复杂数字电路与系统的 Verilog HDL 设计技术

夏宇闻　编著

北京航空航天大学出版社

前　言

国内数字逻辑电路与系统的设计多年来一直采用传统的方法，即先在纸上画出真值表，做布尔代数化简，通过画波形图、有限状态机流程图、动态和静态卡诺图等方法来分析和综合电路，然后在实验板上用 CMOS 或 74 系列集成电路块构成数字逻辑电路。当电路比较复杂时设计和调试这样的实验系统需要花费很多的时间和精力。

二十年前国内开始使用微处理机，中规模集成电路的使用也逐步普及，大学里的电子和计算机类学科普遍开设了汇编语言课程和一些常用的中规模集成电路的使用课程。这些技术大大缩短了开发复杂专用数字系统所需的时间。

近十年来，国外先进工业国家由于计算机电路辅助设计技术和半导体集成工艺技术的快速进步，在生产的电子系统中，专用集成电路(ASIC)和 FPGA 的使用越来越多，特别在先进的电讯设备、计算机系统和网络设备中更是如此。这不仅因为有不少实时的 DSP (数字信号处理)芯片是一般微处理机所无法替代的，而且也因为市场对电子产品的要求越来越高。在电子设计和制造领域，我们与国外先进国家的技术差距越来越大。

作为一名在大学讲授复杂专用数字电路与系统设计课程的教师深深感到身上责任的重大。我个人觉得这与大学的课程设置和教学条件有关，因为我们没有及时把国外最先进的设计技术介绍给学生，也没有给他们创造实践的机会。

1992 年我受学校和系领导的委托，筹建世界银行贷款的电子设计自动化(EDA)实验室。其中 CADENCE 设计环境中数字设计部分由我负责。我们首先掌握了利用电路图输入的方法，并逐步掌握了利用 Verilog HDL 设计复杂数字电路的仿真和综合技术。在此基础上我们为有关单位设计了三万门左右的复杂数字电路，提供给他们经仿真验证的 Verilog HDL 源代码，为此得到很高的评价。我们也为自己的科研项目——小波(Wavelet)图像压缩设计了小波卷积器和改进的零修剪树(EZW)算法(即 SPIHT 算法)的硬线逻辑的 Verilog HDL 模型，并成功地进行了仿真和综合。

从 1994 年找到一些有关 Verilog HDL 的资料起，我就在研究生课程“复杂专用数字系统设计”中，逐步增加有关利用 Verilog HDL 进行复杂数字系统设计的内容。1996 年春，我受张凤言教授的邀请，在国家教委电路教学委员会召集的华北区讨论会上作了三个小时的有关 EDA 和 HDL 设计方法的讲座。会后，张凤言教授就一直鼓励我写一本有关 HDL 设计方法的书。现在，这本书终于出版了。因为我们使用 Verilog HDL 设计复杂数字逻辑电路总共也只有两年的时间，水平有限，书中谬误之处在所难免，敬请读者及时把意见反馈给我。之所以匆匆把这本书推出，是想把我们在采用 Verilog HDL 设计方法上积累的一些经验与读者分享，在大学生和研究生中加快 Verilog HDL 设计技术的推广，尽快培养出一批掌握先进设计技术的跨世纪人才。期望本书能在这一过程中起到抛砖引玉的作用。

本书是我们实验室全体老师和同学的共同劳动成果。EDA 实验室的研究生张琰、山岗、王静璇、田玉文和冯文楠等做了许多基础工作，如部分素材的翻译、录入和一些 Verilog HDL 模块的设计和验证。我只是收集了全书的素材，翻译和理解素材中一些较难的概念，最后对全书

文稿进行组织、整理和补充。实验室的董金明和杨惠军老师也给了我许多帮助和鼓励，特别是董金明老师一直以他自己努力工作的实际行动给我以最有力的鞭策，使我不能懈怠。在出版之际，我衷心地感谢在本书编写过程中所有给过我帮助和鼓励的老师和同学们！

编 者

1997 年 12 月于北京航空航天大学 EDA 实验室

电子邮箱：xyw@dept2.buaa.edu.cn
通信地址：北京 100083 北京航空航天大学 205 信箱 夏宇闻
电话： 010－62017251－7253

目　录

第一章　Verilog HDL 设计方法概述

随着电子设计技术的飞速发展，专用集成电路(ASIC)和用户现场可编程门阵列(FPGA)的复杂度越来越高，数字通信、工业自动化控制等领域所用的数字电路及系统的复杂程度也越来越高。设计这样复杂的电路及系统已不再是简单的个人劳动，而需要综合许多专家的经验和知识才能够完成。在数字逻辑设计领域，迫切需要一种共同的工业标准来统一对数字逻辑电路及系统的描述，把专家们设计的各种常用数字逻辑电路和系统部件建成宏单元(Megcell)或软核(Soft-Core)库供设计者引用，以减少重复劳动，提高工作效率。

VHDL 和 Verilog HDL 这两种工业标准的产生顺应了历史的潮流，因而得到了迅速的发展。作为跨世纪的中国大学生应该尽早掌握这种新的设计方法，成为我国 21 世纪深亚微米百万门级的复杂数字逻辑电路及系统设计的技术骨干，使我国在复杂数字电路及系统设计的竞争中逐步缩小与美国等工业发达国家的差距。

1.1　硬件描述语言(HDL)

硬件描述语言(HDL——Hardware Description Language)是一种用形式化方法来描述数字电路和设计数字逻辑系统的语言。数字逻辑电路设计者可以利用这种语言来描述自己的设计思想，然后利用电子设计自动化(下面简称为 EDA)工具进行仿真，再自动综合到门级电路，最后用 ASIC 或 FPGA 实现其功能。目前，这种被称为高层次设计(High-Level-Design)的方法已被广泛采用。据统计，在美国硅谷目前约有 80%的 ASIC 和 FPGA 已采用硬件描述语言进行设计。

硬件描述语言发展至今已有二十多年的历史，并成功地应用于设计的各个阶段：仿真、验证、综合等。到 80 年代，已出现了上百种硬件描述语言，它们对设计自动化起到了极大的促进作用。但是，这些语言一般各自面向特定的设计领域与层次，而且众多的语言使用户无所适从，因此急需一种面向设计的多领域、多层次并得到普遍认同的标准硬件描述语言。进入 80 年代后期，硬件描述语言向着标准化的方向发展。最终，VHDL 和 Verilog HDL 语言适应了这种趋势的要求，先后成为 IEEE 标准。

1.2　Verilog HDL 的历史

1.2.1　什么是 Verilog HDL

Verilog HDL 是硬件描述语言的一种，用于数字电子系统设计。设计者可用它进行各种级别的逻辑设计，可用它进行数字逻辑系统的仿真验证、时序分析和逻辑综合。它是目前应用最广泛的一种硬件描述语言。据有关文献报道，目前在美国使用 Verilog HDL 进行设计的工程师大约有 15 000 人，预计到 2000 年将发展到 60 000 人。

1.2.2 Verilog HDL 的产生及发展

Verilog HDL 是在 1983 年，由 GDA(GateWay Design Automation)公司的 Phil Moorby 首创的。Phil Moorby 后来成为 Verilog-XL 的主要设计者和 Cadence 公司(Cadence Design System)的第一个合伙人。在 1984－1985 年间，Moorby 设计出了第一个关于 Verilog-XL 的仿真器；1986 年，他对 Verilog HDL 的发展又作出了另一个巨大贡献——提出了用于快速门级仿真的 XL 算法。

随着 Verilog-XL 算法的成功，Verilog HDL 语言得到迅速发展。1989 年，Cadence 公司收购了 GDA 公司，Verilog HDL 语言成为 Cadence 公司的私有财产。1990 年，Cadence 公司决定公开发表 Verilog HDL 语言，于是成立了 OVI(Open Verilog International)组织来负责 Verilog HDL 语言的发展。基于 Verilog HDL 的优越性，IEEE 于 1995 年制定了 Verilog HDL 的 IEEE 标准，即 Verilog HDL1364－1995。

图 1.1 显示出 Verilog 的发展历史和未来。

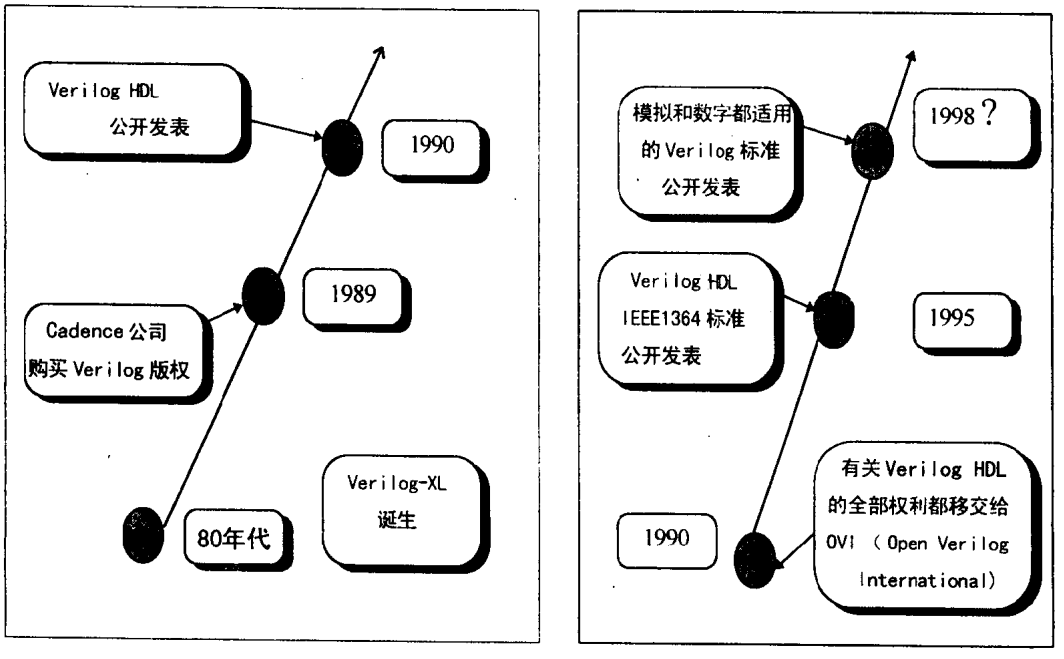


图 1.1 Verilog HDL 的发展历史和未来

1.3 Verilog HDL 和 VHDL 的比较

Verilog HDL 和 VHDL 都是用于逻辑设计的硬件描述语言，并且都已成为 IEEE 标准。VHDL 是在 1987 年成为 IEEE 标准的，Verilog HDL 则在 1995 年才正式成为 IEEE 标准。之所以 VHDL 比 Verilog HDL 早成为 IEEE 标准，这是因为 VHDL 是美国军方组织开发的，而 Verilog HDL 则是从一个普通民间公司的私有财产转化而来，基于 Verilog HDL 的优越性，才成为 IEEE 标准，因而有更强的生命力。

VHDL 的英文全名为 VHSIC Hardware Description Language，而 VHSIC 则是 Very High Speed Integrated Circuit 的缩写词，意为甚高速集成电路，故 VHDL 准确的中文译名为甚高速集成电路的硬件描述语言。

Verilog HDL 和 VHDL 作为描述硬件电路设计的语言，其共同的特点在于：能形式化地抽象表示电路的结构和行为；支持逻辑设计中层次与领域的描述；可借用高级语言的精巧结构来简化电路的描述；具有电路仿真与验证机制以保证设计的正确性；支持电路描述由高层到低层的综合转换；硬件描述与实现工艺无关(有关工艺参数可通过语言提供的属性包括进去)；便于文档管理；易于理解和设计重用。

但是 Verilog HDL 和 VHDL 又各有其自己的特点。由于 Verilog HDL 早在 1983 年就已推出，至今已有 14 年的应用历史，因而 Verilog HDL 拥有更广泛的设计群体，资源也远比 VHDL 丰富。与 VHDL 相比，Verilog HDL 的最大优点为：它是一种非常容易掌握的硬件描述语言，只要有 C 语言的编程基础，通过 20 学时的学习，再加上实际操作，一般可在 2～3 个月内掌握这种设计技术。而掌握 VHDL 设计技术就比较困难，因为 VHDL 不很直观，需要有 Ada 编程基础，一般认为至少需要半年以上的专业培训，才能掌握 VHDL 的基本设计技术。目前版本的 Verilog HDL 和 VHDL 在行为级抽象建模的覆盖范围方面也有所不同。一般认为 Verilog HDL 在系统级抽象方面比 VHDL 略差一些，而在门级开关电路描述方面比 VHDL 强得多。图 1.2 为 Verilog HDL 和 VHDL 建模能力的比较，供读者参考。

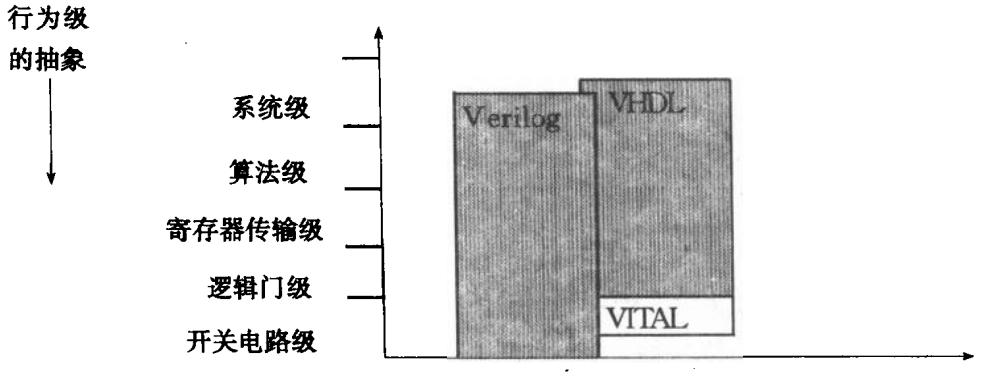


图 1.2 Verilog HDL 与 VHDL 建模能力的比较

但这两种语言也在不断地完善，因此，Verilog HDL 作为学习 HDL 设计方法的入门和基础是比较合适的。学习掌握 Verilog HDL 建模、仿真和综合技术不仅可以对数字电路设计技术有更进一步的了解，而且可以为学习高级的系统综合打下坚实的基础。

1.4 Verilog HDL 目前的应用情况和适用的设计

几年以来，EDA 界一直争论不休的两种硬件描述语言已初见端倪。在美国，高层逻辑电路设计领域中 Verilog HDL 和 VHDL 的应用比率是 60%和 40%；在我国台湾地区各为 50%；在我国大陆目前由于 Verilog HDL 和 VHDL 的使用才刚刚开始，具体应用比率还没有统计。Verilog HDL 是专门为复杂数字逻辑电路和系统的设计仿真而开发的，本身就非常适合复杂

数字逻辑电路和系统的仿真和综合。由于 Verilog HDL 在其门级描述的底层，也就是在晶体管开关的描述方面比 VHDL 有更强的功能，所以即使是 VHDL 的设计环境，在底层实质上也是由 Verilog HDL 描述的器件库所支持的。另外，目前 Verilog HDL-A 标准还支持模拟电路的描述，1998 年即将通过的 Verilog HDL 新标准，将把 Verilog HDL-A 并入 Verilog HDL 新标准，使其不仅支持数字逻辑电路的描述还支持模拟电路的描述。因此在混合信号电路系统的设计中，它必将会有更广泛的应用。在亚微米和深亚微米 ASIC 及高密度 FPGA 已成为电子设计主流的今天，Verilog HDL 的发展前景是非常大的。笔者本人的意见是，若要推广采用硬件描述语言的设计方法，则应首先从推广 Verilog HDL 开始，然后再推广 VHDL。

Verilog HDL 较为适合系统级(System)、算法级(Algorithem)、寄存器传输级(RTL)、门级(Gate)、开关级(Switch)设计；而对于特大型(几百万门级以上)的系统级(System)设计，VHDL 则更为适合。由于这两种 HDL 语言还在发展过程中，它们都会逐步地完善自己。

1.5 采用 Verilog HDL 设计复杂数字电路的优点

1.5.1 传统设计方法——电路原理图输入法

几十年前，当时所做的复杂数字逻辑电路及系统的设计规模比较小也比较简单，其中所用到的 FPGA 或 ASIC 设计工作往往只能采用厂家提供的专用电路图输入工具来进行。为了满足设计性能指标，工程师往往需要花费几天或更长的时间进行艰苦的手工布线。工程师还得非常熟悉所选器件的内部结构和外部引线特点，才能达到设计要求。这种低水平的设计方法延长了设计周期。

近年来，FPGA 和 ASIC 的设计规模不断扩大，复杂度也越来越高，而对逻辑电路及系统的设计时间要求却越来越短。这些因素促使设计人员采用高水准的设计工具，如硬件描述语言(Verilog HDL 或 VHDL)来进行设计。

1.5.2 Verilog HDL 输入法与传统的电路原理图输入法的比较

如 1.5.1 节所述，采用电路原理图输入法进行设计具有设计周期长、需要专门的设计工具、需手工布线等缺陷。而采用 Verilog HDL 输入法时，由于 Verilog HDL 的标准化，可以很容易地把完成的设计移植到不同厂家的不同芯片中去，并在不同规模应用时可以较容易地作修改。这是因为用 Verilog HDL 所完成的设计，其信号位数容易改变，可以很方便地对其进行修改，以适应不同规模的应用；在仿真验证时，仿真测试矢量还可以用同一种描述语言来完成；另外，采用 Verilog HDL 综合器生成的数字逻辑是一种标准的电子设计互换格式(EDIF)文件，独立于所采用的实现工艺。有关工艺参数的描述可以通过 Verilog HDL 提供的属性包括进去，然后利用不同厂家的布局布线工具，在不同工艺的芯片上实现。

采用 Verilog HDL 输入法最大的优点是其与工艺无关性。这使得工程师在功能设计、逻辑验证阶段，可以不必过多考虑门级及工艺实现的具体细节，只需要利用系统设计时对芯片的要求，施加不同的约束条件，即可设计出实际电路。实际上这是利用了计算机的巨大能力并在 EDA 工具的帮助下，把逻辑验证与具体工艺库匹配、布线及时延计算分成不同的阶段来实现，从而减轻了人们的繁琐劳动。

1.5.3 Verilog HDL 的标准化与软核的重用

Verilog HDL 是在 1983 年由 GDA 公司首先开发成功的，经过诸多改进，于 1995 年 11 月正式被批准为 IEEE 标准 1364。

Verilog HDL 的标准化大大加快了 Verilog HDL 的推广和发展。由于 Verilog HDL 设计方法的与工艺无关性，大大提高了 Verilog HDL 模型的可重用性。我们把功能经过验证的、可综合的、实现后电路结构总门数在 5 000 门以上的 Verilog HDL 模型称之为“软核”(Soft Core)，而把由软核构成的器件称为虚拟器件。在新电路的研制过程中，软核和虚拟器件可以很容易地借助 EDA 综合工具与其他外部逻辑结合为一体。这样，软核和虚拟器件的可重用性就可大大缩短设计周期，加快了复杂电路的设计。目前国际上有一个叫作“虚拟接口联盟”(Virtual Socket Interface Alliance)的组织来协调这方面的工作。

1.5.4 软核、固核和硬核的概念以及它们的重用

1.5.3 节中已介绍了软核的概念，下面再介绍一下固核(Firm Core)和硬核(Hard Core)的概念。

我们把在某一种现场可编程门阵列(FPGA)器件上实现的、经验证是正确的、总门数在 5 000门以上电路结构编码文件称之为“固核”。

把在某一种专用半导体集成电路工艺的(ASIC)器件上实现的、经验证是正确的、总门数在 5 000 门以上的电路结构掩膜称之为“硬核”。

显而易见，在具体实现手段和工艺技术尚未确定的逻辑设计阶段，软核具有最大的灵活性，它可以很容易地借助 EDA 综合工具与其他外部逻辑结合为一体。当然，由于实现技术的不确定性，有可能要作一些改动以适应相应的工艺。相比之下，固核和硬核与其他外部逻辑结合为一体的灵活性要差得多，特别是电路实现工艺技术改变时更是如此。而近年来电路实现工艺技术的发展是相当迅速的，为了逻辑电路设计成果的积累和更快更好地设计更大规模的电路，发展软核的设计和推广软核的重用技术是非常必要的。新一代的数字逻辑电路设计师必须掌握这方面的知识和技术。

1.6 Verilog HDL 的设计流程简介

1.6.1 自顶向下(TOP-DOWN)设计的基本概念

现代集成电路制造工艺技术的改进，使得在一个芯片上集成数十万乃至数百万个器件成为可能，但很难设想仅由一个设计师独立设计如此大规模的电路而不出现错误。利用层次化、结构化的设计方法，一个完整的硬件设计任务首先由总设计师划分为若干个可操作的模块，编制出相应的模型(行为的或结构的)，通过仿真加以验证后，再把这些模块分配给下一层的设计师。这就允许多个设计者同时设计一个硬件系统中的不同模块，其中每个设计者负责自己所承担的部分，而由上一层设计师对其下层设计者完成的设计用行为级上层模块进行验证。图 1.3 为自顶向下(TOP-DOWN)的示意图，以设计树的形式绘出。

自顶向下的设计是从系统级开始，把系统划分为基本单元，然后再把每个基本单元划分为下一层次的基本单元，一直这样做下去，直到可以直接用 EDA 元件库中的元件来实现为止。

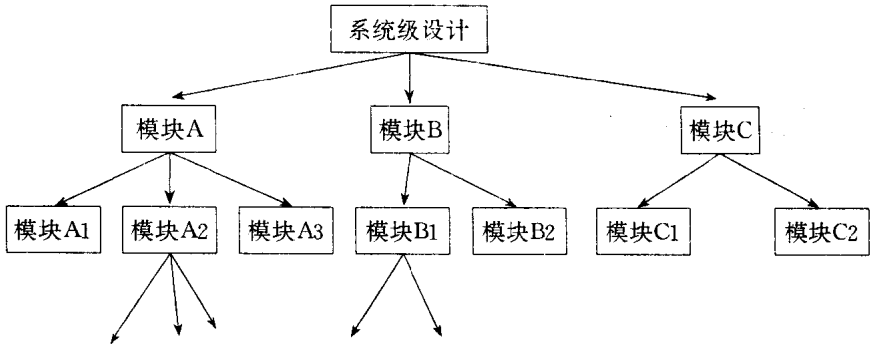


图 1.3 TOP-DOWN 设计思想

对于设计开发整机电子产品的单位和个人来说，新产品的开发总是从系统设计入手，进行方案的总体论证、功能描述、任务和指标的分配。随着系统变得复杂和庞大，特别需要在样机问世之前，对产品的全貌有一定的预见性。目前，EDA 技术的发展使得设计师有可能实现真正的自顶向下的设计。

1.6.2 层次管理的基本概念

复杂数字逻辑电路和系统的层次化、结构化设计隐含着硬件设计方案的逐次分解。在设计过程中的任意层次，硬件至少有一种描述形式。硬件的描述特别是行为描述通常称为行为建模。在集成电路设计的每一层次，硬件可以分为一些模块，该层次的硬件结构由这些模块的互连描述，该层次的硬件行为由这些模块的行为描述。这些模块称为该层次的基本单元，而该层次的基本单元又由下一层次的基本单元互连而成，如此下去，完整的硬件设计就可以由图 1.3 所示的设计树描述。在这个设计树上，节点对应着该层次上基本单元的行为描述，树枝对应着基本单元的结构分解。在不同的层次都可以进行仿真以对设计思想进行验证。EDA 工具提供了有效的手段来管理错综复杂的层次，即可以很方便地查看某一层次某模块的源代码或电路图以改正仿真时发现的错误。

1.6.3 具体模块的设计编译和仿真的过程

在不同层次做具体模块的设计所用的方法有所不同。在高层次上往往需要编写一些行为级的模块，通过仿真加以验证，其主要目的是对系统性能的总体考虑和各模块的指标分配，并非具体电路的实现，因而综合及其以后的步骤往往不需进行。而当设计的层次比较接近底层时，行为描述往往需要用电路逻辑来实现，这时的模块不仅需要通过仿真加以验证，还需进行综合、优化、布线和后仿真。总之，具体电路是从底向上逐步实现的。EDA 工具往往不仅支持 HDL 描述也支持电路图输入，有效地利用这两种方法是提高设计效率的办法之一。图 1.4 简要地说明了模块的编译和测试过程。

从图 1.4 可以看出，模块设计流程主要由两大主要功能部分组成。

(1) 设计开发：即从编写设计文件→综合→布局布线→投片生成这样一系列步骤。

(2) 设计验证：也就是进行各种仿真的一系列步骤，如果在仿真过程中发现问题就返回设计输入进行修改。

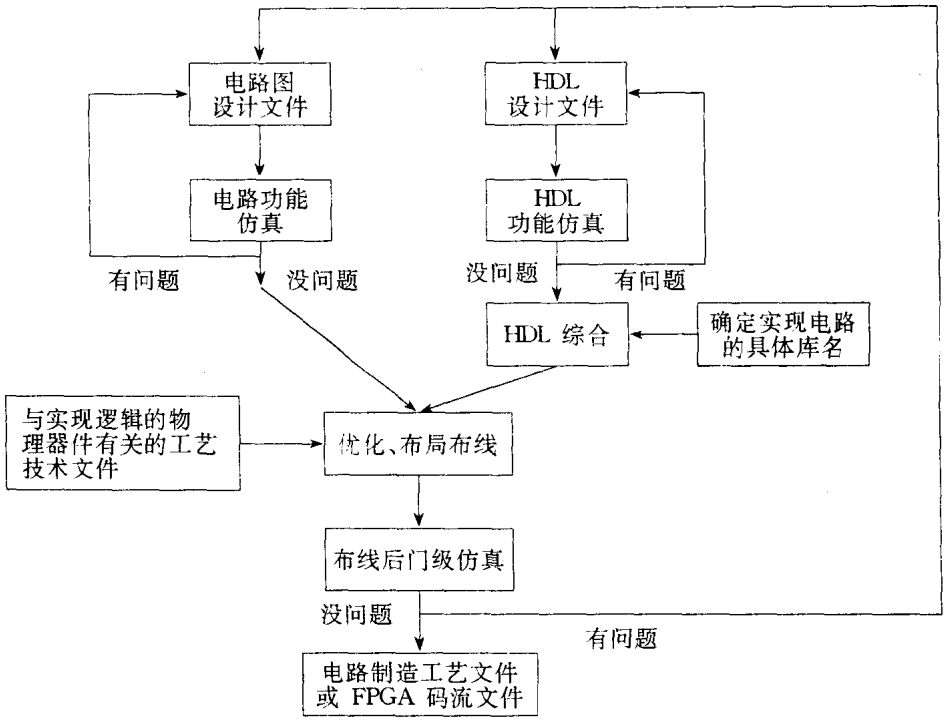


图 1.4 HDL 设计流程图

1.6.4 对应具体工艺器件的优化、映象和布局布线

由于各种 ASIC 和 FPGA 器件的工艺各不相同，因而当用不同厂家的不同器件来实现已验证的逻辑网表(EDIF 文件)时，需要不同的基本单元库与布线延迟模型与之对应才能进行准确的优化、映象和布局布线。基本单元库与布线延迟模型由熟悉本厂工艺的工程师提供，再由 EDA 厂商的工程师编入相应的处理程序，而逻辑电路设计师只需用一文件说明所用的工艺器件和约束条件，EDA 工具就会自动地根据这一文件选择相应的库和模型进行准确的处理，从而大大提高设计效率。

1.7 小 结

采用 Verilog HDL 设计方法比采用电路图输入的方法更有优越性，这就是为什么美国等先进工业国家在进入 90 年代以后纷纷采用 HDL 设计方法的原因。在两种符合 IEEE 标准的硬件描述语言中，Verilog HDL 与 VHDL 相比，更加基础、更易学习。掌握 HDL 设计方法应从学习 Verilog HDL 设计方法开始。Verilog HDL 可用于复杂数字逻辑电路和系统的总体仿真、子系统仿真和具体电路综合等各个设计阶段。

由于 TOP-DOWN 的设计方法是首先从系统设计入手的，因而从顶层进行功能划分和结构设计。系统的总体仿真是顶层进行功能划分的重要环节，这时的设计是与工艺无关的。由于设计的主要仿真和调试过程是在高层次完成的，所以能够早期发现结构设计上的错误，避免设计工作的浪费，同时也减少了逻辑仿真的工作量。自顶向下的设计方法方便了从系统级划分和管理整个项目，使得几十万门甚至几百万门规模的复杂数字电路的设计成为可能，并可减少设

计人员，避免不必要的重复设计，提高了设计的一次成功率。

从底向上的设计在某种意义上讲可以看作上述 TOP-DOWN 设计的逆过程。虽然设计也是从系统级开始，即从设计树的树根开始对设计进行逐次划分，但划分时首先考虑的是单元是否存在，即设计划分过程必须从存在的基本单元出发。设计树最末枝上的单元要么是已经制造出的单元，要么是其他项目已开发好的单元或者是可外购得到的单元。

自顶向下的设计过程中，在每一层次划分时都要对某些目标作优化，TOP-DOWN 的设计过程是理想的设计过程，它的缺点是得到的最小单元不标准，制造成本可能很高。从底向上的设计过程全部采用标准基本单元，通常比较经济，但有时可能不能满足一些特定的指标要求。复杂数字逻辑电路和系统的设计过程通常是这两种设计方法的结合，设计时需要考虑多个目标的综合平衡。

思 考 题

1. 什么是硬件描述语言？它的主要作用是什么？
2. 目前世界上符合 IEEE 标准的硬件描述语言有哪两种？它们各有什么特点？
3. 什么情况下需要采用硬件描述语言的设计方法？
4. 采用硬件描述语言设计方法的优点是什么？有什么缺点？
5. 简单叙述一下利用 EDA 工具并采用硬件描述语言的设计方法和流程。
6. 硬件描述语言可以用哪两种方式参与复杂数字电路的设计？
7. 用硬件描述语言设计的数字系统需要经过哪些步骤才能与具体的电路相对应？
8. 为什么说用硬件描述语言设计的数字逻辑系统具有最大的灵活性，可以映射到任何工艺的电路上？
9. 软核是什么？虚拟器件是什么？它们的作用是什么？
10. 固核是什么？硬核是什么？与软核相比它们各有什么优缺点？
11. 简述 TOP-DOWN 设计方法和硬件描述语言的关系。

第二章　Verilog HDL 的基本语法

Verilog HDL 是一种用于数字逻辑电路设计的语言。用 Verilog HDL 描述的电路设计就是该电路的 Verilog HDL 模型。Verilog HDL 既是一种行为描述的语言也是一种结构描述的语言。也就是说，既可以用电路的功能描述也可以用元器件和它们之间的连接来建立所设计电路的 Verilog HDL 模型。Verilog 模型可以是实际电路的不同级别的抽象。这些抽象的级别和它们所对应的模型类型共有以下 5 种：

(1) 系统级(system-level)：用高级语言结构实现设计模块外部性能的模型；

(2) 算法级(algorithem-level)：用高级语言结构实现设计算法的模型；

(3) RTL 级(Register Transfer Level)：描述数据在寄存器之间流动和如何处理这些数据的模型；

(4) 门级(gate-level)：描述逻辑门以及逻辑门之间连接的模型；

(5) 开关级(switch-level)：描述器件中三极管和储存节点以及它们之间连接的模型。

一个复杂电路的完整 Verilog HDL 模型是由若干个 Verilog HDL 模块构成的，每一个模块又可以由若干个子模块构成。利用 Verilog HDL 语言结构所提供的这种功能就可以构造一个模块间的清晰层次结构来描述极其复杂的大型设计。

Verilog HDL 行为描述语言作为一种结构化和过程性的语言，其语法结构非常适合于算法级和 RTL 级的模型设计。这种行为描述语言具有以下功能：

(1) 可描述顺序执行或并行执行的程序结构；

(2) 用延迟表达式或事件表达式来明确地控制过程的启动时间；

(3) 通过命名的事件来触发其他过程里的激活行为或停止行为；

(4) 提供了条件(如 if _ else,case 等)循环程序结构；

(5) 提供了可带参数且非零延续时间的任务(task)程序结构；

(6) 提供了可定义新的操作符的函数(function)结构；

(7) 提供了用于建立表达式的算术运算符、逻辑运算符、位运算符。

Verilog HDL 语言作为一种结构化的语言也非常适合于门级和开关级的模型设计。因其结构化的特点又使它具有以下功能：

(1) 提供了一套完整的组合型原语(primitive)；

(2) 提供了双向通路和电阻器件的原语；

(3) 可建立 MOS 器件的电荷分享和电荷衰减动态模型。

Verilog HDL 的构造性语句可以精确地建立信号的模型。这是因为在 Verilog HDL 中，提供了延迟和输出强度的原语来建立精确程度很高的信号模型。信号值可以有不同的强度，可以通过设定宽范围的模糊值来降低不确定条件的影响。

Verilog HDL 作为一种高级的硬件描述编程语言，有着类似 C 语言的风格。其中有许多语句，如 if 语句、case 语句等，和 C 语言中的对应语句十分相似。如果读者已经掌握 C 语言编程的基础，那么学习 Verilog HDL 并不困难，只要对 Verilog HDL 某些语句的特殊方面着重加

以理解，并加强上机练习就能很好地掌握它，利用它的强大功能来设计复杂的数字逻辑电路。下面将对 Verilog HDL 中的基本语法逐一加以介绍。

2.1 简单的 Verilog HDL 模块

2.1.1 简单的 Verilog HDL 程序介绍

下面先介绍几个简单的 Verilog HDL 程序，然后从中分析 Verilog HDL 程序的特点。

例[1]
```
module  adder ( count,sum,a,b,cin );
        input [2 : 0]  a,b;
        input  cin;
        output  count;
        output [2 : 0]  sum;
            assign {count,sum}=a+b+cin;
endmodule
```

此例描述了一个 3 位的加法器。从例子中可以看出整个 Verilog HDL 程序是嵌套在 module 和 endmodule 声明语句里的。

例[2]
```
module  compare ( equal,a,b );
  output  equal;                    //声明输出信号 equal
  input [1 : 0]  a,b;               //声明输入信号 a,b
  assign  equal=(a==b)? 1 : 0;
  /* 如果两个输入信号相等,输出为 1;否则为 0 */
endmodule
```

这个程序描述了一个比较器。其中，/ * …… * /和//……表示注释部分。注释只是为了方便程序员理解程序，对编译是不起作用的。

例[3]
```
module  trist2(out,in,enable);
   output  out;
   input  in, enable;
   bufif1  mybuf(out,in,enable);
endmodule
```

这个程序描述了一个三态驱动器。程序通过调用一个实例元件 bufif1 来实现其功能。

例[4]
```
module  trist1(out,in,enable);
    output  out;
    input  in, enable;
    mytri  tri_inst(out,in,enable);
endmodule

module  mytri(out,in,enable);
    output  out;
    input  in, enable;
    assign  out = enable? in : 'bz;
endmodule
```

这个程序通过另一种方法描述了一个三态门。在这个例子中存在着两个模块。模块 trist1 调用模块 mytri 的实例元件 tri _inst。模块 trist1 是顶层模块，模块 mytri 则被称为子模块。

通过上面的例子可以看到：

(1) Verilog HDL 程序是由模块构成的。每个模块的内容都是嵌在 module 和 endmodule 两个语句之间。每个模块实现特定的功能。模块是可以进行层次嵌套的。正因为如此，才可以将大型的数字电路设计分割成不同的小模块来实现特定的功能，最后通过顶层模块调用子模块来实现整体功能。

(2) 每个模块要进行端口定义，并说明输入、输出口，然后对模块的功能进行行为逻辑描述。

(3) Verilog HDL 程序的书写格式自由，一行可以写几个语句，一个语句也可以分多行写。

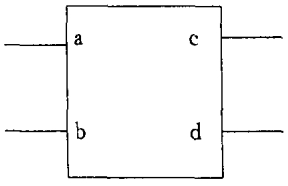
(4) 除了 endmodule 语句外，每个语句和数据定义的最后必须有分号。

(5) 可以用/ * …… * /和//……对 Verilog HDL 程序的任何部分作注释。一个好的、有使用价值的源程序都应当加上必要的注释，以增强程序的可读性和可维护性。

2.1.2 模块的结构

Verilog 的基本设计单元是“模块(block)”。一个模块是由两部分组成的，一部分描述接口；另一部分描述逻辑功能，即定义输入是如何影响输出的。下面举例说明。

```
module  block(a,b,c,d);
 input  a,b;
 output  c,d;

 assign  c=a|b;
 assign  d=a &b;
 endmodule
```



请看上面的例子，程序模块旁边有一个电路图的符号。在许多方面，程序模块和电路图符号是一致的，这是因为电路图符号的引脚也就是程序模块的接口，而程序模块描述了电路图符号所实现的逻辑功能。上面的 Verilog 设计中，模块中的第 2、第 3 行说明接口的信号流向，第 4、第 5 行说明了模块的逻辑功能。以上就是设计一个简单的 Verilog 程序模块所需的全部内容。

从上面的例子可以看出，Verilog 结构完全嵌在 module 和 endmodule 声明语句之间，每个 Verilog 程序包括 4 个主要部分：端口定义、I/O 说明、内部信号声明和功能定义。

2.1.3 模块的端口定义

模块的端口声明了模块的输入、输出口。其格式如下：

module 模块名(口 1,口 2,口 3,口 4, ……);

2.1.4 模块内容

模块的内容包括 I/O 说明、内部信号声明和功能定义。

I/O 说明的格式如下：

输入口： input 端口名 1,端口名 2,……,端口名 N;

输出口： output 端口名 1,端口名 2,……,端口名 N;

I/O 说明也可以写在端口声明语句里。其格式如下：

```
module  module_name(input port1,input port2,…
                    output port1,output port2,…);
```

模块中最重要的部分是逻辑功能定义。有 3 种方法可在模块中产生逻辑。

1. 用“assign”声明语句

如：

```
assign  a = b & c;
```

这种方法的句法很简单，只需写一个“assign”，后面再加一个方程式即可。例中的方程式描述了一个有两个输入的与门。

2. 用实例元件

如：

```
and  and_inst( q, a, b );
```

采用实例元件的方法同在电路图输入方式下调入库元件一样，键入元件的名字和相连的引脚即可。要求每个实例元件的名字必须是唯一的。

3. 用“always”块

如：

```
always @(posedge clk or posedge clr)
    begin
            if(clr)   q<=0;
            else  if(en)  q<=d;
    end
```

采用“assign”语句，是最常用的方法之一。“always”块可用于产生各种逻辑，它常用于描述时序逻辑，这个例子用“always”块生成了一个带有异步清除端的 D 触发器。“always”块可用很多种描述手段来表达逻辑，例如本例中就用了 if_else 语句来表达逻辑关系。

注意：如果用 Verilog 模块实现一定的功能，首先应该清楚哪些是同时发生的，哪些是顺序发生的。上面 3 个例子分别采用了“assign”语句、实例元件和“always”块。这 3 个例子描述的逻辑功能是同时执行的。也就是说，如果把这 3 项写到一个 Verilog 文件中去，它们的顺序不会影响实现的功能。这 3 项是同时执行的，也就是并发的。

然而，在“always”模块内，逻辑是按照指定的顺序执行的。“always”块中的语句称为“顺序语句”，因为它们是顺序执行的。请注意，两个或更多的“always”模块是同时执行的，但是模块内部的语句是顺序执行的。看一下“always”内的语句，就会明白它是如何实现功能的。if_else_if 必须顺序执行，否则其功能就没有任何意义。如果 else 语句在 if 语句之前执行，功能就会不符合要求！为了能实现上述描述的功能，“always”模块内部的语句将按照书写的顺序执行。

2.2 数据类型及其常量、变量

Verilog HDL 中总共有 19 种数据类型。数据类型是用来表示数字电路硬件中的数据储存

和传送元素的。在此我们先介绍 4 个最基本的数据类型，它们是：integer 型、parameter 型、reg 型和 wire 型。其他数据类型在后面的章节里逐步介绍，也可通过查阅 Verilog HDL 语法参考书逐步掌握。其他的类型有 large 型、medium 型、scalared 型、time 型、small 型、tri 型、trio 型、tri1 型、triand 型、trior 型、trireg 型、vectored 型、wand 型和 wor 型。

Verilog HDL 语言中也有常量和变量之分。它们分别属于以上这些类型。下面对最常用的几种进行介绍。

2.2.1 常 量

在程序运行过程中，其值不能被改变的量称为常量。下面首先对在 Verilog HDL 语言中使用的数字及其表示方式进行介绍。

一、数 字

1. 整 数

在 Verilog HDL 中，整型常量(即整常数)有以下 4 种进制表示形式：

(1) 二进制整数(b 或 B)；

(2) 十进制整数(d 或 D)；

(3) 十六进制整数(h 或 H)；

(4) 八进制整数(o 或 O)。

数字表达方式有以下 3 种：

(1) <位宽><进制><数字>，这是一种全面的描述方式。

(2) 在<进制><数字>这种描述方式中，数字的位宽采用缺省位宽(这由具体的机器系统决定，但至少 32 位)。

(3) 在<数字>这种描述方式中，采用缺省进制(十进制)。

在表达式中，位宽指明了数字的精确位数。例如：一个 4 位二进制数的位宽为 4，一个 4 位十六进制数的位宽为 16(因为每单个十六进制数就要用 4 位二进制数来表示)。见下例：

```
8'b10101100    //位宽为 8 的数的二进制表示，'b 表示二进制
8'ha2          //位宽为 8 的数的十六进制表示，'h 表示十六进制
```

2. x 和 z 值

在数字电路中，x 代表不定值，z 代表高阻值。x 可以用来一次定义十六进制数的 4 位二进制数的状态，八进制数的 3 位，二进制数的 1 位。z 的表示方式同 x 类似。z 还有一种表达方式是可以写作“?”。在使用 case 表达式时建议使用这种写法，以提高程序的可读性。见下例：

```
4'b10x0      //位宽为 4 的二进制数从低位数起第 2 位为不定值
4'b101z      //位宽为 4 的二进制数从低位数起第 1 位为高阻值
12'dz        //位宽为 12 的十进制数，其值为高阻值(第 1 种表达方式)
12'd?        //位宽为 12 的十进制数，其值为高阻值(第 2 种表达方式)
8'h4x        //位宽为 8 的十六进制数，其低 4 位值为不定值
```

3. 负 数

一个数字可以被定义为负数，只需在位宽表达式前加一个减号，减号必须写在数字定义表达式的最前面。注意，减号不可以放在位宽和进制之间，也不可以放在进制和具体的数之间。见下例：

```
-8'd5          //这个表达式代表 5 的补数(用 8 位二进制数表示)
8'd-5          //非法格式
```

4. 下划线(underscore _)

下划线可以用来分隔数的表达以提高程序可读性,但不可以用在位宽和进制处,只能用在具体的数字之间。见下例:

```
16'b1010_1011_1111_1010        //合法格式
8'b_0011_1010                  //非法格式
```

当常量不说明位数时,默认值是 32 位,每个字母用 8 位的 ASCII 值表示。例:

```
10=32'd10=32'b1010
1=32'd1=32'b1
-1=-32'd1=32'hFFFFFFFF
'BX=32'BX=32'BXXXXXXX…X
"AB"=16'B01000001_01000010
```

二、参数(parameter)型

在 Verilog HDL 中用 parameter 来定义常量,即用 parameter 来定义一个标识符代表一个常量,称为符号常量,即标识符形式的常量。采用标识符代表一个常量可提高程序的可读性和可维护性。parameter 型数据是一种常数型的数据,其说明格式如下:

```
parameter  参数名 1=表达式,参数名 2=表达式,……,参数名 n=表达式;
```

parameter 是参数型数据的确认符。确认符后跟着一个用逗号分隔开的赋值语句表。在每一个赋值语句的右边必须是一个常数表达式,也就是说,该表达式只能包含数字或先前已定义过的参数。见下例:

```
parameter  msb=7;                                  //定义参数 msb 为常量 7
parameter  e=25, f=29;                             //定义两个常数参数
parameter  r=5.7;                                  //声明 r 为一个实型参数
parameter  byte_size=8, byte_msb=byte_size-1;      //用常数表达式赋值
parameter  average_delay = (r+f)/2;                //用常数表达式赋值
```

参数型常数经常用于定义延迟时间和变量宽度。

在模块或实例引用时,可通过参数传递改变在被引用模块或实例中已定义的参数。下面将通过两个例子进一步说明在层次调用的电路中改变参数常用的一些方法。

例[1] 在引用 Decode 实例时,D1,D2 的 Width 将采用不同的值 4 和 5,且 D1 的 Polarity 将为 0。可用本例中所用的方法来改变参数,即用 #(4,0)向 D1 中传递 Width=4,Polarity=0,用#(5)向 D2 中传递 Width=5,Polarity 仍为 1。

```
module  Decode(A,F);
    parameter  Width=1, Polarity=1;
        ……
endmodule
module  Top;
    wire [3:0]  A4;
    wire [4:0]  A5;
    wire [15:0]  F16;
    wire [31:0]  F32;
```

```
    Decode  #(4,0)  D1(A4,F16);
    Decode  #(5)  D2(A5,F32);
endmodule
```

例[2] 图 2.1 是一个多层次模块构成的电路，在一个模块中改变另一个模块的参数时，需要使用 defparam 命令。

```
Module   Test;
  wire  W;
  Top  T();
endmodule

module  Top;
  wire  W
  Block  B1();
  Block  B2();
endmodule

module  Block;
  Parameter  P=0;
endmodule

module  Annotate;
  defparam
  Test.T. B1.P=2,
  Test.T. B2.P=3;
endmodule
```

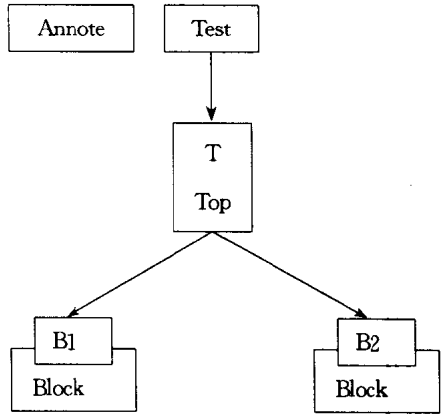


图 2.1 由多层次模块构成的电路

2.2.2 变 量

变量即在程序运行过程中其值可以改变的量。在 Verilog HDL 中变量的数据类型有很多种，这里只对常用的几种进行介绍。

网络数据类型表示结构实体(例如门)之间的物理连接。网络类型的变量不能储存值，而且它必须受到驱动器(例如门或连续赋值语句，assign)的驱动。如果没有驱动器连接到网络类型的变量上，则该变量就是高阻的，即其值为 z。常用的网络数据类型包括 wire 型和 tri 型。这两种变量都是用于连接器件单元，它们具有相同的语法格式和功能。之所以提供这两种名字来表达相同的概念，是为了与模型中所使用的变量的实际情况相一致。wire 型变量通常是用来表示单个门驱动或连续赋值语句驱动的网络型数据，tri 型变量则用来表示多驱动器驱动的网络型数据。如果 wire 型或 tri 型变量没有定义逻辑强度(logicstrength)，在多驱动源的情况下，逻辑值会发生冲突，从而产生不确定值。下表为 wire 型和 tri 型变量的真值表(注意：这里假设两个驱动源的强度是一致的，关于逻辑强度建模请参阅参考书)。

wire/tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

一、wire 型

wire 型数据常用来表示以 assign 关键字指定的组合逻辑信号。Verilog 程序模块中输入、输出信号类型缺省时自动定义为 wire 型。wire 型信号可以用作任何方程式的输入，也可以用作“assign”语句或实例元件的输出。

wire 型信号的格式同 reg 型信号的很类似。其格式如下：

wire [n－1：0] 数据名 1，数据名 2，……，数据名 n；

或

wire [n：1] 数据名 1，数据名 2，……，数据名 n；

wire 是 wire 型数据的确认符；[n－1：0]和[n：1]代表该数据的位宽，即该数据有几位；最后跟着的是数据的名字。如果一次定义多个数据，数据名之间用逗号隔开。声明语句的最后要用分号表示语句结束。看下面的几个例子：

```
wire  a;          //定义了一个 1 位的 wire 型数据
wire [7：0] b;    //定义了一个 8 位的 wire 型数据
wire [4：1] c，d;//定义了两个 4 位的 wire 型数据
```

二、reg 型

寄存器是数据储存单元的抽象。寄存器数据类型的关键字是 reg。通过赋值语句可以改变寄存器储存的值，其作用与改变触发器储存的值相当。Verilog HDL 语言提供了功能强大的结构语句，使设计者能有效地控制是否执行这些赋值语句。这些控制结构用来描述硬件触发条件，例如时钟的上升沿和多路器的选通信号等。在介绍行为模块一节中还要详细地介绍这些控制结构。reg 类型数据的缺省初始值为不定值 x。

reg 型数据常用来表示“always”模块内的指定信号，常代表触发器。通常，在设计中要由“always”模块通过使用行为描述语句来表达逻辑关系。在“always”模块内赋值的每一个信号都必须定义成 reg 型。

reg 型数据的格式如下：

reg [n－1：0] 数据名 1，数据名 2，……，数据名 n；

或

reg [n：1] 数据名 1，数据名 2，……，数据名 n；

reg 是 reg 型数据的确认标识符；[n－1：0]和[n：1]代表该数据的位宽，即该数据有几位；最后跟着的是数据的名字。如果一次定义多个数据，数据名之间用逗号隔开。声明语句的最后要用分号表示语句结束。下面看几个例子：

```
reg  rega;              //定义了一个 1 位的名为 rega 的 reg 型数据
reg [3：0] regb;        //定义了一个 4 位的名为 regb 的 reg 型数据
reg [4：1] regc，regd;  //定义了两个 4 位的名为 regc 和 regd 的 reg 型数据
```

对于 reg 型数据，其赋值语句的作用就如同改变一组触发器的存储单元的值。在 Verilog

中有许多构造(construct)来控制何时或是否执行这些赋值语句。这些控制构造用来描述硬件触发器的状态,如时钟的上升沿,或用来描述判断逻辑,如多路选择器。

reg 型数据的缺省初始值是不定值。reg 型数据可以赋正值,也可以赋负值。但当一个 reg 型数据是一个表达式中的操作数时,它的值被当作是无符号值,即正值。例如,当一个 4 位的寄存器用作表达式中的操作数时,如果开始寄存器被赋以值-1,则在表达式中进行运算时,其值被认为是+15。

注意:reg 型只表示被定义的信号将用在"always"块内,理解这一点很重要。并不是说 reg 型信号一定是寄存器或触发器的输出,虽然 reg 型信号常常是寄存器或触发器的输出,但并不一定总是这样。

三、memory 型

Verilog HDL 通过对 reg 型变量建立数组来对存储器建模,可以描述 RAM 存储器、ROM 存储器和 reg 文件。数组中的每一个单元通过一个数组索引进行寻址。在 Verilog 语言中没有多维数组存在。

memory 型数据是通过扩展 reg 型数据的地址范围来生成的。其格式如下:

```
reg [n-1:0]  存储器名[m-1:0];
```

或

```
reg [n-1:0]  存储器名[m:1];
```

在这里,reg[n-1:0]定义了存储器中每一个存储单元的大小,即该存储单元是一个 n 位的寄存器;存储器名后的[m-1:0]或[m:1]则定义了该存储器中有多少个这样的寄存器;最后用分号结束定义语句。下面举例说明:

```
reg [7:0]  mema[255:0];
```

这个例子定义了一个名为 mema 的存储器,该存储器有 256 个 8 位的寄存器。该存储器的地址范围是从 0 到 255。注意:对存储器进行地址索引的表达式必须是常数表达式。

另外,在同一个数据类型声明语句里,可以同时定义存储器型数据和 reg 型数据。见下例:

```
parameter  wordsize=16,          //定义两个参数
           memsize=256;
reg [wordsize-1:0]  mem[memsize-1:0],writereg, readreg;
```

尽管 memory 型数据和 reg 型数据的定义格式很相似,但要注意其不同之处。如一个由 n 个 1 位寄存器构成的存储器组是不同于一个 n 位的寄存器的。见下例:

```
reg [n-1:0]  rega;       //一个 n 位的寄存器
reg mema  [n-1:0];       //一个由 n 个 1 位寄存器构成的存储器组
```

一个 n 位的寄存器可以在一条赋值语句里进行赋值,而一个完整的存储器则不行。见下例:

```
rega =0;          //合法赋值语句
mema =0;          //非法赋值语句
```

如果想对 memory 中的存储单元进行读写操作,必须指定该单元在存储器中的地址。下面的写法是正确的:

```
mema[3]=0;     //给 memory 中的第 3 个存储单元赋值为 0
```

进行寻址的地址索引可以是表达式,这样就可以对存储器中的不同单元进行操作。表达式的值可以取决于电路中其他的寄存器的值。例如可以用一个加法计数器来做 RAM 的地址索引。

本小节只对以上几种常用的数据类型和常数进行了介绍，其余的在以后的章节中用到之处再逐一介绍。

2.3 运算符及表达式

Verilog HDL 语言的运算符范围很广，按功能可分为以下几类：

(1) 算术运算符(+，-，×，/，%)；

(2) 赋值运算符(=，<=)；

(3) 关系运算符(>，<，>=，<=)；

(4) 逻辑运算符(&&，||，!)；

(5) 条件运算符(? :)；

(6) 位运算符(~，|，^，&，^~)；

(7) 移位运算符(<<，>>)；

(8) 拼接运算符({ })；

(9) 其 他。

在 Verilog HDL 语言中，运算符所带的操作数是不同的，按其所带操作数的个数运算符可分为 3 种：

(1) 单目运算符(unary operator)：可以带一个操作数，操作数放在运算符的右边；

(2) 二目运算符(binary operator)：可以带两个操作数，操作数放在运算符的两边；

(3) 三目运算符(ternary operator)：可以带三个操作数，这三个操作数用三目运算符分隔开。

见下例：

```
clock = ~ clock;          //~是 1 个单目取反运算符,clock 是操作数
c = a | b;                //|是 1 个二目按位或运算符,a 和 b 是操作数
r = s ? t : u;            //? :是 1 个三目条件运算符,s,t,u 是操作数
```

下面对常用的几种运算符进行介绍。

2.3.1 基本的算术运算符

在 Verilog HDL 语言中，算术运算符又称为二进制运算符，共有下面几种：

(1) + (加法运算符，或正值运算符，如 rega+regb，+3)；

(2) - (减法运算符，或负值运算符，如 rega-3，-3)；

(3) × (乘法运算符，如 rega * 3)；

(4) / (除法运算符，如 5/3)；

(5) % (模运算符，或称为求余运算符，要求%两侧均为整型数据，如 7%3 的值为 1)。

在进行整数除法运算时，结果值要略去小数部分，只取整数部分；而进行取模运算时，结果值的符号位采用模运算式里第一个操作数的符号位。见下例：

模运算表达式	结 果	说 明
10%3	1	余数为 1
11%3	2	余数为 2
12%3	0	余数为 0，即无余数
-10%3	-1	结果取第一个操作数的符号位，所以余数为-1

11%3　　2　　结果取第一个操作数的符号位，所以余数为 2

注意：在进行算术运算操作时，如果某一个操作数有不确定的值 x，则整个结果也为不定值 x。

2.3.2 位运算符

Verilog HDL 作为一种硬件描述语言，是针对硬件电路而言的。在硬件电路中信号有 4 种状态值——1，0，x，z。在电路中信号进行与、或、非时，反映在 Verilog HDL 中则是相应的操作数的位运算。Verilog HDL 提供了以下 5 种位运算符：

(1) ~ ——取反；

(2) &——按位与；

(3) |——按位或；

(4) ^ ——按位异或；

(5) ^ ~ ——按位同或(异或非)。

说明：

(1) 位运算符中除了~ 是单目运算符以外，均为二目运算符，即要求运算符两侧各有一个操作数。

(2) 位运算符中的二目运算符要求对两个操作数的相应位进行运算操作。

下面对各运算符分别进行介绍。

1. “取反”运算符~

~ 是一个单目运算符，用来对一个操作数进行按位取反运算。其运算规则见下表：

~	结　果
1	0
0	1
x	x

举例说明：

```
rega='b1010;  //rega 的初值为 'b1010
rega=~ rega;  //rega 的值进行取反运算后变为 'b0101
```

2. “按位与”运算符 &

按位与运算就是将两个操作数的相应位进行与运算。其运算规则见下表：

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

3. “按位或”运算符|

按位或运算就是将两个操作数的相应位进行或运算。其运算规则见下表：

|	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

4. “按位异或”运算符^（也称之为 XOR 运算符）

按位异或运算就是将两个操作数的相应位进行异或运算。其运算规则见下表：

^	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

5. “按位同或”运算符^~

按位同或运算就是将两个操作数的相应位进行同或运算。其运算规则见下表：

^~	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

6. 不同长度的数据进行位运算

两个长度不同的数据进行位运算时，系统会自动地将两者按右端对齐，位数少的操作数会在相应的高位用 0 填满，以使两个操作数按位进行操作。

2.3.3 逻辑运算符

在 Verilog HDL 语言中存在 3 种逻辑运算符：

(1) &&——逻辑与；

(2) ||——逻辑或；

(3) ！——逻辑非。

“&&”和“||”是二目运算符，它要求有两个操作数，如(a>b)&&(b>c)及(a<b)||(b<c)。“!”是单目运算符，只要求一个操作数，如！(a>b)。下表为逻辑运算的真值表，它表示当 a

和 b 的值为不同的组合时，各种逻辑运算所得到的结果。

a	b	! a	! b	a&&b	a||b
真	真	假	假	真	真
真	假	假	真	假	真
假	真	真	假	假	真
假	假	真	真	假	假

逻辑运算符中“&&”和“||”的优先级别低于关系运算符，“!”高于算术运算符。见下例：

(a>b)&&(x>y)　　可写成：a>b && x>y

(a==b)||(x==y)　　可写成：a==b || x==y

(! a)||(a>b)　　可写成：! a || a>b

为了提高程序的可读性，明确表达各运算符间的优先关系，建议使用括号。

2.3.4 关系运算符

关系运算符共有以下 4 种：

(1) < ——小于；

(2) > ——大于；

(3) <= ——小于或等于；

(4) >= ——大于或等于。

在进行关系运算时，如果声明的关系是假的(flase)，则返回值是 0；如果声明的关系是真的(true)，则返回值是 1；如果某个操作数的值不定，则关系是模糊的，返回值是不定值。

所有的关系运算符有着相同的优先级别。关系运算符的优先级别低于算术运算符的优先级别。见下例：

```
a < size-1          //这种表达方式等同于下面这种表达方式
a < (size-1)
size - ( 1 < a )    //这种表达方式不等同于下面这种表达方式
size - 1 < a
```

从上面的例子可以看出这两种不同运算符的优先级别。当表达式 size-(1<a)进行运算时，关系表达式先被运算，然后返回结果值 0 或 1 被 size 减去；而当表达式 size-1<a 进行运算时，size 先被减去 1，然后再同 a 相比。

2.3.5 等式运算符

在 Verilog HDL 语言中存在 4 种等式运算符：

(1) ==——等于；

(2) !=——不等于；

(3) ===——等于；

(4) !==——不等于。

这 4 个运算符都是二目运算符，它要求有两个操作数。“==”和“!=”又称为逻辑等式运

算符，其结果由两个操作数的值决定。由于操作数中某些位可能是不定值 x 和高阻值 z，结果可能为不定值 x。而“===”和“!==”运算符则不同，它在对操作数进行比较时对某些位的不定值 x 和高阻值 z 也进行比较，两个操作数必须完全一致，其结果才是 1，否则为 0。“===”和“!==”运算符常用于 case 表达式的判别，所以又称为“case 等式运算符”。这 4 个等式运算符的优先级别是相同的。下面画出“==”与“===”的真值表，帮助理解两者间的区别。

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

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

下面举一个例子说明“==”和“===”的区别。

```
if(A==1'bx) $display("AisX");      //当 A 等于 X 时，这个语句不执行
if(A===1'bx) $display("AisX");     //当 A 等于 X 时，这个语句执行
```

2.3.6 移位运算符

在 Verilog HDL 中有两种移位运算符：“<<”(左移位运算符)和“>>”(右移位运算符)。其使用方法如下：

a >> n 或 a << n

a 代表要进行移位的操作数，n 代表要移几位。这两种移位运算都用 0 来填补移出的空位。下面举例说明：

```
module  shift;
  reg [3 : 0]  start, result;
     initial
     begin
        start = 1; //start 在初始时刻设为值 0001
        result = (start<<2);
        //移位后，start 的值 0100，然后赋给 result
     end
endmodule
```

从上面的例子可以看出，start 在移过两位以后，用 0 来填补空出的位。

进行移位运算时应注意移位前后变量的位数，下面给出几个例子：

```
4'b1001<<1=5'b10010;     4'b1001<<2=6'b100100;
1<<6=32'b1000000;        4'b1001>>1=4'b0100; 4'b1001>>4=4'b0000;
```

2.3.7 位拼接运算符

Verilog HDL 语言有一个特殊的运算符：位拼接运算符(concatenation operator){}。用这

个运算符可以把两个或多个信号的某些位拼接起来进行运算操作。其使用方法如下：

{信号 1 的某几位，信号 2 的某几位，……，信号 n 的某几位}

即把某些信号的某些位详细地列出来，中间用逗号分开，最后用大括号括起来表示一个整体信号。见下例：

```
{a,b[3:0],w,3'b101}
```

也可以写成为：

```
{a,b[3],b[2],b[1],b[0],w,1'b1,1'b0,1'b1}
```

在位拼接表达式中不允许存在没有指明位数的信号。这是因为在计算拼接信号位宽的大小时必须知道其中每个信号的位宽。

位拼接还可以用重复法来简化表达式。见下例：

```
{4{w}}    //这等同于{w,w,w,w}
```

位拼接还可以用嵌套的方式来表达。见下例：

```
{b,{3{a,b}}}    //这等同于{b,a,b,a,b,a,b}
```

用于表示重复的表达式，如上例中的 4 和 3，必须是常数表达式。

2.3.8 缩减运算符

缩减运算符(reduction operator)是单目运算符，也有与、或、非运算。其与、或、非运算规则类似于位运算符的与、或、非运算规则，但其运算过程不同。位运算是对操作数的相应位进行与、或、非运算，操作数是几位数则运算结果也是几位数。而缩减运算则不同，缩减运算是对单个操作数进行与、或、非递推运算，最后的运算结果是 1 位的二进制数。缩减运算的具体运算过程是这样的：第 1 步先将操作数的第 1 位与第 2 位进行与、或、非运算；第 2 步将运算结果与第 3 位进行与、或、非运算，依次类推，直至最后一位。

例如：

```
reg [3:0]  B;
reg  C;
C = &B;
```

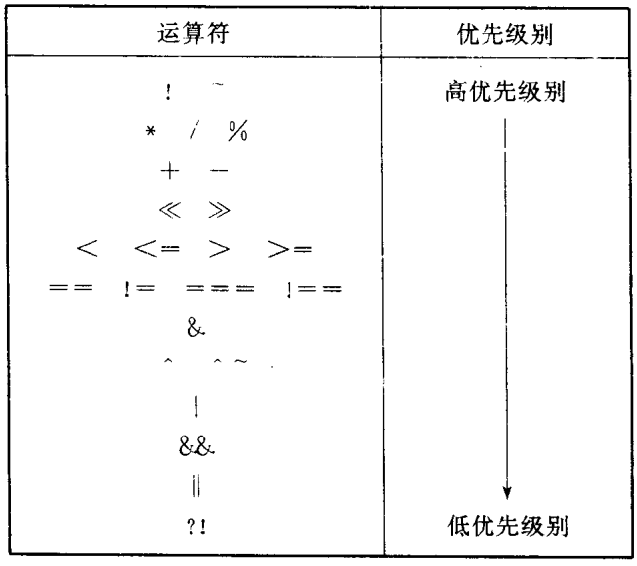
相当于：

```
C = ( (B[0]&B[1]) & B[2] ) & B[3];
```

由于缩减运算符的与、或、非运算规则类似于位运算符与、或、非运算规则，这里不再详细讲述，请参照位运算符的运算规则。

2.3.9 优先级别

下面对各种运算符的优先级别关系做一总结。见下表：



2.3.10 关键词

在 Verilog HDL 中，所有的关键词是事先定义好的确认符，用来组织语言结构。关键词是用小写字母定义的，因此在编写源程序时要注意关键词的书写，以避免出错。下面是 Verilog HDL 中使用的关键词：

always, and, assign, begin, buf, bufif0, bufif1, case, casex, casez, cmos, deassign, default, defparam, disable, edge, else, end, endcase, endmodule, endfunction, endprimitive, endspecify, endtable, endtask, event, for, force, forever, fork, function, highz0, highz1, if, initial, inout, input, integer, join, large, macromodule, medium, module, nand, negedge, nmos, nor, not, notif0, notif1, or, output, parameter, pmos, posedge, primitive, pull0, pull1, pullup, pulldown, rcmos, reg, releses, repeat, mmos, rpmos, rtran, rtranif0, rtranif1, scalared, small, specify, specparam, strength, strong0, strong1, supply0, supply1, table, task, time, tran, tranif0, tranif1, tri, tri0, tri1, triand, trior, trireg, vectored, wait, wand, weak0, weak1, while, wire, wor, xnor, xor。

注意：在编写 Verilog HDL 程序时，变量的定义不要与这些关键词冲突。

2.4 赋值语句和块语句

2.4.1 赋值语句

在 Verilog HDL 语言中，信号有两种赋值方式。

1. 非阻塞(non _ blocking)赋值方式(如 b <= a;)

(1) 块结束后才完成赋值操作；

(2) b 的值并不是立刻就改变的；

(3) 这是一种比较常用的赋值方法。

2. 阻塞(blocking)赋值方式(如 b = a;)

(1) 赋值语句执行完后,块才结束;

(2) b 的值在赋值语句执行结束后立刻就改变;

(3) 可能会产生意想不到的结果。

非阻塞赋值方式和阻塞赋值方式的区别常给设计人员带来问题。问题主要是对"always"块内的 reg 型信号的赋值方式不易把握。到目前为止,前面所举的例子中"always"模块内的 reg 型信号都是采用下面的这种赋值方式:

```
b<=a;
```

这种方式的赋值并不是马上执行的,也就是说"always"块内的下一条语句执行后,b 并不等于 a,而是保持原来的值,"always"块结束后,才进行赋值。而阻塞赋值方式(如 b=a;)是马上执行的,也就是说执行下一条语句时,b 已等于 a。尽管这种方式看起来很直观,但是可能引起麻烦,下面举例说明。

例[1]

```
always @(posedge clk)
  begin
    b<=a;
    c<=b;
  end
```

例[1]的"always"块中用了非阻塞赋值方式,定义了两个 reg 型信号 b 和 c。clk 信号的上升沿到来时,b 就等于 a,c 就等于 b。这里用到了两个触发器。请注意:赋值是在"always"块结束后执行的,c 应为原来 b 的值。这个"always"块实际描述的电路功能如图 2.2 所示。

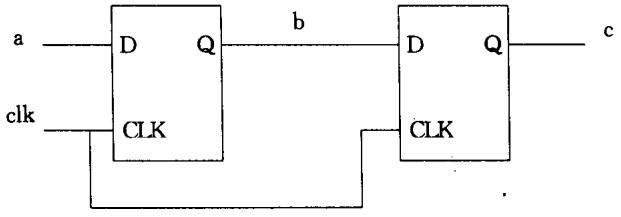


图 2.2 例[1]描述的电路功能

例[2]

```
always @(posedge clk)
  begin
    b=a;
    c=b;
  end
```

例[2]的"always"块用了阻塞赋值方式。clk 信号的上升沿到来时,将发生如下的变化:b 马上取 a 的值,c 马上取 b 的值(即等于 a),生成的电路如图 2.3 所示,只用了一个触发器来寄存 a 的值,又输出给 b 和 c。这大概不是设计者的初衷,如果采用例[1]所示的非阻塞赋值方式

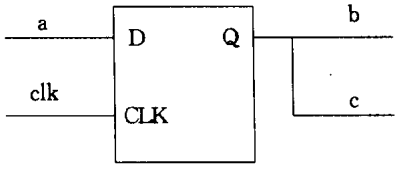


图 2.3 例[2]描述的电路功能

就可以避免这种错误。

2.4.2 块语句

块语句通常用来将两条或多条语句组合在一起，使其在格式上更像一条语句。块语句有两种，一种是 begin _ end 语句，通常用来标识顺序执行的语句，用它来标识的块称为顺序块；一种是 fork _ join 语句，通常用来标识并行执行的语句，用它来标识的块称为并行块。

一、顺序块

顺序块有以下特点：

(1) 块内的语句是按顺序执行的，即只有上面一条语句执行完后下面的语句才能执行。

(2) 每条语句的延迟时间是相对于前一条语句的仿真时间而言的。

(3) 直到最后一条语句执行完，程序流程控制才跳出该语句块。

顺序块的格式如下：

```
begin                          begin：块名
    语句 1;                        块内声明语句
    语句 2;                        语句 1;
    ……                             语句 2;
    语句 n;                        ……
end                                语句 n;
或                             end
```

其中：

- 块名即该块的名字，一个标识名，其作用后面再详细介绍。
- 块内声明语句可以是参数声明语句、reg 型变量声明语句、integer 型变量声明语句及 real 型变量声明语句。

例[1]

```
begin
    areg=breg;
    creg=areg;      //creg 的值为 breg 的值
end
```

从该例可以看出，第 1 条赋值语句先执行，areg 的值更新为 breg 的值，然后程序流程控制转到第 2 条赋值语句，creg 的值更新为 areg 的值。因为这两条赋值语句之间没有任何延迟时间，creg 的值实为 breg 的值。当然，可以在顺序块里延迟控制时间来分开两个赋值语句的执行时间，见下例。

例[2]

```
begin
    areg=breg;
    #10  creg=areg;
    //在两条赋值语句间延迟 10 个时间单位
end
```

例[3]

```
parameter  d=50;      //声明 d 是一个参数
reg [7:0]  r;         //声明 r 是一个 8 位的寄存器变量
begin                 //由一系列延迟产生的波形
```

```
        #d  r = 'h35;
        #d  r = 'hE2;
        #d  r = 'h00;
        #d  r = 'hF7;
        #d  -> end_wave;  //触发事件 end_wave
    end
```

这个例子中用顺序块和延迟控制组合来产生一个时序波形。

二、并行块

并行块有以下 4 个特点：

(1) 块内语句是同时执行的，即程序流程控制一进入到该并行块，块内语句则开始同时并行地执行。

(2) 块内每条语句的延迟时间是相对于程序流程控制进入到块内的仿真时间的。

(3) 延迟时间是用来给赋值语句提供执行时序的。

(4) 当按时间时序排列在最后的语句执行完后，或一个 disable 语句执行时，程序流程控制跳出该程序块。

并行块的格式如下：

```
fork                                fork：块名
    语句 1;                         块内声明语句
    语句 2;                             语句 1;
    ……                                  语句 2;
    语句 n;                             ……
join                                    语句 n;
或                                  join
```

其中：

- 块名即标识该块的一个名字，相当于一个标识符。
- 块内声明语句可以是参数声明语句、reg 型变量声明语句、integer 型变量声明语句、real 型变量声明语句、time 型变量声明语句和事件(event)声明语句。

下面举例说明：

例[4]

```
fork
    #50     r='h35;
    #100    r='hE2;
    #150    r='h00;
    #200    r='hF7;
    #250    -> end_wave;    //触发事件 end_wave
join
```

例[4]用并行块替代了例[3]中的顺序块来产生波形，用这两种方法生成的波形是一样的。

三、块 名

在 Verilog HDL 语言中，可以给每个块取一个名字，只需将名字加在关键词 begin 或 fork 后面即可。这样做的原因有以下几点：

(1) 这样可以在块内定义局部变量,即只在块内使用的变量。

(2) 这样可以允许块被其他语句调用,如被 disable 语句调用。

(3) 在 Verilog 语言里,所有的变量都是静态的,即所有的变量都只有一个唯一的存储地址,因此进入或跳出块并不影响存储在变量内的值。

基于以上原因,块名提供了一个在任何仿真时刻确认变量值的方法。

四、起始时间和结束时间

在并行块和顺序块中都有一个起始时间和结束时间的概念。对于顺序块,起始时间就是第一条语句开始被执行的时间,结束时间就是最后一条语句执行结束的时间。而对于并行块来说,起始时间对于块内所有的语句是相同的,即程序流程控制进入该块的时间,其结束时间是按时间排序在最后的语句执行结束的时间。

当一个块嵌入另一个块时,块的起始时间和结束时间是很重要的。至于跟在块后面的语句只有在该块的结束时间到了才能开始执行,也就是说,只有该块完全执行完后,后面的语句才可以执行。

在 fork _ join 块内,各条语句不必按顺序给出,因此在并行块里,各条语句在前还是在后是无关紧要的。见下例:

例[5]
```
fork
        #250    ->end_wave;
        #200    r='hF7;
        #150    r='h00;
        #100    r='hE2;
        #50     r='h35;
join
```

在这个例子中,各条语句并不是按被执行的先后顺序给出的,但同样可以生成例[4]中的波形。

2.5 条件语句

2.5.1 if _ else 语句

if 语句是用来判定所给的条件是否满足,根据判定的结果(真或假)决定执行给出的两种操作之一。Verilog HDL 语言提供了 3 种形式的 if 语句。

(1) if(表达式)语句

例如:

```
if (a>b)    out1 <= int1;
```

(2) if(表达式)　　语句 1;

　　else　　语句 2;

例如:

```
if(a>b)     out1<=int1;
else        out1<=int2;
```

(3) if(表达式 1)　　语句 1;

```
else    if(表达式 2)    语句 2;
else    if(表达式 3)    语句 3;
……
else    if(表达式 m)    语句 m;
else                    语句 n;
```

例如：

```
if(a>b)    out1<=int1;
   else if(a==b)  out1<=int2;
   else           out1<=int3;
```

说明：

(1) 3 种形式的 if 语句在 if 后面都有“表达式”，一般为逻辑表达式或关系表达式。系统对表达式的值进行判断，若为 0,x,z,按“假”处理；若为 1,按“真”处理，执行指定的语句。

(2) 第 2、第 3 种形式的 if 语句，在每个 else 前面有一分号，整个语句结束处有一分号。

例如：

```
if(a>b)
    out1<=int1;
else                    各有一个分号
    out1<=int2;
```

这是由于分号是 Verilog HDL 语句中不可缺少的部分，是 if 语句中的内嵌套语句所要求的。如果无此分号，则出现语法错误。但应注意，不要误认为上面是两个语句(if 语句和 else 语句)，它们都属于同一个 if 语句。else 子句不能作为语句单独使用，它必须是 if 语句的一部分，与 if 配对使用。

(3) 在 if 和 else 后面可以包含一个内嵌的操作语句，也可以有多个操作语句，此时用 begin 和 end 这两个关键词将几个语句包含起来成为一个复合块语句。如：

```
if(a>b)
    begin
      out1<=int1;
      out2<=int2;
    end
else
    begin
      out1<=int2;
      out2<=int1;
    end
```

注意：在 end 后不需要再加分号，因为 begin _ end 内是一个完整的复合语句。

(4) 允许一定形式的表达式简写方式。如下面的例子：

```
if(expression)    等同与    if(expression==1)
if(expression!)   等同与    if(expression!=1)
```

(5) if 语句的嵌套。

在 if 语句中又包含一个或多个 if 语句称为 if 语句的嵌套。一般形式如下：

```
if(expression1)
        if(expression2)  语句 1     (内嵌 if)
        else     语句 2
else
        if(expression3)  语句 3     (内嵌 if)
        else     语句 4
```

应当注意 if 与 else 的配对关系，else 总是与它上面的最近的 if 配对。如果 if 与 else 的数目不一样，为了实现程序设计者的意图，可以用 begin _ end 块语句来确定配对关系。例如：

```
if(  )
    begin
    if(  )  语句 1     (内嵌 if)
    end
else
    语句 2
```

这时，begin _ end 块语句限定了内嵌 if 语句的范围，因此 else 与第一个 if 配对。注意 begin _ end 块语句在 if _ else 语句中的使用，因为有时 begin _ end 块语句的不慎使用会改变逻辑行为。见下例：

```
if(index>0)
    for(scani=0;scani<index;scani=scani+1)
        if(memory[scani]>0)
          begin
            $display("...");
            memory[scani]=0;
          end
else /* WRONG */
$display("error-indexiszero");
```

尽管程序设计者把 else 写在与第一个 if(外层 if)同一列上，希望与第一个 if 对应，但实际上 else 是与第二个 if 对应的，因为它们相距最近。正确的写法应当是这样的：

```
if(index>0)
    begin
    for(scani=0;scani<index;scani=scani+1)
        if(memory[scani]>0)
            begin
              $display("...");
              memory[scani]=0;
            end
    end
```

```
else / * WRONG * /
 $ display("error-indexiszero");
```

(6) if _ else 举例。

下面的例子是取自某程序的一部分。这部分程序用 if _ else 语句来检测变量 index,以决定 3 个寄存器 modify _ segn 中哪一个的值应当与 index 相加作为 memory 的寻址地址,并且将相加值存入寄存器 index 以备下次检测使用。

```
//定义寄存器和参数
reg [31 : 0]    instruction, segment _ area[255 : 0];
reg [7 : 0]     index;
reg [5 : 0]     modify _ seg1, modify _ seg2, modify _ seg3;
parameter
    segment1=0,   inc _ seg1=1,
    segment2=20,  inc _ seg2=2,
    segment3=64,  inc _ seg3=4,
    data=128;
//检测寄存器 index 的值
if(index<segment2)
     begin
        instruction = segment _ area[index + modify _ seg1];
        index = index + inc _ seg1;
     end
else if(index<segment3)
    begin
        instruction = segment _ area[index + modify _ seg2];
        index = index + inc _ seg2;
    end
else if (index<data)
    begin
       instruction = segment _ area[index + modify _ seg3];
       index = index + inc _ seg3;
    end
else
      instruction = segment _ area[index];
```

2.5.2 case 语句

case 语句是一种多分支选择语句,if 语句只有两个分支可供选择,而实际问题中常常需要用到多分支选择,Verilog 语言提供的 case 语句直接处理多分支选择。case 语句通常用于微处理器的指令译码,它的一般形式如下:

(1) case(表达式)　<case 分支项>　endcase

(2) casez(表达式)　<case 分支项>　endcase

(3) casex(表达式) <case 分支项> endcase

case 分支项的一般格式如下：

分支表达式： 语句

缺省项(default 项)： 语句

说明：

(1) case 括号内的表达式称为控制表达式，case 分支项中的表达式称为分支表达式。控制表达式通常表示为控制信号的某些位，分支表达式则用这些控制信号的具体状态值来表示，因此分支表达式又可以称为常量表达式。

(2) 当控制表达式的值与分支表达式的值相等时，就执行分支表达式后面的语句。如果所有的分支表达式的值都没有与控制表达式的值相匹配的，则执行 default 后面的语句。

(3) default 项可有可无，一个 case 语句里只许有一个 default 项。

下面是一个简单的使用 case 语句的例子。该例子中对寄存器 rega 译码以确定 result 的值。

```
reg [15 : 0]  rega;
reg [9 : 0]   result;
case(rega)
16 'd0 :   result = 10 'b0111111111;
16 'd1 :   result = 10 'b1011111111;
16 'd2 :   result = 10 'b1101111111;
16 'd3 :   result = 10 'b1110111111;
16 'd4 :   result = 10 'b1111011111;
16 'd5 :   result = 10 'b1111101111;
16 'd6 :   result = 10 'b1111110111;
16 'd7 :   result = 10 'b1111111011;
16 'd8 :   result = 10 'b1111111101;
16 'd9 :   result = 10 'b1111111110;
default :   result = 'bx;
endcase
```

(4) 每一个 case 分项的分支表达式的值必须互不相同，否则就会出现矛盾现象(对表达式的同一个值，有多种执行方案)。

(5) 执行完 case 分项后的语句，则跳出该 case 语句结构，终止 case 语句的执行。

(6) 在用 case 语句表达式进行比较的过程中，只有当信号对应位的值能明确进行比较时，比较才能成功。因此，要注意详细说明 case 分项的分支表达式的值。

(7) case 语句的所有表达式值的位宽必须相等，只有这样，控制表达式和分支表达式才能进行对应位的比较。一个经常犯的错误是用 'bx，'bz 来替代 n'bx，n'bz，这样写是不对的，因为信号 x，z 的缺省宽度是机器的字节宽度，通常是 32 位(此处 n 是 case 控制表达式的位宽)。

下面给出 case，casez，casex 的真值表：

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

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

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

case 语句与 if _else _if 语句的区别主要有两点：

(1) 与 case 语句中的控制表达式和多分支表达式这种比较结构相比，if _else _if 结构中的条件表达式更为直观一些。

(2) 对于那些分支表达式中存在值为不定值 x 或高阻值 z 的位时，case 语句提供了处理这种情况的手段。下面的两个例子介绍了处理值为 x，z 的位的 case 语句。

例[1]
```
case ( select[1 : 2])
    2 'b00 :   result = 0;
    2 'b01 :   result = flaga;
    2 'b0x,
    2 'b0z :   result = flaga? 'bx : 0;
    2 'b10 :   result = flagb;
    2 'bx0,
    2 'bz0 :   result = flagb? 'bx : 0;
    default :  result = 'bx;
endcase
```

例[2]
```
case(sig)
    1 'bz :   $display("signal is floating");
    1 'bx :   $display("signal is unknown");
    default :  $display("signal is %b", sig);
endcase
```

Verilog HDL 针对电路的特性提供了 case 语句的其他两种形式，用来处理 case 语句比较过程中的不必考虑的情况(don't care condition)。其中，casez 语句用来处理不考虑高阻值 z 的比较过程，casex 语句则将高阻值 z 和不定值 x 都视为不必关心的情况。所谓不必关心的情况，即在表达式进行比较时，不将该位的状态考虑在内。这样，在 case 语句表达式进行比较时，就可以灵活地设置以对信号的某些位进行比较。见下面的两个例子。

例[3]
```
reg[7 : 0]  ir;
casez(ir)
    8 'b1???????:   instruction1(ir);
    8 'b01??????:   instruction2(ir);
    8 'b00010???:   instruction3(ir);
    8 'b000001??:   instruction4(ir);
endcase
```

例[4]
```
reg[7 : 0]  r, mask;
mask = 8'bx0x0x0x0;
```

```
casex(r ^ mask)
    8 'b001100xx :   stat1;
    8 'b1100xx00 :   stat2;
    8 'b00xx0011 :   stat3;
    8 'bxx001100 :   stat4;
endcase
```

2.5.3 使用条件语句不当生成锁存器的情况

Verilog HDL 设计中容易犯的一个错误是由于不正确使用语言，生成了并不想要的锁存器。图 2.4 给出了一个在“always”块中不正确使用 if 语句，造成这种错误的例子。

```
always@ (alord)
  begin
    if(al)   q<=d;
  end

有锁存器
```

```
always@ (alord)
  begin
    if(al)   q<=d;
    else   q<=0;
  end

无锁存器
```

图 2.4

检查一下左边的“always”块，if 语句保证了只有当 al=1 时，q 才取 d 的值。这段程序没有写出 al = 0 时的结果，那么当 al=0 时会怎么样呢？

在“always”块内，如果在给定的条件下变量没有赋值，这个变量将保持原值，也就是说会生成一个锁存器！

如果设计人员希望当 al = 0 时，q 的值为 0，则 else 项就必不可少了。请注意看右边的“always”块，整个 Verilog 程序模块综合出来后，“always”块对应的部分不会生成锁存器。

Verilog HDL 程序另一种偶然生成锁存器是在使用 case 语句时缺少 default 项的情况下发生的。

case 语句的功能是：在某个信号（如图 2.5 中的 sel）取不同的值时，给另一个信号（如图 2.5 中的 q）赋不同的值。注意看图 2.5 左边的例子，如果 sel=00，q 取 a 值；而 sel=11，q 取 b 的值。这个例子中不清楚的是：如果 sel 取 00 和 11 以外的值时 q 将被赋予什么值？在图 2.5 左边的这个例子中，程序是用 Verilog HDL 写的，即默认为 q 保持原值，这就会自动生成锁存器。

图 2.5 右边的例子很明确，程序中的 case 语句有 default 项，指明了如果 sel 不取 00 或 11 时，编译器或仿真器应赋给 q 的值。该程序所示情况下，q 赋值为 0，因此不需要锁存器。

以上就是怎样来避免偶然生成锁存器的错误。如果用到 if 语句，最好写上 else 项；如果用 case 语句，最好写上 default 项。遵循上面两条原则，就可以避免发生这种错误，同时也增强了 Verilog 程序的可读性。

```
always @(sel [1:0] or a or b)
  case(sel [1:0])
    'b00:  q<=a;
    'b11:  q<=b;
  endcase
```

有锁存器

```
always @(sel [1:0] or a or b)
  case(sel [1:0])
    'b00:  q<=a;
    'b11:  q<=b;
    default:  q<='b0;
  endcase
```

无锁存器

图 2.5

2.6 循环语句

在 Verilog HDL 中存在着 4 种类型的循环语句，用来控制执行语句的执行次数。

(1) forever——连续的执行语句。

(2) repeat——连续执行一条语句 n 次。

(3) while——执行一条语句直到某个条件不满足。如果一开始条件即不满足(为假)，则语句一次也不能被执行。

(4) for 通过以下 3 个步骤来决定语句的循环执行：

① 先给控制循环次数的变量赋初值。

② 判定控制循环的表达式的值，如为假则跳出循环语句，如为真则执行指定的语句后，转到第 3 步。

③ 执行一条赋值语句来修正控制循环变量次数的变量的值，然后返回第 2 步。

下面对各种循环语句进行详细的介绍。

2.6.1 forever 语句

forever 语句的格式如下：

```
forever    语句;
```

或

```
forever    begin    多条语句    end
```

forever 循环语句常用于产生周期性的波形，作为仿真测试信号。它与 always 语句不同之处在于它不能独立写在程序中，而必须写在 initial 块中。

2.6.2 repeat 语句

repeat 语句的格式如下：

```
repeat(表达式)  语句;
```

或

```
repeat(表达式)    begin    多条语句    end
```

在 repeat 语句中，其表达式通常为常量表达式。下面的例子中使用 repeat 循环语句及加法和移位操作来实现一个乘法器。

```
parameter  size=8,longsize=16;
reg [size : 1]  opa, opb;
reg [longsize : 1]  result;

begin :  mult
          reg [longsize : 1] shift_opa, shift_opb;
          shift_opa = opa;
          shift_opb = opb;
          result = 0;
          repeat(size)
             begin
                if(shift_opb[1])
                     result = result + shift_opa;

                shift_opa = shift_opa <<1;
                shift_opb = shift_opb >>1;
             end
end
```

2.6.3 while 语句

while 语句的格式如下：

while（表达式）语句

或

while（表达式）begin　多条语句　end

下例用 while 循环语句对 rega 这个 8 位二进制数中值为 1 的位进行计数。

```
begin :  count1s
   reg[7 : 0]  tempreg;
   count=0;
   tempreg = rega;
   while(tempreg)
      begin
         if(tempreg[0]) count = count + 1;
         tempreg = tempreg>>1;
      end
end
```

2.6.4 for 语句

for 语句的一般形式为：

for　（表达式 1;表达式 2;表达式 3）　语句

它的执行过程为：

(1) 求解表达式 1。

(2) 求解表达式 2，若其值为真(非 0)，则执行 for 语句中指定的内嵌语句，然后执行下面的第 3 步；若为假(0)，则结束循环，转到第 5 步。

(3) 若表达式 2 为真，在执行指定的语句后，求解表达式 3。

(4) 转回第 2 步继续执行。

(5) 执行 for 语句下面的语句。

for 语句最简单的应用形式是很容易理解的，其形式如下：

```
for(循环变量赋初值;循环结束条件;循环变量增值)
    执行语句
```

for 循环语句实际上相当于采用 while 循环语句建立以下的循环结构：

```
begin
  循环变量赋初值;
  while (循环结束条件)
      begin
        执行语句
        循环变量增值;
      end
end
```

这样，对于需要 8 条语句才能完成的一个循环控制，for 循环语句只需两条即可。

下面分别举两个使用 for 循环语句的例子。例[1]用 for 语句来初始化 memory，例[2]则用 for 循环语句来实现前面用 repeat 语句实现的乘法器。

例[1]

```
begin : init_mem
  reg [7 : 0] tempi;
    for(tempi=0;tempi<memsize;tempi=tempi+1)
      memory[tempi]=0;
end
```

例[2]

```
parameter size = 8, longsize = 16;
reg[size : 1] opa, opb;
reg[longsize : 1] result;

begin : mult
  integer bindex;
  result=0;
  for( bindex=1; bindex<=size; bindex=bindex+1 )
      if(opb[bindex])
        result = result + (opa<<(bindex-1));
end
```

在 for 语句中，循环变量增值表达式可以不必是一般的常规加法或减法表达式。下面是对 rega 这个 8 位二进制数中值为 1 的位进行计数的另一种方法。见下例：

```
begin : countls
   reg[7 : 0] tempreg;
   count=0;
   for(tempreg=rega; tempreg; tempreg=tempreg>>1)
   if(tempreg[0])
      count=count+1;
end
```

2.7 结构说明语句

Verilog 语言中的任何过程模块都从属于以下 4 种结构的说明语句：

(1) initial 说明语句；

(2) always 说明语句；

(3) task 说明语句；

(4) function 说明语句。

initial 和 always 说明语句在仿真的一开始即开始执行。initial 语句只执行一次。相反，always 语句则是不断地重复执行，直到仿真过程结束。在一个模块中，使用 initial 和 always 语句的次数是不受限制的。

task 和 function 语句可以在程序模块中的一处或多处调用，其具体使用方法以后再详细介绍，这里只对 initial 和 always 语句加以介绍。

2.7.1 initial 语句

initial 语句的格式如下：

```
initial
   begin
     语句 1;
     语句 2;
     ……
     语句 n;
   end
```

下面举例说明。

例[1]

```
initial
   begin
     areg=0;      //初始化寄存器 areg
     for(index=0;index<size;index=index+1)
         memory[index]=0;      //初始化一个 memory
   end
```

在这个例子中则是用 initial 语句在仿真的初始状态对各变量进行初始化。

例[2]

```
initial
   begin
```

```
        inputs = 'b000000;          //初始时刻为 0
        #10  inputs = 'b011001;
        #10  inputs = 'b011011;
        #10  inputs = 'b011000;
        #10  inputs = 'b001000;
    end
```

从这个例子中，我们可以看到 initial 语句的另一用途，即生成激励波形作为电路的仿真信号。

2.7.2 always 语句

always 语句在仿真过程中是不断重复执行的。其声明格式如下：

always <时序控制> <语句>

always 语句由于其不断重复执行的特性，只有和一定的时序控制结合在一起才有用。如果一个 always 语句没有时序控制，则这个 always 语句将会生成一个仿真死锁。见下例：

例[1] always areg = ~ areg;

这个 always 语句将会生成一个 0 延迟的无限循环跳变过程，这时会发生仿真死锁。如果加上时序控制，则这个 always 语句将变为一条非常有用的描述语句。见下例：

例[2] always #half_period areg=~ areg;

这个例子生成了一个周期为 2 * half_period 的无限延续的信号波形。

例[3]
```
reg[7 : 0] counter;
reg  tick;
always @(posedge areg)
        begin
          tick = ~ tick;
          counter = counter + 1;
        end
```

这个例子中，每当 areg 信号的上升沿出现时则把 tick 信号反相，并且 counter 增加 1。这种时间控制是 always 语句最常用的。

2.7.3 task 和 function 说明语句

task 和 function 说明语句分别用来定义任务和函数。利用任务和函数可以把一个很大的程序模块分解成许多较小的任务和函数，便于理解和调试。输入、输出和总线信号的值可以传入、传出任务和函数。任务和函数往往还是大的程序模块中在不同地点多次用到的相同的程序段。学会使用 task 和 function 语句可以简化程序的结构，使程序简明易懂，是编写较大型模块的基本功。

一、task 和 function 说明语句的不同点

任务和函数有些不同，主要表现为以下 4 点：

(1) 函数只能与主模块共用同一个仿真时间单位，而任务可以定义自己的仿真时间单位。

(2) 函数不能启动任务，而任务能启动其他任务和函数。

(3) 函数至少要有一个输入变量，而任务可以没有或有多个任何类型的变量。

(4) 函数返回一个值,而任务则不返回值。

函数的目的是通过返回一个值来响应输入信号的值。任务却能支持多种目的,能计算多个结果,这些结果只能通过被调用的任务的输出或总线端口送出。Verilog HDL 模块使用函数时是把它当作表达式中的操作符,这个操作的结果就是这个函数的返回值。

例如,定义一任务或函数对一个 16 位的字进行操作,让高字节与低字节互换,把它变为另一个字(假定这个任务或函数名为: switch _ bytes)。

任务返回的新字是通过输出端口的变量,因此 16 位字字节互换任务的调用源码是这样的:

```
switch _ bytes(old _ word,new _ word);
```

任务 switch _ bytes 把输入 old _ word 的字的高、低字节互换放入 new _ word 端口输出,而函数返回的新字是通过函数本身的返回值,因此 16 位字字节互换函数的调用源码是这样的:

```
new _ word = switch _ bytes(old _ word);
```

下面分别介绍任务和函数语句的要点。

二、task 说明语句

如果传给任务的变量值和任务完成后接收结果的变量已定义,就可以用一条语句启动任务。任务完成以后控制就传回启动过程。如任务内部有定时控制,则启动的时间可以与控制返回的时间不同。任务可以启动其他的任务,其他任务又可以启动另外的任务,可以启动的任务数是没有限制的。不管有多少任务启动,只有当所有的任务启动完成以后,控制才能返回。

1. 任务的定义

定义任务的语法如下。

任务:

```
task <任务名>;
    <端口及数据类型声明语句>
    <语句 1>
    <语句 2>
    ……
    <语句 n>
endtask
```

这些声明语句的语法与模块定义中的对应声明语句的语法是一致的。

2. 任务的调用及变量的传递

启动任务并传递输入、输出变量的声明语句的语法如下。

任务的调用:

```
<任务名>(端口 1,端口 2,……,端口 n);
```

下面的例子说明怎样定义任务和调用任务。

任务定义:

```
task  my _ task (a,b,c,d,e);
  input  a, b;
  inout  c;
  output  d, e;
```

```
    <statement>      //执行任务工作相应的语句
    c = foo1;        //赋初始值
    d = foo2;        //对任务的输出变量赋值
    e = foo3;
  endtask
```

任务调用：

```
my_task(v,w,x,y,z);
```

任务调用变量(v,w,x,y,z)和任务定义的 I/O 变量(a,b,c,d,e)之间是一一对应的。当任务启动时，由 v,w 和 x 传入的变量赋给了 a,b 和 c，而当任务完成后的输出又通过 c,d 和 e 赋给了 x,y 和 z。

下面是一个具体的例子用来说明怎样在模块的设计中使用任务，使程序容易读懂。

```
module  traffic_lights;
  reg  clock, red, amber, green;
  parameter  on=1, off=0, red_tics=350,
  amber_tics=30,green_tics=200;
  //交通灯初始化
      initial    red=off;
      initial    amber=off;
      initial    green=off;
  //交通灯控制时序
      always
        begin
            red=on;                      //开红灯
            light(red,red_tics);         //调用等待任务
            green=on;                    //开绿灯
            light(green,green_tics);     //等待
            amber=on;                    //开黄灯
            light(amber,amber_tics);     //等待
        end
  //定义交通灯开启时间的任务
  task  light(color,tics);
    output  color;
    input[31:0]  tics;
  begin
     repeat(tics) @(posedge clock);//等待 tics 个时钟的上升沿
     color=off;//关灯
     end
  endtask
  //产生时钟脉冲的 always 块
  always
     begin
        #100  clock=0;
```

```
        #100  clock=1;
    end
endmodule
```

这个例子描述了一个简单的交通灯的时序控制，并且该交通灯有它自己的时钟产生器。

三、function 说明语句

函数的目的是返回一个用于表达式的值。

1. 定义函数的语法

```
function <返回值的类型或范围>(函数名);
    <端口说明语句>
    <变量类型说明语句>
    begin
    <语句>
    ……
    end
endfunction
```

请注意：<返回值的类型或范围>这一项是可选项，如缺省则返回值为一位寄存器类型数据。下面举例说明：

```
function [7:0]  getbyte;
input [15:0]  address;
begin
        <说明语句>                          //从地址字中提取低字节的程序
        getbyte = result_expression;        //把结果赋予函数的返回字节
end
endfunction
```

2. 从函数返回的值

函数的定义蕴含声明了与函数同名的、函数内部的寄存器。如在函数的声明语句中<返回值的类型或范围>为缺省，则这个寄存器是一位的，否则是与函数定义中<返回值的类型或范围>一致的寄存器。函数的定义把函数返回值所赋给寄存器的名称初始化为与函数同名的内部变量。上面的例子说明了这个概念：getbyte 被赋予的值就是函数的返回值。

3. 函数的调用

函数的调用是通过将函数作为表达式中的操作数来实现的。其调用格式如下：

<函数名>(<表达式><,<表达式>>*)

其中，函数名作为确认符。下面的例子通过对两次调用函数 getbyte 的结果进行位拼接运算来生成一个字。

```
word = control? {getbyte(msbyte),getbyte(lsbyte)} : 0;
```

4. 函数的使用规则

与任务相比较函数的使用有较多的约束，下面给出的是函数的使用规则：

(1) 函数的定义不能包含有任何的时间控制语句，即任何用#，@或 wait 来标识的语句。

(2) 函数不能启动任务。

(3) 定义函数时至少要有一个输入参量。

(4) 在函数的定义中必须有一条赋值语句给函数中的一个内部变量赋以函数的结果值，该内部变量具有和函数名相同的名字。

5. 举例说明

下面的例子定义了一个叫 factorial 的函数，该函数返回一个 32 位的寄存器类型的值，该函数可后向调用自身，并且打印出部分结果值。

```
module  tryfact;
        //函数的定义----------------------------------------
        function[31 : 0]  factorial;
          input[3 : 0]  operand;
          reg[3 : 0]  index;
          begin
            factorial=operand? 1 : 0;
            for(index=2;index<=operand;index=index+1)
            factorial = index * factorial;
          end
        endfunction
        //函数的测试----------------------------------------
        reg[31 : 0]  result;
        reg[3 : 0]  n;
        initial
        begin
          result=1;
          for(n=2;n<=9;n=n+1)
          begin
            $display("Partial result n= %d result= %d", n, result);
            result = n * factorial(n)/((n * 2)+1);
          end
           $display("Finalresult=%d",result);
          end
endmodule//模块结束
```

前面已经介绍了足够的语句类型，可以编写一些完整的模块。第三章我们将举许多实际的例子进行介绍。这些例子都给出了完整的模块描述，因此可以对它们进行仿真测试和结果检验。通过学习和练习我们就能逐步掌握利用 Verilog HDL 设计数字系统的方法和技术。

2.8 系统函数和任务

Verilog HDL 语言中共有以下一些系统函数和任务：$bitstoreal，$rtoi，$display，$setup，$finish，$skew，$hold，$setuphold，$itor，$strobe，$period，$time，$printtimescale，$timefoemat，$realtime，$width，$real tobits，$write，$recovery。

在 Verilog HDL 语言中每个系统函数和任务前面都用一个标识符 $ 加以确认。这些系统函数和任务提供了非常强大的功能。下面对一些常用的系统函数和任务逐一加以介绍。

2.8.1 $display 和 $write 任务

格式：

```
$display(p1,p2,…,pn);
$write(p1,p2,…,pn);
```

这两个函数和系统任务是用来输出信息，即将参数 p2～pn 按参数 p1 给定的格式输出。参数 p1 通常称为“格式控制”，参数 p2～pn 通常称为“输出表列”。这两个任务的作用基本相同。$display 自动地在输出后进行换行，$write 则不是这样。如果想在一行里输出多个信息，可以使用 $write。在 $display 和 $write 中，其输出格式控制是用双引号括起来的字符串，它包括两种信息。

1. 格式说明

由“%”和格式字符组成。它的作用是将输出的数据转换成指定的格式输出。格式说明总是由“%”字符开始的。对于不同类型的数据用不同的格式输出。表 2.1 中给出了常用的几种输出格式。

表 2.1 输出格式说明

输出格式	说　明
%h 或%H	以十六进制数的形式输出
%d 或%D	以十进制数的形式输出
%o 或%O	以八进制数的形式输出
%b 或%B	以二进制数的形式输出
%c 或%C	以 ASCII 码字符的形式输出
%v 或%V	输出网络型数据信号强度
%m 或%M	输出等级层次的名字
%s 或%S	以字符串的形式输出
%t 或%T	以当前的时间格式输出
%e 或%E	以指数的形式输出实型数
%f 或%F	以十进制数的形式输出实型数
%g 或%G	以指数或十进制数的形式输出实型数，无论何种格式都以较短的结果输出

2. 普通字符

即需要原样输出的字符。其中一些特殊的字符可以通过表 2.2 中的转换序列来输出。表 2.2 的字符形式用于格式字符串参数中，用来显示特殊的字符。

表 2.2 换码序列的功能

换码序列	功 能
\n	换行
\t	横向跳格(即跳到下一个输出区)
\\	反斜杠字符\
\"	双引号字符"
\o	1～3 位八进制数代表的字符
%%	百分符号%

在 $display 和 $write 的参数列表中,"输出表列"是需要输出的一些数据,可以是表达式。下面举几个例子加以说明。

例[1]
```
module  disp;
    initial
      begin
        $display("\\\t%%\n\"\123");
      end
endmodule
```

输出结果为:

```
\%
"S
```

从上面的这个例子中可以看到一些特殊字符的输出形式(八进制数 123 就是字符 S)。

例[2]
```
module  disp;
    reg[31 : 0]  rval;
    pulldown(pd);
    initial
      begin
      rval=101;
        $display("rval=%h hex %d decimal", rval, rval);
        $display("rval=%o otal %b binary", rval, rval);
        $display("rval has %c ascii character value",rval);
        $display("pd strength value is %v",pd);
        $display("current scope is %m");
        $display("%s is ascii value for 101",101);
        $display("simulation time is %t", $time);
      end
endmodule
```

其输出结果为:

```
rval=00000065 hex 101 decimal
rval=00000000145 octal 00000000000000000000000001100101 binary
rval has e ascii character value
```

```
pd strength value is StX
current scope is disp
e is ascii value for 101
simulation time is 0
```

在 $display 中，输出列表中数据的显示宽度是自动按照输出格式进行调整的。这样，在显示输出数据时，经过格式转换以后，总是用表达式的最大可能值所占的位数来显示表达式的当前值。在用十进制数格式输出时，输出结果前面的 0 值用空格来代替。对于其他进制，输出结果前面的 0 仍然显示出来。例如，对于一个值的位宽为 12 位的表达式，如按照十六进制数输出，则输出结果占 3 个字符的位置，如按照十进制数输出，则输出结果占 4 个字符的位置。这是因为这个表达式的最大可能值为 FFF(十六进制)、4095(十进制)。可以通过在“%”和表示进制的字符中间插入一个“0”，自动调整显示输出数据宽度的方式。见下例：

```
$display("d=%0h a=%0h",data,addr);
```

这样，在显示输出数据时，在经过格式转换以后，总是用最少的位数来显示表达式的当前值。下面举例说明。

例[3]

```
module  printval;
    reg[11 : 0]  r1;
    initial
    begin
       r1=10;
        $display("Printing with maximum size=%d=%h",r1,r1);
        $display("Printing with minimum size=%0d=%0h",r1,r1);
    end
enmodule
```

输出结果为：

```
Printing with maximum size=10=00a
Printing with minimum size=10=a
```

如果输出列表中表达式的值包含有不确定的值或高阻值，其结果输出遵循以下规则。

1. 输出格式为十进制

(1) 如果表达式值的所有位均为不定值，则输出结果为小写的 x。

(2) 如果表达式值的所有位均为高阻值，则输出结果为小写的 z。

(3) 如果表达式值的部分位为不定值，则输出结果为大写的 X。

(4) 如果表达式值的部分位为高阻值，则输出结果为大写的 Z。

2. 输出格式为十六进制和八进制

(1) 每 4 位二进制数为一组代表 1 位十六进制数，每 3 位二进制数为一组代表 1 位八进制数。

(2) 如果表达式值相对应的某进制数的所有位均为不定值，则该位进制数的输出结果为小写的 x。

(3) 如果表达式值相对应的某进制数的所有位均为高阻值，则该位进制数的输出结果为小写的 z。

(4) 如果表达式值相对应的某进制数的部分位为不定值,则该位进制数输出结果为大写的X。

(5) 如果表达式值相对应的某进制数的部分位为高阻值,则该位进制数输出结果为大写的Z。

对于二进制输出格式,表达式值的每一位的输出结果为0,1,x,z。下面举例说明:

语　句	输出结果
$display("%d", 1'bx);	x
$display("%h", 14'bx0_1010);	xxXa
$display("%h %o",12'b001x_xx10_1x01,12'b001_xxx_101_x01);	XXX 1x5X

因为$write在输出时不换行,要注意它的使用。可以在$write中加入换行符\n,以确保明确的输出显示格式。

2.8.2 系统任务$monitor

格式:

```
$monitor(p1,p2,…,pn);
$monitor;
$monitoron;
$monitoroff;
```

任务$monitor具有监控和输出参数列表中的表达式或变量值的功能。其参数列表中输出控制格式字符串和输出表列的规则与$display中的一样。当启动一个带有一个或多个参数的$monitor任务时,仿真器则建立一个处理机制,使得每当参数列表中变量或表达式的值发生变化时,整个参数列表中变量或表达式的值都将输出显示。如果同一时刻,两个或多个参数的值发生变化,则在该时刻只输出显示一次。但在$monitor中,参数可以是$time系统函数,这样,参数列表中变量或表达式的值同时发生变化的时刻可以通过标明同一时刻的多行输出来显示。如:

```
$monitor($time,,"rxd=%b txd=%b",rxd,txd);
```

在$display中也可以这样使用。注意,在上面的语句中,",,"代表一个空参数。空参数在输出时显示为空格。

$monitoron和$monitoroff这两个任务的作用是通过打开和关闭监控标志来控制、监控任务$monitor的启动和停止,这样使得程序员可以很容易地控制$monitor何时发生。其中$monitoroff任务用于关闭监控标志,停止监控任务$monitor;$monitoron则用于打开监控标志,启动监控任务$monitor。通常在调用$monitoron来启动$monitor时,不管$monitor参数列表中的值是否发生变化,总是立刻输出显示当前时刻参数列表中的值,这样可在监控的初始时刻设定初始比较值。在缺省情况下,控制标志在仿真的起始时刻就已经打开了。在多模块调试的情况下,许多模块中都调用了$monitor,因为任何时刻只能有一个$monitor起作用,因此须配合$monitoron与$monitoroff使用,把需要监视的模块用$monitoron打开,在监视完毕后及时用$monitoroff关闭,以便把$monitor让给其他模块使用。$monitor与$display的不同之处还在于$monitor往往在initial块中调用,只要不调用$monitoroff,$monitor便不间断地对其所设定的信号进行监视。

2.8.3 时间度量系统函数 $time

在 Verilog HDL 中有两种类型的时间系统函数：$time 和 $realtime。用这两个时间系统函数可以得到当前的仿真时刻。

一、系统函数 $time

$time 可以返回一个以 64 位的整数来表示的当前仿真时刻值。该时刻是以模块的仿真时间尺度为基准的。下面举例说明。

例[1]

```
'timescale  10ns/1ns
module  test;
    reg  set;
    parameter  p=1.6;
    initial
      begin
        $monitor($time,,"set=",set);
        #p  set=0;
        #p  set=1;
      end
endmodule
```

输出结果为：

```
0 set=x
2 set=0
3 set=1
```

在这个例子中，模块 test 想在时刻为 16ns 时将寄存器 set 设置为 0，在时刻为 32ns 时将寄存器 set 设置为 1。但是由 $time 记录的 set 变化时刻却和预想的不一样。这是由下面两个原因引起的：

(1) $time 显示时刻受时间尺度比例的影响。在例[1]中，时间尺度是 10ns，因为 $time 输出的时刻总是时间尺度的倍数，这样将 16ns 和 32ns 输出为 1.6 和 3.2。

(2) 因为 $time 总是输出整数，所以在输出经过尺度比例变换的数字时，要先进行取整。在例[1]中，1.6 和 3.2 经取整后输出为 2 和 3。注意：时间的精确度并不影响数字的取整。

二、$realtime 系统函数

$realtime 和 $time 的作用是一样的，只是 $realtime 返回的时间数字是一个实型数，该数字也是以时间尺度为基准的。下面举例说明。

例[2]

```
'timescale  10ns/1ns
module  test;
    reg  set;
    parameter  p=1.55;
    initial
      begin
        $monitor($realtime,,"set=",set);
        #p  set=0;
        #p  set=1;
```

```
        end
    endmodule
```

输出结果为：

```
0 set=x
1.6 set=0
3.2 set=1
```

从例[2]可以看出，$realtime 将仿真时刻经过尺度变换以后即输出，不需进行取整操作。所以，$realtime 返回的时刻是实型数。

2.8.4 系统任务 $finish

格式：

```
$finish;
$finish(n);
```

系统任务 $finish 的作用是退出仿真器，返回主操作系统，也就是结束仿真过程。任务 $finish 可以带参数，根据参数的值输出不同的特征信息。如果不带参数，默认 $finish 的参数值为 1。下面给出了对于不同的参数值，系统输出的特征信息：

0——不输出任何信息；

1——输出当前仿真时刻和位置；

2——输出当前仿真时刻、位置和在仿真过程中所用 memory 及 CPU 时间的统计。

2.8.5 系统任务 $stop

格式：

```
$stop;
$stop(n);
```

$stop 的作用是把 EDA 工具(例如仿真器)置成暂停模式，给出一个交互式的命令提示符，将控制权交给用户。这个任务可以带有参数表达式。根据参数值(0,1 或 2)的不同，输出不同的信息。参数值越大，输出的信息越多。

2.8.6 系统任务 $readmemb 和 $readmemh

在 Verilog HDL 程序中有两个系统任务 $readmemb 和 $readmemh 用来从文件中读取数据到存储器中。这两个系统任务可以在仿真的任何时刻被执行，其使用格式共有以下 6 种：

(1) $readmemb ("<数据文件名>",<存储器名>);

(2) $readmemb ("<数据文件名>",<存储器名>,<起始地址>);

(3) $readmemb ("<数据文件名>",<存储器名>,<起始地址>,<结束地址>);

(4) $readmemh ("<数据文件名>",<存储器名>);

(5) $readmemh ("<数据文件名>",<存储器名>,<起始地址>);

(6) $readmemh ("<数据文件名>",<存储器名>,<起始地址>,<结束地址>);

在这两个系统任务中，被读取的数据文件的内容只能包含：空白位置(空格、换行、制表格(tab)和 form-feeds)、注释行("//"形式的和"/*……*/"形式的都允许)、二进制或十六进制

的数字。数字中不能包含位宽说明和格式说明。对于 $readmemb 系统任务，每个数字必须是二进制数字，对于 $readmemh 系统任务，每个数字必须是十六进制数字。数字中不定值 x 或 X、高阻值 z 或 Z 和下划线(_)的使用方法及代表的意义与一般 Verilog HDL 程序中的用法及意义是一样的。另外，数字必须用空白位置或注释行来分隔开。

在下面的讨论中，地址一词指对存储器(memory)建模的数组的寻址指针。当数据文件被读取时，每一个被读取的数字都被存放到地址连续的存储器单元中去。存储器单元的存放地址范围由系统任务声明语句中的起始地址和结束地址来说明，每个数据的存放地址在数据文件中进行说明。当地址出现在数据文件中，其格式为字符“@”后跟上十六进制数。如：

@hh…h

对于这个十六进制的地址数，允许大写和小写的数字。在字符“@”和数字之间不允许存在空白位置。可以在数据文件里出现多个地址。当系统任务遇到一个地址说明时，系统任务将该地址后的数据存放到存储器中相应的地址单元中去。

对于上面 6 种系统任务格式，需补充说明以下 5 点：

(1) 如果系统任务声明语句中和数据文件里都没有进行地址说明，则缺省的存放起始地址为该存储器定义语句中的起始地址。数据文件里的数据被连续存放到该存储器中，直到该存储器单元存满或数据文件里的数据存完为止。

(2) 如果系统任务中说明了存放的起始地址，没有说明存放的结束地址，则数据从起始地址开始存放，存放到该存储器定义语句中的结束地址为止。

(3) 如果在系统任务声明语句中，起始地址和结束地址都进行了说明，则数据文件里的数据从该起始地址开始存放到存储器单元中，直到结束地址，而不考虑该存储器的定义语句中的起始地址和结束地址。

(4) 如果地址信息在系统任务和数据文件里都进行了说明，那么数据文件里的地址必须在系统任务中地址参数声明的范围之内，否则将提示错误信息，并且将数据装载到存储器中的操作被中断。

(5) 如果数据文件里的数据个数和系统任务中起始地址及结束地址暗示的数据个数不同，也要提示错误信息。

下面举例说明。

先定义一个有 256 个地址的字节存储器 mem。

```
reg[7:0]  mem[1:256];
```

下面给出的系统任务以各自不同的方式装载数据到存储器 mem 中。

```
initial $readmemh  ("mem.data",mem);
initial $readmemh  ("mem.data",mem,16);
initial $readmemh  ("mem.data",mem,128,1);
```

第 1 条语句在仿真时刻为 0 时，将数据装载到以地址为 1 的存储器单元为起始存放单元的存储器中去；第 2 条语句将数据装载到以单元地址为 16 的存储器单元为起始存放单元的存储器中去，一直到地址为 256 的单元为止；第 3 条语句从地址为 128 的单元开始装载数据，一直到地址为 1 的单元。在第 3 种情况中，当装载完毕，系统要检查在数据文件里是否有 128 个数据，如果没有，系统将提示错误信息。

2.8.7 系统任务 $random

这个系统函数提供了一个产生随机数的手段。当函数被调用时返回一个 32 位的随机数。它是一个带符号的整形数。

$random 一般的用法是：$ramdom % b ，其中 b>0。它给出了一个范围在(−b+1)：(b−1)中的随机数。下面给出一个产生随机数的例子：

```
reg[23 : 0]  rand;
rand = $random % 60;
```

上面的例子给出了一个范围在−59～59 之间的随机数，下面的例子通过位拼接操作产生一个值在 0～59 之间的数。

```
reg[23 : 0]  rand;
rand = {$random} % 60;
```

利用这个系统函数可以产生随机脉冲序列或宽度随机的脉冲序列，以用于电路的测试。下面例子中的 Verilog HDL 模块可以产生宽度随机的随机脉冲序列的测试信号源，在电路模块的设计仿真时非常有用。可以根据测试的需要，模仿下例，灵活使用 $random 系统函数编制出与实际情况类似的随机脉冲序列。

```
'timescale  1ns/1ns
module  random_pulse( dout );
output [9 : 0]  dout;
reg  dout;
integer delay1,delay2,k;
  initial
  begin
     #10  dout=0;
     for (k=0; k< 100; k=k+1)
        begin
        delay1 = 20 * ( $random % 6);
        // delay1 在 0～100 ns 间变化
        delay2 = 20 * ( 1 + $random % 3);
        // delay2 在 20 ns～60 ns 间变化
        #delay1  dout = 1 << ($random %10);
        //dout 的 0～9 位中随机出现 1，并且出现的时间在 0～100ns 间变化
        #delay2  dout = 0;
        //脉冲的宽度在 20 ns～60 ns 间变化
        end
    end
endmodule
```

2.9 编译预处理

Verilog HDL 语言和 C 语言一样也提供了编译预处理的功能。“编译预处理”是 Verilog

HDL 编译系统的一个组成部分。Verilog HDL 语言允许在程序中使用几种特殊的命令(它们不是一般的语句)。Verilog HDL 编译系统通常先对这些特殊的命令进行“预处理”,然后将预处理的结果和源程序一起再进行通常的编译处理。

在 Verilog HDL 语言中,为了和一般的语句相区别,这些预处理命令以符号“ '”开头(注意这个符号是不同于单引号“ '”的)。这些预处理命令的有效作用范围为定义命令之后到本文件结束或到其他命令定义替代该命令之处。Verilog HDL 提供了以下预编译命令:

'accelerate, 'autoexpand _ vectornets, 'celldefine, 'default _ nettype, 'define, 'else, 'endcelldefine, 'endif, 'endprotect, 'endprotected, 'expand _ vectornets, 'ifdef, 'include, 'noaccelerate, 'noexpand _ vectornets, 'noremove _ gatenames, 'noremove _ netnames, 'nounconnected _ drive. 'protect, 'protecte, 'remove _ gatenames, 'remove _ netnames, 'reset, 'timescale, 'unconnected _ drive。

本节只对常用的 'define, 'include, 'timescale 进行介绍,其余的请查阅参考书。

2.9.1 宏定义 'define

用一个指定的标识符(即名字)来代表一个字符串,它的一般形式为:

'define 标识符(宏名) 字符串(宏内容)

如:

```
'define signal string
```

它的作用是指定用标识符 signal 来代替 string 这个字符串,在编译预处理时,把程序中在该命令以后所有的 signal 都替换成 string。这种方法使用户能以一个简单的名字代替一个长的字符串,因此把这个标识符(名字)称为“宏名”。在编译预处理时将宏名替换成字符串的过程称为“宏展开”。'define 是宏定义命令。

例[1]

```
'define WORDSIZE 8
module
reg['1 : 'WORDSIZE] data; //这相当于定义 reg[1 : 8] data;
```

关于宏定义的 8 点说明:

(1) 宏名可以用大写字母表示,也可以用小写字母表示。建议使用大写字母,以与变量名相区别。

(2) 'define 命令可以出现在模块定义里面,也可以出现在模块定义外面。宏名的有效范围为定义命令之后到源文件结束。通常,'define 命令写在模块定义的外面,作为程序的一部分,在此程序内有效。

(3) 在引用已定义的宏名时,必须在宏名的前面加上符号“'”,表示该名字是一个经过宏定义的名字。

(4) 使用宏名代替一个字符串,可以减少程序中重复书写某些字符串的工作量。而且,记住一个宏名要比记住一个无规律的字符串容易,这样在读程序时能立即知道它的含义。当需要改变某一个变量时,可以只改变 'define 命令行,一改全改。如例[1]中,先定义 WORDSIZE 代表常量 8,这时寄存器 data 是一个 8 位的寄存器。如果需要改变寄存器的大小,只需把该命令行改为:'define WORDSIZE 16。这样寄存器 data 则变为一个 16 位的寄存器。由此可见,使用宏定义可以提高程序的可移植性和可读性。

(5) 宏定义是用宏名代替一个字符串,也就是作简单的置换,不作语法检查。预处理时照样代入,不管含义是否正确。只有在编译已被宏展开后的源程序时才报错。

(6) 宏定义不是 Verilog HDL 语句,不必在行末加分号。如果加了分号会连分号一起进行置换。如:

例[2]
```
module  test;
    reg  a, b, c, d, e, out;
      'define  expression  a+b+c+d;
        assign  out = 'expression + e;
        ...
endmodule
```

经过宏展开以后,该语句为

```
assign  out= a+b+c+d;+e;
```

显然出现语法错误。

(7) 在进行宏定义时,可以引用已定义的宏名,并可以层层置换。如:

例[3]
```
module  test;
    reg  a, b, c;
    wire  out;
    'define  aa  a + b
    'define  cc  c + 'aa
        assign  out = 'cc;
endmodule
```

这样经过宏展开以后,assign 语句为

```
assign  out = c + a + b;
```

(8) 宏名和宏内容必须在同一行中进行声明。如果在宏内容中包含有注释行,注释行不会作为被置换的内容。如:

例[4]
```
module
    'define  typ_nand  nand #5 //define a nand with typical delay
      'typ_nand  g121(q21,n10,n11);
      ......
endmodule
```

经过宏展开以后,该语句为

```
nand #5  g121(q21,n10,n11);
```

宏内容可以是空格,在这种情况下,宏内容被定义为空的。当引用这个宏名时,不会有内容被置换。

注意:组成宏内容的字符串不能够被以下的语句记号分隔开。

- 注释行;
- 数字;
- 字符串;
- 确认符;
- 关键词;

• 双目和三目字符运算符。

如下面的宏定义声明和引用是非法的。

```
'define  first_half  "start of string
$display('first_half end of string");
```

在使用宏定义时要注意以下情况：

(1) 对于某些 EDA 软件，在编写源程序时，使用与预处理命令名相同的宏名会发生冲突，因此建议不要使用与预处理命令名相同的宏名。

(2) 宏名可以是普通的标识符(变量名)。例如 signal_name 和 'signal_name 的意义是不同的。因此，建议不要使宏名与程序内部的变量名相同。

2.9.2 "文件包含"处理 'include

所谓"文件包含"处理是一个源文件可以将另外一个源文件的全部内容包含进来，即将另外的文件包含到本文件之中。Verilog HDL 语言提供了 'include 命令来实现"文件包含"的操作。其一般形式为：

'include "文件名"

图 2.6 表示"文件包含"的含义。图 2.6(a)为文件 File1.v，它有一个 'include "File2.v"命令，并且还有其他的内容(以 A 表示)。图 2.6(b)为另一个文件 File2.v，文件的内容以 B 表示。在编译预处理时，要对 'include 命令进行"文件包含"预处理：将 File2.v 的全部内容复制插入到 'include "File2.v"命令出现的地方，即 File2.v 被包含到 File1.v 中，得到图 2.6(c)所示的结果。在接着往下进行的编译中，将"包含"以后的 File1.v 作为一个源文件单位进行编译。

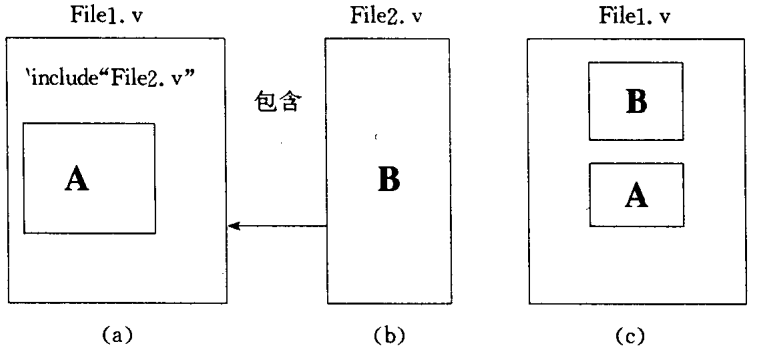


图 2.6 "文件包含"的含义

"文件包含"命令是很有用的，它可以避免程序设计人员的重复劳动，可以将一些常用的宏定义命令或任务(task)组成一个文件，然后用 'include 命令将这些宏定义包含到自己所写的源文件中，相当于将工业上的标准元件拿来使用。另外，在编写 Verilog HDL 源文件时，一个源文件可能经常要用到另外几个源文件中的模块，遇到这种情况即可用 'include 命令将所需模块的源文件包含进来。

例

1. 文件 aaa.v

```
module  aaa(a,b,out);
    input  a, b;
```

```
    output  out;
    wire  out;
        assign  out = a ^ b;
endmodule
```

2. 文件 bbb.v

```
'include  "aaa.v"
module  bbb(c,d,e,out);
    input  c,d,e;
    output  out;
    wire  out_a;
    wire  out;
        aaa(.a(c),.b(d),.out(out_a));
        assign  out=e&out_a;
endmodule
```

在上面的例子中,文件 bbb.v 用到了文件 aaa.v 中的模块 aaa 的实例器件,通过"文件包含"处理来调用。模块 aaa 实际上是作为模块 bbb 的子模块被调用的。在经过编译预处理后,文件 bbb.v 实际相当于下面的程序文件 bbb.v:

```
module  aaa(a,b,out);
    input  a, b;
    output  out;
    wire  out;
      assign  out = a ^ b;
endmodule

module  bbb( c, d, e, out);
    input  c, d, e;
    output  out;
    wire  out_a;
    wire  out;
        aaa(.a(c),.b(d),.out(out_a));
        assign  out= e & out_a;
endmodule
```

关于"文件包含"处理的 4 点说明:

(1) 一个 'include 命令只能指定一个被包含的文件,如果要包含 n 个文件,要用 n 个 'include 命令。注意下面的写法是非法的:'include "aaa.v" "bbb.v";

(2) 'include 命令可以出现在 Verilog HDL 源程序的任何地方,被包含文件名可以是相对路径名,也可以是绝对路径名。例如:'include "parts/count.v"

(3) 可以将多个 'include 命令写在一行,在 'include 命令行,只可以出空格和注释行。例如下面的写法是合法的。

```
'include "fileB" 'include "fileC" //including fileB and fileC
```

(4) 如果文件 1 包含文件 2,而文件 2 要用到文件 3 的内容,则可以在文件 1 用两个 'include 命令分别包含文件 2 和文件 3,而且文件 3 应出现在文件 2 之前。例如在下面的例子

中,即在 file1.v 中定义:

```
'include  "file3.v"
'include  "file2.v"

module  test(a,b,out);
input[1:'size2]  a, b;
output[1:'size2]  out;
wire[1:'size2]  out;
assign  out= a+b;
endmodule
```

file2.v 的内容为:

```
'define  size2  'size1+1
⋮
```

file3.v 的内容为:

```
'define  size14
⋮
```

这样,file1.v 和 file2.v 都可以用到 file3.v 的内容。在 file2.v 中不必再用 'include "file3.v"了。

(5) 在一个被包含文件中又可以包含另一个被包含文件,即文件包含是可以嵌套的。例如上面的问题也可以这样处理,如图 2.7 所示。

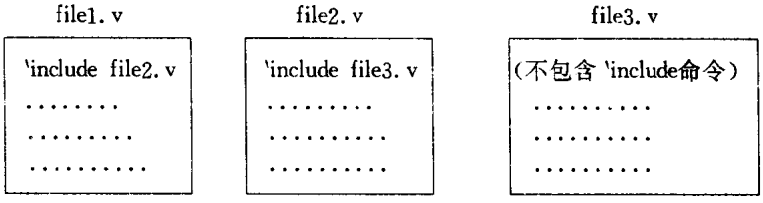


图 2.7

图 2.7 与图 2.8 的作用是相同的。

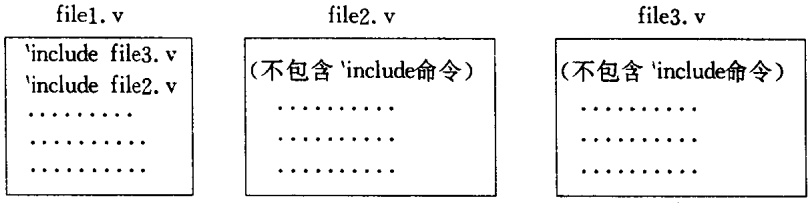


图 2.8

2.9.3 时间尺度 'timescale

'timescale 命令用来说明跟在该命令后的模块的时间单位和时间精度。使用 'timescale 命令可以在同一个设计里包含采用不同时间单位的模块。例如,一个设计中包含了两个模块,其中一个模块的时间延迟单位为 ns,另一个模块的时间延迟单位为 ps。EDA 工具仍然可以对这个设计进行仿真测试。

'timescale 命令的格式如下：

'timescale<时间单位>/<时间精度>

在这条命令中，时间单位参量是用来定义模块中仿真时间和延迟时间的基准单位的。时间精度参量是用来声明该模块仿真时间的精确程度的，该参量被用来对延迟时间值进行取整操作（仿真前），因此该参量又可以被称为取整精度。如果在同一个程序设计里，存在多个 'timescale 命令，则用最小的时间精度值来决定仿真的时间单位。另外，时间精度至少要和时间单位一样精确，时间精度值不能大于时间单位值。

在 'timescale 命令中，用于说明时间单位和时间精度参量值的数字必须是整数，其有效数字为 1，10，100，单位为 s，ms，μs，ns，ps，fs。这几种单位的意义见下表。

时间单位	定　义
s	秒(1s)
ms	千分之一秒(10^{-3}s)
μs	百万分之一秒(10^{-6}s)
ns	十亿分之一秒(10^{-9}s)
ps	万亿分之一秒(10^{-12}s)
fs	千万亿分之一秒(10^{-15}s)

下面举例说明 'timescale 命令的用法。

例[1] 'timescale 1ns/1ps

在这个命令之后，模块中所有的时间值都表示为 1ns 的整数倍。这是因为在 'timescale 命令中，定义了时间单位是 1ns。模块中的延迟时间可表达为带 3 位小数的实型数，因为 'timescale 命令定义时间精度为 1ps。

例[2] 'timescale 10μs/100ns

在这个例子中，'timescale 命令定义后，模块中时间值均为 10μs 的整数倍。因为 'timesacle 命令定义的时间单位是 10μs，延迟时间的最小分辨度为 0.1μs(100ns)，即延迟时间可表示为带一位小数的实型数。

例[3] 'timescale 10ns/1ns

```
module test;
reg set;
parameter d=1.55;
  initial
    begin
      #d set=0;
      #d set=1;
    end
endmodule
```

在这个例子中，'timescale 命令定义了模块 test 的时间单位为 10ns、时间精度为 1ns。因此在模块 test 中，所有的时间值应为 10ns 的整数倍，且以 1ns 为时间精度。这样经过取整操作，

存在参数 d 中的延迟时间实际是 16ns(即 1.6×10ns),这意味着在仿真时刻为 16ns 时寄存器 set 被赋值为 0,在仿真时刻为 32ns 时寄存器 set 被赋值为 1。仿真时刻值是按照以下的步骤来计算的:

(1) 根据时间精度,参数 d 值从 1.55 被取整为 1.6。

(2) 因为时间单位是 10ns,时间精度是 1ns,所以延迟时间 #d 为时间单位的整数倍,即 16ns。

(3) EDA 工具预定在仿真时刻为 16ns 的时候将寄存器 set 赋值为 0(即语句 #d set=0;执行时刻),在仿真时刻为 32ns 的时候将寄存器 set 赋值为 1(即语句 #d set=1;执行时刻)。

注意,如果在同一个设计里,多个模块中用到的时间单位不同,需要用到以下的时间结构:

(1) 用 'timescale 命令来声明本模块中所用到的时间单位和时间精度。

(2) 用系统任务 $printtimescale 来输出显示一个模块的时间单位和时间精度。

(3) 用系统函数 $time 和 $realtime 及%t 格式声明来输出显示 EDA 工具记录的时间信息。

2.9.4 条件编译命令 'ifdef,'else,'endif

一般情况下,Verilog HDL 源程序中所有的行都将参加编译。但是有时希望对其中的一部分内容只有在满足条件时才进行编译,也就是对一部分内容指定编译的条件,这就是"条件编译"。有时,希望当条件满足时对一组语句进行编译,而当条件不满足时则编译另一部分。

条件编译命令有以下几种形式:

(1) 'ifdef 宏名 (标识符)
    程序段 1
    'else
    程序段 2
    'endif

它的作用是当宏名已经被定义过(用 'define 命令定义),则对程序段 1 进行编译,程序段 2 将被忽略;否则编译程序段 2,程序段 1 被忽略。其中,'else 部分可以没有,即:

(2) 'ifdef 宏名 (标识符)
    程序段 1
    'endif

这里的"宏名"是一个 Verilog HDL 的标识符;"程序段"可以是 Verilog HDL 语句组,也可以是命令行。这些命令可以出现在源程序的任何地方。注意:被忽略掉不进行编译的程序段部分也要符合 Verilog HDL 程序的语法规则。

通常在 Verilog HDL 程序中用到 'ifdef,'else,'endif 编译命令的情况有以下几种:

(1) 选择一个模块的不同代表部分;

(2) 选择不同的时序或结构信息;

(3) 对不同的 EDA 工具,选择不同的激励。

2.10 小 结

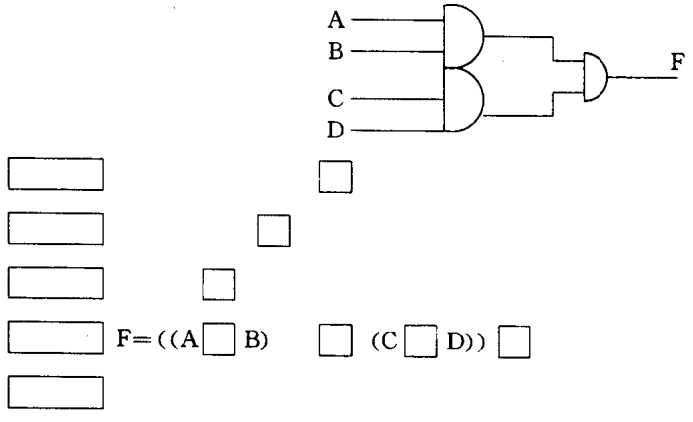
Verilog HDL 的语法与C语言的语法有许多类似的地方，但也有许多不同之处。我们学习 Verilog HDL 语法要善于找到不同点，着重理解如：阻塞(blocking)和非阻塞(non-blocking)赋值的不同；顺序块和并行块的不同；块与块之间的并行执行的概念；task 和 function 的概念。Verilog HDL 还有许多系统函数和任务也是C语言中没有的，如：$monitor，$readmemb，$stop 等等，而这些系统任务在调试模块的设计中是非常有用的，我们只有通过阅读大量的 Verilog 调试模块实例，经过长期的实践，经常查阅 Verilog 语法参考书才能逐步掌握。

思 考 题

我们将针对本章介绍的基本语法做一些练习。希望读者能在仔细阅读本章内容后，认真思考以下的习题，这将有效地帮助你正确地理解基本语法和要点。

1. 请将所给选项根据电路图填入程序中的适当位置。

```
assign  module; ~ | &  input output inputs outputs endmodule
A, B, C, D
AOI (A, B, C, D, F)
```

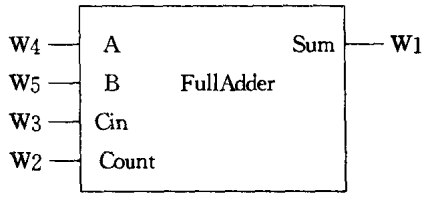


标准答案：

```
module  AOI(A,B,C,D,F);
input   A,B,C,D;
output  F;
assign  F = ((A&B)&(C&D));
endmodule
```

2. 假设已有全加器模块 FullAdder，若有一个顶层模块调用此全加器，连接线分别为 W4，W5，W3，W1 和 W2，请在调用时正确地填入 I/O 的对应信号。(本题为有关层次电路的练习，通过这个练习，将加深对模块间调用时管脚间连接的理解。)

```
module  FullAdder(A,B,Cin,Sum,Cout);
input  A, B, Cin;
output  Sum, Cout;

endmodule
module  Top…
  FullAdder  FA(
```




```
                [          ],//W1
                [          ],//W2
                [          ],//W3
                [          ],//W4
                [          ]);//W5
endmodule
```

标准答案：

```
module  Top...
FullAdder  FA(      .Sum(W1), //W1
                    .Cout(W2),//W2
                    .Cin(W3), //W3
                    .A(W4), //W4
                    .B(W5)); //W5
endmodule
```

3. 本题是一个测试模块，因此没有输入、输出端口，请将相应项填入合适的位置。

```
module  TestFixture;
  [          ]      MUX2  M(SEL,A,B,F)
  [          ]      reg  A,B, SEL;
                    wire  F;
initial
begin
  [          ]      $monitor(SEL,A,B,,F);
end
initial
  [          ]      SEL=0;A=0;B=0;
                    #10  A=1;
                    #10  SEL=1;#10  B=1;
endmodule
```

标准答案：

```
module  TestFixture
reg  A,B,SEL;
wire  F;
MUX2  M(SEL,A,B,F);
```

```
initial
begin
  SEL=0; A=0; B=0;
  #10  A=1;
  #10  SEL=1; #10  B=1;
end
initial
  $monitor(SEL,A,B,,F);
endmodule
```

4. 指出下面几个信号的最高位和最低位。

reg [1:0] SEL; input [0:2] IP; wire [16:23] A;

标准答案：

MSB：SEL[1] MSB：IP[0] MSB：A[16]

LSB：SEL[0] LSB：IP[2] LSB：A[23]

5. P,Q,R 都是 4 位的输入矢量，下面哪一种表达形式是正确的？

(1) input P[3:0],Q,R;

(2) input P,Q,R[3:0];

(3) input P[3:0],Q[3:0],R[3:0];

(4) input [3:0] P,[3:0]Q,[0:3]R;

(5) input [3:0] P,Q,R;

标准答案：(5)

6. 请将下面选项中的正确答案填入空的方括号中。

(1) (0:2) (2) (P:0) (3) (Op1:Op2) (4) (7:7) (5) (2:0)

(6) (7:0)

```
reg [7:0]  A;
reg [2:0]  Sum, Op1, Op2;
reg   P, OneBit;

initial
begin
Sum=Op1+Op2;
P=1;
A[  ]=Sum;
……
end
```

标准答案：(5)

7. 请根据以下两条语句，从选项中找出正确答案。

① reg [7:0] A;

A=2'hFF;

(1) 8'b0000_0011 (2) 8'h03 (3) 8'b1111_1111 (4) 8'b11111111

标准答案：(1),(2)

② reg [7：0] B；

B=8'bZ0；

(1) 8'0000_00Z0 (2) 8'bZZZZ_0000 (3) 8'b0000_ZZZ0 (4) 8'bZZZZ_ZZZ0

标准答案：(4)

8. 请指出下面几条语句中变量的类型。

① assign A=B；

② always #1

Count=C+1；

标准答案： A(wire) B(wire/reg) Count(reg) C(wire/reg)

9. 指出下面模块中 Cin,Cout,C3,C5 的类型。

```
module FADD(A,B,Cin,Sum,Cout);
input A, B, Cin;
output Sum, Cout;
....
endmodule
module Test;
...
FADD M(C1,C2,C3,C4,C5);
...
endmodule
```

标准答案： Cin(wire) Cout(wire/reg) C3(wire/reg) C5(wire)

10. 在本程序段中，当 ADDRESS 的值等于 5'b0X000 时，问 casex 执行完后 A 和 B 的值是多少？

```
A=0;
B=0;
casex(ADDRESS)
5'b00??? : A=1;
5'b01??? : B=1;
5'b10?00,5'b1 1?00 :
            begin
              A=1;
              B=1;
            end
endcase
```

标准答案：A=1 B=0

11. 事件 A 分别在 10,20,30 发生，而 B 一直保持 X 状态，问在 50 时 Count 的值是多少？

```
reg [7：0] Count;
initial
  Count=0;
always
  begin
```

```
@(A) Count=Count+1;
@(B) Count=Count+1;
end
```

标准答案:Count=1

(这是因为当 A 第一次发生时,Count 的值由 0 变为 1,然后事件控制 @(B) 阻挡了进程。)

12. 在下列程序中 initial 块执行完后,I,J,A,B 的值会是多少?

```
reg [2:0] A;
reg [3:0] B;
integer I, J;
  initial
    begin
      I=0;
      A=0;
      I=I-1;
      J=I;
      A=A-1;
      B=A;
      J=J+1;
      B=B+1;
    end
```

标准答案:

I=-1　　(整数可为负数)

J=0

A=7　　(A 为 reg 型的非负数,又因为 A 为 3 位,即为 111)

B=8　　(在 B=A 时,B=0111,然后 B=B+1,所以 B=4'b1000)

13. 在下列程序中,当 V 的值发生变化且为-1 时,执行完 always 块后 Count 的值应是多少?

```
reg[7:0] V;
reg[2:0] Count;

always @(V)
begin
Count=0;
while(~V[Count])
Count=Count+1;
end
```

标准答案:Count=0

14. 在下列程序中,循环执行完后 V 的值是多少?

```
reg [3:0] A;
reg  V ,W;
```

```
integer  K;
....
A=4'b1010;
for (K=2;K>=0;K=K-1)
   begin
      V=V^ A[K];
      W=A[K]^ A[K+1];
   end
```

标准答案：V 的值是它进入循环体前值的取反。

(因为 V 的值与 0,1,0 进行了异或，与 1 的异或改变了 V 的值。)

15. 下面给出了几种硬件实现，问以下的模块被综合后可能是哪一种？

```
always @(posedge Clock)
    if(A)
        C=B;
```

(1) 不能综合。

(2) 一个上升沿触发器和一个多路器。

(3) 一个输入是 A,B,Clock 的三输入与门。

(4) 一个透明锁存器。

(5) 一个带 clock 有始能引脚的上升沿触发器。

标准答案：(2),(5)

16. 在下列程序中，always 状态将描述一个带异步 Nreset 和 Nset 输入端的上升沿触发器，则空括号内应填入什么？可从以下 5 种答案中选择。

```
always @(          )
if(!Nreset)
Q<=0;
else if(!Nset)
Q<=1;
else
Q<=D;
```

(1) negedge Nset or posedge Clock

(2) posedge Clock

(3) negedge Nreset or posedge Clock

(4) negedge Nreset or negedge Nset or posedge Clock

(5) negedge Nreset or negedge Nset

标准答案：(4)

17. 下面给出了几种硬件实现，问以下的模块被综合后可能是哪一种？

(1) 带异步复位端的触发器。

(2) 不能综合或与预先设想的不一致。

(3) 组合逻辑。

(4) 带逻辑的透明锁存器。

(5) 带同步复位端的触发器。

① always @(posedge Clock)

```
begin
  A<=B;
  if(C)
    A<=1'b0;
end
```

标准答案:(5)

② always @(A or B)

```
case(A)
  1'b0 :  F=B;
  1'b1 :  G=B;
endcase
```

标准答案:(2)

③ always @(posedge A or posedge B)

```
if(A)
  C<=1'b0;
else
  C<=D;
```

标准答案:(1)

④ always @(posedge Clk or negedge Rst)

```
if(Rst)
  A<=1'b0;
else
  A<=B;
```

标准答案:(2)(产生了异步逻辑)

18. 在下列程序中,模块被综合后将产生几个触发器?

```
always @(posedge Clk)
begin :  Blk
  reg  B, C;
    C = B;
    D <= C;
    B = A;
end
```

(1) 2 个寄存器 B 和 D。

(2) 2 个寄存器 B 和 C。

(3) 3 个寄存器 B, C 和 D。

(4) 1 个寄存器 D。

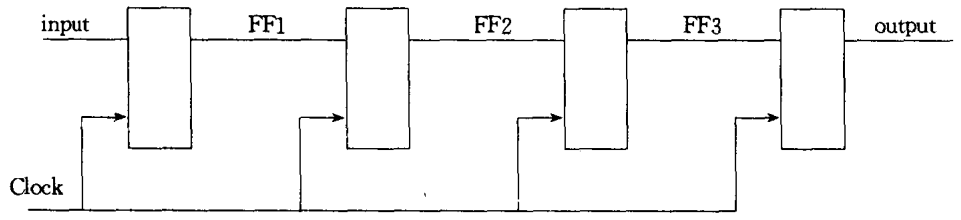
(5) 2 个寄存器 C 和 D。

标准答案:(2)

19. 在下列程序中,各条语句的顺序是错误的,请根据电路图调整好它们的次序。

```
Output=FF3;
```

```
reg  FF1, FF2, FF3;
          FF2=FF1;
always @ (posedge Clock)
end
     FF3=FF2;
begin
   FF1=Input;
```



标准答案：

```
reg  FF1,FF2,FF3;
always @(posedge Clock)
begin
   Output<=FF3;
   FF3=FF2;
   FF2=FF1;
   FF1=Input;
end
```

20. 根据左表中 SEL 与 OP 的对应关系，在右边模块的空括号中填入相应的值。

```
SEL :  OP
000 :  1
001 :  3          casex(SEL)
010 :  1          3'b(  ) :  OP=3;
011 :  3          3'b(  ) :  OP=1;
100 :  0          3'b(  ) :  OP=0;
101 :  3          endcase
110 :  0
111 :  3
```

标准答案：

```
casex(SEL)
3'bXX1 :  OP=3;
3'b0X0 :  OP=1;
3'b1X0 :  OP=0;
endcase
```

21. 在以下表达式中选出正确的。

(1) 4'b1010 & 4'b1101 = 1'b1

(2) ~ 4'b1100 = 1'b1

(3) !4'b1011 || !4'b0000 = 1'b1

(4) & 4'b1101 = 1'b1

(5) 1b'0 ||1b'1 = 1'b1

(6) 4'b1011 && 4'b0100 = 4'b1111

(7) 4'b0101<<1 =5'b01011

(8) !4'b0010 is 1'b0

(9) 4'b0001 || 4'b0000 = 1'b1

标准答案:(3), (5), (8), (9)

22. 在下列程序的括号中填入 display 的正确值。

```
integer  I;
reg[3 : 0]  A;
reg[7 : 0]  B;
initial
begin
I=-1;  A=I;  B=A;
 $display("%b",B);(     )
A=A/2;
 $display("%b",A);(     )
B=A+14;
 $diaplay("%d",B);(     )
A=A+14;
 $display("%d",A);(     )
A=-2;  I=A/2;
 $display("%d",I);(     )
end
```

标准答案:

```
...
I=-1;  A=I;  B=A;
 $display("%b",B);(00001111)
A=A/2;
 $display("%b",A);(0111)
B=A+14;
 $diaplay("%d",B);(21)
A=A+14;
 $display("%d",A);(5)  //A 为 4 位,所以 21 被截为 5
A=-2;  I=A/2;
 $display("%d",I);(7)  //A=-2,则是 1110
...
```

23. 请问{1,0}与下面哪一个值相等。

(1) 2'b01 (2) 2'b10 (3) 2'b00

(4) 64'H0000000000002 (5) 64'H0000000100000000

标准答案：(5)

(位拼接运算符必须指明位数，若不指明则隐含着为 32 位的二进制数(即整数)。)

24. 根据下面给出的程序，确定应将哪一个选项填入尖括号内。

(1) defs. Reset　　(2) "defs. v". Reset

(3) M. Reset　　(4) Reset

```
module defs;
    parameter Reset=8' b10100101;
endmodule                                    (file defs. v)
```

```
module   M;
.....
  if (OP==<     >)
  Bus=0;                                     (file M. v)
endmodule
```

① 标准答案：(1)

(模块间调用时，若引用其他模块定义的参数，要加上其他模块名，作为这个参数的前缀。)

```
module  M
'include "defs. v"
....
if(OP==<defs. Reset>)
Bus=0;
endmodule
```

② 标准答案：(4)

```
parameter  Reset=8'b10100101; (File defs. v)
module  M
'include "defs. v"
....
if(OP==<Reset>)
Bus=0;
endmodule
```

25. 如果调用 Pipe 时，想把 Depth 的值变为 8，问程序中的空括号内应填入何值？

```
Module  Pipe(IP,OP)
parameter  Option=1;
parameter  Depth=1;
...
endmodule
Pipe( )  P1(IP1,OP1);
```

标准答案：#(1,8)

(其中 1 对应参数 Option,8 对应参数 Depth。)

26. 若想使 P1 中的 Depth 的值变为 16,则应向空括号中填入哪个选项?

```
module  Pipe (IP ,OP);
    parameter  Option =1;
    parameter  Depth = 1;
    ..........
endmodule

module
   Pipe  P1(IP1 ,OP1);
   (              );
endmodule
```

(1) defparam P1. Depth=16;

(2) parameter P1. Depth=16;

(3) parameter Pipe. Depth=16;

(4) defparam Pipe. Depth=16;

标准答案:(1)

(用后缀改变引用模块的参数要用 defparam 及用本模块名作为引用参数的前缀,如 P1. Depth。)

27. 如果我们想在 Test 的 monitor 语句中观察 Count 的值,则在空括号中应填入什么?

```
Module  Test
Top  T();
initial
$monitor(    )
endmodule

module  Top;
Block  B1();
Block  B2();
endmodule

module  Block;
Counter  C();
endmodule

module  Counter;
reg [3 : 0]  Count;
....
endmodule
```

标准答案:T. B1. C. Counter Test. T. B1. C. Count

28. 下面方框中用 initial 块给 reg[7 : 0] V 赋值,请指明每种情况下 V 的 8 位都是什么

值。

该题说明在数的表示时，已标明字宽的数若用 XZ 表示某些位，只有在最左边的 X 或 Z 具有扩展性。

```
Reg [7 : 0]  V

initial
begin
      V=8'b0;
      V=8'b1;
      V=8'bX;
      V=8'BZX;
      V=8'BXXZZ;
      V=8'b1X;
end
```

标准答案：

8'b00000000
8'b00000001
8'bXXXXXXXX
8'bZZZZZZZX
8'BXXXXXXZZ
8'b0000001X

第三章　不同抽象级别的 Verilog HDL 模型

从第二章我们知道，Verilog 模型可以是实际电路的不同级别的抽象。这些抽象的级别和它们对应的模型类型共有以下 5 种：

(1) 系统级(system-level)；

(2) 算法级(algorithem-level)；

(3) RTL 级(Register Transfer Level)；

(4) 门级(gate-level)；

(5) 开关级(switch-level)。

本章将通过许多实际的 Verilog HDL 模块的设计来了解不同抽象级别模块的结构和可综合性的问题。对于数字系统的逻辑设计工程师而言，熟练地掌握门级、RTL 级、算法级、系统级是非常重要的。对于电路基本部件(如门、缓冲器、驱动器等)库的设计者而言，需要掌握开关级的描述。本书由于篇幅有限，略去了开关级的描述。

一个复杂电路的完整 Verilog HDL 模型是由若干个 Verilog HDL 模块构成的，每一个模块又可以由若干个子模块构成。这些模块可以分别用不同抽象级别的 Verilog HDL 描述，在一个模块中也可以有多种级别的描述。利用 Verilog HDL 语言结构所提供的这种功能就可以构造一个模块间的清晰层次结构来描述极其复杂的大型设计。

3.1　门级结构描述

一个逻辑网络由许多逻辑门和开关所组成，因此用逻辑门的模型来描述逻辑网络是最直观的。Verilog HDL 提供了一些门类型的关键字，可以用于门级结构建模。

3.1.1　与非门、或门和反向器等及其说明语法

Verilog HDL 中有关门类型的关键字共有 26 个，本书只介绍最基本的 8 个。有关其他的门类型关键字，读者可以通过翻阅 Verilog HDL 语言参考书，在设计的实践中逐步掌握。下面列出了 8 个基本的门类型(GATETYPE)关键字和它们所表示的门的类型。

and——与门；

nand——与非门；

nor——或非门；

or——或门；

xor——异或门；

xnor——异或非门；

buf——缓冲器；

not——非门。

门与开关的说明语法可以用标准的声明语句格式和一个简单的实例引用加以说明。

门声明语句的格式如下：

<门的类型>[<驱动能力><延时>]<门实例 1>[,<门实例 2>,……,<门实例 n>];

门的类型是门声明语句所必需的，它可以是 Verilog HDL 语法规定的 26 种门类型中的任意一种。驱动能力和延时是可选项，可根据不同的情况选不同的值或不选。门实例 1 是在本模块中引用的第一个这种类型的门，而门实例 n 是引用的第 n 个这种类型的门。有关驱动能力的选项我们在以后的章节里再详细介绍，下面用一个具体的例子来说明门类型的引用：

```
nand  #10  nd1(a,data,clock,clear);
```

这说明在模块中引用了一个名为 nd1 的与非门(nand)，输入为 data，clock 和 clear，输出为 a，输出与输入的延时为 10 个单位时间。

3.1.2 用门级结构描述 D 触发器

下面的例子是用 Verilog HDL 语言描述的 D 型主从触发器模块，通过这个例子，可以学习门级结构建模的基本技术。

```
module          flop(data,clock,clear,q,qb);
    input       data,clock,clear;
    output      q,qb;

    nand  #10   nd1(a,data,clock,clear),
                nd2(b,ndata,clock),
                nd4(d,c,b,clear),
                nd5(e,c,nclock),
                nd6(f,d,nclock),
                nd8(qb,q,f,clear);
    nand  #9    nd3(c,a,d),
                nd7(q,e,qb);
    not   #10   iv1(ndata,data),
                iv2(nclock,clock);

endmodule
```

在这个 Verilog HDL 结构描述的模块中，flop 定义了模块名，设计上层模块时可以用这个名(flop)调用这个模块；module，input，output，endmodule 等都是关键字；nand 表示与非门；#10 表示 10 个单位时间的延时；nd1，nd2，…，nd8，iv1，iv2 分别为图 3.1 中的各个基本部件，而其后面括号中的参数分别为图 3.1 中各基本部件的输入、输出信号。

3.1.3 由已经设计成的模块构成更高一层的模块

如果已经编制了一个模块，如 3.1.2 节中的 flop，我们可以在另外的模块中引用这个模块。引用的方法与门类型的实例引用非常类似，只需在前面写上已编的模块名，紧跟着写上引用的实例名，按顺序写上实例的端口名即可，也可以用已编模块的端口名按对应的原则逐一填入，见下面的两个例子。

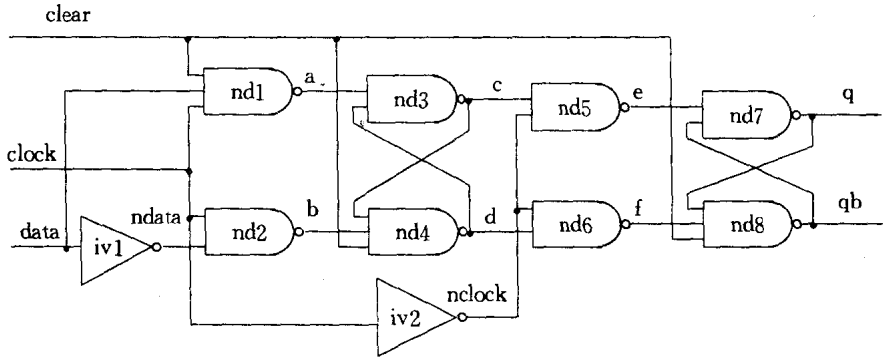


图 3.1　D 型主从触发器的电路结构图

例[1]　flop　flop_d(d1, clk, clrb, q, qn);

例[2]　flop　flop_d (.clock(clk),.q(q),.clear(clrb),.qb(qn),.data(d1));

这两个例子都表示实例 flop_d 引用已编模块 flop。从中可以看出，引用时 flop_d 的端口信号与 flop 的端口对应有两种不同的表示方法。模块的端口名可以按序排列也可以不必按序排列。如果模块的端口名按序排列，只需按序列出实例的端口名(见例[1])；如果模块的端口名不按序排列，则实例的端口信号和被引用模块的端口信号必须一一列出(见例[2])。

下面的例子中引用了 3.1.2 节中已设计的模块 flop，用它构成一个 4 位寄存器。

```
module          hardreg(d,clk,clrb,q);
input           clk,clrb;
input[3 : 0]    d;
output[3 : 0]   q;
flop    f1(d[0],clk,clrb,q[0],),
        f2(d[1],clk,clrb,q[1],),
        f3(d[2],clk,clrb,q[2],),
        f4(d[3],clk,clrb,q[3],);
endmodule
```

在上面这个结构描述的模块中，hardreg 定义了模块名；f1，f2，f3，f4 分别为图 3.2 中的各个基本部件，而其后面括号中的参数分别为图 3.2 中各基本部件的输入、输出信号。请注意，当 f1 到 f4 实例引用已编模块 flop 时，由于不需要 flop 端口中的 qb 口，故在引用时把它省去，但逗号仍需要留着。

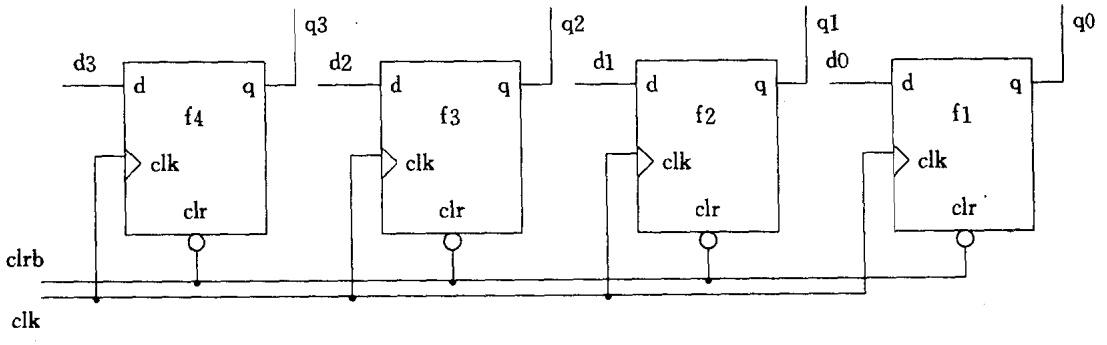


图 3.2　4 位寄存器电路结构图

显而易见，通过 Verilog HDL 模块的调用，可以构成任何复杂结构的电路。这种以结构方式所建立的硬件模型不仅是可以仿真的，也是可综合的，这就是以门级为基础的结构描述建模的基本思路。

3.2 Verilog HDL 的行为描述建模

3.2.1 仅用于产生仿真测试信号的 Verilog HDL 行为描述建模

为了对已设计的模块进行检验，往往需要产生一系列信号，输入到已设计的模块，并检查已设计模块的输出，看它们是否符合设计要求。这就要求我们编制测试模块，也称作测试文件，常用带.tf 扩展名的文件来描述测试模块。

下面的 Verilog HDL 行为描述模型用于产生时钟信号，以验证电路功能。其输出的仿真信号共有两个，分别是时钟 clk 和复位信号 reset。

初始状态时，clk 置为低电平，reset 为高电平。reset 信号输出一个复位信号之后，维持在高电平。这一功能可利用下面的语句来实现：

```
initial
  begin
    reset=1;              //初始状态
    clk=0;
    #3  reset=0;
    #5  reset=1;
end
```

以后每隔 5 个时间单位，时钟就翻转一次，这一功能可利用下面的语句来实现：

```
always  #5  clk = ~ clk;
```

从而该模块所产生的时钟的周期为 10 个时间单位。

完整的源程序如下：

```
module  gen_clk ( clk, reset);
            output  clk;
            output  reset;
            reg  clk, reset;

          initial
            begin
              reset = 1;              //initial state
              clk=0;
              #3  reset = 0;
              #5  reset = 1;
            end

          always  #5  clk = ~ clk;

endmodule
```

用这种方法所建立的模型主要用于产生仿真时测试下一级电路所需的信号，如下一级电路有输出反馈到上一级电路，并对上一级电路有影响时，也可以在这个模型中再加入输入信号，用于接收下一级电路的反馈信号。可以利用这个反馈信号再在这个模块中编制相应的输出信号，这样就比用简单的波形描述信号能更好地仿真实际电路。

再举一个简单的例子，即编制 3.1.3 节中完成的设计(即 hardreg 模块)的测试文件。这个测试文件不仅要包括时钟信号(clock)、数据(data[3：0])、清零信号(clearb)的变化，还须引用 4 位寄存器(hardreg)模块，以观测各种组合信号输入到该 4 位寄存器(hardreg)模块后，它的输出(q[3:0])的变化。这个测试文件完整的源程序如下：

```
module  hardreg_top;
    reg  clock, clearb;
    reg[3：0]  data;
    wire  qout;
    'define   stim   #100 data=4'b
      //宏定义 stim,可使源程序简洁
      event   end_first_pass;          //定义事件 end_first_pass
      hardreg reg_4bit (.d(data), .clk(clock), .clrb(clearb), .q(qout));
/*************************************************
把本模块中产生的测试信号 data,clock,clearb 输入实例 reg_4bit 以观察输出信号 qout。实例 reg_4bit 引用了 hardreg
*************************************************/
    initial
      begin
      clock = 0;
      clearb = 1;
      end
    always   #50  clk = ~ clk;
    always @(end_first_pass)
        clearb = ~ clearb;
    always @(posedge clock)
        $display("at time %0d clearb= %b data= %d qout= %d", $time, clearb, data, qout);
/*************************************************
类似与 C 语言的 printf 语句,可打印不同时刻的信号值
*************************************************/
    initial
    begin
      repeat(2)              //重复两次产生下面的 data 变化
      begin
          data=4'b0000;
          'stim0001;
/*************************************************
宏定义 stim 引用,等同于 #100 data=4'b0001;。注意引用时要用 ' 符号
*************************************************/
```

```
            'stim0010;
            'stim0011;
            'stim0100;
            'stim0101;
            :
            'stim1110;
            'stim1111;
          end
          #200 -> end_first_pass;
/*************************************************************
延迟 200 个单位时间,触发事件 end_first_pass
*************************************************************/
          $finish;              //结束 repeat(2)
        end
    endmodule
```

在上面的例子中,大家看到了一个前面未见过的语法现象:event。它是用来定义一个事件,以便在后面的操作中触发这一事件。它的触发方式是:

time (触发的时刻)->(事件名)

上面简单地介绍了利用 Verilog HDL 门级结构建模,来设计复杂数字电路的最基本思路。而实用的电路设计往往并没有那么简单,常需要利用多种方法来建立电路模型,既利用电路图输入的方法又利用 Verilog HDL 各种建模的方法,发挥各自在不同类型电路描述中的长处。而且,要在层次管理工具的协调下把各个既独立又互相联系的模块组织成复杂的大型数字电路,只有这样才能有效地设计出高质量的数字电路。

3.2.2 Verilog HDL 建模在 TOP-DOWN 设计中的作用和行为建模的可综合性问题

Verilog HDL 行为描述建模不仅可用于产生仿真测试信号对已设计的模块进行检测,也常常用于复杂数字逻辑系统的顶层设计,也就是通过行为建模把一个复杂的系统分解成可操作的若干个模块,每个模块之间的逻辑关系通过行为模块的仿真加以验证。虽然这些子系统在设计的这一阶段还不都是电路逻辑,也未必能用综合器把它们直接转换成电路逻辑,但还是能把一个大的系统合理地分解为若干个较小的子系统,然后,每个子系统再用可综合风格的 Verilog HDL 模块(门级结构或 RTL 级、算法级、系统级的模块)或电路图输入的模块加以描述。当然,这种描述可以分很多个层次来进行,但最终的目的是要设计出具体的电路来。所以,在任何系统的设计过程中接近底层的模块往往都是门级结构或 RTL 级的 Verilog HDL 模块,或电路图输入的模块。

由于 Verilog HDL 高级行为描述用于综合的历史还只有短短的几年,可综合风格的 VHDL 和 Verilog HDL 的语法只是它们自己语言的一个子集。又由于 HDL 的可综合性研究近年来发展很快,可综合子集的国际标准目前尚未形成,因此各厂商的综合器所支持的可综合 HDL 子集也略有所不同。本书有关可综合风格的 Verilog HDL 的内容,只着重介绍门级逻辑结构、RTL 级和部分算法级的描述,而系统级(数据流级)的综合由于还不太成熟,暂不作介绍。

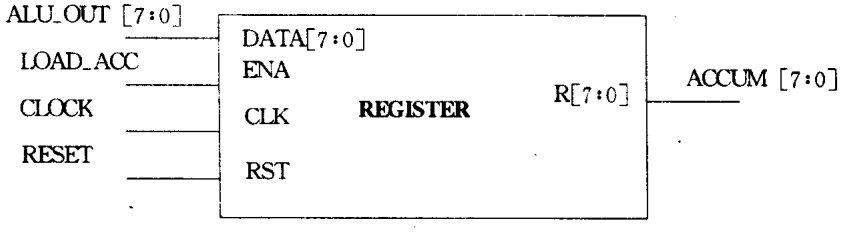
所谓综合就其实质而言是设计流程中的一个阶段，在这一阶段中综合器把 HDL 程序转换成标准的门级结构网表，而并非真实具体的门级电路。真实具体的电路还需要利用 ASIC 和 FPGA 制造厂商的布局布线工具，根据综合后生成的标准的门级结构网表来产生。为了能转换成标准的门级结构网表，HDL 程序的编写必须符合特定综合器所要求的风格。由于门级结构、RTL 级的 HDL 程序的综合是很成熟的技术，所有的综合器都支持这两个级别 HDL 程序的综合，因而是本书综合方面介绍的重点。

3.3　用 Verilog HDL 建模进行 TOP-DOWN 设计的实例

下面是一个用 Verilog HDL 建模来设计一个用于教学的、经简化只有 8 条指令的、字长为一字节的 RISC 中央处理单元(CPU)的顶层设计。

大家知道 RISC _ CPU 是一个极其复杂的数字逻辑电路，但是它基本部件的逻辑并不复杂，可把它分割成 9 个基本部件：累加器(ACCUMULATOR)、RISC 算术运算单元(RISC _ ALU)、数据控制器(DATACTRL)、动态存储器(RAM)、指令寄存器(INSTRUCTION REGISTER)、状态控制器(STATE CONTROLLER)、程序计数器(PROGRAMM COUNTER)、地址多路器(ADDRMUX)和时钟发生器(CLKGEN)。用 Verilog HDL 把各基本部件的功能描述清楚，并把每个部件的输入、输出之间的逻辑关系通过仿真加以验证，并不是很困难的一件事。然后，用结构建模的方法把它们组成一个顶层模块，也就是 RISC _ CPU 的 Verilog HDL 整体模型，经仿真验证各部件之间的逻辑关系后，再逐块用可综合风格的 Verilog HDL 语法检查并改写为可综合的 Verilog HDL，或用电路图描述把它们设计出来。经综合、优化、布局、布线后再做后仿真。如果仿真结果正确，电路就设计完毕。下面就是这些基本部件的 Verilog HDL 模块。

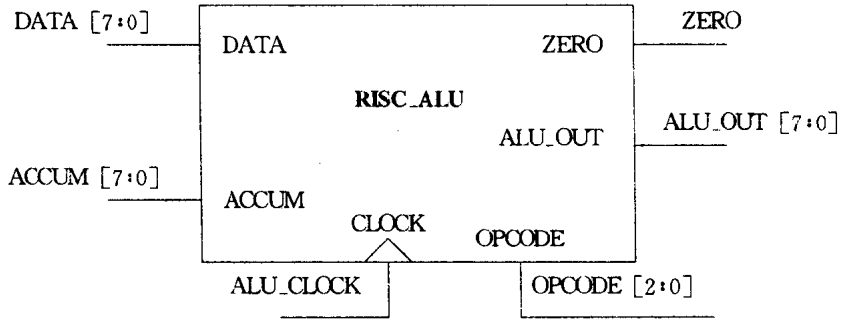
1. 累加器用寄存器(ACCUMULATOR REGISTER)




```
'timescale  1ns/1ns
module  register(r,clk,data,ena,rst);
output [7:0]  r;
input [7:0]  data;
input  clk, ena, rst;
wire  load;

  and  a1(load,clk,ena);
  DFF  d7(r[7],,load,data[7],rst);
  DFF  d6(r[6],,load,data[6],rst);
  DFF  d5(r[5],,load,data[5],rst);
```

```
    DFF   d4(r[4],,load,data[4],rst);
    DFF   d3(r[3],,load,data[3],rst);
    DFF   d2(r[2],,load,data[2],rst);
    DFF   d1(r[1],,load,data[1],rst);
    DFF   d0(r[0],,load,data[0],rst);
endmodule
```

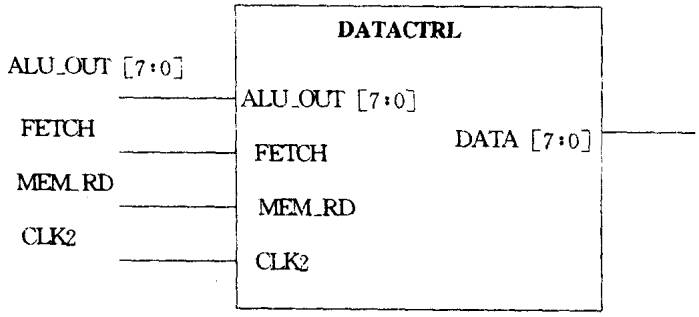
2. RISC 算术运算单元(RISC _ ALU)




```
'timescale  1ns/100ps
module  riscalu ( alu _out, zero, opcode, data, accum, clock );
output [7 : 0]  alu _out;
reg[7 : 0]  alu _out;
output  zero;
input [2 : 0]  opcode;
input [7 : 0]  data, accum;
input  clock;

'define  Zdly  1.2
'define  ALUdly  3.5

wire  #'Zdly  zero=(!accum);
//* * * 即 zero=1'b1 if accum==0,else zero=1'b0

always @(negedge clock)
begin
   case (opcode)
       3'b000 :   #'ALUdly  alu _out=accum;          //Pass Accumulator
       3'b001 :   #'ALUdly  alu _out=accum;          //Pass Accumulator
       3'b010 :   #'ALUdly  alu _out=data+accum;     //ADD
       3'b011 :   #'ALUdly  alu _out=data&accum;     //AND
       3'b100 :   #'ALUdly  alu _out=data^ accum;    //XOR
       3'b101 :   #'ALUdly  alu _out=data;           //Pass Data
       3'b110 :   #'ALUdly  alu _out=accum;          //Pass Accumulator
       3'b111 :   #'ALUdly  alu _out=accum;          //Pass Accumulator
```

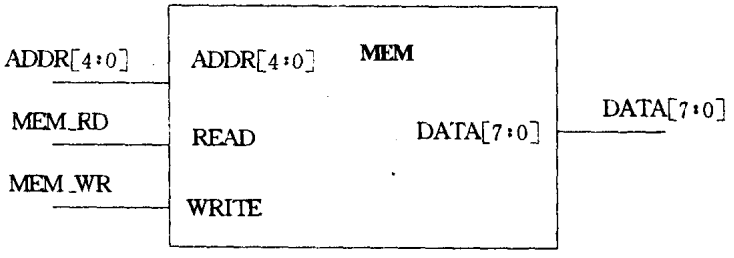
```
        default :   begin
        $display("Unknown OPcode");
        #1ALUdly   alu_out=8'bXXXXXXXX;
        end
    endcase
    end
    endmodule
```

3. 数据控制器(DATACTRL)




```
module  datactrl(data,alu_out,fetch,mem_rd,clk2);
    output [7:0]  data;
    input [7:0]  alu_out;
    input   fetch, mem_rd, clk2;

        assign  data=(( !fetch & !mem_rd & !clk2 )? alu_out : 8'bz);
endmodule
```

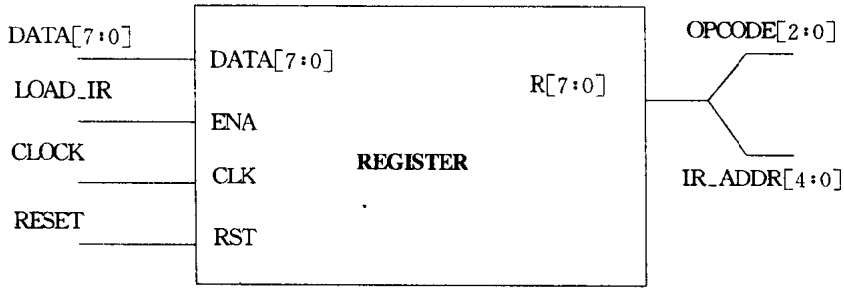
4. 动态存储器(RAM)




```
'timescale  1ns/1ns
module mem(data,addr,read,write);
    inout [7 : 0]  data;
    input [4 : 0]  addr;
    input   read, write;
    reg [7 : 0]  memory[0 : 'h1F];
    wire[7 : 0]  data =( read? memory[addr] : 8'bZZZZZZZZ );

    always @(posedge write)
      begin
```

```
        memory[addr]=data;
    end

endmodule
```

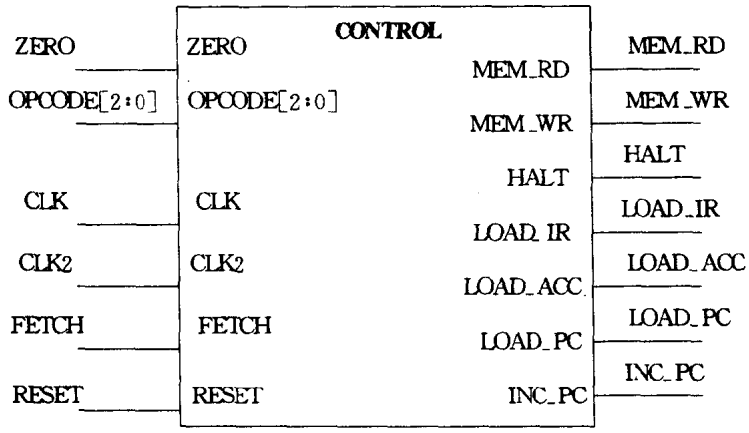
5. 指令寄存器(INSTRUCTION REGISTER)




```
module  register(r,clk,data,ena,rst);
  output [7 : 0]  r;
  input [7 : 0]  data;
  input  clk, ena, rst;
  wire  load;

  and  a1(load,clk,ena);
  DFF  d7(r[7],,load,data[7],rst);
  DFF  d6(r[6],,load,data[6],rst);
  DFF  d5(r[5],,load,data[5],rst);
  DFF  d4(r[4],,load,data[4],rst);
  DFF  d3(r[3],,load,data[3],rst);
  DFF  d2(r[2],,load,data[2],rst);
  DFF  d1(r[1],,load,data[1],rst);
  DFF  d0(r[0],,load,data[0],rst);

endmodule
```

6. 状态控制器(STATE CONTROLLER)

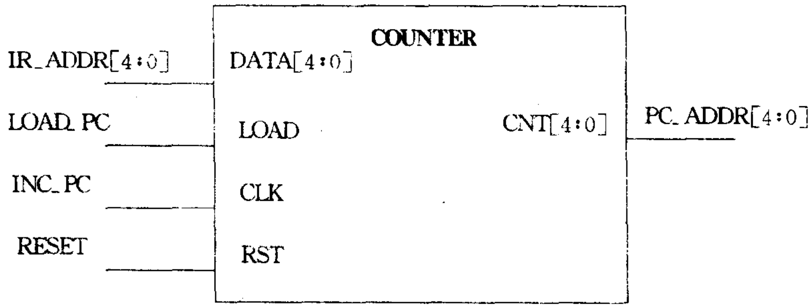
```
'timescale  1ns/1ns
module  control (load_acc, mem_rd, mem_wr, inc_pc, load_pc, load_ir,
                 halt, opcode, fetch, zero, clk, clk2, reset);
output  load_acc, mem_rd, mem_wr, inc_pc, load_pc, load_ir, halt;
reg load_acc, mem_rd, mem_wr, inc_pc, load_pc, load_ir, halt;
input [2 : 0]  opcode;
input  fetch, zero, clk, clk2, reset;

'define  HLT  3'b000
'define  SKZ  3'b001
'define  ADD  3'b010
```




```
'define  AND  3'b011
'define  XOR  3'b100
'define  LDA  3'b101
'define  STO  3'b110
'define  JMP  3'b111

always @(posedge fetch)
        if(reset)
            ctl_cycle;

always @(negedge reset)
begin
   disable ctl_cycle;
   {inc_pc,load_acc,load_pc,mem_wr,mem_rd,load_ir,halt}=7'b0000000;
end

always @(posedge reset)
    @(posedge fetch) ctl_cycle;
task  ctl_cycle;
begin
      //state 0——first Address Setup
      {inc_pc,load_acc,load_pc,mem_wr,mem_rd,load_ir,halt}=7'b0000000;
      //state1——Instruction Fetch
      @(posedge clk)
      {inc_pc,load_acc,load_pc,mem_wr,mem_rd,load_ir,halt}=7'b0000100;
      //state2——InstructionLoad
      @(negedge clk)
      {inc_pc,load_acc,load_pc,mem_wr,mem_rd,load_ir,halt}=7'b0000110;
      //state3——Idle
      @(posedge clk)
      {inc_pc,load_acc,load_pc,mem_wr,mem_rd,load_ir,halt}=7'b0000110;
```

```
        //state4——Second Address Setup
        @(negedge clk)
        if(opcode==='HLT)
        {inc_pc,load_acc,load_pc,mem_wr,mem_rd,load_ir,halt}=7'b1000001;
        else
        {inc_pc,load_acc,load_pc,mem_wr,mem_rd,load_ir,halt}=7'b1000000;
        //state5——Operand Fetch
        @(posedge clk)
        if((opcode==='ADD)||(opcode==='AND)||(opcode==='XOR)||(opcode==='LDA))
        {inc_pc,load_acc,load_pc,mem_wr,mem_rd,load_ir,halt}=7'b0000100;
        else
        {inc_pc,load_acc,load_pc,mem_wr,mem_rd,load_ir,halt}=7'b0000000;
        //state6——ALU operation
        @(negedge clk)
        if(opcode==='JMP)
        {inc_pc,load_acc,load_pc,mem_wr,mem_rd,load_ir,halt}=7'b0010000;
        else if((opcode==='SKZ)&&(zero))
        {inc_pc,load_acc,load_pc,mem_wr,mem_rd,load_ir,halt}=7'b1000000;
        else if ((opcode==='ADD)||(opcode==='AND)||(opcode==='XOR)||(opcode ==
        'LDA))
        {inc_pc,load_acc,load_pc,mem_wr,mem_rd,load_ir,halt}=7'b0100100;
        else
        {inc_pc,load_acc,load_pc,mem_wr,mem_rd,load_ir,halt}=7'b0000000;
        //state7——Store Result
        @(posedge clk)
        if(opcode ==='JMP)
        {inc_pc,load_acc,load_pc,mem_wr,mem_rd,load_ir,halt}=7'b1010000;
        else if(opcode==='STO)
        {inc_pc,load_acc,load_pc,mem_wr,mem_rd,load_ir,halt}=7'b0001000;
        else if((opcode==='SKZ)&&(zero))
        {inc_pc,load_acc,load_pc,mem_wr,mem_rd,load_ir,halt}=7'b1000000;
        else if ((opcode==='ADD)||(opcode==='AND)||(opcode==='XOR)||(opcode==
        'LDA))
        {inc_pc,load_acc,load_pc,mem_wr,mem_rd,load_ir,halt}=7'b0100100;
        else
        {inc_pc,load_acc,load_pc,mem_wr,mem_rd,load_ir,halt}=7'b0000000;
    end         //task ctl_cycle
    endtask

    endmodule
```

7. 程序计数器(PROGRAMM COUNTER)

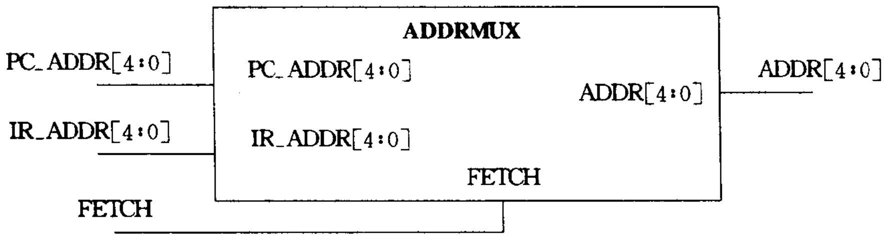



```
/*****************************************************
*     Behavior of a 5-bit counter
*****************************************************/
 `timescale  1ns/1ns
 module  counter (cnt, clk, data, rst, load);
         output [4 : 0]  cnt;
         input [4 : 0]  data;
         input  clk, rst, load;
         reg [4 : 0]  cnt;

         //asynchronous reset
         always @(rst)
            begin
               if(rst==0)
               cnt<=5'h00;
               wait(rst!=0);
            end

         always @(posedge clk)
            if (load==1)          //load counter
                cnt <= data;
            else           //(load!=1) therefore increment
                if(cnt==5'h1F)     //counter roll over
                  begin
                  cnt<=5'h00;
                  end
                else
                  cnt<=cnt+1;
 endmodule
```

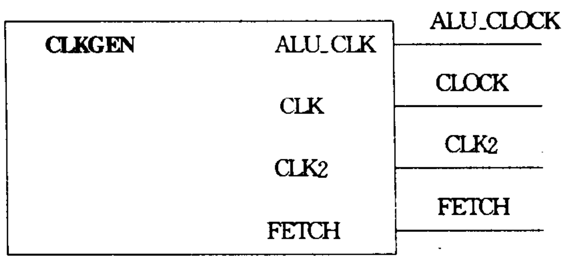
8. 地址多路器(ADDRMUX)

```
module  addrmux(addr, pc_addr, ir_addr, fetch);
        output [4 : 0]  addr;
        input [4 : 0]  pc_addr, ir_addr;
```

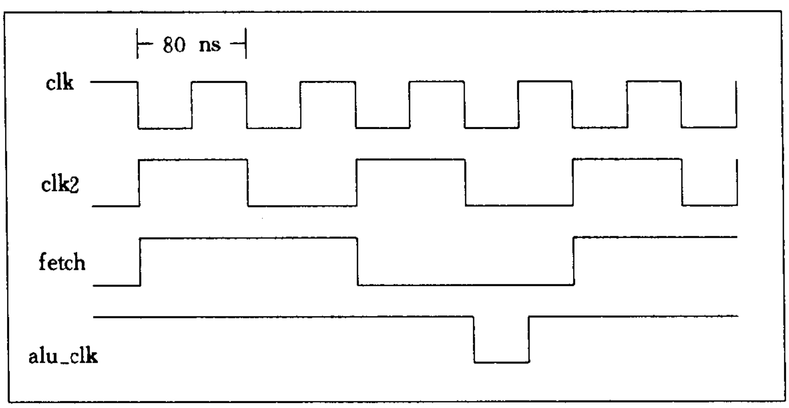
```
        input  fetch;
            assign  addr = ( fetch? pc _addr : ir _addr );
endmodule
```



9. 时钟发生器(CLKGEN)




```
/*************************************************
*** A free-running multi phase clock ocillator for the
*** Verification of Risc CPU system.
*** This module generates 4 clocks with the following
*** specification:
*************************************************/
```




```
  `timescale  1ns/1ns
module  clkgen(fetch,clk2,clk,alu _clk);
      output  fetch,clk2,clk,alu _clk;
      reg fetch,clk2,clk;
      `define  period  80
      assign  alu _clk = ( fetch | clk2 | clk );
      initial
         fork
```

```
        clk=0;
        clk2=1;
        fetch=1;
        forever  #('period/2)  clk =~clk;
        forever  #('period)  clk2 =~clk2;
        forever  #('period * 2)  fetch =~fetch;
        join
    endmodule
```

最后,用一个顶层模块把这些基本部件联系起来。下面所列出的就是这个顶层模块的 Verilog HDL 的结构描述:

```
'timescale  1ns/100ps
module  risc_top;
wire  reset, load_acc, load_ir, load_pc, halt, zero;
wire  clock, clk2, alu_clock, fetch, inc_pc;
wire [7:0]  alu_out, accum, data, opcode_iraddr;
wire [4:0]  addr, ir_addr, pc_addr;
wire [2:0]  opcode;

assign  {opcode,ir_addr} = opcode_iraddr;

register  accumulator(.r(accum), .clk(clock), .data(alu_out),
                      .ena(load_acc),.rst(reset));
riscalu  risc_alu(.alu_out(alu_out), .zero(zero),
                  .opcode(opcode),.data(data),.accum(accum),.clock(clock) );
datactrl  data_control(.data(data),.alu_out(alu_out),
                       .fetch(fetch),.mem_rd(mem_rd),.clk2(clk2) );

mem  ram_mem(.data(data),.addr(addr),.read(mem_rd),.write(mem_wr));

register  instr_register(.r(opcode_iraddr),.clk(clock),
                         .data(data),.ena(load_ir),.rst(reset) );
control  state_controler(.load_acc(load_acc),.mem_rd(mem_rd),.mem_wr(mem_wr),
                         .inc_pc(inc_pc), .load_pc(load_pc),.load_ir(load_ir),
                         .halt(halt), .opcode(opcode),.fetch(fetch),
                         .zero(zero), .clk(clock), .clk2(clk2),.reset(reset) );

counter  program_counter(.cnt(pc_addr),.clk(inc_pc),.data(ir_addr),
                         .rst(reset),.load(load_pc) );

addrmux  addr_mux( .addr(addr), .pc_addr(pc_addr),.ir_addr(ir_addr),.fetch(fetch) );

clkgen  clock_risc( .fetch(fetch),.clk2(clk2),.clk(clock),.alu_clk(alu_clock) );
```

```
endmodule
```

这个例子是可以仿真的，但并非其所有的模块都是可综合的。若需要综合成逻辑网表(EDIF)，有许多模块还需按可综合风格改写；若需制成具体的电路芯片，还需要在综合成标准逻辑网表后，编制具体实现工艺的约束文件，再利用厂家的布局布线工具生成电路制造文件，然后从中提取后仿真模型，再做后仿真。若在后仿真中发现问题则需降低时钟频率，或改写部分模块并重复前面的过程，直至后仿真中发现的问题都解决为止。若只做前仿真，只需输入表示程序指令的数据文件到 RAM 中就可按时钟节拍观测到指令的执行过程和波形。

通过这个例子我们想要说明的是：非常复杂的数字电路可以借助于 Verilog HDL 建模、仿真的方法分解成较为简单的模块，经验证后，再逐块地用电路图输入或用可综合风格的 Verilog HDL 加以实现。这就是我们理解的 TOP-DOWN 设计的概念。

我们将在第五章中较详细地介绍怎样逐块地把这一模型改造为可综合风格的 Verilog HDL 模型，并把寻址空间扩大到 8KB。

3.4 小 结

本章介绍了不同抽象级别的 Verilog HDL 模块。一个复杂数字系统的设计往往是由若干个模块构成的，每一个模块又可以由若干个子模块构成。这些模块可以是电路图描述的模块，也可以是 Verilog HDL 描述的模块。各 Verilog HDL 模块可以是不同级别的描述，同一个 Verilog HDL 模块中也可以有不同级别的描述。利用 Verilog HDL 语言结构所提供的这种功能不仅可以用来描述，也可以用来验证极其复杂的大型数字系统的总体设计，把一个大型设计分解成若干个可以操作的模块，分别用不同的方法加以实现。目前，用门级和 RTL 级抽象描述的 Verilog HDL 模块可以用综合器转换成标准的逻辑网表；用算法级描述的 Verilog HDL 模块，只有部分综合器能把它转换成标准的逻辑网表；而用系统级描述的模块，目前尚未有综合器能把它转换成标准的逻辑网表，往往只用于系统仿真。

思 考 题

1. Verilog HDL 的模型共有哪几种类型(级别)？
2. 每种类型的 Verilog HDL 各有什么特点？主要用于什么场合？
3. 为什么说 Verilog HDL 的语言结构可以支持构成任意复杂的数字逻辑系统？
4. 什么是综合？是否任何符合语法的 Verilog HDL 程序都可以综合？
5. 综合后生成的是不是真实的电路？若不是，还需要哪些步骤才能真正变为具体的电路？
6. 什么是 TOP-DOWN 设计方法？通过什么手段来验证系统分块的合理性？

第四章　有限状态机和可综合风格的 Verilog HDL

由于 VHDL 和 Verilog HDL 这两种行为描述用于综合的历史还只有短短的几年，可综合风格的 VHDL 和 Verilog HDL 的语法只是它们各自语言的一个子集。又由于 HDL 的可综合性研究近年来非常活跃，但可综合子集的国际标准目前尚未最后形成，因此各厂商的综合器所支持的 HDL 子集也略有所不同。本书有关可综合风格的 Verilog HDL 的内容，只着重介绍 RTL 级、算法级和门级逻辑结构的描述，而系统级（数据流级）的综合由于还不太成熟，暂不作介绍。由于寄存器传输级（RTL）描述是以时序逻辑抽象所得到的有限状态机为依据的，所以，把一个时序逻辑抽象成一个同步有限状态机是设计可综合风格的 Verilog HDL 模块的关键。本章将通过实例由浅入深地介绍各种可综合风格的 Verilog HDL 模块，并把重点放在时序逻辑的可综合有限状态机的 Verilog HDL 设计要点上。至于组合逻辑，因为比较简单，只需阅读典型的、用 Verilog HDL 描述的、可综合的组合逻辑的例子就可以掌握。

4.1　有限状态机

有限状态机是由寄存器组和组合逻辑构成的硬件时序电路，其状态（即由寄存器组的 1 和 0 的组合状态所构成的有限个状态）只能在同一时钟跳变的情况下才能从一个状态转向另一个状态，究竟转向哪一状态不但取决于各个输入值，还取决于当前状态。（这里指的是 Mealy 型有限状态机，而 Moore 型有限状态机究竟转向哪一状态只取决于当前状态。）

在 Verilog HDL 中可以用许多种方法来描述有限状态机，最常用的是利用 always 语句和 case 语句。图 4.1 表示了一个有限状态机，例[1]～例[4]的程序就是该有限状态机的 4 种 Verilog HDL 模型。

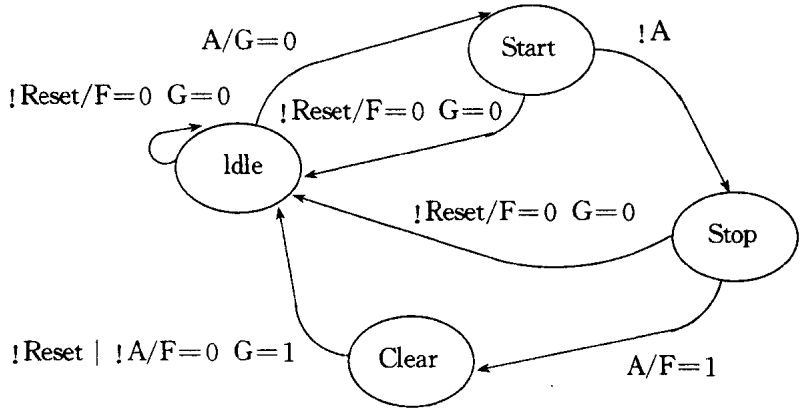


图 4.1　状态转移图

图 4.1 所示的状态转移图表示了一个 4 状态的有限状态机，它的同步时钟是 Clock，输入信号是 A 和 Reset，输出信号是 F 和 G。状态的转移只能在同步时钟（Clock）的上升沿处发生，

往哪个状态转移则取决于目前所在的状态和输入的信号(Reset 和 A)。下面的例子是该有限状态机的 Verilog HDL 模型之一。

例[1]

```
module  fsm (Clock, Reset, A, F, G);
  input  Clock, Reset, A;
  output  F,G;
  reg  F,G;
  reg [1 : 0]  state ;

parameter  Idle = 2'b00, Start = 2'b01,
           Stop = 2'b10, Clear = 2'b11;

  always @(posedge Clock)
   if (!Reset)
      begin
          state <= Idle; F<=0; G<=0;
      end
   else
     case (state)
       idle :   begin
                if (A)  state <= Start;
                G<=0;
                end
       start :  if (!A)  state <= Stop;
       Stop :   begin
                if (A)  state <= Clear;
                F <= 1;
                end
       Clear :  begin
                if (!A)  state <=Idle;
                F<=0; G<=1;
                end
        endcase
endmodule
```

我们还可以用另一个 Verilog HDL 模型来表示同一个有限状态机,见下例。

例[2]

```
module  fsm (Clock, Reset, A, F, G);
  input  Clock, Reset, A;
  output  F,G;
  reg  F,G;
  reg [3 : 0]  state ;

parameter  Idle     = 4'b1000,
           Start    = 4'b0100,
```

```
        Stop    = 4'b0010,
        Clear   = 4'b0001;

always @(posedge clock)
  if (!Reset)
    begin
      state <= Idle; F<=0; G<=0;
    end
  else
    case (state)
      Idle :  begin
                if (A)  state <= Start;
                G<=0;
              end
      Start :  if (!A)  state <= Stop;
      Stop :  begin
                if (A)  state <= Clear;
                F <= 1;
              end
      Clear: begin
               if (!A)  state <=Idle;
               F<=0; G<=1;
             end
      default :  state <=Idle;
    endcase
endmodule
```

例[1]与例[2]的主要不同点是状态编码，例[2]采用了独热编码(one-hot-coding)，而例[1]则采用 Gray 码，究竟采用哪一种编码好要依具体情况而定。对于用 FPGA 实现的有限状态机建议采用独热码，因为虽然采用独热编码多用了两个触发器，但可省下许多组合电路，因而可使电路的速度和可靠性有显著提高，而总的单元数却增加不多。采用独热编码后会出现多余的状态，即一些不可到达的状态，为此，在 case 语句的最后需要增加 default 分支项，以确保多余状态能回到 Idle 状态。

例[3]是用另一种风格的 Verilog HDL 模型来表示同一个有限状态机。在这个模型中，用 always 语句和连续赋值语句把状态机的触发器部分和组合逻辑部分分成两个部分来描述。

例[3]

```
module  fsm (Clock, Reset, A, F, G);
input  Clock, Reset, A;
output  F,G;
reg [1 : 0]  state ;
wire [1 : 0]  Nextstate;

parameter  Idle = 2'b00, Start = 2'b01,
           Stop = 2'b10, Clear = 2'b11;
```

```
always @(posedge Clock)
   if (!Reset)
      begin
         state <= Idle;
      end
   else
     state <= Nextstate;

assign Nextstate=(state==Idle)?   (A? Start : Idle) :
                 (state==Start)?  (!A? Stop : Start) :
                 (state== Stop)?  (A? Clear : Stop) :
                 (state== Clear)? (!A? Idle : Clear) : Idle;

assign  F = (state == Clear);
assign  G = (state == Idle);

endmodule
```

例[4]是用另一种风格的 Verilog HDL 模型来表示同一个有限状态机。在这个模型中,分别用沿触发的 always 语句和电平敏感的 always 语句把状态机的触发器部分和组合逻辑部分分成两个部分来描述。

例[4]

```
module  fsm (Clock, Reset, A, F, G);
  input  Clock, Reset, A;
  output  F,G;
  reg [1 : 0]  state, Nextstate;

parameter  Idle = 2'b00, Start = 2'b01,
           Stop = 2'b10, Clear = 2'b11;

  always @(posedge Clock)
    if (!Reset)
       begin
          state <= Idle;
       end
    else
      state <= Nextstate;

  always @( state or A )
    begin
       F=0;
       G=0;
       if (state == Idle)
       begin
```

```
        if (A)
          Nextstate = Start;
        else
          Nextstate = Idle;
        G=1;
      end
      else
        if (state == Start)
          if (!A)
            Nextstate = Stop;
          else
            Nextstate = Start;

        else
          if (state == Stop)
            if (A)
              Nextstate = Clear;
            else
              Nextstate = Stop;

          else
            if (state == Clear)
              begin
                if (!A)
                  Nextstate = Idle;
                else
                  Nextstate = Clear;
                F=1;
              end
            else
                  Nextstate= Idle;
    end
endmodule
```

上面 4 个例子是同一个状态机的 4 种不同的 Verilog HDL 模型，它们都是可综合的，在设计复杂程度不同的状态机时有它们各自的优势。如用不同的综合器对这 4 个例子进行综合，综合出的逻辑电路可能会有些不同，但逻辑功能是相同的。下面总结了有限状态机设计的一般步骤，供大家参考。

1. 逻辑抽象，得出状态转换图

就是把给出的一个实际逻辑关系表示为时序逻辑函数。可以用状态转换表来描述，也可以用状态转换图来描述。这就需要：

(1) 分析给定的逻辑问题，确定输入变量、输出变量以及电路的状态数。通常是取原因(或条件)作为输入变量，取结果作为输出变量。

(2) 定义输入、输出逻辑状态的含义,并将电路状态顺序编号。

(3) 按照要求列出电路的状态转换表或画出状态转换图。

这样,就把给定的逻辑问题抽象到一个时序逻辑函数了。

2. 状态化简

如果在状态转换图中出现这样两个状态:它们在相同的输入下转换到同一状态,并得到相同的输出,则称它们为等价状态。显然等价状态是重复的,可以合并为一个。电路的状态数越少,存储电路也就越简单。状态化简的目的就在于将等价状态尽可能地合并,以得到最简的状态转换图。

3. 状态分配

状态分配又称状态编码。通常有很多编码方法,编码方案选择得当,设计的电路就简单;反之,选得不好,则设计的电路就会复杂许多。在实际设计时,须综合考虑电路复杂度与电路性能这两个因素。在触发器资源丰富的 FPGA 或 ASIC 设计中,采用独热编码既可以使电路性能得到保证又可充分利用其触发器数量多的优势。

4. 选定触发器的类型并求出状态方程、驱动方程和输出方程

5. 按照方程得出逻辑图

用 Verilog HDL 来描述有限状态机,可以充分发挥硬件描述语言的抽象建模能力,使用 always 块语句和 case(if)等条件语句及赋值语句即可实现。具体的逻辑化简、逻辑电路及触发器映射均可由计算机自动完成,上述设计步骤中的第 2,4,5 步不再需要很多的人为干预,使电路设计工作得到简化,效率也有很大的提高。

4.1.1 用 Verilog HDL 语言设计可综合的状态机的指导原则

因为大多数 FPGA 内部的触发器数目相当多,又加上独热码状态机的译码逻辑最为简单,所以在设计采用 FPGA 实现的状态机时,往往采用独热码状态机(即每个状态只有一个寄存器置位的状态机)。

建议采用 case,casex 或 casez 语句来建立状态机的模型,因为这些语句表达清晰明了,可以方便地从当前状态分支转向下一个状态并设置输出。不要忘记写上 case 语句的最后一个分支 default,并将状态变量设为 'bx,这就等于告知综合器:case 语句已经指定了所有的状态。这样,综合器就可以删除不需要的译码电路,使生成的电路简洁,并与设计要求一致。

如果将缺省状态设置为某一确定的状态(例如,设置 default : state = state1)行不行呢?回答是:这样做有一个问题需要注意。因为尽管综合器产生的逻辑与设置 default : state='bx 时相同,但是状态机的 Verilog HDL 模型综合前和综合后的仿真结果会不一致。为什么会是这样呢?因为启动仿真器时,状态机所有的输入都不确定,因此立即进入 default 状态。这样的设置便会将状态变量设为 state1,但是实际硬件电路的状态机在通电之后,进入的状态是不确定的,很可能不是 state1 的状态,因此还是设置 default : state='bx 与实际情况相一致。但在有多余状态的情况下,还是应将缺省状态设置为某一确定的有效状态,因为这样做能使状态机在偶然进入多余状态后仍能在下一时钟跳变时返回正常工作状态,否则会引起死锁。

状态机应该有一个异步或同步复位端,以便在通电时将硬件电路复位到有效状态,也可以在操作中将硬件电路复位(大多数 FPGA 结构都允许使用异步复位端)。

目前,大多数综合器往往不支持在一个 always 块中由多个事件触发的状态机(即隐含状

态机,implicit state machines),为了能综合出有效的电路,用 Verilog HDL 描述的状态机应明确地由唯一时钟触发。

目前,大多数综合器不能综合采用 Verilog HDL 描述的异步状态机。异步状态机是没有确定时钟的状态机,它的状态转移不是由唯一的时钟跳变沿所触发。

千万不要使用综合工具来设计异步状态机。因为目前大多数综合工具在对异步状态机进行逻辑优化时会胡乱地简化逻辑,使综合后的异步状态机不能正常工作。

如果一定要设计异步状态机,建议采用电路图输入的方法,而不要用 Verilog HDL 输入的方法。

在 Verilog HDL 中,状态必须明确赋值。通常使用参数(parameters)或宏定义(define)语句加上赋值语句来实现。

下例使用参数(parameters)语句赋状态值:

```
parameter  state1 =2'h1, state2=2'h2;
           ...
   current _state = state2;     //把 current state 设置成 2'h2
           ...
```

使用宏定义(define)语句赋状态值见下面的例子:

```
`define  state1  2'h1
`define  state2  2'h2
   ...
current _state=`state2; //把 current state 设置成 2'h2
```

4.1.2 典型的状态机实例

例 宇宙飞船控制器的状态机

```
module  statmch1( launch _shuttle, land _shuttle, start _countdown,
                  start _trip _meter, clk, all _systems _go,
                  just _launched, is _landed, cnt, abort _mission);
   output  launch _shuttle, land _shuttle, start _countdown,
           start _trip _meter;
   input  clk, just _launched, is _landed, abort _mission,
          all _systems _go;
   input [3 : 0]  cnt;
   reg  launch _shuttle, land _shuttle, start _countdown,
        start _trip _meter;
   //设置独热码状态的参数
   parameter  HOLD=5'h1, SEQUENCE=5'h2, LAUNCH=5'h4;
   parameter  ON _MISSION=5'h8, LAND=5'h10;
   reg [4 : 0]  present _state, next _state;

   always @(negedge clk or posedge abort _mission)
     begin
     /* *把输出设置成某个缺省值,在下面的 case 语句中就不必再设置输出的缺省值* * */
```

```
    {launch_shuttle, land_shuttle, start_trip_meter, start_countdown} =4'b0;
    /*检查异步 reset 的值,即 abort_mission 的值*/
    if(abort_mission)
        next_state=LAND;
    else
  begin     // if_else_begin
    /*如果 reset 为零,把 next_state 赋值为 present_state*/
    next_state = present_state;
    /*根据 present_state 和输入信号,设置 next_state 和输出 output*/
    case ( present_state )
    HOLD :   if(all_systems_go)
                  begin
                        next_state = SEQUENCE;
                        start_countdown = 1;
                  end
    SEQUENCE :  if(cnt==0)
                        next_state = LAUNCH;

    LAUNCH :
          begin
              next_state = ON_MISSION;
              launch_shuttle = 1;
          end
    ON_MISSION :
           //取消使命前,一直留在使命状态
           if(just_launched)
                start_trip_meter = 1;
    LAND :   if(is_landed)
                 next_state = HOLD;
               else  land_shuttle = 1;
    /*把缺省状态设置为 'bx(无关)或某种已知状态,使其在做仿真时,在复位前就与实际情
      况相一致*/
    default : next_state = 'bx;
    endcase
  end          // end of if_else
 /*把当前状态变量设置为下一状态,待下一有效时钟沿来到时当前状态变量已设置了正确的
   状态值*/
  present_state = next_state;
 end           //end of always
endmodule
```

4.1.3 综合的一般原则

通常,综合的一般原则为:

(1) 综合之前一定要进行仿真,这是因为仿真会暴露逻辑错误,所以建议大家这样做。如果不做仿真,没有发现的逻辑错误会进入综合器,使综合的结果产生同样的逻辑错误。

(2) 每一次布局布线之后都要进行仿真,在器件编程或流片之前要做最后的仿真。

(3) 用 Verilog HDL 描述的异步状态机是不能综合的,因此应该避免用综合器来设计。如果一定要设计异步状态机,则可用电路图输入的方法来设计。

(4) 如果要为电平敏感的锁存器建模,使用连续赋值语句是最简单的方法。

4.1.4　语言指导原则

1. always 块

(1) 每个 always 块只能有一个事件控制"@(event-expression)",而且要紧跟在 always 关键字的后面。

(2) always 块可以表示时序逻辑或者组合逻辑。也可以用 always 块既表示电平敏感的透明锁存器同时又表示组合逻辑,但是不推荐使用这种描述方法,因为这样容易产生错误和多余的电平敏感的透明锁存器。

(3) 带有 posedge 或 negedge 关键字的事件表达式表示沿触发的时序逻辑,没有 posedge 或 negedge 关键字的则表示组合逻辑或电平敏感的锁存器。

(4) 每个表示时序的 always 块只能由一个时钟跳变沿触发,置位或复位最好也由该时钟跳变沿触发。

(5) 每个在 always 块中赋值的信号都必须定义成 reg 型或整型。整型变量缺省为 32 位,使用 Verilog 操作符可对其进行二进制求补的算术运算。综合器还支持整型变量的范围说明,这样就允许产生不是 32 位的整型变量。句法结构:integer[<msb>:<lsb>]<identifier>。

(6) always 块中应该避免组合反馈回路。每次执行 always 块时,在生成组合逻辑的 always 块中赋值的所有信号必须都有明确的值,否则,需要设计者在设计中加入电平敏感的锁存器来保持赋值前的最后一个值。只有这样,综合器才能正常生成电路;如果不这样做,综合器会发出警告,提示设计中插入了锁存器。如果在设计中当综合器认为不是电平敏感锁存器的组合回路时(例如设计中有异步状态机时),综合器会发出错误信息。

上述原则不太好理解,让我们再解释一下。也就是说,用 always 块设计纯组合逻辑电路时,在生成组合逻辑的 always 块中参与赋值的所有信号都必须有明确的值,即在赋值表达式右端参与赋值的信号都必须在 always @(敏感电平列表)中列出。如果在赋值表达式右端引用了敏感电平列表中没有列出的信号,那么在综合时将会为该没有列出的信号隐含地产生一个透明锁存器。这是因为该信号的变化不会立刻引起所赋值的变化,而必须等到敏感电平列表中某一个信号变化时,它的作用才体现出来,也就是相当于存在着一个透明锁存器,把该信号的变化暂存起来,待敏感电平列表中某一个信号变化时再起作用,纯组合逻辑电路不可能做到这一点。这样,综合后所得到的电路已经不是纯组合逻辑电路了,这时综合器会发出警告,提示设计中插入了锁存器。见下例:

例
```
input  a,b,c;
reg  e,d;
always @(a or b or c)
    begin
```

```
        e =d & a & b;
    /* 因为 d 没有在敏感电平列表中,所以 d 变化时,e 不能立刻变化,要等到 a 或 b 或 c 变化时
       才体现出来。这就是说,实际上相当于存在一个电平敏感的透明锁存器在起作用,把 d 信
       号的变化锁存其中 */
      d =e | c;
end
```

2. 赋 值

(1) 对一个寄存器型(reg)和整型(integer)变量给定位的赋值,只允许在一个 always 块内进行,如在另一 always 块中也对其赋值,这是非法的。

(2) 把某一信号赋值为 'bx,综合器就把它解释成无关状态,因此综合器为其生成的硬件电路最简洁。

4.2 可综合风格的 Verilog HDL 模块实例

4.2.1 组合逻辑电路设计实例

例[1] 8 位带进位端的加法器的设计实例(利用简单的算法描述)

```
module  adder_8(cout,sum,a,b,cin);
    output  cout;
    output [7:0]  sum;
    input  cin;
    input[7:0]  a,b;
    assign {cout,sum}=a+b+cin;
endmodule
```

例[2] 指令译码电路的设计实例(利用电平敏感的 always 块来设计组合逻辑)

```
//操作码的宏定义
'define  plus          3'd0
'define  minus         3'd1
'define  band          3'd2
'define  bor           3'd3
'define  unegate       3'd4

module  alu(out,opcode,a,b);
output [7:0]  out;
input [2:0]  opcode;
input [7:0]  a,b;
reg [7:0]  out;

always @(opcode or a or b)
//用电平敏感的 always 块描述组合逻辑
begin
    case(opcode)
```

```
    //算术运算
        'plus :   out=a+b;
        'minus :  out=a-b;
        //位运算
        'band :   out=a&b;
        'bor :    out=a|b;
        //单目运算
        'unegate :  out=~a;
        default :   out=8'hx;
    endcase
end
endmodule
```

例[3]　利用 task 和电平敏感的 always 块设计比较后置组信号的组合逻辑

```
module  sort4(ra,rb,rc,rd,a,b,c,d);
    parameter  t=3;
    output [t:0]  ra, rb, rc, rd;
    input [t:0]  a, b, c, d;
    reg [t:0]  ra, rb, rc, rd;

    always @(a or b or c or d) '
    //用电平敏感的 always 块描述组合逻辑
      begin
        reg [t:0]  va, vb, vc, vd;
        {va,vb,vc,vd}={a,b,c,d};
        sort2(va,vc);
        sort2(vb,vd);
        sort2(va,vb);
        sort2(vc,vd);
        sort2(vb,vc);
        {ra,rb,rc,rd}={va,vb,vc,vd};
      end

    task  sort2;
      inout [t:0]  x, y;
      reg [t:0]  tmp;
      if( x > y )
        begin
          tmp = x;
          x = y;
          y = tmp;
        end
    endtask
```

```
endmodule
```

例[4] 比较器的设计实例(利用赋值语句设计组合逻辑)

```
module  compare(equal,a,b);
parameter  size=1;
output  equal;
input [size-1:0]  a, b;
    assign  equal =(a==b)? 1 : 0;
endmodule
```

例[5] 3-8 译码器设计实例(利用赋值语句设计组合逻辑)

```
module  decoder(out,in);
output [7:0]  out;
input [2:0]  in;
    assign  out = 1'b1<<in;
/* * * * 把最低位的 1 左移 in(根据从 in 口输入的值)位,并赋予 out * * * * /
endmodule
```

例[6] 8-3 编码器的设计实例

编码器设计方案之一:

```
module  encoder1(none_on,out,in);
    output  none_on;
    output [2:0]  out;
    input [7:0]  in;
    reg [2:0]  out;
    reg  none_on;
    always @(in)
      begin: local
        integer  i;
        out = 0;
        none_on = 1;
        /* returns the value of the highest bit number turned on */
        for( i=0; i<8; i=i+1 )
          begin
            if( in[i] )
              begin
              out = i;
              none_on = 0;
              end
          end
      end
endmodule
```

编码器设计方案之二:

```
module  encoder2 ( none_on, out2, out1, out0, h, g, f, e, d, c, b, a);
input  h, g, f, e, d, c, b, a;
```

```
output   none_on, out2, out1, out0;
wire [3:0]  outvec;

assign  outvec= h? 4'b0111 : g? 4'b0110 : f? 4'b0101 :
  e? 4'b0100 : d? 4'b0011 : c? 4'b0010 : b? 4'b0001 :
  a? 4'b0000 : 4'b1000;

assign  none_on = outvec[3];
assign  out2 = outvec[2];
assign  out1 = outvec[1];
assign  out0 = outvec[0];

endmodule
```

编码器设计方案之三：

```
module  encoder3 (none_on, out2, out1, out0, h, g,f, e, d, c, b, a);
    input  h, g, f, e, d, c, b, a;
    output  out2, out1, out0;
    output  none_on;
    reg [3:0]  outvec;

    assign {none_on,out2,out1,out0} = outvec;

  always @( a or b or c or d or e or f or g or h)
    begin
       if(h)          outvec=4'b0111;
       else if(g)     outvec=4'b0110;
       else if(f)     outvec=4'b0101;
       else if(e)     outvec=4'b0100;
       else if(d)     outvec=4'b0011;
       else if(c)     outvec=4'b0010;
       else if(b)     outvec=4'b0001;
       else if(a)     outvec=4'b0000;
       else           outvec=4'b1000;
    end
endmodule
```

例[7]　多路器的设计实例

使用连续赋值、case 语句或 if_else 语句可以生成多路器电路。如果条件语句(case 或 if_else)中分支条件是互斥的话，综合器能自动地生成并行的多路器。

多路器设计方案之一：

```
modul  emux1(out, a, b, sel);
  output  out;
  input  a, b, sel;
```

```
    assign  out = sel? A : b;
endmodule
```

多路器设计方案之二：

```
module  mux2( out, a, b, sel);
    output  out;
    input  a, b, sel;
    reg  out;
    //用电平触发的 always 块来设计多路器的组合逻辑
    always @( a or b or sel )
    begin
    /* 检查输入信号 sel 的值,如为 1,输出 out 为 a;如为 0,输出 out 为 b */
    case( sel )
    1'b1 :  out = a;
    1'b0 :  out = b;
    default :  out = 'bx;
    endcase
    end
endmodule
```

多路器设计方案之三：

```
module  mux3( out, a, b, sel);
     output  out;
     input  a, b, sel;
     reg  out;
     always @( a or b or sel )
     begin
        if( sel )
            out = a;
        else
            out = b;
     end
endmodule
```

例[8] 奇偶校验位生成器设计实例

```
module  parity( even_numbits,odd_numbits,input_bus);
        output  even_numbits, odd_numbits;
        input [7 : 0]  input_bus;
             assign  odd_numbits = ^ input_bus;
             assign  even_numbits =~odd_numbits;
endmodule
```

例[9] 三态输出驱动器设计实例(用连续赋值语句建立三态门模型)

三态输出驱动器设计方案之一：

```
module  tristl( out, in, enable);
  output  out;
```

```
  input  in, enable;
      assign  out = enable? in : 'bz;
endmodule
```

三态输出驱动器设计方案之二：

```
module  trist2( out, in, enable );
  output  out;
  input  in, enable;
      //bufif1 是一个 Verilog 门级原语(primitive)
      bufif1  mybuf1(out, in, enable);
endmodule
```

例[10] 三态双向驱动器设计实例

```
module  bidir(tri_inout, out, in, en, b);
  inout  tri_inout;
  output  out;
  input  in, en, b;
      assign  tri_inout = en? in : 'bz;
      assign  out = tri_inout ^ b;
endmodule
```

4.2.2 时序逻辑电路设计实例

例[1] 触发器设计实例

```
module  dff( q, data, clk);
  output  q;
  input  data, clk;
  reg  q;
  always @( posedge clk )
    begin
      q = data;
    end
endmodule
```

例[2] 电平敏感型锁存器设计实例之一

```
module  latch1( q, data, clk);
  output  q;
  input  data, clk;
      assign  q = clk? data : q;
endmodule
```

例[3] 带置位和复位端的电平敏感型锁存器设计实例之二

```
module  latch2( q, data, clk, set, reset);
  output  q;
  input  data, clk, set, reset;
      assign  q= reset? 0 : ( set? 1 : (clk? data : q ) );
endmodule
```

例[4] 电平敏感型锁存器设计实例之三

```
module  latch3( q, data, clk);
  output  q;
  input data, clk;
  reg  q;
    always @(clk or data)
       begin
           if(clk)
             q=data;
       end
endmodule
```

注意:有的综合器会产生一警告信息,告诉你产生了一个电平敏感型锁存器。因为我们设计的就是一个电平敏感型锁存器,所以不用管这个警告信息。

例[5] 移位寄存器设计实例

```
module  shifter( din, clk, clr, dout);
  input  din, clk, clr;
  output [7 : 0]  dout;
  reg [7 : 0]  dout;
    always @(posedge clk)
       begin
         if(clr)   //清零
           dout = 8'b0;
         else
         begin
           dout = dout<<1;     //左移一位
           dout[0] = din;     //把输入信号放入寄存器的最低位
         end
       end

endmodule
```

例[6] 8 位计数器设计实例之一

```
module  counter1( out, cout, data, load, cin, clk);
    output [7 : 0]  out;
    output  cout;
    input [7 : 0]  data;
    input  load, cin, clk;
    reg [7 : 0]  out;
      always @(posedge clk)
        begin
          if( load )
            out = data;
          else
```

```
            out = out + cin;
    end
    assign cout= &out &cin;
    //只有当 out[7:0]的所有各位都为 1 并且进位 cin 也为 1 时才能产生进位 cout
    endmodule
```

例[7]　8 位计数器设计实例之二

```
module  counter2( out, cout, data, load, cin, clk);
    output [7:0]  out;
    output  cout;
    input [7:0]  data;
    input  load, cin, clk;
    reg [7:0]  out;
    reg  cout;
    reg [7:0]  preout;
    //创建 8 位寄存器
    always @(posedge clk)
      begin
        out = preout;
      end
    /****计算计数器和进位的下一个状态。注意:为提高性能不希望加载影响进位****/
    always @( out or data or load or cin )
       begin
           {cout, preout} = out + cin;
           if(load)
                      preout = data;
       end
endmodule
```

4.2.3　状态机的置位与复位

一、状态机的异步置位与复位

异步置位与复位是与时钟无关的。当异步置位与复位到来时,它们立即分别置触发器的输出为 1 或 0,不需要等到时钟沿到来才置位或复位。把异步置位与复位列入 always 块的事件控制括号内就能触发 always 块的执行,因此,当它们到来时就能立即执行指定的操作。

状态机的异步置位与复位是用 always 块和事件控制实现的。先让我们来看一下事件控制的语法。

1. 事件控制语法

```
@( <沿关键词 时钟信号
      or 沿关键词 复位信号
            or 沿关键词 置位信号> )
```

沿关键词包括 posedge(用于高电平有效的 set,reset 或上升沿触发的时钟)和 negedge(用于低电平有效的 set,reset 或下降沿触发的时钟),信号可以按任意顺序列出。

2. 事件控制实例

(1) 异步高电平有效的置位(时钟的上升沿)

```
@(posedge clk or posedge set)
```

(2) 异步低电平有效的复位(时钟的上升沿)

```
@(posedge clk or negedge reset)
```

(3) 异步低电平有效的置位和高电平有效的复位(时钟的上升沿)

```
@( posedge clk or negedge set or posedge reset )
```

(4) 带异步高电平有效的置位与复位的 always 块样板

```
always @(posedge clk or posedge set or posedge reset)
  begin
    if(reset)
      begin
      /* 置输出为 0 */
      end
    else
      if(set)
        begin
        /* 置输出为 1 */
        end
      else
        begin
        /* 与时钟同步的逻辑 */
        end
  end
```

(5) 带异步高电平有效的置/复位端的 D 触发器实例

```
module  dff1( q, qb, d, clk, set, reset );
    input  d, clk, set, reset;
    output  q, qb;
    //声明 q 和 qb 为 reg 类型,因为它需要在 always 块内赋值
    reg  q, qb;

    always @( posedge clk or posedge set or posedge reset )
    begin
       if(reset)
         begin
           q = 0;
           qb = 1;
         end
       else
         if (set)
           begin
             q = 1;
```

```
            qb = 0;
          end
        else
          begin
            q=d;
            qb=~d;
          end
    end
endmodule
```

二、状态机的同步置位与复位

同步置位与复位是指只有在时钟的有效跳变时刻置位或复位，信号才能使触发器置位或复位(即，使触发器的输出分别转变为逻辑 1 或 0)。

不要把 set 和 reset 信号名列入 always 块的事件控制表达式，因为当它们有变化时不应触发 always 块的执行。相反，always 块的执行应只由时钟有效跳变沿触发，是否置位或复位应在 always 块中首先检查 set 和 reset 信号的电平。

1. 事件控制语法

@(<沿关键词　时钟信号>)

沿关键词指 posedge(正沿触发的时钟)或 negedge(负沿触发的时钟)。

2. 事件控制实例

(1) 正沿触发

```
@(posedge clk)
```

(2) 负沿触发

```
@(negedge clk)
```

(3) 同步的具有高电平有效的置位与复位端的 always 块样板

```
always @(posedge clk)
    begin
      if(reset)
        begin
          /* 置输出为 0 */
        end
      else
          if(set)
            begin
            /* 置输出为 1 */
            end
          else
            begin
            /* 与时钟同步的逻辑 */
            end
end
```

(4) 同步的具有高电平有效的置位/复位端的 D 触发器

```
module  dff2( q, qb, d, clk, set, reset);
    input  d, clk, set, reset;
    output  q, qb;
    reg  q, qb;
    always @(posedge clk)
      begin
        if(reset)
          begin
            q=0;
            qb=1;
          end
        else
          if(set)
            begin
              q=1;
              qb=0;
            end
          else
            begin
              q=d;
              qb=~d;
            end
      end
endmodule
```

4.2.4 复杂时序逻辑电路设计实践

[例 1] 一个简单的状态机设计——序列检测器

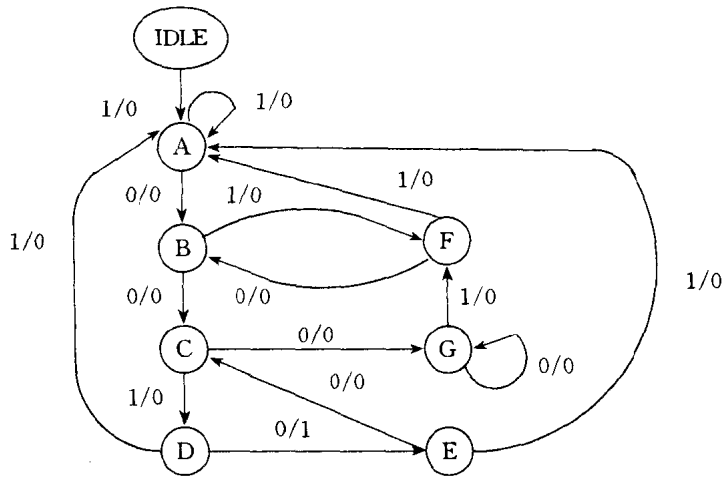
序列检测器是时序数字电路设计中经典的教学范例。下面我们将用 Verilog HDL 语言来描述、仿真并实现它。

序列检测就是将一个指定的序列从数字码流中识别出来。本例中，我们将设计一个“10010”序列的检测器。设 X 为数字码流输入，Z 为检出标记输出，高电平表示“发现指定序列”，低电平表示“没有发现指定序列”。考虑码流为“110010010000100101…”，则有下表：

时钟	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
X	1	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	1	…
Z	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	…

在时钟 2～6，码流 X 中出现指定序列“10010”，对应输出 Z 在第 6 个时钟变为高电平“1”，表示“发现指定序列”。同样地，在时钟 13～17，码流 X 中再次出现指定序列“10010”，Z 输出为“1”。注意，在时钟 5～9 还有一次检出，但它是与第一次检出的序列重叠的，即前者的前面两位同时也是后者的最后两位。

根据以上逻辑功能描述，我们可以分析得出状态转换图如下：



其中状态 A～E 表示 5 位序列“10010”按顺序正确地出现在码流中。考虑到序列重叠的可能，转换图中还有状态 F，G。另外，电路的初始状态设为 IDLE。

进一步，我们得出 Verilog HDL 代码。

```
//文件:seqdet.v
module seqdet( x, z, clk, rst);
input x,clk, rst;
output z;
reg [2 : 0] state;//状态寄存器
wire z;

parameter   IDLE= 'd0,    A='d1,  B='d2,
                          C='d3,  D='d4,
                          E='d5,  F='d6,
                          G='d7;

assign  z=(state==D && x==0) ? 1 : 0;
always @(posedge clk or negedge rst)
    if(!rst)
                begin
                state<=IDLE;
                end
            else
                casex( state)
                    IDLE : if(x==1)
                                begin
                                    state<=A;
                                end
                    A :     if (x==0)
```

```
                begin
                    state<=B;
                end
        B:  if (x==0)
                begin
                    state<=C;
                end
            else
                begin
                    state<=F;
                end
        C:  if(x==1)
                begin
                    state<=D;
                end
            else
                begin
                    state<=G;
                end
        D:  if(x==0)
                begin
                    state<=E;
                end
            else
                begin
                    state<=A;
                end
        E:  if(x==0)
                begin
                    state<=C;
                end
            else
                begin
                    state<=A;
                end
        F:  if(x==1)
                begin
                    state<=A;
                end
            else
                begin
                    state<=B;
                end
```

```
            G:    if(x==1)
                        begin
                            state<=F;
                        end
            default : state<=IDLE;
            endcase
endmodule
```

为了验证其正确性，我们接着编写测试用代码。

```
//文件:seqdet.tf
'timescale  1ns/1ns
module  t;
reg clk, rst;
reg [23 : 0]  data;
wire z,x;
assign  x=data[23];

initial
    begin
        clk<=0;
        rst<=1;
        #2  rst<=0;
        #30  rst<=1;  //复位信号
        data='b1100_1001_0000_1001_0100; //码流数据
    end

always  #10  clk=~clk; //时钟信号
always @ (posedge clk) // 移位输出码流
        data={data[22 : 0],data[23]};

seqdet  m ( .x(x), .z(z), .clk(clk), .rst(rst)); //调用序列检测器模块

// Enter fixture code here

endmodule
```

其中，X 码流的产生采用了移位寄存器的方式，以方便更改测试数据。仿真结果如图 4.2 所示：

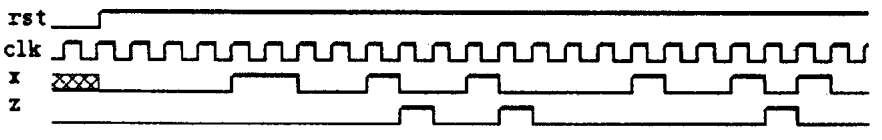


图 4.2　例[1]的仿真结果

从图 4.2 的波形中可以看到程序代码正确地完成了所要设计的逻辑功能。另外，seqdet.v 的编写采用了可综合的 Verilog HDL 风格，它可以通过综合器的综合最终实现到 FPGA 中。

说明：以上编程、仿真、综合是在 PC WINDOWS NT 4.0 操作系统及 QuickLogic SPDE 环境下通过。

例[2] 下面将介绍一个串行 EEPROM 读写器件的设计过程和有关的 Verilog HDL 程序。这是我们利用 Verilog HDL 完成的第一个较大的工程设计。这个器件能把并行数据和地址信号转变为串行 EEPROM 能识别的串行码，并把数据写入相应的地址，或根据并行的地址信号从 EEPROM 相应的地址读取数据并把相应的串行码转换成并行的数据放到并行地址总线上。当然，还需要有相应的读信号或写信号和应答信号配合才能完成以上的操作。如图 4.3 所示为 EEPROM 读写电路及其测试电路。

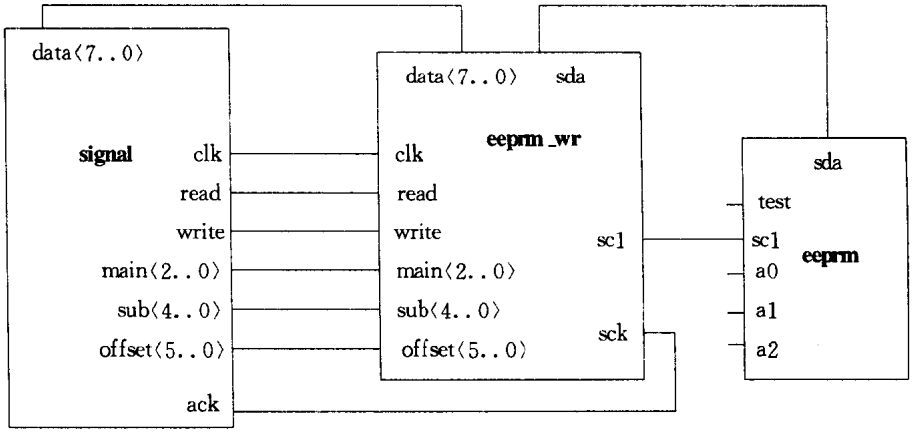


图 4.3 EEPROM 读写电路及其测试电路

为了设计这样一个电路，首先要设计一个 EEPROM 的 Verilog HDL 模型。而设计这样一个模型需要仔细地阅读和分析 EEPROM 器件的说明书，因为 EEPROM 不是我们要设计的对象，而是我们验证设计对象所需要的器件，所以只需设计一个 EEPROM 的行为模型，而不需要可综合风格的模型，这就大大简化了设计过程。下面的 Verilog HDL 程序就是这个 EEPROM(AT24C02/4/8/16) 的行为模型，请读者查阅 AT24C02/4/8/16 说明书，对照下面的 Verilog HDL 程序理解设计的要点。

```
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* * * eeprm is a verilog HDL model for AT24C02/4/8/16 2-Wire Serial CMOS
//* * * EEPROMs. It is used in simulation to substitute the real EEPRM to
//* * * verify if the writing or reading of serial eeprom is Okay.
//* * * This module is a behavioral model for simulation only, not
//* * * synthesizable. It's writing and reading functions are verified.
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
'define  timeslice2  100
module  eeprm(a0, a1, a2, scl, sda,test);
    input  a0,a1,a2;
    input  scl,test;
    inout  sda;
```

```
    wire  sda;

    reg [7:0]  memory [2047:0], memory_buf, sda_buf;
               //16k bits memory means 2048 bytes
    reg [7:0]  shift_buf, addr_word, cmm_word;
    reg  out_flag;
    reg[1:0]  W_R_state;
    integer  i;
//------------------------------------------------------------
parameter     r7= 8'b10101111, w7= 8'b10101110,      //main7
              r6= 8'b10101101, w6= 8'b10101100,            //main6
              r5= 8'b10101011, w5= 8'b10101010,            //main5
              r4= 8'b10101001, w4= 8'b10101000,            //main4
              r3= 8'b10100111, w3= 8'b10100110,            //main3
              r2= 8'b10100101, w2= 8'b10100100,            //main2
              r1= 8'b10100011, w1= 8'b10100010,            //main1
              r0= 8'b10100001, w0= 8'b10100000;            //main0
//------------------------------------------------------------
initial
  begin
     addr_word = 0; cmm_word = 0; out_flag = 0;
     sda_buf= 0; W_R_state = 2'b00;
     memory_buf = 0;
     for ( i=0; i<=2047; i=i+1 )
     memory[ i ] = 0;
  end

//------------------------------------------------------------
assign sda = (out_flag == 1) ? sda_buf[7] : 1'bz;
//* * * out_flag is used to notify sda is linked to sda_buf
//* * * out_flag == 0 means sda_buf is seperated from sda.
//------------------------------------------------------------

//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* * * *        start receiving
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
always @(negedge sda)
    if(scl == 1 )
      begin
         W_R_state = W_R_state + 1;
         if ( W_R_state == 2'b11)
         begin    disable  work2;  end
      end
```

```
//-------------------------------------------------------------
//* * * * * main state machine
//-------------------------------------------------------------
always @(posedge sda)
    if (scl == 1 )
      stop _ W _ R;
    else
    begin
        casex(W _ R _ state)

        2'b01 : begin
                  work1;
                  if ( cmm _ word==w7 || cmm _ word==w6 || cmm _ word==w5
                       || cmm _ word==w4 || cmm _ word==w3 ||
                       cmm _ word==w2 ||cmm _ word==w1 || cmm _ word==w0 )
                  begin
                       W _ R _ state = 2'b10;
                       work2;
                  end
              else
                  W _ R _ state = 2'b00;
        end
    2'b11 : work3;

    default : W _ R _ state=2'b00;

    endcase
end
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* * * *        stop receiving
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
task  stop _ W _ R;
  begin
     W _ R _ state = 2'b00;
     addr _ word = 0;
     cmm _ word = 0;
     out _ flag = 0;
     sda _ buf = 0;
  end
endtask
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* * * Task work1 is used to do the real work of analysis sda bit stream command,
//* * * and according to the command words in bit stream to decide what action it
```

```
//* * * should take, writing into memory or reading out from memory. Then the real
//* * * work of bit by bit storing into memory or bit by bit reading out from memory
//* * * begins and continues, according to the command word it received.
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//-------------------------------------------------------------------------
task  work1;
      begin
         shift_in(cmm_word);
         shift_in(addr_word);
         if (cmm_word==w7 || cmm_word==w6 || cmm_word==w5 ||
            cmm_word==w4 || cmm_word==w3 ||
            cmm_word==w2 || cmm_word==w1 ||cmm_word==w0 )
            W_R_state = 2'b10;
         else
           W_R_state = 2'b00;
      end
endtask
//-------------------------------------------------------------------------
task  work2;
  begin
     shift_in ( memory_buf );
     memory[cmm_word[3:1]*256+addr_word] = memory_buf;
     $display ( "eeprm%d--------memory[%d*256+%d]=%d",
                cmm_word[3:1], cmm_word[3:1], addr_word,
                memory[cmm_word[3:1]*256 + addr_word] );
     W_R_state =2'b00;
  end
endtask
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* * * * * *     work3
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
task  work3;
  begin
     shift_in(cmm_word);
     if (cmm_word==r7 || cmm_word==r6 || cmm_word==r5 || cmm_word==r4
        || cmm_word==r3 || cmm_word==r2 || cmm_word==r1 || cmm_word==r0 )
     begin
        sda_buf = memory[cmm_word[3:1]*256+addr_word];
        shift_out;
        W_R_state= 2'b00;
     end
  end
endtask
```

```
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* * * Task shift_in is used to get the sda serial bits into shift[7:0],
//* * * one bit every one scl negedge, when serial bits are required to
//* * * be stored into EEPROM.
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
task  shift_in;
  output [7:0]  shift;
  begin
     @ (posedge scl) shift[7] = sda;
     @ (posedge scl) shift[6] = sda;
     @ (posedge scl) shift[5] = sda;
     @ (posedge scl) shift[4] = sda;
     @ (posedge scl) shift[3] = sda;
     @ (posedge scl) shift[2] = sda;
     @ (posedge scl) shift[1] = sda;
     @ (posedge scl) shift[0] = sda;
     @(negedge scl)        //Send out acknowledge
        begin
          #'timeslice2  out_flag = 1;        //link sda_buf to sda ready to send serial data
          sda_buf = 0;                       //make sda low to send an acknowledge
        end
      @(negedge scl)
            #'timeslice2  out_flag = 0;  //delay half scl2 seprate sda_buf with sda .
  end
endtask
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* * * Task shift_out is used to send out the byte, sda_buf[7:0], to sda;
//* * * one bit every one scl negedge, when the byte stored in memory is
//* * * required, out_flag is used to link sda with the MSB of sda_buf[7:0].
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
task  shift_out;
  begin
     out_flag = 1;
     for ( i=6; i>=0; i=i-1 )
     @ (negedge scl)
          #'timeslice2  sda_buf = sda_buf<<1;
     @(negedge scl)
          #'timeslice2  sda_buf[7] = 1;    //no ack
     @(negedge scl)
          #'timeslice2  out_flag = 0;     //seprate sda_buf with sda
  end
endtask
//-----------------------------------------------------------------
```

```
endmodule
```

为了设计一个串行 EEPROM 读写器件还需要设计一个信号源,这个信号源能产生相应的读信号、写信号、并行地址信号、并行数据信号,并能接收串行 EEPROM 读写器件的应答信号 (ack)来调节发送或接收数据的速度。下面的程序就是这个信号源的 Verilog HDL 模型,它可用于串行 EEPROM 读写器件的调试和验证。

```
`timescale 1ns/1ns

`define timeslice 200

module signal (ack, clk, read, write, main, sub, offset, data, a, test);

    input ack;
    output clk;
    output read,write;
    output [2:0] main;
    output [4:0] sub;
    output [3:0] offset;
    output a;
    output test;
    inout [7:0] data;

    reg clk;
    reg read,write;
    reg [2:0] main;
    reg [4:0] sub;
    reg [3:0] offset;
    reg a;
    reg test;
//*******************************************
//*** the registers used within this module
//*******************************************
reg write_read;
reg [7:0] datatoeeprm;
//-------------------------------------------

assign data = (write_read)? 'bz : datatoeeprm;
initial
  begin
    a=0;
    test=0;
    clk=0;
    write_read=0;
```

```
      write =0; #(3 * 'timeslice);
      write =1; #('timeslice);
      write =0;
      read =0;
      #(4500 * 'timeslice); write_read=1;
   end

//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* * * the clock
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
always   #('timeslice/2)
         clk =~ clk;

//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* * * the write and read signal
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
always @(posedge ack)
    if(write_read==0)
       begin #(5 * 'timeslice);
            write=1; #('timeslice);
            write=0;
       end
    else
       begin #(5 * 'timeslice);
            read=1; #('timeslice);
            read=0;
       end
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* * * datatoeeprm send to eeprm_wr
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
initial
begin
//* * * * * * * * * * * * * * * * * * * * * * write to EEPROM * * * * * * * * * * * * * *
datatoeeprm='d15;sub=0;offset=0;main=0;                           //memory[0]=='d15
@(posedge ack)   datatoeeprm='d01; offset=1;                      //memory[1]=='d01
@(posedge ack)   datatoeeprm='d06; offset=6;                      //memory[6]=='d06
@(posedge ack)   datatoeeprm='d12; offset=12;                     //memory[12]=='d12
@(posedge ack)   datatoeeprm='d16; sub=1;offset=0;                //memory[16]=='d16
@(posedge ack)   datatoeeprm='d22; offset=6;                      //memory[22]=='d22
@(posedge ack)   datatoeeprm='d24; sub=2;offset=0;                //memory[24]=='d24
@(posedge ack)   datatoeeprm='d27; offset=3;                      //memory[27]=='d27

@(posedge ack)   datatoeeprm='d32;sub=3; offset=0;main=1;         //memory[32]=='d32
```

```
@(posedge ack)   datatoeeprm='d33; offset=1;                    //memory[33]=='d33
@(posedge ack)   datatoeeprm='d40;sub=4; offset=0;              //memory[40]=='d40
@(posedge ack)   datatoeeprm='d41; offset=1;                    //memory[41]=='d41
@(posedge ack)   datatoeeprm='d48;sub=5; offset=0;              //memory[48]=='d48
@(posedge ack)   datatoeeprm='d49; offset=1;                    //memory[49]=='d49
@(posedge ack)   datatoeeprm='d56;sub=6; offset=0;              //memory[56]=='d56
@(posedge ack)   datatoeeprm='d61; offset=5;                    //memory[61]=='d61

@(posedge ack)   datatoeeprm='d64;sub=7; offset=0;main=2;       //memory[64]=='d64
@(posedge ack)   datatoeeprm='d65; offset=1;                    //memory[65]=='d65
@(posedge ack)   datatoeeprm='d72;sub=8; offset=0;              //memory[72]=='d72
@(posedge ack)   datatoeeprm='d76; offset=4;                    //memory[76]=='d76
@(posedge ack)   datatoeeprm='d80;sub=9; offset=0;              //memory[80]=='d80
@(posedge ack)   datatoeeprm='d85; offset=5;                    //memory[85]=='d85
@(posedge ack)   datatoeeprm='d88;sub=10; offset=0;             //memory[88]=='d88
@(posedge ack)   datatoeeprm='d92; offset=4;                    //memory[92]=='d92

@(posedge ack)   datatoeeprm='d96;sub=11; offset=0;main=3;      //memory[96]=='d96
@(posedge ack)   datatoeeprm='d98; offset=2;                    //memory[98]=='d98
@(posedge ack)   datatoeeprm='d104;sub=12;offset=0;             //memory[104]=='d104
@(posedge ack)   datatoeeprm='d110; offset=6;                   //memory[110]=='d110
@(posedge ack)   datatoeeprm='d112;sub=13;offset=0;             //memory[112]=='d112
@(posedge ack)   datatoeeprm='d117; offset=5;                   //memory[117]=='d117
@(posedge ack)   datatoeeprm='d120;sub=14;offset=0;             //memory[120]=='d120
@(posedge ack)   datatoeeprm='d121; offset=1;                   //memory[121]=='d121

@(posedge ack)   datatoeeprm ='d128; sub=15;offset=0;main=4;    //memory[128]=='d128
@(posedge ack)   datatoeeprm='d133; offset=5;                   //memory[133]=='d133
@(posedge ack)   datatoeeprm='d136;sub=16;offset=0;             //memory[136]=='d136
@(posedge ack)   datatoeeprm='d141; offset=5;                   //memory[141]=='d141
@(posedge ack)   datatoeeprm='d144;sub=17;offset=0;             //memory[144]=='d144
@(posedge ack)   datatoeeprm='d145; offset=1;                   //memory[145]=='d145
@(posedge ack)   datatoeeprm='d152;sub=18;offset=0;             //memory[152]=='d152
@(posedge ack)   datatoeeprm='d156; offset=4;                   //memory[156]=='d156

@(posedge ack)   datatoeeprm='d160;sub=19;offset=0;main=5;      //memory[160]=='d160
@(posedge ack)   datatoeeprm='d166; offset=6;                   //memory[166]=='d166
@(posedge ack)   datatoeeprm='d168;sub=20;offset=0;             //memory[168]=='d168
@(posedge ack)   datatoeeprm='d169; offset=1;                   //memory[169]=='d169
@(posedge ack)   datatoeeprm='d176;sub=21;offset=0;             //memory[176]=='d176
@(posedge ack)   datatoeeprm='d182; offset=6;                   //memory[182]=='d182
@(posedge ack)   datatoeeprm='d184;sub=22;offset=0;             //memory[184]=='d184
@(posedge ack)   datatoeeprm='d185; offset=1;                   //memory[185]=='d185
```

```
@(posedge ack)    datatoeeprm='d192;sub=23;offset=0;main=6;      //memory[192]=='d192
@(posedge ack)    datatoeeprm='d196; offset=4;                   //memory[196]=='d196
@(posedge ack)    datatoeeprm='d200;sub=24;offset=0;             //memory[200]=='d200
@(posedge ack)    datatoeeprm='d207; offset=7;                   //memory[207]=='h207
@(posedge ack)    datatoeeprm='d208;sub=25;offset=0;             //memory[208]=='d208
@(posedge ack)    datatoeeprm='d212; offset=4;                   //memory[212]=='d212
@(posedge ack)    datatoeeprm='d216;sub=26;offset=0;             //memory[216]=='d216
@(posedge ack)    datatoeeprm='d217; offset=1;                   //memory[217]=='d217

@(posedge ack)    datatoeeprm='d224; sub=27; offset=0; main=7;   //memory[224]=='d224
@(posedge ack)    datatoeeprm='d230; offset=6;                   //memory[230]=='d230
@(posedge ack)    datatoeeprm='d232;sub=28;offset=0;             //memory[232]=='d232
@(posedge ack)    datatoeeprm='d235; offset=3;                   //memory[235]=='d235
@(posedge ack)    datatoeeprm='d240;sub=29;offset=0;             //memory[240]=='d240
@(posedge ack)    datatoeeprm='d243; offset=3;                   //memory[243]=='d243
@(posedge ack)    datatoeeprm='d248;sub=30;offset=0;             //memory[248]=='h248
@(posedge ack)    datatoeeprm='d255; offset=7;                   //memory[255]=='h255
//* * * * * * * * * * * * * * * * read data from EEPROM begins here * * * * * * *
@(posedge write_read)  offset=0;sub=0;main=0;                    //memory[0]=='d15
@(posedge ack)
@(posedge ack)    offset=1;                                      //memory[1]=='d01
@(posedge ack)    offset=6;                                      //memory[6]=='d06
@(posedge ack)    offset=12;                                     //memory[12]=='d12
@(posedge ack)    sub=1; offset=0;                               //memory[16]=='d16
@(posedge ack)    offset=6;                                      //memory[22]=='d22
@(posedge ack)    sub=2; offset=0;                               //memory[24]=='d24
@(posedge ack)    offset=3;                                      //memory[27]=='d27

@(posedge ack)    sub=3; offset=0;main=1;                        //memory[32]=='d32
@(posedge ack)    offset=1;                                      //memory[33]=='d33
@(posedge ack)    sub=4; offset=0;                               //memory[40]=='d40
@(posedge ack)    offset=1;                                      //memory[41]=='d41
@(posedge ack)    sub=5; offset=0;                               //memory[48]=='d48
@(posedge ack)    offset=1;                                      //memory[49]=='d49
@(posedge ack)    sub=6; offset=0;                               //memory[56]=='d56
@(posedge ack)    offset=5;                                      //memory[61]=='d61

@(posedge ack)    sub=7; offset=0;main=2;                        //memory[64]=='d64
@(posedge ack)    offset=1;                                      //memory[65]=='d65
@(posedge ack)    sub=8; offset=0;                               //memory[72]=='d72
@(posedge ack)    offset=4;                                      //memory[76]=='d76
@(posedge ack)    sub=9; offset=0;                               //memory[80]=='d80
```

```
@(posedge ack)    offset=5;                          //memory[85]==='d85
@(posedge ack)    sub=10;offset=0;                   //memory[88]==='d88
@(posedge ack)    offset=4;                          //memory[92]==='d92

@(posedge ack)    sub=11;offset=0;main=3;            //memory[96]==='d96
@(posedge ack)    offset=2;                          //memory[98]==='d98
@(posedge ack)    sub=12;offset=0;                   //memory[104]==='d104
@(posedge ack)    offset=6;                          //memory[110]==='d110
@(posedge ack)    sub=13;offset=0;                   //memory[112]==='d112
@(posedge ack)    offset=5;                          //memory[117]==='d117
@(posedge ack)    sub=14;offset=0;                   //memory[120]==='d120
@(posedge ack)    offset=1;                          //memory[121]==='d121

@(posedge ack)    sub=15;offset=0;main=4;            //memory[128]==='d128
@(posedge ack)    offset=5;                          //memory[133]==='d133
@(posedge ack)    sub=16;offset=0;                   //memory[136]==='d136
@(posedge ack)    offset=5;                          //memory[141]==='d141
@(posedge ack)    sub=17;offset=0;                   //memory[144]==='d144
@(posedge ack)    offset=1;                          //memory[145]==='d145
@(posedge ack)    sub=18;offset=0;                   //memory[152]==='d152
@(posedge ack)    offset=4;                          //memory[156]==='d156

@(posedge ack)    sub=19;offset=0;main=5;            //memory[160]==='d160
@(posedge ack)    offset=6;                          //memory[166]==='d16
@(posedge ack)    sub=20;offset=0;                   //memory[168]==='d168
@(posedge ack)    offset=1;                          //memory[169]==='d169
@(posedge ack)    sub=21;offset=0;                   //memory[176]==='d176
@(posedge ack)    offset=6;                          //memory[182]==='d182
@(posedge ack)    sub=22;offset=0;                   //memory[184]==='d184
@(posedge ack)    offset=1;                          //memory[185]==='d185

@(posedge ack)    sub=23;offset=0;main=6;            //memory[192]==='d192
@(posedge ack)    offset=4;                          //memory[196]==='d196
@(posedge ack)    sub=24;offset=0;                   //memory[200]==='d200
@(posedge ack)    offset=7;                          //memory[207]==='h207
@(posedge ack)    sub=25;offset=0;                   //memory[208]==='d208
@(posedge ack)    offset=4;                          //memory[212]==='d212
@(posedge ack)    sub=26;offset=0;                   //memory[216]==='d216
@(posedge ack)    offset=1;                          //memory[217]==='d217

@(posedge ack)    sub=27;offset=0;main=7;            //memory[224]==='d224
@(posedge ack)    offset=6;                          //memory[230]==='d230
@(posedge ack)    sub=28;offset=0;                   //memory[232]==='d232
```

```
@(posedge ack)   offset=3;                    //memory[235]=='d235
@(posedge ack)   sub=29;offset=0;             //memory[240]=='d240
@(posedge ack)   offset=3;                    //memory[243]=='d243
@(posedge ack)   sub=30;offset=0;             //memory[248]=='h248
@(posedge ack)   offset=7;                    //memory[255]=='h255
end
endmodule
```

下面的程序是一个串行 EEPROM 读写器的可综合的 Verilog HDL 模型,它接收来自信号源模型产生的读信号、写信号、并行地址信号、并行数据信号,并把它们转换为相应的串行信号发送到串行 EEPROM(AT24C02/4/8/16) 的行为模型中去。它还将应答信号(ack)发送到信号源模型,以便让信号源来调节发送或接收数据的速度以配合 EEPROM 模型的接收(写)和发送(读)数据。因为它是我们的设计对象,所以它的仿真不但要正确无误,还需要可综合。

这个程序是我们所做的真实设计的一部分,它实际上是一个嵌套的状态机,由主状态机和从状态机通过由控制线启动的总线在不同的输入信号的情况下构成不同功能的较复杂的有限状态机。这个有限状态机只有唯一的驱动时钟——clk。程序由两个大的任务和若干个较小的任务所组成。因为这一设计最终要在 ASIC 上实现,故其状态机没有采用独热编码。若需改变状态编码,只需改变程序中的 parameter 定义即可。读者可以通过模仿这一程序来编写较复杂的可综合 Verilog HDL 模块程序。这个设计已通过后仿真,并可在 FPGA 上实现布局布线。

```
//*********************************************
//***** module name : eeprm_wr.v
//*********************************************
module  eeprm_wr (clk, read, write, main, sub, offset, data, ack, scl, sda );
input  clk;
input  read, write;
input [2:0]  main;
input [4:0]  sub;
input [3:0]  offset;

output  ack;
output  scl;

inout  sda;
inout [7:0]  data;

reg  ack;
reg  scl;
//*******************************************
//*** the register used within this module
//*******************************************
reg  scl_flag, scl_o;
reg read_flag, write_flag;
```

```
reg [7 : 0]  sh8out _ buf;
reg [4 : 0]  sh8out _ state;
reg [4 : 0]  sh8in _ state;
reg [1 : 0]  head _ state;
reg [1 : 0]  stop _ state;
reg [3 : 0]  main _ state;
reg [1 : 0]  head _ buf, stop _ buf;

reg  finish _ F;
reg [7 : 0]  data _ from _ rm;
wire  sda1, sda2, sda3, sda4;
reg  link _ sda, link _ read _ eeprm;
reg  link _ head, link _ write _ eeprm, link _ stop ;
wire  reset;
///* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* * * the buses used within this module
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
assign  sda2 = (link _ write _ eeprm) ? sh8out _ buf[7] : 1'b0;
assign  sda3 = (link _ stop) ? stop _ buf[1] : 1'b0;
assign  sda1 = (link _ head) ? head _ buf[1] : 1'b0;
assign  sda4 = (sda1 | sda2 | sda3);

assign  sda = (link _ sda) ? sda4 : 1'bz;
assign  data =( (link _ read _ eeprm) ? data _ from _ rm : 8'hzz);
//————————————————————————————————————————

//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* * * the parameter definition
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
parameter
//————————————————————Main State Machine
Main _ start1          = 1,
Main _ dev _ write     = 2,
Main _ addr _ write    = 3,
Main _ data _ write    = 4,
Main _ start2          = 5,
Main _ dev _ read      = 6,
Main _ data _ read     = 7,
Main _ stop            = 8,
Main _ stop _ ack      = 9,
//————————————————Shift 8bits out state machine
sh8out _ idle = 0,
sh8out _ bit0 = 1,
```

```
sh8out_bit1 = 2,
sh8out_bit2 = 3,
sh8out_bit3 = 4,
sh8out_bit4 = 5,
sh8out_bit5 = 6,
sh8out_bit6 = 7,
sh8out_bit7 = 8,
sh8out_ack = 9,
sh8out_end = 10,

//-------------------shift 8bits in state machine
sh8in_idle = 0,
sh8in_bit7 = 1,
sh8in_bit6 = 2,
sh8in_bit5 = 3,
sh8in_bit4 = 4,
sh8in_bit3 = 5,
sh8in_bit2 = 6,
sh8in_bit1 = 7,
sh8in_bit0 = 8,
sh8out_end = 10,
linkbus_state = 9,

//---------------shift head out state machine
sh_head_idle=0,
sh_head_bit1=1,
sh_head_end=2,
//------------shift stop out state machine
sh_stop_idle =0,
sh_stop_bit1 =1,
sh_stop_end =2,
//-------------------common parameters
idle   = 0,
YES    = 1,
NO     = 0;

//****************************************
//**** scl is two divison frequency of clk
//**** posedge of clk is in the middle of scl high or low level
//****************************************
//*****************************************
//**** scl_o is also two divison frequency of clk but has half phase
//**** difference with scl
```

```
//* * * * negedge of clk is in the middle of scl _o high or low level
//* * * * posedge of clk is in the same edge of scl _o
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

always @(posedge read or posedge write or negedge clk )
      if ( write || read )
        begin
           scl _flag<=0;
           scl <= 0;
        end
      else
          begin
            scl _flag<=1;
            scl <=~ scl;
          end

always @(posedge read or posedge write or posedge clk)
      if ( read || write)
          begin     scl _o<=1;     end
      else if ( scl _flag )
          begin     scl _o <=~ scl _o;  end
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* * * * *THE MAIN PROGRAM
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

always @( posedge clk )
    if ( write )
      begin
         write _flag<=1;
         read _flag<=0;
         ack<=0;
         main _state<=idle;
      end
    else if ( write _flag )
      begin      Writing _To _Eeprm;      end
    else if ( read )
         begin
            read _flag<=1;
            write _flag<=0;
            ack<=0;
            main _state<=idle;
         end
    else if ( read _flag )
```

```
        begin   Reading _ From _ Eeprm;   end
//-------------The End of the Main Program -------------

//*************************************************
//*** TASK OF WRITEING DATA TO EEPRM
//*************************************************

task   Writing _ To _ Eeprm;
begin
casex(main _ state)
//-------------------------------state==0
idle : if(scl)
       begin
          link _ read _ eeprm <= NO;
          link _ write _ eeprm <= NO;
          link _ stop <= NO;
          link _ sda <= YES;
          link _ head <= YES;
          head _ buf[1 : 0] <= 2'b10;
          head _ state <= sh _ head _ bit1;
          sh8out _ buf <= 0;
          sh8out _ state <= sh8out _ bit0;
          stop _ buf[1 : 0] <= 2'b01;
          stop _ state <= sh _ stop _ idle;
          finish _ F <= NO;
          ack <= NO;
          main _ state <= Main _ start1;
       end

//-------------------------------state==1
Main _ start1 : if (finish _ F == 0)
                  begin   shift _ head;   end
                else if (finish _ F == 1 )
                     begin
                        head _ state <= idle;
                        sh8out _ buf[7 : 0] <= {1'b1,1'b0,1'b1,1'b0,main[2 : 0],1'b0};
                          //write device address
                        link _ head <= 0;
                        link _ write _ eeprm <= YES;
                        finish _ F <= 0;
                        sh8out _ state <= sh8out _ bit0;
                        main _ state <= Main _ dev _ write;
                     end
```

```
//----------------------------------state==2
Main_dev_write: if (finish_F==0)
                  begin
                      shift8_out;
                  end
                   else if ( finish_F == 1 )
                   begin
                      sh8out_state <= idle;
                      if(sub==0)          //write data address.
                          sh8out_buf[7:0]<=offset[3:0];
                      else
                          sh8out_buf[7:0]<=(sub+1)*8+offset[2:0];
                      main_state<=Main_addr_write;
                      finish_F <= 0;
                   end
//----------------------------------state==3
Main_addr_write: if(finish_F == 0)
                   begin    shift8_out;    end
                 else
                 if(finish_F == 1)
                   begin
                     sh8out_state <= idle;
                     sh8out_buf[7:0]<=data;
                     main_state <= Main_data_write;
                     finish_F <= 0;
                   end
//----------------------------------state==4
Main_data_write: if (finish_F == 0)
                     begin    shift8_out;    end
                   else
                   if(finish_F == 1)
                   begin
                      link_write_eeprm <= 0;
                      stop_state <= 0;
                      main_state <= Main_stop;
                      link_stop <= YES;
                      finish_F <= 0;
                   end
//----------------------------------state==8
Main_stop: if ( finish_F == 0 )
             begin    shift_stop;    end
           else
             if( finish_F == 1 )
```

```
                begin
                  ack<=YES;
                  finish_F <= 0;
                  main_state <= Main_stop_ack;
                end
//-------------------------------------------------------------//state==9
Main_stop_ack :    begin
                     ack<=NO;
                     main_state<=idle;
                     write_flag<=0;
                   end
//-------------------------------------------------------------
default :     begin
                main_state <= idle;
              end
endcase
end
endtask

//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* * * TASK OF READ DATA FROM EEPRM
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
task  Reading_From_Eeprm;

begin
casex ( main_state )
//-------------------------------------------------------------state==0
idle : begin
         link_read_eeprm <= NO;
         link_write_eeprm<= NO;
         link_sda <= YES;
         link_head <= YES;
         link_stop <= NO;
         data_from_rm <= 0;
         head_buf[1 : 0] <= 2'b10;
         head_state <= sh_head_bit1;
         sh8out_buf <= 0;
         sh8out_state <= sh8out_bit0;
         stop_buf[1 : 0] <= 2'b01;
         stop_state <= sh_stop_idle;
         finish_F <= NO;
           //sh8in_state <= 0;
         ack <= NO;
```

```
        main_state <= Main_start1;
    end
//-----------------------------------------------state==1
Main_start1 : if (finish_F == 0)
                begin  shift_head;  end
              else
                if (finish_F == 1)
                  begin
                    head_state <= idle;
                    sh8out_buf[7:0] <= {1'b1,1'b0,1'b1,1'b0,main[2:0],1'b0};
                    //write device address
                    sh8out_state <= sh8out_bit0;
                    link_head <= NO;
                    link_write_eeprm <= YES;
                    finish_F <= 0;
                    main_state <= Main_dev_write;
              end
//-----------------------------------------------state==3
Main_dev_write : if (finish_F == 0)
                   begin  shift8_out;  end
                 else
                   if (finish_F == 1)
                     begin
                        if (sub==0) //write data address
                            sh8out_buf[7:0]<=offset[3:0];
                        else
                            sh8out_buf[7:0]<=(sub+1)*8+offset[2:0];
                            sh8out_state<=sh8out_idle;
                            head_buf <= 2'b10;
                            finish_F <= 0;
                            main_state<=Main_addr_write;
                   end
//-----------------------------------------------//state==4
Main_addr_write : if (finish_F == 0)
                    begin  shift8_out;  end
                    else
                        if(finish_F == 1)
                        begin
                            head_state<=sh_head_idle;
                            finish_F <= 0;
                            main_state <= Main_start2;
                        end
```

```
//-------------------------------------------------//state==6
Main_start2 : if ( finish_F == 0 )
                begin  shift_head;  end
              else
                if ( finish_F == 1 )
                  begin
                    head_state <= idle;
                    link_head <= NO;
                    link_sda <= YES;
                    link_write_eeprm <= YES;
                    sh8out_buf[7 : 0]<={1'b1,1'b0,1'b1,1'b0,main[2 : 0],1'b1};
                    //read
                    sh8out_state <= sh8out_bit0;
                    finish_F <= 0;
                    main_state <= Main_dev_read;
                  end
//-------------------------------------------------state==7
Main_dev_read : if (finish_F == 0 )
                  begin  shift8_out;  end
                else
                  if (finish_F == 1)
                    begin
                      link_sda <= NO;
                      link_head<= NO;
                      link_write_eeprm <= NO;
                      finish_F <= 0;
                      sh8in_state<=sh8in_idle;
                      main_state <= Main_data_read;
                    end
//-------------------------------------------------state==8
Main_data_read : if ( finish_F == 0 )
                   begin  shift8in;  end
                 else
                   if (finish_F == 1)
                     begin
                       link_write_eeprm <= 0;
                       link_stop <= YES;
                       link_sda <= YES;
                       stop_state <= sh_stop_bit1;
                       finish_F <= 0;
                       main_state <= Main_stop;
                     end
//-------------------------------------------------state==9
```

```
Main _ stop : if ( finish _ F == 0 )
                begin  shift _ stop;  end
                else
                  if (finish _ F == 1)
                    begin
                      ack<=YES;
                      finish _ F <= 0;
                      main _ state <= Main _ stop _ ack;
                    end
//-------------------------------------state==10
Main _ stop _ ack : begin
                      ack<=NO;
                      read _ flag<=0;
                      main _ state <= idle;
                    end
//-------------------------------------------------
default : begin
            main _ state <= idle;
          end

endcase
end
endtask

//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* * * TASK OF SHIFT IN 8 BIT DATA THROUGH SDA BUS FROM EEPRM
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
task  shift8in;
begin

casex(sh8in _ state)
idle :        begin
                sh8in _ state <= sh8in _ bit7;
              end

sh8in _ bit7 : if ( scl )      //read bit the 8th from sda when scl =1 read
                               begin
                                 data _ from _ rm[7] <= sda;
                                 sh8in _ state <= sh8in _ bit6;
                               end
               else
                                 sh8in _ state <= sh8in _ bit7;
```

```
sh8in_bit6 : if ( scl )      //read bit the 7th
                begin
                   data_from_rm[6] <= sda;
                   sh8in_state <= sh8in_bit5;
                end
             else
                   sh8in_state <= sh8in_bit6;

sh8in_bit5 : if (scl)        //read bit the 6th
                begin
                   data_from_rm[5] <= sda;
                   sh8in_state <= sh8in_bit4;
                end
             else
                   sh8in_state <= sh8in_bit5;

sh8in_bit4 : if ( scl )      //read bit 5
                begin
                   data_from_rm[4] <= sda;
                   sh8in_state <= sh8in_bit3;
                end
             else
                   sh8in_state <= sh8in_bit4;

sh8in_bit3 : if ( scl )      //read bit 4
                begin
                   data_from_rm[3] <= sda;
                   sh8in_state <= sh8in_bit2;
                end
             else
                   sh8in_state <= sh8in_bit3;

sh8in_bit2 : if ( scl )      //read bit 3
                begin
                   data_from_rm[2] <= sda;
                   sh8in_state <= sh8in_bit1;
                end
             else
                   sh8in_state <= sh8in_bit2;

sh8in_bit1 : if ( scl )      //read bit 2
                begin
                   data_from_rm[1] <= sda;
```

```
                    sh8in_state <= sh8in_bit0;
                end
            else
                sh8in_state <= sh8in_bit1;

sh8in_bit0 : if ( scl )        //read bit 1
                begin
                    data_from_rm[0] <= sda;
                    sh8in_state <= linkbus_state;
                end
            else
                sh8in_state <= sh8in_bit0;

linkbus_state : if ( scl )      //bit the 9th
                    begin
                        sh8in_state <= sh8in_bit7;
                        link_read_eeprm <= YES;
                        finish_F <= finish_F + 1;
                    end
                else
                    sh8in_state <=linkbus_state;

default : begin
            link_read_eeprm<=NO;
            sh8in_state<=sh8in_bit7;
          end
endcase
end
endtask

//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* * * TASK OF SHIFT OUT 8 BIT DATA THROUGH SDA BUS TO EEPRM
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

task shift8_out;
begin
casex(sh8out_state)
//----------state==0
sh8out_idle : if ( ! scl )
                begin
                  link_sda <= YES;
                  link_write_eeprm <= YES;
                  sh8out_state <= sh8out_bit0;
```

```
                end
              else
                begin
                  sh8out_state <= sh8out_idle;
                end
//----------state==1
sh8out_bit0: if(!scl)
                begin
                  link_sda <= YES;
                  link_write_eeprm <=YES;
                  sh8out_state <= sh8out_bit1;
                  sh8out_buf <= sh8out_buf<<1;
                end
              else
                begin
                  sh8out_state <= sh8out_bit0;
                end
//----------state==2
sh8out_bit1: if(!scl)
                begin
                  sh8out_state <= sh8out_bit2;
                  sh8out_buf <= sh8out_buf<<1;
                end
              else
                begin
                  sh8out_state <= sh8out_bit1;
                end
//----------state==3
sh8out_bit2: if(!scl)
                begin
                  sh8out_state <= sh8out_bit3;
                  sh8out_buf <= sh8out_buf<<1;
                end
              else
                begin
                  sh8out_state <= sh8out_bit2;
                end
//----------state==4
sh8out_bit3: if(!scl)
                begin
                  sh8out_state <= sh8out_bit4;
                  sh8out_buf <= sh8out_buf<<1;
                end
```

```
                else
                  begin
                    sh8out_state <= sh8out_bit3;
                  end
//----------state==5
sh8out_bit4 : if ( !scl )
                  begin
                    sh8out_state <= sh8out_bit5;
                    sh8out_buf <= sh8out_buf<<1;
                  end
                else
                  begin
                    sh8out_state <= sh8out_bit4;
                  end
//----------state==6
sh8out_bit5 : if ( !scl )
                  begin
                    sh8out_state <= sh8out_bit6;
                    sh8out_buf <= sh8out_buf<<1;
                  end
                else
                  begin
                    sh8out_state <= sh8out_bit5;
                  end
//----------state==7
sh8out_bit6 : if ( !scl )
                  begin
                    sh8out_state <= sh8out_bit7;
                    sh8out_buf <= sh8out_buf<<1;
                  end
                else
                  begin
                    sh8out_state <= sh8out_bit6;
                  end
//----------state==8
sh8out_bit7 : if ( !scl )
                  begin
                    sh8out_state <= sh8out_ack;
                    link_sda <= NO;
                    link_write_eeprm <= NO;
                    finish_F <= finish_F + 1;
                  end
                else
```

```
                    begin
                        sh8out _ state <= sh8out _ bit7;
                    end
//------------state==9
sh8out _ ack : if ( !scl ) //may be of no use, because finish _ f have added 1
                    begin
                        link _ sda <= NO;
                        link _ write _ eeprm <= NO;
                    end
                  else
                    begin
                        sh8out _ state <=sh8out _ ack;
                    end
endcase
end
endtask

//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* * * TASK OF SHIFT OUT HEAD FLAG
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
task shift _ head;
begin
casex(head _ state)
//-------------------------------------------------state==0
sh _ head _ idle : if ( !scl )
                     begin
                        link _ write _ eeprm<= NO;
                        link _ sda <= YES;
                        link _ head <=YES;
                        head _ state<=sh _ head _ bit1;
                     end
                   else
                        head _ state<=sh _ head _ idle;
//-------------------------------------------------state==1
sh _ head _ bit1 : if ( scl ) //scl _ o initial
                     begin
                        link _ write _ eeprm<= NO;
                        link _ sda <= YES;
                        link _ head <=YES;
                        head _ buf <= head _ buf<<1;
                        head _ state <= sh _ head _ end;
                        finish _ F<=finish _ F+1;
                     end
```

```
                else
                  head_state<= sh_head_bit1;
//-----------------------------state==default
default :  begin
             link_sda<=NO;
             link_head<=0;
             head_state<=sh_head_bit1;
           end

endcase
end

endtask

//*********************************************
//*** TASK OF SHIFT OUT STOP FLAG
//*********************************************

task shift_stop;
begin
casex(stop_state)
sh_stop_idle :  if ( !scl )
                  begin
                    link_sda <= YES;
                    link_write_eeprm<=0;
                    link_stop <=YES;
                    stop_state<=sh_stop_bit1;
                  end
                else
                  stop_state<=sh_stop_idle;
sh_stop_bit1 :  if ( scl_o )
                  begin
                    stop_buf <= stop_buf<<1;
                    stop_state <= sh_stop_end;
                  end
                else
                  stop_state<= sh_stop_bit1;
sh_stop_end :  if ( !scl )
                  begin
                    link_stop <= NO;
                    stop_state <= sh_stop_idle;
                    finish_F <= finish_F + 1;
                  end
```

```
                else
                    stop_state <= sh_stop_end;

default : begin
                stop_state<=sh_stop_idle;
            end
endcase
end
endtask

endmodule
```

图 4.4 和图 4.5 分别为 EEPROM 读、写时序的仿真波形。

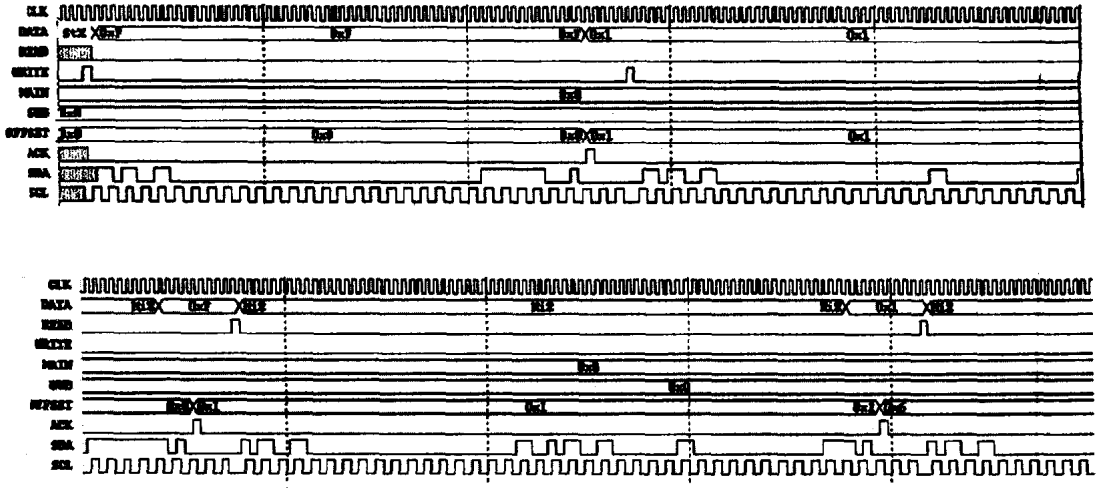


图 4.4 EEPROM 的写时序

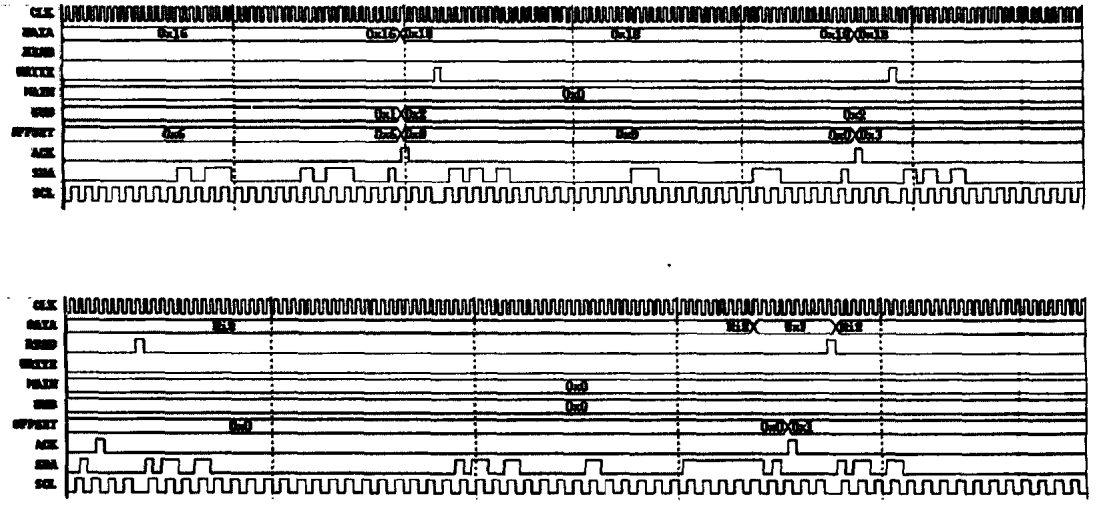


图 4.5 EEPROM 的读时序

第五章　可综合的 Verilog HDL 设计实例
——简化的 RISC _ CPU 设计简介

前四章已经介绍了 Verilog HDL 的基本语法、简单组合逻辑和简单时序逻辑模块的编写及 TOP-DOWN 设计方法，还学习了可综合风格的有限状态机的设计。其中，EEPROM 读写器的设计实质上是一个较复杂的、嵌套的有限状态机的设计，它与实际工程设计很接近，也可以说已是真实的设计。

在这一章里将通过一个经过简化的、用于教学目的的 RISC _ CPU 的设计过程来说明可综合的 Verilog HDL 新设计方法的潜力。这个模型实质上是第三章的 RISC _ CPU 模型的改进。第三章的 RISC _ CPU 模型是一个顶层模型，它关心的只是总体设计的合理性，它的模块中有许多是不可综合的，只可以进行仿真。而本章中构成 RISC _ CPU 的每一个模块不仅是可仿真的也都是可综合的，因为它们符合可综合风格的要求。为了能在这个虚拟的 CPU 上运行较为复杂的程序并进行仿真，因而把寻址空间扩大到 8K(即 15 位地址线)。下面让我们一步一步地来设计这样一个 CPU，并进行仿真和综合，从中可以体会到这种设计方法的魅力。本章的全部程序在 Cadence 公司的 LWB (Logic Work Bench)环境下通过运行测试，并用 Synergy 综合器进行了综合，用 XACT(Xilinx 公司的布局布线工具)在 Xilinx3098 实现了布线。

5.1　什么是 CPU

CPU 即中央处理单元的英文缩写，它是计算机的核心部件。计算机进行信息处理可分为两个步骤：

(1) 将数据和程序(即指令序列)输入到计算机的存储器中；

(2) 从第一条指令的地址起开始执行该程序，得到所需结果，结束运行。

CPU 的作用是协调并控制计算机的各个部件执行程序的指令序列，使其有条不紊地进行。因此它必须具有以下基本功能：

(1) 取指令——当程序已在存储器中时，首先根据程序入口地址取出一条程序，为此要发出指令地址及控制信号。

(2) 分析指令——即指令译码，是对当前取得的指令进行分析，指出它要求什么操作，并产生相应的操作控制命令。

(3) 执行指令——根据分析指令时产生的“操作命令”形成相应的操作控制信号序列，通过运算器、存储器及输入/输出设备的执行，实现每条指令的功能，其中包括对运算结果的处理以及下条指令地址的形成。

将 CPU 的功能进一步细化，可概括如下：

(1) 能对指令进行译码并执行规定的动作；

(2) 可以进行算术和逻辑运算；

(3) 能与存储器及外设交换数据；

(4) 提供整个系统所需要的控制。

尽管各种 CPU 的性能指标和结构细节各不相同,但它们所能完成的基本功能相同。由功能分析可知,任何一种 CPU 内部结构至少应包含下面这些部件:

(1) 算术逻辑运算部件(ALU);

(2) 累加器;

(3) 程序计数器;

(4) 指令寄存器、译码器;

(5) 时序和控制部件。

RISC 即精简指令集计算机(Reduced Instruction Set Computer)的缩写。它是一种 80 年代才出现的 CPU,与一般的 CPU 相比不仅只是简化了指令系统,而且通过简化指令系统使计算机的结构更加简单合理,从而提高了运算速度。从实现的途径看,RISC _ CPU 与一般的 CPU 的不同之处在于:它的时序控制信号形成部件是用硬布线逻辑实现的,而不是采用微程序控制的方式。所谓硬布线逻辑也就是将触发器和逻辑门直接连线所构成的状态机和组合逻辑,故产生控制序列的速度比用微程序控制方式快得多,因为这样做省去了读取微指令的时间。RISC _ CPU 也包括上述这些部件,下面就详细介绍一个简化的 RISC _ CPU 的可综合 Verilog HDL 模型的设计和仿真过程。

5.2 RISC _ CPU 的结构

RISC _ CPU 是一个复杂的数字逻辑电路,但是它的基本部件的逻辑并不复杂。从第三章我们知道可把它分成 8 个基本部件:

(1) 时钟发生器;

(2) 指令寄存器;

(3) 累加器;

(4) RISC _ CPU 算术逻辑运算单元;

(5) 数据控制器;

(6) 状态控制器;

(7) 程序计数器;

(8) 地址多路器。

各部件的相互连接关系见图 5.1。其中,时钟发生器利用外来时钟信号进行分频生成一系列时钟信号,送往其他部件用作时钟信号。各部件之间的相互操作关系则由状态控制器来控制。各部件的具体结构和逻辑关系在下面各节中一一进行介绍。

5.2.1 时钟发生器

时钟发生器(如图 5.2 所示)clkgen 利用外来时钟信号 clk 生成时钟信号 clk1,fetch,alu _ clk,送往 CPU 的其他部件。其中,fetch 是外部时钟 clk 的 8 分频信号,利用 fetch 的上升沿来触发 CPU 控制器开始执行一条指令,同时 fetch 信号还将控制地址多路器输出指令地址和数据地址;clk1 信号用作指令寄存器、累加器、状态控制器的时钟信号;alu _ clk 则用于触发算术逻辑运算单元。

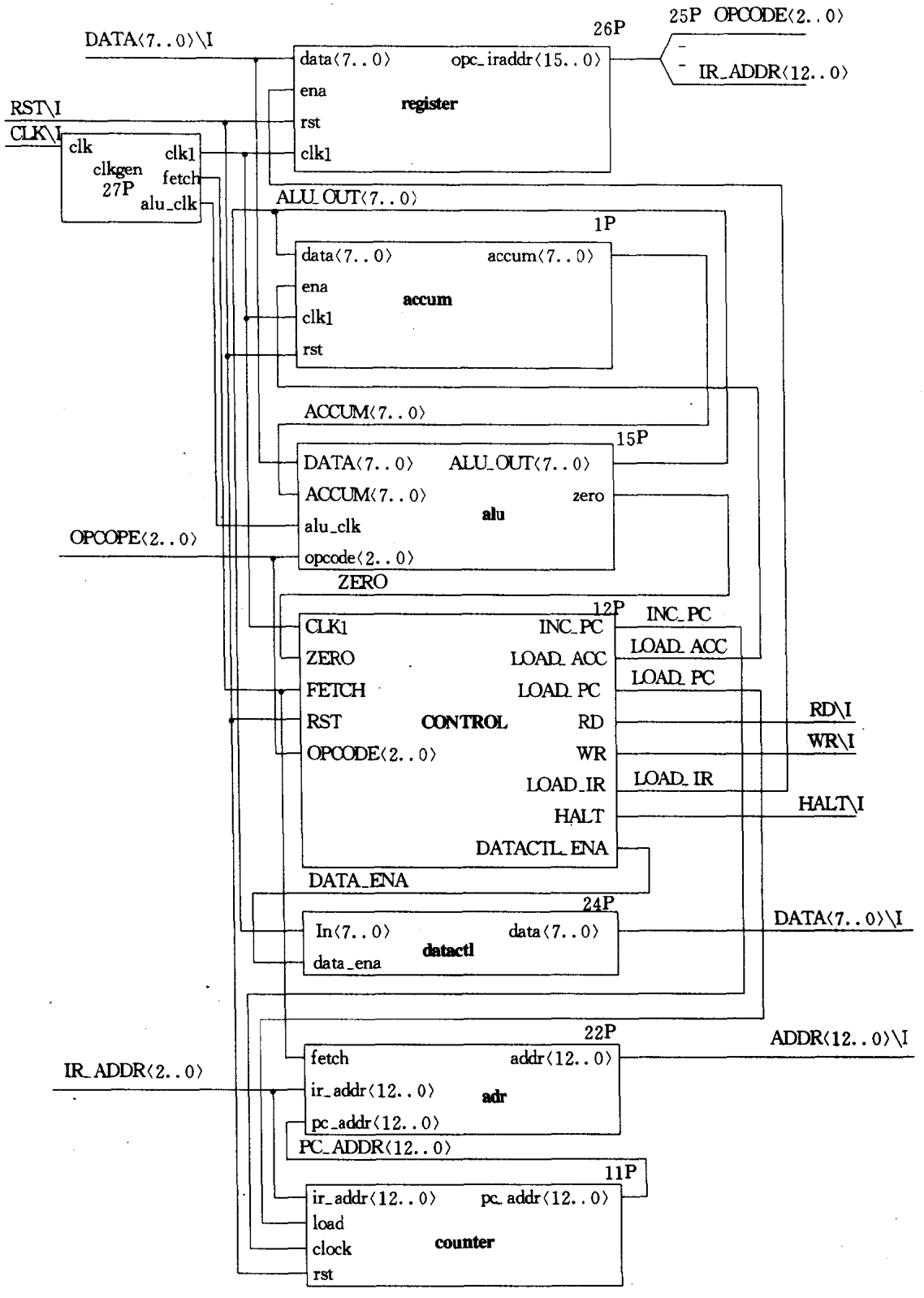


图 5.1　RISC_CPU中各部件的相互连接关系

时钟发生器 clkgen 的波形如图 5.3 所示。

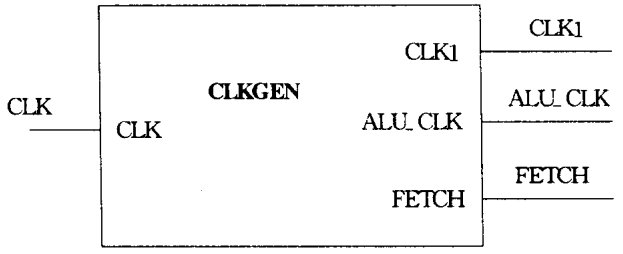


图 5.2 时钟发生器

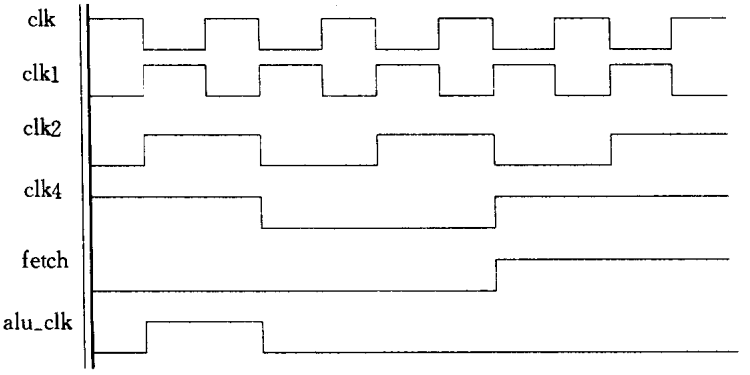


图 5.3 时钟发生器的波形图

其 Verilog HDL 程序见下面的模块：

```
'timescale  1ns/100ps
module  clkgen(clk,clk1,fetch,alu _clk);
output  clk1,fetch,alu _clk;
input  clk;
reg  clk2,clk4,fetch;

initial
  begin
     clk2 = 0;
     clk4 = 1;
     fetch = 0;
  end

assign  clk1=~ clk;
assign  alu _clk = clk2 & clk4 & !fetch;

   always @( posedge clk1 )
   begin
      clk2 <=~ clk2;
   end
```

```
always @(negedge clk2)
   begin
      clk4<=~ clk4;
   end

always @(posedge clk4)
    begin
       fetch<=~ fetch;
    end
endmodule
```

5.2.2　指令寄存器

顾名思义，指令寄存器(如图 5.4 所示)用于寄存指令。

指令寄存器的触发时钟是 clk1，在 clk1 的正沿触发下，寄存器将数据总线送来的指令存

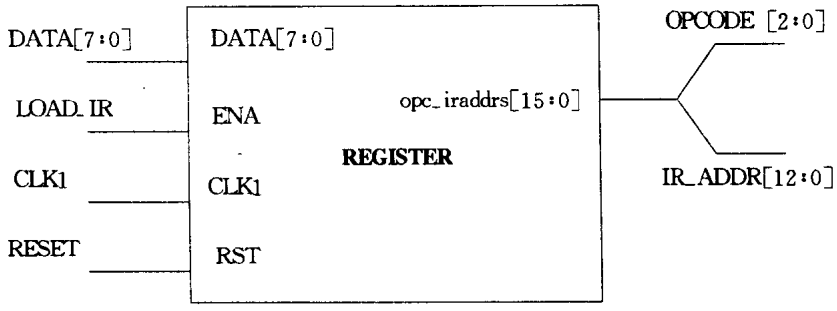


图 5.4　指令寄存器

入高 8 位或低 8 位寄存器中，但并不是每个 clk1 的上升沿都寄存数据总线的数据，因为数据总线上有时传输指令，有时传输数据。什么时候寄存，什么时候不寄存由 CPU 状态控制器的 load _ ir 信号控制。load _ ir 信号通过 ena 口输入到指令寄存器，复位后，指令寄存器被清为零。

每条指令为两个字节，即 16 位。高 3 位是操作码，低 13 位是地址(CPU 的地址总线为 13 位，寻址空间为 8K 字节)。本设计的数据总线为 8 位，所以每条指令需取两次。先取高 8 位，后取低 8 位。而当前取的是高 8 位还是低 8 位，由变量 state 记录。state 为 0 表示取的是高 8 位，存入高 8 位寄存器，同时将变量 state 置为 1。下次再寄存时，由于 state 为 1，可知取的是低 8 位，存入低 8 位寄存器中。

其 Verilog HDL 程序见下面的模块：

```
module  register(opc _ iraddr,data,ena,clk1,rst);
output [15 : 0]  opc _ iraddr;
input [7 : 0]  data;
input  ena, clk1, rst;
reg [15 : 0]  opc _ iraddr;
reg  state;

always @(posedge clk1)
```

```
begin
   if(rst)
     begin
        opc _ iraddr<=16'b0000 _ 0000 _ 0000 _ 0000;
        state<=1'b0;
     end
   else
     begin
        if (ena) //如果加载指令寄存器信号 load _ ir 到来,
          begin //分两个时钟每次 8 位加载指令寄存器
             casex(state) //先高字节,后低字节
               0 : begin
                     opc _ iraddr[15 : 8]<=data;
                     state<=1;
                   end
               1 : begin
                     opc _ iraddr[7 : 0]<=data;
                     state<=0;
                   end
               default : begin
                           opc _ iraddr[15 : 0]<=16'bxxxxxxxxxxxxxxxx;
                           state<=1'bx;
                         end
             endcase
          end
        elsde
          state<=1'b0;
     end
end
endmodule
```

5.2.3 累加器

累加器(如图 5.5 所示)用于存放当前的结果,它也是双目运算中一个数据来源。复位后,累加器的值是零。当累加器通过 ena 口收到来自 CPU 状态控制器 load _ acc 信号时,在 clk1 时钟正跳沿时就收到来自于数据总线的数据。

其 Verilog HDL 程序见下面的模块:

```
module  accum( accum, data, ena, clk1, rst);
   output[7 : 0]  accum;
   input[7 : 0]  data;
   input  ena,clk1,rst;
   reg[7 : 0]  accum;
```

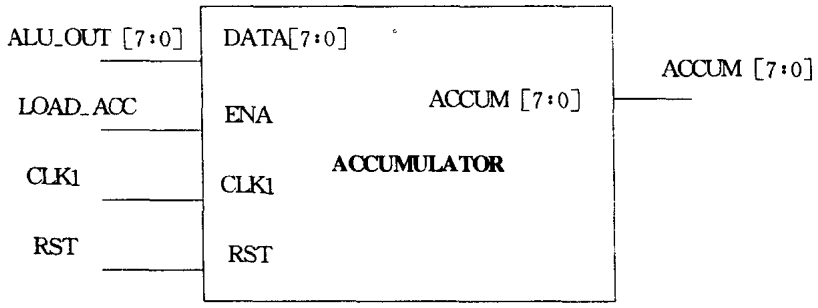


图 5.5 累加器

```
always@(posedge clk1)
   begin
      if(rst)
        accum<=8'b0000_0000;        //Reset
      else
        if(ena)        //当 CPU 状态控制器发出 load_acc 信号
          accum<=data;        //Accumulate
   end
endmodule
```

5.2.4 算术运算器

算术逻辑运算单元(如图 5.6 所示)根据输入的 8 种不同操作码分别实现相应的加、与、异或、跳转等 8 种基本操作运算。利用这几种基本运算可以实现很多种其他运算以及逻辑判断等操作。

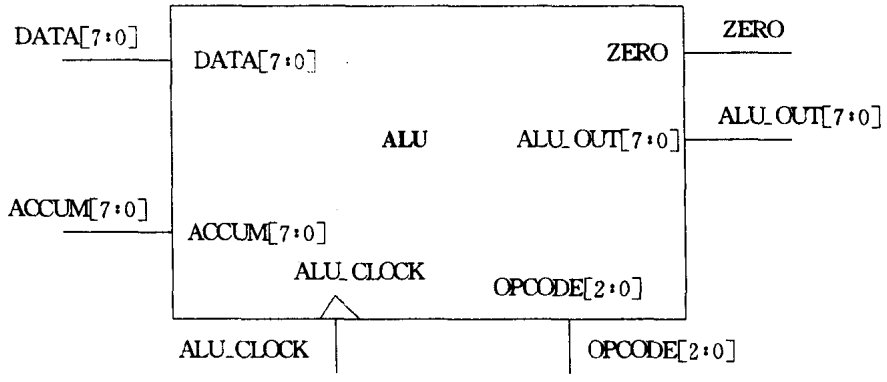


图 5.6 算术运算器

其 Verilog HDL 程序见下面的模块:

```
module  alu (alu_out, zero, data, accum, alu_clk, opcode);
output [7:0]  alu_out;
output  zero;
input [7:0]  data, accum;
input [2:0]  opcode;
input  alu_clk;
```

```
reg [7:0] alu_out;

  parameter   HLT=0,
              SKZ=1,
              ADD=2,
              ANDD=3,
              XORR=4,
              LDA=5,
              STO=6,
              JMP=7;

  assign  zero = !accum;
  always @(posedgealu_clk)
    begin //操作码来自指令寄存器的输出 opc_iaddr<15..0>的低 3 位
      casex (opcode)
        HLT : alu_out<=accum;
        SKZ : alu_out<=accum;
        ADD : alu_out<=data+accum;
        ANDD : alu_out<=data & accum;
        XORR : alu_out<=data ^ accum;
        LDA : alu_out<=data;
        STO : alu_out<=accum;
        JMP : alu_out<=accum;
        default : alu_out<=8'bxxxx_xxxx;
      endcase
    end
endmodule
```

5.2.5 数据控制器

数据控制器(如图 5.7 所示)的作用是控制累加器数据输出,由于数据总线是各种操作时传送数据的公共通道,不同的情况下传送不同的内容。有时要传输指令,有时要传送 RAM 区

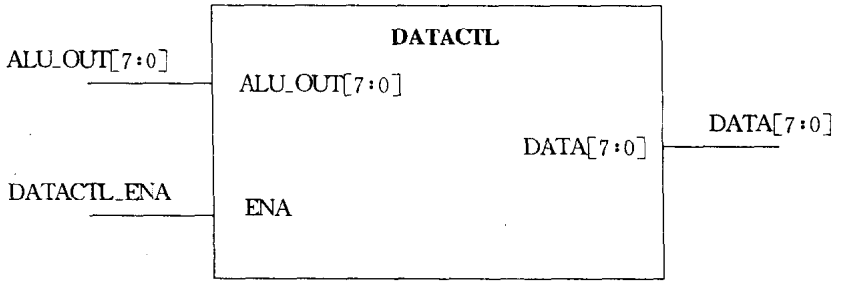


图 5.7 数据控制器

或接口的数据。累加器的数据只有在需要往 RAM 区或端口写时才允许输出,否则应呈现高阻态,以允许其他部件使用数据总线。所以任何部件往总线上输出数据时,都需要一控制信号。

而此控制信号的启、停则由 CPU 状态控制器输出的各信号控制决定。数据控制器何时输出累加器的数据则由状态控制器输出的控制信号 datactl _ ena 决定。

其 Verilog HDL 程序见下面的模块：

```
module   datactl (data,in,data _ ena);
    output [7 : 0]  data;
    input [7 : 0]  in;
    input   data _ ena;

    assign   data = (data _ ena)? in  :  8'bzzzz _ zzzz;

endmodule
```

5.2.6　地址多路器

地址多路器(如图 5.8 所示)用于选择输出的地址是 PC(程序计数)地址还是数据/端口地址。每个指令周期的前 4 个时钟周期用于从 ROM 中读取指令，输出的应是 PC 地址；后 4 个时钟周期用于对 RAM 或端口的读写，该地址由指令给出。地址的选择输出信号由时钟信号的 8 分频信号 fetch 提供。

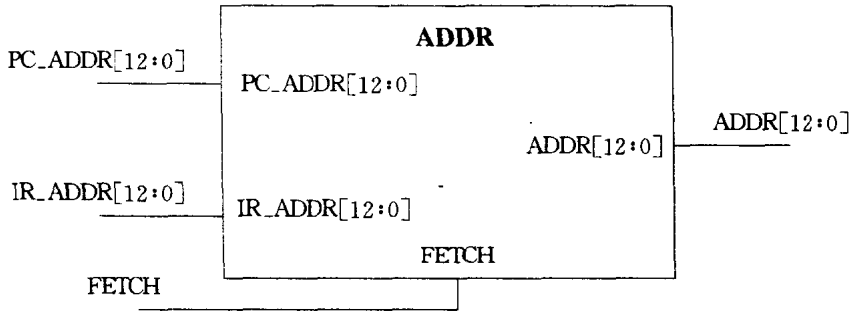


图 5.8　地址多路器

其 Verilog HDL 程序见下面的模块：

```
module   adr(addr,fetch,ir _ addr,pc _ addr);
output [12 : 0]  addr;
input [12 : 0]  ir _ addr, pc _ addr;
input   fetch;

assign   addr = fetch? pc _ addr  :  ir _ addr;

endmodule
```

5.2.7　程序计数器

程序计数器(如图 5.9 所示)用于提供指令地址，以便读取指令。指令按地址顺序存放在存储器中。有两种途径可形成指令地址：其一是顺序执行的情况，其二是遇到要改变顺序执行程序的情况，例如执行 JMP 指令后，需要形成新的指令地址。下面就来详细说明 PC 地址是如何

建立的。

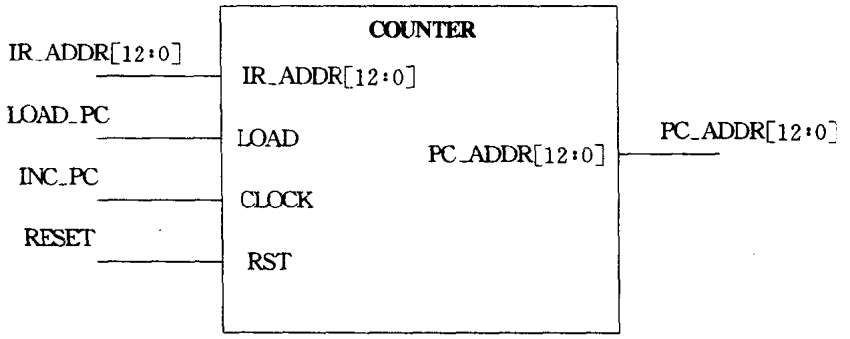


图 5.9 程序计数器

复位后，指令指针为零，即每次 CPU 重新启动将从 ROM 的零地址开始读取指令并执行。每条指令执行完需两个时钟，这时 pc _ addr 已被增 2，指向下一条指令（因为每条指令占两个字节）。如果正执行的指令是跳转语句，这时 CPU 状态控制器将会输出 load _ pc 信号，通过 load 口进入程序计数器。程序计数器(pc _ addr)将装入目标地址(ir _ addr)，而不是增 2。

其 Verilog HDL 程序见下面的模块：

```
module  counter ( pc _addr, ir _addr, load, clock, rst);
    output [12 : 0]  pc _addr;
    input [12 : 0]  ir _addr;
    input  load, clock, rst;
    reg [12 : 0]  pc _addr;

    always @( posedge clock or posedge rst )
       begin
          if(rst)
            pc _addr<=13'b0 _0000 _0000 _0000;
          else
            if(load)
              pc _addr<=ir _addr;
            else
              pc _addr<=pc _addr + 1;
       end
endmodule
```

5.2.8 状态控制器

状态控制器（如图 5.10 所示）由两部分组成：

(1) 状态机（图 5.10 中的 MACHINE 部分）；

(2) 状态机控制器（图 5.10 中的 MACHINECTL 部分）。

状态机控制器接受复位信号 RST，当 RST 有效时通过信号 ena 使其为 0，输入到状态机中停止状态机的工作。

状态控制器的 Verilog HDL 程序见下面模块：

```
module  machinectl( ena, fetch, rst);
```

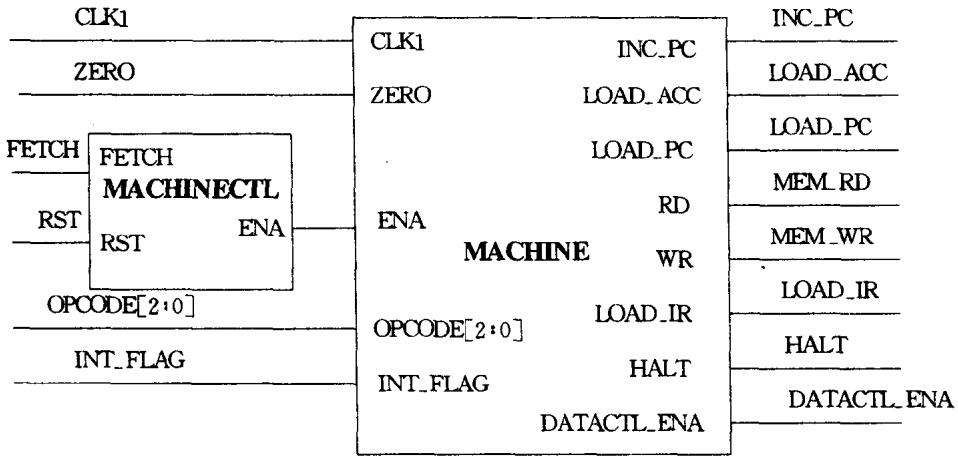


图 5.10 状态控制器

```
    output   ena;
    input   fetch, rst;
reg   ena;

     always @(posedge fetch or posedge rst)
        begin
           if(rst)
             ena<=0;
           else
             ena<=1;
        end
    endmodule
```

状态机是 CPU 的控制核心,用于产生一系列的控制信号,启动或停止某些部件。CPU 何时进行读指令读写 I/O 端口、RAM 区等操作,都是由状态机来控制的。状态机的当前状态,由变量 state 记录,state 的值就是当前这个指令周期中经过的时钟数(从零计起)。

指令周期由 8 个时钟周期组成,每个时钟周期都要完成固定的操作。

第 0 个时钟,因为 CPU 状态控制器的输出 rd 和 load _ir 为高电平,其余均为低电平。指令寄存器寄存由 ROM 送来的高 8 位指令代码。

第 1 个时钟,与上一时钟相比只是 inc _pc 从 0 变为 1,故 PC 增 1,ROM 送来低 8 位指令代码,指令寄存器寄存该 8 位代码。

第 2 个时钟,空操作。

第 3 个时钟,PC 增 1,指向下一条指令。若操作符为 HLT,则输出信号 HLT 为高;如果操作符不为 HLT,除了 PC 增 1 外(指向下一条指令),其他各控制线输出为零。

第 4 个时钟,若操作符为 AND,ADD,XOR 或 LDA,读相应地址的数据;若为 JMP,将目的地址送给程序计数器;若为 STO,输出累加器数据。

第 5 个时钟,若操作符为 ANDD,ADD 或 XORR,算术运算器就进行相应的运算;若为 LDA,就把数据通过算术运算器送给累加器;若为 SKZ,先判断累加器的值是否为 0,如果为 0,PC 就增 1,否则保持原值;若为 JMP,锁存目的地址;若为 STO,将数据写入地址处。

第 6 个时钟，空操作。

第 7 个时钟，若操作符为 SKZ 且累加器值为 0，则 PC 值再增 1，跳过一条指令，否则 PC 无变化。

状态机的 Verilog HDL 程序见下面模块：

```
module  machine( inc_pc, load_acc, load_pc, rd,wr, load_ir,
                 datactl_ena, halt, clk1, zero, ena, opcode );
output  inc_pc, load_acc, load_pc, rd, wr, load_ir;
output  datactl_ena, halt;
input  clk1, zero, ena;
input [2:0]  opcode;
reg  inc_pc, load_acc, load_pc, rd, wr, load_ir;
reg datactl_ena, halt;
reg [2:0]  state;

parameter  HLT = 3'b000,
           SKZ = 3'b001,
           ADD = 3'b010,
           ANDD = 3'b011,
           XORR = 3'b100,
           LDA = 3'b101,
           STO = 3'b110,
           JMP = 3'b111;

  always @( negedge clk1 )
    begin
      if ( !ena )       //接收到复位信号 RST,进行复位操作
        begin
          state<=3'b000;
          {inc_pc,load_acc,load_pc,rd}<=4'b0000;
          {wr,load_ir,datactl_ena,halt}<=4'b0000;
        end
      else
        ctl_cycle;
    end
//-------------begin of task ctl_cycle-------
task  ctl_cycle;
begin
casex(state)
0:      //load high 8bits in struction
  begin
    {inc_pc,load_acc,load_pc,rd}<=4'b0001;
    {wr,load_ir,datactl_ena,halt}<=4'b0100;
```

```
        state<=1;
    end
1 :     //pc increased by one then load low 8bits instruction
    begin
        {inc _ pc,load _ acc,load _ pc,rd}<=4'b1001;
        {wr,load _ ir,datactl _ ena,halt}<=4'b0100;
        state<=2;
    end
2 :     //idle
    begin
        {inc _ pc,load _ acc,load _ pc,rd}<=4'b0000;
        {wr,load _ ir,datactl _ ena,halt}<=4'b0000;
        state<=3;
    end
3 :     //next instruction address setup 分析指令从这里开始
    begin
        if(opcode==HLT) //指令为暂停 HLT
        begin
        {inc _ pc,load _ acc,load _ pc,rd}<=4'b1000;
        {wr,load _ ir,datactl _ ena,halt}<=4'b0001;
        end
        else
          begin
              {inc _ pc,load _ acc,load _ pc,rd}<=4'b1000;
              {wr,load _ ir,datactl _ ena,halt}<=4'b0000;
          end
        state<=4;
    end
4 :     //fetch oprand
    begin
        if(opcode==JMP)
          begin
              {inc _ pc,load _ acc,load _ pc,rd}<=4'b0010;
              {wr,load _ ir,datactl _ ena,halt}<=4'b0000;
          end
        else
          if( opcode==ADD || opcode==ANDD ||
              opcode==XORR || opcode==LDA)
              begin
                  {inc _ pc,load _ acc,load _ pc,rd}<=4'b0001;
                  {wr,load _ ir,datactl _ ena,halt}<=4'b0000;
              end
          else
```

```
        if (opcode==STO)
          begin
            {inc_pc,load_acc,load_pc,rd}<=4'b0000;
            {wr,load_ir,datactl_ena,halt}<=4'b0010;
          end
        else
          begin
            {inc_pc,load_acc,load_pc,rd}<=4'b0000;
            {wr,load_ir,datactl_ena,halt}<=4'b0000;
          end
       state<=5;
    end
5 :     //operation
    begin
      if ( opcode==ADD||opcode==ANDD||
          opcode==XORR||opcode==LDA )
        begin       //过一个时钟后与累加器的内容进行运算
          {inc_pc,load_acc,load_pc,rd}<=4'b0101;
          {wr,load_ir,datactl_ena,halt}<=4'b0000;
        end
      else
        if ( opcode==SKZ && zero==1)
          begin
            {inc_pc,load_acc,load_pc,rd}<=4'b1000;
            {wr,load_ir,datactl_ena,halt}<=4'b0000;
          end
        else
          if (opcode==JMP)
            begin
              {inc_pc,load_acc,load_pc,rd}<=4'b1010;
              {wr,load_ir,datactl_ena,halt}<=4'b0000;
            end
          else
            if(opcode==STO)
            begin
            //过一个时钟后把 wr 变 1 就可写到 RAM 中
            {inc_pc,load_acc,load_pc,rd}<=4'b0000;
            {wr,load_ir,datactl_ena,halt}<=4'b1010;
            end
            else
              begin
                {inc_pc,load_acc,load_pc,rd}<=4'b0000;
                {wr,load_ir,datactl_ena,halt}<=4'b0000;
```

```
                end
            state<=6;
        end
    6 :     //idle
        begin
            if ( opcode==STO )
                begin
                    {inc _ pc,load _ acc,load _ pc,rd}<=4'b0000;
                    {wr,load _ ir,datactl _ ena,halt}<=4'b0010;
                end
            else
                if ( opcode==ADD||opcode==ANDD||
                     opcode==XORR||opcode==LDA)
                    begin
                        {inc _ pc,load _ acc,load _ pc,rd}<=4'b0001;
                        {wr,load _ ir,datactl _ ena,halt}<=4'b0000;
                    end
                else
                    begin
                        {inc _ pc,load _ acc,load _ pc,rd}<=4'b0000;
                        {wr,load _ ir,datactl _ ena,halt}<=4'b0000;
                    end
            state<=7;
        end
    7 :     //
        begin
            if( opcode==SKZ && zero==1 )
                begin
                    {inc _ pc,load _ acc,load _ pc,rd}<=4'b1000;
                    {wr,load _ ir,datactl _ ena,halt}<=4'b0000;
                end
            else
                begin
                    {inc _ pc,load _ acc,load _ pc,rd}<=4'b0000;
                    {wr,load _ ir,datactl _ ena,halt}<=4'b0000;
                end
            state<=0;
        end
    default :
        begin
            {inc _ pc,load _ acc,load _ pc,rd}<=4'b0000;
            {wr,load _ ir,datactl _ ena,halt}<=4'b0000;
            state<=0;
```

```
    end
   endcase
   end
endtask
//————————————————end of task ctl_cycle———————
endmodule
```

状态机和状态机控制器组成了状态控制器，它们之间的连接关系很简单，如图 5.10 所示。

5.2.9 外围模块

为了对 RISC_CPU 进行测试，需要有存储测试程序的 ROM 和装载数据的 RAM 及地址译码器。下面来简单介绍一下。

1. 地址译码器

```
module  addr_decode( addr, rom_sel, ram_sel);
    output  rom_sel, ram_sel;
    input [12:0]  addr;
    reg  rom_sel, ram_sel;
    always @( addr )
    begin
       casex(addr)
          13'b1_1xxx_xxxx_xxxx : {rom_sel,ram_sel}<=2'b01;
          13'b0_xxxx_xxxx_xxxx : {rom_sel,ram_sel}<=2'b10;
          13'b1_0xxx_xxxx_xxxx : {rom_sel,ram_sel}<=2'b10;
          default : {rom_sel,ram_sel}<=2'b00;
       endcase
    end
endmodule
```

地址译码器用于产生选通信号，选通 ROM 或 RAM。

FFFFH———1800H RAM

1800H———0000H ROM

2. RAM 和 ROM

```
module  ram( data, addr, ena, read, write );
    inout [7:0]  data;
    input [9:0]  addr;
    input  ena;
    input  read, write;
    reg [7:0]  ram [10'h3ff:0];

    assign  data = ( read && ena )? ram[addr] : 8'hzz;

    always @(posedge write)
       begin
          ram[addr]<=data;
```

```
        end
endmodule

module  rom( data, addr, read, ena );
    output [7 : 0]  data;
    input [12 : 0]  addr;
    input  read, ena;
    reg [7 : 0]  memory [13'h1fff : 0];
    wire [7 : 0]  data;

    assign  data= ( read && ena )? memory[addr] : 8'bzzzzzzzz;
endmodule
```

ROM 用于装载测试程序，可读不可写。RAM 用于存放数据，可读可写。

5.3　RISC _ CPU 的操作和时序

一个微机系统为了完成自身的功能，需要 CPU 执行许多操作。以下是 RISC _ CPU 的主要操作：

(1) 系统的复位和启动操作；

(2) 总线读操作；

(3) 总线写操作。

下面详细介绍每个操作。

5.3.1　系统的复位和启动操作

RISC _ CPU 的复位和启动操作是通过 rst 引脚的信号触发执行的。当 rst 信号一进入高电平，RISC _ CPU 就会结束现行操作，并且只要 rst 停留在高电平状态，CPU 就维持在复位状态。在复位状态，CPU 各内部寄存器都被设为初值，全部为零。数据总线为高阻态，地址总线为 0000H，所有控制信号均为无效状态。rst 回到低电平后，接着到来的第一个 fetch 上升沿将启动 RISC _ CPU 开始工作，从 ROM 的 0000H 处开始读取指令并执行相应操作。波形图如图 5.11所示，虚线标志处为 RISC _ CPU 启动工作的时刻。

5.3.2　总线读操作

每个指令周期的前 0～3 个时钟周期用于读指令，在 5.2.8 节中已详细讲述，这里就不再重复；第 3.5 个周期处，存储器或端口地址就输出到地址总线上；第 4～6 个时钟周期，读信号 rd 有效，数据送到数据总线上，以备累加器锁存，或参与算术、逻辑运算；第 7 个时钟周期，读信号无效；第 7.5 个周期，地址总线输出 PC 地址，为下一个指令做好准备。图 5.12 是 CPU 从存储器或端口读取数据的时序。

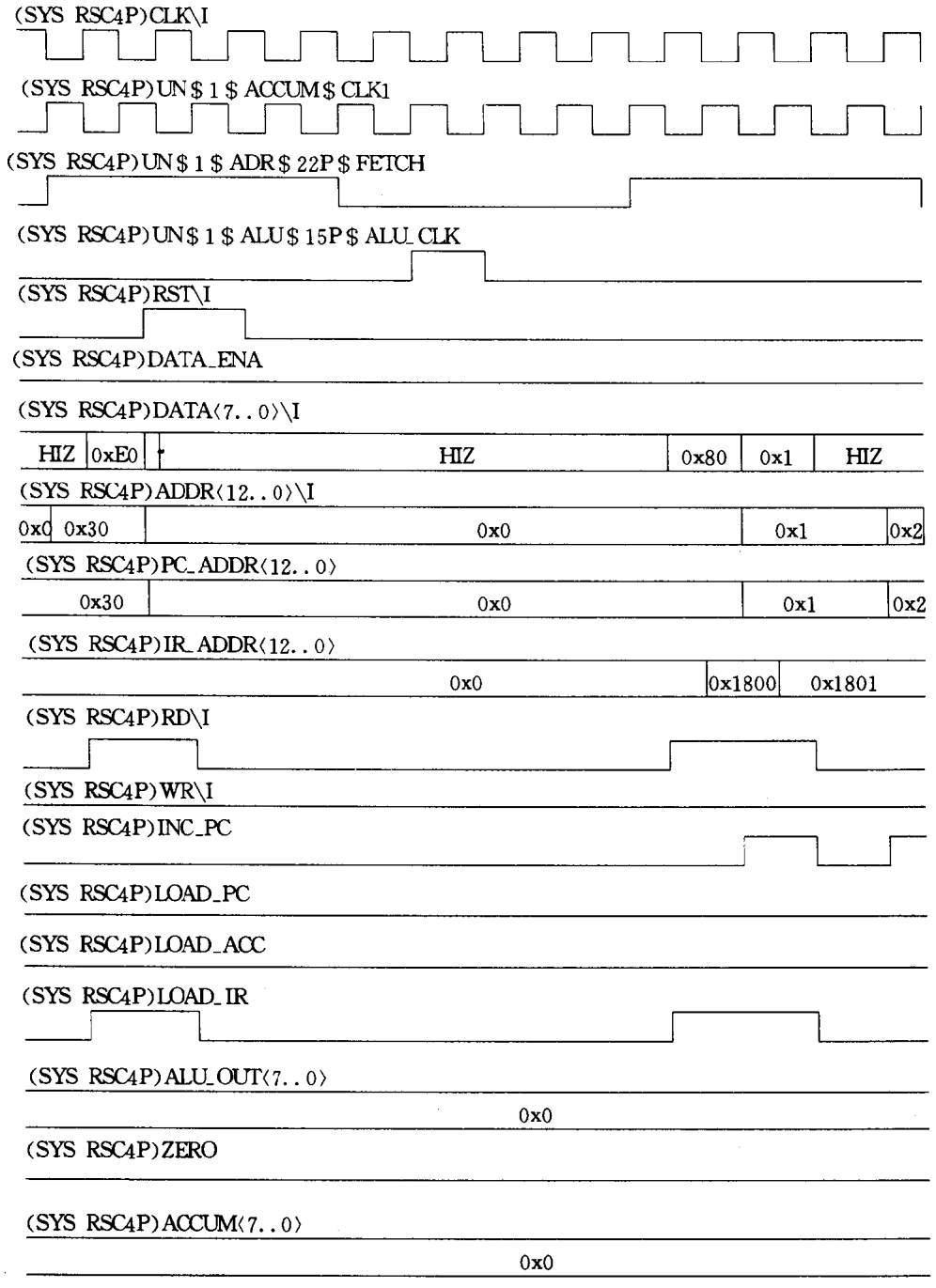


图 5.11 RISC_CPU的复位和启动操作波形

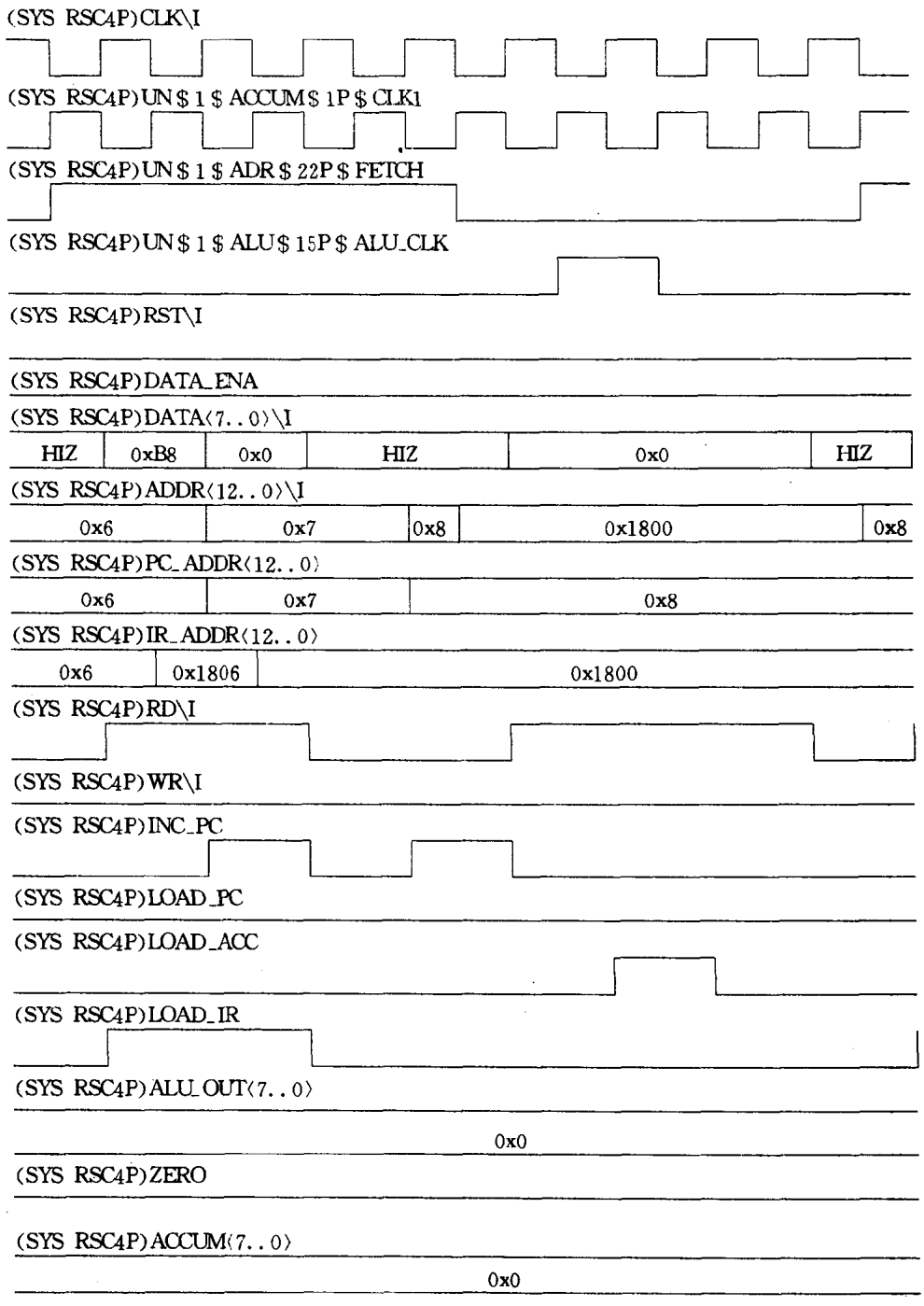


图 5.12　CPU从存储器或端口读取数据的时序

5.3.3　写总线操作

每个指令周期的第 3.5 个时钟周期处，写的地址就建立了；第 4 个时钟周期输出数据；第 5 个时钟周期输出写信号；至第 6 个时钟结束，数据无效；第 7.5 时钟地址输出为 PC 地址，为

下一个指令周期做好准备。图 5.13 是 CPU 对存储器或端口写数据的时序。

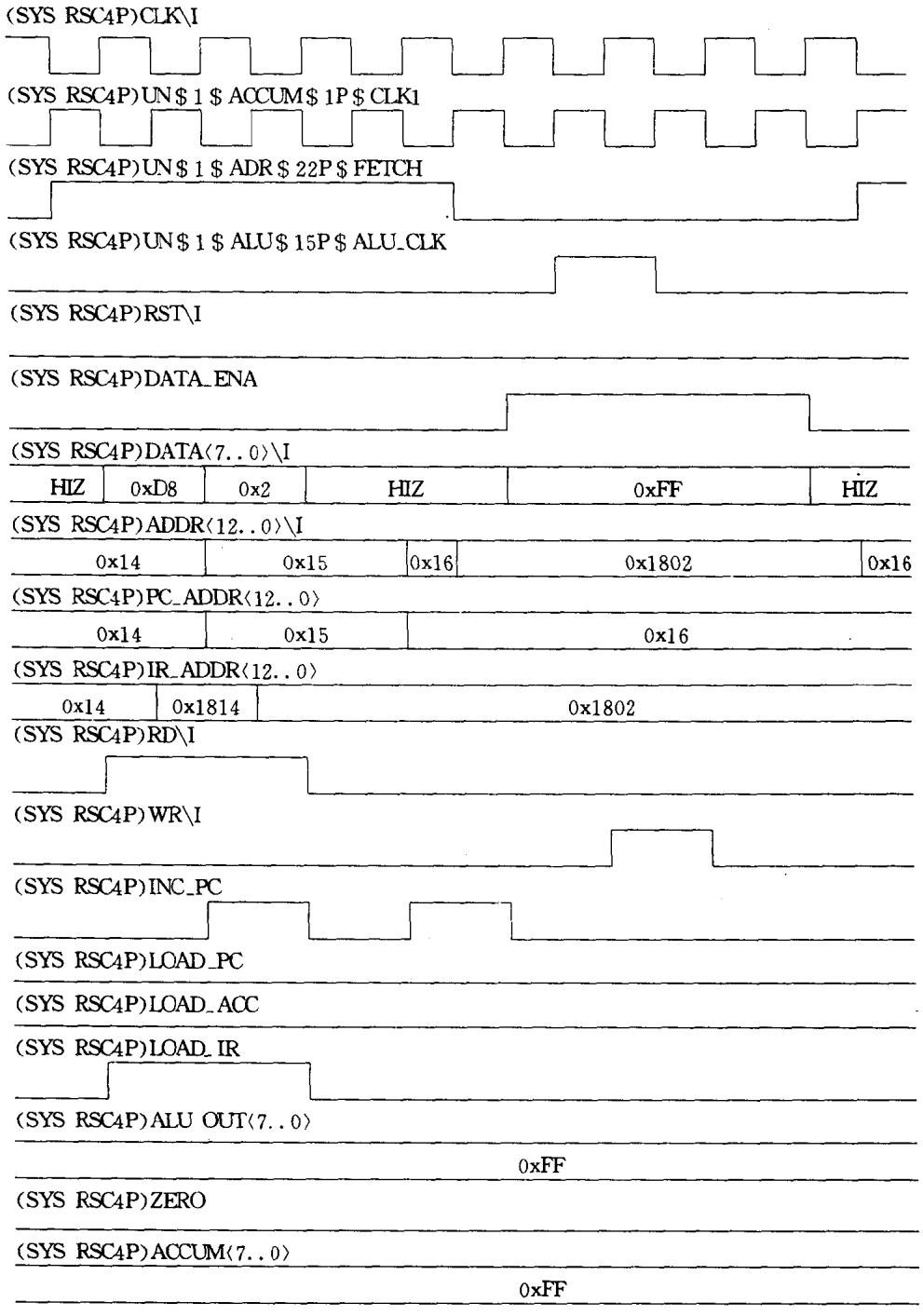
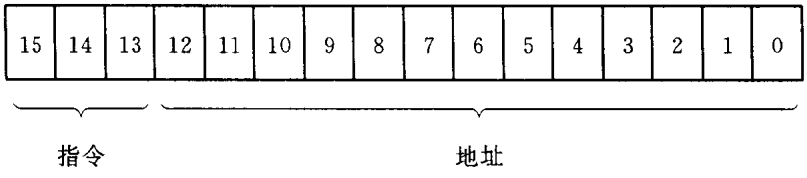


图 5.13 CPU对存储器或端口写数据的时序

5.4　RISC _ CPU 的寻址方式和指令系统

RISC _ CPU 的指令格式一律为：



它的指令系统仅由 8 条指令组成。

(1) HLT——停机操作。该操作将空一个指令周期，即 8 个时钟周期。

(2) SKZ——若为零则跳过下一条语句。该操作先判断当前 alu 中的结果是否为零，若是零就跳过下一条语句，否则继续执行。

(3) ADD——相加。该操作将累加器中的值与地址所指的存储器或端口的数据相加，结果仍送回累加器中。

(4) AND——相与。该操作将累加器的值与地址所指的存储器或端口的数据相与，结果仍送回累加器中。

(5) XOR——异或。该操作将累加器的值与指令中给出地址的数据异或，结果仍送回累加器中。

(6) LDA——读数据。该操作将指令中所给地址的数据放入累加器。

(7) STO——写数据。该操作将累加器的数据放入指令中所给的地址。

(8) JMP——无条件跳转语句。该操作将跳转至指令给出的目的地址，继续执行。

RISC _ CPU 是 8 位微处理器，一律采用直接寻址方式，即数据总是放在存储器中，寻址单元的地址由指令直接给出。这是最简单的寻址方式。

5.5　RISC _ CPU 模块的调试

5.5.1　RISC _ CPU 模块的前仿真

为了对所设计的 RISC _ CPU 模型进行验证，需要把 RISC _ CPU 包装在一个模块下，这样其内部连线就隐蔽起来，从系统的角度看显得简洁。另外，还需要建立一些必要的外围器件模型，例如储存程序用的 ROM 模型、储存数据用的 RAM 和地址译码器等。这些模型都可以用 Verilog HDL 描述。由于不需要综合成具体的电路，只要保证功能和接口信号正确就能用于仿真。也就是说，用虚拟器件来代替真实的器件对所设计的 RISC _ CPU 模型进行验证，检查各条指令是否执行正确，与外围电路的数据交换是否正常。这种模块是很容易编写的，5.2.9 节中的 ROM 和 RAM 模块就是简单虚拟器件的例子，可在下面的仿真中来代替真实的器件，用于验证 RISC _ CPU 模型是否能正确地运行装入 ROM 和 RAM 的程序。在 RISC _ CPU 的电路图上加上这些外围电路把有关的电路接通，如图 5.14 所示；也可以用 Verilog HDL 模块调用的方法把这些外围电路的模块连接上，这跟用真实的电路器件调试情况很类似。

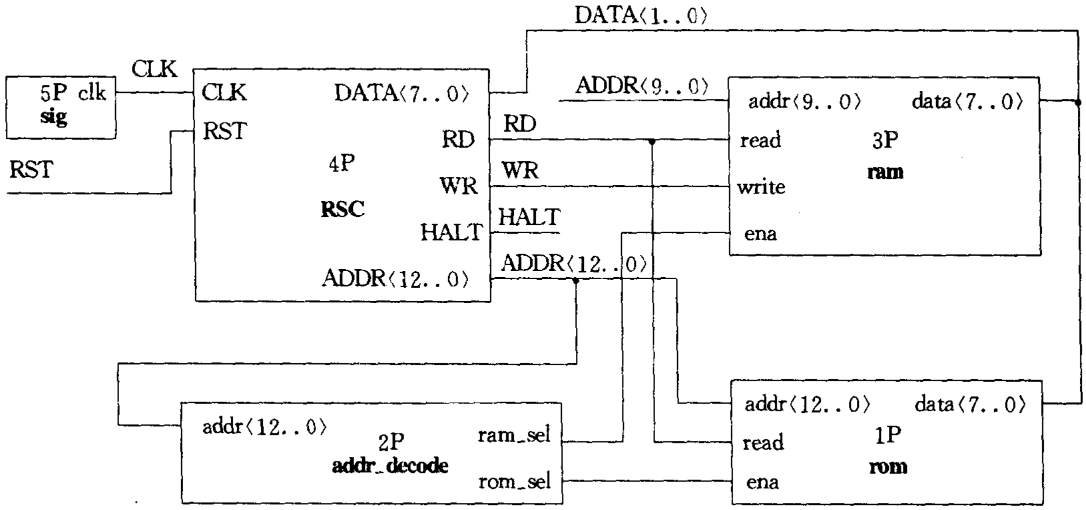


图 5.14 RISC_CPU和它的外围电路

下面介绍的是在 Cadence 公司的 LWB(Logic Work Bench)环境下运行的仿真测试程序 rscsys _ ex1.v ,可用于对所设计的 RISC _ CPU 进行验证。程序很容易读懂,只有很少的地方需要解释,其中用到下面两条语句需要解释一下:

```
$readmemb ( "test1.pro", 'VLP._sys _rom1p _.memory );
```

和

```
$readmemb ( "test1.dat", 'VLP._sys _ram3p _.ram);
```

即可把编译好的汇编机器码装入虚拟 ROM,把需要参加运算的数据装入虚拟 RAM 就可以开始仿真。上面语句中的第一项为打开的文件名,后一项为系统层次管理下的 ROM 模块和 RAM 模块中的存储器 memory 和 ram。

下面程序清单所列出的是用于测试 RISC _ CPU 基本功能而分别装人虚拟 ROM 和 RAM 的机器码和数据文件,其文件名分别为 test1.pro,test1.dat,test2.pro,test2.dat,test3.pro,test3.pro 和调用这些测试程序进行仿真的程序 rscsys _ ex1.v 文件。

```
//------------- 文件 test1.pro ------------------
/*****************************************************
* Test 1 程序是用于验证 RISC _ CPU 的功能,是设计工作的重要环节。本程序测试 RISC _ CPU
* 的基本指令集,如果 RISC _ CPU 的各条指令执行正确,它应在地址为 2E(hex)处,在执行 HLT
* 时停止运行。如果该程序在任何其他地址暂停运行,则必有一条指令运行出错,可参照注释找到
* 出错的指令。
*****************************************************
机器码        地址        汇编助记符          注  释
//-------------- test1.pro 开始 ----------------
@00                                        //address statement
```

```
111 _ 00000  // 00    BEGIN :    JMP TST _ JMP
0011 _ 1100
000 _ 00000  // 02               HLT              //JMP did not work at all
0000 _ 0000
000 _ 00000  // 04               HLT              //JMP did not load PC, it skipped
0000 _ 0000
101 _ 11000  // 06    JMP _ OK : LDA DATA _ 1
0000 _ 0000
001 _ 00000  // 08               SKZ
0000 _ 0000
000 _ 00000  // 0a               HLT              //SKZ or LDA did not work
0000 _ 0000
101 _ 11000  // 0c               LDA DATA _ 2
0000 _ 0001
001 _ 00000  // 0e               SKZ
0000 _ 0000
111 _ 00000  // 10               JMP SKZ _ OK
0001 _ 0100
000 _ 00000  // 12               HLT              //SKZ or LDA did not work
0000 _ 0000
110 _ 11000  // 14    SKZ _ OK :  STO TEMP         //store non-zero value in TEMP
0000 _ 0010
101 _ 11000  // 16               LDA DATA _ 1
0000 _ 0000
110 _ 11000  // 18               STO TEMP         //store zero value in TEMP
0000 _ 0010
101 _ 11000  // 1a               LDA TEMP
0000 _ 0010
001 _ 00000  // 1c               SKZ              //check to see if STO worked
0000 _ 0000
000 _ 00000  // 1e               HLT              //STO did not work
0000 _ 0000
100 _ 11000  // 20               XOR DATA _ 2
0000 _ 0001
001 _ 00000  // 22               SKZ//check to see if XOR worked
0000 _ 0000
111 _ 00000  // 24               JMP XOR _ OK
0010 _ 1000
000 _ 00000  // 26               HLT              //XOR did not work at all
0000 _ 0000
100 _ 11000  // 28    XOR _ OK :  XOR DATA _ 2
0000 _ 0001
001 _ 00000  // 2a               SKZ
```

```
    0000_0000
    000_00000  // 2c              HLT              //XOR did not switch all bits
    0000_0000
    000_00000  // 2e    END:      HLT              //CONGRATULATIONS-TEST1
                                                    PASSED!
    0000_0000
    111_00000  // 30              JMP BEGIN        //run test again
    0000_0000
@3c
    111_00000  // 3c    TST_JMP:  JMP JMP_OK
    0000_0110
    000_00000  // 3e              HLT              //JMP is broken
//————————————————test1.pro 的结束————————————————
//————————————————test1.dat 开始————————————————
@00                                                 //address statement at RAM
    00000000   // 1800  DATA_1:                     //constant 00(hex)
    11111111   // 1801  DATA_2:                     //constant FF(hex)
    10101010   // 1802  TEMP:                       //variable-starts with AA(hex)
//————————————————test1.dat 的结束————————————————
/*************************************************
* Test 2 程序是用于验证 RISC_CPU 的功能，是设计工作的重要环节。本程序测试 RISC_CPU
* 的高级指令集，如果 RISC_CPU 的各条指令执行正确，它应在地址为 20(hex)处，在执行 HLT 时
* 停止运行。如果该程序在任何其他地址暂停运行，则必有一条指令运行出错，可参照注释找到
* 出错的指令。
* 注意：必须先在 RISC_CPU 上运行 test1 程序成功后，才可运行本程序。
*************************************************
机器码            地址      汇编助记符          注  释
//————————————————————test2.pro 开始————————————————
@00
    101_11000  // 00   BEGIN:    LDA DATA_2
    0000_0001
    011_11000  // 02             AND DATA_3
    0000_0010
    100_11000  // 04             XOR DATA_2
    0000_0001
    001_00000  // 06             SKZ
    0000_0000
    000_00000  // 08             HLT              //AND doesn't work
    0000_0000
    010_11000  // 0a             ADD DATA_1
    0000_0000
    001_00000  // 0c             SKZ
    0000_0000
```

```
    111_00000   //0e                  JMP ADD_OK
    0001_0010
    000_00000   //10                  HLT              //ADD doesn't work
    0000_0000
    100_11000   //12      ADD_OK:     XOR DATA_3
    0000_0010
    010_11000   //14                  ADD DATA_1       //FF plus 1 makes -1
    0000_0000
    110_11000   //16                  STO TEMP
    0000_0011
    101_11000   //18                  LDA DATA_1
    0000_0000
    010_11000   //1a                  ADD TEMP         //-1 plus 1 should make zero
    0000_0011
    001_00000   //1c                  SKZ
    0000_0000
    000_00000   //1e                  HLT              //ADD Doesn't work
    0000_0000
    000_00000   //20      END:        HLT              //CONGRATULATIONS-TEST2
                                                         PASSED!
    0000_0000
    111_00000   //22                  JMP BEGIN        //run test again
    0000_0000
//-----------------test2.pro 结束---------------
//----------------test2.dat 开始-----------------
@00
    00000001    //1800                DATA_1:          //constant 1(hex)
    10101010    //1801                DATA_2:          //constant AA(hex)
    11111111    //1802                DATA_3:          //constant FF(hex)
    00000000    //1803                TEMP:
//---------------est2.dat 结束 ---------------
/*****************************************
* Test 3 程序是一个计算从 0～144 的 Fibonacci 序列的程序,用于进一步验证 RISC _ CPU 的功能。
* 所谓 Fibonacci 序列就是一系列数,其中每一个数都是它前面两个数的和(如:0,1,1,2,3,5,8,13,
* 21,…)。这种序列常用于财务分析。
* 注意:必须在成功地运行前两个测试程序后才运行本程序,否则很难发现问题所在。
*****************************************/
机器码          地  址          汇编助记符          注  释
//-------------test3.pro 开始----------------
@00
    101_11000    //00    LOOP:    LDA FN2      //load value in FN2 into accum
    0000_0001
    110_11000    //02             STO TEMP     //store accumulator in TEMP
```

```
    0000_0010
    010_11000    // 04              ADD FN1     //add value in FN1 to accumulator
    0000_0000
    110_11000    // 06              STO FN2     //store result in FN2
    0000_0001
    101_11000    // 08              LDA TEMP    //load TEMP into the accumulator
    0000_0010
    110_11000    // 0a              STO FN1     //store accumulator in FN1
    0000_0000
    100_11000    // 0c              XOR LIMIT   //compare accumulator to LIMIT
    0000_0011
    001_00000    // 0e              SKZ         //if accum = 0, skip to DONE
    0000_0000
    111_00000    // 10              JMP LOOP    //jump to address of LOOP
    0000_0000
    000_00000    // 12    DONE:     HLT         //end of program
    0000_0000
//————————————————————test3.pro 结束————————————————————
//————————————————————test3.dat 开始————————————————————
@00
    00000001     // 1800  FN1:      //data storage for 1st Fib. No.
    00000000     // 1801  FN2:      //data storage for 2nd Fib. No.
    00000000     // 1802  TEMP:     //temproray data storage
    10010000     // 1803  LIMIT:    //max value to calculate 144(dec)
//————————————————————test3.pro 结束————————————————————
```

下面的模块是仿真测试模块 rscsys_ex1，它的作用是按模块的要求执行仿真，并显示仿真的结果。测试模块 rscsys_ex1 中的 $display 和 $monitor 等系统调用能在计算机的显示屏幕上显示部分测试结果，可以同时用波型观察器观察有关信号的波形。

```
//————————————————————rscsys_ex1.v ————————————
/*************************************************
 * rscsys_ex1.v 程序是用于验证 RISC_CPU 的功能的仿真程序，这是设计工作的重要环节。
 * 本仿真程序可在虚拟 RISC_CPU 的外围电路 ROM 和 RAM 上加载 3 个不同的测试程序和数据，
 * 也就是上面介绍的 3 个汇编程序和它们的数据文件：Test1.pro，Test1.dat，Test2.pro，Test2.dat
 * 和 Test3.pro，Test3.dat。当虚拟 RISC_CPU 复位后，它就会执行所加载的程序中的指令。
 * 本仿真程序在程序运行过程中会在屏幕上显示程序计数器的值和当时所执行的指令，并会显示程序
 * 是否运行正常。
 *************************************************/
'timescale  1ns / 100ps

'define  VLP  vlog1
'define  PERIOD  100      // matches clkgen/vlog_behaviorial/verilog.v
module  rscsys_ex1;
```

```
reg   reset _ req;
integer   test;
reg [(3 * 8) : 0]   mnemonic;        //array that holds 3 8-bit ASCII characters
reg [12 : 0]   PC _ addr,IR _ addr;
assign   'VLP. rst = reset _ req;

initial        //display time in nanoseconds
  begin
     $ timeformat ( --9, 1, " ns", 12);
     display _ debug _ message;
  end

always @(posedge 'VLP. halt) //STOP when HALT instruction decoded
   begin
      #500
            $ display("\n * * * * * * * * * * * * * * * * * * * * * * * * * *");
            $ display(" *  A HALT INSTRUCTION WAS PROCESSED !!!  *");
            $ display(" * * * * * * * * * * * * * * * * * * * * * * * * * *\n");
            display _ debug _ message;
   end

task   display _ debug _ message;
   begin
      $ display("\n * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *");
      $ display(" *  THE FOLLOWING DEBUG TASK ARE AVAILABLE :  *");
      $ display(" *  Enter \"test1;\" to load the 1st diagnostic progran.  *");
      $ display(" *  Enter \"test2;\" to load the 2nd diagnostic program.  *");
      $ display(" *  Enter \"test3;\" to load the Fibonacci program.  *");
      $ display(" * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *\
      n");
      $ stop;
   end
endtask

task   test1;
   begin
      test = 0;
      disable   MONITOR;
      $ readmemb("memory. files/CPUtest1. pro",'VLP. _ sys _ rom1p _. memory);
      $ display("rom loaded successfully!");
      $ readmemb("memory. files/CPUtest1. dat",'VLP. _ sys _ ram3p _. ram);
      $ display("ram loaded successfully!");
      #1   test = 1;
```

```
        sys _ reset;
    end
endtask

task  test2;
    begin
        test = 0;
        disable  MONITOR;
        $readmemb("memory.files/CPUtest2.pro",'VLP._sys _rom1p _.memory);
        $display("rom loaded successfully!");
        $readmemb("memory.files/CPUtest2.dat",'VLP._sys _ram3p _.ram);
        $display("ram loaded successfully!");
        #1  test = 2;
        sys _ reset;
    end
endtask

task  test3;
    begin
        test = 0;
        disable  MONITOR;
        $readmemb("memory.files/CPUtest3.pro",'VLP._sys _rom1p _.memory);
        $display("rom loaded successfully!");
        $readmemb("memory.files/CPUtest3.dat",'VLP._sys _ram3p _.ram);
        $display("ram loaded successfully!");
        #1  test = 3;
        sys _ reset;
    end
endtask

task  sys _ reset;
    begin
        reset _ req = 0;
        # ('PERIOD * 0.7)  reset _ req = 1;
        # (1.5 * 'PERIOD)  reset _ req = 0;
    end
endtask

always @(test)
    begin : MONITOR
        case (test)
            1 : begin       //display results when running test 1
                $display("\n* * * RUNNING CPUtest1 ——The Basic CPU Diagnostic Program * *
```

```
        *");
        $display("\n TIME PC INSTR ADDR DATA "); $display(" ----------
        ----- ------ --- --- ");
        while (test == 1)
        @('VLP._sys_rsc4p_adr22p_.pc_addr)
        if (('VLP._sys_rsc4p_adr22p_.pc_addr%2 == 1)
        && ('VLP._sys_rsc4p_adr22p_.fetch == 1))
        begin
        # 60  PC_addr <= 'VLP._sys_rsc4p_adr22p_.pc_addr -1 ;
        IR_addr <= 'VLP._sys_rsc4p_adr22p_.ir_addr;
        # 340  $strobe("%t %h %s %h %h",
        $time, PC_addr, mnemonic, IR_addr,'VLP.data );
        end
end
2 : begin
        $display("\n* * * RUNNING CPUtest2 ——The Advanced CPU Diagnostic Program *
        * *");
        $display("\n TIME PC INSTR ADDR DATA ");
        $display(" ---------- ----- ------ ------ ------ ");
        while (test == 2)
        @('VLP._sys_rsc4p_adr22p_.pc_addr)
        if (('VLP._sys_rsc4p_adr22p_.pc_addr%2 == 1)
        && ('VLP._sys_rsc4p_adr22p_.fetch == 1))
        begin
        # 60  PC_addr <= 'VLP._sys_rsc4p_adr22p_.pc_addr - 1 ;
        IR_addr <= 'VLP._sys_rsc4p_adr22p_.ir_addr;
        # 340  $strobe("%t %h %s %h %h",
        $time, PC_addr, mnemonic, IR_addr,'VLP.data );
        end
end
3 : begin
        $display("\n* * * RUNNING CPUtest3 ——An Executable Program * * *");
        $display(" * * * This program should calculate the fibonacci * * *");
        $display(" * * * number sequence from 0 to 144 * * *");
        $display("\n TIME FIBONACCI NUMBER");
        $display(" ---------- ----------");
        while (test == 3)
          begin
            wait ( 'VLP._sys_rsc4p_alu15p_.opcode == 3'h1) // display Fib. No. at end of
            program  loop
            $strobe("%t %d", $time, 'VLP._sys_ram3p_.ram[10'h2]);
            wait ( 'VLP._sys_rsc4p_alu15p_.opcode ! = 3'h1);
          end
```

```
        end
    endcase

    end

    always @(VLP._sys_rsc4p_alu15p_.opcode)
      //get an ASCII mnemonic for each opcode
    case('VLP._sys_rsc4p_alu15p_.opcode)
       3'h0 : mnemonic = "HLT";
       3'h1 : mnemonic = "SKZ";
       3'h2 : mnemonic = "ADD";
       3'h3 : mnemonic = "AND";
       3'h4 : mnemonic = "XOR";
       3'h5 : mnemonic = "LDA";
       3'h6 : mnemonic = "STO";
       3'h7 : mnemonic = "JMP";
       default : mnemonic = "???";
  endcase
endmodule
//————————————————rscsys_ex1.v 结束————————————————
```

下面列出的是执行 rscsys_ex1.v 仿真程序加载运行 3 个汇编测试程序后的输出文件。

```
//————————————————仿真程序运行结果开始————————————————
RSCSYS_EX1
VLOG1
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* THE FOLLOWING DEBUG TASK ARE AVAILABLE : *
* Enter "test1;" to load the 1st diagnostic program. *
* Enter "test2;" to load the 2nd diagnostic program. *
* Enter "test3;" to load the Fibonacci program. *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
L52 "rscsys_ex1.v" : $stop at simulation time 0.0 ns
Type ? for help
C1 > test1;
C2 > .
rom  loaded successfully!
ram  loaded successfully!

* * * RUNNING CPUtest1 ——The Basic CPU Diagnostic Program * * *
TIME          PC      INSTR      ADDR      DATA
————————  ————  ————————  ————————  ————————
850.0 ns      0000    JMP        003c      zz
1650.0 ns     003c    JMP        0006      zz
2450.0 ns     0006    LDA        1800      00
```

```
3250.0 ns       0008    SKZ     0000    zz
4050.0 ns       000c    LDA     1801    ff
4850.0 ns       000e    SKZ     0000    zz
5650.0 ns       0010    JMP     0014    zz
6450.0 ns       0014    STO     1802    ff
7250.0 ns       0016    LDA     1800    00
8050.0 ns       0018    STO     1802    00
8850.0 ns       001a    LDA     1802    00
9650.0 ns       001c    SKZ     0000    zz
10450.0 ns      0020    XOR     1801    ff
11250.0 ns      0022    SKZ     0000    zz
12050.0 ns      0024    JMP     0028    zz
12850.0 ns      0028    XOR     1801    ff
13650.0 ns      002a    SKZ     0000    zz
14450.0 ns      002e    HLT     0000    zz
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* A HALT INSTRUCTION WAS PROCESSED !!! *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* THE FOLLOWING DEBUG TASK ARE AVAILABLE : *
* Enter "test1;" to load the 1st diagnostic program. *
* Enter "test2;" to load the 2nd diagnostic program. *
* Enter "test3;" to load the Fibonacci program. *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

L52 "rscsys _ ex1.v" : $stop at simulation time 14750.0 ns
C2 > test2;
C3 > .
rom  loaded successfully!
ram  loaded successfully!
* * * RUNNING CPUtest2 —— The Advanced CPU Diagnostic Program * * *
    TIME        PC      INSTR     ADDR      DATA
    ----------  ------  --------  --------  --------
    16050.0 ns  0000    LDA       1801      aa
    16850.0 ns  0002    AND       1802      ff
    17650.0 ns  0004    XOR       1801      aa
    18450.0 ns  0006    SKZ       0000      zz
    19250.0 ns  000a    ADD       1800      01
    20050.0 ns  000c    SKZ       0000      zz
    20850.0 ns  000e    JMP       0012      zz
    21650.0 ns  0012    XOR       1802      ff
    22450.0 ns  0014    ADD       1800      01
```

```
    23250.0 ns    0016    STO    1803    ff
    24050.0 ns    0018    LDA    1800    01
    24850.0 ns    001a    ADD    1803    ff
    25650.0 ns    001c    SKZ    0000    zz
    26450.0 ns    0020    HLT    0000    zz
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* A HALT INSTRUCTION WAS PROCESSED !!! *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* THE FOLLOWING DEBUG TASK ARE AVAILABLE : *
* Enter "test1;" to load the 1st diagnostic program. *
* Enter "test2;" to load the 2nd diagnostic program. *
* Enter "test3;" to load the Fibonacci program. *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

L52 "rscsys _ ex1.v" : $stop at simulation time 26750.0 ns
C3 > test3;
C4 > .
rom  loaded successfully!
ram  loaded successfully!
 * * * RUNNING CPUtest3 ——An Executable Program * * *
 * * * This program should calculate the fibonacci * * *
 * * * number sequence from 0 to 144 * * *
    TIME          FIBONACCI NUMBER
    ————————————————————————
    33200.0 ns    0
    40400.0 ns    1
    47600.0 ns    1
    54800.0 ns    2
    62000.0 ns    3
    69200.0 ns    5
    76400.0 ns    8
    83600.0 ns    13
    90800.0 ns    21
    98000.0 ns    34
    105200.0 ns   55
    112400.0 ns   89
    119600.0 ns   144
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * A HALT INSTRUCTION WAS PROCESSED !!! *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//————————————仿真程序运行结果结束————————————
```

在运行了以上程序后，如仿真程序运行的结果正确，前仿真(即布局布线前的仿真)可告结

束。

5.5.2　RISC _ CPU 模块的综合

在对所设计的 RISC _ CPU 模型进行验证后，如没有发现问题就可开始做下一步的工作——综合。综合工作往往要分阶段来进行，这样便于发现问题。

第一阶段：先对构成 RISC _ CPU 模型的各个子模块，如状态控制机模块（包括 machine 模块，machinectl 模块）、指令寄存器模块（register 模块）、算术逻辑运算单元模块（alu 模块）等，分别加以综合以检查其可综合性。综合后及时进行后仿真，这样便于及时发现错误，及时改进。

第二阶段：把要综合的模块从仿真测试信号模块和虚拟外围电路模型（如 ROM 模块、RAM 模块、显示部件模块等）中分离出来，组成一个独立的模块，其中包括了所有需要综合的模块。然后，给这个大模块起一个名字，如本章中的例子。我们要综合的只是 RISC _ CPU，并不包括虚拟外围电路，可以给这一模块起一个名字，例如称它为 RSC _ CHIP 模块。如用电路图描述的话，还需给它的引脚加上标准的引脚部件并加标记，如图 5.15 所示。

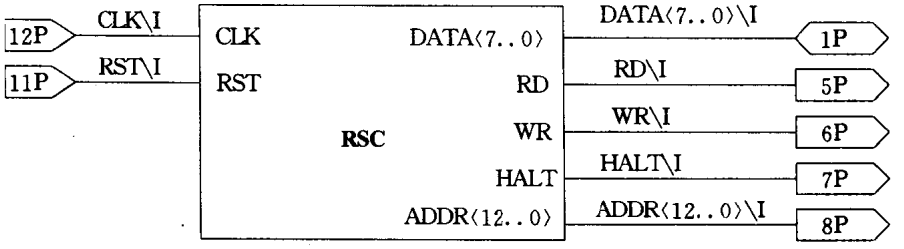


图 5.15　用于综合的RISC_CPU模块（RSC）

第三阶段：加载需要综合的模块到综合器，选定某一厂家的部件库（如 Xilinx 的库）或某一类部件库（如通用 FPGA 库，pic _ lib ）进行综合。

综合器综合的结果会产生一系列的文件，其中有一个文件报告用了多少基本单元。见下面的报告，它就是这个 RISC _ CPU 芯片所用的基本单元数。综合所用的库是通用的 FPGA 类库（PIC _ LIB）。

```
//------------ RISC _ CPU 芯片综合结果报告开始----------
MODULE : RISC _ CPU _ CHIP
```

Cell	Count	unit _ area each	unit _ area total
PLD _ AND5	1	1.00	1.00
PLD _ GND	1	1.00	1.00
PLD _ NOR4	1	1.00	1.00
PLD _ OR8	1	1.00	1.00
PLD _ ADDER8	1	1.00	1.00
PLD _ AND6	2	1.00	2.00
PLD _ JKFF	1	2.00	2.00
PLD _ AND3	4	1.00	4.00

PLD_NAN3	5	1.00	5.00
PLD_BUFTL	8	1.00	8.00
PLD_XOR	10	1.00	10.00
PLD_OR3	10	1.00	10.00
PLD_NOR3	10	1.00	10.00
PLD_NAN2	13	1.00	13.00
PLD_MUX1	21	1.00	21.00
PLD_NOR2	29	1.00	29.00
PLD_INV	30	1.00	30.00
PLD_DFF	59	1.00	59.00
PLD_OR2	61	1.00	61.00
PLD_AND2	65	1.00	65.00
Total	333		334.00

//————————— RISC_CPU 芯片综合结果报告结束—————————

5.5.3 RISC_CPU 模块的优化和布局布线

选定部件库后就可以对所设计的 RISC_CPU 模型进行综合，综合后产生了一系列的文件，其中 XXXXX.edf 文件就是与所选定的厂家部件库对应的电子设计交换格式(Electronic Design Interchange Format)文件或是与某一类部件库(如通用 FPGA 库)对应的电子设计交换格式文件，这也就是在电路设计工业界常说的 EDIF 格式文件。若是选用了某一类部件库(如通用 FPGA 库)所生成的 EDIF 文件，还需要编写物理文件，说明究竟用了哪一厂家的哪一种部件，才能做优化，由优化所生成的文件才能加载到有关厂家的布局布线工具做布局布线等后续工作。从做完布局布线工序后所生成的文件中，可以提取标准延时格式(Standard Delay Format)文件(即.SDF 文件)，生成后仿真(post-simulation)所需的带实际布线延时的 Verilog HDL 模型，再用原先仿真时所用的测试信号模块接到这个带布线延时的 Verilog HDL 模型，即可开始做后仿真，如图 5.16 所示。不同厂家的布局布线提供不同的后仿真解决方法，所以很难用一句话作全面的介绍。

如后仿真正确无误，就可以把布局布线后生成的一系列文件送 ASIC 厂家或加载到 FPGA 器件的编码工具，使其变为专用的电路芯片。如后仿真中发现有错误，先降低测试信号模块的时钟频率，如该问题解决了，则需要找到造成问题的关键路径，下一次在布局布线时应先布关键的路径(即在约束文件中注明该路径是关键路径后，再重做自动布局布线)。若还有问题，则需检查各模块中是否有个别模块没有按照同步设计的原则，若是，则需改写有关的 Verilog HDL 模块。重复以上工作，直到后仿真正确无误。以上所述的就是用 Verilog HDL 设计一个复杂数字电路系统的步骤。读者可以参考以上步骤，自己来设计一个可在 FPGA 上实现的小 RISC_CPU 系统。

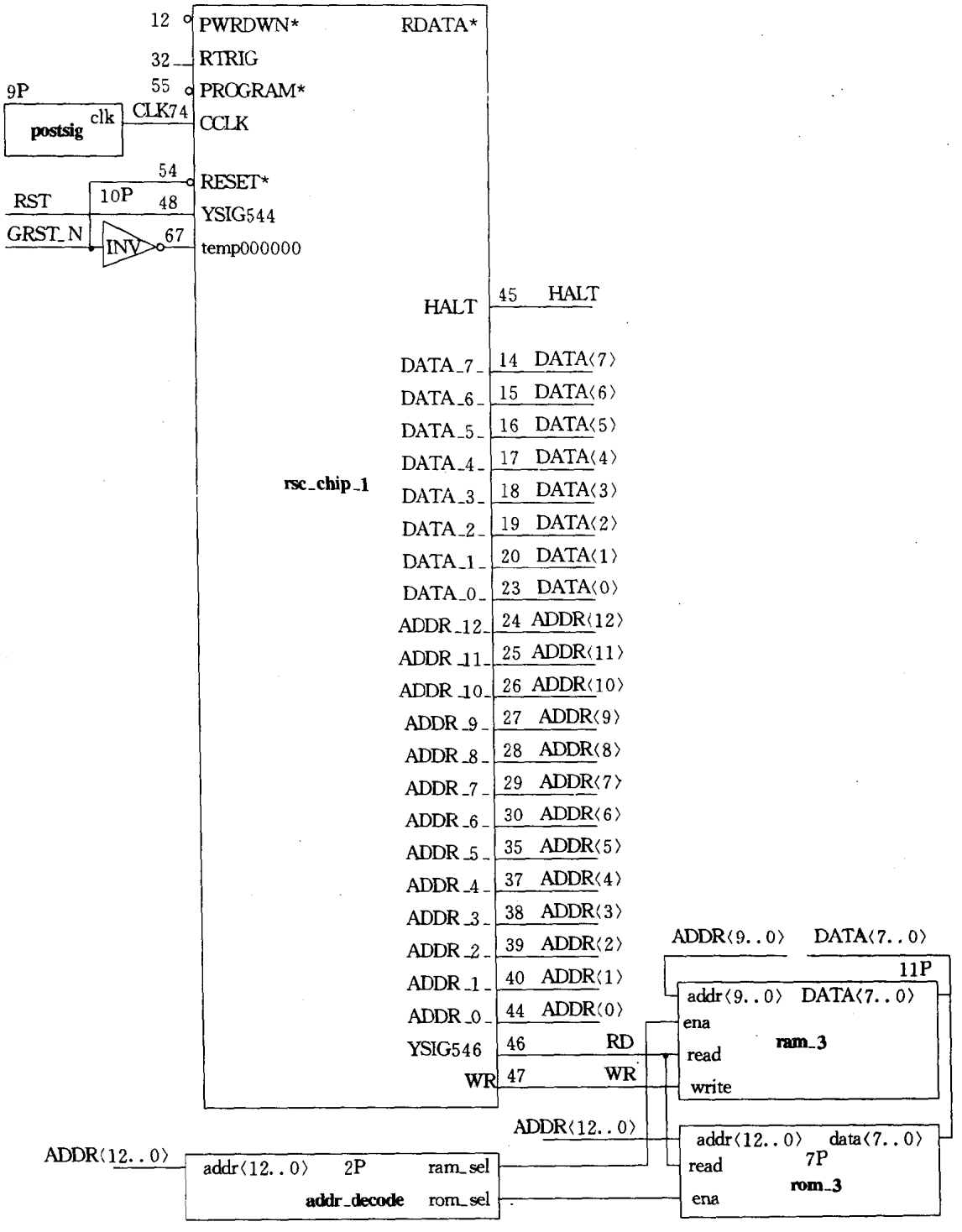


图 5.16　后仿真的接线图

思 考 题

1. 请叙述一下设计一个复杂数字系统的步骤。
2. 综合一个大型的数字系统需要注意什么?
3. 请改进以上 RISC _ CPU,把指令数增至 16,寻址空间降为 4K。

第六章　虚拟器件和虚拟接口模型

宏单元(Macrocells 或 Megacells)或核(Cores)是预先设计好的,其功能是经过验证的、由总数超过 5 000 个门构成的、一体化的电路模块。这个模块可以是以软件为基础的,也可以是以硬件为基础的。这就是在第一章的 1.5.3 节和 1.5.4 节中讨论过的软核和硬核。所谓虚拟器件(Virtual Chips)也就是用软核构成的器件,即用 Verilog HDL 或 VHDL 语言描述的常用大规模集成电路模型。在新电路研制过程中,借助 EDA 综合工具,软核和虚拟器件可以很容易地与其它外部逻辑结合为一体,从而大大扩展了设计者可选用的资源。掌握软核和虚拟器件的重用技术可大大缩短设计周期,加快高新技术芯片的投产和上市 。而所谓虚拟接口模型则是用系统级 Verilog HDL 或 VHDL 语言描述的常用大规模集成电路(如 ROM 和 RAM)或总线接口的行为模型等,往往是不可综合的,也没有必要综合成具体电路,但其所有对外的性能与真实的器件或接口完全一致,在仿真时可用来代替真实的部件,用以验证所设计的电路(必须综合的部分)是否正确。

在美国,各种微处理器芯片(如 8051)、通用串行接口芯片(如 8251)、中断控制器芯片(如 8259)、并行输入输出接口芯片(PIO)、直接存储器存取芯片(DMA)、数字信号处理芯片(DSP)、RAM 和 ROM 芯片、PCI 总线控制器芯片以及 PCI 总线控制接口等都有其相对应的商品化的虚拟器件和虚拟接口模型可供选用。虚拟器件往往只提供门级和 RTL 级的 Verilog HDL 或 VHDL 源代码,而虚拟接口模型往往提供系统级代码。这是因为门级和 RTL 级的 Verilog HDL 或 VHDL 是可综合的,它与具体的逻辑电路有着精确的对应关系。

近年来,在现代数字系统设计领域中发展最快的一个部门就是提供虚拟器件和虚拟接口模型的设计和服务。目前国际上有一个叫作虚拟接口联盟(VSIA)的组织,它是协调虚拟器件和虚拟接口模型的设计标准和服务工作的国际组织。虚拟器件和虚拟接口模型必须符合通用的工业标准和达到一定的质量水准,才能发布。这对选用虚拟器件和虚拟接口模型来设计复杂系统的工程师们无疑有很大的帮助。如果他们采用虚拟器件和虚拟接口模型技术来设计复杂的数字系统,必将大大缩短设计周期并提高设计的质量,也为百万门级单片系统(one million gate system on a chip)的实现铺平了道路。

6.1　虚拟器件和虚拟接口模块的供应商

在这一节中列出一些虚拟器件和接口的供应商的 E－mail 地址及它们提供的产品和服务供读者参考。

表 6.1 部分器件及接口的供应商地址及其产品

公司名	虚拟器件类型	所用语言	加密否	语言级别
American Microsystem 电子信箱： tdrake @poci. amis. com	算术运算函数 异步同步 FIFO DSP 微处理器 UART 和 USARTs RAM 和 ROM	Verilog VHDL	不	门级 RTL 级
ARMSemiconductor 电子信箱： armsemi @netcom. com	微处理器： 8031,8032,8051 通信器件： 8530 总线控制器： 82365 (PCMCIA Host i/f)	Verilog	可选	系统级(只可用于仿真)
Scenix Semiconductor 电子信箱： sales@scenix. com	控制器： NS COP8 PCI arbiter,master & target 8237 DMA	Verilog	不	门级 RTL 级
Sierra Research and Technology 电子信箱： core@srti. com	ATM SAR 622 Mbits Ethernet 控制器 100/10-Mbits CPU R3000 核	Verilog	不	门级 RTL 级
Silicon Engineering 电子信箱： info@sei. com	Micro VGA	Verilog	不	门级 RTL 级
Lucent Technology 电子信箱： attfpga @aloft. att. com	DSP PCI Master PCI target	Verilog VHDL	不	门级、RTL 级和系统级(只可用于仿真)3 种都可提供

6.2 虚拟接口模块的实例

下面我们列出了一个常用的大规模集成芯片——通用串行收发控制器 USART8251 的 Verilog HDL 源代码。这个 Verilog HDL 模块是由 Verilog HDL 语言的创始人 P. R. Moorby 和 D. E. Thomas 合作编写的(这是我们从 Internet 网络上下载得到的)。

因为商品化的虚拟器件和虚拟接口模型价格非常昂贵，不可能免费得到。下面的模块从严格意义上说来并非是真正的虚拟接口模型，只是作为学习较复杂的 Verilog HDL 模块编写的样本。

在 8251A 虚拟接口模块程序的原始材料上有下面这样一段话：

/*************************************

THE 8251A EXAMPLE

CADENCE DESIGN SYSTEMS, Inc. does not guarantee the accuracy or completeness of this model. Anyone who using this does so at their own risk.

*************************************/

显然，这一模块是一个仅供参考的实例，本书姑且把它当作虚拟接口模型来看待，因为它的 Verilog HDL 程序是严格地按照 8251A 的说明书而编写的，读者可以对照 8251A 的说明书仔细阅读这一程序。下面就是 USART8251A 虚拟模块的 Verilog HDL 程序，供读者参考。

```
/************************************
THE 8251A EXAMPLE
CADENCE DESIGN SYSTEMS, Inc. does not guarantee the accuracy or completenessof this model.
Anyone who using this does so at their own risk.
************************************/
module  I8251A ( dbus, rcd, gnd, txc _, write _, chipsel _, comdat _, read _, rxrdy, txrdy, syndet,
                 cts _, txe, txd, clk, reset, dsr _,rts _,dtr _,rxc _,vcc);
  parameter [7:0]  instance_id        =8'h00;
  parameter [8:1]   dlags             =8'b00000100;//Defaults for diagnostics
  //diagnostic  dflags:                          ||||
  //bit 5 (16)operation event trace      ________|||
  //bit 4 (8) communication errors       _________||
  //bit 3 (4) timing check          ______________||
  //bit 2 (2) print receiving       _______________|
  //bit 1 (1) print transmitting    _______________|

 /* timing constants ,for A. C. timing check, only non-zero times are specified,in nano-sec */
 /* read cycle */
 'define  TRR  250
 'define  TRD  200
 'define  TDF  100 // max. time used
 /* write cycle */
 'define  TWW  250
 'define  TDW  150
 'define  TWD  20
 'define  TRV  6 // in terms of clock cycles

 /* other timing */
 'define  TTXRDY  8 // 8 clock cycle
```

```
input   rcd,                //receive data
        rxc_,               //receive clock
        txc_,               //transmit clock
        chipsel_,           //chip selected when low
        comdat_,            //command /data_select
        read_,write_,
        dsr_,               // data set ready
        cts_,               // clear to send
        reset,              // reset when high
        clk,                // at least 30 times of the transmit/rexeibe data bit rates
        gnd,
        vcc;
output  rxrdy,              //receive data ready when high
        txd,                //transmit data lone
        txrdy,              //transmit buffer ready to accept another byte to transfer
        txe,                // transmit buffer empty
        rts_,               // request to send
        dtr_;               // data terminal ready
inout [7:0]  dbus;

inout   syndet;             //outside synchonous detect or output to indicate syn det

supply0  gnd;
supply1  vcc;

reg  txd, rxrdy, txe, dtr_, rts_;

reg [7:0]  instance_id, receivebuf, rdata, status;

reg [3:0]  dflags;

reg  read, chipel_, recvdrv, statusdrv;
    // if recvdrv 1 dbus is driven by rdata
assign  dbus = recvdrv ? rdata : 8'bz;
        assign  dbus = statusdrv ? status : 8'bz ;

reg[7:0]  command,
          tdata_out, // data being transmitted serially
          tdata_hold, // data to be transmitted next if tdata_out is full
          sync1,sync2, // synchronous data bytes
          modreg;
and  (txrdy, status[0], command[0],~ cts_);
reg  transmitter_reset, // set to 1 upon a reset ,cleared upon write data
```

```
    tdata _ out _ full, // 1 if data in tdata _ out has not been transmitted
    tdata _ hold _ full, // 1 if data in tdata _ hold has not been transferred to tdata _ out for serial
                         transmission
    tdata _ hold _ cts; // 1 if tdata _ hold _ full and it was cts when data was //transferred to tdata _
                         hold
    //0 if tdata _ hold is empty or is full but was filled while it was //not cts
reg  tdata _ out _ wait; //0 if a stop bit was just sent and we do not need to wait for a negedge on txc
                         before.transmitting
reg[7 : 0]  syncmask;
nmos syndet _ gate1(syndet,status[6], ~ modreg[6]);

reg  sync _ to _ receive; // 1(2) if looking for 1st(2nd) sync on rxd
reg  syncs _ received;   // 1 if sync chars received, 0 if lookinf for sync(
reg  rec _ sync _ index; // indicating the syn. character to be matched

integer  breakcount _ period; // number of clock periods to count as break
reg  sync _ to _ transmit; //1(2) if 1st(2nd) sync char should be sent next
reg[7 : 0]  data _ mask; //masks off the data bits (if char size is not 8)
// temporary registers
reg[1 : 0]  csel; //indicates what next write means if comdat _=1 :
              //(0=mode instruction ,1=sync1,2=sync2,3=command)
reg[5 : 0]  baudmx,
            tbaudcnt,
            rbaudcnt;  // baud rate
reg[7 : 0]  tstoptotal; // no. of tranmit clock pulses for stop bit (0 if sync mode
reg[3 : 0]  databits; // no. of data bits in a character (5,6,7 or 8)
reg  rdatain; // a data byte is read in if 1

reg  was _ cts _ when _ received; // 0 : if cts _ was high when char was received
                                  // 1 : if cts _ was low wheb char was received
                                  // (and so char was sent before shutdown)
event  resete, start _ receiver _ e,hunt _ sysnc1 _ e;
reg  receive _ in _ progress;
event  txende;
                        /* * * COMMUNICATION ERRORS * * */
task  frame _ error;
      begin
        if(dflags[4])
          $ display("I8251A (%h)at %d :  * * * frame error ",instance _ id, $ time);
        status[5]=1;
      end
endtask
```

```
task  parity _ error;
      begin
        if(dflags[4])
        $ display("I8251A (%h) at %d : * * * parity error data : %b",
                                  instance _ id, $ time, receivebuf);
        status[3]=1;
      end
endtask

task  overrun _ error;
      begin
        if(dflags[4])
          $ display("I8251A (%h) at %d : * * * oerrun error",instance _ id, $ time);
        status[4]=1;
      end
endtask

                    /* * *        TIMING VIOLATIONS        * * */

integer  time _ dbus _ setup,
         time _ write _ begin,
         time _ write _ end,
         time _ read _ begin,
         time _ read _ end,
         between _ write _ clks; // to check between write recovery

reg  reset _ signal _ in; //to check the reset signal pulse width

initial
  begin
     time _ dbus _ setup = -9999;
     time _ write _ begin = -9999;
     time _ write _ end = -9999;
     time _ read _ begin = -9999;
     time _ read _ end = -9999;
     between _ write _ clks = 'TRV; //start : TRV clk periods since last write
  end

                    /* * *     Timing analysis for read cycles     * * */

always @( negedge read _)
       if (chipsel _ ==0)
       begin
```

```
        time _ read _ begin = $ time;
        read _ address _ watch;
    end

                /* Timing violation : read pulse must be TRR ns */

always @(posedge read _)
    if (chipsel _ == 0)
    begin
        disable read _ address _ watch;
        time _ read _ end = $ time;

        if(dflags[3] && (($ time - time _ read _ begin) < 'TRR))
            $ display("I8251A (%h) at %d : * * * read pulse width violation", instance _ id,
            $ time);
    end
        /* Timing violation : address (comdat _ and chipsel _) must be stable */
        /*                    stable throughout read                          */

task  read _ address _ watch;
    @(comdat _ or chipsel _) //if the "address" changes
      if (read == 0) // and read _ did not change at the same time
        if (dflags[3])
          $ display("I8251A (%h) at %d : * * * address hold error on ready", instance _ id,
                                                $ time);
endtask

                /* * Timing analysis for write cycles * */

always @(negedge write _)
    if (chipsel _ == 0)
    begin
        time _ write _ begin = $ time;
        write _ address _ watch;
    end

/* Timing violation : read pulse must be TRR ns */
/* Timing violation : TDW ns bus setup time before posedge write _ */
/* Timing violation : TWD ns bus hold time after posedge write _ */
always @(posedge write _)
    if (chipsel _ == 0)
    begin
        disable write _ address _ watch;
```

```
        time _ write _ end= $ time;
        if(dflags[3] && (( $ time-time _ write _ begin) < 'TWW))
        $ display(" I8251A (%h) at %d : * * * write pulse width violation", instance _ id,
        $ time);
      end

always @dbus
      begin
        time _ dbus _ setup= $ time;
        if(dflags[3] && (( $ time-time _ write _ end < 'TWD)))
          $ display("I8251A (%h) at %d : * * * data
                    hold violation on write",
                    instance _ id , $ time);
        end

        /* Timing violation : address (comdat _ and chipsel _ ) must be stable   */
        /*                    stable throughout write                            */

task  write _ address _ watch;
      @(comdat _ or chipsel _ ) //if the "address" changes
            if (write _==0) // and write _ did not change at the same time
              if (dflags[3])
                $ display("I8251A (%h) at %d : * * * address
                          hold error on write",instance _ id , $ time);
endtask
          /* Timing violation : minimum of TRV clk cycles between writes */
always @( negedge write _ )
      if ( chipel _==0 )
      begin
        time _ write _ begin= $ time;
        if(dflags[3] && between _ write _ clks < 'TRV)
          $ display("I8251A (%h) at %d : * * * between write
                    recovery violation",instance _ id, $ time);
         end

always @(negedge write _)
      repeat ('TRV) @(posedge clk)
          between _ write _ clks = between _ write _ clks +1 ;

                    /* * Timing analysis for reset sequence * */
              /* Timing violation : reset pulse must be 6 clk cycles */
always @(posedge reset )
      begin : reset _ block
```

```
        reset _ signal _ in=1;
        repeat(6) @(posedge clk);
        reset _ signal _ in=0;
        //external reset
        -> resete;
        end
always @(negedge reset)
        begin
        if(dflags[3] && (reset _ signal _ in==1))
          $ display("I8251A (%h) at %d : * * * reset pulse too short ", instance _ id , $ time);
        disable reset _ block;
        end

    /* * * BEHAVIORAL DESCRIPTION * * */
    /* Reset sequence */
initial
    begin //power-on reset
        reset _ signal _ in=0;
        -> resete;
    end

always @ resete
        begin
        if(dflags[5])
          $ display("I8251A (%h) at %d : performing reset sequence",
                    instance _ id, $ time);
        csel=0;
        transmitter _ reset=1;
        tdata _ out _ full=0;
        tdata _ out _ wait=0;
        tdata _ hold _ full=0;
        tdata _ hold _ cts=0;
        rdatain=0;
        status=4; //only txe is set
        txe=1;
        statusdrv=0;
        recvdrv=0;
        txd=1; //line at mark state upon reset until data is transmitted
            // assign not allowed for status ,etc.
        rxrdy=0;
        command=0;
        dtr _=1;
        rts _=1;
```

```
        status[6]=0; // syndat is reset to output low
        sync_to_transmit=1; //transmit sync char #1 when sync are transmit
        sync_to_receive=1;
        between_write_clks = 'TRV;
        receive_in_progress=0;
        disable read_address_watch;
        disable write_address_watch;
        disable trans1;
        disable trans2;
        disable trans3;
        disable trans4;
        disable rcv_blk;
        disable sync_hunt_blk;
        disable double_sync_hunt_blk;
        disable parity_sync_hunt_blk;
        disable syn_receive_internal;
        disable asyn_receive;
        disable break_detect_blk;
        disable break_delay_blk;
        end

always @ ( negedge read_)
        if (chipsel_==0)
        begin
            #('TRD) // time for data to show on the data bus

            if (comdat_==0) //8251A DATA ==> DATA BUS
            begin
                recvdrv=1;
                rdatain=0; // no receive byte is ready
                rxrdy=0;
                status[1]=0;
            end
            else // 8251A STATUS ==> DATA BUS
            begin
                statusdrv=1;
                if (modreg [1:0] ==2'b00) // if sync mode
                    status[6]=0; // reset syndet upon status ready
                    //note : is only reset upon reset or rxd=1 in async mode
            end
        end

always @ ( posedge read_)
```

```
        begin
            #('TDF) //data from read stays on the bus after posedge read _
            recvdrv=0;
            statusdrv=0;
        end

always @(negedge write _)
begin
    if((chipsel _==0)&&(comdat _==0))
    begin
        txe=0;
        status[2]=0;//transmitter not empty after receiving data
        status[0]=0;//transmitter not ready after receiving data
    end
end

always @(posedge write _)          //read the command/data from the CPU
      if (chipsel _==0)
      begin
          if (comdat _==0) //DATA BUS ==> 8251A DATA
          begin
              case (command[0] &~ cts _)
              0 :                  //if it is not clear to send
              begin
                  tdata _ hold=dbus;
                  tdata _ hold _ full=1; //then mark the data as received and
                  tdata _ hold _ cts=0; // that it should be sent when cts
              end
              1 : // if it is clear to send ···
              if(transmitter _ reset) // ··· and this is 1st data since reset
              begin
                  transmitter _ reset=0;
                  tdata _ out=dbus;
                  tdata _ out _ wait=1; // then wait for a negedge on txc
                  tdata _ out _ full=1; // and transmit the data
                  tdata _ hold _ full=0;
                  tdata _ hold _ cts=0;
                  repeat('TTXRDY) @(posedge clk);
                  status[0]=1; // and set the txrdy status bit
              end
              else
              begin
                  tdata _ hold=dbus; // then mark the data as being receive
```

```
        tdata _ hold _ full=1; // and that it should be transmitted
        tdata _ hold _ cts=1; // it becomes not cts,
                              // but do not set the txrdy status bit
    end
    endcase
end
else // DATA BUS ==> CONTROL
begin
    case (csel)
    0 : // case 0 : MODE INSTRUCTION
    begin
        modreg=dbus;
        if(modreg[1 : 0]==0) // synchronous mode
        begin
            csel=1;
            baudmx=1;
            tstoptotal=0; // no stop bit for synch. Op.
        end
        else                        //synchronous mode
        begin
            csel=3;
            baudmx=1; //1X baud rate
            if (modreg[1 : 0]==2'b10) baudmx=16;
            if(modreg[1 : 0]==2'b11) baudmx=64;
            //set up the stop bits in clocks
            tstoptotal=baudmx;
            if(modreg[7 : 6]==2'b10) tstoptotal= tstoptotal + baudmx/2;
            if(modreg[7 : 6]==2'b11) tstoptotal= tstoptotal+tstoptotal;
        end
        databits=modreg[3 : 2]+5; // bits per char
        data _ mask=255 >> (3-modreg[3 : 2]);
    end

    1 : //case 1 : 1st SYNC CHAR -SYNC MODE
    begin
        sync1=dbus;
        /* the syn. character will be adjusted to the most
            significant bit to simplify syn, hunt,
            syncmask is also set to test the top data bits */
        case (modreg[3 : 2])
        0 :
        begin
            sync1=sync1<<3;
```

```
        syncmask=8'b11111000;
    end

    1:
    begin
        sync1=sync1<< 2;
        syncmask=8'b11111110;
    end
    2:
    begin
        sync1=sync1<< 1;
        syncmask=8'b11111110;
    end
    3:
        syncmask=8'b11111111;
    endcase

    if(modreg[7]==0)
            csel=2; //if in double sync char mode, get 2 syncs
    else
            csel=3; // if in single sync char mode, get 1 sync
end

2: //case 2: 2nd SYNC CHAR - SYNC MODE
begin
    sync2=dbus;
    case (modreg[3:2])
    0: sync2=sync2<< 3;
    1: sync2=sync2<< 2;
    2: sync2=sync2<< 1;
    endcase

    csel=3;
end

3: // case 3: COMMAND INSTRUCTION - SYNC/ASYNC MODE
begin
    status[0]=0; //Trick: force delay txtdy pin if command[0]
    command=dbus;
    dtr_= ! command[1];

    if(command[3])          //if send break command
            assign txd=0; //set txd=0 (ignores/overrid
```

```
                else                        //later non-assign assignment
                    assign txd;

                if(command[4]) status[5:3]=0; //Clear Frame /Parity/Overrun
                rts_= !command[5];
                if(command[6]) ->resete; //internal reset
                if(modreg[1:0]==0 && command[7])
                begin //if sync mode and enter hunt
                    disable // disasble the sync receiver
                        syn_receive_internal;
                    disable syn_receive_external;
                        syn_receive_internal;
                    receivebuf=8'hff; //reset receive buffer 1's
                    ->start_receiver_e; // restart sync mode receiver
                end

                if(receive_in_progress==0)
                        ->start_receiver_e;

                repeat('TTXRDY) @(posedge clk);
                status[0]=1;
            end
            endcase
        end
    end
reg [7:0] serial_data;
reg  parity_bit;

always wait (tdata_out_full==1)
begin: trans1

    if(dflags[1])
        $display("I8251A (%h) at %d: transmitting data: %b", instance_id, $time, tdata_
        out);
    if(tdata_out_wait)          //if the data arrived any old time
        @(negedge txc_);        //wait for a negedge on txc_
                                // but if a stop bit was just sent
                                // do not wait
    serial_data=tdata_out;

    if (tstoptotal != 0) //if async mode ...
    begin
        txd=0;          //then send a start bit 1st
```

```
    repeat(baudmx) @(negedge txc_);
  end

  repeat(databits)        //send all start,databits
  begin
    txd=serial_data[0];
    repeat(baudmx) @(negedge txc_);
    serial_data=serial_data>>1;
  end

  if (modreg [4])          // if parity is enabled ...
  begin
    parity_bit=^ (tdata_out & data_mask);
    if(modreg[5]==0) parity_bit =~ parity_bit;     // odd parity

    txd=parity_bit;
    repeat(baudmx) @(negedge txc_); //then send the parity bit
  end

  if(tstoptotal != 0)        // if sync mode
  begin
    txd=1;        //then send out the stop bit (s
    repeat(tstoptotal) @(negedge txc_);
  end

  tdata_out_full=0;  //block this routine until data/sync char to be sent
                     // is immediately transferred to tdata_out.

->txende; //decide what data should be sent (data/sync/stop bit)
end

event transmit_held_data_e,
    transmitter_idle_e;

always @txende //end of transmitted data/sync character
begin : trans2
    case (command[0] &~ cts_)
    0 :          //if its is not now cts
                 //but data was received while it was c
    if (tdata_hold_full && tdata_hold_cts)
      -> transmit_held_data_e; // then send the data char
    else
      ->transmitter_idle_e; //else send sync char(s) or 1 stop bit
```

```
        1 :       //if its is now cts                    ;
        if (tdata _ hold _ full) // if a character has been received but now yet
                                 //transmitted ...
            ->transmit _ held _ data _ e;// then send the data char
        else          // else (no character has been received)
            -> transmitter _ idle _ e; // send sync char(s) or 1 stop bit
        endcase
end

always @ (transmitter _ idle _ e) //if there are no data chars to send ...,
begin : trans3
        status[2]=1;          // mard transmitter as being empty
        txe=1;
        if (tstoptotal !=0 || command[0] ==0 ||cts _ ==1)
        // if async mode or after areset or TxEnable = false or cts =false
        begin
           if (dflags[1])
                $display ("I8251A (%h) at %d : transmitting data :
                        1 (stop bit)   "instance _ id , $time);
           txd=1; //then send out 1 stop bit and make any writes
           tdata _ out=1; // go to tdata _ hold
           repeat(baudmx) @(negedge txc _);
           ->txende;
        end
        else       // if sync mode
        case (sync _ to _ transmit)
        1 :
              begin
              tdata _ out=sync1 >> (8-databits);
              tdata _ out _ wait=0; // without waiting on negedge t
              tdata _ out _ full=1;
              if(modreg[7] == 0) // if double sync mode
                sync _ to _ transmit =2;// send 2nd sync after 1st
              end
        2 :
              begin
              tdata _ out =sync2 >> (8-databits);
              tdata _ out _ wait =0 ; // without waiting on negedge t
              tdata _ out _ full =1 ;
              sync _ to _ transmit = 1; //send 1st sync char next
              end
        endcase
        end
```

```
always @ (transmit _ held _ data _ e) // if a character has been received
begin : trans4
      tdata _ out=tdata _ hold; // but not transmitted ...
      tdata _ out _ wait = 0;        // then do not wait on negedge txc
      tdata _ out _ full = 1;        // and send the char immediately
      tdata _ hold _ full = 0 ;
      repeat ('TTXRDY ) @(posedge clk);
      status[0] = 1;        // and set the txrdy status bit
end

//* * * * * * * * * * * RECEIVER PORTION OF THE 8251A * * * * * * * * *
             // data is received at leading edge of the clock
event   break _ detect _ e,    //
          break _ delay _ e;    //
event   hunt _ sync1 _ e, //hunt for the 1st sync char
          hunt _ sync2 _ e, //hunt for the 2nd sync char (double sync mode)
          sync _ hunted _ e, //sync char(s) was found (on abit aligned basis
          external _ syndet _ watche; //external sync mode : whenever syndet pin
                                      // goes high, set the syndet status bit
always @start _ receiver _ e
begin : rcv _ blk
    receive _ in _ progress = 1;
    case (modreg[1 : 0])
    2'b 00 :
        if (modreg[6] ==0) // if internal syndet mode ...
        begin
            if (dflags[5])
                $ display("I8251A (%h) at %d : starting internal sync receive",
                        instance _ id, $ time);
            if (dflags[5] && command[7])
                $ display("I8251A (%h) at %d : hunting for syncs", instance _ id, $ time);
            if (modreg[7]==1) // if enter hunt mode
            begin
                if(dflags[5])
                    $ display("I8251A (%h) at %d : receiver waiting on syndet",
                            instance _ id, $ time);
                ->hunt _ sync1 _ e; //start search for sync char(s
                @(posedge syndet);
                if(dflags[5])
                    $ display("I8251A (%h) at %d : receiver DONE
                            waiting on syndet", instance _ id, $ time);
            end
```

```
                syn _ receive _ internal; //start sync mode receiver
            end
            else
            begin
                if(dflags[5])
                    $ display("I8251A (%h) at %d : starting external sync receive",
instance _ id, $ time);
            if(dflags[5] && command[7])
                $ display("I8251A (%h) at %d : hunting for syncs",
                        instance _ id, $ time);
            ->external _ syndet _ watche; // whenever syndet pin goes to 1
                                          // set syndet status bit
            if (command[7]==1)
            begin : external _ syn _ hunt _ blk
                fork
                    syn _ receive _ external; // assemble chars while waiting
                    @(posedge syndet) // after rising edge of syndet
                    @(negedge syndet) // wait for falling edge
                                      // before starting char assemble
                    disable external _ syn _ hunt _ blk;
                join
            end
        syn _ receive _ external; // start external sync mode receiving
        end
    default : // if async mode ...
        begin
            if(dflags[5])
                $ display("I8251A (%h) at %d : starting asynchronous receiver",
instance _id, $ time);
            ->break _ detect _ e; // start check for rcd=0 too long
            asyn _ receive; // and start async mode receiver
        end
    endcase
end

        /* * * * * EXTERNAL SYNCHRONOUS MODE RECEIVE * * * * */
task syn _ receive _ rexternal;
forever
    begin
        repeat(databits) //Whether in hunt mode or not,assemble a character
        begin
            @(posedge rxc _)
            receivebuf={rcd, receivebuf[7 : 1]};
```

```
        end
        get _ and _ check _ parity; //receive and check parity bit, if any
        mark _ char _ received; //set rxrdy line, if enalbed
    end
endtask

always @(external _ syndet _ watche)
    @(posedge rxc _)
        status[6]=1;
        /* * * *  INTERNAL SYNCHRONOUS MODE RECEIVE    * * * /
        /*          Hunt for the sync char(s)                  * /
        /*          (if in synchronous internal sync detect mode)  * /
        /*         Syndet is set high when the sync(s) are found  * /

always @ (hunt _ sysnc1 _ e) //search for 1st sync char in the data stream
begin : sync _ hunt _ blk
    while(! (((receivebuf ^ sync1) & syncmask) === 8'b0000 _ 0000))
    begin
        @(posedge rxc _)
        receivebuf = {rcd, receivebuf[7 : 1]};
    end
    if (modreg[7]==0) // if double sync mod
        ->hunt _ sync2 _ e; //check for 2nd sync char directly agter 1
    else
        -> sync _ hunted _ e; // if single sync mode , sync hunt is complete
end
always @ (hunt _ sync2 _ e) // find the second synchronous character
begin : double _ sync _ hunt _ blk
    repeat(databits)
    begin
        @(posedge rxc _)
        receivebuf={rcd,receivebuf[7 : 1]};
    end
    if(((receivebuf ^ sync2)& syncmask===8'b0000 _ 0000)
        ->sync _ hunted _ e; // if sync2 followed syn1,sync hunt is complete
    else
        ->hunt _ sync1 _ e; //else hunt for sync1 again

    // Note : the data stream [sync1 sync1 sync2] will have sync detected.
    // Suppose sync1=11001100 :
    // Then [1100 1100 1100 sync2]will NOT be detected .
    // In general : never let a suffix of sync1 also be a prefix of sync1.
end
```

```
always @ (sync _ hunted _ e)
begin : parity _ sync _ hunt _ blk
      get _ and _ check _ parity;
      status[6]=1; //set syndet status bit (sync chars detected )
end

task syn _ receive _ internal;
forever
begin
   repeat(databits) //no longer in hunt mode so read entire chars and
   begin       // then look for syncs (instead of on bit boundaries)
         @(posedge rxc _)
         receivebuf={rcd,receivebuf[7 : 1]};
   end
   case (sync _ to _ receive)
   2 :         // if looking for 2nd sync char ...
   begin
      if(((receivebuf ^ sync2) & syncmask)===0)
      begin     //... and 2nd sync char is found
            sync _ to _ receive =1; //then look ofr 1st sync (or data)
            status[6]=1; // and mark sync detected
      end
      else if (((receivebuf ^ sync1) & syncmask)===0)
      begin //... and 1st sync char is found
      begin
         sync _ to _ receive = 2; //then look for 2nd sync char
      end
   end
   1 :
   begin
      if ((( receivebuf ^ sync1) & syncmask) ===0) // ... and 1st sync is found
      begin
         if(modreg[7]==0) //if doulbe sync mode
            sync _ to _ receive =2; // look for 2nd sync to foll
         else
              status[6]=1; //else look for 1st or data and mark sync detected
      end
      else; //and data was found , do nothing
   end
   endcase
   get _ and _ check _ parity; // receive and check parity bit, if any
   mark _ char _ received;
```

```
end
endtask

task get _ and _ check _ parity;
begin
    receivebuf=receivebuf >> (8-databits);
    if(modreg[4] == 1)
    begin
        @(posedge rxc _)
        if (( ^ receivebuf ^ modreg[5] ^ rcd) ! = 1)
            parity _ error;
    end
end
endtask

task mark _ char _ received;
begin
    if(command[2]==1) // if receiving is enabled
    begin
    rxrdy=1; //set receive read status bit
    status[1]=1; //if previous data was not read
    if(rdatain == 1)
      overrun _ error; // overrun error
    rdata=receivebuf; //latch the data
    rdatain=1; //mark data as not having been read
    end
if(dflags[2])
    $display("I8251A (%h) at %d : receive data : %b", instance _ id,
$time,receivebuf);
end
endtask

/* * * * * * * * * ASYNCHRONOUS MODE RECEIVER * * * * * * * */
/* CHECK FOR BREAK DETECTION (RCD LOW THROUGH 2 */
/* RECEIVE SEQUENCES IN THE ASYNCHRONOUS MODE . */

always @ (break _ detect _ e)
begin : break _ detect _ blk
      #1 /* to be sure break _ delay _ clk is waiting on break _ delay _ e
            after it triggered break _ detect _ e */
      if (rcd==0)
      begin
         ->break _ delay _ e; // start + databits +parity +stop bit
```

```
        breakcount_period = 1 +databits + modreg[4] + (tstoptotal! =0);
        // the number of rxc periods needed for 2 receive sequence
        breakcount_period = 2 * breakcount_period * baudmx;
        //if rcd stays low through 2 consecutive (start ,data,prity ,stop )sequences ...
        repeat(breakcount_period)
            @(posedge rxc_);
        status[6]=1; // ... then set break detect (status[6]) high
    end
end

always @(break_delay_e)
begin : break_delay_blk
    @(posedge rcd ) //but if rcd goes high during that time
    begin : break_delay_blk
        disable break_detect_blk;
        status[6] = 0; //... then set the break detect low
        @(negedge rcd ) //and when rcd goes low again ...
        ->break_detect_e; // ... start the break detection again
    end
end

/* * * * * ASYNCHRONOUS MODE RECEIVE TASK * * * * * * * * */
task asyn_receive;
forever
    @(negedge rcd) // the receive line went to zero, maybe a start bit
    begin
        rbaudcnt = baudmx /2;
        if (baudmx == 1)
            rbaudcnt=1;
        repeat(rbaudcnt) @(posedge rxc_); // after half a bit ...
        if(rcd == 0) //if it is still a start bit
        begin
            rbaudcnt = baudmx;
            repeat(databits) // receive the data bits
            begin
                repeat(rbaudcnt ) @(posedge rxc_);
                #1  receivebuf={rcd,receivebuf[7 : 1]};
            end
            repeat (rbaudcnt) @(posedge rxc_);

              //shift the data to the low part
            receivebuf = receivebuf >> (8-databits);
            if(modreg[4]==1) ///if parity is enabled
```

```
            begin
               if ((^ receivebuf ^ modreg[5]^ rcd)!=1)
                    parity_error; //check for a parity error
               repeat(rbaudcnt) @(posedge rxc_);
            end

            #1  if (rcd == 0 ) // if middle of stop bit is 0
                    frame_error; // frame error (should be 1)

            mark_char_received;
         end
      end
endtask
endmodule
```

参考文献

1 CADENCE Version 9502 Verilog-XL User Guide

2 CADENCE Version 9502 Verilog-XL Reference

3 CADENCE Version 9502 Verilog Constructs Used in Synthesis

4 CADENCE Version 9502 Verilog-XL Tutorial

5 Synplify-Lite Synthesis User's Guide

6 Silos III Simulator User's Guide

7 Quick Works Version 5.1 User's Guide with SpDE Reference

8 A Hardware Designer's Guide to Verilog

9 Verilog Hardware Description Language Reference Manual Version 2.0. Los Gatos (Califa.)c Open Verilog International,1993

10 Thomas D E, Moorby P R. The Verilog Hardware Description Language. 2nd ed. Boston(Mass.):Kluwer Academic Publishers, 1995

11 Sternheim, Singh, Madhavan. Digital Design and Synthesis with Verilog HDL. San Jose(Califa.): Automata Publishing Company,1993

12 刘宝琴.数字电路与系统. 北京：清华大学出版社,1993

13 阎石主编,清华大学电子学教研组编. 数字电子技术基础：高等学校教材.第3版. 北京：高等教育出版社,1994

[General Information]
书名=复杂数字电路与系统的Verilog HDL设计技术1998
作者=
页数=196
SS号=
DX号=
出版日期=
出版社=