从算法设计到硬线逻辑的实现

复杂数字逻辑系统的 VerilogHDL 设计技术和方法

夏宇闻　　编著

高等教学出版社
2000 年 9 月

内容简介

本书从算法和计算的基本概念出发，讲述把复杂算法逐步分解成简单的操作步骤，最后由硬线逻辑电路系统来实现该算法的技术和方法。这种硬线逻辑电路系统就是广泛应用于各种现代通讯电子设备与计算机系统中的专用集成电路（ASIC）或 FPGA。本书着重介绍进入九十年代后才开始在美国等先进的工业国家逐步推广的用硬件描述语言（Verilog HDL） 建模、仿真和综合的设计方法和技术。本书可作为电子或计算机类大学本科高年极和研究生的教材，也可供在数字系统设计领域工作的工程师参考或作为自学教材。

内容详介

本书中有关数字逻辑系统的设计方法采用了九十年代初才开始在美国等先进的工业国家逐步推广的硬件描述语言（Verilog HDL）Top Down 设计方法 。全书共分为十章，第一章为数字信号处理、计算、程序、算法和硬线逻辑的基本概念，第二章为 Verilog HDL 设计方法概述，第三章为 Verilog HDL 的基本语法，第四章为不同抽象级别的 Verilog HDL 模型，第五章为基本运算逻辑和它们的 Verilog HDL 模型，第六章为运算和数据流动控制逻辑，第七章为有限状态机和可综合风格的 Verilog HDL，第八章为可综合的 Verilog HDL 设计实例（简化的 RISC CPU 设计简介），第九章为虚拟器件和虚拟接口模块。第十章为设计练习进阶。每章后都附有思考题，可帮助读者加深理解该章讲述的概念和方法。在附录中还有符合 IEEE1364-95 标准的 VerilogHDL 语法的中文译本资料，可供参考。本书面向的对象是大学电子类和计算机工程类本科高年级学生和研究生，以及在专用数字电路与系统设计领域工作的工程师们。阅读本书所需的基础知识是数字电路基础、C 语言编程基础知识和最基本的信号处理知识。讲述的重点是数字电路与系统的 Verilog HDL 建模基本方法，其中包括用于仿真的和用于综合的模块建模。掌握了这种基本方法后，就可以设计极其复杂的硬线(hard-wired)数字逻辑电路与系统，如实时数字信号处理(DSP)电路系统。书中各章中都有大量的例题，可以帮助读者理解书中的基本概念并掌握从简单到非常复杂的各种风格模块的设计技术。因为本书的内容是独立于开发环境的，所以本书中并不介绍具体工具的使用，只介绍有关 Verilog HDL 建模、仿真和综合以及 TOP-DOWN 等现代设计思想、技术、方法和要点。本书的目的是用最少的经费尽快培养和造就一批掌握这种九十年代设计思想和方法的跨世纪人才。这些人才的涌现无疑会迅速地缩小我国与美国等技术先进国家在复杂数字系统设计领域的差距。

作者前言

数字信号处理(DSP)系统的研究人员一直在努力寻找各种优化的算法来解决相关的信号处理问题。当他们产生了比较理想的算法思路后，就在计算机上用 C 或其他语言，通过编写并运行程序来验证该算法，并不断修改程序以期完善，然后与别的算法作性能比较。在现代通信和计算机系统中对于 DSP 算法评价最重要的指标是看它能否满足工程上的需要。而许多工程上的需要都有实时响应的要求，也就是需要数字信号处理(DSP)系统在限定的时间内，如在几个毫秒甚至于几个微秒内，对所输入的大量数据完成相当复杂的运算，并输出结果。这时如果我们仅仅使用通用的微处理器，即使是专用于信号处理的微处理器，也往往无法满足实时响应的要求。我们不得不设计专用的高速硬线逻辑来完成这样的运算。设计这样的有苛刻实时要求的复杂的高速硬线运算逻辑是一件很有挑战性的工作，即使有了好的算法而没有好的设计工具和方法也很难完成。

近三十年来，我国在复杂数字电路设计技术领域与国外的差距越来越大。作为一名在大学讲授专用数字电路与系统设计课程的老师深深感到自己身上责任的重大。我个人觉得我国在这一技术领域的落后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同学们，也没有给他们创造实践的机会。

1995 年我受学校和系领导的委托，筹建世行贷款的电路设计自动化(EDA)实验室。 通过几年的摸索、实践，我们掌握了利用 Verilog HDL 设计复杂数字电路的仿真和综合技术。在此基础上我们为航天部等有关单位设计了十三万门左右卫星信道加密用复杂数字电路，提供给他们经前后仿真验证的 Verilog HDL 源代码，得到很高的评价。在其后的几年中又为该单位设计了十万门左右卫星下行信道 RS(255,223)编码/解码电路和三万门左右卫星上行信道 BCH(64,56) 编码/解码电路，这几个项目已先后通过有关单位的验收。我们也为我们自己的科研项目，小波(Wavelet)图象压缩，成功地设计了小波卷积器和改进的零修剪树算法(即 SPIHT 算法)的硬线逻辑的 Verilog HDL 模型，不但成功地进行了仿真和综合，而且制成的可重配置硬线逻辑（采用 ALTERA FLEX10K 系列 CPLD /10 /30 /50 各一片）的 PCI 线路板，能完成约 2000 条 C 语句的程序才能完成的图象压缩/解压缩算法。运算结果与软件完成的完全一致，而且速度比用 PII-333MHZ CPU 的 PC 机更快，而 PCI 线路板上基本时钟仅为 8.33MHZ。可见这种新的设计方法的潜力。

本书是在 1998 年由北航出版社出版的教材基础上补充而成的。由于教学、科研和实验室的各项工作很忙，我只能利用零碎时间，一点一点地把积累的教学经验和新收集到的材料补充输入到计算机中的原教材中并逐步加以整理。到现在又两年又过去了，新版书总算有了一个初步的样子。我把书名改为“从算法设计到硬线逻辑的实现-*复杂数字电路与系统的 Verilog HDL 设计技术*”，这是因为本书是围绕着算法的实现来介绍 Verilog HDL 设计方法的。因为我们使用 Verilog HDL 设计复杂数字逻辑电路总共也只有五年的时间，水平并不高，书中谬误之处在所难免，敬请读者及时把意见反馈给我。我之所以匆匆把这本书改版推出，是想把我们在采用 Verilog HDL 设计方法上新积累的一些经验与读者分享，把原教材中一些不足的地方作一些补充和修改。在大学生和研究生中加快 Verilog HDL 设计技术的推广，尽快培养一批掌握先进设计技术的跨世纪的人才。期望本书能在这一过程中起到抛砖引玉的作用。

回想起来，这本书实质上是我们实验室全体老师和同学们的劳动成果，其中在 EDA 实验室工作过的历届研究生和本科生：张琰、山岗、王静璇、田玉文、冯文楠、杨柳、龚剑、傅红军、王书龙和胡英等都帮我做了许多工作，如部分素材的翻译、整理、录入和一些 Verilog HDL 模块的设计和验证。而我做的工作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并结合教学经验把它们编写成通顺的段落，以及全书文稿最后的组织、整理和补充，使其达到能够出版的标准。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鼓励和鞭策，使我不能懈怠。在本书出版之际，我衷心地感谢在编写本书过程中所有给过我帮助和鼓励的老师和同学们。

编者

2000 年 8 月 30 日

于北京航空航天大学逸夫科学馆 EDA 实验室

目录

第一章 数字信号处理、计算、程序、算法和硬线逻辑的基本概念

引言：

现代计算机与通讯系统电子设备中广泛使用了数字信号处理专用集成电路，它们主要用于数字信号传输中所必需的滤波、变换、加密、解密、编码、解码、纠检错、压缩、解压缩等操作。这些处理工作从本质上说都是数学运算。从原则上讲，它们完全可以用计算机或微处理器来完成。这就是为什么我们常用 C、Pascal 或汇编语言来编写程序，以研究算法的合理性和有效性的道理。

在数字信号处理的领域内有相当大的一部分工作是可以事后处理的。我们可以利用通用的计算机系统来处理这类问题。如在石油地质调查中，我们通过钻探和一系列的爆破，记录下各种地层的回波数据，然后用计算机对这些数据进行处理，去除噪声等无用信息，最后我们可以得到地层的构造，从而找到埋藏的石油。因为地层不会在几年内有明显的变化，因此花几十天的时间把地层的构造分析清楚也能满足要求。这种类型的数字信号处理是非实时的，用通用的计算机就能满足需要。

还有一类数字信号处理必须在规定的时间内完成，如在军用无线通信系统和机载雷达系统中我们常常需要对检测到的微弱信号增强、加密、编码、压缩，在接收端必须及时地解压缩、解码和解密并重现清晰的信号。我们很难想象用一个通用的计算机系统来完成这项工作，因此，我们不得不自行设计非常轻便小巧的高速专用硬件系统来完成该任务。

有的数字信号处理对时间的要求非常苛刻，以至于用高速的通用微处理器芯片也无法在规定的时间内完成必须的运算。我们必须为这样的运算设计专用的硬线逻辑电路，这可以在高速 FPGA 器件上实现或制成高速专用集成电路。这是因为通用微处理器芯片是为一般目的而设计的，运算的步骤必须通过程序编译后生成的机器码指令加载到存贮器中，然后在微处理器芯片控制下，按时钟的节拍，逐条取出指令、分析指令，然后执行指令，直至程序的结束。微处理器芯片中的内部总线和运算部件也是为通用的目的而设计，即使是专为信号处理而设计的通用微处理器，因为它的通用性，也不可能为某一个特殊的算法来设计一系列的专用的运算电路，而且其内部总线的宽度也不能随意改变，只有通过改变程序，才能实现这个特殊的算法。因而其运算速度就受到限制。

本章的目的是想通过对数字信号处理、计算（Computing）、算法和数据结构、编程语言和程序、体系结构和硬线逻辑等基本概念的介绍，了解算法与硬线逻辑之间的关系从而引入利用 Verilog HDL 硬件描述语言设计复杂的数字逻辑系统的概念和方法。向读者展示一种九十年代才真正开始在美国等先进的工业国家逐步推广的数字逻辑系统的设计方法。借助于这种方法，在电路设计自动化仿真和综合工具的帮助下，只要我们对并行的计算结构有

一定程度的了解，对有关算法有深入的研究，我们完全有能力设计并制造出有自己知识产权的 DSP（数字信号处理）类和任何复杂的数字逻辑集成电路芯片，为我国的电子工业和国防现代化作出应有的贡献。

1.1 数字信号处理

大规模集成电路设计制造技术和数字信号处理技术，近三十年来，各自得到了迅速的发展。这两个表面上看来没有什么关系的技术领域实质上是紧密相关的。因为数字信号处理系统往往要进行一些复杂的数学运算和数据的处理，并且又有实时响应的要求，它们通常是由高速专用数字逻辑系统或专用数字信号处理器所构成，电路是相当复杂的。因此只有在高速大规模集成电路设计制造技术进步的基础上，才有可能实现真正有意义的实时数字信号处理系统。对实时数字信号处理系统的要求不断提高，也推动了高速大规模集成电路设计制造技术的进步。现代专用集成电路的设计是借助于电子电路设计自动化（EDA）工具完成的。学习和掌握硬件描述语言（HDL）是使用电子电路设计自动化（EDA）工具的基础。

1.2 计算（Computing）

说到数字信号处理，我们自然就会想到数学计算（或数学运算）。现代计算机和通信系统中广泛采用了数字信号处理的技术和方法。基本思路是先把信号用一系列的数字来表示，如是连续的模拟信号，则需通过采样和模拟数字转换，把信号转换成一系列的数字信号，然后对这些数字信号进行各种快速的数学运算，其目的是多种多样的，有的是为了加密，有的是通过编码来减少误码率以提高信道的通信质量，有的是为了去掉噪声等无关的信息也可以称为滤波，有的是为了数据的压缩以减少占用的频道...。有时我们也把某些种类的数字信号处理运算称为变换如离散傅利叶变换(DFT)、离散余弦变换(DCT)、小波变换(Wavelet T）等。

我们这里所说的计算是从英语 Computing 翻译过来的，它的含义要比单纯的数学计算广泛得多。“Computing 这门学问研究怎样系统地有步骤地描述和转换信息，实质上它是一门覆盖了多个知识和技术范畴的学问，其中包括了计算的理论、分析、设计、效率和应用。它提出的最基本的问题是什么样的工作能自动完成，什么样的不能。”（摘自 Denning et al., “Computing as a Discipline,” Communication of ACM, January,1989)。

本文中凡提到计算这个词处，指的就是上面一段中 Computing 所包含的意思。由传统的观点出发，我们可以从三个不同的方面来研究计算，即从数学、科学和工程的不同角度。
由比较现代的观点出发，我们可以从四个主要的方面来研究计算，即从算法和数据结构、编程语言、体系结构、软件和硬件设计方法学。本课本的主题是从算法到硬线逻辑的实现，因此我们将从算法和数据结构、编程语言和程序、体系结构和硬线逻辑以及设计方法学等方面的基本概念出发来研究和探讨用于数字信号处理等领域的复杂硬线逻辑电路的设计技术和方法。特别强调利用 Verilog 硬件描述语言的 Top-Down 设计方法的介绍。

1.3 算法和数据结构

为了准确地表示特定问题的信息并顺利地解决有关的计算问题，我们需要采用一些特殊方法并建立相应的模型。所谓算法就是解决特定问题的有序步骤，所谓数据结构就是解决特定问题的相应的模型。

1.4 编程语言和程序

程序员利用一种由专家设计的既可以被人理解，也可以被计算机解释的语言来表示算法问题的求解过程。这种语言就是编程语言。由它所表达的算法问题的求解过程就是程序。我们已经熟悉通过编写程序来解决计算问题， C、Pascal、Fortran、Basic 或汇编语言语言是几种常用的编程语言。如果我们只研究算法，只在通用的计算机上运行程序或利用通用的 CPU 来设计专用的微处理器嵌入系统，掌握上述语言就足够了。如果还需要设计和制造能进行快速计算的硬线逻辑专用电路，我们必须学习数字电路的基本知识和硬件描述语言。因为现代复杂数字逻辑系统的设计都是借助于 EDA 工具完成的，无论电路系统的仿真和综合都需要掌握硬件描述语言。在本书中我们将要比较详细地介绍 Verilog 硬件描述语言。

1.5 系统结构和硬线逻辑

计算机究竟是如何构成的？为什么它能有效地和正确地执行每一步程序？它能不能用另外一种结构方案来构成？运算速度还能不能再提高？所谓计算机系统结构就是回答以上问题并从硬线逻辑和软件两个角度一起来探讨某种结构的计算机的性能潜力。比如， Von Neumann（冯诺依曼）在 1945 设计的 EDVAC 电子计算机，它的结构是一种最早的顺序机执行标量数据的计算机系统结构。顺序机是从位串行操作到字并行操作，从定点运算到浮点运算逐步改进过来的。由于 Von Neumann 系统结构的程序是顺序执行的，所以速度很慢。随着硬件技术的进步，不断有新的计算机系统结构产生，其计算性能也在不断提高。计算机系统结构是一门讨论和研究通用的计算机中央处理器如何提高运算速度性能的学问。对计算机系统结构的深入了解是设计高性能的专用的硬线逻辑系统的基础，因此将是本书讨论的重点之一。但由于本书的重点是利用 Verilog HDL 进行复杂数字电路的设计技术和方法，大量的篇幅将介绍利用 HDL 进行设计的步骤、语法要点、可综合的风格要点、同步有限状态机和由浅入深的设计实例。

1.6 设计方法学

复杂数字系统的设计是一个把思想（即算法）转化为实际数字逻辑电路的过程。我们都知道同一个算法可以用不同结构的数字逻辑电路来实现，从运算的结果说来可能是完全一致的，但其运算速度和性能价格比可以有很大的差别。我们可用许多种不同的方案来实现能实时完成算法运算的复杂数字系统电路，下面列出了常用的四种方案：1)以专用微处理机芯片为中心来构成完成算法所需的电路系统；2)用高密度的 FPGA（从几万门到百万门）；3）

设计专用的大规模集成电路（ASIC)；4）利用现成的微处理机的 IP 核并结合专门设计的高速 ASIC 运算电路。究竟采用什么方案要根据具体项目的技术指标、经费、时间进度和批量综合考虑而定。

在上述第二、第三、第四种设计方案中，电路结构的考虑和决策至关重要。有的电路结构速度快，但所需的逻辑单元多，成本高；而有的电路结构速度慢，但所需的逻辑单元少，成本低。复杂数字逻辑系统设计的过程往往需要通过多次仿真，从不同的结构方案中找到一种符合工程技术要求的性能价格比最好的结构。一个优秀的有经验的设计师，能通过硬件描述语言的顶层仿真较快地确定合理的系统电路结构，减少由于总体结构设计不合理而造成的返工，从而大大加快系统的设计过程。

1.7 专用硬线逻辑与微处理器的比较

在信号处理专用计算电路的设计中，以专用微处理器芯片为中心来构成完成算法所需的电路系统是一种较好的办法。我们可以利用现成的微处理器开发系统，在算法已用 C 语言验证的基础上，在开发系统工具的帮助下，把该 C 语言程序转换为专用微处理器的汇编再编译为机器代码，然后加载到样机系统的存储区，即可以在开发系统工具的环境下开始相关算法的运算仿真或运算。采用这种方法，设计周期短、可以利用的资源多，但速度、能耗、体积等性能受该微处理器芯片和外围电路的限制。

用高密度的 FPGA（从几万门到几十万门）来构成完成算法所需的电路系统也是一种较好的办法。我们必须购置有关的 FPGA 开发环境、布局布线和编程工具。有些 FPGA 厂商提供的开发环境不够理想，其仿真工具和综合工具性能不够好，我们还需要利用性能较好的硬件描述语言仿真器、综合工具，才能有效地进行复杂的 DSP 硬线逻辑系统的设计。由于 FPGA 是一种通用的器件，它的基本结构决定了对某一种特殊应用，性能不如专用的 ASIC 电路。

采用自行设计的专用 ASIC 系统芯片（System On Chip)， 即利用现成的微处理机 IP 核或根据某一特殊应用设计的微处理机核（也可以没有微处理机核)，并结合专门设计的高速 ASIC 运算电路，能设计出性能价格比最高的理想数字信号处理系统。这种方法结合了微处理器和专用的大规模集成电路的优点，由于微处理器 IP 核的挑选结合了算法和应用的特点，又加上专用的 ASIC 在需要高速部分的增强，能“量体裁衣”， 因而各方面性能优越。但由于设计和制造周期长、投片成本高，往往只有经费充足、批量大的项目或重要的项目才采用这一途径。当然性能优良的硬件描述语言仿真器、综合工具是不可缺少的，另外对所采用的半导体厂家基本器件库和 IP 库的深入了解也是必须的。

以上所述算法的专用硬线逻辑实现都需要对算法有深入的了解，还需掌握硬件描述语言和相关的 EDA 仿真、综合和布局布线工具。

1.8 C 语言与硬件描述语言在算法运算电路设计的关系和作用

数字电路设计工程师一般都学习过编程语言、数字逻辑基础、各种 EDA 软件工具的使用。

就编程语言而言，国内外大多数学校都以 C 语言为标准，只有少部分学校使用 Pascal 和 Fortran。

算法的描述和验证常用 C 语言来做。例如要设计 Reed-Solomen 编码/解码器，我们必须先深入了解 Reed-Solomen 编码/解码的算法，再编写 C 语言的程序来验证算法的正确性。运行描述编码器的 C 语言程序，把在数据文件中的多组待编码的数据转换为相应的编码后数据并存入文件。再编写一个加干扰用的 C 语言程序，用于模拟信道。它能产生随机误码位（并把误码位个数控制在纠错能力范围内）将其加入编码后的数据文件中。运行该加扰程序，产生带误码位的编码后的数据文件。然后再编写一个解码器的 C 语言程序，运行该程序把带误码位的编码文件解码为另一个数据文件。只要比较原始数据文件和生成的文件便可知道编码和解码的程序是否正确（能否自动纠正纠错能力范围内的错码位）。用这种方法我们就可以来验证算法的正确性。但这样的数据处理其运行速度只与程序的大小和计算机的运行速度有关，也不能独立于计算机而存在。如果要设计一个专门的电路来进行这种对速度有要求的实时数据处理，除了以上介绍的 C 程序外，还须编写硬件描述语言（如 Verilog HDL 或 VHDL）的程序，进行仿真以便从电路结构上保证算法能在规定的时间内完成，并能与前端和后端的设备或器件正确无误地交换数据。

用硬件描述语言（HDL）的程序设计硬件的好处在于易于理解、易于维护、调试电路速度快、有许多的易于掌握的仿真、综合和布局布线工具，还可以用 C 语言配合 HDL 来做逻辑设计的前后仿真，验证功能是否正确。

在算法硬件电路的研制过程中，计算电路的结构和芯片的工艺对运行速度有很大的影响。所以在电路结构确定之前，必须经过多次仿真：

1) C 语言的功能仿真。
2) C 语言的并行结构仿真。
3) Verilog HDL 的行为仿真。
4) Verilog HDL RTL 级仿真。
5) 综合后门级结构仿真。
6) 布局布线后仿真。
7) 电路实现验证。

下面介绍用 C 语言配合 Verilog HDL 来设计算法的硬件电路块时考虑的三个主要问题：

- 为什么选择 C 语言与 Verilog HDL 配合使用？
- C 语言与 Verilog HDL 的使用有何限制？
- 如何利用 C 来加速硬件的设计和故障检测？

1） **为什么选择 C 语言与 Verilog 配合使用**

首先，C 语言很灵活，查错功能强，还可以通过 PLI（编程语言接口）编写自己的系统任务直接与硬件仿真器（如 Verilog-XL）结合使用。C 语言是目前世界上应用最为广泛的一种编程语言，因而 C 程序的设计环境比 Verilog HDL 的完整。此外，C 语言

可应用于许多领域，有可靠的编译环境，语法完备，缺陷较少。 比较起来， Verilog 语言只是针对硬件描述的，在别处使用（如用于算法表达等）并不方便。而且 Verilog 的仿真、综合、查错工具等大部分软件都是商业软件，与 C 语言相比缺乏长期大量的使用，可靠性较差，亦有很多缺陷。所以，只有在 C 语言的配合使用下，Verilog 才能更好地发挥作用。

面对上述问题，最好的方法是 C 语言与 Verilog 语言相辅相成，互相配合使用。这就是既要利用 C 语言的完整性，又要结合 Verilog 对硬件描述的精确性，来更快更好地设计出符合性能要求的硬件电路系统。利用 C 语言完善的查错和编译环境，设计者可以先设计出一个功能正确的设计单元，以此作为设计比较的标准。然后，把 C 程序一段一段地改写成用并型结构（类似于 Verilog）描述的 C 程序，此时还是在 C 的环境里，使用的依然是 C 语言。如果运行结果都正确，就将 C 语言关键字用 Verilog 相应的关键字替换，进入 Verilog 的环境。将测试输入同时加到 C 与 Verilog 两个单元，将其输出做比较。这样很容易发现问题的所在，然后更正，再做测试，直至正确无误。剩下的工作就交给后面的设计工程师继续做。

2） **C 语言与 Verilog 语言互相转换中存在的问题**

这样的混合语言设计流程往往会在两种语言的转换中会遇到许多难题。例如， 怎样把 C 程序转换成类似 Verilog 结构的 C 程序，来增加并行度，以保证用硬件实现时运行速度达到设计要求；又如怎样不使用 C 中较抽象的语法：例如迭代，指针，不确定次数的循环等等，也能来表示算法（因为转换的目的是要用可综合的 Verilog 语句来代替 C 程序中的语句，而可用于综合的 Verilog 语法是相当有限的，往往找不到相应的关键字来替换）。

C 程序是一行接一行依次执行的，属于顺序结构，而 Verilog 描述的硬件是可以在同一时间同时运行的，属于并行结构。这两者之间有很大的冲突。而 Verilog 的仿真软件也是顺序执行的，在时间关系上同实际的硬件是有差异的，可能会出现一些无法发现的问题。

Verilog 可用的输出输入函数很少。C 语言的花样则很多，转换过程中会遇到一些困难。C 语言的函数调用与 Verilog 中模块的调用也有区别。C 程序调用函数是没有延时特性的，一个函数是唯一确定的，对同一个函数的不同调用是一样的。而 Verilog 中对模块的不同调用是不同的，即使调用的是同一个模块，必须用不同的名字来指定。Verilog 的语法规则很死，限制很多，能用的判断语句有限。仿真速度较慢，查错功能差，错误信息不完整。仿真软件通常也很昂贵，而且不一定可靠。C 语言没有时间关系，转换后的 Verilog 程序必须要能做到没有任何外加的人工延时信号，也就是必须表达为有限状态机，即 RTL 级的 Verilog，否则将无法使用综合工具把 Verilog 源代码转化为门级逻辑。

3） **如何利用 C 语言来加快硬件的设计和查错**

下表中列出了常用的 C 与 Verilog 相对应的关键字与控制结构

C	Verilog
sub-function	module, function, task
if-then-else	if-then-else
Case	Case
{,}	begin, end
For	For
While	While
Break	Disable
Define	Define
Int	Int
Printf	monitor, display,strobe

下表中，列出了 C 与 Verilog 相对应的运算符

C	Verilog	功能
*	*	乘
/	/	除
+	+	加
-	-	减
%	%	取模
!	!	反逻辑
&&	&&	逻辑且
‖	‖	逻辑或
>	>	大于
<	<	小于
>=	>=	大于等于
<=	<=	小于等于
==	==	等于
!=	!=	不等于
~	~	位反相
&	&	按位逻辑与
|	|	按位逻辑或
^	^	按位逻辑异或
~^	~^	按位逻辑同或
>>	>>	右移
<<	<<	左移
?:	?:	同等於 if-else 敘述

从上面的讨论我们可以总结如下：

- C 语言与 Verilog 硬件描述语言可以配合使用，辅助设计硬件
- C 语言与 Verilog 硬件描述语言很象，只要稍加限制，C 语言的程序很容易转成 Verilog 的程序

美国和中国台湾地区逻辑电路设计和制造厂家大都以 Verilog HDL 为主，中国大陆地区目前学习使用 VHDL 的较多。到底选用 VHDL 或是 Verilog HDL 來配合 C 一起用，就留給各位自行去决定。但从学习的角度来看，Verilog HDL 比較簡單，也與 C 语言较接近，容易掌握。从使用的角度，支持 Verilog 硬件描述语言的半导体厂家也较支持 VHDL 的多。

总结：

本章介绍了信号处理与硬线逻辑设计的关系，以及有关的基本概念。引入了 Verilog HDL 硬件描述语言，向读者展示一种九十年代才真正开始在美国等先进的工业国家逐步推广的数字逻辑系统的设计方法。借助于这种方法，在电路设计自动化仿真和综合工具的帮助下，我们完全有能力设计并制造出有自己知识产权的 DSP（数字信号处理）类和任何复杂的数字逻辑集成电路芯片，为我国的电子工业和国防现代化作出应有的贡献。在下面的各章里我们将分步骤地详细介绍这种设计方法。

思考题：

1) 什么是信号处理电路？
2) 为什么要设计专用的信号处理电路？
3) 什么是实时处理系统？
4) 为什么要用硬件描述语言来设计复杂的算法逻辑电路？
5) 能不能完全用 C 语言来代替硬件描述语言进行算法逻辑电路的设计？
6) 为什么在算法逻辑电路的设计中需要用 C 语言和硬件描述语言配合使用来提高设计效率？

第二章 Verilog HDL设计方法概述

前言

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其复杂程度也越来越高，特别是需要设计具有实时处理能力的信号处理专用集成电路，并把整个电子系统综合到一个芯片上。设计并验证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。由于电路制造工艺技术进步非常迅速，电路设计能力赶不上技术的进步。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，这样就能把系统设计工作分解为逻辑设计（前端）和电路实现（后端）两个互相独立而又相关的部分。由于逻辑设计的相对独立性就可以把专家们设计的各种常用数字逻辑电路和系统部件（如FFT算法、DCT算法部件）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。电路的实现则可借助于综合工具和布局布线工具（与具体工艺技术有关）来自动地完成。

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，使我国在复杂数字电路及系统的设计竞争中逐步缩小与美国等先进的工业发达国家的差距。为我国下一个世纪的深亚微米百万门级的复杂数字逻辑电路及系统的设计培养一批技术骨干。

2.1.硬件描述语言HDL(Hardware Description Language)

硬件描述语言(HDL)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。它可以使数字逻辑电路设计者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(在下面简称为EDA)工具进行仿真，再自动综合到门级电路，再用ASIC或FPGA实现其功能。目前，这种称之为高层次设计(High-Level-Design)的方法已被广泛采用。据统计，在美国硅谷目前约有90％以上的ASIC和FPGA已采用硬件描述语言方法进行设计。

硬件描述语言的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年代时，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和推动作用。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。把硬件描述语言用于自动综合还只有短短的六、七年历史。最近三四年来，用综合工具把可综合风格的HDL模块自动转换为电路发展非常迅速，在美国已成为设计数字电路的主流。本书主要介绍如何来编写可综合风格的Verilog HDL模块，如何借助于Verilog语言对所设计的复杂电路进行全面可靠的测试。

2.2.Verilog HDL的历史

2.2.1.什么是Verilog HDL

Verilog HDL是硬件描述语言的一种，用于数字电子系统设计。它允许设计者用它来进行各种级别的逻辑设计，可以用它进行数字逻辑系统的仿真验证、时序分析、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60000人，全美国有200多所大学教授用 Verilog 硬件描述语言的设计方法。在我国台湾地区几乎所有著名大学的电子和计算机工程系都讲授Verilog有关的课程。

2.2.2.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，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：即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（*在本书的附录中有该标准的中文翻译，可供同学参考*）。

下面两幅图显示出Verilog的发展历史和将来。

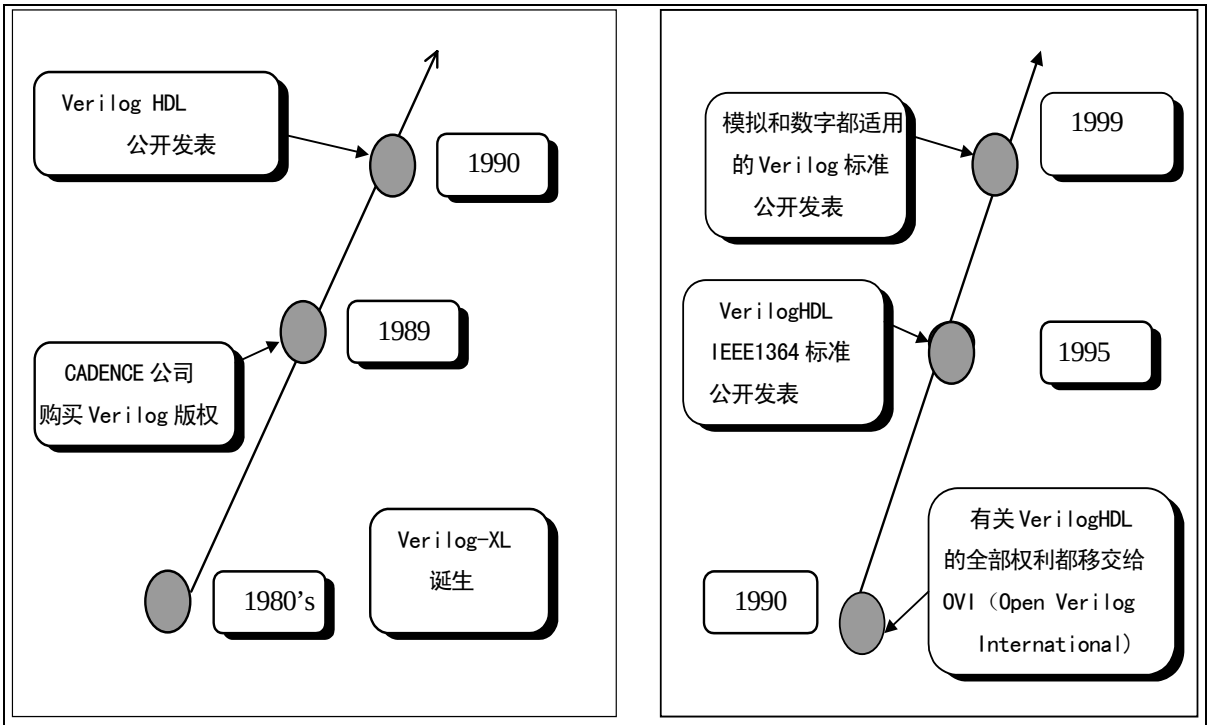


图 2-2 Verilog HDL 的发展历史和未来

2.3.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，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 则是从一个普通的民间公司的私有财产转化而来，基于Verilog HDL的优越性，才成为的IEEE标准，因而有更强的生命力。

VHDL 其英文全名为VHSIC Hardware Description Language,而VHSIC则是Very High Speed Integerated Circuit的缩写词，意为甚高速集成电路,故VHDL其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，至今已有十三年的应用历史，因而Verilog HDL拥有更广泛的设计群体，成熟的资源也远比VHDL

丰富。与VHDL相比Verilog HDL的最大优点是：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，通过二十学时的学习，再加上一段实际操作，一般同学可在二至三个月内掌握这种设计技术。而掌握VHDL设计技术就比较困难。这是因为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。下面图1-3是Verilog HDL和VHDL建模能力的比较图示供读者参考:

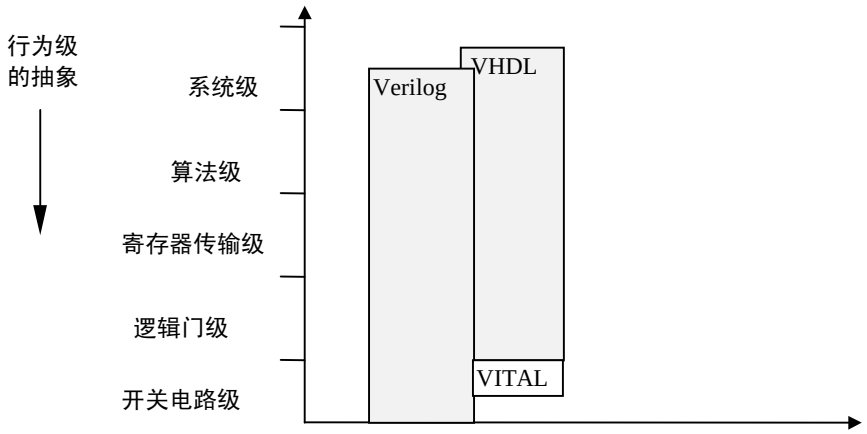


图 2—3 VerilogHDL 与 VHDL 建模能力的比较

但这两种语言也是在不断的完善过程中，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使同学们对数字电路设计技术有更进一步的了解，而且可以为以后学习高级的系统综合打下坚实的基础。

2.4. Verilog HDL目前的应用情况和适用的设计

几年以来，EDA界一直对在数字逻辑设计中究竟采用哪一种硬件描述语言争论不休，目前的情况是两者不相上下。在美国，在高层逻辑电路设计领域Verilog HDL和VHDL的应用比率是60％和40％，在台湾省各为50％，在中国大陆目前由于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有强得多得功能，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，因此在混合信号的电路系统的设计中，它必将会有更广泛的应用。在亚微米和深亚微米ASIC和高密度FPGA已成为电子设计主流的今天，Verilog HDL的发展前景是非常远大的。作者本人的意见是：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(Alogrithem)、寄存器传输级(RTL)、逻辑级(Logic)、门级(Gate)、电路开关级(Switch)设计，而对于特大型（几百万门级以上）的系统级(System)设计,则VHDL更为适合，由于这两种HDL语言还在不断地发展过程中，它们都会逐步地完善自己。

2.5.采用Verilog HDL设计复杂数字电路的优点

2.5.1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，工程师往往需要花好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大大延长了设计周期。

近年来，FPGA和ASIC的设计在规模和复杂度方面不断取得进展，而对逻辑电路及系统的设计的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。

2.5.2. Verilog HDL设计法与传统的电路原理图输入法的比较

如 2.5.1.所述采用电路原理图输入法进行设计，具有设计的周期长，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的厂家的不同的芯片中去，并在不同规模应用时可以较容易地作修改。这不仅是因为用Verilog HDL所完成的设计，它的信号位数是很容易改变的，可以很容易地对它进行修改，来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，而且还因为采用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。

2.5.3. Verilog HDL的标准化与软核的重用

Verilog HDL是在1983年由GATEWAY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，因而大大提高了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5000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综合工具与其它外部逻辑结合为一体。这样，软核和虚拟器件的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的组织（Virtual Socket Interface Alliance）来协调这方面的工作。

2.5.4.软核、固核和硬核的概念以及它们的重用

上一节中我们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5000门以上电路结构编码文件，称之为“固核”。

我们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5000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综合工具与其它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它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有必要的。我们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。

2.6.采用硬件描述语言(Verilog HDL)的设计流程简介

2.6.1.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乃至数百万个器件成为可能，但我们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对其所做的设计进行验证。图1-6-1为自顶向下（TOP-DOWN）的示意图，以设计树的形式绘出。

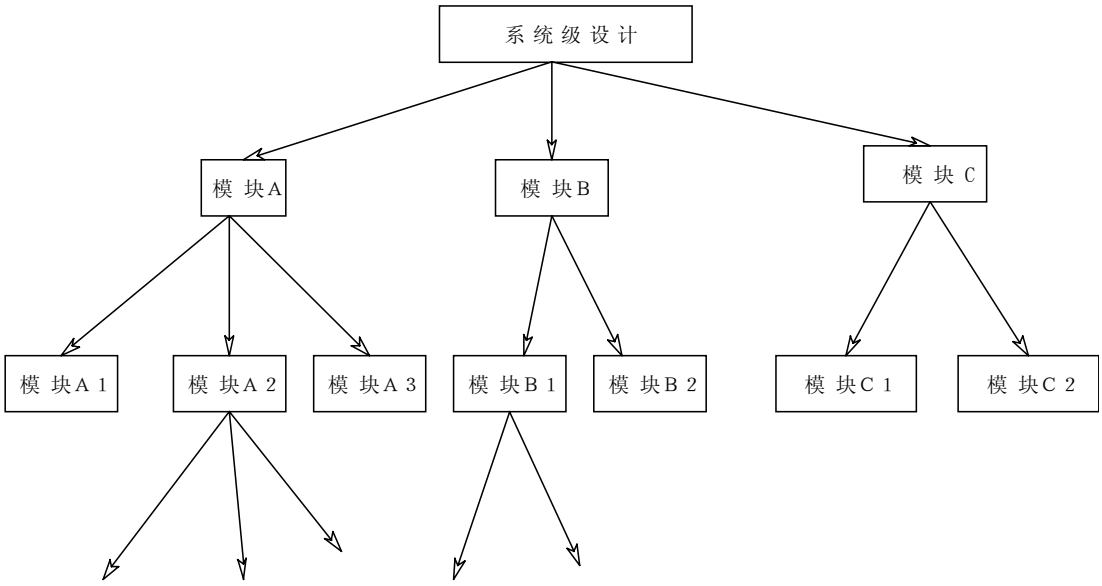


图2-6-1. TOP_DOWN设计思想

自顶向下的设计（即TOP_DOWN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。

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，先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。

2.6.2.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，该层次的硬件的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2-6-1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。

2.6.3.具体模块的设计编译和仿真的过程

在不同的层次做具体模块的设计所用的方法也有所不同，在高层次上往往编写一些行为级的模块通过仿真加以验证，其主要目的是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。下面的流程图简要地说明了模块的编译和测试过程：

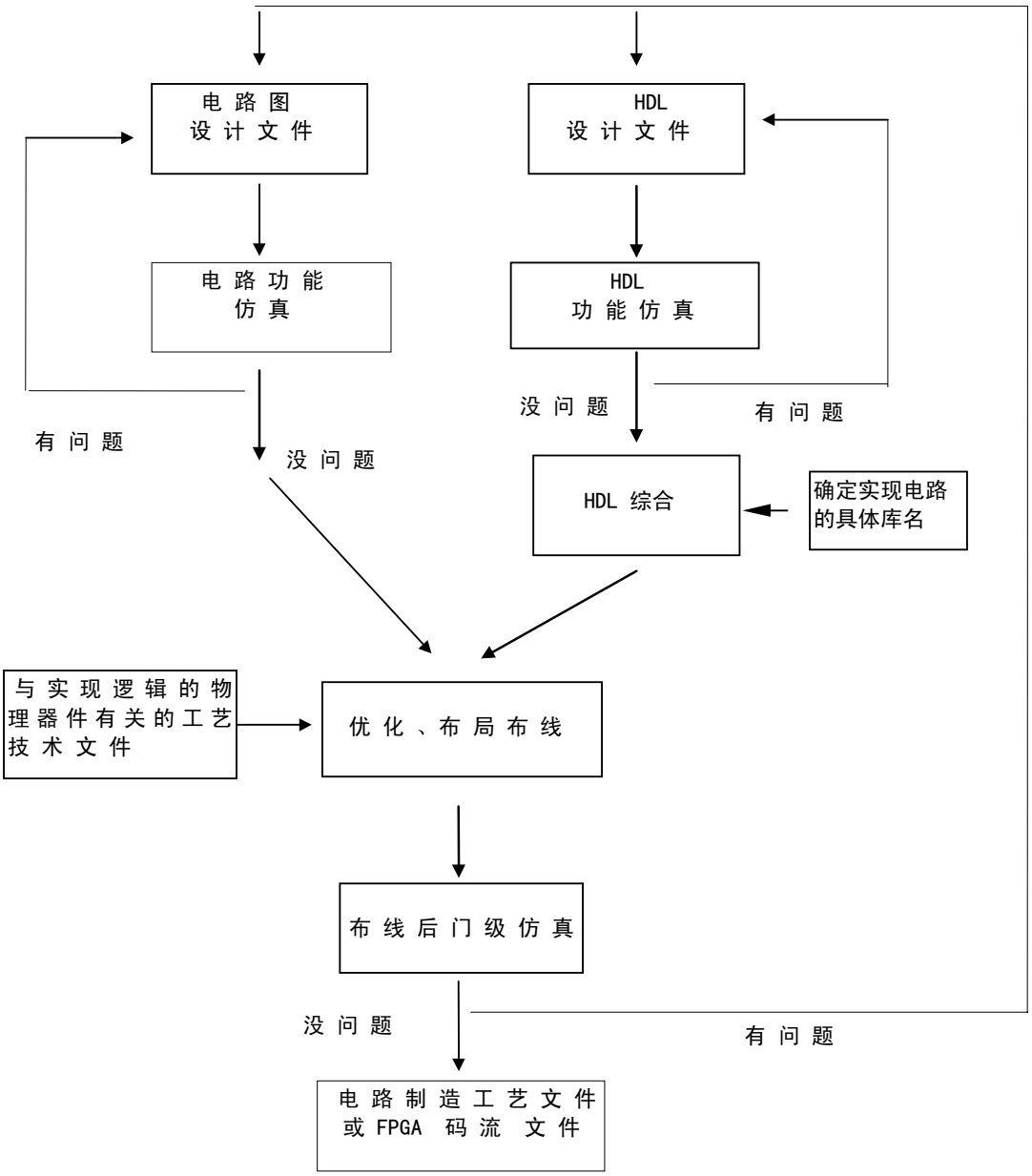


图 2-6-3 HDL 设 计 流 程 图

从上图可以看出，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-->综合到布局布线-->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。

2.6.4.对应具体工艺器件的优化、映象、和布局布线

由于各种ASIC和FPFA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，就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。

2.7.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九十年代以后纷纷采用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，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，要么是其它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。

2.8 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？它门各有什么特点？
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（HDL）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。

第三章 Verilog HDL的基本语法

前言

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。这也就是说，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对应的模型类型共有以下五种：

- 系统级(system):用高级语言结构实现设计模块的外部性能的模型。
- 算法级(algorithm):用高级语言结构实现设计算法的模型。
- 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。
- 门级(gate-level):描述逻辑门以及逻辑门之间的连接的模型。
- 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。

一个复杂电路系统的完整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。其中有些模块需要综合成具体电路，而有些模块只是与用户所设计的模块交互的现存电路或激励信号源。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，并对所作设计的逻辑电路进行严格的验证。

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：

- 可描述顺序执行或并行执行的程序结构。
- 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。
- 通过命名的事件来触发其它过程里的激活行为或停止行为。
- 提供了条件、if-else、case、循环程序结构。
- 提供了可带参数且非零延续时间的任务(task)程序结构。
- 提供了可定义新的操作符的函数结构(function)。
- 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：
  - 提供了完整的一套组合型原语(primitive);
  - 提供了双向通路和电阻器件的原语;
  - 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，我们只要对Verilog HDL某些语句的特殊方面着重理解，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。下面我们将对Verilog HDL中的基本语法逐一加以介绍。

3.1.简单的Verilog HDL模块

3.1.1.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程序的特性。

```
例[3.1.1]: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
```

这个例子通过连续赋值语句描述了一个名为adder的三位加法器可以根据两个三比特数a、b和进位（cin）计算出和（sum）和进位（count）。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[3.1.2]：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a、b 两个输入信号相等,输出为1。否则为0*/
        endmodule
```

这个程序通过连续赋值语句描述了一个名为compare的比较器。对两比特数 a、b 进行比较，如a与b相等，则输出equal为高电平，否则为低电平。在这个程序中,/*.....*/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[3.1.3]：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名为trist2的三态驱动器。程序通过调用一个在Verilog语言库中现存的三态驱动器实例元件bufif1来实现其功能。

```
例[3.1.4]：  module trist1(out,in,enable);
               output  out;
               input  in, enable;
                  mytri  tri_inst(out,in,enable);
                   //调用由mytri模块定义的实例元件tri_inst
             endmodule

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

这个程序例子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调用由模块mytri定义的实例元件tri_inst。模块trist1是顶层模块。模块mytri则被称为子模块。

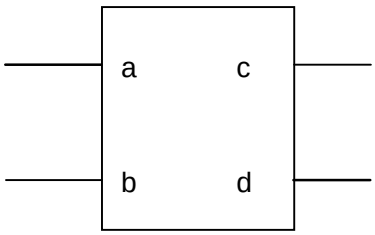
通过上面的例子可以看到:

- 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。
- 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。
- Verilog HDL程序的书写格式自由,一行可以写几个语句,一个语句也可以分写多行。
- 除了endmodule语句外,每个语句和数据定义的最后必须有分号。
- 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。

3.1.2.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，模块中的第二、第三行说明接口的信号流向，第四、第五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程序包括四个主要部分：端口定义、I/O说明、内部信号声明、功能定义。

3.1.3.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,.....);
```

3.1.4.模块内容

模块的内容包括I/O说明、内部信号声明、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i;   //(共有i个输入口)
输出口：      output 端口名1，端口名2，.....,端口名j;   //(共有j个输出口)
```

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...);

- **内部信号说明：**在模块内用到的和与端口有关的wire 和 reg 变量的声明。

如： reg [width-1 : 0] R变量1，R变量2 。。。。。；
wire [width-1 : 0] W变量1，W变量2 。。。。。；
.....

- **功能定义**：模块中最重要的部分是逻辑功能定义部分。有三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。例子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象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，表示在设计中用到一个跟与门（and）一样的名为and_inst的与门，其输入端为a, b，输出为q。要求每个实例元件的名字必须是唯一的，以避免与其他调用与门（and）的实例混淆。

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”语句是描述组合逻辑最常用的方法之一。而“always”块既可用于描述组合逻辑也可描述时序逻辑。上面的例子用“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，例如上例中就用了if...else语句来表达逻辑关系。如按一定的风格来编写“always”块，可以通过综合工具把源代码自动综合成用门级结构表示的组合或时序逻辑电路。

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。上面三个例子分别采用了“assign”语句、实例元件和“always”块。这三个例子描述的逻辑功能是同时执行的。也就是说，如果把这三项写到一个 VeriIog 模块文件中去，它们的次序不会影响逻辑实现的功能。这三项是同时执行的，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”模块也是同时执行的，但是模块内部的语句是顺序执行的。 看一下“always”内的语句，你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。

3.2.数据类型及其常量、变量

Verilog HDL中总共有十九种数据类型,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。在本教材中我们先只介绍四个最基本的数据类型,它们是：

reg型、wire型、integer型、parameter型

其它数据类型在后面的章节里逐步介绍，同学们也可以查阅附录中Verilog HDL语法参考书的有关章节逐步掌握。其它的类型如下：

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、wor型。这些数据类型除time型外都与基本逻辑单元建库有关，与系统设计没有很大的关系。在一般电路设计自动化的环境下，仿真用的基本部件库是由半导体厂家和EDA工具厂家共同提供的。系统设计工程师不必过多地关心门级和开关级的Verilog HDL语法现象。

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。下面就最常用的几种进行介绍。

3.2.1.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

- **整数:**

在Verilog HDL中,整型常量即整常数有以下四种进制表示形式:

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三种:

1) <位宽><进制><数字>这是一种全面的描述方式。
2) <进制><数字>在这种描述方式中,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) <数字>在这种描述方式中,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位二进制数的数字的位宽为4,一个4位十六进制数的数字的位宽为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, 'h表示十六进制。
```

- **x和z值:**

在数字电路中,x代表不定值,z代表高阻值。一个x可以用来定义十六进制数的四位二进制数的状态,八进制数的三位,二进制数的一位。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二位为不定值
4'b101z   //位宽为4的二进制数从低位数起第一位为高阻值
12'dz     //位宽为12的十进制数其值为高阻值(第一种表达方式)
12'd?     //位宽为12的十进制数其值为高阻值(第二种表达方式)
8'h4x     //位宽为8的十六进制数其低四位值为不定值
```

- **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（用八位二进制数表示）
8'd-5  //非法格式
```

- **下划线(underscore_):**

下划线可以用来分隔开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。见下列：

```
parameter  msb=7;          //定义参数msb为常量7
parameter  e=25, f=29;     //定义二个常数参数
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
```

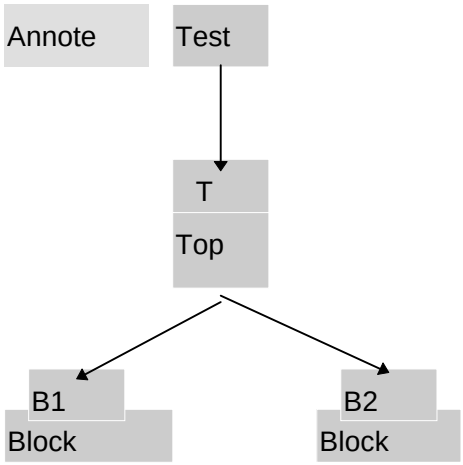
参数型常数经常用于定义延迟时间和变量宽度。在模块或实例引用时可通过参数传递改变在被引用模块或实例中已定义的参数。下面将通过两个例子进一步说明在层次调用的电路中改变参数常用的一些用法。

[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。可用例子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]：下面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 ( );
emdmodule

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
```



3.2.2 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，而且它必需受到驱动器(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(logic strength)，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附录:Verilog语言参考书)。

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型数据常用来表示用于以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i; //共有i条总线，每条总线内有n条线路
或
wire [n:1] 数据名1,数据名2,...数据名i;

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;          //定义了一个一位的wire型数据
wire [7:0] b;     //定义了一个八位的wire型数据
wire [4:1] c, d;  //定义了二个四位的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。在行为模块介绍这一节中我们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型数据常用来表示用于“always”模块内的指定信号，常代表触发器。通常，在设计中要由“always”块通过使用行为描述语句来表达逻辑关系。**在“always”块内被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i;
或
reg [n:1] 数据名1,数据名2,... 数据名i;

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（bit）。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：

```
reg  rega;          //定义了一个一位的名为rega的reg型数据
reg [3:0]  regb;    //定义了一个四位的名为regb的reg型数据
reg [4:1]  regc, regd; //定义了两个四位的名为regc和regd的reg型数据
```

对于reg型数据，其赋值语句的作用就象改变一组触发器的存储单元的值。在Verilog中有许多构造(construct)用来控制何时或是否执行这些赋值语句。这些控制构造可用来描述硬件触发器的各种具体情况，如触发条件用时钟的上升沿等，或用来描述具体判断逻辑的细节，如各种多路选择器。**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：当一个四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。在本书中我们还会对这一点作更详细的解释。

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型存储器，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位的存储器。该存储器的地址范围是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,          //定义二个参数。
           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它的寄存器的值。例如可以用一个加法计数器来做RAM的地址索引。本小节里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的示例中用到之处再逐一介绍。有兴趣的同学可以参阅附录：Verilog语言参考书

3.3. 运算符及表达式

Verilog HDL语言的运算符范围很广，其运算符按其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) 其它

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三种:

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):可以带二个操作数,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,这三个操作数用三目运算符分隔开。

见下例:

```
clock = ~clock;        // ~是一个单目取反运算符, clock是操作数。
c = a | b;             // 是一个二目按位或运算符, a 和 b是操作数。
r = s ? t : u;         // ?: 是一个三目条件运算符, s,t,u是操作数。
```

下面对常用的几种运算符进行介绍。

3.3.1.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。

3.3.2.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四种状态值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提供了以下五种位运算符：

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:

- 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.
- 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:

~	
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先进行异或运算再进行非运算.
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,系统会自动的将两者按右端对齐.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.

3.3.3 逻辑运算符

在Verilog HDL语言中存在三种逻辑运算符:

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),(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值。

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.

3.3.4.关系运算符

关系运算符共有以下四种:

```
a < b        a小于b
a > b        a大于b
a <= b       a小于或等于b
a >= b       a大于或等于b
```

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

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。

3.3.5.等式运算符

在Verilog HDL语言中存在四种等式运算符:

1) == (等于)
2) != (不等于)
3) === (等于)
4) !== (不等于)

这四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,两个操作数必需完全一致，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"。这四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。
例：

```
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)
```

3.3.6.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，下面将给出一例。

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

3.3.7.位拼接运算符(Concatation)

在Verilog HDL语言有一个特殊的运算符：**位拼接**运算符{}。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的位宽的大小时必需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。

3.3.8.缩减运算符(reduction operator)

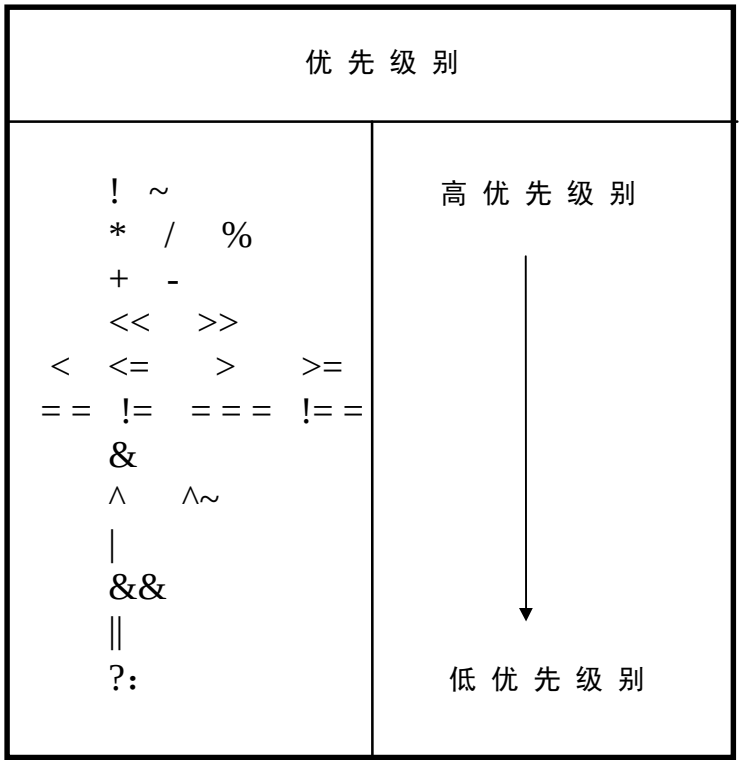
缩减运算符是单目运算符,也有与或非运算。其与或非运算规则类似于位运算符的与或非运算规则,但其运算过程不同。位运算是对操作数的相应位进行与或非运算,操作数是几位数则运算结果也是几位数。而缩减运算则不同,缩减运算是对单个操作数进行或与非递推运算,最后的运算结果是一位的二进制数。缩减运算的具体运算过程是这样的:第一步先将操作数的第一位与第二位进行或与非运算,第二步将运算结果与第三位进行或与非运算,依次类推,直至最后一位。

```
        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介绍。

3.3.9.优先级别

下面对各种运算符的优先级别关系作一总结。见下表:



3.3.10. 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,因此在编写原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(请参阅附录：Verilog语言参考手册)：

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.

3.4 赋值语句和块语句

3.4.1 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。（特别在编写可综合模块时）

(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完后立刻就改变的。**
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。问题主要是给"always"块内的reg型信号的赋值方式不易把握。到目前为止，前面所举的例子中的"always"模块内的reg型信号都是采用下面的这种赋值方式:

```
b <= a;
```

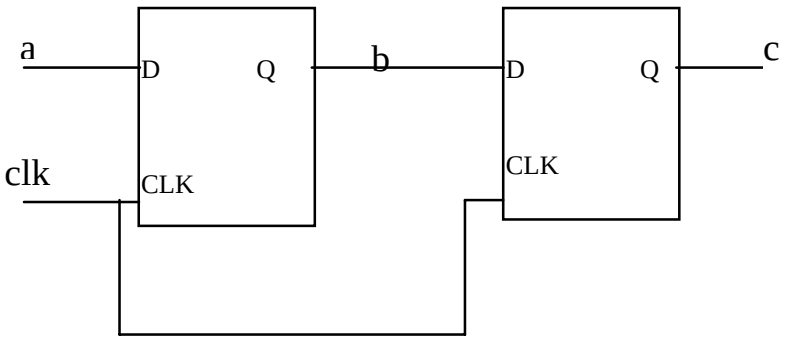
这种方式的赋值并不是马上执行的，也就是说"always"块内的下一条语句执行后，b并不等于a，而是保持原来的值。"always"块结束后，才进行赋值。而另一种赋值方式阻塞赋值方式，如下所示:

```
b = a;
```

这种赋值方式是马上执行的。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:

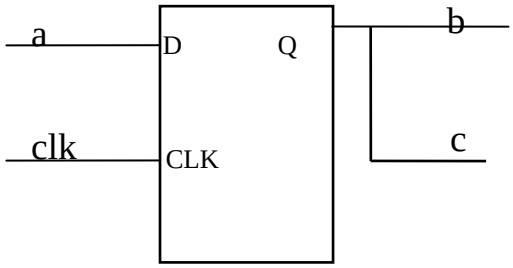
```
[例1]:always @( posedge clk )
          begin
             b<=a;
             c<=b;
          end
```

[例1]中的"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，这里应该用到了两个触发器。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下图所示:




```
[例2]: always @(posedge  clk)
          begin
             b=a;
             c=b;
          end
```

[例2]中的 "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)，生成的电路图如下所示只用了一个触发器来寄存器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就可以避免这种错误。



关于赋值语句更详细的说明请参阅第七章中深入理解阻塞和非阻塞赋值小节。

3.4.2 块语句

块语句通常用来将两条或多条语句组合在一起，使其在格式上看更象一条语句。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。下面进行详细的介绍。

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
    语句1;
    语句2;
    ......
    语句n;
end
```

或

```
begin:块名
    块内声明语句
    语句1;
    语句2;
    ......
    语句n;
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、real型变量声明语句。

下面举例说明：

```
[例1]：begin
        areg = breg;
        creg = areg;    //creg的值为breg的值。
    end
```

从该例可以看出，第一条赋值语句先执行，areg的值更新为breg的值，然后程序流程控制转到第二条赋值语句，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[例2]：

```
[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

并行块有以下四个特点：

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时的仿真时间的。
3) 延迟时间是用来给赋值语句提供执行时序的。
4) 当按时间时序排序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
    语句1;
    语句2;
    .......
    语句n;
join

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

其中：

- 块名即标识该块的一个名字，相当于一个标识符。
- 块内说明语句可以是参数说明语句、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、事件(event)说明语句。

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
```

在这个例子中用并行块来替代了前面例子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VerilgHDL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它语句调用，如被disable语句。
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，块名就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完的时间。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完的时间。

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，但同样可以生成前面例子中的波形。

3.5.条件语句

3.5.1. if_else语句

if语句是用来判定所给定的条件是否满足，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语言提供了三种形式的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;
```

六点说明：

(1).三种形式的if语句中在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).第二、第三种形式的if语句中，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
    out1 <=int2;
```

各 有 一 个 分 号

这是由于分号是Verilog HDL语句中不可缺少的部分，这个分号是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(如上例)，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，不需再附加分号。

(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企图,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，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例子。

下面的例子是取自某程序中的一部分。这部分程序用if_else语句来检测变量index以决定三个寄存器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。程序的前十行定义寄存器和参数。

```
//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];
```

3.5.2. 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:

a) case括弧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。
b) 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，就执行default后面的语句。
c) default项可有可无，一个case语句里只准有一个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
```

d) 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)。

e) 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。

f) 在用case语句表达式进行比较的过程中，只有当信号的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。

g) case语句的所有表达式的值的位宽必须相等，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)。

下面将给出 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不定值x和高阻值z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x,z值位的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语句的其它两种形式用来处理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
```

3.5.3.由于使用条件语句不当在设计中生成了原本没想到有的锁存器

Verilog HDL设计中容易犯的一个通病是由于不正确使用语言，生成了并不想要的锁存器。下面我们给出了一个在“always"块中不正确使用if语句，造成这种错误的例子。

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

有锁存器

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

无锁存器

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，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（本例中的sel)取不同的值时，给另一个信号（本例中的q)赋不同的值。注意看下图左边的例子，如果sel=0,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？在下面左边的这个例子中，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。

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

有 锁 存 器

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

无 锁 存 器

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。程序所示情况下，q赋为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，使设计者更加明确设计目标，同时也增强了Verilog程序的可读性。**

3.6.循环语句

在Verilog HDL中存在着四种类型的循环语句，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通过以下三个步骤来决定语句的循环执行。
   a) 先给控制循环次数的变量赋初值。
   b) 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，转到第三步。
   c) 执行一条赋值语句来修正控制循环变量次数的变量的值，然后返回第二步。

下面对各种循环语句详细的进行介绍。

3.6.1.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，用来作为仿真测试信号。它与always语句不同处在于不能独立写在程序中，而必须写在initial块中。其具体使用方法将在"事件控制"这一小节里详细地加以说明。

3.6.2.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
```

3.6.3.while语句

while语句的格式如下：

```
      while(表达式)  语句
```

或用如下格式：

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

下面举一个while语句的例子，该例子用while循环语句对rega这个八位二进制数中值为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
```

3.6.4.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
```

它的执行过程如下：

1) 先求解表达式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) 若表达式为真，在执行指定的语句后，求解表达式3。
4) 转回上面的第2步骤继续执行。
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这个八位二进制数中值为1的位进行计数的另一种方法。见下例：

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

3.7.结构说明语句

Verilog语言中的任何过程模块都从属于以下四种结构的说明语句。

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地加以介绍。这里只对initial和always语句加以介绍。

3.7.1.initial语句

initial语句的格式如下：

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

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
```

在这个例子中用initial语句在仿真开始时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，即用initial语句来生成激励波形作为电路的测试仿真信号。一个模块中可以有多个initial块，它们都是并行运行的。initial块常用于测试文件和虚拟模块的编写，用来产生仿真测试信号和设置信号记录等仿真环境。

3.7.2.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语句将会发成一个仿真死锁。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:period(=2*half_period) 的无限延续的信号波形，常用这种方法来描述时钟信号，作为激励信号来测试所设计的电路。

```
[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把tick信号反相，并且把counter增加1。这种时间控制是always语句最常用的。

always 的时间控制可以是沿触发也可以是电平触发的，可以单个信号也可以多个信号，中间需要用关键字 or 连接，如：

```
always @(posedge clock or posedge reset)  //由两个沿触发的always块
   begin
   ......
   end

always @( a or b or c )                   //由多个电平触发的always块
  begin
   ......
```

```
end
```

沿触发的always块常常描述时序逻辑，如果符合可综合风格要求可用综合工具自动转换为表示时序逻辑的寄存器组和门级逻辑，而电平触发的always块常常用来描述组合逻辑和带锁存器的组合逻辑，如果符合可综合风格要求可转换为表示组合逻辑的门级逻辑或带锁存器的组合逻辑。一个模块中可以有多个always块，它们都是并行运行的。

3.7.3.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，使程序明白易懂，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，主要的不同有以下四点：

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，而任务能启动其它任务和函数。
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，能计算多个结果值，这些结果值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，这个操作的结果值就是这个函数的返回值。下面让我们用例子来说明：

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);
```

下面分两节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。任务可以启动其它的任务，其它任务又可以启动别的任务，可以启动的任务数是没有限制的。不管有多少任务启动，只有当所有的启动任务完成以后，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;
  input a, b;
  inout  c;
  output d, e;
  ...
  <语句>    //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t
  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。

二．function说明语句

函数的目的是返回一个用于表达式的值。

- 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。下面用例子说明：

```
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
```

- 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值寄存器的名称初始化为与函数同名的内部变量。下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。

- 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。下面的例子中通过对两次调用函数getbyte的结果值进行位拼接运算来生成一个字。

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

- 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。

- 举例说明

下面的例子中定义了一个可进行阶乘运算的名为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
```

前面我们已经介绍了足够的语句类型可以编写一些完整的模块。在下一章里，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。

3.8.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, $recovery,
在Verilog HDL语言中每个系统函数和任务前面都用一个标识符$来加以确认。这些系统函数和任务提供了非常强大的功能。有兴趣的同学可以参阅附录：Verilog语言参考手册。下面对一些常用的系统函数和任务逐一加以介绍。

3.8.1.$display和$write任务

格式：

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

这两个函数和系统任务的作用是用来输出信息，即将参数p2到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至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：

- 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。表一中给出了常用的几种输出格式。

表一

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

- 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二中的转换序列来输出。下面表中的字符形式用于格式字符串参数中，用来显示特殊的字符。

表二：

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到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，其“输出表列”是需要输出的一些数据，可以是表达式。下面举几个例子说明一下。

```
[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
```

输出数据的显示宽度

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，在经过格式转换以后，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。对于其它进制，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).在输出格式为十进制的情况下：

- 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。
- 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。
- 如果表达式值的部分位为不定值，则输出结果为大写的X。
- 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).在输出格式为十六进制和八进制的情况下：

- 每4位二进制数为一组代表一位十六进制数，每3位二进制数为一组代表一位八进制数。
- 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的结果为小写的x。
- 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。
- 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。
- 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);                                          输出结果为：x
$display("%h", 14'bx0_1010);                                   输出结果为：xxXa
$display("%h %o",12'b001x_xx10_1x01,12'b001_xxx_101_x01);      输出结果为：XXX 1x5X
```

注意：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。

3.8.2.系统任务$monitor

格式：

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

任务$monitor提供了监控和输出参数列表中的表达式或变量值的功能。其参数列表中输出控制格式字符串和输出表列的规则和$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$monitor的启动和停止,这样使得程序员可以很容易的控制$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。通常在通过调用$monitoron启动$monitor时，不管$monitor参数列表中的值是否发生变化，总是立刻输出显示当前时刻参数列表中的值，这用于在监控的初始时刻设定初始比较值。在缺省情况下，控制标志在仿真的起始时刻就已经打开了。**在多模块调试的情况下，许多模块中都调用了$monitor,因为任何时刻只能有一个$monitor起作用，因此需配合$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便不间断地对所设定的信号进行监视。**

3.8.3.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。

- 系统函数$time

$time可以返回一个64比特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时设置寄存器set为0，在时刻为32ns时设置寄存器set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) $time显示时刻受时间尺度比例的影响。在上面的例子中，时间尺度是10ns，因为$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，所以在将经过尺度比例变换的数字输出时，要先进行取整。在上面的例子中，1.6和3.2经取整后为2和3输出。注意：时间的精确度并不影响数字的取整。

- $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]: `timescale10ns/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
```

从上面的例子可以看出，$realtime将仿真时刻经过尺度变换以后即输出，不需进行取整操作。所以$realtime返回的时刻是实型数。

3.8.4.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

3.8.5.系统任务$stop

格式：

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

$stop任务的作用是把EDA工具(例如仿真器)置成暂停模式，在仿真环境下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。

3.8.6.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贮器中。这两个系统任务可以在仿真的任何时刻被执行使用，其使用格式共有以下六种：

1) $readmemb("<数据文件名>",<存贮器名>);
2) $readmemb("<数据文件名>",<存贮器名>,<起始地址>);
3) $readmemb("<数据文件名>",<存贮器名>,<起始地址>,<结束地址>);
4) $readmemh("<数据文件名>",<存贮器名>);
5) $readmemh("<数据文件名>",<存贮器名>,<起始地址>);
6) $readmemh("<数据文件名>",<存贮器名>,<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，地址一词指对存贮器(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贮器单元中去。存贮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中，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贮器中相应的地址单元中去。

对于上面六种系统任务格式,需补充说明以下五点：

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贮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贮器中，直到该存贮器单元存满为止或数据文件里的数据存完。
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，存放到该存贮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按该起始地址开始存放到存贮器单元中，直到该结束地址，而不考虑该存贮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，并且装载数据到存贮器中的操作被中断。
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的话，也要提示错误信息。

下面举例说明：

先定义一个有256个地址的字节存贮器 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贮器mem中。

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

第一条语句在仿真时刻为0时，将装载数据到以地址是1的存贮器单元为起始存放单元的存贮器中去。第二条语句将装载数据到以单元地址是16的存贮器单元为起始存放单元的存贮器中去，一直到地址是256的单元为止。第三条语句将从地址是128的单元开始装载数据，一直到地址为1的单元。在第三种情况中，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。

3.8.7.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32bit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到59之间的随机数，下面的例子通过位并接操作产生一个值在0到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。同学们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。

[例]
```
`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 * ( {$random} % 6);
       // delay1 在0到100ns间变化
       delay2 = 20 * ( 1 + {$random} % 3);
       // delay2 在20到60ns间变化
```

```
                #delay1  dout = 1 << ({$random} %10);
                //dout的0--9位中随机出现1，并出现的时间在0-100ns间变化
                #delay2  dout = 0;
                //脉冲的宽度在在20到60ns间变化
                end
            end
        endmodule
```

3.9.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在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它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

在这一小节里只对常用的`define、`include、`timescale进行介绍，其余的请查阅参考书。

3.9.1.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，也可以用一个有含义的名字来代替没有含义的数字和符号，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;
```

关于宏定义的八点说明：

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原文件结束。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，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的。

- 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");
```

注意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，如使用和预处理命令名相同的宏名会发生冲突，因此建议不要使用和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。但是这样容易引起混淆，建议不要这样使用。

3.9.2.“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命令用来实现“文件包含”的操作。其一般形式为:

```
`include “文件名”
```

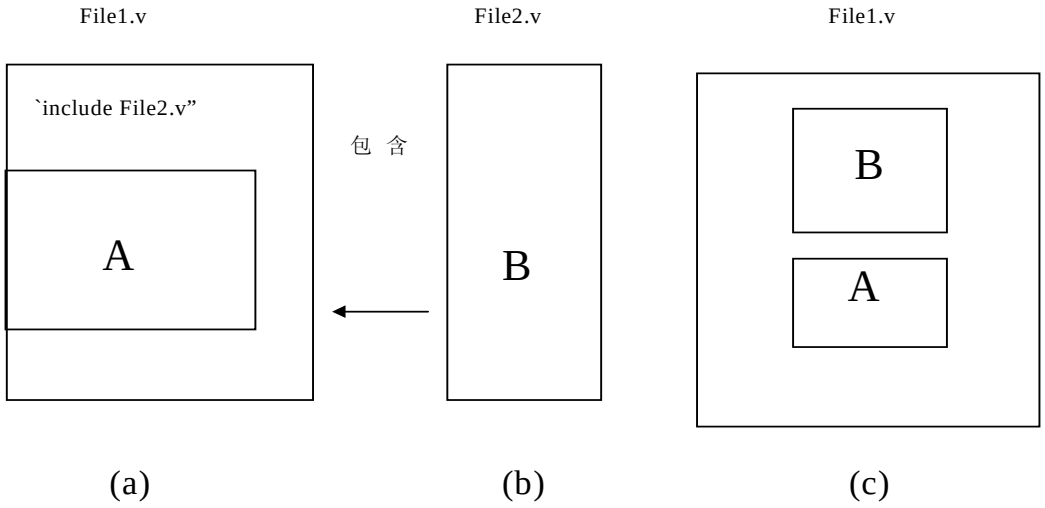


图 3－9－2

图3-9-2表示“文件包含”的含意。图3-9-2(a)为文件File1.v,它有一个`include "File2.v"命令，然后还有其它的内容(以A表示)。图3-9-2(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3-9-2(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。

“文件包含”命令是很有用的，它可以节省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

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]：
(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  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的子模块来被调用的。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  aaa(.a(c),.b(d),.out(out_a));
      assign out= e & out_a;
endmodule
```

关于“文件包含”处理的四点说明：

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 size1  4
.
.
.
```

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，见图3-9-3.

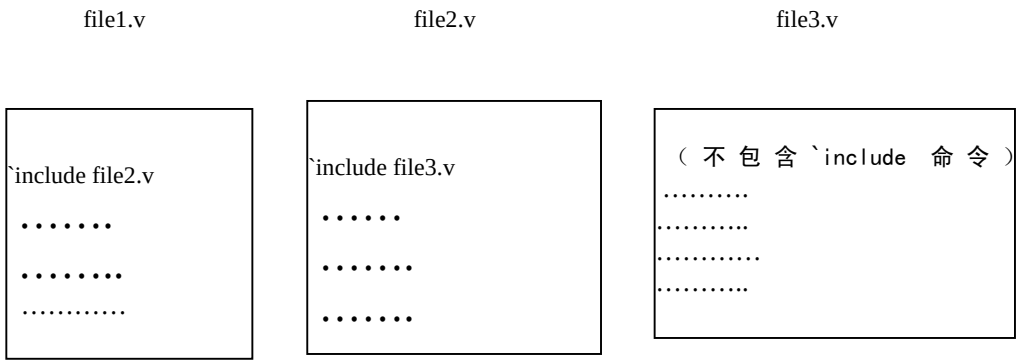


图 3-9-3

它的作用和图3-9-4的作用是相同的。

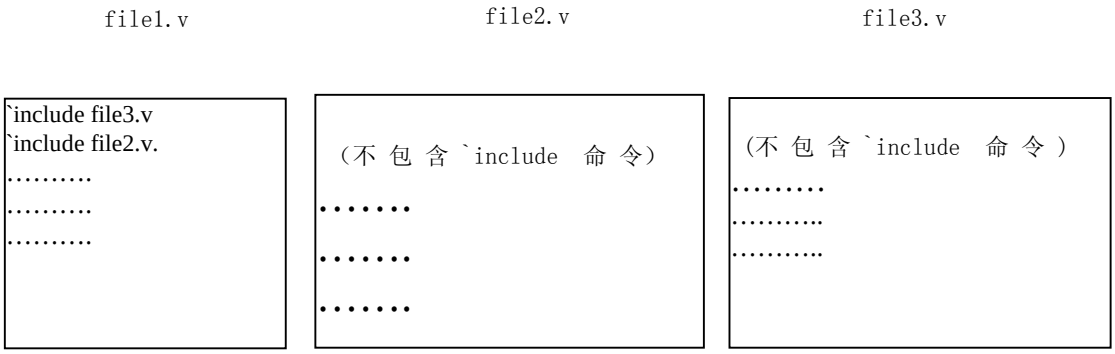


图3-9-4

3.9.3.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了不同的时间单位的模块。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的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，单位为秒(s)、毫秒(ms)、微秒(us)、纳秒(ns)、皮秒(ps)、毫皮秒(fs)。这几种单位的意义说明见下表。

时间单位	定义
s	秒(1S)
ms	千分之一秒(10^{-3}S)
us	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是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三位小数的实型数，因为`timescale命令定义时间精度为1ps.

[例2]： `timescale 10us/100ns
在这个例子中，`timescale命令定义后，模块中时间值均为10us的整数倍。因为`timesacle 命令定义的时间单位是10us。延迟时间的最小分辨度为十分之一微秒(100ns)，即延迟时间可表达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0，在仿真时刻为32ns时寄存器set被赋值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值被从1.55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作为时间单位的整数倍为16ns。
3) EDA工具预定在仿真时刻为16ns的时候给寄存器set赋值0(即语句 #d set=0;执行时刻)，在仿真时刻为32ns的时候给寄存器set赋值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。

3.9.4.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，希望当满足条件时对一组语句进行编译，而当条件不满足是则编译另一部分。

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：

- 选择一个模块的不同代表部分。
- 选择不同的时序或结构信息。
- 对不同的EDA工具，选择不同的激励。

3.10.小结

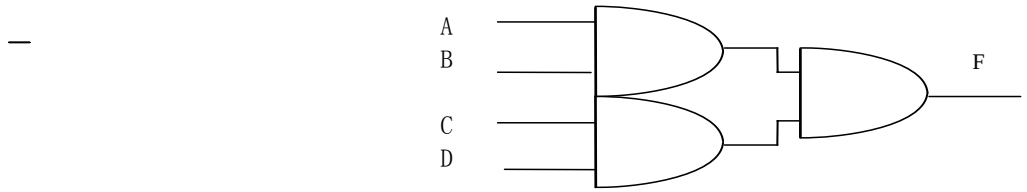
Verilog HDL的语法与C语言的语法有许多类似的地方，但也有许多不同的地方。我们学习Verilog HDL语法要善于找到不同点，着重理解如：阻塞〔Blocking〕和非阻塞〔Non-Blocking〕赋值的不同；顺序块和并行块的不同；块与块之间的并行执行的概念；task和function的概念。Verilog HDL还有许多系统函数和任务也是C语言中没有的如:$monitor、$readmemb、$stop等等，而这些系统任务在调试模块的设计中是非常有用的，我们只有通过阅读大量的Verilog调试模块实例，经过长期的实践，经常查阅附录中的Verilog语言参考手册才能逐步掌握，。

3.11.思考题

在这一小结中我们将针对以上介绍的基本语法做一些练习。希望读者能在仔细阅读以上的内容后，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的要点。

1）以下给出了一个填空练习，请将所给各个选项根据电路图，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 )
```

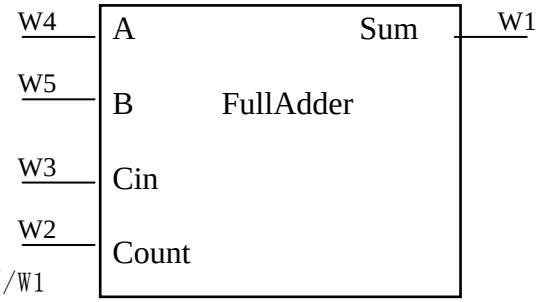


F =((A [ ] B) [ ] (C [ ] D)) [ ]

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〕 在这一题中，我们将作有关层次电路的练习，通过这个练习，你将加深对模块间调用时，管脚间连接的理解。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。




```
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：

```
moduleTop...
FullAdderFA(    .Sum(W1), //W1
                .Cout(W2),//W2
                .Cin(W3), //W3
                .A(W4),   //W4
                .B(W5));  //W5
endmodule
```

3)下面这道题是一个测试模块，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;
MUX2M(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4bit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人空的方括号中。

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。

```
7.1) 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)

```
7.2) 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。

```
8.1) assign A=B;
8.2) 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;
...
FADDM(C1,C2,C3,C4,C5);
...
endmodule
```

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

10）在下一个程序段中，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'b11?00:
            begin
               A=1;
               B=1;
            end
endcase
```

标准答案: A=1 and B=0;
Case语句只要执行完其中的一句话就会跳出endcase

11)在下题中，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)在下题中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型为非负数，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)在下题中，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)在下题中循环执行完后，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的值是它进人循环体前值的取反。

(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)在下题中，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)在下题中，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，可从以下五种答案中选择。

```
        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)在下题中，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)在下题中,模块被综合后将产生几个触发器?

```
    always @(posedge Clk)
        begin: Blk
```

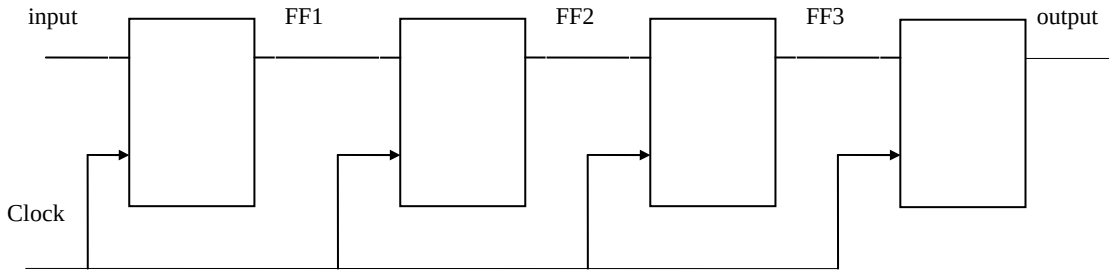
```
            reg B, C;
               C = B;
               D <= C;
               B = A;
         end
```

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)在下题中，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
010：1
011：3
100：0
101：3
110：0
111：3
```

```
casex(SEL)
3'b(  ):  OP=3;
3'b(  ):  OP=1;
3'b(  ):  OP=0;
endcase
```

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)在下一个模块旁的括号中填入display的正确值。

```
integerI;
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)根据下题给出的程序，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，做为这个参数的前缀。)

```
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.Count or Test.T.B1.C.Count

28) 下题中用initial块给reg[7:0]V符值，请指明每种情况下V的8位都是什值。
这道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
```

第四章 不同抽象级别的Verilog HDL模型

前言

从第三章我们知道，Verilog模型可以是实际电路不同级别的抽象。这些抽象的级别和它们对应的模型类型共有以下五种：

1) 系统级(system)
2) 算法级(algorithmic)
3) RTL级(RegisterTransferLevel):
4) 门级(gate-level):
5) 开关级(switch-level)

在本章的各节中我们将通过许多实际的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。而对于电路基本部件（如门、缓冲器、驱动器等）库的设计者而言，则需要掌握用户自定义源语元件（UDP）和开关级的描述。在本教材中由于篇幅有限，我们只简单介绍了UDP，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。

4.1.门级结构描述

一个逻辑网络是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。

4.1.1.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个之多，在本教材中我们只介绍最基本的八个。有关其它的门类型关键字，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。下面列出了八个基本的门类型（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加以介绍.最后我们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。

4.1.2.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，我们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4.1.2中的各个基本部件，而其后面括号中的参数分别为图4.1.2 中各基本部件的输入输出信号。

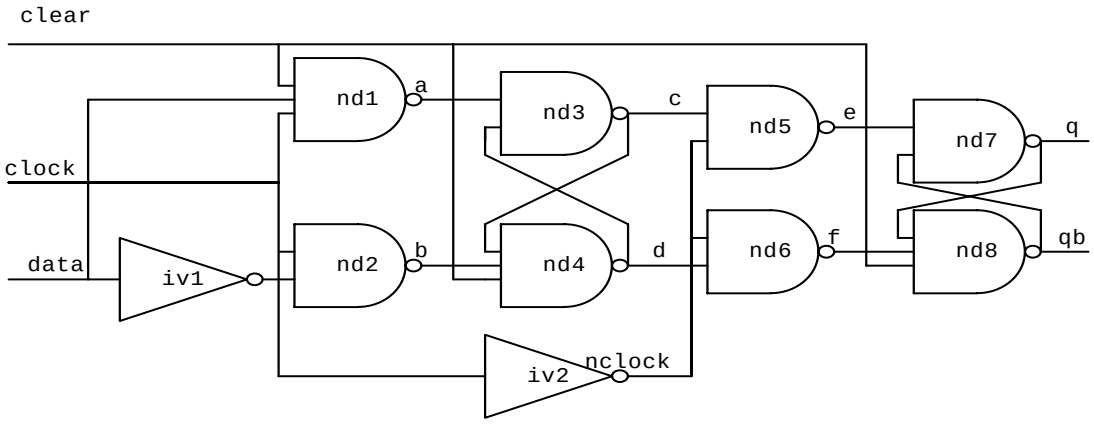


图4.1.2.　D型主从触发器的电路结构图

4.1.3.由已经设计成的模块来构成更高一层的模块

如果已经编制了一个模块，如4.1.2.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：

```
1)  flop  flop_d( d1, clk, clrb, q, qn);
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。从上面的两个例子可以看出引用时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需一一列出(见例2)。

下面的例子中引用了4.1.2 中已设计的模块flop，用它构成一个四位寄存器。

```
  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5中的各个基本部件，而其后面括号中的参数分别为图5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。

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。

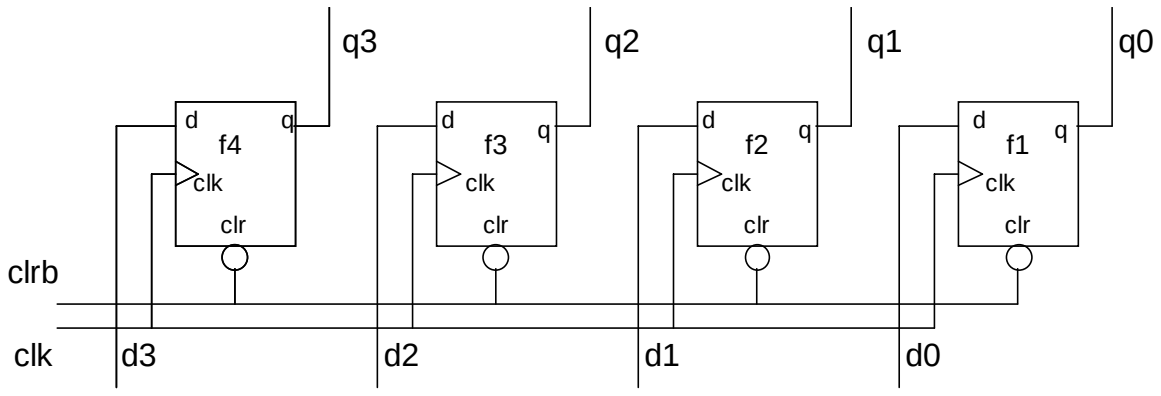


图4.1.3　　四位寄存器电路结构图

4.1.4 用户定义的原语（UDP）

用户定义的原语是从英语User Defined Primitives直接翻译过来的,在Verilog HDL 中我们常用它的缩写UDP来表示。利用UDP用户可以定义自己设计的基本逻辑元件的功能，也就是说，可以利用UDP来定义有自己特色的用于仿真的基本逻辑元件模块并建立相应的原语库。这样，我们就可以与调用Verilog HDL基本逻辑元件同样的方法来调用原语库中相应的元件模块来进行仿真。由于UDP是用查表的方法来确定其输出的，用仿真器进行仿真时，对它的处理速度较对一般用户编写的模块快得多。与一般的用户模块比较，UDP更为基本，它只能描述简单的能用真值表表示的组合或时序逻辑。UDP模块的结构与一般模块类似，只是不用module而改用primitive关键词开始，不用endmodule而改用endprimitive关键词结束。在Verilog的语法中还规定了UDP的形式定义和必须遵守的几个要点，与一般模块有不同之处，我们在下面加以介绍。

定义UDP的语法：

```
primitive 元件名（输出端口名，输入端口名1，输入端口名2，...）
 output 输出端口名;
 input  输入端口名1, 输入端口名2，...;
 reg 输出端口名;
 initial  begin
            输出端口寄存器或时序逻辑内部寄存器赋初值（0，1，或 X）；
          end
  table
       //输入1    输入2    输入3   ...    :  输出
         逻辑值   逻辑值   逻辑值  ...    : 逻辑值 ;
         逻辑值   逻辑值   逻辑值  ...    : 逻辑值 ;
         逻辑值   逻辑值   逻辑值  ...    : 逻辑值 ;
          ...       ...       ...   ...    :  ...    ;
    endtable
endprimitive
```

注意点：

1) UDP只能有一个输出端，而且必定是端口说明列表的第一项。
2) UDP可以有多个输入端，最多允许有10个输入端。
3) UDP所有端口变量必须是标量，也就是必须是1位的。
4) 在UDP的真值表项中，只允许出现0、1、X三种逻辑值，高阻值状态Z是不允许出现的。
5) 只有输出端才可以被定义为寄存器类型变量。
6) initial语句用于为时序电路内部寄存器赋初值，只允许赋0、1、X三种逻辑值，缺省值为X。

对于数字系统的设计人员来说，只要了解UDP的作用就可以了，而对微电子行业的基本逻辑元器件设计工程师，必须深入了解UDP的描述，才能把所设计的基本逻辑元件，通过EDA工具呈现给系统设计工程师。有兴趣的同学可以参阅本书的附录：“Verilog语言参考手册”中有关UDP的语法和使用说明。

4.2.Verilog HDL的行为描述建模

4.2.1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作为输出，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。这就要求我们编写测试模块，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2个，分别是时钟clk、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。

我们再举一个简单的例子，即编制4.1.3.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，还需引用四位寄存器

(hardreg)模块，以观测各种组合信号输入到该四位寄存器(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 [3:0] 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 clock = ~clock;

       always @(end_first_pass)
            clearb = ~clearb;

       always @(posedge clock)
         $display("at time %0d clearb= %b data= %d qout= %d", $time, clearb, data, qout);
/*********************************************************
   类似于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;       //结束仿真
        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)
```

上面我们简单地介绍了利用Verilog HDL门级结构建模，来设计复杂数字电路的最基本的思路。而实用的电路设计往往并没有那么简单。我们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。

4.2.2.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门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级或算法级或系统级的模块)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最后形成*注[1]*，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。本教材中有关可综合风格的Verilog HDL的内容，我们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。

所谓逻辑综合就其实质而言是设计流程中的一个阶段，在这一阶段中将较高级抽象层次的描述自动地转换成较低层次描述。就现在达到的水平而言，所谓逻辑综合就是通过综合器把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。而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*注[1]*。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。

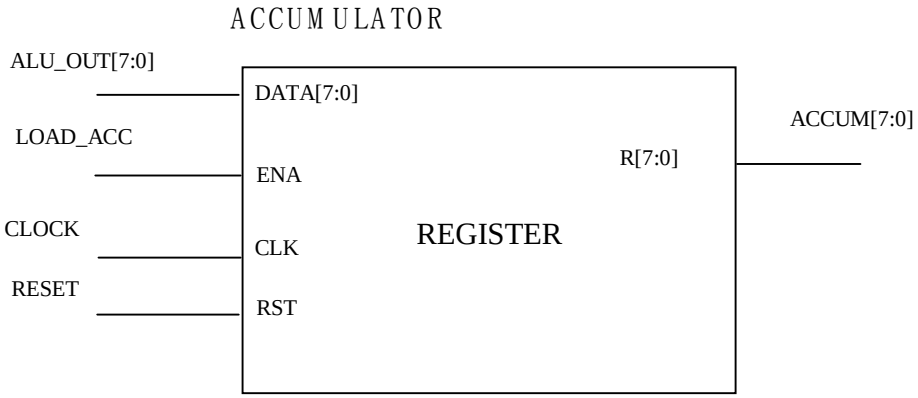
注[1]：请参阅参考资料[1]

4.3.用Verilog HDL建模进行TOP-DOWN设计的实例

下面是一个用Verilog HDL的建模来设计一个用于教学的经简化的只有八条指令、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是一个复杂的数字逻辑电路，但是它基本部件的逻辑并不复杂，我们可把它分割成九个基本部件：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：

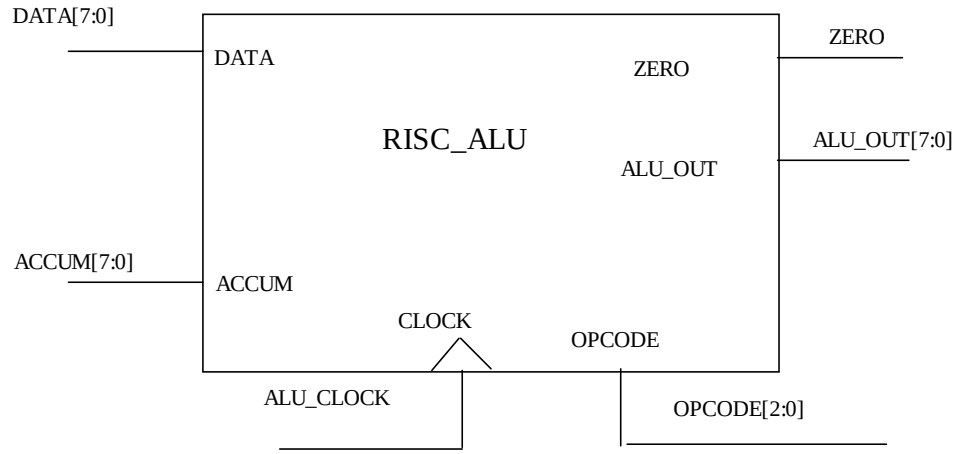
（1）累加器用寄存器（ACCUMULATOR RREGISTER）




```
`timescale1ns/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
```

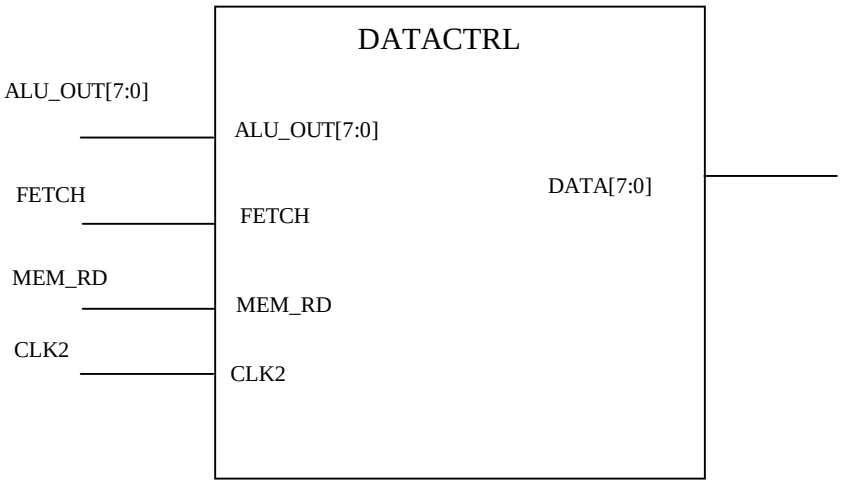
其中 DFF和and都是Verilog语言中保留的关键字分别表示带复位端的D触发器和与门。

（2）RISC算术运算单元（RISC_ALU）



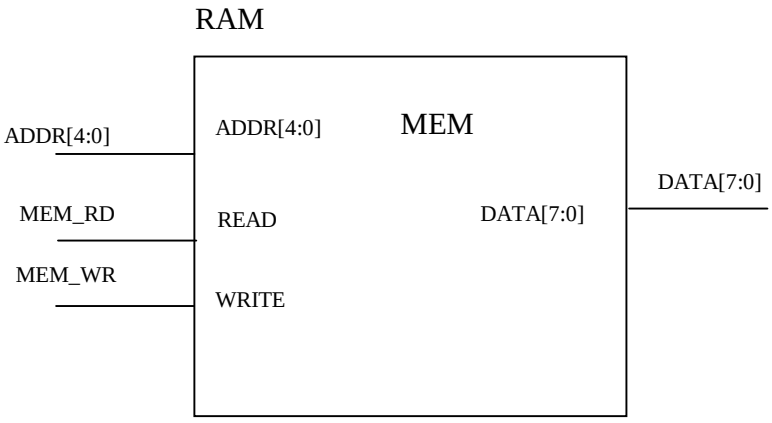

```
`timescale1ns/100ps
module riscalu ( alu_out, zero, opcode, data, accum, clock );
output [7:0] alu_out;
reg[7:0] alu_out;
output zero;
input [2:0] opcode;
input [7:0] data, accum;
input clock;

`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");
        #`ALUdly 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
```

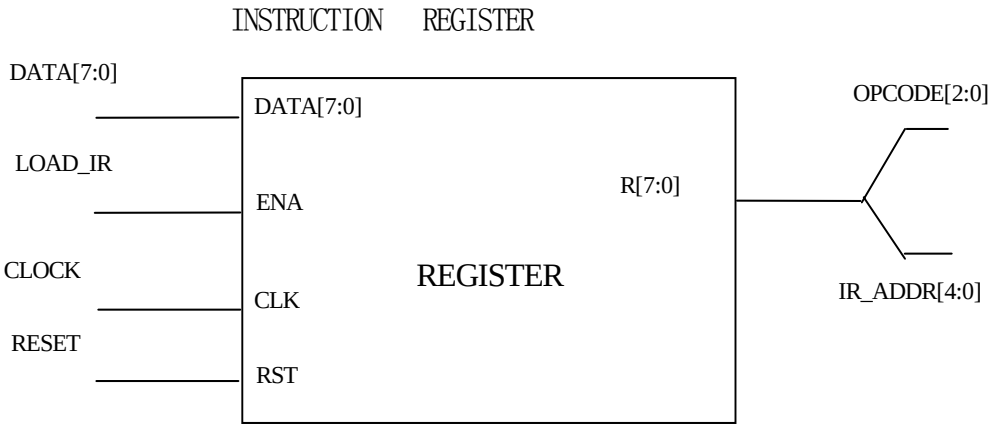
（4）动态存储器（RAM）




```
`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
```

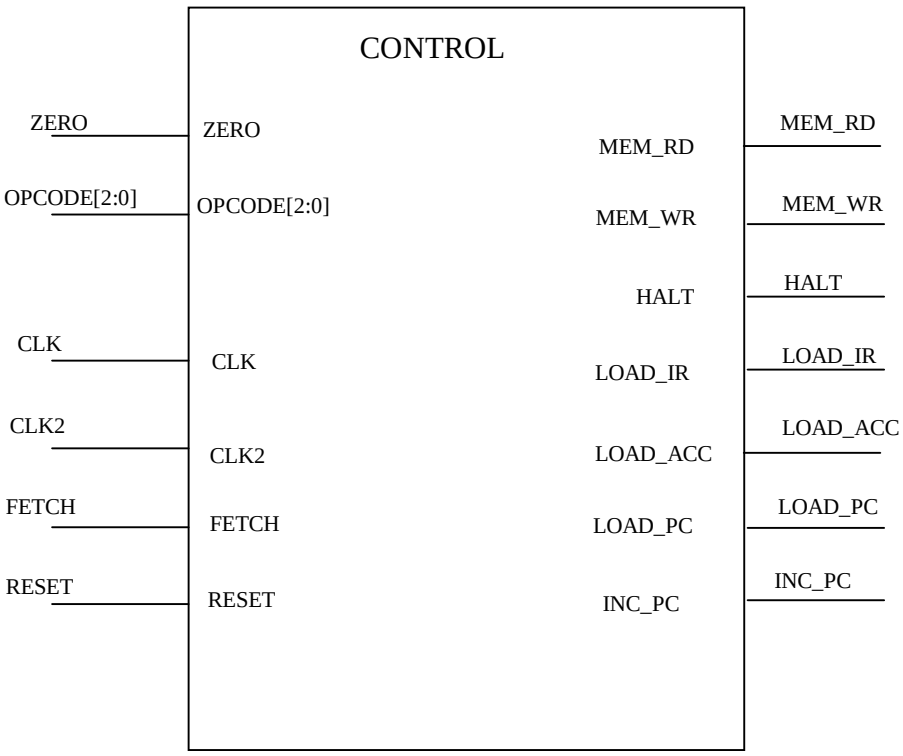
endmodule

（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（STATECONTROLLER）

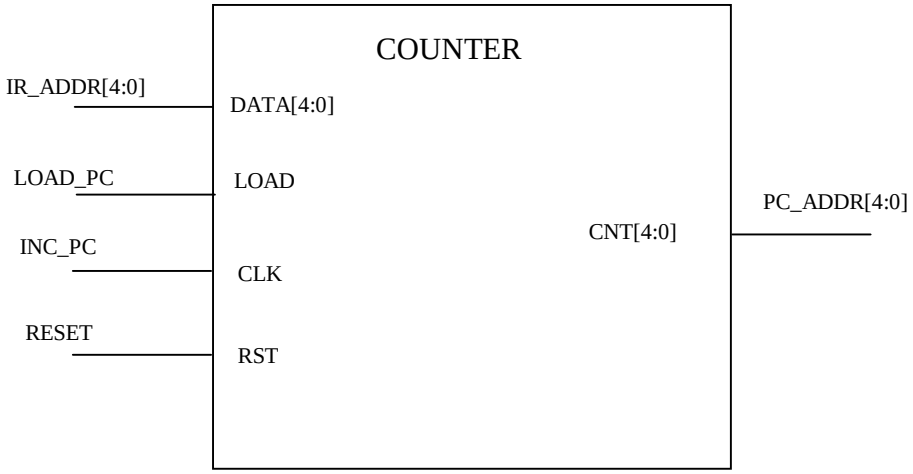



```
`timescale 1ns/1ns
module control (load_acc, mem_rd, mem_wr, inc_pc, load_pc, load_ir,
                halt, opcode, fetch, zero, clk, clk2, reset);
output load_acc, mem_rd, mem_wr, inc_pc, load_pc, load_ir, halt;
reg load_acc, mem_rd, mem_wr, inc_pc, load_pc, load_ir, halt;
input [2:0] opcode;
input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

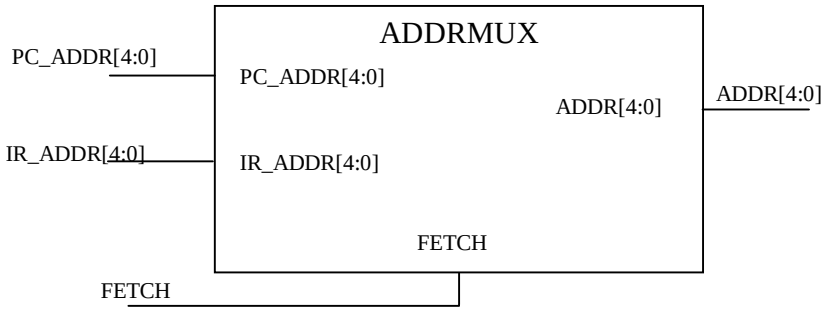
```
endmodule
```

（7）程序计数器（PROGRAMMCOUNTER）



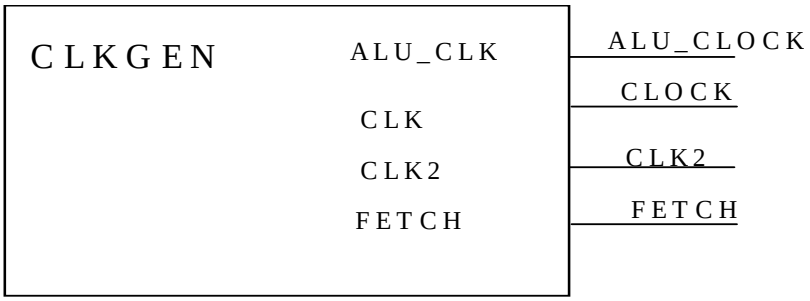

```
/*********************************************************
*   Behavior of a 5-bit counter
*********************************************************/
`timescale 1ns/1ns
module counter (cnt, clk, data, rst, load);
     output [4:0] cnt;
     input [4:0] data;
     input clk, rst, load;
     reg [4:0]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

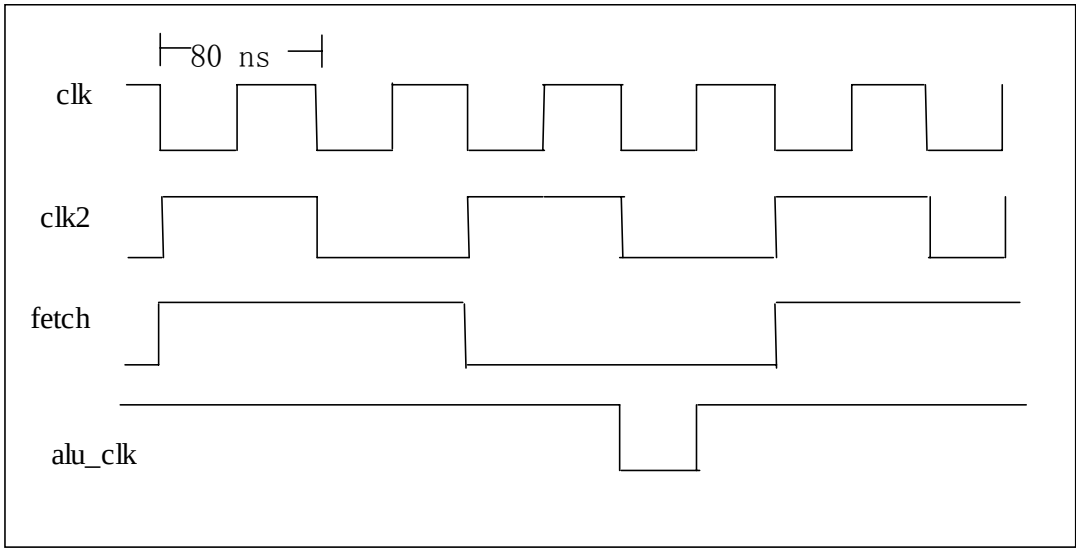



```
module addrmux(addr, pc_addr, ir_addr, fetch);
     output [4:0] addr;
     input [4:0] pc_addr, ir_addr;
     input fetch;
       assign  addr = ( fetch?  pc_addr : ir_addr );
     endmodule
```

（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:
*******************************************************/
```




```
`timescale1ns/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，但并非其所有的子模块都是可综合的，若需要综合成逻辑网表（EDIF），部分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。

我们将在第八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8K字节。

4.4.小结

在本章中我们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。

4.5.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. 为什么说的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

第五章 基本运算逻辑和它们的 Verilog HDL 模型

前言

复杂的算法数字逻辑电路是由基本运算逻辑、数据流动控制逻辑和接口逻辑电路所构成的。对基本运算逻辑的深入了解是设计复杂算法逻辑系统电路结构的基本功。虽然 Verilog 硬件描述语言能帮助我们自动地综合出极其复杂的组合和时序电路，并帮助我们对所设计的电路进行全面细致的验证，但对于速度要求很高的特殊数字信号处理电路，其结构还是由设计者来定夺。为了提高算法的运算速度除了提高制造工艺技术外，逻辑结构设计是最重要的环节。而设计出结构合理的基本运算组合电路是算法逻辑结构设计的基础，只有深入理解复杂组合电路的许多基本特点，才有可能通过电路结构的改进来提高算法逻辑系统的基本时钟速度，为结构合理的高速复杂算法的数字逻辑系统的构成打下坚实的基础。这部分知识应该是数字系统和计算机结构课程讲述的内容，为了使同学们能熟练地把学过的基础知识运用到设计中去，有必要在这里把提高加法器、乘法器速度的电路结构原理和方法简单地复习一下，并把流水线设计的概念也在这一章中引入。希望同学们能灵活地把这些电路结构的基本概念应用到设计中，来提高设计的水平。

5.1 加法器

在数字电路课程里我们已学习过一位的加法电路，即全加器。它的真值表很容易写出，电路结构也很简单仅由几个与门和非门组成。

X_i	Y_i	C_{i-1}	S_i	C_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

表 5.1　　一位全加器的真值表

表中 X_i 、Y_i 表示两个加数，S_i 表示和，C_{i-1} 表示来自低位的进位、C_i 表示向高位的进位。从真值表很容易写出逻辑表达式如下：

$$C_i = X_i Y_i + Y_i C_{i-1} + X_i C_{i-1}$$

$$S_i = X_i \overline{C_i} + Y_i \overline{C_i} + C_{i-1} \overline{C_i} + X_i Y_i C_{i-1}$$

全加器和 S_i 的表达式也可以表示为：

$$S_i = P_i \oplus C_i \qquad \text{其中 } P_i = X_i \oplus Y_i \qquad (5.1)$$

$$C_i = P_i \cdot C_{i-1} + G_i \qquad \text{其中 } G_i = X_i \cdot Y_i \qquad (5.2)$$

5.2 式就是进位递推公式。参考清华大学出版社出版的刘宝琴老师编写的《数字电路与系统》，可以很容易地写出超前进位形成电路的逻辑，在这里不再详细介绍。

在数字信号处理的快速运算电路中常常用到多位数字量的加法运算，这时需要用到并行加法器。并行加法器比串行加法器快得多，电路结构也不太复杂。它的原理很容易理解。现在普遍采用的是 Carry-Look-Ahead-Adder 加法电路（也称超前进位加法器），只是在几个全加器的基础上增加了一个超前进位形成逻辑，以减少由于逐位进位信号的传递所造成的延迟。下面的逻辑图表示了一个四位二进制超前进位加法电路。

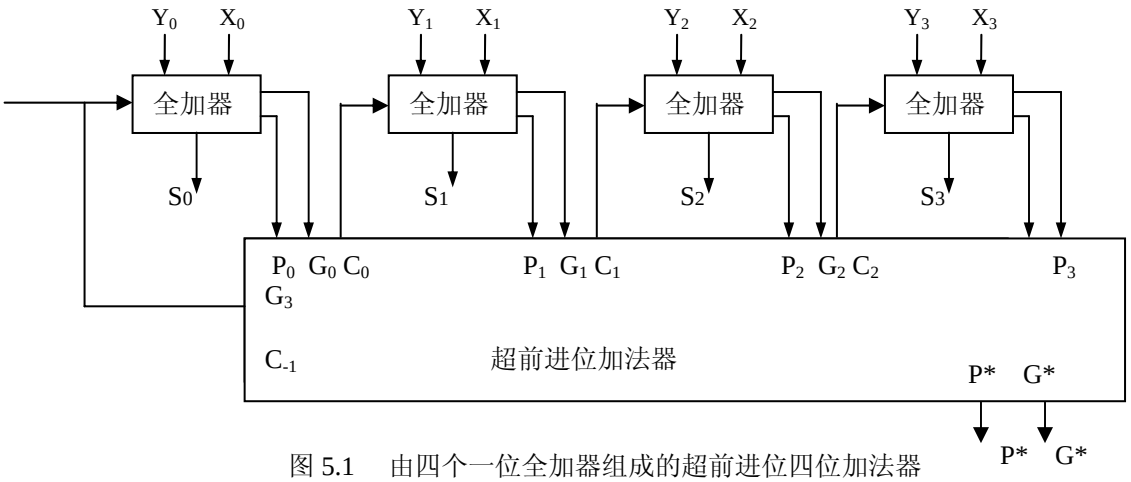


图 5.1　由四个一位全加器组成的超前进位四位加法器

同样道理，十六位的二进制超前进位加法电路可用四个四位二进制超前进位加法电路再加上超前进位形成逻辑来构成。同理，依次类推可以设计出 32 位和 64 位的加法电路。

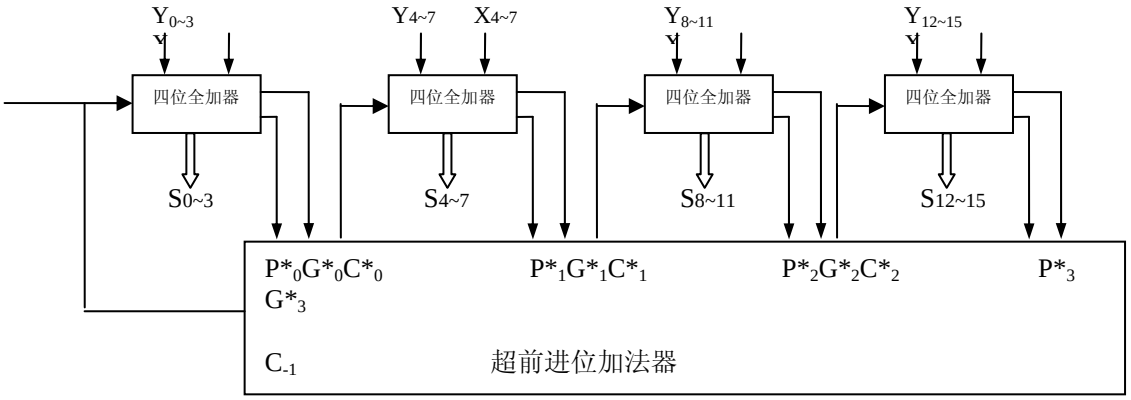


图 5.2 由四个四位全加器组成的超前进位十六位加法器

在实现算法时（如卷积运算和快速富里叶变换），常常用到加法运算，由于多位并行加法器是由多层组合逻辑构成，加上超前进位形成逻辑虽然减少了延迟，但还是有多级门和布线的延迟，而且随着位数的增加延迟还会积累。由于加法器的延迟，使加法器的使用频率受到限制，这是指计算的节拍（即时钟）必须要大于运算电路的延迟，只有在输出稳定后才能输入新的数进行下一次运算。如果设计的是 32 位或 64 位的加法器，延迟就会更大。为了加快计算的节拍，可以在运算电路的组合逻辑层中加入多个寄存器组来暂存中间结果。也就是采用数字逻辑设计中常用的流水线（ pipe line ） 办法，来提高运算速度，以便更有效地利用该运算电路，我们在本章的后面还要较详细地介绍流水线 结构的概念和设计方法。我们也可以根据情况增加运算器的个数，以提高计算的并行度。

用 Verilog HDL 来描述加法器是相当容易的，只需要把运算表达式写出就可以了，见下例。

```
module add_4( X, Y, sum, C);
input [3 : 0] X, Y;
output [3: 0]  sum;
output C;

assign   {C,  Sum } = X + Y;
```

```
endmodule
```

而 16 位加法器只需要扩大位数即可，见下例：

```
module add_16( X, Y, sum, C);
input [15 : 0] X, Y;
output [15 : 0]  sum;
output C;

assign   {C,  Sum } = X + Y;

endmodule
```

这样设计的加法器在行为仿真时是没有延时的。借助综合器，可以根据以上 Verilog HDL 源代码自动将其综合成典型的加法器电路结构。综合器有许多选项可供设计者选择，以便用来控制自动生成电路的性能。设计者可以考虑提高电路的速度，也可以考虑节省电路元件以减少电路占用硅片的面积。综合器会自动根据选项为你挑选一种基本加法器的结构。有的高性能综合器还可以根据用户对运算速度的要求插入流水线结构，来提高运算器的性能。可见在综合工具的资源库中存有许多种基本的电路结构，通过编译系统的分析，自动为设计者选择一种电路结构，随着综合器的日益成熟它的功能将越来越强。然后设计者还需通过布局布线工具生成具有布线延迟的电路，再进行后仿真，便可知道该加法器的实际延时。根据实际的延迟便可以确定使用该运算逻辑的最高频率。若需要重复使用该运算器，则需要在控制数据流动的状态机中为其安排必要的时序。

5.2 乘法器

乘法电路：

在数字信号处理中经常需要进行乘法运算，乘法器的设计对运算的速度有很大的影响。本节讨论两个二进制正数的乘法电路和运算时间延迟问题以及怎样用 VerilogHDL 模型来表示乘法运算。还将讨论当用综合工具生成乘法运算电路时，怎样来控制运算的时间延迟。

设两个 n 位二进制正数 X 和 Y：

$$X : X_{n-1} \cdot\cdot\cdot X_1 X_0$$

$$Y : Y_{n-1} \cdot\cdot\cdot Y_1 Y_0$$

则 X 和 Y 的乘积 Z 有 2n 位：

并且

式中$Y_i X$称为部分积，记为P_i，有

显然，两个一位二进制数相乘遵循如下规则:

$0\times 0=0;\quad 0\times 1=0;\quad 1\times 0=0;\quad 1\times 1=1$

因此$Y_i X_j$ 可用一个**与**门实现，记$P_{i,j} = Y_i X_j$

例：两个四位二进制数 X 和 Y 相乘。

被乘数: X_3 X_2 X_1 X_0

×) 乘 数: Y_3 Y_2 Y_1 Y_0

				Y_0X_3	Y_0X_2	Y_0X_1	Y_0X_0	
			Y_1X_3	Y_1X_2	Y_1X_1	Y_1X_0		
		Y_2X_3	Y_2X_2	Y_2X_1	Y_2X_0			
	Y_3X_3	Y_3X_2	Y_3X_1	Y_3X_0				
乘积:	Z_7	Z_6	Z_5	Z_4	Z_3	Z_2	Z_1	Z_0

快速乘法器常采用网格形式的迭带阵列结构，图 5.3 示出两个四位二进制数相乘的结构图，图中每一个乘法单元 MU 的逻辑图如图 5.4 所示，即每一个 MU 由一个**与**门和一个全加器构成。事实上，图 5.3 中第一行的每个 MU 可用一个**与**门实现，每一行最右边一个 MU 中的全加器可用半加器实现。图 5.3 实现乘法的最长延时为 1 个**与**门的传输延时加上八个全加器的传输延时。假设每个全加器产生和与产生进位的传输延时相同，并且均相当 4 个**与**门的传输延时，则图 5.3 逐位进位并行乘法器的最长延时为 1+8×4=33 个门的传输延时。

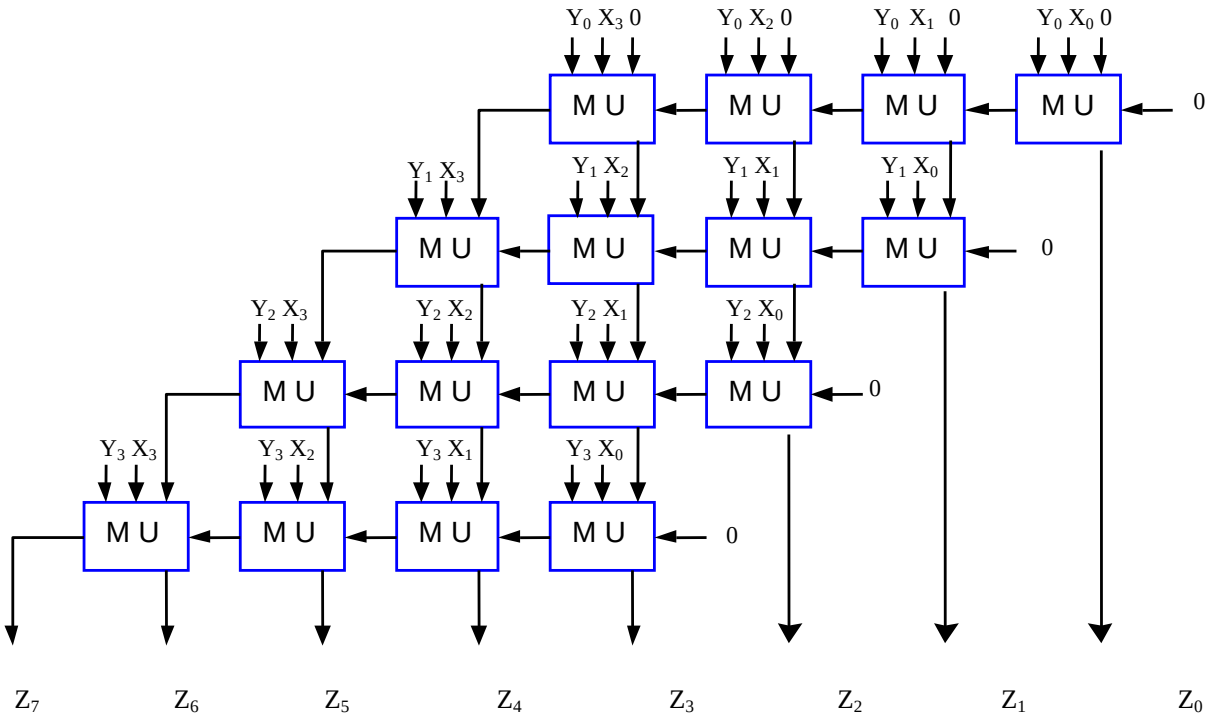


图 5.3　　逐位进位并行乘法器

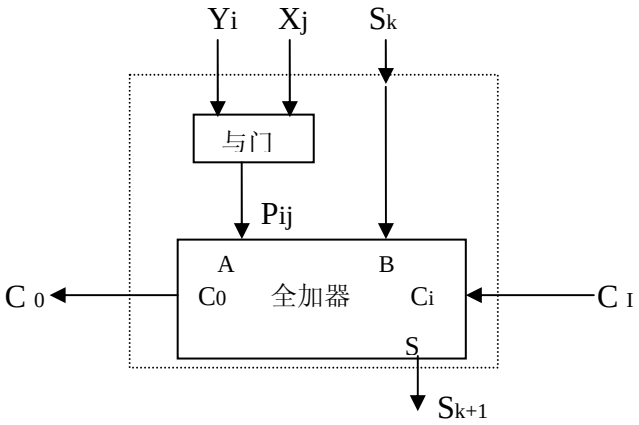


图 5.4　乘法单元（MU）

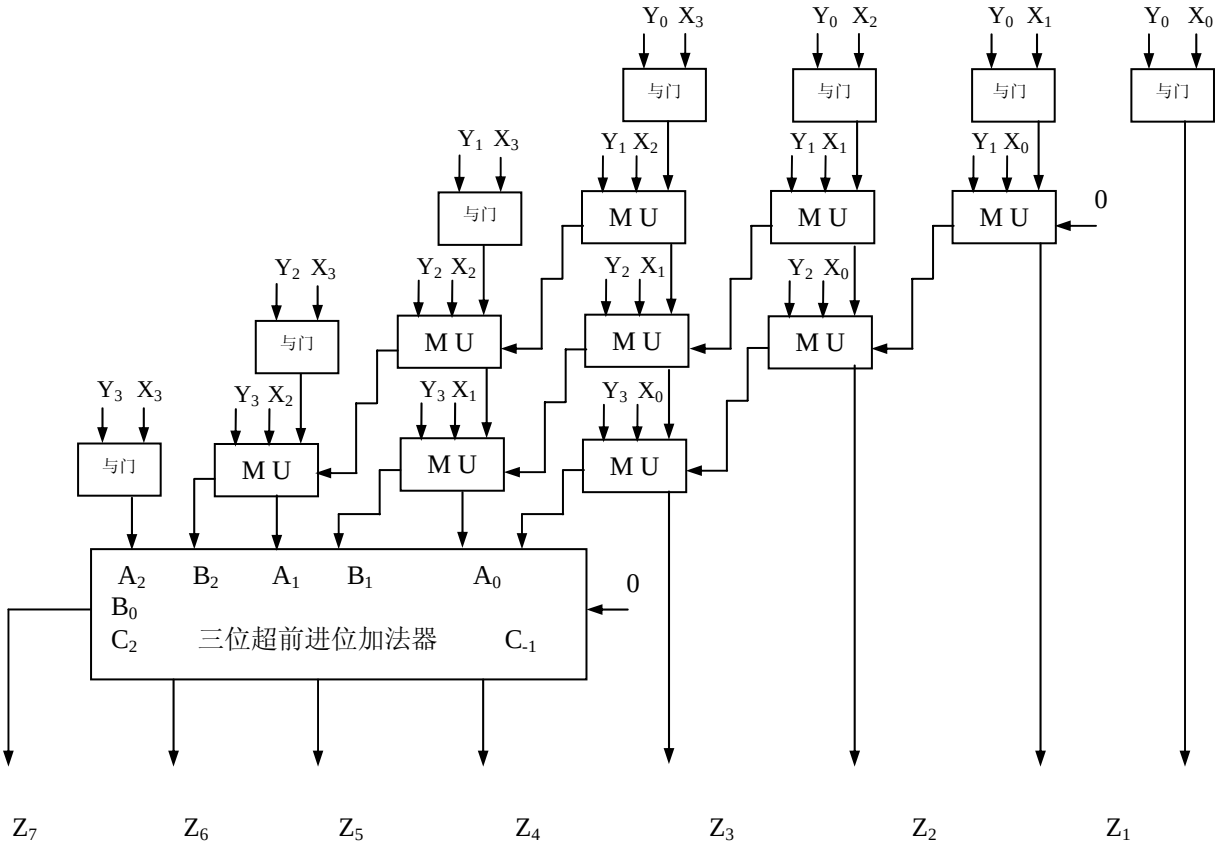


图 5.5　进位节省乘法器

为了提高乘法运算速度可以改为图 5.5 所示的进位节省乘法器（Carry-Save Multiplier）。图中用了一个三位的超前进位加法器，九个图 5.4 所示的乘法单元，七个**与**门。显然，图 5.5 中第二行的乘法单元中全加器可改为半加器。图 5.5 执行一次乘法运算的最长延时为 1 个与门的传输延时加上 3 个全加器的传输延时，再加上三位超前进位加法器的传输延时。设三位超前进位加法器的传输延时为 5 个门的传输延时，则最长延时为 1+3×4+1×5=18 的传输延时。节省乘法运算时间的关键在于每个乘法单元的进位输出向下斜送到下一行，故有进位节省乘法器之称。

根据加法器类似的道理，八位的二进制超前进位乘法电路可用两个四位二进制超前进位乘法电路再加上超前进位形成逻辑来构成。同理，依次类推可以设计出 16 位、32 位和 64 位的乘法电路。

用 Verilog HDL 来描述乘法器是相当容易的，只需要把运算表达式写出就可以了，见下例。

```
module mult_4( X, Y, Product);
input [3 : 0] X, Y;
output [7 : 0]  Product;

assign    Product = X *  Y;

endmodule
```

而 8 位乘法器只需要扩大位数即可，见下例：

```
module mult_8( X, Y, Product);
input [7 : 0] X, Y;
output [15 : 0]  Product;

assign    Product = X * Y;

endmodule
```

这样设计的乘法器在行为仿真时是没有延时的。借助综合器，可以根据以上 Verilog HDL 源代码自动将其综合成典型的乘法器电路结构。综合器有许多选项可供设计者选择，以便用来控制自动生成电路的性能。设计者可以考虑提高速度，也可以考虑节省电路元件以减少电路占用硅片的面积。综合器会自动根据选项和约束文件为你挑选一种基本乘法器的结构。有的高性能综合器还可以根据用户对运算速度的要求插入流水线结构，来提高运算器的性能。随着综合工具的发展，其资源库中将存有越来越多种类的基本电路结构，通过编译系统的分析，自动为设计者选择一种更符合设计者要求的电路结构。然后设计者通过布局布线工具生成具有布线延迟的电路，再进行后仿真，便可精确地知道该乘法器的实际延时。根据实际的延迟便可以确定使用该运算逻辑的最高频率。若需要重复使用该运算器，便可以根据此数据在控制数据流动的状态机中为其安排必要的时序。所以借助于硬件描述语言和综合工具大大加快了计算逻辑电路设计的过程。

5.3 比较器

数值大小比较逻辑在计算逻辑中是常用的一种逻辑电路，一位二进制数的比较是它的基础。下面列出了一位二进制数比较电路的真值表：

X	Y	(X > Y)	(X >= Y)	(X = Y)	(X <= Y)	(X < Y)	(X != Y)
0	0	0	1	1	1	0	0

0	1	0	0	0	1	1	1
1	0	1	1	0	0	0	1
1	1	0	1	1	1	0	0

从真值表很容易写出一位二进制数比较电路的布尔表达式如下：

$(X > Y) = X \cdot (\sim Y)$

$(X < Y) = (\sim X) \cdot Y$

$(X = Y) = (\sim X) \cdot (\sim Y) + X \cdot Y$

也很容易画出逻辑图。

位数较多的二进制数比较电路比较复杂，以前我们常用 7485 型四位数字比较器来构成位数较多的二进制数比较电路，如 8 位、16 位、24 位、32 位的比较器。同学们可以参考清华大学出版社刘宝琴老师编写的“数字电路与系统”中，有关多位并行比较器的设计的章节，在这里不再详细介绍。

用 Verilog HDL 来设计比较电路是很容易的。下面就是一个位数可以由用户定义的比较电路模块：

```
module compare_n ( X, Y, XGY, XSY, XEY);
input [width-1:0]  X, Y;
output  XGY,  XSY, XEY;
reg   XGY,  XSY,  XEY;
parameter width = 8;

always @ ( X or Y )                  // 每当 X 或 Y 变化时
      begin
            if ( X = = Y )
                  XEY = 1;       // 设置 X 等于 Y 的信号为 1
            else  XEY = 0;

            if (X > Y)
                  XGY = 1;       // 设置 X 大于 Y 的信号为 1
            else  XGY = 0;

            if (X < Y)
                   XSY = 1;      // 设置 X 小于 Y 的信号为 1
            else    XSY =  0;
      end
endmodule
```

综合工具能自动把以上原代码综合成一个八位比较器。如果在实例引用时分别改变参数 width 值为 16 和 32 综合工具就能自动把以上原代码分别综合成 16 位和 32 位的比较器。

5.4 多路器

多路选择器（Multiplexer）简称多路器，它是一个多输入、单输出的组合逻辑电路，在数字系统中有着广泛的应用。它可以根据地址码的不同，从多个输入数据中选取一个，让其输出到公共的输出端。在算法电路的实现中多路器常用来根据地址码来调度数据。我们可以很容易地写出一个有两位地址码，可以从四组输入信号线中选出一组通过公共输出端输出的功能表。

地址 1	地址 0	输入 1	输入 2	输入 3	输入 4	输出
0	0	1	0	0	0	输入 1
0	1	0	1	0	0	输入 2
1	0	0	0	1	0	输入 3
1	1	0	0	0	1	输入 4

可以很容易地写出它的布尔表达式，也很容易画出逻辑图，但是当地址码比较长，比如有 12 位长，而且每组输入信号位数较宽（如位宽为 8）信号组的数目又较多时，再加上又需多路选择使能控制信号时，其逻辑电路的基本单元需要量是较大的，如画出逻辑图来就显得很复杂，电路具体化后不易于理解，（同学们可以参考阎石老师主编的“数字电子技术基础”教材，复习多路选择器的概念）。

用 Verilog HDL 来设计多路选择器电路是很容易的。下面就是带使能控制信号的数据位宽可以由用户定义的八路数据选择器模块：

```
module Mux_8( addr,in1, in2, in3, in4, in5, in6, in7, in8, Mout, nCS);
input [2:0] addr;
input [width-1] in1, in2, in3, in4, in5, in6, in7, in8;
output [width-1] Mout;
parameter width = 8;

always @ (addr or in1 or in2 or in3 or in4 or in5 or in6 or in7 or in8)
  begin
      if (!ncs)
         case(addr)
         3'b000: Mout = in1;
         3'b001: Mout = in2;
         3'b010: Mout = in3;
         3'b011: Mout = in4;
         3'b100: Mout = in5;
         3'b101: Mout = in6;
         3'b110: Mout = in7;
         3'b111: Mout = in8;
      endcase
  else
        Mout = 0;
  end
endmodule
```

综合工具能自动把以上原代码综合成一个数据位宽为 8 的八路选一数据多路器。如果在实例引用时分别改变参数 width 值为 16 和 32，综合工具就能自动把以上原代码分别综合成数据宽度为 16 位和 32 位的八选一数据多路器。

5.5 总线和总线操作

总线是运算部件之间数据流通的公共通道。在硬线逻辑构成的运算电路中只要电路的规模允许，我们可以比较自由地来确定总线的位宽，因此可以大大提高数据流通的速度。适当的总线的位宽，配合适当并行度的运算逻辑和步骤能显著地提高专用信号处理逻辑电路的运算能力。各运算部件和数据寄存器组可以通过带控制端的三态门与总线的连接。通过对控制端电平的控制来确定在某一时间片段内，总线归哪两个或哪几个部件使用（任何时间片段只能有一个部件发送，但可以有几个接收）。用 Verilog 来描述总线和总线操作是非常简单的。下面就是一个简单的与总线有接口的模块是如何对总线进行操作的例子：

```
module SampleOfBus( DataBus, link_bus, write );
inout [11:0] DataBus;          // 总线双向端口
input link_bus;                // 向总线输出数据的控制电平
reg [11:0] outsigs;

assign DataBus = (link_bus) ? outsigs : 12 'h zzz ;
      //当 link_bus 为高电平时通过总线把存在 outsigs 的计算结果输出

always @(posedge write)        //每当 write 信号上跳沿时
   begin                       //接收总线上数据并乘以五
    outsigs <= DataBus * 5;    //把计算结果存入 outsigs
   end

endmodule
```

通过以上例子我们可以理解使这个总线连接模块能正常工作的最重要的因素是与其他模块的配合，如：何时提供 write 信号？此时 DataBus 上数据是否已正确提供？何时提供 link_bus 电平？输出的数据是否能被有效地利用？控制信号的相互配合由同步状态机控制的开关阵列控制。在第七章里我们将详细介绍如何用 Verilog HDL 来设计复杂的同步状态机并产生精确同步的开关控制信号来控制数据的正确流动。

5.6 流水线（pipeline）

流水线（pipe-line）设计技术:

流水线的设计方法已经在高性能的、需要经常进行大规模运算的系统中得到广泛的应用，如 CPU（中央处理器）等。目前流行的 CPU，如 intel 的奔腾处理器在指令的读取和执行周期中充分地运用了流水线技术以提高它们的性能。高性能的 DSP（数字信号处理）系统也在它的构件（building-block functions)中使用了流水线设计技术。通过加法器和乘法器等一些基本模块，本节讨论了有关流水线的一些基本概念，并对采用两种不同的设计方法：纯组合逻辑设计和流水线设计方法时,在性能和逻辑资源的利用等方面的不同进行了比较和权衡。

流水线设计的概念:

所谓流水线设计实际上就是把规模较大、层次较多的组合逻辑电路分为几个级，在每一级插入寄存器组暂存中间数据。K 级的流水线就是从组合逻辑的输入到输出恰好有 K 个寄存器组（分为 K 级，每一级都有一个寄存器组）上一级的输出是下一级的输入而又无反馈的电路。

图 5.6 表示了如何将把组合逻辑设计转换为相同组合逻辑功能的流水线设计。这个组合逻辑包括两级。第一级的延迟是 T1 和 T3 两个延迟中的最大值；第二级的延迟等于 T2 的延迟。为了通过这个组合逻辑得到稳定的计算结果输出，需要等待的传播延迟为[max(T1,T3)+T2]个时间单位。在从输入到输出的每一级插入寄存器后，流水线设计的第一级寄存器所具有的总的延迟为 T1 与 T3 时延中的最大值加上寄存器的 Tco（触发时间）。

同样，第二级寄存器延迟为 T2 的时延加上 Tco。采用流水线设计为取得稳定的输出总体计算周期为:

max(max(T1,T3)+Tco,(T2+Tco))

流水线设计需要两个时钟周期来获取第一个计算结果，而只需要一个时钟周期来获取随后的计算结果。开始时用来获取第一个计算结果的两个时钟周期被称为采用流水线设计的首次延迟（latency)。对于 CPLD 来说，器件的延迟如 T1、T2 和 T3 相对于触发器的 Tco 要长得多，并且寄存器的建立时间 Tsu 也要比器件的延迟快得多。只有在上述关于硬件时延的假设为真的情况下，流水线设计才能获得比同功能的组合逻辑设计更高的性能。

采用流水线设计的优势在于它能提高吞吐量（throughput)。假设 T1、T2 和 T3 具有同样的传递延迟 Tpd。对于组合逻辑设计而言，总的延迟为 2*Tpd。对于流水线设计来说，计算周期为(Tpd+Tco)。前面提及的首次延迟（latency）的概念实际上就是将（从输入到输出）

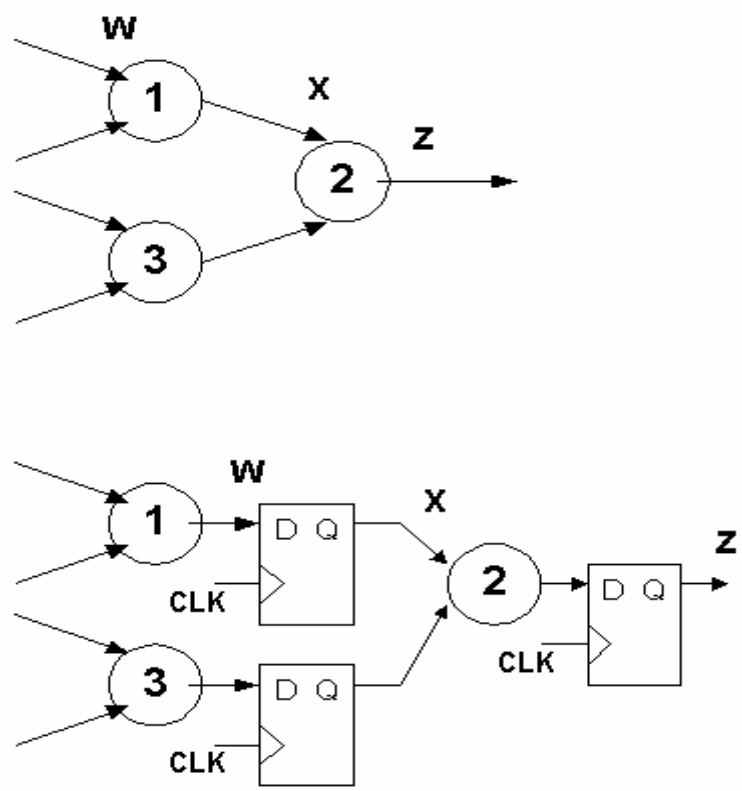


图 5.6　　组合逻辑设计转化为流水线设计

最长的路径进行初始化所需要的时间总量；吞吐延迟则是执行一次重复性操作所需要的时间总量。在组合逻辑设计中，首次延迟和吞吐延迟同为 2*Tpd。与之相比，在流水线设计中，首次延迟是 2*（Tpd+Tco)，而吞吐延迟是 Tpd+Tco。如果 CPLD 硬件能提供快速的 Tco，则流水线设计相对于同样功能的组合逻辑设计能提供更大的吞吐量。典型的富含寄存器资源的 CPLD 器件（如 Lattice 的 ispLSI 8840）的 Tpd 为 8.5ns，Tco 为 6ns。

流水线设计在性能上的提高是以消耗较多的寄存器资源为代价的。对于非常简单的用于数据传输的组合逻辑设计，例如上述例子，将它们转换成流水线设计可能只需增加很少的寄存器单元。随着组合逻辑变得复杂，为了保证中间的计算结果都在同一时钟周期内得到，必须在各级之间加入更多的寄存器。如果需要在 CPLD 中实现复杂的流水线设计，以获取更优良的性能，具有丰富寄存器资源的 CPLD 结构并且具有可预测的延迟这两大特点的 FPGA 是一个很有吸引力的选择。

流水线加法器与组合逻辑加法器的比较

采用流水线技术可以在相同的半导体工艺的前提下通过电路结构的改进来大幅度地提高重复多次使用的复杂组合逻辑计算电路的吞吐量。下面是一个 n 位全加器的例子，如图 5.7 所示为实现该加法功能需要三级电路：（1）加法器输入的数据产生器和传送器；（2）数据产生器和传送器的超前进位部分；（3）数据产生、传送功能和超前进位三者求和部分。

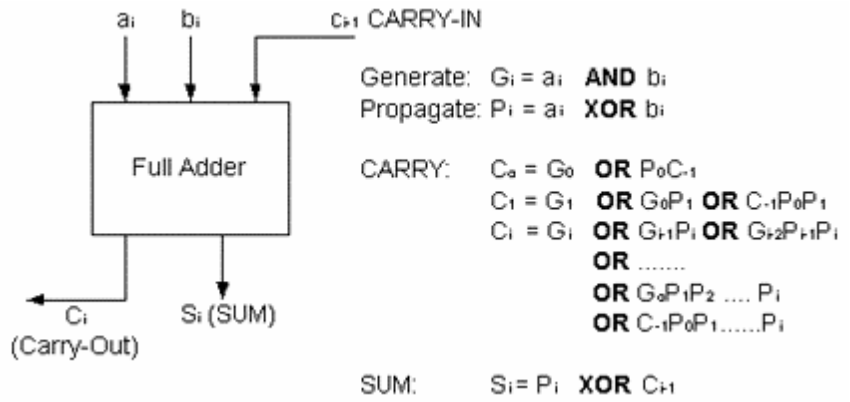
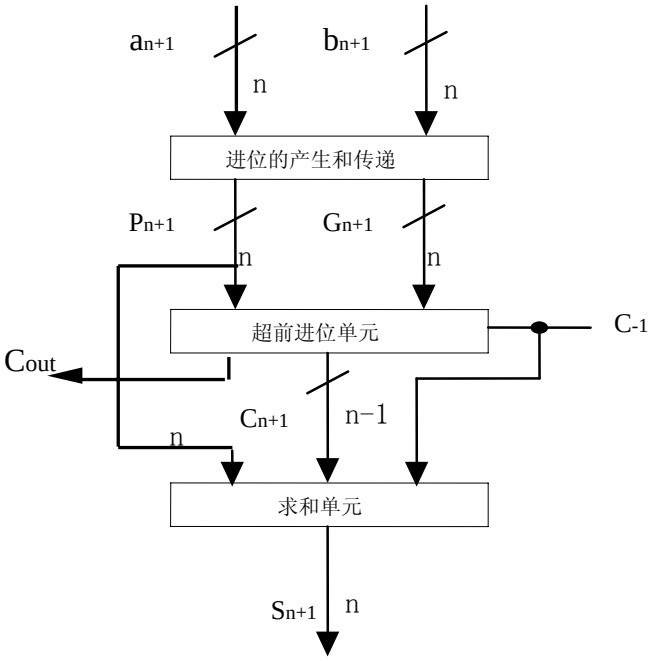


图 5.7　　n 位全加器的方程式

在 n 位组合逻辑全加器中插入三层寄存器或寄存器组，将它转变为 n 位流水线全加器，如图 5.8(b)所示。由于进位 C_{-1} 既是第一级逻辑的输入，又是第二级逻辑输入，因此将 C_{-1} 进位改为流水线结构时需要使用两级寄存器。同样地，发生器输出在作为求和单元的输入之前，也要多次插入寄存器。作为求和单元的输出，进位 Cout 要达到同一流水线的级别也需要插入两层寄存器。



(a) n 位纯组合逻辑全加器

若用拥有 840 个宏单元和 312 个有寄存能力 I/O 单元的 Lattice ispLSI8840 分别来实现 16 位组合逻辑全加器和 16 位流水线全加器并比较它们的运行速度， 对于 16 位组合逻辑全加器， 共用了 34 个宏单元。执行一次计算需经过 3 个 GLB 层，每次计算总延迟为

45.6ns。而 16 位流水线全加器共用了 81 个宏单元。执行一次计算只需经过 1 个 GLB 层，每次计算总延迟为 15.10ns（但第一次计算需要多用三个时钟周期），吞吐量约增加了三倍。

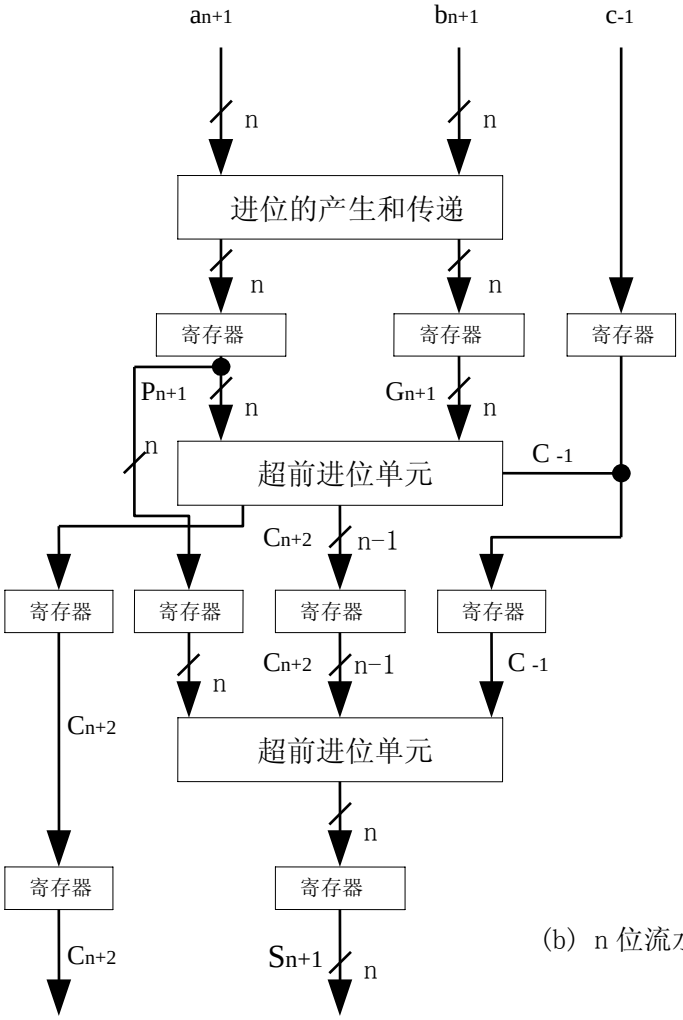


图 5.8 n 位纯组合逻辑全加器（a） 改进为 n 位流水线全加器 (b)

流水线乘法器与组合逻辑乘法器的比较：

首先，我们使用一个 4*4 乘法器的例子来说明部分积乘法器的基本概念。然后，通过一个复杂得多的 6*10 乘法器来比较流水线乘法器和组合逻辑乘法器这两个不同设计方法的实现在性能上有何差异。

如图 5.9 所示，4*4 乘法器可以被分解为部分积的向量和（或称加权和），比如说是 16 个 1*1 乘法器输出的向量和。这里并没有直接在 4*4 乘法器的每一级都插入寄存器以达到改为流水线结构的目的，而是将其分割为 1*4 乘法器来产生所有的部分积向量。这样分割的结果是形成了两级的流水线设计，相对 1*1 乘法器的组合具有更短的首次延迟,而吞吐延迟相同。每一级的流水线求和用图 5.8(b)所示的流水线加法器来实现。

我们用一个类似图 5.9 中的 4*4、但更为复杂的 6*10 流水线乘法器来比较流水线乘法器与非流水线乘法器之间性能上的差异。如图 5.10 所示，该 6*10 流水线乘法器采用 6 个 10 位

乘法器来实现 1*10 乘法--a0 * b[9:0], a1 * b[9:0], a2 * b[9:0], a3 * b[9:0] , a4 * b[9:0], a5 * b[9:0] 。由于 ai 非 0 即 1，那么 1*10 乘法器的结果是 b[9:0] 或 0。这表示下一级的两个输入不是 b[9:0] 就是 0。

这六个多路器的输出被两两一组分成三个相互独立的组合，并分别用一个 3 层的流水线加法器加起来。每一组的两个多路输入的下标号差为 3。在这个例子里，这些组是如下组织的：[a5，a2],[a4,a1],[a3,a0]。[a5,a2]意味着第一个多路器的输出 M（10 位）和第四个多路器的输出 N（10 位）是流水线加法器 O 的输入。同样地，其余的两组分别用流水线加法器 P 和 Q 加在一起。这样的两两组合能在加的过程中去除额外的部分积项。以[a5,a2]为例，其等式一般表示为：

G(j,0) = {000, M(i,0)} and {N(i,0), 000}
P(j,0) = {000, M(i,0)} xor {N(i,0), 000}　　(0 <= i <= 9, 0<= j <= 12)
Cj = Gj or Gj-1Pj or Gj-2Pj-1Pj　　(0<= j <= 12)
　　or ...or G0P1P2P3...Pj
Sk = Pk xor Ck-1 (0 <= k <= 13)

由于 M 与 N 的间隔为 3，M 的高三位和 N 的低三位必定是 0。因此，M 和 N 完成与操作后，G0，G1，G2 和 G10，G11，G12 必定为 0。进一步地说，因为存在这样一些结果为 0 的发生器，进位的计算就可以得到简化。既然进位计算得到了简化，那么求和运算也就自然得到了简化。同样地，流水线加法器 P，Q 的输入间隔也是 3。流水线加法器 T 和 S 的输入之间的间隔分别为 1 和 2。由于加法器 T 是一个三层流水线加法器，所以在 Q 和 S 之间也插入了三层寄存器组从而达到与 T 相同的流水线级别。

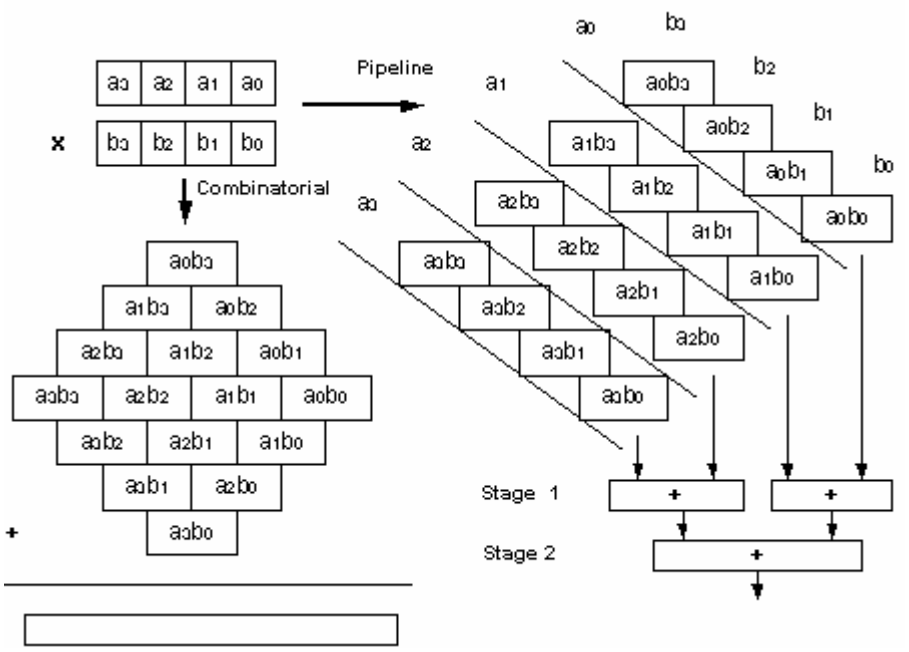


图 5.9　4 位组合逻辑乘法器与 4 位流水线乘法器的比较

这里依然使用 Lattice 的 ispLSI8840 来比较 6*10 乘法器分别用组合逻辑和流水线实现的执行情况。组合逻辑的 6*10 乘法器在用 HDL 实现时消耗了 14 个 GLB 中的 93 个宏单元。执行一次运算需要经过 5 个 GLB 层，具有的最大传递延迟为 73.5ns。相应的是，流水线设计的 6*10 乘法器在用 HDL 实现时消耗了 22 个 GLB 中的 360 个宏单元。执行一次运算只需经过一个 GLB 层，计算周期只要 15.30ns，比组合逻辑的实现快 4 倍有余。该设计的相应首

次延迟是 9 个时钟周期。

总结：

改为流水线结构是提高组合逻辑吞吐量从而增强计算性能的一个重要办法。为获取高性能所付出的代价是要使用更多的寄存器。要实现这样大规模的运算部件，只含少量寄存器资源的普通 PLD 器件是无法办到的，必须使用拥有大量寄存器资源的 CPLD 或 FPGA 器件或设计专用的 ASIC。当用 Verilog 语言描述流水线结构的运算部件时，要使用结构描述，才能够真正综合成设计者想要的流水线结构。简单的运算符表达式只有在综合库中存有相应的流水线结构的宏库部件时，才能综合成流水线结构从而显著地提高运算速度。从这一意义上来说，深入了解和掌握电路的结构是进行高水平 HDL 设计的基础。

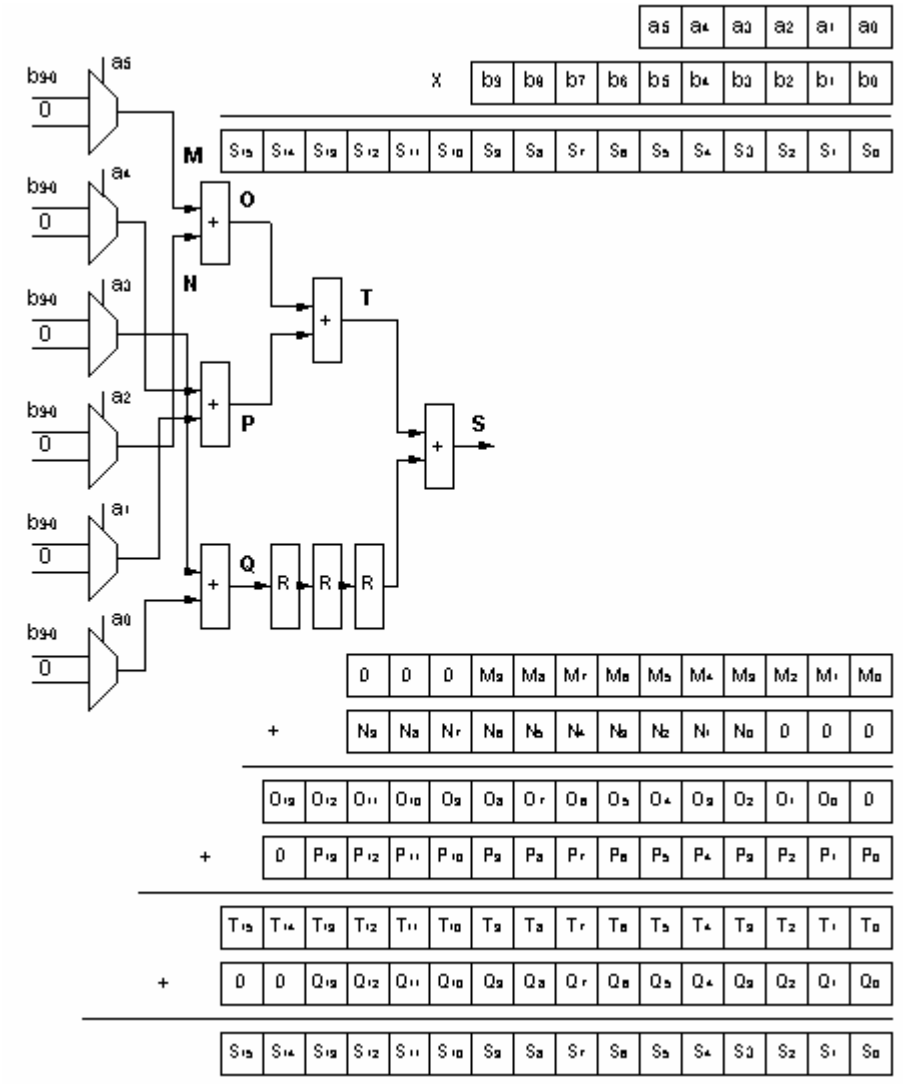


图 5.10　　6*10 流水线乘法器

思考题：

1）写出 8 位加法器和 8 位乘法器的逻辑表达式，比较用超前进位逻辑和不用超前进位逻辑的延迟。

2）提高复杂运算组合逻辑运算速度有哪些办法？

3）详细解释为什么采用流水线的办法可以显著提高层次多的复杂组合逻辑的运算速度。

第六章 运算和数据流动控制逻辑

6. 前言：

6.1 数字逻辑电路的种类：

- 组合逻辑：输出只是当前输入逻辑电平的函数（有延时），与电路的原始状态无关的逻辑电路。也就是说，当输入信号中的任何一个发生变化时，输出都有可能会根据其变化而变化，但与电路目前所处的状态没有任何关系。
- 时序逻辑：输出不只是当前输入的逻辑电平的函数，还与电路目前所处的状态有关的逻辑电路。

同步有限状态机是同步时序逻辑的基础。所谓同步有限状态机是电路状态的变化只可能在在同一时钟跳变沿时发生的逻辑电路。但状态是否发生变化还要看输入条件，如输入条件满足，则进入下一状态，否则即使时钟不断跳变，电路系统仍停留在原来的状态。利用同步有限状态机可以设计出极其复杂灵活的数字逻辑电路系统，产生各种有严格时序和条件要求的控制信号波形，有序地控制计算逻辑中数据的流动。

6.2 数字逻辑电路的构成

- 组合逻辑：由与、或、非门组成的网络。常用的组合电路有：多路器、数据通路开关、加法器、乘法器....
- 时序逻辑： 由多个触发器和多个组合逻辑块组成的网络。常用的有：计数器、复杂的数据流动控制逻辑、运算控制逻辑、指令分析和操作控制逻辑。同步时序逻辑是设计复杂的数字逻辑系统的核心。时序逻辑借助于状态寄存器记住它目前所处的状态。在不同的状态下，即使所有的输入都相同，其输出也不一定相同。

组合逻辑举例之一：一个八位数据通路控制器

它的Verilog HDL描述如下：

```
`define   ON   1 'b 1
`define   OFF  1 'b 0
 wire ControlSwitch;
 wire [7:0] Out, In;
 assign  Out =  (ControlSwith = = `ON) ? In  : 8 'h00;
```

它的逻辑电路结构如下：

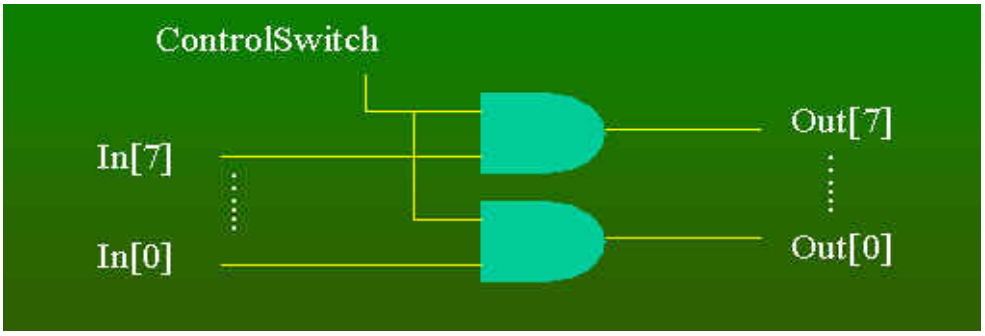


图 6.1 数据通道开关的逻辑图

它对数据通路所起的作用如下：

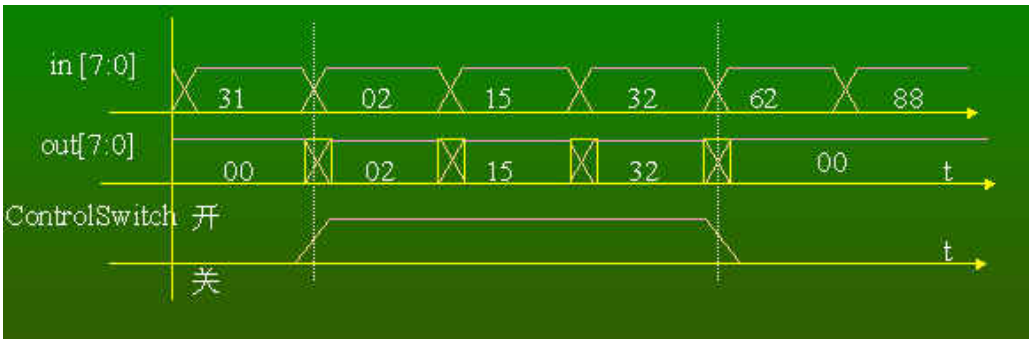


图 6.2　　数据通道的开关和数据流波形图

组合逻辑举例之二：一个八位三态数据通路控制器

它的Verilog HDL描述如下：

```
`define   ON   1 'b 1
`define  OFF  1 'b 0
 wire  LinkBusSwitch;
 wire [7:0]  outbuf;
 inout [7:0] bus;
 assign  bus =  (LinkBusSwitch== `ON) ? outbuf : 8 'hzz
```

它的逻辑电路结构和对数据通路的作用如下：

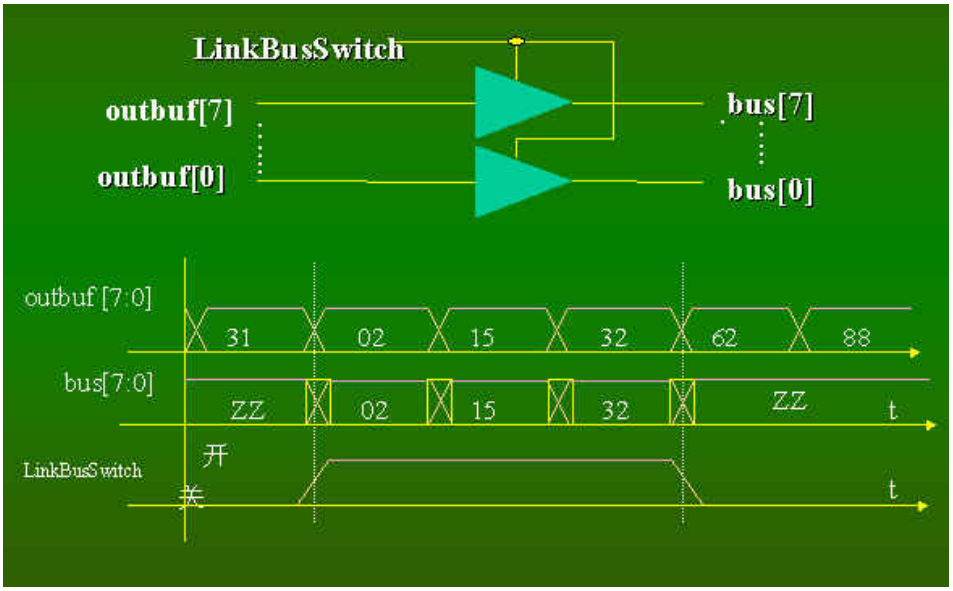


图 6.3 三态数据通道开关逻辑图和数据流通断波形图

它与组合逻辑举例之一的差别只在前者在开关断开时输出为零，而后者在开关断开时输出为高阻，即与总线脱离连接。

6．3 数据流动的控制：

我们知道，诸如加、减、乘、除、比较等运算都可以用组合逻辑来实现，但运算的输入必须稳定一段时间，才可能得到稳定的输出，而输出要被下一阶段的运算作为输入，也必须要有一段时间的稳定，因而输出结果必须保存在寄存器组中。在计算电路中设有许多寄存器组，它们是用来暂存运算的中间数据。对寄存器组之间数据流动进行精确的控制，在算法的实现的过程中有着极其重要的作用。这种控制是由同步状态机实现的。

开关逻辑应用举例：

设想下面的组合逻辑是一个乘法器，把输入的数乘 3，然后输出。因为乘法器是由门组成的，所以会有延迟，从图上看，为了取得稳定的输出需要 10ns 的延迟。如果能有效地控制 Sn 的开关时间就可以取得稳定的输出，把运算结果存入寄存器。

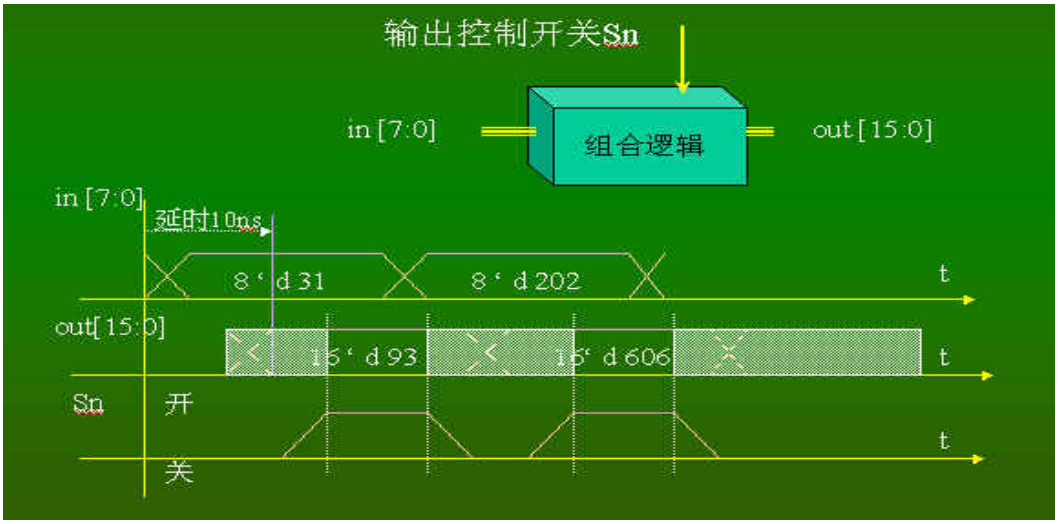


图 6.4　带输出控制开关的运算组合逻辑和数据流波形

设想下图中由开关 S1 和开关 S2 控制的两个组合逻辑都是运算逻辑，例如乘法器或加法器等，而寄存器 A，B，C 是用来寄存运算的输入、中间和输出数据的。如果能与时钟配合来精确地控制开关的闭合和断开，在寄存器中暂存的中间或输出数据都会是上一步运算的稳定结果，而不会出现冒险和竞争的现象。

设想下图中由开关 S1、S3、S5 控制的三个组合逻辑都是运算逻辑，例如乘法器或加法器等，而寄存器组 A，B，C 是用来寄存运算的输入、中间和输出数据的。开关 S2、S4、S6 是三态门，能控制寄存器组 A，B，C 的输出到总线上还是与总线隔离。 如果能与时钟配合来精确地控制 S1 到 S6 开关的闭合和断开，在寄存器中暂存的中间或输出数据都会是上一步运算的稳定结果，而不会出现冒险和竞争的现象。运算的过程可以在这几个寄存器组内反复地执行，直到通过开关的控制使其停止。下面让我们通过简单的描述来说明一个极其重要的概念：**生成与时钟精确配合的开关时序是计算逻辑的核心。**

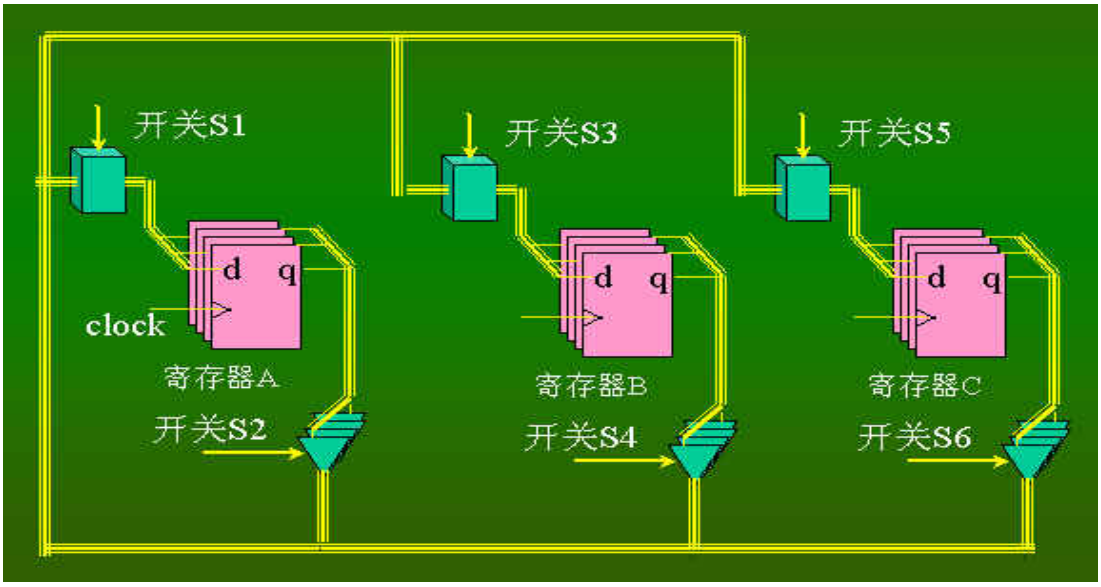


图 6.6 由开关逻辑控制的数据流动和计算逻辑结构示意图

我们在“数字电子技术基础”中已经知道当时钟正跳变沿到来时，在 D 触发器数据端口的数据才能存入触发器中。我们也知道当组合逻辑的输入变化时，输出必须经过一段时间后才能稳定，这是由于门级电路和布线的延迟造成的。只有稳定的输出对运算才是有意义的。如果我们想把寄存器组 C 中的数据经过组合逻辑的运算存入寄存器组 A 中，我们应该如何来控制这几个开关呢？从上面的描述我们知道开关 S1、S3、S5 分别控制着三个组合运算逻辑的输出。如果 S1、S3、S5 切断数据通道，则组合运算逻辑的输出总是为零（即每一个输出线总为低电平）。当时钟正跳变沿到来时这三个寄存器组将全部清零。为了要把寄存器组 C 中的数据送出，必须在时钟正跳变沿到来前接通 S6，待运算组合逻辑输出稳定后接通 S1,在时钟正跳变到来时便可把稳定的结果存入寄存器组 A 中。此时寄存器组 C 中的数据已被清零，因为这时开关 S3、S5、S2、S4 必须把数据通道断开，寄存器组 C 端口的零电平被存入寄存器组 C。但由原来寄存在寄存器组 C 中的数据所生成的结果已稳定地存入寄存器组 A 中。同理断开所有的通道,只按时序先后接通 S2、S3，在下一运算时钟前沿到来时就能稳定地把由 S3 控制的运算结果存入寄存器组 B。这个简单的例子说明：如果我们能设计出一个状态机，在这个状态机的控制下生成一系列的开关信号，严格按时钟的节拍来开启或关闭数据通道，我们就能用硬件来构成复杂的计算逻辑，如果硬件的规模可以达到几十到几千万门，我们就可以设计出并行度很高的高速计算逻辑。在下一章里我们将详细地介绍怎样用 VerilogHDL 来编写可综合的复杂同步状态机。

6．4 为什么在 Verilog HDL 设计中一定要用同步而不能用异步时序逻辑

同步时序逻辑是指表示状态的寄存器组的值只可能在唯一确定的触发条件发生时刻改变。只能由时钟的正跳沿或负跳沿触发的状态机就是一例。always @(posedge clock) 就是一个同步时序逻辑的触发条件，表示由该 always 控制的 begin end 块中寄存器变量重新赋值的情形只有可能在 clock 正跳沿发生。而异步时序逻辑是指触发条件由多个控制因素组成，任何一个因素的跳变都可以引起触发。记录状态的寄存器组其时钟输入端不是都连结在同一个时钟信号上。例如用一个触发器的输出连结到另一个触发器的时钟端去触发的就是异步时序逻辑。

用 Verilog HDL 设计的可综合模块,必须避免使用异步时序逻辑，这不但是因为许多综合器不支持异步时序逻辑的综合，而且也因为用异步时序逻辑确实很难来控制由组合逻辑和延迟所产生

的冒险和竞争。当电路的复杂度增加时，异步时序逻辑无法调试。工艺的细微变化也会造成异步时序逻辑电路的失效。因为异步时序逻辑中触发条件很随意，任何时刻都有可能发生，所以记录状态的寄存器组的输出在任何时刻都有可能发生变化。而同步时序逻辑中的触发输入至少可以维持一个时钟后才会发生第二次触发。这是一个非常重要的差别，因为我们可以利用这一个时钟的时间在下一次触发信号来到前，为电路状态的改变创造一个稳定可靠的条件。因此我们可以得出结论：同步时序逻辑比异步时序逻辑具有更可靠更简单的逻辑关系。**如果我们强行作出规定，用 Verilog 来设计可综合的状态机必须使用同步时序逻辑,有了这个前提条件，实现自动生成电路结构的综合器就有了可能。**因为这样做大大减少了综合工具的复杂度，为这种工具的成熟创造了条件。也为 Verilog 可综合代码在各种工艺和 FPGA 之间移植创造了条件。Verilog RTL 级的综合就是基于这个规定的。下面我们将详细说明同步与异步时序逻辑的差异。

在同步逻辑电路中，触发信号是时钟（clock）的正跳沿（或负跳沿）；触发器的输入与输出是经由两个时钟来完成的。第一个时钟的正跳沿（或负跳沿）为输入作准备，在第一个时钟正跳沿（或负跳沿）到来后到第二个时钟正跳沿（或负跳沿）到来之前的这一段时间内，有足够的时间使输入稳定。当第二个时钟正跳沿（或负跳沿）到来时刻，由前一个时钟沿创造的条件已经稳定，所以能够使下一个状态正确地输出。

若在同一时钟的正跳沿（或负跳沿）下对寄存器组既进行输入又进行输出，很有可能由于门的延迟使输入条件还未确定时，就输出了下一个状态，这种情况会导致逻辑的紊乱。而利用上一个时钟为下一个时钟创造触发条件的方式是安全可靠的。但这种工作方式需要有一个前提：确定下一个状态所使用的组合电路的延迟与时钟到各触发器的差值必须小于一个时钟周期的宽度。只有满足这一前提才可以避免逻辑紊乱。在实际电路的实现中，采取了许多有效的措施来确保这一条件的成立，其中主要有以下几点：

（1）全局时钟网络布线时尽量使各分支的时钟一致；

（2）采用平衡树结构，在每一级加入缓冲器，使到达每个触发器时钟端的时钟同步。（如图 1、2 所示）

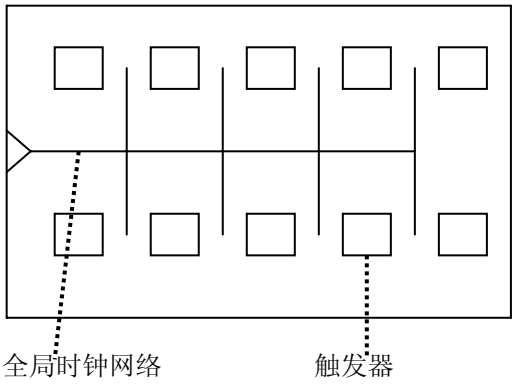


图 1　全局时钟网示意图

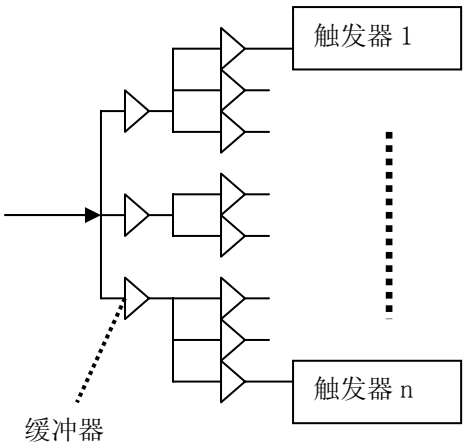


图 2　平衡树结构示意图

通过这些措施基本可以保证时钟的同步，在后仿真时，若逻辑与预期设计的不一样，可降低时钟频率，就有可能消除由于时钟过快引起的触发器输入端由延迟和冒险竞争造成的不稳定从而使逻辑正确。

在组合逻辑电路中，多路信号的输入使各信号在同时变化时很容易产生竞争冒险，从而结果难以预料。下面就是一个简单的组合逻辑的例子：C = a & b;

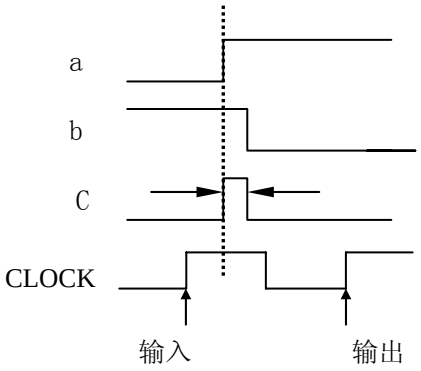


图 3　由于 a,b 变化不同步导致组合电路竞争冒险产生毛刺和防止办法

a 和 b 变化不同步使 C 产生了一个脉冲。这个结果也许与当初设计时的想法并不一致，但如果我们能过一段时间，待 C 的值稳定后再来取用组合逻辑的运算结果，就可以避免竞争冒险。同步时序逻辑由于用上一个时钟的跳变沿时刻（置寄存器作为组合逻辑的输入）来为下一个时钟的跳变沿时刻的置数（置下一级寄存器作为该组合逻辑的输出）做准备，只要时钟周期足够长，就可以在下一个时钟的跳变沿时刻得到稳定的置数条件，从而在寄存器组中存入可靠的数据。而这一点用异步电路是做不到的，因此在实际设计中应尽量避免使用异步时序逻辑。若用弥补的方法来避免竞争冒险，所耗费的人力物力是很巨大的。也无法使所设计的 Verilog HDL 代码和已通过仿真测试的电路模块结构有知识产权的可能，因为工艺的细微改变就有可能使电路无法正常工作。显而易见使用异步时序逻辑会带来设计的隐患，无法设计出能严格按同一时间节拍操作控制数据流动方向开关的状态机。而这种能按时钟节拍精确控制数据流动开关的状态机就是在下一章里我们将详细介绍的同步有限状态机。它是算法计算过程中数据流动控制的核心。计算结构的合理配置和运算效率的提高与算法状态机的设计有着非常密切的关系。我们只有通过阅读有关计算机体系结构的资料和通过大量的设计实践才能熟练地掌握复杂算法系统的设计。

思考题：

1) 利用数字电路的基本知识解释：为什么说即使组合逻辑的输入端的所有信号同时变化，其输出端的各个信号不可能同时达到新的值？ 各个信号变化的快慢由什么决定？。

2) 为使运算组合逻辑有一个确定的输出，为什么在运算组合逻辑的输入端和输出端必须具有寄存器组来寄存数据？

3) 对每一个寄存器组来说，上一个时钟的正跳沿是为置数做准备，下一个时钟正跳沿是把本寄存器组置数（并为下一级运算组合逻辑输入），为下一级寄存器组的置数做准备的先决条件是什么？

第七章 有限状态机和可综合风格的Verilog HDL

前言

由于Verilog HDL和 VHDL 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Verilog HDL 和VHDL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，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。本教材中有关可综合风格的Verilog HDL的内容，我们只着重介绍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。在本章中我们将通过各种实例由浅入深地来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。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。为了更好地掌握可综合风格，还需要较深入地了解阻塞和非阻塞赋值的差别和在不同的情况下正确使用这两种赋值的方法。只有深入地理解阻塞和非阻塞赋值语句的细微不同，才有可能写出不仅可以仿真也可以综合的Verilog HDL模块。只要按照一定的原则来编写代码就可以保证Verilog模块综合前和综合后仿真的一致性。符合这样条件的可综合模块是我们设计的目标，因为这种代码是可移植的，可综合到不同的FPGA和不同工艺的ASIC中，是具有知识产权价值的软核。

7.1.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）只可能在同一时钟跳变沿的情况下才能从一个状态转向另一个状态，究竟转向哪一状态还是留在原状态不但取决于各个输入值，还取决于当前所在状态。*（这里指的是米里Mealy型有限状态机，而莫尔Moore型有限状态机究竟转向哪一状态只决于当前状态。）*

在Verilog HDL中可以用许多种方法来描述有限状态机，最常用的方法是用always语句和case语句。下面的状态转移图表示了一个有限状态机，例1的程序就是该有限状态机的多种Verilog HDL模型之一：

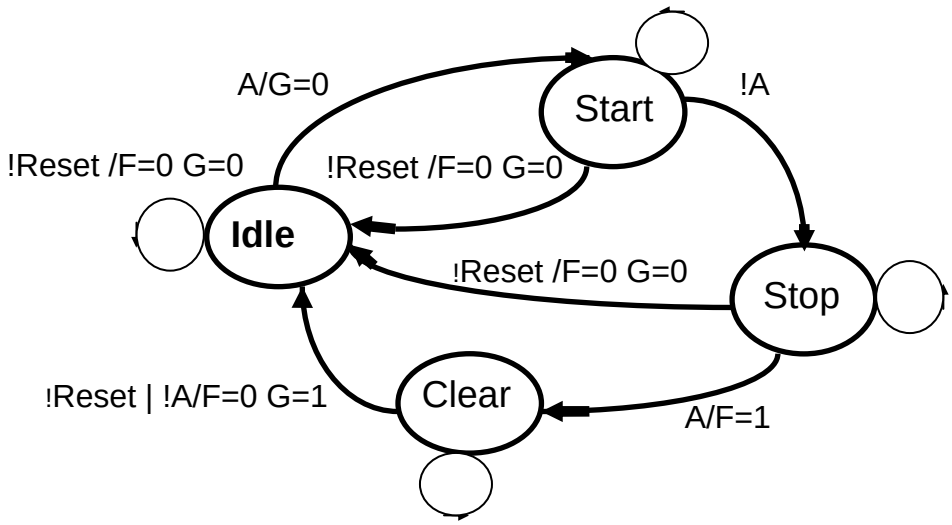


图7.1　状态转移图

上面的状态转移图表示了一个四状态的有限状态机，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）的上升沿时发生，往哪个状态的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)  begin
                                 state <= Start;
                                 G<=0;
                               End
                       else    state <= idle;
                     end
                start: if (!A)  state <= Stop;
                        else    state <= start;
                Stop:  begin
                         if (A) begin
                                  state <= Clear;
                                  F <= 1;
                              end
                         else   state <= Stop;
                       end
                Clear: begin
                        if (!A)  begin
                                   state <=Idle;
                                   F<=0;  G<=1;
                                 End
                         else   state <= Clear;
                       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，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)  begin
                                 state <= Start;
                                   G<=0;
                                end
                       else state <= Idle;
                     end
              Start: if (!A)  state <= Stop;
                      else  state <= Start;
              Stop:  begin
                       if (A) begin
                                state <= Clear;
                                F <= 1;
                              End
                       Else  state <= Stop;
                     end
              Clear: begin
                       if (!A)  begin
                                  state <=Idle;
                                  F<=0;  G<=1;
                                 End
                       else   state <= Clear;
                     end
              default:  state <=Idle;
             endcase
endmodule
```

[例2]与[例1]的主要不同点是状态编码，[例2]采用了独热编码，而[例1]则采用Gray码，究竟采用哪一种编码好要看具体情况而定。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，但所用组合电路可省下许多，因而使电路的速度和可靠性有显著提高，而总的单元数并无显著增加。采用了独热编码后有了多余的状态，就有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。

我们还可以再用另一种风格的Verilog HDL模型来表示同一个有限状态，在这个模型中，我们用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部分来描述。见下例：

[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 == Stop) && A );
  assign   G =  (( state == Clear) && (!A || !Reset));

endmodule
```

我们还可以再用另一种风格的Verilog HDL模型来表示同一个有限状态，在这个模型中，我们分别用沿触发的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部分来描述。见下例：

[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
```

上面四个例子是同一个状态机的四种不同的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四个例子进行综合，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。

有限状态机设计的一般步骤：

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：

- 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。
- 定义输入、输出逻辑状态的含意，并将电路状态顺序编号。
- 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去，并得到一样的输出，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，设计的电路可以简单，反之，选得不好，则设计的电路就会复杂许多。实际设计时，需综合考虑电路复杂度与电路性能之间的折衷，在触发器资源丰富的FPGA或ASIC设计中采用独热编码（one-hot-coding）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方便实现。具体的逻辑化简及逻辑电路到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步及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。

7.1.1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（one hot state machine)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和设置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若偶然进入多余状态后任能在下一时钟跳变沿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，我们建议采用电路图输入的方法，而不要用Verilog HDL输入的方法。**

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。使用参数(parameters)语句赋状态值见下例：

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

使用宏定义(define)语句赋状态值见下例:

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

7.1.2典型的状态机实例

[例1] 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
```

7.1.3.综合的一般原则

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。

7.1.4.语言指导原则

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又同时表示组合逻辑。但是不推荐使用这种描述方法，因为这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 关键字的表示组合逻辑或电平敏感的锁存器，或者两种都表示。在表示时序和组合逻辑的事件控制表达式中如有多个沿和多个电平，其间必须用关键字 “ or ” 连接 。**

4) **每个表示时序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需定义成reg型或整型。**整型变量缺省为32bit，使用Verilog操作符可对其进行二进制求补的算术运算。综合器还支持整型量的范围说明，这样就允许产生不是32位的整型量。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需都有明确的值；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。如果在设计中存在综合器认为不是电平敏感锁存器的组合回路时，综合器会发出错误信息(例如设计中有异步状态机时)。

上面这一段不太好理解,让我们再解释一下,这也就是说,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需有明确的值[即在赋值表达式右端参与赋值的信号都必需在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,它的作用才显现出来,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
```

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块也对其赋值，这是非法的。

2) 把某一信号值赋为'bx，综合器就把它解释成无关状态，因而综合器为其生成的硬件电路最简洁。

7.2. 可综合风格的Verilog HDL模块实例：

7.2.1.组合逻辑电路设计实例

[例1] 八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块设计比较后重组信号的组合逻辑.

```
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:

```
module 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 trist1( 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
```

7.2.2. 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，就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]. 八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]. 八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
```

7.2.3.状态机的置位与复位

7.2.3.1. 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。把它们列入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:

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。

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
```

7.2.3.2. 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沿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。

事件控制语法:

@(<沿关键词 时钟信号>)

其中沿关键词指 posedge（正沿触发）或 negedge（负沿触发）

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
```

7.2.4.深入理解阻塞和非阻塞赋值的不同

阻塞和非阻塞赋值的语言结构是 Verilog 语言中最难理解概念之一。甚至有些很有经验的 Verilog 设计工程师也不能完全正确地理解：何时使用非阻塞赋值何时使用阻塞赋值才能设计出符合要求的电路。他们也不完全明白在电路结构的设计中，即可综合风格的 Verilog 模块的设计中，究竟为什么还要用非阻塞赋值，以及符合 IEEE 标准的 Verilog 仿真器究竟如何来处理非阻塞赋值的仿真。本小节的目的是尽可能地把阻塞和非阻塞赋值的含义详细地解释清楚，并明确地提出可综合的 Verilog 模块编程在使用赋值操作时应注意的要点,按照这些要点来编写代码就可以避免在 Verilog 仿真时出现冒险和竞争的现象。我们在前面曾提到过下面两个要点：

- **在描述组合逻辑的 always 块中用阻塞赋值，则综合成组合逻辑的电路结构。**
- **在描述时序逻辑的 always 块中用非阻塞赋值，则综合成时序逻辑的电路结构。**

为什么一定要这样做呢？回答是，这是因为要使综合前仿真和综合后仿真一致的缘故。如果不按照上面两个要点来编写 Verilog 代码，也有可能综合出正确的逻辑，但前后仿真的结果就会不一致。

为了更好地理解上述要点，我们需要对 Verilog 语言中的阻塞赋值和非阻塞赋值的功能和执行时间上的差别有深入的了解。为了解释问题方便下面定义两个缩写字：

RHS – 方程式右手方向的表达式或变量可分别缩写为： RHS 表达式或 RHS 变量。
LHS – 方程式左手方向的表达式或变量可分别缩写为： LHS 表达式或 LHS 变量。

IEEE Verilog 标准定义了有些语句有确定的执行时间，有些语句没有确定的执行时间。若有两条或两条以上语句准备在同一时刻执行，但由于语句的排列次序不同（而这种排列次序的不同是 IEEE Verilog 标准所允许的）， 却产生了不同的输出结果。这就是造成 Verilog 模块冒险和竞争现象的原因。为了避免产生竞争，理解阻塞和非阻塞赋值在执行时间上的差别是至关重要的。

阻塞赋值

阻塞赋值操作符用等号(即 =)表示。为什么称这种赋值为阻塞赋值呢？这是因为在赋值时先计算等号右手方向（RHS）部分的值，这时赋值语句不允许任何别的 Verilog 语句的干扰，直到现行的赋值完成时刻，即把 RHS 赋值给 LHS 的时刻，它才允许别的赋值语句的执行。一般可综合的阻塞赋值操作在 RHS 不能设定有延迟，(即使是零延迟也不允许)。从理论上讲，它与后面的赋值语句只有概念上的先后，而无实质上的延迟。 若在 RHS 加上延迟，则在延迟期间会阻止赋值语句的执行, 延迟后才执行赋值，这种赋值语句是不可综合的，在需要综合的模块设计中不可使用这种风格的代码。

阻塞赋值的执行可以认为是只有一个步骤的操作：

计算 RHS 并更新 LHS，此时不能允许有来自任何其他 Verilog 语句的干扰。所谓阻塞的概念是指在同一个 always 块中，其后面的赋值语句从概念上（即使不设定延迟）是在前一句赋值语句结束后再开始赋值的。

如果在一个过程块中阻塞赋值的 RHS 变量正好是另一个过程块中阻塞赋值的 LHS 变量，这两个过程块又用同一个时钟沿触发，这时阻塞赋值操作会出现问题，即如果阻塞赋值的次序安排不好，就会出现竞争。若这两个阻塞赋值操作用同一个时钟沿触发，则执行的次序是无法确定的。下面的例子可以说明这个问题：

[例 1]. 用阻塞赋值的反馈振荡器

```
module fbosc1 (y1, y2, clk, rst);
  output y1, y2;
  input  clk, rst;
  reg    y1, y2;

  always @(posedge clk or posedge rst)
    if (rst) y1 = 0;  // reset
    else     y1 = y2;

  always @(posedge clk or posedge rst)
    if (rst) y2 = 1;  // preset
    else     y2 = y1;
endmodule
```

按照 IEEE Verilog 的标准，上例中两个 always 块是并行执行的，与前后次序无关。如果前一个 always 块的复位信号先到 0 时刻，则 y1 和 y2 都会取 1，而如果后一个 always 块的复位信号先到 0 时刻，则 y1 和 y2 都会取 0。这清楚地说明这个 Verilog 模块是不稳定的会产生冒险和竞争的情况。

非阻塞赋值

非阻塞赋值操作符用小于等于号（即 <= ）表示。为什么称这种赋值为非阻塞赋值？这是因为在赋值操作时刻开始时计算非阻塞赋值符的 RHS 表达式，赋值操作时刻结束时更新 LHS。在计算非阻塞赋值的 RHS 表达式和更新 LHS 期间，其他的 Verilog 语句，包括其他的 Verilog 非阻塞赋值语句都能同时计算 RHS 表达式和更新 LHS。非阻塞赋值允许其他的 Verilog 语句同时进行操作。非阻塞赋值的操作可以看作为两个步骤的过程：

1) 在赋值时刻开始时，计算非阻塞赋值 RHS 表达式。
2) 在赋值时刻结束时，更新非阻塞赋值 LHS 表达式。

非阻塞赋值操作只能用于对寄存器类型变量进行赋值，因此只能用在"initial"块和"always"块等过程块中。非阻塞赋值不允许用于连续赋值。下面的例子可以说明这个问题：

[例 2]. 用非阻塞赋值的反馈振荡器

```
module fbosc2 (y1, y2, clk, rst);
  output y1, y2;
  input  clk, rst;
  reg    y1, y2;

  always @(posedge clk or posedge rst)
    if (rst) y1 <= 0;  // reset
```

```
    else     y1 <= y2;

  always @(posedge clk or posedge rst)
    if (rst) y2 <= 1;  // preset
    else     y2 <= y1;
endmodule
```

同样，按照 IEEE Verilog 的标准，上例中两个 always 块是并行执行的，与前后次序无关。无论哪一个 always 块的复位信号先到， 两个 always 块中的非阻塞赋值都在赋值开始时刻计算 RHS 表达式,，而在结束时刻才更新 LHS 表达式。所以这两个 always 块在复位信号到来后,在 always 块结束时 y1 为 0 而 y2 为 1 是确定的。从用户的角度看这两个非阻塞赋值正好是并行执行的。

Verilog 模块编程要点：

下面我们还将对阻塞和非阻塞赋值做进一步解释并将举更多的例子来说明这个问题。在此之前，掌握可综合风格的 Verilog 模块编程的八个原则会有很大的帮助。在编写时牢记这八个要点可以为绝大多数的 Verilog 用户解决在综合后仿真中出现的 90-100% 的冒险竞争问题。

1) **时序电路建模时，用非阻塞赋值。**
2) **锁存器电路建模时，用非阻塞赋值。**
3) **用 always 块建立组合逻辑模型时，用阻塞赋值。**
4) **在同一个 always 块中建立时序和组合逻辑电路时，用非阻塞赋值。**
5) **在同一个 always 块中不要既用非阻塞赋值又用阻塞赋值。**
6) **不要在一个以上的 always 块中为同一个变量赋值。**
7) **用$strobe 系统任务来显示用非阻塞赋值的变量值**
8) **在赋值时不要使用 #0 延迟**

我们在后面还要对为什么要记住这些要点再做进一步的解释。Verilog 的新用户在彻底搞明白这两种赋值功能差别之前,一定要牢记这几条要点。照着要点来编写 Verilog 模块程序，就可省去很多麻烦。

Verilog 的层次化事件队列

详细地了解 Verilog 的层次化事件队列有助于我们理解 Verilog 的阻塞和非阻塞赋值的功能。所谓层次化事件队列指的是用于调度仿真事件的不同的 Verilog 事件队列。在 IEEE Verilog 标准中,层次化事件队列被看作是一个概念模型。设计仿真工具的厂商如何来实现事件队列,由于关系到仿真器的效率，被视为技术诀窍,不能公开发表。本节也不作详细介绍。

在 IEEE 1364-1995 Verilog 标准的 5.3 节中定义了: 层次化事件队列在逻辑上分为用于当前仿真时间的 4 个不同的队列, 和用于下一段仿真时间的若干个附加队列。

1) 动态事件队列（下列事件执行的次序可以随意安排）
   - 阻塞赋值
   - 计算非阻塞赋值语句右边的表达式
   - 连续赋值
   - 执行$display 命令
   - 计算原语的输入和输出的变化

2) 停止运行的事件队列
   - #0 延时阻塞赋值

3) 非阻塞事件队列
   - 更新非阻塞赋值语句 LHS（左边变量）的值

4) 监控事件队列
   - 执行$monitor 命令

- 执行$strobe 命令

5) 其他指定的PLI命令队列
   - (其他 PLI 命令)

以上五个队列就是 Verilog 的“层次化事件队列”

大多数 Verilog 事件是由动态事件队列调度的，这些事件包括阻塞赋值、连续赋值、$display 命令、实例和原语的输入变化以及他们的输出更新、非阻塞赋值语句 RHS 的计算等。而非阻塞赋值语句 LHS 的更新却不由动态事件队列调度。

在 IEEE 标准允许的范围内被加入到这些队列中的事件只能从动态事件队列中清除。而排列在其他队列中的事件要等到被“激活”后，即被排入动态事件队列中后，才能真正开始等待执行。IEEE 1364-1995 Verilog 标准的 5.4 节介绍了一个描述其他事件队列何时被“激活”的算法。

在当前仿真时间中，另外两个比较常用的队列是非阻塞赋值更新事件队列和监控事件队列。细节见后。

非阻塞赋值 LHS 变量的更新是按排在非阻塞赋值更新事件队列中。而 RHS 表达式的计算是在某个仿真时刻随机地开始的,与上述其他动态事件是一样的。

$strobe 和 $monitor 显示命令是排列在监控事件队列中。在仿真的每一步结束时刻，当该仿真步骤内所有的赋值都完成以后，$strobe 和 $monitor 显示出所有要求显示的变量值的变化。

在 Verilog 标准 5.3 节中描述的第四个事件队列是停止运行事件队列， 所有#0 延时的赋值都排列在该队列中。采用#0 延时赋值是因为有些对 Verilog 理解不够深入的设计人员希望在两个不同的程序块中给同一个变量赋值，他们企图在同一个仿真时刻，通过稍加延时的赋值来消除 Verilog 可能产生的竞争冒险。这样做实际上会产生问题。因为给 Verilog 模型附加完全不必要的#0 延时赋值，使得定时事件的分析变得很复杂。我们认为采用#0 延时赋值根本没有必要，完全可用其他的方式来代替，因此不推荐使用。

在下面的一些例子中，常常用上面介绍的层次化事件队列来解释 Verilog 代码的行为。时件队列的概念也常常用来说明为什么要坚持上面提到的 8 项原则。

自触发 always 块

一般而言，Verilog 的 always 块不能触发自己，见下面的例子：

[例 3] 使用阻塞赋值的非自触发振荡器

```
module osc1 (clk);
     output clk;
     reg    clk;

       initial #10 clk = 0;
       always @(clk) #10 clk = ~clk;

endmodule
```

上例描述的时钟振荡器使用了阻塞赋值。阻塞赋值时，计算 RHS 表达式并更新 LHS 的值，此时不允许其他语句的干扰。阻塞赋值必须在@(clk)边沿触发到来时刻之前完成。当触发事件到来时，阻塞赋值已经完成了，因此没有来自 always 块内部的触发事件来触发@(clk)，是一个非自触发振荡器。

而例 4 中的振荡器使用的是非阻塞赋值，它是一个自触发振荡器。

[例 4] 采用非阻塞赋值的自触发振荡器

```
module osc2 (clk);
      output clk;
      reg    clk;

      initial #10 clk = 0;
      always @(clk) #10 clk <= ~clk;

endmodule
```

@(clk)的第一次触发之后，非阻塞赋值的 RHS 表达式便计算出来，把值赋给 LHS 的事件被安排在更新事件队列中。在非阻塞赋值更新事件队列被激活之前，又遇到了@(clk)触发语句，并且 always 块再次对 clk 的值变化产生反应。当非阻塞 LHS 的值在同一时刻被更新时，@(clk)再一次触发。该例是自触发式，在编写仿真测试模块时不推荐使用这种写法的时钟信号源。

移位寄存器模型

下图表示是一个简单的移位寄存器方框图。

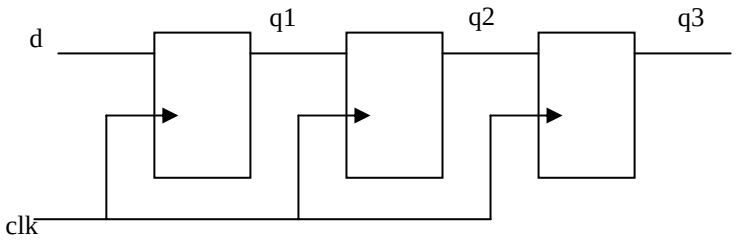


图 2 移位寄存器电路

从例 5 至例 8 介绍了四种用阻塞赋值实现图 2 移位寄存器电路的方式，有些是不正确。

[例 5]　不正确地使用的阻塞赋值来描述移位寄存器。(方式 #1)

```
module pipeb1 (q3, d, clk);
      output [7:0] q3;
      input  [7:0] d;
      input        clk;
      reg    [7:0] q3, q2, q1;

      always @(posedge clk)
        begin
           q1 = d;
           q2 = q1;
           q3 = q2;
       end
    endmodule
```

在上面的模块中，按顺序进行的阻塞赋值将使得在下一个时钟上升沿时刻，所有的寄存器输出值都等于输入值 d。在每个时钟上升沿，输入值 d 将无延时地直接输出到 q3。

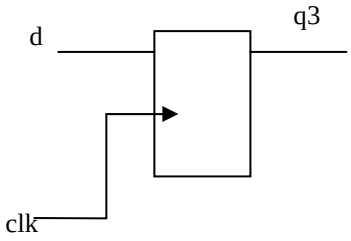


图 3　实际综合的结果

显然，上面的模块实际上被综合成只有一个寄存器的电路（见图 3），这并不是当初想要设计的移位寄存器电路。

[例 6] 用阻塞赋值来描述移位寄存器也是可行的，但这种风格并不好。（方式 #2 ）

```
module pipeb2 (q3, d, clk);
      output [7:0] q3;
      input  [7:0] d;
      input        clk;
      reg    [7:0] q3, q2, q1;

      always @(posedge clk)
        begin
          q3 = q2;
          q2 = q1;
          q1 = d;
        end
    endmodule
```

在上面[例 6]的模块中，阻塞赋值的次序是经过仔细安排的，以使仿真的结果与移位寄存器相一致。虽然该模块可被综合成图 2 所示的移位寄存器，但我们不建议使用这种风格的模块来描述时序逻辑。

[例 7] 不好的用阻塞赋值来描述移位时序逻辑的风格（方式 #3）

```
 module pipeb3 (q3, d, clk);
      output [7:0] q3;
      input  [7:0] d;
      input        clk;
      reg    [7:0] q3, q2, q1;

      always @(posedge clk) q1 = d;
      always @(posedge clk) q2 = q1;
      always @(posedge clk) q3 = q2;
    endmodule
```

在[例 7]中，阻塞赋值分别被放在不同的 always 块里。仿真时，这些块的先后顺序是随机的，因此可能会出现错误的结果。这是 Verilog 中的竞争冒险。按不同的顺序执行这些块将导致不同的结果。但是，这些代码的综合结果却是正确的流水线寄存器。也就是说，前仿真和后仿真的结果可能会不一致。

[例 8] 不好的用阻塞赋值来描述移位时序逻辑的风格（方式 #4）

```
    module pipeb4 (q3, d, clk);
      output [7:0] q3;
      input  [7:0] d;
      input        clk;
      reg    [7:0] q3, q2, q1;
```

```
  always @(posedge clk) q2 = q1;
  always @(posedge clk) q3 = q2;
  always @(posedge clk) q1 = d;
endmodule
```

若在[例 8]中仅把 always 块的次序的作些变动，也可以被综合成正确的移位寄存器逻辑，但仿真结果可能不正确。

如果用非阻塞赋值语句改写以上这四个阻塞赋值的例子，每一个例子都可以正确仿真，并且综合为设计者期望的移位寄存器逻辑。

[例 9] 正确的用非阻塞赋值来描述时序逻辑的设计风格 #1

```
module pipen1 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q1 <= d;
    q2 <= q1;
    q3 <= q2;
  end
endmodule
```

[例 10] 正确的用非阻塞赋值来描述时序逻辑的设计风格 #2

```
module pipen2 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk)
   begin
      q3 <= q2;
      q2 <= q1;
      q1 <= d;
   end
endmodule
```

[例 11] 正确的用非阻塞赋值来描述时序逻辑的设计风格 #3

```
module pipen3 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) q1 <= d;
  always @(posedge clk) q2 <= q1;
```

```
    always @(posedge clk) q3 <= q2;
  endmodule
```

[例 12] 正确的用非阻塞赋值来描述时序逻辑的设计风格 #4

```
module pipen4 (q3, d, clk);
    output [7:0] q3;
    input  [7:0] d;
    input        clk;
    reg    [7:0] q3, q2, q1;

    always @(posedge clk) q2 <= q1;
    always @(posedge clk) q3 <= q2;
    always @(posedge clk) q1 <= d;
  endmodule
```

以上移位寄存器时序逻辑电路设计的例子表明：

- 四种阻塞赋值设计方式中有一种可以保证仿真正确
- 四种阻塞赋值设计方式中有三种可以保证综合正确
- 四种非阻塞赋值设计方式全部可以保证仿真正确
- 四种非阻塞赋值设计方式全部可以保证综合正确

虽然在一个 always 块中正确的安排赋值顺序,用阻塞赋值也可以实现移位寄存器时序流水线逻辑。但是，用非阻塞赋值实现同一时序逻辑要相对简单，而且，非阻塞赋值可以保证仿真和综合的结果都是一致和正确的。**因此我们建议大家在编写 Verilog 时序逻辑时要用非阻塞赋值的方式。**

阻塞赋值及一些简单的例子

许多关于 Verilog 和 Verilog 仿真的书籍都有一些使用阻塞赋值而且成功的简单例子。例 13 就是一个在许多书上都出现过的关于触发器的例子。

[例 13]

```
module dffb (q, d, clk, rst);
     output q;
     input  d, clk, rst;
     reg    q;

    always @(posedge clk)
      if (rst) q = 1'b0;
      else     q = d;
  endmodule
```

虽然可行也很简单，但我们不建议这种用阻塞赋值来描述 D 触发器模型的风格。

如果要把所有的模块写到一个 always 块里，是可以采用阻塞赋值得到正确的建模、仿真并综合成期望的逻辑。但是，这种想法将导致使用阻塞赋值的习惯，而在较为复杂的多个 always 块的情况下可能会导致竞争冒险。

[例 14] 使用非阻塞赋值来描述 D 触发器是建议使用的风格

```
  module dffx (q, d, clk, rst);
    output q;
    input  d, clk, rst;
    reg    q;

    always @(posedge clk)
```

```
    if (rst) q  <= 1'b0;
    else     q  <= d;
endmodule
```

养成在描述时序逻辑的多个 always 块（甚至在单个 always 块）中使用非阻塞赋值的习惯比较好，见例 14 所示。

现在来看一个稍复杂的时序逻辑——线性反馈移位寄存器或 LFSR。

时序反馈移位寄存器建模

线性反馈移位寄存器（Linear Feedback Shift-Register 简称 LFSR）是带反馈回路的时序逻辑。反馈回路给习惯于用顺序阻塞赋值描述时序逻辑的设计人员带来了麻烦。见 15 所示。

[例 15] 用阻塞赋值实现的线性反馈移位寄存器，实际上并不具有 LFSR 的功能

```
module lfsrb1 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;
  wire   n1;

  assign n1 = q1 ^ q3;

  always @(posedge clk or negedge pre_n)
    if (!pre_n)
       begin
          q3 = 1'b1;
          q2 = 1'b1;
          q1 = 1'b1;
       end
    else
        begin
         q3 = q2;
         q2 = n1;
         q1 = q3;
        end
endmodule
```

除非使用中间暂存变量，否则用例 15 所示的赋值是不可能实现反馈逻辑的。

有的人可能会想到将这些赋值语句组成单行等式（如例 16 所示），来避免使用中间变量。如果逻辑再复杂一些，单行等式是难以编写和调试的。这种方法不推荐使用。

[例 16] 用阻塞赋值描述的线性反馈移位寄存器，其功能正确，但模型的含义较难理解。

```
module lfsrb2 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) {q3,q2,q1} = 3'b111;
    else        {q3,q2,q1} = {q2,(q1^q3),q3};
endmodule
```

如果将例 15 和例 16 中的阻塞赋值用非阻塞赋值代替，如例 17 和例 18 所示，仿真结果都和 LFSR 的功能相一致。

[例 17] 用非阻塞语句描述的 LFSR，可综合其功能正确。

```
module lfsrn1 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;
  wire   n1;

  assign n1 = q1 ^ q3;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) begin
      q3 <= 1'b1;
      q2 <= 1'b1;
      q1 <= 1'b1;
    end
    else begin
      q3 <= q2;
      q2 <= n1;
      q1 <= q3;
    end
endmodule
```

[例 18] 用非阻塞语句描述的 LFSR，可综合其功能正确。

```
  module lfsrn2 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) {q3,q2,q1} <= 3'b111;
    else        {q3,q2,q1} <= {q2,(q1^q3),q3};
endmodule
```

从上面介绍的移位寄存器的例子以及 LFSR 的例子，建议使用非阻塞赋值实现时序逻辑。而用非阻塞赋值语句实现锁存器也是最为安全的。

原则 1 ：时序电路建模时，用非阻塞赋值。
原则 2 ：锁存器电路建模时，用非阻塞赋值。

组合逻辑建模时应使用阻塞赋值：
在 Verilog 中可以用多种方法来描述组合逻辑，但是当用 always 块来描述组合逻辑时，应该用阻塞赋值。

如果 always 块中只有一条赋值语句，使用阻塞赋值或非阻塞赋值语句都可以，但是为了养成良好的编程习惯，应该尽量使用阻塞赋值语句来描述组合逻辑。

有些设计人员提倡非阻塞赋值语句不仅可以用于时序逻辑，也可以用于组合逻辑的描述。对于简单的组合 alwasys 块是可以这样的，但是当 always 块中有多个赋值语句时，如例 19 所示的四输入与或门逻辑，使用没有延时的非阻塞赋值可能导致仿真结果不正确。有时需要在 always 块的入口附加敏感事件参数，

才能使仿真正确，因而从仿真的时间效率角度看也不合算。

[例 19] 使用非阻塞赋值语句来描述组合逻辑——不建议使用这种风格。

```
module ao4 (y, a, b, c, d);
  output y;
  input  a, b, c, d;
  reg    y, tmp1, tmp2;

  always @(a or b or c or d)
      begin
         tmp1 <= a & b;
         tmp2 <= c & d;
         y    <= tmp1 | tmp2;
      end
endmodule
```

例 19 中，输出 y 的值由三个时序语句计算得到。由于非阻塞赋值语句在 LHS 更新前，计算 RHS 的值，因此 tmp1 和 tmp2 仍是应进入该 always 块时的值，而不是在该步仿真结束时将更新的数值。输出 y 反映的是刚进入 always 块时的 tmp1 和 tmp2 的值，而不是在 always 块中经计算后得到的值。

[例 20]使用非阻塞赋值来描述多层组合逻辑，虽可行，但效率不高。

```
module ao5 (y, a, b, c, d);
  output y;
  input  a, b, c, d;
  reg    y, tmp1, tmp2;5

  always @(a or b or c or d or tmp1 or tmp2)
     begin
         tmp1 <= a & b;
         tmp2 <= c & d;
            y <= tmp1 | tmp2;
  end
endmodule
```

例 20 和例 19 的唯一区别在于，tmp1 和 temp2 加入了敏感列表中。如前所描述，当非阻塞赋值的 LHS 数值更新时，always 块将自触发并用最新计算的 tmp1 和 tmp2 的值计算更新输出 y 的值。将 tmp1 和 tmp2 加入到敏感列表中后，现在输出 y 的值是正确的。但是，一个 always 块中有多次参数传递降低了仿真器的性能，只有在没有其他合理方法的情况下才考虑这样做。

只需要在 always 块中使用阻塞赋值语句就可以实现组合逻辑，这样做既简单仿真又快是好的 Verilog 代码风格，建议大家使用。

[例 21] 使用阻塞赋值实现组合逻辑是推荐使用的编码风格。

```
module ao2 (y, a, b, c, d);
  output y;
  input  a, b, c, d;
  reg    y, tmp1, tmp2;

  always @(a or b or c or d) begin
    tmp1 = a & b;
    tmp2 = c & d;
    y    = tmp1 | tmp2;
```

```
  end
endmodule
```

例 21 和例 19 的唯一区别是，用阻塞赋值替代了非阻塞赋值。这样做可以保证仿真时经一次数据传递输出 y 的值便是正确的，仿真效率高。因此有以下原则；

原则 3 ：用 always 块描述组合逻辑时，应采用阻塞赋值语句。

时序和组合的混合逻辑——使用非阻塞赋值

有时候将简单的组合逻辑和时序逻辑写在一起很方便。当把组合逻辑和时序逻辑写到一个 always 块中时，应遵从时序逻辑建模的原则，使用非阻塞赋值，如例 22 所示。

[例 22] 在一个 always 块中同时实现组合逻辑和时序逻辑

```
module nbex2 (q, a, b, clk, rst_n);
  output q;
  input  clk, rst_n;
  input  a, b;
  reg    q;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0; // 时序逻辑
    else        q <= a ^ b;// 异或，为组合逻辑
endmodule
```

用两个 always 块实现以上逻辑也是可以的，一个 always 块是采用阻塞赋值的纯组合部分，另一个是采用非阻塞赋值的纯时序部分。见例 23。

[例 23] 将组合和时序逻辑分别写在两个 always 块中

```
module nbex1 (q, a, b, clk, rst_n);
  output q;
  input  clk, rst_n;
  input  a, b;
  reg    q, y;

  always @(a or b)
    y = a ^ b;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= y;
endmodule
```

原则 4：在同一个 always 块中描述时序和组合逻辑混合电路时，用非阻塞赋值。

其他将阻塞和非阻塞混合使用的原则

Verilog 语法并没有禁止将阻塞和非阻塞赋值自由地组合在一个 always 块里。虽然 Verilog 语法是允许这种写法的，但我们不建议在可综合模块的编写中采用这种风格。

[例24] 在always块中同时使用阻塞和非阻塞赋值的例子。
（应尽量避免使用这种风格的代码，在可综合模块中应严禁使用）

```
module ba_nba2 (q, a, b, clk, rst_n);
  output q;
  input  a, b, rst_n;
  input  clk;
  reg    q;

  always @(posedge clk or negedge rst_n) begin: ff
    reg tmp;
    if (!rst_n) q <= 1'b0;
    else begin
      tmp = a & b;
      q <= tmp;
    end
  end
endmodule
```

例24可以得到正确的仿真和综合结果，因为阻塞赋值和非阻塞赋值操作的不是同一个变量。虽然这种方法是可行的，但并不建议使用。

[例25] 对同一变量既进行阻塞赋值,又进行非阻塞赋值会产生综合错误。

```
module ba_nba6 (q, a, b, clk, rst_n);
  output q;
  input  a, b, rst_n;
  input  clk;
  reg    q, tmp;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q = 1'b0;  // 对 q 进行阻塞赋值
    else begin
      tmp = a & b;
      q <= tmp;            // 对 q 进行非阻塞赋值
    end
endmodule
```

例25在仿真时结果通常是正确的，但是综合时会出错，因为对同一变量既进行阻塞赋值,又进行了非阻塞赋值。因此，必须将其改写才能成为可综合模型。

为了养成良好的编程习惯，建议：

原则5：不要在同一个always块中同时使用阻塞和非阻塞赋值。

对同一变量进行多次赋值

在一个以上always块中对同一个变量进行多次赋值可能会导致竞争冒险,即使使用非阻塞赋值也可能产生竞争冒险。在例26中，两个always块都对输出q进行赋值。由于两个always块执行的顺序是随机的，所以仿真时会产生竞争冒险。

[例25] 使用非阻塞赋值语句，由于两个always块对同一变量q赋值
产生竞争冒险的程序：

```
module badcode1 (q, d1, d2, clk, rst_n);
```

```
  output q;
  input  d1, d2, clk, rst_n;
  reg    q;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d1;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d2;
endmodule
```

当综合工具（如 Synopsys）读到[例 25]的代码时，将产生以下警告信息：

```
Warning: In design 'badcode1', there is 1 multiple-driver
         net with unknown wired-logic type.
```

如果忽略这个警告，继续编译例 26，将产生两个触发器输出到一个两输入与门。其综合级前仿真与综合后仿真的结果不完全一致。

原则 6：严禁在多个 always 块中对同一个变量赋值。

常见的对于非阻塞赋值的误解

- 非阻塞赋值和$display

  误解 1： “使用$display 命令不能用来显示非阻塞语句的赋值”

  事实是： 非阻塞语句的赋值在所有的$display 命令执行以后才更新数值

  [例]

```
module display_cmds;
  reg a;

  initial $monitor("\$monitor: a = %b", a);

  initial
    begin
        $strobe  ("\$strobe : a = %b", a);
         a = 0;
        a <= 1;
        $display ("\$display: a = %b", a);
        #1 $finish;
    end
endmodule
```

下面是上面模块的仿真结果说明$display 命令的执行是安排在活动事件队列中，但排在非阻塞赋值数据更新事件之前。

```
$display: a = 0
$monitor: a = 1
$strobe : a = 1
```

- #0 延时赋值

  误解 2： “＃0 延时把赋值强制到仿真时间步的末尾”

事实是： #0 延时将赋值事件强制加入停止运行事件队列中。

[例]

```
module nb_schedule1;
  reg a, b;

  initial
    begin
       a = 0;
       b = 1;
      a <= b;
      b <= a;

       $monitor ("%0dns: \$monitor: a=%b  b=%b", $stime, a, b);
       $display ("%0dns: \$display: a=%b  b=%b", $stime, a, b);
       $strobe  ("%0dns: \$strobe : a=%b  b=%b\n", $stime, a, b);
    #0 $display ("%0dns: #0      : a=%b  b=%b", $stime, a, b);

    #1 $monitor ("%0dns: \$monitor: a=%b  b=%b", $stime, a, b);
       $display ("%0dns: \$display: a=%b  b=%b", $stime, a, b);
       $strobe  ("%0dns: \$strobe : a=%b  b=%b\n", $stime, a, b);
       $display ("%0dns: #0      : a=%b  b=%b", $stime, a, b);

    #1 $finish;
  end
endmodule
```

下面是上面模块的仿真结果说明#0 延时命令在非阻塞赋值事件发生前，在停止运行事件队列中执行。

```
0ns: $display: a=0  b=1
0ns: #0      : a=0  b=1
0ns: $monitor: a=1  b=0
0ns: $strobe : a=1  b=0

1ns: $display: a=1  b=0
1ns: #0      : a=1  b=0
1ns: $monitor: a=1  b=0
1ns: $strobe : a=1  b=0
```

原则 7：用$strobe 系统任务来显示用非阻塞赋值的变量值

- 对同一变量进行多次非阻塞赋值

误解 3:“在 Verilog 语法标准中未定义可在同一个 always 块中对某同一变量进行多次非阻塞赋值”。
事实是： Verilog 标准定义了在同一个 always 块中可对某同一变量进行多次非阻塞赋值但多次赋值中，只有最后一次赋值对该变量起作用。

引用 IEEE 1364-1995 Verilog 标准【2】，第 47 页，5.4.1 节关于决定论的内容如下：

“非阻塞赋值按照语句的顺序执行,请看下例：

```
initial begin
    a <= 0;
    a <= 1;
end
```

执行该模块时，有两个非阻塞赋值更新事件加入到非阻塞赋值更新队列。以前的规则要求将非阻塞赋值更新事件按照它们在源文件的顺序加入队列，这便要求按照事件在源文件中的顺序，将事件从队列中取出并执行。因此，在仿真第一步结束的时刻，变量 a 被设置为 0，然后为 1。”
结论：最后一个非阻塞赋值决定了变量的值。

总结：
本节中所有的原则归纳如下：

- **原则 1：时序电路建模时，用非阻塞赋值。**
- **原则 2：锁存器电路建模时，用非阻塞赋值。**
- **原则 3：用 always 块写组合逻辑时，采用阻塞赋值。**
- **原则 4：在同一个 always 块中同时建立时序和组合逻辑电路时，用非阻塞赋值。**
- **原则 5：在同一个 always 块中不要同时使用非阻塞赋值和阻塞赋值。**
- **原则 6：不要在多个 always 块中为同一个变量赋值。**
- **原则 7：用$strobe 系统任务来显示用非阻塞赋值的变量值**
- **原则 8：在赋值时不要使用 #0 延迟**

结论：遵循以上原则，有助于正确的编写可综合硬件，并且可以消除 90－100％在仿真时可能产生的竞争冒险现象。

7.2.5. 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。

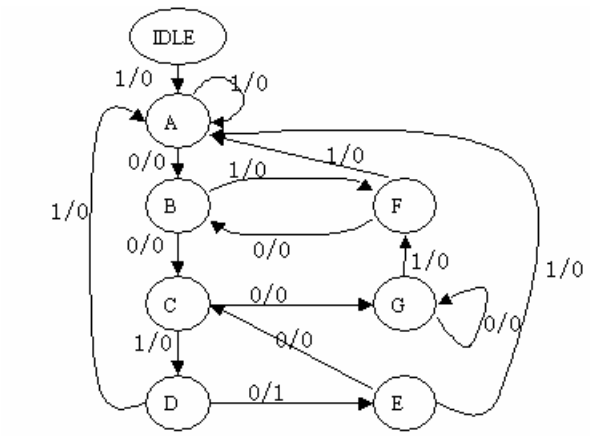
序列检测器的逻辑功能描述：

序列检测指的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“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比特序列“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：sequence.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;
```

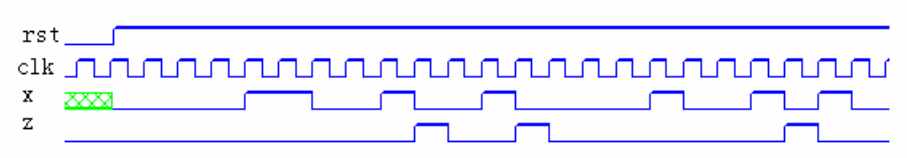
```
                                    end
                              else
                                    begin
                                          state<=A;
                                    end
                        E:      if(x==0)
                                    begin
                                          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：sequence.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 // t
```

其中、X码流的产生，我们采用了移位寄存器的方式，以方便更改测试数据。仿真结果如下图所示：



从波形中，我们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，sequence.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、综合和后仿真在PC WINDOWS NT 4.0操作系统及QuickLogic SPDE环境下通过。

[例2]EEPROM读写器件的设计

下面我们将介绍一个经过实际运行验证并可综合到各种FPGA和ASIC工艺的串行EEPROM读写器件的设计过程。列出了所有有关的Verilog HDL程序。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。

1. 二线制I^2C CMOS 串行EEPROM的简单介绍

二线制I^2C CMOS 串行EEPROM AT24C02/4/8/16 是一种采用CMOS 工艺制成的串行可用电擦除可编程只读存储器。串行EEPROM 一般具有两种写入方式，一种是字节写入方式，还有另一种页写入方式，允许在一个写周期内同时对一个字节到一页的若干字节进行编程写入，一页的大小取决于芯片内页寄存器的大小，不同公司的同一种型号存储器的内页寄存器可能是不一样的。为了程序的简单起见，在这里只编写串行 EEPROM 的一个字节的写入和读出方式的Verilog HDL 的行为模型代码，串行EEPROM读写器的Verilog HDL模型也只是字节读写方式的可综合模型，对于页写入和读出方式，读者可以参考有关书籍，改写串行EEPROM 的行为模型和串行EEPROM读写器的可综合模型。

2. I^2C （Inter Integrated Circuit）总线特征介绍

I^2C 双向二线制串行总线协议定义如下：

只有在总线处于“非忙”状态时，数据传输才能被初始化。在数据传输期间，只要时钟线为高电平，数据线都必须保持稳定，否则数据线上的任何变化都被当作“启动”或“停止”信号。图 1 是被定义的总线状态。

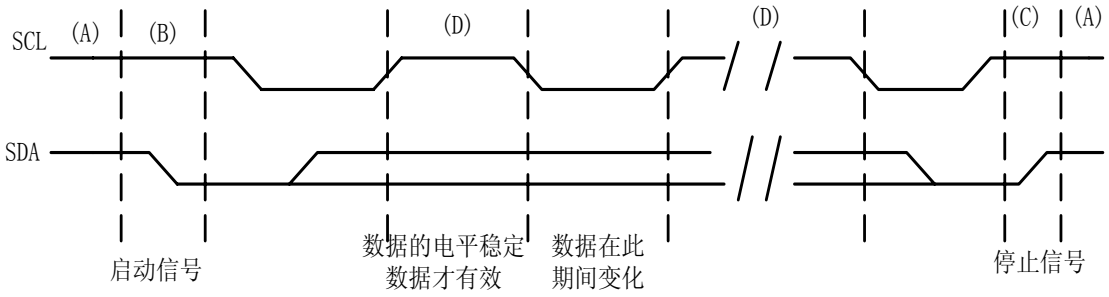


图**1**. I^2C 双向二线制串行总线特征

① 总线非忙状态（A 段）
数据线SDA 和 时钟线 SCL 都保持高电平。
② 启动数据传输（B 段）
当时钟线（SCL）为高电平状态时，数据线（SDA）由高电平变为低电平的下降沿被认为是“启动”信号。只有出现“启动”信号后，其它的命令才有效。
③ 停止数据传输（C 段）
当时钟线（SCL）为高电平状态时，数据线（SDA）由低电平变为高电平的上升沿被认为是“停止”信号。随着“停在”信号出现，所有的外部操作都结束。
④ 数据有效（D 段）
在出现“启动”信号以后，在时钟线（SCL）为高电平状态时数据线是稳定的，这时数据线的状态就要传送的数据。数据线（SDA）上的数据的改变必须在时钟线为低电平期间完成，每位数据占用一个时钟脉冲。每个数传输都是由“启动”信号开始，结束于“停止”信号。
⑤ 应答信号
每个正在接收数据的EEPROM 在接到一个字节的数据后，通常需要发出一个应答信号。而每个正在发送数据的EEPROM 在发出一个字节的数据后，通常需要接收一个应答信号。EEPROM 读写控制器必须产生一个与这个应答位相联系的额外的时钟脉冲。在EEPROM 的读操作中，EEPROM 读写控制器对EEPROM 完成的最后一个字节不产生应答位，但是应该给EEPROM 一个结束信号。

3. 二线制I^2C CMOS 串行EEPROM读写操作

1） EEPROM 的写操作（字节编程方式）

所谓EEPROM的写操作（字节编程方式）就是通过读写控制器把一个字节数据发送到EEPROM 中指定地址的存储单元。其过程如下：EEPROM 读写控制器发出“启动”信号后，紧跟着送4位I^2C总线器件特征编码1010 和3 位EEPROM 芯片地址/页地址XXX 以及写状态的R/W 位(=0)，到总线上。这一字节表示在接收到被寻址的EEPROM 产生的一个应答位后，读写控制器将跟着发送1个字节的EEPROM 存储单元地址和要写入的1个字节数据。EEPROM 在接收到存储单元地址后又一次产生应答位以后，读写控制器才发送数据字节，并把数据写入被寻址的存储单元。EEPROM 再一次发出应答信号，读写控制器收到此应答信号后，便产生“停止”信号。字节写入帧格式如图2所示：

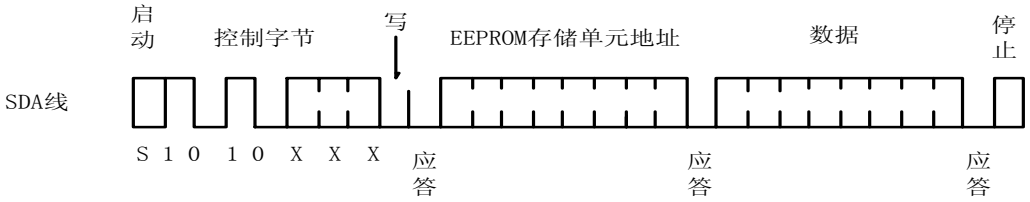


图**2**：24C02/4/8/16字节写入帧格式

2） 二线制I^2C CMOS 串行EEPROM 的读操作

所谓EEPROM的读操作即通过读写控制器读取 EEPROM 中指定地址的存储单元中的一个字节数据。串行EEPROM 的读操作分两步进行：读写器首先发送一个“启动”信号和控制字节(包括页面地址和写控制位)到EEPROM，再通过写操作设置EEPROM 存储单元地址（注意：虽然这是读操作，但需要先写入地址指针的值），在此期间EEPROM 会产生必要的应答位。接着读写器重新发送另一个“启动”信号和控制字节(包括页面地址和读控制位R/W = 1)，EEPROM收到后发出应答信号，然后，要寻址存储单元的数据就从SDA 线上输出。读操作有三种：读当前地址存储单元的数据、读指定地址存储单元的数据、读连续存储单元的数据。在这里只介绍读指定地址存储单元数据的操作。读指定地址存储单元数据的帧格式如图3：

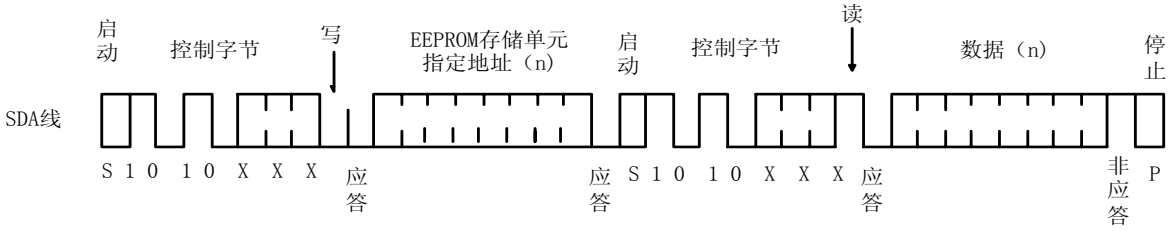


图3 ：24C02/4/8/16读指定地址存储单元的数据帧格式

4. EEPROM的Verilog HDL 程序

要设计一个串行EEPROM读写器件，不仅要编写EEPROM读写器件的可综合Verilog HDl的代码，而且要编写相应的测试代码以及EERPOM的行为模型。EEPROM的读写电路及其测试电路如图4。

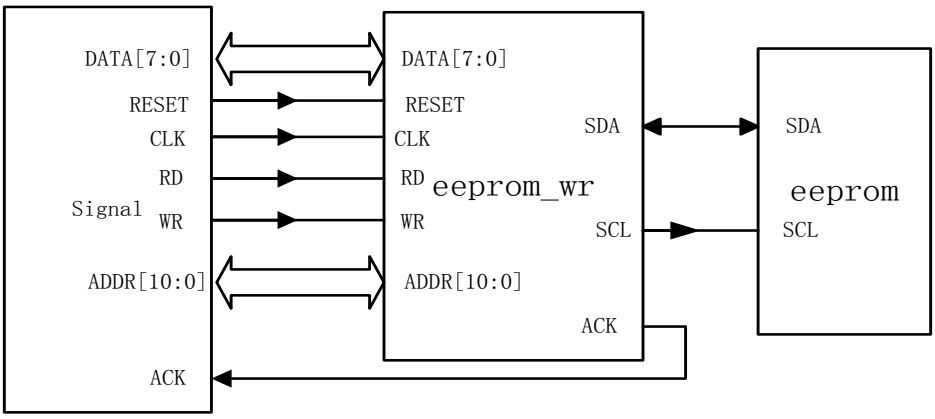


图4： EEPROM读写电路和它的测试电路

1) EEPROM的行为模型

为了设计这样一个电路我们首先要设计一个EEPROM的Verilog HDL模型,而设计这样一个模型我们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） 能完成一个字节数据读写的部分行为模型,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。因为这一程序是我们自己编写的有不完善之处敬请指正。

这里只对在操作中用到的信号线进行模拟，对于没有用到的信号线就略去了。对EEPROM用于基本总线操作的引脚SCL和SDA说明如下：SCL，串行时钟端，这个信号用于对输入和输出数据的同步，写入串行EEPROM的数据用其上升沿同步，输出数据用其下降沿同步；SDA,串行数据（/地址）输入/输出端。

```
`timescale 1ns/1ns
`define timeslice 100
module EEPROM(scl, sda);
input  scl;    //串行时钟线
inout  sda;    //串行数据线
reg out_flag;  //SDA数据输出的控制信号
reg[7:0]  memory[2047:0];
reg[10:0] address;
reg[7:0]  memory_buf;
reg[7:0]  sda_buf;   //SDA 数据输出寄存器
reg[7:0]  shift;     //SDA 数据输入寄存器
reg[7:0]  addr_byte; //EEPROM 存储单元地址寄存器
reg[7:0]  ctrl_byte; //控制字寄存器
reg[1:0]  State;     //状态寄存器
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
//-------------------------------------------------------------
assign sda = (out_flag == 1) ? sda_buf[7] : 1'bz;
//－－－－－－－－寄存器和存储器初始化－－－－－－－－－－－－－－－－－
initial
  begin
    addr_byte     = 0;
    ctrl_byte     = 0;
    out_flag      = 0;
    sda_buf       = 0;
    State         = 2'b00;
    memory_buf    = 0;
    address       = 0;
    shift         = 0;
    for(i=0;i<=2047;i=i+1)
      memory[i]=0;
  end
//------------ 启动信号 -----------------------------
always @ (negedge sda)
  if(scl == 1 )
    begin
      State = State + 1;
      if(State == 2'b11)
```

```
          disable write_to_eeprm;
      end
  //------------ 主状态机 -------------------------
  always @(posedge sda)
    if (scl == 1 )      //停止操作
      stop_W_R;
    else
      begin
       casex(State)
          2'b01:
                    begin
                      read_in;
                     if(ctrl_byte==w7||ctrl_byte==w6||ctrl_byte==w5
                        ||ctrl_byte==w4||ctrl_byte==w3||ctrl_byte==w2
                        ||ctrl_byte==w1||ctrl_byte==w0)
                  begin
                    State = 2'b10;
                    write_to_eeprm;  //写操作
                   end
                else
                  State = 2'b00;
              end

          2'b11:
                read_from_eeprm;      //读操作

       default:
                State=2'b00;

     endcase
    end
  //------------- 操作停止------------------------------
  task stop_W_R;
    begin
      State =2'b00;  //状态返回为初始状态
      addr_byte = 0;
      ctrl_byte = 0;
      out_flag  = 0;
      sda_buf   = 0;
    end
  endtask
  //------------- 读进控制字和存储单元地址 ------------------------
  task  read_in;
    begin
      shift_in(ctrl_byte);
      shift_in(addr_byte);
    end
  endtask
  //------------EEPROM 的写操作-------------------------------------
  task write_to_eeprm;
    begin
      shift_in(memory_buf);
```

```
        address          = {ctrl_byte[3:1],addr_byte};
        memory[address]  = memory_buf;
        $display("eeprm----memory[%0h]=%0h",address,memory[address]);
        State =2'b00;              //回到0状态
      end
    endtask
//-----------EEPROM 的读操作---------------------------------------
    task read_from_eeprm;
      begin
        shift_in(ctrl_byte);
        if(ctrl_byte==r7||ctrl_byte==r6||ctrl_byte==r5||ctrl_byte==r4
           ||ctrl_byte==r3||ctrl_byte==r2||ctrl_byte==r1||ctrl_byte==r0)
          begin
            address = {ctrl_byte[3:1],addr_byte};
            sda_buf = memory[address];
            shift_out;
            State= 2'b00;
          end
      end
    endtask
//-----SDA 数据线上的数据存入寄存器，数据在SCL的高电平有效-------------
    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)
           begin
             #`timeslice ;
              out_flag = 1;      //应答信号输出
              sda_buf  = 0;
           end
        @(negedge scl)
           #`timeslice out_flag  = 0;
      end
    endtask
    //---EEPROM 存储器中的数据通过SDA 数据线输出，数据在SCL 低电平时变化
    task shift_out;
      begin
        out_flag = 1;
        for(i=6;i>=0;i=i-1)
          begin
            @ (negedge scl);
            #`timeslice;
            sda_buf = sda_buf<<1;
          end
```

```
      @(negedge scl)  #`timeslice sda_buf[7] = 1;  //非应答信号输出
      @(negedge scl)  #`timeslice out_flag  = 0;
    end
  endtask
endmodule
```

2 ） EEPROM读写器的可综合的Verilog HDL模型

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；它还发送应答信号 (ACK)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，所以它不但要仿真正确无误，还需要可综合。

这个程序基本上由两部分组成：开关组合电路和控制时序电路，见图5。开关电路在控制时序电路的控制下按照设计的要求有节奏的打开或闭合，这样SDA可以按I^2C 数据总线的格式输出或输入，SDA和SCL一起完成EEPROM的读写操作。

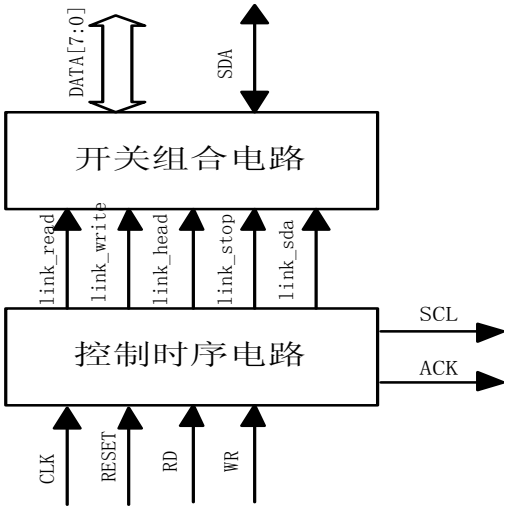


图5 ： EEPROM读写器的结构

电路最终用同步有限状态机（FSM）的设计方法实现。程序实则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。根据串行EEPROM的读写操作时序可知，用5个状态时钟可以完成写操作，用7个状态时钟可以完成读操作，由于读写操作的状态中有几个状态是一致的，用一个嵌套的状态机即可。状态转移如图6，程序由一个读写大任务和若干个较小的任务所组成，其状态机采用独热编码，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。

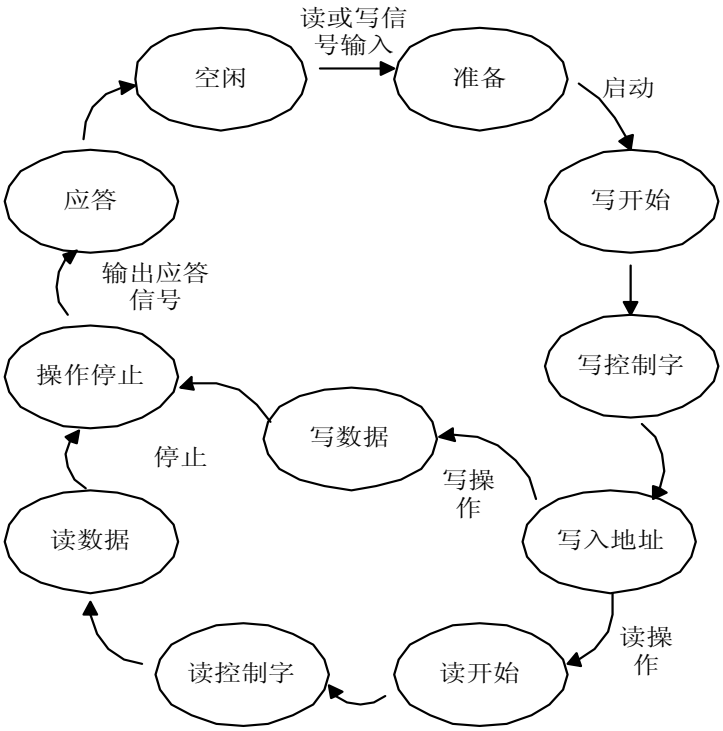


图6 ： 读写操作状态转移

```
`timescale 1ns/1ns
module EEPROM_WR(SDA,SCL,ACK,RESET,CLK,WR,RD,ADDR,DATA);
output SCL;                 //串行时钟线
output ACK;                 //读写一个周期的应答信号
input  RESET;              //复位信号
input  CLK;                //时钟信号输入
input  WR,RD;             //读写信号
input[10:0] ADDR;           //地址线
inout SDA;                  //串行数据线
inout[7:0] DATA;            //并行数据线
reg ACK;
reg SCL;
reg WF,RF;                 //读写操作标志
reg FF;                     //标志寄存器
reg [1:0] head_buf;          //启动信号寄存器
reg[1:0] stop_buf;           //停止信号寄存器
reg [7:0] sh8out_buf;         //EEPROM写寄存器
reg [8:0] sh8out_state;        //EEPROM 写状态寄存器
reg [9:0] sh8in_state;         //EEPROM 读状态寄存器
reg [2:0] head_state;          //启动状态寄存器
reg [2:0] stop_state;          //停止状态寄存器
reg [10:0] main_state;        //主状态寄存器
reg [7:0] data_from_rm;      //EEPROM读寄存器
reg link_sda;                //SDA 数据输入EEPROM开关
reg link_read;               //EEPROM读操作开关
reg link_head;               //启动信号开关
reg link_write;              //EEPROM写操作开关
reg link_stop;               //停止信号开关
wire sda1,sda2,sda3,sda4;
```

```
//--------------串行数据在开关的控制下有次序的输出或输入--------------------
assign sda1  = (link_head)     ?   head_buf[1]    :  1'b0;
assign sda2  = (link_write)    ?   sh8out_buf[7]  :  1'b0;
assign sda3  = (link_stop)     ?   stop_buf[1]    :  1'b0;
assign sda4  = (sda1 | sda2 | sda3);
assign SDA  = (link_sda)       ?  sda4            :  1'bz;
assign DATA = (link_read)      ?  data_from_rm    :  8'hzz;

//--------------------------------主状态机状态-----------------------------------------
parameter
                 Idle        = 11'b00000000001,
                  Ready      = 11'b00000000010,
               Write_start   = 11'b00000000100,
                Ctrl_write   = 11'b00000001000,
                 Addr_write  = 11'b00000010000,
               Data_write    = 11'b00000100000,
               Read_start    = 11'b00001000000,
               Ctrl_read     = 11'b00010000000,
                Data_read    = 11'b00100000000,
                Stop         = 11'b01000000000,
                 Ackn        = 11'b10000000000,
//-------------------------并行数据串行输出状态-----------------------------
               sh8out_bit7     = 9'b000000001,
               sh8out_bit6     = 9'b000000010,
               sh8out_bit5     = 9'b000000100,
               sh8out_bit4     = 9'b000001000,
               sh8out_bit3     = 9'b000010000,
               sh8out_bit2     = 9'b000100000,
               sh8out_bit1     = 9'b001000000,
               sh8out_bit0     = 9'b010000000,
               sh8out_end     = 9'b100000000;
//-------------------------串行数据并行输出状态-----------------------------
 parameter     sh8in_begin    = 10'b0000000001,
               sh8in_bit7     = 10'b0000000010,
               sh8in_bit6     = 10'b0000000100,
               sh8in_bit5     = 10'b0000001000,
               sh8in_bit4     = 10'b0000010000,
               sh8in_bit3     = 10'b0000100000,
               sh8in_bit2     = 10'b0001000000,
               sh8in_bit1     = 10'b0010000000,
               sh8in_bit0     = 10'b0100000000,
               sh8in_end      = 10'b1000000000,
//--------------------------------启动状态-----------------------------------
               head_begin   = 3'b001,
               head_bit     = 3'b010,
               head_end     = 3'b100,
//--------------------------------停止状态-----------------------------------
               stop_begin   = 3'b001,
               stop_bit     = 3'b010,
               stop_end     = 3'b100;

 parameter       YES             = 1,
```

```
                NO              = 0;
//-------------产生串行时钟，为输入时钟的二分频-------------------
always @(negedge CLK)
  if(RESET)
     SCL <= 0;
  else
    SCL <= ~SCL;
//----------------------------主状态程序---------------------------------
always @ (posedge CLK)
  if(RESET)
    begin
     link_read  <= NO;
     link_write <= NO;
     link_head  <= NO;
     link_stop  <= NO;
     link_sda   <= NO;
     ACK        <= 0;
     RF         <= 0;
     WF         <= 0;
     FF         <= 0;
     main_state <= Idle;
    end
  else
   begin
     casex(main_state)
           Idle:
                begin
                 link_read  <= NO;
                 link_write <= NO;
                 link_head  <= NO;
                 link_stop  <= NO;
                 link_sda   <= NO;
                 if(WR)
                   begin
                    WF <= 1;
                    main_state <= Ready ;
                   end
                 else if(RD)
                   begin
                    RF <= 1;
                    main_state <= Ready ;
                   end
                else
                  begin
                   WF <= 0;
                   RF <= 0;
                   main_state <= Idle;
                  end
                end
        Ready:
             begin
                   link_read        <= NO;
```

```
                link_write       <= NO;
                link_stop        <= NO;
                link_head        <= YES;
                link_sda         <= YES;
                head_buf[1:0]    <= 2'b10;
                stop_buf[1:0]    <= 2'b01;
                head_state       <= head_begin;
                FF               <= 0;
                ACK              <= 0;
                main_state       <= Write_start;
              end
Write_start:
             if(FF == 0)
                  shift_head;
                    else
                      begin
                        sh8out_buf[7:0]  <= {1'b1,1'b0,1'b1,1'b0,ADDR[10:8],1'b0};
                        link_head        <= NO;
                        link_write       <= YES;
                        FF               <= 0;
                        sh8out_state     <= sh8out_bit6;
                        main_state       <= Ctrl_write;
                      end
Ctrl_write:
            if(FF ==0)
                  shift8_out;
                else
                  begin
                       sh8out_state    <= sh8out_bit7;
                       sh8out_buf[7:0] <= ADDR[7:0];
                       FF              <= 0;
                       main_state      <= Addr_write;
                    end
Addr_write:
            if(FF == 0)
                      shift8_out;
                    else
                      begin
                        FF <= 0;
                        if(WF)
                          begin
                            sh8out_state    <= sh8out_bit7;
                            sh8out_buf[7:0] <= DATA;
                            main_state      <= Data_write;
                          end
                        if(RF)
                          begin
                            head_buf        <= 2'b10;
                            head_state       <= head_begin;
                            main_state      <= Read_start;
                          end
                      end
```

```
Data_write:
               if(FF == 0)
                        shift8_out;
                 else
                       begin
                        stop_state       <= stop_begin;
                        main_state       <= Stop;
                        link_write       <= NO;
                        FF               <= 0;
                       end

Read_start:
              if(FF == 0)
                     shift_head;
                  else
                       begin
                         sh8out_buf       <= {1'b1,1'b0,1'b1,1'b0,ADDR[10:8],1'b1};
                         link_head        <= NO;
                         link_sda         <= YES;
                         link_write       <= YES;
                         FF               <= 0;
                         sh8out_state     <= sh8out_bit6;
                         main_state       <= Ctrl_read;
                       end
Ctrl_read:
              if(FF == 0)
                    shift8_out;
            else
                       begin
                         link_sda         <= NO;
                         link_write       <= NO;
                         FF               <= 0;
                         sh8in_state      <= sh8in_begin;
                         main_state       <= Data_read;
                    end
Data_read:
             if(FF == 0)
                   shift8in;
              else
                      begin
                         link_stop        <= YES;
                         link_sda         <= YES;
                         stop_state       <= stop_bit;
                         FF               <= 0;
                         main_state       <= Stop;
                       end
Stop:
             if(FF == 0)
                       shift_stop;
                     else
                       begin
                      ACK        <= 1;
```

```
                    FF         <= 0;
                        main_state <= Ackn;
                end
  Ackn:
              begin
                        ACK         <= 0;
                        WF          <= 0;
                        RF          <= 0;
                        main_state  <= Idle;
                      end
  default:      main_state <= Idle;
  endcase
 end
//------------------------串行数据转换为并行数据任务----------------------------------
task shift8in;
  begin
   casex(sh8in_state)
    sh8in_begin:
              sh8in_state <= sh8in_bit7;
    sh8in_bit7: if(SCL)
                  begin
                       data_from_rm[7] <= SDA;
                       sh8in_state     <= sh8in_bit6;
                      end
                else
                  sh8in_state <= sh8in_bit7;
    sh8in_bit6: if(SCL)
                  begin
                    data_from_rm[6] <= SDA;
                    sh8in_state     <= sh8in_bit5;
                  end
                else
                  sh8in_state <= sh8in_bit6;
    sh8in_bit5: if(SCL)
                  begin
                    data_from_rm[5] <= SDA;
                    sh8in_state     <= sh8in_bit4;
                  end
                else
                  sh8in_state <= sh8in_bit5;
    sh8in_bit4: if(SCL)
                  begin
                    data_from_rm[4] <= SDA;
                    sh8in_state     <= sh8in_bit3;
                  end
                else
                    sh8in_state <= sh8in_bit4;
    sh8in_bit3: if(SCL)
                  begin
                    data_from_rm[3] <= SDA;
                    sh8in_state     <= sh8in_bit2;
                  end
```

```
            else
              sh8in_state <= sh8in_bit3;
   sh8in_bit2: if(SCL)
               begin
                 data_from_rm[2] <= SDA;
                 sh8in_state     <= sh8in_bit1;
               end
             else
               sh8in_state <= sh8in_bit2;
   sh8in_bit1: if(SCL)
               begin
                 data_from_rm[1] <= SDA;
                 sh8in_state     <= sh8in_bit0;
               end
            else
              sh8in_state <= sh8in_bit1;
   sh8in_bit0: if(SCL)
               begin
                 data_from_rm[0] <= SDA;
                 sh8in_state     <= sh8in_end;
               end
            else
               sh8in_state <= sh8in_bit0;
    sh8in_end: if(SCL)
               begin
                 link_read   <= YES;
                 FF          <=  1;
                 sh8in_state <= sh8in_bit7;
               end
            else
               sh8in_state  <= sh8in_end;
     default:      begin
              link_read    <= NO;
              sh8in_state  <= sh8in_bit7;
            end
endcase
end
endtask


//------------------------------ 并行数据转换为串行数据任务 ----------------------------
task shift8_out;
 begin
  casex(sh8out_state)
        sh8out_bit7:
                    if(!SCL)
                      begin
                            link_sda     <= YES;
                            link_write   <= YES;
                         sh8out_state <= sh8out_bit6;
                       end
                     else
```

```
                sh8out_state <= sh8out_bit7;
 sh8out_bit6:
           if(!SCL)
             begin
                  link_sda      <= YES;
                  link_write    <= YES;
                sh8out_state  <= sh8out_bit5;
                sh8out_buf    <= sh8out_buf<<1;
          end
           else
                sh8out_state <= sh8out_bit6;
 sh8out_bit5:
           if(!SCL)
             begin
                 sh8out_state <= sh8out_bit4;
                 sh8out_buf   <= sh8out_buf<<1;
             end
            else
              sh8out_state <= sh8out_bit5;
 sh8out_bit4:
           if(!SCL)
             begin
                 sh8out_state <= sh8out_bit3;
                 sh8out_buf   <= sh8out_buf<<1;
             end
           else
                 sh8out_state <= sh8out_bit4;
 sh8out_bit3:
           if(!SCL)
             begin
                 sh8out_state <= sh8out_bit2;
                 sh8out_buf   <= sh8out_buf<<1;
             end
           else
                 sh8out_state <= sh8out_bit3;
sh8out_bit2:
          if(!SCL)
            begin
                 sh8out_state <= sh8out_bit1;
                 sh8out_buf   <= sh8out_buf<<1;
            end
          else
                 sh8out_state <= sh8out_bit2;
sh8out_bit1:
          if(!SCL)
            begin
                 sh8out_state <= sh8out_bit0;
                 sh8out_buf   <= sh8out_buf<<1;
            end
          else
                 sh8out_state <= sh8out_bit1;
sh8out_bit0:
```

```
                if(!SCL)
                  begin
                       sh8out_state <= sh8out_end;
                       sh8out_buf   <= sh8out_buf<<1;
                  end
                 else
                       sh8out_state <= sh8out_bit0;
        sh8out_end:
                if(!SCL)
                  begin
                       link_sda         <= NO;
                       link_write       <= NO;
                     FF               <= 1;
                end
                 else
                       sh8out_state <= sh8out_end;
    endcase
  end
endtask
//--------------------------  输出启动信号任务  --------------------------------
task shift_head;
 begin
  casex(head_state)
        head_begin:
                if(!SCL)
                          begin
                              link_write   <= NO;
                           link_sda     <= YES;
                           link_head    <= YES;
                          head_state   <= head_bit;
                           end
                         else
                              head_state <= head_begin;
         head_bit:
                if(SCL)
                     begin
                           FF          <= 1;
                        head_buf     <= head_buf<<1;
                       head_state    <= head_end;
                          end
                else
                       head_state <= head_bit;
         head_end:
                if(!SCL)
                  begin
                           link_head    <= NO;
                            link_write   <= YES;
                     end
                  else
                           head_state <= head_end;
        endcase
 end
```

```
endtask
//--------------------------  输出停止信号任务  ------------------------------------
task shift_stop;
 begin
  casex(stop_state)
     stop_begin:  if(!SCL)
                          begin
                             link_sda      <= YES;
                              link_write   <= NO;
                             link_stop     <= YES;
                             stop_state    <= stop_bit;
                          end
                    else
                          stop_state <= stop_begin;
    stop_bit:    if(SCL)
                       begin
                             stop_buf   <= stop_buf<<1;
                            stop_state <= stop_end;
                       end
                   else
                           stop_state<= stop_bit;
    stop_end:   if(!SCL)
                  begin
                           link_head  <= NO;
                           link_stop  <= NO;
                           link_sda   <= NO;
                           FF         <= 1;
                         end
                       else
                              stop_state  <= stop_end;
    endcase
  end
endtask
endmodule
```

程序最终通过Synplify器的综合，并在Actel 3200DX 系列的FPGA上实现布局布线，通过布线后仿真 。

3 ） EEPROM的信号源模块和顶层模块

完成串行EEPROM读写器件的设计后，我们还需要做的重要一步是EEPROM读写器件的仿真。仿真可以分为前仿真和后仿真，前仿真是Verilog HDL的功能仿真，后仿真是Verilog HDL 代码经过综合、布局布线后的时序仿真。为此，我们还要编写了用于EEPROM读写器件的仿真测试的信号源程序。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。在这个程序中，我们为了保证串行EEPROM读写器件的正确性，可以进行完整的测试，写操作时输入的地址信号和数据信号的数据通过系统命令$readmemh 从addr.dat 和 data.dat 文件中取得，而在addr.dat 和data.dat文件中可以存放任意数据。读操作时从EEPROM 读出的数据存入文件eeprom.dat ,对比三个文件的数据就可以验证程序的正确性。$readmemh和$fopen等系统命令读者可以参考Verilog HDL的语法部分。最后我们把信号源、EEPROM和EEPROM读写器用顶层模块连接在一起。在下面的程序就是这个信号源的Verilog HDL模型和顶层模块。

信号源模型：

```
`timescale 1ns/1ns
`define timeslice 200
module Signal(RESET,CLK,RD,WR,ADDR,ACK,DATA);
output RESET;         //复位信号
output CLK;           //时钟信号
output RD,WR;         //读写信号
output[10:0] ADDR;     //11位地址信号
input ACK;            //读写周期的应答信号
inout[7:0] DATA;       //数据线
reg RESET;
reg CLK;
reg RD,WR;
reg W_R;             //低位：写操作；高位：读操作
reg[10:0] ADDR;
reg[7:0]  data_to_eeprom;
reg[10:0] addr_mem[0:255];
reg[7:0]  data_mem[0:255];
reg[7:0]  ROM[1:2048];
integer i,j;
integer OUTFILE;
assign DATA = (W_R) ?  8'bz : data_to_eeprom ;

//-----------------------------------时钟信号输入-----------------------------
always #(`timeslice/2)
 CLK = ~CLK;
//---------------------------------- 读写信号输入-----------------------------

initial
 begin
   RESET = 1;
   i   = 0;
   j   =0;
   W_R = 0;
   CLK = 0;
   RD  = 0;
   WR  = 0;
   #1000 ;
   RESET = 0;
    repeat(15)  //连续写15次数据
    begin
      #(5*`timeslice);
         WR = 1;
         #(`timeslice);
         WR = 0;
        @ (posedge ACK);
   end
  #(10*`timeslice);
  W_R = 1;   //开始读操作
  repeat(15)  //连续读15次数据
    begin
     #(5*`timeslice);
     RD = 1;
```

```
      #(`timeslice);
      RD = 0;
         @ (posedge ACK);
      end
  end
 //---------------------------------------写操作-----------------------------
 initial
 begin
   $display("writing-----writing-----writing-----writing");
   # (2*`timeslice);
   for(i=0;i<=15;i=i+1)
     begin
      ADDR = addr_mem[i];
      data_to_eeprom = data_mem[i];
      $fdisplay(OUTFILE,"@%0h  %0h",ADDR, data_to_eeprom);
      @(posedge ACK) ;
     end
end
 //---------------------------------------读操作-----------------------------
 initial
 @(posedge W_R)
  begin
   ADDR = addr_mem[0];
   $fclose(OUTFILE);
   $readmemh("./eeprom.dat",ROM);
   $display("Begin READING-----READING-----READING-----READING");
    for(j = 0; j <= 15; j = j+1)
    begin
       ADDR = addr_mem[j];
       @(posedge ACK);
       if(DATA == ROM[ADDR])
         $display("DATA %0h == ROM[%0h]---READ RIGHT",DATA,ADDR);
       else
         $display("DATA %0h != ROM[%0h]---READ WRONG",DATA,ADDR);
    end
 end

 initial
 begin
  OUTFILE = $fopen("./eeprom.dat");
  $readmemh("./addr.dat",addr_mem); //地址数据存入地址存储器
  $readmemh("./data.dat",data_mem); //写入EEPROM的数据存入数据存储器
 end

 endmodule
```

顶层模块：

```
`include “./Signal.v”
`include “./EEPROM.v”
`include “./EEPROM_WR.v”
`timescale 1ns/1ns
```

```
module Top;
wire RESET;
wire CLK;
wire RD,WR;
wire ACK;
wire[10:0] ADDR;
wire[7:0]  DATA;
wire SCL;
wire SDA;
Signal          signal(.RESET(RESET),.CLK(CLK),.RD(RD),
                       .WR(WR),.ADDR(ADDR),.ACK(ACK),.DATA(DATA));
EEPROM_WR   eeprom_wr(.RESET(RESET),.SDA(SDA),.SCL(SCL),.ACK(ACK),
                          .CLK(CLK),.WR(WR),.RD(RD),.ADDR(ADDR),.DATA(DATA));
EEPROM       eeprom(.sda(SDA),.scl(SCL));

endmodule
```

通过前后仿真可以验证程序的正确性。这里给出的是EEPROM读写时序的前仿真波形。后仿真波形除SCL和SDA与CLK有些延迟外,信号的逻辑关系与前仿真一致：

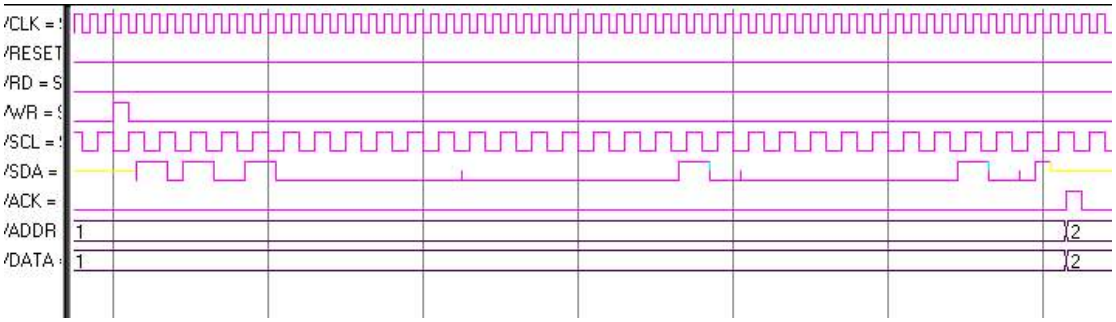


图7：EEPROM 的写时序

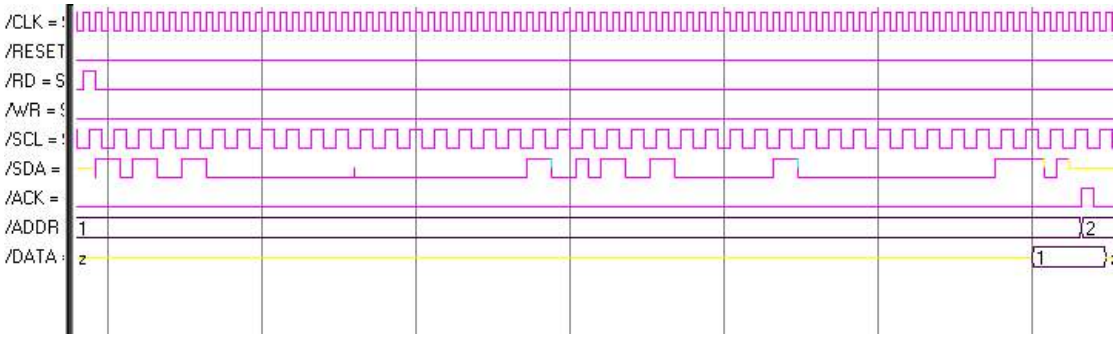


图8：EEPROM 的读时序

说明：以上编程、仿真、综合在PC WINDOWS NT 4.0操作系统、Synplify 、Actel Designer 、Altera Maxplus9.3及ModelSim Verilog环境下通过前后仿真,也在Unix Cadence Verilog-XL上通过前、后仿真（可综合到各种FPGA和ASIC工艺）。

思考题：

1) 什么是同步状态机？
2) 设计有限同步状态机的一般步骤是什么？
3) 为什么说把具体问题抽象成嵌套的状态机的思考方式可以处理极其复杂的逻辑关系？
4) 为什么要用同步状态机来产生数据流动的开关控制序列？
5) 什么是 HDL RTL级的描述方式？它与行为描述方式有什么不同？
6) 什么是综合？为什么要编写可综合模块？
7) 在设计中可综合模块和行为模块的作用分别是什么？
8) 可综合的Verilog HDL RTL级的描述方式的样板是什么？
9) 用RTL级描述方式的Verilog HDL模块是否都能综合？保证能综合的要点是什么？
10) 可综合的Verilog HDL RTL级模块的编写中用阻塞赋值和非阻塞赋值的原则是什么？
11) 保证可综合模块前后仿真一致性的关键是什么？
12) 改写7.2.5节中的例1：序列检测器的Verilog RTL级模块，使它能检测111000011序列，并进行前后仿真。
13) 读懂7.2.5节中的例2：EEPROM读写器的设计，并改写Verilog模块，使得它不只能进行随机读/写还能进行连续方式的读/写，并进行前后仿真。

第八章 可综合的VerilogHDL设计实例
---简化的RISC CPU设计简介---

前言：

在前面七章里我们已经学习了VerilogHDL的基本语法、简单组合逻辑和简单时序逻辑模块的编写、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，它是根据我们完成的实际工程项目设计为教学目的改写而来的,可以说已是真实的设计。

在这一章里，我们将通过一个经过简化的用于教学目的的 RISC_CPU 的设计过程,来说明这种新设计方法的潜力。这个模型实质上是第四章的RISC_CPU模型的改进。第四章中的RISC_CPU模型是一个仿真模型，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，因为他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，从中我们可以体会到这种设计方法的魅力。本章中的VerilogHDL程序都是我们自己为教学目的而编写的，全部程序在CADENCE公司的LWB (Logic Work Bench)环境下和 Mentor 公司的ModelSim 环境下用Verilog语言进行了仿真,通过了运行测试，并分别用Synergy和Synplify综合器针对不同的FPGA进行了综合。分别用Xilinx和Altera公司的的布局布线工具在Xilinx3098上和Altera Flex10K10实现了布线。 顺利通过综合前仿真、门级结构仿真以及布线后的门级仿真。这个 CPU 模型只是一个教学模型，设计也不一定合理，只是从原理上说明了一个简单的RISC _CPU的构成。我们在这里介绍它的目的是想说明：Verilog HDL 仿真和综合工具的潜力和本文介绍的设计方法对软硬件联合设计是有重要意义的。我们也希望这一章能引起对 CPU 原理和复杂数字逻辑系统设计有兴趣的同学的注意,加入我们的设计队伍。由于我们的经验与学识有限，不足之处敬请读者指正。

8.1.什么是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：
   a)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。
   b)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。
   c)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。

将其功能进一步细化，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，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。它是一种八十年代才出现的CPU，与一般的CPU 相比不仅只是简化了指令系统，而且是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用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用于教学目的的RISC_CPU的可综合VerilogHDL模型的设计和仿真过程。

8.2. RISC CPU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。从第四章我们知道可把它分成八个基本部件：

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8.2。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的小节里逐一进行介绍。

8.2.1时钟发生器

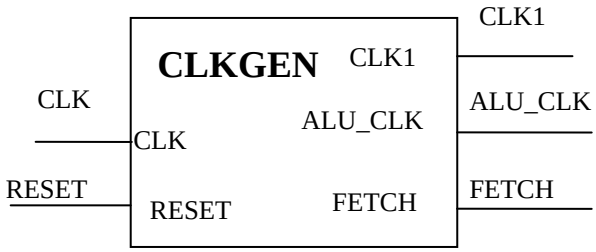


图1. 时钟发生器

时钟发生器 clkgen 利用外来时钟信号clk 来生成一系列时钟信号clk1、fetch、alu_clk 送往CPU的其他部件。其中fetch是外来时钟 clk 的八分频信号。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。
时钟发生器clkgen的波形见下图8.2.2所示：

其VerilogHDL 程序见下面的模块：

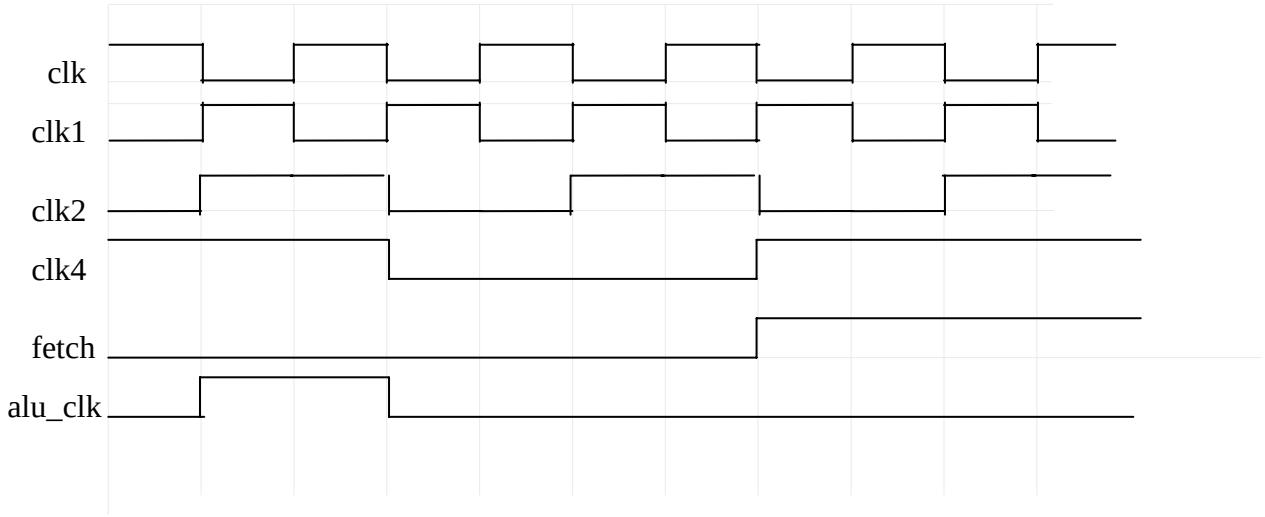


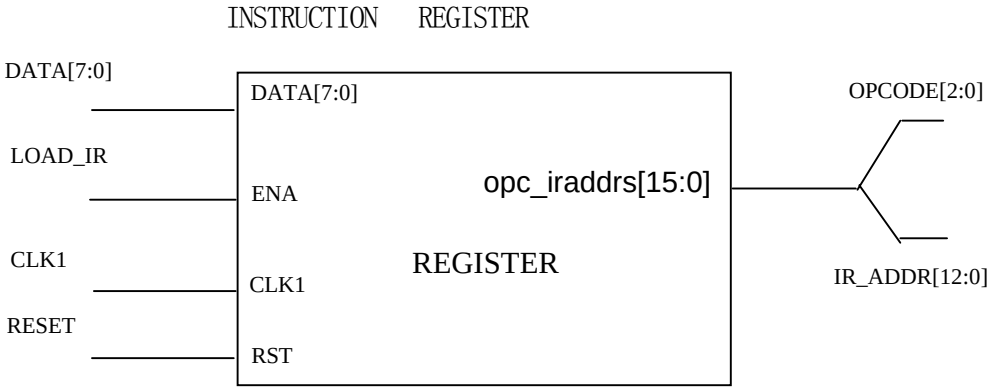
图8.2.2　时钟发生器clkgen的波形

```
module clk_gen (clk,reset,clk1,clk2,clk4,fetch,alu_clk);
input clk,reset;
output clk1,clk2,clk4,fetch,alu_clk;
wire clk,reset;
reg clk2,clk4,fetch,alu_clk;
reg[7:0] state;
parameter    S1 = 8'b00000001,
             S2 = 8'b00000010,
             S3 = 8'b00000100,
             S4 = 8'b00001000,
             S5 = 8'b00010000,
             S6 = 8'b00100000,
             S7 = 8'b01000000,
             S8 = 8'b10000000,
           idle = 8'b00000000;

assign clk1 = ~clk;

always @(negedge clk)
   if(reset)
           begin
                clk2 <= 0;
                clk4 <= 1;
                fetch  <= 0;
                alu_clk <= 0;
                state <= idle;
           end
   else
           begin
              case(state)
                 S1:
                      begin
                            clk2 <= ~clk2;
```

```
                    alu_clk <= ~alu_clk;
                      state <= S2;
                end
            S2:
                begin
                    clk2 <= ~clk2;
                    clk4 <= ~clk4;
                alu_clk <= ~alu_clk;
                  state <= S3;
                end
            S3:
                begin
                    clk2 <= ~clk2;
                   state <= S4;
                end
            S4:
                begin
                    clk2  <= ~clk2;
                    clk4  <= ~clk4;
                    fetch <= ~fetch;
                    state <= S5;
                end
            S5:
                 begin
                    clk2  <= ~clk2;
                    state <= S6;
                 end
            S6:
                 begin
                     clk2  <= ~clk2;
                      clk4 <= ~clk4;
                     state <= S7;
                  end
            S7:
                  begin
                       clk2  <= ~clk2;
                       state <=  S8;
                  end
            S8:
                  begin
                       clk2 <= ~clk2;
                      clk4  <= ~clk4;
                      fetch <= ~fetch;
                      state <= S1;
                  end
           idle:       state <= S1;
           default:    state <= idle;
        endcase
    end
endmodule
//-------------------------------------------------------------------------------
```

由于在时钟发生器的设计中采用了同步状态机的设计方法，不但使clk_gen模块的源程序可以被各种综合器综合，也使得由其生成的clk1、clk2、clk4、fetch、alu_clk 在跳变时间同步性能上有明显的提高，为整个系统的性能提高打下了良好的基础。

8.2.2 指令寄存器



顾名思义，指令寄存器用于寄存指令。

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。

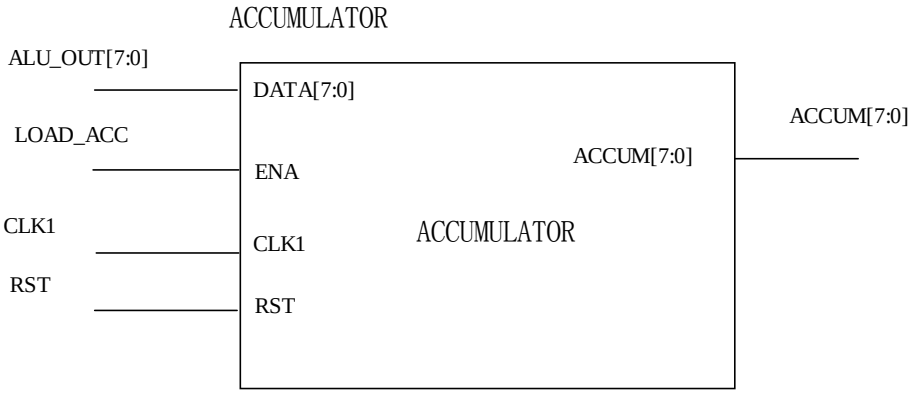
每条指令为2个字节，即16位。高3位是操作码，低13位是地址。（CPU的地址总线为13位，寻址空间为8K字节。）本设计的数据总线为8位，所以每条指令需取两次。先取高8位，后取低8位。而当前取的是高8位还是低8位，由变量state记录。state为零表示取的高8位，存入高8位寄存器，同时将变量state置为1。下次再寄存时，由于state为1，可知取的是低8位，存入低8位寄存器中。

其VerilogHDL 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
```

```
                    1'b0:  begin
                           opc_iraddr[15:8]<=data;
                           state<=1;
                           end
                    1'b1:  begin
                           opc_iraddr[7:0]<=data;
                           state<=0;
                           end
                    default:  begin
                                opc_iraddr[15:0]<=16'bxxxxxxxxxxxxxxxx;
                                state<=1'bx;
                              end
              endcase
            end
          else
            state<=1'b0;
          end
        end
        endmodule
//-------------------------------------------------------
```

8.2.3.累加器



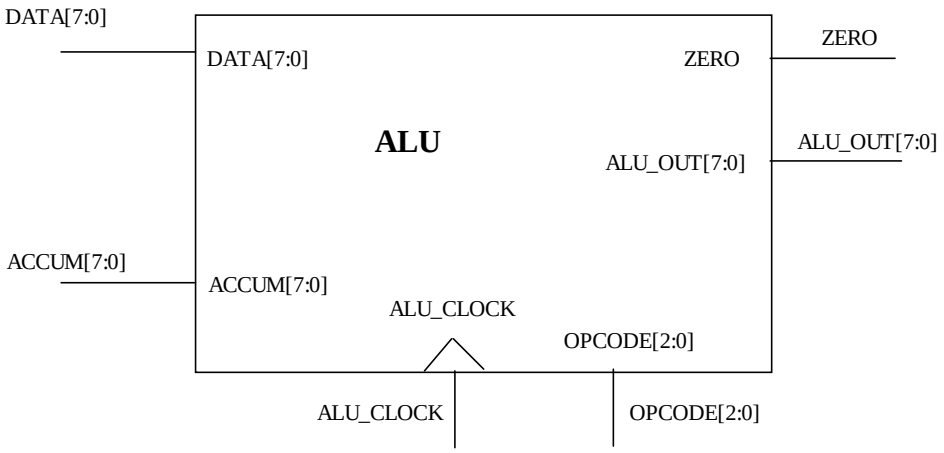
累加器用于存放当前的结果，它也是双目运算其中一个数据来源。复位后，累加器的值是零。当累加器通过ena口收到来自CPU状态控制器load_acc信号时，在clk1时钟正跳沿时就收到来自于数据总线的数据。

其VerilogHDL 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;

          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
```

8.2.4.算术运算器



算术逻辑运算单元 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它运算以及逻辑判断等操作。

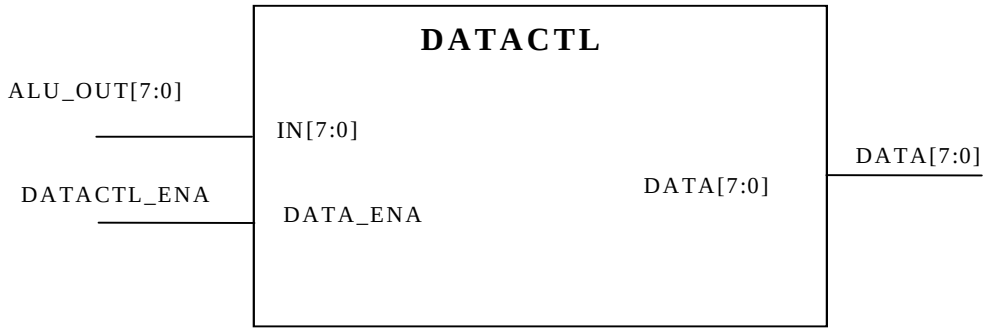
其VerilogHDL 程序见下面的模块：

```
//-------------------------------------------------------------------------
      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  =3'b000,
                       SKZ  =3'b001,
                       ADD  =3'b010,
                       ANDD =3'b011,
                       XORR =3'b100,
                       LDA  =3'b101,
                       STO  =3'b110,
                       JMP  =3'b111;

        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
//-------------------------------------------------------------------------
```

8.2.5.数据控制器

数据控制器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，

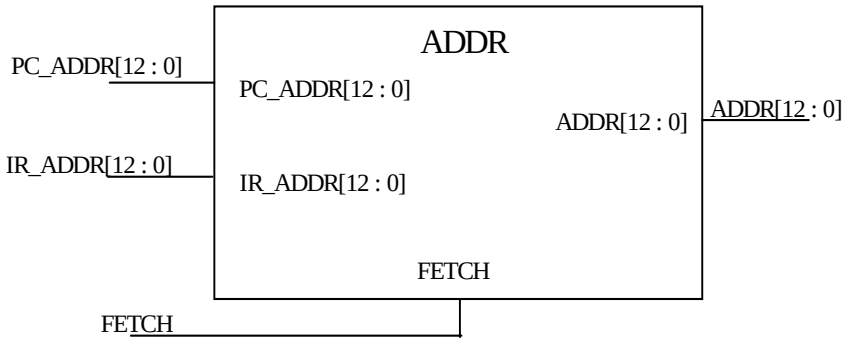


不同的情况下传送不同的内容。有时要传输指令，有时要传送RAM区或接口的数据。累加器的数据只有在需要往RAM区或端口写时才允许输出，否则应呈现高阻态，以允许其它部件使用数据总线。 所以任何部件往总线上输出数据时，都需要一控制信号。而此控制信号的启、停，则由CPU状态控制器输出的各信号控制决定。数据控制器何时输出累加器的数据则由状态控制器输出的控制信号datactl_ena决定。

其VerilogHDL 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
//-------------------------------------------------------------------
```

8.2.6.地址多路器



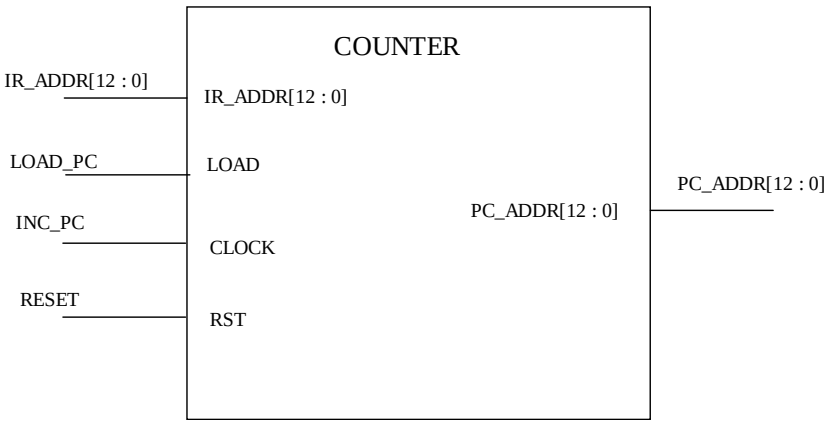
地址多路器用于选择输出的地址是PC（程序计数）地址还是数据/端口地址。每个指令周期的前4个时钟周期用于从ROM中读取指令，输出的应是PC地址。后4个时钟周期用于对RAM或端口的读写，该地址由指令中给出。地址的选择输出信号由时钟信号的8分频信号fetch提供。

其VerilogHDL 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
//-------------------------------------------------------------------------
```

8.2.7.程序计数器

程序计数器用于提供指令地址。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。



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2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VerilogHDL 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
//-------------------------------------------------------------------------------
```

8.2.8.状态控制器

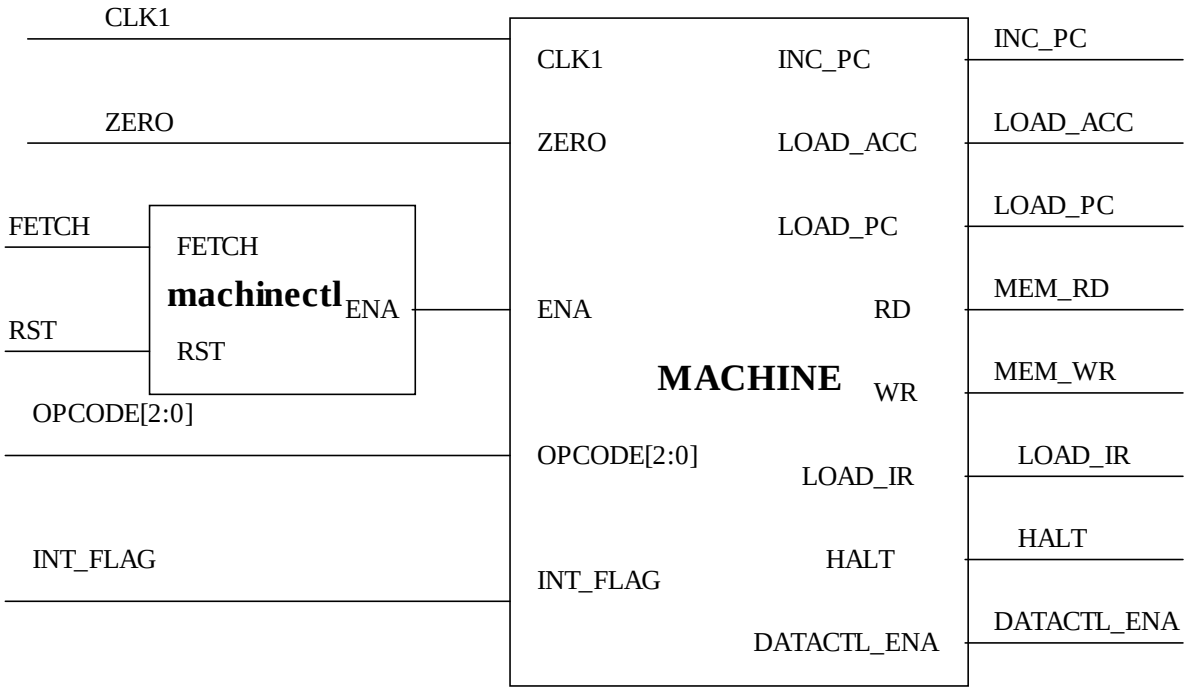


图8.2.8状态控制器

状态控制器由两部分组成：

1.状态机(上图中的MACHINE部分)
2.状态控制器(上图中的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VerilogHDL程序见下面模块：

```
//------------------------------------------------------------------------------
        module machinectl( ena, fetch, rst);
        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已经过的时钟数（从零计起）。

指令周期是由8个时钟周期组成，每个时钟周期都要完成固定的操作。

1) 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。

2) 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。

3) 第2个时钟，空操作。

4) 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增一外（指向下一条指令），其它各控制线输出为零。

5) 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。

6) 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。

7) 第6个时钟，空操作。

8) 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VerilogHDL 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)
    3'b000:          //load high 8bits in struction
            begin
               {inc_pc,load_acc,load_pc,rd}<=4'b0001;
               {wr,load_ir,datactl_ena,halt}<=4'b0100;
               state<=3'b001;
            end
    3'b001:         //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<=3'b010;
            end
    3'b010:         //idle
            begin
               {inc_pc,load_acc,load_pc,rd}<=4'b0000;
               {wr,load_ir,datactl_ena,halt}<=4'b0000;
               state<=3'b011;
            end

    3'b011:         //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<=3'b100;
            end
    3'b100:               //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<=3'b101;
          end
3'b101:          //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<=3'b110;
              end
            3'b110:          //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<=3'b111;
              end

            3'b111:          //
              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<=3'b000;
              end
            default:
              begin
                 {inc_pc,load_acc,load_pc,rd}<=4'b0000;
                 {wr,load_ir,datactl_ena,halt}<=4'b0000;
                 state<=3'b000;
              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。见本小节的图8.2.8。

8.2.9.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、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。

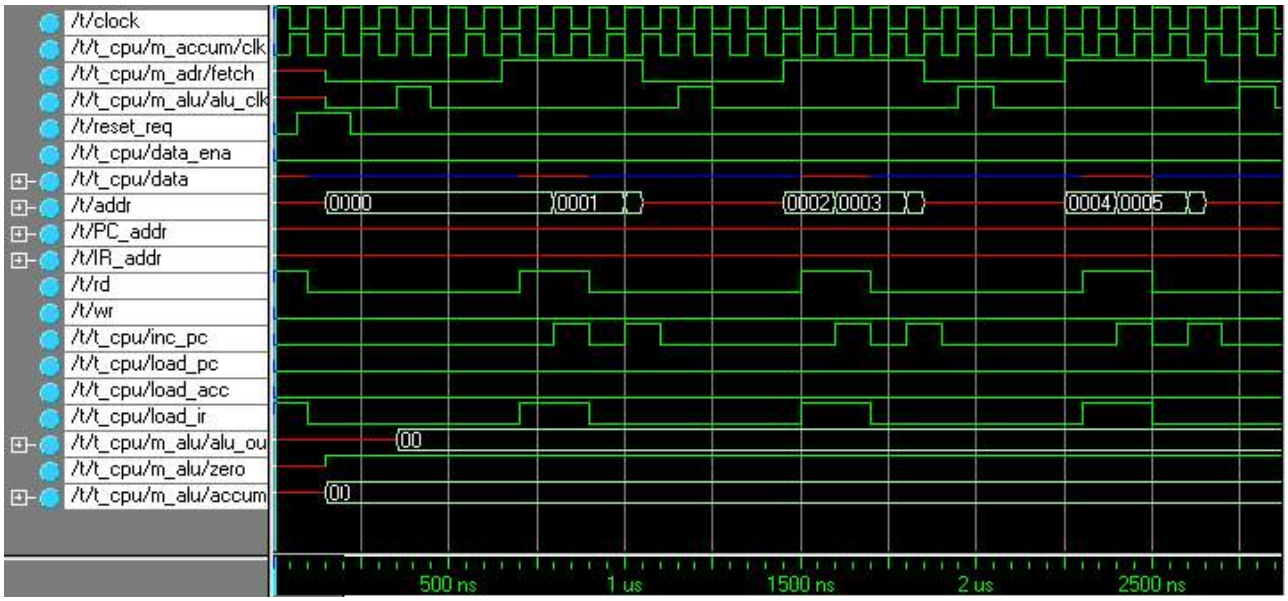
8.3. RISC_CPU 操作和时序

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
下面详细介绍一下每个操作：

8.3.1.系统的复位和启动操作

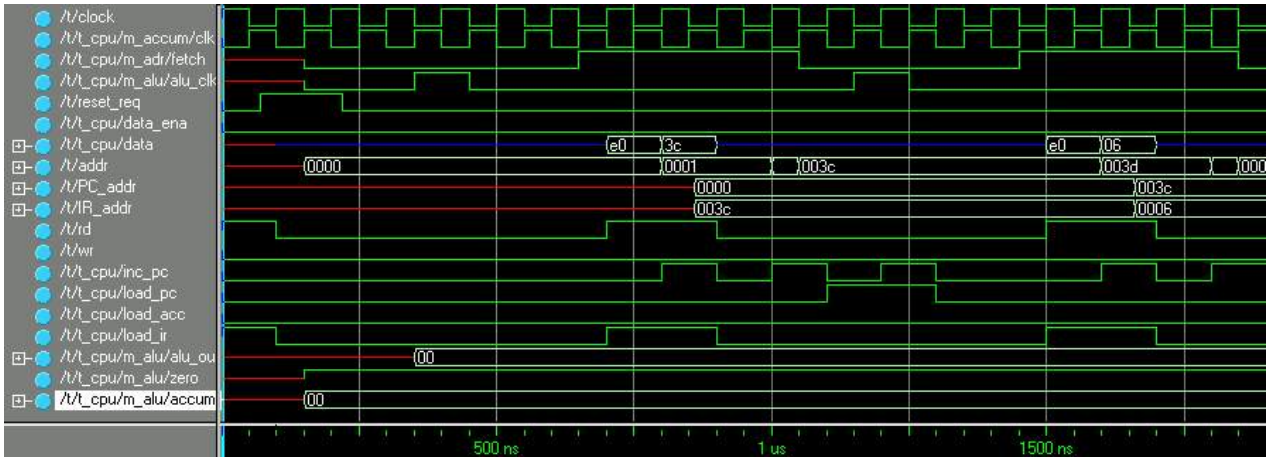
RISC_CPU的复位和启动操作是通过rst引脚的信号触发执行的。当rst信号一进入高电平，RISC_CPU就会结束现行操作，并且只要rst停留在高电平状态，CPU就维持在复位状态。在复位状态，CPU各内部寄存器都被设为初值，全部为零。数据总线为高阻态，地址总线为0000H，所有控制信号均为无效状态。rst回到低电平后，接着到来的第一个fetch上升沿将启动RISC_CPU开始工作，从ROM的000处开始读取指令并执行相应操作。波形图见8.3.1。虚线标志处为RISC_CPU启动工作的时刻。

8.3.2.总线读操作



RISC_CPU的复位和启动操作波形

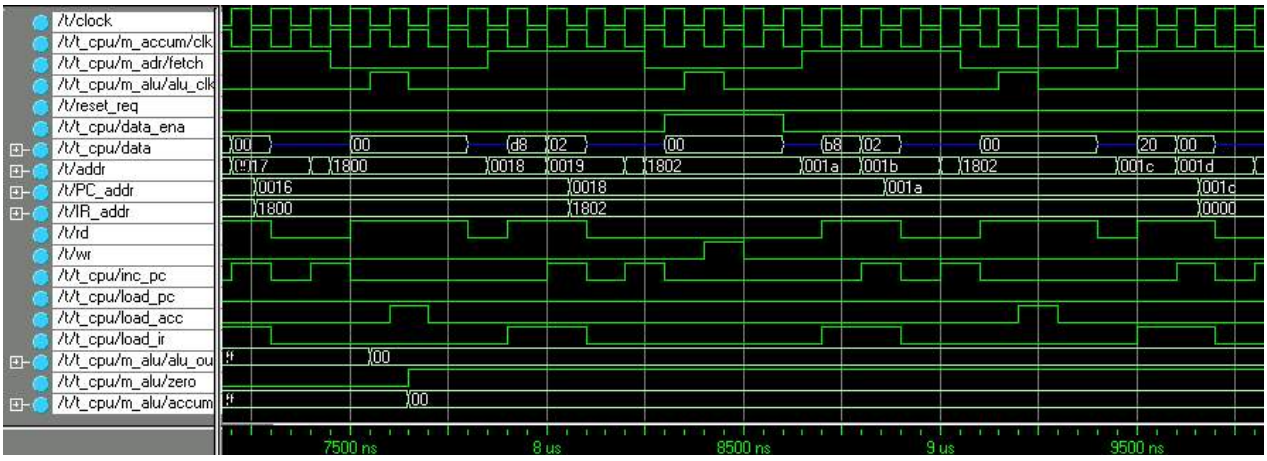
每个指令周期的前0--3个时钟周期用于读指令，在状态控制器一节中已详细讲述，这里就不再重复。第3.5个周期处，存储器或端口地址就输出到地址总线上，第4--6个时钟周期，读信号rd有效，数据送到数据总线上，以备累加器锁存，或参与算术、逻辑运算。第7个时钟周期，读信号无效，第7.5个周期，地址总线输出PC地址，为下一个指令做好准备。



CPU从存储器或端口读取数据的时序

8.3.3写总线操作

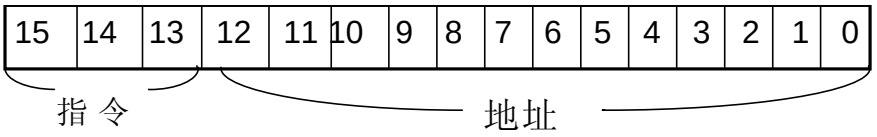
每个指令周期的第3.5个时钟周期处，写的地址就建立了，第4个时钟周期输出数据，第5个时钟周期输出写信号。至第6个时钟结束，数据无效，第7.5时钟地址输出为PC地址，为下一个指令周期做好准备。



CPU对存储器或端口写数据的时序

8.4.RISC_CPU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。

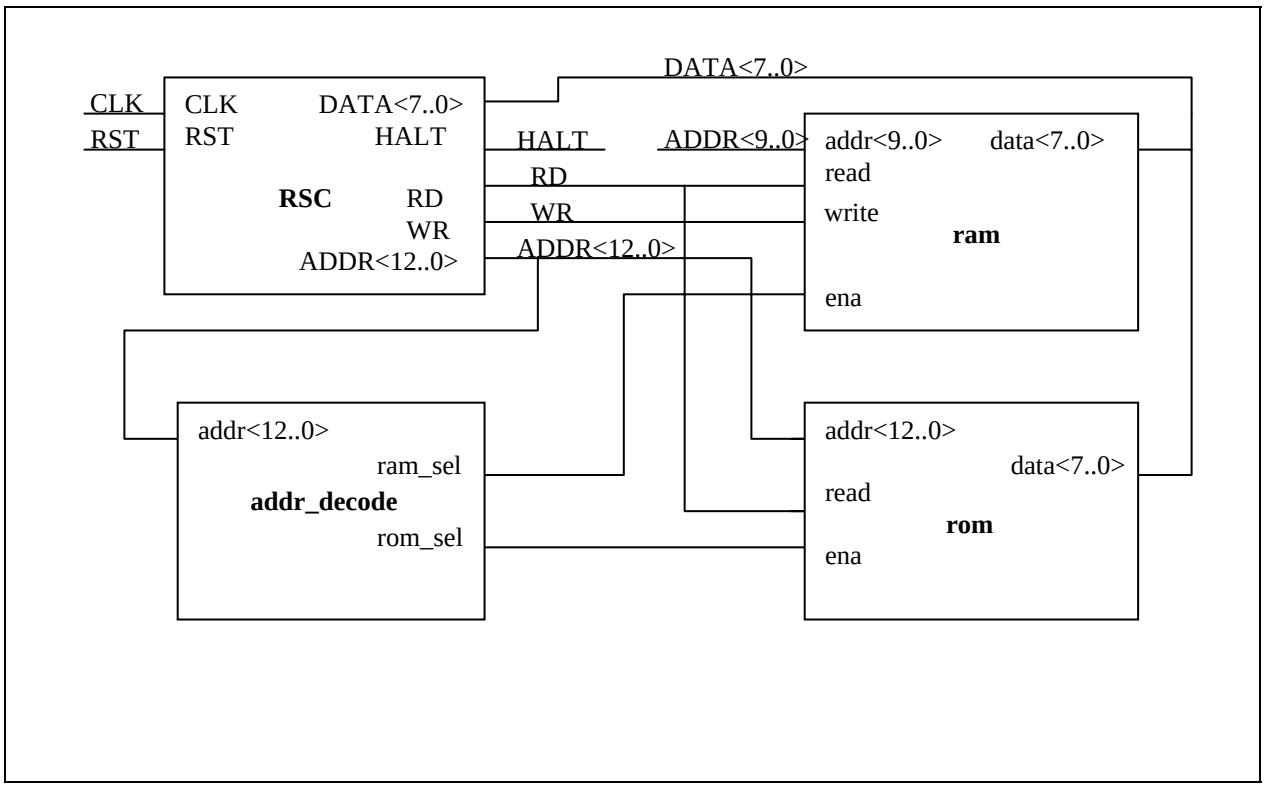
1) HLT 停机操作。该操作将空一个指令周期，即8个时钟周期。
2) SKZ 为零跳过下一条语句。该操作先判断当前alu中的结果是否为零，若是零就跳过下一条语句，否则继续执行。
3) ADD 相加。该操作将累加器中的值与地址所指的存储器或端口的数据相加，结果仍送回累加器中。
4) AND 相与。该操作将累加器的值与地址所指的存储器或端口的数据相与，结果仍送回累加器中。
5) XOR 异或。该操作将累加器的值与指令中给出地址的数据异或，结果仍送回累加器中。
6) LDA 读数据。该操作将指令中给出地址的数据放入累加器。
7) STO 写数据。该操作将累加器的数据放入指令中给出的地址。
8) JMP 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。

8.5. RISC_CPU模块的调试

8.5.1. 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，从系统的角度看就显得简洁，见图8.5.2。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VerilogHDL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，上面8.2.9节中的ROM和RAM模块就是简化的虚拟器件的例子，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，见图8.5.1;也可以用VerilogHDL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，



RISC_CPU和它的外围电路

下面介绍的是在modelsim 5.4 下进行调试的仿真测试程序cputop.v。可用于对以上所设计的RISCCPU进行仿真测试，下面是前仿真的测试程序cputop.v。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cputop.v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。

```
`include "ram.v"
`include "rom.v"
`include "addrdecode.v"
`include "cpu.v"
`timescale 1ns / 100ps
`define PERIOD 100                  // matches clk_gen.v
module t;
  reg reset_req,clock;
  integer test;
  reg [(3*8):0] mnemonic;           //array that holds 3 8-bit ASCII characters
  reg [12:0] PC_addr,IR_addr;
  wire [7:0] data;
  wire [12:0] addr;
  wire rd,wr,halt,ram_sel,rom_sel;
//--------------------------------------------------------------------------
cpu   t_cpu (.clk(clock),.reset(reset_req),.halt(halt),.rd(rd),
                                    .wr(wr),.addr(addr),.data(data));

ram   t_ram  (.addr(addr[9:0]),.read(rd),.write(wr),.ena(ram_sel),.data(data));

rom   t_rom  (.addr(addr),.read(rd),.ena(rom_sel),.data(data));
```

```
addr_decode   t_addr_decode (.addr(addr),.ram_sel(ram_sel),.rom_sel(rom_sel));

//------------------------------------------------------------------------
initial
  begin
    clock=1;
   //display time in nanoseconds
    $timeformat ( -9,  1, " ns", 12);
    display_debug_message;
    sys_reset;
    test1;
    $stop;
    test2;
    $stop;
    test3;
    $stop;
end
  task display_debug_message;
    begin
      $display("\n*********************************************************");
      $display("*  THE FOLLOWING DEBUG TASK ARE AVAILABLE:           *");
      $display("* \"test1; \" to load the 1st diagnostic progran. *");
      $display("*  \"test2; \" to load the 2nd diagnostic program. *");
      $display("*  \"test3; \" to load the Fibonacci program.      *");
      $display("*********************************************************\n");
    end
  endtask
   task test1;
    begin
      test = 0;
      disable MONITOR;
      $readmemb ("test1.pro", t_rom.memory);
      $display("rom loaded   successfully!");
      $readmemb("test1.dat",t_ram.ram);
      $display("ram loaded   successfully!");
      #1 test = 1;
      #14800  ;
      sys_reset;
    end
   endtask

  task test2;
    begin
      test = 0;
      disable MONITOR;
      $readmemb("test2.pro",t_rom.memory);
      $display("rom loaded  successfully!");
      $readmemb("test2.dat",t_ram.ram);
      $display("ram loaded  successfully!");
      #1 test = 2;
      #11600;
      sys_reset;
```

```
    end
   endtask

  task test3;
    begin
      test = 0;
      disable MONITOR;
      $readmemb("test3.pro",t_rom.memory);
      $display("rom loaded  successfully!");
      $readmemb("test3.dat",t_ram.ram);
      $display("ram loaded  successfully!");
      #1 test = 3;
      #94000;
      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)
                  @(t_cpu.m_adr.pc_addr)//fixed
                  if ((t_cpu.m_adr.pc_addr%2 == 1)&&(t_cpu.m_adr.fetch == 1))//fixed
               begin
                 # 60    PC_addr <=t_cpu.m_adr.pc_addr -1 ;
                         IR_addr <=t_cpu.m_adr.ir_addr;
                # 340   $strobe("%t   %h     %s     %h
%h",$time,PC_addr,mnemonic,IR_addr,data );//HERE DATA HAS BEEN CHANGED
T-CPU-M-REGISTER.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)
                @(t_cpu.m_adr.pc_addr)
```

```
            if ((t_cpu.m_adr.pc_addr%2 == 1)
                                && (t_cpu.m_adr.fetch == 1))
            begin
             # 60    PC_addr  <= t_cpu.m_adr.pc_addr - 1 ;
                     IR_addr  <= t_cpu.m_adr.ir_addr;
             # 340   $strobe("%t  %h  %s  %h %h", $time, PC_addr,
                                          mnemonic, IR_addr, data );
            end

        end

      3: begin
           $display("\n***   RUNNING CPUtest3 - An Executable Program   ***");
           $display("*** This program should calculate the fibonacci  ***");
           $display("\n    TIME      FIBONACCI NUMBER");
           $display(  "  ---------   -----------------");
           while (test == 3)
            begin
               wait ( t_cpu.m_alu.opcode == 3'h1) // display Fib. No. at end of program loop
               $strobe("%t     %d", $time,t_ram.ram[10'h2]);
               wait ( t_cpu.m_alu.opcode != 3'h1);
            end
        end
    endcase

   end
//-------------------------------------------------------------------------------
--
always @(posedge halt)       //STOP when HALT instruction decoded
    begin
      #500
         $display("\n************************************************************");
         $display("*  A HALT INSTRUCTION WAS PROCESSED  !!!  *");
         $display("************************************************************\n");
    end
always #(`PERIOD/2) clock=~clock;
always  @(t_cpu.m_alu.opcode)
     //get an ASCII mnemonic for each opcode
    case(t_cpu.m_alu.opcode)
     3'b000  : 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
```

针对程序做如下说明：测试程序中用`include"“ 形式包含了“ rom.v ”，“ ram.v”和“addrdecode.v”三个外部模块，它们都是检测RISCCPU时必不可少虚拟设备。代表RAM ,ROM和地址译码器，对于RISCCPU，已将它做成一个独立的模块“cpu.v”。具体程序如下：

```
//-------------------------------------------------------------------
`include "clk_gen.v"
`include "accum.v"
`include "adr.v"
`include "alu.v"
`include "machine.v"
`include "counter.v"
`include "machinect1.v"
`include "register.v"
`include "datact1.v"

module cpu(clk,reset,halt,rd,wr,addr,data);
 input clk,reset;
 output rd,wr,addr,halt;
 inout data;
 wire clk,reset,halt;
 wire [7:0]  data;
 wire [12:0] addr;
 wire rd,wr;
 wire clk1,fetch,alu_clk;
 wire [2:0] opcode;
 wire [12:0] ir_addr,pc_addr;
 wire [7:0] alu_out,accum;
 wire zero,inc_pc,load_acc,load_pc,load_ir,data_ena,contr_ena;

clk_gen  m_clk_gen (.clk(clk),.clk1(clk1),.fetch(fetch),
                              .alu_clk(alu_clk),.reset(reset));

register  m_register (.data(data),.ena(load_ir),.rst(reset),
                                 .clk1(clk1),.opc_iraddr({opcode,ir_addr}));

accum      m_accum    (.data(alu_out),.ena(load_acc),
                                    .clk1(clk1),.rst(reset),.accum(accum));

alu        m_alu      (.data(data),.accum(accum),.alu_clk(alu_clk),
                             .opcode(opcode),.alu_out(alu_out),.zero(zero));

machinect1  m_machinec1(.ena(contr_ena),.fetch(fetch),.rst(reset));

machine    m_machine  (.inc_pc(inc_pc),.load_acc(load_acc),.load_pc(load_pc),
                       .rd(rd), .wr(wr), .load_ir(load_ir), .clk1(clk1),
                       .datactl_ena(data_ena), .halt(halt), .zero(zero),
                       .ena(contr_ena),.opcode(opcode));

datactl    m_datactl  (.in(alu_out),.data_ena(data_ena),.data(data));

adr        m_adr (.fetch(fetch),.ir_addr(ir_addr),.pc_addr(pc_addr),.addr(addr));

counter    m_counter  (.ir_addr(ir_addr),.load(load_pc),.clock(inc_pc),
```

```
                                        .rst(reset),.pc_addr(pc_addr));

endmodule
```

其中 contr_ena 用于machinect1与machine之间的ena的连接。cputop.v 中用到下面两条语句需要解释一下：

```
        $readmemb  ( "test1.pro",t_rom_.memory ); 和
        $readmemb  ( "test1.dat",t_ram_.ram);
```

即可把编译好的汇编机器码装人虚拟ROM,把需要参加运算的数据装人虚拟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。

下面清单所列出是用于测试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cputop.v文件：

```
//------------------ 文件  test1.pro ----------------------------------------
/*****************************************************************************
 * Test1 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-----------------------------------------

/*****************************************************************************************
**
下面文件中的数据在仿真时需要用系统任务$readmemb读入RAM，才能被上面的汇编程序test1.pro使用。
*****************************************************************************************
****/
//-----------------------------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-----------------------------------------


/*****************************************************************************************
**
下面文件中的数据在仿真时需要用系统任务$readmemb读入RAM，才能被上面的汇编程序test2.pro使用。
*****************************************************************************************
****/
//----------------------------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:
//----------------------------test2.dat结束 .---------------------------------------------


/*****************************************************************************
 * Test 3 程序是一个计算从0到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-----------------------------------------

/*************************************************************************************
**
下面文件中的数据在仿真时需要用系统任务$readmemb读入RAM，才能被上面的汇编程序test3.pro使用。
*************************************************************************************
****/
//-----------------------------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-----------------------------------------
```

以下介绍前仿真的步骤，首先按照表示各模块之间连线的电路图编制测试文件，即定义Verilog的wire变量作为连线，连接各功能模块之间的引脚，并将输入信号引入，输出信号引出。如若需要，可加入必要的语句显示提示信息。例如，risc_cpu 的测试文件就是cputop.v。其次，使用仿真软件进行仿真，由于不同的软件使用方法可能有较大的差异，以下只简单的介绍modelsim的使用。在进入modelsim的环境之后，在file项选择change direction来确定编制的文件所在的目录，然后在design项选择或创建一个library，完成后即可开始编译。在design项选compile...项，进入编译环境，选定要编译的文件进行编译。Modelsim的编译器语法检查并不严格，有时会出现莫名其妙的逻辑错误，书写时应注意笔误。完成编译后，还是在compile...项，选择load new design项，选中编译后提示的top module的名字，然后开始仿真。在view项可选波形显示，信号选择，功能和操作简单明了，这里就不一一赘述。

仿真结果如下：

```
run -all
#
# ****************************************************
```

```
# *  THE FOLLOWING DEBUG TASK ARE AVAILABLE:           *
# * "test1; " to load the 1st diagnostic progran. *
# *  "test2; " to load the 2nd diagnostic program. *
# *  "test3; " to load the Fibonacci program.      *
# ****************************************************
#
# rom loaded   successfully!
# ram loaded   successfully!
#
# *** RUNNING CPUtest1 - The Basic CPU Diagnostic Program ***
#
#     TIME         PC        INSTR    ADDR     DATA
#    ----------  ---       -----    ----    -----
#    1200.0 ns   0000       JMP     003c       zz
#    2000.0 ns   003c       JMP     0006       zz
#    2800.0 ns   0006       LDA     1800       00
#    3600.0 ns   0008       SKZ     0000       zz
#    4400.0 ns   000c       LDA     1801       ff
#    5200.0 ns   000e       SKZ     0000       zz
#    6000.0 ns   0010       JMP     0014       zz
#    6800.0 ns   0014       STO     1802       ff
#    7600.0 ns   0016       LDA     1800       00
#    8400.0 ns   0018       STO     1802       00
#    9200.0 ns   001a       LDA     1802       00
#   10000.0 ns   001c       SKZ     0000       zz
#   10800.0 ns   0020       XOR     1801       ff
#   11600.0 ns   0022       SKZ     0000       zz
#   12400.0 ns   0024       JMP     0028       zz
#   13200.0 ns   0028       XOR     1801       ff
#   14000.0 ns   002a       SKZ     0000       zz
#   14800.0 ns   002e       HLT     0000       zz
#
# ************************************************************
# *  A HALT INSTRUCTION WAS PROCESSED  !!!  *
# ************************************************************
#
# Break at H:/seda/w/Cputop.v line 109
run -continue
# rom loaded  successfully!
# ram loaded  successfully!
#
# *** RUNNING CPUtest2 - The Advanced CPU Diagnostic Program ***
#
#      TIME        PC     INSTR  ADDR  DATA
#    ----------   ----   -----  ----  -----
#   16200.0 ns   0000    LDA   1801   aa
#   17000.0 ns   0002    AND   1802   ff
#   17800.0 ns   0004    XOR   1801   aa
#   18600.0 ns   0006    SKZ   0000   zz
#   19400.0 ns   000a    ADD   1800   01
#   20200.0 ns   000c    SKZ   0000   zz
#   21000.0 ns   000e    JMP   0012   zz
```

```
#   21800.0 ns   0012    XOR   1802   ff
#   22600.0 ns   0014    ADD   1800   01
#   23400.0 ns   0016    STO   1803   ff
#   24200.0 ns   0018    LDA   1800   01
#   25000.0 ns   001a    ADD   1803   ff
#   25800.0 ns   001c    SKZ   0000   zz
#   26600.0 ns   0020    HLT   0000   zz
#
# ************************************************************
# *  A HALT INSTRUCTION WAS PROCESSED  !!!  *
# ************************************************************
#
# Break at H:/seda/w/cputop.v line 111
run -continue
# rom loaded  successfully!
# ram loaded  successfully!
#
# ***   RUNNING CPUtest3 - An Executable Program   ***
# *** This program should calculate the fibonacci  ***
#
#     TIME      FIBONACCI NUMBER
#   ---------   -----------------
#   33250.0 ns       0
#   40450.0 ns       1
#   47650.0 ns       1
#   54850.0 ns       2
#   62050.0 ns       3
#   69250.0 ns       5
#   76450.0 ns       8
#   83650.0 ns      13
#   90850.0 ns      21
#   98050.0 ns      34
#  105250.0 ns      55
#  112450.0 ns      89
#  119650.0 ns     144
#
# ************************************************************
# *  A HALT INSTRUCTION WAS PROCESSED  !!!  *
# ************************************************************
#
# Break at H:/seda/w/cputop.v line 112
```

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。

8.5.2. 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即综合。综合工作往往要分阶段来进行，这样便于发现问题。

所谓分阶段就是指：

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。

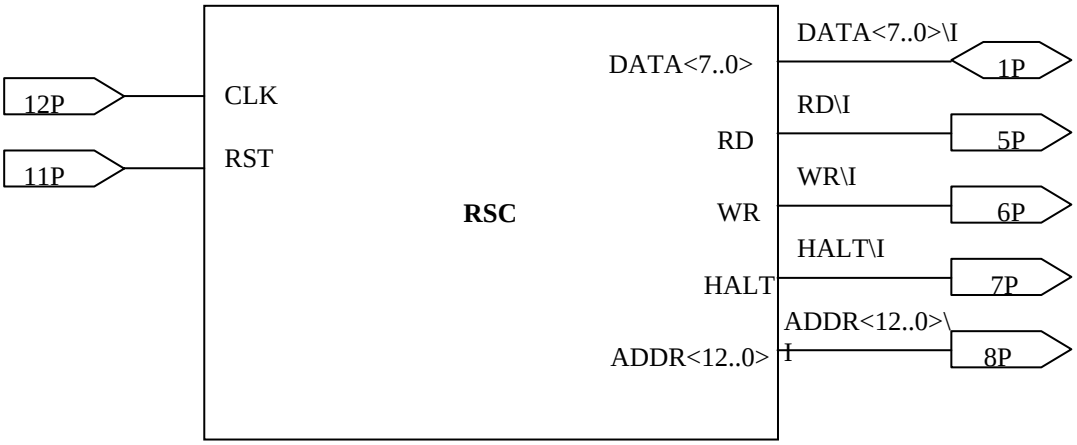


图8.5.2 用于综合的RISC_CPU模块（RSC）

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，我们还需给它的引脚加上标准的引脚部件并加标记，见图8.5.2。

第三阶段：加载需要综合的模块到综合器，本例所使用的综合器是 Synplify，选定的 FPGA 是 Altera FLEX10K，针对它的库进行综合。

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所使用的基本单元，各部件的时间参数以及综合的过程。见下面的报告,它就是这个RISC_CPU芯片所用的综合报告，综合所用的库为Altera FLEX10K系列的FPGA库。

```
$ Start of Compile
#Fri Jul 21 10:11:03 2000

Synplify Verilog Compiler, version 5.2.2, built Aug 20 1999
Copyright (C) 1994-1999, Synplicity Inc.  All Rights Reserved

@I::"h:\seda\w\cpu.v"
Verilog syntax check successful!
Selecting top level module cpu
Synthesizing module clk_gen
Synthesizing module register
Synthesizing module accum
Synthesizing module alu
Synthesizing module machinect1
Synthesizing module machine
Synthesizing module datactl
```

```
Synthesizing module adr
Synthesizing module counter
Synthesizing module cpu
@END
Process took 0.491 seconds realtime, 0.54 seconds cputime
Synplify Altera Technology Mapper, version 5.2.2, built Aug 31 1999
Copyright (C) 1994-1998, Synplicity Inc.  All Rights Reserved
Loading timing data for chip EPF10K10-3
List of partitions to map:
   view:work.cpu(verilog)
Automatic dissolve at startup in view:work.cpu(verilog) of m_counter(counter)
Automatic dissolve at startup in view:work.cpu(verilog) of m_adr(adr)
Automatic dissolve at startup in view:work.cpu(verilog) of m_datactl(datactl)
Automatic dissolve at startup in view:work.cpu(verilog) of m_machinec1(machinect1)
@N:"h:\seda\w\cpu.v":347:0:347:5|Found counter in view:work.cpu(verilog) inst
m_counter.pc_addr[12:0]
Automatic dissolve during optimization of view:work.cpu(verilog) of m_alu(alu)
Automatic dissolve during optimization of view:work.cpu(verilog) of m_accum(accum)
Loading timing data for chip EPF10K10-3
Found clock m_machine.inc_pc with period 100ns
Found clock m_clk_gen.fetch with period 100ns
Found clock m_clk_gen.alu_clk with period 100ns
Found clock clk with period 100ns

              ##### START TIMING REPORT #####
Set the Environment Variable SYNPLIFY_TIMING_REPORT_OLD to get the old timing report


              Performance Summary
             *********************

                        Requested     Estimated     Requested     Estimated
Clock                   Frequency     Frequency     Period        Period        Slack
--------------------------------------------------------------------------------
m_machine.inc_pc        10.0 MHz      95.2 MHz      100.0         10.5          89.5
m_clk_gen.alu_clk       10.0 MHz      59.5 MHz      100.0         16.8          83.2
clk                     10.0 MHz      16.8 MHz      100.0         59.5          40.5
================================================================================

              Interface Information
             ***********************

Input Ports:

Port        Reference                       User        Arrival     Required
Name        Clock                           Constraint  Time        Time        Slack
--------------------------------------------------------------------------------
----
clk         System                          0.0           0.0         >2000.0     NA
data[0]     m_clk_gen.alu_clk [rising]      0.0           0.0         85.9        85.9
data[1]     m_clk_gen.alu_clk [rising]      0.0           0.0         86.2        86.2
data[2]     m_clk_gen.alu_clk [rising]      0.0           0.0         86.5        86.5
```

```
data[3]     m_clk_gen.alu_clk [rising]     0.0          0.0         87.0         87.0
data[4]     m_clk_gen.alu_clk [rising]     0.0          0.0         88.8         88.8
data[5]     m_clk_gen.alu_clk [rising]     0.0          0.0         89.1         89.1
data[6]     m_clk_gen.alu_clk [rising]     0.0          0.0         89.4         89.4
data[7]     m_clk_gen.alu_clk [rising]     0.0          0.0         89.9         89.9
reset       clk [falling]                  0.0          0.0         91.9         91.9
================================================================================
====

Output Ports:

Port          Reference                      User       Arrival     Required
Name          Clock                       Constraint     Time        Time         Slack
--------------------------------------------------------------------------------
-----
addr[0]      clk [falling]                  0.0          6.9         100.0        93.1
addr[1]      clk [falling]                  0.0          6.9         100.0        93.1
addr[2]      clk [falling]                  0.0          6.9         100.0        93.1
addr[3]      clk [falling]                  0.0          6.9         100.0        93.1
addr[4]      clk [falling]                  0.0          6.9         100.0        93.1
addr[5]      clk [falling]                  0.0          6.9         100.0        93.1
addr[6]      clk [falling]                  0.0          6.9         100.0        93.1
addr[7]      clk [falling]                  0.0          6.9         100.0        93.1
addr[8]      clk [falling]                  0.0          6.9         100.0        93.1
addr[9]      clk [falling]                  0.0          6.9         100.0        93.1
addr[10]     clk [falling]                  0.0          6.9         100.0        93.1
addr[11]     clk [falling]                  0.0          6.9         100.0        93.1
addr[12]     clk [falling]                  0.0          6.9         100.0        93.1
data[0]      m_clk_gen.alu_clk [rising]     0.0          0.0         100.0        100.0
data[1]      m_clk_gen.alu_clk [rising]     0.0          0.0         100.0        100.0
data[2]      m_clk_gen.alu_clk [rising]     0.0          0.0         100.0        100.0
data[3]      m_clk_gen.alu_clk [rising]     0.0          0.0         100.0        100.0
data[4]      m_clk_gen.alu_clk [rising]     0.0          0.0         100.0        100.0
data[5]      m_clk_gen.alu_clk [rising]     0.0          0.0         100.0        100.0
data[6]      m_clk_gen.alu_clk [rising]     0.0          0.0         100.0        100.0
data[7]      m_clk_gen.alu_clk [rising]     0.0          0.0         100.0        100.0
halt         clk [rising]                   0.0          1.0         100.0        99.0
rd           clk [rising]                   0.0          1.0         100.0        99.0
wr           clk [rising]                   0.0          1.0         100.0        99.0
================================================================================

             Detailed Timing Report for  clock : clk
             ****************************************
Requested Period      100.0 ns
Estimated Period      59.5 ns
Worst Slack          40.5 ns

Start Points for Paths with Slack Worse than 42.8 ns :

                                                                    Arrival
Instance                Type      Pin     Net                       Time        Slack
--------------------------------------------------------------------------------
```

```
m_machine.load_ir       S_DFF     Q       m_machine.load_ir       51.4        40.5
m_machine.load_acc      S_DFF     Q       m_machine.load_acc      51.0        41.1
================================================================================

End Points for Paths with Slack Worse than 42.8 ns :

                                                                      Required
Instance                    Type      Pin     Net                     Time        Slack
--------------------------------------------------------------------------------
----------
m_register.opc_iraddr[8]    S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
m_register.opc_iraddr[7]    S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
m_register.opc_iraddr[6]    S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
m_register.opc_iraddr[5]    S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
m_register.opc_iraddr[4]    S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
m_register.opc_iraddr[3]    S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
m_register.opc_iraddr[2]    S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
m_register.opc_iraddr[1]    S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
m_register.opc_iraddr[15]   S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
m_register.opc_iraddr[14]   S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
================================================================================

A Critical Path with worst case slack = 40.5 ns:

Instance/Net                          Pin       Pin      Arrival     Delta     Fan
Name                        Type      Name      Dir      Time        Delay     Out
--------------------------------------------------------------------------------
m_machine.load_ir           S_DFF     Q         Out      51.4        51.4
m_machine.load_ir           Net                                              2
m_register.un1_un1_rst      S_LUT     I1        In       51.4
m_register.un1_un1_rst      S_LUT     OUT       Out      57.3        5.9
m_register.un1_un1_rst      Net                                              16
m_register.opc_iraddr[0]    S_DFFE    ENA       In       57.3
================================================================================
Setup requirement on this path is 2.2 ns.


            Detailed Timing Report for  clock : m_clk_gen.alu_clk
            ********************************************
Requested Period      100.0 ns
Estimated Period      16.8 ns
Worst Slack          83.2 ns

Start Points for Paths with Slack Worse than 85.5 ns :

                                                                      Arrival
Instance                    Type      Pin     Net                     Time        Slack
--------------------------------------------------------------------------------
m_accum.accum[1]            S_DFFE    Q       m_accum.accum[1]          3.0         83.2
m_accum.accum[0]            S_DFFE    Q       m_accum.accum[0]          2.6         83.3
m_accum.accum[2]            S_DFFE    Q       m_accum.accum[2]          2.6         83.9
m_accum.accum[3]            S_DFFE    Q       m_accum.accum[3]          2.6         84.4
```

```
m_register.opc_iraddr[14]    S_DFFE    Q     m_register.opc_iraddr[14]    4.4       85.5
================================================================================

End Points for Paths with Slack Worse than 85.5 ns :

                                                                  Required
Instance              Type      Pin     Net                       Time          Slack
--------------------------------------------------------------------------------
m_alu.alu_out[3]      S_DFF     D       m_alu.alu_out_11_5[3]     97.8          83.2
m_alu.alu_out[2]      S_DFF     D       m_alu.alu_out_11_5[2]     97.8          83.5
m_alu.alu_out[0]      S_DFF     D       m_alu.alu_out_11_5[0]     97.8          84.4
m_alu.alu_out[7]      S_DFF     D       m_alu.alu_out_11_5[7]     97.8          84.9
m_alu.alu_out[6]      S_DFF     D       m_alu.alu_out_11_5[6]     97.8          85.2
m_alu.alu_out[5]      S_DFF     D       m_alu.alu_out_11_5[5]     97.8          85.5
================================================================================

A Critical Path with worst case slack = 83.2 ns:

Instance/Net                          Pin      Pin      Arrival     Delta     Fan
Name                        Type      Name     Dir      Time        Delay     Out
--------------------------------------------------------------------------------
m_accum.accum[1]            S_DFFE    Q        Out      3.0         3.0
m_accum.accum[1]            Net                                              6
m_alu.un2_alu_out_add1      S_CAR     I1       In       3.0
m_alu.un2_alu_out_add1      S_CAR     COUT     Out      4.2         1.2
m_alu.un2_alu_out_carry_1   Net                                              1
m_alu.un2_alu_out_add2      S_CAR     CIN      In       4.2
m_alu.un2_alu_out_add2      S_CAR     COUT     Out      4.5         0.3
m_alu.un2_alu_out_carry_2   Net                                              1
m_alu.un2_alu_out_add3      S_CAR     CIN      In       4.5
m_alu.un2_alu_out_add3      S_CAR     OUT      Out      6.7         2.2
m_alu.un2_alu_out_add3      Net                                              1
m_alu.alu_out_11_1[3]       S_LUT     I1       In       6.7
m_alu.alu_out_11_1[3]       S_LUT     OUT      Out      9.6         2.9
m_alu.alu_out_11_1[3]       Net                                              1
m_alu.alu_out_11_2[3]       S_LUT     I2       In       9.6
m_alu.alu_out_11_2[3]       S_LUT     OUT      Out      12.5        2.9
m_alu.alu_out_11_2[3]       Net                                              1
m_alu.alu_out_11_5[3]       S_LUT     I1       In       12.5
m_alu.alu_out_11_5[3]       S_LUT     OUT      Out      14.6        2.1
m_alu.alu_out_11_5[3]       Net                                              1
m_alu.alu_out[3]            S_DFF     D        In       14.6
================================================================================
Setup requirement on this path is 2.2 ns.


             Detailed Timing Report for  clock : m_machine.inc_pc
             ****************************************
Requested Period      100.0 ns
Estimated Period      10.5 ns
Worst Slack          89.5 ns
```

Start Points for Paths with Slack Worse than 91.8 ns :

Instance	Type	Pin	Net	Arrival Time	Slack
m_machine.load_pc	S_DFF	Q	m_machine.load_pc	1.0	89.5
m_counter.pc_addr[0]	S_DFF	Q	m_counter.pc_addr[0]	1.4	90.5
m_counter.pc_addr[1]	S_DFF	Q	m_counter.pc_addr[1]	1.4	90.8
m_register.opc_iraddr[0]	S_DFFE	Q	m_register.opc_iraddr[0]	1.8	91.0
m_register.opc_iraddr[1]	S_DFFE	Q	m_register.opc_iraddr[1]	1.8	91.0
m_register.opc_iraddr[2]	S_DFFE	Q	m_register.opc_iraddr[2]	1.8	91.0
m_register.opc_iraddr[3]	S_DFFE	Q	m_register.opc_iraddr[3]	1.8	91.0
m_register.opc_iraddr[4]	S_DFFE	Q	m_register.opc_iraddr[4]	1.8	91.0
m_register.opc_iraddr[5]	S_DFFE	Q	m_register.opc_iraddr[5]	1.8	91.0
m_register.opc_iraddr[6]	S_DFFE	Q	m_register.opc_iraddr[6]	1.8	91.0

End Points for Paths with Slack Worse than 91.8 ns :

Instance	Type	Pin	Net	Required Time	Slack
m_counter.pc_addr[0]	S_DFF	D	m_counter.pc_addr_lm0	97.8	89.5
m_counter.pc_addr[1]	S_DFF	D	m_counter.pc_addr_lm1	97.8	89.5
m_counter.pc_addr[2]	S_DFF	D	m_counter.pc_addr_lm2	97.8	89.5
m_counter.pc_addr[3]	S_DFF	D	m_counter.pc_addr_lm3	97.8	89.5
m_counter.pc_addr[4]	S_DFF	D	m_counter.pc_addr_lm4	97.8	89.5
m_counter.pc_addr[5]	S_DFF	D	m_counter.pc_addr_lm5	97.8	89.5
m_counter.pc_addr[6]	S_DFF	D	m_counter.pc_addr_lm6	97.8	89.5
m_counter.pc_addr[7]	S_DFF	D	m_counter.pc_addr_lm7	97.8	89.5
m_counter.pc_addr[8]	S_DFF	D	m_counter.pc_addr_lm8	97.8	89.5
m_counter.pc_addr[9]	S_DFF	D	m_counter.pc_addr_lm9	97.8	89.5

A Critical Path with worst case slack = 89.5 ns:

Instance/Net Name	Type	Pin Name	Pin Dir	Arrival Time	Delta Delay	Fan Out
m_machine.load_pc	S_DFF	Q	Out	1.0	1.0	
m_machine.load_pc	Net					1
m_machine.load_pc_i	S_LUT	I0	In	1.0		
m_machine.load_pc_i	S_LUT	OUT	Out	6.8	5.8	
m_machine.load_pc_i	Net					13
m_counter.pc_addr_lm0	S_MUX21	SEL	In	6.8		
m_counter.pc_addr_lm0	S_MUX21	Z	Out	8.3	1.5	
m_counter.pc_addr_lm0	Net					1
m_counter.pc_addr[0]	S_DFF	D	In	8.3		

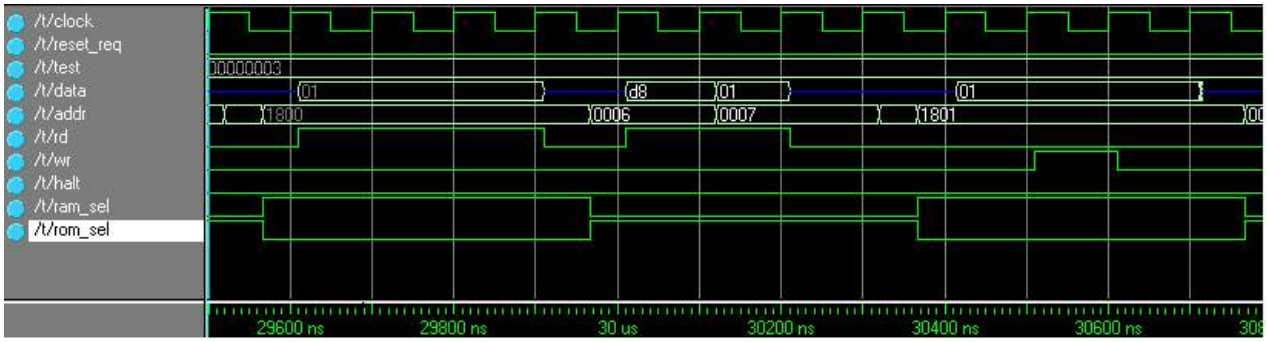
Setup requirement on this path is 2.2 ns.

```
                ##### END TIMING REPORT #####


---------------------------------------
Resource Usage Report

Synplify is performing all technology mapping
Post place and route resource use may vary a small
amount due to logic cell replication and register packing
decisions during place and route.

Design view:work.cpu(verilog)
Selecting part epf10k10lc84-3

Logic resources:  149 LCs of 576 (25%)
Number of Nets:   265
Number of Inputs: 926
Register bits:    69 (26 using enable)
I/O cells:       26

Details:
Cells in logic mode:   116
Cells in arith mode:   8
Cells in cascade mode: 10
Cells in counter mode: 13
DFFs with no logic:    2  (uses cell for routing)
LUTs driving both DFF and logic: 2


Found clock clk with period 100ns
Found clock alu_clk with period 100ns
Found clock fetch with period 100ns
Found clock inc_pc with period 100ns
Enabling timing driven placement for new ACF file.
All Constraints processed!
Mapper successful!
Process took 7.03 seconds realtime, 7.1 seconds cputime


//-------------------- RISC_CPU芯片综合结果报告结束----------------------------
```

8.5.3.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。在产生了XXXXX.edf文件之后，就要进行后仿真。以下介绍的是Altera Max+II 9.3 进行布线，在使用时在软件相应项选取所得到的cpu.edf文件，相对应的部件库（Altera FLEX10K）以及输出格式（verilog）。布线完成后得到两个文件cpu.vo和alt_max2.vo。cpu.vo是所设计的RISC_CPU的门级结构，即利用Verilog 语法描述的用alt_max2部件库中的基本元件构成的复杂电路连线网络，而alt_max2.vo是cpu.vo所引用的门级模型的库文件，包含各种基本类

型的电路的门级模型，它们的参数与真实器件完全一致，包括如延迟等参数。将这两个文件包含在cputop.v中，来代替原来的RTL级子模块，用仿真器再进行一次仿真，此时称为后仿真。实际上，后仿真与前仿真的根本区别在于测试文件所包含的模型的结构不同。前仿真使用的是一种RTL级模型，如cpu.v,而后仿真使用的是门级结构模型，其中不但有逻辑关系还包含实际门级电路和布线的延迟，还有驱动能力的问题。仔细观察后仿真波形就会发现与前仿真相比较，各信号的变化与时钟沿之间存在着延迟，这在前仿真时并未反映出来。



后仿真波形

下面的Verilog程序是由布局布线工具生成的，分别命名为cpu.vo和alt_max2.vo。由于cpu.vo是门级描述，共有上千行，而alt_max2.vo是仿真用库用UDP描述，也有几百行，无法在课本上全部列出，只能从中截取一小片段供同学参考。有兴趣的同学可以用 Verilog 语法中有关门级描述和用户自定义源语（UDP）来理解，由于是门级模型，又有布线的延迟，所以可以来验证电路结构是否符合设计要求。

```
/**********************cpu.vo开始*************************
// MAX+plus II Version 9.3 RC3 7/20/1999
// Sun Jul 30 10:53:36 2000

//
`timescale 100 ps / 100 ps


module cpu (
       addr,
       data,
       CLK,
       reset,
       halt,
       rd,
       wr);

output [12:0] addr;
inout [7:0] data;
input CLK;
input reset;
output halt;
output rd;
output wr;
supply0 gnd;
```

```
supply1 vcc;

wire
    \|accum:m_accum|accum_0_.CLK , \|accum:m_accum|accum_0_.D ,
\|accum:m_accum|accum_0_.ENA ,
    \|accum:m_accum|accum_0__Q , \|accum:m_accum|accum_1_.CLK ,
\|accum:m_accum|accum_1_.D ,
    …
    …
    …

    TRIBUF0_cpu TRIBUF_2
( .Y(data[0]), .IN1(N_126), .OE(\|datactl:m_datactl|data_0_.OE ) );
    TRIBUF0_cpu TRIBUF_4
( .Y(data[1]), .IN1(N_135), .OE(\|datactl:m_datactl|data_1_.OE ) );
    TRIBUF0_cpu TRIBUF_6
( .Y(data[2]), .IN1(N_144), .OE(\|datactl:m_datactl|data_2_.OE ) );
    TRIBUF0_cpu TRIBUF_8
( .Y(data[3]), .IN1(N_153), .OE(\|datactl:m_datactl|data_3_.OE ) );
    …
    …
AND1 AND1_49 ( \|datactl:m_datactl|data_0_.OE , N_124 );
DELAY DELAY_50 ( N_124, \|machine:m_machine|datactl_ena1_Q  );
defparam DELAY_50.TPD = 40;
DELAY DELAY_51 ( N_126, N_127 );
XOR2 XOR2_52 ( N_127, N_128, N_132 );
....

module DFF0_cpu ( Q, D, CLK, CLRN, PRN );
    input D;
    input CLK;
    input CLRN;
    input PRN;
    output Q;
    PRIM_DFF (Q, D, CLK, CLRN, PRN);

    wire legal;
    and(legal, CLRN, PRN);
    specify

        specparam TREG = 9;
        specparam TRSU = 13;
        specparam TRH  = 14;
        specparam TRPR = 10;
        specparam TRCL = 10;

        $setup  (  D, posedge CLK &&& legal, TRSU  ) ;
        $hold   (  posedge CLK &&& legal, D, TRH   ) ;

        ( negedge CLRN => (Q  +: 1'b0)) = ( TRCL, TRCL ) ;
        ( negedge PRN  => (Q  +: 1'b1)) = ( TRPR, TRPR ) ;
        ( posedge CLK  => (Q  +: D)) = ( TREG, TREG ) ;
```

```
    endspecify
endmodule
…
…
/******************cpu.vo结束********************

/*****************alt_max2.vo开始****************
//
// MAX+plus II Version 9.3 RC3 7/20/1999
// Sun Jul 30 10:53:36 2000

//

//`define SDF_IOPATH
`timescale 100 ps / 100 ps

primitive PRIM_DFF  (Q, D, CP, RB, SB);

    output Q;
    input  D, CP, RB, SB;
    reg    Q;

    initial Q = 1'b0;

    // FUNCTION : POSITIVE EDGE TRIGGERED D FLIP-FLOP WITH ACTIVE LOW
    //            ASYNCHRONOUS SET AND CLEAR. ( Q OUTPUT UDP ).

    table

    //  D   CP      RB  SB  :   Qt  :   Qt+1

        1   (01)    1   1   :   ?   :   1;  // clocked data
        1   (01)    1   x   :   ?   :   1;  // pessimism

        1    ?      1   x   :   1   :   1;  // pessimism

        0    0      1   x   :   1   :   1;  // pessimism
        0    x      1 (?x)  :   1   :   1;  // pessimism
        0    1      1 (?x)  :   1   :   1;  // pessimism
…….

primitive PRIM_LATCH  (Q, ENA, D);
    input  D;
    input  ENA;
    output Q; reg Q;

    table

    // ENA    D    Q   Q+
         0    ?  : ? : -;
         1    0  : ? : 0;
```

```
        1    1  : ? : 1;  //

    endtable

endprimitive
……

`celldefine
module AND1 ( Y, IN1 );
    parameter TPD = 0;
    input IN1;
    output Y;

    and #TPD (Y, IN1);


`ifdef SDF_IOPATH
    specify
        (IN1 => Y) = (0,0);
    endspecify
`endif

endmodule
…
/**********************alt_max2.vo结束******************
```

不同FPGA厂家的布局布线工具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，读者应阅读FPGA厂家的布局布线工具的说明书中有关章节，选用正确的Verilog门级结构的后仿真解决方案。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，可先降低测试信号模块的主时钟频率，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VerilogHDL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VerilogHDL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。

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。
4） 什么叫软硬件联合仿真?为什么说Verilog语言支持软硬件联合设计?

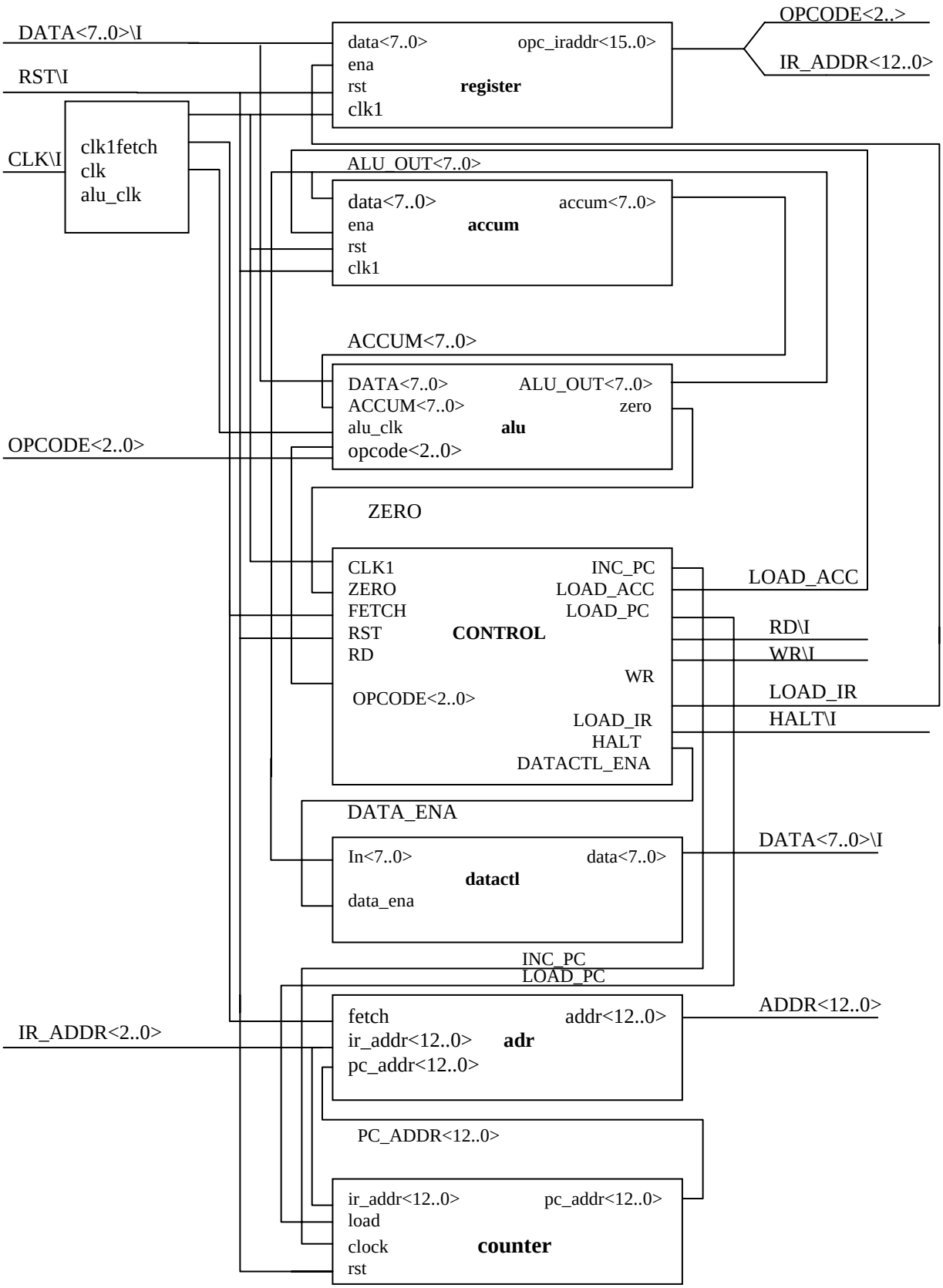


图8。1 RISC——CPU中各部件的相互连接关系

第九章　虚拟器件和虚拟接口模型

以及它们在大型数字系统设计中的作用

前言

宏单元(Macrocells 或 Megacells)或核(Cores)是预先设计好的，其功能经过验证的、由总数超过 5000 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。这就是我们在第一章的 1.5.3 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（也称接口模型）的重用技术可大大缩短设计周期，加快高技术新芯片的投产和上市 。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。

在美国和电子工业先进的国家，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 芯片和 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，也为千万门级单片系统的实现铺平了道路。

9.1 虚拟器件和虚拟接口模块的供应商

在这一节中我们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：

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
ARM Semiconductor 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） 三种都可提供

9.2 虚拟模块的设计

我国大陆地区由于复杂芯片的设计工作开展较晚，经费也比较少，目前许多单位有还不能及时得到商业化的虚拟模块和接口，因此就有必要自己来设计虚拟接口模型。下面的例子说明了怎样根据数据手册和波形图来编写虚拟的接口模型。

[例 1]. 模数转换器 AD7886 仿真模型（虚拟模块）的设计：

下面介绍的名为 ADC 的 Verilog 模块在设计中可以用来模拟实际的模数转换器（下面简称 A/D）AD7886。因此，该仿真模型的输入与出各输出信号间的逻辑关系，必须严格按照数据手册描述的波形编写，信号间的时间关系也必须完全符合手册要求，这样才能起到虚拟模块的作用。只有这样在设计电路的测试中才能用它来代替实际器件。同时，虚拟模块还应具备实际电路所没有的功能： 如对于不符合要求的输入信号还能产生错误提示。 在实际的电路中，我们很难控制 A/D 的输出数据，然而在该设计中，我们可以编写数据文件，得到我们想要的各种类型的数据， 来测试后续电路的功能，并可以随时根据测试要求更改数据，非常方便。虚拟模块的编写是 Verilog 语言应用的重要方面。它为 ASIC 设计投片一次成功提供了可能。

在实际电路中，A/D 包括模拟部分，和许多必要的控制和参考电平输入，而在这里为了简单和说明问题起见，只介绍 A/D 模块有关数字接口的一部分功能，把这部分功能编写成虚拟模块。其中只包括了 A/D 控制信号的输入、数据总线和“忙”信号的输出。为了进一步简化还假设选片信号 $\overline{CS}$ 和读信号 $\overline{RD}$ 总是为低电平（有效）。因此，该模型实质上是为教学目的而编写的简化虚拟模块，在仿真时仅能代替真实 A/D 的一部分功能。它类似一个数据发生器，根据输入控制信号和 A/D 自身的特性输出一个字节（8 位）数据和“忙”信号。同时根据手册规定，不断检测输入信号是否符合要求。虚拟模型的精确与否，直接影响到设计是否能够一次投片成功。因此在 ASIC 系统芯片的设计中应予以充分的重视。

AD7886 是具有一个高速的 8 位三态数据输出接口的模数转换器，转换的过程由输入信号 $\overline{CONVST}$ 控制,数据的存取由选片信号 $\overline{CS}$ 和读信号 $\overline{RD}$ 输入信号控制(低电平有效)。下面的 Verilog 源代码描述了该 A/D 的转换启动和数据读出功能，（假设 $\overline{CS}$ 和 $\overline{RD}$ 都为低电平）根据手册的说明，输入和输出波形如下图所示：

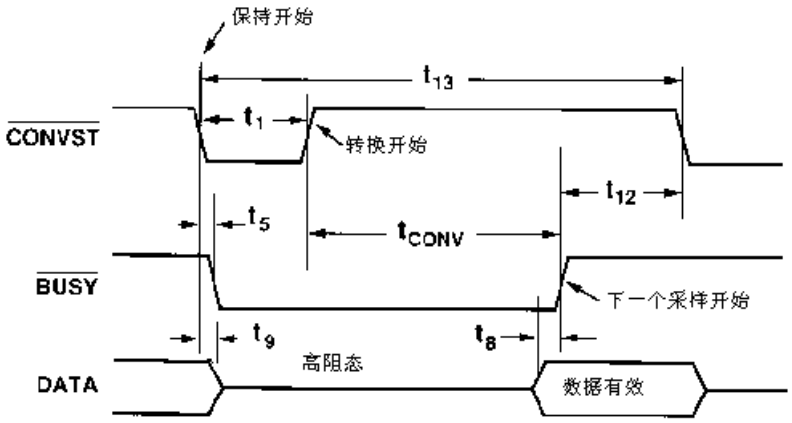


图 1　A/D 转换启动和数据读出时序（$\overline{CS}$ = $\overline{RD}$ =0）

为使所设计的虚拟模块对输入信号有检测功能，还在模块中加入了提示输入信号有错的语句。输出的 8 位数据可以根据要求自己编制，从数据文件 AD.DATA 中读取。下面是一个名为 ADC.V 的文件，描述了该 A/D 转换器波形所示的这一部分功能。其仿真模块的具体源代码如下：

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
`timescale 1ns/100ps
module   adc (nconvst, nbusy, data);
input     nconvst;          // A/D 启动脉冲 ST，即上图中
output    nbusy;        // A/D 工作标志，即上图中
output       data;          // 数据总线，从 AD.DATA 文件中读取数据后经端口输出
reg[7:0]  databuf,i;   // 内部寄存器
reg       nbusy;
wire[7:0] data;
reg[7:0]  data_mem[0:255];
reg       link_bus;
integer   tconv,
          t5,
          t8,
          t9,
          t12;
integer   wideth1,
          wideth2,
          wideth;
//时间参数定义(依据 AD7886 手册)：
always @(negedge nconvst)
   begin
         tconv =9500+{$random}%500; //(type 950, max 1000) Conversion Time
         t5 ={$random}%1000;  //(max 100)   CONVST to BUSY Propagation Dlay
                              // CL = 10pf
         t8 = 200;      //(min 20)  CL=20pf  Data Setup Time Prior to BUSY
                       //(min 10)  CL=100pf
         t9 = 100+{$random}%900; //(min 10, max 100) Bus Relinquish Time After
                                 //CONVST
         t12 = 2500;   //(type) BUSY High to CONVST Low, SHA Acquisition Time
   end

initial
      begin
       $readmemh("adc.data",data_mem);  //从数据文件 adc.data 中读取数据
       i = 0;
       nbusy  = 1;
           link_bus = 0;
      end
```

```
assign data = link_bus? databuf:8'bzz; //三态总线

/*-------------------------------------------------------------------------
在信号 nconvst 的负跳降沿到来后，隔 t5 秒 nbusy 信号置为低，tconv 是 AD 将模拟信号转换为数字信号的时间，在信号 nconvst 的正跳降沿到来后经过 tconv 时间后，输出 nbusy 信号变为高。
-------------------------------------*/
always @(negedge nconvst)
        fork
         #t5  nbusy =0;
           @(posedge nconvst)
                  begin
                       #tconv  nbusy=1;
                  end
        join
/*-------------------------------------------------------------------
nconvst 信号的下降沿触发，经过 t9 延时后，把数据总线输出关闭置为高阻态，如图示。
 nconvst 信号的上升沿到来后,经过(tconv－t8)时间,输出一个字节(8 位数据)到 databuf，该数据来自于 data_mem。而 data_mem 中的数据是初始化时从数据文件 AD.DATA 中读取的。此时应启动总线的三态输出。
-------------------------------------------------------------------------*/
always @(negedge nconvst)
      begin
         @(posedge nconvst)
            begin
              #(tconv-t8)    databuf=data_mem[i];
            end

         if(wideth <10000  && wideth>500)
            begin
              if(i==255) i=0;
              else i=i+1;
            end
         else  i = i;
      end
//在模数转换期间关闭三态输出，转换结束时启动三态输出
always @(negedge nconvst)
  fork
       #t9 link_bus = 1'b0;     //关闭三态输出，不允许总线输出
       @(posedge nconvst)
            begin
            #(tconv-t8)    link_bus=1'b1;
            end
  join
/*-------------------------------------------------------------------
```

当 nconvst 输入信号的下一个转换的下降沿与 nbusy 信号上升沿之间时间延迟小于 t12 时，将会出现警告信息，通知设计者请求转换的输入信号频率太快，A/D 器件转换速度跟不上。仿真模型不仅能够实现硬件电路的输出功能，同时能够对输入信号进行检测，
当输入信号不符合手册要求时，显示警告信息。
----------*/

```
// 检查 A/D 启动信号的频率是否太快
     always @(posedge nbusy)
       begin
          #t12;
          if (!nconvst)
             begin
               $display("Warning!  SHA Acquisition Time is too short!");
             end
        //  else  $display(" SHA Acquisition Time is enough! ");
       end
// 检查 A/D 启动信号的负脉冲宽度是否足够和太宽

   always @(negedge nconvst)
     begin
          wideth=$time;
          @(posedge nconvst)  wideth=$time-wideth;
          if (wideth<=500 || wideth > 10000)
            begin

             $display("nCONVST Pulse Width = %d",wideth);
             $display("Warning! nCONVST Pulse Width is too narrow or too wide!");
             //$stop;
            end
    end

endmodule


//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

对商业化的虚拟模块有着严格的要求，不但要求在系统设计的仿真中能完全来代替真实的器件，而且还希望能提示产生错误的原因。**虚拟模块的精确与否，直接决定设计的成败。** ASIC 的投片成本很高，编写虚拟模块时任何小的疏忽都有可能造成投片的失败，造成大量资金的浪费。**因此编写这样的模块是一件复杂而细致的工作，需要极其认真的工作态度和作风，**必须认真对待。为了简单起见，本节介绍的模块只具有 AD7886 的一部分功能，所以还不能称为 AD7886 完整的虚拟模块。

通过上述简单的例子我们能了解一个虚拟模块是如何设计的，对大多数的电路系统工程师来说，我们尽量利用商业化的虚拟模块来设计自己的电路系统。**只有在没有办法得到商业化的**

虚拟模块时，才利用器件手册来编写虚拟模块，因为编写精确的虚拟模块需要花费很多时间和精力。

9.3 虚拟接口模块的实例

下面我们介绍两个常用的大规模集成芯片：通用串行收发控制器 USART8251 和 Intel8085 微处理器 CPU 的虚拟接口模块。这两个用 Verilog HDL 描述的虚拟接口的行为模块是由 Verilog 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是知识产权（简称IP），必须保证设计所需的参数绝对正确，因此价格非常昂贵，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，因为它们并不对用户设计的成败负责。**把它们列在这里只是拿它们作为学习编写较复杂的Verilog HDL行为模块的样本而已。

[例 1].“商业化”的虚拟模块之一：Intel USART 8251A (通用串行异步收发器芯片)

在 8251A 虚拟接口模块程序的原始材料上有下面这样一段话：

```
/*****************************************************************************
CADENCE DESIGN SYSTEMS, Inc. does not guarantee the accuracy or completeness of
this model. Anyone who using this does so at their own risk.
*****************************************************************************/
```

显然这一模块只是一个仅供参考的实例，在本教材上我们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：

```
/*****************************************************************************
                通用串行异步收发器 8251 的 Verilog HDL 源代码
          注意: 作者不能保证本模块的完整和精确，使用本模块者如遇问题一切责任自负
*****************************************************************************/

module I8251A ( dbus, rcd, gnd, txc_, write_, chipsel_, comdat_,
                        read_, rxrdy, txrdy, syndet, cts_, txe, txd,
                                clk, reset, dsr_,rts_,dtr_,rxc_,vcc);

/* 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]       receivebuf, rdata, status;

//*****ADD BY FWN
reg [3:0] dflags;
reg [7:0] instance_id;
reg read,chipel_;
//*****

reg     recvdrv,statusdrv;

// if recvdrv 1 dbus is driven by rdata
assign  dbus = recvdrv ? rdata : 8'bz; //*****:,->;
assign   dbus = statusdrv ? status : 8'bz ; //*****:->; assign abscent
```

```
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)))
            $display("I8251A (%h) at %d: *** datahold violation on write",
                                                    instance_id ,$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);// lack of ;

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/override ***** only
                                    // candence synerngy support assign,deassign
                else             // later non-assign assignment
                    deassign 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 syn_receive_internal;
                             // disasble the sync receiver
                        disable syn_receive_external;

                        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 ransmitted ...
                ->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 *****add ()
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'b00:
        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
                                            // begore 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
            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 syn_receive_external;
forever
begin
// have not found the original programs
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
```

[例 2]. “商业化”的虚拟模块之二: Intel 8085a 微处理器的行为描述模块

```
/*******************************************************************************
                Intel 8085a 微处理器仿真模块的 Verilog 源代码
          注意: 作者不能保证本模块的完整和精确, 使用本模块者如遇问题一切责任自负
*******************************************************************************/

module intel_8085a
        (clock, x2, resetff, sodff, sid, trap, rst7p5, rst6p5, rst5p5,
         intr, intaff, ad, a, s0, aleff, writeout, readout, s1,iomout,
         ready, nreset, clockff, hldaff, hold);

    reg [8:1]      dflags;
    initial         dflags = 'b011;
    // diag flags:
    // 1 = trace instructions
    // 2 = trace IN and OUT instructions
    // 3 = trace instruction count

    output
        resetff, sodff, intaff, s0, aleff,
        writeout, readout, s1, iomout, clockff, hldaff;

    inout[7:0] ad, a;
```

```
input
        clock, x2, sid, trap,
        rst7p5, rst6p5, rst5p5,
        intr, ready, nreset, hold;

reg[15:0]
    pc,         // program counter
    sp,         // stack pointer
    addr;       // address output

reg[8:0]
    intmask;    // interrupt mask and status

reg[7:0]
    acc,        // accumulator
    regb,       // general
    regc,       // general
    regd,       // general
    rege,       // general
    regh,       // general
    regl,       // general
    ir,         // instruction
    data;       // data output

reg
    aleff,      // address latch enable
    s0ff,       // status line 0
    s1ff,       // status line 1
    hldaff,     // hold acknowledge
    holdff,     // internal hold
    intaff,     // interrupt acknowledge
    trapff,     // trap interrupt request
    trapi,      // trap execution for RIM instruction
    inte,       // previous state of interrupt enable flag
    int,        // interrupt acknowledge in progress
    validint,   // interrupt pending
    haltff,     // halt request
    resetff,    // reset output
    clockff,    // clock output
    sodff,      // serial output data
    read,       // read request signal
    write,      // write request signal
    iomff,      // i/o memory select
    acontrol,   // address output control
```

```
    dcontrol,  // data output control
    s,         // data source control
    cs,        // sign condition code
    cz,        // zero condition code
    cac,       // aux carry condition code
    cp,        // parity condition code
    cc;        // carry condition code

wire
    s0 = s0ff & ~haltff,
    s1 = s1ff & ~haltff;

tri[7:0]
    ad = dcontrol ? (s ? data : addr[7:0]) : 'bz,
    a = acontrol ? addr[15:8] : 'bz;

tri
    readout = acontrol ? read : 'bz,
    writeout = acontrol ? write : 'bz,
    iomout = acontrol ? iomff : 'bz;

event
    ec1, // clock 1 event
    ec2; // clock 2 event

// internal clock generation
always begin
    @(posedge clock) -> ec1;
    @(posedge clock) -> ec2;
end

integer instruction; // instruction count
initial instruction = 0;

always begin:run_processor
    #1 reset_sequence;
    fork
        execute_instructions;             // Instructions executed
        wait(!nreset)                     // in parallel with reset
            @ec2 disable run_processor; // control. Reset will
    join                                  // disable run_processor
end                                       // and all tasks and
                                          // functions enabled from
                                          // it when nreset set to 0.
```

```
task reset_sequence;
begin
    wait(!nreset)
    fork
        begin
            $display("Performing 8085(%m) reset sequence");
            read = 1;
            write = 1;
            resetff = 1;
            dcontrol = 0;
            @ec1 // synchronized with clock 1 event
                pc = 0;
                ir = 0;
                intmask[3:0] = 7;
                intaff = 1;
                acontrol = 0;
                aleff = 0;
                intmask[7:5] = 0;
                sodff = 0;
                trapff = 0;
                trapi = 0;
                iomff = 0;
                haltff = 0;
                holdff = 0;
                hldaff = 0;
                validint = 0;
                int = 0;
            disable check_reset;
        end
        begin:check_reset
            wait(nreset)               // Check, in parallel with the
                disable run_processor; // reset sequence, that nreset
        end                            // remains at 0.
    join
    wait(nreset) @ec1 @ec2 resetff = 0;
end
endtask


/* fetch and execute instructions */
task execute_instructions;
forever begin
    instruction = instruction + 1;
    if(dflags[3])
        $display("executing instruction %d", instruction);
```

```
@ec1 // clock cycle 1
    addr = pc;
    s = 0;
    iomff = 0;
    read = 1;
    write = 1;
    acontrol = 1;
    dcontrol = 1;
    aleff = 1;
    if(haltff) begin
        haltff = 1;
        s0ff = 0;
        s1ff = 0;
        haltreq;
    end
    else begin
        s0ff = 1;
        s1ff = 1;
    end
@ec2
    aleff = 0;

@ec1 // clock cycle 2
    read = 0;
    dcontrol = 0;
@ec2
    ready_hold;

@ec2 // clock cycle 3
    read = 1;
    data = ad;
    ir = ad;

@ec1 // clock cycle 4
    if(do6cycles(ir)) begin
        // do a 6-cycle instruction fetch
        @ec1 @ec2 // conditional clock cycle 5
            if(hold) begin
                holdff =1 ;
                acontrol = 0;
                dcontrol = 0;
                @ec2 hldaff = 1;
            end
            else begin
```

```
                    holdff = 0;
                    hldaff = 0;
                end

            @ec1; // conditional clock cycle 6
        end

        if(holdff) holdit;
        checkint;
        do_instruction;

        while(hold) @ec2 begin
            acontrol = 0;
            dcontrol = 0;
        end
        holdff = 0;
        hldaff = 0;
        if(validint) interrupt;
end
endtask


function do6cycles;
input[7:0] ireg;
begin
    do6cycles = 0;
    case(ireg[2:0])
        0, 4, 5, 7: if(ireg[7:6] == 3) do6cycles = 1;
        1: if((ireg[3] == 1) && (ireg[7:5] == 7)) do6cycles = 1;
        3: if(ireg[7:6] == 0) do6cycles = 1;
    endcase
end
endfunction


task checkint;
begin
    if(rst6p5)
        if((intmask[3] == 1) && (intmask[1] == 0)) intmask[6] = 1;
    else
        intmask[6] = 0;

    if(rst5p5)
        if((intmask[3] == 1) && (intmask[0] == 0)) intmask[5] = 1;
    else
```

```
        intmask[5] = 0;

    if({intmask[7], intmask[3:2]} == 6)
        intmask[4] = 1;
    else
        intmask[4] = 0;

    validint = (intmask[6:4] == 7) | trapff | intr;
end
endtask


// concurently with executing instructions,
// process primary inputs for processor interrupt
always @(posedge trap) trapff = 1;

always @(negedge trap) trapff = 0;

always @(posedge rst7p5) intmask[7] = 1;


/* check condition of ready and hold inputs */
task ready_hold;
begin
    while(!ready) @ec2;
    @ec1
        if(hold) begin
            holdff = 1;
            @ec2 hldaff = 1;
        end
end
endtask


/* hold */
task holdit;
begin
    while(hold) @ec2 begin
        acontrol = 0;
        dcontrol = 0;
    end
    holdff = 0;
    @ec2 hldaff = 0;
end
endtask
```

```
/* halt request */
task haltreq;
forever begin
    @ec2
        if(validint) begin
            haltff = 0;
            interrupt;
            disable haltreq;
        end
        else begin
            while(hold) @ec2 hldaff = 1;
            hldaff = 0;
            @ec2;
        end

    @ec1 #10
        dcontrol = 0;
        acontrol = 0;
        checkint;
end
endtask

/* memory read */
task memread;
output[7:0] rdata;
input[15:0] raddr;
begin
    @ec1
        addr = raddr;
        s = 0;
        acontrol = 1;
        dcontrol = 1;
        iomff = int;
        s0ff = int;
        s1ff = 1;
        aleff = 1;
    @ec2
        aleff = 0;

    @ec1
        dcontrol = 0;
        if(int)
```

```
            intaff = 0;
        else
            read = 0;
    @ec2
        ready_hold;
        checkint;

    @ec2
        intaff = 1;
        read = 1;
        rdata = ad;
    if(holdff) holdit;
end
endtask

/* memory write */
task memwrite;
input[7:0] wdata;
input[15:0] waddr;
begin
    @ec1
        aleff = 1;
        s0ff = 1;
        s1ff = 0;
        s = 0;
        iomff = 0;
        addr = waddr;
        acontrol = 1;
        dcontrol = 1;
    @ec2
        aleff = 0;

    @ec1
        data = wdata;
        write = 0;
        s = 1;
    @ec2
        ready_hold;
        checkint;

    @ec2
        write = 1;
    if(holdff) holdit;
end
endtask
```

```
/* reads from an i/o port */
task ioread;
input[7:0] sa;
begin
    @ec1
        aleff = 1;
        s0ff = 0;
        s1ff = 1;
        s = 0;
        iomff = 1;
        addr = {sa, sa};
        acontrol = 1;
        dcontrol = 1;

    @ec2
        aleff = 0;

    @ec1
        dcontrol = 0;
        if(int)
            intaff = 0;
        else
            read = 0;

    @ec2
        ready_hold;

        checkint;

    @ec2
        intaff = 1;
        read = 1;
        acc = ad;
        if(dflags[2])
            $display("IN %h   data = %h", sa, acc);
end
endtask

/* writes into i/o port */
task iowrite;
input[7:0] sa;
begin
    @ec1
        addr = {sa, sa};
```

```
            aleff = 1;
            s0ff = 1;
            s1ff = 0;
            s = 0;
            iomff = 1;
            acontrol = 1;
            dcontrol = 1;

        @ec2
            aleff = 0;

        @ec1
            data = acc;
            write = 0;
            s = 1;

            if(dflags[2])
                $display("OUT %h   data = %h", sa, acc);

        @ec2
            ready_hold;

            checkint;

        @ec2
            write = 1;
            if(holdff) holdit;
    end
    endtask

    task interrupt;
    begin
        @ec1
            if(hold) begin
                holdff = 1;
                holdit;
                @ec2 hldaff = 1;
            end
            if(trapff) begin
                inte = intmask[3];
                trapi = 1;
                intic;
                pc = 'h24;
                trapi = 1;
                trapff = 0;
```

```
        end
        else if(intmask[7]) begin
            intic;
            pc = 'h3c;
            intmask[7] = 0;
        end
        else if(intmask[6]) begin
            intic;
            pc = 'h34;
            intmask[6] = 0;
        end
        else if(intmask[5]) begin
            intic;
            pc = 'h2c;
            intmask[5] = 0;
        end
        else if(intr) begin
            //?
        end
    end
    endtask

    task intic;
    begin
        aleff = 1;
        s0ff = 1;
        s1ff = 1;
        s = 0;
        iomff = 1;
        addr = pc;
        read = 1;
        write = 1;
        acontrol = 1;
        dcontrol = 1;

        @ec2 aleff = 0;
        @ec1 dcontrol = 0;
        repeat(4) @ec1;
        push2b(pc[15:8], pc[7:0]);
    end
    endtask


    /* execute instruction */
    task do_instruction;
```

```
begin
  if(dflags[1])
    $display( "C%bZ%bM%bE%bI%b A=%h B=%h%h D=%h%h H=%h%h S=%h P=%h IR=%h",
              cc, cz, cs, cp, cac, acc, regb,regc, regd,rege, regh,regl,
              sp, pc, ir);

    pc = pc + 1;
    @ec2 // instruction decode synchronized with clock 2 event
        case(ir[7:6])
            0:
                case(ir[2:0])
                    0: newops;
                    1: if(ir[3]) addhl; else lrpi;
                    2: sta_lda;
                    3: inx_dcx;
                    4: inr;
                    5: dcr;
                    6: movi;
                    7: racc_spec;
                endcase
            1:
                move;
            2:
                rmop;
            3:
                case(ir[2:0])
                    0,
                    2,
                    4: condjcr;
                    1: if(ir[3]) decode1; else pop;
                    3: decode2;
                    5: if(ir[3]) decode3; else push;
                    6: immacc;
                    7: restart;
                endcase
        endcase
end
endtask

/* move register to register */
task move;
    case(ir[2:0])
        0: rmov(regb); // MOV -,B
        1: rmov(regc); // MOV -,C
        2: rmov(regd); // MOV -,D
```

```
            3: rmov(rege); // MOV -,E
            4: rmov(regh); // MOV -,H
            5: rmov(regl); // MOV -,L
            6:
                if(ir[5:3] == 6) haltff = 1; // HLT
                else begin // MOV -,M
                    memread(data, {regh, regl});
                    rmov(data);
                end

            7: rmov(acc); // MOV -,A
        endcase
    endtask

    /* enabled only by move */
    task rmov;
    input[7:0] fromreg;
        case(ir[5:3])
            0: regb = fromreg; // MOV B,-
            1: regc = fromreg; // MOV C,-
            2: regd = fromreg; // MOV D,-
            3: rege = fromreg; // MOV E,-
            4: regh = fromreg; // MOV H,-
            5: regl = fromreg; // MOV L,-
            6: memwrite(fromreg, {regh, regl}); // MOV M,-
            7: acc = fromreg; // MOV A,-
        endcase
    endtask

    /* move register and memory immediate */
    task movi;
    begin
        case(ir[5:3])
            0: memread(regb, pc); // MVI B
            1: memread(regc, pc); // MVI C
            2: memread(regd, pc); // MVI D
            3: memread(rege, pc); // MVI E
            4: memread(regh, pc); // MVI H
            5: memread(regl, pc); // MVI L
            6: // MVI M
                begin
                    memread(data, pc);
                    memwrite(data, {regh, regl});
                end
```

```
        7: memread(acc, pc); // MVI A
    endcase
    pc = pc + 1;
end
endtask

/* increment register and memory contents */
task inr;
    case(ir[5:3])
        0: doinc(regb); // INR B
        1: doinc(regc); // INR C
        2: doinc(regd); // INR D
        3: doinc(rege); // INR E
        4: doinc(regh); // INR H
        5: doinc(regl); // INR L
        6: // INR M
            begin
                memread(data, {regh, regl});
                doinc(data);
                memwrite(data, {regh, regl});
            end

        7: doinc(acc); // INR A
    endcase
endtask

/* enabled only from incrm */
task doinc;
inout[7:0] sr;
begin
    cac = sr[3:0] == 'b1111;
    sr = sr + 1;
    calpsz(sr);
end
endtask

/* decrement register and memory contents */
task dcr;
    case(ir[5:3])
        0: dodec(regb); // DCR B
        1: dodec(regc); // DCR C
        2: dodec(regd); // DCR D
        3: dodec(rege); // DCR E
        4: dodec(regh); // DCR H
        5: dodec(regl); // DCR L
```

```
        6: // DCR M
            begin
                memread(data, {regh, regl});
                dodec(data);
                memwrite(data, {regh, regl});
            end

        7: dodec(acc); // DCR A
    endcase
endtask

/* enabled only from decrm */
task dodec;
inout[7:0] sr;
begin
    cac = sr[3:0] == 0;
    sr = sr - 1;
    calpsz(sr);
end
endtask

/* register and memory acc instructions */
task rmop;
    case(ir[2:0])
        0: doacci(regb);
        1: doacci(regc);
        2: doacci(regd);
        3: doacci(rege);
        4: doacci(regh);
        5: doacci(regl);
        6:
            begin
                memread(data, {regh, regl});
                doacci(data);
            end

        7: doacci(acc);
    endcase
endtask


/* immediate acc instructions */
task immacc;
begin
    memread(data, pc);
```

```
        pc = pc + 1;
        doacci(data);
    end
    endtask


    /* operate on accumulator */
    task doacci;
    input[7:0] sr;
    reg[3:0] null4;
    reg[7:0] null8;
        case(ir[5:3])
            0: // ADD ADI
                begin
                    {cac, null4} = acc + sr;
                    {cc, acc} = {1'b0, acc} + sr;
                    calpsz(acc);
                end

            1: // ADC ACI
                begin
                    {cac, null4} = acc + sr + cc;
                    {cc, acc} = {1'b0, acc} + sr + cc;
                    calpsz(acc);
                end

            2: // SUB SUI
                begin
                    {cac, null4} = acc - sr;
                    {cc, acc} = {1'b0, acc} - sr;
                    calpsz(acc);
                end

            3: // SBB SBI
                begin
                    {cac, null4} = acc - sr - cc;
                    {cc, acc} = {1'b0, acc} - sr - cc;
                    calpsz(acc);
                end

            4: // ANA ANI
                begin
                    acc = acc & sr;
                    cac = 1;
                    cc = 0;
```

```
                calpsz(acc);
            end

        5: // XRA XRI
            begin
                acc = acc ^ sr;
                cac = 0;
                cc = 0;
                calpsz(acc);
            end

        6: // ORA ORI
            begin
                acc = acc | sr;
                cac = 0;
                cc = 0;
                calpsz(acc);
            end

        7: // CMP CPI
            begin
                {cac, null4} = acc - sr;
                {cc, null8} = {1'b0, acc} - sr;
                calpsz(null8);
            end
    endcase
endtask

/* rotate acc and special instructions */
task racc_spec;
    case(ir[5:3])
        0: // RLC
            begin
                acc = {acc[6:0], acc[7]};
                cc = acc[7];
            end

        1: // RRC
            begin
                acc = {acc[0], acc[7:1]};
                cc = acc[0];
            end

        2: // RAL
            {cc, acc} = {acc, cc};
```

```
        3: // RAR
            {acc, cc} = {cc, acc};

        4: // DAA, decimal adjust
            begin
                if((acc[3:0] > 9) || cac) acc = acc + 6;
                if((acc[7:4] > 9) || cc) {cc, acc} = {1'b0, acc} + 'h60;
            end

        5: // CMA
            acc = ~acc;

        6: // STC
            cc = 1;

        7: // CMC
            cc = ~cc;
    endcase
endtask


/* increment and decrement register pair */
task inx_dcx;
    case(ir[5:3])
        0: {regb, regc} = {regb, regc} + 1; // INX B
        1: {regb, regc} = {regb, regc} - 1; // DCX B
        2: {regd, rege} = {regd, rege} + 1; // INX D
        3: {regd, rege} = {regd, rege} - 1; // DCX D
        4: {regh, regl} = {regh, regl} + 1; // INX H
        5: {regh, regl} = {regh, regl} - 1; // DCX H
        6: sp = sp + 1;                     // INX SP
        7: sp = sp - 1;                     // DCX SP
    endcase
endtask

/* load register pair immediate */
task lrpi;
    case(ir[5:4])
        0: adread({regb, regc}); // LXI B
        1: adread({regd, rege}); // LXI D
        2: adread({regh, regl}); // LXI H
        3: adread(sp);           // LXI SP
    endcase
endtask
```

```
/* add into regh, regl pair */
task addhl;
begin
    case(ir[5:4])
        0: {cc, regh, regl} = {1'b0, regh, regl} + {regb, regc}; // DAD B
        1: {cc, regh, regl} = {1'b0, regh, regl} + {regd, rege}; // DAD D
        2: {cc, regh, regl} = {1'b0, regh, regl} + {regh, regl}; // DAD H
        3: {cc, regh, regl} = {1'b0, regh, regl} + sp;           // DAD SP
    endcase
    holdreq;
    holdreq;
end
endtask


/* store and load instruction */
task sta_lda;
reg[15:0] ra;
    case(ir[5:3])
        0: memwrite(acc, {regb, regc}); // STAX B
        1: memread(acc, {regb, regc});  // LDAX B
        2: memwrite(acc, {regd, rege}); // STAX D
        3: memread(acc, {regd, rege});  // LDAX D

        4: // SHLD
            begin
                adread(ra);
                memwrite(regl, ra);
                memwrite(regh, ra + 1);
            end
        5: // LHLD
            begin
                adread(ra);
                memread(regl, ra);
                memread(regh, ra + 1);
            end

        6: // STA
            begin
                adread(ra);
                memwrite(acc, ra);
            end
        7: // LDA
```

```
            begin
                adread(ra);
                memread(acc, ra);
            end
    endcase
endtask

/* push register pair from stack */
task push;
    case(ir[5:4])
        0: push2b(regb, regc); // PUSH B
        1: push2b(regd, rege); // PUSH D
        2: push2b(regh, regl); // PUSH H
        3: push2b(acc, {cs,cz,1'b1,cac,1'b1,cp,1'b1,cc}); // PUSH PSW
    endcase
endtask

/* push 2 bytes onto stack */
task push2b;
input[7:0] highb, lowb;
begin
    sp = sp - 1;
    memwrite(highb, sp);
    sp = sp - 1;
    memwrite(lowb, sp);
end
endtask


/* pop register pair from stack */
task pop;
reg null1;
    case(ir[5:4])
        0: pop2b(regb, regc); // POP B
        1: pop2b(regd, rege); // POP D
        2: pop2b(regh, regl); // POP H
        3: pop2b(acc,
            {cs, cz, null1, cac, null1, cp, null1, cc}); // POP PSW
    endcase
endtask

/* pop 2 bytes from stack */
task pop2b;
output[7:0] highb, lowb;
begin
```

```
        memread(lowb, sp);
        sp = sp + 1;
        memread(highb, sp);
        sp = sp + 1;
    end
    endtask

    /* check hold request */
    task holdreq;
    begin
        aleff = 0;
        s0ff = 0;
        s1ff = 1;
        iomff = 0;
        addr = pc;
        if(hold) begin
            holdff = 1;
            acontrol = 0;
            dcontrol = 0;
            @ec2 hldaff = 1;
        end
        else begin
            acontrol = 1;
            dcontrol = 1;
        end
        @ec1 dcontrol = 0;
        @ec1 @ec2;
    end
    endtask

    /* conditional jump, call and return instructions */
    task condjcr;
    reg branch;
    begin
        case(ir[5:3])
            0: branch = !cz; // JNZ CNZ RNZ
            1: branch = cz;  // JZ  CZ  RZ
            2: branch = !cc; // JNC CNC RNC
            3: branch = cc;  // JC  CC  RC
            4: branch = !cp; // JPO CPO RPO
            5: branch = cp;  // JPE CPE RPE
            6: branch = !cs; // JP  CP  RP
            7: branch = cs;  // JM  CM  RM
        endcase
        if(branch)
```

```
        case(ir[2:0])
            0: // return
                pop2b(pc[15:8], pc[7:0]);

            2: // jump
                adread(pc);

            4: // call
                begin :call
                    reg [15:0] newpc;
                    adread(newpc);
                    push2b(pc[15:8], pc[7:0]);
                    pc = newpc;
                end

            default no_instruction;
        endcase
    else
        case(ir[2:0])
            0: ;
            2, 4:
                begin
                    memread(data, pc);
                    pc = pc + 2;
                end
            default no_instruction;
        endcase
end
endtask


/* restart instructions */
task restart;
begin
    push2b(pc[15:8], pc[7:0]);
    case(ir[5:3])
        0: pc = 'h00; // RST 0
        1: pc = 'h08; // RST 1
        2: pc = 'h10; // RST 2
        3: pc = 'h18; // RST 3
        4: pc = 'h20; // RST 4
        5: pc = 'h28; // RST 5
        6: pc = 'h30; // RST 6
        7: pc = 'h38; // RST 7
    endcase
```

```
end
endtask


/* new instructions - except for NOP */
task newops;
    case(ir[5:3])
        0: ; // NOP

        4: // RIM
            begin
                acc = {sid, intmask[7:5], intmask[3:0]};
                if(trapi) begin
                    intmask[3] = inte;
                    trapi = 0;
                end
            end

        6: // SIM
            begin
                if(acc[3]) begin
                    intmask[2:0] = acc[2:0];
                    intmask[6:5] = intmask[6:5] & acc[1:0];
                end
                intmask[8] = acc[4];
                if(acc[6]) @ec1 @ec1 @ec2 sodff = acc[7];
            end

        default no_instruction;
    endcase
endtask


/* decode 1 instructions */
task decode1;
    case(ir[5:4])
        0: pop2b(pc[15:8], pc[7:0]); // RET
        2: pc = {regh, regl}; // PCHL
        3: sp = {regh, regl}; // SPHL
        default no_instruction;
    endcase
endtask


/* decode 2 instructions */
```

```
task decode2;
reg[7:0] saveh, savel;
    case(ir[5:3])
        0: adread(pc); // JMP

        2: // OUT
            begin
                memread(data, pc);
                pc = pc + 1;
                iowrite(data);
            end

        3: // IN
            begin
                memread(data, pc);
                pc = pc + 1;
                ioread(data);
            end

        4: // XTHL
            begin
                saveh = regh;
                savel = regl;
                pop2b(regh, regl);
                push2b(saveh, savel);
            end

        5: // XCHG
            begin
                saveh = regh;
                savel = regl;
                regh = regd;
                regl = rege;
                regd = saveh;
                rege = savel;
            end

        6: // DI, disable interrupt
            {intmask[6:5], intmask[3]} = 0;

        7: // EI, enable interrupt
            intmask[3] = 1;

        default no_instruction;
    endcase
```

```
endtask


/* decode 3 instructions */
task decode3;
    case(ir[5:4])
        0: // CALL
            begin :call
                reg [15:0] newpc;
                adread(newpc);
                push2b(pc[15:8], pc[7:0]);
                pc = newpc;
            end

        default no_instruction;
    endcase
endtask


/* fetch address from pc+1, pc+2 */
task adread;
output[15:0] address;
begin
    memread(address[7:0], pc);
    pc = pc + 1;
    memread(address[15:8], pc);
    if(!int) pc = pc + 1;
end
endtask


/* calculate cp cs and cz */
task calpsz;
input[7:0] tr;
begin
    cp = ^tr;
    cz = tr == 0;
    cs = tr[7];
end
endtask


/* undefined instruction */
task no_instruction;
begin
```

```
        $display("Undefined instruction");
        dumpstate;
        $finish;
    end
    endtask


    /* print the state of the 8085a */
    task dumpstate;
    begin
        $write( "\nDUMP OF 8085A REGISTERS\n",
            "acc=%h regb=%h regc=%h regd=%h rege=%h regh=%h regl=%h\n",
                acc, regb, regc, regd, rege, regh, regl,
            "cs=%h cz=%h cac=%h cp=%h cc=%h\n",
                cs, cz, cac, cp, cc,
            "pc=%h sp=%h addr=%h ir=%h data=%h\n",
                pc, sp, ir, addr, data,
            "intmask=%h aleff=%h s0ff=%h s1ff=%h hldaff=%h holdff=%h\n",
                intmask, aleff, s0ff, s1ff, hldaff, holdff,
            "intaff=%h trapff=%h trapi=%h inte=%h int=%h validint=%h\n",
                intaff, trapff, trapi, inte, int, validint,
            "haltff=%h resetff=%h clockff=%h sodff=%h\n",
                haltff, resetff, clockff, sodff,
            "read=%h write=%h iomff=%h acontrol=%h dcontrol=%h s=%h\n",
                read, write, iomff, acontrol, dcontrol, s,
            "clock=%h x2=%h sid=%h trap=%h rst7p5=%h rst6p5=%h rst5p5=%h\n",
                clock, x2, sid, trap, rst7p5, rst6p5, rst5p5,
            "intr=%h nreset=%h hold=%h ready=%h a=%h ad=%h\n\n",
                intr, nreset, hold, ready, a, ad,
            "instructions executed = %d\n\n", instruction);
    end
    endtask

endmodule /* of i85 */
```

上面两个例子是常用的微处理机 CPU 和外围芯片。在系统芯片的设计中，我们可以用虚拟模型来代替真实的器件对自己所设计的电路功能进行仿真，全面精确地验证自己所设计的部分是否正确。在 ASIC 的制造过程中我们可以利用现存的与之对应的门级结构的电路实体来实现电路的功能。这样就能用较快的速度把许多人的劳动成果集合在一起，把一个极其复杂的数字系统集成在一个很小的硅片上。

思考题：

1) 为什么要设计虚拟模块？
2) 虚拟模块有几种类型？

3) 为什么在 ASIC 设计中要尽量利用商业化的虚拟模块和 IP?

4) 为什么说编写完整精确的虚拟模块，编写者不但需要全面熟练地掌握 Verilog 语言，还需要有高度的责任性，并且需要有一个严格的质量保证体系来确保与工艺的电路的一致性？

第十章 设计练习进阶

前言：

在前面九章学习的基础上， 通过本章十个阶段的练习，一定能逐步掌握 Verilog HDL 设计的要点。我们可以先理解样板模块中每一条语句的作用，然后对样板模块进行综合前和综合后仿真，再独立完成每一阶段规定的练习。当十个阶段的练习做完后，便可以开始设计一些简单的逻辑电路和系统。很快我们就能过渡到设计相当复杂的数字逻辑系统。当然，复杂的数字逻辑系统的设计和验证，不但需要系统结构的知识和经验的积累，还需要了解更多的语法现象和掌握高级的 Verilog HDL 系统任务，以及与 C 语言模块接口的方法（即 PLI），这些已超出的本书的范围。有兴趣的同学可以阅读 Verilog 语法参考资料和有关文献，自己学习，我们将在下一本书中介绍 Verilog 较高级的用法。

练习一. 简单的组合逻辑设计

目的: 掌握基本组合逻辑电路的实现方法。

这是一个可综合的数据比较器，很容易看出它的功能是比较数据 a 与数据 b，如果两个数据相同,则给出结果 1,否则给出结果 0。在 Verilog HDL 中,描述组合逻辑时常使用 assign 结构。注意 equal=(a==b)?1:0,这是一种在组合逻辑实现分支判断时常使用的格式。

模块源代码：

```
//---------------- compare.v ------------------
module compare(equal,a,b);
input a,b;
output equal;
   assign  equal=(a==b)?1:0; //a 等于 b 时，equal 输出为 1；a 不等于 b 时，
                             //equal 输出为 0。
endmodule
```

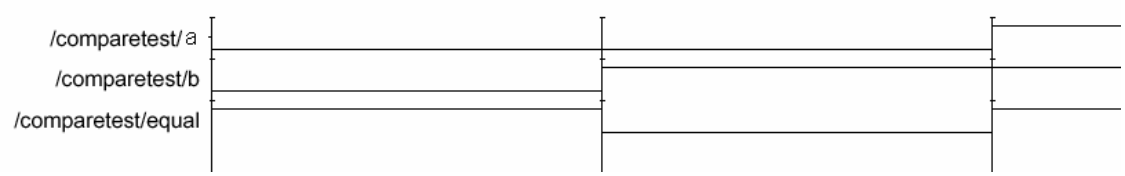
测试模块用于检测模块设计得正确与否，它给出模块的输入信号，观察模块的内部信号和输出信号，如果发现结果与预期的有所偏差，则要对设计模块进行修改。

测试模块源代码：

```
`timescale 1ns/1ns       //定义时间单位。
`include  ".\compare.v" //包含模块文件。在有的仿真调试环境中并不需要此语句。
                        //而需要从调试环境的菜单中键入有关模块文件的路径和名称
module  comparetest;
   reg a,b;
   wire equal;
   initial               //initial 常用于仿真时信号的给出。
```

```
    begin
      a=0;
      b=0;
    #100   a=0; b=1;
    #100   a=1; b=1;
    #100   a=1; b=0;
    #100   $stop;       //系统任务，暂停仿真以便观察仿真波形。
    end

   compare  compare1(.equal(equal),.a(a),.b(b));    //调用模块。

endmodule
```

仿真波形（部分）：



练习：
设计一个字节（8 位）比较器。
要求：比较两个字节的大小，如 a[7:0]大于 b[7:0]输出高电平,否则输出低电平,改写测试模型,使其能进行比较全面的测试 。

练习二. 简单时序逻辑电路的设计

目的：掌握基本时序逻辑电路的实现。

在 Verilog HDL 中，相对于组合逻辑电路，时序逻辑电路也有规定的表述方式。在可综合的 Verilog HDL 模型，我们通常使用 always 块和 @(posedge clk)或 @(negedge clk)的结构来表述时序逻辑。下面是一个 1/2 分频器的可综合模型。

```
// half_clk.v:

   module half_clk(reset,clk_in,clk_out);
   input clk_in,reset;
   output clk_out;
   reg clk_out;

   always @(posedge clk_in)
```

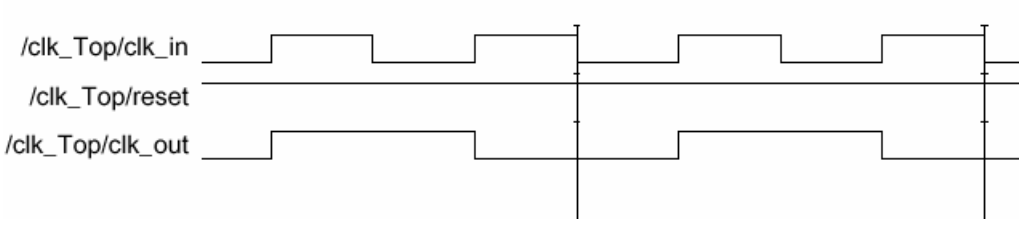
```
      begin
        if(!reset)  clk_out=0;
        else      clk_out=~clk_out;
      end
    endmodule
```

在 always 块中，被赋值的信号都必须定义为 reg 型，这是由时序逻辑电路的特点所决定的。对于 reg 型数据，如果未对它进行赋值，仿真工具会认为它是不定态。为了能正确地观察到仿真结果，在可综合风格的模块中我们通常定义一个复位信号 reset，当 reset 为低电平时，对电路中的寄存器进行复位。

测试模块的源代码：

```
//-------------------  clk_Top.v ------------------------------

`timescale 1ns/100ps
`define clk_cycle 50

module clk_Top.v
reg clk,reset;
wire clk_out;

always  #`clk_cycle  clk = ~clk;

initial
  begin
         clk = 0;
         reset = 1;
    #100 reset = 0;
    #100 reset = 1;
    #10000 $stop;
  end

half_clk half_clk(.reset(reset),.clk(clk_in),.clk_out(clk_out));

endmodule
```

仿真波形：



练习：依然作 clk_in 的二分频 clk_out，要求输出与上例的输出正好反相。编写测试模块，给出仿真波形。

练习三. 利用条件语句实现较复杂的时序逻辑电路

目的：掌握条件语句在 Verilog HDL 中的使用。

与常用的高级程序语言一样,为了描述较为复杂的时序关系，Verilog HDL 提供了条件语句供分支判断时使用。在可综合风格的 Verilog HDL 模型中常用的条件语句有 if...else 和 case...endcase 两种结构，用法和 C 程序语言中类似。两者相较，if...else 用于不很复杂的分支关系，实际编写可综合风格的模块、特别是用状态机构成的模块时，更常用的是 case...endcase 风格的代码。这一节我们给的是有关 if...else 的范例，有关 case...endcase 结构的代码已后会经常用到。

下面给出的范例也是一个可综合风格的分频器，是将 10M 的时钟分频为 500K 的时钟。基本原理与 1/2 分频器是一样的，但是需要定义一个计数器，以便准确获得 1/20 分频

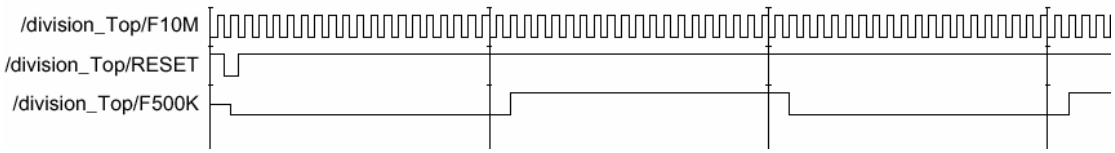
模块源代码：

```
// ---------------- fdivision.v -------------------------------
module fdivision(RESET,F10M,F500K);
    input F10M,RESET;
    output F500K;
    reg F500K;
    reg [7:0]j;
      always @(posedge F10M)
        if(!RESET)          //低电平复位。
          begin
            F500K <= 0;
            j <= 0;
          end
        else
          begin
            if(j==19)       //对计数器进行判断，以确定 F500K 信号是否反转。
              begin
                j <= 0;
                F500K <= ~F500K;
              end
            else
              j <= j+1;
          end
    endmodule
```

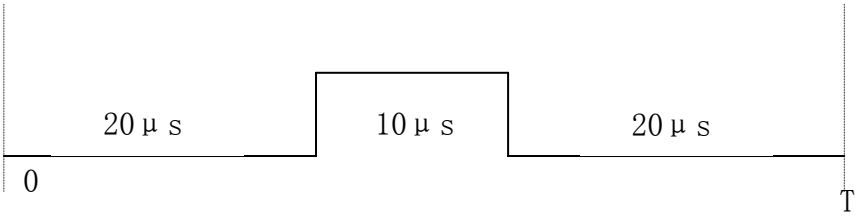
测试模块源代码：

```
//---------------  fdivision_Top.v ------------------------

`timescale 1ns/100ps
`define clk_cycle 50

module division_Top;

reg F10M,RESET;

wire  F500K_clk;

always #`clk_cycle   F10M_clk  =  ~ F10M_clk;

initial
  begin
    RESET=1;
    F10M=0;
    #100 RESET=0;
    #100 RESET=1;
    #10000 $stop;
  end

fdivision fdivision (.RESET(RESET),.F10M(F10M),.F500K(F500K_clk));

endmodule
```

仿真波形：



练习：利用 10M 的时钟，设计一个单周期形状如下的周期波形。



练习四. 设计时序逻辑时采用阻塞赋值与非阻塞赋值的区别

目的：1.明确掌握阻塞赋值与非阻塞赋值的概念和区别；
2.了解阻塞赋值的使用情况。

阻塞赋值与非阻塞赋值，在教材中我们已经了解了它们之间在语法上的区别以及综合后所得到的电路结构上的区别。在 always 块中，阻塞赋值可以理解为赋值语句是顺序执行的，而非阻塞赋值可以理解为赋值语句是并发执行的。实际的时序逻辑设计中，一般的情况下非阻塞赋值语句被更多地使用，有时为了在同一周期实现相互关联的操作，也使用了阻塞赋值语句。(注意：在实现组合逻辑的 assign 结构中，无一例外地都必须采用阻塞赋值语句。

下例通过分别采用阻塞赋值语句和非阻塞赋值语句的两个看上去非常相似的两个模块 blocking.v 和 non_blocking.v 来阐明两者之间的区别。

模块源代码：

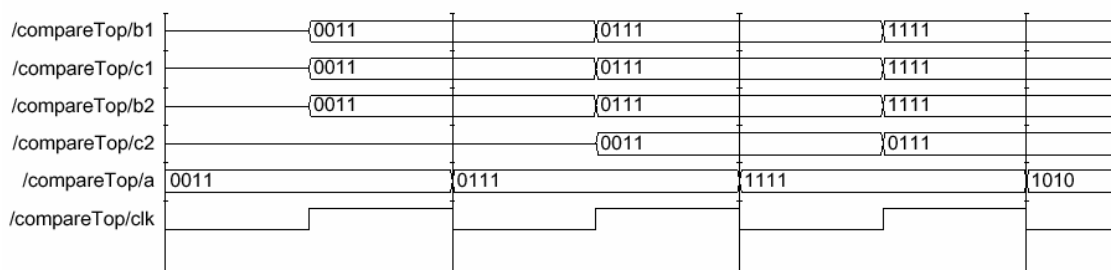
```
// -------------   blocking.v ---------------

module blocking(clk,a,b,c);
  output [3:0] b,c;
  input  [3:0] a;
  input        clk;
  reg    [3:0] b,c;
  always @(posedge clk)
   begin
    b = a;
    c = b;
    $display("Blocking: a = %d, b = %d, c = %d.",a,b,c);
   end
endmodule


//-------------  non_blocking.v --------------------
module non_blocking(clk,a,b,c);

  output [3:0] b,c;
  input  [3:0] a;
  input        clk;
  reg    [3:0] b,c;

  always @(posedge clk)
  begin
    b <= a;
    c <= b;
    $display("Non_Blocking: a = %d, b = %d, c = %d.",a,b,c);
  end
```

```
endmodule
```

测试模块源代码：

```
//-------------  compareTop.v ------------------------------

`timescale 1ns/100ps
`include "./blocking.v"
`include "./non_blocking.v"

module compareTop;

  wire [3:0] b1,c1,b2,c2;
  reg  [3:0] a;
  reg        clk;

  initial
  begin
    clk = 0;
    forever #50 clk = ~clk;
  end

  initial

  begin
    a = 4'h3;
    $display("_____________________________");
    # 100 a = 4'h7;
    $display("_____________________________");
    # 100 a = 4'hf;
    $display("_____________________________");
    # 100 a = 4'ha;
    $display("_____________________________");
    # 100 a = 4'h2;
    $display("_____________________________");
    # 100  $display("_____________________________");
    $stop;
  end
  non_blocking  non_blocking(clk,a,b2,c2);
  blocking      blocking(clk,a,b1,c1);

endmodule
```

仿真波形（部分）：



思考：在 blocking 模块中按如下写法，仿真与综合的结果会有什么样的变化？作出仿真波形，分析综合结果。

1.
```
always @(posedge clk)
  begin
    c = b;
    b = a;
  end
```

2.
```
always @(posedge clk) b=a;
always @(posedge clk) c=b;
```

练习五. 用 always 块实现较复杂的组合逻辑电路

目的: 1.掌握用 always 实现组合逻辑电路的方法;
2.了解 assign 与 always 两种组合逻辑电路实现方法之间的区别。

仅使用 assign 结构来实现组合逻辑电路,在设计中会发现很多地方会显得冗长且效率低下。而适当地采用 always 来设计组合逻辑，往往会更具实效。已进行的范例和练习中，我们仅在实现时序逻辑电路时使用 always 块。从现在开始，我们对它的看法要稍稍改变。

下面是一个简单的指令译码电路的设计示例。该电路通过对指令的判断，对输入数据执行相应的操作，包括加、减、与、或和求反，并且无论是指令作用的数据还是指令本身发生变化，结果都要作出及时的反应。显然，这是一个较为复杂的组合逻辑电路，如果采用 assign 语句，表达起来非常复杂。示例中使用了电平敏感的 always 块，所谓电平敏感的触发条件是指在@后的括号内电平列表中的任何一个电平发生变化，(与时序逻辑不同，它在@后的括号内没有沿敏感关键词，如 posedge 或 negedge）就能触发 always 块的动作，并且运用了 case 结构来进行分支判断，不但设计思想得到直观的体现，而且代码看起来非常整齐、便于理解。

```
//---------------  alu.v ----------------------------
`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;
reg[7:0]    out;
input[2:0] opcode;
input[7:0] a,b;              //操作数。

always@(opcode or a or b)    //电平敏感的 always 块
begin
      case(opcode)
          `plus:  out = a+b;   //加操作。
          `minus: out = a-b;  //减操作。
          `band:  out = a&b;   //求与。
          `bor:   out = a|b;   //求或。
          `unegate:  out=~a; //求反。
          default:   out=8'hx;//未收到指令时，输出任意态。
      endcase
end
endmodule
```

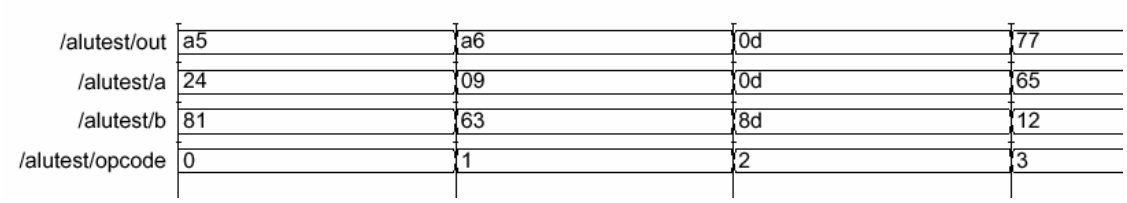
同一组合逻辑电路分别用 always 块和连续赋值语句 assign 描述时，代码的形式大相径庭，但是在 always 中适当运用 default（在 case 结构中）和 else（在 if…else 结构中），通常可以综合为纯组合逻辑，尽管被赋值的变量一定要定义为 reg 型。不过，如果不使用 default 或 else 对缺省项进行说明，则易生成意想不到的锁存器，这一点一定要加以注意。

指令译码器的测试模块源代码：

```
//------------- alu_Top.v -----------------
`timescale 1ns/1ns
`include  "./alu.v"
module    alutest;
   wire[7:0] out;
   reg[7:0]  a,b;
   reg[2:0]  opcode;
   parameter   times=5;
   initial
    begin
        a={$random}%256;  //Give a radom number blongs to [0,255] .
        b={$random}%256;  //Give a radom number blongs to [0,255].
        opcode=3'h0;
        repeat(times)
          begin
          #100     a={$random}%256;  //Give a radom number.
                   b={$random}%256;  //Give a radom number.
                   opcode=opcode+1;
          end
```

```
        #100  $stop;

    end

    alu    alu1(out,opcode,a,b);

endmodule
```

仿真波形（部分）：



练习：运用 always 块设计一个八路数据选择器。要求：每路输入数据与输出数据均为 4 位 2 进制数，当选择开关(至少 3 位)或输入数据发生变化时，输出数据也相应地变化。

练习六. 在 Verilog HDL 中使用函数

目的：掌握函数在模块设计中的使用。

与一般的程序设计语言一样，Veirlog HDL 也可使用函数以适应对不同变量采取同一运算的操作。Veirlog HDL 函数在综合时被理解成具有独立运算功能的电路，每调用一次函数相当于改变这部分电路的输入以得到相应的计算结果。

下例是函数调用的一个简单示范，采用同步时钟触发运算的执行，每个 clk 时钟周期都会执行一次运算。并且在测试模块中，通过调用系统任务$display 在时钟的下降沿显示每次计算的结果。

模块源代码：

```
module tryfunct(clk,n,result,reset);

  output[31:0] result;
  input[3:0]  n;
  input reset,clk;
  reg[31:0] result;

  always @(posedge clk)     //clk 的上沿触发同步运算。
    begin
         if(!reset)              //reset 为低时复位。
            result<=0;
        else
            begin
```

```
                result <= n * factorial(n)/((n*2)+1);
            end
    end

  function [31:0] factorial;      //函数定义。
    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

endmodule
```

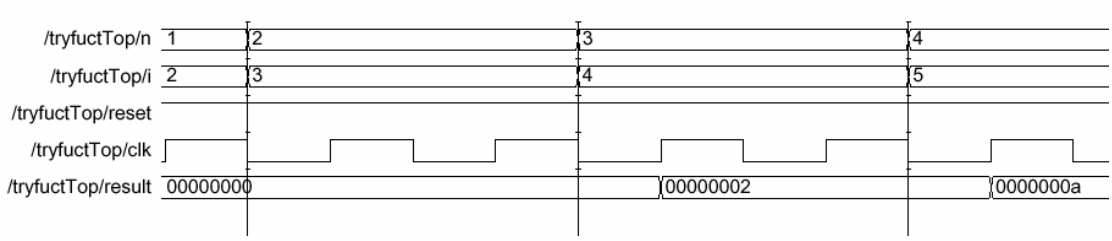
测试模块源代码：

```
`include "./step6.v"
`timescale 1ns/100ps
`define clk_cycle 50

module tryfuctTop;

reg[3:0] n,i;
reg reset,clk;

wire[31:0] result;

initial
  begin
    n=0;
    reset=1;
    clk=0;
    #100 reset=0;
    #100 reset=1;
    for(i=0;i<=15;i=i+1)
      begin
        #200 n=i;
      end
    #100 $stop;
  end

always #`clk_cycle clk=~clk;
```

```
tryfunct tryfunct(.clk(clk),.n(n),.result(result),.reset(reset));

endmodule
```

上例中函数 factorial(n)实际上就是阶乘运算。必须提醒大家注意的是，在实际的设计中，我们不希望设计中的运算过于复杂，以免在综合后带来不可预测的后果。经常的情况是，我们把复杂的运算分成几个步骤，分别在不同的时钟周期完成。

仿真波形(部分)：



练习：设计一个带控制端的逻辑运算电路，分别完成正整数的平方、立方和阶乘的运算。编写测试模块，并给出仿真波形。

练习七. 在 Verilog HDL 中使用任务（task）

目的：掌握任务在结构化 Verilog HDL 设计中的应用。

仅有函数并不能完全满足 Veirlog HDL 中的运算需求。当我们希望能够将一些信号进行运算并输出多个结果时，采用函数结构就显得非常不方便，而任务结构在这方面的优势则十分突出。任务本身并不返回计算值，但是它通过类似 C 语言中形参与实参的数据交换，非常快捷地实现运算结果的调用。此外，我们还常常利用任务来帮助我们实现结构化的模块设计，将批量的操作以任务的形式独立出来，这样设计的目的通常一眼看过去就很明了。

下面是一个利用 task 和电平敏感的 always 块设计比较后重组信号的组合逻辑的实例。可以看到，利用 task 非常方便地实现了数据之间的交换，如果要用函数实现相同的功能是非常复杂的；另外，task 也避免了直接用一般语句来描述所引起的不易理解和综合时产生冗余逻辑等问题。

模块源代码：

```
//------------------ sort4.v ------------------
module sort4(ra,rb,rc,rd,a,b,c,d);
  output[3:0] ra,rb,rc,rd;
  input[3:0] a,b,c,d;
  reg[3:0] ra,rb,rc,rd;
  reg[3:0] va,vb,vc,vd;

  always @ (a or b or c or d)
    begin
```

```
      {va,vb,vc,vd}={a,b,c,d};
      sort2(va,vc);                //va 与 vc 互换。
      sort2(vb,vd);                //vb 与 vd 互换。
      sort2(va,vb);                //va 与 vb 互换。
      sort2(vc,vd);                //vc 与 vd 互换。
      sort2(vb,vc);                //vb 与 vc 互换。
      {ra,rb,rc,rd}={va,vb,vc,vd};
    end

  task sort2;
    inout[3:0] x,y;
    reg[3:0] tmp;
    if(x>y)
      begin
        tmp=x;       //x 与 y 变量的内容互换，要求顺序执行，所以采用阻塞赋值方式。
        x=y;
        y=tmp;
      end
  endtask

endmodule
```

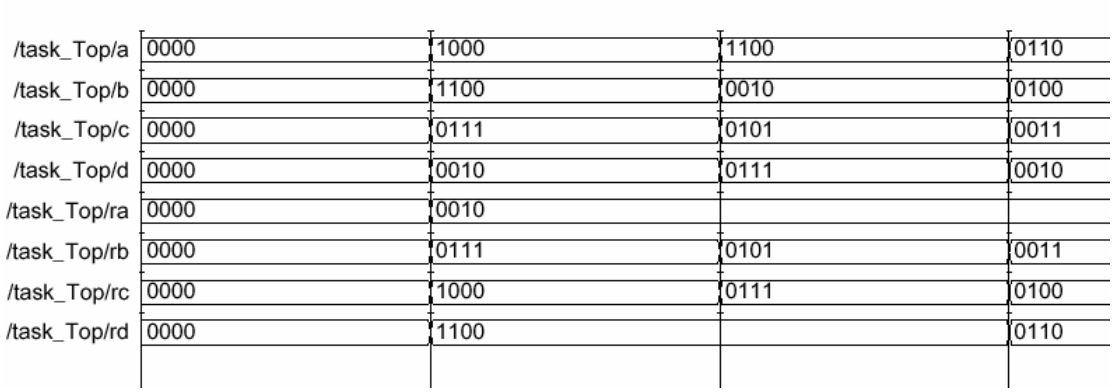
值得注意的是 task 中的变量定义与模块中的变量定义不尽相同,它们并不受输入输出类型的限制。如此例,x 与 y 对于 task sort2 来说虽然是 inout 型,但实际上它们对应的是 always 块中变量，都是 reg 型变量。

测试模块源代码：

```
`timescale 1ns/100ps
`include "sort4.v"

module task_Top;
 reg[3:0] a,b,c,d;
 wire[3:0] ra,rb,rc,rd;

 initial
  begin
    a=0;b=0;c=0;d=0;
    repeat(5)
    begin
      #100  a ={$random}%15;
            b ={$random}%15;
            c ={$random}%15;
            d ={$random}%15;
    end
```

```
    #100 $stop;

sort4 sort4 (.a(a),.b(b),.c(c),.d(d), .ra(ra),.rb(rb),.rc(rc),.rd(rd));

endmodule
```

仿真波形（部分）：



练习：设计一个模块，通过任务完成 3 个 8 位 2 进制输入数据的冒泡排序。要求：时钟触发任务的执行，每个时钟周期完成一次数据交换的操作。

练习八. 利用有限状态机进行复杂时序逻辑的设计

目的：掌握利用有限状态机实现复杂时序逻辑的方法；

在数字电路中我们已经学习过通过建立有限状态机来进行数字逻辑的设计，而在 Verilog HDL 硬件描述语言中，这种设计方法得到进一步的发展。通过 Verilog HDL 提供的语句，我们可以直观地设计出适合更为复杂的时序逻辑的电路。关于有限状态机的设计方法在教材中已经作了较为详细的阐述，在此就不赘述了。

下例是一个简单的状态机设计，功能是检测一个 5 位二进制序列“10010”。考虑到序列重叠的可能，有限状态机共提供 8 个状态（包括初始状态 IDLE）。

模块源代码：

```
seqdet.v
module seqdet(x,z,clk,rst,state);
input  x,clk,rst;
output z;
output[2:0] state;
reg[2:0] state;
wire z;
```

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

assign  z = ( state==E && x==0 )? 1 : 0;     //当 x=0 时，状态已变为 E，
                                            //状态为 D 时，x 仍为 1。因此
                      //输出为 1 的条件为 ( state==E && x==0 )。
always @(posedge clk)
   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;
```

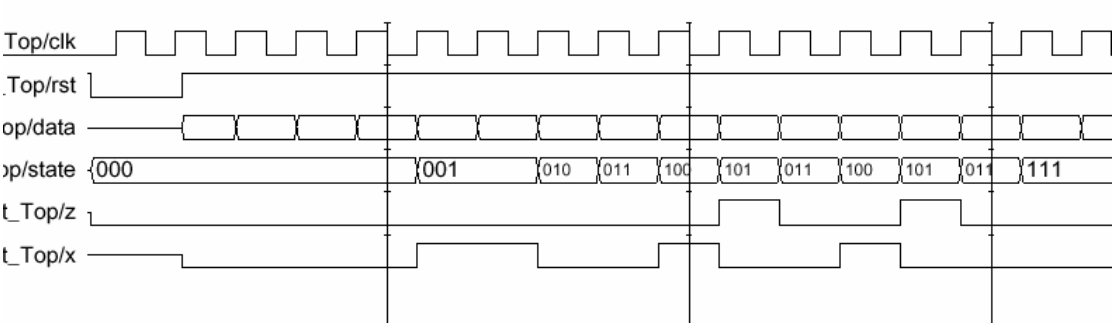
```
                    end
                else
                    begin
                        state <= A;
                    end
            E:      if(x==0)
                    begin
                        state <= C;
                    end
                else
                    begin
                        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;      //缺省状态为初始状态。
          endcase
endmodule
```

测试模块源代码：

```
//------------------- seqdet.v --------------------
`timescale 1ns/1ns
`include "./seqdet.v"
module seqdet_Top;
  reg clk,rst;
  reg[23:0] data;
  wire[2:0] state;
  wire z,x;
  assign x=data[23];
  always  #10 clk = ~clk;
  always @(posedge clk)
         data={data[22:0],data[23]};

  initial
```

```
    begin
      clk=0;
      rst=1;
      #2 rst=0;
      #30 rst=1;
      data ='b1100_1001_0000_1001_0100;
      #500 $stop;
    end

  seqdet  m(x,z,clk,rst,state);

endmodule
```

仿真波形：



练习：设计一个串行数据检测器。要求是：连续 4 个或 4 个以上的 1 时输出为 1，其他输入情况下为 0。编写测试模块并给出仿真波形。

练习九.利用状态机的嵌套实现层次结构化设计

目的： 1.运用主状态机与子状态机产生层次化的逻辑设计；
2.在结构化设计中灵活使用任务（task）结构。

在上一节，我们学习了如何使用状态机的实例。实际上，单个有限状态机控制整个逻辑电路的运转在实际设计中是不多见，往往是状态机套用状态机，从而形成树状的控制核心。这一点也与我们提倡的层次化、结构化的自顶而下的设计方法相符，下面我们就将提供一个这样的示例以供大家学习。

该例是一个简化的 EPROM 的串行写入器。事实上，它是一个 EPROM 读写器设计中实现写功能的部分经删节得到的，去除了 EPROM 的启动、结束和 EPROM 控制字的写入等功能，只具备这样一个雏形。工作的步骤是： 1.地址的串行写入； 2.数据的串行写入；3.给信号源应答，信号源给出下一个操作对象； 4.结束写操作。通过移位令并行数据得以一位一位输出。

模块源代码：

```
module writing(reset,clk,address,data,sda,ack);
  input reset,clk;
  input[7:0] data,address;

  output sda,ack;  //sda 负责串行数据输出;
                   //ack 是一个对象操作完毕后，模块给出的应答信号。
  reg link_write;  //link_write 决定何时输出。
  reg[3:0] state;  //主状态机的状态字。
  reg[4:0] sh8out_state;  //从状态机的状态字。
  reg[7:0] sh8out_buf;    //输入数据缓冲。
  reg finish_F;           //用以判断是否处理完一个操作对象。
  reg ack;

  parameter
    idle=0,addr_write=1,data_write=2,stop_ack=3;
  parameter
    bit0=1,bit1=2,bit2=3,bit3=4,bit4=5,bit5=6,bit6=7,bit7=8;

  assign   sda = link_write? sh8out_buf[7] : 1'bz;

  always @(posedge clk)
    begin
    if(!reset)               //复位。
      begin
         link_write<= 0;
         state    <= idle;
         finish_F <= 0;
         sh8out_state<=idle;
               ack<= 0;
         sh8out_buf<=0;
      end
    else
      case(state)

      idle:
        begin
        link_write  <= 0;
           state    <= idle;
           finish_F <= 0;
           sh8out_state<=idle;
                 ack<= 0;
           sh8out_buf<=address;
          state    <= addr_write;
        end
```

```
      addr_write:         //地址的输入。
        begin
        if(finish_F==0)
          begin  shift8_out; end
        else
          begin
             sh8out_state <= idle;
             sh8out_buf   <= data;
                    state <= data_write;
                finish_F <= 0;
          end
        end

      data_write:       //数据的写入。
        begin
        if(finish_F==0)
          begin  shift8_out; end
        else
          begin
              link_write <= 0;
                   state <= stop_ack;
                 finish_F <= 0;
                      ack <= 1;
          end
        end

      stop_ack:             //完成应答。
        begin
          ack <= 0;
          state <= idle;
        end

      endcase
    end

task shift8_out;                //串行写入。
  begin
     case(sh8out_state)

     idle:
       begin
           link_write  <= 1;
          sh8out_state <= bit0;
```

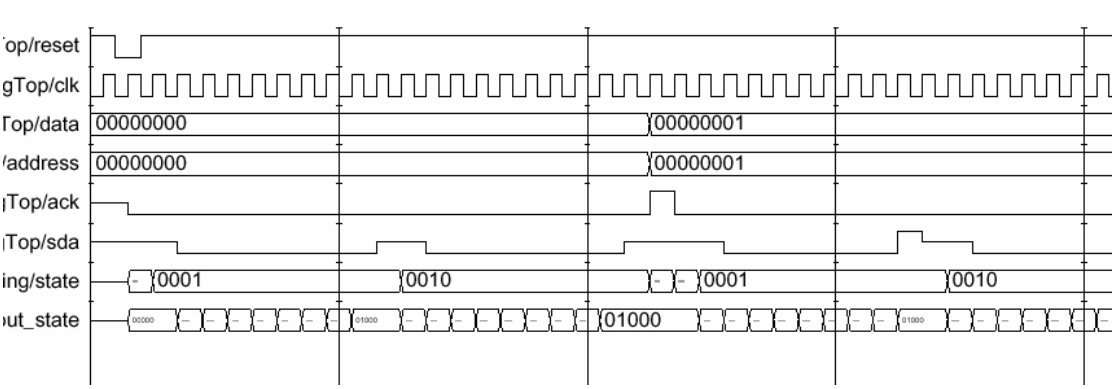
```
    end

bit0:
  begin
      link_write <= 1;
    sh8out_state <= bit1;
      sh8out_buf <= sh8out_buf<<1;
  end

bit1:
  begin
    sh8out_state<=bit2;
    sh8out_buf<=sh8out_buf<<1;
  end

bit2:
  begin
    sh8out_state<=bit3;
    sh8out_buf<=sh8out_buf<<1;
  end

bit3:
  begin
    sh8out_state<=bit4;
    sh8out_buf<=sh8out_buf<<1;
  end

bit4:
  begin
    sh8out_state<=bit5;
    sh8out_buf<=sh8out_buf<<1;
  end

bit5:
  begin
    sh8out_state<=bit6;
    sh8out_buf<=sh8out_buf<<1;
  end

bit6:
  begin
    sh8out_state<=bit7;
    sh8out_buf<=sh8out_buf<<1;
  end
```

```
     bit7:
       begin
         link_write<= 0;
         finish_F<=finish_F+1;
       end

     endcase
  end
endtask

endmodule
```

测试模块源代码：

```
`timescale 1ns/100ps
`define clk_cycle 50
module writingTop;
  reg reset,clk;
  reg[7:0] data,address;
  wire ack,sda;

  always #`clk_cycle  clk = ~clk;

  initial
    begin
        clk=0;
        reset=1;
        data=0;
        address=0;
        #(2*`clk_cycle) reset=0;
        #(2*`clk_cycle) reset=1;
        #(100*`clk_cycle) $stop;
    end

  always @(posedge ack)      //接收到应答信号后，给出下一个处理对象。
    begin
        data=data+1;
        address=address+1;
    end
  writing writing(.reset(reset),.clk(clk),.data(data),
                  .address(address),.ack(ack),.sda(sda));
endmodule
```

仿真波形：



练习：仿照上例，编写一个实现 EPROM 内数据串行读取的模块。编写测试模块，给出仿真波形。

练习十. 通过模块之间的调用实现自顶向下的设计

目的：学习状态机的嵌套使用实现层次化、结构化设计。

现代硬件系统的设计过程与软件系统的开发相似，设计一个大规模的集成电路的往往由模块多层次的引用和组合构成。 层次化、结构化的设计过程，能使复杂的系统容易控制和调试。 在 Verilog HDL 中，上层模块引用下层模块与 C 语言中程序调用有些类似，被引用的子模块在综合时作为其父模块的一部分被综合，形成相应的电路结构。在进行模块实例引用时，必须注意的是模块之间对应的端口，即子模块的端口与父模块的内部信号必须明确无误地一一对应，否则容易产生意想不到的后果。

下面给出的例子是设计中遇到的一个实例，其功能是将并行数据转化为串行数据送交外部电路编码，并将解码后得到的串行数据转化为并行数据交由 CPU 处理。显而易见，这实际上是两个独立的逻辑功能，分别设计为独立的模块，然后再合并为一个模块显得目的明确、层次清晰。

```
// -----------------    p_to_s.v ------------------------------------
module  p_to_s(D_in,T0,data,SEND,ESC,ADD_100);
    output        D_in,T0;              // D_in 是串行输出，T0 是移位时钟并给
                                        // CPU 中断，以确定何时给出下个数据。
    input   [7:0] data;                 //并行输入的数据。
    input         SEND,ESC,ADD_100;     //SEND、ESC 共同决定是否进行并到串
                                        //的数据转化。ADD_100 决定何时置数。
    wire      D_in,T0;
    reg [7:0] DATA_Q,DATA_Q_buf;

    assign        T0 = ! (SEND & ESC);      //形成移位时钟。.
    assign        D_in = DATA_Q[7];         //给出串行数据。

    always @(posedge T0 or negedge ADD_100)  //ADD_100 下沿置数，T0 上沿移位。
      begin
```

```
        if(!ADD_100)
          DATA_Q = data;
        else
          begin
          DATA_Q_buf = DATA_Q<<1;          //DATA_Q_buf 作为中介，以令综合器
          DATA_Q = DATA_Q_buf;             //能辨明。
          end
      end

endmodule
```

在 p_to_s.v 中，由于移位运算虽然可综合，但是不是简单的 RTL 级描述，直接用 DATA_Q<=DATA_Q<<1 的写法在综合时会令综合器产生误解。另外，在该设计中，由于时钟 T0 的频率较低，所以没有象以往那样采用低电平置数，而是采用 ADD_100 的下降沿置数。

```
//---------------------- s_to_p.v ----------------------------
module s_to_p(T1, data, D_out,DSC,TAKE,ADD_101);
       output       T1;                      //给 CPU 中断，以确定 CPU 何时取转化
                                             //得到的并行数据。
       output [7:0] data;
       input   D_out, DSC, TAKE, ADD_101;  //D_out 提供输入串行数据。DSC、TAKE
                                             //共同决定何时取数。
       wire   [7:0] data;
       wire         T1,clk2;
       reg    [7:0] data_latch, data_latch_buf;

       assign       clk2 = DSC  & TAKE ;   //提供移位时钟。
       assign       T1 = !clk2;

       assign       data =  (!ADD_101) ? data_latch : 8'bz;
       always@(posedge clk2)
            begin
               data_latch_buf = data_latch << 1;   //data_latch_buf 作缓冲
               data_latch     = data_latch_buf;   //，以令综合器能辩明。
                data_latch[0] = D_out;
           end
endmodule
```

将上面的两个模块合并起来的 sys.v 的源代码：

```
//------------------- sys.v ----------------------------
`include "./p_to_s.v"
`include "./s_to_p.v"
module sys(D_in,T0,T1, data, D_out,SEND,ESC,DSC,TAKE,ADD_100,ADD_101);
  input         D_out,SEND,ESC,DSC,TAKE,ADD_100,ADD_101;
  inout  [7:0]  data;
  output        D_in,T0,T1;
```

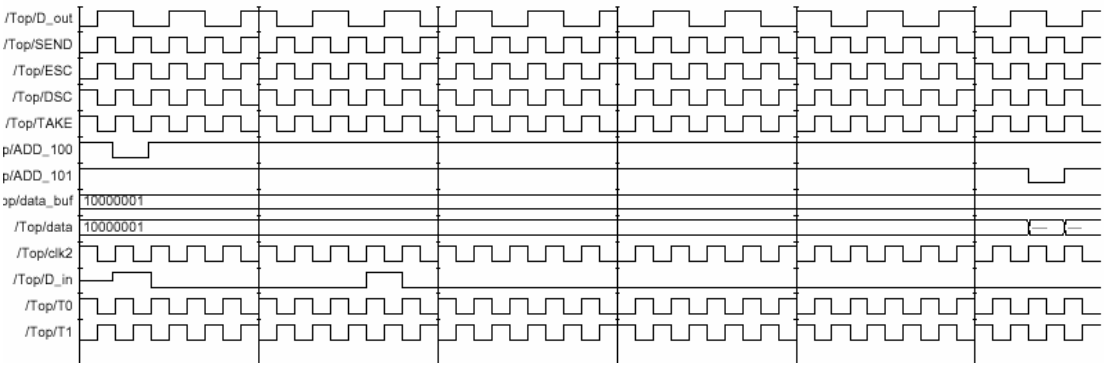
```
  p_to_s   p_to_s(.D_in(D_in),.T0(T0),.data(data),
                .SEND(SEND),.ESC(ESC),.ADD_100(ADD_100));
  s_to_p   s_to_p(.T1(T1),.data(data),.D_out(D_out),
          .DSC(DSC),.TAKE(TAKE),.ADD_101(ADD_101));

endmodule
```

测试模块源代码：

```
//-------------Top test file for sys.v ------------------
`timescale 1ns/100ps
`include "./sys.v"
module Top;
 reg D_out,SEND,ESC,DSC,TAKE,ADD_100,ADD_101;
 reg[7:0] data_buf;
 wire [7:0] data;
 wire clk2;

 assign  data = (ADD_101) ? data_buf : 8'bz;
                                  //data 在 sys 中是 inout 型变量，ADD_101
                                  //控制 data 是作为输入还是进行输出。
assign  clk2 =DSC && TAKE;

initial
  begin
     SEND = 0;
     ESC = 0;
     DSC = 1;
     TAKE = 1;
     ADD_100 = 1;
     ADD_101 = 1;
  end

initial
  begin
    data_buf = 8'b10000001;
    #90 ADD_100 = 0;
    #100 ADD_100 = 1;
  end

always
  begin
    #50;
    SEND = ~SEND;
```

```
    ESC = ~ESC;
  end

initial
  begin
    #1500 ;
    SEND = 0;
    ESC  = 0;
    DSC  = 1;
    TAKE = 1;
    ADD_100 = 1;
    ADD_101 = 1;
    D_out = 0;
    #1150 ADD_101 = 0;
    #100 ADD_101 =1;
    #100 $stop;
  end

always
  begin
    #50 ;
    DSC = ~DSC;
    TAKE = ~TAKE;
  end

always @(negedge clk2) D_out = ~D_out;

sys    sys(.D_in(D_in),.T0(T0),.T1(T1),.data(data),.D_out(D_out),
             .ADD_101(ADD_101), .SEND(SEND),.ESC(ESC),.DSC(DSC),
                                .TAKE(TAKE),.ADD_100(ADD_100));
endmodule
```

仿真波形：



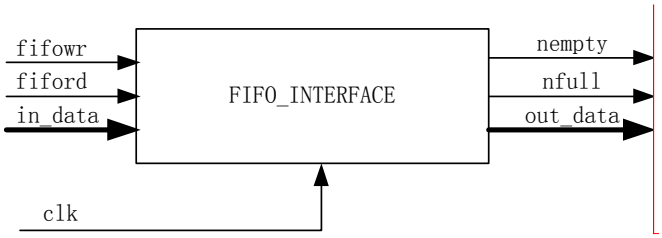
练习：设计一个序列发生器。要求根据输入的 8 位并行数据输出串行数据,如果输入数据在 0—127 之间则输出一位 0，如果输入数据在 128—255 之间则输出一位 1，同步时钟触发；并且和范例 8 的序列检测器搭接，形成一个封闭系统。编写测试模块，并给出仿真波形。

练习十二　利用 SRAM 设计一个 FIFO

在本练习中，要求同学利用练习十一中提供的 SRAM 模型，设计 SRAM 读写控制逻辑，使 SRAM 的行为对用户表现为一个 FIFO（先进先出存储器）。

1）**设计要求：**

本练习要求同学设计的 FIFO 为同步 FIFO，即对 FIFO 的读/写使用同一个时钟。该 FIFO 应当提供用户读使能（fiford）和写使能(fifowr)输入控制信号，并输出指示 FIFO 状态的非空（nempty）和非满（nfull）信号，FIFO 的输入、输出数据使用各自的数据总线：in_data 和 out_data。下图为 FIFO 接口示意图。

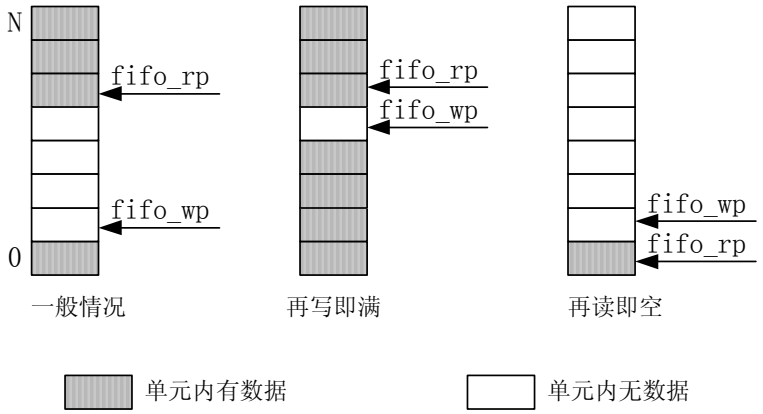


批注 [fwn1]: 这些不是设计的具体内容，而是为检验设计正确与否所提供的验证环境。此处如描述一下 SRAM 与 FIFO 的差异，并由此得到 FIFO 接口设计的关键在与 SRAM 地址产生这一结论会好一些。

2）**FIFO 接口的设计思路**

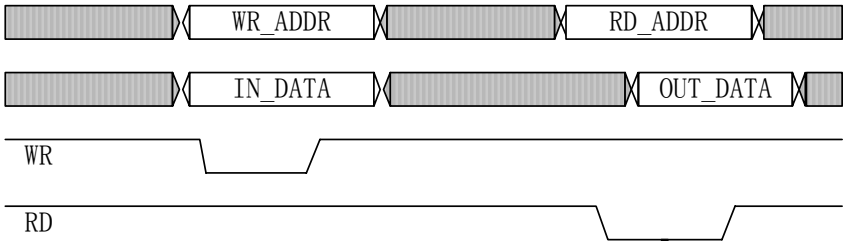
FIFO 的数据读写操作与 SRAM 的数据读写操作基本上相同，只是 FIFO 没有地址。所以用 SRAM 实现 FIFO 的关键点是如何产生正确的 SRAM 地址。

我们可以借用软件中的方法，将 FIFO 抽象为环形数组，并用两个指针：读指针（fifo_rp）和写指针（fifo_wp）控制对该环形数组的读写。其中，读指针 fifo_rp 指向下一次读操作所要读取的单元，并且每完成一次读操作，fifo_rp 加一；写指针 fifo_wp 则指向下一次写操作时存放数据的单元，并且每完成一次写操作，fifo_wp 加一。由 fifo_rp 和 fifo_wp 的定义易知，当 FIFO 被读空或写满后，fifo_rp 和 fifo_wp 将指向同一单元，但在读空和写满之前 FIFO 的状态是不同的，所以如果能区分这两种状态，再通过比较 fifo_rp 和 fifo_wp 就可以得到 nempty 和 nfull 信号了。下图为 FIFO 工作状态的示意。



在得到 nfull 和 nempty 信号后，就需要考虑如何应用这两个信号来控制对 FIFO 的读写，使得 FIFO 在被写满后不能再写入，从而防止覆盖原有数据，并且在被读空后也不能再进行读操作，防止读取无效数据。

此外，在进 SRAM 读写操作时，应该注意建立地址、数据和控制信号的先后顺序。一般情况下，希望对 SRAM 读写的波形时序如下图所示：



即写 SRAM 时，先建立地址和数据，然后置写使能信号 WR 有效，在 WR 保持有效一定时间后，先复位 WR，然后释放地址和数据总线。而读取 SRAM 时，则先建立地址，然后置读使能 RD 有效，在 RD 维持有效一定时间后，复位 RD，同时读取数据总线上的值，然后再释放地址总线。在进行 FIFO 操作时，用户一般希望除了没有地址外，其它三个信号的时序关系能保持不变。请同学们在设计 FIFO 控制信号与 SRAM 控制信号间逻辑关系时注意这一点。

3）FIFO 接口的测试

在完成一个设计后，需要进行测试以确认设计的正确性和完整性。而要进行测试，就需要编写测试激励和结果检查程序，即测试平台（testbench）。在某些情况下，如果设计的接口能够预先确定，测试平台的编写也可以在设计完成之前就进行，这样做的好处是在设计测试平台的同时也在更进一步深入了解设计要求，有助于理清设计思路，及时发现设计方案的错误。

编写测试激励时，除了注意对实际可能存在的各种情况的覆盖外，还要有意针对非正常情况下的操作进行测试。在本练习中，就应当进行在 FIFO 读空后继续读取、FIFO 写满后继续写入、FIFO 复位后马上读取等操作的测试。

测试激励中通常会有一些复杂操作需要反复进行，如本练习中对 FIFO 的读写操作。这时可以将这些复杂操作纳入到几个 task 中，即减小了激励编写的工作量，也使得程序的可读性更好。

下面的测试程序给同学们做为参考，希望同学们能先用这段程序测试所设计的 FIFO 接口，然后编写自己更全面的测试程序。

```
`define FIFO_SIZE 8
`include “sram.v”       // 有的仿真工具不需要加这句, 只要 sram.v 模块编译过就可以了
`timescale 1ns/1ns

module t;

reg [7:0]        in_data;                  //FIFO 数据总线
reg                fiford,fifowr;          //FIFO 控制信号

wire[7:0]         out_data;
```

```
wire                nfull, nempty;          //FIFO 状态信号

reg                 clk,rst;

wire[7:0]           sram_data;              //SRAM 数据总线
wire[10:0]          address;                //SRAM 的地址总线
wire                rd,wr;                  //SRAM 读写控制信号

reg [7:0]           data_buf[`FIFO_SIZE:0]; //数据缓存，用于结果检查
integer index;                              //用于读写 data_buf 的指针

//系统时钟
initial    clk=0;
always     #25 clk=~clk;

//测试激励序列
initial
    begin
        fiford=1;
        fifowr=1;
        rst=1;
        #40 rst=0;
        #42 rst=1;

        if (nempty) $display($time,"Error: FIFO be empty, nempty should be low.\n");

        //连续写 FIFO
        index = 0;
        repeat(`FIFO_SIZE) begin
            data_buf[index]=$random;
            write_fifo(data_buf[index]);
            index = index + 1;
        end

        if (nfull) $display($time,"Error: FIFO full, nfull should be low.\n");
        repeat(2) write_fifo($random);
        #200

        //连续读 FIFO
        index=0;
        read_fifo_compare(data_buf[index]);
        if (~nfull) $display($time,"Error: FIFO not full, nfull should be high.\n");

        repeat(`FIFO_SIZE-1) begin
```

```
            index = index + 1;
            read_fifo_compare(data_buf[index]);
        end

        if (nempty) $display($time,"Error: FIFO be empty, nempty should be low.\n");

        repeat(2) read_fifo_compare(8'bx);

        reset_fifo;

        //写后读 FIFO
        repeat(`FIFO_SIZE*2)
        begin
            data_buf[0] = $random;
            write_fifo(data_buf[0]);
            read_fifo_compare(data_buf[0]);
        end

        //异常操作
        reset_fifo;
        read_fifo_compare(8'bx);
        write_fifo(data_buf[0]);
        read_fifo_compare(data_buf[0]);

        $stop;
    end

fifo_interface    fifo_interface(
                      .in_data(in_data),.out_data(out_data),
                      .fiford(fiford),.fifowr(fifowr),
                      .nfull(nfull),.nempty(nempty),
                      .address(address),.sram_data(sram_data),
                      .rd(rd),.wr(wr),
                      .clk(clk),.rst(rst)
        );

sram m1( .Address(address),
            .Data(sram_data),
            .SRG(rd),                //SRAM 读使能
            .SRE(1'b0),              //SRAM 片选,低有效
            .SRW(wr));               //SRAM 写使能

task write_fifo;
input [7:0] data;
```

```
begin
        in_data=data;
    #50 fifowr=0;                   //往 SRAM 中写数
    #200 fifowr=1;
    #50;
end
endtask

task read_fifo_compare;
input [7:0] data;
begin
    #50 fiford=0;                   //从 SRAM 中读数
    #200 fiford=1;
        if (out_data != data)
            $display($time,"Error: Data retrieved (%h) not match the one stored (%h). \n",
                    out_data, data);

    #50;
end
endtask

task reset_fifo;
begin
    #40 rst=0;
    #40 rst=1;
end
endtask

endmodule
```

4) FIFO 接口的参考设计

FIFO 接口的实现有多种方案，下面给出的参考设计只是其中一种。希望同学们在完成自己的设计后，和参考设计做一下比较。

```
`define SRAM_SIZE 8   //为减小对 FIFO 控制器的测试工作量,置 SRAM 空间为 8Byte
`timescale 1ns/1ns

module fifo_interface(
        in_data,      //对用户的输入数据总线
        out_data,     //对用户的输出数据总线,
        fiford,       //FIFO 读控制信号，低电平有效
        fifowr,       //FIFO 写控制信号，低电平有效
        nfull,
```

```
                nempty,

                address,      //到 SRAM 的地址总线
                sram_data,    //到 SRAM 的双向数据总线
                rd,           //SRAM 读使能，低电平有效
                wr,           //SRAM 写使能，低电平有效

                clk,          //系统时钟信号
                rst);         //全局复位信号，低电平有效

//来自用户的控制输入信号
input           fiford, fifowr, clk, rst;

//来自用户的数据信号
input[7:0]      in_data;
output[7:0]     out_data;


reg[7:0]        in_data_buf,          //输入数据缓冲区
                out_data_buf;         //输出数据缓冲区

//输出到用户的状态指示信号
output          nfull, nempty;
reg             nfull, nempty;

//输出到 SRAM 的控制信号
output          rd, wr;

//到 SRAM 的双向数据总线
inout[7:0]      sram_data;

//输出到 SRAM 的地址总线
output[10:0]    address;
reg[10:0]       address;

//Internal Register
reg[10:0]       fifo_wp,              //FIFO 写指针
                fifo_rp;              //FIFO 读指针

reg[10:0]       fifo_wp_next,         //fifo_wp 的下一个值
                fifo_rp_next;         //fifo_rp 的下一个值

reg             near_full, near_empty;
```

```
reg[3:0]        state;                  //SRAM 操作状态机寄存器

parameter       idle            = 'b0000,
                read_ready      = 'b0100,
                read            = 'b0101,
                read_over       = 'b0111,
                write_ready     = 'b1000,
                write           = 'b1001,
                write_over      = 'b1011;

//SRAM 操作状态机
always @(posedge clk or negedge rst)
    if (~rst)
        state <= idle;
    else
        case(state)
            idle:                                   //等待对 FIFO 的操作控制信号
                if (fifowr==0 && nfull)     //用户发出写 FIFO 申请,且 FIFO 未满
                    state<=write_ready;
                else if(fiford==0 && nempty)//用户发出读 FIFO 申请,且 FIFO 未空
                    state<=read_ready;
                else                                //没用对 FIFO 操作的申请
                    state<=idle;

            read_ready:                             //建立 SRAM 操作所需地址和数据
                    state <= read;

            read:                                   //等待用户结束当前读操作
                if (fiford == 1)
                    state <= read_over;
                else
                    state <= read;

            read_over:                              //继续给出 SRAM 地址以保证数据稳定
                    state <= idle;

            write_ready:                            //建立 SRAM 操作所需地址和数据
                    state <= write;

            write:                                  //等待用户结束当前写操作
                if (fifowr == 1)
                    state <= write_over;
                else
                    state <= write;
```

```
            write_over:             //继续给出 SRAM 地址和写入数据以保证数据稳定
                        state <= idle;

            default: state<=idle;
    endcase

//产生 SRAM 操作相关信号
assign rd = ~state[2];          //state 为 read_ready 或 read 或 read_over
assign wr = (state == write) ? fifowr : 1'b1;

always @(posedge clk)
    if (~fifowr)
        in_data_buf <= in_data;

assign sram_data = (state[3]) ? //state 为 write_ready 或 write 或 write_over
                            in_data_buf : 8'hzz;


always @(state or fiford or fifowr or fifo_wp or fifo_rp)
    if (state[2] || ~fiford)
        address = fifo_rp;
    else if (state[3] || ~fifowr)
        address = fifo_wp;
    else
        address = 'bz;

//产生 FIFO 数据
assign out_data = (state[2]) ?
                            sram_data : 8'bz;

always @(posedge clk)
    if (state == read)
        out_data_buf <= sram_data;


//计算 FIFO 读写指针
always @(posedge clk or negedge rst)
    if (~rst)
        fifo_rp <= 0;
    else if (state == read_over)
        fifo_rp <= fifo_rp_next;

always @(fifo_rp)
```

```
    if (fifo_rp == `SRAM_SIZE-1)
        fifo_rp_next = 0;
    else
        fifo_rp_next = fifo_rp + 1;

always @(posedge clk or negedge rst)
    if (~rst)
        fifo_wp <= 0;
    else if (state == write_over)
        fifo_wp <= fifo_wp_next;

always @(fifo_wp)
    if (fifo_wp == `SRAM_SIZE-1)
        fifo_wp_next = 0;
    else
        fifo_wp_next = fifo_wp + 1;


always @(posedge clk or negedge rst)
    if (~rst)
        near_empty <= 1'b0;
    else if (fifo_wp == fifo_rp_next)
        near_empty <= 1'b1;
    else
        near_empty <= 1'b0;

always @(posedge clk or negedge rst)
    if (~rst)
        nempty <= 1'b0;
    else if (near_empty && state == read)
        nempty <= 1'b0;
    else if (state == write)
        nempty <= 1'b1;

always @(posedge clk or negedge rst)
    if (~rst)
        near_full <= 1'b0;
    else if (fifo_rp == fifo_wp_next)
        near_full <= 1'b1;
    else
        near_full <= 1'b0;

always @(posedge clk or negedge rst)
    if (~rst)
```

```
        nfull <= 1'b1;
    else if (near_full && state == write)
        nfull <= 1'b0;
    else if (state == read)
        nfull <= 1'b1;

endmodule
```

练习十一. 简单卷积器的设计

下面我们将共同来完成一个用于教学的但有实际接口器件背景的小型设计—“简单卷积器的设计”。希望通过这个设计，使同学们建立起专用数字计算系统设计的基本概念。设计分成许多步骤进行，具体过程排列如下：

1）明确设计任务：

在设计之前必须明确设计的具体内容。卷积器是数字信号处理系统中常用的部件。它对模拟输入信号实时采样，得到数字信号序列。然后对数字信号进行卷积运算，再将卷积结果存入RAM中。对模拟信号的采样由A/D转换器来完成，而卷积过程由卷积器来实现。为了设计卷积器，首先要设计RAM和A/D转换器的Verilog HDL模型。在电子工业发达的国家，可以通过商业渠道得到非常准确的外围器件的虚拟模型。如果没有外围器件的虚拟模型，就需要仔细地阅读和分析RAM和A/D转换器的器件说明书，来自行编写。因为RAM和A/D转换器不是我们设计的硬件对象，所以需要的只是它们的行为模型，精确的行为模型需要认真细致地编写，并不比可综合模块容易编写。它们与实际器件的吻合程度直接影响设计的成功。在这里我们把重点放在卷积器的设计上，直接给出RAM 和A/D转换器的Verilog HDL模型和它们的器件参数(见附录)，同学们可以对照器件手册，认真阅读RAM和A/D转换器的Verilog HDL模型。对RAM 和A/D转换器的Verilog HDL模型的详细了解对卷积器的设计是十分必要的。

到目前为止，我们对设计模块要完成的功能比较明确了。总结如下：首先它要控制AD变换器进行AD变换，从AD变换器得到变换后的数字序列，然后对数字序列进行卷积，最后将结果存入RAM。 下面让我们一起来设计它。

2）卷积器的设计

通过前面的练习我们已经知道，用高层次的设计方法来设计复杂的时序逻辑，重点是把时序逻辑抽象为有限状态机，并用可综合风格的Verilog HDL把这样的状态机描述出来。 下面我们将通过注释来介绍整个程序的设计过程。我们选择8位输入总线，输出到RAM的数据总线也选择8位，卷积值的高、低字节被分别写到两个RAM中。地址总线为11位。为了理解卷积器设计中的状态机，必须对A/D转换器和RAM的行为模块有深入的理解。

```
`timescale 100ps/100ps
module  con1(address,indata,outdata,wr,nconvst,nbusy,
                                        enout1,enout2,CLK,reset,start);

 input  CLK,      //采用10MHZ的时钟
        reset,    //复位信号
        start,    //因为RAM的空间是有限的，当RAM存满后采样和卷积都会停止。
                  //此时给一个start的高电平脉冲将会开始下一次的卷积。
        nbusy;    //从A/D转换器来的信号表示转换器的忙或闲
 output wr,       //RAM 写控制信号
        enout1,enout2, //enout1是存储卷积低字节结果RAM的片选信号
                       //enout2是存储卷积高字节结果RAM的片选信号
```

```
       nconvst, //给 A/D 转换器的控制信号,命令转换器开始工作,低电平有效
       address;  //地址输出

input [7:0]  indata;  //从 A/D 转换器来的数据总线
output[7:0] outdata;  //写到 RAM 去的数据总线

wire  nbusy;
reg   wr;
reg   nconvst,
      enout1,
      enout2;
reg[7:0]  outdata;

reg[10:0] address;
reg[8:0]  state;
reg[15:0] result;
reg[23:0] line;
reg[11:0] counter;
reg    high;
reg[4:0] j;
reg    EOC;



parameter  h1=1,h2=2,h3=3;  //假设的系统系数
parameter  IDLE=9'b000000001,  START=9'b000000010,  NCONVST=9'b000000100,
           READ=9'b000001000,  CALCU=9'b000010000,  WRREADY=9'b000100000,
             WR=9'b001000000,  WREND=9'b010000000,  WAITFOR=9'b100000000;

parameter  FMAX=20;  //因为 A/D 转换的时间是随机的，为保证按一定的频率采样，A/D
                     //转换控制信号应以一定频率给出。这里采样频率通过 FMAX 控制
                     // 为 500KHZ。

always @(posedge CLK)
      if(!reset)
         begin
              state<=IDLE;
              nconvst<=1'b1;
              enout1<=1;
              enout2<=1;
              counter<=12'b0;
              high<=0;
              wr<=1;
              line<=24'b0;
```

```
        address<=11'b0;
    end
else
        case(state)
             IDLE:if(start==1)
                    begin
                         counter<=0;      //counter 是一个计数器，记录已
                                          //用的 RAM 空间
                         line<=24'b0;
                         state<=START;
                    end
                  else
                         state<=IDLE;
             //START 状态控制 A/D 开始转换
              START: if(EOC)
                        begin
                             nconvst<=0;
                             high<=0;
                             state<= NCONVST;
                        end
                     else
                             state<=START;
             //NCONVST 状态是 A/D 转换保持阶段
              NCONVST: begin
                           nconvst<=1;
                           state<=READ;
                       end


             //READ 状态读取 A/D 转换结果，计算卷积结果
              READ:  begin
                       if(EOC)
                         begin
                           line<={line[15:0],indata};
                           state<=CALCU;
                         end
                       else
                           state<=READ;
                    end

              CALCU: begin
                      result<=line[7:0]*h1+line[15:8]*h2+line[23:16]*h3;
                      state<=WRREADY;
                     end
```

```
//将卷积结果写入 RAM 时，先写入低字节，再写入高字节
//WRREADY 状态是写 RAM 准备状态，建立地址和数据信号
 WRREADY:begin
                 address<=counter;
                 if(!high)   outdata<=result[7:0];
                 else        outdata<=result[15:8];
                 state<=WR;
         end
//WR 状态产生片选和写脉冲
 WR: begin
          if(!high)  enout1<=0;
          else       enout2<=0;
          wr<=0;
          state<=WREND;
     end
//WREND 状态结束一次写操作，若还未写入高字节则转到 WRREADY 状
// 态开始高字节写入
 WREND:begin
            wr<=1;
            enout1<=1;
            enout2<=1;
            if(!high)
              begin
                   high<=1;
                   state<=WRREADY;
              end
            else   state<=WAITFOR;
       end
//WAITFOR 状态控制采样频率并判断 RAM 是否已被写满
 WAITFOR: begin
               if(j==FMAX-1)
                 begin
                      counter<=counter+1;
                      if(!counter[11])   state<=START;
                      else
                        begin
                             state<=IDLE;
                             $display($time,"The ram is used
                                      up.");
                             $stop;
                        end
                 end
               else     state<=WAITFOR;
          end
```

```
                    default:state<=IDLE;
                endcase

 // assign rd=1;    //RAM 的读信号始终保持为高

 //j 记录时钟，与 FMAX 共同控制采样频率
 //由于直接用 CLK 的上升沿对 nbusy 判断以
 //决定某些操作是否运行时，会因为两个信号
 //的跳变沿相隔太近而令状态机不能正常工作。因此
 //利用 CLK 的下降沿建立 EOC 信号与 nbusy 同步，相位
 //相差 180 度，然后用 CLK 的上升沿判断操作是否进行。

 always @(negedge CLK )
   begin
        EOC <= nbusy;
        if(!reset||state==START)
                  j<=1;
        else
                  j<=j+1;
   end


endmodule
```

3). 前仿真及后仿真

程序写完后首先要做前仿真，我们可用仿真器（如 ModelSim SE/EE PLUS 5.4）来做。为检查我们写的程序，需要编写测试程序，测试程序应尽可能检测出各种极限情况。这里给出一个测试程序供参考。

```
//------------------------- testcon1.v ------------------
`timescale 100ps/100ps
module  testcon1;
  wire  wr,
        enin,
        enout1,
        enout2;
  wire[10:0] address;
  reg  rd,
       CLK,
       reset,
       start;
  wire  nbusy;
  wire nconvst;
  wire[7:0] indata;
```

```
  wire[7:0] outdata;
  integer i;

  parameter HALF_PERIOD=1000;

  //产生 10KHZ 的时钟
  initial
    begin
         rd=1;
         i=0;
         CLK=1;
         forever #HALF_PERIOD CLK=~CLK;
    end
  //产生置位信号
  initial
    begin
         reset=1;
         #(HALF_PERIOD*2 + 50) reset=0;
         #(HALF_PERIOD*3) reset=1;
    end
  //产生开始卷积控制信号
  initial
    begin
         start=0;
         #(HALF_PERIOD*7 + 20) start=1;
         #(HALF_PERIOD*2) start=0;
         #(HALF_PERIOD*1000) start=1;
         #(HALF_PERIOD*2)   start=0;
    end

  assign enin =1;

  con1 con(.address(address),.indata(indata),.outdata(outdata),.wr(wr),
                .nconvst(nconvst),.nbusy(nbusy),.enout1(enout1),
           .enout2(enout2),.CLK(CLK),.reset(reset),.start(start));

  sram ramlow(.Address(address),.Data(outdata),.SRW(wr),.SRG(rd),.SRE(enout1));
  adc  adc(.nconvst(nconvst),.nbusy(nbusy),. data(indata));

endmodule
```

因测试程序已经包括了各模块，只需编译测试程序并运行它。通过仿真器中的菜单（如 ModelSim 仿真器中功能列表中 view 的下拉菜单选择 structure, signal 和 wave），可以根

据需要看到各种信号的波形，由此检测程序。

图 XXXXX 是一个参考波形图，由它我们可以看清整个程序的时序。

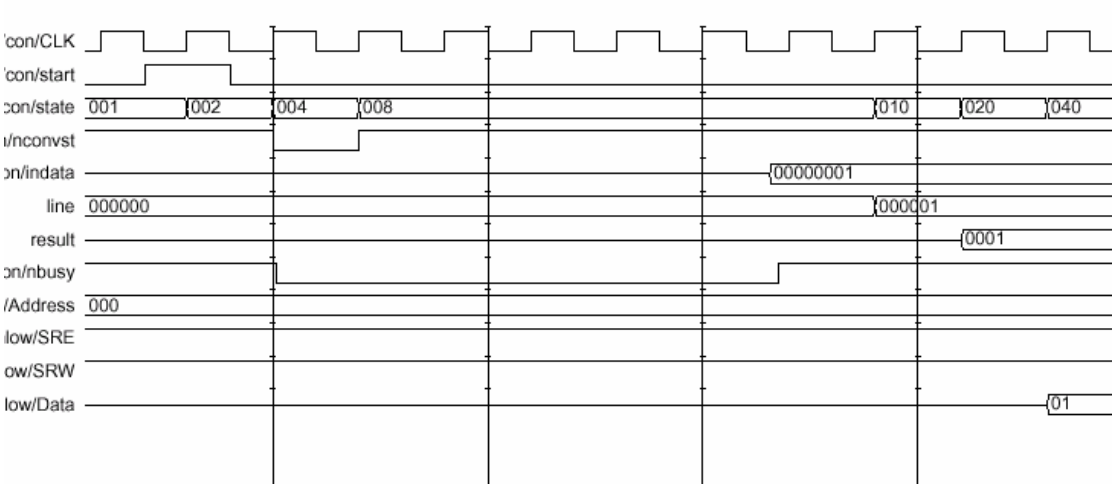


图 XXXXXX

如果前仿真通过了，则可以做后仿真了。后仿真考虑了器件的延时，更具可靠性。首先我们用综合器（如：Synplify ）进行综合。在综合时应注意选择器件库，可选择如 Altera FLEX10K 系列 FPGA 或其他类型的 FPGA。综合完后生成了与原程序名相应的一个扩展名为 edf 的文件。然后我们用布线工具（如：MAX+PLUS II ver. 9.3）对刚才得到的扩展名为 edf 的文件进行编译。如果编译不出错我们就可得到扩展名为 vo 的两个文件，一个文件名与原文件名相同，另一文件名为 alt_max2.vo.

现在我们就可用仿真器（如 ModelSim）来做后仿真。步骤与前仿真一样，对于 Altera 系列的 FPGA 只需将 con1.vo 和 alt_max2.vo 两个文件重新编译，取代原先用 con1.v 编译的模型就可以了，不同的 FPGA 具体方法有些不同，但原理都是一样的。这时将后仿真波形与前仿真波形比较就会发现后仿真把器件的延时考虑进去了。看波形，检查结果是否正确。若不正确则改动原程序，重新进行上述步骤。

4)．卷积器的改进

我们希望设计出快速高效的卷积器。而通过对上面设计的卷积器仿真波形的分析不难发现，有很多时间被浪费在等待 A/D 转换上。同时因 A/D 转换，计算卷积和写入 RAM 是串行工作的，效率很低。为提高效率我们可以采用三片 A/D 转换器同时工作，并将采样过程和计算，写入 RAM 的控制改为并行工作。以下就是改进后的程序。原采样频率为 500KHZ，改进后采样频率为 2.22MHZ，为原采样频率的四倍多。

```
//--------------------- con3ad.v ---------------------------------------
`timescale 1ns/100ps
module
con3ad(indata,outdata,address,CLK,reset,start,nconvst1,nconvst2,nconvst3,
              nbusy1,nbusy2,nbusy3,wr,enout1,enout2);
  input indata,
        CLK,
        reset,
```

```
       start,
       nbusy1,
       nbusy2,
       nbusy3;
output outdata,
       address,
       nconvst1,     // 采用三根控制线控制三片 A/D 转换器
       nconvst2,
       nconvst3,
       wr,
       enout1,
       enout2;
wire[7:0] indata;
wire      CLK,
          reset,
          start,
          nbusy1,
          nbusy2,
          nbusy3;
reg[7:0]  outdata;
reg[10:0] address;
reg       nconvst1,
          nconvst2,
          nconvst3,
          wr,
          enout1,
          enout2;
reg[6:0]  state;
reg[5:0]  i;
reg[1:0]  j;
reg[11:0] counter;
reg[23:0] line;
reg[15:0] result;
reg high;
reg  k;
reg EOC1,EOC2,EOC3;

parameter h1=1,h2=2,h3=3;
parameter    IDLE = 7'b0000001,  READ_PRE = 7'b0000010,
             READ = 7'b0000100,     CALCU = 7'b0001000,
          WRREADY = 7'b0010000,        WR = 7'b0100000,
            WREND = 7'b1000000;

always @(posedge CLK)
```

```
begin
     if(!reset)
       begin
            state<=IDLE;
            counter<=12'b0;
            wr<=1;
            enout1<=1;
            enout2<=1;
            outdata<=8'bz;
            address<=11'bz;
            line<=24'b0;
            result<=16'b0;
            high<=0;
       end    // end of "if"
     else
       begin
            case(state)
                 IDLE:if(start)
                        begin
                             counter<=0;
                             state<=READ_PRE;
                        end
                      else       state<=IDLE;

                 READ_PRE: if(EOC1||EOC2||EOC3)  //由于频率相对改进前的卷积
                                                 //器大大提高，所以加入
                                                 //READ_PRE 状态对取数操作
                                                 //予以缓冲。
                             state<=READ;
                           else
                             state<=READ_PRE;


                 READ:begin
                           high<=0;
                           enout2<=1;
                           wr<=1;
                           if(j==1)
                             begin
                                  if(EOC1)
                                   begin
                                        line<={line[15:0],indata};
                                        state<=CALCU;
                                   end
```

```
                    else   state<=READ_PRE;
                end
             else if(j==2&&counter!=0)
                begin
                    if(EOC2)
                      begin
                           line<={line[15:0],indata};
                           state<=CALCU;
                      end
                    else   state<=READ_PRE;
                end
            else if(j==3&&counter!=0)
              begin
                    if(EOC3)
                      begin
                           line<={line[15:0],indata};
                           state<=CALCU;
                      end
                    else   state<=READ_PRE;
              end
            else  state<=READ;
        end

CALCU:begin
          result<=line[7:0]*h1+line[15:8]*h2+line[23:16]*h;
          state<=WRREADY;
       end

WRREADY:begin
              wr<=1;
              address<=counter;
              if(k==1)state<=WR;
              else    state<=WRREADY;
          end
WR: begin
          if(!high)  enout1<=0;
          else       enout2<=0;
          wr<=0;
          if(!high)   outdata<=result[7:0];
          else        outdata<=result[15:8];
          if(k==1)    state<=WREND;
          else        state<=WR;
      end
WREND:begin
```

```
                                wr<=1;
                                enout1<=1;
                                enout2<=1;
                                if(k==1)
                                  if(!high)
                                    begin
                                         high<=1;
                                         state<=WRREADY;
                                    end
                                  else
                                    begin
                                         counter<=counter+1;
                                         if(counter[11]&&counter[0])
                                                state<=IDLE;
                                         else   state<=READ_PRE;
                                    end
                                else   state<=WREND;
                           end
                      default:state<=IDLE;
                endcase //end of the case
          end  // end of "else"
  end  // end of "always"

//计数器 i 用来记录时间
always @(posedge CLK)
  begin
       if(!reset)  i<=0;
       else
         begin
              if(i==44) i<=0;
              else      i<=i+1;
         end
  end

//j 是控制信号，协调卷积器轮流从三片 A/D 上读取数据。
always @(posedge CLK)
  begin
       if(i==4) j<=2;
       else if(i==10) j<=0;
       else if(i==19) j<=3;
       else if(i==25) j<=0;
       else if(i==34) j<=1;
       else if(i==40) j<=0;
  end
```

```
  //k 是计数器，用以控制写操作信号
  always @(posedge CLK)
    begin
         if(state==WRREADY||state==WR||state==WREND)
            if(k==1)  k<=0;
            else      k<=1;
         else  k<=0;
    end

  //根据计数器 i 控制三片 A/D 转换信号 NCONVST1，NCONVST2，NCONVST3
  always @(posedge CLK)
    begin
         if(!reset) nconvst1<=1;
         else if(i==0) nconvst1<=0;
         else if(i==3) nconvst1<=1;
    end

  always @(posedge CLK)
    begin
         if(!reset) nconvst2<=1;
         else if(i==15) nconvst2<=0;
         else if(i==18) nconvst2<=1;
    end

  always @(posedge CLK)
    begin
         if(!reset) nconvst3<=1;
         else if(i==30)  nconvst3<=0;
         else if(i==33)  nconvst3<=1;
    end

  always @(negedge CLK)
      begin
         EOC1<=nbusy1;
         EOC2<=nbusy2;
         EOC3<=nbusy3;
      end

endmodule
```

测试程序如下：

```
`timescale 1ns/100ps

module  testcon3ad;
  wire  wr,
        enin,
        enout1,
        enout2;
  wire[10:0] address;
  reg   clk,
        reset,
        start;
        rd;
  wire  nbusy1,
        nbusy2,
        nbusy3;

  wire nconvst1,
       nconvst2,
       nconvst3;
  wire[7:0] indata;
  wire[7:0] outdata;

  parameter HALF_PERIOD=15;//时钟周期为 30ns

  initial
    begin
         clk=1;
         forever #HALF_PERIOD clk=~clk;
    end

  initial
    begin
         reset=1;
         #110 reset=0;
         #140 reset=1;
    end
  initial
    begin
         start=0;
         rd=1;
         #420 start=1;
         #120 start=0;
         #107600 start=1;
         #150   start=0;
```

```
    end

  assign enin=1;

  con3ad con3ad(.indata(indata),.outdata(outdata),.address(address),
                .CLK(clk),.reset(reset),.start(start),
                .nconvst1(nconvst1),.nconvst2(nconvst2),.nconvst3(nconvst3),
                .nbusy1(nbusy1),.nbusy2(nbusy2),.nbusy3(nbusy3),
                .wr(wr),.enout1(enout1),.enout2(enout2));

  sram ramlow(.Address(address),.Data(outdata),.SRW(wr),.SRG(rd),.SRE(enout1));

  adc  ad_1(.nconvst(nconvst1),.nbusy(nbusy1),. data(indata));
  adc  ad_2(.nconvst(nconvst2),.nbusy(nbusy2),. data(indata));
  adc  ad_3(.nconvst(nconvst3),.nbusy(nbusy3),. data(indata));

endmodule
```

与前面同样，我们给出仿真波形的片段供大家参考，如图 XXXXXX 。

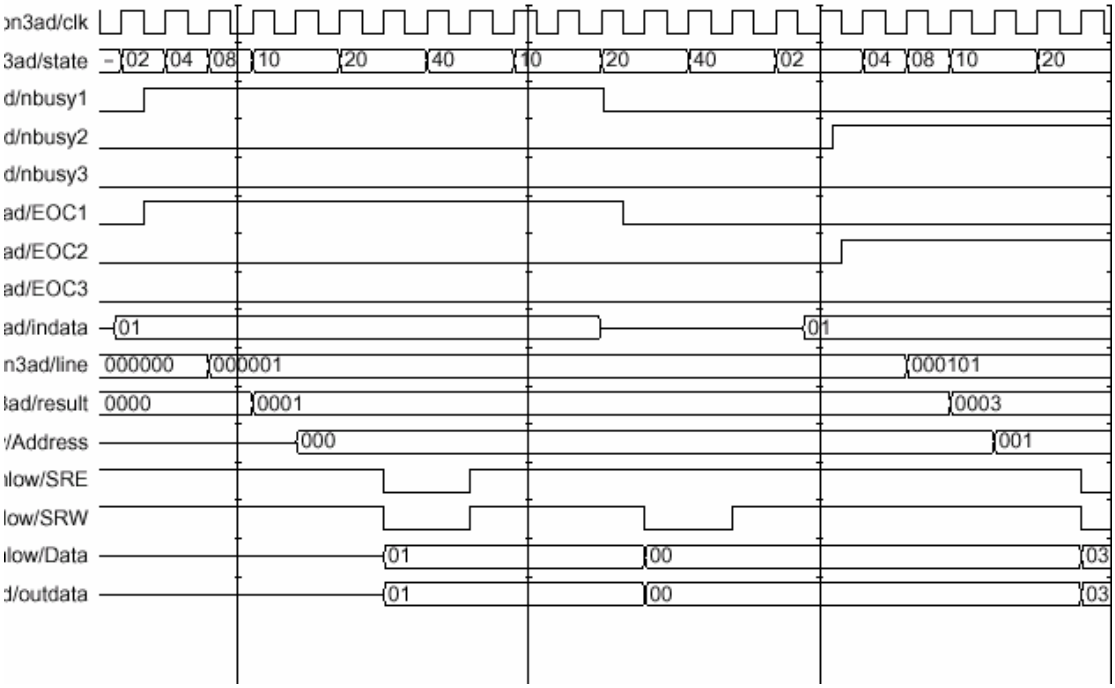


图 XXXXX

经过前仿真确定设计在逻辑上成立后，我们同样可以对这个改进后的卷积器使用合适的器件库进行后仿真，步骤与前面讲述的完全一样，在这里就不在赘述了。

附录一：A/D 转换器的 Verilog HDL 模型和建立模型所需要的技术参数。

我们所用的 A/D 转换器是 AD7886。

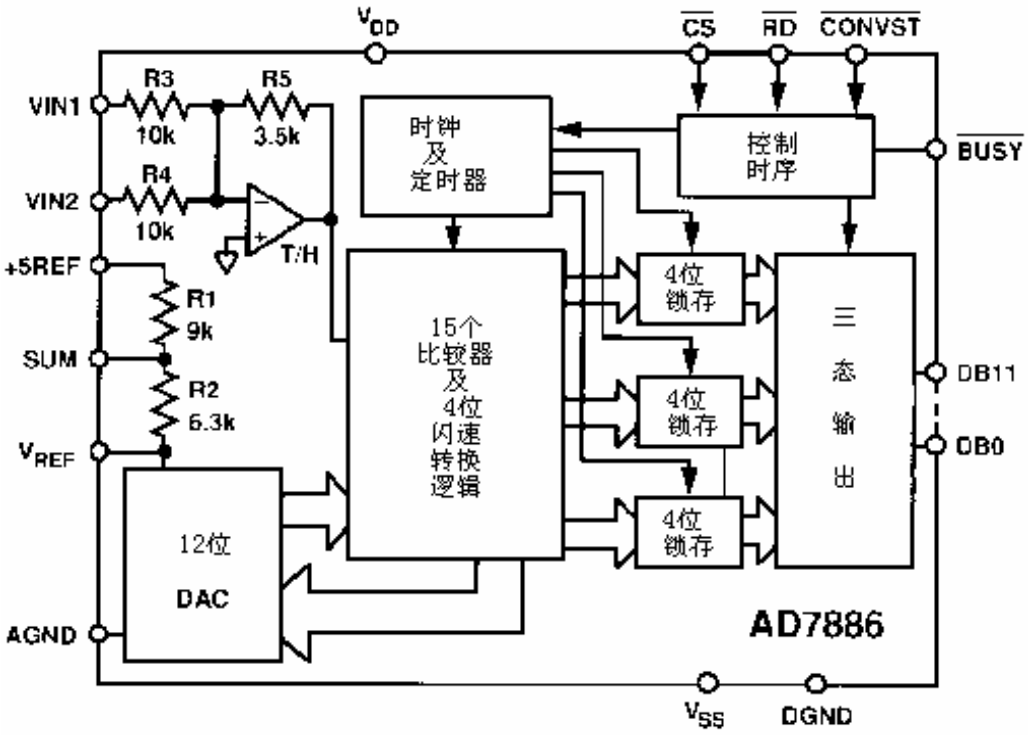


图 XXXXXXXXXX　　AD7886 的功能图

AD7886 的时序控制有两种方法：第一种是 CS 和 RD 输入信号控制 AD7886 三态输出，如图 1XXXX 所示，但 A/D 转换中三态输出封锁，这种方法适合微处理器在 AD7886 转换结束后直接把数据读出；第二种是 CS 和 RD 接到低电平，启动 A/D 转换开始后，数据线输出封锁，直到转换结束，数据输出才有效，如图 2XXXXX 所示，这种方法可以用 A/D 转换结束 BUSY 的上升沿触发外部锁存器锁存数据。

在上述两种时序中，AD7886 进行转换都是由 NCONVST 控制的。NCONVST 的下降沿使采保开始跟踪信号，直到 NCONVST 上升沿来了，ADC 才进行转换。NCONVST 低脉宽度决定了跟踪-保持的建立时间。在 A/D 转换过程中，BUSY 输出位低，转换结束，BUSY 变为高，表示可以取走转换结果。在本设计中，我们用第二种时序控制 AD7886 工作。

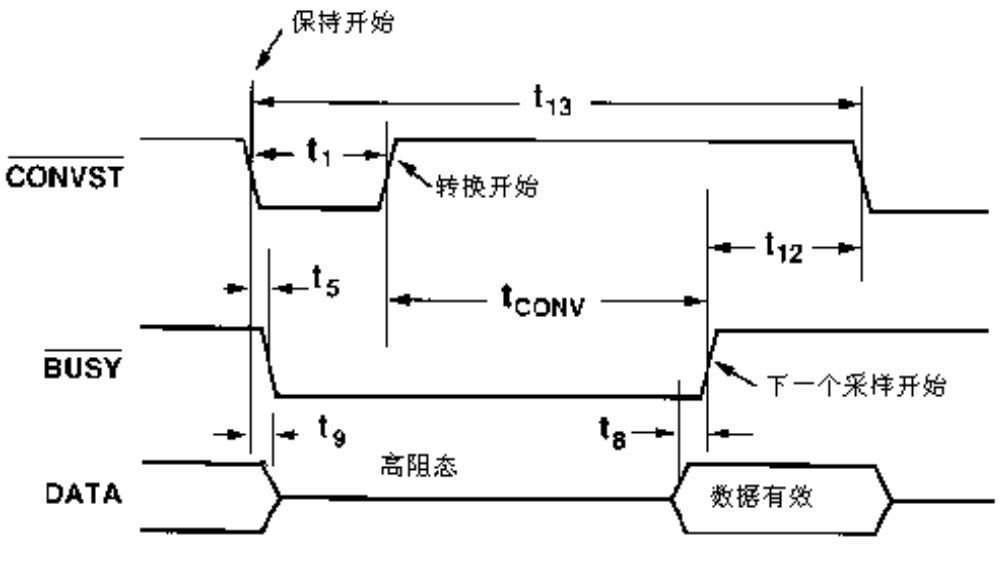


图 XXXXXXX A/D 转换启动和数据读出时序（CS=RD=0）

A/D 转换器的 Verilog HDL 行为模型如下：

```
//------------------- adc.v ----------------------
module   adc (nconvst, nbusy, data);
input     nconvst;          // A/D 启动脉冲 ST，即上图中
output    nbusy;        // A/D 工作标志，即上图中
output       data;          // 数据总线，从 AD.DATA 文件中读取数据后经端口输出
reg[7:0]  databuf,i;   // 内部寄存器
reg       nbusy;
wire[7:0] data;
reg[7:0]  data_mem[0:255];
reg       link_bus;
integer   tconv,
          t5,
          t8,
          t9,
          t12;
integer   wideth1,
          wideth2,
          wideth;

//时间参数定义(依据 AD7886 手册)：
always @(negedge nconvst)
   begin
         tconv =9500+{$random}%500; //(type 950, max 1000) Conversion Time
         t5 ={$random}%1000;  //(max 100)   CONVST to BUSY Propagation Dlay
                              // CL = 10pf
         t8 = 200;      //(min 20)  CL=20pf  Data Setup Time Prior to BUSY
                       //(min 10)  CL=100pf
         t9 = 100+{$random}%900; //(min 10, max 100) Bus Relinquish Time After
```

```
CONVST
         t12 = 2500;   //(type) BUSY High to CONVST Low, SHA Acquisition Time
   end

initial
      begin
       $readmemh("adc.data",data_mem);  //从数据文件 adc.data 中读取数据
       i = 0;
       nbusy  = 1;
           link_bus = 0;
      end

assign data = link_bus? databuf:8'bzz; //三态总线
/*--------------------------------------------------------------------------------
在信号 nconvst 的负跳降沿到来后，隔 t5 秒 nbusy 信号置为低，tconv 是 AD 将模拟信号转
换为数字信号的时间，在信号 nconvst 的正跳降沿到来后经过 tconv 时间后，输出 nbusy
信号变为高。
--------------------------------------------------------------------------------*/
always @(negedge nconvst)
        fork
         #t5  nbusy =0;
             @(posedge nconvst)
                    begin
                         #tconv  nbusy=1;
                    end
        join

/*--------------------------------------------------------------------------------
nconvst 信号的下降沿触发，经过 t9 延时后，把数据总线输出关闭置为高阻态，如图示。
 nconvst 信号的上升沿到来后,经过(tconv - t8)时间,输出一个字节(8 位数据)到 databuf，
该数据来自于 data_mem。而 data_mem 中的数据是初始化时从数据文件 AD.DATA 中读取的。
此时应启动总线的三态输出。
--------------------------------------------------------------------------------*/

always @(negedge nconvst)
      begin
         @(posedge nconvst)
            begin
              #(tconv-t8)    databuf=data_mem[i];
            end

         if(wideth <10000  && wideth>500)
            begin
              if(i==255) i=0;
```

```
              else i=i+1;
            end
         else  i = i;

      end

//在模数转换期间关闭三态输出，转换结束时启动三态输出
always @(negedge nconvst)
  fork
       #t9 link_bus = 1'b0;     //关闭三态输出，不允许总线输出
       @(posedge nconvst)
            begin
            #(tconv-t8)   link_bus=1'b1;
            end
  join


/*-------------------------------------------------------------------------
当 nconvst 输入信号的下一个转换的下降沿与 nbusy 信号上升沿之间时间延迟小于 t12 时，
将会出现警告信息，通知设计者请求转换的输入信号频率太快，A/D 器件转换速度跟不上。
仿真模型不仅能够实现硬件电路的输出功能，同时能够对输入信号进行检测，
当输入信号不符合手册要求时，显示警告信息。
-------------------------------------------------------------------------*/

// 检查 A/D 启动信号的频率是否太快
     always @(posedge nbusy)
       begin
          #t12;
          if (!nconvst)
             begin
               $display("Warning!  SHA Acquisition Time is too short!");
             end
        //  else  $display(" SHA Acquisition Time is enough! ");
       end

// 检查 A/D 启动信号的负脉冲宽度是否足够和太宽
   always @(negedge nconvst)
     begin
          wideth=$time;
          @(posedge nconvst)  wideth=$time-wideth;
          if (wideth<=500 || wideth > 10000)
            begin

             $display("nCONVST Pulse Width = %d",wideth);
```

```
            $display("Warning! nCONVST Pulse Width is too narrow or too wide!");
            //$stop;
          end
    end

endmodule
```

附录二. 2K*8 位 异步 CMOS 静态 RAM HM-65162 模型：

```
/*****************************************************************************
 *   File Name           : sram.v                                            *
 *   Function            : 2K*8bit Asynchronous CMOS Static  RAM             *
 *****************************************************************************/
/*****************************************************************************
 * Module Name           : sram                                              *
 * Description           : 2K*8bit Asynchronous CMOS Static  RAM             *
 * Reference             : HM-65162 reference book                           *
 *****************************************************************************/
/*****************************************************************************
 * sram is a Verilog HDL model for HM-65162,2K*8bit Asynchronous CMOS Static   *
 * RAM. It is used in simulation to substitute the real  RAM to verify whether
 * the writing or reading of the RAM is  OK. This module is a behavioral model *
 * for simulation only, not synthesizable. It's writing and reading function  *
 * are verified.                                                               *
 *****************************  ******************************************/
//---------------------- sram.v ----------------------------------------
module sram(Address, Data, SRG,  SRE, SRW);
input [10:0]   Address;

input          SRG, // Output enable
           SRE, // Chip enable
           SRW; // Write enable

inout [7:0]    Data;// Bus

wire  [10:0]   Addr = Address;
reg   [7:0]    RdData;
reg   [7:0]    SramMem [0:'h7ff];
reg            RdSramDly, RdFlip;
wire  [7:0]    FlpData, Data;
reg    WR_flag;  //To judge the signals according to the specification of HM-65162
integer        i;

wire       RdSram = ~SRG & ~SRE;
```

```
wire       WrSram = ~SRW & ~SRE;
reg   [10:0]   DelayAddr;
reg   [7:0]    DelayData;
reg        WrSramDly;

integer file;

assign FlpData = (RdFlip) ? ~RdData : RdData;
assign Data    = (RdSramDly) ? FlpData: 'hz;

/********************parameters of read circle*****************************/
                      //参数序号、最大或最小、参数含义
parameter TAVQV=90,    //2          (max)    Address access time
          TELQV=90,    //3          (max)   Chip enable access time
          TELQX=5,     //4          (min)   Chip enable output enable time
          TGLQV=65,    //5          (max)   Output enable access time
          TGLQX=5,     //6          (min)   Output inable output enable time
          TEHQZ=50,    //7          (max)   Chip enable output disable time
          TGHQZ=40,    //8          (max)   Output enable output disable time
          TAVQX=5;     //9          (min)   Output hold from address change

/********************parameters of write  circle***************************/

parameter TAVWL=10,    //12        (min)  Address setup   time,
          TWLWH=55,    //13         (min)  Chip enable pulse setup time,
                                          //write enable pluse width,
          TWHAX=15,    //14        (min10)    Write enable read setup time,
                                            //读上升沿后地址保留时间
          TWLQZ=50,    //16        (max)  Write enable output disable time
          TDVWH=30,    //17        (min)    Data setup time
          TWHDX=20,    //18        (min15)  Data hold time
          TWHQX=20,    //19        (min0)   Write enable output enable time,0
          TWLEH=55,    //20        (min)   Write  enable pulse setup time
          TDVEH=30,    //21        (min)   Chip enable data setup time
          TAVWH=70;    //22        (min65)  Address valid to end of write

initial
  begin
       file=$fopen("ramlow.txt");
       if(!file)
     begin
          $display("Could not open the file.");
          $stop;
     end
```

```
  end

initial
  begin
     for(i=0 ; i<'h7ff ; i=i+1)
       SramMem[i] = i;
  //   $monitor($time,,"DelayAddr=%h,DelayData=%h",DelayAddr,DelayData);
  end

initial  RdSramDly = 0;
initial  WR_flag=1;

/*************************READ CIRCLE*****************************/
always @(posedge RdSram)  #TGLQX  RdSramDly = RdSram;
always @(posedge SRW)     #TWHQX  RdSramDly = RdSram;

always @(Addr)
       begin
      #TAVQX;
      RdFlip = 1;
      #(TGLQV - TAVQX);          //address access time
      if (RdSram)     RdFlip = 0;
       end

always @(posedge RdSram)
       begin
      RdFlip = 1;
      #TAVQV;   // Output enable access time
      if (RdSram) RdFlip = 0;
       end

always @(Addr)        #TAVQX RdFlip = 1;

always @(posedge SRG)     #TEHQZ RdSramDly  = RdSram;
always @(posedge SRE)     #TGHQZ RdSramDly  = RdSram;
always @(negedge SRW)     #TWLQZ RdSramDly  = 0;

always @(negedge WrSramDly or posedge RdSramDly)   RdData = SramMem[Addr];

/*************************WRITE CIRCLE*****************************/
always @(Addr)    #TAVWL DelayAddr  = Addr; //Address setup
always @(Data)    #TDVWH DelayData  = Data; //Data setup
always @(WrSram)  #5 WrSramDly =WrSram;
always @(Addr or Data or WrSram)  WR_flag=1;
```

```
always @(negedge SRW )
        begin
           #TWLWH;       //Write enable pulse width
           if (SRW)
            begin
                 WR_flag=0;
                 $display("ERROR! Can't write!
                             Write enable time (W) is too short!");
            end
        end

always @(negedge SRW )
        begin
           #TWLEH;       //Write enable pulse setup time
           if (SRE)
            begin
                 WR_flag=0;
                 $display("ERROR! Can't write! Write enable
                               pulse setup time (E) is too short!");
            end
        end

always @(posedge SRW )
       begin
            #TWHAX;        //Write enable read setup time
            if(DelayAddr !== Addr)
              begin
                   WR_flag=0;
                   $display("ERROR! Can't write!
                             Write enable read setup time is too short!");
              end
       end

always @(Data)
   if (WrSram)
     begin
          #TDVEH;         //Chip enable data setup time
          if (SRE)
            begin
                 WR_flag=0;
                 $display("ERROR! Can't write!
                         Chip enable Data setup time is too short!");
            end
```

```
     end

always @(Data)
  if (WrSram)
    begin
         #TDVEH;
         if (SRW)
           begin
                WR_flag=0;
                $display("ERROR! Can't write!
                        Chip enable Data setup time is too short!");
           end
    end

always @(posedge SRW )
  begin
       #TWHDX;          //Data hold time
       if(DelayData !== Data)
        $display("Warning!  Data hold time is too short!");
  end

always @(DelayAddr or DelayData or WrSramDly)
  if (WrSram &&WR_flag)
    begin
         if(!Addr[5])
           begin
                #15 SramMem[Addr]=Data;
               // $display("mem[%h]=%h",Addr,Data);
                $fwrite(file,"mem[%h]=%h    ",Addr,Data);
                if(Addr[0]&&Addr[1])  $fwrite(file,"\n");
           end
         else
           begin
                $fclose(file);
                $display("Please check the txt.");
                $stop;
           end
    end

endmodule
```

参考资料：

[1] IEEE P1364.1 Draft Standard For Verilog Register Transfer Level Synthesis

[2] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Computer Society, IEEE Std 1364-1995

[3] Clifford Cummings, "Correct Methods For Adding Delays To Verilog Behavioral Models," International HDL Conference 1999 Proceedings, pp. 23-29, April 1999.

[4] Mark Gordon Amald, "Verilog Digital Computer Design Algorithms into Hardware" 1999 by Prentice Hall PTR Prentice Hall Inc.

[5] Thomas D E, Moorby P R. "The Verilog Hardware Description Language." 2 second ed 1995 by Kluwer Academic Publishers

[6] Quick Works Version 7.1 User's Guide with SpDE Reference 1998

[7] CADENCE Version 9502 Verilog-XL Reference 1997

[8] Synplify- Lite Synthesis User's Guide 1997

[9] 刘宝琴 "数字电路与系统" 北京清华大学出版社，1993

[10]夏宇闻 "复杂数字电路与系统的 Verilog HDL 设计技术" 北京航空航天大学出版社，1998

作者编后记

三十三年前我从清华大学自动控制系计算技术与装置专业毕业时，国内数字逻辑电路设计刚刚才开始采用半导体电路。数字逻辑电路的分析和综合还是采用传统的方法：即先在纸上画真值表,做布尔代数化简，画波形图，画有限状态机流程图，画静态和动态卡诺图等方法来设计电路。在实验板上先用晶体三极管、二极管、电阻、电容等搭出门电路和触发器电路，做成线路板，测出各项参数。每块线路板上只有几个触发器和几个与、或、非门。再用这样的线路板来构成数字逻辑电路。以现在的眼光看，当时的数字逻辑系统是相当简陋的。做一个简单的可控制步进马达按照输入做 X 和 Y 方向的直线、斜线和圆弧运动的数字控制系统，就需要用 20 多块线路板，设计和调试这样的系统需要花费很长的时间和很多的精力。

二十年前国内逐步开始使用微处理机，中规模的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的集成电路的使用课程，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。这不仅是因为有不少实时的 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系统设计课程的老师深深感到自己身上责任的重大。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同学们，也没有给他们创造实践的机会。

1992 年我受沈校长和系领导的委托，与董金明老师一起筹建世行贷款的电路设计自动化(EDA)实验室。在有限的经费中，沈士团校长为我们挤出十五万美圆，其中三万美圆购买了一套 CADENCE 设计环境，其余的购买工作站和网络设备。其中 CADENCE 设计环境中数字设计部分由我负责。自 1995 年起，工作站和 CADENCE 软件逐步到货，由于经费有限我们没有机会到美国去学习，只好自己在工作站上一边看着参考手册一边学着干，先掌握了利用电路图输入的方法，再逐步掌握了利用 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一万门左右的复杂数字电路，提供给他们经前后仿真验证的 Verilog HDL 源代码，得到很高的评价。 我们也为我们的科研项目，小波(Wavelet)图象压缩,设计了小波卷积器和改进零修剪树(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，在 Altera 10k50 系列的 CPLD 上成功地布线和通过的后仿真，并制成了带 PCI 接口的电路板。近年来我们为航天部 501 所完成了多项五万门级以上的编码/解码和加密电路的设计都取得很好的效果。这一类设计很难用传统的电路图输入方法来设计的，这些设计的成功得益于我们对于 Verilog HDL 设计方法的掌握。

从 94 年拿到一些有关 Verilog HDL 的资料起，我就在我所讲授的研究生课程“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，我受张凤言老师的邀请，到国家教委电路教学委员会召集的华北区讨论会上作了一个三小时的有关 EDA 和 HDL 设计方法的讲座。会后张凤言老师就一直鼓励我写一本有关 HDL 设计方法的书。当时我虽然逐渐学会了一些 Verilog HDL 的设计方法，但是很不系统，也找不到好的教材作参考，总觉得很难下手。1996 年春夏之交，校园网接通，我从 Internet 网络上找到一些网址，陆续找到一些有关 Verilog HDL 的素材，但好的完整教材和光盘需要上千美圆才能购得，而我们没有外汇，也无法去购买，这使我感到很沮丧。1996 年秋，我为

QuickLogic 公司的 FPGA 芯片和设计工具的讲座作翻译后，外商送我一套 QuickLogic 设计工具，可以在 PC586 平台上运行，这套工具包括电路图输入和 Verilog HDL 输入工具、Verilog HDL 仿真器、一个小巧的综合器(Synplify)等。这套工具价格并不贵，工作平台是 PC586 机，在光盘上还有一套比较完整而简单的教学资料。我仔细地阅读了这些资料并使用了这套工具后，觉得在大多数学校里推广 Verilog HDL 设计方法是很有可能的。从此我就更积极地从 Internet 网络上找一些有关 Verilog HDL 设计方法的资料片段和有代表性的样板程序为写一本 Verilog HDL 设计方法的入门书而作准备。1997 年由于教学的需要，经过近一年的努力，98 年夏在北航出版社出版了“复杂数字逻辑与系统的 Verilog HDL 设计技术与方法”。出版后我总觉得有许多问题还没有叙述清楚，特别在算法系统的总体结构考虑和组成上与语法没有任何联系，状态机的概念引入也太突然，因此做了一些大的改动，补写了与电路结构有关的章节。加上了第一章，引入算法硬件实现的概念，也补充了较完整的 Verilog 语法作为附录可供设计参考，还加上一个上机练习的十个阶段的作业，便于同学通过自学掌握。

由于科研和实验室的各项工作很忙，我只能利用零碎时间，在我的研究生帮助下一点一点地把材料输入到计算机中并逐步加以整理。到现在两年又过去了，书总算初步有了一些新的面貌。我们使用 Verilog HDL 设计复杂数字逻辑电路已有近五年的时间，虽积累了一些经验，但水平并不高，书中谬误之处在所难免，敬请读者及时把意见反馈给我。我之所以在原学校出版的教材的基础上把这本新书推出，并起名为《从算法到硬线逻辑的实现-*复杂数字逻辑与系统的 Verilog HDL 设计技术与方法*》， 目的是想把我们在近两年来在 Verilog HDL 教学和设计方法上积累的一些新经验与读者分享，并把旧版本中许多没讲清楚的概念尽量阐述明白些，把系统设计的主要思路连贯起来。在大学生和研究生中加快 Verilog HDL 设计技术的推广，尽快培养一批掌握先进设计技术的跨世纪的人才。期望本书能在这一过程中起到抛砖引玉的作用。

读者如果配合一套可在PC586 平台上运行的Verilog HDL仿真和综合工具，如QuickLogic 公司的Spade的教学软件包(Verilog HDL版)、Mentor公司的Modelsim 等开发环境下，只要带Verilog HDL仿真器(如Verilog-XL)和综合器（如Synplify），就可以运行本书所有的Verilog HDL程序，可在这一环境下通过做各种仿真和综合的练习，学会并掌握Verilog HDL设计技术，把设计思想逐步转变为万门级的具体的电路。在掌握了基本设计技术后再购买如CADENCE、SYNOPISYS等高级的设计环境，在设计几十万门以上电路时就容易成功，为进入ASIC设计做充分的准备，不会造成浪费。希望读者能通过电子邮件跟我交流设计的心得和经验，有条件时我将在北航的网页上开设Verilog HDL设计经验交流角，共同来促进这一新设计技术的成长和发展。

批注 [fwn1]: 如果把 QuickLogic 公司的网址写出来，读者就可以自己去下载这套工具了。而且现在 QuickLogic 提供的仿真器也是 Modelsim，所以可以建议读者下载 Modelsim 和 Synplify 的评估版本。

编者
2000 年 8 月 30 日
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

目录

Verilog 硬件描述语言参考手册（按英文字母顺序查找部分）

Procedural Assignment 过程赋值语句
Procedural Continuous Assignment 过程连续赋值语句
Programming Language Interface 编程语言接口
Register 寄存器
Repeat 重复执行语句
Reserved Words 关键词
Specify 指定的块延时
Specparam 延时参数
Statement 声明语句
Strength 强度
String 字符串
Task 任务
Task Enable 任务的启动
Timing control 定时控制
User Defined Primitive 用户自定义原语
While 条件循环语句
Compiler Directives 编译器指示
Standard Compiler Directives 标准的编译器指示
Non-Standard Compiler Directives 非标准编译器指示

系统任务和函数 System task and function

标准的系统任务和函数

$display, $monitor, $strobe, $write 等
$fopen 和$fclose
$readmemb 和 $readmemh
$timeformat[(Units,Precision,Suffix,MinFieldWidth)];
$printtimescale
$stop
$finish
$time,$stime,和 $realtime
$realtobits 和 $bitstoreal
$rtoi 和 $itor

随机数产生函数

1) $random
2) $dist_chi_square
3) $dist_erlang
4) $dist_exponential
5) $dist_normal
6) $dist_poisson
7) $dist_t
8) $dist_uniform

指定块内的定时检查系统任务 Specify Block Timing Checks

1) $hold
2) $nochange
3) $period
4) $recovery
5) $setup
6) $setuphold
7) $skew
8) $width

记录数值变化的系统任务 Value Change Dump Tasks

1) $dumpfile
2) $dumpvars
3) $dumpoff;
4) $dumpon;
5) $dumpall;
6) $dumplimit(FileSize);
7) $dumpflush;

非标准的系统任务和函数

$countdrivers
$list
$input
$scope and $showscopes
$key, $nokey, $log and $nolog
$reset[
$reset_count
$reset_value
$save(“FileName”);
$incsave(“FileName”);
$restart(“FileName”);
$showvars[(NetOrRegister,...)];
$getpattern(MemoryElement);
$sreadmemb (Memory, StartAddr, FinishAddr, String, ...);
$sreadmemh (Memory, StartAddr, FinishAddr, String, ...);
$scale(DelayName); {Returns realtime}

常用系统任务和函数的详细使用说明

$display 和 $write
$fopen and $fclose
$monitor 等
$readmemb 和 $readmemh
$strobe
$timeformat
随机模型 Stochastic Modelling

$q_initialize
$q_add
$q_remove
$q_full
$q_exam

定时检查　　Timing Checks

$hold
$nochange
$period
$recovery
$setup
$setuphold
$skew
$width

记录数值变化的系统任务　Value Change Dump Tasks

$dumpfile
$dumpvars
$dumpoff;
$dumpon;
$dumpall;
$dumplimit
$dumpflush;

Command Line Options　命令行的可选项

Verilog 硬件描述语言
参考手册

一. 关于 IEEE 1364 标准

本Verilog 硬件描述语言参考手册是根据IEEE 的标准“*Verilog 硬件描述语言参考手册 1364-1995*”编写的。OVI (Open Verilog International) 根据Cadence 公司推出的Verilog LRM（1.6 版）编写了Verilog 参考手册 1.0 和 2.0 版。OVI又根据以上这两个版本制定了IEEE1364-1995 Verilog标准。在推出Verilog标准前，由于Cadence公司的 Verilog-XL 仿真器广泛使用，它所提供的Verilog LRM成了事实上的语言标准。许多第三方厂商的仿真器都努力向这一已成事实的标准靠拢。

Verilog 语言标准化的目的是将现存的通过 Verilog-XL 仿真器体现的 Verilog 语言标准化。IEEE 的 Verilog 标准与事实上的标准有一些区别。因此，仿真器有可能不完全支持以下的一些功能：

- 在UDP（用户自定义原语）和模块实例中使用数组（见Instantiation说明）。
- 含参数的宏定义（见‘define）。
- ‘undef.
- IEEE标准不支持用数字表示的强度值（见编译预处理命令）。
- 有许多Verilog-XL支持的系统任务、系统函数和编译处理命令在IEEE标准中不支持。
- 若在模块中其Net或寄存类型变量只有一个驱动，IEEE标准允许在一个指定块中， 延迟路径的最终接点可以是一个寄存器或Net类型的变量。而在此标准推出之前，对最终接点的类型有着严格得多的要求（见Specify说明）。
- 指定路径的延迟表达式最多可以达到12个延迟表达式，表达式之间需用逗号隔开。而在此标准推出之前，最多只允许六个表达式（见Specify说明）。
- 在Net类型变量的定义中，标量保留字**scalared**与矢量保留字 **vectored**的位置也做了改动。原先，保留字位于矢量范围的前面。在IEEE标准中，它应位于Net类型的后面（见Net说明）。
- 在最小-典型-最大常量表达式中，对于最小、典型与最大值的相对大小并无限制。而原先最小值必须小于或等于典型值，典型值必须小于或等于最大值。
- 在IEEE标准中，表示延迟的最小-典型-最大表达式不必括在括号里。而原先，它必需括在括号里。

二. Verilog 简介

在Verilog HDL 中，我们可通过高层模块调用低层和基本元件模块，再通过线路连接（即下文中的NET）把这些具体的模块连接在一起，来描述一个极其复杂的数字逻辑电路的结构。所谓基本元件模块就是各种逻辑门和用户定义的原语模块（即下文中的UDPs）。而所谓NET实质上就是表示电路连线或总线的网络。端口连接列表用来把外部NET连接到模块的端口(即引脚）上。寄存器可以作为输入信号连接到某个具体模块的输入口。NET和寄存器的值可取逻辑值 0 ，1 ，x（不确定）和 z（高阻）。除了逻辑值外，NET还需要有一个强度（Strength）值。在开关级模型中，当NET的驱动器不止一个时，还需要使用强度值来表示。逻辑电路的行为可以用Initial和Always 的结构和连续赋值语句，并结合设计层次树上各种层次的模块直到最底层的模块(即UDP及门)来描述。

模块中每个Initial块、Always块、连续赋值、UDP 和各逻辑门结构块都是并行执行的。而Initial及Always块内的语句与软件编程语言中的语句在许多方面非常类似，这些语句根据安排好的定时控制（如时延控制）和事件控制执行。在Begin-End块内的语句按顺序执行，而在Fork-Join块中的语句则并行执行。 连续赋值语句只可用于改变NET的值。寄存器类型变量的值只能在Initial及Always块中修改。Initial及Always块可以被分解为一些特定的任务和函数。PLI （即可编程语言接口的英语缩写）是完整的Verilog语言体系的一个的组成部分，利用PLI便可如同调用系统任务和函数一样来调用C语言编写的各种函数。

编译

Verilog的原代码通常键入到计算机的一个或多个文本文件上。然后把这些文本文件交给Verilog编译器或解释器处理，编译器或解释器就会创建用于仿真和综合必需的数据文件。有时候，编译完了马上就能进行仿真，没有必要创建中间数据文件。

三. 语法总结

典型的 Verilog 模块的结构：

```
module M (P1, P2, P3, P4);
 input P1, P2;
 output [7:0] P3;
 inout P4;
 reg [7:0] R1, M1[1:1024];
 wire W1, W2, W3, W4;
 parameter C1 = “This is a string”;
   initial
        begin : 块名
           // 声明语句

        end

    always @ (触发事件)
         begin
          // 声明语句
```

```
        end
// 连续赋值语句..
assign W1 = Expression;
wire  (Strong1, Weak0) [3:0] #(2,3) W2 = Expression;
// 模块实例引用
   COMP U1 (W3, W4);
   COMP U2 (.P1(W3), .P2(W4));

 task T1;   //任务定义
    input A1;
    inout A2;
    output A3;
      begin
        // 声明语句

      end
  endtask

  function [7:0] F1;     //函数定义
    input A1;
      begin
        // 声明语句
         F1 = 表达式;
      end
  endfunction

endmodule       //模块结束
```

声明语句：

```
  #delay
   wait (Expression)
  @(A or B or C)
  @(posedge Clk)

     Reg= Expression;
     Reg <= Expression;

   VectorReg[Bit] = Expression;
   VectorReg[MSB:LSB] = Expression;
   Memory[Address] = Expression;
    assign Reg = Expression
   deassign Reg;

   TaskEnable(...);
   disable TaskOrBlock;
  EventName;

    if (Condition)
```

```
  ...
else if (Condition)
    ...
else
   ...

case (Selection)
  Choice1 :
    ...
  Choice2, Choice3 :
    ...
  default :
    ...
 endcase

for (I=0; I<MAX; I=I+1)
     ...
    repeat (8)
     …

  while (Condition)
      ...

  forever
       …
```

上面简要语法总结可供读者快速参照，请注意其语法表示方法与本指南中其他地方不同。

四. 编写 Verilog HDL 源代码的标准

编写 Verilog HDL 源代码应按标准进行，其标准可分成两种类别。第一种是语汇代码的编写标准，标准规定了文本布局，命名和注释的约定，其目的是为了提高源代码的可读性和可维护性。第二种是综合代码的编写标准，标准规定了 Verilog 风格，其目的是为了避免常常碰到的不能综合和综合结果存在缺陷的问题，也为了在设计流程中及时发现综合中会发生的错误。

下面列出的代码编写标准可根据所选择的工具和个人的爱好自行作一些必要的改动。

语汇代码的编写标准：

- 每一个Verilog源文件中只准编写一个模块，也不要把一个模块分成几部分写在几个源文件中。
- 源文件的名字应与文件内容有关，最好一致（例 ModuleName.v）。
- 每行只写一个声明语句或说明。
- 如上面的许多例子所示，用一层层缩进的格式来写。
- 用户定义变量名的大小写应自始至终一致（例如，变量名第一个字母大写）。
- 用户定义变量名应该是有意义的，而且含有一定的有关信息。而局部名（例如循环变量）可以是简单扼要。

- 通过注释对Verilog源代码作必要的说明，当然没有必要把Verilog源代码已能说明的再注释一遍，对接口（例如模块参数、端口、任务、函数变量）作必要的注释很重要。。
- 尽可能多地使用参数和宏定义，而不要在源代码的语句中直接使用字母、数字和字符串。

可综合代码的编写标准：

- 把设计分割成较小的功能块，每一块用行为风格去设计这些块。除了设计中对速度响应要求比较临界部分外，都应避免使用门级描述。
- 应建立一个定义得很好的时钟策略，并在Verilog源代码中清晰地体现该策略（例如采用单时钟、多相位时钟、经过门产生的时钟、多时钟域等）。保证在Verilog源代码中时钟和复位信号是干净的（即不是由组合逻辑或没有考虑到的门产生的）
- 要建立一个定义得很好的测试（制造）策略，并认真编写其Verilog代码，使所有的触发器都是可复位的，使测试能通过外部管脚进行，又没有冗余的功能等。
- 每个Verilog源代码都必须遵守并符合在Always声明语句中介绍过的某一种可综合标准模板。
- 描述组合和锁存逻辑的always块，必须在always块开头的控制事件列表中列出所有的输入信号。
- 描述组合逻辑的always块一定不能有不完全赋值，也就是说所有的输出变量必须被各输入值的组合值赋值，不能有例外。
- 描述组合和锁存逻辑的always块一定不能包含反馈，也就是说在always块中已被定义为输出的寄存器变量绝对不能再在该always块中读进来作为输入信号。
- 时钟沿触发的always块必须是单时钟的，并且任何异步控制输入（通常是复位或置位信号）必须在控制事件列表中列出。
- 避免生成不想要的锁存器。在无时钟的always块中，由于有的输出变量被赋了某个信号变量值,而该信号变量没在该always块的电平敏感控制事件中列出，这会在综合中生成不想要的锁存器。
- 避免不想要的触发器。在时钟沿触发的always 块中，用非阻塞的赋值语句对寄存器类型的变量赋值，综合后就会生成触发器；或者当寄存器类型的变量在时钟沿触发的always 块中经过多次循环它的值仍保持不变，综合后也会生成触发器。
- 所有内部状态寄存器必须是可复位的，这是为了使 RTL 级和门级描述能够被复位成同一个已知的状态以便进行门级逻辑验证。（这并不适用于流水线或同步寄存器）
- 对存在无效状态的有限状态机和其他时序电路（例如，四位十进制计数器有六个无效状态），如果想要在这些无效状态下，硬件的行为也能够完全被控制，那么必须用Verilog明确地描述所有的二的N次幂种状态下的行为，当然也包括无效状态。只有这样才能综合出安全可靠的状态机。
- 一般情况下，在赋值语句中不能使用延迟，使用延迟的赋值语句是不可综合的，除了在Verilog的RTL级描述中需要解决零延迟时钟的倾斜问题是个例外。
- 不要使用整型和time型寄存器，否则将分别综合成32位和64位的总线。
- 仔细检查Verilog代码中使用动态指针（例如用指针或地址变量检索的位选择或存储单元）、循环声明或算术运算部分，因为这类代码在综合后会生成大量的门，而且很难进行优化。

五. 设计流程

用 Verilog 和综合工具设计 ASIC 或复杂 FPGA 的基本流程如下：
围绕着设计流程作多次反复是必要的，但下面的流程没有对此加以说明。而且设计流程必须根据所设计的器件的和特定的应用作必要的改动。

1 系统分析和指标的确定
2 系统划分
    2.1 顶级模块
    2.2 模块大小估计
    2.3 预布局

3 模块级设计，对每一模块：
    3.1 写 RTL 级 Verilog
    3.2 综合代码检查
    3.3 写 Verilog 测试文件
    3.4 Verilog 仿真
    3.5 写综合约束、边界条件、层次
    3.6 预综合以分析门的数量和延时

4 芯片综合
    4.1 写 Verilog 测试文件
    4.2 Verilog 仿真
    4.3 综合
    4.4 门级仿真

5 测试
    5.1 修改门级网表以便进行测试
    5.2 产生测试向量
    5.3 对可测试网表进行仿真

6 布局布线以使设计的逻辑电路能放入芯片
7 布局布线后仿真、故障覆盖仿真、定时分析

Verilog 硬件描述语言参考手册（按字母顺序查找部分）

Always 声明语句

包含一个或一个以上的声明语句(如：进程赋值语句、任务启动、条件语句、case语句和循环)，在仿真运行的全过程中，在定时控制下被反复执行。

语法

```
always
    声明语句
```

在程序中位于何处：

module- <here> -endmodule

规则：

在always块中被赋值的只能是寄存器类型的变量，如 reg, integer, real, time, realtime。每个always在仿真一开始时便开始执行，在仿真的过程中不断地执行，当执行完always块中最后一个语句后，继续从always的开头执行。

注意！

如果always块中包含有一个以上的语句，则这些语句必须放在 begin_end或 fork_join块中。如果always中没有时间控制，将会无限循环。

可综合性问题：

always声明语句是用于综合过程的最有用的Verilog声明语句之一，然而always语句经常是不可综合的。为了得到最好的综合结果，always块的Verilog 程序应严格按以下的模板来编写。

```
always @ (Inputs)   // 所有的输入信号都必须列出，在它们之间插入逻辑关系词 or
begin
... …..   // 组合逻辑关系
end

always @(Inputs)      // 所有的输入信号都必须列出，在它们之间插入逻辑关系词 or

if (Enable)
begin
……....    //锁存动作
end

always @(posedge Clock) // Clock only
begin
……    // 同步动作
end

always @(posedge Clock or negedge Reset)
// Clock and Reset only
begin
if (!Reset)         //测试异步复位电平是否有效
……..              // 异步动作
else
…….              // 同步动作
end                 // 可产生触发器和组合逻辑
```

举例说明：

下面是一个寄存器级 always 的例子

```
always @(posedge Clock or negedge Reset)
    begin
       if (!Reset) // Asynchronous reset
              Count <= 0;
       else
       if (!Load) // Synchronous load
              Count <= Data;
```

```
      else
            Count <= Count + 1;
   End
```

下面是一个描述组合逻辑电路的always块的例子：

```
always @(A or B or C or D)
  begin
         R = {A, B, C, D}
         F = 0;
      begin : Loop
           integer I;
             for (I = 0; I < 4; I = I + 1)
                if (R[I])
                begin
                   F = I;
                   disable Loop;
                end
      end // Loop
 end
```

还请参阅：
Begin, Fork, Initial, Statement, Timing Control

Assign 连续赋值声明语句

每当表达式中 NET（即连线）或寄存器类型变量的值发生变化时，使用连续赋值声明语句就可在一个或更多的电路连接中创建事件。

语法：{either}
assign [Strength] [Delay] NetLValue = Expression,
NetLValue = Expression,
...;
NetType [Expansion] [Strength] [Range] [Delay]
NetName = Expression,
NetName = Expression,
...; {See Net}

NetLValue = {either}
NetName
NetName[ConstantExpression]
NetName[ConstantExpression: ConstantExpression]
{ NetLValue,...}

在程序中位于何处：
module-<HERE>-endmodule

规则：
两种形式的连续赋值语句效果相同。
在连续赋值声明语句之前，赋值语句左边的 NET（即连线类型的变量）必须明确声明。

注意！
连续赋值并不等同于进程连续赋值语句，虽然它们相似。确保把 assign 放在正确的地方。连续赋值语句必须放在任何initial和always块外。进程连续赋值语句可放在该语句被允许放的地方执行（在initial 、always、task、 function等块内部）。

可综合性问题：
- 综合工具不能处理连续赋值语句中的延迟和强度，在综合中被忽略。请用综合工具指定的定时约束来代替。
- 连续赋值语句将被综合成为组合逻辑电路。

提示：
用连续赋值语句去描述那些用简洁的表达式就能够很容易表达的组合逻辑电路。函数能够用来构建表示式。在描述较复杂的组合逻辑电路方面，用always块比用许多句分开的连续赋值语句更好，而且在仿真的速度更快一些。当Verilog需要电路连线时，可用连续赋值语句把寄存器的值传送到电路连线上（即NET上）。例如，把一个initial块中产生的测试激励信号加到一个实例模块的输入输出端口。

举例说明：
```
wire cout, cin;
wire [31:0] sum, a, b;
  assign {cout, sum} = a + b + cin;

wire enable;
reg [7:0] data;
wire [7:0]  #(3,4)  f = enable ? data : 8'bz;
```

还请参照：
Net、Force、进程连续赋值语句

Begin 声明语句

用于把多个声明语句组合起来成为一个语句,而其中每个声明语句的执行是按顺序。Verilog语法经常严格要求只有一个声明语句，例如always就是这样。如果always需要有多个声明语句，那么这些声明语句必须被包含在一个begin-end块中。

语法
```
  begin [: Label
       [ Declarations...]]
      Statements...
  End

Declaration = {either} Register Parameter Event
```

在程序中位于何处：
请参照 Statement 的说明

规则：
begin-end块必须包含至少一个声明语句。声明语句在begin-end块中被顺序执行。定时控制

是相对于前一声明语句的。当最后的声明语句执行完毕后，begin-end块便结束。Begin-end和fork-join块可以自我嵌套或互相嵌套。如果begin-end块包含局部声明，则它必须被命名（即必须有一个标识）。如果要禁止（disable）某个begin-end块，那么被禁止的begin-end块必须有名字。

注意！
Verilog LRM 允许begin-end块在仿真时被交替执行。这就是说如果begin-end块包含两个相邻且其间没有时间控制的声明语句时，仿真器仍有可能在同一时刻在这两个语句之间执行另一个进程的部分语句（例如另一个always块中的语句）。这就是Verilog 语言如果不加约束的话，便不能与硬件有确定对应关系的原因。

提示:
甚至在并不需要局部声明，也不想禁止Begin-end块时，也可以对该Begin-end块加标识命名，以提高其可读性。给不用在别处的寄存器作局部声明，能使声明的意图变得清楚。

举例说明：

```
initial
   begin : GenerateInputs
           integer I;
           for (I = 0; I < 8; I = I + 1)
           #Period {A, B, C} = I;
   end

initial
      begin
          Load = 0; // Time 0
          Enable = 0;
          Reset = 0;
          #10 Reset = 1; // Time 10
          #25 Enable = 1; // Time 35
          #100 Load = 1; // Time 135
      end
```

还请参照：

Fork, Disable, Statement.

Case 声明语句:

如果 case 控制表达式与标号分支表达式相等，则执行该分支的声明语句。

语法：

```
CaseKeyword ( Expression)
Expression,... : Statement {Expression may be variable}
Expression,... : Statement
                ... {Any number of cases}
      [default [:] Statement] {Need not be at the end}
endcase

CaseKeyword = {either} case casex casez
```

在程序中位于何处：

请参照 Statement 的说明

规则：

- 不确定值（Xs）和高阻值（Zs）在casex声明语句中，以及（Zs ）在casez声明语句的表达式匹配中都意味着“不必考虑”。
- 在case语句中最多只允许有一个default项。当没有一个分支标号表达式能与case表达式的值相等时，便执行default项。（标号是位于冒号左边的一个表达式或用逗号隔开的几个表达式，标号也可以是保留字default,在其后面可以跟冒号也可以不跟冒号。）
- 如果某标号是用逗号隔开的两个或两个以上表达式，只要其中任何一个表达式与case表达式的值相等时，就可执行该标号的操作。
- 如果没有一个标号表达式与case表达式的值相等，又没有default声明语句，该case声明语句没有任何作用。

注意：

- 如果在标号分支中有一个以上的声明语句，这些声明语句必须放在一个begin-end或fork-join块中。
- 只有第一个与case表达式的值相等的标号分支才被执行。Case语句的标号并不一定是互斥的，所以当错误地重复使用相同的标号时，Verilog编译器不会提示出错。
- Casex 或 casez 声明语句的语法是用保留字 endcase 作为结束，而不是用 endcasex 或 endcasez 来结束。
- 在 casex 声明语句的表达式中的 X（不定值）或Z（高阻值）可以和任何值相等，casez 中的 Z 也是如此。这有可能会给仿真结果带来混乱。

可综合性问题：

Case 声明语句中的赋值语句通常被综合成多路器。如果变量（如寄存器或 Net 类型）被用作 Case 语句的标号，它就会被综合成优先编码器（priority encoders）。

在一个无时钟触发的 always 块中，如有不完整的赋值（即对某些输入信号的变化其输出仍保持不变，未能及时赋值），它将被综合成透明锁存器。

在一个有时钟触发的 always 块中，如有不完整的赋值，它将被综合成循环移位寄存器。

提示：

- 为了使仿真能顺利进行，常常用 default 作为 case 声明的最后一个分支，以控制无法与标号匹配的 case 变量。
- 通常情况下用casez比用casex更好一些，因为X的存在可能会导致仿真出现令人误解和混乱的结果。

- 在casex 和 casez声明的标号中用“？”来代替“Z”比较好，因为这样做比较清楚，是一个无关项，而不是一个高阻项。

举例说明：

```
case (Address)
     0 : A <= 1;       // Select a single Address value
     1 : begin         // Execute more than one statement
          A <= 1;
          B <= 1;
         end
 2, 3, 4 :   C <= 1;   // Pick out several Address values
default :              // Mop up the rest
        $display("Illegal Address value %h in %m at %t", Address, $realtime);
endcase

casex (Instruction)
  8'b000xxxxx : Valid <= 1;
  8'b1xxxxxxx : Neg <= 1;
  default
      begin
         Valid <= 0;
          Neg <= 0;
      end
endcase

casez ({A, B, C, D, E[3:0]})
    8'b1??????? : Op <= 2'b00;
    8'b010????? : Op <= 2'b01;
    8'b001???00 : Op <= 2'b10;
          default : Op <= 2'bxx;
endcase
```

还请参照：
If

Comment　注释语句

注释应该位于 Verillog 源代码文件中。

语法

单行注释
//
多行注释
/* ... */

在程序中位于何处：
可以放在源代码的几乎任何地方，但是注意不能把运算符、数字、字符串、变量名和关键字分开。

规则：
单行注释以两个斜杠符开始，结束于该行的末尾 。
多行注释以“/*” 符开始，中间可能有多行，结束于“*/” 符。
多行注释不能嵌套，但是，在多行注释中可以有单行注释，但在这儿它没有别的特殊含义。

注意：
/* ... /* ... */ ... */- 这样的注释会出现语法错误，要注意注释符的匹配。

提示：
建议在源代码文件中自始至终用单行注释。只有在必需注释一大段的地方才用多行注释，例如在代码的开发和调试阶段，常需要详细地注释。

举例说明：
```
// This is a comment
/*
    So is this - across three lines
*/
module ALU /* 8-bit ALU */ (A, B, Opcode, F);
```

请参照：
Coding Standard 编码标准

Defparam 定义参数声明语句

编译时可重新定义参数值。如果是分层次命名的参数，可以在该设计层次内或外的任何地方重新定义参数。

语法：
```
Defparam ParameterName = Constant Expression
ParameterName = ConstantExpression,

…;
```

在程序中位于何处：
module-<HERE>-endmodule

可综合性问题：
一般情况下是不可综合的。

提示：
不要使用 defparam 声明语句！该声明语句过去常用于布线后的时延参数反标中，但现在时延参数反标一般用指定模块和编程语言接口（PLI）来做。在模块的实例引用时可用“#”号后跟参数的语法来重新定义参数。

举例说明：
```
‘timescale 1ns / 1ps
module LayoutDelays;
   defparam Design.U1.T_f = 2.7;
   defparam Design.U2.T_f = 3.1;
    ...
```

```
endmodule
module Design (...);
   ...
   and_gate U1 (f, a, b);
   and_gate U2 (f, a, b);
   ...
endmodule

module and_gate (f, a, b);
  output f;
  nput a, b;
  parameter T_f = 2;
    and #(T_f) (f,a,b);
endmodule
```

还请参照：
Name, Instantiation, Parameter

Delay　时延

可以为 UDP 和门的实例指定时延，也可以为连续赋值语句和线路连接指定时延。时延是在网表中线路连接和元件传输时延的模型。

语法：
{either}
DelayValue
#(DelayValue[, DelayValue[, DelayValue]]) {Rise,Fall,Turn-Off}
DelayValue = {either}
UnsignedNumber
ParameterName
ConstantMinTypMaxExpression

在程序中位于何处：
请参照：连续赋值语句、实例引用、线路连接。

规则：
- 如果只给出一个延迟，则它既表示上升延迟也表示下降延迟 （即从0转变到1或从1转变到0的时延），并且还表示关闭延迟（如果电路中有这样开关）。
- 如果给出两个延迟值，则第一个表示上升延迟，第二个表示下降延迟，除了tranif0, tranif1, rtranif0和rtranif1外，第一个值也可表示接通延迟，第二个表示关闭延迟。
- 如果给出三个延迟，第三个延迟表示关闭延迟（转变到高阻），除了三态电路外，第三个延迟表示电荷衰减时间。
- 延迟到X表示最小的指定延迟。
- 对于向量，从非零到零的转变被看作下降，转变到高阻被看作关闭，其余的变化被看作是上升。

注意！
许多工具要求MinTypMax延迟表达式必须用括号括起来。例如 #（1：2：3）是合法的，而 #1：2：3是非法的。

可综合性问题：
一般综合工具不考虑延迟。综合后网表中的延迟是由综合工具的命令项强制生成的，如在综合工具中可设置本次设计综合生成的门级电路所允许的最高时钟频率。

提示：
指定块的延迟（即线路路径延迟）通常是一种更加精确的延迟建模方法，可提供延迟计算机制和布线后反标信息。

还请参照：
线路连接，实例引用，连续赋值，Specify，定时控制 等声明语句

Disable 禁止

在运行激活的任务或命名的块时 Disable 能使在所在块执行完毕以前，终止该块的执行。

语法：
disable BlockOrTaskName;

在程序中位于何处：
请参照 Statement.

规则：
- 禁止（disable）命名块（即定义了名称的 begin_end 或 fork_joink 块）或任务便禁止了所有由该块或该任务激活的任务，直达该块或该任务层次树的底层。继续执行禁止（块或任务）语句后的声明语句。
- 命名块或任务可以通过其内部的禁止声明语句实现自我禁止
- 当一个任务被禁止时，以下内容未被指定：任何一个输出值或输入输出值；尚未起作用的非阻塞赋值语句、赋值和强制声明语句所预定的事件。
- 函数不能被禁止。

注意！
如果一个任务被自我禁止，这跟任务返回不一样，因为输出未定义。

可综合性问题：
只有当命名块或任务自我禁止时，禁止才是可综合的，一般情况下是不可综合的。

提示：
用禁止作为一种及早跳出任务的方法，用来跳出循环或继续下一步循环。

举例说明：

```
begin :  Break      //命名 Break 块
     forever
      begin :  Continue     //命名 Continue 块

          ...
        disable Continue;     // Continue with next iteration
          ...
```

```
        disable Break;      // Exit the forever loop
            ...
    end     // Continue
end         // Break
```

Errors　错误

下面列出的是编写 Verilog 源代码时最常犯的错误。前面的五个错误大约占所有错误的 50%。

最容易犯的五大错误：

- 进程赋值语句的左侧变量没有声明为寄存器类型。
- Begin – end 声明语句忘了配套。
- 写二进制数时忘了标明数基（即‘b）。这样，在编译时会把它们当作十进制数来处理。
- 编译引导语句用了错误的撇号，应该用向后的撇号也就是用表示重音的撇号；而表示数基的撇号，应该是普通的撇号，也就是反向的逗号。
- 在声明语句的末尾忘了写上分号。

其他常犯的错误：

- 在定义任务或函数时，试图在任务或函数名后用括号来定义变量。
- 在调试时，忘了在测试文件中引用实例模块。
- 使用进程连续赋值语句而没有使用连续赋值语句（即赋值语句用错了地方）
- 把保留字作为标识符（例如用 xor 做标识符）。
- Always 块忘了声明定时控制（导致无休止的循环）。
- 在事件控制列表中错误地使用了逻辑或操作符（即||）而没有使用或保留字or，例如把 always @(a or b),写成了always @(a||b)。
- 用缺省定义的 wire 类型变量来做矢量端口的连线。
- 模块实例引用时端口的连接次序搞错。
- 在嵌套的 if-else 语句中 begin-end 配套错误。
- 错误地使用等号。“ = ”用于赋值，“ = = ”用于作数值比较，“ = = = ”用于需要对 0、1、X、Z 这四种逻辑状态作准确比较的场合 。

Event　事件

在行为模型中 Events 可以用来描述通信和同步。

语法：

```
event Name ,...; {Declare the event}
-> EventName; {Trigger the event}
```

语法

事件名，...；（事件声明）

->事件名　　（触发事件）

在程序中位于何处：

请参照为 -> 所作的声明语句。

事件声明语句可以放在下面这些地方：

```
module-<HERE>-endmodule
begin : Label-<HERE>-end
fork : Label-<HERE>-join
task-<HERE>-endtask
function-<HERE>-endfunction
```

规则：
事件没有值，也没有延迟，它们仅被事件触发声明所触发，由沿敏感定时控制启动检测。

可综合性问题：
通常是不可综合的。

提示：
在测试文件和系统级模块中，命名事件可用于同一个模块的不同 always 块间或不同模块（用层次名）的 always 块间传递信息。

举例说明：

```
event StartClock, StopClock;
always
   fork
      begin : ClockGenerator
              Clock = 0;
              @StartClock
                forever
                    #HalfPeriod Clock = !Clock;
      end
      @StopClock disable ClockGenerator;
   join


initial
    begin : stimulus
              ...
       -> StartClock;
              ...
       -> StopClock;
              ...
       -> StartClock;
              ...
       4-> StopClock;
   end
```

还请参阅：
定时控制

Expression 表达式

表达式可以通过一系列的操作符、变量名、数字以及次级表达式来算出一个值。其中常量表达式是一种其值可在编译过程中计算出来的表达式。标量表达式的值是一比特二进制数。时间延迟可以用最小-典型-最大（即 MinTypMax）表达式来表示。

语法：

Expression = {either}	表达式 = {以下任取其一}
Primary	基本表达式
Operator Primary {unary operator}	运算符 基本表达式 {单目运算符}
Expression Operator Expression {binary operator}	表达式 运算符 表达式 {双目运算符}
Expression ? Expression : Expression	表达式 ？ 表达式 ： 表达式
String	字符串

Primary = {either}	基本表达式 = {以下任取其一}
Number	数字
Name {of parameter, net, or register}	变量名 {参数，网络，或者寄存器的}
Name[Expression] {bit select}	变量名[表达式] {位选择}
Name[Expression: Expression] {part select}	变量名[表达式：表达式] {部分位选择}
MemoryName[Expression]	存储器名[表达式]
{ Expression,...} {concatenation}	{表达式，……} {位拼接}
{ Expression{ Expression,...}} {replication}	{表达式{表达式,……}} {复制}
FunctionCall	函数调用
(MinTypMaxExpression)	（MinTypMax 表达式）
{MinTypMax expressions are used for delays}	{ MinTypMax 表达式用于延迟}
MinTypMaxExpression = {either}	MinTypMax 表达式 = {任取其一}
Expression	表达式
Expression: Expression: Expression	表达式：表达式：表达式

规则：

- 只有矢量类型的 NET 和寄存器、整数及时间类型变量才允许选取某位及某些位。
- 某些位的选取必须将高位列在冒号的左侧，低位列在右侧。（最高位是在 NET 或寄存器类型声明中位于冒号左边的数值。）
- 某位或某些位的选取若其中包含 X 或 Z 的位，或超出位的定义范围，在编译时可能会也可能不会被认定为是错误的。如果不被认定是错的，编译器会给出一个值为 X 的表达式。
- 没有为存储器建立某位或某些位选取的机制。
- 当整型常量在表达式中作为操作数时，未标明进制的有符号数(例如-5)与标明进制的有符号数(例如-‘d5)是有所区别的。前者被视作一个有符号数，而后者被视作一个无符号数。

注意！

许多工具要求在常量 MinTypMax 表达式中必须指定最小、典型和最大延迟值 (例如：min<=typ<=max)。

举例说明：

```
A + B
!A
(A && B) || C
A[7:0]
B[1]
-4'd12/3                          // 是一个很大的正数
"Hello" != "Goodbye"          // 此表达式为真（1）
$realtobits®;                     //  系统函数调用
{A, B, C[1:6]}                    // 位拼接（8 位）
1:2:3                             //最小-典型-最大表达式
```

还请参阅：

延迟，函数调用，变量名，数字，操作数。

For 循环声明语句

一般用途的循环语句。允许一条或更多的语句能被重复地执行。

语法：

```
for ( RegAssignment;  {initial assignment}
      Expression;     {loop condition}
      RegAssignment)  {iteration assignment}
    Statement
```

```
RegAssignment = RegisterLValue = Expression   寄存器赋值 = 寄存器值 = 表达式
RegisterLValue = {either}                     寄存器值 = {任取其一}
RegisterName                                  寄存器名
RegisterName[ Expression]                     寄存器名[表达式]
RegisterName[ ConstantExpression: ConstantExpression]
                                              寄存器名[常量表达式：常量表达式]
Memory[ Expression]                           存储器[表达式]
{ RegisterLValue,...}                         {寄存器值，.....}
```

在程序中位于何处：

请参照 Statement 说明。

规则：

当 for 循环开始执行时，循环计数变量已赋于初始值。在每一次循环执行之前（包括第一次），都必须首先检查表达式的值；如果它为假（即为 0、X、或 Z），则立刻退出循环。而在每一次循环重复执行之后，都要对迭代次数寄存器重新赋值。

注意！

不要使用位宽小的 reg 类型变量作为循环变量。在测试存有负数值的寄存器变量时要格外注意。由于加减操作是可替换的，并且 reg 类型变量被看作是无符号数，所以循环表达式可能永远不会为假，从而导致循环无限止地进行。

```
reg [2:0] i;                //i 始终界于 0 至 7 之间
...
for ( i= 0; i<8; i=i+1 )      //循环永远不会停止
...
for ( i=-4; i<0; i=i+1 )     //  循环不可能执行
...;
```

在以上这些情况中，应将循环变量 i 定义为整型。

可综合性问题：

如果循环的边界是固定的，那么在综合时该循环语句被认为是重复的硬件结构。

举例说明：

```
V = 0;
for ( I = 0; I < 4; I = I + 1 )
begin
F[I] = A[I] & B[3-I];     //    四个独立的与门
V = V ^ A[I];             //  四个级连的异或门
end
```

还请参阅：

Forever, Repeat, While 语句的说明。

Force 强迫赋值

类似于进程连续赋值语句，可对 Net 和寄存器类型变量实行强制赋值。常用于调试。

语法：

{either}	{任取其一}
force NetLValue = Expression ;	force 网络参数值=表达式;
force RegisterLValue = Expression ;	force 寄存器值=表达式;
{either}	{任取其一}
release NetLValue;	release 网络参数值;
release RegisterLValue;	release 寄存器值
NetLValue = {either}	网络参数值={任取其一}
NetName	网络变量名

{NetName,...}	{网络变量名}
RegisterLValue = {either}	寄存器值={任取其一}
RegisterName	寄存器变量名
{RegisterName,...}	{寄存器变量名}

在程序中位于何处：
请参照声明语句

规则：

- 不能对网络变量或寄存器变量的某位或某些位实行强制赋值或释放。 **force**具有比进程连续赋值声明语句更高的优先级。force 将会一直发挥作用直到另一个force对同一Net变量或寄存器变量执行强迫赋值，或者直到这个Net变量或寄存器变量被释放。
- 当作用在某一寄存器上的 force 被释放，寄存器并无必要立刻改变其值。如果此时没有进程连续赋值对这个寄存器赋值，则强制赋入的值会一直保留到下一个进程赋值语句的执行。
- 当作用在某个 Net 变量上的 force 被释放，该 Net 变量的值将由它的驱动决定，其值有可能会立刻更新。

可综合性问题：
强迫赋值语句是不可综合的。

提示：
强迫赋值常用于测试文件的编写，调试时常需要强制对某些变量赋值。不能用于模块的行为建模（此时应使用连续赋值语句）。

举例说明：

```
force   f = a && b;
...
release f;
```

还请参照：
进程连续赋值语句。

Forever 声明语句

使一个或一个以上语句无限循环地执行。

语法：

```
forever     Statement
```

在程序中位于何处：

请参阅 Statement 说明。

注意！
forever 循环应包括定时控制或能够使其自身停止循环，否则循环将无限进行下去。

可综合性问题：
一般情况下是不可综合的。如果 forever 循环被@(posedge Clock)形式的时间控制打断，则是可综合的。

提示：
forever 在测试模块中描述时钟时很有用。常用 disable 来跳出循环。

举例说明：

```
initial
    begin : Clocking
        Clock = 0;
        forever
        #10 Clock = !Clock;
    end

initial
    begin : Stimulus
        . . .
        disable Clocking;   // 停止时钟
    end
```

还请参阅：
For, Repeat, While, Disable 说明。

Fork 声明语句

可将多个语句集合在一个块中，以使它们能被并发地执行。

语法：

```
fork [ : Label                          fork   [:块名
[ Declarations...]]                             [块内声明语句……]]
Statements...                            语句……
join                                    join
Declaration = {either}                  块内声明语句={任选其一}
Register                                寄存器变量
Parameter                               参数
```

Event 事件

在程序中位于何处：
请参照 Statement 说明。

规则：
fork-join 块必须至少包括一条语句。Fork-join 块里的语句是并发执行的，因此 Fork-join 块内语句的顺序是无所谓的。时间控制是相对于块的开始时刻的。 当 fork-join 块里所有的语句执行完毕后，块也就执行完毕了。Begin-end 和 fork-join 块可以自身嵌套或互相嵌套。
如果想在某 fork-join 块内包含块内局部声明语句，那么必须对该块命名（即该块必须有一个标识符号）。如果想要禁止某 fork-join 块的运行，则该块必须已被命名。

可综合性问题：
fork 语句不可综合。

注意！
Fork-join 语句在描述并发形式的行为时很有用。

举例说明：
```
initial
    fork : stimulus
        #20 Data = 8'hae;
        #40 Data = 8'hxx; // 本句最后执行
        Reset = 0;        // 本句最先执行
        #10 Reset = 1;
    join                  // 在第 40 个时间单位时结束
```

还请参照：
Begin, Disable, Statement

Function 函数

用于把多个语句组合在一起，来定义新的数学或逻辑函数。函数是在模块内部定义的，并且通常在本模块中调用，也能根据按模块层次分级命名的函数名从其他模块调用。

语法：

function [RangeOrType] FunctionName;	function [返回值的类型或范围] 函数名；
Declarations...	端口声明...
Statement	语句
endfunction	endfunction

RangeOrType = {either}　　返回值的类型或范围={任取其一}
integer　　整数、
real　　实数
time　　时间
realtime
Range
Range = [ConstantExpression: ConstantExpression]　范围=[常量表达式：常量表达式]
Declaration = {either}　　端口声明={任取其一}
input [Range] Name,...;　　input [范围] 变量名，……;
Register
Parameter
Event

在程序中位于何处：
module　-<HERE>- endmodule

规则：
- 函数必须至少含有一个输入变量。它不能有任何输出或输入/输出双向变量。
- 函数不能包含时间控制语句（如延迟#，事件控制@ 或等待 wait）。
- 函数是通过对函数名赋值的途径返回其值的，（就好比它是一个寄存器）。
- 函数不能启动任务。
- 函数不能被禁用。

注意！
- 函数的输入变量不能象模块的端口那样列在函数名后的括弧里；在声明输入时把这些输入端口列出即可。
- 如果函数包含一条以上的语句，这些语句必须包含在 begin-end 或 fork-join 块中。

可综合性问题：
函数的每一次调用都被综合为一个独立的组合逻辑电路块。

举例说明：
```
function [7:0] ReverseBits;
    input [7:0] Byte;
    integer i;
    begin
        for (i = 0; i < 8; i = i + 1)
        ReverseBits[7-i] = Byte[i];
    end
endfunction
```

还请参阅：
Function Call, Task 语句的说明。

Function Call　函数调用

函数的调用可返回一个供表达式使用的值。

语法：
FunctionName (Expression，...);　　函数名（表达式，……）;

在程序中位于何处：
请参照 Expression 说明。

规则：
函数必须至少含有一个输入变量，所以函数调用时总是至少含有一个表达式。

可综合性问题：
函数的每一次调用在综合后都会生成一个独立的组合逻辑电路块。

举例说明：
Byte = ReverseBits (Byte);

还请参阅：
Function, Expression, Task Enable 说明。

Gate　门

Verilog 已有一些建立好的逻辑门和开关的模型。在所设计的模块中能通过实例引用这些门与开关模型，从而对模块进行结构化的描述。

逻辑门：
and (Output, Input,...)
nand (Output, Input,...)
or (Output, Input,...)
nor (Output, Input,...)
xor (Output, Input,...)
xnor (Output, Input,...)

缓冲器与非门：
buf (Output,..., Input)
not (Output,..., Input)

三态门：

bufif0 (Output, Input, Enable)
bufif1 (Output, Input, Enable)
notif0 (Output, Input, Enable)
notif1 (Output, Input, Enable)

MOS 开关：
nmos (Output, Input, Enable)
pmos (Output, Input, Enable)
rnmos (Output, Input, Enable)
rpmos (Output, Input, Enable)

CMOS 开关：
cmos (Output, Input, NEnable, PEnable)
rcmos (Output, Input, NEnable, PEnable)

双向开关：
tran (Inout1, Inout2)
rtran (Inout1, Inout2)

双向可控开关：
tranif0 (Inout1, Inout2, Control)
tranif1 (Inout1, Inout2, Control)
rtarnif0 (Inout1, Inout2, Control)
rtranif1 (Inout1, Inout2, Control)

上拉源与下拉源：
pullup (Output)
pulldown (Output)

真值表

在这些表中，逻辑值 L 与 H 代表部分未知值。L 表示 0 或 Z，H 表示 1 或 Z。

and	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

nand	0	1	X	Z
0	1	1	1	1
1	1	0	X	X
X	1	X	X	X
Z	1	X	X	X

or	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

nor	0	1	X	Z
0	1	0	X	X
1	0	0	0	0
X	X	0	X	X
Z	X	0	X	X

xor	0	1	X	Z
0	0	1	X	X
1	1	0	X	X
X	X	X	X	X
Z	X	X	X	X

xnor	0	1	X	Z
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

buf	
Input	Output
0	0
1	1
X	X
Z	X

Not	
Input	Output
0	1
1	0
X	X
Z	X

缓冲门、非门都可以有多个输出，但这些输出值都是相同的。

Bufif0		Enable			
		0	1	X	Z
DATA	0	0	Z	L	L
	1	1	Z	H	H
	X	X	Z	X	X
	Z	X	Z	X	X

Bufif1		Enable			
		0	1	X	Z
DATA	0	Z	0	L	L
	1	Z	1	H	H
	X	Z	X	X	X
	Z	Z	X	X	X

notif0		Enable			
		0	1	X	Z
DATA	0	1	Z	H	H
	1	0	Z	L	L
	X	X	Z	X	X
	Z	X	Z	X	X

notif1		Enable			
		0	1	X	Z
DATA	0	Z	1	H	H
	1	Z	0	L	L
	X	Z	X	X	X
	Z	Z	X	X	X

PMOS RPMOS		Control			
		0	1	X	Z
DATA	0	0	Z	L	L
	1	1	Z	H	H
	X	X	Z	X	X
	Z	Z	Z	Z	Z

NMOS RNMOS		Control			
		0	1	X	Z
DATA	0	Z	0	L	L
	1	Z	1	H	H
	X	Z	X	X	X
	Z	Z	Z	Z	Z

```
cmos (W, Datain, NControl, PControl);
```
等价于:
```
nmos (W, Datain, NControl);
pmos (W, Datain, PControl);
```

规则：

- 当 nmos, pmos, cmos, tran, tranif0 和 tranif1 类型的开关开启时，信号从输入传到输出并不改变其强度。

- 当有电阻的开关，如 rnmos, rpmos, rcmos, rtran, rtranif0 和 rtranif1 类型的开关，开启时，信号从输入传到输出会按下表减小其强度：

Strength	减至
supply	pull
strong	pull
pull	weak
large	medium
weak	medium
medium	small
small	small
highz	highz

还请参阅：
UDP（用户自定义原语），Instantiation(实例引用)语句的说明。

IF 条件声明语句

根据条件表达式的逻辑值（真/假），执行两条/块语句中的一条/块。

语法：
if (Expression)　　if(表达式)
Statement　　语句
[else　　[else
Statement]　　语句]

在程序中放在何处：
请参阅 Statement 说明.

规则：
当表达式的值为非零时被认为是真，当值为零、X 或 Z 时被认为是假。

注意：
- 如果在 **if** 或 **else** 分支中有超过一条的语句需要执行，则必须用 begin-end 或 fork-join 将其包括。
- 在使用嵌套的 if-else 语句时，当 else 分支省略时，要特别注意。Else 只与离它最近的前面的那个 if 相关联。 Verilog 编译器不能判别源代码中省略的 else 分支。

可综合性问题：
- If 声明语句中的赋值语句通常被综合为多路器。在无时钟的 **always** 块中，当输入变化

时而输出仍能保持不变的那些赋值语句，将被综合为透明锁存器，而在有时钟的 **always** 块中，它们则被综合为循环锁存器。

- 在某些情况下，嵌套的 if 语句会被综合为多层的逻辑。用 case 语句可以避免出现这种情况。

提示：

如果对某些条件需要先进行测试，在这种情况下应选用嵌套的 if-else 语句。如果所有的条件优先权一致，则应选用 **case** 语句。

举例说明：

```
if (C1 && C2)
    begin
        V = !V;
        W = 0;
            if (!C3)
                X = A;
                else    if (!C4)
                            X = B;
            else
                X = C;
    end
```

还请参阅：

Case, Operators 说明

Initial 声明语句

在仿真一开始就执行并只执行一次的声明语句，可执行只包含一条语句或多条语句组成的块。

语法：

```
initial
    Statement
```

在程序中放在何处：

module-<HERE>-endmodule

可综合性问题：

initial 语句是不可综合的。

注意！
包含多个语句的 **initial** 块需要用 begin-end 或 fork-join 块将这些语句合成一块。

提示：
在仿真测试文件中可使用 **initial** 语句来描述激励。

举例说明：
下面的例子给出如何使用 **initial** 在测试文件中产生矢量：

```
reg Clock, Enable, Load, Reset;
reg [7:0] Data;
parameter HalfPeriod = 5;
   initial
      begin : ClockGenerator
         Clock = 0;
         forever
            #(HalfPeriod) Clock = !Clock;
      end
   initial
      begin
         Load = 0;
         Enable = 0;
         Reset = 0;
         #20    Reset = 1;
         #100   Enable = 1;
         #100   Data = 8'haa;
                Load = 1;
         #10 Load = 0;
         #500   disable ClockGenerator;     //停止时钟的产生。
      end
```

还请参阅：
Always 语句说明

Instantiation 实例引用

实例（instance）是模块、UDP 或门的唯一拷贝。 通过实例的引用可以生成设计的各个层次。设计的行为也能通过引用 UDP、门和其他的模块的实例，并用电路连线（Net）将它们连接起来，从结构上加以描述。
述。

语法：

{either}	{任取其一}
ModuleName	
[#(Expression,...)] ModuleInstance,...;	模块名 [#（表达式，…）] 实例模块，……;
UDPOrGateName [Strength]	
[Delay] PrimitiveInstance,...;	UDP 或门名 [强度] [延迟] 原始实例，…;
ModuleInstance =	实例模块=
InstanceName [Range] ([PortConnections])	实例名[范围]（[端口连线]）
PrimitiveInstance =	原始实例=
[InstanceName [Range]] (Expression,...)	[实例名[范围]] （表达式，……）
Range =	
[ConstantExpression: ConstantExpression]	范围 = [常量表达式：常量表达式]
PortConnections = {either}	端口连线={任取其一}
[Expression] ,... {ordered connection}	［表达式］，……{有顺序的连线}
・PortName([Expression]) ,... {named connection}	・端口名([表达式]),……{指定连线}

在程序中位于何处：

module-<HERE>-endmodule

规则：

- 命名的端口连线只能用于模块实例。
- 如果给出端口的连线次序列表，则在引用实例时其端口必须按次序与模块或门的端口一一对应。
- 如果给出命名的端口连线列表时，则在引用实例时其端口顺序是无关紧要的，但其端口的名字必须与模块的端口名字一致。
- 如果给出端口的连线次序列表，在引用实例时，其端口列表中若有两个邻近的逗号，则会因为没有表达式而导致相应端口未连线。如果给出命名的端口连线列表时，在引用实例时，其端口列表中若没有某端口的名字或虽有端口的名字但在括号内没有表达式，也会在导致该端口未连线。
- 任何表达式都可用来与输入端口相连，但输出端口只能与 Net（线路）、一位或多位的连线或这些位的拼接线相连。输入表达式生成隐含的连续赋值。
- 如果在模块实例定义时给出了范围，其含义是定义了一个含有同种的多个子实例的实例模块。如果端口表达式的位长与定义的实例模块相应端口位长（即多个同种 UDP 或门端口位数的总和）一致时，整个表达式都将与每个子实例的端口相连。如果位长不一致，太多或太少，都会出错误。
- “#”符号有两种不同的用途。它既可用来强制修正实例模块中的一个或多个参量，也可用于为 UDP 或门实例指定延迟。对于实例模块，“#”符号后的第一个表达式替代模块中声明的第一个参量，第二个表达式替代模块中声明的第二个参量，依次类推。
- 实例引用 pullup、pulldown、tran 和 rtran 这些类型的门时不允许有延迟。
- 对于 nmos、pmos、cmos、rnmos、rpmos、rcmos、tran,、rtran,、tranif0、tranif1,、rtranif0 和 rtranif1 这些类型的开关不能定义强度。

注意！

- 在按顺序的端口的列表中很容易不小心将两个端口的次序弄混。若这些端口的位宽和方

向相同，不会报告出错，只有在仿真出现错误结果后，才能找到。这类错误往往很难发现。使用命名的端口连线能避免实例模块引用中出现这类问题。

- 多个模块、UDP 或门的成组实例引用的语法是最近才加入 Verilog 语言的标准中，目前还没有工具支持这语法。

可综合性问题：

UDP 和开关的实例引用一般是不能综合的。

提示：

使用命名的端口连接方式以提高程序的可读性并减少发生错误的可能性（见前文）。

端口表达式只使用位、部分位和位拼接的变量名。如果需要，则应尽量使用独立的连续赋值语句，对实例模块引入信号。

举例说明：

UDP 实例引用：

```
Nand2 (weak1,pull0) #(3,4) (F, A, B);
```

模块实例引用：

```
Counter  U123 (.Clock(Clk), .Reset(Rst), .Count(Q));
```

在下面的两个例子中 QB 端口没有连接：

```
DFF Ff1 (.Clk(Clk), .D(D), .Q(Q), .QB());
DFF  Ff2  (Q,, Clk, D);
```

下面是在端口连线表中使用门表达式的例子：

```
nor (F, A&&B, C) // 不要这样使用
```

下面是一个多实例模块引用的例子。

```
module Tristate8 (out, in, ena);
output [7:0] out;
input [7:0] in;
input ena;

bufif1 U1[7:0] (out, in, ena);
/* 上面的一条语句等同下面 8 条语句
bufif1 U1_7 (out[7], in[7], ena);
bufif1 U1_6 (out[6], in[6], ena);
bufif1 U1_5 (out[5], in[5], ena);
bufif1 U1_4 (out[4], in[4], ena);
bufif1 U1_3 (out[3], in[3], ena);
bufif1 U1_2 (out[2], in[2], ena);
bufif1 U1_1 (out[1], in[1], ena);
bufif1 U1_0 (out[0], in[0], ena);
*/
```

endmodule

还请参阅：
Module, User Defined Primitive, Gate, Port 语句的说明。

Module　模块定义

在 Verilog 语言中，模块是层次的基本单元。模块中包括声明语句、功能描述和引用一些现存的硬件部件。有些模块只用来声明可被别的模块调用的参量，任务和函数。在这类模块中没有任何 **initial** 块、**always** 块、连续赋值语句和实例引用，因而实际上不存在相应的硬件元件与之对应 。

语法：

{either}	{任取其一}
module ModuleName [(Port,...)];	module 模块名[（端口，……）];
ModuleItems...	模块条款
endmodule	endmodule
macromodule ModuleName [(Port,...)];	macromodule 模块名[（端口,…）];
ModuleItems...	模块条款
endmodule	endmodule
ModuleItem = {either}	模块条款={任取其一}
Declaration	声明
Defparam	参数定义
ContinuousAssignment	连续任务
Instance	实例引用
Specify	详细说明块
Initial	初始化块
Always	总是执行块
Declaration = {either}	声明={任取其一}
Port	端口
Net	网络
Register	寄存器
Parameter	参量
Event	事件
Task	任务
Function	函数

在程序中位于何处：

在其它模块或 UDP 外。

规则：

- 几个模块或几个 UDP（或它们的混合）可以在一个文件中进行描述。（事实上，一个模块也可以分开在两个或更多的文件中描述，但不推荐这种做法）
- 模块也可使用关键字 **macromodule** 来定义。其语法与用关键字 **module** 来定义模块是完全一样的。
- Verilog 编译器在编译宏模块时与编译一般模块时有所不同，比如不必为宏模块实例创建层次。这样，从仿真速度或存储介质的开销两方面来说，宏模块的编译更有效率。为了达到这个目的，宏模块的编译可能受制于实现时的某些特殊条件的限制。如果遇到这种情况，宏模块将被按一般模块编译。

注意！

模块与宏摸块都以关键字 endmodule 作为结束标志。

可综合性问题：

- 每一个模块都被综合为一个独立的分层块，虽然有些工具的缺省配置规定把层次展平（为单层），但仍允许用户对综合后生成的网表层次进行控制，。
- 不是所有的工具都支持宏模块的综合。

提示：

尽量使每一个文件只包含一个模块。在大型设计中，这样做易于源代码的维护。

举例说明：

```
macromodule nand2 (f, a, b);
    output f;
    input a, b;
        nand (f, a, b);
endmodule

module PYTHAGORAS (X, Y, Z);
    input [63:0] X, Y;
    output [63:0] Z;
    parameter Epsilon = 1.0E-6;
    real RX, RY, X2Y2, A, B;
        always @(X or Y)
            begin
                RX = $bitstoreal(X);
            RY = $bitstoreal(Y);
            X2Y2 = (RX * RX) + (RY * RY);
                B = X2Y2;
                A = 0.0;
                 while ((A - B) > Epsilon || (A - B) < -Epsilon)
                    begin
```

```
                A = B;
                B = (A + X2Y2 / A) / 2.0;
            end
        end
    assign Z = $realtobits(A);
endmodule
```

还请参阅：

User Defined Primitive, Instantiation, Name 语句的说明。

Name 名字

任何用 Verilog 语言描述的“东西”都通过它的名字来识别。

语法：

Identifier	标识符
\EscapedIdentifier {terminates with white space}	\ 扩展标识符{空格表示结束}

规则：

- 标识符可由字母，数字，下划线和美元符号构成。第一个字符必须是字母或下划线，而不能是数字或美圆符号。
- 一个扩展标识符用反斜杠引出，用空格结束（空格符、制表符、回车键或换行键），并且可包含除空格外的任何可印刷的字符。反斜杠和空格并不算作标识符的部分，例如，标识符 Fred 与扩展标识符 \Fred 是相同的。
- 在 Verilog 中变量名是大小写敏感的。
- 在 Verilog 文件中，一个名字不能有多于一个以上的含义。名字的内部声明（例如在 begin-end 块中的名字）能屏蔽外部声明 (例如包含有该命名 begin-end 块的上层模块的变量声明语句)。

Hierarchical Names 分级名字

- Verilog HDL 中的每个标识符都有唯一的分级名字。这意味着任何 Net、寄存器、事件、参量、任务和函数都能通过使用它的分级名，在标识符的声明块外对它进行访问。
- 在名字层次的最上层是不需要实例引用的模块名。顶层测试模块就是一个最上层模块例子（尽管可能会有不止一个顶层模块在同一仿真中运行）。
- 在每个实例模块、命名块、任务和函数的定义时，便定义了名字层次树上的新层。

- Verilog 变量的分级名字是从顶层模块名字开始直到包含该变量的模块实例名、命名块、

任务和函数名构成，其间用小圆点隔开。

Upwards Name Referencing　向上索引名

包含两个标识符中间用点号隔开的分级名可能是下列情况中的一种：

- 当前模块所引用的实例模块中的一项。（这是向下引用）
- 顶层模块中的一项。（这是一个分级名字）
- 当前模块的父模块中的引用的实例模块中的一项。（这是向上引用。）
- 向上引用名字的第一个标识符既可能是一个模块名也可能是一个模块实例的名字。

可综合性问题：

分级名字和向上索引名在一般底档综合工具上是不可综合的。

提示：

- 通常，应选择对读者来说有含义的名字。相对本地名而言，这一点对于全局名显得更为重要。例如，给全局复位信号起名为 G0123，这名字不好，因没有含义，而 I 作为循环变量却是易于接受的。
- 名字中不要使用扩展字符。如网表生成或综合这一类 EDA 工具，它们具有与 Verilog 不同的命名规则，常留给这些扩展字符某些特殊含义。
- 分级名字仅用于测试模块或那些无法改用别的合适的名字的高层系统模型中。
- 避免使用向上索引名，因为它们会导致代码非常难理解，从而给调试和维护带来麻烦。

举例说明：

以下是合法名的例子：

```
A_99_Z
Reset
_54MHz_Clock$
Module       //与“module”是不一样的
\$%^&*()     //扩展标识符
```

以下是因上述原因而不合法的名字：

```
123a                 //名字不能用数字开始
$data                // 名字不能用美圆符号
module               // 名字不能用保留字
```

下面的例子说明了分级名字和向上引用名字：

```
module Separate;
    parameter P = 5;        // Separate.P
endmodule

module Top;
```

```
    reg R;            // Top.R
        Bottom    U1 ();
Endmodule

Module Bottom;
   reg R;         //Top.U1.R

        task T;      // Top.U1.T
           reg R;         //Top.U1.T.R;
            ...
        endtask

initial
      begin : InitialBlock
        reg R; // Top.U1.InitialBlock.R;
           $display(Bottom.R); // 向上索引名指向 Top.U1.R
           $display(U1.R); //  向上索引名指向 Top.U1.R
           ...
      end
endmodule       //end of Bottom module
```

Net 线路连接

Net 是结构描述中为线路连接（连线和总线）建立的模型。net的值是由net的驱动器所决定的。驱动器可以是门、UDP、实例模块或者连续赋值语句的输出。

语法：

{either}

```
NetType [ Expansion] [ Range] [ Delay] NetName,...;
trireg [ Expansion] [ Strength] [ Range] [ Delay]
NetName,...;
{Net declaration with continuous assignment}     用连续声明语句对 net 进行声明
NetType [ Expansion] [ Strength] [ Range] [ Delay]
NetAssign,...;
NetAssign = NetName = Expression
NetType = {either}
wire  tri {equivalent}
wor  trior {equivalent}
wand  triand {equivalent}
tri0
tri1
supply0
supply1
Expansion = {either}
vectored scalared
Range = [ ConstantExpression: ConstantExpression]
```

在程序中位于何处：

module-<HERE>-endmodule

规则：

- **supply0 和 supply1** 类型的 net 分别具有逻辑值0和1，并可以为它定义驱动能力(Supply strength)。
- **tri0** 和 **tri1** 类型的nets，当没有驱动时，分别具有逻辑值0和1，并可以为它定义驱动能力(Pull strength)。
- 如果net的扩展（Expansion）选项选用了关键词 **vectored**，则不允许对它进行某位和某些位的选择，也不允许对它定义强度，PLI会认为该 net 是不可扩展的；如果扩展（Expansion）选项选用了关键词**scalared**，则允许对它进行某位和某些位的选择，也允许对它定义强度，PLI将会认为该 net 是可扩展的，这些关键词是有参考价值的。
- 除了结构描述中的端口和标量连线不用声明其 net 类型外，其他类型的net变量在应用之前必须声明。

Truth Table 真值表

当Net具有两个或两个以上驱动时，同时假定其驱动器强度值均相等，这些真值表则告诉我们输出的结果。如果不相等，则驱动强度大者，驱动该 Net。

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

wand triand	0	1	X	Z
0	0	0	0	0
1	0	1	X	1
X	0	X	X	X
Z	0	1	X	Z

wor trior	0	1	X	Z
0	0	1	X	0
1	1	1	1	1
X	X	1	X	X
Z	0	1	X	Z

tri0	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	0

tri1	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	1

注意!

- 当net未被驱动时，对**tri0** 或 **tri1** 类型的net 的连续赋值不影响其值和强度，经常为强度（strength）保持为Pull，和逻辑值保持为 0（对tri0）或 1（对tri1）。
- 在IEEE 标准和已成事实的Cadence公司标准中，扩展可选项的保留字**scalared** 或 **vectored**的位置有所不同，在Cadence标准中，保留字位于范围（range）选项的跟前。

可综合性问题：

- Net类型的变量被综合成线路连接，但是某些线路连接经优化后有可能被删去。
- 综合工具只支持 Net类型中wire型的综合，其它的 Net类型均不支持。

提示:

- 在每个模块的块首明确地声明所有的 nets，即使是缺省的类型也应该明确地加以说明。通过清楚地说明设计意图，可以提高 Verilog 程序的可读性和可维护性。
- 只能用supply0 和 supply1来声明地和电源。
-

举例说明：

```
wire Clock;
wire [7:0] Address;
tri1 [31:0] Data, Bus;
trireg (large) C1, C2;
wire f = a && b,
```

```
g = a || b; // 连续赋值
```

还请参照：
连续赋值，寄存器类型说明

Number 数

整数或者实数。在Verilog中整数是通过若干位来表示的，其中某些位可以是不定值（X）或高阻态（Z）。

语法：

```
{either}
BinaryNumber                （二进制数）
OctalNumber                 （八进制数）
DecimalNumber               （十进制数）
HexNumber                   （十六进制数）
RealNumber                  （实数）
BinaryNumber = [ Size] BinaryBase BinaryDigit...
OctalNumber  = [ Size] OctalBase OctalDigit...
DecimalNumber = {either}
[ Sign] Digit... {signed number}
[ Size] DecimalBase Digit...
HexNumber = [ Size] HexBase HexDigit...
RealNumber = {either}
[ Sign] Digit... .Digit...
[ Sign] Digit...[. Digit...]e[ Sign] Digit...
[ Sign] Digit...[. Digit...]E[ Sign] Digit...
BinaryBase = {either} 'b 'B
OctalBase = {either} 'o 'O
DecimalBase = {either} 'd 'D
HexBase = {either} 'h 'H
Size = Digit...
Sign = {either} +
-Digit = {either} _ 0 1 2 3 4 5 6 7 8 9
BinaryDigit = {either} _ x X z Z ? 0 1
OctalDigit = {either} _ x X z Z ? 0 1 2 3 4 5 6 7
HexDigit = {either} _ x X z Z ? 0 1 2 3 4 5 6 7 8 9 a A
b B c C d D e E f F
UnsignedNumber = Digit...
```

在程序中位于何处：
请参阅表达式。

规则：
- 表示进制的字母、十六进制数、X和Z在数的表示中是不区分大小写的，字符Z和？在数的表示中是等价的。
- 数字中不能有空格，但是在表示进制的字母两侧可以出现空格。
- 负数表示为其二进制的补数。
- 数字的第一个字符不允许出现下划线'_'，但标识符可以。为了提高数字的可读性可用下划线把长的数字分段，在处理数字时下划线将被忽略。
- 位宽指明了数字的准确位数。
- 不指明位宽的数字，它的位宽应为32位或32位以上，取决于主机字长。
- 如果位宽大于实际的二进制位数时，高位部分补0，但除非左边最高位是X或Z，在这种情况下，则补X或Z。
- 如果位宽小于实际的二进制数位时，超过位宽的高位（左边）被舍去。

注意！
- 定义了位宽的负数被赋值到寄存器后，它将被认为是无符号的数。

```
reg [7:0] byte;
reg [3:0] nibble;
initial
begin
nibble = -1;     //例如 4'b1111
byte = nibble;  // 变为 8'b0000_1111
end
```

当寄存器类型的数或者定义了位宽的数被用在表达式中时，其值通常被当作一个无符号数。
```
integer i;
initial
i = -8'd12 / 3;   // i变成 81 (即 8'b11110100 / 3)
```

可综合性问题：
- 0和1分别被综合成接地和接电源的连线，赋值为X的则被认为是无关项。 除了使用 **casex** 语句，如用其它的条件语句，与X的比较都认为是假的。 （case 等式运算符 ＝＝＝ 和！＝＝ 一般情况下都是不可综合的）
- 除了在casex 和 casez语句中Z被认为是无关项，在其它情况下 Z 则被用来表示三态驱动器。

提示：
- 在 **case** 语句的标号中，通常用 ？要比用Z好。在程序的其它地方不要使用 ？号，否则会产生混淆。
- 用下划线来分隔较长的数字，从而提高可读性。

举例说明：

```
-253          // 有符号的十进制数
'Haf          // 未定义位宽的十六进制数
6'o67         // 位宽为六的八进制数
8'bx          // 位宽为8的二进制数，其值为不定值
4'bz1         // 位宽为四的二进制数，最低位为1其余高三位均为高阻值(4'bzzz1)。
```

下面所列的数为不合法的数并解释其原因：

```
_23           //  以_开头
8' HF F       //  包含两个非法空格
0ae           //  十进制数中出现十六进制数字
x             //  是名字，不是数字（应用1'bx）
.17           //  应该是0.17
```

还请参照：
表达式，字符串说明

Operators　运算符

在表达式中，使用运算符便可根据操作数（诸如数字、参量以及其它子表示式）计算出表达式的值。Verilog 语言中的运算符和 C 语言中的很相似。

单目运算符：

+ -	正负号
!	逻辑非
~	按位取反
& ~& | ~| ^ ~^ ^~	缩位运算符（~^ 和 ^~ 等价）

二目运算符：

+ - * /	算术运算符
%	取模运算符
> >= < <=	关系运算符
&& ||	逻辑运算符
== !=	逻辑等式运算符
=== !==	case等式运算符
& | ^ ^~ ~^	逐位运算符（^~ 和 ~^ 等价）
<< >>	移位运算符

其它运算符：

A ? B : C	条件运算符
{ A, B, C }	位拼接运算符
{ N{A} }	重复运算符

在程序中位于何处：
参阅表达式说明。

规则：

- 逻辑运算符把它的操作数当作布尔变量。例如，非零的操作数被认为是真（1'b1)；零被认为是假(1'b0)；不确定的值，例如4'bXX00，因不能判断其值为真还是假，就被认为是不确定的（1'bX)。
- 位运算符(~ & | ^ ^~ ~^)和全等运算符(= = =　　!= =)把它们操作数的逐位分别进行处理。
- 在包含 = = 或　!= 的逻辑比较式中，如果有任何一个操作数为X或Z，其结果便是不确定的(1'bX)。(请仔细看注意事项）
- 在包含(< > <= >=)的比较式中，如果操作数不确定，其结果为不定值。(1'bX)例如：

  2'b10 > 1'b0X　　//　结果为真
  2'b11 > 1'b1X　　//　结果不定(1'bX)
  （请看注意事项）
- 缩位运算符（ & ~& | ~| ^ ~^ ^~)将一个矢量缩减为一个标量。
- 位宽确定的表达式的运算采用溢出的位不计的办法，例如：4'b1111 + 4'b0001 = 4'b0000.
- 整数作除法运算时，小数部分被截掉。
- 取模运算(%)的结果是第一个操作数被第二个操作数除的余数，符号与第一个操作数一致。
- 只有某些特定的运算符允许出现在实数表达式中,例如单目运算符＋和－、算术运算符、关系运算符、逻辑运算符以及条件运算符。实数逻辑或关系运算符的结果是一个只有一位的值。

运算符的优先级：

```
+ - ! ~          单目(unary) -          最高优先级
* / %
+ -              双目(binary)
<< >>
< <= > >=
== != === !==
& ~&
^ ^~
| ~|
&&
||
?:                                      最低优先级
```

注意：

- 应用 ==、 !=、 <、　> 、 <= 和 >= 对某些位不确定的值进行比较的规则并不适用于

所有的仿真器，这点请特别注意!

- 注意单目缩位运算符与逐位逻辑运算符之间的区别，运算符本身是相同的，可根据上下文的关系来判断是哪一种，有时必须要用括号才能表达清楚。

可综合性问题：

- 逻辑运算符、逐位运算符、移位运算符是可综合的，都被综合成逻辑运算。
- 条件运算符是可综合的，被综合成多路器或带使能端的三态门。
- 运算符+、 -、 * 、< 、<= 、>、 >=、 == 和 != 都是可综合的，被分别综合成加法器、减法器、乘法器和比较器。
- 运算符 / 和 % 一般是不可综合的,只有当能用移位寄存器来表示运算时才是可综合的。而常量的 / 和 %运算是可综合，但结果只能用二进制数表示。
- 其它运算符均不能被任何工具所综合。

提示：

在写表达式的时候，运用括号要比依靠运算符的优先级要好，这样可预防错误产生，并且使那些不太了解 Verilog 语言的人更容易理解你的意思。

举例说明：

```
-16'd10              // 这是表达式，是负运算，不是有符号数!
a + b
x % y
Reset && !Enable     // 与 Reset && (!Enable)相同
a && b || c && d     // 与 (a && b) || (c && d)相同
~4'b1101             // 结果为 4'b0010
&8'hff               // 结果为 1'b1,即一位的逻辑值1
```

还请参照：

Expression 的说明

Parameter 参数

参数是为常数命名的一种手段。在 Verilog 代码模块编译时（而不是在仿真期间），可以改写参数的值。使用参数就有可能重新定义 Verilog 代码中的常数，如数组的宽度等。

语法：

```
parameter Name = ConstantExpression,
Name = ConstantExpression,
... ;
```

有些工具支持下列非标准的语法：

```
parameter [ Range] Name = ConstantExpression,
Name = ConstantExpression,
```

... ;
Range = [ConstantExpression: ConstantExpression]

在程序中位于何处：
module-<HERE>-endmodule
begin : Label-<HERE>-end
fork : Label-<HERE>-join
task-<HERE>-endtask
function-<HERE>-endfunction

规则：
- 参数是常量，在仿真期间更改参数的值是非法的。
- 在编译期间用 defparam 或者当包含参数的模块被引用时， 可以改写其参数的值。

可综合性问题：
有些综合工具能把含有参数的模块当作模板，一旦读入模板，便能够用不同的参数值多次对该模板进行综合。所有的综合工具都支持不带改动参数的模块实例的综合。

提示：
尽可能用参数给常数起一个有含义的名字。

举例说明：
下面的例子是一个 N 位宽的（可通过参数改变位宽的）移位寄存器。实例引用该参数化移位寄存器时可重新定义不同的位宽。

```
module Shifter (Clock, In, Out, Load, Data);
parameter NBits = 8;
input Clock, In, Load;
input [NBits-1:0] Data;
output Out;
always @(posedge Clock)
if (Load)
ShiftReg <= Data;
else
ShiftReg <= {ShiftReg[NBits-2:0], In}
assign Out = ShiftReg[NBits-1];
endmodule

module TestShifter;
...
defparam U2.NBits = 10;
Shifter #(16) U1 (...);      //  6位移位寄存器
Shifter U2 (...)             // 10位移位寄存器
Endmodule
```

还请参阅：
`define, Defparam, Instantiation, Specparam语句的说明。

PATHPULSE$　路径脉冲参数

- 在指定块中用指定参数 (即用specparam)对PATHPULSE$参数赋值可控制脉冲的传输。这里所谓的脉冲是指在模块输出端出现的两个跳变沿和它们之间的一段持续时间，其持续时间必须小于信号从模块的输入端直到输出端的延时。
- 如果使用缺省的PATHPULSE$参数值，仿真器将不考虑脉冲，这就是指因为路径脉冲的持续时间比模块传输延时短，故脉冲不能传过该模块，这种效应被称为“时延惯性”。用指定参数 (即用specparam)可给PATHPULSE$参数赋新的值。

语法：
```
{either}
PATHPULSE$ = ( Limit[, Limit]); {(Reject, Error)}
PATHPULSE$Input$Output = ( Limit[, Limit]);
Limit = ConstantMinTypMaxExpression
```

在程序中位于何处：
specify-<HERE>-endspecify

规则：
- 如果PATHPULSE$的第二个极限参数（即 Error）没有给定，它就应该与第一个极限参数（即 reject）相同。
- 维持时间比第一个极限参数（即 reject）短的脉冲不会输出。
- 维持时间比第一个极限参数（即 reject）长而比第二个极限参数（即 Error）短的脉冲将输出一位的不确定值（即 1'bX)。
- 维持时间比第二个极限参数长的脉冲将正常地输送出去。
- 用specparam对PATHPULSE$input$output参数重新赋值将改写常规值。
- 在同一个模块中可通过使用specparam对PATHPULSE$赋值来描述从输入到输出的延时。

可综合性问题：
综合工具不考虑延时结构，包括指定块的定义。

举例说明：
```
specify
        (clk => q) = 1.2;
        (rst => q) = 0.8;
        specparam PATHPULSE$clk$q = (0.5,1),
        PATHPULSE = (0.5);
Endspecify
```

还请参阅：
Specify, Specparam 说明

————————————————————————————————

Port 端口

模块的端口是硬件器件的引脚或接口的模型。

语法：

{definition}
{either}
PortExpression {ordered list}
.PortName([PortExpression]) {named list}
PortExpression = {either}
PortReference
{ PortReference,...}
PortReference = {either}
Name
Name[ConstantExpression]
Name[ConstantExpression: ConstantExpression]
{declaration}
{either}
input [Range] Name,...; {of port reference}
output [Range] Name,...; {of port reference}
inout [Range] Name,...; {of port reference}
Range = [ConstantExpression: ConstantExpression]
{在上述部分位选择（即 Range）选项内，冒号左侧常量表达式表示最高位（即 MSB），冒号右侧常量表达式表示最底位（即LSB）}

在程序中位于何处：

module (<HERE>); {definition}
<HERE> {declaration}
...
endmodule

规则：

- 在端口列表中列出的所有端口必须按次序排列或按端口名称排列，这两种排列方式是不同的，不能混合使用。
- 有端口的名称但没有端口表达式，如 .A()，则表示在本模块中定义了不与任何东西相连的端口。
- 每个端口除了必须在端口列表中列出外， 还必须声明该端口是输出(output)、输入

(input)、还是双向端口(inout)。

- 每个端口不但要声明是输出、输入、还是双向端口，而且还要声明是连线（wire）还是寄存器(reg)类型，如果没声明，则会隐含地认为该端口是连线（wire）类型，且其位宽与相应的端口一致。如果某端口已被声明为一矢量，则其端口的方向和类型两个声明中的位宽必须一致。
- 输入和双向端口不能声明为寄存器类型。
- 输出端口的类型不能声明为实型（Real）或实时型（realtime）。

提示：

- 在测试模块中不要定义端口。
- 在模块定义时不建议使用命名的端口的列表，因为很少有人这样来定义模块端口，大家都不了解这种端口的定义形式。

举例说明：

```
module (A, B[1], C[1:2]);
input A;
input [1:1] B;
output [1:2] C;

module (.A(X), .B(Y[1]), .C(Z[1:2]);
input X;
input [1:1] Y;
output [1:2] Z;
```

还请参照：

Module, User Defined Primitive, Instantiation 的说明。

Procedural Assignment　过程赋值语句

改变寄存器的值，或者安排以后的变化。

语法：

```
{Blocking assignment}                              阻塞赋值
RegisterLValue = [ TimingControl] Expression ;
{Non-blocking assignment}                          非阻塞赋值
RegisterLValue <= [ TimingControl] Expression ;
RegisterLValue = {either}
RegisterName
RegisterName[ Expression]
RegisterName[ ConstantExpression: ConstantExpression]
Memory[ Expression]
```

{ RegisterLValue,...}

在程序中位于何处：

请参阅statement语句的说明。

规则：

- 对寄存器的赋值（不包括正负号）
- 对于实型和实时数据类型的寄存器不允许选择某位和某几位的。
- 当赋值语句执行时，右侧的表达式被计算出值，但是直到定时控制事件或延时（也被称为‘内部指定的延时’）发生后，左侧的表达式才更新。
- 直到左侧的表达式更新后（例如内部定义的延时过后）阻塞赋值语句才算完成。在begin-end模块中，只有当前一条语句执行完后，才能执行其后面的一条语句。在fork-join模块中，只有当块中所有的阻塞赋值语句结束后，整个块才算结束。
- 如果仿真时刻相同，要待所有的阻塞赋值语句执行后，非阻塞赋值语句才执行。

```
A <= #5 0;
A = #5 1;      //5个时间单位后，A将变为0，而不是变为1
```

注意！

寄存器变量可以在一个或几个**initial** 或 **always** 语句中赋值。无论何时，寄存器变量的值都是由最近的赋值所决定，与事件的来源无关。这一点与 net 类型的变量不同。net可以由两个或更多的源驱动，其结果值则取决于net变量的类型（wire型,wand型等）。

可综合性问题：

- 综合工具不考虑延时。
- 定时控制或延时是不可综合的。
- 同一个寄存器类型变量虽然可以在几个**always**语句中赋值，但只有在一个 **always**语句中赋值的才有可能被综合。
- 同一个寄存器类型变量不能既用阻塞赋值和非阻塞赋值。
- 在描述组合逻辑的**always**块中，右侧表达式被综合成组合逻辑，左侧的表达式被综合成连线，如有不完整的赋值则综合成锁存器。在描述时序逻辑的用时钟沿触发的**always**块中，非阻塞赋值符的左侧被综合成触发器，阻塞赋值符的左侧则被综合成一个连接，除非它被用在该**always**块之外，或者在赋值之前它的值已被读取。

提示：

- 通常采用非阻塞赋值语句来生成触发器组成的时序逻辑，而阻塞赋值常用于其它方面，这样做可以防止时钟沿触发的 **always** 块中发生竞争冒险。这样做也可使设计意图更加清晰，又能避免生成不需要的触发器。
- 在时钟树的模型已确定的情况下，可用一个简单的内部指定的延时来避免RTL时钟沿对不齐的问题。

举例说明：

```
always @(Inputs)
    begin : CountOnes
        integer I;
```

```
        f = 0;
        for (I=0; I<8; I=I+1)
            if (Inputs[I])
                    f = f + 1;
    end
always @Swap
    fork // 交换a和b的值
        a = #5 b;
        b = #5 a;
    join // 延时5秒后完成

always @(posedge Clock)
    begin
        c <= b;  // 用旧的b值
        b <= a;  // b被a值替换
    end
```

用非阻塞赋值语句时加一个延时来做输出与时钟沿有些偏移的仿真：

```
  always @(posedge Clock)
        Count <= #1 Count + 1;
```

在时钟周期的第五个下降沿插入复位信号：

```
  initial
    begin
       Reset = repeat(5) @(negedge Clock) 1;
       Reset = @(negedge Clock) 0;
    End
```

还请参照：

Timing Control, Continuous Assignment 的说明。

Procedural Continuous Assignment 过程连续赋值语句

启动过程连续赋值语句，将给一个或多个寄存器赋值，并同时防止一般的过程赋值语句影响已赋值的寄存器。

语法：

```
assign RegisterLValue = Expression ;
deassign RegisterLValue ;
RegisterLValue = {either}
RegisterName
```

RegisterName[Expression]
RegisterName[ConstantExpression: ConstantExpression]
MemoryName[Expression]
{ RegisterLValue,...}

在程序中位于何处：
请参阅 Statement的说明。
规则：

- 过程连续赋值语句执行后，它会对指定的寄存器（组）强制地维持过程连续赋值直到解除赋值（deassign）语句的执行，或直到另一个过程连续赋值语句又对该寄存器（组）赋值。
- 用force（强制）语句可以改写已由过程连续赋值语句赋值的寄存器类型变量，直到 release 语句的执行，此时强制赋值被解除而原过程连续赋值对该寄存器类型变量的作用又重新恢复。

注意！
连续赋值语句与过程连续赋值语句尽管很相似，但并不是完全一致。在编写程序时，确认将 assign 写在正确的位置。 过程连续赋值语句可以写在声明语句允许出现的位置（在 initial, always, task, function 等内部），而连续赋值语句则必须写在任何 initial 或 always 块之外。

可综合性问题：
无论用什么综合工具，过程连续赋值语句是都不能综合的。

提示：
过程连续赋值语句可以用来为异步复位和中断建立仿真模型。

举例说明：

```
always @(posedge Clock)
    Count = Count + 1;        //受下面always块控制的计时钟个数的计数器

always @(Reset) // 异步复位
    if (Reset)
            assign Count = 0; // 当 Reset为高时，使Count为0，不计数
        else
            deassign Count; // 当 Reset为低时，解除Count为0，
                            //于是下一个时钟的上升沿又重新开始计数。
```

还请参阅：
Continuous Assignment, Force 的说明

Programming Language Interface 编程语言接口

Verilog编程语言接口 (PLI)为用户提供了在Verilog模块中调用用C语言编写的函数的手段。这些函数可以动态地访问和修改被引用的Verilog数据结构中的数据，用PLI编写的系统任务使上述功能变得容易使用。通过调用用户定义的系统任务和函数的可以来启动PLI，用户编写自己的PLI 模块的目的是扩大系统任务和函数的内容。用户自定义的系统任务和函数在调用时都用以$字符开头的任务和函数名。这与Verilog语言提供的系统任务和函数库名一致。如用户自定义的系统任务和函数名与原系统任务或函数名相同时，则执行用户自定义的系统任务和函数。

下面列举的是PLI在某些方面的应用：

- 延迟计数
- 测试矢量读入
- 波形演示
- 源代码调试

接口模型可用C语言或其他语言（例如VHDL或硬件建模工具）编写或生成。
对于PLI的全面讨论超出了本参考指南的范围。

Register　寄存器

寄存器可存储在**initial**、 **always**、**task** 和 **function** 块中所赋的值，广泛地应用在行为建模中。

语法：

```
{either}
reg [ Range] RegisterOrMemory,...;
integer RegisterOrMemory,...;
time RegisterOrMemory,...;
real RegisterName,...;
realtime RegisterName,...;
RegisterOrMemory = {either}
RegisterName
MemoryName Range
Range = [ ConstantExpression: ConstantExpression]
```

在程序中位于何处：

```
module-<HERE>-endmodule
begin : Label-<HERE>-end
fork : Label-<HERE>-join
task-<HERE>-endtask
function-<HERE>-endfunction
```

规则：

- 寄存器类型变量只能用过程赋值语句赋值。
- 在具体实现时，整数（integer）类型的变量至小用32位，时间（time）类型的变量至小用64位寄存器。
- integer 或 time类型的寄存器变量与位数相同的reg类型的寄存器变量行为是相同的。Integer 和 time 型的寄存器变量也可像reg类型的寄存器变量一样对某位或某些位操作。而在表达式中，整数类型的值被当作有符号值，而reg、time类型的值被当作无符号值。
- 存储器类型数组中的每个元素作为整体可以进行读或写操作，如果要单独访问数组中某个元素的个别位，则必须先把这个元素的内容复制到某个位数相同的寄存器变量中才能进行。

注意！

- 虽然 register 这个词指的是硬件寄存器（例如触发器），而寄存器（register）这个名字，在这里是指软件寄存器（即变量）。Verilog寄存器常用于组合逻辑电路、锁存器、触发器和接口电路的描述和综合。
- realtime 类型寄存器变量是Verilog语言新增加的变量类型，目前还没有任何工具支持这种类型的变量。
- 有符号和无符号值的概念，不同版本的Verilog和用不同厂家的仿真器时，并不是完全一致的。因此，当使用位宽大于32位的有符号数或矢量时要特别注意。

可综合性问题： .

- Real, time 和 realtime类型的寄存器变量是不可综合的。
- 在描述组合逻辑的always块中，寄存器被综合成 wire型；如果存在不完整赋值的情况，则被综合成锁存器。在描述时序逻辑的always块中，寄存器根据块内语句的内容被综合成连线（wire）或者触发器。
- 运用目前的综合工具，整数被综合成32位，其值用二进制数表示，负数则用其二进制补码表示。
- 根据所用语句，存储器数组会被综合成触发器或连线，而不会被综合成RAM 或 ROM的器件。

提示：

运用reg类型变量来描述寄存器逻辑，integer类型变量用于循环变量和计数，real类型变量用于系统模块，time 和 realtime类型变量用于测试模块中记录仿真时刻。

举例说明：

```
reg a, b, c;
reg [7:0] mem[1:1024], byte; // byte 不是数组只是一个8位的reg类型矢量
integer i, j, k;
time now;
real r;
realtime t;
```

下面的部分显示了reg 类型 和integer 类型变量的一般用法

```
integer i;
reg [15:0] V;
reg Parity;
always @(V)
    for ( i = 0; i <= 15; i = i + 1 )
        Parity = Parity ^ V[i];
```

还请参阅：
Net的说明。

Repeat　　重复执行语句

把一个或多个声明语句重复地执行指定的次数。

语法 ：

```
repeat ( Expression)
    Statement
```

在程序中位于何处：
参见 Statement 的说明

规则：
重复执行的次数是由表达式的数值所决定的，如果该值为0, X 或 Z，则不会有重复。

可综合性问题：
只有部分综合工具可以综合repeat语句,而且只有当该循环中的每个循环的分支都被时钟事件，如被 @(posedge Clock)，所中断时才有可能被综合成电路。

举例说明：

```
initial
    begin
        Clock = 0;
            repeat (MaxClockCycles)
                begin
                    #10 Clock = 1;
                    #10 Clock = 0;
                end
    end
```

还请参阅：
For, Forever, While, Timing Control的说明。

Reserved Words　　关键词

下列词汇是Verilog语言规定的所有的关键词，请注意，千万不要把这些标识符用作自定义的标识符，除非把他们改写为大写的字符或扩展字符。

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

Specify　　指定的块延时

Specify块（指定延时块）用于描述从模块的输入到输出的路径延时以及定时约束，例如信号的建立和保持时间。用指定延时块可以在设计时把模块的信号传输延时与行为或结构分开来进行描述。

语法：

```
specify
SpecifyItems...
endspecify
SpecifyItem = {either}
Specparam
PathDeclaration
TaskEnable {Timing checks only}
PathDeclaration = {either}
SimplePath = PathDelay;
EdgeSensitivePath = PathDelay;
StateDependentPath = PathDelay;
SimplePath = {either}
( Input,... [ Polarity] *> Output,...)  {full}
( Input [ Polarity] => Output)          {parallel}

EdgeSensitivePath = {either}
([ Edge] Input,... *> Output,... [ Polarity]: Expression)
([ Edge] Input => Output [ Polarity]: Expression)
StateDependentPath = {either}
if ( Expression) SimplePath = PathDelay;
if ( Expression) EdgeSensitivePath = PathDelay;
ifnone SimplePath = PathDelay;

Input = {either}
InputName
InputName[ ConstantExpression]
InputName[ ConstantExpression: ConstantExpression]
Output = {either}
OutputName
OutputName[ ConstantExpression]
OutputName[ ConstantExpression: ConstantExpression]
Edge = {either} posedge negedge
Polarity = {either}
 +
 -
PathDelay = {either}
ListOfPathDelays
( ListOfPathDelays)
ListOfPathDelays = {either}
t
t,t                          {Rise,Fall}
t,t,t                        {Rise,Fall,Turn-Off}
```

t,t,t,t,t,t　　{01,10,0Z,Z1,1Z,Z0}
t,t,t,t,t,t,t,t,t,t,t,t　　{01,10,0Z,Z1,1Z,Z0,0X,X1,1X,X0,XZ,ZX}
t = MinTypMaxExpression

在程序中位于何处：
module-<HERE>-endmodule

规则：
- 路径必须从模块的输入端开始，在该模块的输出端结束，而且在模块内部只能有一个驱动器。
- 在路径声明中可使用全连接符(*>)或者并行连接符 (=>)来描述。全连接符指所有从输入端到输出端可能的路径，并行连接符指命名的输入端的某些位到命名的输出端的某些位的路径。
- 模块路径的极性可选是指着路径可以选择正极性或者负极性，分别指路径是同相的或是反相的（即路径的输入端若是正跳沿，输出端也是正跳沿，则路径是同相的。反之是反相的）。无论选哪一个都不影响仿真，但路径的相位可改变的选项便于时序分析等工具使用。
- 跳变沿敏感的路径数据表达式同样也不影响仿真。
- 与状态有关的路径延时(SDPD)表达式只跟端口、常量、局部定义的寄存器或者net类型变量有关。只有部分运算符在SDPD表达式中是有效的：如逐位计算符(~ & | ^ ^~ ~^)、逻辑运算和逻辑等式运算符(== != && || !)、 缩位运算符(& | ^ ~& ~| ^~ ~^)、位拼接运算符、重复拼接运算符和条件运算符({} {{}} ?:)。如果条件表达式的值为真（在SDPD 表达式中 1，X，Z 均被认为是真），路径延时只影响路径。
- 如果没有一个if 条件为真，则用Ifnone来定义缺省的SDPD。如果同一条路径既定义为ifnone 与状态有关的路径延时（SDPD）又定义为一般简单的路径延时，则是非法的。
- 无条件的路径优先于SDPD路径。
- 对于同一条路径，跳变沿敏感的SDPD路径声明必须是唯一的，必须用不同的电平或不同的沿（或者两者）。在每条语句中，必须以同样的方式（整个端口、某一位、某些位）来引用输出端的信号。
- 如果模块的延时既包括指定的块延时（specify delays）又包括分布的延时（即由门、UDP、Net引起的延时），应选用最长的延时作为每条路径的延时。

可综合性问题：
综合工具不能综合指定延时块。指定延时块只用于模块的时延仿真建模。

注意！
- 目前还没有一个仿真工具支持上面语法中列出的那种可用12种不同跳变参数表示某条路径延时的方法。
- 有关路径目的地的规则比当前许多仿真工具所能支持的灵活。

提示：
- 运用指定延时块来描述库中的单元（cell）延时。请注意建立库模型时延时是怎样计算的。以PLI（编程语言接口）为基础的延时计算，需要依据并访问设计中所有单元的指定延时块中的信息。

- 可运用指定延时块来描述“黑匣子”元件的定时特性， 但这时还需要借助于支持指定延时块特性的时序验证工具或综合工具。

举例说明：

```
module M (F, G, Q, Qb, W, A, B, D, V, Clk, Rst, X, Z);
    input A, B, D, Clk, Rst, X;
    input [7:0] V;
    output F, G, Q, Qb, Z;
    output [7:0] W;
    reg C;
        // Functional Description ...功能描述
        specify
            specparam TLH$Clk$Q = 3,
            THL$Clk$Q = 4,
            TLH$Clk$Qb = 4,
            THL$Clk$Qb = 5,
            Tsetup$Clk$D = 2.0,
            Thold$Clk$D = 1.0;
    // 单一路径，全连接
            (A, B *> F) = (1.2:2.3:3.1, 1.4:2.0:3.2);
    // 单一路径，并行连接，正极性
            (V + => W) = 3,4,5;
    // 沿敏感路径，带极性
            (posedge Clk *> Q +: D)  = (TLH$Clk$Q,THL$Clk$Q);
            (posedge Clk *> Qb -: D) = (TLH$Clk$Qb,THL$Clk$Qb);
    // 电平敏感路径
            if (C) (X *> Z) = 5;
            if (!C && V == 8'hff) (X *> Z) = 4;
            ifnone (X *> Z) = 6; // 缺省为SDPD,从 X（不定值）到 Z（高阻值）
    // 时序检测
        $setuphold(posedge Clk, D,Tsetup$Clk$D, Thold$Clk$D, Err);
    endspecify
endmodule
```

还请参阅：

Specparam, PATHPULSE$, $setup 的说明。

Specparam　　延时参数

类似于parameter(参数)，但只能用在指定延时块中。

语法：

```
specparam Name = ConstantExpression,
Name = ConstantExpression,
... ;
```

在程序中位于何处：

```
specify -<HERE>- endspecify
```

规则:

- Specify块中的常量表达式可以用数字和 specparam 来定义，但不能用参数(parameter)来定义，specparam 不能用在 Specify块（即指定延时模块）外。
- 利用 **defparam** 或在模块的实例引用时使用＃，可以改写用 specparam 定义的延时参数值，用编程语言接口（PLI）也可以修改其值。

提示

- 在Specify块中，用specparam来定义命名的延时参数比直接用数字要好。
- 这些延时参数应有一个命名的规则，这样便于对它们进行修改，如果有必要的话，也可以采用PLI的延时计数来进行修改。

举例说明：

```
specify
    specparam   tRise$a$f = 1.0,
                tFall$a$f = 1.0,
                tRise$b$f = 1.0,
                tFall$b$f = 1.0;
                (a *> f) = (tRise$a$f, tFall$a$f);
                (b *> f) = (tRise$b$f, tFall$b$f);
endspecify
```

还请参阅：

PATHPULSE$, Specify的说明。

Statement　　声明语句

运用声明语句可以描述硬件模块的行为。声明语句在定时控制的（延时、控制程序、等待）时刻执行。若两个或两个以上的语句是一起的，必须把它们写在 begin-end 或 fork-join 块中。在 begin-end 块中每条语句是顺序执行的，在fork-join块中，它们是并行执行的。initial 或 always块中的语句是同其它initial 或 always块中的语句是同时执行的。

语法：

```
{either}
```

; {Null statement}
TimingControl Statement {Statement may be Null}
Begin
Fork
ProceduralAssignment
ProceduralContinuousAssignment
Force
If
Case
For
Forever
Repeat
While
Disable
-> EventName; {Event trigger}
TaskEnable

在程序中位于何处：
initial-<HERE>
always-<HERE>
begin-<HERE>-end
fork-<HERE>-join
task-<HERE>-endtask {Null allowed}
function-<HERE>-endfunction
if()-<HERE>-else-<HERE> {Null allowed}
case- label:-<HERE>-endcase {Null allowed}
for(<HERE>)-<HERE>
forever-<HERE>
repeat()-<HERE>
while()-<HERE>

还请参阅：
Timing Control的说明。

Strength　　强度

除了逻辑值外，Net类型的变量还可以定义强度，因而可以更精确地建模。Net的强度来自于动态 Net 驱动器的强度。在开关级仿真时，当 Net由多个驱动器驱动且其值互相矛盾时，常用强度（Strength）的概念来描述这种逻辑行为。

语法：

```
{either}
( Strength0, Strength1)
( Strength1, Strength0)
( Strength0)              {pulldown primitives only}
( Strength1)              {pullup primitives only}
( ChargeStrength)         {trireg nets only}
Strength0 = {either}
supply0
strong0
pull0
weak0
highz0
Strength1 = {either}
supply1
strong1
pull1
weak1
highz1
ChargeStrength = {either}
large
medium
small
```

在程序中位于何处：

请参照： Net、Instantiation、Continuous Assignment 的说明。

规则：

- 关键词Strength0 和 Strength1用于定义Net的驱动器强度。其中Strength表示强度，与紧跟着的0和1连起来分别表示输出逻辑值为0和1时的强度。
- 在强度声明中可选择不同的强度关键字来代替strength，但(highz0,highz1) 和(highz1,highz0)这两种强度定义是不允许的，在pullup （上拉）和 pulldown（下拉）门的强度声明中 highz0 和 highz1是不允许的。
- 默认的强度定义为 (strong0,strong1)，但下述情况除外：

1) 对于pullup and pulldown门，默认强度分别为(pull1) 和 (pull0)。
2) 对于trireg 的 Net，默认强度为 medium

3) 强度定义为supply0 和 supply1的Net，总是能提供强度。

- 在仿真期间，Net 的强度来自于 Net 上的主驱动强度（即具有最大强度值的实例或连续赋值语句）。如果 Net 未被驱动，它会呈现高阻值，但以下情况除外：

1) tri0 和 tri1 类型的 net 分别具有逻辑值0和1，并为pull强度。
2) trireg 类型的 net 保持它们最后的驱动值。
3) 强度为supply0 和 supply1 的 nets分别具有逻辑值0和1，并能提供驱动能力。

- 强度值有强弱顺序，可从 supply（最强的）依次减弱排列到 highz（最弱的），当需要确定Net的确实逻辑值和强度时，或者当 Net由多个驱动器驱动而且驱动相互间出现冲突时，出现冲突的两个强度值在强弱顺序表中的相对位置就会对该Net的真实逻辑值起作用。

Supply
Strong
Pull
Large
Weak
Medium
Small
Highz

可综合性问题：

不可综合

提示：

可以在$display和$monitor等中用特定的格式控制符 %V 显示其强度值。

举例说明：

```
assign (weak1,weak0) f= a + b;
trireg  (large ) c1,c2;
and (strong1,weak0)  u1(x,y,z);
```

请参阅：

Continous Assignment、Instantiation、Net、$display 的说明

String 字符串

字符串能够用在系统任务（诸如$display和$monitor等）中作为变量，字符串的值可以像数字一样储存在寄存器中，也可以像对数字一样对字符串进行赋值、比较和拼接。

语法

见“string”说明。

在程序中位于何处：

请参见 Expression 说明。

规则

- 一条字符串不能占原代码的多行。
- 字符串可以包含下列扩展字符。

• \n	• 换行
• \t	• Tab符
• \\	• 反斜杠字符\
• \”	• 双引号字符”
• \nnn	• 八进制的ASCII字符
• %%	• 百分号%

- 诸如$display和$monitor等的系统任务中的打印字符串可以包含特殊的格式控制符(如%b)(参见$display的说明)。
- 当字符串存储于寄存器中，每个字符要占8位，字符以ASCII代码形式存储。VerilogHDL语言的字符串的定义和 C 语言的不一样。在C 语言中需要用，而在VerilogHDL语言中不需要用ASCII代码的 0字符来表示字符串的结束。

注意！

在表达式中使用字符串时，请注意填加物。对字符串的处理跟对数字的处理方式一样，当字符所占的位数少于寄存器的数目时，则在字符串的左边的寄存器中填加0。

举例说明：

```
reg [23:0] MonthName[1:12];
initial
begin
MonthName[1] = "Jan";
MonthName[2] = "Feb";
MonthName[3] = "Mar";
MonthName[4] = "Apr";
MonthName[5] = "May";
MonthName[6] = "Jun";
MonthName[7] = "Jul";
MonthName[8] = "Aug";
MonthName[9] = "Sep";
MonthName[10] = "Oct";
MonthName[11] = "Nov";
MonthName[12] = "Dec";
end
```

请参阅：

NUMBER, $display 的说明。

Task　　任务

任务常用于把模块代码分割成由若干声明语句构成的较大的块，便于模块代码的理解和维护，也可以从模块代码的不同位置执行这样一个常见的顺序声明语句块。

语法：

```
task TaskName;
[ Declarations...]
Statement
endtask
Declaration = {either}
input [ Range] Name,...;
output [ Range] Name,...;
inout [ Range] Name,...;
Register
Parameter
Event
Range = [ ConstantExpression: ConstantExpression]
```

规则：

- 若用于任务中的命名变量或参数没有在任务块中声明，则指的是在模块中声明的命名变量或参数。
- 任务中的 input、output 和 inout 的个数不受限制（也可以为零个）。
- 任务中的变量（包括输入和双向端口（inout））可以声明为寄存器型。如果没有明确地声明，则默认为寄存器型，且其位宽与相应的变量匹配。
- 当启动任务时，相应于任务的输入和双向端口（inout）的变量表达式的值被存入相应的变量寄存器中。当任务结束时，输入和双向端口（inout）的变量寄存器中的值又被代入启动任务的语句中相应的表达式。

注意！

- 和模块的端口定义不一样，任务的变量不能在任务名后的括号中定义。
- 任务中若包括一句以上的语句，必须要用begin –end 或fork-join 将其包含成块。
- 任务的输入、双向端口（inout）、输出和局部寄存器的值都是静态储存的，也就是说即使多次启动任务，也只有一份这些寄存器的拷贝。若第一次启动的任务还未完成，便第二次启动该任务，其输入、双向端口（inout)、输出和局部寄存器的值便会被覆盖。
- 当被启动的任务运行结束时，输出和双向端口（inout)的值被代入任务中相应的寄存器表达式。如果任务中的输出和双向端口（inout)在赋值后有时间的控制，则相应的寄存器只能在定时控制延迟后才被更新。
- 同样，对输出和双向端口（inout)寄存器变量的非阻塞赋值语句也不会起作用，因为当任务返回时，赋值语句可能还未生效。

可综合性问题：

包含定时控制语句的任务是不可综合的。启动的任务往往被综合成组合逻辑。

提示：

- 复杂 RTL 模块通常需要用多个 always 块来构造。建议最好不要采用一个always块运行多个任务的方案。
- 在测试块中可用任务来产生重复的激励序列。例如，对存储器的数据读写（见例）序列。
- 某任务如果被多个模块引用，可以把它定义为一个独立的模块（只包括该任务），并可用层次命名来引用它。

举例说明：

这个例子表示一个简单的可以综合的RTL任务

```
task Counter;
    inout [3:0] Count;
    input Reset;
        if (Reset)      // 同步复位
            Count = 0;  // 对 RTL 必须用非阻塞方式赋值
        else
            Count = Count + 1;
Endtask
```

下面这个例子说明如何在测试模块中运用任务。

```
 module TestRAM;

   parameter AddrWidth = 5;
   parameter DataWidth = 8;
   parameter MaxAddr = 1 << AddrBits;

   reg [DataWidth-1:0] Addr;
   reg [AddrWidth-1:0] Data;
   wire [DataWidth-1:0] DataBus = Data;
   reg Ce, Read, Write;

   Ram32x8 Uut (.Ce(Ce), .Rd(Read), .Wr(Write),
                .Data(DataBus), .Addr(Addr));
   initial
   begin : stimulus
      integer NErrors;
      integer i;
 // 错误开始记数
      NErrors = 0;
 // 为每个地址写上地址值
```

```
     for ( i=0; i<=MaxAddr; i=i+1 )
        WriteRam(i, i);
// 读且比较
     for ( i=0; i<=MaxAddr; i=i+1 )
     begin
        ReadRam(i, Data);
        if ( Data !== i )
            RamError(i,i,Data);
     end
//小结错误个数
      $display(Completed with %0d errors, NErrors);
end

task WriteRam;
   input [AddrWidth-1:0] Address;
   input [DataWidth-1:0] RamData;
begin
   Ce = 0;
   Addr = Address;
   Data = RamData;
   #10 Write = 1;
   #10 Write = 0;
   Ce = 1;
end
endtask

task ReadRam;
   input [AddrWidth-1:0] Address;
   output [DataWidth-1:0] RamData;
begin
   Ce = 0;
   Addr = Address;
   Data = RamData;
   Read = 1;
   #10 RamData = DataBus;
   Read = 0;
   Ce = 1;
end
endtask

task RamError;
   input [AddrWidth-1:0] Address;
   input [DataWidth-1:0] Expected;
   input [DataWidth-1:0] Actual;
```

```
        if ( Expected !== Actual )
        begin
           $display("Error reading address %h", Address);
           $display(" Actual %b, Expected %b", Actual,
                         Expected);
           NErrors = NErrors + 1;
        end
        endtask

endmodule
```

请参阅：

Task Enable，Function 的说明。

Task Enable　任务的启动

在模块代码中只需用任务名便可启动任务。当任务启动时，输入值通过任务的端口变量（输入和 inout变量）传递到任务中。当任务结束时，返回值通过任务的端口寄存器变量（输出和 inout变量）传出。

语法

```
TaskName[( Expression,...)];
```

规则

- 任务可以从 **initial** 或 **always** 块或其它任务中启动。任务可以多次调用。但任务不能被函数调用。
- 调用任务的语句中，端口表达式的顺序和任务端口变量声明的顺序必须一致。端口的个数必须与任务声明的端口变量的个数一致。
- 若任务的端口变量是输入时，则对应的端口变量可以是任何一种表达式；若端口变量为输出和 **inout** 时，对应的端口变量必须位于进程赋值语句的左边而且必须是有效的。
- 当任务启动时，输入和 **inout** 表达式复制到相应的变量寄存器中。当任务结束时，输出和 inout 寄存器的值会复制到启动任务相应的端口寄存器中。
- 可以在任务内部或任务外部把任务禁止（**disable**）。

注意！

任务中变量寄存器默认为静态的，所以当一个任务正在执行时又启动该任务时，输入和 **inout** 寄存器的值会被覆盖。

可综合性问题：

若任务不包含定时控制，是有可能被综合的。调用的任务往往被综合成组合逻辑。

举例说明：

```
    task Counter;
        inout [3:0] Count;
        input Reset;
        ...
    endtask

 always @(posedge Clock)
      Counter(Count, Reset);
```

请参阅：

Disable、Task、Function Call 的说明。

Timing control　　定时控制

用于延迟语句的执行或按排语句的执行顺序。定时控制可以放在语句的前面，或者在程序的进程赋值语句表达式中的赋值操作符（即 = 或 <= ）之间。前一种延迟语句的执行，后一种延迟声明的语句生效。

语法：

```
    {Timing controls before statements}
     {either}
     DelayControl
     EventControl
     WaitControl
    {Intra-assignment Timing controls}
    {either}
     DelayControl
     EventControl
     repeat ( Expression) EventControl
     DelayControl = {either}
     # UnsignedNumber
     # ParameterName
     # ConstantMinTypMaxExpression
     # ( MinTypMaxExpression)
     EventControl = {either}
     @Name {of Register, Net or Event}
     @( EventExpression)
     EventExpression = {either}
     Expression
     Name {of Register, Net or Event}
     posedge Expression {01, 0X, 0Z, X1 or Z1}
```

```
negedge Expression {10, 1X, 1Z, Z0 or X0}
EventExpression or EventExpression
WaitControl = wait ( Expression)
```

在程序中位于何处：

请参阅：statement、procedural assignment (for intra-assignment timing control) 的说明。

规则：

- 在某声明语句前面插入的事件或延迟控制使原本立刻要执行的该条语句延迟执行。
- 当执行到wait时，如果其表达式为假（0或X），wait 控制只延迟 wait 语句后的下一条语句；当表达式为真（非0）时，下一条语句才执行。当执行到 wait 时，如果表达式为真，下一句不延迟马上执行。
- 执行进程赋值语句时，要检查赋值语句右边的表达式，如果没有内部赋值延迟，若用的是阻塞赋值，则左边的寄存器类型变量立即更新，若用的是非阻塞赋值，则在下一个仿真周期更新。如果有内部赋值延迟，左边的寄存器类型变量只有在发生内部赋值延迟后才更新。
- 内部赋值延迟必须是常数的，但语句前的延迟可以是常数或变量（即Net或reg 型变量）。
- or 列表中的任何一个信号（事件）变化（发生）时，即触发事件控制。
- 对于 posedge（上升沿）和 negedge（下降沿）事件触发控制，只测试表达式的最低位。要不然的话，表达式的任何变化都会触发事件。

注意！

对于阻塞赋值语句而言，指定内部赋值延迟为零（# 0）与不指定是不一样的，也与没有赋值延迟的非阻塞赋值语句不同。 对于阻塞赋值语句而言，指定#0意味着该语句在所有待定事件完成以后，而在非阻塞赋值完成以前进行。（不指定内部赋值延迟和指定内部赋值延迟为零（#0）的赋值语句是一样的。）

可综合性问题：

- 综合时延迟被忽略。
- 综合工具不支持 wait语句和内部赋值延迟以及repeat(重复)语句。
- 事件控制用于控制always块的执行，从而能确定综合出的逻辑是组合的还是时序的。一般情况下，always后紧跟着的就是事件控制，这有时也称为敏感列表。

提示：

在用RTL（寄存器传输级HDL语言）描述电路时，可用内部赋值延迟可来描述在给表示触发器的寄存器变量赋值时的时钟偏移现象。

举例说明：

```
#10
#(Period/2)
#(1.2:3.5:7.1)
@Trigger
@(a or b or c)
```

```
@(posedge clock or negedge reset)
wait (!Reset)
```

非阻塞赋值时使用延迟来克服时钟的偏移：

```
always @(posedge Clock)
        Count <= #1 Count + 1;
```

在周期时钟的第五个下降沿复位：

```
initial
    begin
        Reset = repeat(5)  @(negedge Clock)  1;
        Reset = @(negedge clock) 0;
End
```

请参阅：

Procedural Assignment、Always、Repeat语句的说明

User Defined Primitive　　用户自定义原语

用户自定义原语（UDPS）可以为小型元件建立模型，这也是模块的另一种表示方法。可以用引用由门构建的实例同样的方式来实例引用用户自定义原语（UDPS）。

语法：

```
primitive UDPName (OutputName, InputName,...);
   UDPPortDeclarations ...
   UDPBody
endprimitive
UDPPortDeclaration = {either}
  output OutputName;
  input InputName,...;
  reg OutputName; {Sequential UDP}
UDPBody = {either} CombinationalBody SequentialBody
CombinationalBody =
  table
    CombinationalEntry...
  endtable
SequentialBody =
  [initial OutputName = InitialValue;]
  table
     SequentialEntry...
  endtable
InitialValue =
```

```
  {either} 0 1 1'b0 1'b1 1'bx {not case sensitive}
CombinationalEntry = LevelInputList : OutputSymbol ;
SequentialEntry =
  SequentialInputList : CurrentOutput : NextOutput;
SequentialInputList = {either}
   LevelInputList
   EdgeInputList
LevelInputList = LevelSymbol...
EdgeInputList =
   [ LevelSymbol...] EdgeIndicator [ LevelSymbol...]
CurrentOutput = LevelSymbol
NextOutput = {either} OutputSymbol
EdgeIndicator = {either}
    ( LevelSymbol LevelSymbol)
    EdgeSymbol
OutputSymbol = {either} 0 1 x {not case sensitive}
LevelSymbol = {either} 0 1 x ? b {not case sensitive}
EdgeSymbol = {either} r f p n * {not case sensitive}
```

规则：

- UDP只允许有一个输出端，至少允许有一个输入端。具体实施时，对输入端的个数是有限制的，但必须至少允许10个输入端口。
- 如果某UDP的输出端定义为reg 型（寄存器类型）变量，则该UDP是时序逻辑的UDP,否则为组合逻辑的UDP。
- 如果已对时序逻辑的UDP的输出进行了初始化，则只有待到在仿真开始时，初始值才开始从引用的原语实例的输出传出。
- 描述时序逻辑的UDPS可以是电平敏感的或边沿敏感的。若在真值表中有边沿敏感的指示（至少一个），则该描述时序逻辑的UDP为边沿敏感的。
- UDP的行为在表中定义。表的行定义为不同输入条件下的输出。对于描述组合逻辑的UDP，每一行定义为一个或多个输入的组合逻辑的输出。对于描述时序逻辑的UDP，每一行都要考虑reg类型变量的当前输出值。一行最多只能有一个边沿变化入口。行定义了在指定的边沿发生变化时，由输入值和当前输出值所产生的输出值。
- UDP表中所用的特殊的电平和边沿符号含义如下：

?	0 1 或 x

b or B	0 或 1
-	输出不变
(vw)	由 v 变为 w
r or R	(01)
f of F	(10)
p or P	(01)(0x)或(x1)
n or N	(10)(1x)或(x0)
*	(??)

- 若组合逻辑的输入值和触发边沿没有明确指定将会导致输出的不确定。
- 不支持Z值。输入时Z看成是X；输出值不允许设为X。注意 ？符号的特殊含义，它和在数字中的 ？符号意思不一样，在数字的表示中符号？和 z 含义相同。

注意！

在描述时序逻辑的UDP中，若在表中任何地方出现边沿触发的条件，则输入信号所有可能的边沿都要认真考虑并列出，因为默认的只是一种边沿的触发的条件，这将导致输出的不确定性。

可综合性问题：

任何一种工具都不能综合UDP，它只被用来建立基本的门级逻辑器件的逻辑仿真模型。

提示：

- 和行为模块相比较，用UDP来做仿真非常有效。为ASIC单元库的元件建立模型时应该使用UDP。
- 输出端口在一个以上的元件，应该对每个输出建立独立的UDP。
- 在表的第一行加上注释，指明每一列的含义。

举例说明：

```
primitive Mux2to1 (f, a, b, sel); // 组合UDP
  output f;
  input a, b, sel;
  table
// a b sel : f
    0 ? 0   : 0;
    1 ? 0   : 1;
    ? 0 1   : 0;
    ? 1 1   : 1;
    0 0 ?   : 0;
    1 1 ?   : 1;
  endtable
endprimitive

primitive Latch (Q, D, Ena);
  output Q;
  input D, Ena;
```

```
    reg Q; // Level sensitive UDP
    table
  // D Ena : old Q : Q
      0 0   :  ?      : 0;
      1 0   :  ?      : 1;
      ? 1   :  ?      : -; // 保持原先值
      0 ?   :  0      : 0;
      1 ?   :  1      : 1;
   endtable
  endprimitive

  primitive DFF (Q, Clk, D);
     output Q;
     input Clk, D;
     reg Q; // Edge sensitive UDP
     initial
       Q = 1;
      table
           / / Clk  D : old Q : Q
               r    0 :  ?    : 0; // Clock '0'
               r    1 :  ?    : 1; // Clock '1'
               (0?) 0 :  0    : -; // Possible Clock
               (0?) 1 :  1    : -; // " "
               (?1) 0 :  0    : -; // " "
               (?1) 1 :  1    : -; // " "
               (?0) ? :  ?    : -; // Ignore falling clock
               (1?) ? :  ?    : -; // " " "
                ?   * :  -    : -; // Ignore changes on D
      endtable
endprimitive
```

请参阅：

Module、Gate、Instantiation 的语法说明。

While　条件循环语句

只要控制表达式为真（即不为零），循环语句就重复进行。

语法

```
  wile {Expression}
     Statement
```

可综合性问题：
只有当循环块有事件控制（即@(posedge Clock)）才可综合。

举例说明：

```
reg [15:0]  Word
while (Word)
begin
  if  (Word[0])
     CountOnes = CountOnes + 1;
  Word = Word >>1;
End
```

请参阅：
For、Forever、Repeat 语句的说明。

Compiler Directives　　编译器指示

编译器指示是在源代码中对Verilog 编译器所发出的指令。在编译指示需要用反引号（`）做前导。编译器指示从它在源代码出现的地方开始生效，并一直继续生效到随后运行的所有的文件，直到编译器指示结束的地方或一直运行的最后的文件。

下面有Verilog 编译指示的摘要。摘要后面详细介绍了一些比较重要的编译指示。

注意!
编译器指示的生效依赖于编译时源代码中所包含文件的执行顺序。

Standard Compiler Directives　标准的编译器指示

在Verilog LRM中定义了以下编译器指示：

1) `celldefine 和 `endcelldefine
可用来作为分别加在模块的前面和后面的标记，以表示该模块是一个库单元（cell）。单元可被 PLI 子程序调用来做某种应用，比如延迟的计算。
例子:

```
`celldefine
module  Nand2 {...};
 . . .
endmodule
```

`endcelldefine

2) `default_nettype

改变Net类型的默认类型。如果没有该声明，默认的Net类型是wire 型。

例子: `default_nettype tril

3) `define 和 `undef

`define 定义一个文本宏，`undef 取消已定义的文本宏定义。

在编译的第一阶段期间, 宏（macro）被它所定义的文本字符串取代。宏也可以用来控制条件编译（请参阅 `ifdef)。想要知道关于 `define 应用的更多细节见下面说明。

4) `ifdef, `else 和 `endif

根据是否定义了特殊的宏，来指示编译器是否要编译这一段 Verilog 源代码。详细细节见下面。

5) `include

指示编译器读入包含文件的内容，并在`include所在的地方编译该文件。

例子： `include “definitions.v”

6) `resetall

把现行的已启动的所有编译器指示复位到原默认值。该编译指示可以写在每个Verilog源文件的第一行，以防止前面别的源文件的编译指示在该源文件编译时产生不需要的结果。

例子： `resetall

7) `timescale

定义仿真的时间单位和精度。细节请见下面说明。

8) `unconnected_drive 和 `nounconnected_drive

`unconnected_drive 编译指示把模块没连接的输入端口设置为上拉 pull up（pull1，即逻辑1）或为下拉 pull down（pull0, 即逻辑0)。`nounconnected_drive 编译指示把模块没连接输入端口的设置恢复到默认值，即把没连接的输入端口值设置为高阻浮动（Z)。

例子： `unconnected_drive pull0 //或 pull1 (即逻辑值为1)

Non-Standard Compiler Directives 非标准编译器指示

下面的编译指示并不属于Verilog HDL 语言的IEEE标准。但在CADENCE公司的Verilog LRM中提及。并不是所有的Verilog工具都支持以下这些编译指示。

1) `default_decay_time

若未明确给定衰减时间，则由该编译指示将其设置为默认的三态寄存器(trireg) 类型的线路连接（Net）的衰减时间。

例子：

```
`default_decay_time  50
`default_decay_time  infinite   //表示无衰减时间
```

2) `default_trireg_strength

把三态寄存器(trireg) 类型的线路连接（Net）的默认强度设置为整数。用整数来表示强度并不符合IEEE规定的Verilog语言标准，但仍属于Verilog语言非标准扩展部分。

例子：

```
`default_trireg_strength  30
```

3) `delay_mode_distribute、`delay_mode_path、`delay_mode_unit 和`delay_mode_zero 这些编译指示都会影响延迟的仿真方式。分布式延迟是在原语实例中的延迟、赋值延迟和线路连接延迟。路径延迟是在Specify(指定)块中定义的延迟。若用单位和零延迟代替分布式延迟和路径延迟将加快仿真的过程，但会丢失真实的延迟信息。在默认情况下，仿真器会自动选择最长的延迟仿真方式，即分布式延迟和路径延迟仿真方式。

4) `define

`define 定义一个文本宏。宏在编译的第一阶段被由它定义的文本所代替。在用参数和函数表达不适合或不允许的情况下，用宏可以提高 Verilog 源代码的可读性和可维护性。

语法：

```
{declaration}
`define Name[(Argument,...)] Text
{usage}
`Name [(Expression,...)]
```

在程序中位于何处：

宏可以在模块内或模块外定义。

规则：

- 像所有的编译指示一样，宏定义在整个文件中生效，除非被后面的`define、`undef 和`resetall编译指示改写或清除。宏定义没有范围的限制。
- 若定义的宏内有参数，即在宏文本中用到参数，则当宏调用时，宏的参数被实际的参数表达式所代替。

```
`define add(a , b)  a + b
  f = `add(1, 2);              //f= 1 + 2;
```

- 宏定义可以用反斜杠（\）跨越几行。新的一行是宏文本中的一部分。
- 宏文本不允许分下列语言记号：注释，数字，字符串，名称，保留名称，操作符。
- 不能把编译器指示名用作宏名。

注意！

- 所有的具体电路实现工具都不支持带参数的宏。
- 若定义了宏，则必须把撇号（`）写在宏名的紧前面才能调用该宏。没有撇号（`）打头的名，即使名称与宏名一致，则为独立的标识符与宏定义无关。
- 要区别撇号（`）和表示数制的前引号（‘）的不同。
- 不要用分号来结束宏定义，除非真要在用宏代替分号。否则会引起语法错误。

提示：

- 通常更喜欢用参数而不是用宏给无含义的字符起一个有含义的名字。
- 仿真时，用带参数的宏要比用同样功能的函数效率高。

举例说明：

本例子说明在分层设计中如何用文本宏来选择不同的模块实现。这在综合时很有用，特别是当必须用RTL源代码模块和已综合成门级电路的模块做混合仿真时。

```
`define  SUBBLOCK1 subblock1_rt1
`define  SUBBLOCK2 subblock2_rt1
`define  SUBBLOCK3 subblock3_gates
module TopLevel ...
  `SUBBLOCK1 sub1_inst (...);
  `SUBBLOCK2 sub2_inst(...);
  `SUBBLOCK3 sub3_inst(...);
 ...
endmodule
```

下面的例子说明带参数的文本宏的定义和调用：

```
`define nand (delay)  nand #(delay)
 nand(3)  (f,a,b);
 nand(4)  (g,f,c);
```

请参阅：

‘ifdef 的说明。

5) `ifdef

根据是否定义了特定的宏，来决定是否编译这部分Verilog源代码。

语法：

```
`ifdef MacroName
   VerilogCode...
[ `else
  VerilogCode...]
`endif
```

规则：

- 如果宏名已经用`define定义，只编译Verilog编码的第一块。
- 如果宏名没有定义和`else 指示出现，只编译第二块。
- 这些编译指示是可以嵌套的。
- 没被编译的代码仍然必须是有效的 Verilog 代码。
-

提示：

这些编译指示可以用来调试模块。例如，可以在同一个模块的两种形式之间切换（如布线前仿真模块和带布线延迟的门级仿真模块之间）或有选择地开启诊断信息的打印输出。

例子

```
`define primitiveModel
```

```
module Test
...
`ifdef primitiveModel
  Mydesign_primitives  UUT (...);
`else
  Mydesign_RTL UUT(...);
`endif
endmodule
```

请参阅：

`define的说明。

6) `timescale

定义时间单位和仿真精度

语法：

```
`timescale TimeUnit / PrecisionUnit
TimeUnit = Time Unit
PrecisionUnit = Time Unit
Time = {either} 1 10 100
Unit = {either} s ms us ns ps fs
```

规则：

- 像所有的编译指示一样，`timescale影响在该指示后的所有模块，无论位于同一个文件的还是位于独立编译的多个文件中的模块，直到碰到下一个`timescale或`resetall指示将其改写或复位到默认为止。
- 精度单位必须小于或等于时间单位。
- 仿真器运行的精度就是在`timescale指示中所定义的最小精度单位。所有的延迟时间都以精度单位为准取整。

提示：

在每个模块文件的第一句应写上`timescale指示，即使在模块中没有延迟，也是如此，因为有的仿真器必需要有`timescale指示才能正常工作。

举例说明：

```
`timescale  10ns / 1ps
```

请参阅：

$timeformat 的说明。

System task and function

系统任务和函数

Verilog语言包含一些很有用的系统命令和函数。用户可以像自己定义的函数和任务一样调用它们。所有符合IEEE标准的Verilog工具中一定都会有这些系统命令和函数。CADENCE公司的Verilog 工具中还有另外一些常用的系统任务和函数，它们虽并不是标准的一部分，但在一些仿真工具中也经常见到。

请注意，各种不同的 Veriog 仿真工具可能还会加入一些厂商自己特色的系统任务和函数。用户也可以通过编程语言接口（PLI）把用户自定义的系统任务和函数加进去，以便于仿真和调试。

所有的系统任务和系统函数的名称（包括用户自定义的系统任务），前面都要加$以区别于普通的任务和函数。下面是Verilog 工具中常用的系统任务和函数的摘要。详细资料在后面介绍。

标准的系统任务和函数

Verilog HDL的IEEE标准中包括下面的系统任务和函数：

- **$display, $monitor, $strobe, $write 等**
  用于把文本送到标准输出和或写入一个或多个文件中的系统任务。详细说明在后面介绍。

- **$fopen 和$fclose**
  $fopen(“FileName”);　　{Return an integer}
  $fclose(Mcd);
  $fopen 是一个系统函数，它可以打开文件为写文件做准备。
  而$fclose也是一个系统函数，它关闭由 $fopen 打开的文件。
  有关的详细说明在后面介绍。

- **$readmemb 和 $readmemh**
  $readmemb(“File”, MemoryName [,StartAddr[,FinishAddr]]);
  $readmemh(“File”, MemoryName [,StartAddr[,FinishAddr]]);
  把文本文件中的数据赋值到存储器中。有关的详细说明在后面介绍。

- **$timeformat[(Units,Precision,Suffix,MinFieldWidth)];**
  定义用$display等显示仿真时间的格式。有关的详细说明在后面介绍 。

- **$printtimescale**
  $printtimescale([ModuleInstanceName]);
  以如下格式显示一个模块的时间单位和精度：

Time scale of (module_name) is unit /precision。
如果没有参数，则显示模块的时间单位和精度。

- **$stop**

  $stop[(N)]; {N is 0,1,2}
  暂停仿真。可选的参数决定诊断输出的类型。 0 输出最少， 1 个多点， 2 输出最多。

- **$finish**

  $finish[(N)]; {N is 0,1,2}
  退出仿真，把控制权返回给操作系统。如果给出参数N，则根据N值打印不同的诊断信息，见下面的解释：
  - 0 ——不打印
  - 1 ——打印仿真时间和地点（默认值）。
  - 2 ——打印仿真时间和地点和仿真所使用的CPU时间和内存的统计数据。

- **$time,$stime,和$realtime**

  $time;
  $stime;
  $realtime;
  系统函数返回仿真的当前时间值。返回时间值的单位由调用该系统函数语句的模块的`timescale定义。
  - $time 返回一个根据时间单位四舍五入取整的64位无符号整数。
  - $stime返回一个截去高位保留低32位的无符号整数。
  - $realtime 返回一个实数。

  请注意，这些系统函数没有输入，与Verilog的其它函数不同。

- **$realtobits 和 $bitstoreal**

  $realtobits(RealExpression) {return a 64 bit value}
  $bitstoreal(BitValueExpression) {return a real value}
  实数和用位（bit）表示的数之间的互相转换。因为模块的端口不允许传输实数，故需要把实数转换为用位表示的数后才能输入/输出模块。请参阅Module的说明。

- **$rtoi 和 $itor**

  $rtoi(RealExpression) {return an integer}
  $itor(IntegerExpression) {rerurn a real number}
  实数和整数之间的互相转换。$rtoi把实数截断后转换为整数。$itor把整数转换为实数。

- **随机数产生函数**

  1) $random[(Seed)];
  2) $dist_chi_square(Seed, DegreeOfFreedom);
  3) $dist_erlang(Seed, K_ stage, Mean);
  4) $dist_exponential(Seed, Mean);
  5) $dist_normal(Seed, Mean, StandardDeviation);
  6) $dist_poisson(Seed, Mean);

7) $dist_t(Seed, DegreeOfFreedom);
8) $dist_uniform(Seed, Start, End);

当重复调用上述函数时，根据不同概率分布的随机数产生函数条件返回其相应的随机数序列。若伪随机序列的源种相同，则伪随机序列也总是一样的。请参考关于概率与统计理论的教科书，详细了解其中分布函数及其应用部分。

- **Specify Block Timing Checks 指定块内的定时检查系统任务**

1) $hold(ReferenceEvent, DataEvent, Limit [, Notifier]);
2) $nochange(ReferenceEvent, DataEvent,
StartEdgeOffset, EndEdgeOffset [, Notifier]);
3) $period(ReferenceEvent, Limit [, Notifier]);
4) $recovery(ReferenceEvent, DataEvent, Limit [, Notifier]);
5) $setup(DataEvent, ReferenceEvent, Limit [, Notifier]);
6) $setuphold(ReferenceEvent, DataEvent,
SetupLimit, HoldLimit [, Notifier]);
7) $skew(ReferenceEvent, DataEvent, Limit [, Notifier]);
8) $width(ReferenceEvent, Limit [, Threshold [, Notifier]]);

以上8个系统任务均为常用的定时检查系统任务。这些专用的系统任务只能在specify block（指定块）里被调用，详细说明请参阅后面的材料。

- **Value Change Dump Tasks 储存数值变化的系统任务**

1) $dumpfile(“FileName”);
2) $dumpvars[(Levels, ModuleOrVariable,...)];
3) $dumpoff;
4) $dumpon;
5) $dumpall;
6) $dumplimit(FileSize);
7) $dumpflush;

以上七个系统任务用于把数值的变化储存到 VCD 文件中。VCD 文件是把仿真激励或结果传递到另一个程序（例如一个波形显示程序）的一种手段。详见后面。

非标准的系统任务和函数

以下这些系统任务和函数在 CADENCE 公司的 Verilog 工具中有，但它们并不属于 IEEE 标准必须包括的范围。其中有部分系统任务和函数与 Verilog 仿真工具操作时的交互方式有关。如果仿真工具支持交互方式的操作，则接受这些系统任务和函数作为其指令。

$countdrivers

$countdrivers (Net, [IsForced, NoOfDrivers, NoOfDriversTo0,
NoOfDriversTo1, NoOfDrivrsToX]);

该系统函数能返回某指定的 Net 类型标量或 Net 型矢量的某个选定位上的驱动器个数。驱动器包括原语的输出和连续赋值语句（强迫（force）启动的除外）。若 Net 含有一个以上的驱动器时，该系统函数（$countdrivers）返回 0；其他情况下返回 1。在该系统任务中除第一个变量外，其余的都返回整型数。若 Net 为 force ，则 IsForced 返回 1 ，否则返回 0 。

NoOfDrivers 返回驱动器个数。其他变量返回数的总和等于 NoOfDrivers 。

- **$list**

  $list [(ModuleInstance)];

在交互模式中调用此系统函数可列出在本设计中当前（或指定）范围内的源程序。.

- **$input**

  $input(“FileName”);

从某个文本文件中读出交互命令。

- **$scope and $showscopes**

  $scope(ModuleInstance);

  $showscopes[(N)];

本系统命令用于在交互模式中设置和显示当前范围，若给定 N 并为非零，则还显示下面的范围。

- **$key, $nokey, $log and $nolog**

  $key[(“FileName”)];

  $nokey;

  $log[(“FileName”)];

  $nolog;

“key”文件记录用交互方式输入的命令，“log”文件记录在仿真期间所有写入标准设备的信息，而运行 $nokey 和 $nolog 系统任务可分别禁止这两项功能。用$key 和$log（无参数）可恢复其记录功能。如有参数，则$key 和$log 创建新的记录文件。

- **$reset、$reset_count 和 $reset_value**

  $reset[(StopValue[, ResetValue[, DiagnosticsValue]]);

  $reset_count; {Returns an integer}

  $reset_value; {Returns an integer}

系统任务$reset 使仿真器复位，并使仿真从头重新开始执行。StopValue 为 0 表示仿真器复位到交互模式，允许用户自己来启动和控制仿真。而非 0 值表示仿真将会自动地从头开始仿真。ResetValue 的值可以通过$reset_value 系统函数读出。DiagnosticsValue 是指复位前仿真工具所显示信息的类型。$reset_count 返回已调用$reset 系统任务的次数。$reset_value 返回传给$reset.系统任务的值。

- **$save、$restart 和 $incsave**

  $save(“FileName”);

  $incsave(“FileName”);

  $restart(“FileName”);

$save 将完整的仿真状态保存在文件中，$restart.可以读出保存的文件。$incsave 只保存自上次调用$save 后的变化。$restart 将仿真复位并把完整的或只记录变化的文件读出。若是$restart 只记录变化的文件，原完整的仿真状态记录文件必须存在，记录变化的文件会引用完整的仿真状态文件。

- **$showvars**

  $showvars[(NetOrRegister,...)];

在标准输出设备显示 Net 和寄存器的状态。这个系统任务用于交互模式。所显示的状态信息

在 Verilog LRM 工具中未作定义。状态信息可以包括当前的 Net 和寄存器值、这些 Net 和寄存器上的预定事件、以及 Net 的驱动器。如果未给出变量表，将显示所有当前范围的 Net 和寄存器。

- **$getpattern**

```
$getpattern( MemoryElement);
```

$getpattern 是一个只能用于连续赋值语句的系统函数，连续赋值语句的左边必须为 Net 类型标量的位拼接。$getpattern 常与$readmemb 和 $readmemh 一起使用，可从文本文件中提取测试矢量。当有大量的标量需要输入时，$getpattern 能提供快速的处理。

- **$sreadmemb and $sreadmemh**

```
$sreadmemb (Memory, StartAddr, FinishAddr, String, ...);
$sreadmemh (Memory, StartAddr, FinishAddr, String, ...);
```

这两个任务与$readmemb 和$readmemh 类似，只是存储器中的初始数据不是由文件输入，而是由一个或多个字符串输入。字符串格式与 $readmemb 和$readmemh 系统任务所要求的相应文件格式一致。

- **$scale**

```
$scale(DelayName); {Returns realtime}
```

将一模块的时间值转换为调用$scale 系统任务的模块中所定义的时间单位来表示。$scale 可以引用用模块层次命名的参数（如延迟值），并将它转换为调用$scale 的模块中所定义的时间单位来表示。

常用系统任务和函数的详细使用说明

$display 和 $write

把格式化文本输出到标准输出设备及仿真器日志或其他文件。

- **语法：**

```
$display( Argument,...);
$fdisplay( Mcd, Argument,...);
$write( Argument,...);
$fwrite( Mcd, Argument,...);
Mcd = Expression {Integer value}
```

- **规则：**

$display与 $write的唯一区别为前者在输出结束后会自动换行而后者不会自动换行。Arguments 可以是字符串或表达式或空格（,,）。字符串内可包含以下格式控制符。若包含格式控制符（%m 除外），则每个字符串后必须有足够的表达式来为字符串中的格式控制符提供数值。

字符串中也可包含以下扩展字符：

- \n 换行（Newline）
- \t 制表符（Tab）
- \” 双引号
- \\ 反斜杠
- \nnn 用八进制表示的 ASCII 字符

不定值和高阻值这样表示（注意：八进制数的每一个数字代表 3 位，而十制数和十六进制数的每一个数字代表 4 位）：对十进制数而言，若有某个数字为不定值和高阻值则写作 x，z，X，Z。若用大写的 X，Z 表示，则该数字中并非所有位（bit）为不定值和高阻值，若用小写 x，z 表示，则表示该数字中所有位（bit）为不定值和高阻值时。若参数表含二相邻逗号，则输出显示或打印一空格。

- **格式控制符**

在字符串中允许出现下面这些格式控制符：

1) %b %B 二进制数（Binary）
2) %o %O 八进制数 （Octal）
3) %d %D 十进制数 （Decimal）
4) %h %H 十六进制数（Hexadecimal）
5) %e %E %f %F %g %G 实型数（Real）
6) %c %C 字符（Character）
7) %s %S 字符串（String）
8) %v %V 二进制数和强度（Binary and Strength）
9) %t %T 时间类型数（Time）
10) %m %M 分级实例名 （Hierarchical Instance）

格式控制符 %v 按如下的形式打印出变量的强度值：若强度值为supply， 则打印 Su; 若为strong，则打印St；若为Pull ，则打印Pu；若为Large，则打印La；若为Weak，则打印We；若为Medium，则打印Me；若为Small，则打印Sm；若为Highz，则打印Hi。 %v也能把变量值打印为H和 L（这些值若用 %b 格式控制符，则只能打印为X）。% 号后常跟有一个数用于表示打印变量值区域的宽度（例如%10d，表示至少保留 10位宽度给要打印的十进制数）。对十进制数，高位不足此值者以空格代替，其他进制以0代替，若% 号后的数为0，则表示打印变量值区域的宽度随其值的位数自动调节。

Verilog HDL 实型数的格式符(%e, %f and %g) 其格式控制功能和C语言的格式符完全一样。例如，%10.3g指至少保留 10位宽度给要打印的十进制数，小数点后还保留3个数字位。
若相应的变量未用格式控制符声明，则默认为是十进制数。有些系统打印任务有其自己的缺省值，如$displayb, $fwriteo，$displayh 的缺省值分别是二进制，八进制，十六进制。

- **举例说明：**

```
$display ( “Illegal opcode %h in %m at %t”, Opcode, $realtime);
$writeh  (“Register values (hex .): “ , reg1 , ,  reg2, ,  reg3, ,  reg4, ” \n”);
```

请参阅：

$monitor, $strobe 的说明

$fopen and $fclose

$fopen 是用于打开某个文件并准备写操作的系统任务，而 $fclose 则是关闭文件的系统任务。把文本写入文件还需要用$fdisplay、$fmonitor 等系统任务。

语法：

```
$fopen(“FileName”); {Returns an integer}
$fclose( Mcd);
Mcd = Expression {Integer value}
```

在程序中位于何处：

请参阅 Statement 的说明;

规则：

- 一般情况下一次最多可打开32个文件，但若所用的操作系统不同，一次最多可打开的文件数可能不到32。当调用$fopen时，它返回一个32 位（bit）（与文件有关）的无符号多通道描述符或者返回0值，0值表示文件不能打开。多通道描述符可以被认为是32个标志每个代表32个文件中的一个。多通道描述符的第0位与标准输出设备有关，第1位为第 1 个文件打开的标志位，第 2 位为第 2 个文件的标志位，依次类推。当输出文件的系统任务，如$fdisplay，被调用时，其第一个参数为多通道描述符，它表示向何处写。文本被写入那些多通道描述符内标志位已设的相应文件中。

举例说明：

```
integer MessagesFile, DiagnosticsFile, AllFiles;
    initial
        begin
            MessagesFile = $fopen(“messages.txt”);
            if (!MessagesFile)
                begin
                    $display(“Could not open \”messages.txt\””);
                    $finish;
                end
            DiagnosticsFile = $fopen(“diagnostics.txt”);
            if (!DiagnosticsFile)
                begin
                    $display(“Could not open \”diagnostics.txt\””);
                    $finish;
                end

        AllFiles = MessagesFile | DiagnosticsFile | 1;
        $fdisplay(AllFiles, “Starting simulation ...”);
```

```
        $fdisplay(MessagesFile, “Messages from %m”);
        $fdisplay(DiagnosticsFile, “Diagnostics from %m”);
            ...
        $fclose(MessagesFile);
        $fclose(DiagnosticsFile);
    end
```

请参阅：

$display, $monitor, $strobe 的说明。

$monitor 等

当$monitor 系统任务所指定的参数表中中任何一个或多个 Net 或寄存器类型变量值发生变化时，便立即显示一行文本。此系统任务常用于测试模块中，以监测仿真行为的细节。

语法：

```
$monitor( Argument,...);
$fmonitor( Mcd, Argument,...);
$monitoron;             {turns monitor flag on}
$monitoroff;        {turns monitor flag off}
Mcd = Expression    {Integer value}
```

规则：

上面这些系统任务在变量使用的语法上与$display 系统任务完全相同。有一点与$display 系统任务不同：只能同时运行一个$monitor 系统任务。但$fmonitor 系统任务却能同时运行多个。第二次或下一次调用$monitor 系统任务，就把上一次正在执行的$monitor 系统任务取消了，用新的$monitor 系统任务取而代之。$monitoroff 系统任务关闭监视的功能，而$monitoron 则恢复监视的功能，它能把现存的$monitor 进程所监测到的信号不管其值是否变化立即显示出来。对$fmonitor 而言，没有与之对应的$monitoron 和 $monitoroff 系统任务。系统函数$time、 $stime 和 $realtime 不会从$monitor 或 $fmonitor 等系统任务触发出一行显示。

提示：

在测试模块里使用$monitor 可以从任何一种 Verilog 兼容的仿真器获得仿真结果。用于生成波形图显示的任务往往与仿真器相关。

举例说明：

```
initial
$monitor ( “ %t : a = %b, f = %b ”,$realtime, a, f );
```

请参阅：
$display, $strobe, $fopen 语句的说明。

$readmemb 和 $readmemh

把文本文件中的数据读到存储器阵列中，以对存储器变量进行初始化。此文本文件的内容可以是二进制格式（用 $readmemb）的， 也可以是十六进制格式（用$readmemh）的。

语法：

```
{System task call}
$readmemb (“File”, MemoryName [, StartAddr[, FinishAddr]]);
$readmemh (“File”, MemoryName [, StartAddr[, FinishAddr]]);
{Text file}
{either} WhiteSpace DataValue @ Address
WhiteSpace = {either} Space Tab Newline Formfeed
DataValue = {either}
BinaryDigit... {$readmemb}
HexDigit... {$readmemh}
Address = HexDigit...
```

规则：

- 第一个参数是 ASCII 文件名，文件中可以包含空格、Verilog 注释语句、十六进制地址和二进制或十六进制数据。
- 第二个参数是存储器阵列名。
- 数据的位宽必须与存储器阵列的每个存储单元的位宽相同,而且每个数据之间必须用空格间隔开。数据被一个挨一个地读入连续相邻的存储器阵列中，从存储器阵列的第一个地址（若指定起始地址，则从指定的起始地址）开始，直到数据文件结束或直到存储器阵列的最后一个地址（若指定结束地址，则到指定的结束地址）为止。
- 地址均用十六进制数字表示且以@符号开头（对$readmemb 亦然）。当遇到一个地址后，下一个文本数据将被读入这个地址的存储单元。

可综合性问题：
不可综合。 综合工具忽略这些系统任务的存在。在可综合的设计里，从存储器阵列导出的触发器不能用这种方法初始化。如果需要上电复位对存储器阵列(RAM)初始化，则必须对其明确地编码。

提示：
存储器阵列可以储存从文本文件读出的激励源。这是把数据读进 Verilog 仿真器的唯一方式，而无须另外使用编程语言接口（PLI）或非标准语言扩展来做到这一点。

举例说明：

```
module Test;
```

```
    reg a,b,c,d;
    parameter NumPatterns = 100;
    integer Pattern;
    reg [3:0] Stimulus[1:NumPatterns];
    MyDesign  UUT (a, b, c, d, f);
    initial
        begin
            $readmemb("Stimulus.txt", Stimulus);
            Pattern = 0;
            repeat (NumPatterns)
                begin
                    Pattern = Pattern + 1;
                    {a,b,c,d} = Stimulus[Pattern];
                    #110;
                    end
        end
    initial
        $monitor("%t a= %b b= %b c= %b d= %b : f= %b", $realtime, a, b, c, d, f);

endmodule
```

$strobe

在所有事件都已处理完毕后的时刻打印出一行格式化的文本。

语法：

```
$strobe( Argument,...);
$fstrobe( Mcd, Argument,...);
Mcd = Expression {Integer value}
```

规则：

本系统任务（$strobe）有关参数以及文本打印的语法与系统任务$display 完全一样。 但$strobe 只打印调用此系统任务的时刻且当所有活动事件都已结束后的信息，其中可包括所有阻塞和非阻塞赋值产生的效果。

提示：

在写仿真激励模块时，若想打印出仿真结果，应优先考虑使用$strobe系统任务。因为与使用$display或$write比较，系统任务$strobe 可以保证显示出写入Net和寄存器类型变量的是一个稳定的数值。

举例说明：

```
    initial
        begin
```

```
        a = 0;
        $display(a); // displays 0
        $strobe(a);  // displays 1 ...
        a = 1;       //... because of this statement
    end
```

请参阅：

$display, $monitor, $write 语句的说明。

$timeformat

定义仿真时间的打印格式。系统任务 $timeformat 应配合格式控制符 %t 使用。

语法：

$timeformat[(Units, Precision, Suffix, MinFieldWidth)];

规则：

- Units（单位）是指打印的时间单位，它是一个 0 到 -15 之间整型数，0 表示秒（s），-3 表示毫秒（ms），-6 表示微秒（us），-9 表示纳秒（ns），-12 表示皮秒（ps），-15 表示浮秒（femtosecond），中间的整数也可用，如 -10 表示 100 皮秒（ps），依此类推。
- Precision 是指打印的十进制数小数点后保留的位数。
- Suffix 指打印时间值后跟的字符串。
- MinFieldWidth 指打印出的字符的最少个数，其中包括前面的空格。若需要打印的字符多，则需要取较大的整数。
- 缺省形式，即不指定参数，自动设置为：Units（单位）为仿真的时间精度；Precision（精度）为 0；Suffix 无；MinFieldWidth 为 20。

提示：

在使用$display、$monitor 或其他显示任务时，应使用 ‘timescale, $timeformat 和 $realtime（并配合 %t）来指定和显示仿真时间。

举例说明：

```
$timeformat  (-10, 2, “ x100ps”, 20);   // 20.12 x100ps
```

请参阅

‘timescale, $display 的说明。

Stochastic Modelling 随机模型

Verilog 提供了一整套系统任务和函数，可用来启动随机序列的生成和管理以支持建立随机

模型。

语法：

```
$q_initialize( q_id, q_type, max_length, status);
$q_add( q_id, job_id, inform_id, status);
$q_remove( q_id, job_id, inform_id, status);
$q_full( q_id, status); {Returns an integer}
$q_exam( q_id, q_stat_code, q_stat_value, status);
```

在程序中位于何处：

请参阅 Statement 的说明。

概论：

所有这些系统任务和函数的参数都是整型数。每个系统任务和函数都返回一整数型的状态（status）值，它为下列值之一：

0- OK
1- 队列已满：不能再增加工作($q_add)
2- 未定义的 q_id
3- 队列空：不能再删除工作($q_remove)
4- 不支持的队列形式：不能创建这个队列($q_initialize)
5- 最大长度小于等于 0：不能创建这个队列($q_initialize)
6- 两个相同的 q_id:：不能创建这个队列($q_initialize)
7- 内存不足：不能创建这个队列($q_initialize)

系统任务 $q_initialize

创建一个队列。q_id (输出)是唯一的队列标识符，当程序需要调用多个队列任务和函数时，可用该标识符来区别各个队列。 q_type (输入)可为 1 或 2，1 表示 FIFO（先进先出）队列，2 表示 LIFO（后进先出）队列。max_length (输入)为队列所允许的最多的输入个数（即最大长度）。

系统任务 $q_add

向队列加进一个入口。q_id (输入)表示向哪个队列加输入口。job_id (输入)表示是哪个工作（job），它通常为一整型数，每次向队列加入一个新元素其值加 1，这样当队列的某一元素需要移走，可以用 job_id 来识别。 inform_id (输入)用于定义与队列入口有关的信息，由用户自己来定义。

系统任务 $q_remove

从队列取一个入口。 q_id (输入)表示从哪个队列取走该入口。 job_id (输出)确定是哪个工作(请参阅 $q_add 的说明)。inform_id (输出)是由$q_add 储存的数值。

系统任务 $q_full

检查队列是否满。若返回值为 1 则队列满；为 0 则不满。

系统任务 $q_exam

取得队列不同类型的统计信息。下列描述中提及的时间是基于何时队列元素被加入到队列中（到达时间）以及队列元素从加入队列到被删除出去的时间差（等待时间）。时间单位为仿真的时间精度。其中 q_stat_value (输出)参数返回取得的消息，而其中 q_stat_code (输入) 参数可以取 1—6，分别表示要求取得的信息类型：

1. 当前队列长度。
2. 平均达到时间间隔。
3. 最大队列长度。
4. 最短等待时间
5. 当前队列中队列元素的最长等待时间。
6. 本队列的平均等待时间。

举例说明：

```
module Queues;
    parameter Queue = 1; // Q_id
    parameter Fifo = 1, Lifo = 2;
    parameter QueueMaxLen = 8;
    integer Status, Code, Job, Value, Info;
    reg     IsFull;

   task Error; // Write error message and quit
       ...
   endtask

   initial
    begin
       // 生成后进先出队列，其标号为为 1, 队列长度为 8。
          $q_initialize (Queue, Lifo, QueueMaxLen, Status);
            if ( Status )
              Error("Couldn't initialize the queue");
       // 向1号队列加入从1号到8号共8个工作，每个job 之间间隔10个单位时间
       // 每次从1号队列加入的信息为job号加100
          for (Job = 1; Job <= QueueMaxLen;  Job = Job + 1)
            begin
                  #10  Info = Job + 100;
                $q_add (Queue, Job, Info, Status);
                  if ( Status )
                      Error("Couldn't add to the queue");
                  $display("Added Job %0d, Info = %0d", Job, Info);
                  $write("Statistics: ");
   /**要求取得有关当前队列长度、平均达到时间间隔、最大队列长度、最短等待时间、
   当前队列中队列元素的最长等待时间，和本队列的平均等待时间共 6 种队列信息 **/

                      for ( Code = 1; Code <= 6; Code = Code + 1 )
```

```
                        begin
                           $q_exam(Queue, Code, Value, Status);
                            if ( Status )
                                Error("Couldn't examine the queue");
                           $write("%8d", Value);  //显示 6 种队列信息
                        end
                    $display("");
                end
    //  队列此时应是满的
        IsFull = $q_full(Queue, Status);
            if ( Status )
                Error("Couldn't see if queue is full");
            if ( !IsFull )
            Error("Queue is NOT full");
    // 去除工作
             repeat (10 )
             begin
               #5  $q_remove(Queue, Job, Info, Status);
               if ( Status )
                    Error ("Couldn't remove from the queue");
               $display("Removed Job %0d, Info = %0d", Job,Info);
               $write("Statistics: ");
               for ( Code = 1; Code <= 6; Code = Code + 1 )
                begin
                    $q_exam(Queue, Code, Value, Status);
                   if ( Status )
                        Error("Couldn't examine the queue");
                    $write("%8d", Value);
                end
               $display("");
            end
     end
endmodule
```

请参阅：

$random、$dist_chi_square 等系统任务的说明。

Timing Checks　定时检查

Verilog 提供了一些系统任务，这些系统任务仅能在“specify block”（ 指定块）里调用，以进行常见的定时检查。

语法：

```
$hold( ReferenceEvent, DataEvent, Limit [, Notifier]);
$nochange( ReferenceEvent, DataEvent, StartEdgeOffset, EndEdgeOffset [, Notifier]);
$period( ReferenceEvent, Limit [, Notifier]);
$recovery( ReferenceEvent, DataEvent, Limit [, Notifier]);
$setup( DataEvent, ReferenceEvent, Limit [, Notifier]);
$setuphold( ReferenceEvent, DataEvent, SetupLimit, HoldLimit [, Notifier]);
$skew( ReferenceEvent, DataEvent, Limit [, Notifier]);
$width( ReferenceEvent, Limit [, Threshold [, Notifier]]);

ReferenceEvent = EventControl PortName [&&& Condition]
DataEvent = PortName
Limit = {either} ConstantExpression SpecparamName
Threshold = {either} ConstantExpression SpecparamName
EventControl = {either}
   posedge
   negedge
edge [ TransitionPair,... ]
TransitionPair = {either}  01 0x 10 1x x0 x1
Condition = {either}
ScalarExpression
~ ScalarExpression
  ScalarExpression == ScalarConstant
  ScalarExpression === ScalarConstant
  ScalarExpression != ScalarConstant
  ScalarExpression !== ScalarConstant
```

规则：

- 参考事件（ReferenceEvent）的变化提供了定时检查的时间基准，参考事件（ReferenceEvent）必须通过模块的输入口（input）或输入/输出口（inout）引入。
- 数据事件（DataEvent）的变化会启动定时检查，数据事件也必须通过模块的输入口（input）或输入/输出口（inout）引入。
- 如果参考事件与数据事件同时发生，这时虽不会产生建立违例报告，但会产生保持违例报告。
- 对于系统任务$width, 脉冲如果低于门限（Threshold）参数的设定（若设置了门限），则不会发生违例报告。
- 下列定时检查系统任务中的参考事件（ReferenceEvent）必须是沿触发的：$width, $period, $recovery, $nochange。
- 以上系统任务中 ReferenceEvent 参数都可以用关键字 **edge**，除了$recovery 和 $nochange 这两个系统任务例外，它们的 ReferenceEvent 参数只能用 **posedge** 和 **negedge**。
- 使用&&&做注释的条件定时检查仅在条件为真时才执行。
- 在以上的系统任务里如果设置了 notifier 参数，则其必须为寄存器类型变量，当违例发生时，寄存器变量数值发生变化：若原为不定值则变为 0，若原为 0 值则变为 1，若

原为 1 则变为 0，若原为高阻则不变。

注意！
这些系统任务仅能在 specify （指定）块中调用 ，而不能用作程序声明语句。
参数 ReferenceEvent 和 DataEvent 在系统任务$setup 里是颠倒的。

提示：
若条件比较复杂，应在指定块（specify block）外描述条件，而把驱动条件的信号（wire 或 reg 类型）放在指定块内。

举例说明：

```
reg Err, FastClock;     // Notifier registers
specify
     specparam Tsetup = 3.5, Thold = 1.5,
     Trecover = 2.0, Tskew = 2.0,
     Tpulse = 10.5, Tspike = 0.5;
     $hold(posedge Clk, Data, Thold);
     $nochange(posedge Clock, Data, 0, 0 );
     $period(posedge Clk, 20, FastClock)];
     $recovery(posedge Clk, Rst, Trecover);
     $setup(Data, posedge Clk, Tsetup);
     $setuphold(posedge Clk &&& !Reset, Data, Tsetup, Thold, Err);
     $skew(posedge Clk1, posedge Clk2, Tskew);
     $width(negedge Clk, Tpulse, Tspike);
endspecify
```

请参阅：
Specify, Specparam 的说明。

Value Change Dump

以下七个系统任务用于把数值的变化储存到 VCD 文件中。VCD 文件是把仿真激励或结果传递到另一个程序（例如一个波形显示程序）的一种手段。

语法：

```
$dumpfile(“FileName”);
$dumpvars[( Levels, ModuleOrVariable,...)];
$dumpoff;    {suspend dumping}
$dumpon;         {resume dumping}
$dumpall;        {dump a checkpoint}
$dumplimit ( FileSize);
$dumpflush;  {update the dump file}
```

在程序中位于何处：

请参阅 Statement 说明。

规则：

- 系统任务$dumpvars 中的参数 Levels 表示想要把指定模块的哪些层次的数值变化记录到 VCD 文件中：若设置为 1 表示仅记录指定层次模块中的变化，0 表示不但记录该层次模块还记录所有与该模块有关的下层模块中的变化。
- 如果没有设置任何参数，则设计中所有变量的变化均记录到 VCD 文件。
- FileSize 参数是用于设置 VCD 文件可记录的最多字节数。
- 在测试程序中可写多个系统任务$dumpvars 调用，但是每个调用必须在同一时刻（通常在仿真开始时）。

举例声明：

```
module Test;
    ...
    initial
        begin
            $dumpfile(“results.vcd”);
            $dumpvars(1, Test);
        end
// Perform periodic checkpointing of the design.
    initial
        forever
            #10000 $dumpall;
endmodule
```

Command Line Options 命令行的可选项

虽然怎样选用启动 Verilog 仿真器工作的命令行可选项并不是的 Verilog 语法的一部分，大多数有关 Verilog 语法学习参考材料里也不提供这方面的资料，但是为了更快掌握仿真工具还是有必要介绍一些这方面的资料，因为绝大多数仿真器都支持一些常见共同的 Verilog 编译命令选项（尽管有的 Verilog 仿真工具还有自己的一些命令选项），熟练地掌握这些共同的选项能更有效地进行仿真，提高 Verilog 仿真工具的使用效果。

UNIX Verilog 编译命令选项分为两类：一类是一个字符的，前面带有一个减号 -（如 -s）；另一类是多个字符的，前面带有一个加号（如 +word）。有些 UNIX Verilog 编译命令选项后还可跟一个值，例如跟一个文件名（如 -f file）。

下面介绍一些最有用和最常见的 Verilog编译命令选项（注意：并非所有仿真器都支持这些选项）：

-f CommandFile	除从命令行输读入命令选项外，还从命令文件读入更多的命令选项
-k KeyFile	在 KeyFile 里记录仿真期间所有键入的交互命令
-l LogFile	除了在显示器输出外还把所有仿真信息（包括$display 等的输出）记录

到 LogFile 中。

-r SaveFile　从由非标准的系统任务$save 生成的文件再次开始仿真。

-s　在 0 时刻中断仿真器，以便采用交互方式来控制仿真的进行。

-u　将 Verilog 源代码中的字符都视为大写字符（字符串除外），用此选项时要小心。

-v LibraryFile　在 LibraryFile 中寻找设计文件中缺少的 UDP 或模块。只有那些在设计文件中并未定义但已实例引用的 UDP 或模块编译器才会在 LibraryFile 中寻找。而设计中没有引用的在 LibraryFile 中 UDP 或模块不会进行编译。

-y LibraryDirectory

在LibraryDirectory中的文件中寻找设计文件中缺少的UDP或模块，期望在该库的目录中有一个文件已定义了一个与其同名的模块。如果给出+libext+扩展名 的命令行选项，则是指在搜寻文件时将带扩展名搜寻。例如，-y mylib + libext+.v 是指在mylib目录中，在扩展名为.v的文件范围内搜寻在设计中未定义的同名的模块。

+define+ MacroName

定义一段文字作宏名（无值）。这种零值的宏可以用于 ‘ifdef 中来控制（条件）编译的范围。

+incdir+ Directory[+ Directory...]

定义搜索路径，搜索用‘include 包含的文件。搜索开始于当前路径，如果没有找到，就去由+incdir+定义的搜索路径依次去找。

+libext+ Extension

定义库文件扩展名。（请参阅 –y 的说明）

+notimingchecks

关闭指定块的定时检查，此举可以加速仿真，或抑制伪定时错误信息，使用此选项时需小心。

+mindelays, +typdelays, +maxdelays

以上分别是指仿真时使用最小，典型和最大延迟，缺省为使用典型延迟。仿真时不要混淆以上三种延迟。

注意！

Verilog 仿真器不能检查出+号后选项参数的拼写错误，这是因为 Verilog 命令行允许用户自己来定义+号后的参数，所以一定小心注意选项的拼写，例如“ +maxdelays”要注意拼写正确。