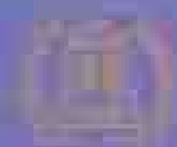
VHDL语言100例详解

北京理工大学ASIC研究所

Detailed Solution of VHDL by 100 Examples



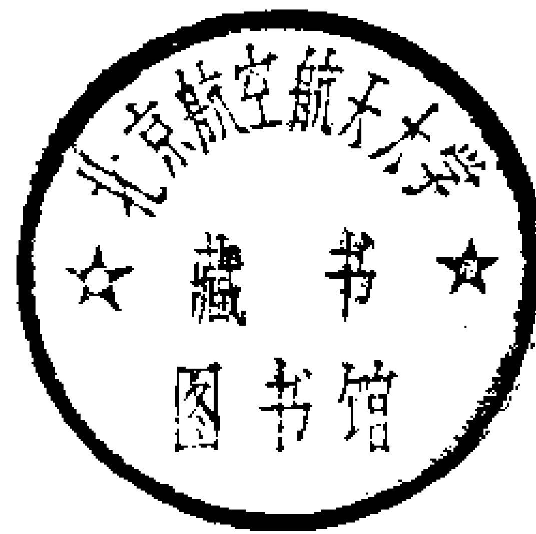
清华大学出版社
http://www.tup.tsinghua.edu.cn

TP312VH
01

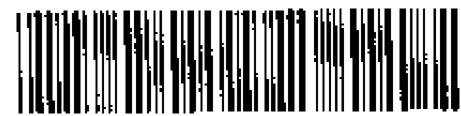
00015086

VHDL 语言 100 例详解

北京理工大学 ASIC 研究所



清 华 大 学 出 版 社


C0491346

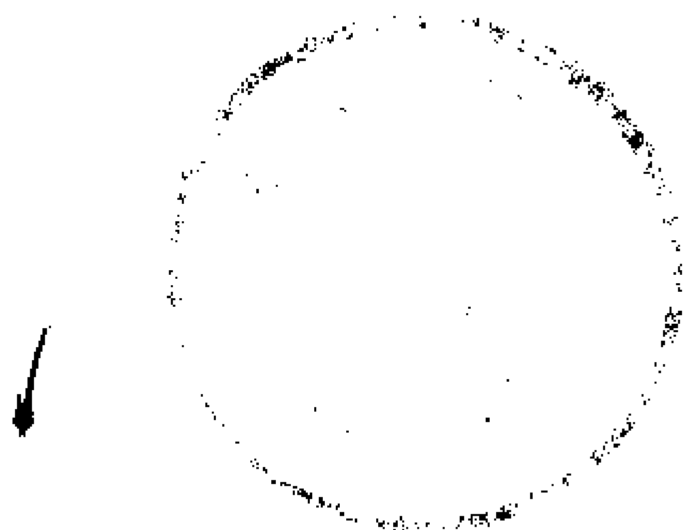
(京)新登字 158 号

内 容 简 介

本书通过 100 个实例，详细介绍硬件描述语言 VHDL 的各种语法现象及其在专用集成电路(ASIC)设计描述中的使用方法。书后附有光盘，其中包括北京理工大学 ASIC 研究所自行研制的有自主版权的 Talent 高层次自动设计系统的多媒体演示软件和 VHDL 模拟器(学习版)及 100 例的描述与模拟测试向量文件，读者可直接在微机上运行这些模拟题目，借以更深入地掌握 VHDL 语言及其使用方法。本书的突出特点是实用性强，理论联系实际，是 ASIC 设计者难得的一本 VHDL 语言设计工具书。

本书适合于从事数字系统/ASIC 自动设计的研究、开发人员参考，也适合于尚未掌握 VHDL 语言但已熟悉高级程序设计语言（如 C 语言或 ADA 语言）的读者学习 VHDL 语言，也可以作为高等学校计算机、自动控制、信息处理、电子工程和通信等专业的研究生及高年级本科生的教学参考书。

版权所有，翻印必究。
本书封面贴有清华大学出版社激光防伪标签，无标签者不得销售。



书　名：VHDL 语言 100 例详解
作　者：北京理工大学 ASIC 研究所
出版者：清华大学出版社（北京清华大学学研楼，邮编 100084）
　　　　http://www.tup.tsinghua.edu.cn
印刷者：清华大学印刷厂
发行者：新华书店总店北京发行所
开　本：787×960 1/16　印张：32　字数：698 千字
版　次：1999 年 12 月第 1 版　1999 年 12 月第 1 次印刷
书　号：ISBN 7-900625-02-X
印　数：00001~5000
定　价：52.00 元(含盘)

绪论——专用集成电路（ASIC）设计与 VHDL 语言

（代前言）

Ⅰ. ASIC 设计面临严峻的挑战

人类即将迈进 21 世纪，半导体器件制造技术高度微细化在迅猛发展。原计划 1998 年实现 0.25 微米技术的生产，已于 1997 年提前实现；而预计 2001 年能实现的 0.18 微米技术的生产，有迹象表明今年(1999 年)即可实现。这使得芯片的设计产生重大变化，设计制造集成度在 1000 万门以上的 ASIC 芯片成为可能。人们渴望已久的、在单块芯片上实现一个系统的全部功能，即系统级芯片（System Level IC，简称 SLIC；或 System-on-Chip，简称 SoC）的设计与制造正在或已经成为现实。

迈进 21 世纪，网络与通信技术、多媒体技术以及新型体系结构的计算机设计，随时都在向 SoC 的设计与制造提出新的严峻挑战。系统级芯片设计是电子信息整机和集成电路产品设计相结合的最完美体现。开展系统级芯片设计工作有利于摆脱电子信息产品设计与集成电路电路芯片设计相脱节的局面，不但能取得重大经济效益，而且能够加快我们民族电子产品，特别是整机产品的发展步伐。

迈进 21 世纪，制造技术与产品设计的需求都向集成电路设计业提出更严峻的挑战。要求设计行业能更快更好地设计出性能更优良、功能更完美、更复杂的 ASIC 产品。这迫使我们必须研究新的设计策略、设计方法和设计工具。在策略上最重要的是设计重用（reuse）。集成电路产品的集成度，目前仍然保持每 18 个月增长一倍的发展速度（摩尔定律），而产品的生命周期却日趋缩短，因此迫切要求提高 ASIC 芯片的设计速度。其中最重要的是尽可能重复运用已有的设计成果，采用具有知识产权的功能单元块（称 IP）。因此，必须重视 IP 的开发和重用。在设计方法方面是要研究在更高的层次上运用设计自动化（EDA）工具进行设计的方法。目前我国 ASIC 设计业的基本状况是芯片设计开发工作严重滞后于电子产品发展的需求，滞后于芯片生产线的吞吐能力。并且设计和投产的 ASIC 产品门类单一，品种太少，性能较低。要改变这种状况，急需提高设计能力。除了加强技术人才培养和设计队伍建设外，最有效的方法之一是要大力发展高层次 VHDL/Verilog 自动设计技术。

Ⅱ. 硬件描述语言 VHDL 的出现与发展状况

硬件描述语言（HDL，Hardware Description Language）至今约有 40 年的历史，现已成功地应用于 ASIC 自动设计的模拟验证和综合优化等方面。其最大特点是借鉴高级程

序设计语言的功能特性对电路的行为与结构进行高度抽象化、规范化的形式描述，并对设计进行不同层次、不同领域的模拟验证与综合优化等处理，使设计过程达到高度自动化。至 80 年代末，硬件描述语言的发展趋势进入多领域、多层次并且迫切要求标准化和集成化。最终，只有 VHDL 和 Verilog 适应了这种发展趋势，先后成为 IEEE 制定的硬件描述语言的工业标准。

VHDL 语言的全称是“超高速集成电路硬件描述语言”(VHSIC Hardware Description Language)。VHDL 的结构和方法受到 ADA 语言的影响，并吸收了其他硬件描述语言的某些优点。1986 年 3 月，IEEE 开始致力于 VHDL 的标准化工作，为此，成立了审查和完善 VHDL 的标准化小组。美国空军全力支持这项工作，并与 Intermetrics 签订发展 VHDL（IEEE-1076）的支撑软件合同。1987 年 12 月 IEEE 推出 IEEE Std1076-1987。

VHDL 语言成为 IEEE 的标准后，很快在世界各地得到广泛应用，逐渐成为数字系统/ASIC 设计中的主要硬件描述语言。1995 年中国国家技术监督局组织编撰并出版《CAD 通用技术规范》，推荐 VHDL 语言作为我国电子设计自动化硬件描述语言的国家标准。

为了增强 VHDL 语言的描述能力，方便设计应用，IEEE 在广泛征求各方面意见的基础上，对 IEEE Std 1076-1987 标准进行了修改和扩充。修订版于 1993 年 4 月成为美国国家标准局（ANSI）标准，并于同年 9 月被 IEEE 认可为标准，即 IEEE Std 1076-1993。新版本提供了共享变量（shared Variable）、组（group）、层次化路径名（hierarchical path Name）、异族模块（foreign Model）、签名（signature）等描述机制，增加了一些逻辑和移位操作，修订了 87 版中语法的不一致性。

此外，IEEE 为了促进 VHDL 的应用还成立了专门机构 VASG（VHDL Analysis and Standards Group），下设多个专题组进行有关 VHDL 子标准的建立工作，如 VHDL 综合包标准工作组（1076.3），ASIC 建模标准工作组（1076.4）、综合互操作性工作组（1076.6）等，并已经制订了一系列 VHDL 的子标准，如 VHDL 模型的标准多值逻辑系统 IEEE 1164（Std-Logic 包），VITAL（VHDL Initiative Towards ASIC Libraries）等。这些标准的建立使得不同 EDA 工具间可以通过 VHDL 进行各种设计信息的数据交换，无疑将推动 VHDL 的更广泛的应用。

III. VHDL 语言的特点

VHDL 是一种独立于实现技术的语言，它不受某一特定工艺的束缚，允许设计者在其使用范围内选择工艺和方法。为了适应未来的数字硬件技术，VHDL 还提供了将新技术引入现有设计的潜力。VHDL 语言的最大特点是描述能力极强，覆盖了逻辑设计的诸多领域和层次，并支持众多的硬件模型。具体而言，VHDL 较其他的硬件描述语言有如下优越之处：

1. 支持从系统级到门级电路的描述，同时也支持多层次的混合描述；描述形式可以

是结构描述，也可以是行为描述，或者二者兼而有之。

2. 既支持自底向上（bottom-up）的设计，也支持自顶向下（top-down）的设计；既支持模块化设计，也支持层次化设计；支持大规模设计的分解和设计重用。

3. 既支持同步电路，也支持异步电路；既支持同步方式，也支持异步方式。

4. 支持传输延迟，也支持惯性延迟，可以更准确地建立复杂的电路硬件模型。

5. 数据类型丰富，既支持预定义的数据类型，又支持自定义的数据类型；VHDL 是强类型语言，设计电路安全性好。

6. 支持过程与函数的概念，有助于设计者组织描述，对行为功能进一步分类。

7. 提供了将独立的工艺集中于一个设计包的方法，便于作为标准的设计文档保存，也便于设计资源的重用。

8. VHDL 语言的类属提供了向设计实体传送环境信息的能力。

9. VHDL 语言的断言语句可用来描述设计本身的约束信息，支持设计直接在描述中书写错误条件和特殊约束，不仅便于模拟调试，而且为综合化简提供了重要信息。

Ⅳ. VHDL 语言高级综合

由于 VHDL 是标准的硬件描述语言，因此国际上越来越多的高级综合系统都以 VHDL 作为设计输入。但是 VHDL 语言的本质是基于模拟而非综合的，其丰富的语法成份和描述机制无法且没有必要都进行综合。要实现 VHDL 综合系统，首先需确立 VHDL 的可综合子集。国际上对 VHDL 可综合子集的确立进行了许多研究，取得了一些有意义的结果，但并没有形成统一的标准。为了满足开发综合系统的需要，IEEE 正积极着手 VHDL 可综合子集的标准化工作，并推出了征求意见的草案，目前国内可查到的最新版本是 IEEE 1076.6/D2.0。

Ⅴ. VHDL 语言混级模合拟

1. 硬件结构特性的体现——元件、信号与进程

VHDL 具有许多与数字硬件结构直接相关的概念，其中最主要的是元件，它是数字硬件结构“未知方框”的抽象。VHDL 中，元件由实体与结构体两个概念共同描述，其中实体描述元件与外部环境的接口，其功能及结构是完全隐蔽的。实体的功能定义在称为结构体的单元中，而结构体规定设计实体输入/输出之间的关系。一个实体可存在多个对应的结构体，即可分别以行为、结构、数据流及各种方式混合的描述手段实现。元件的存在使 VHDL 脱离普通程序语言的范畴，成为描述数字电路的专用硬件设计语言。

VHDL 中的信号概念是数字电路中连线的抽象，它是各元件、各进程之间通信的数据通路。VHDL 中的信号的状态可影响与信号相关的进程的运行，体现数字系统各单元的输入及输出的关系。

VHDL 中的进程完成电路行为的描述，由一系列顺序语句组成，是 VHDL 设计中进行功能描述的基本单元。由于进程的执行是并发的，因此在 VHDL 中引入 delta 延迟概念，用于表示时间上无穷小的模拟步，是 VHDL 中模拟进程同步机制的关键。一个模拟时刻包括若干 delta 延迟，所有进程均可能在特定条件下，在同一时刻的任一 delta 延迟点上激活。设计者的设计意图有时希望忽略在 delta 延迟点上的变化，着重于计算一个模拟时刻结束时的稳定状态，因此 VHDL'93 引进延迟进程的概念。此类进程只在某一时刻的最后一个 delta 延迟时激活，这样可降低处理频度，尤其是当用于时序检查方面时。例如对于信号赋值语句

S1 <= A;

信号 S1 并不是立即得到所赋的信号 A 的值，而是必须经历 delta 延迟之后，S1 才更新为信号 A 的值。delta 延迟在模拟中由两阶段模拟算法实现。而对于包含以上信号赋值语句的进程，在一个模拟周期内可能频繁激活。有时设计者希望忽略这些延迟激活，因而引进延迟进程概念免除不必要的 delta 延迟处理。

因此，包含上述信号赋值语句的延迟进程将仅在满足激活条件时刻的最后一个 delta 延迟（即一时刻的稳定阶段）激活，激活频度将会大大降低。

延迟进程与非延迟进程的区别在于进程挂起等待之后的唤醒执行的时间不同。进程的激活要素包括三方面：一是敏感信号集，其次是激活条件，再次是等待时间。这些条件相互制约，当激活要素满足时，进程在指定时刻立即激活，所谓指定时刻可细化到某一 delta 延迟时刻。若是延迟进程，则激活推迟到当前模拟时刻的最后一个 delta 延迟时刻（即某一周期的稳定状态），且如果在最后一个 delta 时刻，有多个激活的延迟进程，则这些进程是执行顺序相关的。

2. 传输延迟、惯性延迟与阈值

VHDL'87 标准为信号传输的延迟提供两种延迟模式：传输延迟和惯性延迟。其中传输延迟相应于输入波形没有变化的传输，即任何宽度的脉冲均被传送，无滤除处理，类似电流通过电线上的延迟。而惯性延迟模式，宽度小于惯性延迟的脉冲均被滤除。这种延迟模式体现了开关电路的特性，如果脉冲的宽度小于开关电路的转换时间，或小于指定的脉宽，则不能传播。为了便于明确地定义最小脉宽限制，VHDL'93 引入阈值（reject）的概念。

3. 硬件的并发性模拟

VHDL 的并发性体现在两个方面，首先使用 VHDL 进行数字电路设计时存在并发性，即 VHDL 支持设计分解，可使被分解的各子部分的设计并行完成。其次，模型的设计主要由三部分组成：定义实体——确立模型与环境的接口；定义结构体——完成模型的功能描述；定义测试部分——为模型生成测试向量，并捕获模型输出信号状态以供分析。下

面，通过模型的实际设计过程加以说明。首先，在系统分析阶段，系统分析者可将设计对象分为若干独立的子元件，交给若干设计小组实现。系统分析者严格定义元件接口，并将元件之间的相互作用以文档形式提供给各设计小组。然后，各设计小组可独立并行地对子元件进行详细设计，并对子元件进行模拟验证，确保正确性。最终，系统设计者集成各子元件形成完整的设计，对整个设计进行模拟验证。设计的并发性可大大加快整体设计进程和提高设计质量。

其次，VHDL 之所以称为硬件描述语言，很重要的一点是因为它在模拟执行上具有并发性，这点很适于描述电路活动的并发性特点，是其他程序设计语言所不具备的。VHDL 中的进程类似于 UNIX 操作系统中的进程，它们的挂起、活动均是独立的。并发性使得 VHDL 的设计模拟可在并行机上进行，这样大大提高了模拟效率，是解决模拟时间瓶颈的方法之一。

4. 混合级描述及混合级模拟

VHDL 的描述范围覆盖系统级、算法级、寄存器传输级、逻辑电路级，具有连续性和完整性。VHDL 的结构描述方式和行为描述方式有机结合，各描述层次之间彼此衔接，协调一致。目前，较常用的大规模集成电路的设计方法包括基于标准单元库的自底向上的设计方法和自顶向下便于早期优化的 Top-Down 设计方法，以及自底向上和自顶向下相结合的设计方法。由于设计规模日益增大，设计复杂度急剧增加，传统的设计起点偏重低层的方法，会因设计规模的庞大增加很大的工作量。因此提高设计层次，注重早期优化，是现行较好的设计方式。目前，设计对象整体的设计过程经历多个层次。首先，在较高的抽象层次，进行前期的概念设计，优化设计模型；然后经由高级综合工具综合，产生寄存器传输级网表；最后经低层次综合工具，形成最终的设计结果。因此，由于存在多层次设计，就需要多个层次上的模拟，VHDL 模拟器可完成混合级模拟，可为各个层次的硬件设计提供有效模拟，反映设计意图，供设计者调试其设计。是适应当前电路设计的最佳选择之一。

Ⅵ. VHDL 语言高级综合系统 Talent

以硬件描述语言（VHDL/Verilog）高级综合为核心的高层次设计（HLD）方法正日益成为 EDA 的主流。但由于 HLD 跨越设计的多个层次与领域，完成整个流程涉及多种关键技术，如硬件描述语言可视化输入与编译、模拟与验证、综合（行为级、RTL、逻辑级）与工艺映射等，因此国际上只有少数几家 EDA 公司掌握了 HLD 的核心方法，所推出的相应的 EDA 工具也都价格昂贵。为了打破国外的技术垄断，推出具有独立知识产权的高层次 EDA 工具，为国内的集成电路设计业服务，北京理工大学 ASIC 研究所于“八五”期间进行了 VHDL 语言高级综合的研究，完成了原型的 VHDL 高级综合及混合级模拟系统 HLS/BIT，并在“九五”期间开展了相应的实用化工作，研制面向实用的专用集成电路高

层次自动设计系统（命名为 Talent）。

Talent 的系统目标是利用硬件描述语言 VHDL 进行数字系统设计的高层次行为功能描述，并通过综合将设计描述自动转换为低层次的设计实现，从而实现设计过程的高度自动化。其特点是基于硬件描述语言，以高级综合为核心，从高层次进行电路的自顶向下设计。其主要功能包括 VHDL 的编辑、编译、模拟验证，设计的自动综合与工艺映射，逻辑图自动生成等。其系统结构如图 0.1 所示，从图中可以看出，Talent 可分为设计输入、设计综合及设计验证三大部分。其设计过程如下。

1．利用 VHDL 对设计进行功能和算法描述

通过 Talent 系统中 VHDL 智能编辑器可以方便地进行设计描述的录入和编辑，它针对 VHDL 语言的特点特别提供了标识符自动记忆、单词联想及 VHDL 固定语法结构联想式输入等功能。须注意进行设计描述时应根据 Talent 系统所确立的综合子集，使用综合所能接受的语法现象和描述方式。

2．对 VHDL 设计描述进行编译

Talent 系统的 VHDL 编译器支持 VHDL87/93 全集，以语法分析器为核心，采取语法制导、分别编译（按次序编译）、一次扫描等技术，使系统具有很好的实用性。

3．通过综合自动生成与工艺无关的 RTL 设计实现

综合又可分为数据流综合与控制流综合两部分，前者自动生成电路的数据通道部分并提取相应的控制信息，后者将所提取的控制信息通过时序逻辑综合及组合逻辑综合完成控制器的综合。其中数据流综合子系统完成高级综合的任务，是整个系统的核心。

4．通过工艺映射与工艺无关的综合结果转换为与工艺相关的设计结构

综合结果的 RTL 网表与工艺无关，当 ASIC 投片制造时可根据特定的目标工艺，通过工艺映射将综合结果转换成工艺厂商所接收的设计格式。Talent 工艺映射子系统中提出了两级 RTL 映射策略，即 RTL 工艺无关的映射与工艺相关的映射，并采取了知识制导的工艺映射方法。目前通过工艺映射，Talent 系统已实现与几种现场可编程器件开发系统的联结，并生成实际的器件，借以完成器件实现电路的仿真。

5．在设计的各个阶段利用 VHDL 模拟进行设计的模拟验证

Talent 系统的 VHDL 混合级模拟器（Vsim/Talent）全面支持 VHDL87 和 VHDL93，并提供了强大的调试功能。其模拟核心采用事件驱动算法，对于同步电路设计采用基于周期的算法。模拟核心采用层次模拟，保留设计原型的元件之间的互连及嵌套关系，便于加载完善、灵活的调试系统，进行调试定位，信息查找和运行控制，符合设计者的思维习惯。

6. 利用逻辑图自动生成工具直观地观察设计结果

逻辑图自动生成工具将综合及工艺映射的结果分页自动生成逻辑图，并作为设计文档保存。其成图迅速，布局美观，走线均匀合理，合乎人的阅读习惯，并具有友好的用户界面及缩放、滚动等完善的编辑功能。

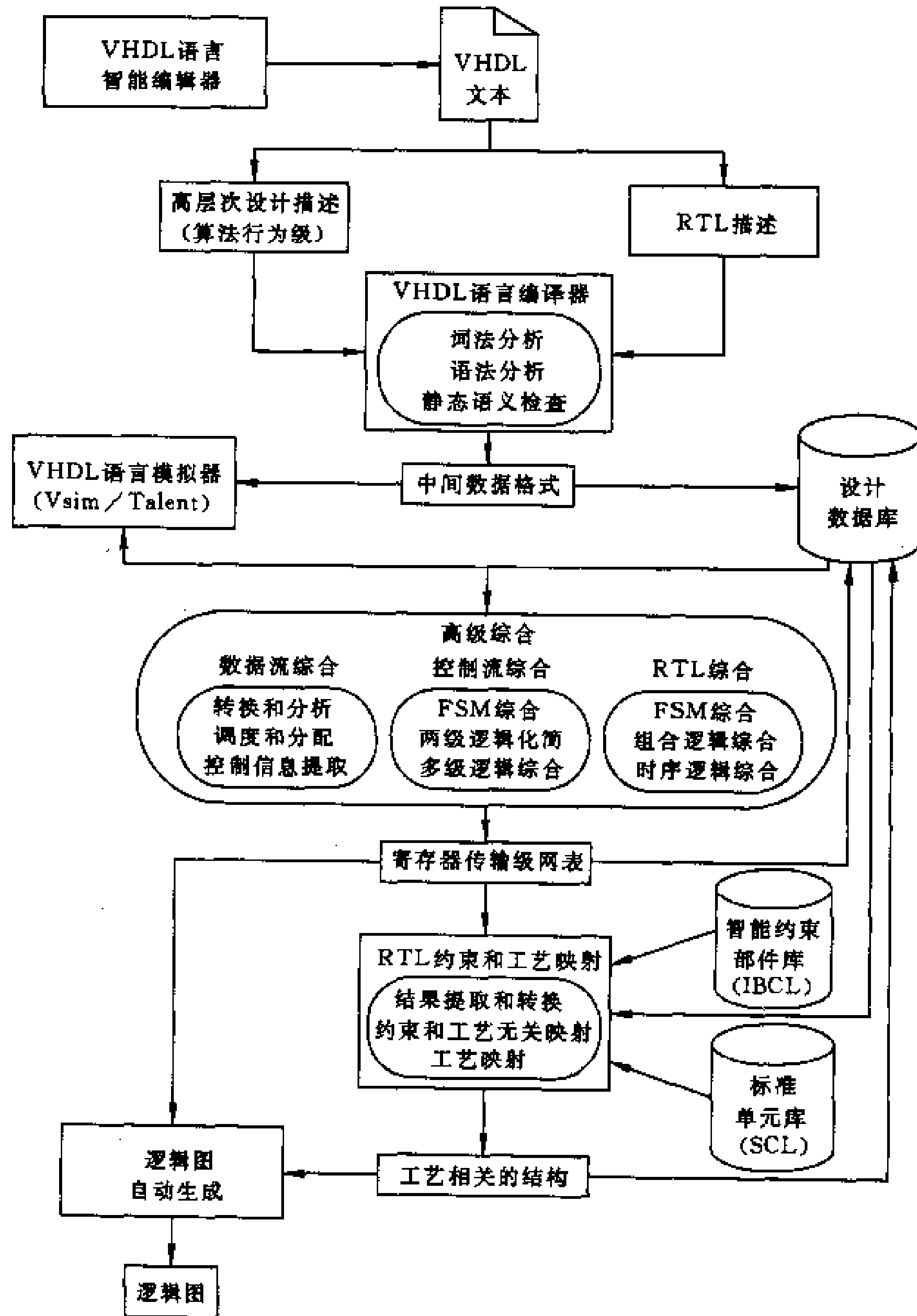


图 0.1 Talent 系统结构图

Ⅶ. VHDL 语言 100 例的选取与本书的撰写目标

与其他以 VHDL 为输入的 EDA 软件一样，Talent 自动设计系统与设计者的通信界面（工具）是 VHDL 语言。应用该系统完成 ASIC 设计的至关重要的前题是，设计者必须熟练掌握 VHDL 语言，必须能够运用 VHDL 语言对所要设计的 ASIC 完成其行为功能或结构的描述。VHDL 语言规模很大，语法现象十分复杂。经验表明，一个初学者要想达到能得心应手地描述出一个有一定规模的 ASIC 芯片的程度，总需要一个学习和实践的过程。特别是对某些规模较大有实用意义的设计题目的描述，上机模拟和结果分析是一个相当枯燥艰难的历程。本书的撰写恰是为缩短这一进程，为读者提供一个 VHDL 语言学习和借鉴的捷径。

本书选取的 100 个例题全部在 Talent 系统上通过编译和模拟。其中许多题目曾用于调试和测试 Talent 系统。这些题目大致可分为 4 类。

首先选择组成数字系统/ASIC 的某些基本单元，如加（减）法器、乘（除）法器、比较器、选择器、寄存器等（第 1~8 例）。通过对这些单元电路的行为功能描述，阐明 VHDL 语言设计实体说明、结构体描述、并发进程语句与激活条件、顺序语句、变量与信号、位向量、延迟以及重载等 VHDL 语言的基本概念和描述方法。即使没学过硬件描述语言的人，只要有高级程序设计语言（如 C 语言或 ADA 语言）的基本知识，也可借助这些题目的详解掌握 VHDL 语言。

其次选择的一批例题是为了拓宽讨论 VHDL 语言的一些更为重要而复杂的语法现象。特别是对一些较难理解的语法问题和使用技巧，则通过反复举例进行充分解释。例如数据类型、函数及七值逻辑问题（第 9~18 例），死锁与振荡（第 20~23 例），分辨函数、分辨信号与属性（第 24~29 例），进程（第 30~33 例），类属（第 40，41，46 例）以及延迟分析（第 48~51 例）等都有重点地列举若干例题进行深入浅出的讨论。

随后开始触及某些单元电路的应用，例如各种功能的寄存／计数器（第 52~56 例），译码器（第 57~59 例），基本计算电路（第 60~63 例）及有限状态自动机（第 64 和 65 例）等等。通过这些举例读者可进一步深入而全面地理解和掌握前述语法现象的使用方法。

最后给出某些有一定实用意义的举例。如 DSP（第 66 和 67 例），整机性的设计举例（第 68~77、90~93 例），四位微处理器芯片 Am2901 和 Am2910（第 78~89 例）以及流水线结构的 RISC 机（第 94~100 例）。这些举例具有典型性和实用性。它们的 VHDL 描述较长，语法结构也相对复杂。读者需熟悉它们的组成原理和体系结构后，再来分析相应的 VHDL 源描述。弄清楚这些描述，非常有利于您用 VHDL 进行 ASIC 设计工作。

附录Ⅰ中的 100 例内容一览表，摘要给出每一例涵盖的基本语法内容，可供读者迅速查寻所需要参考的题目。为了使读者准确掌握与理解 VHDL 语言的用语，附录Ⅱ特地给出 VHDL 语言专用术语的中英文对照表。

书后所附光盘给出 Talent 系统的多媒体演示系统、100 例描述和模拟测试向量的全部有关文件，以及 Talent 系统 VHDL 模拟器（学习版本），读者可在熟读附录Ⅲ的 Vsim/Talent 使用方法后，到微机上直接运行这些模拟文件，借以更深入地掌握有关的内容。

刘明业教授主持本书稿的撰写，并最后统稿全书。参加撰写工作的有石峰副教授/博士、韩曙副教授、张东晓博士、袁媛、陈东瑛、刘沁楠、吴清平、刁岚松、王作建、李春、李杰、谢巍、张俭锋等 13 位同志。叶梅龙教授精心审阅了本书各例的初稿，并进行了修改，使全书撰写格式和措辞用语等趋于统一和规范，同时归纳出书后附录Ⅰ的一览表。袁媛为本书稿的校对、改错和部分录入工作付出了大量艰辛的劳动。

本书的素材是作者们多年从事科研开发工作的积累。他们的工作长期以来得到国家"八五""九五"科技攻关项目、国防微电子技术预研项目、国家自然科学基金项目以及国家教委博士点建设基金项目的支持。对上述有关部委、国防科工委、电科院、北京华大集成电路设计中心、兵科院及北京理工大学的各级领导、师长和朋友的长期关怀、指导和帮助，作者在此谨致以诚挚的谢意。

限于我们的工作实践和认识水平，书中难免存在缺点、疏忽甚至错误。恳切希望广大读者批评指正。

刘明业

1999 年 9 月 9 日

于北京理工大学 ASIC 研究所

目　录

绪论——专用集成电路（ASIC）设计与 VHDL 语言（代前言）……刘明业（Ⅴ）

Ⅰ　ASIC 设计面临严峻的挑战……（Ⅴ）

Ⅱ　硬件描述语言 VHDL 的出现与发展状况……（Ⅴ）

Ⅲ　VHDL 语言的特点……（Ⅵ）

Ⅳ　VHDL 语言高级综合……（Ⅶ）

Ⅴ　VHDL 语言混合级模拟……（Ⅶ）

Ⅵ　VHDL 语言高级综合系统 Talent……（Ⅸ）

Ⅶ　VHDL 语言 100 例的选取与本书的撰写目标……（Ⅻ）

第 1 例　带控制端口的加法器……袁　媛（1）

第 2 例　无控制端口的加法器……袁　媛（4）

第 3 例　乘法器……袁　媛（6）

第 4 例　比较器……袁　媛（8）

第 5 例　二路选择器……袁　媛（11）

第 6 例　寄存器……袁　媛（13）

第 7 例　移位寄存器……袁　媛（16）

第 8 例　综合单元库……袁　媛（22）

第 9 例　七值逻辑与基本数据类型……袁　媛（29）

第 10 例　函数……袁　媛（32）

第 11 例　七值逻辑线或分辨函数……袁　媛（35）

第 12 例　转换函数……袁　媛（38）

第 13 例　左移函数……袁　媛（40）

第 14 例　七值逻辑程序包……袁　媛（42）

第 15 例　四输入多路器……陈东瑛（51）

第 16 例　目标选择器……吴清平（57）

第 17 例　奇偶校验器……陈东瑛（61）

第 18 例　映射单元库及其使用举例……陈东瑛（69）

第 19 例　循环边界常数化测试……陈东瑛（75）

第 20 例　保护保留字……袁　媛（77）

第 21 例　进程死锁……刘沁楠（79）

第 22 例　振荡与死锁……袁　媛（81）
第 23 例　振荡电路……刁岚松（83）
第 24 例　分辨信号与分辨函数……袁　媛（87）
第 25 例　信号驱动源……刘沁楠（92）
第 26 例　属性 TRANSACTION 和分辨信号……陈东瑛（96）
第 27 例　块保护及属性 EVENT，STABLE……陈东瑛（101）
第 28 例　形式参数属性的测试……刘沁楠（104）
第 29 例　进程和并发语句……刁岚松（107）
第 30 例　信号发送与接收……刁岚松（111）
第 31 例　中断处理优先机制建模……吴清平（113）
第 32 例　过程限定……刘沁楠（116）
第 33 例　整数比较器及其测试……刘沁楠（119）
第 34 例　数据总线的读写……刁岚松（129）
第 35 例　基于总线的数据通道……李　春（134）
第 36 例　基于多路器的数据通道……李　杰（148）
第 37 例　四值逻辑函数……袁　媛（152）
第 38 例　四值逻辑向量按位或运算……刁岚松（156）
第 39 例　生成语句描述规则结构……袁　媛（159）
第 40 例　带类属的译码器描述……袁　媛（164）
第 41 例　带类属的测试平台……袁　媛（169）
第 42 例　行为与结构的混合描述……袁　媛（171）
第 43 例　四位移位寄存器……刘沁楠（174）
第 44 例　寄存/计数器……袁　媛（185）
第 45 例　顺序过程调用……陈东瑛（189）
第 46 例　VHDL 中 generic 缺省值的使用……王作建（191）
第 47 例　无输入元件的模拟……王作建（196）
第 48 例　测试激励向量的编写……袁　媛（201）
第 49 例　delta 延迟例释……吴清平（206）
第 50 例　惯性延迟分析……吴清平（210）
第 51 例　传输延迟驱动优先……陈东瑛（213）
第 52 例　多倍（次）分频器……刁岚松（216）
第 53 例　三位计数器与测试平台……刘沁楠（220）
第 54 例　分秒计数显示器的行为描述……陈东瑛（226）
第 55 例　地址计数器……陈东瑛（234）
第 56 例　指令预读计数器……吴清平（242）

第 57 例 加、减、乘指令的译码和操作 …… 吴清平（245）
第 58 例 2-4 译码器结构描述 …… 刘沁楠（248）
第 59 例 2-4 译码器行为描述 …… 吴清平（255）
第 60 例 转换函数在元件例示中的应用 …… 王作建（258）
第 61 例 基于同一基类型的两分辨类型的赋值相容问题 …… 王作建（261）
第 62 例 最大公约数的计算 …… 刁岚松（266）
第 63 例 最大公约数七段显示器编码 …… 吴清平（269）
第 64 例 交通灯控制器 …… 吴清平（272）
第 65 例 空调系统有限状态自动机 …… 刁岚松（276）
第 66 例 FIR 滤波器 …… 谢 巍（280）
第 67 例 五阶椭圆滤波器 …… 刘沁楠（290）
第 68 例 闹钟系统的控制器 …… 张东晓（302）
第 69 例 闹钟系统的译码器 …… 陈东瑛（311）
第 70 例 闹钟系统的移位寄存器 …… 陈东瑛（315）
第 71 例 闹钟系统的闹钟寄存器和时间计数器 …… 陈东瑛（317）
第 72 例 闹钟系统的显示驱动器 …… 陈东瑛（322）
第 73 例 闹钟系统的分频器 …… 陈东瑛（325）
第 74 例 闹钟系统的整体组装 …… 张东晓（327）
第 75 例 存储器 …… 李 春（333）
第 76 例 电机转速控制器 …… 张俭锋（337）
第 77 例 神经元计算机 …… 袁 媛（343）
第 78 例 Am2901 四位微处理器的 ALU 输入 …… 韩 曙（347）
第 79 例 Am2901 四位微处理器的 ALU …… 韩 曙（353）
第 80 例 Am2901 四位微处理器的 RAM …… 韩 曙（359）
第 81 例 Am2901 四位微处理器的寄存器 …… 韩 曙（363）
第 82 例 Am2901 四位微处理器的输出与移位 …… 韩 曙（365）
第 83 例 Am2910 四位微程序控制器中的多路选择器 …… 韩 曙（370）
第 84 例 Am2910 四位微程序控制器中的计数器/寄存器 …… 韩 曙（374）
第 85 例 Am2910 四位微程序控制器的指令计数器 …… 韩 曙（379）
第 86 例 Am2910 四位微程序控制器的堆栈 …… 韩 曙（382）
第 87 例 Am2910 四位微程序控制器的指令译码器 …… 韩 曙（390）
第 88 例 可控制计数器 …… 韩 曙（399）
第 89 例 四位超前进位加法器 …… 韩 曙（406）
第 90 例 实现窗口搜索算法的并行系统（1）——协同处理器 …… 李 杰（410）
第 91 例 实现窗口搜索算法的并行系统（2）——序列存储器 …… 李 杰（416）

第 92 例　实现窗口搜索算法的并行系统（3）——字符串存储器....李　春（419）
第 93 例　实现窗口搜索算法的并行系统（4）——顶层控制器........李　春（422）
第 94 例　MB86901 流水线行为描述组成框架........石　峰（428）
第 95 例　MB86901 寄存器文件管理的描述........石　峰（434）
第 96 例　MB86901 内 ALU 的行为描述........石　峰（437）
第 97 例　移位指令的行为描述........石　峰（440）
第 98 例　单周期指令的描述........石　峰（442）
第 99 例　多周期指令的描述........石　峰（445）
第 100 例　MB86901 流水线行为模型........石　峰（458）
参考文献........（467）

附录Ⅰ　100 例内容摘要一览表........叶梅龙　袁　媛（468）
附录Ⅱ　VHDL 专用术语中英文对照........刘沁楠（475）
附录Ⅲ　Talent 系统 VHDL 模拟器使用说明........张俭锋（486）

附光盘：内容包括 100 例有关的 VHDL 描述文件及 Talent 系统 VHDL 模拟器
（张东晓、吴清平、袁媛监制）

第1例　带控制端口的加法器

袁　媛

1. 电路系统工作原理

本例针对一个典型的加法器进行 VHDL 语言的描述，比较特殊的是该加法器带有一个控制端口。它用于完成两个位向量的相加，其电路系统示意图如图 1.1 所示。

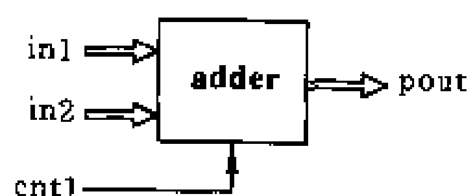


图 1.1　带控制端口的加法器

该加法器有两个输入端口 in1 和 in2，一个输出端口 pout，还有一个控制端口 cntl，其中输入端口和输出端口的类型为位向量；控制端口的类型则是位，或者为“0”，或者为“1”。当控制端口 cntl 为“1”时，对输入端口的两个位向量进行加法操作，否则不进行任何操作。

2. 电路的 VHDL 语言描述方法及语法分析

本例的描述与语法是一些基本的结构，十分简单，现详述如下。

(1) 带控制端口加法器的源描述

```
entity    adder     is
     port(     in1   :    bit_vector;
               in2   :    bit_vector;
               cntl:      bit;
               pout :     out bit_vector);
end  adder;
architecture   func      of   adder      is
begin
     process(cntl)
          begin
               if   (cntl='1')   then      pout <= in1+in2;
               end  if;
     end  process;
end  func;
```

注意：只要将语句 pout <= in1 + in2 中的 “+” 号改成 “-” 号，此源描述就成为带控制端口的减法器的描述。

(2) 设计实体

设计实体是 VHDL 描述中的基本单元和最重要的抽象，它可以代表整个系统、一块电路板、一个芯片、一个单元或一个门电路。在本例中，设计实体是一个加法器。

(3) 实体说明

实体说明是在 VHDL 描述中，以保留字 **entity** 开始的部分，通常形式如下：

```
entity    the_name_of_entity      is
     port (
                ...
          );
end       the_name_of_entity;
```

实体说明所描述的是单元或电路等器件的外貌，不管器件内部如何复杂，只把该器件当作一个黑匣子来看待。

实体说明中包括器件的端口，本例中需要说明的是两个输入端口，一个控制端口和一个输出端口，即图 1.1 中的外部端口。

需要说明的是，本例中的实体说明很简单，只包括端口 **port**，在较复杂的电路中，实体说明中还有其他许多的内容，比如说还可以通过类属 **generic** 引入参数。

(4) 结构体

结构体也是 VHDL 描述必不可少的部分，其中描述器件的行为或结构，说明该器件的功能以及如何完成这些功能。对器件的行为进行描述，需要使用行为描述，而对器件的结构进行描述，就需要使用结构描述，有时两种描述也可混合使用。

结构体中有一条或多条并发语句，一条大的并发语句（如进程语句）可能由多条顺序语句组成。

(5) 进程

进程通过进程语句来描述，多条进程语句之间是并发关系，而进程语句本身则定义一组在整个模拟期间连续执行的顺序语句，进程语句用来描述部分硬件的行为。

进程用保留字 **process** 标识，通常在 **process** 之后带有一个敏感信号表，表中可以有一个或多个敏感信号。所谓敏感信号是指当其值发生改变时，会引起进程中语句执行的那些信号，也就是重新激活进程。

本例中的结构体内只有一条进程语句，用来描述加法器的行为，且敏感信号是控制

信号 cntl。当 cntl 发生变化时（由 “0” 变到 “1”，或由 “1” 变到 “0”），都会激发进程执行。

(6) 顺序语句 if

本例的进程中包含一条 **if** 语句，它与高级语言中的 **if** 语句类似，比较好理解。该语句意即，如果 cntl 为 “1”，则做加法操作；否则不进行任何操作。

(7) 位向量

注意到本例中所做的加法并不是简单的两个整数相加或两个二进制位的相加，它是将两个位向量的值相加，输出结果也是位向量。

所谓位向量是一种数据类型，用 bit_vector 表示，指多个二进制位连在一起，如 “10010110”，“01010101” 等，即将多个位的操作集成在一起完成。

(8) 重载

既然本例所描述的不是单个二进制位的相加，那么语句中的 “+” 操作符就不是通常意义上的加法操作，它实际上是一个重载运算符。重载运算符允许运算符执行相同的运算，但是操作数的数据类型可以不同。重载使设计者避免了为实际上做同一操作的子程序必须去生成无数个相同的名字。这样，无论是整数相加、位相加还是位向量的相加，都可以只用 “+” 这个运算符。

（源描述文件名：1_adder.vhd）

第2例 无控制端口的加法器

袁 媛

1. 电路系统工作原理

本例与第1例中的加法器都是用于完成两个位向量的相加，所不同的是此例中的加法器不带任何控制端口，其电路系统示意图如图2.1所示。该加法器有两个输入端口 in1 和 in2，一个输出端口 pout，它们的类型皆为位向量。

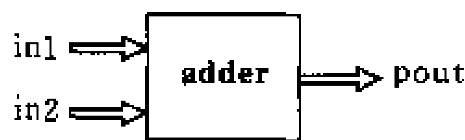


图 2.1 不带控制端口的加法器

其功能与第1例中的加法器的功能有所区别，这个加法器每隔 2ns 就将输入端口两个位向量相加的结果送到输出端口，它所进行的加法操作由时间控制，而不是由控制端口的信号来控制。

2. VHDL 语言描述方法及语法分析

本例除了进程中的语句与第1例有所不同之外，其余基本相同。当然，实体说明中的端口说明项不包括控制端口。下面重点讲述进程中的内容。

(1) 不带控制端口的加法器的源描述

```
entity    adder     is
     port(  in1      :  bit_vector;
            in2      :  bit_vector;
            pout     :  out bit_vector);
end       adder;
architecture  func      of   adder      is
begin
     process(in1, in2)
          begin
               pout <= in1 + in2  after  2ns;
          end  process;
end  func;
```

注意：只要将语句 pout <= in1 + in2 **after** 2ns 中的“+”号改成“-”号，此

源描述就成为不带控制端口的减法器的描述。

(2) 敏感信号

本例中，敏感信号不再是控制端口信号，而是两个输入端口的信号 in1 和 in2，这意味着当输入信号中有任何一个或者两个都发生变化时，进程就要再重新执行一遍。从电路功能的角度来理解：只要在输入端有新的数据输入，那么加法器就必然又要开始工作，将新的数值相加并送到输出端口，这样周而复始地做加法操作，由敏感信号决定什么时候开始新一轮的工作。

(3) 延迟

本例中特别值得注意的是带有延迟的语句

```
pout <= in1 + in2  after 2ns;
```

延迟是硬件描述的典型特征，一般硬件单元的值从发生到生效有一段时间间隔，这个时间间隔是相对于当前时刻的相对量，称为延迟。延迟包括惯性延迟和传输延迟。

惯性延迟的保留字为 **inertial**，缺省为惯性延迟，在大多数情况下，它相当近似地反映了实际器件的行为。例如，有一个电容器，当给这个电容器的两端加上电压时，由于电容器本身的物理特性，它会有一个充电的过程，当充电过程完成以后，电容器本身就会产生一个值的输出，而充电需要时间，这就产生了延迟。

需要注意的是，如果在充电过程中发生了意外，充电没有完成，那么，显然输出的值就不会是正确的，不满足惯性延迟所要求的时间，因此就会保持原来输出的值，这样才能反映实际器件的行为。

传输延迟的保留字为 **transport**，它表示在连线上的延迟。例如，在两个电路单元之间有连线相接，如一个加法器和一个乘法器相连，加法器的输出为乘法器的输入，通过连线，加法器的输出能够到达乘法器作为输入，信号的传输也是需要时间的，这就是传输延迟。与惯性延迟不同的是，无论发生什么情况，连线一端的值都能够传输到连线的另一端。

本例中的延迟为惯性延迟。将输入端口的两个位向量相加后所得到的值要在 2 纳秒之后才会到达输出端口。如果没有新的值输入，则输出端口保持原值。

（源描述文件名：2_adder.vhd）

第3例 乘 法 器

袁 媛

1. 电路系统工作原理

本例是一个带控制端口的乘法器，该乘法器用于完成两个位向量的相乘，其电路系统示意图如图3.1所示。该乘法器有两个输入端口分别为in1和in2，一个输出端口pout，还有一个控制端口cnt1，其中输入端口和输出端口的类型为位向量；控制端口的类型则是位，或者为“0”，或者为“1”。当控制端口cnt1为“1”时，对输入端口的两个位向量进行乘法操作，否则不进行任何操作。

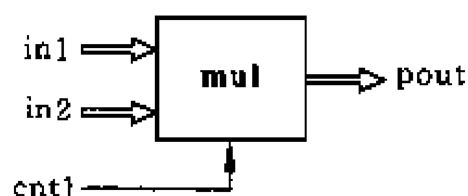


图3.1 带控制端口的乘法器

2. VHDL语言描述方法及语法分析

首先给出乘法器的源描述如下：

```
entity    mul  is
     port( in1       :  bit_vector;
            in2      :  bit_vector;
            cnt1     :  bit;
            pout     :  out bit_vector);
end       mul;

architecture  func     of   mul   is
begin
     process(cnt1)
          begin
               if   (cnt1= ' 1' )  then      pout <= in1 * in2;
          end  if;
     end  process;
end  func;
```

除了语句：

```
if   (cnt1=' 1' )   then      pout <= in1 * in2;
```

中的 “+” 号改为 “*” 号以外，本例与第 1 例几乎完全相同，所以有关这一源描述的描述方法及语法分析请参考第 1 例。

还需要注意的是，只要将语句 pout <= in1 * in2 中的 “*” 号改成 “/” 号，此源描述就成为带控制端口的除法器的描述。

（源描述文件名：3_mul.vhd）

第4例 比 较 器

袁 媛

1. 电路系统工作原理

本例是一个比较器，用于比较两个位串所代表的整数的大小。其电路系统示意图如图 4.1 所示。此比较器两个输入端口分别为 in1 和 in2，类型为位向量，比较器所比较的就是这两个位串所代表的整数的大小。另外还有一个输出端口 pout，将比较的结果输出。如果 in1 小于 in2，则 pout 输出为“1”，否则输出为“0”。

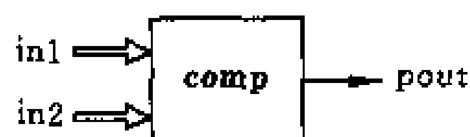


图 4.1 比较器

2. VHDL 语言描述方法及语法分析

本例的重点语法现象是变量和函数，下面一一说明。

(1) 比较器的源描述

```
entity    bit_rtl_lt_nc     is
    port (  in1, in2    :   bit_vector;
            pout        :   out bit );
end bit_rtl_lt_nc;

architecture     func     of   bit_rtl_lt_nc      is
begin
    process (in1 , in2)
        variable  left:    integer;
        variable  right    :    integer;
        begin
            left      :=   bit_to_int(in1);
            right     :=   bit_to_int(in2);
            if   (left < right) then pout <= ' 1'   after 1ns;
            else pout <= ' 0'   after 1ns;
            end if;
    end process;
    end func;
```

(2) 变量与信号

VHDL 语言中有三类对象，分别是信号、变量和常量。信号表示把元件的输入输出端口连接在一起的互连线；变量用于对中间数据的临时存储；而常量则是固定的值。

在前面的举例中接触到的都是端口信号赋值，如 pout <= in1+in2 就是向端口信号 pout 赋值。而在本例中，涉及到变量及对变量赋值，并将对变量和信号、变量赋值和信号赋值进行比较。

① 信号

信号是实体间动态交换数据的手段，用信号对象把实体连接在一起形成模块。用关键字 **signal** 说明信号。实体说明、结构体说明和包说明都能说明信号。

② 变量

变量用于进程语句和子程序中中间的数据存储。用关键字 **variable** 说明变量。

③ 变量与信号的区别

- 信号赋值有延迟；而变量赋值没有延迟。
- 信号除当前值外有许多相关的信息，如历史信息和波形值；而变量只有当前值。
- 进程对信号敏感而对变量不敏感。
- 信号可以是多个进程的全局信号；而变量只在定义它的顺序域中才可见。
- 信号是硬件中连线的抽象描述，功能是保存变化的数据值和连接子元件，信号在元件的端口连接元件；变量在硬件中没有类似的对应关系，而是用于硬件特性的高层次建模所需要的计算中。

④ 变量赋值与信号赋值

在描述中，如果给一个信号赋值，则该信号的值不会立即生效，而是要等待一个 delta 延迟之后，该值才会真正赋给信号，否则该信号的值在 delta 延迟之前仍是原来的值；而对于变量赋值，只要赋了值，该值就立即生效。下面举例说明变量赋值和信号赋值之间的区别。

变量赋值：

```
architecture    example  of  example  is
variable     a, b, c : integer;
begin
    a := 40;
    b := 30;
    a := b;
    b := a;
end example;
```

显然，这一代码段的结果就是最后变量 a 和 b 的值都为 30。

信号赋值：

```
architecture    example  of  example  is
signal   a, b, c : integer;
begin
     a <= 40;
     b <= 30;
     wait for 10ns;
     a <= b;
     b <= a;
     wait for 10ns;
end example;
```

这一代码段所得出的结果就不一样了，a 的初值为 40，b 的初值为 30，在 a<= b 这一语句之后，在 delta 时间之内，a 的值并没有立即变化到 b 所赋予它的 30 这一值，紧接其后的语句 b <= a 中，a 的值仍旧是 40，而不是 30，而在 **wait for** 10ns 这一语句之后，给信号赋的值才全部生效，因为 10 ns 比 delta 延迟的时间要长很多，足够让所赋的值生效，所以这一代码段的结果是 a 和 b 的值互相交换。由上述两段描述可以看出信号赋值与变量赋值之间的区别。

还需要注意的一点是，变量赋值用的符号是:=，而信号赋值用的符号是<=。

⑤ 为什么不用信号存储数据

当然，信号也是能够存储数据的，但不用信号存储数据有 3 个原因：

- 变量的赋值立即生效，而信号必须为此事件做一个排序处置；
- 变量所要用到的存储器比较少，而信号则需要保存更多的信息；
- 信号可能需要一个 **wait** 语句，以便为执行相同迭代信号做信号赋值的同步处理。

本例中用到的变量主要是为了进行计算的需要，与硬件的物理特性本身并没有关系。而对于信号，可以将它理解为硬件中的元件之间的连线。

(3) 函数及函数的调用

本例中用到了函数 bit_to_int，后面的第 10 例中将详细讲述该函数的写法，这里只是直接引用，其功能是将二进制数转换为相应的十进制整数值。

函数定义了以后，可在源描述需要的地方对其进行调用。VHDL 语言的函数调用与高级语言的函数调用类似，只要在合适的地方写上函数名及相应的参数即可。具体的调用请参考第 10 例和第 18 例的源描述。

（源描述文件名：4_comp.vhd）

第5例 二路选择器

袁 媛

1. 电路系统工作原理

本例针对一个多路选择器进行 VHDL 语言的描述，为简单起见，对二路选择器进行说明。此二路选择器用于从两路输入信号中选择一路信号并将其输出。其电路系统示意图如图 5.1 所示。该二路选择器有两个输入端口 in1 和 in2，一个输出端口 pout，还有一个控制端口 cnt1，其中输入端口和输出端口的类型是位向量；控制端口则是位，或者为“0”，或者为“1”。

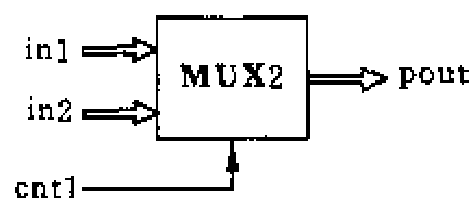


图 5.1 二路选择器

由于是进行二路选择，所以控制端口 cnt1 只需要 1 位。当 cnt1 为高电平时，选择一路，当 cnt1 为低电平时，选择另一路。相应地，如果是对 8 路信号进行选择，则需要 3 位控制端口。

2. VHDL 语言描述方法及语法分析

二路选择器的源描述如下：

```
entity mux2 is
    port(    in1  :    in  bit_vector;
             in2  :    in  bit_vector;
             pout :    out bit_vector;
             cnt1 :    in  bit );
end mux2;
architecture func of mux2 is
begin
    pout <= in1   when cnt1='0'
                  else in2;
end func;
```

本例重点在于介绍条件信号赋值语句。条件信号赋值语句是并发信号赋值语句的一

种。一条并发信号赋值语句等价于对该信号赋值的进程语句，它有两种形式：条件型和选择型，这里详述条件型信号赋值语句 **when**。

when 语句的意义是根据一个或多个信号的不同值来决定输出，如本例中的 **when** 语句：

```
pout <= in1 when cnt1=' 0'
         else in2;
```

其意义是，当 cnt1 等于 '0' 的时候选择输入 in1 作为输出，而当 cnt1 等于 '1' 的时候选择输入 in2 作为输出，这就达到了二路选择的目的。

条件信号赋值语句等价于一条给 pout 端口赋值的进程语句，可以将此条件信号赋值语句改写为一个进程，而所完成的功能不变。改写后的进程如下所示：

```
architecture func of bit_rtl_mux2 is
begin
    process(cnt1)
    pout <= in1   when cnt1=' 0'
                  else in2;
    end process;
end func;
```

（源描述文件名：5_mux2.vhd）

第6例 寄 存 器

袁 媛

1. 电路系统工作原理

本例是一个保存位向量的寄存器，它受一个时钟控制，只有在时钟的上升沿才进行操作。电路系统示意图如图 6.1 所示。该寄存器有一个时钟端口、一个输入端口、一个控制端口以及一个输出端口。

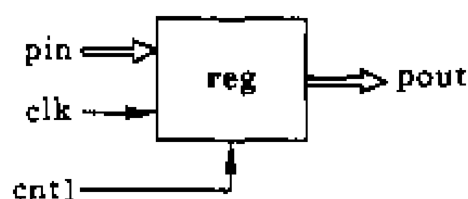


图 6.1 寄存器

此寄存器的功能是在时钟上升沿时，如果控制端 cntl=‘1’，将输入信号 pin 送到 reg 中寄存，由 pout 输出。如果 pin 发生变化，而输出条件不满足，则 reg 仍保存上一次 pin 的值，pout 中输出的值也是上一次 pin 的值，从而起到寄存的作用。

2. VHDL 语言描述方法及语法分析

本例中的重点语法现象是 **wait** 语句的使用和时钟、属性。

(1) 源描述

```
entity bit_rtl_reg_clk is
    port( pin     :   bit_vector;
          cntl    :   bit;
          pout    :   out bit_vector;
          clk     :   bit  );
end bit_rtl_reg_clk;
architecture func of bit_rtl_reg_clk is
begin
    process
    begin
    wait until clk' EVENT and clk = ' 1' ;
    if (cntl = ' 1' ) then pout <=  pin;
    end if;
    end process;
```

```
end func;
```

(2) wait 语句

在第 1 例中已提到进程中的敏感信号表，如果在一个进程中没有敏感信号表，则需要用另外的方法来重新激活进程，即使用 **wait** 语句。

wait 语句的主要形式及其作用有以下 4 种：

① **wait on** 信号表

wait on 使进程暂停，直到信号表中的某个信号值发生改变时，才激活进程。

② **wait until** 条件表达式

wait until 使进程暂停，直到条件表达式中的条件为真。

③ **wait for** 时间表达式

wait for 使进程暂停时间为表达式规定的时间。

④ **wait** 则为永远等待，一般不用。

本例用到的是第 2 种形式：

```
wait until    clk'EVENT    and    clk='1';
```

检测时钟上升沿，等待时钟上升沿到来时，才能够根据控制端口的值来决定是否寄存数据；否则，进程暂停，将时间片让给其他并发语句或进程。

在这条 **wait** 语句中，用到一个很重要的信号预定义属性 EVENT，下面详述有关属性的内容。

(3) 属性

VHDL 中的某些项目类可以具有属性。信号属性在检测信号变化和建立详细的时域模型时非常重要。这类属性报告究竟一个信号是否正好有值的变化，或者从上次事件中的跳变过了多少时间，或者该信号原来的值是什么等等。

属性 EVENT 对决定时钟边沿是非常有效的，如果一个信号位于一个特定值，并且假定该信号刚发生变化，则能推断出在信号上已发生了一个跳变沿。本例中就是用该属性来判断是否达到了时钟信号 clk 的上升沿。

属性 STABLE 用于决定在指定的时间周期内，信号是否被激活，也就是信号是刚好发生了变化，还是没有变化，最后输出的值本身是能用于启动另一个进程的信号。所以，语句

```
wait until clk'EVENT and clk='1';
```

与以下语句效果相同：

```
wait until (not(clk'STABLE)) and (clk='1');
```

但是采用’EVENT 的语句在存储空间和速度上更加有效。

(4) 时钟

在电路器件的设计中，总是要有时钟对整个器件系统的操作进行控制或同步，不同的器件系统采用不同的时钟，如四相时钟、两相时钟等。时钟信号 clk 必须定义在进程外部。本例的源描述中没有显示出时钟信号是如何定义的，常见的时钟建模方式如下所示：

①
```
process
begin
      clk <= not clk;
      wait for 10ns;
end process;
```
②
```
process
begin
      clk <= not clk after 10ns;
      wait on clk;
end process;
```
③
```
process(clk)
begin
      clk <= not clk after 10ns;
end process;
```
④
```
clk <= not clk after 10ns;
```

（源描述文件名：6_reg.vhd）

第7例 移位寄存器

袁 媛

1. 电路系统工作原理

本例是一个带移位的寄存器，电路系统示意图如图 7.1 所示。该带移位功能的寄存器有 5 个输入端口 F，CLK，I，Q0 和 Q3。CLK 是时钟控制端口，还有一个输入输出端口 Q。所谓输入输出端口是指端口的值既可以在赋值语句的左边，也可以在赋值语句的右边，换句话说，该端口的值与它本身上一次操作得到的值有关。在端口中，F 为输入数据，CLK 为时钟信号，I 为线选信号，Q0，Q3 为用于移位的数据。

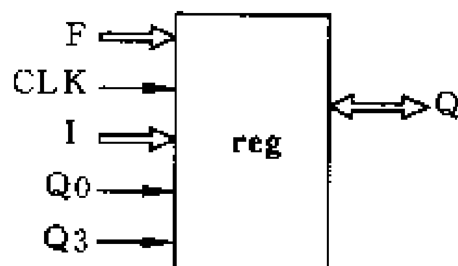


图 7.1 带移位的寄存器

当 I 的高 3 位为 000 时，输出 Q 中放入 F 中的输入数据；当 I 的高 3 位为 100 时，Q 右移 1 位，高位为 Q3；当 I 的高 3 位为 110 时，则 Q 左移一位，低位为 Q0；在其他情况下，Q 中的内容不变，从而起到移位和寄存的功能。

2. VHDL 语言描述方法及语法分析

本例的结构体是一条被保护的块语句，因此介绍的重点是被保护的语句和块语句。另外，还有测试模块，用一些测试向量模拟所描述的带移位功能的寄存器。

(1) 移位寄存器源描述

```
use work.TYPES.all;
use work.MVL7_functions.all;
use work.synthesis_types.all;

entity Q_reg is
     port (
          F    :     in MVL7_vector(3 downto 0);
          CLK  :     in clock;
```

```
        I     :    in MVL7_vector(8 downto 0);
        Q0, Q3 :   in MVL7;
        Q     :    inout MVL7_vector(3 downto 0)
        );
end Q_reg;

architecture Q_reg of Q_reg is
begin
    Q_reg1 : block ( (clk = '1' ) and (not clk' STABLE ) )
    begin
    Q <= guarded         F          when (I(8 downto 6) = "000")
        else Q3 & Q(3 downto 1) when (I(8 downto 6) = "100")
        else Q(2 downto 0) & Q0 when (I(8 downto 6) = "110")
        else Q;
    end block Q_reg1;
end Q_reg;
```

(2) 块（block）语句

块可以看作是结构体中的子模块。**block** 语句把许多并发语句包装在一起，形成一个子模块。**block** 语句的格式如下：

```
块标号：block  [（保护表达式）]
        [类属子句 [类属接口表；]]
        [端口子句 [端口接口表；]]
    < 块说明语句 >
    begin
        < 并发语句 >
end  block    [块标号]
```

类属将在第 40 例中讲解。

VHDL 中所有并发语句都是根据 **block** 语句以及等价的进程语句给出的。所有 VHDL 的分级描述和结构描述的语义都是用块或嵌套的块定义的，结构体本身就等价于一个块。

(3) 被保护的块

被保护的块含有保护表达式，该表达式可使内部的驱动源起作用或失去作用。保护表达式是布尔表达式：当布尔表达式值为 True 时，块中包括的驱动源起作用；当布尔表达式值为 False 时，所有驱动源失去作用。

本例中，块 Q_reg1 的保护条件为(clk='1')**and**(**not** clk'STABLE)。该保护条件意味着只有在时钟跳变到上升沿时，块语句中的语句才会执行。也就是说，时钟跳变到上升沿时，保护条件才为真，否则，块中的任何语句都不会执行。

在被保护的块中，隐含说明一个新的命名为GUARDED的布尔类型信号Q，信号GUARDED的值为保护表达式的值。

(4) 毗连符号&

符号&是毗连符号，意即将两个位串连接在一起，如 '0001'&'1101'就等于'00011101'。有时候，毗连符号&在条件判断中很有用，例如：

```
architecture  entity_name   of   entity_name   is
variable v, v1, v2  : bit_vector;
begin
        v    :=  v1 & v2;
    case       v     is
        when       ' 00'          =>   c <= a;
        when       ' 01'          =>   c <= b;
        when       ' 10'          =>   c <= d;
        when       ' 11'          =>   c <= e;
        when     others           =>  null;
    end   case;
end        entity_name;
```

变量 v 将 v1 和 v2 联合起来作为判断条件，从而避免了分别判断 v1 和 v2 的值，可减少语句来完成同样的功能。

(5) 七值逻辑

读者可能已经注意到本例中的数据类型不是 VHDL 语言定义的数据类型，实际上，该数据类型是在一个程序包（也简称"包"）中定义的，属于七值逻辑，此处只是直接引用。七值逻辑程序包在第 14 例中将详细叙述。

(6) 有关测试码及其编写

写完 VHDL 描述并通过编译是远远不够的，还需要知道所写模块的描述是否正确，是否完成了预期的功能以及设计的意图，这就需要写测试码并进行模拟。测试是将激励用于在测模块的模型，它将在测模块的响应与期望响应相比较，并报告模拟期间发现的差异。如图 7.2 所示。所谓在测模块是指把所描述的模块从外部当成一个整体，而不管内部包含一个元件还是成百上千个元件来进行测试。

测试码的编写不是一个简单的过程，有时模块描述比较简单，但测试码却可能非常复杂。根据测试、模拟的结果，可以对描述进行反复设计与修改，直至达到最终目的。当然，激励信号的选择应该是全面的、典型的。

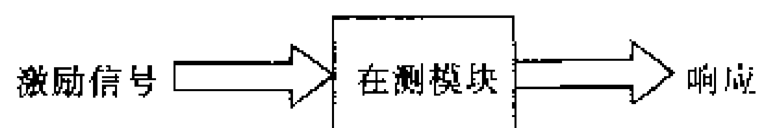


图 7.2　测试模块

下面是本例的测试码文件 7_test_vector.vhd：

```
use work.TYPES.all;
use work.MVL7_functions.all;
use work.synthesis_types.all;
entity E is
end;

architecture A of E is
    component Q_reg_inst
    port (
        F       :   in MVL7_vector(3 downto 0);
        CLK     :   in clock;
        I       :   in MVL7_vector(8 downto 0);
        Q0, Q3  :   in MVL7;
        Q       :   inout MVL7_vector(3 downto 0)
        );
    end component;
    signal F        :   MVL7_vector(3 downto 0);
    signal CLK      :   clock;
    signal I        :   MVL7_vector(8 downto 0);
    signal Q0, Q3   :   MVL7;
    signal Q        :   MVL7_vector(3 downto 0);
    for all         :   Q_reg_inst use entity work.Q_reg(Q_reg);
begin
    Q_reg_inst1     :   Q_reg_inst port map( F, CLK, I, Q0, Q3, Q );
    process
    begin
        I <= "000000000";       --#1
        F <= "0111";
        Q0 <= 'Z';
        Q3 <= 'Z';                  --将F的值装入Q中
        wait for 1 ns;
        CLK <= '1';
        wait for 1 ns;
```

```
            CLK <= '0';
            wait for 1 ns;
            assert (Q = "0111")
                report
                    "Assert 1 : < Q /= '0111' > "
                severity warning;
            wait for 1 ns;
        end process;
    end A;
```

在测试码文件 7_test_vector.vhd 中，一开头是一个空的实体，因为测试是根据被测实体的端口给它加激励信号，测试的本身并不是一个具体的元件，它只是一些激励信号，所以，**entity** 是空的。

本例的测试码共有 10 组，选取其中一组进行说明。程序段如上所示。

这组测试码中，大部分语句是给各个输入端口赋值，以便观察输出的值如何，并与寄存器的功能相对照。此处值得注意的是断言语句的引用。

(7) 断言语句

断言语句主要用于向用户报告信息，如系统信息和错误信息。断言语句用关键字 **assert** 标识。断言语句的形式如下：

assert 条件 ［**report** 报告信息］ ［**severity** 出错级别］

如果条件为真，则不执行该语句；如果条件为假，则报告[报告信息]中的信息，报告信息必须是字符串类型的一段文字。

出错级别有 4 个：note，warning，error 和 failure，这 4 个级别规定了输出字符串出错的严重程度，它允许设计者有能力按适当的类别将提示的信息分类。

级别 note 告诉用户当前正在发生的一些事情；级别 warning 警告现在的设计有错误，虽然暂时不会引起完全失败，但是如果不修改，则会引起后续程序出现问题；级别 error 则认为是出错了，必须现在就修改；而级别 failure 说明程序中可能有破坏性的错误。

本例中的断言语句是为了提示用户，如果得到的输出值不是条件中的值，那么设计一定是有问题的，也就是说，根据各个输入端口的赋值，输出端口的值应当是断言语句条件中的值。

3. 模拟及结果分析

模拟后产生的部分波形如图 7.3 所示。根据模拟波形，读者可以对照测试码分析该移位寄存器的功能。

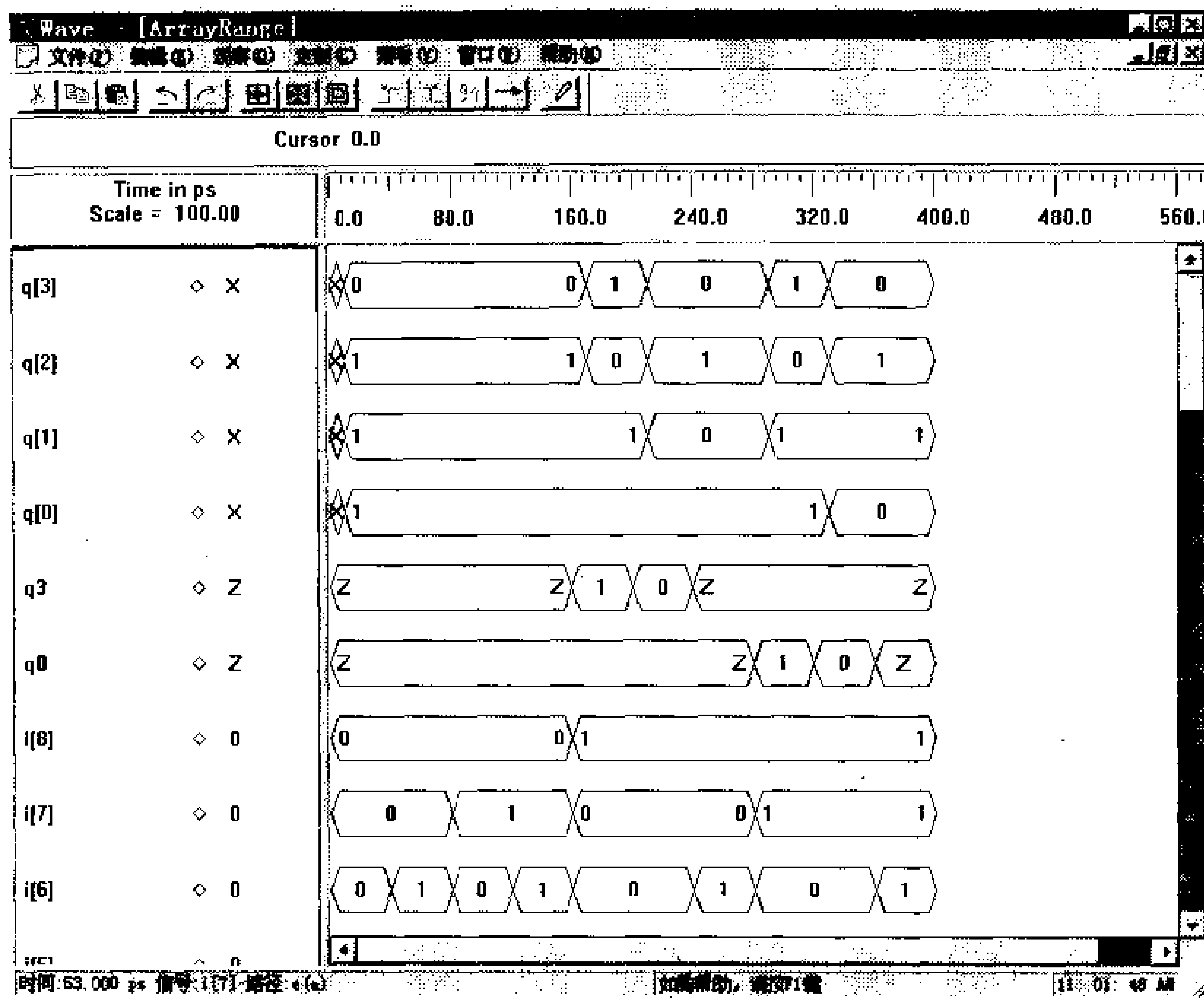


图 7.3 模拟的波形

（源描述文件名：7_shiftreg.vhd
测试码文件名：7_test_vector.vhd）

第8例 综合单元库

袁 媛

1. 电路系统工作原理

在前面的举例中，讨论的都是一些基本的元件，如加法器、比较器、寄存器等等，这些基本的元件统称为单元。电路中比较大的元件都是由基本单元组成的，所以，在设计复杂电路芯片时，就需要由上至下或由下至上地搭建，而最底层就是单元。根据不同的需要，使用不同的元件搭建不同的电路芯片。

既然基本的元件是要经常使用的，那么在 VHDL 语言中，就将这些基本元件放入一个综合单元库中，在使用时，无需再对这些基本单元进行描述，只需引用单元库中的那些已经定义好的元件，这样可以省去许多重复的工作。

本例就是一个用于高层次综合的综合单元库，前面举例中讲到的基本单元都包括在该综合单元库中。下面详细讲述有关单元库和程序包的内容，以及如何定义和使用它们。

2. VHDL 语言描述方法及语法分析

本例包括两个主要文件（VHDL 描述）：功能单元库 bit_rtl_lib 和包 bitpkg。

(1) 包 bitpkg

① 源描述

```
package bit_rtl_pkg is
--包头（包说明）
    subtype short is integer range 0 to 65535;
    --short 为数据子类型，是整数的一部分
    subtype int4  is integer range 0 to 16;
    subtype int8  is integer range 0 to 255;
    --以下均为函数头，具体定义见包体
    function bit_to_int (in1:bit_vector) return integer;
    function "+" ( in1, in2 : bit_vector ) return integer;
    function "-" ( in1, in2: bit_vector ) return integer;
    function "*" ( in1,in2: bit_vector ) return bit_vector;
    function "/" ( in1,in2: bit_vector ) return bit_vector;
end bit_rtl_pkg;

package body bit_rtl_pkg is
--包体，由关键字 package body 标识
```

```
--函数 bit_to_int 将位向量转化为十进制整数值
function bit_to_int (in1:bit vector) return integer is
    alias v1: BIT_VECTOR(in1' LENGTH-1 downto 0) is in1;
    variable inpv : bit_vector(in1' LENGTH-1 downto 0 );
    variable SUM: integer:= 0;
    variable negative :boolean:= false;
begin
    inpv := in1;
    if  v1(v1'  LENGTH -1) = ' 1'  then
        for i in v1'  LENGTH -1 downto 0 loop
            inpv(i) := not inpv(i);
        end loop;
    lp1:
        for i in 0 to v1' LENGTH-1 loop
            if   inpv(i) = ' 1'  then
                inpv(i) := ' 0' ;
            else
                inpv(i) := ' 1' ;
                exit lp1;
            end if;
        end loop;
    negative := true;
    end if;
    for  i in v1' LENGTH-1 downto 0 loop
        if inpv(i) = ' 1'  then SUM := SUM + 2**i;
        end if;
    end loop;
    if negative then return (0-SUM);
    else return SUM;
    end if;
end bit_to_int;

    --重载函数 "+" 将两个位向量相加，得到相应的整数值
function "+" ( in1, in2 : bit_vector )    return integer is
begin
    --用到了前面定义的函数 bit_to_int，将两个位向量先分别转化为
    --整数，然后再相加
    return (bit_to_int(in1)+bit_to_int(in2));
end "+";

--重载函数 "-" 将两个位向量相减，得到相应的整数值
function "-" ( in1, in2 : bit_vector ) return integer is
begin
    return (bit_to_int(in1)-bit_to_int(in2));
end "-";
```

```
--重载函数 "*" 将两个位向量相乘，得到相应的整数值
function "*" ( in1, in2 : bit_vector ) return integer is
begin
        return ( bit_to_int(in1)*bit_to_int(in2) );
    end "*";

    --重载函数 "/" 将两个位向量相除，得到相应的整数值
    function "/" ( in1, in2 : bit_vector ) return integer is
    begin
        return (bit_to_int(in1)/ bit_to_int(in2));
    end "/";
end bit_rtl_pkg;
```

② 包

在实体说明和结构体内部定义的数据类型、常量及子程序对其他设计单元是不可见的。为了使类型、常量及子程序对若干个设计单元可见，VHDL 提供了包机制。包是一个库单元（设计单元），包含有可用于其他设计单元的一系列说明。

bitpkg 包中包含了功能单元库中用于描述基本单元的一系列说明。

一般的包包括两个部分：说明部分及包体部分，bitpkg 也不例外。在说明部分中有各种类型说明、常量说明以及过程、函数说明；子程序体（过程、函数）则必须放在包体中。

bitpkg 内包含有 bit_rtl_lib 所需用到的类型及函数定义，读者可以把高级语言（如 C 语言）中的头文件及过程函数定义与 VHDL 中的包相比较，以便更好地理解。

③ 重载函数 **function** "+"

前面的举例中已经提到过重载，因为重载是一个很重要的概念，而且本例中又有大量的重载函数，所以再重新提起。

这里所指的重载与高级语言面向对象设计中的"重载"概念是一样的。**function** "+" 是一个操作符重载函数，它所定义的"+"操作符已不是普通意义上的相加。**function** "+" 的功能是把两个位向量逐位相加，并返回相加后的位串。描述段如下：

```
function "+" (in1,in2:bit_vector) return bit_vector is
begin
return
    (int_to_bit1(bit_to_int(in1)+bit_to_int(in2),in1'LENGTH);
end "+";
```

算法是一目了然的，其中还用到了在 bitpkg 中定义的两个函数 bit_to_int 及 int_to_bit。bit_to_int 将在第 10 例中详细讲述，int_to_bit 可在源描述中找到，情况类似。从上述函数中也可看到 VHDL 语言中是如何调用函数的。

本例中还有许多其他的重载函数，如"*"，"/"，"-"等，道理都是一样的，读者可

以参考本书附带光盘中的源描述 8_bit_rtl_pkg.vhd，逐一研究。

有关 bitpkg 先讨论到这里，其中还有许多函数及类型定义，读者可以自己仔细阅读源描述，这些函数在写 VHDL 描述时都是常用的，应该熟悉它们。

(2) 功能库 bit_rtl_lib

① 源描述

```
use work.bit_rtl_pkg.all;
--USE 子句引用程序包中的一切内容
entity bit_rtl_mux2_bit is --带控制端口的二选一多路选择器，输入输出为位
port (
      in1 : in bit;
      in2 : in bit;
      pout : out bit;
      cntl: in bit );
end bit_rtl_mux2_bit;

architecture func of bit_rtl_mux2_bit is
begin
   pout <= in1      when cntl =' 0'
                    else  in2;
end  func;

use work.bit_rtl_pkg.all;
entity bit_rtl_adder is  --带控制端口的加法器，输入输出为位向量
      port
        in1  : bit_vector;
        in2  : bit_vector;
        cntl : bit;
        pout : out bit_vector
  );
end bit_rtl_adder;

architecture func of bit_rtl_adder is
begin
   process(cntl)              --带有信号敏感表的进程，敏感信号为 cntl
   begin
       if (cntl = ' 1' ) then     pout <= in1+in2;
      end if;           --如果控制信号为 '1'，则做加法
   end process;
end  func;

use work.bit_rtl_pkg.all;
entity bit_rtl_substracter is  --减法器，输入输出为位向量，带个控制端口
```

```
   port
       in1  : bit_vector;
       in2  : bit_vector;
       cntl : bit;
       pout : out bit_vector
   );
end bit_rtl_substracter;

architecture func of bit_rtl_substracter is
begin
   process(cntl)
   begin
      if (cntl = '1') then pout <= in1-in2 after 1ns ;
      end if;
   end process;
end  func;

use work.bit_rtl_pkg.all;
entity bit_rtl_multiplier is  --带控制端口的乘法器，输入输出为位向量
   port
       in1  : bit_vector;
       in2  : bit_vector;
       cntl : bit;
       pout : out bit_vector
   );
end bit_rtl_multiplier;

architecture func of bit_rtl_multiplier is
begin
   process(cntl)
   begin
      if (cntl = '1') then pout <= in1*in2;
      end if;
   end process;
end  func;

use work.bit_rtl_pkg.all;
entity bit_rtl_divider is  --带一个控制端口的除法器，输入输出均为位向量
   port (
       in1  : bit_vector;
       in2  : bit_vector;
       cntl : bit;
       pout : out bit_vector
   );
end bit_rtl_divider;
```

```
architecture func of bit_rtl_divider is
begin
   process(cntl)
   begin
      if (cntl = '1') then pout <= in1 - in2;
      end if;
   end process;
end func;
use work.bit_rtl_pkg.all;
entity bit_rtl_lt_nc is  --不带控制端门的比较器“>”，输入输出均为位向量
   port (
       in1  : bit_vector;
       in2  : bit_vector;
       pout : out bit
   );
end bit_rtl_lt_nc;

architecture func of bit_rtl_lt_nc is
begin
   process(in1, in2)
       variable left : integer;
       variable right: integer;
   begin
       left  := bit_to_int(in1);
       right := bit_to_int(in2);
       if ( left < right ) then pout <= '1'  after 1 ns;
      else pout <= '0'  after 1 ns;
      end if;
   end process;
end  func;

entity bit_rtl_reg_clk is  --寄存器，输入输出均为位向量
   port (
       pin  : bit_vector;
       cntl : bit;
       clk  : bit;
       pout : out bit_vector
   );
end bit_rtl_reg_clk;

architecture func of bit_rtl_reg_clk is
begin
   process
   begin
       wait until clk'EVENT and clk ='1' ;
       if (cntl = '1') then pout <= pin ;
```

```
      end if;
    end process;
end func;
```

② 包的引用

在 bit_rtl_lib 描述的开始，有这样一条语句：

```
use work.bit_rtl_pkg.all
```

用 **use** 子句可以使包中的所有说明被引用时可见。work 库为当前设计单元库，bit_rtl_pkg 是 bitpkg 的包名，**all** 是指引用所有说明。因此，该条语句指出 bit_rtl_lib 可以使用 bit_rtl_pkg 中所定义的所有类型、常量、函数等内容。

无论是自己定义的包也好，标准的包（如 IEEE 包）也好，都可以用 **use** 子句来引用，只要引用包，该包中定义的一切内容都可供使用，十分方便。就像在 Visual C++语言中，将一些变量定义和方法定义都放在.H 文件中，供其他 C++程序使用一样。

③ 功能单元库

bit_rtl_lib 是一个自定义的设计库，它包含许多常用的功能单元，如多路选择器、加法器、寄存器等等。

所谓 VHDL 设计库是将已分析的设计单元以文件方式保存起来，以便供其他设计单元使用。设计库中的设计单元可用作其他 VHDL 描述的资源，可以通过如下语句引用 bit_rtl_lib 中的资源。

```
library bit_rtl_lib;
use bit_rtl_lib.all;
```

如果只是引用其中某特定项，则将 **all** 改为该项名称即可。

④ 底层单元行为描述

在 bit_rtl_lib 中，将所定义的一些功能单元作为底层单元进行描述，以供上层单元直接引用。作为底层单元，一般用行为描述进行定义，一方面因为底层单元用行为描述比较直观，另一方面它们中有一些不是由其他单元组装起来的，也无法用结构描述。

在前面的举例中只提取了 bit_rtl_lib 功能单元库中的几个典型示例，其他功能单元类似，有的只是端口位数不同。这个单元库中描述的都是常用的功能单元，读者可仔细阅读源描述，并且可以在自己书写的 VHDL 描述中引用它们。

（源描述文件名：8_bitpkg.vhd
8_bit_rtl_lib.vhd）

第9例　七值逻辑与基本数据类型

袁　媛

本例的关键在于解释七值逻辑，描述和语法都比较简单。这里主要讲述如何自定义数据类型，是七值逻辑包（第14例）中的一部分。

1. 逻辑系统原理

数字系统内部信息的表示和传输通常有两个状态，分别表示0和1，这是理想化的模型。但是，数字系统是千变万化的，而且在大部分情况下都不可能达到理想化的程度。因此，有必要使用多值逻辑，如三值逻辑、六值逻辑、七值逻辑以及九值逻辑等。用这些多值逻辑来表示复杂的模拟信号状态值。

多值逻辑有它们各自的优点及不足，如二值逻辑可以检查稳定情况下逻辑上的正确性，而对于过渡瞬态情况无法体现，不能检查线路中的竞争、冒险等问题；三值逻辑的模拟信号更接近真实信号的波形，能够解决竞争和冒险问题，但是仍有理论上的缺陷，常得出比实际情况更坏的结果；四值逻辑能够使用户给定的初值在线路中得以保存和传播；为了克服三值逻辑理论上的不足，提出了六值逻辑；为了弥补三值逻辑不能推测在组合线路中存在动态冒险的缺陷，提出了七值、八值甚至九值逻辑。所以，逻辑信号状态值的选择对于数字电路的设计是十分重要的。

在七值逻辑中，表示电路器件所处的7种状态，如下所示（信号的强弱从左至右递减）：

'X'（高阻不定），'0'（逻辑0或假），'1'（逻辑1或真），

'Z'（高阻），'W'（弱不定），'L'（弱0），'H'（弱1）

一般在设计数字电路的时候，要根据系统功能要求的不同采用不同的逻辑。本例将七值逻辑定义在一个包中，这样，用户在需要时可直接引用，具体的七值逻辑包在第14例中有完整的叙述，请参考后续举例。七值逻辑源描述如下：

```
package MVL7_types is
type MVL7 is (     'X',            -- 高阻不定
                   '0',            -- 逻辑0或假
                   '1',            -- 逻辑1或真
                   'Z'             -- 高阻
                   'W'             -- 弱不定
                   'L'             -- 弱0
                   'H'             -- 弱1
```

```
        );
type MVL7_VECTOR is array ( natural range <>) of MVL7;
end package MVL7_types;
```

2. VHDL 语言描述方法及语法分析

(1) 数据类型

我们已经接触到 VHDL 语言中所有的对象（信号、变量和常量），而为了规定这些对象的特征，可用数据类型的详细说明来描述它们。其实，我们也已经接触到 VHDL 语言中的许多数据类型，在此只是做一个归纳总结。

VHDL 语言中的数据类型包括有标量类型、复合数据类型、寻址类型、文件类型和子类型等 5 种，下面分别讲述。

① 标量类型

标量包括下面 4 种类型：整数类型（integer）、实数类型（real）、可枚举数据类型和物理类型。前三种类型都很好理解，分别举例说明：

```
variable  a : integer;       a := 1;
signal    b : real;          b <= 1.5E-20;
type  color   is   (red, yellow, blue, green, orange);
```

既然 VHDL 语言是一种硬件描述语言，而物理特性又是硬件的重要特征（如延迟），那么 VHDL 语句自然就应该包括物理类型，而高级语言中一般是不会有物理类型的。

用物理类型表示距离、电流和时间等物理量。物理类型提供基本单位，然后在这种单位条目中定义次级单位。例如对物理量电流的物理类型可描述如下：

```
type current  is range 0   to   1000000000
units
        na;                          -- 纳安
        ua   =    1000 na;           -- 微安
        ma   =    1000 ua;           -- 毫安
        a    =    1000 ma;           -- 安培
end units;
```

VHDL 只有一个预定义物理类型：时间（TIME），如飞秒、纳秒、皮秒等等，用得比较多的是纳秒。

② 复合数据类型

复合数据类型包括数组类型和记录类型，与高级语言类似，不再赘述。

③ 寻址类型

多数用 VHDL 的硬件工程师或许绝不直接使用寻址类型，但是寻址类型提供了一种非常有效的与高级语言中的指针非常类似的操作，可以认为它是一种寻址，或者处理一种

特殊的对象。下面举例说明：

```
type    Line        is  access       String;
variable  line_buffer : Line;
```

④ 文件类型

在主系统环境中用文件类型定义代表文件的对象，文件对象的值是主系统文件中值的序列。文件对象类型实际上是一个变量对象类型的子集，变量对象能够用变量赋值语句赋值，而文件对象却不能被赋值。文件对象只能由规定的过程和函数读出、写入并检查文件的结尾。

(a) 文件类型说明

文件类型说明指定文件类型名字和文件的基本类型。举例说明如下：

```
type  integer_file is   fileinteger;
```

(b) 文件对象说明

文件对象产生一个文件类型的用法和说明一个文件类型的对象，文件对象说明指出文件对象的名字、文件的模块和实际硬盘通道的名字。举例说明如下：

```
FILE myfile : integer_file is in "/bit903/test/examples/data_file";
```

⑤ 子类型

用子类型说明定义类型的一个子集，该子集包括整个的基本类型范围但并不一定都在此子类型内，一般子类型都加有对现有类型的限制。举例说明如下：

```
type      integer    is  -2,147,483,647to   +2,147,483,647;
subtype   NATURAL   is   integer   0    to   +2,147,483,647;
```

(2) 自定义数据类型

本例采用枚举类型自定义七值逻辑，而一般枚举类型都是用户自定义的。数据类型MVL7_VECTOR 则是定义的七值逻辑位向量。在包中定义了这些数据类型以后，用户就可以直接使用。

（源描述文件名：9_MVL7_types.vhd）

第10例 函　　数

袁　媛

函数实际上包含很多的内容，在不同的示例中有不同的形式及应用，在一个单独的示例中不可能包含函数的全部内容。这里只是讨论本例涉及到的函数的某些内容。有关函数的其他应用，请参考后续举例。

1. bit_to_int 函数的功能

本例是一个将二进制数转换为十进制整数值的函数，在第 4 例中，已经对该函数进行了调用，下面详细讲述该函数。

2. VHDL 语言描述方法及语法分析

在 VHDL 语言中，函数的形式与高级语言中的函数十分类似，但是由于 VHDL 语言是一种硬件描述语言，所以涉及许多与电路有关的特性。

(1) 函数的一般形式

函数的一般形式如下：

```
函数定义        is
                函数说明部分
begin
     函数语句部分
end       [函数类型]          [函数名]
```

在函数定义部分，要说明函数名、函数的参数及返回的类型；在函数说明部分，有类似于进程语句的说明区，说明变量、常量和类型，但不能说明信号；在函数语句部分用具体的语句说明此函数的功能，这些语句包括全部并发语句。

另外还需要注意，函数总是要返回一个值，该值的类型也在函数说明部分说明。

(2) 函数源 bit_to_int 的描述

```
1    function bit_to_int(in1:bit_vector) return interger is
2    alias V1:bit_vector(in1' LENGTH-1 downto 0) is in1;
3    variable inpv:bit_vector(in1'  LENGTH -1 downto 0);
4    variable sum:integer:=0;
5    variable negative:boolean:=false;
```

```
6    begin
7    inpv:=in1;                  --将要转换的数放入一临时变量
8    if V1(V1' LENGTH 1)= ' 1'  then
9        for I in V1' LENGTH -1 downto 0 loop
10            inpv(I):=not inpv(I);
11        end loop;
12   --以上是判断位向量所代表的数是否为负数（最高位为1则为负），
13   --如果是，则转化为正数。
14   lp1:                        --此为循环语句标号
15   for I in 0 to V1' LENGTH -1 loop
16       if inpv(I)= ' 1'  then inpv(I):= ' 0'
17       else inpv(I):= ' 1'  ;   exit lp1;
18   end if
19   end loop;
20   negative:=true;        --negative为正负标志位。
21   end if;
22   --到此为止，将负数取反码加1得到正数。
23   --以下将二进制位转化为正数。
24   for I in 0 to V1' LENGTH -1 loop
25    if inpv(I)= ' 1'  then
26        sum:=sum+2**I;
27    end if;
28   end loop;
29   --如果正负标志位为true，表明要返回的应为一负数。
30   if negative then return(0-sum);
31   else return sum;
32   end if;
33   end bit_to_int
```

(3) 别名

关键字 **alias** 用来标识一个变量的别名。别名典型用于数组或为指令的每个子字段提供命名机制。

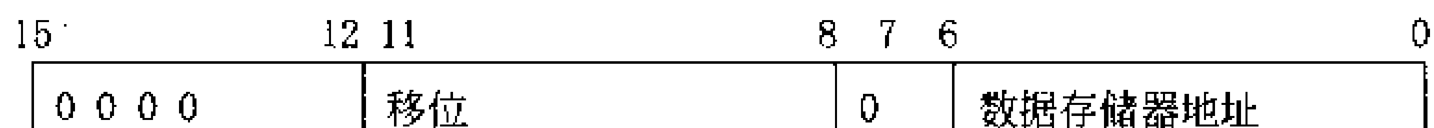
① 在数组中的使用

本例中，在描述的第2行将输入参数命名为别名V1，为什么要取别名呢？因为有的数组、位向量是以枚举变量做下标（如四值逻辑中，BV('Z', 'X')= 'Z'等，请参看第38例），如果要进行循环处理就不方便了。给参数取个别名可以用整数做下标，与原来的下标一一对应，并可按1,2,3,4排序，相当于进行转化。即使用别名可以限定形式参数、数组变量的大小和方向，这样再进行其他处理就方便多了，例如可使用别名中的整数下标作为步长来控制循环。

② 在指令中的使用

在计算机的指令中，很多指令都包含许多子字段，如SUN工作站的SPARC MB86901

芯片中的一条加法指令 ADD，它的机器代码如下：



15　　12	11　　8	7	6　　0
0 0 0 0	移位	0	数据存储器地址

该机器代码总共 16 位，15 到 12 位代表此为加法指令，11 到 8 位代表移位，第 7 位表示此加法为直接寻址的加法，6 到 0 位则为存储器地址。显然，这 4 个子字段代表不同的意义，可以用 4 个别名分别代表它们，这样在进行描述时就方便多了。

(4) 预定义属性 LENGTH

预定义属性是值类数组属性，对给定的数组类型，这个属性将返回指定的数组范围的总长度，此属性用于带某种标量类型的数组范围和带标量类型范围的多维数组。

本例中用属性 LENGTH 检测位向量的位数。

(5) 循环语句 loop

从本例循环语句中，可以看到取别名的好处，取了别名以后，限定了形式参数的大小和方向，就可以很方便地进行循环处理。

一条 **loop** 语句包含要重复执行的一组顺序语句，执行零次或[重复模式]中规定的次数。一般格式如下：

```
[loop 标号：] [重复模式] loop
               顺序语句；
    end   loop  [loop 标号]；
```

重复模式有 **while** 与 **for** 两种，类似于高级语言中的 **while** 和 **for** 循环。若无重复模式，则为无限循环。

本例中采用的是 **for** 模式，循环的次数由 **for** 后面的变量值决定。因为本例中要对位向量中的每一位进行处理，所以采取循环的形式，这与高级语言中为什么要采用循环的道理一样。

（源描述文件名：10_bit_to_int.vhd）

第11例 七值逻辑线或分辨函数

袁 媛

分辨函数（resolved function）是VHDL语言中具有代表性的一类函数，本例是一个线或（WiredOr）分辨函数。

1. 分辨函数的功能

根据驱动器件工艺的不同，有多个驱动源的信号可以表现为竞争信号的“或”关系或者“与”关系，分辨函数就是为了解决这一问题而设计的。

本例中分辨函数的功能是从一个信号的多个驱动源中选择一个作为实际的输出结果。因为是线或，所以如果驱动源有一个为‘1’，则返回‘1’，否则根据七值逻辑表的强度大小来决定返回结果。

此分辨函数的参数是一个七值逻辑的位向量，参数位向量中各位都是要返回的结果信号的驱动源，而返回结果则是一个七值逻辑位，这一位就是参数中位向量的一位。

2. VHDL语言描述方法及语法分析

分辨函数是用户定义的函数，它返回带有多个驱动源信号的单一值。分辨函数用于实现适当的冲突仲裁。在每个模拟周期内，对应的欲分辨信号活跃时，分辨函数被隐含地调用。即不论何时对有多个驱动源的信号的驱动源赋值，必须调用分辨函数来决定实际的结果。用户不能控制该函数调用的发生。

(1) 函数WiredOr的源描述

```
function WiredOr (V: MVL7_VECTOR) return MVL7 is
-- 线或函数的真值表
   constant tbl_WIREDOR: MVL7_TABLE :=
    ((' X' , ' X' , ' 1' , ' X' , ' X' , ' L' , ' H' ),
     (' X' , ' 0' , ' 1' , ' 0' , ' 0' , ' L' , ' H' ),
     (' 1' , ' 1' , ' 1' , ' 1' , ' 1' , ' 1' , ' 1' ),
     (' X' , ' 0' , ' 1' , ' Z' , ' W' , ' L' , ' H' ),
     (' X' , ' 0' , ' 1' , ' W' , ' W' , ' W' , ' W' ),
     (' L' , ' L' , ' 1' , ' L' , ' W' , ' L' , ' W' ),
     (' H' , ' H' , ' 1' , ' H' , ' W' , ' W' , ' H' ));
  variable result: MVL7;
begin
```

```
    result := 'Z';
    for i in V'RANGE loop
      result := tbl_WIREDOR(result, V(i));
      exit when result = '1';
    end loop;
    return result;
  end WiredOr;
```

(2) 信号驱动源

当一个进程给信号赋值时，也许要经过一段指定的时间之后才让该信号生效，即在给定时间之后安排一个事务。一个信号可能要安排任意数目的事务。一个信号的事务的序列称为该信号的驱动源。给某个信号赋值的任意一个进程（或等价进程）都给该信号建立一个驱动源。无论在进程中给信号赋几次值，每个进程对一个信号赋值都只给该信号建立一个驱动源。

对信号而言，如果是分辨信号，那么可能有几个驱动源，否则只有一个驱动源。分辨信号意味着该信号对应一个分辨函数，用于对各驱动源进行分辨，成为一个事务。

(3) 分辨信号

分辨信号包括多个驱动源和一个分辨函数。本例中线或的分辨函数定义如下：

```
function WiredOr (V: MVL7_VECTOR) return MVL7;
```

那么，就可以按如下方式定义一个称为 S1 的分辨信号：

```
signal S1 : WiredOr MVL7;
```

(4) 分辨函数

分辨函数的输入必须是与分辨信号类型相同的一维非限定性数组。返回类型必须与信号类型匹配。在书写分辨函数时，不能人为地设定竞争源到达的次序。

在 VHDL 中，与一个给定实体输出端口相关的分辨函数用于解决该实体结构内部的冲突。

(5) 七值逻辑分辨函数

在第 9 例中已给出七值逻辑，为了定义分辨函数，我们做如下简单规定：

① 最强的强度总是赢家；

② 如果强度相同而值不同，返回的强度相同但值为'X'。

可以根据本例中常量 tbl_WIREDOR 的定义来判断分辨函数应该选取哪个作为返回值。可以将常量 tbl_WIREDOR 理解为表 11.1。

本例中，每个驱动源的值和存储在 result 变量中的当前值比较，按照表 11.1 规定的原则，如果新值更强，那么将用新值更新当前值。根据表 11.1，如果 result 的当前值为 Z，而当前驱动源的值为 W，则 result 的新值为 W，依此类推。

表 11.1　七值逻辑的线或分辨

	X	0	1	Z	W	L	H
X	X	X	1	X	X	L	H
0	X	0	1	0	0	L	H
1	1	1	1	1	1	1	1
Z	X	0	1	Z	W	L	H
W	X	0	1	W	W	W	W
L	L	L	1	L	W	L	W
H	H	H	1	H	W	W	H

(6) 属性 RANGE

RANGE 是一个范围类属性，此属性限定仅用于数组类型，并由所选的输入参数返回指定的指数范围。

a'RANGE[(n)]中属性 RANGE 将返回由参数 n 的值指明的某一个范围，属性 RANGE 返回按指定排序的范围，也可以不带参数 n。

可以用属性 RANGE 决定输入矢量的范围，然后在循环语句中该范围用来决定循环执行的次数并做完整个转换。本例中，属性 RANGE 所起的就是如上所述的作用。

(7) exit 语句

exit 语句用在循环语句内部。它可有条件或无条件地终止当前循环迭代并终止该循环。若 loop 标号缺省，则 **exit** 语句作用于当前最内层循环，否则转到指定的循环中。

本例中，由于是线或分辨函数，所以只要有一个驱动源为'1'，则返回的必然为'1'，不再需要继续判断，所以使用 **exit** 语句退出循环。

（源描述文件名：11_wiredor.vhd）

第12例　转 换 函 数

袁　媛

1. 转换函数的功能

本例的转换函数是将一个整型值转换为七值逻辑的位向量。这是为了让用户能够更方便地使用七值逻辑程序包而设计的。

该转换函数有两个形式参数：需要转换的整型数 number 及该整数的长度 len。返回值则为经过转换以后的一个七值逻辑位向量。

2. VHDL 语言描述方法及语法分析

(1) 转换函数源描述

```
function I2B( Number : integer ; len : integer) return MVL7_VECTOR is
  variable temp: MVL7_VECTOR (len - 1 downto 0);
  -- 临时变量，保存得到的位向量的各个位
  variable NUM: integer:=0;
  -- 计数变量
  variable QUOTIENT: integer:=0;
  -- 整数不断除以 2 得到的余数
begin
  QUOTIENT := Number;
  for I in 0 to len - 1 loop
      NUM := 0;
      while QUOTIENT > 1 loop
         QUOTIENT := QUOTIENT - 2;    --不断除以 2，得到余数
         NUM := NUM + 1;
      end loop ;
      case QUOTIENT is
         when 1 =>
            temp(I) := '1' ;
         when 0 =>
            temp(I) := '0' ;
         when others =>
            null;
      end case;
      QUOTIENT := NUM;
    end loop ;
  return temp;
```

```
end;
```

(2) 转换函数

VHDL 属于强类型语言，每一个对象只能有一种类型，并且只能取该类型的值。由于 VHDL 中没有隐式类型转换，所以在赋值时，若值类型和对象类型不一致，需要使用显式类型转换，而转换函数就可以用来将一种类型的对象转换到另一种类型的对象，在元件例示语句中允许不同类型的信号和端口之间进行映射。通常当设计者想用的实体来自另一个设计时就会出现这种类型转换的情况。可在程序包中定义许多类型的转换函数。

(3) while 循环模式

在第 10 例中提到过 VHDL 语言有两种循环模式，一种是 **for** 循环模式，而另一种是 **while** 循环模式。在 **while** 循环中，关键字 **while** 后面紧接着的是代表循环次数的变量。与 **for** 循环模式不同的是，在 **while** 循环中，可以根据需要改变循环的步长。在高级语言中，**for** 与 **while** 循环的区别也是如此。

(4) case 语句

case 语句的一般格式如下：

```
case      表达式        is
    {when    表达式值   =>  顺序语句};
    [when    others   =>  顺序语句; ]
end case;
```

case 语句的作用是根据表达式的各个不同的值来执行不同的顺序语句，完成不同的功能，它属于分支语句。

本例中，如果十进制整数值为 1，则将位'1'放到七值逻辑位向量数组中；如果得到的十进制整数值为 0，则将位'0'放到七值逻辑位向量数组中；否则不进行任何操作，**null** 是 VHDL 语言中的空语句，表示不进行任何操作。

（源描述文件名：12_convert.vhd）

第13例 左移函数

袁 媛

1. 左移函数的功能

本例是一个左移函数，其源描述如下：

```
function SHL(v2: MVL7_VECTOR ; fill : MVL7 ) return MVL7_VECTOR is
  variable v1: MVL7_VECTOR (v2' HIGH downto v2' LOW);
  variable shift_val: MVL7_VECTOR (v1'HIGH downto v1'LOW);
  variable I: integer;
begin
  v1 := v2;
  -- 变量 v1 先保存参数 v2 的值，只对 v1 进行处理
  for I in v1' HIGH downto (v1' LOW + 1) loop
     shift_val(I) := v1(I - 1);
  end loop ;
  -- 将 v1 的每一位向左移一位，
  -- 移位后的新位向量保存在 shift_val 数组中
  shift_val(v1' LOW) := fill;      -- 最后一位由参数 fill 填充
  return shift_val;                -- 返回移位后的位向量
end;
```

此左移函数将一个七值逻辑位向量向左移一位，而最后一位用参数 fill 所指定的位来填充。

该函数有两个参数，一个是要向左移位的七值逻辑位向量，另一是为用来填充移位后位向量的最后一位参数，该参数的数据类型为位。返回值仍然是一个七值逻辑位向量。

2. VHDL 语言描述及语法分析

本例比较简单，也很好理解，许多语法现象在前面举例中已出现过。这里主要说明两个属性：HIGH 和 LOW。

HIGH 和 LOW 是两个值类型属性。HIGH 返回类型或者子类型的上限值，而 LOW 则返回类型或者子类型的下限值。类型或子类型的上限值是具有最大值的边界，而下限值是具有最小值的边界。例如，对于下面定义的 smallint 子类型，它的上限值是 32767，而下限值是-32767。

```
type smallint is -32767 to 32767;
```

本例中，变量 v1 位数组的大小是从参数 v2 的上限值到下限值，而变量 shift_val 位数组的大小是从变量 v1 的上限值到下限值。

（源描述文件名：13_shl.vhd）

第14例　七值逻辑程序包

袁　媛

1. 程序包的功能

第8例介绍的综合单元库是一个程序包，其中主要包含一些典型的单元。而本例中的七值逻辑包主要包含的是一些典型的逻辑函数，如左移函数、右移函数以及循环移位函数等等。在第10~13例中已经提取了其中的几个函数作为代表，对它们进行了详细的讨论。本例程序包中的函数不仅可以直接调用，也为编写这一类函数提供了一个模板。

2. 源描述及注释

```
MVL7 function

use work.TYPES.all;
-- 以下是包头，含有所有函数的定义
package MVL7_functions is
   function SHL(v2: MVL7_VECTOR ;  fill :  MVL7 ) return MVL7_VECTOR;
   function SHL0(v2:MVL7_VECTOR;dist:  integer) return MVL7_VECTOR;
   function SHL1(v2:MVL7_VECTOR;dist:  integer) return MVL7_VECTOR;
   function SHR(v2:MVL7_VECTOR; fill  : MVL7 )  return MVL7_VECTOR;
   function SHR0(v2:MVL7_VECTOR;dist:  integer) return MVL7_VECTOR;
   function SHR1(v2:MVL7_VECTOR;dist:  integer) return MVL7_VECTOR;
   function ROTR(v2:MVL7_VECTOR;dist:  integer) return MVL7_VECTOR;
   function ROTL(v2:MVL7_VECTOR;dist:  integer) return MVL7_VECTOR;
   function I2B(Number:  integer;  len  :  integer)  return MVL7_VECTOR;
   function B2I( v2  :   MVL7_VECTOR  ) return integer;
   function COMP( v2  :   MVL7_VECTOR  ) return MVL7_VECTOR;
   function TWOs_COMP( v2  :   MVL7_VECTOR  )  return MVL7_VECTOR;
   function ODD_PARITY( v1  :   MVL7_VECTOR  )  return MVL7;
   function EVEN_PARITY( v1  :   MVL7_VECTOR  )  return MVL7;
   function REVERSE( v2  :   MVL7_VECTOR  )  return MVL7_VECTOR;
   function SUM( v2  :   MVL7_VECTOR  ) return integer;
   function PAD(v:MVL7_VECTOR ; width  :   integer) return MVL7_VECTOR;
   function DEC( x  :   MVL7_VECTOR  ) return MVL7_VECTOR;
   function INC( x  :   MVL7_VECTOR  ) return MVL7_VECTOR;
   function CARRY_ADD( x1:MVL7_VECTOR;x2  :MVL7_VECTOR) return MVL7_VECTOR;
   -- 以下3个是重载函数
  function "+"  ( x1  :   MVL7_VECTOR  ;   x2  :   MVL7_VECTOR  )  return MVL7_VECTOR;
  function "-"( x1  :   MVL7_VECTOR  ;   x2  :   MVL7_VECTOR  )  return MVL7_VECTOR;
```

```
  function "*" ( x1 :  MVL7_VECTOR ;  x2 :  MVL7_VECTOR ) return MVL7_VECTOR;

-- 用于"WiredOr" 线或函数的真值表
-- 常量定义
   constant tbl_WIREDOR: MVL7_TABLE :=
--  ----------------------------------------------------
--  | X    0    1    Z    W    L    H |
--  ----------------------------------------------------
    (('X', 'X', '1', 'X', 'X', 'L', 'H'),  -- | X |
     ('X', '0', '1', '0', '0', 'L', 'H'),  -- | 0 |
     ('1', '1', '1', '1', '1', '1', '1'),  -- | 1 |
     ('X', '0', '1', 'Z', 'W', 'L', 'H'),  -- | Z |
     ('X', '0', '1', 'W', 'W', 'W', 'W'),  -- | W |
     ('L', 'L', '1', 'L', 'W', 'L', 'W'),  --   L |
     ('H', 'H', '1', 'H', 'W', 'W', 'H')); -- | H |

  --线或分辨函数
  function WiredOr (V: MVL7_VECTOR) return MVL7;
end;

package body MVL7_functions is

--***********************************************************
   -- 该左移函数在第 13 例中有详细的讲述
   function SHL(v2: MVL7_VECTOR ; fill : MVL7 ) return MVL7_VECTOR is
      variable v1: MVL7_VECTOR (v2'HIGH downto v2'LOW);
      variable shift_val: MVL7_VECTOR (v1'HIGH downto v1'LOW);
      variable I: integer;
   begin
      v1 := v2;
      for I in v1'HIGH downto (v1'LOW + 1) loop
         shift_val(I) := v1(I - 1);
      end loop ;
      shift_val(v1'LOW) := fill;
      return shift_val;
   end;

--***********************************************************
   -- 左移函数，但最后一位用'0'填充
   function SHL0(v2:MVL7_VECTOR;dist: integer) return MVL7_VECTOR is
      variable v1: MVL7_VECTOR (v2'HIGH downto v2'LOW);
      variable I: integer;
   begin
      v1 := v2;          -- 取出参数的值进行处理，以免破坏原值
      for I in 1 to dist loop
         v1 := SHL(v1, '0');         -- 左移一位，最后一位用'0'填充
```

```
                                        -- 调用了前面定义的SHL函数
      end loop;
      return v1;
   end;
--*************************************************************
   -- 左移函数，但最后一位用'1'填充
   function SHL1( v2:MVL7_VECTOR;dist: integer) return MVL7 VECTOR is
      variable v1: MVL7_VECTOR (v2'HIGH downto v2'LOW);
      variable I: integer;
   begin
      v1 := v2;
      for I in 1 to dist loop
         v1 := SHL(v1, '1');
      end loop;
      return v1;
   end;
--*************************************************************
   -- 右移函数，最后一位由参数fill决定
   function SHR(v2:MVL7_VECTOR;fill:MVL7 ) return MVL7_VECTOR is
      variable v1: MVL7_VECTOR (v2'HIGH downto v2'LOW);
      variable shift_val: MVL7_VECTOR (v1'HIGH downto v1'LOW);
   begin
      v1 := v2;
      for I in v1'LOW to (v1'HIGH - 1) loop
         shift_val(I) := v1(I + 1);
      end loop;
      shift_val(v1'HIGH) := fill;
      return shift_val;
   end;
--*************************************************************
   -- 右移函数，但最后一位用'0'填充
function SHR0( v2 : MVL7_VECTOR ; dist : integer) return MVL7_VECTOR is
      variable v1: MVL7_VECTOR (v2'HIGH downto v2'LOW);
      variable I: integer;
   begin
      v1 := v2;
      for I in 1 to dist loop
         v1 := SHR(v1, '0');
      end loop;
      return v1;
   end;
--*************************************************************
   -- 右移函数，但最后一位用'1'填充
function SHR1( v2 : MVL7_VECTOR ; dist : integer) return MVL7_VECTOR is
      variable v1: MVL7_VECTOR (v2'HIGH downto v2'LOW);
      variable I: integer;
```

```
  begin
    v1 := v2;
    for I in 1 to dist loop
      v1 := SHR(v1, '1');
    end loop;
    return v1;
  end;
--**********************************************************
  -- 循环右移函数
function ROTR( v2 :  MVL7_VECTOR ;  dist :  integer) return MVL7_VECTOR is
    variable v1: MVL7_VECTOR (v2'HIGH downto v2'LOW);
    variable I: integer;
  begin
    v1 := v2;
    for i in 1 to dist loop
      v1 := SHR(v1,v1(v1'LOW));
    end loop;
    return v1;
  end;
--**********************************************************
  -- 循环左移函数
function ROTL( v2 :  MVL7 VECTOR ;  dist :  integer) return MVL7_VECTOR is
    variable v1: MVL7_VECTOR (v2'HIGH downto v2'LOW);
    variable I: integer;
  begin
    v1 := v2;
    for i in 1 to dist loop
      v1 := SHL(v1,v1(v1'HIGH));
    end loop;
    return v1;
  end;
--**********************************************************
  -- 将一个整数转换为一个七值逻辑位向量，第 12 例中有详细的描述
  function I2B( Number :  integer;  len :  integer) return MVL7_VECTOR is
    variable temp: MVL7_VECTOR (len - 1 downto 0);
    variable NUM: integer:=0;
    variable QUOTIENT: integer:=0;
  begin
    QUOTIENT := Number;
    for I in 0 to len - 1 loop
      NUM := 0;
      while QUOTIENT > 1 loop
        QUOTIENT := QUOTIENT - 2;
        NUM := NUM + 1;
      end loop;
      case QUOTIENT is
```

```
            when 1 =>
               temp(I) := '1';
            when 0 =>
               temp(I) := '0';
            when others =>
               null;
         end case;
         QUOTIENT := NUM;
      end loop;
      return temp;
   end;
--***************************************************************
   -- 将一个七值逻辑位向量转换为一个整数值
   function B2I( v2 :  MVL7_VECTOR ) return integer is
      variable v1: MVL7_VECTOR (v2'HIGH downto v2'LOW);
      variable SUM: integer:=0;
   begin
      v1 := v2;
      for N in v1'LOW to v1'HIGH loop
         if v1(N) = '1'  then
            SUM := SUM + (2 ** (N - v1'LOW));
         end if;
      end loop ;
      return SUM;
   end;
--*************************************************************
   -- 取反函数，将参数中的位向量的每一位取反
   function COMP( v2 :  MVL7_VECTOR ) return MVL7_VECTOR is
      variable v1: MVL7_VECTOR (v2'HIGH downto v2'LOW);
      variable temp: MVL7_VECTOR (v1'HIGH downto v1'LOW);
      variable I: integer;
   begin
      v1 := v2;
      for I in v1'LOW to v1'HIGH loop
         if v1(I) = '0'  then
            temp(i) := '1';
         else
            temp(i) := '0';
         end if;
      end loop;
      return temp;
   end;
--***************************************************************
   -- 取补码函数，取反加 1
   function TWOs_COMP( v2 :  MVL7_VECTOR ) return MVL7_VECTOR is
      variable v1: MVL7_VECTOR (v2'HIGH downto v2'LOW);
```

```
      variable temp: MVL7_VECTOR (v1' HIGH downto v1' LOW);
   begin
      v1 := v2;
      temp := comp(v1);
      temp := INC(temp);
      return temp;
   end;
--************************************************************
   -- 重载函数，两个七值逻辑位向量相减
function "-" ( x1 : MVL7_VECTOR ; x2 : MVL7_VECTOR ) return MVL7_VECTOR is
      variable v1: MVL7_VECTOR (x1' HIGH - x1' LOW downto 0);
      variable v2: MVL7_VECTOR (x2' HIGH - x2' LOW downto 0);
      variable SUM: MVL7_VECTOR (v1' HIGH downto v1' LOW);
   begin
      v1 := x1;
      v2 := x2;
      assert v1' LENGTH = v2' LENGTH
      report "MVL7 vector -: operands of unequal LENGTHs"
      severity failure;
      SUM := I2B(B2I(v1) - B2I(v2),SUM' LENGTH);
      return (SUM);
   end;
--************************************************************
   -- 两个七值逻辑相减
   function DEC( x :  MVL7_VECTOR ) return MVL7_VECTOR is
      variable v: MVL7_VECTOR (x' HIGH downto x' LOW);
   begin
      v := x;
      return I2B(B2I(v) - 1,v' LENGTH);
   end;
--************************************************************
   -- 带进位的加法函数，两个七值逻辑位向量相加
function CARRY_ADD(x1: MVL7_VECTOR ; x2: MVL7_VECTOR )return MVL7_VECTOR is
      variable v1: MVL7_VECTOR (x1' HIGH - x1' LOW downto 0);
      variable v2: MVL7_VECTOR (x2' HIGH - x2' LOW downto 0);
      variable SUM: MVL7_VECTOR (x1' HIGH - x1' LOW + 1 downto 0);
      -- + 1 is for carry
   begin
      v1 := x1;
      v2 := x2;
      assert v1'LENGTH = v2' LENGTH
         report "MVL7vector carry add:operands of unequal LENGTHs"
         severity FAILURE;
      SUM := I2B(B2I(v1) + B2I(v2),SUM'LENGTH);
      return (SUM);
   end;
```

```
--****************************************************************
  -- 重载函数，两个七值逻辑位向量相加
 function "+" (x1 : MVL7_VECTOR ; x2 : MVL7_VECTOR ) return MVL7_VECTOR is
     variable v1: MVL7_VECTOR (x1' HIGH - x1' LOW downto 0);
     variable v2: MVL7_VECTOR (x2' HIGH - x2' LOW downto 0);
     variable SUM: MVL7_VECTOR (v1' HIGH downto v1' LOW);
  begin
     v1 := x1;
     v2 := x2;
     assert v1' LENGTH = v2' LENGTH
     report "MVL7 vector +: operands of unequal LENGTHs"
     severity failure;
     SUM := I2B(B2I(v1) + B2I(v2),SUM' LENGTH);
     return (SUM);
  end;
--****************************************************************
  -- 增 1 函数
  function INC( x :  MVL7 VECTOR ) return MVL7 VECTOR is
     variable v: MVL7_VECTOR (x' HIGH downto x' LOW);
  begin
     v := x;
     return I2B(B2I(v) + 1,v' LENGTH);
  end;
--****************************************************************
  -- 奇校验函数
  function ODD_PARITY( v1 :  MVL7_VECTOR ) return MVL7 is
  begin
     if ((SUM(v1) mod 2) = 1) then
        return ' 0' ;
     else
        return ' 1' ;
     end if;
  end;
--****************************************************************
  -- 偶校验函数
  function EVEN_PARITY( v1 :  MVL7_VECTOR ) return MVL7 is
  begin
     if ((SUM(v1) mod 2) = 1) then
        return ' 1' ;
     else
        return ' 0' ;
     end if;
  end;
--****************************************************************
  -- 将七值逻辑位向量反向，如将"1x001"变为"100x1"
  function REVERSE( v2 :  MVL7_VECTOR ) return MVL7_VECTOR is
```

```
      variable v1: MVL7_VECTOR (v2' HIGH downto v2' LOW);
      variable temp: MVL7_VECTOR (v1' HIGH downto v1' LOW);
   begin
      v1 := v2;
      for I in v1' HIGH downto v1' LOW loop
         temp(I) := v1(v1' HIGH - I + v1' LOW);
      end loop ;
      return temp;
   end;
--*************************************************************
   -- 计算七值逻辑位向量中' 1' 的数目
   function SUM( v2 :  MVL7_VECTOR ) return integer is
      variable v1: MVL7_VECTOR (v2' HIGH downto v2' LOW);
      variable count: integer:=0;
   begin
      v1 := v2;
      for I in v1' HIGH downto v1' LOW loop
         if (v1(I) = ' 1' ) then
            count := count + 1;
         end if;
      end loop ;
      return count;
   end;
--*************************************************************
   -- 装填函数
   function PAD(v:MVL7_VECTOR; width : integer) return MVL7_VECTOR is
   begin
      return I2B(B2I(v),width);
   end;
--*************************************************************
   -- 重载函数，两个七值逻辑位向量相乘
  function "*" (x1 : MVL7_VECTOR ; x2 : MVL7_VECTOR ) return MVL7_VECTOR is
      variable v1: MVL7_VECTOR (x1' HIGH - x1' LOW downto 0);
      variable v2: MVL7_VECTOR (x2' HIGH - x2' LOW downto 0);
      variable PROD: MVL7_VECTOR (v1' HIGH downto v1' LOW);
   begin
      v1 := x1;
      v2 := x2;
      assert v1' LENGTH = v2' LENGTH
      report "MVL7 vector MUL: operands of unequal LENGTHs"
      severity failure;
      PROD := I2B(B2I(v1) * B2I(v2),PROD' LENGTH);
      return (PROD);
   end;
--*************************************************************
-- 线或函数
```

```
function WiredOr (V: MVL7_VECTOR) return MVL7 is
  variable result: MVL7;
begin
  result := ' Z' ;
  for i in V' RANGE loop
    result := tbl_WIREDOR(result, V(i));
    exit when result = ' 1' ;
  end loop;
  return result;
end WiredOr;
--************************************************************
end;
```

（源描述文件名：14_MVL7_function.vhd）

第 15 例　四输入多路器

陈东瑛

1. 电路系统工作原理

本例系统功能为从四路中选择一路的选择器，如图 15.1 所示。当控制端 OEbar 信号有效（低电平）时，把信号有效的（高电平）选择控制端（R_sel，D_sel，uPC_sel 和 stack_sel）所对应的数据输入端的相应数据（D，RE，uPC 和 reg_file(sp)）送到端口 Y，其中 4 个选择控制端口上最多同时有一个信号有效。而数据输入端 sp 的信号不是数据本身，而是指向 reg_file 堆栈单元的指针。

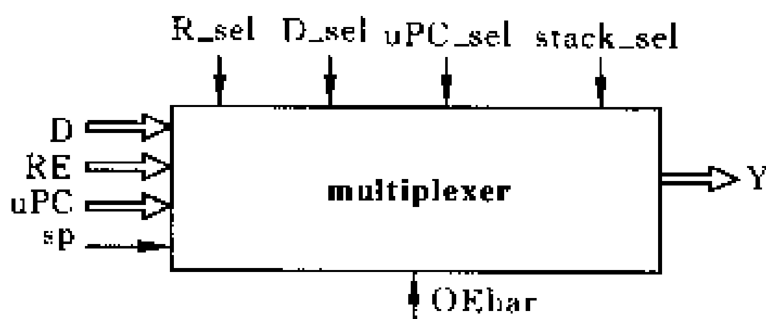


图 15.1　四输入多路选择器

除了 sp 端输入信号应为整数之外，其他输入端口接收的信号均为七值逻辑，详见第 9 例。

2. VHDL 语言描述方法及语法分析

本例的系统源描述文件名为 15_mux41.vhd，其中实体说明为 multiplexer，对应一个同名结构体 multiplexer，采取行为描述风格。实体说明 multiplexer 的各端口与系统功能要求的接口（如图 15.1 所示）相一致，结构体 multiplexer 的所有工作组合成一个块（muxr）。

```
--四输入多路器系统源描述文件 15_mux41.vhd
use work.types.all;
use work.MVL7_functions.all;
use work.synthesis_types.all;

entity multiplexer is
port (
```

```
        R_sel :       in MVL7;
        D_sel :       in MVL7;
        uPC_sel :     in MVL7;
        stack_sel :   in MVL7;
        OEbar :       in MVL7;
        sp :          in integer range 0 to 5;
        D :           in MVL7_VECTOR(11 downto 0);
        RE :          in MVL7_VECTOR(11 downto 0);
        uPC :         in MVL7_VECTOR(11 downto 0);
        Y :           out MVL7_VECTOR(11 downto 0)
        );
end multiplexer;
architecture multiplexer of multiplexer is
begin
muxr : block
signal Y_temp : MVL7_VECTOR(11 downto 0);
signal reg_file : MEMORY_12_BIT(5 downto 0) := (
                                              ("000000000000"),
                                              ("111111111111"),
                                              ("000000000000"),
                                              ("111111111111"),
                                              ("000000000000"),
                                              ("111111111111")
                                              );

begin
Y_temp <= RE              when R_sel = '1' --根据输入控制信息选择输入数据信息
     else D               when     D_sel = '1'
     else uPC             when     uPC_sel = '1'
     else reg_file(sp)    when     stack_sel = '1'
     else "000000000000";

Y    <= Y_temp when OEbar = '0'             --"输入允许"控制
     else "ZZZZZZZZZZZZ";
end block muxr;
end multiplexer;
```

虽然输入信号 X_sel（数据选择）与 OEbar 对输出的控制各由一条并发条件赋值语句实现，即两语句执行顺序是随机的，但是 VHDL 语言的 delta 延迟的语法现象可以保证同时刻一种输入只有一种输出。例如，在 3ns 时刻，系统输入 OEbar = '0'，R_sel = '1'，RE = "111111111111"，原有 Y_temp = "000000000000"，Y = Y_temp，系统状态

变化分为以下几步，如表 15.1 所示。

表 15.1　系统状态变化

时间	Y_temp	Y
3 ns	000000000000	000000000000
3 ns + 1*delta	111111111111	000000000000
3 ns + 2*delta	111111111111	111111111111

系统处理信号七值逻辑类型 MVL7 由另一个文件 15_types.vhd 中的包 TYPES 描述。该包中还定义了若干以 MVL7 为基础的复合类型及特殊常量，通过文件 15_mux41.vhd 中的 **use** work.types.**all** 语句，实体说明 multiplexer 与结构体 multiplexer 可以使用该包中所有的内容。另外两条 **use** 子句，使系统描述中还可以使用某些七值逻辑的操作函数(MVL7_functions.vhd 文件中的包定义)和以 MVL7 为基础的数组(synthesis_types.vhd 文件中的包定义)。本例涉及的七值逻辑类型 MVL7 的意义和作用与第 7 例移位寄存器所涉及的完全一致，在第 9 例中已有专门说明。本例与第 7 例所引用的若干 MVL7 程序包分别在各自库内的不同文件中描述，数据重复，共享性差。VHDL 语言电路设计提倡的方法是：当系统处理对象的类型比较复杂，或者有一些固有（专用）操作，或者有一些操作需要被其他系统引用时，就把它们描述在特定的包中，甚至放在其他文件的包中，再通过 **use** 子句引用时，可以大大提高 VHDL 描述的可读性、共享性，并易于修改维护。如第 78 例到第 87 例的 Am2901 系列的电路涉及的 MVL7 程序包都在 L2901 库中描述，为多个不同的设计库所共享。

```
--相关类型定义文件 15_types.vhd
package TYPES is
--定义七值逻辑数据类型
type MVL7 is ('X',    --强未知
              '0',    --强低电平
              '1',    --强高电平
              'Z',    --高阻
              'W',    --弱未知
              'L',    --弱低电平
              'H');   --弱高电平
... ...
--定义七值逻辑数组类型
type MVL7_VECTOR is array (natural range < >) of MVL7;   --定义非限定型数组
type MVL7_TAB1D is array (MVL7) of MVL7;                 --定义一维数组
type MVL7_TABLE is array (MVL7,MVL7) of MVL7;            --定义二维数组
--定义其他相关数据类型及常量
... ...
```

3. 模拟测试向量的选择及模拟结果分析

本例的测试文件为 15_test_vectors_mux41.vhd，包含一个空的实体说明 E，对应结构体 AA，即测试外壳与具体测试方案相分离，使测试内容易于扩展的结构。结构体 AA 说明一个虚拟设计实体——元件 cmux，用以把外部激励信号传给电路系统描述 15_mux41.vhdl 中的实体 multiplexer，由元件例示语句和组装规定实现。

```
--四输入多路器测试台 15_test_vectors_mux41.vhd
use work.types.all;
use work.MVL7_functions.all;
use work.synthesis_types.all;

entity E is
end E;

architecture AA of E is                                    --虚拟设计实体
    component cmux
      port (
            R_sel     :    in MVL7;
            D_sel     :    in MVL7;
            uPC_sel   :    in MVL7;
            stack_sel :    in MVL7;
            OEbar     :    in MVL7;
            sp        :    in integer range 0 to 5;
            D         :    in MVL7_VECTOR(11 downto 0);
            RE        :    in MVL7_VECTOR(11 downto 0);
            uPC       :    in MVL7_VECTOR(11 downto 0);
            Y :            out MVL7_VECTOR(11 downto 0)
             );
    end component;

signal R_sel       :  MVL7;                                    --外部激励信号
signal D_sel       :  MVL7;
signal uPC_sel     :  MVL7;
signal stack_sel   :  MVL7;
signal OEbar       :  MVL7;
signal sp          :  integer range 0 to 5;
signal D           :  MVL7_VECTOR(11 downto 0);
signal RE          :  MVL7_VECTOR(11 downto 0);
signal uPC         :  MVL7_VECTOR(11 downto 0);
signal Y           :  MVL7_VECTOR(11 downto 0);
```

```
for all : cmux use entity work.multiplexer(multiplexer);  --组装规定

begin
CMUX1 : cmux port map(                              --元件例示语句
                     R_sel, D_sel, uPC_sel, stack_sel, OEbar,
                     sp, D, RE, uPC, Y  );

process
begin
    R_sel       <= ' 1' ;                                   --测试向量 0
    D_sel       <= ' 0' ;
    uPC_sel     <= ' 0' ;
    stack_sel   <= ' 0' ;
    OEbar       <= ' 0' ;
    sp          <= 2;
    RE          <= "000000000000";
    D           <= "111111111111";
    uPC         <= "111111111111";
    wait for 1 ns;
    assert (Y = "000000000000")                              --自判断
    report "Assert 0 : < Y /= 000000000000 >"
    severity warning;
    wait for 1 ns;
    ... ...

    R_sel       <= ' 0' ;                                   --测试向量 9
    D_sel       <= ' 1' ;
    uPC_sel     <= ' 0' ;
    stack_sel   <= ' 0' ;
    OEbar       <= ' 1' ;
    sp          <= 1;
    RE          <= "000000000000";
    D           <= "111111111111";
    uPC         <= "000000000000";
    wait for 1 ns;
    assert (Y = "ZZZZZZZZZZZZ")                              --自判断
    report "Assert 9 : < Y /= ZZZZZZZZZZZZ >"
    severity warning;
    wait for 1 ns;
    assert false                                           --结束整个模拟
    report "---End of Simulation---"
    severity error;
```

```
end process;
end AA;
```

AA 中的进程提供 10 个有自判断功能的测试向量，以验证系统描述的功能。每个测试向量均有一段信号保持时间，使系统的状态变化在时间轴（使用 Vsim/Talent 的波形显示器观察）上可见。

本测试文件把对系统描述的输出的期望值作为断言语句的判断条件，以判断系统描述的实际输出的正误，这是一种常用的自判断方法。前 10 个断言语句的严谨性级别均为 warning，不结束模拟。最后由条件永假的严谨性级别为 error 的断言语句结束整个模拟。

（源描述文件名：15_mux41.vhd
15_types.vhd
测试平台文件名：15_test_vectors_mux41.vhd）

第16例 目标选择器

吴清平

1. 电路系统工作原理

本设计单元是一个目标选择器（即多路选择器），其功能为：

- 决定是加载输入信号 A 还是 F 到输出端 Y;
- 决定是否加载输入信号 F(0) 到输出端 RAM0;
- 决定是否加载输入信号 F(1) 到输出端 RAM3;
- 决定是否加载输入信号 Q(3) 到输出端 Q3;
- 决定是否加载输入信号 Q(1) 到输出端 Q0;

其示意图如图 16.1 所示。

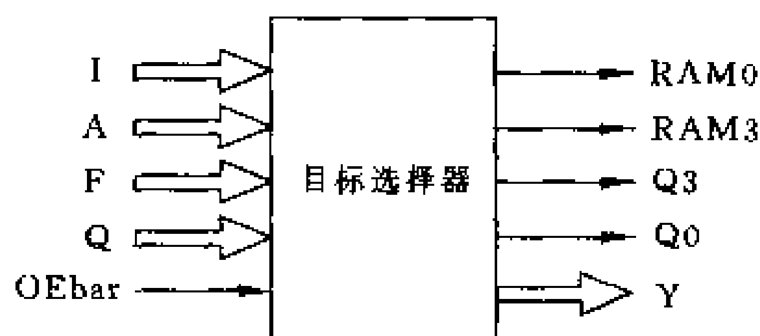


图 16.1 目标选择器示意图

2. VHDL 语言描述方法及语法分析

(1) 实体分析

实体 output_and_shifter 定义了如下端口。

端口名	端口类型	作用	数据类型
I	输入	控制信号	MVL7_VECTOR(8 **downto** 0)
A	输入	数据输入信号	MVL7_VECTOR(3 **downto** 0)
F	输入	数据输入信号	MVL7_VECTOR(3 **downto** 0)
Q	输入	数据输入信号	MVL7_VECTOR(3 **downto** 0)
OEbar	输入	使能信号	MVL7
RAM0	输出	数据输出信号	MVL7
RAM3	输出	数据输出信号	MVL7
Q0	输出	数据输出信号	MVL7
Q3	输出	数据输出信号	MVL7

其中类型 MVL7 是七值逻辑，其定义已在前面的举例中说明。MVL7_VECTOR 是七值逻辑向量，它的值是一个数组，定义如下：

```
type MVL7_VECTOR is array (Natural range < >) of MVL7;
```

此类型定义类型 MVL7_VECTOR 为类型 MVL7 的非限定数组，在使用时必须指明它的边界。它们均在 16 types.vhd 文件中的 types 包中定义。

(2) 结构体分析

结构体 output_and_shifter 是实体 output_and_shifter 的实现，在结构体中有 5 条并发信号赋值语句。

VHDL 中并发信号赋值语句共有两种：条件信号赋值语句和选择信号赋值语句。

① 条件信号赋值语句

该语句满足以下约定：

- 当条件为真时，将波形赋给信号
- 条件表达式的结果应为布尔值
- 条件信号赋值语句中允许包含多个形如“波形 **when** 条件”的条件赋值子句
- 以波形结束（即没有以 **when** 为标志的条件）

条件信号赋值语句的格式为：

```
[ 标号：][postponed]
目标信号 <= [guard][transport][inertial] {波形 when 条件 else}
              波形；
```

② 选择信号赋值语句

该语句当条件为真时，将波形赋给信号。其格式如下：

```
[标号：][postponed]with 表达式 select
目标信号 <= [guard][transport][inertial]{波形 when 选择条件，}
              波形 when 选择条件；
```

本例中用到的 5 条并发信号赋值语句均属于条件信号赋值语句。描述如下：

```
Y <= A when (( I(8 downto 6) = "010") and ( OEbar = ' 0' )) else
     F when (not(( I(8 downto 6) = "010")) and ( OEbar = ' 0' )) else "ZZZZ";
RAM0 <= F(0) when ( I(8) = ' 1' ) and ( I(7) = ' 0'  ) else ' Z' ;
RAM3 <= F(3) when ( I(8) = ' 1' ) and ( I(7) = ' 1'  ) else ' Z' ;
Q3    <= Q(3) when ( I(8) = ' 1' ) and ( I(7) = ' 1' ) else ' Z' ;
Q0    <= Q(0) when ( I(8) = ' 1' ) and ( I(7) = ' 0' ) else ' Z' ;
```

第 1 条语句指明：当 OEbar 为 0 且 I(8 downto 6) = “010”时，将 A 的值赋给 Y。

当 OEbar 为 0 且 I(8 downto 6) = "010"时，将 F 的值赋给 Y。否则 Y 的值等于 "ZZZZ"，即高阻状态。

第 2 和第 5 条语句指明：当 I(8) = ' 1' 且 I(7)= ' 0' 时，分别将 F(0)赋给 RAM0、Q(0)赋给 Q0，否则将使 RAM0 和 Q0 为' Z' ，即高阻状态。

第 3 和第 4 条语句指明：当 I(8) = ' 1' 且 I(7) = ' 1' 时，分别将 F(3)赋给 RAM3、Q(3)赋给 Q3，否则将使 RAM3 和 Q3 为' Z' ，即高阻状态。

在结构体中的一条并发信号赋值语句相当于一条进程语句，也就是说任何一条信号赋值语句都有一个与之等效的进程。比如，选择信号赋值语句，我们就可用一个包含一条 **case** 顺序语句的进程来替代。同样条件信号赋值语句也可由其等价的进程来替代，如上面的第 2 条语句：

```
RAM0 <= F(0) when ( I(8) = ' 1' ) and ( I(7) = ' 0'  ) else ' Z' ;
```

可以用以下进程来替代：

```
process( I(8),I(7) )
     begin
          if( I(8) = ' 1'  and I(7) = ' 0'  )
               RAM0 <= F(0);
          else
               RAM0 <= ' Z' ;
          end if;
     end process;
```

一条并发信号赋值语句的激活条件为其波形元素中的信号的值发生变化，如上句并发信号赋值语句：

```
RAM0 <= F(0) when ( I(8) = ' 1' ) and ( I(7) = ' 0'  ) else ' Z' ;
```

当 I(8)或 I(7)的值发生变化时，此条语句将被激活，将对信号 RAM0 进行赋值。因此与之等效的进程语句可加一个显式的敏感信号表，并将此进程所敏感的信号列在关键字 **process** 之后。此进程的激活条件与上一条并发信号赋值语句相同。而在进程中使用一条 **if** 语句完成对信号 RAM0 的赋值。可以看出，进程语句的写法比较清楚，容易理解，而并发信号赋值语句则显得十分简洁。设计者可根据实际情况灵活使用。

(3) 测试平台分析

测试平台是用来对设计单元施加激励，并检查设计中单元的工作结果以确定设计单元的正确性的。因此它不需要端口，只在结构体中定义一个元件，加以例示语句，并将此元件组装到要测试的设计单元的实体和结构体，然后加激励向量并检查设计单元的输

出以确定其正确性完成测试。

在写测试台时，首先应使用例示语句和组装语句在测试平台中加入要测试的设计单元，然后在一个进程中对与所测试的设计单元的输入端口相连接的信号赋值（加激励向量），再通过观察与被测设计单元相连接的信号的值来判定设计单元的工作结果是否正确。通过完整的测试向量集可确定被测设计单元的正确性。

在测试码中经常使用 **assert** 语句（断言语句）判断结果的正确性。**assert** 语句为写源描述时检查结果提供了一种机制。**assert** 语句的格式如下：

assert　条件表达式

report　提示信息

severity 错误级别

其中的错误级别有 4 种：说明(note)、警告(warning)、错误(error)及故障(failure)。

在模拟过程中，当模拟此句时，模拟器将判断 **assert** 语句的条件表达式此时的值，当其值为 false 时，模拟器会根据错误级别分别采取以下动作：

当错误级别为 note 时，无错误，只将 **assert** 语句中的提示信息提示给用户，模拟不中断，继续模拟。

- 当错误级别为 warning 时，表示警告，将 **assert** 语句中的提示信息提示给用户，模拟不中断，继续模拟。
- 当错误级别为 error 时，有错误，将 **assert** 语句中的提示信息提示给用户，模拟停止，但可以强制继续模拟。
- 当错误级别为 failure 时，有严重错误，将 **assert** 语句中的提示信息提示给用户，模拟停止，且模拟不可再继续。

本例中使用了许多 **assert** 语句来判断输出的正确性。如在对输入端口信号加上激励之后，使用如下 **assert** 语句则表明，在这种激励下，信号 RAM0 的值应该是'Z'（RAM0 与被测元件的某一输出段连接）。

```
assert (RAM0 = 'Z')
report  "assert b1 : < RAM0 /= 'Z' >"
severity  warning;
```

这种方法可以用来代替观察最终模拟结果中大量波形图的方法，以省去观察复杂波形之苦，是书写测试平台的一种好方法。

（源描述文件名：16_multiple_mux.vhd
16_types.vhd
16_MVL7_functions.vhd
测试平台文件名：16_test_vectors.vhd）

第17例 奇偶校验器

陈东瑛

1. 电路系统工作原理

本例的系统功能是对八位二进制数据及其奇偶校验位的输入进行校验，输出正确的奇、偶校验位，如图 17.1。其中另有 3 个输入端，当 IN_READY 端口输入信号有效时，表示输入数据已经准备好，可以处理；当 OUT_REQ 端口输入信号有效时，表示要求输出数据；CLK 端口用于接收时钟信号，支持系统的时钟上升沿同步。而当输出端口 OUT_READY 输出信号有效时，表示输出数据已经准备好，可以为下级电路使用。上述控制端口均为高电平有效。

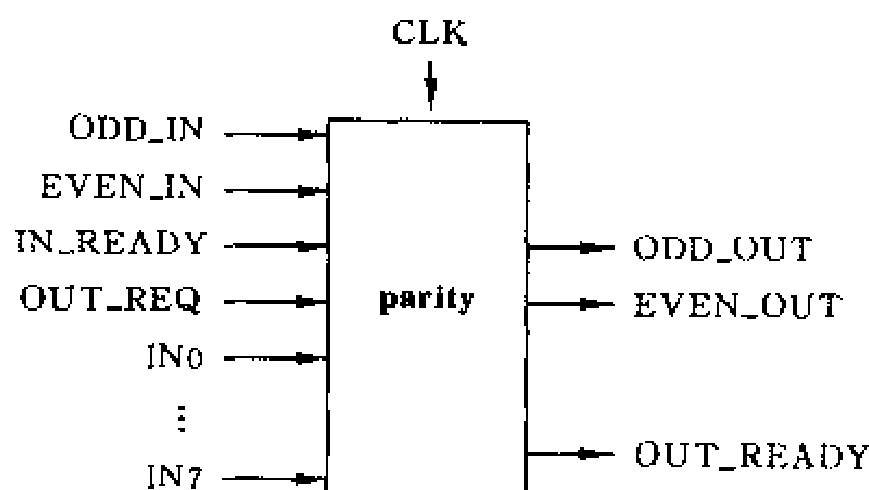


图 17.1 奇偶校验器示意图

2. 电路的 VHDL 语言描述方法及语法分析

本例的系统源描述文件名为 17_parity.vhd，其中实体说明为 parity，相应的结构体为 algorithm，在行为级描述电路的功能。实体说明 parity 的端口对应于电路系统功能要求的接口（如图 17.1 所示）。系统行为在一个进程内描述，针对奇偶校验器的工作特性，以 **if** 语句为主进行描述。对该进程的分析如下。

```
--奇偶校验器系统描述文件 17_parity.vhd
package types is
   subtype short is integer range 0 to 255;
end types;
use work.types.all;
```

```
entity parity is
    port(IN0        : in bit;
         IN1        : in bit;
         IN2        : in bit;
         IN3        : in bit;
         IN4        : in bit;
         IN5        : in bit;
         IN6        : in bit;
         IN7        : in bit;
         EVEN_IN    : in bit;
         ODD_IN     : in bit;
         IN_READY   : in bit;
         OUT_REQ    : in bit;
         CL         : in bit;
         OUT_READY  : out bit;
         ODD_OUT    : out bit;
         EVEN_OUT   : out bit);
end parity;

architecture algorithm of parity is
begin
    process
        variable cond:  boolean := true;                --变量初始化
        variable count: short;
    begin
--时钟上升沿同步及 IN_READY 控制
wait until CLK' EVENT and CLK= ' 1'  and IN_READY = ' 1' ;
if EVEN_IN = ODD_IN then                                --奇偶校验位预判断
   cond := false;
   end if;

   count := 0;

   if IN0 = ' 1'  then
      count := count + 1;
   end if;
   if IN1 = ' 1'  then
      count := count + 1;
   end if;
   if IN2 = ' 1'  then
      count := count + 1;
   end if;
```

```
if IN3 = '1' then
    count := count + 1;
end if;
if IN4 = '1' then
    count := count + 1;
end if;
if IN5 = '1' then
    count := count + 1;
end if;
if IN6 = '1' then
    count := count + 1;
end if;
if IN7 = '1' then
    count := count + 1;
end if;

--判断输入数据中'1'的个数的奇偶
  L1: while count > 1 loop
        count := count - 2;
  end loop L1;

--判断输入奇偶校验位的正确性
  if count = 1 and odd_in = '0' then
       cond := false;
  end if;
  if count = 0 and even_in = '0' then
       cond := false;
  end if;

--输出奇偶校验位
  if count = 1 then
       EVEN_OUT <= '0';
       ODD_OUT  <= '1';
  else
       EVEN_OUT <= '1';
       ODD_OUT  <= '0';
  end if;

--同步控制，设置输出控制信息
  wait until CLK'EVENT and CLK = '1' and OUT_REQ = '1';
  OUT_READY <= '1';
  wait until CLK'EVENT and CLK = '1' and OUT_REQ = '0';
```

```
    OUT_READY <= ' 0' ;

--提示出错信息
    assert cond
    report "odd_in even_in error"
    severity warning;
    end process;

end algorithm;
```

进程说明的变量 cond 用于标识奇、偶校验位输入的正确性，变量 count 用于对八位二进制数据中 '1' 的个数计数。

本例的算法描述了系统的功能，是设计实体 parity 的固有行为，它没有考虑输入信号的各种状态是如何产生的，即没有考虑环境激励。

3. 模拟测试向量的选择及模拟结果分析

前面提到，本例对系统的描述未涉及外部激励的设计，另有与系统描述文件分离的测试文件。这样可以把系统的固有功能与运行实况分离开，使前者更稳定，使后者更灵活。而模拟是通过运行实况来检查系统描述的正确性，所以必须有专门设计的外部激励描述，即激励产生器，其作用如图 17.2 所示，其中的在测模块即系统固有功能的 VHDL 描述。以人工或自动方式对响应进行判断，可以验证系统描述的正确性。

图 17.2　模拟运行情况示意图

本例的激励产生器的实体说明 test_parity 与结构体 bench，分别在 17_test_parity.vhd 和 17_test_bench.vhd 文件中描述，对它们进行模拟，与对一个测试文件（包含实体说明和结构体）模拟的效果相同。

```
--测试台实体说明 17_test_parity.vhd
library  std;
library  work;
library  parity;
use  std.standard.all;
use  parity.types.all;
use  work.all;

entity test_parity is                                  --空实体说明
```

```
end test_parity;
--测试台的结构体 17_test_bench.vhd
architecture bench of test_parity is
    component parity                                        --虚拟设计实体
    port(
        IN0             : in bit;
        IN1             : in bit;
        IN2             : in bit;
        IN3             : in bit;
        IN4             : in bit;
        IN5             : in bit;
        IN6             : in bit;
        IN7             : in bit;
        EVEN_IN         : in bit;
        ODD_IN          : in bit;
        IN_READY        : in bit;
        OUT_REQ         : in bit;
        CLK             : in bit;
        OUT_READY       : out bit;
        ODD_OUT         : out bit;
        EVEN_OUT        : out bit );
    end component;
    signal IN0          : bit;                              --外部激励信号
    signal IN1          : bit;
    signal IN2          : bit;
    signal IN3          : bit;
    signal IN4          : bit;
    signal IN5          : bit;
    signal IN6          : bit;
    signal IN7          : bit;
    signal EVEN_IN      : bit;
    signal ODD_IN       : bit;
    signal IN_READY     : bit;
    signal OUT_REQ      : bit;
    signal CLK          : bit;
    signal OUT_READY    : bit;
    signal ODD_OUT      : bit;
    signal EVEN_OUT     : bit;
for all:parity use entity work.parity;                      --组装规定
begin
```

```
parity_I1: parity                                      --元件例示语句
        port map (
        IN0 => IN0,
        IN1 => IN1,
        IN2 => IN2,
        IN3 => IN3,
        IN4 => IN4,
        IN5 => IN5,
        IN6 => IN6,
        IN7 => IN7,
        EVEN_IN      => EVEN_IN,
        ODD_IN       => ODD_IN,
        IN_READY     => IN_READY,
        OUT_REQ      => OUT_REQ,
        CLK          => CLK,
        OUT_READY    => OUT_READY,
        ODD_OUT      => ODD_OUT,
        EVEN_OUT     => EVEN_OUT
       );
    parity_driver: process                       —时钟上升沿同步下产生外部激励
    begin
      wait until  clk = '1' ;
      IN_READY <= '0' ;
      IN7 <= '1' ;
      IN 6 <= '1' ;
      IN 5 <= '0' ;
      IN 4 <= '1' ;
      IN 3 <= '0' ;
      IN 2 <= '1' ;
      IN 1 <= '1' ;
      IN 0 <= '0' ;
      ODD_IN <= '0' ;
      EVEN_IN <= '1' ;
      OUT_REQ <='0' ;
      wait until CLK = '1' ;
      IN_READY <= '1' ;
      wait until CLK = '1' ;
      OUT_REQ <= '1' ;
      wait until CLK = '1' ;
      IN_READY <='0' ;
```

```
      wait until CLK' EVENT and CLK = '1' and OUT_READY ='1' ;
      OUT_REQ <= '0' ;
      wait for 150ns;
      assert false                                    --结束整个模拟
      report "—End of Simulation---"
      severity error;
   end process;
   CLK <= not CLK after 50 ns;                        --时钟振荡
end bench;
```

结构体 bench 中定义一个虚拟设计实体——元件 parity——和一组与其端口对应的信号。这些外部激励信号的值通过元件例示语句（parity_I1）传入元件 parity 的对应端口，再通过组装规定，传入设计实体 parity 的对应端口，再经过设计实体 parity 内部的作用传向输出。该信号的状态变化由进程 parity_driver 产生。

结构体 bench 中的并发信号赋值语句反复执行，产生系统时钟，该语句等价于如下进程：

```
process
begin
      CLK <= not CLK after 50 ns;
      wait on CLK;
end process;
```

由 Vsim/Talent 模拟运行本测试文件，波形观察如图 17.3 所示（为了便于观察，将信号 IN0～IN 7 打包为一个信号束 IN 0_7）。其中的延迟体现了 VHDL 语言的 delta 延迟的语法现象，同时也体现了系统功能中的 IN_READY 和 OUT_REQ 的控制作用。

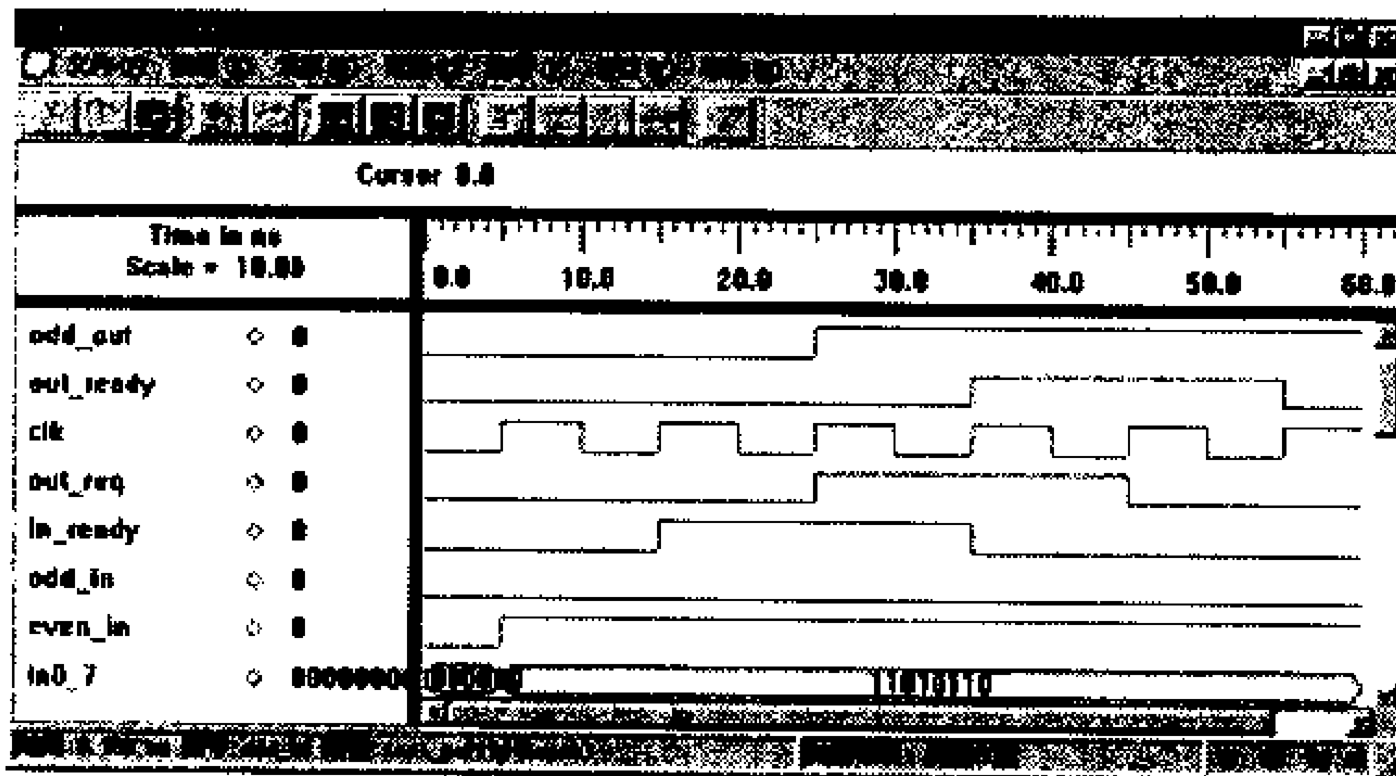


图 17.3 模拟波形

整个模拟由进程 parity_driver 最后的断言语句结束，而 **wait for** 150 ns;语句的作用在于使系统模拟总时间能够包括所有预定事件。

（源描述文件名：17_parity.vhd
测试平台文件名：17_test_parity.vhd
17_test_bench.vhd）

第18例　映射单元库及其使用举例

陈东瑛

1. 电路系统工作原理

本例为提供若干独立功能单元的映射单元库，包括：1，2，4，8，16，32位加法器及乘法器；2，4，8，16，32位比较器（相等比较，大于/小于比较）；2，3，4，5输入的与（非）门、或（非）门、异或（非）门及反向器；还有2，4，8，16，32路选一的多路器和RD、FDRD触发器等，供给具体电路系统使用。

2. 电路的VHDL语言描述方法及语法分析

本例的系统描述文件为18_tech_lib.vhd，每个独立元件由一个独立设计实体描述，其实体说明的端口对应于实际元件的端口，结构体FUNC以基于逻辑表达式的数据流风格进行描述。包types中定义了两个函数bit_to_ int和int_to_bit，实现位型数据与整型数据的相互转换，以简化需要处理多位数据的元件的功能描述。

```
-- 映射单元库系统描述文件 18_tech_lib.vhd
package  types is                                  --基本数据类型及函数定义说明
    subtype nat16 is integer range 0 to 65535;
    subtype nat8  is integer range 0 to 255;
    subtype nat4  is integer range 0 to 15;
    function bit_to_int(bit1: bit_vector) return integer;
    function int_to_bit(in1: integer; len:integer) return bit_vector;
end types;

package body types is                                          --自定义函数
--位型数据到整型数据的转换函数
     function bit_to_int(bit1: bit_vector) return integer is
     alias v1: bit_vector(bit1'LENGTH-1 downto 0) is bit1;
        variable SUM: integer := 0;
        variable i: integer;
     begin
     for i in v1'length - 1 downto 0 loop
          if v1(i) = '1'  then
               SUM := SUM + 2**i;
          end if;
```

```
    end loop;
    return SUM;
    end bit_to_int;

--整型数据到位型数据的转换函数
    function int_to_bit(in1: integer; len:integer) return bit_vector is
    variable i, in2: integer;
    variable digit1: integer:= 2**(len - 1);
    variable result: bit_vector((len-1) downto 0);
    begin
    in2 := in1;
    for i in (len - 1) downto 0 loop
        if in2 >= digit1 then
            result(i) := '1';
            in2 := in2 - digit1;
        else
            result(i) := '0';
        end if;
        digit1 := digit1 / 2;
    end loop;
    return result;
    end int_to_bit;
end types;

-- 映射单元库元件
-- 一位全加器
use work.types.all;
entity ADD1 is
    A  : in  bit;
    B  : in  bit;
    CI : in  bit;
    CO : out bit;
    S  : out bit);
end ADD1;

architecture FUNC of ADD1 is
    signal   X, Y: bit;
begin
    X  <= A xor B;
    Y  <= X and CI;
    S  <= X xor CI;
    CO <= Y or (A and B);
```

```
end FUNC;

--二位全加器
use work.types.all;
entity ADD2 is port(
     A0 : in  bit;
     A1 : in  bit;
     B0 : in  bit;
     B1 : in  bit;
     CI : in  bit;
     CO : out bit;
     S0 : out bit;
     S1 : out bit);
end ADD2;
architecture FUNC of ADD2 is
begin
     process(A0, A1, B0, B1, CI)
          variable A, B: bit_vector(1 downto 0);
          variable S: bit_vector(2 downto 0);
          variable INT_A, INT_B, INT_S: nat4;
     begin
-- 初始化
          A := A1 & A0;
          B := B1 & B0;
-- 把 A, B 转换为整型
          INT_A := bit_to_int(A);
          INT_B := bit_to_int(B);

          INT_S := INT_A + INT_B;

          if CI = '1' then
             INT_S := INT_S + 1;
          end if;
-- 把 INT_S 转换为位型
          S := int_to_bit(INT_S, 3);
-- 输出
          S0 <= S(0);
          S1 <= S(1);
          CO <= S(2);
     end process;
end FUNC;
... ...
```

```
-- FDRD 触发器
use work.types.all;
entity FDRD is
        (
        D  : in  bit;
        RD : in  bit;
        CE : in  bit;
        C  : in  bit;
        Q  : out bit);
end FDRD;
architecture FUNC of FDRD is
begin
   process
   begin
      wait until C'event and C ='1' ;                    —时钟上升沿同步
      if (CE = '1'  and RD = '0' ) then                    --CE/RD 控制
         Q <= D;
      end if;
   end process;
end FUNC;
```

3. 模拟测试向量的选择及模拟结果分析

要测试本例单元库中的任一个单元（设计实体），需要在测试文件中说明与之对应的虚拟设计实体，定义与其端口对应的外部信号，并通过元件例示语句、组装规定（或组装语句）把三者联系起来。再通过对外部信号赋值，加入激励，检测响应，还可以用断言语句实现自判断功能。

对多个单元的测试可以在多个测试实体中分别实现，或者在同一测试实体下的多个结构体中分别实现，或者在同一个结构体中实现；这三种情况相当于在一个测试台上测试一个单元，或者在同一测试台上以不同的方案测试不同的单元，或者用同一测试台上的同一方案测试多个单元。当同一方案（结构体）中检测多个单元时，单元之间可以无关，也可以有联系。对待测单元的一定方式的组合进行总体测试，可以大大减少测试台总的输入和输出信号量，简化人工或自动的判断工作；但也有可能出现不能定位具体出错单元的情况（出错单元的端口不与系统的输入、输出相连）。

测试文件 18_test_lib.vhd 中的空实体说明 E 对应结构体 test_add，其中块 t_add8 以元件例示语句、组装规定和外部信号初始化来实现对设计实体 ADD8 的测试。以 Vsim/Talent 的对象跟踪工具可以观察无保持时间的外部信号的状态变化，以验证 ADD8 的功能。

```
--八位全加器测试台 18_test_lib.vhd
```

```
entity E is
end E;
architecture test_add of E is
begin
t_add8:block
 signal a0 : bit := '1' ;                    --外部信号产生初始激励
 signal a1 : bit:= '0' ;
 signal a2 : bit:= '0' ;
 signal a3 : bit:= '1' ;
 signal a4 : bit:= '0' ;
 signal a5 : bit:= '0' ;
 signal a6 : bit:= '1' ;
 signal a7 : bit:= '1';

 signal b0 : bit:= '1' ;
 signal b1 : bit:= '0' ;
 signal b2 : bit:= '1' ;
 signal b3 : bit:= '1' ;
 signal b4 : bit:= '0' ;
 signal b5 : bit:= '0' ;
 signal b6 : bit:= '0' ;
 signal b7 : bit:= '1' ;
 signal ci : bit:= '1' ;
 signal co : bit;
 signal s0, s1, s2, s3, s4, s5, s6, s7 : bit;

component add8                                  --虚拟设计实体
  port(
     A0 : in  bit;
     A1 : in  bit;
     A2 : in  bit;
     A3 : in  bit;
     A4 : in  bit;
     A5 : in  bit;
     A6 : in  bit;
     A7 : in  bit;
     B0 : in  bit;
     B1 : in  bit;
     B2 : in  bit;
     B3 : in  bit;
     B4 : in  bit;
     B5 : in  bit;
```

```
      B6 : in  bit;
      B7 : in  bit;
      CI : in  bit;
      CO : out bit;
      S0 : out bit;
      S1 : out bit;
      S2 : out bit;
      S3 : out bit;
      S4 : out bit;
      S5 : out bit;
      S6 : out bit;
      S7 : out bit);
  end component;
  for k1 : add8 use entity work.ADD8;                          --组装规定
  begin
  k1: add8 port map(a0, a1, a2, a3, a4, a5, a6, a7,            --元件例示语句
                    b0, b1, b2, b3, b4, b5, b6, b7,
                    ci, co,
                    s0, s1, s2, s3, s4, s5, s6, s7);
  end block t_add8;
  end test_add;
```

（源描述文件名：18_tech_lib.vhd
测试平台文件名：18_test_lib.vhd）

第 19 例　循环边界常数化测试

陈东瑛

1. 电路的 VHDL 语言描述方法及语法分析

本例不涉及具体的电路结构和硬件功能，只是对某些 VHDL 语法现象进行测试，它包含一个测试文件 19_test_194.vhd。

```
--循环边界常数化测试台 19_test_194.vhd
entity test_194 is
end test_194;
architecture behave_1 of test_194 is
begin
    process
        variable Lower : Natural := 5;
        variable Upper  : Natural := 10;
        variable Count: Natural := 0;
    begin
        for I in Lower to Upper loop
            Count := Count + I;
            Lower := Lower + 1;
            Upper := Upper - 1;
        end loop;
        wait for 50ns;
    end process;
end behave_1;
```

该文件由一个实体说明 test_194 和一个结构体 behave_1 组成。test_194 采取实体说明的最简形式，即只命名，无内容。behave_1 由一个进程构成，其主体为一个 **for** 循环，即要说明的语法现象。

本例以实体说明抽象定义一个测试，用结构体具体描述测试内容，这种分离描述的方式与对电路系统或功能器件的描述方式相类似，它们把系统功能抽象为实体说明，而具体行为或实现则由结构体描述。由此保证了 VHDL 语言描述的规范性，也给一个测试准备多种方案提供了方便（一个测试实体对应于多个结构体）。

2. 模拟测试向量的选择及模拟结果分析

以 Vsim/Talent 对本例进行模拟，由于进程只能暂停，所以，运行时要预先设置断点，或者由“中断模拟”按钮强制中断模拟运行，以防止死循环的发生。借助对象跟踪工具，可以观察不同时刻各变量的值。模拟运行前，各变量已经得到初始化的值；以单步方式运行，观察到循环体执行 6 次后结束，同时模拟也结束。count 的值从 5 累加到 10（即 45），Low，Upper 分别加、减 6（即 11，4）。循环边界最初由 Low 和 Upper 设定后，就不再受相应变量的影响，由此验证了 VHDL 语言的循环边界常数化的特征。

（源描述文件名：19_test_194.vhd）

第20例 保护保留字

袁 媛

本例很短小，主要是为了说明一个问题：如果在写源描述时，把 VHDL 的保留字或属性名用作自己的变量、信号或端口、常量名，会有什么后果，应如何解决。

描述如下：

```
entity test_158 is
end test_158;
architecture Behave_1 of test_158 is
     signal ns:natural :=55;
begin
     process
          begin
               ns <= 77;
               wait for 10ns;          (*)
     end process;
end Behave_1;
```

其中，(*)语句不会被系统编译，因为此时 ns 被视为一个信号名而不再是一个物理时间单位。当然，这样的程序也不能模拟。

在 test_159 中，我们把 ns 改换为 nss，并把进程主体改为：

```
nss <= 77 after 100 ns;
wait for 100ns;
nss <= 99 after 200ns;
wait for 200ns;
```

再加上与第 48 例中相同的 Finish 进程，但用 **wait on** 600ns，模拟后的波形如图 20.1 所示。

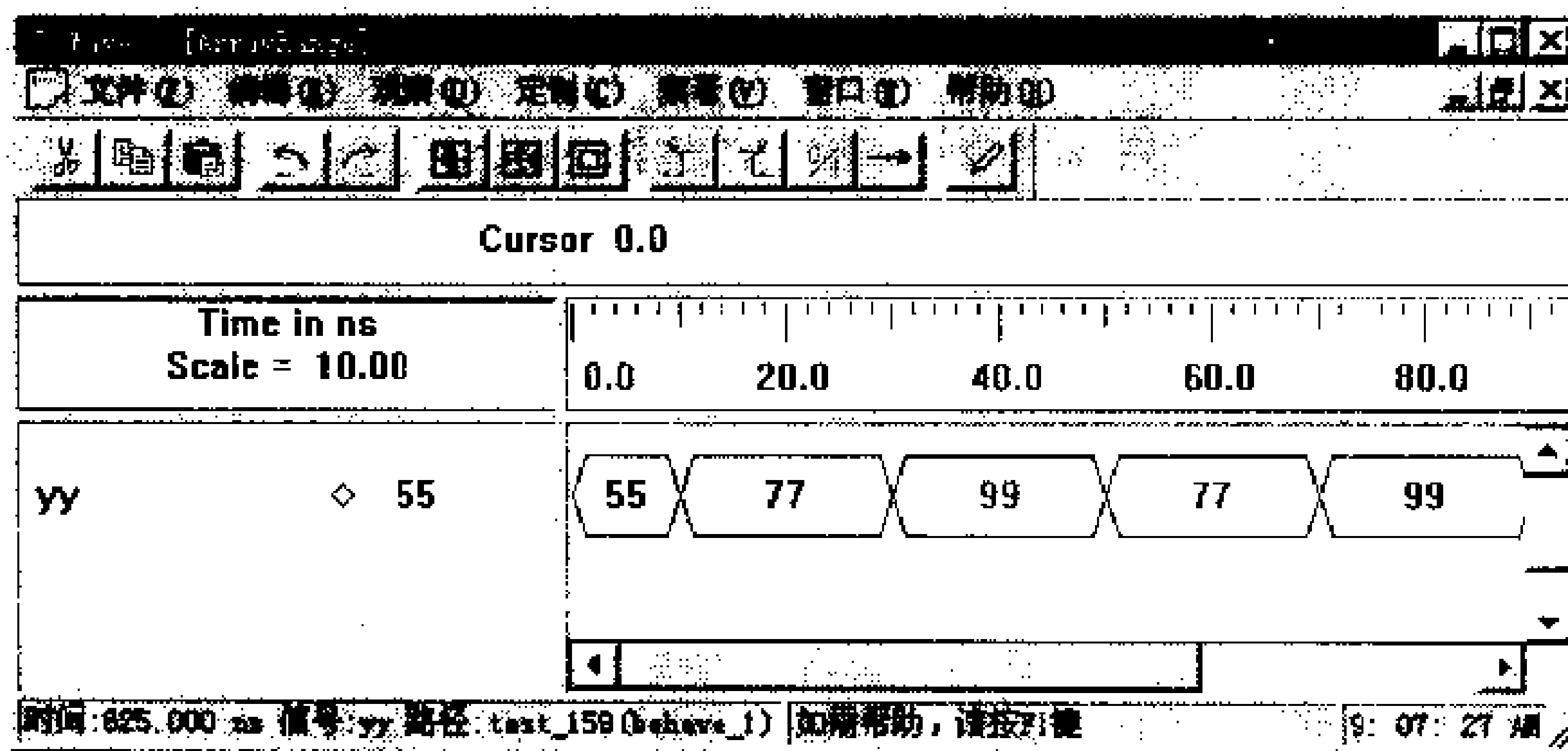


图 20.1　模拟后的波形

20_test_159 同样带有测试向量，有关内容将在第 48 例中详细阐述，在此不再赘述。

（源描述文件名：20_test_159.vhd）

第21例 进程死锁

刘沁楠

1. “死锁”的概念

本例不针对特定的设计电路，旨在说明VHDL中“死锁”的概念。所谓“死锁”，是指各并发进程彼此互相等待调用对方所拥有的资源，且这些并发进程在得到对方的资源之前不会释放自己所拥有的资源，从而造成各进程都想得到资源而又得不到资源，不能继续向前推进的状态。图21.1形象地表示出“死锁”的概念。

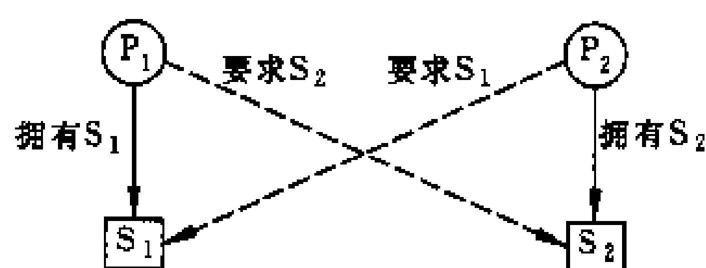


图21.1 “死锁”的概念

下面将结合本例进一步阐述“死锁”的概念。

2. VHDL语法分析及模拟结果

本例由文件21_test_13a.vhd构成，其VHDL源描述如下：

```
—定义测试台(空实体)

entity Test_13a is
end Test_13a;

architecture Behave_1 of Test_13a is
  --信号说明
  signal A : NATURAL := 1;
  signal B : NATURAL := 1;

begin
--进程 Inc_A
  Inc_A:
  process
  begin
```

```
        wait on B; --B 为敏感信号
        A <= A-1;
        assert FALSE
          report "Simulation time has processed : Inc_A"
          severity note;
    end process Inc_A;

  --进程 Inc_B
    Inc_B:
    process
    begin
        wait on A; -- A 为敏感信号
        B <= B+1;
        assert FALSE
          report "Simulation time has processed : Inc_B"
          severity note;
    end process Inc_B;
  end Behave_1;
```

描述中首先定义一个空实体 Test_13a，该实体对应的结构体为 Behave1，其中包含两个进程 Inc_A 和 Inc_B，它们之间是并发执行的。

对进程 Inc_A 而言，语句"**wait on** B;"会暂停进程的执行，直到信号 B 发生变化，进程才会执行信号赋值语句"A <= A + 1;"。同样，在进程 Inc_B 中，语句"**wait on** A;"将进程 Inc_B 挂起，直到信号 A 发生变化为止。本例中，信号 A 仅在进程 Inc_A 中发生变化，而该进程的激活条件为信号 B 发生变化，信号 B 又仅在进程 Inc_B 中发生变化。由此可见，信号 A 的变化依赖于信号 B 的变化，反过来，信号 B 的变化又取决于信号 A 是否改变。这样，在进程 Inc_A 和 Inc_B 中存在着进程循环链，其中每一个进程拥有的资源同时被另一个进程请求，在没有外力驱动的情况下，该组并发进程将停止向前推进，陷入永久等待状态。

借助于 VSim/Talent 对本例进行模拟，模拟结果与上述分析完全一致。程序一经运行就停滞于"**wait on** B;"语句，模拟时钟保持在 0ns，不再向前推进，两个进程陷入了互相等待的死锁状态。

要解决此死锁的一个简单方法是，只需添加一条并发信号赋值语句对信号 A 或 B 赋值即可。

（源描述文件名：21_test_13a.vhd）

第22例 振荡与死锁

袁　媛

本例只是为了说明VHDL语言中由于编写不当而可能导致的振荡与死锁现象，所以只是一个测试示例，并不涉及具体的电路。

1. 源描述

```
entity test_13 is
end test_13
architecture Behave_1 of test_13 is
    signal A:natural:=1;
    signal B:natural:=1;
begin
Inc_A:
process
begin
    A <= A+1;
    wait on B;
    if (Now > 0ns) then
        assert false
        report "Absolute simulation time has processed:Inc_A"
        severity note;
    end if;
end process;
Inc_B:
process
begin
    wait on A;
    if(Now > 0ns) then
        assert false
        report "Absolute simulation time has processed:Inc_B"
        severity note;
    end if;
    B <= B+1;
end process;
Finish:
process
```

```
begin
    wait for 100ns;
    assert false report "End of Simulation"
    severity error;
end process;
end Behave_1;
```

注意，由于本例是一个测试的示例，所以它的实体为空。

2. 振荡与死锁

在进程 Inc_A 中，首先给信号 A 增 1，然后等待信号 B 的变化，以再次激活进程 Inc_A。接下来是一条断言语句，如果条件(Now 大于 0ns)满足，则报告 “Absolute Simulation Time Has Processed:Inc_”，其中 Now 是指当前模拟时间。

在进程 Inc_B 中，语句与 Inc_A 大体相似，但关键的不同点是 Inc_B 先等待信号 A 发生变化，然后再将信号 B 的值增 1。

可以看出，信号 A 增 1 后，激活 Inc_B，Inc_B 中信号 B 增 1，又会再次激活 Inc_A，如此循环往复。但是整个模拟时间只是不断地再增加 delta 时间，Now 始终不会超过 0ns，因而形成无限振荡。

如果 Inc_A 中的两条语句 A <= A+1 与 **wait on** B 换一下顺序的话，则会产生 “死锁” 现象，这在操作系统中是一定要避免的。

（源描述文件名：22_deadlock.vhd）

第23例 振荡电路

刁岚松

1. 电路系统的工作原理

本示例描述的振荡电路如图 23.1 所示。

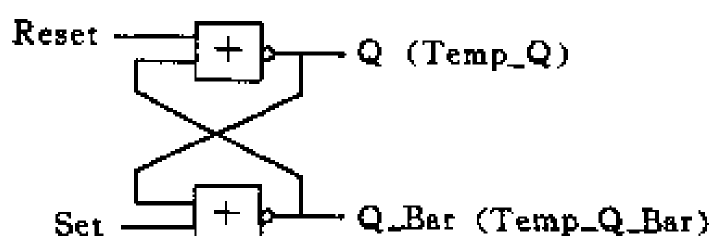


图 23.1 RS 触发电路

从原理上讲，若 Reset 为'0'、Set 为'0'、Q 为'0'、Q_Bar 为'1'，电路处于稳定状态。当 Set 由'0'变为'1'时，在第 1 个 delta 延迟之后 Q_Bar 由'1'变为'0'，接着在第 2 个 delta 延迟之后 Q 由'0'变为'1'，此后，电路处于稳定状态。若 Reset 为'0'、Set 为'0'、Q 为'0'、Q_Bar 为'0'，电路处于振荡状态，本例就是这种情况。其中，尽管实体 RS_Flip_Flop_Bit 的结构体 Behave_1 中 Temp_Q_Bar 被赋值为'1'，但由于输出端 Cout 的缺省值为'0'，所以这个赋值操作没有起作用（实际上，从驱动元件向后继元件传递参数应该通过类属来实现）。在 0 时刻的第 1 个 delta 延迟之后 Q_Bar 由'0'变为 '1'，接着在第 2 个 delta 延迟之后 Q 由'0'变为'1'，第 3 个 delta 延迟之后 Q_Bar 由'1'变为'0'，第 4 个 delta 延迟之后 Q 由'1'变为'0'，这时回到了初始状态，如此无限重复，模拟时间不能前进，始终停留在 0 时刻。

2. 电路的 VHDL 语言描述方法及语法分析

本例首先给出或非门的实体以及结构体描述，再把或非门作为基本元件，通过元件例示语句描述该电路结构。部分描述如下所示：

```
entity RS_Flip_Flop_Bit is
  port (Set : in bit; Reset : in bit; Q : out bit; Q_Bar : out bit);
end RS_Flip_Flop_Bit;
--以上语句描述了电路（RS 触发器）的输入和输出端口
architecture Behave_1 of RS_Flip_Flop_Bit is
  --说明一个或非门
  component Nor_Gate_Bit
```

```
            port (Ain : in bit; Bin : in bit; Cout : out bit);
    end component;
    signal Temp_Q     : bit := ' 0' ;
    signal Temp_Q_Bar : bit := ' 1' ;
begin
--元件例示语句
-- Ain 对应 Reset, Bin 对应 Temp_Q_Bar, Cout 对应 Temp_Q
 Out_Q      : Nor_Gate_Bit
              port map (   Ain => Reset,
                           Bin => Temp_Q_Bar,
                           Cout => Temp_Q);
Out_Q_Bar : Nor_Gate_Bit
              port map (   Ain => Set,
                           Bin => Temp_Q,
                           Cout => Temp_Q_Bar);

 Q      <= Temp_Q;
 Q_Bar <= Temp_Q_Bar;

 Intermediate_Monitor:
 process
   variable Temp_Q_Var     : bit := ' 0' ;
   variable Temp_Q_Bar_Var: bit := ' 0' ;
 begin
   Temp_Q_Var := Temp_Q;
   Temp_Q_Bar_Var := Temp_Q_Bar;
   wait on Temp_Q, Temp_Q_Bar;
 end process Intermediate_Monitor;
end Behave_1;
```

语法分析：

上面的两条元件例示语句描述了图 23.1 所示的电路结构。

尽管 Temp_Q_Bar 被赋值为'1'，但是它将被 Cout 缺省值'0'重新赋值。元件例示语句用来指示一个元件出现于另一个元件中的例示，在例示中仅有子元件的外貌（名称，类型，端口的模式）是可见的，元件例示语句用来标志子元件，并且指定元件的端口和信号之间的一一对应关系。

测试台中用到了组装语句，如下所示：

```
configuration Config_Test_120 of Test_120 is  --组装语句
  for Behave_1                                --测试台结构体名
     for RS_FF_1 : RS_Flip_Flop_Bit            --元件指定
```

```
            use entity work.RS_Flip_Flop_Bit(Behave_1);
            for Behave_1                              --振荡电路结构体名
                for all : Nor_Gate_Bit                 --元件指定
                    use entity work.Nor_Gate_Bit(Behave_1);
                end for;
            end for;
        end for;
    end for;
end Config_Test_120;
```

语法分析：

组装语句允许设计者为每个元件例示选用实体说明及结构体。这里测试台选用了振荡电路的实体说明及相应的结构体，而振荡电路又选用了或非门的实体说明及相应的结构体，所以，这里的组装语句有嵌套现象。

3. 模拟测试向量的选择及模拟结果分析

测试台的部分描述如下：

```
entity Test_120 is

end test_120;
architecture Behave_1 of Test_120 is
  component RS_Flip_Flop_Bit
      port (Set : in bit; Reset : in bit;
            Q : out bit; Q_Bar : out bit);
  end component;

  signal Set    : bit := '0' ;
  signal Reset  : bit := '0' ;
  signal Q      : bit := '0' ;
  signal Q_Bar  : bit := '0' ;
begin
  --触发器的4个端口值均为'0'，电路不稳定，是一个振荡电路。
  RS_FF_1: RS_Flip_Flop_Bit
            port map (Set => Set,Reset => Reset,
                      Q => Q, Q_Bar => Q_Bar);
  RS_Stimulus:
  process
      variable Stimulus_Count : natural := 0;
  begin
    Set <= '1' after 5 ns, '0' after 10 ns;     --激励
```

```
    Stimulus_Count := Stimulus_Count + 1;
    wait for 40 ns;
  end  process RS_Stimulus;
    --监视器进程，用于观察信号值
  RS_Monitor:
  process
    variable Q_Var      : bit := '0' ;
    variable Q_Bar_Var  : bit := '0' ;
  begin
    Q_Var       := Q;
    Q_Bar_Var   := Q_Bar;
--如果 Q 或 Q_Bar 变化，将激活此进程
    wait on Q,Q_Bar;
  end process RS_Monitor;
end Behave_1;
--组装语句
configuration Config_Test_120 of Test_120 is
  for Behave_1                                    --测试台结构体名
     for RS_FF_1 : RS_Flip_Flop_Bit               --元件指定
       use entity work.RS_Flip_Flop_Bit(Behave_1);
       for Behave_1                               --振荡电路结构体名
          for all : Nor_Gate_Bit                  --元件指定
               use entity work.Nor_Gate_Bit(Behave_1);
          end for;
       end for;
     end for;
  end for;
end Config_Test_120;
```

由于本例的模拟时间不能前进，模拟在 0 时刻振荡，因此没有波形输出。

（源描述文件名：23_test_120.vhd）

第 24 例　分辨信号与分辨函数

袁　媛

1. 电路系统工作原理

本例 24_test_195 所描述的电路系统只有一个类型为 **inout** 的输入输出端口，其示意图如图 24.1 所示。

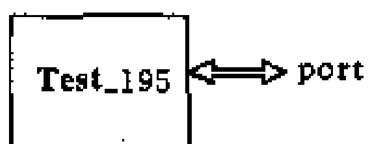


图 24.1　电路系统示意图

2. VHDL 语言描述方法及语法分析

本例是一个结构与行为的混合描述，所要说明的一个关键问题是分辨信号与分辨函数。

(1) 源描述

```
entity Component_Test_195 is
    port (Clock : inout bit);
end Component_Test_195;

architecture Behave_1 of Component_Test_195 is
begin
    Gen_Waveform:
    process
    begin
        wait for 50ns;
        Clock <= not Clock;
    end process;
end Behave_1;

entity Test_195 is
end Test_195;
```

```
architecture Behave_1 of Test_195 is
     component Component_Test_195
          port (Clock : inout bit);
     end component;

     function Or_Pull_Down (V : bit_vector ) return bit is
          variable Result : bit := ' 0' ;
     begin
          for I in V' RANGE loop
               if V(I) = ' 1'  then
                    Result := ' 1' ;
               end if;
          end loop;
          return Result;
     end Or_Pull_Down;

     signal Clock_Signal_A : Or_Pull_Down bit;
     signal Clock_Signal_B : bit;

for all:Component_Test_195 use entity work.Component_Test_195(Behave_1);
begin
     Instance_A: Component_Test_195
                    port map (Clock => Clock_Signal_A);

     Instance_B: Component_Test_195
                    port map (Clock => Clock_Signal_B);
     Waveform: Clock_Signal_A <= ' 0'  after 75 ns,
                                        ' 1'  after 125 ns;
    Finish:
     process
     begin
    wait for 200ns;
    assert false
      report "------End of Simulation------"
    severity error;
     end process Finish;

     Monitor_Clock_Signal_A:
```

```
    process
        variable Clock_Var_A : bit;
    begin
        wait on Clock_Signal_A' TRANSACTION;
        Clock_Var_A := Clock_Signal_A;
    end process Monitor_Clock_Signal_A;

    Monitor_Clock_Signal_B:
    process
        variable Clock_Var_B: bit;
    begin
        wait on Clock_Signal_B' TRANSACTION;
        Clock_Var_B := Clock_Signal_B;
    end process Monitor_Clock_Signal_B;
end Behave_1;
```

(2) 分辨信号与分辨函数

描述首先定义一个实体 component_test_195 以及它的结构体，component_test_195 实际上是一个脉冲发生器，clock 每隔 50ns 翻转一次，产生脉冲信号。在 test_195 的 architecture 内定义一个分辨函数 or_pull_down，不难看出，它是一个线或分辨函数，只要有一个驱动源为'1'，则返回值为'1'，否则为'0'。 在分辨函数之后定义信号 clock_signal_A，它的类型是 or_pull_down，这说明该信号是一个分辨信号，它的取值由分辨函数 or_pull_down 决定。

继之是 test_195 实体，为一个空实体。

此处用两元件例示语句组装两个模块 Instance_A 和 Instance_B，并且将两个定义的信号 clock_signal_A 和 clock_signal_B 分别与端口 clock 对应，所不同的是 clock_signal_A 为分辨信号，而 clock_signal_B 为普通信号。这只是为了进行比较，让读者看一看分辨信号的特殊之处。

在元件例示语句 Instance_A 中：

```
Instance_A:Component_Test_195
port map (clock => clock_signal_A);
```

说明分辨信号 clock_signal_A 已有一个驱动源 clock，在考虑 clock_signal_A 的最终取值（某一时刻）时应该考虑 clock 这一驱动源。其后的语句

```
WaveForm:clock_signal_A <= '0' after 75ns, '1' after 125ns;
```

又给 clock_signal_A 一个驱动源。

把这两个驱动源对 clock_signal_A 的作用综合起来考虑：

① 在 50ns 时，clock 由'0'变'1'，此时第 2 个驱动源还未给 clock_signal_A 赋值，因此由分辨函数可知，clock_signal_A 的值也应由'0'到'1'；

② 在 75ns 时，clock 保持为'1'，第 2 个驱动源赋给分辨信号的值为'0'，但分辨函数的返回值是线或，故 clock_signal_A 的值仍为'1'；

③ 在 100ns 时，clock 由'1'变'0'，第 2 个驱动源在 75ns 到 125ns 之间给 clock_signal_A 的值均为'0'，由分辨函数可知 clock_signal_A 的值由'1'变为'0'；

④ 在 125ns 时，clock 仍为'0'，但第 2 个驱动源给分辨信号赋值为'1'，故 clock_signal_A 的值由'0'变到'1'。

综上所述，经过模拟后的波形应如图 24.2 所示。

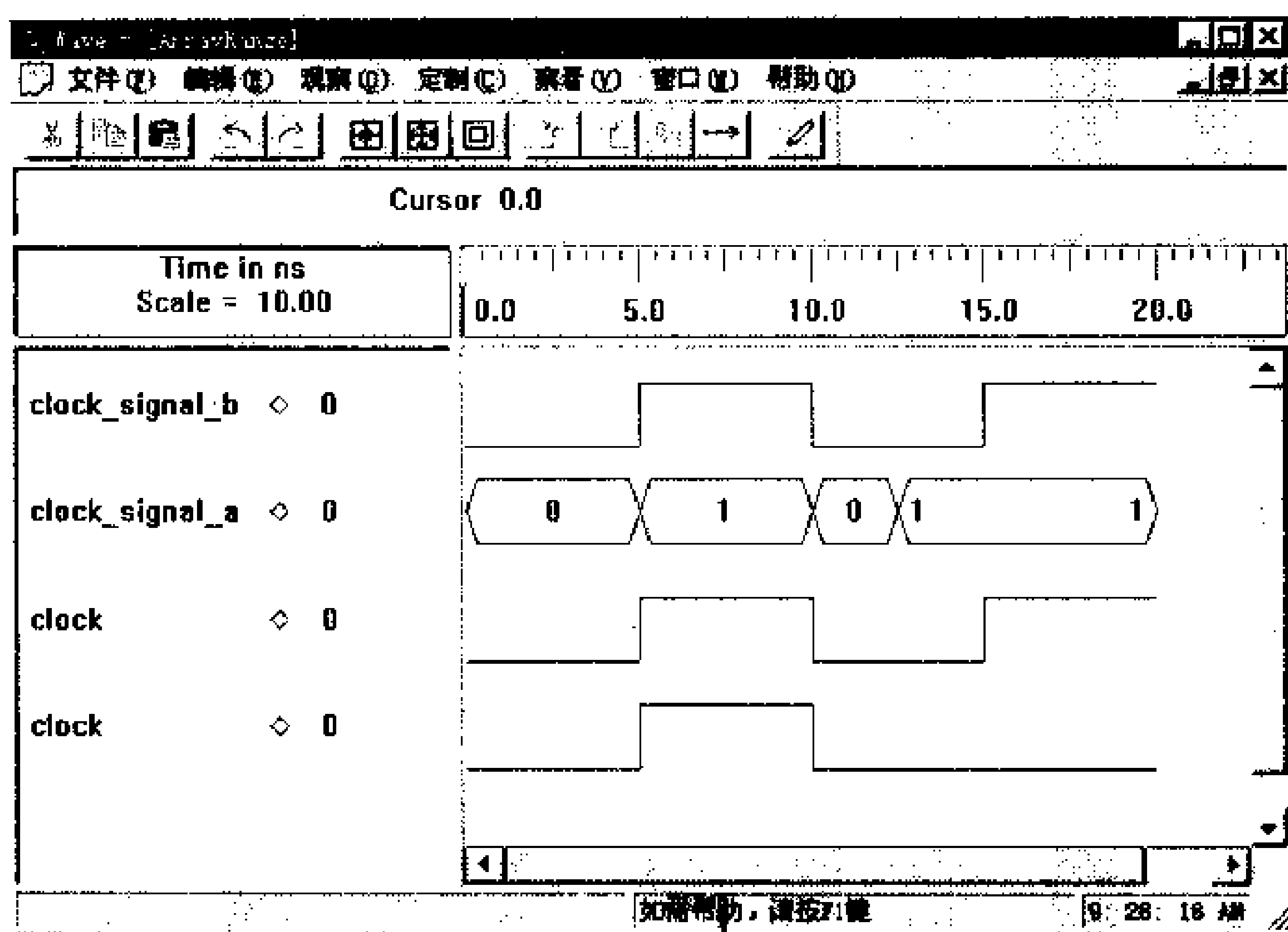


图 24.2 模拟后的波形

从波形中可以看到clock_signal_B的波形与clock完全一样，这与描述也是符合的。

如果将 clock 信号改为每 20 ns 翻转一次，clock_signal_A 的第 2 个驱动源改为如下语句：

```
WaveForm:Clock_signal_A <=  '0' after 75ns, '1' after 85ns,
                            '0' after 95ns, '1' after 105ns,
                            '0' after 115ns, '1' after 125ns,
                            '0' after 135ns;
```

则模拟后的波形应如图 24.3 所示。

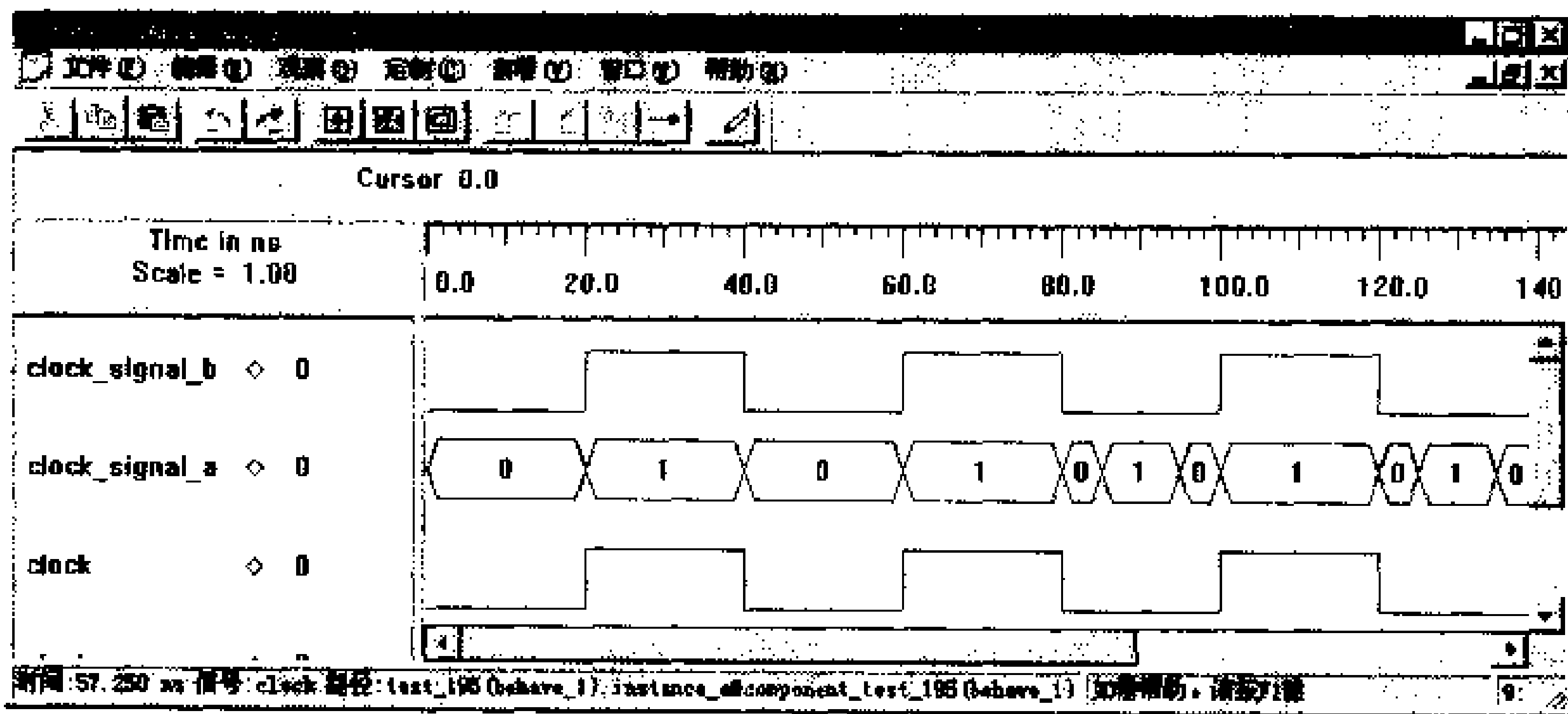


图 24.3 模拟后的波形

请读者自己分析一下上述波形，以便更好地理解分辨函数、分辨信号及其驱动源。

（源描述文件名：24_test_195.vhd）

第25例 信号驱动源

刘沁楠

1. 信号驱动源

当进程为某个信号赋值时，通常要经过一段指定的时间之后该值才会生效，即在给定时间之后为信号安排了一个事务，称该事务为信号的一个驱动源。显然，为信号赋值的任意一个进程都将建立该信号的一个驱动源。每个驱动源是一组（值、时间）对，用以保存每个事务的值及其发生的时间，如驱动源（6，20 ns）。需要注意的是，尽管一个信号（分辨信号）可以有多个驱动源，但每个为信号赋值的进程，无论在该进程中为该信号赋几次值，都只给该信号建立一个驱动源。例如：

```
architecture  Arch  of  Example  is
     signal   A: integer;
begin
     Assignment:
          process
          begin
               A <= transport 1 after 1ns;
               A <= transport 2 after 2ns;
               A <= transport 3 after 3ns;
               wait for 20ns;
          end process;
end Arch;
```

尽管在进程 Assignment 中，为信号 A 进行了三次赋值，但由于只有一个进程为该信号赋值，因而事实上只建立了信号 A 的一个驱动源。

2. 分辨信号

通常，一个信号只有一个驱动源，但如果是分辨信号，则可以有多个驱动源。分辨信号包括多个源和一个分辨函数。分辨函数通过在信号的说明或信号子类型的说明中包含分辨函数名而与需要分辨的信号相关联，它是按设计者意图完成的，用来对各驱动源进行分辨。当分辨信号的驱动值改变时调用分辨函数。

3. 本例涉及的语法现象及模拟结果分析

本例包含文件 25_test_1.vhd，源描述如下：

```
--定义测试台
entity Test_1 is
end Test_1;

--测试台结构体
architecture Behave_1 of Test_1 is
    --信号说明
    signal Sample : bit_vector ( 4 downto 0 ) := (others => '1' );
begin
    --进程 Load_N，为 Sample(4)建立驱动源
    Load_N:
    process
    begin
        Sample(4) <= '1' ;
        wait for 2 ns;
         assert false
            report "Test went OK"
            severity note;
         wait;
    end process;

    --进程 Load_M，为 Sample(1)建立驱动源
    Load_M:
    process
    begin
        Sample(1) <= '1' ;
        wait for 12 ns;
        assert false
            report "Test went OK"
            severity note;
        wait;
    end process;

end Behave_1;
```

在结构体 Behave_1 中首先说明信号 Sample，它是一个五位的位向量，初值为“11111”。结构体的语句部分由进程 Load_N 和进程 Load_M 构成，它们分别为 Sample(4) 和 Sample(1)赋值，即建立 Sample(4)和 Sample(1)的驱动源。对其编译通过后，借助

Vsim/Talent 即可对其进行模拟。

由此可得出以下结论：非分辨复合信号的子元素可视为独立的信号。本例中就是把 Sample(4)和 Sample(1)视为两个独立的信号，进程 Load_N 和 Load_M 分别为这两个信号建立各自的驱动源。如果不这样理解，而将 Sample(4)和 Sample(1)视为一个信号，那么就会产生一个非分辨信号 Sample 有多个驱动源的错误。下面分析另一例，描述如下：

```
--定义测试台（空实体）
entity Test_1a is
end Test_1a;
--结构体
architecture Behave_1 of Test_1a is
    signal Sample : bit_vector ( 4 downto 0 );

begin
    Load_N:
    process
    begin
        for Index in 4 to 4 loop
            Sample ( Index ) <= '1' ;
            wait for 2 ns;
            assert false
                report "Test went OK"
                severity note;
        end loop;
        wait;
    end process;

    Load_M:
    process
    begin
        for Index in 1 to 1 loop
            Sample ( Index ) <= '1' ;
            wait for 12ns;
            assert false
                report "Test went OK"
                severity note;
        end loop;
        wait;
    end process;

end Behave_1;
```

表面看来，它与上述举例并无太大差别，但是由于循环语句的存在，实际上为信号 Sample(1)和 Sample(4)分别建立了多个驱动源。因而在编译、模拟该例的过程中，将会出现："非分辨信号有多个驱动源！"的错误提示。

（源描述文件名：25_test_1.vhd
25_test_1a.vhd）

第 26 例　属性 TRANSACTION 和分辨信号

陈东瑛

1. 电路的 VHDL 语言描述方法及语法分析

本例以一个文件 26_test_74s.vhd（包含一个空实体说明 test_74s 和一个结构体 behave_1）来描述 VHDL 语言分辨信号的语法现象，体现了分辨函数对多源驱动的选择作用。其总体结构与第 19 例类似。

```
--分辨信号与属性 TRANSACTION 的测试台文件 26_test_74s.vhd
entity test_74s is
end test_74s;

architecture behave_1 of test_74s is

  type Logic4 is ('x', '0', '1', 'z');                    --自定义四值逻辑类型
  type Logic4_Vector is array (natural range < > ) of Logic4;
  type Logic4_Table is array (Logic4,Logic4) of Logic4;

  constant Table  : logic4_Table := (('x', 'x', 'x', 'x'),  --分辨选择数组
                                     ('x', '0', 'x', '0'),
                                     ('x', 'x', '1', '1'),
                                     ('x', '0', '1', 'z'));
 function Tristate_RF (V :Logic4_Vector) return Logic4 is
      variable Result : Logic4 := 'z';
  begin
    for I in V'RANGE loop
       Result := Table (Result ,V(I));
       exit when Result = 'x';
    end loop;
    return Result;

  end Tristate_RF;

     subtype Tristate_RS is Tristate_RF Logic4;
     signal Signal_Tristate_RS_S : Tristate_RS := 'z';     --分辨信号说明
     signal Signal_RS_1 : Logic4 ;                         --驱动源信号说明
     signal Signal_RS_2 : Logic4 ;                         --驱动源信号说明
```

```
signal Count : natural := 0;
signal Sevent : natural := 0;
```

behave_1 的说明中定义了分辨函数 Tristate_RF，该函数在子类型指示为 Tristate_RF 的分辨信号被多源驱动时自动调用，同时赋给该分辨信号的多个值组成该函数的数组参数。而对多源驱动的选择规律已经固化在常量数组 Table 中。特例（当驱动源中有'x'，或者有'0'和'1'时，不必判断其他驱动源，立即选定'x'值）由 **exit** 语句结合 Table 的内容也能实现。在用 VHDL 语言描述电路逻辑时，常常需要表示复杂的真值表。利用数组表示和操作可以大大简化程序，代替繁琐的条件赋值语句，是一种清晰、简便、有效的描述方法。

在 behave_1 的说明中相应地定义了一个指示为 Tristate_RF 的分辨信号 Signal_Tristate_RS，进程 res_1 和 res_2 对其进行两源驱动。为验证分辨函数的功能，在进程 res_1 和 res_2 中分别对分辨信号 Signal_Tristate_RS 进行 10 次赋值，即尝试了四值逻辑两源驱动的所有可能（10 种组合）。

```
begin
res_1:                                                  --驱动源 1
process
begin
   Signal_Tristate_RS_S <= ' 1' ;
   Signal_RS_1 <= ' 1' ;
   wait for 10 ns;
   Signal_Tristate_RS_S <= ' 1' ;
   Signal_RS_1 <= ' 1' ;
   wait for 10 ns;
   Signal_Tristate_RS_S <= ' 1' ;
   Signal_RS_1 <= ' 1' ;
   wait for 10 ns;
   Signal_Tristate_RS_S <= ' 1' ;
   Signal_RS_1 <= ' 1' ;
   wait for 10 ns;
   Signal_Tristate_RS_S <= ' 0' ;
   Signal_RS_1 <= ' 0' ;
   wait for 10 ns;
   Signal_Tristate_RS_S <= ' 0' ;
   Signal_RS_1 <= ' 0' ;
   wait for 10 ns;
   Signal_Tristate_RS_S <= ' 0' ;
   Signal_RS_1 <= ' 0' ;
   wait for 10 ns;
   Signal_Tristate_RS_S <= ' x' ;
```

```
    Signal_RS_1 <= 'x' ;
    wait for 10 ns;
    Signal_Tristate_RS_S <= 'x' ;
    Signal_RS_1 <= 'x' ;
    wait for 10 ns;
    Signal_Tristate_RS_S <= 'z' ;
    Signal_RS_1 <= 'z' ;
    wait for 10 ns;
    wait;
end process res_1;

res_2:                                              --驱动源 2
process
begin
    Signal_Tristate_RS_S <= '0' ;
    Signal_RS_2 <= '0' ;
    wait for 10 ns;
    Signal_Tristate_RS_S <= '1' ;
    Signal_RS_2 <= '1' ;
    wait for 10 ns;
    Signal_Tristate_RS_S <= 'z' ;
    Signal_RS_2 <= 'z' ;
    wait for 10 ns;
    Signal_Tristate_RS_S <= 'x' ;
    Signal_RS_2 <= 'x' ;
    wait for 10 ns;
    Signal_Tristate_RS_S <= '0' ;
    Signal_RS_2 <= '0' ;
    wait for 10 ns;
    Signal_Tristate_RS_S <= 'z' ;
    Signal_RS_2 <= 'z' ;
    wait for 10 ns;
    Signal_Tristate_RS_S <= 'x' ;
    Signal_RS_2 <= 'x' ;
    wait for 10 ns;
    Signal_Tristate_RS_S <= 'x' ;
    Signal_RS_2 <= 'x' ;
    wait for 10 ns;
    Signal_Tristate_RS_S <= 'z' ;
    Signal_RS_2 <= 'z' ;
    wait for 10 ns;
    Signal_Tristate_RS_S <= 'z' ;
```

```
    Signal_RS_2 <= ' z' ;
    wait for 10 ns;
    wait;
end process res_2;

Test_Proc:                                    --测试属性 TRANSACTION
process
begin
    wait on Signal_Tristate_RS_S' TRANSACTION;
    if Signal_Tristate_RS_S' TRANSACTION' EVENT=FALSE
    then        Sevent <= Sevent +1;
    end if;
    Count <= Count + 1;
end process Test_Proc;
end behave_1;
```

进程 Test_Proc 由分辨信号 Signal_Tristate_RS 的属性 TRANSACTION 激活，用以体现该属性的意义。

2. 模拟测试向量的选择及模拟结果分析

以 Vsim/Talent 模拟运行本例，终止于 100ns 时刻。信号 Signal_RS_1 和 Signal_RS_2 为分辨信号的两个驱动源。每经过 10ns，分辨信号 Signal_Tristate_RS 与信号 Signal_RS_1、信号 Signal_RS_2 的值体现了分辨函数的选择作用和规律（如图 26.1 所示）。同时信号 count 的累加说明了属性 TRANSACTION 的意义—— 信号上有事务时，信号的信号类属性 TRANSACTION 活跃；信号 sevent 的不累加，也说明属性 TRANSACTION 活跃，即其上有事件发生——值变化。

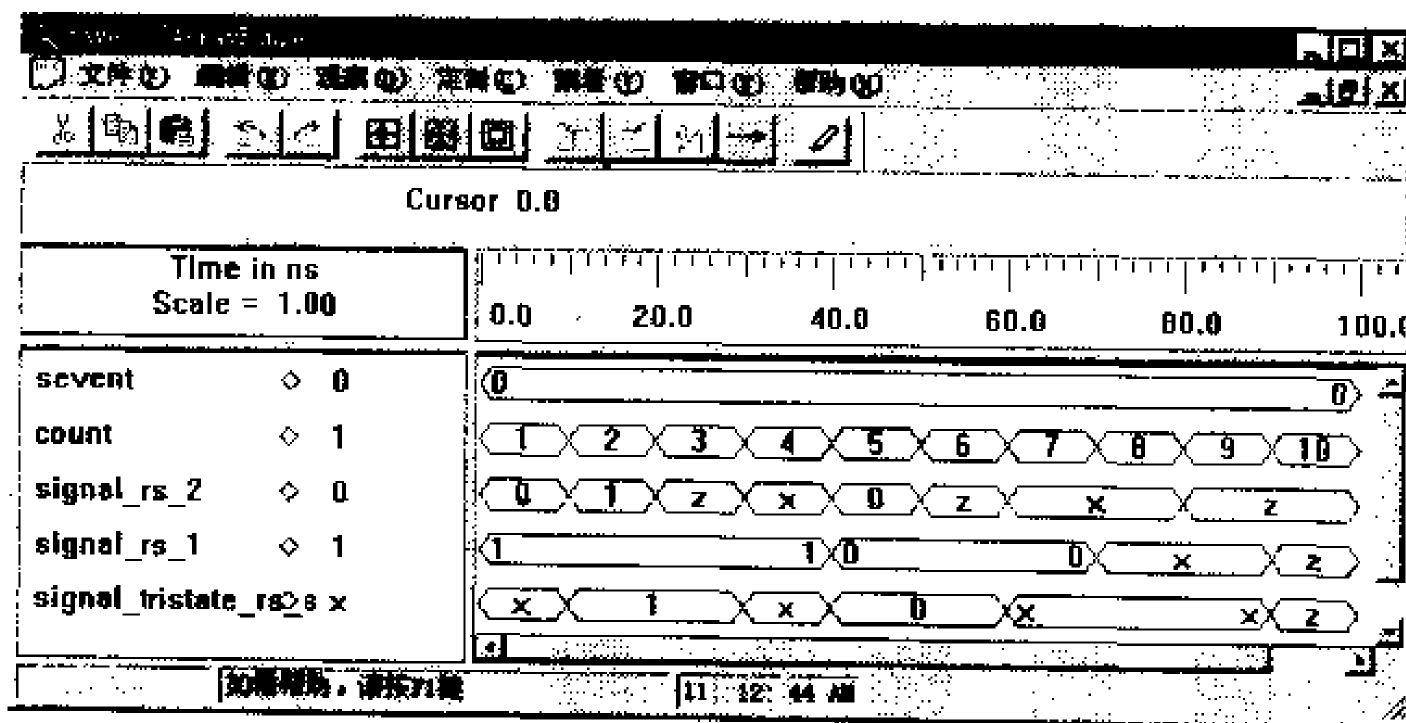


图 26.1　模拟波形

本例通过描述分辨函数，设置分辨信号，并对其施加适当的测试向量，附以各种观察信号，生动体现了分辨函数对多源驱动的作用，验证了属性 TRANSACTION 的意义。

（源描述文件名：26_test_74s.vhd）

第27例　块保护及属性EVENT和STABLE

陈东瑛

1. 电路的VHDL语言描述方法及语法分析

本例以文件27_test_16.vhd（包含一个空实体说明test_16和结构体behave_1）来描述VHDL语言的块保护及属性EVENT和STABLE的语法现象，体现了块保护表达式对信号驱动源的使能作用。其总体结构与第19例类似。

```
--块保护及属性EVENT和STABLE的测试台文件27_test_16.vhd
entity test_16 is
end test_16;

architecture behave_1 of test_16 is
  signal Source : natural := 0;                                   --信号源
  signal Destination_1 : natural:= 0;                             --两组对比信号
  signal Destination_2 : natural:= 0;
  signal Destination_3 : natural:= 0;
  signal Destination_4 : natural:= 0;
  signal Clock : BIT := ' 0' ;

begin

  Blck_Test_1:
  block (Clock = ' 1'  and Clock' EVENT)        --属性EVENT作用的块保护表达式
  begin
    Destination_1 <= guarded Source;
    Destination_2 <= Source;
  end block Blck_Test_1;
  Blck_Test_2:
  block (Clock = ' 1'  and not(Clock' STABLE)) --属性STABLE作用的块保护表达式
  begin
    Destination_3 <= guarded Source;
    Destination_4 <= Source;
  end block Blck_Test_2;
  Monitor:
  process
    variable Source_Var : natural;
```

```
      variable Dest_1_Var: natural;
      variable Dest_2_Var: natural;
      variable Dest_3_Var: natural;
      variable Dest_4_VAr: natural;
   begin
      Source_Var := Source;
      Dest_1_Var := Destination_1;
      Dest_2_Var := Destination_2;
      Dest_3_Var := Destination_3;
      Dest_4_Var := Destination_4;
      wait on Destination_1, Destination_2,
                Destination_3, Destination_4;
   end process Monitor;
   Tick_Tock:                                           --产生时钟振荡
   process
   begin
      wait for 10 ns;
      Clock <=not Clock;
   end process Tick_Tock;
   Source_Wave: Source <=1 after 8 ns, 2 after 15 ns,    --产生信号源
                         3 after 16 ns, 4 after 17 ns,
                         5 after 18 ns, 6 after 19 ns;
end behave_1;
```

VHDL 语言的块可以在保留字 **block** 之后设置保护表达式。保护表达式的值为 true 时，相应块内的被保护的信号赋值语句（标识有 **guarded** 符号）可以实现赋值功能，即被保护的信号驱动源使能（接通）；而保护表达式的值为 false 时，相应块内的被保护的信号赋值语句不可以实现赋值功能，即被保护的信号驱动源不使能（断开）。blck_test_1 和 blck_test_2 两个块用于比较属性'EVENT 和 STABLE 在信号变化过程中的反应。blck_test_1 和 blck_test_2 两个块内部的两条信号赋值语句分别用于比较被保护的信号赋值语句和未被保护的信号赋值语句在被保护块中的作用。进程 monitor 通过变量提取信号的值。

2. 模拟测试向量的选择及模拟结果分析

以 Vsim/Talent 来模拟本例，可以通过波形显示器观察各个信号的变化（如图 27.1 所示），也可以借助对象跟踪工具观察各个变量和信号的值，由此体现块保护及属性 EVENT 和 STABLE 的意义。由于进程 tick_tock 会反复激活，运行时要设置断点或强制中断模拟，以防止模拟死循环。

保护表达式 Clock ='1' **and** Clock'EVENT 与 Clock ='1' **and not** (Clock'STABLE) 的

值表面上应该相等。因为属性 EVENT 在信号上有事件(值变化)时为 true，反之为 false；而属性 STABLE 在信号上无事件时为 true，反之为 false。但是模拟运行时，信号 destination_1 和信号 destination_3 的变化并不一致，在 15~19ns 时段，信号 destination_1的赋值语句并未如信号destination_3的赋值语句那样无作用。因为VHDL语言规定，被保护的块只在其保护表达式中的信号上有事件发生时才计算其保护表达式的值。由此可以解释当 clock 为高电平保持状态时，由被保护块 blck_test_1 隐含定义的信号 guard(体现保护表达式的值)的值为 true，由被保护块blck_test_2 隐含定义的信号 guard 的值为 false，即信号 destination_1 的驱动源接通，而信号 destination_2 的驱动源断开。

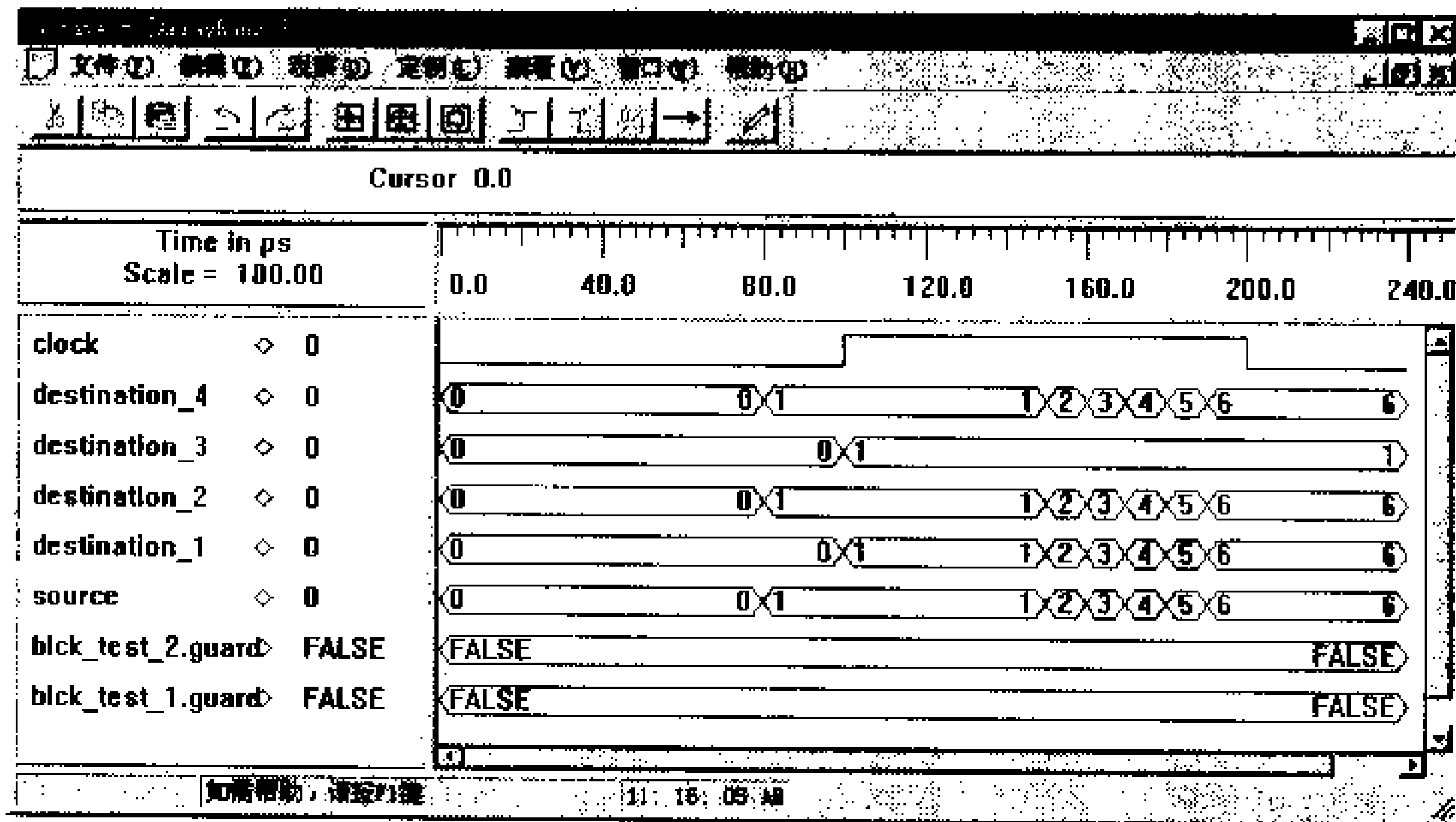


图 27.1 模拟波形

（源描述文件名：27_test_16.vhd）

第 28 例　形式参数属性的测试

刘沁楠

本例不针对具体的设计电路，仅用于测试 VHDL 的语法现象。其中引入了属性的概念，并对信号类属性和函数信号属性加以区别，重点介绍属性 TRANSACTION。

1. 属性

VHDL 中的某些项目类可具有属性，如值类属性、函数类属性、信号类属性、范围类属性等等。借助于这些属性，设计者可以更加简明扼要地组织自己的描述。其中函数类属性又可细分为函数类型属性、函数数组属性、函数信号属性。本例主要区别函数信号属性同信号类属性之间的差异。

函数信号属性用于返回有关信号行为功能的信息，主要包含以下 5 个属性：

① S'EVENT：如果在当前模拟周期内信号发生了某个事件，则返回真；否则，返回假。

② S'ACTIVE：如果在当前模拟周期该信号上有事务发生，则返回真；否则，返回假。

③ S'LAST_EVENT：返回从该信号前一个事件发生至现在耗费的时间。

④ S'LAST_VALUE：返回该信号在最近一个事件发生以前的信号值。

⑤ S'LAST_ACTIVE：返回该信号从前一个事务至现在耗费的时间。

这些属性可以报告一个信号是否有值的变化，该信号原来的值是什么以及从上次事件中的跳变经过了多长时间等等。这类属性可用于函数或过程中。

信号类属性在检测信号变化以及建立时域模型时非常重要。与函数信号属性不同的是，这类属性不能用于子程序内部，否则会给出出错信息。信号类属性有以下四类：

① S'DELAYED：建立与引用信号 S 同类型的信号，其值为该引用信号延迟 t 个时间单位后的值。

② S'STABLE[(t)]：建立一个布尔类型信号，当该引用信号在 t 个时间单位内没有事件发生时，其值为真。

③ S'QUIET[(t)]：若引用信号在 t 个时间单位内无事务发生，则建立一个为真值的布尔信号。

④ S'TRANSACTION：建立一个 bit 类型的信号，当信号上有事务发生时，则 S'TRANSACTION 的值在每个模拟周期对其原来的值进行翻转，即由'0'变为'1'，或由'1'变为'0'。

为理解上述属性的含义，必须搞清楚两个概念：事务和事件。

任何一个信号赋值语句都将在该信号上产生一个事务，无论该信号的值是否改变；而事件则要求该信号的值发生变化。例如：一条信号赋值语句使信号的值从‘0’变为‘1’，则称该信号上有事件发生；如果赋值后，信号值仍为‘0’，则称该信号上有事务。因此，可以说，所有的事件都是事务，但并非所有的事务都是事件，只有那些信号值改变的事务才是事件。

掌握了事务与事件这两个概念，上述属性的含义就显而易见了。

2. VHDL 语法分析及模拟结果

本例由文件 28_test_64a.vhd 构成，源描述如下：

```
--定义测试台(空实体)
entity Test_64a is
end Test_64a;

--结构体
architecture Behave_1 of Test_64a is
  signal Sig_Nat : natural := 0;

--过程定义
  procedure Set_55 ( Signal Sig_Param : in NATURAL;
                                Count_Param : out NATURAL) is
  begin
    wait on Sig_Param' TRANSACTION;
    Count_Param := 55;
  end Set_55;

begin
  Gen_Signal:
  process
    variable Count_PS : natural := 0;
  begin
    Count_PS := Count_PS + 1;
    Set_55 (Sig_Nat, Count_PS);
    wait for 20ns;
  end process Gen_Signal;

  Sig_Nat <= 22 after 20 ns;
end Behave_1;
```

为了对其进行模拟，程序首先定义一个空实体 Test_64a。在结构体说明部分定义一

个子程序，即过程 Set_55。该过程有两个形式参数：Sig_Param 和 Count_Param，它们同为 Natural 类型的信号类对象。其中前者为 **in** 模式，而后者为 **out** 模式（在函数中不允许有 **out** 模式）。子程序的语句部分包含一条 **wait** 语句和一条赋值语句。其中语句"**wait on** Sig_Param' TRANSACTION;"将进程挂起，直到信号 Sig_Param 上有事务发生；它与语句"**wait on** Sig_Param;"的区别在于：对于前者，进程激活的条件为信号 Sig_Param 上有事务发生，而后者则要求信号 Sig_Param 上必须有事件发生。程序的结构体部分则由赋值语句、过程调用以及 **wait** 语句构成，该描述原本的意图是：当信号 Sig_Nat 的值从 0 变为 22 时，Count_PS 的值则从 1 变为 55；但对其编译、模拟时，将被提示为出错信息。原因在于，属性 TRANSACTION 不能用于形式参数，也就是说，它不能在子程序内部被使用。这与前面介绍的属性分类一致，该例的目的也正是为了验证这一点。其解决方案可以将过程改为进程。

（源描述文件名：28_test_64a.vhd ）

第29例　进程和并发语句

刁岚松

1. 电路系统的工作原理

本例给出了信号赋值语句的，用到了进程和并发语句。进程与进程之间、进程与并发语句之间以及并发语句与并发语句之间是并发关系。

2. 电路的 VHDL 语言描述方法及语法分析

部分描述如下所示：

```
--本例并非要设计一个实际电路，只是为了说明 VHDL 语法现象，
--所以这里设计了一个没有输入和输出端口的空实体。
entity Test_35 is

end Test_35;
architecture Behave_1 of Test_35 is
--说明信号并赋初值
    signal   X_Proc_1  : bit    := ' 0' ;
    signal   X_Proc_2  : bit    := ' 0' ;
    signal   X         : bit    := ' 0' ;
    signal   Set       : bit    := ' 0' ;
    signal   Reset     : bit    := ' 0' ;
begin
    Proc_1:                   --进程语句
    --Set 是敏感信号，当 Set 信号值发生变化时，执行该进程
    process (Set)
    begin
        if Set = ' 1'  then
        —当 Set 信号值由' 0' 变为' 1' 时，执行该语句
            X_Proc_1 <= ' 1' ;
        end if;
    end process Proc_1;

    Proc_2:
--进程语句，Reset 是敏感信号
--当 Reset 信号值发生变化时，执行该进程
```

```
        process(Reset)

        begin
            if Reset = '1' then
            --当 Reset 信号值由'0'变为'1'时,执行下面这条语句
                X_Proc_2 <= '0';
            end if;
        end process Proc_2;

        --三条并发条件信号赋值语句
        X <= X_Proc_1 when not X_Proc_1'QUIET else
            X_Proc_2 when not X_Proc_2'QUIET else
            X;

        Set <= '1' after 10 ns, '0' after 20 ns;
        Reset <= '1' after 30 ns, '0' after 40 ns;
    end Behave_1;
```

语法说明：

进程语句之间是并发关系，而进程语句本身则定义单独一组在整个模拟期间连续执行的顺序语句。进程语句用来描述部分硬件的行为。进程语句的格式如下：

```
[进程标号：][postponed]process[（敏感信号表）][is]
        <说明区>
begin
        <顺序语句>
end[postponed]process[进程标号]；
```

进程语句内有一个说明区，区内可以说明数据类型、子程序和变量。在此说明区内说明的变量等于指明一个内存区，在此进程内（仅限于此进程内）可对其进行读/写。如果进程语句中含有敏感信号表，则等价于该进程语句内的最后一条语句是一条隐含的 **wait** 语句。含有敏感信号表的进程语句中不允许再显式出现 **wait** 语句。

注意：

```
Set <= '1' after 10 ns, '0' after 20 ns;
```

等价于

```
Set <= inertial '1' after 10 ns;
Set <= transport '0' after 20 ns;
```

与此类似，Reset <= '1' after 30 ns, '0' after 40 ns;

等价于

```
Reset <= inertial '1' after 30 ns;
```

```
Reset <= transport '0' after 40 ns;
```

在并发条件信号赋值语句中引用了属性：

```
X <=         X_Proc_1 when not X_Proc_1' QUIET
      else   X_Proc_2 when not X_Proc_2' QUIET else
      else   X;
```

X_Proc_1'QUIET 表示 X_Proc_1 信号当前是否有事务发生，如果有事务发生，X_Proc_1'QUIET的值为'0'，否则 X_Proc_1'QUIET的值为'1'。需要说明的是，信号值从'1'变为'0'是一个事务，从'1'变为'1'，从'0' 变为'0'，从'0' 变为'1'也都是事务。只要信号被重新赋值，就是一个事务。这条并发条件信号赋值语句表示，如果信号 X_Proc_1 有事务发生，则信号 X 被 X_Proc_1 赋值；否则，如果信号 X_Proc_2 有事务发生，则信号 X 被 X_Proc_2 赋值。如果 X_Proc_1 和 X_Proc_2 都没有事务发生，则 X 值保持不变。

3. 模拟测试向量的选择及模拟结果分析

下面两条语句为两个进程的敏感信号赋值，使进程和并发条件信号赋值语句能够在适当的时候被激活：

```
Set <= '1' after 10 ns, '0' after 20 ns;
Reset <= '1' after 30 ns, '0' after 40 ns;
```

波形如图 29.1 所示。

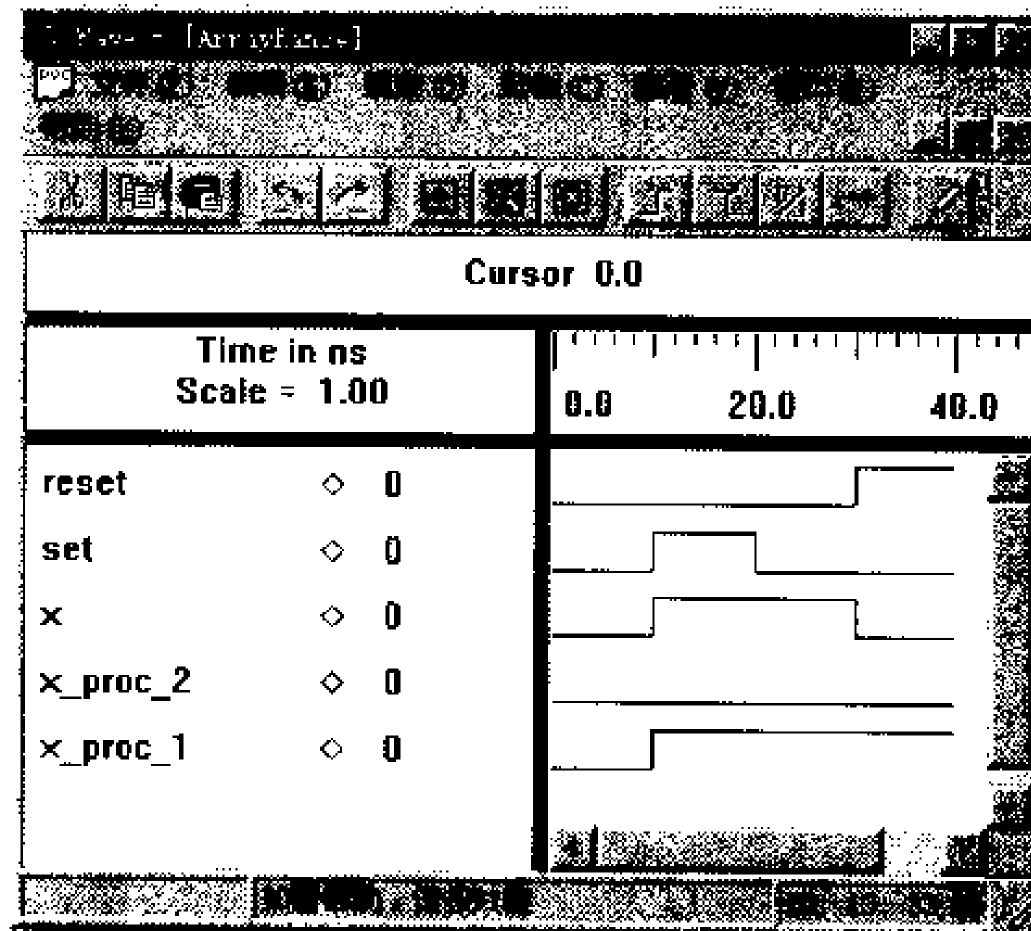


图 29.1 模拟波形

在 0ns 时，Set 和 Reset 的值都是‘0’。10ns 时，Set 的值变为‘1’，进程 Proc_1 被激活，执行语句 X Proc_1 <= ‘1’。接下来执行语句

```
X <= X_Proc_1 when not X_Proc_1' QUIET else
     X_Proc_2 when not X_Proc_2' QUIET else
     X;
```

由于此时 X_Proc_1'QUIET 的值为 false，所以 X 被 X_Proc_1 赋值，值为‘1’。20ns, 30ns 和 40ns 时可以类似分析，不再赘述。

需要特别指出的是，在 30ns 时，Reset 由‘0’变为‘1’，执行进程 Proc_2。在此进程中, X_Proc_2 被赋值为‘0’，尽管 X_Proc_2 在被赋值之前其值就是‘0’，这个赋值操作仍会产生一个事务，所以，X_Proc_2 的 QUIET 属性值为 false。

（源描述文件名：29_test_35.vhd）

第30例　信号发送与接收

刁岚松

1. 电路系统的工作原理

模拟简单的通信，发送进程向接收进程发送一个信号，接收进程检测并接收这个信号。发送进程在 0ns 时对信号 Sample 赋初值'1'，并开始等待接收进程接收该信号。接收进程在 10ns 时开始检测该信号，一旦检测到该信号，接收进程就被挂起。

2. 电路的 VHDL 语言描述方法及语法分析

部分描述如下所示：

```
entity Test_3 is

end Test_3;
architecture Behave_1 of Test_3 is
     signal Sample : bit;
begin
     Sender:
     process
     begin
          Sample <= '1' ;
          wait for 20 ns;
          assert false
               report "Terminated in Sender"
               severity note;
          wait;
     end process;

     Recevier:
     process
     begin
          wait for 10 ns;
          wait until Sample = '1' ;
          assert false
               report "Terminated in Receiver"
               severity note;
```

```
        wait;
    end process;
end Behave 1;
```

工作过程：模拟开始时，Sender 和 Recevier 进程各被执行一次。Sender 进程执行第 1 和第 2 条语句，因为执行了第 2 条 **wait** 语句，所以进程被挂起；Recevier 进程执行第 1 条语句，因为这条语句是 **wait** 语句，所以该进程也被挂起。10ns 时，进程 Recevier 被再次激活，接着执行第 2 条 **wait** 语句。第 2 条语句要求进程等待信号 Sample 上有事件发生，并且值为'1'的条件成立，才继续执行，所以进程 Recevier 在这里被挂起。20ns 时，进程 Sender 被再次激活，未对信号 Sample 产生作用，并且被永久挂起。进程 Recevier 的第 2 条 **wait** 语句的激活条件始终未能满足。

（源描述文件名：30_test_3.vhd）

第31例　中断处理优先机制建模

吴清平

1. 电路系统工作原理

检查多维数组的预定义属性QUIET，并使用此属性进行中断处理优先机制的建模。

2. VHDL语言描述方法及语法分析

本例包含一个实体Test_35b和一个与之相对应的结构体Behave_1。实体Test_35b为空实体：此例为语法现象检查程序，不需要输入输出，所以没有定义端口。结构体Behave_1中包含两个进程和3条并发信号赋值语句，它们完成对中断处理优先级机制的建模。结构体描述如下：

```
architecture Behave_1 of Test_35b is
    type LX01 is ('X', '0', '1');          --自定义数据类型标量
    type LX01_Vector is array (natural range < >) of LX01;
                        --自定义数据类型向量
    signal X            : LX01_Vector(3 downto 0) := (others => 'X');
                        --外部中断标志位，各位代表不同中断请求类型
    signal X_Proc_1     : LX01_Vector(3 downto 0) := (others => '0');
                        --内部中断标志位，各位代表不同中断请求类型
    signal X_Proc_2     : LX01_Vector(3 downto 0) := (others => '0');
                        --内部中断标志位，各位代表不同中断请求类型
    signal Set_X_Proc_1_2 : boolean := FALSE;--内部中断一标志位
    signal Set_X_Proc_2_3 : boolean := FALSE;--内部中断二标志位
    signal Count_1        : natural := 0;    --内部中断一计数器
    signal Count_2        : natural := 0;    --内部中断二计数器
begin
    Proc_1:        --内部中断信号驱动进程一
    process
    begin
        wait until Set_X_Proc_1_2 = TRUE;
                                  --当内部中断一标志位为TRUE时激活
        X_Proc_1(2) <= '1';
        wait for 200 ns;
        Count_1 <= Count_1 + 1; --内部中断计数器加1
    end process Proc_1;
```

```
    Proc_2:          --内部中断信号驱动进程二
    process
    begin
        wait until Set_X_Proc_2_3 = TRUE;
                        --当内部中断二标志位为 TRUE 时激活
        X_Proc_2(3) <= '1';
        wait for 45 ns;
        Count_2 <= Count_2 +1;        --内部中断二计数器加 1
    end process Proc_2;

    --外部中断信号驱动语句
    --根据中断优先级使信号 X 为相应请求
    X <= X_Proc_1 when not X_Proc_1'QUIET else
        X_Proc_2 when not X_Proc_2'QUIET else
        X;
    --两条并发信号赋值语句模拟内部中断触发
    Set_X_Proc_1_2 <= true after 10 ns, FALSE after 400 ns;
    Set_X_Proc_2_3 <= true after 50 ns, FALSE after 500 ns;
end Behave_1;
```

该例给出对中断处理优先机制的建模，Proc_1 进程有较高的优先级，而 Proc_2 进程的优先级较低。每当启动一个进程时，就在信号 X 上放一个相应的中断处理请求，并请求其他进程为此中断处理（注：中断处理进程未在本例中）。模块由两个驱动中断信号 X 的进程和一条条件信号赋值语句组成。条件信号赋值语句完成对两个中断驱动进程所驱动的信号的组合，将 X_Proc_1 或 X_Proc_2 的值驱动至中断请求信号 X 上，然后由中断处理进程根据中断请求信号 X 各位的值决定中断请求类型，分别完成不同中断处理。进程 Proc_1 和 Proc_2 分别等待内部中断请求标志位 Set_X_Proc_1_2 和 Set_X_Proc_2_3，当它们为 TRUE 时，分别将内部中断请求信号 X_Proc_1 或 X_Proc_2 的相应位置 1。

而并发信号赋值语句

```
X <=        X_Proc_1 when not X_Proc_1'QUIET
    else    X_Proc_2 when not X_Proc_2'QUIET
    else    X;
```

则当 X_Proc_1 或 X_Proc_2 发生变化时，将它们的值传给外部中断请求信号 X。这条语句使用了预定义信号属性 QUIET，用于决定信号相对激活程度。属性 QUIET 将建立一个 boolean 型的隐式信号，每当附在属性上面的信号没有事件或者是没有一个指定的时间表达式事件时，该布尔信号为真。因此，使用此预定义属性可以决定一个信号是否刚发生事件。上面的并发信号赋值语句可以在 X_Proc_1 或 X_Proc_2 发生事件时被激活，并

且它能起到判断优先级的作用，即当 X_Proc_1 和 X_Proc_2 上同时发生事件时能决定优先级。在 X_Proc_1 和 X_Proc_2 上同时发生事件时，X_Proc_1'QUIET 和 X_Proc_2'QUIET 的值均为 FALSE，条件信号赋值语句进行计算，并将对第 1 个计算 **when** 表达式返回为真的子句赋值，然后退出不做余下的语句，因此会将 X_Proc_1 的值赋给信号 X，而不再计算 X_Proc_2'QUIET 的值。

3. 模拟结果分析

本例模拟结果的波形图如图 31.1 所示。

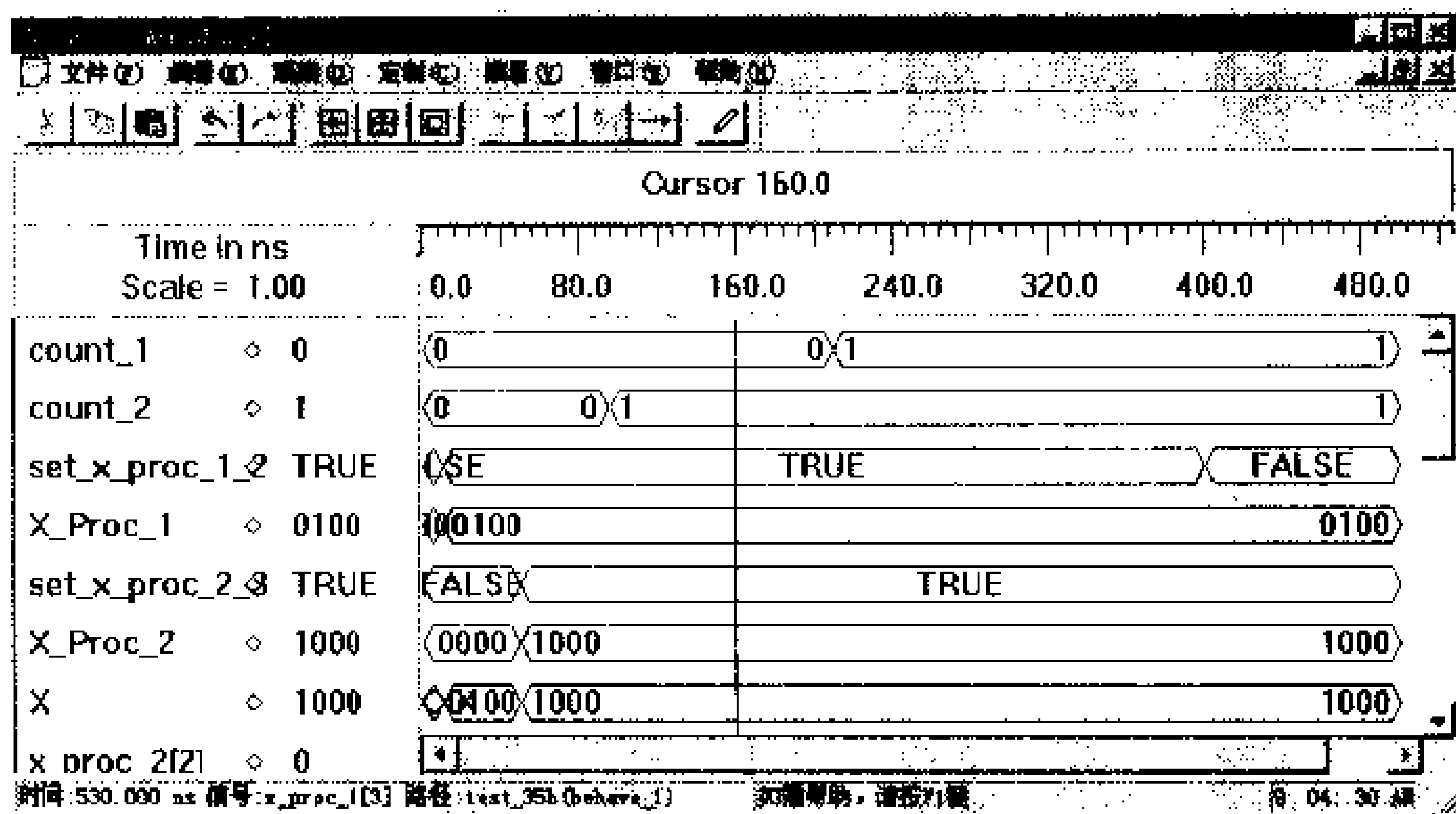


图 31.1　模拟波形

（源描述文件名：31_test_35b.vhd）

第32例 过程限定

刘沁楠

1. 主要语法现象

本例旨在阐述VHDL语言对于过程的一种限定，即不允许在过程中直接建立信号的驱动源。信号或端口的更新必须在进程范围内进行，对于那些具有 **out** 或 **inout** 模式的形式信号参数的赋值同样也只能在进程范围内进行。例如对于以下描述段：

```
B:block
    signal  S1,S2:bit;
begin
    S1<= '1' ;    --合法
    A:process
        procedure P is
        begin
            S2 <= '0' ;  --合法
        end P;
    begin
        P;
        S2 <= '1' ;  --合法
        wait;
    end process;
end;
package body P is
    procedure Q is
    begin
        S <= '1' ;    --不合法
    end P;
```

信号赋值语句“S1 <= ‘1’;”是合法的。因为一个并发信号赋值语句代表着对该信号赋值的等价的进程语句，因而该语句实际上是一个隐含的进程语句；语句“S2 <= ‘0’;”也是合法的，尽管它是过程P中的语句，但该过程定义于进程的说明区，因而这条信号赋值语句是出现于进程范围内的，它是合法的；进程A中的语句“S2 <= ‘1’;”很明显存在于进程中的语句部分，因而也是合法的；对于语句“S <= ‘1’;”而言，存在于过程Q中，而该过程定义于程序包P中，因而语句“S <= ‘1’;”不存在于任何进程语句内，它是不合法的。了解了上述限定，就可以分析本例。源描述如下：

```
--定义测试台(空实体)
entity Test_110b is
end Test_110b;

--结构体
architecture Behave_1 of Test_110b is

--信号说明
signal A_Sig : NATURAL := 0;
signal B_Sig : NATURAL := 0;
signal C_Sig : NATURAL := 0;

--子程序定义
procedure Concur_Proc is
     variable Count_CP : NATURAL := 0;
begin
     Count_CP := Count_CP +1;
     if A_Sig = 55 then
     Count_CP := Count_CP + 1 ;
     wait until B_Sig =65;
     C_Sig <= 75 after 10ns;
     end if ;
     Count_CP :=Count_CP + 1;
end Concur_Proc;

begin

Gen_Signals:
process
     variable Count_GS : Natural := 0 ;
begin
     wait for 10 ns;
     Count_GS := Count_GS + 1;
     A_Sig <= 55;
     wait for 10 ns;
     Count_GS := Count_GS + 1;
     B_Sig <=65;
     wait on C_Sig;
     Count_GS := C_Sig;
end process Gen_Signals;

--过程调用
```

```
Concur_Proc;

end Behave_1;
```

该程序首先定义一个空实体 Test_110b，其对应的结构体 Behave_1 的说明部分定义一个子程序，即过程 Concur_Proc；结构体 Behave_1 的语句部分由一条进程语句构成。在进程 Gen_Signals 中的信号赋值语句“A_Sig <= 55;”和“B_Sig <= 65;”都是合法的，而在过程 Concur_Proc 中的语句 “C_Sig <= 75 **after** 10 ns;”则不合法，因为它不存在于任何进程的范围内。

2. 编译结果的比较

利用 Talent 系统中的编译器对本例进行编译出现报错信息：“C_Sig can not have drivers in procedure.”，这与前面的分析结果一致。为使编译通过，修改的方法很简单，只需将过程 Concur_Proc 放在进程中即可。

在 Cadence 上编译，出现报错信息：“Update of port/signal must occur in process statement.”，这表明 Talent 系统在处理该问题上与 Cadence 一致。

（源描述文件名：32_test_110b.vhd）

第 33 例　整数比较器及其测试

刘沁楠

1. 电路系统工作原理

本例描述一个简单的整数比较器，用于比较两个整数值的大小。其示意图如图 33.1 所示。

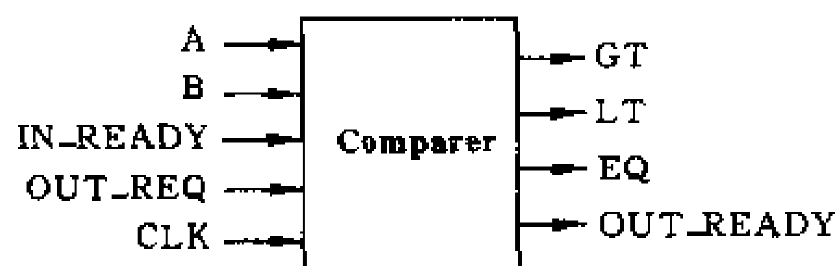


图 33.1　整数比较器

该比较器有 5 个输入端和 4 个输出端口，其中端口 A，B 分别输入要比较的两个整数，取值范围为 0～255；CLK 为时钟输入端，用以支持系统的同步工作，采用时钟上升沿同步；IN_READY 是一个输入端口，当它为高电平时，表明运算数据（信号 A，B）已准备好，可以进行比较运算。另一个输入端口 OUT_REQ 为输出请求信号端口，当其为高电平时，表明本器件请求向输出端口输出比较结果，此时置输出完成信号 OUT_READY 为 "1"。输出端口 OUT_READY 同 IN_READY 和 OUT_REQ 一样，为握手信号。高电平时表明输出准备就绪，可向输出端口输出结果；当输出请求信号 OUT_REQ 变为 "0"，表示结果已取走，此时，OUT_READY 置为 "1"；输出端口 GT，LT，EQ 用于表示比较器的比较结果，其数据类型为 bit。输入与输出间的关系如表 33.1 所示。

表 33.1　整数比较器函数表

输入	GT	LT	EQ
A < B	0	1	0
A = B	0	0	1
A > B	1	0	0

2. VHDL 语言描述方法及语法分析

本例由文件 33_comparer.vhd，33_comp.vhd 以及 33_simu.vhd 完成整数比较器的 VHDL 描述。其中文件 33_comparer.vhd 主要包含 **library** 子句、**use** 子句以及一个空实

体；文件 33_comp.vhd 则是设计实体 comp 完整的行为描述，即从算法级对比较器的功能进行描述；文件 33_simu.vhd 提供了一个测试台(test-bench)，通过将激励作用于被测试模块，以验证设计描述是否能完成比较器的功能。下面结合源描述，详细讨论本例中所蕴涵的重要的 VHDL 语法现象。

(1) 设计库

首先，分析文件 33_comparer.vhd，源描述如下：

```
-- 利用 library 子句和 use 子句使设计库中的资源可见
library   STD;
library   WORK;
library   COMP;
use   STD.STANDARD.all;
use   COMP.TYPES.all;
use   WORK.all;

--定义整数比较器的测试台实体(空实体)
entity test_comp is
end test_comp;
```

通常，设计者把自己的电路设计实现放在一个 VHDL 设计库中。同时，还可访问其他多个设计库，以利用这些库中的设计单元（这些设计单元是指在编译之后插入设计库的实体说明、结构体等），将其作为自己的 VHDL 描述资源。在使用这些设计库时，必须在设计单元的开始部分用 **library** 子句说明，正如描述中的前三条语句。其中逻辑名为 STD 的库为所有设计单元隐含定义，它主要包含有预定义的程序包 STANDARD 与 TEXTIO。在这些程序包中定义了若干类型、子类型、函数、子程序等。WORK 库为当前工作库，它可接受用户分析和修改的设计单元。程序中出现的另一个库名为 COMP，用户需要自己创建该库，并将包 TYPES 置于该库中，这样语句“**use** COMP.TYPES.**all**;”在编译时才不会出错。**library** 子句使 WORK 库、STD 库以及 COMP 库中的所有资源可见，**use** 子句则使设计者可直接访问库中各项，并可缩短项目的引用。本例中语句“**use** WORK.**all**;”表明设计者可直接引用 WORK 库中的所有库单元；语句“**use** COMP.TYPES.**all**;”则允许设计者直接访问 COMP 库中程序包 TYPES 所定义的所有项目。描述最后部分是一个空的实体说明，这是测试台(“test-bench”)所必需的。

(2) 设计实体

下面分析文件 33_comp.vhd：

```
-- 比较两个整数
```

```
-- 定义程序包 Types
package TYPES is
   subtype short is integer range 0 to 255;
end TYPES;
use WORK.TYPES.all;

-- 实体说明
entity COMP is
    port(A          : in short;
         B          : in short;
         IN_READY   : in bit;
         OUT_REQ    : in bit;
         CLK        : in bit;
         OUT_READY  : out bit;
         GT         : out bit;
         LT         : out bit;
         EQ         : out bit);
end COMP;

-- COMP 的结构体

architecture ALGORITHM of COMP is
begin
     -- 比较器的算法级行为描述
     process
     begin
    wait until CLK'EVENT and CLK='1' and IN_READY ='1';
     -- 比较两个数值
     -- A > B
     if A > B then
          GT <= '1' ;
          LT <= '0' ;
          EQ <= '0' ;
     else
          -- A < B
          if A < B then
               GT <= '0' ;
               LT <= '1' ;
               EQ <= '0' ;
          else
          -- A = B
               GT <= '0' ;
```

```
                LT <= ' 0'  ;
                EQ <= ' 1'  ;
            end if;
        end if;
    wait until CLK'EVENT and CLK= ' 1'  and OUT_REQ = ' 1' ;
        OUT_READY <= ' 1' ;
    wait until CLK'EVENT and CLK= ' 1'  and OUT_REQ = ' 0' ;
        OUT_READY <= ' 0' ;
        end process;
    end ALGORITHM;
```

该文件主要包含两部分内容。首先是程序包TYPES的说明，其中定义了子类型short，它以整型作为基类型，值域为[0, 255]。第2部分内容是一个设计实体，它描述比较器的行为功能。在VHDL描述中，大到系统、小至门电路，都对应于一个设计实体。设计实体由一个实体说明和一个结构体组成，如图33.2所示。

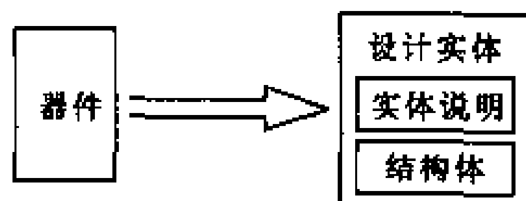


图33.2 设计实体示意图

实体说明是一个器件的外部视图，反映器件的外部端口特征。它通常包含类属表、端口表以及一些接口的公共信息，这些内容可以缺省，即一个空实体也是允许存在的。

本例中的实体说明主要由一个端口说明构成。5个端口为 **in** 模式，其中A, B为用户自定义的short类型，其他为标准数据类型bit；端口GT，LT，EQ，OUT_READY则为 **out** 模式，可向其中写数据。由此可见，端口表规定了端口的数目、类型以及模式，从而描述出本实体对外界的接口。

紧接着实体说明的是实体COMP对应的结构体ALGORITHM。结构体用以描述一个设计的结构或行为，把设计的输入、输出之间的关系建立起来。本例是对比较器行为功能的描述。

(3) 进程

本例中的功能描述由一条进程语句完成。进程语句是行为描述的基本单元，进程语句之间是并发的关系，在一个进程内部出现的语句是顺序语句。进程语句语法规则为：

```
[进程标号：]  process  [(敏感信号表)]
                   < 说明部分 >
                   begin
```

```
        < 顺序语句 >
        end  process  [进程标号];
```

说明部分可说明一些数据类型、子程序、变量等，其中变量仅在本进程内可读／写。本例中说明部分缺省。

紧跟在 **process** 之后的敏感信号表等价于该进程语句内最后一条语句是“**wait on** 敏感信号表；”，每当表中的一个或多个元素值改变时，进程将被激活。含有敏感信号表的进程语句中不允许再显式出现 **wait** 语句。本例中无敏感信号表，由 3 条 **wait** 语句控制进程的同步。

对于“**wait until** 条件表达式”格式的 **wait** 语句，敏感信号表是由条件表达式中出现的信号组成。本例中对于语句“**wait until** CLK'EVENT **and** CLK = '1' **and** IN_READY = '1';”而言，其敏感信号表由 CLK 和 IN_READY 构成。

读者还应当知道，一个进程实际上是一个无限循环；在进程中的顺序语句执行期间，当未遇到 **wait** 语句时，模拟时钟不会向前推进；信号的当前值仅在 **wait** 语句进行更新，此后，模拟时钟才前进一步。理解这一点有助于分析模拟结果。

本例进程中的顺序语句除 **wait** 语句之外，还有 **if** 语句和信号赋值语句。**if** 语句同一般高级语言中的 **if** 语句并无差别，此处通过它可实现比较算法。顺序信号赋值语句指定无延时，在模拟期间，为保证同步，采用 delta 延时。

3. 模拟及结果分析

VHDL 是一种基于模拟的硬件描述语言。为了便于模拟验证，VHDL 提供了“测试台”(test-bench)机制。本例中文件 33_stim.vhd 即为“测试台”的一部分。源描述如下：

```
-- test_comp 的结构体
architecture BENCH of test_comp is

--被测元件的端口说明
    component comp
        port(
         A              : in short;  --端口数据类型定义
         B              : in short;  --端口数据类型定义
         IN_READY       : in bit;
         OUT_REQ        : in bit;
         CLK            : in bit;
         OUT_READY      : out bit;
         GT             : out bit;
         LT             : out bit;
         EQ             : out bit   --端口数据类型定义
```

```
        );
    end component;

    --激励信号
    signal A:short;
    signal B:short;
    signal IN_READY:bit;
    signal OUT_REQ:bit;
    signal CLK:bit;
    signal OUT_READY:bit;
    signal GT:bit;
    signal LT:bit;
    signal EQ:bit;
    --组装
for all: comp use entity work.comp;
begin
    comp_I1: comp  --元件例示
        port map (
        A => A,    --激励信号送到输入端口
        B => B,
        IN_READY => IN_READY,
        OUT_REQ => OUT_REQ,
        CLK => CLK,
        OUT_READY => OUT_READY,
        GT => GT,  --响应信号送到输出端口
        LT => LT,
        EQ => EQ
       );

    --产生激励
    comp_driver:
process
    begin
      wait until  CLK = ' 1' ;
      IN_READY <= ' 0' ;
      A <= 25;
      B <= 23;
      OUT_REQ <= ' 0' ;

      wait until  CLK = ' 1' ;
      IN_READY <= ' 1' ;
```

```
wait until  CLK = '1';
OUT_REQ <= '1';

wait until CLK'EVENT and CLK = '1'  and OUT_READY =  '1';
OUT_REQ <= '0';

wait until  CLK = '1';
IN_READY <= '0';
A <= 35;
B <= 48;
OUT_REQ <= '0';

wait until  CLK = '1';
IN_READY <= '1';

wait until  CLK = '1';
OUT_REQ <= '1';

wait until CLK'EVENT and CLK = '1'  and OUT_READY =  '1';
OUT_REQ <= '0';

wait until  CLK = '1';
IN_READY <= '0';
A <= 58;
B <= 58;
OUT_REQ <= '0';

wait until  CLK = '1';
IN_READY <= '1';

wait until  CLK = '1';
OUT_REQ <= '1';

wait until CLK'EVENT and CLK = '1'  and OUT_READY =  '1';
OUT_REQ <= '0';

--判断结果
assert false
report "---End of Simulation---"
severity error;
```

```
    end process;

        --产生时钟激励
        CLK <= not CLK after 50 ns;
end BENCH;
```

测试台是一个空实体，它不定义任何与外界发生关系的输入输出端口。在前面提到的 33_comparer.vhd 文件中则定义了空实体 test_comp。在测试台的结构体 BENCH 中定义被测元件的模板 COMP，并通过组装将其与实际元件（即文件 33_comp.vhd 中的设计实体 comp）相连，然后利用元件例示将激励信号引到元件的输入端，其输出端连到响应信号，如图 33.3 所示。

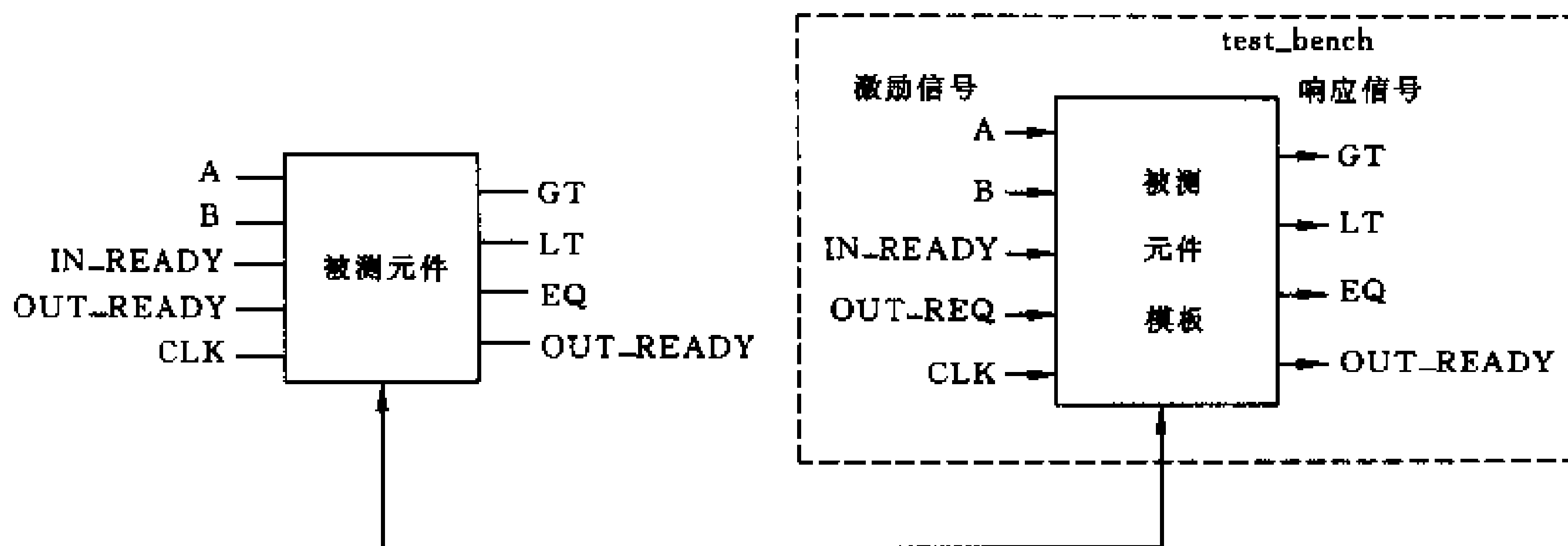


图 33.3　整数比较器“test-bench”示意图

在进程 comp_driver 中是一系列用于产生激励的行为描述，主要由信号赋值语句和 wait 语句完成。此外，并发信号赋值语句“CLK <= **not** CLK **after** 50 ns;”用于产生时钟激励，时钟周期为 100ns。

依次对文件 33_comp.vhd，33_comparer.vhd 以及 33_simu.vhd 编译之后（顺序不可颠倒），就可借助 Vsim/Talent 进行模拟。模拟后产生的波形如图 33.4 所示。

结合图 33.4，以单步跟踪的方式详细讨论该例的模拟全过程。在结构体内部，各语句之间是并发关系。模拟开始时刻，进程 comp_driver 中的语句“**wait until** CLK = ‘1’;”使进程挂起。当模拟时钟为 50ns 时，CLK 变化，执行语句“CLK <= **not** CLK **after** 50 ns;”，激活进程 comp 和进程 comp_driver，由于进程 comp 中条件不满足(IN_READY = ‘0’)，因而该进程被挂起；进程 comp_driver 中条件满足，顺序执行 4 条信号赋值语句：“A <= 25;”，“B <= 23;”，“IN_READY <= ‘0’;”，“OUT_REQ <= ‘0’;”。第 2 条“**wait until** CLK=

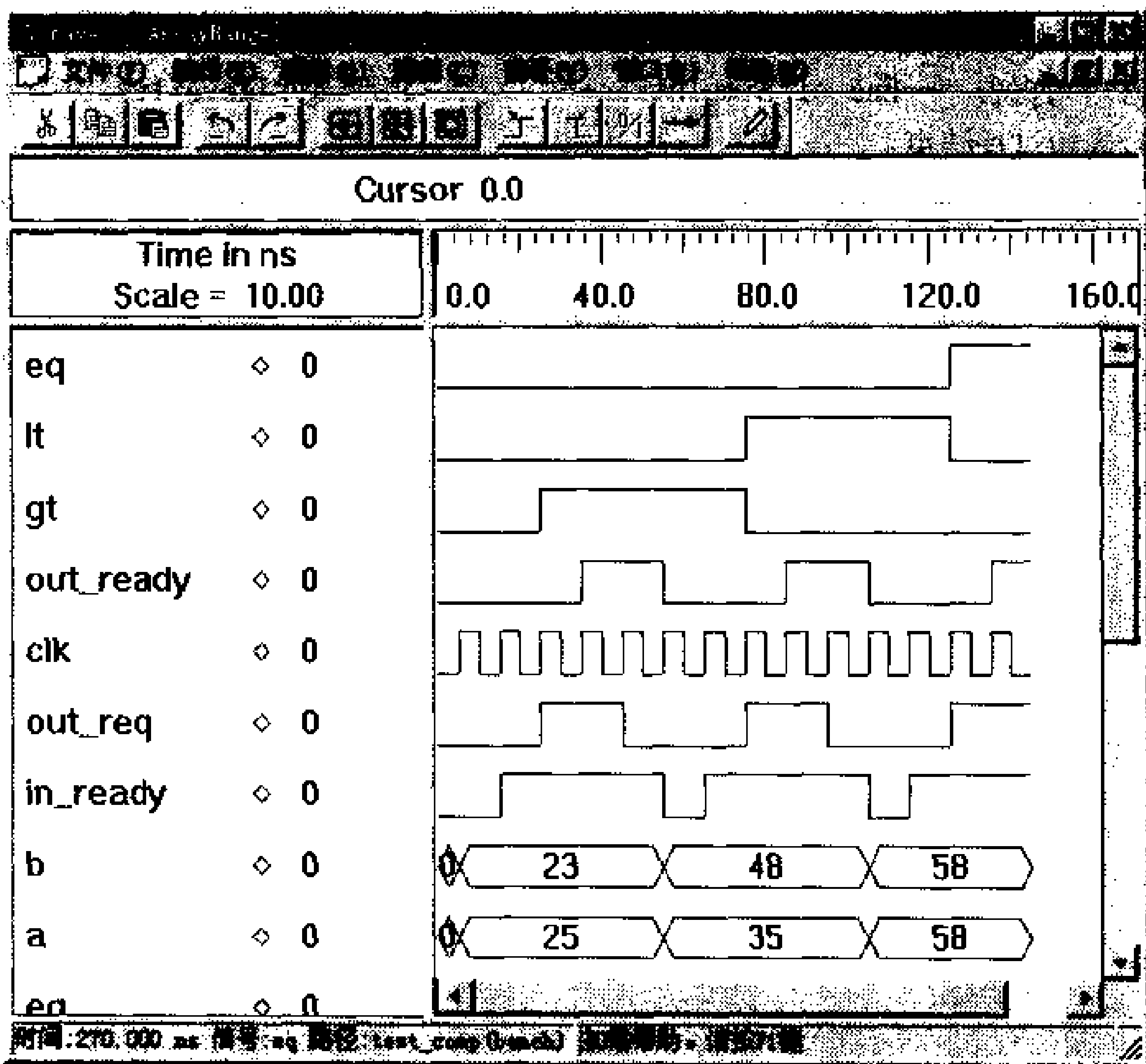


图 33.4　比较器模拟波形图

'1';"语句再次使进程挂起；当模拟时钟为 100ns 时，CLK 的值改变，执行语句"CLK <= **not** CLK **after** 50 ns;"，CLK = '0'，两个进程等待条件均不满足，因而再次被挂起；模拟时钟为 150ns 时，CLK = '1'，激活进程 comp_driver，且条件满足，IN_READY 上跳变，第 3 条 **wait** 语句再次将该进程挂起，同时激活进程 comp，但由于 delta 延迟的存在，使得 **wait** 语句中的条件仍不满足，进程 comp 亦再次挂起；模拟时钟为 250ns 时，两进程再次激活，此时进程 comp 中条件满足，顺序执行 **if** 语句，结果 GT 输出高电平，其后出现的 **wait** 语句将进程挂起。在进程 comp_driver 中，OUT_REQ 为高电平，第 4 条 **wait** 语句再次将其挂起。模拟时钟为 350ns 时，进程 comp 中的条件满足，OUT_READY 为高电平，然后由 **wait** 语句将其挂起；由于 delta 延迟存在，进程 comp_driver 中的条件尚未满足，因而 OUT_REQ 仍为高电平；模拟时钟为 450ns 时，进程 comp_driver 中等待语句条件满足，OUT_REQ 变为低电平，同样由于 delta 延迟，comp 进程仍被挂起，直至 550ns 时，OUT_READY 才变回为低电平，以便为下一次比较作准备。

此后处理另外两组测试向量；即"A=35, B=48"，"A=58, B=58"。整个过程同前所述，

通过波形图可以很容易看出比较结果。

断言语句报告“—End of Simulation--”之后，将严谨性(**severity**)设为 error，从而导致模拟过程结束。

（源描述文件名：33_comparer.vhd
33_comp.vhd
测试平台文件名：33_simu.vhd）

第 34 例 数据总线的读写

刁岚松

1. 电路系统的工作原理

本例旨在描述总线的行为。当总线准备好信号 bus_ready 变为'1'时，从总线读取的是有效值。将总线准备好信号 bus_ready 作为进程的敏感信号，一旦 bus_ready 变为'1'，就激活进程，此时从总线读取数据。本例多次从总线读取数据，并从中找出最大值，然后将最大值写回数据总线。

2. 电路的 VHDL 语言描述方法及语法分析

部分描述如下：

```
package p is
    attribute cycle_time:Time;
    attribute max_cycles:Integer;
    attribute clock_phases:Integer;
    attribute Integer_width:Integer;
    Type my_integer is range -2**15 to 2**15-1;
    Type array_of_integer is array(1 to 16) of my_integer;
    Type my_integer_Vector is array(Natural range< >) of my_integer;
    --说明一个分辨函数
    function wired_and (inputs : my_integer_Vector) return my_integer;
end package p;
package body p is
    function wired_and (inputs : my_integer_Vector) return my_integer is
    --分辨函数定义
    variable m:  my_integer:=0;
    begin
        if  inputs' Length=0 then      --如果所有的驱动源都断开
            return 0;                  --返回 0
        else
            --下面从所有连接的驱动源中寻找最大值并返回
            for i in inputs' Range loop
              if inputs(i) > m then
                   m:=inputs(i);
              end if;
```

```
            end loop;
            return m;
        end if;
    end;
end p;

entity e is
    --总线作为输入/输出端口
    --总线准备好信号作为输入端口
    port(bus1  : inout wired_and my_integer;
           bus_ready : in Bit);
        --自定义实体 e 的属性 cycle_time
       attribute cycle_time of e : entity is 50 ns;

        --自定义实体 e 的属性 clock_phases
       attribute clock_phases of e : entity is 1;

end e;

architecture arch of e is
   begin
      p1:Process
           variable a :array_of_integer;
           variable max:my_integer;
           variable loop_time:time;
           variable i:my_integer;
        begin
        --开始读总线
        loop_time  :=Now;
        --循环 16 次，将 16 个数据读入数组 a 中
        --每次循环中等待 bus_ready 变为'1'
        --每当 bus_ready 变为'1'，bus1 的值被送到数组 a 中
        l11:for i in 1 to 16 loop
            wait until bus_ready='1';
            a(i):=bus1;
       end loop l11;
       --读取数据的时间不能大于实体 e 的 cycle_time 属性的 16 倍
       assert (Now-loop_time)<=(16*e'cycle_time)
              report "loop took too long.";
       wait for 10 ns;
       --下面开始寻找数组 a 中的最大值
        max:=a(1);
```

```
        l12: for i in 2 to 16 loop
            if a(i) > max then max:=a(i);
            end if;
        end loop l12;
        --总线获得了数组 a 中的最大值
        bus1 <= max after 50 ns;
        end process p1;
end arch;
```

语法分析：

在实体说明和结构体内部定义的数据类型、常量及子程序对其他设计单元不可见。为了使类型、常量及子程序对若干设计单元可见，VHDL 提供了程序包机制。包是一个库单元（设计单元），包含有可用于其他设计单元的一系列说明。一个包可分为两个单元：说明单元与包体单元。包体并不总是需要的。但在包中包含有子程序说明时则必须有对应的包体。在这种情况下，子程序体不能出现在包说明中，而必须放在包体中。只有在包说明单元中说明的标识符才在包之外可见。

包单元及其对应的包体单元的一般格式如下所示：

```
package 包名 is
     --说明单元
end [package][包名];
-包体名总是与对应的包说明的名字相同
package body 包名 is
     --说明单元
end [package body][包名];
use work.p.all;
use 子句格式：
  use 包名.标识符名;      --使包中特定的说明可见
  use 包名.all;           --使包中所有的说明可见
```

3. 模拟测试向量的选择及模拟结果分析

测试台的部分描述如下：

```
bus1    <= 3;
bus_ready    <= '0';
wait until clk = '1';
bus_ready    <= '1';
wait until clk = '1';
```

```
    bus1    <= 7;
    bus_ready   <= '0' ;
    wait until clk = '1' ;
    bus_ready   <= '1' ;
    wait until clk = '1' ;
    bus1    <= 5;
    bus_ready   <= '0' ;
    wait until clk = '1' ;
    bus_ready   <= '1' ;
    wait until clk = '1' ;
    bus1    <= 8;
    bus_ready   <= '0' ;
    wait until clk = '1' ;
    bus_ready   <= '1' ;
    wait until clk = '1' ;
    bus1    <= 9;
    bus_ready   <= '0' ;
    wait until clk = '1' ;
    bus_ready   <= '1' ;
    wait until clk = '1' ;
    bus1    <= 6;
    bus_ready   <= '0' ;
    wait until clk = '1' ;
    bus_ready   <= '1' ;
    wait until clk = '1' ;
    bus1    <= 4;
    bus_ready   <= '0' ;
    wait until clk = '1' ;
    bus_ready   <= '1' ;
    wait until clk = '1' ;
    bus1    <= 2;
loop1:for num in 0 to  18 loop
    bus_ready   <= '0' ;
    wait until clk = '1' ;
    bus_ready   <= '1' ;
    wait until clk = '1' ;
    end loop loop1;
    assert false report "End of Simulation" severity error;
```

模拟开始时，bus1 被赋值为 3，bus_ready 被赋值为'0'。在执行下面的语句时，进程 p1 被挂起，等待 bus_ready 的值变为'1'。

```
l11:for i in 1 to 16 loop
  wait until bus_ready=' 1' ;
  a(i):=bus1;
end loop l11;
```

在第 1 个时钟上升沿，bus_ready 被赋值为'1'。此时进程 p1 被激活，向量 a 的第 1 个元素被 bus1 赋值。然后进程 p1 被再次挂起，等待下一次 bus_ready 变为'1'。以后每当 bus_ready 变为'1'时，向量 a 就从 bus1 读入一个值。循环读入 16 次后，向量 a 中就读入了 16 个值。在本例中，这 16 个值分别是：3, 7, 5, 8, 9, 6, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2。然后，通过执行下面的语句，找出最大值并输出：

```
max:=a(1);
l12: for i in 2 to 16 loop
     if a(i) > max then max:=a(i);
     end if;
end loop l12;
bus1<=max after 50 ns;
```

输出波形见图 34.1。从 670ns 处开始，总线输出最大值 9。

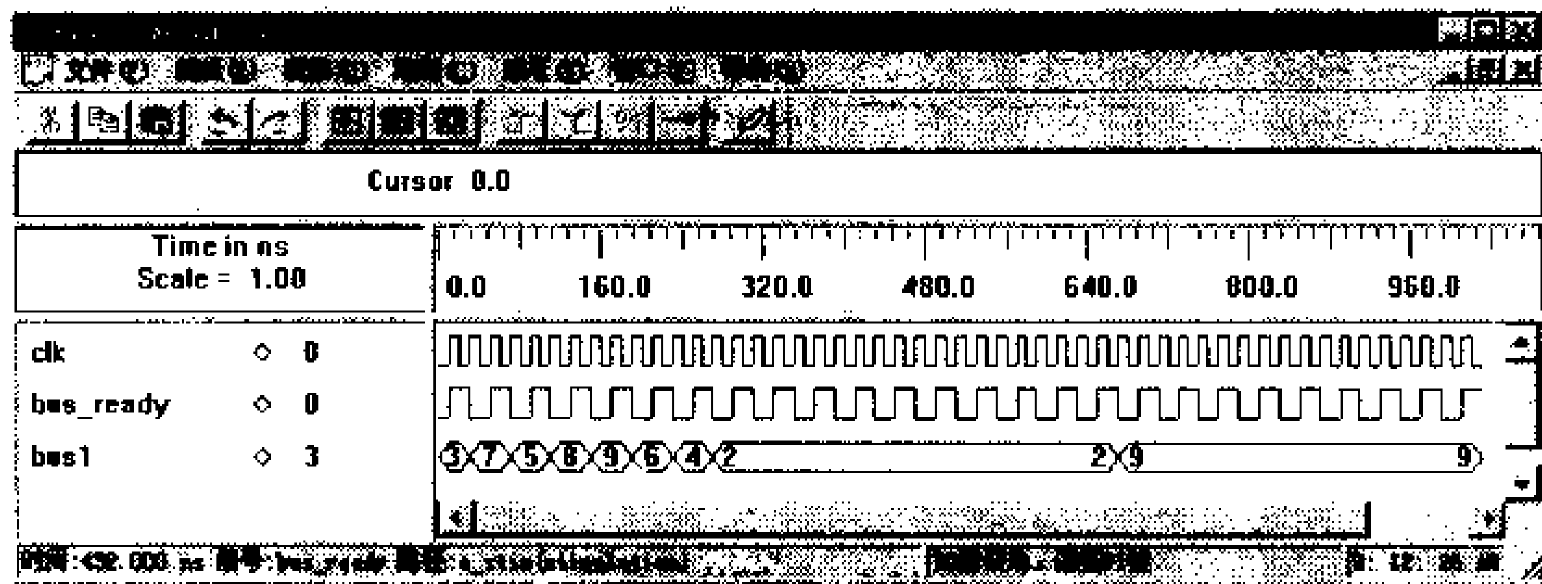


图 34.1　数据总线读写的输出波形图

（源描述文件名：34_bus_readwrite.vhd）

第35例　基于总线的数据通道

李　春

1. I486 总线接口设计

存储器和输入输出设备通常通过一个中介与微处理器相连，这个中介一般是一个三态的总线。为了保证数据在总线上的正常传输，微处理器的总线和存储器的时序特性需要仔细考虑。第 75 例为存储器设计了时序模型，本例将为微处理器的总线设计一个时序模型，在设计中称为 486 微处理器的总线中介单元。实际的 486 总线的接口是很复杂的而且支持多种类型的总线周期。例如，它有更多其他的接口信号控制在同一时间传输 8，16 或 32 位数据；允许外加的存储器周期，用来以更快的速度传输数据块；还允许其他的设备控制外部总线等等。本例中设计的模型只包含一些基本的信号用来控制总线的读写操作。本单元属于 486 微处理器的一部分，它提供了 486 外部总线和微处理器其他部分的一个接口。如图 35.1 是外部总线的接口图。

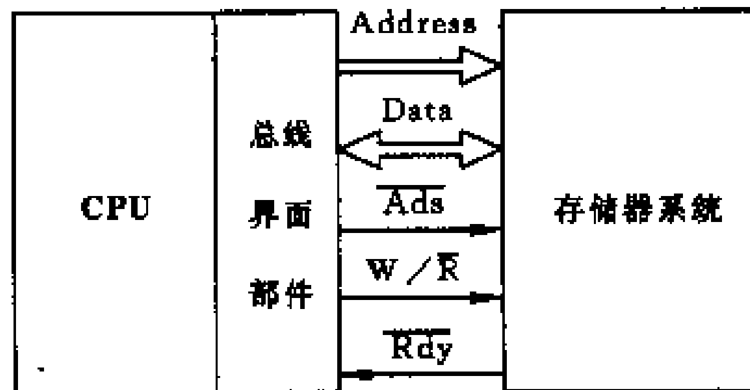


图 35.1　外部总线接口

内部总线是介于 CPU 与 486 总线接口单元的。486 总线接口单元的功能是接收 CPU 发出的控制信号和数据信号，并且把外部设备的数据信号和状态信号返回给 CPU。当 br=1(Bus Request)和 wr(Write)=1 时，CPU 处于写周期；而当 br=1，wr=0 时，CPU 处于读周期。读写周期完成时，总线中介单元将返回信号 done=1 给 CPU。总之，有了 486 总线接口单元 CPU 便可以与外部设备通信。图 35.2 是内部总线的接口图。

486 总线接口单元包含一个状态机用来控制总线的操作。如图 35.3 中使用 SM 图的形式给出状态机的流程。

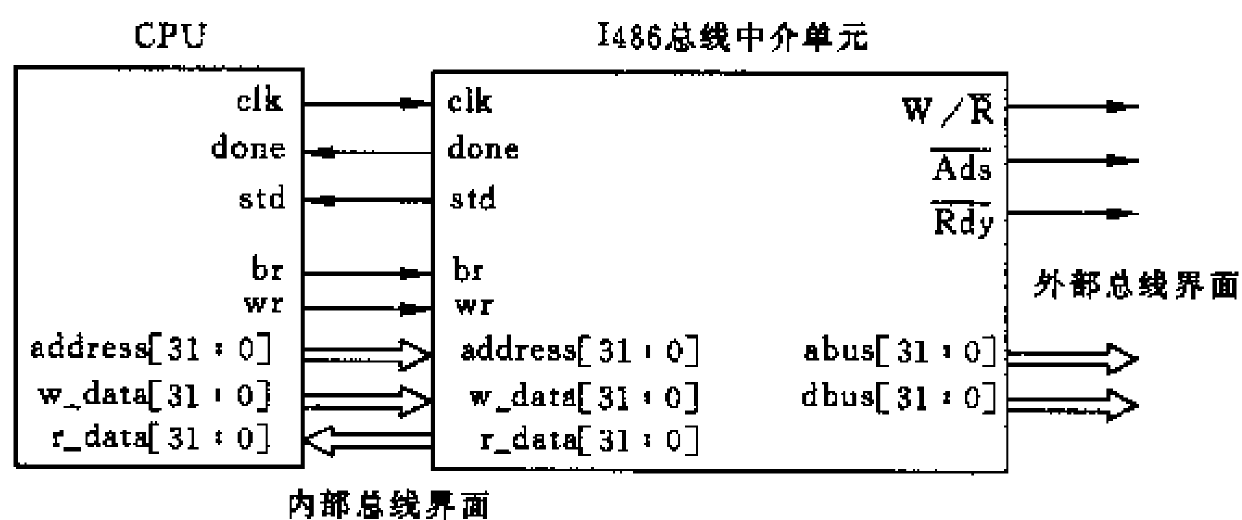


图 35.2 内部总线接口图

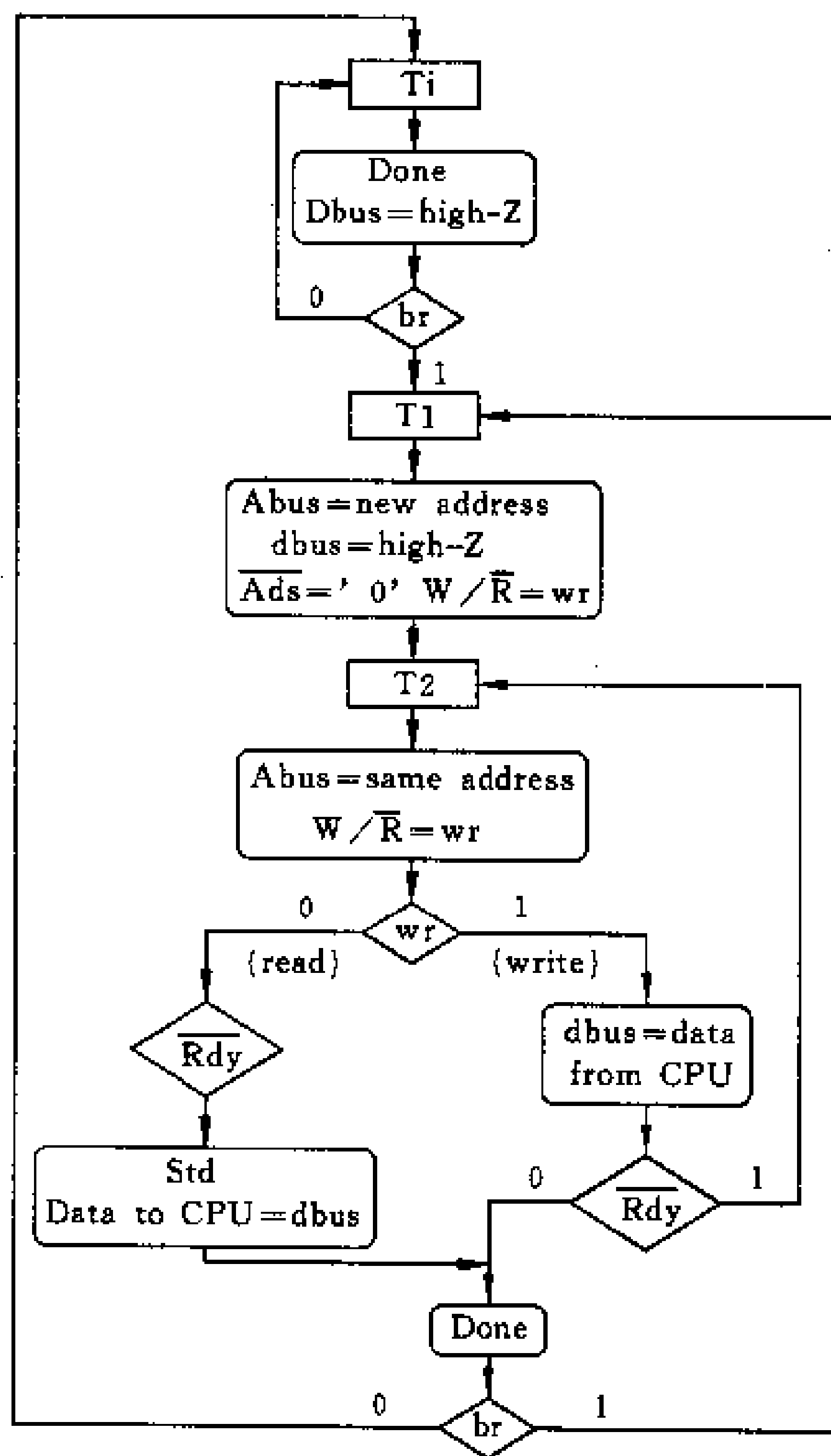


图 35.3 总线接口单元流程图

2. 时序设计

因为 486 处理器总线是同步执行的，所以所有的时序描述都是基于时钟的上升沿。同时要考虑信号在不同部件中的内部时延和传输时延，所以信号都要有一定的信号建立和信号保持时间。如：总线接口单元的 $\overline{Rdy}$ 信号在时钟上升沿到来前保持一段建立时间 t16 并在时钟上升沿到来后保持一段时间 t17。在读数据时，数据到达 CPU 也要满足建立和保持时间。数据建立时间为 t22，保持时间 t23。本系统中使用的总线 I486_bus 的时序是基于 I486DX 50 型号的。为了与这种总线相匹配，前面的 RAM 的时间特性将基于 43258A-25 型的静态 CMOS RAM。下面将仔细分析两者的时序关系，以便总线与存储器之间能够协调工作。图 35.4 画出了从 RAM 读取数据时的有关信号传输的路径。基于本图可获得系统读周期的时间特性。

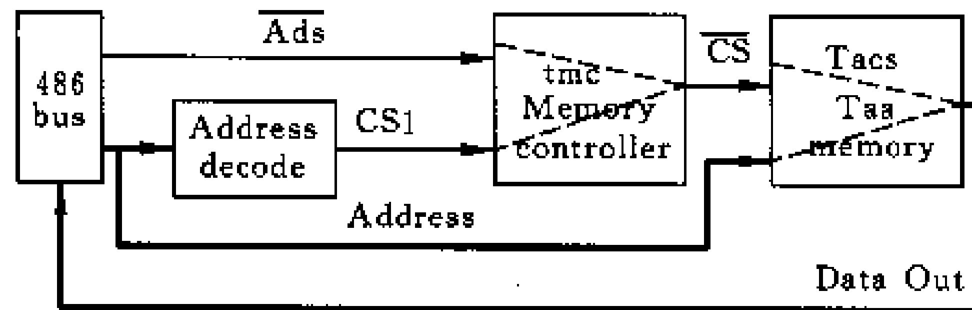


图 35.4 信号通路时延图

为了保证时序正确，要找到图 35.4 中信号传输最长延迟的路径，即图中的关键路径。由于片选信号和地址信号通过存储器的时延相同（Taa=Tacs），而且片选信号生成前要比地址信号多通过一个部件（Address decode），所以片选信号路径延迟要比 Address 的路径延迟长。而信号 Ads 和地址信号在同一时间稳定下来。可以看出，最长的延迟路径是包含部件（Address decode）的路径。取 T1 周期的开始为起点，最长路径的一些延迟如下：

① 从时钟高到地址信号合法的时间=t6_max=12ns，即 486 总线的信号稳定的延迟时间。

② Address decode 的延迟时间=tdecode=5ns，本延迟为估计值，自己假定的。

③ 存储器控制器的延迟时间=tmc=5ns，本延迟为估计值，自己假定的。

④ 存储器的访问时间=tacs=25ns，可从 43258A-25 CMOS RAM 的时序特性得到。

⑤ 数据建立时间=t22=5ns，486 总线的数据建立时间。

从以上的数据可得到读周期中最长的时间延迟为 52ns。如果以最大的时钟频率 50MHz 工作时，时钟周期则为 20ns，所以需要 3 个时钟周期来完成一个读周期。同理读者可以分析得出写周期的时间延迟。

3. VHDL 描述及语法分析

```
library ieee;
```

```
use ieee.std_logic 1164.all;

entity i486_bus is
    --下面的时间特性基于 I486DX 50
            generic (
             constant t6_max      : time := 12 ns;
             constant t10_max     : time := 12 ns;
             constant t10_min     : time := 3  ns;
             constant t11_max     : time := 18 ns;
             constant t16_min     : time := 5  ns;
             constant t17_min     : time := 3  ns;
             constant t22_min     : time := 5  ns;
             constant t23_min     : time := 3  ns);
          --内部总线接口
          port (
          abus : out bit_vector (31 downto 0);
          dbus : inout std_logic_vector (31 downto 0);
          w_rb, ads_b : out bit := ' 1' ;
          rdy_b, clk : in bit;
          --外部总线接门
          address, w_data : in bit_vector(31 downto 0);
          r_data : out bit_vector(31 downto 0);
          wr, br : in bit;
          std, done : out bit);
end i486_bus;
architecture simple_486_bus of i486_bus is
type state_t is (ti, t1, t2);
signal state, next_state : state_t := ti;
begin
  --下面的进程用于产生控制信号和处理器在读写操作时的地址信号
  --进程根据操作的类型给总线设定相应状态
  --在读写操作时 done 信号为低，当总线就绪接受下 --请求时 done 信号为高
  comb_logic : process
   begin
      std <= ' 0' ;
      case (state) is
           when ti =>
                done <=' 1' ;
                if (br=' 1' ) then next_state <= t1;
                else next state <= ti;
                end if;
                dbus <= transport (others =>' Z' ) after t10_min;
```

```
            when t1 =>
                done <=' 0' ;
                ads_b  <= transport ' 0'   after t6_max;
                w_rb  <= transport wr after t6_max;
                abus <= transport address after t6_max;
                dbus <= transport (others =>' Z' ) after t10_min;
                next_state <=t2;
            when t2 =>
                ads_b <= transport ' 1'   after t6_max;
                --读操作
                if (wr=' 0' ) then
                    if (rdy_b =' 0' ) then
                        r_data <= to_bitvector(dbus);
                        std <= ' 1' ;
                        done <=' 1' ;
                        if (br = ' 0' ) then next_state <= ti;
                        else next_state <= t1;
                        end if;
                    else next_state <= t2;
                    end if;
                 --写操作
                 else
                    dbus  <=  transport to_stdlogicvector(w_data)
                             after t10_max;
                    if (rdy_b= ' 0' ) then
                        done <=' 1' ;
                        if (br = ' 0' ) then next_state <= ti;
                        else next_state <=t1;
                        end if;
                    else next_state <=t2;
                    end if;
                end if;
        end case;
        wait on state,rdy_b,br,dbus;
end process comb_logic;
--本进程用于在每一个时钟上升沿更新当前状态
seq_logic : process (clk)
begin
   if (clk=' 1' ) then state <= next_state; end if;
end process seq_logic;
--本进程用于检查所有的建立和保持时间是否与所有的输入控制信号相匹配
wave_check : process (clk, dbus, rdy_b)
```

```
  variable clk_last_rise : time := 0 ns;
begin
   if (now /= 0 ns) then
      --检查信号建立时间
      if  clk' event and clk =' 1'  then
         --下面的 assert 语句中假定 RDY 的建立时间
         --大于或等于 data 的建立时间
         assert (rdy_b /= ' 0' ) or (wr /=' 0' ) or
               (now - dbus' last_event >= t22_min)
         report "i486 bus : data setup too short"
         severity warning;
         assert (rdy_b' last_event >= t16_min)
         report "i486 bus : rdy setup too short"
         severity warning;
      end if;
      --检查保持时间
      if (dbus' event) then
         --下面的 assert 语句中假定 RDY 的保持时间
         --大于或等于 data 的保持时间
         assert (rdy_b /= ' 0' ) or (wr /=' 0' ) or
               (now - clk_last_rise >= t23_min)
         report "i486 bus : data hold too short"
         severity warning;
      end if;
      if (rdy_b' event) then
       assert (now - clk_last_rise >= t17_min)
       report "i486 bus : rdy signal hold too short"
       severity warning;
      end if;
   end if;
end process wave_check;
end simple_486_bus;
```

4. 存储器控制器电路

存储器控制器的框图如图 35.5 所示，存储器控制器带有 4 个 1 位的输入端口，3 个

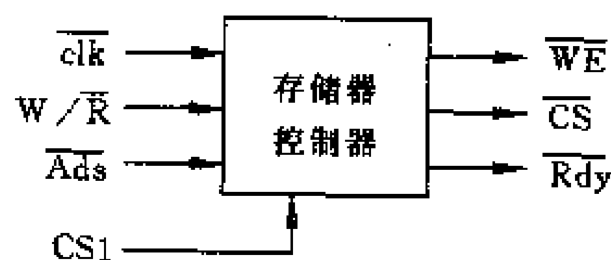


图 35.5　存储器控制器框图

1 位的输出端口，当 CS1=1 时，存储器控制器被选中，接受总线传来的控制信号并根据总线状态产生对 RAM 的控制信号。$\overline{WE}$，$\overline{CS}$ 为对存储器的读写信号和片选信号。当 $\overline{WE}$=0，$\overline{CS}$=0 时为 RAM 写操作；当 $\overline{WE}$=1，$\overline{CS}$=0 时为 RAM 读操作。CS1 信号由 Address decoder 产生，而且只有当地址信号在 0~32767 之间时 CS1 才会为 1，存储器控制器才会被选中开始工作。$\overline{Rdy}$=0 时，代表 RAM 的操作已经完成，如果 RAM 为读操作，则 dbus 上的数据可传给 CPU；如果为写操作，则表明 RAM 写入数据完成，CPU 可以进行下一项操作。存储器控制器 SM 流程如图 35.6 所示。

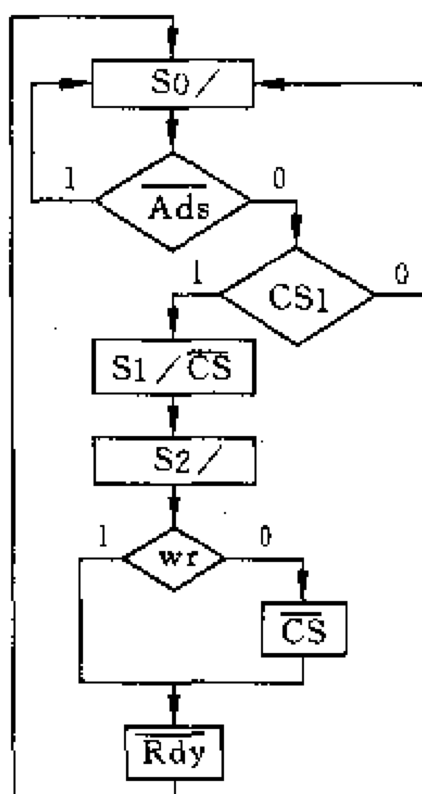


图 35.6　存储器控制器流程图

5. VHDL 描述及语法分析

```
library ieee;
use icee.std_logic_1164.all;
entity  memory_control  is
   port (  clk, w_rb, ads_b, cs1  :  in bit;
           rdy_b, we_b, cs_b  :  out bit := ' 1' );
end memory_control;

architecture  behavel  of  memory_control  is
  constant delay : time := 5 ns;
  signal state, nextstate: integer range 0 to 2;
  signal new_we_b, new_cs_b,  new_rdy_b : bit := ' 1' ;
begin
  process (state, ads_b, w_rb, cs1)
```

```
    begin
        new_cs_b <='1'; new_rdy_b <='1'; new_we_b <='1';
        case state is
                when 0 =>
                        if ads_b ='0' and cs1='1'
                        then nextstate <=1;
                        else nextstate <=0;
                        end if;
                when 1 => new_cs_b <='0';nextstate <=2;
                when 2 =>
                        if w_rb ='1' then new_cs_b <='1';
                        else new_cs_b <='0';
                        end if;
                        new_rdy_b <='0';
                        nextstate <=0;
         end case;
    end process;
  process(clk)
  begin
     if clk='1' then state <= nextstate; end if;
  end process;
  we_b <=not w_rb after delay;
  cs_b <=new_cs_b after delay;
  rdy_b <=new_rdy_b after delay;
  end behavel;
```

6. 测试台设计

电路工作图在 I486 总线单元中已经给出。本测试台是带有端口的测试台，用于对整个系统的正确性进行验证。即对某个地址写入一个数据，再对同一地址进行读操作，两个数据应该相等，于此同时不应该存在总线使用冲突。在一个总线周期完成时，测试台将为下一个总线周期提供数据。为了方便，本例中数据提供采用文件的方式。文件中的字段有读写信号、地址信号以及写入数据后应读出的数据。每一个字段用一个或多个空格分开，数据采用整型格式，因为整型的字段是压缩的而且使用标准的 TEXTIO 函数容易读取。具体文件的格式见表 35.1。

测试台是根据总线周期工作的。总线接口单元在 T2 总线周期结束前对号敏感，测试台则应该在结束 T2 周期的时钟上升沿前检测 Done 信号，判断存储器的操作是否已经完成。为了实现这一点，测试台中定义了一个自己的时钟（testclk）信号，它与总线时钟

表 35.1 文件结构

br	wr	addr	data	总线状态说明
0	1	7	23	空闲
1	1	139	4863	写
1	1	255	19283	写
1	0	139	4863	读
1	0	255	19283	读
0	0	59	743	空闲
1	0	139	4863	读
1	1	139	895	写
1	1	139	895	读
1	1	2483	0	总线挂起

(clk) 信号相同，只是稍有延迟 (1ns)。这样做的目的是，保证测试台可以在 testclk 信号上升沿后及在 clk 信号上升沿前检测 Done 信号。

7. 语法现象

文件对象类型实际上是一个变量对象类型的子集，变量对象能用变量赋值语句赋值，而文件对象不能被赋值。文件对象只能由规定的过程和函数读出、写入和检查文件结束。

文件类型说明：即指定文件类型名称和文件的基本类型。形式如下：

```
type  文件类型名称  is  file  of  基本类型名
```

例：**type** int_file **is** file **of** integer;

文件对象说明：即指出文件对象名、文件打开方式、文件路径名。文件打开方式有 IN 和 OUT 两种。IN 为读方式，OUT 为写方式。形式如下：

```
file  文件对象名：文件类型名  is  打开方式  路径名；
```

例：file myfile : int_file **is** in "/disk/BIT903/work";

8. VHDL 描述方法及语法分析

```
library ieee;
use ieee.std_logic_1164.all;
use work.bit_pack.all;
use std.textio. all;
entity tester is
```

```
  port ( address,w_data:out bit_vector(31 downto 0);
         r_data:in bit_vector(31 downto 0):=(others =>' 0' );
         clk,wr,br:out bit;
         std, done:in bit:= ' 0' );
end tester;
architecture test1 of tester is
   --设定时钟周期为 20 ns
   constant half_period: time:=10 ns;
   signal testclk :bit := ' 1' ;
begin
  testclk<=not testclk after half_period;
  clk<=testclk after 1 ns;
  read_test_file :process(testclk)
     --已读的方式打开 test 类型文件"test2.dat"
     --文件名为 test_file
     file test_file:text is in "test2.dat";
     variable buff:line;
     variable dataint,addrint:integer;
     variable new_wr,new_br :bit;
  begin
     --等待直到总线处理完成且信号稳定
     if testclk=' 1'  and done=' 1'  then
        if std=' 1'  then
           assert dataint =vec2int(r_data)
           report "read data doesn' t match data file!"
           severity error;
        end if;
        if not endfile(test_file) then
             readline(test_file,buff);
             read(buff,new_br);
             read(buff,new_wr);
             read(buff,addrint);
             read(buff,dataint);
             br<=new_br;
             wr<=new_wr;
             address<=int2vec(addrint,32);
             if new_wr=' 1'  and new_br=' 1'  then
                --写操作
                w_data<=int2vec(dataint,32);
             else
```

```
                --读操作
                w_data<=(others=>'0');
            end if;
        end if;
    end if;
  end process read_test_file;
end test1;
```

9. 系统组装

系统组装中用到了测试台、486 总线接口单元、存储器控制器和静态 RAM 这 4 种组件。程序中使用了 generate 语句生成了 4 个静态 RAM，并使用类属表给出了静态 RAM 的时序特性，使之符合 43258A-25 型的 CMOS RAM。本系统中 RAM 只使用了 8 位地址线，而且部件（Address decoder）用了一个单独的并行语句实现。

本例在 Talent 系统上通过编译、模拟，证明此设计是正确的。

10. VHDL 描述及语法分析

```
--本描述是将前述各部件组装成一个完整的系统
library ieee;
use ieee.std_logic_1164.all;
use work.all;
entity i486_bus_sys is
end i486_bus_sys;
architecture bus_sys_bhv of i486_bus_sys is
component i486_bus
   port(
        abus : out bit_vector(31 downto 0);
        dbus : inout std_logic_vector(31 downto 0);
        w_rb,ads_b : out bit;
        rdy_b,clk : in bit:='0';
        address,w_data : in bit_vector(31 downto 0):=
                      (others=>'0');
        r_data : out bit_vector(31 downto 0);
        wr,br : in bit:='0';
        std,done : out bit);
end component;
component static_ram
   generic(constant taa, tacs, tclz, tchz, toh, twc, taw, twp,
          twhz, tdw,tdh,tow:time);
   port ( cs_b,we_b,oe_b:in bit;
```

```
            address : in bit_vector(7 downto 0);
            data :inout std_logic_vector(7 downto 0));
end component;
component memory_control
   port(
         clk,w_rb,ads_b,csl:in bit;
         rdy_b,we_b,cs_b:out bit);
end component;
component tester
   port(
         address,w_data:out bit_vector(31 downto 0);
         r_data:in bit_vector(31 downto 0);
         clk,wr,br:out bit;
         std,done:in bit);
end component;
constant decode_delay : time :=5 ns;
constant addr_decode :bit_vector(31 downto 8):=(others->' 0' );
signal csl:bit:=' 1' ;
signal address,w_data,r_data:bit_vector(31 downto 0);
signal clk,wr,br,std,done:bit;
signal w_rb,ads_b,rdy_b: bit;
signal abus:bit vector(31 downto 0);
signal dbus:std_logic_vector(31 downto 0);
signal cs_b,we_b:bit;
--信号 oe_b 在系统中总设定为低
signal oe_b :bit :=' 0' ;
begin
 bus1:i486_bus port map
             ( abus,dbus,w_rb,ads_b,rdy_b,clk,address,w_data,
              r_data,wr,br,std,done);
control1:memory_control port map
             (clk,w_rb,ads_b,csl,rdy_b,we_b,cs_b);
ram32 : for i in 3 downto 0 generate
   ram :static_ram
         generic map (25 ns,25 ns,3 ns,3 ns, 3 ns,25 ns,15 ns,
                      15 ns,10 ns,12 ns,0 ns,0 ns)
         port map (cs_b,we_b,oe_b,abus(7 downto 0),
                      dbus(8*i+7 downto 8*i));
 end generate ram32;
test:tester port map
```

```
            (address, w_data, r_data, clk, wr, br, std, done);
--判断地址信号是否在 0 到 32,768 之间，因为 RAM 地址信号只有 15 位
cs1 <='1' after decode_delay when (abus(31 downto 0)
                =addr_decode)
   else '0' after decode_delay;
end bus_sys_bhv;
--系统配置文件
configuration conf of i486_bus_sys is
for bus_sys_bhv
for ram32
for ram : static_ram use entity work.static_ram(sram);
end for;
end for;
for bus1 : i486_bus use entity work.i486_bus(simple_486_bus);
end for;
for control1 : memory_control use entity
               work.memory_control(behave1);
end for;
for test : tester use entity work.tester(test1);
end for;
end for;
end conf;
```

11. 附录

表 35.2　43258A-25 型 CMOS RAM 的时间参数表

参数	参数名称	最小值	最大值
读周期	trc	25	—
地址到达时间	taa	—	25
芯片片选到达时间	tacs	—	25
芯片片选到输出为低	tclz	3	—
输出使能到输出合法	toe	—	12
输出使能到输出为低	tolz	0	—
片选撤消到芯片输出为高	tchz	3*	10
芯片使无效到输出为高	tohz	3*	10
地址改变为新值前的保持时间	toh	3	—

续表

参数	参数名称	最小值	最大值
写周期	twc	25	—
片选择到写结束时间	tcw	15	—
地址信号合法到写结束时间	taw	15	—
地址信号建立时间	tas	0	—
写脉冲宽度	twp	15	—
写恢复时间	twr	0	—
写使能到输出为高	twhz	3*	10
数据合法到写结束	twd	12	—
数据保持到写结束	tdh	0	—
输出使能到写结束	tow	0	—

注：表中带*号的参数是估计值。

（源描述文件名：35_486_sys.vhd
35_486_bus.vhd
35_bus_test.vhd
35_ram_controller.vhd
35_bit_pack.vhd）

第 36 例　基于多路器的数据通道

李　杰

1. 电路系统工作原理

利用基于多路器的数据通道模型，我们可以仅仅利用一个多路器在外部端口和寄存器、功能单元之间传输数据。虽然这里仅仅只能够传输一些比较简单的数据类型，但是传输的执行却更加简单，因为只需要较少的功能单元来实现传输，这就意味着更少的控制信号以及更小的传输延迟。

在本例中将用基于多路器的数据通道来实现求最大公约数（GCD）的操作，如图 36.1 所示。其中共用到了 5 个多路器，对比前面基于总线的实现，可以看出其结构更加简单。

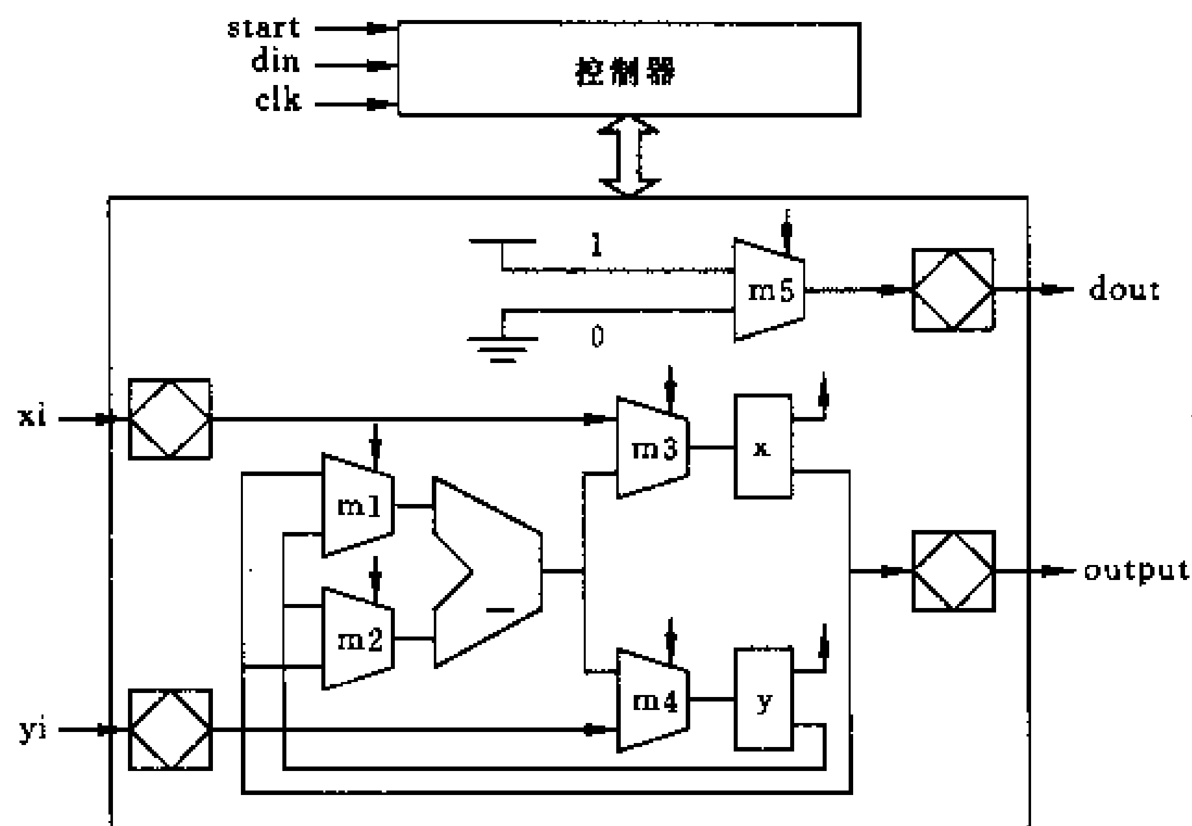


图 36.1　用基于数据通道的数据通道实现 GCD 的示意图

2. VHDL 语言描述及语法分析

下面是用基于多路器的数据通道实现 GCD 的行为描述。

```
entity gcd is
port(start: in bit;
```

```
      clk   : in bit;
      din   : in bit;
      xi,yi: in integer;
      dout  : out bit;
      output:out integer);
end gcd;

architecture behavior of gcd is
begin
  process
     variable x,y:integer;
     begin
       wait until((start='1') and (clk='1' and clk'event));
--以下是计算循环，计算两个数的最大公约数
       calculation:loop
       wait until((din='1') and (clk='1' and clk'event));
       dout <= '0';
       x := xi;
       y := yi;
       while(x/=y)loop
       if(x < y)
            then y := y-x;
            else x := x-y;
      end if;
    end loop;
     wait until((din='0')and(clk='1' and clk'event));
     dout <= '1';
     output <= x;
   end loop;
  end process;
end behavior;
```

以下是测试台的代码：

```
--这是gcd测试台的代码
library ieee;
use ieee.std_logic_1164.all;

entity gcd_tester is
end gcd_tester;

architecture test of gcd_tester is
signal start:bit :='0';
```

```
signal clk:bit :=' 0' ;
signal din:bit :=' 0' ;
signal xi:integer :=4;
signal yi:integer :=8;
signal dout:bit :=' 0' ;
signal output:integer :=0;
--输入输出信号
component gcd
 port (start:in bit;
        clk:in bit;
        din:in bit;
        xi,yi:in integer;
        dout:out bit;
        output:out integer
      );
end component;
begin
 gcd1:gcd port map (start,clk,din,xi,yi,dout,output);
 process --本进程产生时钟信号
 begin
      clk <= not clk;
 wait for 10ns;
 end process;
 process --本进程产生激励信号
 begin
 wait for 50 ns;
 start <= ' 1' ;
 din <= ' 1'  ;
 wait for 100 ns;
 start <= ' 0' ;
 din <= ' 0' ;
 wait for 100 ns;
 xi <=9;
 yi <=3;
 start <= ' 1' ;
 din <= ' 1'  ;
 wait for 100 ns;
 start <= ' 0' ;
 din <= ' 0' ;
 wait for 100 ns;
end process;
end test;
```

如图 36.2 的波形图，两个输入分别是 9 和 3，经过计算后，在 800ns 的时钟上升沿 dout 信号指示结果已经计算出来，同时在 output 上输出两者的最大公约数 3。

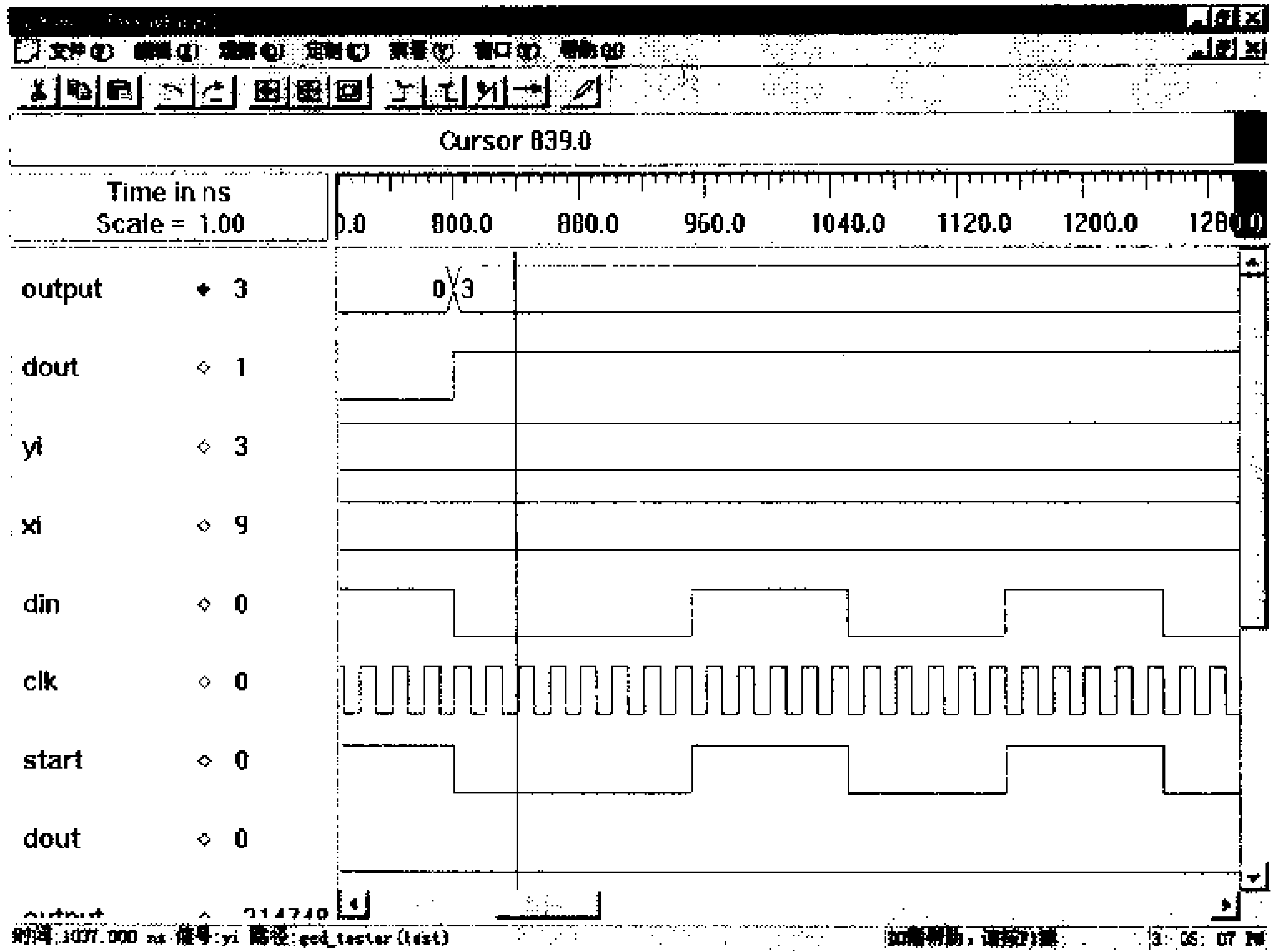


图 36.2　模拟的波形结果

(源描述文件名: 36_gcd.vhd
36_test.vhd)

第 37 例　四值逻辑函数

袁　媛

本例也是一个测试的示例，主要说明几个问题：

四值逻辑及其分辨；

- 函数；
- 取位向量的高几位或低几位及可能出现的错误。

1. 源描述

```
entity Test_105 is
end Test_105;        --空的实体，用于测试

architecture Behave_1 of Test_105 is
  type Logic4 is ('x', '0', '1', 'z');
  -- 四值逻辑，'x' 为不定状态，'0' 为低电平，'1' 为高电平，'z' 为高阻状态
  type Logic4_Vector is array (natural range <> ) of Logic4;
  -- 一维数组，值为四值逻辑中的状态
  type Logic4_Table is array ( Logic4,Logic4) of Logic4;
  -- 二维数组，值为四值逻辑中的状态
  constant Sample_Tbl : Logic4_Table := (('x','x','1','x'),
                                         ('x','0','1','z'),
                                         ('1','1','1','1'),
                                         ('x','z','1','z'));
  -- 常量，二维数组，相当于一个表格
  function Manipulate_Vectors(L_V,R_V : Logic4_Vector)
            return Logic4_Vector is
    alias L_V_Alias : Logic4_Vector(L_V'LENGTH downto 1) is L_V;
    alias R_V_Alias : Logic4_Vector(R_V'LENGTH downto 1) is R_V;
    variable Result : Logic4_Vector(L_V'LENGTH downto 1);
  begin
    for I in Result'RANGE loop
     Result(I) := Sample_Tbl(L_V_Alias(I), R_V_Alias(I));
     -- 根据参数得出二维表中相应的值
    end loop;
    return Result;
  end Manipulate_Vectors;
```

```
        signal Vector_A : Logic4_Vector(0 to 7);
        signal Vector_B : Logic4_Vector(0 to 7);
        signal Vector_C : Logic4_Vector(0 to 7);
      -- 定义三个信号，均为四值逻辑位向量
   begin
      Funct_Slice_Test:
      process
        -- variable Vector_A : Logic4_Vector(0 to 7);
        -- variable Vector_B : Logic4_Vector(0 to 7);
        -- variable Vector_C : Logic4_Vector(0 to 7);
     begin
        Vector_A <= "xxx1z1x1";    -- 信号并发赋值语句
        Vector_B <= "111100x0";
        Vector_C <= Manipulate_Vectors(Vector_A, Vector_B);
        wait for 50 ns;
      end process Funct_Slice_Test;

      Finish:
      process
      begin
           wait for 400 ns;
           assert false
           report "--------End of Simulation--------"
           severity error;
      end process Finish;
      -- 400ns 后结束模拟

 end Behave_1;
```

如果语句

```
Vector_C <= Manipulate_Vectors(Vector_A, Vector_B);
```

写成如下的形式：

```
Vector_C <= Manipulate_Vectors(Vector_A, Vector_B)(4 downto 1);
```

则会导致一个运行时的错误，因为函数参数是按升序排序的，而返回值被说明为按降序排序。而如果写成如下形式：

```
Vector_C <= Manipulate_Vectors(Vector_A, Vector_B)(4 downto 0);
```

也会导致一个运行时的错误，因为元素 result(0)不存在。

2. 四值逻辑及其分辨

四值逻辑的四个值是('X', '0', '1', 'Z')，其中：'X'代表不定状态；'0'代表逻辑 0 或假；'1'代表逻辑 1 或真；'Z'代表高阻状态。

test_105 首先定义了 3 个子类型，logic4 是四值逻辑的枚举类型，logic4_vector 是四值逻辑的一维数组，而 logic4_table 则为二维数组，它的下标为四值逻辑的枚举类型，值也是四值逻辑的枚举类型。

四值逻辑的分辨实际上在定义的常量 sample_tbl 中已经体现出来，如表 37.1 所示。

表 37.1　四值逻辑的分辨

	'X'	'0'	'1'	'Z'
'X'	'X'	'X'	'1'	'X'
'0'	'X'	'0'	'1'	'Z'
'1'	'1'	'1'	'1'	'1'
'Z'	'X'	'Z'	'1'	'Z'

其中：若两驱动值（设一个为 A，一个为 B）都为'X'，则分辨值为'X'；若 A 或 B 中有一个为'1'，则无论另一个为什么，分辨值均为'1'；若 A='0'，B='Z'，则分辨值为'Z'，依此类推。

从表 37.1 中可以看到，'1'的强度最大，接下来依次为'X'，'Z'和'0'。

3. 函数

test_105 中定义的函数 Manipulate_Vectors 有两个参数 L_V 和 R_V，均为一维数组，类型为 logic4_Vector，返回值的类型为 logic4_Vector。

函数的写法由此也可见一斑，我们在第 10 例中对函数及别名已有详细的描述，请读者参考该例。

4. 取位向量的高几位、低几位及可能出现的错误

在 test_105 结构体的说明区，定义 3 个信号 Vector_A, Vector_B 及 Vector_C，前两个信号是 8 位位向量，而 Vectro_C 是 4 位位向量。

在 Funct_slice_Test 进程中，给 Vector_A 和 Vector_B 赋值，而对于 Vector_C，由语句

```
Vector_C <= Manipulate_Vectors(Vector_A,Vector_B)(8 downto 5);
```

赋值：这条语句是以 Vector_A 和 Vector_B 为参数调用函数 Manipulate_Vector，并取返回值的高 4 位赋给 Vector_C。也可以取低 4 位，但需要写成如下形式：

```
Vector_C <= Manipulate_Vectors(Vector_A, Vector_B)(4 downto 1);
```

因为在函数定义中要求位向量是按降序排列的。若写成以下形式，运行时会产生错误：

```
Vector_C<=Manipulate_Vectors(Vector_A, Vector_B)(3 downto 0);
```

因为在函数定义中，无论参数还是返回值都定义为 V'LENGTH **downto** 1，下界均为 1，不存在 result(0)。

若 A= “00000000”, B= “11110000”，得到 Vector_C= “11110000”。

若 A= “01010101”, B= “XXX1Z1X1”, 则 Vector_C= “X1X1Z1X1”。

（源描述文件名：37_test_105.vhd）

第38例　四值逻辑向量按位或运算

刁岚松

1. 电路系统的工作原理

本例旨在实现四值逻辑向量的按位或运算。定义四值逻辑运算表如表38.1。为保存该表，定义一个二维数组，每一位或运算都是通过从该表取值实现。例如计算'x'与'1'的或运算值，只需取四值逻辑运算表中第1行，第3列的值即可得到值为'1'。

表 38.1　四值逻辑运算表

	x	0	1	z
x	x	x	1	x
0	x	0	1	z
1	1	1	1	1
z	x	z	1	z

2. 电路的VHDL语言描述方法及语法分析

部分描述如下所示：

```
entity Test_28 is
end Test_28 ;

architecture Behave_1 of Test_28 is
  --定义四值逻辑
  type Logic4 is ('x', '0', '1', 'z');
  --定义四值逻辑向量
  type Logic4_Vector is array (natural range <>) of Logic4;
  --定义四值逻辑运算表
  type Logic4_Table is array (Logic4,Logic4) of Logic4;
  constant Or_Table : Logic4_Table := (('x', 'x', '1', 'x'),
                                       ('x', '0', '1', 'z'),
                                       ('1', '1', '1', '1'),
                                       ('x', 'z', '1', 'z'));
  function "or" (L_V,R_V : Logic4_Vector)
                return Logic4_Vector is
    variable Result : Logic4_Vector(1 to L_V'LENGTH);
```

```
  --定义四值逻辑或运算函数

begin
--保证两个四值逻辑向量长度一致，如果长度不一致，
--则报告出错信息，并停止模拟。
--L_V'LENGTH 是向量 L_V 的长度属性，返回向量 L_V 的长度。
  assert L_V' LENGTH = R_V' LENGTH
     report "LENGTH mismatch of inputs"
     severity error;

  --for 循环语句，Result' RANGE 是向量 Result 的下标范围的属性，
  --返回 Result 向量的下标范围。
 for I in Result' RANGE loop
      Result(I):=Or_Table(L_V(I-Result' LOW+L_V' LOW),
                          R_V(I- Result' LOW+R_V' LOW));
 end loop;
  return Result;--返回函数值
end "or";

begin
 Or_Range_Test:
 process
   variable Vector_0_8   : Logic4_Vector(0 to 8);
   variable Vector_3_11  : Logic4_Vector(3 to 11);
   variable Vector_15_23 : Logic4_Vector(15 to 23);
 begin
   Vector_0_8  :="000011101";
   Vector_3_11 :="111100000";
   Vector_15_23(18 to 20) := Vector_0_8(0 to 2) or
                             Vector_3_11(8 to 10);
   wait for 50 ns;
 end process Or_Range_Test;

 --终止进程，用于终止模拟的执行
 proc_end:
 process
 begin
   wait for 100 ns;
   assert false
      report "end"
      severity error;
```

```
  end process proc_end;
end Behave_1;
```

语法分析：

VHDL 中的某些项目类可以具有属性。比如：信号名'EVENT:函数类属性。如果在当前模拟周期内该信号发生了某个事件(信号值发生变化)，则返回 True；否则，返回 False。

3. 模拟测试向量的选择及模拟结果分析

当 Vector_0_8 为 "000011101"，并且 Vector_3_11 为"111100000"时，Vector_15_23 的值为"xxx000xxx"。

这样取值的情况下，语句

```
Vector_15_23(18 to 20) := Vector_0_8(0 to 2) or
                          Vector_3_11(8 to 10);
```

执行时，取了向量 Vector_0_8 的第 0 至第 2 位的值和 Vector_3_11 的第 8 至第 10 位的值，它们分别为"000"和"000"。根据运算表，Or_Table('0','0')='0'，所以执行函数 **or** 之后，返回的值为"000"，将这个值返回给向量 Vector_15_23 的第 18 至第 20 位，该向量的其他位没有说明，取缺省值'x'。所以，最终向量 Vector_15_23 的值为"xxx000xxx"。

（源描述文件名：38_test_28.vhd）

第 39 例　生成语句描述规则结构

袁　媛

生成语句给设计中的循环部分或条件部分的确立提供了一种机制，在结构描述中可以用它来描述规则的结构，例如，元件例示是生成语句的一个重要的而且是最常用的一个特例。本例讨论生成语句在元件例示中的作用与用法。

1. 电路系统工作原理

电路系统的示意图如图 39.1 所示。

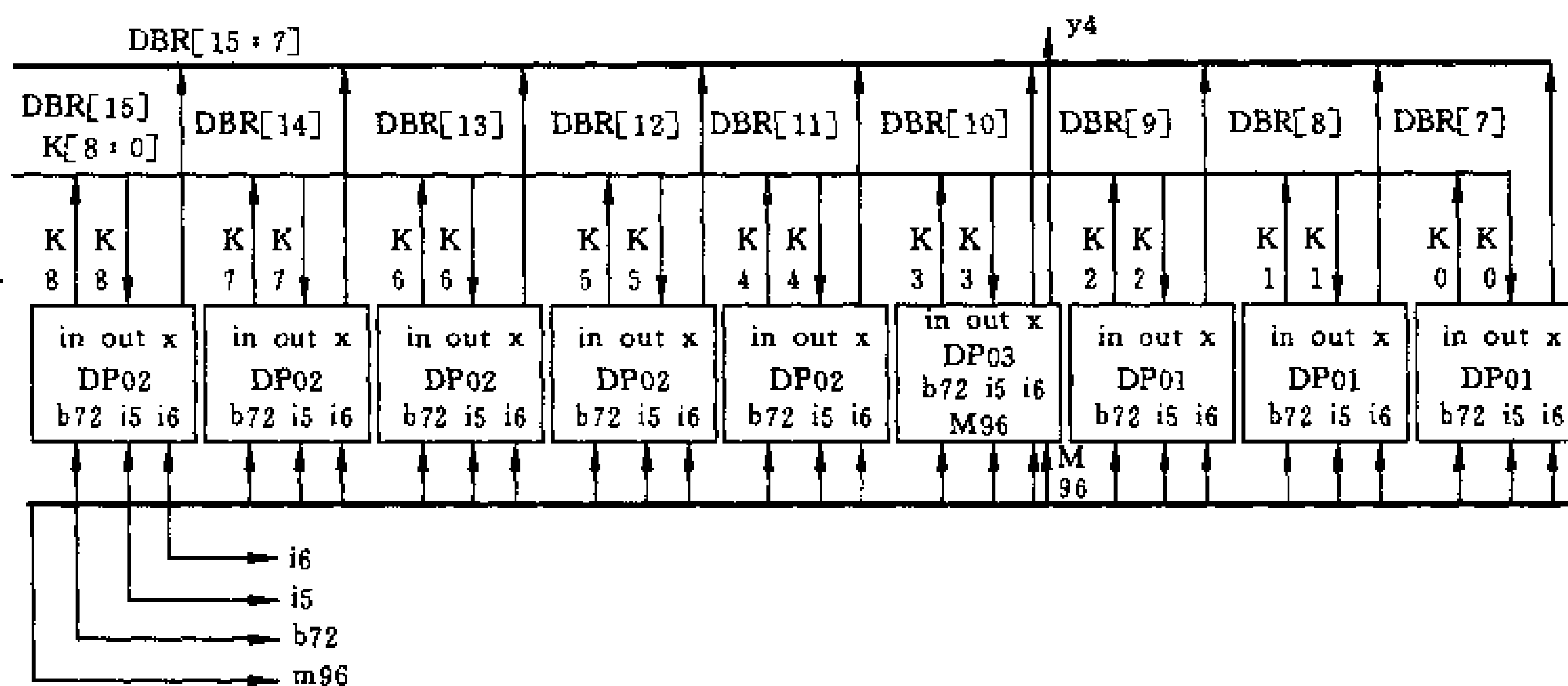


图 39.1　电路系统示意图

该电路系统中，DBR[15:7]是一条数据总线，电路系统的输出都通过这条总线，而 K[8:0]是另外一条数据总线，其类型既可以输入又可以输出，端口 i5，i6，b72 和 m96 则是控制端口。

从图 39.1 中，并不能知道此电路系统的功能到底是什么，由于它是由底层模块 DP01，DP02 及 DP03 搭建而成的，所以只有知道这些底层模块完成什么功能，才能知道此电路系统的功能。本例主要是说明在结构描述中，进行元件例示时生成语句的作用，并没有将底层模块的功能具体写出来，在此，只知道它们的功能是将数据总线 K[8: 0]分别输入到底层模块 DP01，DP02 及 DP03 中，在控制端口 i5，i6，b72 及 m96 的控制下，经过一定的延时，将 K[8:0]重新输入到数据总线 K[8:0]上，并且将 K[8:0]中的数据反相输

出到数据总线 DBR[15:7]上。

2. VHDL 语言描述方法及语法分析

(1) 生成语句的格式

生成语句的格式如下：

```
生成标号：生成方案 gengerate
    { 生成语句 }
end generate [ 生成标号 ]
```

生成方案有两种：**for** 和 **if**。生成方案 **for** 用于描述重复模式，而生成方案 **if** 通常用于描述一个结构中的例外情形。

该电路系统设计实体 wst0dp 总共连接了 9 个底层模块，这 9 个模块的连接方式相同。从示意图中可以看出，wst0dp 是一个规则结构，所以在生成标号为 LINE 的生成语句中采用 **for** 生成方案；但是这 9 个模块中有 5 个模块是 wst0dp02，1 个是 wst0dp03，其余的都是 wst0dp01，即连接的模块并不完全一样，所以在生成标号为 LINE9_5, LINE4 及 LINE3_0 的生成语句中采用的是 **if** 生成方案。

(2) 生成语句的用法

我们取源描述的一小段来说明如何使用生成语句提供建立重复结构的能力。源描述段截取如下：

```
LINE: for I in 8 downto 0 generate
    ------------------------------------------------------------
    LINE9_5: if I < 9 and I > 3 generate
        Udp02: wst0dp02
               port map( b72, i5, i6, k(I), k(I), DBR(I+7));
               s_DBR(I+7)  <= DBR(I+7);
end generate;
```

语句

```
LINE: for I in 8 downto 0 generate
```

说明要对 9 个模块进行元件例示，因为是重复结构，所以采用 **for** 循环统一处理。当 I < 9 且 I > 3 时，采用的模块是 DP02，所以用 **if** 生成方案进行生成：

```
LINE9_5: if I < 9 and I > 3 generate
```

接下来进行元件例示，循环变量还同时控制着端口名称，在 I 等于 8 时，输入端口 K(8)与信号 in 相连，输出端口 K(8)与信号 out 相连，而输出端口 DBR(8+7)与信号 x 相连，这样就将底层模块与上层数据总线及控制端口连接在一起。以此类推，整个电路的结构就这样搭建起来了。

有关结构描述的方法，请参考第 58 例。

生成语句还经常与类属结合在一起使用，这方面的内容，请参考第 40 例及第 41 例。

(3) 源描述

```
library IEEE;
use IEEE.std_logic_1164.all;
------------------------------------------
entity wst0dp is
------------------------------------------
     port(
               b53  : in std_logic;
               b72  : in std_logic;
               i5   : in std_logic;
               i6   : in std_logic;
               m96  : in std_logic;
               k    : in std_logic vector( 8 downto 0 );
               DRB  : in std_logic_vector( 15 downto 7 );
               ----------------------------------------
               w44  : out std_logic);
end wst0dp;

architecture struc of wst0dp is
     -----------------------------------
     component wst0dp01
     -----------------------------------
     port(
               b72  :  in std_logic;
               i5   :  in std_logic;
               i6   :  in std_logic;
               m96  :  in std_logic;
               inn  :  in std_logic;
               outt :  out std_logic;
               OB   :  out std_logic   );
end component wst0dp01;
-------------------------------------
component wst0dp02
```

```
-------------------------------------
    port(
            b72  : in std_logic;
            i5   : in std_logic;
            i6   : in std_logic;
            inn  : in std_logic;
            outt : out std_logic;
            x    : out std_logic );
end component;
-------------------------------------
component wst0dp03
-------------------------------------
    port(
            b72  : in std_logic;
            i5   : in std_logic;
            i6   : in std_logic;
            m96  : in std_logic;
            inn  : in std_logic;
            outt : out std_logic;
            0B   : out std_logic;
            y4   : out std_logic );
end component;

configuration config_wst0dp of wst0dp is
for struc
    for LINE
        for LINE9_5
        for  Udp02 : wst0dp02 use entity work.wst0dp02(FUNC);
        end for;
    end for;
    for LINE4
        for  Udp03 : wst0dp03 use entity work.wst0dp03(FUNC);
            end for;
    end for;
    for LINE3_1
        for  Udp01 : wst0dp01 use entity work.wst0dp01(FUNC);
        end for;
      end for;
    end for;
end for;
end config_wst0dp;
```

```
signal s_DRB : std_logic_vector( 15 downto 11 );
signal s_y4  : std_logic;

begin

LINE: for I in 8 downto 0 generate
         -----------------------------------------------------------
         LINE9_5: if i < 9 and i > 3 generate
              Udp02: wst0dp02
                        port map( b72, i5, i6, k(I), k(I), DRB(I+7));
                   s_DRB(I+7) <= DRB(I+7);
         end generate;
         -----------------------------------------------------------
         LINE4  : if i = 3 generate
              Udp03: wst0dp03
                        port map( b72, i5, i6, m96, k(I), k(I), y4, DRB(I+7));
                   s_y4 <= y4;
         end generate;
         -----------------------------------------------------------
         LINE3_0: if i < 3 and i >= 0 generate
              Udp01: wst0dp01
                        port map( b72, i5, i6, m96, k(I), k(I), DRB(I+7));
         end generate;
         -----------------------------------------------------------
end generate;

end struc;
```

（源描述文件名：39_wst0dp.vhd）

第40例　带类属的译码器描述

袁　媛

如果需要将信息传递给实体的具体元件，常用的一种方法就是类属，特别是用类属来规定端口的大小、实体中子元件的数目、实体的定时特性。类属是VHDL语言中一个非常重要、非常有用但也是比较难于理解的语法现象。下面通过一个带类属的译码器的示例来说明如何用类属规定端口的大小。在第46例中还将讲述类属的其他作用以及使用方法。

1. 电路系统工作原理

带类属的译码器的电路示意图如图40.1所示。其中N是类属值，在模拟的时候指定，模拟所需的测试平台请参见第41例。第58例对一个2-4译码器进行了详细的解释，请读者对照理解。

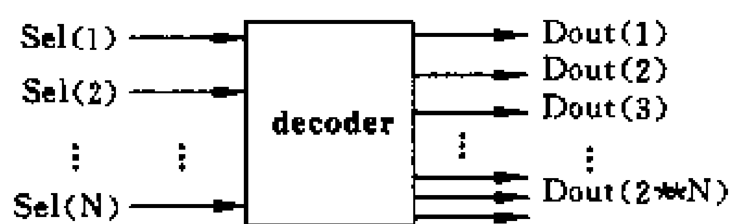


图40.1　带类属的译码器示意图

2. VHDL语言描述方法及语法分析

(1) 类属

译码器有许多种，同种类型的译码器也存在着端口数目不同的问题，如2-4译码器、3-8译码器、4-16译码器等等，而这些译码器除了端口数目不同以外，其他方面（如器件的连接等）都是一样的，如果对这些译码器分别进行描述，则显得繁琐，而类属可以解决这类问题。

首先，在实体说明中加入类属说明：

```
entity decoder is
    generic( N: Positive );
    port(     Sel  : Bit_vector ( 1 to N );
              Dout : out Bit_vector ( 1 to 2**N ));
```

```
end decoder;
```

类属的关键字是 **generic**，其后紧接着的是类属表，在类属表中定义端口数目的大小，如下述语句所示：

```
generic( N: Positive );
```

N 代表输入端口的数目，Positive 是 N 的类型，为正整数。定义了 N 以后，在实体说明中定义端口时，就可以用 N 来代替位向量的维数，输入端口如果为 N 个，按照译码器的功能，输出端口自然就是 2^N 个了。

综上所述定义了一个带类属的译码器。在其他涉及到信号及端口的地方，都要用到所定义的类属，以保证数目的正确性。

(2) 底层元件的行为描述

译码器是由反相器和二输入与门连接而成的，所以源描述首先对底层的两个元件进行行为描述，描述是一目了然的，不用多说。在对底层元件做了行为描述以后，在译码器的描述中可以直接使用，组装规定：

```
for Invert_select : Inverter use entity work.B_INV(FUNC)
                    port map(I1,O1);
for And_each_N_minus_with_Sel: And2 use entity work.B_AND2(FUNC)
                                    port map(I1,I2,O1);
for And_each_N_minus_with_Sel_bar: And2 use entity work.B_AND2(FUNC)
                                       port map(I1,I2,O1);
```

说明译码器的描述中所使用的底层元件来自何处，并且用元件例示语句将具体的元件端口装配起来。work 代表当前工作库，说明使用的底层元件就在当前的工作库中，而不是在别的库（如 IEEE 库）里。

(3) 组装

因为所描述的是带类属的译码器，所以在元件例示之前进行组装时，也要注意组装中的类属问题，带类属的组装语句如下：

```
for N_minus_1 : decoder use entity work.decoder(Generic_structure)
                generic map( N )
                port map(Sel,Dout);
```

(4) 类属与生成语句

类属经常与生成语句结合在一起使用，因为它们都与重复的、规则的结构有关。源

描述主体采用的是生成语句，在生成语句中的元件例示中采用类属传入端口的数目。

(5) 递归结构

本例在描述 $N-2^N$ 译码器时，采用的是一个比较特殊的递归结构，先用语句

```
N_minus_1 : decoder generic map (N-1);
                    port map ( Sel ( 1 to N-1 ),Temp );
```

生成一个 $(N-1)-(2^{N-1})$ 的译码器，然后在此基础上多添加生成 $N-2^N$ 译码器所需的与门即可，连接与门采用的也是生成语句，其中循环变量由类属值控制，如下所示：

```
For_each_output_from_N_minus_1:
for I in 1 to 2**(N-1) generate
   And_each_N_minus_1_with_Sel:
      And2 port map ( Temp(I),Sel(N),Dout ( 2*(I-1) +1 ));
   And_each_N_minus_1_with_Sel_bar:
      And2 port map ( Temp(I),Sel_bar,Dout ( 2*I ));
end generate;
```

3. 源描述

```
entity B_INV is
port(
          I1 : in bit;
          O1 : out bit
     );
end B_INV;

architecture FUNC of B_INV is                    -- 反相器的行为描述
     begin
          O1 <= not I1;
end FUNC;

entity B_AND2 is                                      -- 与门的行为描述
     port(
          I1 : in bit;
          I2 : in bit;
          O1 : out bit
          );
end B_AND2;

architecture FUNC of B_AND2 is
```

```
begin
    O1 <= I1 and I2;
end FUNC;

entity decoder is
  generic( N: Positive );
  port( Sel : Bit_vector ( 1 to N );
        Dout : out Bit_vector ( 1 to 2**N ));
end decoder;

architeture Generic_structure of decoder is
  signal Sel_bar : Bit;

  component And2
    port ( I1,I2 : Bit; O1 : out Bit );
  end component;

  component Inverter
    port ( I1 : Bit; O1 : out Bit );
  end component;

  component decoder
    generic ( N : Positive );
    port(
          Sel : Bit_vector ( 1 to N );
          Dout : out Bit_vector ( 1 to 2**N ));
  end component;

configuration config_decoder of decoder is
for Generic_structure
for Invert_select : Inverter use entity work.B_INV(FUNC)
                             port map(I1,O1);
end for;
for Recursive
   for N_minus_1 : decoder use entity work.decoder(Generic_structure)
                  generic map( N )
                  port map(Sel,Dout);
end for;
for For_each_output_from_N_minus_1
    for And_each_N_minus_with_Sel_bar:
                       And2 use entity work.B_AND2(FUNC)
                       port map(I1,I2,O1);
```

```
                end for;
            for And_each_N_minus_with_Sel:
                                          And2 use entity work.B_AND2(FUNC)
                                          port map(I1,I2,O1);
            end for;
            end for;
        end for;
    end for;
    end config_decoder;

    begin
      Invert_select:
         Inverter port map ( Sel(N),Sel_bar );
    -- 如果 N=1, 则是 1-2 译码器
      Not_recursive:
         if N=1 generate
            Dout(N)  <= Sel(N);
            Dout ( 2**(N-1) + 1 ) <= Sel_bar;
        end generate;

    -- 以下是 N >1 时的情况
      Recursive:
         if N >1 generate                          -- 生成语句
            B1 : block                        -- 块语句
                   signal Temp : Bit_vector ( 1 to 2**(N-1) );
          begin
               N_minus_1 : decoder generic map (N-1);
                           port map ( Sel ( 1 to N-1 ),Temp );
               For_each_output_from_N_minus_1:
               for I in 1 to 2**(N-1) generate
                 And_each_N_minus_1_with_Sel:
                     And2 port map ( Temp(I),Sel(N),Dout ( 2*(I-1) +1 ));
                And_each_N_minus_1_with_Sel_bar:
                     And2 port map ( Temp(I),Sel_bar,Dout ( 2*I ));
              end generate;
          end block;
        end generate;

      end Generic_structure;
```

（源描述文件名：40_generic_dec.vhd）

第41例　带类属的测试平台

袁　媛

本例是第40例的一个测试平台。有关如何编写测试平台的问题，最早在第7例中已经讨论过。但是与普通的测试平台相比较，带类属的测试平台有许多值得注意的地方。

1. 源描述

```
entity test_decoder4 is
end test_decoder4;

architecture BENCH of test_decoder4 is
component decoder4
    port(
        Sel : Bit_vector ( 1 to 4 );
        Dout : out Bit_vector ( 1 to 16 ));
end component;
for I1: decoder4  use entity work.decoder(Gerneric_structure);
                                    generic map (4);
                                    port map ( Sel, Dout);
signal t_S : Bit_vector( 1 to 4 );
signal t_0 : Bit_vector( 1 to 16 );
begin
I1 : decoder4
       port map (
                 Sel => t_S,
                 Dout => t_0);
driver: process
begin
    t_S <=   "0000",
             "0001" after 100 ns,
             "0010" after 200 ns,
             "0011" after 300 ns,
             "0100" after 400 ns,
             "0101" after 500 ns,
             "0110" after 600 ns,
             "0111" after 700 ns,
    wait for 2 us;
    assert false
    report "-----End of Simulation------"
```

```
      severity error;
    end process;
  end BENCH;
```

2. 指定类属值

在第 40 例中，由于采用了类属，所描述的译码器的端口数目由类属值 N 确定。在源描述中并没有指定确切的类属值，而在进行模拟时，就必须指定类属值，因为每次模拟的对象只能是一个确切的对象，如 2-4 译码器、3-8 译码器或 4-16 译码器等等。

传到具体元件的数据是静态数据，一旦模型连接到模拟器，在模拟期间这些数据将不会再改变。

本例是在组装规定中指定类属值的，语句如下：

```
for I1: decoder4  use entity work.decoder(Gerneric_structure);
                                generic map (4);
                                port map ( Sel, Dout);
```

此组装语句将 N 指定为 4，所以本例所测试的是一个 4-16 译码器，指定一个具体的类属值是非常重要的，否则模拟器将无所适从，会报错。

3. 端口及信号的定义

在组装语句指定了类属值以后，端口及信号的数目就不再是不确定的了，既然是 4-16 译码器，那么输入端口为 4 位的位向量，输出端口为 16 位的位向量，所以在结构体的开始定义元件时，语句如下：

```
component decoder4
    port(
          Sel  : Bit_vector ( 1 to 4 );
          Dout : out Bit_vector ( 1 to 16 ));
end component;
```

而测试平台所需的信号也就有了确切的数目：

```
signal t_S : Bit_vector( 1 to 4 );
signal t_O : Bit_vector( 1 to 16 );
```

除了与类属有关的地方，本测试平台与其他普通的测试平台基本上一样，始终不能忘记：在测试平台中，类属值必须是确定的，所要测试的也是一个具体的元件，不允许有不确定的因素存在。

（源描述文件名：41_generic_testbench.vhd）

第42例　行为与结构的混合描述

袁　媛

采用 VHDL 语言进行数字系统自动设计，接触到的数字系统多种多样，单纯地采用行为描述或单纯地采用结构描述可能都不能满足系统的要求，有时候就需要使用行为与结构的混合描述。本例就是一个比较清楚的行为与结构的混合描述。

1. 电路系统的原理

电路系统的示意图如图 42.1 所示。

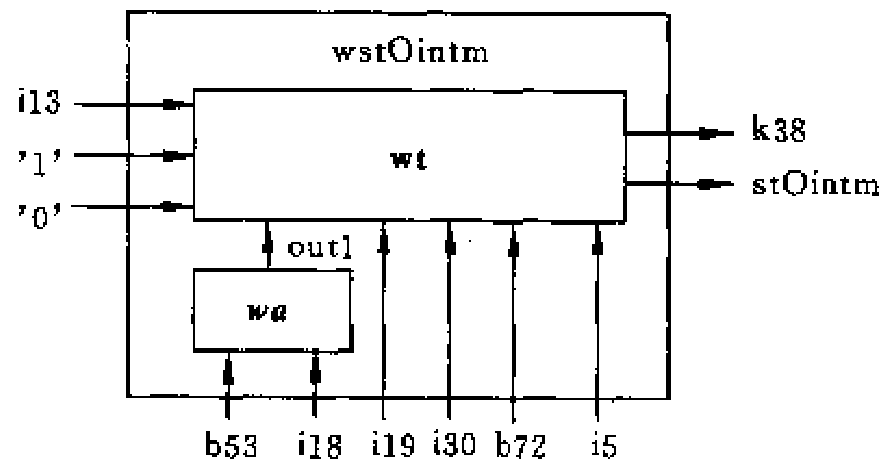


图 42.1　电路系统示意图

此电路的名称为 wst0intm，从示意图中可以看出，它有 11 个外部端口：3 个输入端口 i13、高电平'1'及低电平'0'；6 个控制端口 b53，i18，b72，i19，i30 及 i5 和两个输出端口 k38 和 st0intm。实际上，此电路内部是由两个模块 wa 及 wt 组成的，wa 的输入是 wst0intm 的两个控制端口 b53 及 i18，而它的输出信号 out1 又与 i19、i30、b72 及 i5 一起控制模块 wt。

这是一个多路选择器，在任何时候 out1，b72 及 i5 中只能有一个为'1'。当 out1 = '1'时，选择 i13 作为输入；当 i19 = '1'时，选择高电平'1'作为输入；而当 i30 = '1'时，则选择低电平'0'作为输入。b72 及 i5 是另外一种类型的控制端口，它们能够控制在一定的时延后，将输入送到输出端口。输出端 st0intm 保存输入的值，而 k38 保存经过反相后的输入值。

wa 是此电路的一个底层模块，它所完成的功能是在 b53 的控制下，将 i18 的值输出到 out1 中去，形成 wt 的一个控制端口。

2. 电路的 VHDL 语言描述方法及语法分析

(1) 源描述

```
library IEEE;
use IEEE.std_logic_1164.all;
------------------------------------------------------------
entity wst0intm is
------------------------------------------------------------
    port(
                i13   :  in std_logic;
                i19   :  in std_logic;
                i30   :  in std_logic;
                b53   :  in std_logic;
                i18   :  in std_logic;
                b72   :  in std_logic;
                i5    :  in std_logic;
                -------------------------
                k38   :  out std_logic;
                st0intm   :  out std_logic );
end wst0intm;

architecture struc of wst0intm is
    -------------------------
    component wa
    -------------------------
        port(
                inn   :  in std_logic;
                b53   :  in std_logic;
                -------------------------
                outt  :  out std_logic );
    end component;

for U1 : wa use entity work.wa(FUNC);

signal out1   :  std_logic;
begin
    U1 : wa
        port map(i18,b53,out1);
    process
        variable ic : std_logic;
        variable id : std_logic;
        begin
            id := i19 & i30 & out1
```

```
            case id is
                when "100" => ic := '1';
                when "010" => ic := '0';
                when "001" => ic := i13;
                when others => NULL;
            end case;
            if b72 = '1' then
                st0intm <= ic after 0 ns;
            end if;
            if i5 = '1' then
                if ic = '0' then k38 <= '0';
                else k38 <= k38;
                end if;
            end if;
    end process;
end struc;
```

(2) 混合描述

本例采用结构描述将底层模块 wa 组装到 wst0intm 中去，而其他部分则采用行为描述。

在结构体的开头定义一个 **component**，这个元件就是在其他地方已经定义的底层模块 wa，然后用语句

```
U1 : wa
    port map(i18,b53,out1);
```

将此元件组装到整个电路的描述当中，而 out1 就是我们在示意图中所看到的内部信号，在进行元件例示以后，out1 就可以作为内部信号而被用在其他的行为描述中。当然，在进行元件例示之前，别忘记用组装语句：

```
for U1 : wa use entity work.wa(FUNC);
```

在元件例示语句之后是一个进程，此进程描述了除 wa 以外的电路其他部分的行为，与元件例示语句一起完成对整个电路 wst0intm 的描述。此进程中所涉及到的语法现象比较简单，它描述了一个三路选择器的功能，其中采用毗连符号 & 进行端口的选择，有关这一语法现象，请参考第 7 例。

（*源描述文件名*：42_mix.vhd）

第 43 例 四位移位寄存器

刘沁楠

1. 电路系统工作原理

本例为一时序设计的举例。该移位寄存器能加载一个数值，并且每次能左移或者右移一位，其电路框图如图 43.1 所示。当信号 load 为高电平时，寄存器并行加载输入 din；如果信号 left_right 活跃，则寄存器根据 left_right 取值为'0'或'1'相应地进行右移或左移；在时钟上升沿时刻，移位结果并行输出至端口 dout。

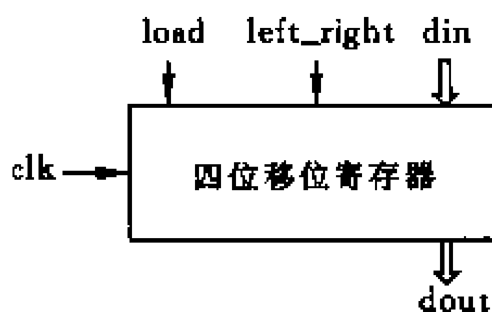


图 43.1 四位移位寄存器框图

2. 涉及的主要语法现象

下面介绍时序电路的 VHDL 描述方法，同时还引入 VHDL 中的重要语法现象——数据类型和子类型。

(1) VHDL 数据类型

VHDL 数据类型相当丰富，正是由于存在这些数据类型，才使得 VHDL 能够创建高层次的系统和算法模型。VHDL 的强类型特点在于：

① 每一个对象具有一个类型且只能具有该类型的值。

② 在定义一个操作时必须指明其操作对象的类型。

VHDL 不但有许多预定义的数据类型，它还允许设计者自己定义数据类型。类型定义的一般形式为：

```
type 类型名 is 类型标志；
```

其中类型标志包容了一种规定类型很广泛的方法，它可以是从一种可枚举所有类型的值到一种复杂记录结构的任何值。

VHDL 预定义类型可以使用户不必显式说明而直接使用它们，主要包括：integer（整数类型）、real（浮点类型）、boolean（枚举类型）、bit（枚举类型）、severity_level（枚举类型）、character（枚举类型）和 time（物理类型）。

VHDL 的数据类型可分为 4 大类，分别是标量类型、复合类型、存取类型和文件类型，下面逐一介绍。

① 标量类型

多数情况下标量类型用于描述一次持有一个值的对象，虽然这种类型本身包含多个值，但一个说明为标量类型的对象在任何时刻最多能持有一种标量值。它包括下面四种类型：整数类型、实数类型、枚举类型和物理类型。

用户经常使用的一种枚举类型 std_ulogic 在程序包 std_logic_1164 中定义如下：

```
type  std_ulogic  is (  'U',     --未定
                        'X',     --强制未知
                        '0',     --强制 0
                        '1',     --强制 1
                        'Z',     --高阻
                        'W',     --弱未知
                        'L',     --弱 0
                        'H',     --弱 1
                        '-'      --无关
                       );
```

② 复合类型

复合类型包括数组和记录。数组类型是同一类型元素的分组，而记录类型允许把不同类型的元素分为一组；数组对线性结构（如 RAM 和 ROM）的建模很有效，而记录则对数据包、指令等的建模有效。

数组可以是一维数组或多维数组，它的元素可以属于任何一种 VHDL 数据类型。在 VHDL 中，类型 bit 和 bit_vector 分别是字符('0','1')和字符数组，它们仅和逻辑运算有关，而没有任何数字值的意义。另一个常用的类型 std_logic_vector 定义如下：

```
type std_logic_vector is array(natural range < >) of std_ulogic;
```

该类型说明描述一个 std_logic 类型数组的类型，数组中的元素数目尚未指定，用 **range**< >表示；natural 定义为从 0 到一个最大整数值，它用于 **range**< >之前，表明 std_logic_vector 类型为分布在从 0 到最大整数值范围的元素。

记录类型把多种类型的对象划作一个单一的对象组，记录的每个元素由它的字段名访问。记录元素可包括任何类型的元素，并可分属不同的类型。

对于复合类型的信号而言，可以整体赋值，也可以对其子元素分别赋值。

③ 存取类型

该类型实际上是指针类型，它用于在对象之间建立联系，或者给新对象分配或释放存储空间。存取类型允许设计者为一些动态性质的对象创建模块，并可建立和维持链表。

④ 文件类型

文件类型用于在主系统环境中定义代表文件的对象，文件对象的值是主系统文件中值的序列。图 43.2 是 VHDL 中可用类型的一个图示，读者可以通过它加深对 VHDL 中数据类型的理解。

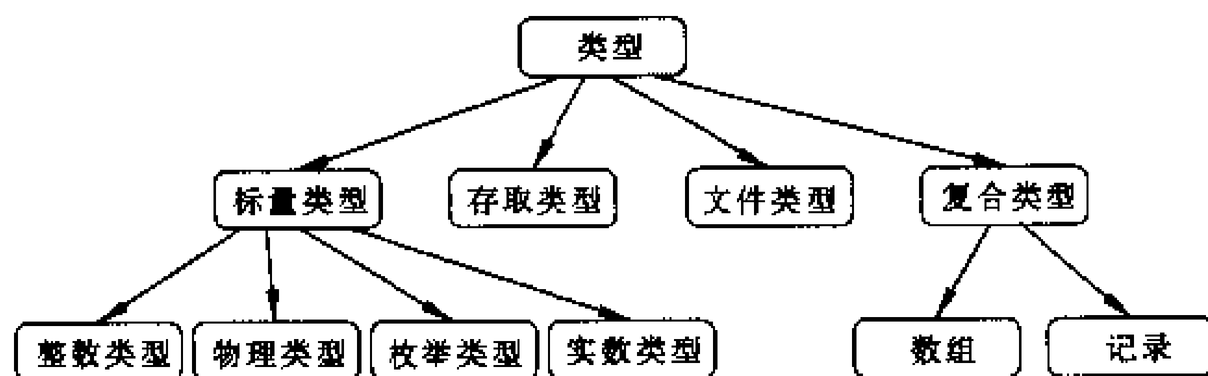


图 43.2 VHDL 数据类型图示

(2) 子类型

类型说明定义一个值域，它和任何其他类型说明定义的值域完全不同，有时候，一个对象可能取值的值域是某个类型说明定义的值域的子集，这就需要用到子类型的概念。子类型和其他类型完全兼容。但是，一旦某个对象被说明为某个特定的子类型时，它的取值范围就不允许超出该子类型的值域。子类型既可以对其先驱类型施加限制条件，也可以简单地对其先驱类型重新命名而不增加任何新意义。

3. VHDL 描述方法

本例的四位移位寄存器的 VHDL 描述如下：

```
library ieee;
use ieee.std_logic_1164.all;

--包说明，说明子类型
package shift_types is
    subtype bit4 is std_logic_vector(3 downto 0);
end shift_types;

use work.shift_types.all;
library ieee;
use ieee.std_logic_1164.all;
--实体说明，说明移位寄存器的外部端口
```

```
entity shifter is
port(
        din:in bit4;
        clk, load, left_right: in std_logic;
        dout: inout bit4
);
end shifter;

--结构体
architecture synth of shifter is
    signal shift_val: bit4;
begin

    nxt:process(load, left_right, din, dout)
        begin
            if(load = ' 1' ) then shift_val <= din;
            elsif(left_right = ' 0' )
                then    shift_val(2 downto 0) <= dout(3 downto 1);
                        shift_val(3)<= ' 0' ;
                else
                        shift_val(3 downto 1) <= dout(2 downto 0);
                        hift_val(0) <=' 0' ;
            end if;
        end process;

    current : process
         begin
                wait until clk' EVENT and clk = ' 1' ;
                dout <= shift_val;
         end process;
 end synth;
```

在包 shift-types 中描述移位寄存器输入和输出用的是类型 bit4，该类型是 std_logic_vector 的子类型；在实体说明部分说明寄存器的输入输出端口 din 和 dout；控制移位寄存器功能的端口 clk，load 和 left_right 是属于类型 std_logic 的信号，其中类型 std_logic 在 std_logic_1164 包中定义如下：

```
type subtype std_logic is resoveld std_ulogic;
```

结构体的描述由两个进程构成，进程（current）保持对移位寄存器当前值的跟踪，它是具有单个 **wait** 语句和单个信号赋值语句的进程。当 clk 信号有上升沿发生时，信号赋值语句开始起作用，并且写 shifter 的下一个值到持有当前状态的 shifter 信号中去。而另一个进程根据上次的值和控制输入计算下次的值。

此处用进程 nxt 计算 shift-val 的新值，以便写到 dout。加载 load 信号是最高优

先级的输入，如果它等于‘1’，将引起 shift_val 接受 din 的值，否则由信号 left_right 可知移位寄存器正在做左移或右移，同时写‘0’值到已被移出的位。

在本例中，对于复合类型信号的赋值采用整体赋值的方法，如语句“shift-val(2 **downto** 0) <= dout(3 **downto** 1);”，也可分别为其子元素赋值：

```
shift_val(2) <= dout(3);
shift_val(1) <= dout(2);
shift_val(0) <= dout(1);
```

两种赋值方法的效果一样。

4. 模拟结果

通过为本例书写测试平台，并对其进行模拟，可以验证上述 VHDL 模型的功能是完全正确的。模拟产生的波形图如图 43.3 所示。测试平台描述请见文件 43_test_register.vhd。

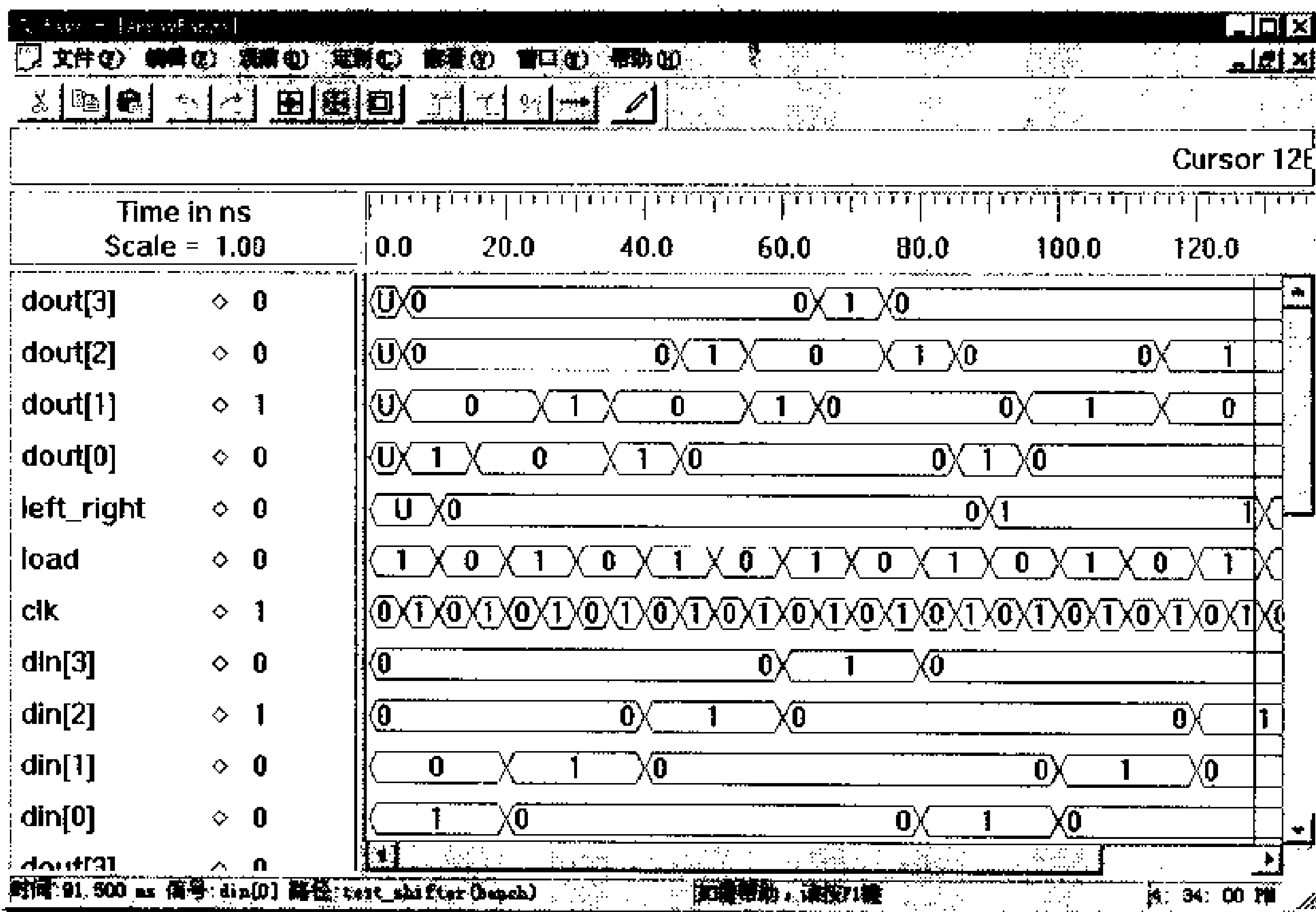


图 43.3　移位寄存器模拟波形图

本例测试平台如下：

```
library work;
use work.shift_types.all;

library ieee;
use ieee.std_logic_1164.all;

entity test_shifter is
end test_shifter;
architecture BENCH of test_shifter is
    component shifter
        port(
            din: in bit4;
            clk, load, left_right: in std_logic;
            dout: inout bit4);
    end component;

    signal din: bit4;
    signal clk: std_logic;
    signal load: std_logic;
    signal left_right: std_logic;
    signal dout: bit4;

    for all: shifter use entity work.shifter(synth);
    begin
    shifter1:shifter
        port map(
        din,
        clk,
        load,
        left_right,
        dout);

shift_driver:process
begin
-------------------------------
--#####     pattern 1     #####
-------------------------------
--设置 load =' 1' ,将 din 输入到 shift_val
clk <= ' 0' ;
load <= ' 1' ;
din <= "0001";
wait for 5 ns;
```

```
-- 将 shift_val 放入 dout
clk <= ' 1' ;
wait for 5 ns;

--shift_to _right
clk <= ' 0' ;
load <=' 0' ;
left_right <= ' 0' ;
wait for 5 ns;

-- 将结果放入 dout
clk <= ' 1' ;
wait for 5 ns;
assert (din/= "0000")
report("assert1 din/= 0000")
severity warning;
-----------------------------
--#####     pattern 2     #####
-----------------------------
-- 设置 load =' 1'  ,将 din 输入到 shift_val
clk <= ' 0' ;
load <= ' 1' ;
din <= "0010";
wait for 5 ns;

-- 将 shift_val 放入 dout
clk <= ' 1' ;
wait for 5 ns;

--shift_to _right
clk <= ' 0' ;
load <=' 0' ;
left_right <= ' 0' ;
wait for 5 ns;

-- 将结果放入 dout
clk <= ' 1' ;
wait for 5 ns;

assert (din/= "0001")
report("assert2 din/= 0001")
```

```
severity warning;
-----------------------------
--#####     pattern 3     #####
-----------------------------
-- 设置 load =' 1'  ,将 din 输入到 shift_val
clk <= ' 0' ;
load <= ' 1' ;
din <= "0100";
wait for 5 ns;

-- 将 shift_val 放入 dout
clk <= ' 1' ;
wait for 5 ns;

--shift_to _right
clk <= ' 0' ;
load <=' 0' ;
left_right <= ' 0' ;
wait for 5 ns;

-- 将结果放入 dout
clk <= ' 1' ;
wait for 5 ns;

assert (din/= "0010")
report("assert1 din/= 0010")
severity warning;
-----------------------------
--#####     pattern 4     #####
-----------------------------
-- 设置 load =' 1'  ,将 din 输入到 shift_val
clk <= ' 0' ;
load <= ' 1' ;
din <= "1000";
wait for 5 ns;

-- 将 shift_val 放入 dout
clk <= ' 1' ;
wait for 5 ns;

--shift_to _right
clk <= ' 0' ;
```

```
load <=' 0' ;
left_right <= ' 0' ;
wait for 5 ns;

-- 将结果放入 dout
clk <= ' 1' ;
wait for 5 ns;

assert (din/= "0100")
report("assert1 din/= 0100")
severity warning;
--------------------------------
--#####      pattern 5      #####
--------------------------------
-- 设置 load =' 1'  ,将 din 输入到 shift_val
clk <= ' 0' ;
load <= ' 1' ;
din <= "0001";
wait for 5 ns;

-- 将 shift_val 放入 dout
clk <= ' 1' ;
wait for 5 ns;

--shift_to _right
clk <= ' 0' ;
load <=' 0' ;
left_right <= ' 1' ;
wait for 5 ns;

-- 将结果放入 dout
clk <= ' 1' ;
wait for 5 ns;

assert (din/= "0010")
report("assert1 din/= 0010")
severity warning;
--------------------------------
--#####      pattern 6      #####
--------------------------------
-- 设置 load =' 1'  ,将 din 输入到 shift_val
clk <= ' 0' ;
```

```
load <= ' 1' ;
din <= "0010";
wait for 5 ns;

-- 将 shift_val 放入 dout
clk <= ' 1' ;
wait for 5 ns;
--shift_to _right
clk <= ' 0' ;
load <=' 0' ;
left_right <= ' 1' ;
wait for 5 ns;

-- 将结果放入 dout
clk <= ' 1' ;
wait for 5 ns;

assert (din/= "0100")
report("assert1 din/= 0100")
severity warning;
-------------------------------
--#####     pattern 7     #####
-------------------------------
-- 设置 load =' 1' ，将 din 输入到 shift_val
clk <= ' 0' ;
load <= ' 1' ;
din <= "0100";
wait for 5 ns;

-- 将 shift_val 放入 dout
clk <= ' 1' ;
wait for 5 ns;

--shift_to _right
clk <= ' 0' ;
load <=' 0' ;
left_right <= ' 0' ;
wait for 5 ns;

-- 将结果放入 dout
clk <= ' 1' ;
wait for 5 ns;
```

```
assert (din/= "1000")
report("assert1 din/= 1000")
severity warning;
-----------------------------
--#####     pattern 8     #####
-----------------------------
-- 设置 load =' 1'  ,将 din 输入到 shift_val
clk <= ' 0' ;
load <= ' 1' ;
din <= "1000";
wait for 5 ns;

-- 将 shift_val 放入 dout
clk <= ' 1' ;
wait for 5 ns;

--shift_to _left
clk <= ' 0' ;
load <=' 0' ;
left_right <= ' 1' ;
wait for 5 ns;

-- 将结果放入 dout
clk <= ' 1' ;
wait for 5 ns;

assert (din/= "0000")
report("assert1 din/= 0000")
severity warning;
--wait for 100 ns;
assert false
report "---end of simulation---"
severity error;

end process;
--   clk <= not clk after 5 ns;
end BENCH;
```

（源描述文件名：43_shift_reg.vhd
测试平台文件名：43_test_register.vhd）

第44例　寄存/计数器

袁　媛

1. 电路系统工作原理

本例是由12个触发器组成的寄存/计数器，有统一的时钟，而且当它作为计数器使用时，是一个减法计数器。此寄存/计数器的电路系统示意图如图44.1所示。

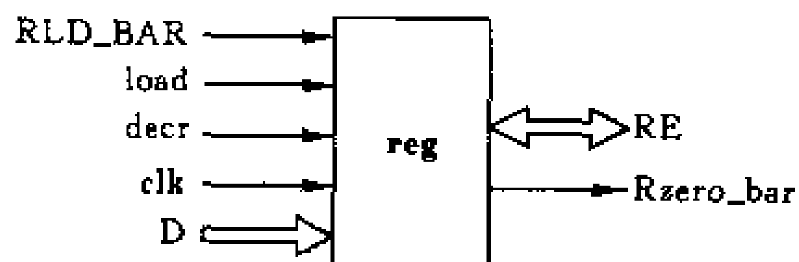


图44.1　寄存/计数器示意图

在图44.1的各个端口中，clk是时钟信号；RLD_BAR是线选信号；load是用于判断是否装载数据的装载信号；decr是减数信号；D是数据输入端口；RE既是数据输入端口，又是数据输出端口；Rzero_bar是判断输出数据是否为0的判断信号。

2. VHDL语言描述方法及语法分析

(1) 寄存/计数器源描述

```
use work.typew.all;
use work.MVL7_functions. all;
use work.synthesis_types. all;

entity reg is
port (
        RLD_BAR     :   in MVL7;
        load        :   in MVL7;
        decr        :   in MVL7;
        clk         :   clock;
        D           :   in MVL7_VECTOR(11 downto 0);
        RE          :   inout MVL7_VECTOR(11 downto 0);
        Rzero_bar : out MVL7
    );
end reg;
```

```
architecture reg of reg is
begin
------------------------------------------------------------
reg_ctr   :   block((clk = ' 1' ) and (not clk' STABLE))
begin
RE <= guarded D              when ((load = ' 1' ) or (RLD_BAR = ' 0' ))
      else RE-"0001"   when ((decr = ' 1' ) and (RLD_BAR = ' 1' ))
      else RE;
Rzero_bar <= RE(0) or RE(1) or RE(2) or RE(3) or RE(4) or RE(5)
             or RE(6) or RE(7) or RE(8) or RE(9) or RE(10) or RE(11);
end block reg_ctr;
------------------------------------------------------------
end reg;
```

(2) 寄存/计数器描述方法

在源描述的结构体中，用了一个保护模块，关键字为 **block**。只有当 **block** 所带的条件满足时，才会执行保护模块中的语句。保护条件(clk = '1') **and** (**not**(clk'STABLE)) 表示的是在时钟跳变到上升沿时执行保护模块中的语句。

当(load = '1') **or** (RLD_BAR = '0')这一条件满足时，输入数据 D 的值赋给 RE，这表示将数据 D 保存在 reg 中，并输出到 RE 端口中，从而起到寄存器的作用。当(decr = '1') **and** (RLD_BAR = '1')这一条件满足时，RE 中的内容减 1 再放入 RE 中，此时是减法计数器。其他情况下，RE 中的内容保持不变，是一个寄存器。最后一条语句：

```
Rzero_bar <= RE(0) or RE(1) or RE(2) or RE(3) or RE(4) or RE(5)
or RE(6) or RE(7) or RE(8) or RE(9) or RE(10) or RE(11);
```

是用于判断 RE 中的内容是否为 0。

3. 测试向量及模拟结果

测试向量文件如下：

```
use work.types.all;
use work.MVL7_functions. all;
use work.synthesis_types. all;

entity E is
end;
architecture AA of E is
    component creg
        port (
                RLD_BAR       :    in MVL7;
```

```
            load          :    in MVL7;
            decr          :    in MVL7;
            clk           :    clock;
            D             :    in MVL7_VECTOR(11 downto 0);
            RE            :    inout MVL7_VECTOR(11 downto 0);
            Rzero_bar :   out MVL7
            );
    end component;
signal RLD_BAR     :    MVL7;
signal load        :    MVL7;
signal decr        :    MVL7;
signal clk         :    clock;
signal D           :    MVL7_VECTOR(11 downto 0);
signal RE          :    MVL7_VECTOR(11 downto 0);
signal Rzero_bar   :    MVL7;
for all : cret use entity work.reg(reg);
begin
    CREG1: creg port map (
                         RLD_BAR,
                         load,
                         decr,
                         clk,
                         D,
                         RE,
                         Rzero_bar
                         );
    process
    begin
        clk <= '0' ;
        wait for 1 ns;
        RLD_BAR <= '1' ;
        load <= '1' ;
        decr <= '1' ;
        D <= "000000000000";
        wait for 4 ns;
        clk <= '1' ;
        wait for 4 ns;
        assert (Rzero_bar = '0' )
        report "Assert 0 : < Rzero_bar /= 0 >" severity warning;
        assert (RE = "0000")
        report "Assert 1 : < RE /= 0000 >" severity warning;
        wait for 1 ns;
```

```
    end process;
end AA;
```

根据所写的测试码，显而易见，最后 RE 中的内容为“000000000000”。

（源描述文件名：44_reg_counter.vhd
测试平台文件名：44_test_vector.vhd）

第45例 顺序过程调用

陈东瑛

1. 电路的 VHDL 语言描述方法及语法分析

本例用于 VHDL 语言的顺序过程调用语法现象的测试，它仅包含一个文件 45_test_63.vhd，其中有空实体说明 test_63 和结构体 behave_1。结构体 behave_1 由两个进程 Proc_Scope 和 Gen_Sig_Nat 构成。

```
--顺序过程调用测试描述文件 45_test_63.vhd

entity test_63 is                                        --测试台空实体说明
end test_63;
architecture Behave_1 of Test_63 is
  signal Sig_Nat           : NATURAL := 0;
  signal Sig_Nat_Procedure : NATURAL := 0;
begin
  Proc_Scope:
  process
      variable Count        : Natural := 0;
      variable Count_Temp_1 : Natural := 0;
      variable Count_Temp_2 : Natural := 0;
      variable Exit_Loop    : BOOLEAN := FALSE;
      procedure Set_55 is        --顺序过程调用的过程体在进程内说明
         begin
           If Sig_Nat = 0 then
              wait until Sig_Nat = 2;
              if Count_Temp_2 /= 55 then
                 Sig_Nat_Procedure <= 55;
                 Count :=55;
            Exit_Loop := true ;
              end if;
          end if;
        end Set_55;
   begin
     Loop_1:
     loop
       Count_Temp_2 :=Count_Temp_1 + 1;
```

```
      Set_55;                                              --顺序过程调用
      If Exit_Loop =TRUE then
         Count_Temp_1 := Count;
         wait on Sig_Nat_Procedure' TRANSACTION;   --信号 Sig_Nat_Procedure
                                                   --上有事务发生为激活条件
         Count_Temp_2 := Sig_Nat_Procedure;
         exit Loop_1;
      end if ;
      wait for 15 ns;
    end loop loop_1;
    Count := Count + 1;
    wait for 20 ns;
  end process Proc_Scope;
  Gen_Sig_Nat:                                     --无穷循环的进程
  process
  begin
    wait for 10 ns;
    Sig_Nat <= Sig_Nat + 1;
  end process Gen_Sig_Nat;
end Behave_1;
```

在 VHDL 语言中，常常把进程内部的一段相对独立的程序段作为一个顺序过程调用的过程体内容，并以该顺序过程调用语句代替原有的程序段，实现等价的功能描述，而使设计的总体结构更加清晰、明了。

本例的顺序过程调用没有参数，比有参数的过程调用要简单，但是通用性差。

2. 模拟测试向量的选择及模拟结果分析

以 Vsim/Talent 系统单步（next，step）模拟运行本例，可以清楚地观察到每一条语句的执行以及过程内部的 **wait** 语句的作用。进程 Gen_Sig_Nat 反复激活运行，而进程 Proc Scope 在第 2 次激活后的运行中，永久挂起于等待语句 **wait on** Sig_Nat_Procedure' TRANSACTION; 的位置。由于进程 Gen_Sig_Nat 的反复激活，所以要注意通过预先设置断点或中断模拟控制，避免本测试例运行的无穷循环。

通过 Vsim/Talent 系统的对象跟踪以及波形显示工具，可以观察本测试例的所有变量、信号的动态变化和信号的整体波形记录。

（源描述文件名：45_test_63.vhd）

第 46 例　VHDL 中 generic 缺省值的使用

王作建

1. 电路系统工作原理及功能

用一个二输入与门、一个带反向输入端的二输入与门及一个二输入或门组合实现二选一多路器。

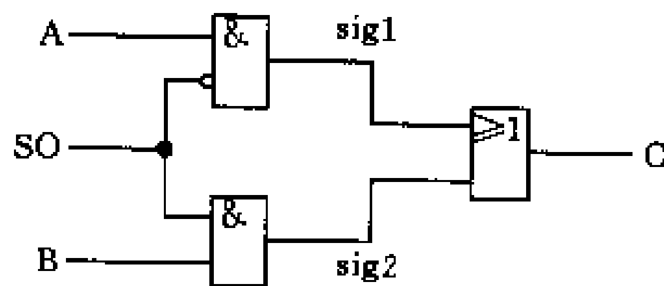


图 46.1　一位二选一多路器的门级实现

2. VHDL 语言描述方法及语法分析

VHDL 源描述首先给出 3 个基本门的行为描述，然后给出二选一多路器完整的结构描述。在多路器的结构体中，包括上述 3 个元件的元件说明，然后是组装说明及信号说明，最后给出元件例示。

在 3 个基本门的行为定义描述中，各自的实体（**entity**）引入类属（**generic**）来刻画各自的时间延迟。实体类属中的 lDelay 的具体值则在多路器的结构描述的元件说明中的类属内指定，即对延迟值采用了在实体说明中缺省指定的方式。

由于采用上述指定方式，因此在接下来的组装规定及元件例示中均未再用类属映射（**generic map**）给 lDelay 赋值，因为在这种情况下，即使赋值也将被上述缺省值覆盖（override）而无效，即（以与门来说明）：

```
component B_AND2
    generic(lDelay : time:=5 ns);--元件说明中给出延时缺省值
    port(......);
end component;
......
for U1:B_AND2 use entity work.B_AND2 (FUNC);
generic map(lDelay => 10 ns);
--组装说明中用 generic map 给 lDelay 赋值无效
```

或：

```
U1:B_AND2
    generic map(lDelay => 10 ns);
                --元件例示中用 generic map 给 lDelay 赋值无效
        port map(A,S0,sigl);
```

如上述，无论组装说明中的 **generic map** 还是元件例示中的 **generic map** 都不起作用，模拟时门的实体描述中的延时将是5ns。

假如元件说明中未进行缺省指定延时值，则可以在组装说明或元件例示中进行 generic map 指定延时值。若在组装说明中进行 generic map，则无需再在元件例示中指定，且元件说明中也可以不必再有 generic 语句。仍以与门为例说明如下：

```
component B_AND2  --不必写 generic 语句
    port(......);
end component;
......
for U1:B_AND2 use entity work.B_AND2 (FUNC);
generic map(lDelay => 10 ns); --组装说明中进行 generic map
    ......
U1:B_AN2  --不必写 generic map 语句
    port map(......);
```

结论：元件的实体描述中引入了 **generic**，在其对应的 **component** 定义中也必须有 **generic** 定义，关于 **component** 定义中类属缺省值有如下规定：

① 在 **component** 定义元件中类属值缺省指定后，在元件例示中可不写 generic map，此时用元件说明中的缺省值。

② 若 **component** 的 **generic** 未给出缺省值，则在元件例示或组装说明中必须有 **generic map** 给出类属中定义对象的值。

③ 由上述可知，**component** 的类属定义"级别最高"，其实体描述和元件例示随其变化，**component** 给出缺省值，元件例示中 **generic map** 可有可无，而 **entity** 中的 **generic** 随元件例示中的值变化；**component** 的 **generic** 无缺省值，则元件例示或组装说明中必须有 **generic map**，实体中 **generic** 值随其变化，此时在实体描述中缺省值不起作用，完全可以省略。

④ 元件定义中有类属缺省规定，在元件例示中可以没有 **generic map**，但在实体描述中却必须有 **generic** 定义，否则，实体对应结构体中的描述使用的一些对象将没有定义。

VHDL 源描述如下：

(1) 三个基本门的 VHDL 行为描述

```
library IEEE;
use IEEE.std_logic_1164.all;
--------------------------------------------------------------
entity B_BAND is
--------------------------------------------------------------
    generic(lDelay: time);
    port(I      : in std_logic;
         B      : in std_logic;
         FOUT   : out std_logic);
end B_BAND;

architecture FUNC of B_BAND is
begin
         FOUT <=(I and (not B)) after lDelay;
end FUNC;

library IEEE;
use IEEE.std_logic_1164.all;
--------------------------------------------------------------
entity B_AND2 is
--------------------------------------------------------------
    generic
         (lDelay   : time);
    port(I0    : in std_logic;
         I1    : in std_logic;
         FOUT  : out std_logic);
end B_AND2;

architecture FUNC of B_AND2 is
begin
         FOUT <= I0 and I1 after lDelay;
end FUNC;

library IEEE;
use IEEE.std_logic_1164.all;
--------------------------------------------------------------
entity B_OR2 is
--------------------------------------------------------------
    generic
         (lDelay   : time);
    port(I0    : in std_logic;
```

```
        I1   : in std_logic;
        FOUT: out std_logic);
end B_OR2;

architecture FUNC of B_OR2 is
begin
        FOUT <= (I0 or I1) after lDelay;
end FUNC;
```

(2) 二选一多路器的 VHDL 结构描述

```
library IEEE;
use IEEE.std_logic_1164.all;
-------------------------------------------------------------
entity B_MUX2 is
-------------------------------------------------------------
    port (A      : in std_logic;
          B      : in std_logic;
          S0     : in std_logic;
          O      : out std_logic);
end B_MUX2 ;

architecture STRUC of B_MUX2 is
begin
    --元件说明
component B_AND2
    generic(lDelay : time:=5 ns);--元件说明中指定延迟时间
    port (I0       : in std_logic;
          I1       : in std_logic;
          FOUT     : out std_logic);
end component;
component B_OR2
    generic(lDelay : time:=5 ns);
    port (I0   : in std_logic;
          I1   : in std_logic;
          FOUT : out std_logic);
end component;
component B_BAND
    generic(lDelay : time:=5 ns);
    port (I    : in std_logic;
          B    : in std_logic;
          FOUT : out std_logic);
end component;
```

```
--组装说明
for U0:B_BAND use entity work.B_BAND(FUNC);
for U1:B_AND2 use entity work.B_AND2 (FUNC);
for U2:B_OR2 use entity work.B_OR2(FUNC);
--信号说明
signal  sig1,sig2: std_logic;

begin
U0:B_BAND
     port map(A,S0,sig1);
U1:B_AND2
     port map(S0,B,sig2);
U2:B_OR2
     port map(sig1,sig2,O);

end STRUC;
```

（源描述文件名：46_default_generic.vhd）

第 47 例　无输入元件的模拟

王作建

1. 电路系统工作原理及功能

本例使用一个常量元件，其功能是由使用它的电路通过 generic 传递一个数，然后将其输出。因为此元件客观上仅起到产生一个常量信号的作用，因此它无需输入而仅有输出。

上述常量元件的测试电路有一个输入和一个输出，结构上由两部分组成：常量元件和一个二输入与门，电路如图 47.1 所示。

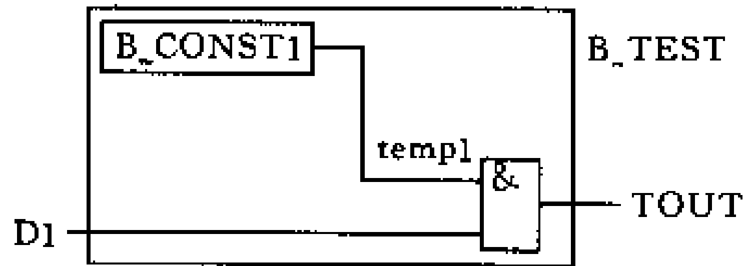


图 47.1　常量元件的测试电路

在测试电路结构描述的元件例示中，对常量元件 B_CONST1 中的 NUM 通过 generic map 赋值为‘1’，然后其输出加到与门的一个输入端，与门的另一个输入端则与测试电路的唯一输入端相连，与门的输出即是测试电路的输出。因此通过对测试电路输入施加激励，观察其输出，可判断与门工作的正确性，从而间接判断常量元件工作的正确性。

2. VHDL 语言描述方法及语法分析

常量元件是一种无输入元件。对无输入元件，直接模拟时无法施加输入激励，从而不能观察输出波形，因此不能直接对此类元件进行模拟。对这类无输入元件的模拟，一般采用间接方法，即为无输入元件构造测试电路，在测试电路中无输入元件的输出作为其余电路的部分输入，其他输入通过测试台（testbench）产生，进而通过观察整个测试电路的输出正确与否来间接判断无输入元件工作是否正常，达到间接模拟的目的。

3. 模拟及结果分析

(1) B_TEST 测试电路的测试台描述

```
library IEEE;
use IEEE.std_logic_1164.all;
entity test_const is
end test_const;

architecture bench of test_const is
    component B_TEST
        port(    D1   : in  std_logic;
                 TOUT: out std_logic);
    end component;
    for U: B_TEST use entity work.B_TEST(FUNC);
    signal t_d1: std_logic;
    signal t_tout : std_logic;
  begin
   U: B_TEST
    port map(t_d1,t_tout);
   driver: process
     begin
      t_d1<= ' 0' ,
             ' 1'  after 100 ns,
             ' 0'  after 200 ns,
             ' 1'  after 300 ns;
     wait for 500 ns;
      assert false
        report "—End of Simulation--"
      severity error;
    end process;
end bench;
```

(2) 模拟结果

模拟波形如图 47.2 所示。

(3) 波形及结果分析

由于在 B_TEST 的结构描述中对 B_CONST1 元件进行例示时，通过 **generic map** 传递的数值是 1，如果常量元件工作正确，则其输出应该是 '1'，即与门的一个输入是 '1'，那么与门的输出完全决定于另一个输入，二者的波形应该完全一样(假设元件延迟为零)。

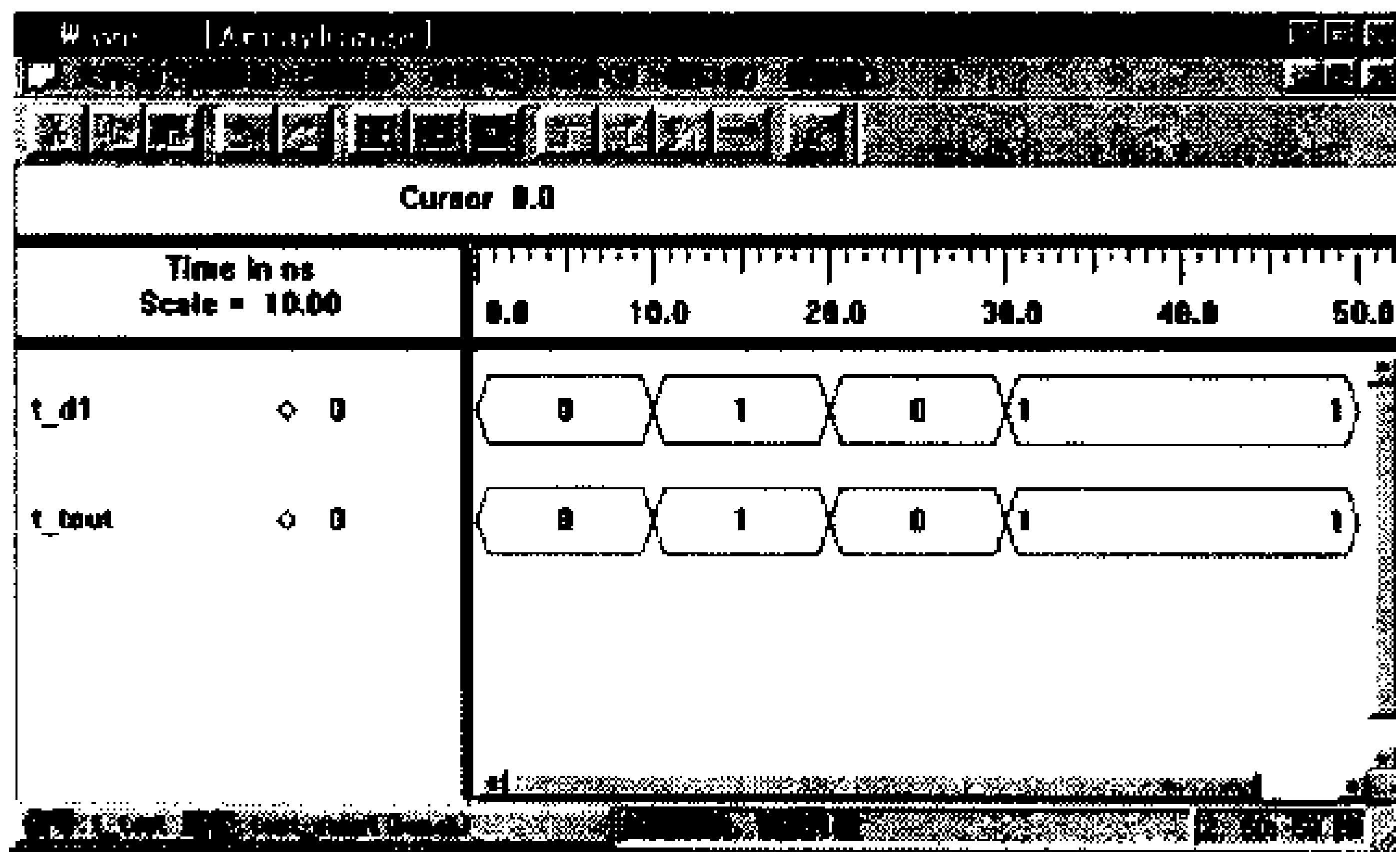


图 47.2 测试电路的波形图

而与门的另一个输入即是 B_TEST 的唯一输入，其输出是 B_TEST 的唯一输出，因此模拟结果中输入输出波形应完全相同。由图 47.2 可知，输入输出波形完全相同；又由上面的推论，可间接判断常量元件工作正确。

因此对无输入元件，采用本方法进行模拟验证是完全可行的。

(4) VHDL 源描述

```
--常量元件（无输入元件）的 VHDL 描述
library IEEE;
use IEEE.std_logic_1164.all;
------------------------------------------------------------
entity B_CONST1 is
------------------------------------------------------------
    generic(
        NUM     : std_logic);
    port(
        POUT: out std_logic);
end B_CONST1;
architecture FUNC of B_CONST1 is
begin
        POUT <=  NUM;
end FUNC;

-- 二输入与门的 VHDL 描述
library IEEE;
```

```
use IEEE.std_logic_1164.all;

entity B_AND2 is
port(
         I0 : in std_logic;
         I1 : in std_logic;
         FOUT : out std_logic
     );
end entity;

architecture FUNC of B_AND2 is
begin
    FOUT <= I0 and I1;
end FUNC;

-- 常量元件的测试电路的 VHDL 描述
library IEEE;
use IEEE.std_logic_1164.all;
------------------------------------------------------------
entity B_TEST is
------------------------------------------------------------
    port ( D1      : in  std_logic;
           TOUT    : out std logic);
end B_TEST;

architecture FUNC of B_TEST is
    component B_CONST1
      generic(
         NUM     : std_logic);
      port(
             POUT: out std_logic);
    end component;
   component B_AND2
        port (I0  : in std_logic;
              I1  : in std_logic;
              FOUT: out std_logic);
   end component;

   for U1: B_CONST1 use entity work.B_CONST1(FUNC);
   for U2: B_AND2 use entity work.B_AND2(FUNC);

   signal templ: std_logic;
```

```
begin
    U1: B_CONST1
      generic map(' 1' )
     port map(
            POUT => templ);
    U2: B_AND2
      port map(
          I0 => templ,I1 => D1,FOUT => TOUT);
end FUNC;
```

（源描述文件名：47_const_test.vhd）

第 48 例　测试激励向量的编写

袁　媛

本例是带有测试向量的一个简例，示例虽小，内容却很丰富。其中关键的几个问题如下：

- 有关测试码及其编写
- VHDL 语言的惯性延迟
- VHDL 的顺序语句与并发语句
- 有关属性 TRANSACTION

1. 有关测试码及其编写

先写测试码然后进行模拟是为了检验所写模块的描述是否正确，是否完成了预期的功能以及设计的意图。测试是将激励作用于在测模块的模型，它将在测模块的响应与期望响应相比较，并报告模拟期间发现的差异。如图 48.1 所示。

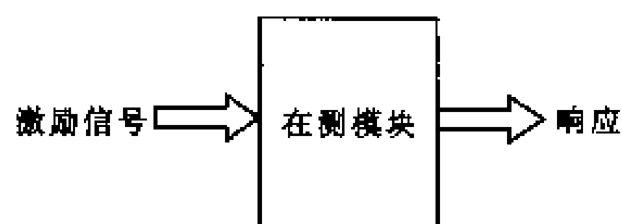


图 48.1　测试模块

所谓在测模块是指，把所描述的模块从外部当成一个整体，而不管内部包含一个元件还是成百上千个元件来进行测试。

测试码的编写不是一个简单的过程，有时模块描述比较简单，但测试码却可以非常复杂。根据测试和模拟的结果，可以对描述进行反复设计与修改，直至达到最终目的。当然，激励信号的选择应该是全面的、典型的。

本例并不是对某个具体模块进行测试，只是为了说明一些问题。因为测试是根据被测实体的端口给它加激励信号，测试本身并不是一个具体的元件，它只是一些激励信号，所以，实体为空。

2. 源描述

```
entity Test_18e is
end Test_18e;
```

```
architecture Behave_1 of Test_18e is
  signal A : bit := '0' ;
  signal B : bit := '0' ;
  signal C : bit := '0' ;
begin
Gen_Wave:
  process
  begin
    A <= '1'  after 5 ns,
          '0'  after 12 ns;
    B <= '1'  after 8 ns;
          '0'  after 14 ns;
    wait on A,B;
  end process Gen_Wave;
  Analysis_C:
  process
    variable Var_C : bit := '0' ;
  begin
    wait on C' TRANSACTION;
    Var_C := C;
  end process Analysis_C;
  Update_C:C <= (A or B) after 40 ns;
  Finish:
  process
  begin
  wait for 100 ns;
  assert false report "End of Simulation" severity error;
  end process Finish;
end Behave_1;
```

3. VHDL 的惯性延迟

延迟是硬件中十分重要的参数，延迟稍有不同，得出的结果就可能大相径庭。延迟分为传输延迟与惯性延迟。本例中用的是惯性延迟。

(1) 信号驱动源上惯性延迟的作用共有三条

- 第 1 个新事务后的所有旧事务均被删除；
- 同一信号赋值语句中第 1 个新事务之后的其他新事务均被添加；
- 对于第 1 个新事务之前的旧事务，若其值与第 1 个新事务之值相同，则保留。

递归执行上述 3 条，直到没有更多的旧事务或旧事务之值与第 1 个新事务之值不同时为止，所有其他旧事务均被删除。

test 18e 中标号为 Gen_wave 的语句及生成波形可以说明上述问题。描述节选如下：

```
Gen_wave:process
        begin
            A <= '1'  after 5 ns, '0'  after 12 ns;
            B <= '1'  after 8 ns;
            wait on A,B;
        end process Gen_wave;
```

按照上述步骤进行分析：

① 第 1 个新事务('1',5)被安排在驱动源中；

② 第 2 个新事务('1',8)安排在('1',5)的后面，删除旧事务('0',12)，因为('0',12)安排在('1',8)之后；

③ **wait on** A,B 此句是在等待 A 或 B 上若有事务处理，进程又再次激活；

④ 因为进程是在前一次运行基础上继续运行，时间起点应该以上一次结束时间为准，这样，('1',8)为第 1 个事务；

⑤ 继续执行时('1',5)被删除；

⑥ 下一事务是('0',12)，因为时间起点为('1',8)，所以在 20ns 后，A 才会由 1 跳到 0；

⑦ B 无变化，继续保持 1；

⑧ 进程再次由于 A 上的事务处理而激活，因为此时 B 不再变化，所以第 1 个事务为('1',5)、第 2 事务为('0',12)；

⑨ 此后情况不再变化，依次类推。

根据分析结果，可得波形如图 48.2。

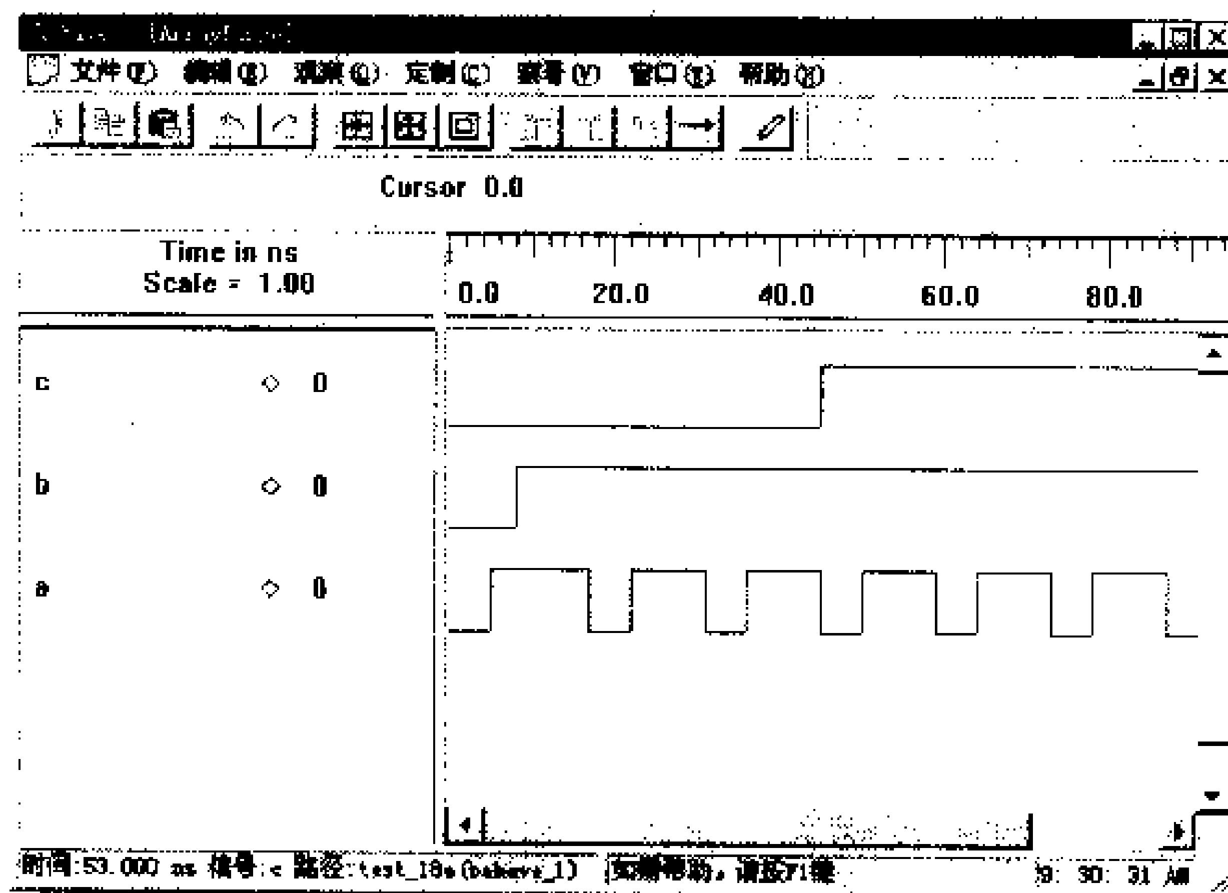


图 48.2　模拟后的波形

如果我们把描述主体改为：

```
A <= '1' after 5ns, '0' after 12ns;
B <= '0' after 8ns, '0' after 10ns;
wait on A,B;
```

则波形如图 48.3 所示。

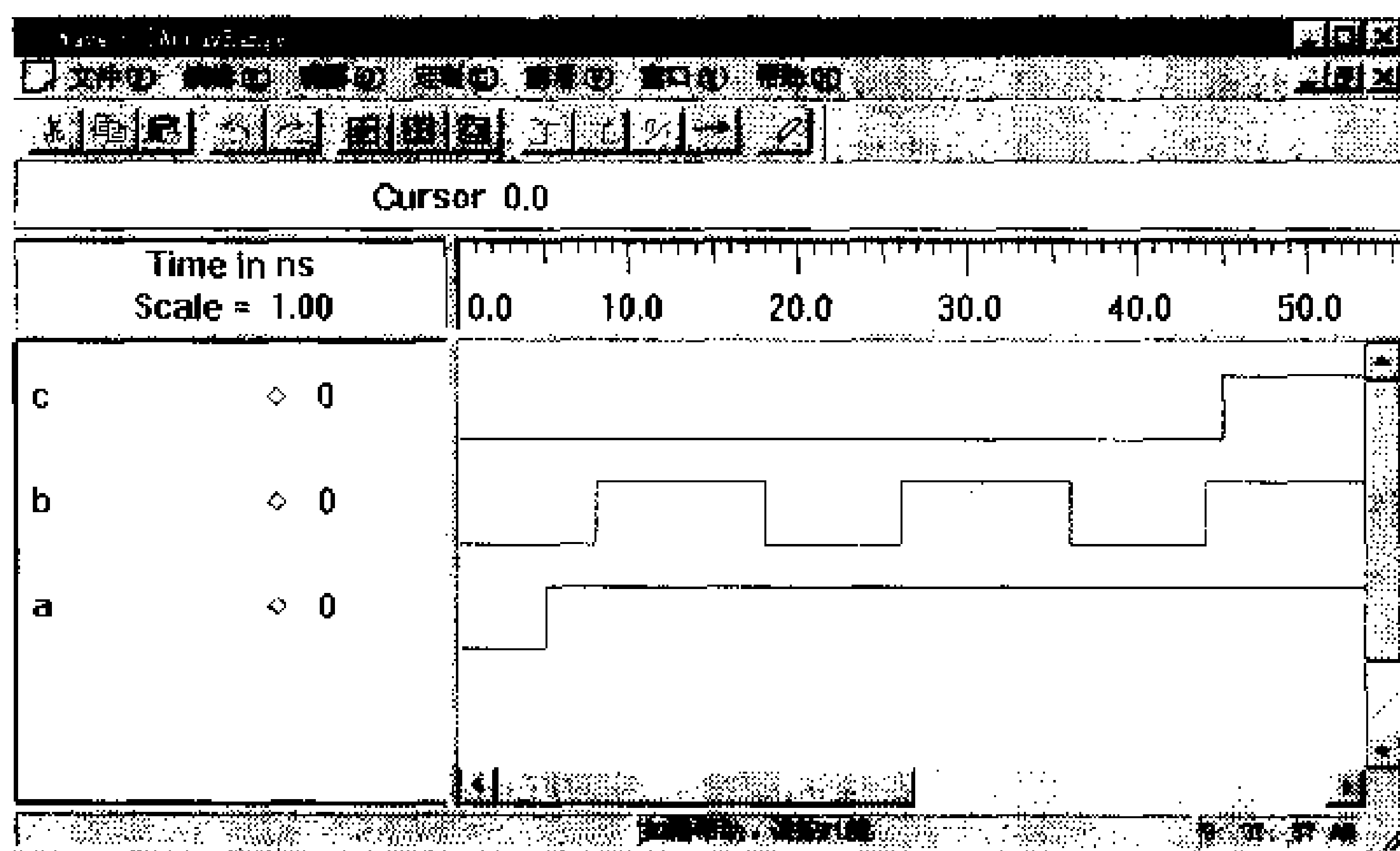


图 48.3 模拟后的波形

具体分析请读者自己进行。

(2) 注意

惯性延迟的语义是，若惯性延迟的赋值语句中的波形给定多个元素，如：A <= '1' **after** 5 ns, '0' **after** 12 ns；则第 1 个元素之外的其他元素均不看作是惯性延迟赋值。这是因为 VHDL 要求信号赋值语句中的波形元素必须是升序，即时间延迟必须是升序的，一旦第 1 个事务加入驱动源，则接下来的每个事务保证能安排在驱动源的所有其他事务之后，这与传输延迟一致。所以 A <= '1' **after** 5 ns, '0' **after** 12 ns；与以下语句的作用不同：

```
A <= '1' after 5 ns;
A <= '0' after 12 ns;
```

4. VHDL 的顺序语句与并发语句

如果按照 test_18e 中的程序模拟，读者会发现程序在运行 0ns+0 后停止，也就是程

序直接跳到最后一条断言语句：

```
assert false report "End of simulation"
severity error;
```

直接告诉模拟系统结束模拟。这是为什么呢？原因在于这是一条并发断言语句。

VHDL 的进程、并发语句之间是并发的，进程内部语句是顺序的。test_18e 中的进程语句 Gen_wave，Analysis_C，Update_C 与并发断言语句 **assert** 是并发的，在某时间点，只要某语句满足条件，就会执行。要使程序可以模拟，得到第 2 点中的波形，则必须把最后的并发断言语句改为

```
Finish:
    process
    begin
        wait for 100ns;
        assert false
        report "End of simulation" severity error;
    end process Finish;
```

这样，程序才会在 100ns 后结束模拟，而 100ns 对 48_test_18e 来说，已足够说明问题了。

5. 有关属性 TRANSACTION

TRANSACTION 是一种信号预定义属性，48_test_18e 中有一条语句 **wait on** C'TRANSACTION 说明 C'TRANSACTION 的值被 C 上的每一种事务处理所启动并引起一事件产生，因而使 **wait** 语句激活。

（源描述文件名：48_test_18e.vhd）

第 49 例　delta 延迟例释

吴清平

本例通过一个简明的示例，讨论 delta 延迟的作用和在写 VHDL 源描述时应该注意的一些原则。

1. VHDL 语言描述方法及语法分析

(1) delta 延迟解释

为了更好地理解描述，必须先分析一下它的功能。它是为方便模拟而引入的一个概念，是一个无限小的时间间隔，也就是任意多个 delta 延迟的和仍为零。引入它的主要是为了确定在模拟过程中各个不同元件的模拟顺序。例如图 49.1 所示的简单电路，设 A 点初始值为 1，假定在输入端 A 处有一个下跳变沿，那么在输出端 D 点的值将会有什么变化呢？要计算 D 点的值，有两条通道：如果先计算与门，将是这样一个变化的顺序：

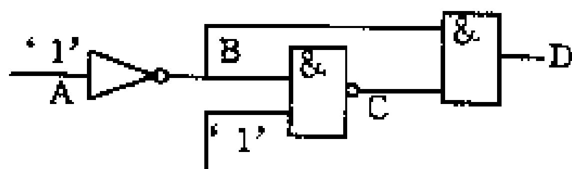


图 49.1　一个简单电路

B <= 1
D <= 1（此时 C=1）
C <= 0
D <= 1

这时在 D 点有一个上跳沿，之后又立即变为 0。如果先计算与非门，将是这样一个变化的顺序：

B <= 1
C <= 0
D <= 0

这时 D 点没有上跳沿。

这两种计算方法虽然结果相同，但在第 1 种方法中出现了一个正脉冲，而在第 2 种计算方法中却没有这个这正脉冲。VHDL 为了解决这种类似现象，提供一个无限小的时间间隔

delta 延迟，信号在被复制之后必须经过 delta 延迟之后才能有效。

如在 10 ns 时 A 上有一个上升沿，则计算过程应该是：

delta 延迟数	计算
1	A <= 0
2	B <= 0
3	D <= 1
	C <= 0
4	D <= 0

这时，在第 3 个 delta 延迟时，要同时进行 D<=1 和 C<=0 两个操作。由于 delta 延迟的存在，因此在写描述时务必小心。请比较如下两个进程结果的区别。

进程 1：

```
signal    a: integer := 30;
signal    b: integer := 40;
process
begin
          a <= b;
          b <= a;
end process;
```

进程 2：

```
process
     variable a: integer := 30;
     variable b: integer := 40;
begin
     a := b;
     b := a;
end process;
```

上面两个进程在写法上非常相似，但是这两个进程的功能完全不一样。进程 1 的作用是使 a，b 两个信号的值相交换，即在模拟之后，信号 a 的值为 40，而信号 b 的值为 30。这是由于在第 1 条语句 a<=b 被模拟（也即执行）以后，信号 a 的值并不立即发生改变，而这条语句的执行结果只是“信号 a 的值将在下一个 delta 延迟时为 40”，同样第 2 条语句 b<=a 被模拟以后的效果也是“信号 b 的值将在下一个 delta 延迟时为 30”。因此信号 a 和 b 的值将在下一个 delta 延迟的时候进行值的更新，最终效果为信号 a, b 的值相交换。

而进程 2 的效果则是变量 a 和 b 的值都变为 40, 这是由于变量赋值语句是没有 delta 延迟的，赋值语句将使目标变量的值立即发生改变，如在此进程中第 1 条变量赋值语句模拟以后，a 的值立即发生变化，其值变为 40；第 2 条语句是变量 b 的值保持为 40。

(2) 源描述分析

本例主要是为了测试 delta 延迟的作用，因此本例中只有一个实体和与之对应的结构体，并且此实体为一个空实体，它没有任何端口。在结构体中定义两个信号：a 和 b，并且只有一个进程，在此进程中有如上所述的信号赋值语句。描述如下：

```
entity delta is
end entity;

architecture archi_delta of delta is
        signal a : integer := 40;
        signal b : integer := 30;
begin
        process
        begin
            a <= b;
            b <= a;
            wait for 10 ns;
            a <= 40;
            b <= 30;
            wait for 20 ns;
        end process;
end;
```

因此，进程中的前两条信号赋值语句的效果是将信号 a 和 b 的值互换，后两条信号赋值语句是普通的信号赋值语句。

2. 模拟结果

此例在模拟后的波形如图 49.2 所示。

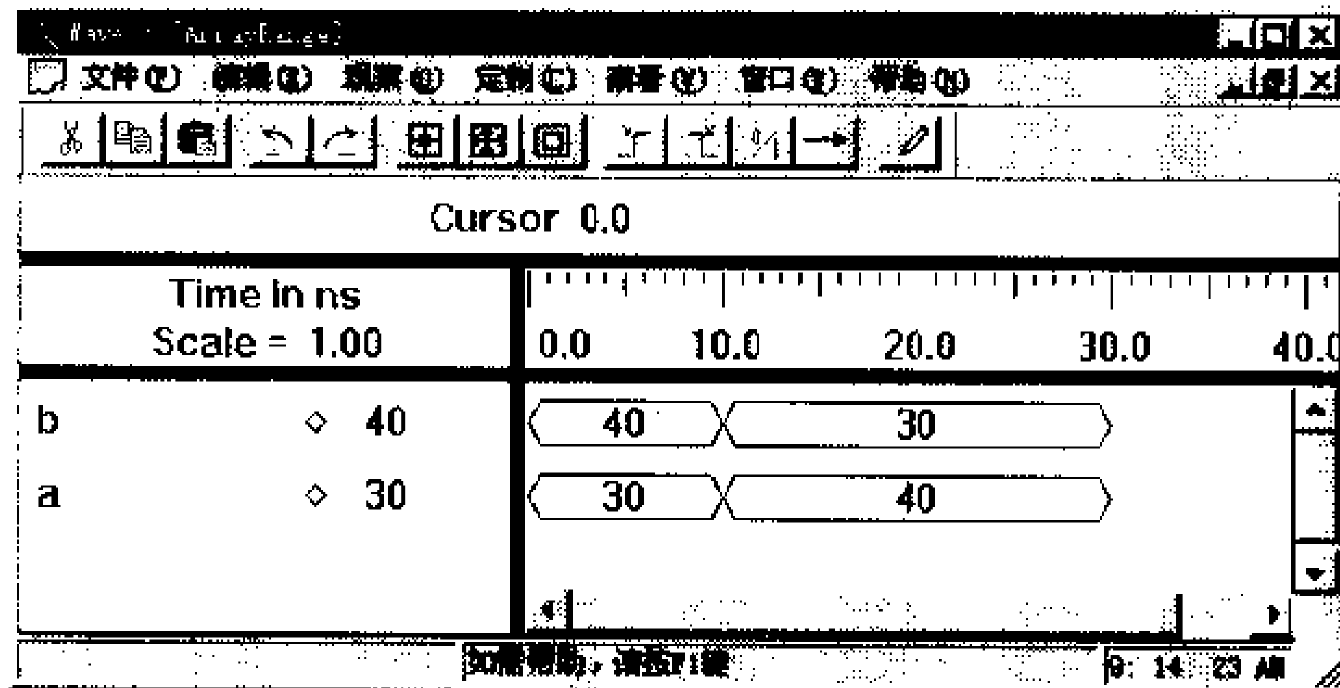


图 49.2　模拟波形

从此例可以看出在模拟中 delta 延迟的重要性。深刻理解 delta 延迟的作用对于正确书写 VHDL 源描述也是非常重要的。

（源描述文件名：49_delta.vhd）

第 50 例　惯性延迟分析

吴清平

本例通过一个简明的示例，讨论信号赋值语句中惯性延迟的作用和写 VHDL 源描述应该注意的一些问题。

1. VHDL 语言描述方法及语法分析

(1) 实体 Test_18b 分析

实体 Test_18b 是一个空实体。因为本例主要用于检查语法现象，因此未定义任何输入或输出端口。

(2) 结构体 Behave_1 分析

结构体中定义了三个信号 A，B，C，其数据类型为 bit，初值均为‘0’，结构体中包含两个进程和一条并发信号赋值语句。如下：

① 进程 Gen_Wave

```
Gen_Wave:
process
begin
    A <= '1'  after 5 ns,  '0'  after 12 ns;
    wait ;
end process Gen_Wave;
```

进程中包含一条顺序信号赋值语句，它分别在 5ns 和 12ns 时给 A 赋值‘1’和‘0’。使得 A 的波形如图 50.1 所示：

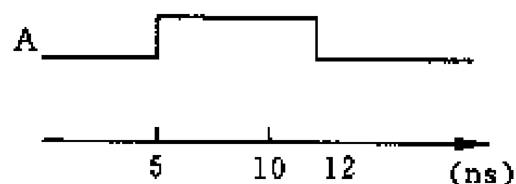


图 50.1　信号 A 的波形图

② 进程 Analysis_C

```
Analysis_C:
process
variable Var_C : BIT := '0' ;
begin
    wait on C'TRANSACTION;
    Var_C :=C;
end process Analysis_C;
```

此进程定义变量 Var_C，类型为 bit，初值为‘0’，它的作用是一旦信号 C 上发生事务，则将 Var_C 的值赋为‘1’。其中 TRANSACTION 是信号的预定义属性，属性 TRANSACTION 将生成一个数据类型为 bit 型的隐式信号，此信号在它所附属的信号发生值更新时，其值将在‘0’和‘1’之间转换。需要注意的是，“所附属的信号发生值更新”并不要求其值发生改变，只要有一个信号更新操作即认为是一个事务，请注意“事务”和“事件”的区别。

③ 并发信号赋值语句

```
Update_C:  C <= A and B after 40 ns;
```

其中 Update_C 是一个标号，此信号赋值语句的作用是在 A 或 B 上发生事件时，则将 A 与 B 的值相与，并在 40ns 之后赋予 C。由于这是一条惯性信号赋值语句，且脉冲宽度限制为 40ns，即任何周期小于 40ns 的脉冲皆不被传输。由于信号 B 上没有事件发生，而信号 A 上的脉冲宽度分别为 5ns 和 7ns，所以均不能传输，因此 C 的值不发生变化，进程 Analysis_C 中的变量 Var_C 的值一直是‘0’。

另外：任何一条并发信号赋值语句都可以转换为一个等价的进程，如以上一条并发信号赋值语句可等价于如下进程语句。

```
process
begin
    wait on a,b;
    C <= A and B after 40 ns;
end process;
```

显然用并发信号赋值语句比使用进程语句更加简单明了。

2. 模拟结果分析

此例的模拟结果如图 50.2 所示，分析见上。

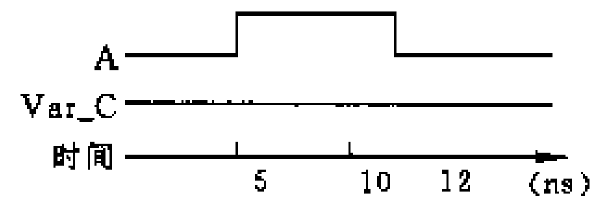


图 50.2　模拟结果

（源描述文件名：50_test_18b.vhd）

第 51 例　传输延迟驱动优先

陈东瑛

1. 电路的 VHDL 语言描述方法及语法分析

本例以一个文件 51_test_113.vhd（包含一个空实体说明 test_113 和一个结构体 behave_1）描述 VHDL 信号传输中的延迟现象，其总体结构与第 19 例类似。

```
--传输延迟驱动优先测试台文件 51_test_113.vhd
entity test_113 is
end test_113;
architecture behave_1 of test_113 is
  signal A : bit := '1' ;
  signal B : bit := '1' ;
  signal C : bit := '1' ;
  function Delay(Value : bit ;Phl :TIME;Plh :TIME ) return TIME is
  begin
       if Value = '1'  then
          return Plh;
      else
          return Phl;
       end if;
  end Delay;
begin
Schedule_A: process
    begin
     wait on B,C;
     A <= transport B and C after Delay(B and C ,16 ns , 10 ns);
  end process Schedule_A;

Analysis_A: process
     variable Var_A : bit := '0' ;
  begin
     wait on A' TRANSACTION;
     Var_A := A;
  end process Analysis_A;

  B <= '0'  after 10 ns , '1' after 12 ns;
```

```
end behave_1;
```

在结构体说明中除了定义并初始化三个信号(A，B，C)之外，还定义一个函数 Delay 用以选择延迟时间。

结构体 behave_1 由两个进程 Schedule_A 和 Analysis_A 以及一条并发信号赋值语句组成。该信号赋值语句可以看成一个等效进程的简化形式，通过对该进程的分析，可以理解该并发信号赋值语句的意义和作用。该进程如下：

```
process
   begin
         B <= '0'  after 10 ns,
              '1'  after 12 ns;
         wait;
end process;
```

首先，由于对信号赋值的一个进程是该信号的一个驱动源，而同一并发信号赋值语句中对同一信号的多次赋值等价于同一进程中对同一信号的多次赋值，所以这里不存在同一信号多源驱动的问题。其次，针对形如本例的信号赋值语句，VHDL 语言规定，除第 1 个赋值为惯性延迟之外，其他赋值都不看成是惯性延迟。这样就提供了一种简便的向信号对象赋一串值的方法(注意：语句中的延迟时间递增设置，才能使每 1 个赋值有效)。最后，**wait** 语句表明该进程向信号 B 做两次赋值后就被永久挂起，即在 behave_1 中，并发信号赋值语句执行一次后就不再执行。

进程 Schedule_A 由信号 B 或 C 激活，以调用函数 Delay 的返回值作为延迟时间向信号 A 赋相应的值。进程 Analysis_A 由 A'TRANSACTION(信号类属性)激活，以变量 var_A 记录信号 A 的值。

以 Vsim/Talent 模拟本例，可以验证上述语法现象。

2. 模拟测试向量的选择及模拟结果分析

由 Vsim/Talent 模拟运行本例，终止于 22ns。其中观察波形如图 51.1 所示。整个信号传输过程为：系统运行到 10ns 时，信号 B 的跳变激活进程 Schedule_A，设置此后 16ns 时刻，向信号 A 赋值；系统运行到 12ns 时，信号 B 再次跳变激活进程 Schedule_A，设置此后 10ns 向信号 A 赋值。根据传输延迟原则，新事务——系统运行到 22ns (=12ns+10ns) 发生，覆盖掉旧事务——系统运行到 26ns (=10ns+16ns) 发生。在系统运行到 22ns 时，向信号 A 赋值'1'，则信号 A 值未变化，无事件发生，但是有事务发生，使信号 A'TRANSACTION 活跃，激活进程 Analysis_A，使变量 var_A 为'1'。由于信号 A 上不再发生事务，并且信号 B，C 不再变化，所有进程不再激活，模拟结束。注意，进程中的每个延迟时间都是相对于本进程的当前时刻。

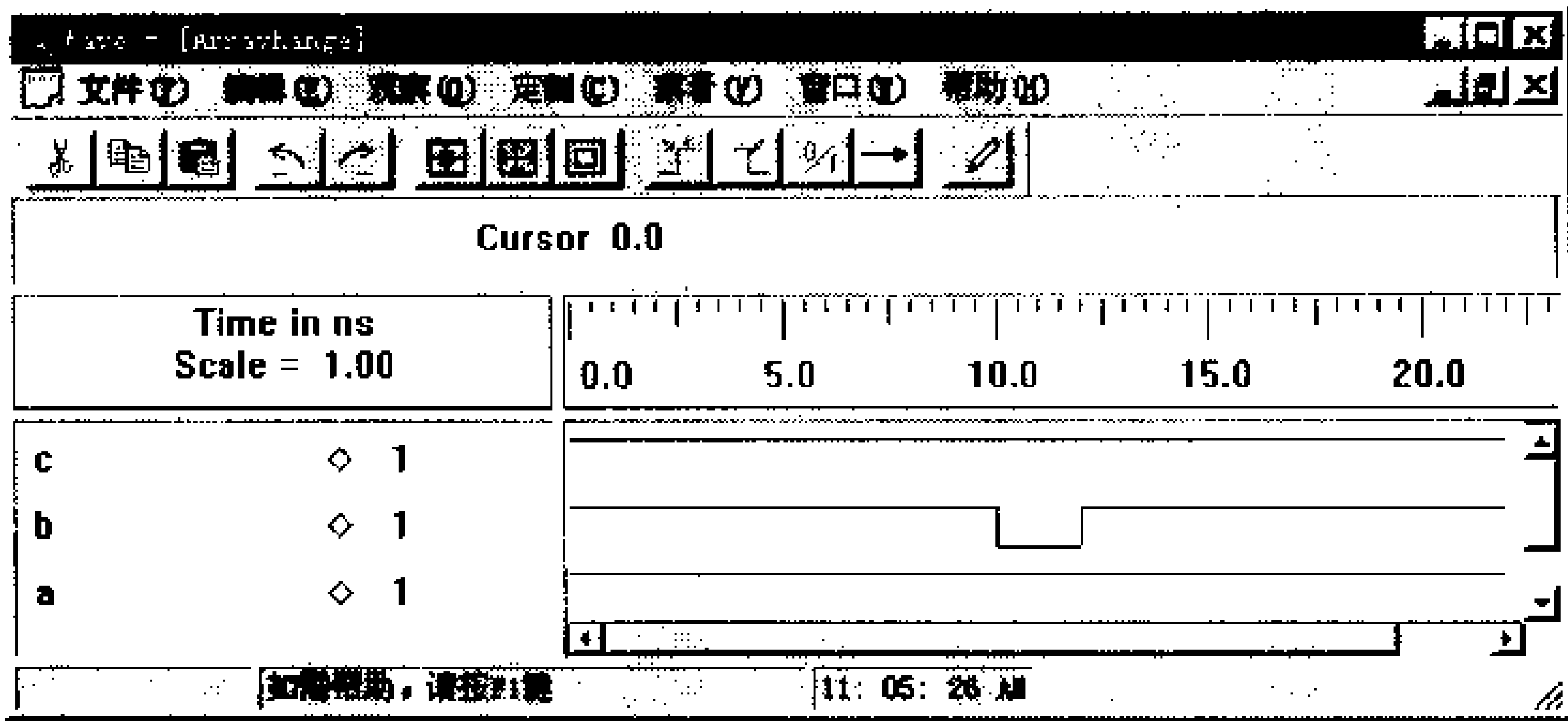


图 51.1　模拟波形

（源描述文件名：51_test_113.vhd）

第52例　多倍（次）分频器

刁岚松

1．电路系统的工作原理

本例是用 VHDL 语言来描述一个分频器。根据端口 SPD 的值，电路可作为二分频、四分频、八分频等分频器。每经过一个输入时钟周期，信号 Cnt 的值就增加 1，所以 Cnt 的最低位每经过两个输入时钟周期发生一次周期变化，次低位每经过 4 个输入时钟周期发生一次周期变化，其他位类推。将 Cnt 不同位的值输出，就能产生不同的分频器。

2．电路的 VHDL 语言描述方法及语法分析

部分描述如下所示：

```
library ieee;
use ieee.Std_logic_1164.all;
use ieee.Std_logic_arith.all;

entity DIVIDER is
    port ( CLK_IN   : in Std_logic;          --时钟信号输入端口
           RESET    : in Std_logic;          --复位信号
           SPD      : in Integer;            --分频器输出周期设定位
           CLK_OUT  : out Std_logic);        --分频器输出位
end;

architecture RTL of DIVIDER is
    signal Cnt: Std_ULogic_Vector (7 downto 0);
begin
    DIVIDE: process (CLK_IN, RESET)
begin
    if RESET = '1' then
        Cnt <= "00000000";
    elsif CLK_IN'EVENT and CLK_IN = '1' then
    --每当 CLK_IN 由其它值变为 '1' 时，Cnt 加 '1'
        Cnt <= Cnt + "1";
    end if;
end process;
```

```
    SELECT SPEED: process (Cnt, SPD)
    begin
    --根据 SPD 不同的值将 Cnt 不同的值输出，产生不同周期的输出信号
    case SPD is
        when 0 => CLK_OUT<=Cnt(0);
        when 1 => CLK_OUT<=Cnt(1);
        when 2 => CLK_OUT<=Cnt(2);
        when 3 => CLK_OUT<=Cnt(3);
        when 4 => CLK_OUT<=Cnt(4);
        when 5 => CLK_OUT<=Cnt(5);
        when 6 => CLK_OUT<=Cnt(6);
        when 7 => CLK_OUT<=Cnt(7);
        when others=> CLK_OUT<=Cnt(7);
    end case;
end process;
end;
```

语法分析：

case 语句的一般格式如下：

```
[case 标号：]case 表达式 is
    {when 表达式值=>顺序语句};
    [when others=>顺序语句;]
end case [case 标号];
```

3. 模拟测试向量的选择及模拟结果分析

测试台的部分描述如下：

```
P_Clk_In :
process
begin
    Clk_In  <= '0';
    --产生周期为 20ns 的输入时钟波形
    --尽管这是一个无限循环，但是在终止进程 EndProc 中有一条断言语句，
    --这条断言语句可以保证结束模拟，当然也就能结束本进程。
    while true loop
        Clk_In <= '1';
        wait for 10 ns;
        Clk_In <= '0';
        wait for 10 ns;
    end loop;
```

```
end process P_Clk_In;

P_RESET :
process
begin
    wait for 0.000 ns;      RESET <=  '1' ;
    wait for 100.000 ns;     RESET <=  '0' ;
    --100ns 以后 Cnt 的值开始变化
    wait;
end process;

    P_Spd:
    --此进程用于设定输出脉冲周期，Spd 的值越大，周斯越长。
process
begin
--0ns 到 1000ns，输出脉冲周期是输入脉冲周期的 2 倍
    wait for 0.000 ns;       Spd <=  0;
--1000ns 到 2000ns，输出脉冲周期是输入脉冲周期的 4 倍
    wait for 1000.000 ns;     Spd <=  1;
--2000ns 到 3000ns，输出脉冲周期是输入脉冲周期的 8 倍
    wait for 2000.000 ns;     Spd <=  2;
--3000ns 到 4000ns，输出脉冲周期是输入脉冲周期的 16 倍
    wait for 3000.000 ns;     Spd <=  3;
--4000ns 到 5000ns，输出脉冲周期是输入脉冲周期的 32 倍
    wait for 4000.000 ns;     Spd <=  4;
--5000ns 以后，输出脉冲周期是输入脉冲周期的 64 倍
    wait for 5000.000 ns;     Spd <=  5;
    wait;
end process;

EndProc :
process
begin
    wait for 10000 ns;
    --10000ns 以后，模拟结束
    assert false report "End of Simulation" severity error;
    wait;
    end process;
end test_bench;
```

输出的部分波形如图 52.1 所示：

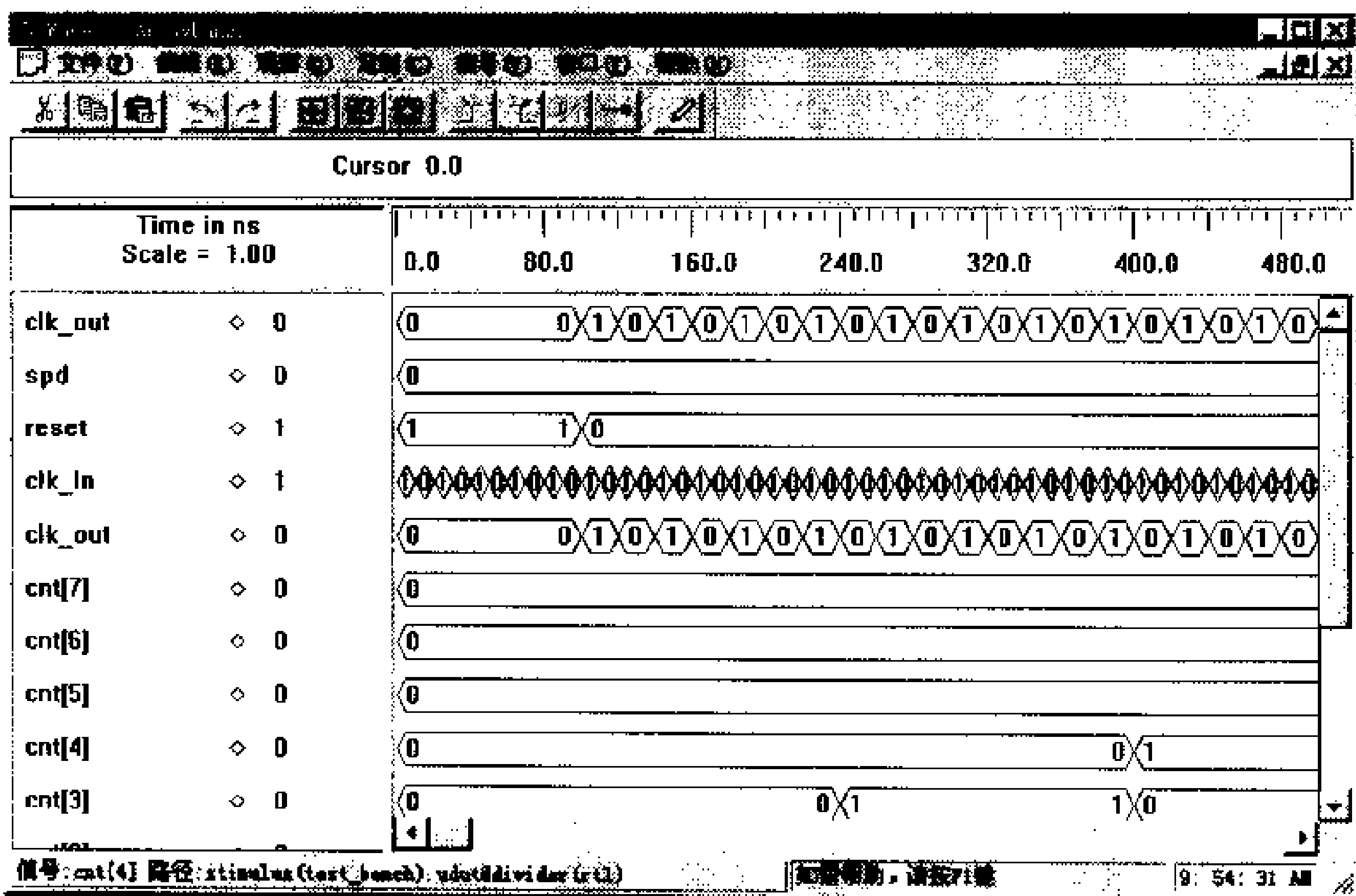


图 52.1 分频器输出波形图

在 100ns 以后，每当遇到 Clk_In 的上升沿时，Cnt 的值增加 1。如果 Spd 的值为 0，Clk_Out 的周期为 Clk_In 的周期的两倍；如果 Spd 的值为 1，Clk_Out 的周期为 Clk_In 的周期的 4 倍。

（源描述文件名：52_divider.vhd
52_divider_stim.vhd）

第53例　三位计数器与测试平台

刘沁楠

1. 测试平台

本例主要讨论描述测试平台(test-bench)的方法。大多数 VHDL 模拟器允许实时交互，即对输入端口赋值、模拟运行，借助波形图观测输出值。该过程可重复进行，直至设计者对设计模型的功能满意为止。测试平台可用于验证设计功能的正确性，它允许将输入测试向量用于一个设计（待测模块），同时通过波形观察输出测试向量，也可将其记录在输出测试文件中，或在测试平台中与预测值进行比较。

通过交互模拟，测试平台提供下述便利：

(1) 可使输入、输出测试向量易于存档。

(2) 通过交互式地输入、观察测试向量，提供一种更为有序的方法。

(3) 一旦创建一个测试平台并确定测试向量之后，在设计者改变设计的过程中，这组测试可反复利用。

(4) 相同的测试平台既可用于 VHDL 源描述的功能验证，也可对综合之后的结果进行验证。

创建测试平台主要有 3 种方法。方法一采用嵌入测试平台中的测试向量表；方法二利用包含测试向量的独立文件；方法三借助于某种算法将实际的输出结果同预期的输出结果进行比较。

2. 测试平台的描述方法

本例通过为三位计数器描述测试平台向读者介绍创建测试平台的一种常用方法——列表法。此方法的主要特点是将所有测试向量按组有序地提交给测试平台，依次对其进行测试。三位计数器源描述如下：

```
library IEEE;
use IEEE.std_logic_1164.all;
--包说明
package mycntpkg is
    component count port(clk,rst : in std_logic;
                         cnt : inout std_logic_vector(2 downto 0));
    end component;
end mycntpkg;
```

```
library IEEE;
use IEEE.std_logic_1164.all;
--实体说明
entity count is port(clk,rst : in std_logic;
                     cnt : inout std_logic_vector(2 downto 0));
end count;

library IEEE;
use IEEE.std_logic_arith.all;

--结构体
architecture archcount of count is
begin
counter:process(clk,rst)
begin
        --复位
        if rst = '1'  then
            cnt <= (others =>'0');
        --记数
        elsif (clk'EVENT and clk = '1') then
            cnt <= cnt +1;
        end if;
end process;
end archcount;
```

从其行为描述不难看出，该计数器的功能为：当复位信号为‘1’时，cnt 被复位，值为“000”；一旦时钟信号上跳变，则对 cnt 进行“+1”操作，以此实现计数的功能。针对上例创建的测试平台如下所示：

```
library IEEE;
use IEEE.std_logic_1164.all;

--测试平台（空实体）
entity testcnt is
end testcnt;

use work.mycntpkg.all;
--结构体
architecture mytest of testcnt is
    --信号说明
    signal clk,rst:std_logic;
```

```
    signal cnt:std_logic_vector(2 downto 0);
--组装
for all :count use entity work.count(archcount);
begin
--例示待测元件
uut:count port map(clk => clk,rst => rst,cnt => cnt);
--输入测试激励向量
verify:process
    variable errors:boolean := false;
    begin
--第1组测试向量，对计数器复位
        clk <= '0';
        rst <= '1';
        wait for 20 ns;
        -- 检查输出向量
        if cnt /= "000" then
        assert false
        report "cnt is wrong value";
        errors := true;
        end if;

--第2组测试向量
        clk <= '1';
        rst <= '1';
        wait for 20 ns;
-- 检查输出向量
        if cnt /= "000" then
        assert false
        report "cnt is wrong value";
        errors := true;
        end if;
    ...
--第29组测试向量
clk <= '0';
        rst <= '0';
        wait for 20 ns;
        if cnt /= "010" then
        assert false
        report "cnt is wrong value";
        errors := true;
        end if;
        --断言报告
```

```
        assert not errors
        report "Test vectors failed."
        severity note;
        assert errors
        report "test vectors passed."
        severity note;
        wait;
    end process;
end;
```

在用此法创建测试平台时，首先要尽可能详尽地提供测试向量，测试向量的选择应全面，尽可能涵盖所有可能发生的状态。本例的测试向量表如表 53.1 所示。

表 53.1　三位计数器测试向量表

序号	clk	rst	Cnt
1	'0'	'1'	"000"
2	'1'	'1'	"000"
3	'0'	'0'	"000"
4	'1'	'0'	"001"
5	'0'	'0'	"001"
6	'1'	'0'	"010"
7	'0'	'0'	"010"
8	'1'	'0'	"011"
9	'0'	'0'	"011"
10	'1'	'0'	"100"
11	'0'	'0'	"100"
12	'1'	'0'	"101"
13	'0'	'0'	"101"
14	'1'	'0'	"110"
15	'0'	'0'	"110"
16	'1'	'0'	"111"
17	'0'	'0'	"111"
18	'1'	'0'	"000"
19	'0'	'0'	"000"
20	'1'	'0'	"001"
21	'0'	'0'	"001"
22	'1'	'0'	"010"
23	'0'	'1'	"000"
24	'1'	'1'	"000"
25	'0'	'0'	"000"
26	'1'	'0'	"001"
27	'0'	'0'	"001"
28	'1'	'0'	"010"
29	'0'	'0'	"010"

上述测试向量针对三位计数器可能出现的各种情况，如复位、加一计数等。其中第1～3组测试向量用于对计数器复位的检测；第4～22组测试向量用于检测计数器的加一计数功能；第23～25组测试向量用于检测复位功能；第26～29组测试向量用于计数功能的检测。

3. 测试平台的语法分析

每个测试平台都是一个空实体，它不定义任何输入输出端口与外界联系，本例中的实体 testcnt 就是这样一个空实体。

在测试平台中，需要说明一系列的信号，这些信号对应于待测元件的端口，通常信号名与相应的局部端口名取为一致。测试平台向这些信号施加激励，并利用元件例示将激励信号引到元件的输入端、将元件输出端连到响应信号，同时，通过组装将例示元件与被测元件相连，从而引发被测元件的行为，并通过比较输出结果来验证设计元件功能的正确与否。

本例中说明的内部信号 clk，rst，cnt 通过语句“uut: count **port map** (clk => clk, rst => rst, cnt => cnt);”分别与元件 count 的端口 clk，rst，cnt 相连；语句“**for all**: count **use entity** work.count(archcount);”将测试平台中的元件 uut 指定为待测元件，即实体 count 对应的结构体 archcount。这样，测试平台中给以的激励信号将作用于待测元件 count 并产生效果。

表 53.1 中的测试向量对应的 VHDL 描述如下（以第 1 组测试向量为例）：

```
clk <= '0';
rst <= '1';
wait for 20 ns;

if cnt /= "000" then
assert false
report "cnt is wrong value";
errors := true;
end if;
```

由于信号赋值不是立即生效，而是经过 delta 延迟之后才改变信号值，因而采用“**wait for** 20 ns;”使模拟时间向前推进。

模拟开始之前，clk，rst，cnt 初值缺省为‘u’，‘u’，“uuu”，进程 counter 执行一次，由于 **if** 语句与 **elsif** 语句的条件均不满足，因而 cnt 的值仍为“uuu”。

语句“**wait for** 20 ns;”使程序执行下一个模拟周期，此时 clk = ‘0’, rst = ‘1’；信号 clk 和 rst 的值发生变化并将激活进程 counter，此时 **if** 语句条件满足，cnt 被赋值为“000”。

20ns 后，对 cnt 的值进行判断，如果它不是预期的值，断言语句将让模拟器报错“Cnt is wrong value”，同时将 errors 赋值为 true。

依次模拟各组测试向量，过程与上述类似。待到所有测试向量都模拟完成后，断言语句将根据 errors 的值报告不同的信息。本例中如果任何一组测试向量未通过，都会报告 “Test vectors failed”，如果所有测试向量的模拟结果都与预期结果一致，则报告 “Test vectors passed”。

4. 模拟结果

对本例进行模拟，产生的波形如图 53.1 所示。显而易见，模拟结果表明该设计是正确的。

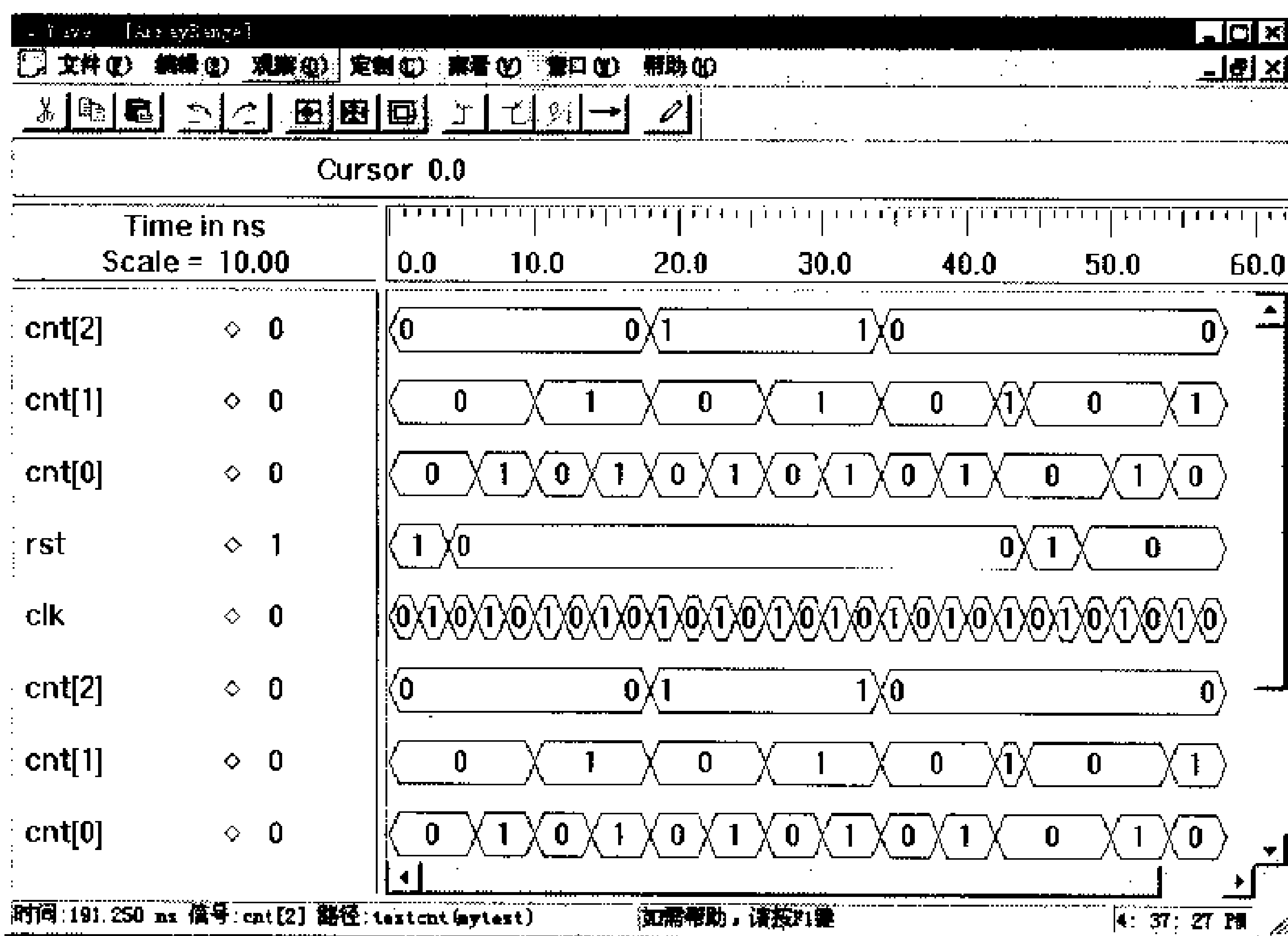


图 53.1　三位计数器模拟波形图

（源描述文件名：53_counter.vhd
测试平台文件名：53_counter_testbench.vhd）

第 54 例　分秒计数显示器的行为描述

陈东瑛

1. 电路系统工作原理

本系统由 1 个分秒计数器控制 4 个七段显示器，实现分秒计数显示器的功能，如图 54.1 所示。其中 en，reset，clk 为输入端口：en 为“计数允许”控制端，reset 为“复位”控制端，两者接收的信号均为高电平有效，起到允许记数累加和计数器清零的作用。clk 为时钟输入端，用于系统时钟上升沿同步。unit0~3 分别输出七位二进制数据到 4 个七段显示器，其对应关系如表 54.1 所示。七段显示器各段显示条对应标号如图 54.2 所示。设七位二进制数据由高位到低位对应显示条标号 6~0，显示条明亮对应数据‘0’，反之为‘1’。

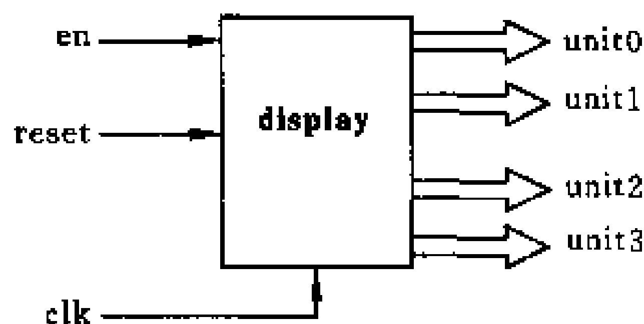


图 54.1　分秒计数显示器示意图

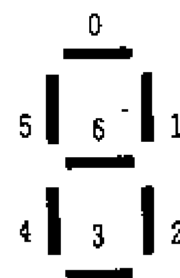


图 54.2　七段数码管显示器

表 54.1　显示数字与二进制数串的对应关系

0	1	2	3	4
1000000	1111001	0100100	0110000	0011001
5	6	7	8	9
0010010	0000010	1111000	0000000	0010000

2. 电路的 VHDL 语言描述方法及语法分析

本例的 VHDL 语言描述文件为 54_display.vhd，其中以一个实体说明 display 体现整个电路系统功能要求的接口，由一个结构体 algorithm 进行行为描述。这种描述不涉及系统具体实现的元件网表结构，而把重点放在系统功能的抽象算法上，与第 15 例，第 17 例相似。实体说明 display 的端口对应于电路系统的要求；结构体 algorithm 针对计数器的特性，采用顺序行为描述方式——进程 display 实现。结构体 algorithm 的说明

中定义个位秒数 secs、十位秒数 tsecs、个位分钟数 mins 和十位分钟数 tmins4 个计数变量。

```
--分秒计数显示器系统描述文件 54_display.vhd
entity display is
port(reset  : in bit;                                  -- 全局复位端口
     clk    : in bit;                                  -- 全局时钟信号端口
     en     : in boolean;
     unit0 : out bit_vector(6 downto 0);
     unit1 : out bit_vector(6 downto 0);
     unit2 : out bit_vector(6 downto 0);
     unit3 : out bit_vector(6 downto 0));
end display;

architecture algorithm of display is

  subtype nat4 is integer range  15 downto 0;
  subtype nat3 is integer range  7 downto 0;

begin
  display: process
    variable   secs,  mins  : nat4;
    variable   tsecs, tmins : nat3;
  begin
    secs  := 0;                                  --初始化计数变量清零
    tsecs := 0;
    mins  := 0;
    tmins := 0;
    unit0 <= "1000000";                          -- 初始化,显示器清零
    unit1 <= "1000000";
    unit2 <= "1000000";
    unit3 <= "1000000";

    RESET_LOOP: loop
wait until clk = '1' ;                              --时钟上升沿同步
exit RESET_LOOP when reset = '1' ;                  --复位控制

    case secs is                                 --根据计数变量的值向显示器赋值
        when 0 => unit0 <= "1000000";
        when 1 => unit0 <= "1111001";
        when 2 => unit0 <= "0100100";
        when 3 => unit0 <= "0110000";
```

```
        when 4 => unit0 <= "0011001";
        when 5 => unit0 <= "0010010";
        when 6 => unit0 <= "0000010";
        when 7 => unit0 <= "1111000";
        when 8 => unit0 <= "0000000";
        when 9 => unit0 <= "0010000";
        when others => unit0 <= "0000000";
    end case;

    case tsecs is
        when 0 => unit1 <= "1000000";
        when 1 => unit1 <= "1111001";
        when 2 => unit1 <= "0100100";
        when 3 => unit1 <= "0110000";
        when 4 => unit1 <= "0011001";
        when 5 => unit1 <= "0010010";
        when others => unit1 <= "0000000";
    end case;

    case mins is
        when 0 => unit2 <= "1000000";
        ... ...
        when 9 => unit2 <= "0010000";
        when others => unit2 <= "0000000";
    end case;

    case tmins is
        when 0 => unit3 <= "1000000";
        ... ...
        when 5 => unit3 <= "0010010";
        when others => unit3 <= "0000000";
    end case;

    if en then                                          --计数允许控制
      if (secs = 9) then                               --计数变量累加进位
        secs  := 0;
        if tsecs = 5 then
            tsecs  := 0;
            if mins = 9 then
                mins  := 0;
                if tmins = 5 then
                    tmins := 0;
```

```
                else
                    tmins := tmins + 1;
                end if;
            else
                mins  := mins + 1;
            end if;
        else
            tsecs  := tsecs + 1;
        end if;
      else
        secs  := secs + 1;
      end if;
    end if;
  end loop RESET_LOOP;
  end process display;
end algorithm;
```

进程 display 由初始化部分和 **loop** 循环组成，**wait until** clk ='1' 语句实现系统的时钟上升沿同步。**exit** 语句使进程在"复位有效"条件上结束，当下一时钟上升沿再次激活进程 display 时，再次执行初始化部分，实现复位清零功能。

wait until clk ='1' 语句实现时钟上升沿触发功能。从表面上看，只是时钟高电平条件下激活，其实不然。当 **wait** 语句含有敏感信号时，其功能为：先挂起进程，直到敏感信号上有事件发生才激活进程；如有条件判断，再判断条件，若成立，才激活进程。即语句 **wait until** clk'EVENT= true **and** clk='1'与上述语句 **wait until** clk ='1' 等价。当只有一个敏感信号时，可以省略对敏感信号的属性 EVENT 的判断。

3. 模拟测试向量的选择及模拟结果分析

本例的测试文件为 54_display_stim.vhd，与第 15 例和第 17 例相似，测试文件与系统描述文件分离。其中实体说明 display_stim 与结构体 stimulation 在同一文件中。

```
--分秒计数显示器测试台 54_diaplay_stim.vhd
entity display_stim is
end  display_stim ;

architecture  stimulation of display_stim is

component display_comp                                      --虚拟设计实体
port(reset    : in bit;
     clk      : in bit;
     en       : in boolean;
```

```
        unit0     : out bit_vector(6 downto 0);
        unit1     : out bit_vector(6 downto 0);
        unit2     : out bit_vector(6 downto 0);
        unit3     : out bit_vector(6 downto 0));
end componet;

    signal  reset       : bit;
    signal  clk         : bit;
    signal  en          : boolean;
    signal  unit0       : bit_vector(6 downto 0);
    signal  unit1       : bit_vector(6 downto 0);
    signal  unit2       : bit_vector(6 downto 0);
    signal  unit3       : bit_vector(6 downto 0);

begin
  comp1 :                                              --元件例示语句
 display_comp  port map(reset, clk, en, unit0, unit1, unit2, unit3);
  display_stim: process
  begin
    en     <= false;
    wait until clk = '1';
    reset     <= '1';
    wait until clk = '1';
    reset     <= '0';
    wait until clk = '1';
    en     <= true;
--通过指定次数的循环使各外部信号激励保持若干时钟周期
    for i in 0 to 2 loop                             —循环变量不需定义
      wait until clk = '1';
    end loop;
    reset    <= '1';

    for i in 0 to 1 loop
      wait until clk = '1';
    end loop;
    reset    <= '0';
    for i in 0 to 2 loop
      wait until clk = '1';
    end loop;
    en     <= false;
    for i in 0 to 1 loop
      wait until clk = '1';
```

```
    end loop;
    en     <= true;
    for i in 0 to 2 loop
      wait until clk = '1' ;
    end loop;
    wait for 90 ns;
    reset    <= '1' ;
    wait for 60 ns;
    reset    <= '0' ;
    for i in 0 to 100 loop
      wait until clk = '1' ;
    end loop;

    assert false report "End of Simulation" SEVERITY error;
   end process;

  display_clk: process
   begin
    clk   <= '0' ;
    wait for 50 ns;
    while true loop
        clk <= '1' ;
        wait for 50 ns;
        clk <= '0' ;
        wait for 50 ns;
    end loop;

     assert false                                        --整个模拟结束
     report "---End of Simulation---"
     severity error;
  end process;

end stimulation;

configuration display_stim_conf of  display_stim is          --组装语句
  for stimulation
    for comp1 : display_comp use entity work.display(algorithm);
    end for;
  end for;
end display_stim_conf;
```

结构体 stimulation 说明一个虚拟设计实体——元件 display_comp，其行为通过组

装语句指定系统描述中的设计实体 display 实现，与第 17 例中以组装规定建立元件与实体之间的联系等效。

进程 display_stim 产生一串专门验证本系统行为功能的测试向量，并以断言语句有效地结束模拟（不仅是该进程本身），因此进程 display_clk 产生时钟振荡的死循环不会造成模拟的死循环。

以 Vsim/Talent 模拟运行本测试文件，波形如图 54.3 及图 54.4 所示。需要注意以下几点：

① 系统在 clk 上升沿查看信号 reset 和 en 的值，若 reset 或 en 发生跳变，系统接收的是它们跳变之前的值，体现了 VHDL 语言的 delta 延迟语法现象。波形中(1)-(2)、(1)'-(2)'、(3)-(4)、(3)'-(4)'的延迟验证了这一点，也体现系统的清零和计数允许功能。

② 若以单步方式模拟运行，每当信号 clk 有上跳变化时，程序运行转入文件 54_display.vhd 的结构体 algorithm 部分，进程 display 激活；当进程 display 挂起时，运行窗口又恢复为测试文件 54_display_stim.vhd，即实现激励产生器与在测模块的联系。

③ 波形中(5)-(6)和(7)-(8)的延迟体现结构体 algorithm 中的“先对计数器累加，下次循环显示计数结果”的算法特点。

④ 初始化和开始计数后显示的第 1 个数字都是“零”——“1000000”，波形中(9)和(2)是初始化，(10)和(2)'是开始计数后显示的第 1 个数字。

⑤ 波形中(11)体现计数的进位功能。

设计全面合理的测试向量对检验系统描述的正确性、完备性具有重要的作用。它还能形象地表现系统描述的算法特点，对指导用户改进设计描述有很大的帮助。模拟的任务也正在于此。

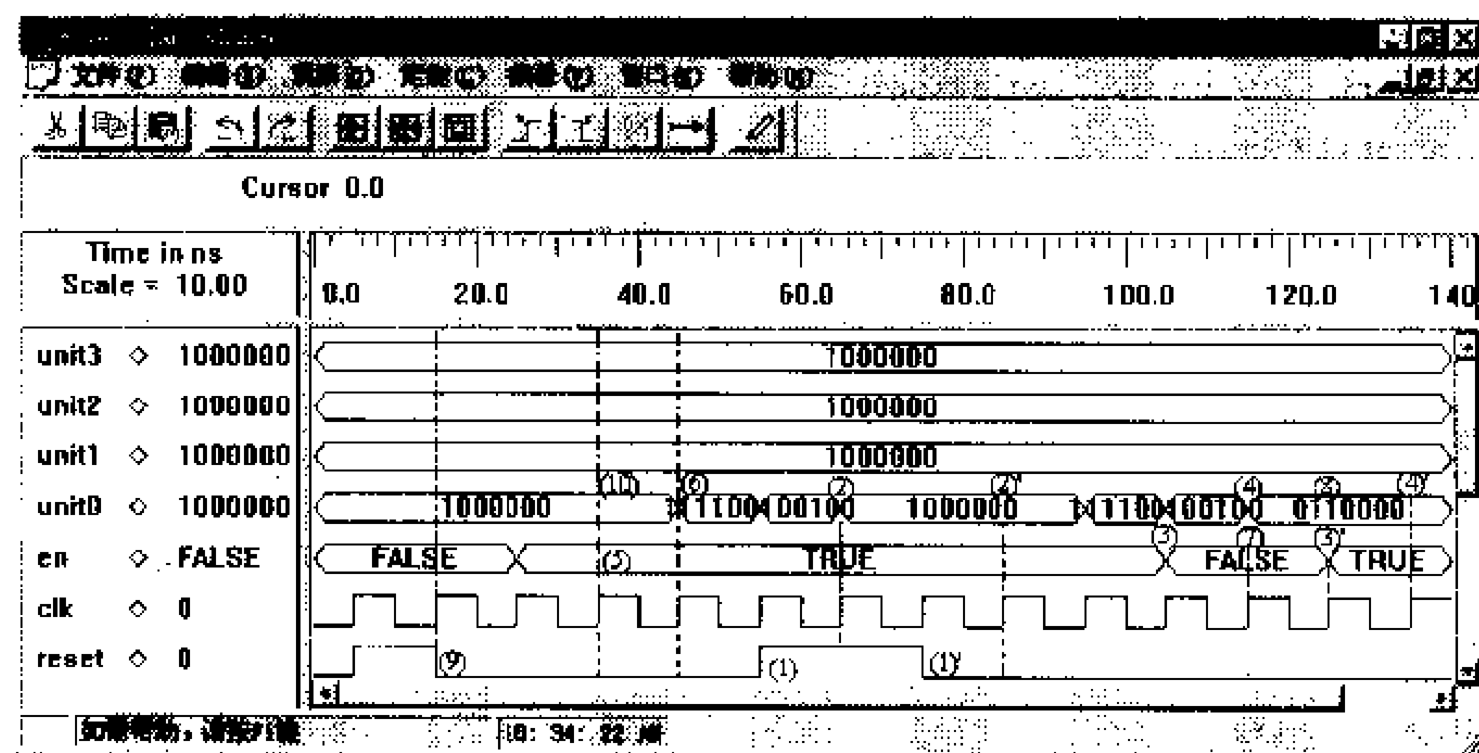


图 54.3 模拟波形

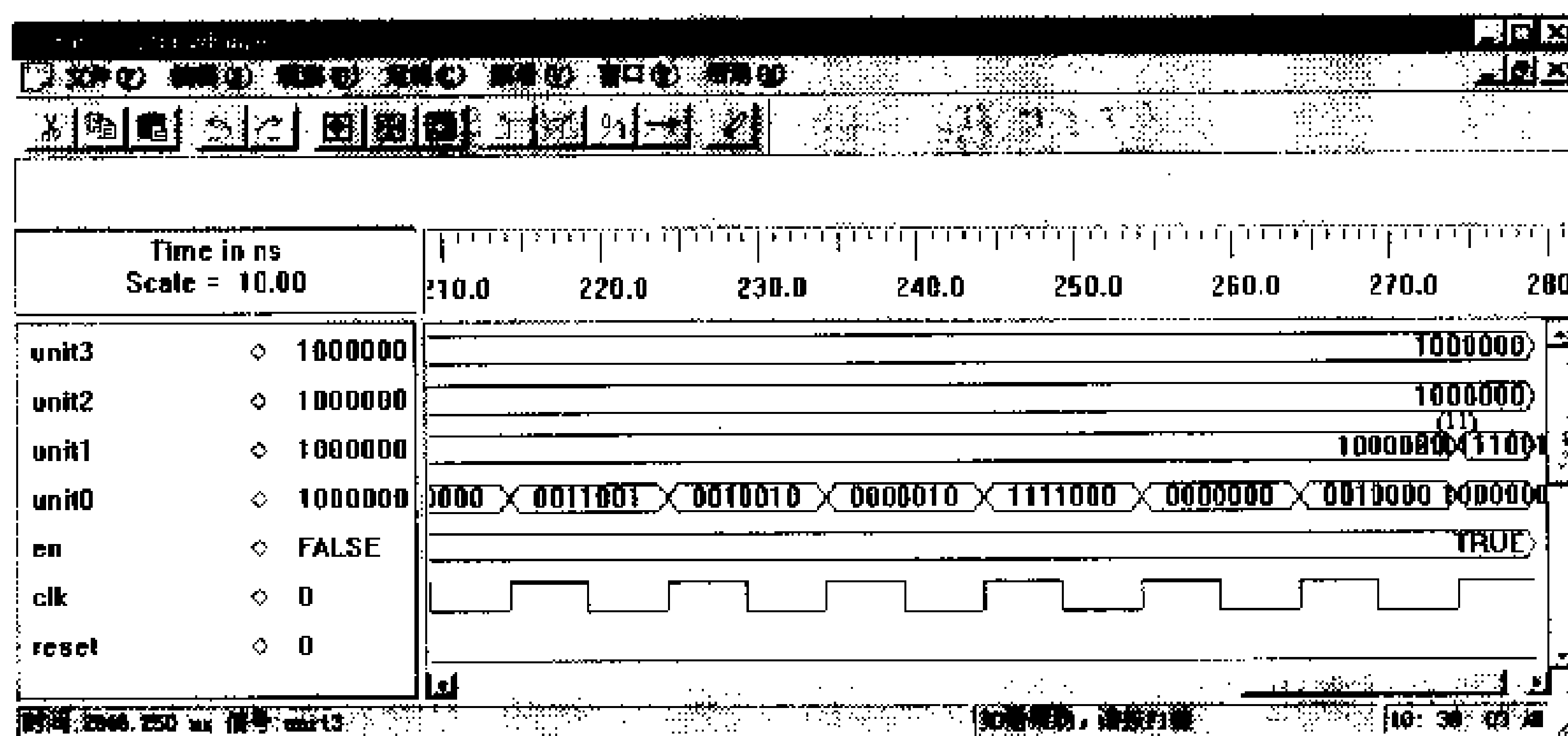


图 54.4　模拟波形

（源描述文件名：54_display.vhd
测试平台文件名：54_display_stim.vhd）

第55例 地址计数器

陈东瑛

1. 电路系统工作原理

本系统的功能是根据指定的寻址方式和形式地址产生相应的有效地址值，如图 55.1 所示。其中输入端 clk 用于产生时钟以支持系统工作在时钟上升沿同步。当 reset 端口接收清零信号（高电平有效）时，置输出信号"零"到 maddr 端口。端口 datainp 接收关于寻址方式的信息，相应的寻址方式见表 55.1。

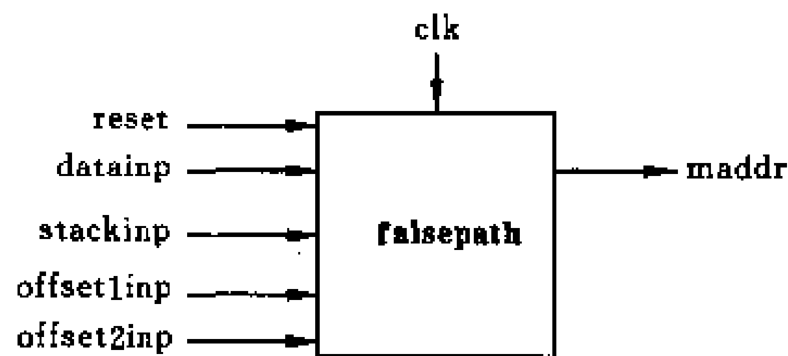


图 55.1 地址计数器芯片示意图

表 55.1 寻 址 方 式

datainp	寻址方式
0	maddr = offset1inp - offset2inp
1	maddr = stackinp + 1
2	maddr = offset2inp
3	maddr = stackinp + offset1inp

2. 电路的 VHDL 语言描述方法及语法分析

本例的系统描述文件为 55_falsepath.vhd，实体说明 falsepath，其端口与系统功能要求的接口相一致；对应一个结构体 algorithm，采用顺序行为描述方式——进程 falsepath 实现。进程 falsepath 的主体为 reset_loop 循环，其中的 **wait** 语句实现系统的时钟上升沿同步和适当延迟，而 **exit** 语句实现清零作用——若信号 reset 有效，跳出循环，并结束进程，下一个 clk 信号上升沿再次激活进程，初始化部分对 maddr 清零。

```
--地址计数器系统描述文件 55_falsepath.vhd
    package types is
    subtype  nat16 is integer range 0 to 65535;
function conv_bit_vector(arg:in integer ; length : integer)
                    return bit_vector;
end types;

package body types is      --整型数据到位型数据的转换函数
function conv_bit_vector(arg: in integer; length: integer)
    return bit_vector is
    variable result: bit_vector(length-1 downto 0);
    variable temp: integer;
    variable i: integer := 0;
begin
    if arg < 0 then
        assert false
        report "you could get problems by converting
                a negative integer to bit_vector"
        severity warning;
        result := not(conv_bit_vector(-arg-1, length));
    else
        temp := arg;
        while temp /= 0 and i <= length-1 loop
            if temp mod 2 = 1 then
                result(i) := '1';
            else result(i) := '0';
            end if;
            temp := temp / 2;
            i := i + 1;
        end loop;
        if i > length-1 then
            assert false
            report "length is not extensive enough,
                    bit_vector truncated to length"
            severity warning;
        else
            while i  <= length - 1 loop
                result(i) := '0';
                i := i + 1;
            end loop;
        end if;
    end if;
```

```
        return result;
    end conv_bit_vector;
    end types;

    use work.types.all;

    entity falsepath is
    port(reset          : in bit;
         clk            : in bit;
         stackinp       : in nat16;
         datainp        : in nat16;
         offset1inp     : in nat16;
         offset2inp     : in nat16;
         maddr          : out nat16);
    end falsepath;

    architecture algorithm of falsepath is
    begin
    falsepath: process
         variable stack   : nat16;
         variable data    : nat16;
         variable offset1 : nat16;
         variable offset2 : nat16;
         variable ir      : nat16;
         variable addr    : nat16;
         variable p       : bit_vector(0 to 1);
         variable z       : bit;
    begin
         maddr <= 0;
         RESET_LOOP: loop
              stack  := stackinp;
              data   := datainp;
              offset1 := offset1inp;
              offset2 := offset2inp;
              wait until clk'EVENT and clk = '1' ;
              exit RESET_LOOP when (reset = '1' );
              wait until clk'EVENT and clk = '1' ;
              exit RESET_LOOP when (reset = '1' );
              ir := data + 1;
              p  := conv_bit_vector(ir,2);
              case p is                                    --选择寻址方式
                   when "00" =>
```

```
                addr := stack + offset1;
                z := p(0);
            when "01" =>
                addr := offset1;
                z := p(1);
            when "10" =>
                addr := stack + 1;
                z := p(1);
            when "11" =>
                addr := 0;
                z := p(0);
        end case;
        if z = '1' then
            maddr <= addr + offset2;
        else
            maddr <= addr;
        end if;
    end loop RESET_LOOP;
    end process falsepath;
    end algorithm;
```

对本例系统描述需要注意以下几点：

① 因为进程 falsepath 中寻址方式的选择是通过对两位二进制数据的分析实现的，所以需要用包 types 中定义的函数 conv_bit_vector(x, len) 对端口 datainp 输入的整型信号进行预处理。

② 进程 falsepath 中定义了与实体说明 falsepath 的数据端口一一对应的变量，在循环开始时，把端口的信号值赋给变量，对变量进行操作，最后把有效地址变量 addr 的值再赋给 maddr 端口，这样可以避免与信号操作相关的 VHDL 语言的 delta 延迟问题。

③ 本例系统描述只考虑系统的固有行为，未涉及外部环境激励的产生。

3. 模拟测试向量的选择及模拟结果分析

与第 54 例相似，本例的测试文件为 55_falsepath_stim.vhd，测试文件与系统描述文件分离。其中实体说明 falsepath_stim 与结构体 stimulation 在同一文件中。结构体 stimulation 中定义了一个虚拟设计实体——元件 falsepath_comp 和一组与其端口对应的信号。这些外部激励信号通过元件 falsepath_comp 的端口，把值传送到实体 falsepath 的端口，同理，实体 falsepath 的响应反向传送到外部信号上。进程 falsepath_stim 产生测试向量，进程 falsepath_clk 产生时钟。整个模拟由进程 falsepath_stim 中最后一条断言语句结束。

```
--地址计数器测试台描述文件 55_falsepath_stim.vhd
use work.types.all;
entity falsepath_stim is
end falsepath_stim ;

architecture  stimulation of falsepath_stim is

component falsepath_comp                                   --虚拟设计实体
port(     reset       : in bit;
          clk         : in bit;
          stackinp    : in nat16;
          datainp     : in nat16;
          offset1inp  : in nat16;
          offset2inp  : in nat16;
          maddr       : out nat16);
end component;

    signal  reset       :  bit;                  -- 外部激励信号
    signal clk          :  bit;
    signal stackinp     :  nat16;
    signal datainp      :  nat16;
    signal offset1inp   :  nat16;
    signal offset2inp   :  nat16;
    signal maddr        :  nat16;

begin
--元件例示语句
comp1 : falsepath_comp  port map (reset, clk, stackinp, datainp,
                                   offset1inp, offset2inp, maddr);
--产生外部信号激励
falsepath_stim: process
begin
    stackinp <= 4;
    offset1inp <= 8;
    offset2inp <= 2;
    datainp <= 3;
    wait until clk' EVENT and clk = ' 1' ;
    reset <= ' 1' ;
    wait until clk' EVENT and clk = ' 1' ;
    reset <= ' 0' ;
    wait until clk' EVENT and clk = ' 1' ;
    datainp <= ' 0' ;
```

```
        wait until clk' EVENT and clk = ' 1' ;
        wait until clk' EVENT and clk = ' 1' ;
        assert (maddr = 12)                                              --自判断
        report "ERROR: Signal maddr should be 12"                        --出错提示
        severity error;

        datainp <= 1;
        wait until clk' EVENT and clk = ' 1' ;
        wait until clk' EVENT and clk = ' 1' ;
        assert (maddr = 10)                                              --自判断
        report "ERROR: Signal maddr should be 10"                        --出错提示
        severity error;

        datainp <= 2;
        wait until clk' EVENT and clk = ' 1' ;
        wait until clk' EVENT and clk = ' 1' ;
        assert (maddr = 5)                                               --自判断
        report "ERROR: Signal maddr should be 5"                         --出错提示
        severity error;

        wait until clk' EVENT and clk = ' 1' ;
        wait until clk' EVENT and clk = ' 1' ;
        assert (maddr = 2)                                               --自判断
        report "ERROR: Signal maddr should be 2"                         --出错提示
        severity ERROR;

        assert false report "End of Simulation" severity error;  --完成模拟
    end process falsepath_stim;

    falsepath_clk: process                                               --产生时钟振荡
    begin
        clk   <= ' 0' ;
        while TRUE loop
            clk <= ' 1' ;
            wait for 50 ns;
            clk <= ' 0' ;
            wait for 50 ns;
        end loop;
    end process falsepath_clk;

    end stimulation;
```

```
--组装语句
configuration falsepath_stim_conf of  falsepath_stim is
for stimulation
     for compl : falsepath_comp use entity work.falsepath(algorithm);
     end for;
end for;
end falsepath_stim_conf;
```

由 Vsim/Talent 模拟本测试文件，波形如图 55.2 所示，其过程分析如下:零时刻元件例示语句 compl、进程 falsepath_stim 和 falsepath_clk 同时开始运行，同时也调用了实体 falsepath(元件例示语句和组装语句产生的联系)，其结构体内的进程 falsepath 也同时开始运行，maddr 初始化清零。进程 falsepath 和 falsepath_stim 都挂起于第 1 条 **wait** 语句，直到 falsepath_clk 进程产生第 1 个时钟上升沿，才重新激活。由于信号 clk 在执行 clk<=‘1’语句后一个 delta 延迟后才变成‘1’，即进程 falsepath_stim 和 falsepath 在零时刻经过一个 delta 延迟后，外部信号的初始化值才真正赋给对应端口并传给变量，此时信号 reset 还未变成‘1’，所以进程 falsepath 进入第 2 条 **wait** 语句的挂起状态。再次激活时，由于 reset=‘1’（虽然此时进程 falsepath_stim 已经运行到 reset<=‘0’语句，但是其值在一个 delta 延迟之后才会变化，如图 55.2 中的（1）所示）而跳出循环，结束进程。第 3 个时钟上升沿激活进程 falsepath，maddr 初始化清零，变量 data 及时接收到外部信号 datainp 的第 1 个值 3（虽然此时进程 falsepath_stim 已经执行到 datainp<=0 语句，但是其值在一个 delta 延迟之后才会变化，如图 55.2 中的（2）所示），所以第 4 个时钟上升沿到来时，对应寻址方式 3 的有效地址值 12 能够输出给 maddr，如图 55.2 中的（3）所示，同时下一轮循环开始，各变量接收 datainp 的第 2 个值 0，此后第 2 个时钟上升沿到来时，输出其对应有效地址值 10，此后同理输出 5 和 2。

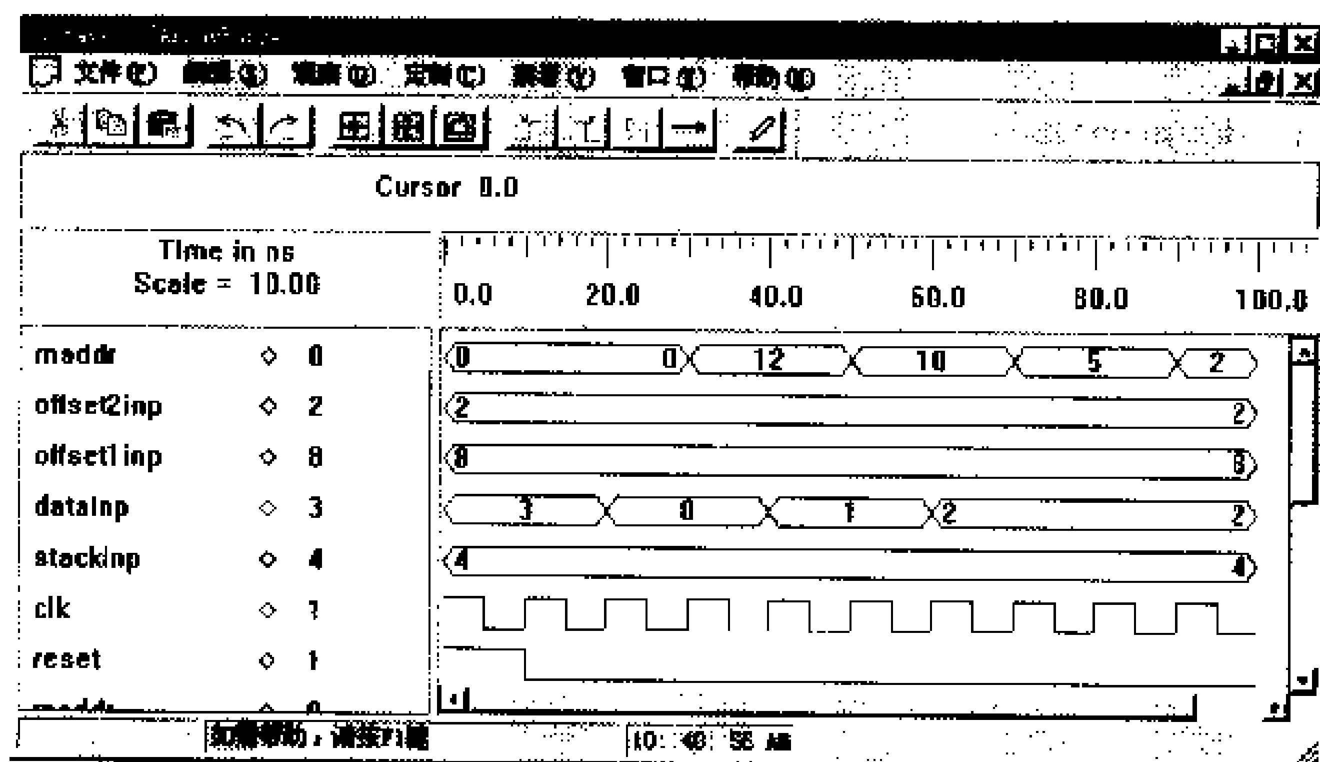


图 55.2 模拟波形

本例测试文件加入自判断功能，在期望输出有效地址值的时刻，插入判断地址值的断言语句和出错提示。这些断言语句的严谨性级别为 error，如果用 warning 代替，就不会出现因一个错误而打断整个模拟，使其他部分得不到测试的现象。

（源描述文件名： 55_falsepath.vhd
测试平台文件名：55_falsepath_stim.vhd）

第 56 例 指令预读计数器

吴清平

1. 电路系统工作原理

本设计实现微处理器的程序计数器的数值控制：计数器不断指向下一条将要执行的指令。

2. VHDL 语言描述方法及语法分析

(1) 包分析

描述中定义了一个包（**package**），包的名字为 types，在此包中定义一个子类型 short，它的基类型是整型，此数据类型的取值范围是 0~255。此类型的定义语句如下：

```
subtype short is integer range 0 to 255
```

设计中的程序计数器的类型是 short，所以该计数器的取值范围是在 0~255 之间。

(2) 实体 prefetch 分析

该实体定义一个程序计数控制器，它包括 8 个端口：

branchpc：端口类型为 **in**，数据类型是在包 types 中定义的 short 型。它的内容是当指令发生跳转时，下一条指令的地址。由于类型 short 的范围是 0~255，所以在该设计中的程序计数器的范围也是 0~255。

ibus ：端口类型为 **in**，数据类型是 short，它对应输入数据总线。

branch ：端口类型为 **in**，数据类型是 bit，此位指示当前程序计数器是否跳转，当执行一条跳转指令时，将置 branch 为‘1’，并且使 branchpc 等于要跳转到的地址。

ppc ：端口类型为 **in**，数据类型是 short，指示当前的程序计数器值。

popc ：端口类型为 **out**，数据类型是 short，指示上一条指令的地址。

obus ：端口类型为 **out**，数据类型是 short，相当于输出数据总线。

(3) 结构体 behavioral 分析

这是实体 prefetch 的结构体，描述如下：

```
architecture behavioral of prefetch is
begin
    process
        variable pc     : short;
        variable oldpc  : short;
        begin
            ppc  <= pc;
            popc <= oldpc;
            obus <= ibus + 4;
            if (branch='1') then pc := branchpc;
            end if;
            wait until (ire = '1');
            oldpc := pc;
            pc := pc + 4;
    end process;
end behavioral;
```

这是描述纯行为的结构体，其中包含一个进程。在进程中定义两个变量 pc 和 oldpc，其数据类型是 short 型，进程一开始将变量 pc 的值送给 ppc，将 oldpc 的值送给 popc，其中输出 ppc 即为当前程序计数器的值，而 oldpc 的值则是前一条指令的值。而后进程将判断输入端 branch 的值是否为'1'，如果为'1'，即此条指令为跳转指令，则 branchpc 的值将被赋给 pc，而下一个 ire 的上升沿时，变量 pc 的值将被送至 ppc 端口。

(4) 测试平台分析

文件 56_vhdl.vhd 中包括一个空实体，它所对应的结构体是前面设计单元的一个测试平台。

文件 56_stim.vhd 中包含一个结构体，它是文件 36_vhdl.vhd 中的空实体所对应的结构体，描述一个测试平台，用作对设计单元 prefetch 的测试向量。测试平台的编写步骤一般分为如下几步：

① 在结构体的说明部分说明一个元件，此元件只是一个虚拟概念。

② 使用组装语句将此元件与一个真正的设计单元，即与要测试的实体相关联。

③ 在结构体的语句部分使用此元件例示一个真正的元件。

在此结构体说明部分首先说明一个元件，即说明要进行测试的元件。同时说明几个信号作为将要测试元件的输入。

在结构体的语句部分包含有：一条元件例示语句，一条激励信号赋值语句和一条并发信号赋值语句。

元件例示语句例示元件 prefetch_I，此元件由组装语句将其与 prefetch 实体相联系。激励信号赋值进程为元件的端口加激励。并发信号赋值语句在 IRE 上产生一个周期

为 10ns 的方波。

3. 模拟结果

本例模拟结果波形图如图 56. 1 所示。

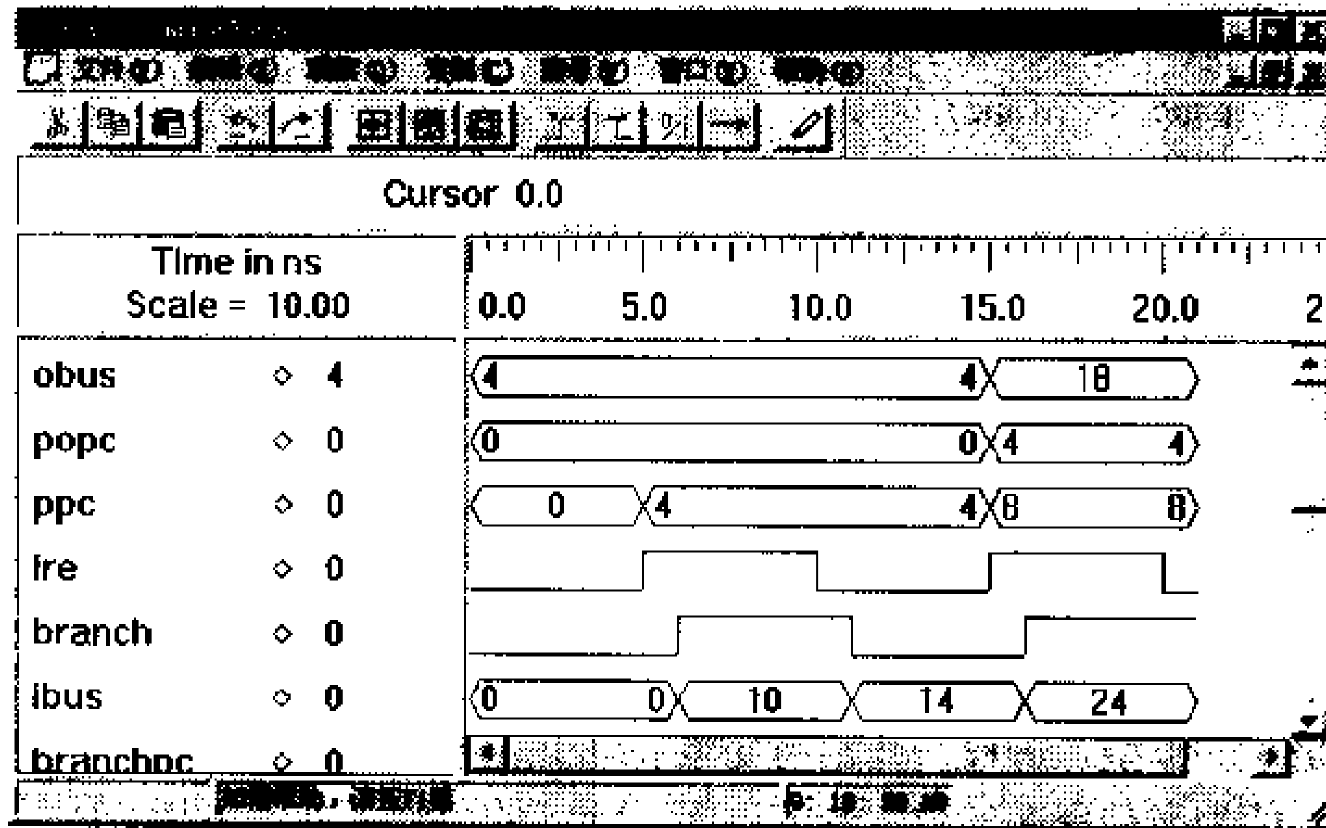


图 56.1 模拟波形

（源描述文件名：56_prefetch.vhd
56_stim.vhd
56_vhdl.vhd）

第 57 例　加、减、乘指令的译码和操作

吴清平

1. 电路系统工作原理

此译码器完成指令译码。在此描述完成指令所指定的功能，其中包含 3 种指令：加法、减法和乘法，分别对两个操作数 count_a 和 count_b 完成加法、减法和乘法。

2. VHDL 语言描述方法及语法分析

(1) 实体 test_67b 分析

该例将功能单元与测试平台写在一起，即将功能部件和激励向量均写在同一个结构体中，因此在实体中未定义任何端口，使此实体成为一个空实体。

(2) 结构体 Behave_1 分析

此结构体是实体 test_67b 所对应的结构体，它包含一个进程和一条并发信号赋值语句。其中并发信号赋值语句用于给出将要执行的指令，而后由进程接收指令，并完成指令译码工作，最后完成实际的指令所要求的工作。即进程是功能单元，完成译码工作；而并发信号赋值语句为测试台部分，它给进程增加激励。

在结构体的声明部分定义枚举数据类型 Instruction_Enum：

```
type Instruction_Enum is (ADD,SUBTRACT,MULTIPLY);
```

用它表示指令类型，此枚举类型包含 3 个枚举值：ADD，SUBTRACT 和 MULTIPLY，分别表示加法、减法和乘法。

① 进程分析（功能单元）

```
Proc_Execute:
process
     variable Count_A   : natural := 5;
     variable Count_B   : natural := 2;
     variable Result    : natural := 0;
variable Command_Enum : Instruction_Enum;
begin
     wait on Command_Nat;
     if (Command_Nat >= 0) and (Command_Nat <= 2) then
```

```
            Command_Enum := Instruction_Enum' VAL(Command_Nat);
        else
            assert false
            report "Invalid command code sequence"
            severity error;
        end if;
        case Command_Enum is
            when ADD       => Result := Count_A + Count_B ;
            when SUBTRACT  => Result := Count_A - Count_B ;
            when MULTIPLY  => Result := Count_A * Count_B ;
        end case;
    end process Proc_Execute;
```

在进程中定义 4 个变量，其中变量 Count_A 和 Count_B 为两个操作数，变量 Result 用于存储计算结果，它们都是 natural（自然数），变量 Command_Enum 用来存储指令译码后得到的指令编码。

进程的第 1 条语句 **wait on** Command_Nat 是一条等待语句。它将使进程等待信号 Command_Nat 上发生事件，如 Command_Nat 上无事件，则进程处于挂起状态；一旦 Command_Nat 上发生事件，即有新的指令到来，进程则被激活。其后，首先要判断此指令是否合法，即它的值是否在 0~2 之间，其中 0，1，2 在译码之后将分别译为 ADD，SUBTRACT 和 MULTIPLY。如指令合法，则对其进行译码，译码采用如下语句完成：

```
Command_Enum := Instruction_Enum' VAL(Command_Nat);
```

其中 Command_Nat 为原指令，Command_Enum 用于存储译码后的编码，译码时使用了在说明部分定义的数据类型 Instruction_Enum，使用预定义的属性 VAL，属性 VAL 用于取得枚举数据类型在指定位置的值，如：

```
Instruction_Enum' VAL (0) = ADD
Instruction_Enum' VAL (1) = SUBTRACT
Instruction_Enum' VAL (2) = MULTIPLY
```

与此对应，预定义属性 POS 能够取得一个枚举数据在它的枚举数据类型中的位置，如：

```
Instruction_Enum' POS (ADD)       = 0
Instruction_Enum' POS (SUBTRACT)  = 1
Instruction_Enum' POS (MULTIPLY)  = 2
```

译码后的编码存放在变量 Command_Enum 中，其后我们使用条件信号赋值语句完成各

条指令的功能。

```
case Command_Enum is
    when ADD          => Result := Count_A + Count_B ;
    when SUBTRACT     => Result := Count_A - Count_B ;
    when MULTIPLY     => Result := Count_A * Count_B ;
end case;
```

并将结果存放在变量 Result 中。

② 并发信号赋值语句分析（测试台）

此结构体的并发信号赋值语句用于模拟指令序列

```
Command_Nat <=1 after 10 ns,
              0 after 20 ns,
              2 after 30 ns,
              0 after 40 ns,
              1 after 50 ns ;
```

模拟每隔 10ns 发送一条指令，根据语句可以知道指令序列为：减法、加法、乘法、加法、减法。此并发信号赋值语句将不断激活上面的进程以完成译码和操作的功能。

3. 模拟结果分析

本例通过模拟使用变量浏览器可观察各个变量的值（因为描述中使用的不是信号，而是变量，所以不能用波形编辑器查看模拟结果）。通过变量浏览可以发现变量 Result 的值等于根据信号 Command_Nat 的值对变量 Count_A 和 Count_B 进行加、减或乘操作的结果。

（源描述文件名：57_instruction_dec.vhd）

第 58 例　2-4 译码器结构描述

刘沁楠

1. 电路系统工作原理

图 58.1 所示为 2-4 译码器，它包含两个输入端口和 4 个输出端口。下面介绍 VHDL 结构描述的基本方法。该译码器的输入输出对应关系见表 58.1。

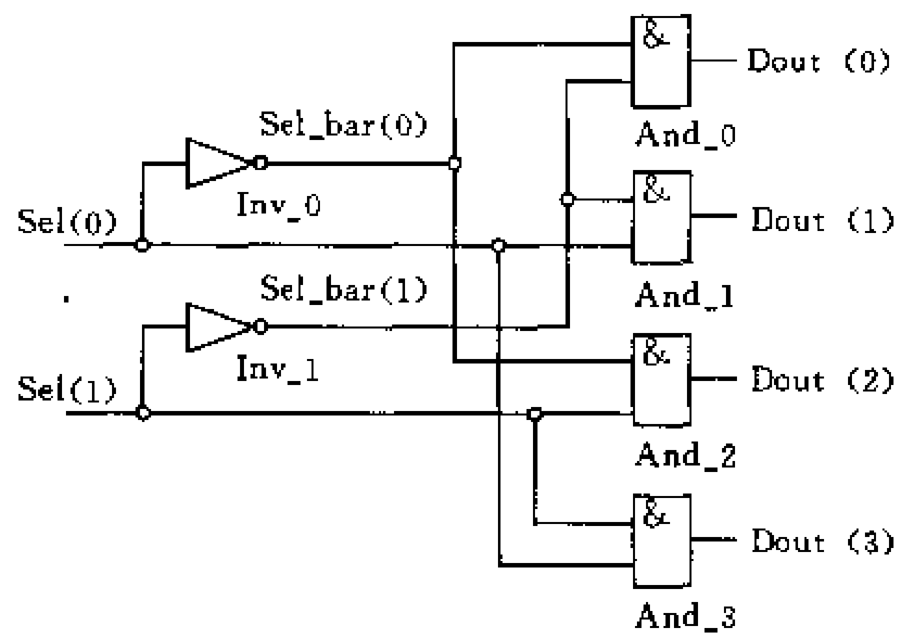


图 58.1　2-4 译码器的原理图

表 58.1　2-4 译码器 I/O 关系

输入			输出		
Sel(1)	Sel(0)	Dout(3)	Dout(2)	Dout(1)	Dout(0)
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

2. VHDL 的建模方法

在 VHDL 中，3 种最常用的描述风格为：行为描述、数据流描述和结构描述。有时也可采用由它们组成的混合描述。行为描述用于表示输入与输出间转换的行为，它不需要包含任何结构方面的信息，它描述的基本单元是进程语句；数据流描述表示行为，也隐

含表示结构（进程间的通信）；而结构描述则是用于表示构成硬件的子元件以及这些子元件之间的连接关系，它比行为描述更具体，特别是在描述与硬件的对应关系上，结构描述要比行为描述更为明显、直观，它描述的基本元件是元件例示语句。

3. VHDL 结构描述的方法

结构描述的核心问题是元件的端口以及端口之间的连接。对一个设计电路进行结构描述，通常遵循以下 3 个步骤：

第 1 步：元件说明。其一般形式为

```
component 子元件名
  [ 类属接口表；]
    端口接口表；
end component;
```

元件说明通常放在结构体的说明部分，有时也会出现在包说明中，主要用于定义局部端口，即描述子元件的外部端口特征。元件说明不同于实体说明，它不是一个可以独立编译的库元，而且它所定义的元件并不一定真正存在于设计库中。事实上，元件说明只是说明一个“模板”，它与实际实体之间的联系需要通过元件组装完成。

第 2 步：元件例示。其一般形式为

```
标识符：元件标志
  [ 类属关联表；]
        端口关联表；
```

该语句通常放在结构体内部。每个元件例示语句以一个带冒号的标识符作为标记，包括一个端口关联表，通过该关联表把实际端口与局部端口联系起来。其中的实际端口可以是在实体说明中说明的形式端口，也可以是在信号说明中定义的信号。

元件例示语句用于在模块中建立具体安装的元件。在例示中仅有子元件的外貌（名称、类型、端口模式）是可见的，子元件内部的信号是不可见的。通过元件例示来标志子元件，并且指定元件的端口和内部信号间的一一对应关系。

第 3 步：元件组装。这可以通过组装规定或组装说明完成。组装规定的一般形式为：

```
for 例示标号：元件名
                use entity 实体名[( 结构体名 )]；
```

组装说明的一般形式为

```
configuration 组装说明名 of 实体名 is
```

```
    for 结构体名
        组装规定;
    end for ;
end 组装说明名;
```

如前所述，利用元件组装将元件与实际的实体联系起来。换言之，就是为元件例示中的子元件指定其所采用的设计实体。同时，对于一个给定的实体，如果有多个可用的结构，还可以通过组装信息决定采用哪个结构。

4. VHDL 语法分析

利用上述方法对 2-4 译码器进行结构描述如下：

```
--反相器的抽象行为描述
--实体说明部分
entity B_INV is
    port(
          I1:in bit;
          O1:out bit
         );
end B_INV;
--结构体部分
architecture FUNCTION1 of B_INV is
begin
         O1 <= not I1;
end FUNCTION1;

--二输入与门的抽象行为描述
entity B_AND2 is
    port(
          I1:in bit;
          I2:in bit;
          O1:out bit
         );
end B_AND2;
--结构体部分
architecture FUNCTION2 of B_AND2 is
begin
    O1 <= (I1 and I2);
end FUNCTION2 ;

--2-4 译码器的结构描述
```

```
--实体说明
entity Decoder is
    port (
        Sel:bit_vector(1 downto 0);--输入端为二位位向量
        Dout: out bit_vector(3 downto 0) --输出端为四位位向量
        );
end Decoder;
--结构体部分
architecture Structure of Decoder is
--元件说明
    component And2
        port(I1,I2: bit;O1:out bit);
    end component;
    component Inverter
        port(I1: bit;O1:out bit);
    end component;
--信号说明
signal Sel_bar:bit_vecter(1 downto 0);
--元件组装
for Inv_0,Inv_1: Inverter use entity work.B_INV(FUNCTION1);
for all: And2 use entity work.B_AND2(FUNCTION2);
--结构体部分
begin
--利用反相器 Inverter 进行元件例示
Inv_0: Inverter
    port map (Sel(0),Sel_bar(0));
Inv_1: Inverter
    port map (Sel(1),Sel_bar(1));
--利用二输入与门 And2 进行元件例示
And_0: And2
    port map (Sel_bar(1),Sel_bar(0),Dout(0));
And_1: And2
    port map (Sel_bar(1),Sel(0),Dout(1));
And_2: And2
    port map (Sel(1),Sel_bar(0),Dout(2));
And_3: And2
    port map (Sel(1),Sel(0),Dout(3));
end Structure;

--测试平台
entity test_decoder is
end test_decoder;
```

```
--结构体
architecture BENCH of test_decoder is
     component decoder
     port (
               Sel:Bit_vector( 1 downto 0 );
               Dout:out Bit_vector( 3 downto 0 )
          );

     end component;
--组装
for I1:  decoder use entity work.decoder(structure);
     --信号说明
signal t_S : Bit_vector( 1 downto 0 );
signal t_0 : Bit_vector( 3 downto 0 );

begin
--元件例示为被测的 2-4 译码器
I1 : decoder
          port map (
                         Sel => t_S,
                         Dout =>t_0
                           );
--测试向量
driver: process
begin
     t_S <= "00";
     wait for 100 ns;
     assert(t_0 = "0001")
     report
     "Assert0  t_0 /= 0001"
     severity warning;

     t_S <= "01";
     wait for 100 ns;
     assert(t_0 = "0010")
     report
     "Assert1  t_0 /= 0010"
     severity warning;

     t_S <= "10";
     wait for 100 ns;
```

```
        assert (t_0 = "0100")
        report
        "Assert2  t_0 /= 0100"
        severity warning;

        t_S <= "11";
        wait for 100 ns;
        assert (t_0 = "1000")
        report
        "Assert2  t_0 /= 1000"
        severity warning;

        wait for 200 ns;
        assert false
        report "----End of Simulation----"
        severity error;
    end process;

    end BENCH;
```

以上为 2-4 译码器完整的结构描述（包含测试台）。开头部分分别给出反相器以及二输入与门的行为描述，这便于说明后面用到的组装语句。在 2-4 译码器结构描述的实体说明部分首先描述了该译码器的外部端口特征，结构体部分则提供了元件的内部视域。按照前面所述的结构描述方法，首先应当对子元件进行元件说明：在结构体的说明部分分别给出二输入与门和反相器的元件说明，反映出局部端口的特征；其后说明的内部信号 Sel_bar，是联接元件时所必需的；接下来在结构体部分对 2-4 译码器中的 6 个子元件分别进行元件例示。其中 Inv_0、Inv_1 是反相器类型的元件，通过端口关联表将实际端口 Sel(0)和 Sel(1)与反相器的输入端相对应，将内部信号 Sel_bar(0)和 Sel_bar(1)与反相器的输出端相对应；同样地，对 4 个两输入与门进行元件例示，通过内部信号将这些子元件相互连接。第 3 步就需要为 2-4 译码器的子元件指定它们的设计实体，即元件组装。这是由结构体说明部分的两条元件组装语句完成的。以语句“**for** Inv_0, Inv_1: Inverter **use entity** work.B_INV(FUNCTION1);”为例，表示了元件 Inv_0, Inv_1 采用的设计实体是当前工作库中的 B_INV；对应的结构体是 FUNCTION1，这是反相器的一种抽象行为描述。至此，完成 2-4 译码器的结构描述。无论系统如何复杂，只要遵循上述原则，就不难描述出电路的结构。

5. 模拟结果

借助 Vsim/Talent 对本例进行模拟，产生的波形图如图 58.2 所示，其结果表明该

2-4 译码器的功能完全正确。

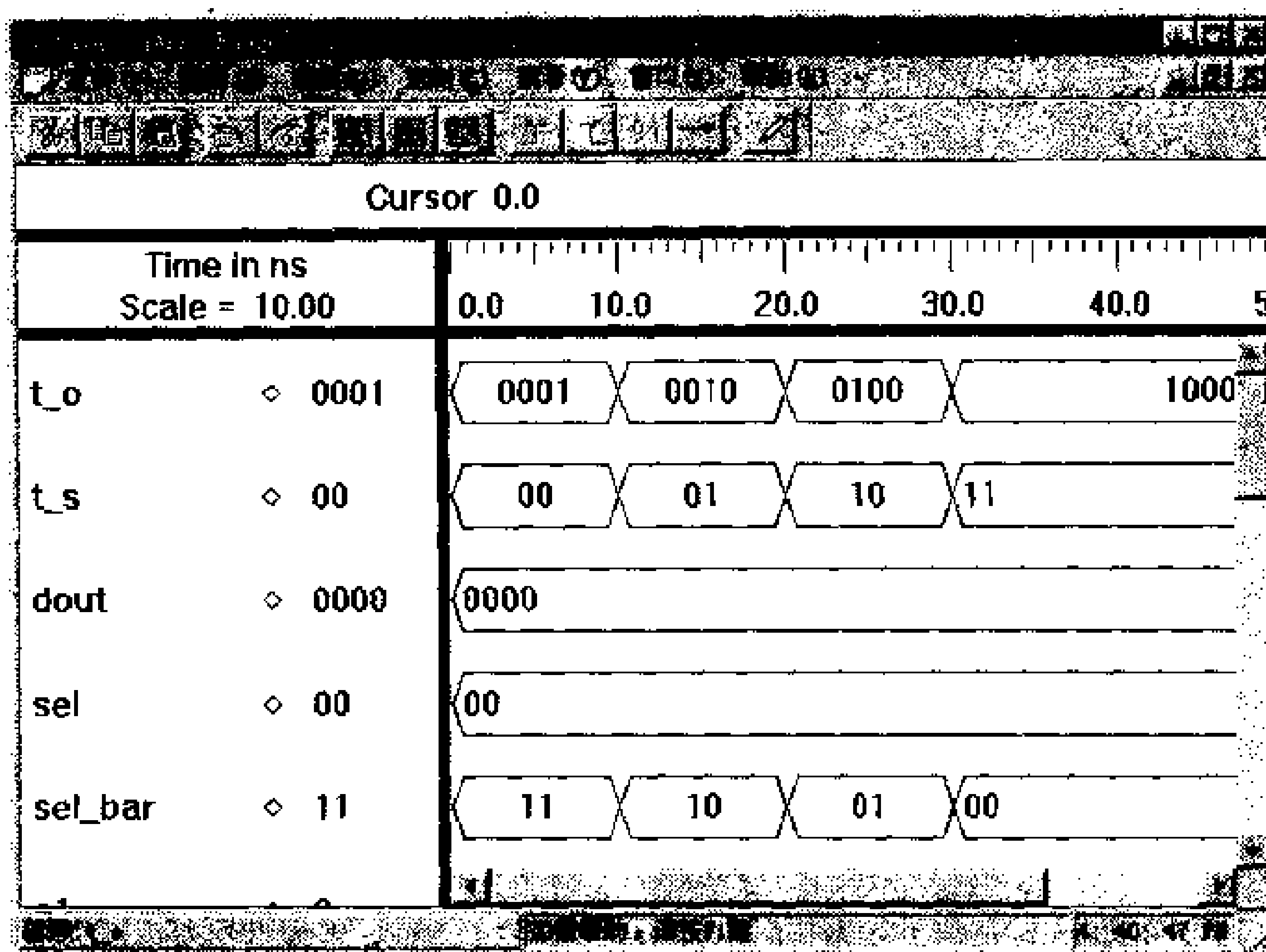


图 58.2　2-4 译码器模拟波形图

（源描述文件名：58_decoder.vhd
58_decoder_stim.vhd）

第 59 例　2-4 译码器行为描述

吴清平

1. 电路系统工作原理

在 VHDL 中，可以使用多种建模的方法，这些建模方法称为描述风格，结构描述和行为描述是其中两种重要的描述风格。本例结合一个 2-4 译码器的行为描述方法，与第 58 例的结构描述进行比较。

2. VHDL 语言描述方法及语法分析

(1) 概念阐述

结构描述和行为描述方法是 VHDL 中两种主要的描述方法。结构描述用来描述元件及元件之间的连接关系，使用结构描述的方法将各个不同的模块连接成一个新的模块。行为描述主要描述输入与输出的转换行为，描述各个元件的具体功能实现，它不包括任何结构信息。

下面分别对行为描述和结构描述进行阐述。

① 行为描述

采用自顶向下的设计方法进行集成电路设计时，经常是在设计的最初阶段采用行为描述的方法，将整个电路的基本框架结构建立起来，即将整个设计分为若干功能模块，每一个功能模块均使用行为描述语句描述其要实现的功能。在确定整个设计思路和方法正确之后，再对整个设计进行细分，将各个功能模块使用结构描述的方法具体实现，以完成设计。在对功能模块进行具体实现时，可以继续细分各个功能模块，并且 VHDL 允许在一个设计中既存在行为描述又存在结构描述，这样设计者在设计过程中将自己的结构化设计与整个设计其余部分的行为设计结合起来非常方便，将各个功能模块独立化，便于采用设计的划分及各个模块的实现。

VHDL 提供了一系列的顺序语句和并发语句，所有的顺序语句和一部分并发语句都属于行为描述。顺序语句包括十二种：变量赋值语句、信号赋值语句、**wait** 语句、**if** 语句、**case** 语句、**loop** 语句、**next** 语句、**exit** 语句、**return** 语句、**null** 语句、过程调用语句、**assert** 语句。属于行为描述的并发语句有：并发信号赋值语句、块语句、进程语句、并发 **assert** 语句、过程调用语句。

② 结构描述

结构描述方法主要用于具体实现各个模块。在设计的初期使用行为的描述方法，而最终是要得到一个设计的结构描述。结构描述中包含的是各个模块之间的连接关系，各个模块使用端口进行数据通信。结构描述的方法是使用元件例示的方法将各个模块进行连接。例示的方法分为 3 个步骤：

(a) 元件说明：在结构体说明部分中说明将要使用的元件。

(b) 元件例示：在结构体中例示上面说明的元件，并将此元件的各个端口与结构体中的某个信号相连接。

(c) 元件组装：在结构体说明部分说明的元件与设计库中已有的具体设计相连接。

通俗一些说，以上 3 步可以解释为："元件说明"说明"将要使用某一模块"；"元件例示"说明"使用了某一模块"；而"元件组装"则说明"此被使用的模块是已有设计模块中的哪一个"。

(2) 源描述分析

本例采用行为描述的方法设计一个 2-4 译码器，它们能完成相同的功能，即对输入的两个位信号进行 2-4 译码，根据不同的输入使 4 个输出端的某一个为'1'，其余为'0'。功能模块示意图如图 59.1 所示。

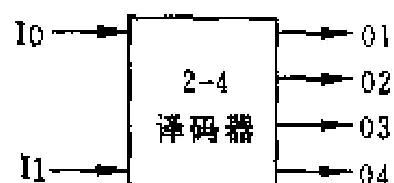


图 59.1　2-4 译码器示意图

下面是 2-4 译码器的行为描述：

```
entity decoder is
     port(    I0 : in Bit;              --输入端 0
              I1 : in Bit;              --输入端 1
              O0 : out Bit;             --输出端 0
              O1 : out Bit;             --输出端 1
              O2 : out Bit;             --输出端 2
              O3 : out Bit);            --输出端 3
end entity;
architecture decoder_archi of decoder is
     signal clk : bit := '0' ;
begin
     --四条并发信号赋值语句
     --当 I0 或 I1 发生变化时激活，同时完成对输出断的赋值
     O0 <= '1'  when I0 = '0'  and I1 = '0'
```

```
            else '0';
    O1 <= '1' when I0 = '0' and I1 = '1'
            else '0';
    O2 <= '1' when I0 = '1' and I1 = '0'
            else '0';
    O3 <= '1' when I0 = '1' and I1 = '1'
            else '0';
    --并发信号赋值语句，形成一个周期为 20ns 的方波
    clk <= not clk after 10 ns;
end;
```

该行为级的 2-4 译码器采用并发信号赋值语句完成译码，这 4 条并发信号赋值语句均是条件信号赋值语句，根据不同的输入来决定用于驱动信号的波形元素（即信号值）。

总之，行为描述和结构描述是 VHDL 中两种重要的描述方法。行为描述用来描述输入与输出的转换行为，而结构描述则用于描述各个元件的连接关系，即端口连接关系。

（源描述文件名：59_decoder.vhd）

第60例　转换函数在元件例示中的应用

王作建

1. 电路系统工作原理

图 60.1 中的与门电路功能非常简单，完成两个 std_logic 类型的输入变量的“与”：

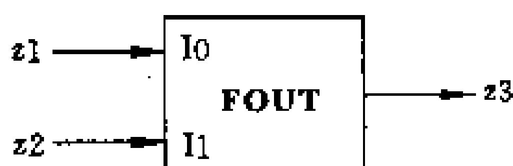


图 60.1　与门示意图

2. VHDL 语言描述方法及语法分析

如图 60.1 所示，元件与门，z1、z2 及 z3 表示测试台上的外围信号，z1、z2 为输入激励，z3 为测试台输出。需特别强调的是，z2 和 z3 是 bit 类型，而与之对应的 I1 及 FOUT 却是 std_logic 类型。由于类型不匹配而 VHDL 语言又是一种强类型语言，因此在进行元件例示时必须进行类型转换，并且对对应的一对信号，一定要转换其中的源信号。

本例给出简单的 VHDL 描述，主要目的是说明类型转换函数在元件例示中的使用方法，即转换函数进行类型转换时一定要转换驱动源。有关重要的语句说明如下：

(1) 元件说明

```
component B_AND2
          port(   I0   : in std_logic;
                  I1   : in std_logic; --端口类型为std_logic
                  FOUT : out std_logic);
end component;
```

(2) 信号说明

```
signal z1      :std_logic;
signal z2,z3   :bit; --信号类型为bit
```

(3) 元件例示语句

```
U0: B_NAND2
```

```
port map( I0=>z1,
        I1=>to_stdulogic(z2),--将 bit 类型转换为 std_ulogic
        to_bit(FOUT) => z3);—将 std_logic 类型转换为 bit
```

对于 I0 和 I1 而言，z1 和 z2 为驱动源，但 z2 的类型为 bit，而 I1 的类型为 std_logic，因此转换源 z2：to_stdulogic(z2)，将 z2 转换为 std_logic 以与 I1 匹配。

FOUT 为 z3 的驱动源，FOUT 的类型为 std_logic，z3 为 bit 类型，因此需转换源 FOUT，即 to_bit(FOUT)，将 FOUT 的类型转换为 bit 以与 z3 相匹配。

因此，当定义了元件（**component**）的端口类型之后，若其外部相连信号与端口类型不一致而使用类型转换函数时：在输入部分转换输入信号以与输入端口相匹配；在输出部分，转换输出端口以与输出信号相匹配。

此外，再说明两点：

① 转换函数 to_stdulogic(z2)将 z2 的类型转换为 std_ulogic，而 I1 的类型是 std_logic 类型，但 I1=>to_stdulogic(z2)是被允许的，而 std_logic 是 std_ulogic 派生出来的分辨子类型，因此可以得出如下结论：定义在一种类型之上的分辨子类型的对象与此基类型的对象是赋值匹配的。

② 上述端口映射也可写成：

```
port map(z1,to_stdulogic(z2),to_bit(FOUT)=>z3);
```

即输入部分可直接位置关联；而对输出部分，在这种情况下只能命名关联。由此可见，命名关联与位置关联使用范围不同，命名关联的使用范围要大于位置关联：

(a) 元件例示中前面使用了位置关联，后面仍可用命名关联；

(b) 相反，若前面使用了命名关联，后面就不能再用位置关联，只能继续使用命名关联；

(c) 元件例示中使用命名关联无次序要求，而使用位置关联则严格要求次序。当然，使用位置关联形式简单一些。

VHDL 源描述及其测试台

```
library IEEE;
use IEEE.std_logic_1164.all;
-------------------------------------------------------------
entity B_AND2 is
-------------------------------------------------------------
    port(
        I0  : in std_logic;
        I1  : in std_logic;
        FOUT: out std_logic
        );
```

```
end B_AND2;

architecture FUNC of B_AND2 is
begin
        FOUT <= I0 and I1 after 5 ns;
end FUNC;

-- 在元件例示语句中尝试使用转换函数
library IEEE;
use IEEE.std_logic_1164.all;

entity wand is
end wand;

architecture bench of wand is
  component B_AND2
    port(    I0      : in std_logic;
             I1      : in std_logic;
             FOUT    : out std_logic);
  end component;
  for U0: B_AND2 use entity work.B_AND2(FUNC);
  signal z1       : std_logic;
  signal z2,z3: bit;
begin
  U0: B_AND2
    port map(I0=>z1,
             I1=>to_stdulogic(z2),
             to_bitl(FOUT) => z3);
  driver:process
  begin
  ...
  end process;
end bench;
```

第 61 例　基于同一基类型的两分辨类型的赋值相容问题

王作建

1. 电路系统工作原理

本例的 4 个描述功能非常简单，仅仅是把输入信号赋给输出信号。

2. VHDL 语言描述方法及语法分析

类型 std_logic 是 IEEE 定义的由 std_ulogic 派生出来的分辨子类型，而类型 logic 则是用户定义的基于 std_ulogic 的派生类型（详细定义见描述部分），因此，std_logic 和 logic 是基于同一基类型 std_ulogic 的两个不同分辨子类型。

代码段 1 和代码段 2 说明类型为 std_loigc 和 logic 的单信号之间可以相互赋值；代码段 3 和代码段 4 说明类型为 std_loigc 和 logic 的向量类信号如何相互赋值。

因此，基于同一基类型的两个不同分辨子类型，其各自的信号相互赋值时有如下要求：

① 支持单信号间的相互赋值；

② 不支持向量类型信号之间的相互赋值。

如：

```
signal    s1    : std_logic;
signal    s     : std_logic_vector(3 downto 0);
signal    t1    : logic;
signal    t     : logic_vector(3 downto 0);
```

则：

```
s1   <=   t1        允许
t1   <=   s1        允许
s    <=   t         不允许
t    <=   s         不允许
```

为解决基于同一基类型的两分辨子类型的向量信号间的相互赋值，可以通过一条进程语句中的循环语句实现。

```
Process(t)
begin
  for I in 3 downto 0  loop
```

```
      s[I] <= t[I];
   end loop;
end process;
```

结论：VHDL 作为一种强类型语言，要求相同类型的对象间相互赋值，对上述两个分辨类型，模拟器支持单信号间的相互赋值，而不允许聚集类型信号相互直接赋值。对于聚集类型信号相互赋值只能通过循环以单信号赋值形式间接完成。

(1) 自定义类型 logic 说明

```
library IEEE;
use IEEE.std_logic_1164.all;

package logic_pack is
  function resolve(s : std_ulogic_vector) return std_ulogic;
  subtype logic is resolve std_ulogic;
  type logic_vector is array (natural range < >) of logic;
  type std_logic_array is array (natural range < >, natural range < >)
                                of std_logic;
end logic_pack;

package body logic_pack is
    type stdlogic_table is array(std_ulogic, std_ulogic) of std_ulogic;
    ---------------------------------------------------------------------
    -- 分辨函数
    ---------------------------------------------------------------------
    constant resolve_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','0','0','0','0','0','X'), -- | 0 |
            ('U','X','0','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')  -- | - |
        );

    function resolve ( s : std_ulogic vector ) return std_ulogic is
```

```
        variable result : std_ulogic := 'Z' ;  -- weakest state default
    begin
        if    (s'LENGTH = 1) then    return s(s'LOW);
        else
            for i in s'RANGE loop
                result := resolve_table(result, s(i));
            end loop;
        end if;
        return result;
    end resolve;
end logic_pack;
```

(2) VHDL 源描述

```
--***********************************************************************
-- 下面的前两例说明类型 std_logic 的对象和类型 logic 的对象可以相互赋值
-- 但 std_logic_vector 的对象和 logic_vector 的对象不能相互直接赋值，而
-- 必须通过单信号的循环赋值予以实现，如例 3、例 4 所示
--***********************************************************************

------------------------------------------------------------------
--(1) 下例说明类型 std_logic 的对象可以赋给类型 logic 的对象
------------------------------------------------------------------
library ieee;
use IEEE.std_logic_1164.all;
library dsp_lib;
use dsp_lib.logic_pack.all;

entity ttl is
  port(i : std_logic;
       o  : out logic);
end ttl;

architecture ss of ttl is
begin
   o <= i;
end ss;
----- ------------------------------------------------------------
--(2) 下例说明类型 logic 的对象可以赋给类型 std_logic 的对象
------------------------------------------------------------------
library ieee;
use IEEE.std_logic_1164.all;
```

```
library dsp_lib;
use dsp_lib.logic_pack.all;

entity tt2 is
  port(i : logic;
       o  : out std_logic);
end tt2;

architecture ss of tt2 is
begin
   o <= i;
end ss;
------------------------------------------------------------------------------
--(3) 下例说明如何将类型 std_logic_vector 的对象赋给类型 logic_vector 的对象
------------------------------------------------------------------------------
library ieee;
use IEEE.std_logic_1164.all;
library dsp_lib;
use dsp_lib.logic_pack.all;

entity tt3 is
  port(il   : std_logic_vector(3 downto 0);
       o   : out logic_vector(3 downto 0));
end tt3;

architecture ss of tt3 is
begin
  process(il)
   begin
     for I in 3 downto 0 loop
     o(I) <= il(I);
    end loop;
  end process;
end ss;
------------------------------------------------------------------------------
--(4) 下例说明如何将类型 logic_vector 的对象赋给类型 std_logic_vector 的对象
------------------------------------------------------------------------------
library ieee;
use IEEE.std_logic_1164.all;
library dsp_lib;
use dsp_lib.logic_pack.all;
```

```
entity tt4 is
  port(i1 : logic_vector(3 downto 0);
        o  : out std_logic_vector(3 downto 0));
end tt4;

architecture ss of tt4 is
begin
 process(i1)
   begin
     for I in 3 downto 0 loop
      o(I) <= i1(I);
     end loop;
  end process;
end ss;
```

（源描述文件名：61_assign.vhd）

第62例　最大公约数的计算

刁岚松

本例是计算两个整数的最大公约数的描述。

1. 算法设计的基本思想

设a,b,c为3个整数，若a-b=c，则a与b的最大公约数等于b与c的最大公约数。依据这个原理，写出VHDL描述如下：

```
if (x /= 0) and (y /= 0) then
    while (y /= 0) loop
        while (x >= y) loop
            wait until clk = '1';
            exit RESET_LOOP when reset = '1';
            x := x - y;
        end loop;
        h := x;
        x := y;
        y := h;
    end loop;
end if;
rdy <= true;
oup <= x;
```

如果x和y都不为零（只对非负整数求解），那么当y没有变为零时，循环执行两重**LOOP**语句，这几条语句的功能是用两个数中的较大的数减去较小的数，然后把较小的数和所求的差做为两个新的有效数。

电路有7个如下的端口：

```
port(reset : in bit;    -- 整个电路重启信号，出现上升沿时，整个电路运算终止，为下次运
    算做准备
     clk   : in bit;          --时钟脉冲信号
     rst   : in boolean;      --重新设置xin,yin信号
     xin   : in nat8;         --8位输入信号
     yin   : in nat8;         --8位输入信号
     rdy   : out boolean;     --输出准备好信号
     oup   : out nat8);       --8位输出端
```

每当 rst 被置为真，根据输入值 xin 和 yin 运算得出结果，将结果值赋给 oup，同时 rdy 被赋为真。

2. 电路的 VHDL 语言描述方法及语法分析

部分描述如下所示：

```
RESET_LOOP : loop
    wait until clk = ' 1' ;
    exit RESET_LOOP when reset = ' 1' ;     --如果 reset 为 '1'，则跳出循环
    while (rst = true) loop
        x := xin;
        y := yin;
        wait until clk = ' 1' ;
        exit RESET_LOOP when reset = ' 1' ;  --如果 reset 为'1'，则跳出循环
    end loop;

    ...

  end loop RESET_LOOP;
```

语法分析：

loop 语句：**loop** 语句包含要重复执行的一组顺序语句，执行零次或多次。一般格式如下：

```
[loop 标号：][重复模式]loop
        顺序语句；
  end loop[loop 标号]；
```

重复模式有两种：**while** 和 **for**，类似于其他程序设计语言中的 **while** 和 **for** 循环。若无重复模式，则为无限循环。

exit 语句：该语句用在循环语句内部。它有条件或无条件地终止当前循环迭代并终止该循环。若 loop 标号缺省，则 **exit** 语句作用于当前最内层循环，否则转到指定的循环中。若有 **when** 子句出现在 **exit** 语句中，但条件为 false，则循环正常继续。**exit** 语句的一般格式如下：

```
exit [loop 标号][when 条件];
```

3. 模拟测试向量的选择及模拟结果分析

测试台的部分描述如下：

```
rst    <= Inactive; --Inactive 为 False
xin    <= 0;
yin    <= 0;
reset <= '1' ;              --终止上次运算，为下次运算做准备
 wait until clk = '1' ;     --等待时钟上升沿

reset <= '0' ;              --为重启电路做准备
rst    <= Active;           --接收输入
wait until clk = '1' ;      --当 clk 变为高电平时，激活进程并计算结果

xin    <= 67;               --下一次计算开始
yin    <= 3;
    wait until clk = '1' ;
    rst    <= Inactive;
    while (rdy = true) loop
    --rdy 上升沿时，oup 端口值有效
       wait until clk = '1' ;
    end loop;
```

模拟结果：

xin=0,　yin=0 时，　结果为 0
xin=67,　yin=3 时，　结果为 1
xin=21,　yin=14 时，结果为 7
xin=36,　yin=8 时，　结果为 4

（源描述文件名：62_gcd.vhd
62_gcd_stim.vhd）

第 63 例　最大公约数七段显示器编码

吴清平

1. 电路系统工作原理

本例实现最大公约数七段显示器的编码。

图 63.1 所示为求两个整数 xin 和 yin 的最大公约数，并带有七段译码器的显示编码示意图。

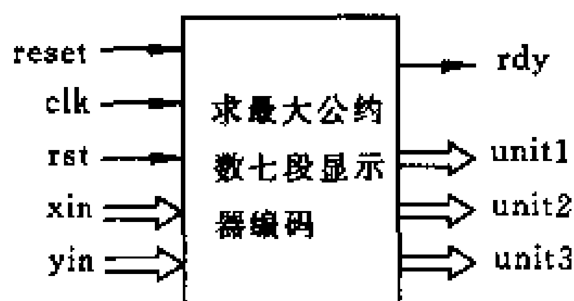


图 63.1　示意图

2. VHDL 语言描述方法及语法分析

(1) 包分析

在包 types 中定义两个子类型 nat8 和 a_dec，它们均是类型 integer 的子类型，nat8 的取值范围是 0~255，而 a_dec 的取值范围是 9~0。定义的语句如下：

```
subtype  nat8  is integer range 255 downto 0;
subtype  a_dec is integer range 9 downto 0;
```

(2) 实体 gcd_disp 分析

实体 gcd_disp 有 9 个端口：

reset:　输入类型，复位位。

clk:　输入类型，时钟信号。

rst:　输入类型，启动元件进行计算。

xin:　输入类型，需要求最大公约数的两个整数之一，数据类型为在 types 包中定义的 nat8。

yin:　输入类型，需要求最大公约数的两个整数之二，数据类型同 xin。

rdy:　输出类型，数据类型为 bit。用于指示输出数据是否准备好，高电平有效。

outp:　输出类型，输出最大公约数。数据类型同 xin。

unit1:　输出类型，数据类型为二进制 7 位位串，表示最大公约数个位数的七段显示器的编码，每一位对应于七段译码器的一段显示码管。

unit2:　输出类型，数据类型为二进制 7 位位串，表示最大公约 数十位数的七段显示器的编码，每一位对应于七段译码器的一段显示码管。

unit3:　输出类型，数据类型为二进制 7 位位串，表示最大公约数百位数的七段显示器的编码，每一位对应于七段译码器的一段显示码管。

(3) 结构体 algorithm 分析

此结构体为实体 gcd_disp 的实现。结构体中包含一个进程，由此进程实现求两个整数的最大公约数，并求出此最大公约数的七段译码器的编码。由于最大公约数的数据类型定义为 nat8，其取值范围 0~255，所以此数如以十进制显示，将有 3 位：百位，十位和个位。因此，该七段译码器的编码也有 3 个。其中求最大公约数的算法如下：

输入：　a, b

输出：　a, b 的最大公约数

步骤：　① 当 b = 0 时，转⑤

② 当 a > b 时，转③

③ a = a - b，转②

④ 交换 a, b 的值，转①

⑤ a 的值即为原两数的最大公约数，输出 a。算法结束。

此算法的一个依据是：若 a 和 b 的最大公约数为 c，设 a > b，则 b 和 d 的最大公约数也是 c，其中 d = a - b。

在求得最大公约数以后，也即 x 中此时存放的是最大公约数，将 x 的值赋给输出端 outp，然后开始计算七段显示器编码。其方法为：先将计数器 midx 置 0，然后将 x 循环减去 100，直到 x < 100 为止，每次减去 100，将计数器 midx 加 1，到循环完时，midx 中的值即为原 x 值的百位数，将 midx 的值存放在 x100 中；其后采用同样的方法将 x 循环减去 10，得到原 x 值的十位数和个位数，分别存放在变量 x10 和 x1 中。然后分别对 x100, x10, x1 求其七段显示器的编码。

七段显示器由 7 根显示码管组成，对每一根码管，用一位二进制表示，当该位为 0 时，表示此码管发光，如为 1，表示此码管不发光。对 7 根码管编号，如图 63.2 所示，然后由一个 7 位二进制数表示一个七段显示器的编码。

采用 **case** 语句求编码的程序描述如下：

```
case x1 is
```

```
        when 0 => unitl <- "1000000";
        when 1 => unitl <= "1111001";
        when 2 => unitl <= "0100100";
        when 3 => unitl <= "0110000";
        when 4 => unitl <= "0011001";
        when 5 => unitl <= "0010010";
        when 6 => unitl <= "0000010";
        when 7 => unitl <= "1111000";
        when 8 => unitl <= "0000000";
        when 9 => unitl <= "0010000";
        when others => unitl <= "1111111";
end case;
```

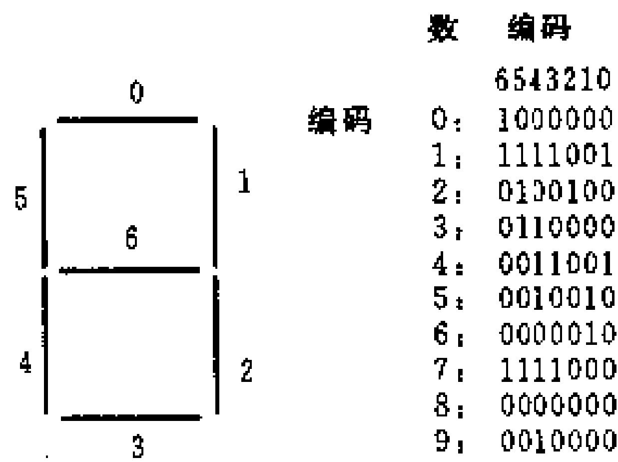


图 63.2 t段数码管显示器

(源描述文件名: 63_gcd_disp.vhd
63_vhdl.vhd
测试平台文件名: 63_stim.vhd)

第 64 例　交通灯控制器

吴清平

1. 电路系统工作原理

本例实现高速公路与乡间小路的交叉路口红绿灯的控制。实现如下要求：

① 只有在小路上发现汽车时，高速公路上的交通灯才可能变为红灯。

② 当有汽车在小路上时，小路的交通灯保持为绿灯，但不能超过给定的延迟时间（注：这段时间定义为 S 时间）。

③ 公路灯转为绿灯后，即使小路上有汽车出现，而公路上并无汽车，也将在给定的时间内保持绿灯（注：这段时间定义为 L 时间）。

2. VHDL 语言描述方法及语法分析

(1) 源描述分析

此交通灯控制器用一个有限自动机来实现上述功能，它定义了 5 种状态，见下表 64.1。

表 64.1　交通灯控制器的 5 种状态

	公路灯	小路灯
状态 0	绿	红
状态 1	黄	红
状态 2	红	绿
状态 3	红	黄
状态 4	红	红

其中状态 4 为初始状态（实际上该状态为冗余状态），其状态转换图如图 64.1 所示：

其中：C 表示小路上有车；L 表示过了一段长的时间；S 表示已过了一段短的时间；操作符 * 表示逻辑与的关系；操作符 + 表示逻辑或的关系。

描述中定义变量 current_state 表示当前状态，其类型为 BIT_VECTOR(2 **downto** 0)，5 种状态分别由它的 5 个值来代表。

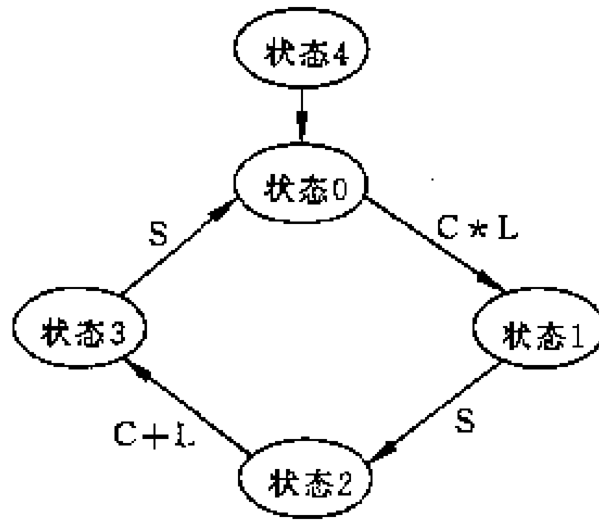


图 64.1　交通灯控制器的状态转换图

值	状态
"000"	0
"100"	1
"010"	2
"110"	3
"111"	4

在结构体中包含一个进程，此进程带一个显式敏感信号表，其敏感信号包括：TimeL、TimeS 和 Cars。所谓敏感信号表是指在关键字 **process** 之后跟随的信号列表，如以下语句：

```
process (cars, timeoutL, timeoutS)
```

带显式敏感信号表的进程，当敏感信号表中的任何一个信号上发生事件时，此进程将被激活。

进程说明部分定义的几个变量的意义如下：

newstate：　下一个状态值。

current_state：　当前状态值。

newHL：　高速公路灯的状态，三位位长的二进制位串，每一位分别表示绿、黄、红灯的亮、灭状态。如："100"表示路灯为红灯，"010"表示为黄灯，"001" 则表示为绿灯。

newFL：　乡间小路灯的状态，三位位长的二进制位串，每一位分别表示绿、黄、红灯的亮、灭状态。

NewST：　用于启动外部计时器的输出位。

在进程中使用一条 **case** 语句完成状态的转换关系。

```
case current_state  is
    when "000" => newHL := "100";  newFL := "001";
```

```
            if (Cars = '1') and (TimeoutL = '1') then
               newstate := "100"; newST := '1';
               newHL := "010"; newFL := "001";
            else
               newstate := "000"; newST := '0';
            end if;
     when "100" => newHL := "010"; newFL := "001";
             if (TimeoutS = '1') then
                newstate := "010"; newST := '1';
                newHL := "001"; newFL := "100";
             else
                newstate := "100"; newST := '0';
             end if;
     when "010" => newHL := "001"; newFL := "100";
             if (Cars = '0') or (TimeoutL = '1') then
                newstate :="110"; newST := '1';
                newHL := "001"; newFL := "010";
             else
                newstate := "010"; newST := '0';
             end if;
     when "110" => newHL := "001"; newFL := "010";
             if (TimeoutS = '1') then
                newstate := "000"; newST := '1';
                newHL := "100";  newFL := "001";
             else
                newstate := "110"; newST := '0';
             end if;
     when "111" => newstate := "000";
                newHL := "100";
                newFL := "001";
                newST := '0';
     when others =>
  end case;
```

在 **case** 语句的不同分支中又使用 **if** 语句完成状态的判断及转换。需要注意的是信号在赋值之后，其值并不立即有效（这是由于 delta 延迟的存在所造成的），只有变量的值在赋值之后会立即有效。因此在本例中用于表示状态的 current_state 被定义为变量。

3. 模拟结果分析

模拟结果部分波形如图 64.2 所示。

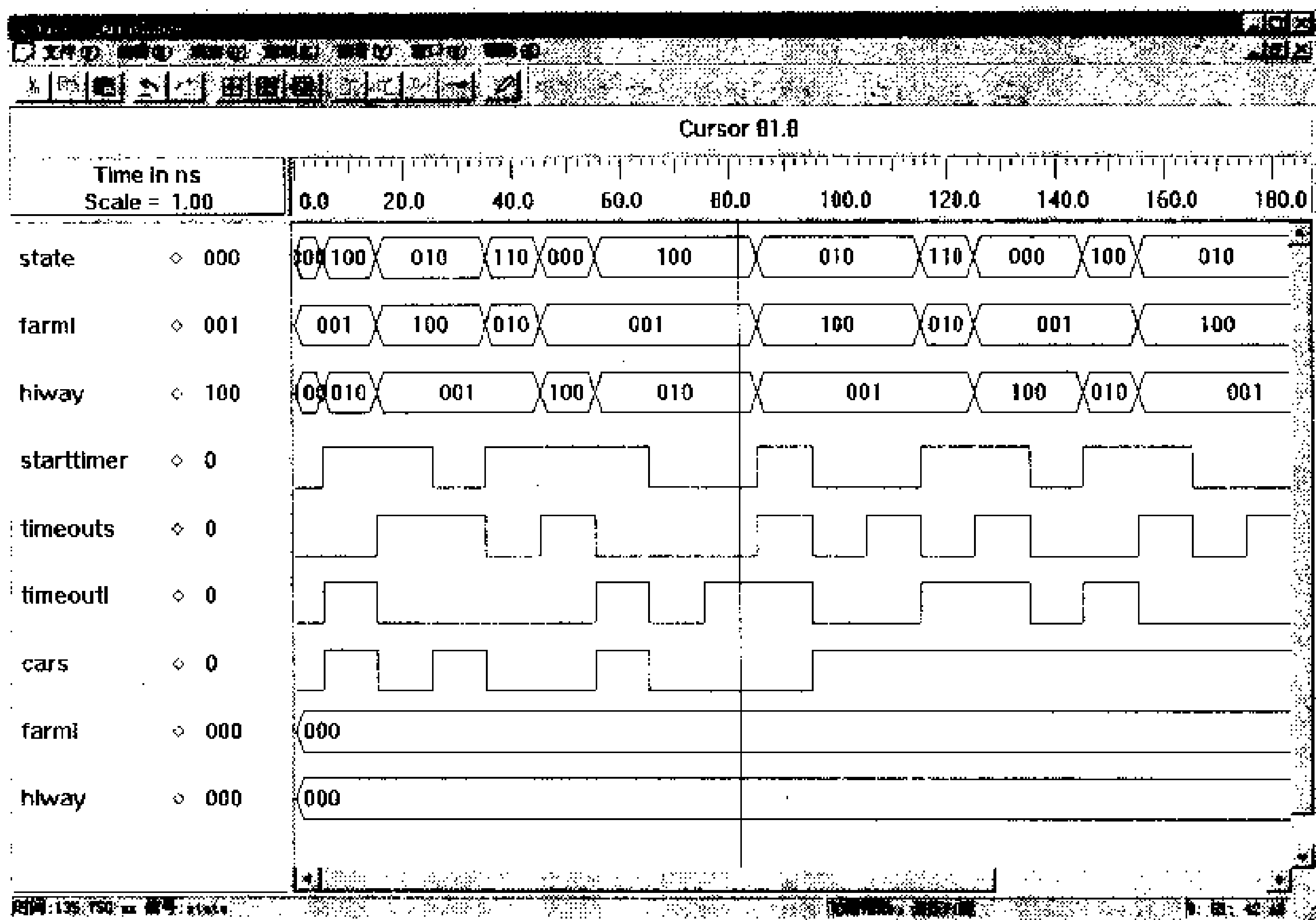


图 64.2 模拟波形

初始时，小路上为红灯，其值 farml 为 001，高速公路上为绿灯，其值 hiway 为 100。当 cars 值变为 1 时，即小路上来车时，并且 timeoutl 为 1（即高速公路上的绿灯已经维持一段较长的时间了），则高速公路上的灯开始转为黄灯，但此时小路上的灯仍为红灯，同时有限状态机的状态开始发生变换。经过一段时间后，高速公路转为红灯，同时小路转为绿灯，同时有限状态机状态变为 010。再经过一段较短的时间后，不管小路上是否有车，小路的灯开始转为黄灯，此时高速公路仍为红灯，其后，小路变为红灯，高速公路变为绿灯。周而复始，不断循环。

（源描述文件名：64_tlc.vhd
测试平台文件名：64_test_vectors.vhd）

第65例　空调系统有限状态自动机

刁岚松

1. 电路系统的工作原理

本例描述空调系统的有限状态自动机。两个输入端 temp_high 和 temp_low 分别与传感器相连，用于检测室内温度。如果室内温度正常，则 temp_high 和 temp_low 均为‘0’。如果室内温度过高，则 temp_high 为‘1’，temp_low 为‘0’。如果室内温度过低，则 temp_high 为‘0’，temp_low 为‘1’。根据 temp_high 和 temp_low 的值来判断当前的状态（太热 too_hot，太冷 too_cold 或适中 just_right），并决定 heat 和 cool 的输出值。

2. 电路的 VHDL 语言描述方法及语法分析

部分描述如下所示：

```
library ieee;
--使用 ieee 库中的标准包 Std_Logic_1164
use ieee.Std_Logic_1164.all;
--空调系统有限状态自动机的实体
entity air_conditioner is
    port(clk : in Std_ULogic;                --时钟输入信号
              temp_high :in Std_ULogic;      --过热传感器输入信号
              temp_low : in Std_ULogic;      --过冷传感器输入信号
              heat   : out Std_ULogic;       --致热输出信号
              cool : out Std_ULogic);        --致冷输出信号
end air_conditioner;

--空调系统有限状态自动机的结构体
architecture style_b of air_conditioner is
     type state_type is (just_right,too_cold,too_hot);
        attribute sequential_encoding : String;
        --定义 state_type 的属性 sequential_encoding
        attribute sequential_encoding of state_type :type is "00 01 10";
        signal stvar: state_type;
        attribute state_vector : String;
        --定义 stvar 的属性 state_vector
        attribute state_vector of style_b:architecture is "stvar";
begin
```

```
controller1 : process
  --clk 是该进程的敏感信号，当 clk 变为 '1' 时，激活进程
begin
  --等待 clk 变为 '1'
  wait  until clk=' 1' ;

  --根据 temp_low 和 temp_high 的值决定 stvar 的值
  --VHDL 语言的 if...then 语句与其他高级语言类似，但也有一些差别
  if (temp_low=' 1' ) then stvar<=too_cold;
  elsif (temp_high=' 1' ) then stvar<=too_hot;
  else stvar<=just_right;
  end if;

  --根据 stvar 的值决定 heat 和 cool 的值
  case stvar is
      when just_right =>  heat<=' 0' ;
                          cool<=' 0' ;
      when too_cold   =>  heat<=' 1' ;
                          cool<=' 0' ;
      when too_hot    =>  heat<=' 0' ;
                          cool<=' 1' ;
  end case;

end process controller1;
end style_b;
```

3. 模拟测试向量的选择及模拟结果分析

测试台的部分描述如下：

```
--temp_high，temp_low 被赋初始值
temp_high   <= ' 1' ;
temp_low    <= ' 0' ;
--第一个时钟上升沿，temp_high 变为 '0'，temp_low 变为 '1'
wait until clk = ' 1' ;
    temp_high   <= ' 0' ;
    temp_low    <= ' 1' ;
wait until clk = ' 1' ;
temp high   <= ' 0' ;
temp_low    <= ' 0' ;
wait until clk = ' 1' ;
temp_high   <= ' 0' ;
temp_low    <= ' 1' ;
```

```
wait until clk = '1';
temp_high  <= '0';
temp_low   <= '0';
wait until clk = '1';
temp_high  <= '0';
temp_low   <= '1';
wait until clk = '1';
temp_high  <= '0';
temp_low   <= '0';
wait until clk = '1';
temp_high  <= '0';
temp_low   <= '1';
wait until clk = '1';
temp_high  <= '1';
temp_low   <= '0';
wait until clk = '1';
temp_high  <= '0';
temp_low   <= '0';
wait until clk = '1';
temp_high  <= '0';
temp_low   <= '0';
wait until clk = '1';
temp_high  <= '0';
temp_low   <= '1';
wait until clk = '1';
temp_high  <= '0';
temp_low   <= '1';
wait until clk = '1';
    assert false report "End of Simulation" severity error;
```

输出波形如图 65.1 所示。在 150ns 处，temp_low 为'1'，temp_high 为'0'，所以 stvar 被赋值为 too_cold，由于 stvar 是信号，它的值并不能立即改变，执行下面的语句时，stvar 的值仍然是旧值(too_hot)。

```
case stvar is
    when just_right => heat<='0';
                       cool<='0';
    when too_cold   => heat<='1';
                       cool<='0';
    when too_hot    => heat<='0';
                       cool<='1';
end case;
```

所以，heat 被赋值为'0'，cool 被赋值为'1'。

在 250ns 处，temp_low 为'0'，temp_high 也为'0'，所以 stvar 被赋值为 just_right，它的值并不能立即改变，仍是旧值（too_cold），所以，heat 被赋值为'1'，cool 被赋值'0'。其他时刻与上面分析类似。

在本例中，输出端 heat 和 cool 的值是由电路判断当前温度后决定的，它有两个时钟脉冲的延迟，由波形图来看，电路似乎不能及时调节温度。在实际情况下，由于温度变化慢，时钟频率很高，几十纳秒的延迟完全可以忽略，所以电路总能及时调节温度。

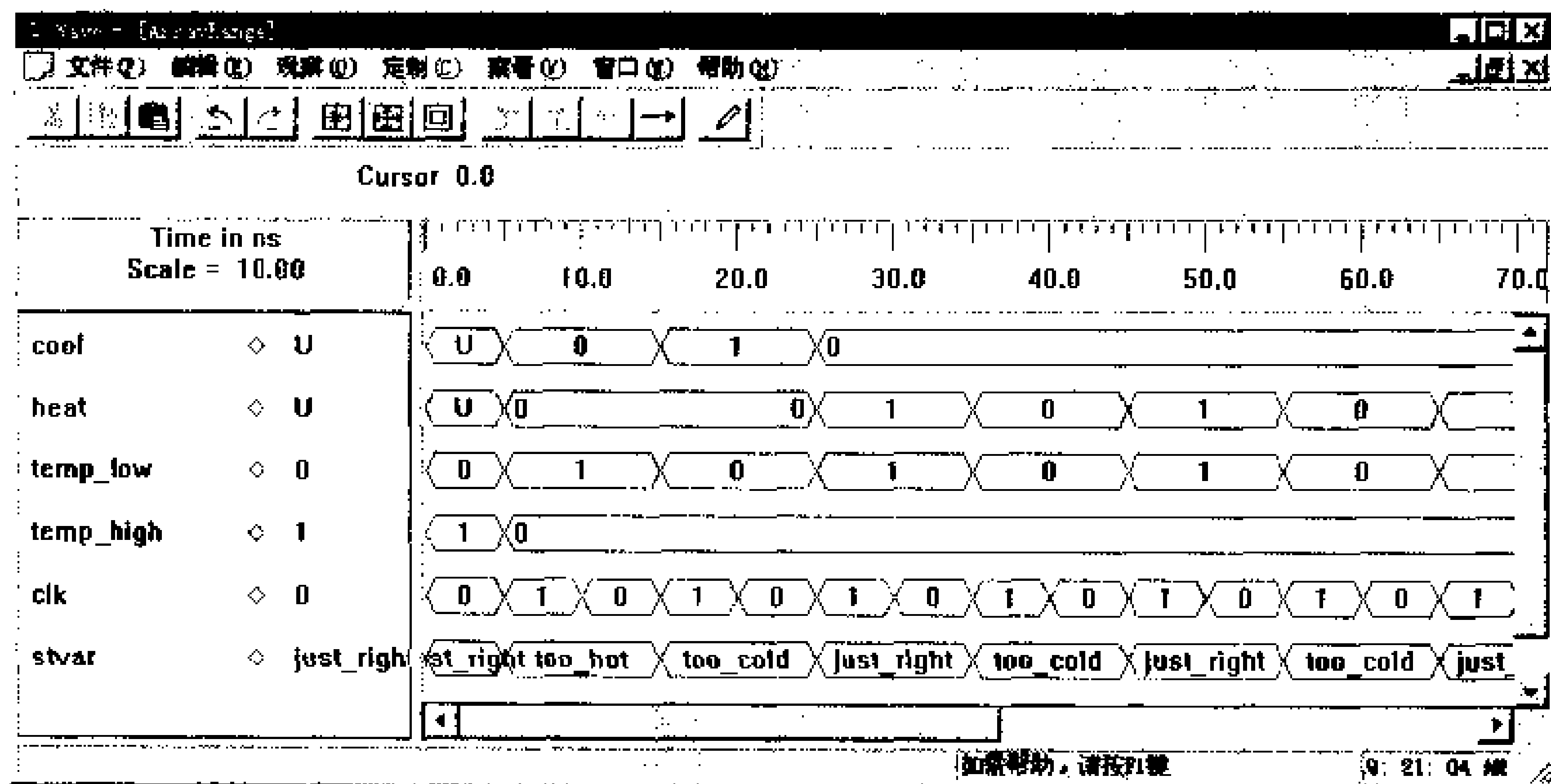


图 65.1　空调系统有限状态自动机输出波形图

（源描述文件名：65_conditioner.vhd
65_conditioner_stim.vhd）

第66例　FIR滤波器

谢　巍

1. 电路系统的工作原理

本例描述了一个 FIR(有限长脉冲响应)滤波器，与五阶椭圆滤波器不同的地方是，其单位脉冲响应包含有限个非零值，即持续时间不是无限长，但与五阶椭圆滤波器相同的是，两者都是低通滤波器。FIR 滤波器属于数字滤波器中的非递归类型，其瞬间的输出响应仅仅取决于当时及以前的激励，而与以前的输出无关。具体来讲，FIR 滤波器对输入采样 $U[k]$进行操作，k 是下标变量，用来表明在输入中特定采样的位置，其结果保存在 $Y[k]$中。系统函数表示为

$$Y[k]=\sum_{i=0}^{L}h[i]U[k-i]$$

对 FIR 滤波器进行分析可知：对于每次采样，都需要利用本次采样之前的 L 个采样值(本例采样值 L 为 16)，这样 $L+1$ 个采样值分别与常系数 $h[i]$相乘，$h[i]$是常数，由它决定滤波器的特征函数，然后累加形成本次采样的结果。

2. VHDL 语言描述方法及语法分析

本例由 3 个文件组成：66_pack.vhd，66_fir.vhd 和 66_testfir.vhd。滤波器中用到的常系数 $h[i]$可以用一个系数数组来表示，在 VHDL 语言中对于常数可以在包中将其定义为一个常数类型。本例在66_pack.vhd确定的包中定义了一个名为coef_arr的类型，该类型实际上是一个数组，其长度为 17。该数组用来保存长度为 9 的带符号数，带符号数的第 1 位为符号位，“1”表示为负数，“0”表示为正数。66_fir.vhd 文件是对 FIR 滤波器的行为描述。66_testfir.vhd 文件提供了一个测试平台，其中有 4 组测试向量，用于验证设计描述的正确性。

如果设计工具的缺省库不支持带符号数，可以将下面有关带符号数的描述作为预定义包，并在 66_pack.vhd，66_fir.vhd，66_testfir.vhd 中各增加一条 **use** 子句：**use** work.SIGNED_ARITH.**all**。

下面是本例中用到的一些有关带符号数操作函数的 VHDL 描述，读者可以将其作为预定义包文件 66_signed.vhd 来保存。应当注意的是，在对本例进行模拟验证时，如果缺省库不支持带符号数，那么应该先对带符号数包 66_signed.vhd 文件进行编译，然后对 coeffs 包 66_pack.vhd 文件进行编译，对各包文件编译后，再对 66_fir.vhd 文件进行

编译，最后对测试台 66_testfir.vhd 进行编译。必须注意编译的顺序。否则编译系统将报错。

(1) 包

① 预定义包 SIGNED_ARITH

由于有关带符号数操作很多，在这里不能一一列举，所以只节选了文件 66_signed.vhd 中本例涉及到的有关操作。

```
--包 SIGNED_ARITH 的包说明部分
library IEEE;
use IEEE.std_logic_1164.all;

package SIGNED_ARITH is
    type SIGNED is array (NATURAL range < >) of STD_LOGIC;
    function "+"(L: SIGNED; R: SIGNED) return SIGNED;
    function "-"(L: SIGNED; R: SIGNED) return SIGNED;
    function "*"(L: SIGNED; R: SIGNED) return SIGNED;

end SIGNED_ARITH;

--包 SIGNED_ARITH 的包体部分
library IEEE;
use IEEE.std_logic_1164.all;

package body SIGNED_ARITH is
--*********************************************************
function mult(A,B: SIGNED) return SIGNED is
--*********************************************************
  --实现两个带符号数的乘法
  --变量说明
      variable BA: SIGNED((A'LENGTH + B'LENGTH -1) downto 0);
      variable PA: SIGNED((A'LENGTH + B'LENGTH -1) downto 0);
      variable AA: SIGNED(A'LENGTH downto 0);
      variable neg: STD_ULOGIC;
      constant one : UNSIGNED(1 downto 0) := "01";
    begin
        if (A(A'LEFT) = 'X' or B(B'LEFT) = 'X' ) then
              PA := (others => 'X' );
              return(PA);
```

```
end if;
    PA := (others => '0');
    neg := B(B'LEFT) xor A(A'LEFT);
    BA := CONV_SIGNED(('0' & SIGNED'(ABS(B))), (A'LENGTH+B'LENGTH));
    AA := '0' & ABS(A);
    --采用左移相加的方法实现乘法
    for i in 0 to A'LENGTH-1 loop
      if AA(i) = '1' then
        PA :- PA+BA;
      end if;
      BA := SHL(BA, one);
    end loop;
    if (neg= '1') then
      return(-PA);
    else
      return(PA);
    end if;
  end;

--*****************************************************
function minus(A, B: SIGNED) return SIGNED is
--*****************************************************
  --实现两个带符号数的减法
    variable carry: STD_ULOGIC;
    variable BV: STD_ULOGIC_VECTOR (A'LEFT downto 0);
    variable sum: SIGNED (A'LEFT downto 0);

    begin
    if (A(A'LEFT) = 'X' or B(B'LEFT) = 'X') then
          sum := (others => 'X');
          return(sum);
    end if;
    carry := '1';
    BV := not STD_ULOGIC_VECTOR(B);
      --采用带进位按位异或的方法实现减法
    for i in 0 to A'LEFT loop
        sum(i) := A(i) xor BV(i) xor carry;
        carry := (A(i) and BV(i)) or
             (A(i) and carry) or
```

```
            (carry and BV(i));
    end loop;
    return sum;
    end;

--*******************************************************
function plus(A, B: SIGNED) return SIGNED is
--*******************************************************
  --实现两个带符号数的加法
    variable carry: STD_ULOGIC;
    variable BV, sum: SIGNED (A' LEFT downto 0);
    begin
    if (A(A' LEFT) = ' X'  or B(B' LEFT) = ' X' ) then
           sum := (others => ' X' );
           return(sum);
    end if;
    carry := ' 0' ;
    BV := B;
      --采用带进位按位异或的方法实现减法
    for i in 0 to A' LEFT loop
        sum(i) := A(i) xor BV(i) xor carry;
        carry := (A(i) and BV(i)) or
             (A(i) and carry) or
             (carry and BV(i));
    end loop;
    return sum;
    end;

--*******************************************************
function "*"(L: SIGNED; R: SIGNED) return SIGNED is
--*******************************************************
  --调用 mult 实现两个带符号数的乘法
    begin
          return   mult(CONV_SIGNED(L, L' LENGTH),
                   CONV_SIGNED(R, R' LENGTH));
    end;

--*******************************************************
function "+"(L: SIGNED; R: SIGNED) return SIGNED is
```

```
--*******************************************************
  --调用 plus 实现两个带符号数的加法
    constant LENGTH: INTEGER := max(L' LENGTH, R' LENGTH);
    begin
    return plus(CONV_SIGNED(L, LENGTH),
             CONV_SIGNED(R, LENGTH));
    end;
--*******************************************************
function "-"(L: SIGNED; R: SIGNED) return SIGNED is
--*******************************************************
  --调用 minus 实现两个带符号数的减法
    constant LENGTH: INTEGER := max(L' LENGTH, R' LENGTH);
    begin
    return minus(CONV_SIGNED(L, LENGTH),
              CONV_SIGNED(R, LENGTH));
    end;
end SIGNED_ARITH;
```

② coeffs 包

```
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.SIGNED_ARITH.all;
package coeffs is
  type coef_arr is array (0 to 16) of signed (8 downto 0);
  constant coefs: coef_arr:=(
   "111111001", "111111011", "000001101", "000010000",
   "111101101", "111010110", "000010111", "010011010",
   "011011110", "010011010", "000010111", "111010110",
   "111101101", "000010000", "000001101", "111111011",
   "111111001");
end coeffs;
```

文件 66_signed.vhd 和 66_pack.vhd 各包含了一个包，其中 SIGNED_ARITH 包在包说明中定义了一个带符号数类型和 3 个有关带符号数的函数，但由于包说明只是为包定义相应接口，所以各函数的具体功能必须在包体中给出。但包体并不是定义包所必需的，在 coeffs 包中，由于没有子程序说明，所以可以不带包体。由于在 coeffs 包中使用了带符号数，所以要使用 **use** 子句使预定义包 SIGNED_ARITH 中的有关说明在 coeffs 包中可用，同样，为了使这两个包的说明在 66_fir.vhd 和 66_testfir.vhd 文件中可用，也

都要在这两个文件中使用 **use** 子句。

(2) 信号赋值语句和变量赋值语句

在文件 66_fir.vhd 中利用移位寄存器 shift 来保存最近的 L 个采样值 $U[k-i]$，在 VHDL 语言中，利用长为 L 的数组来表示，但每次采样后，都要进行移位以确保在该数组中只保存最近的 L 个采样值。

在 reset_loop 循环中，由于 reset 信号为'1'，所以对结果 result 置零，并对移位寄存器 shift 置零，shift(i):=(**others**=>'0')，这样最近 L 个采样值 $U[k-i]$ 都为零。

在 main 循环中，首先计算当前采样值 $U[k]$ 与系数 $h[0]$ 相乘产生的结果，然后向前寻找 L 个原始采样值，并移位。如从 shift 数组中找到第 L 次采样后，将 shift[L]中的采样值用 shift[L-1]取代，依次进行下去，最后 shift[0]就可以用来保存本次采样值，这样 shift 中就始终只保存最近的 L 个采样值。

66_fir.vhd 的源描述如下：

```
--使用 use 子句使 IEEE 库中的 std_logic_1164 和 std_logic_arith 包可见
--使用 use 子句使当前工作库中的 SIGNED ARITH 和 coeffs 包可见
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.SIGNED_ARITH.all;
use work.coeffs.all;

--实体说明
entity fir is
 port(clk,reset: in std logic;
      sample: in signed ( 7 downto 0);
      result: out signed ( 16 downto 0));
end fir;

--结构体说明
architecture beh of fir is
begin
     fir_main :process
     type shift_arr is array (16 downto 0) of signed (7 downto 0);
     variable tmp,old:signed( 7 downto 0);--tmp 用于保存本次采样值,
                                          --old 用于保存在 shift 中的第 i 个采样;
     variable pro:signed (16 downto 0);--pro 用于记录 U[k-i]与 h[i]的乘积
     variable acc:signed (16 downto 0);--acc 用于记录累加值
```

```
    variable shift:shift_arr;
begin
    reset_loop:loop
    for i in 0 to 15 loop --将寄存器 shift 清零
     shift(i):=(others=>'0');--由于 shift(16)只用于存放需丢弃的采样值，所以无需清零。
    end loop;
    result<=(others=>'0'); --将结果清零
    wait until clk'EVENT and clk='1';
    if reset='1' then exit reset_loop;
    end if;
    main:loop                     --计算当前采样的响应
    tmp:=sample;                  --取当前采样值
    pro:=tmp*coefs(0);
    acc:=pro;
    for i in 15 downto 0 loop
    old:=shift(i);
    pro:=old*coefs(i+1);
    acc:=acc+pro;                 --累计最近 L 次采样值
    shift(i+1):=shift(i);         --shift 寄存器移位
    end loop;
    shift(0):=tmp;                --将当前采样值保存在 shift[0]中，
                                  --以便下次采样时，shift 中保存的是最近 L 次采样值
    result<=acc;                  --将结果送给 result
    wait until clk'EVENT and clk='1';
    if reset='1' then exit reset_loop;
    end if;
    end loop main;
  end loop reset_loop;
 end process;
end beh;
```

在 VHDL 语言中，最简单的信号赋值语句，其形式如本例中 result<=acc 是不带 **after** 子句的，这种信号赋值语句指明右侧的波形将在一个 delta 延迟后赋给左侧信号。虽然物理上 delta 延迟是 0ns，但在排序上是有意义的。例如，一个在两个 delta 延迟后进行的信号赋值语句将排在只有一个 delta 延迟的信号赋值语句之后，其结果对于只有一个 delta 延迟的信号赋值语句来讲是不可用的。

在 VHDL 语言中，信号赋值语句的延迟有惯性延迟和传输延迟之分，在缺省情况下为惯性延迟，传输延迟必须是显式的。惯性延迟如果使用 **after** 子句指明具体延迟时间，

如 target1<=waveform **after** 5 ns，那么，在 waveform 中只有持续时间大于等于 5ns 的脉冲，将在 5ns 后出现在 target1 上。

在 VHDL'93 标准中，惯性延迟增加了关键字 **reject**，如 target2<=**reject** 3 ns **inertial** waveform **after** 5 ns，那么在 waveform 中只有持续时间大于 3ns 的脉冲经过 5ns 延迟后显示在 target2 上，持续时间小于等于 3ns 的脉冲都不会显示在 target2 上。

对于传输延迟，若 target3<=**transport** waveform **after** 5 ns，则无论 waveform 中的脉冲持续时间有多长，都将经过 5ns 延迟出现在 target3 中。

变量赋值语句如 tmp:=sample 与信号赋值语句的不同点在于，它无需经过 delta 延迟后才可用，而是赋值后立即可用。

3. 模拟结果分析

测试平台确立了一个组件 filter，并有 3 个进程来驱动滤波器，第 1 个进程是时钟产生器，初始时钟信号为'1'，然后每隔 50ns 翻转极性，在整个期间都运行。第 2 个进程产生两个循环的重置脉冲。第 3 个进程产生激励，首先将输入置为 0，然后等待 6 个时钟周期，再将输入分别置为 2，8，-32。

```
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.SIGNED_ARITH.all

--定义测试台实体
entity tb_e is
end tb_e;

architecture tb_a of tb_e is
signal clock,reset:std_logic;
signal instream: signed ( 7 downto 0);
signal outstream: signed ( 16 downto 0);

component fir                  -- 对 FIR 滤波器进行声明
port (clk,reset: in std_logic;
      sample: in signed (7 downto 0);
      result: out signed (16 downto 0);
end component;

--组装
```

```
for filter:fir use entity work.fir(beh);

begin
   --对滤波器进行例示
   filter:fir
   port map(clk=>clock,
            reset=>reset,
            sample=>instream,
            result=>outstream);
   --产生时钟信号
   clockgen:process
   begin
        clock<= '1';
        loop
            wait for 50 ns;
            clock<=not clock;
        end loop;
  end process clockgen;
  --产生重置信号，以便将 shift 寄存器、结果清零，然后将重置信号置‘0’
  po_reset: process
  begin
       reset<='1';
       wait for 102 ns;
       reset<='0';
       wait;
  end process po_reset;
  --产生激励信号
  stimulus: process
  begin
   instream<="00000000";
   wait for 302 ns;
   instream<="00000010";
   wait for 302 ns;
   instream<="00001000";
   wait for 302 ns;
   instream<="10100000";
   wait for 302 ns;
   --结束模拟
   assert false report "--end of simulation--" severity error;
```

```
   end process stimulus;
end tb_a;
```

模拟输出的部分波形如图 66.1 所示。

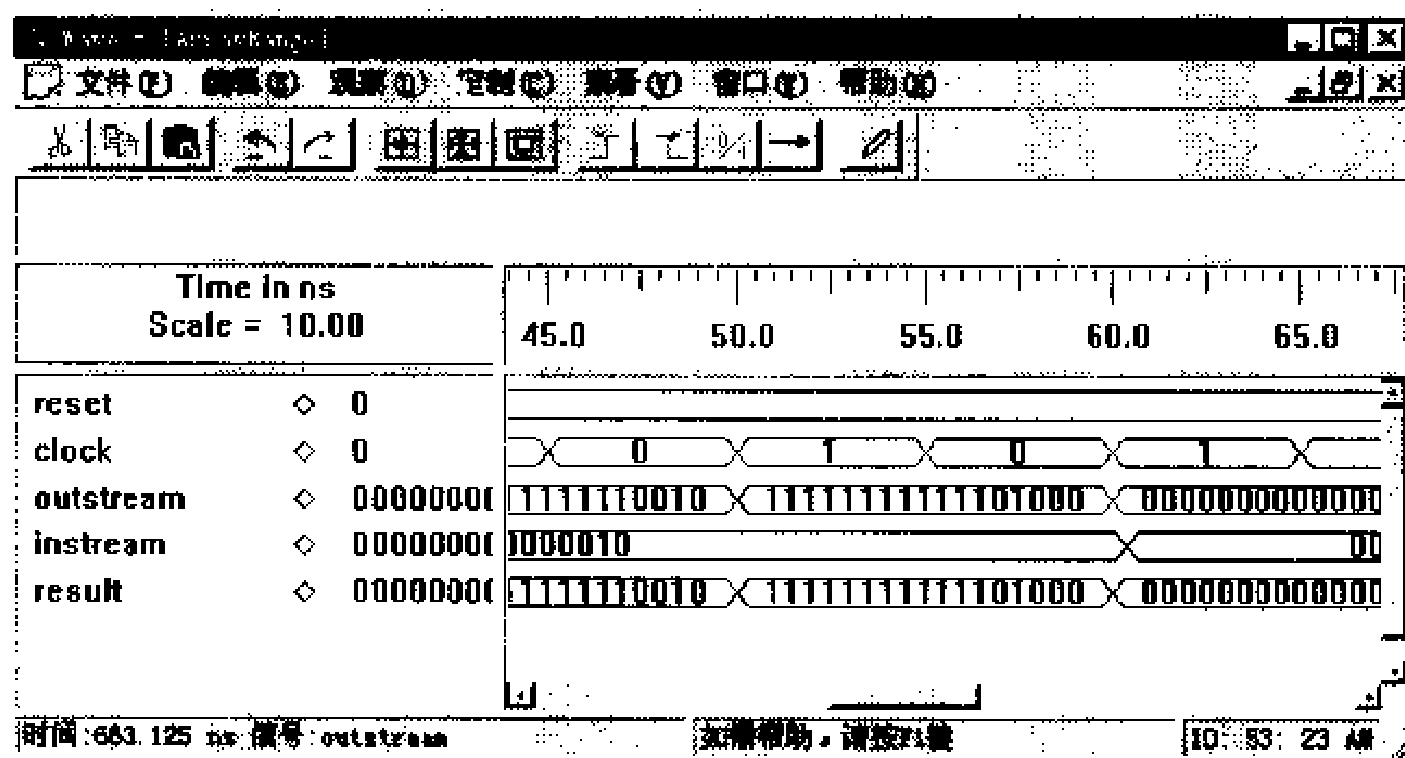


图 66.1　FIR 模拟结果波形图

（源描述文件名：66_signed.vhd
66_pack.vhd
66_fir.vhd
测试平台文件名：66_testfir.vhd）

第 67 例　五阶椭圆滤波器

刘沁楠

1. 电路系统原理

本例是一个标准的 Bench-Mark，用于实现五阶椭圆滤波器。滤波器用于把某个信号中的某些频率分量分离出来或者去掉。广义上讲，就是把输入序列通过一定的运算变换成输出序列的系统。系统函数表示为

$$H(Z)=\frac{\sum_{k=0}^{m} b_k Z^{-k}}{1-\sum_{k=1}^{n} a_k Z^{-k}}$$

椭圆滤波器属于无限长脉冲响应(IIR)数字滤波器，其单位脉冲响应包含无限个非值零，即持续时间无限长。本例处理的滤波器是低通滤波器，它可以滤除高于一定限度的频率，该频率称为截止(cut-off)频率。椭圆滤波器是 Chebysher 滤波器的改进版本，其响应函数采用与椭圆函数类似的形式，该滤波器在通带和阻带之间分布通带波，而保留 Chebysher 滤波器的过渡特性。尽管它比其他滤波器复杂得多，但上述特性使其在实际应用中极具吸引力。Jacobian 椭圆函数以 Jacobian 椭圆正弦和余弦函数的形式予以表达，它决定了椭圆滤波器的响应是一组非线性、二阶微分方程的解。获取滤波器的阶数及其他重要参数，需要经过相当复杂的运算过程，在此着重讨论 VHDL 语法，读者可参考 C.W.Parks 和 J.Burrus 编著的"Digital Filter Theory and Design"一书了解该过程的具体内容。书中还以通带波、阻带波和过渡带宽的形式给出滤波器的响应特征。

2. VHDL 描述及主要语法现象

本例由 3 个文件组成。文件 67_pack.vhd 提供一个程序包，该程序包定义了一系列对位向量进行操作的函数；文件 67_ellipf.vhd 则是对五阶椭圆滤波器的行为描述，通过一条进程语句描述输入与输出之间的转换关系。文件 67_test_vector.vhd 提供一个测试台("test_bench")，其中包含 6 组测试向量，利用它们可以验证设计描述的正确性。

下面结合源描述，介绍本例中出现的 VHDL 主要语法现象。

(1) 包

文件 67_pack.vhd 的描述节选如下：

```
--包 BIT_FUNCTIONS 的包说明部分
```

```
package BIT_FUNCTIONS is

  function SHL0 (v2: bit_vector; dist: integer) return bit_vector;
  function SHL1 (v2: bit_vector; dist: integer) return bit_vector;
  function SHL (v2: bit_vector; fill: bit) return bit_vector;
  function SHR0 (v2: bit_vector; dist: integer) return bit_vector;
  function SHR1 (v2: bit_vector; dist: integer) return bit_vector;
  function SHR (v2: bit_vector; fill: bit) return bit_vector;

  function INT_TO_BIN (number, length:integer) return bit_vector;
  function BIN_TO_INT (v2: bit_vector) return integer;

  function ONES_COMP (v2: bit_vector) return bit_vector;
  function TWOS_COMP (v2: bit_vector) return bit_vector;
  function "-"(x1, x2: bit_vector) return bit_vector;
  function DEC(v2: bit_vector) return bit_vector;

  function "+"     (x1, x2: bit_vector) return bit_vector;
  function INC (v2: bit_vector) return bit_vector;

  function ODD_PARITY ( v1 : bit_vector ) return bit;
  function EVEN_PARITY ( v1 : bit_vector ) return bit;

  function REVERSE (v2: bit_vector) return bit_vector;
  function SUM(v2 : bit_vector) return integer;

  function BIT_SLICE(v2 : bit_vector;high_val, low_val : integer)
             return bit_vector;
  function ASSIGN_TO_SLICE (v1: bit_vector;
                           high_val, low_val: integer;
                           x2: bit_vector          ) return bit_vector;

end BIT_FUNCTIONS;

    --包 BIT_FUNCTIONS 的包体部分
package body BIT_FUNCTIONS is

--*********************************************************************
function SHL0 (v2: bit_vector; dist: integer) return bit_vector is
--*********************************************************************
   variable v1 : bit_vector (v2'HIGH downto v2'LOW);
   variable shift_val: bit_vector (v1'RANGE);
```

```
   variable I: integer;

   begin
      v1 := v2;
      for I in v1' HIGH downto (v1' LOW + dist) loop
         shift_val(I) := v1(I - dist);
      end loop;

      for I in (v1' LOW + dist - 1) downto v1' LOW loop
         shift_val(I) := ' 0' ;
      end loop;

      return shift_val;
end SHL0;
...
--**********************************************************************
function "+"(x1, x2 : bit_vector) return bit_vector is
--**********************************************************************
   --变量说明
   variable v1 : bit_vector (x1' HIGH - x1' LOW downto 0);
   variable v2 : bit_vector (x2' HIGH - x2' LOW downto 0);
   variable CARRY: bit := ' 0' ;
   variable S: bit vector (1 to 3);
   variable NUM: integer range 0 to 3 := 0;
   variable SUM: bit_vector (v1' RANGE);
   variable I,K: integer;

   begin
      v1 := x1;
      v2 := x2;
      --判断 x1、x2 是否等长
      assert v1' LENGTH = v2' LENGTH
      report "BIT_VECTOR +: operands of unequal lengths"
      severity failure;

      --实现两个 bit_vector 类型变量的加法
      for I in v1' LOW to v1' HIGH loop
         S:= v1(I) & v2(I) & CARRY;
         NUM := 0;

 --将两个 bit_vector 类型变量的相应位做加法
         for K in 1 to 3 loop
```

```
            if S(K) = '1' then
               NUM := NUM + 1;
            end if;
         end loop;

      --求和并确定进位值
       case NUM is
            when 0 => SUM(I) := '0'; CARRY := '0';
            when 1 => SUM(I) := '1'; CARRY := '0';
            when 2 => SUM(I) := '0'; CARRY := '1';
            when 3 => SUM(I) := '1'; CARRY := '1';
         end case;
      end loop;

      return SUM;
end "+";

...
end  BIT_FUNCTIONS;
```

不难发现，该文件仅包含一个包，其中定义了 20 个函数。VHDL 语言在实体说明和结构体内部定义的数据类型、常量及子程序，对其他设计单元是不可见的。而包则是允许分享属于实体数据的一种机制，它包含有可用于其他设计单元的一系列说明。包由两部分组成：包说明和包体。包说明为包定义接口，包体则规定其实际功能，包体并非总是必需的，除非在包中包含有子程序说明，此时子程序体必须放在包体中。使用 **use** 子句可使包中的说明可见，如文件 67_ellipf.vhd 中的第 1 条语句 “**use** work.BIT_FUNCTIONS.**all**;”。该语句使得在文件 67_pack.vhd 中定义的程序包内的所有说明对设计实体可见。本例中，包说明部分主要由子程序说明构成，相应的子程序体在包体 BIT_FUNCTIONS 中定义。

(2) 子程序

子程序由过程和函数组成。本例中出现的子程序均为函数。对于函数而言，形式参数均为输入参数（**in** 模式），且对象类型为信号或常量，缺省为常量。函数的返回值为一个变元。程序通过函数和过程，可定义一些公共的操作。下面结合程序，重点讨论函数“+”的实现方法。该函数实现两个等长的位向量的加法运算。该函数定义于包体 BIT_FUNCTIONS 中。

在函数说明部分，说明一系列算法实现过程中可能用到的变量。其中 v_1，v_2 采用标准数据类型 bit_vector，这是一种非限制性的数组类型，数组中的每个元素均为 bit 类型，数组的元素数目可在 0 到整数最大值之间任意取值。本例中数组元素个数通过对输入

参数 x_1、x_2 的属性值计算得到，即 v_1 的位向量个数与 x_1 的位向量个数相同、v_2 的位向量个数与 x_2 的位向量个数相同。

变量 CARRY 为 bit 类型，代表运算过程中所产生的进位。变量 S 为 3 位位向量，它将变量 v_1 的第 n 位 $v_1(n)$ 与变量 v_2 的第 n 位 $v_2(n)$ 以及进位 CARRY 相连接，其中 S 的最高位为 $v_1(n)$，最低位为 CARRY；NUM 是一整型变量，取值范围为 0～3，初值为 0，它用于存放输入位向量第 n 位上的相加结果，即 NUM = $v_1(n)+v_2(n)$+CARRY，因而 NUM 的最大值为 3。

SUM 是与变量 v_1 等长的位向量，它代表最终的输出结果。

了解各个变量的含义之后，该函数的算法实现就显而易见了。此外，读者还应注意以下几点：

① 描述中出现的断言（**assert**）语句要求输入的两个位向量必须等长，否则将报错并停止模拟。

② 运算符&不同于高级语言中的“与”运算符。它是将相同类型的操作数相连接。本例中即是将 $v_1(I)$，$v_2(I)$，CARRY 依次连接。

③ VHDL 支持子程序的重载。它允许多个子程序使用相同的名称，通过参数匹配决定采用哪个子程序。逻辑、关系和算术运算符都是函数，它们也可重载。本例就是对加法运算重载的实例。

（3）变量赋值语句

文件 67_ellipf.vhd 仅包含一条进程语句。该进程由若干条变量赋值语句组成。源描述如下：

```
--use 子句使当前工作库中的包 BIT_FUNCTIONS 可见
use work.BIT_FUNCTIONS.all;

--实体说明
entity ellipf is
    port ( inp: in bit_vector(15 downto 0);--端口说明
           outp: out bit_vector(15 downto 0);
           sv2, sv13, sv18, sv26, sv33, sv38, sv39 :
                in bit_vector(15 downto 0);
           sv2_o, sv13_o, sv18_o, sv26_o, sv33_o, sv38_o, sv39_o :
                out bit_vector(15 downto 0));
end ellipf;

--结构体
architecture ellipf of ellipf is
```

```
begin
--进程语句
process (inp, sv2, sv13, sv18, sv26, sv33, sv38, sv39)
--变量说明
    variable n1, n2, n3, n4, n5, n6, n7 : bit_vector(15 downto 0);
    variable n8, n9, n10, n11, n12, n13 : bit_vector (15 downto 0);
    variable n14, n15, n16, n17, n18, n19 : bit_vector (15 downto 0);
    variable n20, n21, n22, n23, n24, n25 : bit_vector (15 downto 0);
    variable n26, n27, n28, n29 : bit_vector (15 downto 0);

 begin
    n1          := inp + sv2;
    n2          := sv33 + sv39;
    n3          := n1 + sv13;
    n4          := n3 + sv26;
    n5          := n4 + n2;
    n6          := n5 ;
    n7          := n5 ;
    n8          := n3 + n6;
    n9          := n7 + n2;
    n10         := n3 + n8;
    n11         := n8 + n5;
    n12         := n2 + n9;
    n13         := n10 ;
    n14         := n12 ;
    n15         := n1 + n13;
    n16         := n14 + sv39;
    n17         := n1 + n15;
    n18         := n15 + n8;
    n19         := n9 + n16;
    n20         := n16 + sv39;
    n21         := n17 ;
    n22         := n18 + sv18;
    n23         := sv38 + n19;
    n24         := n20 ;
    n25         := inp + n21;
    n26         := n22 ;
    n27         := n23 ;
    n28         := n26 + sv18;
```

```
        n29        := n27 + sv38;
        sv2_o      <= n25 + n15;
        sv13_o     <= n17 + n28;
        sv18_o     <= n28;
        sv26_o     <= n9 + n11;
        sv38_o     <= n29;
        sv33_o     <= n19 + n29;
        sv39_o     <= n16 + n24;
        outp       <= n24;
    end process;
end ellipf;
```

VHDL 语言变量的说明和赋值只能出现在进程、函数和过程中。与信号赋值不同，变量赋值是在该语句执行时立即生效，而无需经过一个 delta 延迟。此外，在子程序中说明的变量，每当子程序被调用时，都被重新初始化。例如函数“+”中的变量 CARRY 和 NUM。

在结构体中，采用一系列的变量赋值语句完成输入与输出之间的转换，从而实现滤波器的功能。

由于所有操作数均为 bit_vector 类型，因而该进程中出现的所有加法运算实际上都是对文件 pack.vhd 中函数“+”的调用。

3. 模拟结果分析

验证五阶椭圆滤波器的行为功能，应当提供一组针对滤波器各种操作的较完备的测试模板。对于数字滤波器，尽管任何输入都将会执行滤波器的所有操作，但是不同的测试向量能够更全面地进行验证。文件 67_test_vectors.vhd 就是按照这样一个策略编写的测试台，包含 6 组测试向量，可对滤波器进行较全面的验证。源描述如下：

```
--定义 ellipf 的测试台实体（空实体）
entity E is
end entity;

architecture A of E is
    --被测元件的端口说明
    component ellipf
        port (
            inp      : in  BIT_VECTOR(15 downto 0);
            outp     : out Bit_Vector(15 downto 0);
            sv2      : in  bit_vector(15 downto 0);
            sv13     : in  Bit_Vector(15 downto 0);
            sv18     : in  Bit_Vector(15 downto 0);
```

```
                sv26      : in  Bit_Vector(15 downto 0);
                sv33      : in  Bit_Vector(15 downto 0);
                sv38      : in  Bit_Vector(15 downto 0);
                sv39      : in  Bit_Vector(15 downto 0);
                sv2_o     : out  bit_vector(15 downto 0);
                sv13_o    : out Bit_Vector(15 downto 0);
                sv18_o    : out  Bit_Vector(15 downto 0);
                sv26_o    : out  Bit_Vector(15 downto 0);
                sv33_o    : out  Bit_Vector(15 downto 0);
                sv38_o    : out  Bit_Vector(15 downto 0);
                sv39_o    : out  Bit_Vector(15 downto 0)
    );
end component ;

--激励信号
signal inp     : bit vector(15 downto 0);
signal outp    : bit_vector(15 downto 0);
signal sv2     : bit_vector(15 downto 0);
signal sv13    : bit_vector(15 downto 0);
signal sv18    : bit_vector(15 downto 0);
signal sv26    : bit_vector(15 downto 0);
signal sv33    : bit_vector(15 downto 0);
signal sv38    : bit_vector(15 downto 0);
signal sv39    : bit_vector(15 downto 0);
signal sv2_o   : bit_vector(15 downto 0);
signal sv13_o  : bit_vector(15 downto 0);
signal sv18 o  : bit_vector(15 downto 0);
signal sv26_o  : bit_vector(15 downto 0);
signal sv33 o  : bit_vector(15 downto 0);
signal sv38_o  : bit_vector(15 downto 0);
signal sv39 o  : bit_vector(15 downto 0);

--组装
for all : ellipf use entity work.ellipf(ellipf) ;

begin
--元件例示
INST1 : ellipf port map (   inp, outp, sv2, sv13, sv18, sv26,
                            sv33, sv38, sv39, sv2_o, sv13_o, sv18_o,
                            v26_o, sv33_o, sv38_o, sv39_o
```

```
                    );

process
begin
--产生激励
wait for 90 ns;
--
— 第1组测试向量
inp <= "0000000000000010";
sv2 <= "0000000000000001";
sv13 <= "0000000000000010";
sv18 <= "0000000000000001";
sv26 <= "0000000000000010";
sv33 <= "0000000000000001";
sv38 <= "0000000000000010";
sv39 <= "0000000000000001";
wait for 100 ns;
--判断结果
assert (outp = "0000000000001111")
report
"Assert 0 : < outp /= 0000000000001111 >"
severity warning;

assert (sv2_o = "0000000000110001")
report
"Assert 0 : < sv2_o /= 0000000000110001 >"
severity warning;

assert (sv13_o = "0000000000111111")
report
"Assert 0 : < sv13 /= 0000000000111111 >"
severity warning;

assert (sv18_o = "0000000000100110")
report
"Assert 0 : < sv18 /= 0000000000100110 >"
severity warning;

assert (sv26_o = "0000000000100010")
report
```

```
"Assert 0 : < sv26 /= 0000000000100010 >"
severity warning;

assert (sv33_o = "0000000000110110")
report
"Assert 0 : < sv33 /= 0000000000110110 >"
severity warning;

assert (sv38_o = "0000000000011101")
report
"Assert 0 : < sv38 /= 0000000000011101 >"
severity warning;

assert (sv39_o = "0000000000011101")
report
"Assert 0 : < sv39 /= 0000000000011101 >"
severity warning;
...
-- 第6组测试向量
inp <= "0000000000000010";
sv2 <= "0000000000001000";
sv13 <= "0000000000010000";
sv18 <= "0000000000001000";
sv26 <= "0000000000010000";
sv33 <= "0000000000001000";
sv38 <= "0000000000010000";
sv39 <= "0000000000001000";
wait for 100 ns;

--判断结果
assert (outp = "0000000001101010")
report
"Assert 6 : < outp /= 0000000001101010 >"
severity warning;

assert (sv2_o = "0000000011111100")
report
"Assert 6 : < sv2 /= 0000000011111100 >"
severity warning;
```

```
assert (sv13_o = "0000000101011110")
report
"Assert 6 : < sv13 /= 0000000101011110 >"
severity warning;

assert (sv18_o = "0000000011011100")
report
"Assert 6 : < sv18 /= 0000000011011100 >"
severity warning;

assert (sv26_o = "0000000011011000")
report
"Assert 6 : < sv26 /= 0000000011011000 >"
severity warning;

assert (sv33_o = "0000000101111000")
report
"Assert 6 : < sv33 /= 0000000101111000 >"
severity warning;
assert (sv38_o = "0000000011001100")
report
"Assert 6 : < sv38 /= 0000000011001100 >"
severity warning;
assert (sv39_o = "0000000011001100")
report
"Assert 6 : < sv39 /= 0000000011001100 >"
severity warning;

--结束模拟
assert false
report "---End of Simulation---"
severity error;

end process;
end;
```

依次对文件 67_pack.vhd，67_ellipf.vhd 以及 67_test_vectors.vhd 编译之后，借助模拟器可对该设计进行模拟，模拟运行过程中没有出现描述中 **assert** 语句提示的信息，表明通过了模拟验证。模拟输出的部分波形如图 67.1 所示。

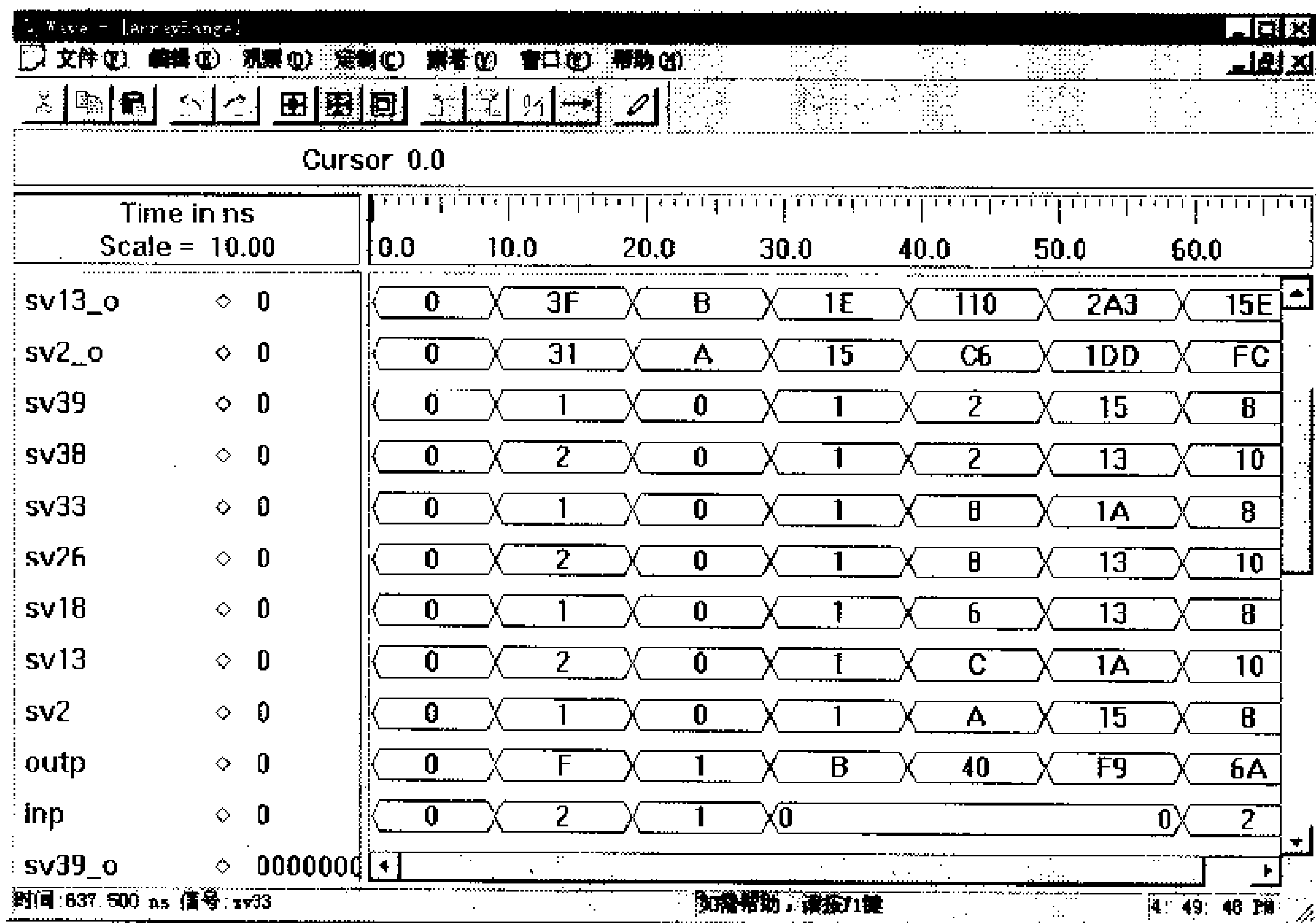


图 67.1　五阶椭圆滤波器模拟波形

（源描述文件名：67_pack.vhd
67_ellipf.vhd
测试平台文件名：67_test_vector.vhd）

第 68 例　闹钟系统的控制器

张东晓

1. 设计简介

从本例开始至第 75 例将说明 VHDL 语言在设计一个带闹钟功能的 24 小时计时器中的应用。计时器的外观如图 68.1 所示。它包括如下几个组成部分：

① 显示屏，由 4 个七段数码管组成，用于显示当前时间（时:分）或设置的闹钟时间；

② 数字键 '0' ~ '9'，用于输入新的时间或新的闹钟时间；

③ Time(时间)键，用于确定新的时间设置；

④ ALARM(闹钟)键，用于确定新的闹钟时间设置，或显示已设置的闹钟时间；

⑤ 扬声器，在当前时钟时间与闹钟时间相同时，发出蜂鸣声。

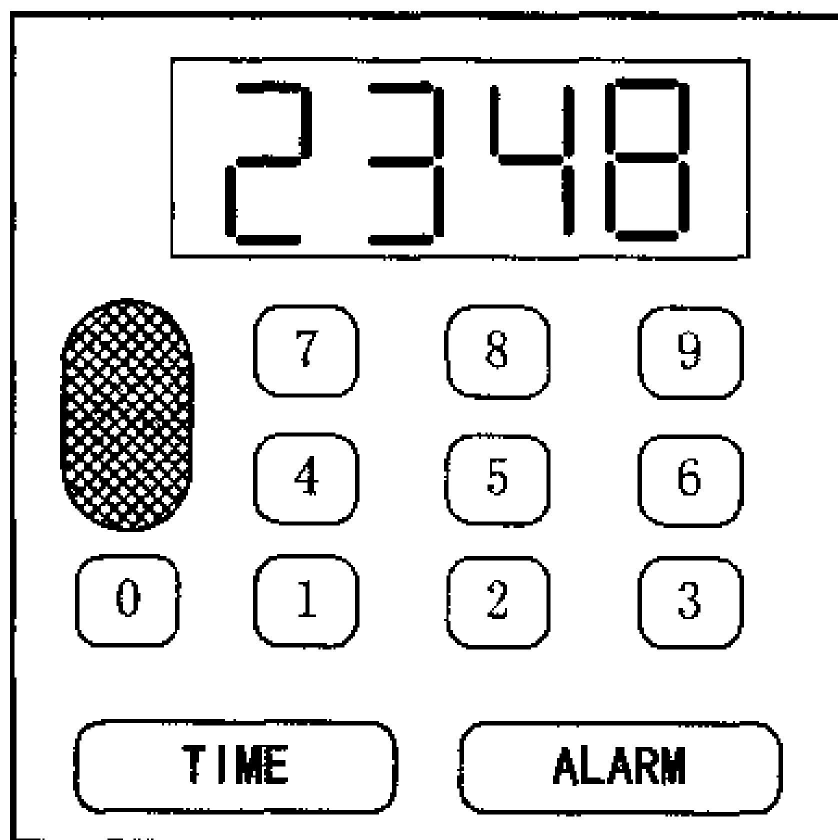


图 68.1　计时器外观

该计时器设计要完成如下功能：

① 计时功能：这是本计时器设计的基本功能，每隔一分钟计时一次，并在显示屏上显示当前时间；

② 闹钟功能：如果当前时间与设置的闹钟时间相同，则扬声器发出蜂鸣声；

③ 设置新的计时器时间：用户用数字键 "0" ~ "9" 输入新的时间，然后按 "Time" 键确认。在输入过程中，输入的数字在显示屏上从右到左依次显示。例如，用户要设置

新的时间 12：34，则按顺序输入"1"，"2"，"3"，"4"键，与之对应，显示屏上依次显示的信息为："1"，"12"，"123"，"1234"。如果用户在输入任意几个数字后较长时间内，例如 5 秒，没有按任何键，则计时器恢复到正常的计时显示状态。

④ 设置新的闹钟时间：用户用数字键"0"～"9"输入新的时间，然后按"ALARM"键确认。过程与 3 类似。

⑤ 显示所设置的闹钟时间：在正常计时显示状态下，用户直接按下"ALARM"键，则已设置的闹钟时间显示在显示屏上。

根据上述设计的功能，大致可以初步设想出整个设计会由哪些部分组成。例如，可能包含用于键盘输入的缓冲器，用于时钟计数的计数器，用于保存闹钟时间的寄存器，用于显示的七段数码显示电路等。但其中最主要部分的是控制各个部分协同工作的电路，即控制器，以按照设计功能产生适当的时序控制。

2. 电路系统工作原理

控制器命名为 alarm_controller，其外部端口如图 68.2所示：

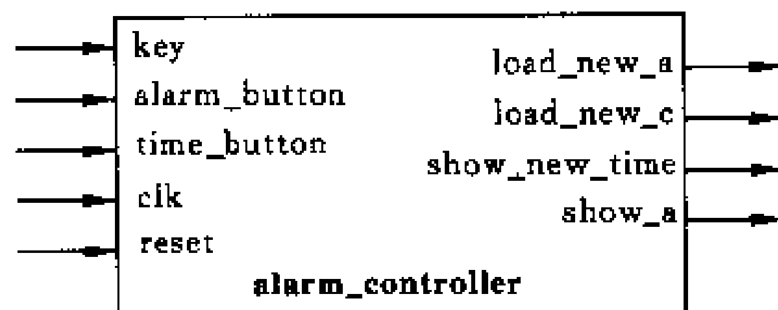


图 68.2 控制器的外部端口

根据图 68.2，控制器 alarm_controller 的 VHDL 实体说明如下：

```
entity alarm_controller is
port(key            : in std_logic;
     alarm_button   : in std_logic;
     time_button    : in std_logic;
     clk            : in std_logic;
     reset          : in std_logic;
     load_new_a     : out std_logic;
     load_new_c     : out std_logic;
     show_new_time  : out std_logic;
     show_a         : out std_logic
     );
end alarm_controller;
```

其中，clk 为外部时钟信号，reset 为复位信号。其他输入端口的说明如下：

当 key 为高电平(key='1')时，表示用户按下数字键("0"~"9")；

当 alarm_button 为高电平时，表示用户按下 "ALARM" 键；

当 time_button 为高电平时，表示用户按下 "TIME" 键。

各输出端口的作用如下：

当 load_new_a 为高电平时，控制（闹钟时间寄存器）加载新的闹钟时间值；

当 load_new_c 为高电平时，控制（时钟计数器）设置新的时间值；

当 show new time 为高电平时，控制（七段数码显示电路）显示新的时间值，即用户正在通过数字键输入的时间；否则：

当 show_new_time 为低电平时，根据 show_a 信号的值控制显示当前时间或闹钟时间。此时，当 show a 为高电平时，控制显示闹钟时间，否则，显示当前时间。

可以通过有限状态自动机（FSM）的方式来实现控制器的功能。根据设计要求及端口设置，需要 5 个状态来实现：

S0：表示电路初态即正常时钟计数状态，完成前面设计功能 1 的工作；

S1：接收键盘输入状态。在状态 S0 时用户按下数字键后进入此状态。在此状态下，显示屏上显示的是用户键入的数字。

S2：设置新的闹钟时间。在状态 S1 时用户按下 ALARM 键后进入此状态。

S3：设置新的计时器时间。在状态 S1 时用户按下 TIME 键后进入此状态。

表 68.1 控制器状态转换及控制输出表

当前状态	控制输入（条件）	下一状态	控制输出（动作）
S0	key = '1'	S1	show_new_time <= '1'
	alarm_button = '1'	S4	Show_a <= '1'
	否则	S0	--
S1	key = '1'	S1	show_new_time <= '1'
	alarm_button = '1'	S2	load_new_a <= '1'
	time_button = '1'	S3	Load_new_c <= '1'
	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]
S2	alarm_button = '1'	S2	load_new_a <= '1'
	否则	S0	--
S3	Time_button = '1'	S3	load_new_c <= '1'
	否则	S0	--
S4	Alarm_button = '1'	S4	show_a <= '1'
	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]

S4：显示闹钟时间。在状态 S0 时用户直接按下 ALARM 键后进入此状态。在此状态下，显示屏上显示的是所设置的闹钟时间。注，在此状态下，用户按下 ALARM 键后，显示屏上保持显示闹钟时间经过一段时间以后，再返回状态 S0 显示计时器时间。

相应的状态转换及控制如表 68.1 所示。

表 68.1 中没有显式说明的控制信号赋值，表示信号的值为零。例如在状态 S0，当信号 key = '1'时，show_new_time 信号的值赋为'1'，而其他信号 load_new_a、load_new_c 和 show_a 的值此时都赋为'0'。另外，表中带阴影部分的处理细节将在下文中详述。

3. VHDL 描述要点说明

(1) 有限状态自动机(FSM)的描述

如上所述，本控制器利用有限状态自动机(FSM)实现。FSM 状态自动机可以形式化地描述成五元组的形式：<S, I, O, f: S×I →S, h>。其中 S 为状态集合，I 为输入集合，O 为输出集合，f 为状态转换函数，h 为控制输出函数。当 FSM 的输出与输入及当前状态都相关时，即：h: S×I → O，称作 Mealy(有限状态自动)机，当 FSM 的输出仅与当前状态相关时，称作 Moore 机。Mealy 机及 Moore 机的模型如图 68.3 所示。

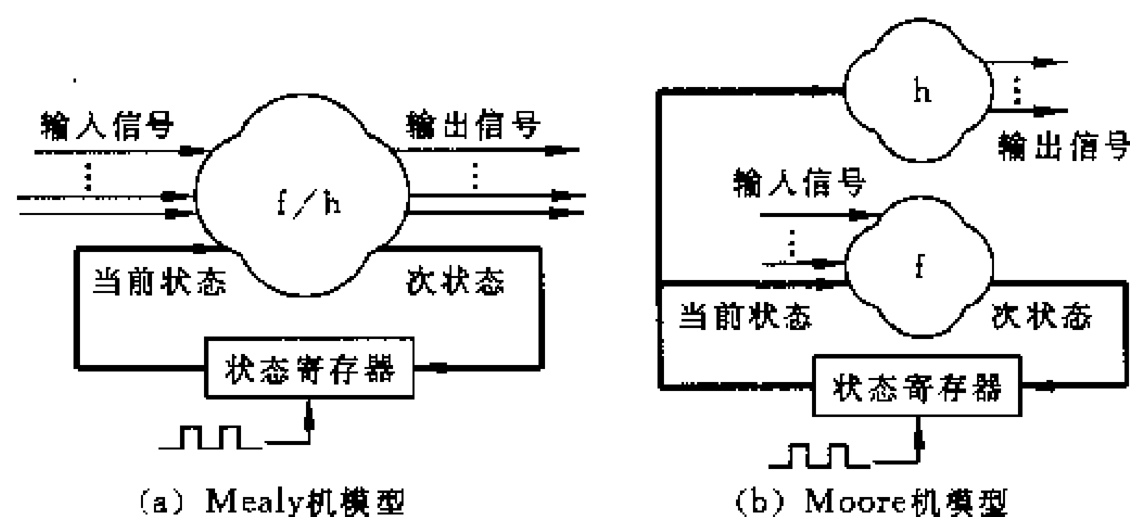


图 68.3 有限状态自动机的模型

描述 FSM 时首先要定义状态信号：当前状态和后继状态。状态信号的类型可定义为整型，但通常的做法是：首先定义一个枚举类型，其中列举 FSM 中用到的所有状态。例如对于本设计，可定义如下类型

```
type t_state is (s0, s1, s2, s3, s4);
```

然后再定义状态信号：

```
signal curr_state  : t_state;
signal next_state  : t_state;
```

这样做，一方面直观，易于理解；另一方面，将设计描述用于综合时便于综合器确定状

态数。

此外，参照图 68.3 可以看出，状态更新只有当状态寄存器的时钟输入信号上升（或下降）沿到来时才进行。在时钟边沿到来之前的时间区段内，对于 Mealy 机，输入信号的变化会影响到输出信号；而对于 Moore 机，输入信号的变化对输出信号不产生任何影响。这一区别造成了两种类型有限状态自动机描述方式的差异。

在描述 Mealy 机时，参照图 68.3 (a)，可以看出它分为两个部分，即状态寄存器和 f/h 产生逻辑部分。因此，一般在描述时分成两个进程。状态寄存器的描述方式为：

```
process(clk, reset)
begin
   if reset = '1'  then
      curr_state <= s0;
   elsif rising_edge(clk)  then
      curr_state <= next_state;
   end if;
end process;
```

其中 clk 为外部时钟信号，reset 为复位信号（图 68.3 中未画出），用于将状态寄存器置于初始状态。可以看出，上面的描述是一个异步复位的寄存器。若想描述同步复位的寄存器，可按下述方式进行：

```
process(clk)
begin
     if rising_edge(clk)  then
         if (reset = '1' ) then  curr_state <= s0;
         else  curr_state <= next_state;
         end if;
     end if;
end process;
```

注意，在上述描述中，进程语句敏感信号表中只有时钟信号 clk。

f/h 产生逻辑部分，可以按如下方式描述：

```
process(curr_state, key, alarm_button,  time_button,  ...)
begin
   -- 初始化
   next_state      <= curr_state;
   load_new_a      <= '0' ;
   load_new_c      <= '0' ;
   ...
   --
   case curr_state is
```

```
    when s0 =>                    -- 初始状态
        -- next state ?
        if (key = ' 1' ) then
            next_state <= s1;
            show_new_time <= ' 1' ;
        elsif (alarm_button = ' 1' ) then
            next_state <= s4;
            show_a <= ' 1' ;
        else
            next_state <= s0;
        end if;
    when ...
        ...
    when others =>
    null;
    end case;
end process;
```

f/h 产生逻辑是组合逻辑。在描述组合逻辑时，应注意如下两点：

① 组合逻辑的所有输入信号必须包含在进程语句的敏感信号表中，在描述 FSM 时，初学者最常犯的错误是忘记将当前状态信号(curr_state)放在敏感信号表中。

② 在进程语句中，无论语句执行经过何种路径，到进程结束时必须保证所有的输出信号都被赋值。否则对某一输出信号就可能产生锁存。由于一般 FSM 的控制较复杂，输出信号也较多，因此在描述时在进行条件判断之前需对所有的输出信号进行赋值，表示各相应信号的缺省值。

在描述 Moore 机时，参照图 68.3 (b)，虽然可以像写 Mealy 机那样，将整个描述分为 f 产生逻辑、h 产生逻辑以及状态控制器 3 部分。但如前所述，对于 Moore 机，在时钟边沿到来之前的时间区段内，输入信号的变化对输出信号不产生任何影响。另外，通过分析 Moore 机的特性可以看出，虽然在时钟边沿到来之前输入信号的变化会影响下一状态的值，但下一状态的有效值最终取决于时钟边沿时刻当前状态与输入信号的值，而此前下一状态值的变化都不起作用。因此可以按如下方式描述 Moore 机：

```
process(clk, reset)
begin
    if (reset = ' 1' )
        curr_state <= s0;
    else if (clk'EVENT and clk = ' 1' ) then
        -- initializtion
        next state      <= curr_state;
        load_new_a      <= ' 0' ;
```

```
load_new_c      <= '0' ;
...
--
case curr_state is
when s0 =>                    -- 初始状态
   -- next state ?
   if (key = '1' ) then
      next_state <= s1;
      show_new_time <= '1' ;
   elsif (alarm_button = '1' ) then
      next_state <= s4;
      show_a <= '1' ;
   else
      next_state <= s0;
   end if;
when ...
   ...
when others =>
   null;
end case;
end if;
end process;
```

(2) 用于判断超时的计数器的描述

在表 68.1中，状态 S1 和 S4 涉及到超时判断的问题：如果没有超时，则一直停留在当前状态；否则，转到其他状态。具体描述时，不能够使用 **wait** 语句，因为描述 f/h 逻辑的进程是由敏感信号触发的。因此，可以考虑利用外部时钟信号进行计数。例如，若外部时钟周期为 10 ms，预定超时时间为 5 s，则对时钟计数 500 次后经过的时间就是超时时间。但何时开始计数，如何得知计数结束（超时），必须引入新的内部信号。这样，整个控制器实际上分为两个部分，一部分是有限状态自动机，另一部分完成计数功能用以进行超时判断，如图 68.4所示。从图中可以看出，有限状态自动机除了要处理外部端口的输入/输出信号外，还要处理内部信号，这在描述时是需要注意的。

以状态 S1 中的超时判断为例，引入 3 个内部信号 enable_count_k，count_k_end 及 counter_k，分别定义如下：

```
signal counter_k      : t_int16;
signal enable_count_k : std_logic;
signal count_k_end    : std_logic;
```

其中，counter_k 用于进行计数，其类型 t_int16 定义为：

```
subtype t_int16 is integer range 0 to 65535
```

信号 enable_count_k 用于启动时钟计数，高电平有效；count_k_end 用于表示计数是否结束，高电平有效。

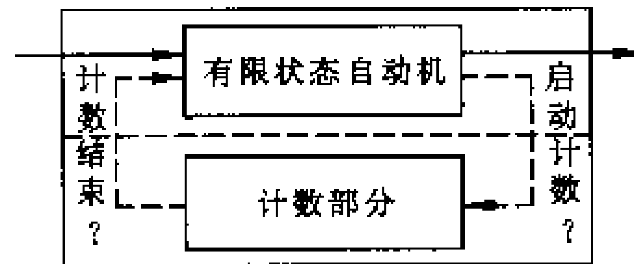


图 68.4 控制器内部结构

另外定义常量 key_timeout 表示计数次数：

```
constant key_timeout : t_int16 := 500;
```

若时钟周期为 10 ms，则该常量表示的超时时间为 5s。不难看出，将计数次数定义为常量能够很容易地表示各种超时时间，并且无需改动描述的其他部分。

下面是计数器的描述：

```
count_key : process(enable_count_k, clk)
begin
   if (enable_count_k = '0') then
      counter_k    <= 0;
      count_k_end <= '0';
   elsif (rising_edge(clk)) then
      if (counter_k >= key_timeout) then -- 时间到了吗?
         count_k_end <= '1';
      else
         counter_k <= counter_k + 1;
      end if;
   end if;
end process;
```

其中，信号 enable_count_k 实际上可以看成是一个异步复位信号，只不过是在低电平时起作用。在状态 S1 中，如何启动计数，如何判断状态结束的描述如下：

```
...
when s1 =>                         -- 按下了键
      next state ?
    if (key = '1') then
        next state <= s1;
    elsif (alarm button = '1') then
```

```
        next_state <= s2;
        load_new_a <= '1';
    elsif (time_button = '1') then
         next_state <= s3;
         load_new_c <= '1';
    else
         if (count_k_end = '1') then  -- 时间到了吗?
            next_state <= s0;
         else
            show_new_time  <= '1';
            next_state <= s1;
         end if;
         enable_count_k <= '1';
    end if;
    --
when S2 =>
   ...
```

这样就完成了状态 S1 中的超时判断处理，而状态 S4 中的处理类似。

（源描述文件名：68_alarm_controller.vhd
68_tb_alarm_controller.vhd）

第69例　闹钟系统的译码器

陈东瑛

1. 电路系统工作原理

本模块的功能是将每次按下闹钟系统的数字键盘后产生的一个数字所对应的十位二进制数据信号，转换为一位十进制整数信号，以作为小时、分钟计数的4个数字之一，如图69.1所示。其中keypad为输入端口，接收十位二进制数据信号；value为输出端口，输出相应的一位十进制整数信号。输入数据与输出数据的译码关系见表69.1。

图 69.1　电路系统示意图

表 69.1　输入、输出数据的译码关系

输入	0000000001	0000000010	0000000100	0000001000	0000010000
输出	0	1	2	3	4
输入	0000100000	0001000000	0010000000	0100000000	1000000000
输出	5	6	7	8	9

2. 电路的 VHDL 语言描述方法及语法分析

在介绍本译码器的VHDL语言设计描述之前，首先要了解闹钟系统共享信息所在的包p_alarm。除控制器组件之外，所有闹钟系统的模块及其测试台以及整体合成的VHDL语言设计描述都引用了该包的相关内容。包p_alarm在文件69_p_alarm_clock.vhd中描述。

```
-- 闹钟系统共享信息描述文件69_p_alarm_clock.vhd
library ieee;
use ieee.std_logic_1164.all;
--------------------
package p_alarm is
--------------------
   subtype t_digital is integer range  0 to 9;
```

```
    subtype t_short   is integer range  0 to 65535;
    type t_clock_time is array (3 downto 0) of t_digital;
    type t_display is array (3 downto 0) of std_logic_vector(6 downto 0);
    type SEG7 is array (0 to 9) of std_logic_vector(6 downto 0);

    constant seven_seg: SEG7 := ( "0111111", "0000110", "1011011",
                                  "1001111", "1100110", "1101101",
                                  "1111101", "0000111", "1111111",
                                  "1110011" );
    type keypad9 is array (0 to 9) of std_logic_vector(9 downto 0);
    constant keynumber: keypad9 := (
                                    "0000000001",   -- 0
                                    "0000000010",   -- 1
                                    "0000000100",   -- 2
                                    "0000001000",   -- 3
                                    "0000010000",   -- 4
                                    "0000100000",   -- 5
                                    "0001000000",   -- 6
                                    "0010000000",   -- 7
                                    "0100000000",   -- 8
                                    "1000000000"    -- 9
                                    );
end p_alarm;
```

闹钟系统以四位十进制整数表示时间，其特定类型为 t_clock_time，每个数字的类型为 t_digital。而译码器输入的十位二进制数据信号的每一位，为 std_logic 九值逻辑分辨类型，体现了电路的硬件特性；常量数组 keynumber 的每一个分量下标及其内容，体现了表 69.1 的译码关系，用于简化闹钟系统外部激励（数字键盘输入）的产生。

闹钟系统以 4 个七段数码显示器显示时间，输出给 4 个七段数码显示器的数据信息的类型为 t_display。每个七段数码显示器对应七位二进制数据，均为 std_logic 九值逻辑分辨类型，体现了电路的硬件特性；常量数组 seven_seg 的每一个分量下标及其内容，体现了七段数码显示器显示数字与七位二进制数据的对应关系，见表 69.2。此七段

表 69.2　七段数码显示器

0111111	0000110	1011011	1001111	1100110
0	1	2	3	4
1101101	1111101	0000111	1111111	1110011
5	6	7	8	9

数码显示器与第 54 例——分秒计数器的七段数码显示器的逻辑相反（见表 54.1）。这些类型及常量用于闹钟系统的显示驱动器组件的 VHDL 语言的设计描述。

本译码器的 VHDL 语言描述文件为 69_decoder.vhd，其中以实体说明 decoder 体现电路系统的外部接口，以结构体 rtl 实现其功能。以选择信号赋值语句实现译码功能是一种典型的方法，也可以采用与之等价的条件信号赋值语句，或者以等价的进程——以 **case** 语句和顺序信号赋值语句为主体，实现译码的功能。注意，**others** 分支处理 10 组有效数据之外的所有可能数据的译码，本译码器规定将所有无效数据译码为整数——0。

```
-- 闹钟系统的译码器的描述文件 69_decoder.vhd
library ieee;
use ieee.std_logic_1164.all;
use work.p_alarm.all;
----------------
entity decoder is
----------------
port(
    keypad :in  std_logic_vector(9 downto 0);
    value  :out t_digital
    );
end decoder;

architecture rtl of decoder is
begin
   with keypad select
   value <= 0  when "0000000001",
            1  when "0000000010",
            2  when "0000000100",
            3  when "0000001000",
            4  when "0000010000",
            5  when "0000100000",
            6  when "0001000000",
            7  when "0010000000",
            8  when "0100000000",
            9  when "1000000000",
            0  when others;
 end rtl;
```

对照第 59 例的 2-4 译码器来看，本译码器的输出为一位十进制整数，更倾向于行为级的设计，如果由实际数字电路元件来实现，还要转换为形如第 59 例的 2-4 译码器输出的多路二进制信号。

3. 模拟测试向量的选择及模拟结果分析

本模块测试台的 VHDL 语言描述文件为 69_tb_decoder.vhd，外部激励信号 keypad 通过虚拟设计实体 tb_decoder 的元件例示语句，将多组激励向量送入设计实体 decoder，以检验本模块对各种输入可能的响应是否实现了功能要求。

以 Vsim/Talent 系统模拟运行本测试台，验证了本组件 VHDL 语言设计描述的正确性。

（源描述文件名：69_decoder.vhd
69_p_alarm_clock.vhd
测试平台文件名：69_tb_decoder.vhd）

第70例　闹钟系统的移位寄存器

陈东瑛

1. 移位寄存器的电路系统工作原理

本模块的功能是在clk端口输入信号的上升沿同步下，将key端口的输入信号移入new_time端口的输出信号最低位，原有信息依次向左移，最高位信息丢失；而reset端口的输入信号对new_time端口输出信号进行异步清零复位。电路系统示意图如图70.1所示。

图70.1　电路系统示意图

2. 移位寄存器电路的VHDL语言描述方法及语法分析

本组件的VHDL语言描述文件为70_buffer.vhd，其中以实体说明key_buffer体现电路系统的外部接口，以结构体rtl实现其功能。

```
-- 闹钟系统的移位寄存器的描述 70_buffer.vhd
library ieee;
use ieee.std_logic_1164.all;
use work.p_alarm.all;
---------------------
entity key_buffer is
---------------------
port(key      : in  t_digital;
     clk      : in  std_logic;
     reset    : in  std_logic;
     new_time : out t_clock_time
     );
end key_buffer;

architecture rtl of key_buffer is
signal n_t:t_clock_time;
```

```
begin
   shift:process(reset,clk)
   begin
      if (reset = '1') then
         n_t <= (0,0,0,0);
      elsif (clk'EVENT and clk = '1' )then
         for i in 3 downto 1 loop
            n_t(i) <= n_t(i-1);
         end loop;
         n_t(0) <= key;
      end if;
   end process;
   new_time <= n_t;
end rtl;
```

key 端口的数据类型为 t_digital，new_time 端口的数据类型为 t_clock_time，符合闹钟系统时间表示的统一规定；而 clk 和 reset 端口的数据类型为 std_logic，体现了电路的硬件特性。

在本模块的 VHDL 语言设计描述中，由 **if** 语句实现 reset 和 clk 端口的控制功能。针对移位的功能要求，VHDL 语言描述中既需向 new_time 端口输出信息，又需从 new_time 端口读取信息，则 new_time 端口的模式应为 **inout**；但是在电路实现时，**inout** 模式的端口需要用三态门等特殊元件实现，为简化最终的硬件实现，本例提供了一个基于 **out** 模式的设计。

插入一个中间信号 n_t，移位或清零时的读写操作都针对 n_t 进行，同时使 new_time 端口始终与 n_t 保持一致，以实现本模块的功能，对 n_t 的移位操作也可以通过聚集的方式更简单地实现——n_t <= (n_t(2), n_t(1), n_t(0), key);。所以本模块的 VHDL 语言设计描述由两个进程实现：其一，以 reset 或 clk 端口的输入信号为敏感信号，实现对中间信号 n_t 的移位功能；其二，并发信号赋值语句等价于以 n_t 为敏感信号的、对 new_time 端口赋值的进程，保持两者的一致。

3. 移位寄存器模拟测试向量的选择及模拟结果分析

本模块测试台的 VHDL 语言描述文件为 70_tb_buffer.vhd，外部激励信号通过虚拟设计实体 tb_buffer 的元件例示语句，将多组激励向量送入设计实体 buffer，以检验本组件对各种输入可能的响应是否实现了功能要求。

以 Vsim/Talent 系统模拟运行本测试台，验证了本组件 VHDL 语言设计描述的正确性。

（源描述文件名：70_buffer.vhd
测试平台文件名：70_tb_buffer.vhd）

第 71 例　闹钟系统的闹钟寄存器和时间计数器

陈东瑛

1. 电路系统工作原理

闹钟寄存器模块的功能是在时钟上升沿同步下，根据 load_new_a 端口的输入信号控制 alarm_time 端口的输出，当控制信号有效（高电平）时，把 new_alarm_time 端口的输入信号值输出；而 reset 端口输入信号对 alarm_time 端口的输出进行异步的清零复位。图 71.1 是闹钟寄存器模块的示意图。

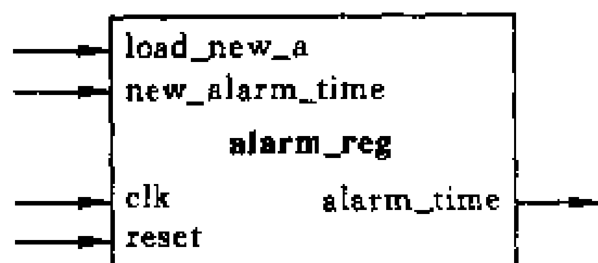


图 71.1　闹钟寄存器示意图

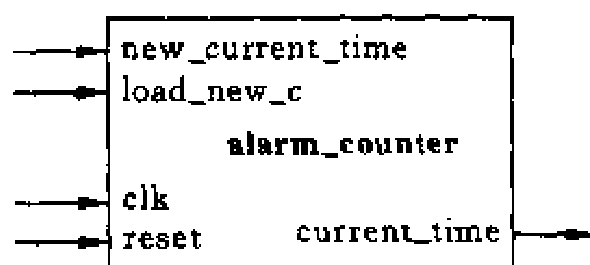


图 71.2　时间计数器示意图

闹钟系统的闹钟时间由闹钟寄存器保存和传递，而当前时间由时间计数器保存、传递并按分钟累加推进。这两个组件的功能和设计描述比较相似，它们之间的区别主要在于自动累加功能的有无和控制信号的优先作用次序。

时间计数器模块的功能是当 reset 端口输入信号有效（高电平）时，对 current_time 端口输出信号清零复位；当 load_new_c 端口输入信号有效（高电平）时，将 new_current_time 端口的输入信号输出给 current _time 端口。reset 端口的控制优先于 load_new_c 端口。当这两个控制信号都无效时，在时钟上升沿同步下，对 current _time 端口输出信号累加 1，并根据小时、分钟的规律处理进位。图 71.2 是时间计数器模块的示意图。

2. 电路的 VHDL 语言描述方法及语法分析

闹钟寄存器组件的 VHDL 语言描述文件为 71_alarm_reg.vhd，其中以实体说明 alarm_reg 体现电路系统的外部接口，以结构体 rtl 实现其功能。

```
-- 闹钟寄存器的描述文件 71_alarm_reg.vhd
library ieee;
use ieee.std_logic_1164.all;
use work.p_alarm.all;
```

```
------------------
entity alarm_reg is
------------------
   port(new_alarm_time: in  t_clock_time;
        load_new_a     : in  std_logic;
        clk            : in  std_logic;
        reset          : in  std_logic;
        alarm_time     : out t_clock_time
        );
end alarm_reg;

architecture rtl of alarm_reg is
begin
   process(clk, reset)
   begin
      if reset = '1' then
         alarm_time <= (0,0,0,0);
      else
         if rising_edge(clk) then
            if load_new_a = '1' then
               alarm_time <= new_alarm_time;
            elsif load_new_a /= '0' then
               assert false report "Uncertain load_new_alarm control!"
               severity warning;
            end if;
         end if;
      end if;
   end process;
end rtl;
```

new_alarm_time 端口、alarm_time 端口的数据类型为 t_clock_time，符合闹钟系统时间表示的统一规定；clk、load_new_a、reset 端口为 std_logic 类型，体现了电路的硬件特性。

闹钟寄存器组件的控制功能主要由 **if** 语句实现，对清零复位信号——reset 端口的检查优先于对其他输入端口的检查，而且控制信号是在时钟上升沿同步下起作用，决定了 **if** 语句中各个条件判断的嵌套关系和顺序。整个设计由一个进程实现，在 clk 或 reset 端口信号活跃时激活。

闹钟寄存器组件的 VHDL 设计描述以聚集的形式简化了对数组的赋值，并调用了 IEEE 的标准函数 rising_edge()，判断时钟信号 clk 是否处于上升沿状态；**assert** 语句在控制信号混乱的情况下输出提示信息，考虑了硬件电路的非理想状态。

时间计数器的 VHDL 语言描述文件为 71_alarm_counter.vhd，其中以实体说明 alarm_counter 体现电路系统的外部接口，以结构体 rtl 实现其功能。

```
--时间计数器的描述文件 71_alarm_counter.vhd
library ieee;
use ieee.std_logic_1164.all;
use work.p_alarm.all;
------------------------
entity alarm_counter is
------------------------
   port(new_current_time: in  t_clock_time;
        load_new_c      : in  std_logic;
        clk             : in  std_logic;
        reset           : in  std_logic;
        current_time    : out t_clock_time
         );
end alarm_counter;

architecture rtl of alarm_counter is
   signal i_current_time : t_clock_time;
begin
   process(clk, reset, load_new_c)
      variable c_t:t_clock_time;
   begin
      if reset = '1' then
         i_current_time <= (0,0,0,0);
      elsif load_new_c = '1' then
            i_current_time <= new_current_time;
      elsif rising_edge(clk) then
            c_t := i_current_time;
            if c_t(0) < 9 then
               c_t(0) := c_t(0) + 1;
            else
               c_t(0) := 0;
               if c_t(1) < 5 then
                  c_t(1) := c_t(1) + 1;
               else
                  c_t(1) := 0;
                  if c_t(3) < 2 then
                     if c_t(2) < 9 then
                        c_t(2) := c_t(2) + 1;
                     else
```

```
                        c_t(2) := 0;
                        c_t(3) := c_t(3) + 1;
                    end if;
                else
                    if c_t(2) < 3 then
                        c_t(2) := c_t(2) + 1;
                    else
                        c_t(2) := 0;
                        c_t(3) := 0;
                    end if;
                end if;
            end if;
        end if;
        i_current_time <= c_t;
    end if;
end process;
current_time <= i_current_time;
end rtl;
```

new_current_time 端口和 current_time 端口的数据类型为 t_clock_time，符合闹钟系统时间表示的统一规定；load_new_c 端口和 reset 端口和 clk 端口的数据类型为 std_logic，体现了电路的硬件特性。

与第 70 例移位寄存器模块类似，时间计数器的 VHDL 语言描述中既要向 current_time 端口输出信息，又要从 current _time 端口读取信息，但是 current _time 端口未采用 **inout** 模式，而是以 **out** 模式实现。与闹钟系统的移位寄存器组件同理，加入一个中间信号 I_current_time，作用与移位寄存器组件中的 n_t 相同。

由于 load_new_c 端口的控制作用为异步的，它在 **if** 语句中条件判断的嵌套关系和顺序与移位寄存器组件中的 load_new_a 端口不同，而且还在进程敏感信号表中成为进程激活条件之一。

对 current _time 端口输出信息的累加操作通过一个中间变量实现，与第 73 例分频器组件中的计数变量 cnt 同理，不对信号对象直接操作，以免电路硬件特性影响纯算法行为的实现。

3. 模拟测试向量的选择及模拟结果分析

闹钟寄存器的测试台的 VHDL 语言描述文件为 71_tb_alarmreg.vhd，外部激励信号通过虚拟设计实体 tb_alarmreg 的元件例示语句，将多组激励向量送入设计实体 alarmreg，以检验本组件对各种输入可能的响应是否实现了功能要求。

时间计数器的测试台的 VHDL 语言描述文件为 71_tb_alarm_counter.vhd，外部激励

信号通过虚拟设计实体 tb_ alarm_counter 的元件例示语句，将多组激励向量送入设计实体 alarm_counter，以检验本组件对各种输入可能的响应是否实现了功能要求。

以 Vsim/Talent 系统模拟运行这两个测试台，验证了这两个组件的 VHDL 语言设计描述的正确性。

（源描述文件名：71_alarm_reg.vhd
71_alarm_counter.vhd
测试平台文件名：71_tb_alarm_reg.vhd
71_tb_alarm_counter.vhd）

第72例　闹钟系统的显示驱动器

陈东瑛

1. 电路系统工作原理

本模块的功能是：当 show_new_time 端口输入信号有效（高电平）时，根据 new_time 端口的输入信号（时间数据），产生相应的 4 个七段数码显示器的驱动数据，并在 display 端口输出该信号。当 show_new_time 端口输入信号无效（低电平）时，判断 show_a 端口的输入信号，为高电平时，根据 alarm_time 端口的输入信号（时间数据）产生相应的 4 个七段数码显示器的驱动数据，并在 display 端口输出该信号；为低电平时，根据 current_time 端口的输入信号，对 display 端口进行驱动。当 alarm_time 端口的输入信号值与 current_time 端口的输入信号值相同时，sound_alarm 端口的输出信号有效（高电平），反之无效。图 72.1 为显示驱动器示意图.

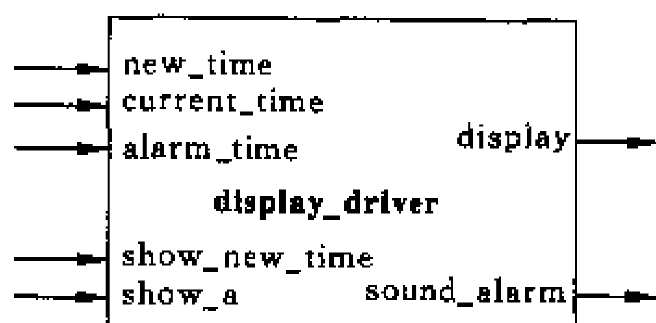


图 72.1　显示驱动器示意图

2. 电路的 VHDL 语言描述方法及语法分析

本组件的 VHDL 语言描述文件为 72_display_driver.vhd，其中以实体说明 display_driver 体现电路系统的外部接口，以结构体 rtl 实现其功能。

```
-- 显示驱动器的描述文件 72_display_driver.vhd
library ieee;
use ieee.std_logic_1164.all;
use work.p_alarm.all;
---------------------------------
entity display_driver is
---------------------------------
port(alarm_time   : in  t_clock_time;
     current_time : in  t_clock_time;
```

```
     new time       : in  t_clock_time;
     show_new_time: in  std_logic;
     show_a         : in  std_logic;
     sound_alarm   : out std_logic;
     display        : out t_display);
end display_driver;

architecture rtl of display_driver is
   signal display_time:t_clock_time;
begin
   -- 输出控制
   ctrl:process(alarm_time, current_time, new_time, show_a, show_new_time)
   begin
      --设置闹钟响铃或不响铃
      sound_lp: for i in alarm_time'RANGE loop
         if not(alarm_time(i) = current_time(i)) then
            sound_alarm <= '0';
            exit sound_lp;
         else
            sound_alarm <= '1';
         end if;
      end loop sound_lp;

      -- 选择显示时间
      if show_new_time = '1' then
         display_time <= new_time;
      elsif show_a = '1' then
         display_time <= alarm_time;
      elsif show_a = '0' then
         display_time <= current_time;
      else
         assert false report "Uncertain display_driver control!"
         severity error;
      end if;
    end process;

    -- 七段数码显示
    disp:process(display_time)
    begin
       for i in display_time'RANGE loop
          display(i) <= seven_seg(display_time(i));
       end loop;
```

```
    end process;
end rtl;
```

current_time，alarm_time 和 new_time 端口的数据类型为 t_clock_time，符合闹钟系统时间表示的统一规定；而 display 端口为 t_display 类型，控制端口 show_new_time 和 show_a 的数据类型为 std_logic，体现了电路的硬件特性。

本例的两个主要功能分别由两个进程实现，加入一个中间信号 display_time（类型为 t_clock_time），进程 ctrl 把多路选择的结果赋给中间信号 display_time；进程 disp 以中间信号 display_time 为敏感信号，将中间信号 display_time 的 4 位整数时间值，转换为 4 组七段数码显示器的二进制数据，送给 display 端口。进程 disp 根据常量数组 seven_seg 的下标与分量内容的对应关系，进行数据转换，是一种典型、简便的方法。而第 54 例中七段数码显示器的数据转换采用了一般的方法——**case** 语句。

3. 模拟测试向量的选择及模拟结果分析

本模块测试台的 VHDL 语言描述文件为 72_tb_display_driver.vhd，外部激励信号通过虚拟设计实体 tb_display_driver 的元件例示语句，将多组激励向量送入设计实体 display_driver，以检验本组件对各种输入可能的响应是否实现了多路选择和数据转换的功能要求。

以 Vsim/Talent 系统模拟运行本测试台，验证了本模块 VHDL 语言设计描述的正确性。

（源描述文件名：72_display_driver.vhd
测试平台文件名：72_tb_display_driver.vhd）

第73例　闹钟系统的分频器

陈东瑛

1. 电路系统工作原理

本模块的功能是将 clk_in 端口输入的时钟信号分频后送给 clk_out 端口。当 reset 端口输入信号有效（高电平）时，clk_out 端口输出信号清零。图 73.1 为分频器示意图。

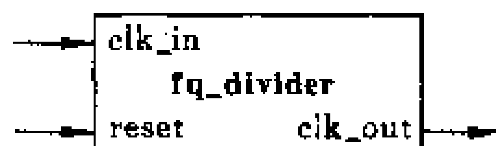


图 73.1　分频器示意图

2. 电路的 VHDL 语言描述方法及语法分析

本组件的 VHDL 语言描述文件为 73_fq_divider.vhd，其中以实体说明 fq_divider 体现电路系统的外部接口，以结构体 rtl 实现其功能。

```
-- 分频器的描述文件 73_fq_driver.vhd
library ieee;
use ieee.std_logic_1164.all;
use work.p_alarm.all;

entity fq_divider is
   port(
        clk_in      :in  std_logic;
        reset       :in  std_logic;
        clk_out     :out std_logic
        );
end fq_divider;

architecture rtl of fq_divider is
   constant divide_period : t_short := 6000;
begin
   divide_clk: process(clk_in,reset)
      variable cnt : t_short;
   begin
       if (reset = ' 1' ) then
```

```vhdl
            cnt := 0;
            clk_out <= '0';
        elsif rising_edge(clk_in) then
            if (cnt < (divide_period/2)) then
                clk_out <= '1';
                cnt := cnt + 1;
            elsif (cnt < (divide_period-1)) then
                clk_out <= '0';
                cnt := cnt + 1;
            else
                cnt := 0;
            end if;
        end if;
    end process; -- divide clk
end rtl;
```

所有端口的数据类型为 std_logic，体现了电路的硬件特性。

实际上分频器是计数器的一种具体应用，对输入时钟信号 clk_in 的上升沿进行计数，当达到规定数量时，产生相应的时钟信号 clk_out 输出，主要由 **if** 语句实现，由变量 cnt 实现计数，而不是信号。

3. 模拟测试向量的选择及模拟结果分析

本模块测试台的 VHDL 语言描述文件为 73_tb_fq_divider.vhd，外部激励信号通过虚拟设计实体 tb_fq_divider 的元件例示语句，将多组激励向量送入设计实体 fq_divider，以检验本组件是否实现了将周期为 10ms 的时钟信号分频为 1min 的时钟信号的功能要求。

以 Vsim/Talent 系统模拟运行本测试台，验证了本模块 VHDL 语言设计描述的正确性。

（源描述文件名：73_fq_divider.vhd
测试平台文件名：73_tb_fq_divider.vhd）

第 74 例　闹钟系统的整体组装

张东晓

1. 电路系统工作原理

前边已经说明了计时器各个组成部分的设计细节及相关描述，本例将把这些组成部分组装起来，形成完整的总体设计。该计时器命名为 alarm_clock，其外部端口如图 74.1 所示。设计的外部端口定义如下：

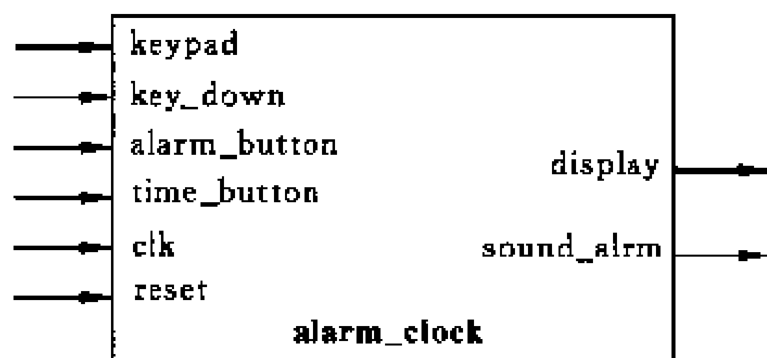


图 74.1　计时器的外部端口

```
entity alarm_clock is
    port(
        keypad       : in  std_logic_vector(9 downto 0);
        key_down     : in  std_logic;
        alarm_button: in  std_logic;
        time_button : in  std_logic;
        clk          : in  std_logic;
        reset        : in  std_logic;
        display      : out t_display;
        sound_alarm : out std_logic);
end alarm_clock;
```

其中 clk 为外部时钟信号，reset 为复位信号。其他输入端口的说明如下：

keypad 是一个十位信号，若其中某一位为高电平，则表示用户按下了相应下标的数字键，例如，若 keypa(5) = '1'，表示用户按下数字键 "5"。本设计不考虑用户同时按下多个数字键的情况，所以任意时刻 keypad 中只能有一位为 '1'；

当 keypad 为高电平时(keypad='1')，表示用户按下某一数字键；

当 alarm_button 为高电平时，表示用户按下 ALARM 键；

当 time_button 为高电平时，表示用户按下 TIME 键。

各输出端口的含义如下：

display 实际上表示了 4 个七段数码显示管，用于显示时间，如 12:20；

sound_alarm 用于控制扬声器发声，当 sound_alarm='1'时，扬声器发出蜂鸣，表示到了设定的闹钟时间。

设计的总体结构如图 74.2所示。下面再简要说明各组成部分的功能：

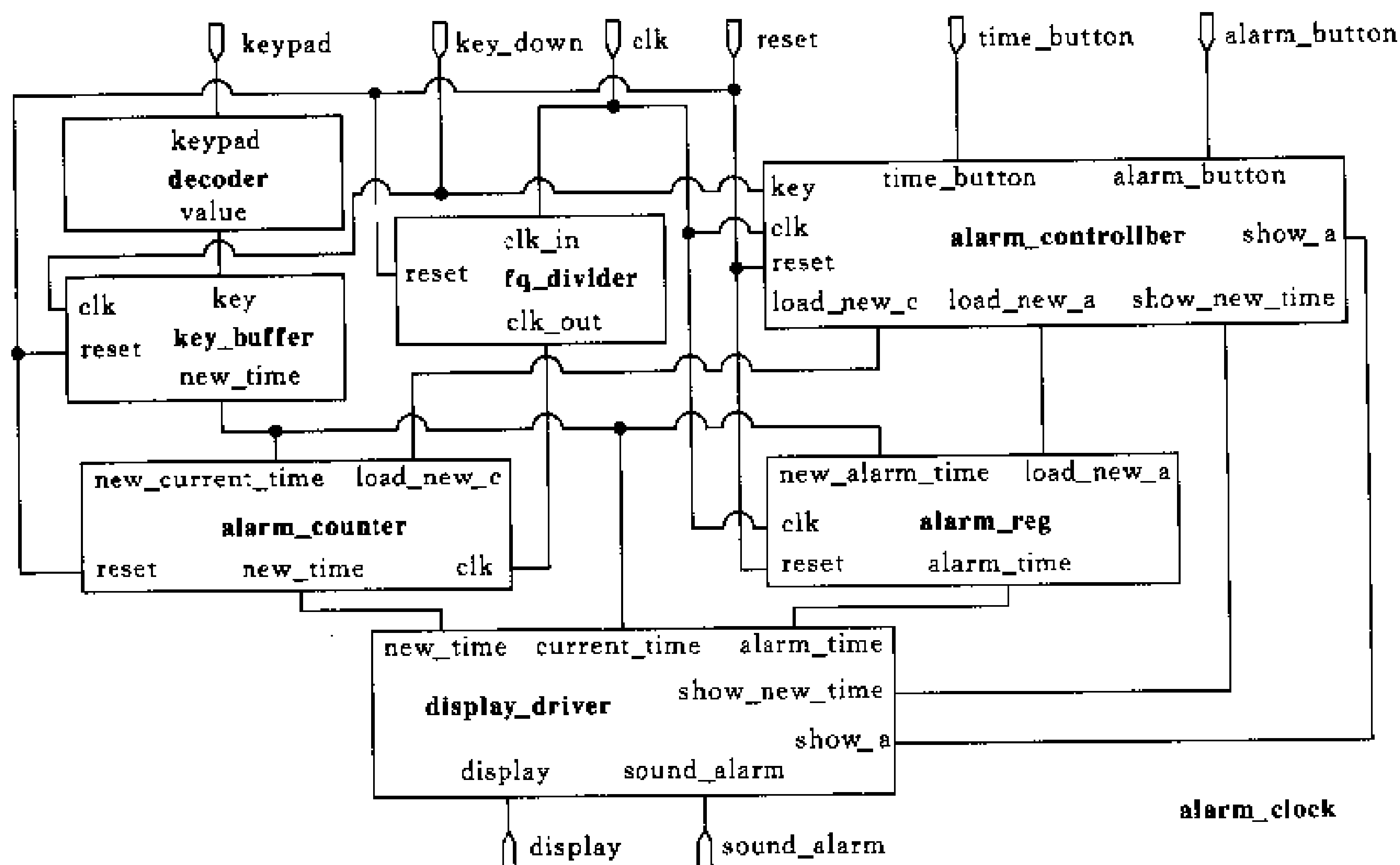


图 74.2　总体结构

译码器(decoder)将 keypad 信号转换为 0~9 的整型数，以直观地表示和处理用户输入的数字；

键盘缓冲器(key buffer)是一个移位寄存器，暂存用户键入的数字，并且实现用户键入数字在显示器上从右到左的依次显示。这里需要注意的是，由图 74.2中可以看出，key_buffer 的时钟端连接的是外部 key_down 信号。这表示用户每输入一个数字，key_buffer 移位一次；

分频器(fq_divider)将较高速的外部时钟频率分频成每分钟一次的时钟频率，以便进行时钟计数；

计数器(alarm_counter)实际上是一个异步复位、异步置数的累加器，通常情况下进行时钟累加计数，必要时可置入新的时钟值，然后从该值开始新的计数；

寄存器(alarm_reg)用于保存用户设置的闹钟时间，是一个异步复位寄存器；

显示器(display_driver)根据需要显示当前时间、用户设置的闹钟时间或用户通过键盘输入的新的时间，同时判断当前时间是否已到了闹钟时间，实际上是一个多路选择器加比较器；

控制器(alarm_controller)是设计的核心部分，按设计要求产生相应的控制逻辑控制其他各部分的工作。

2. VHDL 描述要点说明

本例没有什么特别的 VHDL 语法现象值得说明，只需按照总体结构，将第 68~73 例中的各部分设计描述通过元件例示语句组装起来即可。下面是本例的 VHDL 描述：

```
--闹钟系统整机组装描述文件 74_alarm_clock.vhd
library ieee;
use ieee.std_logic_1164.all;
use work.p_alarm.all;

entity alarm_clock is
   port( keypad      : in  std_logic_vector(9 downto 0);
         key_down    : in  std_logic;
         alarm_button: in  std_logic;
         time_button : in  std_logic;
         clk         : in  std_logic;
         reset       : in  std_logic;
         display     : out t_display;
         sound_alarm : out std_logic);
end alarm_clock;

architecture rtl of alarm_clock is
   component decoder
   port(keypad: in  std_logic_vector(9 downto 0);
        value : out t_digital);
   end component;

   component key_buffer
   port (key      : in  t_digital ;
         clk      : in  std_logic;
         reset    : in  std_logic;
         new_time: out t_clock_time);
   end component;

   component alarm_counter
   port(new_current_time: in  t_clock_time;
        load_new_c      : in  std_logic;
```

```
        clk            : in  std_logic;
        reset          : in  std_logic;
        current_time   : out t_clock_time);
end component;

component alarm_reg
port(new_alarm_time: in  t_clock_time;
        load_new_a    : in  std_logic;
        clk           : in  std_logic;
        reset         : in  std_logic;
        alarm_time    : out t_clock_time);
end component;

component alarm_controller
port(key              : in  std_logic;
        alarm_button  : in  std_logic;
        time_button   : in  std_logic;
        clk           : in  std_logic;
        reset         : in  std_logic;
        load_new_a    : out std_logic;
        load_new_c    : out std_logic;
        show_new_time : out std_logic;
        show_a        : out std_logic);
end component;

component display_driver
port(alarm_time       : in  t_clock_time;
        current_time  : in  t_clock_time;
        new_time      : in  t_clock_time;
        show_new_time : in  std_logic;
        show_a        : in  std_logic;
        sound_alarm   : out std_logic;
        display       : out t_display);
end component;

component fq_divider
port(clk_in    : in  std_logic;
        reset    : in  std_logic;
        clk_out  : out std_logic
        );
end component;

signal inner_key      : t_digital;
signal inner_time     : t_clock_time;
```

```
    signal inner_time_c   : t_clock_time;
    signal inner_time_a   : t_clock_time;
    signal inner_l_c      : std_logic;
    signal inner_l_a      : std_logic;
    signal inner_s_a      : std_logic;
    signal inner_s_n      : std_logic;
    signal inner_sec_clk  : std_logic;

    for all: decoder use entity work.decoder(rtl);
    for all: key_buffer use entity work.key_buffer(rtl);
    for all: alarm_counter use entity work.alarm_counter(rtl);
    for all: alarm_reg use entity work.alarm_reg(rtl);
    for all: alarm_controller use entity work.alarm_controller(rtl);
    for all: display_driver use entity work.display_driver(rtl);
    for all: fq_divider use entity work.fq_divider(rtl);

begin
    ---------------- decoder --------------
    u1: decoder port map(keypad, inner_key);

    -------------- keypad buffer -----------
    u2: key_buffer port map (inner_key, key_down, reset, inner_time);

    ----------- alarm controller ------------
    u3: alarm_controller port map (key_down,
                              alarm_button,
                              time_button,
                              clk,
                              reset,
                              inner_l_a,
                              inner_l_c,
                              inner_s_n,
                              inner_s_a
                              );

    ---------------- counter ------------------
    u4: alarm_counter port map(
                         inner_time,
                         inner_l_c,
                         inner_sec_clk,
                         reset,
                         inner_time_c
                         );
```

```
    --------- alarm register ----------------
u5: alarm_reg port map (inner_time, inner_l_a, clk, reset, inner_time_a);

  ------------- display divider ---------------
u6: display driver port map (
                    inner_time_a,
                    inner_time_c,
                    inner time,
                    inner_s_n,
                    inner_s_a,
                    sound_alarm,
                    display
                    );

 -------------- frequency divider -------------
u7: fq divider port map (clk, reset, inner_sec_clk);

end rtl;
```

（源描述文件名：74_alarm_clock.vhd
74_tb_alarm_clock.vhd）

第75例 存 储 器

李 春

1. 电路系统工作原理

本系统是一个存储器，其电路系统示意图如图 75.1 所示，其中使用的 RAM 的地址信号为 15 位输入端口，数据为 8 位输入输出端口，片选信号（$\overline{CS}$）、使能信号（$\overline{OE}$），写信号（$\overline{WE}$）为 1 位的 3 控制端口。在电路的 VHDL 描述中将加入相应的时序控制。

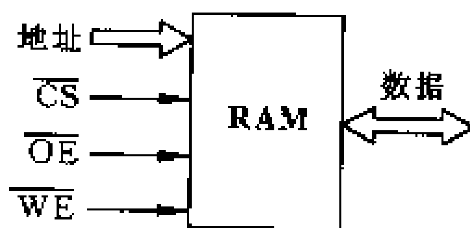


图 75.1 电路系统示意图

2. 存储器的设计思路

下面将设计一个带有时序特性的存储器模型。本例中总假定使能信号为 0。RAM 的时序模型是使用类属进行定义的。其中几个主要的时间参数已给定了缺省值。传输时延是用于避免撤消问题的，这种问题可能会与内部时延同时产生。RAM 进程的敏感信号有片选信号、写信号和地址信号，当写信号的上升沿到来时，若片选信号为 0，或者当片选信号的上升沿到来时写信号为 0，这就表明 RAM 的写操作结束，数据被写入 RAM。在 tow 时延后数据将被读出。若写信号的下降沿到来时片选信号为 0，则 RAM 转换为写模式，数据线被置为高。

若片选信号的上升沿到来时 RAM 未被选中，则数据线的输出将保持一段时间的高后再变为低，而当片选信号的下降沿到来时写信号为 1，则 RAM 为读模式。数据将在时刻 tclz 从高变为低，然而直到时刻 tacs 时才能确保有稳定的数据输出。从 RAM 读出到总线上有稳定的数据出现之间的时间区域称为过渡区，在这个过渡区内总线的状态是不确定的，所以在建模时，这段时间内 I/O 被设定为‘X’。

check 进程是与 RAM 进程同步执行的。本进程的目的是检查存储器的时序设定是否满足要求。NOW 是一个预先定义的变量，用来指示当前的时刻。为了避免无意义的错误消息，check 在 NOW=0 时或 RAM 未被选中时是不执行的。当地址信号改变时，check 进程将检查此信号在写周期 twc 中是否稳定，若信号不稳定，check 将输出警告信息。应注意的是，地址事件发生且 check 执行完成后，地址信号的 stable 属性将一直为 FALSE，

所以在 VHDL 的描述中必须用 address'delay 代替 address，这样，地址信号将延迟一个 delta，check 进程可以在地址改变完成前执行完毕。

3. VHDL 描述方法及语法分析

存储器的源描述如下：

```
library ieee;
use ieee.std_logic_1164.all;
use work.bit_pack.all;
entity static_ram is
          --下面给出的是 RAM6116 的时间特性值
          --在本系统的组装中将被置为 43258A-25CMOS RAM
          --的时间特性值
         generic (
          constant taa    : time := 120 ns;
          constant tacs   : time:= 120 ns;
          constant tclz   : time:= 10 ns;
          constant tchz   : time:= 10 ns;
          constant toh    : time:= 10 ns;
          constant twc    : time:= 120 ns;
          constant taw    : time:= 105 ns;
          constant twp    : time:= 70 ns;
          constant twhz   : time:= 35 ns;
          constant tdw    : time:= 35 ns;
          constant tdh    : time:= 0 ns;
          constant tow    : time:= 10 ns
        );
port (cs_b, we_b, oe_b :in bit;
      address :in bit_vector(7 downto 0);
      data :inout std_logic_vector(7 downto 0) := (others =>'Z')
      );
end static_ram;

architecture sram of static_ram is
type ramtype is array(0 to 255) of bit_vector(7 downto 0);
signal ram1:ramtype;-- := (others =>( others =>0));
signal flag:bit:= '1' ;
begin
 ram :process
 begin
   --使用 flag 信号值对 ram1 清零，且只进行一次
```

```
  if flag='1'  then
   for i in 255 downto 0 loop
     raml(i)<="00000000";
   end loop;
   flag<='0' ;
  end if ;
  if (we_b'event and we_b='1'  and cs_b'delayed ='0' ) or
   (cs_b'event and cs_b='1'  and we_b'delayed ='0' ) then
   --写操作
   raml(vec2int(address'delayed)) <=to_bitvector(data'delayed);
   --写操作完成后将数据读出
   --data'delayed 的值是 data 变为高前的值
   data <=transport data'delayed after tow;
  end if;
  --置 RAM 为写模式
  if (we_b'event and we_b='0'  and cs_b='0' ) then
    data <=transport "ZZZZZZZZ" after twhz;
  end if;
  if cs_b'event and oe_b ='0'  then
     --RAM 未选中
     if cs_b ='1'  then
       data <=transport "ZZZZZZZZ" after tchz;
     --读操作
     elsif we_b ='1'  then
      data <=transport "XXXXXXXX" after tclz;
      data <=transport to_stdlogicvector(raml(vec2int(address))) after
             tacs;
     end if;
  end if;
  --读操作
  if address'event and cs_b ='0'  and oe_b='0'  and we_b='1'  then
    data <="XXXXXXXX" after toh;
    data <=transport to_stdlogicvector(raml(vec2int(address))) after
           taa;
  end if;
  wait on cs_b,we_b,address;
end process ram;

check :process
begin
 if cs_b'delayed='0'  and now/=0 ns then
    if address'event then
```

```
      --假定 trc=twc
      assert (address' delayed' stable(twc))
      report "address cycle time too short"
      severity warning;
    end if;
    if (we_b' event and we_b=' 1' ) then
      assert (address' delayed' stable(taw))
      report "address not valid long enough to end of write"
      severity warning;
      assert (address' delayed' stable(twp))
      report "write pluse too short"
      severity warning;
      assert (data' delayed' stable(tdw))
      report "data setup time too short"
      severity warning;
      assert (data' last_event>=tdh)
      report "address cycle time too short"
      severity warning;
     end if;
  end if;
  wait on we_b, address, cs_b;
 end process check;
end sram;
```

（源描述文件名：75_ram.vhd）

第 76 例　电机转速控制器

张俭锋

1. 电路系统工作原理

本例中的 PID(proportional integral derivation，比例积分微分)控制器是一个电机转速控制系统的一部分，它对转速进行采样，与额定的转速进行比较，并通过微积分计算得到电机的控制电流，从而实现对电机转速的调整。通常，PID 控制器用模拟电路实现，但数字电路的实现方案可以很方便地实现集成。

该 PID 控制器包括一个完成算术和逻辑功能的 ALU 以及一个存储状态变量和有关系数的存储器。其端口说明如图 76.1 所示。

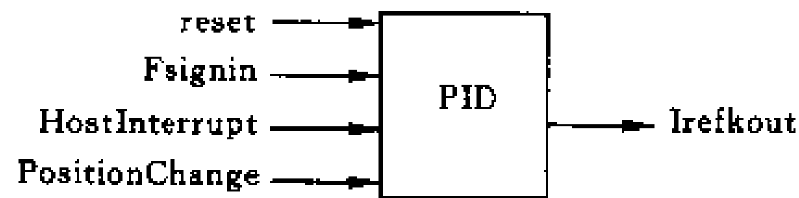


图 76.1　PID 控制器示意图

其中 reset 端口为复位端口，当 reset 为高电平时，PID 控制器复位；Fsign 为输入的偏差方向，若转速超过额定转速时为‘1’，否则为‘0’；HostInterrupt 为主机请求信号，当此信号为‘1’时，PID 控制器开始计算，为‘0’时，则停止计算并输出结果。PositionChange 为电机向 PID 控制器发送的脉冲信号，每转过一周，PositionChange 出现一个脉冲；Irefkout 为最终输出的控制电流值。

主机向 PID 控制器发调用请求 HostInterrupt 之后，PID 控制器进行采样并开始计算，当 HostInterrupt 信号停止时，PID 控制器停止采样计算，并将计算结果输出到主机。

PID 控制器的电流计算公式为

$$Irefk = (Kp+Ek) + Ki\times\int Ekdt + Kd\times dEk/dt$$

采用近似的离散方式表示为

$$Irefk = (Kp+Ek) + Ki\times\sum Ekdt + Kd\times dEk/dt$$

PID 控制器运行时，当电机每转过一周，即 PositionChange 信号发生一次变化时进行采样，得到电机转过一周所需要的时间，并将其作为积分的 dt 值，将该 dt 值取倒数计算出转速，然后与标准转速进行比较，得到转速偏差 Ek，将 Ek 与前一周得

到的 Ek-1 相减得出 dEk。将每一周的 Ek/dt 值相加得到积分 $\int$Ek/dt 的值。当 HostInterrupt 信号出现下降沿时将积分与当前时刻的 (Kp+Ek) 以及 Kd×dEk/dt 线性相加得到控制电流值。然后在下一个时钟周期将该控制电流值输出到主机。

2. VHDL 描述方法及有关语法分析

对应于图 76.1 的实体说明如下：

```
entity pid is
port (    reset             : in bit;
          Fsignin           : in bit;
          HostInterrupt     : in bit;
          PositionChange    : in bit;
          Irefkout          : out real);
end pid;
```

由于本例的端口中并没有时钟信号输入，所以需要用并发赋值语句来实现独立的时钟。如 clock <= **not** clock **after** 50 ns;。

本例中的描述采用模块化设计，将多次用到的数据类型、函数放在包中，这样就能够很容易地实现设计的重用。而且对于一些基本操作，如读数据、进行计算等，都用过程和函数来实现，体现了设计的层次性。计算中用到的常量和主机传递过来的参数存储在存储器中，在描述中用一个浮点数组 val_rom 来表示。

此外，浮点运算用独立的浮点运算单元来实现；由于在计算中要用到求倒数、浮点乘法等一系列的浮点运行功能，所以描述中说明一个小规模的浮点运算部件，包括浮点加减法、浮点乘法、求倒数等功能。相应的描述如下：

```
entity fixedpointunit is
port (  clock     : in bit;      --同步时钟
        reset     : in bit;      --复位信号
        input1    : in real;     --操作数 1
        input2    : in real;     --操作数 2
        sel       : in bit;      --启动信号
        com       : in int3bit;  --操作类型
        output    : out real;    --结果输出
        outdone   : out bit);    --完成信号
end fixedpointunit;

architecture behavior of fixedpointunit is
begin
  process
    variable tmp: real;
```

```
variable in1, in2: real;
procedure mul(A,B : in real) is
begin
 tmp := A * B;       --此处应该为具体的求乘积算法，
                     --为了简便起见用 A*B 代替
end mul;
procedure rep(A: in real) is
begin
  tmp := 1.0/A;      --此处应该为具体的求倒数算法，
                     --为了简便起见用 1.0/A 代替
end rep;
begin
 wait until sel = '1';                          --等待调用
   case com is                                  --判断操作类型
   when 1 => outdone <= '0';                    --浮点乘法
          in1 := input1;
          wait until rising_edge(clock);
          rep(in1);
          wait until rising_edge(clock);
          output <= tmp;
          outdone <= '1';
   when 2 => outdone <= '0';                    --求倒数
          in1 := input1;
          in2 := input2;
          wait until rising_edge(clock);
          mul(in1, in2);
          wait until rising_edge(clock);
          output <= tmp;
          outdone <= '1';
   when 3 => output <= input1 + input2;         --浮点加法
          wait until rising_edge(clock);
          outdone <= '1';
   when 4 => output <= input1- input2;          --浮点减法
          wait until rising_edge(clock);
          outdone <= '1';
   when others => outdone <= '1';
  end case;
 end process;
end behavior;
```

方程中的积分用一个 **while** 循环来计算，通过将时间划分成若干个小的时间片，然后将每个时间片中的函数值相加来近似地求得积分值。如以下描述：

```
wait until (HostInterrupt = '0');           --等待主机调用信号
while (HostInterrupt = '0') loop            --在调用期间求积分
  wait until (PositionChange = '1');        --等待电机转子转过一周
  getN (N);                                 --时间采样
  rep (N);                                  --取倒数求频率
  Ek_1 := Ek;
  waitresult (Fk, done);
  while (done /= '1') loop waitresult(Fk, done);
  end loop;
  if (Fsignin = '0')
  then Ek := Fref - Fk;
  else Ek := Fref + Fk;        --求转速偏差
  end if;
  mul (Kp, Ek);
  Dek := Ek - Ek_1;
  waitresult (Irefk, done);
  while (done /= '1') loop waitresult(Irefk, done);
  end loop;
  mul (Dek, Fk);
  waitresult (Temp, done);
  while (done /= '1') loop waitresult(Temp, done);
  end loop;
  mul (Temp, Kd);
  waitresult (Temp, done);
  while (done /= '1') loop waitresult(Temp, done);
  end loop;
  Irefk := Irefk + Temp;
  mul (Ek, N);
  waitresult(Temp, done);
  while (done /= '1') loop waitresult(Temp, done);
  end loop;
  Ik := Ik + Temp;
  mul (Ik, Ki);
  waitresult(Temp, done);
  while (done /= '1') loop waitresult(Temp ,done);
  end loop;
  Irefk := Irefk + Temp;
end loop;
```

这一段描述中与浮点单元之间的同步通过以下语句实现：

```
waitresult (Temp, done);
while (done /= '1') loop waitresult(Temp ,done);
```

```
end loop;
```

通过不断查询信号 done 的值是否为 '1' 来判断操作是否已经完成，如果未完成则继续等待，直到操作结束。

描述中用到的两个重要语法现象是 **case** 语句和 **while** 循环语句，其中 **case** 语句用来判断操作类型进行相应的计算。语句中 **when** 的选择必须覆盖 **case** 表达式的所有取值，如果不能覆盖，则必须加上 **when others** 语句表示其他选项时的情况。

while 循环语句在本例中用来计算积分，其语法为：

```
while 条件 loop
    语句序列
end loop;
```

在执行过程中，如果条件为真将一直执行循环中的语句序列直到条件为假。

3. 测试台

为对设计进行模拟，描述一个没有任何输入的测试台，并提供如下激励：

```
sig_Fin<= '0' ;
sig_HI <= '1' ;
sig_PC <= '0' ;
wait for 100 ns;
sig_HI<= '0' ;
wait for 100 ns;
sig_PC <= '1' ;

wait for 3500 ns;
sig_Fin<= '0' ;
sig_HI <= '0' ;
sig_PC <= '0' ;
wait for 50 ns;
sig_HI <= '0' ;
wait for 50 ns;
sig_PC <= '1' ;
sig_HI <= '1' ;

wait for 3500 ns;
sig_Fin <= '1' ;
sig_HI  <= '0' ;
sig_PC  <= '0' ;
wait for 100 ns;
```

```
sig_HI  <= ' 0' ;
wait for 100 ns;
sig_PC <= ' 1' ;

wait for 3500 ns;
sig_Fin <= ' 1' ;
sig_PC<= ' 0' ;
wait for 100 ns;
sig_HI<= ' 0' ;
wait for 100 ns;
sig_PC <= ' 1' ;
sig_HI <= ' 1' ;
```

（源描述文件名：76_pid.vhd
测试平台文件名：76_pic_stim.vhd
76_fpu.vhd）

第77例　神经元计算机

袁　媛

1. 神经元计算机简介

神经元芯片（neuron process system，简称NPS）主要由两部分组成：运算单元和控制器。控制器是神经元芯片的核心，它包括指令寄存器、指令译码电路、程序计数器及定时与各种控制信号的产生电路，它把用户程序中的指令逐条译出来，然后以一定时序发出相应的控制信号。

NPS 是支持不同的神经网络模型的通用芯片，所以其指令系统的完备性较好。NPS 有24条指令，分为4组：

访存指令，如 load，store 指令；

运算指令，如实现有符号数及无符号数的 add，sub，mul 等指令；

分支转移指令，如条件满足转移、条件不满足转移及无条件转移等指令；

传输指令，包括立即数传输、寄存器间的传输及输入、输出等指令。

NPS 的指令有立即数型、寄存器型、直接寻址与间接寻址访存等 4 种类型，其中有一字指令和两字指令之分。

2. NPS 的行为级描述

此例面向神经元计算机的指令系统和体系结构，详细描述了 NPS 的行为级描述。这里介绍 NPS 行为模型的建立。NPS 的接口描述保持在硬件级，用位或位串来表示外部控制信号、内存及数据总线等。

(1) 包

例子中定义了 3 个不同意义的包。basic_types 包定义了表示四值逻辑的枚举类型 qit 以及由这种基本类型组成的一维、二维数组，还有发生在这些自定义类型上的基本运算及它们与整型之间的换算等子程序的定义。

NPS_operators 包定义了一些子类型及 NPS 的指令系统所要求的加、减、乘等运算函数。

NPS_parameters 包用于在描述指令译码过程时增加程序的可读性。因为 NPS 的指令格式是用最高的 4 位表示指令操作码，例如，操作码为“0000”时，表示执行有符号数的加法指令：

```
add rd rs  (即 rd = rd + rs);
```

这时，如果用常量 add2cv 代替“0000”，那么将使读者对行为描述中每条指令的译码一目了然。对于指令系统中的各条指令，就用诸如此类的方法将它们区别开来。

(2) NPS 的接口描述

在 NPS 行为描述的接口部分，按照芯片的外部管脚说明了它的输入、输出端口。端口信号的类型除了 qit 和 qit_vector 以外，还有它们的分辨子类型。因为双向数据总线可能有多个驱动源。例如，当从内存读取数据时，内存用读出的数据来驱动数据总线，而 CPU 则用高阻信号驱动总线；当向内存写入数据时，CPU 与内存的驱动信号恰恰相反。这种数据总线的说明类型还允许在 CPU 上挂接多个外部设备。

(3) NPS 行为描述中的结构体

在 NPS 的高层次描述中，是从实现指令系统的角度来建立模型的。这种行为模型与较低层次的模型在功能上相同，但由于在这个设计阶段，芯片的内部硬件结构是未知的，所以就不可能描述出关于实际硬件电路的时序关系的模型。

下面是结构体的轮廓：

```
architecture behavioral of nps is
    寄存器及信号的说明;
begin
  process
    --些变量的说明;
  begin
    wait until clk='1' and clk'LAST_VALUE='0';
    if (reset='1') then
        处理系统复位信号;
        wait until clk='1' and clk'LAST_VALUE='0';
    else
        读取第一个指令字节到 IR 中，并使 PC 增 1;
        if (ir(15 downto 12) = 单字节指令)
        then      执行单字节指令;
        else
          wait until clk='1' and clk'LAST_VALUE='0';
          读取第二个指令字节到 BUFF 中，并使 PC 增 1;
          if ir(15 downto 12) = jn_jnl then
                执行有条件跳转指令
          elsif ir(15 downto 12) = jp then
                执行无条件跳转指令
            elsif ir(15 downto 12) = load then
```

```
                执行读取内存数据到内部寄存器
            elsif ir(15 downto 12) = store then
                执行写数据到内存中
                elsif  执行立即数传输指令
        end if;
      end if;
    end process;
end behavioral;
```

结构体包括一个进程。在进程的说明部分，包括一些临时变量和寄存器的说明。在进程的语句部分，首先检查 NPS 的 reset 引线上是否有初始化的请求，若有效，则将程序计数器和程序状态字寄存器清零，然后等待下一时钟周期的到来。下面的程序段说明了读取并存储一个字节的过程。当将读内存信号 read_mem 置为 '1' 后，要等待一段时间以使内存单元有足够的时间相应一个读请求。然后就可以从数据线上取出指令存入指令寄存器 IR 中，之后为了阻止下次读操作的读信号将这个清零信号覆盖，要在清除读信号的信号赋值语句后面加入一条等待语句。最后，使指令计数器指向下一个字节。对于双字节指令来说，还要继续读取下一个字节，并存入寄存器 buffer 中。

```
adbus      <= pc;
ale        <= '1';
read_mem   <= '1';
wait until clk='1' and clk'LAST_VALUE='0';
ir         <= byte(databus);
buff       <= byte(databus);
ale        <= '0';
read_mem   <= '0';
```

取指操作结束后，就要根据指令的操作码执行不同的指令。首先用 **if** 语句区别单字节指令与双字节指令，之后再用 **case** 语句区别各条不同的指令。

单字节指令有 18 条：7 条有符号数运算指令 add_cv，adc_cv，sub_cv，sbb_cv，inc_cv，dec_cv，mul_cv；4 条无符号数运算指令 add_c，adc_c，sub_c 及 sbb_c；另外，还有访存指令中的两条变址寻址方式指令及传输指令中的寄存器间数据传输和端口输入、输出指令以及空操作 nop 及停止指令 stop。

对于双字节指令中的 3 条跳转指令来说，用 **if** 语句判断跳转条件是否成立，若成立就将指令计数器中的值改成指令中给出的内存地址。另外两条双字节的访存指令，都要按照指令中的地址再次读写内存。还有一条立即数传输指令，是直接将指令中的立即数传输到指定的寄存器内。

NPS 的完整的行为描述请参考光盘中的 77_NPS918_r.vhd 文件，其中有详细的注释。

3. NPS 的测试

在 NPS 的测试台上，首先用一个二维数组表示内存单元，并装入覆盖整个指令系统的指令集，之后给出时钟信号及芯片复位信号的波形，组装说明可以选择前面建立的行为模型或者 RTL 模型，测试台将对被组装的实体进行功能测试。本测试台的主要功能就是模拟内存与 CPU 之间的通信，这由一个内存进程来实现。当接受到一个“读”请求时，它按照地址线上的地址，取出内存数据打入数据总线，并保持到“读”信号为“0”时，再将数据线置为高阻状态，表示内存失去对总线的控制权。当“写”信号活动时，将总线上的数据按访存地址写入，直到“写”信号无效时，再进入下一次访存操作。

第78例　Am2901 四位微处理器的ALU输入

韩　曙

1. Am2901 总论及 ALU 输入逻辑工作原理

Am2901 是 AMD 公司生产的 4 位微处理器逻辑芯片，具有 9 位指令码输入端，最多可以实现 512 种不同的操作（有冗余），片内有一个 16×4 位的 RAM，其示意图如图 78.1 所示。

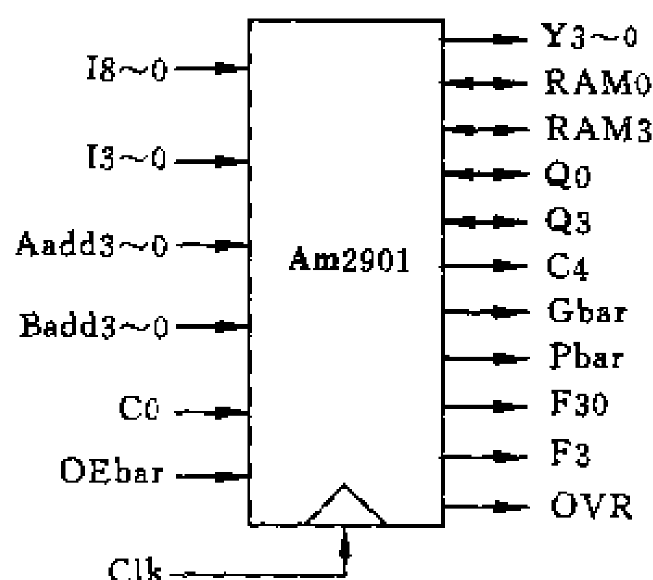


图 78.1　Am2901

其中，I8~0 为指令码输入端，D3~0 为数据输入端，Aadd3~0，Badd3~0 分别为存储器地址输入端，C0 为 ALU 进行算术运算时的低位进位输入端，OEbar 为输出允许控制信号；Y3~0 为数据输出端，RAM0，RAM3 为存储器的移位输入／输出端，Q0，Q3 为寄存器移位输入／输出端，C4 为 ALU 进行算术运算时高位进位输出端，Gbar，Pbar 分别为进位生成（Carry Generate）和进位传递（Carry Propagate）输出信号，其作用是在芯片级连时形成超前进位链（有关这方面的介绍详见第 89 例）；F30 为零标志位，F3 为符号标志位，OVR 为溢出标志位，Clk 为时钟信号端。Am2901 的结构框图如图 78.2 所示。

本系统逻辑功能复杂，涉及的操作较多，若作为一个整体进行描述，势必增大难度；而且由于输入输出信号很多，在进行模拟验证时，测试向量集必然很大，不利于对描述正确性的分析。所以对这类较复杂的数字系统，有效的方法是首先对系统进行合理划分，然后对划分后的各子系统进行分别描述与验证。

本系统按逻辑功能的不同划分成 5 个相对独立的部分：①ALU 输入选择；②ALU 运算；③存储器；④寄存器；⑤输出及移位．它们在指令码 I8~I0 的控制下分别完成相应的逻辑功能。我们将在第 78 例~第 82 例中对各部分作系统介绍，并分析它们用 VHDL 进行描

述时的特点。

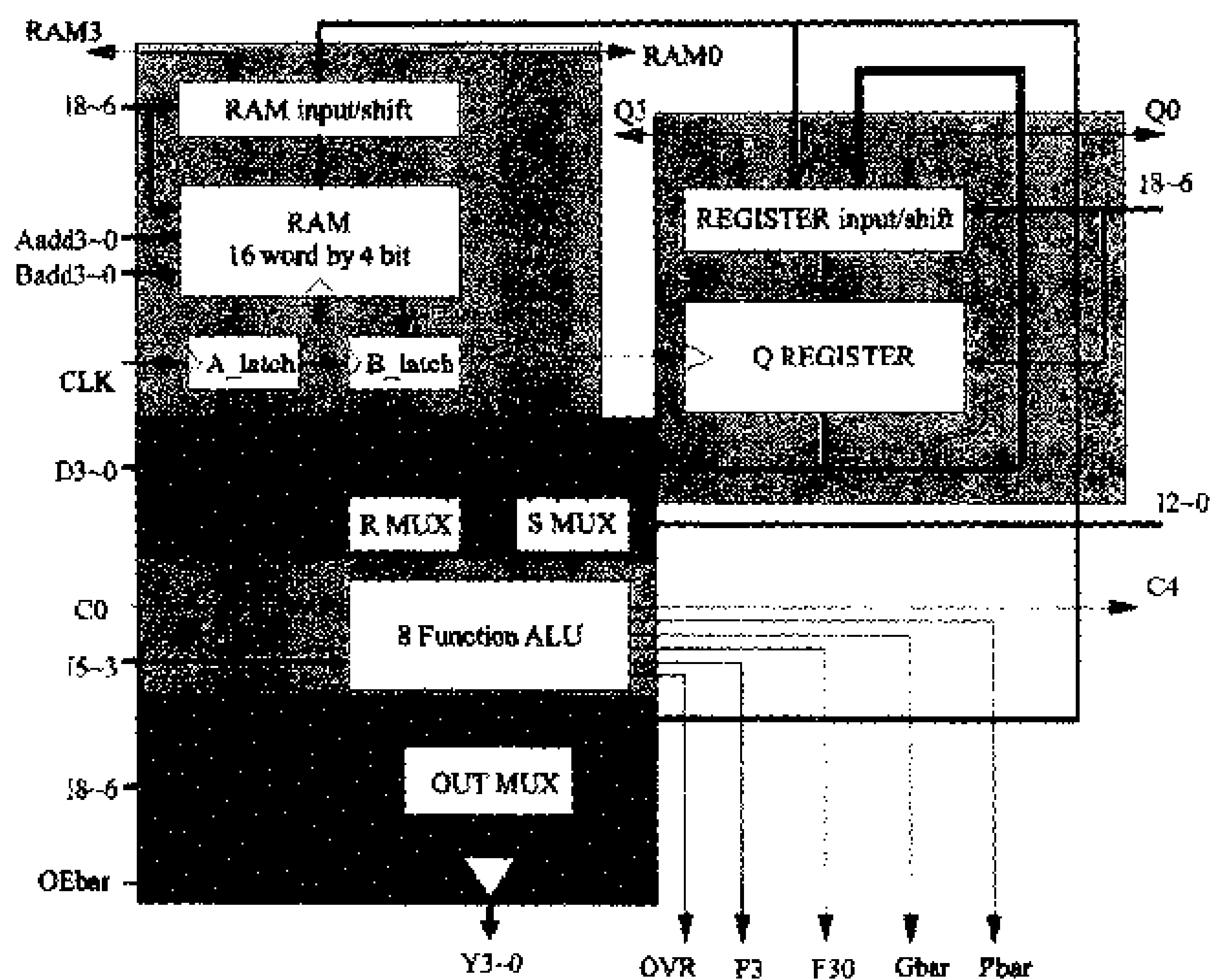


图 78.2 Am2901 结构框图

本例首先讨论 ALU 输入逻辑。

ALU 的两个操作数通过多路选择器 R 和 S 进行选择，其中 R 选择 A、'0000' 和 D3~0 输入端，S 用来选择 Q，A，B 或'0000' 数据端。锁存器 A 和 B 接收 RAM 中的数据，RAM 由 Aadd 和 Badd 进行寻址。选择控制信号为指令码的 I2~I0 位，选择方式见表 78.1。

表 78.1 ALU 输入选择

指令码			输入选择	
I2	I1	I0	R	S
0	0	0	A	Q
0	0	1	A	B
0	1	0	0	Q
0	1	1	0	B
1	0	0	0	A
1	0	1	D	A
1	1	0	D	Q
1	1	1	D	0

根据表中给出的逻辑关系可写出选择端 R 和 S 的逻辑表达式

R = AI2′ I1′ + D(I2 I1 + I2 I0);

S = Q(I2′ I0′ + I1 I0′)+ A(I2 I2′) + B(I0′ I0);

其中 Ix′ 表示该变量的逻辑非。

上述表达式在 VHDL 描述中由 **with...select...when** 语句实现，这种描述方式与真值表具有很明确的对应关系，当然也可以用布尔表达式的形式直接写出。

2. 电路的 VHDL 语言描述及语法分析

本例中的操作涉及 RAM 的存取、锁存和多路选择，在 VHDL 语言描述中，RAM 定义为一个 16×4 的 2 维数组信号，数据类型为 MVL7。实体定义中输入端包括 Aadd，Badd（整型），I（多路选择端，3 位 MVL7 型），输出包括 R 和 S（4 位 MVL7 型）。

首先，在所有的结构描述中，都使用了一个独立的库函数 L2901_lib，在这个库中，定义了一些常用的，基于 MVL_7 数据类型的运算，包括逻辑移位、求反、求补、奇偶校验、算术运算、线或等，详见第 9 例和第 11 例中的描述。

在进行 AM2901 各功能块描述的编译和模拟之前，应首先对这个库函数和包进行编译。

ALU 输入逻辑的 VHDL 语言描述为：

```
library L2901_LIB;                          -- 库函数
use L2901_LIB.types.all;                    -- 数据类型
use L2901_LIB.MVL7_functions.all;           -- 函数定义
use L2901_LIB.SYNTHESIS_TYPE. all;

entity alu_inputs is
port (
     Aadd, Badd      : in integer;                     -- RAM 地址输入
     Q, D            : in MVL7_vector(3 downto 0);     -- 寄存器及外部数据
     I               : in MVL7_vector(2 downto 0);     --指令输入（第 0~2 位）
     R, S            : out MVL7_vector(3 downto 0)     -- 寄存器输出
     );
end alu_inputs;

architecture alu_inputs of alu_inputs is
-- 存储器数据类型
     type Memory is array (integer range <>) of MVL7_vector(3 downto 0);
     signal RAM :                                              -- 存储器两维数组
     Memory (15 downto 0) := (("0000"),("0000"),("0000"),("0000"),
                              ("0000"),("0000"),("0000"),("0000"),
                              ("0000"),("0000"),("0000"),("0000"),
```

```
                    ("0000"),("0000"),("0100"),("1000"));--赋初值
    signal A, B : MVL7_vector(3 downto 0);                  -- 中间信号
    begin
        A <= RAM(Aadd);                                      --读 RAM
        B <= RAM(Badd);                                      -- 操作数选择
        with I(2 downto 0) select                            -- 见表 68.1
        R <= A      when    "000" | "001",
            "0000"  when    "010" | "011" | "100",
             D      when    others;
        with I(2 downto 0) select
        S <= A      when    "100" | "101",
            B       when    "001" | "011",
            "0000"  when    "111",
            Q       when    others;
end alu_inputs;
```

上述描述中，有关存储器的操作将在第 80 例中讨论。

3. 模拟测试向量的选择及模拟结果分析

模拟的目的是验证对实体的功能描述是否正确，即验证 VHDL 语言描述和实际输入输出之间的逻辑关系是否一致。由于本例中用了 3 位指令码，测试向量按指令码分类共有 8（2^3）种，测试台的 VHDL 描述如下：

```
library L2901_LIB;                          --库函数
use L2901_LIB.types.all;                    --数据类型
use L2901_LIB.MVL7_functions.all;           --函数定义

entity ALU_INPUTS_E is                      --定义测试台（空实体）
end;

architecture ALU_INPUTS_A of ALU_INPUTS_E is
component alu_inputs_inst                   — 元件例示
port (                                      -- 端口定义
      Aadd, Badd   : in  integer;
      Q, D         : in  MVL7_vector(3 downto 0);
      I            : in  MVL7_vector(2 downto 0);
      R, S         : out MVL7_vector(3 downto 0)
    );
end component;
-- 信号定义
signal Aadd, Badd  : integer := 0;
```

```
signal Q, D          : MVL7_vector(3 downto 0);
signal I             : MVL7_vector(2 downto 0);
signal RE, S         : MVL7_vector(3 downto 0);
for all : alu_inputs_inst use entity work.alu_inputs(alu_inputs);

begin
    Alu_inputs_inst1 : alu_inputs_inst          -- 元件说明
                    port map(
                              Aadd, Badd,
                              Q, D,
                              I,
                              RE, S
                         );
process
begin
-- 测试向量#1:              I2I1I0="000"        , R<=A , S <=Q
    Aadd <= 0;                                  -- 地址选择
    Badd <= 1;
    D <= "0001";
    Q <= "0010";
    I  <= "000";
    wait for 1 ns;
    assert (R = "1000")                         -- 判断赋值是否正确
        report  "assert 01: < R /= ' 1000' > "  severity warning;
    assert (S = "0010")
        report  "assert 02: < S /= ' 0010' > "  severity warning;
    wait for 1 ns;
-- 测试向量#2:              I2I1I0="001"
        , R<=A , S <=B
    Aadd <= 0;                                  -- 地址选择
    Badd <= 1;
    D <= "0001";
    Q <= "0010";
    I <= "001";
    wait for 1 ns;
    assert (R = "1000")
        report  "assert 11: < R /= ' 1000' > "      severity warning;
    assert (S = "0100")
        report  "assert 12: < S /= ' 0100' > "      severity warning;
    wait for 1 ns;
-- 测试向量#3:              I2I1I0="010"        , R<=0 , S <=Q
    Aadd <= 0;        -- 地址选择
```

```
    Badd <= 1;
    D <= "0001";
    Q <= "0010";
    I <= "010";
    wait for 1 ns;
    assert (R = "0000")
        report   "assert 31: < R /= ' 0000' > "   severity warning;
    assert (S = "0010")
        report   "assert 32: < S /= ' 0010' > "   severity warning;
    wait for 1 ns;
-- 测试向量#8:              I2I1I0="111"       , R<=D , S <=0
    Aadd <= 0;                                  -- 地址选择
    Badd <= 1;
    D <= "0001";
    Q <= "0010";
    I <= "111";
    wait for 1 ns;
    assert (RE = "0001")
        report   "assert 81: < RE /= ' 0001' > "  severity warning;
    assert (S = "0000")
        report   "assert 82: < S /= ' 0000' > "      severity warning;
    wait for 1 ns;

-- 测试结束
    assert false
        report   "---End of Simulation---"       severity error;
end process;
end ALU_INPUTS_A;
```

（源描述文件名：78_alu_inputs.vhd
测试平台文件名：78_alu_inputs_stim.vhd）

第 79 例　Am2901 四位微处理器的 ALU

韩　曙

1. 电路系统的工作原理

ALU 的功能是执行算术和逻辑运算，参与运算的操作数存放在寄存器 R 和 S 中，在指令码 I5~I3 的控制下，可以实现 8 种不同的运算功能，运算结果由 F 输出。如表 79.1 所示。

除上述输入输出以外，ALU 还接受一位进位链输入信号 C0；运算标志位有：进位输出信号 C4，溢出标志位 OVR，符号标志位 F3，零标志位 F30，进位生成 Gbar 和进位传递 Pbar。其中 C4、Pbar 和 Gbar 用于芯片级连，从而可扩大操作数的表示范围。

表 79.1　ALU 操作

指令代码			ALU 操作
I5	I4	I3	
0	0	0	R+S
0	0	1	S–R
0	1	0	R–S
0	1	1	R∨S
1	0	0	R∧S
1	0	1	R’∧S
1	1	0	R⊕S
1	1	1	R⊙S

2. 电路的 VHDL 语言描述方法及语法分析

因为参与运算的操作数实际有 3 个（R，S 为 4 位数，C0 为 1 位），输出若考虑进位 C4，则运算结果可能为 5 位（F 为 4 位）。为了描述方便，这里定义一些辅助信号。其中 R_ext 和 S_ext 分别将 2 个输入信号连同低位的进位信号合并成 5 位信号作为输入，Result 为 5 位输出辅助信号，由它可获得输出 F，C4 以及其他的标志位。Temp_p 和 Temp_g 为进位生成和进位传递的辅助信号，其用法详见结构描述的注释。

和第 78 例一样，本例描述的结构是一个组合逻辑系统，因此在结构体中，各语句的

执行是并发的。另外需要注意的是在整个系统中，控制 ALU 操作的指令码是 I5~3，在描述中为方便起见仍定义为 I2~0。在后续的几个例子中也这样处理。

```
—  库函数引用
library 12901_lib;
use 12901_lib.TYPES.all;
use 12901_lib.MVL7_functions. all;
--  实体定义
entity alu is
   port (
         R , S                          :   in MVL7_vector(3 downto 0);
         I                              :   in MVL7_vector(2 downto 0);
         C0                             :   in MVL7;
         C4, OVR, F30, F3, Pbar, Gbar   :   out MVL7;
         F                              :   out MVL7_vector(3 downto 0)
        );
end alu;
-- 结构体
architecture alu of alu is
-- 中间信号
    signal R_ext,S_ext,result,temp_p,temp_g : MVL7_vector(4 downto 0);
begin
    --为了描述方便，将输入信号扩展成 5 位，第 5 位为扩展符号位。

R_ext <= ' 0'  & not(R) when I(2 downto 0) = "001" --求反，实现 S-R
                        else ' 0'  & R;
S_ext <= ' 0'  & not(S) when I(2 downto 0) = "010" --求反，实现 R-S
                        else ' 0'  & S;
-- ALU 功能选择
-- 加减运算采用补码，由于允许级连，在实际运算中，若是减法运算、
-- 在整个被减数求反后，对最低位的进位信号置 1 ，以实现求补。
with I(2 downto 0) select
Result <=R_ext + S_ext + ("0000" & C0) when  "000" | "001" | "010",
                                                                -- 补码加
             R_ext or S_ext                   when   "011",          -- 或
             R_ext and S_ext                  when   "100",          -- 与
             not(R_ext) and S_ext             when   "101",          -- 与
             R_ext xor S_ext                  when   "110",          -- 异或
             not( R_ext xor S_ext)            when   others;         -- 同或

--其他运算
    F        <= result(3 downto 0);       -- 运算结果取 result 的低 4 位数
```

```
OVR      <= not (R_ext(3) xor S_ext(3)) and (R_ext(3) xor result(3));
                                            -- 溢出标志位
C4       <= result(4);                      -- 进位
temp_p   <= R_ext or S_ext;                 -- 中间信号
temp_g   <= R_ext and S_ext;                -- 中间信号
Pbar     <= not(  temp_p(0) and temp_p(1) and temp_p(2) and temp_p(3));
                                            -- 进位传输逻辑
Gbar     <= not ( temp_g(3) or
              (temp_p(3) and temp_g(2)) or
              (temp_p(3) and temp_p(2) and temp_g(1)) or
              (temp_p(3) and temp_p(2) and temp_p(1) and temp_g(0))
            );                                   -- 进位生成逻辑
F3       <= result(3);                           -- 符号标志位
F30      <= not (result(3) or result(2) or result(1) or result(0));
                                            -- 零标志位
end alu;
```

3. 模拟测试向量的选择及模拟结果分析

本例的模拟验证较为复杂，一方面要验证基本的算术逻辑运算是否正确，另一方面要验证有关的标志位输出是否正确。所以用到的测试向量较多。测试向量的生成原则要考虑覆盖如下几个方面：

① 在不同指令码的控制下，ALU 执行的运算功能正确性；

② 在进行算术运算时，考虑加、减运算的不同、加（减）数和被加（减）数的数值范围对结果的影响及有关标志位是否正确；

③ 在进行逻辑运算时，仅需检测运算结果 F，其他标志位虽然也有输出，但无意义。

在 VHDL 语言描述中，需要注意的是有关的标志位只有在算术运算时才有效，对逻辑运算来说，这些信号是无意义的，在模拟时可以不予考虑。

下面对描述中的部分测试向量做一简单分析：

```
……
R    <= "0001";              -- #1
S    <= "0001";
C0   <= ' 0' ;               -- 算术运算 R + S. ( R = 0001, S = 0001)
I    <= "000";               -- 低位无进位，无高位进位，无溢出，非 0
wait for 1 ns;               -- 符号位为 0
assert (F = "0010")          -- 验证运算结果
    report "assert a1: < F /= ' 0010' > " severity warning;
assert (C4 = ' 0' )                   -- 验证进位
    report "assert a2 : < C4 /= ' 0' > " severity warning;
assert (OVR = ' 0' )                  -- 验证溢出标志
```

```
    report "assert a3 : < OVR /= '0'> " severity warning;
assert (F30 = ' 1' )                    -- 验证零标志
    report "assert a4 : < F30 /= ' 1' > " severity warning;
assert (F3 = ' 0' )                     -- 验证符号位
    report "assert a5 : < F3 /= ' 0' > " severity warning;
assert (Pbar = ' 1' )               -- 验证进位传输信号
    report "assert a6 : < Pbar /= ' 1' > " severity warning;
assert (Gbar = ' 1' )               -- 验证进位生成信号
    report "assert a7 : < Gbar /= ' 1' > " severity warning;
wait for 1 ns;
......
R <= "1000";                 --#4
S <= "1000";
C0 <= ' 0' ;                 -- 计算 R + S ( R = 1000, S = 1000)
I <= "000";                  -- 低位无进位，有高位进位，有溢出，
                             -- 零标志为 1，符号位为 0
wait for 1 ns;
assert (F = "0000")
    report "assert d1 : < F /= ' 0000' > "  severity warning;
assert (C4 = ' 1' )
    report "assert d2 : < C4 /= ' 1' > "  severity warning;
assert (OVR = ' 1' )
    report "assert d3 : < OVR /= ' 1' > "  severity warning;
assert (F30 = ' 1' )
    report "assert d4 : < F30 /= ' 1' > "  severity warning;
assert (F3 = ' 0' )
    report "assert d5 : < F3 /= ' 0' > "  severity warning;
assert (Pbar = ' 1' )
    report "assert d6 : < Pbar /= ' 1' > "  severity warning;
assert (Gbar = ' 0' )
    report "assert d7 : < Gbar /= ' 0' > "  severity warning;
wait for 1 ns;
......
R  <= "0001";                        --#5
S  <= "0010";
C0 <= ' 1' ;                         -- 减法运算 S - R ( R = 0001, S = 0010)
I  <= "001";                         -- 低位进位置 1 (实现补码)，有进位，
wait for 1 ns;                       -- 无溢出，零标志为 0，符号位为 0
assert (F = "0001")
    report "assert e1 : < F /= ' 0001' > " severity warning;
assert (C4 = ' 1' )
    report "assert e2 : < C4 /= ' 1' > " severity warning;
```

```
assert (OVR = ' 0' )
    report "assert e3 : < OVR /= ' 0' > " severity warning;
assert (F30 = ' 0' )
    report "assert e4 : < F30 /= ' 0' > " severity warning;
assert (F3 = ' 0' )
    report "assert e5 : < F3 /= ' 0' > " severity warning;
assert (Pbar = ' 1' )
    report "assert e6 : < Pbar /= ' 1' > " severity warning;
assert (Gbar = ' 0' )
    report "assert e7 : < Gbar /= ' 0' > " severity warning;
wait for 1 ns;
......
R <= "0001";            --#6
S <= "0010";
C0 <= ' 0' ;                -- 计算 S - R -1 ( R = 0001, S = 0010)
I <= "001";             -- 由于 C0=0, 等价于[S]补+[R]反=[S]补+[R]反+1-1=[S]补+[R]补-1
wait for 1 ns;
assert (F = "0000")
    report "assert f1 : < F /= ' 0000' > "  severity warning;
assert (C4 = ' 1' )
    report "assert f2 : < C4 /= ' 1' > "  severity warning;
assert (OVR - ' 0' )
    report "assert f3 : < OVR /= ' 0' > "  severity warning;
assert (F30 = ' 1' )
    report "assert f4 : < F30 /= ' 1' > "  severity warning;
assert (F3 = ' 0' )
    report "assert f5 : < F3 /= ' 0' > "  severity warning;
assert (Pbar = ' 1' )
    report "assert f6 : < Pbar /= ' 1' > "   severity warning;
assert (Gbar = ' 0' )
    report "assert f7 : < Gbar /= ' 0' > "  severity warning;
wait for 1 ns;
......
R  <= "1010";                   --#9
S  <= "1001";
C0 <= ' 0' ;                -- 逻辑运算 R or S. ( R = 1010, S = 1001)
I  <= "011";
wait for 1 ns;
assert (F = "1011")             -- 仅需验证逻辑运算结果
report "assert i2 : < F /= "1011"> " severity warning;
wait for 1 ns;
......
```

部分模拟结果的波形如图 79.1 所示。读者可以结合表 79.1 进行分析。为观察方便，我们对相关的信号做了打包处理，如 i5~3= i[5]i[4]i[3]等。

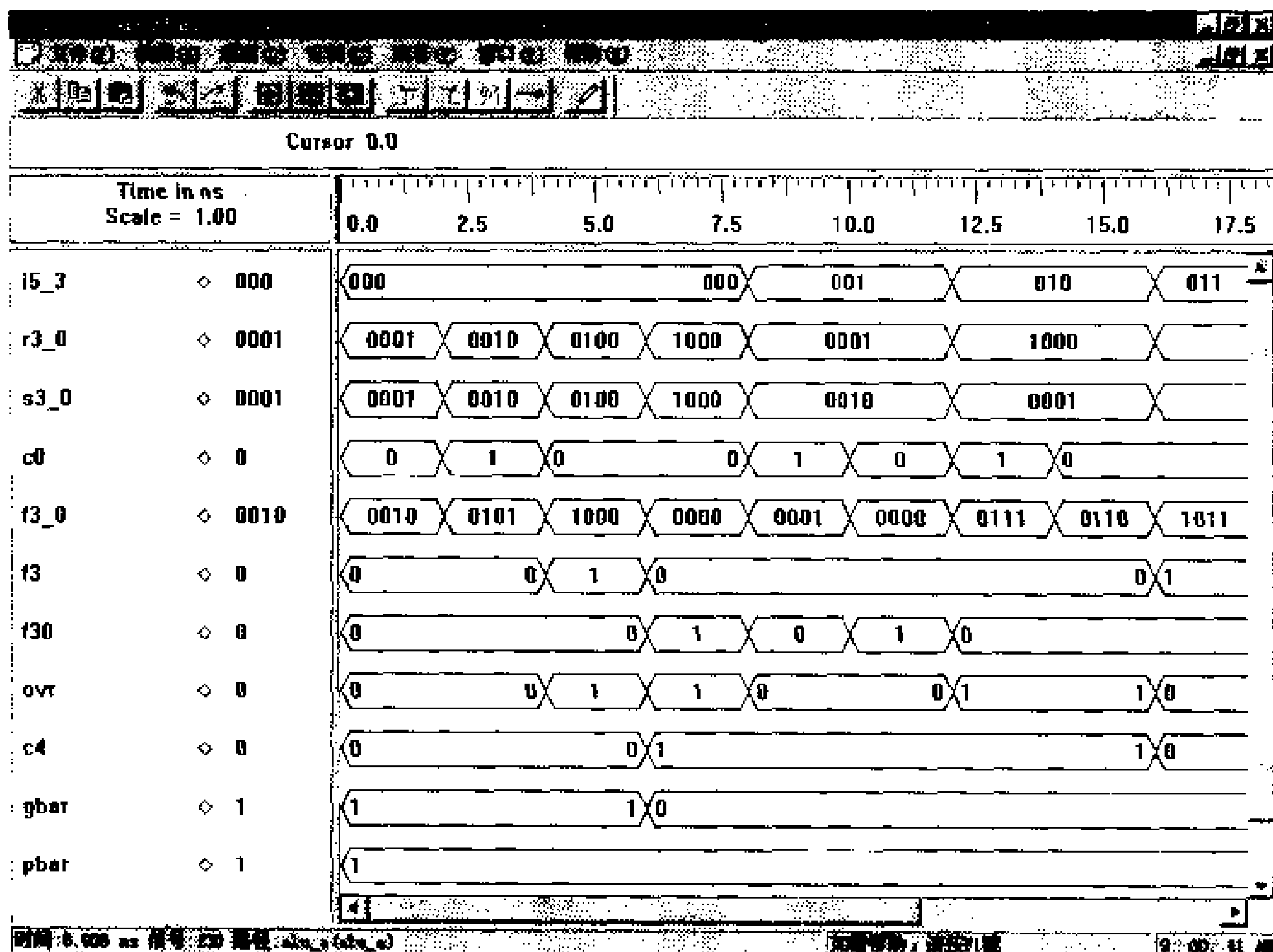


图 79.1 ALU 模拟波形图

（源描述文件名：79_alu.vhd

测试平台文件名：79_alu_stim.vhd）

第 80 例　Am2901 四位微处理器的 RAM

韩　曙

1. 电路系统的工作原理

存储器操作包括在给定的地址信号作用下，实现数据的输入和输出。在本模块中，数据的输入有 3 种不同的方式：①直接输入，将 F 送入由地址 Badd 确定的单元中；②左移输入，将 F 左移 1 位，F2 F1 F0 和 RAM0 作为 4 位操作数送入由 Badd 寻址的 RAM 单元；③右移输入，将 F 右移 1 位，RAM3 和 F3 F2 F1 作为 4 位操作数送入由 Badd 寻址的 RAM 单元；注意在 RAM 操作中，地址统一由 Badd 表示，而 Aadd 不参与寻址。操作方式由指令码 I8~I6 控制，如表 80.1 所示。时钟信号为 clk。

表 80.1　输入输出及移位操作

指令代码			RAM 操作
I8~6			
0	0	0	-
0	0	1	-
0	1	0	F3~0
0	1	1	F3~0
1	0	0	RAM3&F3~1
1	0	1	RAM3&F3~1
1	1	0	F2~0&RAM0
1	1	1	F2~0&RAM0

2. 电路的 VHDL 语言描述方法及语法分析

本模块从概念上讲是时序逻辑子系统，有一个时钟脉冲信号，在它的作用下，系统的“状态”（存储单元的值）发生变化。但它与一般意义上的时序电路又有所不同，它并不在有关状态和输入信号的作用下完成状态的转换和发出有关的控制信号。在 VHDL 描述中，也未采用通常时序逻辑使用的进程的描述方法，而是使用了保护块（**Block**）的描述方法。保护块含有一个保护表达式，本例中为(clk = ‘1’) **and** (**not** clk’ stable)，当此表达式的值为真时，执行块内语句，否则，不执行。即保护表达式的功能类似于进

程中的敏感信号。关键字 guarded 用来对保护信号赋值。关于保护表达式的讨论详见第 27 例。

```
library 12901_lib;
use 12901_lib.TYPES.all;
use 12901_lib.MVL7_functions.all;
use 12901_lib.synthesis_type.all;

entity mem is
port
        RAM        : inout Memory(15 downto 0);
        F          : in MVL7_vector(3 downto 0);
        Clk        : in clock;
        I          : in MVL7_vector(2 downto 0);
        RAM0, RAM3          : in MVL7;
        Aadd, Badd          : in integer range 15 downto 0
       );
end mem;
architecture mem of mem is
begin
    Mem1 : block ( (clk = '1' ) and (not clk' stable) )
                              -- 时钟信号作为保护表达式
    begin                     -- RAM由Badd寻址，首先定义保护的信号赋值方式
    RAM(Badd) <= guarded F when ((not (I(2)) and I(1)) = '1' )
    else  RAM3 & F(3 downto 1) when ((I(2) and not (I(1))) = '1' )   --右移
    else  F(2 downto 0) & RAM0 when ((I(2) and I(1)) = '1' )            --左移
    else  RAM(Badd);                                                  --保持
    end block mem1;
end mem;
```

3．模拟测试向量的选择及分析

测试向量的编写主要考虑覆盖上节描述中的 4 项主要功能，由于是时钟触发方式，在有关测试向量生成以后，时钟信号的描述要注意有一个跳变，然后再验证有关信号的逻辑关系。

```
clk <='0' ;
F <= "0111";          --#1
I <= "010";
RAM0 <= 'Z' ;
RAM3 <= 'Z' ;
Aadd <= 0;            -- RAM(Badd)<- F ,  Badd = 0
```

```
Badd <= 0;
wait for 1 ns;         -- 时钟跳变描述
clk <= ' 1' ;
wait for 1 ns;
clk <= ' 0' ;
wait for 1 ns;
assert (RAM(Aadd) = "0111")        -- 验证结果(由 Aadd 寻址)
    report "assert 1 : < RAM(Aadd) /= ' 0111' > " severity warning;
assert (RAM(Badd) = "0111")        -- 验证结果(由 Badd 寻址)
    report "assert 2 : < RAM(Badd) /= ' 0111' > " severity warning;
wait for 1 ns;
......
F <= "0000";           --#3
I <= "000";
RAM0 <= ' Z' ;
RAM3 <= ' Z' ;
Aadd <= 1;             --  保持
Badd <= 0;
wait for 1 ns;         -- 时钟跳变
clk <= ' 1' ;
wait for 1 ns;
clk <= ' 0' ;
wait for 1 ns;
assert (RAM(Aadd) = "1001")        -- 结果验证
    report "assert b1 : < RAM(Aadd) /= ' 1001' > " severity warning;
assert (RAM(Badd) = "0111")
    report "assert b2 : < RAM(Badd) /= ' 0111' > " severity warning;
wait for 1 ns;
......
F <= "1001";           --#5
I <= "100";                -- 右移
RAM0 <= ' Z' ;
RAM3 <= ' 0' ;
Aadd <= 1;
Badd <= 2;
wait for 1 ns;         -- 时钟跳变
clk <= ' 1' ;
wait for 1 ns;
clk <= ' 0' ;
wait for 1 ns;
assert (RAM(Aadd) = "1001")
    report "assert d1 : < RAM(Aadd) /= ' 1001' > " severity warning;
```

```
assert (RAM(Badd) = "0100")
    report "assert d2 : < RAM(Badd) /= ' 0100' > " severity warning;
wait for 1 ns;
......
F <= "0110";                --#9
I <= "110";                 -- 左移
RAM0 <= ' 0' ;
RAM3 <= ' Z' ;
Aadd <= 4;
Badd <= 2;
wait for 1 ns;
clk <= ' 1' ;
wait for 1 ns;
clk <= ' 0' ;
wait for 1 ns;
assert (RAM(Aadd) = "0110")
    report "assert h1 : < RAM(Aadd) /= ' 0110' > " severity warning;
assert (RAM(Badd) = "1100")
    report "assert h2 : < RAM(Badd) /= ' 1100' > " severity warning;
wait for 1 ns;
```

（源描述文件名：80_mem.vhd
测试平台文件名：80_mem_stim.vhd）

第 81 例　Am2901 四位微处理器的寄存器

韩　曙

1. 电路系统的工作原理

寄存器的操作与 RAM 的操作类似，也具有保持、接收、左移和右移 4 项功能，操作受 clk 的控制，上升沿触发。操作类型受指令码 I8~6 的控制。如表 81.1 所示。

在 VHDL 描述中，也采用与 RAM 的描述类似的方法。即将整个模块定义成一个块结构。只有在保护表达式的值为真时，执行块内语句。而不是采用通常描述时序电路时的进程结构。

表 81.1　输入输出及移位操作

指令代码			Q 操作
I8~6			
0	0	0	F3~0
0	0	1	—
0	1	0	—
0	1	1	—
1	0	0	Q3&Q（3~1）
1	0	1	—
1	1	0	Q（2~0）&Q0
1	1	1	—

2. 电路的 VHDL 语言描述方法及语法分析

由表 81.1 可知，本模块在指令码 I8~6 的控制下，分别实现不同的操作。和第 80 例类似，本例仍采用保护块的描述方法，保护表达式为(clk='1') **and** (**not** clk'STABLE)，即操作是在时钟信号的控制下进行的。

电路功能描述如下：

```
library l2901_lib;
use l2901_lib.TYPES.all;
use l2901_lib.MVL7_functions.all;
```

```
entity Q_reg is
    port (
          F        : in MVL7_vector(3 downto 0);
          clk      : in clock;
          I        : in MVL7_vector(2 downto 0);
          Q0, Q3   : in MVL7;
          Q        : inout MVL7_vector(3 downto 0)
         );
end Q_reg;
architecture Q_reg of Q_reg is
begin
    Q_reg1 : block ( (clk = ' 1' ) and (not clk' stable ) )
    begin                                          -- 保护的信号赋值方式
         Q <= guarded F when (I(2 downto 0) = "000")
              else Q3 & Q(3 downto 1) when (I(2 downto 0) = "100")
                                                   --右移输入
              else Q(2 downto 0) & Q0 when (I(2 downto 0) = "110")
                                                   --左移输入
              else Q;                              --  保持
    end block Q_reg1;
end Q_reg;
```

3. 模拟测试向量的选择及模拟结果分析

详细的模拟分析见第 7 例，在此就不讨论了。

（源描述文件名：81_Q_reg.vhd
测试平台文件名：81_Q_reg_stim.vhd）

第 82 例　Am2901 四位微处理器的输出与移位

韩　曙

1. 电路系统的工作原理

输出部分包括：

① ALU 处理后的数据和从存储器直接读出的数据。二者经多路选择器（输出 MUX）送输出总线 Y3~0，输出总线是三态结构，受控制端 OEbar 的控制。

② 移位输出端，包括 RAM0，RAM3，Q0 和 Q3。它们在存储器或寄存器左／右移位时作为移入／移出端使用，是具有三态输出的双向数据端。

输出部分的逻辑功能、存储器与寄存器的数据选择操作等见表 82.1。

表 82.1　输入输出及移位操作

指令代码			Y 输出	RAM 移位		Q 移位	
I8~6				RAM0	RAM3	Q0	Q3
0	0	0	F	Z	Z	Z	Z
0	0	1	F	Z	Z	Z	Z
0	1	0	A	Z	Z	Z	Z
0	1	1	F	Z	Z	Z	Z
1	0	0	F	F0	Z	Q（0）	Z
1	0	1	F	F0	Z	Q（0）	Z
1	1	0	F	Z	F3	Z	Q（3）
1	1	1	F	Z	F3	Z	Q（3）

2. 电路的 VHDL 语言描述方法及语法分析

结合第 80 例中的真值表可以看出，输出 F 与移位端 RAM3、RAM0 一起构成了 RAM 的输入信号。在直接输入情况下，两个移位信号不起作用，为高阻态。在移位输入时，作为输入端的移位信号传递给 F 的高位或低位，同时另一个移位端接收 F 的低位或高位。如在右移输入情况下（指令码 I8~6="10-"），在 RAM 操作中，移位端 RAM3 作为输入信号，它和 F3~1 共同构成存储器的输入数据，同时 F0 移入 RAM0。注意在表 82.1 中，对应的 RAM3 定义为高阻状态，这是因为此时它仅作为输入信号使用，输出无意义。

还要注意一个现象：由于我们是分模块独立描述的，所以同一个信号（RAM0 或 RAM3）在存储器描述中被定义为输入信号而在此例中却定义为输出信号，当然信号本身应是双向的。若作为一个整体进行描述，应定义为 **inout** 型。第 81 例中该寄存器的描述与本例中 Q3 和 Q0 的关系也具有上述讨论的现象。

电路的 VHDL 语言描述如下：

```
library 12901_lib;
use 12901_lib.TYPES.all;
use 12901_lib.MVL7_functions.all;

entity output_and_shifter is
       port (
             I                        : in MVL7_vector(3 downto 0);
             A, F, Q                  : in MVL7_vector(3 downto 0);
             OEbar                    : in MVL7;
             Y                        : out MVL7_vector(3 downto 0);
             RAM0, RAM3, Q0, Q3       : out MVL7
            );
end output_and_shifter;

architecture output_and_shifter of output_and_shifter is
begin
    -- 输出信号受使能端 OEbar 的控制
    Y <= A   when (( I(2 downto 0) = "010") and ( OEbar = ' 0' ))
    else F   when (not(( I(2 downto 0) = "010"))  and ( OEbar = ' 0' ))
    else "ZZZZ";                                    -- OEbar 无效，输出高阻
    -- 移位信号
    RAM0 <= F(0)        when ( I(2) = ' 1' ) and ( I(1) = ' 0'  )
                        else  ' Z' ;
    RAM3 <= F(3)        when ( I(2) = ' 1' ) and ( I(1) = ' 1'  )
                        else  ' Z' ;
    Q3      <= Q(3)     when ( I(2) = ' 1' ) and ( I(1) = ' 1' )
                        else  ' Z' ;
    Q0      <= Q(0)     when ( I(2) = ' 1' ) and ( I(1) = ' 0' )
                        else  ' Z' ;
end  output_and_shifter;
```

3．模拟测试向量的选择及模拟结果分析

在结构描述中，我们仍然采用的是组合逻辑结构的描述方法，因此，只要输入端有信号，相应地就要产生输出信号，在模拟中无需时钟信号的同步控制。选择测试向量只

要考虑覆盖输出信号的正常输出和高阻状态即可，比较简单。

下面是几组典型的测试向量。

```
F <= "0101";              --#1
A <= "1110";
Q <= "0000";              -- Y 送到 F
I <= "000";
OEBAR <= '0';
wait for 1 ns;
assert (Y = "0101")          -- 验证 Y
    report "assert a1 : < Y /= '0101' > " severity warning;
assert (RAM0 = 'Z')          -- 验证 RAM0
    report "assert a1 : < RAM0 /= 'Z' > " severity warning;
assert (RAM3 = 'Z')          -- 验证 RAM3
    report "assert a1 : < RAM3 /= 'Z' > " severity warning;
assert (Q0 = 'Z')                -- 验证 Q0
    report "assert a1 : < Q0 /= 'Z' > " severity warning;
assert (Q3 = 'Z')                -- 验证 Q3
    report "assert a1 : < Q3 /= 'Z' > " severity warning;
wait for 1 ns;
F <= "0101";              --#2
A <= "0111";
Q <= "0000";              -- 测试 Y 为高阻
I <= "000";
OEBAR <= '1';
wait for 1 ns;
assert (Y = "ZZZZ")
    report "assert b1 : < Y /= 'ZZZZ' > "  severity warning;
assert (RAM0 = 'Z')
    report "assert b1 : < RAM0 /= 'Z' > "  severity warning;
assert (RAM3 = 'Z')
    report "assert b1 : < RAM3 /= 'Z' > "  severity warning;
assert (Q0 = 'Z')
    report "assert b1 : < Q0 /= 'Z' > "    severity warning;
assert (Q3 = 'Z')
    report "assert b1 : < Q3 /= 'Z' > "    severity warning;
wait for 1 ns;
F <= "1110";              --#4
A <= "0110";
Q <= "0000";              -- A 送到 Y
I <= "010";
```

```
OEBAR <= ' 0' ;
wait for 1 ns;
assert (Y = "0110")
    report "assert d1 : < Y /= ' 0110' > " severity warning;
assert (RAM0 = ' Z' )
    report "assert d1 : < RAM0 /= ' Z' > " severity warning;
assert (RAM3 = ' Z' )
    report "assert d1 : < RAM3 /= ' Z' > " severity warning;
assert (Q0 = ' Z' )
    report "assert d1 : < Q0 /= ' Z' > " severity warning;
assert (Q3 = ' Z' )
    report "assert d1 : < Q3 /= ' Z' > " severity warning;
wait for 1 ns;
F <= "1001";              --#6
A <= "1000";
Q <= "0100";              -- F送到Y , F(0)送到RAM0,  Q(0)送到Q0
I <= "100";
OEBAR <= ' 0' ;
wait for 1 ns;
assert (Y = "1001")
    report "assert f1 : < Y /= ' 1001' > " severity warning;
assert (RAM0 = ' 1' )
    report "assert f1 : < RAM0 /= ' 1' > " severity warning;
assert (RAM3 = ' Z' )
    report "assert f1 : < RAM3 /= ' Z' > " severity warning;
assert (Q0 = ' 0' )
    report "assert f1 : < Q0 /= ' 0' > " severity warning;
assert (Q3 = ' Z' )
    report "assert f1 : < Q3 /= ' Z' > " severity warning;
wait for 1 ns;
F <= "0100";              --#8
A <= "0010";
Q <= "1000";              -- F送到Y、F(3)送到RAM3, Q(3)送到Q3
I <= "110";
OEBAR <= ' 0' ;
wait for 1 ns;
assert (Y = "0100")
    report "assert h1 : < Y /= ' 0100' > " severity warning;
assert (RAM0 = ' Z' )
    report "assert h1 : < RAM0 /= ' Z' > " severity warning;
```

```
assert (RAM3 = '0')
    report "assert hl : < RAM3 /= '0' >" severity warning;
assert (Q0 = 'Z')
    report "assert hl : < Q0 /= 'Z' >" severity warning;
assert (Q3 = '1')
    report "assert hl : < Q3 /= '1' >" severity warning;
wait for 1 ns;
```

图 82.1 为模拟结果的部分波形显示，为观察方便，我们将有关信号进行了打包处理，如 Y3~0 表示 Y[3]Y[2]Y[1]Y[0]等。图中显示了输出 Y 在有关控制信号作用下的变化情况，读者可参照表 82.1 进行分析。

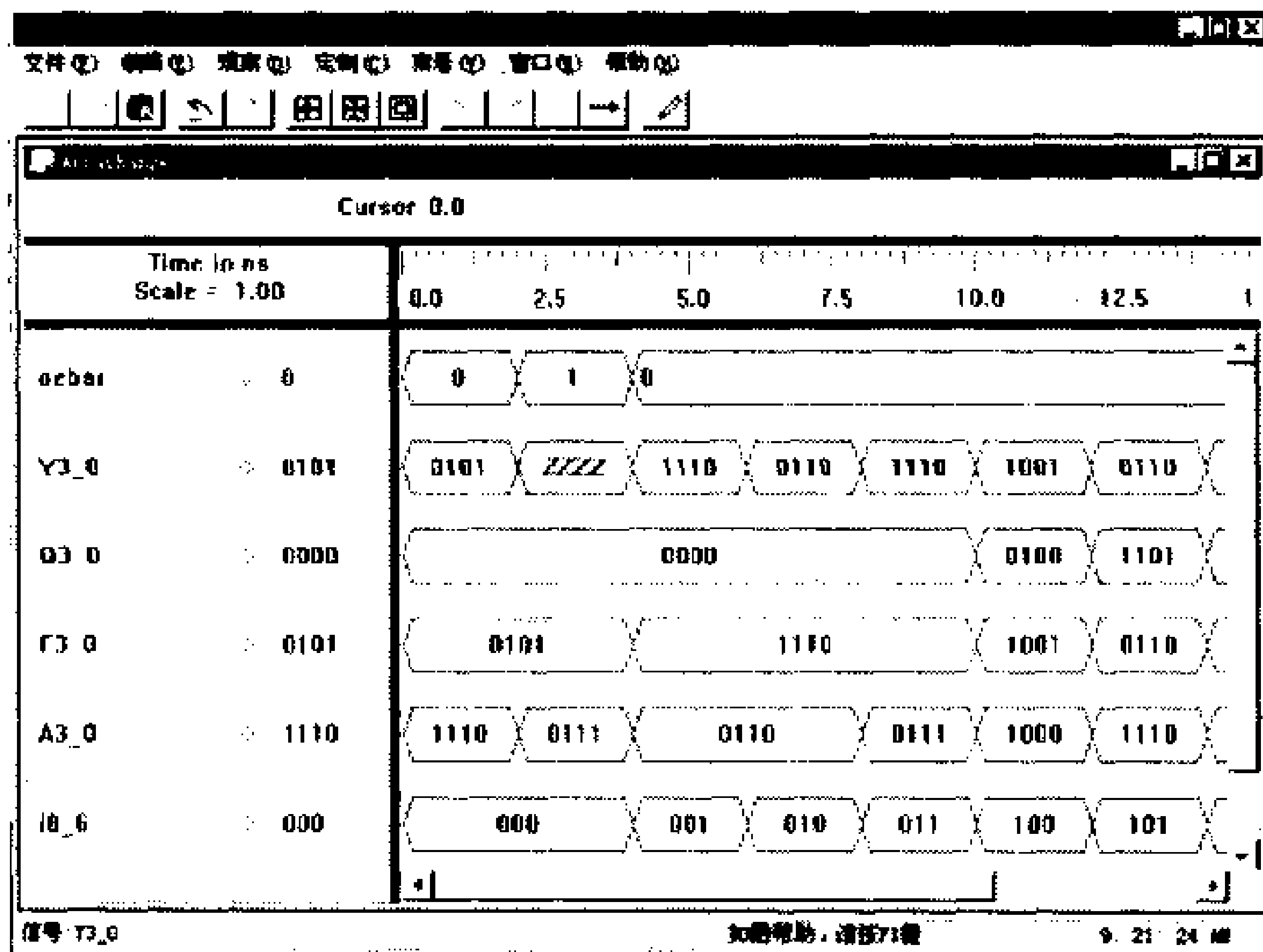


图 82.1 输出/移位模拟波形图

通过第 68 例到第 72 例，我们对 Am2901 的整体性能进行了完整的描述。本例结构较复杂，描述中涉及的 VHDL 语法现象较多。实践证明，采用分模块的方法进行分别描述与验证是有效的。

（源描述文件名：82_output_and_shifter.vhd
测试平台文件名：82_output_shifter_stim.vhd）

第 83 例　Am2910 四位微程序控制器中的多路选择器

韩　曙

1. Am2910 电路系统工作原理

Am2910 是 AMD 公司生产的微程序控制器逻辑芯片，其示意图如图 83.1 所示。

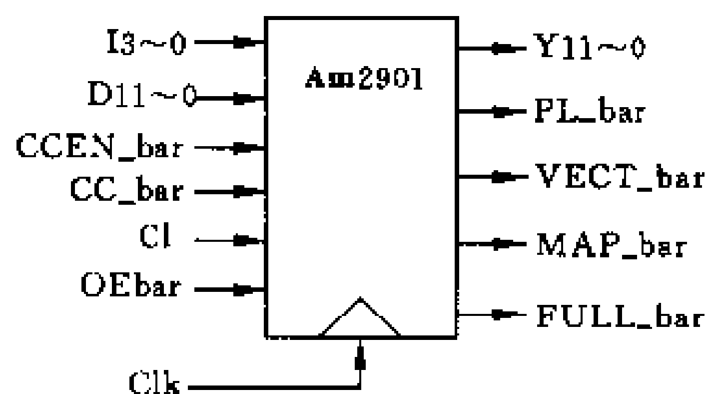


图 83.1　Am2910 实体图

图中，D11~0 和 Y11~0 分别为 12 位微指令地址输入／输出代码，用来表示最多 4096 个不同的微指令地址。I 3~0 为微指令控制码，表示 16 种不同的微操作。其他控制及标志信号的意义见表 83.1。

表 83.1　Am2910 标志 / 控制位解释

名称	I/O	意义	
CC_bar	I	测试判别位，低电平表示测试通过	详见表 87.1
CCEN_bar	I	当它为高电平时，忽略 CC_bar	
CI	I	微程序计数器的低位进位端	
RLD_bar	I	当它为低电平时，将 I11~0 装入寄存器	
Oebar	I	输出 Y 11~0 的控制端，低电平有效，高电平时输出端呈高阻态	
CLK	I	时钟	
FULL_bar	O	堆栈满标志位	
PL_bar	O	直接输入类型控制信号 1	注：在任一个微指令周期，这 3 个输出标志有且只有一个为有效信号，可分别用来控制跳转、PROM或中断／DMA的子程序的入口地址等
MAP_bar	O	直接输入类型控制信号 2	
VECT_bar	O	直接输入类型控制信号 3	

Am2910 的结构框图如图 83.2 所示。从电路功能上 Am2910 可分为 5 个相对独立的部分：

① 多路选择器（multiplexer），分别选择 4 类不同的微指令数据源，以实现微程序的顺序执行、分支、寄存器寻址和堆栈寻址等执行方式；

② 寄存器／计数器（register/counter）：和其他标志位一起用于控制微指令的循环和分支执行方式；

③ 微程序计数器／寄存器（microprogram counter/register），指示下一条要执行的微指令的地址；

④ 堆栈及堆栈指针寄存器（stack/stack pointer）；

⑤ 指令译码器（instruction PLA），根据指令类型，发出相应的微控制命令，使各部分协调执行相应的微指令。

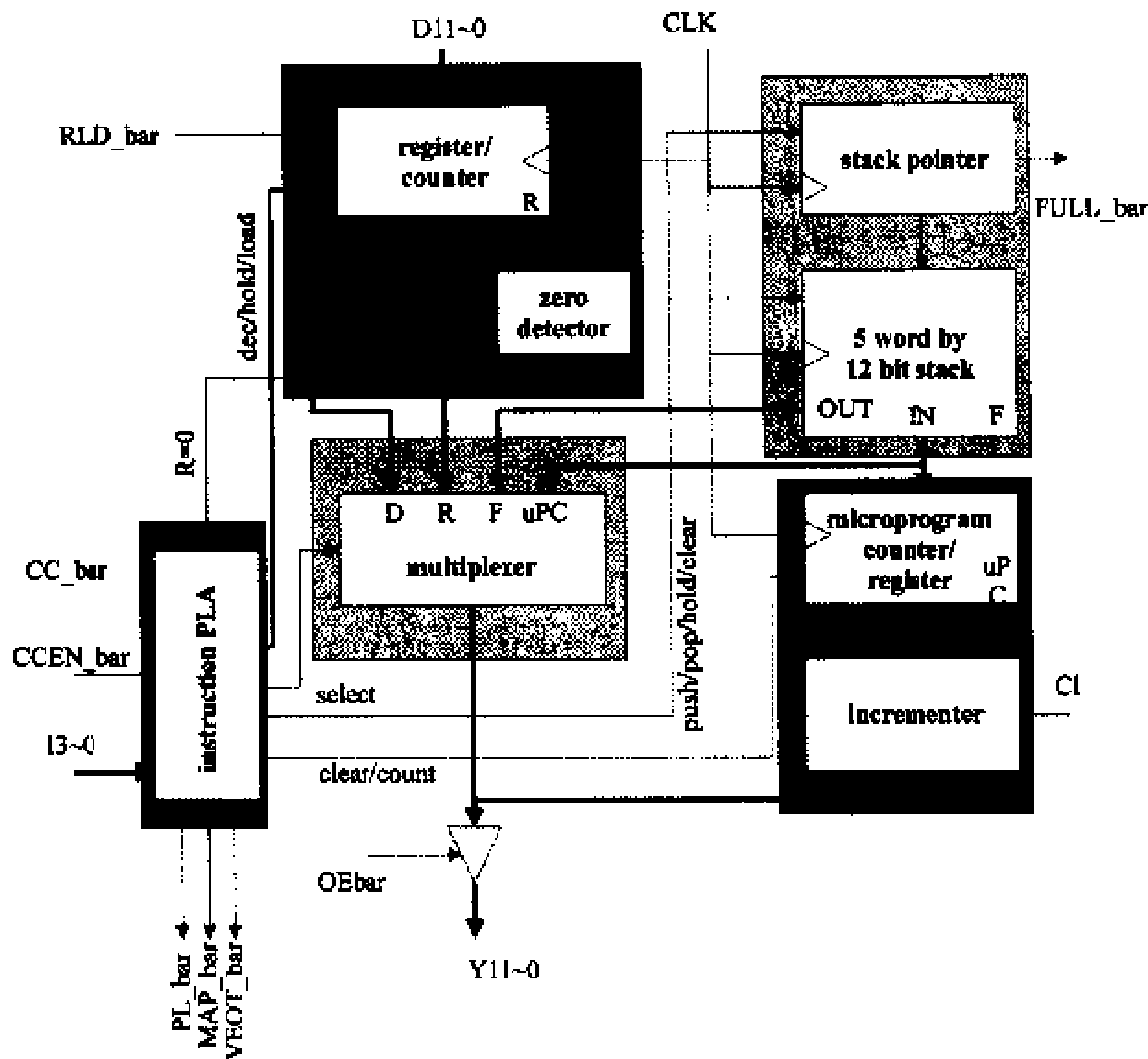


图 83.2 Am2901 结构框图

本电路与第 78 例到第 82 例讨论的 AM2901 电路类似，也是较为复杂的数字系统，为讨论方便，根据其逻辑功能将其划分为 5 个相对独立的部分进行分别描述和模拟验证。在各部分的描述中，使用了数据类型 MVL7。

下面首先分析多路器的功能及描述方法。

多路选择器用于选择下一条欲执行指令的地址，共有 4 类地址可供选择：① 微程序地址寄存器（uPC），这个寄存器每执行一条指令其值增 1 。因此，选择它表示顺序执行微程序的下一条指令。② 外部直接输入地址（D11~0），在程序发生跳转时，它表示跳转后的程序首条指令地址。③ 寄存器/计数器数据（R），可以表示上条指令存入的地址，用于程序执行中的条件分支等。④ 堆栈中的地址，表示执行子程序、响应中断以及循环等。

4 个数据源分别由 uPC_sel，D_sel，R_sel 和 stack sel 进行选择（图中简记为 select），其中关于堆栈的操作实质上是存储器的操作，所以还要给出堆栈指针 Sp，输出一方面送微指令程序计数器（自动增 1），同时通过由 OEbar 控制的三态门输出。

2. 电路的 VHDL 语言描述方法及语法分析

通过前面的分析可知，本子系统是典型的组合逻辑系统，但是这里又涉及存储器（堆栈）的操作，关于存储器的特点及描述方法在第 80 例中已作过论述。本例的描述中用到了块语句结构。块是 VHDL 中对复杂系统模块化描述的一种有效机制，这种机制使得设计者在描述整个系统时，按系统的逻辑功能进行分块描述。任何能在结构体说明中说明的对象都能在块说明部分加以说明，块可以有嵌套结构，任何一个低层的块能引用它本层

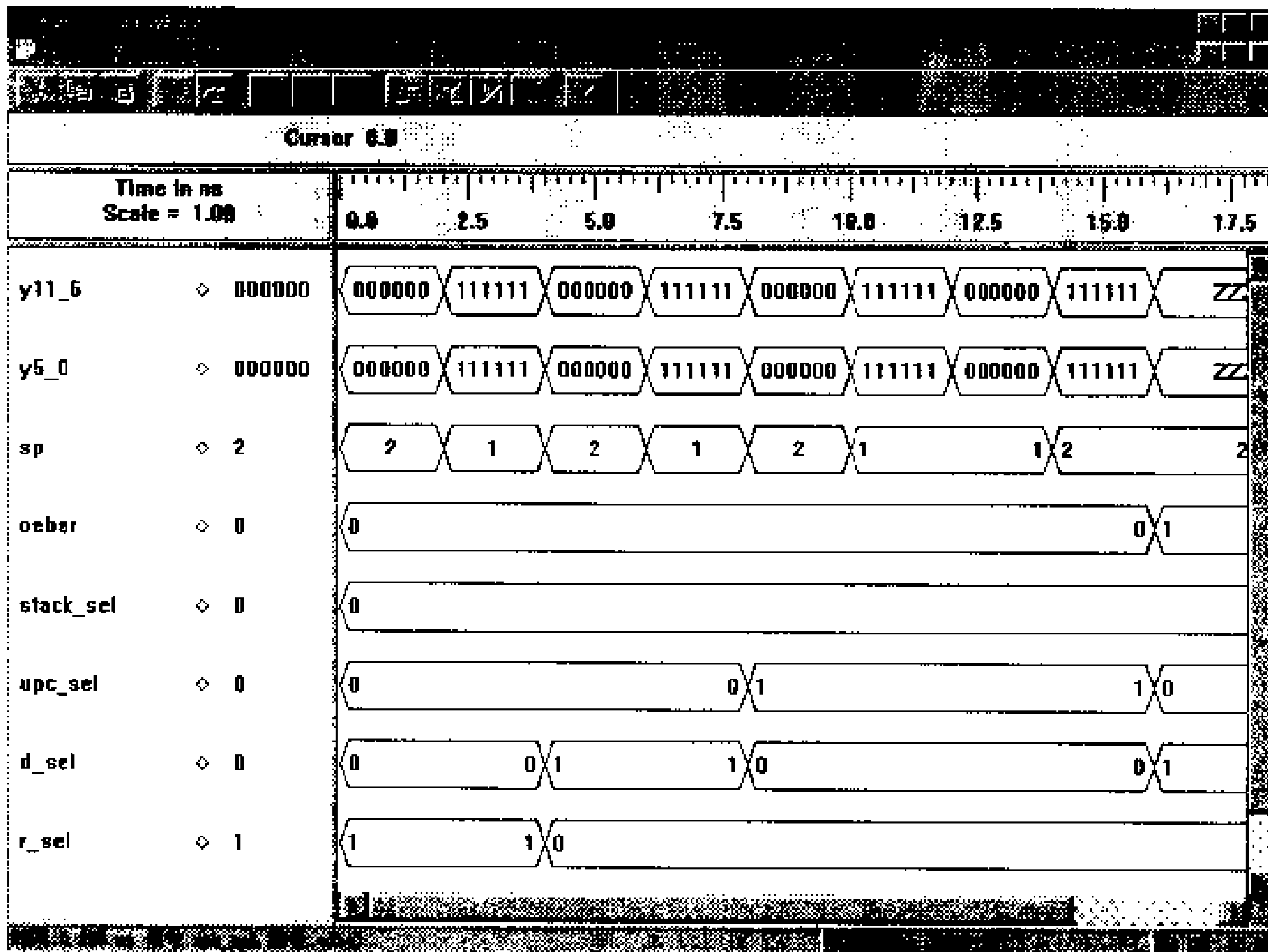


图 83.3　Am2910 多路选择器的模拟波形

以上的信号。块有保护和无保护两种使能形式，对保护的块结构，只有当保护逻辑表达式为真时，才执行块内语句，否则不执行。而无保护的块描述在任何时候都使能。在第 80 例和第 81 例的描述中使用了保护的块结构描述方式，由于本例是组合逻辑系统，采用无保护的形式。

第 15 例对本例涉及的描述中的语法现象等问题做了详细研究，在此恕不赘述。为方便读者分析，图 83.3 也给出了部分模拟结果的波形显示。

（源描述文件名：83_multiplexer.vhd
测试平台文件名：83_multiplexer_stim.vhd）

第 84 例　Am2910 四位微程序控制器中的计数器/寄存器

韩　曙

1. 电路系统工作原理

计数器／寄存器（R）是 12 位递减计数器，它和零标志位一起用于微程序的循环和分支。循环次数 N 是在控制信号 RLD_bar 的作用下通过 D11~0 预先送入 R 的，循环每执行一遍 R 递减 1，直到 R＝0 时结束循环，所以循环次数为 N+1 次。它接收指令译码器的指令信号共有 3 类：

① 预置（load）：在下一个 CP 脉冲到来时，若 load = ‘1’或 RLD_bar = ‘0’，将数据 D11~0 置入计数器；

② 递减（decrement）：在下一个 CP 脉冲到来时，若 decr = ‘1’且 RLD_bar = ‘0’，计数器当前值递减 1；

③ 保持（hold）：在下一个 CP 脉冲到来时，若上述两项的条件都不满足，则计数器的值保持不变。

计数器当前值的状态（是否为零）通过零标志检测电路送回指令译码器。

2. 电路的 VHDL 描述方法及语法分析

这是一个时序逻辑系统，电路中的状态在时钟信号上升沿的作用下进行更新。电路的 VHDL 描述中使用了保护的块结构，保护表达式为时钟信号的上升沿。只有当表达式的值为真时，才执行块内语句。本电路的描述方法与第 80 例和第 81 例类似。在本例的描述中，假设指令译码器发出的 3 个控制信号 decr，hold 和 load 互斥，即在任何时候这三个信号最多有一个有效。在实际描述中，我们把除 decr 和 load 以外的情况都认为是 hold，这样就可以简化描述。否则在描述时就必须考虑它们同时有效的情况，相应的描述也就复杂了（参考第 83 例的讨论）。在电路描述中，保持是通过将输出信号回送给计数器而实现的，所以，RE 被定义为双向数据端。

```
-- 名称：AMD 2910 计数器/寄存器

library l2901_lib;
use l2901_lib.types.all;
use l2901_lib.MVL7_functions.all;
use l2901_lib.synthesis_types.all;
```

```
entity reg is
port (
      RLD_bar  :  in MVL7;          -- 预置命令输入端
         load  :  in MVL7;          -- 预置指令输入
         decr  :  in MVL7;          -- 递减指令
          CLK  :  in clock;         -- 时钟信号
            D  :  in MVL7_VECTOR(11 downto 0);        -- 预置计数值输入
           RE  :  inout MVL7_VECTOR(11 downto 0);     -- 数据输出/保持输入
    Rzero_bar  :  out MVL7
      );
end reg;

architecture reg of reg is

begin
     -- 保护的块结构
reg_ctr : block ( (CLK = '1') and (not CLK' stable) )
begin
     RE <= guarded D when (( load = '1' ) or (RLD_bar = '0' ))        -- 预置
          elseRE - "000000000001" when (decr = '1' ) and (RLD bar = '1' )
                                                                      --递减
          else  RE ;                                                -- 保持(信号回送)
     Rzero_bar <=  RE(0) or RE(1) or RE(2) or RE(4) or RE(5) or RE(6)
                   or RE(7) or RE(8) or RE(9) or RE(10) or RE (11);
                                                                      -- 零标志
end block reg_ctr;

end reg;
```

3. 模拟测试向量的选择及模拟结果分析

构造的测试向量要涵盖如下几种情况：

① 对递减和保持指令，在递减指令有效时，分析 RLD_bar 有效和无效时对计数器值的影响；

② 对预置指令，分别分析在预置指令有效和（或）RLD_bar 有效的情况下，计数器值的变化；

③ 注意输出信号的双向特性；

④ 注意零标志位。

部分测试向量描述如下：

```
CLK <= '0' ;                    -- 时钟信号置零
wait for 1 ns;
RLD_bar <= '1' ;                -- 无效信号
load <= '1' ;                   -- 预置指令
decr <= '0' ;
D <= "000000000000";
wait for 4 ns;
CLK <= '1' ;                    -- 时钟置 1，上跳
wait for 4 ns;
assert (Rzero_bar = '0' )       -- 验证零标志
    report "Assert 0 : < Rzero_bar /= 0 >" severity warning;
assert (RE = "000000000000") -- 验证预置是否正确
    report "Assert 1 : < RE /= 000000000000 >"  severity warning;
wait for 1 ns;

CLK <= '0' ;
wait for 1 ns;
RLD_bar <= '1' ;
load <= '1' ;
decr <= '0' ;
D <= "111111111111";          -- 验证另一组数预置是否正确
wait for 4 ns;
CLK <= '1' ;
wait for 4 ns;
assert (Rzero_bar = '1' )
    report  "Assert 2 : < Rzero_bar /= 1 >"  severity warning;
assert (RE = "111111111111")
    report "Assert 3 : < RE /= 111111111111 >"  severity warning;
wait for 1 ns;

CLK <= '0' ;
wait for 1 ns;
RLD_bar <= '0' ;         -- 非指令预置方式
load <= '0' ;
decr <= '0' ;
D <= "000000000000";
```

```
wait for 4 ns;
CLK <= ' 1' ;
wait for 4 ns;
assert (Rzero_bar = ' 0' )
    report "Assert 4 : < Rzero_bar /= 0 >" severity warning;
assert (RE = "000000000000")
    report "Assert 5 : < RE /= 000000000000 >" severity warning;
wait for 1 ns;

CLK <= ' 0' ;
wait for 1 ns;
RLD_bar <= ' 1' ;          -- 无效
load <= ' 0' ;             -- 无效
decr <= ' 0' ;                                   -- 无效
D <= "111111111111";
wait for 4 ns;
CLK <= ' 1' ;
wait for 4 ns;
assert (Rzero_bar = ' 0' )                       -- 保持上一状态不变
    report "Assert 6 : < Rzero_bar /= 0 >" severity warning;
assert (RE = "000000000000")
    report "Assert 7 : < RE /= 000000000000 >" severity warning;
wait for 1 ns;

CLK <= ' 0' ;
decr <= ' 1' ;                                   -- 递减指令
RLD_bar <= ' 1' ;                                -- 无效
wait for 4 ns;
CLK <= ' 1' ;
wait for 4 ns;                                   -- 上一次状态为 RE="111111111111"
assert (RE = "111111111110")                     --验证结果
    report "Assert 12 : < RE /= 111111111110 >" severity warning;
wait for 1 ns;

CLK <= ' 0' ;
wait for 1 ns;
decr <= ' 1' ;                                   -- 递减指令有效
RLD_bar <= ' 0' ;                                -- 同时外部预置信号有效
```

```
D <= "111111111111";                    -- 执行预置功能
wait for 4 ns;
CLK <= ' 1' ;
wait for 4 ns;
assert (RE = "111111111111")
    report "Assert 21 : < RE /= 111111111111 >" severity warning;
wait for 1 ns;
```

本例的部分模拟波形如图 84.1 所示。

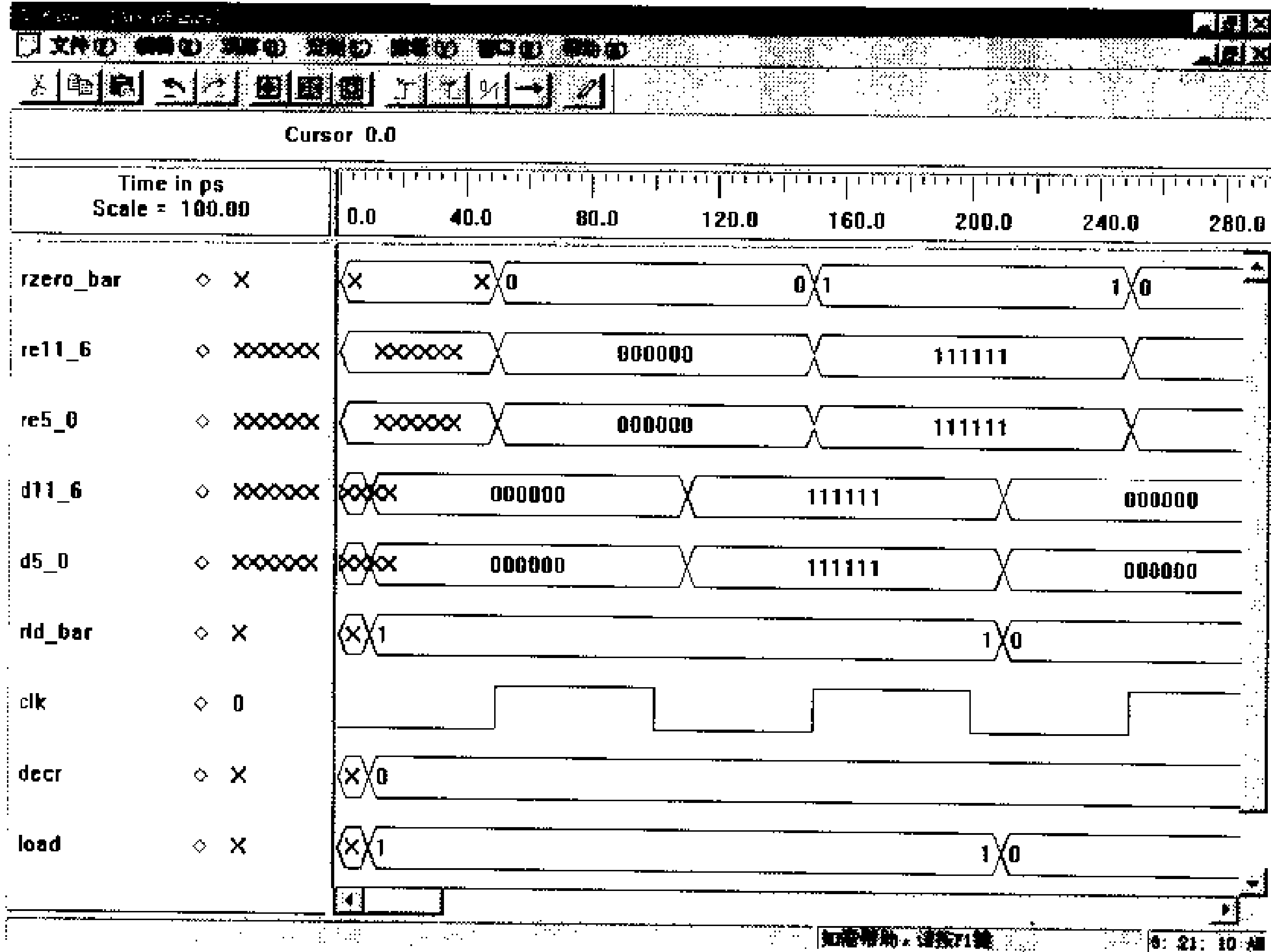


图 84.1 模拟波形

（源描述文件名：84_reg.vhd
测试平台文件名：84_reg_stim.vhd）

第85例　Am2910 四位微程序控制器的指令计数器

韩　曙

1. 电路系统工作原理

指令计数器（uPC）是12位计数器，它和第84例讨论的计数器的原理类似．其功能是控制微指令的执行，它的内容是下一条要执行的微指令的地址。指令计数器有两种工作方式：在指令信号 clear='1'的作用下清零，用于在程序执行以前对系统的初始化，使微程序在运行时都从0号地址开始执行；在指令信号 clear='0'时，实现地址数 Y_temp 和 CI 相加，若 CI=1，在下一个时钟脉冲到来时自动加1，从而实现指令的顺序执行。若 CI=0，实现 uPC<=Y，重复执行上一条指令。

2. 电路的 VHDL 描述方法及语法分析

在电路描述上，仍然采用保护的块结构方式，保护表达式为时钟信号的上升沿。信号 uPC 仍定义为 **inout** 方式，以实现数据的保持功能。电路描述如下：

```
library l2901_lib;
use l2901_lib.types.all;
use l2901_lib.MVL7_functions. all;

entity upc is
port (
        CLK     : in clock;
        CI      : in MVL7;
        clear   : in MVL7;
        Y_temp  : in MVL7_VECTOR(11 downto 0);
        uPC     : inout MVL7_VECTOR(11 downto 0)
     );
end upc;

architecture upc of upc is
begin
PC : block ( (CLK = ' 1' ) and (not CLK' STABLE) ) - 保护块结构
```

```
begin
    uPC <= guarded Y_temp + ("00000000000" & CI) when (clear = '0') --计数
          else    "000000000000"                when clear = '1'  -- 清零
          else     uPC;                                             -- 保持
end block PC;
end upc;
```

3. 模拟测试向量的选择及模拟结果分析

本例的测试向量主要覆盖清零、递增、保持和直接接收这 4 种情况。它们的测试向量分别为：

```
CLK <= '0';
wait for 1 ns;
CI <= '0';          -- 实现 uPC <- Y_temp
clear <= '0';       -- 计数状态
Y_temp <= "000000000000";
wait for 4 ns;
CLK <= '1';         -- 时钟上跳
wait for 4 ns;
assert (uPC = "000000000000")     -- 验证结果
    report "Assert 0 : < uPC = 000000000000 >" severity warning;
wait for 1 ns;

CLK <= '0';
wait for 1 ns;
CI <= '0';          -- 实现 uPC <= Y_temp
clear <= '0';       -- 计数状态
Y_temp <= "111111111111";
wait for 4 ns;
CLK <= '1';
wait for 4 ns;
assert (uPC = "111111111111")          -- 验证结果
    report "Assert 1 : < uPC = 111111111111 >" severity warning;
wait for 1 ns;

CLK <= '0';
```

```
wait for 1 ns;
CI <= '0';
clear <= '1';          -- 清零
Y_temp <= "111111111111";
wait for 4 ns;
CLK <= '1';
wait for 4 ns;
assert (uPC = "000000000000")          -- 验证结果
    report "Assert 2 : < uPC = 000000000000 >" severity warning;
wait for 1 ns;

CLK <= '0';
wait for 1 ns;
CI <= '1';          -- 实现 uPC<=Y_temp + '1'
clear <= '0';     — 计数状态
Y_temp <= "000000000000";
wait for 4 ns;
CLK <= '1';
wait for 4 ns;
assert (uPC = "000000000001")
    report "Assert 3 : < uPC = 000000000001 >" severity warning;
wait for 1 ns;

CLK <= '0';
wait for 1 ns;
CI <= '1';          -- 实现 uPC<- Y_temp + '1'
clear <= '0';     -- 计数状态
Y_temp <= "000000000001";
wait for 4 ns;
CLK <= '1';
wait for 4 ns;
assert (uPC = "000000000010")
    report "Assert 4 : < uPC = 000000000010 >" severity warning;
wait for 1 ns;
```

模拟结果的波形图如图 85.1 所示，请读者自行分析。

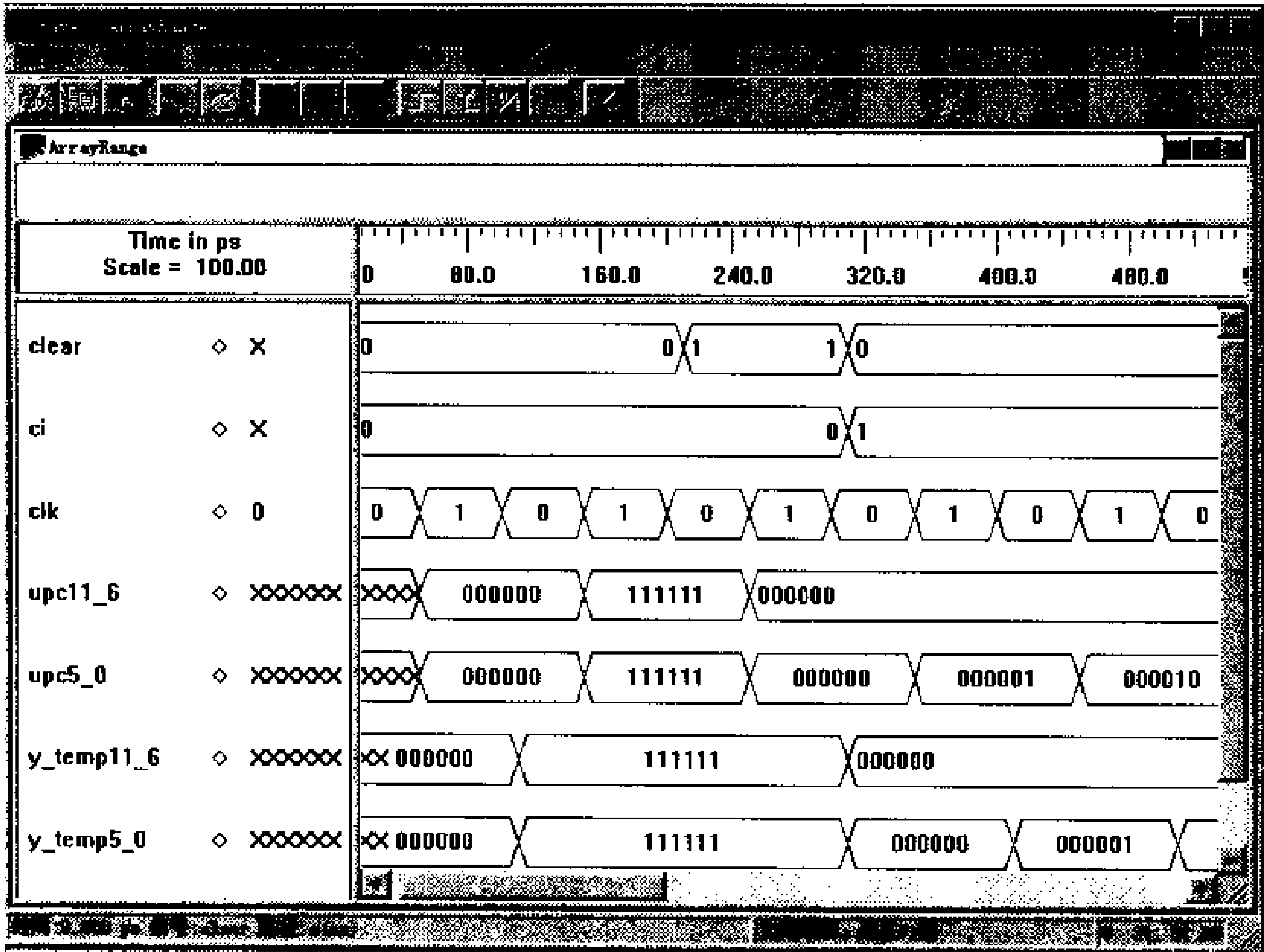


图 85.1　部分模拟结果波形图

（源描述文件名：85_upc.vhd
测试平台文件名：85_upc_stim.vhd）

第 86 例 Am2910 四位微程序控制器的堆栈

韩 曙

1. 电路系统工作原理

AM2910 的堆栈共有 6 个 12 位存储单元，其组织方式遵从先进后出的原则。堆栈指针总指向栈顶，执行 POP 操作时，堆栈指针减 1；执行 PUSH 操作时，堆栈指针增 1。

由于堆栈操作遵从先进后出原则，在电路结构上，它和普通的存储器组织不完全相同，需考虑的因素包括：

- 合理的存储器操作，从堆栈中存取一个操作数只能从栈顶进行存取；
- 堆栈指针操作，对指针的运算及溢出（上溢和下溢）判断；
- 堆栈的初始化。

2. 电路的 VHDL 描述方法及语法分析

描述仍采用保护的块结构方式。保护逻辑表达式为时钟信号的上升沿。

```
library 12901_lib;
use 12901_lib.types.all;
use 12901_lib.MVL7_functions.all;

entity stack is
port (
            CLK : in clock;
            pop : in MVL7;
           push : in MVL7;
          clear : in MVL7;
            uPC : in MVL7_VECTOR(11 downto 0);
             Sp : inout INTEGER range 0 to 5;
       reg_file : inout MEMORY_12_BIT(5 downto 0);-- 存储器数组 6×12 位
       FULL_BAR : out MVL7
       );
end stack;
architecture stack of stack is

begin
stack_and_Sp : block ( (CLK = '1') and (not CLK'STABLE) )-- 保护块结构
```

```
signal write_address : integer range 0 to 5;          -- 堆栈地址信号
begin
    Sp <= guarded (Sp - 1) when (pop = '1') and (Sp /= 0)
                                                     -- 出栈操作，若 sp/=0（栈底）有效
        else        (Sp + 1) when (push = '1') and (Sp /= 5)
                                                     -- 压栈操作，若 sp/=5（栈顶）有效
        else     0 when clear = '1'      -- 清 0
        else     Sp;                     -- 保持地址信号
    write_address <= Sp + 1 when (Sp /= 5)
 else Sp;
    reg_file(write_address) <= guarded uPC when (push = '1')    -- 压栈
                                else    reg_file(write_address);  -- 出栈
    FULL_BAR <= '0' when Sp = 5                                 -- 栈顶满指针
                    else   '1';
    end block stack_and_Sp;
end stack;
```

3. 模拟测试向量的选择及模拟结果分析

下面选择的测试向量从堆栈的初始化开始，逐步进行压栈操作，再从栈顶逐步弹出，直至栈空。

```
CLK <= '0';                  -- 初始化（Sp=0,Full_bar=1）
wait for 1 ns;
pop <= '0';
push <= '0';
clear <= '1';                -- 初始化指令
uPC <= "000000000000";       -- 栈地址
wait for 4 ns;               -- 栈指针 Sp
CLK <= '1';
wait for 4 ns;
assert (Sp = 0)
    report "Assert 0 : < Sp /= 0 >" severity warning;
assert (FULL_BAR = '1')
    report "Assert 0a : < FULL_BAR /= 1 >" severity warning;
wait for 1 ns;

CLK <= '0';                  --压栈操作 1      5:
wait for 1 ns;               --               4:
pop <= '0';                  --               3:
push <= '1';                 --               2:
clear <= '0';                --            -> 1:
```

```
uPC <= "000000000001";--                   0: 000000000001
wait for 4 ns;
CLK <= '1';
wait for 4 ns;
assert (Sp = 1)
    report "Assert 1 : < Sp /= 1 >" severity warning;
assert (FULL_BAR = '1' )
    report "Assert 1a : < FULL_BAR /= 1 >" severity warning;
assert (reg_file(Sp) = "000000000001" )
    report "Assert 1b : < reg_file(Sp) /= 000000000001 >" severity warning;
wait for 1 ns;

CLK <= '0';            --压栈操作2          5:
wait for 1 ns;         --                   4:
pop <= '0';            --                   3:
push <= '1';           --              -->  2:
clear <= '0';          --                   1: 000000000010
uPC <= "000000000010"; --                   0: 000000000001
wait for 4 ns;
CLK <= '1';
wait for 4 ns;
assert (Sp = 2)
    report "Assert 2 : < Sp /= 2 >" severity warning;
assert (FULL_BAR = '1' )
    report "Assert 2a : < FULL_BAR /= 1 >" severity warning;
assert (reg_file(Sp) = "000000000010" )
    report "Assert 2b : < reg_file(Sp) /= 000000000010 >" severity warning;
wait for 1 ns;
CLK <= '0';            --压栈操作3          5:
wait for 1 ns;         --                   4:
pop <= '0';            --              -->  3:
push <= '1';           --                   2: 000000000100
clear <= '0';          --                   1: 000000000010
uPC <= "000000000100"; --                   0: 000000000001
wait for 4 ns;
CLK <= '1';
wait for 4 ns;
assert (Sp = 3)
    report "Assert 3 : < Sp /= 3 >" severity warning;
assert (FULL_BAR = '1' )
    report "Assert 3a : < FULL_BAR /= 1 >" severity warning;
assert (reg_file(Sp) = "000000000100" )
```

```
        report "Assert 3b : < reg_file(Sp) /= 000000000100 >" severity warning;
    wait for 1 ns;

    CLK <= '0';              --压栈操作 4        5:
    wait for 1 ns;           --            --> 4:
    pop <= '0';              --                3: 000000001000
    push <= '1';             --                2: 000000000100
    clear <= '0';            --                1: 000000000010
    uPC <= "000000001000";   --                0: 000000000001
    wait for 4 ns;
    CLK <= '1';
    wait for 4 ns;
    assert (Sp = 4)
        report "Assert 4 : < Sp /= 4 >" severity warning;
    assert (FULL_BAR = '1')
        report "Assert 4a : < FULL_BAR /= 1 >" severity warning;
    assert (reg_file(Sp) = "000000001000")
        report "Assert 4b : < reg_file(Sp) /= 000000001000 >" severity warning;
    wait for 1 ns;

    CLK <= '0';              --压栈操作 5    -->    5:
    wait for 1 ns;           --                     4: 111111111111
    pop <= '0';              --                     3: 000000001000
    push <= '1';             --                     2: 000000000100
    clear <= '0';            --                     1: 000000000010
    uPC <= "111111111111";   --                     0: 000000000001
    wait for 4 ns;
    CLK <= '1';
    wait for 4 ns;
    assert (Sp = 5)
        report "Assert 5 : < Sp /= 5 >" severity warning;
    assert (FULL_BAR = '0')
        report "Assert 5a : < FULL_BAR /= 0 >" severity warning;
    assert (reg_file(Sp) = "111111111111")
        report "Assert 5b : < reg_file(Sp) /= 111111111111 >" severity warning;
    wait for 1 ns;

    CLK <= '0';              --压栈操作 5    -->    5: 000000010000
    wait for 1 ns;           --                     4: 111111111111
    pop <= '0';              --                     3: 000000001000
    push <= '1';             --                     2: 000000000100
    clear <= '0';            --                     1: 000000000010
```

```
uPC <= "000000010000";  --                        0: 000000000001
wait for 4 ns;
CLK <= '1';
wait for 4 ns;
assert (Sp = 5)
    report "Assert 6 : < Sp /= 5 >"  severity warning;
assert (FULL_BAR = '0')    -- 栈顶满指针仍然有效
    report "Assert 6a : < FULL_BAR /= 0 >" severity warning;
assert (reg_file(Sp) = "000000010000" )
    report "Assert 6b : < reg_file(Sp) /= 000000010000 >" severity warning;
wait for 1 ns;

CLK <= '0';            --出栈操作 1        5: 000000010000
wait for 1 ns;         --            —>   4: 111111111111
pop <= '1';            --                 3: 000000001000
push <= '0';           --                 2: 000000000100
clear <= '0';          --                 1: 000000000010
                       --                 0: 000000000001
CLK <= '1';
wait for 4 ns;
assert (Sp = 4)
    report "Assert 7 : < Sp /= 4 >"  severity warning;
assert (FULL_BAR = '1')
    report "Assert 7a : < FULL_BAR /= 1 >"  severity warning;
assert (reg_file(Sp) = "000000001000" )
    report "Assert 7b : < reg_file(Sp) /= 000000001000 >" severity warning;
wait for 1 ns;
CLK <= '0';            --出栈操作 2        5: 000000010000
wait for 1 ns;         --                 4: 111111111111
pop <= '1';            --            —>   3: 000000001000
push <= '0';           --                 2: 000000000100
clear <= '0';          --                 1: 000000000010
                       --                 0: 000000000001
wait for 4 ns;
CLK <= '1';
wait for 4 ns;
assert (Sp = 3)
    report "Assert 8 : < Sp /= 3 >"  severity warning;
assert (FULL_BAR = '1')
    report "Assert 8a : < FULL_BAR /= 1 >"  severity warning;
assert (reg_file(Sp) = "000000000100" )
    report "Assert 8b : < reg_file(Sp) /= 000000000100 >"  severity warning;
```

```
wait for 1 ns;
CLK <= ' 0' ;                --出栈操作 3         5: 000000010000
wait for 1 ns;               --                  4: 111111111111
pop <= ' 1' ;                --                  3: 000000001000
push <= ' 0' ;               --              --> 2: 000000000100
clear <= ' 0' ;              --                  1: 000000000010
                             --                  0: 000000000001
wait for 4 ns;
CLK <= ' 1' ;
wait for 4 ns;
assert (Sp = 2)
    report "Assert 9 : < Sp /= 2 >" severity warning;
assert (FULL_BAR = ' 1' )
    report "Assert 9a : < FULL_BAR /= 1 >" severity warning;
assert (reg_file(Sp) = "000000000010" )
    report "Assert 9b : < reg_file(Sp) /= 000000000010 >" severity warning;
wait for 1 ns;
CLK <= ' 0' ;                --出栈操作 4         5: 000000010000
wait for 1 ns;               --                  4: 111111111111
pop <= ' 1' ;                --                  3: 000000001000
push <= ' 0' ;               --                  2: 000000000100
clear <= ' 0' ;              --              --> 1: 000000000010
                             --                  0: 000000000001
wait for 4 ns;
CLK <= ' 1' ;
wait for 4 ns;
assert (Sp = 1)
    report "Assert 10 : < Sp /= 1 >" severity warning;
assert (FULL_BAR = ' 1' )
    report "Assert 10a : < FULL_BAR /= 1 >" severity warning;
assert (reg_file(Sp) = "000000000001" )
    report "Assert 10b : < reg_file(Sp) /= 000000000001 >" severity warning;
wait for 1 ns;
CLK <= ' 0' ;                --出栈操作 5         5: 000000010000
wait for 1 ns;               --                  4: 111111111111
pop <= ' 1' ;                --                  3: 000000001000
push <= ' 0' ;               --                  2: 000000000100
clear <= ' 0' ;              --                  1: 000000000010
                             --              --> 0: 000000000001
wait for 4 ns;
CLK <= ' 1' ;
wait for 4 ns;
```

```
assert (Sp = 0)
    report "Assert 11 : < Sp /= 0 >" severity warning;
assert (FULL_BAR = '1')
    report "Assert 11a : < FULL_BAR /= 1 >" severity warning;
wait for 1 ns;
```

（源描述文件名：86_stack.vhd
测试平台文件名：86_stack_stim.vhd）

第 87 例　Am2910 四位微程序控制器的指令译码器

韩　曙

1. 电路系统工作原理

指令译码器接收外部输入的指令代码 I3~0 与机器的当前状态标志相结合（R=0 标

表 87.1　微指令译码表

I3~0	助记符	零标志 R=0	FAIL CCEN_bar=L and CC_bar=H		PASS CCEN_bar=H Or CC_bar=L		寄存器／计数器 R	PL_bar/ MAP_bar/ VECT_bar	说明
			输出 Y	堆栈 F	输出 Y	堆栈 F			
0	JZ	—	0	CLEAR	0	CLEAR	HOLD	PL_bar	系统复位
1	CJS	—	PC	HOLD	D	PUSH	HOLD	PL_bar	子程序条件转移
2	JMAP	—	D	HOLD	D	HOLD	HOLD	MAP_bar	MAP 转移
3	CJP	—	PC	HOLD	D	HOLD	HOLD	PL_bar	条件 PL 转移
4	PUSH	—	PC	PUSH	PC	PUSH	注 1	PL_bar	PUSH／条件送计数值
5	JSRP	—	R	PUSH	D	PUSH	HOLD	PL_bar	子程序条件转移
6	CJV	—	PC	HOLD	D	HOLD	HOLD	VECT_bar	条件转移矢量
7	JRP	—	R	HOLD	D	HOLD	HOLD	PL_bar	条件跳转
8	RFCT	≠ 0	F	HOLD	F	HOLD	DEC	PL_bar	条件循环
		= 0	PC	POP	PC	POP	HOLD	PL_bar	
9	RPCT	≠ 0	D	HOLD	D	HOLD	DEC	PL_bar	条件循环
		= 0	PC	HOLD	PC	HOLD	HOLD	PL_bar	
10	CRTN	—	PC	HOLD	F	POP	HOLD	PL_bar	条件返回
11	CJPP	—	PC	HOLD	D	POP	HOLD	PL_bar	条件转移
12	LDCT	—	PC	HOLD	PC	HOLD	LOAD	PL_bar	计数值装入与循环
13	LOOP	—	F	HOLD	PC	POP	HOLD	PL_bar	结束循环检测
14	CONT	—	PC	HOLD	PC	HOLD	HOLD	PL_bar	继续
15	TWB	≠ 0	F	HOLD	PC	POP	DEC	PL_bar	三分支跳转
		= 0	D	POP	PC	POP	HOLD	PL_bar	

注 1：若 CCEN_bar=L AND CC_bar=H HOLD ；否则 LOAD

志位、CC_bar 和 CCEN_bar），经译码发出一系列的控制命令，使微指令控制器协调一致地工作。它是整个电路的核心部分。指令译码如表 87.1 所示。

下面，对表中的各条微指令作一简单说明：

① 指令 0：JZ（JUMP to ZERO 或 RESET）无条件复位指令。它强制输出 Y 为 0，堆栈指针指向栈底。

② 指令 1：JSB（CONDITIONAL JUMP-TO-SUBROUTINE）条件转子指令。若测试通过条件成立（PASS），子程序的地址由（D）端输入，转子前要保护现场（顺序执行的下一条指令地址进栈），子程序的最后一条指令应是 CRTN，从而保证恢复现场（顺序执行的下一条指令地址出栈）；否则（FAIL）顺序执行下条指令（PC）。

③ 指令 2：JMAP（JUMP MAP）映射跳转。这是一条无条件跳转指令，执行这条指令时 MAP_bar 有效，由它控制可将 ROM 等的地址装入微程序控制器，从而执行 ROM 中的程序。

④ 指令 3：CJP（CONDITIONAL JUMP PIPELINE）流水线程序条件跳转。根据测试条件（FAIL 或 PASS）执行流水线程序（地址由流水线寄存器 R 给出）或顺序执行下一条程序。执行这条指令时 PL_bar 有效。它允许微程序控制器执行不同的微程序序列。

⑤ 指令 4：PUSH（PUSH/CONDITIONAL LOAD COUNTER）压栈／条件计数值预置。根据测试条件（FAIL 或 PASS）将循环计数值预置入计数器（R）。

⑥ 指令 5：JSRP（CONDITIONAL JUMP-TO-SUBROUTINE）条件转子指令。若测试通过条件成立（PASS），子程序的地址由（D）端输入；否则（FAIL）转到寄存器表示的子程序执行（R）。转子前要保护现场（顺序执行的下一条指令地址进栈），子程序的最后一条指令应是 CRTN，从而保证恢复现场（顺序执行的下一条指令地址出栈）。

⑦ 指令 6：CJV（CONDITIONAL JUMP VECTOR）条件矢量跳转。根据测试条件，将子程序矢量地址装入微程序控制器。此时 VECT_bar 有效，表示中断或 DMA 操作等。

⑧ 指令 7：JRP（CONDITIONAL JUMP）条件跳转。根据条件，跳转到 D 或 R 指示的指令处执行。

⑨ 指令 8：RFCT（REPEAT LOOP, COUNTER≠ZERO）条件循环。若 R≠0，执行 R <- R-1 操作，继续执行循环操作，若 R=0，循环结束。循环操作指令的地址由堆栈获得。

⑩ 指令 9：RPCT（REPEAT PIPELINE REGISTER, COUNTER≠ZERO）条件循环。若 R≠0，执行 R <- R-1 操作，继续执行循环操作；若 R=0，循环结束。循环操作指令的地址由 R 获得。

⑪ 指令 10：CRTN（RETURN-FROM-SUBROUTINE）条件返回。通常与指令 CJS 和 JSRP 配对使用，用于子程序的条件返回。

⑫ 指令 11：CJPP（CONDITIONAL JUMP PIPELINE）条件跳转与堆栈 POP 操作。

⑬ 指令 12：LDCT（LOAD COUNTER AND CONTINUE）装入计数值并继续执行指令。

⑭ 指令 13：LOOP（TEST END-OF-LOOP）循环结束测试。

⑮ 指令 14：CONT（COUNTINUE）继续执行指令。

⑯ 指令 15：TWB（THREE-WAY BRANCH）三分支转移指令。可跳转到堆栈（F）、寄存器（R）或流水线（D）指示的地址。

从以上分析可以看出，指令译码器的结构较为复杂。但是，由于它仅仅是对输入指令的译码操作，是一个典型的组合逻辑系统，因此，只要遵守上述微指令译码表，它的描述不难写出。

2. 电路的 VHDL 描述方法及语法分析

电路的描述如下：

```
library 12901_lib;
use 12901_lib.types.all;
use 12901_lib.MVL7_functions.all;

entity control is
port (
            I : in MVL7_VECTOR(3 downto 0);       -- 指令码输入
      CCEN_bar: in MVL7;                          -- 外部条件
       CC_bar : in MVL7;                          -- 外部条件
     Rzero_bar: in MVL7;                          -- 零标志输入
       PL_bar : out MVL7;                         -- 标志输出
     VECT_bar : out MVL7;                         -- 标志输出
      MAP_bar : out MVL7;                         -- 标志输出
         R_sel: out MVL7;                         -- 寄存器选择控制信号输出
         D_sel: out MVL7;                         -- 数据选择控制信号输出
      uPC_sel : out MVL7;                         -- 微程序计数器控制信号输出
    stack_sel : out MVL7;                         -- 堆栈选择控制信号输出
         decr : out MVL7;                         -- 递减控制信号输出
         load : out MVL7;                         -- 预置控制信号输出
        clear : out MVL7;                         -- 堆栈控制信号输出
         push : out MVL7;                         -- 堆栈控制信号输出
         pop  : out MVL7                          -- 堆栈控制信号输出
      );
end control;

architecture control of control is
begin
ctrl:block              -- 无保护的块结构
signal fail : MVL7;
begin
fail <= CC_bar and not(CCEN_bar);     -- Fail 和 Pass 条件
D_sel <= '1' when ( I = "0010") or ( Rzero_bar = '1' and I = "1001") or
```

```
                  ( Rzero_bar = '0'  and fail = '1'  and I = "1111") or
                  ( ( fail = '0' ) and
                  ( ( I = "0001" ) or ( I = "0011" ) or
                  ( I = "0101" ) or ( I = "0110" ) or
                  ( I = "0111" ) or ( I = "1011" ) ) )
               else '0' ;
uPC_sel <= '1'  when ( I = "0100" ) or ( I = "1100" ) or ( I = "1110") or
           ( ( fail = '1' ) and ( ( I = "0001" ) or ( I = "0011" ) or
           ( I = "0110" ) or ( I = "1010" ) or
           ( I = "1011" ) or ( I = "1110" ) ) ) or
           ( ( Rzero_bar = '0' ) and ( ( I = "1000" ) or
           ( I = "1001") ) ) or ( (fail = '0' ) and
           ( ( I = "1111" ) or ( I = "1101") ) )
    else '0' ;
stack_sel <= '1'  when ( Rzero_bar = '1'  and I = "1000") or
                    ( fail = '0'  and I = "1010") or
                    ( fail = '1'  and I = "1101") or
                    ( Rzero_bar = '1'  and fail = '1'  and I = "1111")
               else '0' ;
R_sel <= '1'      when (( fail = '1' ) and (( I = "0101" ) or ( I = "0111")))
               else '0' ;
push <= '1'       when ( (fail = '0' ) and ( I = "0001") ) or
                  ( I = "0100" ) or ( I = "0101")
               else '0' ;
pop <= '1'     when ( (fail = '0' ) and ( ( I = "1010" ) or ( I = "1011" ) or
                  ( I = "1101" ) or ( I = "1111" ) )) or
                  ( (Rzero_bar = '0' ) and ( (I = "1000" ) or ( I = "1111") ) )
               else  '0' ;
load <= '1'       when ( (I = "1100") or ( I = "0100" and fail = '0' ))
               else '0' ;
decr <= '1'       when ((Rzero_bar = '1' ) and ( (I = "1000" ) or
                  ( I = "1001" ) or ( I = "1111") ))
               else  '0' ;
MAP_bar <= '0'  when I = "0010"  else '1' ;
VECT_bar <= '0'  when I = "0110" else '1' ;
PL_bar <= '1'  when ( I = "0010" ) or ( I = "0110") else '0' ;
clear <= '1'  when I = "0000" else '0' ;

end block ctrl;

end control;
```

3．模拟测试向量的选择及模拟结果分析

本例描述的结构较为复杂，测试向量集也相应较大。因为对应输入变量的每一种组合（4个指令输入信号，一个零标志位和两个外部条件信号，最多有 2^7=128 种组合），所要考虑的输出状态也较多（最多为12个）。当然，在测试中应尽量排除那些不可能出现的情况，以减少测试向量的数量。

下面对部分测试向量进行分析。

```
-- ***************************
-- * I = 0          系统复位指令 *
-- ***************************

I <= "0000";                -- 指令码
CCEN_bar <= '0';            -- 其他条件
CC_bar <= '1';
Rzero_bar <= '1';           — 零标志位

wait for 1 ns;              -- 有关输出命令及标志位验证
assert (PL_bar = '0')
    report "Assert 0 : < PL_bar /= 0 >" severity warning;
assert (VECT_bar = '1')
    report "Assert 1 : < VECT_bar /= 1 >" severity warning;
assert (MAP_bar = '1')
    report "Assert 2 : < MAP_bar /= 1 >" severity warning;
assert (R_sel = '0')
    report "Assert 3 : < R_sel /= 0 >" severity warning;
assert (D_sel = '0')
    report "Assert 4 : < D_sel /= 0 >" severity warning;
assert (uPC_sel = '0')
    report "Assert 5 : < uPC_sel /= 0 >" severity warning;
assert (stack_sel = '0')
    report "Assert 6 : < stack_sel /= 0 >" severity warning;
assert (decr = '0')
    report "Assert 7 : < decr /= 0 >" severity warning;
assert (load = '0')
    report "Assert 8 : < load /= 0 >" severity warning;
assert (clear = '1')
    report "Assert 9 : < clear /= 1 >" severity warning;
assert (push = '0')
    report "Assert 10 : < push /= 0 >" severity warning;
assert (pop = '0')
    report "Assert 11 : < pop /= 0 >" severity warning;
```

```
wait for 1 ns;

-- ******************************
-- * I = 1    子程序条件转移指令 *
-- ******************************
I <= "0001";            -- 指令码
CCEN_bar <= '0' ;       -- Fail 状态
CC_bar <= '1' ;
Rzero_bar <= '1' ;      -- 本指令与此标志无关

wait for 1 ns;                -- 验证有关输出信号
assert (PL_bar = '0' ) -- 有效
    report "Assert 96 : < PL_bar /= 0 >" severity warning;
assert (VECT_bar = '1' )
    report "Assert 97 : < VECT_bar /= 1 >" severity warning;
assert (MAP_bar = '1' )
    report "Assert 98 : < MAP_bar /= 1 >" severity warning;
assert (R_sel = '0' )
    report "Assert 99 : < R_sel /= 0 >" severity warning;
assert (D sel = '0' )
    report "Assert 100 : < D_sel /= 0 >" severity warning;
assert (uPC_sel = '1' )     -- 有效
    report "Assert 101 : < uPC_sel /= 1 >" severity warning;
assert (stack_sel = '0' )
    report "Assert 102 : < stack_sel /= 0 >" severity warning;
assert (decr = '0' )
    report "Assert 103 : < decr /= 0 >" severity warning;
assert (load = '0' )
    report "Assert 104 : < load /= 0 >" severity warning;
assert (clear = '0' )
    report "Assert 105 : < clear /= 0 >" severity warning;
assert (push = '0' )
    report "Assert 106 : < push /= 0 >" severity warning;
assert (pop = '0' )
    report "Assert 107 : < pop /= 0 >" severity warning;
wait for 1 ns;

I <= "0001";        -- 指令码
CCEN_bar <= '0' ; -- Pass 状态
CC_bar <= '0' ;
Rzero_bar <= '1' ;      -- 无关信号
```

```
wait for 1 ns; --  验证输出信号
assert (PL_bar = '0') -- 有效
    report "Assert 120 : < PL_bar /= 0 >"  severity warning;
assert (VECT_bar = '1')
    report "Assert 121 : < VECT_bar /= 1 >"  severity warning;
assert (MAP_bar = '1')
    report "Assert 122 : < MAP_bar /= 1 >"  severity warning;
assert (R_sel = '0')
    report "Assert 123 : < R_sel /= 0 >"  severity warning;
assert (D_sel = '1')        -- 有效
    report "Assert 124 : < D_sel /= 1 >"  severity warning;
assert (uPC_sel = '0')
    report "Assert 125 : < uPC_sel /= 0 >"  severity warning;
assert (stack_sel = '0')
    report "Assert 126 : < stack_sel /= 0 >"  severity warning;
assert (decr = '0')
    report "Assert 127 : < decr /= 0 >"  severity warning;
assert (load = '0')
    report "Assert 128 : < load /= 0 >"  severity warning;
assert (clear = '0')
    report "Assert 129 : < clear /= 0 >"  severity warning;
assert (push = '1')         -- 有效
    report "Assert 130 : < push /= 1 >"  severity warning;
assert (pop = '0')
    report "Assert 131 : < pop /= 0 >"  severity warning;
wait for 1 ns;

-- *****************
-- * I = 8   指令 8 *
-- *****************
---------------------------
I <= "1000";              -- 指令码 (条件循环)
CCEN_bar <= '0';          -- 与 Pass 或 Fail 无关
CC_bar <= '1';
Rzero_bar <= '1';         -- 有关信号

wait for 1 ns;            -- 验证第 1 种情况 (Rzero_bar = '1')
assert (PL_bar = '0') -- 有效
    report "Assert 768 : < PL_bar /= 0 >" severity warning;
assert (VECT_bar = '1')
    report "Assert 769 : < VECT_bar /= 1 >"  severity warning;
assert (MAP_bar = '1')
```

```
    report  "Assert 770 : < MAP_bar /= 1 >"  severity warning;
assert (R_sel = ' 0' )
    report  "Assert 771 : < R_sel /= 0 >"  severity warning;
assert (D_sel = ' 0' )
    report  "Assert 772 : < D_sel /= 0 >"  severity warning;
assert (uPC_sel = ' 0' )
    report  "Assert 773 : < uPC_sel /= 0 >"  severity warning;
assert (stack_sel = ' 1' )        -- 有效
    report  "Assert 774 : < stack_sel /= 1 >"  severity warning;
assert (decr = ' 1' )             -- 有效
    report  "Assert 775 : < decr /= 1 >"  severity warning;
assert (load = ' 0' )
    report  "Assert 776 : < load /= 0 >"  severity warning;
assert (clear = ' 0' )
    report  "Assert 777 : < clear /= 0 >"  severity warning;
assert (push = ' 0' )
    report  "Assert 778 : < push /= 0 >"  severity warning;
assert (pop = ' 0' )
    report  "Assert 779 : < pop /= 0 >"  severity warning;
wait for 1 ns;

I <= "1000";          -- 指令码
CCEN_bar <= ' 0' ;
CC_bar <= ' 1' ;
Rzero_bar <= ' 0' ;      -- 第 2 种情况

wait for 1 ns;
assert (PL_bar = ' 0' )      -- 有效
    report  "Assert 780 : < PL_bar /= 0 >"  severity warning;
assert (VECT_bar = ' 1' )
    report  "Assert 781 : < VECT_bar /= 1 >"  severity warning;
assert (MAP_bar = ' 1' )
    report  "Assert 782 : < MAP_bar /= 1 >"  severity warning;
assert (R_sel = ' 0' )
    report  "Assert 783 : < R_sel /= 0 >"  severity warning;
assert (D_sel = ' 0' )
    report  "Assert 784 : < D_sel /= 0 >"  severity warning;
assert (uPC_sel = ' 1' )     -- 有效
    report  "Assert 785 : < uPC_sel /= 1 >"  severity warning;
assert (stack_sel = ' 0' )
    report  "Assert 786 : < stack_sel /= 0 >"  severity warning;
assert (decr = ' 0' )
```

```
        report  "Assert 787 : < decr /= 0 >"  severity warning;
    assert (load = '0')
        report  "Assert 788 : < load /= 0 >"  severity warning;
    assert (clear = '0')
        report  "Assert 789 : < clear /= 0 >"  severity warning;
    assert (push = '0')
        report  "Assert 790 : < push /= 0 >"  severity warning;
    assert (pop = '1')          -- 有效
        report  "Assert 791 : < pop /= 1 >"  severity warning;
    wait for 1 ns;
```

图 87.1 为模拟结果的波形图，其中 i3~0 = i[3]i[2]i[1]i[0]，从图上可以看出，模拟结果与表 87.1 的定义一致，表示描述是正确的。

和 Am2901 一样，第 83 例到第 87 例的 Am2910 也是一个复杂的数字系统，采用分模块的方法，对减小系统描述的复杂性是有效的。

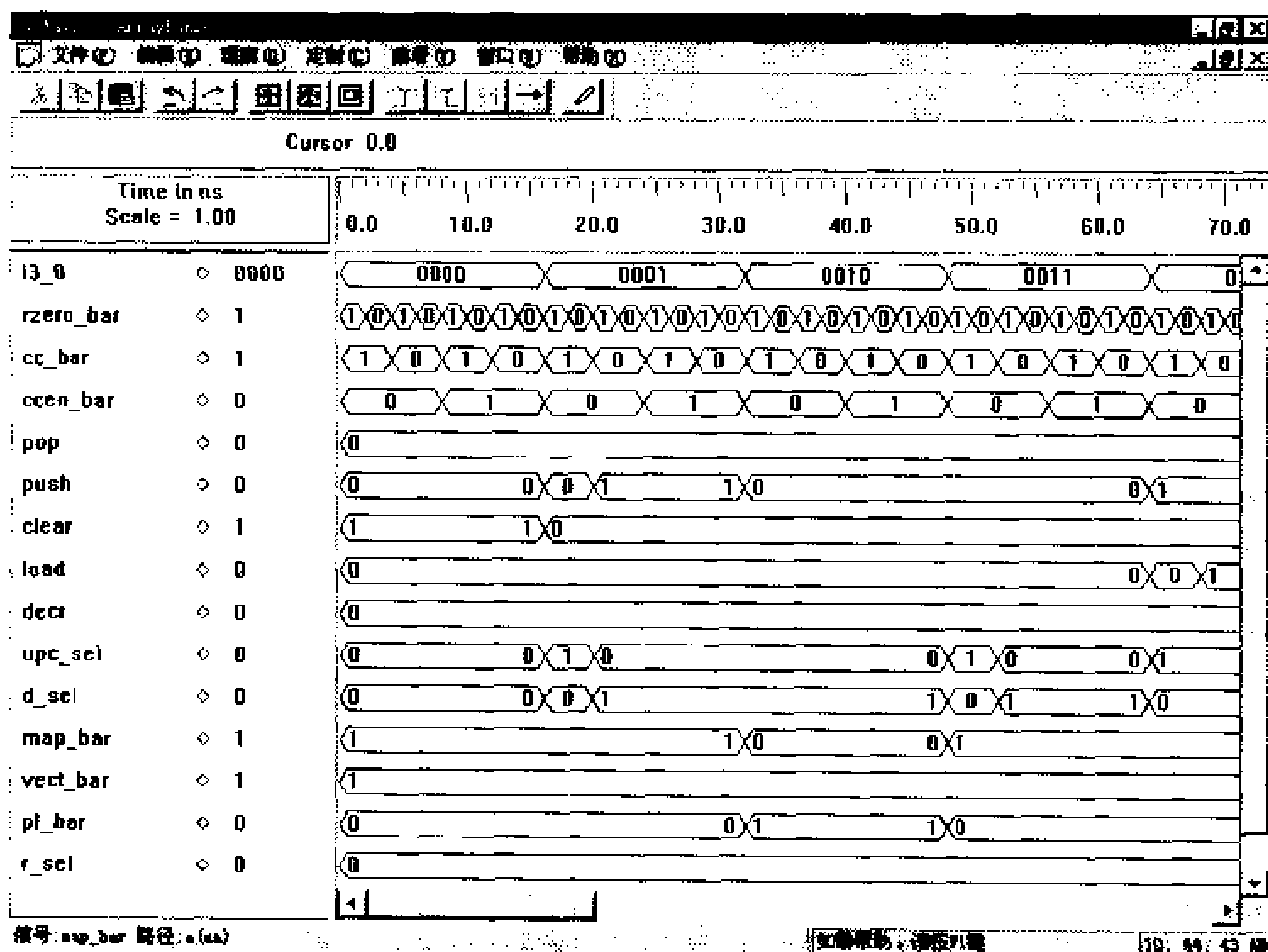


图 87.1　Am2910 指令译码器模拟波形图

（源描述文件名：87_control.vhd
测试平台文件名：87_control_stim.vhd）

第 88 例　可控制计数器

韩　曙

1. 电路系统工作原理

本例描述一个可控制计数器，其示意图如图 88.1 所示。其中，Data3~0 为计数值预置输入端，Y3~0 为计数输出端。Con1~0 为控制代码，Strb 为选通信号，Clk 表示计数时钟。

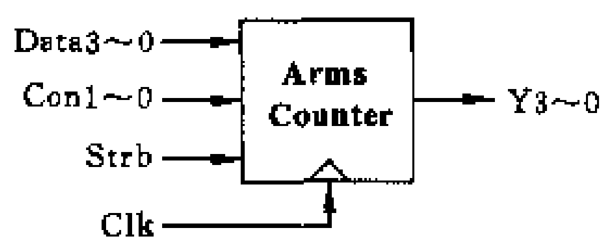


图 88.1　Arms counter 示意图

根据对 Arms counter 的功能描述，其结构框图如图 88.2 所示。分成条件寄存器（condition register）、条件译码器（condition decoder）、上下限比较寄存器（limit register）、计数器（counter）、比较器（comparator）及控制部分（control）等相对独立的部分。下面分别介绍各部分的功能。

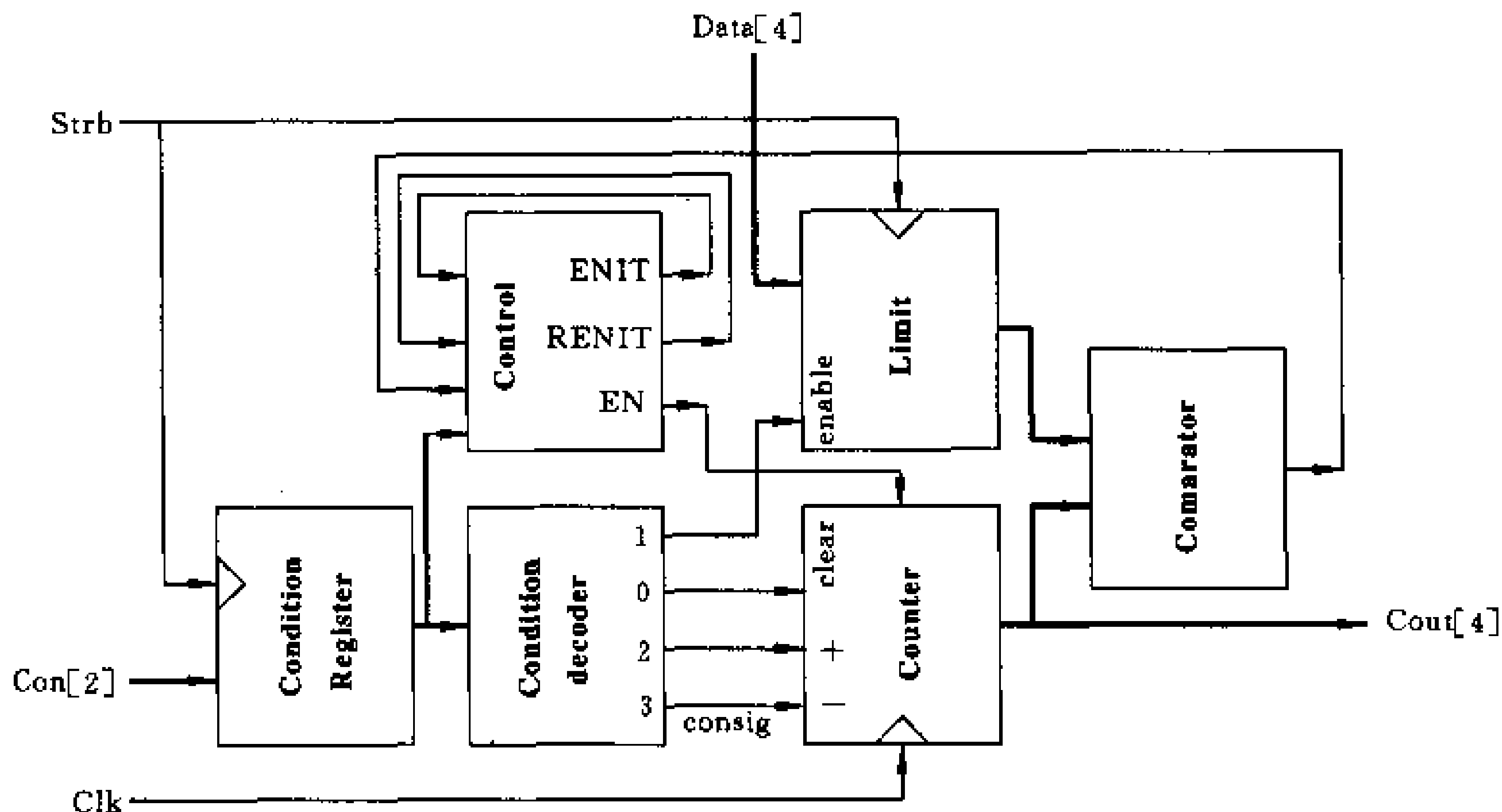


图 88.2　Arms counter 结构框图

(1) 条件寄存器和条件译码器

条件寄存器（Con）在选通信号 Strb 的作用下保存外部输入条件 Con1~0，经条件译码器（ConSig）译码后产生控制信号。控制信号的作用见表 88.1。

表 88.1　条件寄存器与译码器

Condition register			Condition Decoder		
Con1~0			ConSig3~0		
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

表 88.2　计数器工作方式

计数条件	计数方式
ConSig(0)上跳沿	清 0
EN= '1'	使能
EN='0'	停止计数
ConSig(2)='1'	加计数
ConSig(3)='1'	减计数

(2) 比较上下限寄存器

比较上下限寄存器用来存放计数器计数值的上限（加计数）和下限（减计数）。当控制信号 ConSig(1) 有效时，在 Strb 上升沿将上／下限值预先置入。

(3) 计数器

在控制信号作用下，进行加/减计数，计数脉冲为 Clk，计数条件及计数方式见表 88.2。

(4) 比较器

计数器当前计数值与比较限寄存器的值相等时，控制计数器停止计数。

(5) 控制器

根据输入条件 Con 和计数器的当前状态，发出有关的控制命令，包括：

① Con="10" 或 Con="11"时，使 Enit 置 1；

② Enit 的上升沿置位 En 和 REnit，其他时刻使 REnit 置 0；

③ REnit 的上升沿复位 Enit；

④ En 的上升沿使能计数器，在计数期间保持有效值'1'；

⑤ 计数器当前计数值和计数限值相等时（CNT=LIM）复位 En，计数器停止计数。

2. 电路的 VHDL 语言描述方法及语法分析

根据前面的讨论，在描述中分成 4 个相对独立的部分。而将控制信号分别插在这 4 个部分中。下面结合 VHDL 描述分别说明它们的描述特点。

```
use work.BIT_FUNCTIONS.all;
entity ARMS_COUNTER is
  port (
        Clk    : in bit;                          -- 计数时钟脉冲
        Strb   : in bit;                          -- 使能
        Con    : in bit_vector(1 downto 0);       -- 计数条件
        Data   : in bit_vector(3 downto 0);       -- 计数限值
        COUT   : out bit_vector(3 downto 0));     -- 计数值输出
end ARMS_COUNTER;

architecture ARMS_COUNTER of ARMS_COUNTER is
   signal ENIT, RENIT: bit;
   signal EN: bit;
   signal ConSIG, LIM: bit_vector(3 downto 0);
   signal CNT : bit_vector(3 downto 0);

begin

-- 译码部分
-- 敏感信号位 Strb, RENIT
DECODE: process (Strb, RENIT)
variable ConREG: BIT_VECTOR(1 downto 0) := "00";                  -- 内部变量
begin
  if (Strb = '1') and (not Strb'STABLE) then                      -- Strb 上升沿触发
    ConREG := Con;
    case ConREG is      -- 条件译码
         when "00" => ConSIG <= "0001";                           -- 计数器清零
         when "01" => ConSIG <= "0010";                           -- 计数限预置
         when "10" => ConSIG <= "0100"; ENIT <= '1';              -- 加计数
         when "11" => ConSIG <= "1000"; ENIT <= '1';              -- 减计数
         when others =>
    end case;
  end if;
  if (RENIT = '1') and (not RENIT'STABLE) then                    -- 控制信号
    ENIT <= '0';
  end if;
end process DECODE;

-- 计数限预置
LOAD_LIMIT: process (Strb)
```

```
    -- 预置条件： ConSIG(1)=' 1'  且 Strb 下跳
begin
  if (ConSIG(1) = ' 1' ) and (not Strb' STABLE) and (Strb = ' 0' ) then
       LIM <= Data;              — 外部数据置入
  end if;
end process LOAD_LIMIT;

-- 计数
CTR: process (ConSIG(0), EN, Clk)
variable CNTE : BIT := ' 0' ;
begin
  if (ConSIG(0) = ' 1' ) and (not ConSIG(0)' STABLE) then
    CNT <= "0000";       -- 清零
  end if;
  if (not EN' STABLE) then
    if (EN = ' 1' ) then
      CNTE :- ' 1' ;
    else
      CNTE := ' 0' ;
    end if;
  end if;
  if (not Clk' STABLE) and (Clk = ' 1' ) and (CNTE = ' 1' ) then
    if (ConSIG(2) = ' 1' ) then
      CNT <= CNT + "0001";         -- 加计数
    elsif (ConSIG(3) = ' 1' ) then
          CNT <= CNT - "0001";   -- 减计数
    end if;
  end if;
end process CTR;

-- 比较
LIMIT_CHK: process (CNT, ENIT)
begin
  if (not ENIT' STABLE) then
    if (ENIT = ' 1' ) then
      EN <= ' 1' ; RENIT <= ' 1' ;
    else
      RENIT <= ' 0' ;
    end if;
  end if;
```

```
    if (EN = '1') and (CNT = LIM) then  -- 计数值与比较限值是否相等
      EN <= '0';
    end if;

  end process LIMIT_CHK;
  COUT <= CNT;          -- 计数值输出
  end ARMS_COUNTER;
```

3. 模拟测试向量的选择及模拟结果分析

本例中有 4 个并行的进程，各个进程的敏感信号又不完全相同。但电路的主要功能是计数，生成的测试信号要以验证计数功能是否正确为主。在测试主要功能的同时，兼顾对其他信号的测试。

```
--在构造测试向量时首先定义计数时钟信号。  时钟周期为 50 ns。本描述中的断言语
--句都缺省了 severity 子句部分，根据 VHDL 语法，这种情况下取其缺省值 error。
process
    begin
        wait for 1 ns;
        Clk <= transport '0';
        wait for 49 ns;
        Clk <= transport '1';
    end process;
    -- 计数测试
process
begin
    wait for 30 ns;
    -- 测试 1 ： 测试复位、递增、递减计数、计数限预置
    -- 复位测试
    Con <= "00";
    Strb <= '1' after 10 ns, '0' after 20 ns;        -- 第 1 个时钟周期
    wait for 50 ns;                                  -- 第 2 个时钟周期
    assert (COUT="0000") report "ERROR1: COUT not reset to 0";

    -- 预置测试
    Data <= "0010";    -- 预置条件
    Con <= "01";
    Strb <= '1' after 10 ns, '0' after 20 ns;
    wait for 50 ns;                                  -- 第 3 个时钟周期
```

```
Con <= "10";  -- 加计数条件
Strb <= '1' after 10 ns, '0' after 20 ns;
-- 递增计数
wait for 50 ns;                                    -- 第 4 个时钟周期
assert (COUT="0001") report "ERROR2: COUT not incremented to 1";
-- 继续递增计数
wait for 50 ns;                                    -- 第 5 个时钟周期
assert (COUT="0010") report "ERROR3: COUT not incremented to 2";
-- 停止递增，因为当前计数值与计数限值相等。
wait for 50 ns;                                    -- 第 6 个时钟周期
assert (COUT="0010") report "ERROR4: COUT should have hit limit at 2";
-- 再次触发加计数条件
Con <= "10";
Strb <= '1' after 10 ns, '0' after 20 ns;
-- 仍然不会递增，因为当前计数值与计数限值相等。
wait for 50 ns;                                    -- 第 7 个时钟周期
assert (COUT="0010") report "ERROR5: COUT should have hit limit at 2";

-- 触发减计数条件
Con <= "11";  -- 递减计数条件
Strb <= '1' after 10 ns, '0' after 20 ns;
-- 此时也不会递减，因为当前计数值和计数限值相等。
wait for 50 ns;                                    -- 第 8 个时钟周期
assert (COUT="0010") report "ERROR6: COUT not decremented to 2";
-- 预置新的计数限 (0)
Data <= "0000";
Con <= "01";
Strb <= '1' after 10 ns, '0' after 20 ns;
wait for 50 ns;                                    -- 第 9 个时钟周期

-- 递减计数
Con <= "11";        递减计数条件
Strb <= '1' after 10 ns, '0' after 20 ns;
wait for 50 ns;                                    -- 第 10 个时钟周期
-- 开始递减
assert (COUT="0001") report "ERROR7: COUT not decremented to 0";
```

前 10 个时钟周期的模拟波形如图 88.3 所示。根据前面的分析可以看出，模拟结果和描述一致。

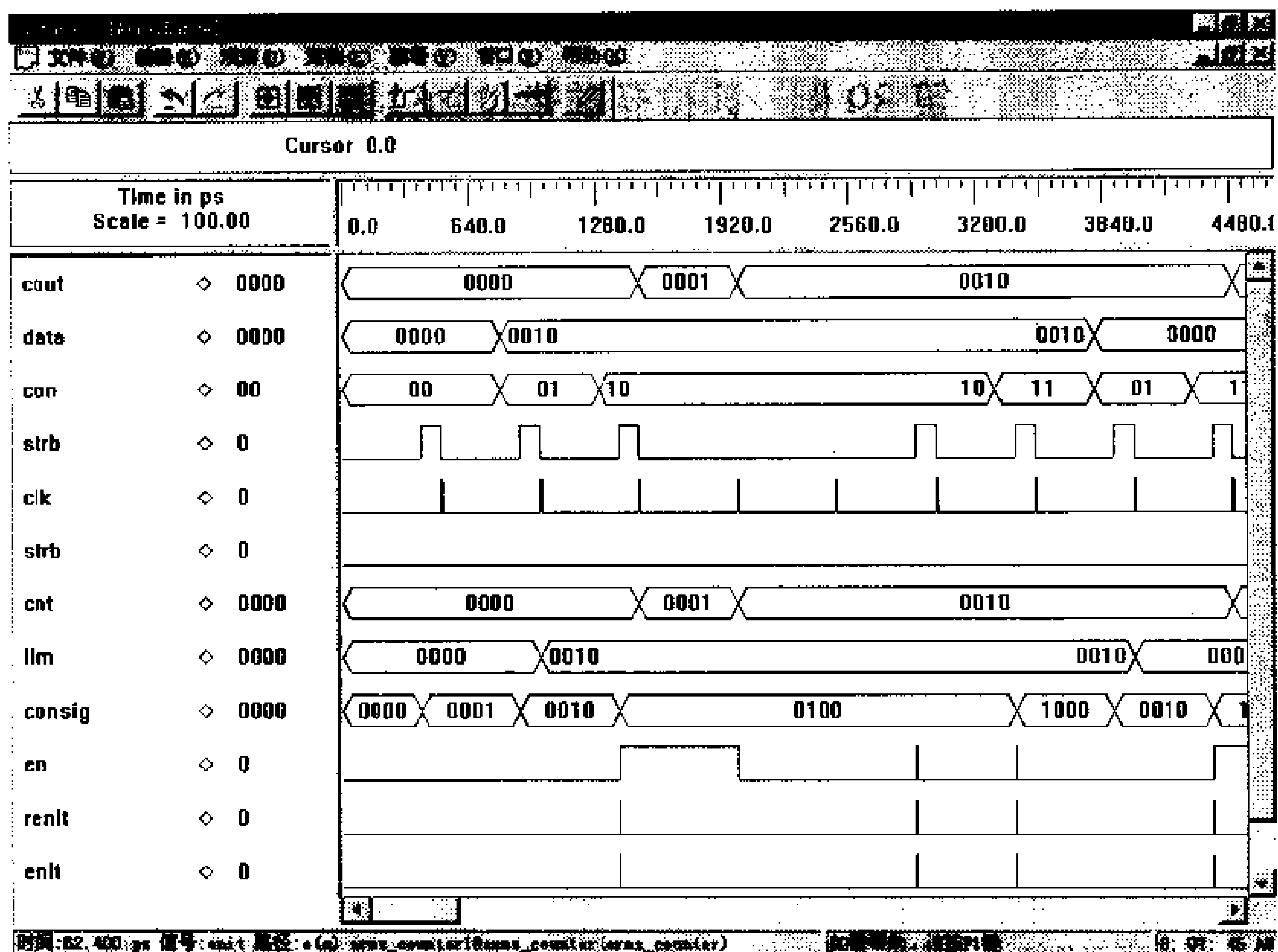


图 88.3　部分模拟结果

（源描述文件名：88_arms_counter.vhd
测试平台文件名：88_arms_counter_stim.vhd
88_pack_2_0.vhd）

第 89 例　四位超前进位加法器

韩　曙

1. 电路系统工作原理

四位超前进位加法器的结构框图如图 89.1 所示。

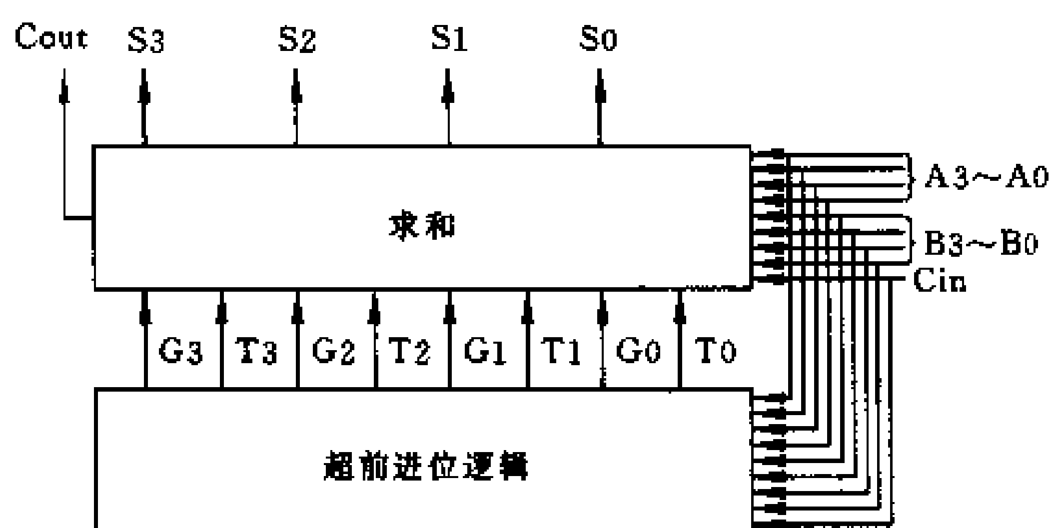


图 89.1　超前进位加法器结构框图

超前进位加法器的逻辑关系中使用了进位生成和进位传输逻辑，我们可以从全加器逻辑推出它的逻辑关系。设 A0，B0 为两个二进制数，Cin 为低位进位信号，生成的本位和 S0 与高位的进位 C0 分别为

$$C0 = A0 \cdot B0 + (A0 \oplus B0) \cdot Cin$$

$$S0 = A0 \oplus B0 \oplus Cin$$

设 $T0=A0 \oplus B0$，$G0=A0 \cdot B0$，分别称为进位传递逻辑和进位生成逻辑，则上式可改写为

$$C0 = G0 + T0 \cdot Cin$$

$$S0 = T0 \oplus Cin$$

由此可递推出高位的逻辑关系为

$$C1=G1+T1C0=G1+T1 \cdot G0+T1 \cdot T0 \cdot Cin$$

$$S1=T1 \oplus C0$$

$$C2=G2+T2 \cdot C1=G2+T2 \cdot G1+T2 \cdot T1 \cdot G0+T2 \cdot T1 \cdot T0 \cdot Cin$$

$$S2=T2 \oplus C1$$

$$C3=Cout=G3+T3 \cdot C2=G3+T3 \cdot G2+T3 \cdot T2 \cdot G1+T3 \cdot T2 \cdot T1 \cdot G0+T3 \cdot T2 \cdot T1 \cdot T0 \cdot Cin$$

$$S3=T3 \oplus C2$$

电路的 VHDL 描述基于上述书写的表达式。

2. 电路的 VHDL 描述方法及语法分析

```
use work.bit_FUNCTIONS.all;

entity FULL_ADDER is
  port (
        A    : in bit_vector(3 downto 0);
        B    : in bit_vector(3 downto 0);
        Cin  : in bit;
        S    : out bit_vector(3 downto 0);
        Cout : out bit);
    end FULL_ADDER;

architecture FULL_ADDER of FULL_ADDER is
   signal sA , sB ,sS : bit_vector(3 downto 0);
   signal sCin , sCout : bit   ;
   signal sC : bit_vector(3 downto 0)  ;
   signal sT : bit_vector(3 downto 0)  ;
   signal sG : bit_vector(3 downto 0)  ;
begin
   sA <= A ;
   sB <= B ;
   sCin   <=  Cin ;
   sT(0) <=  sA(0) xor sB(0)  ;             -- 进位传递逻辑和进位生成逻辑
   sG(0) <=  sA(0) and sB(0)  ;
   sT(1) <=  sA(1) xor sB(1)  ;
   sG(1) <=  sA(1) and sB(1)  ;
   sT(2) <=  sA(2) xor sB(2)  ;
   sG(2) <=  sA(2) and sB(2)  ;
   sT(3) <=  sA(3) xor sB(3)  ;
   sG(3) <=  sA(3) xor sB(3)  ;
   sC(0) <=  sG(0) or sT(0) and sCin ;     -- 超前进位逻辑
   sC(1) <=  sG(1) or sT(1) and (sG(0) or sT(0) and sCin)  ;
   sC(2)<= sG(2) or sT(2) and (sG(1) or sT(1) and (sG(0) or sT(0) and sCin));
   sC(3)<= sG(3) or sT(3) and (sG(2) or sT(2) and (sG(1) or sT(1)
            and (sG(0) or sT(0) and sCin)));
   sS(0) <= sT(0) xor sCin  ;          -- 求和逻辑
   sS(1) <= sT(1) xor sC(0)  ;
   sS(2) <= sT(2) xor sC(1)  ;
   sS(3) <= sT(3) xor sC(2)  ;
```

```
  S <= sS ;                                   输出
  Cout <= sC(3) ;
end ;
```

3. 模拟测试向量的选择及模拟结果分析

本电路的测试向量采用类似列真值表的方法，输入信号共有 512（2^9）种不同的组合。但是测试描述中没有在每一种输入组合后都加入断言语句来判断结果是否正确，因此在模拟时要通过设置断点或单步执行，以控制模拟过程。

```
-- 4位超前进位加法器模拟
entity E is
end;

architecture AAA of E is
  component FULL_Adder
    port (
        A    : in bit_vector(3 downto 0);
        B    : in bit_vector(3 downto 0);
        Cin  : in bit;
        S    : out bit_vector(3 downto 0);
        Cout : out bit);
  end component;
  signal sA , sB , sS : bit_vector(3 downto 0):= "0000" ;
  signal sCin , sCout : bit := '0'    ;
  signal sC : bit_vector(3 downto 0) := "0000" ;
  signal sT : bit_vector(3 downto 0) := "0000" ;
  signal sG : bit_vector(3 downto 0) := "0000" ;
  for all : FULL_ADDER use entity work.FULL_ADDER(FULL_ADDER) ;
begin
  FULL_ADDER1 : FULL_ADDER
 port map (sA, sB, sCin, sS, sCout);
process
   begin
        sCin  <= not sCin  after 5  ns ;
        sA(0) <= not sA(0) after 10 ns ;
        sA(1) <= not sA(1) after 20 ns ;
        sA(2) <= not sA(2) after 40 ns ;
        sA(3) <= not sA(3) after 80 ns ;
        sB(0) <= not sB(0) after 160 ns ;
        sB(1) <= not sB(1) after 320 ns ;
        sB(2) <= not sB(2) after 640 ns ;
```

```
        sB(3) <= not sB(3) after 1280 ns ;
    -- 将变化频率最大的信号作为敏感信号。
        wait on sCin  ;
    end process ;
end AAA;
```

图 89.2 所示为部分模拟波形。

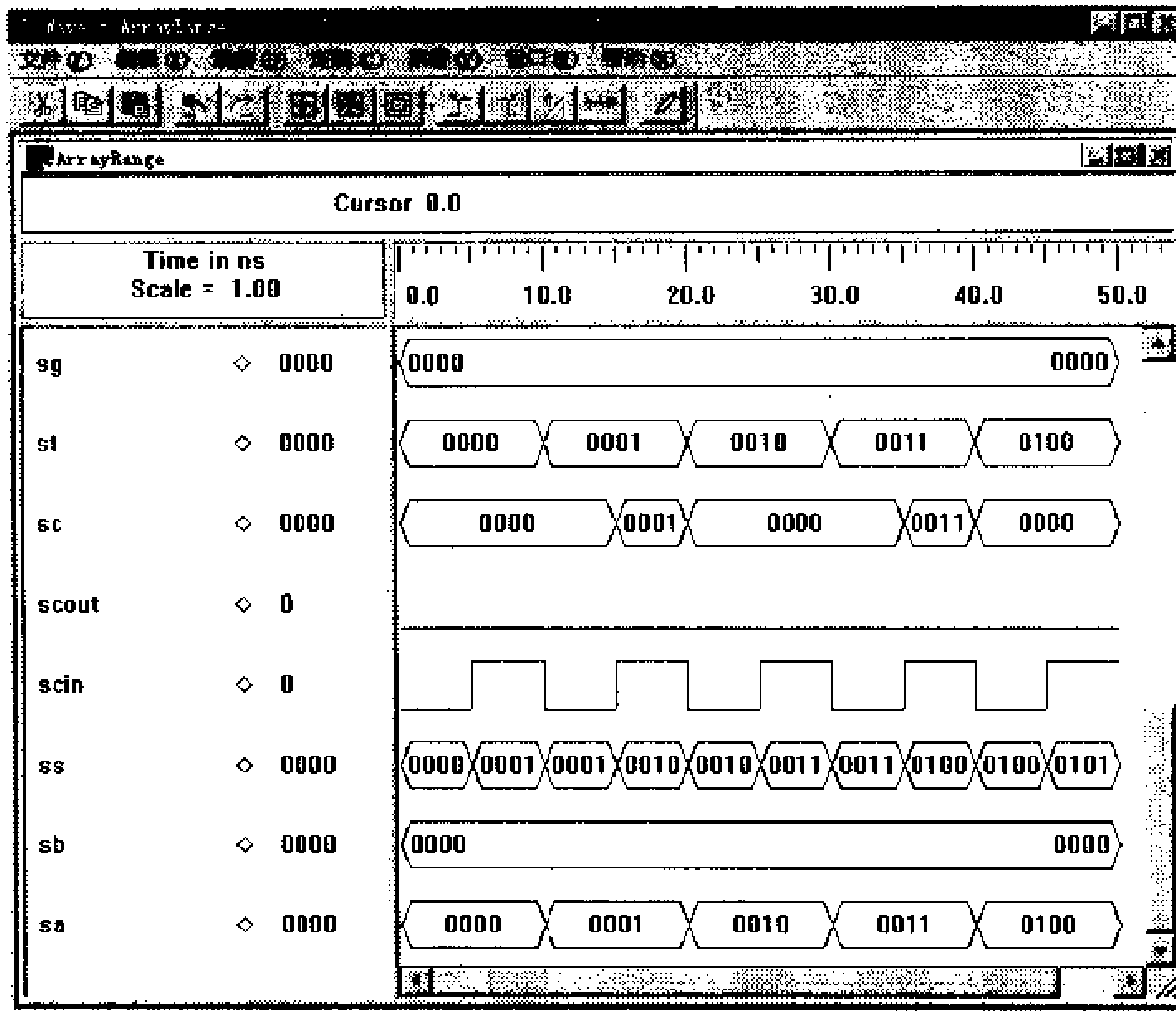


图 89.2　超前进位加法器的模拟波形

从图中可以看出，进位输入端的信号值 scin 变化周期为 10ns，输入信号 sa 从 0000 递增，递增的周期也为 10ns，输入信号 sb 在本图中的 50ns 范围内未发生变化，所以，输出信号 ss 实际上仅为 sa 和 scin 之和，而 scout 在 0~50ns 内始终为 0，表示无进位。

（源描述文件名：89_full_adder.vhd
测试平台文件名：89_full_adder_stim.vhd
89_pack_2_0.vhd）

第 90 例　实现窗口搜索算法的并行系统（1）——协同处理器

李　杰

1．电路系统工作原理

基于窗口搜索算法的系统（window searching algorithm based system, 简称 WSS）是一个比较常用的应用程序，其目的是从字符串 string 中查找一个与特定字符序列 sequence 最接近的序列。为了方便讨论，这里假设 string 的长度为 23，sequence 的长度为 8。

算法的流程如图 90.1 所示。在初始化过程完成之后，首先装入 sequence，string 通过 n 个 8 位字符的 burst 分别装入，一共需要 n-1 次循环。在装入了相邻的两个 burst 之后，开始搜索过程。

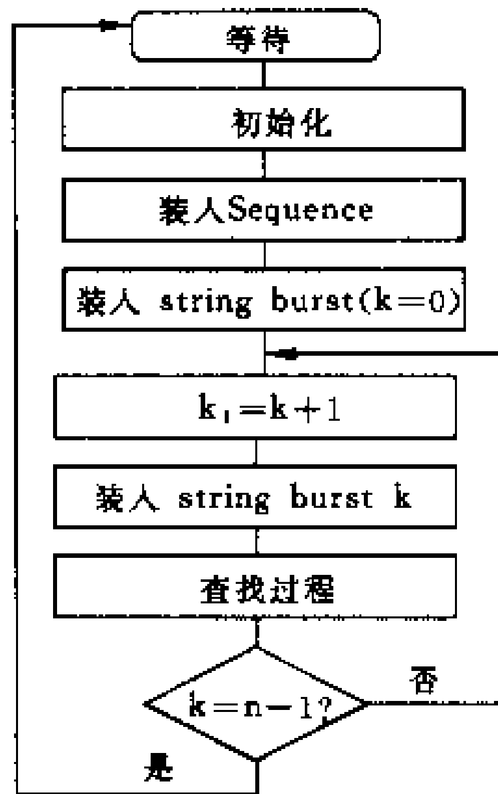


图 90.1　搜索算法

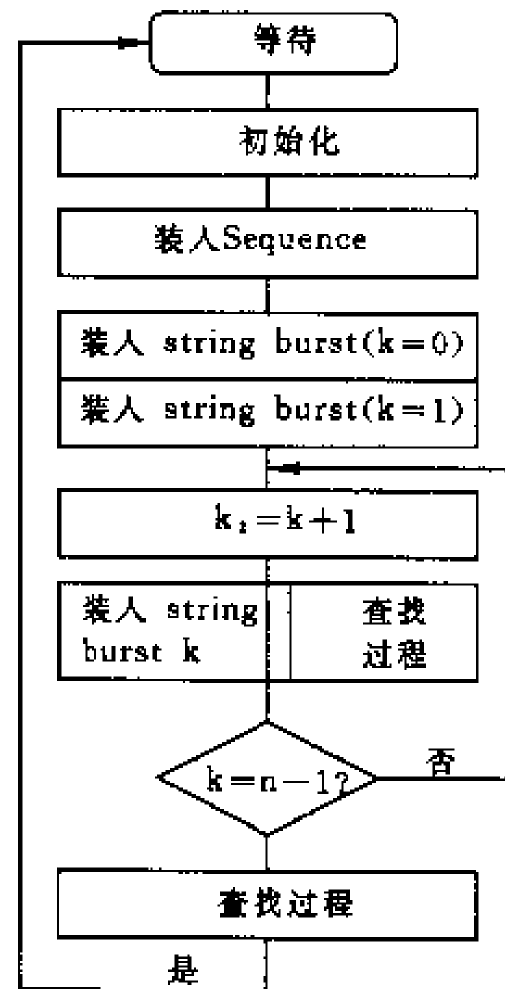


图 90.2　改进的搜索算法

这里的查找过程是将读入的 string burst 中的 8 个字符与 sequence 中的 8 个字符分别相减，再将得到的差值进行累加以反映两者之间的差异程度。差值和最小的那一组 burst 与 sequence 最为相似。

从图 90.1 中可以看出，可能存在两种类型的并行机制：其一是两个存储器的装入操

作，其二是本次搜索过程和待处理的下一个 burst 的装入。由于本系统中仅用到了一条数据总线，因此不可能并行地对两个存储器进行装入操作。但是当字符串存储器采用双端 RAM 时，可以实现后一种并行机制。改进后的算法如图 90.2 所示。

根据上面的算法说明，WSS 至少可以划分为 4 个部分，其中包括两个封装的内存 mem_string 和 mem_sequence，还包括一个执行搜索算法的协同处理器 co_processor，此外，还需要一个代表整个系统的顶层控制器。我们将在第 90 例~第 93 例中对各部分做系统介绍，并分析其 VHDL 描述的特点。

在本例中将首先讨论其中的协同处理器。该处理器执行前面所述的比较过程，与其他各个部分之间的连接关系如图 90.3 所示。

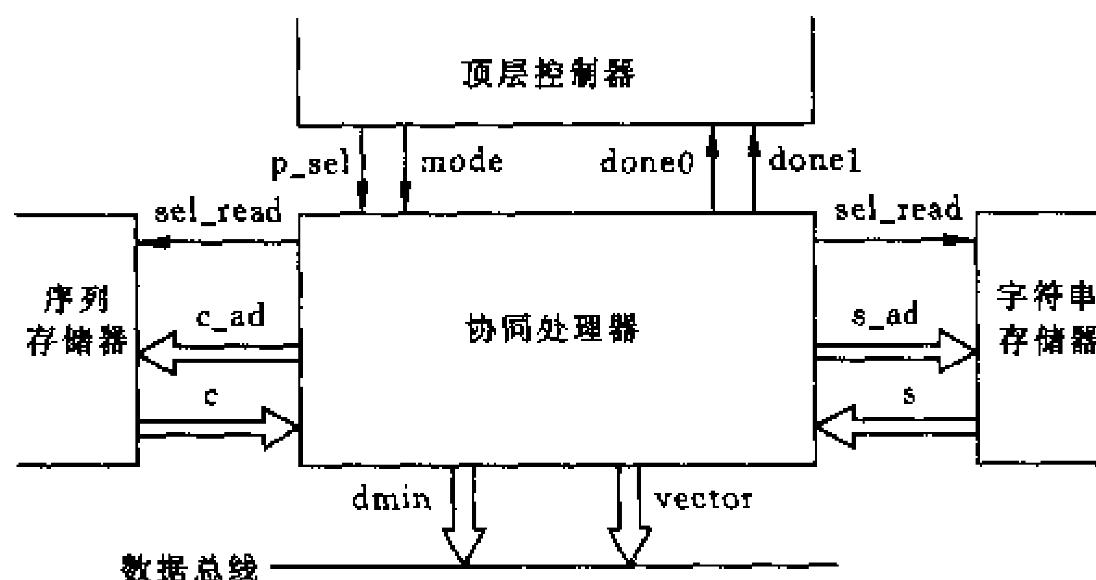


图 90.3　协同处理器与其他各个部分之间的关系

协同处理器由 p_sel 信号和 mode 信号激活，其中 p_sel 选中协同处理器，而 mode 信号则选定其工作方式。根据 string 中正在处理字符的位置、协同处理器具有两种工作方式，因为最后一次循环中需要执行一些特殊的操作，mode 为 1，在计算过程的最后，将 done0 和 done1 置位；除此之外，mode 为 0，并在计算过程的最后将 done0 置位。

协同处理器的抽象结构如图 90.4 所示。有关其中各个函数调用以及操作的具体内容，请参见后续的语言描述及语法分析。

2. VHDL 语言描述及语法分析

首先需要说明的是赋值语句的时序限制问题。从存储器之间的通信关系可以看出，输入信号在时钟上升沿应该是稳定的，但是这种时序限制在无定时的行为模型中很难说明，因为所有的赋值操作需要的延时为零，所以在行为模拟期间所有的信号赋值发生在时钟的上升沿。为了解决这个问题，所有的存储器输入赋值均使用 **after** 语句进行延时（**after** 2 ns）。以下是描述程序。

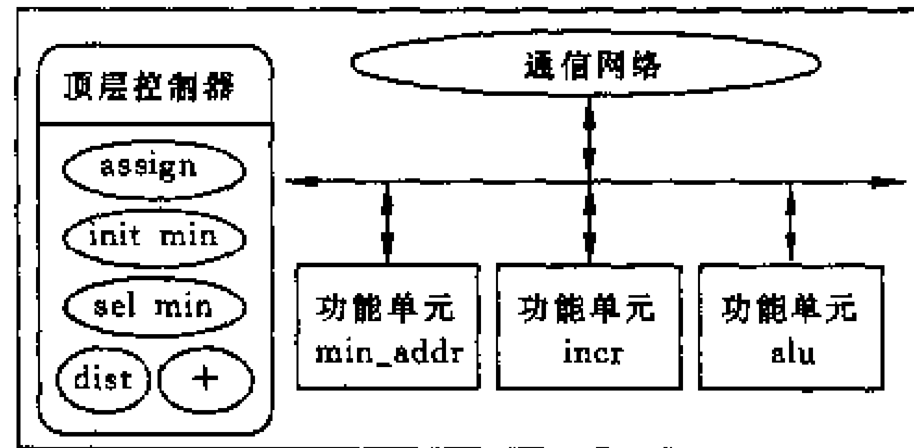


图 90.4　协同处理器的抽象结构图

```
--协同处理器的实体描述
 entity co_processor is
   port(clk            : in bit1;
        reset          : in bit1;
        c              : in bit8;
        s              : in bit8;
        p_sel          : in bit1;
        mode           : in bit1;
        sel_read       : out bit1;
        c_ad           : out bit2;
        s_ad           : out bit3;
        dmin           : out bit11;
        vector         : out bit4;
        done0          : out bit1;
        done1          : out bit1);
     end co_processor;
--以下是协同处理器的结构体描述，仅由一个进程组成
 architecture behavior of co_processor is
 begin
     process
      variable  i            :bit3_r;
      variable  x            :bit4_r;
      variable  d,min        :bit11_r;

--下面的函数是为了求当前 burst 与 sequence 的累加差异而进行的
    function dist(c,s: in bit8;d: in bit11_r) return bit11_r is
      variable  val8         :bit8_r;
      variable  val11        :bit11_r;
      begin
        val8:=to_stdlogicvector(c)-to_stdlogicvector(s);
--以下的 if 语句是求 val8 的绝对值
```

```
          if val8(7)=' 1'
          then
          val8(7):=' 0' ;
          end if;
          val11:=val8 + d;--累加差异值
          return (val11);
        end dist;

--下面的函数是计算并得到地址信息
      procedure assign(i: in bit3_r;
                       x: in bit4_r;
                       signal c_ad:OUT bit3;
                       signal s_ad:OUT bit5) is
      begin
--以下得到在 sequence 中的偏移值
        c_ad <= to_stdulogicvector(i) after 2 ns;
--以下得到在两个相邻 burst 组成的字符串中的偏移值
        s_ad <= to_stdulogicvector(("00"&i)+(' 0' &x)) after 2 ns;
        end assign;

--下面的函数是对一些变量进行初始化
        procedure init_min(signal dmin:OUT bit11) is
        begin
          min := "11111111000";
          dmin <= "00000000000";
        end init_min;

--以下是找到与查找的字符序列最接近的 8 个字符的起点
    procedure sel_min(d: in bit11_r; x: in bit4_r;
              signal dmin:OUT bit11;signal vector:OUT bit4) is
      begin
        if d < min then
          min := d;
          vector <= to_stdulogicvector(x);
        end if;
          dmin <= to_stdulogicvector(min);
      end sel_min;

--以下是主进程
      begin
        sel_read <= ' 0' ;
        done0 <= ' 1' ;
```

```
        done1 <= '1' ;
--以下根据不同的 p_sel 值来选择合适的工作方式
        wait until p_sel = '1'  and rising_edge(clk)
                               and reset = '1' ;
         if mode = '0'  OR reset = '0'  then
--这是除了最后一次循环外的工作方式
                i         :="000";
                x         :="0000";
                d         :="00000000000";
                init_min(dmin);
                vector <= "0000";
                done0  <= '0' ;
         else
                if mode = '1'  then
                     done1 <= '0' ;
              end if;
         end if;

      wait until p_sel = '0'  and rising_edge(clk) and reset = '1' ;
--这是最后一次循环外的工作方式
         loop_x: while reset='1'  loop
              d := "00000000000";
              loop_i: while reset='1'  loop
                        sel_read <= '1' ;
                        assign(i, x, c_ad, s_ad);

--等待一个时钟周期以装入地址
               wait until rising_edge(clk);
               sel_read <= '0' ;
                        d := dist(c, s, d);
                        i := i+1;
                        if i = 0 then
                               exit loop_i;
                        end if;
               end loop loop_i;
               sel_min(d, x, dmin, vector);
               x := x+1;
               if (x=0 OR x=8) then
                      exit loop_x;
               end if;
           end loop loop_x;
       end process;
```

```
        end behavior;
--外层循环对读入的两个 burst 组成的字符串进行扫描
--内层循环计算该部分序列对待查找序列的偏差值
```

（源描述文件名：90_wss_coprocessor.vhd）

第91例　实现窗口搜索算法的并行系统（2）——序列存储器

李　杰

1. 电路系统工作原理

第90例中已经讨论了WSS中的协同处理器，现在开始讨论其中的序列存储器。

序列存储器子系统主要管理搜索序列，将存储器块抽象为执行单一过程的功能单元，单一过程是指装入序列 sequence 的过程 write_ram。需要说明的是，该存储器为双端口存储器，一个端口用于存储器与外部的联系，另一个端口用于与协同处理器通信。

序列存储器与其他各个部分之间的连接关系如图91.1所示。

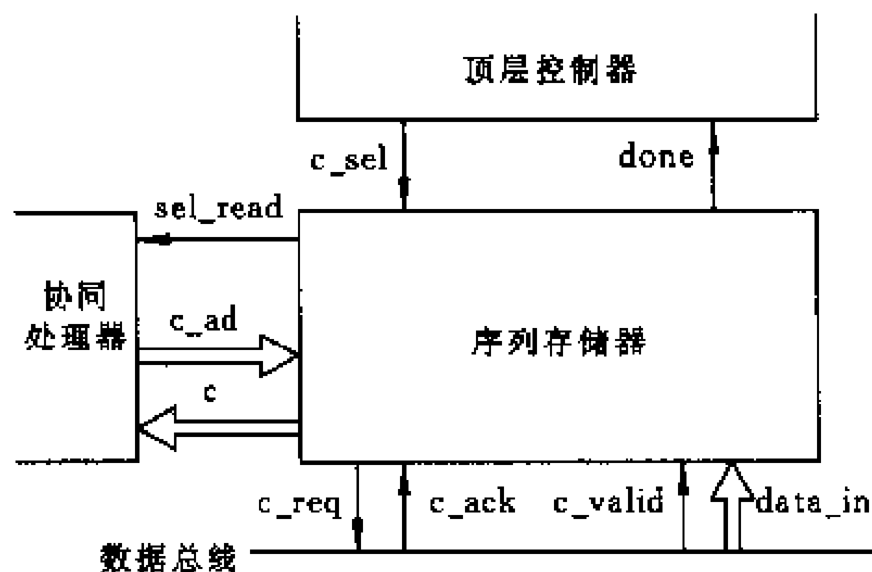


图91.1　序列存储器与其他各个部分之间的关系

2. VHDL语言描述及语法分析

在 mem_sequence 的主进程描述中并未包括与协同处理器的通信，这种联系是通过结构体部分的 **port map** 语句来实现的。

图91.2是序列存储器的抽象结构图。其中的过程 write_ram 和+操作分别由功能单元 mem_8x8 和 c_inc 完成。

mem_sequence 由一个存储序列的存储器和一个局部控制器组成，其行为描述如下。

```
libbrary ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_arith. all;
```

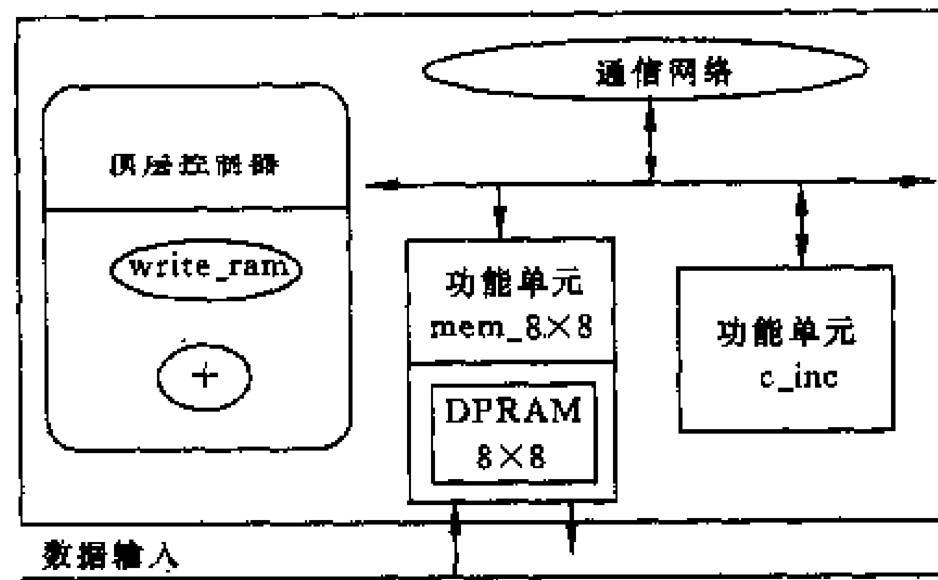


图 91.2　序列存储器的抽象结构图

```
use work.pkg_types. all;
use work.pkg_components. all;

entity mem_sequence is
 port(    clk             : in bit1;
          reset           : in bit1;
          sel_read        : in bit1;
          c_sel           : in bit1;
          c_req           : out bit1;
          c_ack           : in bit1;
          c_valid         : in bit1;
          data_in         : in bit8;
          c_ad            : in bit3;
          c               : out bit8;
          c_done          : out bit1);
 end mem_sequence;

architecture behavior of mem_sequence is
  signal  al         :bit3_r;
  signal  addr       : std_logic_vector(2 downto 0);
  signal  sel_write      :bit1;
begin
process
--以下是写内存的过程
procedure write_ram(sel_write:in bit1;data:in bit8) is
     begin
          al<=addr after 2 ns;
     end write_ram;
     --下面是主进程
```

```
begin
  c_req <= '0';
  c_done <= '1';
  addr <= "000";
  sel_write <= '0';
  wait until(c_sel = '1' and rising_edge(clk) and reset = '1');
  c_done <= '0';
  c_req <= '1';
  wait until(c_ack='1' and rising_edge(clk)) or reset/='1';
  c_req <= '0';
  write_loop:loop
   if c_valid/='1' and addr/="111" then
     wait until(c_valid='1' and rising_edge(clk)) or reset/='1';
        if reset/='1' then
           exit write_loop;
        end if;
   end if;
   addr <= addr+1;
   write_ram(sel_write,data_in);
   sel_write<='1';
   if addr="111" or reset/='1' then
       wait until rising_edge(clk);
       exit write_loop;
       end if;
   wait until rising_edge(clk);
   sel_write<='0';
   end loop write_loop;
   end process;
   sequence_mem:mem_8x8
   port map(c,data_in,c_ad,a1,sel_read,sel_write,clk);
 end behavior;
```

（源描述文件名：91_wss_sequence.vhd）

第 92 例　实现窗口搜索算法的并行系统（3）——字符串存储器

李　春

1. 电路系统工作原理

该子系统 mem_string 是管理字符串的。它同样连接到协处理器、顶层控制器，并与外部相连，如图 92.1 所示。它完成接收字符串序列。它使用一个存储块，这个存储块被抽象成功能性单元用来执行 write_ram 进程。其作用是装入字符串。存储器是一个双向端口的 RAM，其中一个端口用于接收外来输入，另一个是用于与协处理器通信。字符串存储器的 VHDL 描述比序列存储器的 VHDL 描述多了 init 进程，本进程用于对信号 addr 和 a1 赋初值。进程由功能性单元 s_inc 执行。字符串存储器包含 7 个输入端口和 3 个输出端口。s_sel 端口是字符串存储器的片选信号；s_ad 为 5 位的地址信号；sel_read 为读允许信号；burst 指示存入第几个 burst，本例字符串存储器只能存储 3 个 burst；当 s_done=1 时表示字符串存储器处理完成，字符存储器可以接收下一个动作。s_req，s_ack，s_valid 是与测试台（CPU）的握手信号。当 s_req=1 时存储器要求输入数据，s_ack 为 CPU 对存储器的响应信号，存储器接收到 s_ack=1 后，等待 s_valid 信号的到来，s_valid 表示 data_in 数据输入信号已就绪。

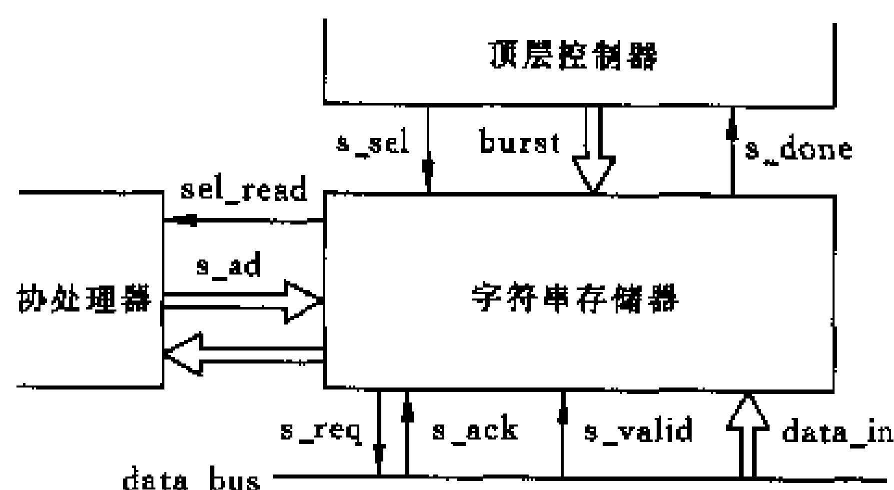


图 92.1　字符串存储器子系统原理图

2. VHDL 描述及语法分析

```
library ieee;
  use  ieee.std_logic_1164.all;
  use ieee.std_logic_arith.all;
  use work.pkg_types.all;
```

```
 use work.pkg_components.all;
entity mem_string  is
port    ( clk        : in  bit1 ;
          reset      : in  bit1 ;
          sel_read   : in  bit1 ;
          s_sel      : in  bit1 ;
          burst      : in  bit2_r ;
          s_req      : out  bit1 ;
          s_ack      : in  bit1 ;
          s_valid    : in  bit1 ;
          data_in    : in  bit8 ;
          s_ad       : in  bit5 ;
          s          : out  bit8 ;
          s_done     : out  bit1 );
end mem_string;

architecture behavior of mem_string is
signal  a1,addr         : bit5_r;
signal  sel_write       : bit1;
begin
 process
  一地址初始化
  procedure init (burst : in  bit2_r;signal a1 : out bit5_r) IS
  begin
     addr <= burst & "000";
     a1    <= burst & "000";
  end init;
  一写 RAM，即更新相应地址信号
  procedure write_ram ( sel_write : in bit1; data : in bit8) IS
  begin
     a1    <= addr;
  end write_ram;
  begin
   一初始化信号值
   s_req        <= ' 0' ;
   s_done       <= ' 1' ;
   addr         <= "00000";
   sel_write    <= ' 0' ;
   --直到 mem_string 为工作状态且被选中，当时钟上升沿到来时，
   wait until (s_sel = ' 1'  and rising_edge(clk) and reset=' 1' );
   --找到相应 burst 的开始位置，并且置 mem_string 为写入状态
   init(burst,addr);
```

```
  s_done <= '0';
  s_req  <= '1';
  wait until ((s_ack='1' and rising_edge(clk)) or reset/='1');
  --得到应答信号 s_ack 后，撤消 s_req 信号
  s_req <= '0';
  write_loop  : loop
      --等到 addr 有效且地址信号不为边界
      --因为一个 burst 设定为 8 个字符
      if s_valid/='1' and addr/=7 and addr/=15 and addr<23 then
             wait until ((s_valid='1' and rising_edge(clk))
                    or reset/='1');
             --复位则退出
             if reset/='1' then
                  exit write_loop;
             end if;
      end if;
      --写入 8 个字符，一个字符为 8 位
      addr <= addr +1;
      write_ram(sel_write, data_in);
      --写入完成后，写信号置位防止写入不正确的数据
      sel_write <= '1';
      --当地址达到边界或 mem_string 复位时退出
      if addr=7 or addr=15 or addr>=23 or reset/='1' then
             wait until rising_edge(clk);
             exit write_loop;
      end if;
      wait until rising_edge(clk);
      sel_write <='0';
  end loop write_loop;
 end process;
 string_mem :mem_24x8
 port map (s, data_in, s_ad, a1, sel_read, sel_write, clk);
end behavior;
```

（源描述文件名：92_wss_stringreg.vhd）

第93例　实现窗口搜索算法的并行系统（4）——顶层控制器

李　春

1. 电路系统工作原理

顶层控制器生成的都是控制信号，用于对序列存储器、协处理器和字符串存储器的协调工作。顶层控制器在 command=1 时开始工作。给出序列存储器和字符串存储器的数据信号和片选信号 c_sel=1 和 s_sel=1，在 s_done=0 和 c_done=0 时，即序列存储器和字符串存储器响应片选后，撤消片选信号，并等待（s_done=1 和 c_done=1）序列存储器和字符串存储器处理完成。另外，对字符串存储器重复一次同样的操作，存入下一个 burst。等待数据存入就绪后，给出协处理器片选信号（p_sel=1）。与此同时，并行地给字符串存储器写入下一个 burst，直到处理过程结束。详见图 93.1。

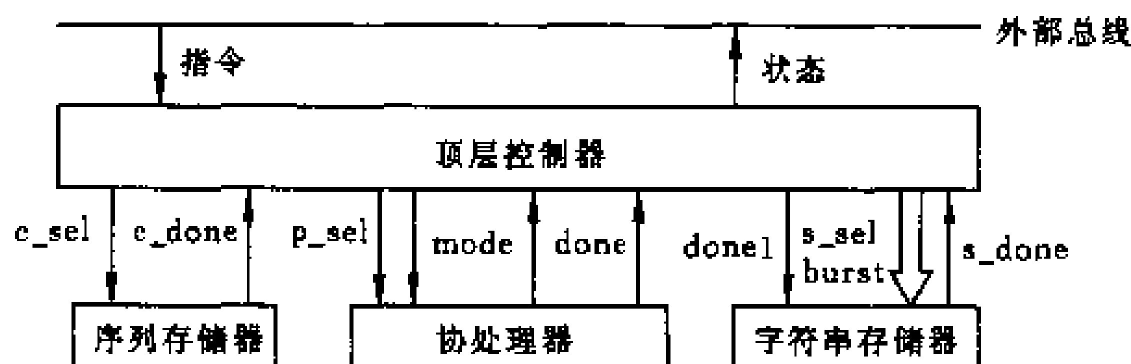


图 93.1　顶层控制器原理图

2. VHDL 描述及语法分析

```
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_arith.all;
   use work.pkg_types.all;
entity top is
port     ( clk         : in    bit1;
         reset         : in    bit1;
         command       : in    bit1;
         c_sel         : out   bit1;
         s_sel         : out   bit1;
         burst         : out   bit2_r;
         p_sel         : out   bit1;
```

```
      mode        : out  bit1;
      c_done      : in   bit1;
      s_done      : in   bit1;
      done0       : in   bit1;
      done1       : in   bit1;
      status      : out  bit1);
end top;

architecture behavior of top is
constant n          :integer:=2;
signal  burst_int   :bit2_r;
begin
 process
 begin
  --初始化
  mode <= '0' ;
  burst_int <= "00";
  p_sel <= '0' ;
  c_sel <= '0' ;
  s_sel <= '0' ;
  status <='0' ;
  main_loop : loop
    --等到所有组件都准备就绪
    wait until c_done='1'  and reset='1' and rising_edge(clk) and
               s_done='1' and done0='1' and done1='1' ;
    status <= '1' ;
    --等命令的到来
    if command /='1'  then
     wait until command='1' and reset='1' and rising_edge(clk);
    end if;
    --设定初始工作信号，但 co_processor 不为最后一个处理过程
    mode <='0' ;
    burst_int <="00";
    status <='0' ;
    c_sel <='1' ;
    s_sel <='1' ;
    --等到 mem_string 和 mem_sequence 有应答信号
    wait until c_done='0' and s_done='0' and rising_edge(clk);
    --选定信号复位
    c_sel <='0';
    s_sel <='0' ;
    --等到 mem_string 和 mem_sequence 的装载过程处理完成
```

```
      wait until c_done='1' and s_done='1' and rising_edge(clk);
      if reset/='1' then
        exit main_loop;
      end if;
      --mem_string 装载下一个 burst
      burst_int <= burst_int + 1;
      s_sel <='1';
      wait until s_done='0' and rising_edge(clk);
      s_sel <='0';
      wait until s_done='1' and rising_edge(clk);
      if reset/='1' then
         exit main_loop;
      end if;
      --当 burst_int 在范围内，找出与 mem_sequence 中最相近的 burst
      while burst_int < n and reset='1' loop
        burst_int <= burst_int + 1;
        s_sel <='1';
        p_sel <='1';
        wait until s_done='0' and done0='0' and rising_edge(clk);
        s_sel <='0';
        p_sel <='0';
        wait until s_done='1' and done0='0' and rising_edge(clk);
      end loop;
      if reset/='1' then
        exit main_loop;
      end if;
      --说明 co_processor 为最后一个处理过程
      mode <='1';
      p_sel <='1';
      wait until done1='1' and rising_edge(clk);
    end loop main_loop;
  end process;
  burst <=burst_int;
  end behavior;
```

3. 系统组装

为了验证 4 个行为描述的可行性，系统中包含了两个存储器模块、一个协处理器和一个顶层控制器。只有当 3 个子系统综合完成后才能进行综合控制。WSS 直接作用于外部的信号有两个：command（开始计算）和 status（忙状态标识或某一结果的合法性）。WSS 还有几个其他的信号是协处理器和存储器模块与外界通信用的，如图 93.2

所示。

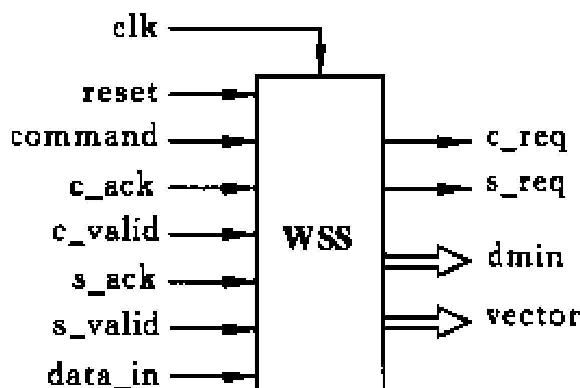


图 93.2　系统原理图

顶层控制器与各子系统的通信使用的是一种非常简单的协议。在启动一个组件时，顶层控制器需要设置相应的选择信号，若该组件处于忙状态，它会重设相应的 done 信号。顶层控制与子系统的互连如图 93.1 所示。

一般有两种方法来描述这样一个系统。子系统可作为一个外部组件或一个功能性单元。在第 1 种情况中，通信信号是直接在描述中赋值的。在 WSS 的结构描述中包含了顶层和 3 个对应于 3 个子系统的实例。于是组件和顶层控制的通信协议的任何改变将导致对顶层控制器的行为描述的修改。使如字符串存储器在顶层控制器描述中出现 3 次，如果字符串存储器与控制器的通信协议改变了，将导致修改顶层控制器描述的 3 个不同地方。本例的描述采用了此方案。

另一种描述模型的方法是使用过程来隐藏顶层控制器与子系统之间的通信协议。在这种情况中，字符串存储器的配置描述可用一个名为 write_string 的过程来替代。选择协议的变化将只引起顶层控制器描述的单一的改变。控制选择字符串存储器的过程 write_string 是唯一需要改变的代码。当通信协议非常复杂并且使用多个信号时，这种隐藏通信协议的方法将更为有效。读者可以根据下面的描述自己来加以实现。

4. VHDL 描述方法及语法分析

```
-- wss 为一个组装程序，把 mem_string, mem_sequence, co_processor
-- 和 top controller 组装为一个系统
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_arith. all;
   use work.PKG_types. all;
   use work.pkg_components. all;
  entity wss is
port
       reset        : in   bit1;
```

```
        clk         : in   bit1;
        command     : in   bit1;
        c_req       : out  bit1;
        c_ack       : in   bit1;
        c_vaild     : in   bit1;
        s_req       : out  bit1;
        s_ack       : in   bit1;
        s_valid     : in   bit1;
        data_in     : in   bit8;
        dmin        : out  bit11;
        vector      : out  bit4;
        status      : out  bit1);
end wss;
architecture rtl of wss is
constant n                         : integer:=2;
signal  c_sel, c_done              : bit1;
signal  s_sel, s_done              : bit1;
signal  p_sel, done0, done1        : bit1;
signal  mode, sel_read             : bit1;
signal  burst                      : bit2_r;
signal  s_ad                       : bit5;
signal  c_ad                       : bit3;
signal  s, c                       : bit8;
begin
 tp : top
 port map
        (clk, reset, command, c_sel, s_sel, burst, p_sel, mode, c_done, s_done,
         done0, done1, status);
 mq : mem_sequence
port map
      (clk, reset, sel_read,  c_sel, c_req, c_ack, c_vaild, data_in, c_ad, c, c_done);
 m1 : mem_string
 port map
 (clk, reset, sel_read, s_sel, burst, s_req, s_ack, s_valid, data_in, s_ad, s, s_done);
 pr :co_processor
 port map
(clk, reset, c, s, p_sel, mode, sel_read, c_ad, s_ad, dmin, vector, done0, done1);
end rtl;
```

5. 小结

本例是一个基于窗口算法的字符串搜索的描述。整个系统划分为 4 个模块，其中包括协处理器、字符串存储器、序列存储器和一个顶层控制器。4 个模块组装成一个单独的 WSS，WSS 用于接收整个系统所需的激励信号。

（源描述文件名：93_wss_top.vhd）

第94例　MB86901流水线行为描述组成框架

石　峰

流水线行为的VHDL语言描述实质上就是将流水线的各流水站(pipeline stage)在某个时钟周期内的操作用VHDL语言描述出来。流水站的操作随该流水站指令寄存器中指令的不同而有所不同，与一般的流水线不同，MB86901的流水线操作还要包括各种处理数据相关机构以及优化分支处理机构的描述。根据描述目的的不同，可以使用不同的描述方法和描述风格，本例及随后6例是根据指令在流水线中的流动过程进行描述的。在描述中没有采用结构描述的方法，但描述过程中尽量追求具体功能单元描述与描述目标在结构上的某种对应。当然，用户也可以根据自己的需要进行不同风格和算法的描述。本例以MB86901的4级流水线为例，说明流水线行为描述的描述算法及描述程序的结构。

1. 描述算法结构

MB86901芯片内部采用4级流水线结构，分为取指、译码、执行以及写回这4级，每级使用一个相应的指令寄存器。4个流水站的操作是并行的，可以按照各流水站在不同时间段执行的操作情况对流水线的整体功能进行描述，因此在进行流水线功能的具体描述前应该首先对流水线流程进行深入的分析。下面以MB86901指令流水线流程图为例加以简要说明。

由指令（单周期指令）组成的指令序列流经流水线时，流水线处于满载状态，流水线的各级同时处理各自指令寄存器中的一条指令，因此，虽然某一条指令从入流水线到出流水线共需4个时钟节拍，但是，由于在每个时钟节拍，有一条单周期指令从流水线中出来的同时，又有一条指令进入流水线，每4个时钟节拍就一定有4条单周期指令从流水线中出来，因此，平均而言，这样一条指令从取指、译码、执行至写回的整个操作过程只需要一个时钟周期，其流程图如图94.1所示。

由图94.1，在水平方向上，可以清楚地说明某一条指令在4个连续的时钟周期内通过指令流水线的时序情况；从垂直方向看，可以清楚地说明在某一个时钟周期内，指令流水线各级处理指令的情况。对应于图94.1，指令流水线各级主要操作如下：

① 取指阶段

在该阶段，处理器向地址总线输出下一条指令的地址（注意：该指令要经过一定时间的延时以后才能出现在地址总线上，然后由外部锁存器锁存，其后应由处理器外部存储控制器控制数据读取，在该时间段内，处理器外部的存储控制器应对数据总线上的数

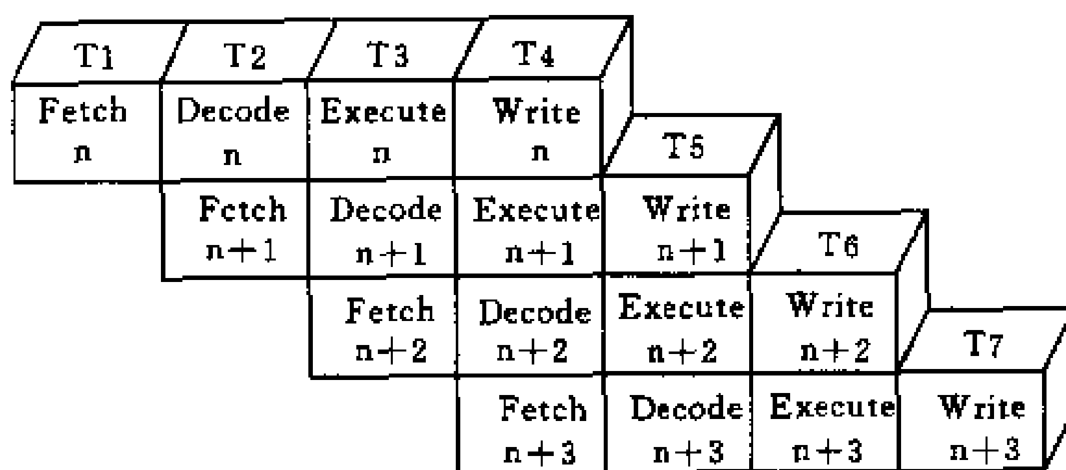


图 94.1 MB86901 流水线操作流程图

据——当前指令——保持不变提供保证），由数据总线读入当前指令到指令译码寄存器IDR 中，以便在下一个时钟周期到来时对其进行译码。

② 译码阶段

在该阶段，处理器计算源操作数的寄存器地址，然后从寄存器文件中读取源操作数到操作数寄存器，以便在下一个时钟周期到来时指令流水线的执行阶段使用。在本阶段，处理器还要计算下一条指令地址。

③ 执行阶段

处理器在该阶段执行算术逻辑运算（在 ALU 中），包括数据传递指令的目标地址等，运算结果将被锁存，在处理器内部的运算结果暂存寄存器 RESBUS 中。

④ 写回阶段：

处理器将操作结果写入目标寄存器。

值得指出的是，在指令流水线的上述流程中，所有操作数都来自寄存器文件，操作结果亦写入寄存器文件。

从流水线流程图和各流水线的操作来看，在某个时钟周期内，流水线的 4 个流水站分别处理各自指令寄存器中的指令，4 个指令寄存器中的 4 条指令的处理是完全并行的，并且在逻辑结构上相对独立。由于上述 4 个阶段的共用资源（如内部总线、寄存器等等）在时间上是分时复用的，因此在描述时只需将各站访问共用资源的时间分开，那么，就可以把流水线的 4 个流水站看作是 VHDL 语言的 4 个并行进程，各进程间以时钟的上升沿同步，在各进程内部放置描述相应流水站应执行操作的 VHDL 语言语句。需要注意的是，根据该站当前所处理指令的不同，该站执行的操作也不尽相同。因此，各站内部操作的描述应该考虑到该站指令寄存器内的指令是各种指令的情况，根据不同指令去分别处理。图 94.2 说明流水线这种基于并行算法的 VHDL 语言源描述程序的结构，图中的 IDR，IER，IWR 分别是对应译码、执行、写回流水站的指令寄存器，在时钟信号的同步下，每个流水站处理完各自的操作后，将各自指令寄存器中的指令传至下一个流水站的指令寄存器中。

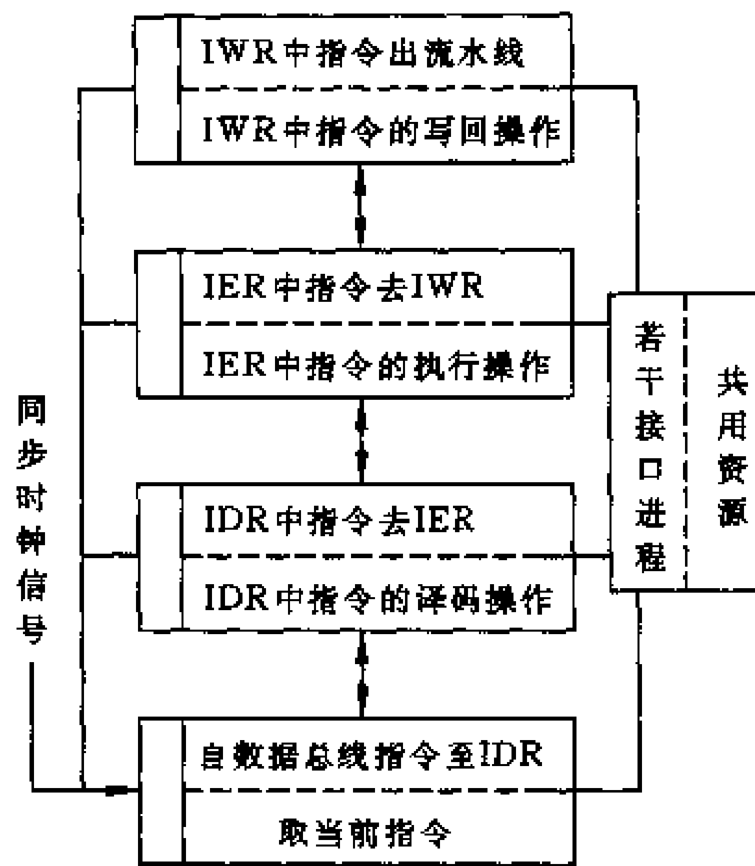


图 94.2 流水线行为描述程序结构

2. 各个模块的主要功能及时序关系

根据上述原理，采取以下步骤可以将流水线的各流水站在某个时钟周期内对相应指令寄存器内部指令的操作用 VHDL 语言进行行为描述，从而最终得到流水线的行为模型。

① 由于存在多周期指令，流水线内部的操作十分复杂，只用一个时钟很难同步某时钟周期内流水站中的各种操作，所以 MB86901 引入两个同步时钟，CLK1 和 CLK2，二者之间的关系如图 94.3 所示。为了清晰地描述源程序结构，在行为描述中把比较复杂的流水站操作对应于两个 VHDL 语言进程，并分别用两个时钟信号作为两个进程的敏感信号，以同步两个进程内部的操作。这样，对应于取指、译码、执行、写回等 4 个流水站的操作，共用七个 VHDL 语言进程描述，分别标记为 fetch1，fetch2；decode1，decode2；exec1、exec2；write1。其中，进程名中后缀为 1 的进程由时钟信号 CLK1 同步，后缀为 2 的进程由时钟信号 CLK2 同步。

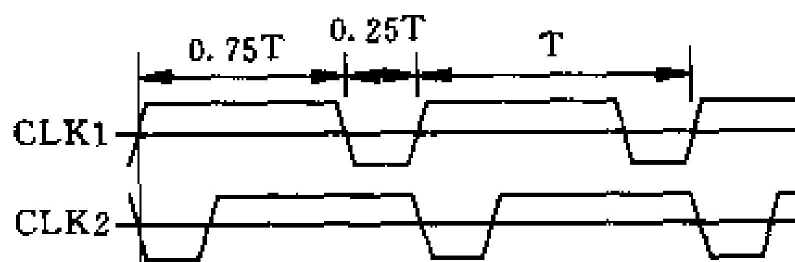


图 94.3 双相时钟时序图

② 与 decode，excute，write 相对应的指令寄存器分别设为 IDR，IER，IWR，IWWR1，IWWR2 等。每个时钟周期结束前，整个寄存器链移位一次，指令依次从一个指令寄存器

移至下一个指令寄存器，同时，进程 fetch 从数据总线上将一条指令读入指令译码寄存器 IDR。

③ 各进程主要完成如下操作：

fetch：计算下一条指令地址，在本时钟周期结束前将其送入地址总线，同时将数据总线上的指令读入到指令译码寄存器 IDR 中，然后更新地址计数器为 PCn+1。有时也需要完成多周期指令后若干指令的缓存等操作。

decode：根据指令译码寄存器中指令的源操作数寄存器地址，将源操作数从这些寄存器中读出送到 ALU 的操作数寄存器 A 或 B 中。有时需处理源操作数数据相关。

excute：根据 IER 的内容和前一时钟周期指令译码的结果，执行相应操作，将结果存入 RESULT 寄存器中。

write：根据 IWR 中的内容，将 RESULT 寄存器中的运算结果送入寄存器堆的目标寄存器中；IWWR1，IWWR2 用于多周期指令的描述。

MB86901 的地址总线和数据总线都是 32 位。一般而言，下一条指令地址产生后，在当前时钟周期结束前由控制器将其放入地址总线 ADR（同时并行地在数据总线上读取指令）。即当前指令的读取和下一条指令地址的输出同时进行，真正读取下一条指令是在下一个时钟周期结束前。这时，这条指令的地址已经在地址总线上保持了一个时钟周期，外部存储器应该在这一个时钟周期内把这个地址的数据放到数据总线上（这里是指 Cashe hit 时）。该过程的时序如图 94.4 所示。

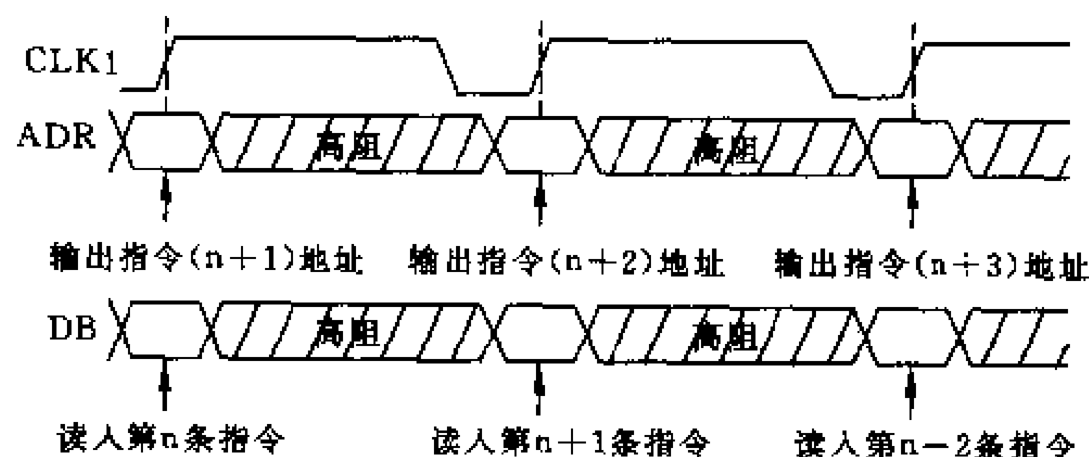


图 94.4　取指操作时序

3. 描述程序基本框架

根据上述建模步骤、各进程主要操作及时序，很容易得到描述程序的基本框架：

```
fetch1: process
begin
    wait on CLK1 until CLK1='0';
    ......
    将地址计数器中的下条指令地址或数据地址送入地址总线 ABUS
    ......
```

```
end process  fetch1;

fetch2: process
begin
     wait on CLK2  until  CLK2='1';
     ......
     if        多周期指令控制信号未赋值        then 单周期指令后取指的相关操作;
     elsif     上一条指令是控制转移指令        then 控制转移指令后取指的相关操作;
     elsif     上一条指令是 CALL 指令          then CALL 指令后取指的相关操作;
     elsif     上一条指令是数据传送指令        then 数据传送指令后取指的相关操作;
     end if;
     ......
end process fetch2;

decode1: process
begin
     wait on CLK1 until CLK1='1';
     ......
     根据不同情况将数据总线 DBUS 上的指令放入 IDR 或缓存器 BUFFER1;
     多周期指令控制信号赋值 after CLK1 上升沿
     计算源操作数地址及目的寄存器地址
     ......
end process decode1;

decode2: process
begin
     wait on CLK2 until CLK2='1';
     ......
     将操作数或立即数送入 RSA 或 RSB
     ......
end process decode2;

execute1: process
begin
wait on CLK1 until CLK1='1';
     ......
     执行 IER 中指令的操作，计算出结果
     ......
end process execute1;

execute2: process
begin
```

```
    wait on CLK2 until CLK2='1';
    ......
    结果送 RESULT
    ......
end process execute2;

write: process
begin
    wait on CLK1 until CLK1='1';
    ......
    RESULT 中结果送入目的寄存器
    ......
end process write;
```

MB86901 流水线结构十分复杂，上面只对其流水线功能的总框架进行了描述，对于该流水线结构中所特有的各种数据相关及控制相关的硬布线处理、高效率控制转移优化技术、窗口寄存器管理等，都应根据具体的处理流程加以描述，有关问题详见第 95 例至第 100 例。

第95例　MB86901寄存器文件管理的描述

石　峰

1. MB86901寄存器的窗口化管理

MB86901的寄存器堆充分体现了SPARC体系结构的特征寄存器堆包括120个32位通用寄存器和3个用于选择(configure)和控制处理器操作的可编程32位寄存器——处理器状态寄存器(processor state register，简称PSR)、窗口非法屏蔽寄存器(window invalid mask register，简称WIM)、陷阱基地址寄存器(trap base register，简称TBR)。

120个通用寄存器又称为工作寄存器，实行窗口化管理，其中有8个全局寄存器(r0~r7)，其他寄存器在逻辑上组成7个窗口，每个窗口24个寄存器，由于有部分重叠，重叠的部分用作上个寄存器窗口输出，所以实际上每个窗口供用户直接使用的只有16个，某个时刻只有一个窗口被激活(由WIM的低七位指定)，其逻辑结构如图95.1所示。图中Rn表示寄存器的物理编号，rn表示寄存器的逻辑编号。即，某个状态下用户可以

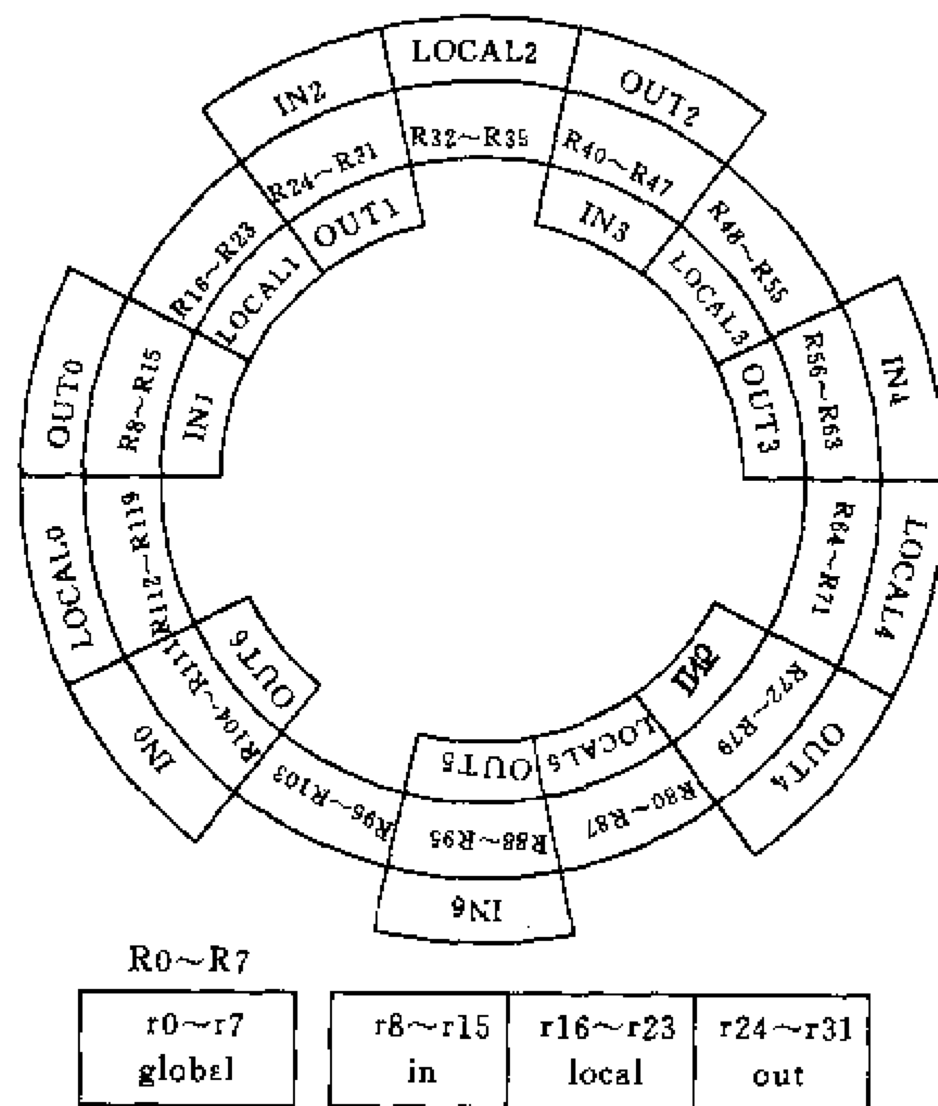


图95.1　寄存器窗口管理

直接使用的寄存器有 32 个，其逻辑编号为 0～31，其中最低的 8 个为全局寄存器，另外 24 个为某个寄存器窗口中的寄存器，其真正的物理地址取决于窗口寄存器指针 CWP(PSR 的低五位)。

2. 窗口寄存器的描述

寄存器的窗口化管理是 SPARC 结构的特点之一，可以将其作为相对独立的模块用结构化的方法去描述。在本书中 MB86901 流水线结构行为描述中有关寄存器管理的操作，采用一种行为描述的方法，用一个函数 AbsNum( )加以描述，该函数的框架如下：

```
subtype RegNoType is integer range 119 downto 0;
function AbsNum  (constant reg_index in BIT_VECTOR( 4 downto 0);
                  constant forcwp   in BIT_VECTOR( 4 downto 0)
                  )return RegNoType is
variable num  :  RegNoType;
begin
case reg_index  IS
     when "00000"  =>  num :=0;
     ......
     when "00111"  =>  num :=7;
     when others  =>
case forcwp(2 downto 0)  is
     when "000" =>
          case reg_index is
               when "01000" => num :=112;
               ......
               when "01111" => num :=119;
               when "10000" => num :=8;
               ......
               when "11111" => num :=23;
               when others => num :=null;
          end case;
     when "001" =>
          case reg_index is
               when "01000" => num :=16;
               ......
               when "11111" => num :=39;
               when others => num :=null;
          end case;
          ......
     when "110" =>
          case reg_index IS
```

```
                when "01000" => num :=96;
                ......
                when "11111" => num :=119;
                when others => num :=null;
            end case;
      when others => null;
      end case;
    end case;
return num;
end AbsNum;
```

这个函数的功能实际上只是根据当前窗口寄存器指针及窗口寄存器逻辑地址（r0~r31）返回该寄存器在寄存器堆（register file）中的物理地址，由于寄存器堆在描述中用一个120个分量的一维32位bit_vector型数组表示，因此寄存器的物理地址实际上就是它的数组下标，所实现的各寄存器窗口的物理地址见图95.1中的Rn。该描述具有很强的结构化描述特点，十分直观，便于理解。

第96例 MB86901内ALU的行为描述

石 峰

MB86901采用了一个快速32位超前进位整数ALU，该ALU有3个端口分别用作操作数的输入和运算结果的输出。两个操作数输入端可以分别接受来自寄存器、流水线硬线旁路及指令所含直接数；运算结果输出端将运算结果送往结果暂存器暂存后再写回到寄存器文件中。ALU能在一个时钟周期内完成所有算术运算和逻辑运算，并根据运算结果设置或改写微处理器的整数条件码，将其写入状态寄存器PSR。

MB86901的ALU可以通过一个过程加以描述，这主要是因为ALU除了要返回运算结果外还要返回有关进位和溢出等信息，具体示意如下：

```
subtype RegType is bit_vector(31 downto 0); --此类型用于所有32位寄存器

function RegToULVector( constant rvector : BIT_VECTOR )
                        return Std_Ulogic_vector  is
    variable result : std_ulogic_vector(rvector' LENGTH -1 downto 0);
    alias sub : bit_vector(rvector' LENGTH -1 downto 0) is rvector;
begin
for i  in  result' RANGE  loop
    case  sub(i)  is
          when ' 0'    => result(i) := ' 0' ;
          when ' 1'    => result(i) := ' 1' ;
          when others => result(i) := ' 0' ;
        end case;
    end loop;
    return result;
end  RegToULVector;

procedure     arith(
    signal          tbus : inout RegType;
    constant  rsa  : in RegType;
    constant  rsb  : in RegType;
    signal          c0   : inout bit;
    signal          c1   : inout bit;
    constant optype: in bit_vector(3 downto 0)
    )is
```

```
variable ursa    :    std_ulogic_vector(32 downto 0);
variable ursb    :    std_ulogic_vector(32 downto 0);
variable ursal   :    std_ulogic_vector(32 downto 0);
variable ursbl   :    std_ulogic_vector(32 downto 0);
variable res     :    std_ulogic_vector(32 downto 0);

begin
    ursal    :=  '0' & RegToULVector(rsa);
    ursbl    :=  '0' & RegToULVector(rsb);
    case optype  is
       when "0000"
       => ursa(32 downto 0) := "00" & ursal(30 downto 0);
          ursb(32 downto 0) := "00" & ursbl(30 downto 0);
          res :=   ursa + ursb;
          cl  <=   To_BIT(res(31), '0') after Delay;
          ursa(32 downto 0) := '0' & ursal(31 downto 0);
          ursb(32 downto 0) := '0' & ursbl(31 downto 0);
          res :=   ursa + ursb;
          c0  <=   To_BIT(res(32), '0') after Delay;

       when "0001"
       => ursa(32 downto 0) := "00" & ursal(30 downto 0);
          ursb(32 downto 0) := "00" & ursbl(30 downto 0);
          res :=         ursa - ursb;
          cl  <=         To_BIT(res(31), '0') after Delay;
          ursa(32 downto 0) := '0' & ursal(31 downto 0);
          ursb(32 downto 0) := '0' & ursbl(31 downto 0);
          res :=         ursa - ursb;
          c0  <=         To_BIT(res(32), '0') after Delay;
    --各种算术运算对整数条件码的改写操作在主描述程序中描述

    --下面为各种逻辑操作
       when "0010" => res := ursal and ursbl;
       when "0011" => res := ursal or ursbl;
       when "0100" => res := ursal xor ursbl;
       when "0101" => res := ursal nand ursbl;
       when "0110" => res := ursal nor ursbl;
       when "0111" => res := not (ursal xor ursbl);
       when others =>  null;

    end case;
    tbus <= To_bitvector(res(31 downto 0), '0') after Delay;
```

```
end arith;
```

在上述描述中，rsa 和 rsb 分别对应于 ALU 的两个操作数输入端口，tbus 为运算结果输出端口，C0 和 C1 用于传递进位、溢出等整数条件码信息，optype 用于指示所需的操作类型。对于 ALU 改写整数条件码的操作是行为模型的其他部分描述，该部分不便于从 MB86901 行为模型中独立出来。

C1 的值等于两个操作数的低 30 位相加所产生的向第 31 位的进位，C0 等于两个 31 位操作数相加所产生的进位。这两个进位值用以判断操作结果是否溢出和产生进位，其基本规律是，当 C0 和 C1 相异时，则发生溢出。有关问题请参见计算机体系结构及组成原理方面的书籍。

描述中的函数 RegToULVector 用于从 RegType 类型到 STD_ULOGIC_VECTOR 类型的转换，从而能够借用 1164 包中定义的有关 STD_ULOGIC_VECTOR 类型信号及变量的加减运算。

需要指出的是，本描述完全是为了建立 MB86901 行为模型，因此仅给出了 ALU 一类器件的描述方法，而描述的目的更不是面向综合的，只是一种较高层次的行为描述。由于行为描述与具体综合目标之间缺乏直接的对应，因此本描述在某种综合系统上的综合目标结构并非一定是 MB86901 所要求的超前进位 ALU，除非用户在综合过程中进行适当的约束，才可能使综合目标结构尽量接近真正的 MB86901 的 ALU。另外，由于在 MB86901 的产品说明中并没有规定 ALU 具体完成哪些操作，返回哪些信息，因此本描述仅仅在整个的 MB86901 行为模型中完成相当于 ALU 的操作，可能有些 ALU 的操作是在行为模型的其他部分描述，也有可能在本描述中包含了一些 MB86901 的 ALU 所有操作之外的操作，因此我们说本描述在行为功能上相当于 MB86901 的 ALU。

第 97 例　移位指令的行为描述

石　峰

MB86901 有 3 条移位指令：逻辑左移 SLL、逻辑右移 SRL 和算术右移 SRA，它们的含义如图 97.1 所示：

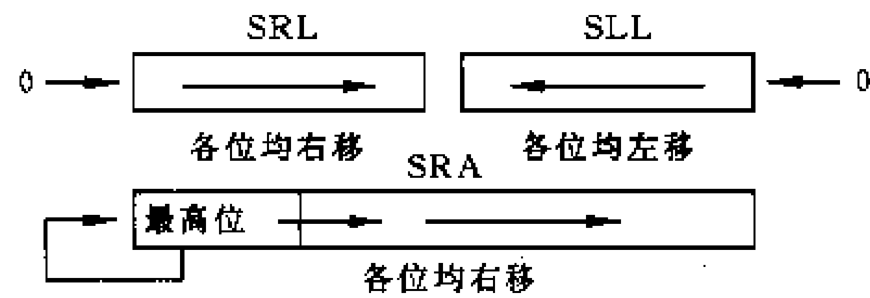


图 97.1　MB86901 移位指令的含义

根据这 3 条指令所执行操作的意义，在 MB86901 的行为模型中以一个函数的形式描述它们的功能，具体如下（请参见书中所附 CD)。

```
subtype  RegNoType is   integer range 119 downto 0;
subtype  RegType   is   bit_vector(31 downto 0);     --32 位寄存器类型
type RegFileType   is   array( RegNoType )  of  RegType;
type ShiftType     is   (SSLL, SSRL, SSRA);

function  shift(
                 constant        op1  :    RegType;
                 constant        op2  :    RegType;
                 constant        stype:    ShiftType
               ) return  RegType  is
    variable res : RegType;
begin
    if   stype=SSLL
    then case op2 (4 downto 0)    is
           when "00000" => res:= op1;
           when "00001" -> res:=op1(30 downto 0) & R032(0 downto 0);
           when "00010" => res:=op1(29 downto 0) & R032(1 downto 0);
           ...... ......
           when "11110" => res:=op1(1 downto 0) & R032(29 downto 0);
           when "11111" => res:=op1(0 downto 0) & R032(30 downto 0);
         end case;
elsif    stype=SSRL
```

```
then case op2(4 downto 0) is
        when "00000" => res := op1;
        when "00001" => res := R032(0 downto 0)& op1(31 downto 1);
        when "00010" => res := R032(1 downto 0)& op1(31 downto 2);
        ......
        when "11110" => res:= R032(29 downto 0)&op1(31 downto 30);
        when "11111"=> res:=R032(30 downto 0) & op1(31 downto 31);
      end case;
    elsif    stype=SSRA
    then
        if op1(31)= ' 1'  then case op2(4 downto 0) is
        when "00000" => res := op1;
        when "00001" => res := R132(0 downto 0)& op1(30 downto 0);
        when "00010" => res := R132(1 downto 0)& op1(29 downto 0);
        ......
        when "11110" => res := R132(29 downto 0)& op1(1 downto 0);
        when "11111" => res := R132(30 downto 0)& op1(0 downto 0);
        end case;

        else case op2(4 downto 0) is
        when "00000" => res := op1;
        when "00001" => res := R032(0 downto 0)& op1(30 downto 0);
        when "00010" => res := R032(1 downto 0)& op1(29 downto 0);
        ......
        when "11110" => res := R032(29 downto 0)& op1(1 downto 0);
        when "11111" => res := R032(30 downto 0)& op1(0 downto 0);
        end case;
        end if;
    end if;
return res;
end shift;
```

上述描述中所用到的 VHDL 语言语法现象并不很多，描述的层次很清晰，具有很强的结构感，因此，对其稍加修改后便能够适应某些综合系统的要求，综合出与源描述具有某种对应关系的综合目标结构，这对于探讨行为描述与综合目标结构之间的对应关系是有意义的。应该注意的是，VDHL 语言中‘&’符号的意义是“毗连”而不是通常的“与”运算。

第98例　单周期指令的描述

石　峰

MB86901的算术逻辑运算等指令为单周期指令，由这类指令组成的指令序列流经流水线时，流水线处于满载状态，流水线的各级同时处理各自指令寄存器中的一条指令，因此，虽然某一条单周期指令从进流水线到出流水线需要4个时钟节拍，但是，由于每个时钟节拍在一条单周期指令从流水线中出来的同时，又有一条单周期指令进入流水线，每4个时钟节拍就一定有4条单周期指令从流水线中出来。因此，平均而言，这样一条指令从取指、译码、执行到写回的整个操作过程只需要一个时钟周期，此时的流水线流程图请参见第94例中的流程图。不考虑其他多周期指令的单周期指令描述很简单，只要按第94例的描述程序框架加入具体的操作描述即可，但需要注意操作数数据相关的处理。

下面的描述程序是从MB86901流水线的行为描述（取指及译码部分）模型中提炼出来的，其目的是为读者读懂MB86901行为描述提供便利。为清晰起见，原描述程序中许多逻辑表达式被省略或由汉语说明所替代。另外，程序中形如{PC3<=PC2, ……}的部分在原描述程序中由一个专门的进程完成。程序中有关信号的多进程赋值问题也被屏蔽掉。

```
constant UZ32 : STD_ULOGIC_VECTOR:="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"
constant SZ32 : STD_LOGIC_VECTOR := "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"
fetch21 : process
begin
    wait on CLK1 until (onfetch2='1') and (复位信号无效);
    if  CLK1='0'            --这部分操作均开始于CLK1下降沿
        then                --单周期指令
         if(现在处理的是单周期指令)and(不处于处理其他多周期指令过程中)
                then ADRBus   <=  PC0      after Delay1;
                     IDR      <=  DBus after Delay1;
        elsif  处理其他指令
        end if;
    end if;
end process fetch21;

fetch22 : process
begin
    wait on  CLK2 until    CLK2='1'
```

```
        if  (第n条指令是单周期指令) and (不处于处理其他多周期指令过程中)
            then          ADRBus  <= UZ32     after Delay;
                Dbus      <= SZ32 after Delay;
                RD        <= '1'        after Delay;
                WE        <= '1'        after Delay;
                {PC3<=PC2;PC2<=PC1;PC1<=PC0;PC0<=PC0+4;}
    end if;
end process fetch22;

decode1 : process   -- 将指令放入 IDR 中
begin
    wait on CLK1 until (ondecode='1'); --ondecode='1' 执行段正常，否则关闭
    if  CLK1='1'
    then case op is
        when "00"=>  BICC 等指令译码   (CLK1 上升沿)
        when "01"=>  CALL 指令译码
        when "10"=>    如为 Mulscc, Jmpl, TICC 等指令则进行相应译码如是单周期指令（算术
                       逻辑运算）则空操作
        when "11"=>  ......
        end case;
    elsif    CLK1='0'                        --CLK1 后沿
    then     IER <= IDR    after Delay1;
    end if;
end process decode1;

decode2 : process       --  Send oprands to RSA or RSB
begin
    wait on CLK2 until (CLK2='1')and(ondecode='1')
    if IDR 中指令是第 0 类指令          then ......
    elsif IDR 中指令是第 1 类指令 then ......
    elsif IDR 中指令是第 2 类指令
    then if 是算术运算指令
        then if (源操作数地址=上一条指令目标寄存器地址) --数据相关
              then RSA<=RESBUS    after Delay2;
                      --需对其他源操作数根据具体类型做类似处理
              elsif(源操作数地址=上两条指令目标寄存器地址)--数据相关
              then  RSA<=RESULT  after Delay2;
                      --需对其他源操作数根据具体类型做类似处理
              end if;
        ......
```

```
      end if;
      ......
  end if;
......
end process decode2;
```

第 99 例　多周期指令的描述

石　峰

多周期指令的描述十分复杂，这主要是因为 MB86901 中对多周期指令带来的各种数据相关的处理完全由硬件解决，而且这种解决机制很复杂，因此描述多周期指令首先要充分理解指令的流水线操作流程图，根据流程图所规定的每个流水站的操作进行描述。本例仅以 LOAD 指令为例，说明多周期指令的描述。首先介绍 LOAD 的流程图，然后给出描述 LOAD 指令的一个框架。

1．LOAD 指令流水线流程图

对于单周期指令，每个时钟周期内都要进行一次取指操作，从数据总线上读取指令代码。然而对于 LOAD，STORE 等多周期指令而言，由于在某个时钟周期要占用数据总线存取数据信息，该时钟周期不能进行正常的取指令操作，因此多指令流水线的流程图与单周期指令流水线的流程图颇有不同。下面说明 LOAD 指令流水线流程图。

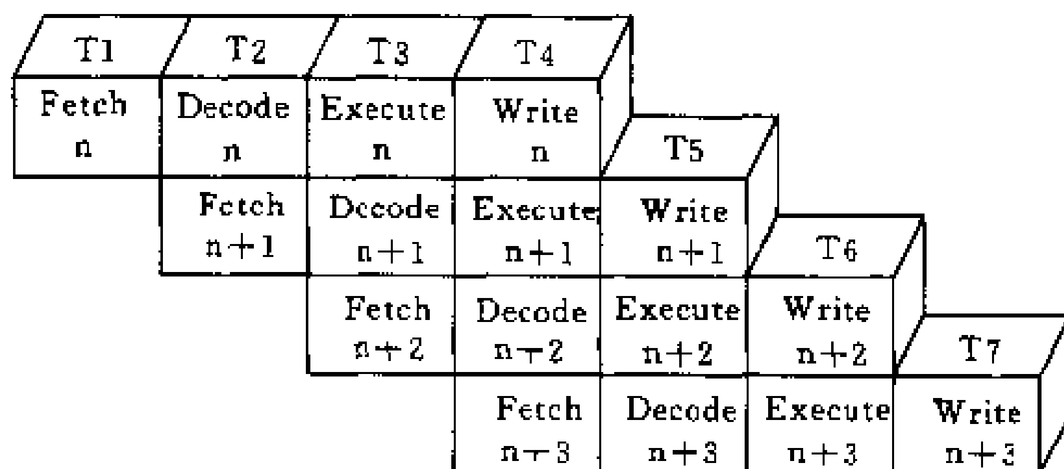


图 99.1　单周期指令流水线流程图

首先，对单周期指令而言，指令流水线满载时，每个时钟周期内并行完成 4 条单周期指令的操作，此时指令流水线流程图如图 99.1 所示。对 LOAD 指令，考虑该指令前后至少 3 条指令为单周期指令的情形：由于上面存在某个取指操作被中断的情况，此时用于从数据总线上读取其他数据信息，在下一个时钟周期恢复正常的取指操作，因此从总体上看，指令流水线在这一段时间内经过 5 个时钟周期，有 4 条指令进入指令流水线，同时有 4 条指令从指令流水线中出来，这 4 条指令中有 3 条为单周期指令，故 LOAD 指令全部操作需两个时钟周期，即指令流水线推迟一个时钟周期，因此对 LOAD 指令，应该相

应地执行一系列空操作用于推迟指令流水线操作时序以便同步 LOAD 指令和其他单周期指令的执行，这时流水线流程如图 99.2 所示（流程以不改变本过程中单周期指令流程为原则设计）。

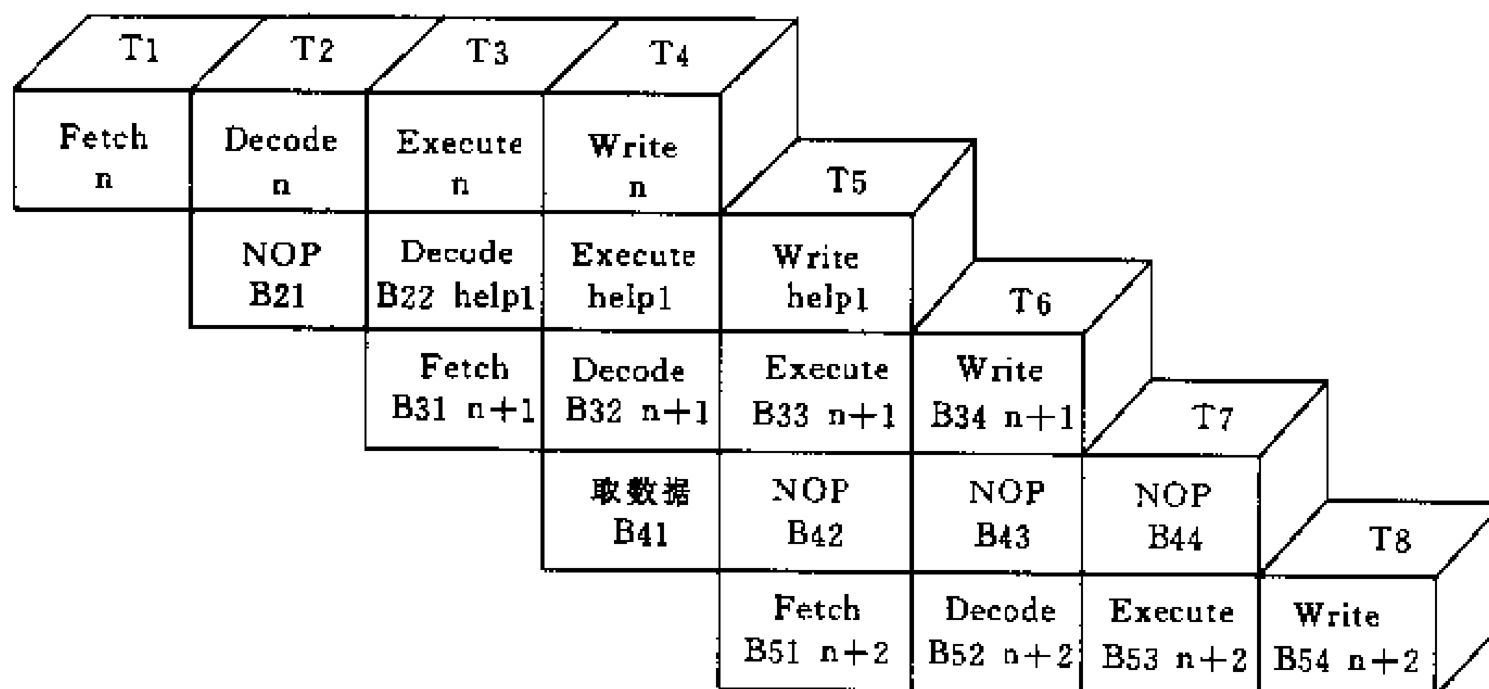


图 99.2 插入内部指令 help1 的 LOAD 指令流水线指令流程图

从图 99.2 中可以看到，help1 指令的插入使得第 n+1 条指令的一系列操作推迟一个时钟周期，同时，为了不改变本过程中单周期指令的流程，B42，B43，B44 相应的流水线操作为空操作，此时平均每条指令执行时间为$\frac{5}{3}$T。图 99.2 的指令流水线流程简单直观，其实现逻辑也很简单，是为一般 CPU 所广为采用的一种方法，但由于在整个过程中尚有 B21，B42，B43，B44 为空操作，降低了流水线的效率。所以 MB86901 相应的流水线流程由图 99.2 改进如下：将图 99.2 的 B31 取第 n+1 条指令的操作放入 B21 中，由于时

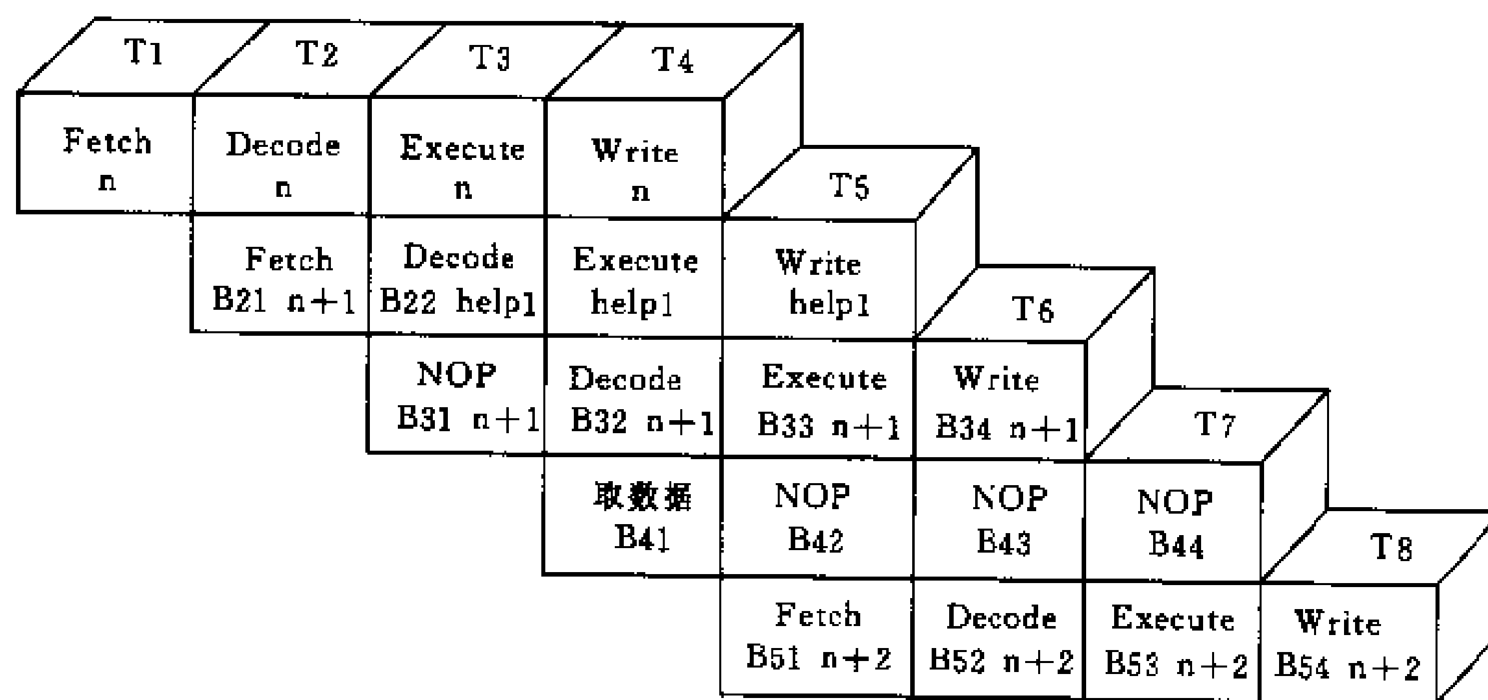


图 99.3 由图 99.2 改进而来的指令流水线流程图

钟周期 T3 指令译码寄存器被 help1 指令所占据，因此需要一个指令缓冲器 Buffer1，将

第 n+1 条指令在 T2 后沿暂时放入 Buffer1 缓存，在 T3 后沿，将其放入指令译码寄存器 IDR 中进行译码，因此得到如图 99.3 所示的流水线流程图。

由图 99.3 可见，原来位于 B21 的空操作现在转移到 B31，进一步地，可以将 B51 的取第 N+2 条指令的操作转移到 B31。在 T3 的后沿，将第 n+1 条指令放入 IDR 后，将第 n+2 条指令取入 Buffer1 中缓存，从而得到图 99.4。

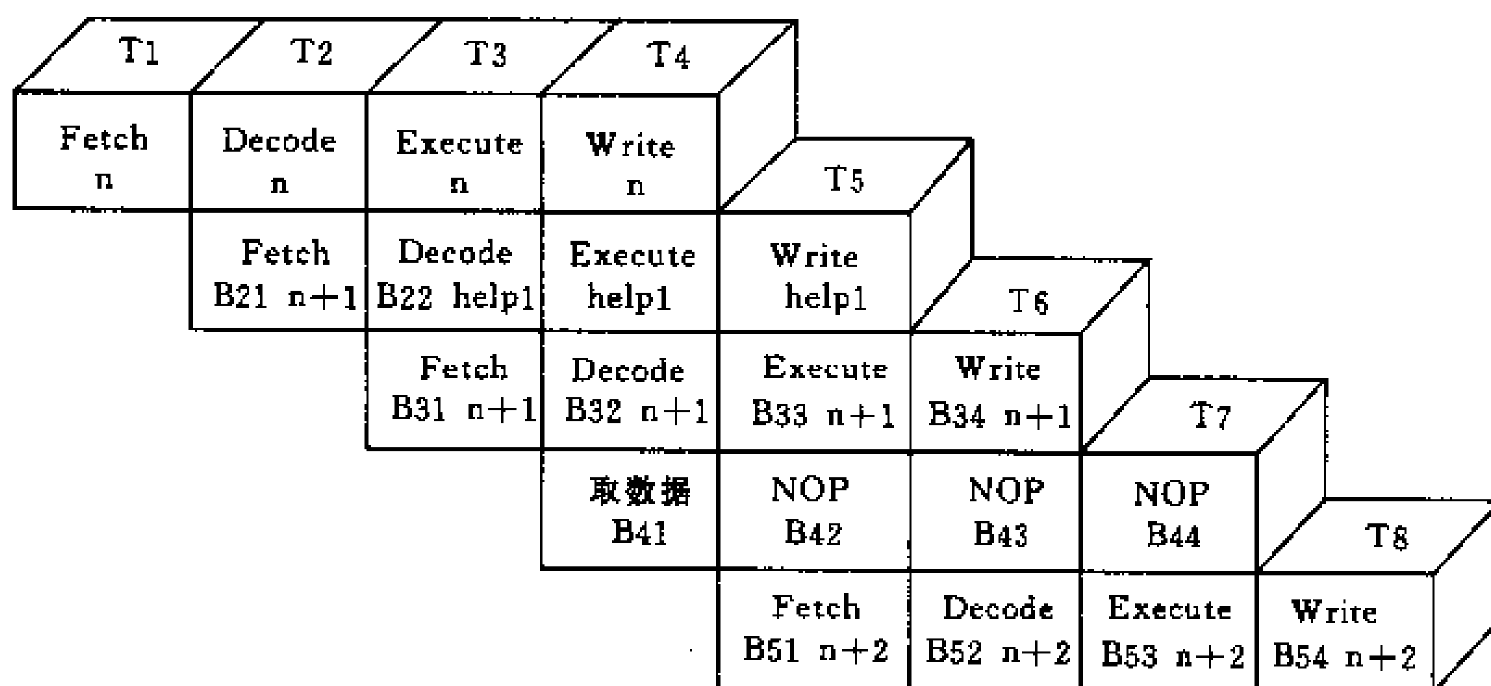


图 99.4 由图 99.3 改进而来的指令流水线流程图

继续将图 99.4 中的 B52，B53，B54 转移到相应的 B42，B43，B44 中，并且把流水线的后续操作提前，从而得到 MB86901 的 LOAD 指令流水线流程图，如图 99.5 所示。此时，由于 4 个时钟周期完成 3 条指令，所以平均每条指令的时间为$\frac{4}{3}$T，因此图 99.5 的流水线效率比图 99.2 的流水线效率提高了 15%。

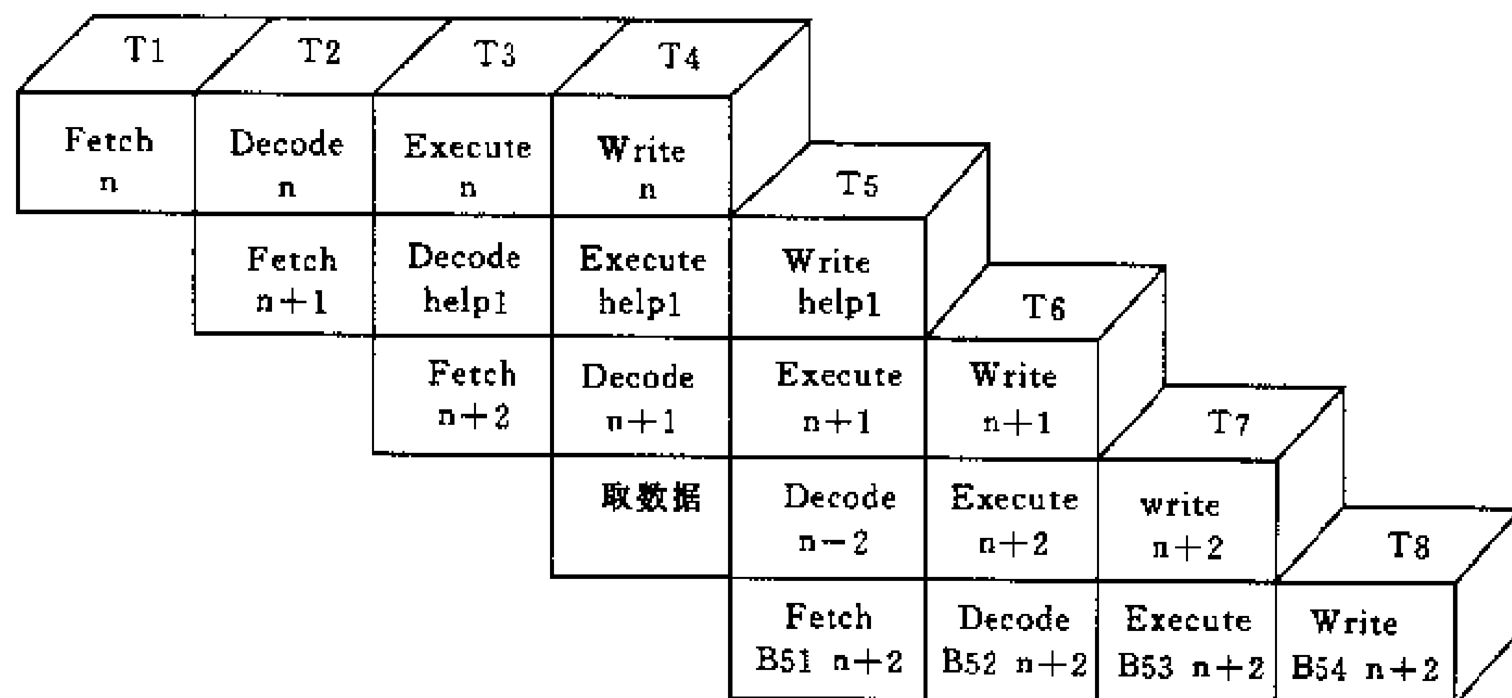


图 99.5 MB86901 LOAD 指令流水线流程图（无 LOADLOCK）

2. 指令间数据相关的流水线流程

考虑某指令序列进入流水线，其中某条指令的操作数是上一条指令运算的结果，由于一条指令从进入流水线到指令操作完毕从流水线中出来需要 4 个时钟周期，所以第 2 条指令无法立即得到第 1 条指令的操作结果，出现两条指令的操作数发生所谓的数据相关；当紧跟在 LOAD 指令之后的一条指令（n+1）以 LOAD 指令取入的数据为操作数时，由于 LOAD 指令要在第 4 个时钟周期结束前或第 5 个时钟周期到来时才能将数据取入目标寄存器，而 LOAD 指令的下一条指令（指令 n+1）却要在第 4 个时钟周期到来时就调用 LOAD 指令取入的数据作为操作数以便（指令 n+1）译码，此时会发生另一种数据相关，即所谓互锁（interlock），简称为 LOADLOCK，类似的情况也发生在 LDD 指令流水线流程中；对条件转移等控制指令的情形，当控制条件不满足时流水线执行条件转移指令的下一条指令，当控制条件满足时流水线将转去执行转移目标地址的指令，而对已经取入流水线的紧跟在控制转移指令之后的指令停止执行，此时发生所谓控制相关。下面仅介绍 LOADLOCK 数据相关的处理。

MB86901 利用硬件解决 loadlock：在 T4 时保留取指寄存器 IDR 中的指令而不进行取操作数操作，并且关闭相应的流水线执行阶段和写回阶段各一个时钟周期，在 T5 时再重新进行译码（这时操作数已经取入）。相应的流程图见图 99.6。对于 LDD 指令也存在类似问题。

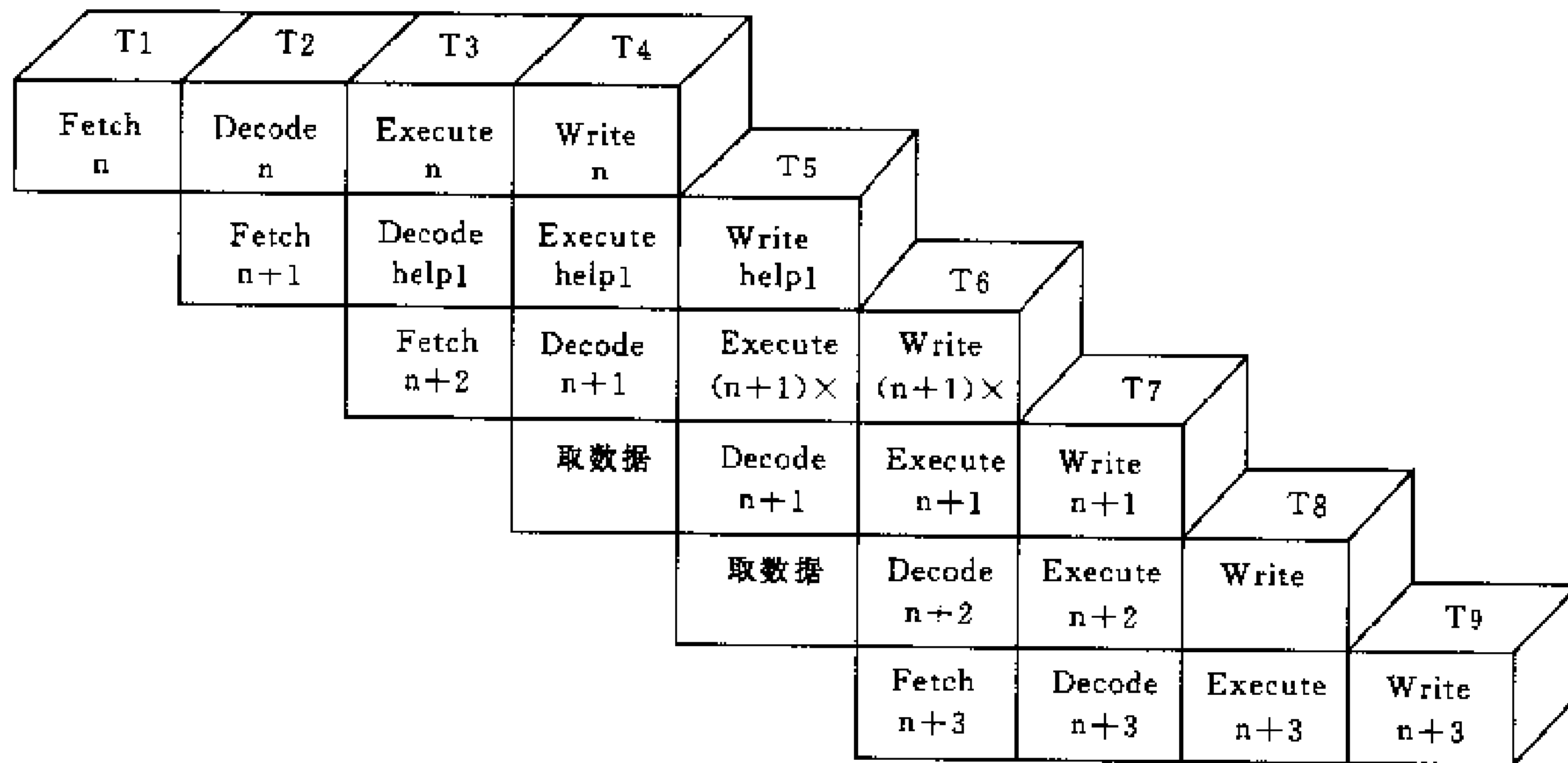


图 99.6 LOAD 指令流水线操作流程图（LOADLOCK）

下面的描述程序是从 MB86901 流水线的行为描述模型中提炼出来的，为清晰起见，原描述程序中的许多逻辑表达式被省略或由汉语说明所替代。另外，程序中形如

{PC3<=PC2, ……}的部分在原描述程序中是由一个专门的进程完成。程序中有关信号的多进程赋值问题也被屏蔽掉。

```
constant UZ32 : STD_ULOGIC_VECTOR:="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"
constant SZ32 : STD LOGIC VECTOR := "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"
fetch21 : process
begin
    wait on CLK1 until (onfetch2=' 1' ) and (复位信号无效);
    if CLK1=' 0' --这个部分操作均开始于 CLK1 下降沿
    then
    if  现在处理的是单周期指令--single cycle instruction
        then ADRBus    <= PC0    after Delay1;
             IDR       <= DBus   after Delay1;
    — 取指
    elsif (ldcount=1) and (不处于其他多周期指令过程)--LD 取指 (周期 2)
        then ADRBus         <= PC0    after Delay1;
             Buffer1   <= DBus   after Delay1;
             IDR       <= help1  after Delay1;
    elsif (ldcount=2) and (不处于其他多周期指令过程)--LD 取指 (周期 3)
        then ADRBus    <= PC0         after Delay1;
             IDR       <= Buffer1     after Delay1;
             wait for 0 NS;
             Buffer1   <=DBus         after Delay1;
    elsif(ldcount=3) and (不处于其他多周期指令过程)--LD 取指 (周期 4)
                  and (loadlock=0)                     --无 loadlock 发生
        then ADRBus <= PC0 after Delay1;
             Result <= DBus after Delay1;
             case  第 n+1 条指令  IS  --IDR 中指令
             when 非访存指令     =>IDR<=Buffer1     after Delay1;
             when 访存指令(多周期指令)=>IDR<=help1 after Delay1;
             end case;

—LOAD 指令取指第 5~8 段，该时段操作描述自然并入其他指令第 2~5 段

--以下为有 loadlock 发生时的情况
      elsif (ldcount=3) and (loadlock=1) --LD 取指 (周期 4) 有 loadlock
         then    ADRBus <= PC0       after Delay1;
                 Result <= DBus      after Delay1;
                 onexec <= ' 0'      after Delay1;
                                          --关闭下个周期流水线的执行段
      elsif (ldcount=4) and (loadlock=1) -- LD 取指 (周期 5)
        then ADRBus    <= PC0    after Delay1;
```

```
            Result    <= DBus    after Delay1;
            onexc     <= ' 1'    after Delay1;
                                       --下个周期恢复流水线执行段
            onwrite   <= ' 0'    after Delay1;
                                       --下个周期关闭流水线写回段
            case  第 n+2 条指令  IS  --IDR 中指令
            when 非访存指令     => IDR<=Buffer1    after Delay1;
            when 访存指令(多周期指令)=> IDR<=help1 after Delay1;
            end case;
        —LD 取指（周期 6 到周期 9）
    —这个时段的操作描述自然并入其他指令周期 2 至周期 5 的描述
    —其他指令流水线流程取指的描述
    end if;

    elsif    CLK1=' 1'  and ((stxount=4)OR(stdcount=4)OR(stdcount=5))
        then DBus <= SZ32  after Delay1;
    end if;
end process fetch21;

fetch22 : process
begin
wait on CLK2 until CLK2=' 1'
-- 单周期指令
if  第 n 条指令  IS         --IDR 中指令
    then ADRBus    <= UZ32    after Delay;
         DBus      <= SZ32    after Delay;
         RD        <= ' 1'    after Delay;
         WE        <= ' 1'    after Delay;
         {PC3<=PC2;PC2<=PC1;PC1<=PC0;PC0<=PC0+4;}
-- LD 指令
elsif (CtrLd=' 1' ) and (不处于其他多周期指令过程) --现在为 LD 指令第 2 段
    then ADRBus    <= UZ32    after Delay;
         DBus      <= SZ32    after Delay;
         RD        <= ' 1'    after Delay;
         WE        <= ' 1'    after Delay;
         {PC3<=PC2;PC2<=PC1;PC1<=PC0;PC0<=PC0+4;}
         ldcount   <= 1       after Delay2; --标识下一周期为 LD 第 3 段
         --ldcount 在 CLK1 后沿后有效
elsif (ldcount=1) and (不处于其他多周期指令过程) --现为 LD 指令第 3 段
    then ADRBus<=UZ32  after Delay;
         DBus      <= SZ32    after Delay;
         RD        <= ' 1'    after Delay;
```

```
        WE        <= '1'   after Delay;
        ldcount   <= 2     after Delay2;
        {PC3<=PC2;PC2<=PC1;PC1<=PC0;ADRT2<=PC0,ADRT1<=PC0+4}
        --PC0: 第n+1条指令地址;  PC0+4: 第n+2条指令地址
elsif (ldcount=2) and (不处于处理其他多周期指令过程中) and (loadlock='0')
--现在为LD指令第4段
    then ADRBus   <= UZ32   after Delay;
        DBus      <= SZ32   after Delay;
        RD        <= '1'    after Delay;
        case  第n条指令  IS    --IWR中指令
        when  LDSB|LDSBA|LDUB|LDUBA => SIZE <= "00" after Delay;
        when  LDSH|LDSHA|LDUH|LDUHA => SIZE <= "01" after Delay;
        when  LD|LDA => SIZE <="10" after Delay;
        when  OTHERS => NULL;
        end case;
        WE        <= '1'   after Delay;
        ldcount   <= 3     after Delay2;
        if  第n+1条指令为BICC,CALL
        then {PC3<=PC2,PC2<=PC1,PC1<=ADRT2}  after Delay1;
        elsif 第n+1条指令为JMPL,LD,LDD,ST,STD,TICC,RETT
        then {PC3<=PC2,PC2<=PC1,PC1<=ADRT2,PC0<=ADRT1}
        xxxcount<=1   after Delay2;
        —xxxount对LD为ldcount,其他类似
    end if
elsif (ldcount=3) and (ldlock='0')          -- LD指令(周期5)
      then      ADRBus    <= UZ32   after Delay;
                DBus      <= SZ32   after Delay;
                RD        <= '1'    after Delay;
                WE        <= '1'    after Delay;
                case  第n+1条指令  IS  --IER中指令
                when 第0类指令
                     =>  ldcount<=0    after Delay2;
                         if  bicccount=1   --  IER中为BICC,现在为第3段
                             then bicccount<=2  after Delay2;
                                  {PC3<=PC2,PC2<=PC1,PC1<=PC0,PC0<=ADRT1}
                         end if;
                when 第1类指令
                     =>  ldcount <= 0   after Delay2;
                         {PC3<=PC2,PC2<=PC1,PC1<=PC0}
                when 第2类指令
                     =>  if  jmplcount=1   --IER中为JMPL,现在为其第3段
                             then ldcount        <= 0      after Delay2;
```

```
                jmplcount <= 2      after Delay2;
                {PC3<=PC2, PC2<=PC1, PC1<=PC0}
      else if  第n+2条指令为BICC
                then ldcount  <= 0 after Delay2;
                                                 --结束LD流程
         {PC3<=PC2, PC2<=PC1, PC1<=PC0, ADRT1<=PC0+4}
                elsif 第n+2条指令为CALL
                then ldcount  <= 0 after Delay2;
                                                 --结束LD流程
                          {PC3<=PC2, PC2<=PC1, PC1<=PC0}
                elsif 第n+2条指令为JMPL
                then ldcount  <= 0 after Delay2;
                                                 --结束LD流程
                          jmplcount  <= 1after Delay2;
                          --将进入jmpl第3段
        {PC3<=PC2, PC2<=PC1, PC1<=PC0, PC0<=PC0+4}
                elsif 第n+2条指令为LD
                then ldcount  <= 1 after Delay2;
                                   --将进入LD第3段
        {PC3<=PC2, PC2<=PC1, PC1<=PC0, PC0<=PC0+4}
                elsif 第n+2条指令为ST
                then ldcount<=0     after Delay2;
                                                 --结束LD流程
                     stcount<=1     after Delay2;
                     --将进入ST流程第3段
        {PC3<=PC2, PC2<=PC1, PC1<=PC0, PC0<=PC0+4}
                elsif  第n+2条指令为LDD
                then ldcount  <= 0 after Delay2;
                                                      —结束LD流程
                          lddcount  <= 1 after Delay2;
                          --将进入LDD第3段
        {PC3<=PC2, PC2<=PC1, PC1<=PC0, PC0<=PC0+4}
                elsif  第n+2条指令为STD
                then ldcount  <= 0 after Delay2;
                                                 --结束LD流程
                          stdcount  <= 1 after Delay2;
                          --将进入STD第3段
        {PC3<=PC2, PC2<=PC1, PC1<=PC0, PC0<=PC0+4}
                else ldcount  <= 0 after Delay2;
                                                 --结束LD流程
                end if
   end if
```

```
            when 第3类指令
            =>  if   第n+1条指令是LD
                then ldcount  <=2 after Delay2;--将进入LD第4段
                elsif     第n+1条指令是ST
                then stcount  <=2 after Delay2;--将进入ST第4段
                     ldcount  <=0 after Delay2;--结束LD流程
                elsif     第n+1条指令是LDD
                then lddcount<=2 after Delay2;--将进入LDD第4段
                     ldcount  <=0 after Delay2;--结束LD流程
                elsif     第n+1条指令是STD
                then stdcount<=2 after Delay2;--将进入STD第4段
                     ldcount  <=0 after Delay2;   --结束LD流程
                end if;
                               {PC3<=PC2, PC2<=PC1, ADRT1<=PC0+4, ADRT2<=PC0}
        end case;
—以下为有loadlock发生时的情况
elsif (ldcount=2) and (ldlock='1') and (不处于处理其他多周期指令过程中)
                                       --LD指令(周期4),带lock
    then ADRBus     <=UZ32    after Delay;
         DBus       <=SZ32    after Delay;
         RD         <='1'     after Delay;
         WE         <-'1'     after Delay;
         case  第n条指令  IS    --IWR中指令
         when  LDSB|LDSBA|LDUB|LDUBA => SIZE <="00" after Delay;
         when  LDSH|LDSHA|LDUH|LDUHA => SIZE <="01" after Delay;
         when  LD|LDA => SIZE <= "10" after Delay;
         when  OTHERS => NULL;
         end case;
         ldcount  <=3 after Delay2;   {PC3<=PC2, PC2<=PC1}
elsif (ldcount=3) and (ldlock='1') and (不处于处理其他多周期指令过程中)
                                           — LD指令(周期5),带lock
    then ADRBus     <=UZ32    after Delay;
         DBus       <=SZ32    after Delay;
         RD         <='1'     after Delay;
         WE         <='1'     after Delay;
         ldcount    <=4       after Delay2;
         SIZE       <="10"    after Delay;
         if   第n+1条指令为BICC,CALL
              then  {PC3<=PC2, PC2<=PC1, PC1<=ADRT2} after Delay1;
              elsif 第n+1条指令为JMPL,LD,LDD,ST,STD,TICC,RETT
              then {PC3<=PC2, PC2<=PC1, PC1<=ADRT2, PC0<=ADRT1}
                   xxxcount <=1  after Delay2;
```

```
            --xxxount 对 LD 为 ldcount,其他类似， 将进入 xxx 第 3 段
            else {PC3<=PC2,PC2<=PC1,PC1<=ADRT2,PC0<=ADRT1};
        end if;
elsif (ldcount=4) and (ldlock='1')        -- LD 指令（周期6），带 lock
    then ADRBus    <= UZ32   after Delay;
         DBus      <= SZ32   after Delay;
         RD        <= '1'    after Delay;
         WE        <= '1'    after Delay;
         case  第n+1 条指令  IS  —  IER 中指令
         when 第 0 类指令
            =>  ldcount <= 0 after Delay2;  --结束 LD 流水线流程
                if  bicccount=1   --IER 中为 BICC，现在为其第 3 段
                then bicccount <= 2 after Delay2;--入 BICC 第 4 段
                     {PC3<=PC2,PC2<=PC1,PC1<=PC0,PC0<=ADRT1}
                end if;
         when 第 1 类指令
            =>  ldcount <= 0 after Delay2;  --结束 LD 流水线流程
                {PC3<=PC2,PC2<=PC1,PC1<=PC0}
         when 第 2 类指令
            =>  if  jmplcount=1   --IER 中为 JMPL，现在为第 3 段
                then ldcount <= 0 after Delay2;--结束 LD 流程
                        jmplcount <= 2 after Delay2;--进入 JMPL 第 4 段
                           {PC3<=PC2,PC2<=PC1,PC1<=PC0}
                elsif 第 n+2 条指令为 BICC
                then ldcount <= 0 after Delay2--结束 LD 流程
                        {PC3<=PC2,PC2<=PC1,PC1<=PC0,ADRT1<=PC0+4}
                elsif 第 n+2 条指令为 CALL
                then ldcount <= 0 after Delay2;--结束 LD 流程
                        {PC3<=PC2,PC2<=PC1,PC1<=PC0}
                elsif 第 n+2 条指令为 JMPL
                then ldcount <= 0 after Delay2;--结束 LD 流程
                        jmplcount<=1 after Delay2;--将进入 jmpl 第 3 段
                        {PC3<=PC2,PC2<=PC1,PC1<=PC0,PC0<=PC0+4}
                elsif 第 n+2 条指令为 LD
                then ldcount <= 1 after Delay2;--将进入 LD 第 3 段
                        {PC3<=PC2,PC2<=PC1,PC1<=PC0,PC0<=PC0+4}
                elsif 第 n+2 条指令为 ST
                then ldcount <= 0 after Delay2;--结束 LD 流程
                        stcount <= 1 after Delay2;--将进入 ST 第 3 段
                        {PC3<=PC2,PC2<=PC1,PC1<=PC0,PC0<=PC0+4}
                elsif 第 n+2 条指令为 LDD
                    then ldcount <= 0 after Delay2;--结束 LD 流程
```

```
                    lddcount<=1    after Delay2;—将进入 LDD 第 3 段
                    {PC3<=PC2, PC2<=PC1, PC1<=PC0, PC0<=PC0+4}
                elsif 第 n+2 条指令为 STD
                then ldcount  <=0 after Delay2;--结束 LD 流程
                    stdcount<=1    after Delay2;--将进入 STD 第 3 段
                    {PC3<=PC2, PC2<=PC1, PC1<=PC0, PC0<=PC0+4}
                else ldcount  <=0 after Delay2;--结束 LD 流程
                end if
        when 第 3 类指令
            =>  if   第 n+1 条指令是 LD
                then ldcount  <=2 after Delay2;--将进入 LD 第 4 段
                elsif 第 n+1 条指令是 ST
                then stcount  <=2 after Delay2;--将进入 ST 第 4 段
                     ldcount  <=0 after Delay2;--结束 LD 流程
                elsif 第 n+1 条指令是 LDD
                then lddcount  <=2after Delay2;--将进入 LDD 第 4 段
                      ldcount  <=0 after Delay2;--结束 LD 流程
                elsif 第 n+1 条指令是 STD
                then stdcount  <=2after Delay2;--将进入 STD 第 4 段
                     ldcount  <=0 after Delay2;--结束 LD 流程
                end if;
        {PC3<=PC2, PC2<=PC1, ADRT1<=PC0＋4, ADRT2<=PC0}
end case;
--其他指令的描述
end if
end process fetch22;

decode1 : process
begin
wait on CLK1 until (ondecode='1');  --ondecode='1' 执行段运行，否则关闭
if CLK1='1'
    then case IDR 中指令 IS
        when 第 0、1、2 类指令 => ......
        when 第 3 类指令
            =>  if IDR 中指令 LD 是指令
                then CtrLd <= '1'  after Delay1;--设标志以进入 LD 流程
                elsif IDR 中指令是 LDD 指令
                then CtrLdd<= '1'  afterDelay1;--设标志以进入 LDD 流程
                ......
                end if;
                —LD（周期 4）带 Loadlock 并且 rs1(rs2) 等于 rd_IWR
                if (第 n+1 条指令源操作数地址是=LD 目标地址)
```

```
                                                --有 ldlock
                then ldlock <= '1'    after Delay1;
            elsif (如果有 lddlock)
                then after lddlock <= '1'  after Delay1;
            end if
    end case;
end if;
end process decode1;

decode2 : process
begin
wait on CLK2 until (CLK2='1') and (ondecode='1')
if IDR 中指令是第 2 类指令
    then if 是算术运算指令
        then if(源操作数地址=上一条指令目标寄存器地址)--数据相关
            then RSA <= RESBUS after Delay2;
            --需对其他源操作数根据具体类型做类似处理
              elsif(源操作数地址=上第 2 条指令目标寄存器地址)
                    --数据相关
            then  RSA <= RESULT after Delay2;
            --需对其他源操作数根据具体类型做类似处理
              end if;
        ......
      end if;
elsif IDR 中指令是其他类指令 then ......
end if;
......
end process decode2;
```

访存指令（比如 LOAD 指令）在某个时钟周期内占用数据总线 DBuss 取数据，因此该时钟周期从数据总线读取指令的操作就不能进行，流水线中取指的连续性被打断，所以访存指令不能在一个时钟周期内完成。LOAD 指令的取指和译码段需要加以注意的操作分析如下：

(1) 由于执行 LOAD 指令（从 DBuss 读数据）的操作与 LOAD 指令后第 3 条指令的读取发生共享总线冲突，需要把取指令的操作推迟一个时钟周期，因此把 LOAD 指令后的两条指令顺序地通过 Buffer1 中延迟一个时钟周期，在 LOAD 后第 1 条指令取指（读指令到 Buffer1 中）时，及时地将内部指令 help1 放入指令译码寄存器 IDR 中，由于只有在译码时才能知道 IDR 中的指令是否是 LOAD（即，是否将插入 help1 指令），所有将 IDR 中指令是否为 LOAD 指令的标识信号 CtrLd（为描述方便而引入）的赋值放置在译码阶段，并在 CLK1 的上升沿附近完成（用 AFTER 子句限制它在 CLK2 上升沿前生效）。

(2) 在取指阶段 CLK2 的上升沿附近，判断 CtrLd 是否为‘1’，从而知道 IDR 中指令是否为 LOAD 指令，如果是 LOAD 指令，除去正常的取指操作外，对 loadcount 信号赋值以标识当前时刻为 LOAD 流水线流程的哪一段。

(3) CLK1 的下降沿，在取指进程中发现 IDR 中为 LOAD 指令（此时 ldcount=1）时，将 DBuss 上的指令读入 Buffer1 缓存，同时将 help1 内部指令放入 IDR。

(4) 注意，在译码阶段判断是否有 Loadlock。

第 100 例　MB86901 流水线行为模型

石　峰

根据上述的描述算法、程序结构和框架，按照各指令的流水线流程图所规定各操作的时序关系，我们建立了 MB86901 的 VHDL 语言行为模型，该模型源描述部分由 6000 余行 VHDL 语句构成，其中主要进程 11 个、辅助进程 25 个、函数和过程 4 个；模型的测试部分根据流水线操作的不同分类方法提供了测试台和大量的测试码。这里所要介绍的只是有关描述中需要注意的两个问题及行为模型的测试问题。

1. 源程序结构

描述源程序 sparc. vhd 的程序结构见图 100. 1。

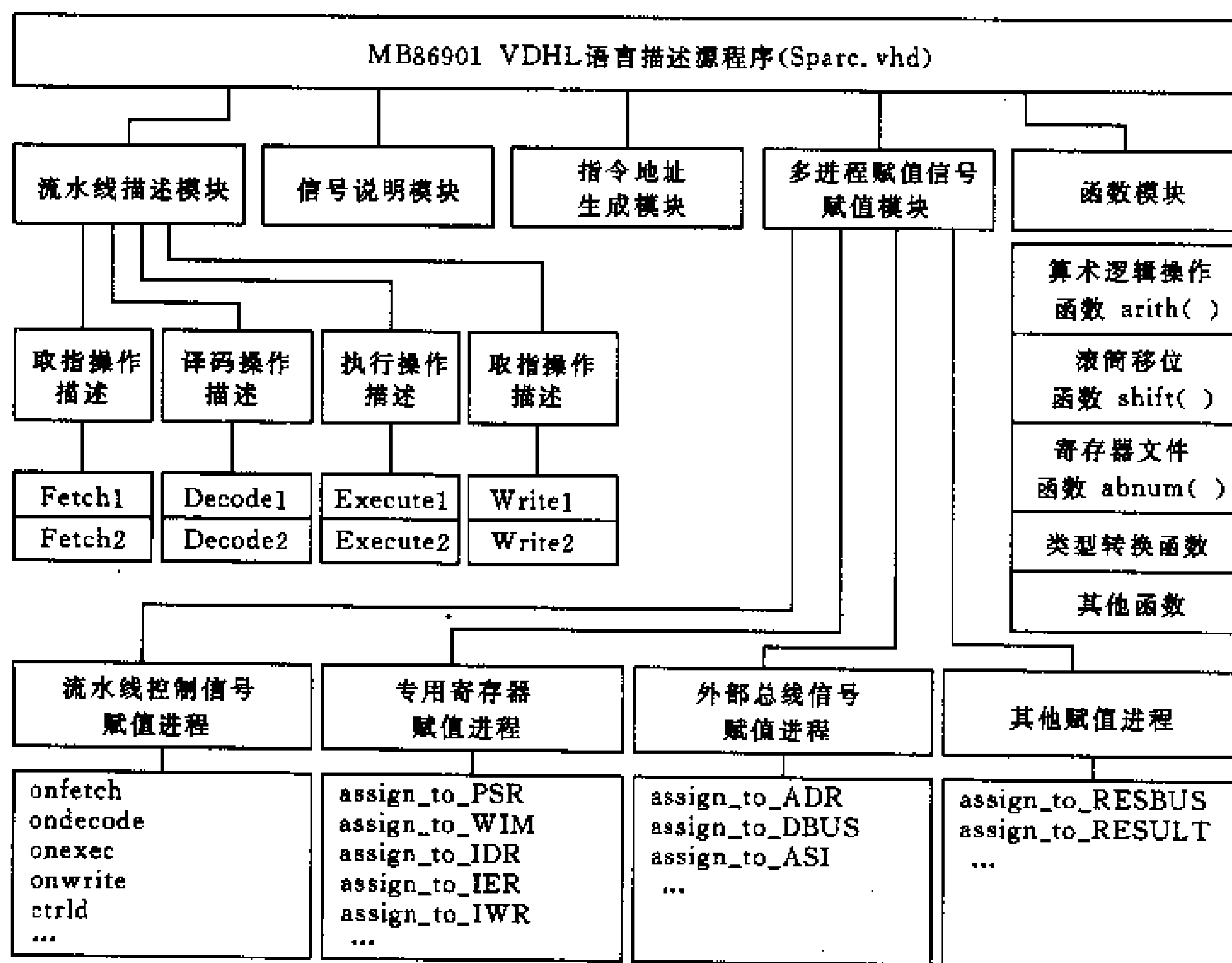


图 100.1　MB86901 流水线行为描述源程序 Sparc.vhd 结构图

2. 实现的功能

MB86901 有指令 95 条，分为数据传送、算术逻辑运算（包括移位）、控制转移、寄存器读写等 4 类。由于时间有限，Sparc.vhd 覆盖了其中的 63 条指令，对与浮点运算有关的指令及多 CPU 指令没有描述。另外，由于建立模型时侧重于对多级流水线的行为描述，因此，此处没有描述 MB86901 的许多接口信号，被描述的具体指令见表 100.1。

表 100.1

OP	OP2	OP3	Mnemonic	OP	OP2	OP3	Mnemonic	OP	OP2	OP3	Mnemonic
00	010		BICC	10		25	SLL	11		09	LDSB
01			CALL	10		26	SRL	11		0A	LDSH
		00	ADD	10		27	SRA	11			
10		01*	AND	10		28	RDY	11		10	LDA
10		02*	OR	10		29	RDPSR	11		11	LDUBA
10		03*	XOR	10		2A	RDWIN	11		12	LDUHA
10		04*	SUB	10		2B	RDTBR	11		13	LDDA
10		05*	AND	10				11		14	STA
10		06*	ORN	10		30	WRY	11		15	STBA
10		07*	XNOR	10		31	WRPSR	11		16	STHA
				10		32	WRWIM	11		17	STDA
10		09*	SUBX	10		33	WRTBR	11			
10		10	ADDCC	10				11		19	LDSBA
10		11*	ANDCC	10		38	JMPL	11		1A	LDSHA
10		12*	ORCC	10		39	RET	11			
10		13*	XORCC	10		3A	TICC	11		20	LDF
10		14*	STA	10				11		21	LDDF
10		15*	ANDNCC	10		3C	SAVE	11		22	
10		16*	ORNCC	10		3D	RESTORE	11		23	STF
10		17*	XNORCC					11		24	STD
10		18	ADDXCC					11			
10				11		00	LD	11		26	STDFQ
10		1C	SUBXCC	11		01	LDUB	11		27	STDF
10				11		02	LDUH				
10		20	TADDCC	11		03	LDD				
10		21	TSUBCC	11		04	ST				
10		22	TADDCCTV	11		05	STB				
10		23	TSUBCCTV	11		06	STH				
10		24	MULSCC	11		07	STD				

3. 关于操作的微时序问题

在流水线结构的行为描述中，由于多周期指令的存在，同一周期内不同进程的两个操作之间有时必须存在顺序关系。比如，在上述 MB86901 的流水线行为描述中，从数据总线读取当前指令时要先判断 IDR 中的上一条指令是否为多周期指令，然后才能根据判断结果进行不同的操作。在 VHDL 标准语法中，通常情况下各进程间，尤其是在各进程内部各语句的执行上（此处指操作生效），不一定存在确定的时序关系。然而，由于在本描述方法中各进程分别由具有确定时序关系的 CLK1 和 CLK2 同步，因此这类问题可这样解决：在 DECODE1 中，通过译码判断此条指令是否为多周期指令，然后设置标识信息(通过 VHDL 的子语句 "**after** CLK2'上升沿"加以约束，使其赋值在 CLK2 上升沿前生效)，与此信息有关的取指操作安排在 FETCH2 中(发生在 CLK2 上升沿后)，即在 CLK2 上升沿读取标识的值（此时标识的值必定有效），根据其值的不同而执行不同的操作。值得注意的是，一般的 VHDL 语言高层次综合系统通常是不支持这种行为描述的，在综合时可能要去掉 **after** 子句，通过多次交互，甚至需要直接修改综合结果。

4. 关于信号的多进程赋值

在流水线的行为描述过程中，许多信号和共用资源需要进行多进程赋值，在 VHDL 语言中通常把这类信号定义为分辨信号，然而由于多级流水线结构十分复杂，如把某些信号定义为分辨信号，则分辨函数的编制十分困难。这时可采取如下方法。比如，fetch1、decode1 和 decode2 这 3 个进程要在不同的时刻对信号 ldcount 赋值，则另定义 3 个分别只在上述 3 个进程中赋值的信号 ldcount_f1，ldcount_d1 和 ldcount_d2。另外编写一个对 ldcount 赋值的进程，该进程由这 3 个信号的属性 TRANSACTION 激活（只要信号被赋值，无论它的值是否发生改变，TRANSACTION 的值均发生变化）。在该进程中，判断上述 3 个信号中哪个信号的 TRANSACTION 属性发生变化（利用 TRANSACTION 的 EVENT 属性，只有当 TRANSACTION 的值变化时，EVENT 才变化），然后把 TRANSACTION 属性变化的信号值赋给 ldcount，这样就可以解决这个问题。具体描述如下。

```
fetch1:  process
begin
     ……
     ldcount_f1<=1 after time1;
      ……
end process fetch1;

fetch1: process
begin
     ……
```

```
    ldcount_f1<=1 after time1;
    ......
end process fetch1;

fetch1:  process
begin
     ......
     ldcount_f1<=1   after  time1;
     ......
end process fetch1;

assign_to_ldcount:  process
begin wait on        ldcount_d1' TRANSACTION, ldcount_f1' TRANSACTION,
                     ldcount_d2' TRANSACTION;
if   ldcount_f1'TRANSACTION'EVENT
    then  ldcount<=ldcount_f1;
elsif     ldcount_d1' TRANSACTION' EVENT
    then  ldcount<=ldcount_d1;
elsif     ldcount_d2' TRANSACTION'EVENT
    then  ldcount<=ldcount_d2;
end if;
end process assign_to_ldcount;
```

5. 模型的模拟验证

模型初步建立后需要对模型进行模拟验证，整个验证工作是在 Talent 系统的模拟器上进行的。由于在 MB86901 中对许多数据相关问题采用的是全硬件的解决方案，这使得本来就很复杂的 CPU 行为模型验证工作变得更加困难。在编制测试码的过程中需要考虑各种复杂的可能情况，并对运行结果进行推算。具体的运行调试过程也是十分复杂的。由于篇幅有限，此处仅介绍 MB86901 中纯粹单周期指令和一些指令数据相关的测试。

(1) 测试码的描述

sparc.vhd 的测试台实际上是 MB86901 外围电路功能的简单行为描述，该描述主要由两个进程组成。一个进程根据 MB86901 行为模型（以下简称模型）的输入输出信号及 MB86901 外部管脚时序图完成模型与测试台之间的信息交互，这些交互主要包括：

① 从用户指令空间（UserInstSpace）或系统指令空间（SysInstSpace）读取指令并在适当的时刻放置到数据总线上；

② 从用户数据空间（UserDataSpace）及系统数据空间（SysDataSpace）读数据并在适当时刻放置到数据总线，或在适当时刻从数据总线读取数据然后放置到两个数

据空间；

③ 在适当的时刻和条件下将数据总线置为高阻，释放数据总线以便模型内部对其赋值；

④ 在适当的时刻和条件下更新其他必要的端口线。

另一个进程的功能是模型外部同步时钟信号 CLK1 和 CLK2，两者的占空比分别是 75% 和 25%。

另外，在系统指令空间和用户指令空间（由 VHDL 的数组描述）中放置一系列 32 位二进制位串（由 4 个 8 位串组成）的测试码。这里测试码实际上是一些 MB86901 机器码指令，在这些机器码后面有一些注释行，标明该机器码助记符及由于该指令的执行，模型内部某个寄存器或信号在某个固定时刻的预测值。如果模型运行至该时刻时，符合该指令后的注释则说明模型运行正确，否则模型或测试台描述有错。在用户数据空间和系统数据空间（由 VHDL 的数组描述）中预先放置一些常数，以便必要时使用。测试台基本框架如下：

```
subtype  ByteVector  is  STD_ULOGIC_VECTOR(7  downto  0);
type MemoryByte  is  array(0  to  799)  of  ByteVector;
variable SysInstspace : MemoryByte :=
(    "10000000",   "00011000",   "00000000",   "00000000",
     --   这是测试初始化
     --   0:   XOR  r0, r0, r0
     --   300nS      [r0]= "00000000_00000000_00000000_00000000"
     "10000000",   "00010000",   "00100000",   "00000001",
     --   4:   OR   r0,  r0,   01H;
     --   360nS      [r0]= "00000000_00000000_00000000_00000001"
...... ...... ...... ......
);
variable UserInstspace  : MemoryByte := (  ...... ...... ...... ......  );
driver : process
begin
     wait on  CLK1  until  (RESET='1');
     if (CLK1 上升沿)
     then if(RD='0')and(WE='0') then   —ST 指令（写数据到存储器）
       case  ASI  is
         when "00001011"    --监控数据空间
           =>if    (SIZE(1 downto 0)="11") or  --双字长
                   (SIZE(1 downto 0)="10")     - 单字长
                   then
                     if (ADR(1 downto 0)="00")  then --STD 类指令
                        SysDataSpace(ADR)<=DBUS(31 downto 24)after 0 ns;
                        SysDataSpace(ADR+1)<=DBUS(23 downto 16)    after 0 ns;
```

```
                    SysDataSpace(ADR+2)<=DBUS(15 downto 8)     after 0 ns;
                    SysDataSpace(ADR+3)<=DBUS(7 downto 0) after 0 ns;
                 else  assert  FALSE  report "Addressing Fail !";
                 end if;
          elsif SIZE(1 downto 0)="01"  then     --半字长
             if (ADR(1 downto 0)="00")or(ADR(1 downto 0)="10")      --STH 指令
             then SysDataSpace(ADR)<=DBUS(15 downto 8)    after 0 ns;
                   SysDataSpace(ADR+1)<=DBUS(7 downto 0) after 0 ns;
             else  assert  FALSE  report  "Addressing Fail !";
             end if;
          elsif     t_SIZE(1 downto 0)="00"  then     --STB 类指令
                   SysDataSpace(ADR)<=DBUS(7 downto 0)    after 0 ns;
          end if;
     when "00001010"     --监控数据空间
          =>与上述类似处理... ...
     when others=> assert  FALSE  report  "Error Data Space! "
end case;
DBUS <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" after 13 ns;
else if   (RD='1')and(WE='1') then     --取指或 LD 类指令
           case t_ASI is
              when "00001001"
                   => DBUS <=     SysInstSpace(ADRh) &
                                  SysInstSpace(ADRh+1)
                                  &SysInstSpace(ADRh+2)&
                                  SysInstSpace(ADRh+3) after 0 ns;
              when "00001000"、"00001011"、"00001010"
                   =>与上述类似处理......
              when others
                    => assert FALSE  report "Invalid Space !";
          end case;
     else assert FALSE report "Warning:  Nothing to do !";
     end if;
end if;
end process driver;

clock : process
variable counter : integer := 0;
begin
wait on onclock until onclock='1' ;
for  I  in  0  TO  400  loop
     if   counter=0 thenCLK1 <= '1'  after 0 ns;
                        CLK2 <= '0'  after 0 ns;
```

```
        elsif counter=1 then CLK2 <= '1' after 0 ns;
        elsif counter=3 then CLK1 <= '0' after 0 ns;
        end if;
        if counter=3 then counter := 0;
        else counter := counter+1;
        end if;
        wait for 15 ns;
    end LOOP;
    end process clock;
```

(3) 测试码的编写

① 单周期指令的测试

在整个模型验证过程中，纯粹的单周期指令运行情况的测试最为简单，这主要是因为除了有时要考虑操作数相关外，各条指令的操作及操作结果可以安排得不相关。因此，如果某条指令的运行出现错误，只需分析该指令的执行过程，而与其他指令的运行过程无关。MB86901 的单周期指令主要包括算术及逻辑运算指令、寄存器读写指令、移位指令。在验证算术及逻辑运算指令时要分别考虑到前后两条指令的两个操作数可能存在数据相关的情况。关于单周期指令位于其他多周期指令后的情况，则放在多周期指令数据相关的测试中进行。本例在 Talent 系统上模拟通过，说明描述是正确的。

```
............
"10000110", "00011000", "01000000", "00000010",
        --12:    XOR  r3,  r1,  r2
        --480nS  [r3]= "00000000_00000000_00000100_01000100"
"10001000", "00110000", "11000000", "00000010",
        --16:    ORN  r4,  r3,  r2
        --540nS  [r4]= "00000000_00000000_00000001_00010001"
"10001010", "00101000", "10000000", "00000001",
        --20:    ANDN r5,  r2,  r1
        --600nS  [r5]= "00000000_00000000_00000101_01010101"
"10001100", "00111000", "01000000", "00000010",
        --24:    XNOR r6,  r1,  r2
        --660nS  [r6]= "11111111_11111111_11111011_10111011"
"10001000", "10010000", "00000000", "00000100",
        --28:    Orcc r4,  r0,  r4;
        --720nS  [r4]= "00000000_00000000_00000001_00010001"
        --660nS  PSR[23:22]="00"  (nz=00)

............
"10000110", "10011000", "11000000", "00000011",
```

```
        --52:     XORcc     r3,  r3,  r3;
        --1020nS  [r3]="00000000_00000000_00000000_00000000"
        --960nS   PSR[23:22]="01"  (nz=00)
"10000000",  "10110000",  "00000000",  "00000000",
        --56:     ORNcc     r0,  r0,  r0;
        --1080nS  [r0]= "00000000_00000000_00000100_01000100"
        --1020nS  PSR[23:22]="00"  (nz=10)
"10000000",  "10110001",  "10100110",  "11001111",
        --60:     ORNcc     r0,  r6,  "0011011001111";
        --1140nS  [r0]= "00000000_00000000_00000000_00000000"
        --1080nS  PSR[23:22]="01"  (nz=00)
"10000010",  "10101000",  "00000000",  "00000000",
        --64:     ANDNcc    r1,  r0,  r0;
        --1200nS  [r1]= "11111111_11111111_11111111_11111111"
        --1140nS  PSR[23:22]="00"  (nz=10)
"10000100",  "10101000",  "01000001",  "00000001",
        --68:     ANDNcc    r2,  r1,  r1;
        --1260nS  [r2]= "00000000_00000000_00000000_00000000"
        --1200nS  PSR[23:22]="01"  (nz=01)
```

② 多周期指令的测试

MB86901 中具有不同流水线流程的多周期指令很多，主要包括 LD 类指令、LDD 类指令、ST 类指令、STD 类指令、BICC 类指令、JMPL 类指令、TICC 类指令、RET 指令和 MULSCC 指令。测试时需要考虑到各类指令在不同条件下的运行情况，比如对 BICC 指令，就要考虑到 16 种不同整数条件码的执行。

由于多周期指令的运行流程与该指令前后的指令都有关，因此还要考虑连续若干条多周期指令的情形。不仅要考虑到连续的同类多周期指令，还要考虑连续的不同类指令的执行。这里我们仅介绍验证单条多周期指令（以 LOAD 指令为例）的执行情况，即每两条多周期指令间均有若干条单周期指令进行隔离，其他情况从略。这些指令的情况也在 Talent 系统上模拟通过，说明描述是正确的。

```
……
"10000000",  "00011000",  "00000000",  "00000000",
--0: XOR  r0,  r0,  r0
--300nS   [r0]= "00000000_00000000_00000000_00000000"
"10000000",  "00011000",  "00000000",  "00000000",
--4: XOR  r0,  r0,  r0
--360nS   [r0]= "00000000_00000000_00000000_00000000"
"11000010",  "00000000",  "00100000",  "00000000",
--8: LD   r1,  r0,  "0000000000000"
--480nS   [r1]= "00000000_11111111_00000001_00000000"
```

```
"10000000",  "00011000",  "00000000",  "00000000",
--12:     XOR  r0,  r0,  r0
--540nS   [r0]= "00000000_00000000_00000000_00000000"
"10000000",  "00011000",  "00000000",  "00000000",
--16:     XOR  r0,  r0,  r0
--600nS   [r0]= "00000000_00000000_00000000_00000000"
"10000000",  "00011000",  "00000000",  "00000000",
--20:     XOR  r0,  r0,  r0
--660nS   [r0]= "00000000_00000000_00000000_00000000"
"10000000",  "00011000",  "00000000",  "00000000",
--24:     XOR  r0,  r0,  r0
--720nS   [r0]= "00000000_00000000_00000000_00000000"
......

"11000100",  "00000000",  "00100000",  "00000100",      --&4
--28:     LD   r2,  r0,  "0000000000100"
--840nS   [r2]= "00000000_11111111_00000001_00000000"
"10000100",  "00000000",  "00100000",  "00010100",  --r2=20
--32:     ADD  r2,  r0,  "0000000010100"
--900nS   [r2]= "00000000_00000000_00000000_00010100"
"10000000",  "00011000",  "00000000",  "00000000",
--36:     XOR  r0,  r0,  r0
--960nS   [r0]= "00000000_00000000_00000000_00000000"
"10000000",  "00011000",  "00000000",  "00000000",
--40:     XOR  r0,  r0,  r0
--1020nS  [r0]= "00000000_00000000_00000000_00000000"
"10000000",  "00011000",  "00000000",  "00000000",
--44:     XOR  r0,  r0,  r0
--1080nS  [r0]= "00000000_00000000_00000000_00000000"
-- 以上是带 LDLOCK 的 LD 指令
......
```

参 考 文 献

1. 刘明业，蒋敬旗，张东晓. EDA 发展新动向——第 35 届设计自动化会议见闻录，《计算机世界》，1998 年 11 月 23 日及《集成电路设计》，1999 年第一期
2. 刘明业，叶梅龙，张东晓，李雁. 专用集成电路高级综合理论. 北京：北京理工大学出版社，1998
3. 刘明业，叶梅龙等. 数字系统自动设计. 北京：高等教育出版社，1996
4. 刘明业，张东晓，许庆平，VHDL 高级综合系统设计中某些关键问题的技术决策，计算机学报，1997，20（6）：501~509
5. 张东晓，刘明业. VHDL 语言高级综合子集的确定及其实现方法. 计算机学报，1997，20（3）：198~205
6. 叶梅龙，张东晓，恒东辉. 高级综合中控制流信息的提取与综合. 软件学报，1997，8(11)，857~863
7. Zongfu Yan，Mingye Liu，etal. The Intelligent Approach to Register Transfer Level Synthesis. AI in Engineering，1998，12(3)：143~147
8. 马聪，刘明业等. 高级综合与底层物理设计实现的衔接，电子学报，1998，26（2）
9. JingYan Zuo，MingYe Liu. Automatic Generation of Schematic Diagrams in High-level Synthesis. Journal of Beijing Institute of Technology，1995，4(2)：188~197
10. 李雁，刘明业等. VHDL 模拟器的设计. 《全国第九届 CAD 与图形学学术会议论文集》，76~81，青岛，1996 年 9 月
11. 杨勋，刘明业. VHDL 混合级模拟器中一种新的信号值计算方法。《全国第九届 CAD 与图形学学术会议论文集》，82~87，青岛，1996 年 9 月
12. 刘明业，张东晓，李雁. 日新月异 EDA 技术与 VHDL 语言. 计算机世界，1996（47）：105~107
13. J M Berge，A Fonkoua，S Maginot，J Rouillard. VHDL Designer's Reference. Kluwer Academic Publishers，1992
14. Roger Lipsett，Carlf Schaefer，Cary Vssery. VHDL:Hardware Description and Design. Kluwer Academic Publishers，1989
15. IEEE Standard VHDL Language Reference Manual. IEEE Std 1076~1987
16. IEEE Standard VHDL Language Reference Manual. IEEE Std 1076~1993
17. Petru Eles，Krzysztof Kuchcinski，Zebo Peng. System Synthesis with VHDL. Kluwer Academic Publishers，1998
18. Ahmed Amine Jerraya，Hong Ding，Polen Kission，Maher Rahmouni. Behavioral Synthesis and Component Reuse with VHDL. Kluwer Academic Publishers，1997

附录Ⅰ 100例内容摘要一览表

叶梅龙 袁 媛

例号	内容摘要	文件名
1	设计实体、实体说明、结构体、进程、**if** 语句、位向量、重载 用进程描述行为功能。	1_adder.vhd
2	敏感信号、延迟 用进程描述行为功能	2_adder.vhd
3	与第1例类似	3_mul.vhd
4	变量与信号的定义、区别、赋值，为何不用信号存储数据，函数与函数的调用 用进程描述行为功能	4_comp.vhd
5	以二路选择器为例说明并发条件赋值语句 **when** 与给端口信号赋值的进程语句等价 用进程描述行为功能	5_mux2.vhd
6	**wait** 语句的4种形式、属性 EVENT 和 STABLE、时钟建模 用进程描述行为功能	6_reg.vhd
7	块语句 **block**、被保护的块、毗连符号、七值逻辑、有关测试码及其编写、断言语句 **assert** 的引用 用进程描述行为功能	7_shiftreg.vhd 7_test_vector.vhd
8	功能单元库 bit_rtl_lib、包 bitpkg、包的意义及引用、库的引用、重载函数"+"、"-"、"*"、"/"、底层单元为何用行为描述 采用行为描述	8_bitpkg.vhd 8_bit_rtl_lib.vhd
9	七值逻辑的定义、标量类型、复合数据类型、寻址类型、文件类型、文件类型说明、子类型、为何自定义数据类型	9_MVL7_types.vhd
10	bit_to_int 函数的功能、别名在数组及指令中的应用、预定义属性 LENGTH、**for** 循环 采用行为描述	10_bit_to_int.vhd
11	七值线或分辨函数、信号驱动源、分辨信号、分辨函数、七值逻辑分辨函数、属性 RANGE、**exit** 语句	11_wiredor.vhd
12	整型值转换为七值逻辑位向量、转换函数概念介绍、**while** 循环、**case** 语句 采用行为描述	12_convert.vhd
13	七值逻辑位向量的左移函数功能、属性 HIGH 和 LOW 采用行为描述	13_shl.vhd
14	包含典型逻辑函数：左移、右移、循环移位、整数转换为七值逻辑位向量及反之、取反、取补、(七值位向量)"+"、"-"、"*"的重载、带进位的七值位向量相加、增1、奇偶校验等等	14_MVL7_function.vhd

续表

例号	内 容 提 要	文 件 名
	采用行为描述	
15	以七值逻辑四输入多路器为例，用行为描述方式以模块的形式进行描述	15_mux41.hvd 15_types.vhd 15_test_vectors_mux41.vhd
16	用并发条件信号赋值语句进行行为描述并举例说明一条并发语句与一个进程的对应关系 断言语句的作用	16_multiple_mux.vhd 16_types.vhd 16_MVL7_functions.vhd 16_test_vectors.vhd
17	以结构描述方式进行描述，涉及元件例示、组装规定等 以进程描述奇偶校验器的行为，用 **if** 语句 测试台的结构描述	17_parity.vhd 17_test_parity.vhd 17_test_bench.vhd
18	映射单元库及其使用举例 以基于逻辑表达式的数据流风格描述设计实体 涉及整型数据、位型数据间的转换函数的引用 测试台采用结构描述	18_tech_lib.vhd 18_test_lib.vhd
19	循环边界常数化测试 不涉及具体电路，以空实体说明，结构体中以 **for** 循环说明需要说明的（行为）语法现象 采用行为描述	19_test_194.vhd
20	保留字、属性不能用作用户的标识符	20_test_159.vhd
21	死锁的概念 用两个进程说明死锁的产生 采用行为描述	21_test_13a.vhd
22	给出编写不当形成振荡与死锁的描述 以进程描述该语法现象，用 **if**，**wait**，**assert** 语句进行描述 采用行为描述	22_deaklock.vhd
23	阐述振荡产生的原因并使用结构描述法描述振荡电路具体结构，用进程描述触发器的翻转 测试台采用结构描述	23_test_120.vhd
24	采用行为及结构混合描述，阐明分辨信号、分辨函数	24_test_195.vhd
25	信号驱动源、分辨信号可有多个驱动源、循环语句为信号建立多个驱动源	25_test_1.vhd 25_test_1a_vhd
26	同 19 例，仅说明某语法现象，说明分辨函数对多驱动源的选择作用 属性 TRANSACTION 的应用 该例以四值逻辑进程形式进行讨论	26_test_74s.vhd
27	块保护及属性 EVENT、STABLE 同第 19 例，仅说明语法现象，用了块保护	27_test_16.vhd
28	属性 EVENT，ACTIVE，LAST_VALUE，DELAYED，STABLE[(t)]，QUIT[(t)]，TRANSACTION 的介绍	28_test_64a.vhd
29	阐明进程、并发语句二者之间的并发关系	29_test_35.vhd
30	用进程模拟通信举例	30_test_3.vhd

续表

例号	内 容 摘 要	文 件 名
31	通过预定义属性 QUIET 进行中断处理优先机制建模 采用进程描述	31_test_35b.vhd
32	过程中不允许直接建立信号驱动源。以进程形式实现	32_test_110b.vhd
33	从算法级对比较器进行描述，介绍库、设计实体以及进程的基本概念	33_comparer.vhd 33_comp.vhd 33_simu.vhd
34	从总线读取数据	34_bus_readwrite.vhd
35	描述基于总线的数据通道，主要说明如何设计一个总线接口、如何设计控制总线的有限状态自动机、如何设计总线的时序以保证数据通道的有效性、如何设计控制器电路以及如何将系统的各个组件有机地、合理地组装在一起	35_486_sys.vhd 35_486_bus.vhd 35_bus_test.vhd 35_ram_controller.vhd 35_bit_pack.vhd
36	描述一个基于多路器的数据通道，说明如何利用一个多路器在外部端口和寄存器、功能单元之间传输数据	36_gcd.vhd 36_test.vhd
37	是测试示例，主要说明四值逻辑及其分辨(给出 A，B 两驱动值不同组合时的分辨值表)	37_test_105.vhd
38	四值逻辑向量按位或运算，利用了属性	38_test_28.vhd
39	生成语句的格式 结构描述中，生成语句在元件例示中的应用	39_wst0dp.vhd
40	通过带类属的译码器说明为何用类属来规定端口以及在元件例示、组装时需要注意的问题 结构描述生成语句	40_generic_dec.vhd
41	带类属的测试平台，在组装语句中指定类属值	41_generic_testbench.vhd
42	以一个多路选择器为例说明底层元件用结构描述，然后组装到多路器中，再用行为与结构的混和描述说明多路器的功能	42_mix.vhd
43	为时序设计的例子，可加载数据，实现左移或右移功能。同时介绍 VHDL 的主要数据类型和子类型 用进程描述行为	43_shift_reg.vhd 43_test_register.vhd
44	减 1 计数器，以块语句描述其功能	44_reg_counter.vhd 44_test_vector.vhd
45	同第 19 例，仅说明某语法现象，注意模拟运行时设置断点以防止死循环 以进程描述该语法现象	45_test_63.vhd
46	用类属说明延迟，类属值在结构体的元件说明的指定，即延迟值在实体说明中缺省，元件说明中的 **generic**，元件例示中及组装说明中若再用 **generic map** 则无效。若元件说明中未指定延迟值，则可在元件例示或组装说明中使用 **generic map** 指定延迟值，且在元件说明中不必用 **generic**	46_default_generic.vhd
47	以一常量元件作为无输入元件进行讨论。使其与某电路连接，通过某电路工作是否正常间接验证无输入元件工作是否正常	47_const_test.vhd

续表

例号	内容摘要	文件名
48	测试码及其编写、VHDL 惯性延迟、顺序和并发语句、属性 TRANSACTION	48_test_18e.vhd
49	引入 delta 的作用及功能。以变量赋值、信号赋值为例，说明 delta 延迟的意义 用进程进行描述	49_delta.vhd
50	事务与事件的区别 行为描述（进程、并发信号赋值）	50_test_18b.vhd
51	传输延迟驱动优先 同第 19 例，仅说明某语法现象，以行为描述说明测试台。	51_test_113.vhd
52	用进程描述分频器功能。 **case** 语句格式。	52_divider.vhd 52_divider_stim.vhd
53	测试平台允许实时交互。建立测试平台有 3 种方法：①嵌入测试平台中的测试向量表；②利用包含测试向量的独立文件；③借助算法对实际输出与预期输出相比较。本例采用第 1 种方法 三位计数器用行为描述来描述其功能	53_counter.vhd 53_counter_testbench.vhd
54	分秒计数器显示器的描述 采用结构描述	54_display.vhd 54_display_stim.vhd
55	地址计数器 采用结构描述	55_falsepath.vhd 55_falsepath_stim.vhd
56	采用纯行为描述来描述指令预读计数器 并有测试台描述	56_prefetch.vhd 56_stim.vhd 56_vhdl.vhd
57	以加、减、乘三条指令为例阐述指令译码器的工作过程 用选择型条件信号赋值语句描述	57_instruction_dec.vhd
58	行为、数据流、结构三种描述风格的比较，还可用混合式描述 给出 2-4 译码器完整的结构描述（包含测试台） 元件说明、元件例示、组装规定的格式	58_decoder.vhd 58_decoder_stim.vhd
59	2-4 译码器行为描述方法 第 58 例为结构描述，可与之比较	59_decoder.vhd
60	转换函数进行类型转换时，一定要转换驱动源 命名关联、位置关联	无源描述
61	基于同一基类型的两分辨子类型赋值相容问题 同一基类型的两分辨子类型向量信号间的相互赋值可通过进程语句中的循环语句实现	61_assign.vhd
62	a、b、c 为 3 个整数，若 a-b=c，则 a 与 b 的最大公约数等于 b 与 c 的最大公约数，依据此原理写描述 讲述了 **exit** 的格式	62_gcd.vhd 62_gcd_stim.vhd
63	最大公约数七段显示器编码 利用 **case** 语句得到编码	63_gcd_disp.vhd 63_vhdl.vhd 63_stim.vhd
64	交通灯控制器，用 **case** 语句完成控制转换的描述	64_tlc.vhd 64_test_vectors.vhd

续表

例号	内 容 摘 要	文 件 名
65	空调系统有限状态自动机 行为描述，使用自定义属性	65_conditioner.vhd 65_conditioner_stim.vhd
66	FIR 滤波器 库语句、use 子句的应用、FIR 滤波器采样值的确定	66_signed.vhd 66_pack.vhd
	用行为描述实现 FIR 滤波器的功能	66_fir.vhd 66_testfir.vhd
67	五阶椭圆滤波器 是一个标准的 Bench-Mark，use 子句打开当前库 work 用进程描述五阶椭圆滤波器的功能	67_pack.vhd 67_ellipf.vhd 67_test_vector.vhd
68	带闹钟功能的计时器的控制器部分 分别以 Moore、Mealy 机阐明控制器设计的不同，皆以进程实现	68_alarm_controller.vhd 68_tb_alarm_controller.vhd
69	带闹钟功能的计时器的译码器部分 以并发语句 **with...select** 进行行为描述	69_decoder.vhd 69_p_alarm_clock.vhd 69_tb_decoder.vhd
70	带闹钟功能的计时器的移位寄存器部分 以进程描述移位的行为	70_buffer.vhd 70_tb_buffer.vhd
71	带闹钟功能的计时器的闹钟寄存器部分 以进程描述寄存器功能，测试中用到组装和元件例示	71_alarm_reg.vhd 71_alarm_counter.vhd 71_tb_alarm_counter.vhd 71_tb_alarm_reg.vhd
72	带闹钟功能的计时器的显示驱动器部分 以进程描述计数器功能 测试台涉及元件说明、元件例示及组装	72_display_driver.vhd 72_tb_display_driver.vhd
73	带闹钟功能的计时器的分频器部分 以进程描述计数器功能 测试台涉及元件说明、元件例示及组装	73_fq_divider.vhd 73_tb_fq_divider.vhd
74	带闹钟功能的计时器的整机组装 以结构描述阐述该设计	74_alarm_clock.vhd 74_tb_alarm_clock.vhd
75	描述一个带有时序特性的存储器模型 说明设计思路并以行为描述进行描述	75_ram.vhd
76	电机转速控制器 阐明 PID 为何完成电机转速的控制(通过调整电机的控制电流) 描述中涉及浮点加、减、乘、求倒数的运算 给出在一定激励下的波形图	76_pid.vhd 76_pid_stim.vhd 76_fpu.vhd
77	神经元计算机 结构体采用行为描述 神经元计算机行为模型的建立，而接口描述充分注意到硬件关系，用位、位串表示外部信息及内存、数据总线的传递关系	本例未附源代码
78	将 Am2901 四位微处理器分为 ALU 输入选择、ALU 运算、存储器、寄存器和输出及移位 5 部分 讨论 ALU 输入选择逻辑，操作涉及 RAM 的存取、锁存和多路选择	78_alu_inputs.vhd 78_alu_inputs_stim.vhd

例号	内容摘要	文件名
	定义 L2901_lib 库，其中包含七值逻辑的逻辑移位、求反、求补、奇偶校验、算术运算、线或等运算	
79	阐述 ALU 的算术、逻辑运算，引入进位概念	79_alu.vhd 79_alu_stim.vhd
80	Am2901 四位微处理器的 RAM 在给定地址信号作用下实现数据的输入输出。输入方式分直接、左移、右移输入 3 种。RAM 操作包括：保持、接受和左右移。引入块语句，讨论块的作用	80_mem.vhd 80_mem_stim.vhd
81	Am2901 四位微处理器的寄存器 操作同 RAM，描述亦同 RAM，将整个模块定义成一个块结构，保护表达式为真时执行块内语句 并非是通常用进程结构描述时序的方法	81_Q_reg.vhd 81_Q_reg_stim.vhd
82	Am2901 四位微处理器的输出与移位 数据在 ALU 处理后或由 RAM 读出时皆通过三态总线结构输出。输出和移位皆靠条件信号赋值语句完成	82_output_and_shifter.vhd 82_output_shifter_stim.vhd
83	Am2910 四位微处理器的多路选择器 将 Am2910 四位微处理器分为 5 部分：多路选择器、寄存器/计数器、微程序计数器/寄存器、堆栈及堆栈指针寄存器和指令译码器 本例讨论多路选择器功能用于选择下条欲执行指令地址。地址选择共 4 种，用块语句描述	83_multiplexer.vhd 83_multiplexer_stim.vhd
84	Am2910 四位微处理器的计数器/寄存器 计数/寄存器执行预置、递减、保持这 3 个功能 用块语句描述	84_reg.vhd 84_reg_stim.vhd
85	Am2910 四位微处理器的指令计数器 功能：clear = 1 时指令计数器清 “0”， clear = 0 时地址数 Y_temp 的值与 CI 相加 用块语句实现	85_upc.vhd 85_upc_stim.vhd
86	Am2910 四位微处理器的堆栈 先进后出，包括存取一个操作数、对指针的运算及溢出判断、初始化 用块语句描述	86_stack.vhd 86_stack_stim.vhd
87	Am2910 四位微处理器的指令译码器 用四位指令代码 I3~0 译 16 条指令 用块语句描述译码过程	87_control.vhd 87_control_stim.vhd
88	可控制计数器 分成译码、计数限预置、计数、比较 4 个部分，分别用 4 个进程描述	88_arms_counter.vhd 88_arms_counter_stim.vhd 88_pack_2_0.vhd
89	四位超前进位加法器 用到进位生成和进位传输逻辑，从一位全加器逻辑表达式推得该加法器的逻辑关系，基于其表达式写出行为描述	89_full_adder.vhd 89_full_adder_stim.vhd 89_pack_2_0.vhd

续表

例号	内容摘要	文件名
90	描述窗口搜索系统的协同处理器部分，主要说明协同处理器的设计思路以及描述方法	90_wss_coprocessor.vhd
91	描述窗口搜索系统的序列存储器部分，主要说明序列存储器的设计思路与描述方法	91_wss_sequence.vhd
92	描述窗口搜索系统的字符串存储器部分，主要说明字符串存储器的设计思路与描述方法	92_wss_stringreg.vhd
93	描述窗口搜索系统的顶层控制器部分，说明整个窗口搜索系统的控制与组装	93_wss_top.vhd
94	MB86901 流水线行为描述框架 简述流水线工作原理，给出流水线行为描述总框架	
95	MB86901 寄存器文件管理的描述 阐述 MB86901 寄存器的窗口化管理方式，并通过函数 AbsNum( )描述窗口管理的行为	
96	MB86901 内 ALU 的行为描述 MB86901 采用快速 32 位超前进位整数 ALU 函数 RegToULVector 用于从类型 RegType 转换到类型 STD_ULOGIC_VECTOR，使之能够借用 1164 包中定义有关 STD_ULOGIC_VECTOR 信号、变量的加减运算 给出的行为描述仅仅说明 ALU 类器件的描述方法，是较高层次的行为描述，需改写成符合综合要求的格式并加一定的约束才能综合成结构接近 MB86901 的超前进位 ALU	
97	移位指令的行为描述 以函数 shift 形式描述 3 条移位指令的行为	
98	单周期指令的描述 单周期指令的描述很简单，但需注意操作数数据相关的处理	
99	多周期指令的描述 以 LOAD 指令为例说明多周期指令的描述。提出数据相关的概念及互锁、控制相关等概念 以 LOADLOCK 为例介绍数据相关的处理。MB86901 以硬件解决互锁（LOADLOCK）	
100	MB86901 流水线行为模型 MB86901 共有指令 95 条，sparc.vhd 覆盖除与浮点运算有关及多 CPU 指令之外的 63 条指令。以 LD 指令单条执行做多周期指令讨论对象	

附录Ⅱ　VHDL专用术语中英文对照

刘沁楠

一　画

ASIC	Application Specific Integrated Circuit
CASE语句	CASE statement
EXIT语句	EXIT statement
IF语句	IF statement
LOOP语句	LOOP statement
NEXT语句	NEXT statement
NULL语句	NULL statement
RETURN语句	RETURN statement
WAIT语句	WAIT statement
Δ延迟	delta delay

二　画

二元算符	binary operator
二值逻辑	binary valued logic
七值逻辑	seven valued logic
九值逻辑	nine valued logic

三　画

子类型	subtype
子类型说明	subtype declaration
子程序	subprogram
子程序规定	subprogram specification
子程序参量	subprogam parameter
子程序参量关联	subprogram parameter association
子程序形式参量	formal subprogram parameter
子程序说明	subprogram declaration
子程序调用	subprogram call

子程序体	subprogram body
三值逻辑	three valued logic
上界值	upper bound
下界值	lower bound
下标	index, subscript
下标限制	index constraint

四 画

中断	interrupt
互连单元	interconnection unit
元件	component
元件关联	component association
元件例示	component instantiations
元件例示语句	component instantiation statement
元件组装	component configuration
元件表	component table
元件说明	component declaration
引用	reference
分辨信号	resolved signal
分辨函数	resolution function
五元组	five tuple
六值逻辑	six valued logic
六值模拟	six valued simulation
文字	literal
文件对象	file object
文件类型	file type

五 画

四值模拟	four valued simulation
四元式表	four tuple table
当前事件	current event
功能单元	functional unit
包	package
包体	package body

包说明	package declartion
可见性	visibility
未连接端口	unconnected port
外部时钟信号	external clock signal
外部信号	external signal
外部输入激励	external input excitation
生存周期	lifetime intervals
生成语句	generate statement
生成参数	generate parameter
记录元素	record element
记录类型	record type
对象	object
对象断点	object breakpoint
对象说明	object declaration
对象类别	object class

六 画

传输	transfer
传输延迟	transport delay
先驱单元	harbinger unit
延迟	delay
延迟模型	delay modal
关联	association
同步	synchronism
同步电路	synchronous circuit
同步模拟	synchronous simulation
后继	succesor
多驱动源	multi drivers
多值模拟	multiple valued simulation
多值逻辑	multiple valued logic
多重嵌套	iterative nest
多周期操作	multi-cycle operation
多层次模拟	multi-level simulation
多维数组	multi-dimensional array

字面量	literal
并发	concurrent
并发过程调用	concurrent procedure call
并发性	concurrency
并发信号赋值语句	concurrent signal assignment statement
并发条件赋值语句	concurrent conditional assignment statement
并发语句	concurrent statement
并发断言语句	concurrent assertion statement
异步	asynchronous
异步事件	asynchronous event
异步时序电路	asynchronous sequential circuit
自顶向下	top-down
自底向上	bottom-up
行为	behavior
行为描述	behavioral description
行断点	line breakpoint
设计实体	design entity
设计单元	design unit
设计库	design library
设计重用	design reuse
设计描述	design description
死锁	deadlock
访问类型	visiting type
关系算符	relational operator
关联元素	association element
关联表	association list
动态行为	dynamic behaviour
动态冒险	dynamic hazard
自驱动模拟	self driven simulation
约束	constraint
优先级	dominance
过程	procedure
过程调用	procedure call
过程调用语句	procedure call statement
过程规定	procedure specification

七 画

串行语句	sequential statement
串行断言语句	sequential assert statement
串行操作	sequential operation
形式类属	formal generic
形式参数	formal parameter
形式端口	formal port
初值	initial value
别名	alias
块	block
块语句	block statement
块组装	block configuration
层次化设计	level design
层次化结构	level structure
时钟信号	clock signal
时序逻辑	sequential logic
库	library
库子句	library clause
库名	library name
库单元	library unit
完备性	completeness
驱动值	driving value
驱动源	driving source
局部类属	local generic
局部端门	local port
形参	formal parameter
位串	bit string
位向量	bit vector
条件信号赋值	conditional signal assignment
位置关联	positional association
进程	process
进程语句	process statement
严谨性	severity

八 画

函数	function
函数调用	function call
函数说明	function declaration
函数定义	function specification
表达式	expression
事件	event
事件表	event table
事务	transaction
范围限制	range constraint
单目运算符	unary operator
单位延迟时间	unit delay time
组合逻辑	combinational logic
组装	configuration
组装说明	configuration declaration
组装规定	configuration specification
组装语句	configuration statement
命名关联	named association
空	null
空事务处理	null transaction
规则结构	regular structure
参数	parameter
实体	entity
实体外貌	entity aspect
实体说明	entity declaration
实验模型	experiment model
非限定性数组类型	unconstrained array type
物理类型	physical type
物理描述	physical description
变量	variable
变量类	variable class
变量说明	variable declaration
变量赋值语句	variable assignment statement
转换函数	conversion function

枚举类型 enumeration type
波形 waveform
波形元素 waveform elements
抽象描述 abstract description

九 画

保留字 keyword
结构 architecture
结构体 architecture body
结构化设计 structured design
结构描述 structure description
限定性数组类型 constrained array type
选择信号赋值 selected signal assignment
信号 signal
信号说明 signal declaration
信号驱动源 signal drive source
信号类 signal class
信号赋值 signal assignment
信号赋值语句 signal assignment statement
测试平台 test bench
测试向量 test vector
标号 label
标识符 identifier
标识符表 identifier list
标准延迟 standard delay
标准单元 standard element/component
标量类型 scalar type
类属 generic
类属关联 generic association
类属关联表 generic association list
类型 type
类型转换 type conversion
类型说明 type declaration
类型标记 type mark

保护	guard
说明区	declarative regions
响应时间	reaction time
冒险	hazard
指定	specification
重载	overloading
重载操作符	overloading operator
总线	bus
顺序语句	sequential statement

十 画

预定义子类型	predefined subtype
预定义类型	predefined type
预定义运算符	predefined operator
预定义库名	predefined library name
预定义属性	predefined attribute
递归结构	recursive structure
被保护的信号	guarded signal
被保护的赋值	guarded assignment
被保护的信号赋值	guarded signal assignment
扇入	fan in
扇入表	fan in table
扇出	fan out
扇出表	fan out table
浮点文字	float literal
浮点类型	float type
离散类型	discrete type
毗连运算符	concatenation operator
竞争	race

十一画

逻辑名	logical name
逻辑运算符	logical operator
逻辑模拟	logic simulation

接口元素 interface element
接口表 interface list
接口对象 interface object
常数延迟时间 constant delay time
常量 constant
常量说明 constnat declaration
常量类 constant class
敏感信号 sensitive signal
敏感信号表 sensitive signal list
敏感(性) sensitivity
属性 attribute
属性名 attribute name
属性说明 attribute declaration
属性指定 attribute specification
基类型 base type
惯性延迟 inertial delay
描述范畴 description domain
混合模拟 hybrid simulation
断言语句 assertion statement
断点 breakpoint

十二 画

最大(小)延迟时间 maximal/Minimal delay time
硬件描述语言 hardware description language
循环结构 loop constructs
循环语句 loop statement
嵌套 nest

十三 画

数值类型 numeric type
数组 array
数组范围 array bounds
数组类型 array type
数组维数 array dimension

源	source
源描述	source description
零延迟	zero delay
输出函数	output function

十四 画

模式	mode
模块化设计	modularization design
模块化结构	modularization structure
模拟	simulation
模拟周期	simulation cycle
模拟时钟	simulation clock
模拟层次	simulation level
模拟环境	simulation environment
模拟结构	simulation structure
模拟步长	simulation step length
模拟器	simulator
模拟模型	simulation model
模拟波形	simulation waveform
算术运算符	arithmetic operator
算符符号	operator symbol
端口	port
端口说明	port declaration
端口关联	port association
端口关联表	port association list
静态	static
静态冒险	static hazard
静态测试	static test

十六 画

默认(缺省)值	default value
默认(缺省)的端口关联	default port association
整数文字	integer literal
整数类型	integer type

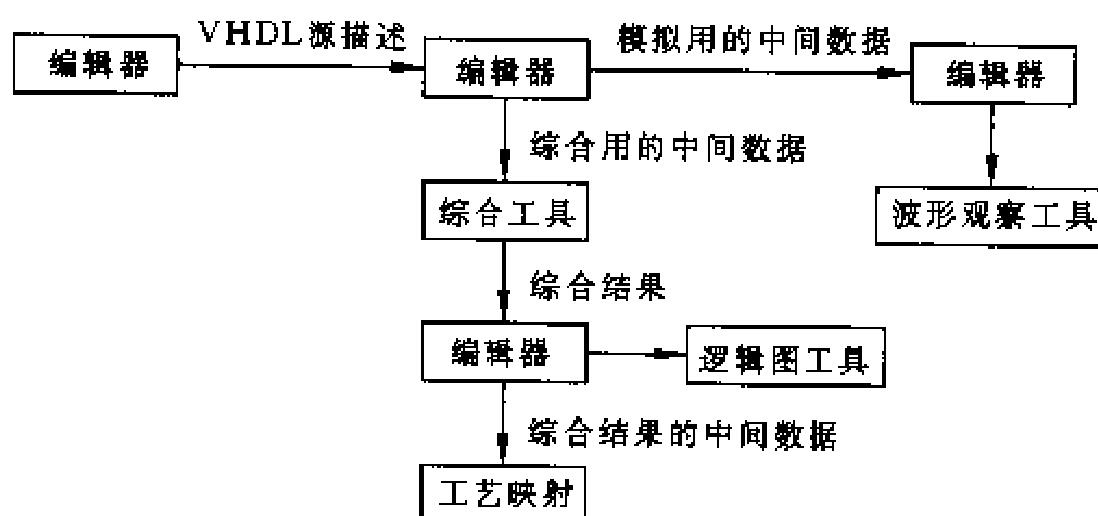
激励	excitation
激励产生器	excitation generator
激励响应	excitation response
激活	active
激活条件	activition condition
操作并发性	operation concurrency
操作数	operand
操作算符	operator

附录III Talent 系统 VHDL 模拟器使用说明

张俭锋

1. 系统概述

Talent 专用集成电路(ASIC)高层次自动综合系统支持 VHDL IEEE STD1993 的 VHDL93 标准，包括了 VHDL 专用编辑器、编译器、模拟器、高级综合工具、工艺映射等子系统，可以完成 VHDL 源描述的输入、编译、模拟、综合以及工艺映射等一系列完整的设计流程。此外，用户还能通过波形工具和逻辑图工具对模拟生成的波形文件和综合生成的逻辑图进行观察和分析。其设计流程如附图III.1 所示。



附图III.1 设计流程

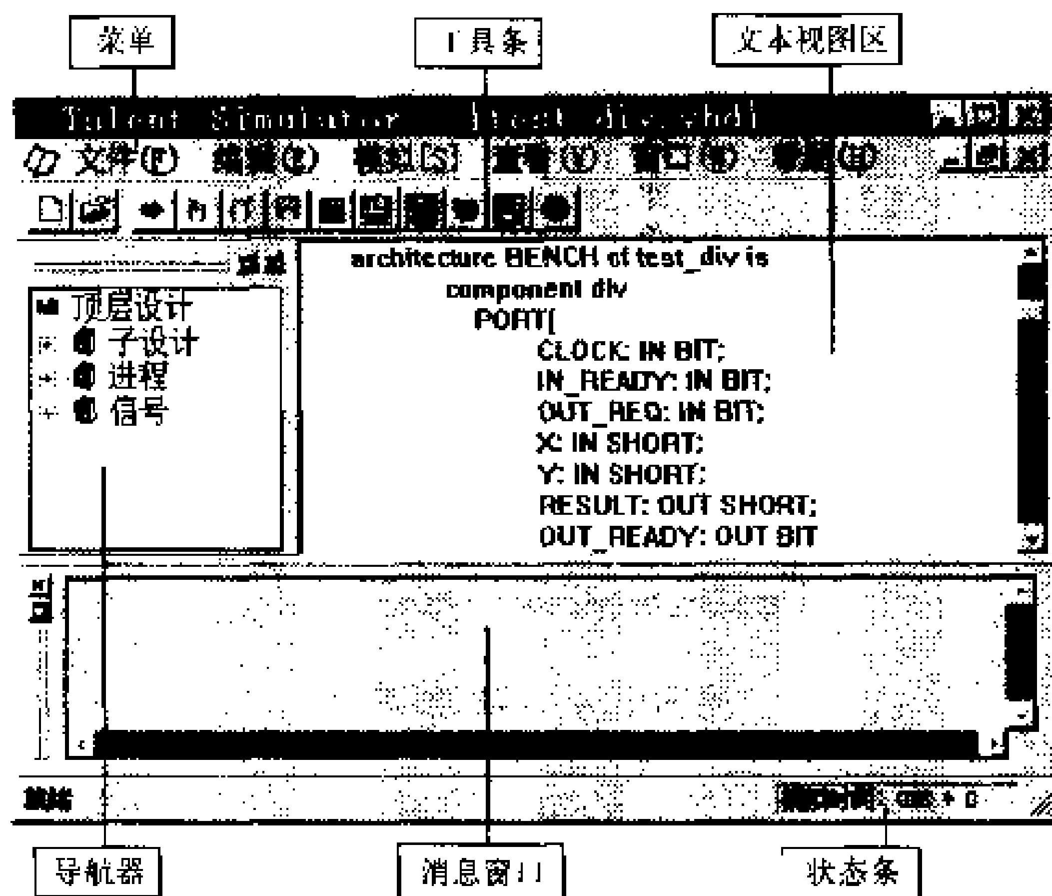
2. 模拟器功能说明

模拟器接收编译器提供的中间数据文件，动态地对设计进行模拟，能够对 VHDL 混合描述（包括行为功能级描述和结构描述）进行模拟验证，对于验证设计的正确性起着至关重要的作用，使得用户能够对设计进行动态的观察和分析。Talent 系统的模拟器是 Windows 下的标准应用程序，具有界面友好、简单易用的特点。此外，模拟核心采用基于周期模拟和事件驱动两种模拟算法，对同步电路和异步电路都能够准确地完成模拟。

3. 模拟器的结构

(1) 模拟器的外观

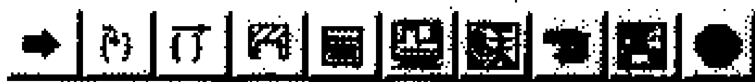
如附图III.2 所示，Talent 模拟器包括如下部分：菜单、工具条、导航器、消息窗口、状态条以及文本视图区。



附图III.2　模拟器结构

(2) 模拟工具条和模拟菜单

模拟工具条为模拟提供了几种常用模拟命令的快捷方法，包括以下按钮：直接执行、单步执行、单步跟踪、断点设置、断点浏览、波形显示、对象跟踪和退出模拟，如附图III.3 所示。



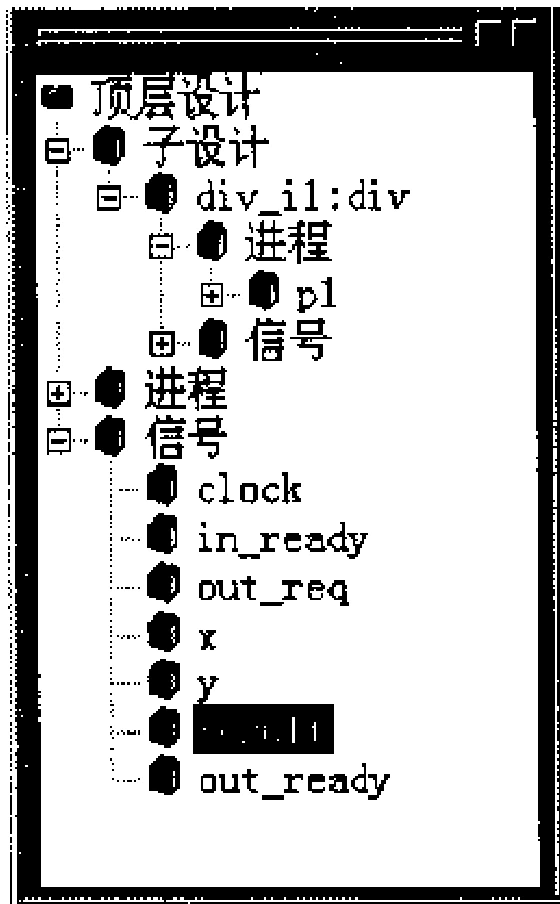
附图III.3　工具条

模拟菜单中包括了模拟工具条中的所有功能，同时还有按周期执行、按时间执行、按相位执行等菜单项。各个按钮和菜单项的功能和使用方式将在下面进行说明。

(3) 导航器窗口

导航器窗口在主窗口的左侧，是模拟时要用到的一个重要窗口。导航器窗口中包含了设计中的所有层次信息，包括设计的层次、每一层中的进程、对象等信息，如附图III.4

所示。



附图III.4　导航器窗口

通过导航器窗口，用户可以对设计的每一层进行检查，浏览其中的进程、信号和变量。用户还可以通过导航器窗口对指定的对象设置跟踪，使得在模拟过程中随时可以观察该对象的值，具体操作请参看对象跟踪中的说明。

4. 模拟器的使用

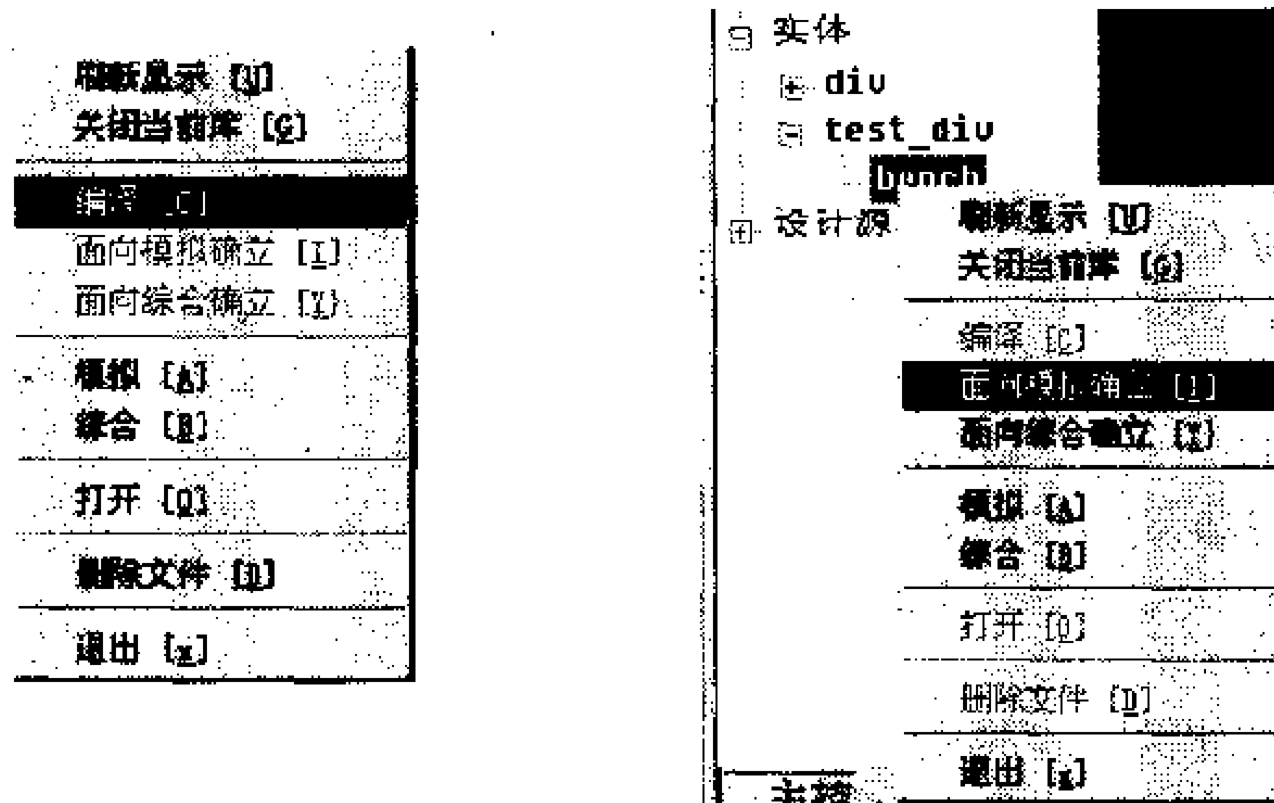
(1) 开始模拟

要对一个 VHDL 设计进行模拟首先要对其源描述进行编译，提取该设计的设计信息，方法如下：在 Talent 的“工程”窗口中选中“设计源”项目下需要编译的 VHDL 语言源描述文件（该文件必须具有扩展文件名 VHD 或 TED），然后按鼠标右键，在弹出菜单中选择“编译”菜单项即可，如附图III.5 所示，编译过程所产生的信息包括错误提示都显示在下方的编译消息输出窗口中。

编译之后要对设计中的层次化信息进行确立，即提取所引用的其他库单元的设计信息。确立的方法如下：在 Talent 的“项目”窗口中按鼠标左键，选中“实体”菜单项最底层的将要进行确立的结构体，然后按鼠标右键，在弹出菜单中选择“面向模拟确立”或“面向综合确立”即可。

在主控界面左侧的“项目”窗口中选中要模拟的库单元（通常是测试台的结构体）之后，在工具条中单击“模拟”按钮，或在“工具”菜单中选中“模拟”菜单项，如果

该库单元已经过面向模拟的确立，则启动模拟器进入模拟调试状态，否则会提示“请先对库单元进行确立”，要求用户对库单元进行面向模拟的确立之后再进行模拟。进入模拟



附图III.5 编译和确立

器之后，库单元对应的源描述将被打开，显示在相应的文档窗口之中，并将光标定位在该单元的描述语句的首行。用户可以对设计的模拟进行各种控制，如设立断点、对象跟踪、观察波形等。

(2) 执行

观察库单元的行为特性必须执行此单元对应的描述语句，调试系统提供了如下 6 种执行方式：

- 直接执行（Run）：一直执行，遇到下一断点时中断。
- 按周期执行（Run Cycle）：执行一个时钟周期。
- 按时间执行(Run Time)：执行一个时间单位。
- 按相位执行(Run Phase)：执行一个相位。
- 单步执行(Next)：执行当前语句。
- 单步跟踪(Step)：执行当前语句，如果遇到子程序则跟进子程序内部。

在调试过程中，经常用到的是直接执行、单步执行和单步跟踪。若要精确地对设计的时序进行分析，通常要用到按周期执行、按时间执行或按相位执行操作。每进行一次执行操作后，调试系统会自动跳转到当前所执行的语句，如果该语句在另外的描述文件内，调试系统会自动打开该语句所在的描述文件，跳转到该语句所在的行，并用当前行符号标记出来。

(3) 断点的设置、浏览和删除

(a) 断点

在调试过程中，设置断点对于分析模拟过程、在模拟过程中检查设计的正确性都是十分重要的。断点可以使模拟过程在指定时刻或是指定的语句处中断，使得用户可以观察模拟过程中的各种信息。本调试系统提供了以下 6 种形式的断点：

① 时间断点(Time)

时间断点可以使用户在运行了指定的时间之后能够中断模拟过程，对此时模拟的状态，如变量、信号的值以及时序关系进行检查。时间断点又可分为相对时间断点和绝对时间断点。相对时间断点指的是从当前时间开始执行完指定的时间长度，而绝对时间长度指的是从模拟的时间零点开始执行指定的时间长度，所以，绝对时间断点只执行一次，而相对时间断点可以执行许多次。

② 周期断点(Cycle)

与时间断点相似，周期断点提供的是在运行指定数目的时钟周期之后中断模拟过程的功能。同样，周期断点也分为相对周期断点和绝对周期断点。相对周期断点从当前周期开始计算，而绝对周期断点从模拟的时间零点开始计算。

③ 对象断点(Object)

对象断点使得用户可以在指定对象（包括信号和变量）的值发生变化时中断模拟过程，对各个对象、进程以及时序关系进行检查。

④ 进程断点(Process)

进程断点是指确定某个进程。当该进程被激活时中断模拟，让用户对模拟进行检查和分析。

⑤ 子程序断点(Subprogram)

子程序断点与进程断点十分相似，当执行到指定的子程序语句时，调试系统中断模拟。它们之间所不同的是，当子程序中断时，用户如果不愿意进入子程序内部执行，则可以用 Next 跳过此子程序的执行，而进程被激活之后是必须进入进程内部的。

⑥ 行断点(Line)

本调试系统允许用户在指定的行设置断点。当执行到该行时，调试系统中断模拟过程。

所有断点都具有以下属性：

断点别名：用户用来标志此断点的唯一名称；

执行次数：指明此断点执行完指定的次数后不再有效；

跳过次数：指明此断点先跳过指定的次数后才开始有效；

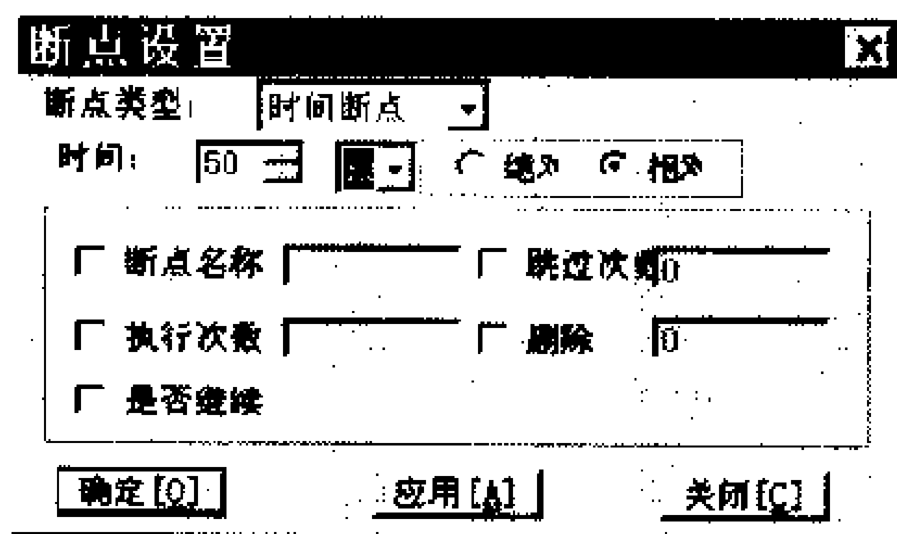
删除：指明此断点执行指定次数后将被删除；

是否继续：指明执行到该断点之后是否继续。

这些属性对于用户来说都是可选的，用户可以忽略其中的选项，甚至可以全部忽略。对于用户没有设置的项，模拟时将略过不予处理。

(b) 断点的设置

在模拟工具条中单击“断点设置”按钮，或在“模拟”菜单中选择“断点设置”菜单项，就会弹出“断点设置”对话框，如附图III.6所示。



附图III.6 断点设置对话框

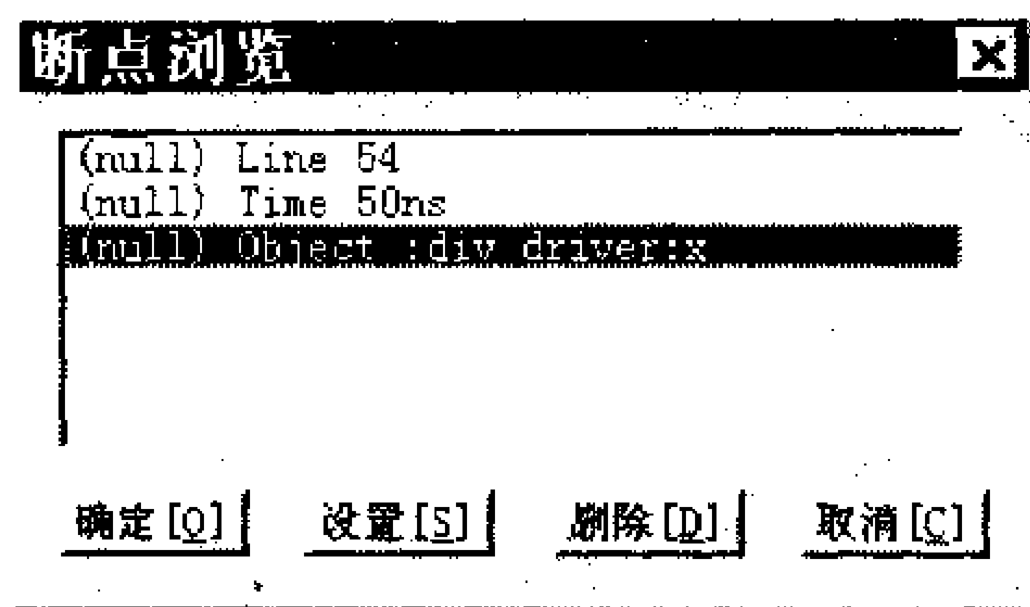
用户可以方便地在此对话框中设置上述六类断点。但是，随着设置断点的类型不同，具体的设置方式也不一样。其中，对象、进程以及子程序断点的设置都是在源描述所在的视窗口中用鼠标点击该对象（进程或子程序）名，如果点击的是合法的对象（进程或子程序）名，相应的名称就会出现在“断点设置”对话框中，同时对应的断点类型也会相应地自动变换；而对于行断点，则应该在该行前面的空白栏中点击鼠标，对应的行号和单元名称就会自动出现在对话框中，断点类型也会变成“行断点”；时间断点与周期断点的设置几乎一样，都是先在“断点设置”对话框中将“断点类型”设置为时间（周期）断点，然后在出现的时间（周期）项中填入指定的数值，并确定是相对断点还是绝对断点，稍有不同的是，时间断点需要设定时间单位，而周期断点则没有此项。需要指出的是，建议用户尽量使用用鼠标在视窗口中点击的方式来设置对象、进程、子程序和行断点，因为人工填入的方式有可能造成单元或层次的错误从而不能正确地设置断点。

(c) 断点的浏览和删除

在模拟工具条和“模拟”菜单中均有“断点浏览”选项，选中该项将弹出“断点浏览”对话框，如附图III.7所示。

所设置的所有类型的断点都显示在对话框中，并标出了断点的类型、名称、别名以及其他的断点信息，在对话框中用户可以看到所有的断点的全部信息。在“断点浏览”对话框中，可以对所选定的某个断点进行删除。断点的删除十分简单，在断点列表中选定某个断点，然后单击对话框中的“删除”按钮，该断点就从列表中删除掉了。此后，

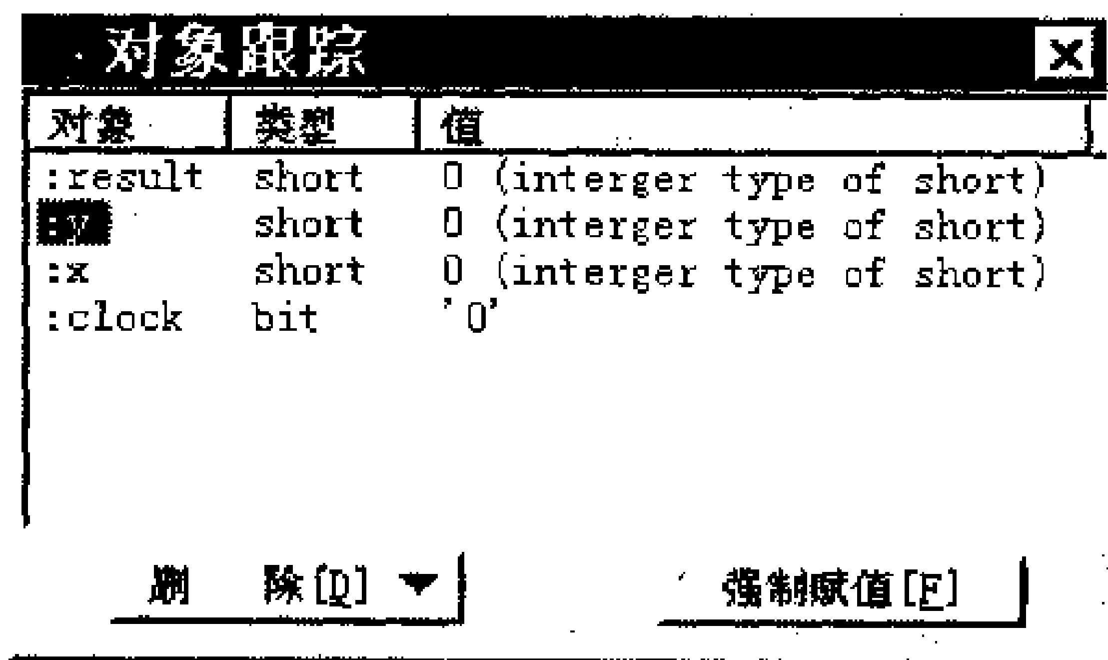
用户执行 Run 操作时，该断点不再有效。需要注意的是，断点删除时没有删除提示消息框，所以用户在删除时，一定要注意，不要将有用的断点删除掉。



附图III.7　断点浏览对话框

(4) 对象跟踪

对象跟踪功能是模拟时非常有用的一项功能，当用户在导航器窗口中选定了某个变量或信号，然后单击工具条中的对象跟踪按钮时，就会弹出“对象跟踪”对话框。对话框的列表中列出了所选对象的带路径名称和值，如附图III.8 所示。



附图III.8　对象跟踪对话框

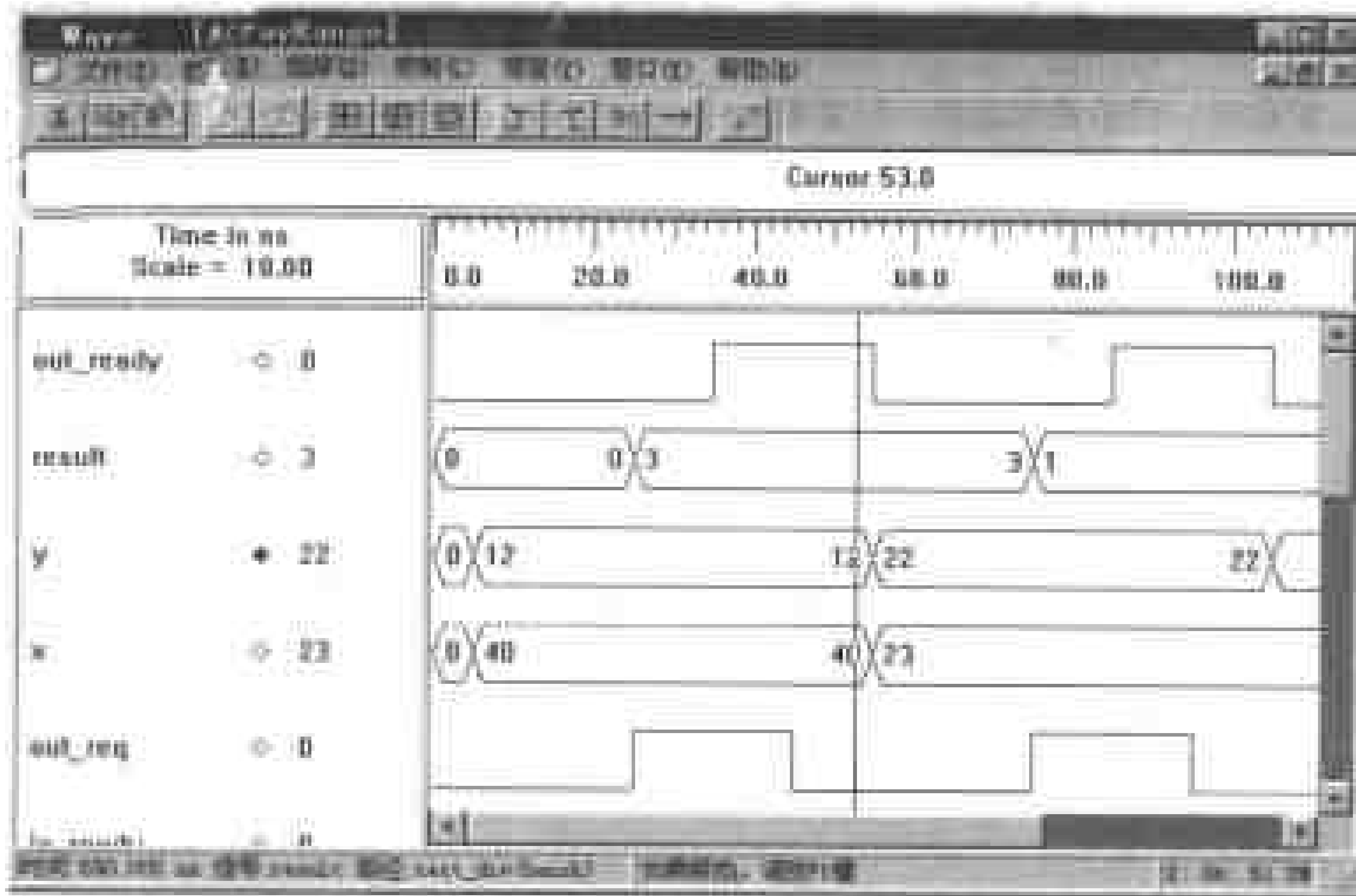
此后用户每次选定的对象都会加入到此对话框中，而且，列表中显示的对象值也会随着模拟过程的进行而动态地不断改变。

(5) 波形显示

模拟过程中为了检验设计的正确性，必须对模拟所产生的波形进行分析。在本调试系统中提供了方便的波形观察工具。用户只要在模拟工具条中单击“波形显示”按钮，波形观察工具就会被激活，用户可以通过波形工具对波形进行详细的分析。

(6) 波形观察工具的使用

波形观察工具的主界面如附图Ⅲ.9 所示，波形显示器最上层是标题条，包括本系统的名称以及专用的图示。标题条之下为菜单条，所有的命令都在菜单中，工具条中是一些常用的命令。



附图Ⅲ.9 波形观察器主界面

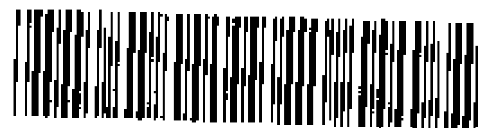
中间的显示区显示信号波形，该显示区中的 Cursor 可用鼠标左键拖动，也可在某个位置双击鼠标左键，将 Cursor 定位到该位置。在这个显示区中可点按鼠标右键弹出菜单，它包括一些常用的对信号操作的命令。

上边的显示区显示带刻度的标尺。左上角的标签显示所用的时间单位以及标尺上单位 1 所代表的数值。信号波形上某一位置的时间值等于该位置在标尺上所对应的值乘上 Scale 值，单位就是左上角的标签上的时间单位。该显示区的上方有一个标签，显示 Cursor 以及它所在位置的时间值。当 Cursor 移动时，所显示的时间值也不断发生变化。

波形工具中还有信号浏览器对话框，可以用于浏览整个设计层次中的信号。信号浏览器对话框分为左右两部分，左边的树型视图用于浏览整个设计层次中的信号，右边滚动列表中显示当前所选中的信号。根据层次信号的特点，以树型形式显示整个设计层次中的所有信号，根据需要可以灵活地在多个层次中进行切换。此外，用户还可以对显示的信号进行打包、查找、跳转到前沿、跳转到后沿、剪切、复制、粘贴等操作，以及对整个显示进行放大、缩小等操作。

(7) 暂停、恢复与退出模拟

在模拟的任[illegible]候，用户都可以方便[illegible]单击工具条中的退出模拟按钮，或单击窗口的关闭按钮，或从模拟菜单中选[illegible]出，[illegible]可以直接退出模拟器。此外，用户也可以暂停模拟过程，单击工具条中的暂[illegible]过程将停止。但此时，模拟并未结束，只要重新单击运行按钮，即可以恢复[illegible]过程。



［Ｇｅｎｅｒａｌ Ｉｎｆｏｒｍａｔｉｏｎ］
书名＝ＶＨＤＬ语言１００例详解
作者＝ＢＥＸＰ
页数＝４９４
下载位置＝ｈｔｔｐ：／／ｂｏｏｋ８．ｓｓｒｅａｄｅｒ．ｃｏｍ／ｄｉｓｋｊｓｊ／ｊｓ９６／０３／！００００１．ｐｄｇ