清华大学电子与信息技术系列教材

数字集成系统的结构化设计与高层次综合

王志华　邓仰东　编著

清华大学出版社

(京)新登字 158 号

内 容 简 介

采用结构化设计技术，可以有效地提高集成电路设计方案的可重复使用性；采用高层次综合技术，可以有效地提高数字系统的设计效率。本书讨论有效提高数字集成系统设计能力的结构化设计方法和高层次综合技术。重点讨论结构化设计思想在不同层次的应用，同时还包含了高层次综合的基本思想。前 8 章讨论数字系统的设计，第 9 章简单介绍了模拟与混合系统设计中的特殊问题。

本书可作为高等院校电子类高年级本科生与研究生的教材，也可以作为相关领域的工程技术人员的参考资料。

书　名：数字集成系统的结构化设计与高层次综合
作　者：王志华　邓仰东　编著
出版者：清华大学出版社(北京清华大学学研大厦，邮编 100084)
http：//www.tup.tsinghua.edu.cn
印刷者：北京市人民文学印刷厂
发行者：新华书店总店北京发行所
开　本：787×1092 1/16　**印张**：25.5　**字数**：582 千字
版　次：2000 年 7 月第 1 版　2000 年 7 月第 1 次印刷
书　号：ISBN 7-302-03837-6/TN・106
印　数：0001～3000
定　价：29.50 元

前 言

当前，集成电路产业的发展日新月异，1999 年世界集成电路的销售额达 1570 亿美元。以集成电路为基础的电子信息产品的世界贸易总额于 1998 年就超过了 8000 亿美元，成为世界第一大产业。据国际权威机构预测，到 2012 年世界集成电路的年销售额将达到 1 万亿美元，并支持 6 ～ 8 万亿美元的电子设备和 30 万亿美元的电子信息服务，相当于今天全世界 GNP 的总和。集成电路的发展加速了人类社会的信息化进程，已经成为信息产业乃至 21 世纪实现知识经济的技术基础之一。我国微电子产业的发展，经历了自主研发创业、引进吸收提高、重点建设三个阶段。1999 年我国集成电路的总消耗量为 436 亿元，其中国产芯片的总量约为 83.8 亿元，占芯片消耗总量的 19%。预计到 2010 年我国集成电路市场的容量将达到 6000 亿元，国产集成电路的产值将达到 2000 亿元。

与集成电路产业迅速发展的同时，微电子技术也在不断改进，电路性能迅速提高。根据国际权威机构估计，2010 年动态存储器(DRAM)的存取时间将降低至 10ns 以下；电源电压有可能降至 0.6V，数字电路的时钟频率可以提高到 3GHz。与此同时，集成度和芯片规模也在提高，工业化生产的典型硅集成电路的管芯面积将达 2.5cm×2.5cm，特征尺寸将下降为 0.07μm。对于以 DRAM 为特征的硅存储器，集成度将达到 8.5Gbit/cm^2；全定制设计的数字系统，集成度将达 25Mgate/cm^2；电子系统集成中常用的标准单元，集成度也将达到 10Mgate/cm^2。

随着微电子技术的进步，人们对集成系统的需求也在提高。计算机、通信、消费类电子产品、工业及军事等各种领域都大量需要集成电路，比如在现代战争中，指挥、通信、计算机与情报获取都必须以集成电路为核心。在军舰、战车、飞机、导弹和航天器中集成电路的成本分别占到总成本的 22%、24%、33%、45%和 66%。在这些领域大量需求的是集成系统，即将完整的电子系统或子系统集成在单个芯片内。单个芯片内可能集成了微控制器核心(MCU core)、数字信号处理器核心(DSP core)、存储器核心(memory core)、射频处理器(RF processor)、数模和模数转换器等嵌入式硬件以及完成特定功能所必须的嵌入式软件。

对集成系统需求的提高也对设计者提出了挑战。集成系统的规模通常在 1000 万个等效门以上，完成集成系统设计，除了设计时间很长之外，还要花费高额的一次性研制(nonrecurring engineering，NRE)费用，因此，保证首次投片成功非常重要。对于高技术产业，新产品上市时间(time to market)的压力越来越大，如果由于产品的开发周期过长而错过了时机，那么将花费很大的代价才能挤入市场，甚至会彻底失去产品市场。

技术的发展使得集成电路的制造能力不断提高，工业的发展需要规模越来越大、性能

越来越好的集成电路，设计能力不足已成为充分利用集成电路的制造能力，满足社会需求的瓶颈。为了提高设计能力，可以采取两条途径：① 采用更为结构化的设计方法学，使得设计者可以大量重复使用他人过去设计的部件。② 采用高层次设计工具，使得设计者可以在高层次开始设计。

本书主要讨论数字集成系统的结构化设计方法。数字系统可结构化地划分为从顶向下的不同层次，在每个设计层次，设计者需要构造系统的可仿真模型，需要对系统进行结构性划分，还需要对划分得到的子系统构造可仿真模型。对于每个设计层次构造的系统模型，向上可以作为更大系统的子系统，使更大系统的设计进程独立于本层次的设计过程；向下可以作为子系统的设计规范，使各个子系统的设计过程相互独立。采用结构化的设计技术，数字集成系统的设计就可以重复使用不同层次的 IP(intellectual property)芯核，完整的数字集成系统的设计就不再局限在单一的设计部门，甚至不再局限于同一个企业内部。在完整的系统中，各个 IP 芯核全由相应的专家完成，通过技术的转移即可迅速地实现系统。采用结构化的设计技术，可以有效地提高设计方案的可重复使用性，从而提高人们的设计能力。本书还介绍了数字系统高层次综合的技术，包括算子调度、资源分配的基本方法和技术。为读者使用高层次设计工具进行数字系统的设计建立了理论基础。

全书共分 9 章。第 1 章是引言，首先分析了电子系统集成的瓶颈所在，讨论了系统设计的重要性。然后介绍了集成系统的设计层次，讨论了各设计层次的问题、描述方法及使用的设计工具，在讨论了数字系统设计的不同描述方法之后，简单介绍了系统的分解、设计和综合的概念。

第 2 章介绍使用硬件描述语言 VHDL 构造数字系统的模型的技术，对使用 VHDL 进行结构化集成电路设计和综合的过程进行了解释。VHDL 本身的语法及其特征已有大量图书资料进行介绍，本书只简单讨论用 VHDL 构造硬件模型时应特殊考虑的问题。

第 3 章主要介绍了基本逻辑单元的 VHDL 模型，在介绍了各种组合逻辑电路和时序逻辑电路的模型之后，讨论了逻辑门级模型的精度问题。为了能够精确构造数字系统的模型，使模型精确描述时序关系，提高仿真效率和模型的通用性，该章还介绍了工业标准的 VITAL、预定义库、建模指导方针和从标准延时格式获得反向标注的语法。最后讨论了多值逻辑的需求，并介绍了 IEEE 9 值逻辑。

第 4 章讨论系统级设计问题，即把硬件的自然语言描述转换为真值表、状态图或算法模型的过程。结合设计实例对进程模型图、子系统间的互连、时分复用等问题进行了讨论，并重点介绍了如何将硬件系统的自然语言描述转换为算法模型。最后以与 Intel 8031/51 兼容的 8 位微控制器设计为例说明了系统设计的概念。

第 5 章讨论集成电路的寄存器传输级设计，包括如何将数字系统划分为控制单元和数据路径两部分，以有限状态机方式和微码 ROM 两种方式描述控制单元，以及如何用 VHDL 数据流方式和寄存器级单元序列设计数据路径，再以互连的形式将两部分连接成完整的系统。作为设计实例，该章详细讨论了超级精简指令集计算机 URISC 的设计，并对第 4 章中的实例 Intel 8031/51，完成了从系统级到寄存器级的变换。由于数字系统设计过程中主导的设计方法是把寄存器传输级代码作为综合器的输入，因此，该章还从综合的角度讨论了寄存器级的 VHDL 代码与综合得出的电路的关系。

第 6 章讨论组合逻辑及时序逻辑电路的多层次混合设计问题，即如何混合使用硬件描述语言、卡诺图、状态表等多种方法设计组合逻辑及时序逻辑电路。涉及的电路设计覆盖了系统级的算法描述直至逻辑门级电路描述。此外，该章用较多的篇幅讨论了微代码控制单元的设计问题，介绍了一个基本微代码控制器 BMCU，并设计了该控制器的微代码，用以实现串并转换电路的控制单元。

采用高层次设计工具，可以有效提高数字设计能力。第 7 章讨论高层次综合问题，即从系统级的行为描述出发，由 EDA 工具经过一系列自动转换，生成寄存器传输级描述。其中，行为级描述是设计对象的功能或算法表示，而寄存器传输级描述则是实现这些功能或算法的某个硬件结构的表示。该章首先概要地介绍了行为综合的发展，然后论述行为级描述的内部表示（中间格式）和行为级综合的目标体系（目标结构），最后讨论行为综合的流程和行为级 VHDL 建模，并以一个实例说明了行为综合的建模过程。

高层次综合或者算法综合的意义是把硬件的抽象行为描述自动转换为寄存器级模型。对于给定的系统级的算法描述，由自动综合可以转换为多种不同寄存器级模型，而自动综合本身也是一个困难的、不成熟的而同时又很有吸引力的研究领域。第 8 章介绍综合过程中需要的调度和分配算法，对各种调度和分配算法的优点及局限性进行了讨论。

第 9 章讨论硬件描述语言在数字系统设计领域之外的应用，包括数模混合信号系统、超大规模集成电路测试、微机械/微结构设计、性能分析等方面。这部分的研究已吸引了世界各地的学者进行广泛的研究，并已经取得了一些成果。

阅读本书要求读者具备数字电路与逻辑设计、程序设计语言或 VHDL 语言等方面的基础知识，尽管 VHDL 语言贯穿在全书之中，但并不要求读者在阅读本书之前全面掌握 VHDL。随着本书对硬件造型内容的深入讨论，对 VHDL 的语言特征，特别是与硬件设计和综合密切相关的语言特征也逐步进行了介绍。

本书是在清华大学电子工程系研究生课程讲义的基础上改写而成，它是清华大学电子工程系有关教师长期工作的积累，许多教师对本书的内容以及课程本身提出过许多宝贵的建议，包括刘润生教授、董在望教授、刘序明教授和汪蕙教授等。此外，刘宝琴教授认真地审阅了原稿，在此一并感谢。

本书取材于国际上关于结构化集成电路设计的最新专著及文献，综合了分散于不同文献中的知识和设计经验，也包含了笔者在清华大学期间的研究工作的总结。笔者有幸通过本书的编写与读者交流数字集成系统设计方面的体会，并期望为我国集成系统设计人才的培养与信息产业的发展贡献微薄之力。书中出现的错误和不足，欢迎指正。

作　者

2000 年 2 月

目　录

第1章 引 言

超大规模集成电路(VLSI)可以划分为通用集成电路和专用集成电路两大类。存储器等大量生产且设计规则者称为通用集成电路,面向某一应用背景而专门设计者称为专用集成电路(ASIC)。目前,专用集成电路设计中面临着设计能力几乎没有改进而设计复杂度却不断提高的矛盾。统计表明,对于一个规模较大的专用集成电路设计,通常由10～15个人在18个月的时间内完成,每个设计者的平均设计能力为每天十几个器件。与此同时,专用集成电路的设计复杂度却在成指数增加,1984年专用集成电路的设计所要求的电路规模大致在几十万个晶体管,1996年的专用集成电路要求的设计规模达到了100万至200万个晶体管,估计到2000年专用集成电路要求的设计规模会达到700万至800万个晶体管,到2010年要求的设计规模将达4000万个晶体管。按目前的设计能力,使用最先进的逻辑和寄存器变换层次的综合工具,10～15个人在18个月中能完成的设计规模大致为100万～200万个晶体管。这意味着如果不改变设计方法,到2010年,设计能力与设计需求之间的差距会达20～40倍。显然,人们必须改进设计方法,在保证设计质量的前提下提高设计能力,即提高专用集成电路设计的"生产率"。实现这一目标有两条途径:

(1) 采用高层次设计工具,使得设计者可以在高层次开始设计。现在寄存器变换层次的综合工具已经被设计者广泛接受,今后要采用更高层次的行为综合工具(有时称为结构综合或者高层次综合)。

(2) 采用更为结构化的设计方法学,使得设计者可以大量重复使用前人过去设计的部件。统计表明,新设计中通常会有70%是过去用过的部件,现在由于设计方法不合适或者缺乏设计工具而不得不重新设计这些部件。

1.1 设计层次

标准硬件描述语言(HDL)的出现(比如VHDL和Verilog语言),促进了集成电子系统的高层次设计。硬件描述语言可以用来描述完整的电子系统,也可以描述电子系统的子系统,可以在一个层次化的设计描述中把多个子系统连接在一起。过去十几年间,开发出了许多电子设计自动化工具,包括综合和仿真工具,这些工具的采用,有助于在高层次上描述和设计集成电路。今天硬件描述语言已应用到从逻辑门到行为等各种抽象层次的设计描述中,并已为集成电路设计界的广大技术人员所接受。

在集成电路的设计过程中,时序是一个重要的概念。事实上,如果不特别强调抽象层次,则集成电路的设计过程可以看作是时序描述逐次精确的过程,即从高层次的概念描述

(运算、结构、语句、原始单元等)到可详细给出时序特征的基本单元描述的变换过程。通常将集成电路设计的层次划分为 6 个层次,如表 1.1 所示。最高层次为系统级,最低层次为版图级(又称物理级)。硬件设计方案可以在任何层次描述。

表 1.1 集成电路设计的层次

抽象层次	时序单位	基本单元	电路的功能(行为)描述
系统级	数据处理	进程及通信	自然语言描述或者相互通信的进程
算法级	运算步	运算的控制	行为有限状态机、数据流图、控制流图
寄存器变换级	时钟周期	寄存器、运算、变换	布尔方程、二元决策图、有限状态机
逻辑门级	延时	逻辑门、器件(晶体管)	原理图
电路级	物理时间	晶体管、R、L、C 等	电压、电流的微分方程
物理(版图)级	·	几何图形	

设计描述的最低层次为版图级。版图级用几何图形描述硬件,即用硅表面上的扩散区、多晶硅和金属等来表示硬件。在这一层次,硬件的描述只是结构,而硬件的功能则隐含在器件的物理方程中。

版图级层次之上的层次是电路级。电路级用有源器件和无源器件的互连关系描述硬件。电路级的基本单元可以为双极型晶体管、MOS 晶体管、电阻、电容等。器件的互连关系描述了电路的结构,电压电流之间所满足的微分方程描述了电路的功能。

电路级之上是逻辑门级,这是数字系统设计的主要层次。在逻辑门级设计时,电路的基本单元通常是与门(AND)、或门(OR)、异或门(XOR)、反向器等逻辑门,有时还带有少量晶体管构成的开关以及 D 触发器、J-K 触发器、锁存器等逻辑单元。在这一层次,基本的时序单元是延时。基本电路单元的互连构成逻辑门级的结构描述。这一层次最典型的描述是原理图。当然,也可以用 VHDL,Verilog 这样的硬件描述语言来描述,设计所需元件(逻辑门、线网、晶体管器件)的特征由延时描述。为了计算整个设计的性能,需要仿真工具,也需要时序分析工具。时钟周期通常由两个存储器件之间的最长路径确定,一条路径上可能包含多个非存储部件。布尔方程是这一层次上行为描述的基本形式。

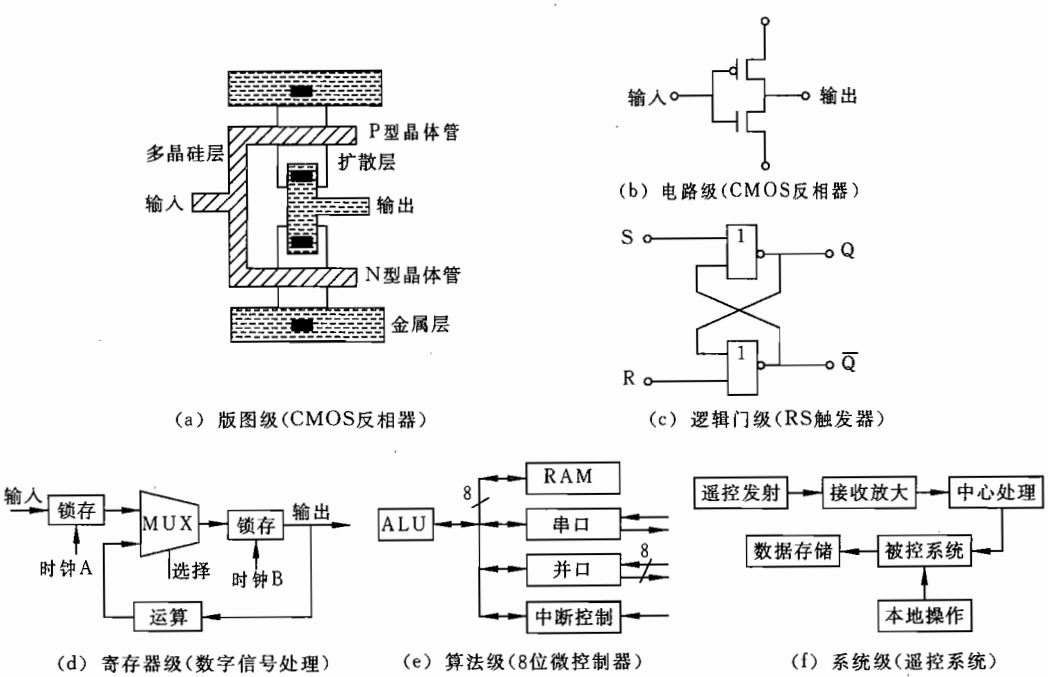
逻辑门级之上的层次是寄存器变换级(RTL 级)。寄存器级最基本的设计单元是寄存器、计数器、多路选择器、算术逻辑单元(ALU)等。这里的基本单元有时也称为功能块,相应于 VLSI 设计时的宏单元,因此 VLSI 设计的寄存器级也称为宏单元级。尽管寄存器级设计的基本单元也可以继续展开成基本逻辑门,但在寄存器级设计时通常不作这种展开,而是把它们作为整体处理。寄存器级设计的结构描述是其基本单元的互连。由于寄存器级设计时基本单元的行为通常由真值表或状态表描述,这一层次上完整硬件的行为通常也由真值表或状态表描述。有时将这一层次的行为描述成为数据流图,它反映了流经实际硬件的数据分布。在寄存器变换级,通常在时钟周期意义上定义系统。典型的描述方法为在哪个时钟周期中完成哪些运算。在这一层次上,可以用于综合工具的典型描述为布尔方程、有限状态机(FSM)和二元决策图(BDD)。这种描述可以从电路的硬件描述语言中自动提取出来。这一层次的主要设计工作是优化、综合、状态编码以及工艺映射。这里的主

要优化工作是减少完成运算所需要的时钟周期数和逻辑门数。优化过程通常要在这两个指标之间进行折衷。如果使用硬件描述语言 VHDL 写出一个系统的 RTL 级描述，总是假定两个 wait 语句之间的所有运算都可以用互连的逻辑门实现，且这些逻辑门可以在一个时钟周期内完成相应的运算。将时钟周期分解成延时单元的工作由 RTL 综合器自动完成。当然，集成电路设计者可以通过写出不同风格的 VHDL 源代码或者设定不同的约束条件控制综合结果。不同风格的描述包括同步描述和以时钟周期为基础的描述。

RTL 级之上的层次是行为级，也称为算法级。在这一层次，设计由运算步来描述。运算语句和控制语句(loop，wait)被用来组织输入、输出以及不同运算。这一层次的电路描述通常是事件驱动的描述，一般由外部世界和内部运算之间交换数据的协议构成。运算步指的是两次相邻的输入、输出或者同步点之间的数据处理。一个运算步通常会占去多个时钟周期。某些情况下运算步可能包含与输入数据有关的运算，这意味着运算步的计算时间不可预测。这一层次的典型描述方法是行为有限状态机(BFSM，behavioral finite state machine)、控制流图(CFG，control flow graph)、数据流图(DFG，data flow graph)和控制数据流图(CDFG，control data flow graph)。

层次化设计的最高层次是系统级。系统级的基本单元可以是计算机、磁盘驱动器、总线接口等大的数字单元。在系统级，行为描述的内容通常为一些性能指标。对于计算机系统，性能指标可能为每秒兆指令(MIPS)、总线传输速率等。对于通信系统，性能指标可能为系统带宽、传输速率等。如果性能指标是确定性物理量，通常在系统级要采用很高级的数据类型，比如复数、实数、时间、频率等等。

图 1.1 给出了数字系统设计时不同层次中所采用的基本单元实例。图(a)所示为最低



(a) 版图级(CMOS反相器)　(b) 电路级(CMOS反相器)　(c) 逻辑门级(RS触发器)

(d) 寄存器级(数字信号处理)　(e) 算法级(8位微控制器)　(f) 系统级(遥控系统)

图 1.1　集成电路设计的不同层次中所采用的基本单元举例

层次，即物理级（版图级）层次，它是一个 CMOS 反相器的物理版图，它由分布在硅片表面的不同层的几何图形描述，器件的行为隐含在器件几何尺寸、工艺参数和器件物理方程中。图(b)为电路级层次，CMOS 反相器由互连的两个 MOS 晶体管构成，电路行为隐含在电路方程中。图(c)为逻辑门级层次，反相器就是一个逻辑器件，其行为由布尔方程描述。在更高的层次，电子系统由更大的模块构成，其实例如图(d)，(e)，(f)所示。

对于大部分硬件描述语言，都允许在中间三个层次（逻辑门级、RTL 级、算法级）描述电路。在每个层次相应于使用硬件描述语言的一个子集写出系统的描述。当描述一个大型系统时通常将不同层次的描述方法混合使用，很少单独采用某一层次的描述。一般情况下，完整的系统由多个模块构成，每个模块分别在不同层次描述。例如，图 1.2 给出了一个电话答录机的方框图。答录机工作于电话线和电话机之间，它通过一个 D/A 和 A/D 与这些外部环境相连接，D/A 和 A/D 由逻辑门层次和电路层次描述。电话答录机还连接到磁带录音接口和定时电路，这两个模块在 RTL 层次描述。电话答录机的中央控制单元则在行为（算法）层次描述。

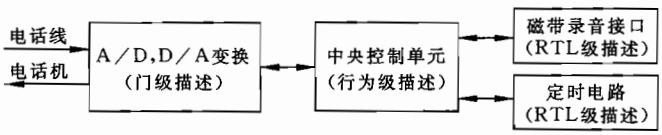


图 1.2　多层次描述示例（磁带式电话答录机）

由于在不同层次描述了一个系统，所以需要不同层次的设计工具。比如，在设计 A/D 和 D/A 转换器时，需要原理图输入工具、电路和逻辑仿真工具以及布图工具；设计两个接口部分时需要 RTL 层次的设计工具（比如逻辑综合工具）；中央控制单元的设计则需要高层次设计工具。

1.2　设计描述的文字表示及图形表示

按照集成电路的描述方式划分，设计方案可以由图形表示，也可以由文字表示，两种

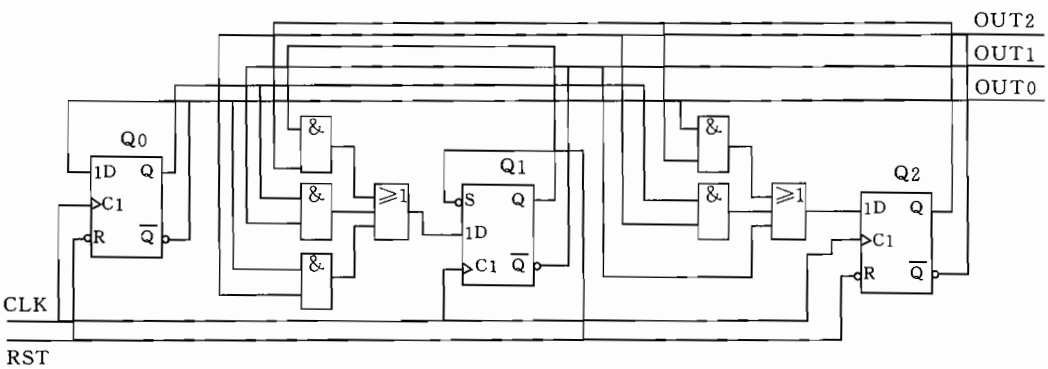
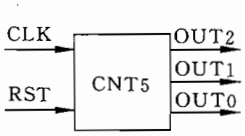
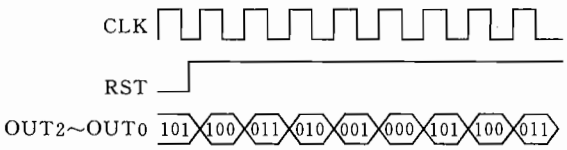


图 1.3　同步 5—0 减法计数器 CNT5

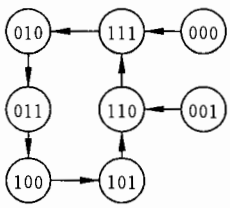
方式各有其优点。过去，对于数字电路设计，人们更喜欢图形方式，即用方块图、时序图、逻辑图（原理图）描述硬件。然而有了硬件描述语言之后，文字描述变得越来越重要。按照集成电路描述的内容划分，可以分为结构描述和行为描述。结构描述用基本单元表示设计方案，行为描述采用算法表示电路的功能。图 1.3 是一个二进制同步减法计数器，在复位状态时计数值为 5，在计数状态下，每输入一个时钟脉冲，计数值减 1。这是典型的结构描述。（在以后的讨论中，把此电路称为 CNT5。）同一电路可由图 1.4 所示的方框图、时序图、状态转换图或状态表来描述，这些都是硬件系统的图形表示。其中方框图是一种结构描述，时序图、状态转换图、状态表则是行为描述。



（a）方框图



（b）时序图



（c）状态转换图

计数值			当前状态			下一状态		
OUT2	OUT1	OUT0	Q2	Q1	Q0	Q2	Q1	Q0
1	0	1	0	1	0	0	1	1
1	0	0	0	1	1	1	0	0
0	1	1	1	0	0	1	0	1
0	1	0	1	0	1	1	1	0
0	0	1	1	1	0	1	1	1
0	0	0	1	1	1	0	1	0

（d）状态表

图 1.4　电路 CNT5 的图形描述方式

除了图形表示方式之外，还可以用文字描述硬件。与图形描述一样，文字描述也可以分为行为描述和结构描述。硬件的不同表述方法如图 1.5 所示。图中硬件描述可以按描述方法分为文字描述和图形描述，文字和图形都可以用来描述电路结构，也可以用来描述电路行为。硬件描述可以按描述内容分为结构描述和行为描述，结构描述和行为描述都可以用文字方式表述，也都可以用图形方式表述。

文字描述可以是自然语言、方程（布尔方程或微分方程）以及硬件描述语言等。这里所谓硬件描述语言指的是用于硬件造型的具有特定结构的高级编程语言。对于上述电路 CNT5，可以分别写出其自然语言描述和 VHDL 描述。

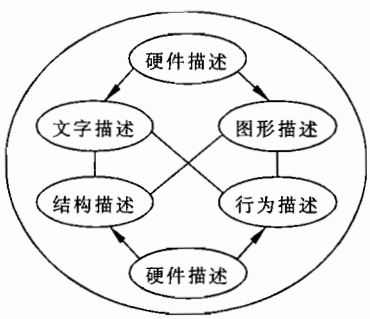


图 1.5　硬件的不同描述方法

用自然语言描述如下：

电路 CNT5 有两个输入端口和三个输出端口，其功能为二进制减法计数器。如果复位信号 RST 输入低电平，则计数值复位为 5（OUT2～OUT0 为 101）。在复位信号 RST 输入为 1 的前提下，电路对于时钟 CLK 的输入脉冲实现循环减法计数。

用 VHDL 行为描述如下：

```
library IEEE;
use IEEE. Std_logic_1164. all;
entity CNT5 is
    port (CLK, RST: in Std_logic; OUT: out Std_logic_vector(2 downto 0);
end CNT5;

architecture DATAFLOW of CNT5 is
    signal Q2, Q1, Q0: Std_logic;
    variable D2, D1, D0: Std_logic;
begin
    D2 := (not Q1) or (not Q2 and Q0) or (Q2 and not Q0);
    D1 := (not Q2 and not Q0 ) or (not Q1 and Q0) or (Q2 and Q1);
    D0 := nor Q0;
    STATE: block(CLK = '1' and (not CLK'STABLE) or RST = '0')
    begin
        Q2 <= guarded '0' when RST = '0' else D2;
        Q1 <= guarded '1' when RST = '0' else D1;
        Q0 <= guarded '0' when RST = '0' else D0;
    end block STATE;
    OUT <= (not Q2) & (not Q1) & (not Q0);
end DATAFLOW;
```

上述 VHDL 源代码中使用了 IEEE 颁布的 1164 标准程序包中定义的数据类型 Std_logic 和 Std_logic_vector。本书的 VHDL 程序中，除了特别声明之外，都使用这个程序包。使用这个程序包时，需要在代码中书写下述的两行代码：

```
library IEEE;
use IEEE. Std_logic_1164. all;
```

为了书写简单，本书中经常略去这两行代码。

文字形式是描述硬件行为的重要方式，而文字形式的主要语言是硬件描述语言。用硬件描述语言描述硬件的行为，通常有两个主要类型：算法描述型和数据流描述型。它们的定义如下：

算法描述型：通过定义输入/输出响应的方式描述硬件行为，行为描述与硬件的物理实现无关。

数据流描述型：描述硬件行为时，数据的相关性与硬件的物理实现一致。

根据上述定义，硬件行为的算法描述是纯过程或者说是一段程序，用以检查硬件功能是否正确，并不关心如何制造出这个器件；而硬件行为的数据流描述则表明了数据如何在硬件内部流动。一般可以认为，数据流型硬件行为描述是硬件的寄存器级实现，算法型硬

件行为描述是硬件的芯片级实现。当然，这里的“实现”是硬件描述语言实现，而不是物理实现。

例如，前面讨论过的电路 CNT5 的算法型描述如下：

```
architecture ALGORITHM of CNT5 is
begin
    process(CLK)
    begin
        if RST = '0' then
            NUM := "101";
        else
            if CLK'EVENT and CLK = '1' then
                OUT <= DEC(OUT);
            endif;
        endif;
    end process;
end ALGORITHM;
```

其中，DEC 是一个实现 Std_logic_vector 型矢量减 1 的函数。其定义如下：

```
function DEC(X : in Std_logic_vector ) return Std_logic_vector is
    variable XV: Std_logic_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;
```

事实上，电路 CNT5 的数据流描述 DATAFLOW 和算法描述 ALGORITHM 并不完全等效。算法描述 ALGORITHM 并没有严格定义循环减法。

在前面讨论的例子中，结构描述大都是图形方式，行为描述大都是文字方式，但也有例外，比如也可以用图形（状态表、状态图以及时序图）描述硬件的行为。作为一般原则，可以得出的结论是：一般情况下文字方式适合描述行为，特别是复杂行为；图形方式适合描述器件的内部互连关系，即描述结构。在大规模系统设计时，两种形式缺一不可，通常要交

叉使用两种形式。

1.3 集成电路设计与综合

集成电路的设计指的是从硬件的一种描述形式到另一种描述形式的变换，设计的最终目标是得出集成电路的某种可制造的描述形式。集成电路设计过程是从顶向下的过程，硬件设计者可以从系统设计开始直到设计出最底层的物理版图，也可以利用某些自动综合工具完成设计过程中的某些步骤。设计过程中可以把硬件描述从同一层次的一种描述转换为另一种描述(一般是从行为描述转化为结构描述)，也可以把上一层次的描述转换为下一层次的描述。

硬件的物理实现则是一个从底向上的过程。如果是一般的电子系统设计，设计者可以用规模较小的集成电路单元实现系统级的硬件系统；设计者也可以先完成硬件系统的逻辑门级结构描述，然后用现场可编程门阵列(FPGA)或用电可编程逻辑器件(EPLD)实现系统。对于集成电路设计，设计者可以只完成硬件的逻辑门级结构描述，然后由集成电路制造者用门阵列或标准单元方法将逻辑门级结构描述映射到版图，最后制造集成电路，这种设计方法称为半定制集成电路设计方法。设计者也可以自行设计出集成电路的掩膜版图，由集成电路制造者根据版图数据制造集成电路，这种设计方法称为全定制集成电路设计方法。硬件系统的设计过程与物理实现的关系如图 1.6 所示。

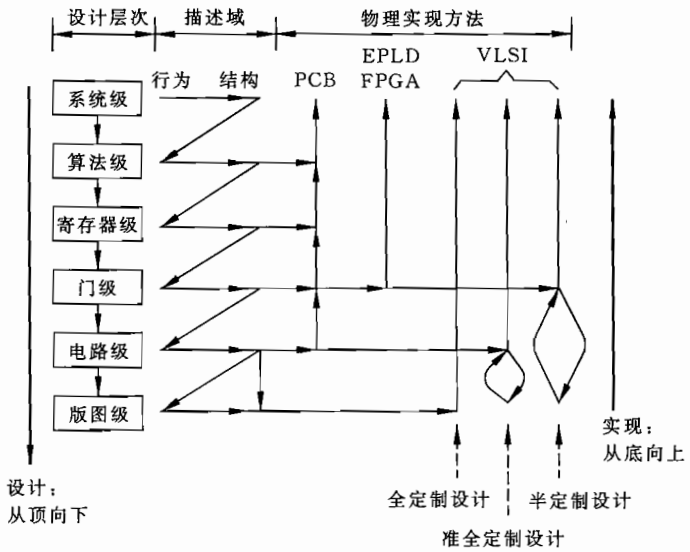


图 1.6 集成电路的设计过程和物理实现过程的关系

按照层次化的概念，集成电路设计过程定义为从硬件的高层次抽象描述向低层次物理描述的一系列转换过程，直到得到可制造的硬件描述为止。从最高层次的描述开始到得出可制造的某种描述为止的过程称为设计周期。采用手工方式把硬件的一种描述形式转换为另一种描述形式的过程称为设计；采用自动化的计算机辅助设计工具把硬件的一种

描述形式转换为另一种描述形式的过程称为自动综合，简称综合。

综合来源于英文单词 synthesis，其基本含义是把一些抽象的对象组合为一个统一的整体。在集成电路设计中，综合有特殊含义。其定义为“集成电路设计过程中设计描述的一种形式向另一形式的(自动)变换过程”。

综合工具用于硬件设计过程中的自动设计或帮助设计者进行设计。根据上述设计(综合)过程的定义，可以将集成电路设计中的综合器定义为“把硬件设计的一种描述形式向另一种描述形式进行自动转换或帮助硬件设计者进行这种转换的软件工具”。

在本书中并不特别强调是某一种综合工具，原因在于当前大部分综合工具都是应用于某些专用的行业或由某个公司所开发，由于技术专利等方面的因素使得一般用户并不能自由得到或使用这些工具。除此之外，高层次综合还是不成熟的技术，也是当前学术界和工业界广泛关心的问题。因此，本书只简单讨论各种综合工具所涉及的基本原则。

综合器可以把硬件的高层次描述转换为低层次描述，也可以把同层次的行为描述转换为结构描述。由综合器自动完成这种转换，类似于软件开发时利用编译器把高级语言写出的程序转换为可执行机器码。例如，C 语言编译器可以用来把某算法的 C 语言描述转化为机器语言(二进制代码)描述，在特定的计算机上，这种二进制代码可以执行。与此类似，高层次综合器可以把硬件的高层次描述(可能为算法级描述)转换为中间层次的描述(可能为逻辑门级描述)，如果利用门阵列技术或标准单元技术或利用中规模集成电路(7400 系列)，则可以实现这种中间层次的硬件描述。事实上，利用称之为硅编译器的版图级综合器，可以把硬件的中间层次的描述转换为最低层次的描述(几何版图描述)，这种最低层次的描述可以用某种集成电路制造工艺实现硬件。图 1.7 示意给出了软件编译与硬件综合的简单类比。图 1.7 中，在高层次采用硬件描述语言描述电路相应于软件设计中使用高级程序设计语言编写软件；高层次综合器相应于软件设计中的高级语言编译器；高层次综合器产生的逻辑门或电路层次的设计描述相应于软件设计的汇编程序；逻辑门层次或电路层次描述经版图综合得到可制造的集成电路掩膜数据相应于软件设计中汇编程序编译得到可执行的二进制代码。

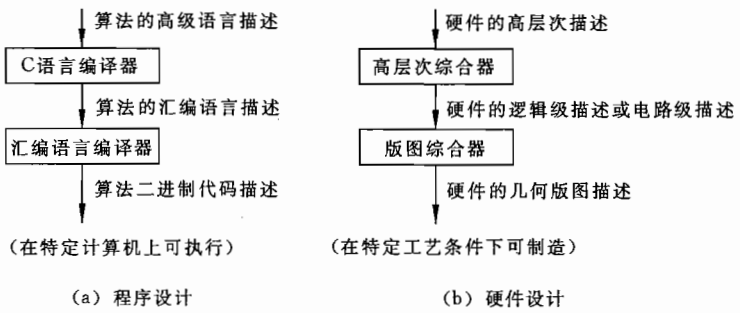


图 1.7　软件的编译与硬件的综合

与软件程序设计类似，集成电路硬件设计时，利用综合工具从高层次向低层次自动转换与直接在低层次设计硬件相比各有优缺点。这与软件设计时用高级语言编程和用汇编语言编程的对比很类似。对于大型软件设计，直接采用机器代码或汇编程序设计软件几乎

是不可能的，必须采用高级语言设计软件。用高级语言设计程序，设计者的精力可以集中在算法的实现上，而不必考虑计算机系统中寄存器的使用、物理内存的分配等细节问题，可以节省设计时间，减少设计错误。然而，由于高级语言编译器必须考虑从高级语言向低级语言转换时的各种可能情况，对于某些具体问题，编译得出的机器码可能不是最优的。即使是带有优化功能的编译器，也只能进行某些特定类型的优化，永远也不可能考虑到所有可优化的情况。一般地讲，有经验的程序员直接用汇编语言写出的程序要比用高级语言写出的程序质量好。对于大规模集成电路设计，情况与软件设计完全类似。直接在版图级或电路级设计大规模集成电路几乎是不可能的。在高层次用硬件描述语言设计硬件，并利用综合工具得到可制造的集成电路版图数据，可以使设计人员集中精力考虑算法的实现，不必考虑电路细节，从而可以节省设计时间，减少设计错误，降低设计成本。然而由于综合器的局限性，综合得出的电路无论从面积还是性能上都不一定是最优的设计。一般地讲，有经验的电路设计者直接在中间层次（逻辑或电路级）设计的硬件质量要比综合器产生的好。

下面是一个奇偶校验电路的 VHDL 行为描述。8 位寄存器 R 中保存着数据，如果数据各位中'1'的个数为偶数，则输出 P 为'1'；如果数据各位中'1'的个数为奇数，则输出 P 为'0'。VHDL 源代码中用 for 循环语句描述硬件的行为。

```
entity IPAR is
    generic(PROP_DEL:time);
    port(R: in Std_logic_vector(7 down to 0); P: out Std_logic);
end IPAR;

architecture LOOP4 of IPAR is
    signal CLOCK: Std_logic := '0';
begin
    OPAR: process(R)
        variable X: Std_logic ;
    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;
```

上述 VHDL 源代码中的 for 循环语句可以直接转换为图 1.8 (a)所示的异或门级连结构。由于这种级连网络适合于任何 VHDL 循环语句结构，因此一般综合器都把 for 循环语句转换成这种类型的级连网络。然而，正如某些计算机语言（软件）编译器可以对编译

过程进行某种程度的优化一样，硬件综合器也可以对综合过程进行某种程度的优化。在一般综合器中，优化通常放在转换之后。在这个例子中，综合器中的优化工具应该发现图 1.8 (b)所示的异或门树状结构与图 1.8 (a)所示的异或门级连结构具有完全相同的功能，但树状结构从输入信号到输出信号经过的延时较少，电路工作速度较快。再对电路进行细致的研究，根据布尔关系(X xor 0 = X)，可以把第一个异或门去掉。如果允许使用有不同输入端的异或门，可以得到图 1.8 (c)所示的电路结构。与前两种电路结构相比，这种电路门数最少，延时最短。当然，这种优化方法只对异或门级连结构有效，对于其他电路单元的级连结构不一定适用。具有优化功能的综合器应该能判断出什么情况下能优化，什么情况下不能。从这个例子也可以看出，优化通常是迭代过程，硬件设计者必须为优化过程设定一些条件，比如使用的基本单元的层次、允许使用的基本单元的类型等等。

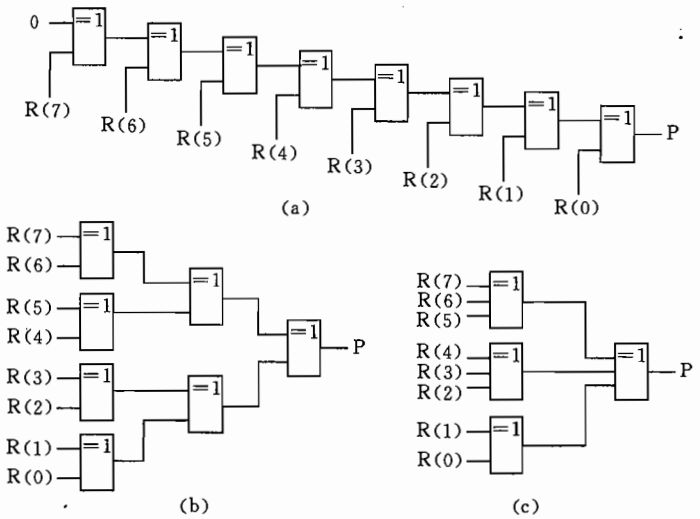


图 1.8　VHDL 的 for 循环转换成的硬件结构及其优化

从上述例子可以看出，从硬件描述的一种形式向另一种形式自动转化(综合)的过程由两部分组成：第一步是简单的转换，第二步是优化。从上述讨论还可以看出，综合器永远也不可能找出硬件设计的全部可能实现方式，从而不可能得出最优的方案。在大多数情况下，设计方案是否最优并不重要，而设计过程的成本和设计的可靠性(不出设计错误)是必须最优先考虑的因素。这种情况下应首先使用自动综合工具。正是由于这个原因，自动综合工具在许多大公司被大量应用。当然，一般情况下，要对自动综合产生的电路进行人工修改，以期得到尽可能好的设计方案。

对于软件设计，如果用汇编语言写应用程序，要求程序员有很好的编程训练，程序设计所花费的代价比较大。同样，在较低的层次设计硬件，也要求硬件设计者有丰富的设计经验，硬件设计成本较高。用什么方法设计硬件主要取决于所设计硬件的预期生产量。对于生产量小的集成电路，应该在较高的层次进行设计，从高层次向低层次的转换，由综合工具完成，目的是降低设计成本。对于生产量较大的集成电路，应该在较低的层次进行设计，目的是降低生产成本和提高电路性能，而在设计上所多花费的代价可以通过批量生产

时所降低的生产成本来抵消。选择合适的设计方法还会影响到集成电路产品上市的时间。在高层次完成设计后，由自动综合工具完成从高层次到低层次的转换，所用的设计时间短；在低层次设计电路，所用的设计时间长，但得到的电路性能较好。

在不同层次设计集成电路可以使用不同的综合器。对于综合器来讲，输入描述为系统级、算法级、寄存器级等逻辑门级以上描述的综合器称为高层次综合器。输入描述为逻辑门级、电路级硬件描述的综合器称为低层次综合器。对于高层次综合器，通常在综合器的名字上带有其输入端的硬件描述层次。比如：把硬件的系统级描述转换为较低一层次的描述的综合器被称为算法综合器；把寄存器级描述转换为下一层次的描述的综合器称为数据流综合器；算法综合器和数据流综合器有时都称为行为综合器。对于低层次综合器，综合器的命名方式有些混乱。有的文献把综合器输入端硬件描述的层次作为综合器的名字。比如，把逻辑门级描述转换为电路级或版图级描述的综合器可被称为逻辑综合器。但也有文献把综合器输出端硬件描述层次作为综合器的名字。比如，把硬件的逻辑门级描述或电路级描述转化为版图级描述的综合器称为版图综合器或硅编译器。

在本书中，用 VHDL 在各种设计层次对硬件进行描述。在特定的设计层次，可能不允许使用某些 VHDL 语法结构，在该层次进行硬件造型时，会特意避开这些语法结构，即只使用在该设计层次有意义的 VHDL 语法结构描述该层次上的硬件行为或结构。随着设计过程的发展，对硬件的描述从一个层次转换到另一个层次，允许使用的 VHDL 语法结构越来越多，关于 VHDL 的知识也会越来越完善。

对于具体的集成电路设计周期，有时要跳过某些层次，即不需经过图 1.6 中的全部步骤。设计周期中的每一步骤都是从硬件的一种描述形式到另一种描述形式的变换，如果在同一层次变换，通常是将行为域的描述变换为结构域的描述；如果是从高层次到低层次的变换，一般是向相邻的下一层次变换。如果变换由计算机辅助设计工具自动完成，则称为综合。图 1.9 是某硬件的设计周期中可能出现的不同综合的例子。在这个例子中包括：① 自然语言综合——把硬件的自然语言描述向算法描述变换；② 算法综合——把硬件的算法描述向数据流图描述或门级描述变换；③ 逻辑综合——从硬件的数据流图描述向

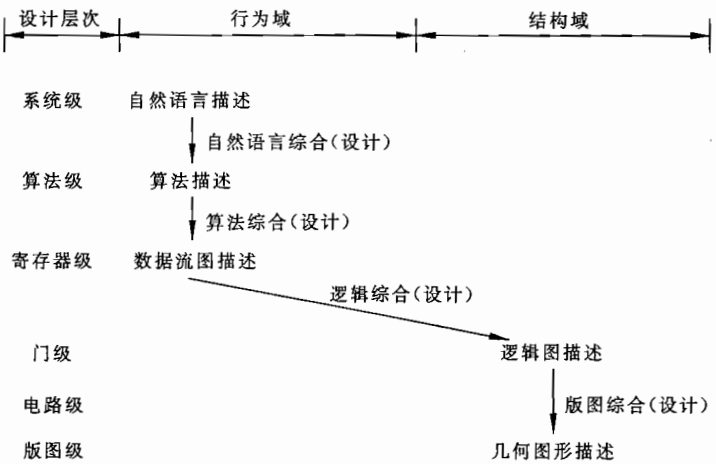


图 1.9　某集成电路的设计过程

门级结构描述转换；④ 版图综合——把硬件的门级结构描述向版图级描述转换。这个例子中没有出现从逻辑门级向电路级的变换过程。在系统级、芯片级以及寄存器级是在行为域中描述电路，在逻辑门级是在结构域描述硬件并从逻辑门级直接变换为版图级的几何描述。

当前，集成电路设计周期中的所有变换步骤中，除了从系统级的自然语言行为描述到系统级的结构描述或到算法级的变换之外，其他各种变换都有自动综合工具。尽管这些自动综合工具的自动化程度不同，综合得出的电路质量也不同，但它们还是为硬件设计者提供了很大的帮助，特别是对于经验不丰富的硬件设计者帮助更大。

1.4 从顶向下和从底向上的设计方式

集成电路的层次化、结构化设计隐含着硬件设计方案的逐次分解。在设计过程中的任意层次，硬件至少有一种描述形式。硬件的描述特别是行为描述通常称为硬件模型。在集成电路设计过程中的每一层次，硬件都可以分为一些模块，该层次的硬件结构由这些模块的互连描述，该层次的硬件功能由这些模块的行为描述。这些模块称为该层次的“基本单元”，而该层次的“基本单元”又由下一层次的“基本单元”互连而成。如此下去，完整的硬件设计就可以由图 1.10 所示的设计树描述。在这个设计树上，节点对应着该层次上的“基本单元”的行为描述，树枝对应着“基本单元”的结构分解。

因为在硬件设计的电路级、逻辑门级、寄存器级以及系统级都可以用行为模型描述，所以按结构分解的方式可以把硬件设计一直分解到电路级，分解得到的最小模块也具有行为域的模型。图 1.10 中的设计树就示意画出了划分到电路级的设计树。如果设计树中树枝最末端的节点全是同一层次的硬件行为模型，即硬件结构分解得到的最小模块全在同一层次描述硬件，则这样的设计树称为全设计树。如果设计树中树枝最末端的节点是不同层次的硬件描述，即硬件结构分解得到的最小模块不在同一层次描述硬件，这样的设计树则称为部分设计树。图 1.10 给出了全设计树和部分设计树的概念。

部分设计树通常出现在硬件设计还没有完全完成的阶段。比如某些模块已经完成了逻辑门级设计，而另外一些模块还只完成了高层次的行为描述，这时的设计树就是部分设计树。硬件设计过程中如果要求对分解成部分设计树的电路进行全面验证，则需要混合层次的仿真工具。混合层次的仿真工具所带来的最大好处是仿真效率较高。

与设计树有关的两个概念是从顶向下(top-down)和从底向上(bottom-up)设计。这里顶指的是设计树的树根，底指的是设计树的末枝。如果硬件设计者开始设计时只知道系统级的内容，即只知道树根的内容，那么可以从树根开始，把系统划分为单元，然后再把每个单元划分为下一层次的单元，一直这样做下去，直到设计树的最末枝可以造出，这样的设计过程称之为从顶向下的设计。从顶向下的设计方法的特点在于每一层次的分解全经过优化，优化的目标可能是工作速度、芯片面积、芯片成本或它们的组合，但每次划分并不考虑分解后得到什么样的单元，以及得到的单元是否是已存在的单元。

从底向上的设计在某种意义上讲可以看作上述从顶向下设计的逆向过程。尽管设计者也是从系统级开始，即从设计树的树根开始对设计进行逐次划分，但划分时首先考虑的

是单元是否存在，即设计划分过程必须从存在的基本单元出发，设计树最末枝上的单元必须是已经制造出的单元，或者是其他项目已开发好的单元以及可外购得到的单元。

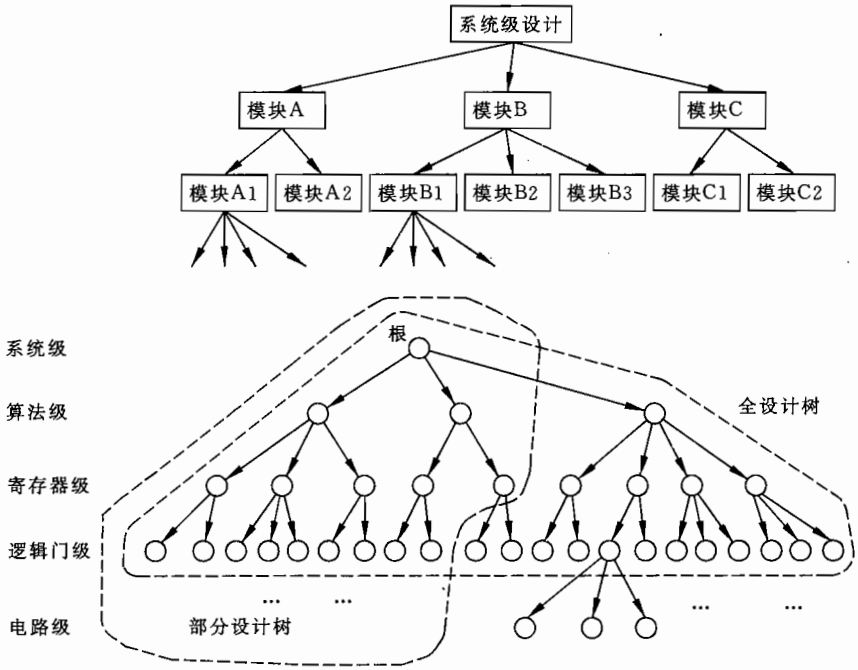


图 1.10　设计的分解及设计树

从顶向下的设计过程中在每一层次划分时都要对某些目标进行优化，这些目标包括工作速度、芯片面积、芯片成本。在集成电路设计时通常将这三个目标组成的空间称为设计空间。由工作速度、芯片面积、芯片成本组成的三维设计空间如图 1.11 所示。从顶向下的设计过程是理想的设计过程，但它的缺点是得到的最小单元不标准，制造成本可能很高。从底向上的设计过程全采用标准基本单元，通常比较经济，但有时可能不能满足一些特定的指标要求。大多数集成电路设计过程是这两种设计方法的结合，设计时要考虑多个目标的综合平衡。

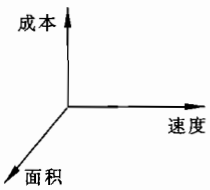


图 1.11　设计空间

第2章 硬件的VHDL模型

2.1 硬件描述语言和VHDL

超大规模集成电路需要结构化的设计方法，硬件描述语言的使用大大促进了结构化集成电路设计的发展。事实上，硬件描述语言并不是新概念。CDL，ISP和AHPL早在20世纪70年代就得到了应用，当时硬件描述语言主要用来验证设计好的电路功能是否正确，用它们来描述硬件时不太精确，原因在于时序模型本身不够精确，或者是依赖于某些特定的硬件结构。20世纪70年代末开始定义新的硬件描述语言，例如Veralog和VHDL，它们具备了更为通用的时序模型，不再依赖于特定的硬件结构。VHDL(VHSIC hardware description language)是由美国国防部在20世纪70年代末提出的VHSIC (very high speed integrated circuit)计划的产物。最初的设想是定义一种语言用来交换硬件设计数据，在开发过程中得到了计算机工业界、EDA工业界和集成电路生产厂的支持，包容了现代硬件描述语言应该具备的全部特征。1985年发布的版本VHDL7.2已经是完备的语言，可以用来描述硬件的结构和行为。此后，IEEE继续支持了语言的定义，并对语言进行了详细的评估和修改，1987年6月被接受为IEEE标准。VHDL语言的作用主要有两点：第一，用来描述复杂系统，硬件的VHDL描述本身就是硬件的设计文档。第二，电子设计自动化领域的研究者以及软件工具的开发商也用VHDL进行各种算法的研究和不同软件工具的开发。本书使用VHDL对结构化集成电路设计和综合的过程进行解释。关于VHDL本身，有大量的教科书和参考文献叙述其语法及其特征，有些语法和句法特征和一般的程序设计语言十分相近。本章只简单讨论使用VHDL进行硬件设计时应特殊考虑的有关问题。本章将讨论VHDL模型的风格对综合的影响，并使用“调度”、“分配”等VHDL综合的术语，关于这些术语的解释，在第7，8两章详细介绍。

2.2 VHDL实体、结构体和进程

硬件的VHDL模型由实体(entity)和结构体(architecture)构成。实体定义了硬件的输入输出端口，定义了硬件与外部环境的接口。实体定义的是一个黑盒子，黑盒子上带有输入和输出的端口说明，但并不包含如何将输入映射到输出的信息。下面的VHDL源代码定义了一个实体BUBBLE，它的输入端口为DATA_IN，DATA_READY和START，输出端口为DATA_OUT。它定义了图2.1所示的一个硬件模块，但没有定义输入信号和

输出信号之间的关系。简单地讲，实体定义了一个“插座”，这个插座只定义了硬件的端口，可以在这个插座上插上任何端口匹配的硬件。

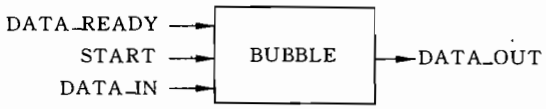


图 2.1　实体 BUBBLE 定义的硬件

```
entity BUBBLE is
port(DATA_READY: in Boolean;
     DATA_IN: in Integer;
     DATA_OUT: out Integer;
     START: in Std_logic);
end BUBBLE;
```

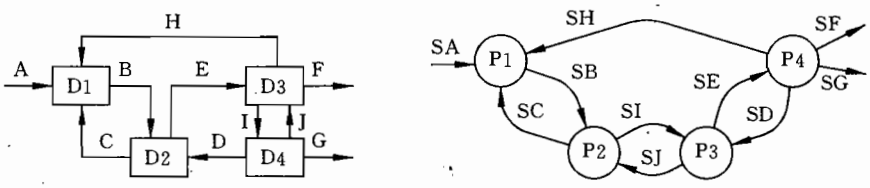
结构体定义了硬件设计的输入端口和输出端口之间的映射关系，用来说明相应的实体的行为。换句话说，结构体定义了插座(实体)上插入了什么器件。设计者可以在任何层次定义结构体，算法级、RTL 级或者逻辑门级都可以写出相应的结构体。结构体通常由说明部分和算法部分组成，算法部分通常是一些并发执行的语句。下面的 VHDL 源代码是实体 BUBBLE 的一种行为描述。结构体 BEHAVIOR 中的算法部分只有一个并发执行的进程(process)语句 SORT。其中，进程实际上是包含在 begin 和 end 之间的无限循环语句。

```
architecture BEHAVIOR of BUBBLE
    -- 结构体的说明部分
    type MEMORY is array (1 to 256) of Integer;
    procedure READ_MEM(DATA_READY; in Boolean;
                       DATA_IN: in Integer;
                       REM: out MEMORY) is
    begin
        for I in 1 to 256 loop
            wait until DATA_READY;
                RAM(I) := DATA_IN;
        end loop;
    end READ_MEM;
    procedure WRITE_MEM(DATA_READY; in Boolean;
                       DATA_IN: in Integer;
                       REM: out MEMORY) is
    begin
        for I in 1 to 256 loop
```

```
            wait until DATA_READY;
                DATA_OUT <= RAM(I);
        end loop;
    end WRITE_MEM;

begin
    -- 算法部分,只包含一个进程
    SORT: process
    variable RAM: MEMORY;
    variable TMP, T1, T2: Integer;
    begin
        wait until (START = '1');
            READ_MEM(DATA_READY, DATA_IN, RAM)
            for I in 1 to 256 loop
                T1 := 257;
                while T1 > I loop
                    T2 := T1 - 2;
                    T1 := T1 - 1;
                    if RAM(T1) < RAM(T2) then
                        TMP := RAM(T1);
                        RAM(T1) := RAM(T2)
                        RAM(T2) := TMP;
                    end if;
                end loop;
            end loop;
            WRITE_MEM(DATA_READY, DATA_OUT, RAM);
    end process SORT;
end BEHAVIOR;
```

在 VHDL 中,进程是一段用来描述硬件功能和延时的可计算的代码,上面结构体中的 SORT 就是一个进程的实例。由多个数字器件互连组成的网络同样可以由进程的互连描述,图 2.2 给出了一个这样的例子。在这个例子中,器件与进程之间的对应关系如下:

器件	进程
D1	P1
D2	P2,P3
D3,D4	P4



(a) 硬件网络　　(b) 进程网络

图 2.2　器件的互连与进程的互连

这个例子说明了三个问题：

(1) 一个器件可以映射为一个进程；

(2) 一个器件可以映射为多个进程；

(3) 多个器件可以映射为一个进程。

硬件网络中的连线可以由进程网络中的信号线描述。对于上述例子，硬件网络中的连线与进程网络间的信号线间的对应关系如下：

器件间连线	进程间信号线
A	SA
B	SB
C	SC
D	SD
E	SE
F	SF
G	SG
H	SH
I,J	无
无	SI,SJ

硬件网络中的连线 A～H 对应着进程网络中的信号线 SA～SH；硬件网络中连线 I,J 在进程网络中没有对应信号；进程网络中的信号 SI,SJ 是由于将器件 D2 划分为两个进程而产生，在硬件网络中没有对应连线。

对于实际的数字器件，如果它的某些输入信号值发生变化，其输出值也会发生变化，称该器件对这些信号敏感。这些使器件输出值发生变化的信号定义为该器件的触发信号，根据定义，器件对其触发信号敏感。与此类似，对于进程，如果某些输入信号值发生变化，进程会被激活，以重新计算进程的输出值，则称进程对这些信号敏感。这些激活进程的信号定义为进程的触发信号。在本书中，如果需要，则用不同的箭头区别触发信号和非触发信号。

大部分 VHDL 仿真器和综合器的输入通常是实体-结构体对。对于复杂系统，结构体的表述也较为复杂。例如，第 1 章中讨论的电话答录机中，系统要与电话交换机通信，也要与答录机内部的部件(比如磁带录音机)通信。通常，描述硬件行为的结构体会包含有多个

进程，有些进程描述接口协议，有些进程描述数据处理功能。图 2.3 是一个复杂系统的实例，图中有两类进程，即数据接口进程和数据处理进程。数据接口进程定义系统与外部世界通信的协议，比如时钟速率、信号的物理特征等等。这些进程通常在较低的设计层次给出，比如 RTL 级或者逻辑门级。相应的 VHDL 描述包含时钟和同步信号。数据处理进程描述由外部事件驱动的电路元件，用于实现硬件的预期功能，这些进程通常在较高的设计层次给出。

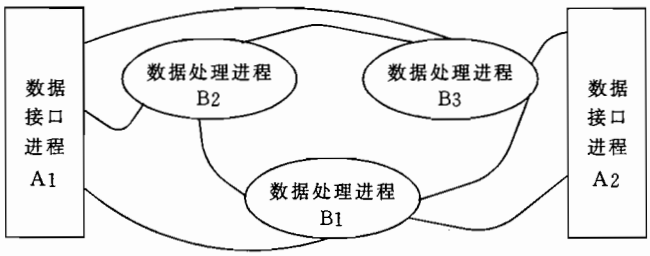


图 2.3　复杂系统结构体举例

2.3　延　　时

电信号必须服从基本物理定律。在实际电路中，电信号'1'和'0'通常用不同电平表示。比如在 CMOS 电路中，通常用+5V 电平表示逻辑'1'，用 0V 电平表示逻辑'0'。在实际电路中，电路中一个节点的电平不可能立即从一个电平切换到另一电平。因此，在数字电路中，从一个逻辑门的输入值发生变化到其输出值产生响应总存在一定延时。VHDL 中，信号赋值用符号 <= 表示，其意义是信号值将在给定的传输延时之后发生变化。信号赋值延时有如下两种方式：

```
Y <= X;                    -- Delta 延时
Y <= X after 10ns;         -- 标准时间单位延时
```

前一个语句中，信号 Y 在 Delta 延时之后得到信号 X 的数值。在 VHDL 中的 Delta 延时大于零，但小于任何指定时间的延时。在语句执行的时刻，Y 的数值并不立即更新为 X 的数值，而是要延时一个时间量 Delta 之后才更新为 X 的数值。在仿真时钟的当前时刻，任何用到 Y 的数值的其他语句，都会使用 Y 的旧值。对于 VHDL 造型来讲，这是很重要的概念，是写出合法 VHDL 模型的关键，也是 VHDL 新使用者错误理解 VHDL 源代码或写出错误 VHDL 源代码的主要原因之一。在后一个语句中，使用了标准时间单位延时。信号 Y 在 10ns 之后得到信号 X 的数值。

VHDL 中的变量赋值语句与一般程序设计语言（比如 C 语言）中的变量赋值语句类似，其作用是在硬件的高层次模型中允许以人们所习惯的方式描述算法。变量赋值用符号 := 表示，变量赋值的作用是即时的，即变量赋值没有延时。下面的例子说明了这两种赋值的差别：

(1) AS <= X * Y after 2ns;

(2) BS <= AS + Z after 2ns;

这两个语句在 VHDL 语言中称为信号赋值语句。在这两个语句中,指定的传输延时为 2ns。变量赋值的例子如下:

(3) AV := X * Y ;

(4) BV := AV + Z ;

虽然延时的概念看上去简单,但如果不注意延时的效果,可能会产生意想不到的错误。为了进一步说明两种赋值的差别,可以跟踪这四个赋值语句的执行情况。假定上述四句赋值语句从 $t1$ 时刻开始执行,在 $t1$ 之后时间轴上的增量为 2ns,注意到信号赋值语句的延时也是 2ns;又假定信号赋值语句和变量赋值语句的初始条件相同,变量 AV,BV 和信号 AS,BS 的初始条件也相同;那么可以得到有延时的信号赋值语句和无延时的变量赋值语句运行的结果,如表 2.1 所示。在 $t = t1$ 时刻,执行信号赋值语句(1),计算出信号 AS 的新值应为 8,但并没有立即将信号 AS 更新为其新值 8,只是规定 2ns 之后将 AS 的值更新为新值 8,所以在 $t = t1$ 时刻,计算信号赋值语句(2)时,赋值号右边所使用的信号 AS 的值仍为旧值 2,计算得出 BS 的新值为 5。同样,由于传输延时的作用,信号 BS 的值也要在 2ns 之后才会更新。执行变量赋值语句(3)时,得出变量 AV 的新值为 8,由于变量值更新发生在赋值语句执行的瞬时,执行语句(3)时变量 AV 立刻更新。在执行语句(4)时,赋值号右边使用的是 AV 的新值,因此得出变量 BV 的值是 11,BV 也立即更新为其新值 11。

表 2.1 信号赋值与变量赋值的对比

时 间		0	$t1$	$t1+2$	$t1+4$	$t1+6$
输入值	X	1	4	5	5	3
	Y	2	2	2	3	2
	Z	0	3	2	2	2
信号赋值	AS	2	2	8	10	15
	BS	2	2	5	10	12
变量赋值	AV	2	8	10	15	6
	BV	2	11	12	17	8

在 $t = t1+2$ 时刻,信号 AS 和 BS 分别更新为其新值 8 和 5,这时仍按上述顺序执行四个语句。执行信号赋值语句(1)时,算得信号 AS 的新值应为 10,但并没有立即将信号 AV 更新为其新值 10,只是规定 2ns 之后将 AS 的值更新为新值 10,AS 的值保持为旧值 8。在计算信号赋值语句(2)时,赋值号右边所使用的信号 AS 的值仍为旧值 8,计算得出 BS 的新值为 10。同样,由于传输延时的作用,信号 BS 的值也要在 2ns 之后才会更新。执行变量赋值语句(3)时,得到变量 AV 的新值为 10,由于变量值更新发生在赋值语句执行的瞬时,执行语句(3)时变量 AV 立刻更新。在执行语句(4)时,赋值号右边使用的是 AV 的新值,因此得出变量 BV 的值是 12,BV 也立即更新为其新值 12。

读者可以自行验证 $t1+4$ 和 $t1+6$ 时刻的情况。从这个例子可以看出,由于传输延时的影响,输入信号 X 和 Y 在 $t1$ 时刻的变化,直到 $t1+4$ 时刻才影响输出信号 BS 的值。由此可以看出信号赋值与变量赋值的明显差别,尽管信号 BS 和变量 BV 的表达式相同,但

信号 BS 的值并不是变量 BV 的简单延时。这是一个很简单的例子，但它说明了传输延时所带来的影响。在本书后面的讨论中，很多地方都要求预先了解延时的影响。

由于变量赋值语句中不考虑延时的影响，主要用于算法研究，所以在 VHDL 中规定变量赋值只能出现在进程、函数(function)和过程(procedure)等语法结构中。对变量的定义只能在过程之中或子程序(sub-program)内，变量说明语句不允许出现在结构体或模块(block)内。

在 VHDL 中，实体的端口相应于实际硬件中的信号接口，而结构体用来描述实体的行为，因此端口定义及相应的结构体中的对象只能是信号。在结构体或模块的说明域中，可以定义内部信号，信号赋值语句可以出现在 VHDL 源代码中的任何地方。下面的 VHDL 源代码给出了满足这些限制条件的信号、变量的定义以及完整的赋值语句。

```
entity STATEMENT is
    port (X, Y, Z: in Integer; B: out Integer);
    -- 注意:实体端口总是信号
end STATEMENT;

architecture PROP_DELAY of STATEMENT is
    signal AS: Integer;
begin
    process(X, Y, Z)
    begin
        AS <= X * Y after 2ns;      -- 语句(1)
        BS <= AS + Z after 2ns;     -- 语句(2)
    end process;
end PROP_DELAY;

architecture INSTANTANEOUS of STATEMENT is
    variable AV, BV: Integer;
begin
    process(X, Y, Z)
    begin
        AV := X * Y ;       -- 语句(3)
        BV := AV + Z ;      -- 语句(4)
        BS <= BV;
    end process;
end INSTANTANEOUS;
```

根据 VHDL 中的规定，进程中的语句执行的先后顺序是它们在源代码中写出的顺序。永远按源代码中写出的顺序执行的语句称为顺序执行语句。在上例中的两个结构体

PROP_DELAY 和 INSTANTANEOUS 内的信号赋值语句和变量赋值语句全是顺序执行语句。

在结构体 INSTANTANEOUS 中的进程中，变量语句的顺序非常重要。如果语句(4)比语句(3)先执行，所得到的运算结果会与表 2.1 完全不同。与此不同，在结构体 PROP_DELAY 中的进程中，信号赋值语句的顺序并不很重要。由于信号赋值语句执行时，赋值号右边的信号全取当前值，计算所得到的赋值号左边的信号新值要到 2ns 之后才出现在信号上，所以将语句(1)与语句(2)交换顺序不会影响运算结果。这就是说，从仿真时间的角度来看，语句(1)和语句(2)是“同时”执行的。事实上，VHDL 仿真时，还可以使不同进程中的信号赋值语句“同时”执行。

可以同时执行的语句称为并发执行语句。VHDL 中支持并发赋值语句，因此有可能把上例中的语句(1)和(2)描述成为并发赋值语句。可以实现并发执行语句的原因是延时的存在。读者可以利用 VHDL 仿真器对上述结论进行验证。下节中讨论 VHDL 仿真器如何利用延时实现并发执行的语句。

2.4 延时与并发性

硬件电路中的信号总是同时工作的，这就是硬件电路的并发性。描述硬件行为的 VHDL 模型必须保证仿真时的并发性，图 2.4 示意给出了这种并发的概念。在这个图中有三个逻辑模块，假定模块 B1 和模块 B2 同时作用，即模块 B1 和 B2 并行激活，模块 B3 应该在模块 B1 的输出(Z1)或模块 B2 的输出(Z2)值发生变化时立即被激活。在信号通过模块 B3 传输的同时，输入信号 IN1 或 IN2 的变化会通过模块 B1 或 B2 传输，这就是说，信号流可以同时在三个模块中流动。

硬件描述语言应该有能力描述实际电路中信号“同时”流动的特性。在 VHDL 中，这种要求通过进程满足。每个进程表示一个逻辑模块，各个进程之间并行执行。当然，一般仿真器都在单 CPU 的计算机上运行，不同进程的执行实际上存在先后顺序，但从仿真时间的角度来看，不同进程是同时执行的。当一个进程的敏感信号的信号值发生变化时，该进程被激活。进程的敏感信号保存在其敏感信号表中，一般情况下，敏感信号表中保存的就是该进程描述的模块的输入信号。图 2.4 中包含了为每个模块建立 VHDL 进程描述所需要的信息，比如模块 B1 可以表示为下述 VHDL 进程：

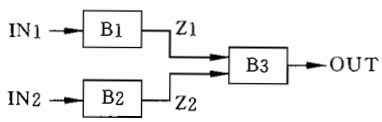


图 2.4 逻辑模块向进程的映射

```
LOGIC_BLOCK: process(X1,X2,X3)
    variable YINT: Std_logic;
begin
    YINT := X1 and X2;
    Z1 <= YINT or X3 after 30ns ;
```

end process LOGIC_BLOCK1;

在这个例子中，进程体内包含有变量说明部分，在说明部分定义了局部变量 YINT，它是这个进程的内部变量。进程的运算部分是关键字 begin 和 end 之间的源代码部分。在这个例子中，进程的运算部分中包含对中间变量 YINT 的赋值，最后的信号赋值语句在完成进程逻辑功能的同时指定了该进程的传输延时。进程中的传输延时的效果与信号赋值语句中的延时效果相同。

对于下述两个信号赋值语句：

```
AS <= X * Y ;          -- 语句 S1（没有显式指定延时）
BS <= AS + Z ;         -- 语句 S2（没有显式指定延时）
```

如果这两个语句在 VHDL 源代码中按上述顺序出现，自然可以认为这两个语句会按顺序执行。可以想象在语句 S1 中计算得到信号 AS 的新值，而 AS 的新值会被语句 S2 用作输入，即在语句 S2 执行时，AS 的新值将会作为计算 BS 时的输入值。赋值语句的这种执行方式称为顺序执行。

然而，这两个语句还可以用另外的方式理解，即语句 S1 和 S2 也可以作为并发信号赋值语句理解，这就是说，它们会像进程一样，按同步方式执行。S1 的敏感信号表中包括信号 X 和 Y，S2 的敏感信号表中包括 AS 和 Z，按并发执行方式的解释，语句 S1 在 X 或 Y 发生变化的时刻执行，语句 S2 可以在 Z 或 AS 发生变化的时刻执行。由于在信号赋值语句中没有显式指定延时，缺省的延时时间就是 VHDL 中规定的 Delta 延时。如果把 S1 和 S2 全看作是并发执行的信号赋值语句，并假定在 $t = t1$ 时刻 X 和 Z 的值都发生变化，那么 X 值的变化会激活语句 S1，Z 值的变化会激活语句 S2，即语句 S1 和 S2 都在 $t1$ 时刻被激活。由于 Delta 延时的作用，语句 S1 计算得出的信号 AS 的新值要在 $t1$+Delta 时刻才出现在 AS 上，在 $t1$ 时刻语句 S2 中用到的 AS 的值是其 $t1$ 时刻之前的旧值。如果 AS 的新值与旧值不同，那么在 $t1$+Delta 时刻，信号 AS 的值会发生变化。AS 值的变化会再次激活语句 S2，S2 再次执行时会使用在 $t1$ 时刻计算出的信号 AS 的新值。

表 2.2 是用上述并发信号赋值语句解释得到的语句 S1 和 S2 执行的结果。在 $t=t1$ 时刻，X 和 Z 的值都发生变化，X 的变化激活 S1，语句 S1 在 $t1$ 时刻执行，计算出 AS 的新值为 8，但这个新值直到 $t1$+Delta 时刻才会出现在 AS 之上。在 $t1$ 时刻，Z 的变化激活 S2，语句 S2 在 $t1$ 时刻执行计算出 BS 的新值应为 5，但这个新值也要在 $t1$+Delta 时刻才会出现在 BS 上，$t1$ 时刻 BS 仍保持其旧值 2。在 $t1$+Delta 时刻，AS 和 BS 的值都应该更

表 2.2　并发信号赋值语句的执行结果

初　值		$t1$	$t1$+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

新为在 $t1$ 时刻计算得出的新值，即 AS 和 BS 分别更新为 8 和 5。由于这时($t1$+Delta 时刻)AS 的值发生了变化，语句 S2 再次被激活，计算得出 BS 的新值为 11。这个新值将在 $t1+2*$Delta时刻出现在信号 BS 之上。由此看出，由于内建 Delta 延时的作用，并发信号赋值语句的行为与指定了延时的信号赋值语句完全相同。

2.5 顺序执行语句与并发执行语句

如果只根据语句本身，并不能判断它会按顺序执行还是并发执行。因此必须在硬件描述语言的语义上规定什么样的语句会顺序执行，什么样的语句会并发执行。在 VHDL 中规定，如果语句出现在结构体之内，它们会并发执行，如果语句出现在进程或子程序之内，它们会按顺序执行。

利用下面的 VHDL 源代码，可以解释顺序执行和并发执行的概念。实体 STATEMENTS 的输入信号为 X，Y 和 Z，在结构体 CONCURRENT 中，AS 和 BS 被定义为两个信号，AS 是内部信号，BS 是端口信号。两个信号赋值语句 S1 和 S2 恰好出现在结构体内部，按 VHDL 的语义规定，这两个信号赋值语句应并发执行。可以交换这两个语句的先后顺序，交换后不影响仿真结果。

在结构体 SEQUENTIAL 中，两个变量赋值语句 V3 和 V4 出现在进程体内，变量 AV 和 BV 也在该进程内定义，根据 VHDL 语义的规定，这两个变量赋值语句是顺序执行语句。顺序执行语句执行时的先后顺序与它们在源代码中写出的顺序完全相同。在这种情况下，语句 V3 和 V4 的先后顺序不能交换。在进程内部，把运算结果传给了信号 BS，所以在进程外部可以得到运算结果。由这个例子可以看出，对信号的运算，可以用并发语句实现，也可以用顺序执行语句实现。

```
entity STATEMENTS is
    port(X, Y, Z: in Integer; BS: out Integer);
end STATEMENTS;

architecture CONCURRENT of STATEMENTS is
    signal AS: Integer;
begin
    AS <= X * Y;          -- 语句 S1
    BS <= AS + Z;         -- 语句 S2
end CONCURRENT;

architecture SEQUENTIAL of STATEMENTS is
begin
    process(X, Y, Z);
        variable AV, BV: Integer;
    begin
```

```
        AV := X * Y;          -- 语句 V3
        BV := AV + Z;         -- 语句 V4
        BS <= BV
    end process;
end SEQUENTIAL;
```

并发执行和顺序执行的概念不但出现在信号赋值语句中，而且出现在许多其他 VHDL 句法结构中。比如，若所有的进程语句全是直接写在结构体内，则它们一定并发执行。同一个结构体的多个进程语句不按源代码中写出的顺序执行，而是按它们的敏感信号发生变化的先后顺序执行。与信号赋值语句不同的是进程语句只能并发执行，而信号赋值语句既可以顺序执行，也可以并发执行。

2.6 仿真、仿真周期及仿真器中延时的实现

在数字系统仿真时，硬件由进程网络描述，进程对信号的作用有两种：一种是“操作”，另一种是“事件”。如果某进程对信号赋值，不论赋给信号的新值与信号上原来的旧值是否相同，都称对信号进行了“操作”；如果某进程对信号赋值，且赋给信号的新值与信号上原来的旧值不同，则称信号上发生了“事件”。在仿真过程中，仿真时钟从 0 开始向后走，仿真器对进程网络的作用由信号操作序列表示，其作用方式如图 2.5 所示。如果对进程的输入信号有操作，进程对这种操作会有响应。对信号进行操作必须指明三项内容：① 信号名，② 操作发生时刻，③ 信号值。比如，对于图 2.5 所示的例子，假定仿真时钟当前时刻为'0'，进程 P1 的延时为 100ns，信号 B 将要于仿真时钟的 100ns 时刻变为'1'，对信号 B 的操作可以记为(100,B,1)；与此类似，可以得出对信号 C 和 D 的操作(150,C,0)和(175,D,1)。在数字电路仿真器中，通常把由这三项内容指定的对信号的操作也称为事件，并把对信号的操作保存在一个称为“信号操作队列”(也称“事件队列”)的链表中，队列中的内容按对信号操作发生的先后顺序排列。事件队列中的每项内容都是一个数据对(SV,V)，其中 SV 是信号名，V 为要分配给该信号的值。

仿真时钟	事件队列中内容
…	A,1
…	…
100	B,1
…	…
150	C,0
…	…
175	D,1

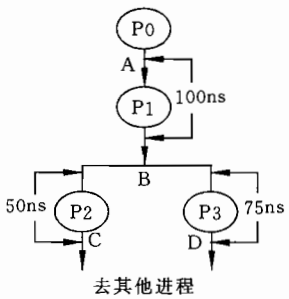


图 2.5 仿真与进程

仿真过程中，随着仿真时钟的增加，在每个时刻，仿真器对事件队列中的当前时刻所发生的所有事件并行进行处理。如果某进程以发生事件的信号为敏感信号，则该进程在这

个仿真时刻被激活。但是，由于仿真所用的计算机大多是单 CPU 的计算机，所以必须使用串行过程来模拟并行操作，在串行过程中，依次处理仿真时钟的同一时刻所发生的所有事件，以及这些事件所激活的所有进程。处理该时刻所发生的所有事件和这些事件所激活的所有进程的全过程构成了一个仿真周期。在一个仿真周期中，首先判断哪些信号上有事件发生，如果发现某信号上发生事件，则被这个信号触发的进程被激活。

在 VHDL 仿真过程中，时间轴是仿真时钟，仿真时钟的坐标是标准时间单位。根据 Delta 延时的定义，任意有限个 Delta 延时相加并不能使仿真时钟的数值增加。VHDL 仿真器中实现了这种延时的机制。考虑 Delta 延时后，在给定的时刻，完整的仿真周期由如下几个步骤组成：

(1) 增加仿真时钟到事件队列中下一项内容所在时刻，如果队列中没有新内容，则停止；否则，转第(2)步。

(2) 不增加仿真时钟，启动一个新的仿真周期，激活所有敏感信号上有事件发生的进程。

(3) 执行被激活的进程，如果需要，则在事件队列中插入新内容，事件队列中的某些新内容可能包含 Delta 延时。

(4) 如果某信号上有事件发生，且该事件由第(3)步中带有 Delta 延时的信号赋值语句产生，则转第(2)步；否则，转第(1)步。

可以看出，在仿真时钟的同一时刻，可能伴随有多个由 Delta 延时产生的仿真周期(或子仿真周期)。在仿真时钟的同一时刻，所执行的仿真周期(或子仿真周期)的时间间隔为 Delta。这样 Delta 延时带来的效果就是，使仿真进入一个新的仿真周期(或子仿真周期)而不增加仿真时钟。例如，对于信号赋值语句 Y <= X，语句执行结束之后，Y 的值并不改变，旧值一直维持到当前仿真周期结束，在下一仿真周期(或子仿真周期)开始时，Y 上的值才得到前一仿真周期(或子仿真周期)中计算得到的新值。如果在当前仿真周期内其他语句要用到 Y 的值，无论该语句在源代码中出现在该语句之后，还是出现在它之前，所使用的都是 Y 的旧值。在语句信号赋值语句 Y<= X after 10ns 中，信号 Y 在当前仿真时钟后 10ns 的那个仿真周期内才得到新值。显然，这种情况下，从仿真时钟当前时刻直到其后 10ns 之间用到 Y 值时，使用的全是 Y 的旧值。

解释 Delta 延时的另外一种方法是用宏时间表示仿真时钟，用微时间表示 Delta 延时。在宏时间的一个时间单位内，可以包含有无穷多个微时间的增量。图 2.6 表示了这一概念。在图中的第一点，仿真时钟(宏时间)为 15ns，在这一时刻，执行了两个要求 Delta 延

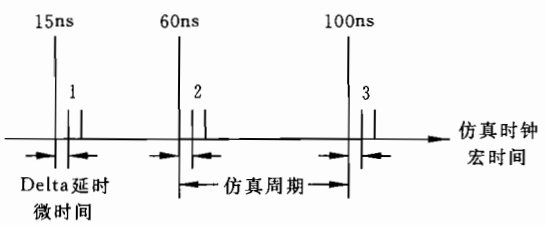


图 2.6 宏时间与微时间

时的仿真周期或子仿真周期(微时间),但仿真时钟不变。接下来的仿真周期将仿真时钟增加到60ns,在这一时刻仍执行了两个要求Delta延时的仿真周期或子仿真周期。下一时刻的仿真时钟为100ns,在100ns处的情况与前两个时刻的情况相同。

在系统造型时采用不同的延时模型可以得到三种不同的仿真模型。

(1) 只考虑Delta延时:用来进行模型的纯功能验证。

(2) 只考虑标准时间单位延时:用来验证系统时序。

(3) 同时考虑两种延时:Delta延时用于延时较小的器件,标准单位延时用于延时较大的逻辑模块。这种混合延时模型常用于研究系统时序。

为了说明不同的延时模型,下面给出了实体BUFF的行为的不同描述。实体BUFF是一个简单缓冲单元,输出Z是输入X延时后的值。在结构体ONE中,输入X在零延时后被复制到变量Y1,信号Z在1ns后得到Y1的值。信号的X传输总延时为1ns。在结构体TWO中,信号Y2在Delta延时之后得到X的值,输出Z在另一个Delta延时之后得到Y2的值,信号通过结构体TWO的总延时为2 * Delta。在结构体THREE中,Y3在Delta延时之后得到X的值,再经1ns之后,Z变为Y3的值,总延时是1ns或者说是1ns+Delta。在结构体FOUR中,总延时为2ns。

```
entity BUFF is
    port(X: in Std_logic; Z: out Std_logic );
end BUFF;

architecture ONE of BUFF is
begin
    process(X)
        variable Y1: Std_logic;
    begin
        Y1 := X;
        Z <= Y1 after 1ns;
    end process;
end ONE;

architecture TWO of BUFF is
    signal Y2: Std_logic;
begin
    Y2 <= X;
    Z <= Y2;
end TWO;

architecture THREE of BUFF is
```

```
    signal Y3: Std_logic;
begin
    Y3 <= X;
    Z <= Y3 after 1ns;
end THREE;

architecture FOUR of BUFF is
    signal Y4: Std_logic;
begin
    Y4 <= X after 1ns;
    Z <= Y4 after 1ns;
end FOUR;
```

对于同一输入信号 X，图 2.7 给出了这四个 VHDL 结构体所产生的输出。对应于结构体 ONE，TWO，THREE 和 FOUR，图中分别用 Z1，Z2，Z3 和 Z4 标出了它们的输出。

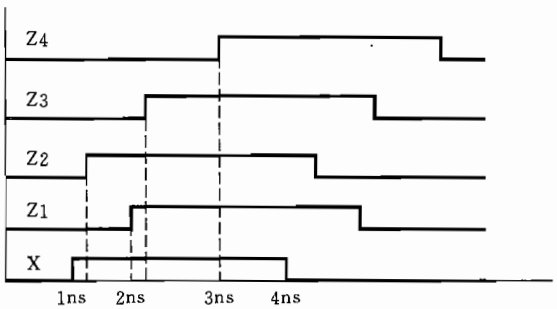


图 2.7　缓冲器 BUFF 的输出

如果信号赋值语句出现在进程之内，特别要注意延时的作用。下面给出了实体 BUFF 的另一种行为描述，即结构体 FIVE。对于这个结构体，如果 X 的值在 $t1$ 时刻发生变化，那么 X 的值在 $t1$+Delta 时刻赋给信号 Y5，在同一个仿真周期内，赋给 Y5 的值是其旧值。由于进程只对 X 的变化敏感，所以该进程只执行一次。如果 X 的值不再发生变化，则 $t1$ 时刻 X 值的变化永远也不会出现在输出端 Z 上。为了改正这种设计错误，可以对结构体 FIVE 做一点修改。结构体 FIVE_A 是修改后的结构体，修改后将信号 Y5 也放在进程的敏感信号表中。这样，在 $t1$+Delta 时刻，Y5 的变化会再次激活进程，其结果是在 $t1+2*$Delta 时刻，输出信号 Z 变为输入信号 X 的值。图 2.8 给出了结构体 FIVE 和结构体 FIVE_A 的输出结果。图中 Z5 和 Z5A 分别表示结构体 FIVE 和结构体 FIVE_A 的输出。

```
architecture FIVE of BUFF is
begin
    process(X)
        signal Y5: Std_logic;
```

```
    begin
        Y5 <= X;
        Z <= Y5;
    end process;
end FIVE;

architecture FIVE_A of BUFF is
    signal Y5: Std_logic;
begin
    process(X,Y5)
    begin
        Y5 <= X;
        Z <= Y5;
    end process;
end FIVE_A;
```

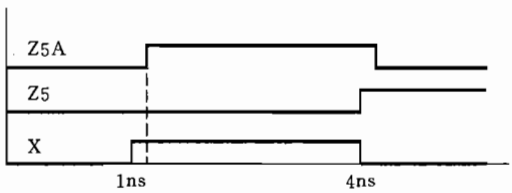


图 2.8　缓冲器 BUFF 错误的结构体及其修改后的输出

事实上，将结构体 FIVE 中的进程语句去掉，换成两个并发信号赋值语句，可以得到结构体 FIVE_B。结构体 FIVE_B 与 FIVE_A 效果相同。在下述的结构体 FIVE_B 中，信号 X 发生变化时，第一个语句开始执行，第一个语句的执行后再经 Delta 延时，Y5 会发生变化，这会进一步引起第二个语句的执行。第二个信号赋值语句的执行将在另一个 Delta 延时之后把 X 的值传递给 Z。

```
architecture FIVE_B of BUFF is
    signal Y5: Std_logic;
begin
    Y5 <= X;
    Z <= Y5;
end FIVE_B;
```

在 VHDL 中，信号延时的描述可以有两种形式：一种是传输延时，另一种是惯性延时。传输延时的一个典型例子是传输线上的延时，其输入端的信号经过一定的延时时间之后在其输出端完全再现。惯性延时的特点是对于脉宽很窄的脉冲信号，不能在输出端得到响应，即窄脉冲信号传播不到输出端。下面是惯性延时和传输延时的两个例子：

```
Z <= I after 10ns;                    --惯性延时
Z <= transport I after 10ns;          --传输延时
```

前一个语句指定了惯性延时，就是说当驱动信号 I 的值保持足够长的时间后，才能传播到信号 Z。按 VHDL 语言的规定，所谓足够长的时间指的就是关键字 after 之后给定的时间。按这种规定，信号 I 的值只有保持 10ns 或 10ns 以上的时间，才会影响到信号 Z 的值。后一个语句是传输延时，按这种延时机制，不论信号 I 上的变化持续时间的长短，信号 I 上的任何变化都会延时 10ns 之后出现在信号 Z 上。

惯性延时滤除了输入信号脉宽较窄的成分，比传输延时更准确地反映了实际电路中的情况。在实际电路中，信号的逻辑值通常由电路中的节点电压所表示。由于电容的影响，节点电压不能实时变化。为了使电路中节点电压从一个电平转换到另一个电平，驱动信号的能量必须足够大，且要保持足够长的时间。在 VHDL 中，缺省的延时方式是惯性延时。传输延时主要用于高层次抽象模型。如果要用传输延时，必须在赋值语句中加上关键字 transport。

2.7 硬件的行为描述和条件语句

对于大部分数字系统，都可以划分为控制单元和数据单元两个组成部分。图 2.9 示意给出了这种划分方法。通常，控制单元的主体是一个有限状态机，它接收外部信号以及数据单元产生的状态信息，产生控制信号序列，用来决定何时进行何种数据处理。数据单元是完成数据处理所需要的电路元件，包括组合电路构成的数据处理器、保存运算数据和运算结果的数据寄存器等。它从控制单元接收控制信号序列，对数据进行相应处理，产生系统的输出以及控制单元所需的状态信号。

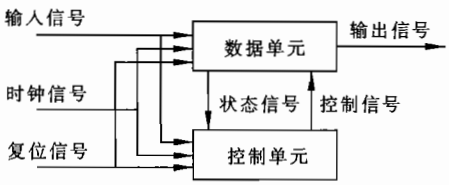


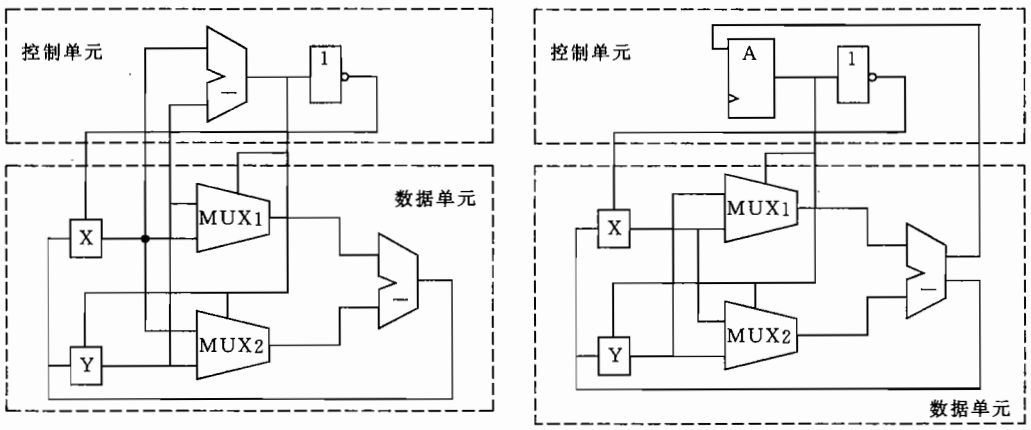
图 2.9 数字系统的划分

尽管硬件的 VHDL 模型中“实体-结构体对”与硬件本身并不存在直接对应关系，但硬件描述的风格会影响到硬件结构，这种影响与综合器本身的特征有关。使用 VHDL 构造数字系统的模型时，必须考虑到综合器的综合效果。条件语句是影响硬件结构的重要语法元素之一。例如，if-elseif-else-end if 和 case-when-end case 是 VHDL 硬件造型中常用的条件语句。对于下述条件语句，不同的综合工具可能会得到完全不同的硬件。

```
if (X < Y)
    then Y := Y - X;
else
```

```
        X := X - Y;
    end if;
```

图 2.10 即为该 VHDL 源代码的两种不同实现。在图 2.10(a)中，在控制单元中判断条件 X<Y 还是 X>Y，并决定资源的利用。在控制单元中由加(减)法器计算寄存器 X 和 Y 中的数值之差，由该差值的正负决定多路选择器 MUX1 和 MUX2 选择哪一个数据进行运算，并控制运算结果存储在哪个寄存器中。在图 2.10(b)中，在数据单元中计算条件，



(a)　　(b)

图 2.10　条件语句结构的不同实现

并确定资源的使用情况。第一个时钟节拍中，在数据单元中计算 X 和 Y 的差，并将该差值送入控制单元中的寄存器 A，在后续的时钟节拍，由 A 值的正负决定如何进行数据运算。一般情况下，由于条件结构的出现，在控制单元部分及数据单元中均需要增加部分电路。控制单元部分需要增加的电路如下：

(1) 在有限状态机中需要额外的操作。

(2) 如果条件表达式中包含数据单元中使用的数据对象，则需要额外的输入信号，比如图 2.10(a)中的信号 X 和 Y 的状态控制信号就是额外的输入信号。

(3) 如果条件复杂，则控制单元中需要代数或逻辑运算单元。

在数据单元中需要增加的电路结构如下：

(1) 一个对象有两个以上的输入时，需要额外的连接(在此电路中为多路选择器)。比如图 2.10(a)和(b)中的多路选择器 MUX1 和 MUX2。

(2) 运算单元需要改变。比如在图 2.10(b)中，条件 X<Y 通过运算器作减法得到，一般情况下，计算条件需要额外的时钟周期。

条件结构通常会导致几个互不相容的分支。在硬件设计的一个执行路径上，根据计算得到的条件值，只能有一个分支得到执行。对于给定的综合器，VHDL 模型中条件分支语句的风格严重影响综合后的电路性能，电路设计者必须认真考虑在什么条件下，如何使用这些条件分支语句。下面的例子说明了不同 VHDL 条件语句的差别。

```
-- 第一种描述
if COND1 then
    b := a * d;
elseif COND2 then
    b := a * c;
end if;

-- 第二种描述
if COND1 then
    b := a * d;
else
    if COND2 then
        b := a * c;
    endif
end if;

-- 第三种描述
if COND1 then
    b := a * d;
end if;
if COND2 then
    b := a * c;
end if;
```

假定条件 COND1 和 COND2 完全互不相容，且不会同时出现，则三种条件语句的行为完全相同。但是，三种条件语句会得到完全不同的硬件实现。在第一种描述得到的硬件中，根据条件{COND1，COND2}的不同，在两个完全不同的执行路径中切换。从执行时间考虑，会得到最优的设计（需要一个控制状态就可以完成运算）；从硬件资源的占用来讲，也会得到最优的硬件设计（只需要一个乘法器）。在第二种描述得到的硬件中，仍然可以共享乘法器，但执行时间与调度算法有关，可能需要一个控制状态，也可能需要两个控制状态。事实上，某些调度算法会把满足每个条件的可执行语句调度到不同的控制状态上执行。对于第二种描述，控制单元中的有限状态机中应该产生三个控制状态，分别处理三个不同条件{COND1，(not COND1 and COND2)，(not COND1 and not COND2)}。至于第三种描述，控制单元需要产生四种不同状态，分别对应条件{(COND1 and COND2)，(COND1 and not COND2)，(not COND1 and COND2)，(not COND1 and not COND2)}。在现在的高层次综合工具中，大部分都不能够将{COND1 and COND2}识别为 false。如果不能够消除这一个条件下的数据操作，则第三种描述需要两个乘法器。这种情况下，如果调度时约束条件为："只能使用一个乘法器"，则只能将两个数据运算调度到不同的控制状态。

2.8 同步语句

在高层次硬件描述中，wait 语句是并发语句同步的重要语法元素。从语法的角度来讲，wait 语句的基本形式为 wait，sensitivity_clause，condition_clause，timeout_clause。其中 sensitivity_clause 定义了一个信号集合，wait 语句对这些信号敏感。即如果集合中的任何信号上有事件发生，wait 语句都会使其控制的进程被激活。例如，wait 语句：wait on S1 until S2；控制的进程被执行的条件为信号 S1 上有事件发生，且信号 S2 的新值必须为 true；并非所有的综合工具都支持这种类型的 wait 语句。

condition_clause 用来为 wait 语句的执行定义了一个条件。在这个条件为 true 时，wait 语句控制的进程才能执行。为了保证不同进程之间的显式同步，必须定义这个条件。例如，对于下述 VHDL 源代码中的两个进程 P1 和 P2，P2 必须等待 P1 为信号 DATA_READY 赋值之后才能执行。现在几乎所有综合工具全支持这种条件 wait 语句，有少数的综合工具限制条件信号必须为时钟信号。

```
P1:process
begin
    -- 其他语句
    DATA_READY <= true;
    -- 其他语句
end process P1;

P2: process
begin
    -- 其他语句
    wait until DATA_READY;
    -- 其他语句
end process P2;
```

从硬件综合的角度来看，wait 语句有可能是同步 wait 语句或异步 wait 语句。同步 wait 语句只在时钟的边沿执行。典型的同步 wait 语句如下：

```
wait until condition and rising_edge(CLK);
```

异步 wait 语句则可以独立于时钟执行。对于高层次综合工具来讲，异步 wait 语句的处理相对比较困难。事实上，行为综合产生的 RTL 模型中，所有 wait 语句全部为同步 wait 语句。如果在高层次行为描述中使用了异步 wait 语句，两种模型的对应就会变得困难。

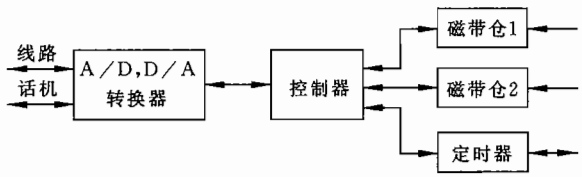
wait 语句中的 timeout_clause 指定了进程被抑制的时间区间，时间区间由绝对时间给定。有些综合工具会忽略这个条件，原因在于它不能保证实现的硬件完全满足这些显式指定的时间要求。在 VHDL 中，一个 wait 语句指定了硬件行为描述中的一个控制状态。

在进程模拟过程中，每当执行到一个 wait 语句，进程就会挂起等待，直到满足 wait 语句中指定的条件被满足后再开始重新执行。在行为综合中，各 wait 语句通常被解释成一些中断点，它会使控制单元的有限状态机产生新的控制状态。硬件设计者在硬件的行为描述中写出一个 wait 语句，意味着建立了一个控制状态，两个相邻 wait 语句之间的可执行代码构成一个执行路径。在 wait 语句与控制状态等效，wait 语句之间的可执行语句与执行路径等效的意义上，硬件的行为描述与一个有限状态机等效。事实上，VHDL 行为描述中唯一的可观测点是计算信号值的 wait 语句。两个 wait 语句之间的代码中还可能包含复杂的控制语句，比如循环语句，在调度时这些语句可以进一步划分为不同的控制状态。

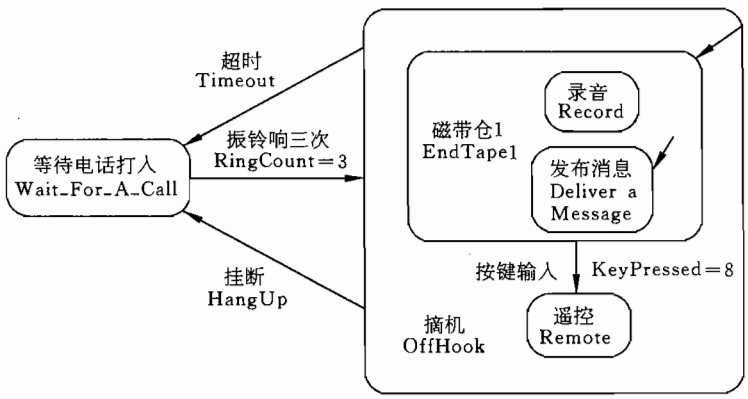
2.9 循　　环

与一般的程序设计语言类似，VHDL 也支持 loop，while 和 for 等循环语句。与循环语句同时使用的还有 next 和 exit 语句。next 语句使得执行过程跳转到循环语句的开始处开始执行，exit 语句使得可以在循环体内部多点跳出。

循环可以用来表示复杂行为。第 1 章中提到的电话答录机可以用来说明这个问题。图 2.11(a)是电话答录机的方框图，系统由一个控制器、两个磁带仓、一个定时器和与电话系统的接口构成。图 2.11(b)是其控制器的工作状态的层次示意图。系统加电后，进入的状态为等待电话呼入状态(Wait_For_A_Call)，在这一状态下，系统被初始化并等待出现三声振铃(Get_3_Ring)。如果出现三声振铃声，则系统变换为摘机状态(OffHook)。在摘机状态下，答录机给电话呼入者发送一段声音，比如，“主人不在，请留言”等。这时，电话呼入



(a) 系统方框图



(b) 控制器模型

图 2.11　电话答录机

者有三种可能的动作:① 挂机,② 留言,③ 遥控回放过去的留言或者更改留给呼入者的声音。如果呼入者选择挂机,则系统立即恢复到 Wait_For_A_Call 状态。如果呼入者保持摘机状态超过了时间限制(Timeout),则系统也恢复到 Wait_For_A_Call 状态。

上述控制器可以用含有 6 个循环语句的 VHDL 代码描述,其总体结构如下:

```
(loop behavior
    (loop Wait_For_A_Call
        (loop Get_3_Ring)
    )
    (loop OffHook
        (loop Get_3_Digits)
        (loop ManualControl)
    )
)
```

为了清楚起见,这里没有给出控制器完整的 VHDL 描述,而是集中讨论读取三个数字输入的一个循环(Get_3_Digits)。在这个循环中,系统接收呼入者通过电话双音频拨号键盘输入的 3 位密码数字,以判断呼入者是否有权遥控操作电话答录机。这部分的 VHDL 代码如下:

```
-- 循环 OffHook 之外的其他语句集合
OffHook: loop
    Get_3_Digits: loop
        wait until (KeyPressed = 0)
        Timeout := FU_REM(ElapsedTime, 20);
        wait until ((ElapsedTime = Timeout) or
                    (KeyPressed /= 0) Or (HangUp = '1');
        if ((HangUp = '1') or (ElapsedTime = Timeout))
            then exit OffHook;
        end if;
        nextDigit := PasswdROM(DigitCount);
        if (KeyPressed /= NextDigit)
            then FalsePasswd :=1;
        end if;
        DigitCount := DigitCount + 1;
        if (DigitCount = 3)
            then exit Get_3_digits;
        end if;
    end loop Get_3_Digits;
-- 循环 Get_3_Digits 之外的其他语句
```

```
end loop OffHook;
-- 循环 OffHook 之外的其他语句
```

上面只给出了 6 个循环之中的一个。在其中任何一个循环执行过程中，如果出现意外情况，比如呼入者挂断电话，则循环的各个层次都必须保证能返回到循环 Wait_For_A_Call 之中。在控制器行为描述中，各个循环中的每个 wait 语句中，对这些意外情况都进行了检查。

2.10 过程与函数

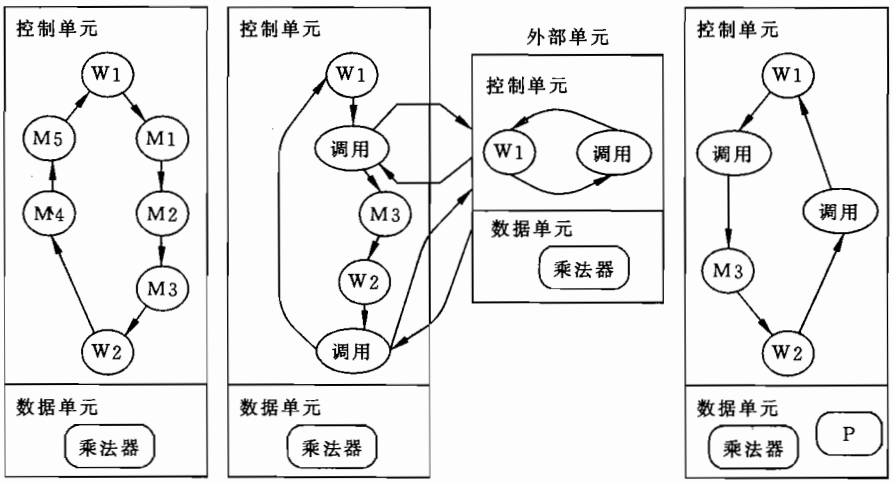
过程和函数是 VHDL 中的子程序，用以将 VHDL 代码中的公共部分隔离出来，以简化硬件的 VHDL 描述。过程和函数的主要区别有两点：① 函数有返回值而过程没有；② 过程可以有多个输出值，而函数只能有一个输出值。与其他程序设计语言类似，在过程和函数中定义和使用的变量只在其内部有效，过程和函数内部可以嵌套调用其他过程和函数。从硬件综合的角度来看，过程和函数的处理则与一般程序设计语言不同。例如，对于下述 VHDL 源代码，可以有三种不同的处理方法。

```
P1:process
--
    procedure P(A: in Integer; B: out Integer);
    begin
        C := A * D;
        B := A * C;
    end P;
begin
    wait until Rising_edge(CLOCK);
    P(X, Y);
    Z :=  Y * K;
    wait until Rising_edge(CLOCK)
    P(Z, Y)
end process P1;
```

第一种方法是将内部调用完全展开，对进程或函数的调用由相应的源代码本身替代。这种处理不但使行为本身的描述变得复杂，而且会使综合后的硬件结构变得复杂。完全展开对硬件实现的主要影响表现在控制单元中，展开后的运算单元则可以被共享。下面进程 P1_1 由进程 P1 完全展开得出，图 2.12(a)是对进程 P1_1 调度后得到的硬件结构，可以看出，控制单元中有 5 个运算状态和 2 个等待状态，相对较为复杂，而硬件的数据单元则只包含一个乘法器，相对较为简单。

```
P1_1: process
--
```

```
    procedure P(A: in Integer; B: out Integer)
    begin
        C := A * D;
        B := A * C;
    end P;
begin
    wait until Rising_edge(CLOCK);          -- W1
    C := X * D;                             -- M1
    Y := X * C;                             -- M2
    Z :=  Y * K;                            -- M3
    wait until Rising_edge(CLOCK);          --W2 P(Z, Y)
    C := Z * D ;                            -- M4
    Y := Z * C;                             -- M5
end process P1_1;
```



(a) 过程完全展开　　(b) 过程作为独立模块　　(c) 过程作为协处理器

图 2.12　进程或函数不同处理举例

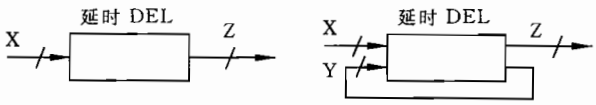
第二种方式如图 2.12(b)所示，被调用的过程或函数被编译成独立的模块。这种情况下，过程或函数的调用被通信协议所替换，主模块和外部模块(过程或函数编译成的独立模块)之间按照通信协议传递参数。

第三种处理方式如图 2.12(c)所示。这种方式下过程和函数被编译成数据单元中的模块，称之为协处理器。这些协处理器在系统设计时被重复使用。

第3章 基本逻辑单元的VHDL模型

VHDL提供了对各种数字电路进行造型、对电路进行仿真和综合的语法结构。在集成电路设计过程中，VHDL主要用在仿真和综合两个方面，设计者必须了解如何用VHDL描述电路，并了解VHDL语言语法结构对应的综合结果。

图3.1示意给出组合逻辑电路与时序逻辑电路的基本结构。如果电路的输出信号Z



(a) 组合逻辑电路　　(b) 时序逻辑电路

图3.1　组合逻辑与时序逻辑电路

表3.1　基本电路单元

电路类型	典型单元
组合逻辑电路	简单逻辑门(simple logic gate) 复杂逻辑门(composite logic gate) 三态缓冲器(tri-state buffer) 多路选择器(multiplexer) 译码器(decoder) 编码器(encoder) 比较器(comparator) 移位器(shifter) 加法器(adder) 乘法器(multiplier) 算术逻辑单元(ALU) 可编程逻辑阵列(PLA)
时序逻辑单元	触发器(flip-flop) 锁存器(latch) 计数器(counter) 有限状态机(FSM) 扫描触发器(scan flipflop)
存储器	只读存储器(ROM) 随机存取存储器(RAM) 寄存器阵列(register file) 堆栈(stack)

只是其输入信号 X 的函数，即 Z=F(X)，则该电路是组合逻辑电路。组合逻辑电路由基本逻辑门互联形成，其电路的网络图是单向图，不存在反馈回路。时序逻辑电路带有反馈，输出对输入信号 X 的响应可分为两类：第一类为输出 Z，第二类为状态变量 Y。状态变量 Y 被反馈到电路的输入端，输出 Z 与输入 X 有关，也与状态变量 Y 有关。由于存在反馈，所以时序逻辑电路有记忆特性。存储器是常用的逻辑电路，但由于它们的特殊性，本书中把它们单独算做一类电路。表 3.1 列出了以上三类电路的主要单元。

本章的 3.1～3.3 节讨论用 VHDL 描述基本电路模块，包括组合逻辑单元、时序逻辑单元和存储器件。对于这部分内容，主要从综合的角度进行论述，也兼顾仿真的问题。接下来论述模型精度的问题。为了提高 VHDL 造型的精度，主要有三方面的工作：① 以某种统一的方式建立精确的时域模型，从而更好地描述电路的延时特性；② 计算延时，根据门级网单的负载信息动态地计算电路延时；③ 用多值逻辑描述电路状态，从而取得更加精确的仿真结果。3.4 节介绍了前两个方面的基本原理，3.5 节介绍的 VITAL 是关于这两方面工作标准化的进展，3.6 节将论述如何用多值逻辑描述电路。

3.1 组合逻辑电路的造型

用 VHDL 描述组合逻辑电路有两种方法，即使用并行语句或使用进程。使用进程可以大大提高程序的可读性，特别是在算法比较复杂的情况下，例如乘法器，通常只有使用进程才能够处理。但是使用进程时必须注意源代码书写风格，以免综合后形成时序逻辑电路，不必要地增加电路规模。在本章中，这两种基本方法都会被用到，与这两种方法对应的结构体分别以 DF 和 ALG 命名。

3.1.1 逻辑门

逻辑门如与门、非门、或非门、异或门、与或非门等是最基本的组合逻辑单元。下面的 VHDL 源代码是 2 输入与门的模型。该模型计算两个输入信号 I1 和 I2 的逻辑与，延时 DEL 后将与运算的结果送到输出 O。这段代码的综合结果就是一个与门。综合时，用 after 描述的延时通常被忽略。这是因为综合时可以通过设置综合约束条件来控制综合工具选择相应的器件，但不可能精确到与源代码中的定义完全相同。因此，这里的延时语句只在仿真时有意义。

```
entity AND2 is
    generic(DEL: TIME);
    port(I1,I2 : in Std_logic; O: out Std_logic);
end AND2;

architecture DF of AND2 is
begin
    O <= I1 and I2 after DEL;
end DF;
```

上面这段 VHDL 源代码可以改写为使用进程的等价方式，如下面的代码所示。虽然在代码中使用了进程，但综合工具可以检验到 if 的两个分支上进行的都是数据操作，不需要储存电路状态，所以综合后仍然形成组合逻辑电路。这种描述风格的好处是可以分别给出从输入变化到输出'1'或输出'0'的延时，提高了造型精度。现在的 VLSI 设计中，通常在布局布线得到物理版图之后，根据物理版图计算出精确延时参数，并把延时参数反向标注(back-annotate)到原始的输入描述，然后再进行仿真，以验证电路的正确性，这时宜采用下面的代码风格。

```
entity AND2 is
    generic(DEL_1, DEL_2: TIME);
    port (I1,I2: in Std_logic; O: out Std_logic);
end AND2;

architecture DF of AND2 is
begin
    AND2: process(I1, I2)
    begin
        if I1 = '1' and I2 = '1' then
            O <= '1' after DEL_1;
        else
            O <= '0' after DEL_2;
        end if;
    end process AND2;
end DF;
```

读者可以自己完成其他逻辑门的描述。综合工具的标准单元库中，常会有一些复杂逻辑门，例如与或非门。这些标准单元具有预制版图，集成电路设计中使用这些标准逻辑门，可以综合出面积较小的芯片。

3.1.2 三态缓冲器

三态缓冲器的作用是缓冲数据、增强驱动能力以及把功能模块与总线相连接。如果缓冲器的使能端 EN 为'1'，则缓冲器的输入端 I 的信号值被复制到输出端；如果缓冲器的使能端 EN 为其他数值，则缓冲器的输出端为高阻态。其输出可以用线与的方式和其他缓冲器的输出接在一起，下面的 VHDL 源代码给出了缓冲器的模型。IEEE 1164 标准程序包中用'Z'表示高阻态。现在的 EDA 综合工具一般都能够根据这种描述综合得到三态器件，这就是所谓的"三态推断"。对于大部分的应用情况，这种行为模型已足够精确。在使用总线互连方式时，与总线通信的器件通常要通过三态缓冲器与总线相连。

```
entity BUF is
    generic(DATA_DEL, Z_DEL: TIME);
```

```
    port (I, EN : in Std_logic; O: out Std_logic);
end BUF;

architecture ALG of BUF is
begin
    Tri_state_buf: process(I, EN)
    begin
        if EN = '1' then
            O <= I after DATA_DEL;
        else
            O <= 'Z' after Z_DEL;
        end if;
    end process Tri_state_buf;
end ALG;
```

3.1.3 多路选择器

在数字系统设计时,需要从多个数据源中选择一个,这时就需要用到多路选择器。下面的 VHDL 源代码是四选一电路的模型,被选择数据字的宽度为 4。模型 DF 中使用了条件赋值语句,所以写得很简短。模型 ALG 是与其等价的进程描述形式。由于使用了 Std_logic 和 Std_logic_vector 的数据类型,SEL 可能的数值不止四种,所以模型 DF 和 ALG 中各有一个分支处理无效数值,其中 NULL 语句的作用是不进行任何操作。在综合时,EDA 综合工具一般忽略这一分支。除了三态器件之外,综合工具对 Std_logic 和一般 Bit 数据类型采用完全相同的处理方法。

```
entity FOUR_TO_1_MUX is
    generic(DEL: TIME);
    port (IN0, IN1, IN2, IN3 : in Std_logic_vector (3 downto 0);
          SEL: in Std_logic_vector (1 downto 0);
          O: out Std_logic_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 when SEL = "11" else
         NULL;
```

```
end DF;

architecture ALG of FOUR_TO_1_MUX is
begin
    MUX: process(SEL, IN0, IN1, IN2, IN3)
    begin
        case SEL is
            when "00" => O <= IN0;
            when "01" => O <= IN1;
            when "10" => O <= IN2;
            when "11" => O <= IN3;
            when others => NULL;
       end case;
    end process MUX;
end ALG;
```

3.1.4 译码器

译码器的输入为 N 位二进制代码，输出为 2^N 个表征代码原意的状态信号，即输出信号的 2^N 位中有且仅有一位有效(本例为'1')。对于二—十进制译码器，输出中'1'出现的位置与输入信号所表示的十进制数相同。常见的译码器用途是把二进制表示的地址转换为单线选择信号。下面的 VHDL 源代码是译码器的模型。

```
entity TWO_TO_4_DEC is
    generic(DEL: TIME);
    port (I : in Std_logic_vector(1 downto 0);
          O: out Std_logic_vector (3 downto 0) );
end TWO_TO_4_DEC;

architecture ALG of TWO_TO_4_DEC is
begin
    Decoder: 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;
            when others => NULL;
```

```
        end case;
    end process Decoder;
end ALG;
```

3.1.5 编码器

编码器的行为是译码器的逆过程，它把 2^N 个输入转化为 N 位编码输出。有的编码器要求输入信号的各位中最多只有一位有效(本例为'1')，且规定如果所有输入位全无效(本例为'0')时，编码器输出某个指定状态。编码器的用途很广，使用最多的情况是键盘编码。下面的 VHDL 源代码是一种优先编码器的行为模型，在这个模型中，为输入信号的各位设定了优先级，第 0 位输入信号的优先级最高，第 3 位信号的优先级最低。

```
    entity FOUR_TO_2_ENC is

    generic(DEL: TIME);
    port (I : in Std_logic_vector (3 downto 0);
          O: out Std_logic_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 when I(3) = '1' else
         NULL;
end DF;
```

3.1.6 比较器

比较器的作用是比较两个输入数据的数值大小，由于表示数值的方法有很多种，因此实现比较器的方式也很多。下面 VHDL 源代码定义了一种比较器，它对两个以原码表示的 4 位正整数 A 和 B 进行比较，G='1'表示 A>B；E='1'表示 A=B；L='1'表示 A<B。这段程序用循环语句依次检查 A，B 的每一位。如果在较高位上，A(I)>B(I)或 A(I)<B(I)，则已经能够判别 A，B 的关系，用 exit 语句退出循环；否则，继续检查较低位。这种比较器的结构比较简单，但如果被比较数据的字长较大，计算时间会很长。

```
entity Comparator is
    generic(DEL: TIME);
    port (A, B : in Std_logic_vector (3 downto 0);
```

```
        G, E, L: out Std_logic_vector (1 downto 0) );
end Comparator;

architecture ALG of Comparator is
begin
    Comparing: process(A,B)
    begin
        for I in 3 downto 0 loop
            if A(I) = '1' and B(I)= '0' then
                G <= '1'  after DEL;
                E <= '0'  after DEL;
                L <= '0'  after DEL;
                exit;
            elseif A(I)= '0'  and B(I)= '1' then
                G <= '0'  after DEL;
                E <= '0'  after DEL;
                L <= '1'  after DEL;
                exit;
            else
                G <= '0'  after DEL;
                E <= '1'  after DEL;
                L <= '0'  after DEL;
            end if;
        end loop;
    end process Comparing;
end ALG;
```

3.1.7 移位器

数据的移位是很重要的操作。在一定条件下,右移意味着被 2 除,左移意味着乘以 2。下面的 VHDL 源代码给出了移位器的行为模型。如果 SR='1'且 SL='0',将输入信号右移一位后赋给输出信号;如果 SR='0'且 SL='1',则将输入信号左移一位后赋给输出信号;对于 SR 和 SL 的其他两种输入模式,将输入信号直接赋给输出信号。信号 IL 和 IR 是左移操作(或右移操作)时在输入数据右端(或左端)补上的数据。使用"&"进行移位操作是常用的处理方法,但在 1993 年修订的 VHDL 语言中,具备了移位语句,可以直接实现移位操作。

```
entity SHIFTER is
    generic(DEL: TIME);
```

```
    port (DATA_IN : in Std_logic_vector (3 downto 0);
          SR, SL, IR, IL : in Std_logic;
          DATA_OUT : out Std_logic_vector (3 downto 0) );
end SHIFTER;

architecture ALG of SHIFTER is
begin
    Shift_It: process (SR, SL, DATA_IN, IR, IL)
      variable CON: Std_logic_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 down to 0) & IL after
        DEL;
        when "10" => DATA_OUT <= IR & DATA_IN(3 down to 1) after
        DEL;
        when "11" => DATA_OUT <= DATA_IN after DEL;
      end case;
    end process;
end ALG;
```

3.1.8 加法器

加法器是最基本的运算单元。加法器中最小的单元是全加器，全加器中有两个数据输入端 A 和 B，一个进位输入端 CI，一个和数输出端 SUM 和一个进位输出端 COUT。下面的 VHDL 源代码是全加器的行为模型。

```
entity FULL_ADDER is
    generic(SUM_DEL, CARRY_DEL: TIME);
    port (A, B, CI: in Std_logic; SUM, COUT : out Std_logic);
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 ALG;
```

用以上的全加器级联形成加法器是最简单的实现方式，这种电路每个位单元的结构

都相同。但是在操作数的字长较大时，由于进位要经过多次传递，限制了这种电路的速度，且和数的各位产生的时刻也不相同。为提高运算速度，可以采用行波进位加法器(ripple-carry adder)。行波进位加法器的原理实际上与全加器级联相同，只是换成从进位角度考虑加法问题。下面是一个 4 位行波进位加法器的 VHDL 模型。其中，A，B 为两个输入操作数，SUM 为相加结果，COUT 和 CI 分别是进位输出和输入。

```
entity RIPPLE_CARRY_ADDER is
    generic(DEL1, DEL2, DEL3, DEL4: TIME);
    port ( A, B: in Std_logic_vector(3 downto 0);
           CI: in Std_logic;
           SUM: out Std_logic_vector(3 downto 0);
           COUT: out Std_logic);
end RIPPLE_CARRY_ADDER;

architecture DF of RIPPLE_CARRY_ADDER is
    signal G, P, C : Std_logic_vector(3 downto 0);
begin
    P(0) <= A(0) xor B(0) after DEL1; P(1) <= A(1) xor B(1) after DEL1;
    P(2) <= A(2) xor B(2) after DEL1; P(3) <= A(3) xor B(3) after DEL1;
    G(0) <= A(0) and B(0) after DEL2; G(1) <= A(1) and B(1) after DEL2;
    G(2) <= A(2) and B(2) after DEL2; G(3) <= A(3) and B(3) after DEL2;
    C(0) <= G(0) or (P(0) and CI) after DEL3;
    C(1) <= G(1) or (P(1) and G(0)) or (P(1) and P(0) and CI) after DEL3;
    C(2) <= G(2) or (P(2) and G(1)) or (P(2) and P(1) and G(0)) or
            (P(2) and P(1) and P(0) and CI) after DEL3;
    C(3) <= G(3) or (P(3) and G(2)) or (P(3) and P(2) and G(1)) or
            (P(3) and P(2) and P(1) and G(0)) or (P(3) and P(2)
              and P(1) and P(0) and CI) after DEL3;
    COUT <= C(3);

    SUM(0) <= A(0) xor B(0) xor C(0) after DEL4;
    SUM(1) <= A(1) xor B(1) xor C(1) after DEL4;
    SUM(2) <= A(2) xor B(2) xor C(2) after DEL4;
    SUM(3) <= A(3) xor B(3) xor C(3) after DEL4;
end DF;
```

3.1.9 乘法器

乘法和除法是数字系统中的重要基本运算，许多高级运算如倒数、平方根、指数、三角

函数等都与其有关。实现乘除法有多种算法，现在仍有许多学者致力于新算法的研究。为实现乘除法指令，一般有下列一些方法：

(1) 用软件实现乘除法运算。在已有加法器、移位器的基础上，用基本运算指令编制乘除法子程序，供用户编程使用。该方法的优点是硬件简单，缺点是运算速度慢。在低档微控制器中经常采用这种方案。

(2) 在加法器、移位器的基础上适当增添一些硬件，使机器的指令系统中包含乘除法指令。这是折衷的办法。

(3) 设置专用乘除法部件，这样虽然会使集成电路结构复杂，硬件规模增大，成本增加，然而大大地提高了运算速度。高性能微处理器和数字信号处理器通常采用这种方案。

下面是原码移位乘法器的 VHDL 代码。它采用上述第二种方案，其中 A_PORT 和 B_PORT 是输入的两个 4 位操作数，PRODUCT 是 8 位乘积项，DONE 是工作信号。当 DONE 为'1'时表示一次乘法运算结束。进程 Shift_And_Add 中用 COUNT 变量表示迭代次数，经过 4 次移位相加得到乘法结果。移位相加过程中，部分积之和保存在 M 和 A 连成的移位链中。

```
entity Mult is
    generic(MULT_DEL, DONE_DEL: TIME);
    port (A_PORT, B_PORT: in Std_logic_vector(3 downto 0);
        PRODUCT: out Std_logic_vector(7 downto 0);
        DONE: out Std_logic);
end Mult;
architecture Shift_Mult of MULT is
begin
    Shift_And_Add: process(A_PORT, B_PORT)
        variable A,B,M: Std_logic_vector;
        variable: COUNT: Integer;
    begin
        A := A_PORT; B := B_PORT;
        COUNT := 0;
        M := "0000";
        DONE <= '0';
        while COUNT < 4 loop
            if A(0) = '1' then
                M := M+B;
            end if;
            A := M(0) & A(3 downto 1);
            M := '0' & M(3 downt 1);
            COUNT := COUNT + 1;
```

```
        end Loop;
        PRODUCT <= M & A after MULT_DEL;
        DONE <= '1' after DONE_DEL;
    end process Shift_And_Add;
end Shift_Mult;
```

3.1.10 算术逻辑单元(ALU)

为了实现 ALU,PLA(可编程逻辑阵列)的几个基本组合逻辑单元需要定义几个常用的函数及进程。下面的 VHDL 程序包中给出了这些函数及进程的定义。这几个函数及进程全是对不定长度的数据字进行运算,第 0 位是最低位。在函数及进程内部,定义了局部变量,该局部变量的参数域按降序排列。调用函数进行运算时把原变量按一定的规则赋给该局部变量,无论原变量的各位是升序还是降序,该局部变量都是按降序排列,即局部变量的最低位是数据字最右边的一位。关于这几个函数和进程的进一步说明如下:

(1) 加法运算 ADD:加法运算用 for 循环语句实现。循环的迭代次数与输入数据的字长相同,并假定两个输入变量的字长相同。

(2) 加 1 函数 INC:函数 INC 也使用了 for 循环语句。从输入变量的最低位开始检查,如果该位为'1',则将它变为'0',直到遇到第一个'0'时将其变为'1'并终止循环。

(3) 减 1 函数 DEC:函数 DEC 同样使用了 for 循环语句。从输入变量的最低位开始检查,如果该位为'0',则将它变为'1',直到遇到第一个'1'时将其变为'0'并终止循环。

(4) 函数 INTVAL:该函数用 for 循环语句把数据字转化为十进制整数。

```
package PRIMS is
    procedure ADD(A, B: in Std_logic_vector; CIN: in Std_logic ;
                  SUM: out Std_logic_vector; COUT: out Std_logic);
    function INC(X : in Std_logic_vector ) return Std_logic_vector;
    function DEC(X : in Std_logic_vector ) return Std_logic_vector;
    function INTVAL(VAL : in Std_logic_vector ) return Integer;
end PRIMS;

package body PRIMS is
    procedure ADD(A, B: in Std_logic_vector; CIN: in Std_logic ;
        SUM: out Std_logic_vector; COUT: out Std_logic) is
        variable SUMV, AV, BV: Std_logic_vector(A'Length - 1 downto 0);
        variable CARRY: Std_logic;
    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 : in Std_logic_vector ) return Std_logic_vector is
    variable XV: Std_logic_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 : in Std_logic_vector ) return Std_logic_vector is
    variable XV: Std_logic_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 : in Std_logic_vector ) return Integer is
        variable VALV: Std_logic_vector(VAL'Length - 1 downto 0);
        variable SUM : INTEGER := 0;
    begin
        VALV := VAL;
        for I in VALV'Low to VALV'High loop
            if VALV(I) = '1' then
                SUM := SUM + ( 2 * * I);
            end if;
        end loop;
        return SUM;
    end INTVAL;
end PRIMS;
```

ALU 是计算机系统中实现基本算术及逻辑运算的组合逻辑电路，下面的 VHDL 源代码是一个简单 ALU 的模型，它有三个数据输入端，其中 A 和 B 是数据字，CI 是进位输入；有两个数据输出端，其中 F 是数据字，COUT 是进位输出；还有一个控制输入端 FSEL，用来控制 ALU 要完成的功能。对于这个简单的 ALU，FSEL 是 2 位数据的数据字。该 ALU 可以完成如下四种运算：

(1) F = A；

(2) F = not (A)；

(3) F = A + B；

(4) F = A and B。

这里的符号＋表示二进制加法，可以接受进位输入并产生进位输出。

```
use work. PRIMS. all
entity ALU is
    generic(DEL : TIME);
    port( A, B: in Std_logic_vector (3 downto 0); CI: in Std_logic ;
            F: out Std_logic_vector(3 downto 0); COUT: out Std_logic
            FSEL in Std_logic_vector(1 downto 0) );
end ALU;

architecture ALG of ALU is
begin
    process(A, B, CI, FSEL)
        variable FV: Std_logic_vector(3 downto 0);
        variable COUTV : Std_logic;
```

```
    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;
```

这一简单 ALU 提供了基本算术及逻辑运算功能，可以将其改进，得到功能更全的 ALU。实际上，如果将 ALU 运算的结果存储起来，按适当的要求反馈回 ALU 的输入端，就可以实现其他的算术及逻辑运算功能。实际设计 ALU 时，要考虑如下两个问题：

(1) ALU 的运算集是否是一个完备集？是否可以通过对这个 ALU 的不同反馈调用实现所有可能的算术及逻辑运算功能？

(2) 如何在 ALU 的运算功能、运算速度以及所需要的其他基本单元(寄存器、多路选择器等)的数目之间进行综合平衡。

在实际应用中，通常将简单 ALU 级连起来处理字长较长的数据。图 3.2 是由 4 个 4 位 ALU 级连构成 16 位 ALU。在这个电路中，同样的控制信号 C0 和 C1 连接在所有 4 个 ALU 上。

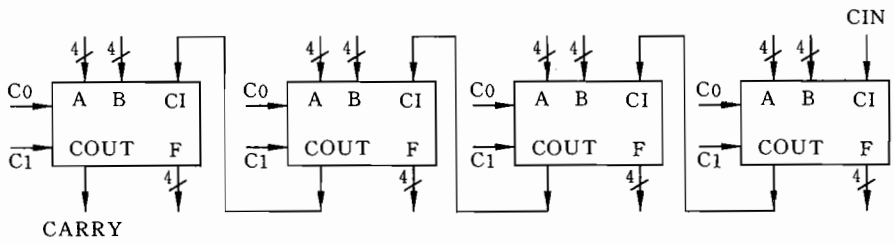


图 3.2　4 个 4 位 ALU 级连成为 16 位 ALU

以上的 ALU 模型是单纯行为模型，下面按照由顶向下的方法，以结构化的方式构造一个能够对 8 位数据进行加、减、与、或、异或运算的 8 位 ALU。

8 位 ALU 由 8 个结构完全相同的位单元和控制电路构成，可以对两个 8 位操作数进行加、减、与、或、异或等运算，此外还可以把其中一个操作数取反或清零。ALU 由三个层次的模块组成，下面给出的是各个层次的 VHDL 描述。ALU 的两个操作数存储在 TMP1 和 TMP2 中。L_XAOMD 为'1'表示进行逻辑运算，此时要关闭进位链。CIN 是进位输入，COUT 是进位输出。运算结果存储在 ALU_Q 中，这是一个 buffer 类型的端口，即实体内可读的输出端口。在位单元中，用 4 选 1 多路选择器作为布尔函数发生器，用曼彻斯特进

位链传递进位。

```
entity ALU is
    port(TMP1,TMP2: in Std_logic_vector(7 downto 0);
         Y    : in Std_logic_vector(3 downto 0);
         L_XAOMD : in Std_logic;
         CIN : in Std_logic;
         ALU_Q   : buffer Std_logic_vector(7 downto 0);
         COUT : out Std_logic);
end ALU;

architecture Netlist of ALU is
    signal ALU_CARRY : Std_logic_vector(7 downto 0);
    component ALU_BIT
       port(OP1,OP2 : in Std_logic;
            Y    : in Std_logic_vector(3 downto 0);
            CIN : in Std_logic;
            L_XAOMD : in Std_logic;
            SUM   : out Std_logic;
            COUT : buffer Std_logic);
    end component;
begin
    ALU_GEN : for I in 0 to 7 generate
        I0 : if I=0 generate
            ALU0 : ALU_Bit port map(TMP1(0),TMP2(0),Y,CIN,
            L_XAOMD,ALU_Q(0),ALU_CARRY(0));
        end generate I0;
        I1_7 : if I>0 generate
            ALU1_7 : ALU_Bit port map(TMP1(I),TMP2(I),Y,ALU_
            CARRY(I-1),
            L_XAOMD,ALU_Q(I),ALU_CARRY(I));
        end generate I1_7;
    end generate ALU_GEN;
end Netlist;
```

这段代码中使用生成语句(generate)对元件进行例化，采用这种办法可以比较灵活地控制各个子单元之间的连接关系。综合工具在处理生成语句时，通常就是简单地在网单中加入一系列相应的子模块。有些综合工具如 Synopsys 可以把子模块展平，即把全部电路打散后进行优化。但是对于 ALU 这类结构规则的电路，展平的优化并不一定有效，有

时甚至反而增大芯片面积或在综合的迭代过程中出现死循环。使用生成语句，在综合上的效果相当于强制综合工具重复使用相应的子模块。在这一点上，生成语句不如 for 循环语句灵活。对于 for 循环语句，综合结果可以是多次操作共享一个运算单元，也可以是并行使用多个单元，这视时钟周期和设计要求而定。

ALU 位单元的 VHDL 表述由实体和结构体 ALU_BIT 给出。电路采用比较经典的结构，其中，各种函数的运算用 4 选 1 多路选择器实现，进位链采用曼彻斯特结构。后面的 MUX4_1 和 Manchest_Chain 分别是这两个电路的描述。

```
entity ALU_BIT is
   port(OP1,OP2 : in Std_logic;
        Y   : in Std_logic_vector(3 downto 0);
        CIN : in Std_logic;
        L_XAOMD : in Std_logic;
        SUM : out Std_logic;
        COUT: buffer Std_logic);
end ALU_BIT;

architecture Dataflow of ALU_BIT is
    component MUX4_1
        port(A, B : in  Std_logic;
             Y    : in  Std_logic_vector(3 downto 0);
             F    : out Std_logic);
    end component;
    component Manchester_Chain
        port(CIN : in Std_logic;
             KILL,PROPAGATE : in Std_logic;
             L_XAOMD : in Std_logic;
             COUT : buffer Std_logic);
    end component;

    signal HALF,Q : Std_logic;
    signal KILL   : Std_logic;
begin
    Bool_Func   : MUX4_1 port map(OP1,OP2,Y,Q);
    Carry_ Chain   : Manchester_ Chain port map (CIN, KILL, HALF, L_
    XAOMD,COUT);
        HALF <= not Q;
        KILL <= (not OP2) and (not HALF);
```

```
        SUM <= HALF xor CIN;
end Dataflow;
```

一个二输入、一输出的组合逻辑电路可以看作一个布尔函数：

$$Y = F(X0, X1)$$

其中输出 Y 和输入 X0,X1 为二进制位。这个函数的输入、输出对应关系可以由表 3.2 的真值表所示。两输入变量的布尔函数最多可能有 16 种。若一个布尔函数的 4 个 Y 值(对应表 3.2 中的一列 F*n*)传送到 4 选 1 电路 MUX4 的 4 个数据输入端，将 X0,X1 送到 4 选 1 电路的选择控制端，则 MUX4 就实现了两个数据选择控制端的布尔函数。例如，将(0，1，1，0)送入 4 选 1 电路的 D0～D3，则 MUX4 就成了一个以 X0 和 X1 为输入的异或门。因此，将 MUX4 的选择控制信号看作数据输入，就实现了可变功能的逻辑门。根据指令的需要，将适当数值送到 4 选 1 的数据输入，就可以随时改变其逻辑功能。

表 3.2　两个变量的布尔函数

X1	X0	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
函数		0	与		X1		X0	异或	或	或非	异或非	X0 非		X1 非		与非	1

```
--4 选 1 多路选择器
entity MUX4_1 is
    port(A,B : in  Std_logic;
         Y   : in  Std_logic_vector(3 downto 0);
         F   : out Std_logic);
end MUX4_1;

architecture Behavior of MUX4_1 is
begin
    process(A,B,Y)
        variable SEL : Std_logic_vector(1 downto 0);
    begin
        SEL := Std_logic_vector(1 downto 0) (B & A);
        case SEL is
            when "00" =>  F <= Y(0);
            when "01" =>  F <= Y(1);
            when "10" =>  F <= Y(2);
```

```
            when "11" =>  F <= Y(3);
            when others =>  Null;
        end case;
    end process;
end Behavior;
```

以上的布尔函数可以完成各种逻辑运算，但是对于算术运算还是不够的。最基本的算术运算是加法，而减法可以看作是加法的特例，乘除法则由多次的加法和移位来实现。所以，ALU 只要实现加法功能即可。加法的算法采用了传统的异或生成和的算法，令 OP1 和 OP2 为两个操作数的对应位，CARRYIN 为进位输入，该算法如下：

HALF = OP1 xor OP2 ——(1)

SUM = HALF xor CARRYIN ——(2)

CARRYOUT = (CARRYIN and HALF) or (OP1 and OP2) ——(3)

其中，HALF 是半加和，SUM 是全加和，CARRYOUT 是进位输出。从进位的角度重新排列语句(1)至(3)，则有如下语句：

KILL = (not OP1) and (not OP2) ——(4)

PROPAGATE = HALF ——(5)

PRODUCE = OP1 and OP2 ——(6)

KILL 为'1'时，表示不会向高位产生进位；PROPAGATE 为'1'时，表示向高位传递本位的进位输入；PRODUCE 为'1'，则表示无论进位输入为何值，本位都会产生进位。根据语句(1)～(3)，可以得到异或生成和电路的算法；语句(4)～(6)就是所谓的曼彻斯特进位链结构。下面的 VHDL 源代码中的进位链采用了曼彻斯特结构。

```
--曼彻斯特进位链
entity Manchester_Chain is
    port(CIN : in Std_logic;
         KILL,PROPAGATE : in Std_logic;
         L_XAOMD : in Std_logic;
         COUT : buffer Std_logic);
end Chain;

architecture Dataflow of Chain is
begin
    Update : process(CIN,KILL,PROPAGATE,L_XAOMD)
    begin
        if L_XAOMD='1' then
            COUT <= '1';
        else
```

```
            if KILL='1' then
                COUT <= '0';
            elsif PROPAGATE='1' then
                COUT <= CIN;
            else
                COUT <= '1';
            end if;
        end if;
    end process;
end Dataflow;
```

3.1.11 可编程逻辑阵列(PLA)

在集成电路设计时,组合逻辑电路经常由可编程逻辑阵列实现。图 3.3 是一个三变量(X1,X2,X3)、四输出(Z1,Z2,Z3,Z4)的 PLA 阵列。虽然这个阵列可以实现输入为三个变量的多种逻辑函数,但函数中"乘积项"的个数不能超过 4。

图 3.3 所示的 PLA 由与(AND)阵列和或(OR)阵列构成。与阵列用来产生乘积项,即产生输入变量或输入变量的反变量的与,或阵列用来求各乘积项的或以产生输出函数。实际上,图 3.3 中的与阵列和或阵列全由或非(NOR)逻辑实现,但习惯上还是将它们称为与阵列和或阵列。

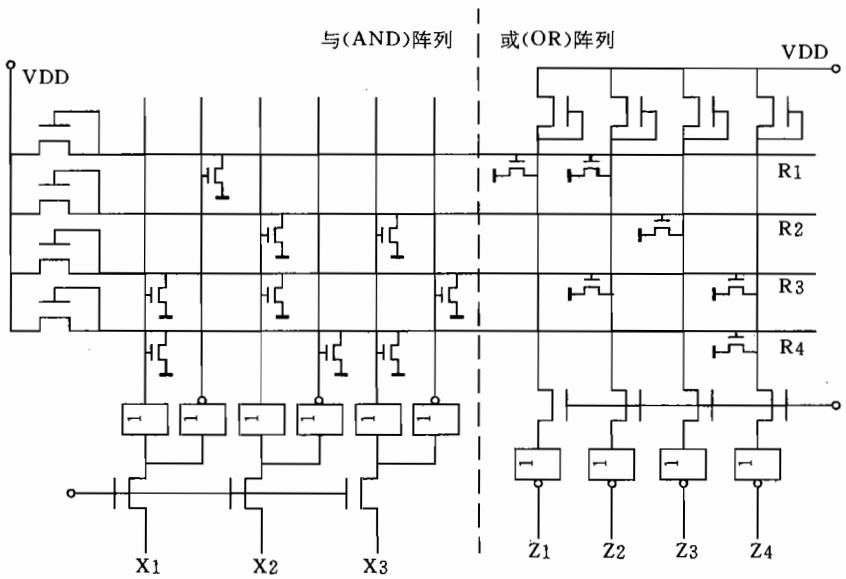


图 3.3 三输入变量、四输出函数的 PLA

如果逻辑函数中包含乘积项$\overline{A}B\overline{C}$,根据 DeMorgan 定理,$\overline{A}B\overline{C}=\overline{A+\overline{B}+C}$,即与门可以转换为或非门,也就是说应该将 A,$\overline{B}$ 和 C 连接到与阵列的或非门输入端。在图 3.3

中，每一条水平线(R1～R4)相应于一个乘积项，乘积项直接连接到或阵列中或非门的输入端以计算乘积项的或，构成所需的逻辑函数。图中的连接靠晶体管实现，完整的原理图实现了下述逻辑关系：

$$R1=X1$$
$$R2=\overline{X2}\cdot\overline{X3}$$
$$R3=\overline{X1}\cdot\overline{X2}\cdot X3$$
$$R4=\overline{X1}\cdot X2\cdot\overline{X3}$$

与

$$Z1=R1=X1$$
$$Z2=R1+R3=X1+\overline{X1}\cdot\overline{X2}\cdot X3$$
$$Z3=R2=\overline{X2}\cdot\overline{X3}$$
$$Z4=R3+R4=\overline{X1}\cdot\overline{X2}\cdot X3+\overline{X1}\cdot X2\cdot\overline{X3}$$

X2 和 X3 连接到第一个或非逻辑的输入端，所以 R2 相应于乘积项$\overline{X2}\cdot\overline{X3}$。简单地讲，如果在乘积项中需要输入变量 X，就将变量 X 的反变量 $\overline{X}$ 连接到与阵列中的或非门。读者可以用这种规则检查其他几个乘积项是否与电路相符。在图 3.3 所示的 PLA 中，与阵列最多可以有 24 个连接点，或阵列最多可以有 16 个连接点。

PLA 的模型可以在开关级描述，但需要多值逻辑。对于大多数应用，并不需要精确到开关级的 PLA 模型，可用下面的 VHDL 源代码描述该 PLA 的行为。这个模型通过扫描两个连接矩阵，计算 PLA 的输出函数值。模型分为两个进程：AND_PLANE 和 OR_PLANE，通过信号 R 把与阵列的输出和或阵列的输入连接起来。

在进程 AND_PLANE 中，连接矩阵存储在一个三维数组中，第一维选择与阵列的输出函数(R^I)，第二维选择输入变量(X^J)，第三维选择用 X 或是 $\overline{X}$ 作为与阵列中或非门的输入。连接矩阵用两个嵌套的循环扫描，内循环的循环变量是 I，外循环的循环变量是 J，如果矩阵元素 AND_PLANE(I, J, 1) = '1'，输入变量 X^J 连接到与阵列中的或非门，即变量$\overline{X^J}$包含在第 I 个乘积项中。如果 AND_PLANE(I, J, 2) = '1'，变量$\overline{X^J}$连接到与阵列中的或非门，即输入变量$\overline{X^J}$包含在第 I 个乘积项中。应该注意，输入变量 X^J 和变量$\overline{X^J}$不应该同时出现在同一乘积项中，源代码中的 assertion 语句用来检查这一条件。一旦循环过程结束，运算结果就被转换为所期望的或非形式，在循环体内部并没有直接使用或非运算。

在进程 OR_PLANE 中，连接矩阵是一个二维矩阵。当且仅当乘积项包含在输出逻辑函数中时，OR_PLANE(I, J) = '1'。或阵列同样也由两个嵌套的循环实现。与阵列及或阵列的延时由延时参数 AND_DEL 和 OR_DEL 表述。

```
entity PLA is
    generic(AND_DEL, OR_DEL: TIME);
    port(X: in Std_logic_vector (1 to 3); Z: out Std_logic_vector(1 to 4));
end PLA;

architecture CONNECRION_MATRIX of PLA is
```

```
begin
    AND_PLANE: Prcocess(X)
        variable RV: Std_logic_vector(1 to 4);
        type AND_ARRAY is ARRAY(1 to 4, 1 to 3, 1 to 2) of Std_logic;
        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 writing";
                if AND_PL(I, J, 1) = '1' then
                    RV(I) := RV(I) or (not X(J));
                endif;
                if AND_PL(I, J, 2) = '1' then
                    RV(I) := RV(I) or X(J);
                endif;
            end loop;
            R(I) <= not RV(I) after AND_DEL;
        end loop;
    end process AND_PLANE;

    OR_PLANE: Prcocess(R)
        variable ZV: Std_logic_vector(1 to 4);
        type OR_ARRAY is array(1 to 4, 1 to 4) of Std_logic;
        variable OR_PLAN : 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_PLAN(I, J) = '1' then
```

```
                    ZV(I) := ZV(I) or R(J);
                endif;
            end loop;
            Z(I) <= ZV(I) after OR_DEL;
        end loop;
    end process OR_PLANE;
end CONNECRION_MATRIX;
```

3.2 时序逻辑电路的造型

时序逻辑电路一般用进程或保护赋值语句描述。本节从单状态的时序电路开始，依次介绍主要的时序逻辑电路单元。

3.2.1 触发器

触发器是最基本的时序逻辑电路单元。下面是最常用的D触发器的VHDL模型。触发器在时钟CLK上升沿将D端的数值载入Q端。值得注意的是，在结构体ALG_if中，由于if语句只说明了上升沿时的电路行为，隐含表示其他情况下电路保持原有状态，因此可以引导综合工具生成D触发器。结构体ALG_wait是ALG_if的等价形式，也就是说，仿真和综合结果都完全相同。但是因为使用wait语句表示时钟沿，所以具有较高的仿真效率。仿真这种风格的代码时，只有在CLK上升沿进程才会被激活；而ALG_if中的进程，则只要CLK和D发生变化都要被激活。仿真时，活动进程的个数直接决定了CPU的负载，当前活动进程越多，则CPU负载越重。在集成电路设计过程中，特别是电路规模很大时，仿真效率是必须注意的问题，所以建议使用ALG_wait的建模风格。这段代码之所以会被综合为D触发器，是由于wait语句实际上隐含表示电路在状态更新之间要保持原来的数值。

```
entity D_FF is
    generic(DEL: TIME);
    port(CLK: in Std_logic;
         D: in Std_logic;
         Q: out Std_logic);
end D_FF;

architecture ALG_if of D_FF is
begin
    D_Flip-flop: process(CLK, D)
    begin
       if CLK='1' and CLK'Event then
```

```
            Q <= D after DEL;
        end if;
    end process;
end ALG;

architecture ALG_wait of D_FF is
begin
    D_Flip-flop: process
    begin
        wait until CLK='1' and CLK'Event;
            Q <= D after DEL;
    end process;
end ALG;
```

触发器需具备复位功能。复位有两种方式:同步复位和异步复位。下面的 VHDL 源代码是具有复位功能的 D 触发器的 VHDL 模型。其中,结构体 ALG_wait 是同步复位 D 触发器,结构体 ALG_if 是异步复位 D 触发器。

```
entity D_FF_with_Reset is
    generic(RESET_DEL, UPDATE_DEL: TIME);
    port(CLK, RST: in Std_logic;
         D: in Std_logic;
         Q: out Std_logic);
end D_FF_with_Reset;

architecture ALG_wait of D_FF_with_Reset is
begin
    D_Flip-flop: process
    begin
        wait until CLK= '1' and CLK'Event;
            if RST = '1' then
                Q <= '0'  after RESET_DEL;
            else
                Q <= D after UPDATE_DEL;
            end if;
    end process;
end ALG_wait;

architecture ALG_if of D_FF_with_Reset is
```

```
begin
    D_Flip-flop: process(RST, CLK, D)
    begin
        if RST= '1' then
            Q <= '0' after RESET_DEL;
        elsif CLK= '1' and CLK'Event then
            Q <= D after UPDATE_DEL;
        end if;
    end process;
end ALG_if;
```

下面的 VHDL 源代码是 J-K 触发器的行为模型。在这个模型中，置位端 S 和复位端 R 的作用比时钟端 CLK 的优先级高。如果 R=S='1'，则触发器不进行任何操作。由于 R=S='1'是非法操作，一般地讲触发器的输出应是不定值，所以该行为模型在 R=S='1'时不进行操作，与实际触发器的性能规定略有差别。读者可以参照 D 触发器和 J-K 触发器，写出其他触发器的模型。

```
entity JK_FF is
    generic(SR_DEL, CLK_DEL: TIME);
    port(S, R, J, K, CLK : in Std_logic;Q, QN: out Std_logic);
end JK_FF;

architecture ALG of JK_FF is
begin
    JK_Flip-flop: Prcocess(CLK, S, R)
    begin
        if S = '1' and R= '0' then
            Q <= '1' after SR_DEL;
            QN <= '0' after SR_DEL;
        elsif S = '0' and R = '1' then
            Q <= '0' after SR_DEL;
            QN <= '1' after SR_DEL;
        elsif CLK'Event and CLK= '1' and R= '0' and S = '0' then
            if J = '1' and K = '0 then
                Q <= '1' after CLK_DEL;
                QN <= '0' after CLK_DEL;
            elsif J = '0' and K = '1' then
                Q <= '0' after CLK_DEL;
                QN <= '1' after CLK_DEL;
```

```
            elsif J = '1' and K = '1'
                Q <= not Q after CLK_DEL;
                QN <= not QN after CLK_DEL;
            end if;
        end if;
    end process ;
end ALG;
```

3.2.2 锁存器

锁存器用来存储数据。锁存器一般可分成两种类型:电平锁存器和同步锁存器。如果使用多相时钟,比如在微处理器芯片中,常常使用电平锁存器。电平锁存器经常有多路数据输入,下面是二输入电平锁存器的 VHDL 模型。

```
entity Latch_2DR is
    generic(DEL: TIME);
port(D1,D2 : in Std_logic;
   C1,C2 : in Std_logic;
   R : in Std_logic;
   Q : out Std_logic);
end Latch_2DR;

architecture ALG of Latch_2DR is
begin
    Latch : process(D1,D2,C1,C2,R)
    begin
        if R='1' then
            Q <= '0' after DEL;
        elsif C1='0' then
            Q <= D1 after DEL;
        elsif C2='0' then
            Q <= D2 after 1ns;
        end if;
    end process Latch;
end ALG;
```

下面是一个完全同步的锁存器,就是说,复位和加载功能全与时钟同步,复位端的优先级较高。与此类似,还可以写出复位与时钟不同步的锁存器,在数字系统设计时,采用完全同步的锁存器,可以避免时序错误。这里给出的锁存器模型是一个可控的锁存器,在 LOAD='1'且时钟的上升沿才能被加载。控制信号 LOAD 使得锁存器可以由其他单元

控制。

```
entity REG is
    generic(DEL: TIME);
    port( RST, LOAD, CLK: in Std_logic;
          DATA_IN: in Std_logic_vector(3 downto 0);
          Q: buffer Std_logic_vector(3 downto 0) );
end REG;

architecture DF of REG is
begin
    REG: block(CLK'Event and CLK = '1')
    begin
        Q <= guarded "0000" after DEL when RST='1' else
                    DATA_IN after DEL when LOAD='1' else
                    Q;
    end block REG;
end DF;
```

3.2.3 计数器

计数是数字系统的基本操作。下面给出的是16位计数器的VHDL模型，它可以被复位和加载，并可以由信号UP控制进行加计数或减计数。各操作的优先级顺序(从高到低)为“复位”、“加载”、“计数”，所有操作均以同步方式工作。这段代码使用了程序包types_of_Counter，其中定义了子类型INT16，即0至15的正整数。计数器的数据输入和输出被定义为INT16类型。实际上，就仿真而言，把COUNTER_Q简单地定义为整数与INT16效果相同。但从综合的角度看，定义这个子类型可以引导综合器以4根信号线处理COUNTER_Q，并用4个触发器储存结果。否则，有些综合工具会以整数的默认宽度(该默认宽度随不同EDA平台而不同，一般是65536以上的2的整数幂)处理，这显然是不能接受的。加减计数就用+、-运算表示，现在比较先进的综合工具都可以利用16位加减计数的规律性来实现电路，而不是引入一个加法器。

```
package types_of_Counter is
    constant SIZE : Integer := 16;
    subtype INT16 is Integer range 0 to SIZE - 1;
end types_of_Counter;

library IEEE;
use IEEE. Std_logic_1164. all;
use WORK. types_of_Counter. all;
```

```
entity Counter16 is
   port(RST: in Std_logic;
        CLK: in Std_logic;
        COUNTIN: in INT16;
        UP: in Std_logic;
        LOAD: in Std_logic;
        COUNTER_Q:  buffer INT16);
end Counter16;

architecture ALG of Counter16 is
begin
    Update_Counter : process
    begin
        wait CLK='1' and CLK'Event;
            if RST ='1' then
                    COUNTER_Q <= 0;
            elsif LOAD='1' then
                    COUNTER_Q <= COUNTIN;
            else
                    if UP='1' then
                        COUNTER_Q <= (COUNTER_Q + 1) mod SIZE;
                    else
                        COUNTER_Q <= (COUNTER_Q - 1) mod SIZE;
                    end if;
               end if;
    end process Update_Counter;
end ALG;
```

3.2.4 有限状态机

有限状态机是时序电路的通用模型，任何时序电路都可以表示为有限状态机。在第2章中已经介绍过硬件系统可以分成相互作用的两部分，即控制器和数据路径，并且也给出了有限状态机的定义。控制器通常都是有限状态机。有限状态机分为两类：Moore 和 Mealy 型有限状态机。下一状态只由当前状态决定的有限状态机称为 Moore 型；下一状态不但与当前状态有关，而且与当前输入值有关的有限状态机称为 Mealy 型。实际上，Moore 型有限状态机是 Mealy 型的特例。

下面的 VHDL 源代码是一个 Mealy 型有限状态机的 VHDL 模型。该有限状态机共

有 4 个状态 S0～S3，信号 CURRENT_STATE 和 NEXT_STATE 分别表示当前状态和下一状态。复位或仿真初始化时，电路进入 S0 状态。进程 Compute_State 根据当前状态和输入信号计算下一状态，进程 Update_State 在时钟沿上更新电路状态。这种描述方式可以较好地引导综合工具生成 FSM 电路。

```
entity Mealy_FSM is
    port ( RST   : in Std_logic;
           CLK: in Std_logic;
           C1, C2, C3 : in Std_logic;          ——输入条件信号
           O1, O2, O3 : out Std_logic);
end Mealy_FSM;

architecture ALG of Mealy_FSM is
   type States is (S0, S1, S2, S3);
    signal CURRENT_STATE,NEXT_STATE : States ;
begin
    Compute_State: process(RST, CURRENT_STATE, C1, C2, C3)
    begin
        if RST='1' then
           NEXT_STATE <= S0;
        else
           case CURRENT_STATE is
               when S0 =>
                   O1 <= '0'; O2 <= '0'; O3 <= '0';
                   if C1='1' then
                       NEXT_STATE <= S1;
                   else
                       NEXT_STATE <= S0;
                   end if;
               when S1 =>
                   O1 <= '1';
                   NEXT_STATE <= S2;
               when S2 =>
                   O1 <= '0';
                   O2 <= '1';
                   if C2='1' and C3='1' then
                      NEXT_STATE <= S3;
                   elsif ( (C2 xor C3) ='1' ) then
```

```
                NEXT_STATE <= S1;
            else
                NEXT_STATE <= S2;
            end if;
        when S3 =>
            O2 <= '0';
            O3 <= '1';
            NEXT_STATE <= S0;
    end case;
  end if;
end process Compute_State;

Update_State: process
begin
  wait until CLK='1' and CLK'Event;
  CURRENT_STATE <= NEXT_STATE;
end process Update_State;

end ALG;
```

3.3 存 储 器

存储器单元实际上是时序逻辑电路的一种。用 VHDL 语言描述存储器时，通常使用数组。

3.3.1 只读存储器(ROM)

ROM 在数字电路中用于存放指令或常数。下面的 VHDL 代码描述 32 字节×8 位 ROM 的行为。在程序包 ROMS 中，定义了常量数组 ROM。结构体中，如果 RD 端为'1'，则进程 Output 在时钟上升沿对 ROM 进行读操作；反之，输出为高阻态。

```
package ROMS is
    -- declare a 5x8 ROM called ROM
    constant ROM_WIDTH: Integer := 8;
    subtype ROM_WORD is Std_logic_vector (Rom_Width-1 downto 0);
    subtype ROM_RANGE is Integer range 0 to 31;
    type ROM_TABLE is array (0 to 31) of ROM_WORD;
    constant ROM: ROM_TABLE := ROM_TABLE'(
        ("10101000"), ("11100000"), ("00000000"), ("01110101"),
```

```
        ("00000000"), ("01111111"), ("11110010"), ("00000000"),
        ("00000001"), ("00000000"), ("00000000"), ("00000011"),
        ("01110010"), ("00000111"), ("00000000"), ("11000010"),
        ("00000000"), ("00100000"), ("00000000"), ("10100010"),
        ("00000000"), ("10010010"), ("00000000"), ("00000001"),
        ("01110101"), ("00000111"), ("11110001"), ("11000000"),
        ("11100000"), ("11010000"), ("11100000"), ("00011111"));
end ROMS;

use work.PRIMS.all;
use work.ROMS.all;

entity ROM_32x8 is
    generic(R_DEL, DIS_DEL: TIME);
    port(ADDR: in Std_logic_vector(4 downto 0);
         CLK: in Std_logic;
         RD: in Std_logic;
         DATA : out Std_logic_vector(7 downto 0));
end ROM_32x8;

architecture BEHAVIOR of ROM_32x8 is
begin
    Output : process
    begin
        wait until CLK'Event and CLK='1';
        if RD='1' then
            DATA <= ROM(INTVAL(ADDR)) after R_DEL;
        else
            DATA <= (others=>'Z') after DIS_DEL;
        end if;
    end process Output;
end BEHAVIOR;
```

3.3.2 随机存取存储器(RAM)

RAM 是重要的时序逻辑单元,它是一个存储器阵列,根据地址信号,经由译码电路选择欲访问的存储单元。RAM 可以由前面讨论过的逻辑单元来组合出结构描述,但对于仿真来讲,行为模型的效率更高。

下面的 VHDL 源代码是随机存取存储器的行为模型。该模型由进程实现,进程的敏

感信号是低电平有效的信号选择线 NCS、读控制线 RD、写控制线 WR、数据输入线 DATA 和地址线 ADDRESS。在进程内部检查 NCS 是否为低电平。如果 NCS='0',再检查是否有 RD='1'或 WR='1',如果二者之一为'1',则执行所选择的数据操作;如果 NCS='1',将高阻态"ZZZZ"赋给数据总线 DATA。在存储器操作时,要用到数据类型的转换,就是将数据字类型的地址信号 ADDRESS 转换为整数,用来指明 RAM 中的数组元素。对大部分应用,行为级设计时都可以用这一模型进行仿真。

```
use work. PRIMS. all;
entity RAM is
    generic(R_DEL, DIS_DEL: TIME);
    port(RD, WR, NCS : in Std_logic;
         ADDRESS: in Std_logic_vector(4 downto 0);
         DATA: inout Std_logic_vector(3 downto 0) );
end RAM;

architecture Behavior of RAM is
    type MEMORY is array(0 to 31) of Std_logic_vector(3 downto 0);
    signal MEM: MEMORY;
begin
    Access_RAM: process(RD, WR, NCS, ADDRESS, DATA)
    begin
        if NCS='0' then
            if RD='1' then
                DATA <= MEM(INTVAL(ADDRESS)) after R_DEL;
            elseif WR='1' then
                MEM(INTVAL(ADDRESS)) <= DATA;
            endif;
        else
            DATA <= "ZZZZ" after DIS_DEL;
        endif;
    end process Access_RAM;
end Behavior ;
```

以上模型能够满足行为级仿真的需要,但是如果是在寄存器传输级进行设计,需要具体设计 RAM 的控制信号时,这个模型的精度不够。此时,设计师必须能够描述 RAM 的行选、列选、地址线复用等行为。下面是描述 4 兆位 DRAM 的 VHDL 模型。这个 DRAM 被组织为 512 千字节,地址用 19 位描述,其中行地址 10 位,列地址 9 位,复用 10 位的地址线 A,数据以字节为单位存取。无效信号组合时,DRAM 输出高阻态。

```
library IEEE;
```

```
use IEEE. Std_logic_1164. all;
use IEEE. Std_logic_arith. all;
entity DRAM8 is
    generic(R_DEL, W_DEL, RAS_DEL, CAS_DEL, DIS_DEL: TIME);
    port( A: in Std_logic_vector(9 downto 0);
        D: inout Std_logic_vector(7 downto 0) := (others => 'Z');
        RAS, CAS, WE: Std_logic);
end DRAM8;

architecture Behavior of DRAM8 is
    signal ADRESS: Std_logic_vector(9 downto 0);
begin
    RAS_entry: process(RAS)
    begin
        if RAS'Event and RAS ='0' then
            ADRESS <= A after RAS_DEL;
        end if;
    end process;

    Data_IO: process(CAS,WE)
        subtype MEMORY_DATA is Integer range 0 to 255;    --8bit data
        type MEMORY_ARRAY is array ( 0 to 524287) of MEMORY_DATA;
        variable INT_ADDR: integer;
        variable ADRESS_ALL: Std_logic_vector(18 downto 0);
        variable DRAM_ARRAY: MEMORY_ARRAY;
    begin
        if CAS'Event and CAS ='0' then
            ADRESS_ALL := ADRESS & A(8 downto 0) after CAS_DEL;
        end if;
        INT_ADDR := CONV_INTEGER(UNSIGNED(ADRESS_ALL));
        if WE='0' and CAS='0' and RAS='0' then
            DRAM_ARRAY(INT_ADDR) := CONV_INTEGER(UNSIGNED
            (D))after W_DEL;
        elsif WE='1' and CAS='0' and RAS='0' then
            D <= Std_logic_VECTOR(CONV_UNSIGNED(
            DRAM_ARRAY(INT_ADDR),8)) after R_DEL;
        elsif CAS'Event and CAS='1' then
```

```
            D <= "ZZZZZZZZ" after DIS_DEL;
        end if;
    end process Data_IO;
end Behavior;
```

3.3.3 堆栈

各种队列，比如先进先出队列、后进先出队列(即堆栈)等，是数字电路(特别是计算机电路)中经常使用的存储器单元。这一小节中，以堆栈为例说明了该类单元的 VHDL 模型构造方法。下面是堆栈的 VHDL 行为描述，所有操作均由时钟信号 CLK 同步。PUSH 和 POP 控制信号分别将数据压入和弹出堆栈，EMPTY 为'1'，表示堆栈中没有合法数据；FULL 为'1'，表示堆栈已满。信号 C 用来计算堆栈中合法数据的个数，函数 TO_BIT 把 Boolean 数据类型转换为 Std_logic。这个纯行为式的描述是不能综合的。

```
entity Stack is
    generic(N: Positive;          --堆栈元素的字长
            K: Positive);         --堆栈中元素的个数
    port(RST, CLK: in Std_logic;
         PUSH, POP: in Std_logic;
         EMPTY, FULL: out Std_logic;
         DIN: in Std_logic_vector(N-1 downto 0);
         DOUT: out Std_logic_vector(N-1 downto 0));
end Stack;

architecture ALG of Stack is
    signal NUM: Integer Range 0 to K-1;
    function TO_BIT (B: in Boolean) return Std_logic is
    begin
        case B is
            when true => return '1';
            when false => return '0';
        end case;
    end To_Bit;
begin
    EMPTY <= To_Bit(C=0);
    FULL <= To_Bit(C= K - 1);

    Access_Stack: process
        type TYPE_Stack is array (Natural range K-1 downto 0) of
```

```
                Std_logic_vector(N-1 downto 0);
        variable S: TYPE_Stack;
    begin
        wait until CLK'Event and CLK='1';
            if RST='1' then
                C <= 0;
            elsif PUSH='1' then
                S(K-1 downto 1) := S(K-2 downto 0);
                S(0) := DIN;
                C <= C+1;
            elsif POP='1' then
                DOUT <= S(0);
                S(K-2 downto 0) := S(K-1 downto 1);
                C <= C-1;
            end if;
    end Access_Stack;
end ALG;
```

3.4 逻辑门级精度的模型

本节中讨论什么样的模型具有逻辑门级精度，即讨论逻辑门级模型有哪些特征。首先通过2输入或门来研究逻辑门级模型的特征。下面的VHDL源代码是VHDL语言初学者写出的或门的三种模型。结构体DELTA_DEL实现了或门的功能，指定了Delta延时。在集成电路设计中，Delta延时通常就意味着“单位延时”。集成电路的这种模型只能用于验证电路的逻辑功能。结构体FIXED_DEL规定了电路模型的传输延时，但这仍不是通用的模型。因为如果要改变器件的延时参数，必须要改变结构体内的延时值。当逻辑门的个数较少时，可以逐个器件改变延时参数，但如果电路规模较大，逻辑门的个数较多，逐个器件改变延时值则很不现实。结构体GEN_DEL采用了类属(generic)参数指定器件延时，解决了这一问题。类属参数出现在器件接口描述中，在器件被使用时才给定其数值。

```
entity OR2 is
    port (IN1, IN2: in Std_logic; OUT: out Std_logic);
end OR2;

architecture DELTA_DEL of OR2 is
begin
    OUT <= IN1 or IN2;
end DELTA_DEL;
```

```
architecture FIXED_DEL of OR2 is
begin
    OUT <= IN1 or IN2 after 3ns;
end FIXED_DEL;

entity OR2G is
    generic(DEL: TIME);
    port (IN1, IN2: in Std_logic; OUT: out Std_logic);
end OR2G;

architecture GEN_DEL of OR2G is
begin
    OUT <= IN1 or IN2 after DEL;
end GEN_DEL;
```

如果研究逻辑器件的基本时序问题，类似于结构体 GEN_DEL 的器件模型就足够精确。但如果对一般的集成电路器件手册进行研究，就会发现在手册中大部分器件的延时参数并不是按这种方式给出。图 3.4 给出了一般器件的非对称延时模型。该模型的输入为 IN，输出为 OUT。延时参数 TPLH 定义为：输入信号 IN 变化时，输出信号 OUT 的响应从低电平'0'变为高电平'1'所需的时间。另一个延时参数 TPHL 定义为：输入信号 IN 变化时，输出信号 OUT 的响应从高电平'1'变为低电平'0'所需的时间。

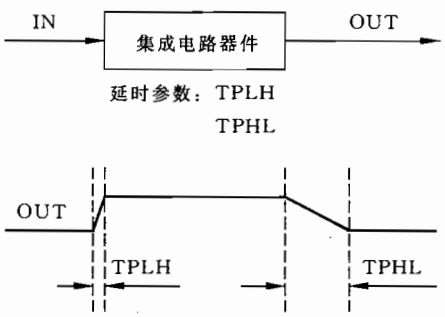


图 3.4　非对称的时序模型

根据定义，TPLH 和 TPHL 都是输出信号对输入信号的变化产生的响应，对于输入信号 IN 本身是从'0'变为'1'还是从'1'变为'0'并没有作出规定。TPLH 和 TPHL 数值不同的主要原因是，数字器件的输出端以高电平或以低电平驱动外部负载时内阻不同。

下面的 VHDL 模型考虑了或门的 TPLH 和 TPHL 数值不同的情况。模型中使用了两个内部变量 OR_NEW 和 OR_OLD，分别用来存储当前进程调用时和前一次进程调用时的运算结果。比较这两个变量，可以确定是否需要为进程的输出信号分配新值，如果对输出信号赋新值，也通过对比 OR_NEW 和 OR_OLD 的数值确定采用什么样的延时数值。

```
entity OR2GV is
    generic(TPLH, TPHL: TIME);
    port (IN1, IN2: in Std_logic; OUT: out Std_logic);
end OR2GV;
```

```
architecture VAL_DEL of OR2GV is
begin
    process(IN1, IN2)
        variable OR_NEW, OR_OLD : Std_logic;
    begin
        OR_NEW := IN1 or IN2;
        if OR_NEW = '1' and OR_OLD = '0' then
            OUT <= OR_NEW after TPLH;
        else if OR_NEW = '0' and OR_OLD = '1' then
            OUT <= OR_NEW after TPHL;
        endif;
        OR_OLD := OR_NEW;
    end process;
end VAR_DEL;
```

对于孤立的或门造型，上面的 VHDL 模型可以很好地描述器件行为。但是，这个模型没有考虑两个因素：① 没有考虑器件互连产生的延时；② 没有考虑器件驱动的负载对延时值的影响。下面通过一个实例讨论器件互连对延时数值的影响。在图 3.5(a)中，或门由另一个与门驱动，与门的输出到或门的输入之间可能是金属连线或多晶硅连线。这一段连线对电路延时值的影响有两个。第一，从与门的输出到输出的扇出点的延时 DEL1；第二，从与门输出的扇出点到或门的输入端 IN2 的延时 DEL2。处理这种情况的最简单的方法是把 DEL1 和 DEL2 之和看作从或门的输入端 IN2 到其输出端 OUT 的延时。这时必须注意到，若一个电路中所有的逻辑门都采用这种方法处理延时，则互连对延时作用不能正确反映在系统模型之中。图 3.5(b)示意给出了或门的一种较好的延时模型，在这个模型中，每个输入端都有一个延时，或运算本身被认为是无延时运算，传输延时出现在输出端。

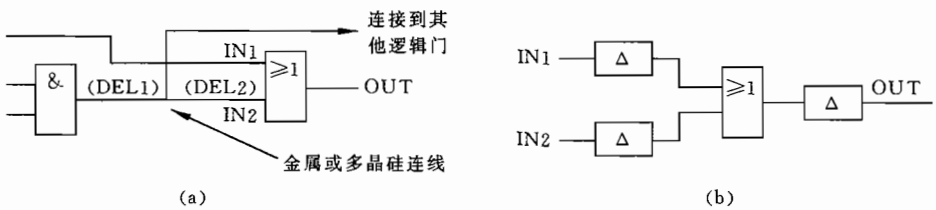


图 3.5　输入延时的模型

下面的 VHDL 源代码考虑了输入延时，对于每个输入端 IN，给定了两个类属参数 TPLHi(i=1,2)和 TPHLi(i=1,2)，在实体说明时用这两个类属参数指定输入延时。在集成电路版图设计完成以后，这两个类属参数的数值通过反向标注过程确定。首先，测量指定的逻辑门输入管栅极连线长度，将这个长度乘以单位长度金属(或多晶硅)的电容 (pF/μm)，再把得到的结果乘以单位电容对应的延时因子(ns/pF)，就得到了该栅极的输入延时。对于每个逻辑门的每个输入栅极依次进行上述计算，就可以得到逻辑门各个输入端的延时参数。

```
entity OR2GI is
    generic(TPLH1, TPHL1, TPLH2, TPHL2, TPLHO, TPHLO: TIME);
    port (IN1, IN2: in Std_logic; OUT: out Std_logic);
end OR2GI;

architecture VAR_DEL of OR2GI is
    signal IN1_D, IN2_D: Std_logic;
begin
    process(IN1, IN2)
    begin
        if IN1'Event then
            if IN1 = '0' then
                IN1_D <= IN1 after TPHL1;
            else
                IN1_D <= IN1 after TPLH1;
            end if;
        end if;
        if IN2'Event then
            if IN2 = '0' then
                IN2_D <= IN2 after TPHL2;
            else
                IN2_D <= IN2 after TPLH2;
            end if;
        end if;
    end process;

    process (IN1_D, IN2_D)
        variable OR_NEW, OR_OLD : Std_logic;
    begin
        OR_NEW := IN1_D or IN2_D;
        if OR_NEW = '1' and OR_OLD = '0' then
            OUT <= OR_NEW after TPLHO;
        else if OR_NEW = '0' and OR_OLD = '1' then
            OUT <= OR_NEW after TPHLO;
        endif;
        OR_OLD := OR_NEW;
    end process;
end VAR_DEL;
```

根据金属或多晶硅连线计算出的延时数值并没有将器件扇出对延时的影响考虑在模型中。对于 MOS 电路，由于栅极的输入阻抗很高，可以采用上述输入延时模型，而省略输出延时。输出延时模型通常利用输出负载电容的数值计算延时值。其计算公式为

$$T'_{pd} = (T_{pd} + l_{delay}(C_{load})) \cdot K_v \cdot K_t$$

其中，T'_{pd}为考虑输出延时模型的传输延时值；T_{pd}为制造厂提供的传输延时常数；C_{load}是反向标注得到的电容负载值(计算得到的负载电容)；l_{delay}是负载电容引起的延时值，它是负载电容的函数；K_v是延时值的电压因子；K_t 是延时值的温度因子。

采用 l_{delay}的线性模型，其计算公式为

$$l_{delay}(C_{load}) = (C_{load} - C_{ref}) \cdot K_C$$

其中，C_{load}是该器件所驱动的总负载电容；C_{ref}是数据手册上给出的电容值；K_C 是线性电容负载因子，其量纲为“时间/电容”。

在集成电路版图设计完成后的反向标注过程中，可以计算出负载电容的总数值，通常为器件输出端所驱动的各条连线的电容以及所驱动负载的输入总电容之和。

除了在输入端增加延时和输出端增加延时之外，采用延时函数也可以实现不同延时。对于实体 OR2GV，下面的结构体 FUNC_DEL 是采用函数实现延时的一种方法。这个结构体通过程序包 DELAY 中的函数 VAR_DEL，确定应该采用延时值 TPHL 还是延时值 TPLH。可以看出，这个结构体是最简洁的或门延时模型。但是，当或门的两个输入信号发生变化时，无论其输出值是否发生变化，总是要对这个模型的输出信号进行操作。与此不同，结构体 VAR_DEL 只在或门的输出新值与旧值不同时，才对其输出信号进行操作，所以与结构体 VAR_DEL 相比，FUNC_DEL 仿真效率较低。

```
package DELAY is
     function VAR_DEL (TPLH, TPHL: TIME; FNEW: Std_logic) return
     TIME;
end DELAY;

package body DELAY is
    function VAR_DEL(TPLH,TPHL: TIME; FNEW: Std_logic) return TIME
      is
    begin
        if FNEW = '0' then
            return TPHL;
        else
            return TPLH;
        end if;
    end VAR_DEL;
end DELAY;
```

```
use work. DELAY. all
architecture FUNC_DEL of OR2GV is
begin
    process(IN1, IN2)
        variable OR_NEW: Std_logic;
    begin
        OR_NEW := IN1 or IN2;
        OUT <= OR_NEW after VAR_DEL(TPLH, TPHL, OR_NEW);
    end process;
end FUNC_DEL;
```

大部分情况下，对于每一个器件，需要三套延时数据，即延时的最小值、最大值和典型值。对于纯功能验证，设计者又期望能用单位延时，即 Delta 延时进行仿真验证。这就是说硬件模型中的延时函数至少应该能处理四种延时值。除了延时参数的选取之外，有时需要按全局方式选择延时数据，比如规定所有器件都采用典型延时值或最大延时值。下面的 VHDL 源代码说明了如何写出满足这些要求的器件模型。在程序包 TIMING_CONTROL 中，定义了数据类型 TIMING，用来描述四值实时模型。常数 TIMING_SEL 用来说明所选择的延时条件，其缺省值为 TYP，即规定在未加说明的情况下，延时值应该取典型值。函数 T_CHOICE 用来根据 TIMING_SEL 的值选择适当的时序模型。在程序包 TIMING_CONTROL 之后，给出了或门的一种模型。语句 use work. TIMING_CONTROL. all 用来使程序包 TIMING_CONTROL 中的内容在所有逻辑门中可见。即每当常数 TIMING_SEL 发生改变时，所有器件模型都会使用改变后的新延时值。严格地讲，改变程序包中的参数后需要重新编译程序包，还需要重新编译与程序包有关的所有器件模型。但是，不少 VHDL 工具都允许在仿真过程中改变程序包中的参数值，而不需要对程序包以及器件模型重新编译。

```
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(IN1, IN2: in Std_logic; OUT: out Std_logic);
end OR2_TV;

architecture VAR_T of OR2_TV is
begin
    OUT <= IN1 or IN2 after T_CHOICE(TIMING_SEL, TMIN, TMAX,
        TTYP);
end VAR_T;
```

3.5 VITAL 规 范

长期以来，由于缺乏高效可靠的用 VHDL 描述的 ASIC 库，一定程度上影响了 VHDL 的广泛应用。而建立 ASIC 库的最大困难在于 VHDL 中没有统一、有效的方法处理时域参数。为此，工业界和 IEEE 开展了一系列研究工作，尝试解决这一问题，其中 VITAL(VHDL initiative towards ASIC library)是最重要的结果。现在工业界有专门的委员会负责 VITAL 的修订工作，该委员会由来自 Synopsys，Cadence，HP，TI 及 VHDL 技术组的成员组成。一些 EDA 工具如 Synopsys 支持 VITAL 风格的 VHDL 模型，Exemplar Logic 公司的 Galileo 支持用 VITAL 进行 Xilinx FPGA 设计。

为了能够对 ASIC 精确建模，VITAL 必须要解决如下关键问题：

(1) 精确描述时序关系。包括延时模型(可能是引脚到引脚的延时模型或分布式延时模型)、时序检查(如建立、保持时间检查)、尖峰脉冲处理和宏单元间的互连延时等。

(2) 高仿真效率。VITAL 主要处理门级逻辑单元，因此必须具备高的仿真效率，否则，常因 ASIC 的规模很大，导致仿真时间长得难以令人接受。

(3) 具有反向标注能力。VLSI 的设计是层次式的迭代与提炼的过程，层次越低，就可以得到更精确的延时信息，因此，需要一种机制把低层次的延时信息反向标注到较高层次。此外，现在已经有描述延时信息的工业标准延时格式 SDF(standard delay format)，所以 VITAL 应具有按 SDF 进行反向标注的能力；

(4) 通用性，即 VITAL 模型应该适用于各种 VHDL 仿真器。

为了同时解决以上这些问题，VITAL 没有对 VHDL 语言本身进行扩展，而是提供了一定的编程指导方针和预定义的包，提供了对功能和延时造型所需的子程序库，从而实现了对 ASIC 宏单元的准确建模。使用 VITAL 时，功能在模型内部描述，而所有延时的计算则在模型外部完成。VITAL 版本 3.0 规范主要包括：预定义库、建模指导方针、从 SDF 获得回注的语法。

3.5.1 预定义的 VITAL 库

VITAL 提供了两个程序包，VITAL_Timing 和 VITAL_Primitive。VITAL_Timing 包含与 ASIC 单元中时间行为有关的各种类型定义、常数、属性和过程，其中包括计算模型输入、输出端口连线延时的过程 VitalWireDelay，计算引脚到引脚延时的过程 VitalPathDelay，检查建立、保持时间的过程 VitalSetupHoldCheck 等。VITAL_Primitive 提供一系列子程序，描述数字电路建模中常见的各种电路模块的行为。表 3.3 列出 VITAL_Primitive 中的所有模块。每种模块的代码均有函数和过程两种形式；还提供了子程序 VitalTruthTable，使设计者能以真值表的方式描述电路的行为。VITAL_Primitive 库的做法类似于在 VHDL 引入了 Verilog 的一些特点。

表 3.3 VITAL_Primitive 中的模块

标志符	说 明
AND	2,3,4 和 n 输入
OR	2,3,4 和 n 输入
XOR	2,3,4 和 n 输入
NAND	2,3,4 和 n 输入
NOR	2,3,4 和 n 输入
XNOR	2,3,4 和 n 输入
BUFFER	使能端高有效、低有效和无使能端
INVERTER	使能端高有效、低有效和无使能端
MUX	2,3,4 和 n 输入
DECODER	2,4,8 和 n 输入

3.5.2 VITAL 的建模指导方针

VITAL 用标准化的方法定义了 VITAL 风格的建模指导方针，分为两个层次。第 0 层规定模型接口定义应遵循的原则和从 SDF 获得反向标注的方法，实际主要是定义了实体的语言风格；第 1 层规定结构体内代码应遵循的原则。设计者可以只遵循第 0 层方针，以得到反向标注，而依旧以平常的风格编写结构体内的代码；但是，若结构体内遵循第 1 层方针，则实体声明必须遵循第 0 层方针。

第 0 层方针首先规定 VITAL 模型中的所有端口必须是 IEEE 标准库 Stdlogic_1164 中所定义的数据类型，其中一位的端口应为 Std_logic 类型或子类型，总线型端口应为 Std_logic_vector 类型；其次，规定 VITAL 模型中要被用于反向标注的类属参数必须是

Vital_Timing 库中定义的数据类型，它们的命名遵循一定规范，以便从 SDF 中直接获得反馈信息，例如：tpd_A_Y 表示从输入端口 A 到输出端口 Y 的传输延时，tsetup_D_CLK 表示触发器 D 端与时钟端的建立时间；第三，定义了从 SDF 语言结构到 VITAL 模型中类属的直接映射方案，从而保证能够有效地从各个设计阶段（如综合后、布局布线后等）的 SDF 文件中得到反向标注信息。下面是 VITAL 第 0 层实体的例子。

```
--Example of a VITAL Level-0 compliant dntity declaration
library IEEE;
use Std_logic_1164.all;
library VITAL;
use VITAL.vital_timing.all;
use VITAL.vital_primitives.all;
entity AO2 is
    generic(tpd_A1_Q: VitalDelaytype01 := (10 ns, 11 ns);
                                          --A1 到 Q 的延时
            tpd_A2_Q: VitalDelaytype01 := (11 ns, 12 ns);
                                          --A2 到 Q 的延时
            tpd_B1_Q: VitalDelaytype01 := (10 ns, 12 ns);
                                          --B1 到 Q 的延时
            tpd_B2_Q: VitalDelaytype01 := (11 ns, 12 ns));
                                          --B2 到 Q 的延时
    port(A1, A2, B1, B2: in Std_ulogic;
        Q: out Std_ulogic);
    attribute VITAL_Level0 of AO2 : entity is true;
end AO2;
```

第 1 层方针规定了如何用 VITAL 库中提供的基本单元来构造完整的 ASIC 模型。遵循第 1 层方针的模型由以下几部分代码组合而成：

(1) 连线延时。ASIC 单元之间的互连延时在 VITAL 模型中被表示为输入端口的连线延时，通过调用过程 VitalWireDelay 实现，这个过程为输入端口添加了一个连线延时信号，即由输入端口经过传输延时得到。

(2) 时序关系检查。这部分代码检查建立、保持时间等时序关系，如果发生违反时序关系的情况，则 VITAL 例程会把相应输出置为'X'。

(3) 功能描述。这部分代码描述电路的功能，可以使用 VITAL 的预定义模块、VITAL 真值表（组合逻辑电路）、状态表（时序电路）以及各种标准 VHDL 运算符号。

(4) 延时选择和传播。这部分代码完成延迟路径选择、毛刺电平处理等。

这层模型的框架如下：

```
architecture VITAL of XXX is
 --信号、属性等定义
begin
```

```
    --下面的代码计算输入路径延时
        Wire_Delay: Block
        begin
    --为每一输入调用函数 VitalProgragateWireDelay(…).
        end Block;
    --下面是描述行为的进程
        VitalBehaviour: process( * _ipd)
    --定义各种常量、变量等
        begin
    --时序检查部分
    --调用过程 VitalTimingCheck(…)和 VitalPeriodCheck(…)
    --功能描述部分
    --路径延时部分
    -- 为每一输出调用函数 VitalPropragatePathDelay(…)
        end process;
end VITAL;
```

在 VITAL 第 1 层模型中，功能和时域行为用 VITAL 定义的并行过程调用、VITAL 进程或真值表来描述。其中，并行过程调用就是用各种基本的逻辑模块组成整个电路，这种风格适于采用分布式的延时模型，如下面的第(a)段源代码所示。使用进程描述时，第 1 层模型内只能出现 VITAL 进程，并在关键字 process 前加上标号 VITAL_Behavior 注明。VITAL 进程通常包含三部分：时间检查、功能描述和路径延时(每部分都是可选的)。功能部分可以使用真值表或状态表、IEEE 标准 VHDL 运算符和 VITAL 定义的电路单元。为保证仿真速度，VITAL 进程内只能使用某些 VHDL 标准顺序语句。下面的第(b)段源代码即为 VITAL 进程的例子。在描述延时方面，VITAL 提供两种选择，分布参数延时和引脚到引脚的延时。第(a)段源代码是分布参数延时的例子，这时输入描述提供各个器件的延时特性，经过计算获得整个单元的延时参数；第(b)段源代码是引脚到引脚式的延时的例子。真值表法相对简单，这里不再给出。

```
-- 以下为第(a)段源代码
library IEEE;
use Std_logic_1164.all;
library VITAL;
use VITAL.vital_timing.all;
use VITAL.vital_primitives.all;
entity AO2 is
    generic(tpd_A1_Q: VitalDelaytype01 := (10 ns, 11 ns);
                                          --A1 到 Q 的延时
            tpd_A2_Q: VitalDelaytype01 := (11 ns, 12 ns);
                                          --A2 到 Q 的延时
            tpd_B1_Q: VitalDelaytype01 := (10 ns, 12 ns);
```

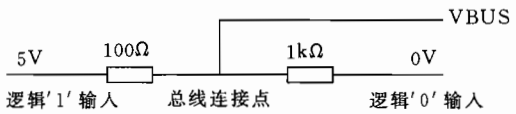
```
                                        --B1 到 Q 的延时
        tpd_B2_Q: VitalDelaytype01 := (11 ns, 12 ns);
                                        --B2 到 Q 的延时
        tdevice_O1: VitalDelaytype01 := (9 ns, 8 ns);
                                        --或门输入 1 到输出的延时
        tdevice_O2: VitalDelaytype01 := (7 ns, 8 ns));
                                        --或门输入 2 到输出的延时
    port(A1, A2, B1, B2: in Std_ulogic;
        Q: out Std_ulogic);
    attribute VITAL_Level0 of AO2 : entity is TRUE;
end AO2;

architecture DistributedDelay of AO2 is
    attribute VITAL_LEVEL1 of DistributedDelay: architecture is TRUE;
    signal Temp1, Temp2: Std_logic;
begin
    I1: VITALAnd2(Q => Temp1, A => A1, B => A2,
        tpd_A_Q => tpd_A1_Q, tpd_B_Q => tpd_A2_Q);
    I2: VITALAnd2(Q => Temp2, A => B1, B => B2,
        tpd_A_Q => tpd_B1_Q, tpd_B_Q => tpd_B2_Q);
    I3: VITALOr2(Q => Q, A => Temp1, B => Temp2,
        tpd_A_Q => tdevice_O1, tpd_B_Q => tdevice_O2);
end DistributedDelay;
-- 以上为并行过程调用和分布参数延时

-- 以下为第(b)段源代码
library IEEE;
use Std_logic_1164.all;
library VITAL;
use VITAL.vital_timing.all;
use VITAL.vital_primitives.all;
entity AO2 is
    generic(TIPD_A1: VitalDelaytype01 := (tr01 => 0 ns, tr10 => 0 ns);
      TIPD_A2: VitalDelaytype01 := (tr01 => 0 ns, tr10 => 0.1 ns);
      TIPD_B1: VitalDelaytype01 := (tr01 => 0.1 ns, tr10 => 0 ns);
      TIPD_B2: VitalDelaytype01 := (tr01 => 0.1 ns, tr10 => 0.1 ns);
      TPD_A1_Q: VitalDelaytype01 := (10 ns, 11 ns);
                                        --A1 到 Q 的延时
```

```
            TPD_A2_Q: VitalDelaytype01 := (11 ns, 12 ns);
                                        --A2 到 Q 的延时
            TPD_B1_Q: VitalDelaytype01 := (10 ns, 12 ns);
                                        --B1 到 Q 的延时
            TPD_B2_Q: VitalDelaytype01 := (11 ns, 12 ns);
                                        --B2 到 Q 的延时
            VitalTimingCheckOn: Boolean := FALSE);
    port(A1, A2, B1, B2: in Std_ulogic;
        Q: out Std_ulogic);
    attribute VITAL_Level0 of AO2 : entity is TRUE;
end AO2;

architecture PinToPinDelay of AO2 is
    attribute VITAL_LEVEL1 of PinToPinDelay: architecture is TRUE;
    signal A1_IPD, A2_IPD, B1_IPD, B2_IPD: Std_ulogic := 'U';
    signal TEMP1, TEMP2: Std_logic;
begin
    Wire_Delay: Block
    begin
        VitalWireDelay(A1_IPD, A1, TIPD_A1);
        VitalWireDelay(A2_IPD, A1, TIPD_A2);
        VitalWireDelay(B1_IPD, A1, TIPD_B1);
        VitalWireDelay(B2_IPD, A1, TIPD_B2);
    end Block;

    VITAL_Behavior: process(A1, A2, B1, B2)
        variable Q_ZD: Std_logic: 0_DeladyOutput;
        variable GLITCHDATA_Q: GlitchDatatype;
    begin
        Q_ZD := (A1_IPD and A2_IPD) or (B1_IPD and B2_IPD);
        VitalPathDelay01(Q, GLITCHDATA_Q, 'Q', Q_ZD,
            (A1'Last_Event, TPD_A1_Q, TRUE),
            (A2'Last_Event, TPD_A2_Q, TRUE),
            (B1'Last_Event, TPD_B1_Q, TRUE),
            (B2'Last_Event, TPD_B2_Q, TRUE));
    end process;
end PinToPinDelay;
-- 以上为引脚到引脚的延时
```

VITAL 的库文件和建模指导方针使得仿真软件制造商能够把以前门级仿真器的许多优化技术移植到 VHDL 仿真器上，这些技术包括子程序优化、状态表/真值表优化等，从而能够大大提高仿真效率。Viewlogic 公司的研究工作表明 VITAL 仿真器在性能上已经与 Verilog HDL 的门级仿真器相当。Cadence 公司的工作证明 VITAL 可以提供与 Verilog 相当的仿真精度和仿真性能。

3.6 多值逻辑

在本章的前几节中，已经使用了 IEEE 定义的 9 值逻辑，但只用到了'0'，'1'，'Z'和'U'这几个数值。如果只考虑理想情况，'0'和'1'两个状态就足以描述数字电路；但是实际的电路中，会发生各种复杂问题。比如，某个总线信号由两个驱动器驱动，如果第一个驱动器输出'0'，第二个驱动器输出为'1'，总线上的数值就是未知'X'。因为这种情况下不知道两个驱动器输出值的强度，所以说总线上的值是'X'。对于不同的工艺，逻辑'0'和逻辑'1'的输出电阻不同，或者说强度不同，若把两个驱动器的输出连接在一起，连接点的取值并不是完全未知的。例如对于图 3.6 所示的情况，逻辑信号'1' 的输出电阻为 100Ω，逻辑信号'0'的输出电阻为 1kΩ，两个信号同时驱动总线，总线上的信号电平应该为 4.54V。对于一般逻辑电路来讲，这样的电平应该被认为是逻辑'1'。这就是逻辑信号'1'的强度比逻辑'0'的信号强度高的情况。由于向总线写数据是数字电路中常见的操作，所以在逻辑门级器件建模时应该考虑信号的强度。在仿真时根据信号强度来确定某些信号的取值，使仿真给出更精确的信息。



VBUS=(1000*5V)/1100=4.54V

图 3.6 强'1'和弱'0'相连接的情况

如果对每个信号给定信号强度，信号强度应该分为多少等级呢？如果在电路级设计电路，用 SPICE 对电路进行仿真，信号强度的等级实际上有无限多。对于数字器件的造型，显然并不需要对信号等级划分很细。通常可以对数字信号的强度分成两个等级：强信号或弱信号。下面采用两级强度构造出 9 值逻辑系统 Std_logic_1164，即 IEEE 9 值逻辑。

在 IEEE 9 值逻辑中如果把状态看作器件内部存储的信息，状态与强度的组合看作是器件接口处提供的信息，并把数字电路的不同状态和不同强度组合起来，就可以得到不同的逻辑值。三种状态('1'，'0'和'X')和两种强度的组合可以得到六种不同的逻辑值。本节所采用的 Std_logic 中还有第七个逻辑值'Z'。假定高阻态的输出电阻很高，使得无论其内部状态是'1'，'0'还是'X'，都不会影响器件输出值。此外，为了全面描述电路，还需要增加以下两个状态。

(1) 'U'：没有初始化的状态。这一状态用来检查时序逻辑电路是否被用适当的方式初始化。带存储功能的器件，比如触发器，在仿真开始时首先被初始化为'U'。如果电路初

始化的逻辑功能正确，所有'U'都应该在有限仿真时间内变成确定态'0','1','H','L'或'X'。状态'U'不同于'X'的地方在于'U'只能在初始化过程中产生，'X'则可以在仿真过程中产生。在运算过程中，'X'和'U'的代数特征相同。

(2)'—'：无关态。这一数值用于表示逻辑综合过程中与布尔函数优化无关的状态。在仿真过程中，'—'的效果与'Z'完全相同。这样，就得到了下面定义的数据类型std_logic。各数值的意义如表3.4所示。

表3.4 IEEE标准1164的9值系统中各逻辑值的意义

内部状态	强度	取值
高电平	强	'1'
	弱	'H'
	高阻态	'Z'
低电平	强	'0'
	弱	'L'
	高阻态	'Z'
不确定	强	'X'
	弱	'W'
	高阻态	'Z'
未初始化态	/	'U'
无关态	/	'—'

```
TYPE Std_logic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '—');
```

相应地，可以定义数组如下：

```
type Std_ulogic_vector is Array (Natural Range <> ) of Std_ulogic;
```

采用9值系统构造逻辑器件模型时，如果把多个输出连接在一起，需要决断函数确定多驱动产生的信号的数值。Std_ulogic的决断函数是线与函数，其真值表由下面代码定义。

```
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')   -- |—|
  );
```

为了解释这段代码定义的真值表的作用，下面首先给出信号驱动器的优先级的定义。假定 X 和 Y 是某个信号的两个驱动器，即 X 和 Y 的输出都送到这个信号，用 * 来表示两变量的总线决断函数，如果 X * Y = X，则说驱动器 X 的强度高于驱动器 Y，或说驱动器 X 的优先级高于驱动器 Y。

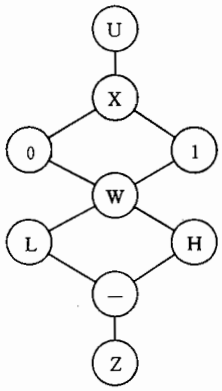


图 3.7　IEEE 9 值逻辑系统的取值晶格图

信号驱动器的优先级可以用图 3.7 所示的 Std_logic 取值的晶格图来描述。在这个晶格图中，上部节点优先级高于下部节点。如果晶格图中连线两端的两个值驱动同一信号，决断函数得到的值是连线上方节点的值。若晶格图中高度相同的两个值驱动同一信号，决断函数得到的值是这两个值最近上方的值。按信号强度讲，可以得到下述 4 条结论：① 强信号的优先级高于弱信号；② 弱信号的优先级高于高阻态；③ 如果两个信号的强度相同，则它们驱动同一信号得到的结果是与驱动信号强度相同的不确定值；④ 如果两个高阻输出驱动同一信号，得到的信号值仍为高阻值。按上面代码定义的真值表，可以得到如下总线决断函数：

```
function resolved (s : Std_ulogic_vector) return Std_ulogic is
    variable result  :  Std_logic  := '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;
```

在这个总线决断函数中，通过查表的方法，每次对两个驱动器进行比较，对各驱动器出现的顺序并没有特殊要求。

定义了信号的总线决断函数之后，就可以定义将 Std_ulogic 型信号决断后的数据子类型。Std_logic 和 Std_logic_vector 分别为 Std_ulogic 型信号决断后得到的标量数据类型和矢量数据类型：

```
type Std_logic is resolved Std_ulogic;
type Std_logic_vector is array (Natural Range <> ) of Std_logic;
```

第4章 系统级设计

本章讨论结构化的集成电路设计过程，图 4.1 示意地给出了本章讨论的问题。在系统级，硬件的指标规范用自然语言描述，还可能伴随有方框图或时序图。系统级设计是硬件设计过程的第一步，是把硬件的自然语言描述转换为真值表、状态图或算法模型的过程。将自然语言描述转换为真值表或状态图，要求设计者对硬件的功能有较深入的理解，或对硬件结构做了一些限制（比如选用组合逻辑或时序逻辑电路实现等），而且一旦真值表或状态图确定后，硬件结构也随之确定。将硬件的自然语言描述转换为算法模型，是将自然语言描述直接映射为 VHDL 模型，不对电路结构和形式预先作任何限制。设计开始时，可以把硬件行为的自然语言描述转换为这三种形式中的任何一种。本章重点讨论如何将硬件系统的自然语言描述转换为算法模型。

图 4.1 中在算法模型和真值表之间、算法模型和状态图之间画出了双向连线，这意味着可以先设计出真值表或状态图，然后将它们转换为算法模型；也可以先设计出算法模型，然后将它转换为真值表或状态图。虽然存在这种双向关系，但通常是首先建立算法模型。与真值表和状态图相比，硬件描述语言构造的算法模型更加灵活，有利于全面搜索设计空间，从而有可能实现更高的设计质量。

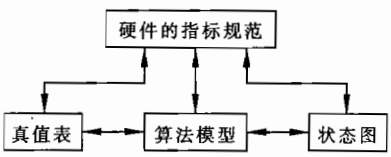


图 4.1 结构化集成电路设计的最高层次

系统级设计中，需要解决的主要问题是系统算法的研究、系统的设计划分以及硬件系统通信协议的确定。本章着重说明如何从系统设计的角度建立算法模型，如何把系统划分为寄存器传输级的子系统以及如何在寄存器传输级实现通信。

4.1 在行为域内构造硬件的行为模型

用 VHDL 语言构造的硬件算法模型由一系列相互关联的过程组成。构造算法模型实际上就是把描述系统功能的自然语言翻译为一组进程，每个进程完成不同功能。完成这一转换，需经过如下几步：

（1）把描述系统功能的文字分组，每组映射为一个进程或块语句。这一步隐含着设计的划分。

（2）对每个进程确定激活进程的条件以及进程激活后的动作。

（3）写出 VHDL 源代码，完成进程激活后的动作。

4.1.1 进程模型图(PMG)

进程模型图是 VHDL 行为描述的图形表示,它可以帮助设计者完成将自然语言行为描述转换为算法模型的第一步工作。在 VHDL 中,结构体用来描述硬件的行为,每个结构体内都包含一些并行运行的进程。这里所说的进程是广义的,既包括用 VHDL 关键字 process 明确指出的进程,也包括并行语句。按照 IEEE 标准,并行语句相当于简化的进程,对赋值号右边的所有信号敏感。进程模型图实际上是结构体行为的图形表示。图 4.2 是一个典型的进程模型图,图中的支路是进程间的信号,有些支路上不但标出了信号名而且标出了相应的传输延时。比如,图 4.2 中的 S(DEL_S),S 是信号名,DEL_S 是进程间传输信号 S 时的传输延时。图中的节点表示进程,比如第二个节点可以为下面给出的 VHDL 进程。

```
PROC2: process(X, Y, Z)
    variable SVL, OVL: Std_logic;
begin
    -- 进程的行为描述
    -- 计算变量 SVL 和 OVL 的数值
    S <= SVL after DEL_S;
    OUT1 <= OVL after DEL_OUT1;
end PROC2;
```

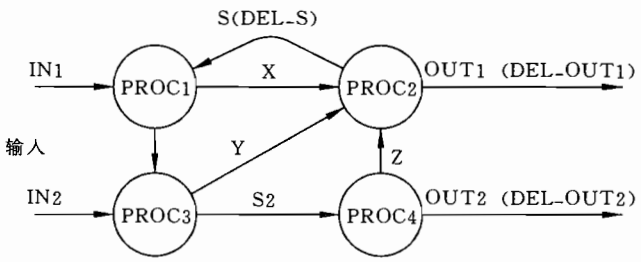


图 4.2 典型的进程模型图

在这个进程模型图中,进程对信号 X,Y,Z 敏感,用来计算 S 和 OUT1 的数值,其中 DEL_S 是为信号 S 赋值时的延时,DEL_OUT1 是为信号 OUT1 赋值时的延时。

除了描述器件的行为之外,进程模型图还表示了硬件模型的划分,这里的划分可以是物理意义上的划分,也可以是纯功能上的划分。如果表示的是物理意义上的划分,图中支路上的延时值表示信号经过逻辑门、触发器、寄存器等实际硬件时的信号传输延时。在某些情况下,特别是在硬件设计的初级阶段,进程模型图只表示纯功能上的划分,图 4.3 示意了这种情况。这是一个微处理器的进程模型图,其中的取指、运行等节点的功能描述与硬件的模块划分没有任何关

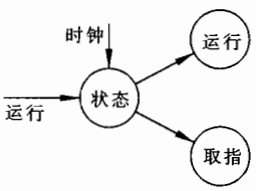


图 4.3 微处理器的一种进程模型图

系。在实际实现芯片硬件时，不同功能模块可能会共享某些电路模块。比如在这个微处理器中，取指和运行两个节点可能共享同一物理存储区，而取指的地址计算和执行中的算术逻辑运算也可能共享运算电路。

4.1.2 实例 1：并串转换电路

这个例子是一个并行数据到串行数据转换电路的算法模型，图 4.4 是电路的方框图，电路行为的自然语言描述如下：在时钟沿上，如果控制信号 LD 为高电平'1'，则把并行输入信号 PAR_IN 加载到转换电路内部，并将状态标志信号 BUSY 置位为高电平'1'。此后并行输入的数据在时钟 CLK 上升沿上逐位移出转换电路。当加载的并行数据全被移出电路之后，状态标志 BUSY 复位为'0'。

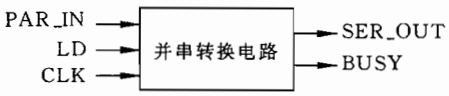


图 4.4 并串转换电路的方框图

按照 4.1 节说明的方法，先将自然语言描述分组，然后把每组描述映射为一个 VHDL 进程，就可以把并串转换电路映射为加载进程 Load 和移位进程 Shift。

(1) 进程 Load：① 在时钟上升沿，若信号 LD 为'1'，8 位并行输入信号 PAR_IN 被加载到转换电路的内部寄存器，并将状态标志 BUSY 置位为'1'；② 在数据被全部移出转换电路之前，BUSY 保持为'1'。在标志 BUSY 为'1'期间，禁止再次加载数据。

(2) 进程 Shift：① 数据在时钟 CLK 上升沿逐位移出电路；② 8 位数据全部移出之前，移位中止信号 SH_COMP 一直保持为'0'。

图 4.5 是并串转换电路的算法模型的进程模型图。下面的 VHDL 源代码是依照这个进程模型图书写的 VHDL 算法级模型。模型由两个进程组成，进程 Load 在时钟沿上检查 LD 是否为'1'，如果满足，则开始转换过程。首先将并行数据载入内部寄存器，并将 BUSY 位置为'1'，此后用一个 while 循环等待 8 位数据移出，如果 SH_COMP 为'1'，表示 8 位数据串行输出已经结束，则将 BUSY 复位为'0'。进程 Shift 在时钟沿上检查 BUSY 是否为'1'，如果满足，则进行必要的初始化，即将 SH_COMP 置为'0'，把移位计数值置为 7，并用 9 位的内部移位寄存器锁存 8 位数据(在低端补'0')。在接下来的时钟上升沿，依次右移一位，并将最低位输出；8 次移位结束之后，进程置 SH_COMP 为'1'，表示并串转换结束。两个进程使用握手协议实现进程间的通信。

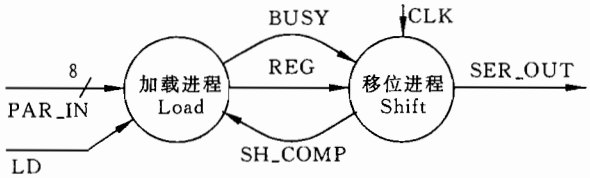


图 4.5 并串转换电路的进程模型图

```
entity Par_to_Ser is
    port(CLK, LD : in Std_logic;
         PAR_IN : in Std_logic_vector(7 downto 0);
         BUSY : buffer Std_logic;
         SER_OUT : out Std_logic);
end Par_to_Ser;

architecture ALG of Par_to_Ser is
    signal SH_COMP : Std_logic;
    signal REG : Std_logic_vector(7 downto 0);
begin
    Load: process
    begin
        wait until CLK'Event and CLK='1' and LD='1';
        REG <= PAR_IN;
        BUSY <= '1';
        Waiting: loop
            wait until CLK'Event and CLK='1';
            if SH_COMP='1' then
                BUSY <= '0';
                exit Waiting;
            end if;
        end loop Waiting;
    end process Load;

        Shift: process
            variable COUNT : Integer;
            variable OREG : Std_logic_vector(8 downto 0);
        begin
            wait until CLK'Event and CLK='1' and BUSY='1';
            COUNT := 7;
            OREG := REG & '0';
            SER_OUT <= OREG(0);
            SH_COMP <= '0';
            while (COUNT >= 0) loop
                wait until CLK'Event and CLK='1';
                OREG := '0' & OREG(8 downto 1);
                SER_OUT <= OREG(0);
```

```
            COUNT := COUNT - 1;
        end loop;
        wait until CLK'Event and CLK='1';
        SH_COMP <= '1';
    end process Shift;
end ALG;
```

4.1.3 实例 2：移位乘法器的算法模型

图 4.6 是一个移位乘法器的方框图，该电路行为的自然语言描述如下：

移位乘法器的输入为两个 4 位操作数 A 和 B，闸门信号 STB 启动乘法操作，时钟信号 CLK 提供系统定时。乘法结果为 8 位信号 RESULT，乘法结束后置信号 DONE 为'1'。

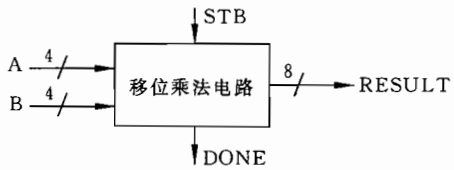


图 4.6　移位乘法器的方框图

乘法算法采用原码移位乘法，即对两操作数进行逐位的移位相加，迭代 4 次后获得乘法结果。具体算法为：① 在被乘数和乘数的高位补'0'后扩展成 8 位。② 乘数依次向右移位，并且检查其最低位，如果该位为'1'，则将被乘数与部分积的和相加，然后被乘数向左移位；如果最低位为'0'，则仅仅对被乘数进行移位操作。移位时，乘数的高端和被乘数的低端均移入'0'。③ 当乘数变成全'0'后(最坏情况下需要 4 次移位)，乘法结束。如前所述，将硬件描述的自然语言形式转换为算法模型的第一步，要求把自然语言描述分组，然后把每组描述映射为一个 VHDL 进程。下面把乘法器电路映射为控制器进程 CONTROLLER、锁存移位进程 SRA 和 SRB、加法进程 ADDER 以及锁存结果的进程 ACC。进程的具体情况如下。

控制器进程 CONTROLLER：接收到有效 STB 信号后，产生初始化信号，然后依次在各个有效时钟沿进行状态转移，并在各个状态发出不同控制信号，使其他部件执行相应操作，在 SRA 输出的 8 位数据均为'0'时，结束乘法操作；

锁存移位进程 SRA：初始化时锁存 B 路数据，并在 4 位数据前补'0'至 8 位，然后在移位信号 SHIFT 有效时依次右移，在左端移入'0'；

锁存移位进程 SRB：初始化时锁存 A 路数据，并在 4 位数据前补'0'至 8 位，然后在移位信号 SHIFT 有效时依次左移，在右端移入'0'；

加法进程 ADDER：这是一个组合逻辑进程，计算 SRB 进程的输出与 ACC 进程锁存结果之和，即部分积的和；

锁存结果的进程 ACC：在 CONTROLLER 进程产生的 ADD 信号有效时锁存部分积的和。

图 4.7 是这个乘法器算法模型的进程模型图。由于以上的移位和锁存操作都在时钟上升沿进行，所以这是完全同步式的设计，图中省略了 CLK 信号。

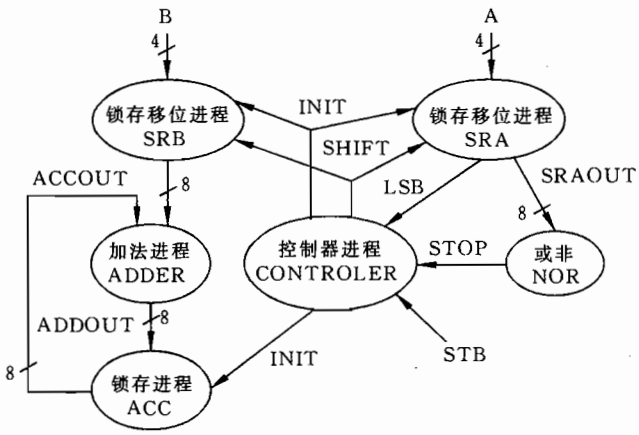


图 4.7　移位乘法器的进程模型图

控制器是整个硬件的核心，下面给出的是控制器 VHDL 代码。这是一个纯算法式的描述，状态是隐含的。控制器在时钟上升沿检测 STB，若 STB 为'1'，就启动一次乘法操作。控制器首先发出一个 INIT 脉冲，初始化输入和输出锁存器。然后进入 while 循环，用一个状态检查 LSB，如果 LSB 为'1'，则先执行加法操作（置 ADD 为'1'），再执行移位操作（置 SHIFT 为'1'）；否则，只执行移位。当 SRAOUT 的各位均为'0'时，循环中止，乘法结束，控制器置 DONE 信号为'1'，表示乘法结束，然后进程挂起直到下一次 STB 有效。

```
CONTROLLER: process
begin
    wait until CLK'Event and CLK='1' and STB = '1';
    DONE <= '0';
    INIT <= '1';                                    --初始化各个锁存器
    SHIFT <= '0'; ADD <= '0';
    wait until CLK'Event and CLK='1';
    INIT <= '0';
    run_loop: while (Stop /= '1') loop
                --Stop <= SRAOUT 各位的或非运算结果
        if LSB='1' then                             --LSB 是 SRAOUT 的最低位
            wait until CLK'Event and CLK='1';
            ADD='1';
            wait until CLK'Event and CLK='1';
            ADD <= '0';
            SHIFT <= '1';
            wait until CLK'Event and CLK='1';
```

```
            else
                wait until CLK'Event and CLK='1';
                SHIFT <= '1';
                wait until CLK'Event and CLK='1';
            end if;
            SHIFT <= '0';
        end loop run_loop;
        DONE <= '1';
    end process;
```

锁存移位进程 SRA 和 SRB 的 VHDL 代码如下，可以看出，这是一个典型的移位锁存器。锁存移位进程 SRB 与 SRA 的不同之处只是移位方向的不同。

```
    SRA: process
    begin
        wait until CLK'Event and CLK='1';
        if INIT = '1' then
                SRAOUT <= "0000" & A;
        elsif SHIFT = '1' then
                SRAOUT <= '0' & SRAOUT(7 downto 1);
        end if;
    end process SRA;

    SRB: process
    begin
        wait until CLK'Event and CLK='1';
        if INIT = '1' then
                SRBOUT <= "0000" & B;
        elsif SHIFT = '1' then
                SRBOUT <= SRBOUT(6 downto 0) & '0';
        end if;
    end process SRB;
```

下面给出的是加法进程 ADDER 的 VHDL 代码，这是一个组合逻辑进程，用于计算部分和。这个加法器使用了普通的“异或”生成和算法。

```
    ADDER: process(ACCOUT, SRBOUT)
        variable SUM, TMP1, TMP2 : Std_logic_vector(7 downto 0);
        variable CARRY : Std_logic;
    begin
        TMP1 := ACCOUT;
```

```
    TMP2 := SRBOUT;
    CARRY := '0';
    for I in 0 to 7 loop
        SUM(I) := TMP1(I) xor TMP2(I) xor CARRY;
        CARRY := (TMP1(I) and TMP2(I)) or (TMP1(I) and CARRY) or
                 (TMP2(I) and CARRY);
    end loop;
    ADDOUT <= SUM;
end process ADDER;
```

最后是锁存器进程 ACCD 的 VHDL 代码。这个模型非常简单，就是在初始化时清'0'，在加法操作时同步锁存运算结果。

```
ACC: process
begin
    wait until CLK'Event and CLK='1';
    if INIT = '1' then
        ACCOUT <= (others => '0');
    elseif ADD='1' then
        ACCOUT <= ADDOUT;
    end if;
end process ACC;
```

在乘法器的设计中，除了控制器和加法器之外的电路都比较简单，因此 SRA，SRB，ACC 这三个进程实际上是使用寄存器传输级的风格书写的。下面的例子由两个稍微复杂些的进程组成。

4.1.4 实例 3：考虑时序关系的算法模型

在前面的例子中，并没有将硬件的时序关系考虑在算法模型中，本节讨论如何考虑硬件的输入输出的时序关系。图 4.8 是一种带有输出缓冲单元的寄存器的方框图(a)及其时序关系(b)，该寄存器行为的自然语言描述如下：在闸门信号 STRB 的上升沿加载寄存

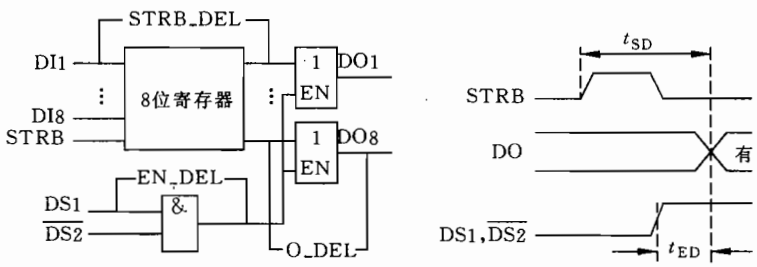


图 4.8 带有输出缓冲单元的寄存器

器，如果使能信号打开输出缓冲单元，被加载的数据延时 t_{SD} 后，出现在输出端。打开输出缓冲单元的控制信号（使能信号）是 DS1 and (not DS2)。使能信号的变化延时 t_{ED} 之后，影响缓冲单元的输出。

分析硬件的方框图及指标规范，不难发现从输入到输出有两条路径：第一条是从寄存器的数据输入到缓冲器的输出，第二条是从使能信号输入到缓冲器的输出。建立硬件的算法模型时，每条路径应该对应一条信号线。实际上，建立硬件模型时要求遵循下述准则：如果要求算法模型包含硬件的时序关系，则从输入到输出之间的每一条路径必需由一个单独的 VHDL 信号描述，这些信号可以并行活动。

遵循这一准则，在每条路径上的信号同时活动时，可以保证算法模型行为的正确性。为了构造带缓冲单元的寄存器的算法模型，假定 t_{SD} = STRB_DEL + O_DEL，t_{ED} = EN_DEL + O_DEL。如图 4.8(a)所示，STRB_DEL 是从闸门信号 STRB 的上升沿到数据寄存器输入端的延时，EN_DEL 是从使能信号输入到缓冲单元打开的延时，O_DEL 是信号经过缓冲单元的延时。上述带缓冲单元的寄存器可以由三个进程描述，各进程的行为描述如下。

(1) 进程 PREG：在闸门信号 STRB 的上升沿将 DI 的输入数据加载到内部寄存器。

(2) 进程 ENABLE：产生缓冲单元的使能信号，缓冲单元的使能条件是 DS1 and (not DS2)。

(3) 进程 OUTPUT：如果使能信号将输出缓冲单元打开，则输出在 t_{SD} 之后变化；如果使能条件发生变化，则输出在 t_{ED} 之后变化。

图 4.9 是该带缓冲单元的寄存器的进程模型图。进程 PREG 接收闸门信号 STRB 和数据输入信号 DI，数据延时后通过信号 REG 送到进程 OUTPUT。进程 ENABLE 的作用与进程 PREG 类似，把延时后的使能信号 ENABLD 送到进程 OUTPUT。进程 OUTPUT 则接收信号 REG 和 ENABLD，产生输出信号 DO。进程模型图中三个路径的延时分别用 STRB_DEL，EN_DEL 和 O_DEL 标在图中。这个进程模型图符合提出的算法模型构造准则，即不同的数据路径分别用不同的信号描述。

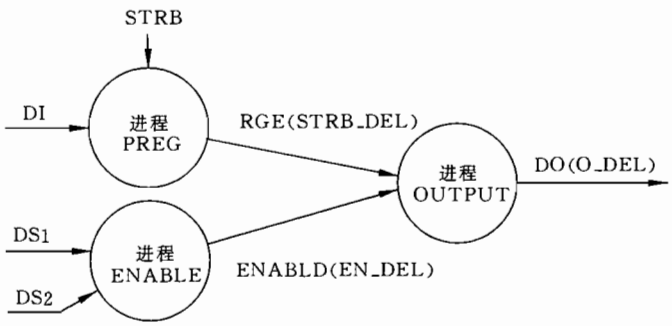


图 4.9　带缓冲单元的寄存器的进程模型图

下面是这三个进程的 VHDL 源代码，其中接口描述及三个延时都通过类属参数描述。如果将该模型连接到某系统中成为系统内部的一个具体寄存器，需要指定这些类属参数的具体数值。

```
entity Buff_Reg is
    generic(STRB_DEL, EN_DEL, O_DEL: TIME);
    port (STRB, DS1, DS2: in Std_logic; DI: in Std_logic_vector(7 downto 0);
        DO: out Std_logic_vector(7 downto 0) );
end Buff_Reg;

architecture Three_Proc of Buff_Reg is
    signal ENABLD: Std_logic;
    signal REG: Std_logic_vector(1 to 8);
begin
    PREG: process(STRB)
    begin
        wait until STRB='1';
        REG <= Di after STRB_DEL;
    end process PREG;

    ENABLE: process (DS1, DS2)
    begin
        ENABLD<= DS1 and not DS2 after EN_DEL;
    end process ENABLE;

    OUTPUT: process(REG, ENABLD)
    begin
        if ENABLD='1' then
            DO <= REG after O_DEL;
        else
            DO <= "ZZZZZZZZ" after O_DEL;
        end if;
    end process OUTPUT;
end Three_Proc;
```

当然,还可以写出其他形式的算法模型,比如可以将两个进程 PREG 和 ENABLE 合并为一个进程,用一个进程完成这两个进程的工作,这种方式如下:

```
FRONT_END: process
begin
    if STRB'Event and STRB='1' then
       REG <= DI after STRB_DEL;
    end if;
```

```
    if DS1'Event or NDS2'Event then
        ENABLD <= DS1 and not NDS2 after EN_DEL;
    endif;
end process ENABLE;
```

这里用包含两个动作的一个进程替代了原来的两个只包含单个动作的进程，对信号REG和ENABLD赋值的方式并没有变化。这就提出了一个问题：在设计硬件的算法模型时，多用几个进程好呢还是少用几个进程好？回答这个问题需要考虑如下几个因素：

(1) 信号的个数。进程越多所要求的进程间的信号个数就越多。在仿真器中，信号个数越多，则要求事件队列的个数越多且队列长度越长，这将影响仿真效率。

(2) 进程的复杂程度。一个进程要完成的动作太多，进程就会复杂，进程的通用性、可再用性就会变差。

(3) 映射的困难程度。进程个数多一些有利于将硬件行为的自然语言描述转换为进程描述。

把算法模型分解为一系列进程就是系统划分。在对系统进行划分时，子模块的粒度，即把多大规模的逻辑电路组织在同一模块(在算法级主要指进程)中是很重要的问题。如果规模太大，则仿真、综合或改写成寄存器传输级代码的时间都比较长，由于VLSI设计是一个迭代过程，所以这样就会延长设计周期；反之，若规模太小，则限制了设计能力，同样会影响设计效率。合理的划分，不仅可以改善设计的总体性能，提高设计的可再用性，而且可以节省仿真时间和综合时的编译时间。系统划分应该按照下面的原则来进行：

(1) 把相关的组合逻辑划分到同一进程中；

(2) 把结构规则的逻辑模块(如多位加法器)和随机逻辑模块(如指令译码的硬连线逻辑)分开到不同的子模块中；

(3) 每一进程完成的任务尽量单一化。

根据VHDL语言的规定，单独出现的信号赋值语句与进程等效。因此，可以把上面描述带缓冲单元的寄存器的结构体写得更紧凑一些，即

```
architecture Data_Flow of Buff_Reg is
begin
    B1: block(STRB='1' and STRB'Event)
        signal ENABLD:Std_logic;
        signal REG: Std_logic_vector(1 to 8);
    begin
        REG <= guarded DI after STRB_DEL ;          --等效于进程 PREG
        ENABLD <= DS1 and not NDS2 after EN_DEL;
                                                    -- 等效于进程 ENABLE
        DO <= REG after O_DEL when ENABLD = '1' else
                    "ZZZZZZZZ" after O_DEL;         --等效于进程 OUTPUT
    end block B;
end Data_Flow;
```

这里采用了保护模式赋值语句对信号 REG 赋值，只有在保护条件为 TRUE，即在信号 STRB 从'0'变为'1'的上升沿，赋值语句才会执行。这种方式得到的硬件模型很简洁，但模型的结构不明显。这种简洁的算法模型通常只用于数据流式硬件描述。

4.1.5 时序检查

按照 VHDL 中的规定，信号赋值的缺省延时方式为惯性延时，信号赋值语句执行时会把输入信号中窄于指定延时时间的脉冲滤除。这样，在算法模型中，输入信号的值必需保持足够长才能保证模型的行为正确。比如在 D 触发器中 CLK 的上升沿之前信号 D 的值至少要保持一定的建立时间，在 CLK 的上升沿之后至少要保持一定的保持时间等等。如果输入信号不满足这些时序要求，输出信号的值可能不会按设计者设想的方式变化，除此之外，硬件模型并不能直接告诉设计者输入信号的一些变化被滤除了，这不利于模型的仿真验证。实际中对于一个信号本身来讲，其上升时间、下降时间、最小脉冲宽度也常常需要检查，即所谓波形检查。图 4.10 以图形化的方式说明了这些概念。

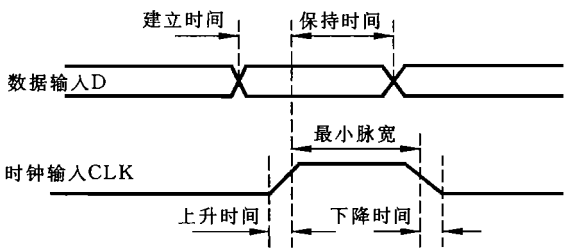


图 4.10 D 触发器的输入信号时序要求

VHDL 语言允许在模型中检查这些时序要求，这要用到断言语句(assert 语句)。断言语句的一般形式为 assert 布尔表达式，report 错误信息，severity 错误级别。

断言语句执行时，先检查关键字 assert 后面的布尔表达式的值，如果该值为 TRUE，则认为是正常条件；如果表达式的值为 FALSE，则认为是非正常工作条件，并报告出指定的错误信息。实际上，断言语句可以用来报告很多有用的信息，本节只用它来进行时序检查。对于时序检查，断言语句中的布尔表达式中通常要用到信号的 VHDL 预定义属性，使用比较多的属性是 X'Event，X'Last_value，X'Stable(T)和 X'Delayed(T)等。

检查时序的断言语句可以放在设计实体的接口描述中，也可以放在结构体中。如果把断言语句放在结构体内，则时序检查对设计实体的特定结构有效；如果断言语句放在实体的接口描述中，则它是对该设计实体检查时序，即对该实体的所有可能的结构体全进行时序检查。如果时序检查代码放在结构体中，可以出现过程调用、并行断言语句或进程内的顺序断言语句三种形式。通常认为，过程调用形式的时序检查代码具有可再用性好、代码简洁、仿真效率高(仿真效率包括内存的使用量和仿真时间两个方面)等优点。下面的 VHDL 代码是一种通用的波形检查过程，代码利用了 VHDL 中预定义的数据 now，即当前仿真时间，通过追踪波形的上升和下降沿来计算波形的各种宽度。

```
procedure Waveform_Check(signal SIG : Std_ALU;
```

```
    constant T_MIN_HL, T_MAX_HL : time := 0 ns;
    constant T_MIN_LH, T_MAX_LH : time := 0 ns;
    constant PW_HI_MAX, PW_HI_MIN : time := 0 ns;
    constant PW_LO_MAX, PW_LO_MIN : time := 0 ns;
    constant SEVERITY_SEL : Severity_level := Warning) is
    variable RISING_AT, t1 : time := 0 ns;
    variable FALLING_AT,t0 : time := 0 ns;
begin
    Checking: loop
        wait until SIG='1' and SIG'Event;
        --上升沿时序检查
        if now > 0 ns then
            assert (now - RISING_AT >= T_MIN_LH) or (T_MIN_LH =
            0 ns)
                report "上升沿过窄"
                severity SEVERITY_SEL;
            assert (now - RISING_AT <= T_MAX_LH) or (T_MAX_LH
            = 0 ns)
                report "上升沿过宽"
                severity SEVERITY_SEL;
        end if;
        t1 := now;
        wait until SIG'Event and SIG'Last_value='1';
        FALLING_AT := now;
        --高电平宽度检查
        assert (now - t1 >= PW_HI_MIN) or (PW_HI_MIN=0)
            report "高电平脉冲过窄"
            severity SEVERITY_SEL;
        assert (now - t1 <= PW_HI_MAX) or (PW_HI_MAX=0)
            report "高电平脉冲过宽"
            severity SEVERITY_SEL;
        wait until SIG'Event and SIG'='0';
        t0 := now;
        --下降沿时序检查
        assert (now - FALLING_AT >= T_MIN_HL) or (T_MIN_HL =
        0 ns)
            report "下降沿过窄"
            severity SEVERITY_SEL;
```

```
        assert (now - FALLING_AT <= T_MAX_HL) or (T_MAX_HL =
        0 ns)
            report "下降沿过宽"
            severity SEVERITY_SEL;
        wait until SIG'Event and SIG'Last_value='0';
        RISING_AT := now;
        --低电平宽度检查
        assert (now - t0 >= PW_LO_MIN) or (PW_LO_MIN=0)
            report "低电平脉冲过窄"
            severity SEVERITY_SEL;
        assert (now - t0 <= PW_LO_MAX) or (PW_LO_MAX=0)
            report "低电平脉冲过宽"
            severity SEVERITY_SEL;
    end loop Checking;
end Waveform_check;
```

与此类似，可以写出用于检查建立时间和保持时间的检查 VHDL 代码，如下面的 VHDL 源代码所示。检查建立时间时，要求时钟上升沿出现时被采样信号必须稳定足够时间。检查保持时间时，要求时钟上升沿后被采样信号至少保持一定时间。

```
-- 建立时间检查的 VHDL 源代码
procedure Setup_Check(signal SIG, CLK : Std_ALU;
    constant SETUP_DEL : time := 0 ns;
constant SEVERITY_SEL : SEVERITY_LEVEL := Warning) is
begin
assert not ((CLK'Event and CLK='1') and (not SIG'Stable(SETUP_DEL)))
    report "建立时间不足"
    severity SEVERITY_SEL;
end Setup_check;

-- 保持时间检查的 VHDL 代码
procedure Hold_Check(signal SIG, CLK : Std_ALU;
    constant HOLD_DEL : time := 0 ns;
constant SEVERITY_SEL : SEVERITY_LEVEL := Warning) is
begin
assert not ((CLK'DEL(HOLD_DEL)'Event and CLK'DEL(HOLD_DEL)='1')
and (not SIG'Stable(HOLD_DEL)))
     report "保持时间不足"
     severity SEVERITY_SEL;
```

end Hold_check;

需要指出的是，现在的综合工具一般也能够进行时序检查，并且可以按照用户输入的建立、保持时间约束来综合电路。那么，在代码中加入时序检查代码，是否还有必要呢？实际上，时序检查代码不仅能够对时序进行检查，还可以帮助设计师编写更加准确而有效的测试矢量，并且通过 assert 语句的报告，设计师还可以在仿真时获得电路的内部信息，这对于 VHDL 这种并行语言程序的调试是十分重要的。因此，通常建议在程序中加入时序检查代码。

4.1.6 构造硬件的系统级 VHDL 模型的不同方式

VHDL 语言提供了多种硬件造型方式，而硬件的 VHDL 模型可能用于不同领域，比如，可以利用 VHDL 模型建立硬件的设计文档、硬件的仿真验证、综合、测试生成等。这就提出了一个问题：按某种方式建立的硬件模型是否可以用于所有这些领域？一般地讲，答案是否定的，即某些应用领域只能采用特定的造型方式。在 VLSI 设计中，书写 VHDL 代码的主要目标之一就是进行综合，那么怎样才能达到优化的综合结果呢？下面通过几个例子说明这些问题。

首先来看一个实现乘 2 电路的例子，下面给出了三种不同风格的 VHDL 模型。第一种代码（代码(a)）是直截了当的，2 被赋给信号 TWO，在时钟上升沿锁存乘法结果。在硬件设计的初始阶段，不考虑如何将行为模型转换为硬件，可以用这种模型进行仿真，以验证系统的功能。如果对这个结构体进行综合，可以得到图 4.11(a)所示的硬件。显然，综合

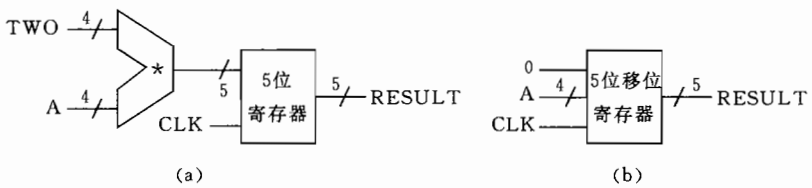


图 4.11　乘 2 电路的综合结果

工具把 TWO 和 A 映射为 4 位信号线，把 RESULT 映射为 5 位信号线，用乘法器计算乘法，用寄存器锁存 RESULT。但是，乘 2 电路通过移位就能够完成，并不需要成本很高的乘法器电路。为此，可以把下面代码(a)所示的代码改写为(b)所示的第二种形式。这样一来，综合工具将用移位器实现乘法操作，硬件代价明显降低，综合结果如图 4.11(b)。但是，这样做的缺点在于无法使用形如 Integer 的抽象的数据类型，而只能使用位矢量，且乘法运算以隐含方式表示，因此在改善综合质量的同时，降低了程序的可读性。实际上，现在先进的综合工具如 Synopsys 和 Cadence 所采用的综合算法已经能够很好地处理与常数相乘的运算，因此，可以写出下面代码(c)标出的第三种风格的代码。这段代码把 TWO 定义为常数，从而引导综合进行常数乘法优化，所以综合结果与(b)的代码是相同的。必须指出的是，(c)是优化的代码书写风格，但并不是所有的综合工具都能够进行必要的综合优化，因此设计师还要根据具体综合工具确定合适的代码风格。

```
-- (a) 代码形式之一
signal TWO, A : Integer range 0 to 15;
signal RESULT : Integer range 0 to 32;

TWO <= 2;
process_A: process
begin
    wait until CLK='1' and CLK'event;
    RESULT <= A * TWO;
end process process_A;

-- (b) 代码形式之二
signal A : Std_logic_vector(3 downto 0);
signal RESULT : Std_logic_vector(4 downto 0);

process_B: process
begin
    wait until CLK='1' and CLK'event;
    RESULT <= A &'0';
end process process_B;

-- (c) 代码形式之三
constant TWO : Integer range 0 to 15;
signal A : Integer range 0 to 15;
signal RESULT: Integer range 0 to 32;
process_C: process
begin
    wait until CLK='1' and CLK'event;
    RESULT <= A * TWO;
end process process_B;
```

下面再来看一个延时器的例子。在有些数字电路中,系统需要等待一个固定延时之后,再开始处理下一事件,延时通过延时电路完成。该电路在 15 个输入时钟周期之后,输出 TIME_OUT 为'1'。用 4 位寄存器存储计数值,复位时(RESET 为'1')载入初值。下面的一段 VHDL 代码(a)和(b)是两种不同的 VHDL 描述,图 4.12(a)和(b)分别是其综合后的电路。由(a)标出的代码使用传统的十六进制计数器来实现,复位时初值为"0000",然后在时钟沿上计数,计满为"1111"后输出 TIME_OUT 有效。这个描述对于仿真是没有问题的,可读性也较好。但是存在着一些缺点,首先是由于电路由 4 位锁存器和加 1 电路构

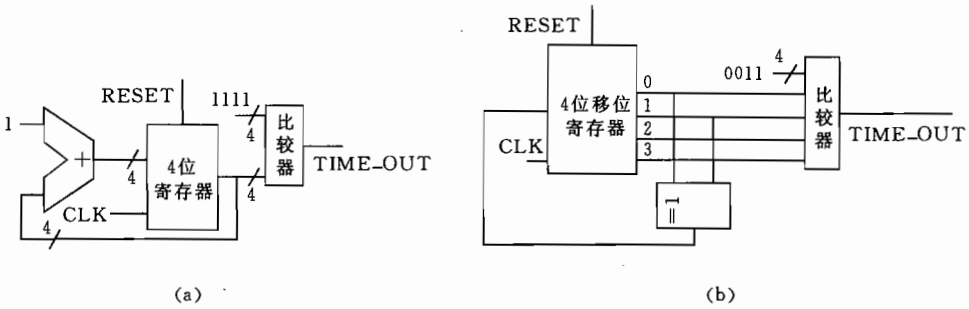


图 4.12　用不同算法综合出的延时器电路

成，所以芯片面积比较大，在综合工具优化能力较差的情况下，甚至会用加法器来实现加1过程！其次，这种计数器电路的固有延时相对较大（因为进位产生电路的扇入大），只能用于低速工作的场合。由(b)标出的是用线性反馈移位寄存器(LFSR，linear feedback shift register)实现的延时器，根据线性反馈移位寄存器理论，n 位 LFSR 计数器可以产生最大长度 2^n-1 的计数序列，也就是说，n 位 LFSR 将遍历 2^n-1 个状态，然后返回初值。在电路结构上，LFSR 要简单得多，由 4 位的锁存器和一个异或门反馈回路组成。用这种方式实现延时，虽然牺牲了顺序的计数序列（即计数序列不再是从 1 到 15 的自然顺序），但是电路规模小，而且运算速度要比十六进制计数器快得多。这个例子说明仿真上等效的 VHDL 模型的综合结果可能会很不相同，因此设计师应寻找有效的算法。

```
-- (a) 用计数器实现延时器
Delay_Counter: process(RESET, CLK)
    variable COUNT : Std_logic_vector(3 downto 0);
    function Inc(X : in Std_logic_vector ) return Std_logic_vector is
        variable XV: Std_logic_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;
begin
    if RESET='1' then
```

```
        COUNT := "0000";
    elsif CLK='1' and CLK'Event then
        COUNT := INC(COUNT);
    end if;
    TIME_OUT <= (COUNT = "1111");
end process Delay_Counter;

-- (b) 用线性反馈移位寄存器实现延时
Delay_LFSR: process
    variable COUNT : Std_logic_vector(3 downto 0);
begin
    if RESET='1' then
        COUNT := "0001";
    elsif CLK='1' and CLK'Event then
        COUNT := ( COUNT(1) xor COUNT(0) ) & COUNT(3 downto 1);
    end if;
    TIME_OUT <= (COUNT = "0011");
end process Delay_LFSR;
```

下面的实体 Count_ONE 用来确定输入信号中'1'的个数，实体是电路的系统级行为模型。电路的输入为 3 位信号 A，输出信号为 2 位信号 C，其数值为输入信号中'1'的个数。

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

archtecture ALG of Count_ONE is
begin
    A: 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 A;
end ALG;
```

在这个结构体中，先用 for 循环扫描输入信号 A 的各位，如果发现 A 中有一个'1'，则将 NUM 的数值增加 1。扫描结束后，再用 case 语句将十进制数据转换为二进制输出。如果不考虑如何将行为模型转换为硬件，可以用这种模型进行仿真，以验证系统的功能。如果把这个结构体综合为硬件，由于这个 for 循环内部有一个不完全的条件判断(即没有 else 分支)，加之 for 循环的迭代特征，所以某些综合工具会用时序逻辑电路实现硬件。如果使用 Synopsys 的综合工具 DC，则虽然能够展开循环以组合逻辑实现电路，但是电路中会形成很长的比较链，也就是说，关键路径上延时很大，所以这个描述作为综合输入是不可取的。

事实上，如果写出电路 Count_ONE 的输入输出关系的真值表，就会发现可以用组合逻辑实现该电路。如果人工设计电路，很容易发现这个问题。但如果是编制自动综合代码，这样的 for 循环最直观的实现形式是时序电路。采用这种综合工具进行电路自动设计，有可能会得出很不经济的硬件实现方案，也有可能需要很长的计算时间。如果把结构体 ALG 进行改进，用 case 语句直接检查输入信号的码型，并产生相应的输出信号，则很容易将该行为模型转换为如图 4.13 所示的硬件。这是一个数据宽度为 2 的 8 选 1 电路，集成电路的自动综合工具能够以很直观的形式把这种 case 结构转换为多路选择器。

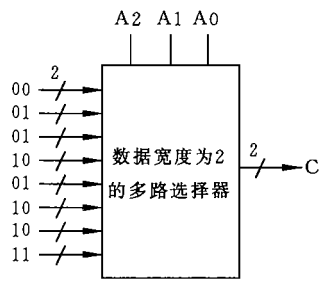


图 4.13　结构体 Modified_ALG 的硬件实现

```
architecture Modified_ALG of Count_ONE is
begin
    B: process(A)
    begin
        case A is
            when "000" => C <= "00";
            when "001" | "010" | "100" => C <= "01";
            when "011" | "110" | "101" => C <= "10";
            when "111" => C <= "11";
        end case;
    end process B;
```

end Modified_ALG;

一般说来，如果期望将硬件的系统级模型用于综合，建议采用下述准则书写 VHDL 模型：

(1) 将硬件的行为指标以合理的方式映射为一些进程；

(2) 对于每个进程完成的操作，尽量选择最有效的算法；

(3) 了解综合器的性能，特别是了解综合工具支持的 VHDL 可综合子集，并以合理的代码风格引导综合工具生成硬件；

(4) 在条件允许的情况下，尽量用变量代替信号，对于固定值的信号要用常量代替；

(5) 尽量共享复杂运算，可以共享的数据处理用函数或过程定义；

(6) 明确指出电路的无关态(don't care condition)，引导综合器进行优化；

(7) 使用能够满足需要的最小数据宽度；

(8) 用组合逻辑实现的电路和用时序逻辑实现的电路要分配到不同的进程中 。

现在再给出一个例子，其 VHDL 源代码按本章建议的风格写出，可以用于自动综合。实体 ALU 给出的是该电路的 VHDL 行为描述，根据输入信号 CON 的不同状态，进程 functions 可能完成 4 种不同的功能：F = A，F = not A，F = A+B，F = A and B。在时钟 CLK 的上升沿，进程 Store 将 F 的内容复制到 FOUT。图 4.14 是该电路的方框图，这个结构可以通过 VHDL 描述综合得出。VHDL 源代码中的 case 语句映射为多路选择器，VHDL 源代码中的数据运算“非”、“加”、“与”等被映射为组合逻辑电路。进程 Store 被映射为触发器，信号 F 映射为多路选择器的输出端与寄存器的输入端之间的连线。自动综合工具很容易就可以识别出 case 语句，并把它对应于一个多路选择器。因为进程 Store 由时钟 CLK 触发，所以按照寄存器推断原则，被映射为触发器。由于采用了适合综合的 VHDL 源代码设计风格，将算法描述转换为硬件变得相对简单。

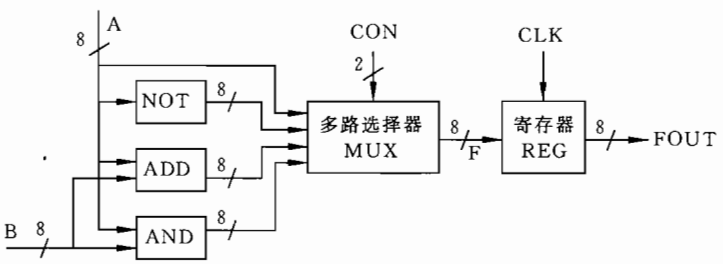


图 4.14 实体 ALU 的硬件实现

```
entity ALU is
    port(CON: in Std_logic_vector(1 downto 0);
         A, B: in Std_logic_vector(7 downto 0);
         CLK: in Std_logic;
         FOUT: out Std_logic_vector(7 downto 0));
end ALU;
```

```
architecture ALG of ALU is
    procedure ADD(A, B: in Std_logic_vector; SUM: out Std_logic_vector) is
    variable SUMV, AV, BV: Std_logic_vector(A'Length - 1 downto 0);
    variable CARRY: Std_logic;
    begin
        AV := A;BV := B;
        CARRY := '0';
        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;
        SUM := SUMV;
    end ADD;

    signal F : Std_logic_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" => ADD(A, B, F);
            when "11" => F <= A and B;
        end case;
    end process functions;

    Store: process(CLK)
    begin
        if CLK = '1' and CLK'Event then
           FOUT <= F;
        end if;
    end process STORE;
end ALG;
```

4.2 系统间互连的表示

在 VHDL 中，系统中元件的互连通过实体和结构体描述，其中结构体是电路的结构

描述。对于下面图 4.15 所示的简单系统 X，系统中有两个元件 A 和 B，两个元件的外部接口分别由实体 A 和 B 描述。实体 A 和 B 的外部接口描述如下：

```
entity A is
    port(P: in Std_logic, Q: out Std_logic);
end A;

entity B is
    port(R: in Std_logic, S: out Std_logic);
end B;
```

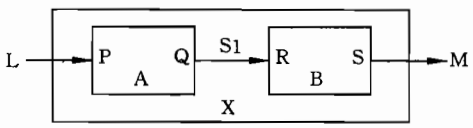


图 4.15　两个元件互连组成的简单系统

系统 X 则可以由下面 VHDL 代码中的实体和结构体描述：

```
-- 系统 X 的结构式描述
entity X is
    port(L: in Std_logic; M: out Std_logic);
end X;

architecture Structure of X is
    signal S1: Std_logic;
    component A
        port(P: in Std_logic; Q: out Std_logic);
    end component;
    component B
        port(R: in Std_logic; S: out Std_logic);
    end component;
begin
    A1: A port map(L, S1);
    B1: B port map(S1, M);
end Structure;
```

系统内元件的互连由端口映射(port map)和元件实例化语句实现，端口映射定义了连接到各个元件的每个端口的信号，在图 4.15 这个简单例子中，信号 S1 定义为元件 A 和 B 之间的连线。在上述 VHDL 代码的例子中，当元件实例化时，在端口映射中指定互连关系采用的是位置关联。除了位置关联之外，还可以采用名称关联来指定互连关系。比如，

下面两个元件实例化语句与前面 X 的结构体 Structure 中的两个语句完全等效。采用名称关联时，端口映射语句中信号名出现的先后顺序与连接关系无关。

A1：A port map (Q => S1; P => L);

B1：B port map (S => M; R => S1);

为了给读者一个完整的概念，下面讨论一个多模块系统的实例。这是一个 8 位全双工异步接收发送器(UART)电路，其主要功能如下：① 从计算机接收 8 位并行数据并写到串行输出；② 从串口读入外部数据并将其转换为 8 位并行数据送往计算机。串入并出操作由串行输入的下降沿触发，串行输入要保持低电平半个周期以上。这个半周期时标也用作输入移位时钟，8 位数据输入结束后，结束信号 NINTI 变为 0 并维持到下一次数据输入。并入串出操作由输入信号 LOAD 的高电平触发，串行输出结束后，结束信号 NINTO 变为 0。8 位数据口是双向的。图 4.16 是 UART 的硬件框图，该系统由 4 个主要模块构成，其中 PAR_IN_SER_OUT 执行并入串出操作，SER_IN_PAR_OUT 执行串入并出操作，INTERFACE 是并行数据与外部的接口，CLOCK_GEN 产生工作时钟。前 3 个模块是可综合的，时钟电路需要另行生成。

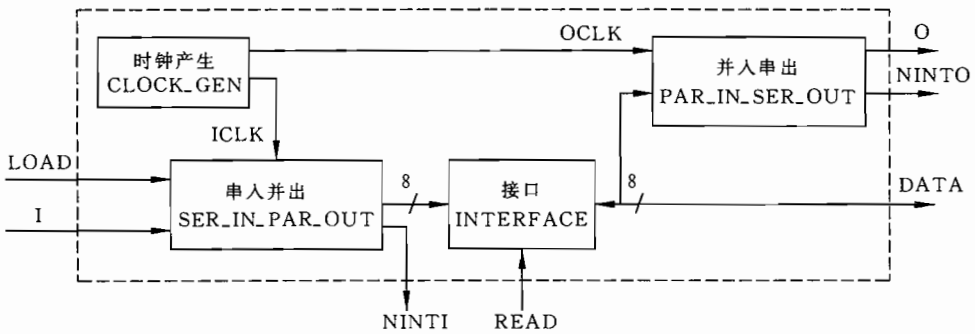


图 4.16 UART 的硬件框图

按照由顶向下的办法，首先作出系统的顶层描述，由下面的 VHDL 代码所示。这段代码通过例化元件的方式，表示了元件的互连关系；这段代码也表现了系统的划分，但此时仅定义了元件接口，元件的具体实现并未确定。

```
use IEEE. Std_1164. all;
package UART_package is
    constant WORD_LENGTH : integer :=8;
    type WORD is Std_logic_vector(WORD_LENGTH-1 downto 0);
    component CLOCK_GEN
        generic(ICLK_DEL, OCLK_DEL : Time :=0 ns);
        port(ICLK, OCLK : buffer Std_logic);
    end component;
    component Interface_Std_logic
```

```
        port(A : in Std_logic;
             E : in Std_logic;
             Z : out Std_logic);
    end component;
    component INTERFACE
        port(IN_WORD : in WORD;
             ENABLE : in Std_logic;
             OUT_WORD : out WORD);
    end component;
    component PAR_IN_SER_OUT
        port(DATA : in WORD;
             LOAD : in Std_logic;
             CLOCK : in Std_logic;
             NINTO : out Std_logic;
             O : out Std_logic);
    end component;
    component SER_IN_PAR_OUT
        port(I : in Std_logic;
             CLOCK : in Std_logic;
             NINTI : buffer Std_logic;
             OUTPUT : out Word);
    end component;
end UART_package;

    use work.UART_package.all;
entity UART is
    generic(CLK_PER, ODEL, INDEL, INTDEL : Time := 0 ns);
    port(DATA : inout WORD;
         I : in Std_logic;
         LOAD, READ : in Std_logic;
         O : out Std_logic;
         NINTO : out Std_logic;
         NINTI : buffer Std_logic);
end UART;
architecture Structure of UART is
    signal ICLK, OCLK : Std_logic;
    signal Output : WORD;
begin
```

```
    U1: CLOCK_GEN port map(ICLK, OCLK);
    U2: PAR_IN_SER_OUT port map(DATA, LOAD, OCLK,NINTO, O);
    U3: SER_IN_PAR_OUT port map(I, ICLK, NINTI, Output);
    U4: INTERFACE port map(OUTPUT, READ, DATA);
end Structure;
```

UART 电路的一个问题是处理 8 位并行数据双向端口 DATA，即并行输入和输出都要经过这个端口。因此，可以使用元件 INTERFACE 隔离串入并出电路的输出与端口 DATA 。下面的 VHDL 代码给出的是 INTERFACE 及其“位”单元的 VHDL 模型。在其位单元中，当使能端为'0'时，输出高阻态，这时可以从外部向 DATA 端写入数据。通过 IEEE 的 Std_logic 的决断函数的判断，PAR_IN_SER_OUT 执行并串转换。当使能端为'1'时，执行串并转换时，SER_IN_PAR_OUT 的输出写到 DATA 端。在综合时，综合器将根据三态推断的原则，用三态缓冲器实现这个电路。实体 INTERFACE 使用 generate 语句形成 8 位的缓冲器电路。

```
entity INTERFACE_Std_logic is
        port(A : in Std_logic;
             E : in Std_logic;
             Z : out Std_logic);
end INTERFACE_Std_logic;

architecture ALG of INTERFACE_Std_logic is
begin
    Tri-state-latch: process(A, E)
    begin
        if E='0' then
            Z <= 'Z';
        else
            Z <= A;
        end if;
    end process;
end ALG;

use work. UART_package. all;
    entity INTERFACE is
    port(IN_WORD : in WORD;
         ENABLE : in Std_logic;
         OUT_WORD : out WORD);
    end INTERFACE;
```

```
architecture Structure of is
begin
    GEN_INTERFACE: for I in WORD_LENGTH-1 downto 0 generate
        U1: INTERFACE_Std_logic port map(IN_WORD(I), ENABLE,
        OUT_WORD(I));
    end generate;
end Structure;
```

下面给出的是时钟生成电路的 VHDL 代码。这段代码产生两个时钟信号,OCLK 和 ICLK,分别用于 PAR_IN_SER_OUT 和 SER_IN_PAR_OUT 模块。在串入并出电路中,工作时钟需要高于移位时钟,才能检测是否开始一次输入过程,因此 ICLK 的频率是 OCLK 频率的 4 倍。这个模块的代码只能用于仿真,不能被综合。需要注意的是,按照 IEEE 的 VHDL 标准,Std_logic 类型的信号在仿真初始化时初值为'0',下面的代码能够形成振荡的效果。

```
entity CLOCK_GEN is
    generic(ICLK_DEL, OCLK_GEN : Time := 0 ns);
    port(ICLK, OCLK : buffer Std_logic);
end CLOCK_GEN;
architecture Sim of CLOCK_GEN is
    signal TMP_ICLK, TMP_OCLK : Std_logic;
begin
    Update: process(TMP_ICLK, TMP_OCLK)
    begin
        TMP_ICLK <= not TMP_ICLK after ICLK_DEL/4;
        TMP_OCLK <= not TMP_OCLK after OCLK_DEL;
    end process;
    ICLK <= TMP_ICLK;
    OCLK <= TMP_OCLK;
end Sim;
```

下面给出的是并入串出模块的 VHDL 代码,实质上描述了一个 9 位移位寄存器。LOAD 信号有效('1')时,移位寄存器低 8 位载入 8 位并行数据,并在最高位写入'1'。然后在时钟沿上,寄存器依次右移,并在左端移入'0',当移位开始时位于最高位的'1'被移到最低位时,即移位寄存器高 8 位全为'0'时,移位过程结束,输出结束信号 NINTO='0'。这段代码由两个进程组成,进程 Seq_Par_in 在 LOAD 为'1'有效时装载寄存器;否则,在时钟沿上更新状态,即载入移位数据。进程 Com_Par_in 在每次移位完成后,检查是否已经结束,并产生各路输出。

```
use work. UART_package. all;
```

```
use IEEE. std_logic_1164. all;
entity PAR_in_SER_OUT is
    port(DATA : in WORD;
         LOAD : in Std_logic;
         CLOCK : in Std_logic;
         NINTO : out Std_logic;
         O : out Std_logic);
end PAR_IN_SER_OUT;
architecture ALG of PAR_IN_SER_OUT is
    signal OREG, NEXT_OREG : Std_logic_vector(WORD_LENGTH downto
    0);
begin
    SEQ_PAR_in: process(LOAD, CLOCK, DATA, NEXT_OREG)
    begin
        if LOAD='1' then
            OREG <= '1' & DATA;
        elsif CLOCK='1' and CLOCK'Event then
            OREG <= NEXT_OREG;
        end if;
    end Porcess;
    Com_Par_in: process(OREG)
        variable GO : Std_logic;
    begin
        GO := '0';
        for I in WORD_LENGTH downto 0 loop
            GO := GO or OREG(I);
        end loop;
        NINTO <= GO;
        NEXT_OREG <= '0' & OREG(WORD_LENGTH downto 0);
        O <= OREG(0);
    end process;
end ALG;
```

UART 电路的最后一个模块是串入并出模块，下面给出其 VHDL 代码。该模块也采用了一个类似移位寄存器的结构，输入时钟是采样率的 4 倍，该时钟被 4 分频后用作移位时钟。该模块内嵌有一个有限状态机控制器。控制器在串行输入转为低电平之后继续进行检查，如果串行输入保持两个'0'状态则确认开始串并转换。结束信号也由有限状态机产生，这时置复位信号 RESET 为'1'，使移位寄存器载入初值。

```
use work.UART_package.all;
use IEEE.std_logic_1164.all;
entity SER_IN_PAR_OUT is
    port(I : in Std_logic;
         CLOCK : in Std_logic;
         NINTI : buffer Std_logic;
     OUTPUT : out Word);
end SER_IN_PAR_OUT;
architecture ALG of SER_IN_PAR_OUT is
    type I_STATE is (RESET_STATE, I_LOW_1, I_LOW_2, I_LOW_3,
                                RUNNING);
    signal STATE, NEXT_STATE : I_STATE;
    signal IREG : Std_logic_vector(WORD_LENGTH downto 0);
    signal DIV_CLOCK, RESET, RUNNING_STATE : Std_logic;
begin
    Divide_Clock: process
        variable STATE : integer range 0 to 3;
    begin
        wait until CLOCK'Event and CLOCK='1';
        if STATE < 3 then
            STATE := STATE + 1;
            DIV_CLOCK <= '0';
        else
            STATE := 0;
            DIV_CLOCK <= RUNNING_STATE;
        end if;
    end process;

    Shift_IN: process(RESET, DIV_CLOCK)
    begin
        if RESET='1' then
            IREG(WORD_LENGTH downto 1) <= "00000000";
            IREG(0) <= '1';
        elsif DIV_CLOCK'Event and DIV_CLOCK='1' then
            IREG <= IREG(WORD_LENGTH-1 downto 0) & I;
        end if;
    end process;
```

```
OUTPUT <= IREG(WORD_LENGTH-1 downto 0);

Save_State: process
begin
    wait until CLOCK'Event and CLOCK='1';
    STATE <= NEXT_STATE;
end process;

Find_State: process(I, STATE, IREG)
begin
    case STATE is
        when RESET_STATE =>
            if I='0' then NEXT_STATE <= I_LOW_1;
            else NEXT_STATE <= RESET_STATE;
            end if;
            NINTI <= '1';
        when I_LOW_1=>
            if I='0' then NEXT_STATE <= I_LOW_2;
            else NEXT_STATE <= RESET_STATE;
            end if;
            NINTI <= '1';
        when I_LOW_2=>
            if I='0' then NEXT_STATE <= I_LOW_3;
            else NEXT_STATE <= RESET_STATE;
            end if;
            NINTI <= '1';
        when I_LOW_3=>
            if I='0' then NEXT_STATE <= RUNNING;
            else NEXT_STATE <= RESET_STATE;
            end if;
            NINTI <= '1';
        when RUNNING =>
            if IREG(WORD_LENGTH)='1' then
                NEXT_STATE <= RUNNING;
                NINTI <= '1';
            else
                NEXT_STATE <= RESET_STATE;
                NINTI <= '1';
```

```
                end if;
            end case;
            if STATE=RUNNING then
                RESET <= '0';
                RUNNING_STATE <= '1';
            else
                RESET <= '1';
                RUNNING_STATE <='0';
            end if;
        end process;
end ALG;
```

4.3 系统的算法模型

本节以一个具体设计为例，讨论系统级设计常会遇到的一些问题，如系统设计的划分、行为级描述向寄存器传输级代码的转换、多值逻辑、复用和模块之间的通信协议等。

4.3.1 简单四模块系统

下例可以说明如何用多值逻辑系统构造硬件系统的模型。图 4.17 示意地给出了由三个模块组成的系统，三个模块分别是带缓冲的寄存器(BUFF_REG)、随机存取存储器(RAM)、加法-存储模块(ADD_STORE)。该硬件将完成如下功能：

(1) 当闸门信号(STRB)变高时，数据通过端口 DI 送入带缓冲的寄存器单元中，并将数据有效标志 DAV 置位为'1'；

(2) 当 DAV 变为'1'后，模块 ADD_STORE 作出响应：首先设 EN='1'，该信号使得带缓冲寄存器中的数据送上总线 DATA_BUS，并将信号 DAV 复位为'0'；

(3) ADD_STORE 从 DATA_BUS 上得到数据并把数据存储在其内部寄存器中；

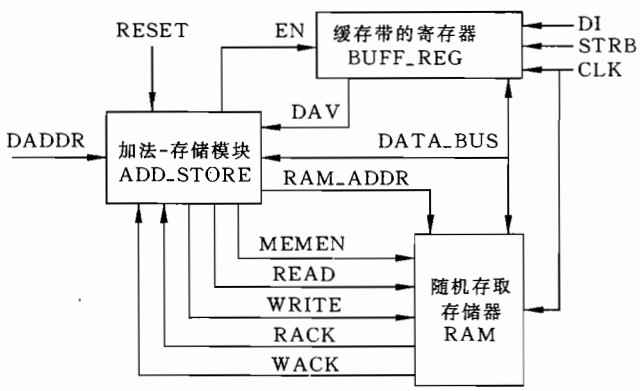


图 4.17 三个模块组成的硬件系统

(4) ADD_STORE 按地址线 RAM_ADDR 中指明的地址,从 RAM 中读取数据;RAM_ADDR 通过端口 DADDR 预先设定数值。读操作结束后,信号 RACK 给出标志;

(5) ADD_STORE 通过数据总线 DATA_BUS 得到 RAM 中的数据,把它与内部寄存器中原先存储的数据相加后,再放回到内部寄存器;

(6) ADD_STORE 开始 RAM 写操作,把内部寄存器中的数据写回到 RAM。RAM 地址由 RAM_ADDR 指定,WACK='1'是写操作结束的标志;

(7) ADDR_STORE 返回到初始状态,等待从带缓冲的寄存器来的下一次数据有效信号(DAV);

(8) 同步时钟为 CLK。

下面的 VHDL 代码是带缓冲的寄存器的算法模型,模型本身很容易读懂,在时钟上升沿上如果 STRB 有效,则锁存数据;EN 有效时寄存器输出 REG 的值,否则输出高阻态。

```
-- 带缓冲的寄存器的 VHDL 模型
use IEEE. std_logic_1164. all;
entity BUFF_REG is
    generic(STR_DEL, DAV_DEL, ODEL: TIME);
    port(DI: in Std_logic_vector(7 downto 0);
         CLK : in Std_logic;
         STRB, EN: in Std_logic;
         DAV: out Std_logic;
         DO: out Std_logic_vector(7 downto 0) := "ZZZZZZZZ") );
end BUFF_REG;

architecture TWO_PROC of BUFF_REG is
    signal REG : Std_logic_vector(7 downto 0);
begin
    Front_end: process(CLK, STRB, EN)
    begin
        if EN='1' then
            DAV <= '0'after DAV_DEL;
        elsif CLK='1' and CLK'Event and STRB='1';
            REG <= DI after STRB_DEL;
            DAV <= '1' after DAV_DEL;
        end if;
    end process Front_end;

    Output: process(REG, EN)
```

```
    begin
        if EN = '1' then
            DO <= REG after ODEL;
        else
            DO <= (others=> 'Z');
        end if;
    end process Output;
end TWO_PROC;
```

下面给出了随机存取存储器 RAM 的算法模型，这个模型也很直观，容易理解。它对信号 CS，RD 或 WRITE 的变化作出响应，可以对它进行读操作或写操作。读写操作结束的标志位 RACK 和 WACK 的持续时间为 ACK_PW。

```
-- 随机存取存储器 RAM 的 VHDL 模型
use IEEE. std_logic_1164. all;
entity RAM is
    generic(RDEL, DISDEL, ACK_DEL, ACK_PW: TIME);
    port(DATA: inout Std_logic_vector(7 downto 0) :=(others=>'Z');
         ADDR: in Std_logic_vector(4 downto 0);
         CLK : in Std_logic;
         RD, WR, CS: in Std_logic;
         RACK, WACK: out Std_logic);
end RAM;

architecture ALG of RAM is
    function INTVAL(VAL : in Std_logic_vector ) return Integer is
        variable VALV: Std_logic_vector(VAL'Length - 1 downto 0);
        variable SUM : Integer := 0;
    begin
        VALV := VAL;
        for I in VALV'Low to VALV'High loop
            if VALV(I) = '1' then
                SUM := SUM + ( 2 * * I);
            end if;
        end loop;
        return SUM;
    end INTVAL;
begin
    MEM: process
```

```
        type MEMORY is array(0 to 31) of Std_logic_vector(7 down to 0);
        variable MEM: MEMORY := (other => (other => '0'));
    begin
        wait until CLK='1' and CLK'Event;
            if CS ='1' then
                if RD = '1' then
                    DATA <= MEM(Intval(ADDR)) after RDEL;
                    RACK <='1' after ACK_DEL,
                              '0' after ACK_DEL + ACK_PW;
                elsif WR = '1' then
                    MEM(Intval(ADDR)) := DATA;
                    WACK <='1' after ACK_DEL,
                              '0' after ACK_DEL + ACK_PW;
                end if;
            else
                DATA <= (other=>'Z') after DISDEL;
            end if;
    end process MEM;
end ALG;
```

ADD_STORE 模块控制系统完成各种动作。下面给出了 ADD_STORE 的算法模型。模型中给出了一系列控制操作，每步控制操作分别用注释 CS0～CS6 标出，每步控制操作都包括了一些动作且以 wait 语句结束。

```
use IEEE. std_logic_1164. all;
entity ADD_STORE is
    generic(CON_DEL, DO_DEL, MA_DEL, DIS_DEL: TIME);
            port(RESET, DAV, RACK, WACK: in Std_logic;
            EN, MEMEN, READ, WRITE: out Std_logic;
            DATA: inout Std_logic_vector(7 downto 0) := (others=>'Z');
            DADDR: in Std_logic_vector(4 downto 0);
            MADDR: out Std_logic_vector(4 downto 0) );
end ADD_STORE;

architecture ALG of ADD_STORE is
begin
    Control: process
        variable DATA_REG: Std_logic_vector(7 downto 0) ;
    begin
```

```
Reset_loop: loop
    DATA<= (others=>'Z') after DIS_DEL;          -- CS0
    wait until CLK'Event and CLK ='1';
    ..........
    if RESET = '1' then
        exit Reset_loop;
    end if;
    Run_loop: loop
        wait until CLK'Event and CLK ='1';
        ..........
        if RESET = '1' then
            exit Reset_loop;
        end if;
        wait until CLK'Event and CLK='1' and DAV = '1';
        ..........
        EN <= '1' after CON_DEL;                 -- CS1
        ..........
        wait until CLK'Event and CLK='1';
        EN <= '0' after CON_DEL;                 -- CS2
        DATA_REG := DATA;
        wait until CLK'Event and CLK='1';
        ..........
        MADDR <= DADDR after MA_DEL;
        MEMEN <= '1' after CON_DEL;              -- CS3
        READ <= '1' after CON_DEL;
        wait until CLK'Event and CLK='1' and RACK = '1';
        ..........
        DATA_REG := DATA + DATA_REG;
        READ <= '0' after CON_DEL;
        MEMEN <= '0' after CON_DEL;              -- CS4
        wait until CLK'Event and CLK='1';
        ..........
        DATA<= DATA_REG after DO_DEL;
        WRITE <= '1' after CON_DEL;
        MEMEN <= '1' after CON_DEL;              -- CS5
        wait until CLK'Event and CLK='1' and WACK='1';
        ..........
        WRITE <= '0' after CON_DEL;
```

```
                MEMEN <= '0' after CON_DEL;                    -- CS6
                DATA<= (others=>'Z') after DIS_DEL;
                wait until CLK'Event and CLK='1';
                ..........
            end loop Run_loop;
        end loop Reset_loop;
    end process Control;
end ALG;
```

在 VHDL 中，wait 语句有如下几种形式：

```
wait until condition;
wait for time;
wait on signal [until condition];
```

wait 语句使进程挂起，直到满足给定条件后再恢复执行。上述几种形式的 wait 语句中，第一种形式规定在条件 condition 为 true 时恢复执行，常用于检测时钟沿。由于本例是同步电路，因此 wait until CLK'Event and CLK= '1'实际隐含表示每来一个时钟，对应一个控制器的内部状态。在每一状态上，控制器发出相应的控制信号。上面的实体 ADD_STORE 的行为描述 ALG 中使用了这一语句，在时钟沿上检测输入控制信号 DAV，RACK 和 WACK，这也体现了同步操作的概念。这种 wait 语句的好处是可以综合，一般说来，现在的 EDA 综合工具把每个进程中 wait until 后面以 SIGNAL'Event and SIGNAL='1'(或'0')形式出现的信号当作时钟处理；同时现有综合工具只能处理一个进程内只有单一时钟的情况，而且 ASIC 设计方法学也提倡单时钟、同步操作的工作方式，所以书写可综合代码时，必须保证进程内只有一个信号以上述形式出现在 wait 语句中。

wait 语句的第二种形式用来描述硬件中的固定延时，如果时钟信号 CLK 的周期为 CLK_PER，那么从仿真的角度看，wait for CLK_PER 与 wait until CLK'Event and CLK='1'的效果是完全相同的，但是前一种形式的 wait 语句不能综合。

wait 语句的第三种形式在后面不带有条件的情况下，在功能上相当于进程敏感信号列表，但可以出现在进程中的任意部分，这种形式常用于描述组合逻辑电路进程。如果后面带有条件，则可以描述更复杂的电路，如异步电路，但不一定能够被综合。

对于这个模型，另一个值得注意的问题是它们的输出。三个模块的输出通过信号 DATA_BUS 连接在一起，DATA_BUS 被初始化为"ZZZZZZZZ"，由于 Std_logic 型信号的初始默认值为'U'，所以除非特别指明，被 DATA_BUS 驱动的所有信号的初始缺省值全为'U'。为了使总线决断函数正常工作，所有不活动的信号驱动器的值应都为'Z'。此外，当某个信号的输出与总线信号的连接断开之后，它也应返回'Z'态。这三个模块的模型全按这种方式设计。

在模块 ADD_STORE 的算法模型中，使用了控制操作序列描述硬件的行为，每步控制操作中，都使用了 wait 语句。我们不但要使用这样的语法结构描述硬件的行为，而且要了解这样的语法结构对应着什么样的硬件实现。这就提出了一个问题：这样的控制操作序

列的硬件背景是什么？为此，下面要把模块 ADD_STORE 进一步划分为控制器部分和数据路径部分。图 4.18 示意给出了这种划分，其中控制器部分由控制单元 CU 和时钟生成单元 CLK_GEN 组成，数据路径部分则是简单的加法单元 ADU。在设计划分确定之后，就可以书写寄存器传输级代码，然后进行综合。一般说来，在寄存器传输级代码中，控制器部分以有限状态机的形式书写，而数据路径则由运算电路和存储电路组成。寄存器传输级的综合技术已经非常成熟，除非在一些极端情况下，对这个层次的代码进行综合完全可以达到理想的效果。

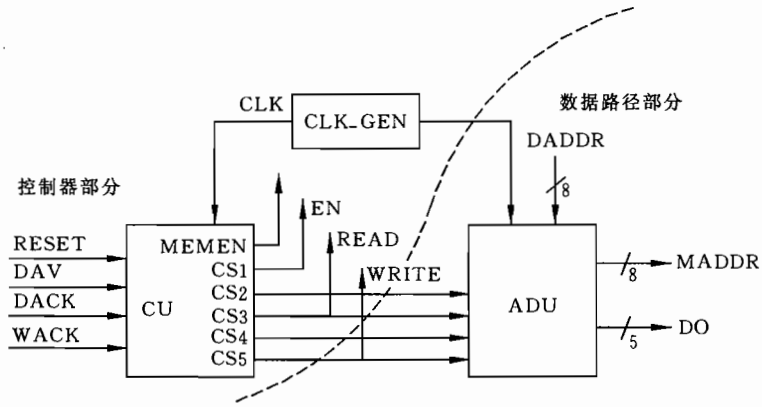


图 4.18　模块 ADD_STORE 的进一步划分

下面的 VHDL 源代码是时钟生成单元 CLK_GEN 的行为模型，该模型仅可用于仿真。

```
use IEEE.std_logic_1164.all;
entity CLK_GEN is
generic(PER: TIME);
port(CLK: out Std_logic);
end CLK_GEN;
architecture Sim of CLK_GEN is
    signal TMP_CLK : Std_logic;
begin
    Update: process(TMP_CLK)
    begin
        TMP_CLK <= not TMP_CLK after PER;
    end process;
    CLK <= TMP_CLK;
end Sim;
```

下面的实体 ADU 和相应的结构体 Behavior 给出的 VHDL 源代码是加法单元 ADU 的算法模型。加法单元 ADU 在 CLK 的上升沿上检查控制信号 CS2，CS3，CS4 和 CS5，在

这 4 个信号有效时该单元分别完成下列操作：① 加载数据寄存器；② 执行加法运算；③ 输出存储器地址；④ 把数据值送上数据总线。

```
use IEEE.std_logic_1164.all;
entity ADU is
generic(DO_DEL, MA_DEL: TIME);
    port(CLK : in Std_logic;
         CS2, CS3, CS4, CS5: in Std_logic;
         DATA: inout Std_logic_vector(7 down to 0) := (others=>'Z');
         DADDR: in Std_logic_vector(4 downto 0);
         MADDR: out Std_logic_vector(4 downto 0) );
end ADU;

architecture Behavior of ADU is
begin
    DU: process(CS2, CS3, CS4, CS5);
        variable DATA_REG: Std_logic_vector(7 downto 0);
    begin
        wait until CLK='1' and CLK'Event;
        if CS2 = '1' then
            DATA_REG := DATA;
        elsif CS4 = '1' then
            DATA_REG := DATA + DATA_REG;
        elsif CS3 = '1' then
            MADDR <= DADDR after MA_DEL;
        elsif CS5 = '1' then
            DATA<= DATA_REG after DO_DEL;
        else
            DATA<= (others=>'Z');
        endif;
    end process DU;
end Behavior;
```

下面的实体 CU 和相应的结构体给出了控制单元 CU 的 VHDL 模型。CU 的输入信号为：RESET，CLK，DAV，RACK 和 WACK，其输出信号为 5 个控制信号 CS1～CS5 和存储器使能信号 MEMEN。其中 CS1，CS3 和 CS5 分别连接到 EN，READ 和 WRITE。应该指出的是，这个模型是寄存器传输级设计，但为了完整地说明系统设计的概念，这里也列出了全部寄存器传输级代码。在综合过程中，综合工具会对状态编码，一般是把每个控制状态映射为一个 D 触发器，所有 D 触发器全由同一时钟信号 CLK 驱动，另一方面，

RESET 信号有效时，系统复位，全部控制信号复位为'0'。由于状态的更新完全在时钟沿上进行，所以这是完全同步的控制单元。从行为上看，控制单元很像把'1'从 CS1 逐次移位到 CS5 的移位寄存器。

```
use IEEE. std_logic_1164. all;
entity CU is
generic(CS_DEL : TIME);
port(CLK, RESET, DAV, RACK, WACK: in Std_logic;
     CS1, CS2, CS3, CS4, CS5,MEMEN : out Std_logic);
end CU;
architecture FSM of CU is
      type STATES is (CSTEP0, CSTEP1, CSTEP2, CSTEP3, CSTEP4,
      CSTEP5);
    signal STATE, NEXT_STATE : STATES;
begin
    Compute: process(RESET, DAV, RACK, WACK, STATE)
    begin
        case State is
            when CSTEP0 =>
                if DAV='1' then
                NEXT_STATE <= CSTEP1;
            else
                NEXT_STATE <= CSTEP0;
            end if;
            CS1 <= '0'; CS2 <= '0'; CS3 <= '0';
            CS4 <= '0'; CS5 <= '0';
            MEMEN <= '0';
        when CSTEP1 =>
            CS1 <= '1';
            if RESET='1' then
                NEXT_STATE <= CSTEP0;
            else
                NEXT_STATE <= CSTEP2;
            end if;
        when CSTEP2 =>
            CS1 <= '0';
            CS2 <= '1';
            if RESET ='1' then
```

```
                NEXT_STATE <= CSTEP0;
            else
                NEXT_STATE <= CSTEP3;
            end if;
        when CSTEP3 =>
            CS2 <= '0';
            CS3 <= '1';
            if RESET='1' then
                NEXT_STATE <= CSTEP0;
            elsif RACK='1' then
                NEXT_STATE <= CSTEP4;
            else
                NEXT_STATE <= CSTEP3;
            end if;
        when CSTEP4 =>
            CS3 <= '0';
            CS4 <= '1';
            if RESET='1' then
                NEXT_STATE <= CSTEP0;
            else
                NEXT_STATE <= CSTEP5;
            end if;
        when CSTEP5 =>
            CS4 <= '0';
            CS5 <= '1';
            if RESET='1' then
                NEXT_STATE <= CSTEP0;
            elsif WACK='1' then
                NEXT_STATE <= CSTEP0;
            else
                NEXT_STATE <= CSTEP5;
            end if;
            MEMEN <= '1';
        end case;
    end process Compute;
    Update: process
    begin
        wait until CLK='1' and CLK'Event;
```

```
        STATE <= NEXT_STATE;
    end process Update;
end FSM;
```

将实体 CU 的结构体 FSM 同前面给出的实体 ADD_STORE 的结构体 ALG 进行比较，可以发现这样一些问题：首先，实体 ADD_S1ORE 的结构体 ALG 是行为级代码，行为级代码的状态是隐含表示的；而实体 CU 的结构体 FSM 的代码是寄存器级的有限状态机模型，其状态是直接表示的。其次，行为级代码中，复位处理用 exit 退出当前循环实现，但是这实际上隐含着一个问题，就是复位处理只能在循环入口处进行；而寄存器传输级代码中，每一个状态均有检查复位信号的代码，即复位可以在任意状态下开始。第三，行为模型中，控制与运算处理结合在一起；而寄存器传输级代码中，这两部分是分离的，控制器只负责输出控制信号，运算部件由实体 ADU 的结构体 Behavior 给出，它只负责在相应控制信号有效时执行操作。之所以作这样的划分，是为了适应综合工具的要求，因为综合工具在单独处理有限状态机和数据运算电路时，能够取得理想的综合结果。

4.3.2 处理复位的其他方法

4.3.1 节中分别介绍了在行为级和寄存器传输级处理复位的方法，实际上还有另外的复位处理方式。在下面的实体 Wait_Steps 的结构体 ONE 中，描述了一个控制单元，这个控制单元可以在任何状态被复位。通过在循环体内插入控制序列可以实现这样的功能。在控制序列中的每一步的时钟沿上，都用 wait 和 next 语句检查信号 RESET 上是否发生事件，一旦信号 RESET 上发生事件，则程序转移到循环体的标号处，即重新开始这一循环体。程序将在循环体的顶部等待直到 RESET 变为'0'。这样，在结构体 ONE 中，仿真开始时模型等待 RUN='1'，一旦出现 RUN='1'，该控制单元按状态 0，1，2 的顺序循环执行直到出现 R='1'。这时仿真不再执行这一循环而进入等待状态。最初一个 wait 语句用来防止 R='1'时出现无限循环，从而不能省略。

```
use IEEE.std_logic_1164.all;
entity Wait_Steps is
    port(CLK, R, RUN: in Std_logic;
         S: out Integer);
end Wait_Steps;

architecture ONE of Wait_Steps is
begin
    Running: process
    begin
        wait until CLK='1' and CLK'Event and R='0' and RUN = '1';
        Loop1: while R = '0' and RUN = '1' loop
            S <= 0;
```

```
            next Loop1 when R = '1';    -- step 0
            wait until CLK='1' and CLK'Event and R = '1';
            ----------------------------
            S <= 1;
            next Loop1 when R = '1';    -- step 1
            wait until CLK='1' and CLK'Event and R = '1';
            ----------------------------
            S <= 2;
            next Loop1 when R = '1';    -- step 2
            wait until CLK='1' and CLK'Event and R = '1';
        end loop Loop1;
    end process;
end One;
```

4.3.3 时分复用

采用总线决断函数实现总线的分时共享的方法称为时分复用。这种时分复用通常是使所有不活动的总线驱动器维持在'Z'态来实现的。事实上,还可以用其他方式实现时分复用。下面的 VHDL 源代码通过两个保护赋值模块实现时分复用。

```
use IEEE.std_logic_1164.all;
entity Time_Mux is
    generic(DEL1, DEL2: TIME);
    port(PHASE_ONE, PHASE_TWO: in Std_logic;
         ZOUT: out Std_logic);
end Time_Mux;

architecture Guard_Block0 of Time_Mux is
begin
    Ph_One: block(PHASE_ONE = '1')
    begin
        ZOUT <= guarded '0' after DEL1;
    end block Ph_One;
    Ph_Two: block(PHASE_TWO = '1')
    begin
        ZOUT <= guarded '1' after DEL2;
    end block Ph_Two;
end Guard_Block0;
```

模块 Ph_One 在 PHASE_ONE='1'时把信号 ZOUT 赋值为'0',模块 Ph_Two 在

PHASE_TWO='1'时把信号 ZOUT 赋值为'1'。PHASE_ONE 和 PHASE_TWO 是两相互不交迭的时钟(如图 4.19 所示),它们不能同时为'1'。信号 ZOUT 是一个 Std_logic 信号,其默认值为'U'。假定硬件设计者的目的是:① 当 PHASE_ONE 为'1'时,由模块 Ph_One控制信号 ZOUT 的取值;当 PHASE_TWO 为'1'时,由模块 Ph_Two 控制信号 ZOUT 的取值。② 当 PHASE_ONE 和 PHASE_TWO 全为'0'时,信号 ZOUT 保持其原有值。假设模块 Ph_One 中信号 ZOUT 的驱动器的输出值为'0',而模块 Ph_Two 中信号 ZOUT 的驱动器的输出值为'1',按上述 VHDL 源代码仿真得出的实际信号波形会出现错误,如图 4.19 所示。为了分析出现这种情况的原因,假定在 PHASE_ONE 变为'1'之前,信号 ZOUT 的初始状态为'X',当 PHASE_ONE 变为'1'时,信号 ZOUT 的值成为'0'。此后,当 PHASE_TWO 变为'1'时,期望信号 ZOUT 的值会变成'1'。由于总线决断函数是其驱动器输出值的静态函数,决断函数的取值与其驱动器上的值保持了多长时间无关,因此,这时信号 ZOUT 的取值并不是期望值'1',而是变成了'X'。

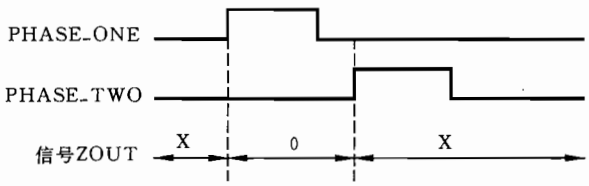


图 4.19　双向互不交迭时钟及不正确的时分复用

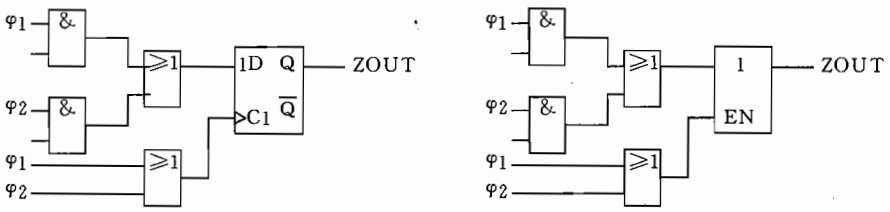
为了解决这种不正确的分时复用问题,可以设想用驱动器最新变化后的值控制信号 ZOUT 的取值。对结构体 Guarded_Block0 稍加改动,使得当保护信号值变为'0'时,信号 ZOUT 的驱动器输出值为'Z',以修正这一错误,即

```
if PHASE_ONE'Event and PHASE_ONE = '0' then
ZOUT <= 'Z';
```

然而,这种修正并不能完全解决问题。如果两个保护信号全为'0',则信号 ZOUT 的两个驱动器的输出值全为'Z',从而信号 ZOUT 的取值也为'Z'。这与设计要求“信号 ZOUT 应保持其原有值”也不相符。事实上,决断函数只能是其输入值的组合逻辑函数,不可能实现“保持其原有值”这种设计要求。不仅如此,以上修正的代码不能被综合,这种设计通常不能被接受。

为了完全解决上述的错误时分复用问题,应该同时使用总线决断函数和保护模式信号赋值。VHDL 为用保护模式赋值的信号提供了特殊处理,如果设计者期望实现时分复用,应该将信号说明为寄存器型(register)或总线型(bus)。对于寄存器型或总线型信号的保护模式赋值,如果信号驱动模块的保护条件为 false,则驱动该信号的赋值语句的作用是将信号与驱动器分离,从而使决断函数忽略该驱动器的作用;如果信号驱动模块的保护条件为 true,则信号驱动器的值进入总线决断函数。这样,假定将信号 ZINT 的所有驱动器全与信号分离,如果 ZINT 是一个寄存器,它的值会保持其原来的数值;如果 ZINT 是一个总线,则它被赋值为总线决断函数的输出默认值。在 IEEE 9 值逻辑中,总线决断函

数输出的缺省值为'Z'。如果 ZINT 为寄存器,它应该为带时钟使能的寄存器,如果 ZINT 为总线,它也应该为带时钟使能的总线。图 4.20 示意给出了带使能时钟的寄存器和总线的结构模型。下面的 VHDL 源代码说明了如何同时使用保护模式赋值和寄存器型、总线型信号实现时分复用。



(a) 带时钟的寄存器时分复用　　(b) 带时钟的总线时分复用

图 4.20　带使能时钟的寄存器和总线时分复用

```
architecture Guard_Block1 of Time_Mux is
    signal ZINT : Std_logic 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;
        ZOUT <= ZINT;
    end Guard_Block1;

architecture Guard_Block2 of Time_Mux is
    signal ZINT : Std_logic 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;
```

```
    ZOUT <= ZINT;
end Guard_Block2;
```

如果对信号赋值的是进程而不是保护模式赋值模块，也存在仿真所得到的信号波形出错的问题，下面的 VHDL 源代码表明了这种问题存在的情况。这里有两个进程驱动信号 ZOUT，由于信号 ZOUT 是其驱动器的静态函数，其取值与该值保存在其驱动器上的时间长短无关，这个模型得出的信号 ZOUT 的波形仍然如图 4.19 所示。

```
architecture Proc of Time_Mux is
begin
    Ph_One: process(PHASE_ONE)
    begin
        if PHASE_ONE = '1' then
            ZOUT <= '0' after DEL1;
        end if;
    end process Ph_One;
    Ph_Two: process(PHASE_TWO )
    begin
        if PHASE_TWO = '1' then
            ZOUT <= '1' after DEL2;
        end if;
    end process Ph_Two;
end Proc;
```

解决这一问题的方法与保护模式模块赋值类似，引入一个寄存器型信号 ZINT，在两个进程中，如果条件 PHASE_ONE 或 PHASE_TWO 变为'0'，则把值 null 赋给 ZINT，这样做的效果是使信号驱动器与决断函数隔离开。如果 PHASE_ONE 和 PHASE_TWO 不同时为'1'，则只有活动驱动器(非 null 值)可以进入决断函数。如果 ZINT 的两个驱动器全不活动，因为 ZINT 为一个寄存器，所以 ZINT 保持其旧值。假定 ZINT 为一个总线信号，在其所有驱动器的输出值全为 null 的情况下，总线的取值为总线决断函数的缺省输出值。下面的 VHDL 源代码实现了这种时分复用。

```
-- 实现时分复用的另一种形式
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;
        elsif
```

```
            ZINT <= null;
        end if;
    end process Ph_One;
    Ph_Two: process(PHASE_TWO )
    begin
        if PHASE_TWO = '1' then
            ZINT <= '1' after DEL2;
        elsif
            ZINT <= null;
        end if;
    end process Ph_Two;
    ZOT <= ZINT;
end Proc_Null;
```

应该指出的是，由于在系统级采用了一定的 VHDL 语法结构描述电路行为才出现了这里的时分复用问题。在硬件设计的较低层次，只要设计出实际电路，就自然可以实现时分复用，可以按这个实际电路写出 VHDL 行为模型。尽管如此，在系统级设计时，在没有设计出电路之前，写出具有时分复用能力的算法模型还是很有用的。另一方面，如果要求采用自动综合工具综合出具有时分复用能力的硬件，硬件的行为描述必须能描述时分复用的行为。

4.4 系统级设计实例

在这一节中，以与 Intel 公司 MCS51 体系兼容的 8 位微控制器为例，总结系统设计各个方面的问题。就微控制器而言，它本身并不服务于单一的算法，因此其系统模型实际上是分别描述微处理器各条指令的算法，这些指令在执行时要共享功能部件。在给出微控制器的 VHDL 算法模型的框架之后，再将其划分为控制单元、数据路径以及由外围 I/O 电路构成的协处理器。本节还对如何确定系统状态、系统划分和总体通信协议进行讨论。

4.4.1 80C51 概述

80C51 是 Intel 公司开发的 8 位微控制器，属于 Intel 的 MCS51 体系。图 4.21 是 80C51 体系框图，其主要特征如下：

(1) 8 位数据总线中央处理单元；

(2) 片上时钟电路；

(3) 32 根 I/O 线(分为 4 个 I/O 端口)；

(4) 64 千字节数据地址空间(片内 128 字节)；

(5) 64 千字节指令地址空间(片内 4 千字节)；

(6) 2 个多功能 16 位定时器/计数器；

(7) 5 个中断源、两优先级的中断结构；

(8) 全双工串行口；

(9) 布尔运算处理。

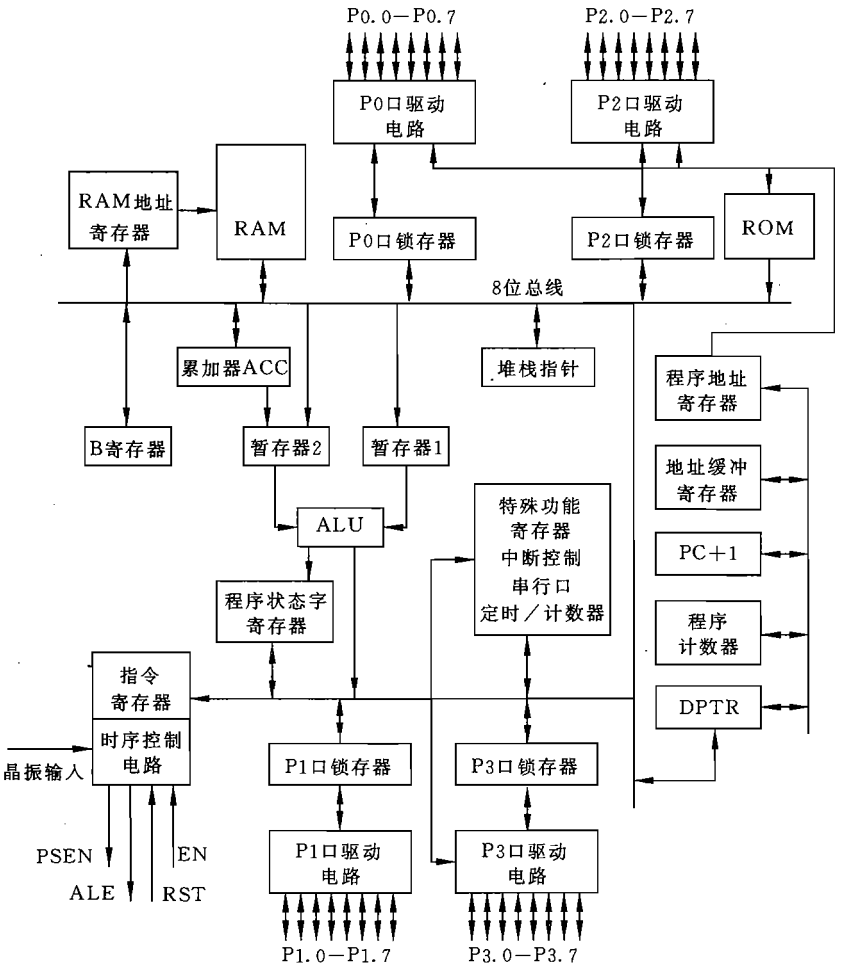


图 4.21　Intel 公司的 80C51 的体系框图

80C51 的外部引脚包括两条电源、两条晶体、四条控制和 32 条 I/O 引脚(分为 4 个 8 位口)。其指令集由 111 种 255 条指令组成,可以进行各种数据传送、算术逻辑操作、转移和位操作。整个体系属于复杂指令集计算机,因此指令长度有 1,2 和 3 字节三种,都以第 1 字节为操作码;使用两相不重叠时钟方案,单条指令的执行时间分为 1,2 和 4 指令周期三种,每个指令周期分为 6 个状态,每状态的长度为 1 个时钟周期。下面的源代码是与 80C51 保持体系兼容的微控制器的 VHDL 行为描述,这是可以仿真和综合的系统级算法模型。因为源代码较长,所以只列出程序框架和少数几条指令的算法描述。该微控制器是按照 80C51 配置的,即指令地址空间为 64kB,片内有 4kB ROM,128 字节数据 RAM(其高端构成堆栈),以及 ACC 等特殊功能寄存器。128 字节内部数据 RAM 的配置比较特殊,低 32 个字节被分成 4 组共 32 个工作寄存器,有专用指令进行寻址,例如 INC R0 指

令的机器码是 08H，只要一个字节。

```
——Title：MCS51.vhd
——Description：VHDL behavior description of a microcontroller with MCS51
    compatible architecture
——date：April，1997

MCS51_CPU：process
    ……；      —— 变量和信号以及其他各种定义
    variable RAM ：Random_Access_MEM；
    constant ROM ：Read_Only_MEM ：=(
            ("01110100"), ("11110000"),        ——MOV A，#F0H
            ("01111000"), ("01110101"),        ——MOV R0，#75H
            ("00101000"),                      ——ADD A，R0
            ("00100100"), ("11110010"),        ——ADD A，#F2H
            ……)；
    procedure Fetch_Instruction(PC ：inout Std16；IR ：out Std8) is
        variable ROM_ADRS ：std_logic_vector(11 downto 0)；
    begin
        ROM_ADRS ：= PC(11 downto 0)；
        wait until CLK'Event and CLK='1'；
        IR ：= ROM(To_Int(ROM_ADRS))；
        wait until CLK'Event and CLK='1'；
        PC ：= INC(PC)；
    end Fetch_Instruction；

    procedure Ram_Decode(IR ：in Std8；RAM_ADRS ：out Std8) is
    begin
        if  IR(3 downto 1)="011" then
            wait until CLK'Event and CLK='1'；
            RAM_ADRS ：= "000" & RS1 & RS0 & "00" & IR(0)；
        else if (IR = "11100010" or IR = "11100011" or IR = "11110010"
                    or IR = "11110011") then
            wait until CLK'Event and CLK='1'；
            RAM_ADRS ：= "000" & RS1 & RS0 & "00" & IR(0)；
        else
            wait until CLK'Event and CLK='1'；
            RAM_ADRS ：= "000" & RS1 & RS0 & IR(2 downto 0)；
```

```
        End if;
        wait until CLK'Event and CLK='1';
    end Ram_Decode;

    procedure Init(RAM : out Random_Access_MEM) is
    begin
        for I in 0 to 127 loop
            RAM(I) := "11110000";
        end loop;
        PC := (others=>'0');
        ACC := (others=>'0');
        SP := "00000111";
        ……
    end Init;
……其他函数和过程
    reset_loop: loop
        Init(RAM);              --为其他特殊寄存器赋初值
        wait until CLK'Event and CLK='1';
        if RESET = '1' then     --RESET 高电平有效
            exit reset_loop;
        end if;
        run_loop: loop
            Fetch_Instruction(PC, IR);      --取指令
            wait until CLK'Event and CLK='1';
            Ram_Decode(IR, RAM_ADRS);  --数据地址译码
            TMP2 := ACC;
            wait until CLK'Event and CLK='1';
            if RESET = '1' then
                exit reset_loop;
            end if;
            case IR_Q is            --指令译码
                when OP_NOP =>      --NOP 指令
                    wait until CLK'Event and CLK='1';
                    ……              --4 个 wait 语句
                when OP_MOV_A_R0|OP_MOV_A_R1|OP_MOV_A_R2|
                    OP_MOV_A_R3|OP_MOV_A_R4|OP_MOV_A_R5|
                    OP_MOV_A_R6 |OP_MOV_A_R7 =>
                                    --MOV A, Rn 指令
```

```
        wait until CLK'Event and CLK='1';
        ACC := RAM(To_Int(RAM_ADRS));
        wait until CLK'Event and CLK='1';
        ……              ——3 个 wait 语句
    when OP_INC_R0|OP_INC_R1|OP_INC_R2|
        OP_INC_R3|OP_INC_R4|OP_INC_R5|
        OP_INC_R6|OP_INC_R7 =>      ——INC Rn 指令
        wait until CLK'Event and CLK='1';
        TMP2 := RAM(To_Int(RAM_ADRS));
        wait until CLK'Event and CLK='1';
        TMP1 := "00000001";
        wait until CLK'Event and CLK='1';
        ADDC_Set(TMP1,TMP2,'0',ALU_Q,CY,AC,OV);
        wait until CLK'Event and CLK='1';
        RAM(To_Int(RAM_ADRS)) := ALU_Q;
        wait until CLK'Event and CLK='1';
    when OP_LCALL =>        ——LCALL 指令
        wait until CLK'Event and CLK='1';
        Fetch_Instruction(PC,IR);
        wait until CLK='1' and CLK='1';
        Buf1 := IR;
        wait until CLK'Event and CLK='1';
        Fetch_Instruction(PC,IR);
        wait until CLK='1' and CLK='1';
        BUF2 := IR;
        wait until CLK'Event and CLK='1';
        SP := INC(SP);
        wait until CLK='1' and CLK='1';
        RAM_ADRS := SP;
        wait until CLK'Event and CLK='1';
        RAM(To_Int(RAM_ADRS)) := PC(7 downto 0);
        wait until CLK ='1' and CLK='1';
        SP := INC(SP);
        wait until CLK'Event and CLK ='1';
        RAM_ADRS := SP;
        wait until CLK='1' and CLK='1';
        RAM(To_Int(RAM_ADRS)) := PC(15 downto 8);
        wait until CLK'Event and CLK='1';
```

```
                    PC(15 downto 8) := BUF1;
                    wait until CLK'Event and CLK='1';
                    PC(7 downto 0)  := BUF2;
                    wait until CLK='1' and CLK='1';
                when ……              ——其他指令
            end case;
            wait until CLK'Event and CLK='1';
        end run_loop;
    end reset_loop;
end process MCS51_CPU;
```

微控制器的工作是一个周而复始的取指令、分析指令和执行指令的过程，指令的执行又可以细分为数据地址译码、取操作数、计算和写回结果这几个步骤。在前面的 VHDL 代码中，整个微控制器的行为用一个进程 MCS51_CPU 描述。

进程的声明部分定义了需要使用的各种变量、常数、函数和过程。其中片内的数据 RAM 被定义为变量数组 RAM，指令存储器被定义为常量数组 ROM，其余特殊功能寄存器均定义成变量，数据类型都使用 IEEE 标准的 std_logic 和 std_logic_vector。用户可以看到寄存器、PC 以及数据 RAM 是不能硬件共享的，也就是说它们的生存期是整个进程。

指令存储在数组 ROM 中，当前指令地址由 16 位寄存器 PC 计算，当前指令由指令寄存器 IR 存储，IR 实际上是 ROM 的输出缓冲器。过程 Fetch_Instruction 的功能是取指令，需要 3 个时钟周期，它首先把 PC 值载入 ROM 地址锁存器 ROM_ADRS，再将 ROM 中相应地址处的内容读入 IR，然后还要把 PC 值加 1，使 PC 指向下一指令地址。MCS51 的指令可能由多个字节构成，这时需要多次调用过程 Fetch_Instruction。过程 Ram_Decode 完成数据地址译码，即根据指令码翻译出相应的内部数据空间地址，载入 RAM 地址锁存器 RAM_ADRS。可以看到，为了满足调度要求，在各个条件分支上都有 wait 语句。过程 Init 的作用是为各个寄存器赋予初值，这包括将内部数据空间的低 128 字节 RAM 初始化为 0；将 PC 清 0，以便程序从指令 ROM 的起始处开始执行；堆栈指针寄存器的初值置为 07H，其他各个特殊功能寄存器也要进行相应的初始化操作。以上全部操作要在一个时钟周期内完成，因为表示各用户可见寄存器和 PC 的变量是不能共享寄存器资源的。

进程 MCS51_CPU 采用同步复位方式，CLK 是输入时钟信号，RESET 是输入复位信号。按照 Intel 的微控制器手册，80C51 芯片采用两相时钟，在两个时钟的有效电平分别进行各种操作，即一个机器周期内有 12 个时间控制点。但由于现有行为综合工具只支持单相沿时钟触发方案，所以代码中只有一个时钟信号。只要把一个指令周期划分为 12 个状态，在这种方式下程序的执行时序与 MCS51 仍然是相同的。这段代码在调度时要采用固定周期 I/O 调度模式，以保证其兼容性。

进程内的代码构成两个嵌套的循环：reset_loop 和 run_loop。reset_loop 的作用是处理复位，在上电复位（在仿真时相当于第一次激活进程）或复位信号 RESET 有效时进程被强制进入这个循环的起始处开始执行，调用 Init 过程完成初始化操作。当复位信号无效

之后，转入循环 run_loop 的执行。此后，只要不发生复位信号有效的情况，程序就在这个循环内反复执行，相当于微控制器正常工作。

run_loop 的代码首先调用 Fetch_Instruction 过程读取当前指令，然后调用 Ram_Decode 过程进行数据地址译码。实际上并不是每条指令的执行都需要数据地址译码，但考虑到很大一部分指令要读写数据 RAM，所以为了降低硬件复杂度，在执行任何指令时都经过这一步骤。由于 80C51 是在以累积器为核心的体系上发展起来的，指令集中有很多指令是围绕累加器 ACC 进行，所以接下来把 ACC 的数值载入暂存器 TMP2(ALU 有两个输入缓冲器 TMP1 和 TMP2)。此时各条指令的公共操作均已执行完毕，开始进行指令译码。在行为级模型中，通过 case 分支语句实现指令译码，每个分支都是一条指令的算法。

代码中给出了 NOP，MOV A，Rn，INC Rn，LCALL 等几条指令的算法描述。其中，NOP 指令不执行任何操作，因而插入了 5 个 wait until 语句。这样，译码前已经过 6 个状态(即有 6 个 wait until 语句)，再加上分支结束循环返回的一个状态，完成单周期指令正好经过 12 个时钟周期。INC Rn 指令(工作寄存器内容加 1)的算法是这样的：首先根据由 Ram_Decode 译出的 RAM 有效地址，将该处的工作寄存器内容装入 TMP2 暂存器，然后将 TMP1 暂存器置为"00000001"，TMP1 和 TMP2 的相加结果由过程 ADDC_Set 计算，最后写回工作寄存器。LCALL(长调用)指令执行时，先从 ROM 中读取被调用程序的入口地址，存入变量 BUF1 和 BUF2，然后将 SP+1 与 SP 内容装入 RAM 地址锁存器，读 PC 低 8 位写入堆栈(内部 RAM 高端)，调用地址低 8 位写入 PC 低 8 位，接下来以同样方式处理高 8 位。一次只处理 8 位数据是为了引导行为综合器生成 8 位的数据路径。

4.4.2 系统状态的确定

微控制器是典型的带有数据路径的有限状态机，其中时钟电路与控制单元组合构成

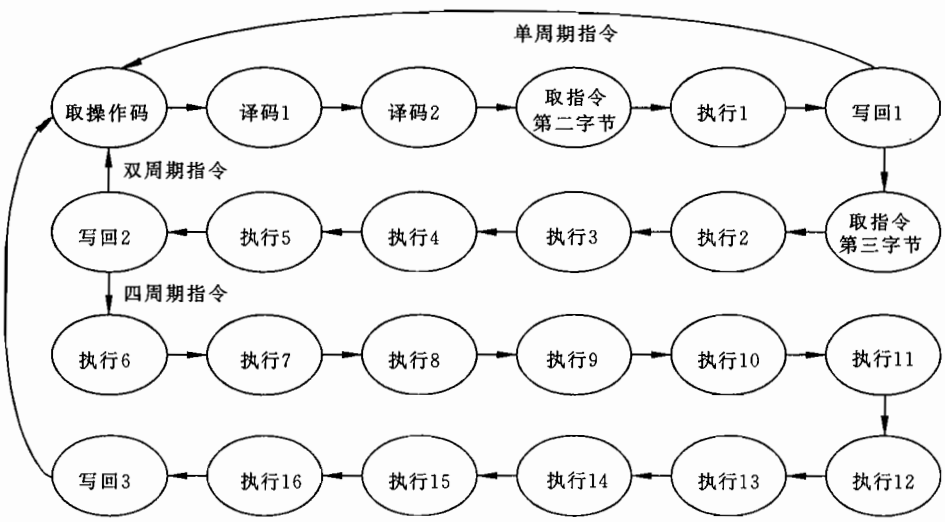


图 4.22　80C51 芯片的状态转换图

了有限状态机，指令的运算在数据路径中进行，外围电路构成 CPU 的协处理器。在这里最重要的是如何确定状态和如何表示状态。根据对 80C51 指令集的分析，确定了 24 个状态，芯片的状态转换关系见图 4.22。

对所有指令来说，前 6 个状态都是相同的，即先从指令 ROM 中读取指令操作码，经过两级译码，形成微操作控制信号，同时进行数据地址译码，形成操作数有效地址。在第 4 个状态，如果是多字节指令，要从 ROM 中读取指令的第二字节；如果是间接寻址，则要形成新的数据有效地址。第 5 个状态进行指令执行的第 1 步骤。第 6 个状态写回结果。对于单周期指令，在此即执行完毕，返回第 1 状态，开始执行下一条指令；否则，还要经过下面的状态，完成指令的执行。

4.4.3 芯片体系的第一步划分

确定系统状态后，可以着手编写芯片的行为级描述，在 4.4.1 节中已经列出了行为级模型的程序框架。行为级的模块适合于作为功能定义和算法描述，但是由于行为描述中没有进行时序调度和硬件资源分配，所以还不适用于作为寄存器级综合的输入代码。现在有些高层次综合工具，可以把行为级的 HDL 描述自动转换为寄存器级的描述，然后对这一级描述进行逻辑综合（这一步骤又可以细分为寄存器级综合和逻辑综合两步）。如果硬件是为了实现某一特定算法，这种设计流程具有极大的优越性，可以大幅度地缩短设计周期。但是对通用微控制器来说，它本身并不服务于特定算法，每一条指令既可以采用专门硬件来实现，也可以通过一系列较简单硬件的协同动作完成。通常是每一条指令的算法不尽相同，但各条指令共享微控制器的全部硬件资源。在这种情况下，行为级综合通常不能达到理想的效果，而设计者的主要任务就是时序调度和硬件资源分配。因此，在本章的设计中，完成行为算法描述之后，按照由顶向下的设计方法，用人工的方法对芯片进行划分。划分的过程分两个步骤，首先把芯片分割为控制器和数据路径，再进一步细划为一系列寄存器级的功能模块。当然，对于比较简单的设计，则可以直接划分到寄存器级的单元。

第一步划分的目的在于利用已经验证过的算法，把芯片的控制和数据运算部分分离。对控制部分来讲，分离后的模型应能够对确定指令发出确定的微操作序列；对数据路径来讲，要确定运算资源分配，也就是说，把执行具体指令的微操作序列分配到一定的硬件资源上，并能够根据微操作控制信号的状态完成相应操作。

在本章的设计中，第一步划分的结果是把芯片分割为控制单元和数据路径。控制单元中包含指令 ROM，控制单元从指令 ROM 中读取当前指令，然后进行相应的状态转换，并在各个状态上发出一定的微操作控制信号。实际上，这一步划分的主要目的就是确定微操作。数据路径包含各个特殊功能寄存器、ALU 和数据存储器，但这种包含关系是隐含的，也就是说，特殊功能寄存器和数据存储器是以信号和信号数组形式表示，而 ALU 的功能体现在各种数据运算函数或过程中。第一步划分后的系统框图如图 4.23。

微控制器的任务主要是用作程序的执行平台。程序的编制可能采用高级语言，也可能直接使用机器码。但最后总要以自动或人工的方式转换为一段指令流，然后才能在微控制器上执行。对于指令集中拥有几百甚至上千条指令的微控制器来说，不可能为每一条指令都配备专用硬件。因此人们提出了微操作的概念。所谓微操作，通常是指 CPU 中的寄存

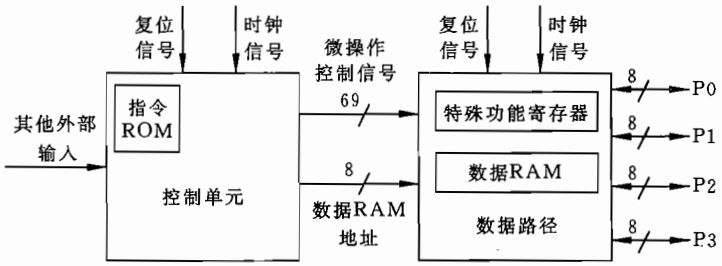


图 4.23　经过第一步划分的微控制器

器级电路单元在某一单一的时间控制点(时钟沿或有效电平)上能够完成的操作。在执行程序时,每一条指令经过指令译码,被翻译成一系列微操作控制信号,控制相应的单元完成一定微操作,从而实现指令的执行。也就是说,每一条指令均对应于一个特定的微操作流,称为微码(微码一词没有严格的定义,也常把微代码控制器的 ROM 中存储的全部译码程序称为微码)。微操作包含三个要素:执行元件、执行动作和控制信号,其中控制信号反映出微操作的执行时序。

微操作的确定和体系划分有着密切的联系,前者建立在后者基础之上,并且反映体系的效率。通常这两个步骤是迭代的过程,需要根据实际情况权衡各种资源,才能够完成。在微操作的确定上,没有统一的方法,在本次设计中遵循以下原则:

(1) 不同指令之间尽量拥有共同的微操作,即微操作的确定要针对整个指令集进行优化,这主要是为了降低译码的复杂度。

(2) 微操作复杂度适当,因为微操作必须在一个时钟控制点上完成,所以如果过于复杂,则必须增大时钟周期,降低工作速度,同时过于复杂的微操作会违背上述第(1)条原则;反之,若过于简单,则指令的微操作序列会很长,即微操作个数很多,这样会增加译码的复杂度,并且同样会造成性能的下降。

(3) 在对全指令集优化的同时,应该重点保证执行频率较高的指令有较简单的微操作流。

根据 80C51 指令和整个体系,把指令完成的操作分解为 80 个左右的微操作,每一条指令均由一系列微操作组成,分别在一定的时序上完成,以实现指令的执行。比如,"ADD A, Rn"这条指令,由取指、程序计数器递增量、锁存新的程序地址、ACC 的值进入暂存器 2、寄存器堆地址译码、寄存器内容上总线、总线内容进入暂存器 1、ALU 执行加法、ALU 内容上总线、总线内容进入 ACC 等 10 个微操作组成。这些微操作的每一个分别由一个 OP 控制信号来控制,指令译码即把指令翻译为一系列 OP 控制信号。

微操作确定后,下一步要书写控制单元的 VHDL 模型。这个模型仍以算法式的风格书写,状态是通过 wait until CLK='1' and CLK'Event 语句隐含表示的,因为在以后的设计中要使用两相时钟,因此图 4.22 中的每一个状态相当于两个时钟周期。如果把代码中的每一个 wait 语句视为一个时钟周期,那么一个机器周期相当于 12 个时钟周期,这里的每两个 wait 状态等于图 4.22 中的一个状态。在这段代码中,控制单元的输入除了复位、时钟和片外输入外,还有来自数据路径的程序状态字寄存器。程序的核心思想是在取指

令、数据地址译码(分别通过 Fetch_Instruction 和 Ram_Decode 过程实现)之后,用 case 语句进行指令译码,然后在各个分支上依次使微操作控制信号有效('1'有效)。需要注意的是,在复位时要将所有微操作控制信号初始化为'0'。下面代码中以 swap A 和 inc R0 指令为例。执行 swap A 指令时,在输入时钟的第 9 个上升沿,微操作控制信号 OP_SWAP 被置为'1',控制数据路径完成相应操作,两个时钟周期后,OP_SWAP 重新被置为'0',而此时 ACC 寄存器已经完成高低 4 位的互换。执行 inc R0 指令时,首先使 OP_RAM_RD 和 OP_BUStoTMP 有效,控制 RAM 内相应字节送到暂存器 2,加法结束后,OP_ALUtoBUS 和OP_RAM_WR 有效,使得相加结果写回 RAM。其他指令的处理与此类似。控制单元的 VHDL 模型如下:

```
-- 第一步划分后的控制单元的 VHDL 算法模型
--Title: ctrl_unit.vhd
-- Description: VHDL behavior description of an MCS51-compatible CPU's
control-unit
--date: May, 1997

entity Ctrl_unit is
    port(RST, CLK : in Std_logic;
         PSW : in Std_logic_vector(7 downto 0);
         ......
             --其他输入
         OP_SWAP : out Std_logic;      --完成 SWAP 指令的微操作控制信号
         OP_XCHD : out Std_logic;      --完成 XCHD 指令的微操作控制信号
         ......
             --其他微操作控制信号,连上面两个信号共 69 个
         RAM_Adrs : out Std_logic_vector(6 downto 0);  --RAM 地址
         ......
             --其他输出
         );
end Ctrl_unit;

architecture ALG of Ctrl_unit is
begin
    Micro_OP: process
        variable PC : Std_logic_vector(15 downto 0);
        variable IR : Std_logic_vector(7 downto 0);
        ......;
                --- 变量和信号以及其他各种定义
```

```
    constant ROM : Read_Only_MEM :=(
                ("01110100"), ("11110000"),     --MOV A, #F0H
                ("01111000"), ("01110101"),     --MOV R0, #75H
                ("00101000"),                   --ADD A, R0
                ("00100100"), ("11110010"),     --ADD A, #F2H
                ......);
begin
    Reset_loop: loop
        PC := (others=>'0');
        --此处将所有微操作控制信号赋值为'0'
        wait until CLK'Event and CLK='1';
        if RESET = '1' then              --RESET 高电平有效
            exit Reset_loop;
        end if;
        Run_loop: loop
            Fetch_Instruction(PC, IR);      --取指令
            wait until CLK'Event and CLK='1';
            OP_BUStoTMP2 <= '1';
            Ram_Decode(IR, RAM_ADRS);  --数据地址译码
            OP_BUStoTMP2 <= '0';
            wait until CLK'Event and CLK='1';
            if RESET = '1' then
                exit Reset_loop;
            end if;
            case IR_Q is
                when INS_SWAP =>      --SWAP ACC 指令
                    wait until CLK'Event and CLK='1';
                    wait until CLK'Event and CLK='1';
                    wait until CLK'Event and CLK='1';
                    OP_SWAP <= '1';
                    wait until CLK'Event and CLK='1';
                    wait until CLK'Event and CLK='1';
                    OP_SWAP <= '0';
                    wait until CLK'Event and CLK='1';
                when INS_INC_R0 =>          --INC R0 指令
                    wait until CLK'Event and CLK ='1';
                    OP_RAM_RD <= '1';
                      --读 RAM(前面已完成数据地址译码)上总线
```

```
                    OP_BUStoTMP2 <= '1'; --总线内容进入暂存器 2
                    --每条指令开始执行之前,暂存器 1 预置为 1
                    --默认情况下 ALU 执行加法
                    wait until CLK'Event and CLK='1';
                    wait until CLK'Event and CLK='1';
                    OP_RAM_RD <= '0';
                    OP_BUStoTMP2 <= '0';
                    OP_ALUtoBUS <= '1';
                    OP_RAM_WR <= '1';
                    wait until CLK'Event and CLK='1';
                    wait until CLK'Event and CLK='1';
                    OP_ALUtoBUS <= '0';
                    OP_RAM_WR <= '0';
                    wait until CLK'Event and CLK='1';
                when ···            --其他指令的处理
            end case;
        end loop Run_loop;
    end loop Reset_loop;
end process Micro_OP;
end ALG;
```

下面给出的是划分后数据路径的 VHDL 代码。这段代码是直截了当的,即在时钟沿上检查各个输入的微操作控制信号,如果为'1',则执行相应操作。

```
-- 第一步划分后的数据路径的 VHDL 算法模型
--Title: Datapath.vhd
--Description: VHDL behavior description of an MCS51-compatible CPU's
datapath
--date: May, 1997

entity Datapath is
    port(RST, CLK : in Std_logic;
         P0, P1, P2, P3 : inout Std_logic_vector(7 downto 0);
         PSW : buffer Std_logic_vector(7 downto 0);
         ·····                    --其他输入
         OP_SWAP:in Std_logic;--完成 SWAP 指令的微操作控制信号
         OP_XCHD:in Std_logic;--完成 XCHD 指令的微操作控制信号
         ·····          --其他微操作控制信号,连上面两个信号共 69 个
         RAM_ADRS : in Std_logic_vector(6 downto 0); --RAM 地址
```

```
            ……          ——其他输出
            );
end Datapath;

architecture ALG of Datapath is
    signal RAM : Random_Access_MEM;
    signal ACC : Std_logic_vector(7 downto 0);
    ……    —— 其他信号、函数和过程
begin
    Execute: process
        variable DBUS : Std_logic_vector(7 downto 0);
        procedure Init is
        begin
            for I in 0 to 127 loop
                RAM(I) := "11110000";
            end loop;
            ACC := (others=>'0');
            SP := "00000111";
            ……
        end Init;

    begin
        Reset_loop: loop
            Init;                    ——为其他特殊寄存器赋初值
            wait until CLK'Event and CLK='1';
            if RESET = '1' then              ——RESET 高电平有效
                exit Reset_loop;
            end if;
            Run_loop: loop
                wait until CLK'Event and CLK='1';
                if RESET = '1' then          ——RESET 高电平有效
                    exit Reset_loop;
                end if;
                if OP_SWAP='1' then
                    wait until CLK'Event and CLK='1';
                    ACC <= ACC(3 downto 0) & ACC(7 downto 4);
                elsif OP_BUStoTMP2='1' then
                    DBUS <= RAM(To_Int(RAM_ADRS));
```

```
                TMP2 <= DBUS;
            elsif (OP_ALU0='0' and OP_ALU1='0' and OP_ALU2='0') then
                    ALU_Q <= TMP1 + TMP2;
            elsif
                ......
                ......          ——对其他微操作控制信号的处理
            end if;
            wait until CLK'Event and CLK='1';
            end loop Run_loop;
        end loop RESET_loop;
    end process Execute;
end ALG;
```

4.4.4 芯片体系的第二步划分

按照划分的原则，把芯片划分为寄存器级单元后，芯片的体系也就随之确定，如图 4.24。把该图与图 4.21 比较，可以看到左边的数据路径与原来基本相同，这主要是为了保持体系上的兼容。右边的控制单元部分则根据实际需要进行了调整，由于这部分是用户不可见的，因此不会影响兼容性。

时钟采用两相不重叠时钟方案。因为 80C51 属于复杂指令系统体系，指令时序较为复杂，而且不同指令间执行时间相差很大，所以引入了周期控制信号 T1，T2 和 T4，对应于一、二和四周期指令。每一指令周期分为 6 个状态，每一状态由两相时钟分为两个节拍。

在内部互连方案的选择上，通常可以采用两种方案：总线或多路选择器方式互连。多路选择器方式速度快，但是需要在芯片中设置很多个不同尺寸的多路选择器电路单元，增大了芯片面积，也不利于版图的规格化。因此，为了节省芯片面积，选用了总线互连方式。考虑到 80C51 的工作情况，没有采用复杂的总线协议，而是使用简单的三态总线，即寄存器通过三态门与总线相连。三态使能有效时，写总线，否则，输出高阻态。

数据路径由累加寄存器 ACC、乘除 B 寄存器、寄存器堆（128 个 8 位寄存器，包括四组共 32 个工作寄存器、16 个可位寻址的 8 位寄存器及其他寄存器，寄存器堆的高端构成堆栈）、两个暂存器、ALU 和布尔处理器组成。ALU 可完成 8 位的算术和逻辑运算。为减小面积，位操作也在 ALU 内完成，运算时取 8 位数据，由布尔处理器产生掩码确定操作位，并在 ALU 内生成相应常数完成运算。如求反运算，则通过相应位与 1 相异或实现。

控制单元由程序计数器 PC、指令译码器、地址生成电路及中断逻辑组成。PC 为 16 位计数器；指令译码采用硬连线译码，而没有用微码，这样可以极大地提高速度，同时由于硬连线译码全部由 VHDL 代码综合生成，所以仍然具备便于修改和升级的特点。地址生成电路是由 0 比较器和 16+8 位加法器构成，用于跳转控制和跳转电路生成。PC 输出到 ROM 地址锁存器。中断逻辑包括改进的菊花链中断判优逻辑和中断向量入口地址生成

电路。

系统级设计的主要任务是确定算法和体系划分，划分为寄存器传输级单元之后，设计转入寄存器传输级设计，本例的寄存器传输级代码将在第 5 章介绍。

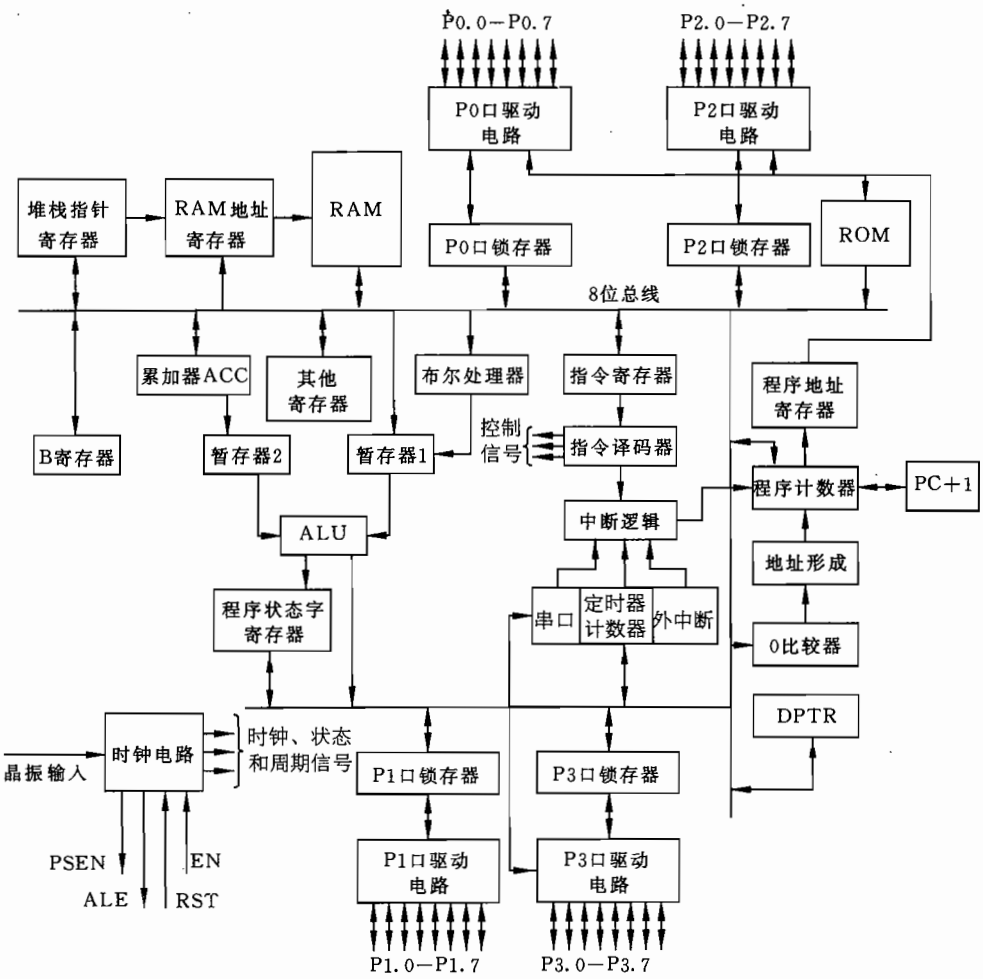


图 4.24　经过第二步划分的微控制器

第5章 寄存器传输级设计

本章讨论集成电路的寄存器传输级设计。在寄存器传输级，硬件通常可以分为控制单元和数据路径两类，控制单元一般用有限状态机方式或微码 ROM 描述，而数据路径则以 VHDL 数据流方式描述，或者被分解为一系列的寄存器级单元，如 ALU、矢量寄存器及其控制逻辑等，然后以互连的形式描述。在当前的超大规模集成电路设计过程中，主导的设计方法仍然是把寄存器传输级代码作为综合输入，因此，本章首先从设计的角度讨论寄存器级 VHDL 模型，这又可以分为两部分，即如何将数据路径的算法模型变化为数据流模型，以及如何将算法模型中的控制电路翻译为寄存器级的控制单元。然后从综合的角度讨论寄存器级代码与综合电路的关系。合理的代码风格是控制综合结果的最有效方式，芯片设计师能够熟练地掌握有效的 VHDL 编程风格，对于设计成功是至关重要的。

5.1 由算法模型向数据流模型的变换

第 4 章讨论了 VHDL 算法模型的设计，并对如何把算法模型映射为硬件电路进行了讨论。这里讨论两个基本问题，第一个问题是寄存器传输级的数据流描述与系统级的算法描述的区别在哪里；第二个问题是什么样的硬件行为适合于在寄存器级用数据流方式造型，而不适合于在系统级用算法方式造型。

首先考虑第一个问题，算法模型意味着将寄存器等电路模块变换为进程，而数据流模型则是具体表示出寄存器等电路模块。具体地讲，硬件的数据流模型有如下特点：

(1) 数据流模型中的信号代表了硬件中数据的实际移动方向以及电路的互连关系；

(2) 数据流模型中的语句与实际寄存器的结构模型之间存在直接的映射关系；

(3) 数据流模型指定了寄存器级的电路元件之间的连接关系，从而隐含了电路结构；

(4) 数据流模型指定了存储单元的复用结构及总线；

(5) 数据流模型中明确指定了各个寄存器的驱动时钟；

(6) 数据流模型中通常不采用抽象的数据类型，如浮点数、记录(record)、文件等，而把它们变换成 Bit，Bit_vector，Std_ulogic，Std_ulogic_vector，Std_logic，Std_logic_vector 等数据，从而指定了寄存器的长度。

对于第二个问题，由于数据流模型显式定义了寄存器间的互连关系，下述内容适合于在寄存器级采用数据流模型进行研究，而不适合在系统级采用算法模型研究。

(1) 寄存器级元件间的时序关系；

(2) 硬件资源分配；

(3) 调度；

(4) 微代码控制单元设计；

(5) 总线设计。

下面采用一个实例说明如何把算法模型变换为寄存器级数据流模型。某系统由两个8bit 寄存器 R1,R2 和一个加法器组成。用一个 2bit 信号 CON 设定操作指令。系统可完成的 4 种操作如下：

(1) CON = "00",加载寄存器 R1；

(2) CON = "01",加载寄存器 R2；

(3) CON = "10",将 R1 中的数值与 R2 中的数值相加；

(4) CON = "11",从 R1 中减去 R2 的值,通过 R1 与 R2 的反码相加再加 1 实现。

下面的 VHDL 源代码是该系统的算法模型。这段代码使用了 IEEE 颁布的 std_logic_arith 算术运算程序包,所以能够直接用"+"实现两个位矢量的相加,综合时也可以使用 std_logic_arith 程序包给出的实现加法的算法来实现加法运算,当然这一定是通用的算法,不能针对具体需要进行优化。同时,在译码、多路选择器和寄存器分配等方面,尽管算法模型说明了该系统完成何种功能,但不能说明硬件如何实现这些功能。因此还需要书写数据流式的代码引导综合器得到优化的电路。

```
-- 某系统的算法 VHDL 模型
library IEEE;
use IEEE. std_logic_1164. all;
use IEEE. std_logic_arith. all;
entity Reg_Sys is
    port (CLK: in Std_logic;
          CON: in Std_logic_vector(0 to 1);
          INP: in Std_logic_vector(7 downto 0));
end Reg_Sys;

architecture ALG of Reg_Sys is
    signal R1, R2: Std_logic_vector(7 downto 0);
begin
    function: process
    begin
        wait until CLK='1' and CLK'Event;
        case CON is
            when "00" => R1 <= INP;
            when "01" => R2 <= INP;
            when "10" => R1<=R1 + R2;
            when "11" => R1<= R1 + not(R2) + "00000001";
```

```
        end case;
    end process;
end ALG;
```

下面的 VHDL 源代码给出了这个系统的寄存器级数据流模型。这个数据流模型具有如下特点:① 信号表示了系统中的互连关系;② 直接定义了寄存器、复用器、译码器、数据运算(加法、求反、增加 1 等)等硬件单元;③ 信号赋值表示了数据的实际移动。

```
-- 数据流模型 1
architecture DF1 of Reg_Sys is
    signal MUX_R1, R1, R2, R 2C, R2TC, MUX_ADD, SUM:
                                    Std_logic_vector(7 downto 0);
    signal D00, D01, D10, D11, R1E: Std_logic;
begin
    D00 <= not CON(0) and not CON(1);   -- 译码部分
    D01 <= not CON(0) and CON(1);
    D10 <= CON(0) and not CON(1);
    D11 <= CON(0) and CON(1);
    MUX_R1 <= SUM when D00 ='0' else INP;   -- 重复使用 R1
    R1E <= D00 or D10 or D11;
    R1_Reg: block (R1E = '1' and CLK = '1' and CLK'Event)
    begin
        R1 <= guarded MUX_R1;   -- 值 R1
    end block R1_Reg;
    R2_Reg: block (D01 = '1' and CLK = '1' and CLK'Event)
    begin
        R2 <= guarded INP;   -- 值 R2
    end block R2_REG;
    R2C <= not R2;   -- 求反
    R2TC <= not R2 + "00000001";   --加 1
    MUX_ADD <= R2TC when D11 = '1' else R2;
    SUM <= R1+MUX_ADD;   -- 加法
end DF1;
```

事实上,对于同一硬件,还可以写出其他形式的数据流模型,下面的结构体 DF2 是该寄存器系统的另一种数据流模型。结构体 DF2 只用了两个保护模式赋值语句,源代码看起来非常简洁。但它不能清楚地表示连接关系,不适合于进行时序分析等项研究。

```
--  数据流模型 2
architecture DF2 of REG_SYS is
```

```
signal R1, R2: Std_logic_vector(0 to 7);
begin
R1_Reg: block ((CON(0) or not CON(1)) = '1' and CLK= '1' and CLK'Event)
begin
R1 <= guarded R1+R2  when (CON(0) and not CON(1)) = '1' else
R1+not(R2)+"00000001" when (CON(0) and CON(1)) = '1' else INP;
end block R1_Reg;

R2_ Reg: block ((not CON(0) and CON(1)) = '1' and CLK = '1' and CLK'
    Event)
begin
R2 <= guarded INP;
end block R2_REG;
end DF2;
```

在寄存器系统的第一种数据流模型 DF1 中，采用了集中译码方案，即先将输入指令 CON 译码为控制指令 D00,D01,D10 和 D11，再用译码后的信号控制系统的动作。可以对该模型作出改变，不采用集中译码方案，而采用分别译码的方案。一般地讲，集中译码方案要比分别译码方案好。首先，从功能划分角度来看，集中译码使得电路复杂性降低，电路及其模型直观易懂，有利于电路自动综合；其次，当需要给电路增加新功能时，若采用集中译码方案，只需对硬件作较小的改动就可以实现。但是，对于较简单的硬件，采用分别译码方案，电路规模会小一些。对于上述简单寄存器系统，由于电路简单，采用分别译码方案好处很多。结构体 DF3 是分别译码方案的数据流模型，图 5.1 实际上是这一分别译码电路的结构模型。

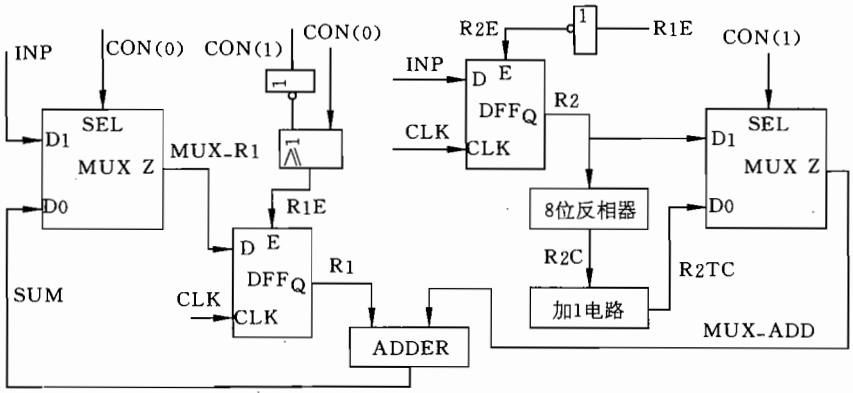


图 5.1　寄存器系统 Reg_Sys 的综合结果

```
-- 分别译码的数据流模型
architecture DF3 of REG_SYS is
```

```
    signal MUX_R1, R1, R2, R2C, R2TC, MUX_ADD, SUM, R1E, R2E:
                Std_logic_vector(7 downto 0);
begin
    MUX_R1 <= SUM when CON(0) = '1' else INP;   -- 复用 R1
    R1E <= CON(0) or not CON(1);
    R1_Reg: block (R1E = '1' and CLK= '1' and CLK'Event)
    begin
        R1 <= guarded MUX_R1;   -- 值 R1
    end block R1_Reg;
    R2E = not R1E;
    R2_Reg: block (R2E = '1' and CLK= '1' and CLK'Event)
    begin
        R2 <= guarded INP;   -- 值 R2
    end block R2_Reg;
    R2C <= not R2;   -- 求反
    R2TC <= R2C+"00000001";   -- 增加 1
    MUX_ADD <= R2TC when CON(1) = '1' else R2;   -- 加法器复用
    SUM <= R1+MUX_ADD;   -- 加法
end DF3;
```

5.2 控制单元设计

在 2.7节中已经介绍过,在集成电路设计时,通常可以将整个系统划分为两部分,一部分是数据单元,另一部分是控制单元,图 5.2 示意给出了这种划分。数据单元中包含有保存运算数据和运算结果的数据寄存器,也包括用来完成数据运算的组合逻辑电路单元。控制单元用来产生控制信号序列,以决定何时进行何种数据运算。控制单元要从数据单元得到条件信号,以决定继续进行哪些数据运算;数据单元要产生输出信号、数据运算状态等有用信息。

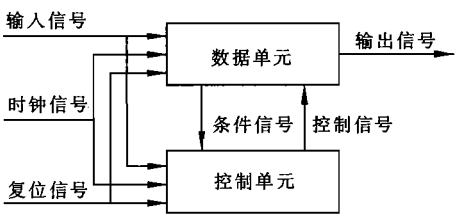


图 5.2 集成电路系统的划分

通常控制单元可以有两种电路形式:一种是微程序(微代码)控制单元,另一种是标准硬连接实现的控制单元,即硬连线式控制单元。硬件直接实现的控制单元一般用有限状态机实现,通常具有较高的运算速度;但是通用性差,对于每个电路,都必须专门设计控制单元,因此种类很多。图 5.3 是由 Huffman 有限状态机实现的控制单元,其中状态寄存器模块 MEM 中包含了所有控制触发器,模块 CL 中包含了控制单元中的所有组合逻辑电路。

一般情况下，控制信号的当前输出值与数据单元中来的状态信息有关，也与控制触发器的当前状态有关，即状态机为 Mealy 状态机；有些情况下，控制信号的当前输出值只与控制触发器的当前状态有关，即状态机为 Moore 状态机。控制触发器当前状态与其过去状态以及由数字单元来的状态信号有关。一般地讲，硬件直接实现的控制单元比微代码控制器工作速度快，但电路复杂，设计成本高，且对每种电路必须专门设计控制器，设计可再用性差。

微代码控制器如图 5.4 所示，它在每个单位时间从 ROM 中读取该时刻所有控制信号的值，把控制信号值读出后，寄存在指令寄存器 MIR 中，指令寄存器的输出端保持其数值直到下一单位时间。微代码控制器通过产生当前地址来依次得到各个时间单位的控制信号值，即通过产生地址序列得到控制信号序列。如果地址生成电路相对简单，这种微代码控制器应该是理想的电路设计方案。通常，地址生成电路是一个计数器，正常情况下地址序列生成是通过简单地加 1 得到下一地址，如果需要在控制序列中跳转到某点，则可以通过对寄存器加载一个特定值来实现。用这种方式产生地址序列，电路相当简单。

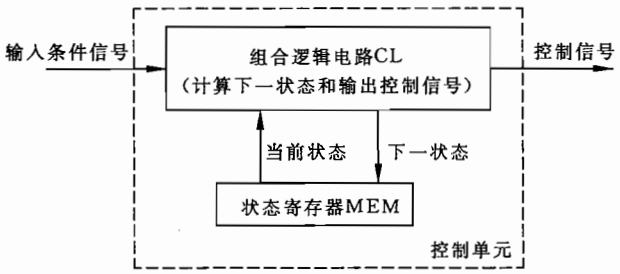


图 5.3　硬件控制器的 Huffman 模型

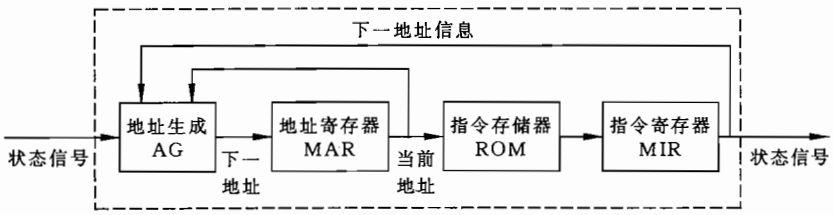


图 5.4　微代码控制器示意图

微代码控制器的主要优点如下：

(1) 对于同一类器件可以采用标准设计。同类器件中的不同控制器的不同之处只是存储在指令存储器 ROM 中的数据不同。因此，只要改变 ROM 中的数据，就可以得到不同的控制器，可以批量生产出电路的主体，对于不同应用，对 ROM 中加入不同的数据。

(2) 容易改变电路设计方案。实际上，只要改变 ROM 中的数据，而不必改变控制器本身，就可以改变控制器的功能。从这个意义上讲，微代码控制器的改变要比硬连接的控制器改变容易。

(3) 一旦微代码控制器的硬件电路设计、调试结束，对控制器的设计就简化为设计

ROM 中的数据。因此，对于一个具体应用，控制器设计就变得很简单。还可以利用成熟的软件工具，比如编译器、仿真器、调试工具等。ROM 数据出错的可能性很小，即使出现错误，也很容易发现，且改变 ROM 中的数据也比改变硬件本身容易。

(4) 在电路的控制结构极其复杂的情况下，硬连线控制电路的规模会变得无法忍受，这时只能使用微代码控制器，比如 Intel 的微处理器系列都采用了微码控制单元。

与硬件直接实现的控制器相比，微代码控制器的主要缺点如下：

(1) 由于执行速度受 ROM 读取速度的影响，所以工作速度相对较慢。

(2) 由于微代码控制器至少包括一定规模的 ROM 和两组寄存器，对于简单应用，成本相对较高。但是对于复杂设计，如果用硬连接实现控制器，可能需要很多触发器，这时应选用微代码控制器。

(3) 一旦选用了标准的微代码控制器，就限制了电路设计只能使用控制器中预先实现的功能。对于微代码控制器本身的设计，需要功能很完善，才能满足不同的需要。但功能越强，设计及制造成本会越高。对于较简单的应用，即使不使用这些功能，也要花费这些额外的代价。通常对控制器的限制包括有：可产生控制信号的个数，可处理状态信号的个数，地址产生电路的字长等。电路设计者需要综合考虑控制器的通用性、灵活性和设计成本代价、制造成本等因素。

通过对 5.3 节超级精简指令计算机(URISC)的设计，可以详细解释这些概念。

5.3 超级精简指令集计算机 URISC

RISC(reduced instruction set computer)的含义是精简指令计算机，这意味着计算机的指令集非常简单，所谓指令集简单是与复杂指令计算机(CISC)相对比而言。RISC 很适合于用超大规模集成电路实现，由于指令简单，所以译码复杂度低，同时可以针对各条指令对硬件进行优化，因此指令的执行速度很高，从而弥补了其指令个数少的缺欠。一旦决定了要设计一个小指令集的计算机，首先可以考虑的设计方案是利用超级 RISC 结构。所谓超级 RISC 结构，指的是 Mavaddat 和 Parham 提出的一种 RISC 结构，由于它只有一条指令，指令集不能再精简，所以称为超级精简指令计算机(URISC)。尽管 URISC 只有一条指令，它也是一种通用的计算机。所有复杂操作都可以用这一种指令完成。

URISC 指令是“做减运算，且在结果为负值时转移”。由于只有一条指令，所以不需要对指令定义操作码。指令中只需指出两个操作数且指出运算结果为负数时的转移地址。URISC 的指令形式如下：

第一个操作数的地址　第二个操作数的地址　运算结果为负时的转移地址

URISC 指令的执行过程如下：

(1) 从第二个操作数中减去第一个操作数，并把运算结果存储在第二个操作数的地址中；

(2) 如果减法运算得到的结果为负数，则转移到指定的地址继续执行，否则执行下面地址中的指令；

(3) 如果转移到地址 0,则停止 RISC 的运行。

按这种方式,每一条指令都是三字节指令,占用三个存储单元,需要三个存储器读周期才可以读出指令,因此只有在存储器的读取速度很高的前提下,URISC 才能有效工作。本章将详细讨论对存储器读取速度的要求。

下面以汇编语言的表示方法讨论如何用 URISC 实现指定的运算。汇编语言的一般格式为 L: F, S, T 其中,L 是语句标号,用来表示存储该条指令的符号地址,F 是第一个操作数的符号地址,S 是第二个操作数的符号地址,T 是减运算得到负结果时的跳转地址。假定在汇编语言中,伪指令 L: Word C 的作用是把常数 C 存储在符号地址 L。

采用上述伪指令,对表 5.1 所示的 URISC 程序进行讨论。这段程序实现的运算为:Z=(X+Y)/2。其中除法通过相继进行减法运算实现。运算结束后,跳转到地址 0,即停止程序的执行。

表 5.1 完成 Z = (X + Y) / 2 的 URISC 程序

地址	符 号	指 令	说 明
0	STOP:	Word 0	停止执行
1	READY:	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, NEXT3	TEMP2 <= −(X+Y)−2
6	COUNT_NEG:	ONE, Z, NEXT3	Z <= Z−1
7	NEXT3:	TWO, TEMP3, NEGATIVE	GOTO NEGATIVE
8	POSITIVE:	TWO, X, STOP	X <= (X+Y)−2
9	COUNT_POS:	MONE, Z, NEXT4	Z <= Z+1
10	NEXT4:	TWO, TEMP4, POSITIVE	GOTO 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	

5.3.1 URISC 处理器结构

本节讨论 URISC 处理器的结构,图 5.5 是 URISC 处理器中的数据单元。程序计数器 PC 保持下一单位时间要执行的指令所在的地址。数据寄存器 MDR 以及地址寄存器 MAR 用来提供 URISC 处理器与存储器之间的接口。寄存器 R 存储减运算的第一个操作数,减运算通过加一个数的补码来实现。加法器的第一个操作数总是从总线 BUS_A 得到,另一个操作数的来源有两种可能:当 COMP 为'1'时,加法器的第二个操作数由 R 中

的数值求反得到；当 COMP 为'0'时，加法器的第二个操作数是常数 0。加法器的输出总是连接到总线 BUS_B。加法器可能完成三种运算：如果 COMP='0'且 CIN='1'，则加法器完成加 1 运算，用来实现 PC+1；如果 COMP='1'且 CIN='1'，则加法器完成的是把第二个操作数与第一个操作数的补码相加，实现了减运算；如果 COMP='0'且 CIN='0'，由于这时是把常数 0 加到第二个操作数，其作用是把 BUS_A 上的数值直接传递到 BUS_B，这一功能用来把程序计数器 PC 的值或数据寄存器 MDR 的值传递到地址寄存器 MAR。加法器带有两个状态输出信号 Z 和 N，其作用分别是指明加运算得到了零结果或负结果。在设定了控制信号 ZIN 和 NIN 的情况下，这两个标志信号可以存入相应的状态触发器。

图 5.5 中用符号×给出了必要的控制点并标明每个控制点的控制信号。用 OUT 标出的控制信号（比如 PCOUT）的作用类似于寄存器的使能信号，用来控制相应的寄存器的输出与总线 BUS_A 的连通或切断。例如：当 PCOUT 为'1'时，程序计数器 PC 的数值送上总线 BUS_A，PCOUT 为'0'时，程序计数器 PC 的输出为高阻态。图 5.5 中用 IN 标出的控制信号（比如 MARIN）用来控制向寄存器加载数据。例如：如果 MARIN 为'1'，则总线 BUS_B 上的数据加载到地址寄存器 MAR。如果 ZIN 为'1'，则加法器运算产生的状态信号 Z 被加载到 Z 状态触发器。

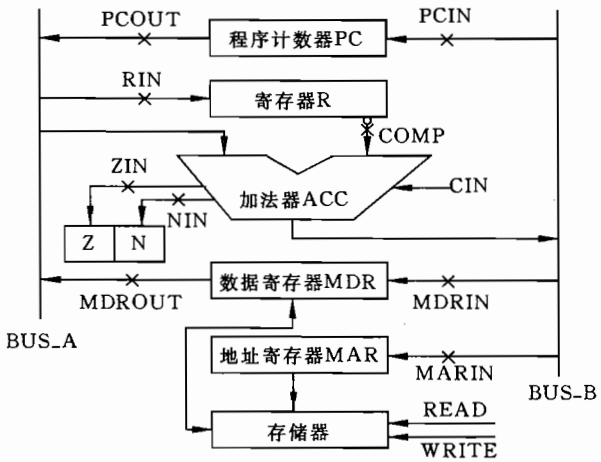


图 5.5　URISC 处理器的数据单元

通过对 PCOUT，CIN 和 PCIN 置不同的数值，就可以实现 PC+1。首先，设置控制信号 PCOUT，这个信号控制把程序计数器 PC 的值送上总线 BUS_A。其次设定 COMP='0'，CIN='1'，这样加法器的第一个操作数是总线 BUS_A 上的数据，第二个操作数是常数 0，由于 CIN='1'，加法器运算的结果是把 PC+1 的值送上总线 BUS_B。随着对控制信号 PCIN 置位，总线 BUS_B 上的数据被加载到程序计数器 PC。这样就完成了一个指令周期，实现了 PC+1。

从指令中的第二个操作数中减去第一个操作数的过程也很简单。首先把第一个操作数加载到寄存器 R 中，把第二个操作数加载到数据寄存器 MDR 中。然后对 MDROUT，CIN 和 COMP 置位，就可以把 MDR 的数据（第二个操作数）送到加法器的第一个输入

端，把 R 中数据（第一个操作数）的反码送到加法器的第二个输入端，由于 CIN='1'，加法器完成的是将第二个操作数与第一个操作数的补码相加，即实现了从第二个操作数中减去第一个操作数的运算。最后，通过对控制信号 MDRIN 置位，将加法器运算结果通过总线 BUS_B 送回到数据寄存器 MDR 中。

如果把控制信号 PCOUT 和 MARIN 置位，则可以把程序计数器 PC 的值通过路径 BUS_A→加法器→BUS_B 传递到地址寄存器 MAR。

5.3.2 URISC 处理器的控制

为了使 URISC 处理器正常工作，数据单元中的每一个控制信号必须在一定时刻按所实现运算的要求置位，而在另一时刻复位。控制单元的作用就是提供这些控制信号。图 5.6 示意给出了 URISC 处理器的方框图。这个图中包括了控制单元 CU、数据单元 DU、存储器 RAM 以及它们之间的控制信号。控制单元的输出就是图 5.5（数据单元）中的各个控制信号，除了图 5.5 中要求的信号之外，控制单元还产生读 READ、写 WRITE 信号用来控制存储器的读写。

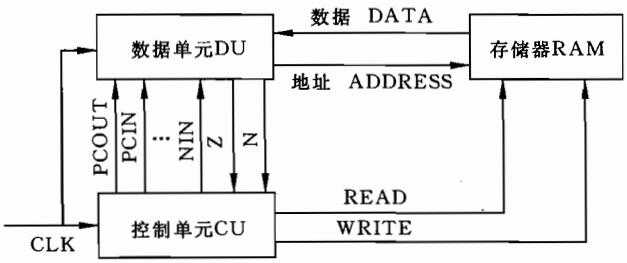


图 5.6 URISC 处理器的方框图

5.3.3 URISC 处理器的状态序列和指令周期

一个指令周期内的每一个时钟周期，构成 URISC 的一个指令状态。在每个指令状态，控制单元要给出各个控制信号的数值。表 5.2 列出了一个指令周期中的 8 个指令状态。这 8 个指令状态组成的状态序列把一个指令从存储器中取出，并执行这个指令。每个

表 5.2 URISC 处理器的一个指令周期

状态	需要置位的信号
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

状态中需要置位的控制信号全列在表 5.2 中，不需置位的控制信号则没有列出。这个状态序列可以正常工作的前提是程序计数器 PC 中保存有将要执行的指令的第一个字节的地址，且假定每个指令占用三个连续的存储单元。第一个操作数占用第一个存储单元，第二个操作数占用第二个存储单元，第二个操作数减去第一个操作数得到负值时的跳转地址占用第三个存储单元。在这个控制序列结束时，下一条指令的第一个字节的地址就驻留在程序计数器 PC 中。重复执行这个序列，可以执行完整的程序。

下面详细解释上述指令周期中的各个状态。

(1) 状态 0 ～ 状态 2：把第一个操作数放入寄存器 R。

在状态 0，先通过设置控制信号 PCOUT 和 MARIN 为'1'把指定地址加载到地址寄存器 MAR 中。由于 COMP 和 CIN 全为'0'，所以 PC 中的数值通过总线与常数 0 相加后传递到总线 BUS_B，MARIN 为'1'使得总线 BUS_B 上的数值加载到 MAR，即把当前指令的第一字节加载到地址寄存器 MAR 中。由于设定了控制信号 ZEND，如果 PC 的值为 0，则控制序列会在状态 0 无限执行，这时称为 URISC 处理器的动态挂起，以下讨论全假定 PC 的值不为 0。由于 READ 为'1'，所以在这一状态把 MAR 指出的地址的内容读入数据寄存器 MDR，即把第一个操作数的地址读到数据寄存器 MDR 中。

在状态 1，由于把控制信号 MDROUT 和 MARIN 设置为'1'，这时数据寄存器 MDR 中的数据(第一个操作数的地址)经过 BUS_A、加法器和 BUS_B 进入地址寄存器 MAR。由于这一状态仍设定了读控制信号 READ，所以此时把 MAR 指明的地址中的内容(第一个操作数)读入数据寄存器 MDR。

在状态 2，设定控制信号 MDROUT 和 RIN 为'1'，使得数据寄存器 MDR 中的数据(第一个操作数)进入寄存器 R。

(2) 状态 3 ～ 状态 5：读第二个操作数，从第二个操作数中减去第一个操作数后把结果存入原第二个操作数的地址。

在状态 3，读第二个操作数的地址。首先因为控制信号 PCOUT 和 CIN 为'1'，加法器对 PC 的值加 1。因为控制信号 PCIN 和 MARIN 为'1'，加法器运算的结果被同时送入程序计数器 PC 和地址寄存器 MAR。由于读控制信号 READ 为'1'，此时将 MAR 指明的地址的内容(第二个操作数的地址)读入数据寄存器 MDR。

在状态 4，由于控制信号 MDROUT 和 MARIN 为'1'，数据寄存器 MDR 中的内容(第二个操作数的地址)经过 BUS_A、加法器和 BUS_B 进入地址寄存器 MAR。由于这一状态仍设定了读控制信号 READ，所以此时把 MAR 指明的地址的内容(第二个操作数)读入数据寄存器 MDR。

在状态 5，控制信号 MDROUT、COMP 和 CIN 为'1'，使得加法器从第二个操作数中减去寄存器 R 中的内容(第一个操作数)。由于设定了控制信号 MDR，减法运算的结果被送回到数据寄存器 MDR。同时由于设定了控制信号 NIN，如果减法运算得到负结果，标志寄存器 N 会被置位。由于在状态 4 中为地址寄存器 MAR 设定的数值并没有被改变，所以写控制信号 WRITE 为'1'使得把数据寄存器中的内容(减运算的结果)重新写入原第二个操作数的地址。

(3) 状态 6:把减法运算得到负值时的转移地址读入数据寄存器 MDR。

由于设定了控制信号 PCOUT、CIN 和 PCIN,所以加法器将程序计数器 PC 的值加 1 后送回到 PC,即 PC 指向了当前指令的第三个字节。同时由于设定了控制信号 MARIN,所以 PC 的新值也被送入地址寄存器 MAR。读控制信号 READ 使得把当前指令的第三个字节读入数据寄存器 MDR。

(4) 状态 7:增加 PC,使它指向下一条指令的第一个字节。

由于设定了控制信号 PCOUT,CIN 和 PCIN,所以加法器将程序计数器 PC 的值加 1 后送回到 PC,即 PC 指向了下一指令的第一个字节。这时还可能出现两种情况:如果在状态 5 中做减法运算时没有产生负结果,即没有把标志寄存器 N 置位为'1',则由于这时设定了控制信号 NNEND,控制序列返回到状态 0,重新开始执行一条新指令。亦即减法运算得到非负结果时,URISC 处理器继续执行下一条指令。如果在状态 5 中设置了标志位 N,即减法运算得到了负结果,则转移到状态 8。

(5) 状态 8:把数据寄存器 MDR 中的数据复制到程序计数器 PC 之后控制单元转状态 0 重新执行一条新指令。

由于将控制信号 MDROUT 和 PCIN 设定为'1',数据寄存器 MDR 中的数据经 BUS_A、加法器和 BUS_B 被复制到 PC。然后,控制单元返回到状态 0。

5.3.4 URISC 处理器的时序

通过研究上述控制序列中描述的 URISC 指令执行过程,可以看到几乎每一个状态中都有两个相继的操作。比如:在状态 0,程序计数器 PC 的值必须先转移到地址寄存器 MAR,然后再读外部存储器,把存储器中的数据转移到数据寄存器 MDR。除此之外,还可以看到在每一个状态中,几乎都是先有一个 URISC 的数据单元内部的操作,然后再有一个存储器读/写操作。用两相互不交迭的时钟可以对这两个相继的操作进行控制,第一相时钟 PH1 控制数据在 URISC 数据单元内部的移动,第二相时钟 PH2 控制数据在 URISC 数据单元和存储器之间移动。图 5.7 定义了 URISC 处理器的时序,在控制序列的开始处,时钟 PH1 的下降沿更新控制信号,控制信号保持为常数直到下一个时钟 PH1 的下降沿。时钟 PH2 的下降沿用来控制数据在 URISC 数据单元内传递。在每个控制状态的结尾处,PH2 的下降沿控制产生存储器读/写操作,并把控制信号更新为下一状态的控制信号。图 5.8 示意给出了前 5 个控制状态的时序关系。

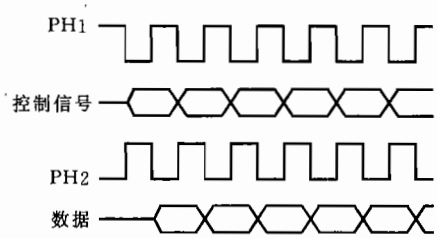


图 5.7 URISC 处理器的两相互不交迭时钟

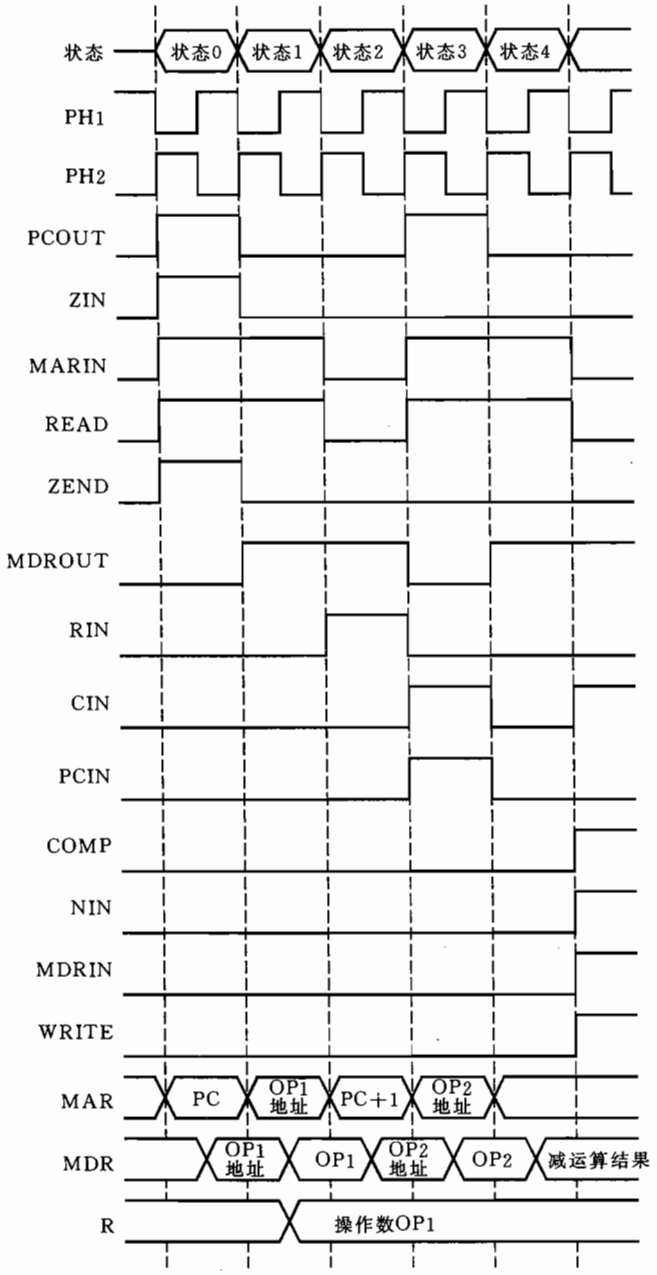


图 5.8　控制状态 0、状态 1、状态 2、状态 3 和状态 4 中的控制信号

5.3.5　URISC 系统

图 5.9 给出了一个采用 URISC 处理器的计算机系统，它包括有一个随机存取存储器 RAM 和输入/输出端口模块 PORT。用信号 RUN 控制处理器的启动和停止。信号

DATA 和 ADDRESS 分别是外部数据总线和地址总线。信号 READ 指定数据从 RAM 向 URISC 处理器传送。信号 WRITE 指定数据从 URISC 处理器向 RAM 传送。信号 DAV 由模块 PORT 给出，如果模块 PORT 接收到一个新数据，且需要 URISC 处理器对该数据进行处理，则模块 PORT 为该信号置位。当 URISC 处理器准备好处理数据时，会为信号 RDIO 置位。模块 PORT 在接收到该信号后，才能把数据送上总线 DATA。

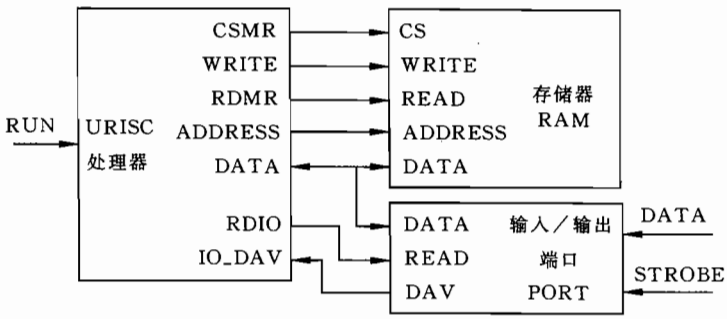


图 5.9　采用 URISC 处理器的计算机系统示意

5.3.6　在寄存器级设计 URISC 处理器

下面给出的 VHDL 源代码是 URISC 处理器的框架，这里仍使用了程序包 IEEE.std_logic_1164。

```
-- URISC 处理器模型的框架
---------- URISC 处理器的实体说明----------
use IEEE. Std_logic_1164. all;
use IEEE. std_logic_arith. 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 Std_logic_vector(7 downto 0) := "ZZZZZZZZ";
         ADDRESS: inout Std_logic_vector(7 downto 0) := "00000001";
         RUN, IO_DAV : in Std_logic;
         RDIO, WRITE: inout Std_logic;
         RDMR, CSMR: out Std_logic);
end URISC;
---------- URISC 处理器的结构体框架----------
architecture Behavior of URISC is
    signal PC1, R, R_NOT, BUS_A, BUS_B, MDR1, MDR:
           Std_logic_vector (7 downto 0);
```

```
    signal PC: Std_logic_vector(7 downto 0) := "00000001";
    signal Z, ZERO, ZIN, N, PH1, PH2, R_IN, MDR_IN, N_IN,
        MAR_IN, C_IN, CLK: Std_logic;
    signal CLEAR, COMP, MDR_OUT, PC_IN, PC_OUT, ZEND,NNEND,
    READ, Z_OUT,
        N_OUT: Std_logic;
    signal C: Std_logic_vector(3 down to 0);

    procedure ADD(A, B: in Std_logic_vector; CIN: in Std_logic ;
        SUM: out Std_logic_vector; Z_OUT, N_OUT: out Std_logic) is
    variable SUMV, AV, BV: Std_logic_vector(A'Length - 1 downto 0);
    variable CARRY: Std_logic;
    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;
      SUM := SUMV;
      Z_OUT <= not (SUMV(0) or SUMV(1) or SUMV(2) or SUMV(3) or
                    SUMV(4) or SUMV(5) or SUMV(6) or SUMV(7));
      N_OUT <= SUMV(7);
    end ADD;
begin
----------程序计数器 PC----------
    PC_Reg: block(PC_IN= '1' and PH2= '0' and PH2'Event)
    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;
----------寄存器 R----------
    R_Reg: block(R_IN= '1' and PH2= '0' and PH2'Event)
    begin
        R <= guarded BUS_A after REG_DEL;
```

```
        R_NOT <= not R when COMP= '1' after INV_DEL else
          "ZZZZZZZZ" after INV_DEL;
    end block R_Reg;
---------- 加法器 ----------
    ADD(BUS_A, R_NOT, C_IN, BUS_B, Z_OUT, N_OUT) after ADD_
    DEL;
---------- 寄存器 N ----------
    N_Reg: block(N_IN= '1' and PH2= '0' and PH2'Event)
    begin
        N <= guarded N_OUT after REG_DEL else
                'Z' after REG_DEL;
    end block N_Reg;
---------- 寄存器 Z ----------
    N_Reg: block(Z_IN= '1' and PH2= '0' and PH2'Event)
    begin
        Z <= guarded Z_OUT after REG_DEL else
                'Z' after REG_DEL;
    end block Z_Reg;
---------- 数据寄存器 MDR ----------
    MDR_Reg: block((MDR_IN= '1' and PH2= '0' and PH2'Event) or
        (READ and PH1= '0' and PH1'Event) )
        ----
        ----
    end block MDR_Reg;
---------- 地址寄存器 MAR----------
    MAR_Reg: block(MAR_IN= '1' and PH2= '0' and PH2'Event)
       ----
        ----
    end block MAR_Reg;
---------- 内部两相时钟 ----------
    process
        ----
        ----
    end process;
---------- 控制单元 ----------
    process
        ----
        ----
```

```
    end process;
end BEHAVIORAL;
```

实体定义中大部分类属参数是 URISC 中要用到的延时参数。比如：ENABLE_DELAY 是 DATA_BUS 的使能信号的延时，PER 是时钟周期。信号 DATA 是连接 RAM 和输入端口的外部数据总线，信号 ADDRESS 是连接 RAM 的外部地址总线。

结构体说明中包含有用于 URISC 处理器内部的信号，有些内部信号已为读者所熟悉，比如 COMP，ZEND 等；另外还有一些新定义的信号。结构体对每个寄存器全用一个模块(block)表示，对于 URISC 中的每个组合逻辑单元也用一个模块表示，控制单元单独是一个模块。加法器模块中使用了过程 ADD，该函数计算两个二进制数的加法，并带有进位输入端，除了生成运算结果之外，该过程还计算和数是否为 0(通过求 8 位的或非)以及和数是否是负数(最高位为 1)。

每个寄存器全设计成负边沿触发的寄存器。比如在上述框架中程序计数器 PC，它由一个受保护模块描述其保护条件，在时钟 PH2 由'1'变为'0'时，保护条件由 PCIN 激活。在保护条件为真时，总线 BUS_B 上的数值加载到程序计数器 PC。当 PCOUT 为'1'时，程序计数器 PC 的值送上总线 BUS_A。PC 的值送上 BUS_A，不受保护条件的影响。如果信号的值 PCOUT 不为'1'，则程序计数器 PC 的输出为高阻态。其他寄存器的设计方法与 PC 类似。

5.3.7 URISC 处理器中的微代码控制器

由于 URISC 处理器的指令执行过程只有两个分支点，且两个分支点的转移入口全为控制状态 0，所以微代码控制器的地址生成逻辑电路十分简单，可以用一个九进制计数器实现地址生成。计数器应该具备同步复位功能，以便在满足分支条件时返回状态 0。如果不满足分支条件，则计数器会从 0 计到 8 且会一直循环下去。如果有一个分支条件满足，则计数器复位为 0，即控制返回状态 0。下面是这一控制器的 VHDL 模型，这一模型可以直接转换成图 5.4 所示的硬件结构。

```
-- 微代码控制器的 VHDL 模型
---------- 计数器 COUNTER ----------
--- The Counter with synchronous Clear
Counter: block (PH1 = '0' and PH1'Event)
begin
    C <= guarded "0000" after COUNT_DEL when CLEAR = '1' else
         C + "0001" after COUNT_DEL;
end block Counter;
---------- 微代码只读存储器 ROM ----------
ROM: process(C)
    type SQ_ARRAY is array(0 to 8, 0 to 8) of Std_logic;
    constant MEM: SQ_ARRAY :=
```

```
    -- MDROUT  MARIN  NIN  RIN  PCIN  ZEND  CIN  WRITE  NNEND
       (('1',    '1',  '0', '0', '0',  '1',  '0', '0',   '0'),
        ('1',    '1',  '0', '0', '0',  '0',  '0', '0',   '0'),
        ('1',    '0',  '0', '1'; '0',  '0',  '0', '0',   '0'),
        ('0',    '1',  '0', '0', '1',  '0',  '1', '0',   '0'),
        ('1',    '1',  '0', '0', '0',  '0',  '0', '0',   '0'),
        ('1',    '0',  '1', '0', '0',  '0',  '1', '1',   '0'),
        ('0',    '1',  '0', '0', '1',  '0',  '1', '0',   '0'),
        ('0',    '0',  '0', '0', '1',  '0',  '1', '0',   '1'),
        ('1',    '0',  '0', '0', '1',  '0',  '0', '0',   '0'));
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
    Z_IN <= ZEND;
    ZERO <= NOR_Std_logicS(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;
    MDMR <= READ and not (ADDRESS(7)) after AND_DEL;
    RDIO <= READ and (ADDRESS(7)) after AND_DEL;
    CSMR <= not (ADDRESS(7)) after INV_DEL;
end block Logic;
```

在时钟信号从'1'变为'0'的时刻,计数器被加载为"0000"。信号 CLEAR 用来定义分支

返回状态 0 的条件。CLEAR 为'1'的第一个条件是信号 Z 和信号 ZEND 的与为'1',其意义是在状态 0 时检测到程序计数器 PC 的值为'0',这时计数器 COUNTER 应该复位为'0',控制状态在状态 0 无限循环,以实现 URISC 处理器的动态挂起。CLEAR 为'1'的第二个条件是 not N and NNEND,用来指明减法运算得到非负结果,计数器应该在状态 7 时终止,这时应该将计数器 COUNTER 复位为 0,返回状态 0 去执行程序存储器中的下一条指令。反之,如果 N 为'1',则应该进入状态 8,为程序计数器加载跳转地址,经过状态 8 后,计数器也会复位为'0',这是 CLEAR 为'1'的第三个条件,即 C="1000",转移到跳转地址后,应该执行跳转地址的下一条指令。对于 CLEAR 为'0'的其他所有条件,COUNTER 全增加 1,控制进入下一个状态。

计数器 COUNTER 的输出是状态信号,只是只读存储器 ROM 的地址输入。每当 COUNTER 的值发生变化时,各控制信号的数值从 ROM 中被读出。在这个设计方案中,控制信号是从 ROM 中直接读出,而不是像图 5.4 中那样,把 ROM 中的数据加载到指令寄存器 MIR。只有当 ROM 的字长足够长时,才可能使用这样的方案,一般情况下,应该使用指令寄存器。

为了减小 ROM 的字长,可以研究各控制信号的相关性。通过对各控制信号出现的时刻可以看出,各控制信号并不独立。比如,控制信号 ZIN 和 ZEND 总是同时出现,因此可以用 ROM 中的同一位数据表示这两个控制信号。控制单元的源代码中的模块 Logic 中还指出了其他控制信号的相关性。

5.3.8 URISC 处理器的硬连接控制器

硬连接控制器设计远没有微代码控制器那么规则,它要根据需要专门设计。下面的 VHDL 源代码给出了一种控制硬件的寄存器级模型。更规则的方法应该对每个控制状态使用单独的一个触发器。

```
-- URISC 的硬连线式控制单元的 VHDL 模型
---------- 计数器 COUNTER ----------
--- The COUNTER has a synchronous CLEAR
COUNTER: block (PH1 = '0' and PH1'Event)
begin
    C <= guarded "0000" after COUNT_DEL when (CLEAR = '1' or C="1000")
         else INC_COUNTER(C) after COUNT_DEL;
end block COUNTER;
---------- 硬件连接的控制单元 ----------
---------- 译码器
---------- 第一级译码
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;
---------- 第二级译码
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;
---------- 各控制信号
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;
ZEND <= ST0;
NNEND <= ST7;
---------- 计数器和各寄存器控制
ZERO <= NOR_Std_logicS(BUS_B) after OR_DEL;
CLEAR <= (Z and ZEND) or (not N and NNEND) after AND_DEL+OR_DEL;
RDMR <= READ and not (ADDRESS(7)) after AND_DEL;
RDIO <= READ and (ADDRESS(7)) after AND_DEL;
CSMR <= not (ADDRESS(7)) after INV_DEL;
```

分析上面的代码，可以看到，实际上是用计数器实现了有限状态机，这是常用的设计技巧。状态信号 STn 是对计数器输出译码得到的，而控制信号则是通过对状态信号译码形成。这种风格的代码几乎是直接描述电路，因此在电路相对简单或者是设计师具备该电路的特殊设计知识时具有较好的综合效果。否则，就应该使用有限状态机式的描述风格。

5.4 80C51 兼容微控制器的寄存器级设计

在以上 URISC 的例子中，由于电路比较简单，采用了 block 语句表示电路模块，但是如果设计比较复杂，通常把电路分解为寄存器级模块，然后依次设计实现。在第 4 章系统级设计时，已经介绍了系统划分，这里继续介绍寄存器级单元的设计。

5.4.1 时钟系统的设计

微处理器和微控制器的基本时钟一般以单频方波信号的形式提供，有三种产生方式：

(1) 用晶体振荡器产生精确而稳定的时钟信号；

(2) 用电压控制振荡器产生可调频率范围较宽的时钟信号；

(3) 结合以上两种技术，用压控晶体振荡器生成时钟信号。

由于此次设计属于常规芯片设计，对时钟没有特殊要求，所以采用外接晶振方案。在 80C51 体系定义中，同时提供了片上晶振。由于设计工具的限制，并且片上晶振精度较低，因此没有在设计中提供相应功能。

在基本时钟方案方面，微处理器和微控制器一般有三种选择：单相时钟、多相时钟和沿触发方案。在当前的设计中，沿触发方案由于在数据传递方面有一定困难已很少被使用。单相时钟因为在时序和传输上比较简单可靠，被一些高性能芯片使用，如 DEC 公司的 Alpha 21164 微处理器。但是对于 CMOS 电路来说，如果采用多相时钟就无法使用动态电路，而且设计将引入过多的双边约束条件。对于多相时钟，则可以消除这些双边约束，而使其转换为单边约束。同时，相数过多又会使设计复杂度提高，因此，这里选择了两相不重叠时钟。

根据 80C51 体系的说明，指令的执行分 1，2 和 4 周期三种。把 1 周期指令的执行时间称为一个机器周期，进一步划分为 6 个状态，每一状态有两相时钟，即为两个节拍。

两相不重叠时钟是时钟系统的核心，在工作频率较低的情况下，两相时钟可以通过综合产生，即用 VHDL 语言描述对输入晶振二分频的电路，并进行综合。然而在工作频率很高的情况下，如各种高性能微处理器芯片，只能采用手工输入电路图甚至物理版图的方式生成。

下面的结构体 A 和 B 列出了两种形成两相时钟的 VHDL 模型，图 5.10 是相应的信号波形，其中输入都是晶振方波信号。

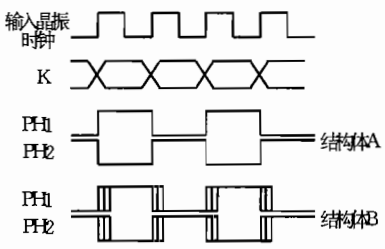


图 5.10 两相时钟电路的波形

```
-- 实现两相时钟的结构体 A
entity Clock is
    port(RESET : in Bit;
         CLK : in Bit;      --输入晶振时钟信号
```

```
        PH1, PH2 : out Bit);
end Clock;
architecture A of Clock is
begin
    P1: process(RESET, CLK)
        variable K : Integer range 0 to 1;
    begin
        if RESET= '1' then
            K := '0';
        elsif CLK= '1' and CLK'Event then
            case K is
                when 0 => PH1 <= '1'; PH2 <= '0';
                          K := 1;
                when 1 => PH1 <= '0'; PH2 <= '1';
                          K := K-1;
            end case;
        end if;
    end process P1;
end A;
-- 实现两相时钟的结构体 B
architecture B of Clock is
    signal K : Integer range 0 to 1;
begin
    PK: process(RESET, CLK)
    begin
        if RESET= '1' then
            K <= 0;
        elsif CLK= '1' and CLK'Event then
            K <= K -1;
        end if;
    end process PK;

        P1: process(K)
        begin
            case K is
                when 0 => PH1 <= '1'; PH2 <= '0';
                when 1 => PH1 <= '0'; PH2 <= '1';
            end case;
```

```
        end process P1;
end B;
```

上面的代码实际上是描述多相时钟的典型 VHDL 模型，因此对 K 的更新采用了把 K－1 赋予 K 的形式，在时钟相数高于 2 时，这种描述方式更为方便。结构体 A 和 B 都使用整数类型 K 计数，隐含实现了有限状态机。但其不同之处在于，在 A 中 K 及 PH1 和 PH2 由触发器锁存，而 B 中 PH1 和 PH2 不必锁存，从而节省了两个触发器。因此，结构体 A 在硬件实现比较昂贵。但是，结构体 B 存在着另外的问题，特别是在相数高于 2 的情况，K 被编码为矢量，在时钟沿上，K 发生变化，比如从"100"变为"011"，由于 PH1 和 PH2 由组合逻辑产生，也就是说，K 的变化直接送到输出，则很容易产生毛刺电平，对于时钟电路，这是很不可取的。因此尽管结构体 B 需要硬件较少，但是性能也较差，使用时需要特别小心。

产生两相时钟后，可以由两相时钟生成状态信号 S1～S6 和周期控制信号 T1，T2 和 T4，分别对应于 6 个状态和三种指令周期。在本节的其他篇幅内，采用 TSF 时刻表示执行时序，如 TSF＝121 表示在第一指令周期、第二个状态的第一相时钟有效时刻。

5.4.2 寄存器的设计

MCS51 体系中含有若干用户可见的寄存器，此外为实现总体功能还需要一些用户不可见寄存器，这些寄存器是微控制器中数据存储和处理的最小单元(即使是位操作指令，也仍要把 8 位寄存器内容全部读入 ALU，但通过一定的算法使得只有被操作位的内容改变)，它们的 VHDL 描述将作为综合器的输入代码，所以如何用 VHDL 对寄存器进行精确的造型具有非常重要的意义。下面给出的是 ACC 寄存器的代码。其他的寄存器可能具有特定功能，但基本结构是类似的。

```
--  ACC 寄存器的 VHDL 描述
entity ACC_Reg is
   port(Rst : in Bit;
     F1,F2 : in Std_logic;                    --时序信号
     T1,SS2,SS5 : in Std_logic;               --时序信号
     OP_RAM_DECODE : in Std_logic;
     OP_RAM_WR,OP_RAM_RD : in Std_logic;
     -----other ports
     DBUS : inout std_logic_vector(7 downto 0);
     ACC_Q : buffer std_logic_vector(7 downto 0));
end ACC_Reg;

architecture Dataflow of ACC_Reg is
     -----signal declaration
begin
```

```
——第一部分:控制信号的生成
begin
    DEC_CLK <= F1 and OP_RAM_DECODE;
    Adrs_Check : process
    begin
        wait until DEC_CLK= '1' and DEC_CLK'Event;
        L_A_ADRS <= DBUS(7) and DBUS(6) and DBUS(5) and (not DBUS
        (4))
and (not DBUS(3)) and (not DBUS(2)) and (not DBUS(1))
and (not DBUS(0));      ——检查地址 E0H 和位操作
end process;

    A_TOBUS_Ctrl : process
    begin
        wait until F2= '1' and F2'Event;
        L_A_TOBUS <= T1 and (SS2 or (SS5 and (OP_MUL or OP_DIV)));
    end process;

    L_A_SELECT <= OP_RAM_DECODE or (not L_A_ADRS);
    L_A_TOBUS <= OP_RAM_WR and (not L_A_SELECT);
——第二部分:装载 ACC 寄存器
    Update : process(RST,MU_BUStoACC,MU_SWAP,MU_XCHD,DBUS)
    begin
        if RST='1' then
            ACC_Q <= "00000000";
        elsif MU_BUStoACC= '0' then
            ACC_Q <= DBUS;
        elsif MU_SWAP= '0' then
            ACC_Q <= ACC_Q(3 downto 0) & ACC_Q(7 downto 4);
        elsif MU_XCHD= '0' then
            ACC_Q<=ACC_Q(7 downto 4) & DBUS(3 downto 0);
        end if;
    end process;
——第三部分:向总线输出
        Output : process(MU_ACCTOBUS,ACC_Q)
        begin
            if MU_ACCTOBUS= '0' then
```

```
            DBUS <= ACC_Q;
        else
            DBUS <= "ZZZZZZZZ";      --其余情况下释放总线
    end if;
    end process;
end Dataflow;
```

根据 MCS51 指令集，ACC 寄存器要完成以下功能：

(1) 与总线交换数据(双向)；

(2) 执行 XCHD 指令时，与总线只交换低 4 位数据；

(3) 在执行 SWAP 指令时，把高 4 位与低 4 位数据交换；

(4) 识别地址 E0H。

分析这些功能可以看到，ACC 在本质上是一个锁存器及其控制逻辑，并有多路数据输入。上面给出的代码可以分为三部分：第一部分生成控制 ACC 各种操作的控制信号，其核心思想是把各种控制信号经过运算后形成直接控制锁存器的微控制信号，在 F1 时钟节拍上有效(低电平有效)。这部分的两个进程中，用 wait until 语句来等待时钟上升沿激活进程，这种写法要比用 if DEC_CLK ='1' and DEC_CLK'Event then 并把 DEC_CLK 加入进程的敏感信号表的方法具有更高的效率，因为在后者的情况下，每一次 DEC_CLK 改变状态进程都要被激活一次，但实际上只需要在时钟的上升沿激活进程进行相应操作。而同时激活的进程越多，进行仿真的 CPU 的负担就越重，使得仿真效率下降。第二部分的进程 Update 描述了一个 8 位的电平锁存器，它根据当前有效的控制信号锁存相应数据，在所有控制信号均处于无效状态时，由于代码没有明确指定硬件行为，隐含表示锁存器保持原来状态，综合工具将自动用电平锁存器来实现这个进程；虽然就总体而言，芯片是同步系统，但电路中不可避免地存在一些异步操作，如上电和复位，上面的代码中 RST 信号(与其他控制信号不同，RST 在任何时刻都可能出现有效电平，而其他控制信号是结合时钟生成的同步信号)有效时将锁存器清 0，综合器生成的是异步清零的电平锁存器。第三部分是 ACC 寄存器与总线的接口，实际上是通过一个三态缓冲器与总线相连接，也就是说在 ACC 寄存器不驱动总线时将缓冲器输出端置为高阻态。总线信号 DBUS 被定义为 Std_logic_vector 类型的决断信号，这种信号在仿真时可以被多个源驱动，其有效值为各个源信号值进行“线或”运算的结果，这段代码使得在 ACC 寄存器不驱动总线时，其他模块可以将数据送上总线。在代码用于综合时，因为在代码中指定了高阻态，综合工具自动选用三态缓冲器实现这个进程。

VHDL 的功能是很强大的，许多硬件结构可以通过一定的语言结构让综合器自行生成，而不必显式地进行元件例化。比如在上面的代码中，并没有出现例化锁存器和三态缓冲器的语句，但仍然可以在综合后形成这些电路。这种方法使程序更加简洁、易读，易于与其他代码融合在一起，而且不必依赖于特定的 EDA 工具和集成电路工艺。

5.4.3 算术逻辑单元

1. 算术逻辑单元 ALU 的算法

算术逻辑单元 ALU 是数据路径的核心，几乎所有指令都要在此执行。因此，ALU 是

电路设计的重点之一。实现 ALU 有很多种算法，不同的算法可以构造出不同性能和代价的 ALU。在本次设计中，强调的是低成本、低代价，所以没有采用复杂的专用运算线路，而是采用 4 选 1 多路选择器作为布尔函数生成电路完成逻辑操作，用曼彻斯特进位链传递进位，然后异或生成和。8 个结构相同的 ALU 位单元互连构成 ALU。位单元的框图如图 5.11。

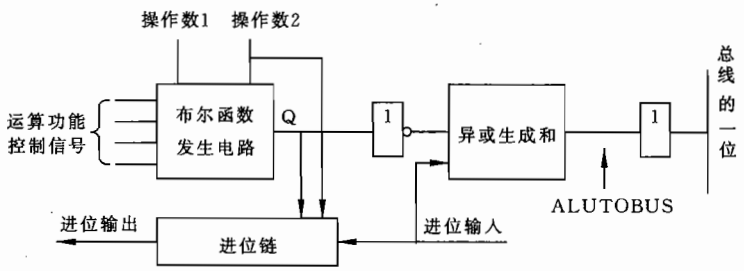


图 5.11　ALU 位单元的原理图

一个二输入、一输出的组合逻辑电路可以看作一个布尔函数：

$$Y = F(X0,X1)$$

其中输出 Y 和输入 X0,X1 取二进制数值。用真值表表示这个函数的输入、输出对应关系，如表 5.3。由逻辑代数原理可知，两变量布尔函数共有 16 种。在表 5.3 中，函数 F0 是全零函数，即无论两个变量 X0,X1 为何值，F0 的值总为零；函数 F0 则实现了两个变量 X0,X1 的异或运算。如果将(0,1,1,0)送入一个 4 选 1 电路的数据输入端 D0～D3，则 4 选 1 电路成为两个数据输入端的异或门。因此，原来 4 选 1 电路的数据选择线可以看作控制信号，而把选择控制信号看作数据信号，这样就实现了可变功能的逻辑门。根据指令的需要，将适当的 Y 值传送给 4 个输入，就可以随时改变其逻辑功能。

表 5.3　16 个两变量布尔函数

X0	X1	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
函数		0	与		X1		X0	异或	或	或非	异或非	X0 非		X1 非		与非	1

以上的布尔函数可以完成各种逻辑运算，但是对于算术运算还是不够的。最基本的算术运算是加法，减法可以看作是加法的特例，乘除法则由多次的加法和移位来实现。所以，ALU 只要实现加法功能即可。加法的算法采用了传统的异或生成和的算法，令 OP1 和 OP2 为两个操作数的对应位，CARRYIN 为进位输入，有

$$HALF = OP1 \text{ xor } OP2 \qquad (1)$$

SUM = HALF xor CARRYIN (2)

CARRYOUT = (CARRYIN and HALF) or (OP1 and OP2) (3)

其中,HALF 是半加和,SUM 是全加和,CARRYOUT 是进位输出。从进位的角度重新排列个式子,可得

KILL = (not OP1) and (not OP2) (4)

PROPAGATE = HALF (5)

PRODUCE = OP1 and OP2 (6)

KILL 为 1 时表示不会向高位产生进位,PROPAGATE 为 1 时表示向高位传递本位的进位输入,PRODUCE 为 1 则表示无论进位输入为何值,本位都会产生进位。根据(1)～(3)式,可以得到异或生成和电路的算法;(4)～(6)式就是所谓的曼彻斯特进位链结构。此外,由于逻辑运算不需要进位,所以在逻辑运算指令时,要关闭进位链。

ALU 的延时主要来自于进位链的传输延时。曼彻斯特进位链结构具有电路简单的优点,但是速度较慢,只能应用于数据宽度较小的场合。经过时序分析,发现用 CMOS 0.6 μm 两层金属工艺,整个 ALU 的加法运算的延时最坏情况在 15 ns 以内,可以满足 25 MHz工作时钟的要求。

2. 算术逻辑单元 ALU 的实现

根据本节 1 中的 ALU 算法,在这里实现这一电路。8 位 ALU 由 8 个结构完全相同的位单元和控制电路构成,可以对两个 8 位操作数进行加、减、与、或、异或等运算,此外还可以把其中一个操作数取反或清零。ALU 是由三个层次的模块组成的,在此给出各个层次的 VHDL 描述,并以此为例来说明模块划分和综合的一些问题。ALU 的 VHDL 描述如下:

```
-- ALU 的 VHDL 描述
entity ALU is
   port(TMP1,TMP2: in Std_logic_vector(7 downto 0);
        F1,F2: in Std_logic;
        OP_ALUTOBUS : in Std_logic;
        -- 其他控制端口
        CY: in Std_logic;
        ALU_Q: buffer Std_logic_vector(7 downto 0);
        DBUS: out Std_logic_vector(7 downto 0));
end ALU;

architecture behavior of ALU is
    --信号定义
    component ALU_Bit
          port(OP1,OP2 : in Std_logic;
       Y: in Std_logic_vector(3 downto 0);
```

```
        CIN: in Std_logic;
        L_XAOMD : in Std_logic;
        SUM: out Std_logic;
        COUT: buffer Std_logic);
    end component;
begin
    -- ALU 控制信号
    ALU_Gen : for I in 0 to 7 generate
        I0 : if I=0 generate
            ALU0 : ALU_Bit port map(TMP1(0),TMP2(0),Y,L_REND,
                MU_XAOMD,SUM(0),ALU_CARRY(0));
        end generate I0;
        I1_7 : if I>0 generate
            ALU1_7 : ALU_Bit port map(TMP1(I),TMP2(I),Y,ALU_
                CARRY(I-1),
                MU_XAOMD,SUM(I),ALU_CARRY(I));
        end generate I1_7;
    end generate ALU_Gen;

    Ctrl_Gen : ALU_Ctrl port map(……); --控制信号生成电路的例化

    Output : process(MU_ALUTOBUS,ALU_Q)
        begin
            if MU_ALUTOBUS= '0' then
            DBUS <= ALU_Q;
        else
            DBUS <= "ZZZZZZZZ";
        end if;
        end process;
end behavior;
```

这段程序中使用生成语句(generate)对元件进行例化,采用这种办法可以比较灵活地控制各个子单元之间的连接关系。综合工具在处理生成语句时,通常就是简单地在网单中加入一系列相应的用户定义的子模块。有些综合工具如 Synopsys 可以把子模块展平(flatten),把全部电路打散进行优化。但是对于 ALU 这类结构规则的电路,展平的优化策略通常并不一定有效,有时甚至反而增大芯片面积或在综合的迭代过程中出现死循环。使用生成语句,在综合上的效果相当于强制综合工具重复使用相应的子模块。在这一点上,生成语句不如 for 循环语句灵活。在后者情况下,综合结果可以是多次操作共享一个运算单元,也可以是并行使用多个单元,视时钟周期和设计要求而定。

```
-- ALU 位单元的 VHDL 描述
entity ALU_Bit is
    port(OP1,OP2 : in Std_logic;
     Y : in Std_logic_vector(3 downto 0);
     CIN : IN Std_logic;
     L_XAOMD : in Std_logic;
     SUM : out Std_logic;
     COUT : buffer Std_logic);
end ALU_Bit;

architecture Dataflow of ALU_Bit is
    component MUX4_1
            port(A,B : in  Std_logic;
              Y : in  Std_logic_Vector(3 downto 0);
              F : out Std_logic);
    end component;
    component Chain
       port(CIN : in Std_logic;
         KILL,PROPAGATE : in Std_logic;
         L_XAOMD : in Std_logic;
         COUT : buffer Std_logic);
    end component;

    signal HALF,Q : Std_logic;
    signal KILL : Std_logic;
begin
    Bool_Func : MUX4_1 port map(OP1,OP2,Y,Q);
    Carry_Chain: Chain port map(CIN,KILL,HALF,L_XAOMD,COUT);

    HALF <= not Q;
    KILL <= (not OP2) and (not HALF);
    SUM <= HALF xor CIN;
end Dataflow;
```

上面给出的是 ALU 位单元的 VHDL 描述。电路采用比较经典的结构，其中，各种函数的运算用 4 选 1 多路选择器实现，进位链采用曼彻斯特结构。下面是多路选择器和进位链的 VHDL 描述

```
--4 选 1 多路选择器
```

```
entity MUX4_1 is
        port(A,B : in  Std_logic;
         Y : in  Std_logic_Vector(3 downto 0);
         F : out Std_logic);
end MUX4_1;
architecture behavior of MUX4_1 is
begin
     process(A,B,Y)
         variable SEL : Std_logic_vector(1 downto 0);
     begin
         SEL := Std_logic_vector(1 downto 0)--(B & A);
        case SEL is
             when "00" => F <= Y(0);
             when "01" => F <= Y(1);
             when "10" => F <= Y(2);
             when "11" => F <= Y(3);
             when others => Null;
        end case;
     end process;
end behavior;

--曼彻斯特进位链
entity Chain is
   port(CIN : in Std_logic;
     KILL,PROPAGATE : in Std_logic;
     L_XAOMD : in Std_logic;
     COUT : buffer Std_logic);
end Chain;
architecture Dataflow of Chain is
begin
     Update : process(CIN,Kill,Propagate,L_XAOMD)
     begin
         if L_XAOMD= '1' then
             COUT <= '1';
         else
             if KILL= '1' then
                 COUT <= '0';
             elsif PROPAGATE='1' then
```

```
                COUT <= CIN;
            else
                COUT <= '1';
            end if;
        end if;
    end process;
end Dataflow;
```

在4选1电路中，用case分支语句进行选择，综合时硬件上正好对应于多路选择器，这是很直观的方法。由于Std_logic和Std_logic_vector都是多值信号，即除'0','1'外还可能取U,X,Z等值，所以在'0'和'1'的四种组合之后还增加了一个when others分支，如果采用Bit类型的信号则没有这个问题。综合时，通常把Std_logic类型信号看作和Bit类型信号相同，因此这个分支将被忽略。在进位链的描述中，比较值得注意的是process中每一个if语句都形成if…else…end if结构，这样综合出的电路将是组合逻辑电路。

ALU控制信号主要有三组，下面给出了其VHDL代码。其中L_XAOMD是进位链控制信号，在它为'1'时，表示发生逻辑运算或乘除指令，进位链不起作用。这时，进位链每一位均为'1'，异或生成和电路将布尔函数电路的结果直接送入ALU的结果寄存器。MU_NOCARRY控制进位链最低位的进位输入。在执行ADDC指令时(OP_With_CY1为1)，若CY标志位为'1'，或在执行SUBB指令时(OP_With_CY0为1)CY为'0'，进位链最低位都要输入一个进位位，使得和的结果加1。4位一组的Y信号控制ALU进行何种运算，Y信号是由OP_ALU2,OP_ALU1和OP_ALU0译码产生的，并在F1上升沿被锁存。

```
-- ALU 控制信号
L_XAOMD <= OP_ALU1 or OP_ALU0;
process(F1,L_XAOMD)
begin
    if F1= '1' and F1'Event then
        MU_XAOMD <= L_XAOMD;
    end if;
end process;

MU_NOCARRY <=  (not MU_XAOMD) and (OP_DIV nor ((OP_With_CY1
and CY) or (OP_With_CY0 and (not CY))));
process(MU_NOCARRY)
begin
    if MU_NOCARRY= '0' then
        L_REND <= '1';
    else
```

```
            L_REND <= '0';
        end if;
    end process;

    Y_Gen : process(F1,OP_ALU2,OP_ALU1,OP_ALU0)
    begin
        if F1 = '1' and F1'Event then
            Y(3) <= (OP_ALU2 nor OP_ALU0) or OP_ALU1;
            Y(2) <= OP_ALU2;
            Y(1) <= OP_ALU0 or (OP_ALU2 and (not OP_ALU1));
            Y(0) <= OP_ALU2 nor OP_ALU1;
        end if;
    end process;
```

5.4.4 控制单元

微处理器和微控制器对信息进行处理是通过程序的执行来实现的，而程序是能够完成某个确定算法的指令序列，因此计算机处理信息的过程就是一个循环式的取指令、分析指令和执行指令的过程。由此可知，控制单元应该具备如下一些功能：

(1) 取指令：当程序已经存放在主存中时，由程序计数器 PC 指出当前指令地址，通过执行“指令地址送主存地址锁存器”和“读主存”微操作，从主存中取出指令。

(2) 分析指令：这个步骤也被称为解释指令或指令译码，即对取出的指令进行分析，指出该指令要求进行哪些微操作，产生相应的微操作控制信号，有时还需要进行操作数地址译码。

(3) 执行指令：根据指令分析得到的结果，按照一定算法形成相应的微操作控制信号序列，通过数据路径、存储器和外设的具体执行，实现每条指令的功能；(2)和(3)经常是混合在一起进行的，其间并没有明显的界限；同时，跳转指令往往就在控制单元内完成。

(4) 对异常情况的处理：这主要是指发生中断的情况，通常 CPU 会停止当前程序的执行，转入中断服务程序，处理完毕后再返回原程序执行。

控制单元由程序计数器 PC、指令译码器、地址生成电路及中断逻辑组成。PC 为一个 16 位计数器；指令译码采用硬连线译码。地址生成电路是由 0 比较器和 16bit＋8bit 的加法器构成，用于跳转控制和跳转电路生成。PC 输出到 ROM 地址锁存器。中断逻辑包括改进的菊花链中断判优逻辑和中断向量入口地址生成电路，其框图如图 5.12。必须指出的是，由于控制单元的设计没有体系兼容性的问题，所以 VHDL 程序并没有把图 5.12 中的每一模块都作为单独的实体(entity)，有的模块是以进程的形式出现的。

1. 指令寄存器 IR 和取指

指令寄存器 IR 就是指令 ROM 的输出缓冲寄存器，它由总线接收指令，输出直接送往指令译码器译码。下面给出了 IR 的 VHDL 描述。在指令执行的 TSF＝112 时刻，

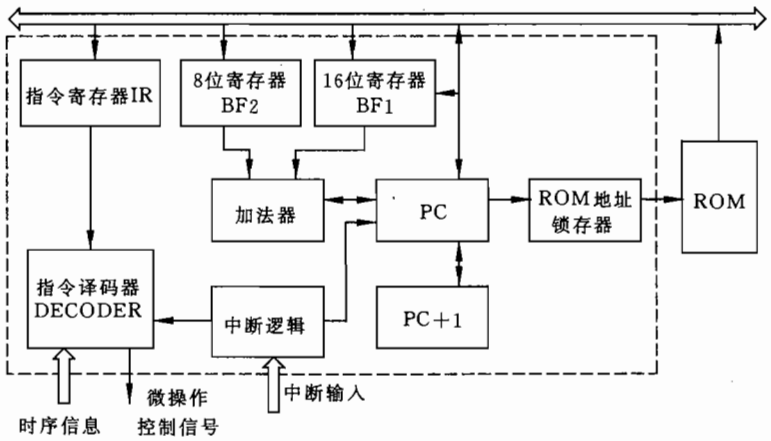


图 5.12　控制单元框图

L_BEGIN信号高有效，表示需要取指；在下一个 F1 时钟高电平期间，IR 从总线获得当前指令。

```
---- 指令寄存器 IR 的 VHDL 代码
entity IR_Reg is
    port(F1,F2: in Std_logic;
         SS1 : in Std_logic;
         T1 : in Std_Logic;
         OP_IRTOBUS : in Std_logic;
         DBUS : inout Std_logic_vector(7 downto 0);
         IR_Q : buffer Std_logic_vector(7 downto 0));
end IR_Reg;

architecture behavior of IR_Reg is
    signal MU_BUSTOIR,MU_IRTOBUS : Std_logic;
    signal L_BEGIN : Std_logic;
begin
    process
    begin
        wait until F2'Event and F2='1';
        L_BEGIN <= SS1 and T1;
    end process;

    MU_BUSTOIR <= F1 nand L_BEGIN;
    MU_IRTOBUS <= F1 nand OP_IRTOBUS;
```

```
    Update : process(MU_BUSTOIR,DBUS)
    begin
        if MU_BUSTOIR= '0' then
            IR_Q <= DBUS;
        end if;
    end process;
-- 向总线输出
    Output : process(MU_IRTOBUS,IR_Q)
    begin
        if MU_IRTOBUS= '0' then
            DBUS <= IR_Q;
        else
            DBUS <= "ZZZZZZZZ";
        end if;
    end process;

end behavior;
```

2. 程序计数器 PC

16 位程序计数器 PC 的作用是计算下一条指令的地址。在执行一般指令时,PC 增量指向紧邻的下一条指令,但应该注意的是 80C51 的指令长度不一,所以 PC 增量必须反映当前指令的长度。在执行跳转指令时,PC 先根据本指令长度增加,然后再看是否符合跳转条件,如果符合,则要将 PC 值修改为目标地址。在上电初始化时,PC 的值为 0000H。

下面的 VHDL 源代码的第(a)部分是 PC 的描述。PC 增量以 PC+1 的形式进行,若指令长度为 2 或 3 字节,则进行 2 或 3 次 PC+1。由于 PC+1 是最频繁进行的微操作,由此设置了专用的加 1 电路 INC16 来实现它,并由 OP_INC 控制进行。在相对跳转指令时,PC 的输入来自地址形成电路的加法器。INC16 实际上是一个传递进位的加 1 进位链,下面 VHDL 代码的第(b)部分是它的描述。在管脚 EA 为 0 或 EA 为 1 且 PC 计数值大于 4kbit 时,微控制器应从外部程序存储器读入指令,因此设置 C_4K 信号用于指示这一情况的发生。C_4K 被送到地址译码器,在为'1'的情况下,PC 的输出不再被 ROM 的地址锁存器锁存,而是送入 P2 口(高 8 位)和 P0 口(低 8 位)。

地址形成电路由 0 检测电路和加法器组成,用于计算相对跳转的目标地址。0 检测器检验总线上是否为 00H,根据当前指令决定是否跳转。若跳转,则向 BF2 载入偏移量,否则向 BF2 装入 00H。加法器从 PC,DPTR 和总线上取数据,输出直接送到 PC。由于 PC 是 16 位寄存器,而相对跳转偏移为 8 位,所以这个加法的输入寄存器为 16 位的 BF1(总线数据传 TBF1 要分两次进行)和 8 位的 BF2。下面一段代码的第(c)部分是 Adder 的代码。

```
-- 第(a)部分: PC 的 VHDL 代码
MU_BFtoPC <= F1 nand OP_BFtoPC;
```

```
MU_PCLtoBUS <= F1 nand OP_PCLtoBUS;
MU_PCHtoBUS <= F1 nand OP_PCHtoBUS;
MU_INC <= F1 nand OP_PC_INIT;
MU_PCLtoPC <= F2 nand OP_PC1toPC;

Update : process(RST,MU_PC1toPC,MU_BFtoPC,PC1_Q,BF_Q)
begin
    if RST= '1' then
        PC_Q <= (others=> '0');
    elsif MU_PCLtoPC= '0' then
        PC_Q <= PC1_Q;
    elsif MU_BFtoPC= '0' then
        PC_Q <= BF_Q;
    end if;
end process;

UUT_INC : INC16 Port Map(PC_Q,MU_INC,PC1_Q);
    PC_LT_4K <= PC_Q(15) or PC_Q(14) or PC_Q(13) or PC_Q(12);
    C_4K <= (not EA) or (EA and PC_LT_4K);

Output : process(MU_PCLtoBUS,MU_PCHtoBUS,PC_Q)
begin
      ......
end process;

-- 第(b)部分: INC16 的 VHDL 代码
INC : process(INPUT,CTRL)
    variable SUM : Std_logic_vector(15 downto 0);
begin
    if CTRL= '0' then
    SUM := INPUT;
    for I in 0 to SUM'High loop
          if SUM(I) = '0' then
             SUM(I) := '1';
             exit;
          else SUM(I) := '0';
          end if;
    end loop;
```

```
    OUTPUT <= SUM;
end if;
end process;
-- 第(c)部分：地址形成电路的 VHDL 代码
MU_INC <= OP_INC nand F1;
MU_DPTRTOBF1 <= OP_DPTRtoBF1 nand F1;
MU_PCTOBF1 <= OP_PCtoBF1 nand F1;
MU_ACJ <= OP_ACJ nand F1;
MU_BUSTOBF1H <= OP_BUStoBF1H nand F1;
MU_BUSTOBF1L <= OP_BUStoBF1L nand F1;
MU_BUSTOBF2  <= OP_BUStoBF2  nand F1;
MU_BUSTEST <= OP_BUSTEST nand F1;
MU_LOAD_REL <= OP_LOAD_REL nand F1;
MU_JC <= OP_JC nand F1;

T1S1P2 <= T1 and SS1 and F2;
S3P2 <= SS3 and F2;

Test : process(MU_BUSTEST,DBUS)
begin
    if MU_BUSTEST= '0' then
      L_ZERO <= not (DBUS(7) or DBUS(6) or DBUS(5) or DBUS(4) or
      DBUS(3) or DBUS(2) or DBUS(1) or DBUS(0));
    end if;
end process;
L_SATISFY <= OP_Not_Z xor L_ZERO;
L_CY_SATISFY <= OP_NC xor CY;

Load_ BF1 : process (MU_ DPTRtoBF1, MU_ PCtoBF1, MU_ ACJ, MU_
BUStoBF1H,MU_BUSTOBF1L,DPTR_Q,PC_Q,IR_765,DBUS)
begin
    if MU_DPTRtoBF1='0' then
        BF1 <= DPTR_Q;
    elsif MU_PCtoBF1='0' then
        BF1 <= PC_Q;
    elsif MU_ACJ= '0' then
        BF1 <= "00000" & IR_765 & DBUS;
```

```
    elsif MU_BUStoBF1H= '0' then
        BF1 <= DBUS & BF1(7 downto 0);
    elsif MU_BUStoBF1L= '0' then
        BF1 <= BF1(15 downto 8) & DBUS;
    end if;
end process;

Load_BF2 : process (T1S1P2, MU_INC, MU_BUStoBF2, MU_LOAD_REL,
L_SATISFY,MU_JC,L_CY_SATISFY,DBUS)
begin
    if MU_BUStoBF2='0' then
        BF2 <= DBUS;
    elsif MU_INC= '0' then
        BF2 <= "00000001";
    elsif T1S1P2='1' then
        BF2 <= "00000000";
    elsif MU_LOAD_REL= '0' then
        if L_SATISFY= '1' then
            BF2 <= DBUS;
        else
            BF2 <= "00000000";
        end if;
    elsif MU_JC= '0' then
        if L_CY_SATISFY= '1' then
            BF2 <= DBUS;
        else
            BF2 <= "00000000";
        end if;
    end if;
end process;

Adding : process(BF1,BF2)
    variable OP1,OP2,SUM : Std_logic_vector(15 downto 0);
    variable CARRY : Std_logic;
begin
    OP1 := BF1;
    OP2 := "00000000" & BF2;
```

```
    CARRY := '0';
    for I in 0 to SUM'High loop
            SUM(I) := OP1(I) xor OP2(I) xor CARRY;
            CARRY := (OP1(I) and OP2(I)) or (OP1(I) and
                    CARRY) or (OP2(I) and CARRY);
    end loop;
    RESULT <= SUM;
end process;
Update : process
begin
    wait until S3P2 = '1';
    BF_Q <= RESULT;
end process;
```

3. 指令译码器

指令译码器在实现上通常有两种思路:随机逻辑译码和结构化的译码电路。随机逻辑译码即硬连线译码是根据译码时序的要求,用组合逻辑单元和存储单元组成控制信号生成电路。这种方法具有较快的运算速度,如果精心优化或译码相对简单(例如 RISC 芯片),常常可以达到极为理想的效果(面积小、速度快)。但是设计时间较长,尤其是在使用传统的电路图输入式的设计方法时,这种方法极易发生错误,而且不易修改。结构化的译码电路是使用微码 ROM 或 PLA 的方式,将微码固化在硬件中。这种方法的电路结构规则,有利于版图的优化,而且由于具有固件(firmware)的特点,修改相对容易。但是这种方法的运算速度较慢,而且微码 ROM 通常要占据很大的芯片面积。在当代的微处理器和微控制器芯片中,RISC 芯片都采用硬连线的译码方式;而 CISC 芯片如 Pentium,由于指令极其复杂,所以采用微码方式。

80C51 的复杂度适中,使用上述的两种方式在代价(主要指面积)上相差不多。为了提高速度,可以采用硬连线的译码方式,下面给出了其 VHDL 描述。由于译码电路是经过 VHDL 语言综合实现的,所以仍然具有易于修改的优点。同时,以下的代码很容易翻译为 PLA 描述语言或微码描述语言。所以,如果设计要求有所改变,可以很容易地修改代码或改变综合约束来满足新的设计要求。

```
-- 指令译码器的 VHDL 描述
entity Decoder is
   port(F1,F2 : in Std_logic;
     SS1,SS2,SS3,SS4,SS5,SS6 : Std_logic;
     T1,T2,T4,T_END : Std_logic;
     IR_Q : Std_logic_vector(7 downto 0);
     OP_SWAP,OP_DA,OP_XCHD,OP_RSHIFT,OP_LSHIFT : out Std_logic;
     ……其他端口);
```

```
end Decoder;
architecture behavior of Decoder is
  ……信号定义
begin
    end_Gen : process(F2,SS6,T_END)
    begin
        if F2= '1' and F2'Event then
            L_END <= SS6 and T_END;
        end if;
    end process;
    And_Clk_Gen : process(L_END_F1,S2P2)
    begin
        if L_END_F1= '1' then
            AND_CLK <= '0';
        elsif S2P2= '1' then
            AND_CLK <= '1';
        end if;
    end process;
    Or_Clk_Gen : process(L_END_F1,S3P2)
    begin
        if L_END_F1= '1' then
            OR_CLK <= '0';
        elsif S3P2= '1' then
            OR_CLK <= '1';
        end if;
    end process;
    ……
    ——以上时钟信号生成
    And_Array : process
    begin
        wait until AND_CLK= '1';
        if IR_Q="11100100" then
            C_C4 <= '1';
        else
            C_C4 <= '0';
        end if;
        ……
    end process;
```

```
    Or_Array : process
    begin
        wait until OR_CLK= '1';
        C0 <= C_C4;
        C1 <= C_Di;
        C2 <= C_03 or C_13;
        C3 <= C_40 or C_50 or C_23 or C_33;
        ……
    end process;

    Output_on_F2 : process
    begin
        wait until F2= '1';
        OP_SWAP <= T1 and SS6 and C0;
        OP_XCHD <= T1 and SS5 and C1;
        ……
    end process;
    ……
end behavior;
```

指令译码器从 IR(指令寄存器)中读取操作码,然后经过三级译码,生成微操作控制信号。前两级译码在一个类似于 PLA 的电路中完成,然后与时序信号叠加。在上面给出的代码中,进程 And_Array 相当于 PLA 中的与阵列,每一个 if…else…end if 实质上形成一个乘积项;进程 Or_Array 相当于 PLA 的或阵列,把乘积项组合成所需要的中间译码信号。当然也可以选择其他非 PLA 的电路结构,这可以通过综合时加以控制来实现,而不需要改变 VHDL 源代码。

与阵时钟与或阵时钟的上跳沿分别发生在 TSF=122 和 TSF=132 时刻。采用这样的语言形式,会引导综合器生成锁存器,从而达到锁存中间译码信号的目的。通过修改综合约束,这样的结构很容易形成流水线式的电路。

上面代码中出现了对 SWAP A 指令的译码过程。首先在与阵时钟上升沿,由指令操作码译出第一级译码信号 C_C4;然后在或阵上升沿,译出第二级译码信号 C0;接下来,在每个 F2 上升沿检查 T1,SS6 和 C0,若均为高电平,则微操作控制信号 OP_SWAP 在一个时钟周期内有效(从 F2 上升沿保持到下一上升沿)。OP_SWAP 信号连至累加器 ACC,经过与 F1 时钟信号作与非运算(F1,F2 为两相不重叠时钟,经过这次与非运算可以消除毛刺电平),形成 ACC 的局部控制信号 MU_SWAP,控制 ACC 完成高低 4 位的互换,从而实现 SWAP A 指令的执行。

指令译码器通常是控制单元中较难设计的部分,因为其规模大,层次多,而且不易调试。使用 VHDL 语言则容易克服这些困难,并且能够较为迅速地在多种综合结果中作出

权衡。

5.5 VHDL 语言结构向硬件的映射

一般说来，硬件描述语言的寄存器传输级代码是作为 EDA 综合工具的输入。本章已经从设计的角度说明了如何用 VHDL 建立硬件寄存器传输级模型，这一节则介绍 VHDL 向硬件的映射。EDA 工业界普遍认为，有效的 VHDL 建模风格是控制综合结果最为有力的手段。为了建立有效的 VHDL 代码，设计师应了解 VHDL 语言结构与综合结果的关系。应该指出的是，由于综合算法的不同，对于同样的硬件描述，不同的 EDA 综合工具可能会得到不同的综合结果。因此，这里着重说明综合过程中具有通用性的处理方法。如果具体工具的处理方式有所不同，请参照该 EDA 工具的使用手册。

VHDL 语言在创立时，主要是为了满足仿真的需要，因此包含了许多用于仿真调试的语法结构。自从 VHDL 被用于综合以来，都是对 VHDL 的子集进行处理，这就是所谓的可综合的 VHDL 子集。不同综合工具支持的可综合子集不尽相同，通常有如下要求：

(1) 延时描述(after 语句、wait for 语句)等被忽略。仿真时，为了具备一定精度，往往在源代码中含有延时语句，这些延时语句情况比较复杂，有时是最大延时约束，有时是典型经验值，还有些是人为加入的，所以现在所有的综合工具都忽略源代码中的延时语句。有些工具干脆把这些语句处理为语法错误，大部分工具忽略延时语句后，给出警告提示。而综合时间约束则在综合过程中通过综合命令输入。

(2) 支持有限类型。VHDL 具有丰富的类型定义，但是有些类型不具备硬件对应物，不可能被综合，如文件类型。通常可综合类型包括枚举类型(包括自定义状态、预定义 Bit 类型和 IEEE 9 值逻辑类型等)、整数、数组等。其余像浮点数类型、记录类型等只得到有限支持，而时间类型等完全不能被综合。

(3) 进程的书写要服从一定的限制。在仿真时，VHDL 进程可以任意书写。而在综合时，通常要求一个进程内只能有一个有效时钟，有的工具还有进一步的限制。

(4) 可综合代码应该是同步式的设计。现在的 EDA 综合工具普遍推荐使用同步式设计风格，即整个芯片电路的状态只能在时钟信号有效时发生改变，也就是说，电路状态不会在仅有数据信号变化而时钟处于无效状态时改变，并且电路中的时钟信号应该都是从某一主时钟经过分频或其他运算得到。当然设计师也可以尝试其它风格的设计，如异步式电路，但这时综合工具产生的结果往往还需要设计师的进一步优化或调整。

本节将依次介绍 VHDL 可综合子集中的语言结构与硬件的对应关系。

5.5.1 VHDL 类型

VHDL 语言中的对象有常量(constant)、信号(signal)和变量(variable)三种，它们都必须定义为如下某种类型。类型定义说明了对象可以使用的数值，并隐含表示了可以对其进行的操作。

1. 可综合类型

面向综合的建模都支持这样一些类型：枚举类型、整数、一维数组。比较先进的综合工

具现在一般也可以处理二维数组和简单的记录类型。

(1) 枚举类型

枚举类型通过列出所有可能的取值来定义,例如:

```
type Std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
type Boolean is (FALSE, TRUE);
type State_type is (HALT, READY, RUN, ERROR);
```

以上 Std_ulogic 的定义实际是对'1','0'等字符进行了重载(overload),由于这个定义已经成为 IEEE 标准,并且描述的是电路连线的状态,因此综合时并不会产生额外硬件。而对于抽象层次更高的 Boolean 和 State_type,则需要进行状态编码,这也是枚举类型在综合时最主要的问题,现在还没有统一的方法解决。一般来说,状态编码是把状态值编码为位矢量(Bit_vector,Std_uloigc_vector 或 Std_loigc_vector),矢量长度是能够表示所有状态的最短位宽。例如,State_type 的 4 个状态值可以分别被编码为"00","01","10"和"11",Boolean 类型编码为'0'和'1'。但是在枚举类型是某个有限状态机的状态时,通常不采用以上办法,而是让各个状态与一定的触发器联系起来。

(2) 整数类型

可综合的整数类型定义总是有界的,例如:

```
type My_Integer is Integer range 0 to 255;
subtype Byte_Int is Integer range -128 to 127;
```

对整数类型进行综合时,综合工具首先将其翻译为位矢量,矢量长度仍取能够满足需要的最短位宽。如果使用默认的整数类型,大部分综合工具按 32 位处理,因此,建议使用子类型定义(subtype)明确指出整数的范围,以便于综合工具进行优化。如果是负整数,通常编码为 2 的补码。综合后的电路中,整数以矢量形式出现,但通常只能以整个矢量为单位访问,即不能单独访问每一位。

(3) 数组

现在综合工具都能够处理一维数组,例如:

```
type Word is array (31 downto 0) of Bit;
type My_RAM is array (1023 downto 0) of Word;
```

对于 Word 类型,综合工具通常将其综合为总线。My_RAM 类型实际是二维的,这种用两个一维数组代替一个两维数组是常用的综合建模技巧。现在先进的综合工具如 Synopsys DC 可以将其综合为 RAM,一般综合工具至少可以把它综合为寄存器堆。

(4) 记录类型

记录类型在定义复杂数据类型时非常方便,能够把不同数据类型的数据组织在一起统一访问。但是,EDA 工业界对综合工具是否应该支持记录类型还没有统一意见,因此大多数综合工具不提供这种能力或只能把组合了简单数据类型的记录进行综合。

2. 预定义类型

VHDL 在 1989 年首次公布时,就提供了两个程序包(package):Standard 和 TextIO,

其中定义了各种预定义数据类型。这些预定义数据类型中，类型 Integer，Bit，Bit_vector，Boolean，Character 和子类型 Natural，Positive 是可综合的。类型 File，Access 和 Real 则不可被综合。

1992 年，IEEE 颁布了标准程序包 Std_logic_1164，其中定义了 9 值数据类型 Std_ulogic，即相应的决断类型 Std_logic。在第 4 章也已经介绍了这个程序包，这里从综合的角度予以讨论。9 值数据类型定义如下：

type Std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');

'0'，'1'，'Z'和'-'在仿真和综合时的处理方式是一致的。'U'和'W'在仿真时是未初始化态，在综合时通常不必考虑。现在'H'和'L'在综合时没有统一的方法。仿真时，'X'表示未知态；综合时，'X'一般当作无关态处理，即等价于'-'。

在面向综合的建模中，合理使用无关态'-'常常能够获得优化的综合结果，下面的例子可以用来说明这个问题，图 5.13 是下面三个结构体对应的综合结果。

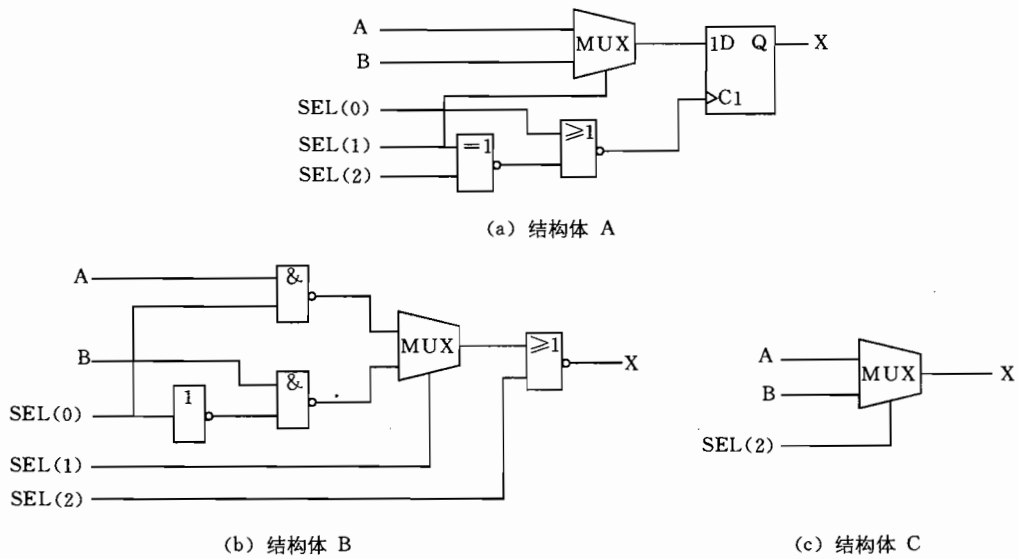


图 5.13　三个结构体综合得到的不同结果

```
-- 结构体 A
architecture A of Dont_Care is
    signal SEL : Bit_vector(0 to 2);
    signal A, B, X : Bit;
begin
    P1: process(A, B, X)
    begin
        case SEL is
            when "001" => X <= A;
```

```
            when "010" => X <= B;
            when others => Null;
        end case;
    end process P1;
end A;

-- 结构体 B
architecture B of Dont_Care is
    signal SEL : Bit_vector(0 to 2);
    signal A, B, X : Bit;
begin
    P1: process(A, B, X)
    begin
        case SEL is
            when "001" => X <= A;
            when "010" => X <= B;
            when others => X <= '0';
        end case;
    end process P1;
end B;

-- 结构体 C
library IEEE;
use IEEE.Std_logic_1164.all;
architecture A of Dont_Care is
    signal SEL : Std_ulogic_vector(0 to 2);
    signal A, B, X : Std_ulogic;
begin
    P1: process(A, B, X)
    begin
        case SEL is
            when "001" => X <= A;
            when "010" => X <= B;
            when others => X <= '-';
        end case;
    end process P1;
end A;
```

在上面的结构体 A 中，由于没有对全部 SEL 的组合指出 X 的对应赋值，在 SEL 的其

他情况下，隐含表示 X 保持原来数值，因此暗示了时序电路。综合器将用 D 触发器(也可能是别种类型的触发器，但在这里 D 触发器是最佳选择)存储 X，用 SEL 进行组合逻辑运算后的信号作为触发器的时钟，这个方案显然硬件代价太高。在结构体 B 中，在 SEL 的其他情况下，把'0'赋给 X，从而使得综合结果成为组合逻辑电路。实际上，如果使用 9 值逻辑中的无关态，还可以得到更好的综合结果，结构体 C 的代码就是使用这种方法，这时可以充分利用综合工具的优化能力。

5.5.2 VHDL 对象

VHDL 语言中定义了三类对象：常量(constant)、变量(variable)和信号(signal)。它们是 VHDL 程序中数据的载体。

1. 常量

常量仅被计算一次。在很多情况下，可以通过使用常量引导综合器获得优化的结果。在综合过程中，常量被处理的方式很多，主要有下述情况。

(1) 当常量被用来描述真值表、ROM 等，或被用于信号赋值，常量在综合时会形成对应的硬件，在下面的 VHDL 代码中，AND_TAB 会被综合为 ROM 硬件或只读寄存器阵列。

```
-- 形成硬件的常量
type TAB2 is array(Bit, Bit) of Bit;
constant AND_TAB : TAB2 := (('0', '0'), ('0', '1'));
signal Z, A, B : Bit;
……
Z <= AND_TAB(A, B);
```

(2) 在常量作为算术运算的一个操作数出现时，综合工具常会对这一算术运算实施特定的优化措施。当然，这样一来常量与综合结果中的硬件就不是一一对应了。例如，在下面的代码中，优化结果是综合工具用左移一位实现乘 2 操作，用右移一位实现除法操作。

```
Y <= A * 2;
Z <= B / 2;
```

(3) 常量在作为条件表达式的一部分(例如下面的代码)时综合工具会对整个语法结构进行布尔优化。

```
constant CST1, CST2, CST3 : Bit_vector(3 downto 0);
signal Value, Z : Bit_vector(3 downto 0);
……
if S=CST1 then …
    case Value is
        when CST1 => …
```

```
……
Z <= CST1
when A= '1' else Value
when B=CST2 else CST3;
```

(4) 常量传播。在下面的 VHDL 代码中，由于数组 ROM 和 ROM(5)的索引都是常数，因此 WORD4 实际上也成为常数，在进一步的优化中，WORD4 将作为常量被处理，这就是常量传播。

```
-- 常量传播
constant ROM : ROM_TYPE := Read("Rom_file.dat");
signal WORD4 : Bit_vector(3 downto 0);
begin
    WORD4 <= ROM(5);
    ……
```

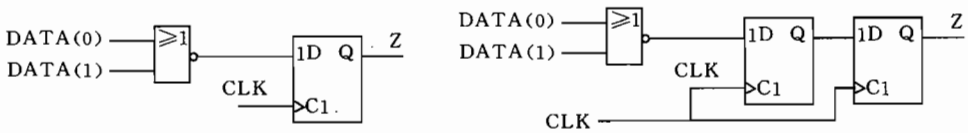
2. 变量与信号

变量和信号有着不同的仿真行为，同样在综合过程中，它们也会产生不同的结果。一般说来，尽量使用变量能够获得比较好的综合结果，因为这样做使得优化的余地较大。但是，并不是所有的综合工具都支持变量的综合，这是需要特别注意的。同时，使用信号可以较好地保持综合前后在 I/O 上的一致性(这里把进程内对信号的读写统称为 I/O)；而且在需要锁存中间结果的时候，经常有必要使用信号。

下面用一个例子来说明变量与信号的不同综合结果，其源代码如下。图 5.14 分别是结构体 A 和 B 对应的综合结果。

```
-- 结构体 A，用变量实现算法
entity Var_Sig is
    port(DATA : in Bit_vector(1 downto 0);
         CLK : in Bit;
         Z : out Bit);
    constant K1 : Bit_vector := "01";
    constant K2 : Bit_vector := "10";
end Var_Sig;
architecture A of Var_Sig is
begin
    Var: process
        variable A1, A2 : Bit_vector(1 downto 0);
        variable A3 : Bit;
    begin
        wait until CLK= '1' and CLK'Event;
        A1 := DATA and K1;
```

```
        A2 := DATA and K2;
        A3 := A1(0) or A2(1);
        Z <= A3;
    end process Var;
end A;

-- 结构体 B,用信号实现算法
architecture B of Var_Sig is
    signal A1, A2 : Bit_vector(1 downto 0);
    signal A3 : Bit;
begin
    A1 := DATA and K1;
    A2 := DATA and K2;
    Sig: process
    begin
        wait until CLK= '1' and CLK'Event;
        A3 <= A1(0) or A2(1);
        Z <= A3;
    end process Sig;
end B;
```



(a) 结构体 A　　(b) 结构体 B

图 5.14　变量和信号对综合的影响

5.5.3　初值

VHDL 中有三种初值:由类型或子类型定义可以得到的默认初值,定义对象时明确指定的初值和进程入口处显示地赋予对象的初值,下面示意给出了这三种情况。

```
-- 设置初值的三种情况
-- type STATES is (RST, FI, ID, IE);
signal STATE : STATES;      --信号 STATE 的默认初值是 RST;
……
signal Z : Bit_vector(3 downto 0) := "0000";      --明确指定的初值
……
P1: process(A, B)
        variable V1, V2 : Std_logic;
```

```
begin
        V1 := '0'; V2 := '1';                    --赋初值
    ......
end process P1;
```

以上三种初值的前两种只在仿真时有意义，在综合时将被忽略。第三种形式将被综合器处理，形成对应电路。在集成电路设计中，复位时赋予各个信号初值是很有必要的，否则很有可能出现不定态。因此无论在仿真还是在综合时，都建议使用系统化的方式给信号和变量赋初值，即上述的第三种形式。

5.5.4 运算符

VHDL 提供了丰富的运算符，表 5.4 分类列出所有 VHDL 预定义运算符，以及相应的优先，表内同一行中运算符优先级相同。

表 5.4 VHDL 预定义运算符

种 类	运 算 符	优先级
二元逻辑运算	or and nor nand xor xnor	最低
关系运算	= /= > < >= <=	↑
移位操作	sll srl sla sra rol ror	↑
加操作	+ − &	↑
一元算术运算	+ −	↑
乘法操作	* / mod rem	↑
其他运算	* * abs not	最高

1. 逻辑运算符

逻辑运算符包括表 5.4 中的 not 和二元逻辑运算符，其操作数可以是 Bit 和 Std_logic 等类型的标量或同长度的矢量对象，也可以是 Boolean 类型的对象。这些算符的综合是直截了当的，直接调用逻辑门单元实现即可，但经过优化后，这些算符可能被合并或改变。图 5.15 是对下述 VHDL 代码中逻辑运算符综合的结果。

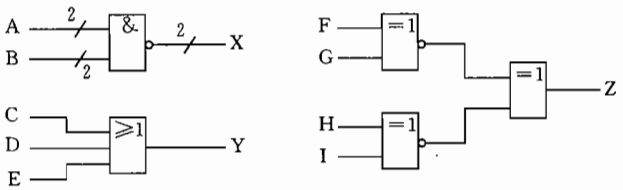


图 5.15 对逻辑运算符综合的例子

```
signal X, A, B : Bit_vector(3 downto 0);
signal Y, C, D, E : Std_logic;
signal Z, F, G, H, I : Boolean;
......
begin
```

```
X <= A nand B;
Y <= C or D or E;
Z <= (F xnor G) xor (H xnor I);
.....
```

2. 关系算符

关系算符的综合没有统一的方法，综合工具常利用被比较数的特点作特定的优化。图 5.16 是对"<"运算符综合的例子。

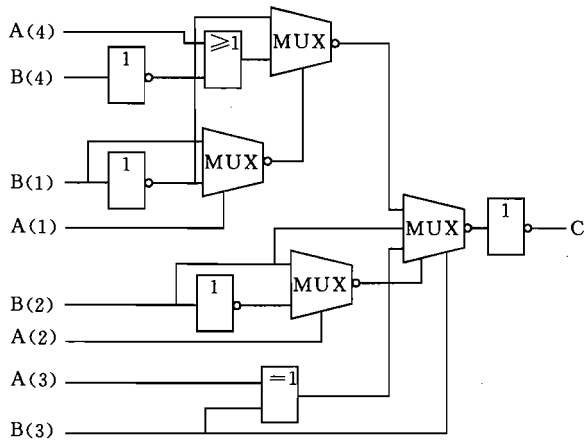


图 5.16　对"<"运算符综合的例子

3. 一元算术运算符

一元算术运算符有三个，即+，-和 abs(取绝对值)。对前两个算符，综合工具大都可以用组合逻辑线路实现，如图 5.17 的例子。abs 算符的处理比较复杂，大部分综合工具尚不能提供支持。

4. 二元算术运算符

现在的综合工具，特别是高层次综合工具，一般都能够直接把+，-，×运算综合为相应的运算线路，部分工具也支持除法运算。mod 和 rem 算符通常不被综合工具支持。如果使用 IEEE 颁布的标准算术运算程序包 Std_logic_arith，那么还可以直接描述对 Bit 或 Std_logic 类型的标量和矢量对象进行算术运算的电路，并进行综合。通常在综合过程中，综合器先把运算符映射为相应的加法器等综合库提供的专用运算部件，然后进行优化，如果某些运算可以用简单线路实现，综合器则用简单线路取代专用运算部件。图 5.18 是对下述 VHDL 代码中+运算符进行综合的例子。

```
entity Adder is
    Port(A, B : in Integer range 0 to 15;
         C : out Integer range 0 to 15);
end Adder;
architecture ALG of Adder is
```

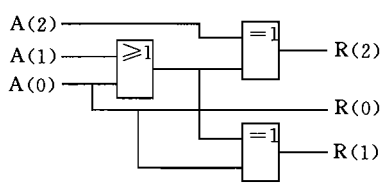


图 5.17　对 R<=-A 综合的例子

```
begin
    C <= A+B;
end ALG;
```

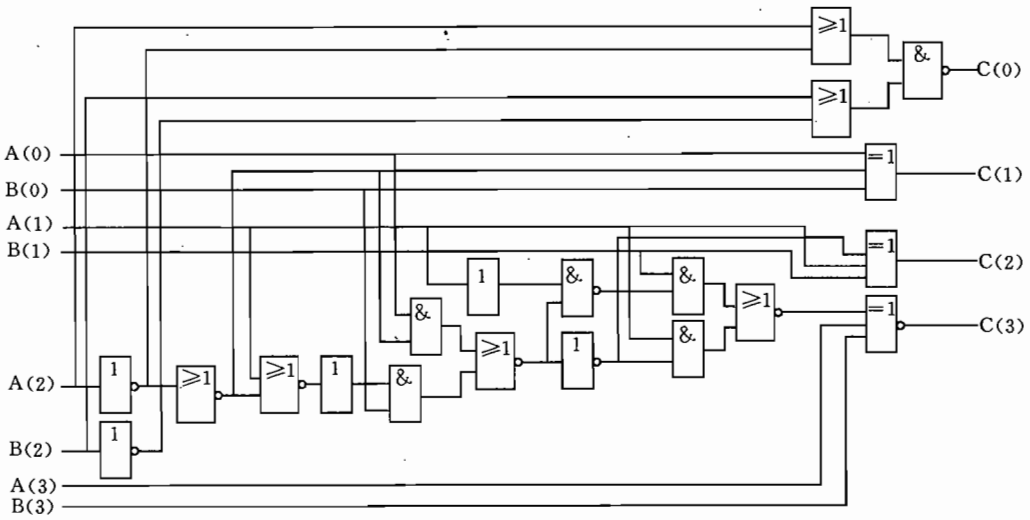


图 5.18　+运算符的综合

5. &算符

&算符为连接运算，通常用于把一组对象捆绑在一起。由于直到 1992 年 VHDL 才引入移位算符，所以也常用连接运算实现移位运算。综合时，一般是在信号之间提供额外的连线实现这一运算，图 5.19 是对下述 VHDL 代码中&算符综合的例子。

```
signal A, C : Bit_vector(2 downto 0);
signal B, S, R : Bit_vector(0 to 5);
signal D : Bit;
begin
    A <= D & not D & D;
    S <= A & B(0 to 2);
    R <= C(1 downto 0) & "000" &
    D;
……
```

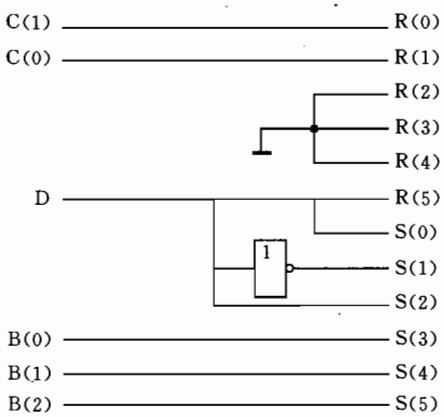


图 5.19　对&算符综合的例子

6. 移位运算

移位运算符是 1992 年 VHDL 再次修订后引入的特征，在综合时有两种处理方式。对于经过锁存的矢量对象，可以在锁存器之间引入附加的数据通路来实现；对于不需锁存的矢量对象，处理方式与&算符相同。

5.5.5 顺序语句

顺序语句只能在进程中出现，而且其出现顺序直接影响到硬件行为。VHDL 能够描述非常复杂的数字电路，很大程度上是由于具有丰富的顺序语句。

1. 变量赋值

变量的引入使得 VHDL 具备了极强的造型能力，能够以类似软件编程的方式描述硬件，特别是在行为综合时，如果没有变量，则描述复杂的 ASIC 行为几乎是不可能的。

变量在综合时要根据整体代码决定其处理方式，主要有以下几种方式：① 用寄存器锁存变量，也就是说，变量经过综合后形成了寄存器，但是生命期不重叠的变量可以共享一个硬件寄存器；② 在逻辑优化过程中被消去，即没有硬件对应物；③ 作为一根硬件连线出现（由组合逻辑输出）。

2. 信号赋值

信号只能在同步语句（显式或隐含的 wait 语句）之后改变状态。通常，综合后的信号赋值语句会被化简为最简形式。综合工具处理信号的方式类似于变量，但有以下区别：① 如果信号需要寄存器锁存，那么被锁存的信号能与其他信号共享硬件寄存器；② 比较先进的综合工具可以按照设计师的要求，决定是否保持综合前后信号的一致性；③ 出现在形如 if CLK＝'1' and CLK'Event 和 then…end if 之间或 wait until CLK＝'0' and CLK'Event 这样的时钟沿同步语句之后的信号都要被锁存，而变量则不一定被锁存。

3. if 语句

if 语句是最频繁出现的条件语句。在综合时，if 的处理分三种情况，下面分别讨论。

(1) if 语句（包括 elsif，else）包含了条件所有可能的取值，称之为完全 if 语句。这时综合器用多路选择器或基本逻辑门的组合来实现电路，如图 5.20 和图 5.21 所示，它们分别是下述进程 P1 和 P2 综合得到的结果。用多路选择器实现电路时，if…elsif…else 中隐含的优先关系会被消去，这是设计师应该注意的。

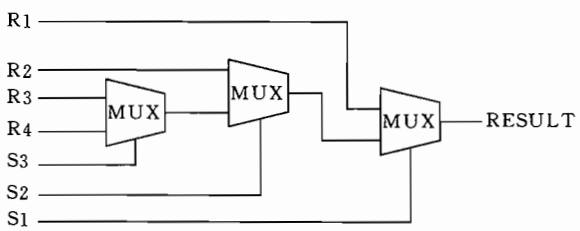


图 5.20　完全 if 语句的综合，使用多路选择器电路

```
P1: process(S1, S2, S3, R1, R2, R3, R4)
begin
    if S1= '1' then
        RESULT <= R1;
    elsif S2='1' then
```

```
        RESULT <= R2;
    elsif S3= '1' then
        RESULT <= R3;
    else
        RESULT <= R4;
    end if;
end process P1;
P2: process(OP, X, Y)
begin
    if OP= '0' then
        RESULT <= X or Y;
    else
        RESULT <= X and Y;
    end if;
end process P2;
```

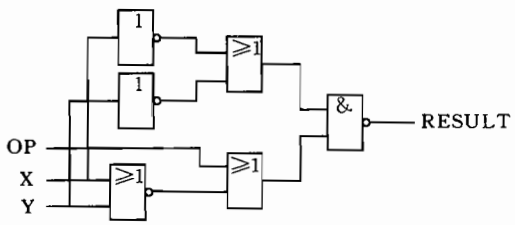


图 5.21　完全 if 语句的综合，使用基本逻辑门电路

(2) 若 if 语句的条件未包含所有可能出现的情况，有效条件是对某信号的跳变进行检测，并且在条件满足时是对信号进行赋值操作，那么会生成触发器。一般把触发器和电平锁存器通称为寄存器，因此这实际上是所谓寄存器推断的一种形式。如果赋值号右边为一复杂表达式，则综合器先用组合逻辑电路计算表达式，计算结果送入触发器的数据输入。图 5.22 是对下述进程 FF 进行寄存器推断的例子。

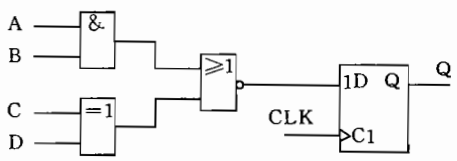


图 5.22　if 语句综合后生成触发器

```
FF: process(CLK, A, B, C, D)
begin
    if CLK= '1' and CLK'Event then
```

```
        Q <= (A and B) nor (C xor D);
    end if;
end process FF;
```

(3) 第三种情况与第二种类似，也属于不完全 if 语句，但有效条件只对信号值进行检测，这时综合器的处理方式也是类似的，只是换为用电平锁存器实现寄存器。

4. case 语句

case 语句与多路选择器电路的对应关系是显而易见的，但是，建模时要注意合理使用无关态和 others 语句，否则会造成电路的复杂化，甚至导致形成时序电路。图 5.23 是综合下述 VHDL 代码中 case 语句的一个例子，注意电路中 CODE 信号已被编码为矢量。

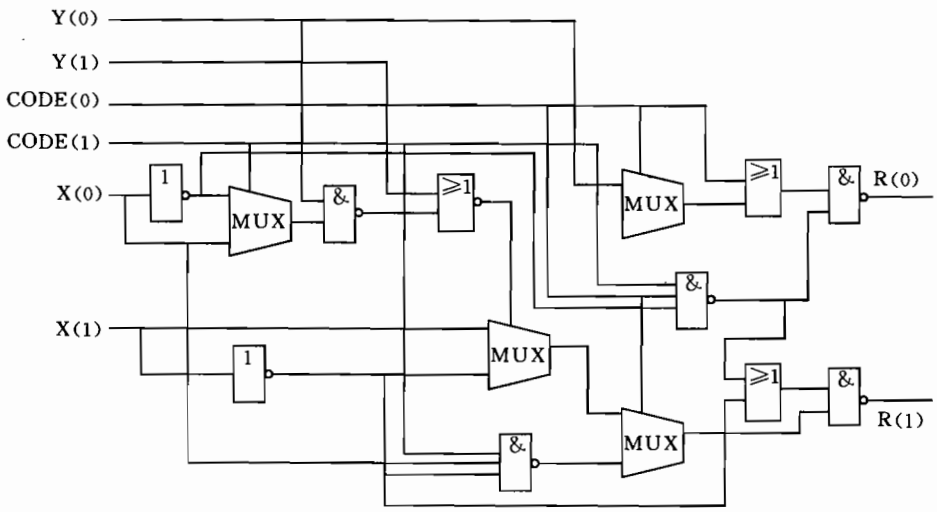


图 5.23 case 语句的综合

```
type CODE_TYPE is (ADD, SUB, RST, INCX);
subtype  WORD is Integer range 0 to 3;
signal CODE : CODE_TYPE;
signal X, Y, R : WORD;
.....
P1: process(CODE, X, Y)
begin
    case CODE is
        when ADD => R <= X + Y;
        when SUB => R <= X - Y;
        when RST => R <= 0;
        when INCX => R <= X + 1;
    end case;
end process P1;
```

5. 循环语句

VHDL 的循环语句有三种：for 循环、while 循环和无限循环 loop…end loop。实际上，loop…end loop 可以看作是循环条件永远为真的 while 循环。在行为综合中，循环语句的处理是极其复杂的，第 3 章对此已作了介绍，这里从寄存器传输级的角度加以讨论。

在寄存器级进行综合，要求 for 循环的上下界必须是静态已知。例如下面给出了两段 VHDL 代码，第一段代码中，由于循环上界 RG 是动态对象，所以是不可综合的。反之，第二段代码则可以被综合工具接受。这段代码通过使用 next 语句，形成了一个选择性的连线网络，如图 5.24 所示。

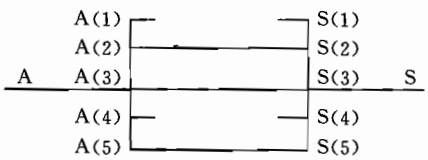


图 5.24　for 循环的综合

```
-- 由于上界不确定而不可综合的代码
...
constant N : Natural := 31;
signal RG, SUM : Natural range 0 to N;
signal CLK : Bit;
signal A : Bit_vector(0 to N);
...
P1: process
    variable CPT : Natural range 0 to N;
begin
    wait until CLK= '1' and CLK'Event;
    for J in 1 to RG loop
        if A(J) = '0' then
            CPT := CPT+1;
        end if;
    end loop;
    SUM <= CPT;
end process P1;
...

-- 可综合的 for 循环语句
constant COND : Bit_vector(1 to 5) := "01101";
signal S, A : Bit_vector(1 to 5);
...
```

```
for I in COND'Range loop
    next when COND(I)= '0';
    S(I) <= A(I);
end loop;
...
```

VHDL 定义了 next 和 exit 语句来中断循环的正常执行，现在的综合工具都可以处理这两种语言结构。图 5.25 是对下述 VHDL 代码中的循环综合的例子，这个电路对信号 V 中的'1'进行计数，代码中使用了 exit 语句。

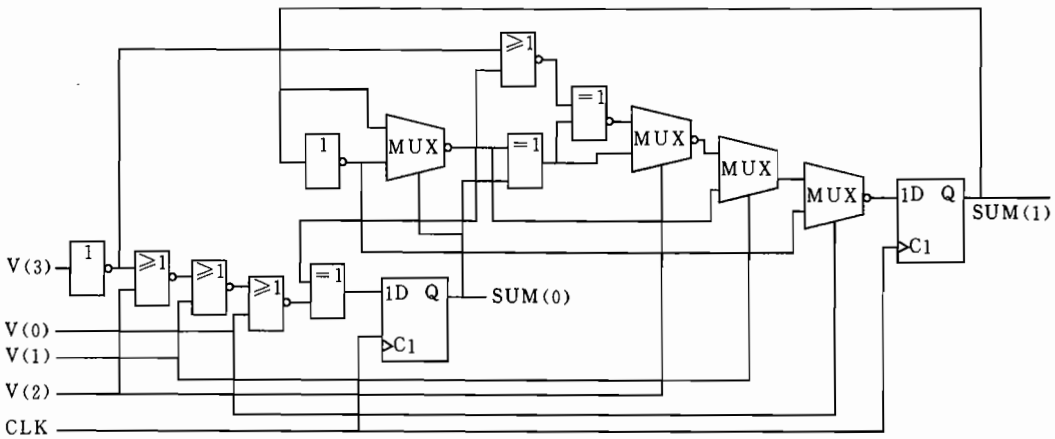


图 5.25　对带有 exit 语句的循环综合

```
signal V : Bit_vector(3 downto 0);
signal SUM : Natural range 0 to 3;
signal CLK : Bit;
...
P1: process
    variable COUNT : Natural range 0 to 3;
begin
    if CLK= '1' and CLK'Event then
        for J in 0 to 3 loop
            exit when V(J)= '1';
            COUNT := COUNT +1;
        end loop;
    end if;
end process P1;
```

6. 子程序调用

VHDL 的子程序调用有函数和过程两种形式。综合时，比较先进的综合工具允许设计师控制子程序展开或是作为单独模块处理。只有描述硬件行为的子程序才是可综合的，

完成数据转换的子程序和决断函数将被综合工具忽略。函数通常被综合为组合逻辑电路；如果过程同时具备输入和输出参数，并且内部没有 wait 语句，则综合后形成组合逻辑电路，否则形成时序逻辑。

5.5.6 并行语句

VHDL 的并行语句出现在结构体内，可综合的并行语言结构包括进程、并行赋值语句、块语句、生成语句等。

1. 进程

进程是 VHDL 中描述硬件行为最有力的方式。进程内的语句属于顺序语句，而进程本身则属于并行语句。进程的综合是比较复杂的，主要有这样一些问题：综合后的进程是用组合逻辑还是时序逻辑电路实现？进程中的对象是否有必要用存储器部件（主要指寄存器、触发器、电平锁存器或 RAM）实现？

进程语句结构需要至少一个同步控制点，否则除了初始化阶段，永远不会再次被激活。同步控制点表现在代码中就是同步语句，同步语句可以是 wait on（包括以敏感信号列表出现的隐含式 wait on 语句）或 wait until。一般说来，只有一个同步点的进程，或者是具有多个同步点、但是都使用完全相同的同步控制信号，当进程不需要“记忆”在哪一个同步点上被挂起时，不会形成存储器。例如下面的代码所示。

```
-- 综合后不需要存储器的 VHDL 进程
P1: process(A, B, C)
    ...
begin
    ...     --其中没有其他同步描述
end process P1;

--需要存储器的 VHDL 进程
P2: process
begin
    wait until CLOCK= '1' and CLOCK'Event;
    S <= '0';
    wait until CLOCK= '1' and CLOCK'Event;
    S <= '1';
end process;
```

进程中会形成存储器的结构还可能来自变量。有些变量纯粹是运算的中间结果，即所谓工作变量。例如下述的代码中的进程 P1，不需要为变量配置寄存器或其他类型的存储器，此时同步语句隐含在进程两次被激活的过程中。概括起来，如果两个同步语句之间的某变量总是在被读之前作为赋值的对象，则不会形成对应的硬件。如果变量需要在进程挂起到下次被激活的过程中存储数据，则显然需要形成硬件存储器，例如下述代码中的进程

P2。这种情况可以概括为，在进程的两个同步语句之间，如果至少有一个变量在作为赋值对象之前被读访问至少一次，那么该变量需要硬件存储器。

```
-- 不会形成对应硬件的变量
P1: process(A, B, C)
    variable VAR : Bit;
begin
    VAR := A xor B;
    S <= VAR and C;
end process P1;

-- 需要存储器的变量
P2: process
    type T_STATE isd (STOP, GO);
    variable STATE : T_STATE;
begin
    wait until CLOCK= '1' and CLOCK'Event;
    case STATE is      --STATE 在被赋值之前先被读访问
        when STOP => STATE := GO;
        when GO => STATE := STOP;
    end case;
end process P1;
```

信号同样可能在综合后形成硬件。例如下面的代码，综合结果通常为一触发器，这是因为虽然进程内只有一个同步点，但是在进程两次被激活的过程中，Q 必须保持原来状态，即使 D 的状态已经发生改变。

```
P1: process
begin
    wait until CLOCK= '1' and CLOCK'Event;
    Q <= D;
end process P1;
```

如果进程综合后的电路含有寄存器，那么自然就是时序电路。此外还有什么依据可以判定进程被综合为时序电路呢？概括地说，在以下两种情况，进程被综合为时序电路：① 进程中所有被读访问的信号不在敏感信号列表之内，如下面代码的进程 P1；② 进程中至少有一个信号不是 if 或 case 中每种可能的分支上的赋值对象，如下面代码中的进程 P2 的信号。

```
-- 被综合为时序电路的进程
type T_STATE isd (STOP, GO);
```

```
signal STATE : T_STATE;
……
P1: process
begin
    wait until CLOCK= '1' and CLOCK'Event;
    case STATE is      --STATE 在被赋值之前先被读访问
        when STOP => STATE <= GO;
        when GO => STATE <= STOP;
    end case;
end process P1;

P2: process(D, G)
begin
    if G= '1' then
        Q <= D;
    end if;
end process P2;
```

最后再给出一个例子。在下面代码的进程 P1 中，由于只有在 G 为'1'时，给 Q 赋值，按照上面的第二条判据，综合器会为 Q 分配一个电平锁存器。由于 Q_BAR 的数值来自对 Q 的运算，如果综合元件库中有同时输出 Q 的正反相的锁存器，则只需一个锁存器就可以实现该进程；如果没有，则综合工具会再用一个锁存器实现 Q_BAR！这显然不是理想的设计，因为即使没有满足需要的锁存器，只需对 Q 进行一次反运算即可获得 Q_BAR。下面代码中的进程 P2 能够解决这一问题，也就是说，把对 Q_BAR 的赋值放在进程之外，这样综合器有足够的余地进行优化。以上的情况是在寄存器级综合中经常遇到的，特别在电路规模很大的情况下，必须注意语句出现的位置，否则会无谓地增大电路规模。

```
--  时序电路综合的例子
P1: process(D, G)
begin
    if G= '1' then
        Q <= D;
    end if;
    Q_BAR <= not Q;
end process P1;

P2: process(D, G)
begin
```

```
    if G= '1' then
        Q <= D;
    end if;
end process P2;
Q_BAR <= not Q;
```

总结以上内容，在下述条件同时满足的情况下，进程会被综合为组合逻辑电路：

(1) 进程有显示定义的敏感信号列表或进程只有单一的同步控制点；

(2) 进程中不含有任何变量，或者含有变量，但变量出现在读操作之前(出现在赋值号 :=的左边)；所有被读访问的信号都出现于敏感信号列表中；

(3) 进程中输出的信号在“条件”的各种可能情况下都被赋值，且赋值号右边不出现该信号。

2. 信号赋值语句

信号赋值语句的处理是直截了当的，下面的语句(a)被综合成一根硬连线；对于语句(b)，R 将被当作常数处理；语句(c)被综合为组合逻辑电路。当然，语句(a)和(c)经过逻辑优化后，可能改变形式或者被消去。

```
S <= A;                                          -- (a)
R <= '1';                                        -- (b)
T <= (B xor C) or (D and E) or (F xnor G);       -- (c)
```

3. 条件和选择赋值语句

VHDL 的并行语句有两种方式进行有条件的赋值，即条件赋值 when…else 和选择赋值 with…select…when。实际上，这两种语法结构都可以改写为等价的顺序语句。例如下面的并行条件赋值(a)和进程 P1 内的代码是完全等价的。同时，可以看到进程也满足用组合逻辑电路实现的特点，因此这段代码在综合后会用多选一网络实现，如图 5.26 所示。

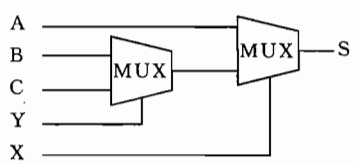


图 5.26 形成组合逻辑的并行条件赋值、与其等价的进程以及综合结果

```
S <= A when X= '1'
    else B when Y= '1'
    else C;                                      -- (a)
P1: process(A, B, C, X, Y)
begin
    if X= '1' then S <= A;
    elsif Y= '1' then S <= B;
    else S <= C;
    end if;
end process P1;
```

如果把前面的代码分别修改为下面的语句(b)和进程 P2 的形式,那么综合后生成时序逻辑电路,如图 5.27 所示。

```
S <= A when X= '1'
  else B when Y= '1'
  else S;                                    --(b)

P2: process(A, B, X, Y)
begin
    if X= '1' then S <= A;
    elsif Y= '1' then S <= B;
    else S <= S;                             --或没有 else 分支
    end if;
end process P2;
```

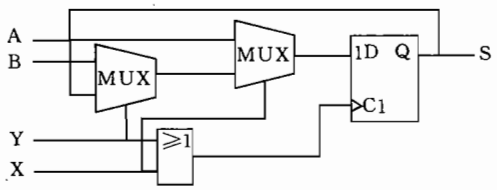


图 5.27 形成时序逻辑的并行条件赋值、与其等价的进程及综合结果

在上面的代码中,最后一个分支上 S 被赋给自身。为了减少仿真时无用的数值传递,1992 年 VHDL 引入了一个新的关键字 unaffected,表示此时被赋值信号保持原来状态。这样,以上情况的代码可以改写为下面语句(c)和(d)的形式。综合后形成时序逻辑电路,如图 5.28 所示。

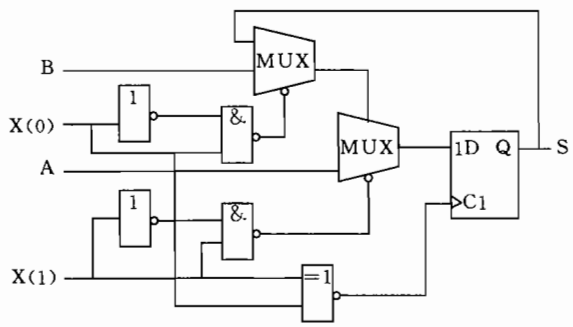


图 5.28 unaffected 语句的综合

```
S <=A when X="01"
      else B when X="10"
```

```
        else unaffected;                                    -- 语句(c)

with X select                                               -- 语句(d)
    S <= A when "01" else
         B when "10" else
         unaffected;
```

选择赋值语句与进程中的 case 语句的对应关系是显而易见的。与条件赋值语句类似，选择赋值语句可能被综合为组合逻辑或时序逻辑电路。例如下面的语句(a)的代码综合后是组合逻辑，而语句(b)综合后为时序逻辑。

```
--  语句(a)
with Sel select
    S <= A and B when '1' else
         C and B when '0';


--  语句(b)
with Sel select
S <= A and B when '1' else
         C and B when '0' else
S when others;
```

4. 元件例化语句

元件例化语句提供了使用以前建立的模块的手段。在综合过程中，提供综合命令的控制。例化语句调用的元件可以用如下一些方式处理：

(1) 展平，即取消层次。把元件本身的描述代入上一级描述，然后整体进行综合和优化；

(2) 只把被例化元件综合一次，然后遇到例化这一元件的语句均使用这一综合结果，也就是说，每遇到这个元件的例化语句均在综合结果中加入一个相同模块；

(3) 只把被例化元件综合一次，但是在每次处理例化元件语句时，根据上下文的代码对元件的接口逻辑重新综合。

元件例化语句常与配置语句联合使用，可以通过配置语句引导综合工具选择适当的设计版本。

5. 块语句

块语句有把一些相关并行语句组织到一起和进行保护赋值两种作用。第一种块语句在综合处理上与一般的并行语句没有区别，只是有些综合工具可以把一个块结构当作模块来处理，也就是提供了一种层次化的手段。第二种块语句一般用来描述寄存器(触发器或电平锁存器)和三态器件。下面的代码描述了一个下降沿激活的触发器和三态总线连接，图 5.29 是三态总线描述的综合后结果。由于 DATA 是 Std_ulogic_vector 类型，所以在赋值时进行了类型转换。

```
-- 块语句描述的触发器
Block1: block(CLK = '0' and CLK'Event)
    signal R : Bit;
begin
    R <= guarded DATA;
    S <= guarded R;
end block Block1;
```

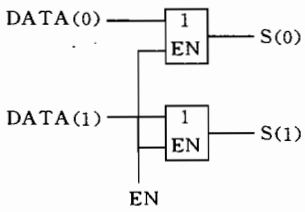


图 5.29　完成保护赋值的块语句

```
-- 三态总线连接的描述
use IEEE. std_logic_1164. all;
signal EN : Std_ulogic;
signal DATA : Std_ulogic_vector(1 downto 0);
signal S : Std_logic_vector(1 downto 0) bus;
...
Tri_State: block(EN = '1')
begin
    S <= guarded Std_logic_vector(DATA);
end block Tri_State;
```

6. 并行过程

并行过程在综合处理上类似于元件例化语句，但是更加灵活。在参数不同时，同一过程可能得到不同的综合结果。并行过程一般在保持层次的条件下单独被综合，并且在同一过程被多次调用时，默认的处理方式是共享这一过程对应的硬件。

7. 生成语句

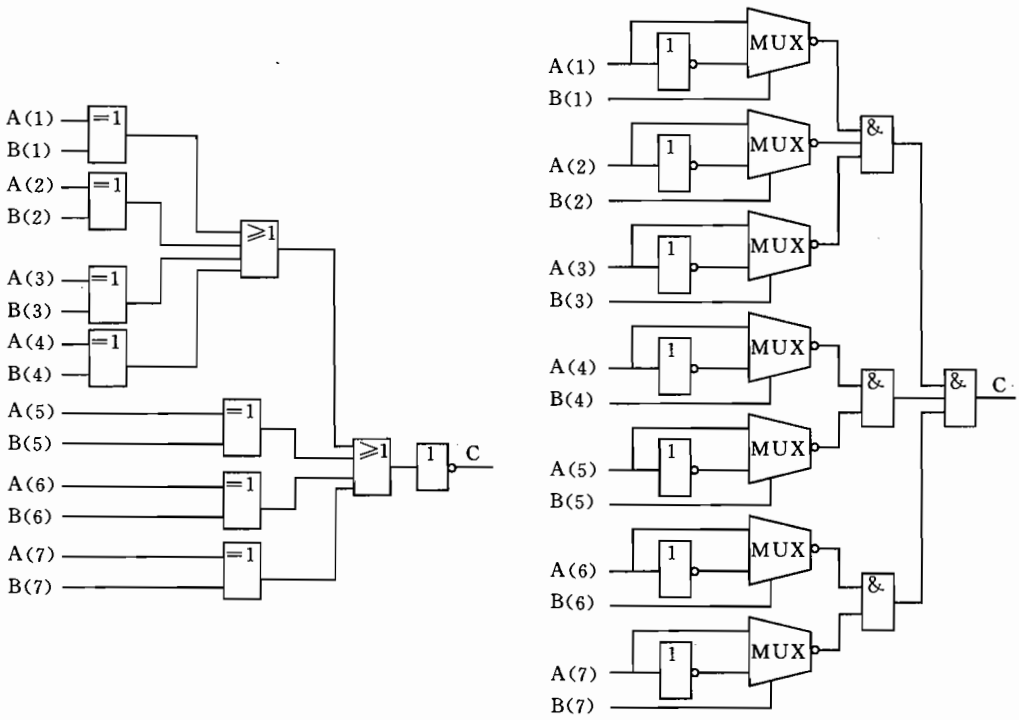
生成语句实际上是系统化地例化元件的方式，在综合时与硬件的对应关系是很明显的。

5.5.7　优化约束

综合时，除了通过编程来控制综合结果之外，一般要设置综合约束来引导综合工具达到需要的综合目的。综合约束的种类很多，例如，对最大/最小延时的约束、建立/保持时间约束、最大扇出约束等等，但对综合产生最重要影响的是优化目标，常见的优化目标是最快运算速度或最小面积，现在比较先进的工具开始支持面向功耗的优化。通常，综合工具以面积为默认优化目标。因此，即使代码完全相同，如果优化目标不同，仍有可能产生完全不同的综合结果。图 5.30 是对下述代码采用不同优化目标综合的例子。

```
entity Check_EQL is
        port(A, B : in Bit_vector(1 to 7);
             EQL : out Boolean);
    end Check_EQL;
    architecture ALG of Check_EQL is
```

```
    begin
        EQL <= (A=B);
end ALG;
```



(a) 面向面积的综合结果　　(b) 面向速度优化综合结果

图 5.30　对相同代码采用不同优化目标

5.5.8　设计实例

5.5 节以较大的篇幅介绍了 VHDL 语言结构与硬件的对应关系，这里给出一个有限状态机的设计实例，其代码如下，综合出的电路图如图 5.31。

```
Package Types is
    type STATES is (STOP, SLOW, MEDIUM, FAST);
end Types;

use work.Types.all;

entity FSM is
    port(ACCELERATOR : in Bit;
         BRAKE : in Bit;
```

```
        CLK : in Bit;
        SPEED : buffer STATES);
end FSM;

architecture ALG of FSM is
begin
    P1: process
    begin
        wait until CLK= '1' and CLK'Event;
        if ACCELERATOR= '1' then
           case SPEED is
                when STOP => SPEED <= SLOW;
                when SLOW => SPEED <= MEDIUM;
                when MEDIUM => SPEED <= FAST;
                when FAST => SPEED <= FAST;
           end case;
        elsif BRAKE='1' then
            case SPEED is
                when STOP => SPEED <= STOP;
                when SLOW => SPEED <= STOP;
                when MEDIUM => SPEED <= SLOW;
                when FAST => SPEED <= MEDIUM;
            end case;
        end if;
    end process P1;
end ALG;
```

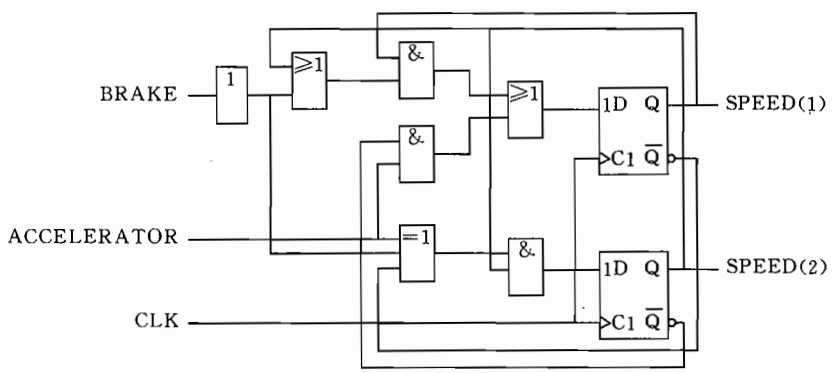


图 5.31 FSM 的综合结果

第6章 多层次混合设计

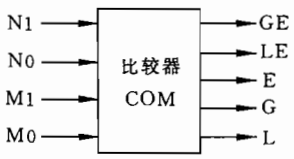
本章讨论组合逻辑及时序逻辑电路的多层次混合设计问题，将混合使用硬件描述语言、卡诺图、状态表等多种方法设计组合逻辑及时序逻辑电路。电路设计覆盖了系统级的算法描述直至逻辑门级电路描述。本章还讨论了综合使用各种方法设计微代码式控制电路单元。

6.1 组合逻辑电路设计

本节讨论组合逻辑电路的设计问题。在使用本节讨论的电路设计方法之前，电路设计者必须对设计指标要求仔细分析，作出采用组合逻辑电路完成设计这一决策。例如对下面的例 6.1 进行分析，可以得出适宜采用组合逻辑电路实现的结论。

例 6.1　二进制比较器的设计。

设计一个器件，使它对两个二进制数进行比较，以确定哪一个大。输入是两个 2 位的二进制数，分别用 N=N1 N0 和 M=M1 M0 表示。输出为二进制信号 GE,LE,G,E 和 L。电路的方框图和功能定义如图 6.1 所示。根据方框图以及功能描述，很容易得出描述电路接口的 VHDL 实体如下：



电路行为描述

L：当 N 的数值比 M 小时该信号输出为 true

E：当 N 的数值与 M 相等时该信号输出为 true

G：当 N 的数值比 M 大时该信号输出为 true

LE：当 N 的数值比 M 小或 N 的数值与 M 相等时该信号输出为 true

GE：当 N 的数值比 M 大或 N 的数值与 M 相等时该信号输出为 true

图 6.1　例 6.1 的电路的方框图

```
entity COM is
    generic(DEL:TIME);
    port(N1,N0,M1,M0: in Std_logic;
        GE,LE,E,G,L: out Std_logic);
```

end COM;

在上述 VHDL 实体中，类属参数 DEL 用来描述从信号输入到信号输出的延时。电路有四个输入端口(N0,N1,M0,M1)，分别用来表示两个二进制输入数据。有五个输出信号，分别表示对两个输入信号比较的结果。

对这个电路的性能指标要求进行分析可知，电路的输出完全由电路当前的输入信号确定，与过去的输入无关。因此有可能用组合逻辑电路实现电路的功能。作出了可以用组合逻辑实现该电路的结论之后，就可以利用本节讨论的方法进行电路设计。

应该指出的是，可以用组合逻辑实现电路，并不意味着只能用组合逻辑实现。事实上，对同一个电路有可能既可以用组合逻辑实现，又可以用时序逻辑实现。

6.1.1 系统级的组合逻辑电路设计

如果输入信号的个数不太多，组合逻辑电路设计的最常用的方法是写出真值表。真值表将电路输入的所有可能组合列出，并给出对应各种输入组合情况下的输出值。上面例 6.1 的真值表如表 6.1 所示。

表 6.1 二进制比较器的真值表

输入				输出				
N1	N0	M1	M0	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

在本小节中，我们讨论将真值表变换为行为域中的算法描述的两种方法。第一种方法是数组式模型，它可以直接映射为 ROM 硬件实现；第二种方法是 case 条件语句模型，它可以直接映射为多路选择器。通过优化过程，还可以用其他多路选择器结构实现真值表。

1. 第一种方法：数组模型

下面的 VHDL 源代码实现了表 6.1 描述的硬件行为。在这个 VHDL 模型中，为了方

```
package body TRUTH4x5 is
    function INTVAL(VAL: ADDR) return INTEGER is
        variabe SUM : INTEGER := 0;
    begin
        for N in VAL'Low to VAL'Hifh loop
            if VAL(N) = '1' then
                SUM := SUM + (2 * * N);
            end if;
        end loop;
    end INTVAL;
end TRUTH4x5;

use work. TRUTH4x5 all
architecture TABLE of COM is
begin
    process(N1,N0,M1,M0)
        variable INDEX: INTEGER;
        variable WOUT: WORD;
    begin
        INDEX := INTVAL (N1&N0&M1&M0);
        WOUT := TRUTH(INDEX);
        GE <= WOUT(4) after DEL;
        LE <= WOUT(3) after DEL;
        E <= WOUT(2) after DEL;
        G <= WOUT(1) after DEL;
        L <= WOUT(0) after DEL;
    end process;
end TABLE;
```

函数 INTVAL 通过对一个位矢量从左向右扫描，确定它的哪一位为'1'，按每位所对应的权重因子把它转换为十进制整数。在函数 INTVAL 中，变量 SUM 用来存储部分和，它被初始化为'0'。信号属性 VAL'Low 等于位矢量下标的最小值(0)；信号属性 VAR'High 等于位矢量下表范围的最大值(4)。这里使用信号属性而不使用常数的作用在于：写出的函数 INTVAL 是一个通用转换函数，它可以把任意长度的位矢量转换为整数。

实体 COM 的结构体名为 TABLE，它由一个进程组成，在任何一个输入信号发生变化时，进程都会被激活。进程的行为只是简单查表。在进程中，首先用函数 INTVAL 把输入信号转换为相应的整数，用此整数作为下标在数组 TRUTH 中选择适当的输出值 WOUT，经延时之后把 WOUT 的值赋给输出信号。

在自动综合程序中，很容易将这个 VHDL 数组模型映射为一个 ROM，ROM 中存储

的内容是常数 TRUTH 中的数值，器件 COM 的 4 个输入信号连接到 ROM 的地址输入，器件 COM 的输出连接到 ROM 的数据输出端，这需要规模为 16×5 的存储区。这样由 ROM 实现的二进制比较器电路如图 6.2 所示。

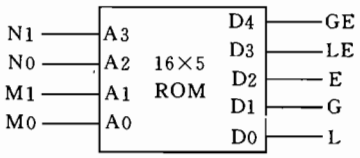


图 6.2 直接将真值表数组转换为 ROM

2. 第二种方法：case 条件语句模型

在这种 VHDL 行为模型中，真值表中的每一行对应于 case 条件语句的一种选择条件。下面 VHDL 源代码是二进制比较器电路的 case 条件语句模型。

```
architecture MUX of COM is
begin
    process(N1,N0,M1,M0)
    begin
        case N1&N0&M1&M0 is
            when "0000"=>GE <='1' after DEL; LE<='1' after DEL;
                E<='1' after DEL; G<='0' after DEL; L<='0' after DEL;
            when "0001"=>GE <='0' after DEL; LE<='1' after DEL;
                E<='0' after DEL; G<='0' after DEL; L<='1' after DEL;
            when "0010"=>GE <='0' after DEL; LE<='1' after DEL;
                E<='0' after DEL; G<='0' after DEL; L<='1' after DEL;
            when "0011"=>GE <='0' after DEL; LE<='1' after DEL;
                E<='0' after DEL; G<='0' after DEL; L<='1' after DEL;
            when "0100"=>GE <='1' after DEL; LE<='0' after DEL;
                E<='0' after DEL; G<='1' after DEL; L<='0' after DEL;
            when "0101"=>GE <='1' after DEL; LE<='1' after DEL;
                E<='1' after DEL; G<='0' after DEL; L<='0' after DEL;
            when "0110"=>GE <='0' after DEL; LE<='1' after DEL;
                E<='0' after DEL; G<='0' after DEL; L<='1' after DEL;
            when "0111"=>GE <='0' after DEL; LE<='1' after DEL;
                E<='0' after DEL; G<='0' after DEL; L<='1' after DEL;
            when "1000"=>GE <='1' after DEL; LE<='0' after DEL;
                E<='0' after DEL; G<='1' after DEL; L<='0' after DEL;
            when "1001"=>GE <='1' after DEL; LE<='0' after DEL;
                E<='0' after DEL; G<='1' after DEL; L<='0' after DEL;
            when "1010"=>GE <='1' after DEL; LE<='1' after DEL;
                E<='1' after DEL; G<='0' after DEL; L<='0' after DEL;
            when "1011"=>GE <='0' after DEL; LE<='1' after DEL;
                E<='0' after DEL; G<='0' after DEL; L<='1' after DEL;
            when "1100"=>GE <='1' after DEL; LE<='0' after DEL;
```

```
            E<='0' after DEL; G<='1' after DEL; L<='0' after DEL;
        when "1101"=>GE <='1' after DEL; LE<='0' after DEL;
            E<='0' after DEL; G<='1' after DEL; L<='0' after DEL;
        when "1110"=>GE <='1' after DEL; LE<='0' after DEL;
            E<='0' after DEL; G<='1' after DEL; L<='0' after DEL;
        when "1111"=>GE <='1' after DEL; LE<='1' after DEL;
            E<='1' after DEL; G<='0' after DEL; L<='0' after DEL;
    end case;
end process;
end MUX;
```

与结构体TABLE类似，这个结构体MUX同样由一个进程组成，当输入信号N1，N0,M1,M0之一发生变化时，进程被激活。整个进程利用一条case条件语句，对应于输入信号的16种不同组合，为输出信号赋不同的值。这里case语句的条件以及为信号赋的值直接从硬件的指标推出，即直接从真值表推导出。

二进制比较器的这种VHDL模型可以直接转换为多路选择器。图6.3是用多路选择器实现的输出信号GE。四个输入信号连接在多路选择器的地址输入端，把N1连接到地址输入的最高位，M0连接到地址输入的最低位。连接到多路选择器的数据输入端的数据是case条件语句中为GE所赋的数值。例如：在case条件语句中，如果输入为"0101"，则为GE赋值'1'，所以在多路选择器中，数据位D5='1'。再利用另外四个多路选择器，可以类似地用硬件实现输出信号LE,E,G和L。

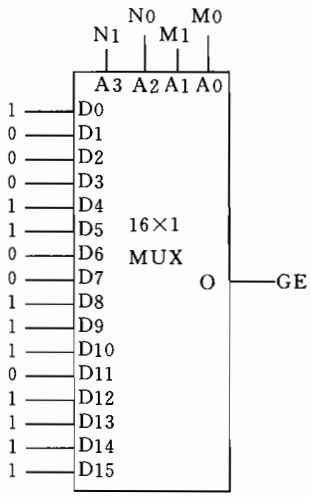


图6.3 把case条件语句式的VHDL模型直接转换为硬件多路选择器

3. VHDL算法模型的优化

前面用多路选择器实现的输出信号GE是直接将case条件语句转换为硬件，事实上，用多路选择器实现这样的输出，也可能存在更有效的方法。通常可以先对硬件的VHDL模型进行一些变换，再将变换后的算法模型转换为硬件，这样的变换过程称为VHDL模型的优化。真正的优化要利用设计者的知识，从而需要人工智能的方法才能自动完成。本节我们只考虑特殊情况，即对这个用多路选择器实现case条件语句式的VHDL模型的特例进行优化。

将某个输入信号从多路选择器的地址输入端消除，是实现VHFDL模型优化的最简单的方法。为了说明这种方法，我们任意选择一个输入信号，不失一般性，可以选择M0，把它从多路选择器的地址输入端消去。优化后的VHDL算法模型如下：

```
architecture MUX3 of COM is
begin
```

```
    process(N1,N0,M1,M0)
    begin
        case N1&N0&M1 is
            when "000"=>GE<= not M0 after DEL; LE<='1'after DEL;
              E<=not M0 after DEL; G<='0' after DEL; L<=M0 after
                DEL;
            when "001"=>GE <='0' after DEL; LE<='1' after DEL;
              E<='0' after DEL; G<='0' after DEL; L<='1' after DEL;
            when "010"=>GE <='1' after DEL; LE<=M0 after DEL;
              E<=M0 after DEL; G<=not M0 after DEL; L<='0' after
                DEL;
            when "011"=>GE <='0' after DEL; LE<='1' after DEL;
              E<='0' after DEL; G<='0' after DEL; L<='1' after DEL;
            when "100"=>GE <='1' after DEL; LE<='0' after DEL;
              E<='0' after DEL; G<='1' after DEL; L<='0' after DEL;
            when "101"=>GE<=not M0 after DEL; LE<='1' after DEL;
              E<=not M0 after DEL; G<='0' after DEL; L<=M0 after
                DEL;
            when "110"=>GE <='1' after DEL; LE<='0' after DEL;
              E<='0' after DEL; G<='1' after DEL; L<='0' after  DEL;
            when "111"=>GE <='1' after DEL; LE<=M0 after DEL;
              E<=M0 after DEL; G<=not M0 after DEL; L<='0' after
                DEL;
        end case;
    end process;
end MUX3;
```

为了说明如何得到这种优化后的 VHDL 模型，我们来看结构体 MUX 中 case 语句的两个条件"0000"和"0001"。现在我们的目的是把条件中的 M0 项消除，用同一个条件 N1&N0&M1="000"来代替这两个条件。对于每个输出变量(GE,LE,E,G 和 L)，在优化前的结构体 MUX 中，对应于信号 M0 的不同值，对输出信号的赋值可能出现下面 4 种可能情况：

(1) 对于 M0 的两种不同值，输出信号的值总与 M0 相同。这时可以把输出信号的值赋为 M0。

(2) 对于 M0 的两种不同值，输出信号的值总与 M0 不同。这时可以把输出信号的值赋为 not M0。

(3) 对于 M0 的两种不同值，输出信号的值总为'1'。这时可以把输出信号的值赋为'1'。

(4) 对于 M0 的两种不同值，输出信号的值总为'0'。这时可以把输出信号的值赋

为'0'。

在优化前的 VHDL 行为描述 MUX 中，在输入信号为"0000"时(这时 M0 为'0')，为信号 GE 赋值'1'；在输入信号为"0001"时(这时 M0 为'1')，为信号 GE 赋值'0'。所以对于输出 GE，可以把结构体 MUX 中的 case 语句中的"0000"和"0001"两种条件简化为结构体 MUX3 中的一种条件，即输入为 N1&N0&M1="000"时，为 GE 赋值 not M0。对于输出信号 LE，在输入信号为"0000"和"0001"两种情况下都是对它赋值为'1'，所以可以把优化前的两种条件简化为一种条件，即输入信号为 N1&N0&M1="000"时，为信号 LE 赋值'1'。与此类似，可以对其他输出信号(E,G 和 L)作同样的优化。对于优化前的 VHDL 模型中的其他各对条件("0010"和"0011","0100"和"0101"等等)，进行类似的优化，就可以得到如结构体 MUX3 所示的优化后的 VHDL 行为模型。

优化后的 VHDL 模型同样可以转换为多路选择器，图 6.4 是把结构体 MUX3 中的输出信号 GE 的行为描述转换为多路选择器后的硬件实现。这里把输入信号 N1,N0 和 M1 映射到多路选择器的地址输入端，同样根据 case 条件语句中的赋值内容确定多路选择器的数据输入。比如 case 语句中输入条件为"000"时为信号 GE 赋值 not M0，所以应将 M0 反相后接到数据端 D0。再比如，case 语句中输入条件为"110"时为信号 GE 赋值'1'，所以应该将常数'1'连接到数据输入端 D6。采用与此类似的硬件结构，可以实现二进制比较器的其他四个输出信号 LE、E、G 和 L。

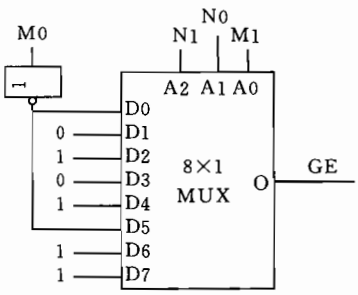


图 6.4 将优化后的 case 条件语句式 VHDL 模型转换为多路选择器

这里介绍的器件模型是 VHDL 算法模型，优化也是对硬件的 VHDL 模型进行，所以有可能写出通用的程序，对 case 条件语句式的硬件模型进行优化，以得到优化后的 VHDL 硬件模型。

6.1.2 组合逻辑电路的行为域数据流式模型

根据第 1 章中关于数字集成电路设计方法学的讨论，设计者通常需要把硬件的算法式行为描述转换为数据流描述。在完成这一转换的过程中，通常要对模型进行优化。在设计变量个数较少时，设计者可以应用卡诺图对硬件模型进行优化，一般认为设计变量个数小于 7 时，可以使用卡诺图。当设计变量个数较多时，应该采用适宜于编程的算法。比如 Quine-McClusky 法，它适宜于设计变量个数为 7～20 的情况。通常也要利用设计者的知识对各设计变量之间的关系进行分析，以简化设计内容。

1. 设计的分解

对于例 6.1 讨论的二进制比较器，可以看出电路的 5 个输出信号并不是完全独立的。这种优化工作主要靠设计者根据经验作出，而一般自动设计工具还做不到这一点。通过对该电路的五个输出进行分析，知道输出信号 E、G 和 L 可以通过信号 GE 和 LE 导出。它们之间的关系如下：

E = GE and LE

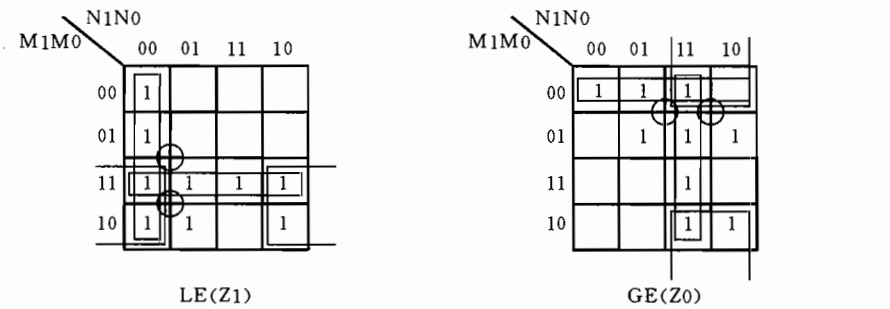
G = GE and (not LE)

L = LE and (not GE)

因此，只需显式确定出信号 GE 和 LE 与输入信号的关系。

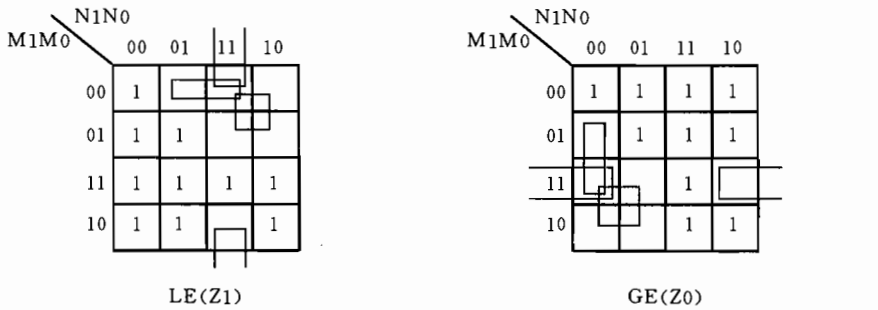
2. 硬件行为数据流描述的综合

理论上，根据真值表就可以直接得到组合逻辑电路的数据流模型，也可以直接得到组合逻辑电路的逻辑门级的结构描述。把真值表中所有乘积项相加，就可以完成这种 VHDL 模型。但是，这样得到的电路可能非常不经济。在变量个数不太大的情况下，通常使用卡诺图进行优化。为了说明如何用卡诺图进行优化，我们用卡诺图对二进制比较器电路 COM 的数据流模型进行优化。图 6.5 是优化后的用“乘积项之和”表示的输出信号 GE 和 LE 的逻辑表达式。图 6.6 是优化后的用“和项之积”表示的输出信号 GE 和 LE 的逻辑表达式。由于“和项之积”表示的逻辑关系在此例中使用的门数较少，且使用的逻辑的输入端个数也较少，所以在后面的讨论中经常将使用“和项之积”表示的逻辑关系。



$Z1=\overline{N1}\ \overline{N0}+\overline{N1}M0+\overline{N1}M1+M1M0+\overline{M1}N0$　$Z0=\overline{M1}\ \overline{M0}+N1\overline{M0}+N1\overline{M1}+N1N0+\overline{M1}N0$

图 6.5　优化后的两级“乘积项之和”的逻辑表达式



$Z1=N0\overline{M1}\overline{M0}+N1\overline{M1}+N1N0\overline{M0}$　　$Z1=(\overline{N0}+M1+M0)(\overline{N1}+M1)(\overline{N1}+\overline{N0}+M0)$

$Z0=\overline{N1}\overline{N0}M0+\overline{N1}M1+\overline{N0}M1M0$　　$Z0=(N1+N0+\overline{M0})(N1+\overline{M1})(N0+\overline{M1}+\overline{M0})$

图 6.6　优化后的两级“和项之积”的逻辑表达式

根据输出信号的逻辑表达式，很容易得出硬件的 VHDL 数据流模型。下面的 VHDL

源代码就是根据优化后的“和项之积”的逻辑表达式得出的二进制比较器的数据流式模型。按当前的技术水平，逻辑优化还不能完全自动完成。对于变量个数较少的问题，自动逻辑优化程序可能得出最优的结果，但对于规模较大的问题，只能通过设计者进行人工干预才能得出令人满意的结果。

```
entity COM is
    generic (DEL: TIME)
    port (N1,N0,M1,M0: in Std_logic; GE,LE,E,G,L: out Std_logic);
end COM;

architecture POSDF of COM is
begin
    process (N1,N0,M1,M0)
        signal Z1,Z0: Std_logic;
    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);
        GE<=Z0 after DEL;
        LE<=Z1 after DEL;
        E<=Z1 and Z0 after DEL;
        G<=Z0 and not Z1 after DEL;
        L<=Z1 and not Z0 after DEL;
    end process;
end POSDF;
```

根据“乘积项之和”或“和项之积”的逻辑表达式，可以得出电路的其他形式的 VHDL 数据流模型。比如，对逻辑表达式按如下规则进行变换，也可以得到一种形式的 VHDL 数据流模型，这个模型中不使用与门，而使用或非门。这个规则如下：

(1) 在每个“和项”前，即每个括号前插入一个算子 not。

(2) 把每个与算子 and 换成或算子 or，并在整个逻辑表达式前增加一个算子 not。

下面的 VHDL 源代码是把结构体 POSDF 按上述规则变换后得到的电路 COM 的数据流模型。读者可以对这两个模型进行仿真，以验证两个模型行为的一致性。

```
entity COM is
    generic (DEL: TIME)
    port (N1,N0,M1,M0: in Std_logic; GE,LE,E,G,L: out Std_logic);
end COM;
```

```
architecture NORDF of COM is
begin
    process (N1,N0,M1,M0)
        signal Z1,Z0: Std_logic;
    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) );
        GE<=Z0 after DEL;
        LE<=Z1 after DEL;
        E<=Z1 and Z0 after DEL;
        G<=Z0 and not Z1 after DEL;
        L<=Z1 and not Z0 after DEL;
    end process;
end NORDF;
```

6.1.3 组合逻辑电路的门级结构域综合

本小节讨论把组合逻辑电路的数据流模型转换为逻辑门级结构描述。这一过程相对规则，容易编制程序实现自动综合。

一般地讲，给定一个硬件的数据流式模型，可以把它转换为不同的几个逻辑门级结构模型。比如，对于上面讨论的二进制比较电路的“乘积项之和”式的数据流模型，可以被直接转换为两级“与-或电路”、两级“与非-与非电路”、两级“或-与非电路”或其他多级电路实现。同样，对于上述电路的“和项之积”式的数据流模型也可以直接转换为两级“或-与电路”、两级“或非-或非电路”、两级“与-或非电路”或其他多级电路实现。

从上面 6.1.2 节讨论的二进制比较电路的数据流模型 POSDF 出发，简单地把每个算子 and 换作 2 输入与门、把每个算子 or 换作 2 输入或门，就可以得到如图 6.7 所示的电路。

为了验证这样的逻辑门级结构模型的行为正确，可以写出下面 VHDL 源代码所示的结构式模型。在二进制比较器的这个结构式模型中，电路用基本逻辑门互连构成，基本逻辑门由行为描述。整个器件的延时转换为多个逻辑门的延时。对这一层次的硬件模型进行仿真，可以得到比算法模型精确的时序信息，仿真结束后，可以根据仿真结果把时序信息反向标注到硬件的算法式模型中。如果将此二进制比较器用于其他电子系统设计，在高层次仿真时，可使用该器件的算法模型。由于算法模型中的时序信息是根据逻辑门级仿真结果反向标注得到，所以模型的精度与门级精度相同，但算法模型的仿真效率会比门级模型高很多。

```
use work.all;
entity COM is
```

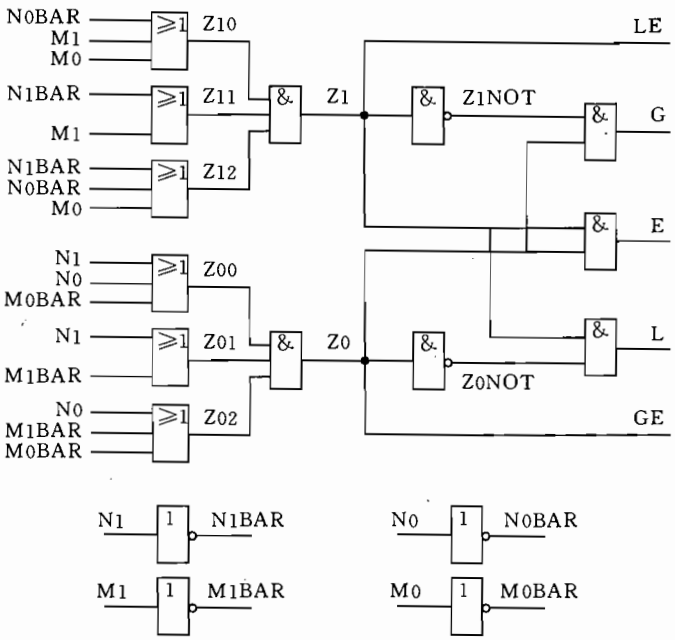


图 6.7　数据流模型直接转换为逻辑门级模型:“两级或-与”电路实现

```
    port (N1,N0,M1,M0: in Std_logic;GE,LE,E,G,L: out Std_logic);
end COM;

architecture TWO_LEVEL_OR_AND of COM is
    signal Z10,Z11,Z12,Z00,Z01,Z02: Std_logic;
    signal N0BAR,N1BAR,M0BAR,M1BAR: Std_logic;
    signal Z0,Z1,Z0BOT,Z1NOT: Std_logic;
    component NOT2G
        generic(DEL: TIME);
        port(I: in Std_logic; O: out Std_logic);
    end component;
    for all: NOT2G use entity NOT2(BEHAVIOR);
    component AND2G
        generic(DEL: TIME);
        port(I1,I2: in Std_logic; O: out Std_logic);
    end component;
    for all: AND2G use entity AND2(BEHAVIOR);
    component AND3G
        generic(DEL: TIME);
        port(I1,I2,I3: in Std_logic; O: out Std_logic);
```

```
    end component;
    for all: AND3G use entity AND3(BEHAVIOR);
    component OR2G
        generic(DEL: TIME);
        port(I1,I2: in Std_logic; O: out Std_logic);
    end component;
    for all: OR3G use entity OR2(BEHAVIOR);
    component OR3G
        generic(DEL: TIME);
        port(I1,I2,I3: in Std_logic; O: out Std_logic);
    end component;
    for all: OR3G use entity OR3(BEHAVIOR);
    component WIREG
        port(I: in Std_logic; O: out Std_logic);
    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);
    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;

entity NOT2 is
    generic(DEL: TIME)
```

```
    port(I: in Std_logic; O: out Std_logic);
end NOT2;

architecture BEHAVIOR of NOT2 is
begin
    O<=not I after DEL;
end BEHAVIOR;

entity AND2 is
    generic(DEL: TIME)
    port(I1,I2: in Std_logic; O: out Std_logic);
end AND2;

architecture BEHAVIOR of AND2 is
begin
    O<=I1 and I2 after DEL;
end BEHAVIOR;

entity AND3 is
    generic(DEL: TIME)
    port(I1,I2,I3: in Std_logic; O: out Std_logic);
end AND3;

architecture BEHAVIOR of AND3 is
begin
    O<=I1 and I2 and I3 after DEL;
end BEHAVIOR;

entity OR2 is
    generic(DEL: TIME)
    port(I1,I2: in Std_logic; O: out Std_logic);
end OR2;

architecture BEHAVIOR of OR2 is
begin
    O<=I1 or I2 after DEL;
end BEHAVIOR;
```

```
entity OR3 is
    generic(DEL: TIME)
    port(I1,I2,I3: in Std_logic; O: out Std_logic);
end OR3;

architecture BEHAVIOR of OR3 is
begin
    O<=I1 or I2 or I3 after DEL;
end BEHAVIOR;

entity WIRE is
    port(I: in Std_logic; O: out Std_logic);
end WIRE;

architecture BEHAVIOR of WIRE is
begin
    O<=I;
end BEHAVIOR;
```

6.1.4 组合逻辑电路设计方法小结

图 6.8 总结了组合逻辑电路设计的一般过程，并表明了 VHDL 模型在组合逻辑电路设计中的用途。从电路性能指标的文字描述开始，首先构造出电路输入输出关系的真值

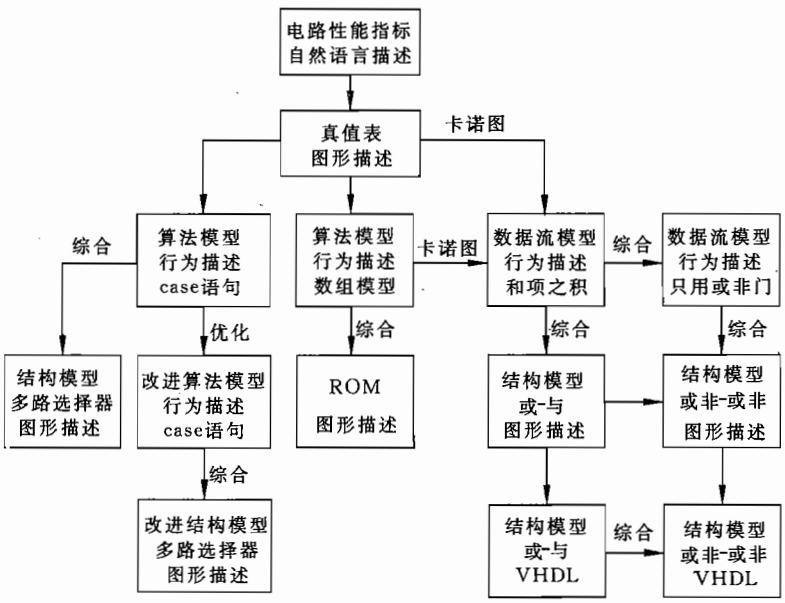


图 6.8 组合逻辑电路设计过程小结

表，这是电路行为的图形描述。利用行为域的算法模型，可以对真值表描述的电路行为进行验证。根据真值表可以建立行为域的两种算法模型：一种是数组式算法模型，另一种是case条件语句式算法模型。数组模型可以直接转换为ROM式硬件实现，而case语句则可以用多路选择器实现。对case语句式算法模型进行简化，可以对多路选择器进行优化。

利用卡诺图，或采用如Quine-McClusky方式的系统化方法，可以把行为域的算法模型转换为行为域的数据流模型。在对数据流模型进行进一步处理之前，可以用仿真的方法对模型进行验证。

行为域的数据流模型可以直接转换为逻辑门级的结构式模型。对于结构式模型同样可以进行仿真，以在逻辑门级验证电路行为的正确性。对于逻辑门级仿真得到的时序信息，可以反向标注到硬件的高层次模型，比如行为域的算法模型或数据流模型。这使得高层次仿真可得到精确的时序信息。

6.2 时序逻辑电路设计

本节综合使用状态表和硬件描述语言设计时序逻辑电路。设计过程包括了集成电路设计的不同层次。

与组合逻辑电路设计类似，仍从电路的自然语言描述开始讨论。我们面对的问题是设计一个串并转换电路。图6.9是这个电路的方框图，输入信号CLK是控制整个系统工作的时钟；输入R是系统复位信号，如果R='1'，则在R为'1'后的时钟结束时，系统进入复位态；输入信号A是同步信号，输入D是串行输入信号。同步信号A在数据输入端D出现数据前一个周期出现，即在A置位为'1'后一个周期后，D上连续出现串行数据。该串并转换电路把相继4位串行输入数据转换为4位并行数据送到输出端口Z。当并行数据输出端口Z上有数据输出时，信号DONE保持为'1'。器件输出数据的时间要持续一个完整的时钟周期，即信号DONE也要保持一个完整时钟周期。图6.9也给出了该器件的时序关系。在并行数据输出端Z输出数据的时钟周期内，器件的同步输入端A可以接收下一个同步脉冲，即器件输出并行数据的下一个时钟周期就可以继续接收新的串行输入数据。如果有数据输入，则器件继续接收数据；如果没有数据输入，器件进入复位态等待新数据出现。

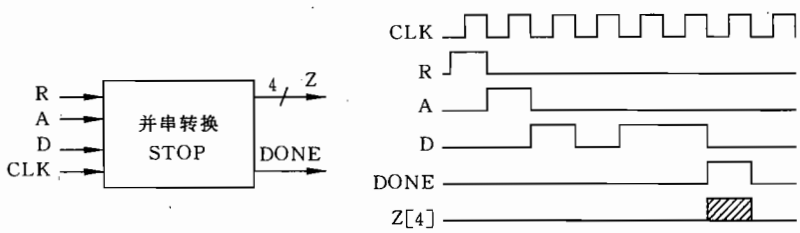


图6.9 串并转换电路的方框图及时序关系

6.2.1 选择 Moore 状态机还是 Mealy 状态机

时序电路的通用模型是有限状态机，有限状态机设计的第一步是决定采用 Moore 状态机还是采用 Mealy 状态机。读者可能对 Moore 状态机和 Mealy 状态机的特性已经很熟悉，这里只简单列出构造状态表时要用到的内容。根据定义 Moore 状态机的输出只与状态机的状态有关，与输入信号的当前值无关。Mealy 状态机的输出不单与状态有关，而且与输入信号的当前值有关。从实现电路功能的角度来讲，这两种状态机都可以实现同样的功能。但它们的输出时序不同，在选择使用哪种状态机时要根据实际情况进行具体分析。Moore 状态机和 Mealy 状态机之间的主要区别如下。

Moore 状态机在时钟脉冲的有效边沿后的有限个门延时之后，输出达到其稳定值。输出会在一个完整的时钟周期内保持其稳定值，即使在该时钟周期内输入信号有变化，输出也不会发生变化。输入对输出的影响要到下一个时钟周期才能反映出来。把输入与输出隔离开来，是 Moore 状态机的一个重要特点。

Mealy 状态机由于输出直接受输入影响，且输入变化可能出现在时钟周期内的任何时刻，这就使得 Mealy 状态机对输入的响应可以比 Moore 状态机对输入的响应早一个时钟周期。输入信号中的噪声可能出现在输出端。

实现同样的功能，Moore 状态机所需的状态个数可能比 Mealy 状态机多。

通常，对于具体的一个电路的指标规范，可能适合于用 Moore 状态机实现，或者适合于用 Mealy 状态机实现，也有可能用两种状态机实现都很合适，硬件设计者需要自行决定采用哪种状态机。对于这里讨论的串并转换电路，在串行信号的最后一位输入后就可以产生输出，而当输出并行数值时输入信号不再存在，所以输出不能直接与输入关联。同时由于输出要保持一个完整的时钟周期，在此时钟周期内输出不能受输入信号的影响，所以不适合采用 Mealy 状态机，最好采用 Moore 状态机。

6.2.2 构造状态表

确定了采用 More 状态机实现硬件，接下来的问题就是构造状态表。构造状态表的工作需要充分利用硬件设计者的设计经验。对于同一个设计问题，不同的设计者可能构造出不同的状态表，这些状态表都可以很好地完成硬件要实现的功能。虽然不可能找到一个系统化构造状态表的成熟算法，但如果设计者遵循结构化设计的思想，则容易构造出效果好的状态表。

为了清楚地描述状态转换关系，设计者可以采用状态转换图，也可以采用状态转换表。状态转换图直观地给出了状态之间的转换关系以及状态转换条件，因而容易理解状态机的工作机理，适合于状态个数不太多的情况。状态转换表则采用表格的方式列出了状态及转换条件，适合于状态个数较多的情况。

6.2.3 建立状态转换图

对于这里要讨论的串并转换电路，由于电路相对简单，状态的个数不会太多，因而选

择采用状态转换图的方式列出状态及其转换条件。构造状态转换图时，通常从一个比较容易描述的状态开始，如果硬件的指标描述中规定了复位状态，这通常是构造状态图的很好的起始态。最好是在建立每个状态时，都清楚地写出关于这个状态的文字描述，这为硬件设计过程提供了清晰的参考材料，也为最后完成的设计提供了完整的设计文档。通常，设计过程中可能会有些反复，在开始设计时写出的状态功能描述可能会随设计过程的深入作必要的修正，某些状态的最后描述可能会与开始设计时写出的描述差别很大，完整的文字描述有利于设计过程的进行。

对于串并转换电路，我们从复位态开始设计，令复位态为 S0。该状态的文字描述如下：

状态 S0：复位状态。

如果输入 R='1'，不论输入数据为什么数值，在相应的时钟周期的下降沿，电路都会进入这一状态。器件会保持在这一状态直到输入信号 A='1'。在状态 S0 中，输出信号 DONE='0'，这意味着并行输出信号 Z 上的值是无效信号，这时无论 Z 为何值，都应该忽略它。

如果当前态是 S0，且输入 R='1'，不论其他输入为什么数值，它会保持在 S0；如果 R='0' 且 A='0'，它也会保持在 S0。信号保持在 S0 的条件如下：

$$R \text{ or } ((\text{not } A) \text{ and } (\text{not } R)) = R \text{ or } (\text{not } A)$$

如果在某个时钟周期有 R='0' 且 A=1，则器件在后面的 4 个时钟周期中接收外部串行输入数据，这意味着硬件会进入一个新状态，将这一状态记作 S1。后面我们会给出状态 S1 的描述，并依次构造该硬件的其他状态。

在构造状态表的过程中，状态转换图是一种状态及其转换关系的直观表示方法。状态转换图是一个有向图，其节点表示器件的一个状态，节点符号的内部写出了该状态的名称。对于 Moore 状态机，节点符号内部还标出了器件的输出。有向图中的支路表示了状态之间的转换关系，图中支路上标出了状态转换的条件。图 6.10 是根据上述描述得出的状态转换图，图中状态符号 S0 中标出的输出值 0—表示 DONE 为'0'，Z 的值无意义。从状态 S0 指向 S1 的弧线上指明了从状态 S0 转换为 S1 的条件为 (A and not R)，其意义是在 R='0' 同时出现 A='1' 的时钟周期，硬件进入状态 S1 以在状态 S1 的输出值仍为 0—，即 DONE='0'，Z 无意义。从状态 S0 指向 S0 的的弧线上标明了条件，其意义是在输入 R='1' 的条件下返回状态 S0。状态 S1 的文字描述如下：

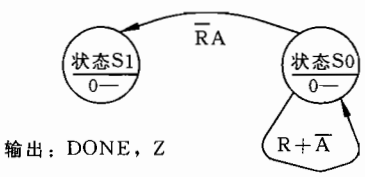


图 6.10　构造串并转换电路的状态转换图的第一步

状态 S1：接收第 1 位串行输入数据。

如果当前状态为 S0，则当 R='0' 且 A='1' 时，器件进入这一状态。在这个状态中，器件可以逐个接收外部第 1 位串行数据，并在数据出现的时钟周期的下降沿把它存储起来，留作以后输出。这个状态中输出 DONE='0'，输出 Z 无意义。如果在状态 S1 中，出现 R='1' 的情况，则器件被复位至状态 S0；否则器件会依次进入接收第 2 位串行输入数据的状态 S2。

对于构造出的每一个新状态，必须对所有可能的输入条件进行分析，以确定在哪些条

件下从哪些状态可以转换到新状态，确定在哪些条件下从新状态可以转换到哪些状态。在状态图中增加相应的节点和支路，在每个节点上标出状态名和输出值，在每个支路上标出转换条件，就可以得出新的状态转换图。图 6.11 是对状态 S1 进行分析后构造出的状态转换图。通常对于给定的电路指标要求，可以在有限的状态内完成所有的操作，因此，通常把这样的硬件称作有限状态机。事实上，有可能某些电路的数据操作不能在有限的状态内完成，确定能否在有限个状态内完成规定的数据操作是一个非常困难的问题。本书不讨论这个问题。

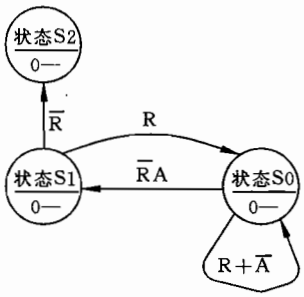


图 6.11　构造串并转换电路的状态转换图的第二步

对状态 S2 的情况进行分析，并依次对产生的新状态进行分析，可以得出对各个状态的文字描述如下：

状态 S2：接收第 2 位串行输入数据。

在状态 S1，如果输入 R＝'0'，则下一个时钟周期第 2 位串行数据应该出现在数据输入端，器件进入状态 S2。在状态 S2，输出信号 DONE＝'0'，数据输出 Z 无意义。如果在状态 S2 中，出现 R＝'1'的情况，则器件被复位至状态 S0，否则，器件会依次进入接收第 2 位串行输入数据的状态 S3。

状态 S3：接收第 3 位串行输入数据。

在状态 S2，如果输入 R＝'0'，则下一个时钟周期第 3 位串行数据应该出现在数据输入端，器件进入状态 S3。在状态 S3，输出信号 DONE＝'0'，数据输出 Z 无意义。如果在状态 S3 中，出现 R＝'1'的情况，则器件被复位至状态 S0，否则，器件会依次进入接收第 4 位串行输入数据的状态 S4。

状态 S4：接收第 4 位串行输入数据。

在状态 S3，如果输入 R＝'0'，则下一个时钟周期第 4 位串行数据应该出现在数据输入端，器件进入状态 S4。在状态 S4，输出信号 DONE＝'0'，数据输出 Z 无意义。如果在状态 S4 中，出现 R＝'1'的情况，则器件被复位至状态 S0，否则，器件会依次进入把 4 位串行数据并行输出的状态 S5。

状态 S5：把 4 位串行输入数据并行输出。

在状态 S4，如果输入 R＝'0'，则下一个时钟周期器件进入状态 S5。在状态 S5，输出信号 DONE＝'1'，数据输出 Z 为 4 位串行输入数据。如果在状态 S5 中，出现 R＝'1'的情况，则器件被复位至状态 S0；若不出现信号 A，即 A＝'0'，器件也进入状态 S0。

在状态 S5，如果复位信号 R＝'0'且出现 A＝'1'，则器件会进入状态 S1，即准备接收下一组数据。这时我们重复使用状态 S1，而不是产生一个新状态，这样就完成了整个状态转换图，如图 6.12 所示。由于重复使用了状态 S1，所以应该修改状态 S1 的文字描述，以反映从状态 S5 转换到 S1 的情况。修改后的状态 S1 的文字描述如下：

状态 S1（修改后）：接收第 1 位串行输入数据。

如果输入条件为 R＝'0'且 A＝'1'，器件可以从状态 S0 或 S5 进入状态 S1。在这个状态中，器件可以逐个接收外部第 1 位串行数据，并在数据出现的时钟周期的下降沿把它存储起来，留作以后输出。这个状态中 DONE＝'0'，输出 Z 无意义。如果在状态 S1 中，出现 R＝'1'的情况，则器件被复位至状态 S0，否则器件会依次进入接收第 2 位串行输入数据的状态 S2。

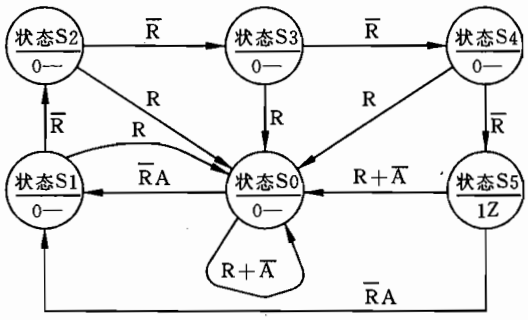


图 6.12　串并转换电路的状态转换图

6.2.4　状态转换表

上面讨论的状态转换图清楚地表明了状态间的转换关系及转换条件，给定了有关的输入序列，很容易根据状态转换图跟踪状态之间的转换关系。但是，对于复杂电路，状态的个数可能很多，状态图变得很复杂，不易画出，也不易读懂。这种情况下有效的状态描述方法是状态转换表。状态转换表的建立过程与状态转换图类似，只是用表格的方式描述了状态间的转换关系即转换条件。表 6.2 是串并转换电路的状态转换表。

表 6.2　串并转换电路的状态转换表

当前状态	转换条件	下一状态	数据转换	输出值
S0	R or (not A)	S0	无	DONE='0',Z 无意义
	(not R) and A	S1	无	
S1	not R	S2	数据移位,存储数据	DONE='0',Z 无意义
	R	S0	无	
S2	not R	S3	数据移位,存储数据	DONE='0',Z 无意义
	R	S0	无	
S3	not R	S4	数据移位,存储数据	DONE='0',Z 无意义
	R	S0	无	
S4	not R	S5	数据移位,存储数据	DONE='0',Z 无意义
	R	S0	无	
S5	(not R) and A	S1	无	DONE='1',Z 为并行输出数据
	R or (not A)	S0	无	

6.2.5　互补原则

在构造硬件的状态转换表或状态转换图时，互补原则可以帮助设计者检查设计过程

中是否出现了错误。所谓互补原则指的是状态转换图中离开一个节点的所有支路上所标出的条件必须互补，即从一个给定状态转换到其他状态的所有输入条件必须互补。如果从一个状态转换到两个不同状态的条件可以同时为逻辑 1，则意味着硬件可能同时进入两个不同状态，显然这种情况不允许出现在硬件中。例如对于图 6.12 中的状态 S0，状态转换图中离开节点 S0 的两个条件分别为 R or (not A)和 (not R) and A，这两个条件的与、或结果为

(R or (not A)) and ((not R) and A)=0, (R or (not A)) or ((not R) and A)=1

这就验证了离开节点 S0 的两个条件的互补性。

在构造状态转换图时，可以应用这个互补原则检查状态图中是否存在错误。显然状态转换表也满足互补原则，也可以用这一互补原则检查状态转换表中可能出现的错误。

6.2.6 建立状态机的 VHDL 模型

状态转换图和状态转换表都可以用来建立 VHDL 模型，VHDL 模型可以用来验证硬件的功能，以便于在进一步进行较低层次的设计之前检查可能出现的错误。建立 VHDL 模型时，通常把硬件划分为如图 6.13 所示的结构，即将硬件划分为数据单元、控制单元和一个输出单元。在 Moore 状态机中，输出值与控制单元的状态有关，也与存储在数据单元中的数据值有关。在 Mealy 状态机中，输出还可能与输入信号有关。

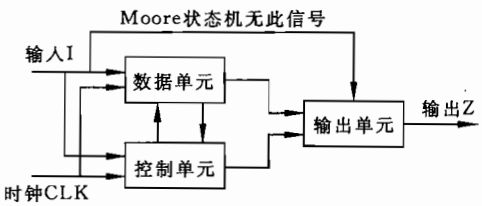


图 6.13 状态机模型的方框图

根据串并转换电路的状态转换表或状态转换图，容易得出下面 VHDL 源代码所示的模型。实体 STOP 定义了硬件的接口，Std_logic 型输入信号为复位信号 R、同步信号 A、数据输入 D 和时钟信号 CLK，输出信号为 DONE 和 Z，其中 Z 是数据宽度为 4 的矢量。

```
entity STOP is
    port (R,A,D,CLK: in Std_logic; done: out Std_logic;
    Z:Std_logic_vector(3 down to 0) )
endSTOP;

architecture FSM_RTL of STOP is
    type STATE_TYPE is (S0,S1,S2,S3,S4,S5);
    signal STATE: STATE_TYPE;
    signal SHIFT_REG: Std_logic_vector(3 downto 0);
begin
    STATE: process(CLK)
    begin
        if CLK = '1' then
```

```
case STATE is
    when S0 =>
        -- 数据单元
        -- 控制单元
        if R = '1' or A = '0' then
            STATE<=S0;
        elseif R= '0' and A = '1' then
            STATE<=S1;
        end if;
    when S1=>
        -- 数据单元,第 1 位数据输入移入
        SHIFT_REG<=D& SHIFT_REG(3 downto 1);
        -- 控制单元
        if R = '1' then
            STATE<=S0;
        elseif R= '0' then
            STATE<=S2;
        end if;
    when S2=>
        -- 数据单元,第 2 位数据输入移入
        SHIFT_REG<=D& SHIFT_REG(3 downto 1);
        -- 控制单元
        if R = '1' then
            STATE<=S0;
        elseif R= '0' then
            STATE<=S3;
        end if;
    when S3=>
        -- 数据单元,第 3 位数据输入移入
        SHIFT_REG<=D& SHIFT_REG(3 downto 1);
        -- 控制单元
        if R = '1' then
            STATE<=S0;
        elseif R= '0' then
            STATE<=S4;
        end if;
    when S4=>
        -- 数据单元,第 4 位数据输入移入
```

```
                    SHIFT_REG<=D& SHIFT_REG(3 downto 1);
                    -- 控制单元
                    if R = '1' then
                        STATE<=S0;
                    elseif R= '0' then
                        STATE<=S5;
                    end if;
                when S5=>
                    -- 数据单元
                    -- 控制单元
                    if R = '1' or A = '0' then
                        STATE<=S0;
                    elseif R= '0' and A= '1' then
                        STATE<=S1;
                    end if;
            end case;
        end if;
    end process STATE;

    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;
```

在结构体 FSM_RTL 中，首先定义了一个枚举型数据类型 STATE_TYPE，数据类型中的枚举元素名直接来自状态转换图或状态转换表。信号 STATE 保存了状态机的当前状态。

写出硬件的模型前，我们必须决定如何存储每一位串行输入数据。实现数据存储可以采用不同的方法，这将导致不同的硬件结构，从而影响硬件成本。在结构体 FSM_RTL 中，我们采用移位寄存器实现数据的存储，移位寄存器由一个 Std_logic 矢量表示，矢量的排序与输出信号 Z 的排序相同。

结构体中有两个进程 STATE 和 OUTPUT。其中 STATE 在每一个时钟周期的末尾

更新状态机的状态，从而它应该对时钟信号 CLK 敏感。输入信号 CLK 出现在进程 STATE 的敏感信号表中。用 if then 语句检查时钟 CLK 的上升沿，如果检测到时钟输入的上升沿，则更新状态机的状态。根据状态机的当前状态用 case 语句决定如何得到新状态，case 语句中的每个选择条件相应于状态的一个当前值。比如：case 语句中的选择条件 S0 相应于状态图中的状态 S0。在每个状态中都可能有两种操作：数据处理和控制处理。对于状态 S0，只有控制操作，没有数据操作。根据状态转换图，对于状态 S0，如果输入条件为 R＝'1'或 A＝'0'，下一状态还应该是 S0，VHDL 模型中通过语句 if R＝'1' or A＝'0' then STATE＜＝S0 实现了这种控制操作。状态转换图中还规定：如果输入条件为 R＝'0' 且 A＝'1'，则下一状态为 S1，VHDL 模型中通过语句 else if R＝'0' and A＝'1' then STATE＜＝S1 实现了这一功能。

case 语句中的其他选择条件定义了其他状态转换的情况。与状态 S0 的唯一不同之处是在某些选择条件下增加了数据操作部分。比如对于状态 S1，与状态操作的同时，还有数据操作，VHDL 模型中通过语句 SHIFT_REG＜＝D&SHIFT_REG(3 downto 0)实现了寄存器 SHIFT_REG 内数据的移位与新数据的存储。

显然，状态转换表中不但提供了状态转换信息及转换条件，而且提供了在每种状态下如何进行数据操作。读者可以对状态转换表和 VHDL 模型进行对比分析，检查 VHDL 模型是否实现了状态转换表规定的数据操作。

检查 OUTPUT 是否定义了器件的输出信号上的逻辑值。对于 Moore 状态机，输出信号只与状态机的状态有关，与当前输入的输入值无关，VHDL 模型中输出进程只对当前状态的变化敏感。对于 Mealy 状态机，输出信号可能与输入信号有关，输出进程的敏感信号可能包括输入信号。由于所设计的器件是一个 Moore 状态机，输出值只与状态机的状态有关，所以进程 OUTPUT 的敏感信号只是 STATE。VHDL 模型中使用 case STATE 语句实现了对输出信号的赋值。根据状态转换图，在状态 S0～S4，信号 DONE 的值是'0'，只有在状态 S5，输出 DONE 的值为'1'。如果 STATE 为 S0～S4，VHDL 模型中通过信号赋值语句 DONE＜＝'0'使输出信号 DONE 变为'0'，如果 STATE 为 S5，则信号赋值语句 DONE＜＝'1'使输出信号 DONE 变为'1'。根据所设计的硬件的指标规范，在状态 S0～S4，对输出 Z 取什么样的值并没有作出规定，只有在状态 S5，才规定输出 Z 的值为串行输入的 4 位数据。在 VHDL 模型中，Z 只在状态 S5 改变数值，而在其他状态，输出 Z 的值保持不变，这满足了硬件的指标规范。这里要指出一个问题，在使用自动综合工具的前提下，通常要求对硬件高层次模型的仿真结果与对较低层次模型的仿真结果一致，如果高层次模型中在状态 S0～S4 输出值保持不变，在硬件的较低层次需要采用特殊措施保持输出值不变，这可能增加较低层次的硬件设计复杂性。这就是说，在硬件造型的较高层次，对某些信号的值不作规定并不是一种最好的方法。采用多值逻辑方法，可以帮助解决这种问题。在本章的练习题中也要求读者对不指定数值的信号赋值语句进行改进。

6.2.7 VHDL 状态机模型的综合

根据有限状态机的 VHDL 模型，可以直接得到硬件电路。这里我们假定状态机的基本结构如图 6.13 所示，并首先讨论控制部分的综合。

对于控制部分，把每个控制状态用一个触发器实现，并搜索 VHDL 源代码中的 if then 语句，可以得到每个状态对应的触发器，从而设计出整个控制单元。图 6.14 是这样得出的控制单元示意图。

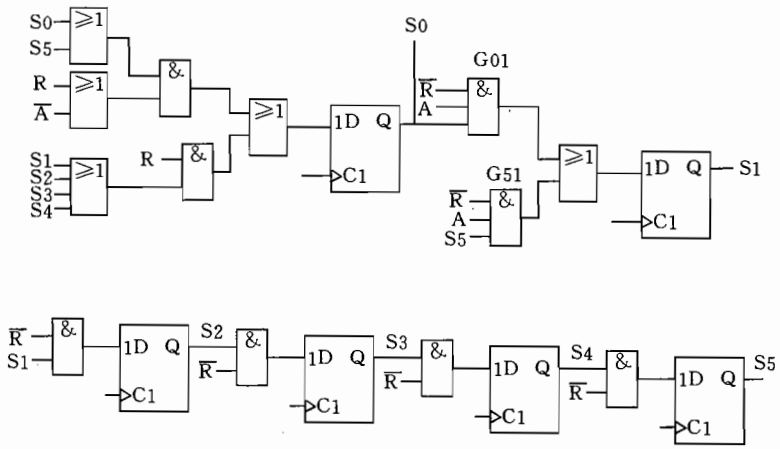


图 6.14　根据串并转换电路的 VHDL 模型综合出的控制单元

对于硬件模型中的每个触发器的数据输入端，可以通过检查 VHDL 模型中的 case 语句来完成。对于每一个状态，首先建立一个表格，列出从所有状态转换到这个状态的条件；然后，根据表中列出的条件，构造对应该状态的触发器的数据输入端的电路。表 6.3 是对于串并转换电路建立的这样的表格。图 6.14 是根据该表格构造出的电路。比如：对于状态 S1 的触发器的数据输入端 D，3 输入与门 G01 在输入条件为(not R) and A 时控制从状态 S0 转换到状态 S1，这对应于表 6.3 中终止态为 S1 的第一项内容。与此类似，3 输入与门 G51 在输入条件为(not R) and A 时控制从状态 S5 转换到 S1，这相应于表 6.3 中终止态为 S1 的第二项内容。用一个或门把 G01 和 G51 的输出送到状态 S1 对应的 D 触发器的数据输入端 D，就完成了状态 S1 的设计。相应于状态 S1 的 D 触发器的输出应该被送到数据单元、输出单元，还应该用于控制单元中需要使用状态 S1 的地方。

表 6.3　串并转换电路的控制单元综合

终止状态	起始状态	转换条件	终止状态	起始状态	转换条件
S0	S0	R or (not A)	S1	S0	(not R) and A
	S1	R		S5	(not R) and A
	S2	R	S2	S1	not R
	S3	R	S3	S2	not R
	S4	R	S4	S3	not R
	S5	R or (not A)	S5	S4	not R

根据 VHDL 模型中的 case 语句，可以综合出该串并转换电路的数据单元。首先，对

VHDL 模型中的 case 语句进行检查，列出每个状态下所要进行的数据操作以及数据操作条件的表格。然后根据表格中列出的条件生成数据操作电路。对于串并转换电路，可以列出如表 6.4 所示的表格。从表 6.4 可以看出，每个状态下数据操作的控制信号为状态信号与表格中相应状态下的数据操作条件的逻辑与。由于 4 个状态中都需要对数据进行移位操作，这种条件相应于将 4 个状态下的数据操作控制信号进行或运算，即移位操作的控制信号的逻辑表达式为

$$SHIFT=(S1)(1)+(S2)(1)+(S3)(1)+(S4)(1)=S1+S2+S3+S4$$

图 6.15 是这样得到的串并转换电路的数据单元。

表 6.4　串并转换电路的数据单元的综合

数　据　操　作	状态	条件
SHIFT_REG<=D&SHIFT_REG(3 downto 1)	S1	1
SHIFT_REG<=D&SHIFT_REG(3 downto 1)	S2	1
SHIFT_REG<=D&SHIFT_REG(3 downto 1)	S3	1
SHIFT_REG<=D&SHIFT_REG(3 downto 1)	S4	1

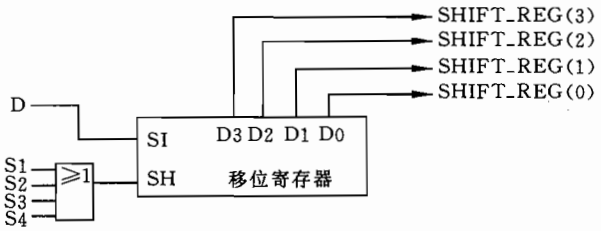


图 6.15　串并转换电路的数据单元

本小节给出的串并转换电路的 VHDL 模型中并没有告诉设计者任何关于如何设计移位寄存器的信息。设计移位寄存器是另外任务，在这里设计者假定移位寄存器是可以从设计库中得到的设计单元。

对 VHDL 模型中进程 OUTPUT 中的 case 语句的每种条件进行检查，也可以得出该串并转换电路的输出单元。由于只有在状态 S5 对输出信号 DONE 赋值，所以信号 DONE 的逻辑表达式为

$$DONE=S5$$

由于串并转换电路的原始指标中对状态 S0～S4 中的输出 Z 没有作出规定，输出信号 Z 的设计有些复杂。VHDL 模型中的输出进程描述了一个组合逻辑电路。如果利用没有规定状态 S0～S4 中输出 Z 的数值这一条件，可以令输出 Z 永远与移位寄存器中的数值相同。这样，得到的输出 Z 的逻辑表达式为

$$Z=SHIFT\_REG$$

VHDL 语言规定：如果没有对信号 Z 进行过操作，则总保持信号线 Z 上原来的数值。但是，上述逻辑表达式得出的输出电路是一个组合逻辑电路，Z 的数值总与移位寄存器中

保持的数据相同。除了在状态 S5 中之外，对 6.2.6 节中 VHDL 算法模型进行仿真得到的结果可能与对逻辑门级模型进行仿真得到的结果不同。但这种电路确实满足了电路的原始指标。

6.3 微程序控制单元设计

本节讨论微代码控制单元的设计问题。在给出了一个通用的基本微代码控制单元的基础上，把硬件描述语言和其他方法结合，讨论如何利用该基本微代码控制单元设计一个具体的微代码控制器。

6.3.1 基本微代码控制单元 BMCU

第 5 章讨论了微代码控制器的基本结构。为了讨论方便，将图 5.4 的微代码控制器重新画出，如图 6.16 所示。在微代码控制器中，控制信号的值预先存储在只读存储器 ROM 中，选择适当的 ROM 地址，可以得到需要的控制信号。ROM 中存储的内容被称为控制字。在每个时钟周期，微代码控制器从 ROM 的给定地址读出所需的控制信号。在图 6.16 的控制器中，包括有 A 位地址寄存器(MAR)、W 位指令寄存器(MIR)、微代码存储器 ROM 和地址生成逻辑电路(AGL)，微代码存储 ROM 的字长为 W，ROM 深度为 2^A。在给定时刻，地址生成电路计算出 ROM 地址，在这个地址中读出下一个时刻的控制信号以及下一次计算 ROM 地址所需的信息。地址生成逻辑确定下一 ROM 地址时还要用到数据单元产生的条件信号。地址生成电路可能很复杂，不同的设计可能采用不同的地址生成电路，通常地址生成电路是限制微代码控制器应用的主要因素。

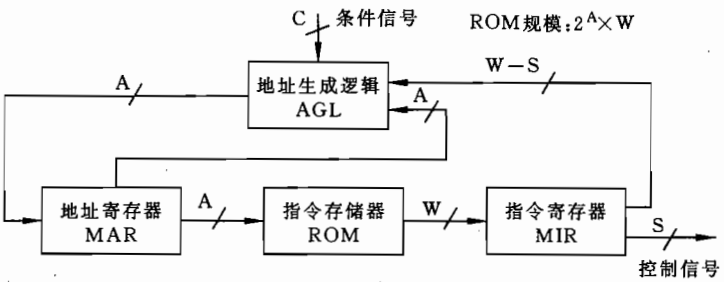


图 6.16 微代码控制器

本小节讨论一种最简单的微代码控制单元，称为 BMCU。这个微代码控制单元中的地址生成逻辑为两分支逻辑，由图 6.17 所示的矢量多路选择器构成。地址生成逻辑中的地址选择信号 AS 为'0'时，下一指令地址为当前指令地址加 1，当地址选择信号 AS 为'1'时，下一指令地址为 NAD 中指定的地址。AS 的取值由指令存储器中的控制字和来自数据单元的条件信号共同确定。

指令存储器中每个控制字有 64 位，其组织方式如图 6.18 所示。对应于图 6.16 的一般形式的方框图，W＝64，S＝31。在 64 位控制字中，最低 32 位是供数据单元使用的控制信号LCS；最高24位为条件选择信号CS，供“地址生成逻辑”生成下一ROM地址；中间

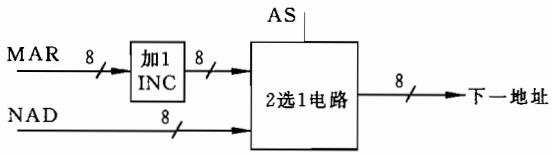


图 6.17　基本微代码控制电路的地址生成部分

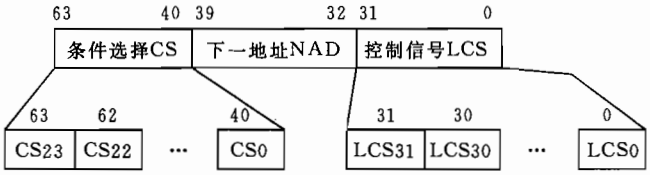


图 6.18　基本微代码控制单元 BMCU 中控制字的构成

8 位数据是分支地址 NAD，也用于生成下一地址。

控制字的条件选择信号 CS 部分（第 63～40 位）包含的信息用来把从数据单元来的条件信号进行分类选择，确定哪些条件应该被用来产生下一条指令的地址。在 BMCU 中，条件选择域 CS 中同时只可能有 1 位为'1'，其他位都为'0'，如果 CS 的某位 CSi 为'1'，则数据单元来的条件信号 Ci 被选择用来产生下一指令地址。在 BMCU 的设计中，条件信号的最大个数为 24。如果条件信号 Ci='1'，同时该信号被选择（即 CSi='1'），则下一条指令的地址是控制字中分支地址 NAD 中指明的地址；如果条件信号 Ci 被选择（即 CSi='1'），但信号 Ci 的值为逻辑'0'，则下一条指令的地址是当前地址（MAR 中的内容）加 1。

控制字中的控制信号 LCS 部分（第 31～0 位）包含了数据单元中使用的所有控制信号。在这个设计中控制信号的最大个数是 32。由于 ROM 地址必须能够通过控制字中的 NAD 部分直接指定，而 NAD 部分只有 8 位，所以这个 BMCU 的最大 ROM 规模为 256 字节（2^8=256）。

在图 6.17 所示的地址生成逻辑中，控制信号 AS 由下式计算：

$$AS=(CS23)(C23)+(CS22)(C22)+\cdots+(CS1)(C1)+(CS0)(C0)$$

即如果某个条件信号 Ci 被选择（LCSi='1'）且该条件信号 Ci 的值为逻辑'1'，则地址选择信号 AS='1'。

6.3.2　基本微代码控制单元 BMCU 的算法模型

为了采用自动综合程序综合出基本微代码控制单元（BMCU），应该写出 BMCU 的 VHDL 算法模型。根据前面的描述，很容易得出图 6.19 所示的进程模型图，这里增加了信号 RESET 对 BMCU 复位。

在进程模型图中，进程 MARP 表示地址寄存器 MAR。当出现复位信号 RESET 时，地址寄存器 MAR 被复位为"0000,0000"。在系统时钟的下降沿，新地址 NMAR 被加载到地址寄存器 MAR 内，根据 NMAR 的值，可以把新的控制字 CWORD 从 ROM 中读出。在系统时钟的上升沿，CWORD 被传递到指令寄存器 MIR，进程模型图中指令寄存器

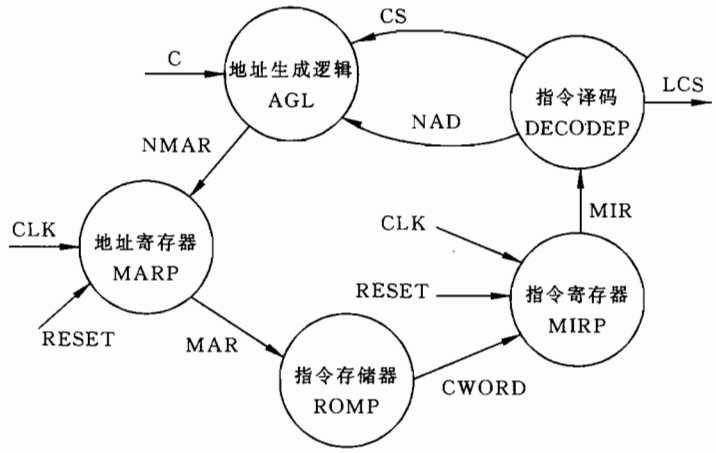


图 6.19　基本微代码控制单元的进程模型图

MIR 由进程 MIRP 表示。在出现系统复位信号时，指令寄存器被复位为零矢量。进程 DECODEP 的功能是把控制字 CWORD 分解为 CS，LCS 和 NAD 三部分。进程 AGL 根据前面描述的规则，计算下一个控制字的地址。如果地址选择条件 AS 为 true，则下一指令地址为 NAD；如果地址选择信号 AS 为 false，则下一地址为 MAR+1。信号 NMAR 用来表示进程 AGL 计算得到的新地址。在系统时钟的下降沿，这一新地址被加载到地址寄存器 MAR。这样就重新启动了一个指令周期。

根据这个进程模型图和第 4 章中的知识，可以写出描述该 BMCU 功能的 VHDL 源代码。在下面的 VHDL 算法模型中，每个器件中全引入了类属参数，用以在高层次描述延时信息。

```
entity BMCU is
    generic(ALG_DELAY,MAR_DELAY,ROM_DELAY,MIR_DELAY:
      TIME);
    port (C: in Std_logic_vector(23 down to 0) :=
        B"0000_0000_0000_0000_0000_0000";
        CLK,RESET : in Std_logic := '0';
        LCS: Std_logic_vector(31 downto 0));
end BMCU;

use work.BMCU_FUNCTIONS.all;
architecture ALGORITHMIC of BMCU is
    signal MAR,NMAR: Std_logic_vector (7 downto 0);
    -- MAR 是 ROM 的地址寄存器
    -- NMAR 是 MAR 的下一数值
    signal CWORD: Std_logic_vector(63 downto 0);
```

```
    -- 从 ROM 来的控制信号
    signal MIR: Std_logic_vector(63 downto 0);
    -- 保存从 ROM 来的控制信号
    signal NAD: Std_logic_vector(7 downto 0);
    -- 分支地址,是 CWORD 的一部分
    signal CS: Std_logic_vector(23 downto 0);
    -- 保存从 ROM 来的控制信号
begin
    MARP: process(CLK,RESET)
    begin
        if RESET = '1' then MIR<=
            MAR<=B"00000000" after MAR_DELAY;
        elseif CLK'Event and CLK ='0' then
            MAR<=NMAR after MAR_DELAY;
        endif;
    end process MARP;
    MIRP: process(CLK,RESET)
    begin
        if RESET = '1' then
            MIR<=X"00_00_00_00_00_00_00_00";
        elseif CLK'Event and CLK ='0' then
            MIR<=CWORD after MIR_DELAY;
        end process MIRP;
    ROMP: process(MAR)
        type MEM_TYPE is array (0 to 255) of Std_logic_vector(63 downto 0);
        constant MEM: MEM_TYPE :=
        -- 下面省略了 ROM 中的数据
        -- |-- CS --|-- NAD --|-- LCS --|
        ;
    begin
        CWORD<=MEM(Std_logic_vector_TO_INT(MAR)) after
                                        ROM_DELAY;
    end process ROMP;
    DECODEP: process(MIR)
    begin
        NAD<=MIR(39 downto 32);
        LCS<=MIR(31 downto 0)
        CS<=MIR(63 downto 40);
```

```
    end process DECODEP;
    AGL: process (MAR,NAD,C,CS)
        variable AS: Std_logic;
    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_DEL;
        end case;
    end process AGL;
end ALGORITHMIC;
```

微代码控制单元 BMCU 的算法模型要用到程序包 BMCU_FUNCTIONS。该程序包中包含了函数 Std_logic_vector_TO_INTEGER 和 INC，分别用来把位矢量转换为整数和计算位矢量加 1。这两个函数全是通用函数，其输入矢量的长度可以为任意值。函数中对信号属性 VEC'Right 和 VEC'Left 进行比较，以确定矢量为升序排列还是降序排列，并对两种情况分别处理。

```
package BMCU_FUNCTION is
    function Std_logic_vector_TO_INT(VEC: Std_logic_vector) return Integer;
    function INC(A:Std_logic_vector) return Std_logic_vector;
end BMCU_FUNCTION;

package body BMCU_FUNCTION is
    function Std_logic_vector_TO_INT(VEC:Std_logic_vector) return
      INTEGER is
        variable SUM,WT: Integer := 0;
    begin
        if VEC'Right<=VEV'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;
    endif;
    return SUM;
end Std_logic_vector_TO_INT;

function INC(A: Std_logic_vector) return Std_logic_vector is
    variable CARRY: Std_logic := '1'
    variable RESULT: Std_logic_vector(A'Range);
begin
    if VEC'Right<=VEV'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_FUNCTION;
```

把位矢量转换为整数的函数很简单，把二进制位矢量的每一位乘以它们的加权值，并把相乘后的数值相加，就可以得到二进制位矢量对应的整数值。在硬件实现中，ROM 地址选择信号总是二进制矢量输入，所以，并不需要实现这个转换函数。在这个 VHDL 模型中，由于 ROM 由数组表示，而数组的下标是整数值，因此使用了这个转换函数。

图 6.20 给出了二进制数加 1 运算的两个例子。当对一个二进制位矢量进行加 1 运算时，位矢量的最低位总是变为原来的反码，即从 1 变为 0 或从 0 变为 1。除此之外，如果二进制位矢量的低位是一串 1，则所有这些 1 都应该变成 0。如果对位矢量从右(LSB)向左(MSB)逐位搜索，第一次遇到的 0 应该变为 1，而该位左边的各位保持不变。在函数 INC 中，变量 CARRY 用来确定何时停止逐位求反。CARRY 被初始化为 1，用来强制对二进制位矢量的最低位求反，随着对二进制位矢量从右向左搜索，只要 CARRY 和 A(N)的与运算得到 true，说明搜索过程还是只发现 1，应该对这一位求反。图 6.20 就是根据上述算法进行加 1 运算的实例。

$$\begin{array}{r} 0101\_0111 \\ +\quad\quad 1 \\ \hline 0101\_1000 \end{array} \qquad \begin{array}{r} 1010\_1010 \\ +\quad\quad 1 \\ \hline 1010\_1011 \end{array}$$

图 6.20　二进制数加 1 的例子

6.3.3　基本微代码控制单元的综合

一般情况下，根据微代码控制器的算法模型，可以利用自动综合工具综合出微代码控制器的硬件本身。为了说明这种方法，我们讨论一种系统化设计微代码控制器的方法。以本章讨论的串并转换电路为例，具体讨论如何利用基本微代码控制器 BMCU 的基本结构，设计串并转换电路所需的 ROM 中内容。

系统化设计微代码控制器的算法由如下几步组成：

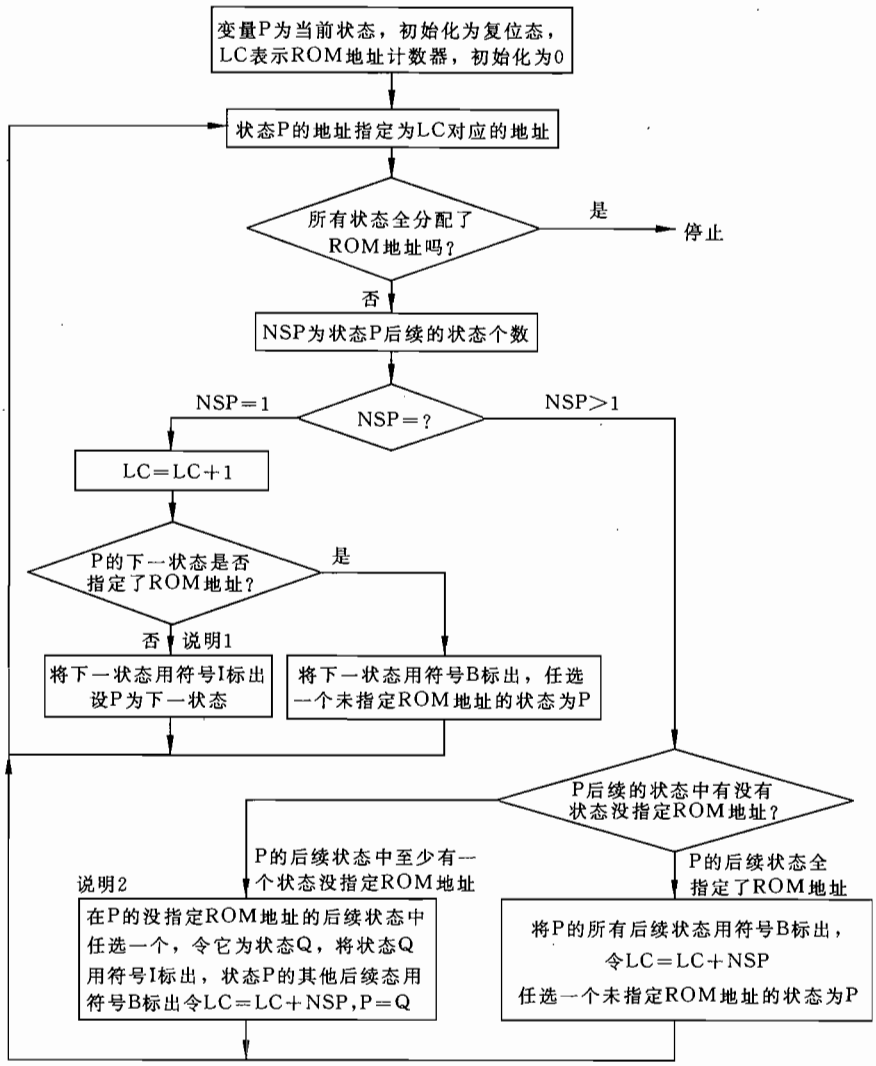
(1) 根据 VHDL 算法模型写出硬件的所有状态，并定义状态之间的转换条件。

(2) 列出第(1)步中出现的所有条件。后面对每个状态指定 ROM 地址之后，这些条件还可能要改变。

(3) 确定复位状态。通常硬件指标中会给出复位态，如果指标中没有指定复位态，则任意选定一个状态作为复位态。

(4) 对每个状态指定 ROM 地址。ROM 地址的确定会直接影响控制器的成本以及性能，不存在有效算法对指定 ROM 地址过程进行优化。ROM 地址的最优指定要寻找状态图中的最长路径，这在图论中是一个复杂的理论问题。虽然按最优的要求指定 ROM 地址是一个很复杂的问题，但下面的简单算法可以用来对每个状态指定 ROM 地址。这个算法类似于汇编语言编译器中的第一遍扫描，即类似于汇编语言编译器的地址分配过程。事实上，对微代码控制器的 ROM 编程通常使用一个类似于汇编语言编译器的工具完成。该算法如图 6.21 所示。

(5) 确定必须从数据单元传递到控制单元的条件信号。把第(2)步中建立的条件中的冗余项消除，把每个非冗余项对应于一个条件信号。这一步不影响控制器的成本及性能。



说明 1：符号 I 表示地址顺序增加；符号 B 表示地址跳转。

说明 2：可以采用最优化算法选择状态 Q；实现状态 P，需要 NSP 个 ROM 地址。

图 6.21　微代码控制器中的 ROM 分配算法

显然，如果在控制单元中多处使用了同一条件信号，也只需从数据单元中产生一次这样的信号。

(6) 列出每个状态下的数据操作、每个状态下的输出内容以及进行数据操作和产生输出的条件。

(7) 根据算法第(6)步列出的表格，确定控制数据单元进行数据操作和产生输出的信号，这些信号在控制单元中产生，从控制单元传递到数据单元。一般地讲，对应每对数据操作/条件或每对输出/条件都需要一个控制信号。但是通过查看各个控制信号之间的关系，可以作一些简化。由于微代码控制单元 BMCU 最多允许 32 个控制信号，可以使用 LCS0

~LCS31 之间的任何一个作为控制信号。

(8) 画出电路方框图。图中应该包括条件信号、控制信号、时钟信号以及复位信号。

(9) 根据前面八步得到知识,即根据对各状态的地址分配以及算法第(1),(4)两步构造的表格,确定 ROM 中的数据内容。这一步的工作类似于汇编程序编译器中的第二遍扫描时所做的工作。

下面以串并转换电路为例,说明如何完成上述几步工作。

第(1)步:根据电路的 VHDL 算法描述,可以得到如表 6.5 所示的状态转换表。读者先不用考虑表中最右列,该列的内容将在算法的第(4)步得出。

表 6.5 执行微代码控制器设计算法的第(1)步得到的状态转换表

当前状态	下一状态	转换条件	类型
S0	S0	R or (not A)	B
	S1	(not R) and A	I
S1	S2	not R	I
	S0	R	B
S2	S3	not R	I
	S0	R	B
S3	S4	not R	I
	S0	R	B
S4	S5	not R	I
	S0	R	B
S5	S1	(not R) and A	B
	S0	R or (not A)	B

第(2)步:对第(1)步产生的状态转换表中的所有条件全都列出如下:

(not R) and A	R or (not A)	not R	R

第(3)步:设定状态 S0 为复位态。

第(4)步:执行图 6.21 中给出的算法,可以把每个状态分别用符号 B(表示地址跳转)和 I(表示地址顺序增加)标注,标注的结果如表 6.5 中的最右边一列所示。对每个状态分配的 ROM 地址如表 6.6 所示。

表 6.6 对每个状态分配的 ROM 地址

状 态	ROM 地址	状 态	ROM 地址
S0	0	S3	3
S1	1	S4	4
S2	2	S5	5

第(5)步：观察表 6.5 中的数据，容易看出：如果表中最右一列用符号 I 标注，则从前一状态转换到当前状态只需要把 ROM 地址加 1，即把 MAR 中的内容加 1，这时不需要将对应的条件从数据单元传递到控制单元。如果表中最右一列用符号 B 标出，则转移到该状态需要 ROM 地址的跳转，因此需要把此信号从数据单元传递到控制单元。把表中所有用符号 B 标出的条件选出，消去冗余条件后，得到的三个条件为 (not R) and A，R or (not A)，R。

根据微代码控制单元 BMCU 的结构，最多可以允许有 24 个条件信号(C0～C23)，任意选择三个信号，任意分配给这三个条件，可以得到控制信号的分配，如表 6.7 所示。

表 6.7　算法第(5)步选定的条件信号

转换条件	控制信号
R	C23
R or (not A)	C22
(not R) and A	C21

第(6)步：根据 VHDL 源代码，可以列出在每个状态要进行的数据操作、实现数据操作的条件、各状态下电路的输出、产生输出的条件等项内容，如表 6.8 所示。表中条件 1 表示无条件进行数据操作或无条件产生输出。事实上，这个表格可以在算法第(1)步中产生，只是为了清楚地划分算法每步要完成的任务，这里把两种不同的任务分成了两步。

表 6.8　算法第(6)步产生的数据操作和输出内容表

	数据操作		输　出		输　出	
状态		条件	DONE	条件	Z	条件
S0	—	—	0	1	—	—
S1	数据右移	1	0	1	—	—
S2	数据右移	1	0	1	—	—
S3	数据右移	1	0	1	—	—
S4	数据右移	1	0	1	—	—
S5	—	—	1	1	数据	1

第(7)步：根据表 6.8 可以得出需要从控制单元传递到数据单元的控制信号。显然可以看出，需要有一个信号控制数据的移位，还需要有一个信号控制产生输出。这个例子中数据操作和输出全无条件。表 6.9 列出了需要的两个控制信号，并把它们与 LCS31 和 LCS30 对应起来。

表 6.9　算法第(7)步确定的控制信号

控制单元输出	信　　号	数据操作/系统输出
LCS31	SHIFT	数据移位
LCS30	DOWN_CONTROL	DONE=1,Z=数据

第(8)步：至上面的第(7)步，就确定了需要在数据单元和控制单元之间传递全部信号，可以根据前面几步的结果构造电路的方框图。图 6.22 示意给出了用微代码控制器 BMCU 实现的串并转换电路的方框图。24 个条件信号 C0～C23 中只使用了 3 个，其他 21 个信号全接到了信号 F，而信号 F 接到了逻辑'0'。

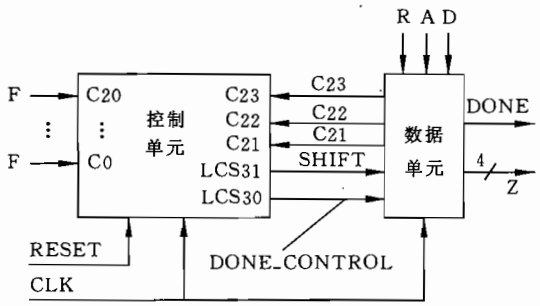


图 6.22　串并转换电路的方框图

第(9)步：根据表 6.5 中内容，很容易构造出 ROM 中的数据。表 6.10 给出了用微代码控制器 BMCU 实现串并转换电路时 ROM 中的数据。根据表 6.6 中对状态的地址分配，ROM 地址 0 对应于状态 S0。根据表 6.5 可知，状态 S0 情况下，产生跳转的条件(用符号 B 标出)为 C22(R or (not A))，所以 ROM 地址 0 中的控制字的 CS 域中只有对应 C22 的一列为 1，其他全为 0。同样根据表 6.5 可知，C22 为 true 时的跳转后的状态为 S0，所以，地址 0 的控制字的 NAD 域的内容为"00000000"。根据表 6.8 可知，在状态 S0 不进行数据操作，也不产生输出，所以对应信号 DONE_CONTROL 和 SHIFT 的两位数据(LCS30 和 LCS31)应该为逻辑 0，而地址 0 的控制字中 LCS 域的其他位也应该全是 0。当 C22 为 false 时，ROM 地址应该增加 1，器件进入状态 S1，状态 S1 的地址为 1。ROM 地址 1～4 对应于状态 S1～S4，确定 ROM 地址 1～4 的内容的方法与 ROM 地址 0 的方法完全相同。

表 6.10　用微代码控制器 BMCU 实现串并转换电路时的 ROM 内容

状态	地址	控制字 CWORD																
		63	62	61	60	…	40	39	…	32	31	30	29	…	0
		CS						NAD			LCS				
		23	22	21	20	…	0	7	…	0	31	30	29	…	0
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	—		

对于串并转换电路，从状态 S5 可以转换到 S0，也可以转换到 S1。由于微代码控制器 BMCU 中的地址生成逻辑中只允许有一个跳转地址和一个增量地址，所以对应状态 S5 的 ROM 地址中的数据要考虑两个跳转地址的情况，实现从一个状态向两个或两个以上的状态跳转需要多个 ROM 地址。对于本例中的状态 S5，在 ROM 地址 5 对条件 C21((not R)and A)进行检查，因此该地址的控制字中的 CS 域中对应 C21 的位设为 1，其他位为 0。如果 C21 为 true，器件应该跳转到 ROM 地址 1(状态 S1)，应该在 ROM 地址 5 的 NAD 域中指定这一地址。如果 C21 为 false，器件进入 ROM 地址 6，在 ROM 地址 6 再对条件 C22(R or (not A))进行检查，对应 ROM 地址 6 的控制字中的 CS 域中 C22 位应该设为 1，其他位全为 0。如果 C22 为 true，硬件跳转到地址 0(状态 S0)，因此 ROM 地址 6 的控制字中 NAD 域要设为 0。可以看出，由于在状态 S5 可能跳转到两个不同状态，实现状态 S5 需要两个 ROM 地址，这就验证了图 6.21 的说明中给出的结论。由于实现状态 S5 用了两个 ROM 地址，硬件的工作也同样需要两个时钟周期。对于控制单元产生的控制信号，设计方法与状态 S0 相同，在状态 S5 不需要对数据进行移位操作，所以 ROM 地址 5,6 中的控制字的 LCS 域中对应 LCS31(SHIFT)位的数据应该为 0。由于在状态 S5 要产生输出 DONE 和 Z，所以 ROM 地址 5,6 中的控制字的 LCS 域中对应 LCS30(DONE_CONTROL)位的数据应该为 1，其他各位全应该为 0。

定义了 ROM 中的数据，就完成了微代码控制器的设计。由于微代码控制器 BMCU 的硬件设计已经完成，对于具体的电路设计，比如串并转换电路，设计者要做的工作就是编制 ROM 中的数据，并把编程后的 ROM 嵌入微代码控制器 BMCU。在对 ROM 编程之前，可以利用 VHDL 对设计的 ROM 数据进行验证。把 6.3.2 节讨论过的 BMCU 的算法模型 ALGORITHMIC 中数组 MEM 中的内容按表 6.10 的内容填充，并把结构体名换为 STOP，可以得到串并转换电路的控制部分的算法模型如下。

```
use work.BMCU_FUNCTIONS.all;
architecture STOP of BMCU is
    signal MAR,NMAR: Std_logic_vector (7 downto 0);
    -- MAR 是 ROM 的地址寄存器
    -- NMAR 是 MAR 的下一数值
    signal CWORD: Std_logic_vector(63 downto 0);
    -- 从 ROM 来的控制信号
    signal MIR: Std_logic_vector(63 downto 0);
    -- 保存从 ROM 来的控制信号
    signal NAD: Std_logic_vector(7 downto 0);
    -- 分支地址，是 CWORD 的一部分
    signal CS: Std_logic_vector(23 downto 0);
    -- 保存从 ROM 来的控制信号
begin
    MARP: process(CLK,RESET)
    begin
```

```
        if RESET = '1' then MIR<=
            MAR<=B"00000000" after MAR_DELAY;
        elseif CLK'Event and CLK ='0' then
            MAR<=NMAR after MAR_DELAY;
        endif;
    end process MARP;
    MIRP: process(CLK,RESET)
    begin
        if RESET = '1' then
            MIR<=X"00_00_00_00_00_00_00_00";
        elseif CLK'Event and CLK ='0' then
            MIR<=CWORD after MIR_DELAY;
    end process MIRP;
    ROMP: process(MAR)
        type MEM_TYPE is array (0 to 255) of Std_logic_vector(63 downto 0);
        constant MEM: MEM_TYPE :=
            ( 0=>X"40_00_00_00_00_00_00_00",
              1=>X"80_00_00_00_80_00_00_00",
              2=>X"80_00_00_00_80_00_00_00",
              3=>X"80_00_00_00_80_00_00_00",
              4=>X"80_00_00_00_80_00_00_00",
              5=>X"20_00_00_00_40_00_00_00",
              6=>X"40_00_00_00_40_00_00_00",
              others=>(others=>'0'));
    begin
        CWORD<=MEM(Std_logic_vector_TO_INT(MAR)) after
                                ROM_DELAY;
    end process ROMP;
    DECODEP: process(MIR)
        NAD<=MIR(39 downto 32);
        LCS<=MIR(31 downto 0)
        CS<=MIR(63 downto 40);
    end process DECODEP;
    AGL: process (MAR,NAD,C,CS)
        variable AS: Std_logic;
    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_DEL:
        end case;
    end process AGL;
end STOP;
```

前面第 6.2 节讨论微代码控制器实现串并转换电路的设计过程中并没有讨论数据单元的设计。无论电路的控制单元由硬连接方法实现，还是由微代码控制器实现，数据单元的设计都是相同的。下面的 VHDL 源代码是数据单元的数据流模型，根据数据流模型可以采用前面已讨论过的方法设计数据单元的硬件。对于具体的硬件设计，数据单元总是要专门设计。

```
entity DATASTOP is
    generic(SHIFT_DELAY,GATE_DELAY: TIME)
    port(R,A,D,CLK,SHIFT,DOWN_CONTROL: in Std_logic;
        Z: out Std_logic_vector(3 downto 0);
        DONE. C23,C22,C21: out Std_logic);
end DATASTOP;

architecture DATAFLOW of DATASTOP is
    signal SHIFT_REG: Std_logic_vector(3 downto 0);
begin
    -- 移位寄存器
    SHIFT_REG<=D & SHIFT_REG(3 downto 1) after SHIFT_DELAY when
        CLK'Event and CLK='1' and SHIFT='1' else SHIFT_REG;
    -- 控制单元需要的条件信号
    C23<=R;
    C22<=R or not A after GATE_DELAY;
    C21<=not R and A after GATE_DELAY;
    -- 输出信号
```

```
    DONE<=DONE_CONTROL;
    Z<=SHIFT_REG;
end DATAFLOW;
```

下面给出的 VHDL 源代码是串并转换电路的系统级模型，可以对该模型进行仿真以确定系统是否能正常工作。在系统级模型中，控制单元和数据都用一个器件表示，器件之间的连接关系如图 6.22 所示。由于模块的所有输入端都不应该开路，这里对没用的端口全连接到信号 F，信号 F 则接到逻辑'0'，对于各模块中不用的输出端口，则任其开路。在实际硬件设计中，这是一种很好的设计习惯。

```
use work.all
entity TEST_BENCH
end TEST_BENCH;

architecture BMCU_TEST of TEST_BENCH is
    signal R,A,D,CLK,INIT,RESET: Std_logic;
    signal C23,C22,C21: Std_logic;
    signal SHIFT,DONE_CONTROL: Std_logic;
    signal DONE: Std_logic;
    signal Z: Std_logic_vector( 3 downto 0);
    signal X: Std_logic_vector(3 downto 1);
    signal F: Std_logic;
    -- 数据单元
    component DATA_UNIT
        generic(SHIFT_DELAY,GATE_DELAY: TIME)
        port (R,A,D,CLK,SHIFT,DONE_CONTROL: in Std_logic;
            Z: out Std_logic_vector(3 downto 0);
            DONE,C23,C22,C21: out Std_logic);
    end component;
    -- 控制单元
    component MICRO_CONTROL_UNIT
        generic (AGL_DELAY,MAY_DELAY,ROM_DELAY,MIR_DELAY,
              MIR_SETUP,MAR_SETUP: TIME);
        port(C: in Std_logic_vector(23 downto 0) -- 数据单元来的条件信号
            CLK,RESET: in Std_logic; -- 系统时钟和复位信号
            LCS: out Std_logic_vector( 31 downto 0));
        -- 送往数据单元的控制信号
    end component;
```

```
    for L1: DATA_UNIT use entity DATASTOP(DADAFLOW);
    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,DOWNCONTROL,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(21)=>SHIFT,LCS(30)=>DONE_CONTROL);
    -- 时钟进程
    process
    begin
        CLK<='0' ;
        wait for 200 ns;
        CLK<='1' ;
        wait for 200 ns;
    end process;
    -- 提供输入信号值的进程
    F<='0';
    RESET<='1' ,'0' after 30 ns;
    process (X)
    begin
        R<=X(3);
        A<=X(2);
        D<=X(1);
    end process;
    -- 对输入 X 赋值
    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;
```

6.3.4 微代码控制单元的局限性

第5章中讨论过微代码控制单元的主要优点，并与硬连接方式的控制单元进行过对比。本小节以微代码控制器BMCU为例，具体讨论利用微代码控制单元进行电路设计的好处及其局限性。

微代码控制单元BMCU只能从数据单元得到不多于24个的条件信号，即控制单元只能对不多于24种条件进行检查。微代码控制器最多只能为数据单元提供32个控制信号，所以它最多只能控制数据单元进行32种独立的数据操作。由于控制字中地址域NAD只有8位，限制了微代码控制器的最大编程量是256个控制字。这三种局限性都由于BMCU中控制字的长度及其三个域的分配方式所造成。一般地讲，任何预先设计好的微代码控制器中控制字的长度都限制了控制器的应用。

事实上，控制字的长度限制并不是微代码控制器最严重的局限性，而产生下一地址的方法对微代码控制器应用的限制更为严重。微代码控制器BMCU采用了最基本的2分支电路确定下一ROM地址。控制单元根据一个输入条件信号确定分支地址。如果图6.17中的选择信号AS为'1'，后续地址可以跳转为ROM区内的任意地址；如果选择信号AS为'0'，则后续地址为ROM中相邻的下一地址。虽然这样的地址生成电路也可以实现3分支或3个以上的分支，但利用这种2分支电路实现多分支状态转移需要多个微指令周期才能完成，这会影响硬件的运算速度。比如，如果对于前面讨论的串并转换电路的状态5，硬件可能转移到两种不同状态S0或S1，实现该状态下的功能需要两个ROM地址，执行该步运算需要两个微指令周期。一般地讲，对于m分支的状态转移，需要$m-1$个跳转地址和一个增量地址，判断向哪个地址跳转需要$m-1$个跳转条件，执行状态转移需要$m-1$个微指令周期。通常一个微指令周期就是一个系统时钟周期，如果控制单元确定状态转移需要的时间多于一个微指令周期，有可能对原始的数据输入输出的时序产生影响。

对于串并转换电路STOP，如果当前状态为S5，需要根据输入条件确定下一状态为S0还是S1。如果用微代码控制器BMCU实现该串并转换电路，在状态S5需要两个微指令周期。假定在状态S5中的第二个微指令周期下一个输入脉冲序列到来(如图6.23所示)，由于这时控制器不对输入端进行检查，该串并转换电路会漏掉图中标出的输入脉冲序列中的第一位数据。这就是说，用微代码控制器BMCU实现的串并转换电路的时序特性与图6.9给出的指标要求并不严格一致。如果严格按图6.9中的时序要求设计串并转换电路，即每位数据只维持一个时钟周期，且在一组串行数据输入后立刻产生并行输出，并可以立即输入下一组串行数据，则不能采用BMCU这样的微代码控制器实现该电路。

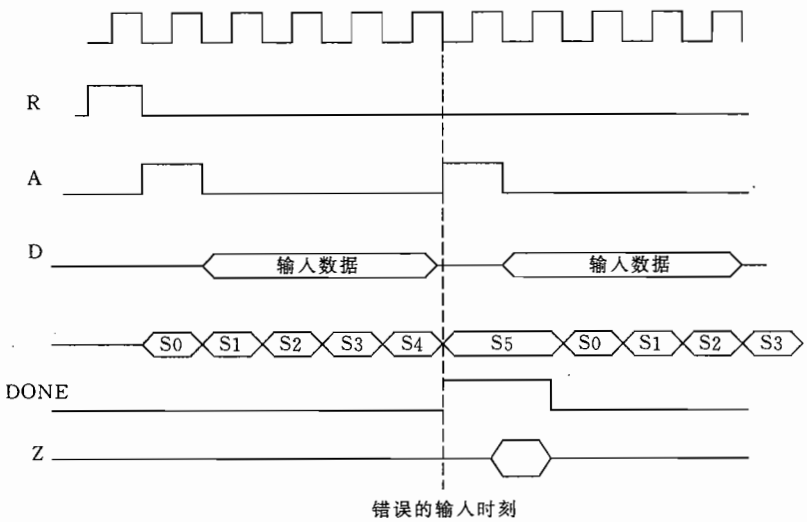


图 6.23　微代码控制器 BMCU 实现的串并转换电路可能会漏掉的输入脉冲

一般地讲，微代码控制器中的地址生成逻辑限制了确定一个状态的后续状态的方法，也就使得在不同的状态需要不同的位指令周期实现。如果把这样的器件与其他器件互连，通常需要采用工作速度较慢的握手协议实现器件之间的通信。除此之外，如果严格规定了器件输入和输出的时序关系，有时不能采用给定的微代码控制器实现特定的硬件。上面讨论了微代码控制器实现串并转换电路时对时序关系的限制。事实上，任何微代码控制器都限制了所能实现的电路的时序关系。

6.3.5　微代码控制器设计时的其他条件选择方法

在微代码控制器 BMCU 中，控制字的 CS 域中的每一位都对应于一个数据单元中产生的条件信号，用条件信号直接控制选择跳转 ROM 地址或顺序 ROM 地址，不需对条件信号再进行译码，被称为条件线性选法。条件线性选法的译码代价最小。当条件信号个数较多时，采用线选方法，需要 CS 域的位数较多。利用它设计具体电路时，编程工作量较大。与此相反，还可以对条件信号进行全译码选择 ROM 地址，这种方法称为条件全译码选择。采用全译码法时，如果控制字的 CS 域有 16 位数据，对 16 位数据全译码可以译出 $2^{16}=65\,536$ 个输出，可以用来对应数据单元中产生的 65 536 个条件信号。这种全译码方法，充分利用了控制字中的 CS 域中的信息，但译码代价较高。比如，假定控制字的字长为 64 位，控制器最大允许的条件信号个数为 16，如果选用条件线性选择方法，则控制字的 CS 域需要 16 位。如果采用全译码方法，则控制字的 CS 域需要 4 位（$2^4=16$）。假定保持控制字的其他内容相同，那么与线选译码实现的硬件相比，全译码方法实现的硬件是以增加一个 4 线-16 线译码器为代价把微程序 ROM 的字长从 64 位降低到了 52 位。条件线性选择和条件全译码选择是两个极端，在两个极端之间可以有多种折衷方案。设计者可以根据译码电路的复杂程度以及 ROM 字长等各种因素综合考虑应该采用什么方法。

6.3.6 选择 ROM 地址的多分支方法

根据前面的讨论可知，地址生成逻辑是限制微代码控制器应用范围的重要因素。改进地址生成逻辑，使得在一个地址可以根据不同情况跳转到多于一个地址，一定可以增加微代码控制器的适应范围，同时也需要增加地址生成电路的复杂性。图 6.24(a)给出了一种控制字分配方法，可以用于 4 种不同的地址生成方法。控制字的 64 位数据的最高 2 位构成了 AT 域，用来在 4 种地址生成方法中选择一种。6 位 CS 域被全译码为 64 位输出，用来对应于数据单元产生的最大 64 个条件信号。三个地址域 NAD1，NAD2 和 NAD3 用来计算后续 ROM 地址。ROM 地址宽度为 16 位，因此最大 ROM 地址为 64kb。32 位的 LCS 域是为数据单元提供的控制信号，最大可以提供 32 个控制信号。

如果 AT＝00，微代码控制器无条件转移到 NAD1 和 NAD2 共同组成的 16 位 ROM 地址，这种情况下 CS 域和 NAD3 域中可以为任意值，地址生成逻辑不考虑 CS 域和 NAD3 域中是什么数值。这种情况下控制字的意义如图 6.24(b)所示。

如果 AT＝01，控制器采用两分支地址生成逻辑。这种情况与 BMCU 类似，CS 域被全译码为 64 个输出，分别用来选择数据单元产生的 64 个条件信号，如果选择条件为 true，则控制器跳转到 NAD1 和 NAD2 组成的 16 位 ROM 地址。如果选择条件为 false，则控制器顺序执行后面的 ROM 地址，即新的 ROM 地址为 MAR＋1。图 6.24(c)给出了这种情况下的控制字分配。

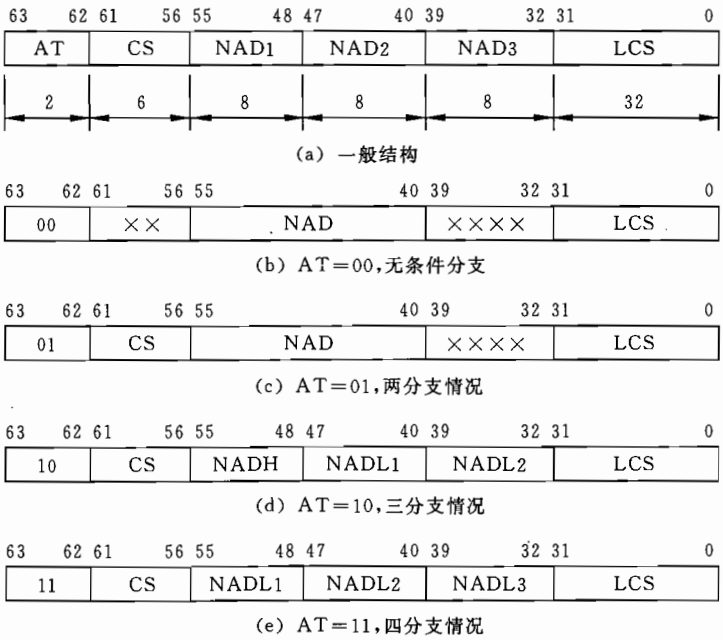


图 6.24　程序分支更灵活的控制字组织

如果 AT＝10，控制器采用三分支地址生成逻辑。硬件设计者要自行选择两组输入条件，为了简化地址生成逻辑，要求两组条件的二进制表示为两个连续的二进制数，且第一

个数为偶数，即 C0='0'。先将控制字的 CS 域全译码，译码结果和第一组输入条件信号按位作与运算，如果得到非零结果，则设 CDL 为逻辑'1'。再将译码结果与第二组输入条件信号按位作与运算，如果得到非零结果，则设 CDH 为逻辑'1'。计算 CDL 和 CDH 的电路原理如图 6.25 所示。硬件设计者应该保证数据单元产生的条件信号不能同时产生 CDL 和 CDH 同时为逻辑'1'的情况。如果 CDL 为 true，则将 NAD1 和 NAD2 组合起来，构成控制器的跳转地址，即将图 6.24(d)中的 NADH 和 NADL1 组合构成跳转地址。如果 CDH 为 true，则将 NAD1 和 NAD3 组合构成控制器的跳转地址，即将图 6.24(d)中的 NADH 和 NADL1 组合构成跳转地址。如果 CDL 和 CDH 全都为 false，则将当前 ROM 地址加 1 构成新的 ROM 地址。图 6.24(d)中给出了三分支地址生成逻辑的示意图，用于两种跳转情况下的跳转地址的高字节相同，全为 NADH，所以两个跳转地址之间的最大距离为 256 字节。当然，也可以在控制字中保存两个完整的 16 位跳转地址，以减小对跳转地址的限制，但这样会把 ROM 字长从 64 位增加为 72 位。设计者可以综合考虑增加 ROM 字长的代价与增加跳转地址灵活性哪种情况更适合设计要求。

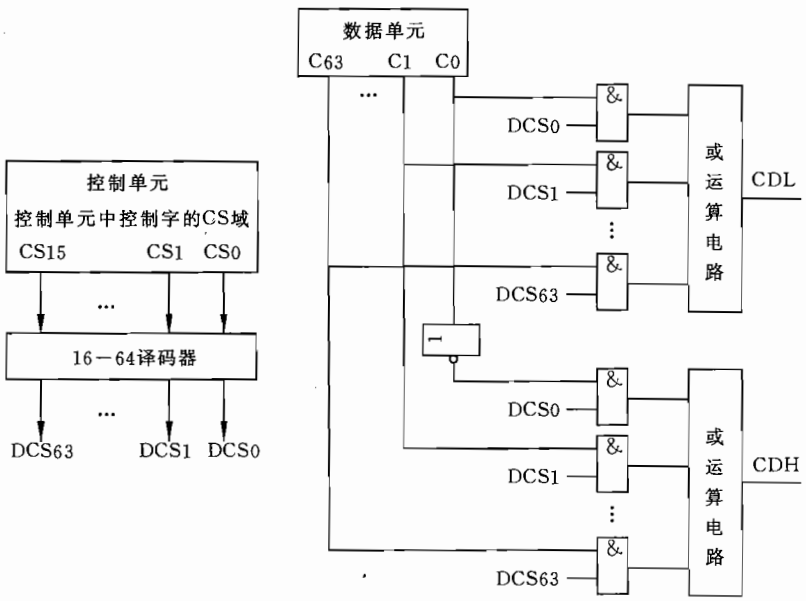


图 6.25　三分支地址生成逻辑中 CDL 和 CDH 的计算

如果 AT=11，控制器采用四分支地址生成逻辑，硬件设计者要自行选择三组输入条件。为了简化地址生成逻辑，要求三组条件的二进制表示为三个连续的二进制数，且第一个数为 4 的整数倍，即第一个二进制数的最低(右)两位全为'0'。比如用户可以选择三组条件信号输入的二进制值分别为 24、25 和 26，其二进制表示分别为 11000，11001 和 11010。将控制字的 CS 域全译码，译码结果和第一组输入条件信号按位作与运算，如果得到非零结果，则设 CD1 为逻辑'1'。将译码结果与第二组输入条件信号按位作与运算，如果得到非零结果，则设 CD2 为逻辑'1'。将译码结果与第三组输入条件信号按位作与运算，如果得到非零结果，则设 CD3 为逻辑'1'。计算 CD1，CD2 和 CD3 的电路原理如图 6.26 所

示。硬件设计者应该保证数据单元产生的条件信号不能同时产生 CD1,CD2 和 CDH 同时为逻辑'1'的情况。如果 CD1 为 true,则将当前 ROM 地址,即 MAR 中内容的高8位和 NAD1 组合起来,构成控制器的跳转地址。如果 CD2 为 true,则将当前 ROM 地址的高 8 位和 NAD2 组合起来,构成控制器的跳转地址。如果 CD3 为 true 则将当前 ROM 地址的高 8 位和 NAD3 组合起来,构成控制器的跳转地址。如果 CD1=CD2=CD3=false,则将当前 ROM 地址的下一地址,即 MAR+1 作为后续地址。图 6.24(e)示意给出了四分支地址生成的情况。采用这里给出的地址生成方法,三个跳转地址的高 8 位与当前地址的高 8 位相同,即当前 ROM 地址与跳转地址总在 256 字节的 ROM 区内。与三分支地址生成相同,由于对地址跳转有最大跳转长度为 256 字节的限制,才保持了 ROM 字长为 64 位的优势,这通常是一种较好的设计方案。

利用本节讨论的组合地址生成算法,对于大部分应用,有可能在不增加每个状态下的微指令周期的前提下,设计出微代码程序。当然,不可能利用这里介绍的地址生成方法直接实现五分支或多于五分支的状态转移。同时应该知道,即使对于三分支或四分支的地址跳转,采用这里介绍的地址生成逻辑对地址的产生也有一些限制。

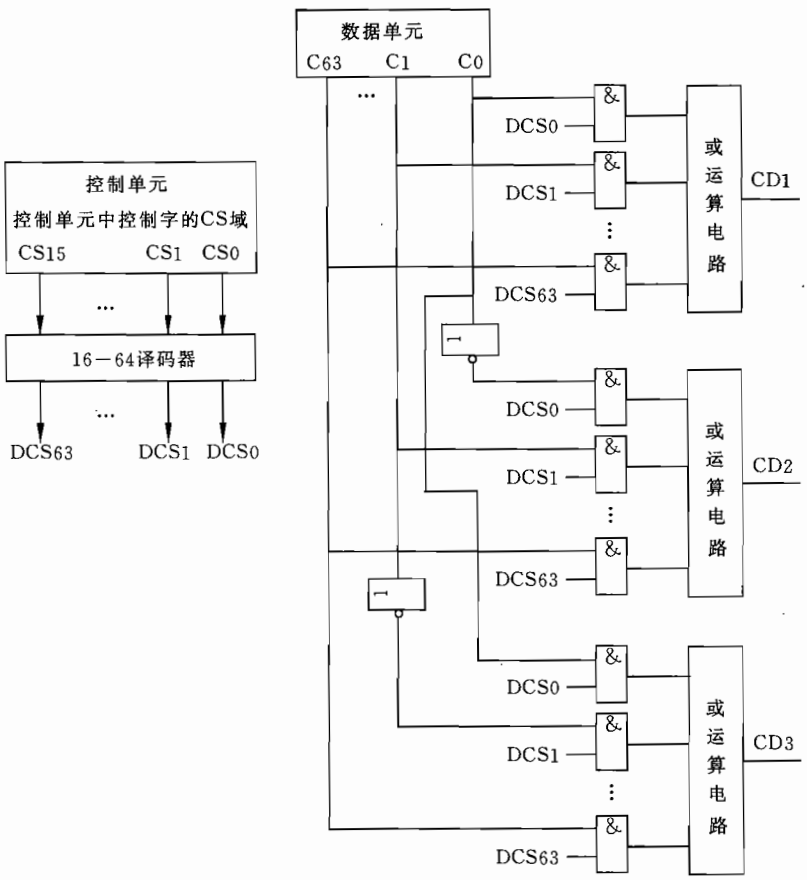


图 6.26 四分支地址生成逻辑中 CD1,CD2 和 CD3 的计算

第7章 行为综合

行为综合的研究工作起始于20世纪60年代，但真正得到发展是在近十几年。特别是20世纪80年代后期以来，学术界和工业界开发了一系列行为级综合工具，其中比较有影响的有法国TIMA/INPG实验室的AMICAL、荷兰埃因霍温工业大学的ESCAPE等，最为成熟并且首先商业化的是美国Synopsys公司的BC(Behavior Compiler)。

当前进行的许多复杂设计，一般都是先用软件算法表达设计意图并进行仿真，然后将软件描述翻译为可综合的寄存器传输级硬件描述，最后用硬件电路实现。其中的软件算法描述就是芯片功能的表示，如果翻译成对应的硬件描述语言，则称为行为级描述。在得到行为级描述后，有两种途径可以完成后续的芯片设计：寄存器传输级综合方法或行为级综合方法。

采用寄存器传输级综合方法，系统设计师首先用C或其他高级编程语言建立算法描述，继而硬件设计师把这一算法描述翻译成与算法描述直接对应的行为级硬件描述(通常用VHDL或Verilog HDL)，接下来用手工的方式把行为级描述翻译成寄存器传输级描述，此后即可进行寄存器传输级综合等后续步骤。这种方法存在着一些问题：① VLSI的设计要求整个过程是一个紧密耦合的过程，各个设计层次上的EDA工具应该具备各种内在的、自动的通信机制，而在这种设计方法中行为级和寄存器传输级之间是脱节的，寄存器传输级以后的设计信息很难反馈到前级，从而可能导致各个层次描述不一致；② 从行为描述向寄存器传输级的手工翻译过程费时费力，且极易发生错误；③ 对于典型的集成电路设计规模，寄存器传输级的仿真速度很慢；④ 这种方法限制了探索设计空间的能力，这是由于寄存器传输级描述只能反映一种特定的硬件结构，而算法级描述和性能要求的改变常常导致寄存器传输级的完全重新设计。

20世纪90年代以来，芯片集成度的增加极为迅速，市场需求的变化迅速，要求大幅度地缩短集成电路的设计周期。传统的寄存器传输级的设计方法在能力上已达到了极限，ASIC设计师必须在寄存器传输级更高的抽象层次上进行设计。因此，行为综合的设计方法和设计工具就应运而生了。

行为综合，也称高层次综合，就是由EDA工具经过一系列自动转换，从行为级描述出发，生成寄存器传输级描述。其中，行为级描述是设计对象的功能或算法表示，而寄存器传输级描述则是实现这些功能或算法的某个硬件结构的表示。

行为综合的方法同样起始于算法描述及其对应的HDL行为描述，这一行为描述具有与软件算法相同或类似的指令、操作、变量和数组等，只是增加了I/O功能描述。行为综合工具根据设计对象的行为描述自动生成寄存器传输级描述，这是行为综合方法与寄

存器传输级综合的关键的不同之处。行为综合方法中行为综合工具代替了设计师的手工工作，自动地对行为级代码进行调度、硬件分配、资源共享、存储器配置等处理，形成可综合的寄存器传输级代码。这种目标代码是按照通用计算模型组织起来的，一般由数据路径、控制器和存储器组成。除了行为描述之外，硬件设计师还要向行为综合器提供综合约束并选择综合策略，从而在设计空间中寻找优化的目标结构。与寄存器传输级综合方法相比，行为综合方法具有这样一些优点：① 代码简洁直观。一般情况下，行为级代码的长度是相应寄存器传输级代码的 1/5～1/3，同时因为是直接描述算法，因而可读性强，便于建立和修改模型。② 由于行为级代码的数据类型更加抽象，所以仿真速度比寄存器传输级快得多。③ 使用行为级综合，可以比较多种不同目标结构，在多种硬件实现方式上进行比较。④ 提高了设计的可再用性，容易适应指标的改变。这是因为行为描述的高度抽象性，使得有较大的余地利用现有设计，同时行为级以下的模型都是自动生成，而寄存器传输级描述在结构上已经相当确定，只有特定功能的硬件才能满足需要。⑤由于行为综合工具的引入，使整个集成电路设计成为一个紧密耦合的自动化过程，各个层次的数据能够以自动的方式反馈，从而有可能获得更高质量的设计结果。

本章首先概要介绍行为综合的发展，然后论述行为级描述的内部表示和行为级综合的目标体系。内部表示也称中间格式，是行为综合工具内部对输入的行为级描述的表示形式；目标体系是行为综合器输出的结构模型，一般由数据路径、控制器及存储器件组成。接下来的部分讨论行为综合的流程和行为级 VHDL 建模，并以一个例子来说明行为综合的建模过程。

7.1 设计表示的中间格式

行为级描述的代码被行为综合器读入后，要转换为一定的内部模型，即某种严格定义的中间格式，通常称为行为综合的设计表示。整个行为综合的过程就是对内部设计表示的连续转换过程。虽然对于硬件设计师来说，综合器内部的设计表示是不可见的，但是对它的了解有助于对行为综合过程的理解和把握，而且行为综合工具的设计结果报告是依据内部表示生产的，所以在这一节中，介绍常用的设计表示方式。

面向语言的表示形式分为三类：数据流图、控制流图和控制数据流图，下面分别介绍。

7.1.1 数据流图

数据流图(data-flow graph)是程序设计中常用的中间格式。在这种图中，节点代表程序中的操作，有向线代表数据。节点的作用是根据输入数据产生新的数据。图 7.1 是一段 VHDL 程序(图(a))和相应的数据流图(图(b))。图中节点 N1～N5 对各种数据进行运算处理，各条线 a～g，v1～v4 则是相应数据，线上的箭头表示数据流动方向。

数据流图有同步和异步两种方式。同步方式中，一条线在某个确定时刻只能保存一个数值，节点在新数值到来之前必须处理完旧的数据；异步方式中，每条线都有一个对应的数据队列，数据进出队列和节点处理数据是异步的。现在的行为综合工具都只能使用同步

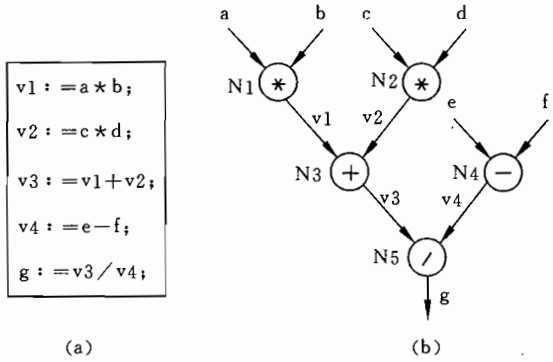


图 7.1　数据流图

数据流图。数据流图是在程序中表示表达式的理想形式，但是在表示控制结构上能力有限。

7.1.2　控制流图

控制流图(control-flow graph)适于表示各种控制结构，例如条件循环、条件转移、子程序调用以及意外情况处理等。控制流图也由节点和有向线组成。与数据流图不同的是，这里节点分为操作节点和分支节点两类，操作节点类似于数据流图，但含义更加广泛，可以是赋值、算术逻辑运算和子程序调用等，分支节点表示各种条件结构，对应到 VHDL 中，就是 if，case，while 等语句；控制流图的有向线不表示数据，而是表示节点之间的依赖关系。

图 7.2(a)中的程序是求两个整数最大公因子的VHDL算法描述，图(b)为控制流

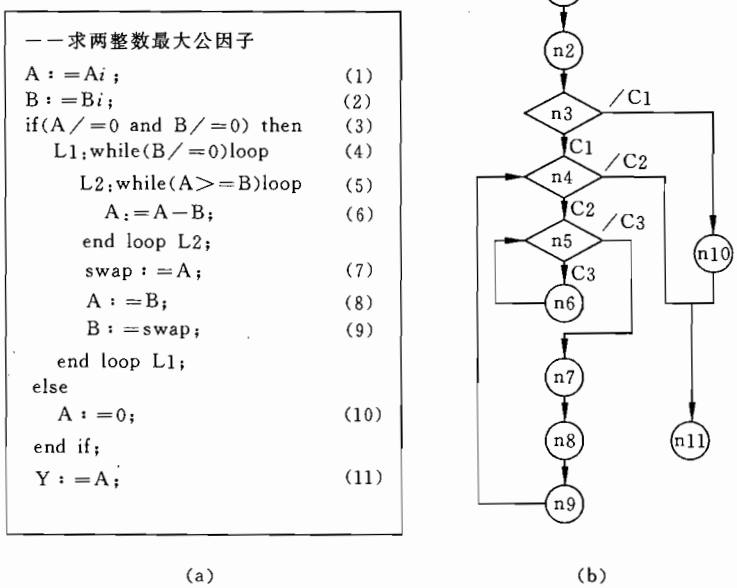


图 7.2　控制流图

图，其中节点的编号分别对应程序中相应行的编号，n3，n4，n5 是三个控制节点，其余都是操作节点。

控制流图非常适合于表示各种控制结构，但在数据流的分析和变换上能力有限。

7.1.3 控制数据流图

控制数据流图(CDFG control-data-flow graph)在数据流图基础上增加控制节点形成，综合了以上两种图的特点，是当前行为综合工具中最常用的设计表示形式。在这种图中，节点表示数据操作和控制结构，有向线表示节点之间的相关性。不同的行为综合工具中，CDFG 在具体格式上变化很大，这里我们用 Synopsys 中的 CDFG 格式来说明。

CDFG 中节点表示一定的操作，在某个控制节拍上完成。控制节拍通常是抽象的机器状态或时钟周期。节点分为 5 类：数据(data)节点、条件(condition)节点、层次(hierarchy)节点、占位(place holder)节点和循环(loop)节点。数据节点表示算术逻辑操作，大致相当于硬件描述语言中的运算符，如“+”，“and”等，有时相当于函数调用；其余 4 种节点都用于表示控制结构。

数据节点可以进一步分为如下几类：

(1) 合成节点，即能够与其他节点共享硬件的节点；

(2) 随机逻辑节点，即无法与其他节点共享硬件的节点；

(3) 补丁盒(patch box)节点，即数位和数域选择、连接运算等；

(4) 读写存储器节点，即访问存储器操作；

(5) I/O 读写节点，在 VHDL 中就是对信号和端口的读写操作。

条件节点对应于 VHDL 中的 if 和 case 语句，条件节点实际上总是成对地出现的，即由分裂节点和合并节点组成。层次节点可以包含下一层次的节点和有向线，对应于

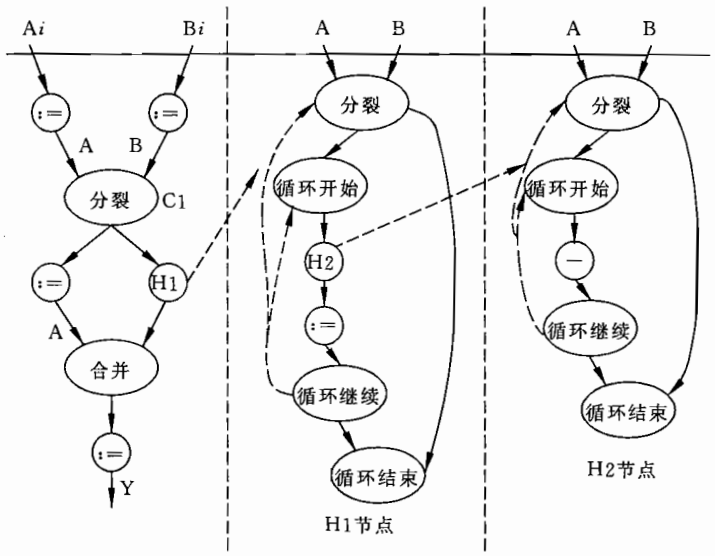


图 7.3 控制数据流图

VHDL 中的函数、过程调用等语句，也可以是一整个 loop 循环。占位节点表示跳过若干个状态，这是 Synopsys 独有的节点类型，用于综合结构报告和时序控制。循环节点分为循环开始、循环结束、循环退出和循环继续 4 种，用于定义循环边界。

CDFG 中的有向线分为两类：数据线和相关性线。前者表示各种数据，后者表示节点间的顺序和控制关系。一般表示顺序相关性的有向线画成实线，表示控制相关性的有向线画成虚线。

图 7.3 是图 7.2 中 VHDL 代码的控制数据流图。左边是整个程序的 CDFG，它有两个层次式节点 H1 及 H2(即中间及右边的图)，对应两个 while 循环语句。实际上，CDFG 表示可以描述任意复杂度的算法描述。通常行为综合工具将行为描述转换为 CDFG 后，还要把节点分配到具体的控制节拍上，然后就可以进行时序调度和硬件分配。对输出的优化可以通过调整节点的相互位置来实现。

7.2 行为综合的目标结构

行为综合器接受输入行为级描述后，将其转换为内部设计表示，然后经过调度、分配等步骤，形成最后的寄存器传输级代码作为输出。输入描述千变万化，而输出的寄存器传输级代码则必须依据一定的通用模型来建立。这个通用模型被称作是行为综合的目标结构，由控制单元、运算单元组成。其中控制单元一般是一个有限状态机，运算单元可以是数据路径或协处理器。本节以层次式的方式说明目标结构。

7.2.1 通用模型

常用的目标结构是带有数据路径的有限状态机(FSMD，FSM with Datapath)与带有数据路径和协处理器的有限状态机(FSMC，FSMD with Co-processor)。

1. FSMD 模型

FSMD 模型由 Gajski 等在 1992 年提出，是一种可以代表全部硬件设计的通用模型。FSMD 是在 FSM 基础上增加了数据操作能力，其定义为 $<S, I\times SS, O\times A, f, h>$。其中，S：FSMD 中 FSM 的状态集合；$I\times SS$：FSM 输入集合 I，被数据路径输出的条件信号扩展；$O\times A$：FSM 对外界的输出集合 O，并被内部变量赋值集合 A 扩展；f：下一状态生成函数，是从 S 到 $S\times(I\times SS)$的映射；h：输出函数，是从$(O\times A)$到 $S\times(I\times SS)$的映射。

图 7.4 是图形化的 FSMD 模型。根据这个模型，任何硬件都可以分解为相互作用的两部分：控制器和数据路径。控制器实际上是一个有限状态机，根据当前机器状态和数据路径的输出确定下一机器状态，并产生控制信号，送到数据路径；数据路径由功能部件(如 ALU、乘法器、浮点处理器等)、存储部件(如寄存器、RAM 等)以及通信部件组成，根据控制器输出的控制信号对数据进行各种处理。

图 7.5 是用 FSMD 实现一个简单电路的例子。其中图 7.5(a)是一个简单的时序电路，由 3 个状态和 5 个状态迁移组成。每一迁移都定义了相应的条件和迁移结束后要执行的操作。图 7.5(b)用 FSMD 实现了这个电路。

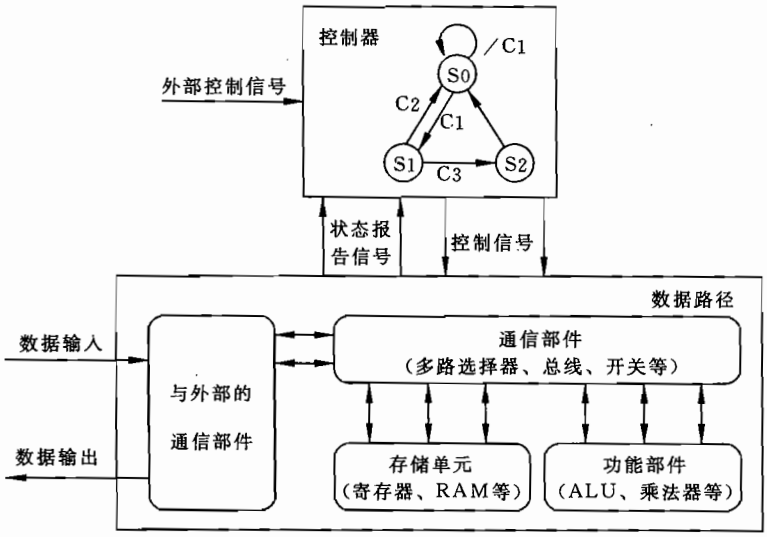
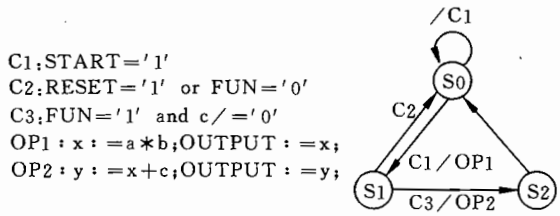
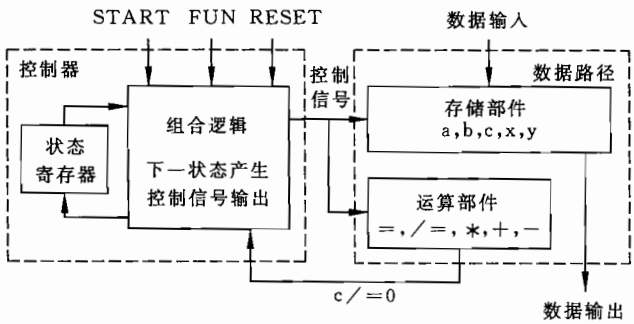


图 7.4　FSMD 模型



(a) 简单时序电路的状态转换图



(b) 与状态转换图等效的FSDM

图 7.5　用 FSMD 实现电路

2. FSMC 模型

FSMC 模型是在 FSMD 基础上发展的，它允许一部分操作在协处理器上进行，如图 7.6 所示。协处理器可以是复杂的运算电路或以异步方式接受控制器控制的功能部件，也允许层次式的结构，即这时协处理器本身就是 FSMD 或 FSMC 模型。

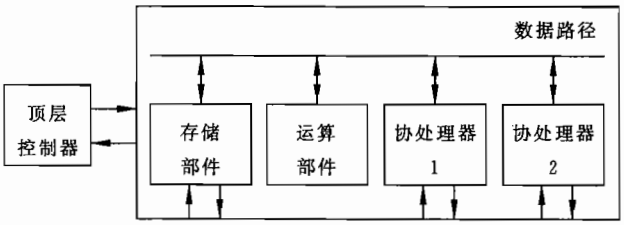


图 7.6　FSMC 模型

FSMC 被定义为 $\langle S \times \prod S_{ci}, I \times SS \times \prod I_{ci}, O \times A \times \prod O_{ci}, f, h \rangle$。

其中：

S_{ci}：协处理器的集合；I_{ci}：协处理器输入的集合；O_{ci}：协处理器输出的集合；$S \times \prod S_{ci}$：FSMC 的状态集合，被定义为 FSM 状态集合与协处理器局部状态集合的积；$I \times SS \times \prod I_{ci}$：FSMC 输入的集合，并被数据路径输出的条件信号和协处理器局部输入扩展；$O \times A \times \prod O_{ci}$：FSMC 输出的集合，并被内部赋值集合 A 和协处理器局部输出集合扩展。

7.2.2　控制器模型

行为综合器总是用有限状态机作为控制器，而且一般采用硬连线方式，也就是说用随机组合逻辑电路计算下一状态，当前状态以一定方式编码后由寄存器锁存。硬连线方式有限状态机不仅有完善的数学模型，更重要的是这种电路的综合技术已有非常充分的研究，因此寄存器传输级综合能够得到很好的结果。图 7.7 是控制器模型。

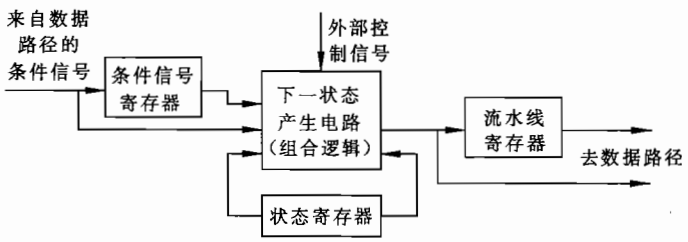


图 7.7　控制器模型

在行为级描述中，各个状态以隐含方式表达，由行为综合根据源代码进行状态分配和编码。现在的行为综合工具中，状态的更新都采用同步方式，用单相沿触发时钟方案。

在每一状态，控制器和数据路径要完成如下任务：

(1) 控制器进入新的状态，计算出控制信号和下一状态；

(2) 控制器把控制信号送往数据路径；

(3) 数据路径根据控制信号进行相应操作；

(4) 数据路径存储运算结果，并把各种条件信号送入控制器。

传统的硬件设计中，上述所有任务在一个时钟周期内完成。但实际上这 4 项任务由两

部分电路分别进行，存在着潜在的并行性。因此，可以引入流水线结构，加快信息的处理。基本的方法是在控制器的 I/O 端插入寄存器（如图 7.7 中所示），同时把工作时钟频率提高。在使用行为综合工具时，设计师通常可以通过综合约束来控制是否采用流水线结构。

7.2.3 数据路径模型

数据路径是以一定的拓扑关系互连的若干功能部件。现在 ASIC 设计的行为综合工具生成的数据路径都使用同步工作方式，由单相时钟沿触发实现同步。图 7.8 是数据路径通用模型的图形化表达。在这个模型中，数据存储在存储部件中，由功能部件进行各种操作，不同功能部件之间的数据交换通过通信部件，沿着控制信号指定的传输通路进行。

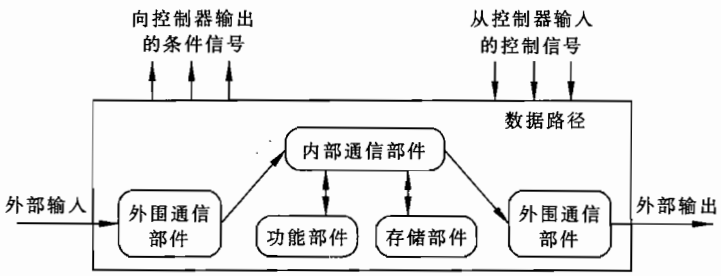


图 7.8 数据路径模型

1. 功能部件

功能部件是数据路径中执行由行为描述定义的各种操作的电路单元，这些操作包括：

(1) 逻辑操作：与、或、异或、非等简单逻辑运算以及如“与或非”等复杂运算等；

(2) 算术运算：+，−，×，÷，加 1 增量等；

(3) 关系运算：>，<，=，<=等；

(4) 移位运算；

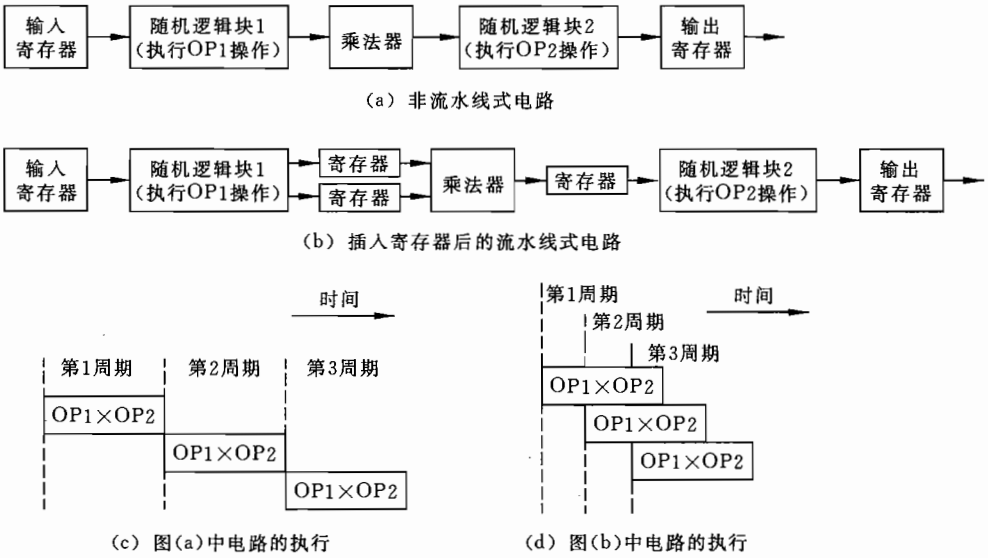
(5) 复杂运算：各种专用运算以及函数、过程调用，有时这些部件构成协处理器。

以上的操作将由行为综合器从设计库中挑选元件或综合出专用硬件来实现，每一功能部件均由一定的执行时间、面积和功耗等参数表征。设计空间的探索在很大程度上就是对不同功能部件的选择。

为了提高运算速度，常在数据路径内设置流水线，设计师能够通过合理的编程风格和综合策略决定是否使用流水线结构。在行为级综合的输出代码中，通过在各个具有大致相同执行时间的功能部件之间插入寄存器，保存每一级的运算结果。数据锁存后，各级功能部件即可处理下一个数据。图 7.9 是使用流水线结构的一个例子。

2. 存储部件

行为描述中的变量和常数在目标结构中由存储部件储存。一般说来，变量数组用寄存器堆或 RAM 实现，常数数组用 ROM 实现；单个变量用寄存器保存，单个常数用常数寄存器（即简化的寄存器）保存。此外，数据路径向控制器送出的条件信号也由寄存器锁存。



(a) 非流水线式电路

(b) 插入寄存器后的流水线式电路

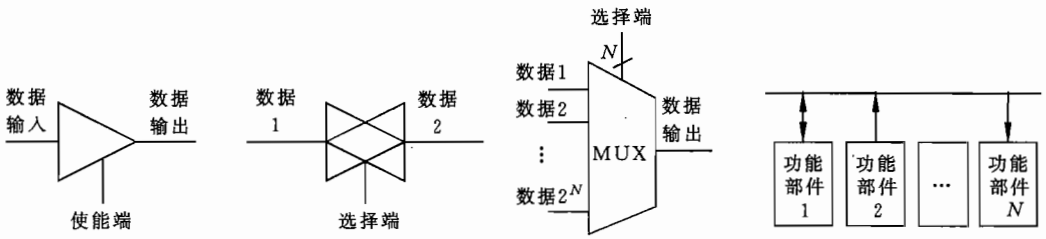
(c) 图(a)中电路的执行

(d) 图(b)中电路的执行

图 7.9 流水线式功能部件的实现

3. 通信部件

通信部件根据控制器的输出，构成各种数据通路，实现其他单元之间的数据交换。最基本的通信部件当然就是简单的信号线，其他常用通信部件是单向或双向开关、多路数据选择器和总线，如图 7.10 所示。



(a) 单向开关

(b) 双向开关

(c) 多路数据选择器

(d) 总线

图 7.10 常用通信部件

整个数据路径由通信部件以一定的拓扑结构连成网络，从全局来看，可以分为如下两类结构：

(1) 点对点式拓扑结构，这是功能部件之间最常用的互连结构。使用这种结构时，在任意两个具有数据依赖关系的部件之间都设置通信部件，如果某个功能部件有多路输入，就用 MUX 连接。

(2) 共享式互连拓扑结构，即使用总线，这是功能部件与存储部件及外围环境之间常用的互连结构。通常，功能部件与总线之间还要通过开关连接。这种结构最大限度地共享硬件通信资源，但是如果全部功能部件之间都使用总线的话，则需要极为复杂的总线协议算法。

7.3 行为综合流程

行为综合是高度交互的过程，在各个步骤中都要施加控制，因此有必要了解行为综合的工作过程。图 7.11 是行为综合的设计流程。设计起始于硬件描述语言行为级描述，首先对源代码进行词法和语法分析，通过检查的源代码被翻译为等价的设计表示，即 DFG，CFG 或 CDFG。接下来设计师设定设计约束，控制设计资源和设计策略，如指定使用哪些硬件资源，某两个操作之间相隔多少个时钟周期，行为级代码与寄存器传输级代码 I/O 的相对关系等。

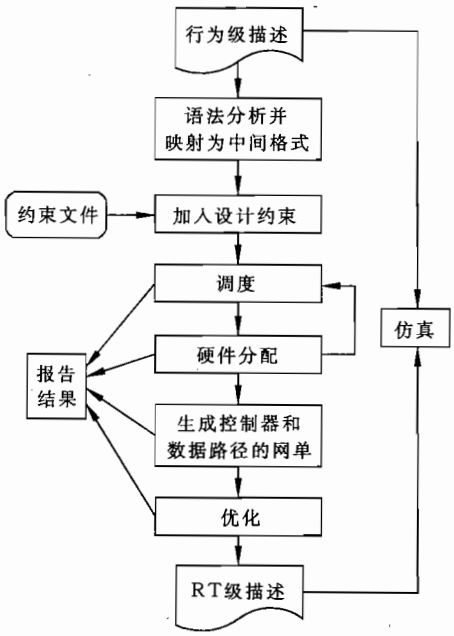


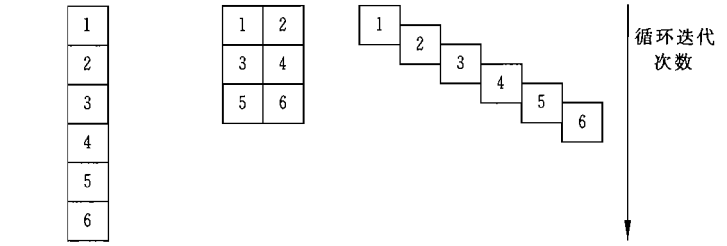
图 7.11 行为综合流程

然后行为综合工具对设计表示进行调度，即从设计表示中提取出所有控制和数据操作，并把它们分配到一定的时钟周期上。调度的首要目标是在满足输入的时间约束下寻找硬件代价最低的方案。通常，行为综合工具根据一些原则如数据是否准备好、硬件代价等，对 CDFG 中的节点（即操作）量化，从而确定调度的优先顺序。调度算法现在已经比较成熟，一般说来各种算法都是利用 CDFG 节点之间的并行性增加硬件，以减小执行时间，或者通过调整 CDFG 结构来获得优化的结果。调度完成后，也就形成了有限状态机控制器。

以上是调度要完成的基本任务，为了得到优化的行为综合结果，现在的行为综合工具还分别采用了下述技术：

（1）循环语句优化。对于有着固定循环次数的循环语句（如 VHDL 中的 for I=1 in 10 loop），常用的优化策略是循环展开；而对于无限循环语句（如 VHDL 中的无条件 loop 语句），可以采用引入流水线结构的办法，即所谓循环折叠技术。这时，loop 循环体内的全部语句被看作单个操作序列，循环的下一次迭代在本次循环结束之前就开始执行。图 7.12 是顺序执行循环、循环展开和循环折叠的示意图。

（2）存储器 I/O 推断。在寄存器传输综合的设计流程中，存储器特别是 RAM 是很难处理的，主要有两方面的问题：第一，ASIC 设计经常使用大量的 RAM，因此在仿真时必须提供 RAM 的精确时序模型，在建模上难度很大，同时，不同的存储器接口特性不尽相同，因此限制了设计的兼容性和工艺无关性；第二，在寄存器传输级综合时，一般无法直接综合存储器，所以常用的办法是在仿真时使用存储器时序模型，在综合时去掉这部分代码进行综合，再用修改网单或编辑电路图的方式与用其他方法生成的存储器相连接。这种办法费时、费力，容易发生错误，特别是在地址译码逻辑比较复杂时极为困难。现在的行为综



(a) 顺序执行循环　(b) 循环展开　(c) 循环折叠

图 7.12　循环结构的处理

合工具，如 AMICAL 和 BC，提供了存储器 I/O 推断能力，即可以在代码中以访问数组的方式隐含表示读写存储器，行为综合器自动调度并生成存储器控制逻辑，并且在下一步的硬件分配中选择适当的存储器。

(3) 可以由用户控制的多种 I/O 调度模式。在行为综合过程中，绝大多数调度算法都要在不改变整体功能的前提下，对输入行为描述中的操作的顺序进行调整，这种调整对于全局优化和资源共享都是必要的。但是对于模块的 I/O 操作，即对模块内信号或端口的读写操作，重新排列顺序不仅会给行为级与寄存器传输级仿真结果的比较带来困难，而且可能导致输出的寄存器传输级代码外部行为的改变。特别是在使用由顶向下的设计方法，而各个模块之间必须遵循一定的通信协议时，这种改变很可能会造成整个设计的失败。因此 BC 和 AMICAL 都引入了用户控制 I/O 调度的概念，用户能够用以下的方式来控制 I/O的调度结果：

① 周期固定(cycle-fixed)I/O 模式。在这种模式下，输入的行为级代码与输出的寄存器传输级代码的 I/O 时序保持严格一致，即在每一时钟沿上，I/O 操作完全相同。如果使用这种模式，行为级与寄存器传输级的仿真可以使用同样的测试激励，输出结果也应该是一样的。

② 超状态固定(superstate-fixed)模式。使用这种模式对 VHDL 行为描述进行调度时，设计师可以把两个 wait 之间的语句定义为超状态。行为综合器将超状态分配到若干个时钟周期上，I/O 读操作可以发生于超状态内的任一周期，但 I/O 写操作总发生在执行整个超状态的最后阶段。

③ 自由浮动(free-floating)模式。在这种模式下，行为综合器自由安排 I/O 操作的时序，也就是说，I/O 操作的相对顺序可能被打乱，同时输出的寄存器传输级代码不再与输入行为级代码保持逐个周期的一致。加入这一模式是为了使设计师能够更加充分地探索设计空间。

以上的三种模式是 Synopsys 公司的提法，其他行为综合工具在具体命名方式上有所不同。VHDL 代码与 I/O 调度模式的关系将在第 7.4 节详细讨论。

(4) 推测式执行。这是 Synopsys 的 BC 独有的特征，在进行调度时，综合器在计算条件的操作结束之前就安排条件分支的操作，并在目标代码中生成控制结构，以便能够忽略未被选中的分支的结构。使用这种方法可以有效地缩短关键路径的长度。

(5) 过程语句的处理。在算法模型中，经常要使用子程序，在 VHDL 中就是过程和函数语句，其中函数可以看作是过程的特例。一般行为综合工具有以下一些处理方式：最基本的方法是在读入源代码时把过程调用语句替换为相应过程的代码，这样当然增加了控制器的复杂度；第二，把调用过程映射到单独的硬件，另外进行综合，如果过程被多次调用，还可以实现资源共享；第三，AMICAL 提供了比较独特的处理方法，它将过程翻译为设计库中某一元件的一系列操作，并将这一元件看作协处理器。

(6) 用可编程式硬件实现 FSM 控制器。通常的行为综合器都用硬连线随机逻辑电路实现 FSM 控制器，AMICAL 提供了用微码 ROM 实现的功能，这主要是为了提高设计的可再用性和灵活性。

硬件分配的任务是将经过调度的各个节点上的操作映射到数据路径中具体的功能部件上，换言之，通过这个步骤就确定了数据路径的寄存器传输级结构。一般说来，硬件分配要结合一定的高层次综合单元库进行。单元库中存放着各种功能部件的高层次模型及其硬件参数，行为综合工具从库中选择能够以最低代价满足时序要求的元件。硬件分配还要解决资源共享的问题，常用的办法是让互斥的 CDFG 节点共享硬件。调度和硬件分配实际上是迭代的过程，例如，操作 OP1 被调度在一个时钟周期内完成，而分配时单元库内没有任何硬件能够达到这一要求，这时必须重新进行调度。

调度和分配完成之后，就可以生成内部网单，经过优化最后形成输出的寄存器传输级代码。

7.4 行为综合的 VHDL 建模

本节讨论用 VHDL 语言建立行为级模型及其与行为综合过程的相互关系，必须指出的是，不同行为综合工具支持的 VHDL 语句集和建模风格都不尽相同，我们主要说明具有通用性的语言结构。

与寄存器传输级描述类似，行为级描述通常也由一系列进程(process)组成。行为综合器只对进程内部的语句进行调度和分配等处理，进程外的并行语句仍然使用寄存器传输级和门级的综合技术。由于 VHDL 中的一些语言结构主要是用于调试和仿真的，所以行为级综合工具只支持 VHDL 中的可综合子集，不支持的语言结构主要有断言语句、延时语句(包括传输和惯性延时)、文件操作语句和一些属性。

7.4.1 行为级与寄存器传输级建模的区别

行为级模型与寄存器传输级模型在建模风格、处理方式等方面都有很大区别，表 7.1 是对典型行为级和寄存器传输级描述的比较。

行为级与寄存器传输级模型最主要的区别在于表示时序电路状态的方式。行为模型中，状态是以隐含方式表达的，因此在行为综合的优化过程中，状态总个数常会发生改变，同时，各个操作不必严格定义在某一状态上。而使用寄存器传输级综合的设计方法时，设计师要用手工方式分配状态，建立状态转换图，并把操作分配到状态上。在寄存器传输级模型中，状态必须以显式的方式表达，而且一般优化时不能增减状态和改变操作时序。下

表 7.1 行为级与寄存器传输级描述的区别

内 容	寄存器传输级描述	行为级描述
描述对象	结构	算法
描述风格	逐个周期描述	电路功能和整体时序
设计划分	由设计师手工进行	自动划分成数据路径、控制单元以及存储器
时序调度	由输入确定	自动进行
有限状态机	由输入确定状态及状态转换关系	自动进行
存储部件	由输入指定	自动进行
程序长度	长	短
可读性	差	好
仿真速度	慢	快
搜索设计空间的能力	由输入确定整体结构，所以只能进行优化	能够尝试多种目标结构

面的 VHDL 源代码分别是 FIR 滤波器的寄存器传输级模型和行为级模型，可以看出行为级模型要简洁得多。

```
--(a) FIR 滤波器的寄存器传输级模型
RTL_FIR: process
    variable X,U,TMP : Integer;
    type state is (S0,S1,S2,S3);
    signal CURENT,NEXT : state;
begin
    Case CURRENT is
        When S0 =>
            X := 0;
            if stop /= '1' then
                NEXT<=S1;
            Else
                NEXT<=S0;
            end if;
        When S1=>
            U := INPUT;
            X := X * A;
            NEXT<=S2;
        When S2=>
```

```
            TMP := U * B;
            NEXT<=S3;
        When S3=>
            X := X + TMP;
            OUTPUT<=X;
            if stop /= '1'then
                NEXT<=S1;
            Else
                NEXT<=S0;
            end if;
    end Case;
    CURRENT<NEXT;
    wait until CLK<='1';
end process RTL_FIR;

-- (b) FIR 滤波器的行为级模型
Behavioral_FIR: process
    variable X,U : Integer;
begin
    while stop /= '1' loop
        U := INPUT;
        X := A * X+B * U;
        OUTPUT<=X;
        wait until CLK<='1';
    end loop;
end process Behavioral_FIR;
```

行为级模型与寄存器传输级模型的另一主要区别是对变量的处理方式。在寄存器传输级综合时，现在大部分综合工具不支持变量；而少数支持变量的综合工具如 Synopsys 的 Design Compiler，根据以下原则来处理：如果在一确定状态某个变量先被读，然后被写入，则要为这个变量分配寄存器，这样的处理方式显然存在很大的局限性，不仅设计师必须仔细推敲是否引入寄存器，而且不同的变量无法共享同一寄存器。行为综合工具都支持变量，行为综合器自动计算存储全部变量所需的寄存器个数，并且能够优化寄存器分配。

此外，在寄存器传输级模型，对存储器的访问、插入流水线、条件操作等都需要专门进行设计，也就是说，必须编写额外的寄存器传输级代码来实现。而在行为级模型中，这些都可以由行为综合器自动生成。

除了建模风格的不同，行为综合器和寄存器传输级综合器在实现一些具体的 VHDL

语言结构时有着不同的方式，表 7.2 列出了这些差别。

表 7.2　寄存器传输级与行为级综合实现 VHDL 语言结构的区别

	寄存器传输级	行　为　级
RAM	数组被综合为一组信号	允许把数组映射到 RAM 上，对数组的访问转换为对 RAM 的读写操作
for 循环	大多数综合器采用循环展开的处理方法，而且 for 循环内不能包含 wait 语句	允许选择顺序执行循环、循环展开或以流水线实现循环，for 循环内部可以出现 wait 语句
时序元件	只能推断单状态的时序模块(如 D 触发器)及与之相连的组合逻辑电路，多状态时序元件只能用元件例化(component port map)的方式调用	能够推断多状态时序元件
多周期组合逻辑操作	所有组合逻辑操作只能在一个时钟周期内完成	能够把复杂逻辑操作映射到多个时钟周期上

7.4.2　行为描述的进程

复杂的 ASIC 算法要依靠进程来描述，而行为综合器也只对进程进行调度和分配。如果输入代码包括寄存器传输级的进程，行为综合器一般不做处理。行为级描述通常由一组进程组成，各个进程被分别处理。也就是说，对某一进程的调度结果不会影响其他进程的调度。同时，现有的行为综合器通常会尽量保持各个进程之间 I/O 的调度结果仍然能够同步。但是，除了使用固定 I/O 模式，行为综合器不能保证输出的同步，为解决这个问题，这时设计师应使用其他的接口通信协议，而不是单纯依靠输入描述中的全局时钟；或者通过设置综合约束控制 I/O 时序。

由本节中的例子可以看出，行为描述的进程在编程风格上类似于软件编程，通常大量使用变量(variable)，这是由于与信号(signal)相比，变量的调度和分配都要灵活得多。在行为综合后，变量被映射到寄存器上。行为进程一般使用 wait 语句作为定时控制。根据进程是否使用状态信息，进程被综合为控制 FSM 及相应的数据路径或组合逻辑块。

7.4.3　定时和复位

为了描述时序系统，首先要解决定时和复位的问题。现有行为综合工具只支持单相沿触发时钟方案，但复位可以选择同步或异步方式。

1. 定时

VHDL 行为级模型中用 wait 语句把算法描述和时序联系起来，也就是说，wait 语句迫使程序挂起，等待下一个有效时钟沿再继续执行。用 wait 语句表示有效时钟沿的典型方式如下：

```
wait until CLK='1' and CLK'Event;        --上升沿
```

```
wait until CLK='0' and CLK'Event;        --下降沿
```

行为进程中最多只能有一个信号作为时钟(行为综合工具把上面 wait 语句内的信号看作时钟),可以使用上升或下降沿,但同一进程内部只能使用同一种时钟沿。下面的代码是使用 wait until 语句同步的一个例子。

```
-- 用 wait until 语句定时
Ex1: process
    ......
begin
    wait until CLK='1' and CLK'Event;
    C := X+D;
    Y := C*X;
    Z := K*Y;
    wait until CLK='1' and CLK'Event;
    C := Z*D;
    Y := Z*E;
end process;
```

在对输入的行为描述进行处理时,每一 wait 语句实际上相当于指定了一个状态,两个 wait 语句(包括循环式执行的 wait 语句)之间的语句被称为线程。这样,行为进程被分解成为控制 FSM 和数据路径,wait 语句相当于 FSM 的状态,线程相当于一个状态上由数据路径上执行的一组操作。但如果线程内结构非常复杂,比如包含循环,则调度时可能把线程分配到若干个控制步上。

2. 复位

为了全面描述电路行为,行为模型可以使用同步或异步复位方式。如果使用同步复位,一般的处理方法是把整个进程的行为描述嵌入一个循环,并使用 exit 语句使得复位信号有效时能够进行相应处理。实际上,进程的行为本身就是一个信号(执行到进程末尾时应跳回进程开始处执行),引入循环的目的是能够有效控制程序执行顺序。下面代码是 80C51 微控制器的行为描述的框架,这段程序实现了同步复位。

```
-- 在行为描述中使用同步复位
80C51_CPU: process
    ......;        -- 变量和信号以及其他各种定义
begin
    reset_loop: loop
        PC_Q := (others=>'0');
        SP_Q := "00000111";
        Init_RAM(ram);
        ......;        --为其他特殊寄存器赋初值
```

```
        wait until CLK'Event and CLK= '1';
        if RESET = '1' then        --reset 高电平有效
            exit reset_loop;
        end if;
        run_loop: loop
            IR_Q := rom(PC_Q);      --取指令
            wait until CLK'Event and CLK= '1';
            if reset = '1' then
                exit reset_loop;
        end if;
        case IR_Q is              --指令译码
            when "00000000"=> ......;
            when "00000001"=> ......;
            ......;
        end case;
        end run_loop;
    end reset_loop;
end process 80C51_CPU;
```

这段代码由两个嵌套的 loop 循环组成。程序从 reset_loop 开始执行，进入进程的第一个状态。这相当于硬件上电复位，进入初始化阶段，把程序计数器 PC 清 0，并把堆栈指针寄存器 SP 置为 7，然后对 RAM 和其他特殊功能寄存器赋予初值。接下来是定时的 wait 语句和判断复位信号是否有效(reset='1')的 if 分支语句。如果有效，则停止后面代码的执行，退出循环，这时执行到进程最后，返回到循环起始处再次进行复位的初始化操作。由于这一动作发生在 wait 语句指定的时钟沿之后，从而实现了同步复位。如果无效，程序转入正常情况的处理，就是 run_loop 循环。取指和译码之间又插入了 wait 和复位判断语句，这样在复位信号无效时，程序反复执行 run_loop 循环；否则，跳出当前循环，回到 reset_loop 的开始部分执行复位代码。

如果使用异步复位风格，可以把前面的 wait 和 if...end if 语句组修改为下面的形式。

```
-- 在行为描述中使用异步复位
wait until ((CLK'Event and CLK='1') or
(RESET'Event and RESET='1'));      --RESET 高电平有效
if RESET = '1' then
    exit reset_loop;
end if;
```

合理使用以上的复位建模方式，不仅可以建立准确的行为级仿真模型，而且能够有效地引导行为综合器形成复位电路。

7.4.4 信号和变量

信号(包括进程内部使用的信号和元件与外界通信时使用的端口)、变量和常量是VHDL语言的三种基本数据对象。常量的处理比较简单,用ROM或常数寄存器存储即可,本小节讨论在行为级模型中如何使用信号和变量,以及行为综合工具对它们的相应处理。

行为级模型中的信号的使用方式与寄存器传输级模型基本相同。值得注意的是,由于行为综合工具忽略信号赋值时的延时,所以如果信号在一个线程内部被赋值多次,那么除最后一次外的其他赋值将被忽略。行为综合过程中,所有信号都将被保留,如果在进程内信号的数值需要保持一个周期以上,则信号被映射到存储单元上,否则用连线实现。

在VHDL行为模型中,变量的行为类似于软件编程语言中的变量。变量只能定义在进程、过程和函数中,进程中的变量在两次访问之间保持原值,而过程和函数中的变量在被调用结束后不保留数值。不同的行为综合工具对变量的综合结果差别很大,但总体上大致有下列几种处理方式:

(1) 把每一变量都映射到单独的寄存器上,这种方式的好处是综合前后的变量是完全一致的,但是显然增大了电路规模;

(2) 根据对变量数据有效期的分析,安排不同变量共享同一个寄存器,这是现在的行为综合工具处理变量的主要方式;

(3) 把变量映射到存储器上,这是行为综合工具处理变量数组的主要方式。

一般说来,行为综合工具都尽量使不同的变量共享寄存器。但在一些特殊情况下,甚至会发生一个变量使用多个寄存器的情况。因此在建立行为模型时,应注意使用合理的编程风格帮助行为综合器实现硬件共享。

变量能否共享同一个物理上的寄存器,取决于变量的生存期。变量的生存期指的是从变量被第一次计算到最后一次被使用的时钟周期数。如果两个变量的生存期不重叠,则可以共享。例如在下面的第(a)段代码中,V1的生存期开始于(A+B)的计算,结束于向X输出;V2的生存期开始于(A−B)的计算,结束于向Y输出。V1和V2的生存期互不重叠,因此可以用同一个物理寄存器存储V1和V2。同样,如果两个变量由互斥的条件路径生成,则也可以共享。但是在第(b)段代码中,变量V的生存期有两种可能:如果条件不满足,那么V的生存期是从计算(A+B)到向X输出;如果条件满足,V的生存期变成从计算(A−B)到向X输出。这样,由于两种情况下V的生存期不同,行为综合器要为V分配两个寄存器,显然是造成了资源的浪费。实际上,只要把(b)中的代码修改为(c)的形式,就可以满足共享的条件,因为这时两条路径上V的生存期是完全相同的。

```
-- (a) 可以共享寄存器的变量
process
    Variable V1,V2 : Integer;
begin
    V1 := A+B;
    wait until CLK'Event and CLK='1';
```

```
        X<=V1;
        V2 := A-B;
        wait until CLK'Event and CLK='1';
        Y<=V2;
    end process;

    -- (b) 一个变量使用两个寄存器
    V := A+B;
    wait until CLK'Event and CLK='1';
    if COND='1' then
        wait until CLK'Event and CLK='1';
        V := A-B;
    else
        wait until CLK'Event and CLK='1';
    end if;
    X<=V;

    -- (c) 把(b)修改为可以共享寄存器
    V := A+B;
    wait until CLK'Event and CLK='1';
    if COND='1' then
        wait until CLK'Event and CLK='1';
        V := A-B;
    else
        wait until CLK'Event and CLK='1';
        V := V;
    end if;
    X<=V;
```

7.4.5 I/O 调度

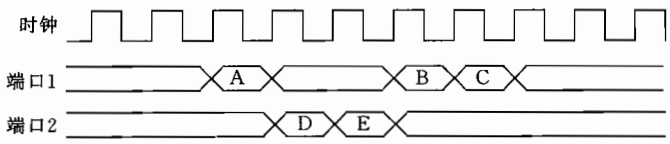
行为综合中必须特殊注意的地方是对信号时序的调度。因为信号的作用是提供进程与外界环境的通信，所以对除了复位和时钟之外的信号的调度又被称作 I/O 调度。7.3 节已经介绍了三种 I/O 调度描述：固定模式、超状态固定模式和自由浮动模式，它们的时序如图 7.13 所示。设计师可以通过综合命令对某一部分代码选择某种调度模式，但同时也要使行为模式代码能够满足这一模式的要求，并且同一个进程只能采用一种 I/O 调度模式。

1. 周期固定 I/O 调度模式

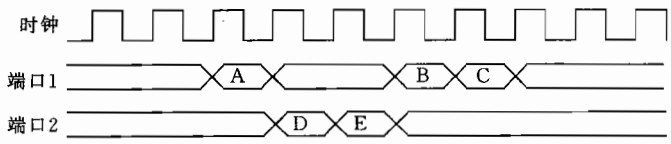
在这种模式下，行为综合生成的寄存器传输级模型的 I/O 操作必须与原来的行为描

述在相同的时钟周期上，因此，计算结果的操作必须能够在读取输入和写回结果之间的时间内完成。同时，所有 wait 语句必须是“wait until 有效时钟沿”的形式，每一 wait 语句必然产生一次状态迁移。

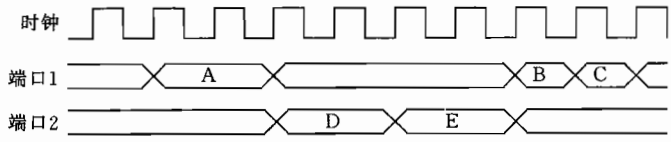
使用固定 I/O 模式，必须注意读写操作发生的时钟周期。比如在下面 VHDL 代码的第(a)段中，X，A，B，C 都是信号，这段程序要求在一个周期内完成读取 A，B，C，计算乘加运算结果和写回 X。如果乘法-加法器完成计算需要两个时钟周期，那么这段代码是无法以固定模式调度的。为了满足要求，需要在程序中加入一个临时变量，从而把读写操作分配到两个周期上，如第(b)段代码所示。



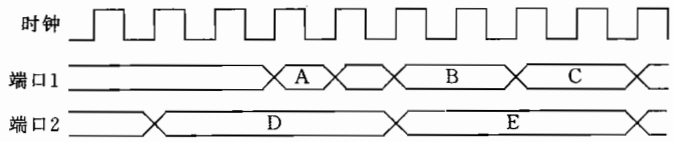
(a) 输入行为描述



(b) 周期固定I／O调度模式输出的结果



(c) 超状态固定I／O调度模式输出的结果



(d) 自由浮动调度模式输出的结果

图 7.13　三种 I/O 调度模式

```
-- (a) 无法以固定模式调度的代码
wait until CLK'Event and CLK='1';
    X<=A * B+C;      --错误，无法调度
wait until CLK'Event and CLK='1';

-- (b) 修改后的代码
wait until CLK'Event and CLK='1';
    Tmp := A * B+C;
```

```
    wait until CLK'Event and CLK='1';
        X<=Tmp;
    wait until CLK'Event and CLK='1';
```

固定 I/O 模式下，循环语句和条件分支语句的处理也是需要认真对待的问题。行为综合时，循环结构(包括 while，for 和无条件 loop 循环)的状态图至少包含一个状态，也就是说，每个循环结构内至少要包含一个 wait 语句。当控制 FSM 要依靠数据路径的某个结果来产生控制信号控制其他数据运算时，称作控制链化(control chaining)。此情况下，必须保留足够的时钟周期，否则，无法调度。例如在下面的 VHDL 源代码的第(a)段中，STATUS 是一个信号，exit L1 暗示存在状态转移，这时若 STATUS 为 true，则下一状态是 end loop 之后的状态，若 STATUS 为 false，则下一状态是循环 L1 的第一状态。状态转移还要求一个额外的周期，因此这段代码不能以固定模式调度，需要修改为下面第(b)段形式。同样，下面第(c)段代码也不能满足调度条件，因为如果 READY 为 1 的话，while 循环仍然需要一个状态去计算条件结果，所以需要插入一个周期的延时，即修改为下面第(d)段的形式。

```
-- (a) 无法以固定模式调度的代码
L1: loop
    wait until CLK'Event and CLK='1';
    if STATUS='1' then
        exit L1;
    end if;
end loop L1;

-- (b) 对(a)做修改后的代码
L1: loop
    wait until CLK'Event and CLK='1';
    if STATUS='1' then
        exit L1;
    end if;
    wait until CLK'Event and CLK='1';
end loop L1;

-- (c) 无法以固定模式调度的代码
while (not ready='1') loop
    wait until CLK'Event and CLK='1';
end loop;
DATAOUT<=DATA;
```

```
-- (d) 对(c)做修改后的代码
while (not ready='1') loop
    wait until CLK'Event and CLK='1';
end loop;
wait until CLK'Event and CLK='1';
DATAOUT<=DATA;
```

总括起来，在使用 if，case，loop 和 end loop 控制流语句时，都需要在其后面添加 wait 语句。

同样，如果分支操作内的数据运算需要多个时钟周期完成，则必须加入足够的周期延时才可以进行固定模式的 I/O 调度。比如在下面 VHDL 源代码的第(a)段中，STATUS 和 OUTPUT 是信号，A，B 是变量，若读取 STATUS 和条件判断需要两个时钟周期，A * B 操作需要三个时钟周期，则需要修改为第(b)段的代码才能满足固定模式 I/O 调度。

```
-- (a) 无法以固定模式调度的代码
if STATUS='1' then
    OUTPUT<=A * B;
end if;

-- (b) 插入周期延时的代码
if STATUS='1' then
    wait until CLK'Event and CLK='1';
    wait until CLK'Event and CLK='1';
    OUTPUT<=A * B;
    wait until CLK'Event and CLK='1';
    wait until CLK'Event and CLK='1';
    wait until CLK'Event and CLK='1';
end if;
```

2. 超状态固定 I/O 调度模式

在超状态固定 I/O 调度模式下，用 wait 语句把进程划分为一些超状态，每个超状态由一个或多个普通状态组成。这时，I/O 写操作只能发生在超状态的最后一个时钟周期，I/O 读操作可以发生于超状态内的任意周期。也就是说，一个超状态内，所有 I/O 读操作均在 I/O 写操作之前。这种调度模式保证输出代码的 I/O 顺序与输入代码相同，但是 I/O 之间的时钟周期数可能发生变化。

由于读操作可以在超状态内浮动，所以被读的输入信号必须在超状态期间保持稳定。同时，由于超状态的周期数是可变的，输入描述的 I/O 协议必须能够处理这种情况。

循环结构中可能包含多个 next 语句，若某个超状态内执行有可能返回循环起始处的语句(如 next 语句)，则称为继续超状态(continuing superstate)，同时循环的最后一个超状态必然也是一个继续超状态。如果循环的第一个超状态含有 I/O 操作，那么继续超状

态内不能包含 I/O 写操作。这是由于实际上继续超状态将与循环的第一个超状态合并，这种 I/O 写操作会造成硬件行为的错误。比如在下面 VHDL 代码的第(a)段中，第二个时钟沿后的语句构成了继续超状态，其中含有 I/O 写操作，与循环的第一个超状态合并后，I/O写操作就发生于循环入口。然而，按照程序语义，循环入口是不应该有 I/O 写操作的。修改的办法是在继续超状态的最后插入 wait 语句。与此类似的是，循环前的写操作与第一个超状态内含有 I/O 写操作的循环结构之间也需要插入 wait。再比如下面第(b)段的代码，如果 loop 之前没有 wait 语句，则循环前的写操作显然与循环内的写操作冲突。

```
-- (a) 继续超状态内的 I/O 写操作
while (NOT READY='1') loop
begin
    A := INPUT_A;
    B := INPUT_B;
    wait until CLK'Event and CLK='1';
    --各种运算
    wait until CLK'Event and CLK='1';
    Z<=RESULT;      --无法以超状态固定 I/O 模式调度
    --应在这里插入 wait 语句
end loop;

-- (b) 循环前后的 I/O 写操作
A_PORT<=A1;
--应在这里插入 wait 语句
L1: Loop
begin
    A_PORT<=A2;
    wait until CLK'Event and CLK='1';
end loop L1;
```

使用超状态固定 I/O 调度模式，由于 I/O 写操作总发生在最后，也就是说，发生在下一个超状态 I/O 读操作之前或与之同时。所以，必须注意在适当位置，特别是在循环结构前后和循环出口(exit)处插入 wait 语句，以分割各个超状态。否则，很容易导致错误。

3. 自由浮动 I/O 调度模式

自由浮动 I/O 调度模式下，I/O 操作的相对顺序可以自由浮动，行为综合工具依照设计约束寻找最低硬件代价的方案。对于同一个信号或端口来说，读操作可以保持原有顺序，而只要被写的数据准备好，则写操作的顺序完全由行为综合工具自行安排。不同信号和端口之间的 I/O 操作的顺序完全是打乱的。这种模式对代码几乎没有什么特殊要求。

7.4.6 条件分支控制结构

行为级描述中的 wait 语句相当于一个状态。VHDL 语言中的一些分支控制结构常常暗示发生状态转移或者需要插入额外状态去计算条件，这时无论以何种 I/O 方式调度，都必须在行为模型中插入相应的 wait 语句，才能满足行为综合的要求。这些分支控制结构包括：if…else…end if，while 和 for 循环，loop 中的 exit，next 等。

例如在下面 VHDL 代码的第(a)段中的 if 语句，输出描述中的控制器需要一个状态来接收来自数据路径的条件值并加以检验，并且此时数据路径不能进行任何运算，只有在下一状态，控制器输出相应控制信号之后，条件分支上的操作才能开始，所以要通过插入 wait 语句来指示电路增加一个状态。同样，while 和 for 循环也要计算条件值，因此在循环入口和 end loop 语句之后都要插入一个状态，如下面 VHDL 源代码中的第(b)段。如果条件值的计算需要多个时钟周期，还要插入相应的状态，比如下面 VHDL 代码第(c)段中条件的计算需要 3 个周期，则需要插入 3 条 wait 语句。当然，在综合之前很难确定具体条件计算所需要的周期数，解决这一问题有两种方法：一种是通过加入综合约束命令条件使计算在一定的周期内完成，另一种办法是根据经验预先插入足够多的状态，然后根据综合结果报告进行调整。在循环结构中包含 exit 或 next 语句时，情况更为复杂一些，但总的原则是只要存在状态转换，就要插入 wait 语句。例如在下面代码的第(d)段，exit 和 next 都造成了状态转移。

```
-- (a) 分支语句的周期延时
if COND='1' then
    wait until CLK'Event and CLK='1' ;
    V := A * B;
else
    wait until CLK'Event and CLK='1' ;
    V := A+B;
end if;

-- (b) 条件循环语句的周期延时
while_loop: while (COND='1') loop
begin
    wait until CLK'Event and CLK='1' ;
    --循环的各种操作
end loop;
wait until CLK'Event and CLK='1' ;

-- (c) 条件的计算需要多个周期的 while 循环
while_loop: while (A * B > 10) loop
```

```
begin
    wait until CLK'Event and CLK='1' ;
    wait until CLK'Event and CLK='1' ;
    wait until CLK'Event and CLK='1' ;
    ——循环的各种操作
end loop;
wait until CLK'Event and CLK='1' ;
wait until CLK'Event and CLK='1' ;
wait until CLK'Event and CLK='1' ;

——(d) loop 循环内含有 exit 和 next 语句
L1: loop
begin
    ......
    if EXIT_COND='1' then
        wait until CLK'Event and CLK='1' ;
        exit L1;
    end if;
    wait until CLK'Event and CLK='1' ;
    ......
    if SKIP_COND='1' then
        wait until CLK'Event and CLK='1' ;
        Next L1;
    end if;
    wait until CLK'Event and CLK='1' ;
    ......
end loop;
```

7.4.7 用流水线方式实现循环

流水线是充分利用硬件内部的并行性，增加数据处理能力的有效方法。使用流水线，硬件一定要执行某种迭代式的操作，比如微处理器周而复始地执行取指、译码、执行和写回结果，或者 FIR 滤波器对输入序列进行的迭代式运算。实际上，VHDL 进程就是反复执行的，但是一般来说，只有部分硬件需要使用流水线，而且像复位等操作不必进入流水线。因此，行为级模型中，需要流水线化的主要是循环结构。

下面 VHDL 源代码的第(a)段描述了能够用流水线实现的 loop 循环，假定其中 OP1～OP4 是由组合逻辑完成的算术操作，都能够在一个时钟周期内执行完毕。应该指出的是，仅有这段代码还不足以使行为综合器生成流水线式电路，还需要施加必要的设计约束。使用流水线实现循环，当然是以增加硬件为代价的，因为如果这样做的话，则 A，B，C，

D 不能共享寄存器，而且 OP1～OP4 也不能共享其他硬件资源。为了提供一个比较，下面 VHDL 代码的第(b)段列出了用寄存器传输级描述实现这个流水线式电路的代码。其中第(b)段中进程 R1～R3 相当于三个 D 触发器，用于锁存中间运算结果，同时 A,B,C,D 是用信号实现的。

```
——  (a) 流水线式电路的行为级描述
Loop
begin
    A := INPUT_PORT;
    wait until CLK'Event and CLK='1';
    B := OP1(a);
    wait until CLK'Event and CLK='1';
    C := OP2(b);
    wait until CLK'Event and CLK='1';
    D := OP3(c);
    wait until CLK'Event and CLK='1';
    OUTPUT_PORT<=OP4(d);
    wait until CLK'Event and CLK='1';
end loop;

—— (b) 流水线式电路的寄存器传输级描述
Proc1: process(INPUT_PORT) begin
    A<=OP1(INPUT_PORT);
end process;
R1: process begin
    wait until CLK'Event and CLK='1';
    A_REG<=A;
end process;
Proc2: process(A_REG) begin
    B<=OP2(A_REG);
end process;
R2: process begin
    wait until CLK'Event and CLK='1';
    B_REG<=B;
end process;
Proc3: process(B_REG) begin
    C<=OP3(B_REG);
end process;
R3: process begin
```

```
    wait until CLK'Event and CLK='1';
    C_REG<=C;
end process;
Proc4: process(C_REG) begin
    D<=OP4(C_REG);
    OUTPUT_PORT<=D;
end process;
```

以上风格的代码在生成流水线电路时，还需要综合命令的配合。现在大多数行为综合工具都采用这种这种方式实现电路的流水线化。Synopsys 的 BC 提供了一种不需要附加综合命令的描述方式，如下面的代码所示。这段代码描述了一个 FIR 滤波器 X[K]=AX[K−1]+BU，通过 transport 语句迫使 OUTPUT_PORT 接受两个时钟周期之前的运算结果，从而引导行为综合工具生成流水线电路。这种描述风格比较简洁，但只被 BC 接受，而且只能在周期固定调度模式下使用。

```
Piprline: loop
begin
    TMP := INPUT_PORT;
    X := X*A+TMP*B;
    OUTPUT_PORT<=transport X after 40 ns;   --设时钟周期为 20 ns,
                                            --即延时两周期
    wait until CLK'Event and CLK'='1';
end loop Pipeline;
```

对条件循环使用流水线式时，要注意的问题是在退出循环后，必须注意刷新流水线上的操作，否则会造成逻辑错误。这可以通过插入刷新周期来实现。在下面的 VHDL 代码中，如果某次循环结束后，COND 变为 false，这时应执行 loop 后的 I/O 写操作，然而流水线中尚有写操作没有完成，即仍然要输出结果，显然造成错误。因此必须在 end loop 语句执行插入三个 wait 语句，使流水线内的写操作全部结束。

```
Pipeline: while (COND='1') loop
begin
    TMP := INPUT_PORT;
    X := X*A+TMP*B;
    OUTPUT_PORT<=transport X after 40 ns;
    wait until CLK'Event and CLK'='1';
end loop Pipeline;
OUTPUT_PORT<=NEW_VALUE;      --错误,应插入 wait until 语句
```

7.4.8 握手协议

使用行为综合的方式设计电路时，由于可能采用不同的 I/O 调度模式，即 I/O 时序

可能浮动，所以应该采用一定的I/O通信协议，以避免出现不同模块之间通信的混乱。最常用的通信协议是硬件握手协议，即使用两根控制线，一根用于发送模块送往接收模块表示数据准备好，另一根用于接收模块送往发送模块表示数据接收完毕。图7.14是握手协议的原理图。

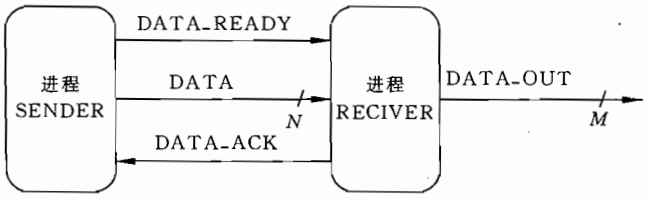


图7.14　握手协议原理图

下面给出了实现握手协议的行为级VHDL代码，其中DATA_READY为1表示数据准备好，DATA_ACK表示数据接收完毕。进程SENDER和RECIVER在时钟沿上检查这两个信号是否有效，以决定是否开始新的一次数据通信过程。

```
SENDER: process
begin
    wait until CLK'Event and CLK='1' and DATA_ACK='0';
    DATA<=Operation_Result;          --取得运算结果
    DATA_READY<='1';                 --数据准备好,可以发送
    wait until CLK'Event and CLK='1' and DATA_ACK='1';
    DATA_READY<='0';
end process SENDER;
RECIVER: process
begin
    wait until CLK'Event and CLK='1' and DATA_READY='1';
    DATA_OUT<=DATA;                  --接收数据
    DATA_ACK<='1';                   --接收完毕
    wait until CLK'Event and CLK='1' and DATA_READY='0';
    DATA_ACK<='0';
end process RECIVER;
```

在这一章中，主要讨论了行为综合的设计表示、行为综合的目标结构、行为综合流程以及行为综合的VHDL建模问题。表示一个设计主要有数据流图、控制流图、数据控制流图三种形式。行为综合的目标结构，主要有FSMD(带有数据路径的有限状态机)和FSMC(带有协处理器的FSMD)，以及相应的控制器、数据路径所包括的功能部件、存储单元和通信部件的模型等。行为综合流程则是包括对输入描述的语法分析、时序调度、硬件分配、网单生成等步骤的迭代过程。对行为综合的VHDL建模，主要讨论了行为级模型与寄存器传输级模型的区别和行为级建模的一些具体问题。这些问题对理解行为综合过程和算法都是有用的。

第8章 调度及分配

高层次综合或者算法综合的意义是把硬件的抽象行为描述自动转换为寄存器级模型。得到的寄存器级模型可以为寄存器级行为模型，也可以为寄存器级结构模型。对于给定的系统级的算法描述，可以转换为许多种不同寄存器级模型，自动综合程序应该在很大的设计空间中选择最好的设计方案。事实上，自动综合工具也不可能得出最优的设计方案。其原因在于：① 由于计算成本的原因，使得不可能对整个设计空间做穷举式搜索。② 实现硬件时可选用的器件种类会受到某些限制。③ 即使对于全定制集成电路设计，电路指标中也会给定最大延时、最大时钟频率、最大功耗等其他要求，这些要求可能互相矛盾。自动综合系统需要搜索足够大的设计空间，需要尽可能降低搜索设计空间的成本，搜索得到的设计方案必须满足约束条件。自动综合工具要花费最小的代价，实现尽可能好的设计。集成电路的自动综合是一个困难的、不成熟的，同时又是很有吸引力的研究领域。集成电路自动综合的中心技术为调度和分配，本章介绍综合过程中需要的调度和分配算法，介绍算法的特点和局限性。

8.1 算法综合的优点

集成电路的自动综合是一项困难的工作，是否有价值对这一问题进行研究？对这一问题进行研究的结果是否有用呢？利用算法综合至少可以得到如下收益：

(1) 缩短设计周期。通过使用自动综合工具，设计者可以只在高层次设计电路，而从高层次向低层次的转换，由自动综合工具完成。这可以加快设计速度，缩短设计周期，提高产品在市场上的竞争力。

(2) 降低设计成本。缩短设计周期自然可以降低设计成本。对于生产批量不大的专用集成电路，设计成本在总成本中占的比例很大，更应该使用自动综合工具。

(3) 降低成品成本。集成电路设计过程中，在设计的较高层次作出的决策对设计的复杂性和系统成本的影响远远大于低层次决策对它们的影响，因此应该在设计的较高层次进行优化设计。按目前的设计方法，如果进行这里所谓的优化需要硬件设计者本身对大量设计方案进行实验对比，需要硬件设计者对集成电路最低层次的工艺条件有足够多的了解。采用高层次综合工具，系统设计者可以不对工艺本身详细了解，而对与工艺相关的设计方案进行对比，得到的硬件设计方案应该是较好的方案。设计复杂性低，成本也会较低。

(4) 减少设计错误。自动化的设计过程消除了人工设计可能带来的设计错误。对于大系统设计，随设计复杂性的提高，人工设计出现错误的可能性急剧增加。采用经过验证的

自动综合工具，则可以消除大部分可能由人工设计带来的设计错误。

(5) 有可能在多个设计方案之间进行对比。采用传统的人工设计，每完成一个设计方案都要花费很大的代价，而采用自动综合技术可以迅速地得到多个满足指标要求的设计方案，因此可以对比不同的设计方案的优劣。

(6) 设计文档规则，容易维护。采用自动综合方法，可以保持设计全过程的所有文件，包括设计过程中每一决策的原因及效果。这些内容全以标准的数据结构组织，容易维护。

(7) 容易适应指标的部分改变。采用传统的人工设计方法，如果已经完成了部分设计，再对设计指标进行改动是一种代价很高的困难工作。采用自动综合工具，硬件的行为模型则相对容易修正，而从行为模型到寄存器级以下的层次，都是自动产生，因此较容易对修改后的性能指标重新开始设计。

图 8.1 给出了高层次算法综合的主要工作内容，是从高层次 VHDL 行为模型开始的综合过程。综合的第一步称为编译，即把 VHDL 描述转换为适合于自动综合的中间格式。第 7 章曾简单介绍过数据流图、控制流图等中间格式，本章首先介绍利用中间格式进行综合的调度算法，对下面 VHDL 源代码给出的模型进行综合。值得注意的是高层次综合处理的 VHDL 模型中总是包含整数、实数等高层次抽象数据类型。本章最后讨论分配算法、对调度分配算法进行评价的准则，以及迭代进行调度和分配的技术。

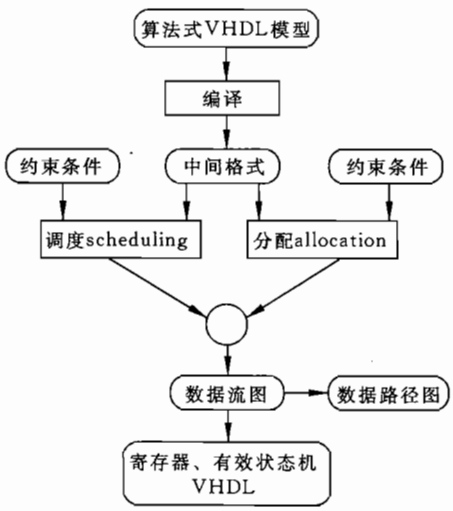


图 8.1 高层次算法综合的内容

编译包括变换及优化。变换指的是从 VHDL 描述变换为第 7 章中讨论的中间格式，优化指的是对某些逻辑表达式进行优化，目的是得到较好的硬件设计。编译包括面向软件的编译和面向硬件的编译两部分内容。面向软件的编译包括死码消除、公共表达式的识别、进程的展开、循环的展开等内容。面向硬件的编译包括整数到二进制位矢量的变换、乘 2 运算转换为左移操作等等。

8.2 调　　度

调度(scheduling)是自动综合中最为复杂的一项工作。其内容是把硬件行为分解为不同状态，把每个算子分配到指定状态内完成。在时序系统中，状态是最基本的时间单位，通常一个状态就是一个系统时钟周期。调度算法可以划分为两类，即基于数据流图的调度算法和基于控制流图的调度算法。对于基于数据流图调度算法，最基本的目标就是用尽可能少的状态完成数据操作，即尽可能快地完成数据操作。除了要减少状态个数之外，还要考虑可利用的运算单元的种类及个数等约束条件。基于数据流图的调度算法是最大并行的算法，由于调度算法的基础是数据流图，而数据流图依赖于运算之间的数据相关性，这

就决定了其并行性。基于数据流图的算法通常等效于二维优化问题，即：① 利用相同的运算资源，尽量减少完成电路功能所需要的状态个数；② 在给定的状态中实现电路功能，尽量减少使用的运算资源。

这种算法假定减少实现电路功能所需的状态个数就可以减少算法的总执行时间。当硬件的行为描述中带有分支结构时，如果每个分支以相同的概率执行，减少状态个数确实可以减少算法的执行时间。实际情况中并不一定满足这种假设，所以减少状态个数并不完全等效于减少算法的运行时间。

控制流图本身是电路行为描述的顺序表示。基于控制流图的调度算法将运算序列分配到一个状态中，同一操作可以分配到不同状态中。每一个运算序列只有一个入口和一个出口。由于 VHDL 支持 if…then…else，case 及 loop 等语法结构，而控制流图的分支节点就表示了这些条件结构，基于控制流图的调度算法适合处理这种结构。

8.3 基于数据流图的调度技术

应用最广泛的调度技术大多基于数据流图。基于数据流图的调度通常通过两种迭代方法解决二维优化问题，这两种迭代方法为：① 使用的运算资源数目固定，优化目标是使用最少的状态个数完成指定算法，通过迭代修改约束条件，以取得最优结果，这种方法称为约束资源的调度。② 状态个数固定，优化目标是运算资源数目最小，通过迭代修改约束条件，取得最优结果，这种调度技术称为约束时间的调度技术。

通常假定对运算单元的种类和个数没有任何约束，对完成硬件行为所需的状态个数也不做限制，这种条件下的调度称为无约束调度技术。为了理解调度的概念，我们先介绍无约束调度技术，然后再介绍约束资源和约束时间的调度技术，以及同时约束时间和资源的调度技术。

8.3.1 无约束调度技术

无约束调度技术不限制运算单元的种类和个数，依次选择数据运算安排到某个状态中完成。每次把数据运算安排到某个状态时作出如下两项决策：① 下一步应该安排哪项数据运算？② 应该把选定的数据运算安排到哪个状态中完成？

上述两项决策可以根据全局准则作出，也可以根据局部准则作出。如果采用局部准则，作出决策时供选择的项数少，计算量小，但得到的结果可能差一些；如果采用全局准则，可能会得到较好的调度结果，但每次选择一个算子或选定一个状态所需的计算量可能会较大。设计者通常要在设计质量和运算量之间进行折衷。无约束调度方法有许多种，其不同之处在于选择安排哪个算子的方法以及选择把算子安排在哪个状态中的方法不同。下面以实体 SYNEX1 为例，介绍几种有代表性无约束调度方法。图 8.2 给出了 SYNEX1 的结构体 HIGH_LEVEL 的数据流图。

```
entity SYNEX1 is
    port (A,B,C,D,E: in Integer;X,Y: out Integer);
end SYNEX1;
```

```
architecture HIGH_LEVEL of SYNEX1 is
begin
    X<=E*(A+B+C);
    Y<=(A+C)*(C+D);
end HIGH_LEVEL;
```

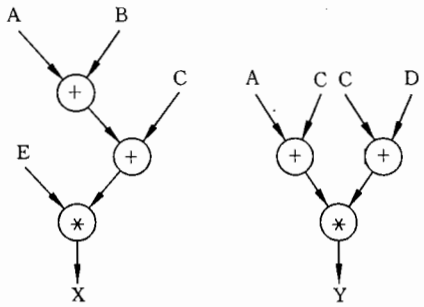


图 8.2　实体 SYNEX1 的结构体 HIGH_LEVEL 的数据流图

1. 无约束 ASAP(as soon as possible)算法

最简单的调度方法就是无约束 ASAP 方法。假定算子的总数已知，且已经构造好硬件的数据流图，则无约束 ASAP 法进行调度的原则是在每一个状态安排尽可能多的算子。安排算子时只考虑是否可以获得参加运算的数据，并假定总是可以获得所需要的运算单元。这就是说，尽可能早地安排每一个算子。按这种原则对上例 SYNEX1 进行调度可以得到图 8.3 所示的结果。通常 ASAP 原则可以得到最快的运算速度，但采用 ASAP 原则调度需要的运算单元个数会很多。例如，在图 8.3 的状态 S1 中需要三个加法器。

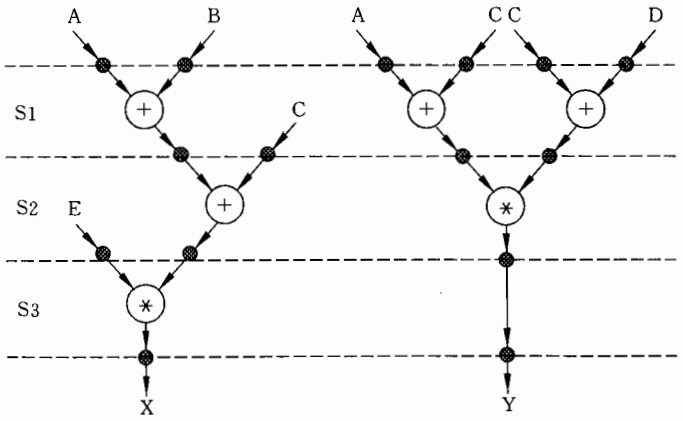


图 8.3　无约束调度的 ASAP 原则：在最早的状态内完成运算

2. 无约束 ALAP(as later as possible)调度

ALAP 调度方法从数据操作的最后输出开始调度。从完成全部数据操作的最后一

个状态开始调度，把每个算子尽可能安排在最后一个状态中。也就是说首先把产生输出信号所需的算子安排在最后一个状态；然后把完成最后状态中的运算所需的输入数据安排在次最后状态中，即次最后状态的运算结果是最后状态中所需的输入数据。如此等等，一直到将所有算子全安排完为止。由于在开始调度时并不知道总共需要多少个状态，因此，ALAP 调度方法是在调度结束后再对状态进行编号。图 8.4 给出了采用 ALAP 方法调度得到的结果。使用了两个乘法器、三个加法器，完成全部数据运算需要三个状态。

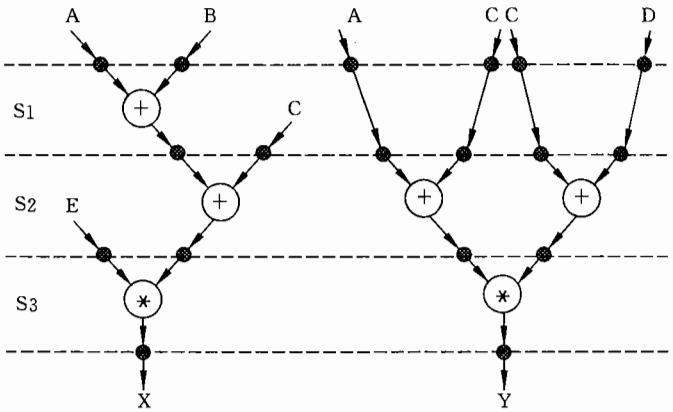


图 8.4　无约束调度的 ALAP 原则：在最晚的状态内完成运算

8.3.2　约束资源的调度技术

无约束调度技术解释了调度的基本问题，但在实际应用中，调度需要满足各种不同的约束。通常要限制芯片面积，即限制所使用的功能模块的个数。前面的 ASAP 调度方案需要 3 个加法器和 1 个乘法器，ALAP 调度方案则需要 3 个加法器和 2 个乘法器。如果限制芯片面积，限制只能使用 1 个加法器和 1 个乘法器，则这两种调度方案都不能满足要求。下面介绍几种约束运算资源的调度方案。

1. 状态撕裂法

状态撕裂法是一种方法变换法，从一个最容易得到的调度方案入手，采用规则的方法依次对方案进行变换，直到得出满足所有约束条件的调度方案为止。比如，可以从 ASAP 法的调度结果起步。对前面讨论过的图 8.3 所示的调度结果，可以依次进行变换以得到满足约束条件的调度结果。状态撕裂法对违反硬件约束条件或时序约束条件的每个状态进一步划分为多个子状态，直到所有状态下全满足约束条件为止。对于图 8.3 的调度结果，如果限制只能使用一个加法器和一个乘法器，由于状态 S1 中使用了三个加法器，所以违反了硬件约束条件，将它分解为三个状态，可以得到满足约束条件的调度方案。事实上，分解一个违反约束条件的状态可以采用多种方法，人们至今没有找到最优的状态撕裂方法。图 8.5 给出一种可能的撕裂方法，调度的结果用了 6 个状态。

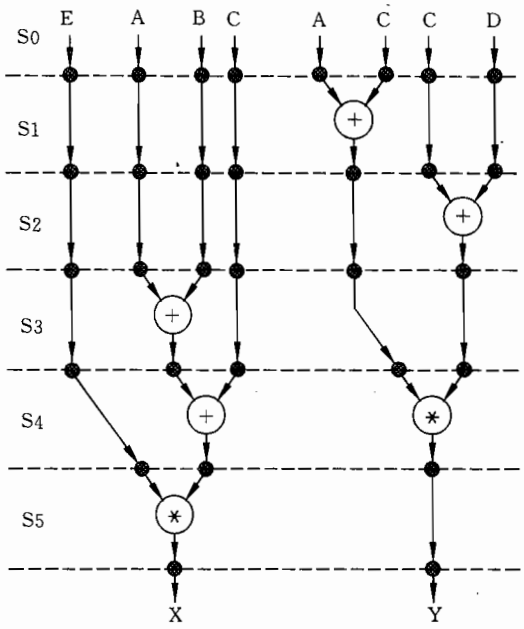


图 8.5　对 ASAP 调度结果中状态 S1 的分解

2. 约束资源的 ASAP 调度技术

ASAP 方法也可以用来考虑资源约束条件。ASAP 法进行调度的原则是在每一个状态安排尽可能多的算子。假如安排算子时不但考虑是否可以获得参加运算的数据，而且考虑是否可以得到所需要的运算单元，就构成了约束资源的 ASAP 调度算法。按这种方法，状态的选择也是从先到后，如果某个算子所需的数据已经准备好，且这种运算所需的运算单元也可以得到，则把该算子安排在当前状态内，下一个状态选择为当前状态的后续状态，在新的状态内还是安排尽可能多的算子，如此一直到所有算子全安排完。一旦安排完一个状态内要完成的数据运算，再也不返回到这个状态。

为了说明约束运算资源 ASAP 法的调度过程，仍以图 8.2 所示的数据流图为例进行讨论，为了引用方便我们把该数据流图重新画出，并对各个算子随机编号，如图 8.6 所示。这里对算子的编号只为讨论所用，编号顺序对算法无影响。假定对运算资源的约束为：最多可以使用两个加法器和两个乘法器。

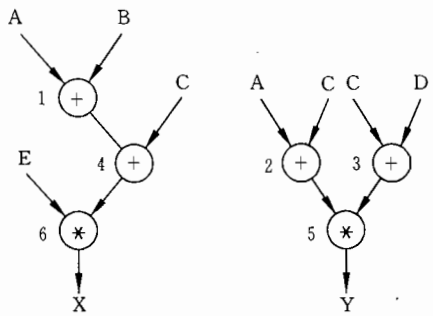


图 8.6　节点编号后的数据流图

在状态 S1，只有算子 1，2 和 3 的输入数据已经准备好，即可安排在状态 S1 中完成的算子为 1，2 和 3。由于最大可能使用两个加法器，所以这时可以在这三个算子中任意选择两个安排在状态 S1 中。可以选择算子 2 和 3 安排在状态 S1 中。由于现在只根据局部准则安排状态 S1 中的算子，所以并不能判断把三个算子中的哪两个安排在状态 S1 中会有利于后面的调度。显然安排在状态 S1 中完成的算子时总共有三种可能性，可以用穷举法对各种安排方案进行对比。

在状态 S1 安排了算子 2 和 3 之后，下一个状态应该是 S2。由于在状态 S1 已经安排了算子 2 和 3，此时可以使用算子 2 和 3 产生的输出结果，所以在当前状态，即状态 S2 只可能安排算子 1 和 5。在状态 S3，唯一可能安排的算子是 4，将算子 4 安排到状态 S3 之后，只留下算子 6 没有安排，把它安排在 S4 就完成了调度过程。

图 8.7 给出了在限制使用两个加法器、两个乘法器的约束条件下，采用 ASAP 算法得到的调度结果。从图 8.7 可以看出，这样的调度结果只使用了两个乘法器之间的一个。这提示我们，还可能存在其他更有效的调度方案。但是，一般地讲，有效的调度方案并不一定要使用所有可能使用的器件。

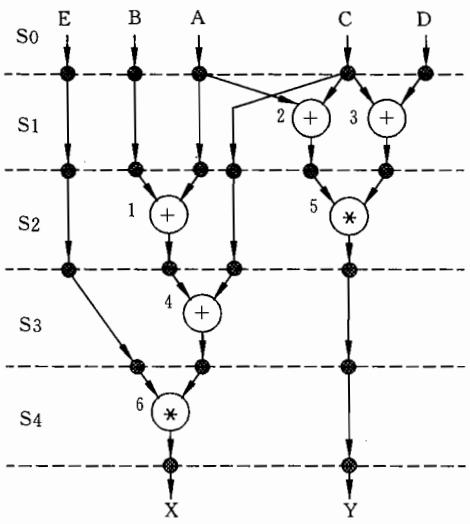


图 8.7 限制使用两个加法器、两个乘法器时 ASAP 方法调度结果

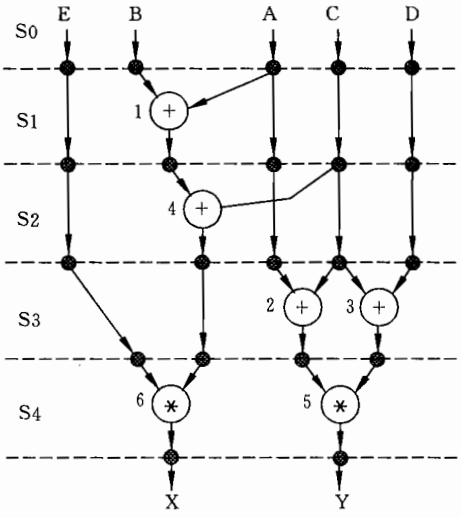


图 8.8 限制使用两个加法器、两个乘法器时 ALAP 方法的调度结果

3. 约束资源的 ALAP 调度

约束运算资源 ALAP 调度方法从产生最后输出算子开始调度。与无约束 ASAP 算法不同之处在于安排每个算子保证不违反资源约束条件，即在不违反运算资源约束条件的前提下，把产生输出信号所需的算子安排在最后一个状态。然后把产生最后状态的输入数据所需的算子安排在次最后状态中，即在次最后状态的数据运算产生最后状态中所需的输入数据。如此等等，一直到将所有算子全安排完为止。

仍以图 8.6 的数据流图为例，资源约束仍为“最多使用两个加法器和两个乘法器”。产生最后输出的两个算子是乘法运算 5 和 6，可以把这两个算子安排在同一个状态内完成。

为了执行算子 5，需要得到算子 2 和 3 的输出，为了执行算子 6 需要得到算子 4 的运算结果，因此，在次最后状态应该安排算子 2，3 和 4。由于算子 2，3 和 4 全是加法运算，而运算资源约束条件为只能使用两个加法器，可以在这三个算子中任选两个，将算子 2 和 3 安排在次最后一个状态中。这时留下算子 1 和 4 没有安排，因为执行算子 4 要用到算子 1 的输出，所以算子 4 和 1 只能安排在两个不同的状态内。图 8.8 给出了调度结果。

在约束运算资源的 ALAP 算法中，由于调度过程中选择将哪个算子安排到哪个状态中时只采用局部准则，有可能得出的结果并非最优结果。例如，在上述调度过程的次最后状态中可以安排算子 2，3 和 4，如果选择安排加法运算 2 和 4，则可以得到图 8.9 所示的调度方案。显然图 8.9 的调度结果比图 8.8 少用了一个状态。

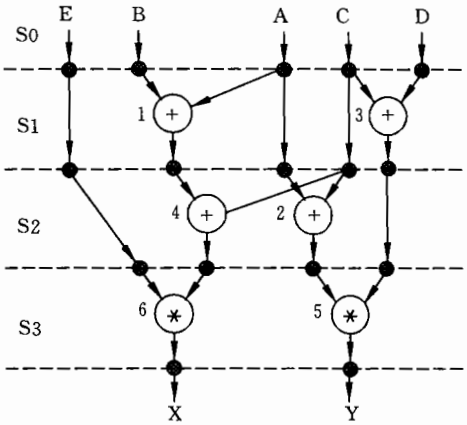


图 8.9　限制使用两个加法器、两个乘法器时 ALAP 方法的另一调度结果

4. 排队调度技术

无论 ASAP 方法还是 ALAP 方法都没有考虑关键路径问题，都没有考虑不同路径上的数据运算应该有不同的加权值。按这种方法进行调度，有可能把有限的硬件资源浪费在不重要的数据运算上。这样就可能造成在状态序列的最后几个状态中，大部分器件处于休止状态，以等待关键路径上的数据运算，造成状态个数的增加。如果选择每个要安排的算子时不只根据局部原则，而且考虑全局原则，可以解决这一问题。

数据流图中最长的数据通路称为关键路径。例如，对于图 8.6 所示的数据流图，沿 1-4-6 的数据通路的长度为 3，是最长的数据通路，是该问题的关键路径。直观看来，如果关键路径上的数据运算被延时，一定会增加系统完成整个数据运算的时间，因此关键路径上的算子应该尽早安排到某些状态中完成，即关键路径上算子的优先级应该比非关键路径上算子的高。应该知道，当某些算子被安排好之后，关键路径可能变得与原始关键路径不同，即关键路径是动态的概念。

在调度过程中选择当前关键路径的方法有许多种，下面介绍一种简单算法。对于数据流图中的每一个节点，计算从此节点到数据流图底部的最长通路的长度，记此长度为该节点的“关键路径优先级”。对于图 8.6 的数据流图，确定各个节点的“关键路径优先级”之

后，把各节点的“关键路径优先级”用括号括起后标在数据流图中，可以得到如图 8.10 所示的数据流图。

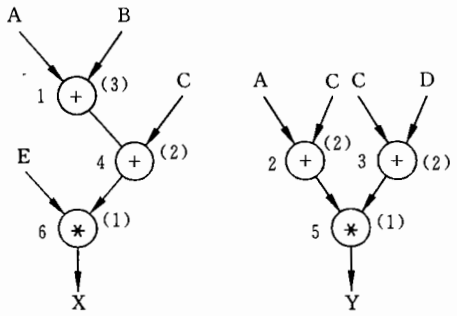


图 8.10　确定节点“关键路径优先级”后的数据流图

根据每个节点的“关键路径优先级”，可以对每个节点按降序排队。对于图 8.10 的数据流图，可以对各个节点排队，如表 8.1。

表 8.1　对图 8.10 各节点排队

关键路径优先级	(3)	(2)	(1)
算子(节点)	1	4,2,3	6,5

在调度过程中，可以根据表 8.1 对各节点优先级的排队来选择先把哪个算子安排在当前状态中。

对于图 8.10 的数据流图，采用 ASAP 排队法进行调度，在状态 S1，算子 1,2 和 3 的输入数据已经完备，即可以将算子 1,2 或 3 安排在状态 S1 中完成。这三个算子全是加运算，由于约束条件限制只能使用两个加法器，所以只能把这三个算子中的两个安排在状态 S1 中完成。由于算子 1 的优先级最高，按排队原则应该把算子 1 安排在状态 S1 中。由于算子 2 和 3 的优先级相同，按排队原则可以在算子 2 和 3 之间任选一个安排在状态 S1。如果选择将算子 2 安排在状态 S1，那么在状态 S2，只有算子 3 和 4 的输入数据是完备的，这两个算子全是加法运算，而约束条件规定可以使用两个加法器，因此自然应该把它们都安排在状态 S2；而在状态 S3，余下的两个算子为 5 和 6，它们的输入数据都已完备，两个算子全为乘法运算，设计约束规定可以使用两个乘法器，因此可以把两个算子全安排在状态 S3。这样就完成了 ASAP 排队方法的调度过程，调度得到的结果如图 8.11 所示。排队调度方法在选择算子时采用了全局准则，而选择当前调度状态时采用了局部准则，即取当前状态的下一状态作为下一次调度的状态。

5. 自由度法

自由度法选择算子和选择把算子项安排在哪个状态全采用全局准则。自由度调度方法的基本考虑是先安排自由度较小的算子，这样做的原因是自由度小的算子对整个调度过程的影响可能会比较大；而自由度较大的算子可以在调度的较后阶段安排，因为它们可能比较容易安排在许多不同的状态中。

自由度法调度的第一步是先根据数据流图进行无约束 ASAP 及无约束 ALAP 调度，分别确定每一个算子可能安排在哪些状态中，以确定出各个算子可能安排到的状态范围。

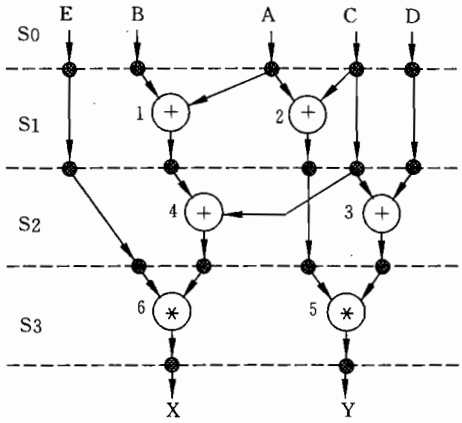


图 8.11　限制使用两个加法器、两个乘法器时
ASAP 排队法的调度结果

例如对于图 8.6 的数据流图，每个算子可以安排的状态范围如表 8.2。

表 8.2　对图 8.6 每个算子可以安排的状态范围

算　子	最早可能安排到状态 （无约束 ASAP 法）	最晚可能安排到状态 （无约束 ALAP 法）	范　围
1	S1	S1	1
2	S1	S2	2
3	S1	S2	2
4	S2	S2	1
5	S2	S3	2
6	S3	S3	1

根据表 8.2 可以知道，自由度最小的算子是 1，4 和 6。在状态 S1，由于只有算子 1，2 和 3 的输入数据完备，同时由于这三个算子全为加法运算，所以只可能安排这三项中的两项。由于算子 1 的自由度低，所以应该首先安排算子 1。由于算子 2 和 3 的自由度相同，可以任选两项中的一项安排在状态 S1。可以选择算子 3 安排在状态 S1。在状态 S2，由于只有算子 2 和 4 的输入数据完备，调度约束允许使用两个加法器，可以把这两个加法运算全安排在状态 S2。在状态 S3，只留下了算子 5 和 6，约束条件中规定可以使用两个乘法器，所以可以把它们全安排在状态 S3。图 8.12 给出了自由度法的调度结果。

8.3.3　约束时间的调度技术

除了约束芯片面积之外，集成电路设计中通常会对运算速度提出要求，这就需要采用约束时间的调度技术。其典型应用是数字信号处理芯片的设计，它们通常要满足抽样率的要求，即对输入数据的码率有一定的要求。在调度过程中，通过对状态数目的限制，可以实现对运算时间的约束。对时间的约束分为两类：① 限制全部数据必须在指定的状态数目

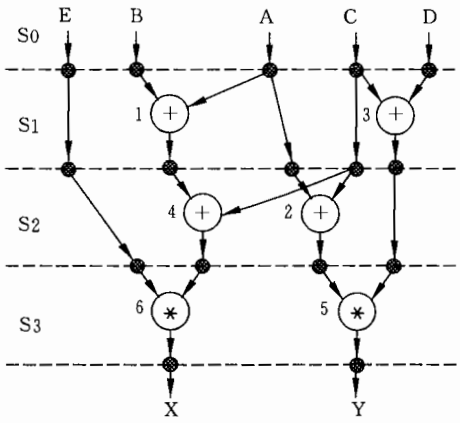


图 8.12　限制使用两个加法器、两个乘法器时自由度法的调度结果

中完成；② 限制某个算子必须在指定的状态中完成。有时会同时存在这两种约束。例如，对于图 8.13 所示的数据流图，可以限制必须在三个状态中完成全部运算，且限制算子 1 和 4 必须在第一个状态中完成，算子 2 必须在第二个状态中完成，算子 3 必须在第三个状态完成。作出这些限制的原因通常是由于这些算子位于一定的关键路径上，无法把它们安排到其他状态。

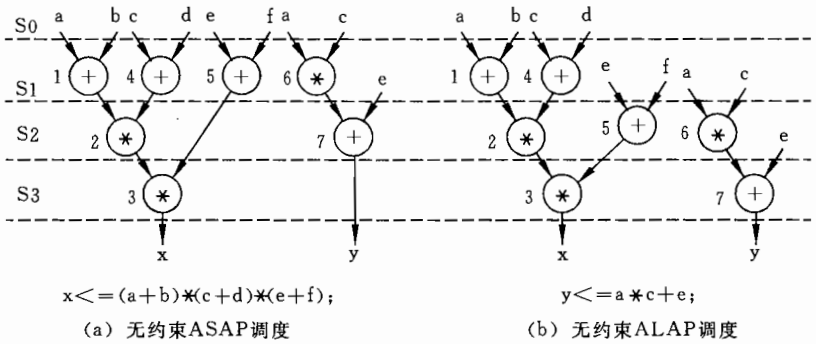


图 8.13　数据流图、无约束 ASAP 调度及 ALAP 调度

约束时间的调度技术中最常用的是 FDS(force directed scheduling)法。这种方法的优化目标是将同一类算子尽可能均匀地分布到各个状态中。均匀分布的结果是器件利用率尽可能地得以提高。与自由度法类似，FDS 调度技术首先要利用无约束 ASAP 调度及 ALAP 调度结果确定各个算子的调度自由度。例如，算子 5 的可能安排的状态区间为[1,2]，这里 1 表示算子 5 最早可以安排到状态 1(ASAP 法)，最晚可以安排到状态 2(ALAP 法)。FDS 法假定每一个算子在其可能安排的状态区间内均匀分布，因此，算子 i 被安排到状态 Sj 的概率为

$$P_{i,j} = 1/(L_i - E_i + 1), \qquad E_i \leqslant j \leqslant L_i$$

其中：E_i 表示算子 i 最早可能安排的状态(ASAP 法)；L_i 表示算子 i 最晚可能安排的状态(ALAP 法)；$P_{i,j}$ 表示算子 i 被安排到状态 Sj 的概率。对于图 8.13 的数据流图，图

8.14(a)标出了各个算子安排到相应状态的概率。图中对应每个算子的方框宽度表示了该算子安排到对应状态的概率。例如，算子 5 被安排到状态 1 和状态 2 的概率全为 0.5。从图 8.14(a)可以看出，算子 1,2,3 和 4 安排到其相应的状态的概率为 1，从而知道算子 1，2,3 和 4 为整个数据运算的关键路径。对于第 k 类算子(比如乘法运算为第 1 类算子)或者说第 k 类数据运算，可以建立在指定状态 j 安排该类算子的分布图，表示在指定状态 j 安排第 k 类算子的成本的数学期望值(平均成本)，其定义为

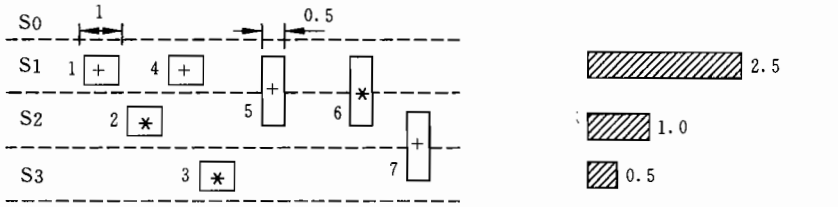
$$FCost_{k,j} = c_k \sum_{i,j \in range(i)} P_{i,j} \tag{8.1}$$

其中 c_k 为第 k 类算子的成本，算子 i 为某个第 k 类算子，其可能安排的状态范围为 $range(i)$。假定每个加法器的成本为 1，图 8.14(b)画出了加法器安排到 3 个状态的分布图，它们分别根据下列 3 式算出：

$$FCost_{add,1} = P_{1,1} + P_{4,1} + P_{5,1} = 1 + 1 + 0.5 = 2.5 \tag{8.2}$$

$$FCost_{add,2} = P_{5,2} + P_{7,2} = 0.5 + 0.5 = 1 \tag{8.3}$$

$$FCost_{add,3} = P_{7,3} = 0.5 \tag{8.4}$$



(a) 将各数据项安排到指定状态的概率　　(b) 加法器安排到3个状态的分布图

图 8.14　各数据项安排到指定状态的概率及加法器安排到三个状态的分布图

由于在不同的状态中可以共享同一个数据处理单元，在全部状态中需要的第 k 类数据运算单元的个数可以确定如下：

$$Cost_k = \max_j \{FCost_{k,j}\} \tag{8.5}$$

FDS 算法的主要目标就是在各个状态之间有效地共享数据处理单元，实现共享处理单元的方法之一就是对每类数据运算，平衡 $FCost$ 的数值。比如对于算子 5 的安排，有两种可能性，即算子 5 可以安排在状态 1，也可以安排在状态 2。如果安排在状态 1，则完成系统功能需要 3 个加法器；如果安排在状态 2，则只需要 2 个加法器。图 8.15 画出了将算子 5 安排到状态 2 时在三个状态内安排加法器的分布图。

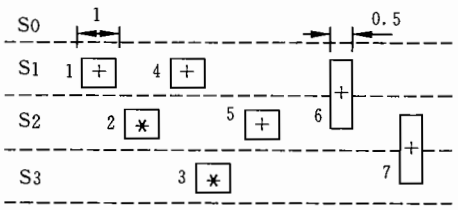
FDS 算法由如下几步构成：

(1) 建立每类数据运算安排到各个状态的分布图。即按照式(8.1)对每类数据运算，计算其安排到各个状态的平均成本。

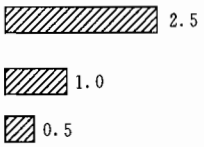
(2) 将每个算子依次安排到其可能安排的状态区间内的各个状态，并按照下式计算 $DForce_{i,k,j}$：

$$DForce_{i,k,j} = FCost_{k,j} - \sum_{s \in range(i)} \frac{FCost_{k,s}}{L_i - E_i + 1} \tag{8.6}$$

这里的 $DForce_{i,k,j}$ 是第 k 类数据运算的算子 i 安排到状态 j 的调度成本与第 k 类数



(a) 将数据操作项安排到指定状态的概率



(b) 数据操作项5安排到状态2时加法运算在3个状态内安排的分布图

图 8.15 算子 5 安排到状态 2 时，各算子安排到指定状态的概率和加法器安排到三个状态的分布图

据运算安排到状态 j 的平均成本之差。

(3) 如果将某个算子 i 安排到某个状态 j 的 $DForce$ 数值最小，则应该将该算子安排到状态 j。

例如，对于图 8.13 的数据将算子 5 安排到状态 1 的 $DForce$ 为

$$DForce_{5,\mathrm{add},1} = 2.5 - (2.5 + 1)/2 = 0.75$$

算子 5 安排到状态 2 的 $DForce$ 为

$$DForce_{5,\mathrm{add},2} = 1 - (2.5 + 1)/2 = -0.75$$

显然，将算子 5 安排到状态 2 是一个较好的方案。

8.3.4 同时约束时间和资源的调度技术

对于同时存在时间和资源约束的调度问题，最常用的解决方法是线性整数规划法，这种方法能够保证得到最优调度。但是，最优调度的获得通常以牺牲运算时间为代价。线性整数规划法将调度问题表达为一组满足不等式约束的线性目标函数。令 FU_k 为可以得到的完成第 k 类数据运算的功能单元集合(比如可以得到的乘法器集合)，C_k 为完成第 k 类数据运算的功能单元的成本(比如乘法器成本)，N_k 为完成第 k 类数据运算的功能单元的个数(比如可以得到的乘法器个数)；令 S 为状态集合；x_{ij} 为决策变量，如果算子 i 被安排到状态 j，x_{ij} 取值为 1，否则 x_{ij} 取值为 0。这样线性整数规划问题的约束条件如下：

(1) 保证每个算子只能被安排到一个状态中，亦即

$$\sum_{j \in range(i)} x_{i,j} = 1 \tag{8.7}$$

(2) 保证在每个状态中，第 k 类数据运算单元的个数不超过 N_k，即有

$$\sum_{i \in type(k)} x_{i,j} \leqslant N_k, \quad \forall\, k, j \tag{8.8}$$

(3) 保证顺序约束。对于算子 j，得到其所有的输入变量必须被安排到比它早的状态中，即

$$\sum_{s \in range(i)} (s \times x_{i,s}) - \sum_{t \in range(j)} (t \times x_{i,t}) \leqslant -1 \tag{8.9}$$

其中算子 i 是算子 j 以前的算子。

对于图 8.13 的数据流图进行调度，假定加法器和乘法器的成本全为 1，并假定时间

约束为“限制全部数据操作在 3 个状态中完成”，资源约束为“只能使用 2 个加法器和 1 个乘法器”，则调度问题的约束函数如下所示。

算子安排约束：

$$\left.\begin{aligned} x_{1,1} &= 1 \\ x_{2,2} &= 1 \\ x_{3,3} &= 1 \\ x_{4,4} &= 1 \\ x_{5,1} + x_{5,2} &= 1 \\ x_{6,1} + x_{6,2} &= 1 \\ x_{7,1} + x_{7,2} &= 1 \end{aligned}\right\} \tag{8.10}$$

资源约束条件：

$$\left.\begin{aligned} x_{1,1} + x_{4,1} + x_{5,1} &\leqslant N_a = 2 \\ x_{5,2} + x_{7,2} &\leqslant N_a = 2 \\ x_{7,3} &\leqslant N_a = 2 \\ x_{6,1} &\leqslant N_m = 1 \\ x_{2,2} + x_{6,2} &\leqslant N_m = 1 \\ x_{3,3} &\leqslant N_m = 1 \end{aligned}\right\} \tag{8.11}$$

调度过程约束：

$$1 \cdot x_{6,1} + 2 \cdot x_{6,2} - 2 \cdot x_{7,2} - 3 \cdot x_{7,3} \leqslant -1 \tag{8.12}$$

满足上述约束条件的解为

$$\left.\begin{aligned} x_{1,1} &= 1 \\ x_{2,2} &= 1 \\ x_{3,3} &= 1 \\ x_{4,1} &= 1 \\ x_{5,2} &= 1 \\ x_{6,1} &= 1 \\ x_{7,2} &= 1 \end{aligned}\right\} \tag{8.13}$$

为了完整地表述整数线性规划问题，可以将整数线性规划问题的目标函数定义为“使运算单元的总成本最小”，这种目标函数如下式所示：

$$\min \sum_{k} C_k \cdot N_k \tag{8.14}$$

也可以定义目标函数为“减少实现系统算法的状态个数”，这种目标函数如下式所示：

$$\min \sum_{s \in range(d)} s \cdot x_{d,s} \tag{8.15}$$

理论上讲，整数线性规划一定存在最优解，也有求解整数线性规划问题的数学方法。但当不等式的个数较大时，求解不等式所需的时间成指数增长趋势，因而不能采用这种方法。调度规模较大的实际设计问题，通常要采用简化算法。

8.4 基于控制流图的调度技术

基于数据流图的调度技术以数据流图作为设计的内部表述，适用于对无限数据流重复进行一系列运算的系统进行综合时的调度，这种情况下某些运算可以并行完成。在以完成控制为基本功能的系统中，控制序列与外部条件有关，系统的算法描述主要包含控制结构，只有少数运算的需求。对于这种系统的综合，不应采用基于数据流图的调度方法，而应采用基于控制流图的调度技术，以控制流图作为设计的内部表述。

在硬件高层次源代码中，条件分支语句对应着不同的执行路径。完成不同代码分支中的运算所需要的状态个数可能不同，但对同一个分支来讲，完成其运算所需的状态则是固定的。采用基于数据流图的调度方法，本质上是假定完成所有代码分支所需的状态个数相同，即为各分支中最大的状态个数。

8.4.1 调度路径

基于控制流图的调度方法基本上是序贯法。需要根据控制流图定义调度路径，即将控制流图划分为调度路径序列，根据调度结果构造有限状态机。下面的 VHDL 源代码完成 AB mod N 的功能。给定，$0 \leqslant A, B \leqslant N$，$1gN \leqslant 15$，图 8.16 是其控制流图。

```
entity MODN is
    port(start: in Std_logic;
         Ai : in Integer;
         Bi : in Integer;
         Ni : in Integer;
         Si : out Integer);
end MODN;

architecture behavior of MODN is
begin
  process
      variable A : Integer;
      variable B : Integer;
      variable N : Integer;
      variable S : Integer;
      variable I : Integer;
  begin
      A := Ai; B := Bi; N := Ni;          -- 1
      S := 0; I := 0;                     -- 2
```

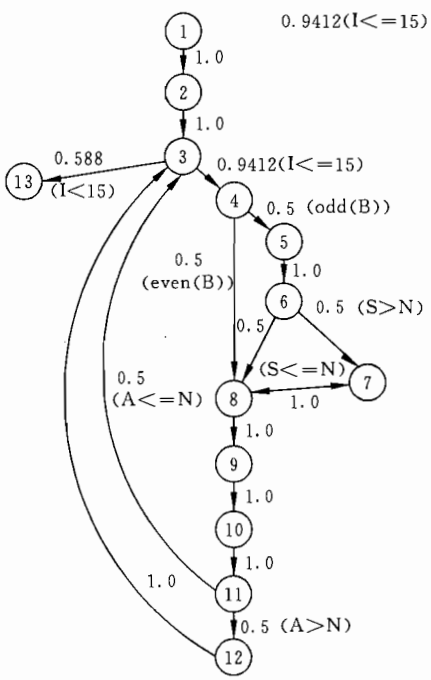


图 8.16 完成运算 AB mod N 的控制流图

```
        while ( I<=15) loop                    -- 3
            if(odd(B))                         -- 4
                then S := S + A;               -- 5
                if (S > N)                     -- 6
                    then S := S - N;           -- 7
                end if;
            end if;
            I := I + 1;                        -- 8
            B := B div 2;                      -- 9
            A := A * 2;                        -- 10
            if (A > N)                         -- 11
                then A := A - N;               -- 12
            end if;
        end loop;
        SI<=S;                                 -- 13
    end process;
end behavior;
```

按照图论中的习惯，控制流图可以用边和节点的集合来表述，即 $G=(V,E)$。其中 G 表示一个控制流图，V 是图的节点集合，E 是控制流图的边的集合。令 $\{C_1,C_2,\cdots,C_r\}$ 为设计约束的集合，则调度路径 SP 为从控制流图 G 中提取出的满足全部约束条件的节点集合 $\{v_1,v_2,\cdots,v_n\}$。这里约束条件指的是：如果对于约束集合 $\{C_1,C_2,\cdots,C_r\}$，某个 $C_k(v_i,v_j)=1$ $(k=1,\cdots,r)$，则说约束 C_k 在节点 v_i 和 v_j 之间不满足约束条件。

调度路径具有如下两个特性：

(1) 对所有 $v_i,v_j\in SP$，有 $C_k(v_i,v_j)=0$ $(k=1,\cdots,r)$。

对于每个调度路径 SP，可以定义三个参数如下：

① 调度路径 SP 的第一个节点，定义为该调度路径的起始点 $Head(SP)$。

② 调度路径 SP 中的最后一个节点的下一个节点定义为该调度路径的后续节点 $Succ(SP)$。

③ 在满足某个条件时，调度路径 SP 会被执行，则称这个条件为该调度路径的执行条件，记作 $Cond(SP)$。控制路径的执行条件 $Cond(SP)$ 由从 $Head(SP)$ 到 $Succ(SP)$ 的全部边表示的条件做与运算得到。

(2) 令 $SP=(v_i,\cdots,v_j)$ 为一个调度路径，则该调度路径执行的条件为

$$Cond(SP)=\left\{\bigcap_{i=1,\cdots,n-1,j=i+1} Cond(v_i,v_j)\right\}\cap Cond(v_n,Succ(SP)) \tag{8.16}$$

对于一个控制流图，将其全部调度路径的起始节点定义为该控制流图的起始节点集，用 $H(G)$ 表示，即有

$$H(G)=\{v_i\in V\,|\,SP\subset V,v_i=Head(SP)\} \tag{8.17}$$

调度的结果是一个有限状态机，可以为 Moore 型状态机，也可以为 Mealy 型。基于控

制流图的调度的目标是将全部具有相同起始节点的调度路径安排到同一个状态，假定节点 v_i 和 v_j 分别对应于两个状态 Si 和 Sj。只有在一个调度路径的起始节点 $Head(SP)$为 v_i，且后续节点 $Succ(SP)$为 v_j 的情况下，状态 Si 和 Sj 之间才有通路。

8.4.2 成本函数

在控制流图中，与数据相关的循环对调度结果平均影响很大，原因在于不知道它们执行多少次迭代才能完成循环过程，调度过程产生的状态个数不反映算法实际的执行时间。为了评价调度结果，需要定义一个测度，该测度应该包含调度结果执行时所需要的时钟周期数。

在控制流图描述的算法中，对于不同输入进行大量模拟，可以估算出控制流图中每条路径执行的概率。对应一条路径的执行概率称之为路径执行概率。令 $p(v_i,v_j)$表示执行节点 v_i 和 v_j 之间的路径执行概率，$P=(v_1,v_2,\cdots,v_n)$为控制流图中的一条路径，$v_m=Succ(P)$，则这条路径被执行的概率为：

$$Prob(P) = p(v_1,v_2)\cdot p(v_2,v_3)\cdot\cdots\cdot p(v_{n-1},v_n)\cdot p(v_n,v_m) \tag{8.18}$$

假定$(P_i,\cdots,P_j)$为状态 Sk 到 Sl 之间的路径集合，即集合中每一条路径全以 Sk 为起始状态，以 Sl 为终止状态，则从状态 Sk 到状态 Sl 的转换概率为

$$p_S(k,l) = Prob(P_i) + Prob(P_{i+1}) + \cdots + Prob(P_j) \tag{8.19}$$

令 S=(S$_1$,…,Sn)为调度过程所得到的状态集合，X_i 表示在整个行为描述的算法执行过程中，状态 Si 执行的次数的数学期望值，则执行完整算法所需的时钟周期个数的数学期望值为

$$X_{sch} = \sum_{i=1}^{n} X_i \tag{8.20}$$

计算 $X_i,\forall\, i\in(1,\cdots,n)$等效于求解下述线性方程组

$$\left.\begin{aligned} X_1 &= 1 \\ X_i &= \sum(X_j\cdot p_S(j,i)) \end{aligned}\right\} \tag{8.21}$$

要求对任意的 j，存在从状态 Sj 到状态 Si 的路径 P。

对于控制为主的行为描述，即行为描述中只包含少量的算术运算，但包含大量的控制操作，控制序列是其基本的外部条件，则不适宜于采用以数据流图为基础的调度算法。这时需要基于路径的调度算法，基于路径的调度算法需要提取路径，这就需要控制流图。

8.4.3 AFAP 调度

AFAP(as fast as possible)方法首先对全部可能路径独立调度，然后设法减小路径长度。调度技术的出发点是在满足约束条件的前提下尽量减少状态个数。在控制流图中，首先需要拆断反馈边以去除图中的环路。例如在图 8.16 中，边(v_{11},v_3)应该被拆断。对除去环路的控制流图进行深度优先的搜索可以得到路径。搜索的起始节点是去除环路控制流图的第一个节点和拆断的各个环路的第一个节点(环路总是重复执行的一系列操作)。搜索得到全部路径之后，需要对每个路径计算约束条件。为了讨论简单，下面只将数据依赖性作为约束条件，即只考虑是否可以获得参与处理的数据。约束条件由区间表示，区间由

一些数据操作序列组成，新的状态只能在某个数据操作处开始。例如，由于节点v_5和节点v_7之间有数据的依赖性，(v_6,v_7)是一个区间。在节点v_6或者节点v_7必须开始一个新的状态。这些约束如图 8.17 所示。对每一条路径，可以建立表示约束条件的区间图，区间图中的每个节点相应于一个区间，两个节点中的一条边表示两个区间有交迭关系。对应区间图，可以计算其最小(节点)团覆盖，即找出区间图中的最小个数的连通子图(连通子图中的节点集为一个团)，使得一个节点只出现在一个连通子图中。最小团覆盖中的每个团相应于必须开始新的状态的数据操作集合，状态将按照路径排序。在图 8.17 中，由 3 个团覆盖了整个图，这意味着执行整个路径需要 4 个状态，其中 3 个状态对应于 3 个团，1 个状态对应于起始节点v_1。对全部路径全做调度，就可以得出最少需要多少个状态才能完成全部数据操作。不同路径的划分可能会造成公共节点，这个问题可以通过约束条件表述和解决。对于一个路径，在满足约束条件的情况下可以将它划分为几个调度路径。例如图 8.17 中的路径(1,2,3,4,5,6,7,8,9,10,11,12)可以划分为$SP_1=(1,2)$，$SP_2=(3,4,5)$，$SP_3=(6,7,8,9,10)$，$SP_4=(11,12)$。对于图 8.16 的控制流图，将全部路径找出，并在满足约束条件的前提下，划分为调度路径，见图 8.18(a)。对于每个调度路径，图中标出了起始节点和后续节点。前面讲过，调度的结果为有限状态机，不同起始节点的个数对应于有限状态机的状态个数。如果节点v_i是某个调度路径的起始节点，v_j是该调度路径的后续节点，则从节点v_i所在的状态到节点v_j所在的状态有一条转换路径。采用 AFAP 法调度所得到的有限状态机如图 8.18(b)所示。

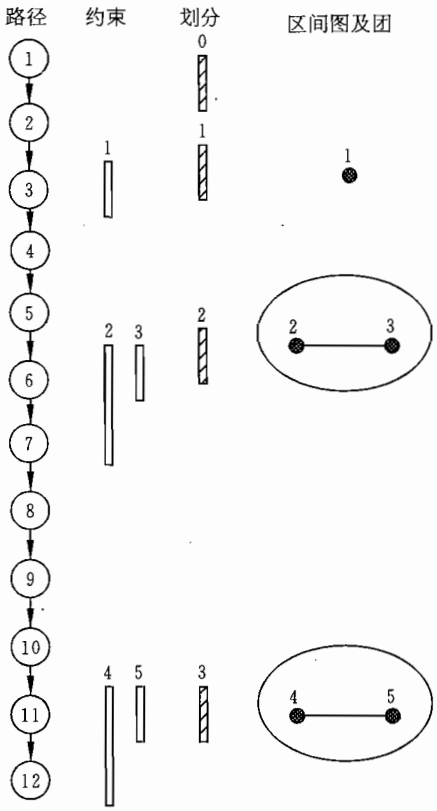
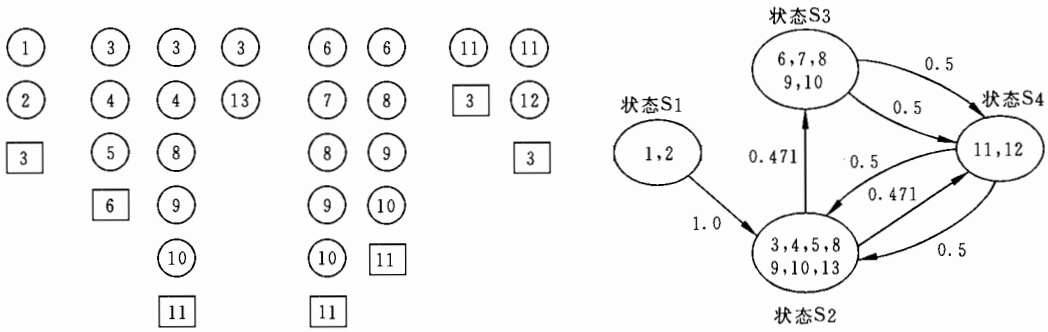


图 8.17　用 AFAP 法调度一条路径

控制流图表示了有限离散时间齐次马尔科夫链，即从节点v_i向节点v_j转移的概率与在节点v_i停留了多长时间无关。在控制流图中，节点表示数据操作，从一个数据操作转移到另外一个数据操作的转换概率与在该数据操作上执行了多长时间无关。因此，如果控制流图中的每条边(v_i,v_j)的执行的概率为p_{ij}，则说明数据操作v_i执行之后，继续执行数据操作v_j的概率为p_{ij}。例如在图 8.16 中，边(v_3,v_4)被执行的概率为$p_{34}=0.9412$。对于每个调度路径，同样可以计算路径执行概率。对于调度路径$SP=(3,4,5)$，起始节点为$Head(SP)=3$，后续节点$Succ(SP)=6$，根据公式(8.18)可以计算该路径执行的概率为$Prob(SP)=p_{34}\times p_{45}\times p_{56}=0.471$。全部路径的执行概率计算出之后，就可以计算有限状态机中的状态转移概率。例如，从状态 S3 到 S4 有两条路径，路径$SP_1=(6,7,8,9,10)$，路



(a) 调度路径　　(b) 有限状态机和AFAP调度

图 8.18　调度路径和 AFAP 调度

径 $SP_2=(6,8,9,10)$。如果要计算从状态 S3 到 S4 的状态转移概率，可以按照式(8.19)计算：$p_S(3,4)=Prob(SP_1)+Prob(SP_2)=0.5+0.5=1$。假定在完整硬件的行为描述被执行的过程中，令随机数 X_1,X_2,X_3,X_4 分别为表示状态 S1,S2,S3 和 S4 的执行时间，则 ASAP 调度结果的执行时间的数学期望值可以通过求解式(8.21)的线性方程组得出，对于现在的问题，线性方程组为

$$\begin{aligned}X_1 &= 1\\ X_2 &= X_1 + X_4\\ X_3 &= 0.471X_2\\ X_4 &= 0.471X_2 + X_3\end{aligned}$$

求解该问题可以得到$\{X_1,X_2,X_3,X_4\}=\{1,17.0,8.01,16.03\}$。最终可以得出 $X_{sch}=1+17.0+8.01+16.03=42.04$ 个时钟周期。

8.4.4　动态环调度

与 AFAP 调度技术不同，动态环调度技术并不切断控制流图中的反馈边，在调度过程中也不中断路径的生成。只有在违反约束条件时才中断路径生成，从而减少了路径个数，减少了路径生成的复杂性。

动态环调度方法从控制流图的第一个节点开始搜索路径，第一条路径以控制流图的第一个节点为起始点，在这个路径上依次增加节点，直到违反约束条件时为止，得到一条路径。下面的伪代码给出了路径生成的算法，算法的入口是函数 SearchPath(n)，在开始搜索路径时，函数的输入变量为控制流图的第一个节点。在节点 n 为搜索的起始节点时，对 n 的每个后续节点 x，调用 BuildPath(P(n),x)以建立一个新路径，每当遇到违反约束条件的情况，则终止一条路径，且将违反约束条件时的节点的后续节点作为新路径的起始节点，再次调用 SaerchPath 进行路径搜索。对于图 8.16 的控制流图，依次进行路径搜索可以得到图 8.19 所示的结果。在这个例子中，约束条件只是数据的相关性。每条路径最下面的方框标出的是路径的后续节点。搜索得到这些路径之后，可以构造有限状态机，动态环调度得到的有限状态机如图 8.20 所示。

```
SearchPath(n)
begin
    令节点 n 为一个起始节点,将节点 n 增加到起始节点表中;
    初始化一个新路径 P(n);
    对于节点 n 的每个后续节点 x,调用 BuildPath(P(n),x)建立路径;
end SearchPath;

BuildPath(P(n),x)
    if (P(n),x) 违反约束条件;
        令路径 P(n)的后续节点为 x;
        如果 x 不是起始节点表中的节点,以 x 为起始节点调用 SearchPtah(x);
    else 将节点 x 增加到路径 P(n);
        对每个节点 x 的每个后序节点 y,调用 BuildPtah(P(n),y);
    end if
end BuildPath;
```

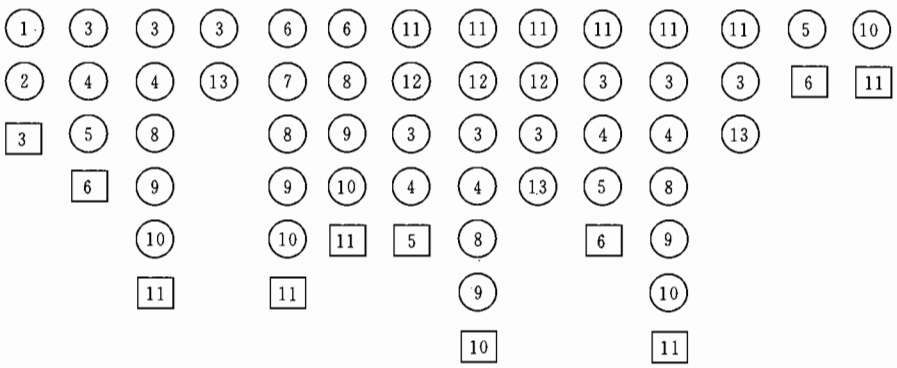


图 8.19　动态环调度得到的路径及其后续节点

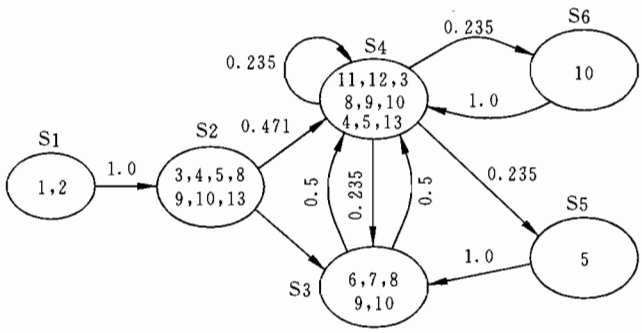


图 8.20　动态环调度得到的有限状态机

动态环调度技术与 AFAP 法的不同之处主要表现在两个方面:① 保持了控制流图中

的反馈环，从而允许并行执行迭代循环内部的部分；② 自由终止路径路径的生成，只考虑是否违反约束条件。在图 8.20 中，路径 $SP_7=(11,12,3,4,8,9)$，$SP_8=(11,12,3,4)$，$SP_9=(11,3,4,5)$，$SP_{10}=(11,3,4,8,9,10)$，循环的第 i 次迭代可以与第 $i+1$ 次迭代同时完成。对于控制流图 8.16 中的数据操作序列(3,4,8,9,10,11,3,4,8,9,10,11,12,3,13)的执行，动态环法的调度结果按照状态序列(S2,S3,S4)的顺序执行，这需要 3 个时钟周期。为了执行同样的数据操作序列，AFAP 法的调度结果按照状态序列(S2,S4,S3,S4,S2)的顺序执行，需要 5 个时钟周期。可以并行处理循环中的不同迭代，意味着动态环法调度的执行速度比 AFAP 法快。对于动态环调度，可以计算其执行时间的期望值为 33.5。当控制流图较为复杂时，AFAP 调度技术得到的路径个数会急剧增加，而动态环调度技术在路径搜索过程中切断环路，从而可以有效减少路径个数。

8.5 分配技术

分配是指确定系统中使用哪几个器件以及器件之间如何连接。分配包括三项内容：分配寄存器或 RAM，以存储数据值；分配运算单元，以完成指定的数据运算；分配器件之间的连接路径，以完成数据在器件间的传递。当然，一般情况下这三项内容相互关联。其中为每个数据操作项分配硬件运算单元和为寄存器(或其他存储单元)分配数值又称为数据通道分配。分配得到的结果是硬件的寄存器级表示。分配过程可能也要受到一些约束条件的限制，比如最大延时、最大面积、最大功耗或者最大成本等等。

分配算法一般可以分为两类：一类是迭代构造法，另一类是全局构造法。迭代法每次只根据局部原则选择下一个要分配的内容，每次只对一项内容进行分配，直到所有内容全分配好为止。全局法总是对所有未分配的内容进行检查，试图发现使总体目标最优的一项并进行分配。穷举搜索是全局分配法的最简单的例子。它能保证结果的最优性，但采用穷举搜索法所需的计算量可能不能容忍。迭代法每次分配时只对很小的空间进行搜索，因此计算效率较高，但一般不能保证找到最优解。

8.5.1 分配的基本问题

分配要解决的第一个问题是分配寄存器以保存数据。图 8.21 给出了对图 8.3 的调度结果分配寄存器和运算单元的情况。开始分配时，必须知道输入和输出的一些要求。这里我们假定必须对输入数据锁存后才能开始运算，因此增加了状态 S0 来完成输入数据锁存。

图 8.21 中的深色小圆点表示数据存储要求。例如在状态 S0 和 S1 之间，要求存储 5 个数据值，它们是输入的 5 个数据。可以任意分别把 5 个输入的数值 A,B,C,D 和 E 分配给 5 个寄存器 R1,R2,R3,R4 和 R5。在状态 S1 和 S2 之间同样需要存储 5 个数据值，因为此时寄存器 R1,R2,R3 和 R4 中的数据已不再有用，所以可以重复使用这些寄存器。确定哪个数据使用哪个存储单元对器件间互连的复杂程度影响很大，也会影响信号延时、芯片面积、器件种类和数量、器件的扇入/扇出要求等项内容，因此会影响系统成本。

分配的第二项工作是分配运算单元。分配可能会受到器件库中运算单元种类的限制，

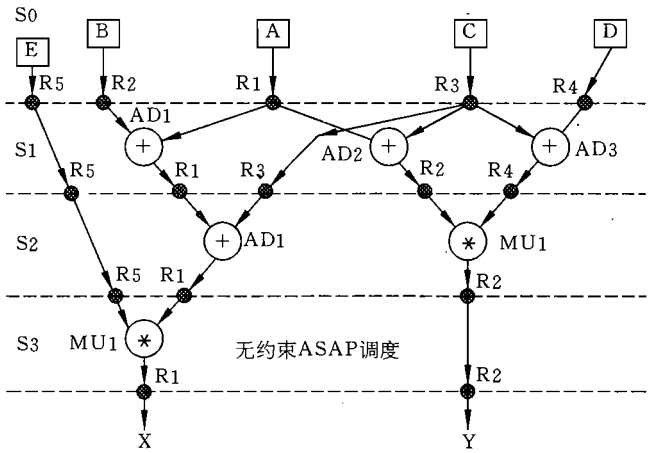


图 8.21　对按 ASAP 原则调度的结果分配寄存器和运算单元

这种情况称为从底向上的分配。如果可以使用任何需要的器件，而不受库单元中是否存在这些器件的限制，称为无约束分配或从顶向下的分配。对于同一设计问题，为满足不同的设计要求可以采用不同的分配方法。对于这里讨论的例子，我们假定器件库中有加法器、乘法器、寄存器、多路选择器等基本单元，采用从底向上的分配方法。

在状态 S1 中，3 个加运算同时运行，需要 3 个加法器，分配 3 个加运算分别由加法器 AD1，AD2 和 AD3 完成。在状态 S2 中，要执行 1 个加运算和 1 个乘运算，加运算可以重复使用加法器 AD1，乘运算可以分配由乘法器 MU1 完成。在状态 S3，由于只执行 1 个乘运算，所以可以重复使用乘法器 MU1。数据单元分配的结果也画在图 8.21 中。

分配的第三项工作是分配数据通路，目的是完成器件之间的互连。图 8.22 给出了按图 8.21 分配寄存器和运算单元后再分配数据通路后的结果，表 8.3 列出了所要求的器件种类和数量。

表 8.3　ASAP 法进行调度、分配结果总结

器　　件	个　　数
加法器	3
乘法器	1
寄存器	5
2×1 多路选择器	4
3×1 多路选择器	2

8.5.2　迭代进行调度和分配

从上述调度、分配的例子可以看出，调度和分配不应该是完全独立的两个过程。对一个算法进行调度和分配过程中，可以看到下述现象：

(1) 如果在同一个状态内完成两个同样的运算，必须使用两个不同的运算单元。如果

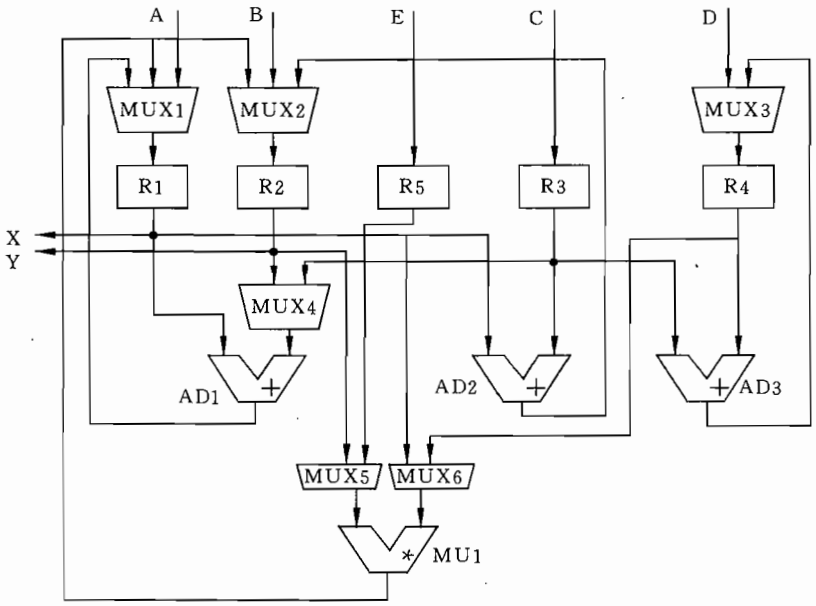


图 8.22　ASAP 原则调度结果的一种数据通路

限制只能使用一个这样的运算单元，必须采用另外的调度方案。即调度过程中隐含确定了某些分配的内容。

(2) 为了得到最优调度，如果某些数据运算需要的运算时间长于一个系统时钟周期，应该在调度时知道这样的运算需要多长时间，一般在对运算单元分配了器件后才能得到这些信息，即调度时需要知道某些分配后才能确定的信息。

(3) 为了减小器件成本，需要减少器件个数。如前所述，调度过程会影响器件个数。一般地讲，减少器件个数，需增加控制状态个数，即降低运算速度。

对这些现象进行分析就可以发现，一方面最优的分配结果(即利用较少的运算单元实现电路)取决于在调度时是否建立了实现最优分配的条件。另一方面，为了调度最优调度结果需要知道分配后才能知道的运算单元的信息。为了实现系统级的最优，需要综合考虑调度和分配，有时需要迭代进行调度和分配。

综合考虑调度和分配的最简单的方法是首先设定分配方案的限制，比如限制可利用的运算单元个数，然后在满足这些限制的前提下进行调度，最后根据调度结果具体进行分配。也可以对于分配方案的不同限制条件，迭代进行调度，调度完成后再进行分配，从中选出最满意的调度、分配方案，每次迭代选择限制条件需要设计者的设计知识。除此之外，还可以同时进行调度和分配，这种方法总是对于特定的目标，比如最大速度、最小面积等，同时考虑调度和分配。

为了说明在分配方案存在限制条件的情况下如何进行调度，假定设计者希望只使用一个加法器和一个乘法器实现图 8.2 讨论的例子，这种情况下可以得到图 8.23 所示的调度结果。由于限制了运算单元的个数，运算速度就可能降低，这种调度方案增加了一个状态，完成运算所需的时间从 4 个时钟周期增加到 5 个时钟周期。同时，根据原始的电路指

标并不能得到这个调度方案，将原始电路 SYNEX1 指标进行变换如下：

```
X<=E*((A + C) + B);
Y<=(A + C) * (C + D);
```

上面两个表达式中存在公共部分(A+C)，调度时可以先计算出这一项。对变换后的表达式进行调度得到的结果要比对原始表达式进行调度的结果好。

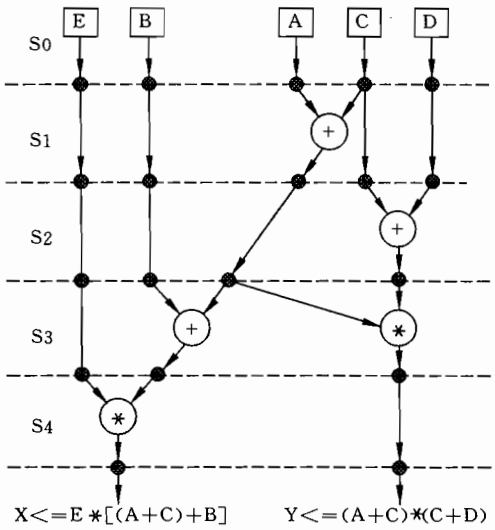


图 8.23　只使用一个加法器、一个乘法器的调度结果

图 8.24 是对图 8.23 的调度方案分配的寄存器和运算单元的结果。对这种调度、分配方案可以按图 8.25 分配数据通路。这个方案所需要的器件种类和个数如表 8.4 所示。这个方案只使用了一个加法器和一个乘法器，多路选择器也比原始方案规模小。如果不考虑连线所占面积，这个方案的芯片面积应该比原始方案小。

表 8.4　限制使用一个加法器、一个乘法器的调度、分配结果

器　　件	个　　数
加法器	1
乘法器	1
寄存器	5
2×1 多路选择器	7

8.5.3　Gantt 表及器件的利用率

Gantt 表是评价调度、分配方案的有效方法。表 8.5 是评价图 8.24 所示的调度、分配方案的 Gantt 表。在这个表中，符号 B 表示器件处于忙态，符号—表示器件处于休止状态。对于系统中的每个器件，Gantt 表中都有独立的一行，对于调度过程中的每个状态，Gantt 表中都有独立的一列。

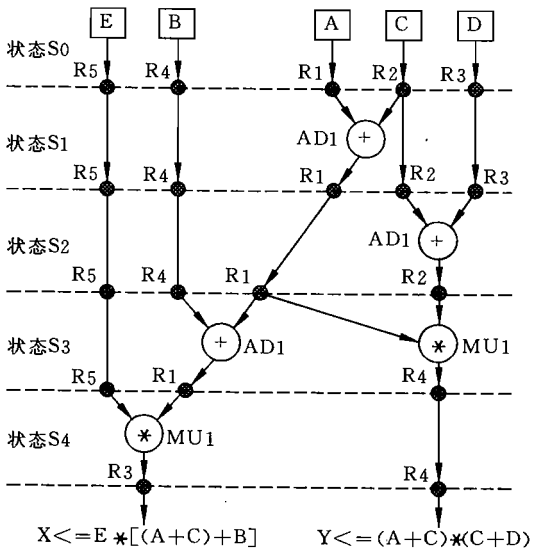


图 8.24　只使用一个加法器、一个乘法器的分配结果

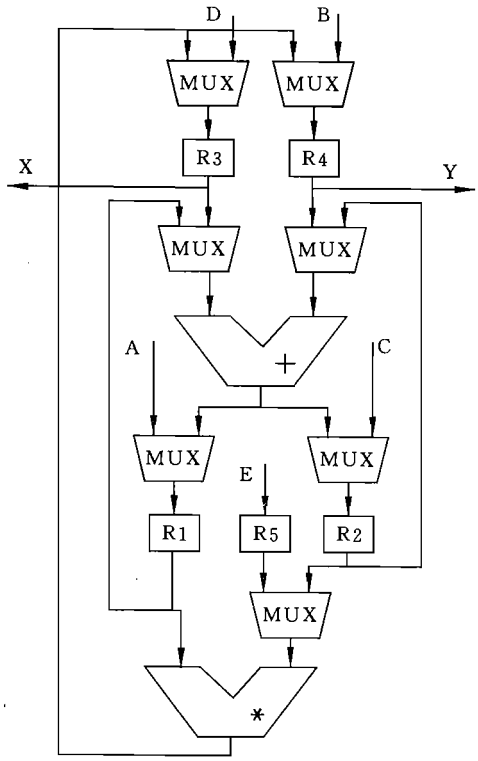


图 8.25　只使用一个加法器、一个乘法器条件下分配的数据通路

表 8.5 评价图 8.24 所示的调度、分配方案的 Gantt 表

运算单元	状态				
	S0	S1	S2	S3	S4
加法器 AD1	—	B	B	B	—
乘法器 MU1	—	—	—	B	B

执行时间是评价调度、分配方案的重要指标。如前所述，通常一个状态对应于一个系统时钟周期，所以可以用完成运算所需的状态个数表示系统的执行时间。对于调度和分配来讲，自然可以把完成运算所需状态个数作为评价调度、分配方案好坏的一个标准，即在调度和分配过程中，可以把减小状态个数作为目标。显然，若完成运算所需状态个数少，则得到的硬件的速度快、延时小。

除了执行速度之外，器件的利用率是评价调度、分配方案好坏的另一个指标。对于一个器件，它处于忙态的状态数与完成运算所需的状态总数之比称为器件的利用率。比如对于表 8.5 所示的 Gantt 表，器件完成运算总共需要 5 个状态，加法器 AD1 在其中 3 个状态处于忙态，所以其利用率为 60%。在许多情况下，调度、分配时对器件的利用率进行优化也是很好的目标函数。通常器件利用率高意味着器件个数少，但同时可能意味着完成运算所需的状态个数多，即器件的工作速度低。在设计过程中通常需要在速度和面积之间进行折衷，即需要在器件个数及状态个数之间进行折衷。在某些情况下，器件利用率很重要，但在另外情况下，器件利用率可能无关重要。无论如何，如果器件利用率很低，一定不是最好的设计方案。

8.5.4 Greedy 分配法

Greedy 分配方法是最简单的迭代构造式分配法，它只根据局部准则进行分配。在分配过程中每一个数据操作项分配给下一个可用的硬件运算单元，每一个数据值分配给下一个可用的寄存器，每一个数据路径分配给下一个可用的总线或多路选择器。在调度过程中的每一时刻，只有在当前存在的器件全处于忙态时才增加新器件。在分配过程中，总是首先选择重新使用当前处于休止状态的器件，而不是增加新器件。调度过程总是根据当前器件是否处于休止状态来决定当前器件是否可用，而不考虑重新使用该器件是否会对未来某时刻的分配产生不利影响。这种方法容易实现自动化，在许多情况下可以得到很好的分配方案，但很难得到最优的分配结果。本小节前面的讨论采用的就是 Greedy 分配算法。

8.5.5 穷举搜索法

全局分配技术采用了比迭代方法更为可靠的全局选择准则来把数据操作项分配给硬件运算单元。因为穷举搜索法可以搜索更大的设计空间，故得到最优分配结果的可能性较大。但穷举搜索技术可能花费很多计算时间，算法复杂性也较大。

穷举搜索是全局分配技术的一个实例，这种方法通过在整个分配空间中搜索来选择最优的分配对象。显然这种方法易于发现最优的分配结果，但除了极简单的情况之外，计

算成本通常是令人难以接受的。即使使用最快的计算机，对于一般复杂度的电路，也很难在合理的计算时间内用穷举搜索法求出分配方案。但因后面对分配算法的讨论时要用到穷举搜索的概念，所以这里简单作了介绍。

8.5.6 左边算法

左边算法通常用在调度结束之后，为各数据项分配寄存器。这也是一种迭代构造式分配算法，算法步骤如下。

对调度后各状态要求存储的各数据项（即调度图中的深色小圆点）用字母按任意顺序标号。例如对于图 8.11 所示的调度图，可以对各个数据项编号，如图 8.26 所示。

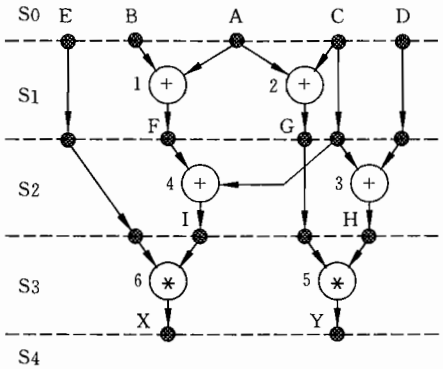


图 8.26　数据存储项编号后的数据流图

根据数据项编号后的数据流图，建立一个反映各数据项生存时间的图。图 8.27 是根据图 8.26 建立的数据项生存时间图。从图 8.26 可以看出，数据项 E 应该在状态 S1，S2 和 S3 全保持其数值，所以在图 8.27(a)中对应于数据项 E 的线段从状态 S1 开始，到状态 S3 终止。与此类似，在图 8.26 中，数据项 G 产生于状态 S1 的结束处，要在状态 S2 和 S3 保持其数值，所以在图 8.27(a)中，对应数据项 G 的线段从状态 S2 开始，到状态 S3 终止。对应其他数据项的线段可以用与此类似的方法作出。这就是说，图 8.27(a)中对应各个数据项的线段不但反映了数据项的生存时间，而且反映了数据项产生于哪个状态，终止于哪个状态。

按照数据项产生的时刻对数据项分组，且对于每一组数据项按照数据项生存时间的长短排序。不同的数据组按照各组数据项产生时间的先后排序。这样排序的结果是：对于不同数据项组，产生时刻早的数据项组排在左边，对于同一组数据项，终止时刻早的数据项排在左边。对于图 8.26 所示的数据流图，按照上述规则排序后的生存时间图如图 8.27(b)所示。

对于排序后的生存时间图中最左边的一组数据项，从左到右每项数据分配一个寄存器。实际上，这时为哪项数据分配哪个寄存器并不重要，从左到右排序只是为了方便。对于上面讨论的问题，对数据项 A～E 中每一项都分配一个寄存器，分配的结果如图 8.27(c)所示。

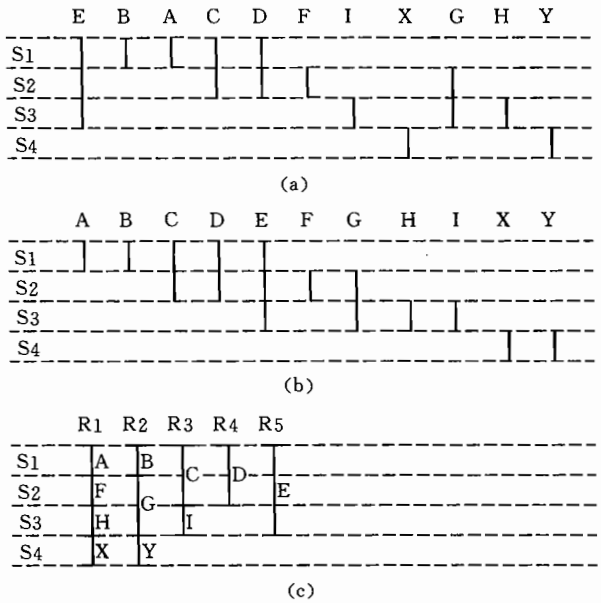


图 8.27　各数据项的生存时间

对于排序生存图中的第二组及第二组之后的数据项，继续分配寄存器。对每个数据项分配寄存器时，首先检查现有的寄存器是否可以重复使用，如果可以重复使用，则重复使用存在的寄存器。如果不可以重复使用，则为它分配一个新寄存器。例如对于图 8.27(b)，数据项 F 产生于状态 S1 的结束时刻，需要在状态 S2 保存其数值，而数据项 A 只需在状态 S1 保存其数值，所以为数据项 F 分配寄存器时可以重复使用寄存器 R1。与此类似，对于图 8.27(b)中的数据项 G，它产生的时刻为状态 S1 的结束时刻，需要在状态 S2 和状态 S3 保持其数值，而在状态 S1 的结束时刻寄存器 R2(原来保存数据项 B 的值)的数值已不需再保持，可以重复使用寄存器 R2 保存数据项 G 的数值。再比如对于数据项 H，它产生的时刻为状态 S2 的结束时刻，这时寄存器 R1 的值已不需再保存，所以可以重复使用寄存器 R1 保存数据项 H 的数值。存取同样的规则，可以继续为数据项 I，X 和 Y 分配寄存器。最后的分配结果如图 8.27(c)所示。

可以看出，这种算法分配寄存器的过程总是从左到右，因此称为左边算法。由于第一组数据项全要在状态 S1 保持其数值，所以需要对第一组数据项中的每一项分配一个寄存器。左边算法只考虑尽可能少地使用寄存器，并没有考虑如何把各个寄存器以及运算单元连接起来，即没有考虑互连路径的成本。比如在上面的例子中，数据项 H 和 I 都可以分配给寄存器 R1 和 R3 中的任一个，不同分配方法可能需要的多路选择器不同。为数据项分配寄存器的过程有时称为把数据项与寄存器连接。把数据项与寄存器连接过程中，如果发现在不增加寄存器个数的前提下，可以有不同连接方法，应该把这些连接推迟，等到为数据操作分配运算单元且分配互连路径时再把这些数据与寄存器连接，这样有可能降低互连路径的成本。

8.5.7 为数据操作分配运算单元并分配互连路径

为数据操作分配运算单元，为数据项分配寄存器以及为互连路径分配多路选择器或总线，都会影响硬件成本，且互相关联。如果采用 8.5.6 节介绍的左边算法为数据项分配寄存器，但不考虑分配运算单元及分配互连路径的成本，一般不可能得到最优分配方案。

为了说明分配运算单元，分配寄存器以及分配互连路径之间的相互影响，我们以图 8.26 的数据流图为例讨论如何分配运算单元及互连路径。假定已经采用左边算法为数据项分配了寄存器，并按 8.5.6 节末尾讨论的方式，标出了有多种分配可能性的数据项，把它们推迟到分配运算单元及分配互连路径时再完成。

我们已经限定了设计最多只能使用两个乘法器、两个加法器，左边算法执行过程中已经知道至少要采用 5 个寄存器存储所有数据项，且 5 个寄存器足以存储所有数据项。图 8.28 给出了设计中所需的运算单元和寄存器。为每个数据操作项分配运算单元以及最后为数据项分配寄存器应该使完成互连路径所需的多路选择器的数目最小。在某些特殊情况下，有可能一个多路选择器也不需要，直接把运算单元和寄存器按要求连接起来就可以完成设计。在必须使用多路选择器的情况下，我们希望多路选择器的规模，即输入端个数最小。应该指出的是，尽管我们使用了多路选择器讨论分配过程，设计时也可以使用三态门和总线实现同样的功能。

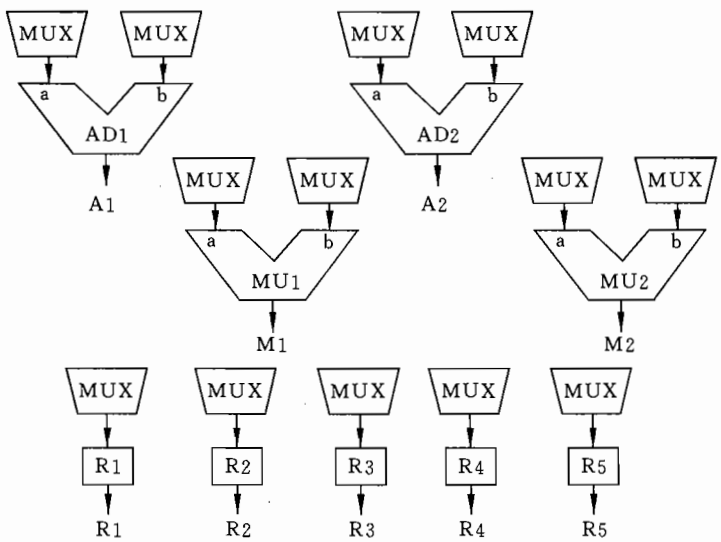


图 8.28 调度、分配中需要的运算单元及寄存器

为了使多路选择器的规模小，要求多路选择器的输入的信号个数少。表 8.6 给出了一个空表格，用来记录分配运算单元和互连路径过程，并帮助设计者作出决策。在这个表格中，每一行对应于一个状态，每一列对应于一个寄存器或运算单元的一个输入。在分配过

程中，将用这个表格记录各种连接关系。

表 8.6　分配运算单元和互连路径的空连接表

状态	器件 AD1			器件 AD2			器件 MU1			器件 MU2			寄存器				
	OP	a	b	OP	a	b	OP	a	b	OP	a	b	R1	R2	R3	R4	R5
S1																	
S2																	
S3																	
S4																	

根据寄存器分配时产生的图 8.27(c)可知，在状态 S1 中，寄存器 R1，R2，R3，R4 和 R5 分别分配给数据项 A，B，C，D 和 E。根据调度图 8.26，在状态 S1 中，我们需要把加法器 AD1 和 AD2 分别分配给数据操作 1 和 2。由于此时我们还没有分配任何运算单元，可以任意把这两个加法器分配给这两个数据操作。不失一般性，可以把加法器 AD1 分配给数据操作 1，把加法器 AD2 分配给数据操作 2。由于算术加法满足交换律，我们可以任意把数据项 A(寄存器 R1 的输出端)或者数据项 B(寄存器 R2 的输出端)连接到加法器 AD1 的 a 输入端，把另外一个连接到加法器 AD1 的 b 输入端。与分配加法器类似，由于此时还没有分配任何互连路径，这一决策可以任意作出。我们任选 R1 连接到 AD1 的 b 输入端，将 R2 连接到 AD1 的 a 输入端。与加法器 AD1 类似，由于我们还没有对加法器 AD2 作任何连接，可以任选 R3 连接到加法器 AD2 的 b 输入端，把 R1 连接到加法器 AD2 的 a 输入端。这样就完成了分配运算单元和互连路径的第一步，这时的连接关系如表 8.7 所示。

表 8.7　分配运算单元和互连路径的第一步

状态	器件 AD1			器件 AD2			器件 MU1			器件 MU2			寄存器				
	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																	

在状态 S2，即分配运算单元和互连路径的第二步，我们必须把加法器 AD1 和 AD2 分别分配给数据操作 3 和 4，把寄存器 R1 和 R2 分别分配给数据项 F 和 G。左边算法把寄存器 R1 分配给数据项 F，寄存器 R2 分配给数据项 G。但如果将它们交换可以降低互连成本的话，那么应该设法将它们交换。

数据操作 3 的输入数据为数据项 C(保存在 R3 中)和 D(保存在 R4 中)，由于这时 R3 的输出已经连接在加法器 AD2 的 b 输入端，如果把加法器 AD2 分配给数据操作 3，可以

降低连接代价。数据操作 4 的输入数据是数据项 F 和 G，如果把寄存器 R1 分配给数据项 F，把寄存器 R2 分配给数据项 G，利用前一状态中已经建立起来的寄存器 R1 与加法器 AD1 的 b 输入端之间的连接关系，可以降低互连代价。由此可知，把运算单元分配给数据操作的方法以及把寄存器分配给数据项的方法，会明显影响状态 S2 中的互连成本。这种分配方法要求把加法器 AD1 的输出端(A1)在状态 S1 结束时连接到寄存器 R1 的输入端，把加法器 AD2 的输出端(A2)在状态 S1 的结束时连接到寄存器 R2 的输入端。这样寄存器 R1 和 R2 都连接了两个输入数据项，寄存器 R1 的输入端通过多路选择器连接到数据项 A 和 F(即加法器 AD1 的输出 A1)，寄存器 R2 的输入端通过多路选择器连接到数据项 B 和 G(即加法器 AD2 的输出 A2)。表 8.8 给出了这里介绍的分配方法。在这个表中，对应加法器 AD1 的输入端 b 的一列全是寄存器 R1，这说明直到状态 S2，加法器的 b 输入端还不需要多路选择器。而对应加法器 AD1 的输入端 a 的一列，在状态 S1 和 S2 分别连接到寄存器 R2 和 R3，这说明加法器的 a 输入端需要通过一个 2 输入多路选择器连接到寄存器 R1 和 R2 的输出端。

表 8.8　分配运算单元和互连路径的第二步

状态	器件 AD1			器件 AD2			器件 MU1			器件 MU2			寄　存　器				
	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																	

在状态 S3，即分配运算单元和互连路径的第三步，我们必须把两个乘法器 MU1 和 MU2 分配给两个数据操作 5 和 6，由于我们还没有为任何乘法运算分配运算单元，所以可以任意将两个乘法器 MU1 和 MU2 分配给数据操作 5 和 6。但是，根据左边算法，在状态 S3 数据项 H 和 I 都可以占用寄存器 R1，R3 或 R4，对数据项分配寄存器方法不同，硬件的连接成本也不同。

数据项 I 由加法器 AD1 在状态 S2 产生，我们必须建立从加法器 AD1 的输出端到保存数据项 I 的寄存器的输入端之间的数据通路。由于加法器 AD1 的输出端已经连接到寄存器 R1 的输入端，所以，如果把寄存器 R1 分配给数据项 I，则不必从 AD1 的输出端到寄存器的输入端建立新的连接通路，可以降低连接成本。这种为数据项分配寄存器的信息不能直接通过左边算法得到。

数据项 H 由加法器 AD2 在状态 S2 产生，我们必须建立从加法器 AD2 的输出端到保存数据项 H 的寄存器的输入端之间的数据通路。由于数据项 H 只能占用寄存器 R1，R3 或 R4 中的一个，而加法器 AD2 的输出没有连接到这三个寄存器之间的任何一个，且数据项 I 占用了寄存器 R1，所以可以把寄存器 R3 或 R4 之间的任何一个分配给数据项 H。我们把寄存器 R3 分配给数据项 H 后，得到表 8.9 所示的连接关系表。

表 8.9　分配运算单元和互连路径的第三步

状态	器件 AD1			器件 AD2			器件 MU1			器件 MU2			寄存器				
	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																	

完成了所有数据操作之后，在状态 S3 结束时，应该把两个乘法运算的结果保存起来。尽管数据运算并不需要状态 S4，我们还是增加了状态 S4 用来输出运算结果。为数据项 X 和 Y 分配寄存器也有不同的考虑因素。首先，这时 5 个寄存器中原来保存的数据全已无用，即可以使用 5 个寄存器中的任意两个保存数据项 X 和 Y。其次，寄存器 R1，R2 和 R3 的输入端已经连接有 2 输入多路选择器，寄存器 R4 和 R5 只有一个数据输入项，不需在输入端连接多路选择器。由于数据项 X 和 Y 由乘法器 MU1 和 MU2 在状态 S3 产生，乘法器的输出端还没有与任何寄存器相连接，所以如果我们选择寄存器 R4 和 R5 保存最后运算结果 X 和 Y，应该在电路中的寄存器 R4 和 R5 的输入端新增加两个 2 输入多路选择器；如果我们选择用 R1，R2 或 R3 中的两个保存最后运算结果 X 和 Y，则需把原来连接在寄存器输入端的两个 2 输入多路选择器换成 3 输入多路选择器。到底哪种寄存器分配方案会节省成本，与具体工艺有关。应该等到确定了集成电路实现工艺后再作这种决策，即知道将两个 2 输入多路选择器换成 3 输入多路选择器的成本高，还是新增加两个 2 输入多路选择器的成本高后才能作出这种决策。

为了完成分配过程，我们决定把寄存器 R1 分配给数据项 X，把寄存器 R2 分配给数据项 Y，即把乘法器 MU1 的输出 M1 通过多路选择器连接到寄存器 R1 的输入端，把乘法器 MU2 的输出 M2 通过多路选择器连接到寄存器 R1 的输入端。表 8.8 给出了最后一步分配结束后的连接表，图 8.29 画出了运算单元和连接路径分配结束后的连接图。

表 8.10　分配运算单元和互连路径的最后一步

状态	器件 AD1			器件 AD2			器件 MU1			器件 MU2			寄存器				
	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	—	—	—

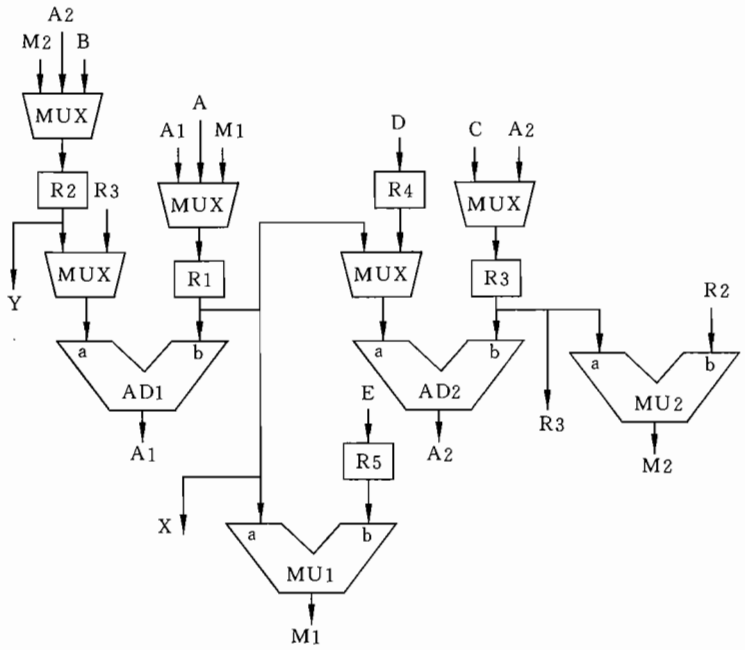


图 8.29　为排队调度法分配寄存器、运算单元及数据通路之后的连接图

8.6　根据分配图建立 VHDL 有限状态机模型

根据分配图很容易建立硬件的有限状态机 VHDL 模型。下面的 VHDL 源代码是根据图 8.23 所示的调度、分配结果建立的有限状态机 VHDL 模型。根据分配图建立 VHDL 模型的过程非常简单，首先定义了存储数据所需的寄存器并用 case 语句定义每个状态，然后在每个状态内添加每个状态内需要完成的数据操作，就完成了建立 VHDL 模型的过程。这里我们采用了 Moore 状态机，输入 CLK 为状态之间的转换提供时钟。尽管硬件的指标规范中并没有对状态 S4 之后系统进入什么状态作出规定，但在 VHDL 模型中仍可以假定状态 S4 之后系统再次进入 S0。由于 Moore 状态机的输出与当前输入值无关，所以我们把输出语句放在 case 语句之外。

```
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
    STATEP: process(CLK)
    begin
        if CLK ='1' then
            case STATE is
                when S0=>
                    -- 数据选择
                    R5<=E; R4<=B; R3<=D; R2<=C; R1<=A;
                    -- 控制选择
                    STATE<=S1;
                when S1=>
                    -- 数据选择
                    R1 < R1 + R2;
                    -- 控制选择
                    STATE<=S2;
                when S2=>
                    -- 数据选择
                    R2<=R2 + R3;
                    -- 控制选择
                    STATE<=S3;
                when S3=>
                    -- 数据选择
                    R1<=R4 + R1;
                    R4<=R1 * R2;
                    -- 控制选择
                    STATE<=S4;
                when S4=>
                    -- 数据选择
                    R3<=R5 * R1;
                    -- 控制选择
                    STATE<=S0;
            end case;
        end if;
    end process STATEP;
    -- 输出赋值
    X<=R3;
    Y<=R4;
end FSM;
```

为了使器件模型与实际情况更为一致，应该在模型中增加启动信号以通知器件开始数据处理。还应该增加运算结束信号以通知外部电路运算已经结束，可以从输出端口获得正确数据。完成 VHDL 模型之后，设计者应该对模型进行仿真验证。如果仿真结果正确，设计者应该把整数寄存器映射为二进制位矢量寄存器，然后按第 7 章中讨论过的方法分别设计控制单元和数据单元。当然控制单元可以为硬连接控制单元，也可以为微代码控制单元。数据单元包括加法器和乘法器，还包括数据存储单元。完成了数据单元和控制单元设计之后，设计者还需要设计位矢量输入、输出电路。设计可以采用从顶向下的全定制设计；也可以从器件库中选择器件，采用从底向上的半定制设计方法。

根据第 7、8 两章中讨论的知识，我们具备了利用 VHDL 设计集成电路的能力。这就是说，我们应该有能力从硬件的自然语言描述开始，设计出寄存器级甚至逻辑门级电路，也应该有能力设计出控制单元和数据单元。图 8.30 总结了利用 VHDL 设计集成电路的方法。

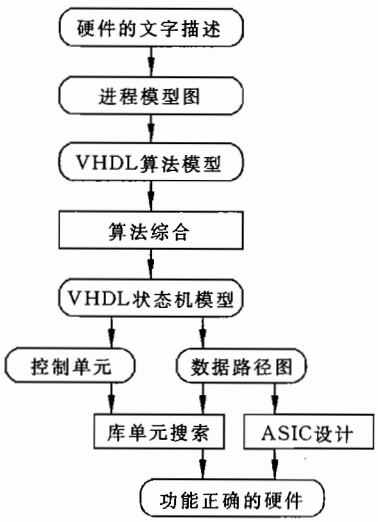


图 8.30　利用 VHDL 设计集成电路的流程

第9章 VHDL在其他设计领域的应用

众所周知，最初 VHDL 的设计目的主要是数字电路的描述和仿真，因此 VHDL 实际上是一种基于事件驱动机制的语言。但在成为 IEEE 标准之后，VHDL 的用途已经扩展到集成电路设计甚至电路板级设计的各个方面。现在，使用 VHDL 进行硬件综合、作为国际通用的设计信息传递手段都已经成为典型应用。不仅如此，VHDL 在模拟电路设计、混合信号电路设计、微机械/微结构设计、性能分析、数字电路测试等方面的应用正在被世界各地的学者进行广泛的研究，并已经取得了一些成果。这一章将简要介绍 VHDL 在这些方面的应用。

9.1 VHDL 在模拟/混合信号系统设计中的应用

IEEE 在 1987 年第一次颁布 VHDL 作为硬件描述语言标准，在 1993 年进行过一次修订。VHDL 最初的设计目的是数字系统的描述和仿真，支持从系统级直至门级等各个抽象层次模型的构造。VHDL 作为国际标准的出现，非常深刻地改变了数字集成电路的设计方法、设计工具，并且极大地推动了电子工业的发展。数字系统的主要特征即其中的信号，在时域和值域上都是离散的，因此 VHDL 在本质上属于离散事件硬件描述语言。属于连续系统的模拟和混合信号系统具有与数字电路同样甚至更加广泛的应用，并且其重要性正在日益提高，但是这些系统的设计尚缺乏一种统一的、高度自动化的方法。由于 VHDL 是一种具有丰富语法结构和强大的描述能力的语言，加之已经在数字系统设计中取得了重大成功，电子产业界和学术界都希望能够把 VHDL 应用于模拟和混合信号系统设计当中。通过一系列可行性研究和对现有混合系统描述语言如 MAST，HDL—A 等的考察，现在这方面的研究已经取得初步成果，1997 年 IEEE 通过了用于这一领域的 VHDL 扩展标准——VHDL 1076.1。

9.1.1 VHDL 1076.1 的发展过程

1989 年开始进行使 IEEE 的扩展 VHDL 语言能够支持连续系统和离散/连续混合系统设计的工作。起初这一工作仍由原来的 VHDL 修订组进行，他们希望能够进行必要的语言修改，并在 1993 年 VHDL 第一次修订时完成。但是，很快就发现工作量过大，于是 IEEE 成立了专门的工作组，在 IEEE 计算机协会设计自动化标准委员会的赞助下工作。1997 年 3 月这个工作组完成了语言草案，称之为 VHDL—A，于 1997 年下半年投票通过，现在改称为 VHDL 1076.1 或 VHDL—AMS(AMS 是 analog/mixed-signal 的缩写)，后

者是为了强调在混合信号系统设计领域的应用。不仅如此，IEEE 将在这个基础上，陆续颁布 VHDL 描述热力学、流体力学、机械等系统的扩展。

9.1.2 有关 VHDL 1076.1 的基础知识

VHDL 1076.1 的设计目标要求扩展后的语言是 VHDL 1076—1993 的一个超集，支持连续系统和连续/离散混合系统的层次式的描述和仿真，支持电子及其他非电子领域多个抽象层次上的建模。VHDL 1076.1 描述集总式系统，也就是说不需要描述偏微分方程，用该语言描述的特征方程的解可以包含不连续点。同时，VHDL 1076.1 模型中的离散和连续部分应通过一种灵活而有效的方式实现交互，并支持频域交流仿真和噪声仿真。

由以上设计目标出发，VHDL 1076.1 在 VHDL 1076—1993 的基础上得到发展，VHDL 1076.1 工作组在修订过程中尽量使用原有的语言结构，同时也进行了必要的扩展。

集总式系统的连续行为可以用微分和代数方程组描述，其中以时间作为独立变量。这类方程具有如下形式：

$$F(\boldsymbol{x},\mathrm{d}\boldsymbol{x}/\mathrm{d}t,t) = 0 \tag{9.1}$$

其中，F 是一列表达式，$\boldsymbol{x}$ 是未知矢量，$d\boldsymbol{x}/\mathrm{d}t$ 是未知矢量对时间的导数矢量，t 是时间。对连续集总式系统来说，微分和代数方程组是唯一系统化的、充分的描述方式。但是，许多系统方程没有解析解，所以实际中经常需要数值方法求近似解。这类数值解法在过去 20 年中已得到充分的研究，具备了很成熟的理论。需要指出的是，对于同一微分和代数方程组通常存在不同的数值解法。

由于描述集总式系统的需要，VHDL 1076.1 必须提供描述微分和代数方程组的方法，然而不必在语言中定义特定的求解方法。修订工作组把求解算法称为模拟求解器。也就是说，VHDL 1076.1 的语言结构能够描述系统方程以及模拟求解器应达到的解的特征，但是不具体要求特定解法，这样为实际的仿真器留有足够余地实现方程的求解。

9.1.3 VHDL 1076.1 语言

VHDL 1076.1 是在描述离散系统的 VHDL 1076—1993 的基础上发展的，只增加了少数描述连续系统的必要特征，因此与传统的连续系统描述语言具有很大的不同，本小节只介绍新增加的特征。

1. 量(quantity)

微分和代数方程系统中的未知量是时间的解析函数，数值解法在求解时，根据算法，在同一时刻解出所有未知量的数值。VHDL 1076—1993 中的对象没有与之类似的更新方式，因此 VHDL 1076.1 引入了一类新的对象，量用关键字 quantity 定义，表示微分和代数方程组中的未知量。

量可以是标量、矢量，这时其数据类型应为浮点类型；量也可以是混合类型，但是必须具备浮点类型的子元素。提出浮点数的要求是为了近似系统的连续特征。

除了程序包之外，量可以在任何能够定义信号的地方定义。以下是定义量的例子，该例定义了两个实数类型的量 qa 和 qb。

```
quantity qa,qb : real;
```

量也可以被定义为端口列表中的接口元素，称为量端口(quantity port)。类似于端口信号，这时也要为其指定 I/O 模式，下面是定义量端口的例子：

```
entity adder is
        port(quantity in1,in2 : in Real;
             quantity sum : out Real);
end entity adder;
```

量有一系列相关属性，分为显示定义和隐含定义属性。本小节后面将介绍显示定义属性。量的隐含定义属性有以下两条：

(1) 对任意量 Q，Q'Dot 表示 Q 对时间的导数，Q'Dot 具有与 Q 相同的数据类型；

(2) 对任意量 Q，Q'Integ 表示 Q 从 0.0 时刻到现在时刻对时间的积分，Q'Integ 具有与 Q 相同的数据类型。

这两条属性都可以向更高阶扩展，即 Q'Dot'Dot 表示 Q 对时间的二阶导数，Q'Integ'Integ 表示对 Q'Integ 的积分，以次类推。

2. 容差组(tolerance group)

求解连续系统微分与代数方程组，必然有解的精度的问题，而且不同的量常常要求不同的精度。为了解决这个问题，VHDL 1076.1 引入了容差组的概念。同一容差组内的元素具有相同的数值精度，其中元素可以是量或特征表达式。尽管没有专门规定，同一容差组内的元素的值域一般是相同的。VHDL 1076.1 没有定义具体的容差组，由仿真工具自行确定。

容差组由一个字符串类型的容差代码(tolerance code)标志。某一量的容差代码在定义时通过子类型直接或间接声明，例如：

```
subtype Charge is Real tolerance "tol1";
```

以上代码定义了浮点子类型 Charge，其容差代码是 tol1。又如：

```
quantity q0 : Charge;
quantity q1,q2 : Charge tolerance "tol2";
```

q0 被定义为 Charge 类型，并继承了容差代码 tol1，而 q1，q2 也属于 Charge 类型，但容差代码被定义为 tol2。应该说明的是，任意量 Q 的时间导数 Q'Dot 与 Q 属于同一容差组，即具有相同的容差代码。

3. 守恒系统(conservative system)

守恒系统是 VHDL 1076.1 的主要描述对象。守恒系统，例如服从 Kirchhoff 定律的电学系统，可以被描述为图论模型。在电学系统中，图的顶点代表电路节点，而图的边代表电流通过的分支。实际上常见的工程系统如电学系统、热学系统、流体力学系统、平动及旋转系统都属于守恒系统，可以由一组守恒方程(在电学中就是 Kirchhoff 定律)以及具体系统的特征方程描述。因此，VHDL 1076.1 采用了特殊的语法结构使得建模时不再需要

输入系统的守恒方程，只需描述特征方程即可。

VHDL 1076.1 语言的设计体现了以上的图论观点。守恒方程中的未知量被称作分支量(branch quantity)，它分为两种类型：across 量和 through 量，前者表示类似作用力的效果，后者表示类似流量的效果。表 9.1 列出不同工程领域的两种分支量。

表 9.1 不同工程领域的两种分支量

工程领域	across 量	through 量
电学	电压	电流
热学	温度	热流率
流体力学	压力	流量
平动动力系统	速度	应力
旋转动力系统	角速度	转矩

守恒系统的本征方程一般总表示为用一定的方式把 across 量和 through 量联系起来，例如，电阻可以看作一个独立分支，欧姆定律就是其本征方程，其中电压 V 和电流 I 的关系是 $V=R\times I$。

分支量总是参考两个终端(terminal)来定义的，而终端总具有特定性质(nature)。下面的代码给出了电学系统的基本定义，包括了定义的终端及其性质。终端的性质也可以是数组或记录类型。任意性质 N 都有预定义属性 N'across 和 N'through，分别表示该性质的 across 量和 through 量，对于电学系统的例子来说，Electrical'across 就是 Voltage，Electrical'through 就是 Current。

```
-- (a) 电学系统的一些定义
subtype Voltage is Real tolerance "voltage";
subtype Current is Real tolerance "current";
natural Electrical is Voltage across Current through;

-- (b) 终端及其性质
terminal t1,t2 : Electrical;
quantity v across i1,i2 through t1 to t2;
```

分支量的数据类型由终端的性质继承得到。在上面的例子中，across 量 v 表示两终端 t1,t2 的电势差，属于 Voltage 类型，容差代码为“voltage”。i1,i2 表示 t1,t2 之间的电流，属于 Current 类型，容差代码为“current”。同一分支量的两个终端的性质必须相同，两个终端分别被称作正端和负端，分支量的方向是从正端指向负端。

分支量的容差由其子类型获得，但可以在定义子性质或定义分支量时被重载。例如在下面的代码中，vh1 和 ih1 从 t5,t6 处继承了容差代码，而 vh2 则被重新定义了容差代码。

```
-- 分支量的容差
subnature Highvoltage is Electrical
```

```
tolerance "highvoltage" across "highcurrent" through;
terminal t5,t6 : Highvoltage;
quantity vh1 across ih1 throuth t5 to t6;
quantity vh2 tolerance "voltage" across t5 to t6;
```

终端可以定义在信号定义能够出现的任意地方，同样也可以作为实体端口的一部分，例如下面的代码中就出现在端口定义中。

```
port(terminal anode,cathode : Electrical);
```

具有类似以上端口的模块在元件例化时，终端式端口就成为电路网络的节点，针对这些节点即可根据 Kirchhoff 定律(或其他守恒定律)建立方程组。

定义性质的同时可以形成一个参考终端，在电学中就是接地点，这一参考终端被具有该性质的所有终端所共享。任意性质 N 的某一终端 T 的参考终端记作 N'Reference。不仅如此，定义性质 N 的终端 T 时，还要产生两个量：

(1) 参考量 T'Reference，是分别以 T 和 N'Reference 作为正负终端的 across 量；

(2) 贡献量 T'Contribution，是一个 through 量，其数值等于所有连到 T 的 through 量之和(要考虑方向，即带符号相加)；

4. 同时语句(simultaneous statement)

在 VHDL 1076—1993 已有的并行语句和顺序语句基础上，VHDL 1076.1 增加了一种新的语言结构，即同时语句，用来描述微分和代数方程组。同时语句可以是各种普通的 VHDL 语句或表达式，但是对计算结果的解释和数据对象的更新机制则与过去的版本不同。同时语句可以出现在并行语句能够出现的任何地方，其标准格式如下：

[标号]：表达式 == 表达式；

例如欧姆定律可以书写为

```
I==V/R ;
```

其中表达式可以是有关量、信号、常量、共享标量及函数的任何计算。

VHDL 1076.1 还引入了三种额外的同时语句：

(1) 同时过程语句，用关键字 procedural 标志，这种语句使得使用者能够以顺序方式描述微分与代数方程组，在同时过程语句块中可以使用 VHDL 1076—1993 中除了 wait 和信号赋值之外的各种顺序语句；

(2) 同时 case 语句，以 case…when use…end use 为标志，各分支上可以出现各种同时语句，也能够嵌套地使用，只有有效条件分支上的同时语句才会被模拟求解器处理；

(3) 同时 if 语句，用 if use…elsif use…else…end use 为标志，各分支上可以出现各种同时语句，也能够嵌套地使用，只有有效条件分支上的同时语句才会被模拟求解器处理。

下面的代码是以上同时语句的例子。

```
-- 实体定义
entity Limiter is
        generic(gain : Real := 1.0; limit : Real);
```

```
        port(terminal INP,INM : Electrical;          ——输入终端
             terminal P,M : Electrical);             ——输出终端
end entity Limiter;

—— 使用同时 if 语句的结构体
architecture Simult of Limiter is
        quantity Vin across INP to INM;
        quantity V across i through P to M;
begin
        if gain * Vin > limit use
                V := limit;
        elsif gain * Vin < -limit use
                V := -limit;
        else
                V := gain * Vin;
        end use;
end architecture Simult;

—— 使用同时过程语句的结构体
architecture Proc of Limiter is
begin
procedural
                variable Vout : Voltage;
        begin
                Vout := gain * Vin;
                V := Vout;
                if Vout > limit then
                        V := limit;
                elsif Vout < -limit then
                        V := -limit;
                end if;
        end procedural;
end archtecture Proc;
```

在进行仿真时，由同时语句描述的方程、根据守恒定律形成的隐含方程(隐含是指不需要语言使用者书写代码)和所有的互连关系被映射为特征表达式(characteristic expression)。实际上，每一同时语句均会形成一组特征表达式，然后由模拟求解器用数值解法进行计算。当所有特征表达式的值均接近于 0 时，系统微分与代数方程组即被求解。类似于量，每一特征表达式属于特定的容差组。对于简单同时语句的特征表达式，如果等

号左边是某一量，则在默认情况下，该特征表达式的容差组与这个量相同；如果特征表达式不能表示成上述形式，而等号右边是一个量，则该特征表达式的容差组与这个量相同；如果上述两种情况均不满足，那么建模时要为该同时语句指定容差组，例如下面的例子：

```
I == V/R tolerance "tol1";
```

以上的同时语句被定义为属于容差组"tol1"。

5. 时间和仿真周期

在仿真过程中，模拟求解器与 VHDL 内核同步工作，这就要求仿真时间能够同时满足连续和离散仿真的需要，因此 VHDL 1076.1 引入了全局时间的概念，通过定义类型 Universal_Time 实现。Universal_Time 必须具备足够的精度来精确描述物理时间，即精度要等于或高于模型中浮点类型的精度。

仿真周期仍像 VHDL 1076—1993 一样由两个内核变量 Tc 和 Tn 表示，但这两个变量改为 Universal_Time 类型。Tc 表示现在的仿真时间，Tn 表示信号驱动器或进程下一次激活的时间。VHDL 仿真周期的概念有所修改，除了原有的 Δ 周期之外，还包括模拟求解器的执行时间。在每个仿真周期内，先进行离散仿真，在仿真时间增加之前模拟求解器工作。也就是说，模拟求解器在现在的数组时间和下一事件有效时间之间，以合适的间隔建立微分与代数方程组的解序列。VHDL 1076.1 的定义要求当某个量被进程进行读操作时，该量的数值必须正确（即已被求解）；同时当含有数字信号的表达式被求解时，该数字信号必须正确。

6. A/D 和 D/A

为了提供方便的 A/D 和 D/A 接口，VHDL 1076.1 定义了一种特殊机制：如果某个量的数值超过了预定义的阈值，则立刻停止方程的求解过程，即使此时下一次离散事件还未到来（如果未超过阈值，则方程的求解应该一直继续下去）。因此，VHDL 1076.1 为任意量 Q 定义了属性信号 Q'Above(level)，该信号为布尔类型，当 Q＞level 时，Q'Above(level)为 true；反之，为 false。当 Q—level 的符号改变时，Q'Above(level)上产生一次事件。下面代码是用这个属性信号描述 A/D 转换的例子。对 Q'Above(level)敏感的进程将在超越阈值电压时刻被激活，尽管这一时刻并不具备物理时间的意义。

```
-- 用 Q'Above(level)描述 A/D 转换
entity comparator is
    generic(level : Real := 2.5);
    port(terminal a : Electrical; signal S : out Bit);
end entity comparator;

architecture simple of comparator is
    quantity v across a;
begin
    S<='1' when V'Above(level) else '0';
```

```
end architecture simple;
```

从数学模型的角度来看，如果微分与代数方程组的解具有不连续性，则需要通知模拟求解器。因此 VHDL 1076.1 引入断点(break)机制，即模型中发生不连续时产生一次事件。由于这种事件不同于传统的离散事件，所以又称为伪事件(pseudo-event)。除了在其发生时显示地通知模拟求解器外，现在还没有统一而有效的办法处理这种不连续性。VHDL 1076.1 中 break 语句用来产生断点，有并行和顺序两种形式。break 语句要明确指出所谓的断点信号(break signal)，当断点信号的驱动器有效时，该信号上发生一次事件。break 语句也用于指定初始化条件以及在不连续情况出现时指定新的数值。下面的代码是用 break 语句描述 D/A 转换器的例子。

```
-- 用 break 语句描述 D/A 转换器
entity dac is
    generic(vhigh : Real := 5.0);
    port(signal S : in Bit; terminal a : Electrical);
end entity dac;

architecture simple of dac is
    quantity v across I through a;
begin
    if S = '0' use
        V == 0.0;
    else
        V == Vhigh;
    end use;
    break on S;
end architecture simple;
```

7. 初始化和静态工作点

对于连续系统来说，系统微分与代数方程组中未知数的求解依赖于初值，初值本身必然也是系统微分与代数方程组的解。也就是说，在连续系统的模型开始仿真之前或形成不连续点之后，必须有适当的算法对模型进行初始化。应该指出的是，由于在初始化阶段通常未知数(包括 x 和 dx/dt)的个数要多于方程个数，所以通常会有多个初值满足系统微分与代数方程组。因此，VHDL 1076.1 的使用者要根据实际情况选择适当的初值。初始化条件的指定用 break 语句实现，例如：

```
break q1=>expression1,q2=>expression2;
```

在 VHDL 1076.1 中，实际上有两个初始化过程，一个与原来 VHDL 1076—1993 的初始化类似，另一个则是求解连续系统的静态工作点。显然，在模拟设计中和频域仿真中，静态工作点的概念都是极其重要的。对于纯粹的连续系统(即模型中只有模拟电路的描

述，而没有任何数字电路描述）来说，静态工作点就是模型的直流工作点。而普遍意义上，静态工作点指仿真时间 0.0 时刻所有被挂起进程开始执行之前系统微分与代数方程组的解的状态。

VHDL 1076.1 预定义了枚举类型 DOMAIN_TYPE 的信号 DOMAIN，用于 VHDL 仿真内核对仿真的控制以及说明当前的工作状态。在初始化阶段，DOMAIN 的值为 INITILIZATION_DOMIAN；当已经计算得到静态工作点时，如果静态工作点是为时域仿真而解得的，则 DOMAIN 的值更新为 TIME_DOMAIN；如果静态工作点是为频域仿真而解得的，则 DOMAIN 的值更新为 FREQUENCY_DOMAIN。

8. 频域建模和仿真

模拟电路以及各种连续系统的设计中，频域分析是非常重要的，有些在时域很难描述的特性在频域却有简洁的描述方法，同时，频域分析有很多有效的算法。因此，VHDL 1076.1 在由系统微分与代数方程组导出的小信号模型基础上，定义了类似于 SPICE 的交流和噪声仿真。按照语言约定，当 DOMAIN 信号值为 FREQUENCY_DOMAIN 时，仿真器对(9.1)式作关于量的台劳展开，取线性项形成小信号模型。

描述小信号模型的方程组在没有特殊频域激励时，应该是齐次方程组。为描述特殊频域激励，VHDL 1076.1 定义两类源量（source quantity）：频谱源量（spectral source quantity）和噪声源量（noise source quantity），前者用于频域交流分析，后者用于噪声分析。源量必须用特定语言格式定义，其数据类型可以为混合类型。下面的例子描述了一个电流源，其数值保持恒定，通过频谱源量 q_ac 和噪声源量 q_ns 定义了频域和噪声激励。

```
-- 使用频谱源量和噪声源量的电流源模型
entity current_source is
    generic(dc,ac_mag,ac_phase,ns_mag : Real :=0.0);
    port(terminal P,M : electrical);
end entity current_source;

architecture simple of current_source is
    quantity i through P to M;
    quantity q_ac : Real spectrum ac_mag,ac_phase;    --定义频谱源量
    quantity q_ns : Real noise ns_mag;                --定义噪声源量
begin
    i == dc + q_ac + q_ns;
end architecture simple;
```

定义源量的幅度和定义频谱源量相位的表达式可以是任意合法的 VHDL 表达式，并且可以包含其他的量，例如定义和偏置电压有关的噪声。同时，这些表达式能够调用预定义函数 FREQUENCY 取得当前仿真频率值。应该指出的是，VHDL 1076.1 支持和特定频率有关的激励，但不支持依赖于特定频率的行为。

9. 其他语言特征

除了上述主要内容之外，VHDL 1076.1 还有一系列其他特征，即各种以属性形式隐含定义的量，这些量的主要目的是简化模型代码的书写。在本小节中，以下 Q 和 S 分别表示量和信号。

(1) Q'Delay(T)表示 Q 在固定时间间隔 T 之前的数值，其中 T 为 Real 类型的静态表达式，Q'Delay(T)的数据类型与 Q 相同。

(2) Q'Zoh(T,Initial_Delay)是在采样周期为 T、第一次采样发生在 Initial_Delay 秒时 Q 的零阶保持，T 和 Initial_Delay 都是实数类型的静态表达式。

(3) Q'Ltf(Num,Den)是对 Q 进行拉普拉斯变换后形成的量，即两个拉普拉斯变量 s 的多项式之比，其中 Num 是分子多项式的系数，Den 是分母多项式的系数，Num 和 Den 属于新定义的数据类型 Real_vector，即一维实数数组，数组元素可以是静态表达式。

(4) Q'Ztf(Num,Den,T,Initial_Delay)是对 Q 进行 Z 变换后形成的量，即两个以(z−1)为变量的多项式之比，其中 Num 是分子多项式的系数，Den 是分母多项式的系数，都为 Real_vector 数据类型，采样周期是 T，第一次采样发生在 Initial_Delay 秒，T 和 Initial_Delay 都是实数类型的静态表达式。

(5) Q'Slew(Max_Rising_Slope,Max_Falling_Slope)在数值上随 Q 变化，但其随时间的变化率由 Max_Rising_Slope 和 Max_Falling_Slope 约束，Max_Rising_Slope 和 Max_Falling_Slope 是 Real 类型的静态表达式，如果没有明确指出 Max_Falling_Slope，则默认 Max_Falling_Slope 的数值与 Max_Rising_Slope 相同，如果 Max_Rising_Slope 和 Max_Falling_Slope 都没有明确指定，则 Q'Slew 的数值和变化率都与 Q 完全相同。

(6) S'Slew(Max_Rising_Slope,Max_Falling_Slope)为浮点类型的量，随浮点信号 S 以 Max_Rising_Slope 和 Max_Falling_Slope 指定的最大斜率线性地变化，当 S 上发生新的事件时，S'Slew 开始变化。Max_Rising_Slope 和 Max_Falling_Slope 的数据类型及其他约定与 Q'Slew 中相同，S'Slew 的数据类型与 S 相同，但是其对象类型是量。

(7) S'Ramp(Tr,Tf)为浮点类型的量，随 S 线性变化，但是上升和下降时间为 Tr 和 Tf，Tr 和 Tf 为实数类型的静态表达式，如果 Tf 未明确指定，则默认 Tf 与 Tr 相等，如果 Tr 和 Tf 都未明确指定，则默认 Tr 和 Tf 为 0。

此外，VHDL 1076.1 的语言定义要求在每一设计实体(entity)内都要满足方程个数和未知数的个数相同，从而保证方程组能够被有效求解。

9.1.4 使用 VHDL 1076.1 的实例

1. 基本定义

在开始介绍用 VHDL 1076.1 描述连续系统之前，首先讨论电学和热学系统建模的基本定义，下面的代码是这两个系统建模所需要的程序包。其中第(a)部分中电学系统的基本定义实际上已经作过说明，只是为了简单起见，为电学参考终端定义了别名"ground"，即接地点。其中第(b)部分的热学系统基本定义完全类似于电学系统，根据表 9.1 定义了 across 和 through 量。

```
-- 电学和热学系统的基本定义程序包
-- (a)电学系统
package electrical_system is
        subtype Voltage is Real tolerance "voltage";
        subtype Current is Real tolerance "current";
        nature Electrical is Voltage across Current through;
        alias ground is Electrical'Reference;
        nature Electrical_vector is array (Natural range <>) of Electrical;
end package electrical_system;

-- (b)热学系统
package thermal_system is
        subtype Temperature is Real tolerance "temp";
        subtype Heatflow is Real tolerance "heatflow";
        nature Thermal is Temperature across Heatflow through;
        alias ground_th is Thermal'Reference;
        nature Thermal_vector is array (Natural range <>) of Thermal;
end package thermal_system;
```

在本小节 2 中,假定以上程序包存放在库(library)disciplines 中。

2. 淬火电路

淬火电路用于超导磁铁,图 9.1 是电路图。在磁铁处于超导状态时,其中电流达到几千安培,而磁铁两端电压为 0。当超导性消失时,磁铁两端电压会迅速增加,为避免电压过高损坏磁铁,设置两个分流二极管提供保护,装配在散热槽上,二极管的电流负载由换能器提供平衡。

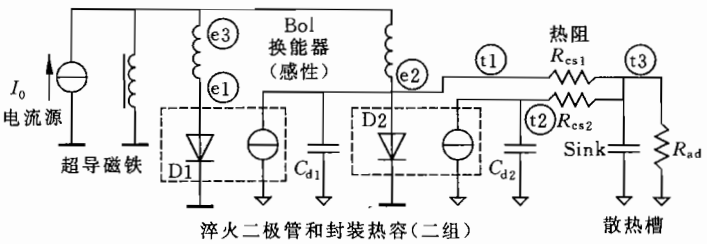


图 9.1 用于超导磁铁的淬火电路

(1) 电流源和电感

先从图 9.1 最左边的超导磁铁开始,介绍使用 VHDL 1076.1 构造连续系统模型的方法。超导磁铁在电性能上等价于并联的电流源和电感,这两种器件都是常用器件,下面的代码是它们的 VHDL 模型。这两端代码都是直接易懂的,电流源模型中只定义了 through 量,电感模型中定义了 across 和 through 量,两个模型都用一条同时语句描述了

器件的特征方程，从而描述了硬件行为。

```
——电流源和电感的 VHDL 1076.1 模型
—— 电流源模型
library disciplines;
use disciplines.electrical_system.all;
entity isrc is
    generic(Idc : Real);
    port(terminal P,M : Electrical);
end entity isrc;
architecture one of isrc is
    quantity I through P to M;
begin
    I := Idc;
end architecture one;

—— 电感模型
library disciplines;
use disciplines.electrical_system.all;
entity inductor is
    generic(inductance : Real);
    port(terminal P,M : Electrical);
end entity inductor;
architecture linear of inductor is
    quantity V across I through P to M;
begin
    V := inductance * I'Dot;
end architecture linear;
```

(2) N 匝理想换能器

接下来介绍 N 匝(即任意匝数)理想换能器的模型。为了突出重点，假定程序包 real_aux 已被编译到当前工作库中，该程序包中定义两维数组类型 Real_Matrix，并对运算符"*"进行了重载，用于实现两维数组与实数矢量的矩阵乘法。下面是换能器的 VHDL 模型。

```
—— 换能器的 VHDL 1076.1 模型
library disciplines;
use disciplines.electrical_system.all;
use work.real_aux.all;
entity xfrm is
```

```
    generic(Ml : Real_matrix);                    ——自导和互导系数
    port(terminal P,M : Electrical_vector);
end entity xfrm;
architecture one of xfrm is
    quantity V across I through P to M;
begin
    V := Ml * Real_vector(I'Dot);
end architecture one;
```

以上模型中定义了类数 Ml，这是一个二维矩阵，其元素为换能器的自导及互导系数。端口中 P 和 M 都是无界数组，在元件被例化时才确定数组长度。结构体的声明部分定义了 across 和 through 量 V 和 I，由于 P，M 是矢量类型，所以 V 和 I 也是矢量，并且具有相同长度。同时语句 V := Ml * Real_vector(I'Dot)描述了理想换能器的行为，其中对 I'Dot 进行了强制类型转换，这是因为重载后的"*"要求操作数为实数矢量，而 I'Dot 属于 Electrical_vector 类型。

(3) 热阻和热容

下面是热阻和热容的 VHDL 模型，显然它们完全类似于电阻和电容，只是性质属于热学系统。

```
—— 热阻和热容的 VHDL 1076.1 模型
—— 热阻模型
library disciplines;
use disciplines.thermal_system.all;
entity resistor_th is
    generic(R_th : Real);
    port(terminal P,M : Thermal);
end entity resistor_th;
architecture one of resistor_th is
    quantity temp across power through P to M;
begin
    assert R_th := 0.0;
    power := temp/R_th;
end architecture one;

—— 热容模型
library disciplines;
use disciplines.thermal_system.all;
entity capacitor_th is
    generic(C_th : Real);
```

```
    port(terminal P,M : Thermal);
end entity capacitor_th;
architecture one of capacitor_th is
    quantity temp across power through P to M;
begin
    power := C_th * temp'Dot;
end architecture one;
```

(4) 自加热电阻

淬火电路的最后一个器件是保护二极管，由于这里的二极管具有自加热功能，因此该模型由两个在二极管正极和负极之间的电学分支和一个在 PN 结和热学参考点之间的热学分支共同组成，如图 9.2。

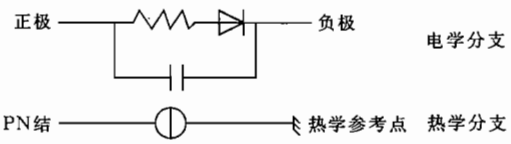


图 9.2 自加热二极管的电学和热学分支

对于电学分支来说，Voltage 和 Current 分别是 across 和 through 量；在热学分支中，对应的 across 和 through 量分别是 Temperature 和 Heatflow。下面的代码是这个二极管的 VHDL 模型。

```
-- 自加热二极管的 VHDL 1076.1 模型
library disciplines;
library IEEE;
use disciplines.electrical_system.all;
use disciplines.thermal_system.all;
use IEEE.math_real.all;

entity diode_th is
    generic(Is0 : Real := 1.0e-14;
            n,area : Real := 1.0;
            tau,Cj0,phi,Rd : Real := 0.0);
    port(terminal anode,cathode : Electrical;
         terminal junction : Thermal);
end entity diode_th;

architecture one of diode_th is
    quantity V across Id,Ic through anode to cathode;
```

```
    quantity Q : Charge;
    quantity Vt : Voltage;
    quantity temp across power through ground_th to junction;
    constant boltzmann : Real := 1.3806226e-23;
    constant electron_charge : Real := 1.602191e-19;
begin
    Id := area * Is0 * (exp((V-Rd * Id) / (n * Vt))-1.0);
    Q := tau * Id-2.0 * Cj0 * sqrt(phi * (phi-V));
    Ic := Q'Dot;
    Vt := temp * boltzmann / electron_charge;
    power := V * Id;
end architecture one;
```

结构体中定义了电学和热学分支。第一个同时语句说明阻性电学分支的电路，Id 依赖于热电压 Vt。第二和第三个同时语句描述了容性电学分支的电流。第四个语句描述 Vt 和温度的关系，最后一个同时语句定义热学分支的 rthrough 量。

(5) 测试程序

下面的代码是图 9.1 淬火电路系统的测试程序，描述了所有元件的互连。请注意平衡换能器的例化方法，使用这一方法可以实现二匝的换能器。

```
-- 淬火电路系统的测试程序
use disciplines.electrical_system.all;
use disciplines.thermal_system.all;
use work.real_aux.all;

entity quenching is
end entity quenching;

architecture test_bench of quenching is
    terminal e1,e2,e3 : Electrical;
    terminal t1,t2,t3 : Thermal;
    constant ml : Real_matrix(1 to 2,1 to 2) :=
                        ((1.0e-4,-1.0e-4),(-1.0e-4,10e-4));
begin
    I0: entity isrc(one) generic map(Idc=>18.0e3)
                port map(P=>ground,M=>e3);
    super1: entity inductor(linear) generic map(inductance=>0.03)
                port map(P=>e3,M=>ground);
    Ba1: entity xfrm(one) generic map(ml=>ml)
```

```
                port map(P(1)=>e3,M(1)=>e1,P(2)=>e3,M(2)=>
                  e2);
    D1: entity diode_th(one) generic map(Is0=>1.0e-6)
                port map(anode=>e1,cathode=>ground,junction=>
                  t1);
    D2: entity diode_th(one) generic map(Is0=>1.0e-6)
                port map(anode=>e2,cathode=>ground,junction=>
                  t2);
    Cd1: entity capacitor_th(one) generic map(C_th=>2.0e-5)
                port map(P=>t1,M=>ground_th);
    Cd2: entity capacitor_th(one) generic map(C_th=>2.0e-5)
                port map(P=>t2,M=>ground_th);
    Rcs1: entity resistor_th(one) generic map(R_th=>0.08)
                port map(P=>t1,M=>t3);
    Rcs2: entity resistor_th(one) generic map(R_th=>0.08)
                port map(P=>t2,M=>t3);
    Sink: entity capacitor_th(one) generic map(C_th=>1.0e-3)

    port map(P=>t3,M=>ground_th);
    Rad: entity resistor_th(one) generic map(R_th=>1.0e-4)
                  port map(P=>t3,M=>ground_th);
end architecture test_bench;
```

3. 反弹的球

现在描述一个封闭式系统，它是在具有无限弹性的地面上反弹的球，下面是其代码。这段代码反映了 VHDL 1076.1 的一些特征：① 使用同时 if 语句在两个简单同时语句之间切换；② 使用并行 break 语句指定初值；③ 使用隐含信号 Q'Above 检测球是否碰到地面；④ 使用并行 break 语句处理球碰到地面后造成的速度的不连续性。

```
-- 在地面上反弹的球的 VHDL 1076.1 模型
entity bouncer is
end entity bouncer;

architecture ball of bouncer is
    quantity V : Real tolerance "velocity";        --球的速度
    quantity Z : Real tolerance "elevation";       --球的位移
    constant G : Real := 9.81;
    constant AIR_RES : Real := 0.001;
begin
```

```
    Z'Dot := V;
    if V > 0.0 use
        V'Dot := -G-V * * 2*AIR_RES;
    else
        V'Dot := -G+V * * 2*AIR_RES;
    end use;
    break V=>-V when not Z'Above(0.0);
    break V=>0.0,Z=>10.0;
end architecture ball;
```

描述系统的方程就是牛顿运动方程，重力将球拉向地面，而空气阻力总是阻碍运动。当球碰到地面时，由于假定地面具有无限弹性，所以球的速度立刻改变方向，即 V 的符号改变，此时 V 的解存在不连续性。上面的模型用隐含信号 Q'Above 检测这个碰撞，当 Z 从地面上方落下到高度 0.0 时，not Z'Above(0.0)变为 true。这时执行第一条 break 语句，把下一求解时间间隔时的速度置为－V。第二条 break 语句描述系统的初始条件，即静态工作点。

4. 可控硅整流器

可控硅整流器(SCR, silicon controlled rectifier)是常用的电子器件，下面是其 VHDL 模型。在正向偏置且控制电压高于开启电压 von 时，SCR 打开；当 SCR 中的电流小于保持电流且控制电压低于开启电压时，SCR 关闭。在模型中，这一条件由信号 off 表示。流过 SCR 的电流的二极管效应可以用两段折线近似，当 SCR 开启并且支路电压高于 vdrop 时，电流流过 SCR。这个条件由信号 zero_current 表示，并由它控制两个简单同时语句。break 语句声明当 zero_current 改变数值时发生的不连续性。

```
-- 可控硅整流器的 VHDL 1076.1 模型
library disciplines;
use disciplines.electrical_system.all;

entity scr is
        generic(vdrop : Voltage := 0.7;          --开启电压降幅
                von : Voltage := 0.7;            --开启电压
                ihold : Current := 0.0;          --保持电流
                ron : Real := 0.1e-9);           --开启电阻
        port(terminal anode,cathode,gate: Electrical);
end entity scr;

architecture ideal of scr is
        quantity vscr across iscr through anode to cathode;
        quantity vcontrol across gate to cathode;
```

```
        signal OFF,ZERO_CURRENT : Boolean := true;
begin
        assert ron /= 0.0;
        OFF<=true when not (vcontrol'Above(von) or iscr'Above(ihold)) else
                flase when vcontrol'Above(von) and vscr'Above(0.0);
        ZERO_CURRENT<=off or not vscr'Above(vdrop);
        if ZERO_CURRENT use
            iscr := 0.0;
        else
            iscr := (vscr-vdrop) / ron;
        end use;
        break on ZERO_CURRENT;
end architecture ideal;
```

9.2 VHDL 在 VLSI 测试中的应用

大多数 VHDL 工具集中于数字集成电路的设计描述、仿真和综合，但不少研究机构一直致力于把 VHDL 的应用扩展到集成电路设计的另一重要领域，即测试领域。目前，VHDL 在测试方面主要有如下应用：

(1) 描述测试矢量，即用于验证器件功能和性能的激励与输出，这方面主要的工作是由 IEEE 修订的 WAVES 完成；

(2) 描述测试电路，如扫描路径(scan path)、内建自测试(built-in self-test，常简称 BIST)、边界扫描(boundary scan)等电路结构，当然对于描述扫描路径和 BIST 来说，与描述其他数字电路元件并没有本质的区别，这里主要介绍用 VHDL 语言描述边界扫描器件；

(3) 描述故障模型和故障插入；

(4) 对行为级 VHDL 进行可测性分析以及行为级测试结构插入；

(5) 对寄存器级或门级的 VHDL 网单进行测试结构插入，例如在 UltraSPARC 芯片的设计过程中，在门级用半自动化的方式把可测试电路插入到用硬件描述语言描述的网单中，当然这方面的应用只是把 VHDL 作为一种信息载体，这里不再具体介绍。

9.2.1 波形描述语言 WAVES

1. 简介

WAVES 是波形和矢量交换规范(waveform and vector exchange specification)的缩写，是 VHDL 语言的一个子集，主要用于描述同时包括测试激励和测试输出在内的测试矢量。WAVES 是在 IEEE 标准 VHDL 1076—1987 的基础上由 IEEE 计算机协会设计自动化标准子委员会和标准协调委员会的 SCC—20 工作组共同发展的，其中标准协调委员会的主要工作方针是发展跨领域标准，SCC—20 工作组的责任是制定测试标准，其主要

成果是 Atlas 测试语言；而设计自动化标准子委员会的主要成果是制定 VHDL 语言。

WAVES 的设计目的是为波形描述的交换提供开放式的环境和通用的格式，从而能够描述逻辑信号的整个变化过程，使得设计和测试成为更加紧密耦合的两个过程。图 9.3 说明了 WAVES 在设计和测试中的作用。

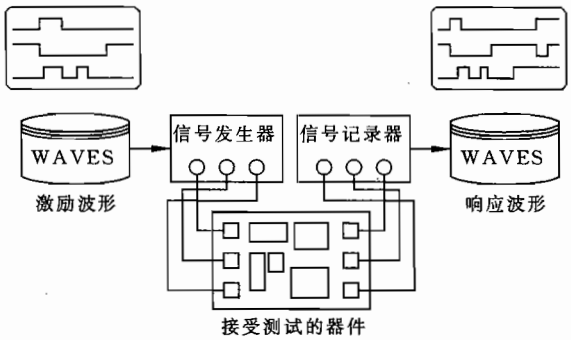


图 9.3　WAVES 在设计和测试中的作用

从图 9.3 可以看到，作为一种测试矢量描述语言，WAVES 为接受测试的器件描述输入和输出波形。被测试的器件可以是在仿真器上运行的 VHDL 模型，也可以是自动测试设备(ATE，automatic test equipment)中的物理硬件，或者是二者的组合。实际上，类似于 VHDL 在不同仿真器之间的作用，WAVES 相当于在不同测试设备之间提供了交换信息的能力。如果 ATE 制造商为其测试仪器配备 WAVES 接口，则用 WAVES 描述的测试数据(也称作 WAVES 数据集合，即 WAVES data set)可以用于不同的测试设备。

不仅如此，WAVES 的开放式框架极大地改善了设计师(这里设计师包括电路设计师和测试设计师)再用设计信息的能力。芯片设计师可以通过 WAVES 再用逻辑仿真器提供的信号历史，而测试设计师可以再用器件仿真获得的信号历史。特别是在技术更新之后，设计师就能够把测试仪器上获得的旧的电路的信号波形施加于新的电路。

WAVES 数据集合是 WAVES 交换信息的基本单位，可以由几个文本文件组成。其描述分为两个层次，一个层次中波形以压缩编码形式描述，另一个层次是以标准化的数据类型描述的编码方式。不同的 WAVES 数据集合可以包含不同类型的信号信息，并能够使用不同的信息表示方式。整个 WAVES 数据集合以 VHDL 语言表示，也可以使用兼容于 VHDL 文件 I/O 的 ASCII 文本文件，当然 WAVES 数据集合要比 VHDL 全集简单得多。因此在任何支持标准 VHDL 的环境中，WAVES 不需要进一步的翻译就可以使用。

2. WAVES 的事件-数值模型

WAVES 在记录信号历史时，主要通过所谓事件-数值模型(event_value model)来实现。这个模型包含足够的信息描述或规定信号值。下面将较为详细地介绍事件-数值模型。

(1) 波形的分解

在 WAVES 中，波形指一定接口上的一组信号在某个确定时间间隔内的逻辑状态，如图 9.4。

WAVES 通过事件、帧和时间片来描述波形。完整的波形首先被分解为一系列连续的

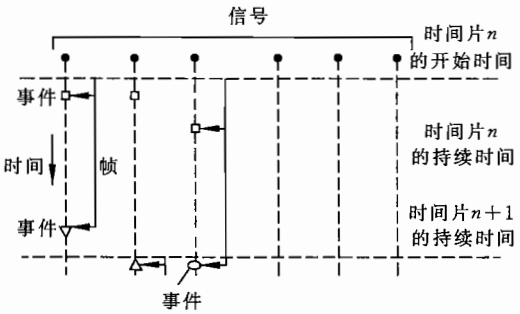


图 9.4 WAVES 的波形

时间片，时间片的持续时间成为周期。每一时间片内单个信号的波形称作帧，一帧内可以包含该信号的若干个事件(允许没有事件)。事件包括事件值(event value)和事件时间(event time)两部分。事件值是电路状态的描述，在事件发生后有效，直至该信号上发生下一次事件为止。事件时间记录时间片的开始时刻与事件发生时刻之间的时间间隔。以上的分解构成 WAVES 描述信号历史的基础，但是实际上 WAVES 描述通常并不只是一系列时间片，而要通过建立在这个模型之上的编码和其他压缩机制来描述信号。比如，为了实现诸如时钟脉冲和回 0 格式(return-to-zero format)等测试仪的标准行为，在 WAVES 中可以使用探针码来定义扩展到帧的测试矢量，而不必以一系列事件值的方式定义。

(2) 事件值

事件值是一个四元组，即由四个元素组成：状态(state)、驱动能力(strength)、方向(direction)和相关度(relevancy)。状态以布尔或二进制数的方式描述信号电平，驱动能力表示信号源在电平竞争(即某一信号线被不同电平的若干个源同时驱动)或不同负载条件下维持该状态的能力，以上两个元素均来源于芯片设计师的术语，其含义也与多值逻辑的相应定义相同。方向和相关度这两个概念来源于测试工程师的术语，方向表示该事件是激励还是响应，即信号源是在接口的内部还是外部；相关度表示该事件的正确性对于器件正确工作的重要程度，即某一事件值是要求的还是仅属于观察之列。以上四个元素均属于枚举类型，其定义如下面的代码所示。

```
-- 事件值四个元素的类型定义
type state is (unspecified,unknown,low,midband,high);
type strength is (unspecified,unknown,disconnected,capacitive,resistive,
drive,supply);
type direction is (unspecified,unknown,stimulus,response,compound);
type relevance is (unspecified,unknown,mask,care);
```

下面的代码是组合四个元素形成事件值的函数。例如，事件值 drive_1 是要求(relevance=care)的激励(direction=stimulus)，信号电平为高(state=high)，驱动能力属于 drive 级(strength=drive)。drive_1 是相当确定的事件值，而事件值 dont_care 则要不确定得多，也就是说涵盖的范围更宽，这里状态和驱动能力都是 unspecified，只知道是响

应信号(direction=response),观察的信号值也不影响器件整体的正确行为(relevance=mask)。

```
-- WAVES 的 value_dictionary 函数
function value_dictionary (value : in logic_value) return event_value is
begin
    case value is
        when drive_0=>return evalue(low,drive,stimulus,care);
        when drive_1=>return evalue(high,drive,stimulus,care);
        when tristate=>return evalue(midband,resistive,stimulus,care);
        when uninitialized=>return evalue;
        when sense_0=>return evalue(low,drive,response,care);
        when sense_1=>return evalue(high,drive,response,care);
        when sense_Z=>return evalue(midband,resistive,response,care);
        when dont_care=>return evalue(unknown,unknown,response,mask);
    end case;
end value_dictionary;
```

WAVES 中的事件值总是在一定的接口上观察信号得到的,这里接口内部就是接受测试的器件,接口外部是整个测试环境。因此,当 direction 的值是 stimulus 或 response 时表示的意义是直截了当的;然而当 direction 的值为 compound 时,并不仅仅意味着该信号是双向端口,而是表示接口外部存在单独的或潜在的驱动信息,例如测试开集输出(open-collector output)的情形。

(3) 波形编码和帧

在一般的测试矢量描述语言中都用特殊符号来表示时钟脉冲,这实际上就是一种波形压缩编码。同时,像在 0 包围(surround-by-zero)或互补包围(surround-by-complement)等测试格式中一样,固定的保护值通常要由不同的测试值包围,这当然也是一种压缩编码。在 WAVES 中,由于波形描述编码在帧级而不是事件级进行,所以以上处理变得更加方便。WAVES 可以把帧序列或事件序列定义为一个编码字母表(因为类似于现有的测试仪编码格式,又称作探针码 pin code),并允许有多种翻译方式。与编码对应的时序信息可以被存放在另外的文件中,这样如果要改变现有测试矢量的时序,仍然可以使用原来的波形编码描述。

WAVES 的大部分波形描述通过对每个时间片或每个矢量使用一组探针码来编码,即探针—帧的对应编码。下面的代码描述了一种归 0 测试仪格式。其中,R0_frame_set 是用 VHDL 的 WAVES 子集书写的用户函数。该函数使用了两个 WAVES 预定义函数:frame_event 根据一个事件值-时间对(event_value,event_time)构成一帧,frame_elist 接受一个事件值-时间对(event_value,event_time)序列并返回一帧。使用重载运算符"+"能够把一组帧合并为帧集合。在本例中,探针码被映射到事件级。

```
-- 用 WAVES 描述的测试矢量
```

```
function R0_frame_set (
——施加激励的时刻
transition_1   : event_time;
transition_1   : event_time;

——观察响应时刻
strobe_1   : event_time;
strobe_2   : event_time;

    return frame_set is
        constant sense_off     : event := (dont_care,strobe_2);
        constant return_zero   : event := (drive_0,transition_2);
begin
    return
        frame_event ((drive_0,transition_1))+
        ——探针码 '0'
        frame_elist (((drive_1,transition_1),return_zero))+
        ——探针码 '1'
        frame_elist (((tristate,transition_1),return_zero))+
        ——探针码 'T'
        frame_elist (((uninitialized,transition_1),return_zero))+
        ——探针码 'U'
        frame_elist (((sense_0,strobe_1),sense_off))+
        ——探针码 'L'
        frame_elist (((sense_1,strobe_1),sense_off))+
        ——探针码 'H'
        frame_elist (((sense_Z,strobe_1),sense_off))+
        ——探针码 'Z'
        frame_event((dont_care,strobe_1));
        ——探针码 'X'
end R0_frame_set;
```

3. 使用 WAVES 描述测试矢量的实例

下面利用一个具体的例子说明如何用 WAVES 描述测试矢量。被测电路是一个 BCD 码环形计数器，即从 0 开始计数到 9 后，电路复位为 0，重新开始一次计数过程。激励和响应的波形如图 9.5，其中已经按照 WAVES 概念进行了分解。图中是三个时钟周期的波形，电路输出是 4 位信号 COUNT，其中 COUNT(3)是最高位。因此，输出值依次是 0000，0001，0010 和 0011。

按照 WAVES 的定义，图 9.5 中各个信号的变化经历统称为波形。波形由一系列时

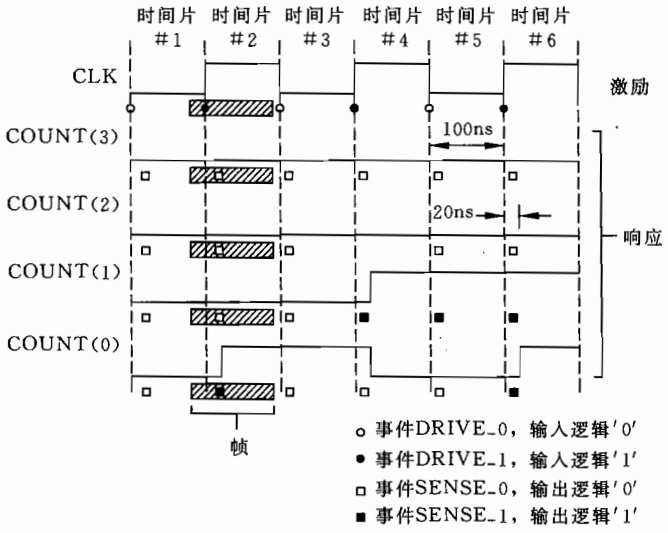


图 9.5　BCD 计数器的激励和响应

间片组成，时间片的长度由用户定义，本例中是半个时钟周期，即 100ns。时间片被分解为一系列帧。对于一个信号来说，在一个时间片内发生的所有事件就构成一帧。

通过执行下面的 WAVES 过程 BCD_TEST 来描述图 9.5 的波形。在这个过程中，通过 6 次调用 WAVES 预定义过程 apply 来逐个时间片地构造 BCD 计数器的测试矢量。Apply 的第一个入口参数 BCD_WAVEFORM 说明存储以该参数为标志符的波形，第三个参数 HALF_CYCLE 说明每个时间片的持续时间，在 BCD_TEST 的起始部分已经把 HALF_CYCLE 定义为 WAVES 的 delay_time 预定义类型，数值是 100ns。构成特定时间片每一帧的事件由 apply 构成的第二和第四个参数定义，分别表示探针码和时间数据对象。实际上，探针码是表示帧的简写式记号，在本例中由于波形比较简单，每一帧内都只有一个事件，采用探针码来表示事件。探针码和事件的对应关系如表 9.2 所示。

```
-- 为 BCD 计数器生成 WAVES 波形的过程
procedure BCD_TEST (signal BCD_WAVEFORM : inout waves_port_list) is
        constant FSF : frame_set :=
            frame_event ((DRIVE_0,etime(0 ns))) +
            frame_event ((DRIVE_1,etime(0 ns))) +
        frame_event ((SENSE_0,etime(20 ns))) +
        frame_event ((SENSE_1,etime(20 ns)));
    constant FSA : frame_set_array := new_frame_set_array(FS,all_pins);
    constant FRAMES_DEF : time_data : new_time_data(FSA);
    constant HALF_CYCLE := delay_time := delay(100 ns);
begin
    apply(BCD_WAVEFORM,"0LLLL",HALF_CYCLE,FRAME_DEF);
```

```
    --slice #1
    apply(BCD_WAVEFORM,"1LLLH",HALF_CYCLE,FRAME_DEF);
    --slice #2
    apply(BCD_WAVEFORM,"0LLLH",HALF_CYCLE,FRAME_DEF);
    --slice #3
    apply(BCD_WAVEFORM,"1LLHL",HALF_CYCLE,FRAME_DEF);
    --slice #4
    apply(BCD_WAVEFORM,"0LLHL",HALF_CYCLE,FRAME_DEF);
    --slice #5
    apply(BCD_WAVEFORM,"1LLHH",HALF_CYCLE,FRAME_DEF);
    --slice #6
end BCD_TEST;
```

表 9.2 BCD 计数器测试矢量中探针码和事件的对应关系

探针码	事　件	
	事件值	事件时间/ns
0	DRIVE_0	0
1	DRIVE_1	0
L	SENSE_0	20
H	SENSE_1	20

在我们的讨论中，事件值 DRIVE_0 表示电路状态为低电平，方向为激励；事件值 SENSE_1 表示电路状态为高电平，方向为响应；依此类推。通过把每一探针码联系到一个称为测试探针的信号上，就实现了把用探针码字符串定义的事件分配到整个波形中。在本例中，探针码字符串最左边的字符对应探针 CLK，即 CLK 信号；第二个字符对应 COUNT(3)，等等。限于篇幅，这里不再说明如何建立探针码和特定事件的映射关系。

接下来通过 BCD_TEST 过程的执行来说明 BCD 计数器的测试矢量是如何依次生成的。开始执行时，波形 BCD_WAVEFORM 起始于时间 $t_{now}=0$。第一个 apply 过程用探针码"0LLLL"(对应于图 9.5 中时间片 1)，来初始化波形中的各个信号。也就是说，一个 DRIVE_0 事件被调度在 $t_{now}+0$ ns$=t_{now}$时刻，发生在信号 CLK 上。四个 SENSE_0 事件将在 t_{now}后 20ns 发生在信号 COUNT(3)到 COUNT(0)上。在时间片 1 的所有事件均得到调度之后，WAVES 时间前进 HALF_CYCLE，调用下一个 apply 过程，产生时间片 2 的事件。

9.2.2 边界扫描描述语言

边界扫描(boundary scan)测试是由 JTAG(joint test action group，1985 年由欧洲和北美众多电子公司共同发起的一个工作委员会)开发的适用于数字芯片和电路板的测试技术，在 1990 年成为 IEEE 的 1149.1—1190 号标准，并在实际的设计中得到了广泛的应

用。由于该标准具有较大的灵活性，并支持用户定义的特征，因此能够极大地增强数字系统的可测性，许多影响可测性设计的障碍都能够被边界扫描技术克服。但是另一方面，边界扫描的丰富性和开放性也造成在使用上的复杂性。其中一个首要的问题是如何在标准的框架下描述器件，因为器件含有边界扫描部分，因此必须对其边界扫描部分进行某种形式的描述。本节介绍一种在 VHDL 语言基础上发展的边界扫描描述语言(boundary scan description language)，简称 BSDL。

在下面的讨论中，假定读者已经具备边界扫描的知识，不熟悉的读者请参考 IEEE 的 1149.1—1190 号标准手册或有关参考文献。BSDL 是 IEEE 标准 VHDL 1076—1990 的一个子集，由 HP 公司开发。在设计该语言时，要求满足两个特征，即首先应该易于用户使用，其次要能够简单而清晰地被计算机处理。

通过使用 BSDL 描述可测性特征，设计中违反标准的特征能够在建模过程中被设计师或语法检查程序尽早发现。同时，BSDL 为 IC 制造商和自动测试设备制造商提供了一种标准的交流方式。在将来的应用中，还可以结合器件的 BSDL 描述和系统电路的 VHDL 描述进行边界扫描电路的自动插入。

1. BSDL 语言的描述范围

BSDL 语言用于描述与 IEEE 标准 1149.1—1190 兼容的器件的可测性特征，并被利用这些特征的 EDA 工具所使用。这里的 EDA 工具包括可测性分析、测试生成和故障分析等。应该指出的是，BSDL 不是通用的硬件描述语言，其功能在于通过 BSDL 的器件描述和 1149.1—1190 标准的规定，使 EDA 工具能够理解器件的特征数据。虽然通过结合 VHDL 代码建立模型，将 BSDL 描述包含在 VHDL 模型之中，就可以对模型进行仿真、验证、综合和兼容性分析等，但这些特征不属于 BSDL 的描述范围。

BSDL 的另一特点是语言本身与标准的正交性，即标准指定的特征不再由 BSDL 描述。例如，BSDL 不描述 BYPASS 寄存器，因为标准本身已经完全确定了该寄存器的特征。实际上，BSDL 描述的是标准中的可选特征和用户自定义特征。

2. 边界扫描特征

所有支持边界扫描的器件都具备两个基本特征：测试接口(TAP，test access port)和边界寄存器(BR，boundary register)。BSDL 即用来描述以上两个特征的参数。并、串行边界寄存器由与器件输入、输出、双向信号以及特定的嵌入控制信号相连的边界单元组成，大部分 1149.1—1190 号标准的灵活性就体现在边界寄存器上。TAP 包含一组专用信号，即测试时钟 TCK、测试模式选择 TMS、测试数据输入 TDI、测试数据输出 TDO 以及可选的测试复位信号 TRST。TAP 还要包含一个指令寄存器和一个 BYPASS 寄存器。TAP 通过一组很少的指令来实现对边界扫描功能的控制，特定 TAP 信号和指令下的操作在标准中有严格的定义。同时，TAP 可以含有可选的数据寄存器和指令以及用户定义的数据寄存器和指令。

实际上，可以把边界扫描器件视为具有与外界连接的黑箱，黑箱内部是由边界寄存器包围的 TAP 和系统逻辑。BSDL 的作用是，在不描述系统逻辑的条件下描述边界寄存器和外部连接。

3. BSDL 语言

BSDL 是 VHDL 的子集，关键字和语言格式均与 VHDL 相同，"—"后面的文字同样被视为注释。相当一部分信息通过 VHDL 属性描述，这些属性一般表示为字符串。

BSDL 描述由三部分组成：entity，package 和 package body。entity 内除了 VHDL 原有的各种说明之外，还包括器件的边界扫描参数。BSDL 有预定义 package，其内容与标准直接相关，因此只有在标准更新的情况下才有必要对之进行修改。用户可以书写自定义的 package 和 package body，其中一般是对边界扫描单元库的描述。下面将结合一个例子说明如何用 BSDL 语言描述边界扫描器件，该器件是 TI 生产的 8 位 D 触发器 74bc8374，插入边界扫描单元后的电路如图 9.6 所示，BSDL 描述如后面的实体 ttl74bc8374 所示。限于篇幅，这里只介绍核心的 entity 部分。

BSDL 描述器件的 I/O 端口和重要的边界扫描特征，entity 的结构如下：

```
entity My_IC is              --描述器件 My_IC 的 entity
    [generic parameter]                --类属参数定义
    [logical port description]         --端口声明
    [use statement(s)]                 --程序包声明
    [package pin mapping]              --封装管脚映射关系
    [scan port identification]         --扫描端口标志
    [TAP description]                  --TAP 描述
    [Boundary Register description]    --边界寄存器描述
end My_IC;
```

对 entity My_IC 说明如下：

(1) generic 参数是 VHDL 向模型传递数据的一种手段，BSDL 中除了一般的 VHDL generic 参数之外，还使用它来选择器件的封装形式。对于不同的封装形式，芯片压焊盘与封装管脚的映射关系通常也是不同的。而在电路板级测试时，必须知道芯片的逻辑结构怎样被映射到一组物理管脚上。BSDL 中的 generic 参数即服务于这一目的。一种常见的用法是：

```
generic (PHYSICAL_PIN_MAP : string := "UNDEFINED");
```

(2) 端口声明部分的格式与 VHDL 完全相同，只是在端口信号的 I/O 模式方面，增加了一种新的类型 linkage，用于说明电源管脚。下面是端口声明的例子：

```
port(CLK : in Bit; Q : buffer Bit_vector(1 to 8); D : in Bit_vector(1 to 8);
     VCC,GND : linkage Bit);
```

(3) use 语句声明所使用的程序包，BSDL 所需的边界扫描标准以及各种属性、常数、类型等的定义存放在程序包 STD_1149_1_1190 中。

(4) 描述管脚与器件 I/O 端口的映射关系，属性 PIN_MAP 从类属参数中获得封装类型信息，映射关系即由与 PIN_MAP 同名的常数确定。例如在下面的代码中，如果传递到模型中的 generic 参数 PHYSICAL_PIN_MAP 是 DW_PACKAGE，则该器件的封装

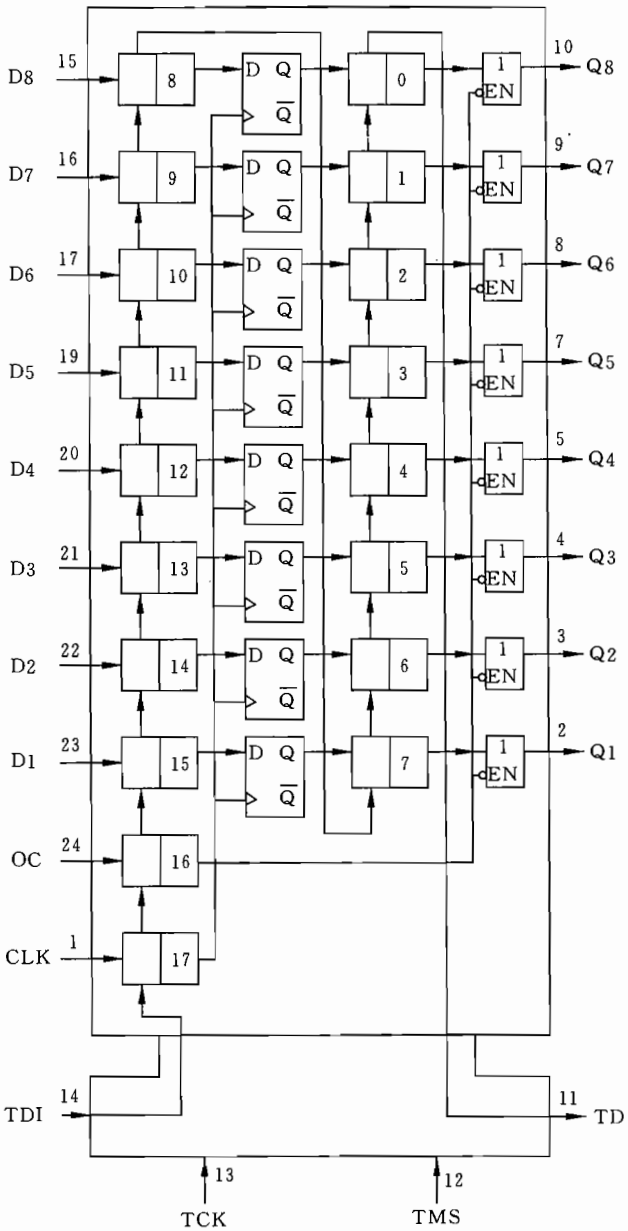


图 9.6　8 位 D 触发器 74bc8374

映射关系定义如下：

"CLK：1,Q(2,3,4,5,7,8,9,10),D：(23,22,21,20,19,17,16,15)" &
"GND：6,VCC：18,OC_NEG：24,TDO：11,TMS：12,TCK：13,TDI：14"

这就是说，逻辑端口 CLK 连到管脚 1,Q(1 to 8)连到 2,3,4,5,7,8,9,10 号管脚，依此类推。

(5) 在扫描端口标志部分,BSDL 用属性定义器件的 TAP 端口,在 ttl74bc8374 的源代码中,attribute TAP_SCAN_IN of TDI : signal is TRUE 语句说明逻辑端口 TDI 就是 IEEE 1149.1—1190 标准中的扫描输入端口 TDI。

(6) 在 TAP 描述部分,BSDL 首先说明 TAP 指令寄存器,属性 INSTRUCTION_LENGTH 指出测试指令的长度,INSTRUCTION_CODE 依次说明各条标准测试指令和用户定义指令的机器码。例如在下面的 ttl74bc8374 的源代码中,标准测试指令 EXTEST 被定义为机器码 00000000 或 10000000。即指令寄存器中装入 00000000 或 10000000 时,器件执行 EXTEST 指令。属性 INSTRUCTION_CAPTURE 和 INSTRUCTION_DISABLE 与边界扫描控制电路(该电路是一有限状态机电路)状态有关,这里不再详细介绍。属性 REGISTER_ACCESS 用于说明具体指令需要访问的寄存器,在 ttl74bc8374 的代码中,BYPASS 寄存器能够且只能被 TOPHIP,SETBYP,RUNT,TRIBYP 这 4 条指令访问。位宽与指令长度不同的寄存器要指出其具体位宽,如 BCR[2]说明 BCR 寄存器位宽为 2。

(7) 边界寄存器描述直接说明边界扫描测试所需要的测试硬件。属性 BOUNDARY_CELLS 说明所使用边界单元的类别,在 ttl74bc8374 例中使用 BC_1,其中该边界单元必须已在程序包中定义过;属性 BOUNDARY_LENGTH 说明所有边界单元连成的边界扫描移位寄存器的长度,在该例中为 18;属性 BOUNDARY_REGISTER 具体说明该移位寄存器每一位与逻辑端口的连接关系,其中每一位的格式为

边界单元编号(cell,port,function,safe,ccell,disval,rslt)

前 4 个域是必需的,后 3 个域可选。cell 域说明边界单元类型,port 就是器件的逻辑端口,以上两个域结合就说明了某端口连到何种类型的边界单元上;function 域说明该边界单元的功能,BSDL 预定义的功能域值及其含义如表 9.3。safe 域表示该单元的输出处于何种状态时不会影响整个电路,取值为 0,1 或 X,这里 X 表示无关态。ccell 域是控制该单元输出的使能信号所使用的单元编号;disval 是 ccell 使该输出无效时的取值;rslt 指明在使能无效时该单元的输出值,可以是高阻态 Z、弱 1 或弱 0。只有在 function 域为 output2,output3 和 bidir 时,后 4 个域才有意义。

表 9.3 BSDL 预定义的功能域值及其含义

功能域值	含 义
input	普通输入
clock	时钟输入
output2	普通输出(输出为高电平或低电平)
output3	三态输出
control	三态驱动或 I/O 方向控制
controlr	在 TAP 控制器处于 Test_Logic_Reset 状态时无效,其余与 control 相同
internal	存放内部常数
bidir	双向
*	该单元是内部单元或输出控制

```
-- 用 BSDL 描述 8 位 D 触发器 74bc8374
entity ttl74bc8374 is
        generic (PHYSICAL_PIN_MAP: string :="UNDEFINED");
                                        --类属参数定义
        port(CLK:in bit;Q:out bit_vector(1 to 8);D:in bit_vector(1 to 8);
            GND,VCC : linkage bit;
            OC_NEG : in bit; TDO : out bit; TMS,TDI,TCK : in bit);
            --端口声明

        use STD_1149_1_1190.all;            --程序包声明
        ------------------------------------------------------------
        --封装管脚映射关系
        attribute PIN_MAP of ttl74bc8374 : entity is PHYSICAL_PIN_MAP;

        constant DW_PACKAGE : PIN_MAP_STRING := "CLK: 1," &
                Q(2,3,4,5,7,8,9,10),D: (23,22,21,20,19,17,16,15)," &
                "GND: 6,VCC: 18,OC_NEG: 24,TDO: 11,TMS: 12," &
                "TCK: 13,TDI: 14";
        constant FK_PACKAGE : PIN_MAP_STRING := "CLK: 9," &
                Q(10,11,12,13,16,17,18,19),D: (6,5,4,3,2,27,26,25)," &
                "GND: 14,VCC: 28,OC_NEG: 7,TDO: 20,TMS: 21," &
                "TCK: 23,TDI: 24";
        ------------------------------------------------------------
        --扫描端口标志
        attribute TAP_SCAN_IN of TDI : signal is TRUE;
        attribute TAP_SCAN_MODE of TMS : signal is TRUE;
        attribute TAP_SCAN_OUT of TDO : signal is TRUE;
        attribute TAP_SCAN_CLOCK of TCK : signal is TRUE;
        ------------------------------------------------------------
        --TAP 描述
        attribute INSTRUCTION_LENGTH of ttl74bc8374 : entity is 8;
        attribute INSTRUCTION_OPCODE of ttl74bc8374 : entity is
            "BYPASS (11111111,10001000,00000101,10000100,00000001),"&
            "EXTEST (00000000,10000000)," &
            "SAMPLE (00000010,10000010)," &
            "INTEST (00000011,10000011)," &
            "TRIBYP (00000110,10000110)," &
            "SETBYP (00000111,10000111)," &
```

```
    "RUNT (00001001,10001001)," &
    "READBN (00001010,10001010)," &
    "CELLTST (00001100,10001100)," &
    "TOPHIP (00001101,10001101)," &
    "SCANCN (00001110,10001110)," &
    "SCANCT (00001111,10001111)";

attribute INSTRUCTION_CAPTURE of ttl74bc8374:entity is
  "10000001";
attribute INSTRUCTION_DISABLE of ttl74bc8374:entity is"TRIBYP";

attribute REGISTER_ACCESS of ttl74bc8374 : entity is
    "BOUNDARY (READBN,READBT,CELLTST)," &
    "BYPASS (TOPHIP,SETBYP,RUNT,TRIBYP)," &
    "BCR[2] (SCANCN,SCANCT)";        --2位边界控制寄存器
-------------------------------------------------------------
--边界寄存器描述
attribute BOUNDARY_CELLS of ttl74bc8374 : entity is "BC_1";
attribute BOUNDARY_LENGTH of ttl74bc8374 : entity is 18;

attribute BOUNDARY_REGISTER of ttl74bc8374 : entity is
    "17 (BC_1,CLK,input,X)," &
    "16 (BC_1,OC_NEG,input,X)," &
    "16 (BC_1, * ,control,1)," &
    "15 (BC_1,D(1),input,X)," &
    "14 (BC_1,D(2),input,X)," &
    "13 (BC_1,D(3),input,X)," &
    "12 (BC_1,D(4),input,X)," &
    "11 (BC_1,D(5),input,X)," &
    "10 (BC_1,D(6),input,X)," &
    "9 (BC_1,D(7),input,X)," &
    "8 (BC_1,D(8),input,X)," &
    "7 (BC_1,Q(1),output3,X,16,1,Z)," &
    "6 (BC_1,Q(2),output3,X,16,1,Z)," &
    "5 (BC_1,Q(3),output3,X,16,1,Z)," &
    "4 (BC_1,Q(4),output3,X,16,1,Z)," &
    "3 (BC_1,Q(5),output3,X,16,1,Z)," &
    "2 (BC_1,Q(6),output3,X,16,1,Z)," &
```

```
        "1 (BC_1,Q(7),output3,X,16,1,Z)," &
        "0 (BC_1,Q(8),output3,X,16,1,Z)";
end ttl74bc8374;
```

9.3 性能模型

在数字系统设计的开始阶段，设计师经常需要建立系统的性能模型(performance model)，然后用这个模型进行性能仿真，这一步骤也常称作性能工程。其主要目的是确定系统的主要功能模块，以及功能模块之间如何相互作用，从而把输入数据转换为所需要的输出。显然，确定功能模块的过程是一个权衡的过程，设计师需要根据不同器件的性能价格比作出选择。在设计规模较小或要求较低的数字系统时，设计师一般可以根据经验或简单的计算来完成这个步骤。但是对于微处理器等规模较大或是军用设备等性能价格比要求很高的数字硬件，需要有更加精确的性能模型。

在性能模型这样的抽象层次，每一功能模块被抽象为在一定时间内执行一定的任务。性能模型并不说明任务怎样被执行，实际上，在这个阶段通常也不知道任务是怎样执行的，设计师只知道某个功能模块在何种条件下被激活以及经过多少时间产生运算结果。因此，性能模型又被称为非解释模型(uninterpreted model)。绝大多数非解释模型都建立在Petri 网(Petri net)或队列理论的基础上，而数字系统一般使用 Petri 网模型。

在以往的设计实践中，设计师们曾经使用各种计算机语言描述性能模型。这些计算机语言可以分成两类：通用的软件编程语言如 APL，Fortran，Ada，C/C++等和专用的性能模型语言。使用通用编程语言的优点在于设计师可以使用现有的编程支持环境，并且不需要学习额外的语言。但是，使用这种方法时，由于需要编写支持性能仿真各种概念的子程序，编程工作量往往很大。因此，人们在通用编程语言基础上发展了一系列专用性能模型语言，如 Simscript，GPSS 和 RESQ 等，使用这些语言使得设计师不必掌握诸如随机过程仿真、概率和统计等专业知识就可以建立性能模型。在传统的设计过程中，以上两类语言的性能模型存在着同样一个问题，就是这些语言与硬件描述语言不兼容。也就是说，在性能模型与后续的算法级、寄存器级、逻辑级之间，需要经过手工或软件的翻译才能互相传递设计信息。而这种翻译过程非常费时，也容易发生错误。由此可以看到，设计师需要在整个设计过程中使用一种标准语言，VHDL 语言具有极强的抽象能力，显然适合于充当这一标准语言。

本节介绍如何用 VHDL 描述 Petri 网，并给出了一个很简单的例子。

9.3.1 Petri 网

Petri 网的命名来源于其创始人 Carl Petri。最初，Carl Petri 用原始的 Petri 网概念去分析计算机系统的异步元件之间的通信。后来，Petri 网逐渐发展到具有了成熟的数学基础，并被成功地用于如有限状态机、软件程序、计算机网络和工艺过程等系统的建模和仿真。

先用图 9.7 的例子来说明 Petri 网的概念。图中的黑色实心圆点称作记号(token)；空

心圆圈称作地点(place),记为 p_x;横线称作跃迁(transition),记为 t_x。记号在 Petri 网中的移动代表数据在系统中的处理,通过激活跃迁(firing transition),记号沿着 Petri 网中的有向边从一个地点转移到另一地点。当所有指向某一跃迁(横线 t_x)的地点均含有记号时,该跃迁被激活。这些地点称作输入地点。跃迁被激活后,所有输入地点的记号被去掉,而所有由跃迁指出的地点都要加上一个记号,这些地点称为输出地点。以上就是 Petri 网的记号规则,也就是 Petri 网的执行语法。

在图 9.7 中,符合激活条件的跃迁只有 t_1。因此 t_x 被激活,此时输入地点 p_1 和 p_2 的记号被去掉,而输出地点 p_3 内将被放置一个记号。这次跃迁之后的 Petri 网如图 9.8。此刻 t_2 被激活,p_3 中的记号被去掉,p_1 和 p_4 内将分别放置一个记号。接下来,跃迁 t_3 和 t_4 分别被激活,Petri 网返回初始状态。

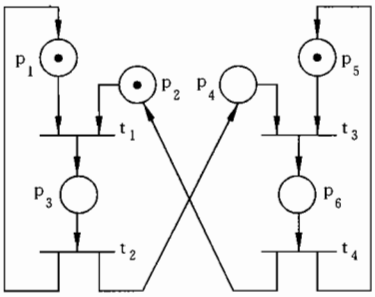


图 9.7 一个简单的 Petri 网

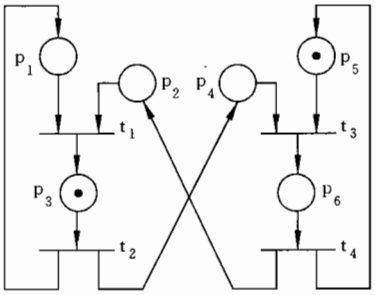


图 9.8 图 9.7 中的 Petri 网经过一次跃迁

实际上,图 9.7 的 Petri 网是一个简化的生成者-消费者模型。左边的部分代表生产者,而右边代表消费者。通过隐含在模型中的握手协议,记号的生产速度将由消费速度决定。生产者通过跃迁 t_1 产生一个记号,通过跃迁 t_2 通知消费者。在消费者通过跃迁 t_3 消除记号并通过跃迁 t_4 通知生产者之前,生产者不能产生新的记号。

在为电子系统构造 Petri 网模型时,可以认为地点表示条件。当地点内含有记号时,认为条件满足。Petri 网内特定的记号分布称作标记(marking),表示满足某一系统状态的条件集合。跃迁可以认为是表示事件,当跃迁的所有输入条件满足时,跃迁被激活。

9.3.2 用 VHDL 描述 Petri 网

研究人员已经发展了各种用 VHDL 描述 Petri 网的方法。一般说来,这些用 VHDL 建立的性能模型都采用扩展 Petri 网作为基础。为了提高 Petri 网的描述能力,特别是对数字系统建模的能力,对 Petri 网进行了如下扩展:

(1) 记号可以具有数值或颜色,也就是说可以存在不同种类的记号,从而使 Petri 网能够表示更加复杂的条件,此时存在相应的记号处理和流动机制;

(2) 跃迁或模型中元素的激活能够具有一定的延时;

(3) 决定跃迁激活的机制被抽象为输入条件的逻辑运算式和输出条件的函数。

模型中元素的控制机制依赖于输入和输出,输入分为数据和控制两类,而输出分为独立和不独立两类。其中,不独立输出只能连接到数据输入,而独立输出只能连到控制输入。

不独立输出只有在其中不含有记号时，才能接受记号。而对独立输出来说，任何时刻都可以在其中放置记号，即可以覆盖已有记号。数据输入中的记号可被去掉，控制输入中的记号只能被拷贝。跃迁在其输入满足用户定义的逻辑表达式和不独立输出条件(如果存在这样的条件)时被激活。

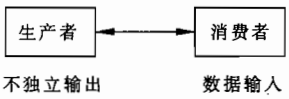


图 9.9　用 VHDL 描述生产者-消费者系统的方框图

图 9.9 是用扩展 Petri 网描述生产者-消费者系统的方框图，下面给出了描述该系统的 VHDL 代码。生产者的行为用带有不独立输出的元素表示，即在输出含有记号时不能产生新的记号。由于消费者必须在激活后去掉记号，因此用带有数据输入的元素表示。

```
-- 用 VHDL 建立的生产者-消费者系统的性能模型
-- 生产者模型
use Petri_Pkg.all;
entity Producer is
        port(Port_Out : inout Protocol Token);
end Producer;
architecture Behavior of Producer is
begin
        process
begin
        if Port_Out /= TK_REMOVED then
                wait until Port_Out = TK_REMOVED;
        end if;
        Port_Out<=TK_PRESENT;
        Wait until Port_Out = TK_ACK;
        Port_Out<=TK_RELEASED;
end process;
end Behavior;

-- 消费者模型
use Petri_Pkg.all;
entity Consumer is
        port(Port_In : inout Protocol Token);
end Consumer;
architecture Behavior of Consumer is
```

```
begin
        process
begin
        if Port_In /= TK_PRESENT then
                wait until Port_Out = TK_PRESENT;
        end if;
        Port_Out<=TK_TK_ACK;
        Wait until Port_Out = TK_REALEASED;
        Port_Out<=TK_REMOVED;
end process;
end Behavior;
```

进程 Producer 和 Consumer 与决断函数 Protocol 合在一起，就描述了生产者-消费者系统的行为，其中决断函数 Protocol 控制两个进程之间的数据通信。为了保证在消费者读取并消除旧的记号之前，生产者不至于产生新的记号，本例中使用了四阶段握手协议(four-phase handshaking protocol)，也称作双轨编码(double-rail coding)。在程序包 Petri_Pkg 中，记号被定义为枚举类型 Token，即

```
type Token is (TK_REMOVED,TK_PRESENT,TK_ACK,TK_RELEASED);
```

在初始化阶段，连接进程 Producer 和 Consumer 的信号被置为 TK_REMOVED，表示没有记号存在。生产者进程产生一个记号(通过置 Port_Out 为 TK_PRESENT)，通知进程 Consumer。然后该进程执行一个 wait 语句等待进程 Consumer 确认收到记号。接下来，进程 Consumer 置 Port_Out 为 TK_ACK，通知进程 Producer，并等待释放记号。收到 TK_ACK 后，进程 Producer 置 Port_Out 为 TK_RELEASED，等待进程 Consumer 消除记号。记号被消除后，整个握手协议完成。

练 习 题

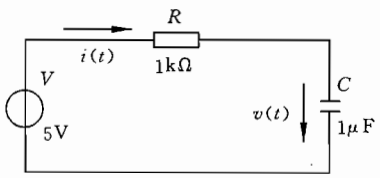
第 1 章

1.1 画出 16 位串行进位加法器的结构模型，其基本单元是全加器。用 C 或 VHDL 语言写出它的行为描述。

1.2 画出全加器的一种逻辑门级描述，其基本单元为与门(AND)、异或门(XOR)、或门(OR)、反向器等基本逻辑门。参考有关教科书画出 CMOS 全加器的电原理图，其基本单元为晶体管、电阻、电容等基本电路元件。对于全加器的门级描述和电路级描述从基本单元个数、连线(线网)条数两方面进行比较。

分别用你所熟悉的逻辑模拟器和电路模拟器 SPICE 在同一计算机上对全加器进行逻辑仿真和电路仿真。设定输入信号包括所有 8 种可能的信号波形，比较仿真时间。

1.3 题图 1.1 画出了一个简单 *RC* 串联电路的结构描述，试写出它的行为描述。



题图 1.1 *RC* 串联电路

1.4 下面是一个逻辑门级电路的行为描述，设计至少两种逻辑门级结构描述实现该电路的功能。

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

1.5 题图 1.2 是一个计数器的逻辑图。试用 C 或 VHDL 语言写出该电路的行为描述。对逻辑门级结构描述和行为描述进行对比，说明各有什么特点。

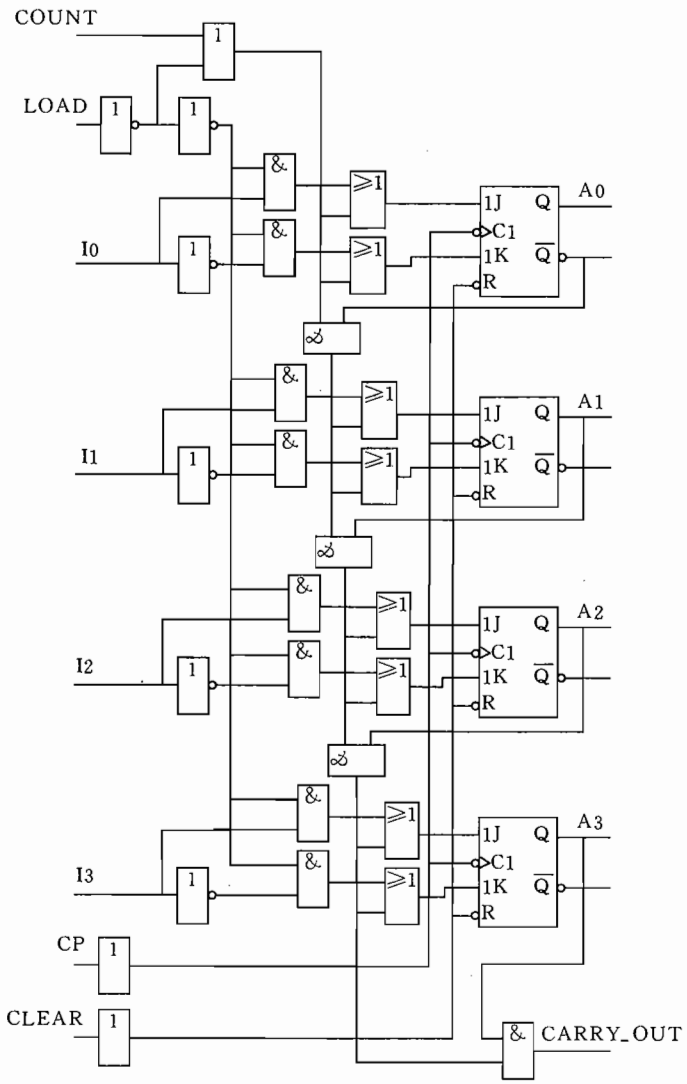
1.6 题图 1.3 表示了两个器件之间的接口。其中 DEVICE_1 为发端器件，DEVICE_2 为收端器件。两个器件之间的接口由 READY，ACCEPT，DATA 和 DVALID 等信号组成。两个器件之间的通信协议如下：

(1) DEVICE_2 发出准备接收信号 READY；

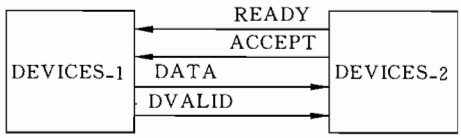
(2) DEVICE_1 检测到 READY 的上升沿后，将数据置在数据线 DATA 上，并给出数据有效信号 DVALID；

(3) DEVICE_2 检测到 DVALID 的上升沿后，从数据线 DATA 上读取数据。读数据结束后发出读数据结束标志 ACCEPT，并对 READY 复位。

试画出 4 个信号 READY，ACCEPT，DATA 和 DVALID 之间的时序关系，并用箭头标明信号之间的触发关系。



题图 1.2　某计数器的电路图



题图 1.3　两个器件之间的接口

1.7　已知如下的算法描述，试画出数据流图。图中节点表示数据运算，支路表示输入和输出值。

for I = 1 to 3 loop

A(I)=B(I) + C(I)

D(I)=A(I) + C(I)

end for;

1.8 对于不同的设计领域，试比较 top_down 和 bottom_up 两种设计方法。从制造成本、设计/制造周期诸方面总结两种方法的主要特点。

讨论如何将 top_down 和 bottom_up 两种方法用于软件设计。

串行进位加法器中，加数和被加数全是并行输入，和数也是并行输出，但各全加器之间的进位却是串行传递的。对于 16 位串行进位加法器，最高位的和数要经 16 级延时才能算出，因此速度较慢。试设计 4 位超前进位加法器，由输入信号直接产生进位所需的信号，从而消除串行进位所用时间，提高电路工作速度。对两种加法器电路进行对比。

第 2 章

2.1 对大系统进行仿真时，仿真效率非常重要。仿真效率定义为

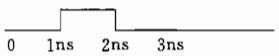
E= 实际逻辑时间 ÷ 仿真所用 CPU 时间

这里的实际逻辑时间指硬件电路实际工作时完成指定运算所需的实际时间，仿真所用 CPU 时间指用仿真器对硬件模型进行仿真时，模拟同样的运算所需的 CPU 时间。试讨论仿真效率与哪些因素有关。

2.2 对于下面的两个信号赋值语句和题图 2.1 所示的输入波形，画出 X1 和 X2 的波形。

X1<=Y after 2 ns;

X2<=transport Y after 2 ns;



题图 2.1 思考题 2.2 的输入波形

2.3 输入信号 X 的波形如题图 2.2 所示，试对于下面两个信号赋值语句，画出 Y 和 Z 的波形。

Y<=X after 2 ns;

Z<=transport X after 2 ns;

题图 2.2 思考题 2.3 的输入波形

2.4 对下面的 VHDL 源代码进行仿真，并回答后面的 5 个问题。

entity EX2 is

```
end EX2;

architecture TEST of EX2 is
    signal A,B,C,D: Std_logic_vector(7 down to 0) := "1000 1100";
    procedure X(signal Y: in Std_logic_vector(7 downto 0);
        signal Z: Std_logic_vector(7 downto 0) ) is
        variable TEM : Std_logic_vector(7 downto 0);
    begin
        for I in 0 to 7 loop
            TEM(I) := Y(7-I);
        end loop;
        Z<=TEM after 1ns;
    end X;
begin
    S1: X(C,D);
    S2: C<=A and B after 2 ns;
    S3: A<="1011 0110" after 5 ns,"1000 1100" after 10 ns;
    S4: B<="1111 0000" after 5 ns,"0000 1111" after 10 ns;
end TEST;
```

(1) 假定在仿真时钟为 0 时刻开始仿真，对于标出的几个语句(S1,S2,S3 和 S4)，列出在哪些时刻会发生信号赋值操作。

(2) 在哪些时刻信号 C 上有事件发生？

(3) 画出信号 C 上的波形。

(4) 信号 D 上的最终值是多少？在什么时刻信号 D 达到其最终值？

(5) 简单解释过程 X 的作用。

2.5 对下面的 VHDL 源代码进行仿真，并回答后面的问题。

```
entity FINAL is
end FINAL;

architecture TEST of FINAL is
    signal A,B,C,D: Std_logic_vector(7 down to 0) := "1100 1100";
    procedure X(signal Y: in Std_logic_vector(7 downto 0);
        signal Z: Std_logic_vector(7 downto 0) ) is
        variable TEM : Std_logic_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<="1011 0110" after 7 ns,"1000 1100" after 15 ns;
    S4: B<="1011 0110" after 7 ns,"0000 1111" after 15 ns;
end TEST;
```

(1) 假定在仿真时钟为 0 时刻开始仿真,对于标出的几个语句(S1,S2,S3 和 S4),列出在哪些时刻会执行这些语句。

(2) 在哪些时刻信号 C 上有事件发生?

(3) 画出信号 C 上的波形。

(4) 信号 D 上的最终值是多少? 在什么时刻信号 D 达到其最终值? 解释你给出的答案。

(5) 简单解释过程 X 的作用。

2.6 下面给出了实体 QUESTION 以及对它进行测试的 VHDL 源代码。假如对它进行仿真测试,画出信号 CON,INP,Q 和 QOUT 的波形,在波形图上准确地标出时间坐标。

```
entity QUESTION is
    port(INP: in Std_logic; CON: in Std_logic; QOUT: out Std_logic)
end QUESTION;

architecture ANSWER of QUESTION is
    signal Q: Std_logic;
begin
    B1: block (CON = '1')
    begin
        Q<=guarded INP after 10ns;
        QOUT<=Q after 5ns;
    end block;
end ANSWER;

entity TB is
end TB;
```

```
use work.all;
architecture QT of TB is
    signal INP,CON,QOUT: Std_logic;
    component QUEST
        port(INP: in Std_logic; CON: in Std_logic; QOUT: out Std_logic);
    end component;
begin
    C1: QUEST
    port map (INP,CON,QOUT);
    process
    begin
        INP<='1' after 5ns;
        CON<='1' after 10ns; '0' after 30ns;
        wait;
    end process;
end QT;
```

2.7 下面给出了实体 BLKBOX 的 VHDL 行为描述：

```
entity BLKBOX is
    generic(T: TIME);
    port(I: in Std_logic; Z: out Std_logic);
end BLKBOX;

architecture INSIDE of BLKBOX is
    signal Y1,Y2: Std_logic;
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 BLKBOX of TB is
    signal I,Z: Std_logic;
    component BLKBOX
```

```
        generic(T: TIME);
        port(I: in Std_logic; Z: out Std_logic);
    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 QT;
```

(1) 利用测试程序(TB),画出信号 I,Y1,Y2 和 Z 的波形。

(2) 描述实体 BLKBOX 的功能。

2.8 下面 VHDL 源代码的实体 INT_SEQ 中给出了两个循环体。粗看起来这两个循环体都可以输出整数 0～9,但实际上这两个循环体中只有一个能正常工作,阅读源代码,分析哪一个循环体不能正常工作,并与仿真结果进行对比。

```
entity INS_SEQ is
    port(OUT21: out Integer; OUT2: out Integer);
end INS_SEQ;

architecture LOOPS of INS_SEQ is
    signal START: Std_logic;
begin
    START<='1';
    process (START)
    begin
        for I in 0 to 9 loop;
            OUT1<=I after I * (1ns);
        end loop;
    end process;

    process (START)
    begin
        for I in 0 to 9 loop;
            OUT2<=transport I after I * (1ns);
        end loop;
    end process;
```

end LOOPS

第 3 章

3.1　用两种方法写出 4 位二进制加法器的模型。

(1) 用 4 个相同的单元级连组成 4 位二进制加法器。

(2) 并行实现 4 位二进制加法器。

3.2　设计一个带有使能端的矢量输入/输出的二选一电路。两个输入信号和输出信号的字长为 8,如果使能信号为'0',则输出量为"11111111";如果使能信号为'1',则输出量由选择信号和输入信号共同确定。在电路中包含有如下三种延时:

(1) DATA_DEL 是从数据输入端到数据输出端的延时;

(2) SELECT_DEL 是从选择信号输入端到数据输出端的延时;

(3) ENABLE_DEL 是从使能端到数据输出端的延时。

3.3　对描述第 3 章图 3.3 所示的 PLA 的 VHDL 源代码作如下改进:

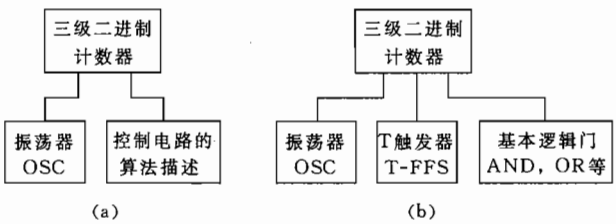
(1) 改变 PLA 的行为模型使它能实现不同的逻辑函数;

(2) 改变 PLA 的行为模型使它能允许任意的输入/输出个数。

3.4　下面是 T 触发器的真值表,试写出它的接口描述和行为描述。

Rst	Clk	T	Q(t)	Q(t+1)
0	上升沿	1	1	0
0	上升沿	1	0	1
0	上升沿	0	1	1
0	上升沿	0	0	0
1	X	X	X	0

3.5　题图 3.1 中给出了三级二进制计数器的设计树,用 VHDL 语言设计该三级二进制计数器。设计过程分为两个阶段,第一阶段用设计树(a),第二阶段用设计树(b)。两个阶段全要用到振荡器,假定振荡器的接口描述如下:



题图 3.1　计数器的两个设计树

```
entity OSC is
    generic(HI_TIME,LOW_TIME:TIME);
    port(RUN: in Std_logic; CLK: out  Std_logic);
```

```
end OSC;
```

其中，HI_TIME 和 LOW_TIME 分别是一个振荡周期中高电平和低电平的持续时间。当 RUN='1'时，振荡器振荡；当 RUN='0'时，振荡器停振。

设计的第一阶段：这一阶段以设计树(a)为依据，要求写出计数器的算法模型，计数器的接口描述应该为

```
entity UP_CNT is
    generic(R_DEL,CLK_DEL: TIME);
    port(RESET,COUNT,CLK: in Std_logic;
    CNT: inout Std_logic_vector(2 downto 0));
end UP_CNT;
```

其中，R_DEL 和 CLK_DEL 分别为计数器的复位延时和时钟延时，当 RESET='1'时，计数器被复位；当 RESET='0'且 COUNT='1'时，在时钟的上升沿计数器进行加计数。设计这个电路的同时，设计一个对计数器电路进行测试的测试电路结构，利用这个测试电路结构，对计数器进行仿真测试，验证电路的正确性。

设计的第二阶段：这一阶段以设计树(b)为依据，要求写出计数器的结构模型，计数器中可以利用的基本单元为 T 触发器和基本逻辑门。设计过程要满足如下要求：

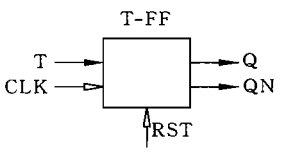
(1) 设计同步计数器，不能使用链式进位。

(2) 采用层次式设计方法，即只能使用振荡器、T 触发器和基本逻辑门的行为模型。

(3) 振荡器产生占空比为 25%，频率为 10MHz 的方波。

T 触发器的接口如题图 3.2 所示。

如果 T='1'，在时钟 CLK 的上升沿触发器反转；如果 T='0'，触发器保持它原来的值。RESET 是复位端，它的优先级最高，如果 RST='1'，则不论 CLK 为何值，触发器的输出端 Q 复位为'0'，QN 为'1'。该触发器的复位延时为 10ns，时钟到输出端的延时为 15ns。



题图 3.2　T 触发器的接口描述

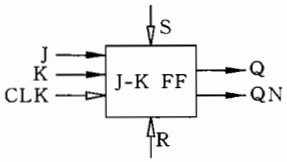
(4) 所有逻辑门的延时全为 5ns。

(5) 用类属参数传递延时数据，使模型的通用性强。即 T 触发器及逻辑门的接口描述应具有下述形式：

```
entity T_FF is
    generic (R_DEL,CLK_DEL: TIME);
    port(RST,T,CLK: in Std_logic; Q,QN: inout Std_logic);
end T_FF;

entity AND2 is
    generic (DEL: TIME);
    port(IN1,IN2: in Std_logic; O: out Std_logic);
end AND2;
```

(6) 写出计数器的结构描述，并采用第一阶段写出的对计数器测试的测试结构对该计数器进行仿真测试。

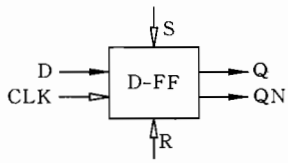
3.6 用正边沿触发的 J-K 触发器重复完成上题的内容。触发器的接口、延时参数以及基本逻辑门和反向器的延时参数如题图 3.3 所示。



	参数	输入端	输出端	参数值
J-K 触发器的延时参数	TPLH	S	Q	16 ns
	TPHL	S	QN	25 ns
	TPLH	R	QN	16 ns
	TPHL	R	Q	25 ns
	TPLH	CLK	Q 及 QN	16 ns
	TPHL	CLK	Q 及 QN	25 ns
基本逻辑门的延时参数	TPLH：5 ns	TPHL：3 ns		
反向器的延时参数	TPLH：3 ns	TPHL：5 ns		

题图 3.3 J-K 触发器的接口描述及延时参数

3.7 用负边沿触发的 D 触发器重复完成上题的内容。触发器的接口、延时参数以及基本逻辑门和反向器的延时参数如题图 3.4 所示。



	参数	输入端	输出端	参数值
D 触发器的延时参数	TPLH	S	Q	16 ns
	TPHL	S	QN	25 ns
	TPLH	R	QN	16 ns
	TPHL	R	Q	25 ns
	TPLH	CLK	Q 及 QN	16 ns
	TPHL	CLK	Q 及 QN	25 ns
基本逻辑门的延时参数	TPLH：5 ns	TPHL：3 ns		
反向器的延时参数	TPLH：3 ns	TPHL：5 ns		

题图 3.4 D 触发器的接口描述及延时参数

3.8 题图 3.5 是一个简单微处理器的模型。用第 3 章讨论过的基本单元设计这个电路，所设计的电路必须满足下列要求：

(1) 数据处理单元只能使用一个 4 位 ALU，但可以使用任意多个所必需的其他逻辑单元。要求设计出的数据处理单元可以进行下述两种运算：

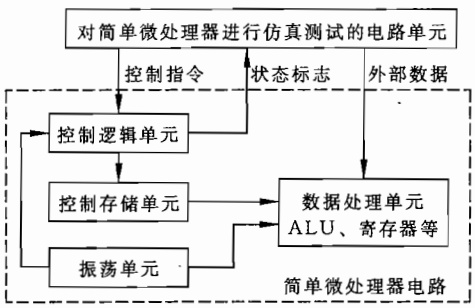
$$Z = X \text{ xor } Y$$
$$Z = 2*(X - Y)$$

数据处理单元中所需的输入数据通过外部接口由测试电路提供，控制运算的信息由控制存储单元提供，数据运算结果保留在 ALU 中的寄存器中。

(2) 在控制存储单元(用 RAM 实现)中，存储一系列控制字，对每一种运算进行控制。控制逻辑单元从外部测试电路接受指令，用来控制数据处理单元完成指定的运算。

(3) 控制逻辑单元应该能与外部测试电路交换信息，以控制运算的开始、结束和加载数据。

(4) 由振荡单元统一产生时钟。



题图 3.5　简单微处理器的模型

(5) 尽量优化设计方案，使完成两种数据运算所需的时间最短，且要求完成两种数据运算时共用同一运算单元(ALU)。

(6) 写出实验报告，对设计过程和设计结果进行总结。

3.9　设计一个功能比第 3 章介绍的 ALU 更强的算术逻辑单元，用新设计的 ALU 重复题 3.8 的设计内容。从运算时间和使用的多路选择器及寄存器的个数两方面对设计结果进行对比。

3.10　试写出 SN7400(2 输入与非门)的 VHDL 模型，模型中使用类属参数 TPHL 和 TPLH 表示下降延时和上升延时，并利用手册中给定的典型延时值对模型进行仿真。

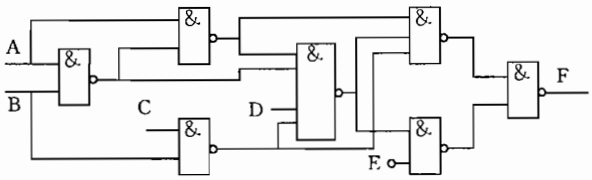
3.11　对于门阵列中的 2 输入与非门，考虑其输入连线的延时，写出其 VHDL 模型。请查阅有关资料，确定延时值与连线长度的关系，2 输入与非门内部延时值可以从门阵列手册中获得。

3.12　按下述要求写出与非门的 VHDL 模型。

(1) 要求是一个通用的 N 输入模型，N 是实体说明中的类属参数。

(2) 延时值与逻辑门的扇出个数有关。当在一个具体网络中实际使用与非门时，根据具体器件的扇出情况确定门延时值。

使用该与非门模型对题图 3.6 的电路进行仿真。



题图 3.6　与非门组成的电路

3.13　利用第 3 章 3.4 节给出的程序包 TIMING_CONTROL，构造上一题中给出的网络的模型，要求分别使用 Delta 延时、延时最小值、典型值和最大值。

3.14　原则上，数字电路只需要两个逻辑值。讨论为什么逻辑门级器件造型需要多值逻辑。

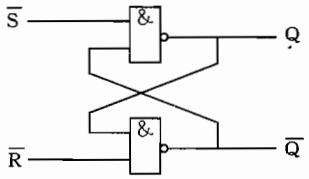
3.15　写出带三态输出的缓冲器 SN74LS240 的 VHDL 模型。模型中要包括能表述数据延时、使能延时等行为的时序关系，要考虑从高阻态进入正常态以及从正常态进入高阻态等各种情况。

3.16　用表 3.4 所示的 9 值逻辑构造 NMOS 反相器的模型并对它进行仿真。

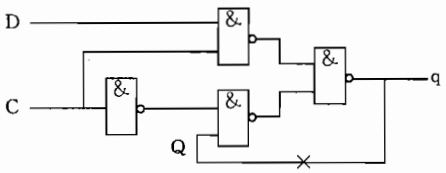
3.17　用 IEEE 的 9 值逻辑系统构造与非门的模型。对于题图 3.7 的 RS 触发器，假定两个与非门的初始输出值全为'U'，两个输入端 $\overline{R}$ 和 $\overline{S}$ 的初始状态全为 '0'，且同时从'0'状态转换为'1'状态。对于下述两种情况，对 RS 触发器进行仿真，并对仿真结果进行对比分析。

(1) 两个与非门的延时值相同；

(2) 两个与非门的延时值不同。



题图 3.7　与非门实现的 RS 触发器



题图 3.8　简单 D 触发器

3.18　多值逻辑的使用可以发现时序逻辑电路中存在的不稳定问题。对于题图 3.8 所示的简单 D 触发器，假定所有器件都采用 Delta 延时，并假定初始状态为：D = C = Q = '1'。按如下方法，对电路进行分析，以验证采用多值逻辑的必要性。

(1) 假定在 t = 0 时，时钟信号 C 变为 X，此后，在 t=10 ns 时，时钟信号 C 变为 0，对该电路进行仿真并验证结论：q = X。这一仿真过程在开关电路理论中被称为 Eichelberger 技术。

(2) 改变电路中逻辑门的延时，检查什么样的延时可以消除输出的 X 状态，什么样的延时会产生输出的 X 态？

(3) 设想电路在图中的×点断开，以 D,C 和 Q 为变量写出 q 的布尔表达式并画出卡诺图。对卡诺图进行分析，通过在布尔表达式中增加适当项的方法消除所有逻辑冒险。反

复修改延时值，以证明电路的稳定性与延时无关。

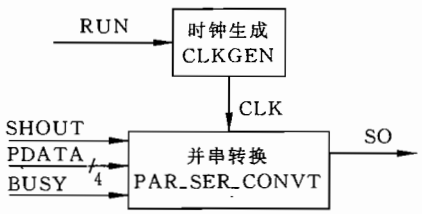
3.19　如果不同器件的 VHDL 模型使用了不同数据类型，把它们互连构造数字系统的 VHDL 模型时将会有些困难。找出一种映射关系，实现 MVL4('0','1','X','Z')和 IEEE 9 值系统之间的相互转换。讨论这种转换会丢失什么信息？根据高等代数中的知识，讨论这种映射关系的代数性质。

3.20　对于第 3 章中讨论过的 PLA，试写出与第 3 章中给出的行为模型等效的结构式模型。

第 4 章

4.1　列出系统级硬件的 VHDL 模型构造的几种方式，并对每种方式做简单评价。

4.2　题图 4.1 是一个两模块数字系统的方框图，这个系统的基本功能是接受 4 位数据 PDATA，然后把它们串行移出。串行移出的码率由连接在并串转换电路之外的振荡器 CLKGEN 提供。当 RUN 为'1'时，CLKGEN 输出时钟信号 CLK，振荡周期为 50ns，当 RUN 为'0'时，振荡器输出 CLK 保持为'0'。并串转换电路的功能如下：在信号 SHOUT 从'0'变为'1'的时刻，并行数据被加载到串并转换电路的内部寄存器 SR，并将状态标志位 BUSY 置位为'1'。从这时开始，数据通过 SO 串行移出。数据第一位移出的时刻是信号 BUSY 变为'1'后时钟 CLK 的第一个上升沿，后面 3 个时钟周期的上升沿依次把后面 3 位数据移出。4 位数据全移出之后，状态标志位 BUSY 被复位为'0'。假定在 BUSY 为'1'期间，不会再有数据加载，试写出该电路的算法模型。



题图 4.1　并串转换电路

4.3　题表 4.1 是一个 Moore 有限状态机的状态表，它对输入数据进行串行偶校验，这里给出了该状态机的部分 VHDL 算法描述，试完成该电路的算法模型。需要完成的工作包括：① 填满表示状态表的数组元素，② 假定在 R 为'1'时，状态机被复位为 S0，写出状态机模型的可执行语句。

题表 4.1　Moore 有限状态机的状态表

当前状态	下一状态		输出值
	X='0'	X='1'	
S0	S0	S1	0
S1	S1	S0	1

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

```
architecture ALG of MOORE is
    type STATE is (S0,S1);
    signal FSM_STATE: STATE := S0;
    type TRANSITION is recode
        OUTPUT: Std_logic;
        NEXT_STATE: STATE;
    end recode;
    type TRANSITION_MATRIX is array(STATE,Std_logic) of
        TRANSITION;
    constant STATE_TRANS: TRANSITION_MATRIX :=
        ---------;
        ---------;
begin
---------;
---------;
end ALG;
```

4.4 对题表 4.2 的状态表，写出其有限状态机的算法模型，并对该状态机仿真。仿真时要求利用尽可能短的输入激励序列，使得模型至少进入每个状态一次。写出的算法模型要满足下述要求：

题表 4.2 某有限状态机的状态表

当前状态 \ 输入		X=0		X=1		X=2		X=3	
		下一状态	输出	下一状态	输出	下一状态	输出	下一状态	输出
Qt	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
	S4	S2	1	S1	0	S2	1	S2	1
	S5	S2	1	S1	1	S2	1	S3	1

(1) 用枚举类型表示所有状态；

(2) 用记录类型表示状态变换及输出信息；

(3) 用数组元素对状态表编码；

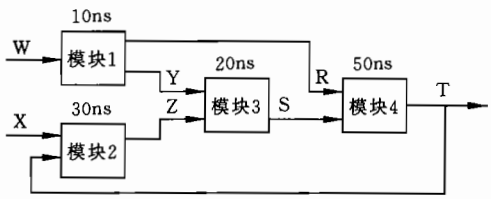
(4) 利用下述程序包和有限状态机实体，只需自行写出结构体。

```
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 Std_logic; X: in INT4; Z: out Std_logic);
end FSM;
```

4.5 写出一段 VHDL 源代码，用来检查信号 X 的上升沿至信号 Y 的下降沿之间的时间是否比规定时间 T 小。

4.6 题图 4.2 给出了由 4 个模块互连组成的电路结构，对于每个模块，图中给出了其传输延时值但并没有给出模块功能。试写出这个电路的 VHDL 模型的框架，模型应该由一个实体描述其接口，假定 W,X,T 为其接口信号，而 Y,Z,R 和 S 为内部信号，并假定任何模块输入值的变化都会在指定延时之后影响其输出值。在已知上述信息的前提下写出尽可能详细的 VHDL 电路模型。



题图 4.2 4 模块组成的电路结构

4.7 设计一个带使能输入的 2-4 译码器。数据输入信号及译码输出信号可以定义为矢量，也可以定义为标量。当使能输入为'0'时，译码输出为高阻态'Z'。当使能输入为'1'时，根据数据输入值的不同，只有一个译码输出可能为'1'，其他译码输出全为'0'。当输入数据和使能信号全为'X'或'Z'时，可自行选择适当的译码输出，但要说明为什么要选择这样的输出值。设计的译码器模型与外部电路的接口应该与一般的硬件译码器一致并带有两个延时数据，这两个延时数据如下：

(1) DATA_DELAY：在使能端为'1'的情况下，从输入数据发生变化到译码输出发生变化的延时。

(2) ENABLE_DELAY：从使能信号发生变化到译码输出发生变化的延时。

设计出的译码器应该包括实体描述和一个系统级行为域的结构体描述。

4.8 设计一个带使能端的 2-1 矢量复用器。两个输入数据全为 8bit 矢量，输出也是 8bit 矢量。如果使能信号为'0'，则输出信号应为"ZZZZZZZZ"，如果使能信号为'1'，则输出信号由选择信号和输入数据所决定。模型中包括三个延时：

(1) DATA_DELAY：在使能端为'1'的情况下，从输入数据发生变化到数据输出发生变化的延时。

(2) SELECT_DELAY：在使能信号为'1'的情况下，从选择输入发生变化到数据输出发生变化的延时。

(3) ENABLE_DELAY：从使能信号发生变化到数据输出发生变化的延时。

设计出的选择器应该包括实体描述和一个系统级行为域的结构体描述。

4.9　设计一个加法/减法计数器，计数器有 4 个控制输入：UP，DOWN，LOAD 和 ENABLE。计数器所有状态变化只能发生在时钟输入 CLK 的上升沿（这是一个同步计数器）。清零信号 CLEAR 按非同步方式工作，即 CLEAR 可以在任何时刻把计数器的所有位全清为零，清零时间与时钟信号 CLK 无关。输入信号 DATA_IN 和输出信号 DATA_OUT 的值全为 4bit 信号，当 ENABLE 为'1'时，DATA_OUT 的值与计数器中存储的数据相同，当 ENABLE 为'0'时，输出信号 DATA_OUT 的值为"ZZZZ"。ENABLE 的作用也与时钟信号 CLK 无关。如果控制信号 UP 为'1'，则计数器在 CLK 的上升沿进行加计数。如果控制信号 DOWN 为'1'，则计数器在 CLK 的上升沿进行减计数。如果控制信号 LOAD 为'1'，则计数器在 CLK 的上升沿把并行输入数据 DATA_IN 的值加载到计数器内部寄存器。输出信号 C 是溢出标志。当加计数计满为"1111"时，C 输出为'1'；当减计数计到"0000"时 C 也输出为'1'。计数器按模 16 计数。如果 UP，DOWN 和 LOAD 中不只一个信号为'1'，则表示没有对计数器的工作状态作出规定。设计电路模型时，可以假定已知函数 INC4 和 DEC4 分别是完成 4bit 信号加 1 运算和减 1 运算的函数。

（1）画出计数器的进程模型图（PMG），应该采用较简单的进程（可以多用几个进程），而不应该用复杂进程。

（2）基于进程模型图写出计数器的系统级 VHDL 模型。

4.10　按下面的要求写出 8214 中断控制器的系统级模型。

（1）模型应该是芯片级行为模型，包括一个实体和一个结构体。

（2）分析说明如何描述该电路的传输延时。

（3）画出该电路的进程模型图（PMG）。

（4）模型应该对建立时间、保持时间违例情况进行检查。

（5）依次检查测试（仿真）该模型。

（6）写出报告。报告中应该说明如何实现这个电路，如何验证这个模型，并给出仿真结果。

4.11　按下面的要求写出 16bit 并行误码校正电路 SN74ALS617 的系统级模型。

（1）模型应该是芯片级行为模型，包括一个实体和一个结构体。

（2）分析说明如何描述该电路的传输延时，这些数据产生的效果应该与器件手册中给出的数据一致。

（3）画出该电路的进程模型图（PMG）。

（4）模型应该对建立时间、保持时间违例情况进行检查，检查的依据是器件手册中给出的数据。

（5）检查测试（仿真）该模型，检查的要求为：检查测试误码校正电路的测试结构中应该包含一个随机存取存储器（RAM）和一个控制单元（CU）。RAM 用来引入误码，要求对 1bit 误码、2bit 误码、3bit 误码、全 1 误码和全 0 误码进行检测。控制单元 CU 用来调度误码检测及校正电路、RAM 和 CU 本身。

（6）写出报告。报告中应该说明如何实现这个电路，如何验证这个模型，并给出仿真结果。

4.12　下面是 Std_logic 型信号决断函数，解释这个函数的工作原理，并说明这个函

数实现了实际硬件电路的什么特性。

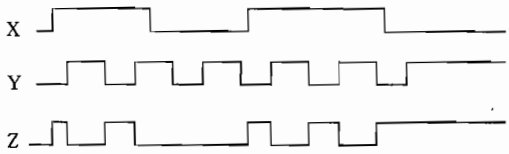
```
function RES_FUNC(signal X: Std_logic_vector) return Std_logic is
begin
    for i in X'Range loop
        if X(i) = '0' then
            return '0';
        end if;
    end loop;
end RES_FUNC;
```

4.13 下面是两进程模块的部分 VHDL 源代码，进程 ONE 和进程 TWO 分别由外部提供的两个脉冲 P1 和 P2 激活，两个进程全要对寄存器 X 加载，因此需要采用时分复用技术，即进程 ONE 在 P1 为'1'时控制 X，而进程 TWO 在 P2 为 '1'时控制 X。

```
type MVL is ('0','1','Z')
    signal X: F MVL;        -- F 是预先定义好的总线决断函数

ONE: peocess(P1)
begin
    -- ----------
    -- ----------
    X<=Y after 100ns;
end process ONE;

TWO: process (P2)
begin
    -- ----------
    -- ----------
    X<=Z after 199 ns;
end process TWO;
```

上面给出的 VHDL 源代码是否能正确工作？如果不能正确工作，对它作必要的改正。

4.14 假定一个器件中有两个 Std_logic 型输入信号 X 和 Y，一个 Std_logic 型输出信号 Z。每当输入信号 X 从'0'变为'1'时，输出信号 Z 都会变成'1'且保持为'1'，直到输入信号 Y 上发生变化。在 X='1'时，Y 上的任何变化都会使输出 Z 的值发生变化。如果输入 X 和 Y 同时发生变化，输出 Z 的状态不定，题图 4.3 示意给出了这个器件的时序波形。对于上述行为描述，写出了下面的算法描述：

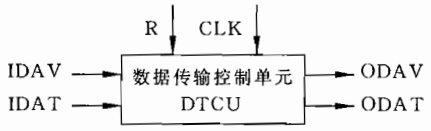


题图 4.3　题 4.14 的时序波形

```
entity DEVICE is
    port (X,Y: in Std_logic; Z: inout Std_logic);

end DEVICE;

architecture ALG of DEVICE is
begin
    PX: process(X)
    begin
        if X = '1' then
            Z<='1';
        end if;
    end process PX;
    PY: process(Y)
    begin
        if X = '1' then
            Z<=not Z;
        end if;
    end process PY;
end ALG;
```

上述 VHDL 源代码中存在错误，试指出该错误并写出正确的结构体。

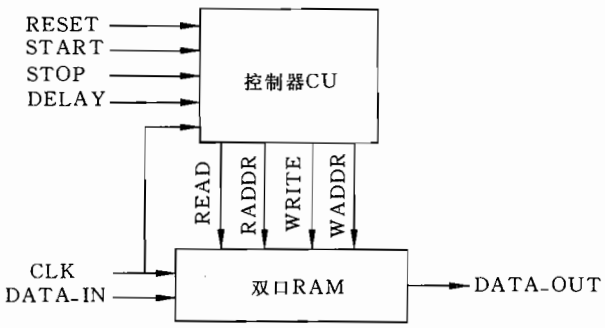
4.15　题图 4.4 给出了一个数据传输控制单元(DTCU)的方框图。当 R 为'1'时，该单元不工作。在 R 的下降沿，单元进入工作状态，这时它等待数据输入允许信号 IDAV 出现。在 IDAV 的上升沿，该电路单元从数据输入线 IDAT 读取数据，并把它传送到数据输出线 ODAT，与此同时，它还将数据允许信号 ODAV 置位为'1'。输出数据 ODAT 和允许信号 ODAV 保持一个时钟周期后会将 ODAV 复位为'0'。这时电路单元进入下一轮等待



题图 4.4　数据传输控制单元 DTCU

状态，等待 IDAV 的下一个上升沿。试写出该电路单元的系统级模型，模型应该使用 wait 语句控制非同步操作，且要保证同步操作(时钟操作)优先。

4.16　在某些计算机系统中(pipeline 式计算机)，数据通常以流(streams)的方式匹配传递。由于数据流可能有不同的来源，需要设计出可控制的延时单元用以控制数据流在指定时刻到达指定点。要求延时单元的延时量可被控制。用模拟电路实现延时显然不太可靠，用移位寄存器实现延时只适合于延时量较小的情况，采用双口随机存取寄存器(双口 RAM)则是一种较好的设计方案。这种方法的电路结构如题图4.5所示。对于双口



题图 4.5　延时量可控制的延时单元

RAM，可以同时对它进行读操作和写操作，只要不同时对同一个地址进行读操作和写操作，它就能正常工作。题图 4.5 所示的延时量可变的延时单元的工作过程如下：

RESET 信号对整个电路单元初始化，当 START 出现一个周期后，写地址(WADDR)被初始化为 0，读地址(RADDR)被初始化为 DELAY，即数据流被延时的时钟周期个数。在下一个时钟周期，第一个数据字出现在 RAM 的输入端，对于每个时钟周期，一组数据被写入 RAM，同时 RAM 写地址 WADDR 增加 1。对于前 DELAY 个时钟周期，每个时钟周期 RAM 读地址 RADDR 减 1，但不进行读操作。当 RADDR 减到 0 后，开始第一次读，每次读操作后，读地址 RADDR 增加 1。这样，经过最初延时，DATA_OUT 就是延时 DELAY 个时钟周期的 DATA_IN。当 STOP 变为'1'后，系统停止传输数据，但保持原来 RAM 中的数据。这是一个完全同步工作的电路系统，控制器和双口 RAM 全由同一时钟触发。可以把双口 RAM 看作是带同步时钟的寄存器阵列，并假定提供输入数据流的电路、产生 START 以及 STOP 的电路全由同一时钟触发。试按下述要求设计延时量可变的延时单元的算法模型。

(1) 写出控制模块和双口 RAM 的算法式模型。控制器应该用 while 循环和 wait 语句等造型方式。延时量为 0～15 个时钟周期，双口 RAM 应该使用尽可能少的定位电路实现这样的延时，数据字长为 4bit。

(2) 对算法模型进行仿真，要模拟不同的延时情况。

(3) 用基本门和触发器设计出控制器部分，对于控制器中对 WADDR 和 RADDR 进行与/或操作的部分，可以再写成一个 VHDL 算法模型。把控制器部分与 RAM 连接在一起组成混合层次的电路模型。

(4) 对上述混合层次的电路模型进行仿真验证。

(5) 写出报告。报告中应该包括设计过程和仿真结果。

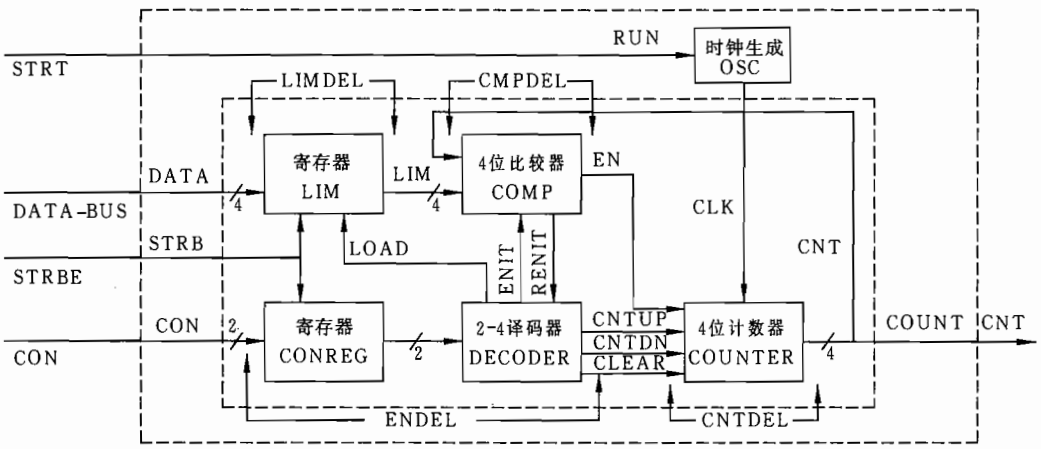
4.17 题图 4.6 示意给出了一个由两个模块组成的硬件框图，由时钟生成模块和受控计数模块组成。时钟生成模块产生 50%占空比的周期信号，周期为 PER ns。计数模块由 5 个逻辑模块组成：一个两位字长的控制寄存器 CONREG，一个 2-4 译码器 DECODER，一个 4 位比较器 COMP，一个 4 位计数器 COUNTER 和一个计数限制寄存器 LIM。整个计数模块接收 2 位字长的输入控制信号 CON，先把它存储在寄存器 CONREG 中，然后对控制信号译码，根据 CON 的不同取值，可以执行以下 4 种操作。

(1) CON="00"：计数器清零；

(2) CON="01"：将 DATA 上的数据加载到计数限制寄存器 LIM；

(3) CON="10"：计数器加计数，直到计数器中的数值与 LIM 中的数值相等为止；

(4) CON="11"：计数器减计数，直到计数器中的数值与 LIM 中的数值相等为止；

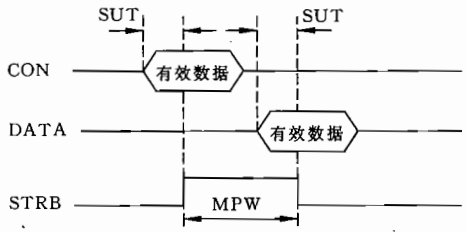


题图 4.6 两模块系统框图

对于计数器 COUNTER，如果接到计数指令且使能信号 EN='1'，则在时钟 CLK 的上升沿进行加计数或者减计数。计数指令译码时，除了送出计数指令之外，还将信号 ENIT 送入比较器 COMP，以将使能信号 EN 初始化为'1'。在开始计数之后，EN 的状态完全由比较器 COMP 控制。

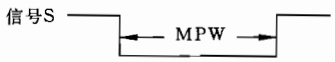
方框图中还给出了 4 个信号传输延时：ENDEL 是控制信号寄存器和译码器两部分的总延时，CMPDEL 是比较器的延时，LIMDEL 是计数限制寄存器的延时，CNTDEL 是计数器的延时。对接口信号也有时序要求，控制信号寄存器 CONREG 在闸门信号 STRB 的上升沿被加载，计数限制寄存器 LIM 在闸门信号 STRB 的下降沿被加载。要求控制信号 CON 和数据 DATA 相对闸门信号 STRB 的有效边沿必须有一个建立时间，其数值是 SUTns。闸门信号的最小宽度为 MPWns。题图 4.7 给出了对输入信号的时序要求。

画出该可控计数器模块的进程模型图，写出该电路的算法模型并对它进行仿真验证。



题图 4.7　输入信号的时序要求

4.18　题图 4.8 是一个'0'有效的输入脉冲，为了使它能可靠地激活它所驱动的器件，通常规定其最小脉宽不小于给定的时间(MPW ns)。VHDL 模型中通常需要报告信号脉宽违例的情况。试写出检测负脉宽违例的 VHDL 源代码。



题图 4.8　负脉冲有效的信号

4.19　解释下列 VHDL 源代码的功能。

```
entity DAVCON is
    generic(CLK_PER: TIME);
    port (R,IDAV: in Std_logic;
        IDAT: in NIT_VECTOR(7 downto 0);
        ODAV: out Std_logic;
        ODAT: out STD_LOGIC_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_LOO
```

第 5 章

5.1　假定一个 VHDL 数据流模型的保护模块中包含有下述两个并发赋值语句：

```
B: block (not CLK'Stable and CLK='1')
begin
    X<=guarded ADD(Y,Z);
    R<=guarded ADD(Q,S);
end block B;
```

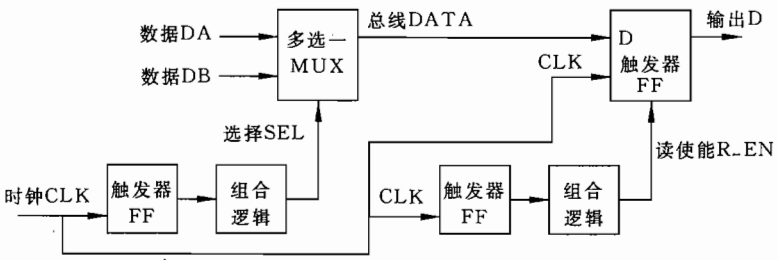
在 VHDL 模型中，这两个语句并发执行。但是，在实际硬件中，两次加法可以并行执行，也可以依次执行。如果依次执行两次加法，可以共用同一个加法器。假定信号 Y,Z,Q 和 S 是外部输入信号，X 和 R 是时钟驱动的寄存器。对于并行进行两次加法和依次执行两次加法，分别画出硬件方框图，图中应该给出必要的多选一电路、加法器和寄存器等。在所需硬件、完成运算所需时钟周期等方面，对两种硬件实现方案进行对比。

5.2　某个硬件的 VHDL 数据流描述包含有下述两条并发赋值语句：

```
C<=A;
D<=B;
```

假定 A,B,C 和 D 全是寄存器，它们之间的数据传递通过总线进行。并行数据操作需要两条总线，顺序数据操作只需要一条总线。请查阅 TTL 器件手册，找出可以用来实现宽度为 8 的总线的器件，用所选择的器件分别实现并行数据操作和顺序数据操作，比较两种情况下所需的硬件数目和完成数据操作所用的时间。

5.3　题图 5.1 是一个总线及其控制逻辑的示意图。选择信号 SEL 用来控制把数据 DA 或 DB 送上总线 DATA。如果在时钟 CLK 的上升沿读使能信号 R_EN 为'1'，则总线上的数据被送入输出寄存器。信号 SEL 和信号 R_EN 全由时钟驱动不同的触发器并经组合逻辑电路译码产生。画出信号 CLK,SEL,REN 和总线 DATA 上的信号的时序关系。指出哪个时序要求是最关键的时序要求？时序关系要考虑各触发器接收数据时的建立时间。



题图 5.1　总线及其控制逻辑

5.4　对于题 5.3 的硬件结构，写出相应的 VHDL 模型，模型要考虑对各种时序关系违例的情况进行检查，并对电路进行仿真验证。

5.5　试写出一段 URISC 程序，用来实现两个无符号二进制数的乘法。

5.6　在第 5 章讨论 URISC 处理器时，曾介绍 URISC 处理器是一种通用的处理器，即任何运算都可以通过足够长的 URISC 指令序列完成。是否可以用 URISC 指令实现布尔运算？写出实现布尔运算的 URISC 指令序列。

5.7　图 5.8 给出了 URISC 处理器的前 5 个状态的时序关系，试画出两个完整的从状态 0 到状态 8 的时序图，一个图表示减法运算得到负结果时的时序关系，另一个图表示减法运算得到非负结果时的时序关系。

5.8　试确定 URISC 处理器的最高工作频率。

(1) 写出一个表达式，把 URISC 的最大可能工作频率写成处理器的 VHDL 模型中的类属参数(generics)的函数。分别对微代码控制器和硬件连接控制器写出这个表达式。

(2) 对于你所选择的工艺条件(TTL，ECL 或 CMOS)，确定各类属参数的实际值，并估算 URISC 处理器的最大可能工作频率。

(3) 对 URISC 模型进行仿真，验证你所得出的结论。

5.9　对微代码控制器和硬件连接控制器的优缺点进行对比。

5.10　用 VHDL 语言写出 URISC 的模型框架中的下述器件模型：寄存器 R，寄存器 N，寄存器 Z，数据寄存器 MDR，地址寄存器 MDR，两相时钟。

5.11　用题 5.10 的结果完整地仿真验证 URISC 处理器的模型。

5.12　设计图 5.9 所示的 URISC 系统中的 RAM 和外部端口两个模块。

5.13　把 URISC 处理器的模型和 RAM 模型以及外部端口模型连接在一起，组成完整的 URISC 系统，用表 5.1 所示的 URISC 程序进行仿真验证。

5.14　在 5.3.7 节中讨论的 URISC 处理器的微代码控制单元中，控制处理器工作所需的各个控制信号都是直接从 ROM 中读出。这要求 ROM 输出的驱动能力足以驱动 URISC 控制器中的数据单元。通常，ROM 的输出只能驱动一个逻辑门，所以需要在 ROM 的输出到 URISC 数据单元的各个受控端之间加入缓存器。修改微代码式控制单元的 VHDL 描述，使在 ROM 输出到数据单元之间加入指令寄存器(MIR)，讨论对系统时钟的时序要求。

5.15　修改第 5.3.8 节中给出的硬件连接式控制单元，使每个控制状态全由一个触发器构成。修改后的控制单元与原控制单元相比有哪些优点？

第 6 章

6.1　对于 6.1.1 节中在讨论器件 COM(二进制比较器)时，所给出的基本 case 条件语句模型进行改进，按下列要求写出改进后的 case 条件语句模型，并对改进后的 case 条件语句模型进行仿真。

(1) 在多路选择器的选择条件中消去变量 N1。

(2) 在多路选择器的选择条件中消去变量 N0。

(3) 在多路选择器的选择条件中消去变量 M1。

6.2　对于题 6.1(1)中写出的 case 条件语句式 VHDL 模型，画出多路选择器式硬件

结构。

6.3 下列 VHDL 语句对应于“和项之积”式的逻辑表达式，利用第 6 章学到的知识，把它们转换为只用 NOR 运算实现的 VHDL 语句。

(1) 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);

(2) X<=(not A or not C) and (B or C) and (A or nor B or not C);

(3) Z<=A and (not B or C or D) and not E;

(4) P<=not A and not B;

6.4 以 6.1.2 节讨论的数据流式 VHDL 模型 NORDF 为例，讨论如何自动把 VHDL 数据流模型转换为逻辑门的互连网络。网络中只用两级或少于两级 NOR 门。这里所谓两级并不排斥使用反相器对输入信号反相，且信号反相器不包括在两级之内。

利用 6.1.1 节给出的把 VHDL 数据流模型自动转换为逻辑门级模型的算法，用 NOR 门和反相器网络实现 6.1.2 节的数据流式模型 NORDF。

写出上述逻辑门互连网络的结构式 VHDL 模型，并对它进行仿真。

6.5 对于题 6.3 中的 VHDL 语句，利用题 6.4 中给出的算法，转换为两级 NOR 门和反相器互连组成的网络。

6.6 图 6.8 中总结了组合逻辑电路设计的过程。这种总结强调的是用“和项之积”实现电路。试画出另一张图，总结组合逻辑电路的设计过程，但强调使用“乘积项之和”实现电路。

6.7 写出图 6.5 中给出的二进制比较器 COM 的“乘积项之和”式的数据流模型，并对模型进行仿真。

6.8 写出一组规则，按照它们可以把“乘积项之和”的逻辑表达式转换为由 NAND 运算组成的 VHDL 数据流模型。

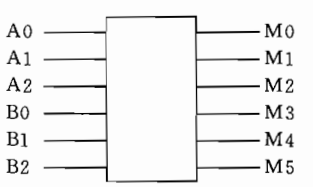
6.9 利用题 6.8 中给出的规则，写出二进制比较器 COM 的由 NAND 运算组成的 VHDL 数据流模型。

6.10 写出一组规则，利用它们可以自动把 NAND 运算组成的 VHDL 数据流模型转换为只包括 NAND 门和反相器的两级或两级以下逻辑门的互连网络。与 6.1 节讨论 NOR 门互连网络时的情况类似，网络中对输入信号反相不计算在两级之内。

利用前面给出的规则，把二进制比较器 COM 的 NAND 运算 VHDL 数据流模型转换为只包括 NAND 门和反相器的逻辑门互连网络。

写出上述逻辑门互连网络的结构式 VHDL 模型，并对它进行仿真。

6.11 设计一个硬件乘法器(M)，用来计算两个 3 位二进制数据 A 和 B 的乘积。题图 6.1 给出了器件 M 的方框图。图中 A2A1A0 表示第一个二进制输入，B2B1B0 表示第二个二进制输入，M5M4M3M2M1M0 表示乘积输出。例如，如果 A2A1A0 = 110，B2B1B0 = 011，则 M5M4M3M2M1M0=010010。



题图 6.1 三位二进制乘法器

根据图 6.8 总结出的组合逻辑电路设计过程，从写出乘法器的指标规范入手，直到得出乘法器的 ROM 实现为止。要求：

(1) 对于从最高层次的指标描述到 ROM 式的硬件实现的每一步变换过程，写出详细的说明。

(2) 对于乘法器的每一种描述都做详细的仿真验证。

(3) 画出最后实现的电路的方框图并标出所有信号名。

(4) 用 FPGA 或硬件仿真的方式验证设计的正确性。

6.12　根据组合逻辑电路设计的流程图 6.8，从硬件的指标描述开始到标准多路选择器实现硬件为止，用标准多路选择器设计题 6.11 的乘法器。设计要求与题 6.11 相同。

6.13　根据组合逻辑电路设计的流程图 6.8，从硬件的指标描述开始到改进多路选择器实现硬件为止，用改进多路选择器设计输入信号为 3 位二进制数据的乘法器。按与题 6.11 相同的要求完成电路设计过程。

6.14　进行传感器接口电路设计。电传感器的输出值发生变化时，过渡时间通常很长。假定一个传感器的输出采用 4 位 BCD 编码信号表示其输出，且假定该传感器的输出值要从 7(0111)变为 8(1000)，即所有 4 位数据全发生变化。对于实际电路，由于所有 4 位数据不可能完全同时变化，在传感器的输出值从 7 变到 8 的过程中，传感器输出值可能为下述序列：

0111 →0110 →0100 →0000 →1000

当然也可能是其他输出序列。除此之外，传感器测量的物理量(比如温度)可能在 7 和 8 的边界上保持相当长的时间，在这期间传感器的输出值可能为上述序列中的任意值。当把传感器的输出值送入计算机进行处理时，通常是每隔一定的时间间隔采样一次(比如 1 秒)。如果采样发生在传感器输出值发生变化期间，计算机采样得到的数值就会是一个错误值。实际上，当传感器的实际值在 7～8 之间时，计算机采样得到的值可能是 0～15 之间的任何数值。

对传感器的输出采用格雷(Gray)码编码可以避免这种问题。题表 6.1 给出了格雷码与 BCD 码的对应关系。按照格雷码，7 的编码为 1011，8 的编码为 1001。传感器输出从 7 变为 8 的过程中只有一位数据发生变化。计算机对传感器的输出进行采样时要么得到 7，要么得到 8，这就避免了采用 BCD 码时可能出现的问题。

信号被采样进入计算机之后，数据处理时使用 BCD 码比较方便。为了适应这种需要，应该设计一个格雷码到 BCD 码的变换电路。该电路的输入是 4 位格雷码，输出是 4 位 BCD 码。对于非法输入组合(0100，0101，0111，1100，1101，1111)可以不作处理，即当作“无关态”处理。根据图 6.8 以及题 6.6 总结的组合逻辑电路设计过程，任意选择下列路径之一，完成这个格雷码到 BCD 码转换电路的设计。

(1) 先写出器件的自然语言性能描述，然后沿从自然语言描述到由 ROM 构成的电路图描述的路径设计电路。

(2) 从器件的自然语言性能描述出发，设计由标准多路选择器构成的电路原理图。

(3) 从器件的自然语言性能描述出发，设计由简化的多路选择器构成的电路原理图。

题表 6.1　格雷码与 BCD 码的对应关系

十进制数	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

(4) 从器件的自然语言性能描述出发，设计由两级 OR-AND 逻辑门构成的电路原理图。

(5) 从器件的自然语言性能描述出发，设计由两级 OR-AND 器件构成的结构式 VHDL 电路模型。

(6) 从器件的自然语言性能描述出发，设计由两级 NOR-NOR 逻辑门构成的电路原理图。

(7) 从器件的自然语言性能描述出发，设计由两级 NOR-NOR 器件构成的结构式 VHDL 电路模型。

(8) 从器件的自然语言性能描述出发，设计由两级 AND-OR 逻辑门构成的电路原理图。

(9) 从器件的自然语言性能描述出发，设计由两级 AND-OR 器件构成的结构式 VHDL 电路模型。

(10) 从器件的自然语言性能描述出发，设计由两级 NAND-NAND 逻辑门构成的电路原理图。

(11) 从器件的自然语言性能描述出发，设计由两级 NAND-NAND 器件构成结构式 VHDL 电路模型。

对于选定的路径，按下述要求完成设计。

(1) 对于从硬件的一种描述转换到另一种描述过程中的每一步，都要写出详细说明。

(2) 对设计过程中得到的每一种电路描述，都要做详细的仿真。

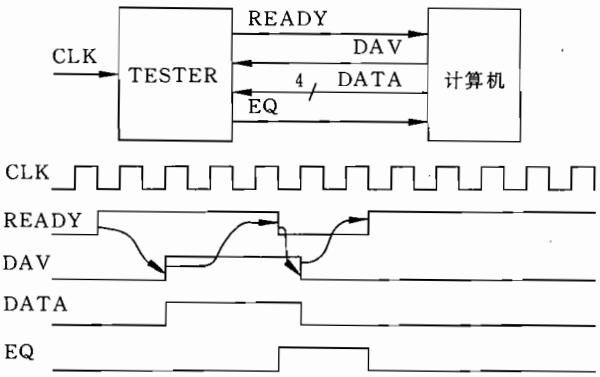
(3) 给出最后得到的电路原理图，并标出所有信号名。

(4) 用 FPGA 或其他方法验证设计的正确性。

6.15　改进题 6.14 设计的格雷码到 BCD 码转换器。在题 6.14 设计的格雷码到 BCD 码转换器中增加一个输出端口 E，当输入码为非法输入时，输出 E 为高电平。

6.16　设计一个时序电路 TESTER，该电路从计算机接收 4bit 宽的数据序列，如果当前从计算机传输到 TESTER 的数据与前两次传输到 TESTER 的数据相等，则 TESTER 给出信号 EQ。计算机和 TESTER 全是同步时序电路，但它们分别由不同的时

钟驱动，不论计算机的时钟比 TESTER 的时钟速度快还是慢，TESTER 都应该能够正常工作。题图 6.2 给出了电路 TESTER 的接口及时序关系。



题图 6.2　电路 TESTER 的接口及时序关系

每当电路 TESTER 完成了对当前输入数据的处理之后，就将信号 READY 置位，以通知计算机它已经作好接收数据的准备。计算机接收到握手信号 READY 之后，把 4 位数据送上数据线 DATA，同时将数据有效信号 DAV 置位。器件 TESTER 接收到数据有效信号 DAV 之后，等待至少一个时钟周期，然后把 DATA 送来的数据传入内部寄存器。在 TESTER 完成内部数据传递之后的那个时钟周期，把握手信号 READY 复位为 0，用以通知计算机数据传递过程已经完成。计算机从把数据送上数据线 DATA 后直到器件 TESTER 通知数据传递成果为止，应该一直保持数据线上的数据不发生改变。计算机检测到 READY 信号复位之后，可以清除数据有效标志 DAV，并清除数据线上的数据。

在 TESTER 对信号 READY 复位，通知计算机数据传递过程结束的同时，TESTER 还要根据输入数据的值确定信号 EQ 的值。如果 DATA 上输入的数据与前两次到来的数据相同，则设定 EQ 为逻辑 1，否则设 EQ 为逻辑 0。计算机检测到信号 READY 的下降沿之后，应该先存取 EQ 的值，后将数据有效信号 DAV 复位。

TESTER 检测到信号 DAV 下降之后，可以清除信号 EQ 上的值，并对握手信号 READY 再次置位，以接收下一个数据输入。为了数据的安全，规定 DAV 下降 1 个时钟周期后，才清除 EQ 的数据，并再次对 READY 置位。

根据本章中讨论的内容，按下述顺序设计电路 TESTER。

(1) 建立状态表。

(2) 写出电路的系统级 VHDL 算法模型。

(3) 写出对电路 TESTER 进行仿真验证的输入信号序列，对 TESTER 的 VHDL 算法模型进行仿真验证。

(4) 根据电路的 VHDL 算法模型，把电路分解成控制单元和数据单元，并采用硬连接方法设计数据单元和控制单元。要求对设计的电路和设计过程给出详细的说明。

(5) 讨论采用自动化的方法完成上述设计过程的可能性。

(6) 用 FPGA 或其他方法验证电路设计。

6.17　对于题 6.16 的电路 TESTER，采用微代码控制器 BMCU 设计其控制单元。设计内容包括：

(1) 按 6.3.3 节中讨论过的微代码控制器综合过程，对于电路 TESTER 设计 BMCU 中的 ROM 程序。

(2) 画出由 BMCU 构成控制单元的电路 TFSTER 的方框图。

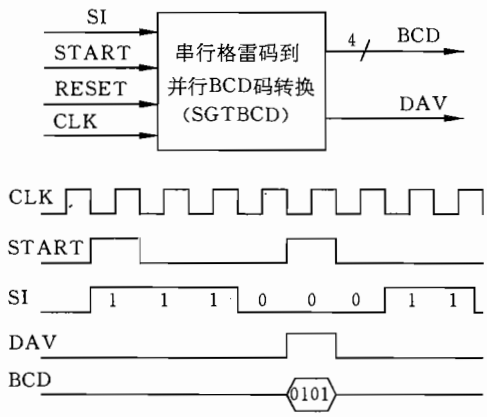
(3) 对于用 BMCU 构成控制单元的电路 TESTER，写出其 VHDL 系统级模型，模型中包括数据单元和控制单元。

(4) 写出仿真验证电路 TESTER 的输入信号序列，对 TESTER 的上述系统级模型进行仿真验证。

(5) 将系统级模型转换为硬件原理图。详细说明转换过程中的每一步，并讨论自动转换的可能性。

(6) 用 FPGA 或其他硬件所设计的电路，并采用与系统级仿真时同样的输入信号对硬件进行实验测量。对实验结果进行分析，如果仿真结果与实验结果不同，分析说明造成这种不同的原因。

6.18　采用状态表法，设计一个将 4 位串行格雷码输入信号转换为 4 位并行 BCD 码输出的电路。电路的接口描述以及时序关系如题图 6.3 所示。



题图 6.3　串行格雷码输入转换为并行 BCD 码输出的电路

关于格雷码的编码，可以见题 6.14。所设计的串行格雷码到并行 BCD 码转换电路 SGTBCD 的系统时钟输入信号为 CLK，启动信号 START 与串行格雷码输入 SI 同步，START 的上升沿与格雷码的最高位同时到来，此后信号线 SI 上依次出现格雷码输入的后 3 位，最后出现的是最低位。在 4 位格雷码输入之后的最后一个时钟周期，器件应该同时输出并行 BCD 码。在最后一位格雷码输入后的下一个时钟周期或者该时钟周期后的任何时刻，器件都可以继续接收数据输入。

6.19　设计一个同步格雷码计数器，计数器从 0 开始计数，数到 9 后循环回 0。分别采用题 6.16 和 6.17 的设计方法进行设计。

6.20　利用状态表法，设计一个 Mealy 序列检测器，用来检测输入信号中出现连 1

的情况。该检测器有一个串行输入端口 X 和一个串行输出端口 Z，如果输入序列是 4 个连 1，则输出为 1，其他情况下输出为 0。输出 1 的时刻与第 4 个 1 输入的时间相同。下面给出了输入/输出序列的一个实例：

X　0 1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 1 0 1

Z　0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 1 1 0 0

6.21　利用状态表法，设计一个 Mealy 序列检测器，用来检测输入信号中出现特定序列的情况。该检测器有一个串行输入端口 X 和一个串行输出端口 Z，如果输入序列是 1100 或 1001，则输出为 1，其他情况下输出为 0。输出 1 的时刻与被检测序列的第 4 位输入同步。下面给出了输入/输出序列的一个实例。

X　1 1 0 0 1 0 0 1 1 0 1 1 0 0 1 1 0 0 0

Z　0 0 0 1 1 0 0 1 0 0 0 0 0 1 1 0 0 1 0

6.22　利用状态表法，设计一个 Moore 序列检测器，用来检测输入信号中出现特定序列的情况。该检测器有一个串行输入端口 X 和一个串行输出端口 Z，如果输入序列是 1100 或 1001，则输出为 1，其他情况下输出为 0。与题 6.21 不同之处在于：输出 1 的时刻比被检测序列的第 4 位输入延时了一个时钟周期。下面给出了输入/输出序列的一个实例：

X　1 1 0 0 1 0 0 1 1 0 1 1 0 0 1 1 0 0 0 -

Z　0 0 0 0 1 1 0 0 1 0 0 0 0 0 1 1 0 0 1 0

6.23　4 位数据组成的字按块传递，每块数据可以为 2～15 个字。每个数据字中的 4 位数据中，最高位是偶校验位，低 3 位是有用信息，偶校验位为 1 或 0 取决于低 3 位数据中 1 的个数，即一个数据字（包括校验位）中 1 的个数总是偶数。在 2～15 个数据字组成的数据块中，最后一个数据字是偶校验字。偶校验字的数值与该数据块中的数据有关，其作用是保证数据块中的数据（包括校验字）中各列数据中 1 的个数为偶数。下面的例子说明了这种数据结构。在 word0 中，数据为 011，由于 3 位数据中 1 的个数为偶数，所以 word0 的偶校验位（最高位）为 0。在 word1 中，数据为 001，由于 3 位数据中 1 的个数为奇数，所以 word1 的偶校验位（最高位）为 1。word4 为校验字，可以检验题表 6.2 每列数据中 1 的个数是否为偶数。

题表 6.2　数据字

	偶校验位	数据位 2	数据位 1	数据位 0	
word 0	0	0	1	1	
word 1	1	0	0	1	
word 2	1	1	1	1	
word 3	0	0	0	0	
word 4	0	1	0	1	
					偶校验字

按这种格式传递数据，数据块的每一行中 1 的个数全为偶数，每一列中 1 的个数也为偶数。如果接收到一个数据块，利用数据块中数据各行、各列中 1 的个数全为偶数的信息可以检查和纠正数据传输中出现的错误。假定数据传输过程中一个数据块中恰好有 1 位数据发生错误，从 1 变成 0 或从 0 变成 1，则误码所在的数据行和列的偶性质将会被破坏，这样就可以知道误码出现在数据块中的哪一行、哪一列，从而可以发现并纠正这种错误。例如：假定题表 6.2 所示的数据块在传输中 word2 的数据位 1 发生了错误，接收到的数据如题表 6.3 所示。表中最右边一列是检查对应的数据字的奇偶性得到的结果，称为行校验矢量。如果某数据字中 1 的个数为偶数，则行校验矢量中对应位为 0；如果某数据字中 1 的个数为奇数，则行校验矢量中该位为 1。表中最下面一行是对数据块中对应的数据列作奇偶检查的结果，称为列校验矢量。如果数据块中一列数据中 1 的个数为偶数，列校验矢量中相应位为 0；如果数据块中一列数据中 1 的个数为奇数，则列校验矢量中该位为 1。由于行校验矢量中只有对应 word2 的 1 位为 1，所以误码一定出现在 word2。同时由于列校验矢量中只有对应数据位 1 的一列为 1，所以可以知道误码是 word2 的数据位 1。

题表 6.3　word2 发生错误的数据字

					行校验矢量
word 0	0	0	1	1	0
word 1	1	0	0	1	0
word 2	1	1	0	1	1
word 3	0	0	0	0	0
word 4	0	1	0	1	0
列校验矢量	0	0	1	0	

一般地讲，可以先根据行校验矢量找出出现误码的数据字，再把发生误码的数据字与列校验矢量按位做异或运算，就可以得到正确的数据字。对于上面给出的例子，纠正误码的过程如题表 6.4 所示。

题表 6.4　纠正误码过程

出现误码的数据字	1101
列校验矢量	0010
XOR	—
误码校正之后的数据字	1111

假定数据块传递过程中恰好有 2 位发生误码，如果两位误码出现在同一数据字，则行校验矢量中的相应位为 0，即行校验矢量不能发现恰好有两个误码的数据字，但这种情况下列校验矢量中一定会有两个 1，即根据列校验矢量可以知道发生了两个误码。如果两个误码出现在不同数据字的同一列，则列校验矢量不能发现两个误码出现在哪一列，但这时行校验矢量中一定会出现两个 1，即根据行校验矢量可以知道发生了两个误码。如果两个

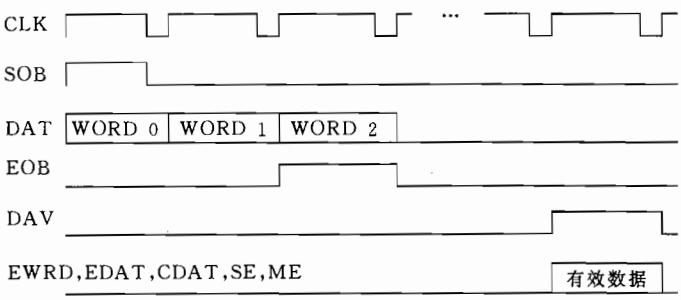
误码出现在不同数据字的不同列，则行校验矢量和列校验矢量会同时指出两个误码所在的行和列，这种情况下可以校正两个误码。这就是说，采用这种偶校验方法传递数据块，可以纠正 1 位误码，可以发现 2 位误码。

如果数据传递过程中出现 2 位以上的误码，则不一定能检测出数据块中出现了误码。对于特定情况，即误码发生在数据块中某些特定位置时，可以检测到误码的存在甚至可以纠正多位误码。但对于其他更多的情况，则可能发现不了误码的存在。

下面给出了检查数据块的奇偶性的电路的指标说明，并给出了电路的时序关系和设计要求，按要求设计题图 6.4 电路。

题图 6.4　检查数据块奇偶性的电路方框图

题图 6.4 给出了电路的接口。DAT 为 4 位数据输入，SOB 是数据块开始标志，EOB 是数据块结束标志，所有数据输入全由系统时钟同步，SOB 和 EOB 的长度全为一个时钟周期。如果数据块中只有一个误码，则用输出信号 SE 为 1 来说明这种情况，同时 4 位矢量 EWRD 给出一个数字指明误码出现在什么地方。比如：如果误码出现在 word2，则 EWRD＝0010。4 位信号 EDAT 用来输出发生误码的数据字，4 位数据 CDAT 用来输出误码校正后的数据字。如果行校验矢量或列校验矢量中不只一位为 1，则认为数据块中出现了多个误码，多误码情况发生时输出信号 ME 为 1。当电路的其他 5 个输出信号要输出有效数据时，输出有效标志信号 DAV 为 1。题图 6.5 给出了数据块由 3 个数据字组成时的时序关系。图中系统时钟 CLK 序列中的省略号表示一定的时钟周期，在几个时钟周期中，数据块奇偶性校验电路完成对输入数据块的校验，计算各个输出信号值。输出信号有效标志 DAV 出现正脉冲，正脉冲保持一个时钟周期，此后电路恢复到初始状态，准备接收下一个数据块。在 DAV 恢复为 0 之后的任何时钟周期，都可以接收数据块开始标志 SOB 以及数据本身。



题图 6.5　检查成数据块奇偶性的电路的时序关系

对题图 6.4 电路的设计要求如下：

(1) 写出电路 BLK_PARITY_CHK 的系统级 VHDL 算法模型。

(2) 对该 VHDL 算法模型进行仿真验证。

(3) 采用硬连接方式，把 VHDL 算法模型转换为数据单元和控制单元。写出转换过程以及转换得到的硬件的详细说明。

(4) 讨论实现自动从算法模型转换到硬件的可能性。

(5) 用适当的器件对所设计的电路进行硬件验证。

6.24 对于题图 6.4 电路 BLK_PARITY_CHK，利用微代码控制器 BMCU 设计一个控制单元，要求按下述步骤设计电路。

(1) 采用 6.3 节中讨论过的综合过程设计 BMCU 的微代码程序，即 ROM 中的数据。

(2) 画出电路 BLK_PARITY_CHK 的方框图，包括用 BMCU 组成的控制单元，也要包括数据单元和信号线名。

(3) 写出电路 BLK_PARITY_CHK 的系统级 VHDL 算法模型，包括由 BMCU 组成的控制单元模型以及数据单元的模型。

(4) 对这个 VHDL 模型进行仿真，要求对各种长度的输入数据块进行仿真验证。

(5) 把系统级 VHDL 算法模型转换为硬件。详细解释转换过程，对得到的硬件电路详细说明，并讨论实现自动转换的可能性。

(6) 用可能得到的器件实现电路设计。采用与仿真时相同的输入序列，检查仿真结果与硬件实验的结果有什么不同？如果发现不同，则解释产生不同的原因。

6.25 对于图 6.24 定义的 4 分支地址生成方法，设计按该方法生成 ROM 地址的微代码控制器 4BMCU。设计须按下述步骤进行。

(1) 参考基本微代码控制器 BMCU 的方框图(图 6.16)，画出 4BMCU 的方框图。

(2) 写出 4BMCU 的系统级 VHDL 算法模型。要求可以利用这个模型设计具体应用电路中的控制单元。

(3) 设计 4BMCU 中的地址生成电路。

6.26 对于第 6 章中曾详细讨论过的串并转换电路 STOP，利用 4BMCU 设计其控制单元。

6.27 利用微代码控制器 4BMCU，重复题 6.17 的设计工作。

6.28 利用微代码控制器 4BMCU，重复题 6.24 的设计工作。

第 7 章

7.1 对于下面的表达式，画出其数据流图。

$$X := (A + C) * (B + D)$$

7.2 下面给出了计算两个数的最大公因子的另一种代码，试画出其控制流图。

```
L1: loop
    X := Xi;                    -- (1)
```

```
    Y := Yi;                          -- (2)
    L2: while (X /= Y) loop           -- (3)
        if (X < Y)                    -- (4)
            then Y := Y-X;            -- (5)
        else X := X-Y;                -- (6)
        end if;
    end loop;              -- L2
    OUT := X;                         --(7)
end loop;                  -- L1
```

7.3 对于题 7.2 所述代码，试画出其控制数据流图。

第 8 章

8.1 对于 8.7 节中的实体 FSMEX1 及其结构体 FSM，增加启动信号以及运算结束标志信号，使器件更为有用。对改进后的 VHDL 模型进行仿真验证。

8.2 下面给出了一段 VHDL 代码：

```
entity SCHED1 is
    port (A,B,C,D,E,F: in INTEGER;
          X,Y: out INTEGER);
end SCHED1;

architecture HIGH_LEVEL of SCHED1 is
begin
    X<=F*(A + B + C*D + D*E);
    Y<=(A*B + E)* * C;
end HIGH_LEVEL;
```

假定对实现硬件时所使用的硬件没有限制，按下述要求完成调度。

(1) 画出数据流图。

(2) 采用 ASAP 法调度。

(3) 采用 ALAP 法调度。

(4) 采用排队调度技术完成调度。

(5) 采用自由度法完成调度。

8.3 对于下面的约束条件，分别重复题 8.2 的调度过程。

(1) 限制使用一个加法器、一个乘法器。

(2) 限制使用一个加法器、两个乘法器。

(3) 限制使用两个加法器、一个乘法器。

(4) 限制使用两个加法器、两个乘法器。

8.4　题 8.2 中 VHDL 源代码模型的两个输出信号 X 和 Y 的表达式中有公共项 C * D，是否可能通过数学变换简化题 8.2 和 8.3 的各个调度方案？是否可以采用其他方法化简这些调度方案？

8.5　下面 VHDL 源代码给出了某电路的行为描述：

```
entity SCHED2 is
    port (A,B,C,D,E,F: in INTEGER;
          X,Y: out INTEGER);
end SCHED2;

architecture HIGH_LEVEL of SCHED2 is
    signal Z: INTEGER;
begin
    X<=Z * (A-B);
    Y<=(A * B) + Z;
    Z<=C * D + (E + F) / D;
end HIGH_LEVEL;
```

假定对实现硬件时所使用的硬件没有限制，按下述要求完成调度。

(1) 画出数据流图。

(2) 采用 ASAP 法调度。

(3) 采用 ALAP 法调度。

(4) 按关键路径的优先级排队完成调度。

8.6　对于下面的约束条件，分别重复题 8.5 的调度过程。

(1) 只使用一个加法器、一个减法器、一个乘法器和一个除法器。

(2) 假定加法和减法由同一个硬件单元(ADD_S)完成，乘法和除法也由同一个硬件单元(M_D)完成，并假定只能使用一个 ADD_S 和一个 M_D。

(3) 假定加法和减法由同一个硬件单元(ADD_S)完成，乘法和除法也由同一个硬件单元(M_D)完成，并假定可以使用两个 ADD_S 和两个 M_D。

(4) 假定所有运算全由 ALU 模块完成，并假定可以使用两个 ALU。

8.7　对于图 8.25 中的各个多路选择器，增加所需的选择控制信号。对图中的各个寄存器加上加载数据的控制信号。复习第 7 章学过的内容并利用实体 FSMEX1 的结构体 FSM，设计出该电路的控制单元。

8.8　对于题 8.2 中完成的最好的调度方案，采用 Greedy 算法分配寄存器、运算单元和数据通路。要求完成：

(1) 画出与图 8.21 类似的寄存器及运算单元分配图。

(2) 画出与图 8.22 类似的数据通路分配图。

(3) 构造 Dantt 表并讨论器件利用率。

(4) 与第 8 章中实体 FSMEX1 的结构体 FSM 类似，写出该电路的控制单元的

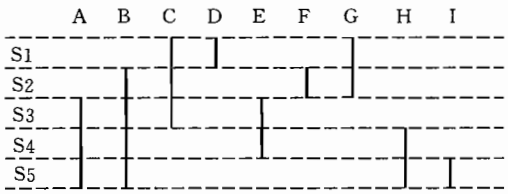
VHDL 有限状态机模型，并对模型进行仿真。

(5) 对于所画出的分配图中的多路选择器和寄存器，分别增加数据选择控制信号和数据加载控制信号。利用第 7 章中学过的知识，设计这个控制单元。

8.9 对于下列每一个调度方案，重复题 8.8 的各项内容。

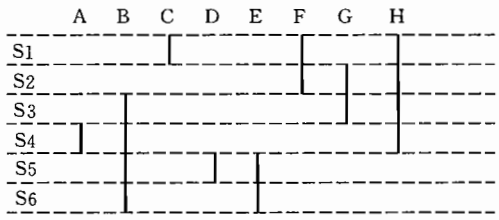
(1) 题 8.3(1)； (2) 题 8.3(2)； (3) 题 8.3(3)； (4) 题 8.3(4)； (5) 题 8.5 中最好的调度方案； (6) 题 8.6(1)； (7) 题 8.6(2)； (8) 题 8.6(3)； (9) 题 8.6(4)。

8.10 对于题图 8.1 所示的数据生存时间图，采用左边算法分配寄存器。



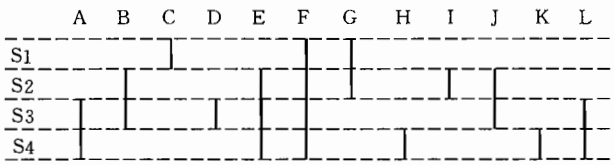
题图 8.1 数据生存时间图(1)

8.11 采用左边算法，为题图 8.2 的数据生存时间图分配寄存器。



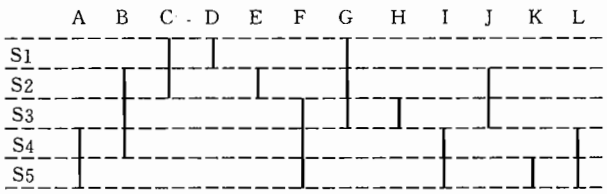
题图 8.2 数据生存时间图(2)

8.12 采用左边算法，为题图 8.3 的数据生存时间图分配寄存器。



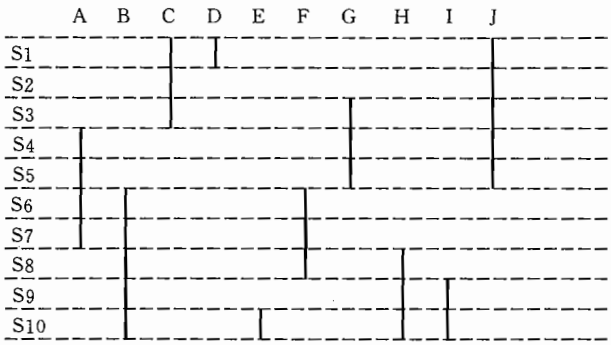
题图 8.3 数据生存时间图(3)

8.13 采用左边算法，为题图 8.4 的数据生存时间图分配寄存器。



题图 8.4 数据生存时间图(4)

8.14　采用左边算法，为题图 8.5 的数据生存时间图分配寄存器。



题图 8.5　数据生存时间图(5)

8.15　采用左边算法，为题图 8.6 的数据生存时间图分配寄存器。

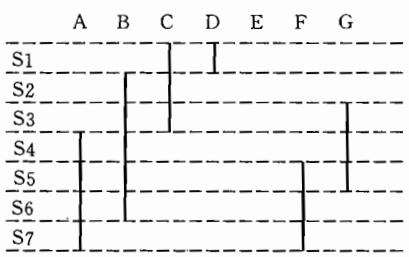
8.16　对于题 8.2 中得到的最好调度方案，采用左边算法分配寄存器，采用连接表法分配运算单元和连接路径(参见表 8.10)。要求：

(1) 给出如表 8.10 所示的连接表。

(2) 给出如图 8.29 所示的硬件连接图。

(3) 构造 Gantt 表并讨论器件利用率。

(4) 对于控制单元写出 VHDL 有限状态机模型，并对它仿真。



题图 8.6　数据生存时间图(6)

(5) 对于分配图中的多路选择器和寄存器，分别增加数据选择控制信号和数据加载控制信号。利用第 7 章中学过的知识，设计这个控制单元。

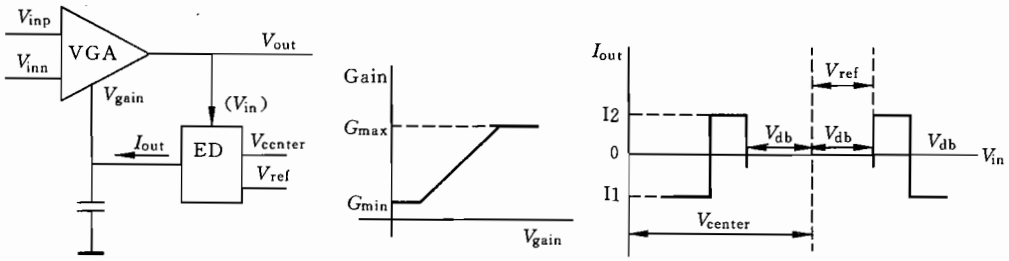
8.17　对于下列每一个调度方案，重复题 8.16 的各项内容。

(1) 题 8.3(1)；　(2) 题 8.3(2)；　(3) 题 8.3(3)；　(4) 题 8.3(4)；　(5) 题 8.5 中最好的调度方案；　(6) 题 8.6(1)；　(7) 题 8.6(2)；　(8) 题 8.6(3)；　(9) 题 8.6(4)。

第 9 章

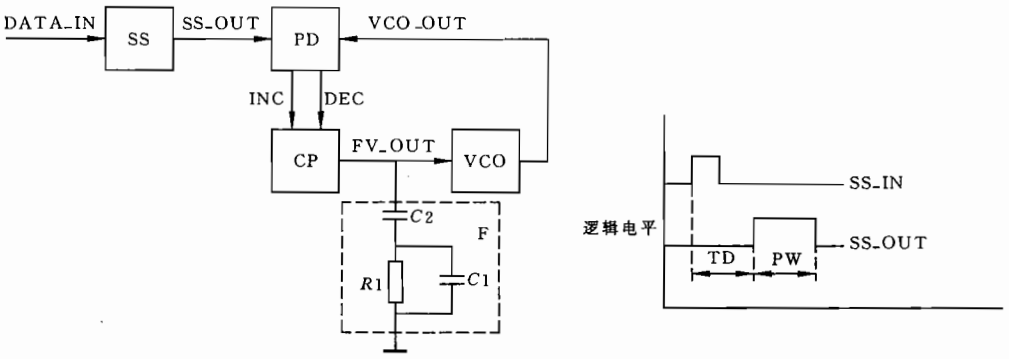
9.1　题图 9.1(a)是一个基本自动增益控制环路，由可变增益放大器(VGA)和包络检测电路(ED)组成，VGA 和 ED 的传输特性分别如题图 9.1(b)和(c)所示。试分别写出 VGA，ED 和电容模块的 VHDL 模型，把它们互连构成自动增益控制环的模型，并进行仿真验证。

9.2　题图 9.2 给出了基本锁相环的原理框图，并通过各模块的传输函数或原理图完整地定义了锁相环的行为。锁相环有延时脉冲形成(SS)、相位检测(PD)、电荷泵电路(CP)、压控振荡器(VCO)以及电容构成的二阶滤波器(F)构成。按下述要求写出各个模块的 VHDL 模型，将其互连构成锁相环的模型并对模型进行仿真验证。

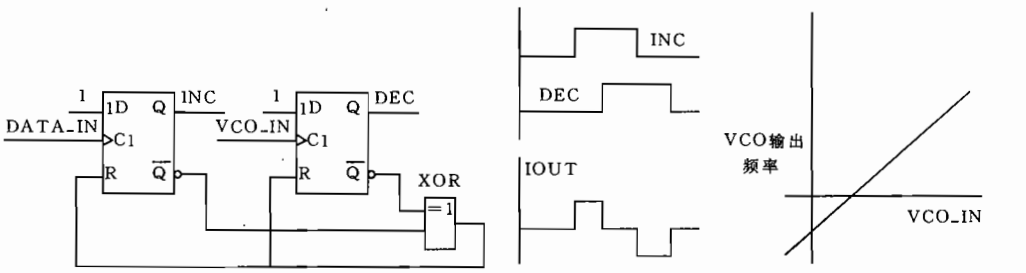


(a) 自动增益控制环路　　(b) VGA的传输特性　　(c) ED的传输特性

题图 9.1　基本自动增益控制环路



(a) 锁相环原理框图　　(b) 延时脉冲形成电路SS的传输特性



(c) 相位检测电路的原理图　　(d) 电流泵的传输函数　　(e) 压控振荡器的传输函数

题图 9.2　基本锁相环的原理框图及行为定义

(1) 利用 VHDL 语言写出二阶滤波器的行为模型。

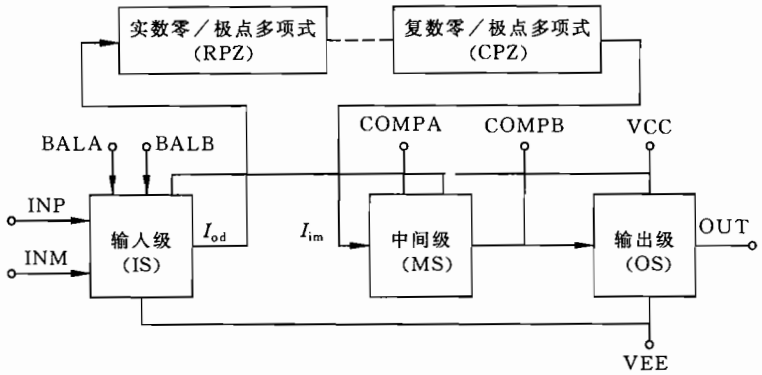
(2) 利用 VHDL 写出压控振荡器的行为模型。

(3) 根据题图 9.2(b)给出的延时脉冲形成电路的时序波形，设计其电路，用 VHDL 写出其结构模型。

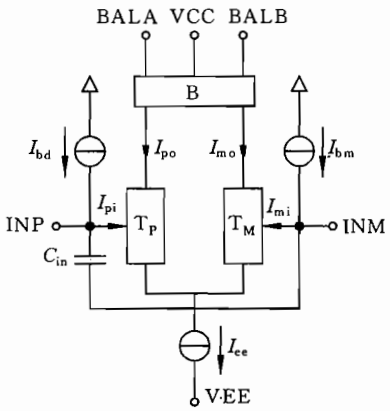
(4) 根据题图 9.2(c)给出的相位检测电路的原理图，分析电路行为，用 VHDL 语言写出其行为模型。

(5) 设计电荷泵的电路，分别写出其结构模型和行为模型。

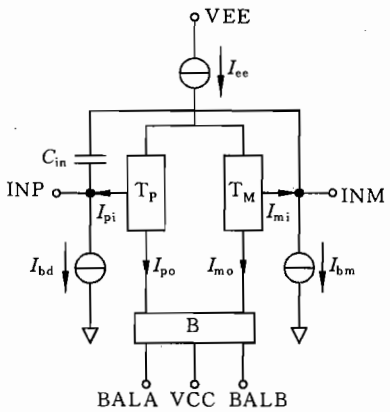
(6) 将各个模块互连，得到锁相环的行为模型，并对其进行仿真验证。



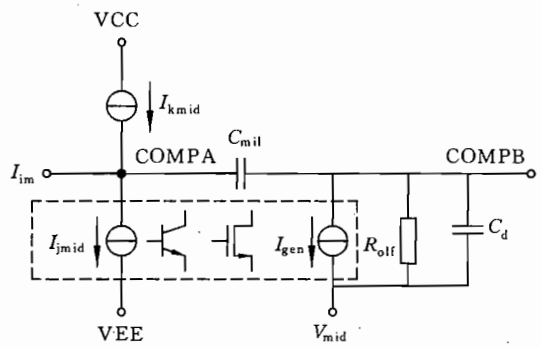
题图 9.3　运算放大器的行为模型的基本结构



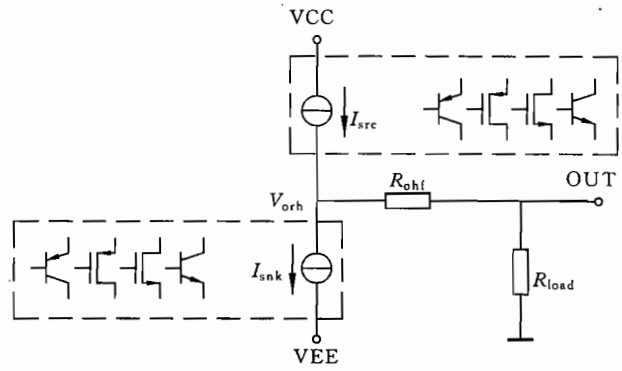
(a) N型输入级



(b) P型输入级



(c) 中间级的简化图



(d) 输出级的简化图

题图 9.4　运算放大器中不同模块

9.3　运算放大器是许多集成系统中的重要器件，行为模型是描述运算放大器功能的重要手段。运算放大器的行为模型通常由一组代数微分方程描述，模型应用方便，不局限于特定的仿真器。运算放大器的行为模型的基本结构如题图 9.3 所示，其中的输入级、中间级以及输出级的电路结构如题图 9.4 所示。利用所学的关于运算放大器的知识，写出运算放大器的行为模型。

参考文献

1. Jerraya A A, Ding H, Kission P et al. Behavioral Synthesis and component reuse with VHDL. Netherlands: Kluwer academic Publishers, 1997
2. Technical Report of the Semi Conductor Industry Association. The National Technology Roadmap for Semi Conductors. San Jose, CA, USA, 1994
3. Mead C A, Conway L A. Introduction to VLSI Systems. Addison-Wesley Publishing Company, 1980
4. Armstrong J R, Gray F G. Structured Logic Design with VHDL. PTR Prentice Hall, Inc, 1993
5. Armstrong J R. Chip Level Modelling with VHDL. Englewood Cliffs, NJ: Prentice Hall, 1989
6. Baumgartner K M. Computer Scheduling Algorithms Past, Present and future. Elsevier Science Publishing Co., Inc., 1991
7. Berge J-M, Fonkoua A, Maginot S et al. VHDL Designer's Reference. Netherlands: Kluwer Academic Publishers, 1992
8. Camposano R, Wolf W. High Level VLSI Synthesis. Netherlands: Kluwer Academic Publishers, 1991
9. Coelho D R. The VHDL Hand book. Netherlands: Kluwer Academic Publishers, 1989
10. Gajski D D et al. High-Level Synthesis: Introduction to Chip and System Design. Netherlands: Kluwer Academic Publishers, 1992
11. Harr R E, Srtanculescu A G. Applications of VHDL to Circuit Design. Netherlands: Kludwer Academic Publishers, 1991
12. Holl D D, Coelho D R. Multi-Level Simulation for VLSI Design. Netherlands: Kluwer Academic Publishers, 1986
13. IEEE Standard VHDL Language Reference Manual. New York: IEEE Press, 1987
14. Leung S S, Shanblatt M. ASIC System Design with VHDL: A Paradigm. Netherlands: Kluwer Academic Publishers, 1989
15. Lipsett R, Schaefer C, Ussery C. VHDL Hardware Description and Design. Netherlands: Kluwer Academic Publishers, 1989
16. Navabi Z. VHDL: Analysis and Modeling of Digital Systems. New York: McGraw-Hill, Inc., 1993
17. Perry D L. VHDL. New York: McGraw-Hill, Inc , 1991.
18. Roesner W. Hardware design Language for Logic Simulation and Synthesis in VLSI. Piscataway, NJ: IEEE Press, 1990
19. Schoen J. Performance and Fault Modeling with VHDL. Englewood Cliffs, NJ: Prentice Hall, 1989
20. Uyemnra J P. Circuit Design for CMOS VLSI. Netherlands: Kluwer Academic Publishers, 1992
21. Fichtner W, Morf M. VLSI CAD Tools and Applications. Netherlands: Kluwer Academic Publishers, 1987
22. Weste N, Eshrayhian K. Principles of CMOS VLSI Design—A Systems Perspective. Addison-Westey Publishing Company, 1992
23. J Mermet. VHDL for Simulation, Synthesis and formal Proofs of Hardware. Netherlands: Kluwer Academic Publishers, 1979