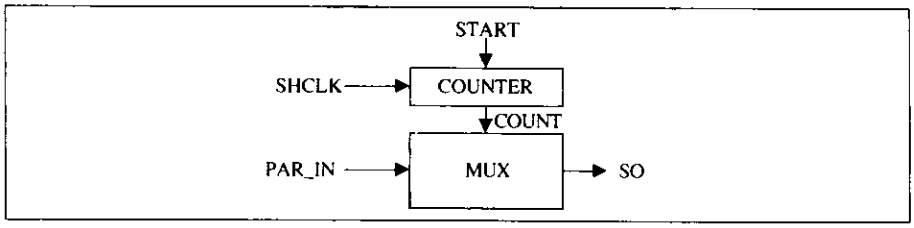


图10-14 串并转换器的时钟驱动

图10-15是转换器的另一种算法实现，它依赖于移位过程产生的内部时钟。这种情况下的所有串行输出都产生于一个执行过程，即循环体执行8次产生了8个相应的输出SO。图10-16是两个转换器的仿真结果。START_SCHED触发预定的转换器，SO_SCHED是它的结果。除了初始信号时钟不同，两个模型的结果是完全相同的。可以认为预定模型要更高级一些，这是因为它是自包含的并且不需要外部时钟。然而，现在让我们考虑由这个模型隐含的硬件。如图10-17所示，隐含的硬件包括一个计数器，它控制一个调度器以便将8个事件插入一个时间

序列。这些事件由一个事件处理器连续地处理以产生相应的输出SO。从仿真的角度来看，这不会产生问题，因为仿真程序处理调度、时间序列以及事件处理功能。然而从综合的角度来看，相对于直接匹配的硬件模型，用硬件实现预定模型的这些功能非常复杂，而且是不必要的。实际中通用的综合工具并不能对预定转换器进行综合，这是由于电路时钟由赋值语句段的*after*语句的延时所控制，该语句将预定的结果赋值给输出SO。

```
use IEEE.std_logic_1164.all;
entity PAR_TO_SER_SCHED is
  generic(PERIOD: TIME);
  port(START: in STD_LOGIC;
       PAR_IN: in STD_LOGIC_VECTOR(7 downto 0);
       SO: out STD_LOGIC);
end PAR_TO_SER_SCHED;
architecture ALG2 of PAR_TO_SER_SCHED is
begin
P1:process(START)
    variable COUNT: INTEGER;
  begin
    if  START = '1' then
     COUNT := 7;
     while COUNT >= 0 loop
      SO <= transport PAR_IN(COUNT) after (7-COUNT)*PERIOD;
      COUNT := COUNT - 1;
     end loop;
    end if;
  end process;
end ALG2;
```

图10-15　调度串并转换器

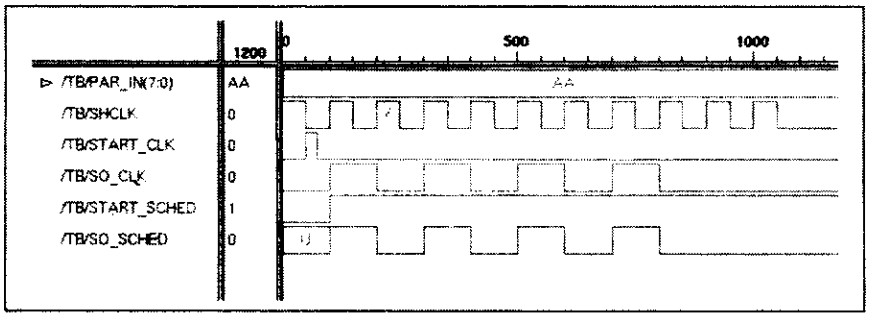


图10-16　时钟和调度电路的仿真响应

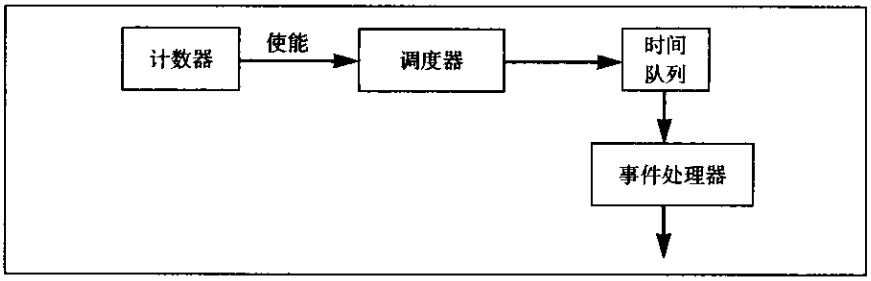


图10-17　调度电路的硬件实现

仿真和综合的另一个语义区别在于进程感应表的使用不同。对于仿真，进程的感应表决定哪一种信号触发进程的执行。对于综合，进程感应表用于以文件形式罗列出进程的输入，它对于综合不会产生影响。下面用几个T触发器模型的例子来说明这种情况。图10-18是第一个模型。由于模型需要读出自身的状态并进行切换，所以使用了内部信号Q作为触发器的内部状态，电路输出信号是QOUT。Q的值使用进程体外的一条信号赋值语句传递到QOUT。图10-19的模型大体上与图10-18的模型相同，不同的是Q到QOUT的赋值发生在进程体内。图10-20是这两个模型的仿真结果。Q1是图10-18模型的输出，Q2是图10-19模型的输出。输出Q2不同于Q1的原因在于：第二个模型的进程体在信号T变为‘1’时被调用，Q在一个delta时钟之后才会发生改变。因此，QOUT只有在该时钟结束之后的下一次调用进程体时才会得到与第一个模型相同的值。第二个模型可以将Q加入进程感应表中。Q3是模型改变后的输出。综合过程并不使用感应表，因此，当图10-18和图10-19的描述被综合时，都能得到图10-21所示的电路。这个例子说明具有不同仿真结果的两个模型可能会被综合为同一个电路。因此，在选择需要加入进程感应表的信号时要非常小心。

```
entity T_FF is
  port(RESET,T,CLK: in STD_LOGIC; QOUT: out STD_LOGIC);
end T_FF;
architecture ALG of T_FF is
  signal Q: STD_LOGIC;
begin
  process(RESET,T,CLK)
  begin
    if (RESET = '1') then
      Q  <= '0';
    elsif (CLK'EVENT and CLK = '1') then
      if T = '1' then
        Q  <=  not Q ;
      end if;
    end if;
  end process;
  QOUT <= Q;
end ALG;
```

图10-18 可以正确仿真的T触发器

```
entity T_FF2 is
  port(RESET,T,CLK: in STD_LOGIC; QOUT: out STD_LOGIC);
end T_FF2;
architecture ALG of T_FF2 is
  signal Q: STD_LOGIC;
begin
  process(RESET,T,CLK)
  begin
    if (RESET = '1') then
      Q  <= '0';
    elsif (CLK'EVENT and CLK = '1') then
      if T = '1' then
        Q <=  not Q ;
      end if;
    end if;
    QOUT <= Q;
  end process;
end ALG;
```

图10-19 不能正确仿真的T触发器

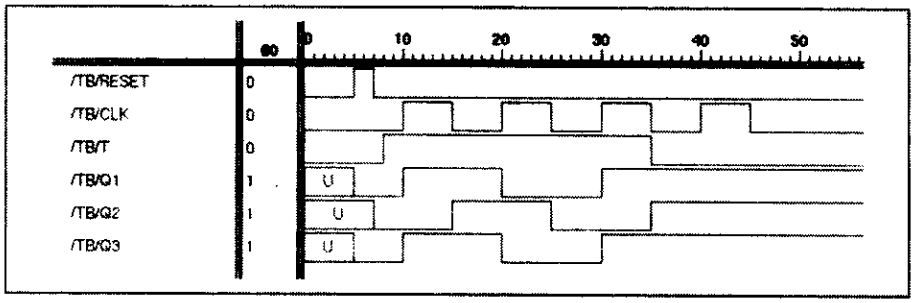


图10-20 T触发器模型的仿真响应

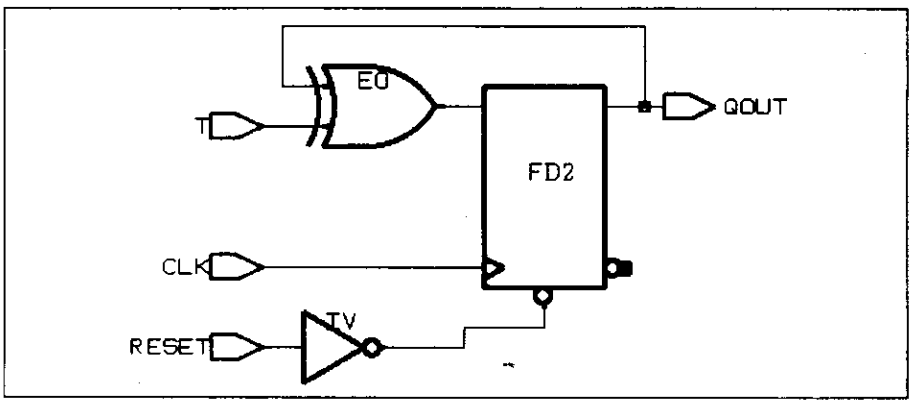


图10-21 T触发器的综合电路

10.2.1 模型中的延时

在上一节中讨论了仿真和综合语义的不同。建模的另一个重要方面是模型延时。在编写代码时会经常写出如下语句：

```
C<=A and B after 10 ns;
```

该语句意味着10ns延时之后进行AND操作。仿真时该代码可以顺利执行，而在综合时要让综合工具实现10ns的延时则会遇到很大的困难。综合工具能够实现功能，也能够实现时钟行为，但要实现一个精确或近似的延时值则非常困难。它所能够做到的只是实现运算AND，而且只能使用在综合器件时使用的ASIC库中的延时。实际上大多数的综合工具都要求只能使用delta延时，因此只能是：

```
C<=A and B;
```

同样，语句“wait for 10ns”在综合时是没有意义的。只能等待事件，如“wait until clock='1';”。

10.2.2 数据类型

模型的数据类型被限制为STD_LOGIC、STD_LOGIC_VECTOR及限定整数。大多数综合工具是为STD_LOGIC类型而设计的。尽管许多综合工具允许源文件中使用类型BIT及BIT_VECTOR，综合后电路依旧使用类型STD_LOGIC和STD_LOGIC_VECTOR。因此，为

了验证综合后的电路，要将结果与不同的数据类型进行比较。此外，所有的IEEE包都采用STD_LOGIC、STD_LOGIC_VECTOR及限定整数类型，以供综合时使用。

10.3 为时序行为建模

综合时序行为是一个有意思且富有挑战性的问题，它需要使用很多方法。在一些HDL如AHPL中，触发器和寄存器都被声明为存储器，存储器赋值语句要使用一种特殊的符号（<=）。这些赋值语句都被假设为同步执行，仅在系统时钟上升或下降沿执行。混合逻辑电路的输出赋值则称为“连接”，它用符号（:=）表示。因此，语言构件就决定了哪一种是时序电路，哪一种是混合电路。这简化了综合的工作，但增加了编写代码的工作。例如，如果所有的存储器赋值语句都是定时的，模型的异步存储设备以及锁存行为如何表示？另一种可能是使用名称来标识同步执行的元件。例如X_CLK标识X是时钟信号，进程A用名字A_FF标示。然而，这种方法也限制了设计者，并需要在语言本身引入命名标准。这两种方法在VHDL综合时都没有被采用。而是由基本语言构件以某种方法结合，以隐含时序行为。

综合时序行为有三个基本概念：1) 确认时序控制机制。2) 决定将被存储的值。3) 初始化电路。为了确认时序控制机制，VHDL将时钟边界定为时钟事件CLK='1'（正向传输）或时钟事件CLK='0'（负向传输）。在STD_LOGIC_1164包中定义了RISING_EDGE和FALLING_EDGE，用于检查这两种条件。RISING_EDGE的代码如下：

```
function RISING_EDGE (signal S: STD_ULOGIC)
  return BOOLEAN is
begin
  return (S'event and (To_X01(S) = '1')
                  and (To_X01(S'last_value) = '0'));
end;
```

函数TO_X01(S)将S的值匹配到X、0、1。匹配过程是：

```
('X', 'Z', 'U', '-') map to 'X'.
('1', 'H') map to '1'.
('0', 'L') map to '0'.
```

因此，函数RISING_EDGE在'0'或'L'变为'1'或'H'时返回值为TRUE。

边界表达式（或函数调用）使用了if或wait语句。if语句具有如下格式：

```
if (RESET = '1') then -- alternatively,(RESET = '0')
    -- perform asynchronous reset and initialization
elsif (CLK'event and CLK = '1') -- alternatively CLK = '0'
    -- load flip-flops and registers
end if;
```

图10-13、图10-18及图10-19都使用了这种方法。通常异步复位及时钟活动可以简化为对寄存器载入一个值，或者由寄存器载入引发更复杂一些的数据操作。它需要注意两点：识别时序控制机制和初始化电路。其中一个关键点是大部分的综合工具都不允许在时钟沿条件上加入其他条件，如CLK'EVENT and CLK='1' and EN='1'是无效的。综合工具需要有独立的时钟沿条件。如果有使能条件EN的话，则需要在elsif范围里测试。这防止了由于设计不当而造成的门时钟。如果EN在*elsif*语句里，EN将应用于时钟控制触发器的使能输入。图10-18中的T触发器模型遵循这个建模过程。

下面考虑使用wait语句测试时钟边界的可能：

```
wait until (CLK'EVENT and CLK = '1');
-- load flip-flops and registers
```

它没有内部复位机制，因此，模型仅限制在同步活动。同步复位依赖于前面的代码或由“upstream”逻辑来产生数据输入的复位条件。实际上，经常使用一个进程表示单纯的组合逻辑，加上一条*wait*语句，由它来驱动单纯的时序进程。最后，只有那些没有感应表的进程才会再次调用*wait*语句。

为了进一步说明*wait*语句的问题，考虑一个时序电路，它执行在等价条件下切换的功能。当单位输入I在三个连续时钟周期有相同值时，输出TEQ进行切换。图10-22是使用*wait*语句检测时钟活动的检测器的初始模型。输入的初始值存储在变量IBK1和IBK2中。该模型在输出TEQDET初始化为‘0’或‘1’的情况下仿真正常。如果输出开始值为‘U’，则它在整个仿真过程中不发生变化。而在模型综合时，综合工具不考虑端口状态的初始值，这里使用*wait*语句不能解决此问题。模型需要有一个显式的复位语句。图10-23是一个使用*if-then*语句初始化电路的模型，CLK’event构件用来确认时钟机制。这种显式复位解决了初始化的问题。图10-24显示了综合后的电路。图10-25是它的仿真结果。图10-24中的变量IBK1和IBK2寄存I延时后的值。*elsif*语句由时序控制，综合工具寄存了所有的输出信号和变量，它们在更新前被读出。因此，IBK1和IBK2被寄存，而EQ没有。同样，代码中IBK1和IBK2的更新顺序是临界的。如果改变了它们的顺序：

```
IBK1 := I;
IBK2 := IBK1;
```

则调用该过程时，IBK1和IBK2的值完全相同，综合工具产生的电路将会是时钟I直接输入到这两个触发器。

```
entity EQDET is
  port(I,CLK: in STD_LOGIC; TEQDET: inout STD_LOGIC :='0');
end EQDET;

architecture ALG of EQDET is
begin
  process
    variable  EQ,IBK1,IBK2: STD_LOGIC;
  begin
    wait until (CLK'EVENT and CLK = '1');
    if(IBK1 =IBK2) and (IBK2 = I) then
      EQ := '1';
    else
      EQ := '0';
    end if;
    TEQDET <= (EQ xor TEQDET);
    IBK2 := IBK1;
    IBK1 := I;
  end process;
end ALG;
```

图10-22　初始等价检测器模型

图10-26是基于信号的模型，IBK1和IBK2用信号表示。这里，更新IBK1和IBK2的顺序与

自动寄存的时序控制赋值语句相比就没有那么重要了。基于信号模型（图10-26）与基于变量模型（图10-23）综合后形成的电路相同。由于基于变量模型在行为级上的仿真效率要高些，选择时一般会被优先考虑。

```
entity EQDET is
  port(RESET,I,CLK: in STD_LOGIC; TEQDET: inout STD_LOGIC);
end EQDET;

architecture ALG of EQDET is
begin
  process(RESET,CLK)
    variable  EQ,IBK1,IBK2: STD_LOGIC;
  begin
    if (RESET = '1') then
      IBK1 :=  '0';
      IBK2 :=  '0';
      TEQDET <= '0';
    elsif (CLK'EVENT and CLK = '1') then
      if (IBK1 = I) and (IBK1 = IBK2) then
        EQ := '1';
      else
        EQ := '0';
      end if;
      TEQDET <= (EQ xor TEQDET);
      IBK2 := IBK1;
      IBK1 := I;
    end if;
  end process;
end ALG;
```

图10-23 在*if-then*结构中使用复位的等价检测器

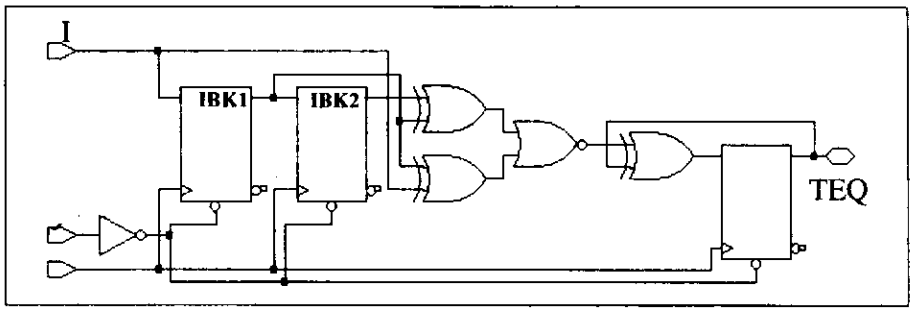


图10-24 综合后的检测器

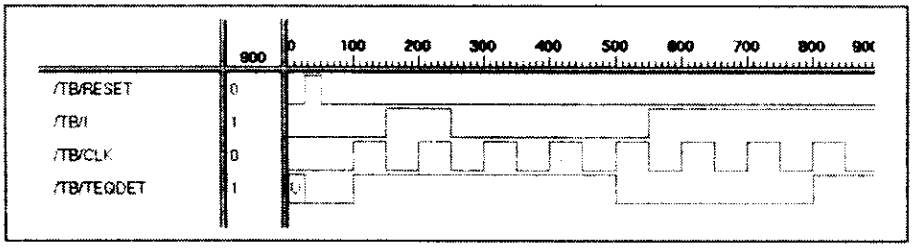


图10-25 检测器的仿真响应

```
entity EQDET is
  port(RESET,I,CLK: in STD_LOGIC; TEQDET: inout STD_LOGIC);
end EQDET;

architecture ALG of EQDET is
  signal IBK1,IBK2: STD_LOGIC;
begin
  process(RESET,CLK)
    variable  EQ: STD_LOGIC;
  begin
    if (RESET = '1') then
      IBK1 <=  '0';
      IBK2  <=  '0';
      TEQDET <= '0';
    elsif (CLK'EVENT and CLK = '1') then
      if (IBK1 = I) and (IBK1 = IBK2) then
        EQ := '1';
      else
        EQ := '0';
      end if;
      TEQDET <= (EQ xor TEQDET);
      IBK1 <= I;
      IBK2  <= IBK1;
    end if;
  end process;
end ALG;
```

图10-26 基于信号的模型

进一步研究复位问题。由于异步复位易于完成状态的初始化，它被优先选择。同步复位只有在同步寄存器的输入可以被控制时才是有效的。对于检测器电路，除了修改模型以外别无选择。

对检测器的另一种设计方法是使用一种有限状态机（FSM），图10-27是其状态图。状态定义如下：1) S0：初始状态；2) S1：接受输入后进入的状态；3) S2：接受输入后，并且当前输入与前一输入具有相同输入值时进入的状态。事件EQ表示当前输入与前一输入相同，NEQ的意思则相反。当进入S2状态后，若当前输入等于前一输入，信号TEQ在时钟周期结束时切换。

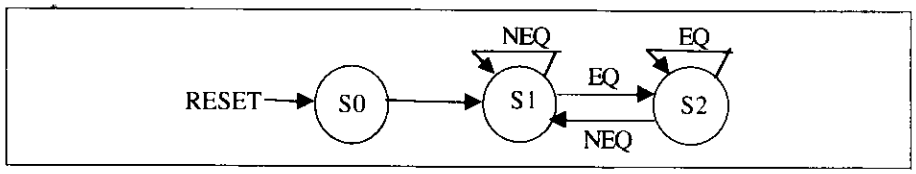


图10-27 检测器的状态图

图10-28是FSM的VHDL模型。状态用枚举类型表示。一条*case*语句控制时钟状态移动，每个状态的行为在*case*语句中的*when*语句里定义。*when*语句包括两个部分：1) 改变为下一状态。2) 数据和输出操作。图10-29是一个同步电路，触发器的左部分存储变量IBK1，右部分存储电路输出TEQOUT，中间的两个触发器实现状态机。这种FSM实现比图10-23和图10-26的模型开销要大，那两种模型在原理上只需要一个触发器就可以了。一般来讲，有限状态机

是一种为强控制电路建模的非常有效的工具。这种方法已经在第8章详细介绍过了。

```
entity EQDET is
  port(RESET,I,CLK: in STD_LOGIC; TEQDET: inout STD_LOGIC);
end EQDET;
architecture FSM of EQDET is
begin
 P1:process(RESET,CLK)
    type STATE_TYPE is (S0,S1,S2);
    variable STATE: STATE_TYPE;
    variable IBK1: STD_LOGIC;
  begin
    if RESET = '1' then
      STATE := S0;
      IBK1 := '0';
      TEQDET <= '0';
    elsif (CLK'EVENT and CLK = '1') then
      case (STATE) is
        when S0 =>
          STATE := S1;
          IBK1 := I;
        when S1 =>
          if (IBK1 = I) then
            STATE := S2;
          else
            STATE := S1;
          end if;
          IBK1 := I;
        when S2 =>
          if (IBK1 = I) then
            STATE := S2;
            TEQDET <= not TEQDET;
          else
            STATE := S1;
          end if;
          IBK1 := I;
      end case;
    end if;
  end process;
end FSM;
```

图10-28 检测器的FSM模型

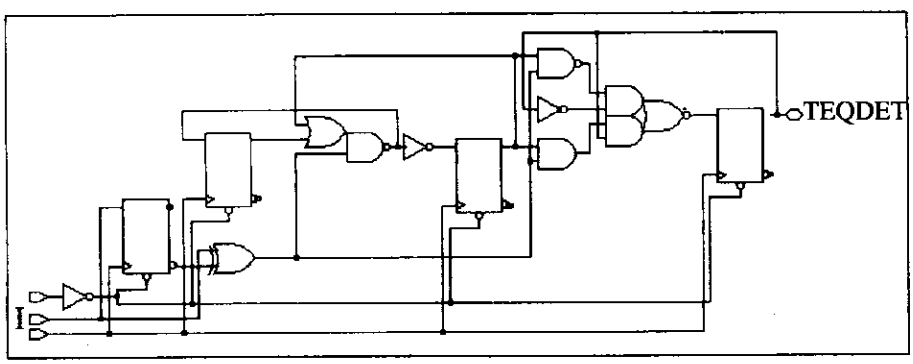


图10-29 检测器FSM综合后的电路

下面列出在为时序电路综合建模时还需要注意的地方：

1) 由于大多数综合工具不考虑声明语句的初始值，所以不要在声明语句里对信号和变量置初始值。这些初始值只会被仿真程序所使用；因此，模型在仿真时可能是正确的。然而，由于综合程序不使用这些值，最终形成的硬件可能没有被正确地初始化。例如，不要使用如下的初始化语句：

variable SUM : INTEGER := 0;

而是在代码段中显式赋值给SUM：

SUM := 0;

2) 不要使用没有制定界限的整数。例如，建模者在用一个整数表示计数器时，可以声明为：

signal COUNT : INTEGER;

由于类型INTEGER依据主机的字长可以有一个很大的可能值范围，综合工具将使用一个大寄存器存放该值，现在机器中普遍采用32位。由于读写该寄存器需要有32位数据通路，这是一个很大的开销。解决办法是使用标定界限的整数来表示COUNT：

signal COUNT : range 0 to 15;

声明一个标定界限的整数可以使得综合工具减少寄存器和数据通路的大小，使其正好在数据大小的范围之内。

3) 在有限状态机模型里使用属性来决定状态。图10-28所示的有限状态机模型里，枚举类型的状态分别为S0、S1和S2。问题是它们在综合工具里如何表示？可以使用二进制编码：S0-00、S1-01、S2-10；或者是增加“独热”码：S0-001、S1-010、S2-100。图10-28模型的实现如图10-29所示，它使用了二进制编码。图8-25有限状态机模型的实现如图8-27所示，它使用了独热编码。比较这两种方法，二进制编码使用了较少的触发器，但下一状态逻辑相对复杂。独热编码使用的触发器多一些，但下一状态逻辑也简单一些。Xilinx 4000 FPGA由于具有较多的触发器，它倾向于使用独热编码。而Synopsys“设计分析程序”倾向于使用二进制编码。在使用综合器时用户可以根据自己定义的属性控制编码类型：

```
type STATE_TYPE is (S0,S1,S2);
attribute ENUM_ENCODING: STRING;
attribute ENUM_ENCODING of STATE_TYPE is "00 01 10";
```

上面的语句强制使用二进制编码。可将第三句改为：

```
attribute ENUM_ENCODING of  STATE_TYPE is "001 010 100";
```

就成了独热编码。

4) 综合RAM模型。当为RAM编码时，一般使用如下声明：

```
type MEMORY is array(0 to 1023) of
     STD_LOGIC_VECTOR(7 downto 0);
type MEM: MEMORY;
```

问题在于这个1024 × 8的RAM将会如何被综合。它可以被综合为一个二位触发器数组。对于ASIC设计，这种综合是可行的，而对于FPGA，如果使用了CLB的两个触发器，则需要512个CLB。这种解决方案将用到由综合工具预先定义的RAM，并把它作为结构模型的组成

部分加以实例化。例如，Xilinx提供了不同组织的16*1 RAM块。工具LogiBLOX可定义更大的RAM部件。这些预定义的块将4输入查找表转化为16*1 RAM部件，有效利用了CLB。

5) 综合锁存。下面是锁存电路的代码：

```
LATCH: process(GATE,DATA)
begin
  if(GATE = '1') then
    Q <= DATA;
  end if;
end process;
```

因为在GATE信号的值不等于‘1’的情况下Q的值是不确定的，所以引入了一个锁存。而ASIC库中已经包含了有效的锁存原语，因而就不存在此问题。对于XC4000XC器件，CLB的两个触发器可以构造一个锁存，这个问题也很好解决。

图10-30是一个门级的锁存电路。如果要用LUT实现它，开销非常大。因此，对于没有内部锁存的FPGA，要尽量避免使用锁存。很多情况下可以使用定时触发器来替代它。

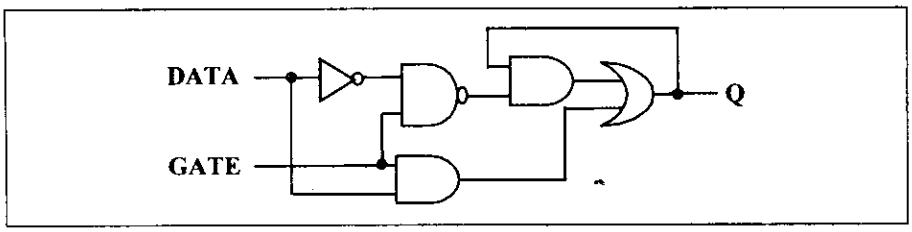


图10-30 门级的锁存电路

10.4 为组合电路综合建模

在为组合电路综合行为建模时，可以使用一个进程或等价于一个进程的VHDL构件。所有的电路输入在模型中都成为列入进程或等价进程感应表中的信号。编码时可以选择不同的抽象级别，级别的选择决定了综合工具将模型转换为门级逻辑的难易程度。

表10-1 ‘1’计数器的真值表

A_2	A_1	A_0	C_1	C_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

表10-1是第3章介绍的‘1’计数器的真值表。两位输出C的输出值是3位输入A中1的个数。图10-31是该计数器的实体描述及五个递增抽象级别的结构表：数据流、MUX、ROM、算法及子程序。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity ONES_CNT  is
  port (A: in STD_LOGIC_VECTOR(2 downto 0);
        C: out STD_LOGIC_VECTOR(1 downto 0));
end ONES_CNT;

architecture DATA_FLOW of ONES_CNT is
begin
  C(1) <= (A(1) and A(0)) or (A(2) and A(0))
          or (A(2) and A(1));
  C(0) <= (A(2) and not A(1) and not A(0))
          or (not A(2) and not A(1) and A(0))
          or (A(2) and A(1) and A(0))
          or (not A(2) and A(1) and not A(0));
end DATA_FLOW;

architecture MUX of ONES_CNT is
begin
  process(A)
  begin
    case A is
      when "000" => C<= "00";
      when "001"|"010"|"100" => C<= "01";
      when "011"|"101"|"110" => C<= "10";
      when "111" => C<= "11";
      when others => null;
    end case;
  end process;
end MUX;

architecture ROM of ONES_CNT is
begin
  process (A)
    type ROM_TABLE is array( 0 to 7)
      of STD_LOGIC_VECTOR(1 downto 0);
    constant ROM: ROM_TABLE :=
      (('0','0'),('0','1'),('0','1'),('1','0'),
       ('0','1'),('1','0'),('1','0'),('1','1'));
  begin
    C <= ROM(CONV_INTEGER(A));
  end process;
end ROM;
```

图10-31 '1'计数器的构架

构架DATA_FLOW、MUX、及ROM都可以在Synopsys里一步优化完成综合，产生的有效电路如图10-32。构架ALGORITHMIC的表示很有意思。图10-33顺序实现了时序电路。令人惊奇的是综合工具可以在描述中消除组合逻辑电路。图10-34是Synopsys工具形成的电路。它包括乘法器、加法器及关联的门。图10-35是一个低效电路，在第一步综合优化时使用了LSI 10K库。该电路占用17个单元，最大路径传输延时为4.98ns。与图10-32所示的高效电路相比，后者占用9个单元，最大路径传输延时为2.49ns。然而，高效电路可以由低效电路通过使用Synopsis工具里的最大结果设置经过两步优化得到。它说明一个非常抽象的模型可以被综合，但要比具有直接物理实现格式的模型花费更多的工作量。

```
architecture ALGORITHMIC of ONES_CNT is
begin
  P1:process(A)
    variable NUM: INTEGER range 0 to 3;
  begin
    NUM := 0;
    for I in 0 to 2 loop
      if A(I) = '1'  then
        NUM := NUM + 1;
      end if;
    end loop;

case NUM is
      when  0 => C <= "00";
      when  1 => C <= "01";
      when  2 => C <= "10";
      when  3 => C <= "11";
    end case;
  end process P1;
end ALGORITHMIC;

architecture PROC of ONES_CNT is
  procedure ONES_CNT_PROC(X : in STD_LOGIC_VECTOR;
       Z_SIZE:INTEGER; signal Z: out STD_LOGIC_VECTOR) is
    variable  RES:  STD_LOGIC_VECTOR(Z_SIZE-1 downto 0);
  begin
    RES := CONV_STD_LOGIC_VECTOR(0,Z_SIZE);
    for I in  X'RANGE loop
      if (X(I) = '1') then
        RES := unsigned(RES) + 1;
      end if;
    end loop;
Z <= RES;
  end ONES_CNT_PROC;
begin
  ONES_CNT_PROC(A,2,C);
end PROC;
```

图10-31 '1'计数器的构架（续）

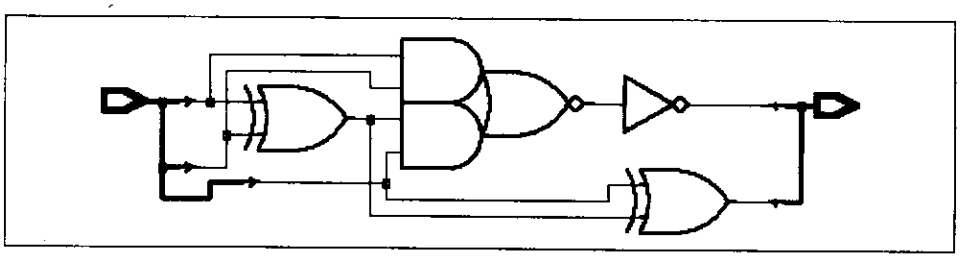


图10-32 '1'计数器的高效电路

构架PROC利用过程实现'1'计数器的另一抽象级的实现。它通过超载+操作完成STD_LOGIC_ARITH包中的无符号标准逻辑向量的加1功能。结果RES初始化为'0'，这里使用了包STD_LOGIC_ARITH中的类型转换函数CONV_STD_LOGIC_VECTOR，该函数将整数

0转换为类型STD_LOGIC_VECTOR。之所以使用转换函数是由过程所能够接受的输入向量的长度所决定的。由于该方法的抽象性，Synopsys工具从构架ALGORITHMIC可以得到同样的结果，即3种优化步骤得到图10-32所示的最简电路。然而，在组合逻辑电路里使用过程和函数建模具有更强的能力，这一点将在以后讨论。

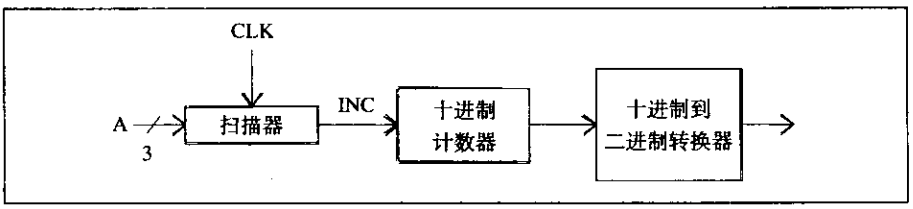


图10-33　隐含的时序电路

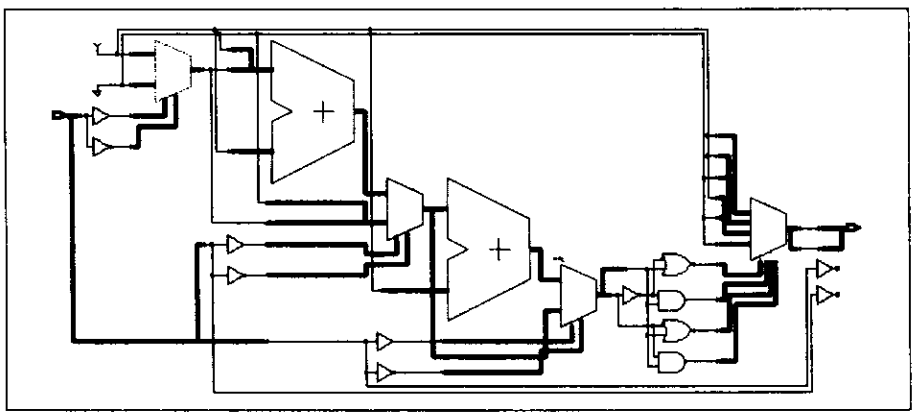


图10-34　算法构架的高级实现

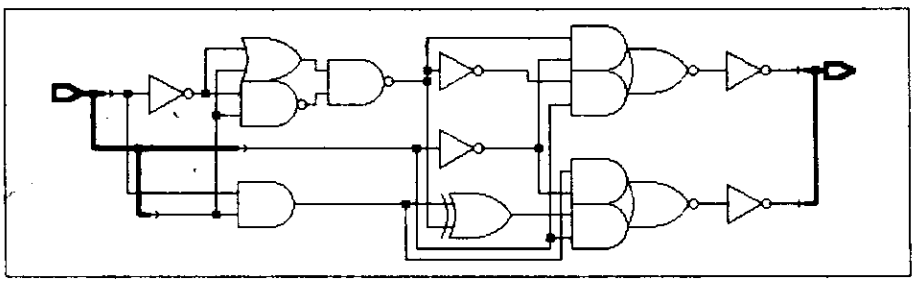


图10-35　'1'计数器的低效实现

10.4.1　运算电路的综合

本节主要讨论运算电路的综合。使用STD_LOGIC_1164包和一些支持运算电路包的IEEE库为这些电路建模非常简单。这里的讨论使用了Synopsys版本的包，特别是在包STD_LOGIC_ARITH里定义了两种类型：

```
type SIGNED is array (NATURAL range <>) of STD_LOGIC;
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
```

类型SIGNED在两种运算电路的实现中都被使用。类型UNSIGNED在无符号二进制数的运算电路中被使用。这些包包括了算术模型载入及与这两种新类型有关的操作，还包括无符号数或有符号数数据类型与整型及STD_LOGIC_VECTOR之间的相互转换操作。

图10-36是一个四位操作数和一位进位输入的加法器，它产生四位输出和及一个进位输出。输入和输出的数据类型都是STD_LOGIC或STD_LOGIC_VECTOR类型。输入A和B被转换为SIGNED类型；输入进位在前三位上补0，并转换为SIGNED类型。将这三个数（A、B及CIN）相加得到一个五位结果数。函数CONV_UNSIGNED作用于A，把它扩展为一个五位数（在最左端加一位0）。载入时可以自动调整其他加数的长度。结果的低四位（C_UNSIGNED）是输出和C。C_UNSIGNED最左端的一位是CAR_OUT。其中产生了三个中间变量，如果没有这些声明，则在进行类型变换时，转换函数无法确定操作数类型。

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
entity SIMP_ADD is
  port(A,B: in STD_LOGIC_VECTOR(3 downto 0);
       CIN: in STD_LOGIC;
       C: out STD_LOGIC_VECTOR(3 downto 0);
       CAR_OUT: out STD_LOGIC);
end SIMP_ADD;
architecture ALG of SIMP_ADD is
begin
  P1:process(A,B,CIN)
    variable  PADDED_CIN: STD_LOGIC_VECTOR(3 downto 0);
    variable A_UNSIGNED: UNSIGNED(3 downto 0);
    variable C_UNSIGNED: UNSIGNED(4 downto 0);
  begin
    A_UNSIGNED := UNSIGNED(A);
    PADDED_CIN :="000"&CIN;
    C_UNSIGNED := CONV_UNSIGNED(A_UNSIGNED,5) +
                  UNSIGNED(B) + UNSIGNED(PADDED_CIN);
    C  <= STD_LOGIC_VECTOR(C_UNSIGNED(3 downto 0));
    CAR_OUT  <= C_UNSIGNED(4);
  end process;
end ALG;
```

图10-36 加法器模型

乘法也可以被载入。从综合的观点来看，乘法操作使用了组合逻辑乘法器。在Synosis系统里，乘法器是设计库的组件。设计者可以选择进位保存数组或Wallace树，进位保存数组是缺省时的选择。同样，也可以指定是无符号数还是有符号数的乘法，这取决于操作数类型是SIGNED还是UNSIGNED。图10-37是一个乘10的乘法器。由于结果的长度是操作数长度的两倍——这是乘法操作普遍而又必须的条件。当使用常数综合乘法器时，并不需要实现整个乘法器。例如，对于乘10操作，可以对乘数先乘2再乘8，这两步乘法都可以通过乘数的移位来完成，最后将两个结果相加，大多数综合工具都是这样实现的。数的移位在实现时使用了布线。

10.4.2 层次算术电路：BCD到二进制的转换器

十进制到二进制的转换是数字电路的一种重要功能。幂等式的Horner规则为实现十进制

到二进制的转换提供了非常有效的方法。十进制数通常可以表示为：

```
library IEEE,DW02;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_arith.all;
use DW02.DW02_components.all;

entity MULT10 is
  port(DATA_IN: in STD_LOGIC_VECTOR(3 downto 0);
       PRODUCT: out STD_LOGIC_VECTOR(7 downto 0));
end MULT10;

architecture ALG of MULT10 is
begin
  process(DATA_IN)
    variable PROD_US: UNSIGNED(7 downto 0);
  begin
    PROD_US := UNSIGNED(DATA_IN)*CONV_UNSIGNED(10,4);
    PRODUCT <= STD_LOGIC_VECTOR(PROD_US);
  end process;
end ALG;
```

图10-37 乘10的乘法器模型

$N=D_3\times10^3+D_2\times10^2+D_1\times10^1+D_0$

其中D_i表示标准4位BCD码。Horner规则将它改写为：

$N=((D_3\times10+D_2)\times10+D_1)\times10+D_0$

因此，BCD码到二进制的转换可以通过反复的“乘10再相加”操作完成。图10-38是“乘后加”（$MSUM=D_i\times10+D_j$）逻辑电路的框图。图10-39是三个该功能块的互连图，它们计算上面N的等式。

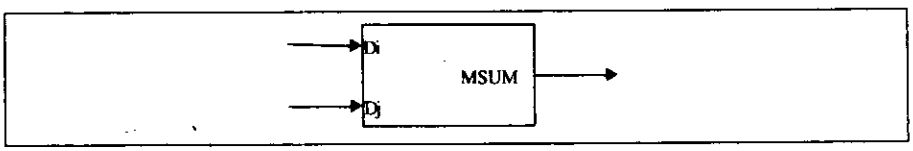


图10-38 乘10再相加的模块

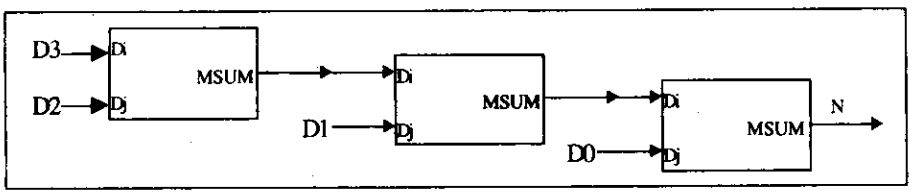


图10-39 三位十进制数到二进制数的转换器

图10-40是乘10后再相加电路的VHDL模型。为了计算乘法，需要变量PROD保存长度为操作数两倍的结果。变量的低位在后来的加法中使用。模型的输入大小有规定，这是因为D_i的值在每一步右移时都会变大。表10-2是D_i规定的大小。电路输出的大小比输入的四位要大得多。

```
library IEEE,DW02;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_arith.all;
use DW02.DW02_components.all;

entity MADD is
  generic(IN_WIDTH: NATURAL := 4);
  port(DI: in STD_LOGIC_VECTOR(IN_WIDTH-1 downto 0);
       DJ: in STD_LOGIC_VECTOR(3 downto 0);
       MSUM: out STD_LOGIC_VECTOR(IN_WIDTH+3 downto 0));
end MADD;

architecture ALG of MADD is
begin
  P1: process(DI,DJ)
    variable MSUM_US: UNSIGNED(IN_WIDTH+3 downto 0);
    variable PROD:UNSIGNED(2*IN_WIDTH-1 downto 0);
  begin
    PROD :=  UNSIGNED(DI)*CONV_UNSIGNED(10,IN_WIDTH);
    MSUM_US := PROD(IN_WIDTH+3 downto 0)+ UNSIGNED(DJ);
    MSUM <= STD_LOGIC_VECTOR(MSUM_US);
  end process;
end ALG;
```

图10-40 乘10相加电路的VHDL模型

表10-2 输入D_i所需的大小

级	最大的D_i	D_i大小
1	9	4
2	99	7
3	999	10
4	9999	14
5	99999	17
6	99999	20

图10-41是BCD到二进制转换器的结构模型，图10-42是仿真结果。

10.4.3 层次电路的综合

为了综合图10-41中的结构模型，必须首先分析MADD，然后是实体BCDCONV。在分析实体BCDCONV时，综合工具产生了MADD的三个实例。4、8和12分别赋值给三个实例中的IN_WIDTH。然后，对结构模型进行编译来产生综合电路。

图10-43是综合BCD到二进制转换器的层次化报告。每一级别上都标出了所使用的单元名称。最高级（级别3）是MADD电路。下一级（级别2）是Synosis实际器件库中的乘法器和加法器。由于乘法器中包括加法器，级别1包括了加法器和基本原语。级别0，也是最低一级，只包括基本原语，如AN2（两位输入AND），以及EO（互斥OR）。

当编译层次电路时，可以采用“取消组”和“归一化”操作。

1) 取消组操作。不考虑电路中的层次，只将电路看作一个级别。有时它也被称为层次展开。这种展开与逻辑级展开不同，以后将讨论后一种展开。图10-44是取消组（展开）电路的

层次报告。该级别上只包括原语。

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity BCDCONV is
port(D0,D1,D2,D3: in STD_LOGIC_VECTOR(3 downto 0);
     BIN_OUT: out STD_LOGIC_VECTOR(15 downto 0));
end BCDCONV;

architecture STRUCTURAL of BCDCONV is
  component MADD
    generic(IN_WIDTH: NATURAL := 4);
    port(DI: in STD_LOGIC_VECTOR(IN_WIDTH-1 downto 0);
         DJ: in STD_LOGIC_VECTOR(3 downto 0);
         MSUM: out STD_LOGIC_VECTOR(IN_WIDTH+3 downto 0));
  end component;
  signal MSUM2: STD_LOGIC_VECTOR(7 downto 0);
  signal MSUM1:  STD_LOGIC_VECTOR(11 downto 0);
begin
  C1: MADD
    generic map(4)
    port map(D3,D2,MSUM2);
  C2: MADD
    generic map(8)
    port map(MSUM2,D1,MSUM1);
  C3: MADD
    generic map(12)
    port map(MSUM1,D0,BIN_OUT);
end STRUCTURAL;
```

图10-41 BCD到二进制转换器的结构模型

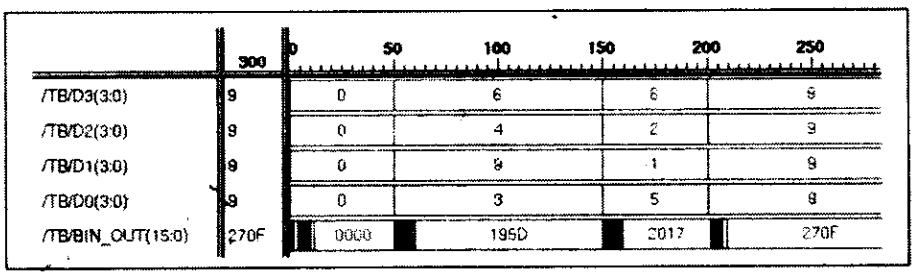


图10-42 BCD到二进制转换器的仿真响应

展开可以极大地减少面积和延时。对于BCD到二进制的转换，最初编译电路面积为342，最大延时为30.34ns。取消组（展开）之后的电路面积为309，最大延时为30.11ns。

2) 归一化操作。如果同一单元在层次中被多次使用，归一化将这些被使用的单元独立于其他单元进行编译。

图10-45是由4位加法器构建的加法器的框图。图10-46是该电路的VHDL模型。构件SIMP_ADD有三个实例。如果先取消组，然后归一化，则产生SIMP_ADD的三个副本，它们具有相同的名称。实体ADD_12_REG编译后的结果层次报告如图10-47。SIMP_ADD产生的三个实例的名字分别为SIMP_ADD_0、SIMP_ADD_1和SIMP_ADD_2。

在归一化之后，模型的编译有两种方法：自顶向下和自低而上。对于自顶向下编译，

综合工具对每个子设计进行编译，自动考虑各子设计的环境因素。子设计重新编译形成整体设计约束。图10-47的层次就是自顶向下编译的例子。图10-48对12位加法器自顶而下进行了编译。

```
****************************************
Report : hierarchy
Design : BCDCONV
Version: 1998.02
Date   : Thu Jun 18 09:17:19 1998
****************************************

BCDCONV
    MADD_IN_WIDTH4
        MADD_IN_WIDTH4_DW01_add_8_0
            AN2                                    lsi_10k
            EO                                     lsi_10k
            FA1A                                   lsi_10k
        MADD_IN_WIDTH4_DW02_mult_4_4_0
            AN2                                    lsi_10k
            EO                                     lsi_10k
            FA1A                                   lsi_10k
            MADD_IN_WIDTH4_DW01_add_6_0
                AN2                                lsi_10k
                EO                                 lsi_10k
    MADD_IN_WIDTH8
        MADD_IN_WIDTH8_DW01_add_12_0
            AN2                                    lsi_10k
            EO                                     lsi_10k
            FA1A                                   lsi_10k
        MADD_IN_WIDTH8_DW02_mult_8_4_0
            AN2                                    lsi_10k
            EO                                     lsi_10k
            FA1A                                   lsi_10k
            MADD_IN_WIDTH8_DW01_add_10_0
                AN2                                lsi_10k
                EO                                 lsi_10k
    MADD_IN_WIDTH12
        MADD_IN_WIDTH12_DW01_add_16_0
            AN2                                    lsi_10k
            EO                                     lsi_10k
            FA1A                                   lsi_10k
        MADD_IN_WIDTH12_DW02_mult_12_4_0
            AN2                                    lsi_10k
            EO                                     lsi_10k
            FA1A                                   lsi_10k
            MADD_IN_WIDTH12_DW01_add_14_0
                AN2                                lsi_10k
                EO                                 lsi_10k
```

图10-43 层次报告

对于自低向上优化，子设计首先独立编译，或采用取消组设计。下一步再编译整个设计。子设计的编译策略是首先被特征化，然后进行整个设计的编译。图10-49是自底向上编译的例子。

自底由上编译的另一种方法是“黄金实例”法。它没有使用归一化操作，而是将子设计的每个实例都特征化并进行编译。由于这种实例的环境适用于所有实例而被认为是“黄金实例”。下一步，不考虑实例的属性，编译整个设计。图10-50是对12位加法器的黄金实例编译，其中C2是黄金实例。

```
****************************************
Report : hierarchy
Design : BCDCONV
Version: 1998.02
Date   : Thu Jun 18 10:27:04 1998
****************************************

BCDCONV
    AN2             lsi_10k
    AN3             lsi_10k
    AO2             lsi_10k
    AO4             lsi_10k
    AO6             lsi_10k
    EN              lsi_10k
    EO              lsi_10k
    EO1             lsi_10k
    FA1A            lsi_10k
    IV              lsi_10k
    ND2             lsi_10k
    ND3             lsi_10k
    NR2             lsi_10k
```

图10-44 取消组（展开）电路的层次

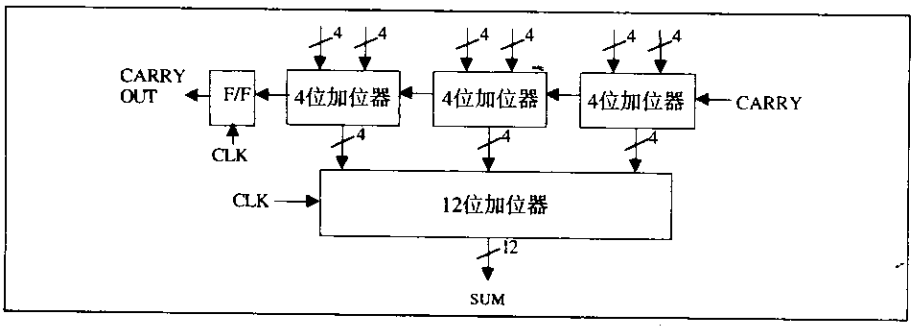


图10-45 12位加法器的框图

表10-3给出了这三种不同的编译策略的比较结果。自顶向下编译产生的电路开销最低。自底向上编译产生的电路速度最快。黄金实例法产生的电路并不是最优电路。换一个黄金实例也许会产生更好的结果。

表10-3 层次编译结果

	自顶向下	自底向上	黄金实例
面积	225	277	254
延时	8.84 ns	8.38 ns	11.19 ns

10.5 指定锁存及无关项

当用*if*或*case*语句为信号或是变量赋值时，语句中必须定义所有*if*或*case*条件下的赋值。否则，将指定一个锁存。考虑图10-51的代码。进程P_A_LATCHED里的信号A_LATCH在IN_EN='1'时被赋值为IN_DAT。代码并没有表示当IN_EN='0'时A_LATCH的操作。从仿真的

观点来看，由于信号具有存储空间，它可以保存旧值，因而并不存在问题。然而从综合的观点来看，锁存的指定是为了确保该值将被保留。图10-52是综合后的模型。信号A_LATCHED被锁住。另一方面，在进程P_A_COMB中，if语句是对称的，即*else*语句在IN_EN='0'情况下将'0'赋值给信号A_COMB。再看看图10-52，信号A_COMB是AND门的输出。

```
entity ADD_12_REG is
  port(X,Y: in STD_LOGIC_VECTOR(11 downto 0);
       CI, CLK: in STD_LOGIC;
       Z:out STD_LOGIC_VECTOR(11 downto 0);
       CO: out STD_LOGIC);
end ADD_12_REG;
architecture STRUCTURAL of ADD_12_REG is
  component SIMP_ADD
    port(A,B: in STD_LOGIC_VECTOR(3 downto 0);
         CIN: in STD_LOGIC;
         C:out STD_LOGIC_VECTOR(3 downto 0);
         CAR_OUT: out STD_LOGIC);
  end component;
  component REG_12
    port(DATA_IN: in STD_LOGIC_VECTOR(11 downto 0);
         CLK: in STD_LOGIC;
         DATA_OUT: out STD_LOGIC_VECTOR(11 downto 0));
  end component;
  component REG
    port(DATA_IN: in STD_LOGIC;
         CLK: in STD_LOGIC;
         DATA_OUT: out STD_LOGIC);
  end component;
  signal LO_CAR,MID_CAR,HI_CAR:STD_LOGIC;
  signal Z_COMB: STD_LOGIC_VECTOR(11 downto 0);
begin
  C1: SIMP_ADD
    port map(X(3 downto 0),Y(3 downto 0),CI,
             Z_COMB(3 downto 0), LO_CAR);
  C2: SIMP_ADD
    port map(X(7 downto 4),Y(7 downto 4),LO_CAR,
             Z_COMB(7 downto 4), MID_CAR);
  C3: SIMP_ADD
    port map(X(11 downto 8),Y(11 downto 8),LO_CAR,
             Z_COMB(11 downto 8), HI_CAR);
  C4: REG_12
     port map(Z_COMB,CLK,Z);
  C5: REG
     port map(HI_CAR,CLK,CO);
end STRUCTURAL;
```

图10-46 12位加法器模型

类似的情况出现在*case*语句里。考虑图10-51中的进程P_B_LATCHED，*when*语句必须对应控制变量的所有可能取值。*case*语句的测试信号SEL具有四种可能的整数取值："00"、"01"、"10"、"11"。然而，SEL是长度为2的STD_LOGIC_VECTOR类型。为了对应9值逻辑系统的其他情况，*when others*语句执行空操作。因此，从仿真语法的观点考虑，该语句是无效的。而当取值为"11"时的空操作也被编码。这表示在这种情况下什么也不做；即，期望信号B_LATCHED在SEL="11"时保留原先值。这里需要使用图10-52所示的锁存。

```
****************************************
Report : hierarchy
Design : ADD_12_REG
Version: 1998.02
Date   : Wed Jul  1 14:01:42 1998
****************************************

ADD_12_REG
    REG
        FD1                                    lsi_10k
    REG_12
        FD1                                    lsi_10k
    SIMP_ADD_0
        SIMP_ADD_0_DW01_add_5_0
            FA1A                               lsi_10k
    SIMP_ADD_1
        SIMP_ADD_1_DW01_addsub_5_1
            EO3P                               lsi_10k
            FA1A                               lsi_10k
    SIMP_ADD_2
        SIMP_ADD_2_DW01_add_5_1
            AN2P                               lsi_10k
            AO7                                lsi_10k
            EN                                 lsi_10k
            EO                                 lsi_10k
            EO3P                               lsi_10k
            IVP                                lsi_10k
            ND2                                lsi_10k
```

图10-47 12位加法器层次报告

```
read -format vhdl add_12_reg.vhd
read -format vhdl reg.vhd
read -format vhdl reg_12.vhd
read -format vhdl sim_add_wc2.vhd
current_design  ADD_12_REG
create_clock -period 10 -waveform {0 5.0} CLK
uniquify
compile
```

图10-48 自顶向下脚本

在不需要锁存的时候，有其他两种选择。一是在进程P_B_COMB_0中，值（'1'）可以赋值到"11"语句。第二，在进程P_B_COMB_1中可以在*case*语句前为B_COMB_1定义一个缺省值。这两种选择的仿真语义都是正确的。对于第二种情况，语句B_COMB<='1'将有一个省缺的赋值，它在*case*语句的同一个仿真周期里执行。从图10-52来看。信号B_COMB_0和B_COMB_1都没有锁存，它们都综合为同一组合电路的输出。

进程P_B_COMB_2中，在*when*子句"11"之下，信号B_COMB_2赋值为'-'，这是IEEE 9值逻辑系统的无关项。从仿真角度看，该值可以同其他枚举类型值一样处理；在SEL="11"时信号B_COMB_2取值为'-'。综合时如何处理该值取决于向量。对于Synopsys工具，无关项用于展开时简化逻辑。图10-52中的信号B_COMB_2是简化后的逻辑输出。

对组合逻辑电路使用'-'值的作用在图10-53中得到了很好的说明。它是两个组合逻辑ROM模型。除了第二个模型的两个地址空间里含有无关项，而在第一个模型里的值为'0'

以外，这两个模型几乎是完全一样的。图10-54是结果电路，第二个模型实现的开销较低。

```
current_design  ADD_12_REG
characterize {C1 C2 C3 C4 C5}
current_design  SIMP_ADD_0
write_script >  SIMP_ADD_0.scr
compile
current_design  SIMP_ADD_1
write_script >  SIMP_ADD_1.scr
compile
current_design SIMP_ADD_2
write_script >  SIMP_ADD_2.scr
compile
current_design  REG_12
write_script >  REG_12.scr
compile
current_design REG
write_script >  REG.scr
compile
current_design  ADD_12_REG
set_dont_touch {C1 C2 C3 C4 C5}
compile
```

图10-49 自底向上脚本

```
read -format vhdl add_12_reg.vhd
read -format vhdl reg.vhd
read -format vhdl reg_12.vhd
read -format vhdl sim_add_wc2.vhd
current_design SIMP_ADD
compile
current_design  ADD_12_REG
create_clock -period 10 -waveform {0 5.0} CLK
characterize {C2}
current_design  SIMP_ADD
compile
current_design  ADD_12_REG
set_dont_touch {C1 C2 C3}
compile
```

图10-50 “黄金实例”脚本

使用无关项的主要问题在于仿真与综合的语义不同。首先考虑输出。如果在仿真原模型时将‘-’赋值到输出，它就是结果所得到的值。然而，综合后的模型将产生'0'或'1'。因此，编译后两个模型的输出是不同的。测试输入中的‘-’信号会带来更加困难的问题。例如，语句：

```
if (A='-') then
```

该语句在Synopsys工具仿真时解释为FALSE，而不是预期的行为。因此，对于if语句要避免测试条件‘-’。

10.6 三态电路

为三态电路建模要考虑以下两种情况：1) 三态缓冲的输出必须要驱动相同的VHDL信号，

这里隐含着一个结果函数。2) 当一个三态电路的使能信号为‘0’时，缓冲的输出必须为‘Z’。如果满足这两个条件，大多数综合工具都可以指定三态缓冲。图10-55是满足这两个条件的两段进程。数据类型STD_LOGIC有一个IEEE 1164库中附加的结果函数。图10-56是综合后的电路。大多数综合工具允许使用“线与”或“线或”总线，如果没有，则通过IEEE 1164判决函数获得一条总线。

```
entity INFERRED is
  port(IN_DAT,IN_EN: in STD_LOGIC;
       SEL: in STD_LOGIC_VECTOR(1 downto 0);
       A_LATCHED, A_COMB, B_LATCHED, B_COMB_0, B_COMB_1,
         B_COMB_2: out STD_LOGIC);
--pragma dc_script_begin
--set_flatten true
--pragma dc_script_end
end INFERRED;

architecture ALG of INFERRED is
begin
  P_A_LATCHED: process(IN_DAT,IN_EN)
  begin
    if IN_EN = '1' then
      A_LATCHED <= IN_DAT;
    end if;
  end process;
  P_A_COMB: process(IN_DAT,IN_EN)
  begin
    if IN_EN = '1' then
      A_COMB <= IN_DAT;
    else
      A_COMB <= '0';
    end if;
  end process;
  P_B_LATCHED: process(IN_DAT,SEL)
  begin
    case (SEL) is
      when "00" => B_LATCHED <= IN_DAT;
      when "01" => B_LATCHED <= not IN_DAT;
      when "10" => B_LATCHED <= '0';
      when "11" =>  null;
      when others => null;
    end case;
  end process;
  P_B_COMB_0: process(IN_DAT,SEL)
  begin
    case (SEL) is
      when "00" => B_COMB_0 <= IN_DAT;
      when "01" => B_COMB_0 <= not IN_DAT;
      when "10" => B_COMB_0 <= '0';
      when "11" => B_COMB_0 <= '1';
      when others => null;
    end case;
  end process;
```

图10-51　用来解释指定锁存和无关项的模型

```
    P_B_COMB_1: process(IN_DAT,SEL)
    begin
      B_COMB_1 <= '1';
      case (SEL) is
        when "00" => B_COMB_1 <= IN_DAT;
        when "01" => B_COMB_1 <= not IN_DAT;
        when "10" => B_COMB_1 <= '0';
        when "11" => null;
        when others => null;
      end case;
    end process;
    P_B_COMB_2: process(IN_DAT,SEL)
    begin
      case (SEL) is
        when "00" => B_COMB_2 <= IN_DAT;
        when "01" => B_COMB_2 <= not IN_DAT;
        when "10" => B_COMB_2 <= '0';
        when "11" => B_COMB_2 <= '-';
        when others => null;
      end case;
    end process;
  end ALG;
```

图10-51 用来解释指定锁存和无关项的模型（续）

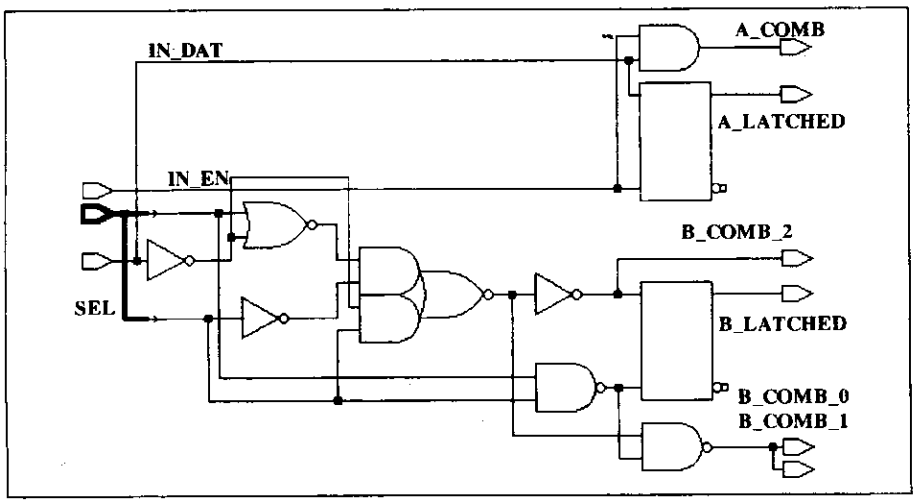


图10-52 综合后的模型

10.7 共享资源

在实现算术部件时使用综合工具来共享不同段落的共享资源非常重要。可以使用综合指导或不同编码风格来完成这一点。这里的第二个例子对它进行了解释。假如，给出如下电路的说明，数据输入为A、B、C、D，控制信号是SEL_CD。当控制信号SEL_CD等于'0'时，电路输出A+B；否则，输出C+D。该电路的设计中有如下两个基本方法：1) 先乘再加。2) 先加后乘。方法的使用对于电路尺寸和延时有显著的影响。编码风格决定综合时使用那一种方法。图10-57是一个乘后相加模型，图10-58是一个加后相乘模型。图10-59是乘后相加模型的高级

电路。图10-60是加后相乘模型的高级电路（高级电路指由Synopsys工具产生的独立于库的电路。这里弄清楚使用了哪个主要逻辑块很有用）。表10-4给出了两种电路的开销/延时的数据。可以看出，由于加法电路比乘法电路开销大，乘后相加电路相对于加后相乘电路要简单一些。然而，由于转换延时与加法延时同时产生，加后相乘电路相对于乘后相加电路的延时要低一些。因此，采用的是一种折衷的方案。

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity FUNCS is
  port(X: in STD_LOGIC_VECTOR(2 downto 0);
       Z1,Z2: out STD_LOGIC);
--pragma dc_script_begin
--set_flatten true
--pragma dc_script_end
end FUNCS;

architecture ROM of FUNCS is
  type ROM_1D is array(0 to 7) of STD_LOGIC;
begin
  FULLY_SPECIFIED: process(X)
    constant ROM1: ROM_1D:= "01101000";
  begin
    Z1 <=ROM1(CONV_INTEGER(X));
  end process;
  PARTIALLY_SPECIFIED: process(X)
    constant ROM2: ROM_1D:= "01101--0";
  begin
    Z2 <=ROM2(CONV_INTEGER(X));
  end process;
end ROM;
```

图10-53　双ROM模型

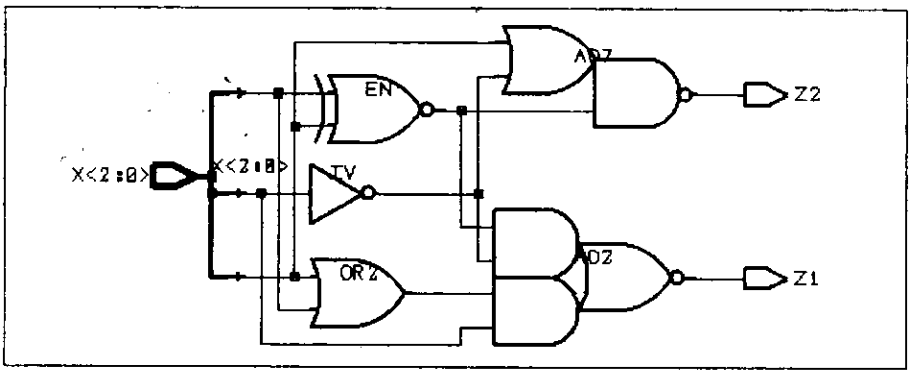


图10-54　双ROM模型的综合电路

表10-4　开销/延时的比较

电　路	面　积	延　时
乘后相加	51	8.47
加后相乘	73	7.09

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity TRISTATE is
  port(A,B,ENA,ENB: in STD_LOGIC; BUS_SIG: out STD_LOGIC);
end TRISTATE;

architecture   ALG of  TRISTATE is
begin
  PROCA: process(A,ENA)
  begin
    if (ENA = '1') then
      BUS_SIG <= A;
    else
      BUS_SIG <= 'Z';
    end if;
  end process;
  PROCB: process(B,ENB)
  begin
    if (ENB = '1') then
      BUS_SIG <= B;
    else
      BUS_SIG <= 'Z';
    end if;
  end process;
end ALG;
```

图10-55 三态电路模型

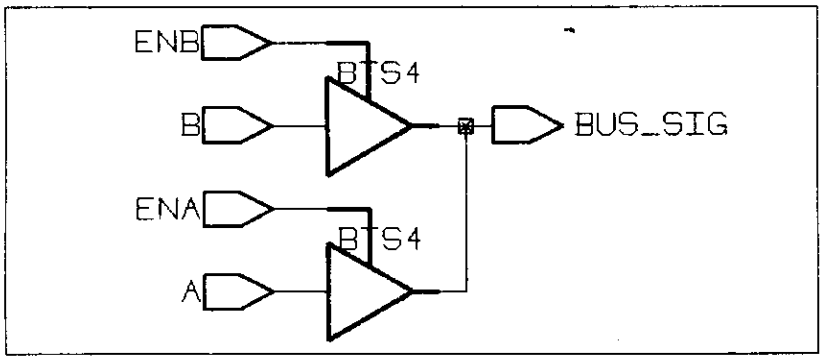


图10-56 综合后的三态电路

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
entity SHARE_1 is
  port(A,B,C,D: in STD_LOGIC_VECTOR(3 downto 0);
       SEL_CD: in STD_LOGIC;
       F: out STD_LOGIC_VECTOR(3 downto 0));
end SHARE_1;
architecture DF of SHARE_1 is
  signal MUX1,MUX2:SIGNED(3 downto 0);
  signal SUM:SIGNED(3 downto 0);
begin
  MUX1 <= SIGNED(A)when SEL_CD = '0' else SIGNED(C);
  MUX2 <= SIGNED(B) when SEL_CD = '0' else SIGNED(D);
  SUM <= MUX1 + MUX2;
  F <= STD_LOGIC_VECTOR(SUM);
end DF;
```

图10-57 乘后相加模型

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
entity SHARE_2 is
  port(A,B,C,D: in STD_LOGIC_VECTOR(3 downto 0);
       SEL_CD: in STD_LOGIC;
       F: out STD_LOGIC_VECTOR(3 downto 0));
end SHARE_2;
architecture DF of SHARE_2 is
  signal ADD1,ADD2:SIGNED(3 downto 0);
begin
  ADD1 <= SIGNED(A) + SIGNED(B);
  ADD2 <= SIGNED(C) + SIGNED(D);
  F <= STD_LOGIC_VECTOR(ADD1) when SEL_CD = '0' else
       STD_LOGIC_VECTOR(ADD2);
end DF;
```

图10-58 加后相乘模型

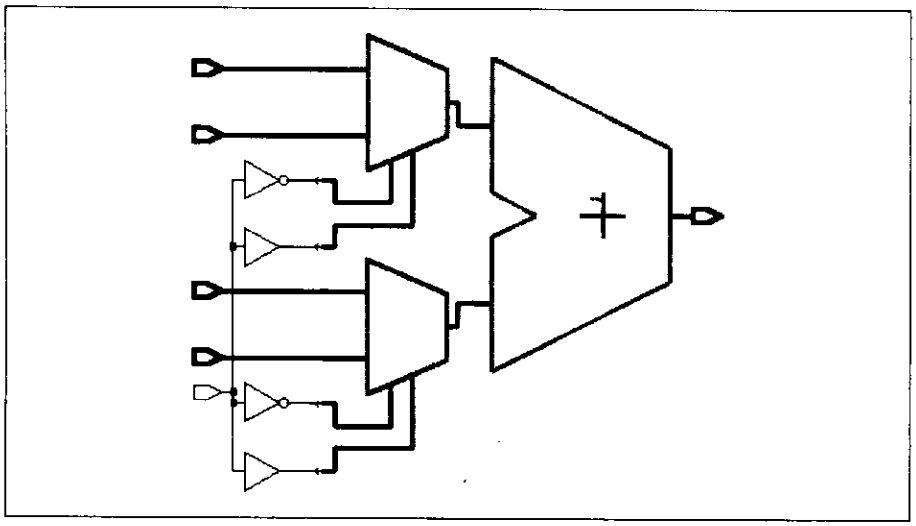


图10-59 乘后相加高级电路

10.8 展开与结构化

完成逻辑设计时，设计者注意到两级积之和或和之积电路是最快的，而因子（混合）格式可以降低开销设计，同样也降低速度。结构化产生了因子电路。通过使用综合命令，综合时可以控制电路类型。图10-61是一个三级电路模型。图10-62是综合后展开的电路。图10-63是隐含和之积实现的模型。图10-64是因子等效电路。

10.9 建模风格对电路复杂性的影响

下面考虑建模风格对电路复杂性的其他影响。

10.9.1 选择单独构件的影响

许多例子中，单独语言构件的选择对综合后电路的复杂程度有重要影响。例如，考虑

图10-65中4选1多路器的两种实现。图10-66和图10-67是使用LSI 10K库组件综合后相应的IF_STATEMENT多路器和CASE_STATEMENT多路器。表10-5总结了它们的开销/延时数据。对于这两种综合后的电路，CASE多路器在这两个方面都有优势。用Xilinx FPGA综合得到类似的结果。显然，使用*case*语句比使用*if-then*语句产生更低开销更高速度的实现电路。从这个例子中可以看出语言构件的选择对综合结果电路的开销和延时都有影响。

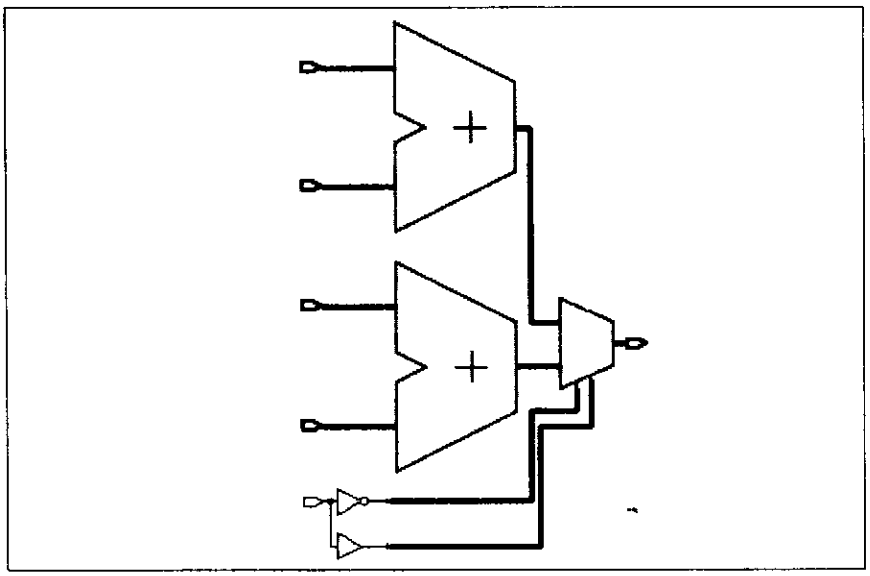


图10-60 加后相乘高级电路

```
entity LOG_FUNC is
port(A, B, C, D: in STD_LOGIC; F: out STD_LOGIC);
--pragma dc_script_begin
--set_flatten true
--pragma dc_script_end
end LOG_FUNC;

architecture DF of LOG_FUNC is
  signal S1,S2: STD_LOGIC;
begin
  S1 <= A or B;
  S2 <= S1 and C;
  F <= S2 and D;
end DF;
```

图10-61 三级电路

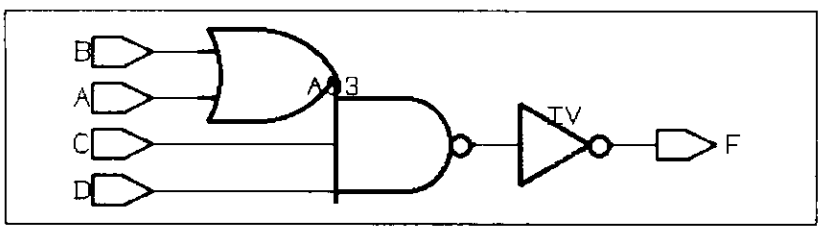


图10-62 展开后的电路

```
entity LOG_FUNC is
  port(A,B,C,D: in STD_LOGIC; F1,F2: out STD_LOGIC);
--pragma dc_script_begin
--set_structure -timing true
--pragma dc_script_end
end LOG_FUNC;

architecture DF of LOG_FUNC is
begin
  F1 <= (A and B) or (A and D);
  F2 <= (B and C) or (C and D);
end DF;
```

图10-63 二级模型

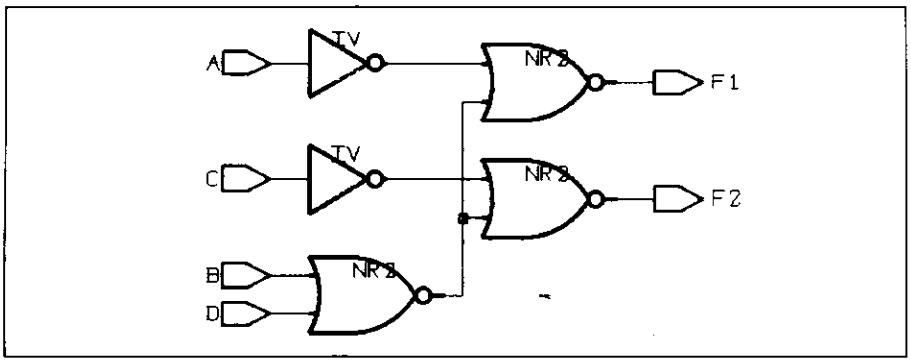


图10-64 因子电路

表10-5 开销/延时比较

电　路	面　积	延　时
if多路器	11	2.75 ns
case多路器	6	1.35 ns

10.9.2 通用建模方法的影响

许多例子中，通用建模方法的选择对综合后电路的复杂度有重要影响。例如，考虑设计一个具有乘法和除法两种功能的电路。该电路的建模方法可能有以下几种：1) 使用IEEE库中的左移（shl）和右移（shr）操作。2) 从IEEE库中选择重载乘法操作，该操作实例化一个混合乘法器块。3) 使用乘法和除法算法。4) 使用定时移位寄存器及串联操作。前三种方法都使用较大的混合逻辑块，第四种方法使用了移位寄存器和一些控制逻辑。一般来说，使用明显、简单的硬件实现具有较高的效率。这些方法的选择将作为习题供读者考虑。

习题

10.1 实现众多工具中的低通数字过滤器，如Cadence中的SPW或Elanix中的System View。使用工具仿真该过滤器。存储仿真的输入输出。使用工具产生一个VHDL模型或一个Xilinx模型，用工具所保存的输入对模型进行仿真，比较高层仿真的结果。

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity MUX is
  port (SEL: in STD_LOGIC_VECTOR(1 downto 0);
        A,B,C,D: in STD_LOGIC;
        MUX_OUT: out STD_LOGIC);
end MUX;
architecture IF_STATEMENT of MUX is
begin
  process(SEL,A,B,C,D)
  begin
    if (SEL = "00") then MUX_OUT <= A;
    elsif (SEL = "01") then MUX_OUT <= B;
    elsif (SEL = "10") then MUX_OUT <= C;
    elsif (SEL = "11") then MUX_OUT <= D;
    else  MUX_OUT <= '0';
    end if;
  end process;
end IF_STATEMENT;

architecture CASE_STATEMENT of MUX is
begin
  process(SEL,A,B,C,D)
  begin
    case SEL is
      when "00" => MUX_OUT <= A;
      when "01" => MUX_OUT <= B;
      when "10" => MUX_OUT <= C;
      when "11" => MUX_OUT <= D;
      when others => MUX_OUT <= '0';
    end case;
  end process;
end CASE_STATEMENT;
```

图10-65 多路器的*if*和*case*实现

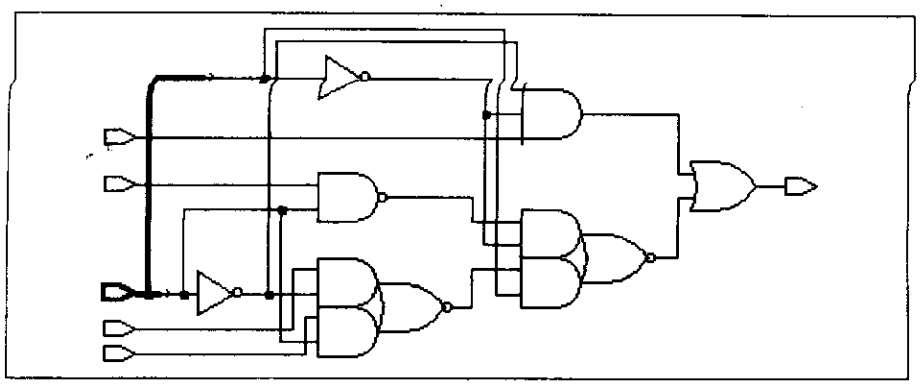


图10-66 综合的*if*语句多路器

10.2 选择可用双状态机建模的一个网络协议，每个状态机在通路的一端，使用状态表对系统建模，并仿真协议操作。产生一个VHDL模型。使用第8章讲述的方法，比较本模型与手工编码所产生模型的效率。

10.3 图3-9是第3章产生实体ONES _ CNT的测试程序，在这个测试程序中，一个进程被用来

以1ns的间隔对ONES _ CNT的输入赋值。解释该进程的工作流程。可以对它进行综合吗？为什么？

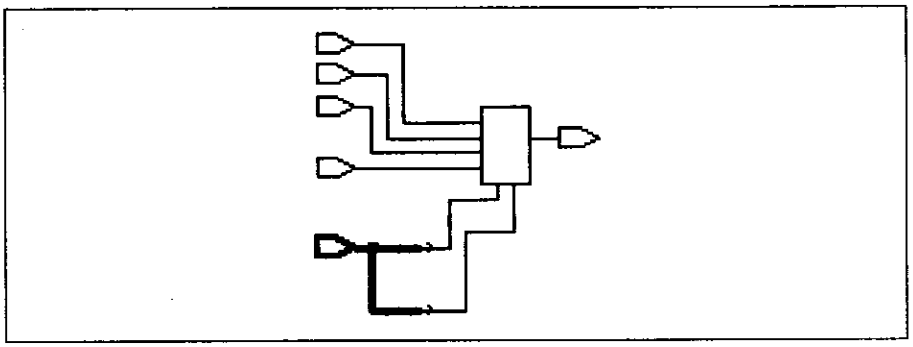


图10-67 综合的*case*语句多路器

10.4

```
 library IEEE;
use IEEE.std_logic_1164.all;
entity CKT is
  port(CLK,I: in std_logic; DOUT: out std_logic);
end CKT;

architecture ALG of CKT is
  signal X: std_logic;
begin
  process(CLK)
  begin
    if (CLK'EVENT and CLk = '0') then
        X  <=  I;
    end if;
        DOUT <= X;
  end process;
end ALG;

library IEEE;
use IEEE.std_logic_1164.all;
entity tb is
end tb;

architecture TEST of tb is
  component CKT
    port(CLK,I: in std_logic; DOUT: out std_logic);
  end component;
  for C1:CKT use entity work.CKT(ALG);
  signal CLK,I,DOUT: std_logic;
begin
  C1: CKT
    port map(CLK,I,DOUT);
  CLK <= '0' after 50 ns,  '1' after 100 ns,
         '0' after 150 ns, '1' after 200 ns,
         '0' after 250 ns, '1' after 300 ns,
         '0' after 350 ns, '1' after 400 ns,
         '0' after 450 ns;
  I <=  '0', '1' after 125 ns, '0' after 225 ns;
end TEST;
```

上面给出了模型和测试程序。假设该代码已经编译通过并且测试程序已经被仿真。画出仿真结果，并认真地画出I、X（部件CKT中的内部信号）、CLK和DOUT。用VHDL仿真程序检查你的结果。

10.5

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity CKT1 is
  port(I,CLK,RESET: in STD_LOGIC; DOUT: out STD_LOGIC);
end CKT1;
architecture  BEHAV1 of CKT1 is
  signal D: STD_LOGIC;
begin
  process(CLK,RESET)
  begin
    if RESET = '1' then
      D <= '0';
    elsif (CLK = '1' and CLK'EVENT) then
      D <= D xor I;
      Dout <= D;
    end if;
  end process;
end BEHAV1;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity CKT2 is
  port(I,CLK,RESET: in STD_LOGIC; DOUT: out STD_LOGIC);
end CKT2;
architecture  BEHAV2 of CKT2 is
  signal D: STD_LOGIC;
begin
  process(CLK,RESET)
  begin
    if RESET = '1' then
      D <= '0';
    elsif (CLK = '1' and CLK'EVENT)then
      D <= D xor I;
    end if;
  end process;
  Dout <= D;
end BEHAV2;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity CKT3 is
 port(I,CLK,RESET: in STD_LOGIC; DOUT: out STD_LOGIC);
end CKT3;

architecture  BEHAV3 of CKT3 is
begin
  process(CLK,RESET)
    variable D: STD_LOGIC;
  begin
    if RESET = '1' then
      D := '0';
    elsif (CLK = '1' and CLK'EVENT) then
      D := D xor I;
      Dout <= D;
```

```
    end if;
  end process;
end BEHAV3;
```

上面给出了三个行为模型，它们看起来具有相似的行为。

a) 仿真所有模型并比较结果。比较它们在初始行为及运行时行为上的不同。

b) 综合所有模型并进行仿真。使用类似图9-46的测试程序对模型进行测试，综合后模型的行为与行为域模型的行为是否相同？为什么？

c) 打印出所有模型的原理图。

d) 从这个实验中可以得出哪些基本结论？

10.6 综合的风格禁止使用信号赋值语句和wait语句的时间表达式。那么如何在模型中加入延时呢？

10.7 建模者希望使用整数变量为计数器建模。应使用整数类型还是约束整数？为什么？

10.8 对于下面的每种器件：1) 产生行为域模型；2) 仿真行为域模型；3) 综合行为域模型；4) 仿真综合后电路。所有器件均使用正沿触发时钟（CLOCK）。评论你所遇到的任何问题：

a) 具有低活跃的异步置位和复位JK触发器。

b) 具有高活跃的异步置位和复位D触发器。

c) 并行装载的左移、右移寄存器（8位宽）。

d) 8位宽，深度为6的FIFO。

10.9 假设你使用的是Synopsys综合风格，先预测下面代码蕴含的逻辑功能，然后综合该代码。把实际综合出来的电路与你的预期相比较：

```
entity SYNQ is
  port(A: in BIT; B: INTEGER range 0 to 3;
       C: BIT_VECTOR(0 to 3);
       D: out BIT);
end SYNQ;
architecture ALG of SYNQ is
  signal X: BIT;
begin
  X <= C(B);
  process(X,A)
  begin
    if  A ='1' then
      D <= X;
    end if;
  end process;
end ALG;
```

10.10 对下面的代码重复习题10.9。

```
library IEEE;
use IEEE.std_logic_1164.all;
entity  CKT is
  port(A,B,C: in STD_LOGIC; Q1,Q2: out STD_LOGIC);
end CKT;
architecture ALG of CKT is
begin
  process(A)
    variable  V :STD_LOGIC;
  begin
    if V = '1' then
      Q2 <= C;
```

```
    end if;
    if A'EVENT and A = '1' then
      Q1 <= B;
      V := not B;
    end if;
  end process;
end ALG;
```

10.11 对下面的代码重复习题10.9。

```
use work.PRIMS.all;
entity LOGIC is
  port(CON: in BIT_VECTOR(1 downto 0);
       A,B: in BIT_VECTOR(7 downto 0);
       CLK: in BIT;
       FOUT: out BIT_VECTOR(7 downto 0));
end LOGIC;

architecture ALG of LOGIC is
  signal F: BIT_VECTOR(7 downto 0);
begin
  FUNCTIONS:process(CON,A,B)
  begin
    case CON is
      when "00" => F <= A;
      when "01" => F <= not A;
      when "10" => F <= INC(A);
      when "11" => F <= A and B;
    end case;
  end process FUNCTIONS;
  STORE:process(CLK)
  begin
    if CLK = '1' then
      FOUT <= F;
    end if;
  end process STORE;
end ALG;
```

10.12 对下面的代码重复习题10.9。

```
use work.PRIMS.all;
entity TEST_QUESTION is
  port(CLK,W,X,Y,Z: in BIT; O: out BIT);
end TEST_QUESTION;

architecture BEHAV of TEST_QUESTION is
  signal  C: BIT_VECTOR(0 to 1);
begin
  BLK: block(not CLK'STABLE and CLK='1')
  begin
    C <= guarded INC(C);  --INC is an increment function
  end block BLK;
  O <= W when C="00" else
       X when C="01" else
       Y when C="10" else
       Z;
end BEHAV;
```

10.13 对下面的代码重复习题10.9。

```
use work.funcs.all;
entity SYS is
```

```
  port(C: in BIT; COM: BIT_VECTOR(0 to 1);
       INP: in BIT_VECTOR(0 to 7));
end SYS;
architecture CODE of SYS is
  signal X,Y: BIT_VECTOR(0 to 7);
begin
  process(C)
  begin
    if C='1' then
      case COM is
        when "00" => X <= INP;
        when "01" => Y <= INP;
        when "10" => X <= ADD8(X,Y);
        when  "11" => X <= ADD8(X,INC8(not(Y)));
      end case;
    end if;
  end process;
end CODE;
```

10.14　图10-68是一个模块的框图，表10-6是它的状态表。该模块是具有异步复位的时钟同步电路，状态表列出了每个当前状态的下一状态并定义了该状态的数据操作。

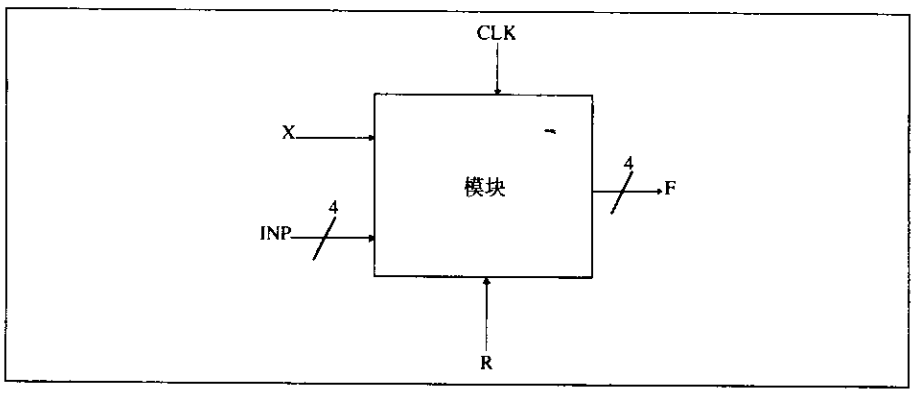


图10-68　习题10.14的框图

表10-6　习题10.14的器件模块的状态表

当前状态	输入X 0	输入X 1	当前状态的数据操作
	下一状态		
S0	S0	S1	F <= not INP
S1	S2	S3	F <= inc(INP)
S2	S2	S3	F <= dec(INP)
S3	S2	S3	F <= INP

a) 为模块写出一个独立构架。构架应具有Synopsys综合工具兼容的风格。假设加1函数和减1函数都在包FUNCPAC中。

b) 手工画出你预期的（a）综合后电路的逻辑图。把图分为两部分，即控制单元和数据单元。在控制单元区域使用门和触发器，在数据单元区域使用方框表示逻辑功能。

c) 使用Synopsys和Xilinx对模型进行综合，对控制单元尝试使用状态机类型，并与（b）

中手工画出的图进行比较。

10.15 Xilinx FPGA中的某些逻辑不能用查找表高效实现。为了改善这种情况，Xilinx需要具有额外的特殊逻辑。哪些类型的特殊逻辑可用来实现栈？

10.16 在把有限状态机综合为Xilinx FPGA如XC4010XL时，应该使用哪种状态编码？

10.17 为了综合第5章的控制计数器（图5-15和图5-16），该模型具有进程间的异步握手控制。它是否是可接受的代码风格？若不是，如何对其修改？讨论其他可能存在的问题。

10.18 考虑设计一个被2除和乘的电路，建模可能用到的方法有：1) 使用IEEE库中的左移和右移操作。2) 使用IEEE库中的重载乘法操作，把操作实例化为一个组合乘法器块。3) 使用乘法和除法算法。4) 使用时钟移位寄存器和相关操作。前三种方法都需要使用大的组合逻辑块，第4种方法使用一个移位寄存器和一些控制逻辑。通常使用清楚而简单的硬件实现的模型会更加有效。用Synopsys(ASIC)和Xilinx(FPGA)综合这几种模型，比较它们的复杂程度和速度。

10.19 用于XESS XS40板（图9-52）的结构化模型有三个组成部分：1) 被设计的模型。2) 处理FPGA输入和输出的DISPLAY模型。3) 把12MHZ板上时钟降为适合该模型的CLKGEN模型，研究这个XESS板并实现这些模型。

10.20 在XESS XS40板上使用Xilinx XC4010XL芯片，产生并综合一个读取8个开关，输出到8个LED的模型。

10.21 （欧几里德算法）下面是求取两个无符号二进制数的最大公约数算法：

```
Registers    X(6:0), Y(6:0)
Input:
  X <= Xi
  Y <= Yi
Calculation:
  while (X /= Y) loop
    if (X < Y) then
      Y <= Y-X
    else
      X <= X-Y
  end loop
Output:
  OUT <= X
```

注意 算法为伪代码格式，必须把它转换为正确的VHDL代码。

a) 把算法翻译为单进程的算法模型，要求遵守用于综合的VHDL风格的限制。通过对算法模型的仿真来验证它的正确性。

b) 使用Synopsys (ASIC)对（a）产生的算法模型进行综合，对结果结构仿真进行仿真验证。画出该电路等价的框图。

c) 在XESS XS40卡上使用Xilinx XC4010XL芯片综合该模型，在板上检测Xilinx模型。

d) 比较ASIC和FPGA实现的复杂度和速度。

10.22 设计一个简单计算器。

概述：设计一个简单的计算器并用Xilinx 4010 FPGA实现，首先为计算器产生一个行为模型并进行仿真。保证使用受综合限制的风格使其能被综合。然后，用Xilinx Foundation Express综合电路并把综合的电路烧制到XTEND板上去。

说明：计算器从Dip开关接收输入序列，最高位（DS(1)）指示每个数据字节的格式，

若DS (1)=1，则字节为一个操作，若DS(1)=0，则字节为一个操作数。若当前字节为一个操作，则最低两位（DS(7)-DS(8)）定义了执行对应于表10-7的操作，若当前字节为操作数，位DS(2)-DS(8)定义了操作数的值，（DS(2)为符号位），它采取7位补码格式。

表 10-7

DS(1)	DS(7)-DS(8)	操 作 码	注　释
0	---	---	DS(2)-DS(8)为操作数
1	00	+	加法
1	01	-	减法
1	10	*	乘法（乘以2的幂）
1	11	/	除法（除以2的幂）

实例：

表 10-8

DS(1)-DS(8)	解　释
1000 0010	乘法操作码
0000 1010	(+10)
0111 1101	(-3)

用户通过对dip开关的位赋值来输入一个操作数或操作码，输入后需按下SPACE按钮后释放。计算机把值读入到一个缓冲器并进行解码。若它是操作数，数据值进行符号扩展，压入栈中并在7段显示器上显示。若它为一个操作码，栈中前两项被移出，操作被执行，结果被压入栈中并在7段显示器上显示。表10-9为操作结果。

表 10-9

顶层的下一层	顶　层	操 作 码	操　作
A	B	+	A+B
A	B	-	A-B
A	B	*	A*B
A	B	/	A/B

计算器的算术表达式为逆向波兰符号表示（RPN），例如下面的表达式：

3*4+6/2

用RPN表示为：

3 4 * 6 2 / +

RPN的使用允许通过对表达式从左至右扫描并利用一个压入栈来计算表达式的值，若扫描到一个值，则它被压入栈顶。若扫描到一个操作码，栈顶两项被移出，执行特定操作，并将结果压入栈顶，表10-10给出了下面RPN表达式如何被计算：

8 7 2 / 2 * -5 +

表 10-10

输入项	DS(1)	DS(2)–DS(8) DS(7)–DS(8)	操作后显示	栈内下一操作
RESET	–	–	0	Empty
8	0	8	8	8
7	0	7	7	8, 7
2	0	2	2	8, 7, 2
/	1	3	3	8, 3
2	0	2	2	8, 3, 2
*	1	2	6	8, 6
–	1	1	2	2
5	0	5	5	2, 5
+	1	0	7	7

当没有超过最大栈长5或没有发生算术溢出时，计算器可以产生正确结果。

详细说明：

显示：三个7段显示器。

最大栈长：5。

输入操作数值的范围：7位补码。计算这个值的范围并在报告中写明。

结果范围：12位补码。计算这个值的范围并在报告中写明。

没有溢出发生。

输入表达式不会出错。

当前的dip开关输入键释放前对输入进行检测并把结果在LED上显示。

过程：为计算器创建一个VHDL算法级模型，并把它嵌入在包括计算器和不同显示实体的结构模型中，系统输入为：SPARE_PUSH_BUTTON，RESET_PUSH_BUTTON，DS(7:0)及XS40_CLK，系统输出为：LED(7:0)，TOP_7S(6:0)，LEFT_7S(6:0)和RIGHT_7S(6:0)，本书CD中的文件dedounce.vhd包括了debouncing开关的代码示例。使用一个测试程序来仿真结构模型。在组装最终结构模型之前可以独立仿真各部件，仿真下面的操作序列（混合符号表示），第一个表达式的RPN表示已经给出，还需要把后面两个表达示翻译成RPN表示，注意：{-5}表示已经用正确的补码输入一个负数。

1) (21+5*2)/2–16 [21 5 2 *+2/16–]

2) 10*(5–(2+(3*10)))

3) (60*32–41/16)*{–64}

在Xilinx 4010XL芯片上实现计算器。

10.23 设计并综合Booth乘法器的芯片。

说明：设计实现Booth算法乘法器的芯片，输入M1和M2为采用补码数值系统的8位有符号数，输出PRODUCT为表示M1和M2乘积的16位有符号数，它是一个组合逻辑器件。

设计过程：使用本书CD中的两个文件“booth.vcf”和“debounce.vhd”。“booth.ucf”为适合本题的用户约束文件，为芯片创建一个VHDL行为模型。使用IEEE标准逻辑包，包中有重载+和–，使用重载+、–来实现Booth乘法算法，用一个VHDL系统验证此模型。

为8位并行载入寄存器创建一个VHDL模型，模型框图如图10-69。DATA_IN为8位并行输

入，DATA_OUT为8位并行输出，LOAD为控制信号，CLOCK为系统时钟。当LOAD为高时，DATA_IN的值在CLOCK的上升沿载入寄存器。DATA总是显示寄存器的当前值。用一个VHDL系统验证此模型。

为乘法器到XSTEND板的接口创建一个结构化VHDL模型。设计接口使得用户可以通过dip开关和按键把一个操作数输入到Xilinx芯片中的内部寄存器，然后用户可以通过dip开关和第二个按键把第二个操作数输入到第二个内部寄存器，按键开关的弹起可能成为问题，项目文件“bounce.vhd”包含软件实现debouncing操作的VHDL代码实现，乘积的高8位由LED显示，低8位作为16进制字符在LEFT AND RIGHT 7段显示器上显示。使用Xilinx工具综合此模型。

使用IEEE包中的重载乘法操作，把Booth乘法器替换为另一种乘法器部件。在Xilinx上综合新的乘法器，比较这两种实现的设计效率（使用的芯片资源数量）和最大频率。

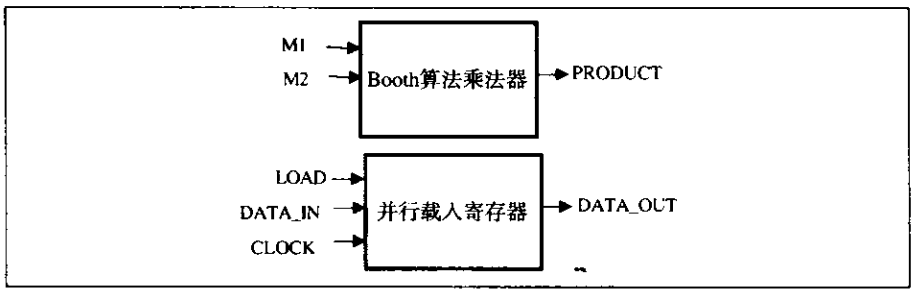


图10-69　Booth算法乘法器的框图

第11章 VHDL与自顶向下设计方法的结合

VHDL语言支持自顶向下设计方法。本章描述VHDL如何集成到这种方法的所有抽象级别。

11.1 自顶向下设计方法学

本章的主要目的是介绍一种自顶向下的设计方法，该过程中设计被逐步求精。一个设计包括从自然语言说明到系统行为描述，到系统分解，到RTL模型，到门级模型，最终到可以物理布线实现的电路。这是从高抽象级别到低抽象级别的整个设计周期。设计还包括物理实现的方法，并逐步地走到细节。

VHDL模型在所有级别上对硬件设计进行说明并建立文档，它们用于测试和仿真。系统/子系统最初的功能和要求在VHDL里体现为可以被VHDL仿真程序验证的可执行部分。最初的可执行体需要被精化，以排除定义时的模糊性。可执行体可以被插入到测试包里，以证明系统功能并为低级别的设计提供需求。综合工具可以为高级别的模型自动产生门级模型。

设计周期中，要定期进行精化和升级。要对模型进行修改，以改进系统功能，更正设计错误，提高操作速度，减少功耗、成本或重量，或者用新技术器件替换陈旧部分。设计的下一步基于当前设计，即使从头开始设计也不妨碍效率。如果前面的设计可以应用于新设计中，重新设计的周期可以显著缩短。因此，设计方法里很强调将前一个VHDL模型重用的方法。随着设计层次的降低，在低级别上使用高级别的测试包来测试模型也很重要，而且时钟和数据类型也变得重要起来。

在系统被分解为各个模块的集合之后，可以对设计的每个独立模块指派不同的设计小组。这些小组可以工作在不同地点，甚至可以分属不同的单位。因此，将不同的模块集成为最终的系统模型，并对其测试和评价的方法也比较重要。

本章中，从自然语言定义里产生一个可执行模型，不断自顶向下对模型进行分解，直到每个叶节点都是寄存器传输级模型，最后使用综合工具将RTL模型转换为门级模型。对于抽象层次，已经在第1章中详细介绍过了。设计时间、开销和效率直接受到所采用的设计方法的影响。

自顶向下是从抽象层次的顶部出发，逐步地到达底部。对于抽象层次及设计方法，目前还没有被普遍接受的标准。我们采取的方法是考虑使用多抽象级及重用VHDL模型。图11-1是一个典型设计过程的流程图。

第一步是为系统产生一个可执行VHDL模型，它精确地反映了系统的定义。完成这一步有多种方法。图11-1中三条并行的路径分别表示产生顶级模型的三种方法。最左边的一条路径是有经验的VHDL设计者直接从定义里为系统建模。而对于没有经验的设计者，常常采用中间或右边的路径，使用工具来辅助构造系统模型。

中间的路径使用语义捕获工具产生系统需求库。重要的系统参数作为类属出现在模型里。语义捕获工具可以自动产生VHDL代码外壳。使用者可以在VHDL外壳内插入行为模型来仿真整个模型。使用工具可以记录系统定义，以及跟踪系统设计过程中发生的系统定义的变化。

设计者必须熟悉VHDL语言来建立插入VHDL外壳中的行为模型。Synopsys SGE就是这样的一种工具，本章后续部分将介绍这种工具。

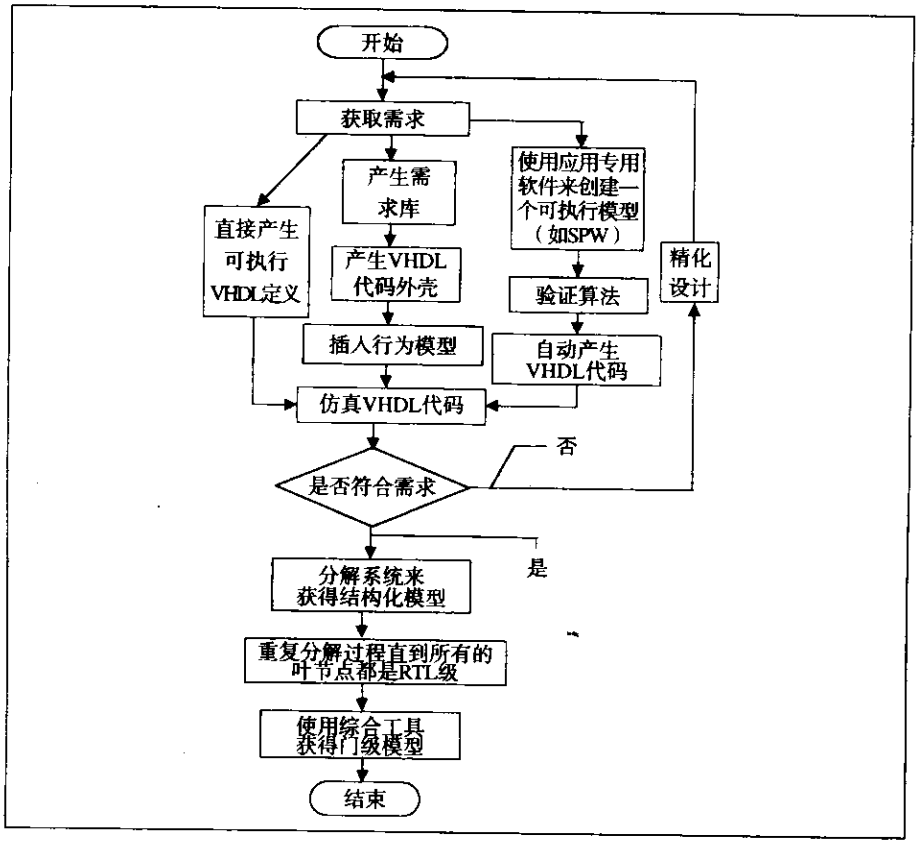


图11-1　自顶向下设计方法

为了进一步让设计者脱离VHDL语言的细节，可以使用应用说明工具产生VHDL的可执行说明。图11-1最右边路径描述了这种方法。例如，信号处理工作台（SPW）工具可以产生信号处理系统的可执行模型。这种工具提供了信号处理应用的功能原语，如快速傅立叶变换（FFT）及有限脉冲响应（FIR）过滤器。使用语义获取工具可以将算法建模为信号处理功能模块原语的互连。对SPW的仿真可以验证算法的正确性。SPW工具将自动产生SPW算法的VHDL模型的输出文件。该模型将被VHDL仿真程序仿真以验证这两个模型是否产生相同的输出。这种方法提供了对设计及重设计的审核。如果要改变设计，只需对SPW进行修改，验证新设计是否符合SPW中仿真的系统定义，然后要求SPW产生一个新的VHDL模型。它假定设计者并不具有VHDL编程的经验。

有时可执行说明的产生会暴露系统规格说明书的错误。最初的书面说明书可能并不完整或不一致。系统的书面规格说明书与可执行定义需要进行替代求精，如图11-1所示，直到可执行定义满足设计需求。

完成了可执行定义之后，系统被分解为几个相同抽象级或下一抽象级的组件。结果产生了系统的结构化模型。这些组件本身可以进一步分解为更小的组件。分解过程一直重复到所有的组件都处在RTL级或可以被综合。然后，使用一种综合工具可以得到门级模型。

现在可以看出，这种设计方法的步骤是从书面系统说明开始，使用该设计方法产生一系列VHDL模型，最终终止于门级设计。下一节介绍一个信号处理应用的背景知识，该应用将作为运行实例。

11.2 Sobel边缘检测算法

图像处理经常应用于在连续图像中跟踪移动物体。图11-2是一个图像处理系统，它从传感器接收图像的连续流，根据输入图像的数据选择跟踪物体。初始图像被不断加强，然后进行分割，以定位物体或找出感兴趣的区域。定位物体或区域之后，检查找出可以最终划分物体的特征。所有确认物体在后续图像中被跟踪，以确定它的速度和运动方向。

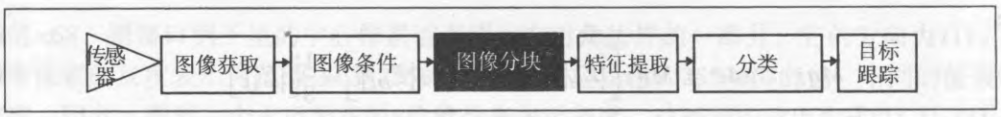


图11-2 图像处理系统

在一幅图像被分割之前，图像中的物体必须被检测并根据形状和边界特征进行粗略的划分。边界指的是图像中明显的局部变化，它也是图像分析的重要特征。边缘检测通常是从图像中恢复信息的第一步。在过去的20年里产生了许多边缘检测器，如Rorberts算子、Sobel算子、Prewitt算子、Laplacian算子等。由于Sobel是图像处理系统中最常用的边缘检测器，我们使用这个算子来解释我们的设计方法。

Sobel算法包括带4个3 × 3掩码的输入图像数据，即Sobel算子，它设置权重来检测水平、垂直、左对角、右对角各不同方向上密度幅度的不同。这个过程通常被称为过滤。

考虑为一个无序pixel[i, j]排序，如图11-3。每个窗口包括9个元素，当前的pixel[i, j]及相邻的8个元素，用a_0、a_1、a_2、a_3、a_4、a_5、a_6、a_7表示。

a_0	a_1	a_2
a_7	$[i,j]$	a_3
a_6	a_5	a_4

图11-3 图像窗口像素的一般排列

水平、垂直、左对角、右对角各图像方向上密度幅度的变化可以用如下式子计算：

$$E_H = (a_0+ca_1+a_2) - (a_4+ca_5+a_6) \tag{11-1}$$

$$E_V = (a_2+ca_3+a_4) - (a_0+ca_7+a_6) \tag{11-2}$$

$$E_{DL} = (a_1+ca_2+a_3) - (a_7+ca_6+a_5) \tag{11-3}$$

$$E_{DR} = (a_1+ca_0+a_7) - (a_3+ca_4+a_5) \tag{11-4}$$

其中$c = 2$、E_H、E_V、E_{DL}及E_{DR}，可以用如下的卷积特征码实现：

H =

1	2	1
0	0	0
-1	-2	-1

V =

-1	0	1
-2	0	2
-1	0	1

DL =

0	1	2
-1	0	1
-2	-1	0

DR =

2	1	0
1	0	-1
0	-1	-2

这四个参数，E_H、E_V、E_{DL}及E_{DR}一同计算梯度大小和方向。对梯度大小的一个普遍估计值为：

$$Magnitude = Max\left[|E_H|, |E_V|, |E_{DR}|, |E_{DL}|\right] + \frac{1}{8}\left\lfloor |E_\perp| \right\rfloor \tag{11-5}$$

其中$|E_H|$、$|E_V|$、$|E_{DL}|$、$|E_{DR}|$是E_H、E_V、E_{DL}及E_{DR}的绝对值，并且$E_\perp$表示不同方向上的密度幅度变量，它与具有最大密度的方向正交。符号$\lfloor\ \rfloor$表示“小于或等于的最大整数”。如果梯度超过一个特定的阈值，窗口正中的像素将被声明为边界的一部分。

边缘方向也称为边缘相位，可以用图11-4和表11-1所示的3位向量表示。边界方向定义为与最大边缘密度幅度关联的方向。例如，如果水平过滤器的输出具有最大的密度幅度，并且是一个正值，则方向为000。如果水平过滤器的输出具有最大密度幅度，但它是一个负值，则方向为100。

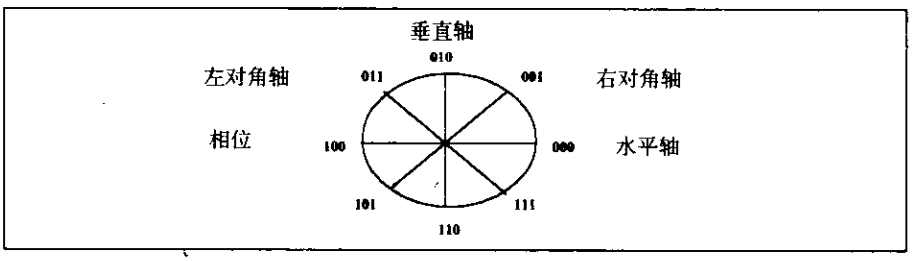


图11-4　量化后方向（相位）的表示

表11-1　边缘方向的代码

方向（相位）	代　码
正水平	000
负水平	100
正垂直	010
负垂直	110
正右对角	001
负右对角	101
正左对角	011
负左对角	111

为了说明幅度和相位的计算，考虑如下3×3的窗口：

12 01 08
05 09 03
40 03 10

使用这四个Sobel特征码计算该图像在不同方向上的密度如下：

$$E_H = (-34) \quad E_V = (-38) \quad E_{DR} = (+4) \quad E_{DL} = (-68)$$

所以，$Max[|-34|,|-38|,|4|,|-68|]=68$。最大值对应于左对角方向。由于该值为负，最大密度变量的方向是111。因此，梯度幅度为：

$$\text{幅度} = Max\left[|-34|,|-38|,|4|,|-68|\right] + \frac{1}{8}\left\lfloor|+4|\right\rfloor = 68$$

如果阈值<68，则窗口例子里的中心像素被声明为左对角边界的一部分，它的方向为111。

通常像素的灰度表示为范围在0到255的整数，0对应黑色，255对应白色，中间的值表示阴影的灰度。因此，需要一个永远为正的8位整数来表示灰度。过滤器的输出由式(11-1)~(11-5)计算，我们只需再加上两位表示部分和的幅度。减操作可能产生一个负值，所以还需要一位来表示最终结果的符号。因此，过滤器输出的表示共需要11位。为了使用标准总线，使用了12位2维数组来表示四个过滤器输出的向量。

边缘检测之后，边界像素被赋值为最大值（255），而非边缘像素都赋值为最小值（0）。在上一个例子里，窗口中心的像素声明为边缘的一部分，它的值被置为最大值（255）。上一个图像里，背景为黑色，边界为白色。

11.3 系统需求级

设计过程的第一步是从如英语等自然语言描述的定义里获取系统需求。这一步需要设计者进行确认。自然语言叙述的定义经常模棱两可且不够完整。获取系统需求的一个主要目的是为了消除这种模糊性。

11.3.1 书面规格说明

书面规格说明使用自然语言，它可能是描述系统的一个框图和/或一个时序图。书面规格说明描述输入输出信号和系统功能，还包括如系统是同步还是异步、信号端口是串行还是并行等基本的系统信息。它是设计者首先对系统的细节所做的介绍。然而，如前面所说，书面规格说明经常可能比较模糊。在设计过程里，设计者需要经常精化书写的规格说明以排除模糊性。下面是Sobel边缘检测系统的书面规格说明：

Sobel边缘检测系统使用Sobel操作检测图像边缘。图像的像素采取光栅扫描顺序（即从左到右、自上而下）扫描。边缘检测器串行输出含有边界信息的数据。该系统使用系统时钟以达到同步。系统必须在设计时预先考虑大小不同的图像。

下一节介绍如何正式获取书面定义。

11.3.2 需求库

这一节描述如何使用图表获取工具Synopsis Graphical Environment（SGE）来获取系统需求。它为系统需求生成一种可以自动产生VHDL模型外壳的文档格式。

SGE模型也称为需求库，这是因为它组织存储了所有的系统需求信息。系统接口在此将被正式定义。设计者将决定数据和控制信号所需的端口，这些端口与外部通信完成系统功能。而且，诸如时钟延时和数据类型等系统参数将在此被声明。设计步骤依赖于最终的可执行定义的抽象层次。在系统接口设计之后，设计者可以生成验证系统功能的行为结构。使用SGE工具产生需求库必须经过以下六个步骤。

步骤1。根据书面说明书，定义接口信号。

这一步需要由设计者产生。满足书面说明书的Sobel边缘检测器声明如下的端口信号。它并不是唯一的选择。

EDGE_START：类型为STD_LOGIC的输入信号，它的值为“1”时数据处理开始执行。该信号也初始化所有的寄存器和控制触发器到期望的开始状态。类型STD_LOGIC是IEEE 1164标准的九值逻辑类型。

CLOCK：类型为STD_LOGIC的输入信号，它作为系统时钟。由于系统定义为同步系统，所以需要这一信号。

INPUT：类型为PIXEL的输入信号。图像数据以光栅扫描顺序作为连续的像素值从该端口输入。每一行像素从左到右输入，行与行自顶而下输入。类型PIXEL是从0~255的整数类型。

THRESHOLD：类型为FILTER_OUT的输入信号，它定义了Sobel算法中决定边界的阈值。该参数之所以作为输入端口而不是类属，是因为期望阈值可以随外部系统条件的变化而变化。这个例子里，其他模块根据当前值使用某些算法来计算阈值。类型FILTER_OUT是从-2408到2407的整数类型。

OUTPUT：类型为PIXEL的双向信号，它决定当前像素是否为边界像素。在Sobel算法里为边界像素赋值为255（白色），为非边界像素赋值为0（黑色）。它的结果是黑色背景下的白色边界。输出图像的像素值在这一引脚表示为输入的像素值经过一段延时后的连续像素值。实际的延时值将由后面具有更多设计细节的设计过程所决定。

DIR：类型为DIRRECTION的双向信号，它定义了包括边界像素的边的方向。它由Sobel算法计算。数据类型DIRECTION是值的范围在000~111的STD_LOGIC_VECTOR数据类型，表示边的方向，它在前一节中已经定义。

步骤2。使用SGE图形获取工具，产生表示带有输入输出端口的设计实体的框图。

图11-5是Sobel边缘检测系统的框图。

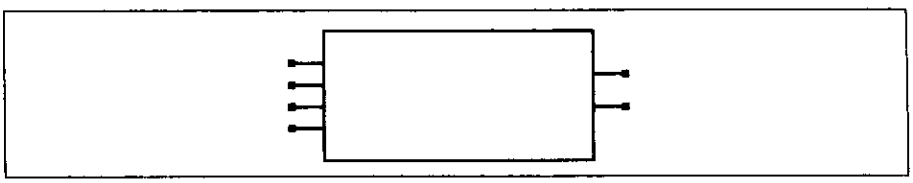


图11-5 Sobel框图

步骤3。使用SGE引脚属性工具，定义端口名称、端口数据类型、端口模式及仿真时每个端口的缺省赋值。

图11-6是为Sobel边缘检测系统里的信号CLOCK定义属性而打开的引脚属性窗口。信号名称（PinName）是CLOCK。信号模式（Polarity）为INPUT。信号CLOCK的数据类型

（VHDL_PinType）是STD_LOGIC，仿真时它的初始值（VHDL_DefValue）为“0”。图11-7是定义了引脚属性信息的SGE符号。

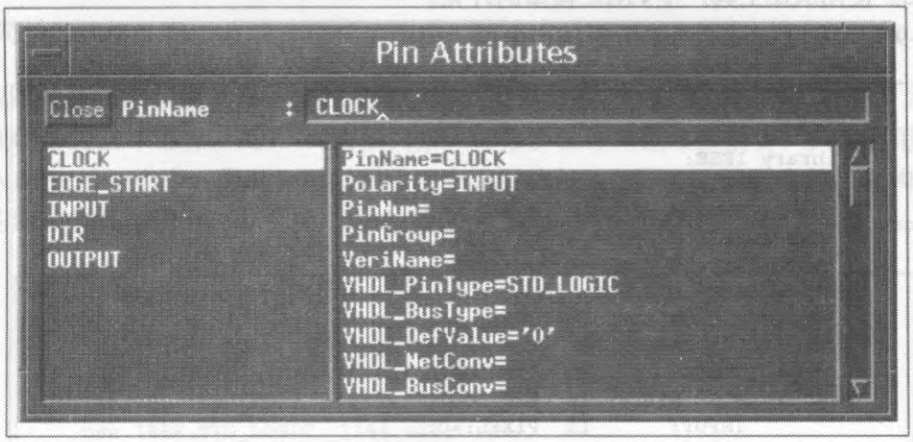


图11-6 Sobel边缘检测框图的引脚属性

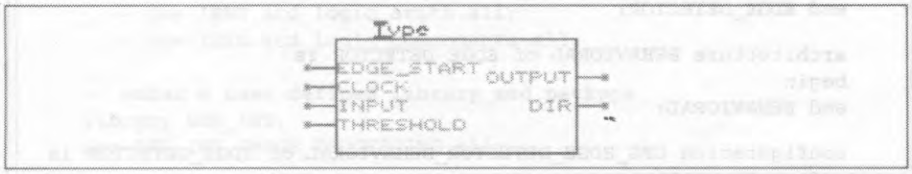


图11-7 Sobel边缘检测器的最终SGE符号

步骤4。使用SGE符号属性工具可以定义器件类属。图11-8是定义Sobel边缘检测系统的符号属性窗口。这里选择使用了IEEE_DEFAULT库。书面定义里的图像大小应该是一个变量，因此为图像大小定义为两个类属参数（NUM_ROWS和NUM_CLOS），它们表示对应的行数和列数。设计者可以在编译时使用过程模块为任意大小的图像定义类属值。它的数据类型是自然数。

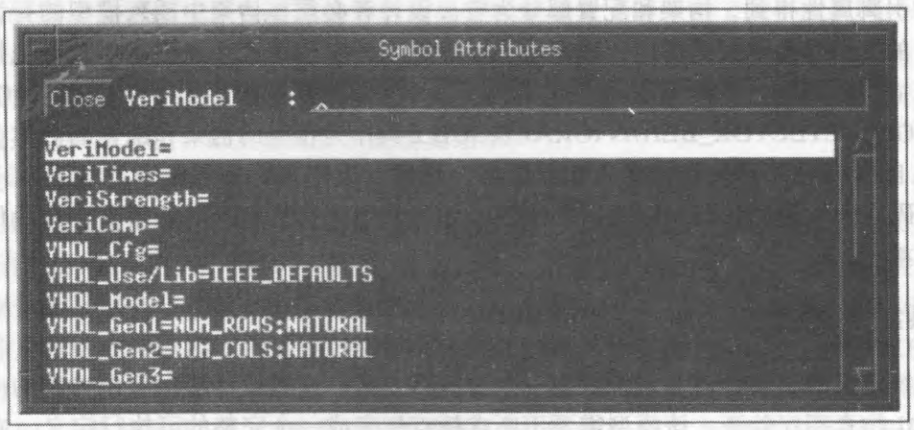


图11-8 Sobel边缘检测框图的符号属性

步骤5。使用SGE工具定义部件的名称。

Sobel边缘检测框图的名称定义为EDGE_DETECTOR。它在由SGE自动产生的VHDL代码里将会作为实体名称。

步骤6。使用SGE工具产生VHDL模型的外壳。

图11-9是SGE为Sobel边缘检测系统产生的代码。

```
--VHDL Model Created from SGE Symbol for the
--Sobel edge_detector.sym May 31 14:32:43 1997
library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.std_logic_misc.all;
  use IEEE.std_logic_arith.all;
  use IEEE.std_logic_components.all;
entity EDGE_DETECTOR  is
  generic( NUM_ROWS: NATURAL;
  NUM_COLS: NATURAL);
  Port (CLOCK:      In  STD_LOGIC := '0';
        EDGE_START: In  STD_LOGIC := '0';
        INPUT:      In  PIXEL:=0;
        THRESHOLD:  In  FILTER_OUT:=0;
        DIR:        InOut  DIRECTION :=0;
        OUTPUT:     InOut  PIXEL:=0 );
end EDGE_DETECTOR;

architecture BEHAVIORAL of EDGE_DETECTOR is
begin
end BEHAVIORAL;

configuration CFG_EDGE_DETECTOR_BEHAVIORAL of EDGE_DETECTOR is
  for BEHAVIORAL
  end for;
end CFG_EDGE_DETECTOR_BEHAVIORAL;
```

图11-9　由SGE产生的Sobel边缘检测器的VHDL代码

首先，IEEE库及四个包（std_logic_1164、std_logic_misc、std_logic_arith及std_logic_component）都是可见化的。然后声明设计实体接口，实体的名称为EDGE_DETECTOR。实体参数（NUM_ROWS和NUM_CLOS）声明为类属，端口声明由SGE框图中的引脚属性得到，构架和配置部分为空。设计者必须在构架中插入模型的行为或结构描述。空的配置作为缺省的绑定，或者由设计者插入特定的绑定。由于该框图不包括相应的原理图，它被用作一个叶节点部件并假设了一个行为模型。因此，BEHEVIORAL和CFG_EDGE_DETECTOR_BEHAVIORAL被SGE工具作为相应的构架和配置的名称。当然设计者可以自由编码并且改变构架名和配置名。然而，改变名称后无法自动连接SGE模型和VHDL代码，因此使得代码难以重用和维护。如果使用缺省绑定，则无需改变这段代码也可以自动连接。

11.4　系统定义级

在系统定义级，设计者为系统产生一个可执行定义。这是设计层次里最抽象的一级，可以由VHDL仿真程序验证。该模型需要不断求精直到完成一个完整的可执行定义。它也需要反复消除书面说明里的模糊性。通常它是直接实现系统需求的系统行为描述。熟练的VHDL编程人员经常直接从书面说明里产生该模型。但是，由工具SGE产生的VHDL代码可以通过对

其进行编辑来进化模型。此外，还可以使用特殊应用软件包，如SPW（参考图11-1的三条路径）。本节使用的是前一节介绍的SGE外壳。

11.4.1 可执行规格说明

生成可执行规格说明的第一步是编辑SGE（图11-9）产生的VHDL外壳。为了方便起见，我们把代码分为三个部分。图11-10所示为第一部分。设计完顶级模型后，只需利用SGE工具自动插入IEEE的一个库。因此，可以删除三个无用的包。此外，为设计中的独立构件建立一个独立的库和包则更加方便。图11-10中包括这个专用库的声明及使用该专用库的use语句。

```
-- Executable Specification of the Sobel edge detector
-------------------------------------------------------
--    SECTION 1:  LIBRARY AND PACKAGE DECLARATION      --
-------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC _1164.all;

---   deleted unused packages
  -- use IEEE.std_logic_misc.all;
  -- use IEEE.std_logic_arith.all;
  -- use IEEE.std_logic_components.all;

-- added a user defined library and package
library BEH_INT;
use BEH_INT.IMAGE_PROCESSING.all;
```

图11-10 编辑后SGE代码的第1节

图11-11是修改后SGE代码的第二部分，这一部分并没有被改动。图11-11也包括SGE修改后的第三段代码。以后将在第三段代码中插入行为描述。尽管可以对SGE产生的代码进行仿真和配置，但一般认为这样做会引来不必要的麻烦。所以我们直接选择构架进行仿真，并删除无用的配置声明。

1. 库结构

库用来组织和管理模型及模型的分析结果。在系统定义级使用了两个库。一个是由SGE自动插入的内部IEEE库。在这个库里，只有STD_LOGIC_1164包是可见的，这是因为这个时候不需要用到其他三个包。第二个库是用户定义的（BIN_INT）。这个库里存储了所有在系统定义级使用的模型和包，以及在此级别的仿真结果。

2. 包

常用的声明和函数并不需要被重复定义。包提供了收集这些声明和函数的方法，它可以通过use语句访问。如何使用该语句在第3章已经详细介绍过。图11-12是Sobel边缘检测系统的图像进程包的一部分，它包括类型和常数声明。两个常数BACKGROUND和FOREGROUND定义为像素值的允许范围。在例子中所使用的值可以用八位数据表示。子类型PIXEL是表示像素值的有界整数类型。

类型PIX3是表示图11-13所示的9像素窗口中任意行中三个像素的线性数组。类型PIXEL3_3是用来表示任意窗口九个像素的二维数组。图11-13表示PIXEL3_3数据类型定义与Sobel算法描述的像素窗口之间的关系。子类型FILTER_OUT用于表示四个Sobel过滤器的输出。

如果该模型被同步，这个类型将受到综合工具的限制而减少位数。子类型DIRECTION是一个三位的STD_LOGIC_VECTOR类型，它表示图11-4所示的八个方向。

```
-------------------------------------------------------
--   SECTION 2:  INTERFACE DECLARATION                --
-------------------------------------------------------
entity EDGE_DETECTOR is
  -- the generics were declared as symbol attributes
  -- in the SGE model
  generic( NUM_ROWS: NATURAL;    -- the number of rows
           NUM_COLS: NATURAL);  -- the number of columns
  -- the ports were declared as port attributes
  -- in the SGE model
  port (CLOCK: in  STD_LOGIC := '0'; -- the system CLOCK
        EDGE_START: in  STD_LOGIC := '0'; -- the START
                    -- signal for the image  processing
        INPUT: in  PIXEL:=0;        -- input image pixel
        THRESHOLD: in  FILTER_OUT:=0;  -- threshold to
                          -- determine edge pixels
        DIR: inout  DIRECTION :=0;    -- edge direction
        OUTPUT: inout  PIXEL:=0);  -- output image pixel
end EDGE_DETECTOR;
-------------------------------------------------------
--  SECTION 3:   ARCHITECTURE BODY DECLARATION        --
-------------------------------------------------------
architecture BEHAVIORAL of EDGE_DETECTOR is
    -- behavioral model declarations will be added here
begin
  SOBEL: process    -- Sobel process will be added here
end process SOBEL;
end BEHAVIORAL;
---- delete unused configuration part
-- configuration CFG_EDGE_DETECTOR_BEHAVIORAL of
-- EDGE_DETECTOR is
--   for BEHAVIORAL
--   end for;
-- end CFG_EDGE_DETECTOR_BEHAVIORAL;
```

图11-11 修改后SGE代码的第2、3节

图11-14是包IMAGE_PROCESSING的函数和过程声明。函数WEIGHT计算三个参数的权值总和，它的结果在式(11-1)到(11-4)中使用。WEIGHT的代码如下：

```
function WEIGHT ( X1,X2,X3: PIXEL)
  return FILTER_OUT is
begin
  return X1+ 2*X2 + X3;
end WEIGHT;
```

四个过滤器函数HORIZONTAL_FILTER、VERTICAL_FILTER、DIAGONAL_L_FILTER及DIAGONAL_R_FILTER相应地实现了式(11-1)~(11-4)。例如，HORIZONAL_FILTER函数的代码如下：

```
function HORIZONTAL_FILTER ( A: PIXEL3_3)
  return FILTER_OUT is
begin
 return  WEIGHT( A(1,1), A(1,2), A(1,3) )
```

```
        - WEIGHT( A(3,1), A(3,2), A(3,3) );
end HORIZONTAL_FILTER;
```

```
package IMAGE_PROCESSING is
------ declare two constants --------------------------
constant FOREGROUND:INTEGER:=255;
    -- define the value for the foreground pixel
constant BACKGROUND:INTEGER:=0;
    -- define the value for the background pixel

------ declare types and subtypes ---------------------
subtype PIXEL is INTEGER range BACKGROUND to FOREGROUND;
type PIX3 is array (1 to 3) of PIXEL;
type PIXEL3_3 is array(1 to 3, 1 to 3) of PIXEL;
subtype FILTER_OUT is INTEGER;
subtype DIRECTION is STD_LOGIC_VECTOR(2 downto 0);

---- declare 8 direction constants --------------------
constant EAST:DIRECTION:="000";
constant NORTHEAST:DIRECTION:="001";
constant NORTH:DIRECTION:="010";
constant NORTHWEST:DIRECTION:="011";
constant WEST:DIRECTION:="100";
constant SOUTHWEST:DIRECTION:="101";
constant SOUTH:DIRECTION:="110";
constant SOUTHEAST:DIRECTION:="111";
```

图11-12 IMAGE_PROCESSING包中的类型和常数

a_0	a_1	a_2
a_7	$[i,j]$	a_3
a_6	a_6	a_4

Sobel窗口

(1,1)	(1,2)	(1,3)
(2,1)	(2,2)	(2,3)
(3,1)	(3,2)	(3,3)

PIXEL3_3数据类型

图11-13 Sobel窗口和PIXEL3数据类型的关系

其他三个过滤器函数的代码类似。

过程COMPARE找到四个过滤器函数的输出（H，V，LD，RD）的最大绝对值。输出X是等于最大过滤器输出的幅度的绝对值正整数。输出DIR表示当前Sobel窗口最大密度变化的方向。通过检测四个过滤器的输出，选择最大幅度并查找图11-4定义的三位编码可以计算方向。输出Y表示与最大幅度过滤器相垂直的过滤器的幅度值。过程COMPARE的代码可以在CD ROM中对应于本文的地方找到。

函数MAGNITUDE根据式(11-5)计算Sobel幅度的斜率。输入A是最大的过滤器输出，输入B是垂直于最大过滤器的过滤器输出。这两个输入都是正整数。MAGNITUDE的代码如下：

```
function MAGNITUDE (A,B: FILTER_OUT)
  return FILTER_OUT is
begin
  return (A + (B/8));
end MAGNITUDE;
```

函数SHIFT_LEFT将一个三像素线性数组（A）向左移动一位。最右端填入输入B的值。

函数剩下的部分是不同类型的转换函数。所有函数的完整代码可以在CD ROM中对应于本节的部分找到。

```
------- declaration of functions and procedures -------
function WEIGHT (X1,X2,X3: PIXEL)
    return FILTER_OUT;
function SHIFT_LEFT (A: PIX3; B: PIXEL)
    return PIX3;
function HORIZONTAL_FILTER (A: PIXEL3_3)
    return FILTER_OUT;
function VERTICAL_FILTER (A: PIXEL3_3)
    return FILTER_OUT;
function DIAGONAL_L_FILTER (A: PIXEL3_3)
    return FILTER_OUT;
function DIAGONAL_R_FILTER (A: PIXEL3_3)
    return FILTER_OUT;
procedure COMPARE (H,V,LD,RD :in FILTER_OUT;
                   X,Y: out FILTER_OUT;
                   DIR:out DIRECTION);
function MAGNITUDE (A,B:  FILTER_OUT)
    return FILTER_OUT;
function INT_TO_STDLOGIC8 (A: INTEGER)
    return STD_LOGIC_VECTOR;
function STDLOGIC_TO_INT (S:STD_LOGIC_VECTOR)
    return INTEGER;
function INT_TO_STDLOGIC12 (A: INTEGER)
    return STD_LOGIC_VECTOR;
function STDLOGIC_TO_BIT (A: STD_LOGIC_VECTOR)
     return BIT_VECTOR;
end IMAGE_PROCESSING;
```

图11-14 IMAGE_PROCESSING包声明

3. Sobel边界监测系统的行为模型

这里不需要考虑创新和灵活性。有多种方法都可以实现Sobel边缘检测算法。下面说明其中的一种，但它并不一定是最优的实现方法。

像素以光栅扫描的顺序到来，由于必须检测任意3 × 3像素值数组，我们将所有行的像素复制到内部存储器中。模型构架的前几行如下：

```
architecture BEHAVIOR of EDGE_DETECTOR is
  type MEMORY_ARRAY is array(1 to 3, 1 to NUM_COLS)
       of PIXEL;
begin
```

类型MEMORY_ARRAY表示内部像素存储器。它是一个三行的数组。列数为图像（NUM_CLOS）每行像素的个数。Sobel边缘检测器将在内部存储器中存放图像像素的三个完整的行。存储器使用环状知更鸟（robin）的样式。即，三行像素的最顶一行首先存储在数组中。到来的第四行也将被存储在数组的第一行，第五行存放在第二行，等等。通过索引指针系统记录像素的图像行在内部存储器中的位置。例如，在图像像素所有行到达Sobel边缘检测系统的输入端之后，内部寄存器存储的内容如下：

Row 1 of internal memory array. First row of image pixels.
Row 2 of internal memory array. Second row of image pixels.
Row 3 of internal memory array. Third row of image pixels.

在第四行图像像素到达输入端后，内部存储器数组所含的像素有：

```
Row 1 of internal memory array. Fourth row of image pixels.
Row 2 of internal memory array. Second row of image pixels.
Row 3 of internal memory array. Third row of image pixels.
```

图像像素的第四行取代了第一行像素。类似地，图像像素的第五行取代内部存储器数组中的第二行图像像素。

可以随意决定使用哪个进程（SOBEL）来表示Sobel边缘检测系统。进程中使用如下变量，每个变量的使用目的在声明部分里说明。

```
SOBEL: process
  variable BUSY1,BUSY2: std_logic:='0';
    -- BUSY1='1' while input pixels are arriving.
    -- BUSY2='1' while edge pixels are being sent out
  variable A: PIXEL3_3:=((0,0,0),(0,0,0),(0,0,0));
    -- The current 3x3 window of pixels being processed.
  variable TEMP: PIXEL;
    -- Temporary storage for the output edge pixel
  variable E_H:  FILTER_OUT;
    -- Temporary storage for output of horizontal filter
  variable E_V:  FILTER_OUT;
    -- Temporary storage for output of  vertical filter
  variable E_DL:  FILTER_OUT;
    -- Temporary storage for output of left_diagonal filter
  variable E_DR:  FILTER_OUT;
    -- Temp storage for output of right_diagonal filter
  variable M:  FILTER_OUT;
    -- Temp storage for max absolute value of the filters
  variable P:  FILTER_OUT;
    -- Temp storage for filter output perpendicular to
    -- the direction of maximum filter value.
  variable MAG:  FILTER_OUT;  -- output edge pixel value
  variable PHASE: DIRECTION;  -- direction of the edge
  variable X1:NATURAL:=1;
    -- Row index for internal memory WRITE operation
  variable Y1:NATURAL:=1;
    -- Column index for internal memory WRITE operation
  variable X2:NATURAL:=1;
    -- Row index for internal memory READ operation
  variable Y2:NATURAL:=1;
    -- Column index for internal memory READ operation
  variable COUNT_R1,COUNT_R2: INTEGER:=0;
    --COUNT_R1 is the row number for input pixels.
    --COUNT_R2 is column number for input pixels
  variable MEMORY: MEMORY_ARRAY;
    -- Internal memory variable.

----- begin the SOBEL process -----
begin
```

变量A是当前的像素窗口。图11-15阐述该窗口在每个时钟周期被如何更新。窗口的每一行左移，新的像素加到该行的最右端。

在每一个时钟周期，都会从INPUT端口读入一个新的像素并存储到内部存储器数组中。当前窗口被更新并开始分析新的像素是否为边界的一部分。如果它是边界像素，则从OUTPUT端口输出的像素幅度被赋值为255，而且DIR端口赋值为边界的方向。如果当前像素并不是边界的一部分，输出端口OUTPUT的值被置为0，DIR端口没有定义。对于稳定状态，

边界输出像素在稳定流中发生，每个时钟周期一个。由于在寻找边界时必须有一个具有9个像素的窗口，因此只有在读取输入图像的第三行时才可以开始窗口进程。

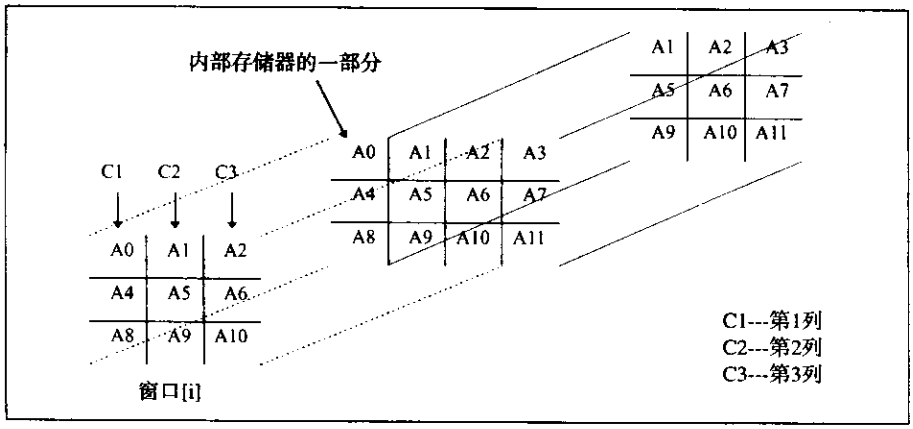


图11-15　更新当前窗口

两个内部变量BUSY1和BUSY2声明的初始值都为‘0’。端口EDGE_START的脉冲为边缘检测器提供了开始信号。它导致内部变量BUSY1被置为‘1’。BUSY1保持此位直到读入整个图像。内部信号BUSY2当开始移动数据列进入活跃窗口（如某一行的第一个像素到达INPUT端口）时被置为‘1’。计数器COUNT_R1记录行数，第一行为0行。控制BUSY1和BUSY2的代码如下：

```
wait until rising_edge(CLOCK);
  --  Set the internal busy variable ----------
if EDGE_START ='1' then
  BUSY1:='1';
end if;

-- Store the input image pixel in the internal memory array
if BUSY1 ='1' then
  MEMORY(X1,Y1):=INPUT;

  -- set internal busy variable BUSY2 and start to
  -- generate the address for the READ operation
  if COUNT_R1=2 and Y1=1 then
    BUSY2:='1';
  end if;
```

注意，变量X1和Y1记录存储当前像素值的内部存储器地址。变量X1记录行数，变量Y1记录列数。X1和Y1的初始值都为1，并且它们的范围都在1到NUM_COLS。因此当第三行（COUNT_R1=2）的第一个像素（Y1=1）到来时开始将数据载入活动窗口。

现在开始计算下一个像素将被写入的存储器地址：

```
  -- calculate the internal memory location for the next
  -- WRITE operation
  Y1:=Y1+1; -- update the column in the same row

  -- start a new row if we are at the end of the row
  if Y1=NUM_COLS+1 then
    Y1:=1;
```

```
    X1:=X1+1;
    COUNT_R1:=COUNT_R1+1;
  end if;

  -- reuse first row of internal memory -----
  if X1=4 then
    X1:=1;
  end if;

  -- Reset BUSY1 if all rows have been read in --
  if COUNT_R1=NUM_ROWS then
    BUSY1:='0';
  end if;
end if; -- End for if BUSY=1 clause
```

为了更新窗口变量，向左移动每一行并在右端插入一个新值，如图11-15。由于以知更鸟的样式重用了内部寄存器的行，所以必须在窗口更新算法中考虑到这一点。它由mod 3操作完成。同样，必须找到输入图像最后一行的结束位置，才可以重置BUSY2来结束窗口进程。

```
-- Update the 3 x 3 buffer window A --
if BUSY2='1' then

  ---- move columns 2 and 3 left one position -----
  for J in 1 to 2 loop
    for I in 1 to 3 loop
      A(I, J):= A(I, J+1);
    end loop;
  end loop;

  -- insert new values for column 3 -----
  for I in 1 to 3 loop
    X2:= (COUNT_R2 +I-1)mod 3+1;
    A(I, 3):=memory(X2,Y2);
  end loop;

  -- calculate the internal memory column from which the
  -- next memory READ operation will occur ----
  Y2:=Y2+1;
  if Y2=NUM_COLS+1 then
    COUNT_R2:=COUNT_R2+1;
    Y2:=1;
  end if;

  -- Reset the internal variable BUSY2 if we reach the
  -- last row of the input image.
  if COUNT_R2 = NUM_ROWS-2 then
    BUSY2 := '0';
  end if;
```

下面使用四个过滤器函数来决定当前像素是否为一个边界像素，相应的输出也被更新。

```
      -- apply the filtering function to the current window -
      E_H  := HORIZONTAL_FILTER(A);
      E_V  := VERTICAL_FILTER(A);
      E_DL := DIAGONAL_L_FILTER(A);
      E_DR := DIAGONAL_R_FILTER(A);

      -- determine the output edge pixels and directions --
      COMPARE(E_H,E_V,E_DL,E_DR,M,P,PHASE);
      MAG:= MAGNITUDE(M,P);
      if MAG >= THRESHOLD then
        TEMP := FOREGROUND;
```

```
      else
        TEMP := BACKGROUND;
      end if;

      -- update output signals ----
      OUTPUT <= TEMP;
      DIR <= PHASE;
    end if; -- This is the end of the BUSY1 section.

 end process SOBEL;  -- end the process
end BEHAVIORAL; -- end the architecture
```

完整的代码可以在CD-ROM中找到。

11.4.2 可执行规格说明的测试包的产生

VHDL模型使用一个叫做“测试包”的封闭模型进行测试。一个VHDL测试包可以定义为一个可执行VHDL模型，它实例化一个被测模型（MUT），同时提供驱动MUT的一组测试向量，并与期望响应进行比较。通过在有效的测试包的方法中结合自顶向下的设计方法，设计者可以提高模型产生效率和设计精确性的可信度。测试包是模型层次的最高一级。它是在模型测试时被仿真执行的实体。测试包为用户提供了通过仿真来进行MUT完整测试的能力。

一个典型的VHDL测试包由一个构架组成，该构架又包括了一个被测试部件的实例和一些进程，这些进程产生了与部件实例相连的信号值的序列。该构架也可以包括一些验证进程，它们比较输出值与部件实例的期望值。另外，可以使用仿真中的监测工具显示研究输出结果。

图11-16是一个基本的测试包的框图。被测试模型（MUT）使用了一组输入测试向量，它的输出响应在比较器里与GOLD模型的响应进行比较，比较器产生PASS/FAIL结果。GOLD模型可以是一个独立的有效模型，也可以是包括了期望输出序列的文件。如果GOLD模型是一个文件，则该文件由设计者制订，或者由外部源获得。

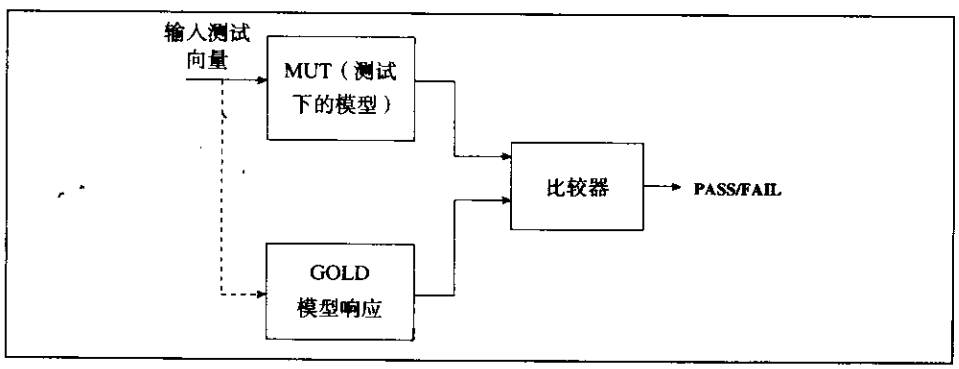


图11-16 基本测试包的框图

VHDL测试包至少可以达到以下几个目标：

1) 在不同测试条件下仿真MUT模型。

2) 自动验证MUT是否符合定义，不符合定义时记录所有的差异。它可以通过将MUT的输出与GOLD模型的输出进行比较，报告每个测试是成功还是失败来完成。在线验证可以精确检查MUT的行为，而且比手工验证花费的时间要少得多。相比之下，手工验证不仅单调冗长、

麻烦，而且低效。自动检测同样减少了将来维护时所需的时间开销，这是因为它可以对改变了的模型进行快速可信的重新验证。

3) 测试包能够完成回归测试。这种测试用于确保系统的设计变化不会反过来影响系统的性能。

在更高的抽象级，仿真的基本目标是为了验证系统定义的功能。在较低的抽象级，仿真被用来验证一系列的测试，可以发现普通的硬件失效，并比较低级别模型的输出和高级别模型的输出是否一致。较高抽象级产生的测试集可以被重用以测试低级别的模型。但是，在低级别模型里还需要其他的测试集来检测特定的硬件失效。测试通过使用一些有重点的测试实例来完成。由于测试的复杂性，需要在仿真之前确定验证目标，并且制定出测试计划以确保测试包符合预期的目标。

这里需要产生一组好的测试向量。尽管可以对测试模型采用简单的穷举测试，但它对于大型系统测试可能是一种不切实际的方法。通常在行为模型抽象级对测试模型采取的策略是选择测试模型的所有控制流路径。这些路径包括嵌套的控制组件、wait语句及嵌套的过程调用，它们都是测试向量的特定目标。

测试应该可以完整地验证系统功能。包括测试设计中的部件及将设计作为一个整体进行测试。对产生的每一个部件都进行测试非常重要，而且不能只假设它具有正确的操作。当部件组合为一个新的层次时，各组件之间的相互依赖将会导致意想不到的行为。每当系统中加入一个新的部件时都需进行一次仿真。最终，整个系统被集成并作为一个整体来测试。大部分的错误是在独立部件测试或少量部件配置测试时发现的。

1. 为可执行定义产生测试程序的方法

在上一节已经指出，行为集测试的主要目的是为了验证可执行定义的功能。这些已知的正确响应可以在以后用来测试较低抽象级别的MUT。

测试包模块（TBM）主要有以下功能：

1) 在仿真开始阶段，TBM从外部文本文件读取输入图像数据到内部帧缓冲区，然后以正确的格式发送给MUT。例如，外部文件的输入测试向量可能是整型，而在MUT端口可以是STD_LOGIC_VECTOR数据类型。因此，TBM必须在发送到MUT之前将输入数据转换为STD_LOGIC-VECTOR类型。

2) 一旦数据被MUT处理之后，它的输出响应就存储在TBM中的内部帧缓冲区。TBM的一个重要任务是决定输出响应写入帧缓冲区的时机。必须考虑内部MUT延时和测试包的人为操作。

3) 最后，帧缓冲区里的输出响应被写入外部文本文件，这些文件由比较器读出，比较器将MUT的输出响应与期望的响应进行比较，产生PASS/FAIL指示。

2. Sobel边缘检测可执行定义的测试实例

下面说明这一方法。图11-17是可以用来验证Sobel边缘检测系统行为实现的一个TBM的框图。它包括MUT，这是为整个系统提供时钟的时钟产生器，以及两个验证输出正确性的比较器。

用来测试MUT的激励值或测试向量由文件提供，即图11-17中的输入文件。测试包模块使用进程MEM1从图像文件中读取输入图像数据并存储到内部帧缓冲区。

TBM提供了四种输入信号到MUT。为清晰起见，输入信号（CLOCK、EDGE_START、

INPUT及THRETHOL）与相连的MUT输入端口名称相同。来自MUT的输出信号（OUTPUT和DIR）也与相连的MUT输出端口名称相同。这些端口的详细情况已经在本章的前面部分做了相应介绍。

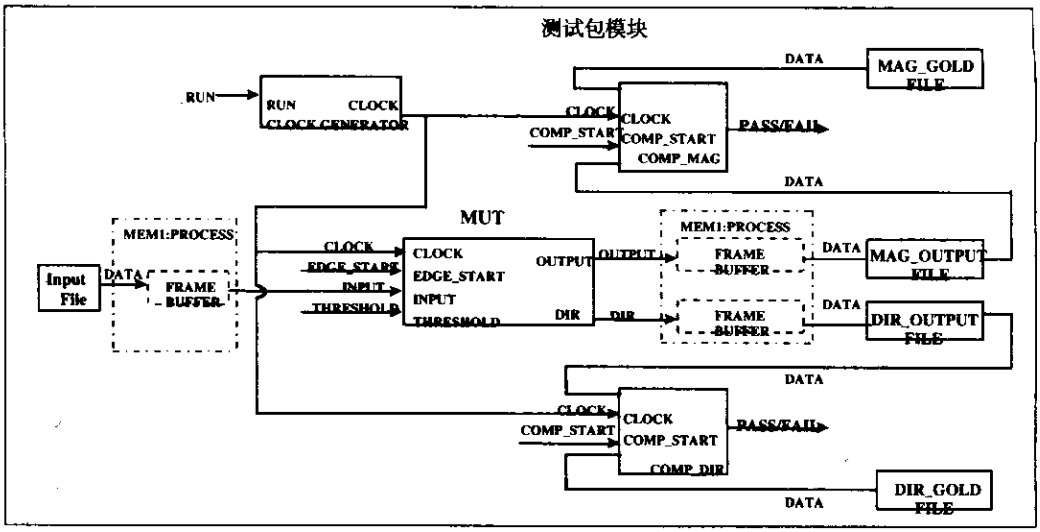


图11-17　Sobel边缘检测系统测试包模块的框图

输出像素的幅度和方向由进程MEM1存储在两个帧缓冲区中，然后再移到两个输出文件（MAG_OUTPUT和DIR_OUTPUT）中。两个比较器组件（COMP_MAG和COMP_DIR）分别把MUT产生的边界幅度、方向和存储在GOLD文件中的MAG_GOLD、DIR_GOLD进行比较。对于每一个测试，GOLD文件必须事先在书面说明书里准备好。比较器为边界幅度和方向提供PASS或FAIL回答。TBM的信号RUN和COMP_START分别用来分辨使能时钟发生器和两个比较器。

1) **时钟发生器组件**。时钟发生器组件在第4章已经描述，如图4-38。

2) **Sobel边缘检测器组件**。图11-18是Sobel边缘检测器组件的框图。它有四个输入信号：CLOCK，INPUT，EDGE_START和THRESHOLD。输出信号是OUTPUT和DIR，分别表示幅度和方向的输出。该模块是MUT。

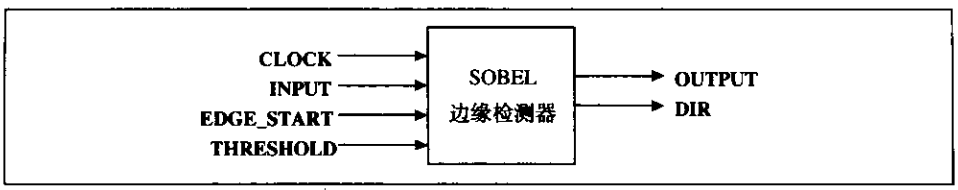


图11-18　Sobel边缘检测器的框图

3) **Sobel边缘检测系统的测试包模块**。测试包模块的源代码分为简单且可读性好的几个部分。图11-19的代码部分是库声明和实体声明。由于测试包的自包含性，实体声明里没有端口列表。类属常数用于在实体内部定义输入和输出文本文件。它由图11-19代码中的注释部分解释。因为这些是类属常数，所有的文件名称必须是11个字符，其中包括文件扩展名。例如，

“tesv_i4.daf ”是一个有效的文件名称。这里还使用了两个类属NUM_ROWS和NUM_CLOS来定义输入图像的大小。

```
library IEEE;
use ieee.STD_LOGIC_1164.all;
use STD.TEXTIO.all;
library BEH_INT;
use BEH_INT.IMAGE_PROCESSING.all;
---------interface declaration---------
entity TEST is

generic(IN_FILE:  STRING(1 to 11);   -- inut image file
  OUT_FILE:  STRING(1 to 11);--file for output magnitudes
  DIR_FILE:  STRING(1 to 11);--file for output directions
  NUM_ROWS:  NATURAL;-- number of rows in the input image
  NUM_COLS:  NATURAL;-- number of columns in the image
  WAIT_CYCLES:  NATURAL);   -- time required before
              -- Outputs  are written into frame buffers.
end TEST;
```

图11-19 Sobel测试包模块的实体声明

使用类属常数WAIT_CYCLES来定义延时，表示从一个像素到达INPUT端口到MUT在端口OUTPUT和DIR产生对应的边界信息之间的时钟周期。这个值之所以被定义为一个类属而不是常数，是因为时钟周期的个数与抽象级别不同。例如，当在行为级仿真模型时，WAIT_CYCLES必须是5个时钟周期；而在RTL级仿真模型时，WAIT_CYCLE需要有13个时钟周期。它是一个重要的类属。内部延时通常随抽象级别的不同而不同，通过提供一个在所有部件里都相似的类属，这些部件可以在各个抽象级被使用。这些类属在本章后面描述多级仿真时特别有用。

图11-20是测试包模块构架的声明部分。两个数组类型FRAME_IMAGE和FRAME_DIRECTION都已经被定义了。FRAME_IMAGE表示整数数据类型的一个帧缓冲区，FRAME_DIRECTION表示BIT_VECTOE数据类型的一个帧缓冲区。图11-20也包括时钟发生器、边缘检测器及比较器的部件声明。

时钟产生器、边缘检测器及比较器部件都被初始化，它们的端口信号匹配到端口匹配语句，如图11-21所示。构架的局部信号用于部件间的互连，如图11-17所示。一个附加信号START用于触发测试包模块。该构架包括进程MEMORY（图11-17的MEM1），它用来控制内部帧缓冲区和数据文件间的数据传送。

图11-22是MEMORY进程源代码的第一部分。在时钟的上升沿，如果信号START为‘1’，则初始化内部变量。这些留到它们被使用时再进行解释。然后为内部变量BUSY使用一条*case*语句，它定义了测试包代码的这部分将在每个时钟周期被执行。当START=‘1’时，BUSY初始化为1。如果条件满足（BUSY=1），输入图像文件到来的数据IMAGEIN读入到INPUT_IMAGE的帧缓冲区。帧缓冲区OUTPUT_MAG_IMAGE和OUTPUT_DIR_IMAGE的所有元素都初始化为“000”，它们准备存储来自MUT的幅度和方向输出。当输入图像文件都读入到帧缓冲区时，BUSY被置为2。一条断言语句产生了该图像文件成功读入帧缓冲区的确认报告。所有的这些帧缓冲区操作在START脉冲后的第一个时钟周期内完成。该时钟周期同信号START一样，并不存在于MUT的正常操作中。它们仅仅是测试包模块在准备系统仿真时的

人为动作。

```
--architecture description----
architecture BENCH of TEST is
  type FRAME_IMAGE is array(1 to NUM_ROWS,1 to NUM_COLS)
                          of INTEGER;
  type FRAME_DIRECTION is array(1 to NUM_ROWS,
              1 to NUM_COLS) of BIT_VECTOR(2 downto 0);

  ---component instantiations----
  component CLOCK_GENERATOR1
    port(RUN : in STD_LOGIC;
         CLOCK: out STD_LOGIC);
  end component;

  component EDGE_DETECTOR1
    port (CLOCK: in STD_LOGIC;
          EDGE_START: in STD_LOGIC;
          INPUT: in PIXEL;
          THRESHOLD: in FILTER_OUT;
          OUTPUT: inout PIXEL;
          DIR: inout DIRECTION);
  end component;

  component COMP_MAG1
    port(CLOCK: in STD_LOGIC;
         COMP_START: in STD_LOGIC:='0');
  end component;

  component COMP_DIR1
    port(CLOCK: in STD_LOGIC;
         COMP_START: in STD_LOGIC:='0');
  end component;
```

图11-20 Sobel边缘检测测试包模块的构架和组件声明

图11-23是测试包模块代码的后续部分。当BUSY=2时，图像数据在每个时钟周期通过信号INPUT从帧缓冲区INPUT_IMAGE中传递一个像素到边缘检测器部件。变量I和J的初始值为1，用来选择从缓冲区里传输的像素。信号EDGE_START在开始向MUT传输数据的一个时钟里的值为‘1’，它用来初始化MUT。内部信号COUNT的初始值为0，它用来为已经处理过的像素计数。

图11-24是过滤器窗口的初始位置，在输入图像的左上部分。第一个边界过滤器操作发生在这个窗口。由于输入像素以光栅扫描顺序到达（自顶而下一行一行，每一行的像素从左到右），最初的过滤器操作只能在第三行的第三个像素到达之后才可以开始。因而，在第一个过滤器操作之前接收的像素总数是（2*NUM_CLOS+3）。

图11-25是测试包模块代码的Sobel边缘检测部分。前一节中叙述的变量WAIT_CYCLES定义了在MUT输出端口获得的第一个输出边界像素的时机。前一节曾经指出，WAIT_CYCLES至少需要三个时钟周期来决定图像左上角的窗口的位置。此外，Sobel模块的内部延时也需要额外的时钟等待边界数据到达输出端口。这个延时在不同的级别有不同的值。在系统定义级，必须额外等待2个时钟周期，因此WAIT_CYCLES的值是5。若COUNT满足条件，边缘检测器输出的幅度和方向写入对应的输出帧缓冲OUTPUT_MAG_IMAGE和OUTPUT_DIR_IMAGE

中。变量X和Y记录正在处理的像素的行数和列数。X和Y都被初始化为2，这是因为最初窗口的中间像素在输入图像的第二行第二列（如图11-24）。若Y=1（左边界像素）或Y=NUM_COLS（右边界像素），则没有输出，这是因为图像边界的窗口不满。X和Y更新后准备接受下一个像素。当处理完最后一个有效像素时，BUSY被置为3。因为在图像边界的窗口不满，所以并不处理最后一行的像素。

```
-----Signal declarations----
  signal RUN:            STD_LOGIC:='0';
  signal CLOCK:          STD_LOGIC:='0';
  signal START:          STD_LOGIC:='0';
  signal INPUT, OUTPUT: PIXEL:=0;
  signal THRESHOLD:      FILTER_OUT:=0;
  signal DIR:            DIRECTION;
  signal EDGE_START:     STD_LOGIC:='0';
  signal COMP_START :    STD_LOGIC:='0';

-- Begin the test bench module.
begin
  START<='0' after 0 ns,'1' after 5 ns,'0' after 105 ns;
  THRESHOLD <= 110 after 0 ns;
  RUN<=transport '1' after 0 ns,'0' after 100000 ns;

---- Component Instantiations ------
P1 : CLOCK_GENERATOR1
  port map(RUN,CLOCK);
P2 : COMP_MAG1
  port map(CLOCK,COMP_START);
P3 : COMP_DIR1
  port map(CLOCK,COMP_START);
P4 : EDGE_DETECTOR1
port map(CLOCK,EDGE_START,INPUT,THRESHOLD,OUTPUT,DIR);

-----Processes------
MEMORY1 : process
  begin
  -- Details in Figures 11.22 through 11.26.
  end process;
end BENCH;
```

图11-21 Sobel边缘检测测试包模块的信号声明、组件初始化和进程声明

图11-26是将输出幅度和方向写入到文本文件IMAGEOUT和DIROUT的代码。断言语句用来指明输出是否成功写入了文本文件。输出写入之后，COMP_START被置为高以触发两个比较器。最后，BUSY重置为0，停止处理数据。数据处理进程一直挂起，直到产生一个新的START信号。

4) **比较器组件**。图11-27是比较器的框图，它通过把MUT的输出与期望的响应值进行比较，产生一个PASS/FAIL断言。该模型有两个输入。CLOCK和START。START信号用来触发比较器。

图11-28是用来比较幅度的比较器组件的部分代码。当内部变量BUSY被置为高时，从文本文件中读入GOLD模型（GOLD_MAGNITUDE）和MUT模型（TEST_MAGNITUDE）的数据，每次比较一个像素。如果存在错误，则置内部变量FLAG为'1'，如图11-29。如果FLAG

为高，则产生一个报告，指明MUT的输出不匹配GOLD的输出。如果FLAG='0'，则报告幅度值匹配。

```
MEMORY1: process
  variable VLINE1,VLINE2:LINE; -- for file I/O
  variable BUSY:INTEGER range 0 to 3;-- internal variable
     -- to trigger the different parts of the test bench

-- Internal frame buffers to store image data
  variable INPUT_IMAGE: FRAME_IMAGE; -- input image  buffer
  variable OUTPUT_MAG_IMAGE:FRAME_IMAGE;-- output magnitude
  variable OUTPUT_DIR_IMAGE:FRAME_DIRECTION;-- directions
  variable Z:INTEGER; -- used for reading the input files
  variable I,J:NATURAL:=1;--frame buffer indexing variables
  variable X,Y:NATURAL:=2; --frame buffer indexing vars
  variable COUNT:INTEGER:=0;-- used to specify when the
  file IMAGEIN: TEXT is in IN_FILE;  -- Input image file
  file IMAGEOUT:TEXT is out OUT_FILE; -- Output magnitudes
  file DIROUT:  TEXT is out DIR_FILE;  -- Output Directions

begin
  wait until rising_edge(CLOCK);
  if START = '1' then   -- Initialize variables
    BUSY :=1; COUNT:=0; I:=1;J:=1; X:=2; Y:=2;
  end if;
  case BUSY is
    when 1 =>
      --load the image to the internal frame buffer---
      for i in 1 to NUM_ROWS loop
        readline(IMAGEIN,VLINE1);
        for j in 1 to NUM_COLS loop
          read(VLINE1,Z);  INPUT_IMAGE(i,j):= Z;
          OUTPUT_MAG_IMAGE(i,j):=0;
          OUTPUT_DIR_IMAGE(i,j):="000";
        end loop;
      end loop;
      BUSY:=2; -- Advance to step 2
      assert (false) report "array read in";
```

图11-22　Sobel测试模块代码读入输入图像并初始化输出帧缓冲器

```
when 2 =>
     COUNT:=COUNT+1;  -- Update counter

      -- Send next pixel to MUT
      INPUT <= (INPUT_IMAGE(i,j));

      -- Update frame buffer index variables
      if (I=1) and (J=1) then
        EDGE_START<='1';
      else
        EDGE_START<='0';
      end if;
      if (I=NUM_ROWS) and (J=NUM_COLS) then
        I:=1; J:=1;
      elsif J=NUM_COLS then
        J:=1; I:=I+1;
      else
        J:=J+1;
      end if;
```

图11-23　Sobel测试包模块代码使用MUT处理输入

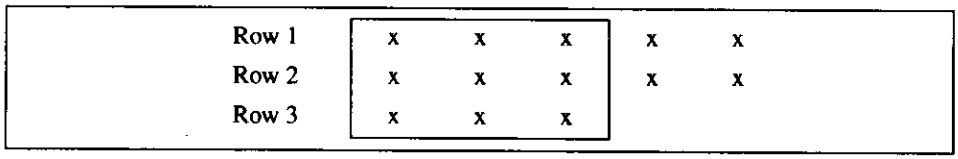


图11-24 过滤器第1个有效应用的窗口位置

```
-- This code is a continuation of the "when 2=>" from
-- Figure 11.23--

-- write the magnitude and direction outputs into internal
-- frame buffers --
     if COUNT>=(2*NUM_COLS+WAIT_CYCLES) then
       if not(Y=1) and not(Y=NUM_COLS) then
         OUTPUT_MAG_IMAGE(X,Y):= OUTPUT;
         OUTPUT_DIR_IMAGE(X,Y):= STDLOGIC_TO_BIT(DIR);
       end if;
       Y:=Y+1;
       if Y=NUM_COLS+1 then          --start new row--
         Y:=1;
         X:=X+1;
       end if;
       if (X=NUM_ROWS-1) and Y=NUM_COLS then
       --entire image has been processed--
       BUSY:=3;
       end if;
     end if;
```

图11-25 Sobel边缘测测试模块将有效边界信息传递到输出缓冲器的代码

```
--Write the outputs to external text files--
    when 3 =>
      RD_START<='0';
      for i in 1 to NUM_ROWS loop
        for j in 1 to NUM_COLS loop
          write(VLINE1,OUTPUT_MAG_IMAGE(i,j));
          write(VLINE1,' ');
          write(VLINE2,OUTPUT_DIR_IMAGE(i,j));
          write(VLINE2,' ');
        end loop;
        writeline(IMAGEOUT,VLINE1);
        writeline(DIROUT,VLINE2);
      end loop;

      -- Processing is complete----
      assert (false) report
       "magnitude and direction outputs written to file";
      COMP_START<='1'; -- Start the compare operation.
      BUSY:=0;         -- Reset the test bench module.
    when others => null;
    end case;
  end process MEMORY1;
end BENCH;
```

图11-26 将帧缓冲器内的边界信息复制到文本文件的代码

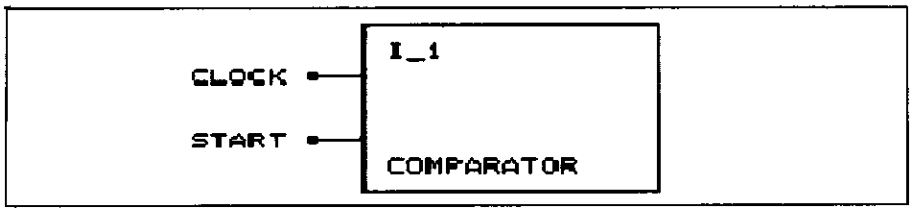


图11-27 比较器组件的框图

```
    wait until rising_edge(CLOCK);
  --Set the internal BUSY signal high to start comparison--
    if START ='1' and END_COMP_MAG ='0' then
      BUSY:='1'; FLAG1:='0';
    end if;

    if BUSY='1' then
  ---Read the files and compare the contents----
      for i in 1 to NUM_ROWS loop
        readline(TEST_MAGNITUDE,VLINE1);
        readline(GOLD_MAGNITUDE,VLINE2);
        for j in 1 to NUM_COLS loop
          read(VLINE1,A);
          read(VLINE2,B);
  -- If there is a mismatch in this row, set FLAG1.
          if A /= B then
            FLAG1:='1';
          end if;
        end loop;
      end loop;
    end if;
```

图11-28 比较器组件的部分代码

```
  if FLAG1='1' then
      -- assert message after comparing all the data
         assert (false) report
           "MAGNITUDE VALUES DO NOT MATCH -- FAIL";
       else
         assert (false) report
           "MAGNITUDE VALUES MATCH -- PASS";
       end if;
       END_COMP_MAG<='1';
     end if;
     BUSY:='0';
   end process;
  end COMPARE;
```

图11-29 比较器组件的代码

方向输出比较器的产生与此类似。

3. 可执行定义的配置声明

VHDL允许为模型配置一个具有多种适应能力的特定的模型。配置可以选择不同部件的构架，也可以选择在这些构架中而不是在部件实例中定义的类属常数。每次实验都有不同的配置文件。因此，以后只需简单地仿真配置文件就可以重复实验。使用配置的另一个目的是为

了定义端口映射，特别是为了说明类型转换函数。选择构架与选择类型转换函数相结合，是构造一个混合抽象级模型的关键，它使用了设计数据库中不同抽象级上建立的部件模型。有效利用配置声明可以为模型的配置管理提供帮助。但是，需要注意配置声明里绑定的结果是集中的，而不是把这些信息分布在不同的编辑单元。例如，如果同一个测试包可以与不同的外部数据文件一同使用，这些数据文件的名字就必须在配置声明里定义，而不是定义在读取该文件的设计实体的构架中。图11-30和11-31是一个配置文件的例子。

```
--declaration of an empty top-level component--
entity TB_CONFIG is
end TB_CONFIG;

architecture TEST_BENCH of TB_CONFIG is
  component TEST1
  end component;

begin
  con: TEST1;
end TEST_BENCH;
```

图11-30 测试包声明

```
---CONFIGURATION DECLARATION---------
library IEEE;
use ieee.STD_LOGIC_1164.all;

library BEH_INT;
use BEH_INT.IMAGE_PROCESSING.all;
use BEH_INT.all;
use STD.TEXTIO.all;

configuration CONFIG_B_INT of TB_CONFIG is
  for TEST_BENCH
    for con:TEST1 use entity BEH_INT.TEST(BENCH)
      generic map("tesv_i2.dat", "test_o1.dat",
                  "test_d1.dat", 10,10,5);

      for BENCH
        for P1:CLOCK_GENERATOR1 use entity
          BEH_INT.CLOCK_GENERATOR(BEHAVIOR)
          generic map(HI_TIME=>75 ns,LO_TIME=>25 ns);
        end for;

        for P2:COMP_MAG1 use entity
          BEH_INT.COMP_MAG(COMPARE)
          generic map("test_o1.dat","test_gm.dat",10,10);
        end for;

        for P3:COMP_DIR1 use entity BEH_INT.COMP_DIR(COMPARE)
          generic map("test_d1.dat","test_gd.dat",10,10);
        end for;

        for P4:EDGE_DETECTOR1 use entity
          BEH_INT.EDGE_DETECTOR(BEHAVIOR)
          generic map(NUM_ROWS=>10,NUM_COLS=>10);
        end for;
      end for;   -- End of BENCH
    end for;     -- End of con:TEST1
  end for;       -- End TEST_BENCH
end;             -- End of configuration
```

图11-31 测试包的配置声明

图11-30是TEST_BENCH的声明部分。它声明了一个实体TB_CONFIG，该实体包含一个构架，这个构架又包含一个部件声明TEST1及TEST1的一个实例。

图11-31显示在下表中定义了绑定的配置声明：

组　件	实体（构架）
con: TEST1	BEH_INI.TEST (BENCH)
P1	BEH_INI.CLOCK_GENERATOR (BEHAVIOR)
P2	BEH_INI.COMP_MAG (COMPARE)
P3	BEH_INI.COMP_DIR (COMPARE)
P4	BEH_INI.EDGE_DETECTOR (BEHAVIOR)

当需要在较低抽象级别测试边缘检测模型时，只需改变配置的名称和配置中的库，不需要再改变模型中的其他部分。类属常数也是在配置文件里被赋值。顶级实体TEST1定义了不同的输入输出文件的名称。NUM_ROWS和NUM_CILS定义为10，WAIT_CYCLES定义为5，这是因为模型在行为级进行仿真。对于时钟产生器，HI_TIME和LO_TIME分别定义为75ns和25ns。

4. Sobel边缘检测系统测试计划的实现

边缘检测模型的功能验证可以通过表11-2所示的测试计划来提高效率。测试计划列出了不同的测试目标、激励源、好的数据源及可接受的输出。

表11-2　测试计划

测试序号	激　励　源	黄金数据源	可接受输出	期望的语句覆盖
1	tesv_i1.dat	test_g1.dat	所有H边缘检测到正确方向	100%
2	tesv_i1.dat	test_g1.dat	所有V边缘检测到正确方向	100%
3	tesv_i2.dat	test_g2.dat	所有L边缘检测到正确方向	100%
4	tesv_i2.dat	test_g2.dat	所有R边缘检测到正确方向	100%
5	tesv_i1.dat	test_g1.dat	所有转角检测到正确方向	100%
6	tesv_i3.dat	test_g3.dat	所有单像素点可以被填充	100%
7	tesv_i4.dat	test_g4.dat	多像素点未被填充	100%
8	tesv_i5.dat	test_g5.dat	所有单像素角被填充	100%
9	tesv_i6.dat	test_g6.dat	多像素角未被填充	100%
10	tesv_i7.dat	test_g7.dat	单像素点被忽略	100%
11	tesv_i8.dat	test_g8.dat	噪声过滤器适当忽略	100%
12	Tesv_i9.dat	Test_g9.dat	确认复杂人工图像的所有边界	100%
13	Tesv_i10.dat	Test_g10.dat	产生真实图像的可辨认框架	100%

表11-3　检测边界的方向

	水平边界	垂直边界
1	“110”-上层边界	“000”-左边界
2	“000”-中间边界	“000”-中间边界
3	“　　”-底层边界	“100”-右边界

为了解释测试计划的使用，首先说明如何为测试数为1和2的两个测试得到测试结果。这两个测试的目的是为了确保MUT检测水平和垂直边界，并且把适当的方向赋值给每个边界。

输入测试图像（文件tesv_il.dat）如图11-32a，它是一个包括水平和垂直边界的综合图像。该输入图像的期望值如图11-32b，存放在文件tesv_gl.dat里。边缘检测输出图像由仿真产生，如图11-32c。图像的水平和垂直边界都被正确地检测。表11-3是仿真输出的水平和垂直边界的方向值。

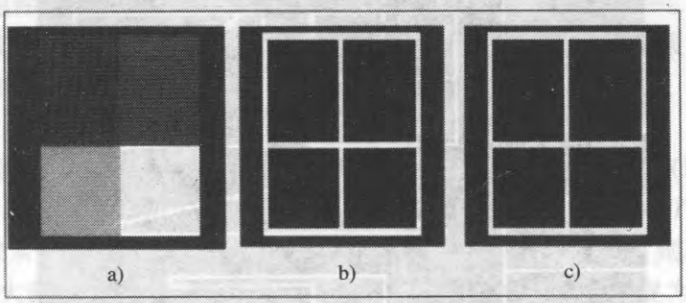


图11-32 输入图像、期望的响应及输出图像

这些方向根据图11-4计算得到。它们的结果对应于表中的可接受结果列。Synopsys仿真程序表明语句的覆盖率是100%。这意味着模型中的每一条指令都至少执行了一次。

图11-33和图11-34是在一个复杂人工图像中根据测试数12被正确确认出的所有边界。

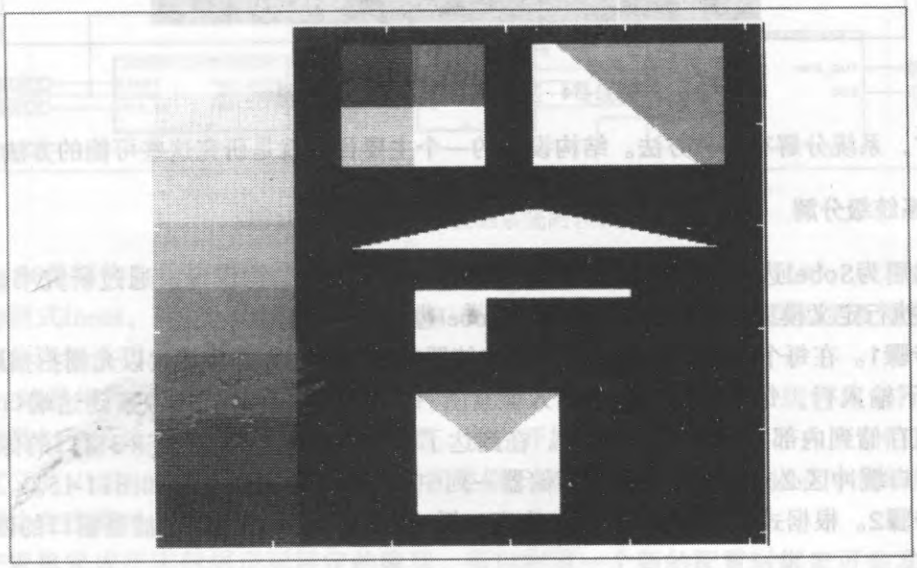


图11-33 复杂人工图像

11.5 结构设计

结构设计时，系统被分解为一些子系统。在前面的设计级别中，系统定义被求精并在VHDL可执行定义模型里获取。工作在结构设计级的设计者可以通过构建可执行定义模型或通过研究由其他小组设计的模型，为整个系统生成文档。基于从该文档中获取的信息，系统被分解为更小的子系统模块。这些子系统可以在同一抽象级或更低一级作为组件来建立模型，必须为每个子系统定义接口和功能。图11-1所示的设计方法必须应用到每一个子系统，从书

面说明书和一个可执行规格说明书开始。然后为原始系统产生一个反映了系统分解的VHDL结构模型。这个结构化模型定义了子系统组件之间的互连。显然，系统的完整接口和功能应该保持与原始可执行定义里的定义相同。

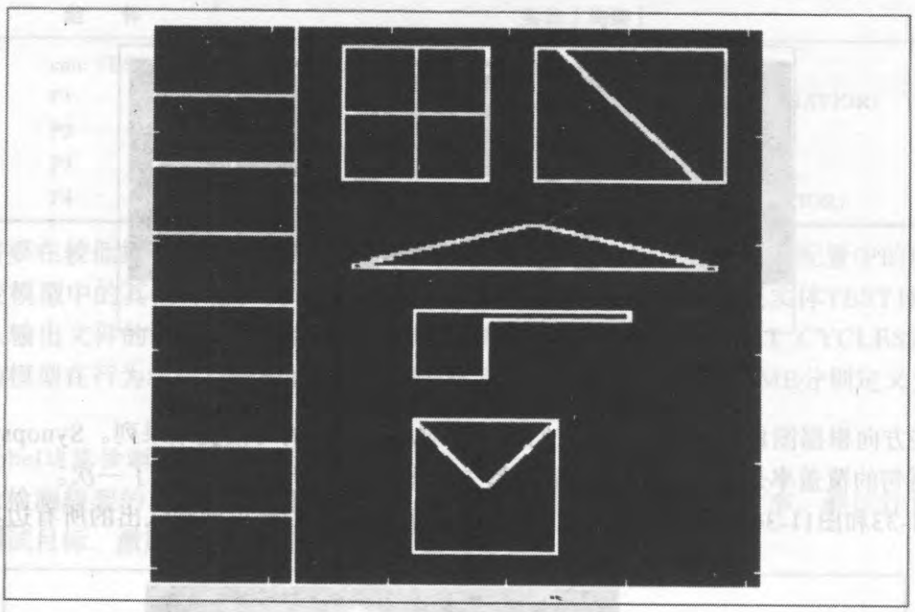


图11-34　复杂人工图像的边界检测

通常，系统分解有多种方法。结构设计的一个主要目标就是研究这些可能的方法。

11.5.1　系统级分解

我们用为Sobel边缘检测系统建立一个结构模型来说明结构设计。通过研究书面说明及VHDL可执行定义模型，可用三个步骤处理Sobel边缘检测算法。

1) **步骤1**。在每个时钟周期，更新内部存储器的数据结构。图像像素以光栅扫描顺序到达（自顶而下输入行，每一行从左到右输入像素），每个时钟周期有一个像素到达端口INPUT。该像素值存储到内部3行存储器数组中。在到达了足够组成一个完整的3 × 3窗口的像素之后，过滤器窗口缓冲区必须通过读入内部存储器一列中的三位来进行更新，如图11-15。

2) **步骤2**。根据式(11-1)~式(11-4)，为窗口缓冲区的像素计算当前过滤器窗口的四个Sobel过滤函数。

3) **步骤3**。决定当前窗口的中间像素是否为边界像素。这一步需要计算四个过滤器输出的最大绝对值、最大值垂直方向上的绝对值，并且根据式(11-5)来计算幅度。如果幅度超过了定义的阈值，则中间的像素是边界像素。然后为它赋值适当的像素值，并且根据图11-4为它计算方向。

系统分解需要设计者具有一定的洞察力。设计者可能需要对系统进行多种分解。图11-35是基于刚才讨论的三个步骤对Sobel边缘检测系统进行分解的一种方法。

1) **存储器处理组件**。组件MEMORY_PROCESSOR包括三行内部存储器缓冲。在每个时钟周期，它把来自端口MEN_IN的像素存储到内部存储器缓冲区的适当位置。该组件也可以

从内部存储器缓冲区里读出适当的三个像素列，并且通过端口MEM_OUT1、MEM_OUT2、MEM_OUT3把它们发送到窗口处理组件，然后窗口处理器更新当前过滤器窗口。存储器处理组件需要有系统时钟CLOCK，并且在处理一幅新图像时需要系统开始信号（端口START）初始化存储器缓冲区的地址指针。

2) **窗口处理组件**。组件WINDOW_PROCESSOR包括3×3窗口缓冲。在每个时钟周期，它从输入（P1，P2，P3）读入一个新的数据列并且更新如图11-15的窗口缓冲。然后它计算四个过滤器函数并输出端口W_H（水平密度斜率）、W_V（数值密度斜率）、W_DL（左对角密度斜率）及W_DR（右对角密度斜率）的四个密度斜率。

3) **幅度处理组件**。这个组件从输入端口读入四个密度斜率，并根据式(11-5)计算幅度斜率及根据图11-4计算最大斜率方向。如果幅度梯度超过了阈值（从输入端口THRESHOLD），则当前窗口的中间像素是一个边界像素。如果中间像素是一个边界像素，则MUG_OUT置为前景值，DIR置为最大斜率的方向。如果中间像素不是边界，MAG_OUT置为背景值，DIR的取值任意。

对应的VHDL结构代码可以由SGE自动产生，或可以根据图11-35的框图很容易地构造出结构化的构架。图11-36和图11-37是图11-35用框图表示的Sobel边缘检测系统的VHDL结构化的构架。

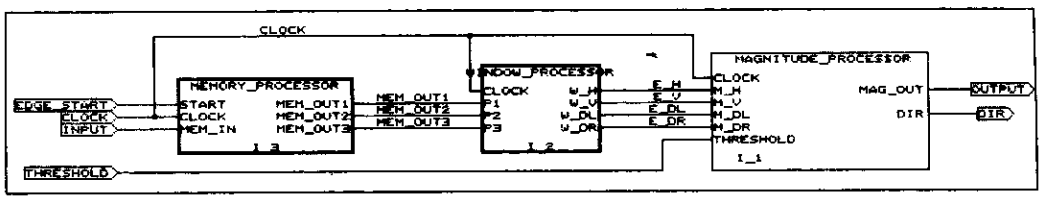


图11-35 Sobel边缘检测系统的顶级分解

图11-36是结构化构架的组件声明。延时类属被加入到修改后的内部延时，所有的输出都被声明为模式inout，所以，配置可以用于端口类型转换函数。混合数据类型的配置将在本章的后面部分详细说明。

图11-37是组件间互连的信号声明及组件的实例。设计者必须为每个组件建立行为模型。这个过程类似于系统定义级的Sobel行为模型。现在开始描述这三个组件，并为它们产生作为可执行定义的VHDL行为模型。由于这个过程类似于顶级Sobel模型，所以在本章结束时才讨论这些模块在建模时会遇到的问题。

在配置里完成把实例绑定到特定的模型，所以在写一个新的配置时绑定可能发生变化。结构化构架声明将不会改变。采用这种方法，可以把实例与行为模型进行绑定，而且在以后如果组件被进一步分解时，可以把实例与结构模型绑定。

图11-38是组件与行为模型的绑定。

第一级结构分解的测试包

在图11-19到图11-26里定义了可执行定义的测试包模型BEH_INT.TEST(BENCH)，它可以在Sobel边缘检测系统的结构化构架里重用。如果已经使用了配置的方法，则不需要重新分析BEH_INT.TEST(BENCH)里的VHDL代码。结构模型只分析和仿真测试在图11-39里的新配置CONFIG_S_INT。它重用了同一个测试包BEH_INT.TEST(BENCH)。同样也重用了测试向量

文件tesv_i2.dat和GOLD数据文件test_gm.dat及tast_gd.dat。所有的参数类属声明都被重用。只有两项有所改变。第一是使用了不同的中间数据文件test_o2.dat和test_d2.dat，使用它们是为了不覆盖行为级测试的结果文件。第二是边缘检测器组件（P4:EDGE_DETECTOT1）使用配置STRUCT_INT.CONFIG_SOBEL_S_L1被绑定到图11-38的结构模型，而不是行为模型BEH_INT.EDGE_DETECTOR(BEHAVIOR)。

```
-- structural description of edge detector ---
architecture STRUCTURE of EDGE_DETECTOR is

  -- memory processor component declaration --
  component MEMORY_PROCESSOR1
    generic (NUM_ROWS, NUM_COLS: NATURAL;
             MEM_OUT_DELAY:TIME)
    port(CLOCK: in STD_LOGIC:='0';    -- system CLOCK
         START: in STD_LOGIC:='0';    -- start signal
         MEM_IN: in PIXEL:=0;         -- input pixel
         MEM_OUT1, MEM_OUT2, MEM_OUT3: inout PIXEL:=0);
  end component;

-- window processor component declaration --
  component WINDOW_PROCESSOR1
    generic(HORIZ_DELAY, VERT_DELAY, LEFT_DIAG_DELAY,
            RIGHT_DIAG_DELAY, WAIT_TIME:TIME)
    port(CLOCK: in STD_LOGIC:='0';  -- system CLOCK
        P1,P2,P3: in PIXEL:=0;      -- input pixels
        W_H: inout FILTER_OUT:=0;  -- horizontal filter
        W_V: inout FILTER_OUT:=0;  -- vertical filter
        W_DL: inout FILTER_OUT:=0; -- left diagonal filter
        W_DR: inout FILTER_OUT:=0);-- right diagonal filter
  end component;

-- magnitude processor component declaration --
  component MAG_PROCESSOR1
    generic(MAG_DELAY: TIME); -- magnitude processor delay
    port(CLOCK: in STD_LOGIC:='0';  -- system CLOCK
         M_H,M_V,M_DL,M_DR: in FILTER_OUT:=0;  -- filter
           -- values from the window processor
         THRESHOLD: in FILTER_OUT:=0;  -- threshold value
         MAG_OUT: inout PIXEL:= 0; -- edge pixel value
         DIR: inout DIRECTION :="000"); -- edge direction
  end component;
```

图11-36　Sobel边缘检测系统的结构化构架第1部分

11.5.2　层次分解

系统级分解是一个迭代过程，它一直分解到所有的底端模块都易于综合时停止。例如，WINDOW_PROCESSOR可以进一步分解为四个过滤器组件（HOR_FILTER，LEFT_FILTER，RIGHT_FILTER，VERT_FILTER），如图11-40。类似地，组件MEM_PROCESSOR可以进一步分解为一个地址产生器组件（ADDR_GEN）和一个存储器组件（MEMORY）。第二级组件的书写定义如下：

1）**地址产生器组件**。组件ADDR_GEN接收时钟输入（CLOCK）和操作开始输入（START），然后对它的所有变量初始化。这个模块产生存储器读控制信号（RD）和存储器写

控制信号（WR），这两个信号控制内部存储器组件（MEMORY）的读和写。如果当前的时钟有一个写操作，ADDR_GEN在端口x_addr1为写操作发出行地址，在端口y_addr为写操作发出列地址，并在时钟的低电平期间产生一个写使能控制信号（WR='1'）。如果窗口缓冲器在当前时钟需要更新内容，ADDR_GEN在端口y_addr发出这三个读操作的列地址，在端口x_addr1发出窗口最上面元素要读的行地址，在端口x_addr2发出窗口中间元素要读的行地址，在端口x_addr3发出窗口最下面元素要读的行地址，并且在时钟的高电平期间产生一个读使能控制信号（RD='1'）。

```
-- intermediate signals between the components --
  signal E_H,E_V,E_DL,E_DR: FILTER_OUT:=0;
  signal MEM_OUT1, MEM_OUT2, MEM_OUT3: PIXEL:=0;

begin
    ---------- component instantiation -------
MEMP1: MEMORY_PROCESSOR1
  generic map(NUM_ROWS => NUM_ROWS, NUM_COLS => NUM_COLS
              MEM_OUT_DELAY => 2 ns)
  port map(CLOCK, EDGE_START, INPUT, MEM_OUT1,
           MEM_OUT2, MEM_OUT3);

WINP: WINDOW_PROCESSOR1
  generic map(HORIZ_DELAY => 3 ns, VERT_DELAY=> 3 ns,
              LEFT_DIAG_DELAY => 3 ns,
              RIGHT_DIAG_DELAY => 3 ns,
              WAIT_TIME => 0 ns)
  port map(CLOCK, MEM_OUT1, MEM_OUT2, MEM_OUT3,
           E_H, E_V, E_DL, E_DR);

MAGP: MAG_PROCESSOR1
  generic map( MAG_DELAY => 3 ns)
  port map(CLOCK, E_H, E_V, E_DL, E_DR,THRESHOLD,
           OUTPUT, DIR)
end STRUCTURE;
```

图11-37 Sobel边缘检测系统的结构化构架第2部分

```
library STRUCT_INT;
use STRUCT_INT.all;

configuration CONFIG_SOBEL_S_L1 of EDGE_DETECTOR is
  for STRUCTURE

    for MEMP1: MEMORY_PROCESSOR1
      use entity STRUC_INT.MEMORY_PROCESSOR(BEHAVIOR);
    end for;

    for WINP: WINDOW_PROCESSOR1
      use entity STRUC_INT.WINDOW_PROCESS(BEHAVIOR);
    end for;

    for MAGP: MAG_PROCESSOR1
      use entity STRUC_INT.MAG_PROCESS(BEHAVIOR);
    end for;
  end for;
end;
```

图11-38 Sobel边缘检测系统顶级分析的配置声明

```
configuration CONFIG_S_INT of TB_CONFIG is
  for TEST_BENCH
    for con:TEST1 use entity BEH_INT.TEST(BENCH)
      generic map("tesv_i2.dat", "test_o2.dat",
                  "test_d2.dat", 10,10,5);
      for BENCH
        for P1:CLOCK_GENERATOR1 use entity
          BEH_INT.CLOCK_GENERATOR(BEHAVIOR)
          generic map(HI_TIME=>75 ns,LO_TIME=>25 ns);
        end for;
        for P2:COMP_MAG1
          use entity BEH_INT.COMP_MAG(COMPARE)
          generic map("test_o2.dat","test_gm.dat",10,10);
        end for;
        for P3:COMP_DIR1
          use entity BEH_INT.COMP_DIR(COMPARE)
          generic map("test_d2.dat","test_gd.dat",10,10);
        end for;
        for P4:EDGE_DETECTOR1
          use entity STRUC_INT.CONFIG_SOBEL_S_L1
          generic map(NUM_ROWS=>10,NUM_COLS=>10);
        end for;
      end for;
    end for;
  end for;
end;
```

图11-39 测试Sobel边缘检测系统结构化构架的测试包的配置声明

2) **存储器组件**。组件MEMORY包括3行存储器像素数组。在输入时钟的上升沿，如果WR='1'，存储器组件把端口Mem_in的新的像素写入行地址为x_addr1、列地址为y_addr的存储器数组中。在输入时钟的下降沿，如果RD='1'，存储器组件从内部存储器数组中并发读出三个像素。行地址为x_addr1、列地址为y_addr的像素发送到输出端口mem_out1。行地址为x_addr2、列地址为y_addr的像素发送到输出端口mem_out2。行地址为x_addr3、列地址为y_addr的像素发送到输出端口mem_out3。这三个像素发送到窗口处理模块以更新图11-15所示的窗口缓冲区的内容。

3) **水平过滤器组件**。组件HOR_FILTER执行式(11-1)的水平过滤器操作。窗口缓冲区最上面一行的像素到达输入端口P1，最下面一行的像素到达端口P3。过滤器不需要考虑窗口缓冲区里中间一行的像素。每个时钟周期到来一个像素。这个组件自身在内部保留缓冲窗口第一行和第三行像素的副本。

4) **垂直过滤器组件**。组件VERT_FILTER执行式(11-2)的垂直过滤器操作。窗口缓冲区最上面一行的像素到达输入端口P1，中间一行的像素到达端口P2，最下面一行的像素到达端口P3。每个时钟周期到来一个像素。这个组件自身在内部保留窗口缓冲区内所有三行像素的副本。

5) **左对角过滤器组件**。组件LEFT_FILTER执行式(11-3)的左对角过滤器操作。窗口缓冲区最上面一行的像素到达输入端口P1，中间一行的像素到达端口P2，最下面一行的像素到达端口P3。每个时钟周期到来一个像素。这个组件自身在内部保留窗口缓冲区内所有三行像素的副本。

6) **右对角过滤器组件**。组件RIGHT_FILTER执行式(11-4)的右对角过滤器操作。窗口缓冲

区最上面一行的像素到达输入端口P1，中间一行的像素到达端口P2，最下面一行的像素到达端口P3。每个时钟周期到来一个像素。这个组件自身在内部保留窗口缓冲区内所有三行像素的副本。

注意 没有考虑分解方法的优化。其他的分解方法也许更好。

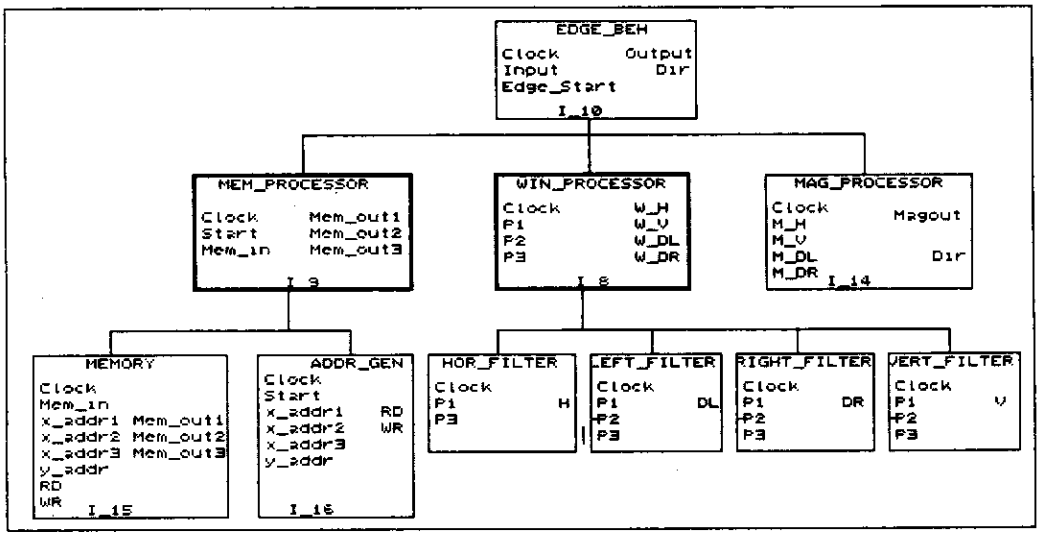


图11-40 Sobel边缘检测系统设计的层次结构

11.5.3 为层次结构模型产生测试包的方法

本节介绍一种测试层次结构模型的方法。尽管层次结构模型的设计使用了自顶向下的方法，但测试时使用自低向上的测试。

1) 首先使用单独的测试包测试层次低端所有独立的组件。它验证每个叶模块行为的性能。

2) 测试完所有的低层模块，再测试层次上一级的模块。这些模块是使用了最低层次组件的结构模型。如果使用了有效的自顶向下方法设计过程，在每一级都包括了产生测试包的配置，这些用来测试行为模型的配置只要稍作修改，就可以产生一个新的测试结构模型的配置。使用了相同测试向量的同一个测试包可以被重新使用。

3) 在层次的最高一级，可执行定义的原始测试包可以对它的原始配置稍作修改，就可以被重用来测试完整的层次结构模型。

由于测试过程从低向上工作，在自顶向下设计方法里使用的测试包可以在层次结构模型的自低而上策略里被重用——该模型为自顶向下的每一级都提供了配置。因此只需对每一级的过程中的配置进行分析和仿真，而无须使用更复杂的测试包。这样可以节省大量的计算时间和存储空间。还有一个好处是它易于跟踪纪录文档。如果以后对模型做了一些变化，可以根据文档恢复修改了的设计。

为了说明这种方法，假设设计小组刚刚设计出水平过滤器组件HOR_FILTER，但还没有对它进行测试。假设组件LEFT_FILTER、RIGHT_FILTER及VERT_FILTER已经被设计出，并且使用独立的测试包进行了测试，但是这些组件并没有作为中间级组件WIN_PROCESSOR的结构组件进行测试。然而，WIN_PROCESSOR的行为已经被设计出并做了测试，它

不仅使用自身的行为测试包进行了独立的测试，而且作为顶级EDGE_BEH组件的结构组件进行了测试。进一步假设设计小组使用了配置为过程建模。例如，图11-39是测试顶级结构化构架的配置。

第一步使用组件自身的测试包对底层组件HOR_FILTER进行测试。图11-41是一个适合底层测试的测试包。因为它是一个底层组件，它的配置定义包括在模型中（使用for语句），而不是作为一个独立的配置声明。

```
entity TB is
end TB;
architecture HORIZ_TB_INT of TB is
  signal RUN: STD_LOGIC;
  signal CLOCK:STD_LOGIC:='0';
  signal P1,P3: PIXEL:=0;
  signal H: FILTER_OUT:=0;
  component CLOCK_GENERATOR1
    generic(HI_TIME,LO_TIME:TIME);
    port(RUN: in STD_LOGIC;
         CLOCK: out STD_LOGIC);
  end component;
  component  HORIZONTAL_FILTER1
    generic(HORIZ_DELAY: TIME;
            WAIT_TIME:TIME);
    port(CLOCK: in STD_LOGIC:='0';
         P1,P3: in  PIXEL:=0;
         H: inout FILTER_OUT:=0);
  end component;
  for L1: CLOCK_GENERATOR1
    use entity BEH_INT.CLOCK_GENERATOR(BEHAVIOR);
  for L2: HORIZONTAL_FILTER1
    use entity STRUC_INT.HORIZONTAL_FILTER(BEHAVIOR);
begin
  L1: CLOCK_GENERATOR1
    generic map(HI_TIME=>75 ns,LO_TIME=>25 ns)
    port map(RUN,CLOCK);
  L2: HORIZONTAL_FILTER1
    generic map(HORIZ_DELAY=>1 ns,WAIT_TIME=>100 ns)
    port map(CLOCK,P1,P3,H);
  RUN<='0' after 0 ns,'1' after 5 ns,'0' after 1000 ns;
  P1<=11 after 10 ns,10 after 110 ns,9 after 210 ns,
      12 after 310 ns,13 after 410 ns,48 after 510 ns;
  P3<=8 after 10 ns,40 after 110 ns,7 after 210 ns,
      10 after 310 ns,5 after 410 ns,110 after 510 ns;
end HORIZ_TB_INT;
```

图11-41 水平过滤器的独立测试包

下一步是重用测试窗口处理器行为模型的测试包。图11-42是原始的测试包TB(WINDOW_TB)，它用来测试窗口处理器的行为域模型。这里对组件WINDOW_PROCESSOR1没有使用绑定。图11-43是应用于行为测试的原始配置（WINDOW_BEH_INT）。图11-42的原始测试包TB(WINDOW_TB)可以通过仿真图11-44中的新配置WINDOW_STRUC_INT，重用于加入了新的过滤器的结构模型的测试。唯一的区别是结构模型（STRUCT_INT.WINDOW_PROCESSOR(BIHAVIOR)）被绑定到窗口处理组件（L2:WINDOW_PROCESSOR1）。

```
------------ENTITY TEST BENCH---------------
entity TB is
end TB;
------------ARCHITECTURE BODY---------------
architecture WINDOW_TB of TB is
  signal P1,P2,P3: PIXEL:=0;
  signal CLOCK:  STD_LOGIC:='0';
  signal W_H: FILTER_OUT:=0;
  signal W_V: FILTER_OUT:=0;
  signal W_DL: FILTER_OUT:=0;
  signal W_DR: FILTER_OUT:=0;
  signal RUN: STD_LOGIC:='0';
----------COMPONENT DECLARATIONS--------------
  component CLOCK_GENERATOR1
    port(RUN: in STD_LOGIC;
         CLOCK: out STD_LOGIC);
  end component;
  component WINDOW_PROCESSOR1
    port (CLOCK: in STD_LOGIC:='0';
          P1,P2,P3: in  PIXEL:=0;
          W_H: inout FILTER_OUT:=0;
          W_V: inout FILTER_OUT:=0;
          W_DL: inout FILTER_OUT:=0;
          W_DR: inout FILTER_OUT:=0 );
  end component;
begin
  L1: CLOCK_GENERATOR1
    port map(RUN,CLOCK);
  L2:WINDOW_PROCESSOR1
    port map(clock,P1,P2,P3,W_H,W_V,W_DL,W_DR);
  RUN<='0' after 0 ns, '1' after 5 ns, '0' after 1000 ns;
  P1<=  11 after 10 ns, 10 after 110 ns, 9 after 210 ns,
        12 after 310 ns,13 after 410 ns,48 after 510 ns;
  P2<=6 after 10 ns,  5 after 110 ns,14 after 210 ns,
     13 after 310 ns,10 after 410 ns,19 after 510 ns;
  P3<=8 after 10 ns,40 after 110 ns,  7 after 210 ns,
     10 after 310 ns,5 after 410 ns,110 after 510 ns;
end WINDOW_TB;
```

图11-42 窗口处理器组件的测试包

```
configuration WINDOW_BEH_INT of TB is
  for WINDOW_TB
    for L1:CLOCK_GENERATOR1
      use entity BEH_INT.CLOCK_GENERATOR(BEHAVIOR)
        generic map(HI_TIME=>75 ns,LO_TIME=>25 ns);
    end for;
    for L2: WINDOW_PROCESSOR1
      use STRUC_INT.WINDOW_PROCESSOR(BEHAVIOR)
        generic map(HORIZ_DELAY=>3 ns,
                    VERT_DELAY =>3 ns,
                    LEFT_DIAG_DELAY =>3 ns,
                    RIGHT_DIAG_DELAY =>3 ns,
                    WAIT_TIME=> 0 ns);
    end for;
  end for;
end;
```

图11-43 测试行为模型的窗口处理器测试包的配置

```
configuration WINDOW_STRUC_INT of TB is
  for WINDOW_TB
    for L1:CLOCK_GENERATOR1
      use entity BEH_INT.CLOCK_GENERATOR(BEHAVIOR)
        generic map(HI_TIME=>75 ns,LO_TIME=>25 ns);
    end for;
    for L2: WINDOW_PROCESSOR1
      use configuration STRUC_INT.W_INT
        generic map(HORIZ_DELAY=>3 ns,
                    VERT_DELAY =>3 ns,
                    LEFT_DIAG_DELAY =>3 ns,
                    RIGHT_DIAG_DELAY =>3 ns,
                    WAIT_TIME=> 0 ns);
    end for;
  end for;
end;
```

图11-44 测试结构模型的窗口处理器测试包的配置

最后，图11-19到图11-26的顶级测试包可以通过用一个新配置取代图11-31里旧的配置重用它，对加入了新的过滤器组件的整个新系统进行测试。图11-45是新的配置。实体CONFIG_SOBEL_S_L2是包含了新的窗口构架的配置。这个例子说明了在每一级都使用配置的情况下，当设计过程不断进入低层时如何重用测试包。

```
configuration CONFIG_S2_INT of TB_CONFIG is
  for TEST_BENCH
    for con:TEST1 use entity BEH_INT.TEST(BENCH)
      generic map("tesv_i2.dat", "test_o2.dat",
                  "test_d2.dat", 10,10,5);
      for BENCH
        for P1:CLOCK_GENERATOR1 use entity
              BEH_INT.CLOCK_GENERATOR(BEHAVIOR)
          generic map(HI_TIME=>75 ns,LO_TIME=>25 ns);
        end for;
        for P2:COMP_MAG1 use entity WORK.COMP_MAG(COMPARE)
          generic map("test_o2.dat","test_gm.dat",10,10);
        end for;
        for P3:COMP_DIR1 use entity WORK.COMP_DIR(COMPARE)
          generic map("test_d2.dat","test_gd.dat",10,10);
        end for;
        for P4:EDGE_DETECTOR1 use entity CONFIG_SOBEL_S_L2
          generic map(NUM_ROWS=>10,NUM_COLS=>10);
        end for;
      end for;
    end for;
  end for;
end;
```

图11-45 测试窗口处理器结构模型的顶级测试包的配置声明

11.6 寄存器传输级详细设计

现在进入详细设计阶段。前一阶段是结构设计阶段，它通过把定义分解到易于管理的子系统来产生结构模型。现在需要详细设计出层次分解的底层部件。达到门级设计的最直接的

方法是首先产生一个寄存器传输级（RTL）模型，然后使用综合工具将其转换为门级电路。本节详细讨论获得RTL模型的方法。在下一节将讨论综合步骤。

11.6.1 寄存器传输级设计

同它的名字一样，RTL模型描述了不同寄存器之间的数据传输。对数据的基本操作包括了算术操作（加法、减法、乘法，等等），逻辑操作（或、与、非，等等），移位和循环操作，以及许多其他的类似操作。本节首先只考虑同步电路，而异步电路的设计也是可能的。产生RTL模型有以下四个步骤。第12章里会详细讨论自动化设计过程的方法。

1) **步骤1**。设计一个寄存器传输级部件时，首先分析该部件的行为模型以决定所需要的寄存器的类型和数量，以及所需要执行的数据操作的类型。

2) **步骤2**。决定操作执行的顺序。这一步通常被称为调度。

3) **步骤3**。将操作映射到硬件计算部件并决定结果的存放位置。这一步通常被称为分配。

4) **步骤4**。产生一个VHDL数据流模型或结构模型来描述RTL设计。

下面通过介绍如何为图11-40所示的层次分解底层中的水平过滤器部件产生一个RTL结构化模型，来解释这种方法。

在RTL级设计水平过滤器组件

图11-46是水平过滤器的行为模型。在端口P1和P3接收两个连续的像素流数据。水平过滤器的计算结果从端口H发送出去，它也是一个连续流。过滤器窗口的最上面一行的像素到达端口P1，最下面一行的像素到达端口P3。行为模型在内部缓冲区FIRST_LINE和THIRD_LINE存储每个输入数据流的三个像素。

该行为模型使用了两个延时：1) HORIZ_DELAY表示过滤器部件的内部延时。2) WAIT_TIME表示混合级仿真时为了平衡而人工插入的延时。在不同抽象级模型的不同部分，这个延时值也不同。通过使用独立参数表示两种不同的作用，可以很容易跟踪表示真实延时的参数，把两个参数用一个来表示则会造成混淆。

水平过滤器对内部缓冲区的每个像素执行WEIGHT函数，并且计算两两值之间的差别。图11-47是WEIGHT函数的VHDL代码。我们将把过滤器最上面一行的像素表示为（X1P1，X2P1，X3P1），把最下面一行的像素表示为（X1P3，X2P3，X3P3）。在所有像素到达之后，这段VHDL代码表示如下的计算过程：

$$H = ((X1P1)+2*(X2P1)+(X3P1)) - ((X1P3) +2*(X2P3)+(X3P3)) \quad (11\text{-}16)$$

这些像素以顺序（X1P1，X1P3）、（X2P1，X2P3）、（X3P1，X3P3）成对到达端口P1和P2。可以根据像素到达的方式把表达式扩展如下：

$$H=((X1P1-X1P3) +2*(X2P1-X2P3) +(X3P1-X3P3)) \quad (11\text{-}17)$$

现在考虑扣除同时到达的一对像素。因为像素是持续到达的，所以需要考虑一个流水线结构。需要有执行加法、减法及乘以2的硬件。由于数据到达的持续性，这些硬件不能被共享使用。这意味着需要一个直线调度，以及一个简单的、对硬件单元操作的一对一映射（分配）。

图11-48是组织一系列寄存器和计算单元来完成流水线操作的一种方法。在Sobel算法描述里的像素值是从1~255之间的正整数。在RTL级，用二进制向量表示所有的数据。需要用八位来表示该范围内的整数。所以输入端口P1和P2都是8位二进制的正整数。在内部，由于计算包

括了减法，结果可能是负数。因此，需要一个额外的位来表示结果的符号。在后面的步骤里相加了两次，精确表示结果可能需要超过8位。因此，增加了两个额外的位来精确表示一个较大的和。一共需要的位数是11位。为了使用标准的寄存器和标准总线尺寸，在过滤器里使用12位的数据通路。因此，在内部使用一个包括2位符号位的12位二进制数表示。

```
library IEEE;
use ieee.std_logic_1164.all;
library STRUC_INT;
use STRUC_RTL.IMAGE_PROCESSING.all;
-- interface declaration -------------------------
entity HORIZONTAL_FILTER is
  generic(HORIZ_DELAY: TIME; -- horizontal filter delay
          WAIT_TIME: TIME); -- abstraction level match
  port(CLOCK: in STD_LOGIC:='0';
       P1,P3: in PIXEL:=0;
       H: inout FILTER_OUT:=0);
end HORIZONTAL_FILTER;
-- behavioral architecture declaration ------------
architecture BEHAVIOR of HORIZONTAL_FILTER is
  signal TEMP1: FILTER_OUT:=0;  -- intermediate signal
begin
  H_FILTER: process
    variable TEMP_H: FILTER_OUT:=0;  -- temporary storage
    variable FIRST_LINE: PIX3:=(0,0,0); -- first scan line
    variable THIRD_LINE: PIX3:=(0,0,0); -- third scan line
  begin
    wait until rising_edge(CLOCK);
--- store the input pixels in the 3-stage buffers ----
    FIRST_LINE:=SHIFT_LEFT(FIRST_LINE,P1);
    THIRD_LINE:=SHIFT_LEFT(THIRD_LINE,P3);
TEMP_H:=WEIGHT(FIRST_LINE(1),FIRST_LINE(2),FIRST_LINE(3))
     -WEIGHT(THIRD_LINE(1), THIRD_LINE(2), THIRD_LINE(3));
    TEMP1 <= TEMP_H after HORIZ_DELAY;
  end process H_FILTER;
---- make H as a delayed version of temp1 -------------
  H <= TEMP1 after WAIT_TIME; -- Horizontal filter output
end BEHAVIOR;
```

图11-46 水平过滤器的VHDL行为模型

```
----- WEIGHT function -----------------
function WEIGHT
   (X1,X2,X3: PIXEL)
    return FILTER_OUT is
begin
    return X1+ 2*X2 + X3;
end WEIGHT;
```

图11-47 WEIGHT函数

输入数据是8位正整数，所以需要在每个数据的最左端扩展4位0，转换为12位双符号数来表示。部件EXT8_12A和EXT8_12B执行两个数据流的转换任务。部件DELAYx是12位寄存器。部件SUM1和SUM2是12位双符号数的加法器，Diff1是12位双符号数减法器。这些部件的VHDL模型都可以在第4章中找到。过滤器输出H是一个12位双符号数。所有RTL级函数的完

整代码都可以在CD-ROM里对应本章的地方找到。

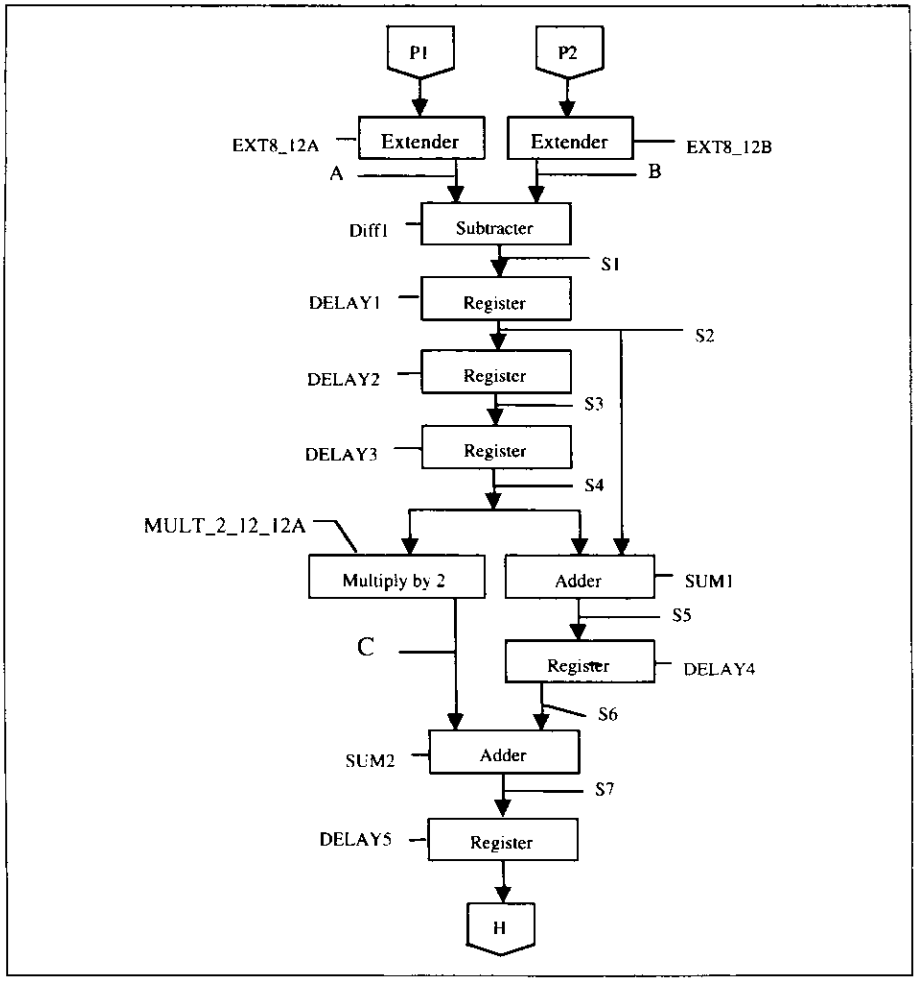


图11-48 水平过滤器RTL模型的数据流图

数据对（X1P1，X1P3）、（X2P1，X2P3）、（X3P1，X3P3）在三个连续的时钟周期里到来时，减法器Diff1记录从流水线进来的数据，计算(X1P1-X1P3)、(X2P1-X2P3)、(X3P1-X3P3)。三个时钟周期之后，计算结果为：

```
S2=X3P1-X3P3
S3=X2P1-X2P3
S4=X1P1-X1P3
S5=(X1P1-X1P3)+(X3P1-X3P3).
```

在4个时钟之后，可以得到以下结果：

```
S6=(X1P1-X1P3)+(X3P1-X3P3)
S4=(X2P1-X2P3)
C=2*(X2P1-X2P3)
S7=((X1P1-X1P3)+(X3P1-X3P3))+2*(X2P1-X2P3)
```

最后，在5个时钟周期后可以得到需要的输出H。

```
H=(X1P1-X1P3)+2*(X2P1-X2P3)+(X3P1-X3P3)
```

从RTL图得到的水平过滤器结构模型的直线代码如图11-48所示。图11-49是结构模型，图11-50和图11-51是结构模型的配置。读者应该仔细研究这些图，这里引入了“空组件声明”这一新的概念。通过使用空组件声明，端口的数据类型可以不同于配置。它允许使用连接配置声明端口的类型转换函数来构造混合数据类型的结构模型。混合数据类型的结构模型将在下一节里介绍。

```
entity HORIZONTAL_FILTER is
  generic(HORIZ_DELAY:TIME);
  port(CLOCK: in STD_LOGIC:='0';
       P1,P3: in PIXEL:="00000000";
       H: inout STD_LOGIC_VECTOR(11 downto 0));
end HORIZONTAL_FILTER;
architecture STRUCTURE of HORIZONTAL_FILTER is
------ Empty component declarations --------
  component SUM12_PM      end component; -- Subtracter
  component REG12         end component; -- Register
  component SUM12_PP      end component; -- Adder
  component EXT8_12       end component; -- Extender
  component MULT_2_12_12 end component; -- Multiply by 2
---- intermediate signal declarations -------
  signal S1,S2,S3,S4,S5,S6,S7,A,B,C:
    STD_LOGIC_VECTOR(11 downto 0);
begin
----- empty component instantiations ---------
  EXT8_12A: EXT8_12;
  EXT8_12B: EXT8_12;
  DIFF1   : SUM12_PM;
  DELAY1  : REG12;
  DELAY2  : REG12;
  DELAY3  : REG12;
  SUM1    : SUM12_PP;
  DELAY4  : REG12;
  MULT_2_12_12A: MULT_2_12_12;
  SUM2    : SUM12_PP;
  DELAY5  : REG12;
end STRUCTURE;
```

图11-49 水平过滤器的RTL结构模型

图11-49的RTL结构模型包括12位双符号数减法器（SUM12_PM）、12位寄存器(REG12)、12位双符号数加法器（SUM12_PP）、8到12位扩展器（EXT8_12）以及完成乘2操作的组件（MULT_2_12_12）。部件类型中实例的名称对应了图11-48数据流图中的标记。由于该模型是流水线模型，而且没有执行乘法操作的组件，因此，映射是一对一的，所以图11-48中的每一个操作都映射到不同的部件。第12章里将对更复杂的映射算法进行讨论。

图11-50和图11-51都是RTL模型的配置声明。端口映射和类属映射都包含在这个声明中，它们不在图11-49中的空部件声明中。这种方法将在下一节加速构建混合数据类型的仿真模型里被使用到。

11.6.2 使用不同数据类型的组件仿真结构模型

在设计周期的这一步，完成了水平过滤器的RTL模型设计。假设已经把RTL模型作为一个

标准单独部件进行了测试。现在希望把它集成为一个窗口处理器的结构模型并且进行测试，这个窗口处理器使用了新的水平过滤器的RTL级模型作为其中一个部件。进一步假设另外一个设计小组在进行其他过滤器的RTL模型建模，他们尚未完成自己的工作。我们希望用水平过滤器的RTL模型替代窗口处理器里的结构模型，而且保留其他过滤器的行为模型。理想情况是写一个窗口处理器的配置，它用水平过滤器的新RTL模型替代绑定在窗口处理器模型中水平过滤器部件的行为模型。例外的情况是两个模型端口使用了不同的数据类型。因此，结构模型的内部互连型号必须也是整数数据类型。如果用结构化模型替代一个使用了标准逻辑数据结构的模型，将会存在类型匹配错误。VHDL里并不允许这种类型匹配错误的出现。

```
configuration H_RTL of HORIZONTAL_FILTER is
  for STRUCTURE
---------------- A <= "0000" & P1 ----------------
    for EXT8_12A:EXT8_12 use entity STRUC_RTL.EXT8_12(BEHAVIOR)
      port map(P1,A);
    end for;
---------------- B <= "0000" & P3 ----------------
    for EXT8_12B:EXT8_12 use entity STRUC_RTL.EXT8_12(BEHAVIOR)
      port map(P3,B);
    end for;
---------------- S1 <= A - B ---------------------
    for Diff1:SUM12_PM use entity STRUC_RTL.SUM12_PM(PARTS)
      port map(A,B,S1);
    end for;
----------------- S2 <= S1 ----------------
    for DELAY1:REG12 use entity STRUC_RTL.REG12(PARTS)
      generic map(HORIZ_DELAY)
      port map( S1, S2, CLOCK );
    end for;
----------------- S3 <= S2 ----------------
    for DELAY2:REG12 use entity STRUC_RTL.REG12(PARTS)
      generic map(HORIZ_DELAY)
      port map( S2, S3, CLOCK );
    end for;
----------------- S4 <= S3 ----------------
    for DELAY3:REG12 use entity STRUC_RTL.REG12(PARTS)
      generic map(HORIZ_DELAY)
      port map( S3, S4, CLOCK );
    end for;
----------------- S5 <= S4 + S2 ----------------
    for SUM1:SUM12_PP use entity STRUC_RTL.SUM12_PP(PARTS)
      port map( S4, S2, S5 );
    end for;
----------------- S6 <= S5 ----------------
    for DELAY4:REG12 use entity STRUC_RTL.REG12(PARTS)
      generic map(HORIZ_DELAY)
      port map( S5, S6, CLOCK );
    end for;
```

图11-50 水平过滤器RTL模型的配置

如果从合作伙伴以外获得模型，也可能存在类似问题。那些模型的数据类型可能与我们所使用的数据类型不同。

在使用不同数据类型的组件产生一个结构模型时，可能会遇到同样的数据类型匹配错误。一种解决方法是为所有具有不同端口数据类型的模型复制多个副本。这种方法使用了大量的

存储器，而且造成了严重的维护问题。如果模型需要被更新，则必须找到所有副本的位置并对每一个副本进行更新。第二种解决方法是写出多个结构模型——对每一种部件数据类型的结合都写出一个模型。如果使用这种系统，结构模型的个数随着部件的个数和数据类型的种类迅速增加。同样，由于对结构模型进行更新和修改，需要找到所有的副本对每一个模型进行更新，从而造成维护的困难。这个问题显然是不能被接受的。

```
------------ C <= S4(10 downto 0) & '0' ----------------
    for MULT_2_12_12A:MULT_2_12_12
      use entity STRUC_RTL.MULT_2_12_12(BEHAVIOR)
      port map(S4,C);
    end for;
----------------- S7 <= C + S6 -------------------------
    for SUM2:SUM12_PP use entity STRUC_RTL.SUM12_PP(PARTS)
      port map(C, S6, S7);
    end for;
------------------ H <= S7 ----------------
    for DELAY5:REG12 use entity STRUC_RTL.REG12(PARTS)
      generic map(HORIZ_DELAY)
      port map( S7, H, CLOCK );
    end for;
  end for; -- End STRUCTURE.
end; -- End configuration.
```

图11-51　水平过滤器RTL模型的配置（续）

现在假设使用了一个普通的结构模型，并且所有需要的数据类型组合都使用一个独立的配置。如果这个结构模型需要被更新，则很容易找到该模型的所有配置，这是因为它们是同一个实体和构架的配置。配置文件通常比结构模型文件要小，因此存储它们所占用的空间也比较小。而且，这里不需要为所有不同数据类型的部件建立多个副本，只需在配置文件中为每个端口赋值语句加上一个类型转换函数。如果该配置恰好是一个测试包，对不同抽象级部件组合的不同配置只需复制一个副本。参见本章前面一节“为层次结构模型产生测试包的方法”。

为了使用不同数据类型的组件来构建一个结构模型，将利用VHDL允许在端口映射语句的关联表中使用类型转换函数的性质。下面将以前面一节中水平过滤器的RTL模型集成为窗口处理器的结构模型为例，说明这种方法的使用。RTL模型在端口中需要使用标准逻辑数据类型；而在高一抽象级别的原始结构模型中使用的是整数数据类型。

混合数据类型仿真的示例：窗口处理器

图11-52是窗口处理器结构模型的VHDL代码。它使用了空的部件声明和这些部件的空实例。类属和端口声明在部件声明里被省略了，类属和端口映射在实例里被省略了。这种情况的一种直观化的方法是使用芯片插槽的类比方法。空的部件声明里定义了一系列空的芯片插槽及互连信号的名称。图11-53是窗口处理器结构模型声明里定义的框图。空的插槽表示四个部件。

对应于部件声明里所定义的部件名称的插槽类型出现在插槽符号中。本例里，每个插槽都有一个唯一的类型（HORIZONTAL_FILTER1、VERTICAL_FILTER1、LEFT_DIAG_FILTER1、RIGHT_DIAG_FILTER1）。通常几个插槽可以是同一类型。芯片插槽的类比非常容易被直观化。不同的芯片类型需要不同的插槽类型。例如，一个40引脚的芯

片当然不符合为14个引脚的芯片所设计的插槽。理解了这一点，芯片插槽的类比对于可视化结构模型声明的意图非常有用。结构模型声明里定义了一系列芯片插槽类型（对应一系列芯片类型）及一系列互连信号。插槽为空，则表示此时没有互连线。

```
entity WINDOW_PROCESSOR is
  generic(HORIZ_DELAY,VERT_DELAY,LEFT_DIAG_DELAY,
          RIGHT_DIAG_DELAY,WAIT_TIME: TIME);
  port(CLOCK: in std_logic:='0';  -- system CLOCK
       P1,P2,P3: in PIXEL:=0;     -- input Pixels
       W_H:  inout FILTER_OUT:=0; -- horizontal output
       W_V:  inout FILTER_OUT:=0; -- vertical output
       W_DL: inout FILTER_OUT:=0; -- left diagonal output
       W_DR: inout FILTER_OUT:=0);-- right diagonal output
end WINDOW_PROCESSOR;
architecture STRUCTURE of WINDOW_PROCESSOR is
------ empty component declarations------------------
  component HORIZONTAL_FILTER1 end component;
  component VERTICAL_FILTER1   end component;
  component LEFT_DIAG_FILTER1  end component;
  component RIGHT_DIAG_FILTER1 end component;
begin
------ empty component instantiations ---------------
  HP: HORIZONTAL_FILTER1;
  VP: VERTICAL_FILTER1;
  LDP: LEFT_DIAG_FILTER1;
  RDP: RIGHT_DIAG_FILTER1;
end STRUCTURE;
```

图11-52 窗口处理器结构模型的VHDL代码

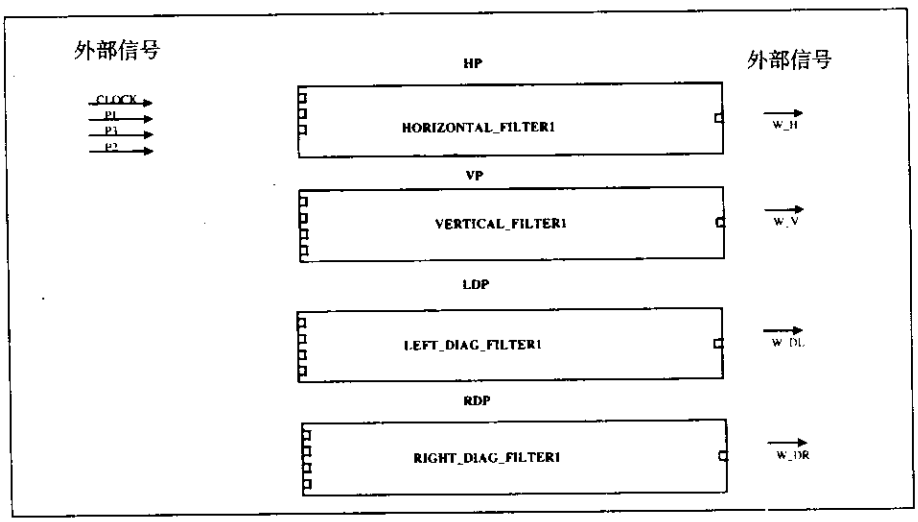


图11-53 图11-52中结构化构架的框图

空的实例声明为每一个被使用的插槽都给出了一个唯一的名称。实例的名称写在插槽符号的外部。实例声明用来把一系列器件映射到一系列部件类型（分配）。

端口和类属映射在配置声明里定义，端口映射关联列表中的项使用了类型转换函数。配置中也定义了将被插入到各个插槽的特定芯片与其他芯片之间的互连信号。图11-54是窗口处

理器的配置，水平过滤器的RTL模型插入到标志为HORIZONTAL_FILTER1的插槽。其他过滤器的行为模型插入到它们对应的插槽中。图11-55是图11-54的配置声明所隐含的框图。由于RTL模型使用了标准逻辑数据类型（STD_LOGIC），其他模型使用的是整数数据类型，因此定义在RTL模型中的每个引脚都使用了类型转换函数。下面的语句定义了连接和必要的类型转换：

```
configuration WINDOW_H_RTL_OTHERS_INTEGER
              of WINDOW_PROCESSOR is
  generic map(HORIZ_DELAY=>3 ns, VERT_DELAY=>3 ns,
        LEFT_DIAG_DELAY=>3 ns,  RIGHT_DIAG_DELAY=>3 ns,
        WAIT_TIME=>0 ns)
  port map(CLOCK, MEM_OUT1, MEM_OUT2, MEM_OUT3,
           E_H, E_V, E_DL, E_DR);
  for structure
    for HP:HORIZONTAL_FILTER1
      use configuration STRUC_RTL.H_RTL
        generic map(HORIZ_DELAY=>HORIZ_DELAY)
        port map (CLOCKH=>CLOCK,
          P1H=>INT_TO_STDLOGIC8(P1),
          P3H=>INT_TO_STDLOGIC8(P3),
          STDLOGIC_TO_INT(H)=>INT_TO_STDLOGIC12(W_H));
    end for;
    for VP: VERTICAL_FILTER1
      use entity STRUC_INT.VERTICAL_FILTER(BEHAVIOR)
        generic map(VERT_DELAY=>VERT_DELAY,
                    WAIT_TIME=>WAIT_TIME)
        port map (CLOCK, P1, P2, P3, W_V);
    end for;
    for LDP: LEFT_DIAG_FILTER1
      use entity STRUC_INT.LEFT_DIAG_FILTER(BEHAVIOR)
        generic map(LEFT_DIAG_DELAY=>LEFT_DIAG_DELAY,
                    WAIT_TIME=>WAIT_TIME)
        port map (CLOCK, P1, P2, P3, W_DL);
    end for;
    for RDP: RIGHT_DIAG_FILTER1
      use entity STRUC_INT.RIGHT_DIAG_FILTER(BEHAVIOR)
        generic map(RIGHT_DIAG_DELAY=>RIGHT_DIAG_DELAY,
                    WAIT_TIME=>WAIT_TIME)
        port map (CLOCK, P1, P2, P3, W_DR);
    end for;
  end for;
end;
```

图11-54 窗口处理器结构模型的配置声明，水平过滤器在RTL级，其他过滤器为更高级别

```
P1_H => INT_TO_STDLOGIC8 (P1)
P3_H => INT_TO_STDLOGIC8 (P3)
```

函数INT_TO_STDLOGIC8是用户定义的函数，它把整数数据类型转换为STD_LOGIC数据类型。内部信号P1和P3的整数值首先被转换为STD_LOGIC类型格式；然后发送到对应的芯片HORIZONTAL_FILTER1的输入引脚P1_H和P3_H。

类似地，HORIZONTAL_FILTER1输出H的STD_LOGIC类型值需要被转换为整数，然后发送到外部信号W_H。下面的语句完成这项任务：

```
STDLOGOC_TO_INT(H) => INT_TO_STDLOGIC12(W_H);
```

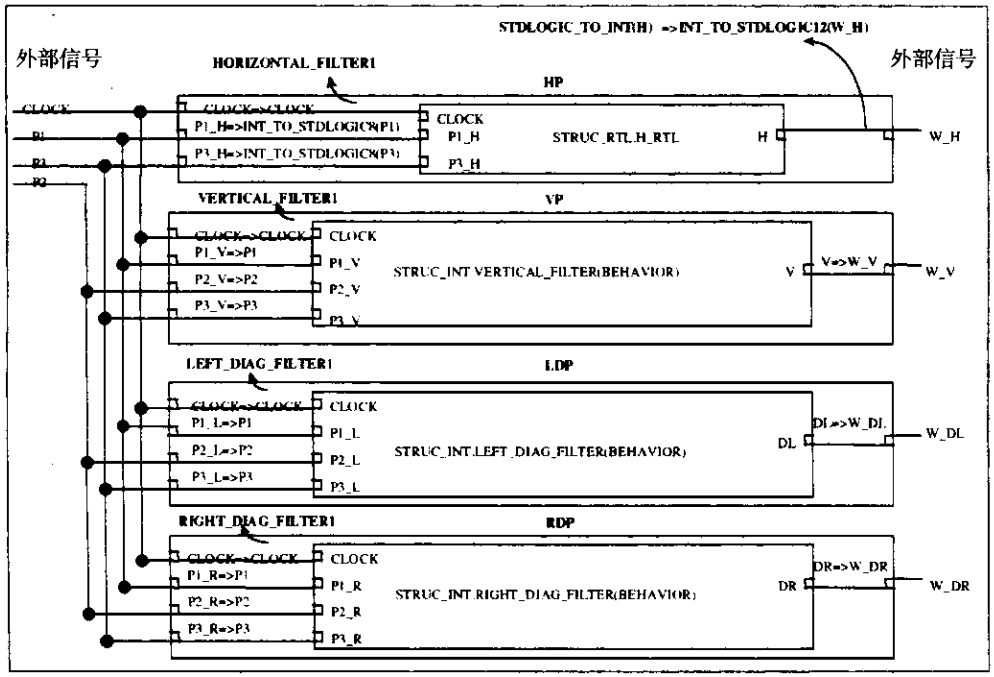


图11-55 窗口处理器配置后的框图

函数STDLOGIC_TO_INT是用户定义函数，它把类型STD_LOGIC的值转换为整数。函数INT_TO_STDLOGIC12是用户定义函数，它把整数转换为12位STD_LOGIC类型。使用两个转换是因为该端口是inout模式。发送到该端口后，数据的值是整数。如果在内部需要被读取，它必须把整数类型转换回内部12位的格式。

配置中也定义了插入到各插槽的芯片类型，例如配置STRUC_RTL里定义的水平过滤器的RTL模型。H_RTL插入到标志为HP的插槽。图11-55是窗口处理器完整的配置结构模型，它通过配置WINDOW_H_RTL_OTHERS_INTEGER定义。把这个模型与图11-53作比较，可以看到加入到结构描述的配置声明。插入到各插槽的特定芯片使用插槽符号的内部方框表示。方框包括即将插入到插槽的特定芯片的名称。考虑插入到水平过滤器插槽的芯片VP，芯片名称包括了实体名（VERTICAL_FILTER）、构架名（BEHAVIOR）及所有模型的库名（STRUC_INT）。

11.6.3 寄存器传输级测试包的产生

可执行定义及结构模型的测试包都可以在RTL模型里重用。为了达到这个目标，只需要修改窗口处理器中所有过滤器的构架名和库名，以及在STRUC_RTL里修改来自STRUC_INT中的幅度和方向处理器的配置声明。组件TEST配置声明里的类属常数WAIT_CYCLES也由5个时钟周期变为13个，这是因为在抽象结构级比较MUT需要额外的寄存器延时。这种测试方法类似于结构级模型里采用的方法。首先使用各自独立的测试包测试每个子部件。然后子部件集成为高一级模型。如果使用了配置的方法，初始时测试高一级模型的测试包可以重用来测试用RTL部件取代了原始行为模型的同一个模型。因为这里并没有引入新的理论，所以不再举出这里的测试包的实例。

11.7 门级详细设计

门级是抽象级的最低一级，它通常用VHDL表示。RTL级的每一个模型都被单独转化为一个门级模型。RTL结构模型可以通过为绑定到门级而不是RTL级的结构部件重写一个新的配置获得重用。

从RTL模型到门级模型的转换可以通过手工或使用综合工具来完成。使用VHDL的目的之一也是因为其自动综合电路的可能性。在RTL级，设计可以用电路表示，这些电路可以是加法器、减法器和寄存器。如果已经分析了包和RTL部件的VHDL代码，则结果通过使用综合工具进行读取。接下来，读取系统的RTL结构模型后综合产生门级电路。在第10章里已经详细描述了综合工具。

“设计分析器”是Synopsis系统里的一种综合工具。在下一节使用了这个工具为水平过滤器获得一个门级电路。

11.7.1 水平过滤器的门级设计

综合工具并不支持所有的VHDL构件。对于图11-49、图11-50和图11-51所示的水平过滤器的RTL模型，唯一不能被综合的构件是时间延时。下面的修改使这些模型可以被综合：

1) 把图11-49中的类属HORIZ_DELAY从实体声明中删除。

2) 把图11-50和图11-51中的类属HORIZ_DELAY从组件绑定中删除。

3) 把所有组件绑定到库STRUC_GATE里的实体，而不是库STRUC_RTL中的组件。

4) 库STRUC_GATE的所有组件都是对库STRUC_RTL中具有相同名称的组件的修改。在每个模型里，类型TIME的所有类属都被删除，并且所有*after*语句都从信号赋值语句中删除。

综合工具不支持信号延时，这一点并不令人感到奇怪。毕竟综合工具是对组件库的设计，它需要使用特定的方法。例如，一个CMOS门具有一个由CMOS技术决定的内部延时。这个延时不随设计者的期望而改变。因此，通常不会使用具有特定延时的实际的门来设计门级电路。出于这个理由，综合工具不支持信号赋值语句中的*after*语句。可参见第10章里对这个约束及其他综合约束的讨论。

图11-56使用综合工具为水平过滤器产生门级电路。在下一节讨论该过滤器对面积和时钟的优化。

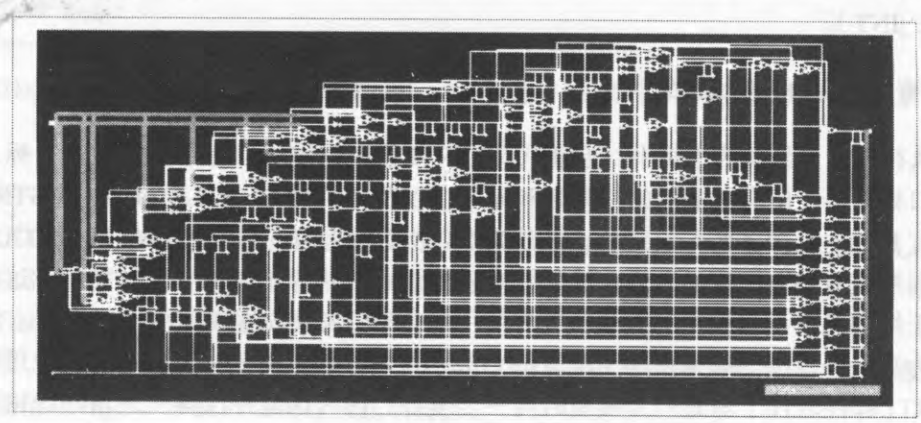


图11-56　由综合工具产生的水平过滤器的门级电路

11.7.2 门级电路的优化

综合RTL模型之后的下一步是对电路进行优化。一种典型的优化方法是首先优化面积然后优化时钟（参见第10章的其他优化策略）。大型设计层次块的优化通常开始于最底端的低层块。综合工具根据用户约束来完成优化。当前的综合工具一般允许最小面积和最小时钟延时的约束。

优化过程使用了不同的算法来寻找符合约束的最佳电路实现方案。一个电路基于优先程序的水平对其进行优化使之达到最小面积。它并不一定是手工仔细设计所产生的绝对最小电路。

约束表示期望的电路特征，它定义为设计目标的一部分。不同的约束导致产生不同的优化电路，它们在功能上相同。现在最常用的约束是面积和时间。

1) **面积**。面积约束对应特定设计模块期望的最大面积。面积数的单位对应于目标技术库中的单元，如等效的门、单元或晶体管。

2) **时间**。时间约束限制了从任何输入到输出的最大延时时间。综合工具里的静态时间分析程序抽取时间信息来计算实际路径延时。它包括寄存器元件的建立时间和保持时间及信号通过组合逻辑部件的延时。

计算得到的模型信号路径延时并与定义的时间限制进行比较；然后使用自动优先程序提高时间特性。一种策略是不断改进最大延时约束以获得一系列不同面积、延时和功耗的电路。设计者可以选择一个设计，在各个参数之间进行折衷。下一节里使用这种策略来优化水平过滤器的门级电路。

水平过滤器的优化

表11-4是对水平过滤器进行一系列综合步骤的结果。表中最上面一行表示没有任何约束的初始电路。第二行中最大面积定义为0，时钟周期定义为50ns，偏移为1ns。通过不断接近0面积，系统将获得这个时间延时上可能的最小面积。注意：它的面积与没有限制的情况相同，但极大地减少了功耗。

表11-4 水平过滤器的综合步骤

电路	最大面积定义	时钟周期定义(ns)	时钟偏移定义（ns）	产生的电路面积	实际电路松弛	实际电路功耗（uW）
1	未定义	未定义	未定义	706	未定义	276.24
2	0	50	1	706	29.02	13.86
3	0	31	1	712	10.53	21.87
4	0	21	1	709	0.81	31.94
5	0	18	1	712	0.3	38.07
6	0	14	1	740	0.05	52.44
7	0	11	1	751	0.02	66.69
8	0	9	1	823	0.01	98.01

然后从50ns~9ns不断减少时钟周期并重复进行优化。表11-4给出了一系列的优化结果。通常，减少时钟周期会得到越来越快的电路但需要更大的面积和功耗。设计者可以在电路速度和面积、功耗这三者之间权衡，选择表中被认为是合理的优化。松弛指的是最大电路延时和最大时钟周期之间的差别。

11.7.3 门级测试

把RTL级的MUT综合成网表，验证综合后网表的功能是否匹配RTL模型的功能非常重要。这一步之所以重要，是因为综合工具对VHDL代码的解释可能却不同于建模者的意图（参见第10章）。

不断降低层次时，模型逐渐被细化。一个主要的问题是门级模型的输出和较高级别模型的输出的比较。这里引入了反向标注（*back annotation*）的问题。在较高级别如何设计延时才可以使得输出与门级模型的输出发生在同一时间？反向标注可以定义为抽取时间信息和电路特征以及在高级别模型中插入这些信息的过程，从而使两个模型的输出时间匹配。

11.7.4 反向标注的方法

可以修改MUT行为模型的测试包模块来测试门级模型。测试包内加入一个比较过程来测试行为级模型和门级模型的输出，它产生一个PASS/FAIL结论。

获得对行为模型反向标注所需要的延时值的过程如下：

1) 系统级模型和门级模型被测试包实例化。

2) 加入了一个时钟产生器和比较过程。时钟产生器驱动行为级模型和门级模型。比较过程比较两个模型的输出并产生PASS/FAIL结论。

3) 为测试包产生一个配置，该配置最初没有为行为模型定义延时值。然后仿真执行测试包。通过研究仿真程序的输出来决定两个模型输出之间的相对延时。

4) 把相对延时反向标注到行为模型并再次进行仿真。如果电路稳定，则两个模型的输出应该一致。由于可能存在电路故障，两个输出可能在输入变化或时钟边沿时有短暂的不同。

反向标注后的行为模型可以用于精确延时的系统级仿真。

习题

11.1 为11.5.1节描述的窗口处理器构件，即图11-40给出的WIN _ PROCESSOR构件写出一个实体声明（名为WINDOW_PRAESSOR）和一个行为域构架（名为BEHAVIOR）。分析该模型，把分析后的模型存放在STRU_INT中。然后使用图11-42给出的测试程序和图11-43给出的配置对这个行为域构架进行测试。

11.2 为11.5.1节描述的存储器处理器构件，即在图11-40中作为MEM_PROCESSOR的构件建立一个行为域VHDL，模型使用如下设计过程：

a) 写出存储处理器的实体声明和行为域构架，分别命名为实体MEMORY_PROCESS和构架BEHAVIOR。分析这个实体和它的行为域构架，将分析后的模型存放到库STRUC_INT中。

b) 为a所设计的存储处理器构件编写测试程序。不对构件和测试程序进行绑定。使用图11-42的格式分析这个测试程序，并把分析后的模型存放到库STRUC_INT中。

c) 写出把存储处理器部件绑定到行为域构架的配置说明。使用图11-44的格式，分析该配置并把分析后的模型存放到STRUC_INT库中。

d) 对c中的配置进行仿真，验证存储处理器的行为域模型。

11.3 对11.5.1节描述的幅度处理器组件，即图11-40的MAG_PROCESSOR构件重复习题11.2的过程，名称分别为实体MAG_PROCESS和行为域构架BEHAVIOR。

11.4 为11.5.2节描述的水平过滤器，即图11-40中给出的部件HOR_FILTER写出一个实体声明和一个行为域构架，使用图11-41给出的独立测试程序对模型进行测试。

11.5 为11.5.2节描述的垂直过滤器，即图11-40给出的部件VERT_FILTER写出一个实体声明和一个行为域构架，写出一个类似图11-41的测试程序来测试垂直过滤器行为域模型。使用一个配置来绑定构件的测试程序和它的行为域构架，通过仿真测试程序配置来验证这个行为域模型。

11.6 为11.5.2节描述的左对角过滤器，即图11-40给出的部件LEFT_FILTER写出一个实体声明和一个行为域构架，写出一个类似图11-41的测试程序来测试左对角过滤器的行为域模型。使用一个配置来绑定构件的测试程序和它的行为域构架，通过仿真测试程序配置来验证这个行为域模型。

11.7 为11.5.2节描述的右对角过滤器，即图11-40给出的部件RIGHT_FILTER写出一个实体声明和一个行为域构架，写出一个类似图11-41的测试程序来测试右对角过滤器的行为域模型。使用一个配置来绑定构件的测试程序和它的行为域构架，通过仿真测试程序配置来验证这个行为域模型。

11.8 为11.5.2节描述的地址产生器，即图11-40给出的部件ADDR_GEN写出一个实体声明和一个行为域构架，写出一个类似图11-41的测试程序来测试地址产生器的行为域模型。使用一个配置来绑定构件的测试程序和它的行为域构架，通过仿真测试程序配置来验证这个行为域模型。

11.9 为11.5.2节描述的存储器，即图11-40给出的部件MEMORY写出一个实体声明和一个行为域构架，写出一个类似图11-41的测试程序来测试存储器的行为域模型。使用一个配置来绑定构件的测试程序和它的行为域构架，通过仿真测试程序配置来验证这个行为域模型。

11.10 为11.5.1节描述的窗口处理器，即在图11-40给出的WINDOW_PROCESSOR构件产生一个结构域VHDL模型，使用如下设计过程。

a) 写出窗口处理器的实体声明和结构化构架，此时没有定义四个过滤器组件的绑定。使用图11-36和图11-37的格式，分析这个实体和它的结构化构架，并把分析结果存放在库STRUC _ INT中。

b) 写出把四个过滤器组件绑定到题11.4、11.5、11.6和11.7时产生的行为域模型的配置，分析这个配置并将分析后的模型存放在库STRUC _ INT中。

c) 重用图11-42的测试程序来验证窗口处理器的结构化模型，如果没有预先完成这些，则分析图11-42的测试程序模型并把分析后的模型存放在库STRUC_INT中。使用图11-44给出的格式把窗口处理器的结构化模型绑定到图11-42测试程序中的构件WINDOW _ PROCESSOR1，仿真该配置以验证窗口处理器的结构模型。注意：该配置使用了 (b) 步写出的配置把窗口处理器构件WINDOWS_PROCESSDR1绑定到在 (a) 中创建的结构化模型。

11.11 为11.5.1节描述的存储处理器，即在图11-40给出的MEM _ PROCESSOR构件产生一个结构域VHDL模型，使用如下设计过程。

a) 写出存储处理器的实体声明和结构化构架，此时没有定义两个构件的绑定。使用图11-36和图11-37的格式，分析这个实体和它的结构化构架，并把分析结果存放在库STRUC_INT中。

b) 写出把两个组件绑定到习题11.8和习题11.9所产生的行为域模型的配置，分析这个配置并将分析后的模型存放在库STRUC_INT中。

c) 重用习题11.2(b)的测试程序来验证存储处理器的结构化模型，如果没有预先完成这些，则分析图11-42的测试程序模型并把分析后的模型存放在库STRUC_INT中。使用图11-44给出的格式把存储处理器的结构化模型绑定到测试程序中的存储器构件，仿真该配置以验证存储处理器的结构模型。注意：该配置使用了 (b) 步写出的配置把存储处理器构件绑定到在 (a) 中创建的结构化模型。

11.12 用11.6.2节的格式为图11-36和图11-37所示的Sobel边缘检测器重写结构化模型代码，在下列情形下用图11-50的格式写出配置：

a) 所有构件都是整型数据类型。

b) 所有构件都是STD_LOGIC数据类型。

c) 两个构件是整型数据类型，一个是STD_LOGIC数据类型。

d) 两个构件是STD_LOGIC类型，一个是整型数据类型。

11.13 用11.6.2节的格式为题11.10产生的窗口处理器结构化模型重写代码，在下列情形下用图11-50的格式写出配置：

a) 所有构件都是整型数据类型。

b) 所有构件都是STD_LOGIC数据类型。

c) 三个构件是整型数据类型，一个是STD_LOGIC数据类型。

d) 三个构件是STD_LOGIC数据类型，一个是整型数据类型。

e) 两个构件是STD_LOGIC数据类型，一个是整型数据类型。

11.14 用11.6.2节的格式为题11.11产生的存储处理器结构化模型重写代码，在下列情形下用图11-50的格式写出配置：

a) 所有构件都是整型数据类型。

b) 所有构件都是STD_LOGIC数据类型。

c) 一个构件是整型数据类型，一个是STD_LOGIC数据类型。

11.15 写出在下面情况下类似于图11-54的配置。为了阐明每个配置，画出类似于图11-53和图11-55的框图：

a) 水平和垂直过滤器都是具有STD_LOGIC数据类型的RTL模型，但左和右对角过滤器都是使用整数数据类型的行为域模型。

b) 水平、垂直及左对角过滤器都是具有STD _ LOGIC数据类型的RTL模型，但左和右对角过滤器都是使用整数数据类型的行为域模型。

c) 所有过滤器都是具有STD_LOGIC数据类型的RTL模型。

11.16 说明如何重用独立的水平过滤器测试程序（图11-41）来测试该过滤器的RTL模型（图11-49、图11-50和图11-51）。

11.17 为图11-40的垂直过滤器VERT_FIITER产生一个RTL模型，使用11.6.1节的方法。说明如何重用习题11.5产生的测试程序来测试这个RTL模型。

11.18 为图11-40的左对角过滤器LEFT_FILTER产生一个RTL模型，使用11.6.1节的方法。说明如何重用习题11.6产生的测试程序来测试这个RTL模型。

11.19 为图11-40的右对角过滤器RIGHT_FILTER产生一个RTL模型，使用11.6.1节的方法。

说明如何重用习题11.7产生的测试程序来测试这个RTL模型。

11.20 为图11-40的地址产生器构件ADDR_GEN产生一个RTL模型，使用11.6.1节的方法。说明如何重用习题11.8产生的测试程序来测试这个RTL模型。

11.21 为图11-40的存储器构件MEMORY产生一个RTL模型，使用11.6.1节的方法。说明如何重用习题11.9产生的测试程序来测试这个RTL模型。

11.22 使用类似表11-4的方法，综合题11.17产生的垂直过滤器的RTL模型。

11.23 使用类似表11-4的方法，综合题11.18产生的左对角过滤器的RTL模型。

11.24 使用类似表11-4的方法，综合题11.19产生的右对角过滤器的RTL模型。

11.25 使用类似表11-4的方法，综合题11.20产生的地址产生器的RTL模型。

11.26 使用类似表11-4的方法，综合题11.21产生的存储器的RTL模型。

11.27 使用11.7.4节描述的方法为下面每个部件的行为域模型加入精确延时注释：

a) 水平过滤器

b) 垂直过滤器

c) 左对角过滤器

d) 右对角过滤器

e) 地址产生器

f) 存储器构件

11.28 项目：图11-40中的幅度处理器MAG_PROCESSOR是否能被逻辑分解？如果可以，利用本章讨论的配置方法进行分解，以指定测试构件及集成系统的测试程序。

11.29 项目：使用本章的方法设计题10.22描述的计算器。

11.30 项目：使用本章的方法来设计题10.23描述的Booth乘法器。

11.31 写出图11-45中使用的CONFIG_SOBEL_S_L2的配置，该配置使用了存储处理器、幅度处理器行为模型及窗口处理器的结构模型。

11.32 写出使用存储处理器和窗口处理器结构化模型及幅度处理器行为模型的配置CONFIG_SOBEL_S_L3。为了适应新的配置，重写图11-45所示的测试程序配置。

11.33 写出使用存储处理器的结构化模型及窗口处理器和幅度处理器行为模型的配置CONFIG_SOBEL_S_L4。为了适应新的配置，重写图11-45所示的测试程序配置。

第12章　设计自动化的综合算法

本章中的算法综合描述了从算法级的抽象行为规格说明到寄存器级（或门级）规格说明的自动转换。目标规格说明可以是行为模型或结构模型。虽然有一种中间的行为有限状态机规格说明可以减少综合中某些步骤的复杂度，但是当前的实验系统都倾向于把目标规格说明用结构模型表示。如果不考虑目标规格说明，综合步骤都是比较类似的。

有多种方法可以把算法级的行为描述转换到寄存器或门级描述。实际上可能的转换结果的数量多到难以处理。综合系统必须从这一巨大的设计空间里选择出最好的设计。穷举搜索一般不能算是一种行得通的方法。很多情况都使用了流行的部件，设计空间受到可用的部件类型的限制。对于用普通方法设计的集成电路（通常指的是ASIC），设计受到面积或功耗的限制。规格说明可能包括最大延时时间、最大时钟频率或其他的时序约束。综合系统必须从巨大的设计空间中找到满足规格说明和技术的全部限制的具有最低开销的结构。这项工作非常困难。

12.1　算法性综合的优点

既然问题很困难，在这上面花费的努力值得吗？解决这个问题所带来的好处可以通过算法综合的能力推导出来。

1) **更短的设计周期**。通过自动完成高层的综合任务，设计过程可以以更适时的方式完成。这有利于公司在高度竞争的市场取得成功。

2) **更低的设计开销**。一个更短的设计周期可以降低设计开销。这对于主要开销因素是设计开销的小批量生产项目非常重要。

3) **更低的产品开销**。因为在算法级的决策对最终系统的复杂程度（从而对系统开销）比在更低级别做出的决策所产生的作用要大得多，通过高层的优化决策可能大大降低系统开销。当今大多数的设计之所以能够使用这种优化，完全是来自于设计小组的设计经验。它需要对低级别技术细节比较了解。如果系统设计者使用高级别综合工具，他们可以利用依赖于技术的权衡策略，而不是更多地学习该技术本身的详细知识。所产生的结果也应该是更好的设计。

4) **更少的设计错误**。自动化设计过程中减少了人为的错误。对于非常复杂的设计，人为错误的产生很难避免。对于手工设计的系统，调试完综合程序之后（这也是一项非常困难的任务），还需要确定所发现的错误没有蔓延到更大的范围。

5) **更易于确定设计的折衷方案**。自动综合工具可以快速经济地产生符合行为规格说明的几种设计。使用传统的设计方法，产生不同的设计备选方案需要很高的开销，这阻碍了对设计方案在较大范围的折衷。自动设计使得设计者可以更有效地对折衷方案进行评估。

6) **更易于维护文档标准**。自动化系统可以产生需要的文档，文档包括设计决策记录及做出这些决策的理由。

7) **规格说明改变时更易于调整**。在传统的设计过程中，当规格说明发生变化时，改变实现的开销非常巨大。使用高级别综合工具，当模型的功能发生变化时模型的行为也可以做出

相应的调整。然后，利用综合工具可以重新设计系统。完成这项任务所需的开销和时间可以被极大地减小。

12.2 算法性综合的任务

图12-1表明了综合的主要任务。本书从对算法级VHDL行为描述的高级综合过程开始。第一个任务通常被称为编译（*compilation*），它把VHDL描述转换为一个适合自动综合的中间格式。本书将使用数据流图（DFG）作为中间格式，同样也可以使用其他的抽象表示。为了说明综合步骤，我们使用了图12-2所示的简单VHDL描述。注意高级别所使用的数据类型通常是整数类型或其他的算术类型。

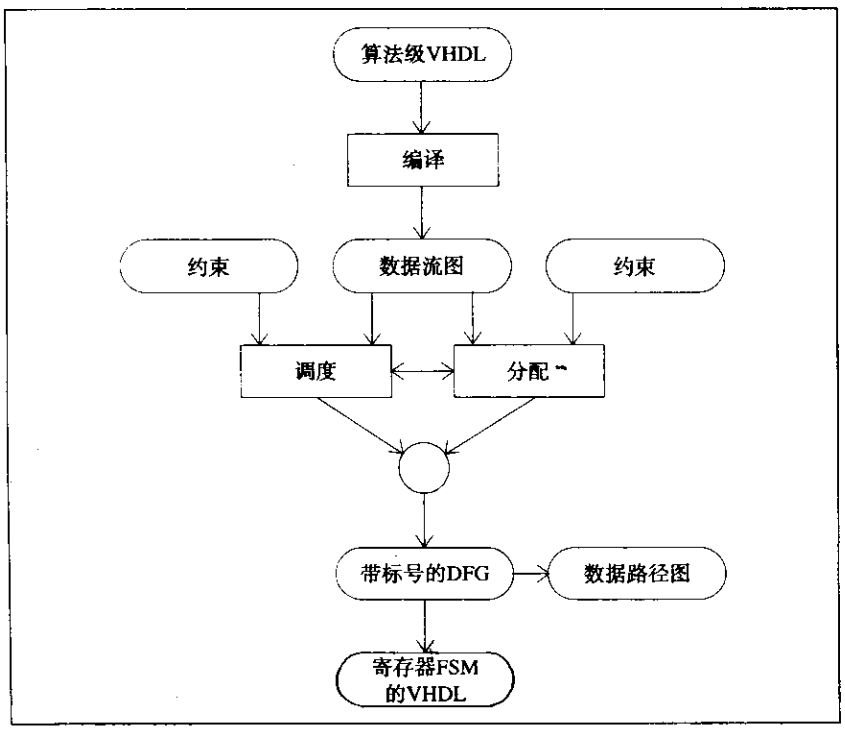


图12-1 算法级综合任务

```
-------------------------------------------------------
-- The entity declaration of synthesis example model.
-------------------------------------------------------
entity SYNEX1 is
  port (A, B, C, D, E: in INTEGER;
        X, Y: out INTEGER);
end SYNEX1;
-------------------------------------------------------
-- The architecture declaration of synthesis example model.
-------------------------------------------------------
architecture HIGH_LEVEL of SYNEX1 is
begin
  X <= E*(A+B+C);
  Y <= (A+C)*(C+D);
end HIGH_LEVEL;
```

图12-2 算法级VHDL描述示例

12.2.1 VHDL描述到内部格式的编译

首要任务是把VHDL变为易于内部处理的格式。这种编译包括转换（*transformation*）和/或优化（*optimization*），优化可以帮助得到低开销的方案。这些优化包括基于软件的行为，如死代码消除、公共子表达式标识、过程的直接插入、循环展开等。转换是面向硬件的，如整数代换二进制向量、用左移1位代替乘2等。通常它使用数据流图作为中间形式。图12-3是图12-2所示例子的数据流图。DFG中的节点表示VHDL中的操作。DFG的弧表示操作之间的先后关系。例如，从+到*之间的弧表示加法操作必须在乘法操作之前完成。

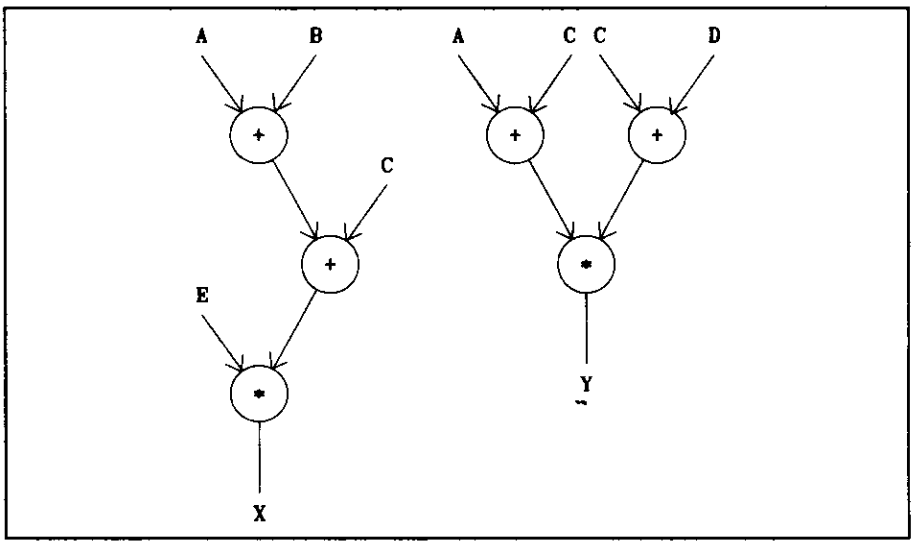


图12-3　数据流图示例

12.2.2 调度

调度包括为每个操作赋予一个控制步骤。控制步骤是同步系统中最基本的时间单元，它对应一个时钟周期。调度的主要目标是使得器件完成所有功能所需要的时间最少。对于计算机系统来说，调度使得原始VHDL程序所描述的算法执行的时钟周期达到最少。这种调度也可以作为对目标技术或可用功能部件的一种约束。图12-4是直线调度的一个例子，假设没有其他约束，器件只使用了三个控制步骤。这种调度使用了ASAP方法，每一个操作被调度发生在可能的最早时刻。尽管这种方法在没有调度约束时可以达到最大吞吐量，但是一般它都需要过多的硬件。例如，这种ASAP调度需要三个独立的加法器在控制步骤s1完成三个并行的加法操作。

12.2.3 分配

分配（*allocation*）是指定义系统中部件和部件之间互连的过程。可以指定几种特定的活动，虽然这些活动并不总是相互独立的。

- 分配寄存器或RAM存储器来存放数据值。
- 分配功能部件来执行特定的操作。

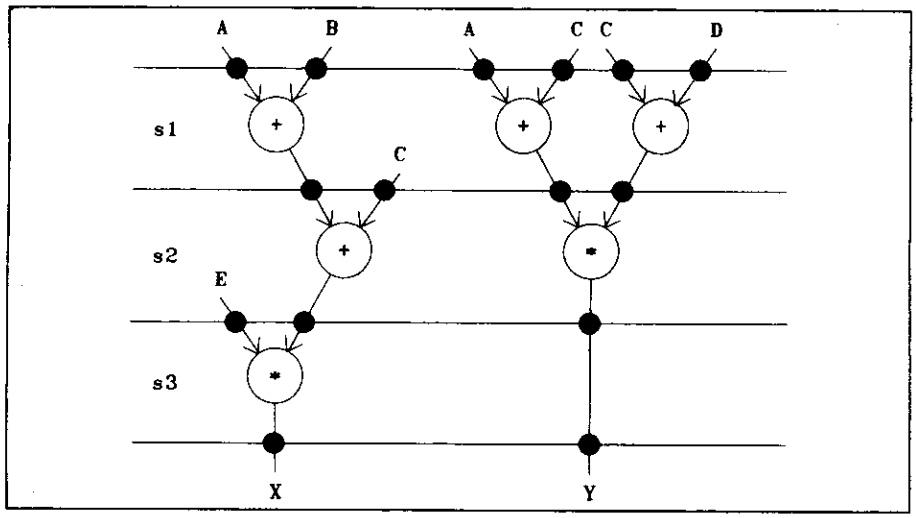


图12-4 示例数据流图的ASAP调度

• 分配互连路径以在部件之间传输数据。

1. 分配寄存器来存储数据值

图12-5表示寄存器和功能部件的一种可能分配，以实现图12-4所示的调度。在分配过程中，需要一些有关输入输出的额外知识。对于这种分配，我们假设所有的输入在算法开始执行前都被锁住。状态s0中的加完成了锁存操作。

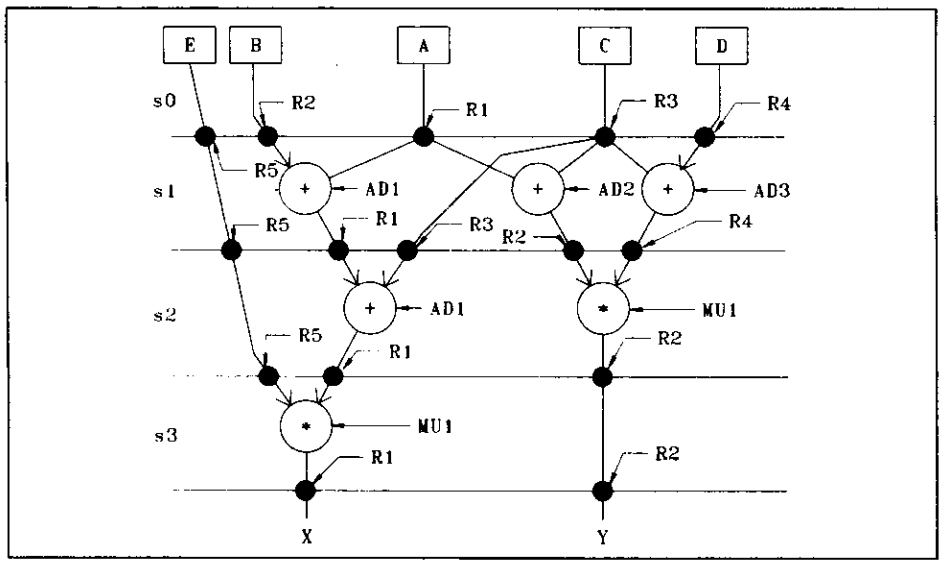


图12-5 示例数据流图ASAP调度的寄存器和功能部件分配

图12-5中的深色圆圈表示需要的数据存储。例如，在步骤s0和s1之间，器件必须存储5个对应于数据输入的数据值。输入A、B、C、D和E分别赋值到寄存器R1、R2、R3、R4和R5。在步骤s1和s2之间，同样也有5个数据值需要被存储。这里重用了同样的5个寄存器。决定使用哪些寄存器存放这些数据将会影响互连的复杂程度，从而决定几个重要的系统参数，如开

销、信号延时、芯片面积及扇入和扇出装载，进而极大地影响系统开销。寄存器赋值的优化比较复杂。

2. 分配功能部件来执行特定操作

分配可以限制在只针对给定的部件库中的元件。这种形式被称为自底向上分配，这是因为该过程由库中部件的可用性所驱动。如果设计者可以自由实现任何期望的部件类型，如与ASIC设计或自顶向下设计里一样，则可以使用无约束形式的分配。显然在同一个设计小组可以为了不同的目标使用几种不同的分配。本设计假设部件库里包括加法器、乘法器、寄存器及多路器（MUX）。我们将使用自底向上的方法。

在步骤s1，三个独立的加法器并行执行。因此，需要三个独立的加法器，分别用AD1、AD2和AD3表示。在步骤s2，只需要执行一个加法操作。它被分派到AD1。由于在每个步骤里最多只有一个乘法操作，所以只需要一个乘法器就足够了。任何乘法操作都被分派到同一个部件（MU1）。

3. 为部件间的互连分配数据路径

图12-6是示例系统中数据路径的一种可能的分配方案。表12-1是图12-6数据路径指派所需要的部件。

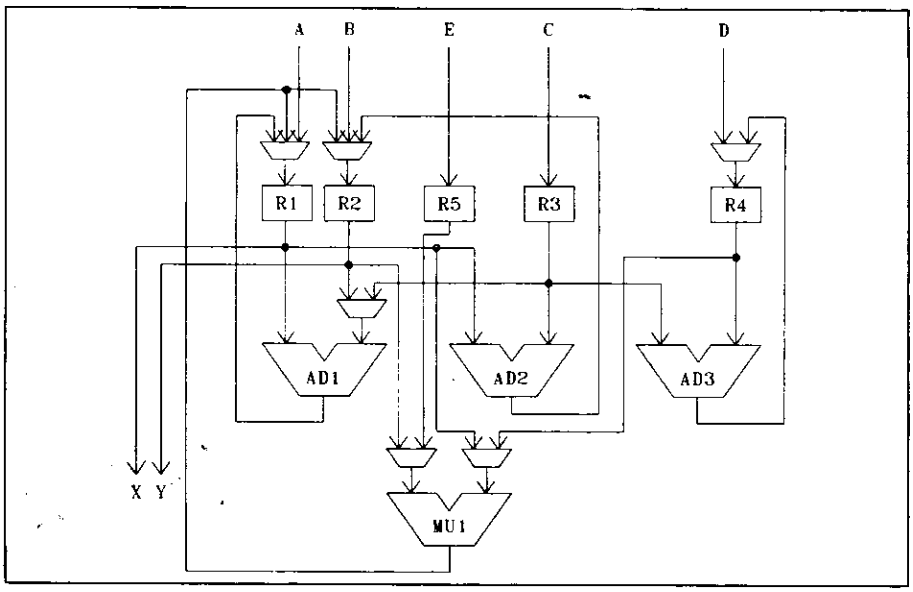


图12-6　示例数据流图ASAP调度的数据路径

表12-1　图12-6数据路径指派的部件

部　件	数　量
加法器	3
乘法器	1
寄存器	5
2×1 MUX	4
3×1 MUX	2

12.2.4 调度和分配的交互

很明显，调度和分配并不是相互独立的操作。考虑如下结果：

- 在同一控制步骤中的两个操作必须使用不同的功能单元。这里隐含着分配过程。
- 当操作执行需要多于一个时钟周期时，为了获得优化调度，必须知道每一步的操作需要多长时间。这里需要了解执行部件的速度，这一信息也来自分配过程。
- 为了减少部件开销，需要把部件的数量减小到最少。前面已经提到，部件的数量受到调度的影响。

因此，为了优化部件操作的分配，必须了解调度。而且，为了优化调度，必须了解功能器件的分配。这两个过程的交互体现了系统优化目标实现的困难。

一种方案是任意分配，或者是系统对分配过程加以限制，并且基于这些限制获得一种优化调度。另一种方法是循环选择不同的分配限制，在每次循环里进行调度。通常使用启发式选择下一循环，它试图使用一种得到整体优化设计的算法。另一种系统设计则同步进行调度和分配，这种系统通常基于特定的设计目标来得到最终设计，例如，最大速度或最小面积。

假设设计工程师决定仅仅使用一个加法器和一个乘法器处理图12-3所示的数据流图。图12-7中的调度就是相应的结果。这里由于硬件的约束造成了一个额外的控制步骤。执行时间由四个时钟周期增加到五个。同时，给出的调度不能直接从原始规格说明里得到。它必须通过使用一些代数规则对原始规格说明进行转换。被转换的表达式如下：

$$X<=E*[(A+C)+B\uparrow]$$

$$Y<=(A+C)*(C+D)$$

这里的公用表达式(A+C)是已知的。

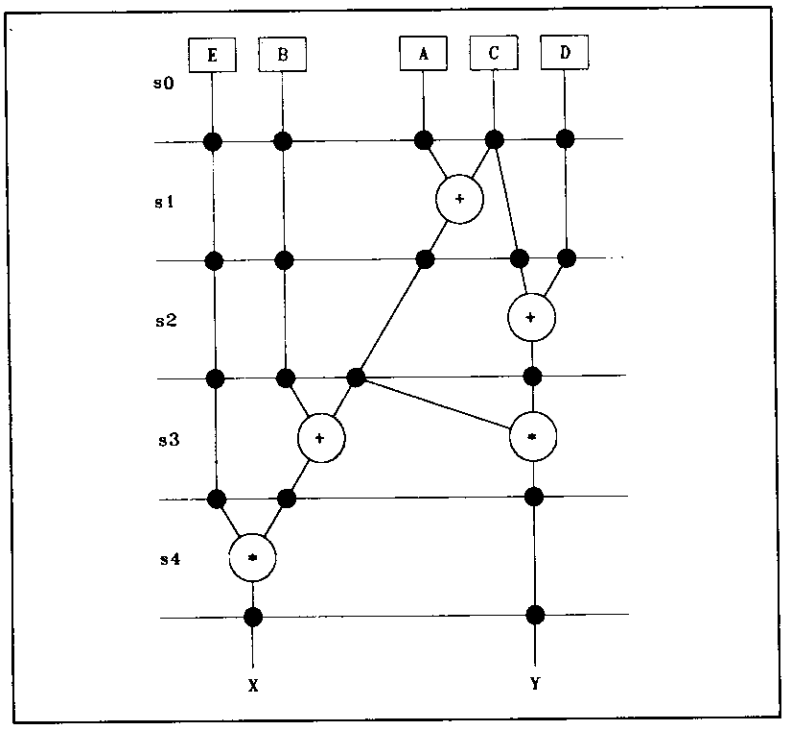


图12-7 带硬件约束的示例数据流图的调度

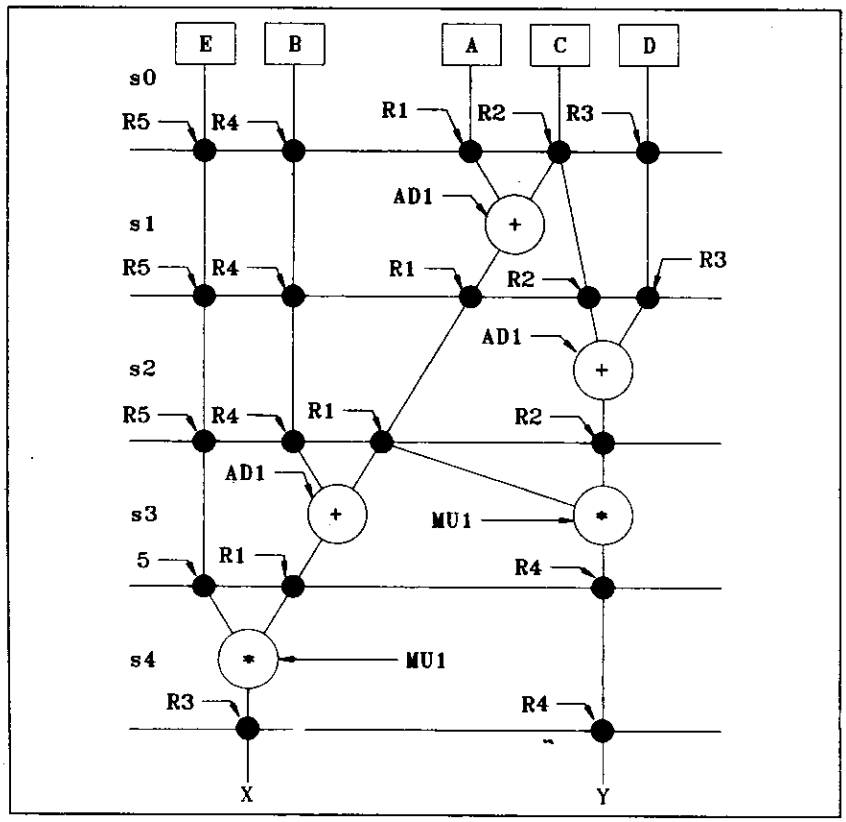


图12-8 约束调度的分配

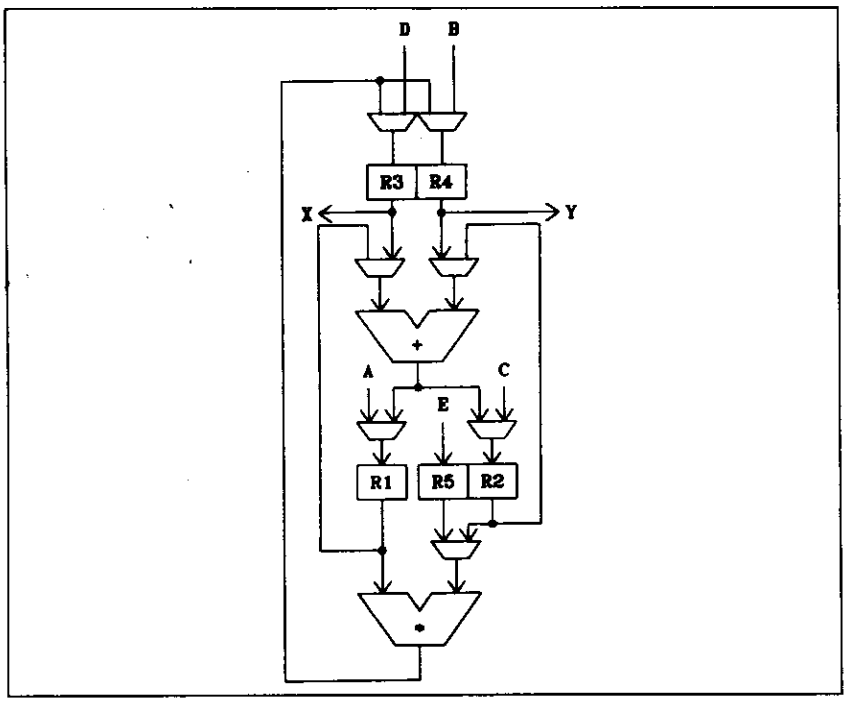


图12-9 约束调度的数据路径

图12-8是约束调度分配寄存器和功能单元的一种方案，图12-9是可能的数据路径分配。表12-2表明了所需的部件，与表12-1所示的原始部件的数量相比，它减少了两个加法器，多路器的尺寸由3×1减少到2×1，而且增加了一个额外的2×1的多路器。这种方案的实现比原始设计所需要的面积要小。这里并没有考虑互连的开销。

表12-2 约束调度所需的部件

部 件	数 量
加法器	1
乘法器	1
寄存器	5
2×1 MUX	7

12.2.5 Gantt图和利用率

在详细描述调度方法之前，首先要讨论一些基本的定义。在操作指派到功能部件之后，Gantt图是评价结果的一种非常有用的工具。图12-10是图12-8所示调度的Gantt图。在这幅图里，符号“b”表示一个部件忙，符号“--”表示该部件闲。Gantt图中一个单独行是系统中的一个部件，一个单独列是一个时间步骤或控制步骤。这个Gantt图表示加法器AD1在步骤s1、s2和s3都被使用，在步骤s0和s4时空闲。相应地，乘法器M1在步骤s3和s4忙，而在其他步骤空闲。

	s0	s1	s2	s3	s4
AD1	--	b	b	b	--
MU1	--	--	--	b	b

图12-10 约束调度的Gantt图

执行时间是进程执行所需的总时间。例如，执行时间为5，这是因为需要五个时间步骤。调度和分配的一个基本目标是使得进程的执行时间最少。当使用了设计方法时，最少执行时间可以使结果硬件的延时最小，从而使得系统的执行速度最快。

部件的利用率是部件忙的时间与系统执行时间的比率。例如，加法器的利用率是0.6（或60%），乘法器的利用率是0.4（或40%）。它有时趋近最大利用率，这是因为从某种意义上说，最大利用率表示对资源的最佳利用。如果利用率很低，可能较少部件的数量。但是，通常减少部件数量的结果会增加执行时间，结果是用速度换取时间。如果部件的利用率过低，可能意味着一种无效的设计。在其他的情况下，包括在许多VLSI设计里，并不在乎低的利用率。在本例中，相对比较低的利用率是进程固有的属性，因此改进利用率并不是一个目标。

12.2.6 从分配图创建FSM VHDL

FSM VHDL模型可以直接由分配图构造。图12-11是图12-8约束调度的FSM VHDL模型。它通过简单声明所需要的寄存器来声明一系列控制步骤（在VHDL程序里称为状态）并为每

个状态插入所需的数据传输，以一种直接的方式构建了模型。这里使用了在第8章里描述过的Moore状态机。时钟输入（CLK）为状态转换提供了定时机制。在算法级模型里并没有定义S4之后的状态，这里假设紧跟S4之后的状态是S0。由于输出与状态无关，Moore状态机的输出线路中并不需要使用case语句。这从某种程度上简化了模板。

```
-------------------------------------------------------
entity FSMEX1 is
  port (A, B, C, D, E: in INTEGER;
        CLK: in BIT;
        X, Y: out INTEGER);
end FSMEX1;
-------------------------------------------------------
architecture FSM of FSMEX1 is
  type STATE_TYPE is (S0, S1, S2, S3, S4);
  signal STATE: STATE_TYPE;
  signal R1, R2, R3, R4, R5: INTEGER;
begin
-- Process to update state and perform register transfers.
STATEP: process (CLK)
  begin
    if CLK'event and CLK='1' then
      case STATE is
        when S0 =>
          -- Data Section
          R5 <= E; R4 <= B; R3 <= D; R2 <= C; R1 <= A;
          -- Control Section
          STATE <= S1;
        when S1 =>
          -- Data Section
          R1 <= R1 + R2;
          -- Control Section
          STATE <= S2;
        when S2 =>
          -- Data Section
          R2 <= R2 + R3;
          -- Control Section
          STATE <= S3;
        when S3 =>
          -- Data Section
          R1 <= R4 + R1;
          R4 <= R1 * R2;
          -- Control Section
          STATE <= S4;
        when S4 =>
          -- Data Section
          R3 <= R5 * R1;
          -- Control Section
          STATE <= S0;
      end case;
    end if;
  end process STATEP;
  -- Assign output.
  X <= R3; Y <= R4;
end FSM;
```

图12-11 FSM VHDL模型示例

为了实现这一器件，必须在输入增加一条开始信号，并在输出增加一条完成信号（参见习题12.1）。图12-11所示的模型可以用来仿真验证电路的正确操作。在验证完成之后，可以开始设计硬件。用整数寄存器来映射位向量，并且设计一种硬连线的控制单元或一种微程序控制单元，它们都在第8章中介绍过。最后，要设计输入和输出都是位向量的加法和乘法功能部件。如果它们都是ASIC单元，则可以使用第11章介绍的方法（自顶向下设计），或者直接从现有的库中选择使用功能单元（自顶向上设计）。

这一节所使用的方法结合了第5章、第8章和第11章介绍的方法。现在使用在VHDL设计系统中的方法，从文字描述开始到寄存器级结构表示为止，其中包括了控制部件的设计。图12-12总结了这一方法。它使用在第5章讲述的方法，从给出的文字描述产生一个算法级VHDL模型。然后，使用第12章讲述的方法把算法级VHDL模型变换为FSM VHDL模型。第8章讲述的方法被用于设计FSM的控制部件。第12章讲述的方法被用于定义数据路径图。最后，通过使用第9、10和11章的设计方法来产生各种独立的功能单元，或者从库中直接引用各单元。

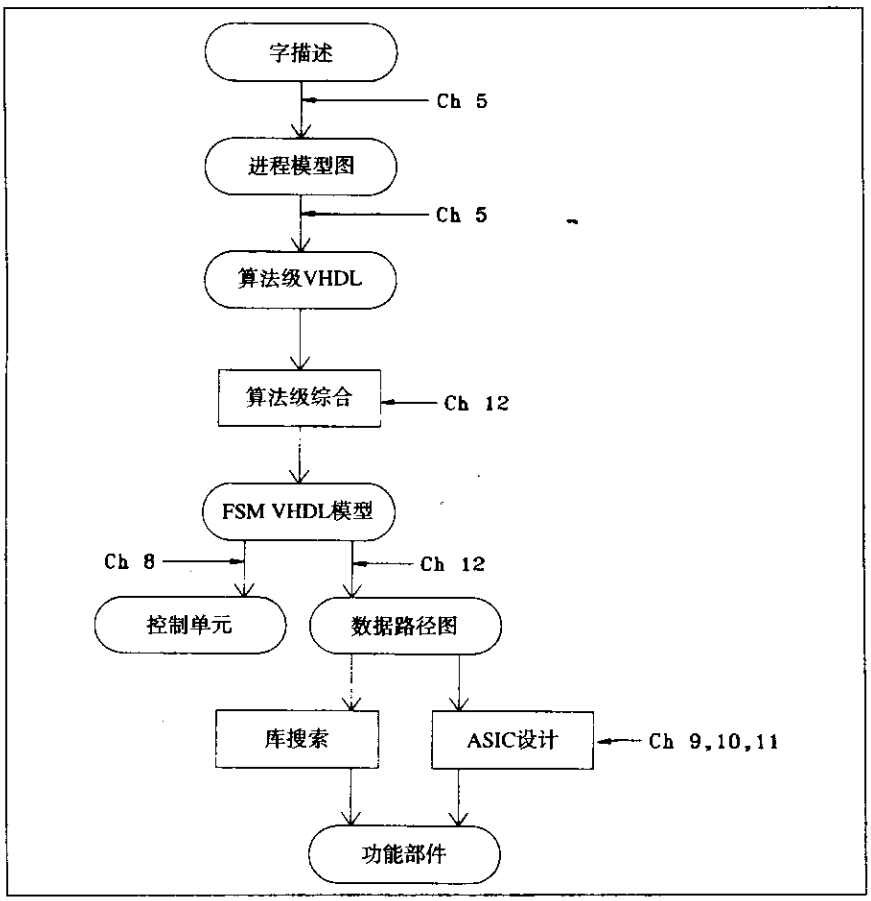


图12-12 使用VHDL的设计方法

12.3 调度方法

这一节包括了对调度方法的描述。调度时有两种方法，第一种是转换调度，它从一个缺

省调度开始，通过一个调度序列不断进行调整直到调度满足存在的约束。第二种方法是迭替代/构造调度，其中最终调度的产生是通过不断为每个时间步骤增加一个操作，直到所有操作均被调度。

12.3.1 转换调度

转换调度通常开始于一个容易构造的调度。例如，图12-4所示的ASAP就是一个最大并行调度的例子。设计者可以从该调度方法开始，不断对其进行修改直到满足所有的约束条件。

有一种被称作状态划分的方法，它把遇到硬件约束或时钟约束的状态划分为一个或多个子状态。例如，如果指明限制只使用一个加法器和一个乘法器，则状态S1将被划分为三个子状态来满足单个加法器的约束。划分的方法有多种，而且还没有一种优化划分的有效过程。图12-13是一种可能的划分方法，最后的调度需要六个控制步骤。

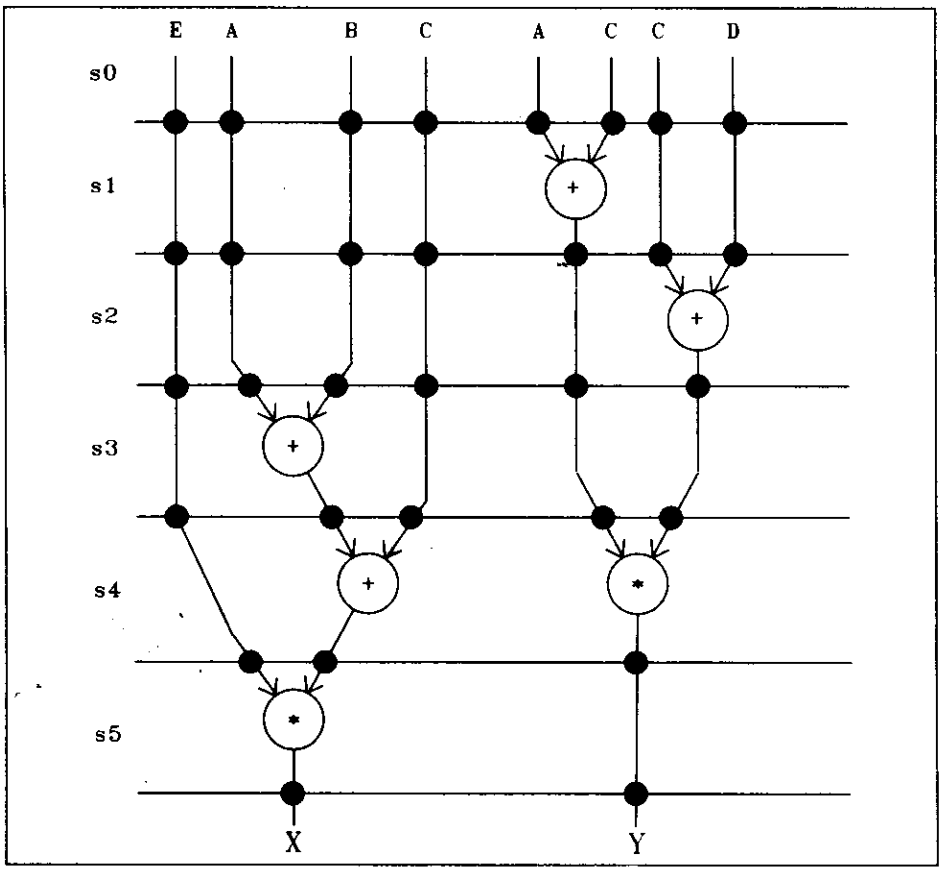


图12-13 状态划分示例

另一种转换调度的方法被称为穷举搜索。它对初始调度列举出所有可能的转换，结果保证一定是最优调度，但除非是最简单的情况，否则它的计算量难以忍受。这里可以采用分支方法和限界方法来消除向下搜索中的无效路径。然而，更普遍的是采用启发式方法直接向下搜索那些包括最优方案的路径。尽管并不能保证其结果是最优的，但是用可控制的计算量，通过几次反复就可以得到比较“好”的设计。这种方法超出了本文介绍的范围。

12.3.2 迭代/构造调度

在迭代/构造调度里，通过在每个时间步骤上不断增加操作直到所有的操作均被调度为止，来生成一个调度序列。每一步需要做出两个决定：

1) 下一个被调度的操作是哪一个？

2) 该操作步骤应属于哪一个控制步骤？

每一步做出的决定由局部或全局标准控制。如果使用了局部标准，可以减少选择的数量，但是会牺牲全局的优化。全局标准通常要检查更多的可能选择，它比局部标准提供了更好的结果。这两者需要在设计质量和计算复杂度之间进行折衷。

这一类方法中各种方法的不同之处在于对下一调度操作的选择和对操作执行步骤的选择。下面讨论几种具有代表性的循环/构造调度方法。

12.3.3 ASAP调度

ASAP调度大概是最简单的一种调度算法。假设功能单元的数量已经确定，而且已经获得了数据流图，ASAP调度方法在每一时钟步骤中尽可能多地指派操作——只要可以使用数据和功能部件，即每个操作都尽量早地被调度。

由于没有预先检查后续控制步骤中潜在的调度冲突，因此选择出的下一个被调度的操作只依据局部标准。因为每个控制步骤的调度是顺序的，它从第一个控制步骤开始，因此对一个给定操作所选择的控制步骤也是局部的。它为每个控制步骤指派尽可能多的操作，所有的操作只要数据可用就可以在当前的控制步骤里被调度。在为当前控制步骤指派了尽量多的操作之后，它将转移到下一个控制步骤，而且不会再返回到当前的控制步骤。

例如，考虑图12-3的数据流图。图12-14对该数据流图的节点随意编号，这些编号只在访问时被使用。假设最多定义了两个加法器和两个乘法器。在控制步骤s1中，可供调度的操作是1、2和3。由于它们都是加法操作，而这里只有两个加法器可以使用，因此任意选择其中的两个进行调度。在为调度选择操作时仅使用了局部标准，我们并不能预测出将来调度中出现的问题。因此，在控制步骤s1里随机选择的是操作2和操作3。显然，选择时可能有3种方法。穷举搜索法将会尝试这三种可能的方法。

由于在控制步骤s1里指派了操作2和操作3，它们产生的结果在控制步骤s2有效。所以，在控制步s2可供调度的操作是1和5。因为这两个操作种类不同，它们可以同时被调度。控制步骤s3中唯一可以调度的是操作4，所以在这一步对此操作进行调度。操作5在控制步骤s4中调度。

图12-15是具有两个加法器和两个乘法器约束的一种ASAP调度。结果表明它只使用了一个乘法器，因而可能存在一种更加有效的调度。但是，一般来说一个有效的调度并不一定需要使用所有的部件。

调度仅仅对每个操作指派一个时钟周期。必须还要为操作分配一个特定的硬件部件。通常使用的是一种贪心分配法来分配功能部件（详见12.4.1节“贪心分配法”）。图12-16是使用ASAP调度获得的Gantt图。贪心分配法的基本原则是为每个操作指派第一个可用的功能单元部件来执行该操作。

可以看出这种方法中功能部件的利用率比较低，这也表明可能存在一种更好的调度。

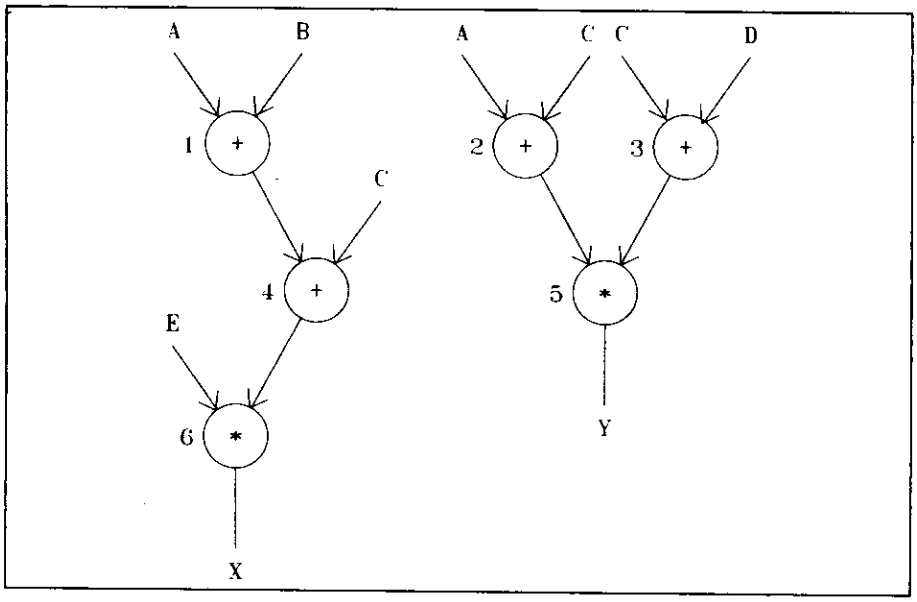


图12-14 随意标记节点的数据流图

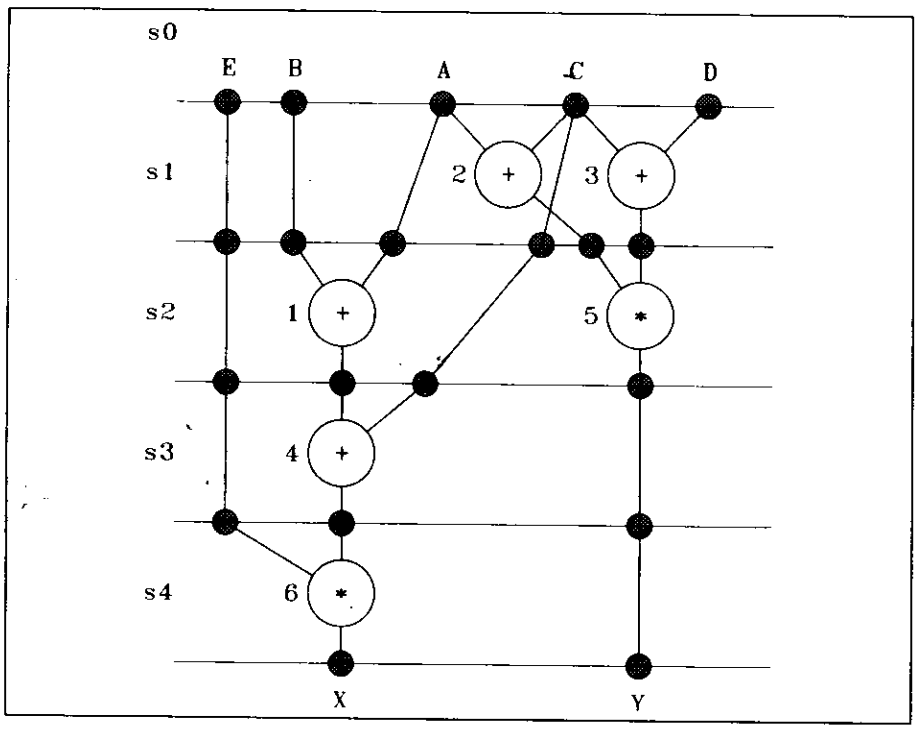


图12-15 ASAP调度

12.3.4 ALAP调度

ALAP方法对每个操作都尽可能晚地调度。该算法很像ASAP调度算法，不同的是它从数据流图的底部开始向上进行，并且从最后一个控制步骤开始向第一步进行。因为在开始时并

不知道总共需要多少个控制步骤，所以这里也不知道最后一步的编号。因此，直到调度完成以后才开始对控制步骤编号。

	s0	s1	s2	s3	s4
A1	--	2	1	4	--
A2	--	3	--	--	--
M1	--	--	5	--	6
M2	--	--	--	--	--

图12-16 ASAP调度的Gantt图

考虑图12-14，数据流图最底部的两个操作都是乘法操作。因为这里有两个可用的乘法器，所以在最后一个控制步骤里对这两个操作都进行调度。

对于倒数第二个控制步骤，需要考虑三个操作（2、3和4）。由于它们三个都是加法操作，而一共只有两个加法器，根据局部标准，这些步骤中我们任意选择出操作2和3，使用两个加法器。这里并不检查这种选择可能造成的潜在冲突。它留下操作1和4在更前的一步里调度。然而，因为这两个操作的数据在这一步中并不可用，所以必须把操作1和操作4调度在不同的控制步骤里。图12-17是ALAP调度后的结果，它也需要5个控制步骤。

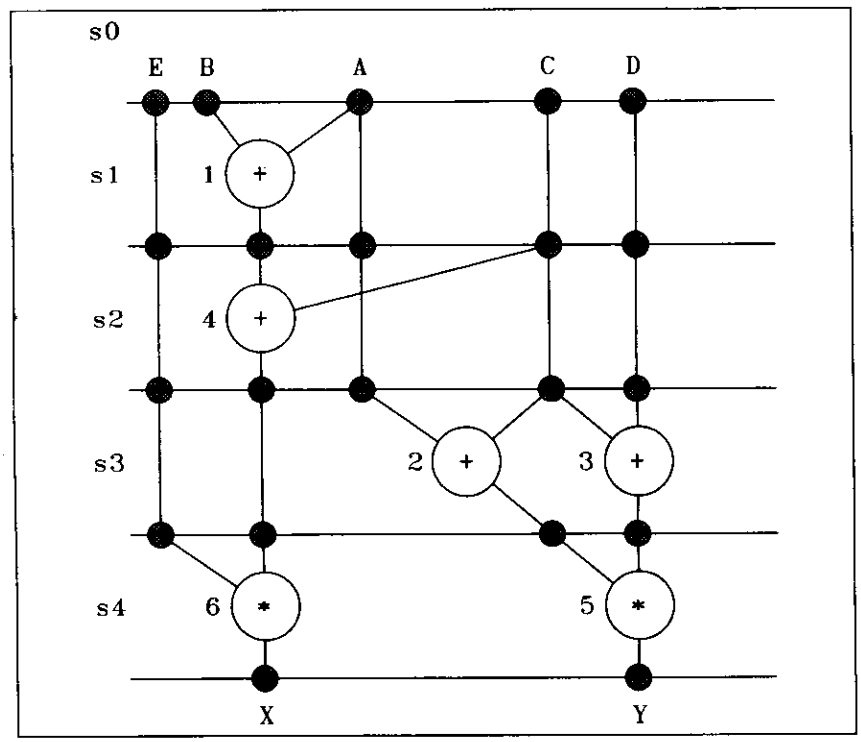


图12-17 ALAP调度

再次使用贪心算法分配功能部件后，所得的Gantt图如图12-18。部件的利用率同样也很低，表明可能存在一种更好的调度方法。

	s0	s1	s2	s3	s4
A1	--	1	4	2	--
A2	--	--	--	3	--
M1	--	--	--	--	5
M2	--	--	--	--	6

图12-18　ALAP调度的Gantt图

12.3.5　列表调度

ASAP和ALAP算法的问题在于它们没有对“关键路径”上的操作赋予优先级。因此，可能会把有限的资源浪费在非关键操作上。结果使得在后面步骤里，资源为等待关键操作的执行结果而闲置。这个问题可以通过使用全局标准选择下一个被调度的操作来解决。

关键路径指的是数据流图上最大长度的路径。例如，图12-14的数据流图上的关键路径长度为2，即路径1-4-6。由于在关键路径上的操作一旦被延时将会增加总的执行时间，所以这些操作应该比其他操作具有更高的优先数。但是，关键路径是一个动态的概念。一旦某些操作被调度了，余下操作中的关键路径就可能不同于初始时的关键路径。

有多种标准支持识别当前关键路径。一种方法是以某个节点到图的底部的最大长度来定义节点的关键路径优先数。图12-19为图12-14数据流图的节点在括号中定义了关键路径优先数。

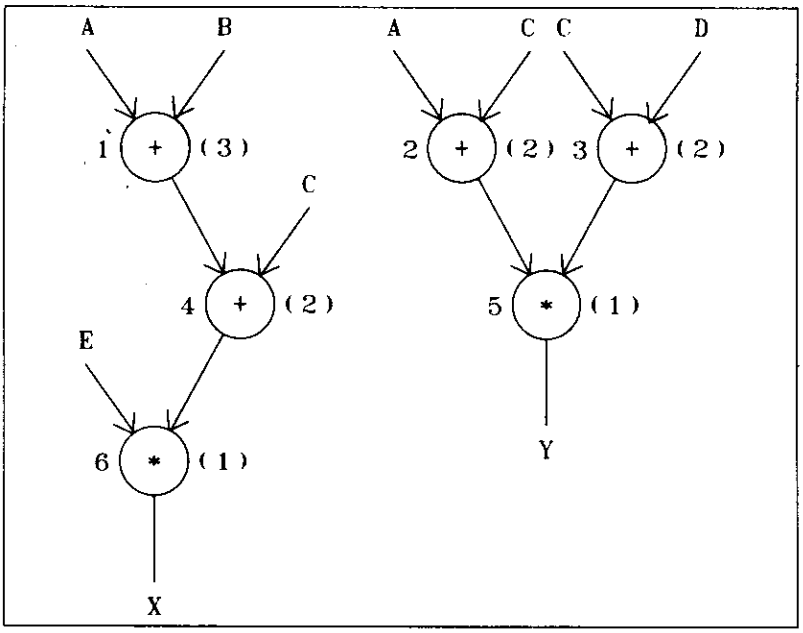


图12-19　关键路径优先数

然后对优先数降序形成一张列表。这种方法从表头开始，调度尽可能多的操作。它在每一步给距离数据流图底部最远的操作给出优先数。这就是该方法被称作列表调度的原因。

对于图12-19，排序后的列表是（1，4，2，3，6，5）。同一关键路径上优先数相同的节点在列表中的位置是任意的。也可以使用其他的标准来为优先数相同的节点排序以改进整体调度。对图12-19的数据流图使用关键路径优先数的列表调度如图12-20。

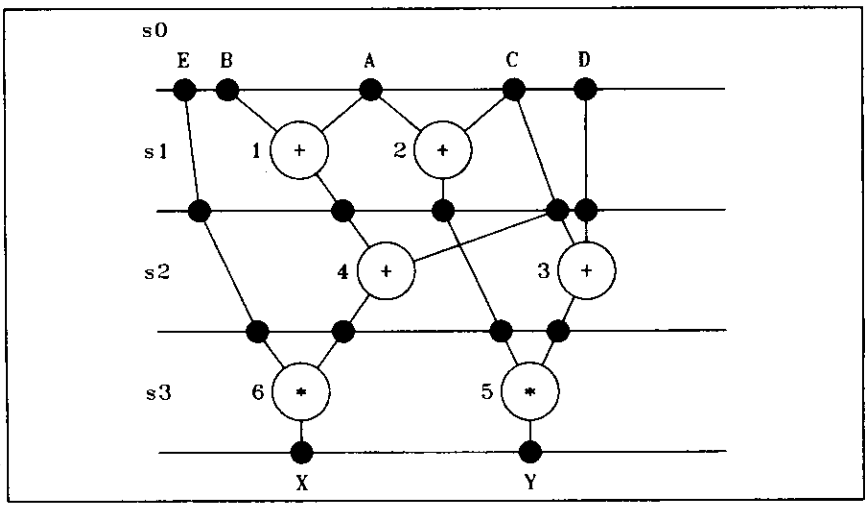


图12-20 采用关键路径优先数的列表调度

列表调度的过程如下。使用关键路径调度，在控制步骤s1可以调度三条加法操作，它们分别是1、2和3。根据关键路径上的节点所具有的优先数，操作1的优先数比2和3要高。因此，在控制步s1里指派操作1和操作2或3。本例中，任意选择的结果是操作2。在控制步骤s2中，只有操作3和操作4可以被调度。因为有两个加法器，这两个操作都可以指派到步骤s2里。操作5和6留到了步骤s3。因为有两个乘法器，所以这两个操作也可以在该步中被调度。

图12-21是使用了贪心算法指派功能部件的调度结果。通过为关键路径上的操作赋优先数，获得了只有四个控制步骤的调度结果，而使用ASAP和ALAP算法的结果都需要五个控制步骤。该调度相对于整个执行时间是一种优化调度，因为在每个控制步骤里只能对关键路径1-4-6中的一个操作进行调度。显然，由于关键路径上的操作只能顺序执行，所以不可能再找到一种更好的调度结果。

	s0	s1	s2	s3
A1	--	1	4	--
A2	--	2	3	--
M1	--	--	--	5
M2	--	--	--	6

图12-21 采用关键路径优先数的列表调度的Gantt图

另一观察结果表明，这种调度相对于ASAP和ALAP调度来说，对三种部件的负载更加均衡。这一结果使得很多研究人员追求把负载均衡概念作为一种优化标准。力量制导（*force directed*）调度的概念就是它的结果，但是在这里并不讨论这种方法。

列表调度在选择下一个被调度的操作时是全局性的，但在选择操作要在其中调度的下一个控制步骤时是局部性的。

12.3.6 自由调度

这种方法在选择下一个被调度的操作以及选择操作在其中调度的控制步骤时都是全局性的。支持自由调度的基本原理是首先调度最不自由的操作，这是因为它们是最难调度的操作而且是以后最容易引起阻塞的操作。延时更加自由的操作是因为它们在调度时可以有更多的选择。

执行ASAP和ALAP调度的第一步是获得操作被调度的控制步骤的范围。控制步骤的上界必须确定。例如，图12-12例子的上界是4（不包括初始步骤s0），每个操作被调度的控制步骤的范围如表12-3。显然，最不自由的操作是1、4和6。在控制步骤s1，可以被调度的指令是1、2和3，它们都是加法操作。由于这里只有两个加法器，在步骤s1时刻只能选择其中的两个。因为操作1是最不自由的，所以它在s1里被调度。操作2和3是同等自由的，可以随意选择其中的一个。假设选择的结果是2。

表12-3　图12-14数据流图控制步骤的范围

操　作	最早ASAP	最晚ALAP	范　围
1	1	2	2
2	1	3	3
3	1	3	3
4	2	3	2
5	2	4	3
6	3	4	2

在控制步骤s2里可以调度的操作是3和4。它们都是加法操作，而且有两个加法部件，所以它们都在此被调度。在控制步骤s3，操作5和6都可以被调度。它们都是乘法操作，这里也有两个乘法器，这两个操作也都在此被调度。它的结果也如图12-20。这个例子表明，自由调度产生一个优化调度。

12.4 分配方法

数据路径分配包括为操作指派功能部件，指派存储数据值的寄存器，以及使用多路复用器和（或）总线来设计寄存器和功能部件间的互连。它的结果是一个寄存器级的表示。分配过程可能受到一个或多个定义限制的约束，如最大的时间延时、最大的实数表示范围、最大的功耗或最大的总开销。结合了目标与限制的定义增加了设计过程的复杂度。例如，在定义里可能要求一种不超过一定延时时间的最小开销方案。

分配方法通常被划分为两种：迭代/构造分配和全局分配。迭代/构造方法使用局部条件选择下一个被分配的操作。每一操作的分配直到它所需要的所有部件均已获得为止。全局方法检查剩下的所有操作，从中找出既可以满足设计目标又不超出任何特定限制的分配项。穷举

搜索属于一种全局方法。它保证结果是一个最优的解决方案，但需要大量的计算时间，迭代/构造方法一般来说更加有效，这是因为它们在很小的空间里搜索，但是它们很难找到一个优化解。

12.4.1 贪心分配法

贪心分配是一种使用局部选择条件的迭代/构造方法。它自顶而下处理调度图。每一个操作都被指派下一个可用的功能部件；每一个数据赋值到下一个可用寄存器，每一条数据路径指派到下一个可用的总线或多路复用器。在每一个步骤，只有在当前所有类型的部件都处于忙状态时才会增加一个新的部件。之所以用贪心法为该过程命名是因为它倾向于重用第一个找到了的可用部件，而不进一步考虑以后可能引起的潜在冲突。换句话说，它贪心地吞噬所有可用的资源而不为以后考虑。一些人声称该算法更合适的名称是国会算法！不管怎么说，该方法易于自动操作，而且在某些环境下可以给出比较好的结果。然而，它很难得到全局优化结果，这一点更像国会的作风。

12.4.2 穷举搜索分配

全局分配方法使用了比迭代/构造更加复杂的选择标准。这种方法增加了搜索空间及获得优化方案的概率。它的代价是复杂度和计算时间的增加。

穷举搜索是一种全局分配方法。这种方法尝试整个分配空间里所有可能的分配方案并从中选出最好的一个。显然它保证了结果是优化的。但除非是一些小例子，该方法的计算复杂度是难以忍受的，即使是最快的计算机也不能在合理的时间内解决一个典型的问题。后面几节讨论的方法都在某些方面与穷举搜索有一定的联系，所以这种方法也是一种值得考虑的方法。

12.4.3 左边界算法

左边界算法经常用于调度之后的寄存器分配。它属于一种迭替代/构造方法。左边界算法的步骤总结如下：

1) 给调度流图中的每个存储设备一个唯一的标签。图12-22给出了调度图12-20的一种标记。

2) 准备一张用于表示数据存储生命周期的图。图12-23a说明了该图的组织方法。例如，数据值E必须从控制步骤s1保持到s2和s3。在s3期间，它作为乘法部件的输入。同样，数据值G在步骤s1结束时产生，它必须在步骤s2一直保持，直到在s3期间被使用。

3) 通过组织时间为生命周期排序。开始时间相同的生命周期被组成一组。组之间的顺序是开始时间早的排在前面。在具有相同开始时间的组的内部，根据结束时间排序，即结束时间越早的生命周期排在越靠前的位置。图12-23b是对图12-23a生命周期排序的结果。

4) 从左向右为可用的寄存器排序，优先级高的寄存器排在左边。通常这种排序是任意的。数据值以图12-23b排列的顺序从左到右分配寄存器。每一个数据值都以图12-23c列表中的顺序从最左端开始分配赋值时可用的寄存器。最终的结果如图12-23c。

在s1结束时产生的数据值可能作为一个数据值存储在s1步骤使用的寄存器里。例如，数据值G可能存储在数据值B使用的寄存器里。另一方面，数据值G不能存储在存放数据值C的寄存器里，这是因为这两个数据值在步骤s2都将被存储。

本例里，因为在s1步骤需要存储五个数据项，因此至少需要五个寄存器。随机对这些寄

存器排序为R1到R5。数据项A、B、C、D和E分别存放到R1、R2、R3、R4和R5。然后数据项F分配到R1，数据项G分配到R2，这两个寄存器在s2期间都是可用的。同样，项H和I分别分配到R1和R3，这两个寄存器在s3期间可用。项X和Y在s4步骤分派到寄存器R1和R2。

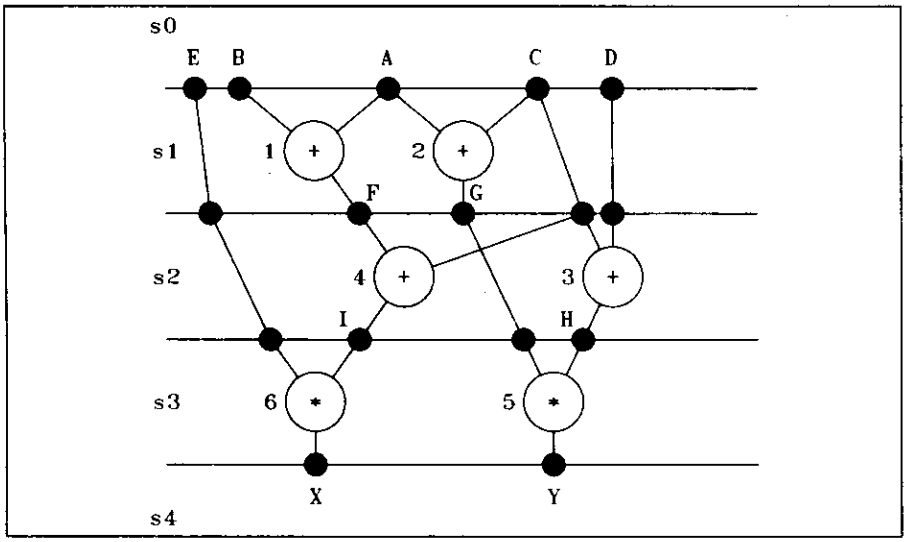


图12-22 标记存储设备的DFG

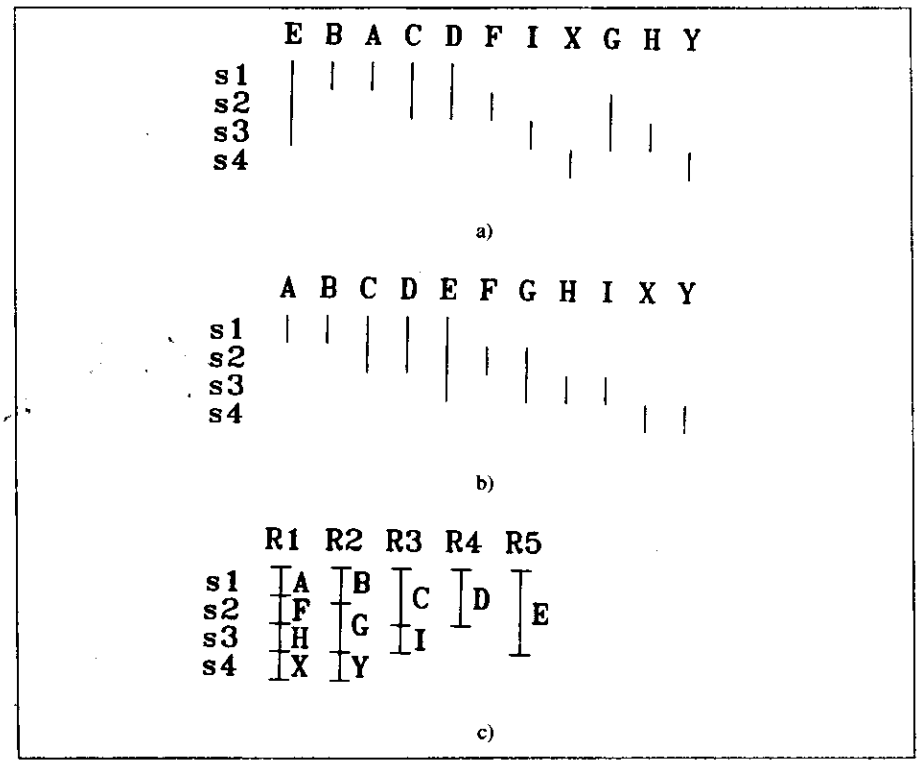


图12-23 解释左边界算法的步骤

左边界算法并不考虑数据路径的开销。它只把所需要的寄存器的数量减到最小。在步骤

s3也可以把项H分配到R2，并把项I分配到R1。为数据值分配寄存器通常被称为把数据值绑定到寄存器，如果在没有增加额外寄存器的情况下还有其他分配选择，我们希望在分配功能部件和互连路径完成之后才开始绑定数据值。有一种谨慎的选择可以节省互连开销，下一节会讨论这种方法。

12.4.4 分配功能部件及互连路径

为操作分配功能部件、为数据值分配存储器及为互连路径分配多路复用器或总线，这三者之间是相互联系的。如果使用前一节讲述的左边界算法把数据绑定到寄存器，而不考虑功能部件的分配和互连开销，得到的结果可能就不是一个优化的设计。

为了说明这种效果，考虑为图12-22中的调度分配功能部件和互连路径。假设左边界算法已经为寄存器分配进行了优化，但是我们对可以延时绑定的部分做出标记，前一节已经介绍了这一点。

设计中的一个约束是只有两个加法器和两个乘法器。通过执行左边界算法，我们知道需要五个寄存器来存放数据值。图12-24列出了所有需要的功能部件和寄存器集合。为操作分配功能部件和最终为数据值分配寄存器都需要减少多路复用器的开销。某些情况下，如果到寄存器输入或功能部件输入的连接只有一个，则不需要使用多路复用器。如果有多个连接，我们希望可以通过合理的分配使得MUX的端口数最少。

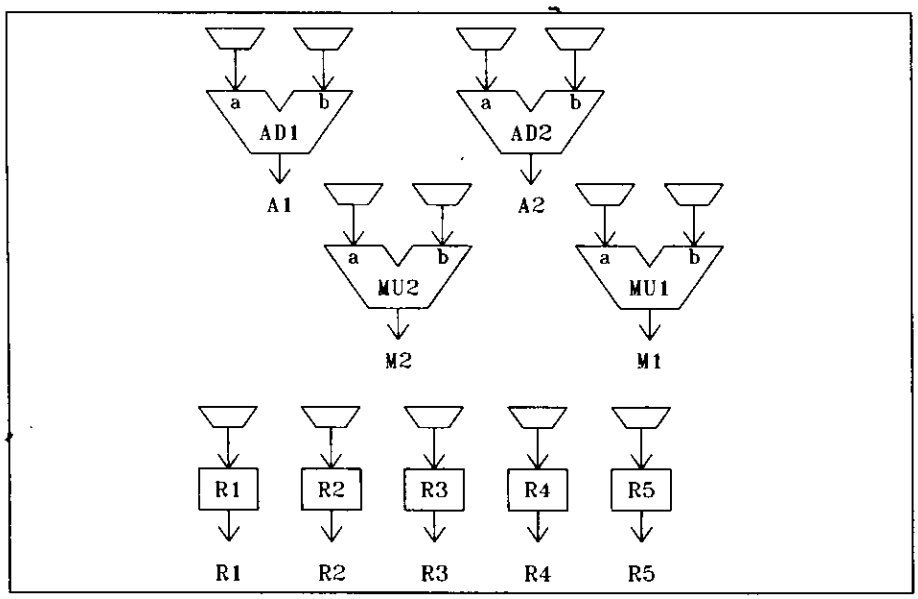


图12-24 列表调度示例所需的功能单元和寄存器

尽管在这里使用MUX操纵数据传输，但三态门对此也同样有效。

减小MUX的大小需要减少作为MUX输入的数据源的数量。图12-25所示的图可以用来记录决定并辅助形成下一决定。每一个时钟步骤对应一行，每一个寄存器或功能部件的输入对应一列。我们将记录图中不断形成的连接。

在步骤s1期间，对于图12-23c的寄存器赋值，数据值A、B、C、D和E分别绑定到寄存器R1、R2、R3、R4和R5。图12-26表示行s1里的标号A、B、C、D、E对应的列为R1、R2、R3、

R4和R5。依据图12-22的调度数据流图，在步骤s1需要把操作1和2绑定到加法器AD1和AD2。由于在此之前还没有绑定任何操作，可以把操作1绑定到AD1，把操作2绑定到AD2，在图12-26中表示为在列AD1-Op写上1，在列AD2-Op写上2。因为算术加法是可交换的，所以可以把A（R1的输出）或B（R2的输出）作为AD1的输入“a”，另一个连到输入“b”。同样，由于之前没有任何连接，可以在s1把R1连到AD1b并把R2连到AD1a。图12-26中表示在列AD1b对应于s1行的位置写入R1，在列AD1a对应于s1行的位置写入R2。同样，由于AD2没有任何连接，可以把A（来自R1）和C（来自R3）分别连接到输入AD2a和AD2b。现在就完成了图12-26中连接图的第一行。

	AD1			AD2			MU1			MU2							
	Op	a	b	Op	a	b	Op	a	b	Op	a	b	R1	R2	R3	R4	R5
s1																	
s2																	
s3																	
s4																	

图12-25　示例列表调度的空白连接图

	AD1			AD2			MU1			MU2							
	Op	a	b	Op	a	b	Op	a	b	Op	a	b	R1	R2	R3	R4	R5
s1	1	R2	R1	2	R1	R3	-	-	-	-	-	-	A	B	C	D	E
s2																	
s3																	
s4																	

图12-26　示例列表调度的部分连接图

在步骤s2，必须把操作3和4绑定到AD1和AD2，并把数据值F和G赋给寄存器R1和R2。左边界算法把F绑定到R1，把G绑定到R2，但是也可以把G绑定到R1、F绑定到R2来降低开销。

操作3需要使用来自R3(C)和R4(D)的数据。通过对AD2的赋值，可以利用已有的来自R3的连接。操作4需要使用F和C的数据。如图12-27，通过把F绑定到R1、把G绑定到R2，可以使用在s1里建立的从R1到AD1的连接。因此，在s2步骤，操作到功能部件、数据值到寄存器的绑定对互连的开销有显著影响。这种绑定要求在s1结束时AD1（A1）的输出连接到R1的输入，并且AD2（A2）的输出绑定到R2的输入。列R1行s2的F-A1表示数据值F在控制步骤s2存储在寄存器R1，并且加法器AD1（A1）的输出必须连接到R1以使数据F传输到R1。寄存器R1和R2现在都有了一个输入。R1的输入连接到A和A1，R2的输入连接到B和A2。图12-27的行s2表示了这种赋值的连接结果。AD1下的“b”列的值都是R1，而“a”列的值为R2和R3。s2步骤的结果是AD1的“b”输入不需要MUX，而“a”输入需要一个二输入MUX。类似地，AD2的“b”输入不需要MUX，而“a”输入需要一个二输入MUX。

在步骤s3，必须为操作5和6指派两个乘法器（M1）和（M2）。由于此时还没有对乘法器指派操作，所以可以随意分派。但是，根据左边界图（图12-23），数据H和I必须绑定到寄存器R1和R3，或者是R4和R3。这种选择提供了一个新的决定。

	AD1			AD2			MU1			MU2							
	Op	a	b	Op	a	b	Op	a	b	Op	a	b	R1	R2	R3	R4	R5
s1	1	R2	R1	2	R1	R3	-	-	-	-	-	-	A	B	C	D	E
s2	4	R3	R1	3	R4	R3	-	-	-	-	-	-	F-A1	G-A2	C	D	E
s3																	
s4																	

图12-27 示例列表调度的部分连接图

因为I的值在步骤s2由AD1产生，所以必须对绑定到I的寄存器提供一条来自AD1输出的连线。由于从AD1的输出到R1已经有了一条连线，所以如果把I绑定到R1，就不需要再对R1的输入增加任何新的连接。左边界算法并不具有这一信息，因此它可能把I绑定到R3。因此，有了这一新的信息，就可以决定把I绑定到R1。

因为H的值在步骤s2由AD2产生，所以必须对绑定到H的寄存器提供一条来自AD2输出的连线。由于AD2的输出并没有与任何寄存器R1、R3或R4连接，所以不论那种情况都需要增加一条新的连线。由于已经决定把I绑定到R1，因此可以把H绑定到R3。现在的结果如图12-28。

	AD1			AD2			MU1			MU2							
	Op	a	b	Op	a	b	Op	a	b	Op	a	b	R1	R2	R3	R4	R5
s1	1	R2	R1	2	R1	R3	-	-	-	-	-	-	A	B	C	D	E
s2	4	R3	R1	3	R4	R3	-	-	-	-	-	-	F-A1	G-A2	C	D	E
s3	-	-	-	-	-	-	6	R1	R5	5	R3	R2	I-A1	G-A2	H-A2	-	E
s4																	

图12-28 示例列表调度的部分连接图

在步骤s3结束时必须把乘法器的值存储到寄存器中。尽管并不需要步骤s4来计算这些结果，但是这些值只有到了s4才会有效。所以我们在图中增加了s4步骤。对X和Y的绑定有另一种选择。从R1到R5的寄存器都可以被使用。寄存器R1、R2和R3的输入都有两个连接，所以使用二输入MUX比较有效。寄存器R4和R5都只有一个连接，所以不需要使用MUX。由于X和Y都来自乘法器，而且从乘法器到任何一个寄存器都没有连线，所以对X和Y都需要增加新的路径。如果增加到R3或R5的路径，必须在新路径上增加一个二输入MUX。如果增加到R1、R2或R3的路径，则必须把二输入MUX至少扩展到三输入。大部分情况都意味着要增加到四输入MUX，最终的决定取决于哪一种情况的开销更小。这一决定也可能取决于技术因素，我们希望在决定了方法之后才进行绑定。

为了完成这一例子，决定把X绑定到R1（需要从M1到R1的连线），并把Y绑定到R2（需要从M2到R2的连线）。图12-29是完整的连接表，图12-30是最终的连接图。

互连图以线性方式直接由连接表构造。例如，因为AD1的“b”列只有一个值R1，所以直接把R1的输出连接到AD1的输入“b”。因为AD2a的“a”列有R1和R4，所以需要一个二输入MUX来选择连接到AD2的输入“a”的R1或R4。

在R1列，有表项A、F-A1、I-A1和X-M1。“A”表明数据项A赋值到寄存器R1。由于A是初始输入，所以有一条从初始输入到R1的连线。F-A1和I-A1表示数据F和I都赋值给R1。因为

F和I都是由AD1产生的输出，所以必需有一条从AD1的输出到R1的连线。同样，X-M1表示数据X赋值给寄存器R1，它需要一条从M1的输出到R1的连接。这里需要一个MUX来选择数据路径，或者这三条数据源使用三态门连接到总线上，通过门来控制寄存器R1到总线的连接。

	AD1			AD2			MU1			MU2							
	Op	a	b	Op	a	b	Op	a	b	Op	a	b	R1	R2	R3	R4	R5
s1	1	R2	R1	2	R1	R3	-	-	-	-	-	-	A	B	C	D	E
s2	4	R3	R1	3	R4	R3	-	-	-	-	-	-	F-A1	G-A2	C	D	E
s3	-	-	-	-	-	-	6	R1	R5	5	R3	R2	I-A1	G-A2	H-A2	-	E
s4	-	-	-	-	-	-	-	-	-	-	-	-	X-M1	Y-M2			
mux inps		2	1		2	1		1	1		1	1	3	3	2	1	1

图12-29　示例列表调度的最终连接图

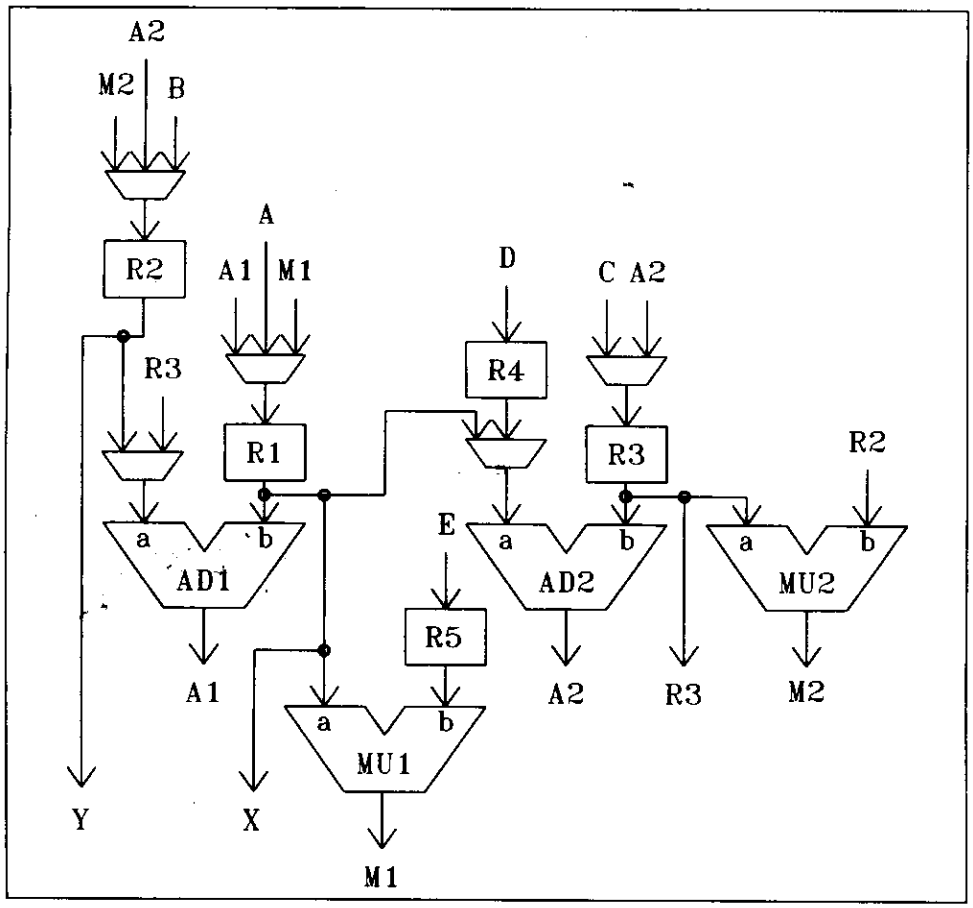


图12-30　示例列表调度的连接图

表12-4给出了这一实现的部分列表。如果使用的是标准部件，将会用4×1的MUX来替换3×1的MUX。

表12-4 示例的部分列表

类 型	数 量
加法器	2
乘法器	2
寄存器	5
2 × 1 MUX	3
3 × 1 MUX	2

注意到在每一步骤里仍然可以有多种选择，而且不同选择对后面的影响并不总是很明显。因此，还需要一些技巧来优化做出的选择。以下几节将描述用正规的数学方法来辅助分配过程。

12.4.5 分配过程的分析

用图形的方法可以很好地表示分配过程。无论是分配寄存器、功能部件、还是互连路径，只要没有使用冲突，都可以把某两项分配给同一个器件。被分配的项被称为变量。变量被指派到资源。在为数据分派寄存器时，数据是变量，寄存器是资源。当把操作指派到功能部件时，操作是变量，功能部件是资源。在定义连接时，可以为连接指派一个多路复用器或数据总线。在这里，连接是变量，多路复用器或数据总线是资源。这三种分配行为都被模拟为一系列变量到一系列资源的指派。每一种指派都受到行为调度中定义的冲突的限制。本节里，假设已经通过一些方法确定了一种调度。把获得的从变量到资源采取的使得总资源数最小的指派称为资源分配问题。

现在定义一种资源分配问题的图形表示。每个变量用无向图的节点表示。如果两个变量可以指派到同一节点，则把这两个节点之间用一条弧线连接。最后的结果称为兼容图。两个节点之间的连接边被称为兼容。图12-31a是为图12-15所示调度进行功能部件分配的兼容图。只有可以指派到同一功能部件并且在不同时钟周期里活跃的两个操作才是相互兼容的。兼容图通常从完全图开始（每对节点之间都有一条边），通过不断消除不兼容节点之间的边进行构造。

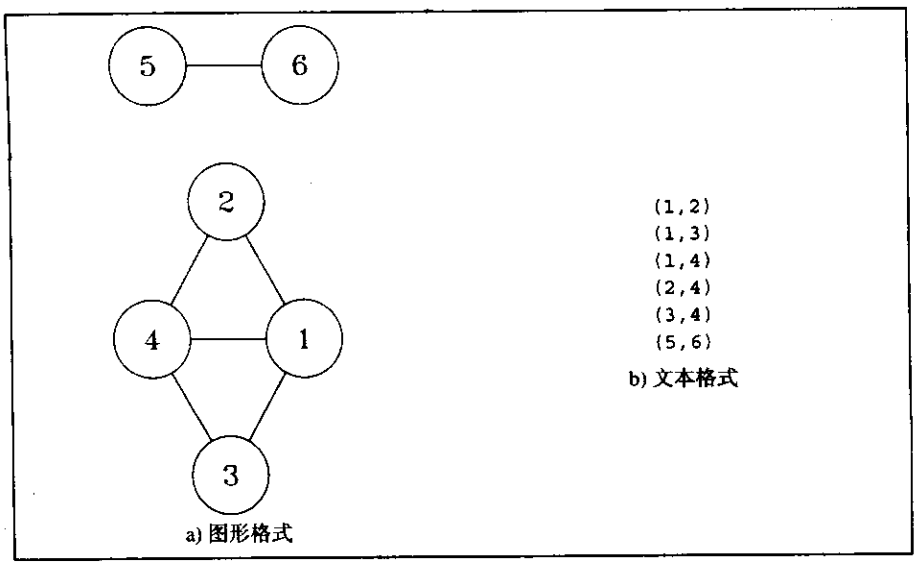


图12-31 ASAP调度的功能部件分配的兼容图

为了构造图12-15中操作的兼容图，首先从六节点的完全图开始。假设有独立的功能部件来执行加法和乘法操作。因为是不同的操作，所以操作5与操作1、2、3和4都是不兼容的。通过调度，操作2和3都是加法操作，但是由于它们发生在同一时钟周期，所以也是不兼容的。另一方面，加法操作1和4因为发生在不同时间周期，所以是兼容的操作。因此，可以使用同一个加法器无冲突地执行这两个操作，在兼容图里我们保留节点1到4之间的边。

也可以方便地使用特定的整数表示每个节点，并用一系列节点对来表示图形，如节点队(i, j) $(i<j)$表示节点i和j之间的边。通过以i的升序和j的升序不断对节点排序，该图可以用图12-31b所示的唯一的整数对顺序表来表示。

在以使用尽量少的功能部件为目标时，至少需要三个功能部件。其中由于操作2和3是并发执行的，所以至少需要两个加法器。同样，至少需要一个乘法器。通过观察图12-31a的兼容图，很明显操作5和6可以用同一个乘法器执行。而且，操作1、2和4可以用同一个加法器完成。表12-5给出了一些把操作映射到三个功能部件的方案。

表12-5　操作到功能部件的可能映射

	映　射			
单　元	1	2	3	4
MUL	5,6	5,6	5,6	5,6
ADD1	2,4	1,2	1,2,4	2
ADD2	1,3	3,4	3	1,3,4

可以映射到同一功能部件的操作集合称为兼容集。只有集合中任意一对操作都兼容的操作集合才构成兼容集。例如，操作{1，3，4}是兼容集，是因为(1, 3)，(1, 4)，(3, 4)都是兼容对。但是，{1，2，3}不是一个兼容集，这是因为尽管(1, 2)和(1, 3)都是兼容对，但(2, 3)不是兼容对。2和3之所以不是一个兼容对是因为它们所表示的两个操作不允许在同一周期内完成，因此需要不同的功能部件。每对节点之间都存在一条边的节点集合的子集称为完全子图。完全子图与兼容集之间存在一一对应关系。

因为每个操作都必须指定一个特定的功能部件，所以兼容集之间使用的功能部件的集合是互不相交的。同样，每个操作都必须指定功能部件。恰好只包括每个节点一次的节点子集的集合称为节点集合的划分。我们把兼容图中的完全子图定义为簇。需要解决的问题可以用图论中的术语表述如下：

最小簇划分问题（MCPP）：找出图形节点划分的簇的最少个数。

下一节将讲述该问题的数学表达式。

12.4.6　近似最小簇划分算法

图G中的一个集团指的是G的一个最大完全子图。对于图12-31a，完全子图{1，2，4}是一个集团，但是完全子图{2，4}不是。集合{2，4}是一个完全子图，但是它包含在子图{1，2，4}里。从图中搜索集团是一个NP完全问题。因此，可能没有一个搜索集团，也就是最大簇的有效算法。下面的算法用来搜索图中节点的近似最小簇的划分，它具有多项式的复杂度。这个算法假设图中所有节点都标记为唯一的整数。该算法来自Tseng和Siewiorek的一篇论文，发表于1986年。

近似最小的簇划分算法（Cluster Partitioning Algorithm, CPA）：

C1. 对于每条边（i, j），

a) 计算i和j的公有相邻节点的个数。i和j的公有相邻节点指的是，如果有节点r，r与i在一条边上相连，并且j与r在一条边上相连，则r是i和j的一个公有相邻节点。

b) 计算当i和j合并为一个节点时将要删除的边的个数。当i和j被合并时，边（i, j）及所有与i相连或与j相连但不同时相连的边将被删除。如果节点r是i和j的一个公有相邻节点，则保留一条从r到i和j的边，另一条边将被删除。

C2. 开始收集下一个簇的节点。

a) 如果图中没有边，则停止。否则继续。

b) 选择具有最大公有节点个数的边(p, q)。如果边的个数多于一条，则选择当p和q合并时删除的边数最少的那一个。如果该选择仍然多于一种，则任意选择其中的一个。

c) 将p和q合并为一个簇。我们将以标号大小递增的顺序列出簇中元素，并且把簇中标号最小的元素称为簇的头。此时，当前簇只包括两个元素p和q。

d) 调用更新图的子程序把节点p和q合并为一个节点，该程序将在后面给出。把合并得到的节点标记为p，p是簇头。此时，p代表簇$\{p, q\}$。

C3. 图中的节点p表示当前簇。通过以下步骤为当前簇增加节点。

a) 如果没有与p相连的边，则节点p代表整个簇。返回到步骤2开始寻找下一个簇。否则，继续为当前簇增加节点。

b) 考虑与p相连的节点。选择具有最多公有相邻节点的边（i, p）或（p, i）。如果有多于一种选择，则选择当i和p合并时删除边数最少的那一种情况。如果仍多于一种选择，则节点i任意选择其中的一个。

c) 把i添加到当前簇。如果$i<p$，则i成为新的簇头。否则p仍然作为簇头。

d) 调用更新图的子程序以合并节点i和p。

e) 置$p=\min\{i, p\}$。节点p成为新的簇头。

f) 重复步骤3。

合并节点x和y（$x<y$）的更新图子程序：

S1. 把节点x和y合并为一个节点，标记为x。

S2. 更新图中的边：

a) 删除所有包括节点y的边。

b) 对于任意节点r，若合并前节点r与y没有一条边，则删除r与x的边。注意：删除边的结果只保留代表节点x和y公有相邻节点的边。

S3. 重新计算公有相邻节点的个数及从合并后的图中删除的边的条数。

划分簇的实例

图12-32是一个兼容图，它具有边的顺序列表、每条边的公有相邻节点（CN）及合并这条边上两个端点时将要被删除边数的信息。为了说明C1的计算过程，考虑边（7, 8）。节点7和8只有一个公有相邻节点，标号为6。如果节点7和8合并为标号为7的节点，将会保留边（6, 7），这是因为6是7和8的公有相邻节点。由于5不是7和8的公有相邻节点，所以边（5, 7）将会被删除。因为节点8将会消失，所以要删除边（7, 8）、（6, 8）和（9, 8）。因此，一共需要删除四条边。7和8合并的结果如图12-33。注意：它只是说明C1计算的结果，并没有真正在算法

里合并节点7和节点8。

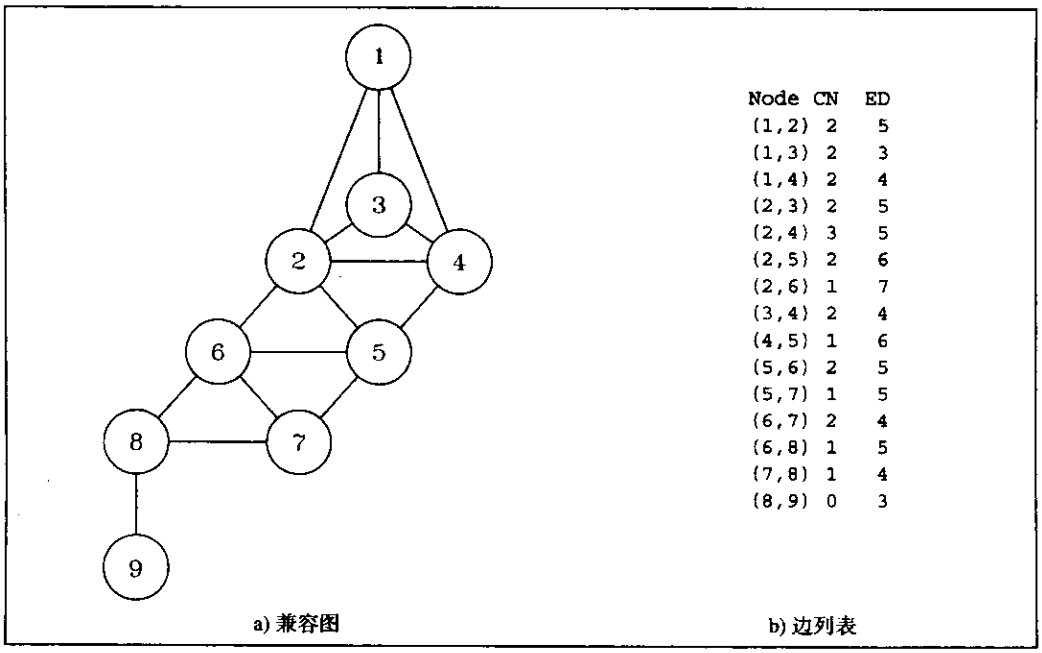


图12-32　兼容图示例

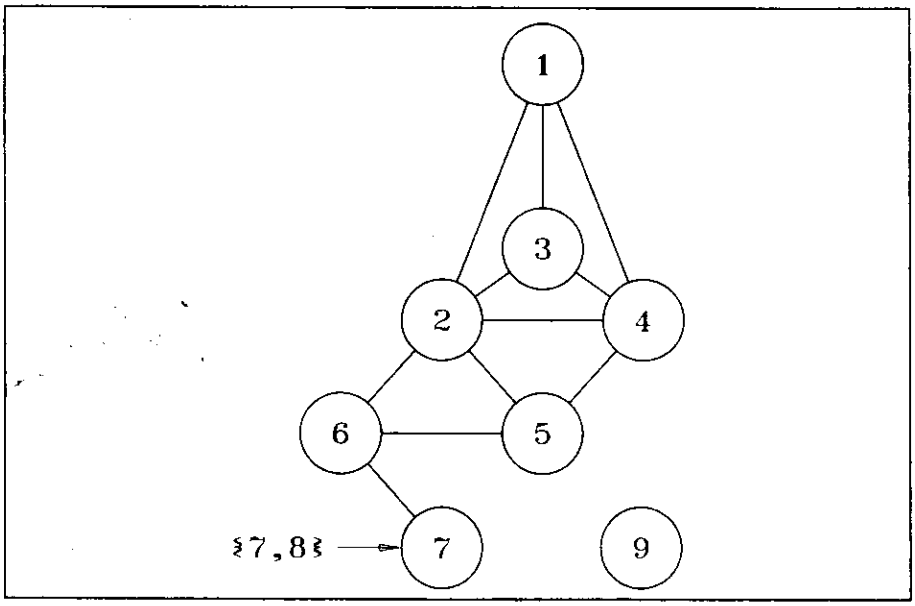


图12-33　合并节点7和节点8的结果

在步骤C2(b)里选择边（2，4），这是因为它具有最大个数的公有相邻节点（三个）。这个例子里，只有这条边有三个相邻节点，所以不需要进一步计算。此时，p=2，q=4。在步骤C2(c)，把节点2和4作为新簇的唯一元素。节点2是当前簇的头。

在步骤C2(d)，调用图更新子程序来合并节点2和节点4。在子程序里，因为2<4，所以x=2，

y=4。在步骤S1，把节点2和节点4合并为节点2。在步骤S2(a)，删除边（1，4）、（2，4）、（3，4）及（4，5）。在步骤S2(b)，由于初始图中不存在边（4，6），所以删除边（2，6）。我们保留了边（1，2）、（2，3）和（2，5），这是因为节点1、节点3、节点5都是节点2和节点4的公有相邻节点。图12-34是合并后图和边的列表。

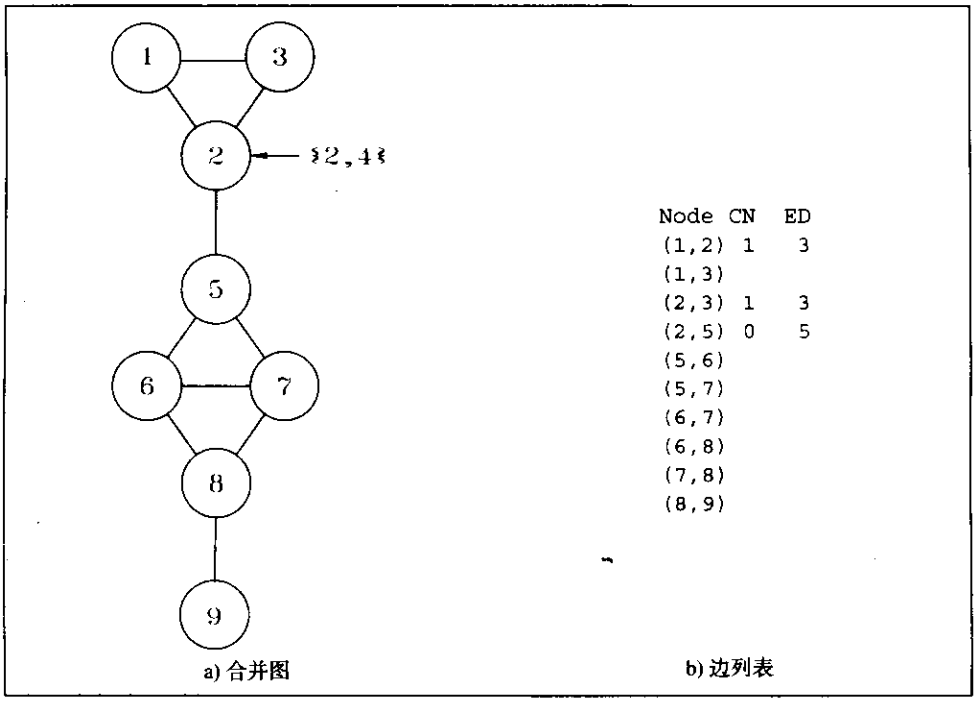


图12-34 合并节点2和节点4的结果

注意 在主算法的步骤C2(b)，只计算了与节点2连接的边的公有相邻节点数，并没有完成当前簇。它只用来在开始一个新簇时计算公有相邻节点和删除边的个数。现在，已经合并了节点2和节点4，对图进行了更新。这一步完成了主算法的步骤C2。

步骤C3(a)的下一步继续执行步骤C3(b)，这是因为图中仍有与节点2相连的边（1，3，5）。因此，还必须考虑边（1，2）、（2，3）和（2，5）。（1，2）和（2，3）都有一个公有相邻节点。由于这两种情况下要删除的边的条数都是三条，所以可任选一种。假设合并节点1和2。在步骤C3(c)，把节点1加入当前簇，现在簇中包括集合{1，2，4}。新的簇头为节点1。在步骤C3(d)调用图更新子程序，其中x=1，y=2。图12-35给出了新图和图中边的列表。在步骤C3(e)，p值为1，并作为新的簇头。

经过步骤C3，合并节点1和3。当前簇为{1，2，3，4}，图和边列表如图12-36。现在节点1代表簇{1，2，3，4}。因为节点1没有相连的边，所以在步骤C3(a)返回到步骤C2，开始下一个簇。现在为了执行步骤C2(b)，必须计算出公有相邻节点的个数及图中每个节点删除的边数。

在步骤C2(a)，由于图中仍有边存在，所以继续执行。在步骤C2(b)，我们选择边（6，7），这是因为只有这条边还存在公有相邻节点。合并了节点6和7之后，得到图12-37的图和边列表。

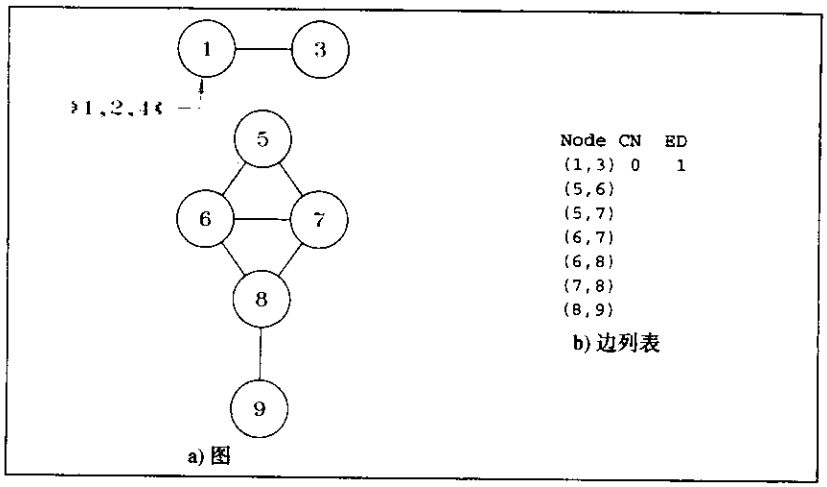


图12-35 合并节点1和节点2的结果

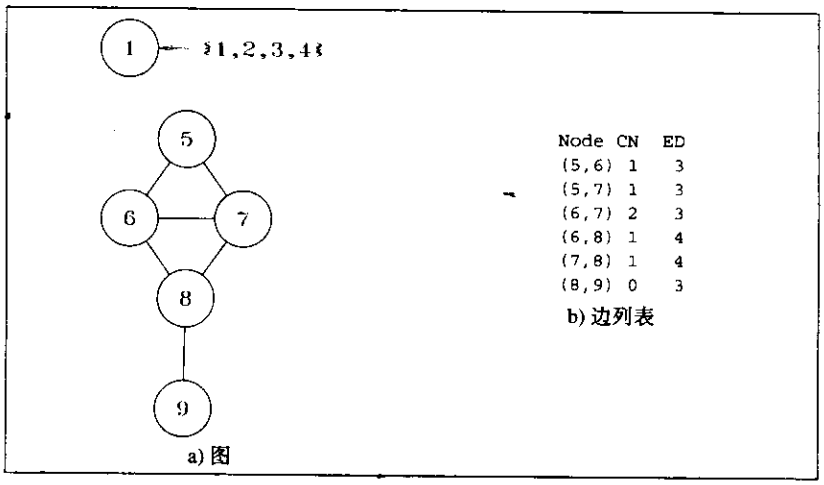


图12-36 合并节点1和节点3的结果

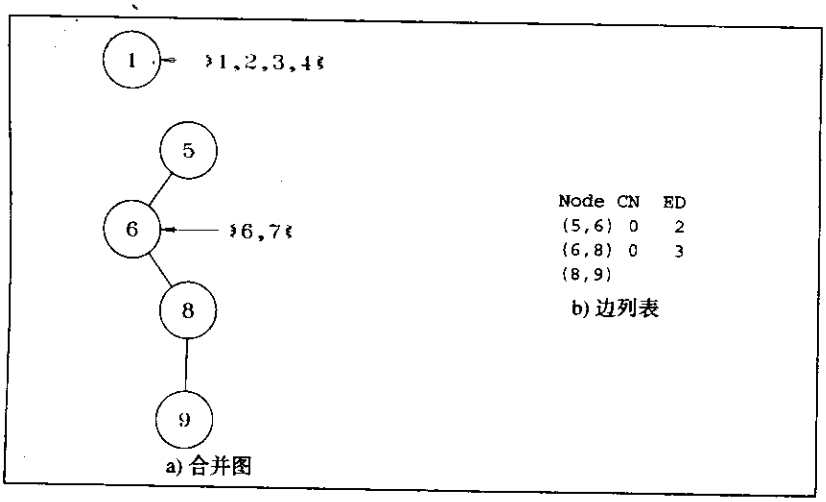


图12-37 合并节点6和节点7的结果

在步骤C3(b)，当前簇为{6，7}，图中的节点6代表这个簇。所有与节点6连接的节点都没有公有节点。但是，此时边（6，8）会导致三条边被删除，边（5，6）导致两条边被删除。因此，选择删除边数较少的边（5，6）。做出这种选择很重要，这是因为选择了（6，8）会把节点5孤立出来，形成一个额外的簇。

现在已经说明了所有情况，下面将给出最终结果。在算法完成之后，最终的簇为{1，2，3，4}、{5，6，7}及{8，9}。结果是九个变量被三个资源所覆盖。

CPA可以独立应用于任何三种资源分配问题。下一节将描述CPA的一种扩展，它允许使用额外的依赖于问题的信息来提高设计方案的质量。

12.4.7 利益制导簇划分算法

尽管前面讲述的簇划分算法可以得到较好的方案，但如果能够获得每次合并带来的潜在利益的信息，则可能改进它的性能。如果我们能够知道某一步合并可以带来更多的好处，则可以直接选择这个过程。

假设可以基于每条边的节点合并所带来的期望利益，把初始图中的边划分到一个类别集合。然后从1开始以利益大小的顺序为这些类别编号。例如，如果有两个类别，则类别1的利益大于类别2的利益。如果有m个类别，将从类别m的边开始，到类别1的边结束。对于每个$k>1$的类别，会产生包含有类别k在图G中所有边的子图，称为G_k。从G_m里包括最有利益的边开始，不断减少G和G_m，直到G_m没有边存在时结束。此时，已执行了最有利益的简化。然后，不断简化G和G_m直到G_m中没有边存在。继续这个过程直到完成边号最小的类别。此时，G等于G_1。前一节的算法将被应用于图G。

对图G和G_k将循环使用下面的算法合并类别k中的节点。该算法来自Tseng和Siewiorek在1986年发表的论文。

近似最小的利益制导簇划分算法（PDCPA）：给出图G和它的具有最大利益边的子图G_k，不断简化G和G_k。对子图G_k使用上一节的近似最小簇划分算法，并使用下面的图更新算法。

合并节点x和y（$x<y$）的利益制导图更新子过程：

1) 在G和G_k里把x和y合并为节点x。

2) 如下更新G和G_k中的边：

a. 从G和G_k中删除所有与节点y相连的边。

b. 如果节点r与y没有连接边，则从图G中删除节点r与节点x连接的边。如果G中要删除的边也在G_k里，则从G_k里也删除这条边。

c. 如果合并前节点r与x和y之间都有边相连，则在G中保留r与x的边。如果合并前这条边也在G_k里，则G_k里也保留这条边。如果合并前G_k里没有，则要在G_k里加入这条边。合并的结果改变了类别里的边。如果图G保留的边产生的新类别大于或等于k，并且这条边并不在G_k里，则在G_k里增加这条边。

d. 重新计算图G_k里所有边的公有相邻节点和删除边的条数。

1. PDCPA在寄存器分配中的应用

寄存器可以独立存在或作为RAM存储器的一部分。通常寄存器分配的目标是使得寄存器的数目最少。对于使用同一存储单元的两个值，它们的生命周期不能重叠。例如，假设图12-38a是使用某种调度算法获得的调度结果。图12-38b是这种调度结果的文本表示。调度是

程序循环的一部分，所以步骤s5之后紧跟着步骤s1。由于V7和V9的生命周期没有重叠，所以可以共享一个寄存器。V7只在步骤s3活跃，V9只在步骤s4活跃。V4、V6和V10的值在整个循环中都没有发生改变，它们被重复使用；即它们只在步骤s0被初始化，但是每次循环都会使用这些值。因为它们在循环期间一直活跃，所以不能共享寄存器。

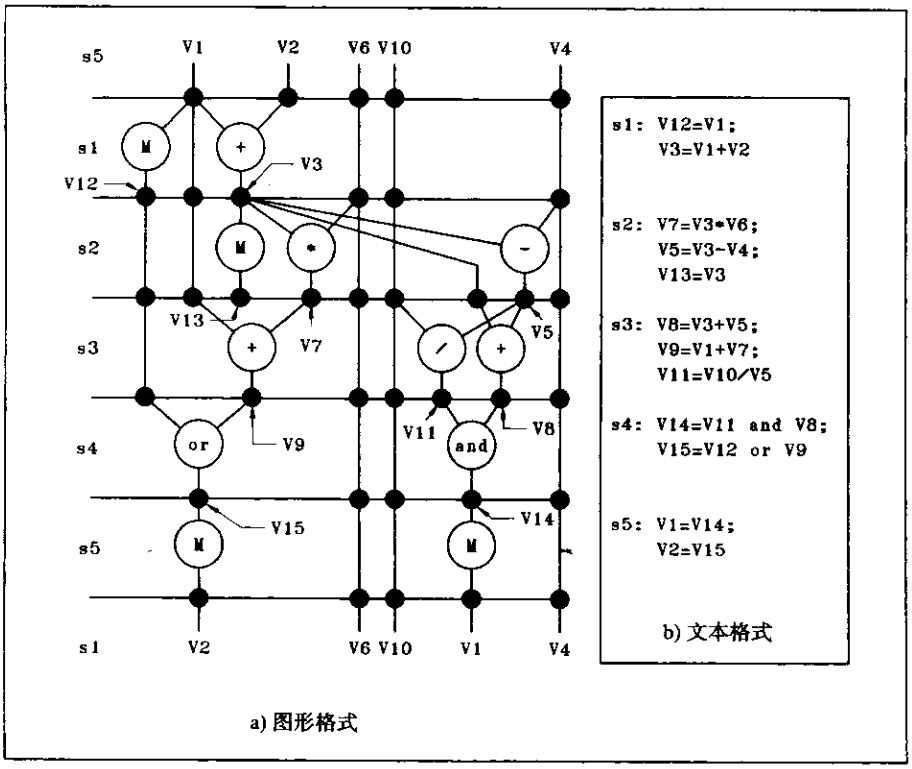


图12-38　示例程序的调度

利益制导簇划分算法可以获得一种使寄存器数目最少的寄存器分配方案。首先要构造出用数值表示节点的兼容图G。节点*i*和*j*之间有一条边，当且仅当*i*和*j*表示的值可以合并为一个寄存器时，它们的生存周期不发生重叠。图12-39给出了图12-38调度的兼容图的文本表示，其中值Vi表示为整数*i*。

```
(1,8)   (1,9)   (1,11)  (1,13)  (1,14)  (1,15)  (2,3)
(2,5)   (2,7)   (2,8)   (2.9)   (2,11)  (2,12)  (2,13)
(2,14)  (2,15)  (3,8)   (3,9)   (3,11)  (3,13)  (3,14)
(3,15)  (4,13)  (5,8)   (5,9)   (5,11)  (5,13)  (5,14)
(5,15)  (6,13)  (7,8)   (7,9)   (7,11)  (7,13)  (7,14)
(7,15)  (8,13)  (8,14)  (8,15)  (9,13)  (9,14)  (9,15)
(10,13) (11,13) (11,14) (11,15) (12,13) (12,14) (12,15)
(13,14) (13,15)
```

图12-39　寄存器分配的兼容图（copyright © 1986 IEEE）

下面定义一种纯数据传输移动操作，该操作把一个变量复制给另一个变量。下面的例子包括四个纯数据传输，用符号M表示。考虑在步骤s5把变量V15复制到V2。如果可以把V15和

V2存储在相同寄存器中，则不需要进行这一操作，这样就可以减少系统中的控制单元、寄存器和数据路径。此外，如果一个控制步里只有纯数据传输，则所有的操作均不必进行，因而可以除去这一步骤，这样可以加快电路操作。这个例子里，步骤s5中只有纯数据传输操作，因此可以消除整个控制步骤s5。

一直没有被使用的数据值，如V13，与其他所有的数据兼容，将被PDCPA自动消除。为了统计消除了纯数据传输后所带来的好处，把图中所有的边划分为两类。表示纯数据传输的边划分到类2，其他的边划分在类1。用G_2表示具有G中所有纯数据传输边的子图，G_1表示具有其他边的子图。当且仅当（i，j）在G中表示一个纯数据传输时，边（i，j）在图G_2中图12-40给出了子图G_2。不必构造子图G_1。

```
(1,14)  (2,15)  (3,13)
```

图12-40 子图G_2，它包括对应纯数据传输的边

对图G的子图G_2和G_1分别应用PDCPA可以减少寄存器的个数。通过对G和G_2应用PDCPA可以得到以下簇划分：

({1，14}，{2，15}，{3，13}，{4}，{5}，{6}，{7}，{8}，{9}，{10}，{11}，{12})

图12-41是简化后的图，其中节点1代表簇{1，14}，节点2代表簇{2，15}，节点3代表簇{3，13}。

```
(1,8)   (1,9)   (1,11)  (2,3)   (2,5)   (2,7)   (2,8)
(2,9)   (2,11)  (2,12)  (3,8)   (3,9)   (3,11)  (5,8)
(5,9)   (5,11)  (7,8)   (7,9)   (7,11)
```

图12-41 简化子图G_2的关系后得到的兼容图

对简化后的图12-41应用CPA可以得到下面的簇划分：

({1，14}，{2，7，9，15}，{3，8，13}，{4}，{5，11}，{6}，{10}，{12})

每个簇中的值（变量）可以指派到相同存储单元，它既可以是寄存器也可以是RAM地址。初始的15个变量被减少到8个。此外，由于步骤s5的所有数据传输都被消除，这一步也不再需要。图12-42a用图表示了简化后的调度，12-42b则用文本形式表示。

2. PDCPA在功能单元分配中的应用

功能部件分配的目标是减少计算单元的个数及数据传输进、出计算单元所需的逻辑门的数量。这里，通过把两个操作数分配到同一个功能单元可以获得的好处根据共有操作数的不同而不同。图12-43阐明了这些概念。

图12-43a显示了用单独的加法器实现两个加法操作所需的硬件。图12-43b显示了使用一个具有一对不同数据值的加法器实现两个加法操作所需的硬件。其中加法器的每个输入都需要一个多路器和选择信号，而且需要两个独立的控制存储结果的信号。较少的开销为节省的一个加法器除去两个多路器和两个额外控制信号的开销。由于两个多路器和两个额外控制信号的开销小于一个加法器的开销，所以这种设计较优。但是，如果三对数据中的每一个都是公有的，需要的硬件如图12-43c。其中当操作数对都不同时，通过两个多路器和三个控制信号所减少的复杂度相当于使用一个加法器的开销，而且使用一个加法器和一个控制信号所减少

的复杂度相当于使用独立加法器进行设计。

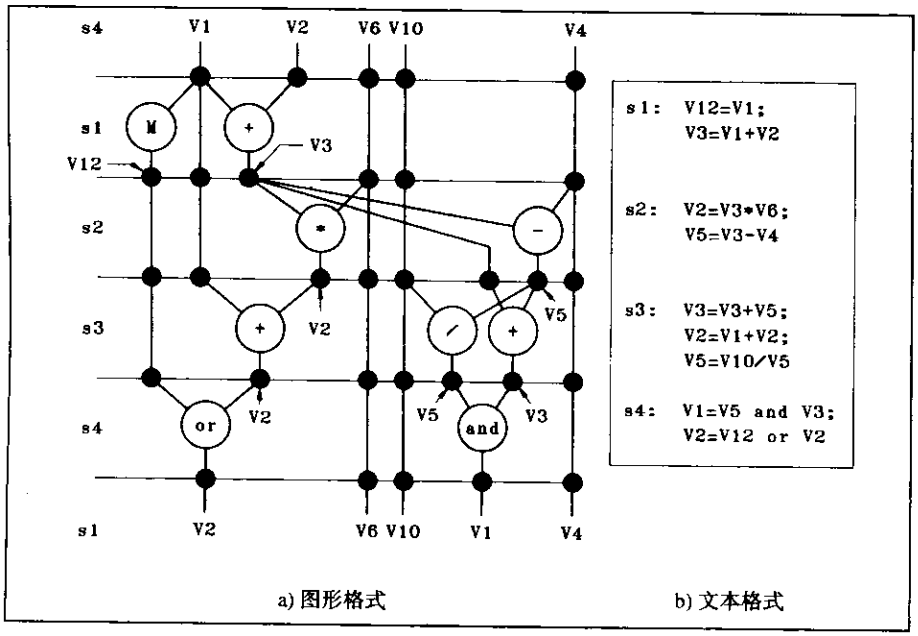


图12-42　寄存器分配后的调度表示

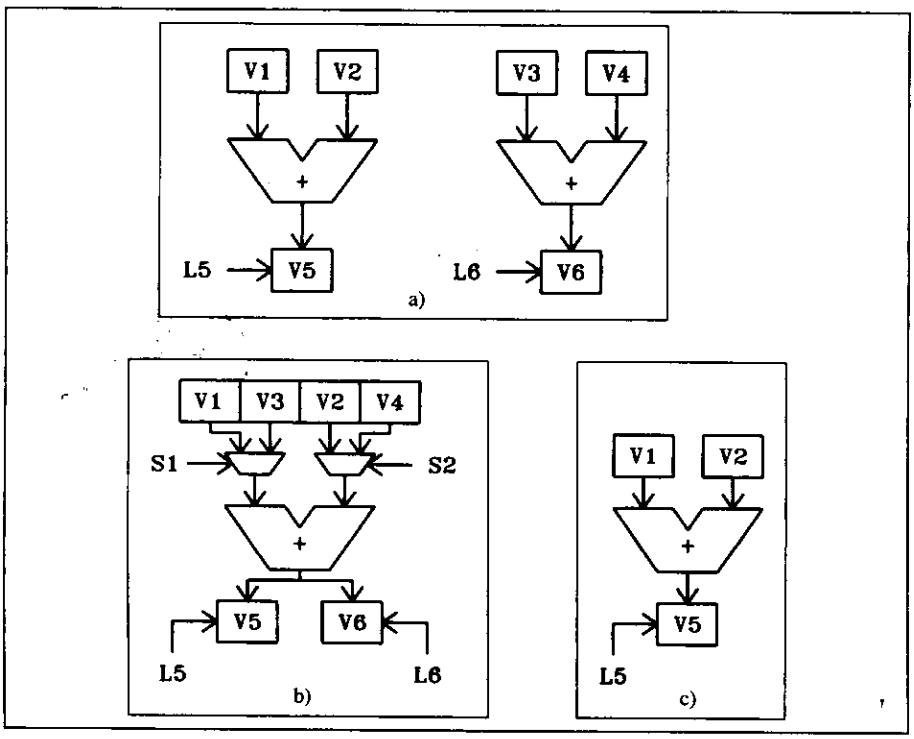


图12-43　利益变化作为普遍操作数的数目的函数

功能部件的分配也可以通过PDCPA完成。兼容图的节点表示一个操作，当且仅当节点i和

节点j所代表的操作可以指派到同一个功能单元时，节点i和节点j之间存在一条边。显然，根据公有操作数个数的不同所得到的好处也不同。表12-6给出了8个子图，分别表示8个利益类，其中第8类的利益最大，第1类的利益最小。该表选自Tseng和Siewiorek在1986年发表的论文。这里仅对类别制定级别。实际上在特定设计环境中制定的级别可能不同。因此，级别用来帮助阐明解决方法。这里允许把两个不同的操作数映射到相同的功能单元。ALU是这种构件的一个实例。

表12-6 功能单元分配利益类的子图

G_8	两种操作和三对变量都是相同的
G_7	操作不同但三对变量是相同的
G_6	操作和两对变量相同，但第三对变量不同
G_5	两对变量相同，但操作不同，并且第三对变量不同
G_4	两种操作和一对变量相同，但其他两对变量不同
G_3	一对变量相同，但两种操作不同，并且其他两对变量不同
G_2	操作相同，但所有的变量对都不相同
G_1	两种操作和三对变量都是不同的

为了解决这个问题，在应用CPA之后应用PDCPA。

获得的方案可以通过使用一个监测可交换操作间变量交换可能性的预处理步骤得以改进。通常，通过倒转操作数的顺序，先前不兼容而现在可以共享的源会变得明显。例如下面的例子：

```
s2: V2=V1+V3
s3: V4=V5+V1
```

如果两个上面给出等式的加法操作被同一个加法器执行，结果的电路如图12-44a。但是如果操作满足可交换性，语句则变为：

```
s2: V2=V1+V3
s3: V4=V1+V5
```

图12-44给出了减少一个多路器的实现结果。

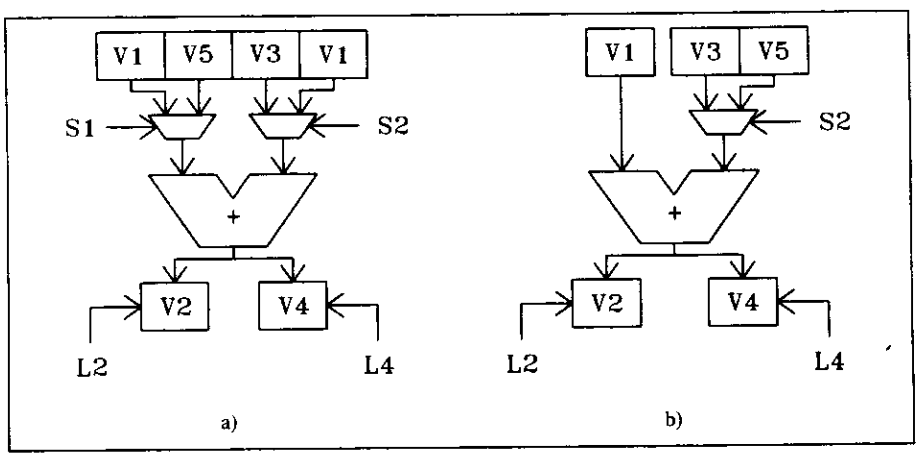


图12-44 通过倒转可交换操作数可以导出多路器的存储数

把图12-42的精简调度作为一个例子考虑。每个操作数随机赋值一个整数，如图12-45。在应用PDCPA之前，将检查可交换的操作。这个例子里，可交换操作有+、*、“and”及“or”。对于“and”操作，V3是右操作数，但是在其他控制步骤里可以指派到相同功能单元的“and”操作，V3是左操作数。因此，决定预先把“and”操作的操作数改变位置。这一变化已经表现在图12-45中。

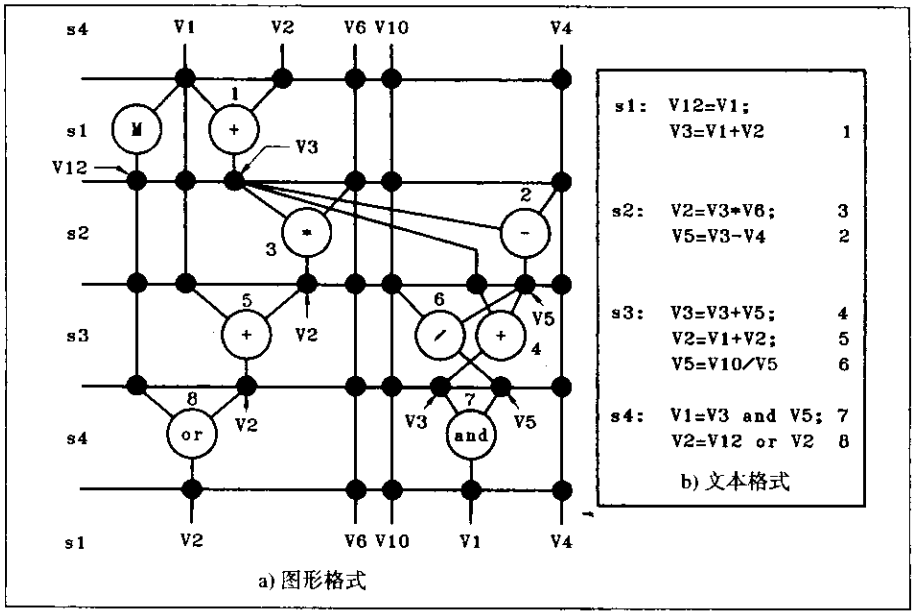


图12-45　为功能单元分配赋整数值

纯数据传输并没有赋值为一个整数，这是因为它不需要功能单元，而只需一条数据路径。在某些系统里，这种假设并不有效。例如，可能使用一条已经存在的通过ALU的路径实现纯数据传输。在那种情况下可以为纯数据传输赋一个整数值。

假设ALU可以作为功能单元被使用，而且ALU可以执行除纯数据传输以外的所有操作。图12-46给出了用于功能单元分配的兼容图G。它选自Tseng和Siewiorek在1986年发表的论文。只有在同一控制步骤发生的操作不能够兼容。圆括号外面的数表示每条边的类别。例如，(1, 4) 4表示操作1和操作4兼容（即，它们发生在不同控制步骤），并且所在的利益类为G_4，即这两个操作相同（都是加法），并且操作数对都是公有的（输出都是V3）。

```
(1,2)1  (1,3)1  (1,4)4  (1,5)6  (1,6)1  (1,7)1  (1,8)3
                (2,4)3  (2,5)1  (2,6)3  (2,7)3  (2,8)1
                (3,4)3  (3,5)3  (3,6)1  (3,7)3  (3,8)3
                                        (4,7)5  (4,8)1
                                        (5,7)1  (5,8)5
                                        (6,7)3  (6,8)1
```

图12-46　功能单元分配的兼容图（copyright © 1986 IEEE）

子图G_8和G_7都为空。子图G_6只包括一条边（1，5）。对G和G_6执行了PDCPA的结果是把节点1和节点5结合在一起。在精简图里，边（1，3）和（1，8）相应升级为类3和类5。图12-47

给出了精简图，节点1代表簇（1，5）。

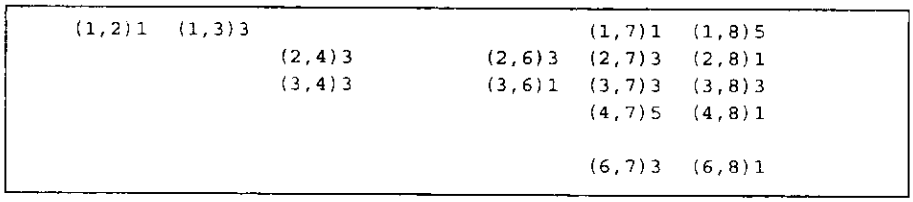


图12-47 处理G_6之后的精简兼容图

同样，精简后的图对子图G_5和G_3分别应用PDCPA，得到如下划分：

```
{{1, 3, 5, 8}, {2, 4, 7}, {6}}
```

图12-48给出了包含这种划分的ALU配置。ALU1必须执行三个不同功能（+，*，“or”）。与每个ALU的连接在ALU的输入和输出处用标号表示。例如，ALU1必须连接到三个寄存器（V1、V3和V12）。

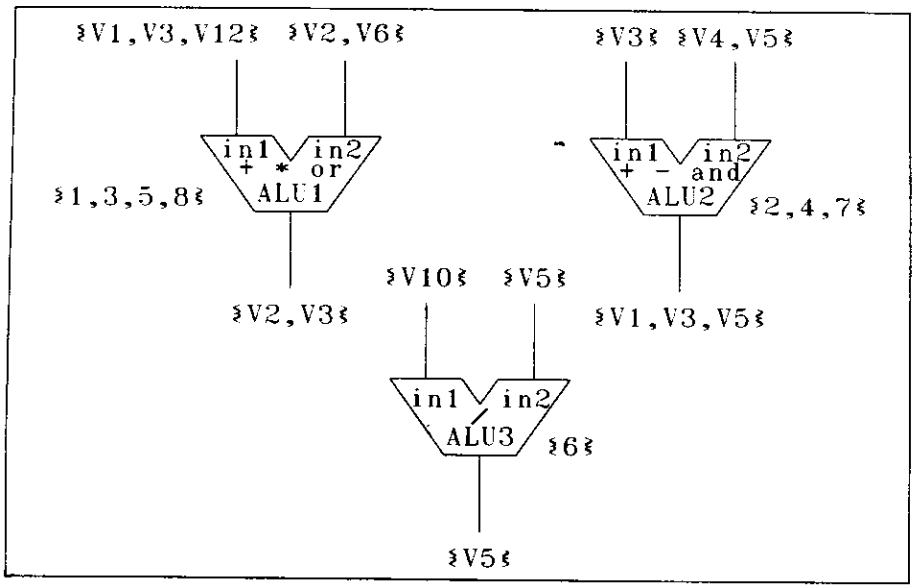


图12-48 由功能单元分配获得的ALU配置

3. PDCPA在数据路径分配中的应用

互连变量不能同时在一个数据路径上传输。可以使用PDCPA把互连变量绑定到数据路径上。需要的总线驱动器和接受器的数目被用来产生类别。

把两个具有不同源、目的的互连变量结合起来所获得的好处比较少，这是因为需要的驱动器的数目比较多。但是，如果两个互连变量具有相同的源，它们可以共享一个总线驱动器或多路复用器。类似地，如果两个互连变量具有相同的目的，它们可以共享相同的终端线和控制信号。

每个节点代表一个互连变量，当且仅当两个变量i和j没有同时传输时，在节点i和节点j之间具有一条边，这样可以获得一个兼容图。同样，一个构造图的简单办法是从互连变量节点

的完全图开始。通过检查调度图或调度的文本表示，不断删除数据同时传输的边。

为了获得更多的利益，把连接划分为用表12-7的子图定义的两个类。

表12-7　数据通路分配利益类的子图

G_2	连接共享一个公用源和公用目标
G_1	既不连接共享一个公用源也不共享一个公用目标

PDCPA可以用来找出互连变量到总线的近似最小簇。

例如，考虑图12-42中的精简调度及上一节获得的ALU分配：

```
ALU1: {1,3,5,8}
ALU2: {2,4,7}
ALU3: {6}
```

图12-49和图12-50把互连变量置不同的索引值。这个例子的图在Tseng和Siewiorek在1986年发表的论文里可以找到。

Source Name	Destination Name	Index
V1	V12	1
V1	ALU1.in1	2
V2	ALU1.in2	3
V3	ALU1.in1	4
V3	ALU2.in1	5
V4	ALU2.in2	6
V5	ALU2.in2	7
V5	ALU2.in2	8
V6	ALU1.in2	9
V10	ALU3.in1	10
V12	ALU1.in1	11
ALU1.out	V2	12
ALU1.out	V3	13
ALU2.out	V1	14
ALU2.out	V3	15
ALU2.out	V5	16
ALU3.out	V5	17

图12-49　为互连赋索引值

使用图12-42的调度和图12-49的索引赋值可以得到12-51的兼容图。

对具有两个利益类的兼容图，使用利益制导簇划分算法可以得到如下划分。

({1，2，4，11}，{3，9}，{5}，{6，7，8}，{10}，{12}，{13，14，15，16}，{17})

图12-52给出了应用利益制导簇划分算法进行寄存器分配、功能单元分配和数据通路分配产生的多路器的框图。该多路器也可以根据需要由三态缓冲器所替换。

12.5　高层综合的发展动态

前面几节给出的算法对各个独立的任务进行综合时非常有效。但是，本章已经指出，各个综合任务之间并不是相互独立的。如何找到对所有任务都有效的解决方案？这个问题一直没有很好解决。

即使对于一个独立的任务，所有的算法都没有保证它产生的是一种优化方案，同样它也

无法保证，这是因为该任务属于NP-完全问题。因此，最好的期望是所有的算法都可以获得近似最好或比较好的结果，或者是每种算法可以解决普通问题中某些特定的情况。当前许多正在进行的研究都以减少综合时的计算量为目标。

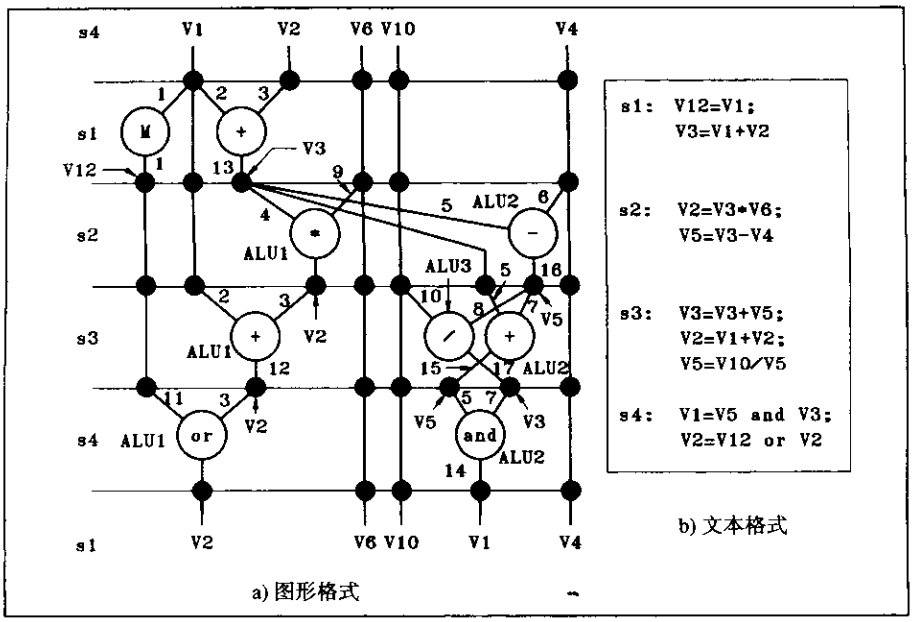


图12-50 为数据通路赋整数值

(1,2)2	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)
(1,9)	(1,10)	(1,11)	(1,12)	(1,14)	(1,15)
(1,16)	(1,17)	(2,4)2	(2,6)	(2,9)	(2,11)2
(2,14)	(2,16)	(3,4)	(3,6)	(3,9)2	(3,16)
(4,5)2	(4,7)	(4,8)	(4,10)	(4,11)2	(4,13)
(4,14)	(4,15)	(4,17)	(5,13)	(6,7)2	(6,8)
(6,10)	(6,11)	(6,13)	(6,14)	(6,15)	(6,17)
(7,8)2	(7,9)	(7,13)	(7,16)	(8,9)	(8,11)
(8,13)	(8,14)	(8,16)	(9,10)	(9,11)	(9,13)
(9,14)	(9,15)	(9,17)	(10,11)	(10,13)	(10,14)
(10,16)	(11,13)	(11,15)	(11,16)	(11,17)	(12,13)2
(13,14)	(13,15)2	(13,16)	(13,17)	(14,15)2	(14,16)2
(14,17)	(15,16)2	(16,17)2			

图12-51 互连单元分配的兼容图（copyright © 1986 IEEE）

这里有两种新的方法。第一种是使用专家知识进行指导来缩短过程。第二种是通过使问题范围变窄以利用更多的专业知识。这两种方法在数字信号处理领域、微处理器领域及流水线领域都取得了不同程度的成功。在综合方面仍然有很多需要解决的问题。

综合程序与人进行交互可能是非常必要的。但是，没有一个现有的程序允许与人交互，而且尚未研究开发出所需的或有效地与人进行交互的工作。

设计验证指的是证明设计结果真正满足给出的定义。它可以使用低级别的仿真程序仿真出最终设计结果的输出，通过与对设计定义的高层仿真的输出进行比较来进行验证。问题是何时停止仿真。要仿真多少数据？另一种验证方法是使用数学证明，从初始定义对每一步设

计进行验证。

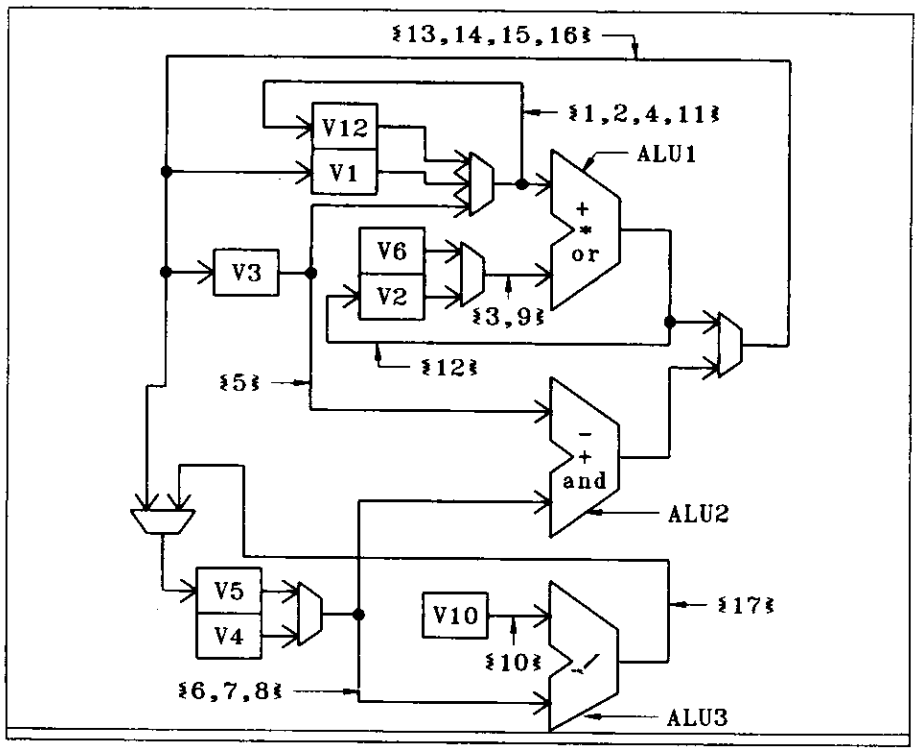


图12-52　使用多路器的数据通路图示例

设计层次的集成可以使得低级别的信息，如布局面积、功耗要求等可以在高级别做出决定时使用，但是这种技术还不成熟。例如，为了在算法级做出智能决定，必须知道一些有关低级别工具如何处理接受的模型的知识。

还有许多更专门的综合任务需要解决。接口设计、时序约束、调度和分配间的交互、不同分配任务间的交互及高级别行为对低级别设计的影响等，这些都是需要进一步研究的问题。

总之，高层综合作为一个抽象问题，包括调度和分配。我们已经很好地理解了这个问题，而且已经有多种方法支持对独立任务的综合。但是，当前还不存在一种解决整个任务的系统集成方法。

12.6　VHDL结构的自动综合

本节讨论把VHDL表示的构件自动翻译为硬件的可能性，着重讨论翻译一些独立的应用。本书的其他章节讨论从特定编程风格到特定硬件集合的翻译。

12.6.1　包含选择的构件

很多VHDL构件都包含从特定集合选择特定的元素。*case*语句包含从特定*case*语句集合里选择其中的一种情况。*if...then...else*语句包含从条件优先级列表中选择优先权最高且满足TRUE的条件。同样，一个向量中的元素可以通过指定索引值进行选择。所有这些语句都包括选择，并且可以用多路复用器硬件实现。图12-53是几个把VHDL构件映射到多路器的例子。

假设该实体被称作MUX，它表示多路器。使用实体MUX的目的是为了提供一种定义三个独立过程的环境。MUX有BIT类型的输入X和Y，BIT_VECTOR类型的输入VECT，枚举数据类型ENUM的输入CHOICE及整数数据类型的输入INDEX。MUX还有三个独立输出Z1、Z2和Z3。每个过程（MUX1、MUX2和MUX3）都具有某种形式的选择，并且都可以用多路器部件来实现。这三个过程在下面三节里详细讨论。

```
package TYPES is
  attribute ENCODING: STRING;
  type ENUM is (A, B, C, D);
  attribute ENCODING of ENUM: type is "00 01 10 11";
end TYPES;
  ------------------------------------------------------------
use work.TYPES.all;
entity MUX is
  port (X, Y: in BIT;  VECT: in BIT_VECTOR(3 downto 0);
        CHOICE: in ENUM; INDEX: in INTEGER range 3 downto 0;
        Z1, Z2, Z3: out BIT);
end MUX;
  ------------------------------------------------------------
architecture MUX_CONSTRUCTS of MUX is
begin
MUX1: process (CHOICE, X, Y)
  begin
    case CHOICE is
      when A => Z1 <= X;
      when B => Z1 <= Y;
      when C => Z1 <= not X;
      when D => Z1 <= not Y;
    end case;
  end process MUX1;
  ------------------------------------------------------------
MUX2: process (X, Y, VECT)
  begin
    if X = '1' then
      Z2 <= VECT(3);
    elsif Y = '1' then
      Z2 <= VECT(2);
    else
      Z2 <= VECT(1) and VECT(0);
    end if;
  end process MUX2;
  ------------------------------------------------------------
MUX3: process (VECT, INDEX)
  begin
    Z3 <= VECT(INDEX);
  end process MUX3;
end MUX_CONSTRUCTS;
```

图12-53 映射到多路器元件的VHDL构造

12.6.2 case语句对多路器的映射

图12-53的进程MUX1是用多路器实现*case*语句的例子。*case*语句根据枚举数据类型的候选值选择其中的一个并把它赋值给Z1。在逻辑电路里，枚举数据类型的元素必须用二进制向量表示。当CHOICE表示有限状态机的状态时，把二进制数据赋值给状态称作状态赋值问题。

在自动系统中，可能允许程序的赋值。但是并没有一种找到最佳赋值的有效算法。所以必须允许设计者定义枚举数据类型中每个元素的二进制值。一种定义赋值的方法是利用属性的概念。在包TYPES里，我们为STRING类型声明属性ENCODING。然后使用如下语句把该属性关联到ENUM类型：

```
attribute ENCODING of ENUM: type is "00 01 10 11"
```

属性ENCODING可以直接用自动工具为ENUM类型的元素置二进制编码值，如表12-8。

表12-8　通过属性ENCODING把二进制代码复制给ENUM

ENUM元素	二进制值
A	00
B	01
C	10
D	11

使用这种信息，自动设计工具可以把进程MUX1翻译成图12-54的硬件电路。在这个图里，信号对（CHOICE_1、CHOICE_0）表示输入CHOICE元素的二进制编码。这种二进制编码连接到多路器的地址输入，并且选择把合适的数据输入传递到输出Z1。例如，MUX1的VHDL代码，当CHOICE=(00)时，Z1=X。这个例子说明*case*语句可以被直接翻译到多路器。CHOICE表达式定义了MUX的地址输入。赋给Z1的数据值等于MUX的数据输入值。

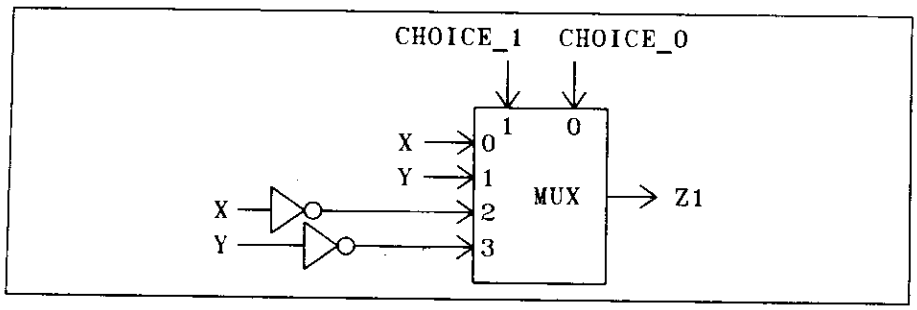


图12-54　进程MUX1的硬件实现

12.6.3　if...then...else语句对多路器的映射

*if...then...else*语句的VHDL构件同样包括从几个被选行为中进行选择。因此，可以用多路器实现*if...then...else*语句。图12-53的进程MUX2是这种构件的一个例子，它包括输入X、Y和VECT。通过扫描*if...then...else*语句，自动设计工具可以产生表12-9中的信息。

表12-9　*if...then...else*语句包含的信息

X	Y	Z2
0	0	VECT(1)和VECT(0)
0	1	VECT(2)
1	0	VECT(3)
1	1	VECT(3)

由于第一条*if*语句的值为TRUE，选择执行该语句。可以有多于一条的*if*条件为TRUE。例如，如果X=Y=1，则两条语句都为TRUE。但是，在这种情况下，Z2赋值给VECT(3)，这是因为语句（*if* X='1'）出现在语句（*elsif* Y='1'）之前。

前面的表的构建过程如下。语句（*if* X='1' *then* Z2<=VECT(3)）表示混合输入XY=10和XY=11的表项的值都为VECT(3)。语句（*elsif* Y='1' *then* Z2<=VECT(2)）表示混合输入XY=01的空白表项应被填写为VECT(2)。因为混合输入XY=11的表项已经填满，所以*elsif*语句对它不产生作用，即使这种组合的Y='1'。语句（*else* Z2<=VECT(1) and VECT(0)）把所有没有填充的项都指派到函数（VECT(1)和VECT(0)）。这个例子里，唯一没有填充的项是混合输入XY=00。

该表可以直接用图12-55的多路器实现，其中信号X和Y与乘法器的地址输入相连，每个XY混合数据的输入都由表项定义。

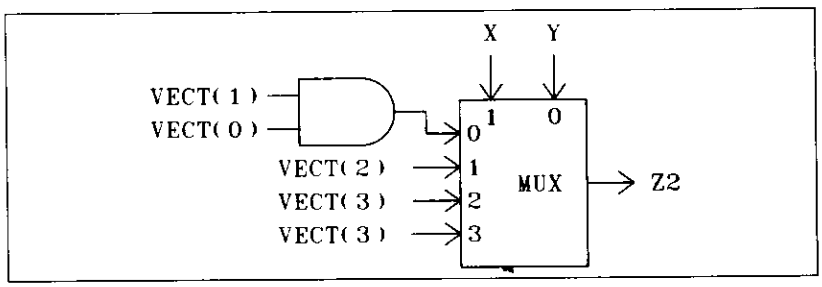


图12-55 进程MUX2的硬件实现

12.6.4 带下标向量引用对多路器的映射

如果VECT是一个向量，并且INDEX是一个整数，则变量赋值具有如下形式：

```
Z<=VECT(INDEX)
```

它也是一种选择行为。在这种情况，向量VECT中的一个元素被选择赋值给变量Z。被选择的特定元素由变量INDEX的值指定。这种语句同样可以直接映射到多路器。图12-53的进程MUX3是这种选择行为的一个例子。变量Z3被赋值到向量VECT的一个元素。一种自动化设计工具可以把这段VHDL构件映射到图12-56所示的硬件电路。本例中，由于向量VECT有4个元素，下标值的范围从3到0（参见实体MUX的VECT声明），自动翻译器需要把变量INDEX转换为2位二进制表示。在没有属性指定的指定代码段里，自动翻译器可以直接用二进制数表示整数值。这些代码直接形成图12-56所示电路。

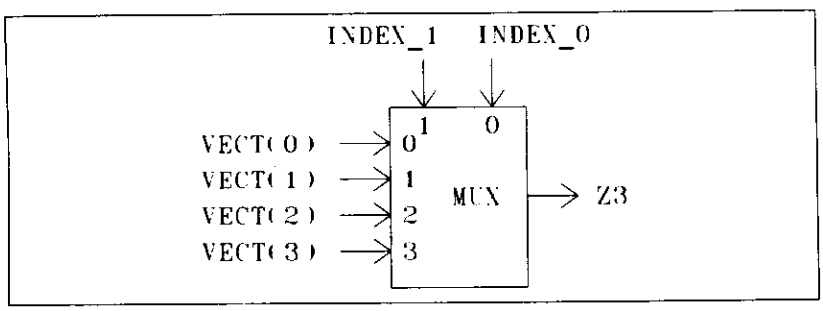


图12-56 进程MUX3的硬件实现

12.6.5 循环结构

第10章讲述的一般算法模型建议不使用循环，除非有明显的硬件可以实现。在本节里讨论一种用迭代电路实现的程序循环。

首先讨论在第1章最后引入的EXCLUSIVE-OR电路的变体。考虑图12-57声明的实体XOR4。实体XOR4的输入是一个4位向量A，它的下标值范围从3到1。它的输出是BIT类型的变量X。我们期望X等于EXCLUSIVE-OR的输入向量A的四位输入。

```
-----------------------------------------------
-- Entity declaration for 4-bit XOR circuit --
-----------------------------------------------
entity XOR4 is
  port (A: in BIT_VECTOR (3 downto 0);
        X: out BIT);
end XOR4;
-------------------------------------------------
-- Standard loop definition for circuit behavior.--
-------------------------------------------------
architecture XOR4_LOOP of XOR4 is
begin
  process (A)
    variable X_INT: BIT;
  begin
    X_INT := '0';
    for I in 0 to 3 loop
      X_INT := X_INT xor A(I);
    end loop;
    X <= X_INT;
  end process;
end XOR4_LOOP;
```

图12-57 带简单程序循环的VHDL程序

构架XOR_LOOP使用一个简单的循环结构描述了这种关系。内部变量X_INT在进程内声明，它初始化为‘0’。在第一遍循环（I=0）里，X_INT被EXCLUSIVE-OR的A(0)替代，A(0)为常数‘0’，它产生的结果等于A(0)。在下一个循环（I=1），X_INT被EXCLUSIVE-OR的当前值X_INT替代（A(0)替代为A(1)）。X_INT的新值是(A(0) xor A(1))。每次循环增加了A的一个元素，直到四次循环后，X_INT的值变为：

X_INT=A(0) xor A(1) xor A(2) xor A(3)

最后，X_INT的值赋给信号X。

这种简单循环可以用图12-58的移位寄存器时序电路实现。它假设构架中的进程XOR_LOOP嵌入在一个没有显示的更大的系统中。向量A存储在移位寄存器里。存储过程并没有被定义。它还使用了一个触发器来实现内部变量X_INT。该触发器被初始化为‘0’。位向量A在寄存器里移位，所以每次循环需要的位用D0表示。四次移位后，FF输出（X），得到期望的结果。这里没有包括控制移位操作的定时信号产生源。

通过对构架XOR4_LOOP应用空间-时间变换可以获得一个迭代的组合逻辑电路。图12-57中赋值到时间序列的内部变量X_INT的值被索引值为4到0的5位空间向量所替换。空间向量元素的值由图12-59所示的修改后的循环进行赋值。元素X_INT(0)初始化为‘0’，它对应最初循

环中变量X_INT的初始值。在循环内部，xor操作的每次结果都赋值到空间向量X_INT的不同元素。

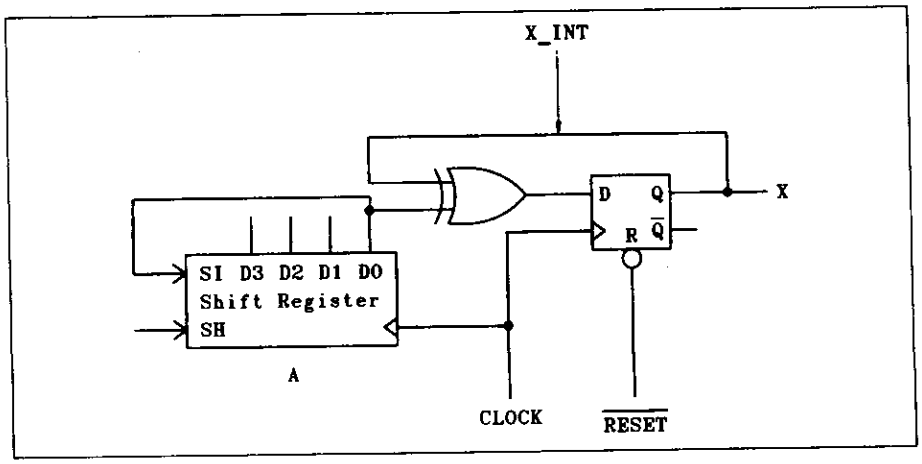


图12-58 实体XOR4的构架XOR4_LOOP的硬件实现

可以对构架XOR_LOOP的任何循环做类似的空间-时间变换。这种变换可以很容易地自动用算法描述。

```
-------------------------------------------
-- Entity declaration for 4-bit XOR circuit--
-------------------------------------------
entity XOR4 is
  port (A: in BIT_VECTOR (3 downto 0);
        X: out BIT);
end XOR4;
-------------------------------------------------------------
-- Space_Time transformation of standard loop definition.--
-------------------------------------------------------------
architecture XOR4_SPACE of XOR4 is
begin
  process (A)
    variable X_INT: BIT_VECTOR (4 downto 0);
  begin
    X_INT(0) := '0';
    for I in 0 to 3 loop
      X_INT(I+1) := X_INT(I) xor A(I);
    end loop;
    X <= X_INT(4);
  end process;
end XOR4_SPACE;
```

图12-59 对构架XOR4_LOOP应用时空转换获得的VHDL程序

构架XOR_LOOP可以直接映射到图12-60中的迭代组合电路。向量X_INT的后续值由单个的XOR门产生。电路可以很自然地与构架XOR4_SPACE的循环计算相对应。

这个*xor*例子代表了一大类迭代组合电路。类的所有成员都可以用基于移位寄存器的时序电路或迭代组合逻辑电路来实现。这些电路都具有相同的结构，只除了它们所执行的函数远远比简单的*xor*函数复杂得多。

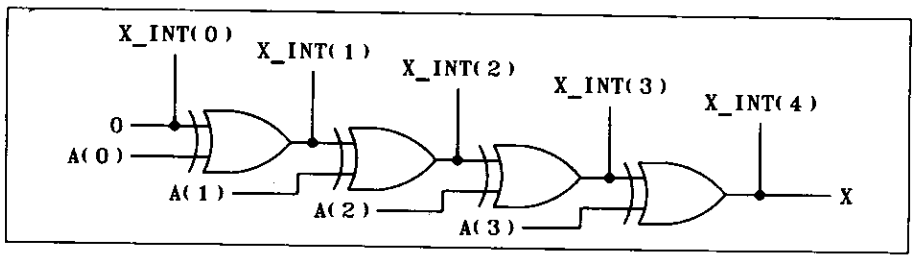


图12-60 实体XOR4的构架XOR4_SPACE的硬件实现

下面用经典的循环进位加法器解释一般的情况。图12-61给出了一段简单循环的VHDL代码。该段代码可以被直接翻译为基于移位寄存器的时序电路，如图12-62。这里没有包含控制信号及寄存器A和B的数据源。假设构架LOOP_ADDER中的进程嵌入在一个未给出的更大的系统内。输入向量A和B的值每次一位地被移入全加器（FA）电路中。FA输出产生的和每次一位地被存储到移位寄存器S中。内部变量CARRY的值存储在触发器中，被下一次循环使用。这一结果称为*串行加法器*。

```
-------------------------------------------------
-- Entity declaration for a 4-bit binary adder --
-------------------------------------------------
entity ADD4 is
  port (A,B: in BIT_VECTOR (3 downto 0);
        CIN: in BIT;
        S: out BIT_VECTOR (3 downto 0);
        COUT: out BIT);
end ADD4;
-------------------------------------------------------
-- Typical definition of adder using a program loop --
-------------------------------------------------------
architecture LOOP_ADDER of ADD4 is
begin
  process (A, B, CIN)
    variable CARRY: BIT := '0';
    variable SUM: BIT_VECTOR (3 downto 0);
  begin
    CARRY := CIN;
    for I in 0 to 3 loop
      SUM(I) := A(I) xor B(I) xor CARRY;
      CARRY := (A(I) and B(I)) or (A(I) and CARRY) or
               (B(I) and CARRY);
    end loop;
    S <= SUM;
    COUT <= CARRY;
  end process;
end LOOP_ADDER;
```

图12-61 循环进位加法器的VHDL程序

对图12-61的VHDL代码应用空间-时间转换可以得到图12-63所示的代码。其中构架LOOP_ADER的内部变量CARRY被构架SPACE_ADDER中具有范围从4到1的索引值的向量CARRY所替换。原始循环的逻辑等式（用符号FA表示）用来为后续输出和后续CARRY向量的元素赋值。CARRY触发器被消除。构架SPACE_ADDER中新的循环直接映射到图12-64的

迭代组合逻辑电路。

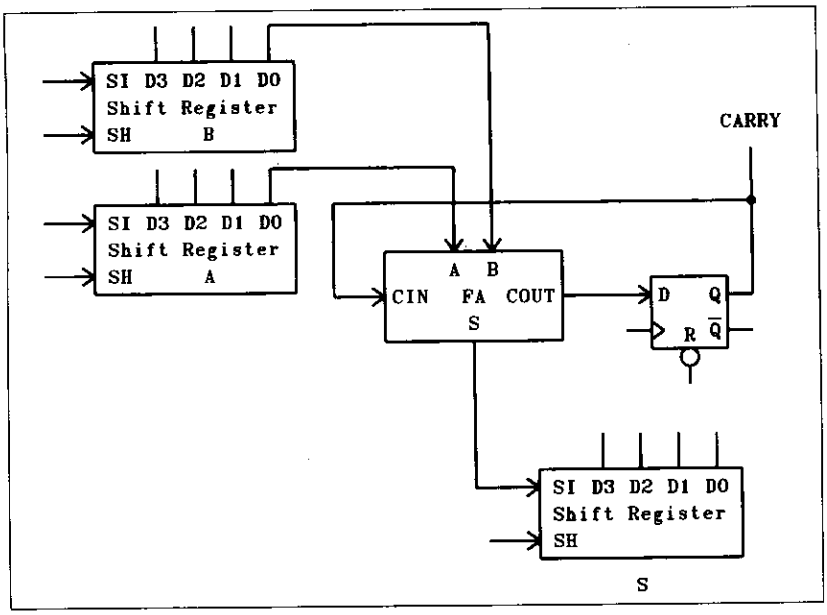


图12-62 实体ADD4的构架LOOP_ADDER的硬件实现

```
-------------------------------------------------------------
-- Applying space-time transformation to LOOP_ADDER-------
-- architecture --
-------------------------------------------------------------
architecture SPACE_ADDER of ADD4 is
begin
  process (A, B, CIN)
    variable CARRY: BIT_VECTOR (4 downto 0) := "00000";
    variable SUM: BIT_VECTOR (3 downto 0);
  begin
    CARRY(0) := CIN;
    for I in 0 to 3 loop
      SUM(I) := A(I) xor B(I) xor CARRY(I);
      CARRY(I+1) := (A(I) and B(I)) or (A(I) and CARRY(I))
                    or (B(I) and CARRY(I));
    end loop;
    S <= SUM;
    COUT <= CARRY(4);
  end process;
end SPACE_ADDER;
```

图12-63 对构架LOOP_ADDER应用时空转换的结果

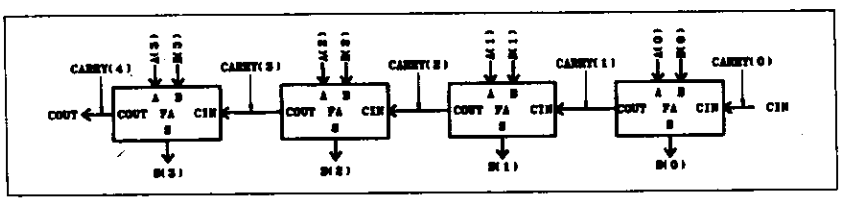


图12-64 实体ADD4的构架SPACE_ADDER的硬件实现

12.6.6 函数和过程

本节阐述函数和过程到硬件的可能映射。

图12-65显示了实体ADD4的构架FUNCTION_ADDER，它的VHDL代码通过替换构架SPACE_ADDER中全加器函数声明里的逻辑等式获得。FA的每个输出都是一个独立的声明。SUM(I)的逻辑等式声明为函数FA_S；CARRY(I+1)的逻辑等式声明为函数FA_C。在程序循环体内，对SUM(I)和CARRY(I+1)的赋值由函数调用完成。由于在算法里只是改变了符号，整个算法的操作并没有发生变化，所以构架FUNCTION_ADDER可以映射到与构架SPACE_ADDER相同的硬件。

```
-------------------------------------------
-- Use of functions to define the adder --
-------------------------------------------
architecture FUNCTION_ADDER of ADD4 is
  -- Function to compute the sum output
  function FA_S (AIN, BIN, CIN: BIT) return BIT is
  begin
    return AIN xor BIN xor CIN;
  end FA_S;
  -- Function to compute the carry output
  function FA_C (AIN, BIN, CIN: BIT) return BIT is
  begin
    return (AIN and BIN) or (AIN and CIN) or (BIN and CIN);
  end FA_C;
begin
  process (A, B, CIN)
    variable CARRY: BIT_VECTOR (4 downto 0) := "00000";
    variable SUM: BIT_VECTOR (3 downto 0) := "0000";
  begin
    CARRY(0) := CIN;
    for I in 0 to 3 loop
      SUM(I) := FA_S(A(I), B(I), CARRY(I));
      CARRY(I+1) := FA_C(A(I), B(I), CARRY(I));
    end loop;
    S <= SUM;
    COUT <= CARRY(4);
  end process;
end FUNCTION_ADDER;
```

图12-65 使用函数表示组合逻辑

一个基本的结论是，函数应该映射到组合逻辑电路。

同样，图12-66给出了用过程实现的进位加法器的VHDL描述。首先，过程FA定义了全加器。然后用程序循环的过程调用表示迭代全加器的连接。这个例子表明过程是一种方便而且简单的主要编程方法。通常，使用过程的VHDL代码能够映射到等价于没有使用过程的代码所使用的硬件上。过程和函数之间的主要区别是函数总是映射到逻辑电路，而过程可以映射到时序电路。上面是把过程映射到组合逻辑电路的例子。本书中还有另外一些例子说明了不同的过程定义。

```
-----------------------------------------
-- Use of a procedure to define the adder. --
-----------------------------------------
architecture PROCEDURE_ADDER of ADD4 is
  procedure FA(AIN, BIN, CIN: in BIT;
               SOUT, COUT: out BIT) is
  begin
    SOUT := AIN xor BIN xor CIN;
    COUT := (AIN and BIN) or (AIN and CIN) or
            (BIN and CIN);
  end FA;
begin
  process (A, B, CIN)
    variable CARRY: BIT_VECTOR (4 downto 0) := "00000";
    variable SUM: BIT_VECTOR (3 downto 0) := "0000";
  begin
    CARRY(0) := CIN;
    for I in 0 to 3 loop
      FA(A(I), B(I), CARRY(I), SUM(I), CARRY(I+1));
    end loop;
    S <= SUM;
    COUT <= CARRY(4);
  end process;
end PROCEDURE_ADDER;
```

图12-66 使用过程表示组合逻辑

习题

12.1 为图12-1中的FSM VHDL模型增加一个开始信号和一个完成信号，以使该器件更有用。仿真模型结果来证明操作的正确性。

12.2 考虑下面的VHDL代码：

```
-----------------------------------------------------------
  -- Entity declaration --
-----------------------------------------------------------
entity SCHED1 is
  port (A, B, C, D, E, F: in INTEGER;
        CLK: in BIT;
        X, Y: out INTEGER);
end SCHED1;
-----------------------------------------------------------
  -- Achitecture declaration --
  -----------------------------------------------------------
architecture HIGH_LEVEL of SCHED1 is
begin
    X <= F*(A+B+C*D+D*E);
    Y <= (A*B+E)*D*C;
end HIGH_LEVEL;
```

下面的任务都使用了上面的代码。假设没有硬件约束。

a) 画出数据流图。

b) 导出ASAP调度。

c) 导出ALAP调度。

d) 使用关键路径优先机制导出一个调度列。

e) 使用自由调度法导出一个调度。

12.3 使用下面的硬件约束重复习题12.2：

a) 硬件限制使用一个加法器和一个乘法器。

b) 硬件限制使用一个加法器和两个乘法器。

c) 硬件限制使用两个加法器和一个乘法器。

d) 硬件限制使用两个加法器和两个乘法器。

12.4 习题12.2或习题12.3的调度是否可以通过应用基本的数学转换加以改进？特别是C*D的积对X和Y都是通用的，同样也存在其他的简化。

12.5 考虑下面的VHDL代码：

```
-------------------------------------------------------
  -- Entity declaration
-------------------------------------------------------
entity SCHED2 is
  port (A, B, C, D, E, F: in INTEGER;
        CLK: in BIT;
        X, Y: out INTEGER);
end SCHED2;
-------------------------------------------------------
  -- Architecture declaration
-------------------------------------------------------
architecture HIGH_LEVEL of SCHED2 is
  signal Z: INTEGER;
begin
  X <= (A-B)*Z;
  Y <= (A*B)+Z;
  Z <= C*D + (E+F)/D;
end HIGH_LEVEL;
```

下列问题是关于以上VHDL代码的。假定没有硬件约束。

a) 画出数据流图。

b) 导出ASAP调度。

c) 导出ALAP调度。

d) 使用关键路径优先机制导出一个调度列。

e) 使用自由调度法导出一个调度。

12.6 使用下面的硬件约束重复习题12.5。

a) 硬件约束使用一个加法器、一个减法器、一个乘法器和一个除法器。

b) 假设加法和减法可以在同一个硬件模块（ADD_S）上执行，并且乘法和除法可以在一个单独模块（MD）上执行。同样，假设每种类型只有一个模块。

c) 假设加法和减法可以在同一个硬件模块（ADD_S）上执行，并且乘法和除法可以在一个单独模块（MD）上执行。同样，假设每种类型有两个模块。

d) 假设所有操作在一个ALU模块上执行，并假设有两个ALU模块可用。

12.7 对图12-9的MUX增加地址控制信号，对寄存器增加载入信号，并且使用图12-11的FSM构架来设计器件FSMEX1的控制单元。使用第8章讲述的方法。

12.8 考虑由习题12.2获得的最优调度，使用简单贪心算法来分配寄存器、功能单元和数据路径。

a) 给出一个类似图12-5的寄存器和功能单元的分配图。

b) 根据类似图12-6的硬件构造图说明数据路径的分配。

c) 构造Gantt图并讨论利用率。

d) 为控制单元产生一个类似图12-11的FSM模型并进行仿真。

e) 对MUX增加地址控制信号，对寄存器增加载入信号。使用FSM结构来设计控制单元。使用第8章的方法。

12.9 使用下面问题所得的最好调度，重复习题12.8，比较并对比这些调度：

a) 习题12.3(a)

b) 习题12.3(b)

c) 习题12.3(c)

d) 习题12.3(d)

e) 习题12.5(a)

f) 习题12.6(a)

g) 习题12.6(b)

h) 习题12.6(c)

i) 习题12.6(d)

12.10 对图12-67的数据生命周期图使用左边界算法获得一个有效的寄存器分配。

12.11 对图12-68的数据生命周期图使用左边界算法获得一个有效的寄存器分配。

12.12 对图12-69的数据生命周期图使用左边界算法获得一个有效的寄存器分配。

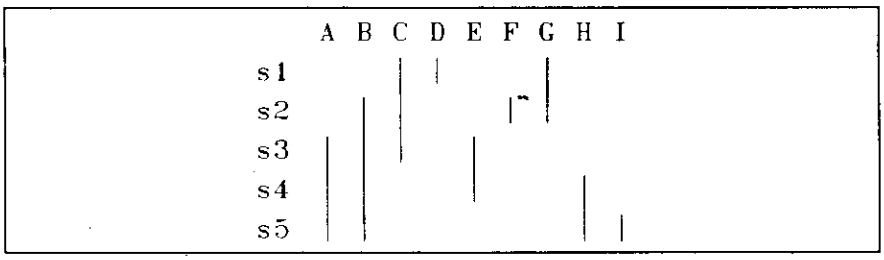


图12-67 习题12.10的数据生命周期图

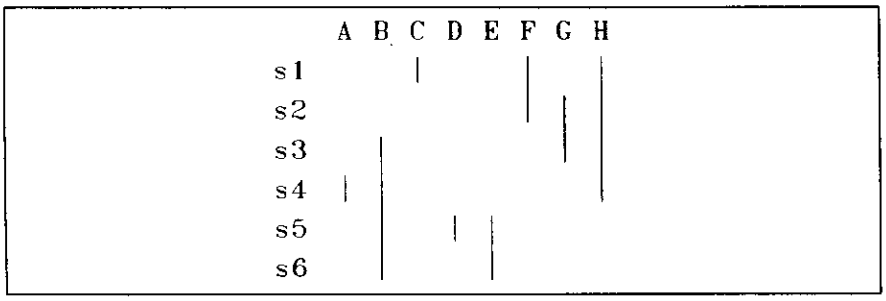


图12-68 习题12.11的数据生命周期图

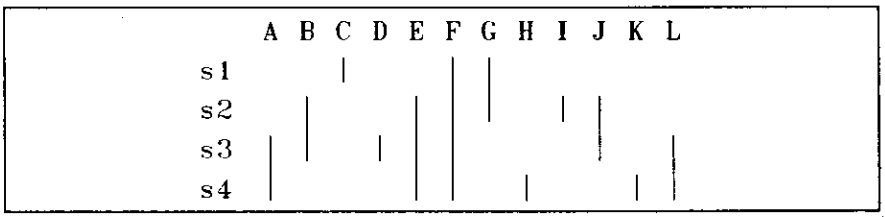


图12-69 习题12.12的数据生命周期图

12.13 对图12-70的数据生命周期图使用左边界算法获得一个有效的寄存器分配。

12.14 对图12-71的数据生命周期图使用左边界算法获得一个有效的寄存器分配。

12.15 对图12-72的数据生命周期图使用左边界算法获得一个有效的寄存器分配。

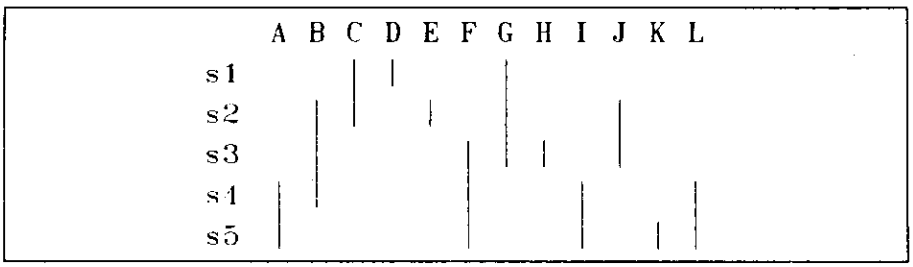


图12-70 习题12.13的数据生命周期图

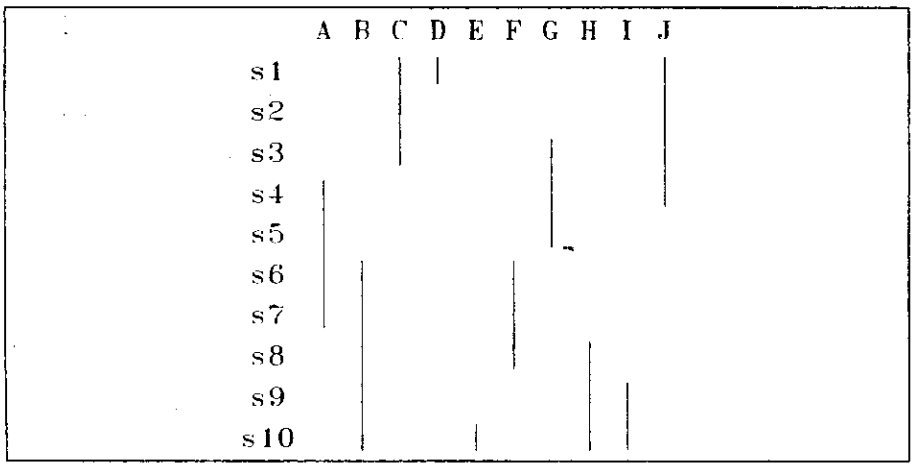


图12-71 习题12.14的数据生命周期图

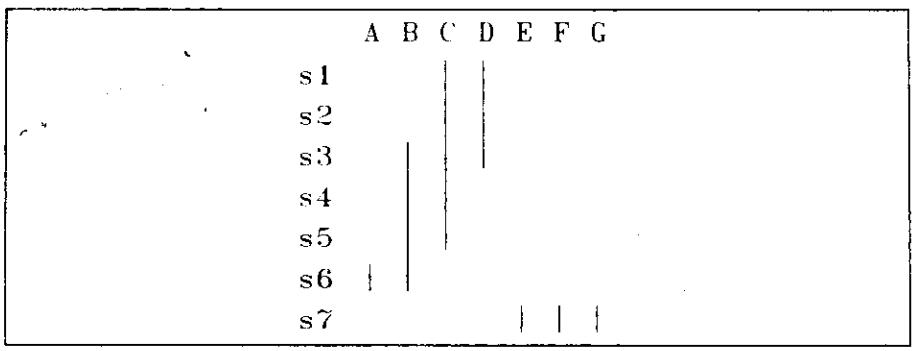


图12-72 习题12.15的数据生命周期图

12.16 对于图12-38给出的调度，使用左边界算法获得一个有效的寄存器分配，把此结果与图12-42给出的用图形算法PACPA产生的结果进行比较。左边界算法是否提供了用PDCPA产生的所有信息？若没有，是否可以修改左边界算法以得出缺少的信息？

12.17 考虑由习题12.2获得的最优调度，使用左边界算法分配寄存器，并且使用互连图方法

（见图12-29）分配功能单元和数据路径。

a) 画出一个类似图12-29给出的分配图。

b) 构造一个类似图12-30的硬件图。

c) 构造一个Gantt图并讨论利用率。

d) 对控制单元产生和模拟一个类似图12-11的FSM模型。

e) 对MUX增加地址控制信号，对寄存器增加载入信号，并增加其他硬件图中所需的控制信号。使用上面产生的FSM构架系设计控制单元。使用第8章的方法。

12.18 使用下面问题获得的最好调度，重复习题12.17：

a) 习题12.3(a)

b) 习题12.3(b)

c) 习题12.3(c)

d) 习题12.3(d)

e) 习题12.5(a)

f) 习题12.6(a)

g) 习题12.3(b)

h) 习题12.3(c)

i) 习题12.3(d)。

12.19 对图12-73给出的兼容图，执行簇划分算法（CPA）决定图中节点的近似最小簇划分。

12.20 对图12-74给出的兼容图，执行簇划分算法（CPA）决定图中节点的近似最小簇划分。

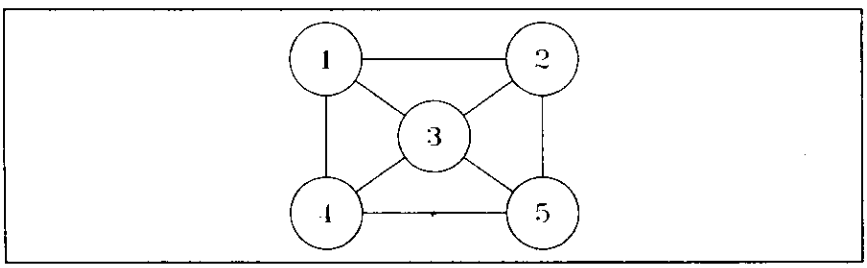


图12-73 习题12.19的兼容图

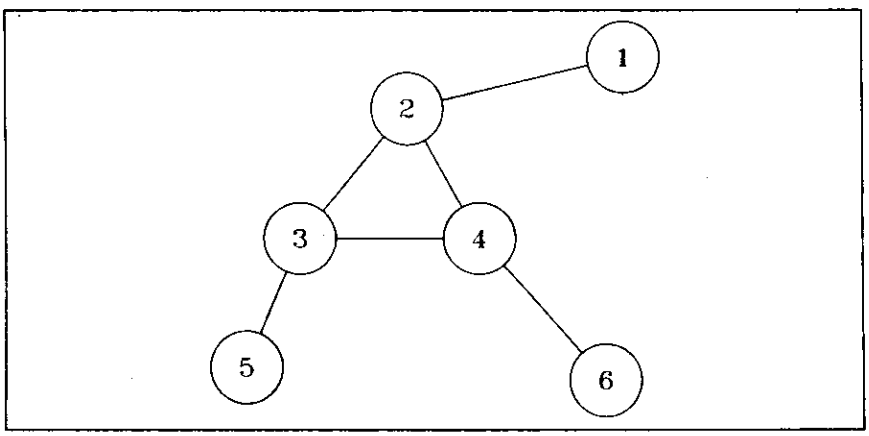


图12-74 习题12.20的兼容图

12.21　对图12-75给出的兼容图，执行簇划分算法（CPA）决定图中节点的近似最小簇划分。

12.22　对图12-76给出的兼容图，执行簇划分算法（CPA）决定图中节点的近似最小簇划分。

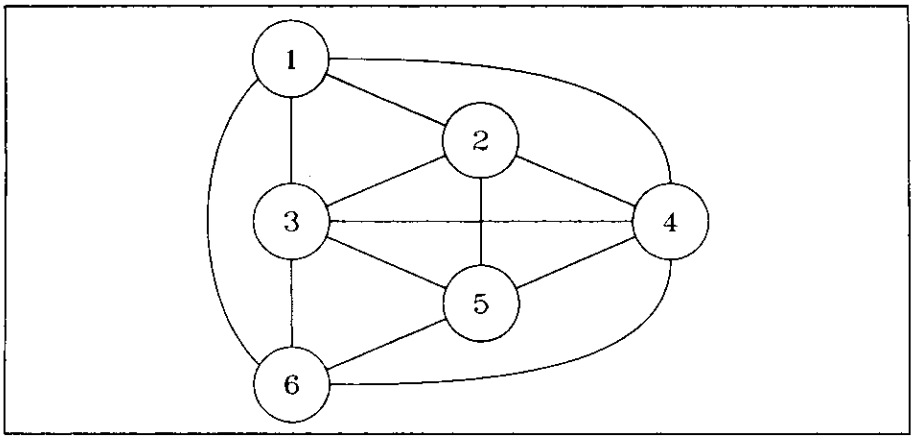


图12-75　习题12.21的兼容图

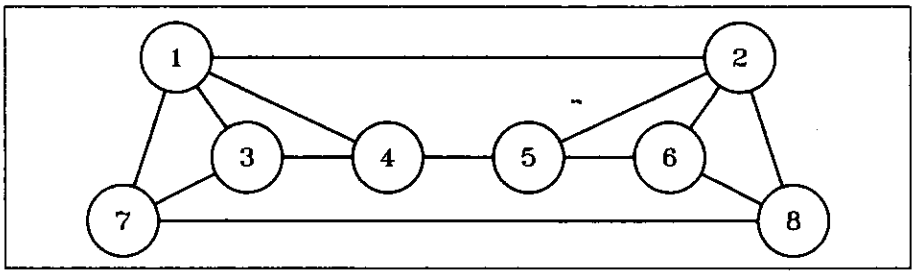


图12-76　习题12.22的兼容图

12.23　对图12-77给出的兼容图，执行簇划分算法（CPA）决定图中节点的近似最小簇划分。

12.24　对图12-78给出的兼容图，执行簇划分算法（CPA）决定图中节点的近似最小簇划分。

12.25　对图12-20的调度列使用利益制导簇划分算法（PDCPA）来分配寄存器、功能单元和数据路径，比较PDCPA获得的结果及使用左边界算法和互连图方法获得的结果。

12.26　为习题12.25设计的器件设计一个控制单元。首先产生一个类似于图12-11中的VHDL模型并对其仿真，使用第8章的方法来设计这个控制单元，讨论设计自动化的可能性。

12.27　考虑习题12.2获得的最优调度，使用利益制导簇划分算法（PDCPA）分配寄存器、功能单元和数据路径。

a) 画出类似于图12-50的图，并用文字说明寄存器和功能单元的分配。

b) 构造一个类似图12-52的硬件图并用文字说明数据路径的分配。

c) 构造Gantt图并讨论利用率。

d) 对控制单元产生类似图12-11的FSM模型并对其仿真。

e) 对上面产生的硬件图中的MUX增加地址控制信号，对功能单元增加功能选择信号，并且对寄存器增加载入信号。使用上面产生的FSM构架设计这个控制单元。使用第8章的方法。

12.28　使用下列问题中获得的最优调度，重复习题12.27：

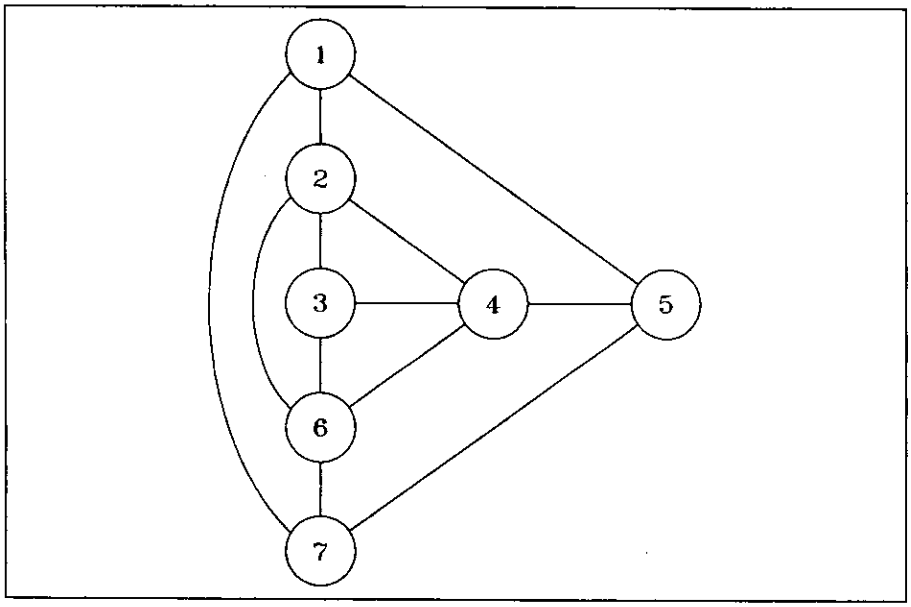


图12-77 习题12.23的兼容图

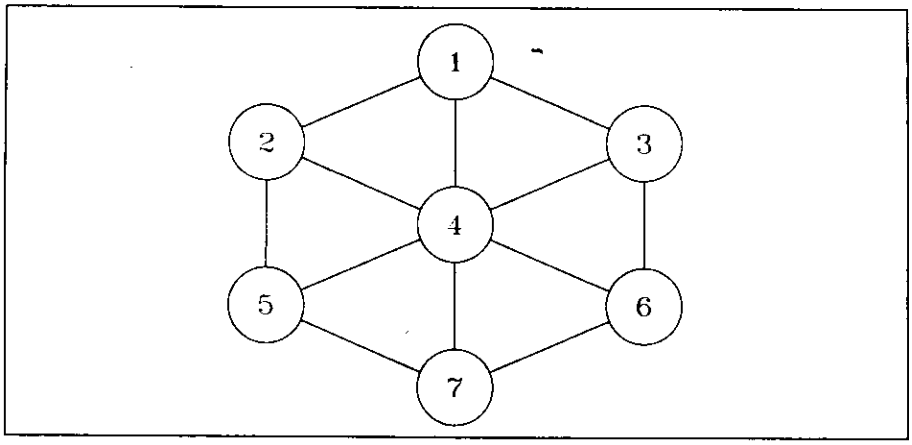


图12-78 习题12.24的兼容图

a) 习题3(a)
b) 习题3(b)
c) 习题3(c)
d) 习题3(d)
e) 习题5
f) 习题6(a)
g) 习题3(b)
h) 习题3(c)
i) 习题3(d)。

12.29 使用利益制导簇划分算法（PDCPA）分配调度图12-79中的寄存器、功能单元和数据路径。假设每个功能单元都是可联系的，而且ALU功能单元可用并可执行所有函数。

a) 画出类似图12-50的图，并用文字说明寄存器和功能单元的分配。

b) 构造一个类似图12-52的硬件图，并用文字说明数据路径的分配。

c) 构造Gantt图并讨论利用率。

d) 对控制单元产生类似图12-11的FSM模型并对其仿真。

f) 对上面产生的硬件图中的MUX增加地址控制信号，对功能单元增加功能选择信号，并且对寄存器增加载入信号。使用上面产生的FSM构架设计这个控制单元。使用第8章的方法。

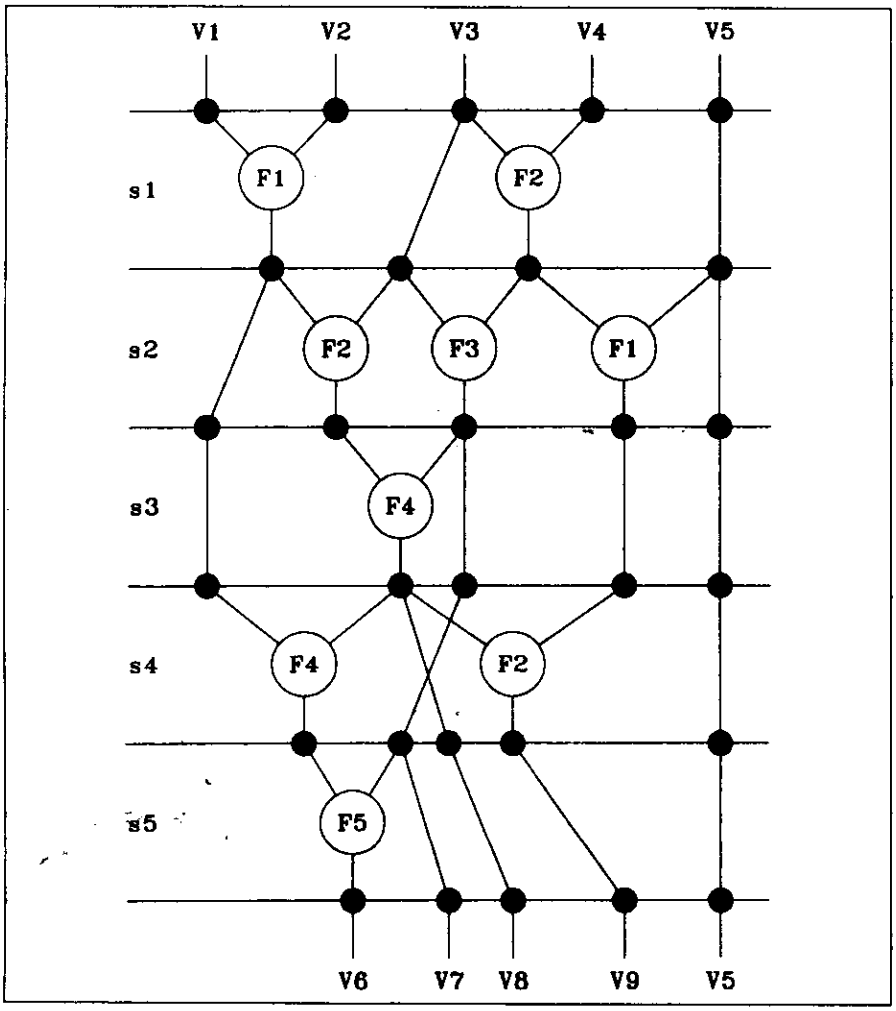


图12-79 典型调度

12.30 设计一个优先级分析器件，假设有8个器件竞争一个数据总线。当器件需要访问数据总线时则产生一个逻辑1的访问请求信号。“优先级分析”器件的输入是8个请求信号，它产生8个应答输出信号，每个对应一个竞争器件。任何时刻只有一个响应输出为逻辑1，其他的输出都是逻辑0。活跃输出对应的竞争器件在当前访问请求中具有最高优先级。把8个请求信号当作8位向量REQ（7~0）。REQ（7）具有最高的优先级，REQ（0）优先级最低。把8个响应输出当作8位向量ACK（7到0），若REQ=00101100，则ACK为00100000。

a) 使用过程循环写出“优先级分析”程序的算法级VHDL描述，对其仿真并验证输入-输出行为的正确性。

b) 把算法级VHDL描述翻译为硬件实现，用文档说明使用的翻译过程。

c) 使用硬件仿真程序仿真该硬件实现。比较这种仿真的行为与算法级VHDL仿真的行为。

12.31 为类似上题中的优先级分析程序设计一个算法级VHDL描述，区别在于使用类属以使该模块可以适用于大于等于2个输入。

12.32 设计一个接收2个8位二进制输入向量A（7~0）和B（7~0）的二进制比较器，它产生下面的三个二进制输出，输入数据均为正整数。

AGTB：当且仅当A严格大于B时信号为逻辑1。

AEQB：当且仅当A等于B时信号为逻辑1。

ALTB：当且仅当A严格小于B时信号为逻辑1。

a) 使用一个过程循环写出比较器的算法及VHDL描述，对其仿真验证行为的正确性。

b) 把算法级VHDL描述翻译为硬件实现，用文档说明使用的翻译过程。

c) 使用硬件仿真程序仿真该硬件实现。比较这种仿真的行为与算法级VHDL仿真的行为。

12.33 对上题描述的比较器（COMPARATOR）的算法级VHDL描述增加类属，使其可以处理任意大小的输入向量。

12.34 设计一个接收2个8位二进制输入向量A（7~0）和B（7~0）的比较器COMPARATOR2，它产生下面的三个二进制输出，输入数据均为正整数。

AGTB：当且仅当A严格大于B时信号为逻辑1。

AEQB：当且仅当A等于B时信号为逻辑1。

ALTB：当且仅当A严格小于B时信号为逻辑1。

a) 使用一个过程循环写出比较器的算法及VHDL描述，对其仿真验证行为的正确性。

b) 把算法级VHDL描述翻译为硬件实现，用文档说明使用的翻译过程。

c) 使用硬件仿真程序仿真该硬件实现。比较这种仿真的行为与算法级VHDL仿真的行为。

12.35 对上题描述的比较器（COMPARATOR2）的算法级VHDL描述增加类属，使其可以处理任意大小的输入向量。

12.36 下面是一个VHDL模型的代码，ADD8函数执行8位加法，INC8函数执行8位加1操作。

a) 用文字解释模型的功能。

b) 画出描述所表达的硬件原理图，此图应属于寄存器抽象级。

```
use work.funcs.all;
entity SYS is
  port(C: in BIT; COM: BIT_VECTOR(0 to 1);
       INP: in BIT_VECTOR(0 to 7));
end SYS;
architecture CODE of SYS is
  signal X,Y: BIT_VECTOR(0 to 7);
begin
  process(C)
  begin
    if C='1' then
      case COM is
        when "00" => X <= INP;
        when "01" => Y <= INP;
```

```
        when "10" => X <= ADD8(X,Y);
        when  "11" => X <= ADD8(X,INC8(not(Y)));
      end case;
    end if;
  end process;
end CODE;
```

12.37 把下面的VHDL代码综合为硬件，使用框图表示，每个框图都定义了功能、输入和输出。

```
use work.PRIMS.all;
entity TEST_QUESTION is
  port(CLK,W,X,Y,Z: in BIT; O: out BIT);
end TEST_QUESTION;
architecture BEHAV of TEST_QUESTION is
  signal  C: BIT_VECTOR(0 to 1);
begin
  BLK: block(not CLK'STABLE and CLK='1')
  begin
    C <= guarded INC(C);  --INC is an increment function
  end block BLK;
  O <= W when C="00" else
       X when C="01" else
       Y when C="10" else
       Z;
end BEHAV;
```

参考文献

1. Acock, S.J.B, Dimond, K.R., "Automatic mapping of algorithms onto multiple FPGA-SRAM modules," *Field-programmable Logic and Applications. 7th International Workshop, FPL '97 Proceedings*. Springer-Verlag, Berlin, Germany, 1997, pp. 255–64.
2. Airiau, R., Berge, J., Olive, V., *Circuit Synthesis with VHDL*, Kluwer Academic Publishers, Boston, MA, 1994.
3. Arato, P., Visegrady, T., "Effective graph generation from VHDL descriptions," *Microelectronics Journal*. vol. 29, no. 3, March 1998, pp. 113–21.
4. Armstrong, J. R., Gray, F. G., *Structured Logic Design With VHDL*, Prentice Hall PTR, Englewood Cliffs, N.J., 1993.
5. Armstrong, J. R., "Process-level modeling with VHDL," *Proceedings International Verilog HDL Conference and VHDL International Users Forum* (Cat. No.98TB100230), IEEE Computer Society Press, Los Alamitos, CA, USA, 1998, p.72–6.
6. Armstrong, J., Cho, C., Shah, S., Kosaraju, C. "The VHDL validation suite," *Proceedings-27th ACM/IEEE Design Automation Conference* (Cat. #90CH2894-4), IEEE, Piscataway, NJ, 1990, pp. 2–7.
7. Armstrong, J. R., *Chip-Level Modeling with VHDL*, Prentice Hall PTR, Englewood Cliffs, NJ, 1989.
8. Armstrong, J. R., "The use of the process model graph in defining the structure of behavioral models," *SIGDA Newsletter*, vol. 18, no. 4, pp. 65–70.
9. Armstrong, J. R., Gray, F. G., Lin, M. W., "VHDL modeling and model testing for DSP applications," *IEEE Transactions on Industrial Electronics*, vol. 46, no. 1, February 1999, pp. 13–22.
10. Armstrong, J. R., Burnette, D. G., "Automated assists to the behavioral modeling process," *First International Workshop on Rapid System Prototyping. Shortening the Path from Specification to Prototype* (Cat. #91TH0380-6), IEEE, Piscataway, NJ, pp. 187–95.
11. Armstrong, J. R., Burnette, D. G., "A systematic approach to chip level modeling with VHDL," *Wescon/89 Conference Record*, Electronic Conventions Management, Ventura, CA, pp. 333–338.
12. Armstrong, J. R., Gray, F. G., Lin, M. W., "VHDL modeling and model testing for DSP applications," *IEEE Transactions on Industrial Electronics*, vol. 46, no. 1, February 1999, pp. 13–22.
13. Ashenden, P., *The Designer's Guide to VHDL*, Morgan Kaufman Publishers, Inc., San Francisco, 1996.
14. Ashenden, P., *The Student's Guide to VHDL*, Morgan Kaufman Publishers, Inc., San Francisco, 1998.
15. Ashenden, P., "Modeling digital systems using VHDL," *IEEE-Potentials*. vol. 17, no. 2, April–May 1998, pp. 27–30.
16. Austin, S. M., "Automated translation of ASIC designs," *1998 IEEE AUTOTESTCON Proceedings, IEEE Systems Readiness Technology Conference. Test Technology for the 21st Century* (Cat. No.98CH36179), IEEE, New York, NY, USA, 1998, p. 667.
17. Barton, D. L., "Behavioral descriptions in VHDL," *VLSI SystemsDesign*, vol. 9, no. 6, pp. 28–31, 33.
18. Baumgartner, K. M., *Computer Scheduling Algorithms: Past, Present and Future*, Elsevier Science Publishing Co., Inc., 1991.

19. Berenyi, A., et al, "Continuously live image processor for drift chamber track segment triggering," *IEEE Transactions on Nuclear Science*, vol. 46, no. 3, pt.1, June 1999, pp. 348–53.
20. Berge, J., Fonkura, A., Maginot, S., Rouillard, J., *VHDL 92: The New Features of the VHDL Language*, Kluwer Academic Publishers, Boston, MA, 1993.
21. Berge, J., Fonkura, A., Maginot, S., Rouillard, J., *VHDL Designer's Reference*, Kluwer Academic Publishers, Boston, MA, 1992.
22. Bhasker, J., "Process-graph analyzer: a front-end tool for VHDL behavioral synthesis," *Software-Practice and Experience*, vol. 18, no. 5, pp. 469–83.
23. Bhasker, J., *A Guide to VHDL Syntax*, Prentice Hall, Englewood Cliffs, NJ, 1995.
24. Bhasker, J., *A VHDL Primer*, Prentice Hall, Englewood Cliffs, NJ, 1995.
25. Bhasker, J., *A VHDL Synthesis Primer*, Star Galaxy Publishing, Allentown, PA, 1996.
26. Bonk, J., Stone, A., Manolakos, E. S., "Synthesis of array architectures for block matching motion estimation," *1999 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings, ICASSP99*, vol. 4 (Cat. No.99CH36258), IEEE, Piscataway, NJ, USA, 1999, pp. 1925–8.
27. Borriello, G., Detjens, E., "High-level synthesis: Current status and future directions," *Proceedings–25th Design Sutomation Conference*, pp. 477–82.
28. Brown, S. D., Francis, R. J., Rose J., Vranesic Z. G., *Field Programmable Gate Arrays*, Kluwer Academic Publishers, Boston, MA, 1992.
29. Buhler, M., Baitinger, U. G., "VHDL-based development of a 32-bit pipelined RISC processor for educational purposes," *9th Mediterranean Electrotechnical Conference Proceedings*, vol. 1 (Cat. No.98CH36056), IEEE, New York, NY, USA, 1998, pp. 138–42.
30. Buzzoni, M., Cardini, D., Gallino, R., Romagnese, R., "ATM traffic management systems: ASIC fast prototyping," *Proceedings Tenth IEEE International Workshop on Rapid System Prototyping. Shortening the Path from Specification to Prototype* (Cat. No.PR00246), IEEE Computer Society Press, Los Alamitos, CA, USA, 1999, pp. 74–80.
31. Campasono, R., Wolf, W., *High Level Synthesis*, Kluwer Academic Publishers, Boston, MA, 1991.
32. Camposano, R., Wolf, W., *High-Level VLSI Synthesis*, Kluwer Academic Publishers, Dordrecht, Netherlands, 1991.
33. Carter, J. W., *Digital Designing with Programmable Logic Devices*, Prentice Hall, Upper Saddle River, NJ, 1997.
34. Chang, K. C., *Digital Design and Modeling with VHDL and Synthesis*, IEEE Computer Society Press, Los Alamitos, CA, 1997.
35. Cleaver, C., Derr, M., "Design automation through synthesis of VHDL," *Design Automation*, vol. 2, no. 1, pp. 20–4.
36. Coelho, D. R., *The VHDL Handbook*, Kluwer Academic Publishers, Netherlands, 1989.
37. Compass Design Automation, *VHDL Scout*, Compass Design Automation, San Jose, CA, 1994.
38. Dalcolmo, J., Lauwereins, R., Ade, M., "Code generation of data dominated DSP applications for FPGA targets," *Proceedings. Ninth International Workshop on Rapid System Prototyping* (Cat. No.98TB100237), IEEE Computer Society Press, Los Alamitos, CA, USA, 1998, pp. 162–7.
39. DeMicheli, G., Ku, D. C., "HERCULES—A system for high-level synthesis," *Proceedings–25th Design Automation Conference*, 1988, pp. 483–8.
40. Devadas, S., Newton, A. R., "Algorithms for hardware allocation in data path synthesis," *IEEE Transactions on Computer-Aided Design*, vol. 8 (July 1989), pp. 768–81.
41. Dewey, A., *Analysis and Design of Digital Systems With VHDL*, PWS Publishing Co., Boston, MA, 1997.
42. Dinu A., Cirstea, M. N., McCormick, M., "Virtual prototyping of a digital neural current controller," *Proceedings Ninth International Workshop on Rapid System Prototyping* (Cat.

No.98TB100237), IEEE Computer Society Press, Los Alamitos, CA, USA, 1998, pp. 176–81.
43. Dipert, B., "Getting a handle on HDLs," *EDN-(US-Edition)*, vol. 43, no. 10, May 1998, pp. 71–2, 75–6, 79–80, 83–4, 86, 90.
44. Dutt, N. D., Gajski, D. D. "Designer controlled behavioral synthesis," *26th ACM/IEEE Design Automation Conference* (ACM #477890 and IEEE Cat. #89CH2734-2), ACM, New York, pp. 754–7.
45. Filippi, E., Licciardi, L., Montanaro, A., Paolini, M., Turolla, M., Taliercio, M., *Proceedings of the IEEE 1998 Custom Integrated Circuits Conference* (Cat. No.98CH36143), IEEE, New York, NY, USA, 1998, pp. 97–100.
46. Frank, G. A., Gray, F. G., Gopalakrishnan, S., Song, W., "Reuse of models and test benches at different levels of abstraction," *Proceedings International Verilog HDL Conference and VHDL International Users Forum* (Cat. No.98TB100230), IEEE Computer Society Press, Los Alamitos, CA, USA, 1998, pp. 130–7.
47. Frenkil, J., "The practical engineer [IC design, power reduction]," *IEEE-Spectrum*, vol. 35, no. 2, February 1998, pp. 54–60.
48. Gajski, D. D., Dutt, N. D., Wu, A., Lin, S. Y-L., *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, Dordrecht, Netherlands, 1992.
49. Gajski, D. D., Vahid F., Narayan S., Gong J., *Specification and Design of Embedded Systems*, Prentice Hall PTR, Englewood Cliffs, N.J., 1994.
50. Gajski, D. D., Dutt, N. D., Wu, A., Lin, S., *High Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, Boston, MA, 1992.
51. Garbergs, B., Sohlberg, B., "Implementation of a state space controller in a FPGA," *MELECON '98 9th Mediterranean Electrotechnical Conference Proceedings*, vol. 1 (Cat. No.98CH36056), IEEE, New York, NY, USA, 1998, pp. 566–9.
52. Gray, F. G., Frank, G. A., Ziegenbein, D., Vuppala, S., Balasubramanian, P., "Tools for rapid construction of VHDL performance models for DSP systems," *Proceedings International Verilog HDL Conference and VHDL International Users Forum* (Cat. No.98TB100230), IEEE Computer Society Press, Los Alamitos, CA, USA, 1998, pp. 77–82.
53. Hamblen, J., "A VHDL synthesis model of the MIPS processor for use in computer architecture laboratories," *IEEE-Transactions-on-Education*, vol. 40, no. 4, November 1997, pp. 10.
54. Harr, R. E., Stanculescu, A. G., *Applications of VHDL to Circuit Design*, Kluwer Academic Publishers, Dordrecht, Netherlands.
55. Hines, J., "Where VHDL fits within the CAD environment," *ConferenceProceedings-24th ACM/IEEE Design Automation Conference*, 1987, IEEE, Piscataway, NJ, pp. 491–4.
56. Hong, Y. S., Park, K. H., Kim, M., "Automatic synthesis of data paths based on the path-search algorithm," *IEEE Int. Conf. on Computer-Aided Design*, 1987, pp. 270–3.
57. *IEEE Standard VHDL Language Reference Manual*. IEEE, Piscataway, NJ, 1987.
58. *IEEE Standard VHDL Language Reference Manual*. IEEE, Piscataway, NJ, 1993.
59. Jenkins, J. H., *Designing With FPGAs and CPLDs*, Prentice Hall PTR, Englewood Cliffs, NJ, 1994.
60. Jong, C. C., Lam, Y. H., Ng, L. S., "FPGAs implementation of a digital IQ demodulator using VHDL, field-programmable logic and applications," *7th International Workshop, FPL '97 Proceedings*, Springer-Verlag, Berlin, Germany, 1997, pp. 410–17.
61. Kapusta, R., "Writing reusable VHDL," *Electronic-Product-Design*, vol. 19, no. 8, Aug. 1998, pp. 17–18, 20.
62. Kelly, M., Hsu, K. W., "A flexible pipelined image processor," *Proceedings Eleventh Annual IEEE International ASIC Conference* (Cat. No.98TH8372), IEEE, New York, NY, USA, 1998, pp. 325–32.

63. Kelly, M., Hsu, K. W., "VHDL implementation of an image processor," *Proceedings of the SPIE The International Society for Optical Engineering*, vol. 3422, 1998, pp. 120–31.
64. Kivioja, M., Isoaho, J., Vanska, L., "Design and implementation of Viterbi decoder with FPGAs," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 21, no. 1, May 1999, pp. 5–14.
65. Komanec, R., Vrba, R., "Considerable advantages of VHDL over classic design approach," *Proceedings, Vol. 1, AMSE*, Assoc. Advancement of Modeling & Simulation Tech. Enterprises, Tassin la Demi Lune, France, 1995, pp. 144–7.
66. Krup, P., Abbasi, T., *Logic Synthesis Using Synopsys*, Kluwer Academic Publishers, Boston, MA, 1997.
67. Kurdahi, F. J., Parker, A.C., "REAL: A program for register allocation," *24th ACM/IEEE Design Automation Conference*, 1987, pp. 210–5.
68. Kuusilinna, K., Hamaainen, T., Saarinen, J., "Field programmable gate array-based PCI interface for a coprocessor system," *Microprocessors-and-Microsystems*, vol. 22, no. 7, January 1999, pp. 373–88.
69. Leung, S. S., Shanblatt, M. *ASIC System Design with VHDL : A Paradigm*, Kluwer Academic Publishers, 1989.
70. Lin, M. W., Armstrong, J. R., Frank, G. A, Concha, L., "A functional test planning system for validation of DSP circuits modeled in VHDL," *Proceedings International Verilog HDL Conference and VHDL International Users Forum* (Cat. No.98TB100230), IEEE Computer Society Press, Los Alamitos, CA, USA, 1998, pp. 172–7.
71. Lin, M. W., Armstrong, J. R., Gray, F. G., "A goal tree based high-level test planning system for DSP real number models," *Proceedings International Test Conference 1998* (IEEE Cat. No. 98CH36270), Washington DC, USA, 1998, pp. 1000–9.
72. Lipman, J., "Covering your HDL chip-design bets," *EDN-(US-Edition)*, vol. 43, no. 22, October 1998, pp. 65–6, 68–70, 72, 74.
73. Lipsett, R., Schaefer, C., Ussery, C. *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, 1989.
74. Lis, J. S., Gajski, D. D. "Synthesis from VHDL," *Proceedings-1988 IEEE International Conference on Computer Design, VLSI Computing Process ICCD 88* (Cat. #88CH2643-5), IEEE, Piscataway, NJ, pp. 378–81.
75. Lis, J. S., Gajski, D. D. "VHDL synthesis using structured modeling," *Proceedings-Design Automation Conference* (Cat. #89CH2734-2), IEEE, Piscataway, NJ, pp. 606–9.
76. Mazor, S., Langstraat, P., *A Guide to VHDL*, Kluwer Academic Publishers, Dordrecht, Netherlands, 1992.
77. McCanny, J. V., Trainor, D., Hu, Y., Ding, T. J., "Rapid design of complex DSP cores," *ESS€IRC '97 Proceedings of the 23rd European Solid-State Circuits Conference*, Editions Frontieres, Paris, France, 1997, pp. 284–7.
78. McCloskey, J., "Application of VHDL to software radio technology," *Proceedings International Verilog HDL Conference and VHDL International Users Forum* (Cat. No.98TB100230), IEEE Computer Society Press, Los Alamitos, CA, USA, 1998, pp. 90–5.
79. McLeod, J. "'Top-down' design is the watchword at DAC," *Electronics*, vol. 63, no. 6, June 1990, pp. 78–80.
80. McLeod, J., "New kind of engineering fuels CAD," *Electronics*, vol. 63, no. 4, April 1990, pp. 50–3.
81. Menchini, P. J., "A minimalist approach to VHDL logic modeling," *IEEE Design & Test of Computers*, vol. 7, pp. 12–23.
82. Meyer, E., "VHDL opens the road to top-down design," *ComputerDesign*, vol. 28, pp. 57–62.
83. Meyer, E., "VHDL strives to cover both synthesis and modeling," *Computer Design*, vol. 28, no. 19, pp. 42–5.

84. Mora, F., Sebastia, A., Muller, H., Fernandes, C., Ermoline, Y., "Design of a high-performance PCI interface for an SCI network," *Computing & Control Engineering Journal*, vol. 9, no. 6, December 1998, pp. 275–82.
85. More, M., Vidal, J., Lecha, E., Rincon, F., Teres, L., "Experiences on VHDL based methodologies on industrial ASIC design," *1998 International Semiconductor Conference. CAS'98 Proceedings* vol. 1 (Cat. No.98TH8351), IEEE, New York, NY, USA, 1998, pp. 167–70.
86. Munch, M., When, N., Glesner, M., "An efficient ILP-based scheduling algorithm for control-dominated VHDL descriptions," *ACM Transactions on Design Automation of Electronic Systems*, vol. 2, no. 4, October 1997, pp. 344–64.
87. Narayan, S., Vahid, F., Gajski, D. D., "Translating system specifications to VHDL," *EDAC Proceedings of the European Conference on Design Automation*, 1991, IEEE Computer Society Press, Los Alamitos CA, pp. 390–4.
88. Navabi, Z. *VHDL: Analysis and Modeling of Digital Systems*, McGraw-Hill, Inc., USA, 1993.
89. Nebhrajani, V. A., Suthar, N., "Finite state machines: a deeper look into synthesis optimization for VHDL," *Proceedings Eleventh International Conference on VLSI Design* (Cat. No. 98TB100217), IEEE Computer Society Press, Los Alamitos, CA, USA, 1997, pp. 516–21.
90. O'Neill, M. D., January, D. D., Cho, C. H., Armstrong, J. R., "BTG: A behavioral test generator," *Computer Hardware Description Languages and Their Applications. Proceedings of the IFIP WG 10.2 Ninth InternationalSymposium*, North-Holland, Amsterdam, Netherlands, pp. 347–61.
91. Olcoz, S., Castellvi, A., Garcia, M., "Improving VHDL soft-cores reuse with software-like reviews and audit procedures," *Proceedings. International Verilog HDL Conference and VHDLInternational Users Forum* (Cat. No. 98TB100230), IEEE Computer Society Press, Los Alamitos, CA, USA, 1998, pp. 143–6.
92. Oldfield, J. V., Dorf, R. C., *Field Programmable Gate Arrays*, John Wiley and Sons, New York, NY, 1995.
93. Ott, D. E., Wilderotter, T. J., *A Designer's Guide to VHDL Synthesis*, Kluwer Academic Publishers, Boston, MA, 1994.
94. Park, N., Parker, A., "Sehwa: A software package for synthesis of pipelines from behavioral specifications," *IEEE Transactions on Computer-Aided Design* (March 1988), vol. 7, no. 3.
95. Paulin, P. G., Kight, J. P., "Algorithms for high-level synthesis," *IEEE Design and Test of Computers* (December 1989), pp. 18–31.
96. Perez, M. A. J, Luque, W. M., Damiani, F., "Biologically-inspired digital circuit for a self-organising neural network," *Proceedings of the 1998 Second IEEE International Caracas Conference onDevices, Circuits and Systems*, ICCDCS 98, On the 70th Anniversary of the MOSFET and 50th of the BJT (Cat. No.98TH8350), IEEE, New York, NY, USA, 1998, pp. 172–7.
97. Perry, D. L., *VHDL*, McGraw-Hill, Inc, New York, 1991.
98. Popp, R. L., Montana, D. J., Gassner, R. R. Vidaver, G., Iyer, S., "Automated hardware design using genetic programming," *VHDL and FPGAs, SMC'98 Conference Proceedings*, 1998 IEEE International Conference on Systems, Man, and Cybernetics, vol. 3 (Cat. No. 98CH36218), IEEE, New York, NY, USA, 1998, pp. 2184–9.
99. Renner, F. M., Becker, J., Glesner, M., "An FPGA implementation of a magnetic bearing controller for mechatronic applications," *Field-Programmable Logic and Applications. From FPGAs to Computing Paradigm. 8th International Workshop, FPL'98 Proceedings*, Springer-Verlag, Berlin, Germany, 1998, pp. 179–88.
100. Rosenstiel, W., Camposano, R. "Synthesizing circuits from behavioral level specifications," *Computer Hardware Description Languages and their Applications*, Elsevier Science Publishers B.V., North-Holland.
101. Ruiz, P. L., Riesgo, T., Uceda, J., "Design and prototyping of DSP custom circuits based on

a library of arithmetic components," *Proceedings of the IECON'97 23rd International Conference on Industrial Electronics, Control, and Instrumentation*, vol. 1 (Cat. No. 97CH36066), IEEE, NewYork, NY, USA, 1997, pp. 191–6.
102. Ryan, R., "X-state handling in VHDL," *Design Automation*, vol. 2, no. 4, pp. 32–5.
103. Salcic, Z. and Smailagic, A., *Digital Systems Design and Prototyping Using Field Programmable Logic*, Kluwer Academic Publishers, Boston, MA, 1997.
104. Sargunaraj, J. J., Rao, S. S., "An optimal implementation approach for discrete wavelet transform using FIR filter banks on FPGAs," *1998 International Semiconductor Conference. CAS'98 Proceedings*, vol. 1 (Cat. No. 98TH8351), IEEE, New York, NY, USA, 1998, pp. 167–70.
105. Schoen, J., *Performance and Fault Modeling with VHDL*, Prentice Hall, Englewood Cliffs, NJ, 1989.
106. Schroeter, J., *Surviving the ASIC Experience*, Prentice Hall, Englewood Cliffs, NJ, 1992.
107. Schutti, M., Pfaff, M., Hagelauer, R., "VHDL design of embedded processor cores: the industry-standard microcontroller 8051 and 68HC11," *Proceedings Eleventh Annual IEEE International ASIC Conference* (Cat. No.98TH8372), IEEE, New York, NY, USA, 1998, pp. 265–9.
108. Shaditalab, M., Bois, G., Sawan, M., "Self-sorting radix-2 FFT on FPGAs using parallel pipelined distributed arithmetic blocks," *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines* (Cat. No. 98TB100251), IEEE Computer Society Press, Los Alamitos, CA, USA, 1998, pp. 337–8.
109. Shahdad, M., "An overview of VHDL language and technology," *Proceedings-23rdACM/IEEE Design Automation Conference, 1986* (Cat. #86CH2288-9), IEEE Computer Society Press, Washington, D.C., pp. 320–6.
110. Shahdad, M., Lipsett, R., Marchschner, E., Sheehan, K., Cohen, H., Waxman, R., Ackley, D. "VHSIC hardware description language," *Computer*, vol. 18, no. 2, pp. 94–103.
111. Skahill, K., *VHDL for Programmable Logic*, Adddison-Wesley, Menlo Park, CA, 1996.
112. Slorach, C. G., Sharman, K. C., "A novel single chip evolutionary hardware design using FPGAs," *Proceedings of the SPIE -The International Society for Optical Engineering*, vol. 3526, 1998, pp. 114–23.
113. Smith, D. J., "To create successful designs, know your HDL simulation and synthesis issues," *EDN-Europe*, November 1997, pp. 135–6, 138, 140, 142, 144.
114. Smith, S. P., Larson, J. "A high performance VHDL simulator with integrated switch and primitive modeling," *Computer Hardware Description Languages and their Applications. Proceedings of the IFIPWG 10.2 Ninth International Symposium*. North-Holland, Amsterdam, Netherlands, pp. 299–313.
115. Smith, S. P., Acosta, R. D., "Value system for switch-level modeling," *IEEE Design & Test of Computers*, vol. 7, no. 3, June 1990, pp. 33–41.
116. Smith, S., Taylor, D, Benaissa, M., "Design automation of Reed-Solomon codecs using VHDL," *Microelectronics-Journal*, vol. 29, no. 12, December 1998, pp. 977–82.
117. Stone, A., Manolakos, E. S., "Using DG2VHDL to synthesize an FPGA implementation of the 1-D discrete wavelet transform," *1998 IEEE Workshop on Signal Processing Systems. SIPS 98. Design and Implementation* (Cat. No.98TH8374), IEEE, New York, NY, USA, 1998, pp. 489–98.
118. Sullivan, R., Asher, L. R., "VHDL for ASIC design and verification," *VLSI Systems Design, Semicustom Design Guide*, 1998, pp. 64–72.
119. Tan. S., Furber, S. B., Wen-Fang-Yen, "The design of an asynchronous VHDL synthesizer," *Proceeding. Design, Automation and Test in Europe* (Cat. No. 98EX123), IEEE Computer Society Press, Los Alamitos, CA, USA, 1998, pp. 44–51.
120. Thomas, D. E., Moorby, P. R. *The Verilog Hardware DescriptionLanguage*, Kluwer Academic Publishers, Dordrecht, Netherlands, 1991.

121. Trimberger, S., *Field-Programmable Gate Array Technology*, Kluwer Academic Publishers, Boston, MA, 1994.
122. Tseng, C.-J., Siewiorek, D. P., "Facet: A procedure for the automated synthesis of digital systems," *20th Design Automation Conference*, 1983, pp. 490–6.
123. Tseng, C.-J., Siewiorek, D. P., "Automated synthesis of data paths in digital systems," *IEEE Transactions on Computer-Aided Design* (July 1986), pp. 379–95.
124. Uht, A. K., Ying-Sun, "The laboratory environment of the URI Integrated Computer Engineering Design (ICED) curriculum," *FIE '98. 28th Annual Frontiers in Education Conference. Moving from 'Teacher-Centered' to 'Learner-Centered' Education. Conference Proceedings*, vol. 1 (Cat. No.98CH36214). IEEE, Piscataway, NJ, USA, 1998, pp. 331–6.
125. Van den Bout, D., *The Practical Xilinx Designer Lab Book*, Prentice Hall PTR, Upper Saddle River, NJ, 1999.
126. Vassileva, T., Tchoumatchenko, V., Shishkov, V., Guyot, A., "High performance adder's synthesis using efficient macro generator," *ECCTD '97 Proceedings of the 1997 European Conference on Circuit Theory and Design*, vol. 3, Tech. Univ. Budapest, Budapest, Hungary, 1997, pp. 1343–6.
127. Walker, P. A., Ghosh, S., "On the nature and inadequacies of transport timing delay constructs in VHDL descriptions," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 16, no. 8, August 1997, pp. 894–915.
128. Ward, P. C., Armstrong, J. R., "Behavioral fault simulation inVHDL," *Proceedings-27th ACM/IEEE Design Automation Conference,1990* (Cat. #90CH2894-4), IEEE, Piscataway, NJ, pp. 587–93.
129. Wicks, J. A., Armstrong, J. R., "Efficiency ratings for VHDL behavioral models," *Proceedings IEEE Southeastcon '98. 'Engineering for a New Era'* (Cat.No.98CH36170), IEEE, New York, NY, USA, pp. 401–4.
130. Wilsey, P. A., Martin, D. E., Subramani, K., "SAVANT/TyVIS/WARPED: components for the analysis and simulation of VHDL," *Proceedings International Verilog HDL Conference and VHDL International Users Forum* (Cat. No. 98TB100230), IEEE Computer Society Press, Los Alamitos, CA, USA, 1998, pp. 195–201.
131. Yin, T. H., Yuan, H. W., Jer, S. H., "Rapid prototyping of hardware/software codesign for embedded signal processing," *Journal of Information Science and Engineering*, vol. 4, no. 3, September 1998, pp. 605–32.
132. Zeidman, B., *Verilog Designer's Library*, Prentice Hall PTR, Upper Saddle River, NJ, 1997.
133. Zhang, D., Liu, M., "VHDL high level design of digital systems with HLS/BIT," *Journal of Computer Science and Technology (English Language Edition)*, vol. 13, supplemental issue, December 1998, pp. 82–8.