ELECTRONIC ENGINEER

XIDIAN UNIVERSITY PRESS

Basis of IC Design

IC 设计基础

任艳颖　王　彬　编著

Specially Designed for Engineers and Technicians of Electronics

西安电子科技大学出版社

http://www.xduph.com

IC 设计基础

任艳颖　王　彬　编著

西安电子科技大学出版社

2003

内 容 简 介

作为IC的一名设计者，应当精通电路基础结构、硬件设计语言、EDA工具、应用协议等知识。本书从工程开发角度出发，结合实际，系统介绍了这些内容，可帮助读者了解标准化设计流程，提高设计技能，跨越芯片设计的门槛。

全书共分7章。第1章介绍了IC设计流程、常用工具的使用、Verilog设计语言；第2章介绍了时序电路的设计；第3章对综合工具DC进行了说明，并分析了基本语言结构的硬件实现；第4章给出了一些常用模块的构造；第5章对存储器的结构及设计进行了说明；第6章介绍了图像视频芯片的设计；第7章介绍了CISC处理器和RISC处理器的设计。本书每章之后附有练习，便于读者检测掌握的程度。

突出应用，强调由电路结构学习设计语言、强调思维方式“硬件化”、强调标准化的设计风格，是本书最大的特色。书中的代码与脚本都经过精心挑选，具有典型性，读者可在实践中借鉴。

本书可作为IC设计培训教材，也可作为有关技术人员的参考书。

图书在版编目（CIP）数据

IC设计基础／任艳颖，王彬编著. —西安：西安电子科技大学出版社，2003.5

ISBN 7-5606-1217-2

Ⅰ.I… Ⅱ.①任… ②王… Ⅲ.集成电路—电路设计 Ⅳ.TN402

中国版本图书馆CIP数据核字（2003）第016303号

策　　划　戚文艳　陈宇光

责任编辑　戚文艳

出版发行　西安电子科技大学出版社（西安市太白南路2号）

电　　话　(029)8242885　8201467　　邮　编　710071

http://www.xduph.com　　E-mail: xdupfxb@pub.xaonline.com

经　　销　新华书店

印刷单位　西安文化彩印厂

版　　次　2003年5月第1版　2003年5月第1次印刷

开　　本　787毫米×1092毫米　1/16　印张23

字　　数　541千字

印　　数　1~4 000册

定　　价　32.00元

ISBN 7-5606-1217-2／TN·0217

XDUP 1488001-1

* * * 如有印装问题可调换 * * *

序　言

从沙子中提出高纯度的硅，制成芯片，单位价值可超过黄金，从这个意义上讲，集成电路设计是真正的点石成金术。

作为通信、计算机、多媒体等众多电子信息产品的核心，集成电路的发展水平对一个国家的科技实力有重大影响。集成电路水平最高的是美国，其次是日本。与它们相比，我们的差距不小，要迎头赶上。近年来，国内集成电路水平得到长足发展，特别是随着一批晶圆厂的开工，直接带动了上游设计公司的发展。这个行业的产值、设计能力、从业人员的规模，都较以往有了质的飞跃。就生产能力而言，我们与国外的差距已大大缩小，因而低下的设计能力已成为急需克服的瓶颈。

由于历史原因，我国集成电路的发展曾一度停滞。对于40年来一直按照摩尔定理突飞猛进的集成电路产业来说，这段停滞期使我们远远落后了。这一点，设计公司与高等学校相关院系的感受尤为明显：由于设计水平低，技术积累匮乏，我们的设计公司只能进行一些低层次的重复开发，大都仅限于几万门的设计；由于人才出现断层，作为人才高地的高校无法提供足够的培训和教育资源。提高设计者的水平非一日之功，提供满足各层次读者需求的教材却是当务之急。现在虽然已经有一些论述集成电路设计的书籍问世，但大都侧重于VHDL/Verilog语言的介绍，或者FPGA工具的解释，或者半导体集成电路理论的阐述，从工程角度出发，展示集成电路设计的实际过程及经验的书籍较少。这种状况，显然不能满足实际需求。

本书结合工程实践，向读者展示了集成电路，特别是专用集成电路设计的全过程，内容涵盖电路基础、EDA工具、设计语言直至特定应用的规划。值得一提的是，作者将自己的经验和体会融入本书中，相信会让读者耳目一新。

郑世宝
于上海交通大学
2003年4月

前　言

能够亲眼见证自己的成果为千家万户所用，从这个意义上讲，IC 设计是一个很有意思的行业。不过，IC 设计也是一个充满挑战的领域。从数万个到数百万个乃至上亿个元器件设计规模、千差万别的应用、错综复杂的构造、日新月异的进步，要求设计者十八般武艺样样皆通 IC 工程师是一种典型的“杂食动物”，既要了解 CPU 的构造、SDRAM/EEPROM/闪存的控制、时钟、总线、射频、版图这样的电路知识，又需要熟悉 Verilog/VHDL 等硬件设计语言，还要掌握主流的综合工具、仿真工具、静态时序分析工具，而更重要的是还要对所做的应用有透彻的了解。比如，要做蓝牙基带芯片，就必须熟悉蓝牙的协议；要做非接触式 IC 卡，就必须熟悉 14443 标准；要做数字电视芯片，就必须熟悉 MPEG/DVB 标准。掌握这些知识，不是一日之功（而且，别忘了，这些东西天天在发展，你也得天天补充新知识），修得正果，非得经过八十一难不可。

另一方面，这个领域的初学者往往得不到有效的帮助。国内产业化的 IC 设计刚刚起步，可供学习的资源匮乏，加之许多有经验的工程师不愿意将自己的经验拿出来共享，所以，在成为合格的 IC 工程师之前，许多人都会经历一个痛苦期：不知道从何着手，不知道该学些什么，得不到指导，无法增长实际经验。

电路结构、设计语言、开发工具、应用这几个方面的知识，是初学者必须掌握的。这些要靠多实践，才能真正掌握。最好是有高手带着，参与实际项目的开发。几个项目做下来，就小成了。但良师难求，所以我们期望本书能够扮演这种指导者的角色，帮助初学者顺利跨过最初的几步。因此，我们无意于将它写成一本 Verilog 的语法书，或者某种工具的说明，或者半导体集成电路教材，而是按照工程上开发芯片的要求，介绍必须掌握的最重要的知识。

本书的第 1 章是 IC 设计基础。这一章给读者提供电路基础与工具软件方面的知识，介绍了实际芯片设计的流程，包括规格定义、架构选择、时序规划等重要内容。在 IC 设计中，常用 FPGA 进行原型机验证，所以本章也对 FPGA 的结构进行了说明。本章对一些常用工具软件的使用进行了说明（包括 Altera 的 quartus 与 xilinx 的 ISE，仿真工具 ModelSim 与 NC simulator，综合工具 FPGA Express 等）。此外，还对应用最广的硬件设计语言 Verilog 进行了介绍。由于本书并不是一本 Verilog 语法书，所以我们只写了那些在设计与仿真时最常用的东西。

第 2 章是时序电路设计，对时钟策略、时钟偏移、总线、同步等重要概念进行了阐述。

综合是IC设计很重要的一步。对电路性能有重要影响。在第3章中，我们着重阐述了最流行的综合工具DC Compiler中一些重要概念、使用方法、脚本文件的书写等。本章中还列出了一些基本的程序描述与相应的电路结构。希望大家能够好好研究这个列表，弄清楚所写的程序到底会综合出什么结果。

第4章介绍了一些重要基本模块的实现，包括数学运算单元、编码器解码器、存储控制器、异步传输收发器、CRC等内容。掌握了这些基本单元，就可以用搭积木的方法来完成较大的设计。

存储器的知识是IC设计者必须掌握的。在第5章中，我们对各种存储器的结构进行了阐述，并对模拟电路的设计进行了说明和演示。

接下来，我们讲述了实际芯片的开发。第6章介绍了视频芯片的设计，第7章介绍了微处理器的设计。读者可以通过这两章了解开发应用芯片的步骤和方法。视频与微处理器的设计都非常有意思，相信许多人都会感兴趣。

由于篇幅限制，我们只是将那些最重要的东西，结合自己的体会进行了论述，希望读者在本书帮助下，能够顺利地开始自己的IC设计历程。

本书错误与不足之处，欢迎大家批评指正。

作　者

2003年4月

目　录

第1章 IC设计基础

1.1 系统设计流程

在设计系统时，我们常采用自顶向下的思路。首先，是设计系统级的功能行为，接下来我们要将系统分解为不同的模块，用 HDL 代码实现。然后将代码综合为具体的门级实现。IC(Integrated Circuit，集成电路)设计中的基本概念如图 1.1 所示。

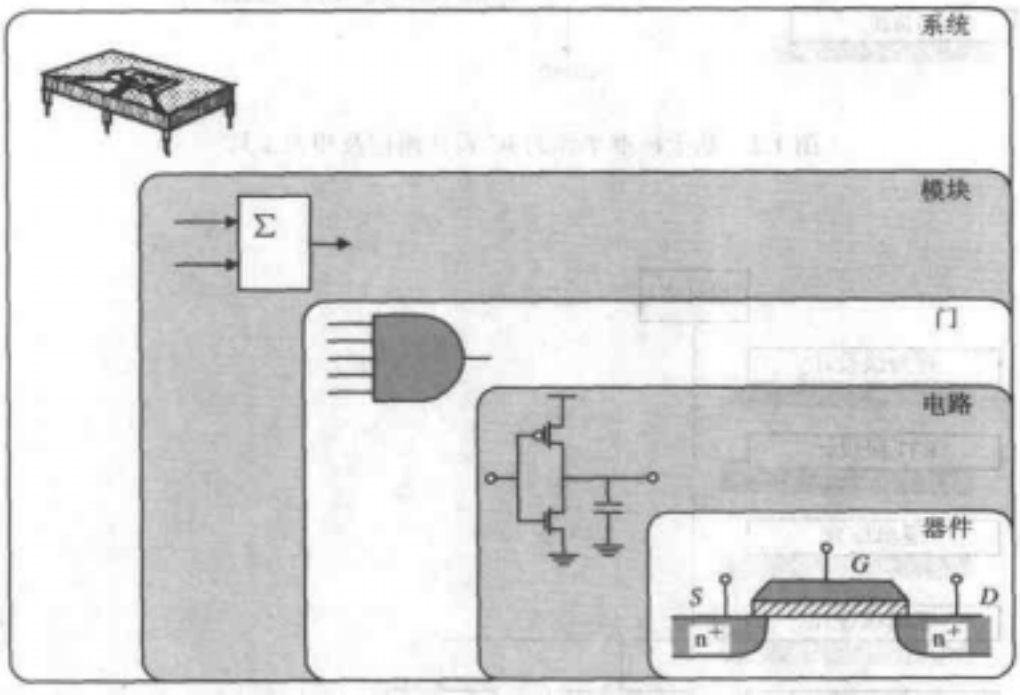


图 1.1 IC 设计中的基本概念

IC 设计分为两类：基于标准单元的设计和基于全定制的设计。图 1.2 是基于标准单元的 IC 设计流程及相关工具。图 1.3 是基于全定制的 IC 设计流程及相关工具。

图 1.4 是一个系统开发流程的例子。

IC 产品的设计过程是：设计者根据设计要求，提出设计构思，并将这个构思逐步细化，直到具体代码实现；再由代码综合出门级网表，生成版图，最终制成产品。在 IC 产品的设计中，好的设计思想价值千金，当然，有了好的设计思想之后，也需要高水平的设计技能来实现。

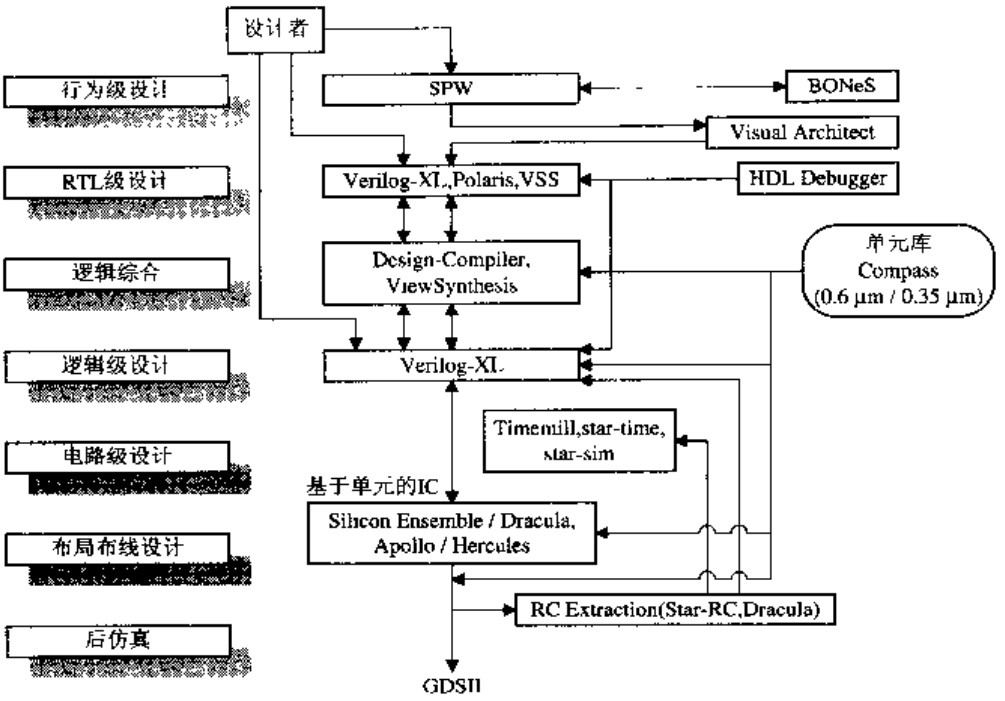


图 1.2　基于标准单元的 IC 设计流程及相关工具

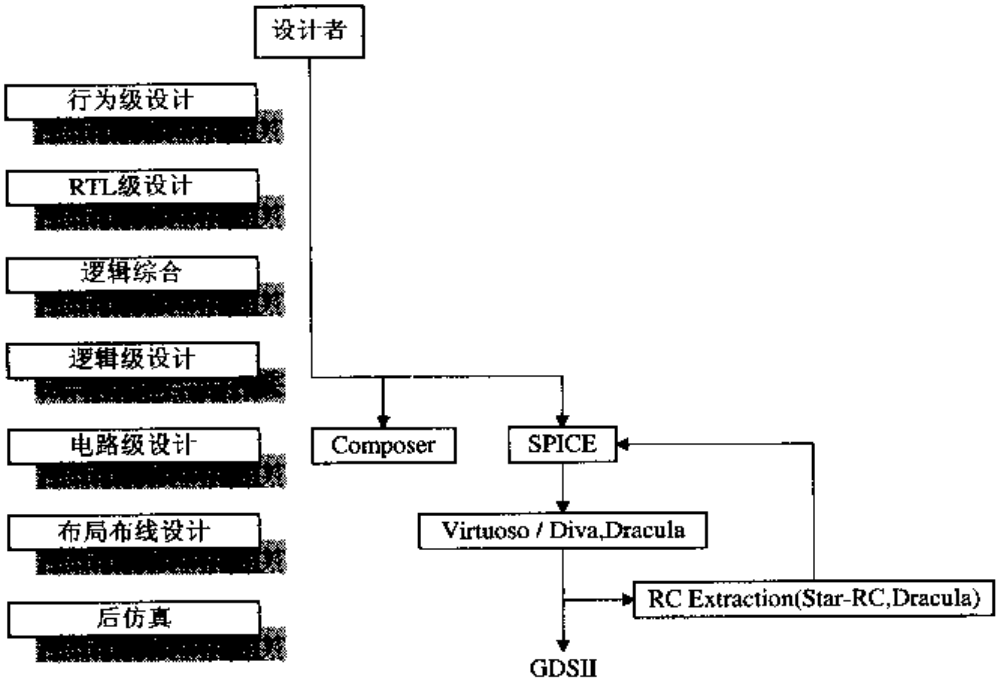


图 1.3　基于全定制 IC 设计流程及相关工具

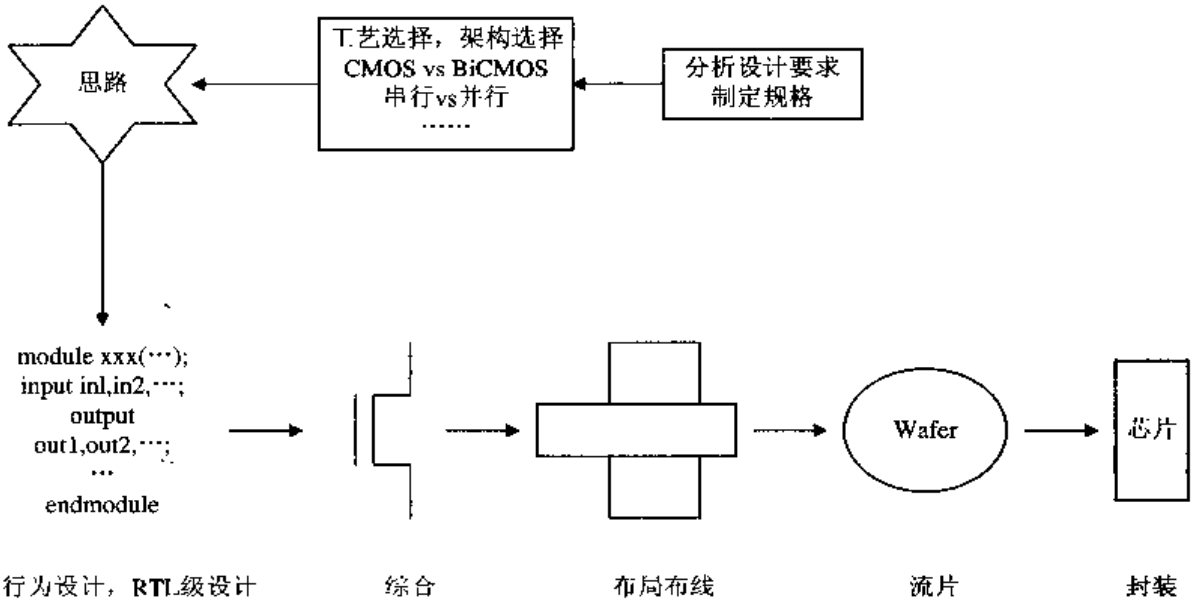


图 1.4　系统开发流程

根据笔者的看法，要成为一个好的 IC 设计师，应具备的条件有：

- 对电路有较好的掌握，并至少熟练掌握一门设计语言(Verilog 或 VHDL)。
- 熟悉工具的使用(仿真工具，综合工具等)。
- 对所做的应用有透彻理解。例如，做蓝牙基带芯片，就要了解蓝牙标准；做非接触 IC 卡，就要了解 14443 标准；做数字电视时，就要了解 MPEG/DVB 标准。

本章对上述三个方面进行了阐述，以后的章节也将围绕这三个方面展开。

笔者推荐使用以下 IC 设计开发工具：

编辑工具：ultraedit32。activeHDL 也不错。

仿真工具：modelsim。在工作站上，则是 Ncsim 最受欢迎。

综合工具：面向 ASIC(专用集成电路)的有 Synopsys 公司的 DC Compiler(已有 NT 版本)；面向 FPGA(现场可编程门阵列)的综合工具有 FPGA Express(Synopsys 已专注于 FPGA Compiler)和 Synplicity 公司的 Synplify。

FPGA 开发工具：Xilinx 公司的产品选用 ISE，Altera 公司的产品选用 Quartus 或 MaxPlus II。

1.2　ASIC 设计流程

ASIC 设计流程如下：

规格定义：规定所需的时钟频率、输入输出时序、功能(输入是什么，要求的输出是什么等等)、面积、功率、信噪比等等。

IC 设计的第一步，是给出设计规格，包括电气规格、功能规格等。电气规格是环境所能容忍的电压范围、直流特性、交流特性等；功能规格是指电路要实现的功能，可以用真值表、状态图、时序图、流程图等表示。此外，还要考虑静电防护(ESD)、I/O 电容、可测

性、封装形式等。

工艺选择：选择哪种工艺(例如，是 CMOS 还是 GaAs，是 0.6 微米还是 0.35 微米)。

架构选择：是并行的还是串行的架构，是否需要流水化结构。

架构要基于设计规格来选择。根据功能要求，设计者要考虑：采用并行设计还是串行设计？设计中各模块都用什么时钟？各时钟的相位有什么关系？是否用流水线？传输信号时是采用差分信号还是单一端口？用什么方式进行补偿？是采用动态逻辑还是静态逻辑？等等。

电路设计：划分模块，定义各模块的功能并定义模块间的连接关系。电路设计一般要从行为级开始，在 RTL 级完成设计。经过综合后，得到门级的网表。

电路仿真：包括功能仿真及时序仿真。在 RTL 级可以进行功能仿真，而时序仿真要在门级才能进行。

布局布线：一般是自动完成的。必要时要进行手工布线。

布局的验证：包括设计规则检查(DRC)等。

布局后的仿真：通常称为后仿真。这个时候可以得到布线的延时，因而时序仿真更接近真实的情况。

可靠性分析：考虑电子迁移/ESD 等。

为了清楚起见，在下面的介绍中，我们将电路仿真、布局布线及布局后的仿真都归到电路设计的范畴。

1.2.1 规格定义

规格定义给出了我们的设计目标，是 IC 设计的第一步。它不仅是设计的依据，也是验证工作的依据。规格定义必须完整、清晰。

1. 规格描述

在具体工程中，我们要写的规格很复杂。下面是工程中要定义的系统规格的框架：

- 系统整体描述
- 管脚封装图
- 架构
- 寄存器说明
- 系统功能描述(模块级)
- 直流特性(DC)
- 交流特性(AC)

我们将在后面的章节给出具体的规格描述示范。这里对 DC 特性及 AC 特性进行一些说明。

2. DC 特性(直流特性)

表 1.1 是 DC 特性描述的一个示例。其中，V_{CC}=5 V，温度范围为-10℃到 70℃。

表 1.1 DC 特 性

符号	含义	最小值	典型值	最大值	单位	说明
V_{cc}	电源电压	4.5		5.5	V	
V_{IL1}	低电平输入电压			1	V	clk, nwr,nrd
V_{IH1}	高电平输入电压	3.7			V	clk, nwr,nrd
V_{IL2}	Input Low Voltage			1		其它的输入引脚
V_{IH2}	Input High Voltage	3.5			V	其它的输入引脚
V_{OL1}	低电平输出电压			0.45	V	out1,out2 I_o= 10 mA
V_{OH1}	高电平输出电压	4			V	out1,out2 I_o= -10 mA
V_{OL2}	低电平输出电压			0.45	V	out3,out4 I_o= -1.5 mA
V_{OH2}	高电平输入电压	4			V	out3,out4 I_o= -1.5 mA
R_{pull}	上拉电阻	20k		80k	Ω	cs, nrst
I_{IL}	低电平输入泄漏电流			1	μA	V_{IN} = 0.4 V
I_{IH}	高电平输入泄漏电流			1	μA	V_{IN} = Vcc-0.4 V

3. AC 特性(交流特性)

表 1.2 是 AC 特性描述的一个示例，各符号的含义如图 1.5 所示。

表 1.2 AC 特 性

符号	最小值	最大值	单位
t_{LHLL}	20		ns
t_{CHWL}	30		ns
t_{AVLL}	8		ns
t_{LLAX}	8		ns
t_{LLWL}	15		ns
t_{RLDV}		100	ns
t_{RHDX}	0		ns
t_{RHDZ}		10	ns
t_{DVWH}	10		ns
t_{WHDX}	5		ns
t_{WLWH}	200		ns
t_{WHWL}	1000		ns
t_{AVWL}	30		ns
t_{WHAX}	5		ns
t_{RHAX}	5		ns

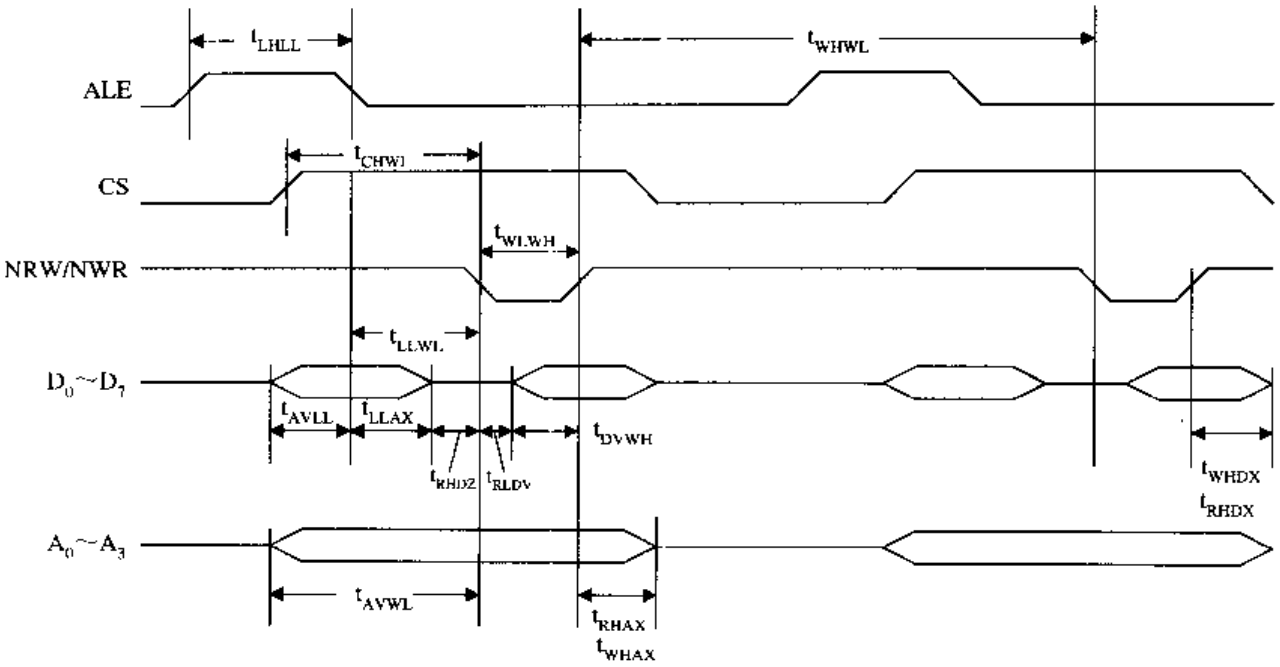


图 1.5　AC 特性

1.2.2　工艺选择

图 1.6 示出了硅工艺的类别。

表 1.3 给出了不同工艺间的差别。

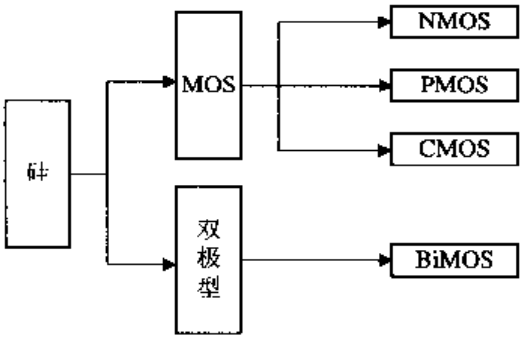


图 1.6　硅工艺类别

表 1.3　不同工艺的选择

	MOS	双极型	GaAs
工作频率	200 MHz	900 MHz	1.8 GHz
驱动能力	低	高	
工作电压	低电压/低阈值 SOI		高电压
类型	数字	模拟	混合
		高精度电容	
设计风格	有可用的单元库		

1.2.3　架构选择

依据系统规范，需要决定设计采用并行还是串行结构，是否需要加流水线等。串/并行结构的特点很好理解：串行结构速度较慢，但面积小；并行结构与之相反。流水线就是在

系统的关键路径上插入寄存器，这样可以使最大时钟频率提高。

1. 并行/串行结构

例如，对一个随机数发生器，我们采用串行设计时，可以采用如图 1.7 所示的结构。

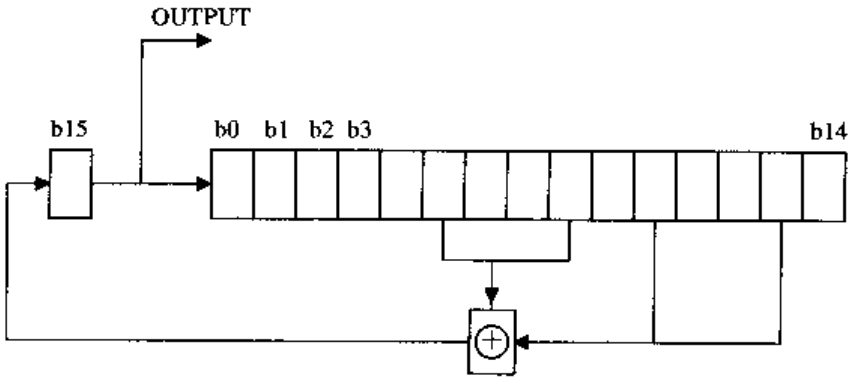


图 1.7　串行的随机数发生器

也可以采用并行的结构来实现随机数发生器。例如，构造一次生成一字节的随机数发生器时，用 8 个移位寄存器，按照与图 1.7 相似的结构并列起来就可以了。并行的随机数发生器的结构图如图 1.8 所示。

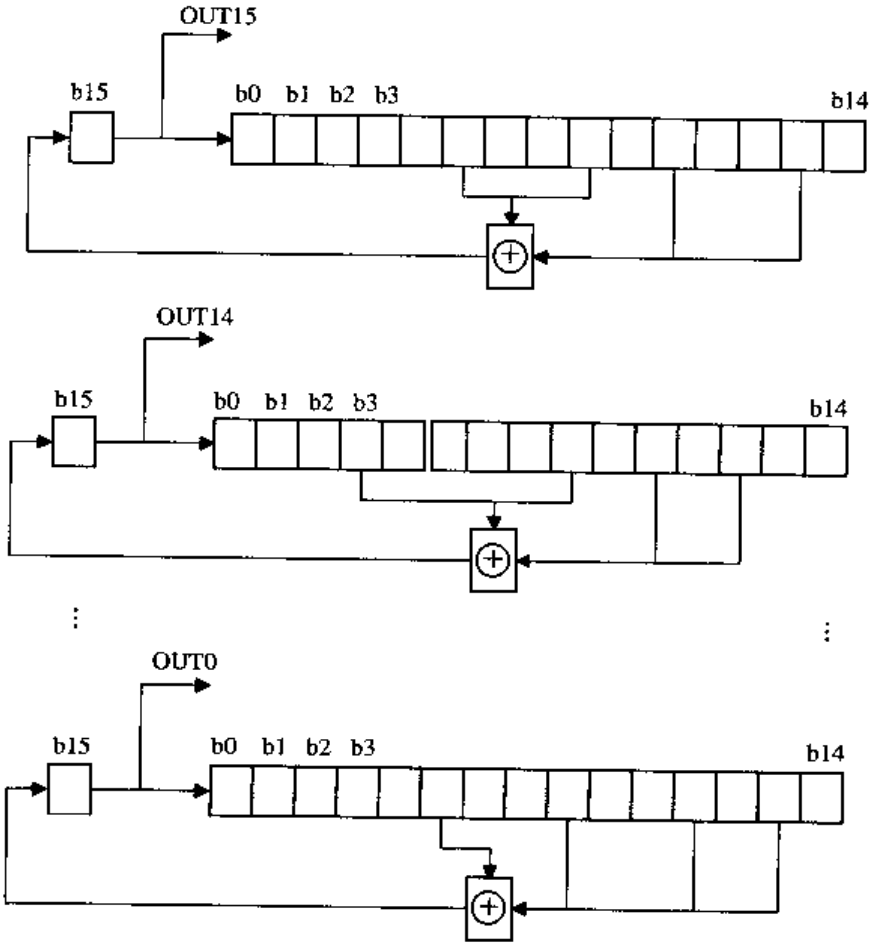
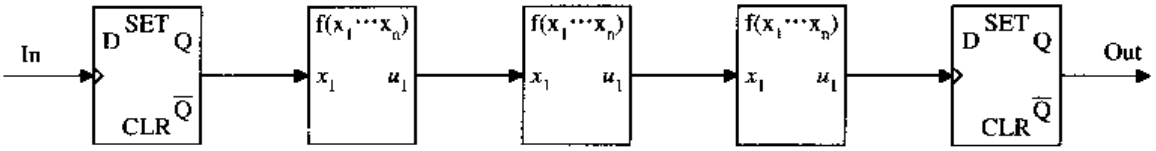


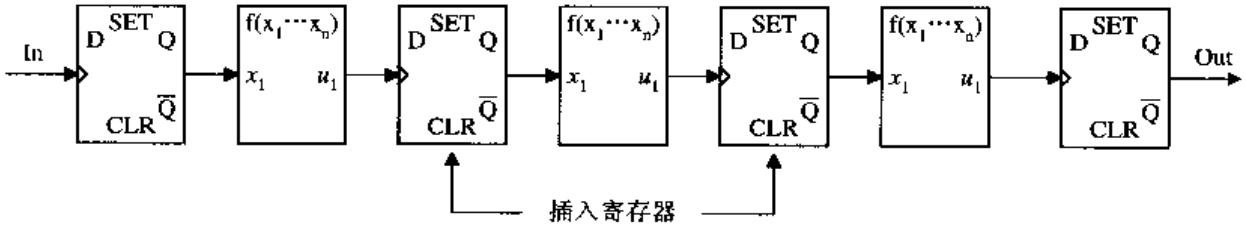
图 1.8　并行随机数发生器

2. 流水线结构(pipeline)

流水线是一种通过增加面积来提高速度的通用的设计技术，在电路的关键路径中插入寄存器，可以提高系统运行速度。图 1.9 的例子说明了此方法的用法。



(a)



(b)

图 1.9　流水线结构

(a) 无流水线；(b) 流水线

对于图 1.9(a)中的结构，有

$$T_{min}=t_{p,reg}+t_{p1,logic}+t_{p2,logic}+t_{p3,logic}+t_{setup,reg}$$

$t_{p,reg}$ 和 $t_{setup,reg}$ 是寄存器的传输延时和建立时间。$t_{p,logic}$ 代表通过组合网络的最坏情况延时路径的延时。通常这个延时要比寄存器的延时大许多，对电路的性能起主要影响。在图 1.9(a)中，三个组合逻辑的延时之和决定了系统时钟的速度。

流水线是一种打破性能瓶颈的方法。假定我们在三个组合逻辑间引入寄存器，如图 1.9(b)所示，则有：

$$T_{minpipe}=t_{p,reg}+\max(t_{p1,logic},t_{p2,logic},t_{p3,logic})+t_{setup,reg}$$

假定所有的逻辑模块的传输延时大致相同，并且锁存器的延时与组合逻辑延时相比可以忽略，则 $T_{minpipe}=T_{min}/3$。流水线网络的性能是原始电路性能的 3 倍。

付出的代价是增加两个额外的寄存器，并使输出延迟了两个时钟。这种代价是很小的。所以流水线在高性能数据通路中应用很多。但要注意，只有在关键路径上增加寄存器才有意义。如果寄存器的延时与组合逻辑的延时相当，增加额外的寄存器只会增加硬件过载，并不会获得性能的改善。

在流水线结构中，可以按流水方式工作，即将一个计算任务细分成若干个子任务，每个子任务由专门的部件处理，多个计算任务依次进行并行处理。该方法可以大大提高指令的执行速度。

图 1.10 是 ARM 结构的流水线的一个例子。

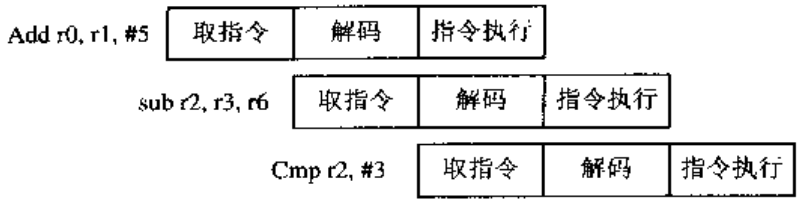


图 1.10 ARM 中的流水线

在这个例子中，指令的总运行时间比非流水线结构要少。例如，对于加、减、比较等指令，假设它们的执行时间都需要三个基本时间单位。采用非流水线方式，执行这三条指令需要 9 个基本时间单位；而采用三级流水线，只需要 5 个基本时间单位。

在实际的 CPU 设计中，由于涉及到条件转移等指令，因而流水线的控制非常复杂。

3. 时钟分配策略

在以往，时钟分配是在版图级设计的 Floorplanning 中实现的。但许多复杂设计，特别是针对深亚微米工艺的设计，要求在架构选择阶段就应该考虑这个问题，以更好地控制电路的性能，减少设计的迭代次数。设计者必须要有时钟分配这方面的知识。

设计者要完成的设计绝大多数是时序电路，图 1.11 就是时序电路的一个典型结构。时序电路可以分成两部分：一部分是组合逻辑，对数据进行运算；一部分是存储部件，通常是寄存器，用于数据的存储。在一定的控制下，数据沿着这样的通路一级级地传递下去。

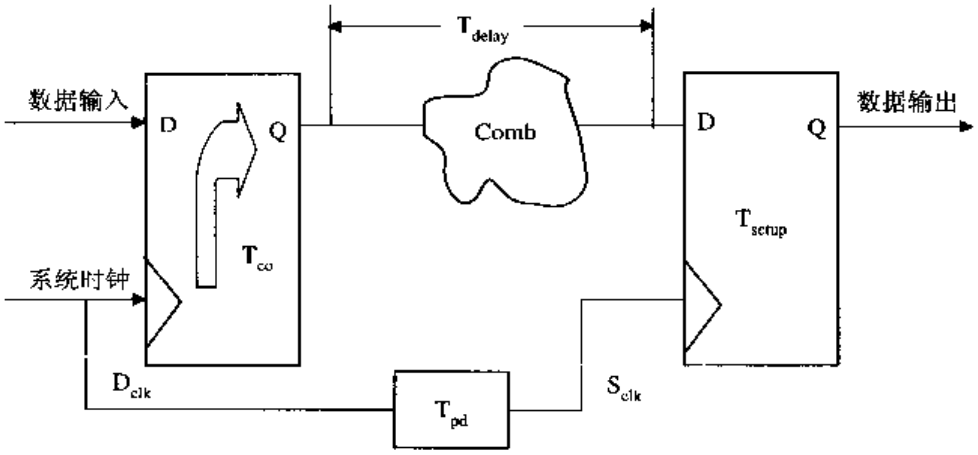


图 1.11 同步时序电路的基本结构

所有的时序电路都有一个共同点：必须对开关事件的顺序进行严格的定义。否则，可能会有错误的数据写入存储单元。大多数时序电路是同步系统，这意味着将数据锁存进存储器的动作是由一系列全局分布的时钟信号来控制的。在整个芯片上时钟信号的产生及它们对存储单元的分配，必须遵守严格的规则；否则，电路会发生错误。

完整的时钟分配策略包括：制定整体的时钟分配计划；定义最小的延时要求；设计标准的时钟缓冲(以提高时钟线的驱动能力)；考虑如何消除时钟偏移带来的影响；如果系统要求尽量省电，可能还需要使系统在某些时候能够禁止时钟。

(1) 整体的时钟分配计划：决定采用主时钟还是时钟树；时钟网络采用几级结构；各个模块的时钟频率是多少等。

(2) 定义最小的延时要求：根据设计规格定义的系统运行速度来定义最小的延时要求。

(3) 时钟缓冲：当负载超过时钟的驱动能力时，要考虑加时钟缓冲，提高驱动能力。

缓冲是由反相器构成的。驱动较大的电容时，用单一反相器构成的缓冲经常是不能满足要求的，这时候需要用 N 个反相器构成的缓冲链。如图 1.12 所示，缓冲的尺寸应该逐渐递增(增大倍数跟工艺有关，一般是 2～5)，这样才能得到最好的性能。

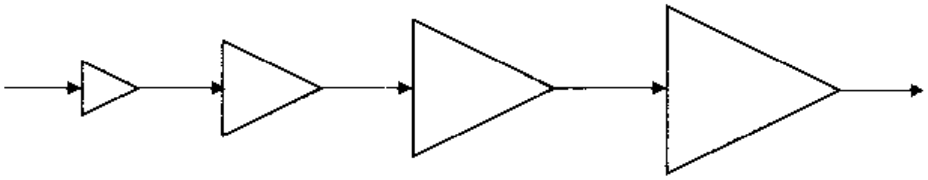


图 1.12　缓冲器链

(4) 消除时钟偏移。具体内容见第 2 章。

(5) 省电模式。省电模式跟具体的应用有关。有关内容这里略去。

1.2.4　电路设计

电路设计包括行为级设计、RTL 级设计、门级设计、晶体管级设计、版图级设计。电路设计即构造行为模型、RTL 模型、门级模型、开关级模型和几何模型。

行为模型：用高级语言结构实现设计的算法。

RTL 模型：描述在寄存器间数据的流动，以及数据是如何处理的。

门级模型：逻辑门的构造以及门的互连。

开关级模型：描述在器件中的晶体管和存储节点，以及它们的互连关系。

几何模型：描述版图实现中不同的结构层。

下面以 FIFO(先进先出栈)为例，说明电路设计的过程。在进行电路设计前首先要完成下面三步工作(该例用 Verilog 来实现，读者若对 Verilog 不太熟悉，可先阅读本书 1.5 小节，其中给出了有关 Verilog 的介绍)：

第一步：明确设计需求。这个 FIFO 是同步的还是异步的；是同步复位还是异步复位；FIFO 的宽度是多少；深度是多少；输入输出是什么等问题。

假设我们要设计的是一个同步 FIFO，异步复位；要求可容纳 15 个字节，宽度是 8 位；要求输出两个状态信号：full 与 empty，以供后继电路使用。

第二步：根据系统要求，画出系统的框图，如图 1.13 所示。

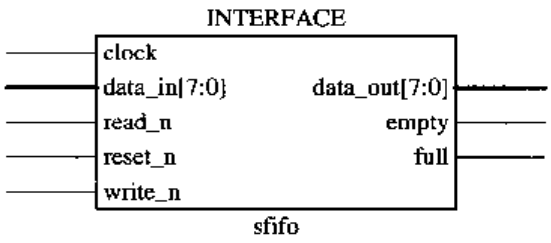


图 1.13　一个同步 FIFO 的框图

第三步：尽量详细地给出这个模块的功能及时序。因为所举的例子非常简单，这里只给出文字说明。

复位信号低电平有效；

在时钟的正边沿进行采样；

读写控制信号都是低电平有效；

满的情况下，不能再写，写指针不能加 1；

空的情况下，不能再读，读指针不能再加 1；

在上面三步准备工作完成之后，就可以开始行为级的设计，并对设计结果进行仿真。

1．行为级设计

规模较小的设计可以直接从 RTL(寄存器传输)级设计开始。但对较大规模的设计，例如设计一个嵌入式微处理器时，一定要先从行为级设计开始，以规范好系统的框架及算法。行为级规范并不直接映射为具体实现，仅仅模拟了设计的输入输出功能。例如，一个用来验证指令集合正确性与完整性的微处理器的行为模型是在每个指令基上执行的。

在行为级设计中使用最普遍的抽象语言是 VHDL 和 Verilog 语言，较著名的开发工具是 Cadence 公司的 SPW。许多设计者倾向于使用传统的编程语言，例如用 C 或 C++来构造行为模型（现在已经出现了用于仿真硬件的 C 语言），也有的设计者喜欢用 MATLAB。

我们先以一个 D 触发器为例，来说明行为级代码的写法，如下所示：

```
module dff(q,d,clear,preset,clock)
output q;
input d,clear,preset,clock;
reg q;
always @(clear or preset)
if(!clear)
 assign q=0;
else if(~preset)
assign q=1;
else
deassign q;
always @(posedge clock)
q<=d;
endmodule
```

同步 FIFO 的行为级代码如下所示：

首先定义模块端口，并给出信号的类型。为了使仿真接近真实情形，我们定义了从时钟到输出的延时。

```
`define DEL            //时钟到输出的延时。如果不定义，仿真时可能会引起混乱
module Sfifo(clock,reset_n,data_in,read_n,write_n,data_out,full, empty);

// 输入信号
input                  clock;        //输入时钟
input                  reset_n;      //复位信号，低电平有效
input [7:0]            data_in;      //输入的数据
input                  read_n;       //读控制信号，低电平有效
```

```
input                   write_n;        //写控制信号，低电平有效

//输出信号
output [7:0]            data_out;       // FIFO 输出数据
output                  full;           // FIFO 满标志信号
output                  empty;          // FIFO 空标志信号

// INOUTS

//信号声明
wire                    clock;
wire                    reset_n;
wire [7:0]              data_in;
wire                    read_n;
wire                    write_n;
reg [7:0]               data_out;
wire                    full;
wire                    empty;
```

接下来，定义需要的一些信号。首先，需要用一个数组来表示 FIFO 的存储体。其次，还需要读指针与写指针，用来指示读操作与写操作的位置。为了表示 FIFO 满还是空，还需要一个计数器，用以标志 FIFO 已使用了多少。

```
reg [7:0]   fifo_mem[14:0];     // FIFO 存储体，用数组表示
reg [3:0]   counter;            // 计数器，表示 FIFO 中已用了多少
reg [3:0]   rd_pointer;         // FIFO 读指针，指向下次读操作的地址
reg [3:0]   wr_pointer;         // FIFO 写指针，指向下次写操作的地址

// PARAMETERS

// 赋值声明，给出满标志与空标志的实现
assign #`DEL full = (counter == `15) ? 1'b1 : 1'b0;
assign #`DEL empty = (counter == 0) ? 1'b1 : 1'b0;

// MAIN CODE

// Look at the edges of reset_n
always @(reset_n) begin
    if (~reset_n) begin
        // Reset the FIFO pointer
        #`DEL;
```

```
            assign rd_pointer = `4'b0;
            assign wr_pointer = `4'b0;
            assign counter = `4'b0;
        end
        else begin
            # DEL;
            deassign rd_pointer;
            deassign wr_pointer;
            deassign counter;
        end
    end
```

最后，完成计数、读写的功能，并且在读写时，实现读写指针的递增，并监视是否有错。

```
    always @(posedge clock) begin
        if (~read_n) begin
            // FIFO 为空时，不能再读出
            if (counter == 0) begin
                $display("\nERROR at time %0t:", $time);
                $display("FIFO Underflow\n");

                // Use $stop for debugging
                $stop;
            end

            // 读操作时，计数器递减
            if (write_n) begin
                // Decrement the FIFO counter
                counter <= # DEL counter - 1;
            end

            //实现读操作
            data_out <= # DEL fifo_mem[rd_pointer];

            //对读指针递增。如果读指针已指向最后，则使它回到起始的位置
            if (rd_pointer == 14)
                rd_pointer <= # DEL `4'b0;
            else
                rd_pointer <= # DEL rd_pointer + 1;
            end
```

```
        if (~write_n) begin
            //检查 FIFO 是否溢出
            if (counter >= `15) begin
                $display("\nERROR at time %0t:", $time);
                $display("FIFO Overflow\n");

                // Use $stop for debugging
                $stop;
            end

            //当进行写操作时(读无效)，计数器加 1
            if (read_n) begin
                // Increment the FIFO counter
                counter <= #`DEL counter + 1;
            end

            //实现写操作
            fifo_mem[wr_pointer] <= #`DEL data_in;

            //写指针递增。如果写指针已指向最后，则让它回到起始位置
            if (wr_pointer == 14)
                wr_pointer <= #`DEL `4'b0;
            else
                wr_pointer <= #`DEL wr_pointer + 1;
        end
    end
    endmodule
```

2. **电路仿真**

在设计完成后，要进行仿真。电路设计的各个阶段都与仿真紧密联系在一起：行为级的仿真用于验证电路框架及算法的正确性；RTL 级的仿真可以验证功能正确性，一般称为**前仿真**；综合出门级网表之后，网表中加入了门的延时信息，仿真结果更接近于真实情况，此时进行的仿真称为**门级仿真**；布局布线之后，可以将连线延时加入到网表中，此时可以验证电路的时序是否正确，这个阶段的仿真一般称为**后仿真**。

仿真工具主要有 Synopsys 公司的 NC Simulator，Mentor 公司的 ModelSim。

实际上，对门级仿真与后仿真的概念有多种解释，不过这并不重要。

进行仿真时需要编写测试程序 testbench。编写 testbench 需要很高的技巧。输入激励应该能够完整地测试出设计的功能。在这个例子中，要测试当读写速度不一致时的情形，以证明 FIFO 确实能起到数据缓冲的作用。

要仿真写速度大于读速度的情形，以及读速度大于写速度的情形，可采用以下方法。首先，给出 testbench 的信号描述。

```
// DEFINES
`define DEL 1                    //时钟到输出的延时

module sfifo_sim();
// INPUTS
// OUTPUTS
// INOUTS
// 信号声明，这些信号与待测试模块中的端口信号一一对应
reg             clock;
reg             reset_n;
reg [7:0]       in_data;
reg             read_n;
reg             write_n;
wire [7:0]      out_data;
wire            full;
wire            empty;

integer         fifo_count;          //记录 FIFO 中的字节数
reg [7:0]       exp_data;            //期望从 FIFO 输出的数据
reg             fast_read;           //标志以高速度进行读操作
reg             fast_write;          //标志以高速度进行写操作
reg             filled_flag;         //标志 FIFO 已填满
reg             cycle_count;         //周期计数，用来生成读写控制信号

//赋值语句
//对 FIFO 进行实例化
Sfifo sfifo(
            .clock(clock),
            .reset_n(reset_n),
            .data_in(in_data),
            .read_n(read_n),
            .write_n(write_n),
            .data_out(out_data),
            .full(full),
            .empty(empty));
//对输入信号进行初始化，复位 FIFO，并设置写速度大于读速度
```

```
initial begin
        in_data = 0;
        exp_data = 0;
        fifo_count = 0;
        read_n = 1;
        write_n = 1;
        filled_flag = 0;
        cycle_count = 0;
        clock = 1;

        //写速度大于读速度
        fast_write = 1;
        fast_read = 0;

        //复位
        reset_n = 1;
        #20   reset_n = 0;
        #20 reset_n = 1;

        //初始情况下，FIFO 应该为空
        if (empty !== 1) begin
            $display("\nERROR at time %0t:", $time);
            $display("After reset, empty status not asserted\n");

            $stop;
        end
        if (full !== 0) begin
            $display("\nERROR at time %0t:", $time);
            $display("After reset, full status is asserted\n");

            $stop;
        end
end

//生成时钟信号
always #100 clock = ~clock;

//对 FIFO 中的字节数进行计数。每次写操作时，计数加 1；读操作时，计数减 1
```

```
always @(posedge clock) begin
        if (~write_n && read_n)
            fifo_count = fifo_count + 1;
        else if (~read_n && write_n)
            fifo_count = fifo_count   1;
end
//检查输出数据是否是期望的数据。如果不是，则报错
always @(negedge clock) begin
        if (~read_n && (out_data !== exp_data)) begin
            $display("\nERROR at time %0t:", $time);
            $display("      Expected data out = %h", exp_data);
            $display("      Actual data out    = %h\n", out_data);

            $stop;
        end
```

接下来，生成读/写控制信号和输入信号。当 FIFO 满时，写控制信号无效。

需要注意的是，读/写控制信号与输入信号都是在时钟的下降沿生成的。而对它们进行采样是在时钟的上升沿(如 sfifo 模块所示)完成的，这样可以保证建立时间与保持时间(建立时间和保持时间的概念在第 2 章中进行详细介绍)。数据的产生与采样不能用同样的时钟沿，以防止锁错数据，这是电路设计的一个普遍的准则。

```
if ((fast_write || (cycle_count & 1'b1)) &&~full)
    begin
      write_n = 0;
      //生成输入数据
      // Set up the data for the next write
      in_data = in_data + 1;
    end
else
      write_n = 1;

//得到读控制信号，并给出期望的数据
if ((fast_read || (cycle_count & 1'b1)) &&
            ~empty) begin
      read_n = 0;

      exp_data = exp_data + 1;
end
else
      read_n = 1;
```

如果 FIFO 已经写满，则使读速度大于写速度，如下所示：

```
if (full) begin
        fast_read = 1;
        fast_write = 0;
  //设置写满标志
        filled_flag = 1;
end

//如果已清空 FIFO，则结束
if (filled_flag && empty) begin
        $display("\nSimulation complete - no errors\n");
        $finish;
end

//对计数周期进行计数
cycle_count = cycle_count + 1;
end

// Check all of the status signals with each change of fifo_count
always @(fifo_count) begin
        // Wait a moment to evaluate everything
        #DEL;
        #DEL
        #DEL;

        case (fifo_count)
                0: begin
                        if ((empty !== 1) ||(full !== 0))
                                begin
                                $display("\nERROR at time %0t:", $time),
                                $display("       fifo_count = %h", fifo_count);
                                $display("       empty = %b", empty);
                                $display("       full   = %b\n", full);

                                // Use $stop for debugging
                                $stop;
                        end

                        if (filled_flag === 1) begin
```

```
                    // The FIFO has filled and emptied
                    $display("\nSimulation complete - no errors\n");
                    $finish;
                end
            end
            end
            15: begin
                if ((empty !== 0) ||(full !== 1))
                    begin
                    $display("\nERROR at time %0t:", $time);
                    $display("      fifo_count = %h", fifo_count);
                    $display("      empty = %b", empty);
                    $display("      full   = %b\n", full);

                    $stop;
                end

                //如果 FIFO 写满，则给出写满标志。这时候，应该使读速度大于写速度
                filled_flag = 1;

                fast_write = 0;
                fast_read = 1;
            end

            default: begin
                if ((empty !== 0) || (full !== 0)) begin
                    $display("\nERROR at time %0t:", $time);
                    $display("      fifo_count = %h", fifo_count);
                    $display("      empty = %b", empty);
                    $display("      full   = %b\n", full);

                    // Use $stop for debugging
                    $stop;
                end

            end
        endcase
end
endmodule
```

3. RTL 级设计

在这个阶段，可以进行“可综合的设计”。这个阶段的设计对芯片性能有着直接影响，也最能体现设计者的技巧。在后面的章节中，我们会对 RTL 级设计进行着重论述。

可用于 RTL 级设计的开发工具很多。综合工具有 Synopsys 公司的 DC Compiler。

下面仍以同步 FIFO 为例，对 RTL 级设计进行说明。

```
`define DEL 1                    // 时钟到输出的延时
module Sfifo(clock,reset_n,data_in,read_n,write_n,data_out,full,empty);
//输入
input                            clock;          //系统时钟
input                            reset_n;        //复位信号，低电平有效
input [`FIFO_WIDTH-1:0]          data_in;        // FIFO 的输入数据
input                            read_n;         //读使能信号，低电平有效
input                            write_n;        //写使能信号，低电平有效

//输出
output [`FIFO_WIDTH-1:0]         data_out;       // FIFO 输出数据
output                           full;           //状态信号，标志 FIFO 满
output                           empty;          //状态信号，标志 FIFO 空

// INOUTS

//端口信号声明
wire                             clock;
wire                             reset_n;
wire [`FIFO_WIDTH-1:0]           data_in;
wire                             read_n;
wire                             write_n;
reg  [`FIFO_WIDTH-1:0]           data_out;
wire                             full;
wire                             empty;
reg [15:0]   fifo_mem[7:0];      // FIFO 的存储体
reg [3:0]    counter;
 //计数器，标志 FIFO 已使用了多少，用于生成 full 与 empty 信号
reg [3:0]    rd_pointer;         // FIFO 读指针，用于表示下一个读操作的位置
reg [3:0]    wr_pointer;         // FIFO 写指针，用于表示下一个写操作的位置
```

下面给出 full 与 empty 信号的生成。

```
//赋值声明
assign #`DEL full = (counter == 15) ? 1'b1 : 1'b0;
assign #`DEL empty = (counter == 0) ? 1'b1 : 1'b0;
```

接下来，完成读指针、写指针和计数器的功能。

```
//本模块实现了读指针、写指针和计数器的功能
always @(posedge clock or negedge reset_n ) begin
    if (~reset_n) begin
        // Reset the FIFO pointer
        rd_pointer <= #DEL 4'b0;
        wr_pointer <= #DEL 4'b0;
        counter <= #DEL 4'b0;
    end
    else begin
        if (~read_n) begin
            //如果 FIFO 为空，不能再读，并报错
            if (counter == 0) begin
                $display("\nERROR at time %0t:", $time);
                $display("FIFO Underflow\n");

                //终止系统任务，用于调试
                $stop;
            end

            //读有效，写无效时，计数器减 1
            if (write_n) begin
                counter <= #DEL counter - 1;
            end

            //如果读指针已指到最后一个位置，则返回起始位置
            if (rd_pointer == 14)
                rd_pointer <= #DEL 4'b0;
            else
                rd_pointer <= #DEL rd_pointer + 1;
        end
        if (~write_n) begin
            //检查 FIFO 是否溢出
            if (counter >= 15) begin
                $display("\nERROR at time %0t:", $time);
                $display("FIFO Overflow\n");

                $stop;
            end
```

```
                //写有效，且读无效时，计数器加 1
                if (read_n) begin
                    // Increment the FIFO counter
                    counter <= #DEL counter + 1;
                end

                //写使能时，使写指针递增
                if (wr_pointer == 14)
                    wr_pointer <= #DEL 4'b0;
                else
                    wr_pointer <= #DEL wr_pointer + 1;
            end
        end
    end
```

其次，实现读/写功能。

```
    always @(posedge clock) begin
        if (~read_n) begin
            // Output the data
            data_out <= #DEL fifo_mem[rd_pointer];
        end
        if (~write_n) begin
            // Store the data
            fifo_mem[wr_pointer] <= #DEL data_in;
        end
    end
    endmodule
```

这个阶段的仿真与行为级的仿真类似，不再赘述。

4. 门级设计

在介绍门级设计过程之前，先解释几个与门级设计有关的基本概念。

(1) 扇出：扇出表示连接到一个驱动门的负载门的数目 N(如图 1.14 所示)。当扇出很大时，增加的负载会使驱动门的动态性能下降。因此，许多通用的组件和库组件都定义了最大扇出，以保证上单元的静态和动态性能满足要求。

(2) 扇入：一个门的扇入定义为门的输入的数目(如图 1.15 所示)。具有大扇入的门通常更复杂，经常导致较差的静态和动态特性。

(3) 综合：综合过程如图 1.16 所示。

用逻辑门来描述电路是最直观的，并且能直接对芯片的性能产生影响。

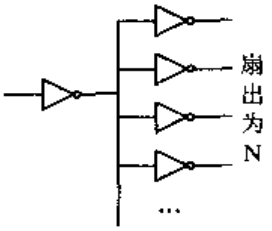


图 1.14 扇出

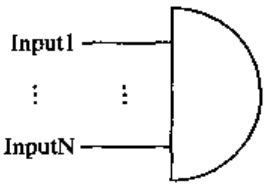


图 1.15 扇入

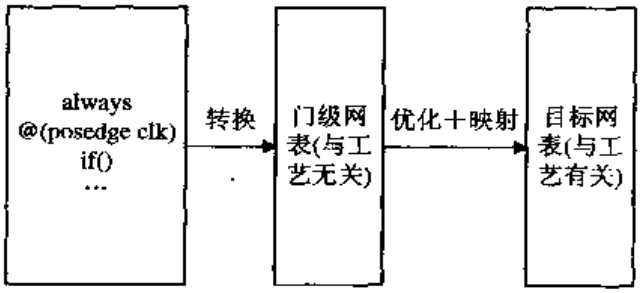


图 1.16 综合的概念

在不同的设计中，所关心的性能指标也不同。例如，在一般产品中，芯片面积是最受关心的问题，因为它直接决定了产品的成本。在许多高速应用中，特别是微处理器，速度是最关键的指标。在移动应用中，功耗非常重要。针对这些不同的要求，要采用不同的电路类型，甚至不同的制造工艺。只有熟悉不同门级结构的优缺点，才能设计出高性能的电路。

1) 静态互补 CMOS 门

一个静态的CMOS门，由一个上拉网络(PUN)和一个下拉网络(PDN)组成。PUN 由 PMOS 晶体管构成，与 V_{DD} 相接；PDN 将输入与 V_{SS} 相连，只包括 NMOS 器件。这类门结构具有以下特点：

- 噪声容限高，不易受干扰。
- 没有静态功耗，因为在稳定状态方式下，V_{DD} 和 V_{SS} 之间不存在直接通路。
- 上升时间与下降时间可比。
- 所用的晶体管较多，占用面积较大。扇入增加时，传输延时急剧增加，不适于实现具有大扇入值的门。这使得它不适用于实现高速复杂门。

2) 成比例逻辑

成比例逻辑门由一个 NMOS 下拉网络(用以实现逻辑功能)和一个简单的负载器件构成。负载可以是电阻，也可以是有源器件(例如晶体管，这时候称为伪 NMOS)。为了保证较高的噪声容限，应该使负载电阻比下拉网络的电阻大许多。

设计一个同等扇入的门，伪 NMOS 逻辑要比静态 CMOS 逻辑所用的晶体管少，速度更快，但功耗较大。在存储器的译码电路中，常用这种逻辑。因为在这种应用中，输出主要是高阻态，消耗的静态功耗是有限的。

3) 传输门逻辑

传输门如图 1.17 所示。

传输门可以看作是一个理想的开关。将两个开关串联起来可以实现与逻辑，将它们并联起来可以实现或逻辑。更复杂的门可以用这两种逻辑组合起来实现。传输门的结构比较简单，常用来实现快速的加法电路和寄存器。缺点是所用的晶体管数目较多。

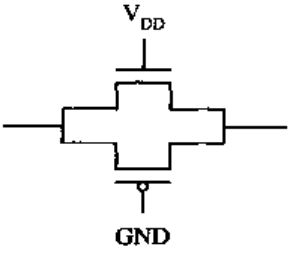


图 1.17 传输门

4) 动态逻辑

动态逻辑利用电容来存储数据。动态逻辑工作时，首先要进行预充。工作时，根据输入信号的不同，决定在输出端与地之间是否存在通路。如果存在通路，则电容放电，输出为 0。如果不存在通路，则输出为 1。一个例子(动态电路的基本结构)如图 1.18 所示。

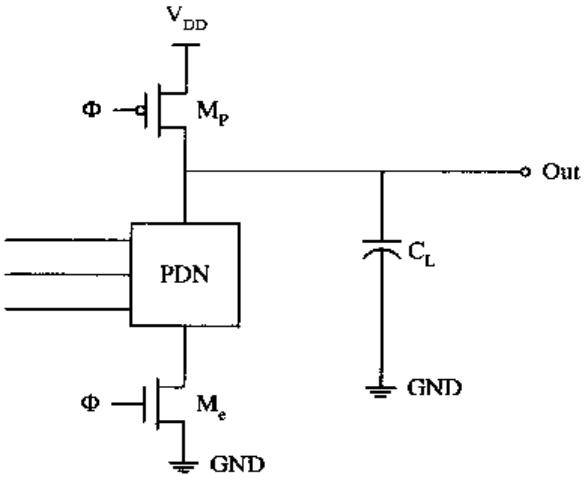


图 1.18　动态电路的基本结构

图 1.18 所示的电路工作时，首先进行预充。此时 Φ=0，输出节点 Out 通过 PMOS 晶体管 M_p 预充到 V_{DD}。预充时 NMOS 晶体管 M_e 断开，因而不管输入信号为何值，都没有直流电流通过。当 Φ=1 时，预充晶体管 M_p 断开，赋值晶体管 M_e 闭合。如果输入的信号及 PDN 都取适当值，且在 Out 和 GND 之间有一条通路，则 Out 放电，直至输出变低。如果不存在这样的通路，则预充的值将保存于输出电容 C_L 上，输出为 1。

表 1.4 给出了几种门级结构的比较，其中互补逻辑、伪 NMOS 逻辑和传输门逻辑都是静态逻辑。

表 1.4　门级结构的比较

类型	静态功耗	晶体管数目	传输延时
互补逻辑	无	8	较大
伪 NMOS 逻辑	有	5	最大
传输门逻辑	无	14	较大
动态(NP)逻辑	无	6	最小

静态逻辑的抗噪声性能较好。这使得设计过程很自由，自动化程度可以很高。其中，互补逻辑不适于设计大扇入的门；伪 NMOS 逻辑简单快速，但噪声容限低，并且具有静态功耗；传输门逻辑适用于乘法器和异或门占主导位置的逻辑(如加法器)。动态逻辑可以实现快速和复杂度低的门，它的缺点是，寄生效应(诸如电荷共享)使设计过程很困难；电荷泄漏要求对存储的电荷周期性地刷新，这使得电路的最低工作频率受到限制。

Verilog 提供了门级结构建模的功能，相应的说明见表 1.5 所示。

表 1.5　Verilog 中的门级模型

逻辑	真值表	Verilog 实现及图示
and	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 图 1.19　与门(1)	and a1(out,in1,in2) 符号表示为： In1 In2 Out 图 1.20　与门(2)

续表

<table>
<tr><th>逻辑</th><th>真值表</th><th>Verilog 实现及图示</th></tr>
<tr><td>nand</td><td>
<table>
<tr><th>nand</th><th>0</th><th>1</th><th>x</th><th>z</th></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>x</td><td>x</td></tr>
<tr><td>x</td><td>1</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>z</td><td>1</td><td>x</td><td>x</td><td>x</td></tr>
</table>
图 1.21　与非门(1)
</td><td>
nand a1(out,in1,in2)
符号表示：


图 1.22　与非门(2)
</td></tr>
<tr><td>or</td><td>
<table>
<tr><th>or</th><th>0</th><th>1</th><th>x</th><th>z</th></tr>
<tr><td>0</td><td>0</td><td>1</td><td>x</td><td>x</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>x</td><td>x</td><td>1</td><td>x</td><td>x</td></tr>
<tr><td>z</td><td>x</td><td>1</td><td>x</td><td>x</td></tr>
</table>
图 1.23　或门(1)
</td><td>
or a1(out,in1,in2)
符号表示为：


图 1.24　或门(2)
</td></tr>
<tr><td>nor</td><td>
<table>
<tr><th>nor</th><th>0</th><th>1</th><th>x</th><th>z</th></tr>
<tr><td>0</td><td>1</td><td>0</td><td>x</td><td>x</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>x</td><td>x</td><td>0</td><td>x</td><td>x</td></tr>
<tr><td>z</td><td>x</td><td>0</td><td>x</td><td>x</td></tr>
</table>
图 1.25　或非门(1)
</td><td>
nor a1(out,in1,in2)
符号表示为：


图 1.26　或非门(2)
</td></tr>
<tr><td>xor</td><td>
<table>
<tr><th>xor</th><th>0</th><th>1</th><th>x</th><th>z</th></tr>
<tr><td>0</td><td>0</td><td>1</td><td>x</td><td>x</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>x</td><td>x</td></tr>
<tr><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>z</td><td>x</td><td>x</td><td>x</td><td>x</td></tr>
</table>
图 1.27　异或门(1)
</td><td>
xor a1(out,in1,in2)
符号表示为：

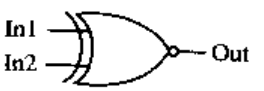

图 1.28　异或门(2)
</td></tr>
<tr><td>xnor</td><td>
<table>
<tr><th>xnor</th><th>0</th><th>1</th><th>x</th><th>z</th></tr>
<tr><td>0</td><td>1</td><td>0</td><td>x</td><td>x</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>x</td><td>x</td></tr>
<tr><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>z</td><td>x</td><td>x</td><td>x</td><td>x</td></tr>
</table>
图 1.29　异或非门(1)
</td><td>
xnor a1(out,in1,in2)
符号表示为：


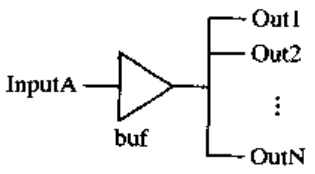
图 1.30　异或非门(2)
</td></tr>
<tr><td>buf</td><td>
<table>
<tr><th colspan="2">buf</th></tr>
<tr><th>Inputs</th><th>Outputs</th></tr>
<tr><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td></tr>
<tr><td>x</td><td>x</td></tr>
<tr><td>z</td><td>x</td></tr>
</table>
图 1.31　缓冲(1)
</td><td>
buf b1(out1,out2,…,outn,in1)
符号表示为：


图 1.32　缓冲(2)
</td></tr>
</table>

续表

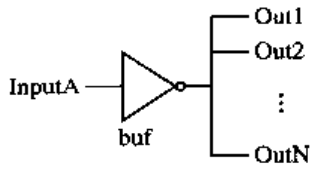
<table>
<tr><th>逻辑</th><th>真值表</th><th>Verilog 实现及图示</th></tr>
<tr><td>not</td><td>
<table>
<tr><th colspan="2">not</th></tr>
<tr><th>Inputs</th><th>Outputs</th></tr>
<tr><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td></tr>
<tr><td>x</td><td>x</td></tr>
<tr><td>z</td><td>x</td></tr>
</table>
图 1.33 非门(1)
</td><td>not b1(out1,out2,…outn,in1)
 

图 1.34 非门(2)</td></tr>
<tr><td>bufif0
具有三态输出的逻辑门</td><td>
<table>
<tr><th rowspan="2">bufif0</th><th colspan="4">CONTROL</th></tr>
<tr><th>0</th><th>1</th><th>x</th><th>z</th></tr>
<tr><td>D 0</td><td>0</td><td>z</td><td>L</td><td>L</td></tr>
<tr><td>A 1</td><td>1</td><td>z</td><td>H</td><td>H</td></tr>
<tr><td>T x</td><td>x</td><td>z</td><td>x</td><td>x</td></tr>
<tr><td>A z</td><td>x</td><td>z</td><td>x</td><td>x</td></tr>
</table>
图 1.35 三态逻辑门(1)
</td><td>bufif0 bf1 (outw,inw,controlw);
第一个端口是输出端口，第二个端口是数据输入，第三个端口是控制输入。根据控制输入，输出可被驱动到高阻状态，即值 z。对于 bufif0，若控制输入为 1，则输出为 z；否则数据被传输至输出端</td></tr>
<tr><td>bufif1
具有三态输出的逻辑门</td><td>
<table>
<tr><th rowspan="2">bufif1</th><th colspan="4">CONTROL</th></tr>
<tr><th>0</th><th>1</th><th>x</th><th>z</th></tr>
<tr><td>D 0</td><td>z</td><td>0</td><td>L</td><td>L</td></tr>
<tr><td>A 1</td><td>z</td><td>1</td><td>H</td><td>H</td></tr>
<tr><td>T x</td><td>z</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>A z</td><td>z</td><td>x</td><td>x</td><td>x</td></tr>
</table>
图 1.36 三态逻辑门(2)
</td><td>bufif1 bf1 (outw,inw,controlw);
对于 bufif1，若控制输入为 0，则输出为 z</td></tr>
<tr><td>notif0
具有三态输出的逻辑门</td><td>
<table>
<tr><th rowspan="2">notif0</th><th colspan="4">CONTROL</th></tr>
<tr><th>0</th><th>1</th><th>x</th><th>z</th></tr>
<tr><td>D 0</td><td>1</td><td>z</td><td>H</td><td>H</td></tr>
<tr><td>A 1</td><td>0</td><td>z</td><td>L</td><td>L</td></tr>
<tr><td>T x</td><td>x</td><td>z</td><td>x</td><td>x</td></tr>
<tr><td>A z</td><td>x</td><td>z</td><td>x</td><td>x</td></tr>
</table>
图 1.37 三态逻辑门(3)
</td><td>notif0 bf1 (outw,inw,controlw);
对于 notif0，如果控制输出为 1，那么输出为 z；否则输入数据值的非传输到输出端</td></tr>
<tr><td>notif1
具有三态输出的逻辑门。
如果控制端为 1，则起非门的作用</td><td>
<table>
<tr><th rowspan="2">notif1</th><th colspan="4">CONTROL</th></tr>
<tr><th>0</th><th>1</th><th>x</th><th>z</th></tr>
<tr><td>D 0</td><td>z</td><td>1</td><td>H</td><td>H</td></tr>
<tr><td>A 1</td><td>z</td><td>0</td><td>L</td><td>L</td></tr>
<tr><td>T x</td><td>z</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>A z</td><td>z</td><td>x</td><td>x</td><td>x</td></tr>
</table>
图 1.38 三态逻辑门(4)
</td><td>notif1 bf1 (outw,inw,controlw);
对于 notif1，若控制输入为 0，则输出为 z</td></tr>
</table>

下面给出一个门级实现的例子，这是一个 2-4 解码器(见图 1.39)。

实现的代码如下：

```
module dec2x4(A,B,Enable,Z);
input   A,B,Enable;
```

```
output[0:3] Z;
wire Abar,Bbar;

not #(1,2)
    V0 (Abar,A),
    V1(Bbar,B);
nand #(4,3)
    N0(Z[3],Enable,A,B),
    N1(Z[0],Enable,Abar,Bbar),
    N2(Z[1],Enable,Abar,B),
    N3(Z[2],Enable,A,Bbar);
endmodule
```

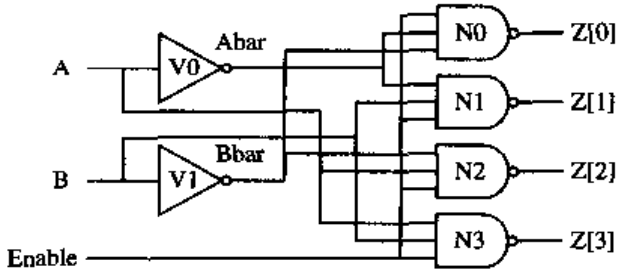


图 1.39　2-4 解码器

随着逻辑综合工具日益成熟，门级设计已不多见。

5. 晶体管级设计

硅半导体器件是用来构建电路的基本单元，包括二极管、MOS 管和双极型晶体管。

MOS 场效应晶体管是一个电压控制器件，控制栅端通过 SiO_2 电容与导电沟道隔离起来。MOS 晶体管的最吸引人的特点是它可以近似为一个电压控制开关：当控制电压为低时，开关是非导通的(开路)；当控制电压为高时，导电沟道形成，开关可以看作是闭合的，这种操作与二进制数字逻辑非常匹配，使得 MOS 管特别适合于数字电路。大多数数字集成电路是由静态 CMOS 构成的。静态 CMOS 器件的最大输出电压与最小输出电压之差等于供电电压，并且与晶体管的尺寸无关。

MOS 晶体管的动态操作主要受器件电容影响。器件电容主要是由栅电容和源极、漏极的耗尽区电容组成的。这些电容的最小化是高性能 MOS 设计的一个重要要求。

在设计全定制电路或者模拟电路时，要从晶体管级着手。缺乏对基本单元构造的深入理解，要利用这些模块进行复杂设计，是不可能成功的(特别是在深亚微米的设计领域中)。

晶体管设计的著名工具是 HSPICE 软件，在第 5 章里有对 HSPICE 软件的介绍。

在 Verilog 中，可以表示 MOS 晶体管、双向开关和上拉/下拉电阻，如表 1.6 所示。

表 1.6　Verilog 中的晶体管模型

晶体管类别	真值表	Verilog 实现和图示
nmos：N 型晶体管，源与漏极间的阻抗较小	图 1.40	nmos p1(outputA,inputB,ControlC); 这里分别用 outputA,inputB,ControlC 表示一个 MOS 管的漏极、源极和栅极 图 1.41　nmos 开关

nmos / rnmos	CONTROL 0	1	x	z
D 0	z	0	L	L
A 1	z	1	H	H
T x	z	x	x	x
A z	z	z	z	z

续表

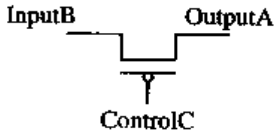
<table>
<tr><th>晶体管类别</th><th>真值表</th><th>Verilog 实现和图示</th></tr>
<tr><td>pmos：P 型晶体管，源与漏极间的阻抗较小</td><td>
<table>
<tr><td>pmos
rpmos</td><td colspan="4">CONTROL
0 1 x z</td></tr>
<tr><td>D 0</td><td>0 z L L</td></tr>
<tr><td>A 1</td><td>1 z H H</td></tr>
<tr><td>T x</td><td>x z x x</td></tr>
<tr><td>A z</td><td>z z z z</td></tr>
</table>
图 1.42
</td><td>pmos p1(outputA,inputB,controlC);
 

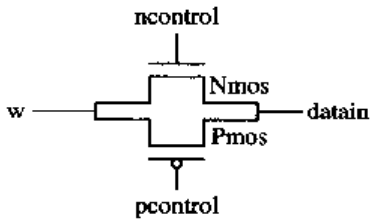
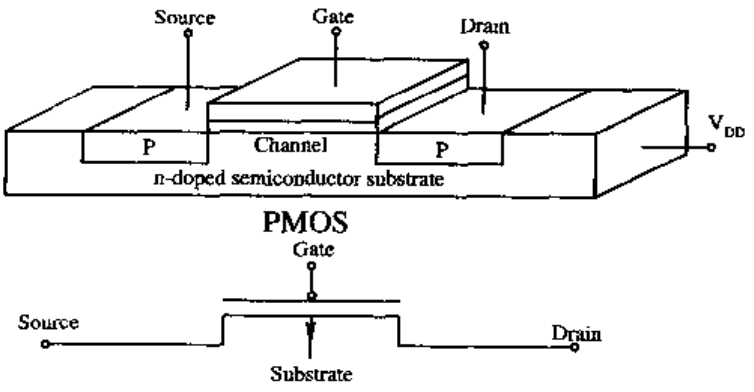
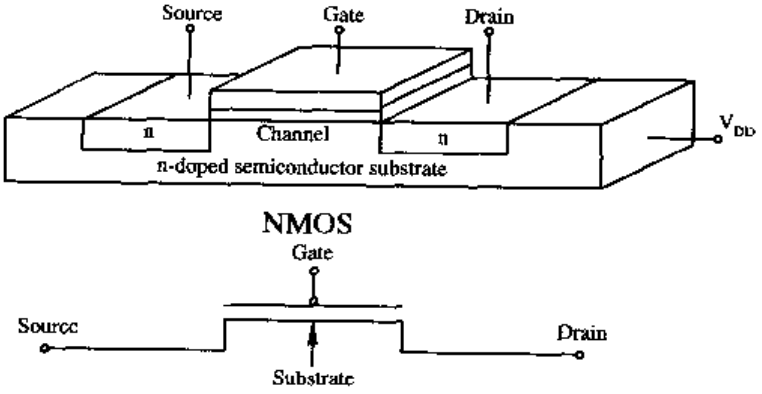
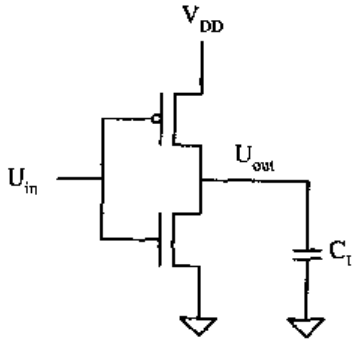
图 1.43 Pmos 开关</td></tr>
<tr><td>rnmos：NMOS 晶体管，源与漏极间的阻抗较大</td><td>同 nmos</td><td>rnmos p1(out,data,control);</td></tr>
<tr><td>rpmos：PMOS 晶体管，源与漏极间的阻抗较大</td><td>同 pmos</td><td>rpmos p1(out,data,control);</td></tr>
<tr><td>cmos：nmos 与 pmos 的组合</td><td colspan="2"> 

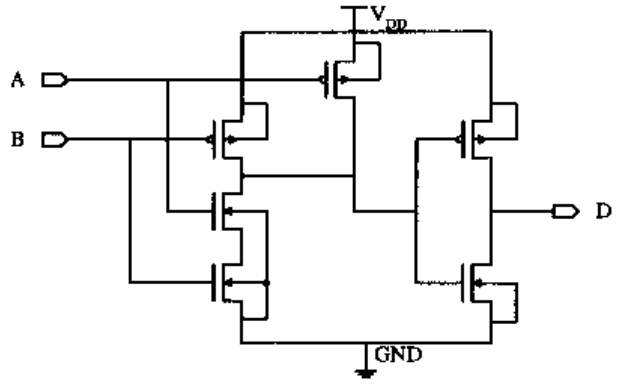
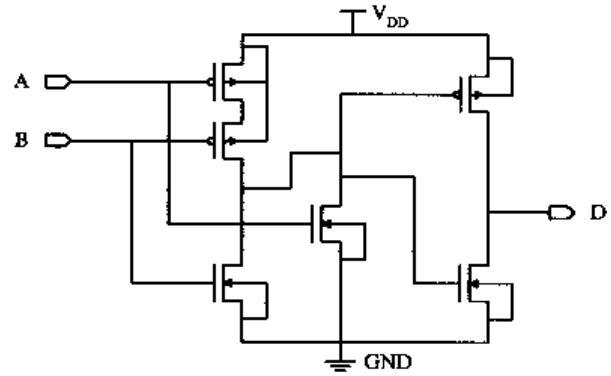
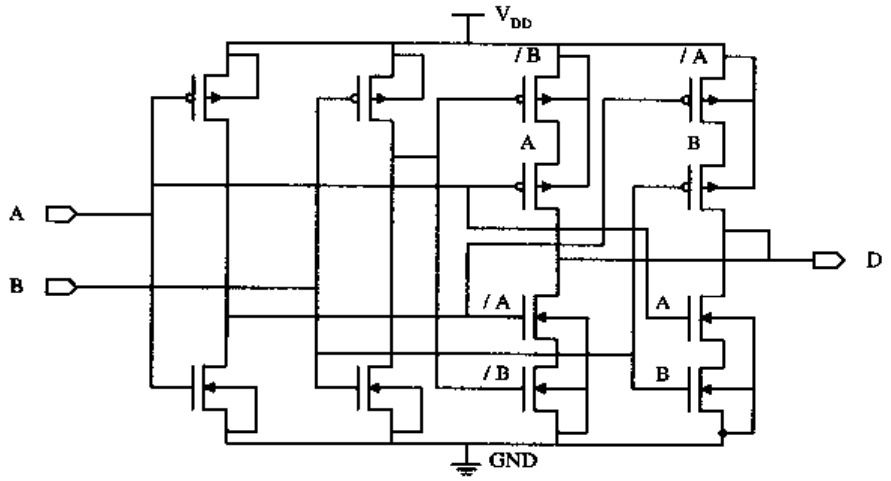
图 1.44 Cmos 结构
cmos(w,datain,ncontrol,pcomtrol);等价于 Nmos (w,datain,ncontrol);
Pmos(w,datain,pcontrol);</td></tr>
<tr><td>rcmos：源与漏极间的阻抗较大</td><td colspan="2">同上</td></tr>
<tr><td>tran：双向导通开关</td><td>数据无条件双向流动</td><td>tran t1(inout1,inout2);</td></tr>
<tr><td>rtran：双向导通开关</td><td>数据无条件双向流动。源与漏极间的阻抗较大,信号通过开关时，信号强度有所减弱</td><td>rtran t1(inout1,inout2);</td></tr>
<tr><td>tranif1</td><td>如果控制端为 1，则数据可以双向流动。如果控制端为 0，则禁止数据传送</td><td>tranif1 t1 (inout1,inout2,control);</td></tr>
<tr><td>rtranif1</td><td>同上，但在传送数据时，信号强度有所减弱</td><td>rtranif1 t1 (inout1,inout2,control);</td></tr>
<tr><td>tranif0</td><td>如果控制端为 0，则数据可以双向流动。如果控制端为 1，则禁止数据传送</td><td>tranif0 t1 (inout1,inout2,control);</td></tr>
<tr><td>rtranif0</td><td>同上，但在传送数据时，信号强度有所减弱</td><td>rtranif0 t1 (inout1,inout2,control);</td></tr>
<tr><td>pulldown：下拉电阻</td><td>只有输入没有输出。将节点的逻辑值下拉到 0</td><td>pulldown (strong0) p1 (neta), p2(netb);</td></tr>
<tr><td>pullup：上拉电阻</td><td>只有输入没有输出。将节点的逻辑值上拉到 1</td><td>pullup (strong1) p1 (neta), p2(netb);</td></tr>
</table>

为了方便读者理解，表 1.7 列出了一些常见的晶体管与门的电路结构。

表 1.7　常见的晶体管与门的电路结构

常见的晶体管与门	结构图示
PMOS	 图 1.45　PMOS 的结构
NMOS	 图 1.46　NMOS 的结构
反相器	 图 1.47　反相器的结构

续表

常见的晶体管与门	结构图示
与门	 图 1.48　与门的结构
或门	 图 1.49　或门的结构
异或门	 图 1.50　异或门

续表

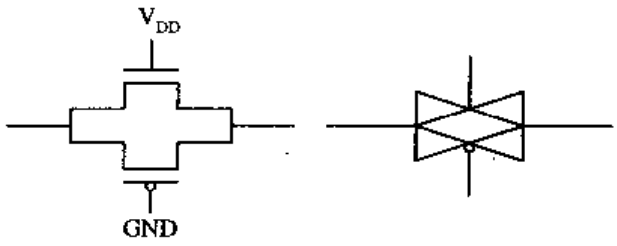
常见的晶体管与门	结构图示
传输门	

图 1.51 传输门的结构与符号

下面是进行晶体管级设计应该了解的一些常识。

1) 简化模型

设计复杂的数字电路时，如果把所有的物理效应都考虑在内，设计就会过于复杂，并且这种复杂没有必要。打个比方，当修建一座桥时，我们没有必要知道水泥的分子结构。因此，我们采用模型(即对器件行为的抽象描述)来简化问题。对器件建模时既要考虑精度，又要考虑复杂性，在两者之间要进行平衡。可以利用简单的一级模型进行手工分析。这种模型精度有限，但有助于我们了解电路的操作和主要的参数。当需要更为精确的结果时，就需要使用二级模型(或称高级模型)来进行计算机辅助仿真。

需要说明的是，随着工艺的进步，器件尺寸越来越小，晶体管不能再用传统的长沟道 MOS 晶体管模型来表示。这中间最重要的差异是速率饱和效应，即晶体管电流与控制电压的关系不再是平方关系，而变为线性。

2) CMOS 闩锁

MOS 工艺含有许多内在的双极型晶体管。这在 CMOS 工艺下会非常讨厌，因为阱与衬底结合会导致寄生的 n-p-n-p 结构。这些类似半导体闸流管的器件会导致 V_{DD} 和 V_{SS} 线的短路，通常会导致芯片破坏或者引起系统错误。

例如，对于一个 n 阱结构。n-p-n-p 结构是由 NMOS 的源、p 衬底、n 阱和 PMOS 的源构成的。当两个双极型晶体管中的一个前向偏置时(例如由于流经阱或衬底的电流引起)，会引起另一个晶体管的基极电流增加。这时正反馈将不断地引起电流增加，直到电路出故障，或者烧掉。

设计者应尽量避免闩锁；而要避免它，就应该使 R_{nwell}(n 阱电阻)和 R_{psubs}(P 型衬底电阻)最小化。这可以通过提供大量的阱和衬底的接触来实现这一点。大电流的器件(例如在输入输出器件中的晶体管)应该用防护环来围绕。这些位于晶体管附近的阱/衬底接触，能减少电阻，甚至减少寄生双极型晶体管的增益。避免闩锁的方法，在许多半导体集成电路书籍中都有介绍，请参看相关资料。闩锁效应在早期的 CMOS 工艺中很重要。在近些年，工艺的改进和设计的优化已经消除了闩锁的危险。

6. 版图级设计

在版图级设计阶段要完成电路的掩膜版图(Floorplanning、布局、布线)，并进行设计规则检查。

设计者综合出网表后，一般都由版图工程师来完成版图。可以用 Cadence 的 Silicon Ensembler、Avanti 的 Apollo 来完成版图设计。

版图完成后，要进行 extract，可以用 Avanti 的 Star_rcxt，然后进行后仿真。如果后仿真不能通过，则需要修改设计。

后仿真完成后，要进行设计规则检查(DRC)、电路规格检查(ERC)、版图验证(LVS)等。完成后，就可以将 GDS II 格式的文件送到制版厂做掩膜版，制作完毕后便可上流水线流片。

说明：GDS II 是指 Geometric Data Standard II。

1) 标准单元的掩膜版图

图 1.52 是一些标准单元的掩膜版图。

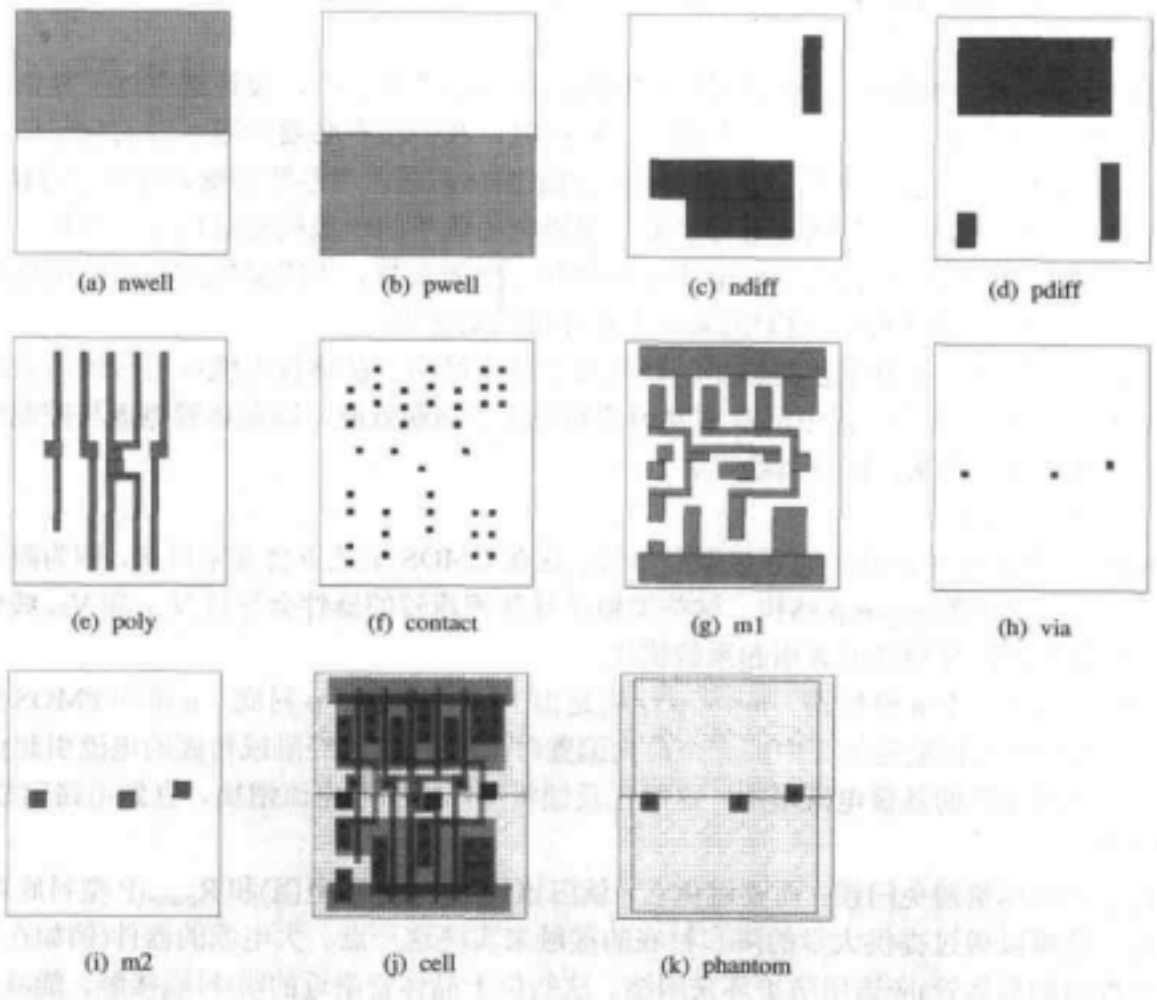


图 1.52 基本单元的版图

在布图时，一般按如下顺序：第一步，放置 I/O 单元，放置兆单元 (Megacell Place)，生成行通道(Row Generation)，再进行电源布线(Power Routing)。第二步，布局(Placement)，生成时钟树，然后进行静态时序分析和优化(STA & OPT)；如果需要的话，要进行布局变更

(Placement ECO)。第三步，布线，进行静态时序分析和优化；如果需要的话，进行布线工程变更(Routing ECO)。最后签收交付(Tape Out)。

流水回来后，还需要对芯片进行测试。

2) 版图与性能

版图设计直接影响电路的延时、功耗等。例如，要降低一个门的传递延时，可以有两种方法：

- 减小负载电容，包括门本身的内部扩散电容、互连线电容、扇出电容。仔细地布版图，可以减少扩散电容和互连线电容。
- 加大晶体管的 W/L(宽度与长度的比率)。但要注意，增加晶体管的尺寸也会增加扩散电容和栅电容。后者会增加驱动门的扇出电容，因而影响速度。

3) Floorplanning、布局、布线

对网表进行处理的第一步是 Floorplanning。Floorplanning 是对系统中的模块进行处理，主要完成以下任务：

- 分配各个模块的位置。
- 分配输出焊盘的位置。
- 规划电源焊盘的数目、位置和分配方式。
- 决定时钟分配的方式(这方面的内容在前面“时钟分配策略”一节中已进行了论述)。

Floorplanning 的目标是使芯片的面积和延时最小。

第二步是进行布局。布局的层次比 Foorplanning 要低，主要定义各元件的位置大小及相关的连线。为提高布局效率，应事先规划好水平及垂直走线的连线层。为确保电路工作的独立性及正确性，应适当加入隔离和屏蔽电路(使阱接触与衬底接触足够大，加入保护等)。

布局时要考虑如下因素：

- 保持元件参数值的变化量在适当范围。
- 使相关元件具有同样的变化趋势(如温度/几何环境相似、隔离、接触点等)，以保证相关元件的匹配。
- 缩短信号线长度，减少耦合电容。
- 提供足够的电流密度裕量。
- 注意信号的干扰。
- 减少芯片的面积。

第三步是进行布线。

4) 设计规则和布局验证

设计规则是版图限制的集合，它保证电路功能正确，没有短路或开路。对版图级设计来说，最主要的要求就是遵守这些规则。这可以通过设计规则检查器(DRC)来进行验证，用该检查器分析一个设计的物理版图是否符合工艺文件所定义的设计规则。

在制造过程中，由于光解析度/化学药品浓度/作用时间/温度等不能完全相同，因此，制造出的产品也不尽相同，在布局时要能容忍这些变异。为了使晶片制作过程的合理变动不致影响制作的结果，版图必须满足流片厂商所提供的布局规范。并且，布局设计要与原电路在功能上一致。为确认所设计的布局满足流片厂商的设计规范及电路的正确性，布局需经布局验证的程序。Diva, dracula 及 hercules 等工具都提供了布局验证的功能。

在设计完成后，仍然不能保证芯片一次性投片(Tape Out)成功。因为在参数性能上的任何失误，都可能导致返工。这就对设计者提出了很高的要求。设计者要仔细检查门级网表，纠正任何很微小的瑕疵(包括文法错误、连接短路、无任何连接的 net、无驱动的输入引脚(pin)、assign 语句、wire 类型以外的 net、数据总线的写法、名字的长度等)，要仔细阅读拿到的时序约束(Timing Constrain)文件，检查时序设定是否完整、合理。

7. I/O 单元的设计

PAD(焊盘)提供了封装接脚到芯片内部的连接。设计 PAD 时，需要对基本电路结构和工艺较为熟悉。PAD 的设计是 IC 设计中比较特殊的一个方面，所以这里用独立的一节来说明。

PAD 的尺寸通常定义为绑定导线需要的最小尺寸，一般是 100 到 150 个平方微米。

在进行 RTL 级设计的时候，可以加入 PAD 的模型，以进行整体上的仿真。下面给出一些常见 PAD 的行为描述和对应的示意图。

1) 输入 PAD

输入 PAD 接收外部信号，并送到 in_data 端，供芯片内部电路使用。实现代码如下：

```
module PAD_IN(pad,in_data);
input      pad;
output     in_data;

assign in_data=pad;
endmodule
```

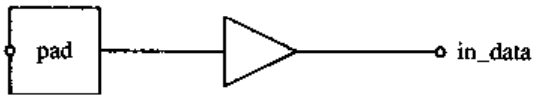


图 1.53 输入 PAD 示意图

图 1.53 给出了输入 PAD 示意图。

根据需要，PAD 可以带上拉晶体管、下拉晶体管、施密特触发器和熔丝。图 1.54 给出了带上拉晶体管的输入 PAD 示意图。

图 1.55 给出了带下拉晶体管的输入 PAD 示意图。

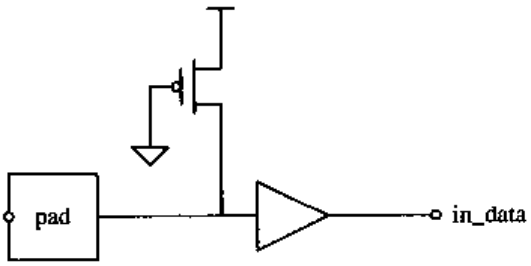


图 1.54 带上拉晶体管的输入 PAD

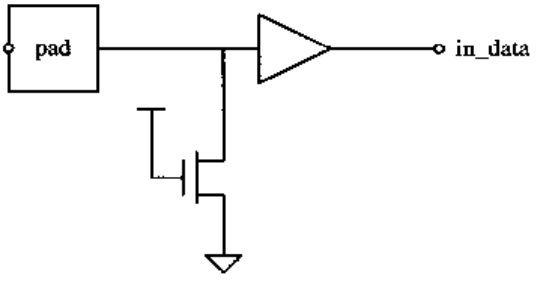


图 1.55 带下拉晶体管的输入 PAD

图 1.56 给出了带施密特触发器的输入 PAD 示意图。

说明：施密特触发器是一种脉冲波形整形电路，可以把变化缓慢的信号或变化不规则的信号转换为陡变信号。更详细的说明请查阅半导体集成电路书籍。

图 1.57 给出了带熔丝的输入 PAD 示意图。

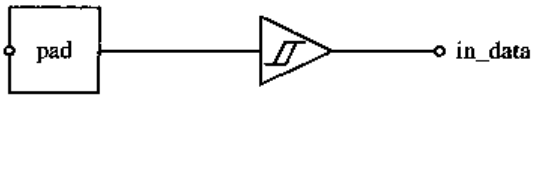


图 1.56　带施密特触发器的输入 PAD

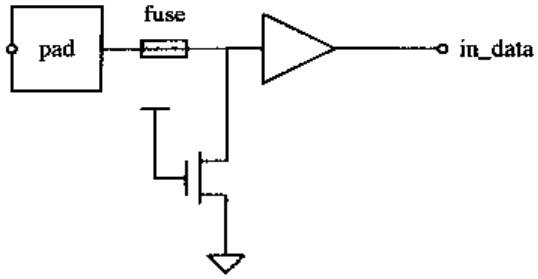


图 1.57　带熔丝的输入 PAD

2) 输出 PAD

图 1.58 给出了输出 PAD 的示意图。实现代码如下：

```
module PAD_OUT(pad,data);
output      pad;
input       data;

assign      pad=data;
endmodule
```

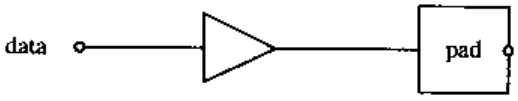


图 1.58　输出 PAD

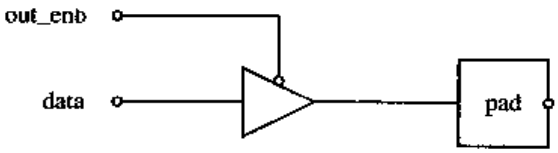


图 1.59　带输出使能的输出 PAD

3) 带输出使能的输出 PAD

图 1.59 给出了带输出使能的输出 PAD 示意图。实现代码如下：

```
module PAD_OUTEN(pad,out_enb,data);
output          pad;
input       data;
input           out_enb;
assign      pad=(out_enb==0)?data:1'bz;
endmodule
```

4) 输入输出 PAD

图 1.60 给出了输入输出 PAD 示意图。实现代码如下：

```
module PAD_IN_OUTEN(pad,data,out_enb,in_data);
inout       pad;
input       data,out_enb;
output      in_data;

assign      pad=(out_enb==0)?data:1'bz;
```

```
assign      in_data=pad;
endmodule
```

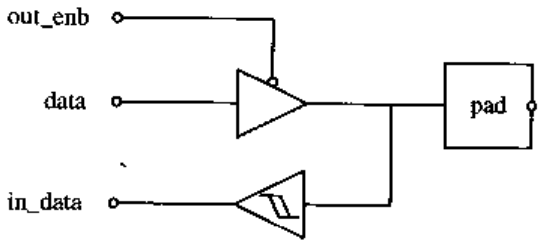


图 1.60　输入输出 PAD

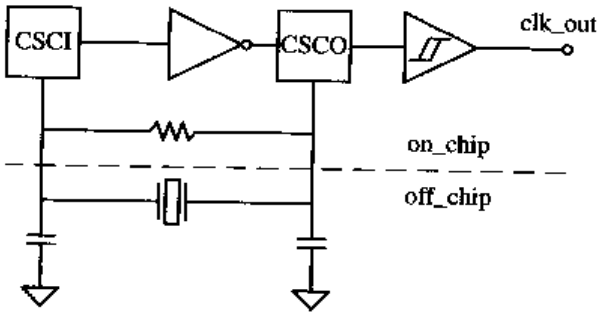


图 1.61　时钟用 PAD

5) 时钟用 PAD

图 1.61 给出了时钟用 PAD 示意图。实现代码如下：

```
module PAD_CLK(OSCI,OSCO,clk_out);
input       OSCI;
output      OSCO;
output      clk_out;

assign      OSCO = OSCI;
assign      clk_out = OSCI;
endmodule
```

说明：在图 1.61 中，晶振、电容构成了时钟发生电路。

6) 电源用 PAD

电源用 PAD 的实现代码如下：

```
module PAD_VDD (pad);
inout     pad;
endmodule
```

电源用 PAD 的示意图略。

7) I/O 单元的设计

设计 I/O 单元需要熟悉制造工艺与电路基本结构。一般用 HSPICE 来完成。HSPICE 可以对 I/O 单元进行仿真，证明其功能符合设计要求后，交给版图工程师进行掩膜设计。

对 I/O 单元进行布局时，一般要考虑如下因素：

- 为提供足够的驱动能力，在输出 PAD 前通常加上驱动电路。
- 为提供内部电路的保护，在输入 PAD 后可加上保护电路。
- 为提供驱动电路及保护电路的电源，I/O PAD 处需有电源。为避免内部电路受 I/O 信号的干扰，I/O 电源及芯片内的电源最好分开。

输出 PAD 因需提供较大的驱动能力，因此一组 power 所驱动的输出不能太多。

放置 I/O 单元时，要考虑到内部各模块的位置、电源 PAD 的个数和种类。同种类的信

号 PAD 需要不同种类的电源，有些相同电压的电源也是不能共用的，特别是模拟信号及其电源本身都需要与其它信号隔离。电源 PAD 的个数计算要兼顾芯片封装的最低要求和芯片内部的功耗。

8. 无源器件的设计

用集成电路制造电阻和电容，精度低，绝对误差大；可制作的范围有限，不能太大，也不能太小；另外，占用芯片的面积大，成本高。因此，在集成电路中应尽量多用有源元件，少用无源元件。

1) 电阻

在 MOS 集成电路中，常用多晶硅电阻，或用 MOS 管做电阻。

单位电阻值：well 井 > diffusion > poly > metal 金属

2) 电容

单层电容：以栅氧化层作为介质，多晶硅作为上电极，衬底作为下电极，是非线性电容。

双层电容：做在氧化层上，电容的上下电极都是掺杂多晶硅，电容值较固定，构造简单。

3) 电感

在通信应用中，电感可增加电路设计的效益，但在 MOS 工艺中难以制造。

CMOS 电感的主要问题：

- 电感值小；
- Q 值小；
- 元件模型未充分建立，现阶段需自行建立等效模型

9. 低功耗设计

随着制造工艺的进步，低功耗的设计技术已成为一种趋势。特别是在移动应用中，对低功耗设计要求更高，以延长电池的使用寿命。

一个 CMOS 电路中，功耗由三部分组成：静态功耗、短路引起的功耗和动态功耗。静态功耗是由泄漏电流引起的，主要存在于伪 NMOS 逻辑中。

静态电路在翻转时会有短路电流存在(在动态电路不存在这样的通路)，从而产生功耗。短路电流所消耗的总功率与运行的晶体管数目、器件的操作模式有关。如果输入/输出端信号的上升、下降时间大致相同，可以减小短路电流。但在实际中，特别是在基于标准单元的设计中，都做不到这一点。因为保证库单元能驱动较大的负载电容，所有的标准单元晶体管都“过大”。这样，输出信号的上升/下降时间常比输入的要小，这就导致了严重的短路功耗。整体的短路功耗可能占到全部功耗的 50%。

采用适当的设计方法，可以控制短路电流和泄漏电流。所以在 CMOS 设计中，主要关心的问题是动态功耗。

动态功耗与电源电压、负载电容、逻辑门的有效翻转活动(0→1 的翻转会产生功耗，称为有效翻转)都有关系。通过控制如下两个参数：电压 V_{DD} 和等效电容 C_{eff}(C_{eff} 是物理电容 C 与翻转概率 $P_{0\to1}$ 的乘积)，可以降低动态功耗。

降低供电电压是降低功耗的最直接的方法，但同时也会降低工作频率，降低噪声容限。

减少门的翻转次数和减少物理电容也能降低功耗。例如，一个 CPL 加法器，比一般的静态 CMOS 加法器的电容要小，因而功耗也小。又例如，用点对点的连线替代单一总线，尽管增加了面积，但减少了节点电容，因而能减小功耗。

10. 可测性设计

以前，电路设计者所关心的主要问题是电路的逻辑功能、速度、时序以及电性能参数等，而不必考虑测试。随着芯片的日益复杂，如果不在设计时考虑芯片的测试，后期的测试会变得很困难。因为电路结构一旦固定，其内部节点的可控制性和可观测性就再也不能改变。

所谓可测试性设计(DFT，Design For Testability)，是指在集成电路设计的初始阶段，将可测试性作为设计目标之一，而不是单纯考虑电路功能、性能和芯片面积。

为方便测试，在设计时应该将正、负边沿触发的时序电路放在不同的模块中。DFT 电路一般在综合的时候加。Synopsys 提倡 DFT 和综合在一块来完成，即 1-pass synthesis。设计者也可以在综合后得到的门级网表上加测试结构。

下面对几种 DFT 技术进行说明。

Ad Hoc 技术是一种早期的 DFT 技术，也称特设法，是针对一个已成型的电路设计中的测试问题而提出的。该技术有增加测试点、分块、利用总线结构等几种主要方法。该技术的缺点是：需要在电路中每个测试点附加可控制的输入端和可观察的输出端，使连线大大增加。

扫描路径测试是一种结构化的 DFT 技术，其基本思想是：把电路中的关键点连接到一个移位寄存器(用作扫描路径)上。当移位寄存器处于串入、并出状态时，可以用来预置电路的状态，从而提高了电路内部节点的可控制性。当用作扫描路径的移位寄存器处于并入、串出状态时，可以把电路内部的关键状态依次移出寄存器链，从而增加了电路内部节点的可观测性，以此达到测试芯片内部的目的。

该方法具有规律性，特别适合于逻辑电路可测试性设计软件工具的应用(几乎所有芯片制造商所提供的单元库中的触发器和计数器单元，都有相应的扫描结构单元)。其次，该方法能够用伪随机序列产生器或自动测试图形产生程序，自动产生测试矢量。第三，该方法仅需要三个附加的管脚供测试用，即扫描输入、扫描输出和测试控制端，就可以控制和观测电路内部的主要节点。

扫描路径测试的缺点是：串、并行锁存器和扫描路径走线占用硅片额外面积。此外，由于在数据路径中增加了多路选择器，从而使电路功能有所下降。第三，它不能直接测试像 RAM 那样的电平敏感元件。

常用的扫描路径测试技术有以下 3 种。

1) 全扫描技术

全扫描技术就是将电路中的所有触发器都用特殊设计的具有扫描功能的触发器代替，使其在测试时链接成一个或几个移位寄存器，这样，电路分成了可以进行分别测试的纯组合电路和移位寄存器，电路中的所有状态可以直接从原始输入和输出端得到控制和观察。这样大大降低了测试的复杂性。

2) 部分扫描技术

部分扫描技术与全扫描技术的主要差异在于部分扫描技术只利用了电路的部分触发器构成移位寄存器，因此，其关键技术在于如何选取触发器。

3) 边界扫描技术。

边界扫描技术是各 IC 制造商支持和遵守的一种扫描设计标准，主要用于对印刷电路板的测试，它通过提供一个标准的芯片/板测试接口，简化了印刷电路板的测试，如图 1.62 所示。

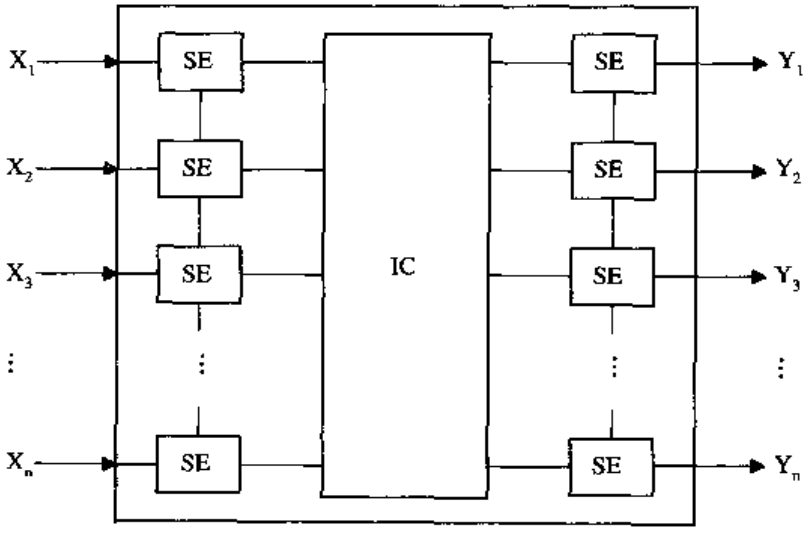


图 1.62　具有边界扫描结构的 IC

该方法的基本思想是：在靠近器件的每一输入、输出引脚处及内部一些关键地方增加一个边界扫描寄存器(BSR)单元，并把这些寄存器单元依次连成扫描链。在正常工作期间，这些单元是"透明的"，不影响电路板的正常工作。在测试期间，由边界扫描寄存器串行地存储和读出测试数据。

利用边界扫描技术，在测试时不需要其它的测试设备，不仅可以测试芯片逻辑功能，还可以测试 IC 之间的连接是否存在故障。相应的标准是 JTAG，即测试联合行动组。人们一般称 IEEE 1149.1 标准为 JTAG。

边界扫描设计的基本结构包括四个部分：

- 测试存取通道(TAP)；
- TAP 控制器；
- 指令寄存器(IR)；
- 测试数据寄存器组(TDR)。

其中，测试数据寄存器组又包括边界扫描寄存器(BSR)、旁路寄存器(BYR)和器件标志寄存器(IDR)，有时还包括一个或几个专用的其它寄存器。

下面分别进行说明。

(1) 测试存取通道(TAP)：提供 IC 测试所需的各种数据。考虑到 IC 的输入输出端是有限的，在边界扫描设计中，TAP 的端口仅有 4 个，分别是：

- 测试时钟 TCK(输入)：边界扫描设计中的测试时钟是独立的，也可以借用原系统的时钟。一般应设计成用 TCK 的上升沿来获取数据，用 TCK 的下降沿来输出数据。
- 测试方案选择 TMS(输入)：由于在测试过程中，需要有数据捕获、移位、暂停和输出更新等不同的工作模式，因此需要用工作方式选择。这里仅有一根输入线控制，所以必须有一个输入序列来确定工作模式。TMS 信号是由 TCK 的上升沿来采样的。采样信号结合已输入的指令以及当前的状态来确定下一个状态，并产生相应的控制信号，以协调整个边界扫描系统有序地工作。
- 串行测试数据输入 TDI(输入)：以串行方式输入的数据 TDI 有两种：一种是信号，它送入到指令寄存器；一种是测试数据(激励、输出响应和其它信号)，它输出到相应的测试数据寄存器中。TDI 的信号在 TCK 的上升沿被采样和输入。
- 串行测试数据输出 TDO(输出)。以串行方式输出的数据有两种，一种是从指令寄存器移出的指令位，另一种是从各数据寄存器移出的数据位。它们决定了 TAP 控制器的当前状态。TDO 的信号是在 TCK 的下降沿开始输出的。

除了上述 4 个 TAP 通道外，标准还提供了一个“测试系统允复位”的输入通道。它的作用是让测试系统强制复位，即扫描系统当前的状态与 TCK 无关。如果没有这个输入端，要求在 TMS 输入序列中确定一个子序列作为“复位”序列。通常以 TMS=1111…1(至少 5 个 1)作为复位序列。

(2) TAP 控制器：可将串行输入的 TMS 信号进行译码，使边界扫描系统进入相应的测试模式，并产生该模式下所需要各个控制信号。例如，允许指令装入指令寄存器；将串行输入信号 TDI 逐位移入数据寄存器，同时将数据寄存器中的数据以串行方式从 TDO 移出；执行捕获测试数据、移位操作和刷新输出数据等操作。TAP 控制器的详细说明见 IEEE 1149 标准。

(3) 指令寄存器：是一个附有锁存器的移位寄存器。它的长度等于指令的长度(指令长度在不同的系统中是不同的)。

在进行测试时，指令经 TDI 移入指令寄存器 IR，然后送入指令锁存器，最后将锁存器中的指令译码后，配合 TMS 信号，产生控制边界扫描电路的各种信号。

在正常的工作条件下，指令寄存器 IR 的最低两位应分别是 0 和 1(最低位是 1)。换句话说，每个指令是 01 结束。这样做的目的，是便于在用移位指令操作和测试 IR 移位功能时，可确定故障所在的 IC。

(4) 测试数据寄存器组：包括一些寄存器，其中旁路寄存器 BR 和边界扫描寄存器 BSR 是必须要有的。旁路寄存器的作用是：当在测试一个很长扫描链上的某一个芯片时，不需要测试的链路短路，以节省测试时间。边界扫描寄存器是最重要的寄存器，完成数据的输入、输出锁存和移位等操作。测试数据寄存器组中的每个寄存器都可以接入扫描链中(即从 TDI 开始到 TDO 之间的扫描路径中)。在某个时刻接入哪一个寄存器取决于当时在指令寄存器中的指令。

采用边界扫描测试方法，可以很方便地观察和控制电路的输入、输出及内部各个节点，从而很方便地对集成电路及印刷电路板进行测试。

另一种降低测试复杂性的方法是让电路自己测试自己。这样的一个测试不要求额外的向量，并且可以在很高的速度下进行。

一个完整的测试结构如图 1.63 所示。测试信号发生器产生的测试序列加到待测电路中，然后由输出响应分析器检查待测电路的输出序列，以确定该电路有无故障。

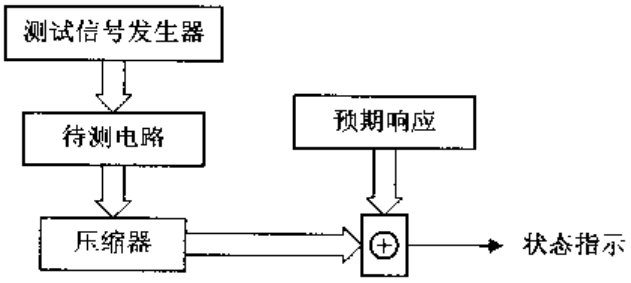


图 1.63　BIST 的一般结构

如果 CUT 具有自己产生测试信号、自己检查输出信号的能力，则称该电路具有内建自测试(BIST)功能。

由图 1.63 可知，对数字电路进行 BIST 测试，需要增加三个硬件部分：测试序列生成器、输出响应分析器和测试控制部分。

内建自测试技术的优点在于：将测试激励的生产和结果检测等放在芯片的内部，只需一条线将检测结果送出片外，或者在片内增加故障校正电路，自动修正电路中的某些故障。

采用内建自测试方法设计的芯片，其测试和使用都很方便。然而，一般情况下，在电路内部生成测试矢量较复杂，并且会增加芯片面积。通常内建自测试只被用于具有大量重复结构的电路，比如存储器等。测试矢量的生成一般采用伪随机序列，测试结果的检测通常采用签名校对或称特征分析技术。

小结：

用有限的输入向量就可以测试大多数组合逻辑模块。但时序电路的测试较为复杂。要测试一个状态机中的一个给定错误，仅仅施加正确的输入激励是不够的，还必须使所测电路进入到一定的状态。这就要求施加一系列的输入，使测试费用变得非常昂贵。

解决方法主要有两个：一个方法是在测试时将反馈环路打断，从而将时序网络转换为组合模块，这就是所谓的扫描测试方法的思想；另一种方法是让电路自己测试自己，这样的一个测试不要求额外的向量，并且可以在很高的速度下进行。

提高集成电路的可测试性有一些一般原则，例如：尽量采用同步逻辑电路；采用有置位或复位端的触发器以便将芯片置于确定的状态；将反馈环用门控制；尽量将多余管脚用于测试等。

11．IC 设计中常见的错误

分析一个电路是否有错误的要点有：

时序是否有问题，是否有竞争冒险(如果电路的输出与输入信号的次序有关，而输入信号几乎同时到达，这时候在输入就存在着竞争问题)；

噪声容限是否达到要求；

是否有冲突 contention(两个以上的信号驱动同一个电路节点)；

信号是否会发生耦合(Cross Coupling)；

是否违反 ERC；

健壮性够不够；

1.2.5　设计验证

在 IC 设计过程中，需要进行大量的验证工作。首先是原型机的验证。根据选用的 FPGA

及设计目标，画出相应的电路板。在 RTL 设计完成后，用相应的综合工具综合成网表。将网表的延时信息反标到网表，进行时序仿真，如果没有错误，则将网表文件下载到 FPGA 中调试。

其次是真实情况的验证，流片后，一般要有 EVChip(一种封装形式，引脚较多，便于调试)和普通封装两种形式的芯片。这时候就可以进行真实情况的验证。

1.2.6 测试

测试是集成电路设计和生产过程中最重要的环节之一。

一般而言，测试可以分为功能测试和制造测试两种。功能测试的目的在于验证设计是否能正确的按照技术条件实现其功能，保证设计与意图相匹配；制造测试的目的在于检查生产的每一芯片是否合格，也称为结构测试。

我们这里讨论的测试是指结构测试。

设计与测试是芯片开发过程中相关联的两个环节。IC 工程师要经常与版图工程师、验证工程师进行交流，了解一些后端的知识是非常必要的，对项目管理者更是如此。当然，本书是面向 IC 设计者的，所以我们对这些内容只进行简明的概述。具体的测试可分为：

分析性测试：对错误进行定位，调试时用。

功能性测试：确定制造出的芯片功能是否正确。由于对每一个模块(die)都进行功能性测试，所以应该尽可能简单。

参数化测试：在一定的工作条件(特定温度与电压)下，测试噪声容限、传递延时、最大时钟频率等。参数化测试通常分为动态(交流)和静态(直流)测试。

一个典型的测试过程流程如下：

首先，根据设计规格，产生一些测试量，测试向量包括输入的数据(时钟，电压，输入信号)及期望的数据(测试向量也可以在仿真的过程中产生)。

将测试向量装入到测试仪中。用一个探测卡将测试仪的引脚跟 die 或封装上的相应引脚连接起来。测试仪执行测试程序，将输入序列加到待测电路上，并将预期的响应与得到的响应进行比较。如果有差异，则被测部分被标注为有错的(常用墨点标注)。划片时，有错误的 die 被抛弃。

此外，还要进行直流、交流特性的测试，进行良率和故障分析。

1.3 FPGA 的设计

FPGA 是 IC 设计的一个重要领域，具有设计周期短(布局布线非常容易，并且 FPGA 芯片在出厂前都经过测试，所以相当于只有 ASIC 设计的前端)、研发费用低(不需要投片的费用)、调试方便(引脚非常多)的优点，适用于上市时间短、中等规模的产品开发。在 ASIC 设计中，常用 FPGA 进行原型机验证。

FPGA 产品一般分为两种类型，一类是基于乘积项(Product-Term)技术，用 EEPROM(电可擦除可编程只读存储器)或 Flash(闪存)制造，多用于 5000 门以下的小规模设计。另一类基于查找表(Look-Up table)技术，用 SRAM(静态存取存储器)工艺制造，密度高，触发器多，

多用于 10 000 门以上的大规模设计。这里我们只介绍第二类产品。采用 SRAM 工艺制造的 FPGA，掉电后数据会消失，因此，调试期间可以用下载电缆配置 FPGA 器件，调试完成后，需要将数据固化在一个专用的 EEPROM 中(用通用编程器烧写)，上电时，由这片配置 EEPROM 先对 FPGA 加载数据，十几个毫秒后，FPGA 即可正常工作。

说明：FPGA 的概念有些混乱。Xilinx 把基于查找表技术、SRAM 工艺的都叫 FPGA，把基于乘积项技术、EEPROM/Flash 工艺的都叫 CPLD。 Altera 把自己的 PLD(即可编程逻辑电路)产品中的 MAX 系列(乘积项技术，EEPROM 工艺)、FLEX 系列(查找表技术，SRAM 工艺)都叫作 CPLD(即复杂可编程逻辑电路)。为避免混乱，我们把这些都称为 FPGA。

1.3.1 FPGA 中逻辑实现原理

下面以基于查找表技术的 FPGA 为例，说明逻辑实现的基理。图 1.64 给出了查找表的一种基本单元结构，该结构实现了特定的组合逻辑功能。

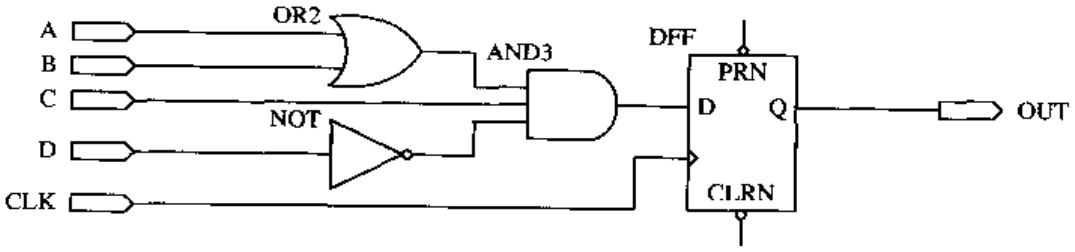


图 1.64 查找表的基本结构

A、B、C、D 由 FPGA 芯片的管脚输入后进入可编程连线，然后作为地址线连到 LUT，LUT 中已经事先写入了所有可能的逻辑结果，通过地址查找到相应的数据，然后输出，这样组合逻辑就实现了。该电路中 D 触发器是直接利用 LUT 后面 D 触发器来实现的。时钟信号 CLK 由 I/O 脚输入后进入芯片内部的时钟专用通道，直接连接到触发器的时钟端。触发器的输出与 I/O 脚相连，把结果输出到芯片管脚。这样 FPGA 就完成了所需的功能。(以上这些步骤都是由软件自动完成的。)对于一个 LUT 无法完成的电路，就需要通过进位逻辑将多个单元相连，这样 FPGA 就可以实现复杂的逻辑。

由此可见，FPGA 的设计较容易，设计者不需要深入了解集成电路的基本结构，只需要熟悉硬件设计语言，掌握软件工具的使用就可以上手了。Xilinx 和 Altera 是两家最著名的 FPGA 厂商。读者可以在它们的网站上查阅相关产品的文档。Xilinx 公司的开发工具称为 ISE(原来的 Foundation 的升级版，现在已经有 5.1 的版本)，Altera 公司最新的开发工具是 Quartus(取代原来的 Maxplus Ⅱ)。这里分别对这两家的产品的构造、开发工具进行简介，并举例进行示范。

1.3.2 Altera 的 FPGA

1. 基本构造

不同档次、不同类别的 FPGA，例如 Altera 的 MAX9000 系统与 APEX20K 系列，其结构是不一样的，但基本构造相似，都由三个部分构成(见图 1.65)：

(1) 逻辑块阵列：构成了 PLD 器件的逻辑组成核心。

(2) 输入 / 输出部分。

(3) 连接逻辑块的互连资源(由连线组成，也包括可编程的连接开关，用于逻辑块之间、逻辑块与输入 / 输出块之间的连接)。

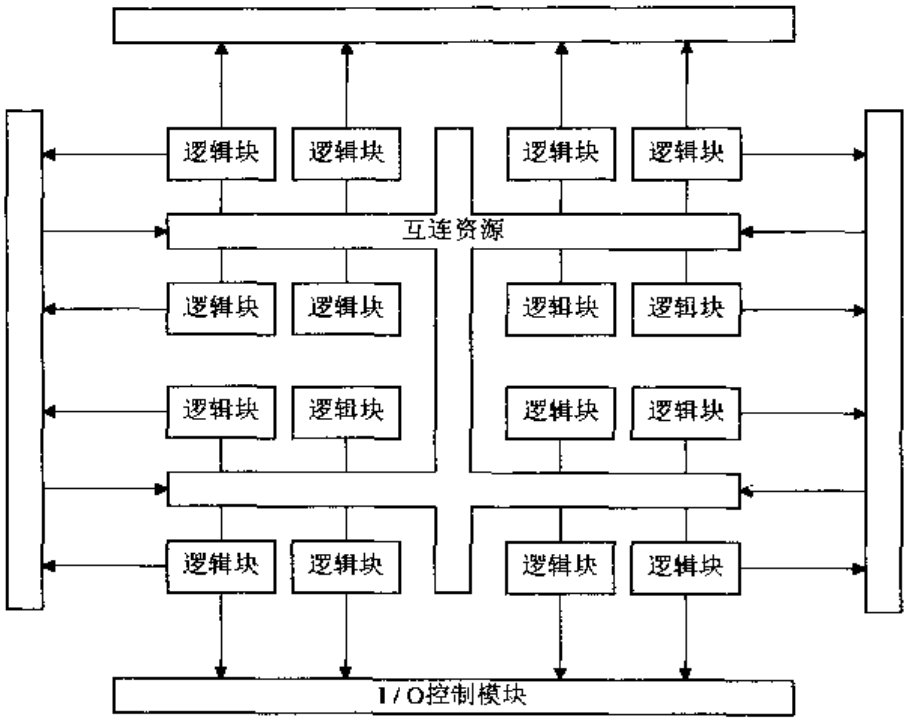


图 1.65　FPGA 的基本构造

Altera 的 FLEX/ACEX 等芯片的结构如图 1.66 所示。

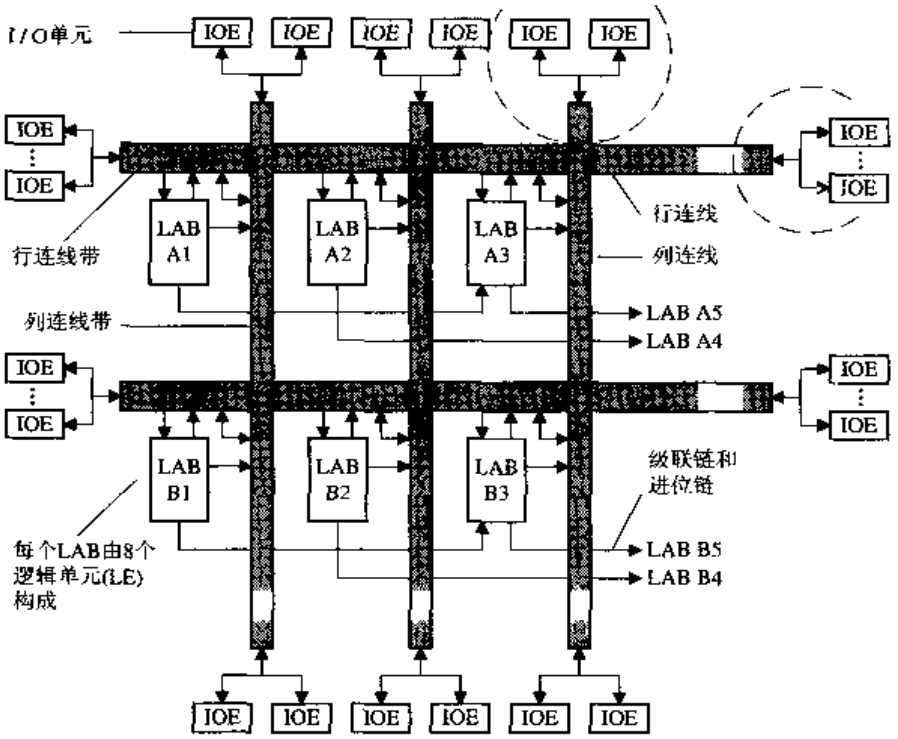


图 1.66　Altera FLEX/ACEX 芯片的内部结构

FLEX/ACEX 的结构主要包括 LAB、I/O 块、RAM 块(未表示出)和可编程行/列连线(见图 1.66)。在 FLEX/ACEX 中，一个 LAB 包括 8 个逻辑单元(LE)，每个 LE 包括一个 LUT、一个触发器和相关的相关逻辑(见图 1.66)。LE 是 FLEX/ACEX 芯片实现逻辑的最基本结构(Altera 其它系列，如 APEX 的结构与此基本相同，具体请参阅数据手册)，如图 1.67 所示。

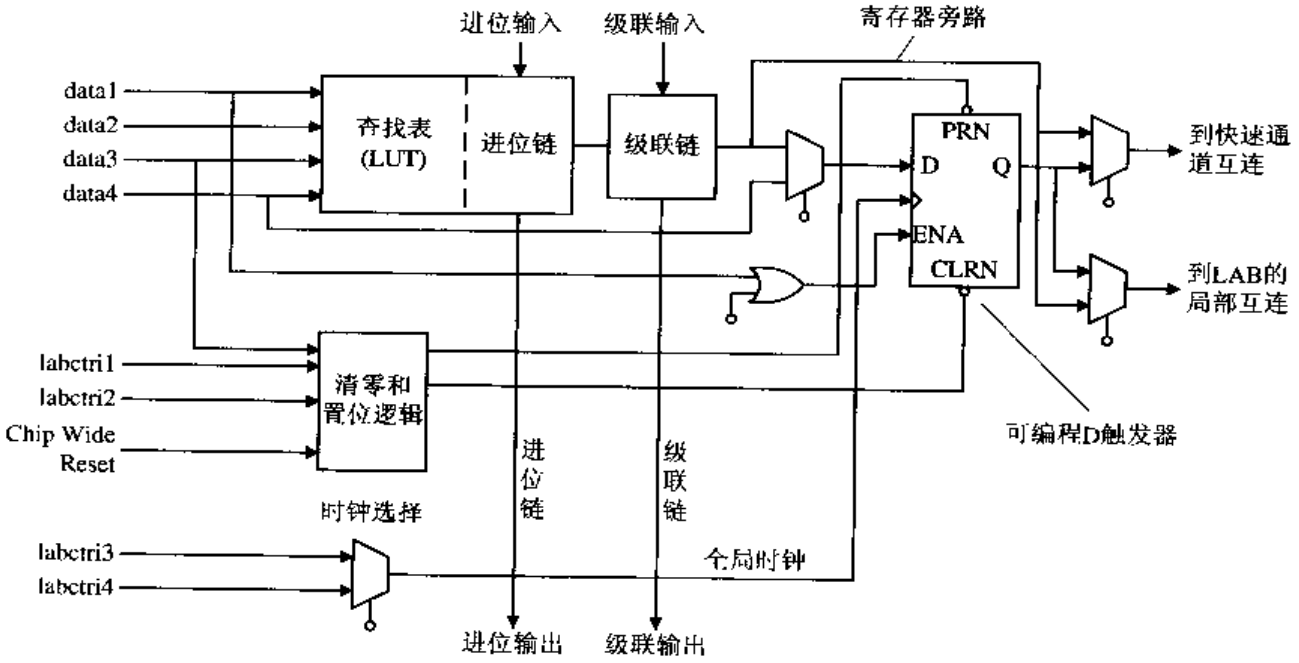


图 1.67　逻辑单元(LE)内部结构

2. Quartus 软件的安装与功能介绍

1) 安装 ByteBlaster

用 Maxplus II 或 Quartus 时，需要安装 ByteBlaster 功能。下面举例说明在 Windows 2000 下安装 ByteBlaster 的过程。

在控制面板中，选择添加/删除硬件，然后选择“下一步”，再选择“添加/排除硬件故障”，选择“添加新设备”，选择“下一步”；计算机询问“是否选择属于搜索新硬件”，这时候选择“否，我想从列表中选择硬件”，硬件类型选择“声音，视频和游戏控制器”。计算机提示选择驱动程序的时候，选择“从磁盘安装”，找到所安装的 Maxplus II 或 Quartus 软件的安装目录下的 drivers 目录，从 Windows 2000 下选择所需的 inf 文件即可。

2) 设计输入

Ququtus 支持多种输入方式，包括框图/原理图、Verilog、VHDL、AHDL、EDIF 等。常用框图来描述顶层设计。选择菜单 File→New→Block/Schematic document，出现图形输入窗口，在工具栏上选择 Block，即可输入相应的输入输出端口。

原理图是最传统的电路输入方式。选择菜单 File→New→Block/Schematic document，出现图形输入窗口。在窗口中双击，弹出选择窗口，从中选择所需的模块就可以了。

Verilog/VHDL 是最流行的输入方式。Ququtus 提供了相应的支持。许多 IC 工程师更喜欢用 ultraedit 或者 ActiveHDL。

AHDL 是 Altera 制定的硬件描述语言，这里不作介绍。

EDIF 是标准的网表文件。设计者给出 RTL 级的 Verilog/VHDL 后，经过综合，得到 EDIF 文件，可以由 Ququtus 调入，完成布局布线等工作。在实际中常用这种方式。

3) 建立工程

选择菜单 File→New project wizard，打开新建项目指南，如图 1.68 所示，第一项选择工作目录，第二项选择项目名称，第三项选择顶层设计的名称。第二项与第三项一般取相同的名字。

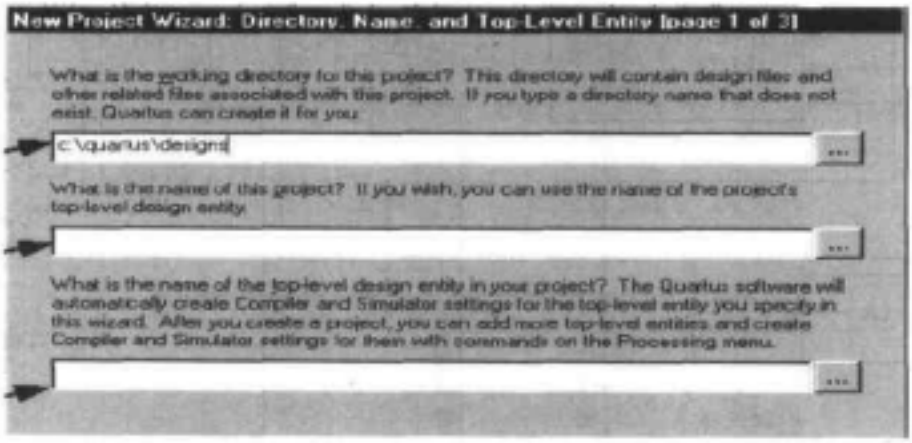


图 1.68　建立工程窗口 1

接下来，选择 Next，进入下一步。如图 1.69 所示，在 File name 一项中加入设计文件。单击 User Library Pathnames，可以加入用户自己定义的库函数的路径和文件名。再单击 Next 就可以了。

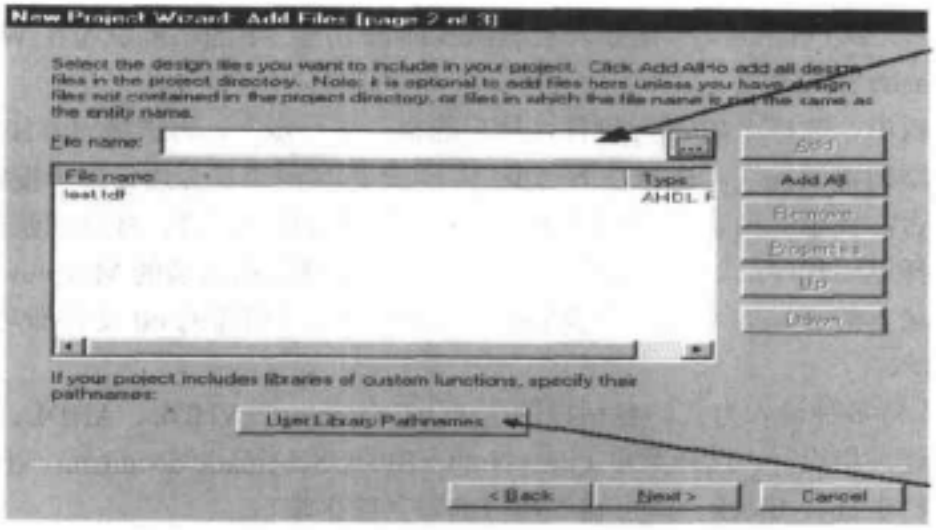


图 1.69　建立工程窗口 2

4) 选项设置

设计完成后，要进行综合、布局布线等工作，在 Ququtus 中，这统称为编译。在编译前，需要先设置编译选项。选择菜单 Processing→Compiler Settings Wizard，在 Focus point 中指

定编译模块，在 Setting name 中设定名字，选择下一步，然后选择是只进行综合(编译只进行到综合就停止)还是进行完全编译(进行综合、网表输出、器件配置，给出真实的时序信息和配置文件)，选择是普通编译还是 Smart 编译(只对更改的部分进行编译，速度较快，但需要对原来的部分进行保存，因而所需空间较多)。接下来，选择器件类型。在 Family 中选择所用的器件类型，并选择是自动选择某一器件还是自己选择。选择菜单 Processing→Compiler Settings，弹出窗口如图 1.70 所示。

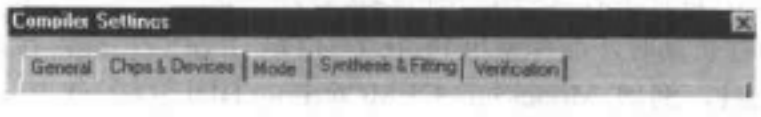


图 1.70　编译设置窗口

选择 General，在“Current Compiler Settings”中选择编译设置，在“Compilation focus”中选择要编译的模块。

选择 Chips& Devices，选中“Specific device selected in ‘Avaiable devices’ list”，选择所需的器件，选中 Assign Pins，进行引脚的设置。

选中 Synthesis&Fitting，选择所用的配置器，并指定是优化 I/O 引脚的延时还是优化内部延时。

5) 支持的第三方工具

(1) 仿真工具。ModelSim(SE 版)；ModelSim(OEM 版，包括 OEM 厂商的后仿真库，Altera 和 Xilinx 都有自己的版本)；Cadence 的 Verilog XL；Synopsys 的 VCS/VSS 和 Primetime(静态时序分析工具)。

(2) 综合工具。Synopsys 的 Design Compiler、FPGA Express；Synplicity 的 Synplify；Mentor 的 Leonardo Spectrum。

在 Ququtus 中设置这些第三方工具时，在菜单 Project→EDA Tool Setting 中，选择相应的综合工具与仿真工具。例如，如果我们用 ModelSim 进行仿真，用 FPGA Express 进行综合，则在 Design Entry/Synthesis tool 一项中选择它，并点击旁边的 Setting，在弹出的窗体中选择相应的数据格式(Verilog 或 VHDL)就可以了。Ququtus 会给出相应的库映射文件.lmf。

6) 编译

设置完毕后，单击完全编译的图标。Ququtus 在状态栏中显示编译进度，并显示编译产生的信息。编译完成后，会给出编译报告，如图 1.71 所示。

在编译报告中，包括了编译设置、Floorplan 显示、器件的资源利用率、状态机实现方式、编译时间、时序分析结果等内容。

图 1.71　编译报告

在时序分析一项中，给出了系统的最大时钟频率 Fmax。其中，最坏情况下的 Fmax 放在最上一行。此外，还给出了 Tsu(建立时间)、Th(保持时间)、Tco(时钟到输出的时间)等信息。

7) 仿真

Quartus 本身可以进行仿真，以图形方式画出输入信号，显示输出结果。它还支持 Testbench 方式，用 Tck 写仿真脚本文件。此外，它还支持第三方的仿真工具。

用第一种方法进行仿真时，先从编译模式切换到仿真模式，对输入信号加入相应激励就可以了。缺省进行后仿真(有延时信息)，也可以进行功能仿真。

用第三种方法进行仿真时，选择菜单 Project→EDA Tool Setting，在 Simulation tool 项中选择所用的仿真工具，例如 ModelSim OEM(Verilog HDL Output from Quartus)。利用 Quartus II 产生.vo 和.sdo 文件，前者是门级网表，后者带有延时信息。在 ModelSim 中，编译门级网表和 testbench，并加入.sdo 文件(延时标注)，选择相应的器件库，就可以进行后仿真了。关于仿真工具更详细的信息，请参看“常用 FPGA 设计工具的使用”一节。

8) 下载

在布局布线后，会产生一个 sof 文件和 pof 文件，两者都可以用于下载。

Altera 的编程电缆称为 Byteblaster，电缆一端装在计算机的并行打印口上，另一端接在 PCB 板上的一个 10 芯插头上，FPGA 芯片有四个管脚(编程脚)与插头相连，如图 1.72 所示。

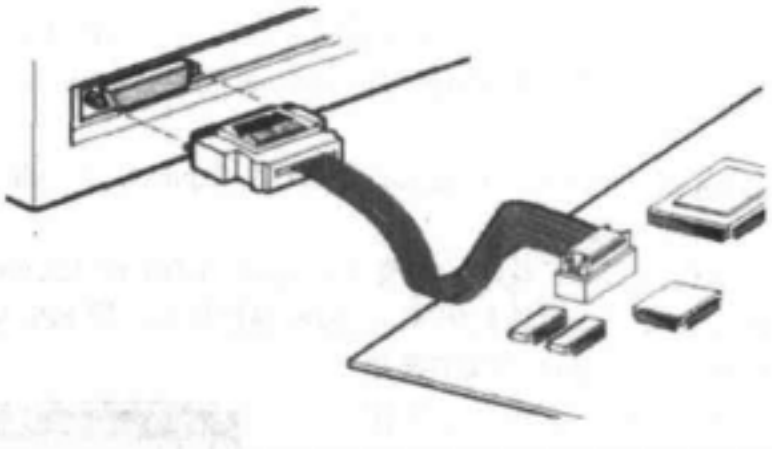


图 1.72　下载示意图

下载之前，要先将 FPGA 焊在 PCB 板上，接好编程电缆。

下载时，选择菜单 Processing→Open Programmer 或者点击快捷方式。在弹出的窗口中的 Mode 项中选择 JTAG 方式，然后单击 Setup，从下拉菜单中选择所用的下载电缆(例如 ByteBlaster)。选择 Add File，再选择生成的 sof 或 pof 文件，单击 start 就可以了。Quartus 会显示下载的进度和相关信息。

9) LPM 库

用户可以利用加入Quartus II 提供的LPMs、宏功能等函数以及用户自己的库函数来设计。

1.3.3 Xilinx 的 FPGA

1. 基本构造

我们看一看 Xilinx Spartan II 的内部结构，如图 1.73 所示。

Spartan II 主要包括 CLBs、I/O 块、RAM 块和可编程连线。在 Spartan II 中，一个 CLB 包括 2 个 Slice，每个 Slice 包括两个查找表(LUT)，两个 D 触发器和相关逻辑。 Slice 可以看成是 Spartan II 实现逻辑的最基本结构 (Xilinx 其他系列，如 SpartanXL、Virtex 的结构与此稍有不同，具体请参阅数据手册)。

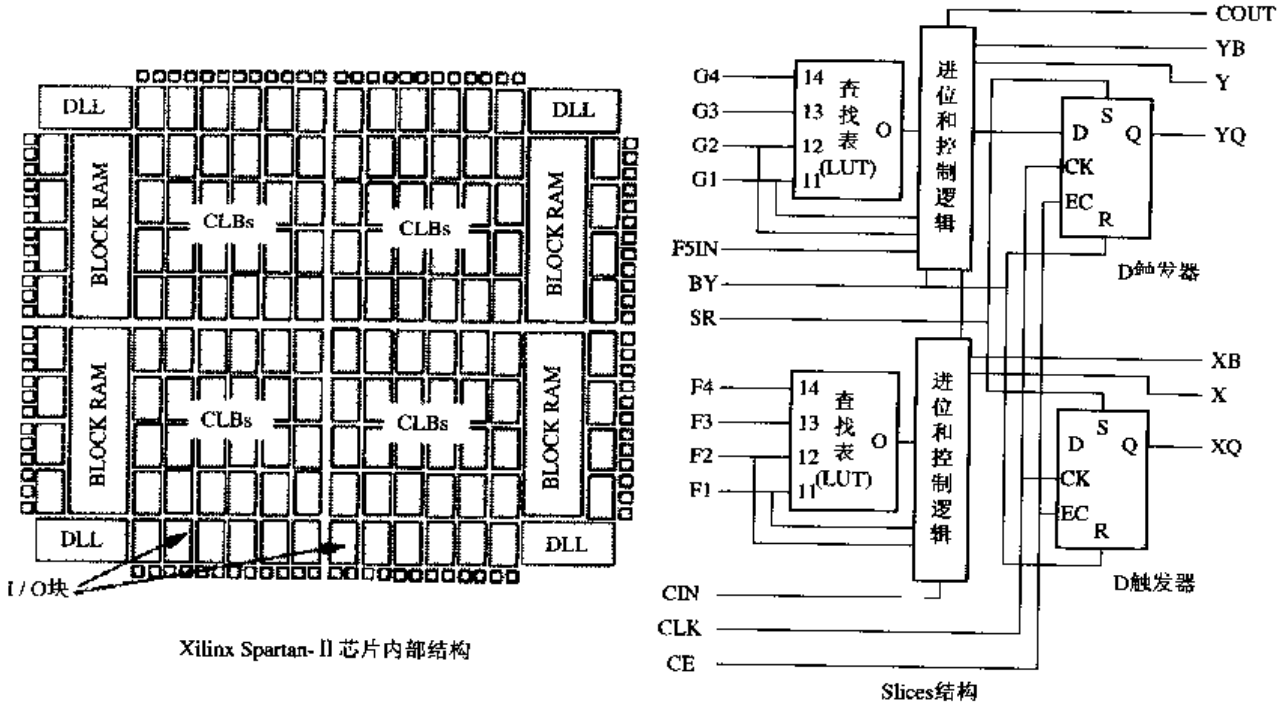


图 1.73 Xilinx Spartan II 的内部结构

2. ISE 软件的安装与功能介绍

ISE 软件是 Xilinx 公司的 FPGA 开发软件，它提供了从设计输入到综合、布线、仿真、下载的功能。为了达到最优的性能，许多功能都是由第三方软件来完成的。其中，原理图输入用的是第三方软件 ECS；HDL 综合可以使用 Xilinx 公司开发的 XST、Synopsys 的 FPGA Express 和 Synplicity 公司的 Synplify/Synplify Pro；测试台输入是图形化的 HDL Bencher；状态图输入用的是 StateCAD；前、后仿真则可以使用 Modelsim XE(Xilinx Edition)或 Modelsim SE。

1) 设置工作环境

通常在第一次使用 ISE 或需要对某些项目进行修改时，才需要设置工作环境。一般有以下几项需要设置：

(1) 常用的。这主要是设置项目管理器中文件的显示方式、字体、窗口的显示方式等。一般使用默认值。

(2) 编辑器。这里可设置跳格键(Tab)的字符个数、编辑器的字体等。

(3) 流程设置。

(4) 工具设置。这里主要设置仿真器 ModelSim、HDL 测试台生成工具 HDL Bencher、状态图输入工具 State CAD 的工作目录。事实上主要是设置仿真工具 ModelSim 的工作目录，因为后两项通常在安装完后 ISE 已经帮你设置好了。

2) 建立工程

ISE 要求在对文件进行综合或布线之前必须要有一个已存在的工程。在建立新工程时，需要设置：工程名；工程所在目录(ISE 所产生的输出文件全部放在该目录下)；器件系列，即你所采用的 FPGA 是 Xilinx 的哪一大类；器件型号；综合软件。

建立工程的步骤如下：

(1) 单击菜单 File→New Project。

(2) 在 New Project 对话框中的 Project Location 下，键入新工程项存放的目录，或者单击 Project Location 旁的浏览按键，选择你想存放新工程项的路径。

(3) 在 Project Name 下输入工程名，这时在 Project Location 下会自动创建一个与工程名同名的子目录。

(4) 使用 Value 处的下拉菜单，可以对工程的属性进行选择，包括器件系列(Device family)、器件(Device)及设计流程(Design flow)，EDIF 还是 XST Verilog、XST VHDL。其中，XST 是 Xilinx 自己开发的综合工具。

(5) 最后，单击 OK 按钮。

ISE 会在工程项导航器(Project Navigator)中创建和显示所建立的新工程。

3) 加入源文件

如果你已有源文件，直接加入即可，否则可采用原理图方式或写 HDL 代码方式新建文件再加入。

在工程中单击鼠标右键，选择“Add New Source”，即可加入已有的源文件。

4) 综合与布局布线

通过综合将设计转换成具体的电路图。如果对设计中的某些性能(如速度等)有要求的话，要先设置好，综合器会根据性能要求来综合电路。

布局布线将综合后生成的电路图分配至对应的 FPGA 中。如果所设计的电路太大，而所选器件型号资源太少，或者将某些专用 I/O 作为输入/输出脚，则布局布线会报错。

具体步骤：

在 sources in process 窗口中选择最顶层的源文件，在 processes for current source 窗口中双击设计实现(implement design)。这样就运行了所有相关的进程，包括 synthesize，translate，map，place & route。在 process for current source 窗口中，打对钩的标记指示进程已经成功地运行，感叹号标记表示进程已经运行，但是包含有系统给出的警告，有关警告的更多信息可以从副本(transcript)窗口中获取。初学者可以分析一下这些警告信息，对增长设计能力非常有好处。

5) 仿真

Xilinx ISE 附带了一套自动产生 Testbench 的程序 HdlBencher，在该程序中它可以自动将设计中的输入、输出找出来，用户可以以图形化的方式编辑好输入波形和所预期的结果，然后调入仿真器中进行仿真。仿真器用 ModelSim。

6) 配置管脚

在该步骤中，将设计中用到的输入/输出端口再分配给 FPGA 的 I/O 端口。

7) 下载

在 sources in process 窗口中选择最顶层的源文件，在 processes for current source 窗口中双击产生配置文件(generate programming file)，这样就产生了配置文件。

将配置文件载入 FPGA 中，下载成功后就可以测试实际电路了。如果需要脱机配置，则必须将配置文件写入 ROM 中。

1.4 常用软件的使用

1.4.1 常用软件的分类

1. 时序图工具

在写设计规范的时候，需要规划好设计的时序。下面介绍用 Timing Designer 来完成此项工作。

2. HDL 语言的输入

许多工程师都喜欢用 UltraEdit，它的确非常好用。ActiveHDL 所带的编辑器也很不错。Xilinx、Altera、Synoplicity 提供的开发工具中也带有编辑器。Modelsim 所带的编辑器不是很好用，特别是在加中文注释的时候很不方便。

3. 状态图的输入

设计者可以采用类似于画流程图的方式，设定好各种状态转换条件后，依靠专用软件自动生成 HDL 源程序，简化了输入过程。这类软件有 Visual Software Solutions、Inc 的 StateCAD、Mentor 公司的 HDL Designer Series 等。

4. 电路图输入

在 Quartus 和 ISE 中都提供了很好的支持。

5. 仿真

HDL 仿真分为功能仿真、门级仿真和后仿真。HDL 仿真软件的数目非常多，例如 VCS、VSS、NC Simulator、Verilog-XL、Modelsim 等。FPGA 厂家在自己的开发环境中也加入了仿真的功能。总体来讲，在 PC 机上，ModelSim 用得最普遍(现在初学者也有许多使用 Active-HDL，界面较友好)。

6. 综合

最著名的综合工具是 Synopsys 公司开发的 FPGA Express、FPGA Compiler、Design Compiler 等。熟悉 FPGA Express 的用户最多。但 Synopsys 公司已不再开发该系列，而用 FPGA Compiler 来取代。此外，Synplicity(Synplify、Amplify、Certify 和 Synplify Asic)、Mentor(Leonardo spectrum)等公司也有自己的产品。一些 FPGA 公司也开发了自己的 HDL 综合器，例如 Xilinx 的 XST。

这里对在 IC 设计中常用到的画时序图的软件 Timing Designer、仿真软件 ModelSim/NC

simulator 和综合软件 FPGA Express 进行简要介绍。至于 IC 设计中，综合软件常用 Design Compiler，有关知识将在第 3 章中进行说明。

1.4.2 Chronology Timing Designer 的使用

该工具可以方便地构造设计的时序图，并具有验证时序的功能。

1．程序的启动

单击该程序，会弹出 Timing Viewer 和 Parameter 两个窗口。

2．加入信号

在 Timing View 窗口的工具栏中单击，或者单击菜单 Tools→Add Signal，弹出信号属性对话框。在其中输入信号名称(假设是 Input1)和初始状态(这里假设是 L)。

然后在 Timing View 窗口中单击鼠标，就可以绘出信号的波形。

3．加入延时

利用该软件，可以很方便地延时。

在工具栏中单击，单击输入信号 Inputa 的某个边沿(25 ns 处)，再单击信号 Outputa 的某个边沿，在弹出的延时属性窗口中输入延时的范围就可以了，如图 1.74 所示。

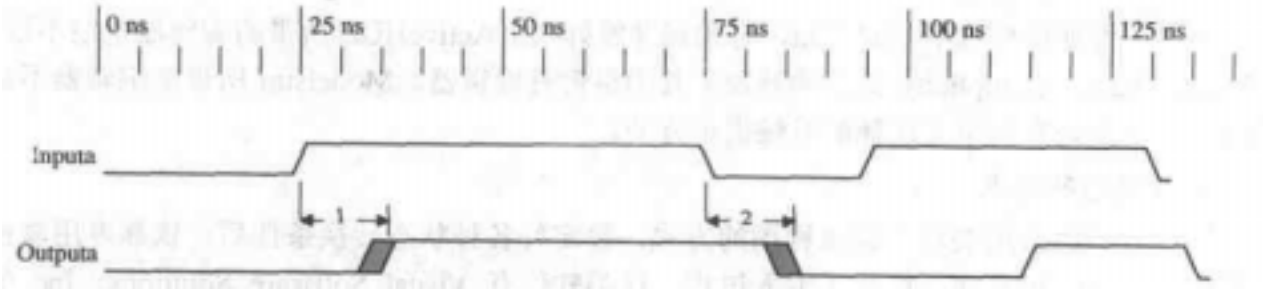


图 1.74　延时的加入

4．加入时钟

在 Timing Designer 中，加入时钟非常方便。

单击工具栏中的，在弹出的时钟属性窗口中，输入信号的名称、频率、占空比、上升下降抖动时间等就可以了。图 1.75 给出了一个频率为 20 MHz、占空比为 40%、上升下降抖动都为 1ns 的时钟信号。

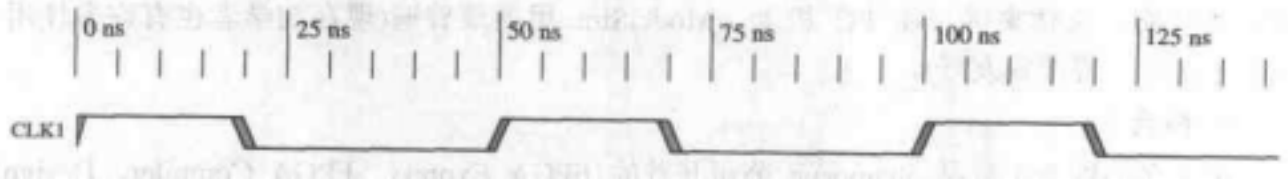


图 1.75　时钟信号

5．设置建立时间、保持时间约束

单击工具栏中的，然后单击 Inputa 的某个边沿，再单击时钟 clk 的某个边沿，弹出约束属性窗口。在对话框中输入约束的名称、约束的时间范围就可以了。

关于 Timing Designer 更深入的用法，请阅读相关帮助。

1.4.3 ModelSim 的使用

对设计进行仿真，需要给被测电路的输入信号加上激励。最直观的方法是用图形化的方法画出波形，如 Quartus，Maxplus II 支持该方法，但在 ModelSim 中不支持。ModelSim 中通过写测试台(即 Testbench)来加入激励。

此外，还可以以命令方式输入波形(用 Force 语句)。

1. RTL 仿真的流程 1 (工程方式)

1) 创建一个新的工程

创建一个新工程的步骤如下：

(1) 通过选择菜单 File→New→Project 命令来创建一个新工程。

(2) 弹出 Create Project 窗口，如图 1.76 所示。

图 1.76 Create Project 窗口

输入工程名 Project Name 和工程所在的目录 Project Location。在这个目录下将存放工程.mpf 文件和拷贝的源文件。默认的库名为“work”。ModelSim 用该命字在工程目录下创建一个子目录，即工作库。创建工程后，可以在主窗口的工作空间区域看到一个空的工程标号。

说明：

在编译一个源文件之前，需要一个设计库来放置编译的结果。在 ModelSim 中，设计库的名称是 work，如图 1.76 所示。这样就在当前目录下建立了一个名为 work 的子目录，即为你的设计库。这个子目录包括一个特殊的文件名为_info。

2) 为所创建的工程加入源文件

在这个例子中，包括两个 Verilog 源文件，每个源文件都包括一个 module。

文件 counter.v 包括一个名为 counter 的 module，实现了简单的 8 位二进制的计数器。另一个文件 tcounter.v，是测试台文件(test_counter)，用来验证计数器。仿真时由测试文件调用计数器。现在，需要将这两个文件编译到工作库中。

(1) 在 Project 标号区域单击右键，选择 Add file to Project，弹出将文件加入工程的对话框。也可以在主菜单选择 Project→Add file to Project，如图 1.77 所示。

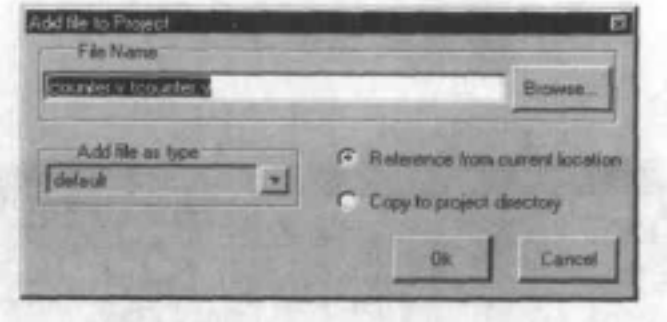


图 1.77 Add file to Project 窗口

(2) 选定一个或多个文件加入工程。这里我们以一个计数器为例，源文件包括计数器设计文件 counter.v 和测试台程序 tcounter.v。

(3) 对于所选定的文件，可以选择是否将它们拷贝到工程目录中(Reference from current location 还是 Copy to project directory)。

3) 编译文件

(1) 在 Project 区域中单击右键，选择 Compile All，也可以在主菜单上选择 Project→CompileAll，还可以单击快捷键，弹出如图 1.78 所示的窗口。

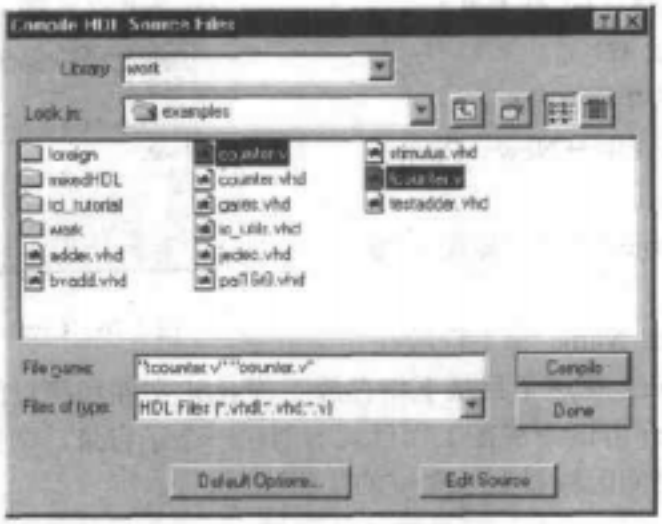


图 1.78 Compile HDL Source Files 窗口

在图中选择要编译的文件，单击 Compile 按钮。编译结束，单击 Done 按钮。

(2) 编译结束，单击 Library 标号，将看到两个已编译过的设计。

4) 对设计进行仿真

(1) 单击快捷键，弹出 Load Design 对话框，如图 1.79 所示。选择 test_counter，再

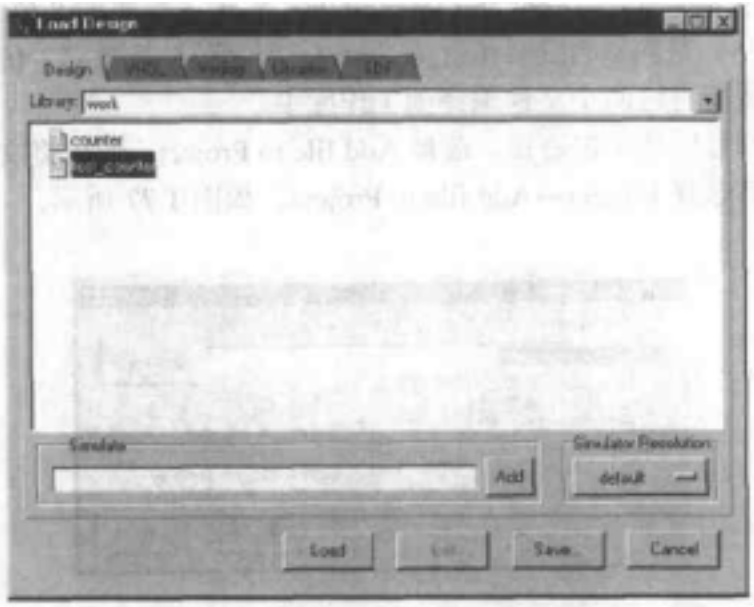


图 1.79 Load Design 窗口

单击 Load 按钮。在 Load Design 对话框中可以选择一个设计单元来进行仿真，还可以选择仿真的精度。默认的库是 work，默认的仿真精度为 1 ns。

(2) 单击菜单 View→Structure，可显示仿真结构；单击菜单 View→Signals，可弹出信号显示窗口；单击菜单 View→Wave，可弹出波形窗口。

(3) 在 Signals 窗口中，选择要显示的信号，单击菜单 View→Waves， 将信号加入到 Wave 窗口(Singlas in Region 可以将当前模块的信号加入到 Wave 窗口，Signals in Design 可以将该设计中的所有信号加入到 Wave 窗口)。

5) 运行仿真

接下来就可以运行仿真，观察波形了。

可以从工具条上运行不同的 Run 功能。在主窗口的工具条上选择 Run 按键，这样就运行了仿真，并在 100 ns(默认的仿真长度)后结束。也可以在提示符后键入 run 1 ms，表示运行 1 ms 的仿真。键入 run @ 3000，可以使仿真运行到 3000 ns 时停止。

如果在主窗口工具条上选择 Run All 按键，或在提示符后键入 RUN –ALL，或在主菜单上选择 Run→Run –All，这样仿真会一直进行下去。

可以选择 Break 来中断运行。

选择 Restart 按键，或在主菜单中选择 File→Restart，或在提示符后键入 Restart，可以重新载入设计，并将仿真时间复位为 0。

在 Wave 窗口中，选择菜单 File→Save Format，或者单击快捷键，可以将要仿真的信号保存起来(后缀为.do 或.tcl)。

如要存储波形，则退出 ModelSim，将 vsim.wlf 改一下名称，例如改成 save.wlf。下次运行 ModelSim 时，可以用 vsim→view saved.wlf 来载入此波形，也可以通过 wave 窗口的菜单 File→Open Dataset 来载入。

在 Wave 窗口中，可以将信号合并为总线或组。用 Edit→Combine 菜单，弹出 Combine Selected Signals 对话框。总线是信号的集合，用一定的顺序连接。

在 Source 窗口中，单击行标号，即在这一行设置了断点。在设置了断点的这一行有红点显示。仿真时，会停止在断点处。在行数上单击鼠标右键，选择 remove breakpoint，可以删除所设的断点。

注意：断点只能设置在可执行的语句上，这种语句的行标号为绿色。

按 Step 可以单步执行。

键入 quit –sim，结束本次仿真。键入 quit -force，退出仿真器。

ModelSim 还具有波形比较、代码覆盖率的测试等功能。详见帮助文件。

2．RTL 仿真的流程 2(非工程方式)

在 ModelSim 以前的版本中，并不支持工程的方式。许多用过 ModelSim 5.2 的工程师更习惯于非工程的方式。这里说明一下流程。

(1) 启动程序后，单击菜单 File→Change Directory，设置工作目录。

(2) 在主程序窗口中，键入 vlib work。

(3) 编译源文件，仿真与上述过程相同。这里略去。

3．**脚本文件**

为了提高效率，许多工程师喜欢用批处理的方式进行仿真，这就需要书写脚本文件。下面给出一个脚本文件的示范。

说明：

号跟的是注释；

VCOM 用于编译.VHD 文件；

Vlog 用于编译.v 文件；

Vsim 载入顶层模块；

View structure，显示仿真结构；

View signals 给出信号窗口。

```
############MODULE1#################3
vcom d:\\project\\myproject1\\xxxx.vhd -explicit
…
################MODULE2##############3
vlog d:\\project\\myproject1\\xxxx.v
…
vlog d:\\project\\myproject1\\ sim\\rtl_sim\\top.v
vsim work.top
view structure
view signals
```

4．**FPGA 的后仿真**

ModelSim具有后仿真的功能，以验证设计的时序是否符合要求。进行后仿真必须要有相应的后仿真库。ModelSim有SE版和OEM版之分，在SE版中不包括FPGA仿真库，OEM版则包括。OEM版主要有Xilinx(XE)和Altera(AE)之分。如果你用的是SE版本，就需要自己来加入相应的后仿真库。这里以Xilinx为例说明加入Xilinx仿真库的方法：

(1) 启动 ModelSim 5.5e 并新建工程，假设工程为 test。

(2) 新建一个库。单击菜单 Design→Create a New Library，在弹出的窗口中输入库名 unisim。

(3) 单击菜单 Design→Compile，或者单击快捷键，弹出编译选择窗口，如图 1.78 所示。

(4) 选择目录为 Xilinx 安装目录下的 unisims 子目录，选择相应的源文件进行编译(对 VHDL，相应文件在 Xilinx/VHDL/Src/unisim 下；对 Verilog，相应文件在 Xilinx/Verilog/Src /unisim 下，下面我们以 VHDL 的仿真库为例)。

(5) 选择源文件 unisim_VPKG.vhd 进行编译；然后选择源文件 unisim_VITAL.vhd 进行编译；再选择源文件 unisim_VCOMP.vhd 进行编译。在编译成功后选择 Done，回到 Modelsim 主窗口，再从 Design 菜单下单击 Browse Libraries…，可以看到 unisim 库已建好。

(6) 以同样方式建立新库 simprim、spartan2_macro 并编译，一定要注意在编译过程中文件的顺序。

(7) 在 Modelsim 的安装目录下(通常是 x:\modeltech_5.5e)创建 xilinx 目录，再在其下创

建 vhdl 目录，即 x:\modeltech_5.5e\xilinx\vhdl，然后将 test 工程所在目录下的三个子目录 unisim、simprim、spartan2_macro 拷贝至 vhdl 下。将 modeltech_5.5e 目录下的 modelsim.ini 的只读属性去掉，并在[library]区增加如下三行：

unisim = $MODEL_TECH/../xilinx/vhdl/unisim

simprim = $MODEL_TECH/../xilinx/vhdl/simprim

spartan2_macro = $MODEL_TECH/../xilinx/vhdl/spartan2_macro

这样就可以了。

加入后仿真库之后，在仿真时，单击菜单 Design→Load Design，选择 SDF，加入相应的.sdo 文件(由 ISE 生成)；单击 Librarys，加入所用器件；然后载入相应的设计就可以了。

5. ASIC 的门级仿真/后仿真流程

门级仿真与后仿真的差异在于：前者只有门的延时，后者还有连线延时，因而更接近真实的情况。这里以 ASIC 门级仿真为例进行说明，后仿真的方法与之相同，只是 SDF 文件不一样。

用 ModelSim 进行门级仿真时，需要将后仿真库加入。例如，如果用 charter 的库 CSM06 进行综合，则我们需要将该库与综合后的网表文件一块进行编译。

例如，一个设计用 mydesign1.v 来实现。综合后得到的网表文件是 mydesign_post.v。进行后仿真时，需要将 csm06.v 一起进行编译。在仿真批处理文件中，我们要加入：

vlog d:\\project\\myproject \\csm06.v

下面的语句要加入综合后的网表(替换相应的 RTL 文件)：

vlog d:\\project\\myproject \\mydesign_post.v

在测试台文件中，将相应的 SDF 文件标注到网表中，如下所示：

$sdf_annotate("mydesign.sdf",top.mydesign);

6. 其它操作

1) 打开/关闭/删除一个工程

当关闭 ModelSim 时，ModelSim 会记住最后一个打开的工程。可以通过选择主窗口中的 File→Open→Project 来打开一个现有的工程。

在主窗口中选择 File→Close→Project，可以将 Project 关闭，但 Library 和 Structure 仍然在工作空间内。

要删除一个工程(Delete a project)，可在主窗口中选择 File→Delete→Project。

2) 创建一个库

在 ModelSim 中，有资源库与设计库之分。设计库即 work。资源库包括 ieee，modelsim_lib，std，synopsys 等。当创建一个工程时，ModelSim 自动建立一个工作的设计库。如果没有建立一个工程，那么在编译以前就需要建立一个工作的设计库。这在上面几节中已经演示过。

建库的命令是 vlib 名称，也可以单击菜单 Design→Create a New Library，这样弹出一个对话框，就可以在其中定义库名和它的逻辑映射。如果选择 Create a new library and a logical mapping to it，则需要在 Library Name 内键入新的库的名字。这样就在当前的工作目录下建立了一个新库。

如果选择 Create a map to an existing library，则可以映射一个已有的库。在 Library Name

内键入新的库的名字，接着在 Library Maps to 框内键入需要映射的库名(可通过点击 Browse 来查找)。

1.4.4 NC Simulator 的使用

NC Simulator 是工作站上使用的仿真软件，功能非常强大。它的使用跟 ModelSim 相似。这里给出它的一个脚本文件。

说明：

set 用来设置变量。

ncvhdl 用来编译.vhd 文件。

ncvlog 用来编译.v 文件。

ncsim top -gui &：以图形方式启动 nc simulator，并调入顶层文件。

```
###note : project1 simulation enviroment .
###
###Author: xxx 2002,06,15
#############################################################################
source /LDV/ldv.csh
rm -r work
rm *log
rm -r *shm
rm *lib
rm *var
set code_path = "/net/wdx03/export/home/myproject/ mydesign/ src/ rtl"
...
echo " ************************************************************ "
echo "                          Welcome to use ncsimulator! "
echo " ************************************************************ "
rm -r wave.shm
set predefine = "TEST_INTERNAL_RAM"
ncvhdl   -linedebug   $code_path/xxx.vhd
...
ncvlog   -linedebug       $code_path/xxx.v
...
ncelab -access +wr top   -timescale 1ns/10ps
echo   "*******************3*************"
echo $DISPLAY
echo "************************************"
```

```
echo   "*********************************"
banner "    ok"
echo   "*********************************"
ncsim top -gui &
```

1.4.5 FPGA Express 的使用

启动 FPGA Express 程序。其菜单如图 1.80 所示。

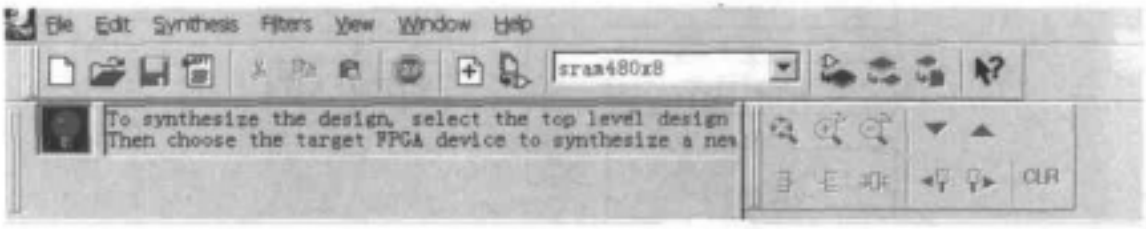


图 1.80 FPGA Express 的菜单

单击菜单 File→New Project，或者单击快捷键，来创建一个新工程。键入工程名，例如，取名 myfifo。

选中设计中的最顶层文件，然后单击快捷键，弹出如图 1.81 所示的对话框，在其中键入所需的器件类别和器件名称，选择是面向速度优化还是面向面积优化，选择时钟频率，选择综合的速度是快是慢(Effort 选 High，则速度慢一些，综合出的网表性能好一些)。

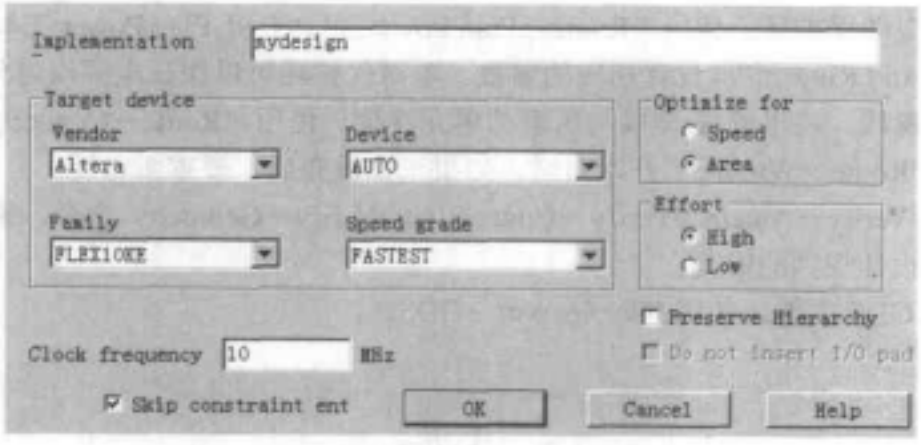


图 1.81 Implementation 对话框

然后选择，输出相应的网表文件。缺省输出为 EDIF 文件，也可以选择.v 或者.vhd。

1.4.6 Silicon Ensemble 的使用

针对基于标准单元的 ASIC，常用 Cadence 公司的 Silicon Ensemble(一般称为 SE)进行布局和布线。针对全定制的 ASIC，常用 LEDIT。

使用 SE 进行布局布线的步骤如下(其过程框图见图 1.82)：

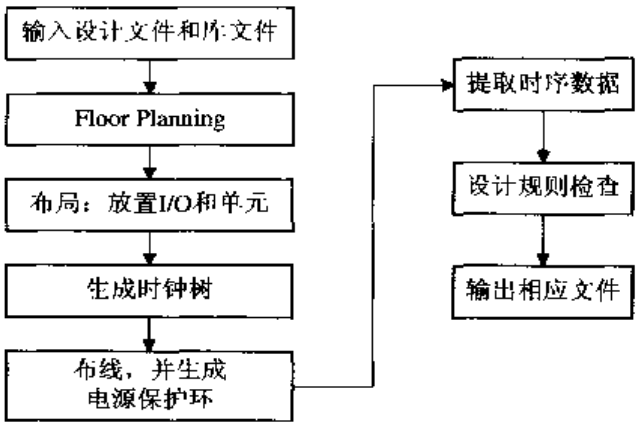


图 1.82　用 SE 进行布局布线的步骤

- 读入库文件，使用菜单“File→Import→LEF”。
- 读入综合后的网表文件，使用菜单“File→Import→Verilog”。
- 开始 floor planning，并设置相应的约束条件。使用菜单“Floor Plan→Initialize Floor Plan”。
- 放置 I/O 引脚，使用“Place Ios...”。
- 放置设计中的标准单元，使用“Place→cells...”。
- 增加电源保护环。使用“Route→Plan Power...”；打开 Plan Power 工具菜单后，单击“Add Rings...”，设置相应的参数。电源保护环可以保证电源均匀分配。
- 布电源线，将电源线/地线与所有的单元连接。使用“Route→Connect Rings...”。
- 单击"Route→Wroute..."开始布线。如果一次没布通，要重新布线。
- 使用 Verify→Antenna/Verify→Connectivity/Verify→Geometry 命令，来验证是否有违反设计规则的情况。
- 生成 GDS 文件，使用 File→export→GDS。

1.5　Verilog

1.5.1　Verilog 语言基础

前面介绍了芯片系统开发的过程。下面介绍 Verilog 和仿真的知识。

读者首先需要了解：Verilog **是面向仿真的语言**(VHDL 也是)，它提供了很强的仿真能力。许多初学者过多地关注于代码的“可综合性”，却忽略了这方面的学习，因而导致开发效率不高。这一章我们侧重于“仿真”，在第 3 章将论述“可综合的设计”。

Verilog具有算法级、门级到开关级描述的能力。随着水平的提高，设计者可能会越来越关注于它在体系结构级描述及其算法级行为上描述设计的能力。任何一本Verilog语法书对它的语言要素都有完整的论述。本书并不是一本语法书，我们不是从该语言标准的制定者的角度描述它，而是从应用者的角度来描述它，因而只对其中较常用的、较重要的东西进行说明。

怎样才能学好 Verilog？这里给出两点建议。

第一点，你并不需要把它的语法了解透彻。最有效的方法是从应用出发，多看多写，遇到不清楚的语法，再查语法书。语法书要有一本，但不应该作为教材。

第二点，要建立时间的概念，要记住，许多操作是并行的，并且信号的传递是有延时的。

1.5.2 基本概念

下面介绍一些 Verilog 的基本概念。

在Verilog HDL 语言中，标识符是区分大小写的。

1．时间单位

在Verilog HDL 模型中，所有时延都用单位时间表述，时延单位为s、ms、μs、ns、ps或fs。

`timescale编译器指令用于定义时延单位和时延精度。其指令格式为：

`timescale 时延单位/时延精度

例如：`timescale 1ns/100 ps，时延单位为1 ns，时延精度为100 ps。

`timescale 编译器指令在模块说明外部出现，并且影响后面所有的时延值。

在编译过程中，`timescale指令影响这一编译器指令后面所有模块中的时延值，直至遇到另一个`timescale指令或`resetall指令。

当一个设计中的多个模块带有自身的`timescale编译指令时，模拟器总是定位在所有模块的最小时延精度上，并且所有时延都相应地换算为最小时延精度。

2．数据类型

Verilog HDL 有两大类数据类型(见表1.8)。

说明：存储器赋值不能在一条赋值语句中完成，但是寄存器可以。因此在存储器被赋值时，需要定义一个索引。

下例说明它们之间的不同。

```
reg [1:5] Dig; //Dig为5位寄存器
...
Dig = 5'b11011;
```

上述赋值都是正确的，但下述赋值不正确：

```
reg Bog[1:5]; //Bog为5个1位寄存器的存储器
...
Bog = 5'b11011;
```

表 1.8　Verilog中的数据类型

数据类型	类别	描述及举例
线网类型(表示Verilog结构化元件间的物理连线。如果没有驱动元件连接到线网，则线网类型的变量缺省值为z)	wire	
	tri	三态线可以用于描述多个驱动源驱动同一根线的线网类型

寄存器类型(表示一个抽象的数据存储单元，它只能在always语句和initial语句中被赋值。寄存器类型的变量的缺省值为X)	reg	reg [msb:lsb] reg1, reg2, . . . regN;
		msb和lsb 定义了范围，并且均为常数值表达式。如果没有定义范围，缺省值为1位寄存器。寄存器可以取任意长度。寄存器中的值通常被解释为无符号数. 例如：
		reg [1:4] Comb;
		Comb = −2; //Comb 的值为14(1110)，1110是2的补码。
		Comb = 5; //Comb的值为15(0101)

	存储器	reg [msb:lsb] memory1 [upper1:lower1], memory2[upper2:lower2],... ; 例如： reg [0:3] MyMem[0:63] / /MyMem为64个4位寄存器的数组。 reg Bog[1:5] / /Bog为5个1位寄存器的数组。 MyMem和Bog都是存储器
	整数	整数寄存器包含整数值。整数寄存器可以作为普通寄存器使用，常用于行为建模

3．操作符

表 1.9 显示了所有操作符的优先级和名称。操作符从最高优先级(左边顶行)到最低优先级(右边底行)排列。同一行中的操作符优先级相同。

表 1.9　Verilog 中的操作符

优先级	名称	优先级	名称
+	一元加	>>	右移
−	一元减	<	小于
!	一元逻辑非	<=	小于等于
~	一元按位求反	>	大于
&	归约与	>=	大于等于
~&	归约与非	==	逻辑相等
^	归约异或	!=	逻辑不等
^~或~^	归约异或非	===	全等
|	归约或	!==	非全等
~|	归约或非	&	按位与
*	乘	^	按位异或
/	除	^~或~^	按位异或非
%	取模	|	按位或
+	二元加	&&	逻辑与
−	二元减	||	逻辑或
<<	左移	?:	条件操作符

除条件操作符从右向左关联外，其余所有操作符自左向右关联。例如：

表达式 A + B – C　等价于 (A + B) – C　//自左向右

而表达式 A ? B : C ? D : F　等价于 A ? B : (C ? D : F)　//从右向左

4．**基本单元 module**

module 是 Verilog HDL 语言中基本的模块，可以表示一个逻辑模块，如一个 CPU 设计中的 ALU 部分，也可以表示整个系统。每个 module 的描述从关键词 module 开始，有一个模块名(如 sfifo，DFF，ALU 等)，以关键词 endmodule 结束。

module 的端口在硬件中与管脚相对应，module 通过端口与外界通信。一个 module 可以有很多端口，都写在 module 名字后面的圆括号中。端口的类型必须在 module 的描述中声明，包括 input、output 或 inout (双向)。

将 module 实例化，并通过它们的端口相互连接，可以创建更大的组件。如图 1.83 所示，REG4 包括 module DEF 的四个实例。每个 module 都有它自己的实例名(d0, d1, d2, d3)。实例的名字给了每个 object 一个独特的名字，这样使每个实例的内部都能被检查到。每个实例都是完整的、独立的。

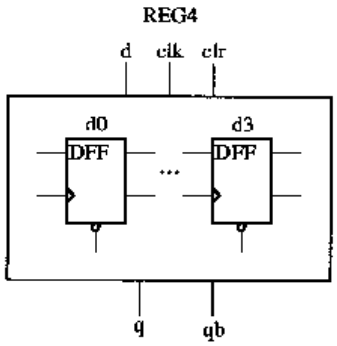


图 1.83　module 实例化

```
module DEF(d,clk,clr,q,qb);
……
endmodule

module     REG4(d,clk,clr,q,qb);
output[3:0] q,qb;
input[3:0] d;
input clk,clr;

DFF d0(d[0],clk,clr,q[0],qb[0]);
DFF d1(d[1],clk,clr,q[1],qb[1]);
DFF d0(d[2],clk,clr,q[2],qb[2]);
DFF d0(d[3],clk,clr,q[3],qb[3]);
endmodule
```

5．**赋值语句**

在Verilog中，要经常用到连续赋值与过程赋值。

1) 连续赋值

连续赋值语句的语法为:

assign [delay] a = b;

当 b 发生变化时，a 要重新计算，并且在指定的时延后，a 被赋值新值。如果没有定义时延，缺省时延为 0。

连续赋值语句是并发执行的，也就是说各语句的执行顺序与其在描述中出现的顺序无关。

2) 过程赋值

过程赋值出现在 initial 和 always 语句中。

initial 语句只执行一次。always 语句总是循环执行。一个模块中可以包含任意多个 initial 或 always 语句。这些语句的执行顺序与其在模块中的顺序无关。所有的 initial 和 always 语句在 0 时刻开始并行执行。

说明：

只有 register 类型数据才能在 initial 和 always 语句中被赋值。但 register 类型的数据并不一定综合出寄存器，也可能综合出纯组合逻辑。需要注意的是，**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。

在 initial 和 always 中，可以使用 begin...end，也可以使用 fork...join。前者是顺序语句块，其中所有的语句顺序执行；后者是并行语句块，其中所有的语句并行执行。

赋值语句分类见表 1.10。

表 1.10 赋值语句分类

过程赋值	连续赋值
在always语句或initial语句内出现	在一个模块内出现
赋值语句是按顺序执行的。 always @(A or B or C or D) begin reg Temp1,Temp2; Temp1=A&B; Temp2=C&D; Temp1=Temp1|Temp2; end	与其它语句并行执行，在右端操作数的值发生变化时执行
驱动寄存器	驱动线网
无assign关键词(在过程性连续赋值中除外)	有assign关键词

6. 描述时延

Verilog 提供的显式语言结构#，用以指定设计中的端口到端口的时延及路径时延。

例如：

```
module MUX2_1 (out, a, b,sel);
output out;
input a, b, sel;
    not #1 not1 (sel_, sel);
    and #2 and1 (a1, a, sel_);
    and #2 and2 (b1, b, sel);
    or #1 or1 (out, a1, b1);
endmodule
```

又例如：

```
Sum = (A ^ B) ^ Cin;
#4 T1 = A & Cin;
```

这两条语句规定：在第一条语句执行后等待 4 个时间单位，然后执行第二条语句。

再看下面的示例。

```
Sum = #3 (A ^ B) ^ Cin;
```

在这个语句中，首先计算右边表达式的值，等待 3 个时间单位，然后赋值给 Sum。如果在过程赋值中未定义时延，则赋值立即发生。

如果右端变化的间隔小于延时，会出现什么情形呢？请看下面的示例。

```
assign #4 Cab = Drm;
```

Cab 与 Drm 的变化如图 1.84 所示。在时刻 5 处的 Drm 的上升边沿，应该在时刻 9 就显示在 Cab 上，但是因为 Drm 在时刻 8 下降为 0，因而在 Cab 上的值被滤掉。同样，Drm 在时刻 18 和 20 之间的脉冲被滤掉。也就是说，如果右端值变化的间隔小于时延间隔，或者在时延间隔内右端值变化，则不能传输到输出。

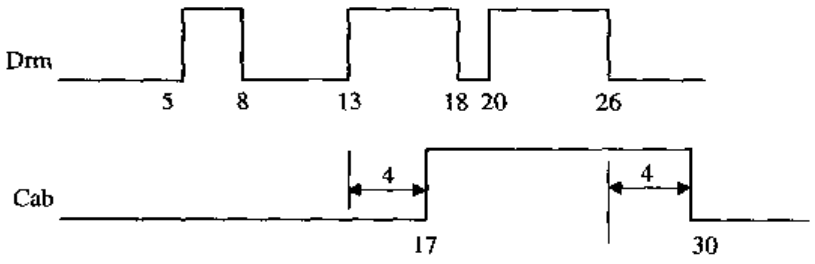


图 1.84　值变化快于时延间隔

1.5.3　设计仿真

作为一种“面向仿真”的语言，Verilog 不仅可以描述设计，还提供对激励、控制、存储响应和设计验证的建模能力。

激励和控制可用初始化语句产生。通过在初始化语句中自动与期望的响应值比较，可以验证设计正确与否。

测试验证程序有三个主要目的：

(1) 产生模拟激励(波形)。

(2) 将输入激励加入到测试模块并收集其输出响应。

(3) 将响应输出与期望值进行比较。

有两种产生激励值的主要方法：

(1) 产生波形，并在确定的离散时间间隔内加载激励。

(2) 根据模块状态产生激励，即根据模块的输出响应产生激励。

可用$readmemb 系统任务从文本文件中读取向量(可能包含输入激励和输出期望值)。而所有输入向量和观察到的输出结果也可以输出到文件中。

1．testbench 的模板

仿真最通用的方法是编写测试台程序，即 testbench。下面是一个 testbench 的一般形式：

```
module testbench1;                    //
//数据类型声明
//对 module 实例化
//施加激励
//显示结果
endmodule
```

2．加法器的仿真

我们以一个 32 位加法器为例，说明 testbench 的写法。这个 32 位加法器是由 8 个 4 位加法器构成的。这个例子很简单，但需要说明的是，代码中定义了 DEL 这个常量。这个常量不是必需的，只是为了防止出现那些只有仿真中才会出现的毛刺。在综合的时候，这个变量会被自动地忽略掉。

```
// DEFINES
`define DEL     1
// 顶层模块
module Adder(clk,a,b,reset_n,add_en,out,cout,valid);
//端口信号
input           clk;            //时钟
input [31:0]a;                  //32 位输入
input [31:0]    b;              //32 位输入
input           reset_n;        //低电平有效，同步复位
input           add_en;         //使能控制信号
output [31:0]   out;            //32 位输出
output              cout;       //进位输出
output              valid;      //是否正确
//类型说明
wire            clk;
```

```
wire [31:0]      a;
wire [31:0]      b;
wire             reset_n;
wire             add_en;
wire [31:0]      out;
wire             cout;
wire             valid;
//局部变量
wire [7:0]       cout4;          // 4 位加法器的进位输出
reg  [2:0]       valid_cnt;      //用于判别输出结果

assign # DEL cout = cout4[7];    //最高进位即输出进位
assign # DEL valid = ~|valid_cnt; //当 8 个 valid_cnt 都正确时，最终结果才正确

//这个 32 位加法器由 8 个 4 位加法器组成
Adder_4bit Add0(.clk(clk), .a(a[3:0]), .b(b[3:0]), .cin(1'b0), .reset_n(reset_n), .add_en(add_en),
    .out(out[3:0]), .cout(cout4[0]));

Adder_4bit Add1(.clk(clk), .a(a[7:4]), .b(b[7:4]), .cin(cout4[0]), .reset_n(reset_n),
    .add_en(add_en), out(out[7:4]), .cout(cout4[1]));

Adder_4bit Add2(.clk(clk), .a(a[11:8]), .b(b[11:8]), .cin(cout4[1]), .reset_n(reset_n),
    .add_en(add_en), .out(out[11:8]), .cout(cout4[2]));

Adder_4bit Add3(.clk(clk), .a(a[15:12]), .b(b[15:12]), .cin(cout4[2]), .reset_n(reset_n),
    add_en(add_en),. out(out[15:12]), .cout(cout4[3]));

Adder_4bit Add4(.clk(clk), .a(a[19:16]), .b(b[19:16]), .cin(cout4[3]), .reset_n(reset_n),
    .add_en(add_en), .out(out[19:16]), .cout(cout4[4]));

Adder_4bit Add5(.clk(clk), .a(a[23:20]), .b(b[23:20]), .cin(cout4[4]), .reset_n(reset_n),
    .add_en(add_en), .out(out[23:20]), .cout(cout4[5]));

Adder_4bit Add6(.clk(clk), .a(a[27:24]), .b(b[27:24]), .cin(cout4[5]), .reset_n(reset_n),
    .add_en(add_en), .out(out[27:24]), .cout(cout4[6]));

Adder_4bit Add7(.clk(clk), .a(a[31:28]), .b(b[31:28]). .cin(cout4[6]), .reset_n(reset_n),
    .add_en(add_en), .out(out[31:28]), .cout(cout4[7]));
```

```
always @(posedge clk) begin    //这里的加法器是同步复位的；如果写成
                               //always @(negedge reset_n or posedge clk)
                               //   if(~reset_n)
                               //.....
                               //的形式，则是一个异步复位加法器

if (~reset_n) begin
      //初始化
      valid_cnt <= #DEL 3'h0;
end
  else if (((valid_cnt == 3'h0) && (add_en == 1'b1)) ||
        (valid_cnt != 3'h0)) begin
      valid_cnt <= #DEL valid_cnt + 1;
  end
end
endmodule      //Adder

//子模块
module    Adder_4bit(    clk,  a,    b,    reset_n,    add_en,    cin,  out,  cout);

//端口信号
input             clk;          // Clock
input [3:0]       a;            // 4-bit A input
input [3:0]       b;            // 4-bit B input
input             cin;          // Carry in
input             reset_n;      // Active low, synchronous reset
input             add_en;       // Synchronous add enable control

output [3:0]      out;          // 4-bit output
output            cout;         // Carry output

wire              clk;
wire [3:0]        a;
wire [3:0]        b;
wire              cin;
wire              reset_n;
wire              add_en;
reg   [3:0]       out;
reg               cout;
```

```
always @(posedge clk) begin
    if (~reset_n) begin
        {cout,out} <= #DEL 33'h00000000;
    end
    else if (add_en) begin
        {cout,out} <= #DEL a+b+cin;
    end
end
endmodule              //Adder_4bit
```

编写 testbench 是系统设计很重要的一个方面。大家可以看到，这个 testbench 要比代码长许多。

```
`define TEST_NUM 255      //要测试的加法操作的数目
`define BITS 32           //操作数位宽
`define EBITS `BITS+1     // Bit width of data plus carry bit

module add_sim();
reg                       clock;
reg                       reset_n;
reg                       add_en;
reg   [`BITS-1:0]a_in;
reg   [`BITS-1:0]b_in;
wire [`BITS-1:0] sum_out;
wire                  overflow;
wire                  valid;

integer                   cycle_count;    //对有效时钟周期进行计数
integer                   val_count;      //对有效数据之间的周期进行计数
reg   [`EBITS-1:0]    expect;             //所期望的输出结果

Adder adder1(.clk(clock),    .a(a_in),        .b(b_in),        .reset_n(reset_n),
    .add_en(add_en),     .out(sum_out),       .cout(overflow),     .valid(valid));
//输入信号的初始化
initial begin
    clock = 0;
    reset_n = 0;
    add_en = 1;
    cycle_count = 0;
```

```
        val_count = 0;
        $random(0);              // 初始化随机数发生器
end

//产生时钟
always #100 clock = ~clock;

always @(negedge clock) begin
        if (valid === 1'b1) begin
                case (cycle_count)
                        0:      begin
                                if ((sum_out === `BITS'h00000000) &&
                                        (overflow === 1'b0))
                                        $display ("Reset is working");
                                else begin
                                        $display("\nERROR at time %0t:", $time);
                                        $display("Reset is not working");
                                        $display("      sum_out = %h", sum_out);
                                        $display("      overflow = %b\n", overflow);
                                        $stop;
                                end
                                //测试有效输出信号
                                if (valid === 1'b1)
                                        $display ("Valid signal is working");
                                else begin
                                        $display("\nERROR at time %0t:", $time);
                                        $display("Valid signal is not working");
                                        $display("      valid = %b\n", valid);

                                        $stop;
                                end

                                reset_n = 1;
                                        //以随机数作为加法器的两个输入
                                a_in = {$random} % (`EBITS'h1 << `BITS);
                                b_in = {$random} % (`EBITS'h1 << `BITS);
                                expect = a_in + b_in;

                                add_en = 0;
```

```
        val_count = 0;
    end
    1:  begin
        if (val_count !== 1'b0) begin
            $display("\nERROR at time %0t:", $time);
            $display("Valid is not held after reset\n");
            $stop;
        end

        if ((sum_out === `BITS'h00000000) &&
                (overflow === 1'b0))
            $display ("Reset is working");
        else begin
            $display("\nERROR at time %0t:", $time);
            $display("Reset is not working");
            $display("      sum_out = %h", sum_out);
            $display("      overflow = %b\n", overflow);
            $stop;
        end
        if (valid === 1'b1)
            $display ("Valid signal is working");
        else begin
            $display("\nERROR at time %0t:", $time);
            $display("Valid signal is not working");
            $display("      valid = %b\n", valid);
            $stop;
        end

        //开始加法操作
        add_en = 1;
        val_count = 0;
    end
    `TEST_NUM+1: begin
        // Check the result for correctness
        if ({overflow, sum_out} !== expect) begin
            $display("\nERROR at time %0t:", $time);

            $display("Adder is not working");
```

```
            $display("      a_in = %h", a_in);
            $display("      b_in = %h", b_in);
            $display("      expected result = %h",    expect);
            $display("      actual output = %h\n",    {overflow, sum_out});
            $stop;
        end

        a_in = {$random} % (`EBITS'h1 << `BITS);
        b_in = {$random} % (`EBITS'h1 << `BITS);

        add_en = 0;
        val_count = 0;
    end
    `TEST_NUM+2: begin
            if (val_count !== 0) begin
            $display("\nERROR at time %0t:", $time);
            $display("Valid is not held\n");
            $stop;
        end
            if ({overflow, sum_out} !== expect) begin
            $display("\nERROR at time %0t:", $time);
            $display("Adder is not working");
            $display("      a_in = %h", a_in);
            $display("      b_in = %h", b_in);
            $display("      expected result = %h", expect);
            $display("      actual output = %h\n",    {overflow, sum_out});
            $stop;
        end

        $display("\nSimulation complete - no errors\n");
        $finish;
    end
    default: begin
            if ({overflow, sum_out} !== expect) begin
            $display("\nERROR at time %0t:", $time);
            $display("Adder is not working");
            $display("      a_in = %h", a_in);
            $display("      b_in = %h", b_in);
            $display("      expected result = %h",    expect);
```

```
                $display("      actual output = %h\n",    {overflow, sum_out});
                $stop;
            end
            a_in = {$random} % (`EBITS'h1 << `BITS);
            b_in = {$random} % (`EBITS'h1 << `BITS);
            expect = a_in + b_in;
            add_en = 1;
            val_count = 0;
        end
    endcase
    cycle_count = cycle_count + 1;
end
else begin
    val_count = val_count + 1;
    if (val_count > 11) begin
        $display("\nERROR at time %0t:", $time);
        $display("Too many cycles for valid data\n");
        $stop;
    end
end
end
endmodule       //
```

1.5.4 系统任务及函数

Verilog 语言中一种最有效的仿真方法就是将一段代码封闭起来形成任务(Task)或函数(Function)结构。

任务和函数之间有几点差异：

(1) 一个任务块可以含有时间控制结构，而函数块则没有。也就是说，函数块从零仿真时刻开始运行，结束后立即返回(实质上是组合功能)。而任务块在继续下面的运行过程之前，其初始代码必须保持到任务全部执行结束或是失效。

(2) 一个任务块可以有输入和输出，而一个函数块至少有一个输入，没有任何输出，函数结构通过自身的名字返回结果。

(3) 任务块的引发是通过一条语句发生的，而函数块只有当它被引用在一个表达式中时才会生效。例如：

"tsk (out, in1, in2);"　则调用了任务结构，名为 tsk

而　"I = func (a, b, c);" 或 "assign x = func(Y);" 则调用了一个函数，名为 func

下面是一个任务模块的例子：

```
task tsk;
    input i1, i2;
    output o1, o2;
    $display ("task tsk, i1 = %b, i2 = %b", i1, i2);
    #1 o1 = i1 & i2;
    #1 o2 = i1 | i2;
endmodule
```

函数块举例如下：

```
function[7:0] func;
    input i1;
    integer i;
    reg[7:0] rg;
    begin
        rg = 1;
        for (i = 1; i <= i1; i = i+1)
        rg = rg+1;
        func = rg;
    end
endfunction
```

函数块在编组代码以及增强其可读性和可维护性方面是一种十分重要的工具。

Verilog 提供了一系列的系统任务和函数。它包括如下几类：

(1) 显示任务(display task)；
(2) 文件输入/输出任务(File I/O task)；
(3) 时间标度任务(timescale task)；
(4) 仿真控制任务(simulation control task)；
(5) 时序验证任务(timing check task)；
(6) PLA 建模任务(PLA modeling task)；
(7) 随机数分析任务(stochastic analysis task)；
(8) 实数变换任务(conversion functions for real)；
(9) 概率分布函数(probabilistic distribution function)。

表 1.11 中为 Verilog 中的系统任务和函数，“$”符号表示 Verilog 系统任务和函数。

利用这些系统任务和函数，Verilog 能够监控模拟验证的执行，即模拟验证执行过程中设计的值能够被监控和显示。这些值也能够用于与期望值比较，在不匹配的情况下，打印报告消息。

表 1.11　Verilog 中的系统任务和函数

类别	关键字	描述及举例
显示任务(显示系统任务用于信息显示和输出。这些系统任务可进一步分为：显示和写入任务；探测监控任务；连续监控任务)	$display	将特定信息输出到标准输出设备 例如： $display ("simulation time is %t",$time);
	$displayb	同上，但输出格式为二进制
	$displayh	同上，但输出格式为十六进制
	$displayo	同上，但输出格式为八进制

	$monitor	连续监控指定的参数。当被监视的参数值发生变化时，整个参数表就在时间步结束时显示。例如：
		$monitor("At %t,D=%d,clk=%d",
		$time,D,Clk, "and Q is %b",Q);
		下面给出该系统任务的一个应用实例：
		module testfixture;
		reg a, b, sel;
		//复用器的例化
		MUX2_1 mux (out, a, b, sel);
		//加激励
		initial
		begin
		a = 0; b = 1; sel = 0;
		#5 b = 0;
		#5 b = 1; sel = 1;
		#5 a = 1;
		#5 $finish;
		end
		//显示结果
		initial
		$monitor ($time,"out = %b a = %b b = %b sel = %b", out, a,b, sel);
		endmodule

	$monitorb	同上，但输出格式为二进制
	$monitorh	同上，但输出格式为十六进制
	$monitoro	同上，但输出格式为八进制
	$monitoroff	禁止所有监控任务 $monitoroff
	$monitoron	启动所有监控任务 $monitoron
	$strobe	在指定的时间内显示仿真数据。在特定的时间步结束时才显示仿真数据。例如： forever @ (negedge clock) $strobe ("at time %d, data is %h", $time, data); 在这个例子中，$strobe 将在时钟的每个下降沿，把时间和数据信息写入标准输出和 log 文件
	$strobeb	同上，但输出格式为二进制
	$strobeh	同上，但输出格式为十六进制

续表

类别	关键字	描述及举例
	$strobeo	同上，但输出格式为八进制
	$write	例如： $write ("simulation is off at %t\n",$time);
	$writeb	同上，但输出格式为二进制
	$writeh	同上，但输出格式为十六进制
	$writeo	同上，但输出格式为八进制
文件输入/输出任务(Verilog提供了强有力的文件读写能力)	$fopen	打开一个文件。例如， integer f1 = $fopen("result.vec");
	$fclose	关闭一个文件。例如， $fclos(f1);
	$fdisplay	向文件中输出相应的信息。例如， integer f1 = $fopen("result.vec"); $fdisplay(f1, "the simulation time is %t",$time); 往 result.vec 文件中写入 "the simulation time is xx"
	$fdisplayb	同上，但输出格式为二进制
	$fdisplayh	同上，但输出格式为十六进制
	$fdisplayo	同上，但输出格式为八进制
	$fmonitor	监控指定参数的变化，将参数表写入到文件中
	$fmonitorb	同上，但输出格式为二进制
	$fmonitorh	同上，但输出格式为十六进制
	$fmonitoro	同上，但输出格式为八进制
	$readmemb	以二进制方式从文件中读取数据。每个数字由空格隔离。每个读取的数字被指派到存储器内的一个地址 reg [0:3] mem1[0:63]; $readmemb("my vec",mem1);
	$readmemh	以十六进制方式从文件中读取数据
	$fstrobe	在指定时间探测数据，并将仿真数据写入到文件中
	$fstrobeb	同上，但输出格式为二进制
	$fstrobeh	同上，但输出格式为十六进制
	$fstrobeo	同上，但输出格式为八进制
	$fwrite	将特定信息写入到文件
	$fwriteb	同上，但输出格式为二进制
	$fwriteh	同上，但输出格式为十六进制
	$fwriteo	同上，但输出格式为八进制

续表

类别	关键字	描述及举例
timescale 任务	$printtimescale	给出指定模块的时间单位和精度。如果没有指定参数，则输出包含该任务调用的模块的时间单位和精度。例如： $printtimescale;
	$timeformat	定义如何报告时间信息。该任务的参数见相关帮助。例如： $timeformat(-4,3, "ps",5); $display ("current simulation time is %t",$time);
仿真控制任务	$finish	退出仿真器，并将控制返回到操作系统。例如： $finish;
	$stop	将仿真挂起。这时候，可以在仿真器中发送交互命令
时序检查任务	$hold	报告数据保持时序冲突。例如： $hold(posedge clk,D,0.1); 如果 D-posedge clk<0.1，即保持时间不够，则报告时序冲突
	$nochange	如果在指定的基准事件区间发生数据变化，则报告时序冲突错误。基准事件必须是边沿触发事件。例如： $nochange(negedge clear,preset,0,0); 如果在 clear 为低时 preset 发生变化，则报告时序冲突错误
	$period	检查信号的周期。例如： $period(reference_event,limit); 如果(数据事件的时刻 - 基准事件的时刻)< limit，则报告时序错误
	$recovery	检查时序状态元件(触发器，锁存器，RAM 和 ROM 等)的时钟信号与相应的置/复位信号之间时序约束关系。例如： $recovery(reference_event,data_event,limit); 如果(数据事件的时刻 - 基准事件的时刻)< limit，则报告时序错误。基准事件必须是边沿触发事件
	$setup	报告时序冲突。例如： $setup(D,posedge ck,1.0); 如果(信号 ck 上升沿的时刻 - 数据 D 的变化时刻)<1.0，即建立时间不够，则报告时序冲突
	$setuphold	是系统任务$setup 的$hold 的结合
	$skew	检查信号之间(尤其是成组的时钟控制信号之间)的偏斜是否满足要求。例如： $skew(reference_event,data_event,limit); 如果(数据事件的时刻 - 基准事件的时刻)> limit，则报告信号之间出现时序偏斜太大的错误
	$width	检查信号的脉冲宽度限制。例如： $width(negedge ck,1.0,0); 如果 0<数据事件的时刻 - ck 下降沿的时刻<1.0，则报告信号上出现脉冲宽度不够宽的时序错误。数据事件来源于基准事件，是带有相反边沿的基准事件

续表

类别	关键字	描述及举例
随机数分析任务	$q_initialize	略
	$q_add	略
	$q_remove	略
	$q_full	略
	$q_exam	略

	$random	根据变量的取值按 32 位的有符号整数形式返回一个随机数。变量(必须是寄存器，整数或时间寄存器类型)控制函数的返回值，如果没有指定变量，则会根据缺省的变量产生随机数。注意，数字产生的顺序是伪随机排序的，即对于同一个值产生相同的数字序列。例如：
		`$random(12);`
		返回一个 32 位的有符号整型随机数。又例如：
		`reg [31:0] rand;`
		`rand = $random%60;`
		返回一个小于 60 的随机数

仿真控制任务	$realtime	向调用它的模块返回实型仿真时间。下面的例子显示了$realtime 的用法：
		`'timescale 10ns/1ns`
		`module test;`
		`reg set;`
		`parameter p = 1.55;`
		`initial`
		`begin`
		`$monitor ($realtime, , "set = %b", set);`
		`#p set = 0;`
		`#p set = 1;`
		`end`
		`endmodule`
		以下是上面例子的结果：
		`0 set=x`
		`1.6 set=0`
		`3.2 set=1`
		在这个例子中，set 的值是 10ns 的倍数，因为 10ns 是这个模块的时间单位。它们是实数，因为$realtime 返回一个实数

	$stime	向调用它的模块返回 64 位的整型仿真时间
	$time	向调用它的模块返回当前仿真时间(整型)。按模块时间单位比例返回值，并且被四舍五入
实数变换任务	$realtobits	将实数变换为 64 位的实数向量表示法
	$bitstroreal	将位模式变换为实数(与$realtobits 相反)
	$itor	将整数变换为实数
	$rtoi	通过截断小数值将实数变换为整数

续表

类别	关键字	描述及举例
随机数分配函数	$dist_chi_square	略
	$dist_erlang	略
	$dist_exponential	根据指数分布函数产生伪随机数
	$dist_normal	根据正态分布函数产生伪随机数
	$dist_poisson	根据泊松分布函数产生伪随机数
	$dist_t	略
	$dist_uniform	根据均匀分布函数产生伪随机数
PLA 建模任务	略去	

1. 用于格式定义的符号

格式定义的符号：

%h 或%H：十六进制

%d 或%D：十进制

%o 或%O：八进制

%b 或%B：二进制

%c 或%C：ASCII 字符

%v 或%V：线网信号长度

%m 或%M：层次名

%s 或%S：字符串

%t 或%T：当前时间格式

下面用一个例子来说明这些格式符号的用法。读者可以运行一下，看看输出结果。

```
module disp;
reg [31:0] rval;
pulldown (pd);
initial begin
    rval = 101;
    $display ("rval = %h hex %d decimal", rval, rval);
    $display ("rval = %o octal\nrval = %b bin", 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
```

2. **输出特殊字符的符号**

特殊字符的符号：

\n：换行

\t：制表符

\\：字符\

\"：字符"

%%：字符%

1.5.5 其它重要的内容

1. Specify

在 Verilog 中，一个模块的输入与输出之间的延时用 Specify 来描述。

Verilog 语言可以对模块中某一指定的路径进行延迟定义，这一路径连接模块的输入端口(或双向端口)与输出端口(或双向端口)。延迟定义块在一个独立的块结构中定义模块的时序部分，这样功能验证就可以与时序验证相独立。这是时序驱动设计的关键部分，因为包含时序信息的这部分程序在不同的抽象层次上可以保持不变。

一般用 Specify 来完成如下功能：

- 描述模块中的不同路径并给这些路径赋值；
- 描述时序核对，以确认硬件设备的时序约束是否能得到满足。

延迟定义块的内容要放在关键字 Specify 和 Endspecify 之间，而且必须在某一模块内部。在定义块中还可以使用 Specparam 关键字定义参数。举例说明如下：

一个简单电路的结构如图 1.85 所示。

进行路径延迟定义如下：

```
module noror (o, a, b, c);
  output o;
  input a, b, c;
    nor n1 (net1, a, b);
    or o1 (o, c, net1);
  specify
    (a => o) = 2;
    (b => o) = 3;
    (c => o) = 1;
  endspecify
endmodule
```

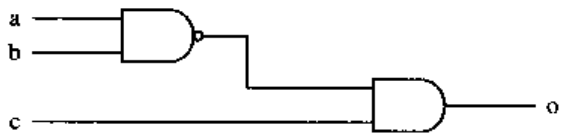


图 1.85 一个简单电路的示例结构

对于这一简单电路的延迟定义，可以采用将所有的延迟集中在最后一个或门上定义的方法，该方法简单但不精确。另一种方法就是如上述模块所做，把延迟分布在每个门上，即定义了从 a 点到 o 点的延迟为 2，从 b 点到 o 点的延迟为 3，从 c 点到 o 点延迟时间为 1，这种做法比前者精确，但工作量大。

2. SDF 标注

SDF(Standard Delay Format)文件可以在布图前得到(不包括连续延时)，也可以在布图后得到(包括连线延时)。它包含了如下的时序信息：

- 模块中组件、IO 端口的延时；
- 时序限制；
- 工艺参数、环境参数和用户自定义的参数等。

工具软件从 HDL 网表中读出连接信息，并从时序库中读出时序信息，然后将延时写到 SDF 文件中，并将 SDF 文件送给 SDF 标注器(SDF annotator)。SDF 标注器使用 PLI 接口将时序信息标注到 Verilog 设计中。

$sdf_annotate 的语法如下：

```
$sdf_annotate
    (["<sdffile>"],[<instance>],[<config_file>",["<log_file>"],
    ["<mtm_spec>"],["<scale_factor>"],[ "<scale_type>"]);
```

这些参数的含义见 ModelSim 或 NC simulator 的相关帮助。这里给出应用此系统任务的一个例子：

```
$sdf_annotate("my.sdf",top.top_unit.myunit);
```

3. 源代码的保护

在许多时候，需要对源代码进行保护。在 Verilog-XL 和 NC simulator 中支持此功能，但在 ModelSim 中不支持。

说明：下列语法在 Verilog-XL 和 NC simulator 中支持，但在 ModelSim 中不支持。

```
`accelerate
`autoexpand_vectornets
`desable_portfaults
`enable_portfaults
`endprotect
`expand_vectornet
`noaccelerate
`noexpand_vectornets
`noremove_tatenames
`noremove_netnames
`nosuppress_faults
`protect
`remove_gatenames
`remove_netnames
`suppress_faults
```

保护源代码有两种方法：一种是只对所选的模块进行保护；另一种是对所有模块进行保护。

1) *第一种方法*

在需要保护的源代码的前面加上`protect；而在源代码的后面加上`endprotect。用命令行编译源文件时，加上“+protect”选项。这样，就可以得到新的源文件，它的被保护区域是不可读的(乱码)。

下面是一个例子。在顶层模块中，对一部分代码进行保护。

```
module top_design(a,b,c);
bottom inst();
`protect
    initial begin
    $display("inside module top_design");
    ...
    end
    `endprotect
  ...
  endmodule
```

在编译时，我们可以用如下命令：

```
verilog mydesign.v +protect .myext
```

该命令将 mydesign.v 文件转换为 mydesign.v.myext。

2) *第二种方法*

这种方法不需要在源文件中加上`protect 语句，只需要编译时加上“+autoprotect”选项就可以了。如下所示：

```
verilog mydesign.v +autoprotect .protall
```

该命令生成 mydesign.v.protall 文件，设计中所有的模块都被保护。

4. `define

`define 在编译时直接提供一个简单的文本替代功能。其语法为：

```
`define <name> <macro_text>
```

`<name>将在编译时代替<macro_text>。用这种方法，能改善设计的可读性。

```
`define not_delay #1          //not_delay 的定义
`define and_delay #2
`define or_delay #1
module MUX_1 (out, a, b, sel);
output out;
input a, b, sel;
    not `not_delay not1(sel_, sel);          //not_delay 的使用
    and `and_delay and1(a1, a, sel_);
    and `and_delay and2(b1, b, sel);
    or `or_delay or1(out, a1, b1);
endmodule
```

5．翻转率的测试

在进行门级仿真和后仿真的时候，需要进行翻转率的测试，以保证能完整地遍历系统的功能。下面以 NC simulator 为例，说明翻转率测试的方法。

在 testbench 中，增加下述语句：

```
$toggle_test("design1_unit", "design2_unit", ..."designN_unit", "1");
initial
begin
  #1010;
  forever begin
  #10000;
        $toggle_test_summary;
  end
end

initial
begin
  #2000000;
        $save_toggle("toggle_rate1.txt");
end
```

这样，进行仿真的时候，仿真工具会自动测试电路中各结点的翻转情况(1→0 是一种翻转，0→1 是一种翻转)，并将结果写入到 toggle_rate.txt 中。在该文本文件中，记录了设计中总的结点数和已翻转的结点数。

一般说来，翻转率要达到 98%以上。

1.6 练　　习

1. 将 1.2.4 节讲的同步 FIFO 用 ModelSim 进行仿真，用综合工具综合到 Altera Flex10k 器件中，并对照 RTL 代码，分析综合出的电路，并进行后仿真。

2. 从网上下载一个 IC 产品的 datasheet，进行分析。

3. 描述 4 位加法、减法器的行为模型，写出相应的测试程序。在测试程序中描述所有输入激励和期望的输出值，将输入激励、期望的输出结果和监控的输出结果转存到文本文件中。

第2章 时序电路的设计

本章讲述了时序电路设计中的一些重要概念，包括基本时序电路、时钟策略、总线和数据系统设计中常见的时序要求。

2.1 时序逻辑电路

我们要设计的大多数电路都是时序电路。时序电路中，输出不仅跟当前的输入值有关，而且跟以前的输入值有关(如果仅与以前的输入有关，而与当前输入无关，则称为 Moore 电路)。

时序电路可分为同步时序电路和异步时序电路两类。

同步时序电路可用状态转换图和状态转换表来描述。在同步时序电路中，有一个时钟信号，电路的状态是每个时钟变化一次(通常，外部输入信号的变化也与时钟同步)。因此，同步时序电路的分析与设计要比异步时序电路简单。该方法构造的电路很健壮，并且容易维护，因此应用很广泛；缺点是时钟偏移带来的问题很难处理，并且电路的性能达不到最优。

在异步时序电路中，没有统一的时钟脉冲信号，状态变化的时刻是不固定的。通常，输入信号只在电路处于稳定状态时才发生变化。也就是说，一个时刻仅允许一个输入发生变化，以避免输入信号之间造成的竞争冒险。在设计和使用异步时序电路时，必须考虑到器件的传输延时，注意竞争冒险问题。同步时序电路由于有时钟的节拍控制，所以一般情况下不会出现竞争冒险。但必须要注意由于时钟偏移而造成的冒险。

时序电路分为两类：Mealy 型和 Moore 型。在 Mealy 型电路中，输出与输入和电路状态都有关；而在 Moore 电路中，某时刻的输出仅与该时刻电路的状态有关，与输入无关。一般而言，Mealy 电路的状态数目较少，相应的记忆元件的数目较少；而 Moore 型电路中组合逻辑电路比较简单。

时序电路要有存储的功能。在一个电路中，有两种方法来实现存储。第一种方法是利用正反馈，一个或更多的输出信号与输入相连接。这一类电路称为多频振荡器电路，其中双稳态元件用得最多(触发器就是一种双稳态元件)，单稳态及无稳态电路也用得非常普遍。另一种方法是利用存储的电荷来存储信号值。因为电荷总是要泄漏的，因此需要对电路不断刷新，此类电路称为动态电路。

2.1.1 双稳态电路

双稳态电路是指一个双稳态电路有两个稳定状态。如果不加任何激励，则电路保持在一个稳定状态(假设一直有电源供电)，因而可以记住特定的数值。要改变电路的状态，必须施加一个激励脉冲。

最常见的双稳态电路是触发器。

触发器种类很多，最简单的是 SR 触发器(也称为置位/复位触发器)。

对 SR 触发器进行改造，可得到 JK 触发器。JK 触发器没有禁止状态(SR 触发器中有)，但有翻转状态。

由 JK 触发器可以导出许多常用的触发器，例如 T 触发器与 D 触发器。

2.1.2 单稳态电路

单稳态电路只有一个稳定状态，并且经常处于此稳定状态下。在外加触发脉冲的激励下，单稳态电路进入暂时稳定的状态。经过一段固定的时间，电路自动返回到原来的稳定状态，从而输出宽度和幅度都固定的矩形脉冲。单稳态电路又称为单稳态多谐振荡器。单稳态电路可作为脉冲发生器。它的另一个应用是作为地址变换探测(ATD)电路，用于在静态存储器中生成时序(此电路探测地址总线的变化，并产生一个脉冲以初始化后续的电路)。

2.1.3 无稳态电路

无稳态电路是指没有稳定状态的电路，其输出在两个亚稳态之间来来回回地振荡，周期由电路参数决定，可作为晶体振荡器，生成芯片上的时钟。

2.1.4 施密特触发器

施密特触发器是一种正反馈电路，又称为鉴幅器，它的直流特性具有类似于磁滞的现象。无论输入信号的波形如何，该电路均输出矩形脉冲。当输入电压超过鉴幅电压时，输出为高电平；当输入电压低于鉴幅电压时，输出为低电平。

施密特触发器不仅可以鉴幅，还可以进行整形，去除信号中的噪声，这使得它在噪声环境中非常有用。施密特触发器的一个主要用途是将一个噪声干扰的或缓慢变化的输入信号转变为一个无噪声的数字输出信号。同时，由于输出信号变陡，抑制了直流通路，因而减少了功耗。

2.2 时 钟 策 略

我们在第一章已经提到过，在芯片设计的早期，就需要决定设计所采用的时钟策略。在一个复杂电路中，如何使不同的操作同步，以使它能在各种条件下都可靠工作，是每一个设计者必须面对的严峻问题。选择正确的时钟结构对电路的功能和性能都有影响。

设计者可以考虑多种时钟方案：可以使用**一对双相非交迭时钟**，可以使用**单一时钟**，也可以使用**多个时钟相**。

多相方案的优点在于能更有效地利用时间，因而使电路的性能最优；缺点是需要将这些时钟进行分配以避免竞争冒险，随着芯片复杂度及尺寸的增加，这个问题变得非常棘手。所以，在当前高性能的 IC 设计中(包括在高性能的微处理器中)，一般趋势是使用简单的时钟方案，即使是牺牲性能也在所不惜。

在一个系统中，各个组件所用的时钟可能不同。分配时钟有两种方法：

(1) 时钟树：由树型结构的时钟线驱动各时钟域。图 2.1 给出了一种 H 树型时钟分配网络。

(2) 主时钟：由主时钟直接驱动各时钟域。如果驱动能力不够，则增加缓冲，如图 2.2 所示。

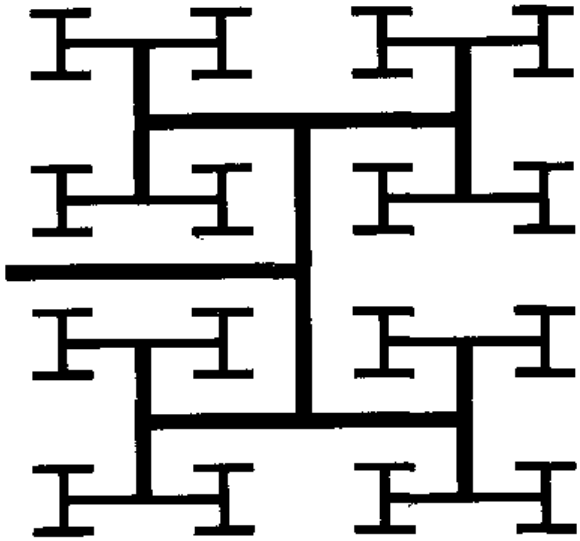


图 2.1　H 树时钟分配网络

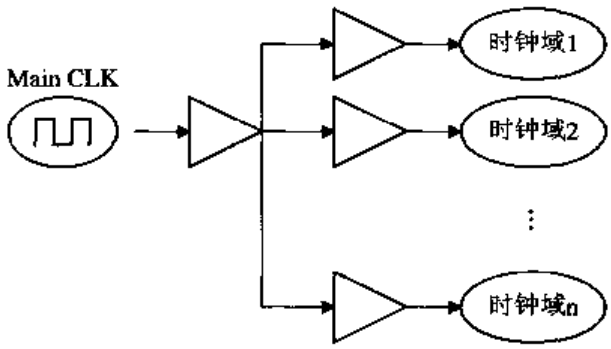


图 2.2　时钟网络

下面给出时钟分配的一些准则：

- 在参考时钟到存储单元(缓冲或者反相器)之间，只能存在一个通路；
- 不鼓励使用时钟分频；
- 不允许时钟信号进行组合运算；
- 在对时钟进行门限化时，要保证不会发生冒险。

2.2.1　全局时钟

稳定可靠的时钟是系统稳定可靠的重要条件。我们不能够将任何可能含有毛刺的输出作为时钟信号，并且尽可能只使用一个全局时钟。全局时钟是最简单和最可预测的时钟。只要可能，就应尽量在设计项目中采用全局时钟。

2.2.2　门控时钟

在许多时候，需要用组合逻辑来控制时钟。需要记住，只有在单频率系统中才可以用门控时钟，并且不能用多个时钟来组合以产生不同频率的时钟。在门控时钟中，逻辑门的

--个输入作为实际的时钟，而该逻辑门的所有其它输入必须当成地址或控制线，它们应遵守相对于时钟的建立和保持时间的约束。

2.2.3 行波时钟

行波时钟是指将一个触发器的输出用作另一个触发器的时钟输入，如图 2.3 所示。

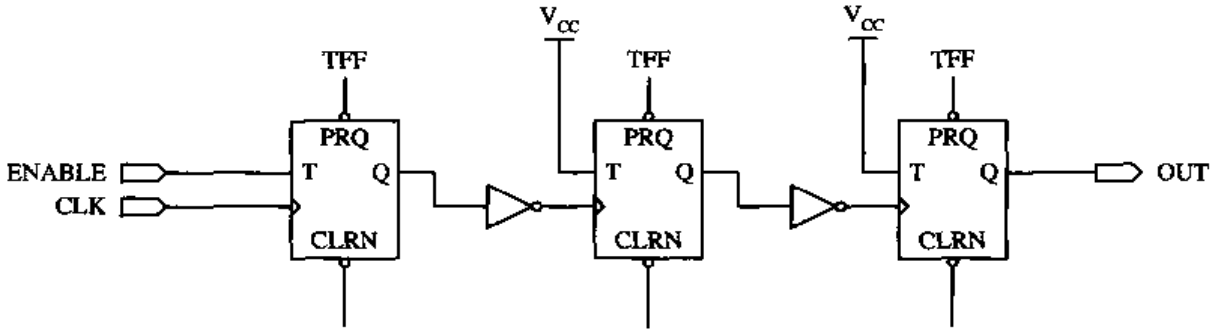


图 2.3 行波时钟

2.2.4 时钟偏移

时钟偏移定义为两个时钟信号之间允许的最大差值，如果超过这个差值，器件工作会发生问题。在多时钟系统中这是一个很关键的问题。

在一个复杂的芯片中，时钟线的扇出非常大，而且它本身的电容与电阻也相当可观，这时候可以把时钟线看作是分布式的 RC 线。用时钟信号控制的触发器，由于触发器离时钟源距离不同，因而时钟信号到达的时间不一样，这种效应称为时钟偏移。时钟信号的目标之一是对系统状态的更新进行同步。由于偏移的存在，同步会受到影响，这可能导致竞争现象以及错误的发生，如图 2.4 所示。

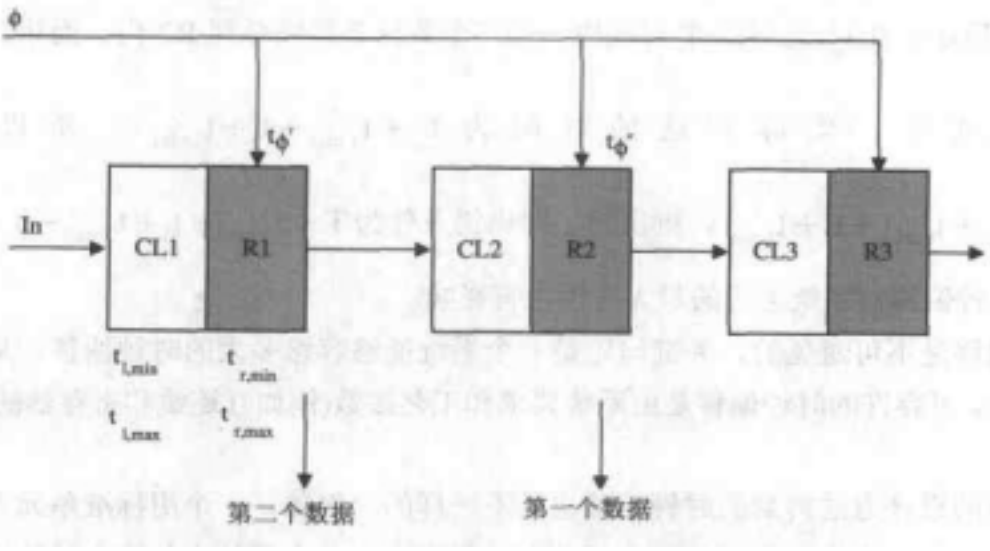


图 2.4 时钟偏移

图 2.4 描述了时钟偏移现象。在这个系统中，每个同步模型由一个组合模块 CL 与一个锁存器 R 构成，并且用六个时序参数来表征：

- 寄存器的最小和最大传递延时($t_{r,min}$ 和 $t_{r,max}$)。
- 组合逻辑的最小和最大延时($t_{l,min}$ 和 $t_{l,max}$)。
- 互连线的传递延时 t_i。
- 局部时钟时间 t_φ。

最大传递延时表示输入信号有变化时，最慢的输出信号响应这种变化所需的时间(最坏情况)。这个延时确定了电路最大的允许速度。一般所说的延时就是指最大传递延时。**最小传递延时**表示当输入有一个变化时，至少有一个输出开始变化所花费的时间。这个时间通常比最大传递延时小得多，更适合于研究时钟偏移(用它可以判断是否有竞争发生)。简化讨论，我们假定锁存器的建立时间等于 0。

现在我们考虑在两个寄存器 R1 和 R2 之间的数据传递。由于存在路径延时，在 R_1 和 R_2 处的局部时钟时间不同，此差异称为时钟偏移δ。

$$\delta = t_{\varphi 2} - t_{\varphi 1}$$

δ可以是正的，也可以是负的，这与布线方向以及时钟源的位置有关。

时钟偏移可能会影响电路的正常功能。什么时候会出错呢？

第一类错误：在一个数据锁存到 R2 之前，R2 的输入就发生了变化。

设某个数据在 $t_{\varphi 1}$ 锁存到 R2，而下一个数据在 R1 中，它最早到达 R2 输入端的时间为 $t_{\varphi 1} + t_{r,min} + t_i + t_{l,min}$，所以，如果 $t_{\varphi 1} + t_{r,min} + t_i + t_{l,min} < t_{\varphi 2}$，则出错，即出错条件为 $\sigma > t_{r,min} + t_i + t_{l,min}$。

第二类错误：一个数据在一个时钟周期内不能传到下一个寄存器。

如图 2.5 所示，在 $t_{\varphi 1}$ 之后的 T 时间内，第二个数据必须锁存到 R2 内。而第二个数据从 R1 传递过来，实际到达的时间为 $t_{\varphi 1} + t_{r,max} + t_i + t_{l,max}$，所以，如果 $t_{\varphi 1} + T < t_{\varphi 2} + t_{r,max} + t_i + t_{l,max}$，则出错，即出错条件为 $T < t_{r,max} + t_i + t_{l,max} - \sigma$。

所以，时钟偏移对系统运行的最大时钟也有影响。

时钟偏移是不可避免的，关键问题是一个系统能够容忍多大的时钟偏移。从上面的分析可以看出，可容许的时钟偏移是由系统要求和工艺参数(例如互连线和寄存器的延时)来决定的。

由不同的设计方法得到的时钟偏移也是不一样的。例如，一个用标准单元方法设计的电路，通常要比一个全定制的偏移大一些。一般而言，一个系统中的流水线阶段越多，则由于时钟偏移导致功能错误的可能性就越大。

表 2.1 给出了解决时钟偏移的三种方案。

表 2.1　解决时钟偏移的三种方案

可用方法	方法描述	优缺点
方案 1	按数据流相反的方向来布时钟走线	此方法会减小电路的最大速度，而且，电路中的数据流并不总是单向的，如图 2.5 所示，存在反馈，则存在正时钟偏移与负时钟偏移
方案 2	如果系统工作在两相时钟下，还可以控制时钟的非交迭时间来消除时钟偏移	应用范围有限，此方法也会减小电路的最大速度
方案 3	通过仔细地分析时钟分配网络来保证时钟偏移在合理的范围内	最现实的解决方法

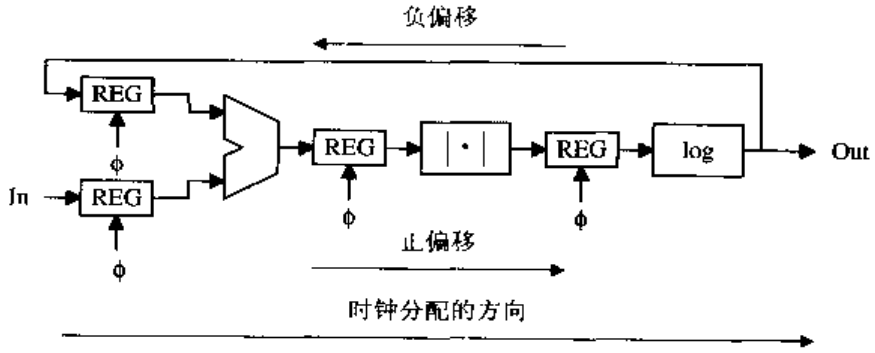


图 2.5　正偏移与负偏移

方案 3 是最现实的方法。设计者可以通过一些设计参数来控制时钟分配网络：

- 时钟网络所用的互连材料。
- 时钟分配网络的形状。
- 时钟驱动和所用的缓冲配置。
- 时钟线上的负载(例如扇出)。
- 时钟的上升和下降时间。

最后一个参数决定了时钟线的大致模型。如果沿着传输线的时钟信号的运行时间与信号的上升/下降时间在同一个数量级，则必须考虑传输线的效应。

这种时钟树中，具有相同时钟偏移的时钟信号之间的延时为 0，因而可以将时钟偏移的影响降到最低。需要说明的是，树形配置是一种理想情况，只对规则阵列有用(假定所有的单元是相同的，时钟可以作为一个二叉树进行分配)。但是，它所包含的思想是通用的。时钟分配网络的目标是：使与时钟信号相连的功能子模块的连接互连线大致等长。

1. 多时钟系统中的时钟偏移

许多设计中都有多频率、多相的时钟。设计这种系统时，必须小心地规划好时序，要防止时钟偏移产生的错误。

设时钟域 1 工作于时钟 CLK_A，时钟域 2 工作于时钟 CLK_B(这两个时钟是由同源时钟产生的)。由时钟域 1 输出的信号 out，经过时钟域 2 中的一个组合逻辑送到时钟域 2 中的

寄存器。因为 out 信号是由 CLK_A 来控制的，它相对于 CLK_B 的建立时间与保持时间都难以保证，可能会导致错误。

解决方法之一，是将 out 信号同步化。也就是说，在将它送到时钟域 2 的组合逻辑之前，先将它送到一个寄存器中。该寄存器的时钟为 CLK_B，如图 2.6 所示。

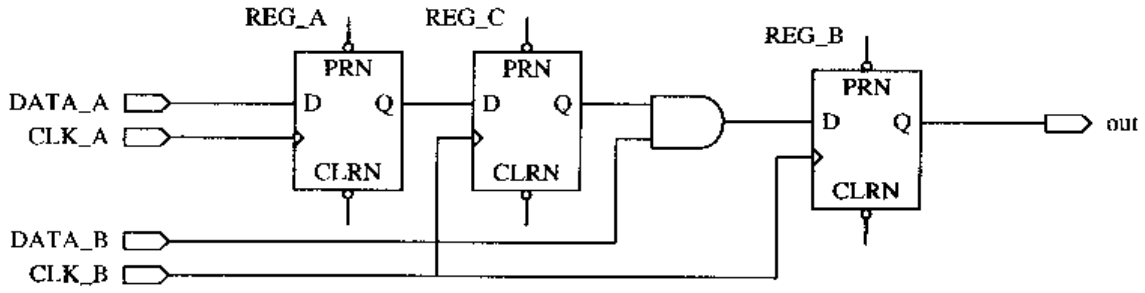


图 2.6　多时钟系统中时钟偏移的消除

但是，如果两个时钟是非同源时钟，这时候时钟偏移的处理就变得非常困难，通常需要引入更高频的时钟。

2. 时钟分频中避免时钟偏移的方法

时钟分频中避免时钟偏移的方法有两种，方法一为如图 2.7(a)所示的传统的时钟分频方法，方法二为如图 2.7(b)中所示的更好的时钟分频方法。

方法二与方法一相比，方法二消除了时钟偏移的影响。

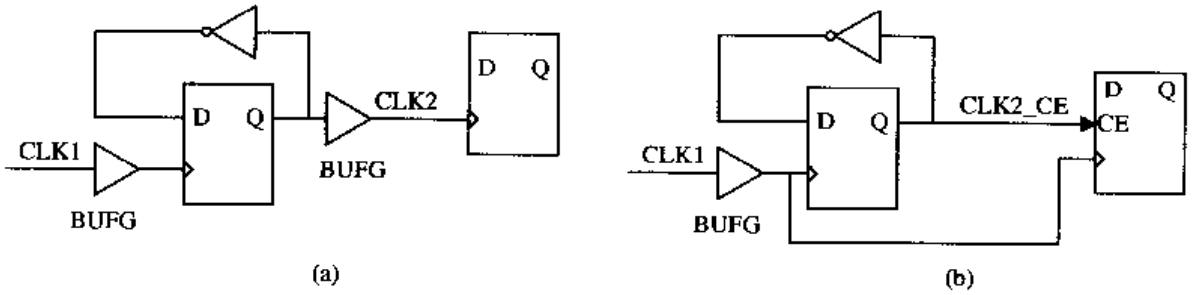


图 2.7　时钟分频中避免时钟偏移的方法

(a) 传统的时钟分频方法；(b) 更好的时钟分频方法

2.2.5　系统级的同步：锁相环

锁相环分为 PLL 和 DLL 两种。简单地说，PLL 是模拟电路，把相位差变成压差，然后控制 VCO，调整输出时钟。DLL 是数字电路，把相位差变成延迟信号，然后通过延迟线调整输出时钟。这里讲的锁相环是指 PLL。在第 4 章中我们将介绍 DLL。

时钟偏移会使电路出错。解决时钟偏移的方法之一是采用锁相环 PLL。它的功能是校正所有时钟信号的相位，以消除时钟偏移。

锁相环是一种复杂的、非线性的电路。它的最基本的结构如图 2.8 所示。它由三个基本的部件组成：鉴相器(PD)、环路滤波器(LPF)和压控振荡器(VCO)。

锁相环是一个相位误差控制系统。它通过比较输入信号和压控振荡器输出信号之间的相位差，从而产生误差控制电压来调整压控振荡器的频率，以达到与输入信号同频。其中，**鉴相器**是个相位比较装置，它把输入信号 $S_i(t)$和压控振荡器的输出信号 $S_o(t)$的相位进行比较，产生对应于两个信号相位差的误差电压 $S_e(t)$。**环路滤波器**的作用是滤除误差电压 $S_e(t)$中的高频成分和噪声，以保证环路所要求的性能，增加系统的稳定性。**压控振荡器**受控制电压 $S_d(t)$的控制，使压控振荡器的频率向输入信号的频率靠拢，直至消除频差为止，这时候也就锁定了输入信号。

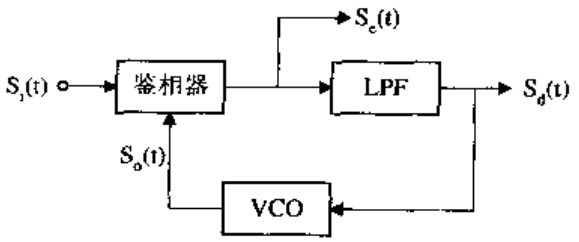


图 2.8　锁相环工作原理图

清楚了上面 PLL 的原理，就很容易明白 PLL 能消除时钟偏移的原因。在电路中，$S_i(t)$一般作为参考时钟信号，$S_o(t)$作为局部时钟信号。在环路开始工作时，两信号之间存在固有的频率差。当局部时钟滞后于参考时钟时，产生一个 Up 信号；当局部时钟领先于参考信号时产生一个 Down 信号。Up 或 Down 信号送到一个电荷泵中，“电荷泵”是一种形象的说法，它将两个时钟信号的差异转换为相应的模拟电压：Up 信号增加控制电压的值，并加速 VCO，这使得局部信号跟上参考时钟；Down 信号将减缓振荡器，并消除局部时钟领先的相。这样，压控振荡器的频率一直在变化，努力跟上输入信号的变化。若压控振荡器的频率能够变化到与输入信号频率相等，在满足稳定性条件时就在这个频率上稳定下来。达到稳定后，输入信号和压控振荡器输出信号之间的频差为零，相差不再随时间变化，误差电压为一固定值，这时环路就进入“锁定”状态。这就是锁相环工作的大致过程。

图 2.9 说明了锁相环消除时钟偏移的作用，设 Φ 是参考时钟，Φ'_1 和 Φ''_1 是两个局部时钟。

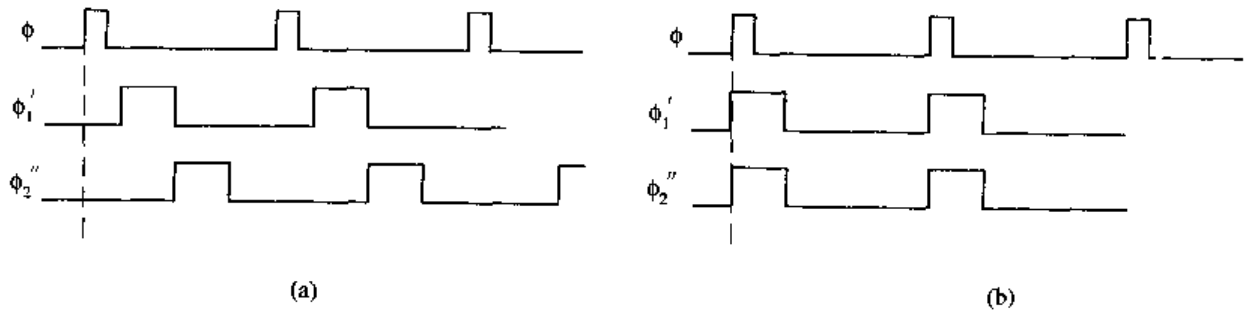


图 2.9　锁相前后的时钟信号

(a) 锁相之前的时钟信号；(b) 锁相之后的时钟信号

将电荷泵的输出直接传到 VCO，产生的时钟信号可能很不稳定，通过引入环形滤波器可以消除这种现象。环形滤形器的级数通常是 1 到 3 级。

注意：PLL 结构是一个反馈结构，额外的相移位(由于高级滤波形引起的)可能导致波形不稳定。

在实际中，VCO 通常用一个环形振荡器来实现，同时产生不同相位的时钟，以满足 IC

的需要。

PLL 除了在内部与外部时钟之间进行同步外，还有其它潜在应用：在局部时钟与鉴相器之间引入分配器，可以使芯片上的内部时钟工作频率超过外部参考时钟。

现在已有完全数字化的 PLL，但很昂贵。

说明：设计者喜欢单一时钟系统，因为它稳定，易于设计和验证。但有时候，不可避免地要面对多时钟的设计。(特别是在电信和数据通讯领域，将数据由一种时钟域传到另一种时钟域的情况很常见。)这种场合，需要设计者使用一些技巧来保证时序的正确。

2.3 时　　序

设计者接触同步时序电路的机会多一些。在同步时序电路中，各种操作都受到时钟信号的控制，而时钟频率是由某一操作的最大延时决定的。设计者要保证电路工作频率尽可能高，又要保证电路在特定的情况下能工作可靠。因此时序往往是最重要又是最棘手的问题。要牢记一句话“timing is everthing”。

2.3.1 时序图

在系统规范及系统设计报告中，时序图是必不可少的工具。时序图可以直观地描述一个数字电路功能行为：通过描述外部信号的时序，可以定义设计的时序要求。通过描述设计的内部关键信号的时序，可以定义各组成部分接口的时序关系(在 IC 设计中，不同组件间接口的时序，是一个非常重要的问题)。

时序图是设计规范的一部分。它不仅包括一段时间内信号的状态，还包括传输延时，时间约束等等。画出关键的时序图，可以在设计的早期发现问题。产生时序图的一个工具是 Chronology 公司的 TimingDesigner。它可以进行最坏情况的时序分析。

下面以一个组合逻辑和一个寄存器为例，说明在电路设计中要关心的时序的内容。

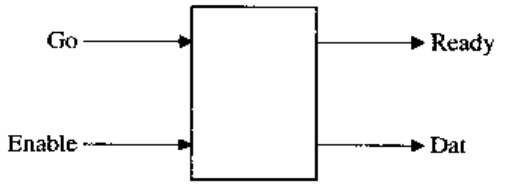


图 2.10　组合逻辑框图

图 2.10 画出了一个组合逻辑的框图。它的时序图如图 2.11 所示。

在电路中，不同的数据通路可能有不同的传递延时(例如，从 Go 到 Ready 端的延时 T_{rdy} 与 Go 到 Dat 的延时 T_{dat} 可能不同)。当信号从低到高变化与从高到低变化时，传递延时也可能不同(如图 2.11 中所示的两个 T_{rdy} 可能不相同)。即使相同通路、相同变化，延时也有可能不同(例如，信号从低变到高的延时 T_{rdy} 可能不同。在图 2.11 中用阴影区域来表示，Ready 可以在该阴影区域的任何时刻有效)。这是因为电路中的延时不仅跟电路结构有关，也跟工作电压、温度、制造工艺有关。所以，在仿真时，这种延时通常有最小延时、典型延时和最大延时之分。在 Verilog 中，用 Sepcify 语句来设置这三种延时。许多设计师更关心最大延时，以保证电路在各种极限条件下都能正常工作。如果所设计的产品制造与工作的环境较为单一，也可以用典型延时。在高速电路设计时，可能需要考虑到最小延时。

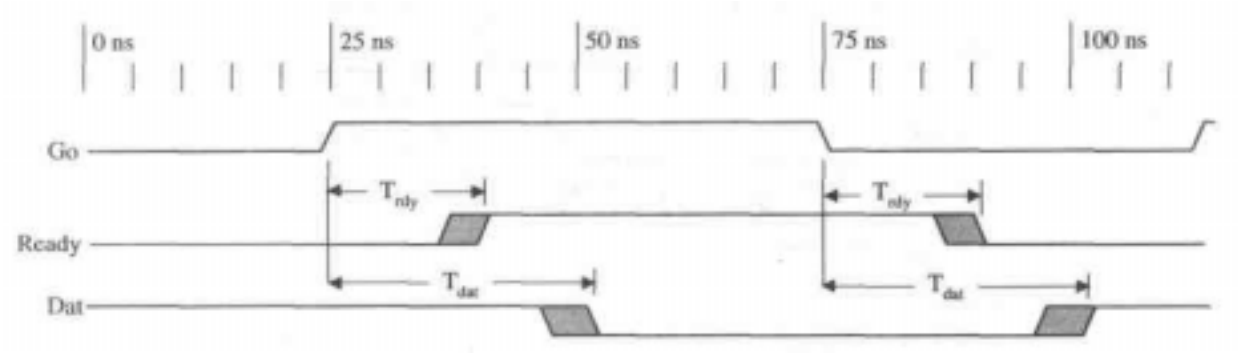


图 2.11　组合逻辑的时序图

说明：从某个输入到某个输出可能有不同的通路，因而传递延时也不同。要分析所有这些通路上的延时会非常复杂，所以设计者一般采用最坏情况的延时，即考虑所有通路中最大的延时 t_{pLH} 和 t_{pHL}。虽然这种考虑能保证电路稳定工作，但电路可能不能发挥最好的性能。

在电路中，存储部件的时序非常重要。图 2.12 所示为寄存器示意图。它的时序图如图 2.13 所示，其中包括：数据输入到输出的延时；建立时间；保持时间。

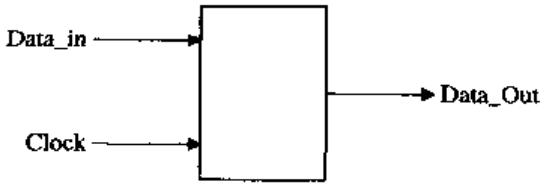


图 2.12　寄存器示意图

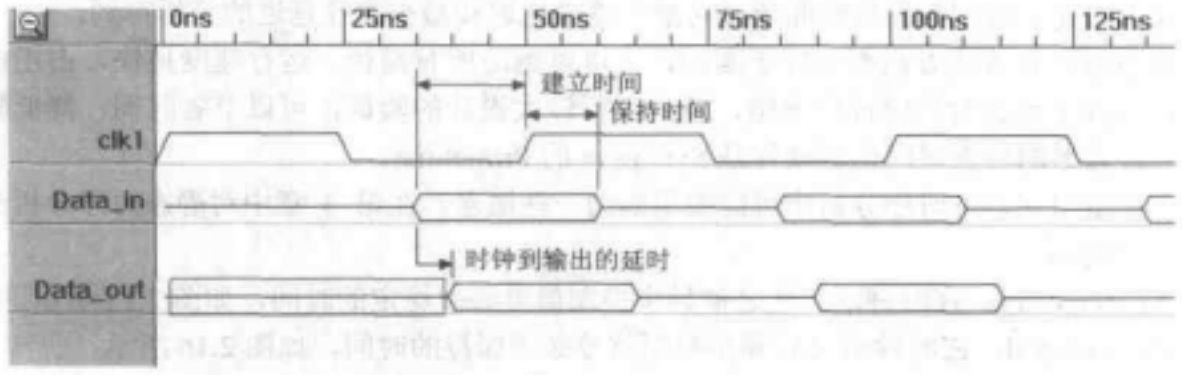


图 2.13　寄存器中的时序

2.3.2　建立时间/保持时间

综合工具在综合后会给出建立时间与保持时间。建立时间与保持时间都应该为正值。如果为负值(如图 2.14 所示)，则电路不能正确工作。

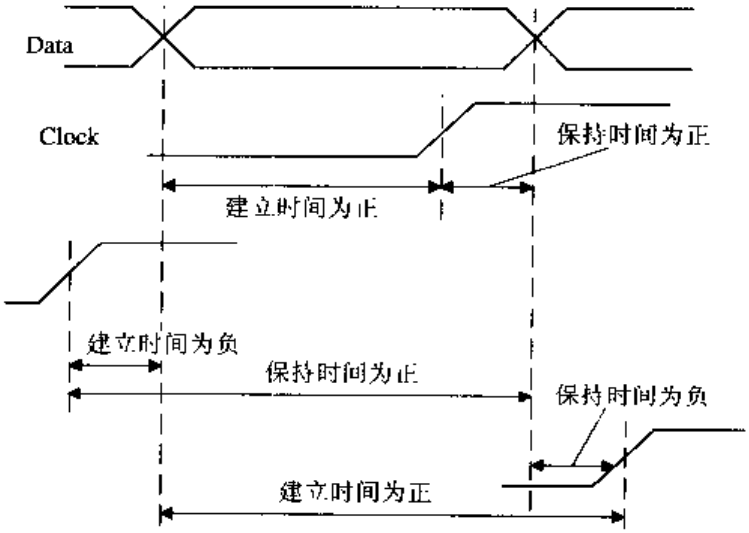


图 2.14　建立时间与保持时间

2.3.3　静态时序分析中的概念

传统上是采用动态仿真来验证一个设计的功能和时序。随着设计规模的增大，验证一个设计所需要的测试向量的数量以指数增长，且这种方法难以保证足够的覆盖率。在大型设计中，如果仅用传统的动态仿真的方法，则时间及工作量都难以承受。

静态时序分析可以降低验证的复杂性。它提取整个电路的所有时序路径，通过计算信号在路径上的延迟传播，找出违背时序约束的错误，主要是检查建立时间和保持时间是否满足要求，建立时间与保持时间通过对最大路径延迟和最小路径延迟的分析得到。

静态时序分析的方法不依赖于激励，且可以遍历所有路径，运行速度很快，占用内存很少，弥补了动态时序验证的缺陷，适合进行较大设计的验证，可以节省时间，降低验证成本。最著名的静态时序仿真软件是 Synopsys 的 Primetime。

下面给出在静态时序分析中可能要用到的一些概念。在第 3 章中对静态时序分析进行了进一步说明。

(1) Recovery：在时钟沿到达之前异步控制信号必须稳定的时间，如图 2.15 所示。

(2) Removal：在时钟沿之后异步控制信号必须保持的时间，如图 2.16 所示。

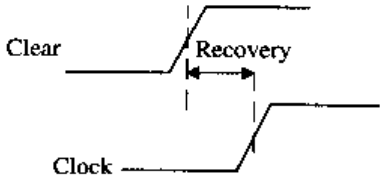


图 2.15　Recovery 示意图

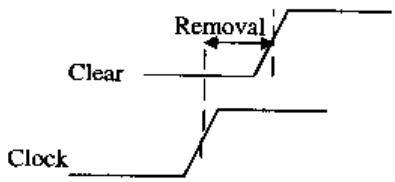


图 2.16　Removal 示意图

2.4 总 线 设 计

总线是一种数据通道。现在许多数字系统都是基于总线的。总线的设计与运行原理是一个相当复杂的问题。总线设计包括总线宽度、总线时钟、总线仲裁和总线操作。

2.4.1 总线宽度

对于地址总线，总线越宽，则系统能够寻址的空间越大。例如地址总线宽度为 n，则系统能够寻址的宽度是 2^n。在 32 位计算机中，地址总线宽度为 32，则可以寻址的存储器空间为 4 GB。对于数据总线，总线越宽，则每次可传送数据越多，带宽越大。但总线宽度越高，则代价越高，并且还有向前兼容的问题。

2.4.2 总线时钟

总线根据时钟类型，可以分为同步和异步两类。同步总线中，所有的操作要受到时钟的控制。PCI 总线是一种同步总线，时钟频率是 33 MHz/66 MHz/100 MHz。下面以图 2.17 为例，说明在一个计算机系统中同步总线的工作原理。图中的关键时序要求在表 2.2 中进行了说明。

在该例中，所用的时钟信号是 20 MHz，因此相应的总线周期是 50 ns。

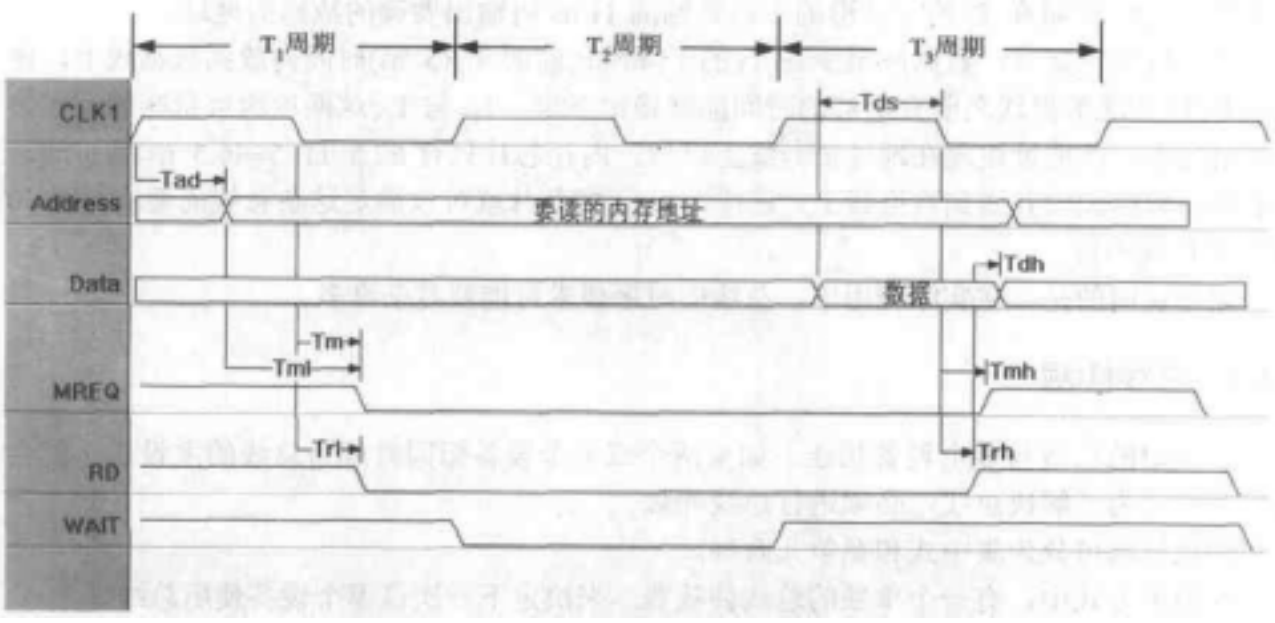


图 2.17 同步总线上的读时序

假设内存在读地址建立后还需要 40 ns 才能输出数据。在地址稳定后，CPU 发出 MREQ 与 RD 信号。前一个信号说明系统要访问的是内存，后一个信号表示要进行的是读操作。由于内存芯片要在地址建立 40 ns 后才能输出数据，所以无法在第一个时钟将 CPU 要读的数据输出。为了通知 CPU 不要期待马上得到数据，内存在 T_2 的起始处发出一个 WAIT 信号，

这将插入一个等待状态。内存完成数据输出后，将 WAIT 信号置反。在本例中，由于内存较慢，插入了一个等待状态(T_2)。在 T_3 的起始位置，内存确知它能够在本周期给出数据，所以将 WAIT 信号置反。

表 2.2　关键时序要求

符号	含义	最小值	最大值	单位
T_{ad}	地址输出时延		11	ns
T_{ml}	MREQ 前的地址稳定时间	6		ns
T_m	MREQ 的 T_1 周期下降沿开始的 MREQ 时延		8	ns
T_{rl}	从时钟的 T_1 周期下降沿开始的 RD 延时		8	ns
T_{ds}	在时钟的下降沿之前的数据建立时间	5		ns
T_{mh}	从时钟的 T_3 周期下降沿开始的 MREQ 时延		8	ns
T_{rh}	从时钟的 T_3 周期下降沿开始的 RD 时延		8	ns
T_{dh}	RD 置反后数据保持时间	0		ns

在 T_3 的前半部分，内存将读出的数据放到数据信号线上。然后，在 T_3 的下降沿，CPU 选择数据信号线，将读出的数据锁存到一个内部寄存器中。读完数据后，CPU 再将 MREQ 与 RD 信号置反。

图 2.17 中给出了一些时序要求。例如，T_{ad} 是从 T_1 的上升沿开始到地址建立好时的地址建立时间。根据时序规格要求，$T_{ad} \leqslant 11$ ns。这就是说，CPU 生产商保证，在任何一个读周期中，CPU 都将在 T_1 的上升沿的中点开始的 11 ns 内输出要读的数据的地址。

时序规格也要求，数据应至少在 T_3 的下降沿之前的 T_{ds}(5 ns)时间内放到数据线上，使其在 CPU 选通数据线之前有足够的时间能够稳定下来。T_{ad} 与 T_{ds} 这两项约束意味着，在最坏的情况下，在地址出现在地址信号线上以后，内存芯片只有 62.5-11-5=46.5 ns 的时间，就必须将数据读出并送到数据线上。这样，40 ns 的芯片就可以满足这条总线的要求，而 50 ns 的芯片则不行。

需要说明的是，在实际应用中，总线的时序要求可能要复杂许多。

2.4.3　总线仲裁

系统中的总线与多个设备相连。如果两个或多个设备想同时成为总线的主设备，就会发生冲突。为了解决总线，必须进行总线仲裁。

仲裁机制可分为集中式和竞争式两种。

在集中方式中，有一个单独的总线仲裁器，来决定下一次该哪个设备使用总线。

在竞争方式中，每个设备需要总线时，都发出总线请求信号，所有的设备都监听所有的总线请求信号。这样，在每个总线周期结束时，每个设备都能知道自己是否是优先级最高的总线请求者，以及能否在下一个总线周期使用总线。这种方式要求的总线信号更多，但防止了总线的潜在浪费。

2.4.4 总线操作

在实际的系统中，存在多种总线周期。例如，普通的读写周期、块传输周期、中断周期等。

2.5 练　　习

1. 查阅相关资料，画出 PCI 总线写操作时序图。

2. 对于具有图 2.18 所示的操作时序的总线，试计算总线的数据传输速率，假设总线上传输的每个字为 32 位，总线时钟频率为 50 MHz。

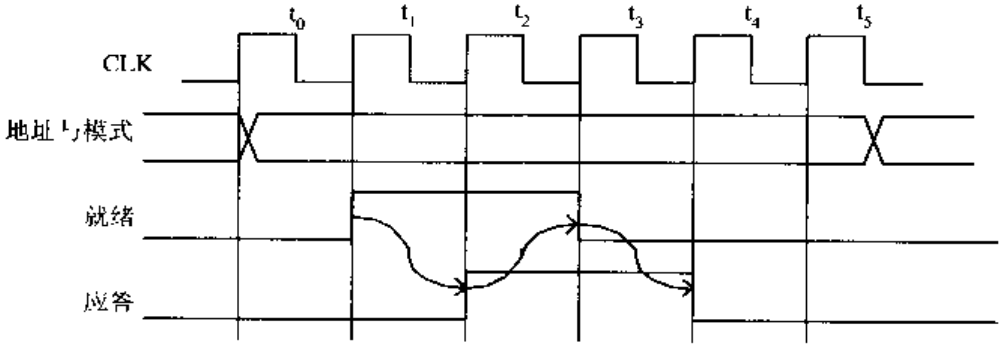


图 2.18　总线时序图

第3章 综 合

本章介绍了综合软件 DesignCompiler 的使用和面向综合的设计技术，并给出了基本设计单元的综合结构，以帮助读者掌握这一在 IC 设计中最难以理解的概念，并了解与硬件设计语言相对应的电路结构，从而设计出高性能的电路。

3.1 综合(Synthesis)的概念

用 Verilog 或 VHDL 设计电路，需要将语言描述转换为电路图描述，即用芯片制造商提供的基本电路单元(综合库)实现我们用硬件描述语言(Verilog 或者 VHDL)描述的(RTL 级)电路的功能，这个过程就称为综合。

综合有逻辑综合和行为综合之分。这里所说的综合，是指逻辑综合(即 RTL 级的综合)。虽然 Synopsys 等公司一直致力于行为综合(即高层综合)方面的研究，并且对特定应用，行为综合也取得了一些进展，但离实际应用还有一定距离。

在 ASIC 设计流程中，逻辑综合、优化及扫描链的插入都是由综合工具完成的。

1. 综合的步骤

综合过程可以细分为三个步骤：

- 转译(Translation)：读入电路的RTL级描述，将语言转译成每条语句所对应的功能块以及功能块之间的拓扑结构，这一过程的结果是在综合器内部生成电路的布尔函数的表达，不做任何的逻辑重组和优化。
- 优化(Optimization)：基于所施加的一定时序和面积的约束条件，综合器按照一定的算法对转译结果作逻辑重组和优化。
- 映射(Mapping)：根据所施加的一定的时序和面积的约束条件，综合器从目标工艺库(Target Technology)中搜索符合条件的单元来构成实际电路。

由芯片制造商(Foundry)提供的工艺库，有一系列的基本单元，如与非、或非、反相器、锁存器、触发器、选择器等等。对这些单元的电器特性可以进行描述，例如：单元的面积、输入端的电容、输出端的驱动能力、单元的逻辑功能、单元的时序等等。综合的目标就是用工艺库文件提供的这些单元来实现用 RTL 代码描述的逻辑功能，并满足设计者提出的面积和时序要求。

综合是由工具自动完成的。但这不等于说，设计者可以对此毫不关心。

2. 对设计者的要求

一个合格的设计者，应该能够在自己脑子里再现“转译”这个过程，即清楚自己用 HDL

语言所构造的电路结构是什么样子的。写代码时，请时刻记着三个准则：

“think hardware”：要时刻想着代码所描述的电路结构，熟悉不同的代码结构，清楚通过综合能获得怎样的硬件实现，这是芯片设计者必须具备的素质。同时必须记住，综合器无法帮助用户实现功能，无法作算法或功能的优化，而只能够在已有功能的基础上按照用户的要求选择较优的实现。

“think synchronous”：要时刻关心电路的同步问题。同步电路设计的优点在于系统中信号流的可预见性，因此避免了诸如时序设定和实现上的困难。在综合时，如果将同步和异步的时序设计混合在一起，采用同样的约束条件，有可能导致最后的结果和预期的不一致，因此为确保综合前后的设计在功能上的一致，设计中应尽可能使用同步电路，或者将设计中同步的部分和异步的部分分成不同的模块，采用不同的综合方法。此外，一个大的设计可能存在两个或者两个以上的时钟，最好将不同的时钟域(clock domain)划分为不同的模块，便于简化综合时时序约束的设定。

“think RTL”：要清楚寄存器构造、电路拓扑及寄存器之间的功能部分。可综合的设计是我们通常所说的 RTL(Register Transfer Level)级，从电路上可知，就是描述电路中寄存器的结构和寄存器之间的功能实现。而综合工具需要做的就是将已确定的寄存器之间的逻辑加以优化。因此寄存器的分配是获得好的设计的关键。

最著名的的综合工具为Synopsys公司的Design Compiler(以下称DC)。相信许多设计者都会接触到这一工具，我们将对其进行简单的介绍。更详细的说明，请参阅该工具的帮助文件及Synopsys公司的培训教材。

3.2 Design Compiler 简介

DC 所提供的功能包括：

- 层次化的综合(如由上而下或由下而上的综合策略)；
- 全面(Full)和递进式(Incremental)的综合方法；
- 针对复杂的触发器和锁存器的时序优化；
- I/O 脚的插入和优化；
- 利用状态表的方法优化有限状态机(FSM)；
- 层次的边界优化。

下面介绍 DC 综合工具中的几个基本概念。这些基本概念对于理解和掌握 DC 工具非常重要，希望读者能够牢记。

表 3.1 基 本 概 念

基本概念	描述
目标工艺库 target_library	目标工艺库是指将 RTL 级的 HDL 描述映射到门级时所需的标准单元综合库，它是由芯片制造商(Foundry)提供的，包含了物理信息的单元模型。通常芯片制造商提供的基本电路单元有：与非，或非，非门等基本的逻辑单元；还有选择器、与或非、或与非等较复杂的组合逻辑，以及 LATCH(锁存器)和 DFF(D 触发器)等时序逻辑单元。它可以有多种表现方式，我们称之为目标工艺库(Technology file)。针对不同的工艺线，需要设置不同的目标库。例如： target_library = { csm06.db }

续表

基本概念	描述
链接库 Link_library	链接库可以是同 target_library 一样的单元库，或者是已综合到门级的底层模块设计，其作用如下： 在由下而上的综合过程中，上一层的设计调用底层已综合模块时，将从 link_library 中寻找并链接起来。 如果需要将已有的设计从一个工艺 A 转到另一个工艺 B 时，可以将当前单元综合库 A 设为 link_library，而将单元综合库 B 设为 target_library，重新映射一下即可。 我们可以一次定义两个链接库，例如： link_library = { csm06.db other.db}
符号库 symbol_library	显示电路时，用于标识器件、单元的符号库。例如，我们可以设置： symbol_library = { csmhd.sdb}
search_path	search_path 给出了 DC 环境下读入的文件(包括综合库和设计文件)的搜寻路径。例如，我们可以写成： search_path = { . /home/csmclib/MyLib/chart06/std-cell/frontend/syth_lib }
建立时间	工艺库定义了一些时序逻辑单元的建立时间，例如一个 DFF，它的数据必须在时钟沿来到之前的某个时刻稳定。如图 3.1 所示，d 应该比 ck 的上沿提前 T_{set} 的时刻稳定。如果 d 在 T_{set} 这段时刻变化，则 DFF 不能保证锁存到正确的数据
保持时间	工艺库还定义了时序逻辑的保持时间。例如一个 DFF，它的数据在时钟沿来到之后的一段时间内不能变化，否则 DFF 不能保证锁存到正确的数据。如图 3.1 所示，d 在 T_{hold} 的时段中不能变化
基本单元的延时	对于所有的电路，电信号通过都需要时间，这在逻辑电路中反映为信号的延时。如图 3.1 的 T_d 反映了数据在 ck 上沿开始，直至数据从 q 端输出所需的时间
连线的延时	除了单元电路中电信号的 DELAY(延迟)，电信号在连线中也有 DELAY，这是有连线的电阻电容导致的
工艺偏差	在流片的阶段，WAFER 在流水线上需要经过几十道工序，如氧化、光刻，离子注入、扩散、淀积等等，这些工序在控制上总会有一些偏差，例如氧化厚度、光刻的精度、离子注入的能量和浓度、扩散的深度等等，都会导致器件的性能出现一些变化。一般在逻辑电路中表现为驱动能力，或者信号延时的变化
工作温度	当温度变化时，会导致沟道电流强度的变化，从而影响逻辑电路的驱动能力和信号延时
工作电压	电压比较高的时候，逻辑电路单元的驱动能力增强，信号延时变小，从而可以运行在比较高的工作频率下
约束	DC 综合是根据设计者给出的约束条件进行的。通常我们读入 RTL 的源代码，并设定各种约束条件，就可以启动 DC 进行综合
约束对象	DC 将一个设计中的不同属性的部分分为 7 个对象。它们分别是：Design、Cell、Reference、Port、Pin、Net、Clock。设置约束条件时，必须说明约束针对的对象。这个约束对象的含义在图 3.2 中进行了说明
fanout_load 与 max_fanout	单元的 fanout 即扇出值是以基本电容值为单位值得到的。 fanout_load 用于输入管脚，而 max_fanout 则用于输出管脚。 任何输出管脚驱动的所有单元输入管脚的 fanout_load 之和不能超过该输出管脚的 max_fanout，否则就是 DRC 出错。

续表

基本概念	描述
max_transition 与 max_capacitance	transition time 是指信号的变化通过连线从一端传递到另一端的时间。 如果综合库中的单元采用 CMOS 延迟模型，DC 通过将驱动管脚的驱动能力(Driving strength)乘以线上所连的电容得到 transition time；如果综合库中的单元采用查表和插值方式的非线性时延模型，则 DC 根据单元驱动的负载值在表中找出或插值计算得到 transition time。 max_transition：用于输出管脚，DC 通过计算连线上的电容值和输出管脚的驱动能力，保证连线上的 transition time 小于等于 max_transition 属性； max_capacitance：用于输出管脚，直接给出输出管脚所能驱动的电容负载，包括连线负载和被驱动单元的输入负载
时间通路 timing path	时间通路是指逻辑信号传递的路线，其起点可以是所有的基本输入端(Primary Input)和所有时序单元的时钟输入端，终点可以是所有的基本输出端(Primary Output)和所有时序单元的数据输入端；DC 综合时的算法均是基于时间通路的
DC 的初始化文件	DC 工作环境由软件系统定义的环境变量来设定，这些环境变量在以下三个目录下的隐含文件.synopsys_dc.setup 中给出： • SYNOPSYS 的安装目录：$SYNOPSYS/admin/setup； • 用户的根目录：User's home directory； • 用户当前的工作目录：User's current working directory。 DC 按上述次序读入其中的.synopsys_dc_setup 文件，如有相同变量的赋值，则后者将覆盖前者的设置或者添加到变量值中
DC 中预定义变量的设置	DC 中的变量很多，用来设定综合器的系统环境、设计的工艺相关属性和施加在设计之上的约束条件和属性等等，而在初始化文件.synopsys_dc_setup 中设置的多是预定义的环境变量。当然用户也可以根据需要另外设置一定的设计变量值或者自定义变量值，建议这些设置仅限于工作目录下的初始化文件。 变量的赋值方式： 变量名= Value or { Value1 Value2 ...} or 变量名+{Value1 Value2 ...} 第一、二种赋值，之前定义的变量值将被当前值覆盖，而第二种则将当前值添加到变量值中

建立时间、保持时间等的概念在第 1 章已经进行了说明。因为在 DC 中这些概念经常用到，所以在图 3.1 中我们再次给出它们的图示。

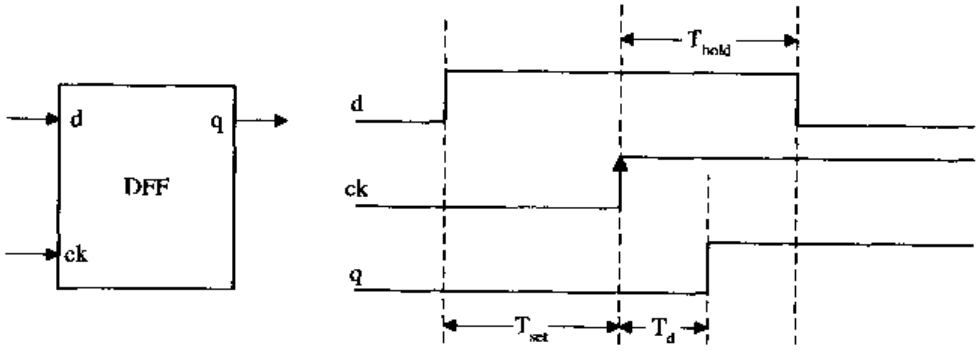


图 3.1 建立时间与保持时间

在 UNIX 下，有四种方式执行 DC。它们分别是：

- 图形界面 design_vision。这个图形界面的工具支持 TCL。
- 图形界面 design_analyzer。这个图形界面的工具不支持 TCL。
- 命令行界面 dc_shell –t。这个命令行界面支持 TCL。
- 命令行界面 dc_shell。在 UNIX 的环境下，键入 dc_shell，即可进入这种模式。然后可以键入各种命令，以命令行的方式执行 DC 的命令。通常情况，我们先编辑一个脚本文件(例如：design.scr)，然后键入 dc_shell –f design.scr，就可以启动 DC 并执行文件中的各条指令。

图 3.2 给出了 DC 中的各种约束对象。该图非常重要，是理解 DC 中所有操作的基础，也是理解下面几节的前提。希望读者一定要把该图记在心里。

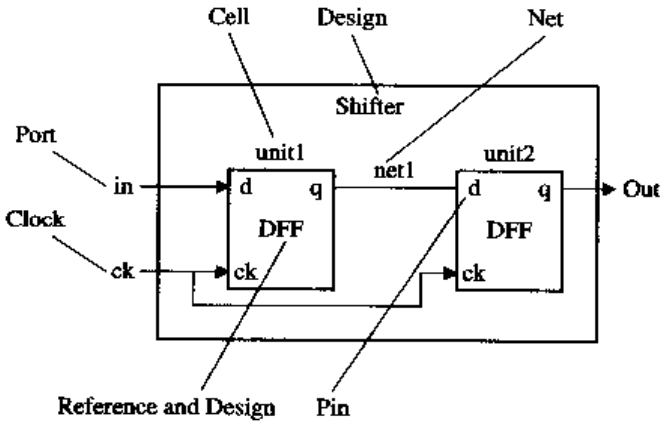


图 3.2 DC 中的各种约束对象

3.3 综合条件的设置

DC 在进行综合时，要进行延时计算和设计规则检查。需要设置的相关内容主要包括：操作环境；导线负载模型；设计约束；设计规则约束。

3.3.1 操作环境

操作环境是将来芯片要工作的环境。操作环境包括：操作温度、供电电压、制造工艺的偏差。芯片的操作温度、供电电压、制造工艺的偏差，以及工作的电压等都会对时序产生影响，如果没有这些参数的设置，则 DC 难以准确地估计时序。

一般来说，电压越高，温度越低，同时生产流片时的工艺偏差系数越小，产品的速度越快；反之则越慢。因此在综合库中一般都会提供 worst、typical、best 三种模式，代表最坏、典型和最好三种情况。如下面的例子：

Operating Conditions:

Name	Library	Process	Temp	Volt	Interconnect Model
BEST	csm06	0.80	0.00	5.50	best_case_tree
TYPICAL	csm06	1.00	25.00	5.00	balanced_tree
WORST	csm06	1.20	125.00	4.50	worst_case_tree

操作条件在工艺文件中进行了定义。下面给出一种操作条件的定义：

```
operating_conditions ("WCCOM") {
    process : 1.2 ;
    temperature : 150 ;
    voltage : 1.65 ;
    tree_type : "worst_case_tree" ;
}
```

选择操作条件时，可以用如下方式：

```
dc_shell> set_operating_conditions WCCOM -library mylib
```

3.3.2 导线负载模型

在实际电路中，导线具有电阻和电容，当连线比较长时，也会对信号的延时产生较大的影响。在深亚微米的设计中，连线延时不能忽略。

下面给出一个导线负载模型：

```
wire_load ("4000") {
    capacitance : 0.0002;
    resistance : 0.25e-3 ;
    area : 0.01 ;
    slope : 80.5 ;
    fanout_length (1, 12.0) ;
    fanout_length (2, 25.0) ;
    fanout_length (3, 40.0) ;
    ...
    fanout_length(20, 500.0) ;
}
```

以上是一个连线模型。这里的电容、电阻和面积值是相对连线长度的比例因子，slope为外插值的斜率；为获得完整的电容、电阻和连线面积，DC 先要预估出该连线的长度。

在工艺库中，可以用如下方式选择此导线负载模型：

```
default_wire_load : "4000" ;
```

3.3.3 设计约束

设计约束主要包括两类，一类是与性能有关的，一类是与面积有关的。

1. 性能约束

性能约束主要包括：

时钟(creat_clock 与 set_clock_skew)；

输入延时(set_input_delay)；

输出延时(set_output_delay)；

驱动(set_drive,set_driving_cell,set_load)；

负载(set_load)。

1) 时钟

系统的时钟可以是施加在实际的基本输入脚上的，也可以是创建的一个虚拟时钟。前者用在当前设计模块中存在的时钟网络的情况，后者一般用于当前所设计的为全组合逻辑电路的情况。

电路中时钟网络的实现有别于其它部分。由于时钟的分布和各个时钟分支的驱动是由后端的版图设计来确定的，因此，在综合时，可以将时钟信号的驱动设为最大，而不考虑其延迟。

如果系统对时钟网络的延迟已有要求，则可以在创建时钟时加上此延迟，以保证模块的工作环境接近真实情况。

create_clock 命令定义了一个时钟。这个时钟是期望芯片所能运行的时钟。对于建立的时钟，最好给出名称(用-name 选项)。在设置输入延时与输出延时时，需要用时钟名称作为参考。如果没有给出时钟名称，则用该信号第一个端口(port)的名称作为时钟名称。

时钟的行为可以用“-waveform {rise_edge,fall_edge,[rise,fall]}”来描述。

例如，使用命令“create_clock -period 10 -waveform {0, 5} clk”得到的时钟信号如图 3.3 所示。

在设计时，一般在布局布线阶段建立时钟树。在实际的时钟树中，各时钟的边沿不能保证完全对齐(DC 称为 clock_skew，即有时钟偏移)。如果 DC 将时钟做为理想的时钟，则计算 setup 和 hold 时间就会和实际情况产生误差。

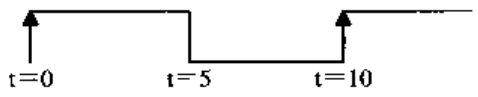


图 3.3 生成的时钟信号

例如，如图 3.4 所示，对于理想的时钟，T_{set} 和 T_{hold} 都满足工艺库中 DFF 的要求，但是实际的时钟，由于其边沿比理想的时钟向后延时了，导致 T_{setup} 可能不满足，从而导致 DFF 不能锁存数据。

导致这种危险的根本原因在于：DC 综合时，根据理想的时钟进行时序分析，而实际的时钟却和理想时钟有差异。

要解决这一问题，可以定义时钟偏移。这样 DC 综合时，会考虑 clock_skew 对时序的影响。

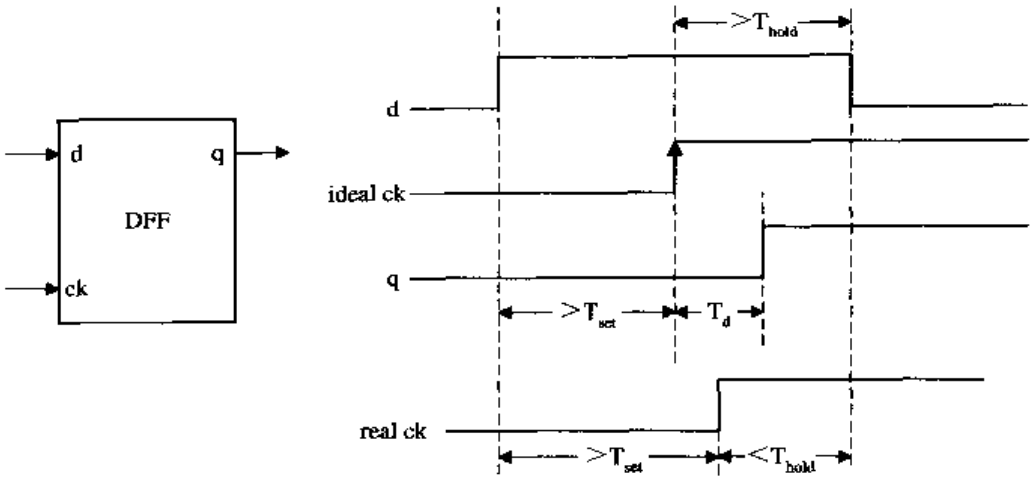


图 3.4　时钟扭曲导致保持时间不满足

set_clk_skew 用来设置时钟偏移。下面给出一个例子：

dc_shell>set_clock_skew -minus_uncertainty 0.1 -plus_uncertainty 0.2 ck3

该命令得到的时钟如图 3.5 所示。该时钟的时钟沿有一个不确定的范围。

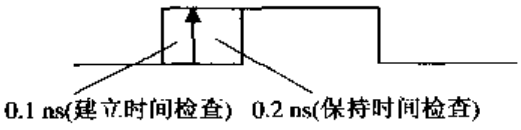


图 3.5　时钟偏移

2) 输入延时

在介绍输入延时之前，首先介绍一下 DC 综合中的时序要求。图 3.6 对此进行了说明。

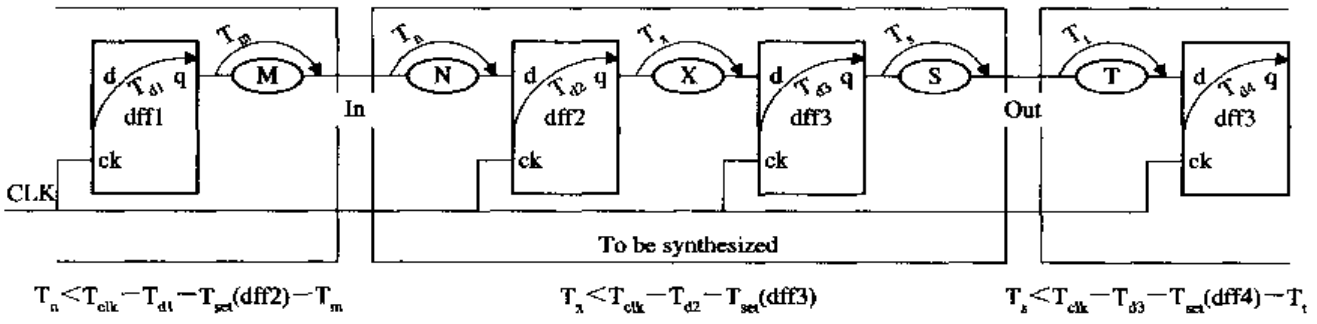


图 3.6　时序要求

在图 3.6 中，中间部分是待综合的设计。根据工艺库的定义，DC 可以获得 DFF2 和 DFF3 的建立时间和数据通过的延时，并且根据 create_clock 定义的时钟周期，可计算得到 T_x。

但对于前面的模块与后面的模块，由于 DC 并不知道 T_m 和 T_t 以及 T_{d1},T_{d3},T_{set}(dff4)等参数，因此无法计算 T_n 和 T_s。所以必须给出输入 PORT 的约束和输出 PORT 的约束。

set_input_delay 可以用来对寄存器路径的输入进行约束。该命令设置了信号到达当前设计输入端口所用的时间。例如：

dc_shell> set_input_delay d0 –clock CLK all_inputs()

图 3.7 对输入延时进行了说明。设电路的时钟周期为 T，如果将输入延时设为 *d0*，表示

从上一级寄存器到达 D 端所用的时间是 d0，因此从 D 到本级寄存器的延时最大值为 T-d0。

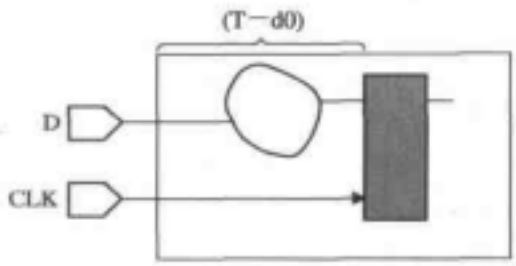


图 3.7　输入延时

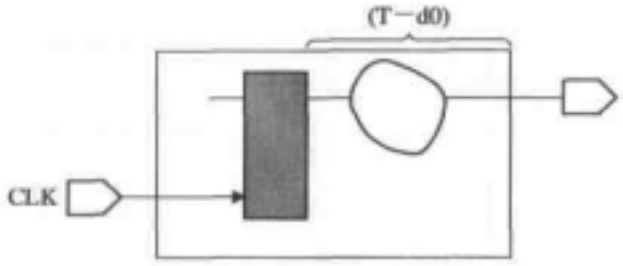


图 3.8　最大输出延时

3) 输出延时

对于输出端的电路，由于 DC 不了解外部的驱动电路的时序，因而无法确定这部分电路的时序要求。set_output_delay 可以解决这一问题。该命令用来设置信号到达外部寄存器所用的时间。例如：

dc_shell> set_output_delay d0 –clock CLK all_outputs()

如图 3.8 所示，设时钟周期为 T，输出延时置为 d0，表示从输出端到达下一个寄存器所需的时间为 d0，因此从本级寄存器送到输出端所用的时间不能大于 T-d0。

4) 驱动

这里所说的驱动是指施加到待综合电路的驱动能力。电路是由外部环境进行驱动的。例如，要设计一个电路 B，它是由电路 A 驱动的。电路 A 的输出有一定范围。比如说，在 5 V 电压下，它输出的电流只能是 1 mA。设计者要告诉 DC，电路 A 的驱动能力是多少。设计规格中的 DC 特性与 AC 特性都包含了这方面的要求。但在实际的综合过程中，这个取值需要设计者的经验来确定。如果取值不恰当，综合出的电路就不能正常工作。例如，设置的驱动过大，综合出的电路 B 的负载可能很大，在实际工作时，外界提供不了这样的驱动能力，信号的变化边沿会很差。

set_drive 命令用特定的驱动阻抗来设置输入端口的驱动强度。例如：

dc_shell>set_drive 200 {Clk}

可以这样理解：在一定电压下，外接电阻跟本电路串联。该电阻越大，则相应的电流越小，驱动能力就越小。如果设为 0，则表明是无限的驱动强度。

也可以用 set_driving_cell 命令来设置。该命令使用驱动单元的驱动阻抗来设置输入端口的驱动强度。例如，可以写成如下语句：

dc_shell>set_driving_cell –lib_cell NI01D1 –library csm06 find (port clk)

这个约束用工艺库中某个单元的输出驱动能力来作为被综合电路输入端的驱动。

5) 负载

如上面所提到的，要设计的电路 B 可能需要驱动其它电路，例如电路 C。电路 C 对电路 B 的输出也有一定要求，例如，要求在 5 V 下 B 输出的电流至少是 1 mA。设计者需要告诉 DC，电流 C 的负载是多少，这样综合出的电路才能给 C 提供足够的驱动。如果负载值取得过小，下级电路可能无法正常工作。如果负载取得太大，会增加上级电路 A 实现的难度。负载的取值由规格决定，但往往也要靠设计者的经验来设置。

set_load 命令可以设置端口(ports)或者线网上的电容负载。

dc_shell>set_load –min 80 * load_of (csm06/IN01D1/A) find (port"ClkCnt[*]")

该约束说明，输出 PORT 端口 ClkCnt[*]的负载等于工艺库中的 IN01D1 单元的输入端 A 的 80 倍。这说明电路 B 能够驱动 80 个反相器。

图 3.9 对驱动和负载进行了说明。

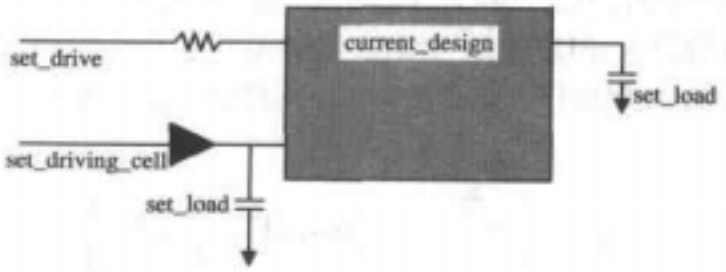


图 3.9　驱动和负载

2．面积约束

面积是芯片成本的一个重要组成部分。芯片越小，成本越低。所以在综合时，常希望芯片的面积尽可能的小。

在工艺库中，对于每个单元的面积都有说明(面积的单位则各有不同)，DC 会根据综合时使用的这些单元估算出芯片的面积。在通常的 ASIC 设计中，面积实际上是用设计中的门数来表示的(使用 2 输出与非门作单位)。而在片上系统(SoC)中，用平方微米来表示面积。用 report_area 命令，可以看到设计的面积。

dc_shell> set_max_area 150

dc_shell> report_area

如果将面积设为 0，如下所示：

dc_shell>set_max_area　0

则 DC 会在时序满足的情况下，尽可能减小面积。

3.3.4　设计规则约束

设计规则通常由制造厂商给出(在工艺文件中进行描述)。

设计规则约束主要包括三种：最大转换时间、最大扇出、最大电容。这些约束用下面的命令进行设置：

set_max_transition：定义整个设计或者某些端口所允许的最大转换时间。

set_max_fanout：定义整个设计或者某些端口的每个驱动单元的最大扇出负载。

set_max_capacitance：设计在整个设计或者某些端口的每个驱动单元的最大电容负载。

下面举例说明这些命令的用法：

dc_shell> set_max_fanout 10 top

dc_shell> set_max_fanout 15 pinA

dc_shell> set_max_transition 1.2 top

dc_shell> set_max_transition 1.0 pinA

3.3.5 其它

在 DC 中，还有一些常用的命令，如下所述。

1. set_dont_touch_network {clock}

该命令禁止往时钟上加入缓冲。

在综合时，如果一个时钟信号驱动了许多门，则这个时钟信号的边沿会很差。DC 对此的解决方案是：为时钟插入 BUFFER，这样就可以驱动更多的门，而且保持时钟信号的边沿陡峭，如图 3.10 所示。但这样可能会使得在不同路径上出现时钟偏移，从而带来一些时序上的问题。

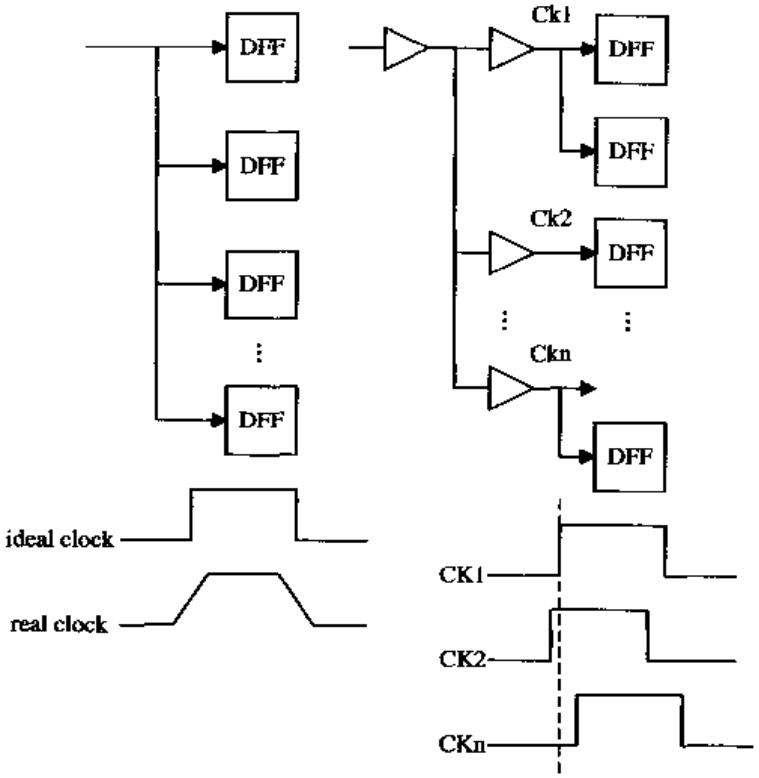


图 3.10 时钟树

DC 推荐的做法是：通过 PR 工具建立时钟树，而不是由 DC 根据驱动的门数来决定。这时，在综合中，我们可以将 clock 看作一个理想的时钟，禁止 DC 插入缓冲。

2. **时序例外(exception)命令**

在电路中，在一个时钟周期内，要将一个数据从一个寄存器传送到下一个寄存器中。如果数据传送超过一个周期，则发生了时序例外。时序例外命令要用到有效起点与有效终点。有效起点包括输入端及寄存器的时钟引脚，有效终点包括输出端及寄存器的数据引脚。

时序例外命令包括：

set_multicycle_path：设置一个多周期路径，从有效起点开始，到有效终点结束。

set_false_path：设置一个无效路径。该命令使得在一些路径上的时序约束无效。例如：

dc_shell> set_false_path -from { ff12 } -to { ff34 }

该命令去掉从 ff12 到 ff34 上的时序约束。

又如：

dc_shell> set_false_path -through {u1/Z u2/Z} -through {u5/Z u6/Z}

该命令去掉所有先经过 u1/Z(或 u2/Z)，然后经过 u5/Z(或 u6/Z)的路径上的约束。

set_max_delay：设置最大延时，从有效起点开始，到有效终点结束。通常用该命令来约束组合路径。

set_min_delay：设置最小延时，从有效起点开始，到有效终点结束。通常用该命令来约束组合路径。

在 DC 中的设置命令需要通过实践加以掌握。图 3.11 对常用的设置命令进行了说明。

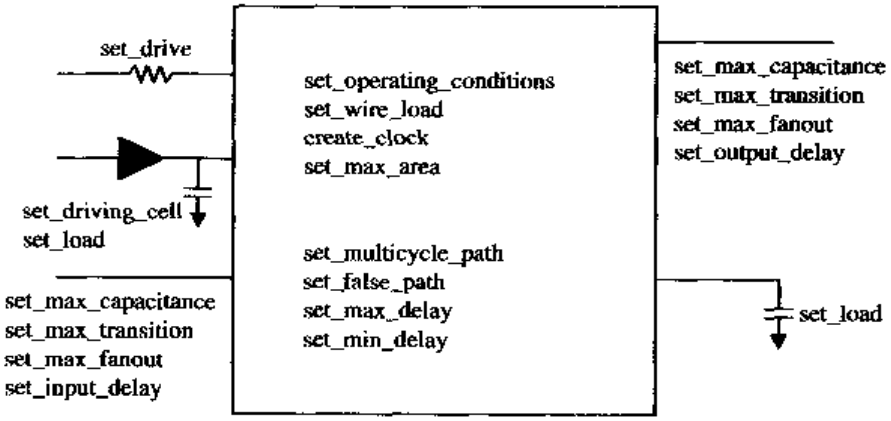


图 3.11　常用设置命令

3.4　综合过程示例

综合的过程如图 3.12 所示。

下面我们用实例来说明这一过程。例如，我们要综合的设计是 example.v，它可用 charters 的 0.6 微米的工艺库进行综合，过程如下。

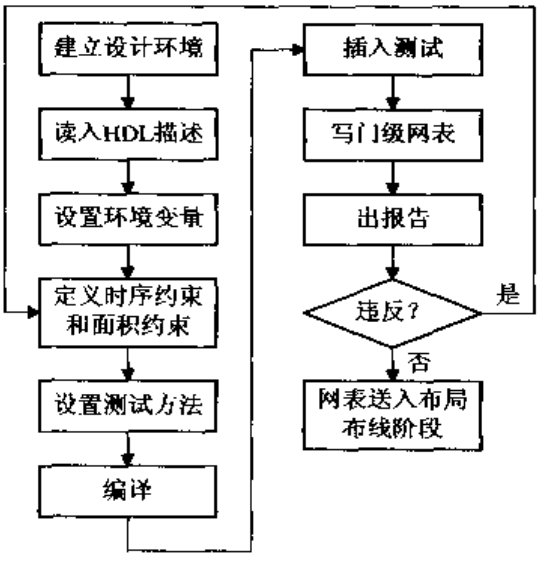


图 3.12　综合的步骤

1．建立设计环境

第一步是要建立设计环境。

target_library = {csm06.db}

link_library = {csm06.db}

search_path = search_path + "/export/home/tools/npu/library/synopsys";

2．读入 HDL 描述

dc_shell> read –f verilog example.v

如果读入多个模块，则首先应该读入的是顶层模块。要将当前的设计设置为顶层模块，使用 current_design 命令。例如：dc_shell> current_design = example

在 DC 中读入设计可以有两种方式：Read、Analyze 和 Elaborate。

在读入 Verilog HDL 和 VHDL 语言的设计描述时，如果设计中不涉及参数的传递，则两种方式效果是一样的。

Read。可以读入包括 Verilog HDL 和 VHDL 在内的其它文件格式，如 EDIF、DB、PLA、EQUATION、MIF、ST 等，而且这些文件格式只能用 Read 命令读入。读入的设计被转译成相应的布尔函数表达式存放在内存中。

Analyze 和 Elaborate。这是一对命令，读入设计时需一起使用。Analyze 对 Verilog HDL 和 VHDL 文件进行编译，并作语法和可综合性的检查，而后将编译的结果保存在由用户指定的目录下；Elaborate 则把该编译结果载入 DC，并转译成相应的布尔函数表达。

这两种读入方式的差别如下：

(1) 在读入设计的格式上，Read 可以载入所有 Synopsys 支持的数据格式，而 Analyze& Elaborate(简称 AE)则只支持 Verilog HDL 和 VHDL 格式。

(2) 在读入的设计中，如果被调用的模块含参数定义，而在被调用的过程中重新被赋值，则必须用 AE 来载入。

3. **定义环境变量**

```
/******** path define ******************/
code_path = "mydesign/ code/ "
netlist_path = "mydesign /syn/netlist/"
report_path = "mydesign/syn/report/"
/*****************************************/
```

4. **定义时序约束和面积约束**

许多时候，综合目标可得到最小的面积，设置这样的约束非常简单，使用 set_max_area 就可以了。例如，可以用如下语句：

```
dc_shell> set_max_area –ignore_tns 1
```

“-ignore tns”选项强制综合工具忽略以前定义的时序约束，从而得到最小的面积。

同样，可以设置时序约束，以得到期望的性能。例如：

```
dc_shell> set_max_delay 0 –from {in} –to {out}
```

约束条件用 set_max_delay 为 0，使延时为最小。这样综合出的网表性能最好，但面积会增加。

在实际综合时，常常要同时给出面积和时序约束。下面给出示例：

```
dc_shell> set_max_area 0
dc_shell> set_max_delay 2 –from {in} –to {out}
```

5. **设置测试方法**

在综合时，可以设置扫描链。有关命令在后面讲述。

6. **编译**

编译命令如下：

dc_shell> compile

7. 插入测试

有关内容在下面讲述。

8. 写门级网表

综合可以用 edif 文件形式保存，也可以用.V 文件形式存放。此外，还需要保存延时信息，以便进行后仿真。延时信息一般放在.sdf 文件中。

生成 edif 的网表文件，采用如下命令：

dc_shell> write –f edif –o example.edf –hierarchy

生成.v 的网表文件，采用如下命令：

dc_shell> write –f verilog –o decode.vs –hierarchy

生成标准延时文件，采用如下命令：

dc_shell> write_sdf –version 2.1 –context verilog decode.sdf

9. 给出报告

使用 report_area 与 report_timing 命令，可以将综合出的面积及时序信息给出。report_constraint 可以用来报告违反约束的路径。

下面举几个例子。

dc_shell> report_area：报告面积。

dc_shell>report_timing -max_paths 100 -delay min -path end：报告所有最短时序路径。

dc_shell>report_timing -max_paths 100 -delay max -path end：报告所有最长时序路径。

dc_shell>report_constraint -all violators：报告所有违反约束的路径。

3.5 综合的 SDF 文件

SDF 文件是一个 ASCII 的文本文件，该文件包含了由综合工具给出的时序信息。EDA 工具可以共享这些时序数据。在一个 SDF 文件中，包括环境、工艺、延时等内容，并包括时序检查(TIMINGCHECK)。下面举出一个实例进行说明。

在 SDF 文件中，第一部分是文件头，一般形式如下：

```
(DELAYFILE
(SDFVERSION "OVI 2.1")
(DESIGN "mymodule")
(DATE "Tue Jun 25 11:05:23 2001")
(VENDOR "xxx")
(PROGRAM "Synopsys Design Compiler cmos")
(VERSION "2000.11-SP1")
(DIVIDER /)
(VOLTAGE 2.70:2.70:2.70)
(PROCESS "NOMINAL")
```

```
(TEMPERATURE 85.00:85.00:85.00)
(TIMESCALE 1ns)
```

接下来，是对各个单元的描述。其形式如下：

```
(CELL
  (CELLTYPE "F153")
  (INSTANCE module1_inst/U206)
  (DELAY
    (ABSOLUTE
    (IOPATH H01 N01 (0.613:0.613:0.613) (0.701:0.701:0.701))
    )
  )
)
(CELL
  (CELLTYPE "L303N2")
  (INSTANCE module1_inst/U373)
  (DELAY
    (ABSOLUTE
    (IOPATH H02 N01 (0.805:0.805:0.805) (1.263:1.263:1.263))
    (IOPATH H03 N01 (0.821:0.821:0.821) (1.318:1.318:1.318))
    (IOPATH H01 N01 (0.467:0.467:0.467) (0.504:0.504:0.504))
    )
  )
)
(CELL
.....
```

在 SDF 文件中，延时可以用绝对值(ABSOLUTE)来表示，也可以用增量(INCREMENT)来表示。增量值与已有的延时值相加，即得到实际的延时。上面的例子中，延时是以绝对值方式表示的。下面给出用增量表示的例子。

```
(CELL (CELLTYPE "DFF")
(INSTANCE B1)
(DELAY (INCREMENT
(IOPATH (posedge clk)
q (-2::1) (3::-3))
(PORT clr (3:4:6) (5:7:9))
))
)
```

在一个 SDF 文件中，包括 IO 通路的延时、互连线的延时。

IOPATH 设置了在一个器件中，从一个输入端口到一个输出端口的延时。图 3.13 给出一个器件的端口示例。要说明从 I1、I2、I3 到 O1 的延时，可以用如下语句：

```
(INSTANCE B1)
(DELAY (ABSOLUTE
(IOPATH (posedege I1) O1 (2:3:4) (4:5:6) (3:5:6))
(IOPATH I2 O1 (2:4:6) (5:6:7) (4:6:7))
(COND I1 (IOPATH I3 O1 (2:4:5) (3:4:5) (1:2:3))
)
```

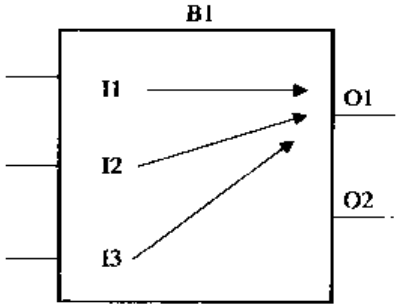


图 3.13　器件的端口示例

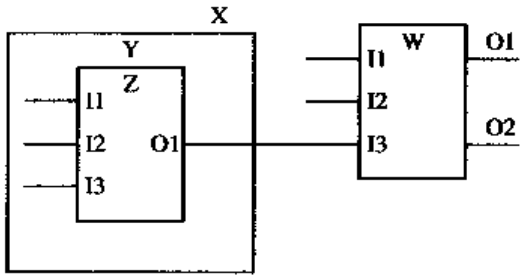


图 3.14　组件间的互连

互连延时设置了在器件间连线上的延时。例如，要描述在图 3.14 中，O1 到 I3 之间的延时，可以用如下方式：

```
(INSTANCE X)
(DELAY (ABSOLUTE
(INTERCONNECT Y.kZ.O1 W.I3
(5:6:7) (5.5:6: 6.4))
))
```

在 SDF 文件中，可以对信号宽度、建立时间与保持时间进行检查。相应的语句如下：

```
(TIMINGCHECK
    (WIDTH (posedge H02) (2.477:2.477:2.477))
    (WIDTH (negedge H02) (2.477:2.477:2.477))
    (SETUP (posedge H01) (posedge H02) (0.541:0.541:0.541))
    (SETUP (negedge H01) (posedge H02) (0.138:0.138:0.138))
    (HOLD (posedge H01) (posedge H02) (0.031:0.031:0.031))
    (HOLD (negedge H01) (posedge H02) (0.654:0.654:0.654))
 )
```

3.6　关 于 测 试

语言描述的设计可以利用仿真器通过监测内部和外部的信号线进行检错和验证，而对制成的硬件电路，如果仅单纯满足设计的功能要求，则其检错和验证的自由度和灵活性都将大为减少，尤其在新产品开发之初。因此为保证电路的可测性和易测性，在设计电路时需要考虑将来的电路测试方案。在第 1 章中已介绍了可测性设计的概念。

测试分功能测试和结构测试两种。功能测试基于系统的定义，对电路进行测试，以保证所有定义功能的遍历和验证；结构测试则是关注电路的结构，目的是定位物理故障的发生位置，因此出发点和功能测试不同。

Synopsys 提供了 BSD Compiler，DFT Compiler，Test Compiler 等工具，用于可测性设计。这些工具都可以在 dc_shell 下运行。

BSD Compiler 可以加入边界扫描单元和 TAP 控制器。边界扫描单元与 TAP 控制器都来自 Synopsys 的 designware 库(这是 Synopsys 的一个 IP 库)。

DFT Compiler 可以检查设计规则，加入扫描链，检查错误覆盖率。

Test Compiler 是一个自动向量生成(ATPG)工具，非常适用于较小的设计。对于较大设计，Synopsys 推荐使用 TetraMax 工具。

下面给出生成扫描电路的一个综合脚本。

```
include /export/home/synopsys_setup_my.inc
read -f verilog example.v
current_design example
test_default_scan_style = "multiplexed_flip_flop"
link
set_test_hold ZERO load
test_disable_find_best_scan_out = true
create_port -direction in {scanin, scanen}
set_signal_type "test_scan_in" find (port, scanin)
set_signal_type "test_scan_out" find (port, shiftout)
set_signal_type "test_scan_enable" find (port, scanen)
report_test -port
set_scan_segment shiftrg -access {test_scan_in shiftblk/si, test_scan_out shiftblk/so} \
    -contains {shiftblk/q_reg[0], shiftblk/q_reg[1], shiftblk/q_reg[2], shiftblk/q_reg[3]}
preview_scan -show segments
set_scan_element false {shiftblk/q_reg[0], shiftblk/q_reg[1], shiftblk/q_reg[2], shiftblk/q_reg[3]}
compile -scan
preview_scan -show segments
insert_scan
report_test -scan_path
report_test -port
create_schematic -hierarchy
plot -o example.ps -hier
write -f verilog -o example.vs -hier
check_test
create_test_patterns
write_test -format verilog
exit
```

3.7 面向综合的设计

代码书写风格并不仅仅跟综合有关，但跟综合的关系最密切，所以把这一节放在了综合这一章中讨论。

一个系统可能需要划分多个层次，同时可能需要多个工程师的合作，因此规范代码的书写风格十分重要。代码书写风格的统一便于设计的管理，便于设计工程师之间的交流。

书写 RTL 代码时，什么才能算是好的风格？

(1) 可综合。相信阅读这本书的许多人都有编写软件代码的经历，需要强调的是，描述硬件和编写软件是有区别的。用硬件描述语言编程时，应时刻牢记所写的代码能否反映期望的硬件及其功能，而且电路结构是可实现的。

为保证设计的可综合，代码中的指令和数据类型必须为综合工具所支持，同时代码的结构也必须使综合得到的硬件功能和代码所体现的功能一致。

(2) 简洁清晰，易于理解。简洁清晰并不能保证一部小说成为一部杰作，却可以保证一段代码成为杰作。要记住，你写的代码是给别人看的，因而最易懂的写法也就是效率最高的方法。

使用一致的命名规则。

(3) 注释清楚。

(4) 模块化程度高，易于修改、移植和复用。

例如，要做到易于修改和复用，就需要将代码中的常数写成参数化的形式。在 Verilog 中，可以使用 parameter 来表示常数。

请看下面的例子：

```
parameter ZERO = 2'b00;
parameter A_AND_B = 2'b01;
parameter A_OR_B = 2'b10;
parameter ONE = 2'b11;
always @ (OPCODE or A or B)
begin
if (OPCODE==ZERO) OP_OUT=1'b0;
else if(OPCODE==A_AND_B) OP_OUT=A&B;
      else if(OPCODE==A_OR_B) OP_OUT=A|B;
           else OP_OUT=1'b1;
end
```

(5) 在模块实例化时，为清楚起见，要使用显式的端口说明。

例如，要写成

```
block block_1 ( .signal_a(signal_a),.signal_b(signal_b) );
```

(6) 为了避免很大的路径延时，不要使用特别长的嵌套的 if 结构。

例如，左边是一个有很多嵌套 if 结构的代码。这种结构的代码不仅延时长，而且非常难懂。如果把这种结构改造为 case 结构，就好多了。表 3.2 为不同风格代码的比较。

表 3.2　不同风格代码比较

坏风格	好风格(用 case 语句)

module nested_if (ADDR_A, ADDR_B. ADDR_C, ADDR_D, RESET, CLK, DEC_Q);
input [1:0] ADDR_A ;
input [1:0] ADDR_B ;
input [1:0] ADDR_C ;
input [1:0] ADDR_D ;
input RESET, CLK ;
output [5:0] DEC_Q ;
reg [5:0] DEC_Q ;
// Nested If Process
always @ (posedge CLK)
begin
if (RESET == 1'b1)
begin
if (ADDR_A == 2'b00)
begin
DEC_Q[5:4] <= ADDR_D;
DEC_Q[3:2] <= 2'b01;
DEC_Q[1:0] <= 2'b00;
if (ADDR_B == 2'b01)
begin
DEC_Q[3:2] <= ADDR_A + 1'b1;
DEC_Q[1:0] <= ADDR_B + 1'b1;
if (ADDR_C == 2'b10)
begin
DEC_Q[5:4] <= ADDR_D + 1'b1;
if (ADDR_D == 2'b11)
DEC_Q[5:4] <= 2'b00;
end
else
DEC_Q[5:4] <= ADDR_D;
end
end
else
DEC_Q[5:4] <= ADDR_D;
DEC_Q[3:2] <= ADDR_A;
DEC_Q[1:0] <= ADDR_B + 1'b1;
end
else
DEC_Q <= 6'b000000;
end
endmodule
	希望勤快的读者把它写出来，作为一个练习

(7) 资源共享(Resource Sharing)。资源共享的主要思想是通过数据缓冲器或多路选择的方法来共享数据通路中的工作单元。例如图 3.15 中(a)和(b)两种电路结构实现的是相同的功能，但在电路结构上显然图(a)要比图(b)节省器件资源。表 3.3 为资源共享与非共享的代码比较

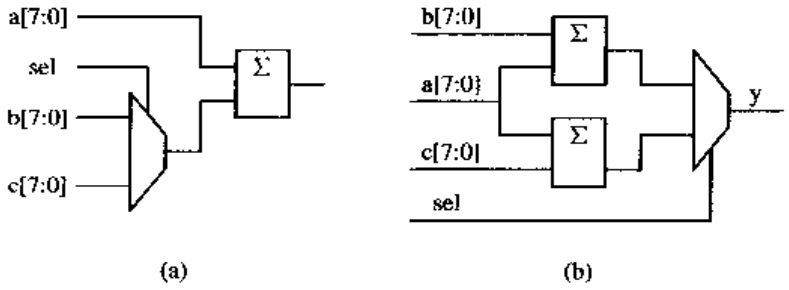


图 3.15　资源共享与非共享的电路比较

(a) 资源共享结构；(b) 资源非共享的结构

表 3.3　资源共享与非共享的代码比较

资源非共享	资源共享

module res_sharing (A, B, C, Sel, Y);
input Sel;
input [7:0] A, B, C;
output [7:0] Y;
reg [7:0] Y;
 	
always @(A or B or C or Sel)
begin
if (Sel)
Z1 <= A + B;
else
Z1 <= A + C;
end
endmodule
	module res_sharing (A, B, C, Sel, Y);
	input Sel;
	input [7:0] A, B, C;
	output [7:0] Y;
	reg [7:0] Y;
	
	wire addtemp;
	assign addtemp= COND_1?B:C;
	
	always @(A or B or C or D or Sel)
	begin
	Z1 <= A + addtemp;
	end
	endmodule

在这个例子中，如果将加法器共享，实现 A+B 和 A+C，可以节省面积。需要说明的是，许多综合器能自动将左边的代码综合为资源共享的结构。

资源共享的典型运用场合是“+、-、*、/”运算。由于综合工具对“*、/”运算综合的效果不是很好，所以经常见到的是“+、-”两种运算的资源共享。资源共享存在有两个条件：一是这些操作数存在于同一个 always 语句中；二是这些操作数必须存在于同一条件赋值语句的不同分支上。类似这样的情况还有很多。例如，图 3.16 给出了使用共享技术与非共享技术的电路结构。

如果可能，可以让一个模块被多个功能模块使用。这样可以减少门数，对布线也有好处。当然，为了提高速度，对于关键路径不要使用该技术。一般来说，在算术单元与关键路径中都不采用资源复用。

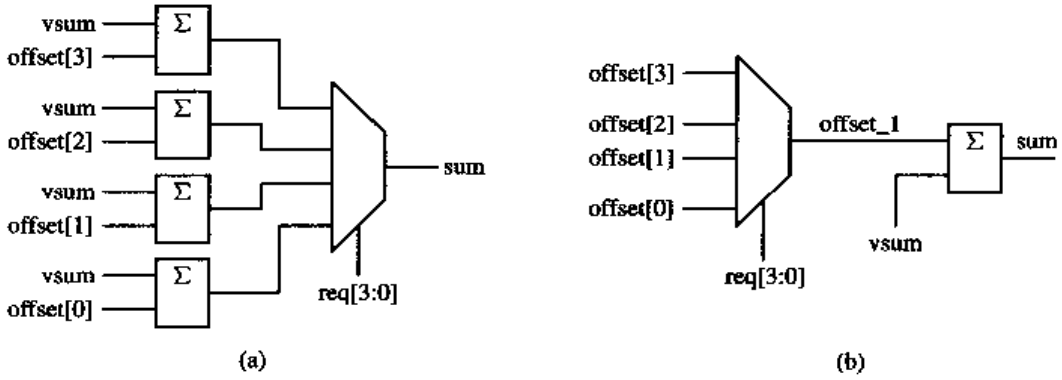


图 3.16　使用资源共享与非共享的电路结构

(a) 共享；(b) 非共享

(8) 避免使用门限时钟。门限时钟可能会引入短脉冲干扰，增加时钟延时、时钟扭曲，以及其它的后果。我们首先给出一个采用门限时钟的例子，如下所示。

```
module gate_clock(IN1, IN2, DATA, CLK, LOAD, OUT1) ;
input IN1 ;
input IN2 ;
input DATA ;
input CLK ;
input LOAD ;
output OUT1 ;
reg OUT1 ;
wire GATECLK ;
assign GATECLK = (IN1 & IN2 & CLK);   //门限时钟
always @(posedge GATECLK)
begin
if (LOAD == 1'b1)
OUT1 = DATA;
end
endmodule
```

综合结果如图 3.17 所示。

对代码进行如下改动：

```
module clock_enable (IN1, IN2, DATA,
CLK, LOAD, DOUT);
input IN1, IN2, DATA;
input CLK, LOAD;
output DOUT;
wire ENABLE;
```

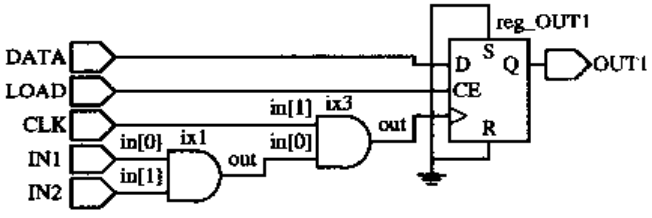


图 3.17　门限时钟的例子

```
reg DOUT;
assign ENABLE = IN1 & IN2 & LOAD;
always @(posedge CLK)
begin
if (ENABLE)
DOUT <= DATA;
end
endmodule
```

综合结果如图 3.18 所示。

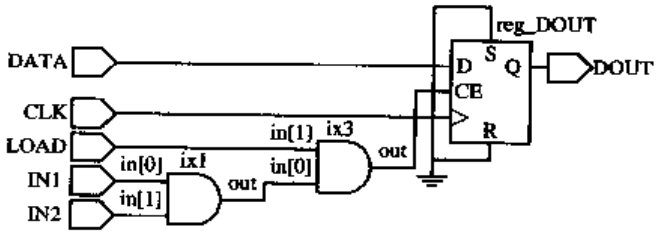


图 3.18　利用时钟使能取代门限时钟

下面给出一个具有较好风格的编码：

```
// Copyright (c) 1999, xx.com
// xx Confidential Proprietary
//
// ----------------------------------------------------------------------
// FILE NAME : prescaler.v
// TYPE : module
// AUTHOR : yy
// AUTHOR'S EMAIL : yy@xx.com
// ----------------------------------------------------------------------
// Release history
// VERSION Date AUTHOR DESCRIPTION
// 1.0 12 Sep initial version
// 2.0 Nov 98 Updated for compatibility
// 2.1 Oct 99 Cleaned up prescaler bypass.
// ----------------------------------------------------------------------
//目的 : 输入时钟16分频
// ----------------------------------------------------------------------
//复位策略: 异步，低电平有效
//时钟域: system_clock, clock_in
//测试特性 : Prescaler is bypassed when scan_mode is asserted
//扫描方法: Mux-D
module prescaler(
clock_out, clock_out_b,
clock_in, system_clock,
scan_mode, reset_b
);
//端口
input clock_in; // 32 MHz clock
input system_clock; // system clock
```

```
input scan_mode; // scan mode clock
input reset_b; // active low hard reset,

output clock_out; // input clock divided by 16
output clock_out_b; // input clock divided by 16 and inverted
//端口信号说明
reg [3:0] count; // counter to make clock divider
reg clock_out; // input clock divided by 16
reg clock_out_b; // input clock divided by 16 and inverted
//局部变量

//实现4位计数器，count[3] is the divide by 16
always @(posedge clock_in or negedge reset_b)
if (!reset_b)
count <= 4'b0000; // 复位计数器
else if (count == 4'b1111)
count <= 4'b0000; // 计数值循环
else
count <= count + 4'b0001; // 计数值递增
// Bypass the prescaler during scan testing. It guarantees
// that the mux will not be optimized away which could
// result in a glitchy test clock.
// Also make sure that the clock_out and clock_out_b are active
// high clocks during scan testing. This ensures that flops
// connected to clock_out and clock_out_b are all on the rising
// edge of the system clock for test purposes.
always @(scan_mode or system_clock or count)
case (scan_mode) //insert appropriate synthesis mux inference
//directive here
1'b0 : begin
clock_out = count[3];
clock_out_b = !count[3];
end // normal operation clock assign
1'b1 : begin
clock_out = system_clock;
clock_out_b = system_clock;
end // scan mode clock assign
endcase // scan_mode_clock
endmodule // prescaler
```

说明 1：在代码中，为清晰起见，最好每个端口都分别声明，如表 3.4 所示。

说明 2：如果有必要，可以在程序中的进程、函数和例程加上标注。这样可以使代码更易读。例如：

```
always @ (posedge CLK or posedge RST)
begin: ASYNC_FF
module d_register (CLK, DATA, Q);
input CLK;
input DATA;
output Q;
reg Q;
always @ (posedge CLK)
begin: My_D_Reg
Q <= DATA;
end
endmodule
```

表 3.4　不同风格的代码比较

好的风格	不好的风格
input a; input b;	input a, input b; 或者 input a, b;

需要说明的是，FPGA 的设计有别于 ASIC。在 ASIC 下，好的设计风格可能并不适用于 FPGA。

(9) 代码中不要带延时信息(当然，行为级描述与 testbench 例外)。例如，像 #5 ns 这样的声明要去掉。

(10) 进行数学运算时，有好的顺序及分组。好的顺序及分组可能影响综合结果。例如：

ADD = A1 + A2 + A3 + A4;

ADD = (A1 + A2) + (A3 + A4);

第一个表达式会综合出串行的三个加法器。而第二个表达式综合出两个并行的加法器，它们再与第三个加法器串行。当四个加数同时有效时，显然第二种实现更快一些。

需要说明的是，为了符合时序要求，综合工具能重新构造算术单元。但我们还是应该按所需的结果书写代码。

(11) 代码中不要带初始化赋值语句。

例如：

wire SUM=1'b0;

因为综合工具会忽略掉这种初始化的语句，所以可能导致综合前后仿真结果不一致。

3.7.1　速度与面积的优化：16 位桶形移位寄存器

综合过程要在速度与面积之间进行折衷考虑。“资源共享”属于面积的优化技术。在设计中，牺牲一些硅片面积，往往会增加速度。如表 3.5 所示的一个 16 位桶形移位寄存器，其左边是优化前的代码，右边是优化后的代码。优化代码使用了两级复用器，其速度得到提高。复用器共 32 个 4 选 1 复用器。第一级，对输入数据进行 0，1，2 或 3 位的循环移位；第二级，对输入数据进行 0，4，8，12 位的循环移位。

表 3.5　优化前后的代码比较

优化前的桶形移位寄存器	优化后的桶形移位寄存器

module barrel_org (S, A_P, B_P);
input [3:0] S;
input [15:0] A_P;
output [15:0] B_P;
reg [15:0] B_P;
 	
always @ (A_P or S)
begin
case (S)
4'b0000 : // Shift by 0
begin
B_P <= A_P;
end
4'b0001 : // Shift by 1
begin
B_P[15] <= A_P[0];
B_P[14:0] <= A_P[15:1];
end
4'b0010 : // Shift by 2
begin
B_P[15:14] <= A_P[1:0];
B_P[13:0] <= A_P[15:2];
end
4'b0011 : // Shift by 3
begin
B_P[15:13] <= A_P[2:0];
B_P[12:0] <= A_P[15:3];
end
4'b0100 : // Shift by 4
begin
B_P[15:12] <= A_P[3:0];
B_P[11:0] <= A_P[15:4];
end
4'b0101 : // Shift by 5
begin
B_P[15:11] <= A_P[4:0];
B_P[10:0] <= A_P[15:5];
end
4'b0110 : // Shift by 6
begin
B_P[15:10] <= A_P[5:0];
B_P[9:0] <= A_P[15:6];
end
4'b0111 : // Shift by 7
	module barrel (S, A_P, B_P);
	input [3:0] S;
	input [15:0] A_P;
	output [15:0] B_P;
	reg [15:0] B_P;
	wire [1:0] SEL1, SEL2;
	reg [15:0] C;
	assign SEL1 = S[1:0];
	assign SEL2 = S[3:2];
	always @ (A_P or SEL1)
	begin
	case (SEL1)
	2'b00 : // Shift by 0
	begin
	C <= A_P;
	end
	2'b01 : // Shift by 1
	begin
	C[15] <= A_P[0];
	C[14:0] <= A_P[15:1];
	end
	2'b10 : // Shift by 2
	begin
	C[15:14] <= A_P[1:0];
	C[13:0] <= A_P[15:2];
	end
	2'b11 : // Shift by 3
	begin
	C[15:13] <= A_P[2:0];
	C[12:0] <= A_P[15:3];
	end
	default :
	C <= A_P;
	endcase
	end
	always @ (C or SEL2)
	begin
	case (SEL2)
	2'b00 : // Shift by 0
	begin
	B_P <= C;
	end
	2'b01 : // Shift by 4
	begin
	B_P[15:12] <= C[3:0];
	B_P[11:0] <= C[15:4];
	end

续表

优化前的桶形移位寄存器	优化后的桶形移位寄存器
begin B_P[15:9] <= A_P[6:0]; B_P[8:0] <= A_P[15:7]; End 4'b1000 : // Shift by 8 begin B_P[15:8] <= A_P[7:0]; B_P[7:0] <= A_P[15:8]; end 4'b1001 : // Shift by 9 begin B_P[15:7] <= A_P[8:0]; B_P[6:0] <= A_P[15:9]; end 4'b1010 : // Shift by 10 begin B_P[15:6] <= A_P[9:0]; B_P[5:0] <= A_P[15:10]; end 4'b1011 : // Shift by 11 begin B_P[15:5] <= A_P[10:0]; B_P[4:0] <= A_P[15:11]; end 4'b1100 : // Shift by 12 begin B_P[15:4] <= A_P[11:0]; B_P[3:0] <= A_P[15:12]; end 4'b1101 : // Shift by 13 begin B_P[15:3] <= A_P[12:0]; B_P[2:0] <= A_P[15:13]; end 4'b1110 : // Shift by 14 begin B_P[15:2] <= A_P[13:0]; B_P[1:0] <= A_P[15:14]; end 4'b1111 : // Shift by 15 begin B_P[15:1] <= A_P[14:0]; B_P[0] <= A_P[15]; end default : B_P <= A_P; endcase end endmodule	2'b10 : // Shift by 8 begin B_P[7:0] <= C[15:8]; B_P[15:8] <= C[7:0]; end 2'b11 : // Shift by 12 begin B_P[3:0] <= C[15:12]; B_P[15:4] <= C[11:0]; end default : B_P <= C; endcase end endmodule

3.7.2 Net 类型与 Register 类型

在 Verilog 中，信号分为 Net 类型与 Register 类型两种。需要说明的是，Register 类型不一定能综合出寄存器。例如以下代码：

```
module example(acr,bar,fra,trq,sqp);
input acr,bar,fra;
output trq,sqp;
reg trq,sqp;

always @(bar or acr or fra)
begin
trq<=bar&acr;
sqp<=trq|fra;
end
endmodule
```

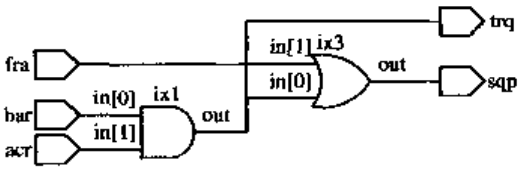


图 3.19 Register 类型的综合结果

综合结果如图 3.19 所示。

如果想要综合出寄存器，则 always 中的敏感列表应为时钟边沿，如下所示：

```
module example(clk,acr,bar,fra,trq,sqp);
input acr,bar,fra;
input clk;

output trq,sqp;
reg trq,sqp;

always @(posedge clk)
begin
trq<=bar&acr;
sqp<=trq|fra;
end
endmodule
```

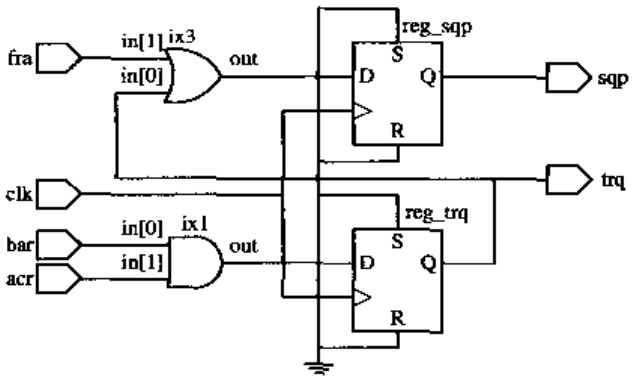


图 3.20 Register 类型的综合结果

综合结果如图 3.20 所示。

3.7.3 if 语句和 Case 语句的综合

if 语句中的信号常作为控制信号。如果 if 语句写完整，则 if 语句中的信号可以综合为复用器的控制信号；否则，会综合出锁存器。例如下面的代码：

```
module if1 (a,b,f);
input a,b;
output f;
```

```
reg f;

always @(a or b) begin
    if (a)
        f = b;
end
endmodule
```

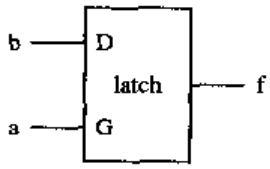


图 3.21　if1 的综合结果

这段代码的综合结果如图 3.21 所示。

接下来，我们将 if 语句写完整。

```
module if2 (a,b,c,f);
input a,b,c;
output f;
reg f;

always @(a or b or c) begin
    if (a)
        f = b;
    else
        f = c;
end
endmodule
```

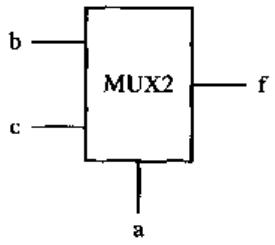


图 3.22　if2 的综合结果

这段代码的综合结果如图 3.22 所示。

在条件表达式中，if-else 语句描述的电路结构具有优先级，而 Case 语句描述的通常没有优先级。通常，对于复杂的解码使用 Case 语句声明，而对速度关键路径使用 if 语句。表 3.6 是具有优先级的复用器的实现。

表 3.6　复用器实现比较

用 if 实现的复用器	用 Case 实现的复用器

module if_example (A, B, C, D, SEL, MUX_OUT);
input A, B, C, D;
input [1:0] SEL;
output MUX_OUT;
reg MUX_OUT;
always @ (A or B or C or D or SEL)
begin
if (SEL == 2'b00)
MUX_OUT = A;
else if (SEL == 2'b01)
MUX_OUT = B;
else if (SEL == 2'b10)
	module case_example (A, B, C, D, SEL, UX_OUT);
	input A, B, C, D:
	input [1:0] SEL;
	output MUX_OUT;
	reg MUX_OUT;
	always @ (A or B or C or D or SEL)
	begin
	case (SEL)
	2'b00:
	MUX_OUT = A;
	2'b01:
	MUX_OUT = B;

续表

用 if 实现的复用器	用 Case 实现的复用器
MUX_OUT = C; else if (SEL == 2'b11) MUX_OUT = D; else MUX_OUT = 0; end endmodule	2'b10: MUX_OUT = C; 2'b11: MUX_OUT = D; default: MUX_OUT = 0; endcase end endmodule

对复杂的解码常用 Case 语句，因为它的结构较为清晰。表 3.7 给出了 if 语句和 Case 语句的比较。

表 3.7　if 语句和 Case 语句的比较

if 语句描述	Case 语句描述

请试着把它描述出来	module rom_rtl(ADDR, DATA) ;
	input [3:0] ADDR ;
	output [3:0] DATA ;
	reg [3:0] DATA ;
	// A memory is implemented
	// using a case statement
	always @(ADDR)
	begin
	case (ADDR)
	4'b0000 : DATA = 4'b0000 ;
	4'b0001 : DATA = 4'b0001 ;
	4'b0010 : DATA = 4'b0010 ;
	4'b0011 : DATA = 4'b0100 ;
	4'b0100 : DATA = 4'b1000 ;
	4'b0101 : DATA = 4'b1000 ;
	4'b0110 : DATA = 4'b1100 ;
	4'b0111 : DATA = 4'b1010 ;
	4'b1000 : DATA = 4'b1001 ;
	4'b1001 : DATA = 4'b1001 ;
	4'b1010 : DATA = 4'b1010 ;
	4'b1011 : DATA = 4'b1100 ;
	4'b1100 : DATA = 4'b1001 ;
	4'b1101 : DATA = 4'b1001 ;
	4'b1110 : DATA = 4'b1101 ;
	4'b1111 : DATA = 4'b1111 ;
	endcase
	end
	endmodule

在实际设计中，可以将这两种语句结合起来，如表 3.8 所示。当 s2 到达时间晚于 s1 与 s3 时，第二种表示方法的速度更快一些。

表 3.8　if 语句和 Case 语句结合的应用

所有数据信号有同样的延时	使信号 s2 得到加速

case(state)
　s1　　Z <= a;
　s2　　Z <= b;
　s3　　Z <= c;
　default:　Z <= d;
endcase
	case(state)
	　s1:　tmp <= a;
	　s3:　tmp <= c;
	　default: tmp <= d;
	endcase
	if (state == s2)
	　Z = b;
	else
	　Z = tmp;

3.7.4　阻塞赋值与非阻塞赋值

对于时序逻辑，一般要使用非阻塞赋值(对那些使用阻塞赋值的中间变量除外)；而对于组合逻辑，需要使用阻塞赋值。一般而言，非阻塞赋值语句用于 always 过程内部，而阻塞赋值语句用于 always 过程外部。阻塞赋值也可以用于 always 过程内部，但一般不推荐这种写法。

在 always 过程内部，阻塞赋值的综合结果与非阻塞赋值不同。下面用实例进行说明。

采用非阻塞赋值的代码如下：

```
always @(posedge clk) begin
  rega <= data;
  regb <= rega;
end
```

其综合结果如图 3.23 所示。

采用阻塞赋值的代码如下：

```
always @(posedge clk) begin
  reg1 = data;
  reg2 = reg1;
end
```

其综合结果如图 3.24 所示。

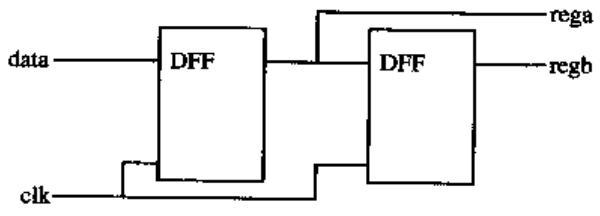


图 3.23　非阻塞赋值的综合结果

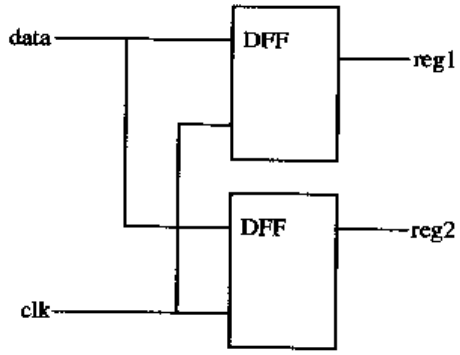


图 3.24　阻塞赋值的综合结果

如果在一个代码中，对一个信号同时使用了阻塞赋值与非阻塞赋值，则发生错误。例如：

count=preset+1;

count<=mask;

编译会报错，因为 count 已经在前面用阻塞赋值给过值了。表 3.9 示出了不同风格代码的比较。

表 3.9 不同风格代码的比较

坏风格	好风格

always @(posedge clk) begin
rega = data;
regb = rega;
	always @(posedge clk) begin
	regb <= rega;
	rega <= data;
	end
	说明：
	在边沿敏感模块中，要使用非阻塞赋值(<=)，而不要使用阻塞赋值(=)，否则可能引起前仿真与后仿真的不匹配

3.7.5 状态机的编码

状态机编码方式，包括二进制编码、枚举编码和独热编码等。

相比较而言，独热编码更适用于 FPGA 结构。设计者可能给每个状态一个触发器，减少组合逻辑的复杂性。独热编码适用于大的基于 FPGA 的实现中。对于小的状态机(小于 8 个状态)，二进制编码更为有效。为了改善设计性能，设计者可以将大的状态机(大于 32 个)分成几个小的状态机，对每一个状态机使用合适的编码风格。

在本节给出了三个设计示例，以说明三种编码方法(二进制，枚举，独热)，如表 3.10 所示。

表 3.10 状态机的三种编码方法

二进制编码	枚举编码	独热编码
module binary (CLOCK, RESET, A, B, C, D, E, SINGLE, MULTI, CONTIG); input CLOCK, RESET; input A, B, C, D, E; output SINGLE, MULTI, ONTIG; reg SINGLE, MULTI, CONTIG; // Declare the symbolic names for states parameter [2:0] S1 = 3'b001, S2 = 3'b010, S3 = 3'b011, S4 = 3'b100, S5 = 3'b101,	module enum (CLOCK, RESET, A, B, C, D, E, SINGLE, MULTI, CONTIG); input CLOCK, RESET; input A, B, C, D, E; output SINGLE,MULTI,CONTIG; reg SINGLE, MULTI, CONTIG; // Declare the symbolic names for states parameter [2:0] S1 = 3'b000, S2 = 3'b001, S3 = 3'b010, S4 = 3'b011, S5 = 3'b100,	module one_hot (CLOCK, RESET, A, B, C, D, E, SINGLE, MULTI, CONTIG); input CLOCK, RESET; input A, B, C, D, E; output SINGLE,MULTI,CONTIG; reg SINGLE, MULTI, CONTIG; // Declare the symbolic names for states parameter [6:0] S1 = 7'b0000001, S2 = 7'b0000010, S3 = 7'b0000100, S4 = 7'b0001000, S5 = 7'b0010000,

续表

二进制编码	枚举编码	独热编码

S6 = 3'b110,
S7 = 3'b111;
// Declare current state and next state variables
reg [2:0] CS;
reg [2:0] NS;
// state_vector CS
always @ (posedge CLOCK or posedge RESET)
begin
if (RESET == 1'b1)
CS = S1;
else
CS = NS;
end
always @ (CS or A or B or C or D or D or E)
begin
case (CS)
S1 :
Begin
MULTI = 1'b0;
CONTIG = 1'b0;
SINGLE = 1'b0;
if (A && ~B && C)
NS = S2;
else if (A && B && ~C)
NS = S4;
else
NS = S1;
end
S2 :
begin
MULTI = 1'b1;
CONTIG = 1'b0;
SINGLE = 1'b0;
if (!D)
NS = S3;
else
NS = S4;
end
S3 :
begin
MULTI = 1'b0;
CONTIG = 1'b1;
SINGLE = 1'b0;
	S6 = 3'b101,
	S7 = 3'b110;
	// Declare current state and next state variables
	reg [2:0] CS;
	reg [2:0] NS;
	// state_vector CS
	always @ (posedge CLOCK or posedge RESET)
	begin
	if (RESET == 1'b1)
	CS = S1;
	else
	CS = NS;
	end
	always @ (CS or A or B or C or D or D or E)
	begin
	case (CS)
	S1 :
	begin
	MULTI = 1'b0;
	CONTIG = 1'b0;
	SINGLE = 1'b0;
	if (A && ~B && C)
	NS = S2;
	else if (A && B && ~C)
	NS = S4;
	else
	NS = S1;
	end
	S2 :
	begin
	MULTI = 1'b1;
	CONTIG = 1'b0;
	SINGLE = 1'b0;
	if (!D)
	NS = S3;
	else
	NS = S4;
	end
	S3 :
	begin
	MULTI = 1'b0;
	CONTIG = 1'b1;
	SINGLE = 1'b0;
		S6 = 7'b0100000,
		S7 = 7'b1000000;
		// Declare current state and next state variables
		reg [2:0] CS;
		reg [2:0] NS;
		// state_vector CS
		always @ (posedge CLOCK or posedge RESET)
		begin
		if (RESET == 1'b1)
		CS = S1;
		else
		CS = NS;
		end
		always @ (CS or A or B or C or D or D or E)
		begin
		case (CS)
		S1 :
		begin
		MULTI = 1'b0;
		CONTIG = 1'b0;
		SINGLE = 1'b0;
		if (A && ~B && C)
		NS = S2;
		else if (A && B && ~C)
		NS = S4;
		else
		NS = S1;
		end
		S2 :
		begin
		MULTI = 1'b1;
		CONTIG = 1'b0;
		SINGLE = 1'b0;
		if (!D)
		NS = S3;
		else
		NS = S4;
		end
		S3 :
		begin
		MULTI = 1'b0;
		CONTIG = 1'b1;
		SINGLE = 1'b0;

续表

进制编码	枚举编码	独热编码

if (A || D)
NS = S4;
else
NS = S3;
end
S4 :
begin
MULTI = 1'b1;
CONTIG = 1'b1;
SINGLE = 1'b0;
if (A && B && ~C)
NS = S5;
else
NS = S4;
end
S5 :
begin
MULTI = 1'b1;
CONTIG = 1'b0;
SINGLE = 1'b0;
NS = S6;
end
S6 :
begin
MULTI = 1'b0;
CONTIG = 1'b1;
SINGLE = 1'b1;
if (!E)
NS = S7;
else
NS = S6;
end
S7 :
begin
MULTI = 1'b0;
CONTIG = 1'b1;
SINGLE = 1'b0;
if (E)
NS = S1;
else
NS = S7;
end
endcase
end
endmodule
	if (A || D)
	NS = S4;
	else
	NS = S3;
	end
	S4 :
	begin
	MULTI = 1'b1;
	CONTIG = 1'b1;
	SINGLE = 1'b0;
	if (A && B && ~C)
	NS = S5;
	else
	NS = S4;
	end
	S5 :
	begin
	MULTI = 1'b1;
	CONTIG = 1'b0;
	SINGLE = 1'b0;
	NS = S6;
	end
	S6 :
	begin
	MULTI = 1'b0;
	CONTIG = 1'b1;
	SINGLE = 1'b1;
	if (!E)
	NS = S7;
	else
	NS = S6;
	end
	S7 :
	begin
	MULTI = 1'b0;
	CONTIG = 1'b1;
	SINGLE = 1'b0;
	if (E)
	NS = S1;
	else
	NS = S7;
	end
	endcase
	end
	endmodule
		if (A || D)
		NS = S4;
		else
		NS = S3;
		end
		S4 :
		begin
		MULTI = 1'b1;
		CONTIG = 1'b1;
		SINGLE = 1'b0;
		if (A && B && ~C)
		NS = S5;
		else
		NS = S4;
		end
		S5 :
		begin
		MULTI = 1'b1;
		CONTIG = 1'b0;
		SINGLE = 1'b0;
		NS = S6;
		end
		S6 :
		begin
		MULTI = 1'b0;
		CONTIG = 1'b1;
		SINGLE = 1'b1;
		if (!E)
		NS = S7;
		else
		NS = S6;
		end
		S7 :
		begin
		MULTI = 1'b0;
		CONTIG = 1'b1;
		SINGLE = 1'b0;
		if (E)
		NS = S1;
		else
		NS = S7;
		end
		endcase
		end
		endmodule

3.7.6 使用流水线

设计者可以使用流水线来改善器件性能。流水线通过将较长的数据通路分开，增加了系统性能。该方法允许更快的时钟周期，因而增加了数据流量(其代价是增加了数据延时)。图 3.25 说明了流水线操作的优势。

流水化之前的设计，系统只能用较慢的时钟。流水化之后的设计，系统时钟可以大大加快。

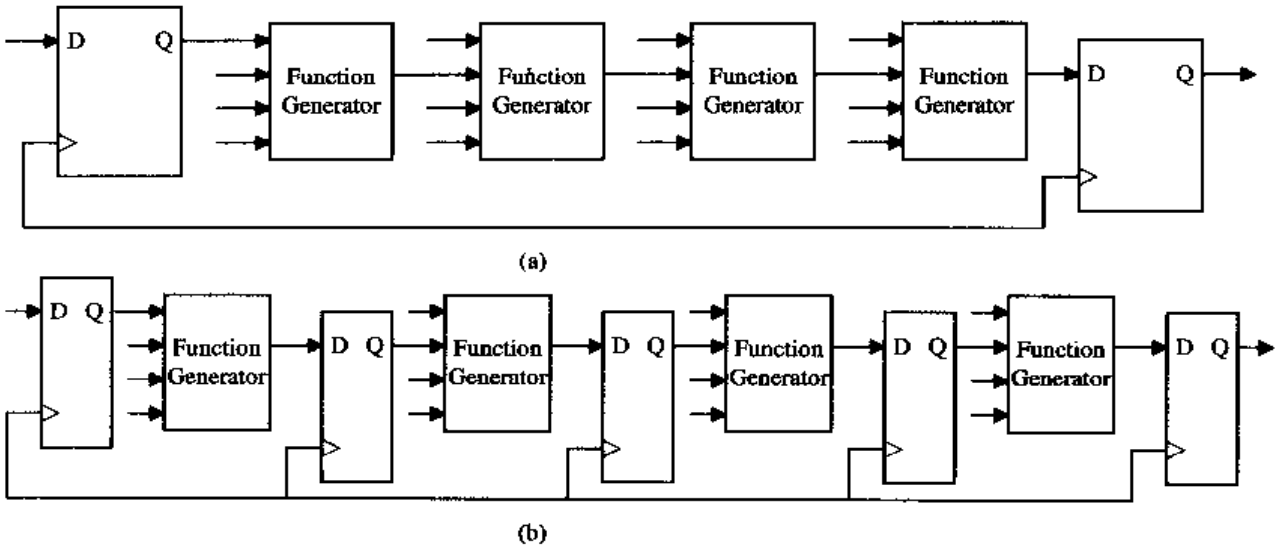


图 3.25 流水化设计

(a) 优化前；(b) 流水化设计

3.7.7 设计中不期望的锁存器

在硬件描述语言中，如果条件表达式没有写完整，往往会引入不期望的锁存器。这些锁存器从电路结构考虑完全没有必要，而且可能会带来毛刺，影响电路的性能，所以一定要避免。初学者经常会犯这种错误，要引起注意。

下面举例说明。请看下面的代码：

```
module module1 (a,b,c,d,f,s);
input a,b,c,d;
input [1:0] s;
output f; reg f;

always @(a or b or c or d or s) begin
    case (s) // independent of full or parallel attributes
    2'b00 : f = a;
    2'b01 : f = b;
    2'b10 : f = c;
    2'b11 : f = d;
```

```
        endcase
    end
endmodule
```

这段代码中，对变量 s 的所有取值的情况都进行了说明，综合出的结果如图 3.26 所示。

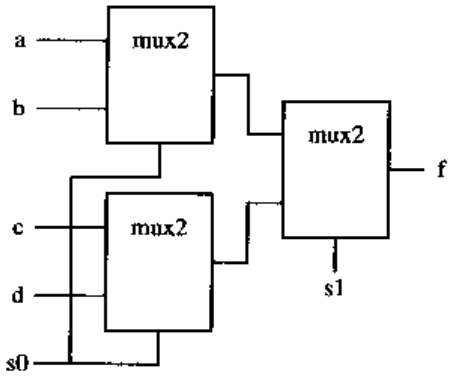


图 3.26　module1 的综合结果

如果 case(s)没有写完整，如下面的代码所示：

```
module module2 (a,b,c,f,s);
input a,b,c;
input [1:0] s;
output f;
reg f;

always @(a or b or c or s) begin
    case (s)
     2'b00 : f = a;
     2'b01 : f = b;
     2'b10 : f = c;
     endcase
end
endmodule
```

则相应的综合结果如图 3.27 所示。

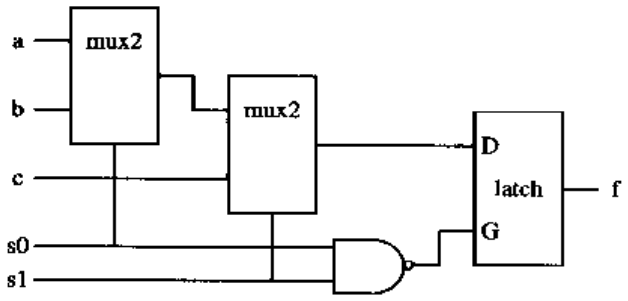


图 3.27　module2 的综合结果

如果 case(s)中使用 default 来定义缺省情况下的取值，则可以避免生成多余的锁存器。下面给出修改后的代码。

```
module module3 (a,b,c,f,s);
input a,b,c;
input [1:0] s;
output f; reg f;

always @(a or b or c or s) begin
    case (s) // case full attribute or with default as below.
```

```
            2'b00 : f = a;
            2'b01 : f = b;
            2'b10 : f = c;
            default: f = 1'bx;
        endcase
    end
    endmodule
```

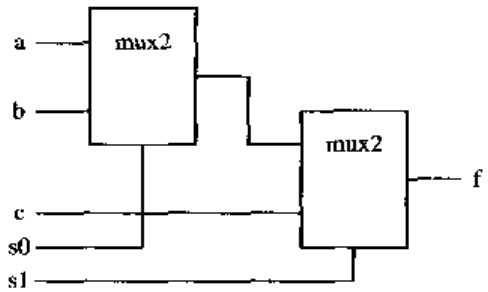


图 3.28　module3 的综合结果

相应的综合结果如图 3.28 所示。

在表 3.11 中，列出了不会引入锁存器和会引入不期望锁存器的情形。

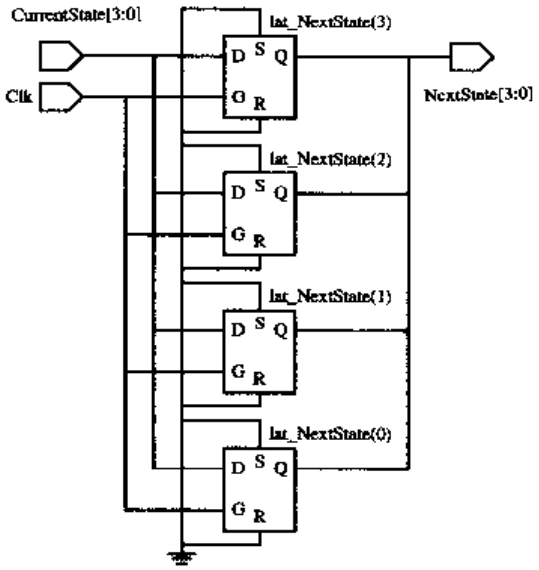
表 3.11　不会引入锁存器与会引入不期望的锁存器的代码比较

	不会引入锁存器的例子	引入不期望的锁存器的例子
情形 1	always @(signal_names) case (signal_names) 3'b000, 3'b001 : output = 4'b0000; 3'b010 : output = 4'b1010; 3'b011, 3'b100, 3'b101 : output = 4'b0101; 3'b110, 3'b111 : output = 4'b0001; endcase 说明：组合表达式要完整	always @(signal_names) case (signal_names) 3'b000, 3'b001 : output = 4'b0000; 3'b010 : output = 4'b1010; 3'b011, 3'b100, 3'b111 : output = 4'b0001; endcase

情形 2	module Compute (marks,Grade);
	input [1:4] Marks;
	output [0:1] Grade;
	reg [0:1] Grade;
	parameter FAIL=1, PASS=2,EXCELLENT =3;
	
	always @(Marks)
	if (Marks <5)
	Grade =FAIL;
	Else if ((Marks >=5)& (Marks <10))
	Grade =PASS;
	Else
	Grade=EXCELLENT;
	endmodule
		module Compute (marks,Grade);
		input [1:4] Marks;
		output [0:1] Grade;
		reg [0:1] Grade;
		parameter FAIL=1, PASS=2,EXCELLENT =3;
		
		always @(Marks)
		if (Marks <5)
		Grade =FAIL;
		Else if ((Marks >=5)& (Marks <10))
		Grade =PASS;
		endmodule
		在这个例子中，参数 EXCELLENT 没有用到

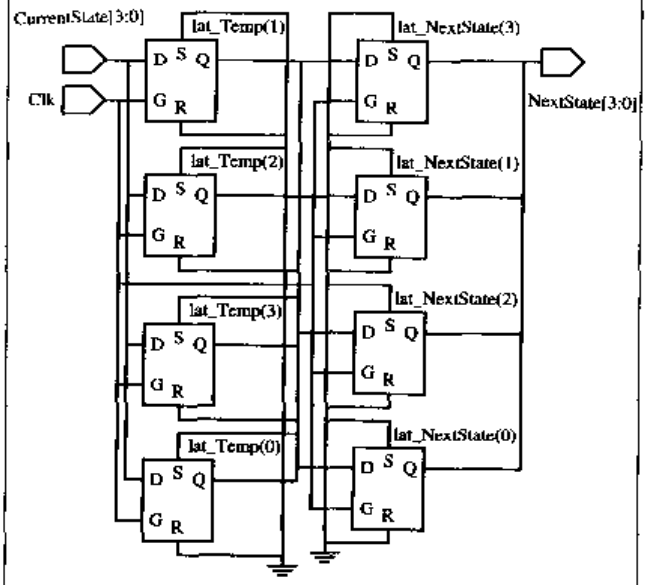
情形 3:
已知某个变量的取值是不完整的，如右边的例子，已知 Toggle 只取 01,10，如果不加 default，则会多出两个锁存器
	module NextStateLg(NextToggle,Toggle);
	input [1:0] Toggle;
	output[1:0] NextToggle;
	reg [1:0] NextToggle;
	
	always @(Toggle)
	case (Toggle)
	2'b01: NextToggle=2'b10;
	2'b10: NextToggle=2'b01;
	default: NextToggle=2'b01;
	endcase
	endmodule
		module NextStateLg(NextToggle,Toggle);
		input [1:0] Toggle;
		output[1:0] NextToggle;
		reg [1:0] NextToggle;
		
		always @(Toggle)
		case (Toggle)
		2'b01: NextToggle=2'b10;
		2'b10: NextToggle=2'b01;
		endcase
		endmodule

续表

	不会引入锁存器的例子	引入不期望的锁存器的例子
情形 4：在敏感列表中的信号未被写完整	module LatchInfer(Clk,CurrentState,NextState); input Clk; input [3:0] CurrentState; output [3:0] NextState; reg[3:0] NextState; always @(Clk or CurrentState) if(Clk) NextState=CurrentState; endmodule 综合结果如图 3.29 所示。  图 3.29　锁存器的引入	

情形 5：如果在一个不完整的条件赋值语句中，一个变量在赋值前就被使用，则会引入锁存器	module RegUsedBeforeDef (Clk,CurrentState,
	NextState);
	input Clk;
	input [3:0] CurrentState;
	output [3:0] NextState;
	
	reg [3:0] Temp;
	
	always @(Clk or CurrentState or Temp)
	　if(Clk)
	　begin
	　NextState=Temp;
	Temp=CurrentState;
	end
	endmodule
	综合结果如图 3.30 所示。
		module RegUsedBeforeDef(Clk,Current-
		State,NextState);
		input Clk;
		input [3:0] CurrentState;
		output [3:0] NextState;
		
		reg [3:0] Temp;
		
		always @(Clk or CurrentState or Temp)
		　if(Clk)
		　begin
		　Temp=CurrentState;
		NextState=Temp;
		end
		endmodule

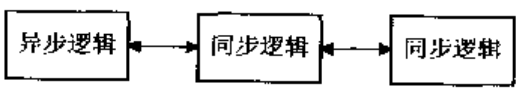
续表

	不会引入锁存器的例子	引入不期望的锁存器的例子
	 图 3.30 变量在赋值前被使用 读者可以综合右边的代码看看，会发现少了四个锁存器	

3.7.8 对可综合设计的一些建议

表 3.12 示出了对可综合设计的一些建议及补充说明。

表 3.12 可综合设计的一些建议及其补充说明

建议	补充说明
将所有模块的输出都寄存器化	时序上更容易一些(但浪费寄存器)
将特定应用的代码与通用代码分开	
将异步逻辑与同步逻辑分开，如图 3.31 所示  图 3.31 异步逻辑与同步逻辑分开	

续表

建议	补充说明
如果要使用门限时钟，内部生成的时钟，或者要使用时钟的两个沿，则应该构造一个单元的时钟生成模块	
建议将状态机分成两部分：一部分用于组合逻辑，一部分用于时序逻辑	
不要使用 casex	因为 casex 将 X 与 Z 状态都看作不关心的状态。这在综合前的仿真与综合后的仿真中可能会出现问题

状态编码最好使用参数化的表示方式，这有利于将来的改动	例如：
	parameter [1:0] // synthesis enum state_info
	RESET_STATE = 2'b00,
	TX_STATE = 2'b01,
	RX_STATE = 2'b10,
	ILLEGAL_STATE = 2'b11;

在设计的顶层，最好不要实例化门，因为这可能导致代码难以优化	
尽量避免使用 inout 类型的端口，而使用 input 与 output 端口	
在 case 声明中，最好避免使用 x 与 z	例如， case (Select) 2'b1z: Dbus =A1; 2'b11: Dbus=A2; default: Dbus= A3; endcase 或者 case (Select) 2'b1x: Dbus =A1; 2'b11: Dbus=A2; default: Dbus= A3; endcase

请找出下面的两行代码的问题：

example: wire [63:0] some_signal;

some_signal <= 1;

答案：一般工具将1看作是32位的整数。因此some_signal信号的63到32位没有给赋值。

3.8 基本设计单元的综合

综合的实现过程如图 3.32 所示。

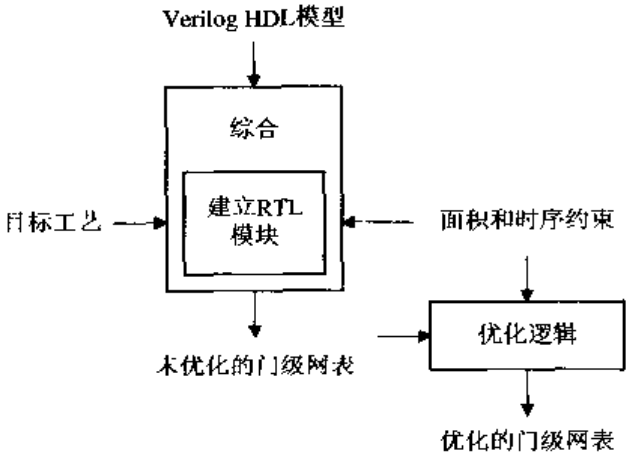
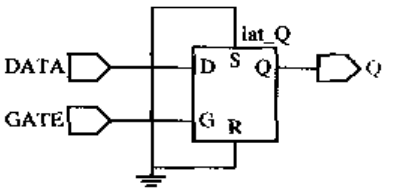


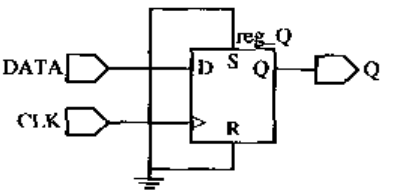
图 3.32 综合的过程

表 3.13 中，列出了一些基本设计单元的综合结果。了解这些设计代码与硬件结构的关系，是掌握“硬件式思维”的第一步。

表 3.13 基本设计单元的代码及其综合结果

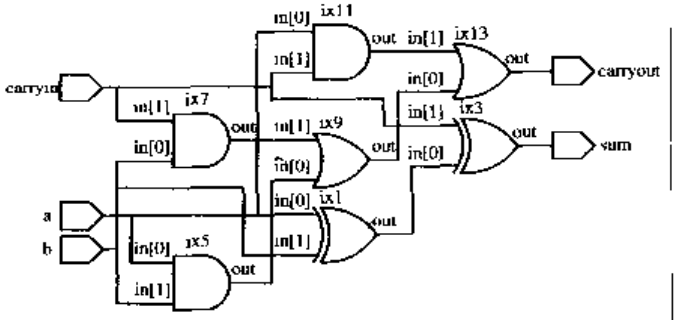
类别	代码	综合结果

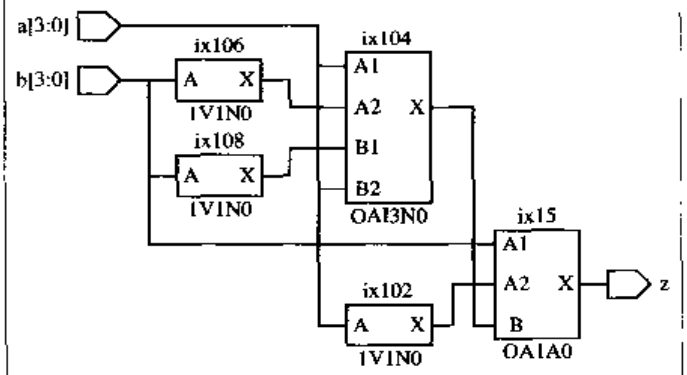
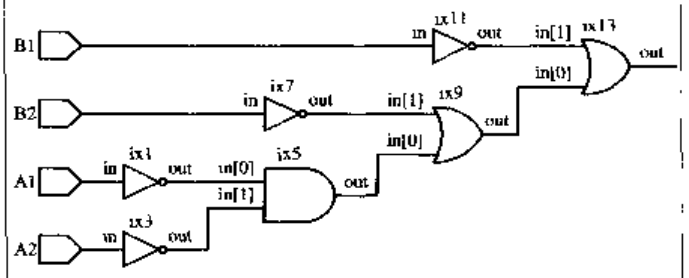
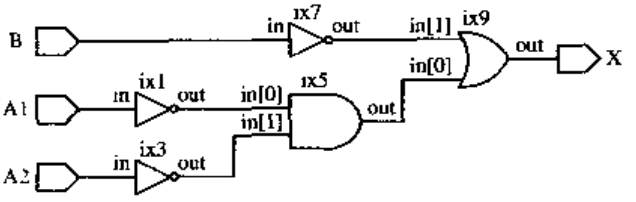
D锁存器	module d_latch (GATE, DATA, Q);
	//ports
	input GATE;
	input DATA;
	output Q;
	
	reg Q;
	
	always @ (GATE or DATA)
	begin: LATCH
	if (GATE == 1'b1)
	Q <= DATA;
	end // End Latch
	endmodule
		
		图 3.33 锁存器的综合结果

寄存器	module d_register (CLK, DATA, Q); input CLK; input DATA; output Q; reg Q; always @ (posedge CLK) begin: My_D_Reg Q <= DATA; end endmodule	 图 3.34 寄存器的综合结果

续表

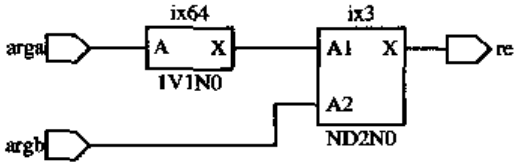
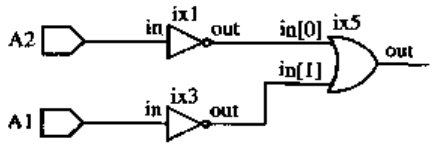
类别	代码	综合结果

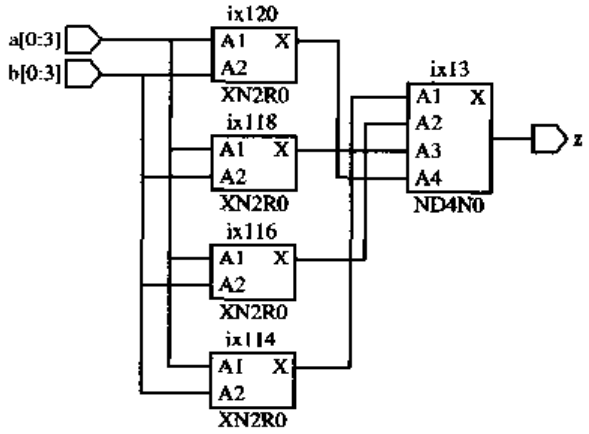
全加器	module
	fulladder(a,b,carryin,sum,carryout);
	input a,b,carryin;
	output sum,carryout;
	
	assign sum=(a^b)^carryin;
	assign
	carryout=(a&b)|(b&carryin)|(a&carryin);
	endmodule
		
		图 3.35 全加器的综合结果

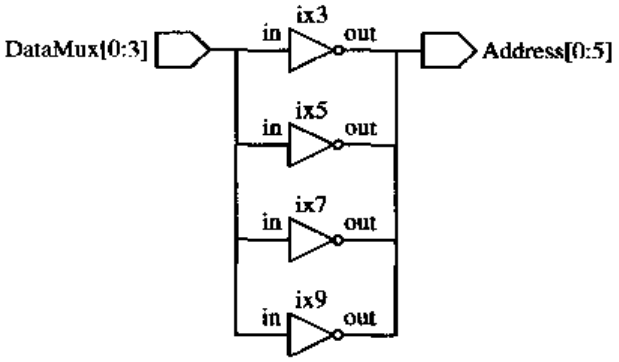
>关系符	module greaterthan(a,b,z);
	input [3:0] a,b;
	output z;
	assign z=a[1:0]>b[3:2];
	endmodule
		
		图 3.36 >关系符的综合
		其中，OAI3N0 的结构如图 3.37 所示。
		 
		图 3.37 OAI3N0 的结构
		而 OAI1A0 结构如图 3.38 所示。
		 
		图 3.38 OAI1A0 的结构

续表

类别	代码	综合结果

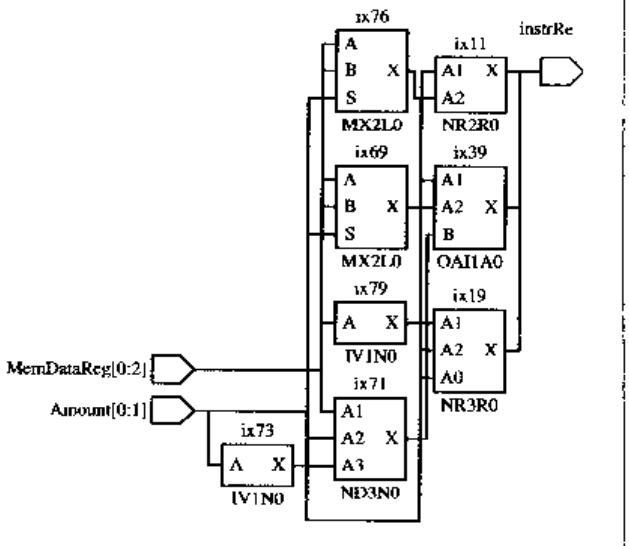
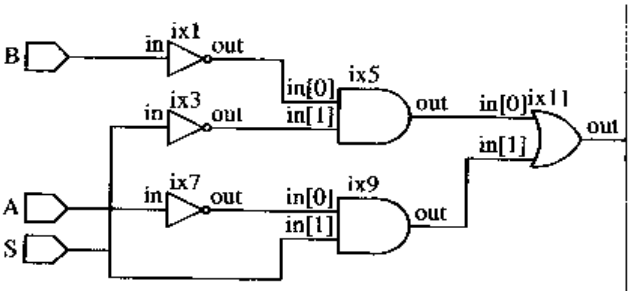
< 关系符	module lesstnanequals(arga,argb,resultz);
	input arga,argb;
	output resultz;
	reg resultz;
	integer argaint;
	integer argbint;
	always@(arga or argb)
	begin
	argaint=-arga;
	argbint=-argb;
	resultz=argaint<=argbint;
	end
	endmodule
		
		图 3.39 <关系符的综合结果
		其中，ND2N0 的结构如图 3.40 所示。
		 
		图 3.40 ND2N0 的结构

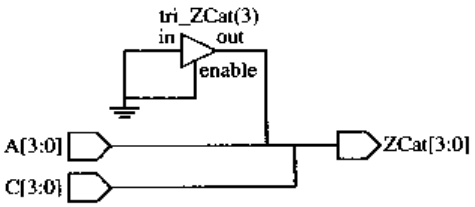
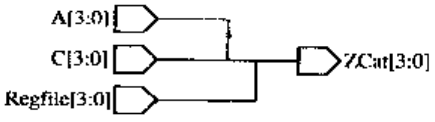
!= 关系符	module notequals(a,b,z);
	input [0:3] a,b;
	output z;
	reg z;
	always @(a or b)
	begin: DF_LABEL
	integer inta,intb;
	inta=a;
	intb=b;
	z=inta!=intb;
	end
	endmodule
		
		图 3.41 !=关系符的综合
		注：XN2R0 是或门，而 ND4N0 是与非门

常数左移 <<	module ConstantShift(DataMux,Address);
	input [0:3] DataMux;
	output [0:5] Address;
	
	assign Address=(~DataMux)<<2;
	endmodule
		
		图 3.42 常数左移<<的综合

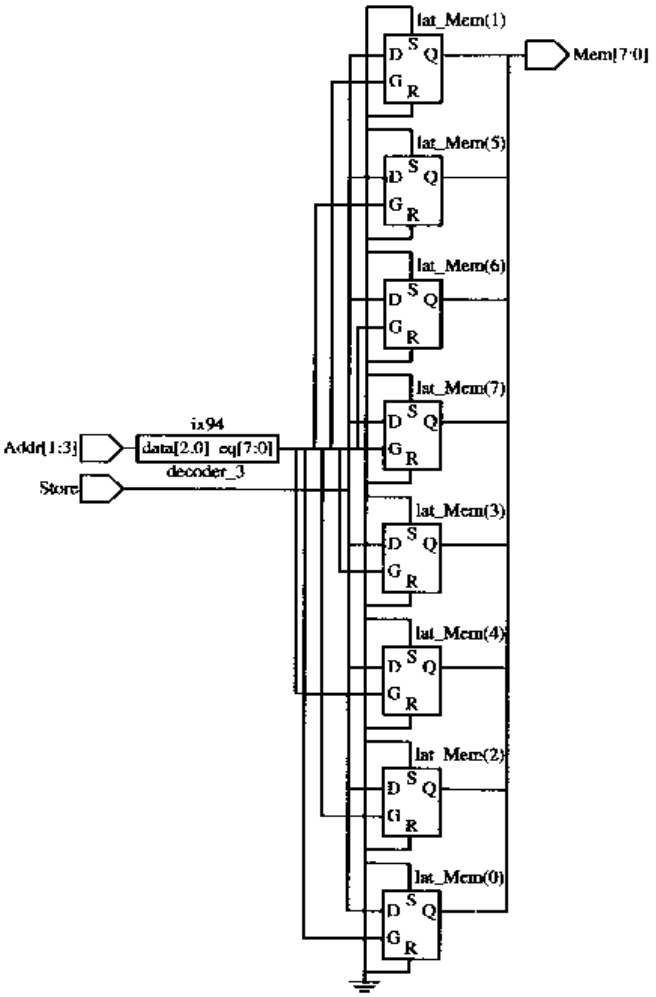
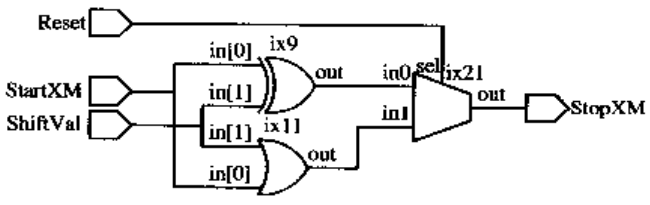
续表

类别	代码	综合结果

变量右移>>	module VariableShif(MemDataReg,Amount, InstrReg);
	input [0:2] MemDataReg;
	input [0:1] Amount;
	output[0:2] InstrReg;
	
	assign InstrReg =MemDataReg >> Amount;
	endmodule
		
		
		图 3.43 变量右移的综合
		其中，MX2L0 的结构如图 3.44 所示。
		 
		
		图 3.44 MX2L0 的结构

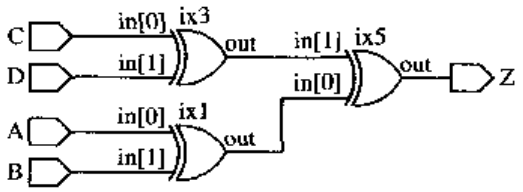
部分选择	module PartSelect(A,C,ZCat); input [3:0] A,C; output [3:0] ZCat; assign ZCat[2:0]= {A[2],C[3:2]}; endmodule	 图 3.45 部分选择的综合结果
索引值为常数	module ConstantIndex (A,C,Regfile,ZCat); input [3:0] A,C; input [3:0] Regfile; output [3:0] ZCat; assign ZCat[3:1] = {A[2],C[3:2]}; assign ZCat[0] = Regfile[3]; endmodule	 图 3.46 索引值为常数的综合结果

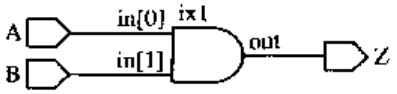
续表

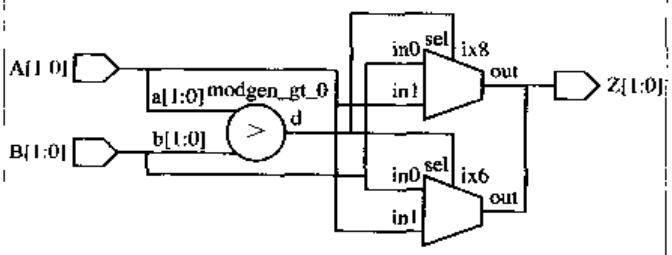
类别	代码	综合结果
索引值为变量	module VariableIndex(Mem,Store,Addr); output [7:0] Mem; input Store; input [1:3] Addr; assign Mem [Addr] = Store; endmodule	 图 3.47　索引值为变量的综合结果
条件表达式	module ConditionEx(StartXM,ShiftVal,Reset, StopXM); input StartXM,ShiftVal,Reset; output StopXM; assign StopXM= (!Reset) ?StartXM ^ShiftVal: StartXM | ShiftVal; endmodule	 图 3.48　条件表达式的结构

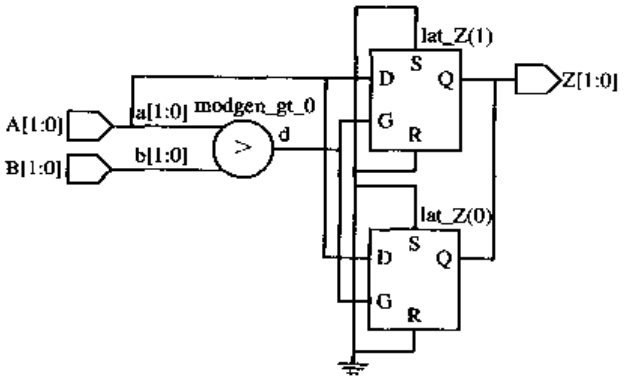
续表

类别	代码	综合结果

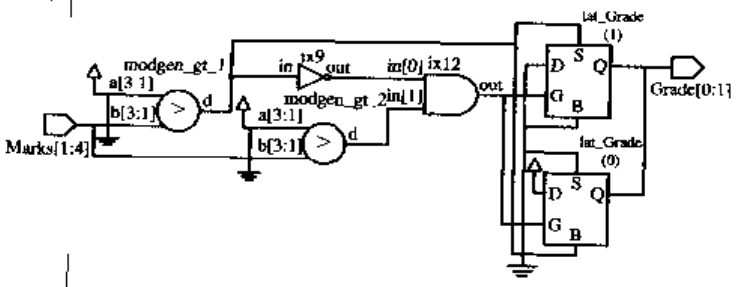
Always 过程	module EvenParity(A,B,C,D,Z);
	input A,B,C,D;
	output Z;
	reg Z, Temp1,Temp2;
	
	always @(A or B or C or D)
	begin
	Temp1= A^B;
	Temp2 =C^D;
	Z=temp1^Temp2;
	End
	说明：always 中，被说明为 reg 类型的信号并不一定会综合出寄存器来
		综合结果如图 3.49 所示。
		 
		
		图 3.49 always 的综合

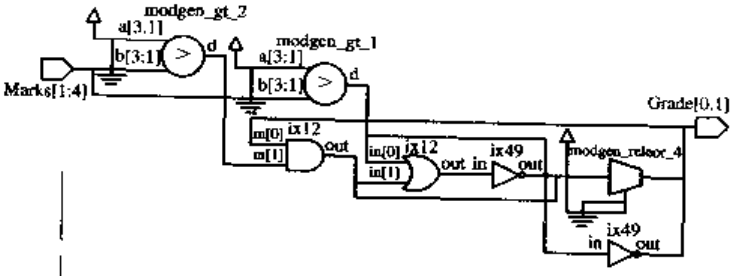
敏感列表不完整时的综合结果	module AndBehavior (Z,A,B); input A,B; output Z; reg Z; always @(B) Z=A&B; endmodule 说明：在代码中，只有当 B 变化时，Z 才变化，而综合的结果，当 A 变化时，Z 也变化，这是在仿真时出现的不匹配的情况。所以一般要在 always 语句中的敏感列表中将变量包括完整	 图 3.50 敏感列表不完整时的综合结果

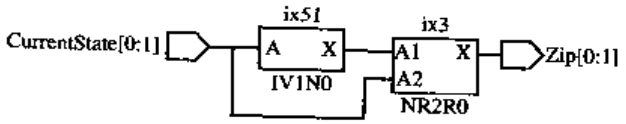
完整的 if 语句	module SelectOne(A,B,Z);
	input [1:0] A,B;
	output [1:0] Z;
	reg [1:0] Z;
	always @(A or B)
	if(A>B)
	Z=A;
	else Z=B;
	endmodule
		
		
		图 3.51 if 语句

不完整的 if 语句	module SelectOne(A,B,Z);
	input [1:0] A,B;
	output [1:0] Z;
	reg [1:0] Z;
	always @(A or B)
	if(A>B)
	Z=A;
	endmodule
	说明：注意与上一个代码的差异。可以看出，该代码综合出了 D 锁存器
		
		
		图 3.52 if 不完整的综合结果

续表

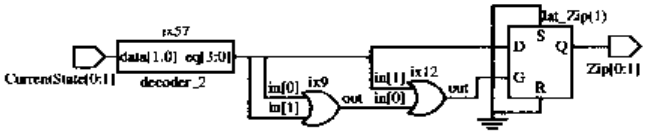
类别	代码	综合结果
代码中有未用到的参量	module Compute (Marks,Grade); input [1:4] Marks; output [0:1] Grade; reg [0:1] Grade; parameter FAIL=1, PASS=2,EXCELLENT =3; always @(Marks) if (Marks <5) Grade =FAIL; else if ((Marks >=5)& (Marks <10)) Grade =PASS; endmodule	 图 3.53 有未用参量的综合结果

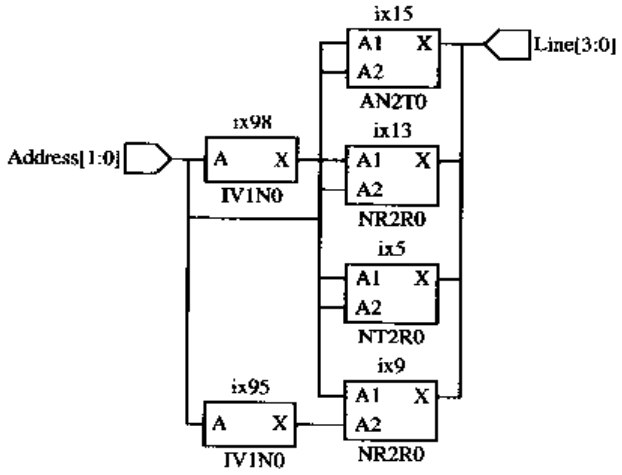
修改后的代码	module Compute (Marks,Grade);
	input [1:4] Marks;
	output [0:1] Grade;
	reg [0:1] Grade;
	parameter FAIL=1,
	PASS=2,EXCELLENT =3;
	
	always @(Marks)
	if (Marks <5)
	Grade =FAIL,
	else if ((Marks >=5)& (Marks <10))
	Grade =PASS;
	else
	Grade=EXCELLENT;
	endmodule
		
		
		图 3.54 修改后的综合结果

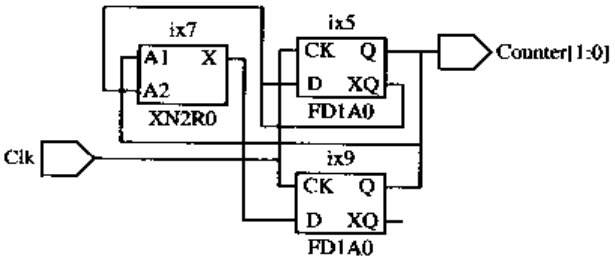
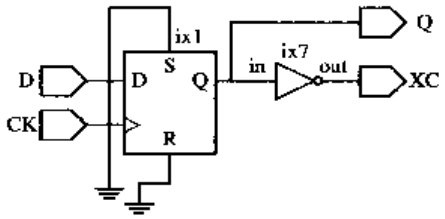
完整的 case 语句	module StateUpdata(CurrentState, Zip);
	input [0:1] CurrentState;
	output [0:1] Zip;
	reg [0:1] Zip;
	
	parameter S0=0,S1=1,S2=2,S3=3;
	
	always @(CurrentState)
	case (CurrentState)
	S0,
	S2,
	S3: Zip=0;
	S1: Zip=3;
	endcase
	endmodule
		
		
		图 3.55 完整的 Case 语句的综合结果

续表

类别	代码	综合结果

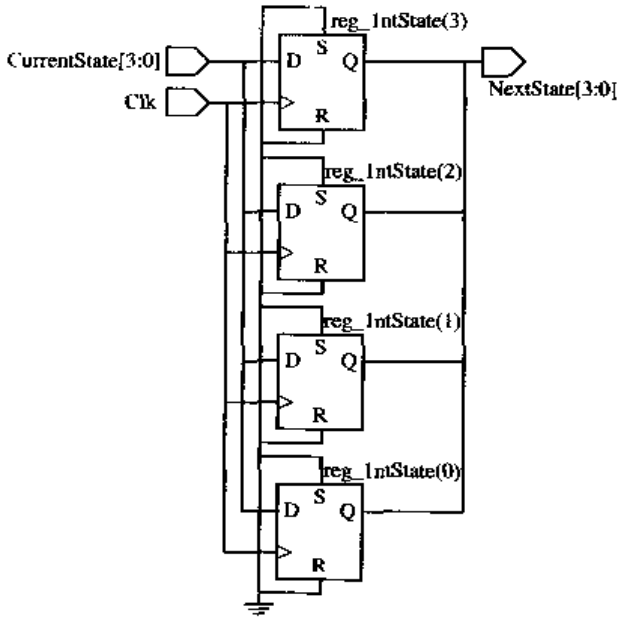
不完整的 case 语句	module StateUpdata(CurrentState, Zip);
	input [0:1] CurrentState;
	output [0:1] Zip;
	reg [0:1] Zip;
	
	parameter S0=0,S1=1,S2=2,S3=3;
	
	always @(CurrentState)
	case (CurrentState)
	S0,
	S3: Zip=0;
	S1: Zip=3;
	endcase
	endmodule
		
		图 3.56 不完整的 Case 语句

For 循环	module Demultiplexer(Address, Line);
	input [1:0] Address;
	
	output [3:0] Line;
	reg [3:0] Line;
	
	integer J;
	
	always @(Address)
	for(J=3;J>=0;J=J-1)
	if(Address ==J)
	Line[J]=1;
	else
	Line[J]=0;
	endmodule
	综合时，自动把循环拆环
		
		图 3.57 For 循环

计数器	module Counter1(Clk,Counter);
	parameter COUNTER_SIZE =2;
	input Clk;
	output [COUNTER_SIZE-1:0] Counter;
	reg [COUNTER_SIZE-1:0] Counter;
	always @(posedge Clk)
	Counter<=Counter+1;
	endmodule
		
		图 3.58 计数器
		其中，FD1A0 的结构如图 3.59 所示。
		 
		图 3.59 FD1A0 的结构

续表

类别	代码	综合结果

触发器（有中间变量的形式）	module FlipFlop1(Clk,CurrentState, NextState);
	input Clk;
	input [3:0] CurrentState;
	output [3:0] NextState;
	
	integer IntState;
	
	always @(posedge Clk)
	IntState<=CurrentState;
	
	assign NextState=IntState;
	endmodule
		

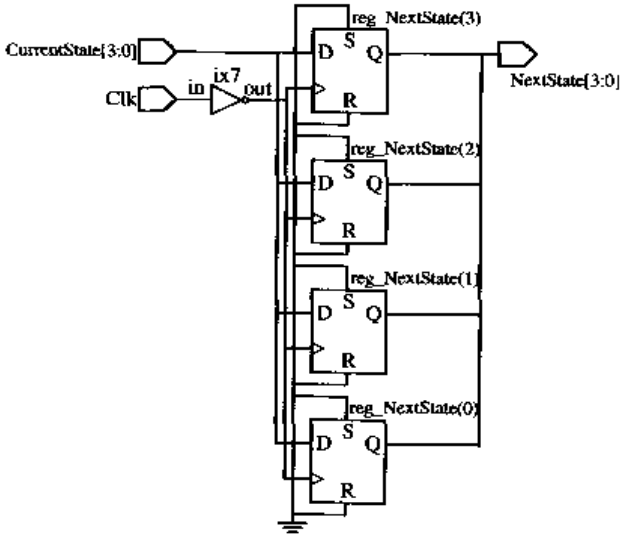
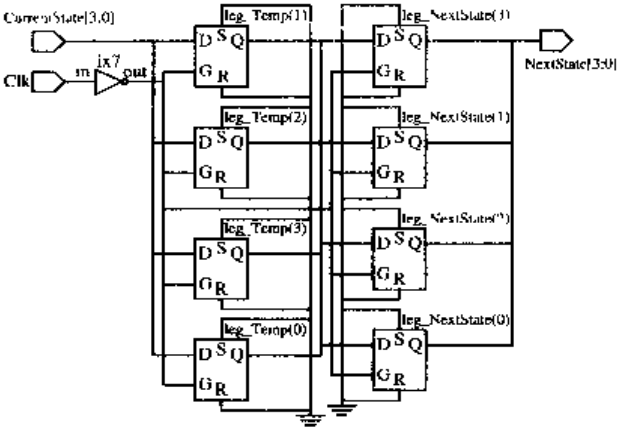
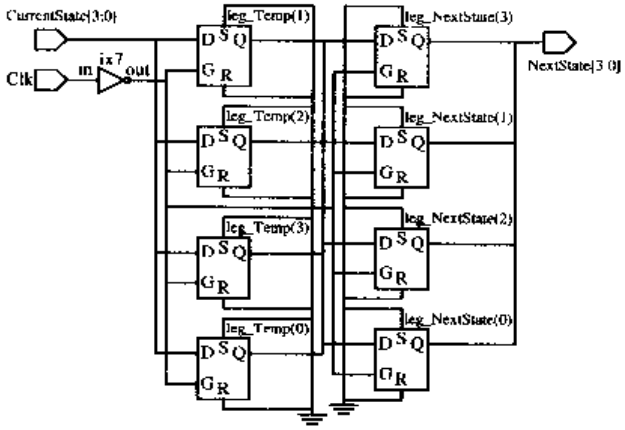
不会综合出触发器的全局变量	module GlobalReg(Clk,CurrentState,NextState); input Clk; input [3:0] CurrentState; output [3:0] NextState; reg [3:0] NextState; reg [3:0] Temp; always @(negedge Clk) begin Temp=CurrentState; NextState<=Temp; end endmodule	

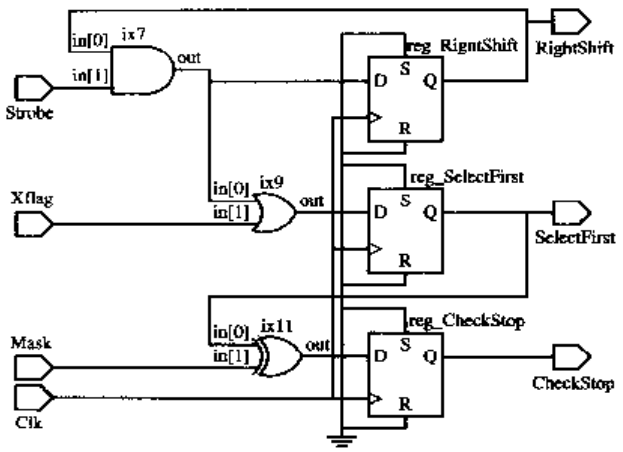
图 3.60 触发器的综合

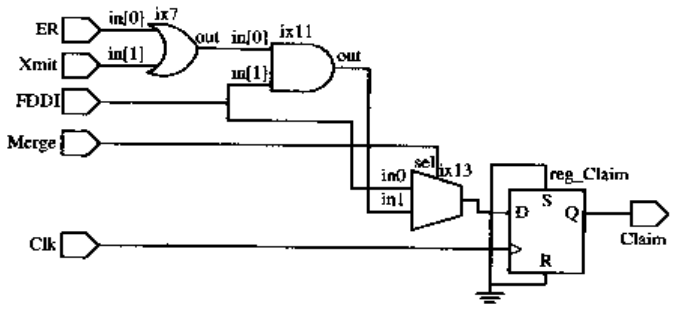
图 3.61 修改后的代码

续表

类别	代码	综合结果
综合出触发器的全局变量	module GlobalReg(Clk,CurrentState,NextState); input Clk; input [3:0] CurrentState; output [3:0] NextState; reg [3:0] NextState; reg [3:0] Temp; always @(negedge Clk) begin Temp<=CurrentState; NextState<=Temp; end endmodule 注意与上一个代码的差异	 图 3.62 由全局变量综合出触发器
Temp 为局部变量	module GlobalReg(Clk,CurrentState,NextState); input Clk; input [3:0] CurrentState; output [3:0] NextState; reg [3:0] NextState; always @(negedge Clk) begin: Label1 reg [3:0] Temp; NextState<=Temp; Temp=CurrentState; end endmodule	 图 3.63 修改后的结果
改变赋值次序	module GlobalReg(Clk,CurrentState,NextState); input Clk; input [3:0] CurrentState; output [3:0] NextState; reg [3:0] NextState; always @(negedge Clk) begin: Label1 reg [3:0] Temp; Temp=CurrentState; NextState<=Temp; end endmodule 说明：这里的 Temp=CurrentState 必须使用阻塞赋值	改变赋值次序的综合结果如图 3.61 所示

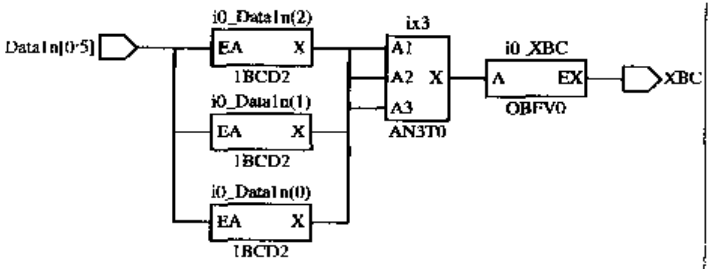
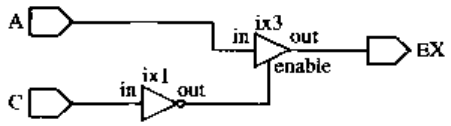
续表

类别	代码	综合结果
阻塞赋值与非阻塞赋值	module BlockAndNon (Clk,Strobe,Xflag,Mask, RightShift,SelectFirst,CheckStop); input Clk,Strobe,Xflag,Mask; output RightShift,SelectFirst,CheckStop; reg RightShift,SelectFirst,CheckStop; always @(posedge Clk) begin RightShift=RightShift&Strobe; SelectFirst<=RightShift |Xflag; CheckStop<=SelectFirst^Mask; end endmodule	 图 3.64　阻塞赋值与非阻塞赋值

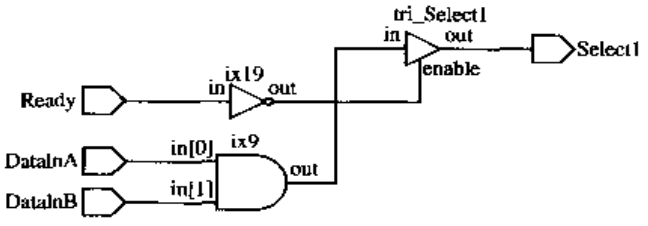
阻塞赋值也能综合出寄存器	module BlockingExample (Clk,Merge,
	Er,Xmit,FDDI,Claim);
	input Clk,Merge,ER,Xmit,FDDI;
	output Claim;
	reg Claim;
	
	reg FCR;
	
	always @(posedge Clk)
	begin
	FCR= ER|Xmit;
	if(Merge)
	Claim=FCR & FDDI;
	else
	Claim=FDDI;
	end
	endmodule
		
		图 3.65　由阻塞赋值综合出的寄存器

续表

类别	代码	综合结果

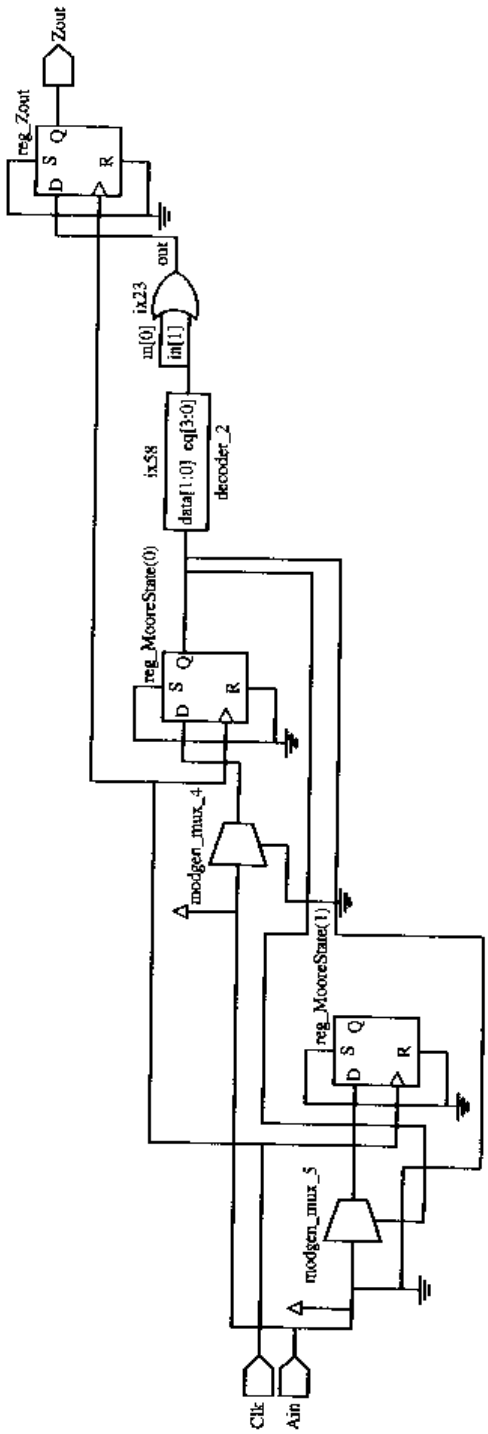
函数调用(通常函数调用总用于组合逻辑的生成)	module
	FunctionExample(XBC,DataIn);
	output XBC;
	input [0:5] DataIn;
	
	function [0:2] CountOnes;
	input [0:5] A;
	integer K;
	begin
	CountOnes=0;
	for(K=0;K<=2;K=K+1)
	if(A[K])
	CountOnes = CountOnes+1;
	end
	endfunction
	
	assign XBC=CountOnes(DataIn)>2;
	endmodule
	说明：在综合时，函数调用首先被解析为如下形式：
	CountOnes=0;
	if(DataIn[0])
	CountOnes=CountOnes+1;
	if(DataIn[1])
	CountOnes=CountOnes+1;
		
		图 3.66　函数调用
		其中，OBFV0 的结构如图 3.67 所示。
		 
		图 3.67　OBFV0 的结构

任务 tasks 的综合	tasks 可以是组合逻辑，也可以是时序逻辑。如果 task 调用发生在一个具有时钟事件的时钟化的 always 声明中，则 task 的输出可能被综合为触发器。 代码及图略	

三态门	module
	ThreeState(Ready,DataInA,DataInB,Select1);
	input Ready,DataInA.DataInB;
	output Select1;
	reg Select1;
	
	always @(Ready or DataInA or DataInB)
	if(Ready)
	Select1= 1'bz;
	else
	Select1= DataInA& DataInB;
	Endmodule
		
		图 3.68　三态门

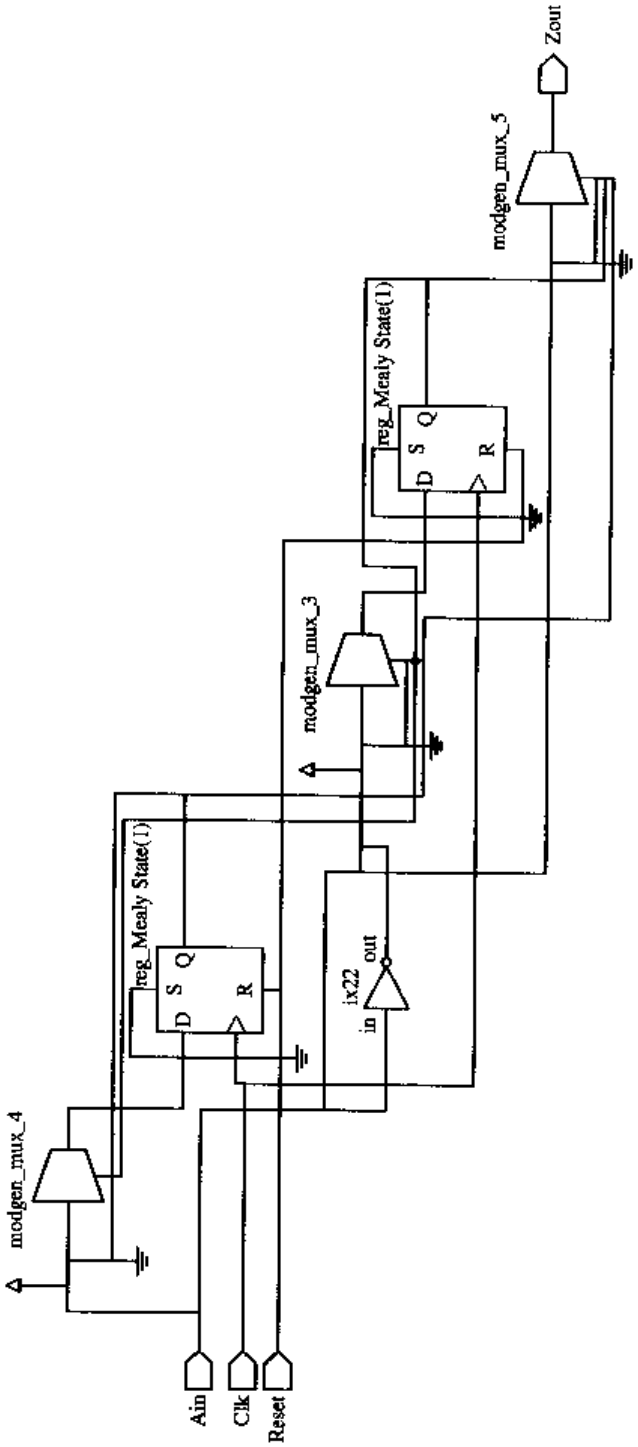
续表

类别	代码	综合结果

moore 状态机	moore 状态机，电路的输出只与状态机的状态有关，与输入无关。因而用 Case 语句来描述很方便，如下：
	module Moorefsm(Ain,Clk,Zout);
	input Ain,Clk;
	output Zout;
	reg Zout;
	
	parameter S0=0,S1=1,S2=2,S3=3;
	reg[0:1] MooreState;
	
	always @(posedge Clk)
	case(MooreState)
	S0:
	begin
	Zout<=1;
	MooreState<=(!Ain)?S0:S2;
	end
	S1:
	begin
	Zout<=0;
	MooreState<=(!Ain)?S0: S2;
	end
	S2:
	begin
	Zout<=0;
	MooreState<=(!Ain)?S2:S3;
	end
	S3:
	begin
	Zout<=1;
	MooreState<=(!Ain)?S1:S3;
	end
	endcase
	endmodule
		
		
		图 3.69 Moore 状态机

续表

类别	代码	综合结果

mealy 状态机	module
	Mealyfsm(Ain,Clk,Reset,Zout);
	input Ain,Clk,Reset;
	output Zout;
	reg Zout;
	
	parameter S0=0,S1=1,S2=2,S3=3;
	reg[0:1] MealyState,NextState;
	
	//时序逻辑部分
	always @(posedge Clk or posedge Reset)
	begin
	if(Reset)
	MealyState<=S0;
	else
	MealyState<=NextState;
	end
	
	//组合逻辑部分
	always @(MealyState or Ain)
	case(MealyState)
	
	S0:
	begin
	Zout=(!Ain)?0:1;
	NextState=(!Ain)?S0:S2;
	end
	S1:
	begin
	Zout=(!Ain)?0:1;
	NextState=(!Ain)?S0: S1;
	end
	S2:
	begin
	Zout=0;
	NextState=(!Ain)?S2:S1;
	end
	default:
	begin
	Zout=0;
	NextState=S3;
	end
	endcase
	endmodule
	注意比较这两种状态机的差别
		
		图 3.70 Mealy 状态机

3.9 静态时序分析

静态时序分析适用于较大规模系统的时序检查。与基于仿真的方法相比，它的速度要快许多倍。实际上，在 Synopsys 的 DC 中内置了静态时序分析功能，但功能更强大的静态时序分析软件是 PrimeTime。由于静态时序分析与综合的关系较密切，所以在这里对它的使用进行一下介绍。

例如，对设计 example.v 进行综合，生成的网表文件为 example.vs。

进行静态时序分析时，首先运行 PrimeTime 软件。接下来设置环境，并读入文件。与库文件进行连接。

```
pt_shell> set search_path " . /export/home/tools/npu/library/synopsys "
pt_shell> set link_path " *    my.db"
pt_shell> read_verilog example.vs (synthesized Verilog )
pt_shell> link
```

然后要设置时序环境，包括设置时钟、时钟上升/下降时间、输出负载、输入延时、输出延时。

设置时间：

```
pt_shell> create_clock -period 100 -name ck1 -waveform {0 50} [get_port CLKA]
```

设置输入延时：

```
pt_shell> set_input_delay –clock ck1 1.5 [all_inputs]
```

设置输出延时：

```
pt_shell> set_output_delay –clock ck1 2.0 [all_outputs]
```

设置上升时间：

```
pt_shell> set_input_transition –rise 0.8 WEA
```

设置下降时间：

```
pt_shell> set_input_transition –fall   0.9 WEA
```

设置负载：

```
pt_shell> set_load –pin_load 1.2 [all_outputs]
```

最后，给出时序分析结果：

```
pt_shell> report_timing –max_paths 100
pt_shell> report_constraint –all_violators
```

3.10 练　　习

用 Verilog 实现一个无符号 32 位原码乘法器，写出综合脚本，用 DC 进行综合，得到最小面积的实现、最优性能的实现，打印综合出的电路结构，并分析综合报告。

第4章 基本模块的设计

IC 设计的一般流程是“自顶向下”，而学习的流程则应该是“自下而上”。从基本模块的设计入手，能更快、更全面、更直接地掌握设计。这里给出的基本模块都是 IC 设计中较常用到的。希望读者也能够建立自己的“基本模块”库。

4.1 差错控制编码

在通信过程中，经常会产生错误。产生差错的原因包括：

- 信道的电气特性引起信号幅度、频率、相位的畸变；
- 信号反射；
- 串扰；
- 闪电、大功率电机的开关等。

线路传输差错是不可避免的，但要尽量减小其影响。通信双方可采取的对策是：接收方进行差错检测，并向发送方应答，告知是否正确接收。差错检测主要有两种方法。

1) 奇偶校验(Parity Checking)

在原始数据字节的最高位增加一个附加比特位，使结果中 1 的个数为奇数(奇校验)或偶数(偶校验)。增加的位称为奇偶校验位。

例如，原始数据=1100010，采用偶校验，则增加校验位后的数据为 11100010。若接收方收到的字节奇偶结果不正确，就可以知道传输中发生了错误。

奇偶校验只能检测出奇数个比特位错，对偶数个比特位错则无能为力。

2) 循环冗余校验(CRC)

CRC 是一种通过多项式除法检测错误的方法。该技术将数据串看成系数为 0 或 1 的多项式。收发双方约定一个生成多项式 G(x)(其最高阶和最低阶系数必须为 1)。发送方在数据串的末尾加上校验和，使带校验和的数据串的多项式能被 G(x)整除。接收方收到后，用 G(x)除多项式，若有余数，则传输有错。

4.1.1 奇偶校验模块

实现奇偶校验的方式有水平冗余校验、垂直冗余校验、水平垂直冗余校验等多种，但基本原理相同。

1. 功能

发送时，该模块在发送数据后加上奇校验位。接收时，进行奇校验。奇偶校验模块输

入输出端口说明如表 4.1 所示，奇偶校验模块框图如图 4.1 所示。

表 4.1　奇偶校验模块输入输出端口说明

输入端口	端口说明
data_in[7:0]	输入数据
pa_in	输入的校验位
even_odd	输入的控制信号，控制是进行奇校验还是偶校验
输出端口	端口说明
pa_out	输出相应的奇偶校验位
error	奇偶校验的结果是否正确

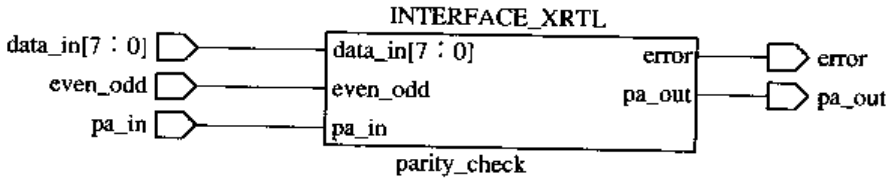


图 4.1　奇偶校验模块框图

2. **思路**

在奇校验中，要保证生成的数据中，1 的个数是奇数个。在偶校验中，要保证生成的数据中，1 的个数为偶数个。奇偶校验位由异或操作得到。

对于奇校验，核心代码为：

```
Pa_out <= data_in[7] ^ data_in[6] ^ data_in[5] ^ data_in[4] ^
          data_in[3] ^ data_in[2] ^ data_in[1] ^ data_in[0] ^ 1'b1;
```

对于偶校验，核心代码为：

```
Pa_out <= data_in[7] ^ data_in[6] ^ data_in[5] ^ data_in[4] ^
          data_in[3] ^ data_in[2] ^ data_in[1] ^ data_in[0] ;
```

下面给出一个简单的实现。

3. **实现代码**

下面是奇偶校验模块的一种实现代码。

```
/*************************************************************/
// MODULE: parity_check
//该模块能生成奇偶校验码，并进行奇偶校验
/*************************************************************/
//定义时钟到输出的延时
`define DELAY   1
//模块定义
module parity_check(   data_in,    pa_in,      even_odd,  pa_out,error);
//端口声明
```

```
input [7:0]  data_in;        //输入数据
input        pa_in;          //输入的校验位
input        even_odd;       //控制信号，选择是奇还是偶校验
output       pa_out;         //奇偶校验位
output       error;          //校验结果

wire [7:0]   data_in;
wire         pa_in;
wire         even_odd;
wire         pa_out;
wire         error;

//赋值声明
assign #`DELAY pa_out = even_odd ? ^data_in[7:0] : ~^data_in[7:0];
assign #`DELAY error = even_odd ? ^{data_in[7:0], pa_in} : ~^{data_in[7:0], pa_in};

endmodule            // parity_check
```

该模块的综合结果如图 4.2 所示。

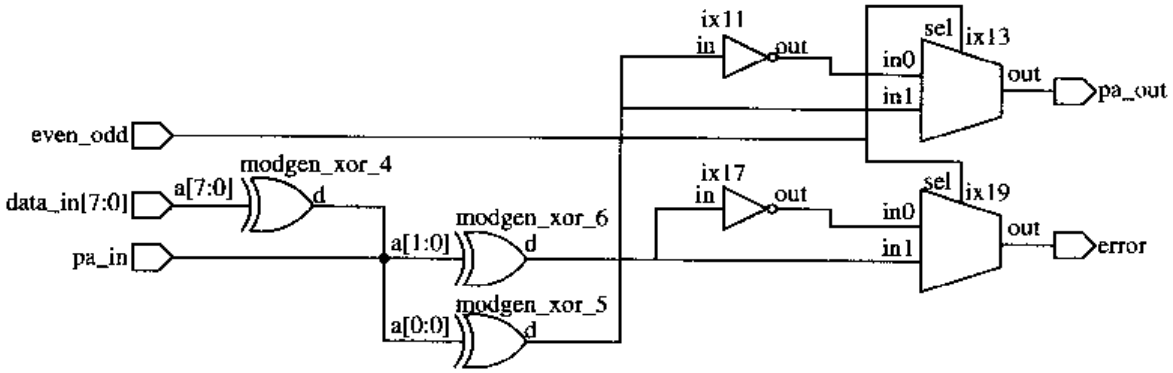


图 4.2　奇偶校验模块的综合结果

4. 扩展

读者可以对这个模块进行扩展：

- 增加时钟和复位信号，使它成为时序电路。
- 增加串并转换模块，可以接收整字节数据，发出带奇偶校验位的串行数据；可以接收串行数据，并进行奇偶校验。

4.1.2　汉明码编解码器

奇偶校验码只有一位监督位，只能指明有、无差错二种情况，故只能检错。为了纠错，必须增加监督位。汉明码是一种能纠正单比特差错的线性分组码。为了纠错，不仅要指明有奇偶差错，还要能确定差错的位置。

设线性分组码(n，k)，有 k 个信息位，r=n-k 个监督位．为了能确定 n 种位置上的差错及线性无差错，要求 r 的值为 $2^r \geqslant n+1=k+r+1$，即

$$2^r \quad r \geqslant k+1$$

例如，k=7(7 位 ASCII 字符)，则 r= 4，汉明码长 n= 7+4= 11。

在汉明码字中，比特位从最左边位(位号为 1)开始依次编号。位号为 2 的幂的位(1，2，4，8 等)是 r 个监督位(也称汉明比特)，其余位(3，5，6，7，9 等)是 k 个信息位。例如，7 个信息位为 $D_1D_2D_3D_4D_5D_6D_7$，则汉明码如表 4.2 所示。

表 4.2　汉明码码位排列

码位号	1	2	3	4	5	6	7	8	9	10	11
码位	P_1	P_2	D_1	P_3	D_2	D_3	D_4	P_4	D_5	D_6	D_7

其中 $P_i(1 \leqslant i \leqslant 4)$为监督位，其取值可由四个偶(或奇)校验关系式来确定。用 S_1、S_2、S_3 和 S_4 表示在这四个校验关系中的校验子，则 $S_4S_3S_2S_1$ 的值和错码位置的对应关系如表 4.3 所示。

表 4.3　校验子错码位置的对应关系

错码位号	汉明码位	$S_4S_3S_2S_1$
1	P_1	0001
2	P_2	0010
3	D_1	0011
4	P_3	0100
5	D_2	0101
6	D_3	0110
7	D_4	0111
8	P_4	1000
9	D_5	1001
10	D_6	1010
11	D_7	1011
无错码		0000

按表 4.3 中的规定可见，与校验子 S_1=1 相对应的码位号均为奇数，对照相应的汉明码位，可得校验关系式：

$$S_1 = P_1 \oplus D_1 \oplus D_2 \oplus D_4 \oplus D_5 \oplus D_7$$

同理，可得其它校验关系式：

$$S_2 = P_2 \oplus D_1 \oplus D_3 \oplus D_4 \oplus D_5 \oplus D_7$$

$$S_3 = P_3 \oplus D_2 \oplus D_3 \oplus D_4$$

$$S_4 = P_4 \oplus D_5 \oplus D_6 \oplus D_7$$

在发端编码时，根据信息码和校验关系式确定监督位。按偶校验关系，监督位应使上面四个关系式中的校验子的值为零(表示无差错)，故得各监督位为：

$$P_1 = D_1 \oplus D_2 \oplus D_4 \oplus D_5 \oplus D_7$$

$$P_2 = D_1 \oplus D_3 \oplus D_4 \oplus D_6 \oplus D_7$$

$$P_3 = D_2 \oplus D_3 \oplus D_4$$

$$P_4 = D_5 \oplus D_6 \oplus D_7$$

接收端收到每个码组后，先按校验关系式计算出 S_1、S_2、S_3 和 S_4，再按表 4.3 判断错码情况。

基于上述汉明码编解码原理，可以得到一种更简便的编解码方法：编码时，按表 4.2 排列信息码位和监督码位，如果信息码位为 1，则将其码位号表示为二进制码，再按模 2 加对所有的二进制码位号求和，所得结果就是监督位。解码时，将收到的汉明码位中为 1 的位的码位号表示为二进制码，再按模 2 加求和。若和为 0，则表示无误码。若和不为 0，其值则指明误码的位号。

例如，某一字符的 ASCII 码为 1100011。编码时，按表 4.2 的格式排成汉明码，如表 4.4 所示。将码位中为 1 的位的码位号表示为二进制码，再按模 2 加求和，和为 0111，这就是监督位 P4P3P2P1，如图 4.3(a)所示。将所得监督位的值填入表 4.4，得汉明码 11111000011。

表 4.4 字符 C 的汉明码码位排列

码位号	1	2	3	4	5	6	7	8	9	10	11
码位	P_1	P_2	1	P_3	1	0	0	P_4	0	1	1

解码时，将收到的汉明码位中为 1 的各位的码位号表示为二进制码，再按模 2 加求和。若无误码，则和为 0，如图 4.3(b)所示。若收到的汉明码为 11110000011，则和为 0101，如图 4.3(c)所示。这指明错码位号是 5。将第 5 位码由 0 改为 1，完成纠错。

“1”码位号	二进制号
3	0011
5	0101
10	1010
11	1011
和	0111

(a)

“1”码位号	二进制号
1	0001
2	0010
3	0011
4	0010
5	0101
10	1010
11	1011
和	0000

(b)

“1”码位号	二进制号
1	0001
2	0010
3	0011
4	0100
10	1010
11	1011
和	0101

(c)

图 4.3 汉明码的编码与解码

(a) 编码；(b) 解码(无错)；(c) 解码(有错)

1. 编码器

1) 功能

要实现一个汉明码编码器，可接收的输入数据为 8 位，产生的码字为 4 位。产生的码字可用于检查和纠正单个比特的数据错误。该模块的端口描述如表 4.5 所示。

表 4.5 汉明码编码器端口说明

端口	宽度	方向	说明
data_in	8	输入	输入数据
ham_out	4	输出	检错码输出

检错码的产生原理如下所示：

```
ham_out[3] = data_in[7] ^ data_in[6] ^ data_in[4] ^ data_in[3] ^ data_in[1];
ham_out[2] = data_in[7] ^ data_in[5] ^ data_in[4] ^ data_in[2] ^ data_in[1];
ham_out[1] = data_in[6] ^ data_in[5] ^ data_in[4] ^ data_in[0];
ham_out[0] = data_in[3] ^ data_in[2] ^ data_in[1] ^ data_in[0];
```

2) 代码

下面是编码器的一个实现代码。

```
/*************************************************************/
// MODULE:hamgen
// 该模块实现了针对 8 位数据的汉明码编码器
/*************************************************************/
// DEFINES
`define DELAY  1            // 时钟到输出的延迟
//模块描述
module hamgen(data_in,ham_out);
// 端口信号
input [7:0]     data_in;    // 输入数据
output [3:0]    ham_out;    // 检错码输出
//信号声明
wire [7:0]      data_in;
wire [3:0]      ham_out;
// 赋值
assign #`DELAY ham_out[3] = data_in[7] ^ data_in[6] ^ data_in[4] ^ data_in[3] ^ data_in[1];
assign #`DELAY ham_out[2] = data_in[7] ^ data_in[5] ^ data_in[4] ^ data_in[2] ^ data_in[1];
assign #`DELAY ham_out[1] = data_in[6] ^ data_in[5] ^ data_in[4] ^ data_in[0];
assign #`DELAY ham_out[0] = data_in[3] ^ data_in[2] ^ data_in[1] ^ data_in[0];
endmodule
```

编码器的综合结构如图 4.4 所示。

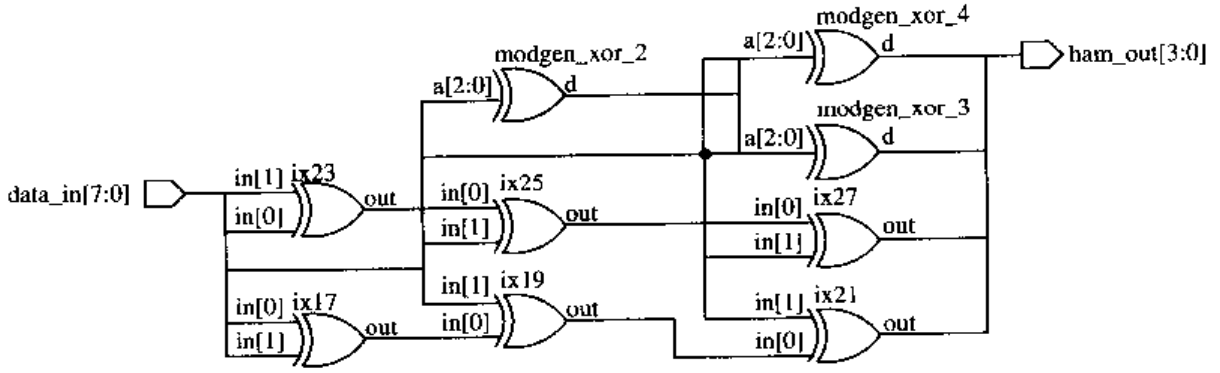


图 4.4　汉明码编码器综合结构

2. **解码器**

1) 功能

汉明码解码模块的框图如图 4.5 所示。该模块的端口说明如表 4.6 所示。

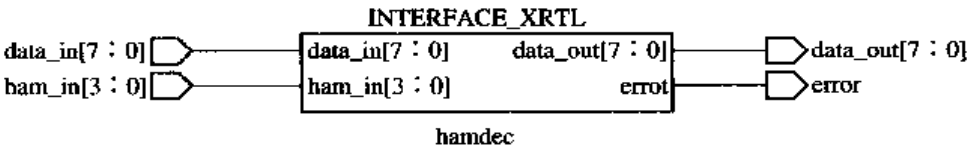


图 4.5 · 汉明码解码模块

表 4.6　汉明码解码模块端口说明

端口	宽度	方向	说明
data_in	8	输入	输入数据
ham_in	4	输入	纠错码
data_out	8	输出	输出数据
Error	1	输出	是否发生错误

2) 代码

该模块的实现思想是：由输入数据重新产生纠错码，并与原纠错码异或，如果相同，则得到 0，不同则为 1。由此判断数据有无错误，并输出标志信号。

根据得到的结果，可以纠正单个比特的错误。

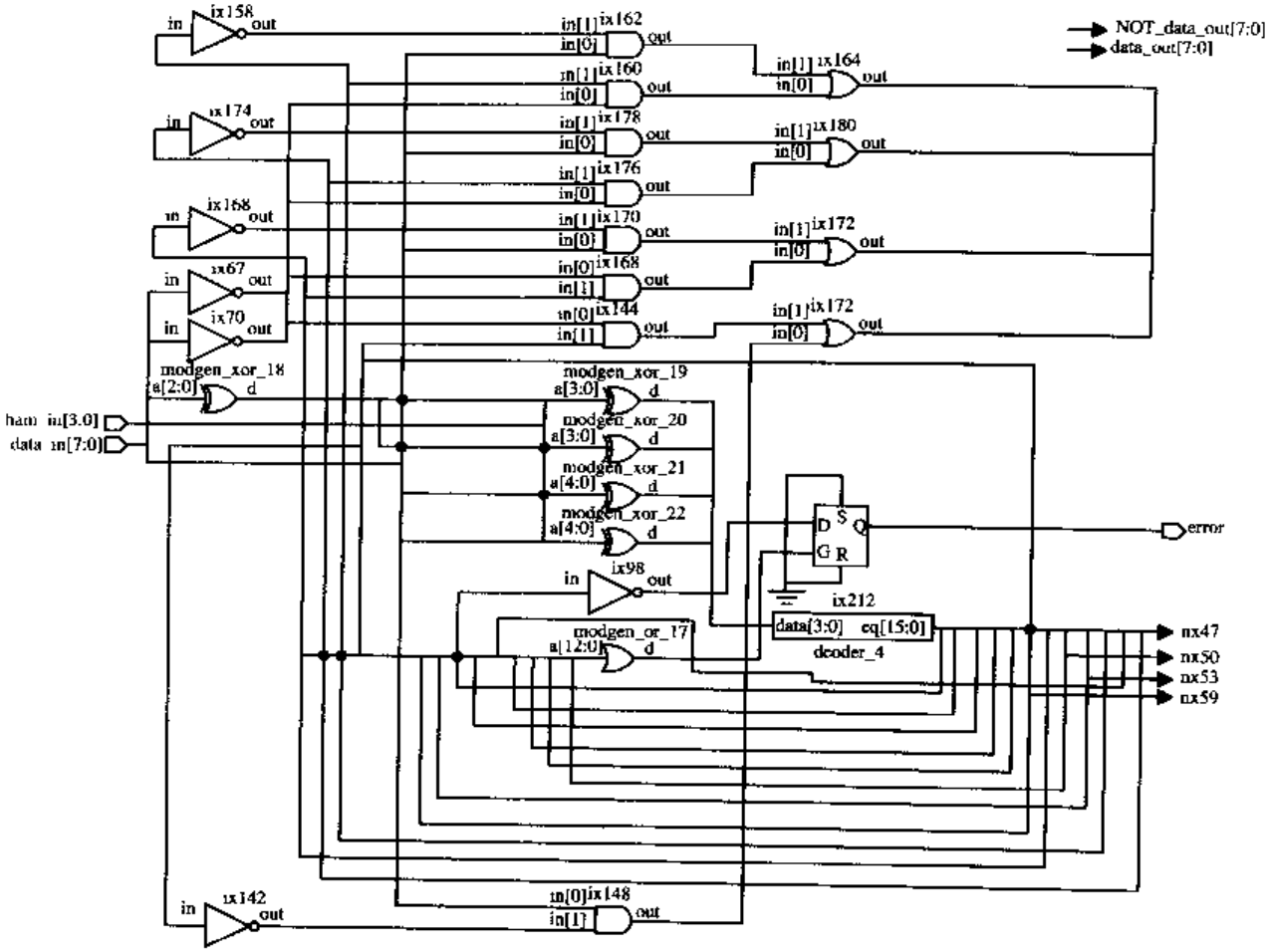
下面是解码器的一个实现代码。

```
//*************************************************************/
// MODULE:hamdec
// 该模块实现了 8 位数据的汉明码解码器
/*************************************************************/
// 常量定义
`define DELAY   1          // 时钟到输出的延迟
```

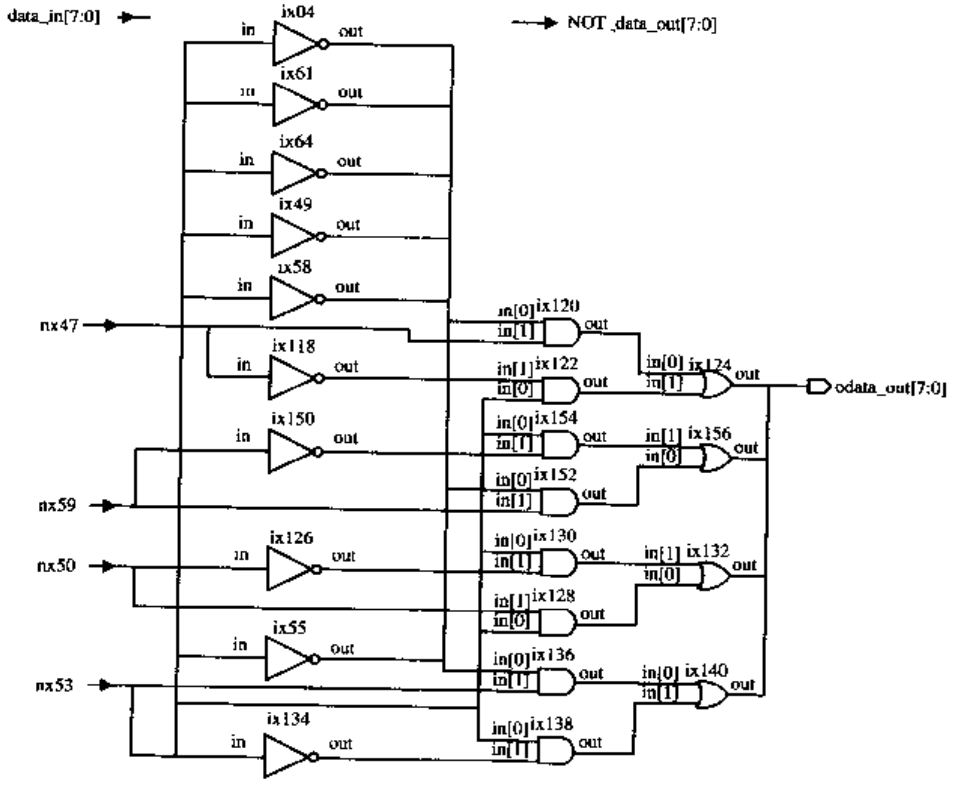
```
//模块描述
module hamdec(data_in,ham_in,data_out,error);
//端口描述
input [7:0]    data_in;              //输入数据
input [3:0]    ham_in;               //输入的纠错码比特

output [7:0]   data_out;             //输出数据
output         error;                //是否发生错误

//信号声明
wire [7:0]     data_in;
wire [3:0]     ham_in;
reg  [7:0]     data_out;
reg            error;

wire [3:0]     temp;

//赋值
 assign #DELAY temp[0] = ham_in[3] ^ data_in[7] ^ data_in[6] ^ data_in[4] ^ data_in[3] ^ data_in[1];
 assign #DELAY temp[1] = ham_in[2] ^ data_in[7] ^ data_in[5] ^ data_in[4] ^ data_in[2] ^ data_in[1];
assign #DELAY temp[2] = ham_in[1] ^ data_in[6] ^ data_in[5] ^ data_in[4] ^ data_in[0];
assign #DELAY temp[3] = ham_in[0] ^ data_in[3] ^ data_in[2] ^ data_in[1] ^ data_in[0];
//由输入数据重新产生纠错码，并与原纠错码异或，如果相同，则得到0，不同则为1

always @(temp or data_in) begin
    data_out = data_in;
    case (temp)
        4'h0: begin    //0000 完全符合，无错
            error = 0;
        end
        4'h1: begin    //0001，
            error = 1;
        end
        4'h2: begin    //0010，
            error = 1;
        end
        4'h4: begin    //0100
            error = 1;
        end
```

```
            4'h8: begin    //1000
                error = 1;
            end
            4'h3: begin    //0011
                data_out[7] = ~data_in[7];
                error = 1;
            end
            4'h5: begin    //0101
                data_out[6] = ~data_in[6];
                error = 1;
            end
            4'h6: begin    //0110
                data_out[5] = ~data_in[5];
                error = 1;
            end
            4'h7: begin    //0111
                data_out[4] = ~data_in[4];
                error = 1;
            end
            4'h9: begin    //1001
                data_out[3] = ~data_in[3];
                error = 1;
            end
            4'ha: begin    //1010
                data_out[2] = ~data_in[2];
                error = 1;
            end
            4'hb: begin    //1011
                data_out[1] = ~data_in[1];
                error = 1;
            end
            4'hc: begin    //1100
                data_out[0] = ~data_in[0];
                error = 1;
            end
        endcase
    end
    endmodule
```

汉明码解码器的综合结构如图 4.6 所示。



(a)



(b)

图 4.6　汉明码解码器

(a) 汉明码解码器第一部分；(b) 汉明码解码器第二部分

4.1.3 CRC 码

CRC 是一种广泛用于检错的重要的循环码。

CRC 的概念可以简单地概述为：一个二进制比特流(1，0 模式)可以被看作代表一个多项式的系数。当这个多项式移位后，被一个生成多项式(除数)所除，就产生了余数多项式，这个余数多项式就是 CRC。

产生的校验用的监督码(即 CRC 码)附在信息后边，构成一个新的二进制码序列数，最后发送出去。在接收端，则根据信息码和 CRC 码之间所遵循的规则进行检验，以确定传送中是否出错。

要理解 CRC，必须先了解循环码。这里首先对循环码的编解码进行说明。

假设有一个信息组，为 101，可以用多项式表示为 $m(x)=x^2+1$。而生成的多项式为 $g(x)=x^4+x^3+x^2+1$。

首先，将 m(x)移位，移位次数是 g(x)的最高次数，即

$$x^4g(x)=x^4(x^2+1)=x^6+x^4$$

用它除以 g(x)，得到的余数多项式 r(x)=x+1。因此，得到的编码为：

$$C(x)=x^4m(x)+r(x)=x^6+x^4+x+1$$

对于上述的循环码编码过程，乘 x^4 可以用移位寄存器的移位来实现，而除以 g(x)可以用 g(x)的多项式除法电路来实现。也就是说，循环码的编码可以用移位寄存器组成的 g(x)除法电路来实现。

图 4.7 是一个生成多项式 g(x)的循环码编码器的框图。设 g(x)的最高次数是 m。

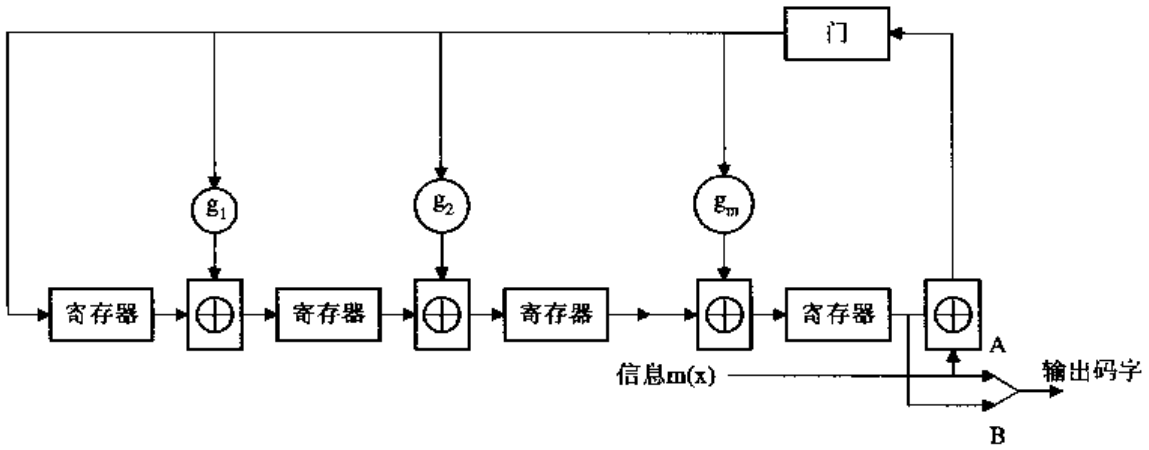


图 4.7 循环码编码器

进行编码时，数据由移位寄存器的右端输入，等价于用 x^m 乘以 m(x)。编码运算过程如下：

(1) 开关置于 A，门打开。k 位数据移入寄存器，同时送到外部。当 k 位信息全部送到外部时，除法运算也正好进行完，寄存器中的 m 个数据就构成了余项的系数序列，即形成了校验码。

(2) 开关置于 B，门关闭。这时候，除法反馈被切断。电路变成了一个 m 级的移位寄存器。

(3) 将移位寄存器的每一位逐位输出到信道中。这些校验码与原来的数据位一起构成了完整的码字。

图 4.8 给出一个循环码编码器的示例。生成多项式为 $g(x)=x^3+x+1$。

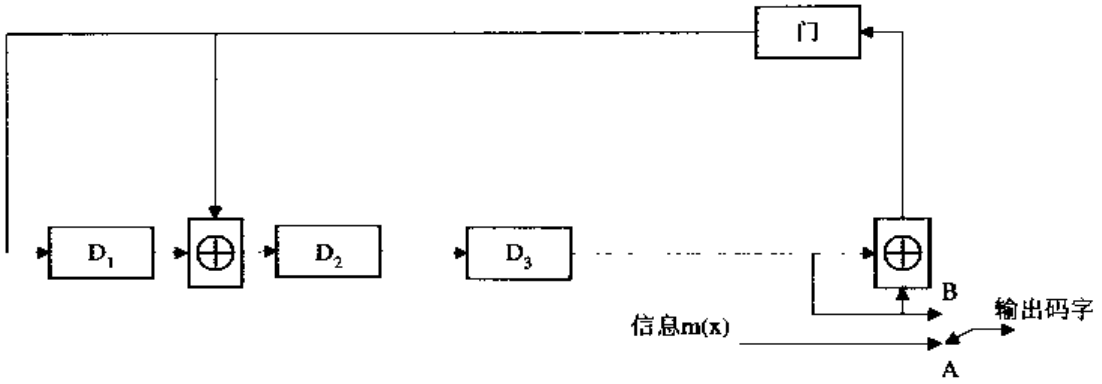


图 4.8　循环码编码器示例

设待编码的信息为 1011，对应的多项式为 $C(x)=x^3+x^2+1$。图中的三级移位寄存器分别用 D_1，D_2，D_3 表示。该编码器的工作过程如表 4.7 所示。

表 4.7　编码器工作过程

序号	输入	反馈 f	$D_1=f$	$D_2=f\oplus D_1$	$D_3=D_2$
0	0	0	0	0	0
1	1	0	1	1	0
2	1	1	1	0	1
3	0	0	1	0	0
4	1	0	1	0	0

由表 4.7 可见，最后寄存器中的值为 100(右边为高)，因此，输出的数据为 1001011(右边为高)。

对于 CRC 码，生成多项式主要有三个，其中 CRC-16 和 CRC-CCITT 产生 16 位的 CRC 码，而 CRC-32 则产生的是 32 位的 CRC 码。

CRC-16：(美国二进制同步系统中采用)　$G(X)=X^{16}+X^{15}+X^2+1$

CRC-CCITT：(由欧洲 CCITT 推荐)　$G(X)=X^{16}+X^{12}+X^5+1$

CRC-32：　$G(X)=X^{32}+X^{26}+X^{23}+X^{22}+X^{16}+X^{12}+X^{11}+X^{10}+X^8+X^7+X^5+X^4+X^2+X^1+1$

16 位的 CRC 码产生的过程是：先将要发送的二进制序列数左移 16 位(即乘以 2^{16})后，再除以一个多项式，最后得到的余数即是 CRC 码，如下式所示，其中，B(X)表示 n 位的二进制序列数，G(X)为多项式，Q(X)为整数，R(X)为余数(即 CRC 码)。

$$\frac{B(X)\cdot 2^{16}}{G(X)} = Q(X) + \frac{R(X)}{G(X)}$$

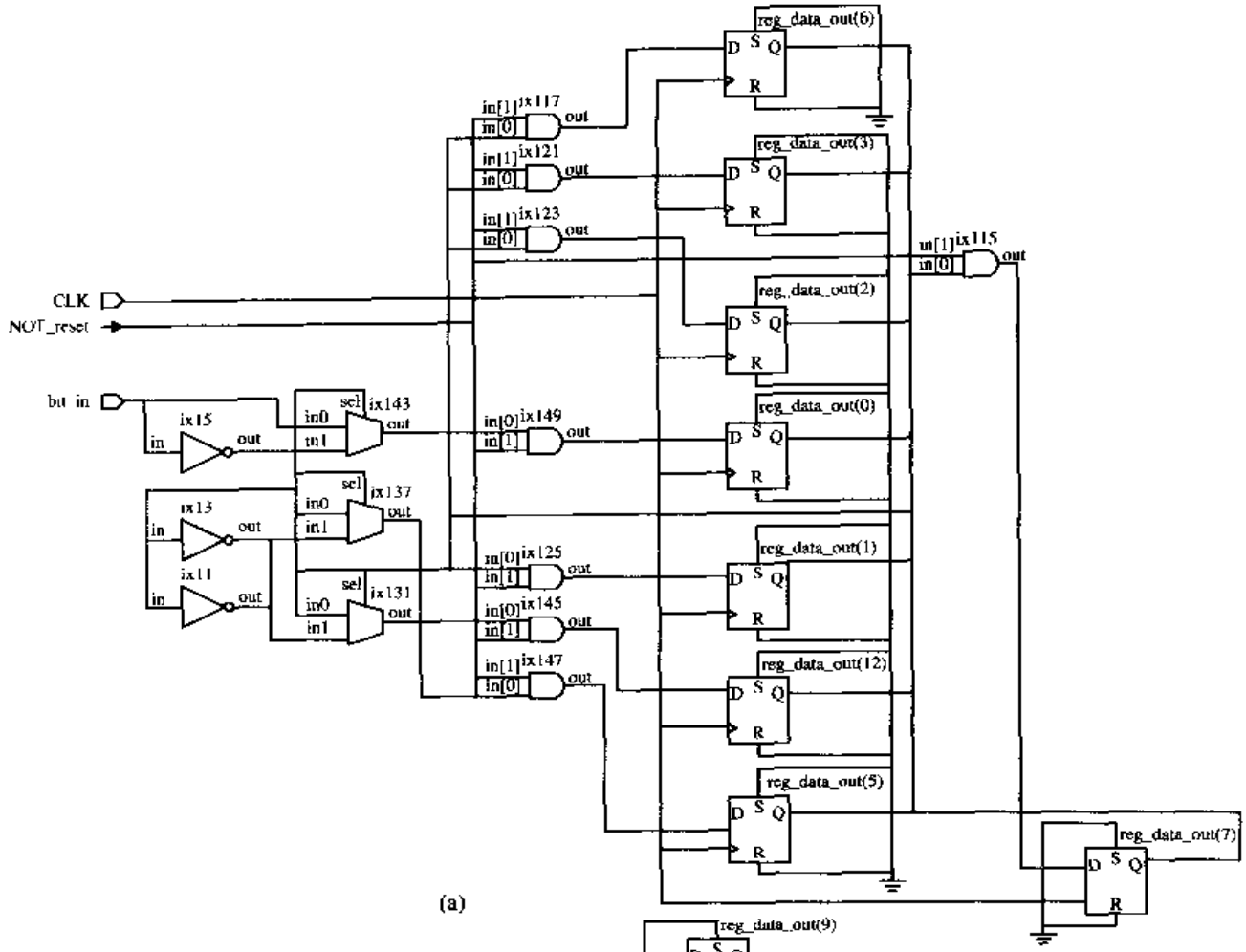
接收方将接收到的二进制序列数(包括信息码和 CRC 码)除以多项式，如果余数为 0，则说明传输中无错误发生，否则说明传输有误。

下面给出一个 CRC 的实现代码，所用的生成函数为 $G(X) = X^{16} + X^{12} + X^5 + 1$。

```
`define DEL      1
`define TAPS     16'b1000100000010000   //生成函数的系数
module     CRC(clk,   reset,bit_in,data_out);
// INPUTS
input  clk;                  // Clock
input  reset;                // Synchronous reset
input  bit_in;               // Input bit stream

output [7:0] data_out;       // data_out output
wire         clk;
wire         reset;
wire         bit_in;
reg [7:0]    data_out;

always @(posedge clk) begin
      if (reset) begin
      data_out <= #`DEL 8'h0;
      end
      else begin
            if (data_out[7]) begin
                  data_out <= #`DEL (data_out ^ `TAPS) << 1;
                  data_out[0] <= #`DEL ~bit_in;
            end
            else begin
                  data_out <= #`DEL data_out << 1;
                  data_out[0] <= #`DEL bit_in;
            end
   end
end
endmodule
```

CRC 的综合结果如图 4.9 所示。

在某些情形下，为了获得高速度，也可以采用组合逻辑直接得到 CRC。具体实现这里略去。



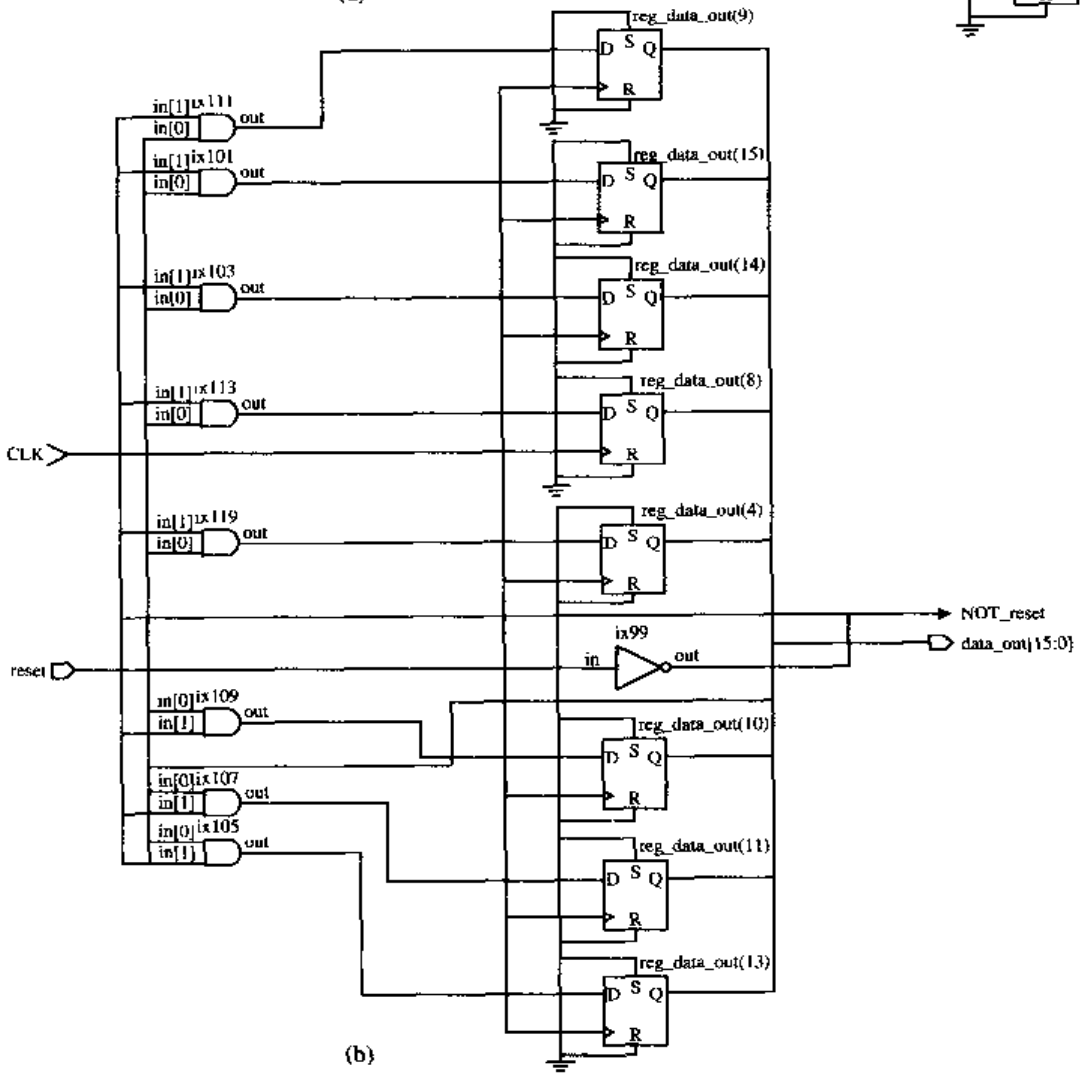


图 4.9　CRC 的综合结构

(a) 第一部分；(b) 第二部分

4.2 基本数学逻辑

4.2.1 加法器

1. 半加器和全加器

半加器的功能可以表示为：

$$C_{i+1} = A_i B_i$$
$$S_i = A_i \oplus B_i$$

图 4.10 给出了半加器的实现。

全加器的实现非常简单。它具有三个输入 A_i、B_i、C_i 和两个输出 S_i、C_{i+1}。

可用数学公式表示为：

$$C_{i+1} = A_i B_i + A_i C_i + B_i C_i$$
$$S_i = A_i \oplus B_i \oplus C_i$$

图 4.11 给出了全加器的实现。

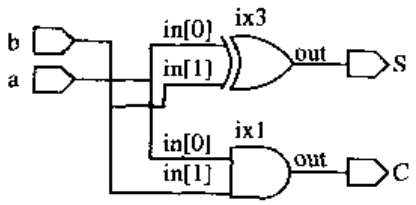


图 4.10 半加器

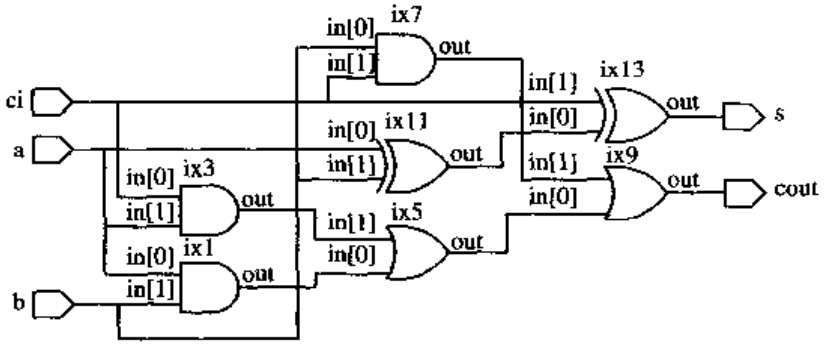


图 4.11 全加器的综合结果

2. 行波进位加法器

行波进位加法器(CPA)是最简单的加法器结构。它是将全加器串接起来，并将前一个全加器的进位输出作为后一个全加器的进位输入。

k=6 时的 CPA 结构如图 4.12 所示。

整个 CPA 的延迟是单个 FA 延迟的 k 倍，且其面积也是单个 FA 面积的 k 倍。对 CPA 的规模进行扩展很容易，只需从最高有效位处开始增加单元。

通过使用 2 的补码运算，CPA 也能完成减法。

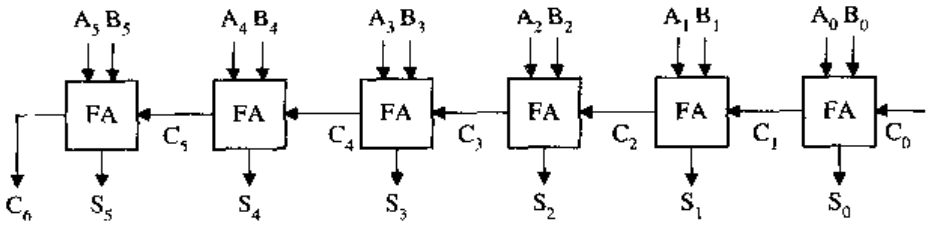


图 4.12　CPA 加法器结构

对 k 位 CPA，我们能将 $[0, 2^{k-1}-1]$范围内的正数编码为二进制向量的形式，且其最高有效位 MSB 为 0。

我们也能编码负数，其最高有效位 MSB 为 1。令 $0 \leqslant x \leqslant 2^{k-1}$，则-x 可表示为 $2^k - x$。

例如，k=3，则 0、1、2、3 编码为 000、001、010、011。下面给出-1、-2、-3、-4 的编码：

-1：$2^3-1=7=111$；-2：$2^3-2=6=110$；-3：$2^3-3=5=101$；-4：$2^3-4=4=100$

该编码方法在模运算中有两个优点：

- 符号位检测很容易，最高位即为符号位。
- 减法很容易：计算 x-y：首先用补码的形式编码-y，然后再加上 x。

CPA 的缺点是：计算时间太长。在计算量较大的应用中，例如，在构造加密芯片时，就需要用更复杂的加法器结构。

3．进位旁路加法器

进位旁路是指在特定条件下，提前得到进位结果。

首先，定义两个在加法器中经常用到的变量 G_i、P_i，作为生成(generate) 和传递(propagate)函数。

$$G_i = A_iB_i$$
$$P_i = A_i + B_i$$

根据 G_0、P_0 和 C_0 来展开 C_1：

$$C_1 = A_0B_0 + C_0(A_0 + B_0) = G_0 + C_0P_0$$

同理可得：

$$C_2 = A_1B_1 + C_1(A_1 + B_1) = G_1 + C_1P_1$$
$$C_3 = G_2 + C_2P_2$$
$$C_4 = G_3 + C_3P_3$$

得到的结构如图4.13所示。

可以发现，当 $P_0=P_1=P_2=P_3=1$ 时，A_i、B_i 中，一个为 1，一个为 0，因此 G_0、G_1、G_2、G_3 都为 0，所以，$C_4 = C_0$。这时候，可以采用如图 4.14 所示的进位结构。这就是进位旁路加法器的基本思想。

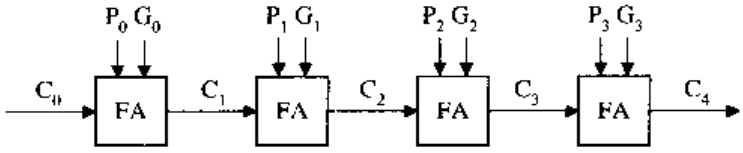


图 4.13　普通加法器进位结构

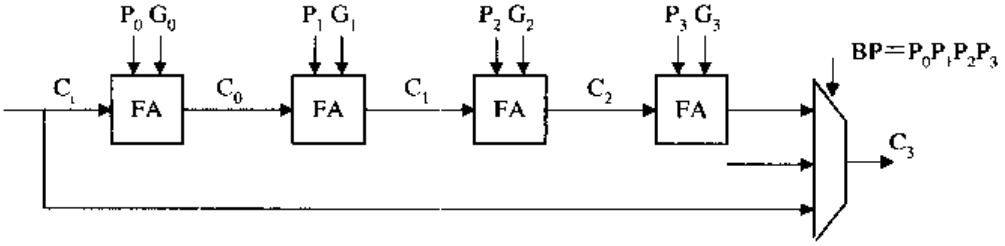


图 4.14　进位旁路加法器的结构

4．线性进位选择加法器

行波进位加法器，每一个全加器必须等待到达的进位，直到可以产生一个输出进位为止。避免这种线性关系的一种方法是：预测进位输入及结果这两个值。一旦知道输入进位的实际值，就可以立刻从多路选择器中得出正确的结果。这种加法器一般称为线性进位选择加法器。该加法器的结构如图 4.15 所示。

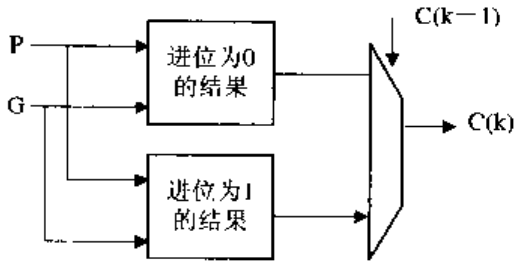


图 4.15　线性进位选择加法器的结构

5．平方根进位选择加法器

下面说明平方根进位选择加法器的原理。图 4.16 是一般的进位选择加法器，每一阶段要处理 4 位，对 16 位数进行操作，得到的最终进位的延时是 9。而在图 4.17 中，第一阶段处理 2 位，第 2 阶段处理 3 位，第 3 阶段处理 4 位，第 4 阶段处理 5 位……，对 20 位数进行操作，得到的最终延时是 8。可见这种结构能加快处理速度。

6．超前进位加法器(CLA)

CLA 利用 C_i、A_i、B_i 之间的关系，在求和前计算进位位 C_i。

可以根据 G_0、P_0 和 C_0 来展开 C_1：

$$C_1 = A_0B_0 + C_0(A_0 + B_0) = G_0 + C_0P_0$$

同理可求：

$$C_2 = A_1B_1 + C_1(A_1 + B_1) = G_1 + C_1P_1 = G_1 + (G_0 + C_0P_0)P_1 = G_1 + G_0P_1 + C_0P_0P_1$$

$$C_3 = G_2 + G_1P_2 + G_0P_1P_2 + C_0P_0P_1P_2$$

$$C_4 = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3 + C_0P_0P_1P_2P_3$$

CLA 逻辑使用这些函数是为了提前计算所有的 C_i，然后将这些值反馈到异或阵列来计算 S。

$$S_i = A_i \oplus B_i \oplus C_i$$

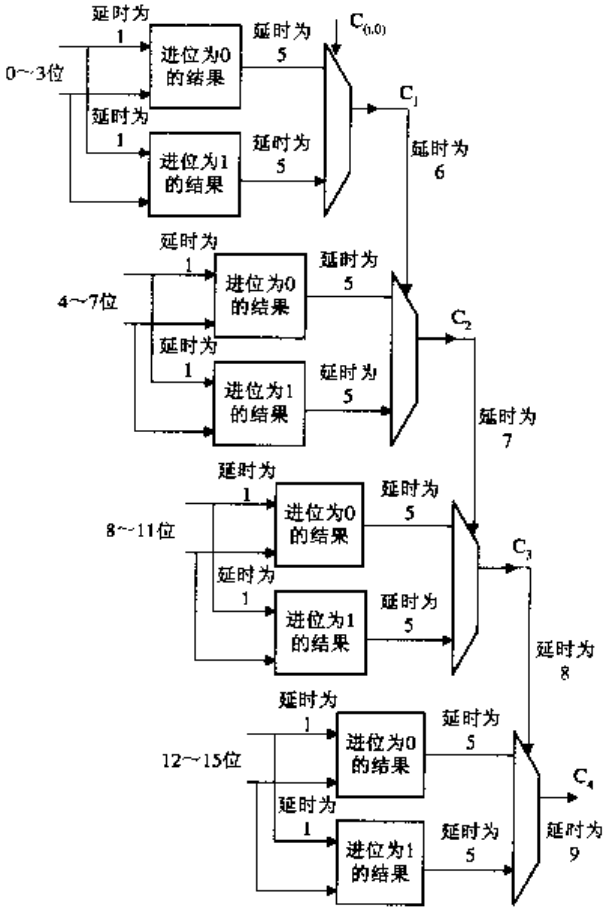


图 4.16　进位操作的线性配置

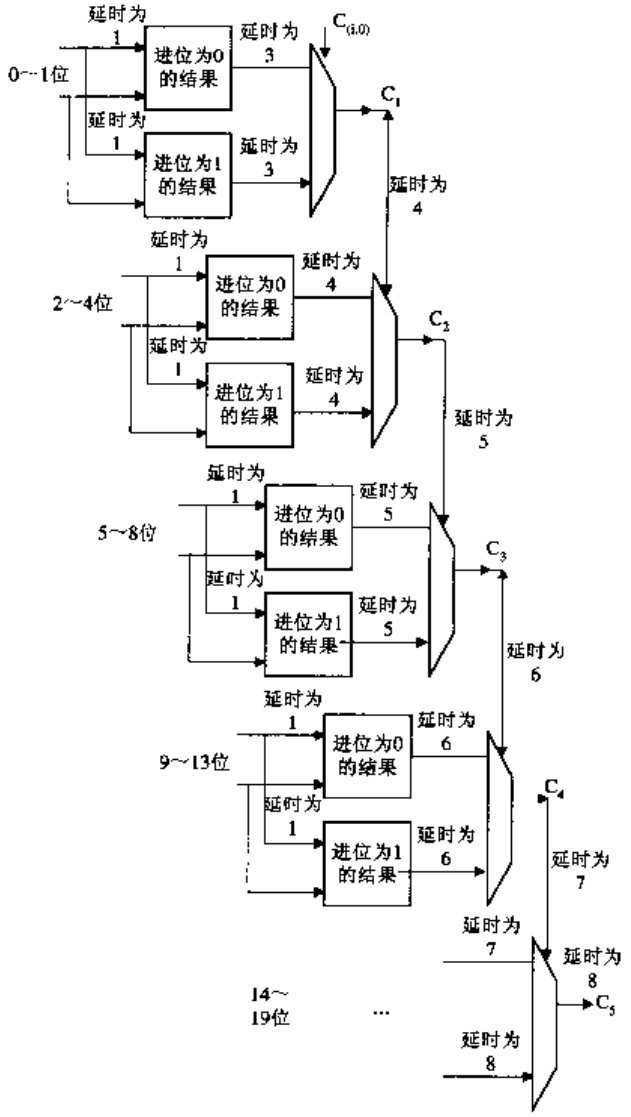


图 4.17　进位操作的平方根配置

当 k=3 时，CLA 的结构如图 4.18 所示。

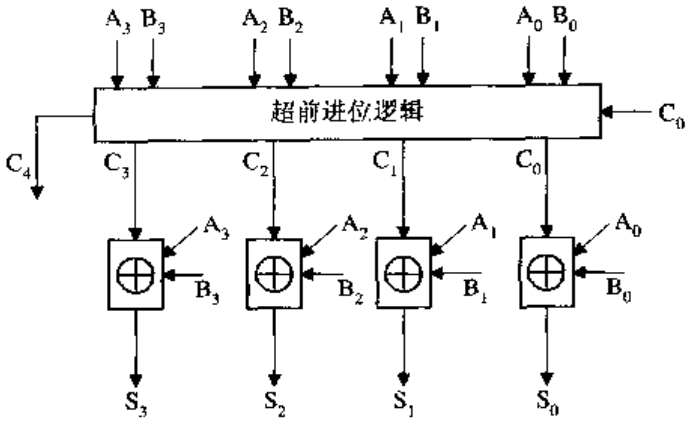


图 4.18　CLA 的结构

CLA 不太容易扩展，为了处理大数操作，可采用两种方法：

- Block CLA：首先建立小的 4 位或 8 位 CLA，然后再以此为模块建立大的 CLA。
- Complete CLA：将 CLA 功能公式化，以便使用并行的预固定好的电路。

CLA 的总延迟时间是 O(lgk)，速度比 CPA 快很多，但面积却增大了。Block CLA 面积为 O(k lgk)，Complete CLA 面积为 O(k)。

当操作数大于 256 位时，CLA 并不经济。此时可考虑用 CSA(Carry Save Adder)。

7. **对数超前进位加法器**

对数超前进位加法器的基本思想是：将 G_0、P_0、G_1、P_1…等操作数按二叉树的形式两两运算。图 4.19 说明了这种思想。可以很容易地看出，图 4.19(b)中的结构速度更快。

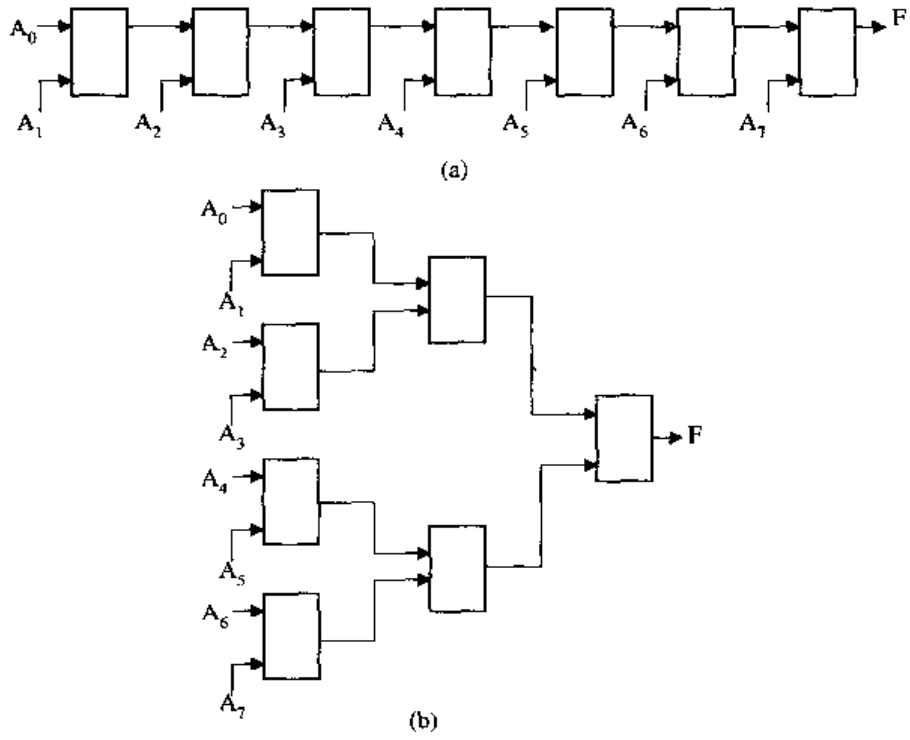


图 4.19　线性结构与树型结构

(a) 线性加法结构；(b) 树型加法结构

这种思想应用到加法器中，可以得到对数超前进位加法器的结构，如图 4.20 所示。

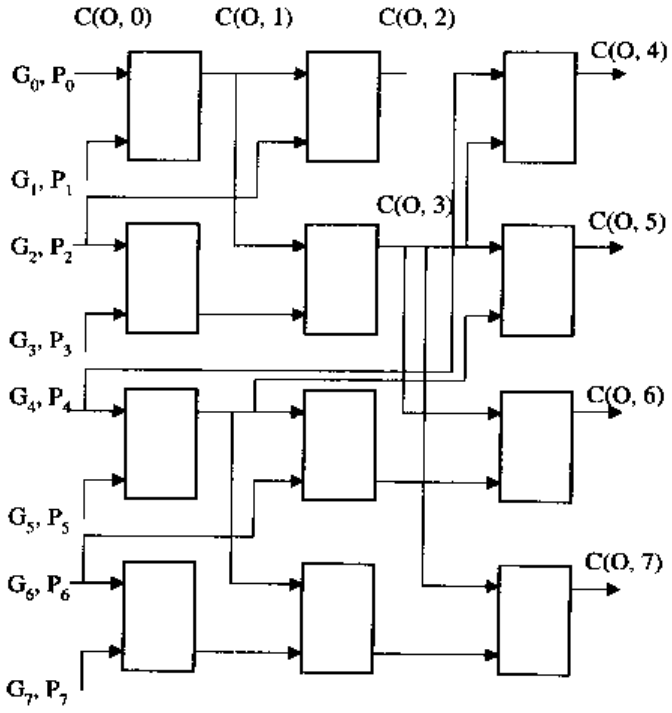


图 4.20　对数超前进位加法器结构

8．**进位保存加法器**(CSA)

CSA 实际上是将 3 个数相加的和用两个数表示出来。CSA 是最有用的加法器，是由一组并行的 k 位全加器组成，不需任何水平连接，可表示为：

$$C' + S = A + B + C$$

例如：A=40,B=25,C=20,可得：

A =40 = 101000

B =25 = 011001

C =20 = 010100

S =37 = 100101

C'=48 =0 11000

可表示为：

$$S_i = A_i \oplus B_i \oplus C_i$$

$$C_i' = A_iB_i + A_iC_i + B_iC_i$$

即一个 CSA 只是一个全加器单元。

当 k=6 时，CSA 的结构如图 4.21 所示。

因为输入 A、B、C 是并行的，所以 CSA 的总延迟时间等于单个全加器的延迟时间。因此，三个数相加产生两个数只需一个 FA 延迟时间，且 CSA 只需 k 倍 FA 的面积。通过加更多的并行的单元，可很容易地扩大规模。也可使用 2 的补码编码来完成减法的运算。

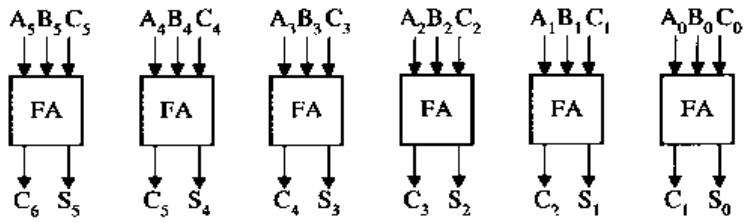


图 4.21　保存进位加法器的结构

CSA 有两个缺点：

- 并没有真正解决两数相加产生一个输出的问题。相反，它完成了三个数相加产生的两个结果，且这两个结果的和才是三数相加的和。因此该方法不适合普通的加法。
- 符号位的检测是困难的。因为当和表示为(C', S)对时，我们并不知道 C'+S 的符号。因此需采用一种方法来解决符号位的检测。可以通过增加 C、S 的 MSB 来估计符号位。

例如：A=−18，B=19，C=6，可得 S=−5，C'=12。因为 A+B+C=C'+S=12−5=7，它的符号位是 0。计算过程如图 4.22 所示。

A=	−18=		1	0	1	1	1	0
B=	19=		0	1	0	0	1	1
C=	6=		0	0	0	1	1	0
S=	5=		1	1	1	0	1	1
C'=	12=	0	0	0	1	1	0	
			1					(1 MSB)
			1	1				(2 MSB)
			0	0	0			(3 MSB)
			0	0	0	1		(4 MSB)
			0	0	0	1	1	(5 MSB)
			0	0	0	1	1	1 (6 MSB)

图 4.22　利用 CSA 进行加法

只有在第一次 3 个最高有效位相加计算得到的符号位才是正确的。

9. 进位延迟加法器(CDA)

CDA 是两级 CSA，可用来减少乘法的复杂性。CDA 产生一个进位延迟对(D，T)，可表示为：

$$S_i = A_i \oplus B_i \oplus C_i$$

$$C_{i+1} = A_iB_i + A_iC_i + B_iC_i$$

$$T_i = S_i \oplus C_i$$

$$D_{i+1} = S_iC_i$$

其中，D_0=0。

注意：C_{i+1} 和 S_i 是输入为 A_i、B_i、C_i 的全加器的输出，D_{i+1} 和 T_i 是半加器的输出。CDA 的一个重要特性是 $D_{i+1}T_i$=0，i=0,1，…，k−1。因为

$$D_{i+1}T_i = S_iC_i(S_i \oplus C_i) = S_iC_i(\overline{S_i}C_i + S_i\overline{C_i}) = 0$$

例如：A=40，B=25，C=20。首先计算 CS 对(C，S)。然后再利用 $T_i = S_i \oplus C_i$，$D_{i+1} = S_iC_i$ 计算(D，T)对。计算过程如图 4.23 所示。

A=	40=		1	0	1	0	0	0
B=	25=		0	1	1	0	0	1
C=	20=		0	1	0	1	0	0
S=	37=		1	0	0	1	0	1
C=	48=	0	1	1	0	0	0	0
T=	21=		0	1	0	1	0	1
D=	64=	1	0	0	0	0	0	0

(a)

T = 21 =	0	1	0	1	0	1
D = 64 =	1	0	0	0	0	0	0
T_iD_{i+1} =	0	0	0	0	0	0

(b)

图 4.23　CDA 计算过程

(a) CDA 计算过程 1；(b) CDA 计算过程 2

所以，(D，T)对(64，21)表示 A+B+C=85，且 $D_{i+1}T_i = 0$。

在 Brickell 算法中将用到 CDA。

当 k=6 时，CDA 的结构如图 4.24 所示。

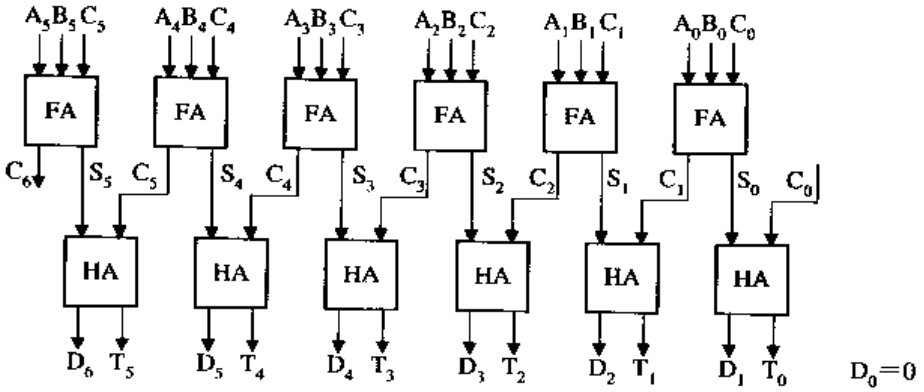


图 4.24　CDA 的结构

10．浮点数加法器

与整数加法器相比，浮点数加法器要复杂得多。这里给出浮点数加法器的原理图。代码留给读者自己来完成。浮点数加法器原理框图如图 4.25 所示。

说明：

数据可以用原码表示，也可以用补码表示。正数的补码是它本身，负数的补码是符号位取反加 1。原码乘法比较简单，但由于补码加减较原码加减简单，故数据在计算机中常采用补码表示。图 4.26 给出了加减法的硬件框图。加法器的两个操作数来自寄存器 A 与 B。输出结果通常存放在 A 中。

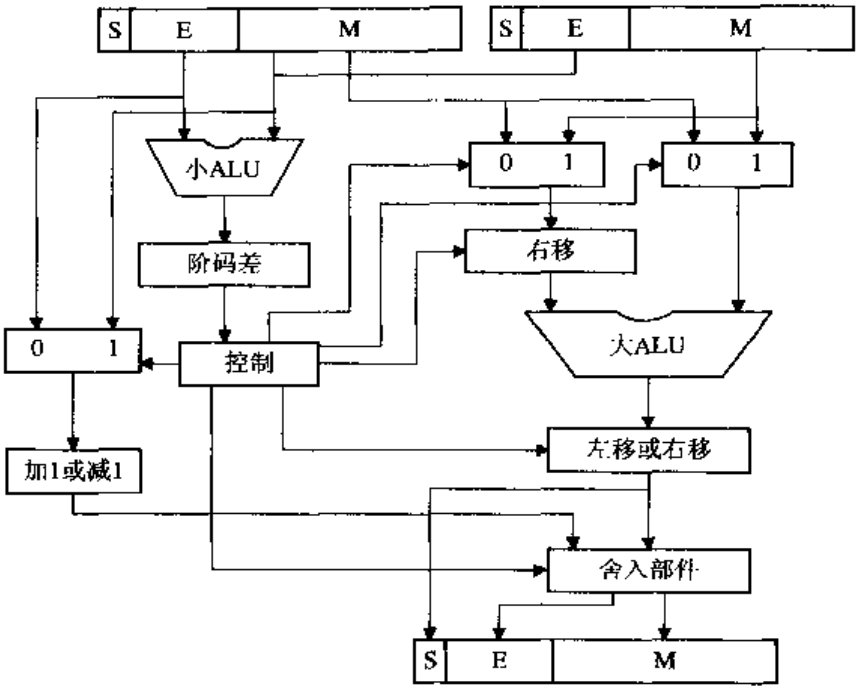


图 4.25　浮点加法器原理框图

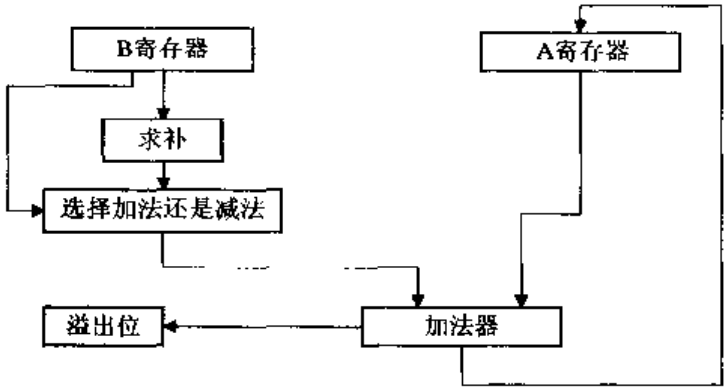


图 4.26　加减法硬件框图

如果同一运算部件对加减运算采用补码算法，而对乘除运算又采用原码算法，就得进行码制转换，不太方便，因此要研究补码乘法。补码乘法中，操作数与结果均以补码表示，连同符号位一起，按相应算法运算。利用补码，减法可以用加法来实现。

4.2.2　乘法器

乘法器对数据处理的速度有着显著影响。在高性能 CPU 和 DSP 中，乘法器的研究一直是个热点。现在已提出了多种实现算法。

与加减法相比，乘法的实现要复杂许多。最一般的方法是按照“手算”的方式。例如，要计算 2×3，其乘法过程如下：

$$
\begin{array}{r}
0010 \\
\times\ 0011 \\
\hline
0010 \\
0010 \\
0000 \\
0000 \\
\hline
0000110
\end{array}
$$

下面给出 32 位无符号二进制乘法的一种实现方案。乘数与被乘数分别载入两个寄存器 R_1 与 R_2 中。此外，还有一个寄存器 A，A 的初始值为 0。

运算时，控制逻辑每次读乘数的一位。若 R_1 的最低位为 1，则被乘数与 A 寄存器相加，并将结果存于 A 寄存器。然后，A、R_1 这两个寄存器整体右移一位，A 的最高位变成 0，且 A_0 进入 R_1 的最高位，而 R_1 的最低位丢弃。若 R_1 的最低位是 0，则只需进行移位，不进行加法。对原始的乘数的每一位重复上述过程，产生的 2n 位乘积存于 A 与 R_1 寄存器。图 4.27 给出了 32 位定点原码一位乘法的结构。图 4.28 给出了乘法流程图。

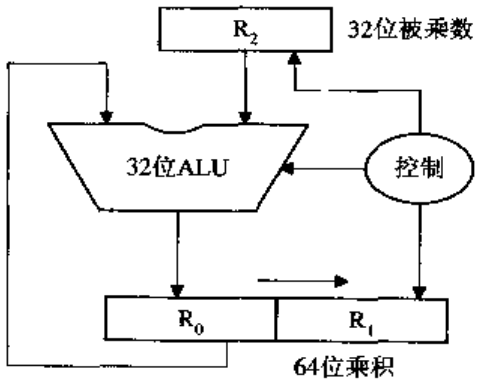


图 4.27　实现 32 位定点原码一位乘法的方案

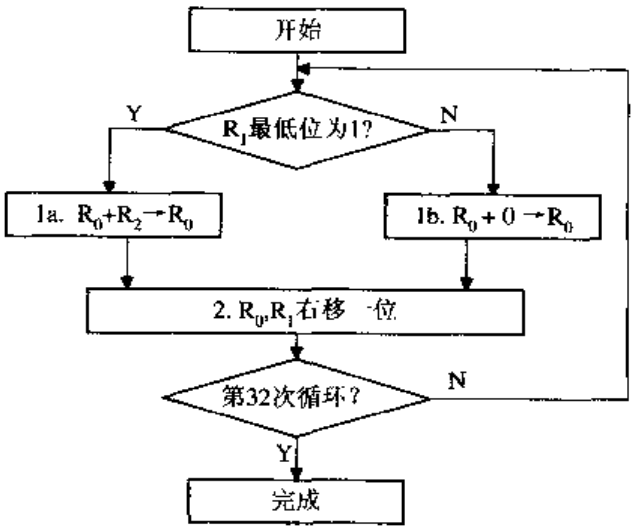


图 4.28　实现 32 位定点原码一位乘法的流程图

如果乘数或被乘数是负数，则不能采用上述方法。一种解决方法是，先把被乘数与乘数都变成正数，然后相乘。如果两个原始数符号不同，要将结果取为 2 的补码。此方法称为校正法。校正法较繁琐，一般不用。另一种方法是将校正法的两种情况结合起来，演变成比较法。该方法称为 Booth 算法。该算法不仅能用于处理负数相乘的情况，而且速度也较快，因此应用很广。

图 4.29 给出了 Booth 算法进行乘法的流程。乘数与被乘数分别载入 Q 和 M 寄存器内，同时，还有一个 1 位寄存器，位于 Q 寄存器最低位 Q_0 的右边，称为 Q'。乘法的结果出现在 A 和 Q 寄存器中。A 与 Q'初始值为 0。控制逻辑也是每次扫描乘数的一位，但同时它也要检查右边的一位。若两位相同(为 1-1 或者 0-0)，则 A、Q 和 Q'寄存器的所有位向右移一位。若两位不同，则根据两位是 1-0 还是 0-1，决定被乘数加到 A 寄存器，还是由 A 寄存器减去被乘数，加减之后再右移一位。也就是说，右移总是要进行的。这里的右移是算术移位，即如果 A_{n-1} 移到 A_{n-2} 之后，原来的值仍保留在 A_{n-1} 中。

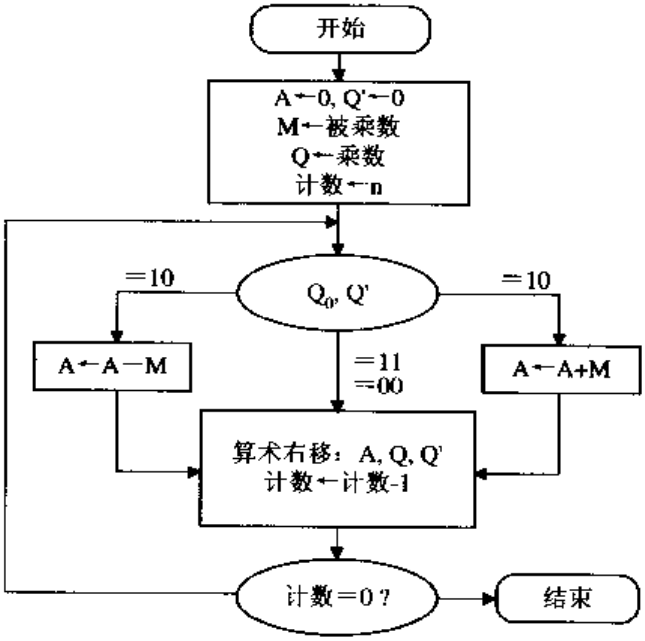


图 4.29　Booth 算法进行乘法的流程

图 4.30 给出 Booth 算法的一个示例。

A	Q	Q'	M		
0000	0011	0	0111	初始	
1001	0011	0	0111	A←A-M	第 1 周期
1100	1001	1	0111	移位	
1110	0100	1	0111	移位	第 2 周期
0101	0100	1	0111	A←A+M	第 3 周期
0010	1010	0	0111	移位	
0001	0101	0	0111	移位	第 4 周期

图 4.30　Booth 算法的一个示例

Booth 乘法器留给读者来完成。为了进一步提高乘法运算速度，可采用类似于人工计算的方法，用阵列乘法器实现乘法运算。该方案所用加法器数量很多，但内部结构规则性强，适于用超大规模集成电路。

4.2.3 除法器

除法要比乘法更复杂。除法运算符是不可综合的。如果需要，设计者可自己来完成除法器。

要完成 7/2，手工计算过程如下：

$$
\begin{array}{r}
0011 \\
0010\overline{)00000111} \\
0010 \\
\hline
00011 \\
0010 \\
\hline
0001
\end{array}
$$

下面给出用硬件实现除法的过程。除数放入 R_2 寄存器，被除数放入 R_1 寄存器，R_0 最初为 0。实现过程如图 4.31 所示。最后，商放在 R_1 中，余数放在 R_0 中。

为了说明除法流程，我们以 7/2 为例来说明计算过程，如表 4.8 所示。。计算步骤请参看图 4.31。

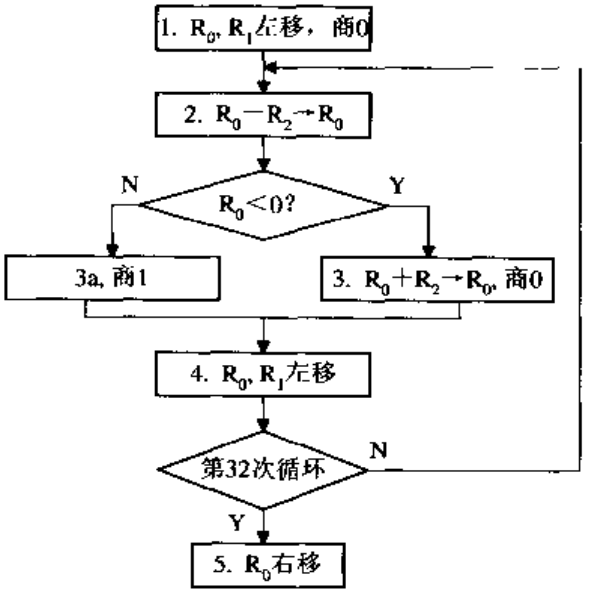


图 4.31 实现 32 位定点原码一位除法的流程图

表 4.8 除法计算过程

循环	步骤	余数(R_0 R_1)
0	初始值	0000 0111
	1. 左移，商 0	0000 1110
1	2. 减 0010	1110 1110
	3b. 加 0010，商 0	0000 1110
	4. 左移 1 位	0001 1100
2	2. 减 0010	1111 1100
	3b. 加 0010，商 0	0001 1100
	4. 左移 1 位	0011 1000
3	2. 减 0010	0001 1000
	3a. 商 1	0001 1000
	4. 左移 1 位	0011 0001
4	2. 减 0010	0001 0001
	3a. 商 1	0001 0001
	4. 左移 1 位	0010 0011
	5. R_0 右移	0001 0011

4.2.4 算术逻辑单元 ALU

1. 功能

ALU 是计算机的核心部件，可完成对数据的算术和逻辑运算功能。它在给出运算结果的同时，还给出结果的某些特征，如溢出否，有无进位，结果是否为零、为负等。ALU 还可以将参加运算的数据和中间结果暂存到内部的一组寄存器中。因为这些寄存器可被汇编程

序员直接访问与使用，通称通用寄存器。为了用硬件完成乘除指令运算，运算器内一般还有一个能自行左右移位的专用寄存器，通称乘商寄存器，这些部件通过几组多路选通门电路实现相互连接和数据传送。运算器与其它几个功能部件连接在一起，必须有接受外部数据输入和送 ALU 运算结果的逻辑电路。

图 4.32 给出了 ALU 实现的一种框架。

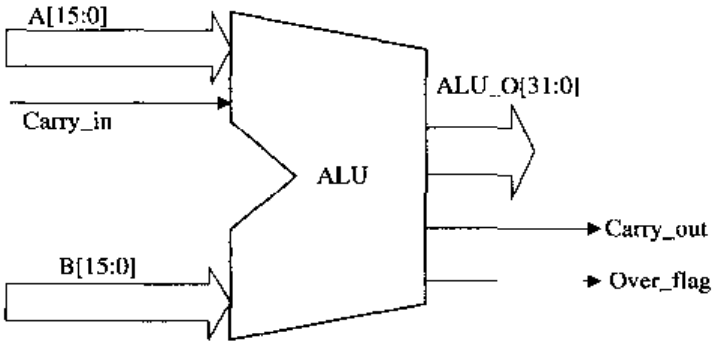


图 4.32 ALU 单元

2. 代码实现

下面是 ALU 的一种实现代码。

```
/**********************************
// addersub
//实现 ALU
**********************************/
module addersub(a,b,add,ci,sum,z,co,n,v);
input[1:0] a;
input add;
input [1:0] b;
input [1:0] ci;
output[1:0] sum;
output z;
output co;
output n;
output v;

reg [1:0] sum;
 reg co;
 reg ign;
 reg co0;
 reg z;
 reg n;
```

```
reg v;
always @(a or b or add or ci) begin
    if (add) begin                                  //加法器
        {co0,sum,ign} = {a,ci} + {b,1'b1};
        co = co0;                                   //co 进位，sum 结果
    end
    else begin                                      //减法器
        {co0,sum,ign} = {a,ci} - {b,1'b1};
        co = ~co0;
    end
    z = sum == 0;                                   //结果为 0，则 z=1
    n = sum[1];                                     //n 为结果的高位
    // sum[3] = a[3] ^ b[3] ^ c3 <===> c3 = sum[3] ^ a[3] ^ b[3]
    v = co0 ^ sum[1] ^ a[1] ^ b[1];
end
endmodule
```

ALU 综合出的结构如图 4.33 所示。

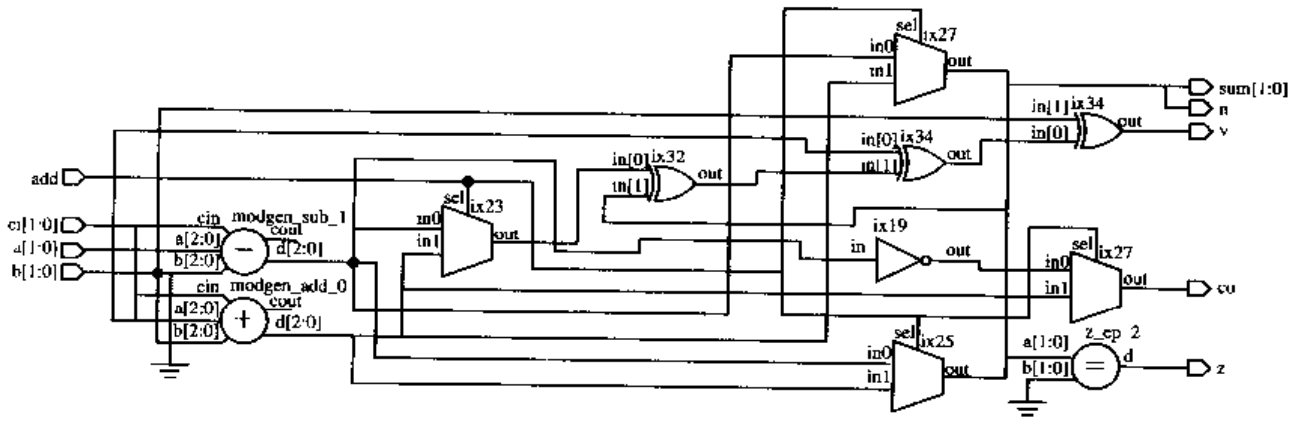


图 4.33　ALU 的综合结构

4.3　线性反馈移位寄存器

线性反馈移位寄存器的应用很广，可进行串/并转换、随机数生成、校验、加密等。

4.3.1　串/并转换模块的功能

1. 功能

串/并转换模块的框图如图 4.34 所示。

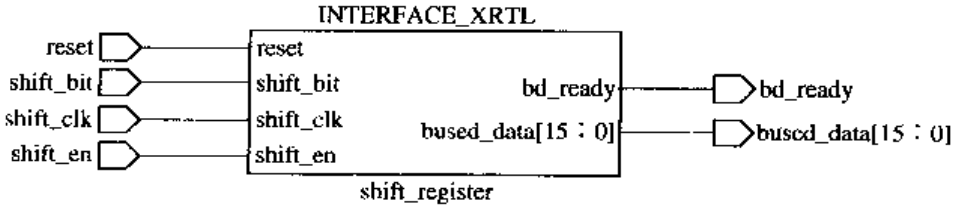


图 4.34　串/并转换模块的框图

该模块完成如下功能：

当移位使能信号 shift_en 为真时，在每个移位时钟 shift_clk 的上升沿，数据右移，输入的移位比特填到空出的最高位。计数器在每个移位时钟的上升沿加 1。计数器共 4 位，即 count 由 0000 计到 1111=15 后，bd_ready 变为 1。这时候，共移入了 16 位数据，即 16 位的输出数据 bused_data 已准备就绪。移位寄存器的端口说明如表 4.9 所示。

表 4.9　移位寄存器端口说明

端口	宽度	方向	说明
shift_bit	1	输入	输入移位比特
shift_en	1	输入	移位使能信号
shift_clk	1	输入	移位时钟
Reset	1	输入	移位加法器的复位信号
bused_data	16	输出	总线输出数据
bd_ready	1	输出	输出的准备标志信号

2. 实现代码

下面是线性移位寄存器的一种实现代码。

```
module shift_register (bused_data, bd_ready, shift_bit, shift_en, shift_clk,reset);
  output [7:0] bused_data;                    //总线输出数据
  output bd_ready;                            //输出的准备标志信号
  input shift_bit;                            //输入移位比特
  input shift_en;                             //移位使能信号
  input shift_clk;                            //移位时钟
  input reset;                                //移位加法器的复位信号

  reg [7:0] bused_data;
  reg bd_ready;
  reg [2:0] count;

parameter
        TRUE = 1'b1,
        FALSE = 1'b0;
```

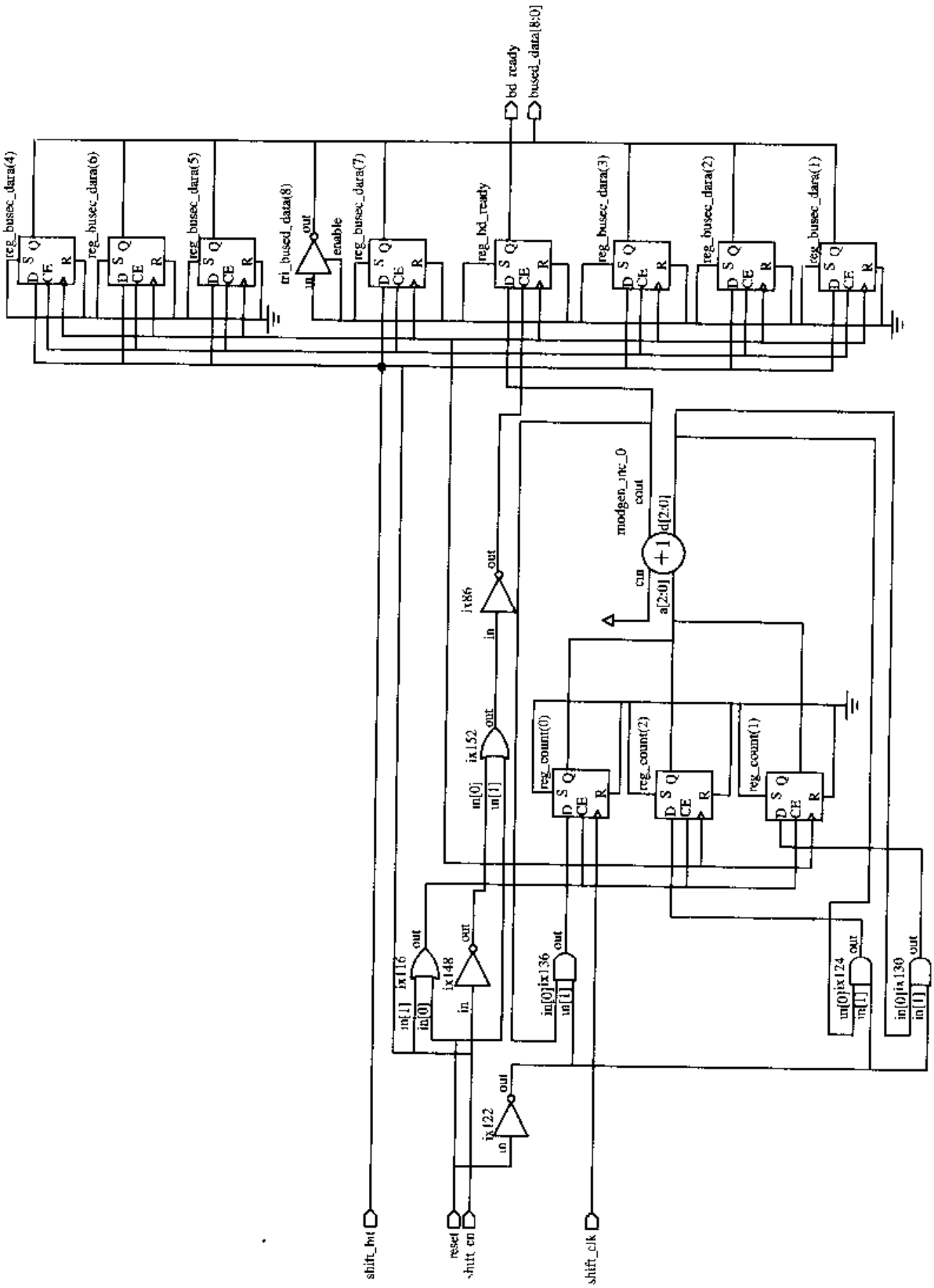


图 4.35 线性反馈移位寄存器综合结果

```
/*****************************************************
当移位使能为真时，在每个移位时钟，移位比特移入移位寄存器
*****************************************************/

always @ (posedge shift_clk)
begin
    if (shift_en)
    begin
        bused_data[6:0] <=  bused_data[7:1];
        bused_data[7] <= shift_bit;
    end
end

always @ (posedge shift_clk)
begin
  if (reset) begin
    count <= 0;          // count 为 3 位，即 count 由 000 计到 111=7 后，bd_ready 变为 1
  end                    // 即输出数据已准备好
  else if (shift_en) begin
    {bd_ready, count} <= count + 1;
  end
end

endmodule //of shift_register
```

线性反馈移位寄存器的综合结果如图 4.35 所示。

4.3.2 生成伪随机数

1. 功能

用线性反馈移位寄存器可以产生伪随机序列。图 4.36 给出了一种示范，这是用 4 级移位寄存器组成的结构。规定移位寄存器的状态是各级从右至左的顺序排列而成的序列。设初始状态是 0001，即 $a_{n-4}=0$，$a_{n-3}=0$，$a_{n-2}=0$，$a_{n-1}=1$。图 4.36 中的反馈逻辑为 $a_n=a_{n-1}\oplus a_{n-3}\oplus a_{n-4}$，初始状态为 0001，经过一个时钟节拍后，各级状态自左向右移到下一级，x_4 输出。与此同时，$x_1\oplus x_3\oplus x_4$ 送到移位寄存器第一级，从而形成移位寄存器的新状态。下一个时钟节拍到来时，又继续上述过程。x_4 输出的序列是一个周期为 15 的序列。对于级数为 r 的线性移位寄存器，周期 $p=2^r-1$(不包括全 0 的状态)。例如，如果线性反馈移位寄存器为 8 位，则输出序列的周期是 255；如果线性反馈移位寄存器是 16 位，则输出序列周期为 65535。这些输出可以看作是一个随机数。

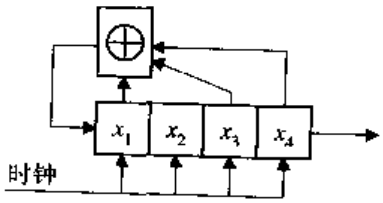


图 4.36 用 4 位移位寄存器构成的随机数发生器

对这种随机数发生器，有如下结论：

- 初始状态不能为 0，否则输出全是 0。
- 反馈逻辑不同，则输出序列不同。
- 初始值不同，则输出序列不同。

2. **实现代码**

下面是伪随机数的一种实现代码。在端口中有 TEST 引脚，用于给移位寄存器一个初值，这样方便将来的验证。该模块输出 8 位的伪随机数。

```
`timescale 1ns/10ps
module random (
        Reset,OutEn,RandomData,sendbegin,sendover,
        TEST,clk                //
        );

input           Reset;
input           clk;
input           TEST;                   //
input           OutEn;
output          [7:0] RandomData;
input           sendbegin;
output          sendover;

reg             sendover;
reg   [15:0]    RandomA;
reg   [1:0] Cnt;

assign      RandomData=(OutEn&& sendbegin)?
            {RandomA[0],RandomA[1],RandomA[2],RandomA[3],
                    RandomA[4],RandomA[5],RandomA[6],RandomA[7]}:8'hz;


wire        RandomACk;
assign      RandomACk=(sendbegin==0)?clk:OutEn;     //
always @(posedge RandomACk or posedge Reset or posedge TEST)
 begin
   if(Reset)
RandomA[3:0] <= 4'ha;
   else if(TEST)
     RandomA[15:0] <= 16'haaaa;
   else begin
     RandomA[15] <=RandomA[7] ^ RandomA[5] ^ RandomA[4] ^ RandomA[2];
     RandomA[14:8]<=RandomA[6:0];
     RandomA[7] <= RandomA[15];
     RandomA[6]<=RandomA[9] ^ RandomA[11] ^ RandomA[12] ^ RandomA[14];
```

```
        RandomA[5]<=RandomA[8] ^ RandomA[10] ^ RandomA[11] ^ RandomA[13];
        RandomA[4]<=RandomA[7] ^ RandomA[9] ^ RandomA[10] ^ RandomA[12];
        RandomA[3]<=RandomA[6] ^ RandomA[8] ^ RandomA[9] ^ RandomA[11];
        RandomA[2]<=RandomA[5] ^ RandomA[7] ^ RandomA[8] ^ RandomA[10];
        RandomA[1]<=RandomA[4] ^ RandomA[6] ^ RandomA[7] ^ RandomA[9];

        RandomA[0]<=RandomA[3] ^ RandomA[5] ^ RandomA[6] ^ RandomA[8];

    end
  end

wire        CntCk;
assign      CntCk= (sendbegin==1 && sendover==0)?OutEn:0;
always @(negedge CntCk or posedge Reset)
  begin
    if(Reset) begin
          Cnt <= 0;
          sendover <= 0;
    end
    else
          {sendover,Cnt} <= {sendover,Cnt} + 1;
  end

endmodule
```

4.3.3 产生定时标志信号

下面是线性反馈移位寄存器的另一种用处：每隔固定的时间产生标志信号。该用法是对伪随机数序列的一种扩展。

在上一节中，我们提到，r 位的线性反馈移位寄存器组可以产生 2^r-1 位的随机数序列。如果能够对移位寄存器的值进行控制，就能控制随机数产生的序列。下面给出说明。

图4.37是所用的线性反馈移位寄存器的框图，线性反馈移位寄存器的端口说明如表4.10所示。

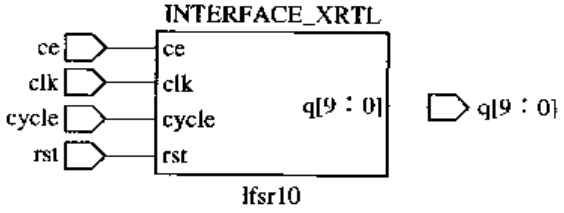


图 4.37　线性反馈移位寄存器

表 4.10　线性反馈移位寄存器的端口说明

端口	宽度	方向	说明
ce	1	输入	计数器时钟使能信号
clk	1	输入	全局时钟
cycle	1	输入	触发移位寄存器的输入，产生循环
rst	1	输入	全局异步复位
q	10	输出	计数器输出

1. 功能

要求每隔一定的时间，时间标志信号有效。

2. 代码

在每个移位时钟的上升沿，输出 q 左移，空出的最低位由 $\overline{q[9] \oplus q[6] \oplus cycle}$ 填充。cycle 为移位寄存器的输入。在此应用中，cycle 在一定周期内的某个时钟周期中有效，其余时间为 0。它可以影响线性反馈移位寄存器的输出，产生一个伪随机序列，每隔固定时间循环。

下面是产生时间标志信号的一种实现代码，以及产生伪随机序列的线性反馈移位寄存器的实现代码。

```
module test;
...
wire [9:0]  hc, vc;
wire        h0          = hc == 10'h31D;        //797
wire        hblank      = hc == 10'h1C4;        //452
wire        hsyncon     = hc == 10'h122;        //290
wire        hsyncoff= hc == 10'h3B6;            //950
wire        v0          = vc == 10'h27D;        //637
wire        vblank      = vc == 10'h01D;        //29
wire        vsyncon     = vc == 10'h3F5;        //1013
wire        vsyncoff= vc == 10'h3D7;            //983

lfsr10      hctr(.clk(clk), .rst(rst), .ce(1'b1), .cycle(h0), .q(hc));
lfsr10      vctr(.clk(clk), .rst(rst), .ce(h0), .cycle(v0), .q(vc));

...
endmodule

module lfsr10(clk, rst, ce, cycle, q);
    input       clk;        //全局时钟
    input       rst;        //全局异步复位信号
    input       ce;         //计数器时钟使能信号
    input       cycle;      //触发移位寄存器的输入，产生循环
    output      [9:0] q;    //计数器输出
    reg         [9:0] q;

    always @(posedge clk or posedge rst) begin
        if (rst)
            q <= 0;
```

```
        else if (c·
            q <= { q[8:0], ~(q[9] ^ q[6] ^ cycle) };
    end
endmodule
```

模块 lfsr10 的综合结果如图 4.38 所示。

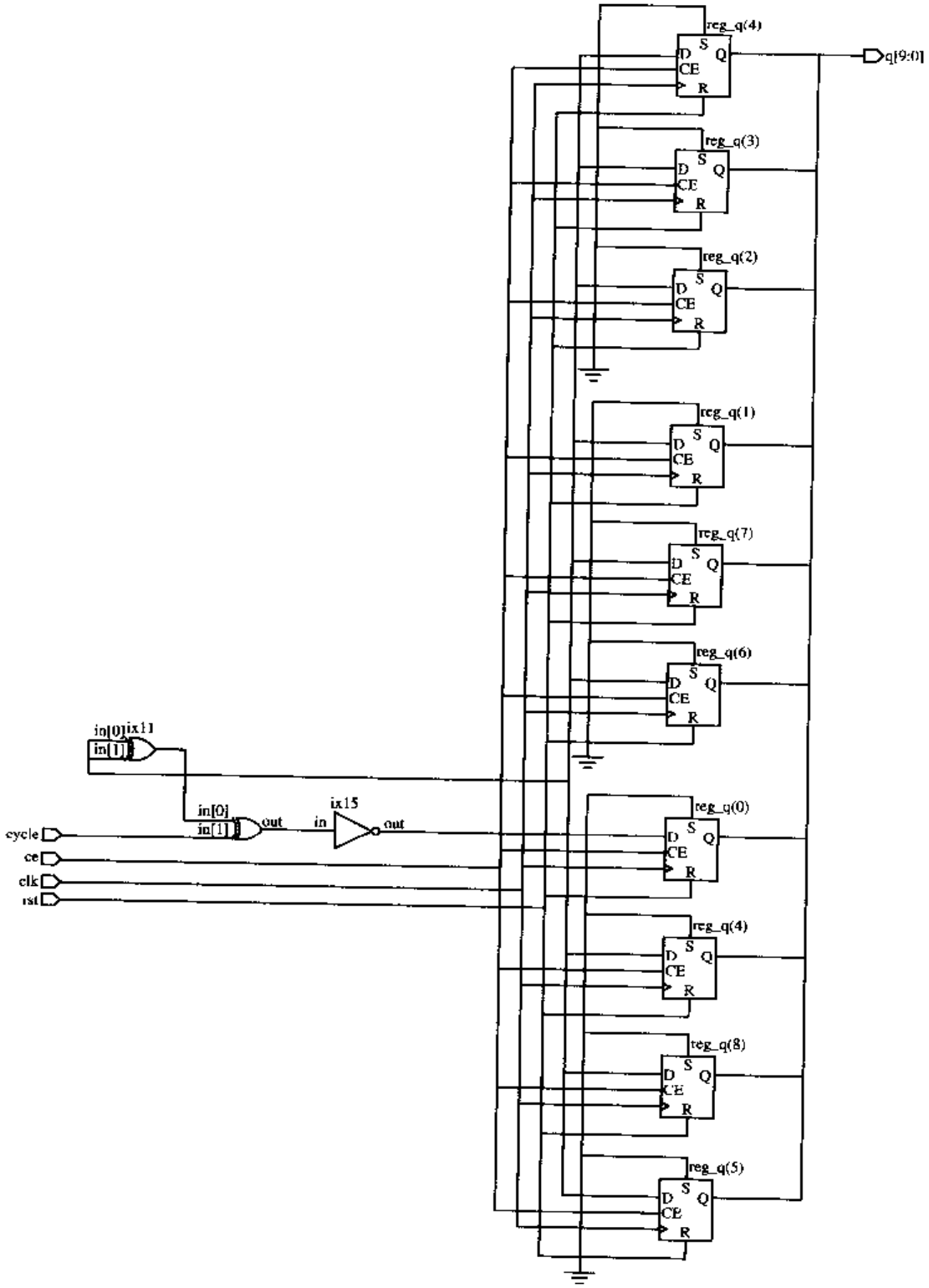


图 4.38　lfsr10 的综合结果

读者可以对该模块进行一下仿真。观察 h0、v0 等信号的产生周期。

补充：

利用线性反馈移位寄存器，反馈可以采用内异或和外异或两种，如图 4.39 所示。

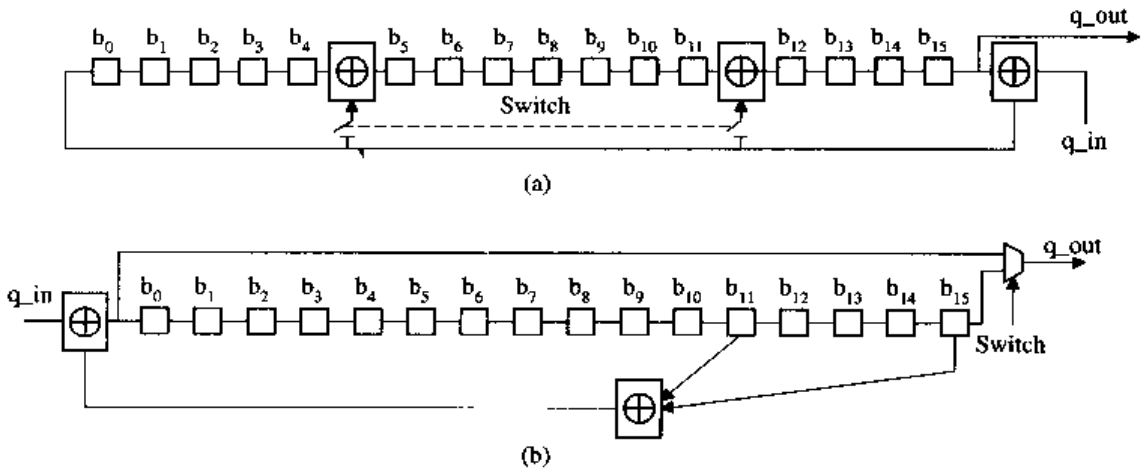


图 4.39　LFSR 的实现方式

(a) 内部 XOR LFSR；(b) 外部 XOR LFSR

4.4　桶形移位寄存器

桶形移位寄存器所占的面积较大，但如果某个应用需要大量的多比特位移，则采用桶形移位器还是有必要的。

1. 功能

8 位桶形移位寄存器的框图如图 4.40 所示。桶形移位寄存器的端口说明如表 4.11 所示。

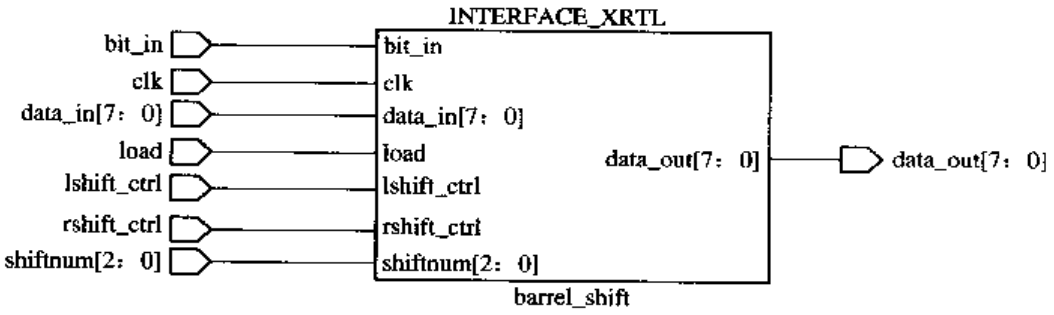


图 4.40　桶形移位寄存器框图

如果 load 有效，则输入 data_in 经过 DELAY 时间的延迟到输出 data_out，不执行任何操作。

如果右移位控制信号 rshift_ctrl 有效，则输出数据 data_out 执行右移位；

如果左移位控制信号 lshift_ctrl 为真，则输出数据 data_out 执行左移位；

移位的个数由 shiftnum 来决定，最多可一次移入 7 位比特；

整个设计为同步设计，一切操作均在时钟 clk 的上升沿执行。

表 4.11　桶形移位寄存器端口说明

端口	宽度	方向	说明
clk	1	输入	时钟
load	1	输入	同步载入
rshift_ctrl	1	输入	同步右移位控制
lshift_ctrl	1	输入	同步左移位控制
shiftnum	3	输入	要移位的比特数目
bit_in	1	输入	要移入空位置的比特
data_in	8	输入	输入字节
data_out	8	输出	输出字节

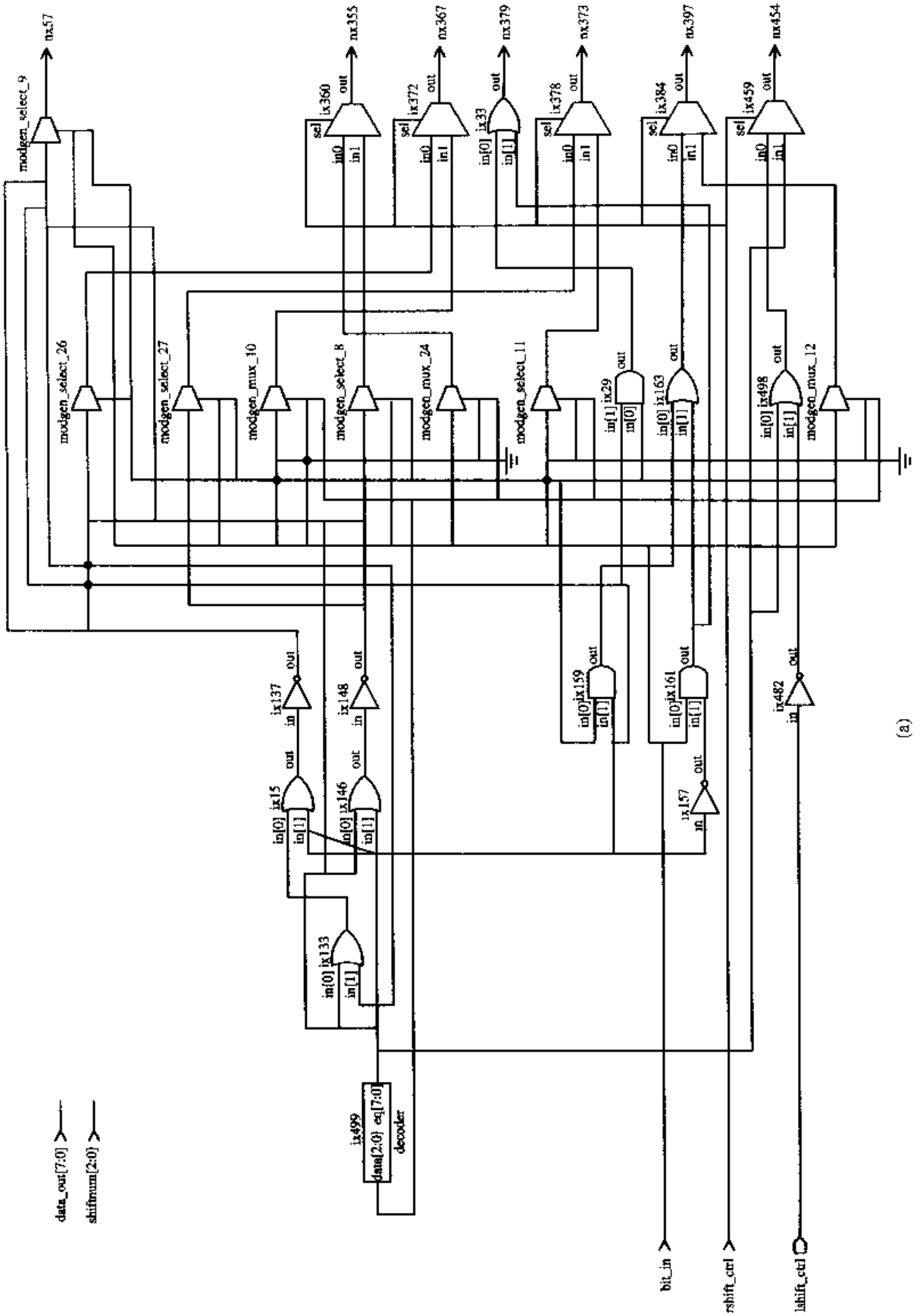
2. 代码

下面是桶形移位寄存器的一种实现代码。

```
/*********************************************************/
// MODULE:barrel_shift
//本模块实现了一个 8 位桶形移位寄存器
/*********************************************************/

// DEFINES
`define DELAY  1              //时钟到输出的延迟

//模块定义
module barrel_shift(clk,load,rshift_ctrl,lshift_ctrl,shiftnum,bit_in,data_in,data_out);

//端口声明
input          clk;           //时钟
input          load;          //同步载入
input          rshift_ctrl;   //同步右移位控制
input          lshift_ctrl;   //同步左移位控制
input [2:0]    shiftnum;      //要移位的比特数目
input          bit_in;        //要填充到空位置上的比特
input [7:0]    data_in;       //输入字节

output [7:0]data_out;         //输出字节

//信号声明
wire           clk;
wire           load;
wire           rshift_ctrl;
wire           lshift_ctrl;
```

```
wire [2:0]      shiftnum;
wire            bit_in;
wire [7:0]      data_in;
reg  [7:0]      data_out;

// MAIN CODE
always @(posedge clk) begin
    if (load) begin                 //在这里，load 有最高的优先级
        data_out <= # DELAY data_in;    //输入到输出有 delay 时间的延迟
    end
    else if (rshift_ctrl) begin
        //这里 rshift_ctrl 比 lshift_ctrl 优先
        case (shiftnum)         //要移位的比特数目不同，操作也不同
            3'h0: data_out <= # DELAY data_out;
            3'h1: data_out <= # DELAY {bit_in,data_out[7:1]};
            3'h2: data_out <= # DELAY {bit_in,bit_in,data_out[7:2]};
            3'h3: data_out <= # DELAY {bit_in,bit_in,bit_in,data_out[7:3]};
            3'h4: data_out <= # DELAY {bit_in,bit_in,bit_in,bit_in,data_out[7:4]};
            3'h5: data_out <= # DELAY {bit_in,bit_in,bit_in,bit_in,bit_in,data_out[7:5]};
            3'h6: data_out <= # DELAY {bit_in,bit_in,bit_in,bit_in,bit_in,bit_in,data_out[7:6]};
            3'h7: data_out <= # DELAY {bit_in,bit_in,bit_in,bit_in,  bit_in,bit_in,bit_in,
                                       data_out[7]};
        endcase
    end
    else if (lshift_ctrl) begin
        // lshift_ctrl 的优先级最低
        case (shiftnum)
            3'h0: data_out <= # DELAY data_out;
            3'h1: data_out <= # DELAY {data_out[6:0],bit_in};
            3'h2: data_out <= # DELAY {data_out[5:0],bit_in,bit_in};
            3'h3: data_out <= # DELAY {data_out[4:0].bit_in,bit_in,bit_in};
            3'h4: data_out <= # DELAY {data_out[3:0],bit_in,bit_in,bit_in,bit_in};
            3'h5: data_out <= # DELAY {data_out[2:0],bit_in,bit_in,bit_in,bit_in,bit_in};
            3'h6: data_out <= # DELAY {data_out[1:0],bit_in,bit_in,bit_in,bit_in,bit_in,bit_in},
            3'h7: data_out <= # DELAY {data_out[0],bit_in,bit_in,bit_in,bit_in,bit_in,bit_in,
                                   bit_in};
        endcase
    end
end
endmodule
```

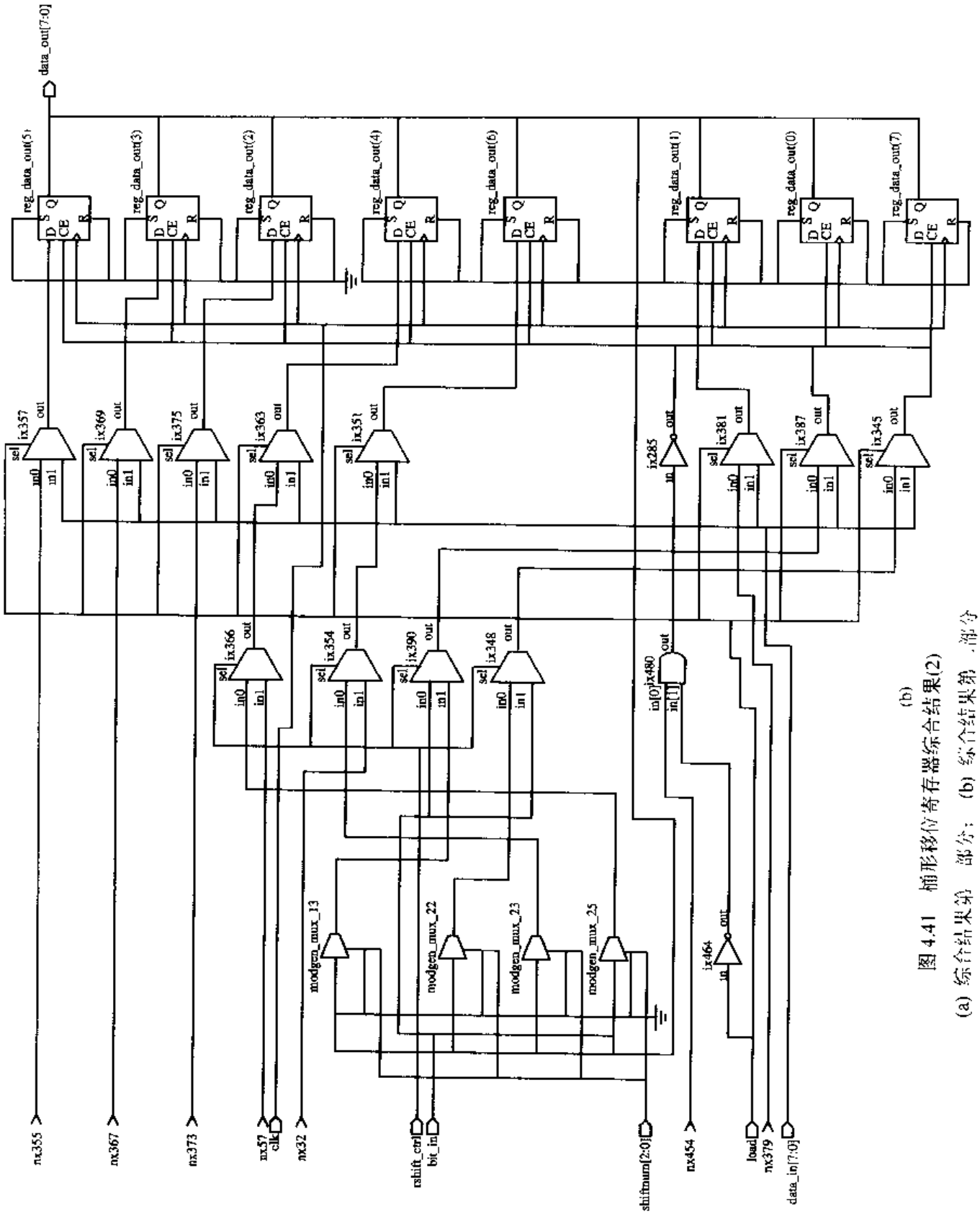
桶形移位寄存器综合结果如图 4.41 所示。



(a)

图 4.41 桶形移位寄存器综合结果(1)

(a) 综合结果第一部分； (b) 综合结果第二部分



(b)

图 4.41 桶形移位寄存器综合结果(2)

(a) 综合结果第一部分； (b) 综合结果第二部分

4.5 串/并转换模块

1. 功能

在电路设计中，经常要用到串/并转换模块。这里设计一个模块，它既能将并行数据转换成串行数据，也能将串行数据转成并行数据。串/并转换模块的端口说明如表 4.12 所示。

表 4.12 串/并转换模块端口说明

输入	端口说明
Clk	时钟信号
Reset	软复位控制信号
Send	发送数据使能信号，允许输出串行数据
Receive	接收数据使能信号，接收串行数据
Data_in[7:0]	接收的并行数据，需要转成串行数据发出
ser_in	接收的串行数据，需要转成并行数据送出
输出	端口说明
Data_out[7:0]	由输入的串行数据转换得到
ser_out	由输入的并行数据转换得到

2. 代码

串/并转换模块中，计数器的作用非常重要。在进行串/并转换时，每计够一个字节，它就要将移位寄存器的内容输出。在进行串/并转换时，可用它来判别输出位。

```
case (BitCnt)
        4'h0:
                SendBit <= SendDataLatch[0];
        4'h1:
                SendBit <= SendDataLatch[1];
        4'h2:
                SendBit <= SendDataLatch[2];
        4'h3:
                SendBit <= SendDataLatch[3];
        4'h4:
                SendBit <= SendDataLatch[4];
        4'h5:
```

```
                SendBit <= SendDataLatch[5];
        4'h6:
                SendBit <= SendDataLatch[6];
        4'h7:
                SendBit <= SendDataLatch[7];
```

在发送的时候，每个时钟周期 BitCnt 要加 1。计到 8 后又重新开始。

```
always @(posedge Clk or posedge Reset)
 begin
   if(Reset)
     BitCnt <= 4'h0;
   else if(BitCntEn==1 && Over==0) begin
        if(BitCnt==8)
            BitCnt <= 0;
        else
            BitCnt <= BitCnt + 1;
     end
 end
```

进行串/并转换时，其核心代码为：

```
always @(negedge Clk or posedge Reset)
  begin
    if(Reset)
        dataoutlatch <= 0;
    else begin
        case (BitCnt)
            4'h0: dataoutlatch[0] <= ser_in;
            4'h1: dataoutlatch[1] <= ser_in;
            4'h2: dataoutlatch[2] <= ser_in;
            4'h3: dataoutlatch[3] <= ser_in;
            4'h4: dataoutlatch[4] <= ser_in;
            4'h5: dataoutlatch[5] <= ser_in;
            4'h6: dataoutlatch[6] <= ser_in;
            4'h7: dataoutlatch[7] <= ser_in;
        endcase
    end
  end
```

4.6 加解密模块

为了保证产品的安全性，经常要用到加密。特别是在通信、金融、军事等领域。许多服务器、网络交换机、智能卡(SIM 卡，金融卡)都需要加解密芯片。加解密芯片的性能，跟所选择的算法联系很密切。

加密算法有很多种，适用于不同的应用场合。一般来说，加密分为单密钥体制和公用密钥体制两种。图 4.42 是一种单密钥体制的原理图。图 4.43 是一种公开密钥体制的原理图。

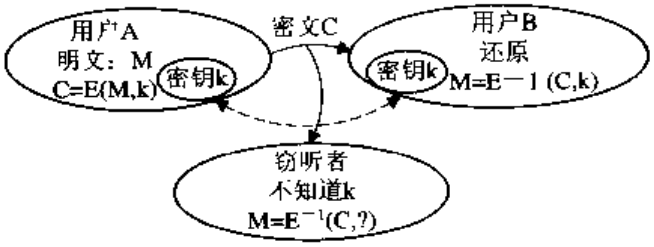


图 4.42 单密钥体制的原理图

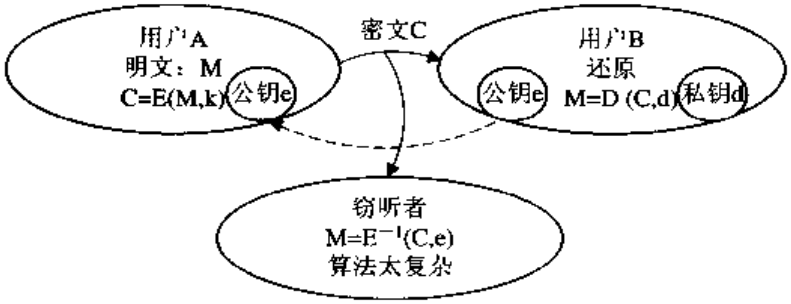


图 4.43 公开密钥体制的原理图

单密钥体制有如下特点：

算法简单，运算速度快，适用于中小系统和安全性要求不是很高的系统。

密钥数量众多。对于一个有着 N 个用户的系统，其所需密钥数为 N*(N-1)/2，交换和管理如此众多的密钥是一项非常巨大的任务，并且无法完成诸如数字签名、用户身份认证等附加服务。

公开密钥体制有如下特点：

所需密钥很少。对于一个有 N 个用户的系统，仅需 N 个公钥和 N 个私钥。公钥不仅可以用于数据加密，而且可以提供数字签名、身份认证等现代电子商务所需的基本技术，具有广泛的应用前景。

单密钥体制中，最著名的加密方法是 DES(数据加密标准)。随着解密技术的进步，DES 的安全性受到挑战。因此，又出现了 3-DES(3 重 DES)，它将密钥长度扩展到 192 位。最近美国又制定了 AES(高级加密标准)。DES/3-DES 与 AES 都是对称加密体制。

公开密钥加密技术是密码学上的一个里程碑。最著名的公开密钥加密技术是 RSA。近

年来，ECC(椭圆曲线加密)技术也受到广泛关注，有可能会成为一种普遍应用的技术。

下面首先介绍一个简单的加密模块，然后对 DES、RSA、ECC 等进行介绍。

4.6.1 简单加密模块

这里实现的加密模块非常简单：利用随机数直接对数据异或运算，完成加密。在实际应用中，加密算法要复杂许多。

1. 功能

要完成一个加密模块，可以对输入数据进行加密。数据的加密的原理如下：

输入数据与线性反馈移位寄存器得到的数据(伪随机数)进行异或操作。线性反馈移位寄存器的初值为输入的密钥。在解密模块中，线性反馈移位寄存器的值跟加密模块中线性反馈移位寄存器的值是一样的。用相同的密钥，解密模块可以生成相同的随机数，将此随机数与加密的数据异或，可将原数据还原出来。

该加密模块的框图如图 4.44 所示。加密模块的端口说明如表 4.13 所示。

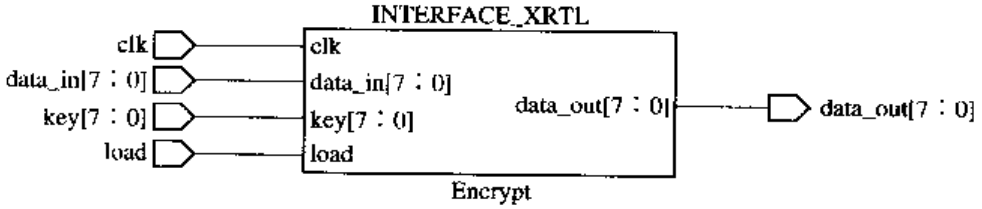


图 4.44 加密模块的框图

表 4.13 加密模块端口说明

端口	宽度	方向	说明
clk	1	输入	时钟
data_in	8	输入	输入数据
key	8	输入	加密的密钥
load	1	输入	指示载入密钥
data_out	8	输出	加密后的数据

2. 代码

在本模块中，载入密钥后，在每个时钟上升沿，将 temp 中的数据全部左移一位，最低位用^(temp & `ENC)填补，从而形成随机字节。

```
/*************************************************************/
// MODULE:encrypt
// 该模块实现了加密的功能
/*************************************************************/
`define DEL 1
```

```
`define ENC      8'b10001110
`define WIDTH 8                          //数据的位数

//模块描述
module     Encrypt(clk,load,key,data_in,data_out);

//端口描述
input              clk;                  //时钟
input              load;                 //载入密钥的控制信号
input [`WIDTH-1:0]    key;               //加密的密钥
input [`WIDTH-1:0]    data_in;           //输入数据

output [`WIDTH-1:0]   data_out;          //加密后的数据

//信号声明
wire               clk;
wire               load;
wire [`WIDTH-1:0]     key;
wire [`WIDTH-1:0]     data_in;
wire [`WIDTH-1:0]     data_out;

reg   [`WIDTH-1:0]    temp;              // temp data

//赋值语句
//对数据加密，将密钥与输入数据异或
assign #`DEL data_out = temp ^ data_in;

always @(posedge clk) begin
      if (load) begin          //当载入控制信号 load 为真时，将密钥载入到 temp 中
            temp <= #`DEL key;
      end
      else begin               //为加密而产生下一个随机字节
            temp[`WIDTH-1:1] <= #`DEL temp[`WIDTH-2:0]; //将 temp 中所有比特左移
            temp[0] <= #`DEL ^(temp & `ENC);     //产生最低位
      end
end

endmodule
```

该加密模块的综合结果如图 4.45 所示。

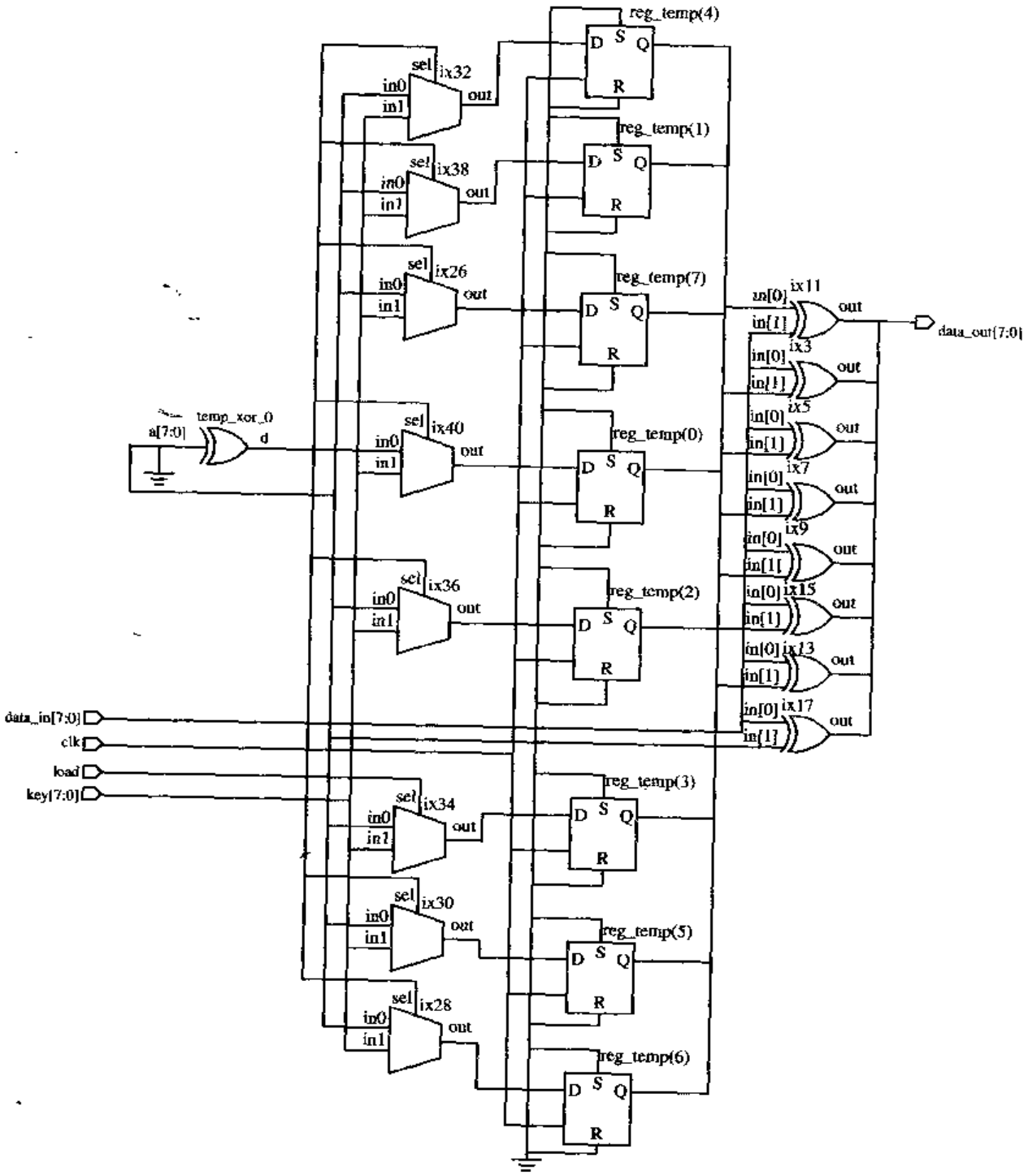


图 4.45　简单加密模块的综合结果

4.6.2　DES 加密

DES 算法把 64 位的明文输入块变换成 64 位的密文输出块。它所使用的密钥是 64 位的，其中 8 位是奇偶校验位。整个算法的过程如图 4.46 所示。

由图 4.46 可见，明文的处理经过了三个阶段。首先，64 位的明文经过一个初始置换后，比特重排，得到经过置换的输入。接下来，完成 16 次循环，称之为 16 轮(round)。轮函数中既包括置换功能，又包括替代功能。最后一轮的输出，送到逆置换矩阵中。这个置换是

初始置换的逆置换。

图 4.46 的右半部分是 56 位密钥的使用方式。密钥首先通过一个置换函数，接着，在 16 轮中的每一轮，通过一个循环左移操作和一个置换操作的组合，产生出一个子密钥 K_i。对每一轮来说，置换函数是相同的，但由于密钥比特的重复移位，产生的子密钥并不相同。

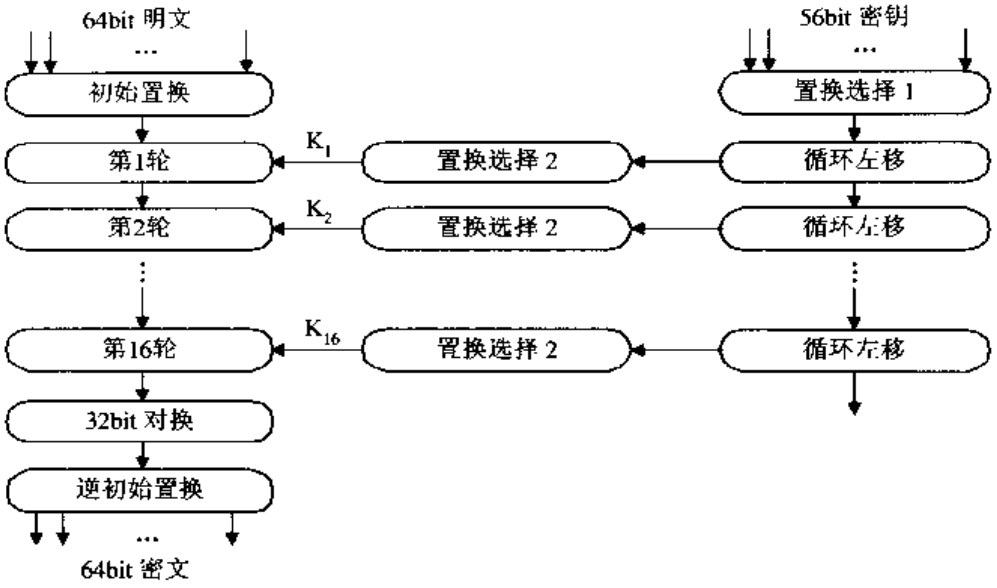


图 4.46 DES 算法的过程

图 4.47 给出了 DES 中某一轮的结构。将这两个图结合起来，可以清楚地了解 DES 加密的过程。

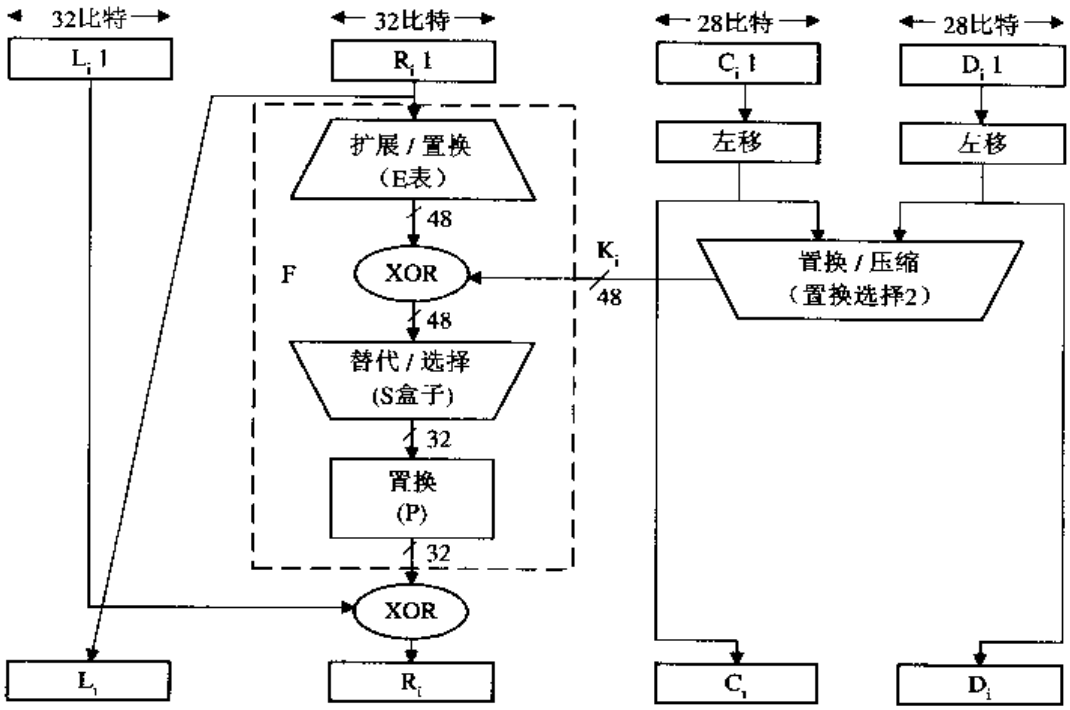


图 4.47 DES 算法的一轮

轮函数是 DES 加密的核心。它的功能如图 4.48 所示。

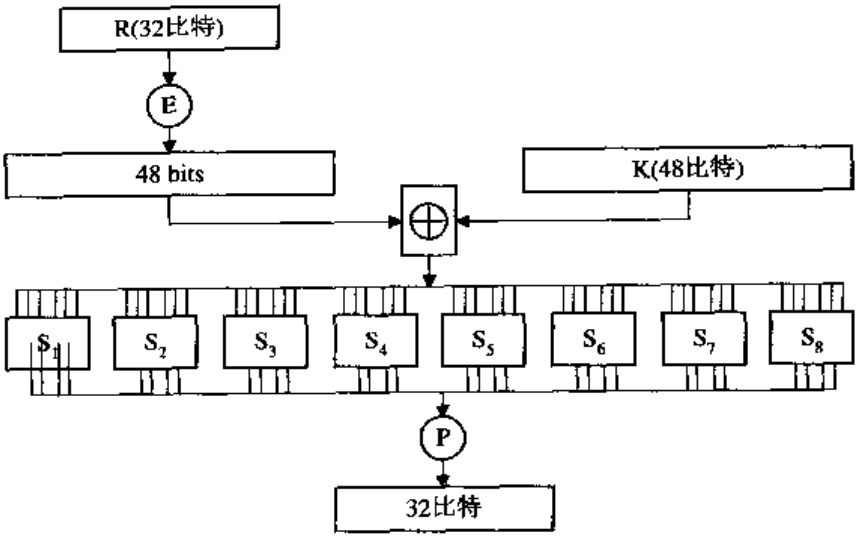


图 4.48　轮函数的结构

每个 S 盒接收 6 位的输入，产生 4 位的输出。这些变换在表 4.14 中给出。该表的使用方法如下：S_i 盒输入的第 1 位和最后 1 位构成一个二进制数，用来从 S_i 表中的 4 行中选择 1 行；而中间的 4 位则从 16 列中选择 1 列。例如，在 S_1 中，如果输入是 011001，则行是 01(第 1 行)，列是 1100(第 12 列)。而第 1 行第 12 列的数是 9，因此输出是 1001。一次替换由 8 个 S 盒完成，总共 32 位数据。

表 4.14　DES 的 S 盒子的定义

S 盒	数据															
S_1	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S_2	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	9
	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	15
	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	5
	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	10
S_3	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S_4	7	13	14	3	0	6	9	10	1	2	8	5	11	7	4	15
	13	8	11	5	6	15	0	3	4	7	2	12	1	2	14	9
	10	6	9	0	12	11	7	13	15	1	3	14	5	10	8	4
	3	15	0	6	10	1	13	8	9	4	5	11	12	12	2	14

续表

S 盒	数据															
S_5	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
S_6	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
S_7	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S_8	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

在 DES 算法中，轮密钥的生成是非常重要的。如图 4.47 所示，56 bit 密钥首先经过一个置换(置换方式见表 4.15)，得到 56 bit 密钥。这 56 bit 密钥分成 C_0 与 D_0 两个 28 bit 的部分。在每一轮中，C_i 与 D_i 都要经过左移与置换操作。

表 4.15 给出了 DES 中的置换表，包括初始置换、逆初始置换、扩展置换与置换函数。

表 4.15　DES 的置换表

(a) 初始置换(IP)								
行号	数据							
1	58	50	42	34	26	18	10	2
2	60	52	44	36	28	20	12	4
3	62	54	46	38	30	22	14	6
4	64	56	48	40	32	24	16	8
5	57	49	41	33	25	17	9	1
6	59	51	43	35	27	19	11	3
7	61	53	45	37	29	21	13	5
8	63	55	47	39	31	23	15	7
(b) 逆初始置换(IP^{-1})								
行号	数据							
1	40	8	48	16	56	24	64	32
2	39	7	47	15	55	23	63	31
3	38	6	46	14	54	22	62	30
4	37	5	45	13	53	21	61	29
5	36	4	44	12	52	20	60	28
6	35	3	43	11	51	19	59	27
7	34	2	42	10	50	18	58	26
8	33	1	41	9	49	17	57	25

(c) 扩展置换(E)

行号	数据					
1	32	1	2	3	4	5
2	4	5	6	7	8	9
3	8	9	10	11	12	13
4	12	13	14	15	16	17
5	16	17	18	19	20	21
6	20	21	22	23	24	25
7	24	25	26	27	28	29
8	28	29	30	31	32	1

(d)置换函数(P)

行号	数据							
1	16	7	20	21	29	12	28	17
2	1	15	23	26	5	18	31	10
3	2	8	24	14	32	27	3	9
4	19	13	30	6	22	11	4	25

下面给出一个 DES 设计。

1. 功能

DES 设计可实现 DES 加解密功能，DES 密钥长度为 56 位。DES 模块框图如图 4.49 所示。

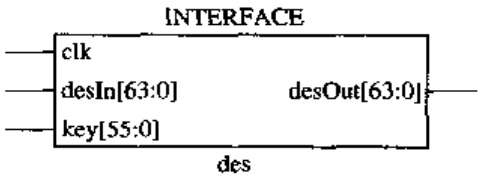


图 4.49 DES 模块框图

在 DES 中，最关键的部分是轮操作和轮密钥的生成。16 轮操作的结构是相同的，大大方便了硬件实现。在 DES 设计中，主要完成了明文的初始置换、16 轮操作、逆初始置换等功能。初始置换与逆置换都可以直接完成，关键是轮函数的实现。

2. 轮函数的实现

每一轮的处理过程可以表示为：

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$$

其中，F 函数包括扩展/置换(E 表)、与轮密钥的异或操作、S 盒操作、置换(P)操作等。下面实现扩展/置换和 S 盒的操作。

(1) 扩展/置换。

```
assign E[1:48] = {R[32], R[1], R[2], R[3], R[4], R[5], R[4], R[5],
        R[6], R[7], R[8], R[9], R[8], R[9], R[10], R[11],
        R[12], R[13], R[12], R[13], R[14], R[15], R[16],
        R[17], R[16], R[17], R[18], R[19], R[20], R[21],
        R[20], R[21], R[22], R[23], R[24], R[25], R[24],
        R[25], R[26], R[27], R[28], R[29], R[28], R[29],
        R[30], R[31], R[32], R[1]};
```

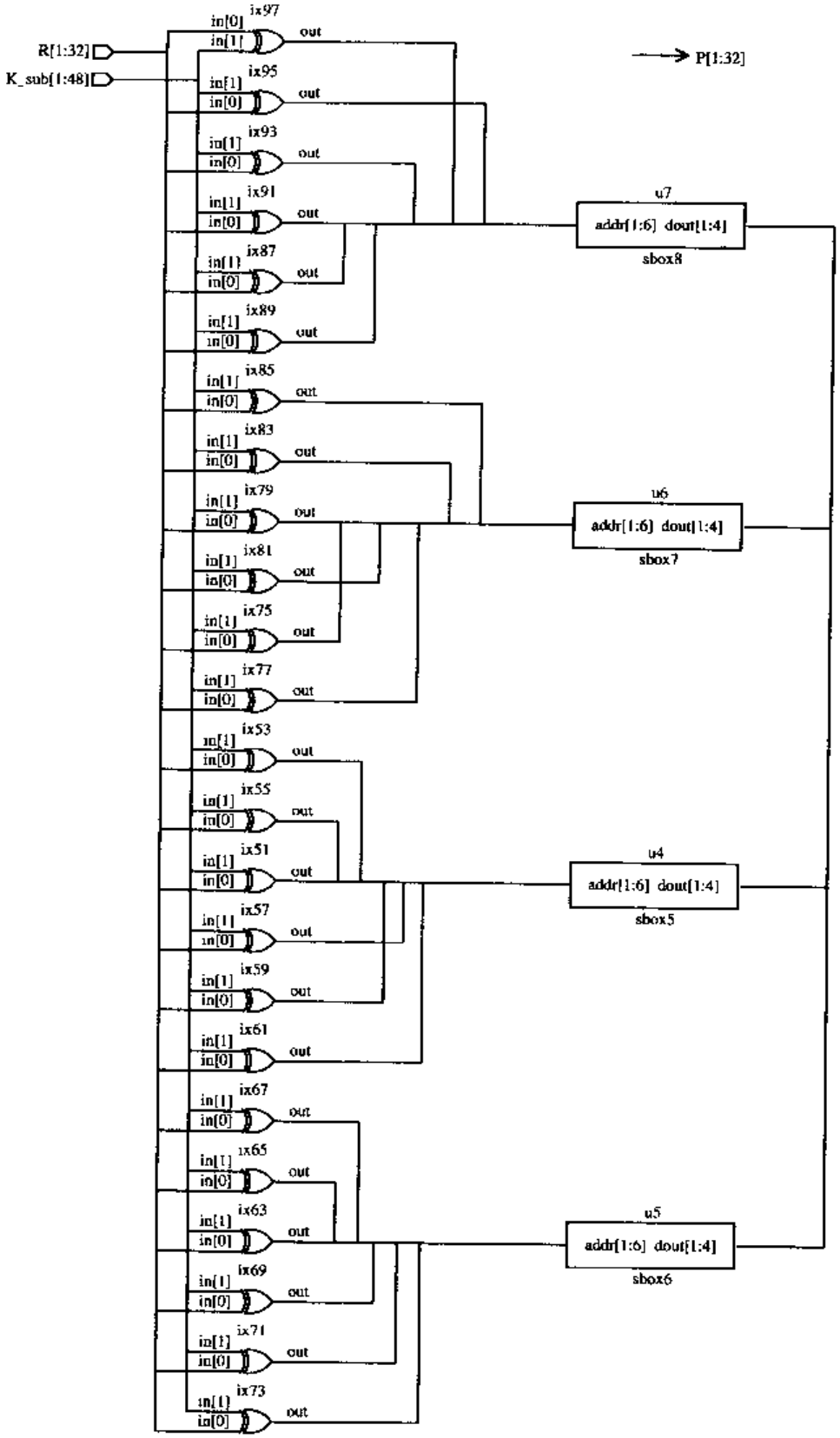
(2) S 盒的实现。

DES 中 S 盒的定义见表 4.14。这里给出第一个 S 盒的一种实现，其它 S 盒的实现与此类似。

```
module   sbox1(addr, dout);
input  [1:6] addr;
output [1:4] dout;
reg    [1:4] dout;

always @(addr) begin
  case ({addr[1], addr[6], addr[2:5]})     //synopsys full_case parallel_case
     0:  dout =  14;
     1:  dout =   4;
     2:  dout =  13;
     3:  dout =   1;
     4:  dout =   2;
     5:  dout =  15;
     6:  dout =  11;
     7:  dout =   8;
     8:  dout =   3;
     9:  dout =  10;
    10:  dout =   6;
    11:  dout =  12;
    12:  dout =   5;
    13:  dout =   9;
    14:  dout =   0;
    15:  dout =   7;

    16:  dout =   0;
    17:  dout =  15;
    18:  dout =   7;
    19:  dout =   4;
    20:  dout =  14;
    21:  dout =   2;
    22:  dout =  13;
    23:  dout =   1;
    24:  dout =  10;
    25:  dout =   6;
    26:  dout =  12;
    27:  dout =  11;
    28:  dout =   9;
    29:  dout =   5;
```

```
        30:   dout =   3;
        31:   dout =   8;

        32:   dout =   4;
        33:   dout =   1;
        34:   dout =  14;
        35:   dout =   8;
        36:   dout =  13;
        37:   dout =   6;
        38:   dout =   2;
        39:   dout =  11;
        40:   dout =  15;
        41:   dout =  12;
        42:   dout =   9;
        43:   dout =   7;
        44:   dout =   3;
        45:   dout =  10;
        46:   dout =   5;
        47:   dout =   0;

        48:   dout =  15;
        49:   dout =  12;
        50:   dout =   8;
        51:   dout =   2;
        52:   dout =   4;
        53:   dout =   9;
        54:   dout =   1;
        55:   dout =   7;
        56:   dout =   5;
        57:   dout =  11;
        58:   dout =   3;
        59:   dout =  14;
        60:   dout =  10;
        61:   dout =   0;
        62:   dout =   6;
        63:   dout =  13;
      endcase
        end
    endmodule
```

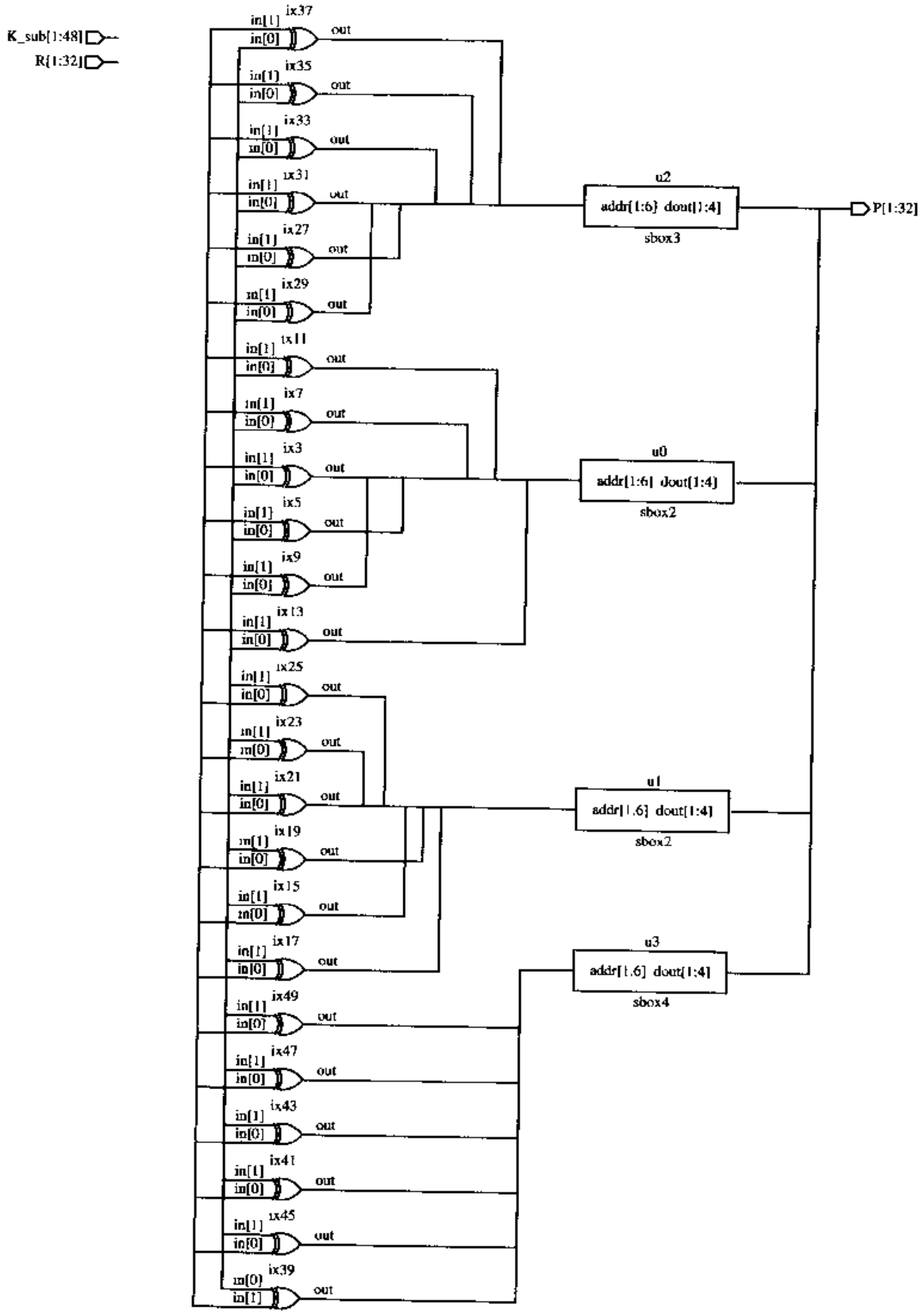
轮函数综合结果如图 4.50 所示。



(a)

图 4.50　轮函数综合结果(1)

(a) 轮函数综合结果第一部分；(b) 轮函数综合结果第二部分



(b)

图 4.50　轮函数综合结果(2)

(a) 轮函数综合结果第一部分；(b) 轮函数综合结果第二部分

4.6.3 其它加密

公开加密算法是研究的热点。公开加密都要利用一些数学难题。

1. RSA

- RSA 算法是由美国麻省理工学院的三位教授于 1978 年提出，并以他们姓氏的第一个字母命名的公开密钥算法。RSA 算法以数论为基础，它的保密性来源于大数分解的困难性。
- RSA 算法是目前国际标准的公开密钥算法。
- 在该算法中，d 表示私钥，e 表示公钥，N 表示模数，Ö(N)表示 N 的欧拉函数。M 表示加密信息(明文)的编码，C 表示加密后的信息(密文)的编码。
- 由于模数 N 的限制，要求所加密解密的信息 M、C 均小于 N，所以在需要加密的数据比较多的情况下就需要对明文进行分块，使分块后的每一个明块都小于模数，然后对每块数据单独运算即可。

该算法的运算过程如下：

(1) 选取模数 N 为两个大的素数 p 和 q 的乘积，Ö (N)为 N 的欧拉函数，有

N=p*q

Ö (N)=(p-1)*(q-1)

(2) 选取公钥 e，满足

1<e< Ö (N)

gcd(e, Ö (N))=1　　;表示 e 和 Ö (N)互素

(3) 求私钥 d，满足

e*d = 1mod (Ö (N))

(4) 加密过程：

C=M^e(mod N)

(5) 解密过程：

M=C^d(mod N)

在 RSA 中实现模运算时，最常用的算法是蒙哥马利算法。下面给出 Matlab 的实现。

```
function [x] = modmul(a,b,n);
    k = ceil(log2(n));
    r = 2^k;

    [d x y] = euclid(r,n);
    rinv = x;

    [d x y] = euclid(r*rinv, n);

    npri = -y;

    apri = mod(a*r, n);
```

```
    x = montpro(apri, b, npri, n, r);
    return;

function [u] = montpro(a, b. npri, n, r):
    t = a*b;
    m = mod(t*npri, r);
    u = (t + m*n)/r;

if u >= n
    u = u - n;
    return;
else
    return;
end
```

2. ECC

椭圆曲线加密与RSA相比，在同等的安全性下，密钥要短许多，处理开销较小，这方面的研究很早就开始了，但只是最近才开始商品化。

椭圆曲线并非椭圆，而是一种三次方程。这种方程在形式上类似于椭圆周长的方程。椭圆曲线三次方程的一般形式是：

$$y^2 + axy + by = x^3 + cx^2 + dx + e$$

在椭圆曲线上定义了一种点的加法。对于曲线上的一个点 $P_1(x1,y1)$和 $P_2(x2,y2)$，它们的连线与椭圆曲线有第三个交点 Q(x3,y3)，与 Q 点具有相同 x 坐标的曲线上的另一个点设为 M(x4,y4)，其中 x4=x3。在椭圆曲线中，定义：

$P_1+P_2=M$

ECC 中的加法操作对应于 RSA 中的取模乘法运算，而多次加法则对应于 RSA 中的取模指数操作。

4.7 信 源 编 码

数字系统中，数据通信无处不在。例如，手机、非接触式智能卡要通过载波接发数据，USB 设备、以太网卡要通过缆线与外界交互。数据在传输过程中，要进行编码。图 4.51 给出一般的通信模型。

在这个模型中，编码器的作用是，将数据转换成适合传输的信号，便于识别、纠错。

调制器的作用是将信号转换成适合传输的形式。调制方式大致可分为调频、调幅和调相三类。解调器的作用是将接收波形转换成数字信号序列。解码器的作用是将传输信号恢复成原始数据。

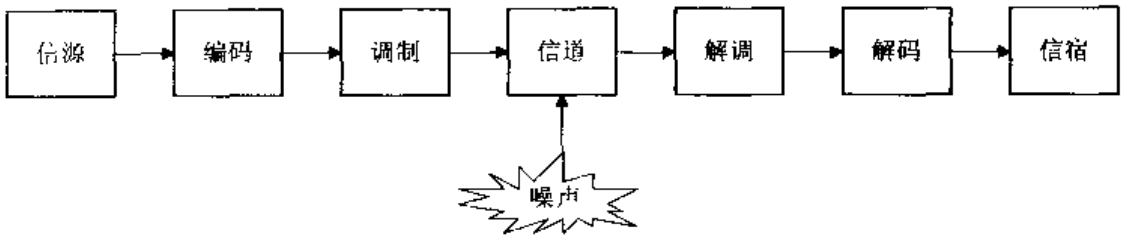


图 4.51　通信系统模型

例如，对于 TypeB 类型的非接触式 IC 卡，向读卡机发送数据时，采用的编码方式为不归零制(NRZ，Non-Return to Zero)，调制方式为二相绝对移相键控(BPSK)的方式。而读卡器的编码也采用 NRZ，调试方式采用 ASK(幅移键控)。

常见的数字信号编码方式有：

1) 不归零制(NRZ)

该方法中，二进制数字“0”、“1”分别用两种电平来表示。常用-5 V 表示“1”，+5 V 表示“0”。该方法的缺点是存在直流分量，传输中不能使用变压器，并且不具备自同步机制，必须使用外同步。

2) 不归零翻转(NRZI)

该方法中，当数据位为“1”时，不翻转；为“0”时，翻转。在 USB 中采用了 NRZI 编码方式。NRZI 编码如图 4.52 所示。

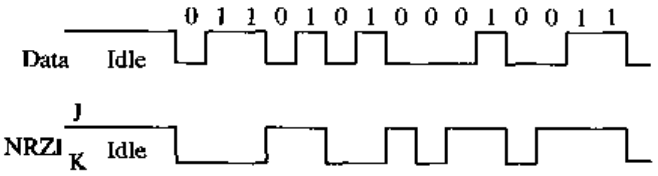


图 4.52　NRZI 编码

3) 曼彻斯特编码(Manchester Code)

该技术用电压的变化表示 0 和 1。规定在每个码元的中间发生跳变。高→低的跳变表示为 0，低→高的跳变表示为 1。每个码元中间都要发生跳变，接收端可将此变化提取出来作为同步信号，使接收端的时钟与发送设备的时钟保持一致。曼彻斯特编码的优点是具有自同步机制(它也称为自同步码)，无需外同步信号；缺点是需要双倍的传输带宽(即信号速率是数据速率的 2 倍)。

4) 差分曼彻斯特编码(Differential Manchester Code)

使用该方法，在每个码元的中间，信号都会发生跳变，这与曼彻斯特编码相同。不同之处在于，这里 0 与 1 的意义不同。如果在码元开始处有跳变，则为 0；否则为 1。

图 4.53 给出不归零制、曼彻斯特、差分曼彻斯特三种数字编码的波形图。

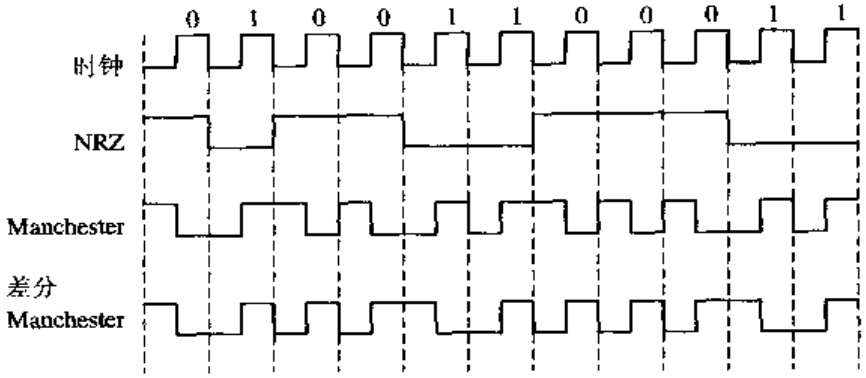


图 4.53　三种常见的信源编码方式

4.8　RAM 存 储 器

由于存储器都是模拟电路，所以要用全定制的方法来生成。在数字电路设计中，仿真时，经常要用到存储器的行为模型。这里我们给出一个 RAM(随机存取存储器)的行为模型。

4.8.1　RAM 的设计

1. 功能

该模块定义了一个 RAM，其数据总线是三态的。

当输出使能信号有效时，将存储器中相应地址内的数据输出，否则数据总线为高阻态。

当写选通信号有效时，将总线上的数据写入存储器的相应地址。

图 4.54 给出该模块的框图。RAM 的端口说明如表 4.16 所示。

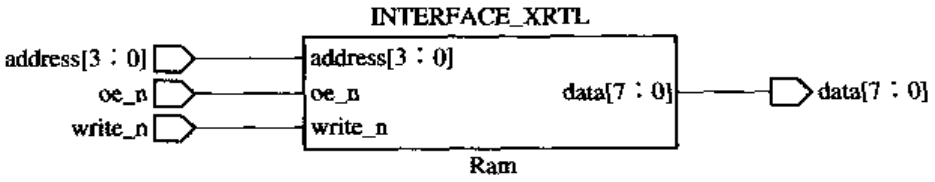


图 4.54　RAM 的行为模型

表 4.16　RAM 的端口说明

端口	宽度	方向	说明
address	4	输入	RAM 地址
oe_n	1	输入	输出使能信号(低有效)
write_n	1	输入	写选通信号(低有效)
data	8	双向输入/输出	RAM 数据

2. 实现代码

下面是 RAM 行为模型的一个实现代码。

```
/*****************************************************/
// MODULE:          RAM
/*****************************************************/

// DEFINES
`define DEL     1               //时钟到输出的延迟
                                //0 延迟有时会带来一些问题

`define RAM_WIDTH 8             // RAM 的宽度(比特的数目)
`define RAM_DEPTH 16            // RAM 的深度(字节的数目)
`define ADDR_SZ 4               //代表 RAM 地址所需要的比特数目

//模块描述
module Ram(data,address,write_n, oe_n);

// 端口描述
input [`ADDR_SZ-1:0] address;       // RAM 地址
input                write_n;       //写选通信号(低有效)
input                oe_n;          //输出使能(低有效)

inout [`RAM_WIDTH-1:0]  data;       // RAM 数据，双向

//信号声明
wire [`ADDR_SZ-1:0]     address;
wire                    write_n;
wire                    oe_n;
wire [`RAM_WIDTH-1:0]   data;

//存储体
reg   [`RAM_WIDTH-1:0]  mem [`RAM_DEPTH-1:0];

//赋值
//输出使能 oe_n 为 0 时，输出存储器相应地址 address 中的数据，否则为高阻态
assign #`DEL data = oe_n ? `RAM_WIDTH'bz : mem[address];

//在写选通信号 write_n 的上升沿，将数据写入存储器相应地址 address 中
always @(posedge write_n) begin
```

```
        mem[address] = data;
    end
    endmodule        // Ram
```

4.8.2 双端口 RAM

1. 功能

双端口 RAM 的行为模型与上面的 RAM 不同之处在于：
输入数据总线和输出数据总线是分开的，有时钟控制，而且是同步设计。
该模块有单独的读控制信号和读地址、写控制信号和写地址。
该模块的读写方式如下：
当写控制信号为真时，将输入数据写入存储器的写地址；
当读控制信号为真时，将数据从存储器的读地址中读出，并输出。
该模块的框图如图 4.55 所示。双端口 RAM 的端口说明如表 4.17 所示。

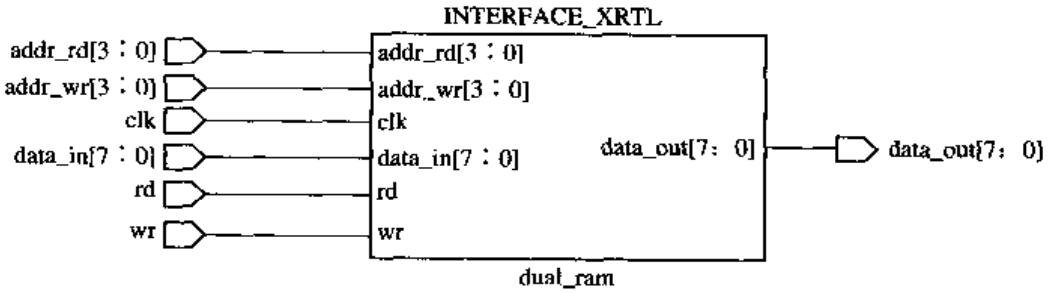


图 4.55　双端口 RAM 框图

表 4.17　双端口 RAM 的端口说明

端口	宽度	方向	说明
clk	1	输入	RAM 时钟
data_in	8	输入	RAM 输入数据
addr_rd	4	输入	RAM 读地址
rd	1	输入	读控制信号
addr_wr	4	输入	RAM 写地址
wr	1	输入	写控制信号
data_out	8	输出	RAM 输出数据

2. 代码

下面是双端口 RAM 行为模型的一个实现代码。

```
/*****************************************************/
// MODULE:dual_ram
//该模块实现了双端口 RAM
```

```
/*************************************************************/

//定义常量
`define DELAY   1           //时钟到输出的延迟
`define RAM_WIDTH 8         // RAM 的宽度
`define RAM_DEPTH 16        // RAM 的深度
`define ADDR_SIZE 4         //代表 RAM 地址所需要的比特数目

//模块描述
module dual_ram(clk,data_in,addr_rd,rd,data_out,addr_wr,wr);

//端口描述
input               clk;        // RAM 时钟
input [`RAM_WIDTH-1:0]  data_in;    // RAM 输入数据
input [`ADDR_SIZE-1:0]  addr_rd;    // RAM 的读地址
input               rd;         //读控制信号
input [`ADDR_SIZE-1:0]  addr_wr;    // RAM 的写地址
input               wr;         //写控制信号

output [`RAM_WIDTH-1:0] data_out;   // RAM 的输出数据

//信号声明
wire                clk;
wire [`RAM_WIDTH-1:0]   data_in;
wire [`ADDR_SIZE-1:0]   addr_rd;
wire                rd;
wire [`ADDR_SIZE-1:0]   addr_wr;
wire                wr;
reg  [`RAM_WIDTH-1:0]   data_out;

//存储体
reg [`RAM_WIDTH-1:0]    mem [`RAM_DEPTH-1:0];

//读写功能实现部分

//在时钟 clk 的上升沿
always @(posedge clk) begin
    if (wr)             //写控制信号为真时，将输入数据写入存储器的写地址
        mem[addr_wr] <= #DELAY data_in;
```

```
    if (rd)      //读控制信号为真时，将数据从存储器的读地址中读出，并输出
        data_out <= #DELAY mem[addr_rd];
end

endmodule      // dual_ram
```

4.9 DRAM 控制器

设计 DRAM(动态 RAM)控制器时，要了解 DRAM 的时序和工作原理。有关内容请参见第 5 章。DRAM 存在着大量、复杂的时序要求，其中访问时间的选择、等待状态以及刷新方法是至关重要的。DRAM 控制器必须正确响应所有总线周期，其访问速度必须足够快，以避免不必要的等待周期。

1. 功能

DRAM 控制器接收地址信号、读写控制信号和地址选择信号，然后产生相应的控制信号，控制 DRAM 工作。该模块的框图如图 4.56 所示。DRAM 控制器的端口说明如表 4.18 所示。

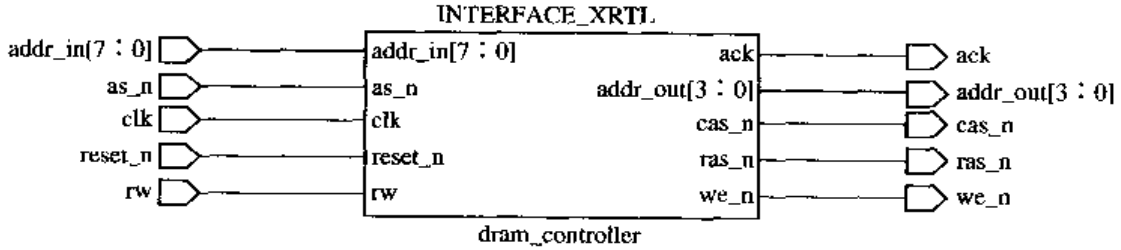


图 4.56 DRAM 控制器框图

表 4.18 DRAM 控制器端口说明

端口	宽度	方向	说明
addr_in	8	输入	从处理器来的地址
as_n	1	输入	地址选通信号，低有效
clk	1	输入	状态机时钟
reset_n	1	输入	同步复位信号，低有效
rw	1	输入	读/写输入，1—读，0—写
ack	1	输出	对处理器的确认信号
addr_out	4	输出	输出到 DRAM 的地址信号
we_n	1	输出	写使能信号
ras_n	1	输出	寄存器的行地址选通信号
cas_n	1	输出	寄存器的列地址选通信号

2. 代码

下面是 DRAM 控制器的一个实现代码。

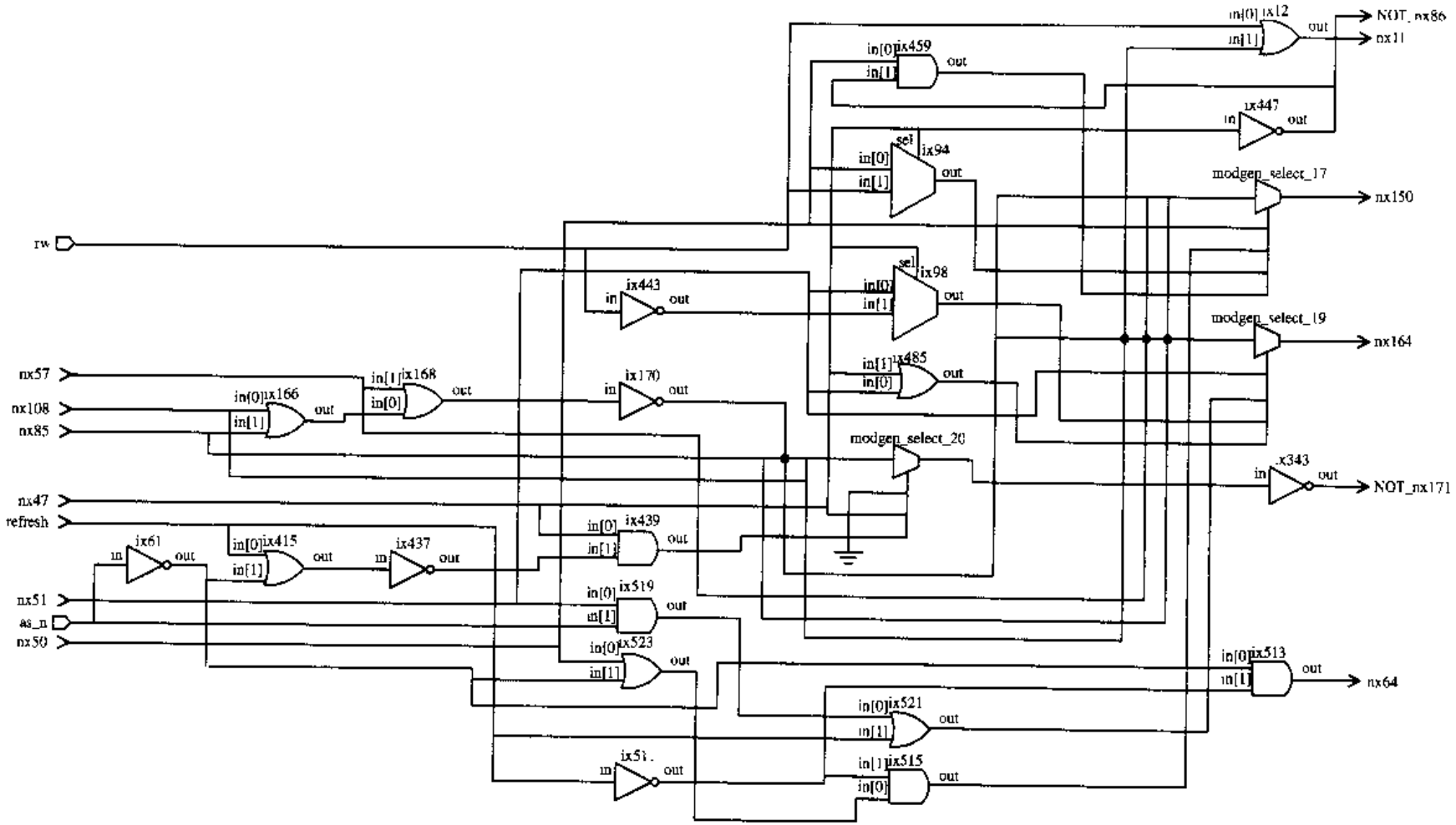
```
/*************************************************************/
// MODULE:dram_contrller
// DRAM 控制器
//该模块实现了 CAS-before-RAS 刷新
/*************************************************************/

//定义常量
`define DELAY  1      //时钟到输出的延迟
`define RAS_CAS 2     //在 CAS 有效前 RAS 有效的周期数目
`define CAS_RAS 1     //在 RAS 有效前 CAS 有效的周期数目
`define RC_WR 1       //对写操作，RAS 和 CAS 都有效的周期数目
`define RC_RD 2       //对读操作，RAS 和 CAS 都有效的周期数目
`define RC_REF 1      //对刷新操作，RAS 和 CAS 都有效的周期数目
`define CNT_BITS 2    //一个计算以上列出的周期数目的计数器所需要的比特数目
`define REF_CNT 24    //在刷新之间的周期数目
`define REF_BITS 5    //一个计算刷新所需周期数目的计数器所需要的比特数目
`define ADDR_OUT_WIDTH 4          // DRAM 的地址比特宽度
`define ADDR_IN_WIDTH 2*`ADDR_OUT_WIDTH     //从处理器来的比特宽度

//模块描述
module dram_controller(clk,reset_n,as_n,addr_in,addr_out,rw,we_n,ras_n,cas_n,ack);

//端口描述
input              clk;        //状态机时钟
input              reset_n;    //同步复位信号，低有效
input              as_n;       //地址选通信号，低有效
input [`ADDR_IN_WIDTH-1:0]   addr_in;    //从处理器来的地址
input              rw;         //读/写输入，1 为读，0 为写

output [`ADDR_OUT_WIDTH-1:0]     addr_out;  // DRAM 的地址
output                we_n;          //写使能输出
output                ras_n;         //对存储器的行地址选通
output                cas_n;         //对存储器的列地址选通
output                ack;           //对处理器的确认信号

//信号声明
```

```
wire                    clk;
wire                    reset_n;
wire [`ADDR_IN_WIDTH-1:0]       addr_in;
wire                    as_n;
wire                    rw;
wire                    we_n;
wire                    ras_n;
wire                    cas_n;
wire                    ack;
wire [`ADDR_OUT_WIDTH-1:0] addr_out;

reg   [3:0]             mem_state;      //可综合的状态机
wire                    col_out;        //此示输出地址是否为列地址，
                                        // 1 则输出为列地址，否则为行地址
reg   [`CNT_BITS-1:0]   count;          //计数器
reg    [`REF_BITS-1:0]  ref_count;      //刷表计数器
reg                     refresh;        // 刷新请求

//参数
parameter[3:0]                          //状态机的状态
IDLE= 4'b0000,
ACCESS   = 4'b0100,
SWITCH   = 4'b1100,
RAS_CAS = 4'b1110,
ACK = 4'b1111,
REF1= 4'b0010,
REF2= 4'b0110;

//赋值
//从不同的状态产生输出
assign col_out = mem_state[3];
assign ras_n = ~mem_state[2];
assign cas_n = ~mem_state[1];
assign ack = mem_state[0];

//在刷新时将 we_n 设为高，使之失效
assign #`DELAY we_n = rw | (mem_state == REF1) | (mem_state == REF2);
```

```
//将行地址或列地址给 DRAM，行地址在高位，列地址在低位
assign #`DELAY addr_out = col_out ? addr_in[`ADDR_OUT_WIDTH-1:0] :
                          addr_in[`ADDR_IN_WIDTH-1:`ADDR_OUT_WIDTH];

//状态变化，在时钟的上升沿
always @(posedge clk or negedge reset_n) begin
    if (~reset_n) begin
        mem_state <= #`DELAY IDLE;
        count <= #`DELAY `CNT_BITS'h0;
        ref_count <= #`DELAY `REF_CNT;
        refresh <= #`DELAY 1'b0;
    end
    else begin
        //用计数器计算刷新请求所需的时间(在刷新之间的周期数目)
        if (ref_count == 0) begin
            refresh <= #`DELAY 1'b1;
            ref_count <= #`DELAY `REF_CNT;
        end
        else
            ref_count <= #`DELAY ref_count - 1;

        //将周期计数器减 1，直至为 0
        if (count)
            count <= #`DELAY count - 1;

        case (mem_state)
            IDLE:    begin
                //刷新请求有最高的优先级
                if (refresh) begin
                    //载入计数器初值 CAS_RAS，使 CAS 有效
                    count <= #`DELAY `CAS_RAS;
                    mem_state <= #`DELAY REF1;   //开始刷新
                end
                else if (~as_n) begin
                    //载入计数器初值 RAS_CAS，使 RAS 有效
                    count <= #`DELAY `RAS_CAS;
                    mem_state <= #`DELAY ACCESS;
                end
            end
```

```
                ACCESS:  begin
                    mem_state <= #`DELAY SWITCH;
                end
                SWITCH:  begin
                    if (count == 0) begin
                        mem_state <= #`DELAY RAS_CAS;
                        if (rw)      //读操作
                            count <= #`DELAY `RC_RD;
                        else         //写操作
                            count <= #`DELAY `RC_WR;
                    end
                end
                RAS_CAS:begin
                    if (count == 0) begin
                        mem_state <= #`DELAY ACK;
                    end
                end
                ACK:     begin
                    mem_state <= #`DELAY IDLE;
                end
                REF1:    begin
                    if (count == 0) begin
                        mem_state <= #`DELAY REF2;
                        count <= #`DELAY `RC_REF;          //刷新操作
                    end
                end
                REF2:    begin
                    if (count == 0) begin
                        mem_state <= #`DELAY IDLE;
                        refresh <= #`DELAY 1'b0;           //刷新结束
                    end
                end
            endcase
        end
    end
    endmodule
```

DRAM 控制器的综合结果如图 4.57 所示。



(a)

图 4.57　DRAM 控制器综合结果(1)

(a) 综合结果第一部分；(b) 综合结果第二部分；(c) 综合结果第三部分

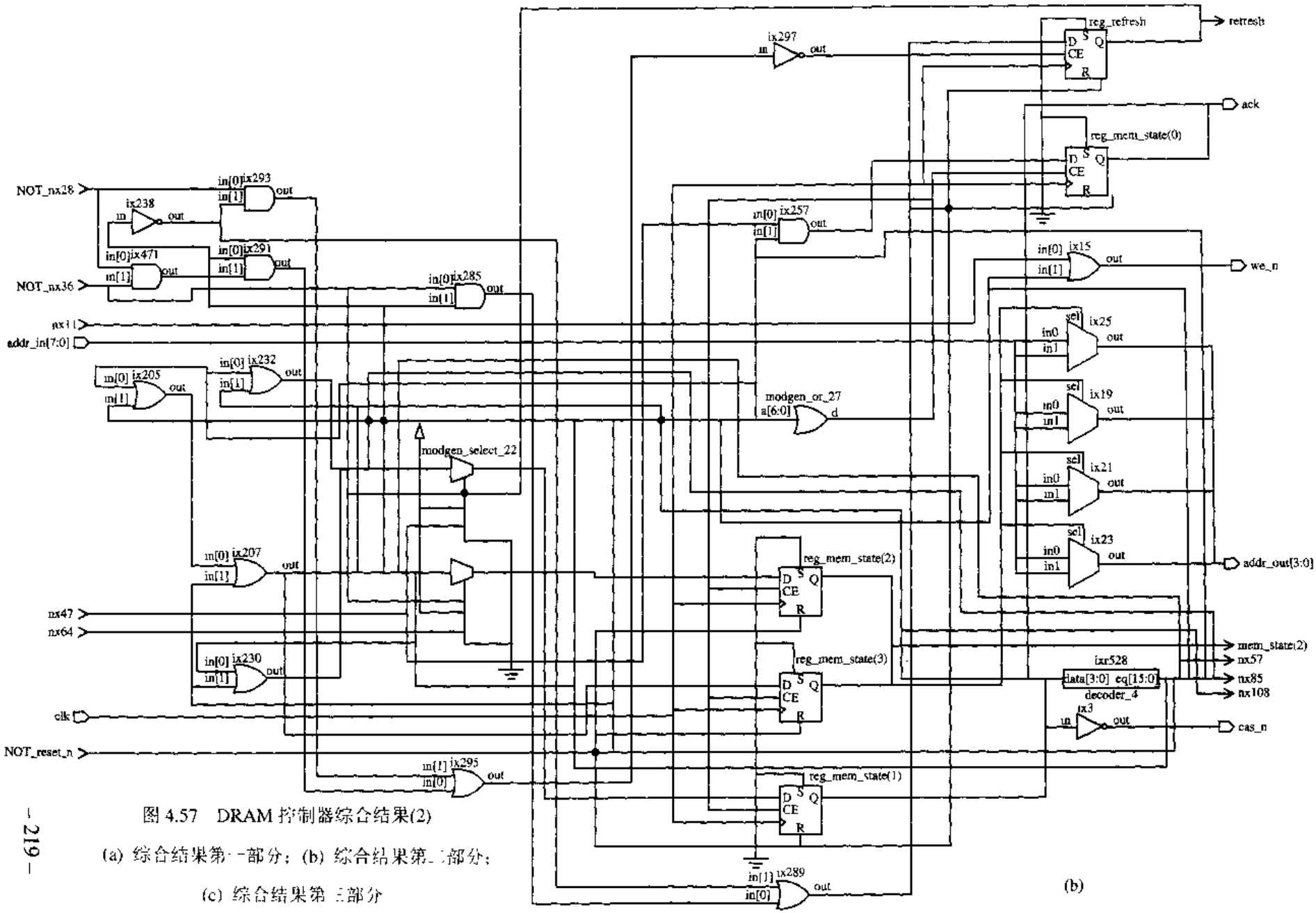
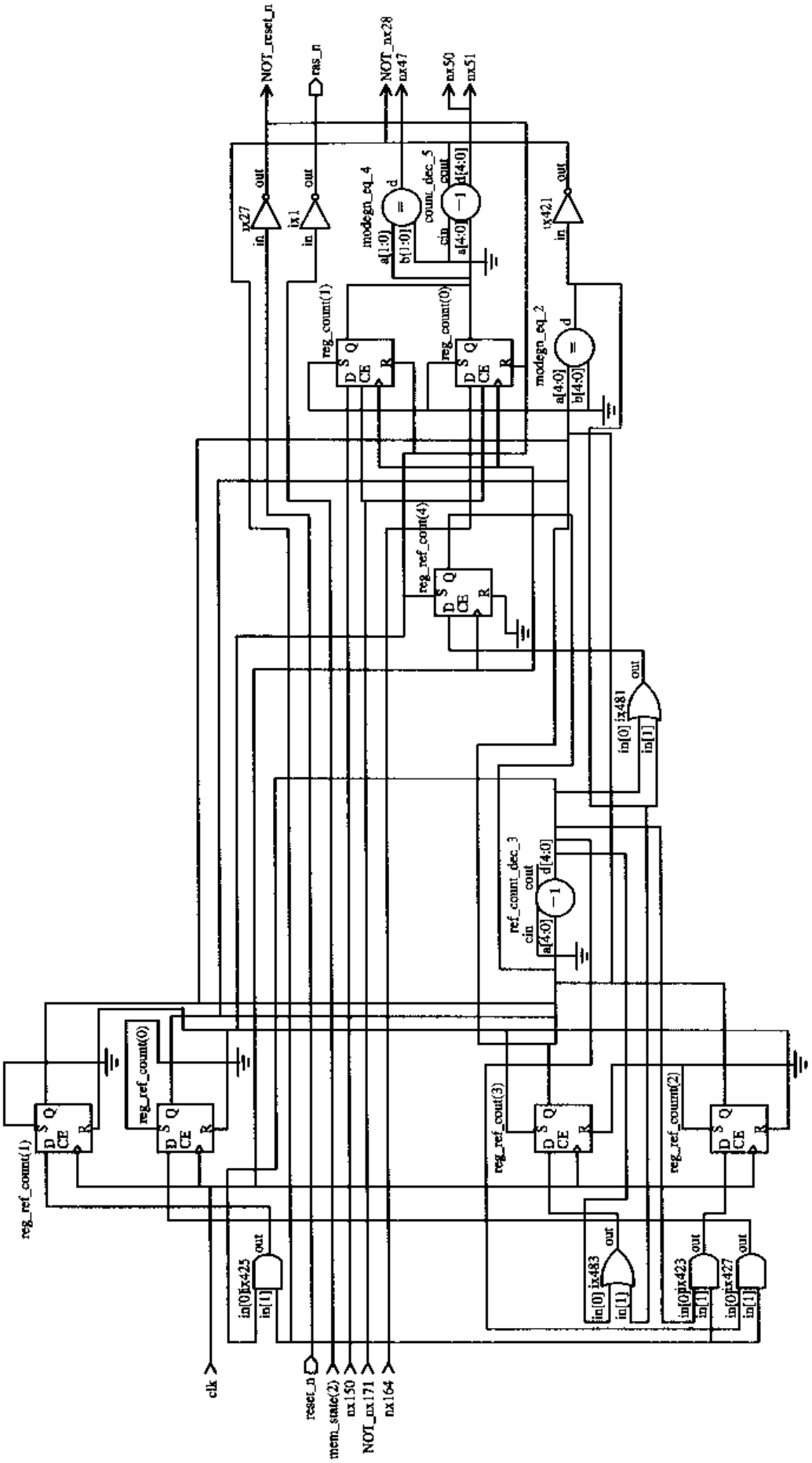


图 4.57　DRAM 控制器综合结果(2)

(a) 综合结果第一部分；(b) 综合结果第二部分；

(c) 综合结果第三部分



(c)

图 4.57 DRAM 控制器综合结果(3)

(a) 综合结果第一部分；(b) 综合结果第二部分；(c) 综合结果第三部分

4.10 SRAM 控制器

阅读本模块之前，读者需要了解 SRAM(静态 RAM)的相关时序。有关内容在第 5 章有描述。

1. 功能

本模块定义了一个 SRAM 控制器；

设计为同步设计，且有异步复位信号；

通过状态机完成对存储器的读写，定义了三个状态(IDLE、READ、WRITE)；

当地址选通后，查看读写命令，并执行相应操作，同时计数器开始计数；

模块定义了完成读和写所需要的周期数目(周期数目可以由使用者根据所用的SRAM和ROM 器件自行规定)；

当计数器计到周期数目时，认为已完成读写操作，停止计数，并回到初始值，等待下一次存取；

如果在读写的过程中，地址选通信号失效，则放弃存取；

如果地址选通执行写操作且已写完，或者执行读操作且已读完，这时，对处理器发出确认信号。

读存储器时，存储器输出使能信号有效；写存储器写时，存储器写使能信号有效。

该模块的框图如图 4.58 所示。SRAM 控制器的端口说明如表 4.19 所示。

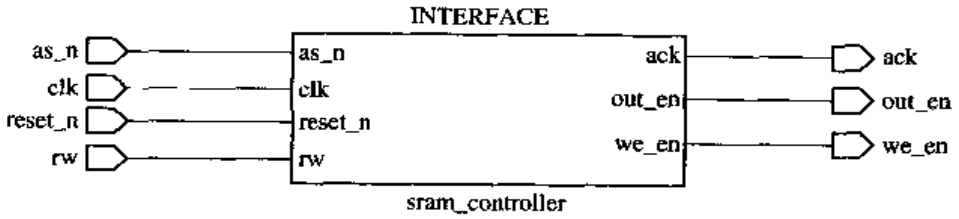


图 4.58 SRAM 控制器框图

表 4.19 SRAM 控制器端口说明

端口	宽度	方向	说明
as_n	1	输入	地址选通信号，低有效
clk	1	输入	状态机时钟
reset_n	1	输入	同步复位信号，低有效
rw	1	输入	读/写命令，1—读，0—写
ack	1	输出	对处理器的确认信号
we_en	1	输出	存储器的写使能信号
out_en	1	输出	存储器的输出使能信号

2. 代码

下面是 SRAM 控制器的一个实现代码。

```
/*************************************************************/
// MODULE:sram_controllerler
// SRAM 控制器
/*************************************************************/

//常量定义
`define DELAY  1      //时钟到输出的延迟
`define WR_CNT 1      //写周期所需要的比特数目
`define RD_CNT 4      //读周期所需要的比特数目
`define CNT_BITS 2    //计数器计算周期所需要的比特数目

//模块描述
module sram_controller(clk,reset_n,as_n,rw,out_en,we_en,ack);

//端口描述
input       clk;        //状态机时钟
input       reset_n;    //同步复位信号，低有效
input       as_n;       //地址选通信号，低有效
input       rw;         //读/写命令，1 为读，0 为写

output          out_en;         //存储器的输出使能
output          we_en;          //存储器的写使能
output          ack;            //对处理器的确认信号

//信号声明
wire        clk;
wire        reset_n;
wire        as_n;
wire        rw;
wire        out_en;
wire        we_en;
wire        ack;

reg   [1:0]       mem_state;          //可综合的状态机
reg   [`CNT_BITS-1:0]     cnt;        //周期计数器

//常数定义，定义状态机的状态
```

```
parameter[1:0]
    IDLE= 0,
    WRITE    = 1,
    READ     = 2;

//赋值,从状态产生输出
assign out_en = mem_state[1];                    //仅当状态为 READ 时为 1
assign we_en = mem_state[0];                     //仅当状态为 WRITE 时为 1

//组合逻辑产生确认信号
assign #DELAY ack = ~as_n && ((~rw && (cnt == `WR_CNT-1)) ||
                                ( rw && (cnt == `RD_CNT-1)));
//在地址选通 as_n=0，并且，
//写 rw=0 且已写完 cnt==`WR_CNT-1，或者读 rw=1 且已读完 cnt==`RD_CNT-1
//此时，对处理器发确认信号

//状态变化，在时钟上升沿
always @(posedge clk or negedge reset_n) begin
    if (~reset_n) begin
        mem_state <= #DELAY IDLE;
        cnt <= #DELAY `CNT_BITS'h0;
    end
    else begin
        case (mem_state)
            IDLE:    begin
                //由地址选通信号控制，开始存取
                if (~as_n) begin          //as_n=0 时，有效
                    if (rw) begin
                        //读，rw=1
                        mem_state <= #DELAY READ;
                    end
                    else begin
                        //写，rw=0
                        mem_state <= #DELAY WRITE;
                    end
                end
            end
            WRITE:   begin
                //如果计数器计数已经到了写的最后，则写结束
```

```
                    //如果地址选通被置 1，即无效，则放弃存取
                    if ((cnt == `WR_CNT-1) || as_n) begin
                    mem_state <= #DELAY IDLE;   //写结束或放弃时，状态回到 IDLE
                        cnt <= #DELAY `CNT_BITS'h0; //且计数回到初始值
                    end
                    else
                        cnt <= #DELAY cnt + 1;          //否则，计数加 1
                end
                READ:   begin
                    //如果计数器计数已经到了读的最后，则读结束
                    //如果地址选通被置 1，即无效，则放弃存取
                    if ((cnt == `RD_CNT-1) || as_n) begin
                    mem_state <= #DELAY IDLE;   //读结束或放弃时，状态回到 IDLE
                        cnt <= #DELAY `CNT_BITS'h0; //且计数回到初始值
                    end
                    else
                        cnt <= #DELAY cnt + 1;          //否则，计数加 1
                end
            endcase
        end
    end
    endmodule
```

SRAM 控制器的综合结果如书末附图所示。

4.11 异 步 FIFO

FIFO 类似于一个环形寄存器。在第 1 章中，以一个同步 FIFO 为例说明了设计流程。因而这里只给出异步 FIFO 的设计。

1. 功能

异步 FIFO 的功能基本与同步 FIFO 相同。不同之处在于，没有时钟，数据的输入输出直接由读/写信号的电平控制。图 4.59 是异步 FIFO 的框图。异步 FIFO 的端口说明如表 4.20 所示。

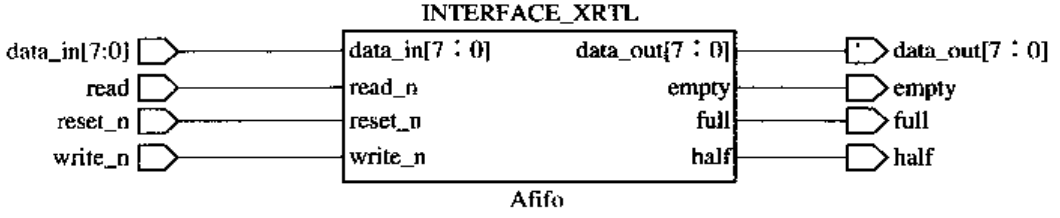


图 4.59 异步 FIFO 的框图

表 4.20　异步 FIFO 的端口说明

端口	宽度	方向	说明
data_in	8	输入	FIFO 的输入数据
read_n	1	输入	读 FIFO(低有效)
reset_n	1	输入	低有效复位信号
write_n	1	输入	写 FIFO(低有效)
data_out	8	输出	FIFO 的输出数据
empty	1	输出	FIFO 全空信号
full	1	输出	FIFO 全满信号
half	1	输出	FIFO 半满或大于半满的指示信号

2. 代码

下面是异步 FIFO 的一个实现代码。

```
/*************************************************************/
// MODULE:     同步 FIFO
//读指针指向缓冲的开始，而写指针指向缓冲的结束位置
/*************************************************************/

//定义常量
`define DEL      1              //时钟到输出的延时
`define FIFO_DEPTH 15           // FIFO 的深度
`define FIFO_HALF 8             // FIFO 深度的一半
`define FIFO_BITS 4             //要表示 FIFO 深度需要的位数
`define FIFO_WIDTH 8            // FIFO 的宽度

//模块描述
module Afifo(reset_n,data_in,read_n,write_n,data_out,    full,empty,half);

//端口描述
input                          reset_n;    //复位信号，低电平有效
input [`FIFO_WIDTH-1:0]        data_in;    //输入数据
input                          read_n;     //读使能信号，低电平有效
input                          write_n;    //写使能信号，低电平有效

output [`FIFO_WIDTH-1:0] data_out;                 // FIFO 输出数据
output                         full;       //满标志
output                         empty;      //空标志
output                         half;       //半满标志

//信号声明
wire                           reset_n;
wire [`FIFO_WIDTH-1:0]         data_in;
```

```
wire                        read_n;
wire                        write_n;
reg   [`FIFO_WIDTH-1:0]     data_out;
wire                        full;
wire                        empty;
wire                        half;

// FIFO 的存储体
reg    [`FIFO_WIDTH-1:0]        fifo_mem[0:`FIFO_DEPTH];
//用计数器来记录 FIFO 中已填充了几个单元
wire [`FIFO_BITS-1:0]counter;
// FIFO 读指针，指向下一次读的位置
reg    [`FIFO_BITS-1:0]    rd_pointer;
// FIFO 写指针，指向下一次写操作的地址
reg    [`FIFO_BITS-1:0]    wr_pointer;

//赋值
assign #`DEL counter = (wr_pointer >= rd_pointer) ?
                        wr_pointer - rd_pointer:
                        `FIFO_DEPTH + wr_pointer - rd_pointer + 1;
assign #`DEL full = (counter == `FIFO_DEPTH) ? 1'b1 : 1'b0;
assign #`DEL empty = (counter == 0) ? 1'b1 : 1'b0;
assign #`DEL half = (counter >= `FIFO_HALF) ? 1'b1 : 1'b0;

//功能实现部分
always @(negedge reset_n or negedge read_n) begin
    if (~reset_n) begin
        //复位读指针
        rd_pointer <= #`DEL `FIFO_BITS'b0;
    end
    else begin
        //读指针递增
        //检查读指针是否已达到边界。如果达到，则读指针指向 FIFO 的起始位置
        //注意使用>=而不是==，因为这会简化逻辑
        if (rd_pointer >= `FIFO_DEPTH)
            rd_pointer <= #`DEL `FIFO_BITS'b0;
        else
            rd_pointer <= #`DEL rd_pointer + 1;
    end
end

always @(negedge read_n) begin
    //检查是否出错
```

```
        if (counter == 0) begin
            $display("\nERROR at time %0t:", $time);
            $display("FIFO Underflow\n");

            //停止
            $stop;
        end

        //输出数据
        data_out <= #DEL fifo_mem[rd_pointer];
    end
    //写操作的实现
    always @(negedge reset_n or negedge write_n) begin
        if (~reset_n) begin
            // Reset the write pointer
            wr_pointer <= #DEL `FIFO_BITS'b0;
        end
        else begin
            //递增写指针，检查写指针是否达到边界。如达到，使它回到 FIFO 的起始位置
            if (wr_pointer >= `FIFO_DEPTH)
                wr_pointer <= #DEL `FIFO_BITS'b0;
            else
                wr_pointer <= #DEL wr_pointer + 1;

        end
    end

    always @(negedge write_n) begin
        //检查是否出错
        if (counter >= `FIFO_DEPTH) begin
            $display("\nERROR at time %0t:", $time);
            $display("FIFO Overflow\n");

            //如果出错，退出
            $stop;
        end

        //将数据写入 FIFO 中
        fifo_mem[wr_pointer] <= #DEL data_in;
    end
    endmodule
```

异步 FIFO 的综合结果如图 4.60 所示。

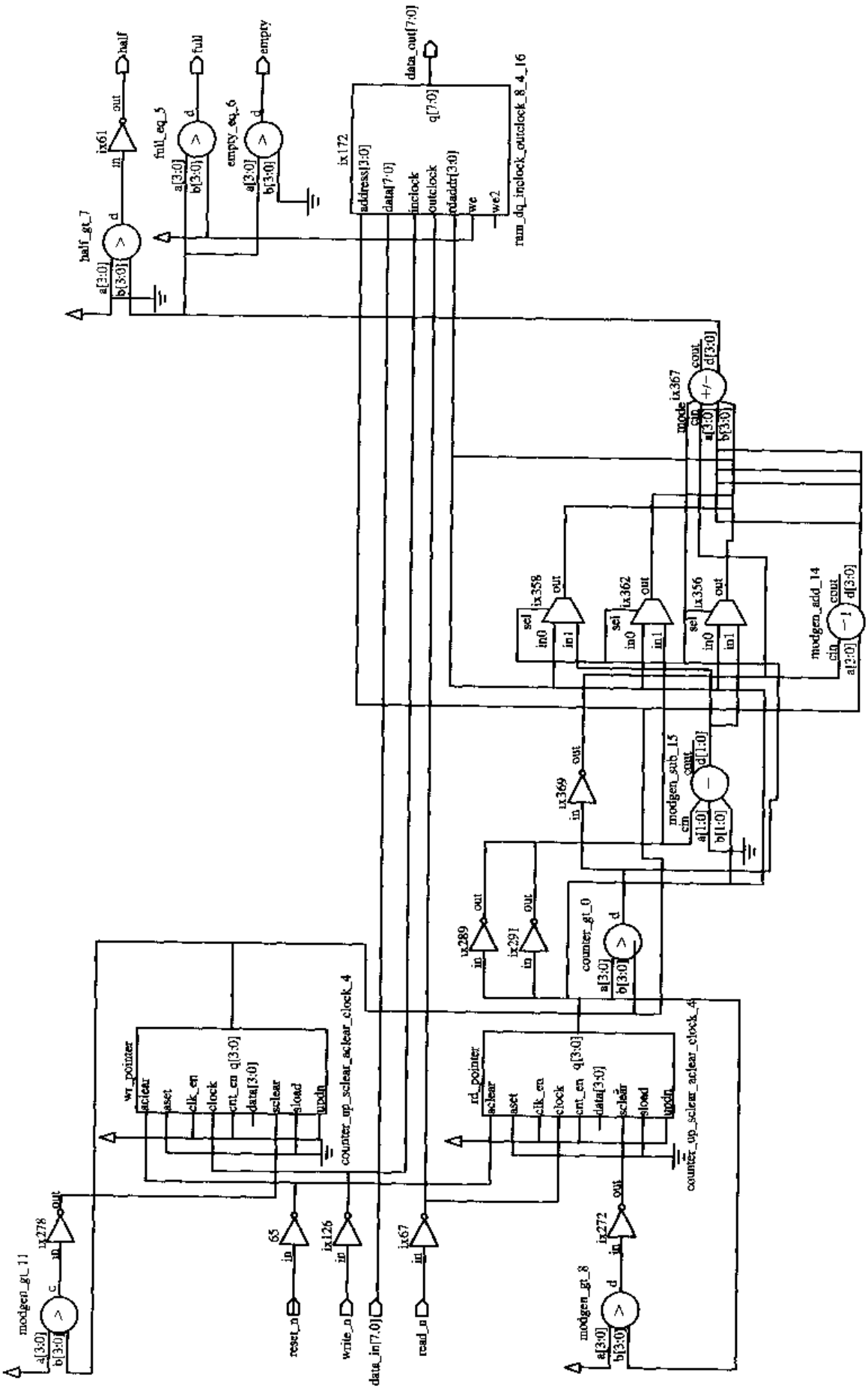


图 4.60 异步 FIFO 的综合结果

4.12 数字锁相环

在第 2 章中已介绍过数字锁相环 PLL，在此仅介绍其中的 DLL(Delayed Lock Loop)。这种锁相环是数字的，常用作数字通信中的时钟同步。

在数据传输系统中，为了能正确地接收和处理外部信号，要求外部信号与系统的内部时钟在频率和相位上都要相同。原因如下：

在数据通信过程中，发送端依相互约定的速率逐个码元发送数据，接收端则以相同速率来接收每一码元。这就需要对码元进行取样判决，如图 4.61 所示。

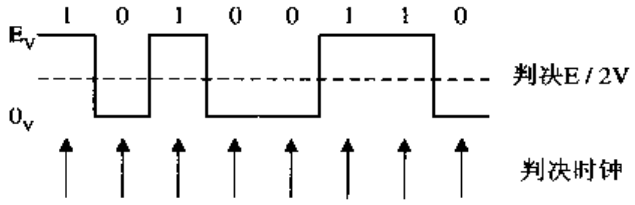


图 4.61 接收数据的判决

判决电平位于波形高电平和低电平的中间，而取样判决的最佳时刻应在码元的正中间。若大于判决电平，则认为接收到“1”，而小于判决电平，则认为接收到“0”。而判决若在码元的边缘，码元在传输中不可避免地会失真，则可能出现判决错误。因此保持时钟同步非常重要。

在通信中，各种数据在采集、传输、分组、交换、判决提取，以及压缩编解码等一系列过程中，不可避免地存在各类信号的延迟、失真、相对关系变化等问题。这就需要用同步技术来加以控制。通信系统常用如下几种方式进行同步。

1) 使用统一时钟

这是指双方在数据信道之外，一般还要提供一条时钟的信道。信道内共有两根传输信号线，一根传输信号，另一根传输外部时钟。这种方法一般用在一个芯片内部，如图 4.62 所示。

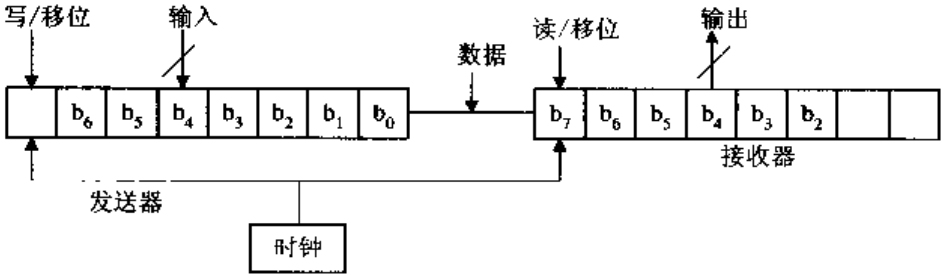


图 4.62 统一时钟同步方式

采用这种方式，不管是否传输数据，收发两端的时钟在每个比特上都保持同步。

2) 利用独立的同步信号

这是指设法在发送的数据信号中插入同步时钟信号，以供接收方采取技术措施来提取，如图 4.63 所示。

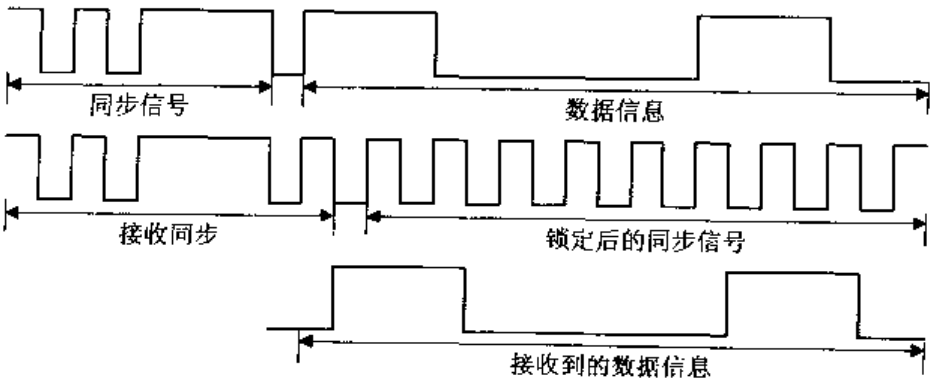


图 4.63　外同步方式

在这种方式中，接收方没有独立的时钟，由数字锁相环把时钟从信号中提取出来。多个芯片构成的系统进行通信，以及无线通信时，经常不可避免地要采用这种方法。图 4.64 给出数字锁相环进行同步的原理图。

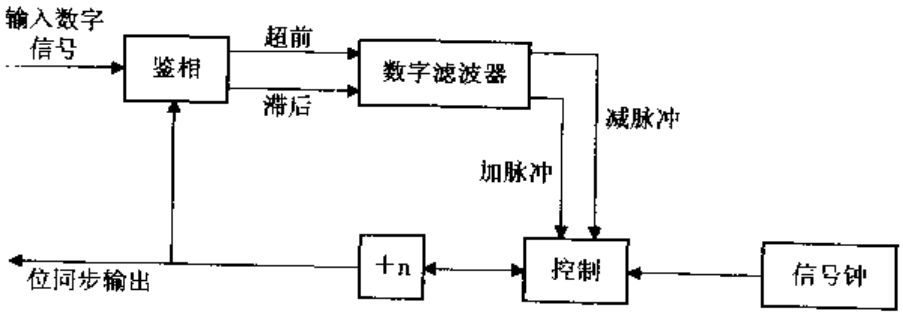


图 4.64　数字锁相的同步提取

在数字锁相环中，主要包括如下几个部分：

(1) 信号钟。它包括一个高稳定度的振荡器(晶振)和整形电路，若输入信号码元速率 $B=1/T$，那么振荡器频率 $f_0=n/T=nB$。经整形电路之后，输出周期性序列，其周期 $T_0=1/f_0=T/n$。

(2) 控制器与分频器。控制器与分频器共同作用。如果发现时钟超前，则减去几个时钟脉冲；如果发现时钟滞后，则增加几个时钟脉冲。这样，就调整了时钟步，保证了相位的同步。

(3) 鉴相器。它将输入信号码与位同步信号进行相位比较，判别位同步信号究竟是超前还是滞后，若超前，就输出超前脉冲；若滞后，就输出滞后脉冲。判别位同步信号是超前还是滞后的鉴相器有两种方法，即微分型和积分型。

(4) 数字滤波器。数字滤波器的作用是滤除噪声对环路工作的影响，提高相位校正的准确性。

下面构造一个简单的数字锁相环。

4.12.1　简单的数字锁相环

1. 功能

数字锁相环的功能：

带有同步复位信号；

仅当输入时钟上升沿时工作；

利用计数器完成对输入时钟的分频，通过输入控制信号来定义输出时钟的频率。简单数字锁相环的框图如图 4.65 所示。数字锁相环的端口说明如表 4.21 所示。

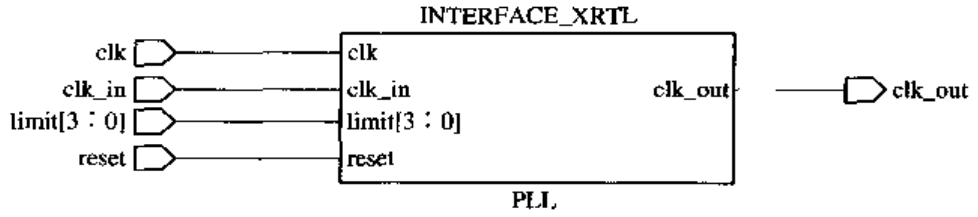


图 4.65　简单的数字锁相环的框图

表 4.21　数字锁相环端口说明

端口	宽度	方向	说明
clk	1	输入	高频系统时钟
clk_in	1	输入	输入时钟
limit	4	输入	计数器的上限，决定了输出时钟的频率。必须接近于输入时钟频率
reset	1	输入	PLL 的复位信号
clk_out	1	输出	输出时钟

2. 代码

下面是锁相环的一个实现代码。

```
/*****************************************************/
// MODULE:          锁相环
/*****************************************************/

//定义常量
`define DEL     1          //时钟到输出的延迟
`define CNT_SZ 4           //计数器中比特的数目，它定义了 PLL 的最大频率
`define DUTY 2             // DUTY 定义了输出时钟的占空比
                           // 2 = 50% 低, 50% 高
                           // 3 = 33% 低, 67% 高
                           // 4 = 25% 低, 75% 高 等等

//模块描述
module     PLL(reset,limit,clk,    clk_in,clk_out);

//端口描述
input              reset;          // PLL 的复位信号
input [`CNT_SZ-1:0]  limit;        //计数器的上限，这里是 4 位，最大值为 15
input              clk;            //高频系统时钟
input              clk_in;         //输入时钟

output             clk_out;        //输出时钟

//信号声明
wire               reset;
```

```
wire [`CNT_SZ-1:0]    limit;
wire                  clk;
wire                  clk_in;
wire                  clk_out;

reg  [`CNT_SZ-1:0]  counter;    //用于锁定时钟的计数器
reg                 reg_in;     //对输入时钟寄存

//赋值
assign #`DEL clk_out = (counter > limit/`DUTY) ? 1'b1 : 1'b0;

always @(posedge clk) begin
    if (reset) begin
        counter <= #`DEL 0;
    end
    else begin
        //寄存输入时钟以找到上升沿
        reg_in <= #`DEL clk_in;

        //在输入时钟的上升沿
        //理想情况下当 counter == limit 时同时发生
        //将计数器周期增加或减少，和输入时钟同步其相位
        if ((!reg_in & clk_in) && (counter != limit)) begin
            if (counter < limit/2) begin
                //输出时钟的边沿到达太晚，
                //所以从计数器减去 2，而不是一般情况下的 1
                if (counter == 1)
                    counter <= #`DEL limit;
                else if (counter == 0)
                    counter <= #`DEL limit - 1;
                else
                    counter <= #`DEL counter - 2;
            end
            //否则，通过不消耗计数器的这个周期来实现加一个另外的周期
        end
        else begin
            if (counter == 0)
                counter <= #`DEL limit;
            else
                counter <= #`DEL counter - 1;
        end
    end
end
endmodule       // PLL
```

简单的数字锁相环的综合结果如图 4.66 所示。

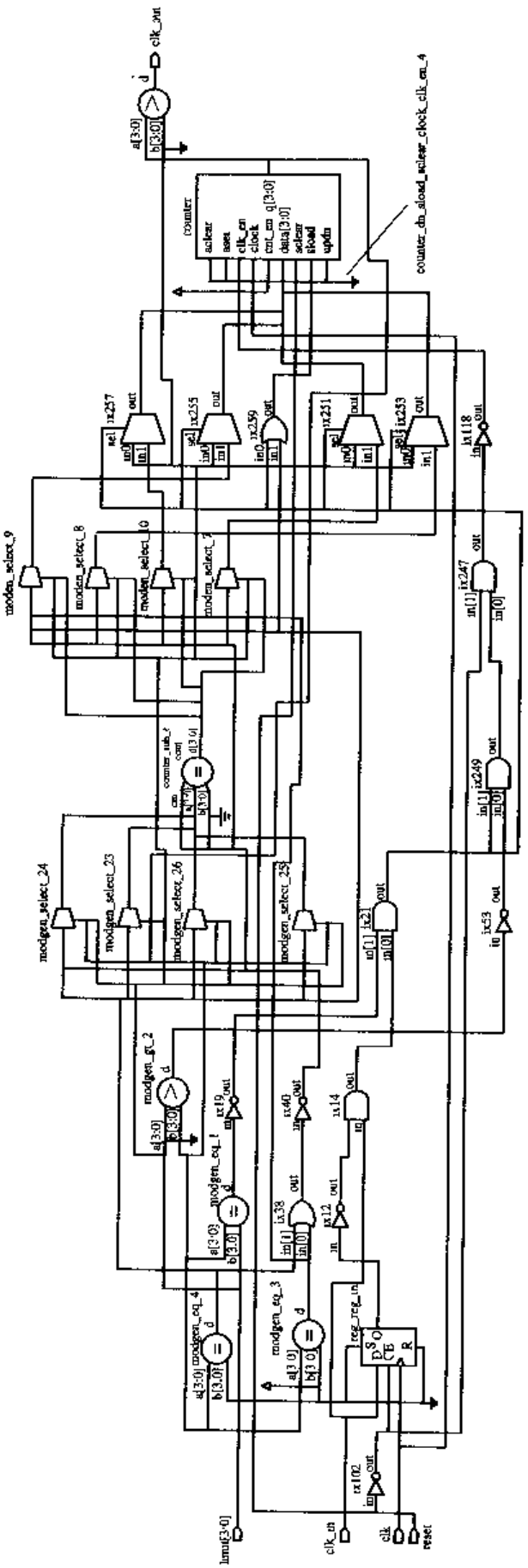


图 4.66　简单的数字锁相环的综合结果

4.12.2 较复杂的锁相环

在实际应用中，锁相环要比上述的设计复杂，这样可以得到更好的调整速度和控制精度。下面我们给出设计思路。

图 4.67 是较复杂锁相环的框图。

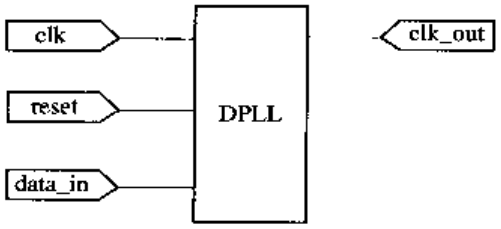


图 4.67 较复杂锁相环框图

设在这个数据通信系统中，每个数据位占的宽度是 64 个基本时钟周期。接收到的数据是 data_in，基本时钟是 clk。

首先，我们去掉输入信号中的短脉冲干扰。

说明：在通信信道上的“毛刺”是很危险的，可能错误地激活接收器。这意味着，输入到接收器瞬时的低信号将被看作一个 1 到 0 的突变，而这事实上只是噪声。所以在通信系统设计时，必须具有抗干扰性。

下面是实现去“毛刺”功能的代码。

```
always @(negedge clk)
 begin
     data_in_Dly1 <= data_in;
     data_in_Dly2 <= data_in_Dly1;
     data_in_Dly3 <= data_in_Dly2;
     data_in_Dly4 <= data_in_Dly3;
 end
always @(posedge clk or posedge Reset)
begin
     if(Reset)
       data_removeglitch <= 0;
     else begin
            if(data_in_dly1 && data_in_dly2 && data_in_dly3 && data_in_dly4)
                 data_removeglitch <= 1;
            else if (data_in_dly1==0 && data_in_dly2==0)          //0309
                 data_removeglitch <= 0;
     end
end
```

接下来，要找出输入信号的上升沿。用如下方式：

```
always @(negedge clk)
 begin
        data_removeglitch_Dly1 <= data_removeglitch;
        data_removeglitch_Dly2 <= data_removeglitch_Dly1;
 end
 wire          data_PosEdge;
assign     data_PosEdge = (data_removeglitch_Dly1 == 1 && data_removeglitch_Dly2 ==0)?1:0;
```

其次，要生成开始信号 receive_begin。在不同的协议中，对起始帧的定义是不相同的，所以这里不给出具体实现。

最后是实现相位的调整。由于一个数据位占用 64 个基本时钟，从接收数据中得到的时钟 clk_receive 应该是基本时钟的 64 分频。我们可以用一个 6 位的计数器 clkcnt 来计量。这个计数器每次递增的步长为 Step。例如：

```
always   @(negedge Clk or posedge Reset)
 begin
   if(Reset)
     ClkCnt <= 6'b100000;
   else
     ClkCnt <= ClkCnt + Step;
end
```

如果时钟同步得很好，则 Step 为 1。如果发现 clk_receive 比接收的数据提前，则需要将 clk_receive 的脉冲宽度变长，以便在下个脉冲跟接收数据的边沿对齐。同理，如果发现 clk_receive 比接收的数据滞后，则需要将 clk_receive 的脉冲宽度缩短。调整脉冲宽度的方法是调整步长 Step 的取值。

输出的时钟可以由如下方式得到：

```
assign    Clk106k = ClkCnt[5];
```

4.13 UART(通用异步收发器)

异步传输方式指收发两端各自有相互独立的位定时时钟，收方利用数据本身来进行同步的传输方式，如图 4.68 所示。

串口都包含通用异步接收器/发送器(UART)。

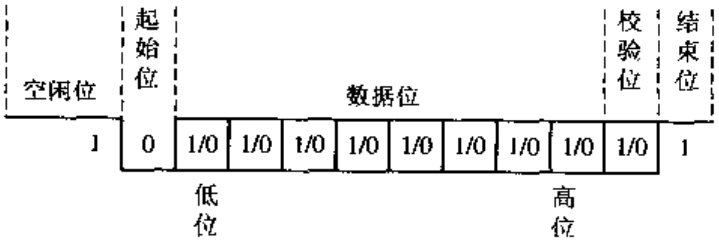


图 4.68 UART 中的帧格式

在 UART 通信过程中，接收要采用 m 倍的采样。

(1) 起始位检测：m 倍取样如图 4.69 所示。

(2) 数据位取样：隔 m 个采样周期后取样一次，共 n 次。

(3) 停止位检测：隔 m 个采样周期后取样并检测。

为了防止由于时钟不匹配而导致的误采样，对接收器，我们采用了较高的频率(频率为位速率的 8 倍)。

UART 中的接口结构如图 4.70 所示。

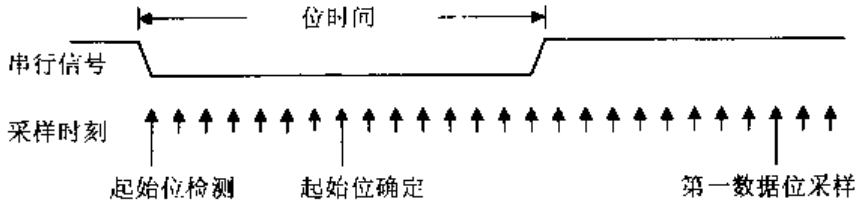


图 4.69　起始位的检测

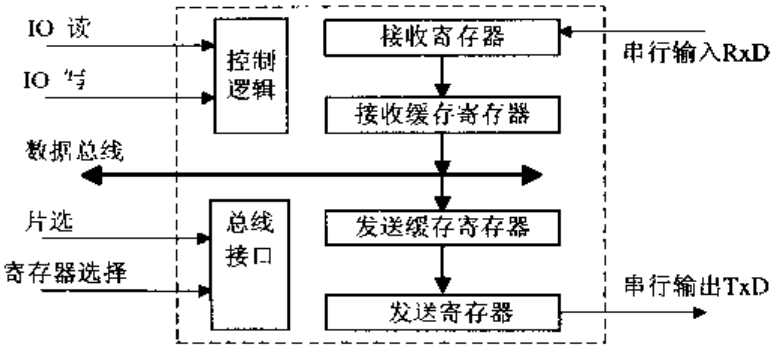


图 4.70　UART 中的接口电路

4.13.1　简单的 UART

1. 发送部分

这里仅给出设计的思路。

发送部分的框图如图 4.71 所示。发送部分的端口说明如表 4.22 所示。

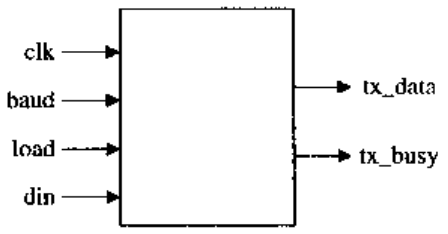


图 4.71　发送部分框图

表 4.22　发送部分的端口说明

输入端口名	端口描述
clk	系统时钟
baud	波特率
load	控制信号，有效时开始接收数据
din[7:0]	输入数据
输出端口名	端口描述
tx_Data	输出的串行数据
tx_Busy	标志信号，有效态，表示数据正在传输。当传输结束后，该标志变为 0

发射机与接收机都用状态机来设计。

发射机的状态机有三个状态：tx_idle、tx_start、tx_Data。其中：

tx_idle 表示发射机当前正处于空闲状态；

tx_start 表示发射机当前正处于发射起始位；

tx_Data 表示发射机当前正在发射数据。

接收到的数据经过并/串转换模块后发送出去，并带上起始位和终止位。

2. 接收部分

接收模块一种可能的框图如图 4.72 所示。接收部分的端口说明如表 4.23 所示。

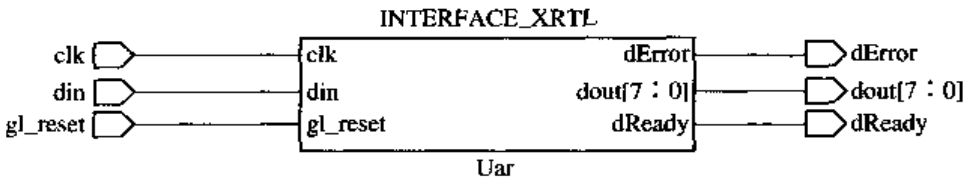


图 4.72　接收部分的框图

表 4.23　接收部分的端口说明

端口	宽度	方向	说明
clk	1	输入	时钟
din	1	输入	输入数据
gl_reset	1	输入	全局复位信号
dError	1	输出	错误标志
dout	8	输出	输出的数据
dReady	1	输出	标志信号

接收状态机有四个状态：rx_idle、rx_start、rx_data 和 rx_stop。其中：

rx_idle 状态表示接收机当前处于空闲状态；

rx_start 表示接收机正在接收起始位；

rx_data 表示接收机当前正在接收数据；

rx_stop 表示接收机当前正在接收停止位。

起始位的测试较简单：将输入数据放到一个移位寄存器中，如果低电平为一定时间，则判断是起始位。

下面是实现起始位检测的代码。

```
always @ (posedge clk)
begin
   if (reset or global_reset)
     shift_reg = 0;
else
    shift_reg = { shift_reg[2:0], dIn };
end
    assign valid = (shift_reg [0] == 0) &(shift_reg [2] == 0) &(shift_reg [3] == 1);
```

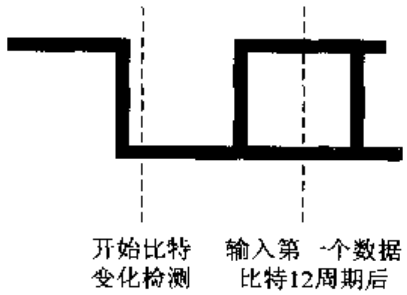


图 4.73　变化检测及采样示意图

为了保证采样的正确性，我们希望在数据比特的中间，而不是在接近边界处进行采样。为了防止由于时钟不匹配而导致的误采样，对接收器，我们采用了较高的频率(频率为位速率的 8 倍)。这种情况下，采样示意图如图 4.73 所示。

在检测到开始比特之后，应在 76 个时钟周期之后检测到结束比特。没有检测到结束位，应该报错。

4.13.2 复杂的 UART

现在考虑复杂一些的 UART，并将它与 CPU 联系起来。

该模块的框图如图 4.74 所示。

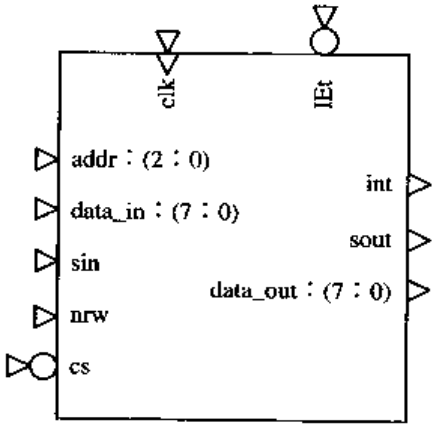


图 4.74　UART 的框图

该模块包括四个部分：地址解码模块、CPU 接口模块、时钟分频模块、串行接口模块，如图 4.75 所示。

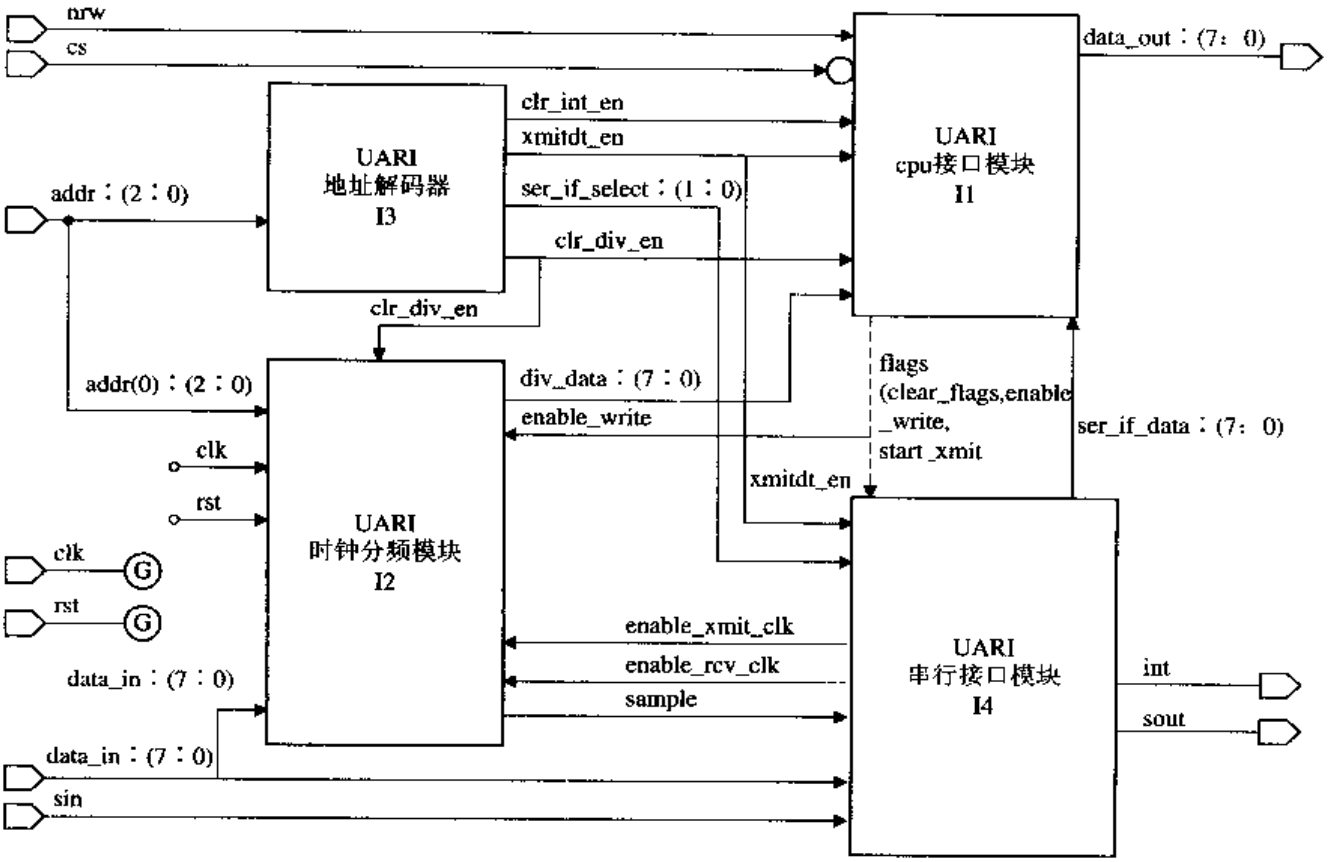


图 4.75　UART 顶层模块的组成

下面是 UART 顶层模块的代码。

代码描述如下：

```
module uart_top( addr, clk, cs, nrw, rst,
        sin, data_out,
        int,
    data_in,sout
);

//端口描述
input   [2:0] addr;
input         clk;
input         cs;
input   [7:0] data_in;   //来自 CPU
input         nrw;       //为 0 时读，为 1 时写
input         rst;       //复位信号
input         sin;       //串行输入
output [7:0] data_out;   //送到 CPU 去的并行数据
output        int;       //中断信号
output        sout;      //发出的串行数据
//信号声明(略)

//内部信号声明
wire        clear_flags;
wire        clk_div_en;
wire        clr_int_en;
wire [7:0] div_data;
wire        enable_rcv_clk;
wire        enable_write;
wire        enable_xmit_clk;
wire        sample;
wire [7:0] ser_if_data;
wire [1:0] ser_if_select;
wire        start_xmit;
wire        xmitdt_en;

//子模块的例化
address_decode I3(
    .addr           (addr),
    .clk            (clk),
    .rst            (rst),
    .clk_div_en     (clk_div_en),
    .clr_int_en     (clr_int_en),
    .ser_if_select  (ser_if_select),
```

```
    .xmitdt_en          (xmitdt_en)
);

clock_divider I2(
    .addr               (addr[0]),
    .clk                (clk),
    .clk_div_en         (clk_div_en),
    .data_in            (data_in),
    .enable_rcv_clk     (enable_rcv_clk),
    .enable_write       (enable_write),
    .enable_xmit_clk    (enable_xmit_clk),
    .rst                (rst),
    .div_data           (div_data),
    .sample             (sample)
);

cpu_interface I1(
    .clk                (clk),
    .clk_div_en         (clk_div_en),
    .clr_int_en         (clr_int_en),
    .cs                 (cs),
    .div_data           (div_data),
    .nrw                (nrw),
    .rst                (rst),
    .ser_if_data        (ser_if_data),
    .xmitdt_en          (xmitdt_en),
    .clear_flags        (clear_flags),
    .data_out           (data_out),
    .enable_write       (enable_write),
    .start_xmit         (start_xmit)
);

serial_interface I4(
    .clear_flags        (clear_flags),
    .clk                (clk),
    .data_in            (data_in),
    .enable_write       (enable_write),
    .rst                (rst),
    .sample             (sample),
    .ser_if_select      (ser_if_select),
    .sin                (sin),
    .start_xmit         (start_xmit),
    .xmitdt_en          (xmitdt_en),
```

```
    .enable_rcv_clk     (enable_rcv_clk),
    .enable_xmit_clk    (enable_xmit_clk),
    .int                (int),
    .ser_if_data        (ser_if_data),
    .sout               (sout)
);
```

对串行接口模块和 CPU 接口模块，我们作进一步说明。

1. 串行接口模块

UART 通过该模块与通信方交互。它包括两个子模块：

(1) 发送接收控制模块。该模块将负责把并行数据串行化，并通过串口发送出去；同时通过串口接收串行数据，转换为并行数据，并产生相应的状态信号。

(2) 状态寄存器模块。该模块将状态信号(发送完成、接收完成、正在发送、正在接收)送入状态寄存器相应的位中，并负责产生中断。

该模块的组成如图 4.76 所示。

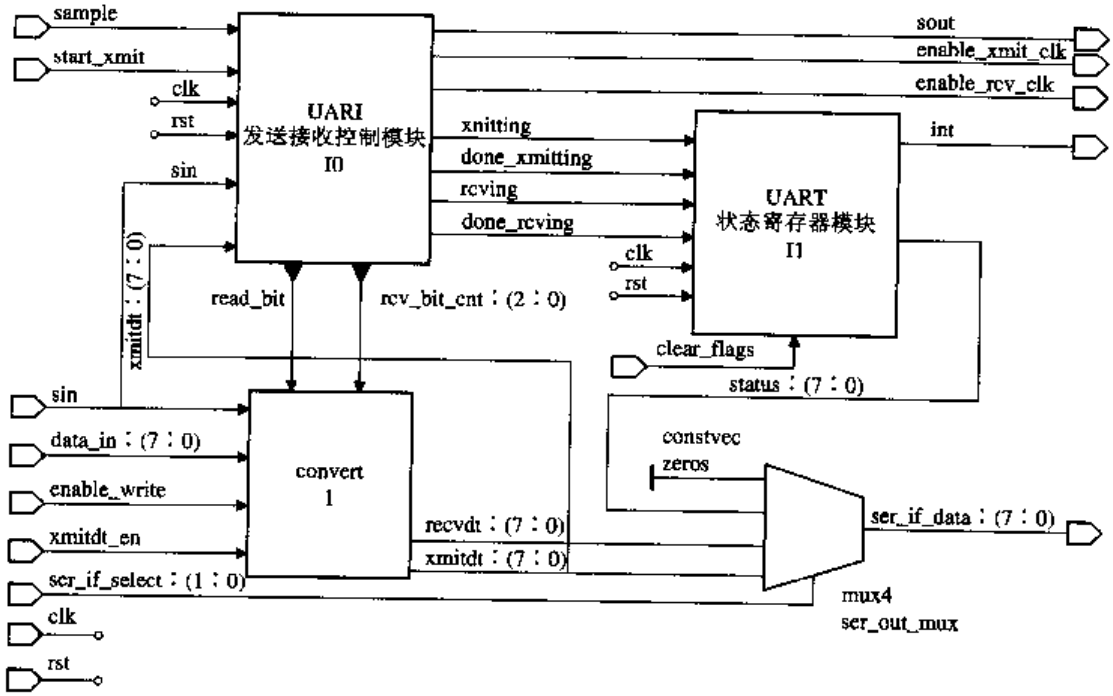


图 4.76　串行接口模块组成框图

代码描述如下：

```
//模块描述
module serial_interface( clear_flags, clk, data_in, enable_write,
    rst, sample, ser_if_select, sin, start_xmit, xmitdt_en,
    enable_rcv_clk,
    enable_xmit_clk,
    int, scr_if_data, sout
);
```

```
//端口描述(略)
//信号声明(略)

//内部信号声明(略)

//子模块例化
status_registers I1(
    .clear_flags        (clear_flags),
    .clk                (clk),
    .done_rcving        (done_rcving),
    .done_xmitting      (done_xmitting),
    .rcving             (rcving),
    .rst                (rst),
    .xmitting           (xmitting),
    .int                (int),
    .status             (status)
);

xmit_rcv_control I0(
    .clk                (clk),
    .rst                (rst),
    .sample             (sample),
    .sin                (sin),
    .start_xmit         (start_xmit),
    .xmitdt             (xmitdt),
    .done_rcving        (done_rcving),
    .done_xmitting      (done_xmitting),
    .enable_rcv_clk     (enable_rcv_clk),
    .enable_xmit_clk    (enable_xmit_clk),
    .rcv_bit_cnt        (rcv_bit_cnt),
    .rcving             (rcving),
    .read_bit           (read_bit),
    .sout               (sout),
    .xmitting           (xmitting)
);
...
endmodule // serial_interface
```

2. CPU 接口模块

该模块与 CPU 交互。通过该接口模块，CPU 向 UART 发出清除、写使能和开始传送标志信号。该模块的组成如图 4.77 所示。

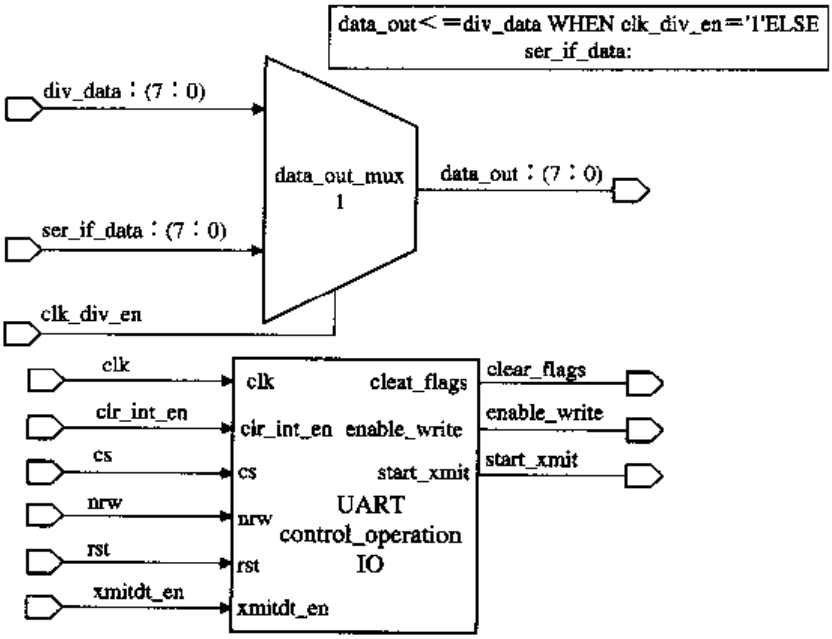


图 4.77　CPU 接口的框图

代码描述如下：

```
//模块描述
module cpu_interface( clk, clk_div_en, clr_int_en, cs,
    div_data, nrw, rst, ser_if_data, xmitdt_en,
    clear_flags, data_out, enable_write, start_xmit
);

//端口描述(略)
//信号声明(略)

//子模块例化 control_operation 是一个状态机。发出清除、写使能和开始传送控制信号
control_operation I0(
    .clk            (clk),
    .clr_int_en     (clr_int_en),
    .cs             (cs),
    .nrw            (nrw),
    .rst            (rst),
    .xmitdt_en      (xmitdt_en),
    .clear_flags    (clear_flags),
```

```
    .enable_write       (enable_write),
    .start_xmit         (start_xmit)
);
assign data_out = (clk_div_en == 1) ? div_data : ser_if_data;
endmodule
```

4.14 FIR 滤 波 器

在许多数字信号处理的应用领域，都用到滤波器来实现特定函数。例如，在视频领域，经常用滤波器来实现插值运算，以改善图像质量。

一个离散时间系统 H(z)=B(z)/A(z)，若分母多项式 A(z)的系数 $a_1=\ldots=a_{N-1}=0$，那么该系统即变成有限冲击响应(FIR)系统，即

$$H(z)=b_0+b_1Z^{-1}+b_{N-1}Z^{-(N-1)}=\sum_{n=0}^{N-1}b_nZ^{-n}$$

显然，系数 b_0，b_1，…，b_{N-1} 即是该系统的单位抽样响应 h(0)，h(1)，…，h(N-1)，且当 n>M 时，h(0)=0。

FIR 滤波器即有限长单位脉冲响应滤波器，其系统函数为

$$H(z)=\sum_{n=0}^{N-1}h(n)Z^{-n}$$

FIR 滤波器稳定性较好，易于硬件实现。它属于数字滤波器中的非递归类型，其瞬间的输出响应仅仅取决于当时及以前的激励，而与以前的输出无关。

对 FIR 滤波器来说，每次采样都需要利用本次采样之前的 L 个值。这样，L+1 个采样值分别与常系数 h(n)相乘(h(n)是常数，它决定了滤波器的特征函数)，然后累加形成本次采样的结果。

FIR 滤波器的主要缺点是它在达到同样给定性能的条件下，其阶次比 IIR 滤波器要高，相应地，其延时比同样性能的 IIR 滤波器要大得多。

下面给出一个 FIR 的实现的思路。

首先，定义 FIR 的输入输出。这里实现的 FIR 较为简单，端口定义如下：

```
module FIR(clk,reset,data_in,data_out);
```

然后定义一个移位寄存器数组来保存最近 L 次采样值。这里取 L 为 8，所以移位寄存器数组的长度也为 L。在每次采样后，都要进行移位以保证该数组中只保存最近 L 次的采样值。

```
reg [7:0] shift_reg[7:0];
```

运行时，计算当前采样的响应(输入数据与滤波器抽头系数相乘)，累计最近 L 次采样，然后将结果输出。同时，要更新移位寄存器数据中的内容。

图 4.78 为滤波器的数据流图。

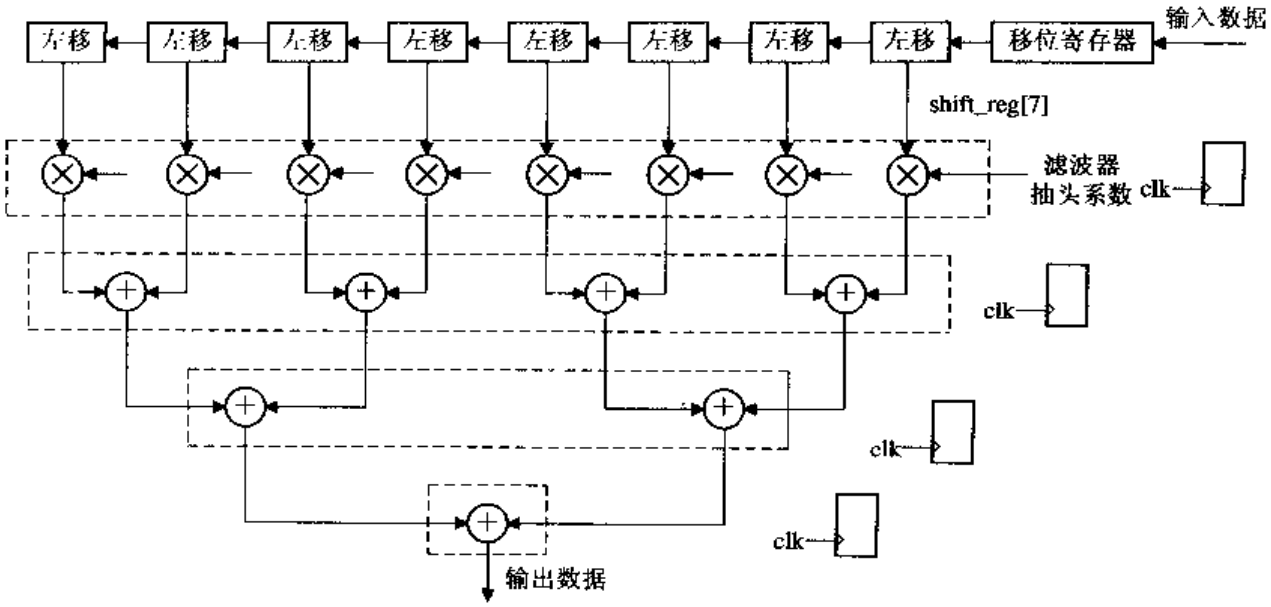


图 4.78　滤波器数据流图

其中，加法采用了三级流水结构，加上乘法共四级流水结构。在每级流水时，数据都用时钟寄存一次，以加快速度。

4.15　练　　习

1. 完成 32 位 booth 乘法器的实现。

2. 在 4.3.3 节中用移位寄存器产生定时标志信号的代码进行仿真，并分析其结果。

3. 完成图 4.78 所示的 8 抽头滤波器设计，要求采用流水线结构，并利用本章所介绍的加法器和乘法器结构。

第5章 存储器的结构和设计

5.1 基础知识

5.1.1 存储机制及存储器类型

现有两种方法可实现存储的功能。

第一种方法是使用正反馈，一个或更多的输出信号与输入相连接。这类电路称为多频振荡器电路。其中，以双稳态元件(触发器)最为普遍，单稳态及无稳态电路也有一定应用(无稳态多频振荡器可以作为晶体振荡器及时钟发生器，而单稳态多频振荡器可以作为脉冲发生器)。利用正反馈的方法来保存数据，信号值可以一直保持下去，所以这类电路称为静态电路。

第二种方法是使用存储电荷来存储信号值。此方法在MOS领域用得非常普遍。因为电荷总是要泄漏的，所以这类电路要不断地进行刷新，此类电路称为动态电路。

存储器具有许多不同的形式及类型。存储器类型与要求的存储器规模、存取所存数据的时间、存取方式以及系统要求有关。

存储器基于存储功能、存取方式以及存储机制来划分，可以分为只读与读写存储器。

读写存储器的优势在于：它既可以读，也可以写，读写时间差不多。它属于易失存储器，当供电电压关掉时，数据也没了。而只读存储器可将信息编码到电路的拓扑结构(例如，通过增加或移出二极管或晶体管)。此拓扑是固化的，数据不能被改动，它只能被读取。ROM结构就属于非易失性存储器，没有电时也不会导致存储数据的损失。此外，还有NVRWM(非易失性读写存储器)。此一类的存储器有EPROM(可擦除可编程只读存储器)、EEPROM(电可擦除可编程只读存储器)以及FLASH MEMORY(闪存)。

存储器也可以基于存储器中数据存取的次序来分类。绝大多数存储器属于随机存取，存储位置可以按随机顺序进行读或写。

某些存储器类型对读取顺序进行了限制，以得到更快的存取时间、更小的面积，或实现特定功能。这种存储器的例子是FIFO(先进先出)、LIFO(后进先出，通常用作堆栈)、移位寄存器、CAM(内容可取址存储器)结构。

半导体存储器的分类如表5.1所示。

表 5.1　半导体存储器的分类

存储器	存储器类别	
RWM	随机存取	SRAM(静态存储器)
		DRAM(动态存储器)
	非随机存取	FIFO
		LIFO
		移位寄存器
		CAM
NVRWM	EPROM	
	EEPROM	
	FLASH MEMORY	
ROM	不可编程	
	可编程(PROM)	

说明：

NVRWM 及 ROM 都属于静态存储器。

有时候，将 EPROM/EEPROM/FLASH MEMORY 归到可编程 ROM 中。

存储器的设计与一般的逻辑设计不同。存储器设计中，最重要的指标是密度与速度，可以对数字门中的一些指标放宽。例如，为了获得更快的速度，可以牺牲噪声容限(通过外围电路可以对抗噪性能进行弥补)。

在存储器阵列中，延迟主要是由内连线寄生产生的，这与一般的逻辑门不同。

下面分别对这些存储器进行介绍。

5.1.2　SRAM

1. 6 管 SRAM

SRAM 的单元一般由 6 个 MOS 管组成：由工作管 M_1、M_2 和负载管 M_3、M_4 构成两个等同的反相器交叉耦合连接成为双稳态触发器，存储的信息通过 M_5、M_6 两个门管与位线 BL 和 nBL 进行交流，由字驱动线 WL 控制门管的开关状态，如图 5.1 所示。这样，位线上的一对反相信号可以通过门管使触发器置数，也就是写入过程。另一方面，触发器内所存储的数据可以通过门管向位线传送，也就是读出过程。

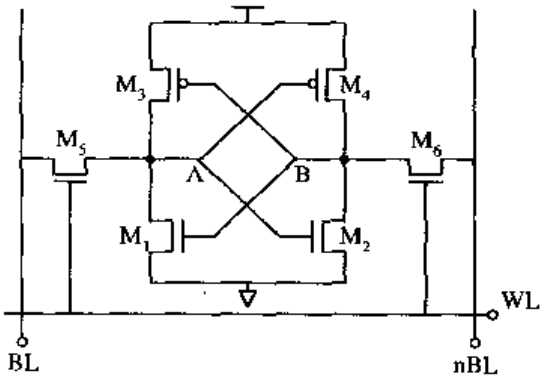


图 5.1　6 管 SRAM 结构

(1) 数据写入过程。在位线 BL 和 nBL 上分别加上高电平和低电平，然后在字驱动线 WL 上加正脉冲，使门管 M_5、M_6 导通，这时 BL 的高电平通过 M_5 对 A 点节点电容充电，使 A 点电平逐渐升高，使 M_2 管导通；同时由于 M_6 管导通，使 B 点节点电容通过 M_6 向位

线 nBL 放电至低电平。当 WL 回落到低电平时，M_5、M_6 截止，触发器内 A 点保持“1”、B 点保持“0”。

(2) 数据读出过程。在字驱动线 WL 上加正脉冲，将门管打开。若 A 为“1”，则位线 nBL 通过 M_6、M_2 形成导电通路，nBL 上有较大的电流通过；而 B 为“0”，M_1 管截止，BL 上基本没有电流流过。把 BL、nBL 接到差动放大器上，根据电流方向鉴别存储单元存储信息。

(3) 数据保持过程。门管截止，将位线和触发器隔断，位线上的电平就不再影响触发器状态；另外，负载管将补充高电平端的漏电流，维持高电平不变。但掉电后或电源电压降低到一定程度时，存储单元的数据就会丢失。

SRAM 除了作为单独的产品应用外，在芯片内部也大量使用，例如用作微处理器的堆栈寄存器、缓存、寄存器阵列等。由于 SRAM 在芯片内部不需要驱动片外负载电容，因而速度可以大大加快。并且因为 SRAM 集成在芯片内部，不存在封装费用，可以大大降低成本。

在 6 管 SRAM 结构中，上拉器件的作用在于补偿电荷损失。上拉器件可以不使用传统的 PMOS 器件，通过薄膜工艺在单元结构的顶部镀膜也可以实现。这种 PMOS 薄膜晶体管(TFTs)与通常的器件相比，性能要差一些。该类器件的优点在于泄漏少，并且对软击穿不敏感，从而增加了器件的可靠性。与下面介绍的电阻负载相比，它具有更低的静默电流。TFT 单元是特大 SRAM 存储器最理想的选择，特别适用于用电池的移动产品。

2. 4 管 SRAM 单元

上面介绍的 6 管 SRAM 单元，尽管简单可靠，但却占据大量的面积。因此在设计大的存储器阵列时，需要对此结构进行改进。

图 5.2 给出了另一种 SRAM 结构，它称为电阻负载 SRAM 单元(也称为 4 管 SRAM 单元)。此单元的特点在于一对交叉反接的 CMOS 反相器用一个电阻负载 NMOS 反相器来代替。PMOS 晶体管用一个电阻取代，因而布线简化，并大大降低了 SRAM 单元的面积。

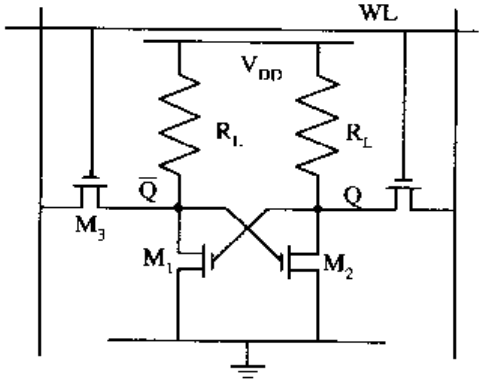


图 5.2　4 管 SRAM 结构

这种拓扑结构有一个主要的缺点：电阻需要足够高，以使得噪声容限 NM_L 维持在一定的范围内，并减少静态功耗。而过高的电阻值使得传播延时变坏，并增加单元尺寸。

说明：

(1) 存储器阵列包含大量的晶体管，如果设计不当，功耗会很可观。所以，在设计 SRAM 单元时，首要目标是将静态功耗降到最低(低功耗设计)。将负载电阻加大(可以通过使用无掺杂多晶硅来实现大的电阻)，能降低功耗。增加负载电阻，会使传播延时也增大。但可以用下述方法来解决延时问题：将位线预充到 V_{DD}，这样只有在预充时，才发生由低到高的转换，而在读操作时不发生。

(2) 众所周知，双极型工艺速度较快，但功耗大，密度小；MOS 工艺正好相反。BiCMOS 存储器将 MOS 存储阵列的高密度与高性能的双极型外围电路(如驱动器及读出放大器)结合起来。其存取时间与三极管存储器相当，并保持了 MOS 存储器的高密度。

3. SRAM 的阵列结构

SRAM 的阵列结构如图 5.3 所示。

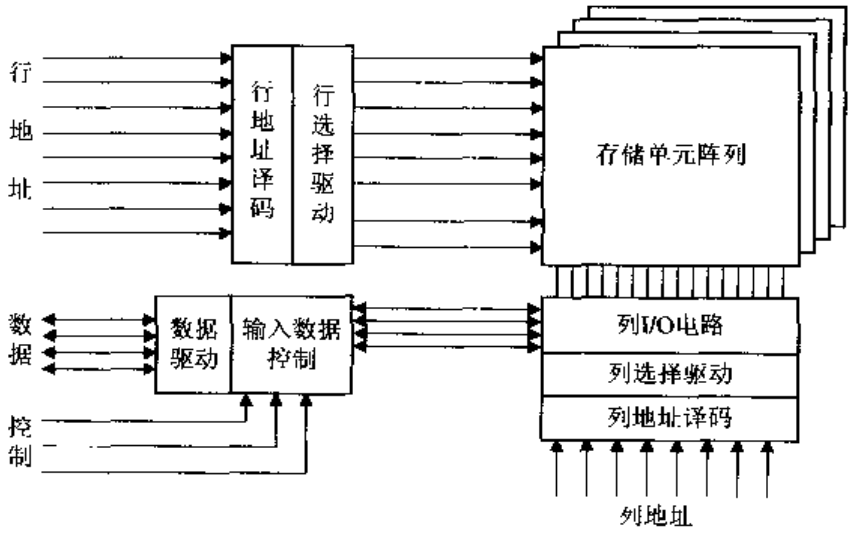


图 5.3 SRAM 的阵列结构

4. SRAM 芯片的控制信号

ADD：地址信号，在芯片手册中通常表示为 A_0，A_1，A_2，…

$\overline{CS}$：芯片选择，低电平时表示该芯片被选中。

$\overline{WE}$：写允许，低电平表示写操作，高电平表示读操作。

D_{out}：数据输出信号，在芯片手册中通常表示为 D_0，D_1，D_2，…

D_{in}：数据输入信号。

$\overline{OE}$：数据输出允许信号。

5. SRAM 的时序

读时序：

地址有效→CS 有效→数据输出→CS 复位→地址撤销

SRAM 的读时序如图 5.4 所示，写时序如图 5.5 所示。

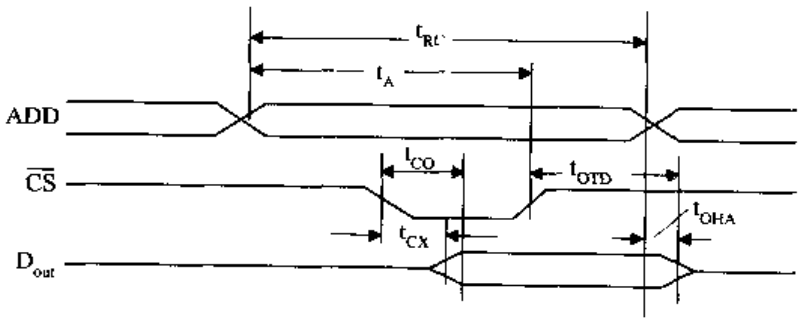


图 5.4 SRAM 读时序

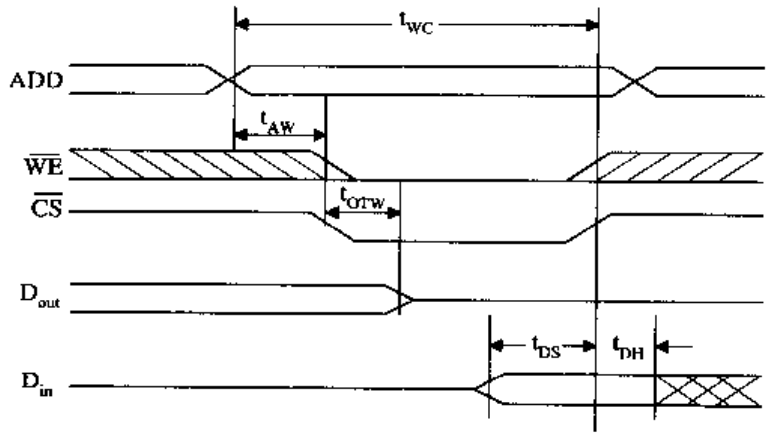


图 5.5　SRAM 写时序

5.1.3　DRAM

从上面对电阻负载 SRAM 单元的描述不难看出，负载电阻唯一的功能是补充泄漏的电荷。能不能去掉这些负载呢？当然可以！通过周期性地重写单元内容，一样可以补充电荷损失。这种刷新操作(包括对单元内容的读操作，然后是写操作)，其发生频率应足够高，这样才不会因为泄漏问题而改变存储器单元的内容。一般而言，刷新操作应该每隔 1 到 4 毫秒发生一次。

这样，就出现了另一类存储器：动态存储器。动态存储器中的内容，是靠电容上的电荷来存储的。

1) 3 管动态存储单元

通过对上面描述的 4 管 SRAM 单元进行改造，我们可以获得 3 管动态存储单元的结构。首先，把其中的负载电阻去掉。其次该结构还可以进一步简化。我们注意到，4 管单元既存储了数据的值，也存储了它的互补值，这是没有必要的，因而可以去掉一个晶体管(例如 M_1)，这就得到了图 5.6 中的 3 管 DRAM 单元。

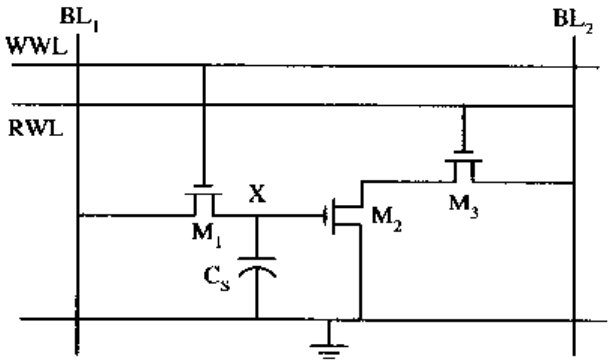


图 5.6　3 管 DRAM 结构

尽管在当今大型存储器中，已不再采用该方式，但由于它在设计与操作上非常简单，所以许多 ASIC 电路中还采用这种结构来构造内嵌的存储器。

通过在 BL_1 上放上合适数据值(0 或 1)，并使写字线(WWL)变高，可以实现对单元的写操作。当 WWL 变低后，写入的数据用电容 C_S 上的电荷来表示。读数据时，读字线(RWL)变高。如果存储的数据为 1，X 为高电平，这时候 M_2 与 M_3 都导通，位线 BL_2 被拉低。如果存储的数据为 2，X 为低电平，M_2 不导通，BL_2 保持为高。注意单元是反相的：也就是说，在位线上的值是所存储信号的相反值。刷新时，一般方法是先读存储数据，将它的相反值放在 BL_1，并使 WWL 一直为高。

与静态单元相比，动态存储单元结构更简单，面积也小了许多。该结构还可以进一步简化。例如，BL_1 与 BL_2 可以合为一条线，或者将 RWL 与 WWL 线合并。当然，这些简化是以操作更复杂为代价的。

与其它的 DRAM 方法不同，在读操作的时候，3 管单元的存储值不受影响。

说明：当写一个 1 时，存储于存储节点 X 的值等于 $U_{WWL}-U_{Tn}$。这种阈值的损失降低了在读操作时流经 M_2 的电流，并增加了读取时间。为了防止此现象，在一些设计中，将 U_{WWL} 值升到高于 V_{DD}。

2) *1 管动态存储器单元*

还可以进一步对单元进行简化，这就是 1 管 DRAM 单元。这是当前 DRAM 产品中用得最多的结构。图 5.7 给出了 1 管 DRAM 的结构。

该结构的操作非常简单。在一个写周期内，数据值传到位线 BL 上，字线 WL 被升高。电容要么充电，要么放电，这取决于所存的数据值。当执行一个读操作前，位线被预充到电压 U_{PRE}(充电电压)时，当字线为高时，在位线与存储电容之间进行电荷重新分配。这将导致在位线上电压的改变，变化的方向决定了存储的数据值。

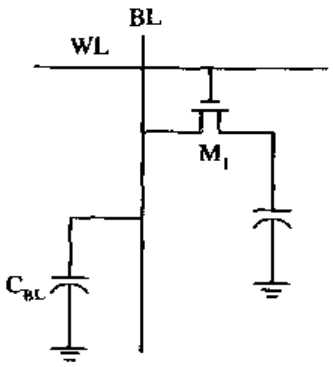


图 5.7　1 管 DRAM 结构

1 管 DRAM 单元与 3 管 DRAM 单元的主要差别包括：

在 1 管 DRAM 中，每个位线都有一个读出放大器。这是由于读操作时电荷重新分配的结果。读出放大器可以加快读操作的速度。需要说明的是，DRAM 存储器是单端的(single-ended)，这与 SRAM 单元不同，在 SRAM 中数据值及它的互补值都出现于位线上。这使得读出放大器的设计变得复杂了。

在 1 管 DRAM 单元中，读操作具有破坏性。这意味着，在读操作期间，存储于单元中的电荷数量会发生变化。读操作后，原来的数据值必须重存。读与刷新操作要在一个 1 管单元中交替进行。

3 管单元存储数据要靠栅电容上的电荷，与 1 管单元不同，它需要一个额外的电容，此电容必须被显式地包含在设计中。为了可靠，此电容的最小值为 10～15 F。将这样一个大的电容放置于很小的面积内是 DRAM 设计的一个难题。

3) DRAM 阵列结构

DRAM 阵列结构如图 5.8 所示。

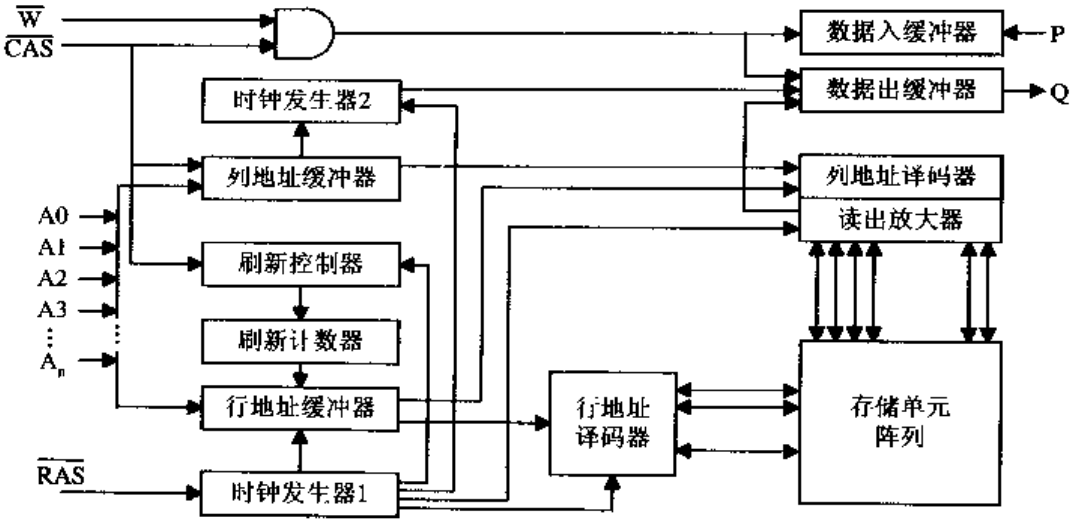


图 5.8　动态存储器芯片阵列结构

4) DRAM 时序

(1) 读周期：

行地址有效→行地址选通→列地址有效→列地址选通→数据输出→行选通、列选通及地址撤销

简单地说，从 DRAM 读数据需要给出行地址和列地址，行地址和列地址信号从同一批引脚走，用两根信号线 RAS(Row Address Signal)和 CAS(Coloumn Address Signal)来区分。所以典型的 RAM 用法是，先给 RAS，再给 CAS，经过一段延时，便可以在数据端读出数据。为什么在 DRAM 中，行地址与列地址要用相同的地址线呢？这是因为在早期，设计者给 DRAM 存储器选择了一种复杂的地址配置，行地址与列地址按顺序出现在同一个地址总线上，以节省封装引脚。此方法尽管不方便，却一直被延续下来。

在复用地址方案中，使用者必须提供两个主要的控制信号 RAS 和 CAS，以分别表明行与列地址。RAS 和 CAS 信号的分离必须足够大，以保证所有的操作能够完成。读周期时序如图 5.9 所示。

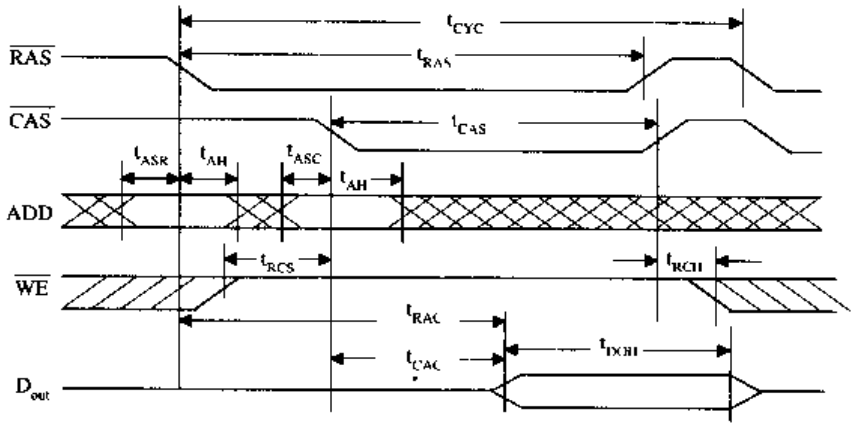


图 5.9　读周期时序

(2) 写周期：

行地址有效→行地址选通→列地址、数据有效→列地址选通→数据输入→行选通、列选通及地址撤销

写周期时序如图 5.10 所示。

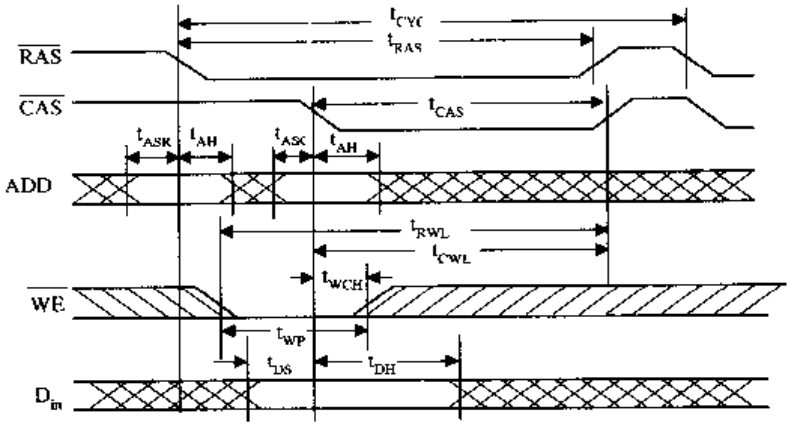


图 5.10　写周期时序

(3) 刷新时序。DRAM 的刷新操作较为复杂。这里仅给出一种，如图 5.11 所示。

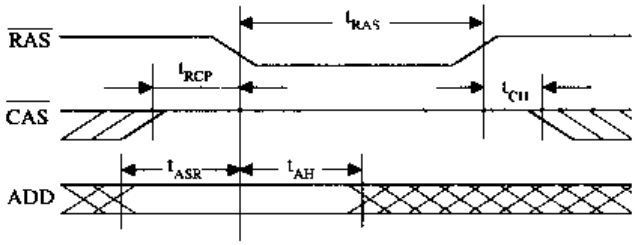


图 5.11　刷新操作时序

下面介绍一些 DRAM 的类型。

1. FPM DRAM(快速页式 DRAM)

快速页式 DRAM 的原理：因为读一个数据给两个地址太慢了，所以给一个 RAS，然后跟着一串 CAS，假设这一串 CAS 都是用的同一个 ROW 地址，这样就可以快一些。当时钟频率太高时，由于没有足够的充电保持时间，将会使读出的数据不可靠。

快速页式 DRAM 的操作时序如图 5.12 所示。

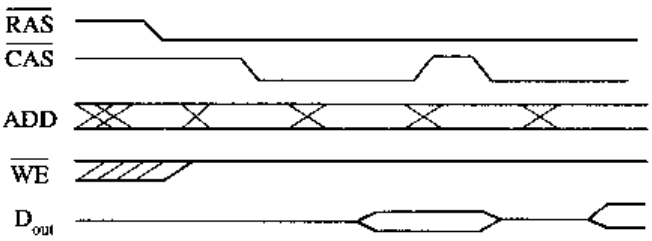


图 5.12　快速页式 DRAM 时序

2. EDO DRAM(扩展数据输出 DRAM)

与快速页式 DRAM 相比，EDO DRAM 的优点在于：读数据时，不需要保持地址线上的信号，在数据线读出数据的同时，就可以给下一个 CAS，所以它的速度要比 FPM DRAM 快。该存储器的时序如图 5.13 所示。

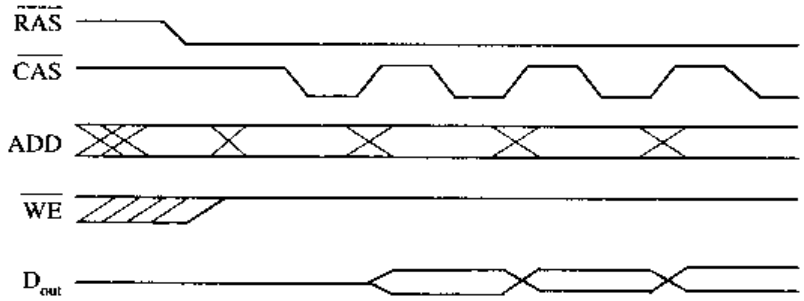


图 5.13　扩展数据输出动态读写存储器

3. BEDO DRAM(突发方式 EDO DRAM)

BEDO DRAM 指 Burst Extended Data Output Dynamic RAM。所谓 Burst 突发方式，是 CPU 一次读 4 个连续的 BYTE，所以就只送一次 RAS 和 CAS，BEDO DRAM 内部的寄存器记下地址，然后在数据线上连续输出 4 个连续的 BYTE。这样后来的 BYTE 根本不给地址，所以所用的 CPU 周期更少。这种存储器可以对可能所需的 4 个数据地址进行预测并自动地预先形成。

4. SDRAM

SDRAM 指 Synchronous Dynamic RAM，也就是同步内存。所谓同步，是指数据的输出和主板的时钟同步，这样 RAM 的读写不再依赖于 RAS 和 CAS，而是和总线的时钟同步的。SDRAM 的控制比较复杂，SDRAM 有命令寄存器，命令由地址线上给出，其操作类似于状态机。实际上它读 4 个连续 BYTE 时是和 BEDORAM 一样的，只给一次行列地址。但由于它有命令，因此还可以一次读 8 个或 16 个连续 BYTE。SDRAM 有两个 Module 可以轮换工作，在 Module 1 输出时就给 Module 2 命令信号。

图 5.14 给出了一个 8 M×8 SDRAM 的系统框图。

传统的 DRAM，对存储器中存放的数据字进行访问时，需要分别给出地址。在 SDRAM 中，仅需要一个首地址就可以对一个存储块进行访问。

与传统 DRAM 相比，SDRAM 增加了一个可编程模式寄存器。模式寄存器用来定义特定的操作模式。它包括选择突发长度、突发类型、CAS 延迟、操作模式和写突发模式。图 5.15 给出了一个模式寄存器的示例。

采用 SDRAM 可大大改善内存条的速度和性能，系统设计者可根据处理器要求，灵活地采用交错或顺序脉冲。

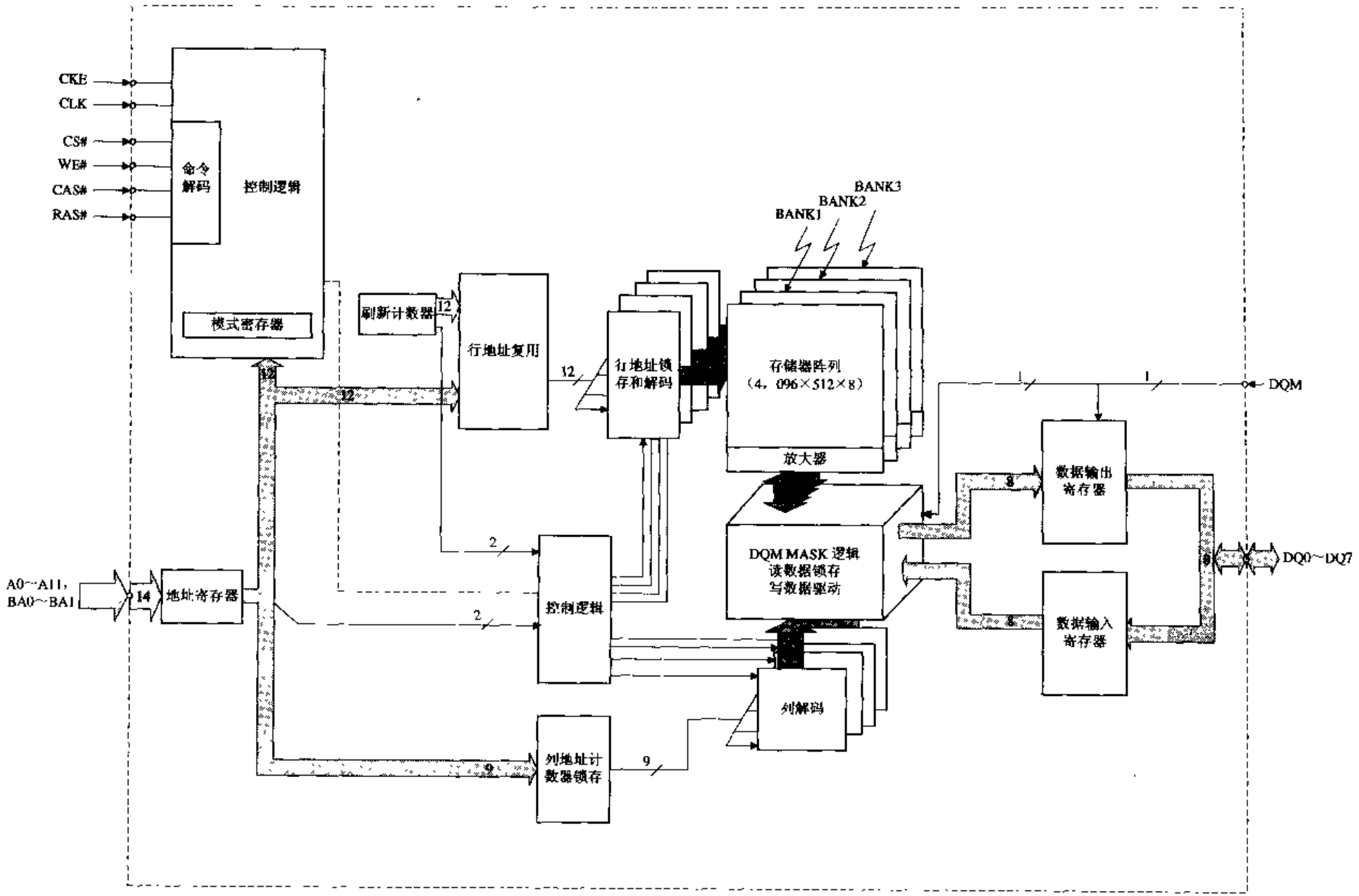


图 5.14　8 M×8 SDRAM 系统框图

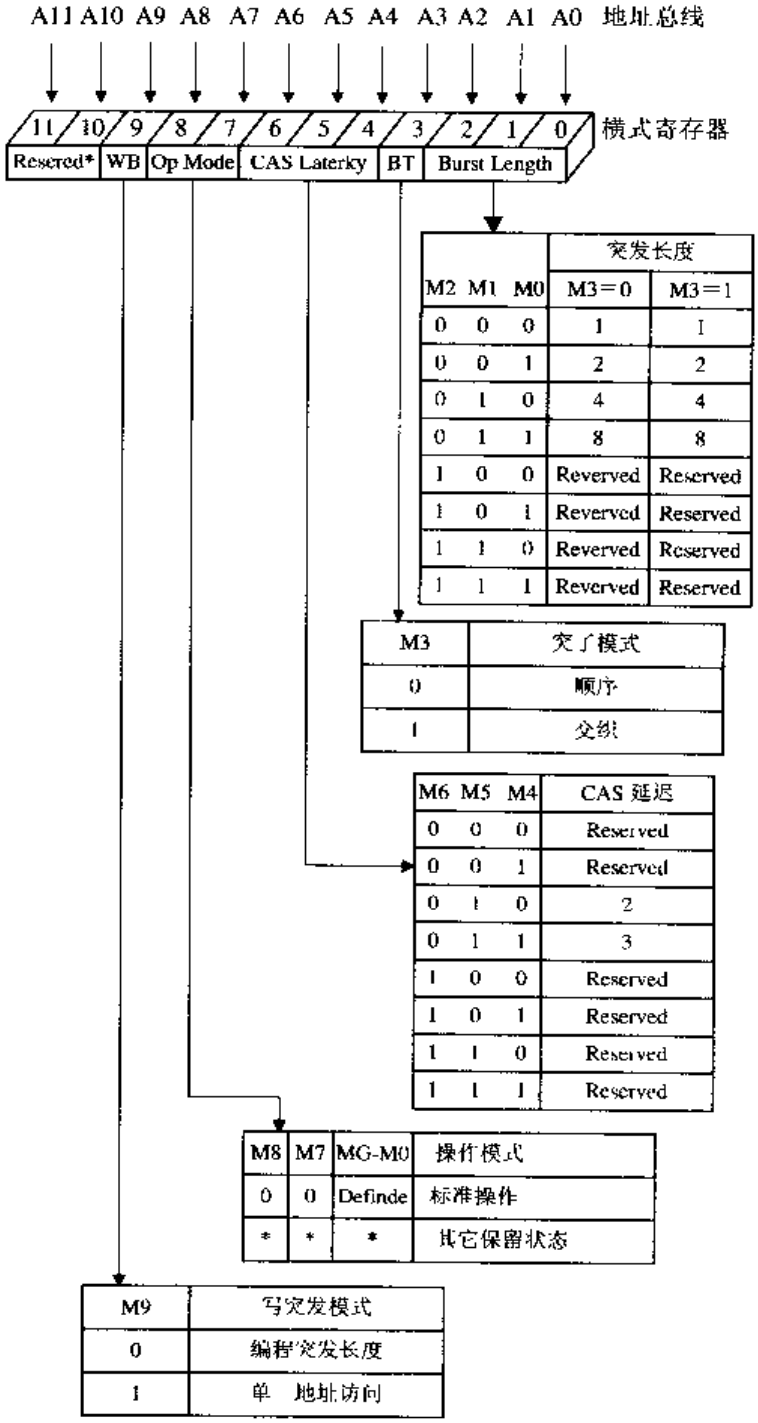


图 5.15 模式寄存器定义

5.1.4 FIFO

FIFO 的设计代码可以参见第 4 章内容。

5.1.5 移位寄存器

移位寄存器的设计代码参见第 4 章内容。

5.1.6 CAM

CAM(关联存储器)框图如图 5.16 所示，其基本结构如图 5.17 所示。

图 5.18 给出了一个关联存储器的访问实例。

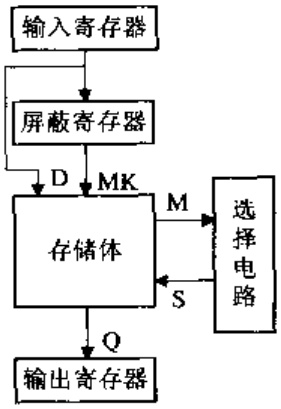


图 5.16 关联存储器框图

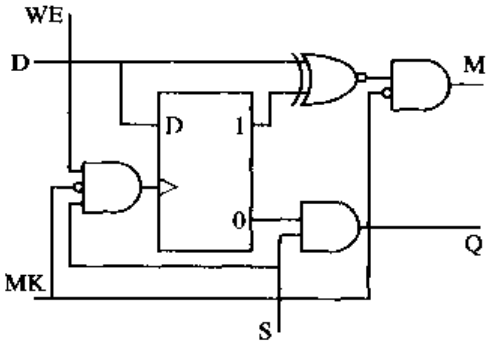


图 5.17 关联存储器基本结构

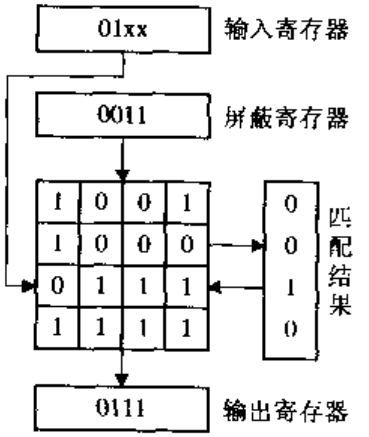


图 5.18 关联存储器访问实例

5.1.7 ROM

ROM(只读存储器)有着大量的应用范围，例如，有固定应用的处理器所用的程序存储，包括洗衣机、计算器和游戏机的处理器等。

ROM 单元中的内容是永久固定的，因而大大简化了它的设计。设计的单元通过字线的活动使得一个 0 或者 1 出现在位线上。图 5.19 展示了几种 ROM 的掩膜结构。

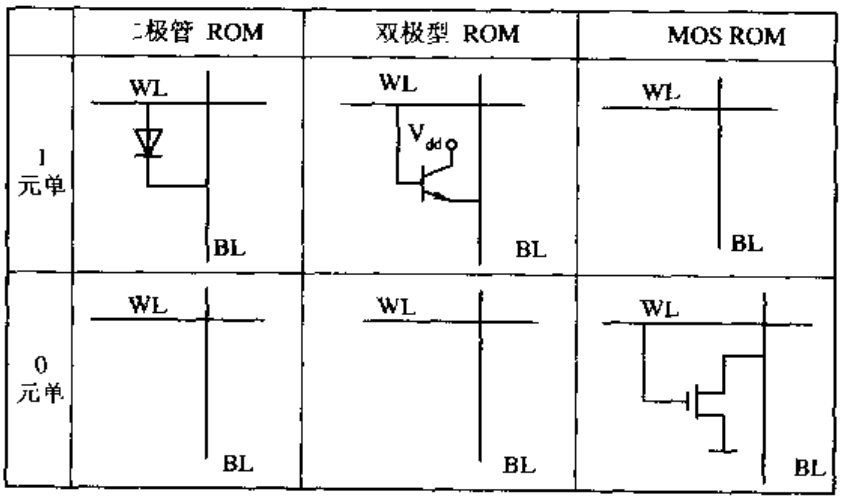


图 5.19 在不同工艺下表示 0 与 1 的 ROM 单元

对于二极管结构的 ROM 单元，如果字线 WL 与位线 BL 之间有一个二极管，一个高电压 U_{WL} 加到单元的字线上时，二极管被选通，位线被拉到 $U_{WL}-U_{D(on)}$，导致在位线上出现一个 1，这表示 ROM 单元存储了“1”。如果字线与位线之间没有任何物理连接，则位线通过电阻接到地上，与 WL 上的值无关，这就表示存储了一个“0”。也就是说，在 WL 与 BL 之间是否存在二极管，可以表示 ROM 单元存储了 1 还是 0。二极管单元的一个缺点是它没有将字线与位线分隔开来。电流要对位线电容进行充电。当存储器较大时，这个电容会非常大。因此，此方法只适用于较小的存储器。

ROM 单元也可以用一个双极型晶体管来取代，它的集电极连接到供电电压上。它的操作与二极管相似，但电容驱动能力较好。其不足之处是单元变得更为复杂，并且需要更大的面积。

尽管双极型晶体管 ROM 单元与二极管 ROM 单元相比，性能得到极大改善，但它还没有将字线与位线完全隔离开来。用 MOS 管来构造 ROM 单元，其隔离性更好，功耗更低，密度更高(但速度有所降低)。在这种结构中，当 WL 与 BL 之间没有晶体管时，表示存储了一个“1”。在位线、字线与地之间加入一个 MOS 管时，表示存储了一个“0”。

5.1.8 PROM

PROM(可编程 ROM)允许用户对存储器进行一次编程，因此称它为一次可写器件。它通常是通过在存储器单元中引入熔丝来实现的。在编程时，某些熔丝通过施加高电流来熔断，使得一些与其连接的晶体管不再起作用。

尽管 PROM 具有用户可编程的能力，但它只能进行一次编程，在编程过程中的一个错误会使器件无用，这很不方便。因此出现了可编程多次的器件。图 5.20 给出了 PROM 的结构。

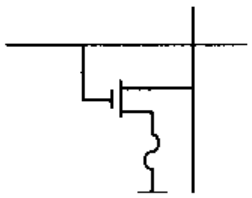


图 5.20 PROM 的结构

5.1.9 NVRWM

NVRWM(非易失性读写存储器)的结构与 ROM 的结构相似，都是由放于字线/位线间的晶体管阵列构成的。NVRWM 存储器包括 EPROM、EEPROM 和 FLASH MEMORY。EEPROM 结构的灵活性高，但密度与性能较差，EPROM 与 FLASH MEMORY 存储器件在密度与速度上都差不多，但 FLASH MEMORY 更为灵活，所以近年来它得到了广泛应用。

在这些结构中，晶体管的阈值是可以改变的。编程后，数据会一直保持(甚至掉电后也一样)。对 NVRWM 存储器进行重编程，旧的值必须被擦掉，然后开始新的编程。一般来说，NVRWM 存储器的编程比读操作一般要慢一个数量级。

1. EPROM

悬浮栅晶体管实际上是当今 NVRWM 存储器的核心。此结构类似于一个传统的 MOS 器件，其差异在于，在栅与通道之间有一个额外的多晶硅带。此带不与任何部分相连，因此称为悬浮栅。

在图 5.21 中，画出了悬浮栅的结构。

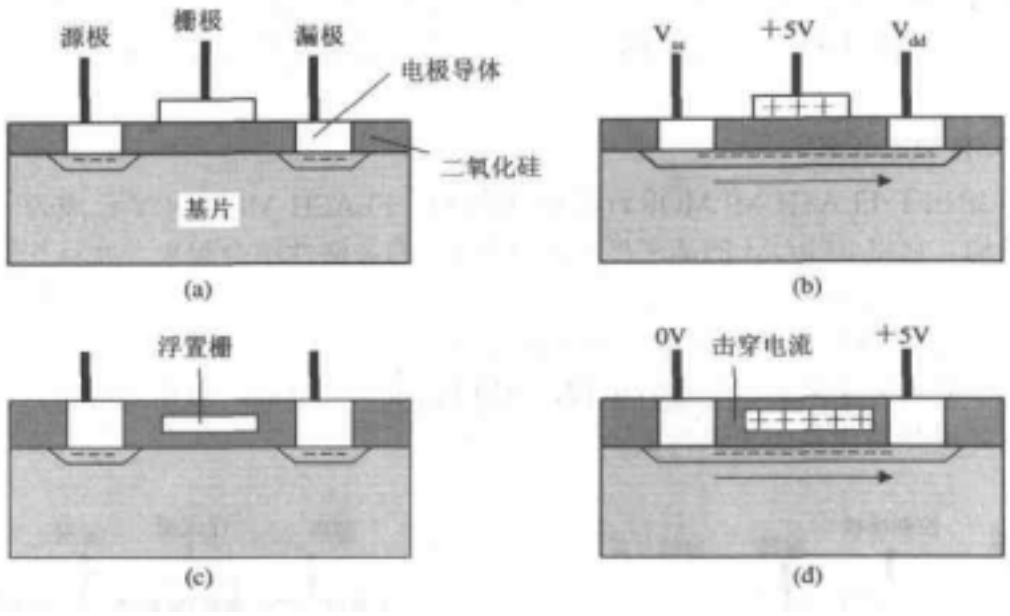


图 5.21　EPROM 与 MOS 对照

(a) MOS 晶体管结构；(b) MOS 晶体管导通状态；

(c) EPROM 晶体管结构；(d) EPROM 晶体管导通状态

由于悬浮栅的存在，器件的阈值电压是可编程的。当在源与栅一漏端加入一个范围在 15～20 V 的高电压时，会产生一个很高的电场，导致雪崩注入效应。电子获得了足够的能量，可穿过第一层氧化绝缘体，陷入悬浮栅中。将外加电压移出，悬浮栅上会留下负电荷，导致在中间栅上出现负电压。从器件的观点来看，这就增加了阈值电压。要开启此器件，需要加更高的电压。

悬浮栅周围是 SiO_2，SiO_2 是一种很好的绝缘体，悬浮栅上的电荷可以保持很多年，因此，它是一个非易失性存储器。几乎所有的非易失性存储器都基于悬浮栅方法，根据其擦除机理的不同，可以将它们分成 EPROM、EEPROM 和 FLASH MEMORY。

一个 EPROM 通过一个封装的透明窗来照射强紫外线以实现擦除。通过紫外线照射在材料中生成电子—空穴对，使得氧化物具有轻微的电导性。这种擦除过程很缓慢，需要几秒到几分钟(取决于紫外线源的强度)。此类器件的擦写次数不够多，并且不够可靠(当重复进行编程时，器件的阈值可能会发生变化)。大多数 EPROM 存储器都具有一些控制电路，以便在编程时将阈值控制在一个特定的范围。图 5.22 给出了 EPROM 的结构。

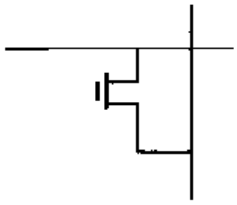


图 5.22　EPROM 的结构

2. EEPROM

图 5.23 给出了 EEPROM 单元结构。具有两个晶体管的 EEPROM 单元要大于 EPROM 单元。由于 FLOTOX 器件要大于 FAMOS 晶体管，所以，这种面积的耗费更多些，而且，制造非常薄的氧化物是很困难的，并且很昂贵。因此 EEPROM 组件要比 EPROM 昂贵。但

是，EEPROM 提供了更高的灵活性，并且更不易损坏，可以提供多达 1 万次的擦除写周期。重复编程时，由于在 SiO_2 上有永久滞留的电荷，会引起阈值电压的漂移，最终导致器件无法编程。

3. FLASH MEMORY

图 5.24 给出了 FLASH MEMORY(闪存)的结构。FLASH MEMORY 已成为一种很通用的存储器结构。它将 EPROM 的高密度与 EEPROM 的灵活性结合起来，并且其造价与功能介于两者之间。

FLASH MEMORY 是 EPROM 与 EEPROM 方法的结合。大多数 FLASH MEMORY 使用雪崩热电子注入方法来对器件进行编程，使用 Fowler-Nordheim 隧道效应进行擦除。它最主要的特点是擦除操作是成组进行的。

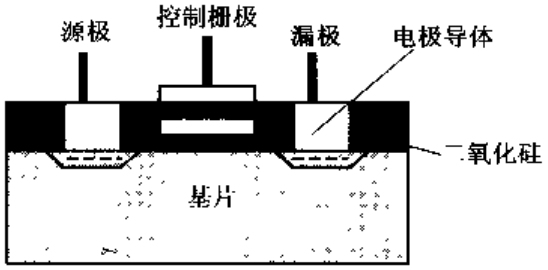


图 5.23 EEPROM 单元

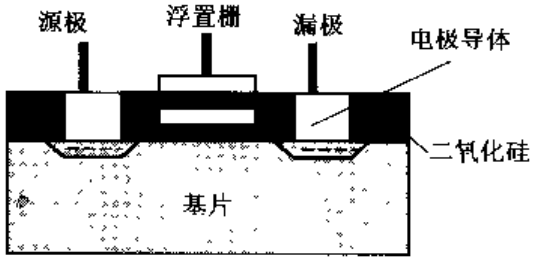


图 5.24 FLASH MEMORY 的结构

5.2 HSPICE 介绍

5.2.1 电路设计中常见的分析类型

1) 直流工作点(Bias Point Detail)分析

该分析可将电路中电容开路，电感短路，对各个信号取其直流电平值，然后用迭代的方法计算电路的直流偏置状态。其分析结果自动存入输出文件中。分析结果包括：各个节点电压、流过各个电压源的电流、总功耗、瞬态的初始条件，以及所有非线性受控源和半导体器件小信号下线性化模型参数。

2) 传输特性(Transfer Function)分析

该分析是计算直流工作点的，可在工作点对电路进行线性化处理，然后计算出线性化电路的小信号增益、输入电阻和输出电阻，并将结果自动存入输出文件中。

3) 直流特性扫描(DC Sweep)分析

该分析当电路中某一参数在一定范围内变化时，可对自变量的每一个取值，计算电路的直流偏置特性的相应变化曲线。它可将电容开路，电感短路，各个信号源取其直流电平值；将每个逻辑器件的延时取为 0，逻辑信号激励源取其 t=0 时进行计算。

4) 交流小信号频率特性/噪声(AC Sweep/Noise)分析

交流小信号频率特性分析是计算电路的交流小信号频率响应特性。首先计算电路的直

流工作点，并在工作点处对电路中各个非线性元件作线性化处理得到线性化的交流小信号等效电路。然后使电路中交流信号源的频率在一定范围内变化，并利用交流小信号等效电路计算电路输出交流信号的变化(即电路的幅频与相频响应)。

噪声分析的计算方法是选定一个节点作为输出节点，将每个电阻和半导体器件噪声源在该节点处产生的噪声电压均方根值迭加。然后选定一个独立电压源或独立电流源，计算电路中从该独立电压源或独立电流源到上述输出节点处的增益，再将计算得到的输出节点处总噪声除以该增益，就得到在该独立电压源(独立电流源)处的等效噪声。

5) 瞬态特性(Transient)分析

该分析是指在给定信号作用下，计算电路输出端的瞬态效用。进行瞬态分析时，首先计算 t=0 时的电路初始状态，然后从 t=0 到某一给定的时间范围内选取一定的时间步长，计算输出端在不同时刻的输出电平。(电路初始状也可以根据要求设定。)

6) 傅里叶(Fourier)分析

该分析是指瞬态分析完成后，通过傅里叶积分，计算瞬态分析输出结果波形的直流、基波和各次谐波分量。

7) 参数扫描(Parametric)分析

该分析是针对电路参数发生变化来分析电路特性的变化。它可以与任何一种分析类型配合起来使用，对电路所执行的分析进行参数扫描。

8) 温度(Temperature)分析

该分析是分析某一特定温度下电路特性的变化。若不特别指明值，元器件参数和模型参数都假定是其常温下的值。

9) 灵敏度(Sensitivity)分析

该分析是分析指定的节点电压对电路中电阻、独立电压源和独立电流源、电压控制开关和电流控制开关、二极管、双极型晶体管等元器件参数的敏感度，并将结果自动存入输出文件中。

灵敏度用两种方式表示：一个是元件灵敏度 S，指电路特性参数 T 对元器件值绝对变化的灵敏度；另一个是相对灵敏度 S_N，指电路特性参数 T 对元器件值 X 相对变化为本%情况下的灵敏度。其中：

$$S(T,X) = \partial T / \partial X$$

$$S_N = X \cdot S(T,X) / 100$$

10) 蒙特卡罗/最坏情况(Monte Carlo/Worst Case)分析

蒙特卡罗分析是根据实际情况确定元器件值的分布规律的，每次分析时采用的元器件值都从元器件值分布规律中随机抽样。完成了多次电路特性分析后，对各次分析结果进行综合统计分析，就可得到电路特性的分散变化规律。

最坏情况分析就是按引起电路特性向同一方向变化的要求，确定每个元器件的变化方向，然后再使这些元器件同时在相应方向按其可能的最大范围变化。估算出电路性能相对标称值时的最大偏差。

5.2.2 HSPICE 基础知识

HSPICE 是一个应用很广的电路仿真程序，可以完成集成电路的稳态分析、瞬态分析和频域分析等功能。它包含了许多 Foundry 模型参数。它还能对 PCB、多芯片系统、封装以及 IC 技术中连线间的几何损耗加以模拟，是 ASIC 设计中必不可少的工具。

表 5.2 给出了 HSPICE 中文件的后缀。

表 5.2 HSPICE 中文件的后缀

文件类别	文件的内容	文件后缀
输入文件	配制文件	Meta.cfg
	初始化文件	Hspice.ini
	直流工作点初始化文件	<design>.ic
	输入网表文件	<design>.sp
	库输入文件	<library_name>
	模拟转移数据文件	<design>.d2a
输出文件	输出列表	.lis 或由用户自己定义
	瞬态分析结果	.tr**
	瞬态分析测量结果	.mt**
	直流分析结果	.sw**
	直流分析测量结果	.ms**
	交流分析结果	.ac**
	交流分析测量结果	.ma**
	硬拷贝图形数据	.gr**
	数字输出	.a2d
	FFT 分析图形数据	.ft**
	子电路交叉列表	.pa**
	输出状态	.st**

要进行 HSPICE 分析，必须给出输入网表文件。输入网表文件的后缀为.sp。输入网表文件中第一行语句是标题行(该行不被编译)，最后一个语句必须是.END 语句。例如：

```
Inverter Circuit
.OPTIONS LIST NODE POST
.TRAN 200P 20N SWEEP TEMP -55 75 10
.PRINT TRAN V(IN) V(OUT)
M1 VCC IN OUT VCC PCH L = 1U W = 20U
M2 OUT IN 0 0 NCH L = 1U W = 20U
VCC VCC 0 5
VIN IN 0 0 PULSE .2 4.8 2N 1N 1N 5N 20N CLOAD OUT 0 .75P
.MODEL PCH PMOS
```

```
.MODEL NCH NMOS
.ALTER
CLOAD OUT 0 1.5P
.END
```

输出列表文件包含了由输入列表文件中的.PLOT、.PRINT 以及分析语句指定的模拟结果，后缀为.lis。

一个电路的描述语句如表 5.3 所示，其分析和控制语句如表 5.4 所示。

表 5.3　电路的描述语句

输入描述语句和规定	标题语句	第一行
	结束语句	最后一行
	.GlOBAL	.GLOBAL　node1　node2　node3…… 输入文件若定义了.GLOBAL 语句，则输入文件所有子电路中与.GLOBAL 节点名相同的节点都被自动定义成有连接关系。一般线路的电源、地被定义成.GLOBAL 语句
	注释语句	一般以*号开头，是用户对程序运算和分析时加以说明的语句。在列出输入程序时会打印出来，但不参与模拟分析。该语句可放在输入文件标题语句以后的任意位置加以注释
元件描述语句		元件语句一般由元件名、元件所连接的电路节点号和元件参数值组成
电源描述语句		SPICE 中提供了一些供激励用的独立源和受控源，电源描述语句也由代表电源名称的关键字、连接情况和有关参数值组成。描述电源的关键字含义为： V:　独立电压源　　I:　独立电流源 E:　电压控制电压源　　F:　电流控制电流源 G:　电压控制电流源　　H:　电流控制电压源
半导体器件描述语句		包括晶体二极管、双极型晶体三极管、结型场效应管或 MESFET、MOS 场效应管
子电路描述语句		包括子电路定义开始语句、子电路终止语句、子电路调用语句
模型描述语句		.MODEL 语句
库文件调用及定义语句		.LIB 语句

表 5.4　电路分析和控制语句

电路性能分析语句		包括直流工作分析语句、交流分析语句、瞬态分析语句
设置初始状态语句		包括初始条件语句.IC 和.DCVOLT、节点电压设置语句.NODESET
统计分析与参数变化分析		包括蒙特卡罗分析.MC、最坏情况分析、温度特性分析.TEMP、参数及表达式定义语句.PARAM
输入控制语句	ALTER 语句	.ALTER 语句功能是针对设定的不同参数和数据自动进行更替来进行电路的模拟的
	.DATA 语句	.DATA 语句针对每一个模拟过程期间，为需要改变的参数提供了一种简便的改变参数并给出数值设置的有效方法。
	任选项语句 .OPTIONS	任选项语句是为了满足用户的需要或特殊的模拟目的，允许用户重新设置程序的参数或控制程序的功能

续表

输出控制语句	打印语句.PRINT	.PRINT 语句规定了要输出打印的变量值。
	打印宽度语句 .WIDTH	.WIDTH 语句决定了打印输出的宽度。一般形式： .WIDTH OUT={80|132} 其中，OUT 表示输出打印的宽度，可设置 80 或 132 两种
	绘图语句 .PLOT	.PLOT 语句能对某种选定分析的结果进行绘图输出，在一个绘图输出中可以多达 32 个变量
	探针显示语句 .PROBE	.PROBE 语句能把输出变量存储到接口文件和图形数据文件中。该语句规定了哪些参数将在输出列表中被打印出来
	图形语句 .GRAPH	.GRAPH 语句产生一个高分辨率的输出绘图结果。这个语句与一个附加了一个可选模型的.PLOT 语句功能一样
	图形显示语句 .OPTION POST	.OPTION POST 语句用于在终端上显示模拟结果的高分辨率图形曲线。POST 的缺省值提供了许多有用的参数。
	统计运行过程语句 .OPTION ACCT	.OPTION ACCT 语句在电路模拟过程结束后，提供了一个运行过程的统计结果

图 5.25 和图 5.26 分别给出了 p-n 结二极管和 MOS 管的 SPICE 模型。

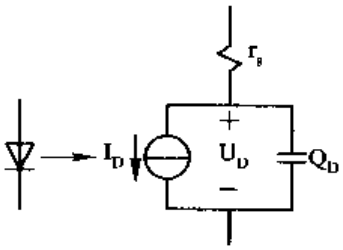


图 5.25 p-n 结二极管的 SPICE 模型

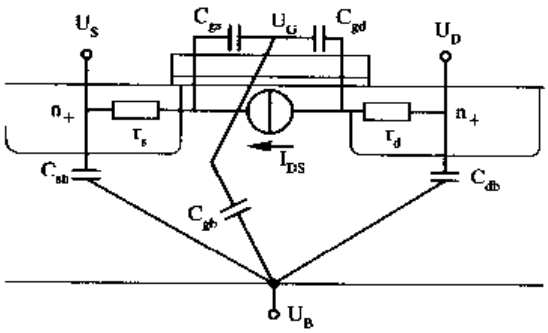


图 5.26 MOS 管的 SPICE 模型

5.3 存储器设计

在数字设计中，存储器的设计是一个很重要的方面。存储器中，重要的时间参数包括读取时间(读请求与数据在输出端输出之间的延时)、写时间(写请求与最终将输入数据写入存储器之间的时间)。另一个重要的参数是存储器的(读或写)周期。

用寄存器作为存储介质，它的速度之快会让设计者心情舒畅，但这是一头吞噬硅片面积的怪兽。所以，人们不得不采用其它途径，例如 DRAM。通常的做法是，将存储单元组合形成大的阵列结构，这样可以使由外围电路引起的过载最小化，并增加存储密度。

在存储器设计中，通过牺牲某些数字电路的性能指标来减少单元面积(例如噪声容限、逻辑摆幅、输入输出分离、扇出，或者速度等)。在存储器核中可以通过降低一些性能指标以换取高密度，但在与其它电路交互时它必须具有良好的信号特性，所以必须用外围电路来进行恢复。

例如，减小位线上的电压幅度，即减少了传播延时，也降低了功耗。这种改动带来的性能恶化是可以忍受的(在存储器单元中，对串扰及其它干扰进行控制，可以保证足够的噪声容限)。但是，存储器与其它组件交互时，电压幅度必须足够大，可以通过读出放大器来放大信号。

较大的存储器都采用层次化的结构。存储器被划分为多个更小的模块。一个额外的地址字，被称为块地址选择，用来选择多个模块中的一个进行读或写。此方法有两个优点：

- 局部的位线长度和字线的长度被控制在一定的范围内，因而存取时间更快。
- 不被激活的模块处于功耗节省模块，其读出放大器、行列译码器都不工作。这就减少了功耗，而功耗是大存储器一个很重要的性能指标。

许多存储器单元只提供一个唯一的端口，它在输入与输出间进行共享。具有更高带宽要求的存储器通常有多个输入与输出端口，例如，用于 RISC 微处理器中的寄存器阵列。增加更多的端口会使存储单元的设计更复杂。下面给出一个五端口寄存器阵列的设计。

1. **设计要求**

(1) 容量为 32×32 bit。

(2) 有三个读端口、两个写端口，每个端口独立工作，可同时进行操作。

(3) 微处理器提供工作时钟，频率为 50 MHz。

(4) 电源电压为 5 V。

2. **读写时序设计**

当需要写操作时，CPU 系统提供写地址、写数据及写使能信号。在系统时钟上升沿来到时，Register File 检测写使能信号是否有效。若有效，则将输入数据写入相应地址单元。

当需要读操作时，CPU 系统将向 Register File 提供读地址、读使能信号。在系统时钟下降沿来到时，Register File 检测读使能信号是否有效。若有效，则将相应地址的内容送到数据输出端口。系统要求在时钟下降沿后 5 ns 即获得稳定的输出数据，该数据要能保持 10 ns 以上，以便 CPU 系统在下一个上升沿到达时能读到有效数据。

写、读操作分别在时钟的上升沿、下降沿进行，这样的设计可以避免多口读写的读写端口竞争问题。读写端口竞争是指当写端口和读端口的地址相同时，应该使读端口的输出数据为此时正待写入的新数据。一般为了解决这种地址冲突，在设计中加入旁路电路。旁路电路将对读端口和写端口上的地址进行比较，若发现相同，则写入通道的数据直接送到输出数据端口上。旁路电路包含地址比较器以及输出数据选择器。对于我们设计的五口(三读两写)32×32Register File 来说，若用旁路的方法，则需要 6 套 5 位地址比较器、3 套 32 位的三选一选择器，这样将增大硬件开销。因此，可以采用上面提到的不同沿触发读写的方法来处理读写端口竞争问题。在上升沿进行写操作，在下降沿进行读操作，这样先写后读的时序就可以使同一地址的读端口得到最新写入的数据。当然，对于不同沿触发的读写时序来说，在速度上的要求比用旁路电路的设计要高得多。要在保证速度的前提下进行结构的优化。

读写时序要求如图 5.27 所示。其中，t_{aw} 为时钟上升沿到数据写入的时间；t_{aa} 为时钟下降沿到数据输出的时间；t_{dh} 为输出数据保持时间。系统提供 50 MHz 时钟，因而 $t_{ckh}=t_{ckl}=$ 10 ns。Register File 的读写性能要求为 $t_{aw}<10$ ns，$t_{aa}<5$ ns，$t_{dh}>10$ ns。

3. 电路总体结构

图 5.28 是 32×32 五口 Register File 电路的整体结构示意。

该电路的核心是 32×32 bit 的存储阵列，它由 32 行、32 列组成，共有 1024 个存储单元。每个单元存储 1 bit 的数据信息，且与存储体外部五个数据通道相连，以支持五口读写的设计。存储体的外围电路是五个数据通道(其中，三个数据通道支持读操作，二个支持写操作)，以及控制电路。写通道电路包括写地址、写使能、写数据的输入寄存器以及写地址译码器。读通道电路包括读地址、读使能的输入寄存器、读地址译码器、读灵敏放大器以及读出数据缓冲电路。控制电路监测时钟信号的沿变化以及各个使能信号，从而产生一系列时序控制信号，它们将控制存储体以及外围各部分电路。

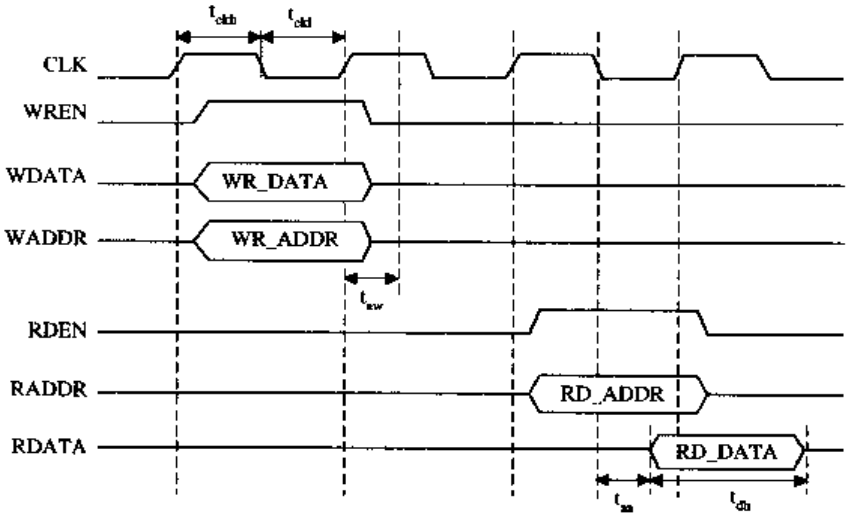


图 5.27　读写时序要求

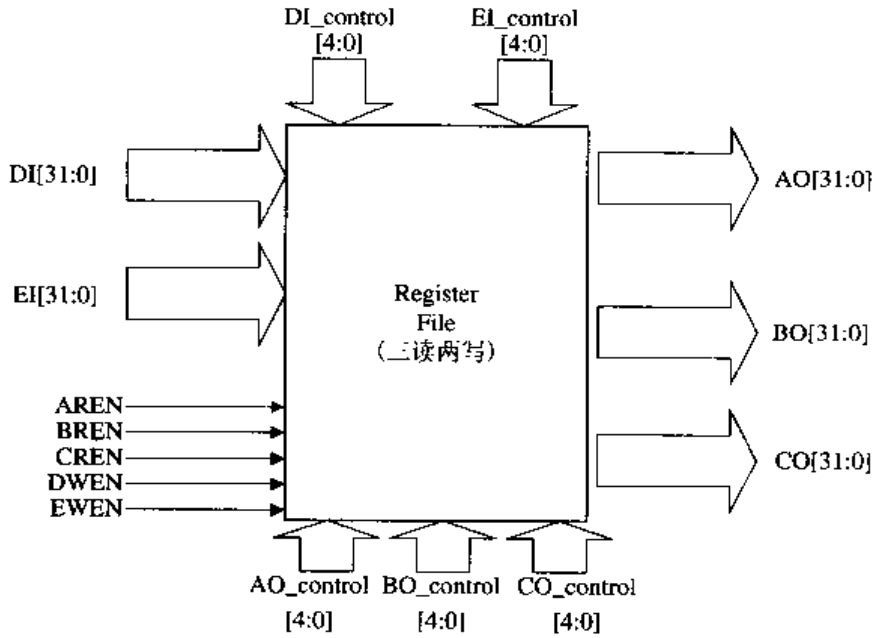


图 5.28　32×32 五口 Register File 电路的整体结构

从功能上讲，该电路包括三个部分：地址译码单元、存储单元、输出单元。

1) 地址译码单元

地址译码单元采用组合逻辑，经组合逻辑译码，选中的单元将被写入或读出 32 位字。地址译码采用两级译码：地址的低 3 位进行行译码(Row Decode)，地址的高 2 位与行译码的结果及 enable 信号进行列译码(Column Decode)。

2) 存储单元

五口存储单元的结构如图 5.29 所示。

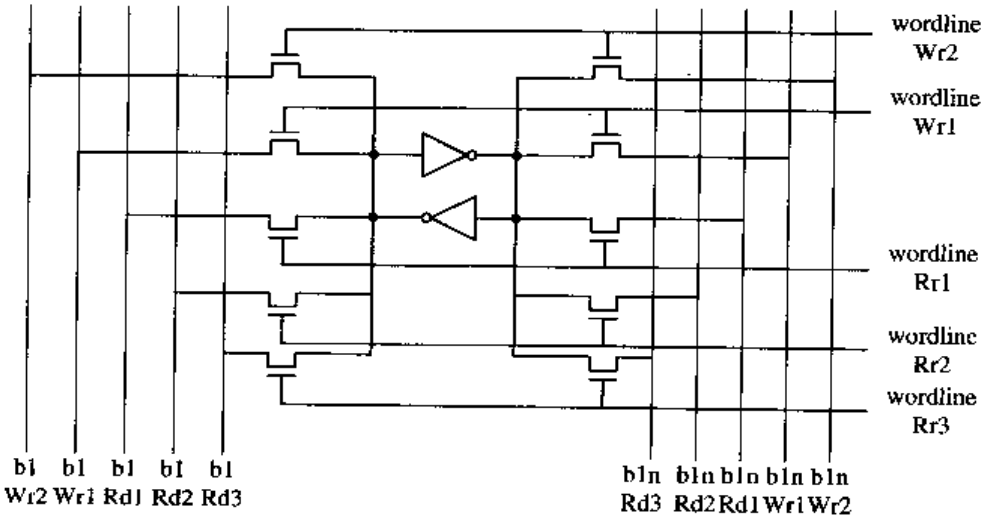


图 5.29　5 口存储单元电路

多口 Register File 存储单元的结构与单口 SRAM 单元类似。它的核心也是两个交叉耦合的反相器构成的双稳态触发器。它与单口 SRAM 不同的地方是，门管数目增多。这是因为每次单元内部和存储体外围电路交换数据都是通过对称的两个门管进行的。存储体的端口数目越多，需要的门管数也越多。我们设计的是五口存储体，因此需要五套对称结构的门管，即 10 个门管。这样，每套门管可以独立支持一路数据的交换。

多口存储单元除了这种对称式(Differential)的结构外，还可以采用单线结构(Single_end)。单线结构的每个端口只用一个门管和单根位线，因此省面积，但它的取数速度却比对称双线式的结构慢(对称式的电路结构可以用差分的方法，经高速双端灵敏放大器作用，来获得较快的读取速度)。这里，我们设计的 Register File 的存储容量较小，因而速度要求较高，故选择了对称性的多口单元结构以保证速度。

说明：

增大门管的 W/L，可提高位线放电速率；同时，为了增大噪声容限，又需要降低门管的 W/L。因此，存在一个速度与稳定性的平衡问题。实际上，一味地增大门管 W/L，将导致读出时单元存储数据翻转；而一味地减小门管 W/L，将导致不能正确地写入信息。因此在设计单元参数时，为了避免读翻转，应保证工作管的 W/L 比门管的 W/L 大几倍；而为了避免写失效，应保证门管的 W/L 比负载管的 W/L 大几倍。另外，在保证性能的前提下，为了减小单元面积应尽量减小各管的 W 和 L。

3) 输出单元

输出单元的作用是增强输出驱动和减少输出的时间。输出采用三态。

4) 外围电路设计

存储器核通过降低性能和可靠性来得到更少的面积，存储器的设计要依赖大量的外围电路来恢复性能。由于存储器核的设计主要是由工艺所决定的，通常不是电路设计者所能决定的，而外围电路，给设计者提供了很大的设计余地。

在32×32的存储阵列外，有寄存器、译码器、灵敏放大器、输入输出缓冲及控制电路等。写地址、写数据和写使能信号在时钟CLK的上升沿锁存，锁存后的写地址经5-32的译码器译码后将获得写行选信号，锁存后的写数据将通过输入缓冲电路产生具有高驱动能力的数据信号。读地址和读使能信号将在CLK的下降沿被锁存，锁存后的读地址经译码后获得读行选信号；读出的数据经过灵敏放大器放大，再经输出缓冲后送到输出数据总线上。内部控制电路将对各部分的时序进行控制。这部分电路的设计在这里略去。

5.4 练　　习

1．设有32片256 K×1位的SRAM芯片，问：

(1) 采用位扩展方法可构成多大容量的存储器？

(2) 该存储器需要多少字节地址位？

(3) 画出该存储器与CPU连接的结构图，设CPU的接口信号有地址信号、数据信号、控制信号MREQ#和R/W#。

2．用HSPICE实现D触发器。

3．用HSPICE完成5.3节的寄存器阵列。

第6章 图像与视频芯片的设计

图像与视频是两个既有联系又有区别的概念：静止的图片称为图像(Image)，运动的图像称为视频(Video)。图像的输入要靠扫描仪、数字照相机或摄像机来完成；而视频的输入只能靠摄像机、录像机、影碟机以及电视接收机等可以输出连续图像信号的设备来完成。

许多视频处理系统同时具有图像处理功能，因此有时不再严格区分视频与图像处理硬件。

图像视频硬件是发展最迅速的领域。各种采集卡、字幕叠加卡、压缩/解压缩卡、电影卡、电视卡层出不穷。这些硬件是多媒体视频会议、可视电话、视频邮件、多媒体通信终端等的基础。随着基于宽带多媒体通信网络的交互式电视(ITV)、点播电视(VOD)以及远程教育系统、远程医疗诊断系统、远程电子图书馆等新技术的付诸实施，视频处理芯片的需求会越来越多。

相关的标准也很多，例如 MPEG-2、H.261 和 JPEG 等。下面介绍几种动/静态图像的压缩标准。

静态图像压缩技术主要是对空间信息进行压缩的，而对动态图像来说，除对空间信息进行压缩外，还要对时间信息进行压缩。目前已形成三种压缩标准：

(1) JPEG(Joint Photographic Experts Group)标准：用于连续色调，多级灰度、彩色/单色静态图像压缩。具有较高压缩比的图形文件，在压缩过程中的失真很小。目前使用范围广泛(特别是 Internet 网页中)。动态 JPEG 可顺序地对视频的每一帧进行压缩，就像每一帧都是独立的图像一样。动态 JPEG 能产生高质量、全屏、全运动的视频，需要依赖附加的硬件。

(2) H.261 标准：适用于视频电话和视频电视会议。

(3) MPEG(Motion Picture Experts Group——全球影像/声音/系统压缩标准)标准：包括 MPEG 视频、MPEG 音频和 MPEG 系统(视音频同步)三个部分。MPEG 压缩标准是针对运动图像而设计的。基本方法是：在单位时间内采集并保存第一帧信息，然后就只存储其余帧相对第一帧发生变化的部分，以达到压缩的目的。MPEG 压缩标准可实现帧之间的压缩，其平均压缩比可达 50：1，压缩率比较高，且又有统一的格式，兼容性好。

MPEG-1 用于传输 1.5 Mb/s 数据传输率的数字存储媒体运动图像及其伴音的编码，视频数据压缩率为 1/100～1/200，音频压缩率为 1/6.5。MPEG-1 提供 30 帧 352×240 每秒分辨率的图像。VCD 采用的就是 MPEG-1 的标准，该标准是一个面向家庭电视质量级的视频、音频压缩标准。

MPEG-2 主要针对高清晰度电视(HDTV)的需要，传输速率为 10 Mb/s，与 MPEG-1 兼容，适用于 1.5～60 Mb/s 甚至更高的编码范围。MPEG-2 有 30 帧 704×480 每秒的分辨率，

是 MPEG-1 的四倍。它适用于高要求的广播和娱乐应用程序，如 DSS 卫星广播和 DVD。

MPEG 4 标准是超低码率运动图像和语言的压缩标准，用于传输速率低于 64 Mb/s 的实时图像传输，它不仅可覆盖低频带，也可向高频带发展。

6.1 色度空间转换器

6.1.1 亮度信号和色差信号

R，G，B 分量分别代表三维色度空间中的“红”、“绿”和“蓝”三个基本颜色。在 YUV 色度空间中，Y 分量代表“黑白”信号的亮度。“色度差”信号 B-Y(Cb)和 R-Y(Cr)补充了 Y 信号与相应色度的差别。

根据三基色原理，在视频领域利用 R(红)、G(绿)、B(蓝)三色不同比例的混合来表现丰富多采的现实世界。

首先，通过摄像机的光敏器件如 CCD(电荷耦合器件)，将光信号转换成 RGB 三路电信号。其次，在电视机或监视器内部也使用 RGB 信号分别控制三支电子枪轰击荧光屏以产生图像。这样，由于摄像机中原始信号和电视机、监视器中的最终信号都是 RGB 信号，因此直接使用 RGB 信号作为视频信号的传输和记录方式会获得极高的信号质量。但这样做会加宽视频带宽，从而增加设备成本，且这也与现行黑白电视不兼容，

因此，在实际应用中，RGB 信号按亮度方程 Y=0.299R+0.587G+0.114B(PAL 制)转换成亮度信号 Y 和两个色差信号 Cb、Cr，形成 YCbCr 分量信号。此种信号利用人眼对亮度细节分辨率高而对色度细节分辨率低的特点，对 Cb、Cr 信号带宽进行压缩。Cb、Cr 信号还可进一步合成一个色度信号 C，进而形成 Y/C 记录方式。由于记录时对 C 信号采取降频处理，因此也称彩色降频方式。

Y 和 C 又可进一步形成复合视频(Composite)，即彩色全电视信号，这种方式便于电视信号的传输和发射。将 RGB 信号转换成 YCbCr 信号、Y/C 信号直至 Composite 信号的过程称为编码，逆过程则为解码。

由此可看出，由于转换步骤的减少，视频输出质量由 YCbCr 端口到 Y/C 端口到 Composite 端口依次降低。因此，在视频捕捉或输出时选择合适的输入、输出端口可提高视频质量。另外，还应提供同步信号以保证传送图像稳定再现。

信号是经过矩阵变换得到的。

$Y = 0.299*R + 0.587*G + 0.114*B$

$Cb = 0.565(B-Y) = -0.169*R - 0.332*G + 0.500*B$

$Cr = 0.713(R-Y) = 0.500*R - 0.419*G - 0.0813*B$

从 RGB 色度空间到 YCbCr 色度空间的转换，应用非常广泛，可应用在不同的视频压缩/处理算法中，例如，JPEG，MPEG1/2，MJPEG 等。

同样，YCbCr 色度空间到 RGB 色度空间的转换也是经过矩阵变换得到的。

$R = Y + 1.403Cr$

$G = Y - 0.344Cb - 0.714Cr$

$$B = Y + 1.770Cb$$

6.1.2 RGB-YCbCr 的模块设计

1. 功能描述

这里实现一个带同步时钟的色度空间转换模块，由 RGB 转到 YCbCr。RGB 及 YcbCr 均是 10 位宽度。输入输出均为无符号整数格式。

2. 设计思路

转换器设计思路比较直观：利用转换公式，使用三级流水结构，首先将 R、G、B 与系数值求积，乘积结果为 20 位宽，第二步将结果相加，第三步将得到的结果截为 10 位，范围限在 0～1023。

其数据流如图 6.1 所示。

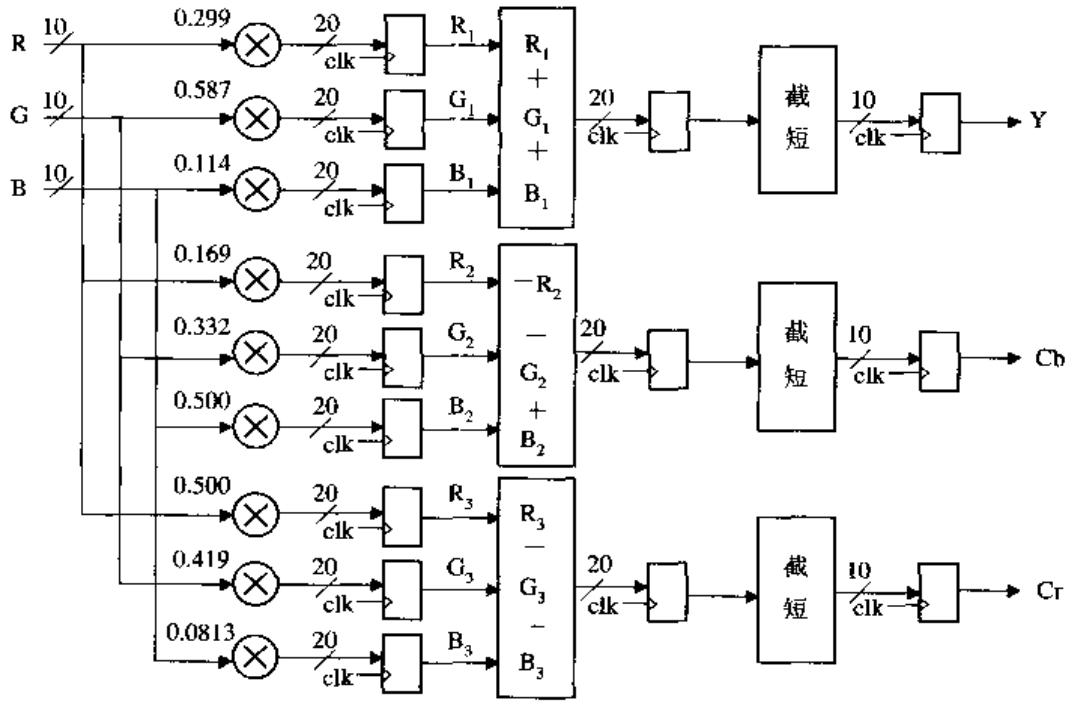


图 6.1 RGB-YCbCr 转换模块数据流图

6.1.3 YCbCr-RGB 的模块设计

1. 功能描述

这里实现一个带同步时钟的色度空间转换模块，由 YCbCr 转到 RGB。RGB 及 YcbCr 均是 10 位宽度。输入输出均为无符号整数格式。

2. 设计思路

与 RGB-YCbCr 转换模块相似，YCbCr-RGB 为三级流水结构。其数据流如图 6.2 所示。

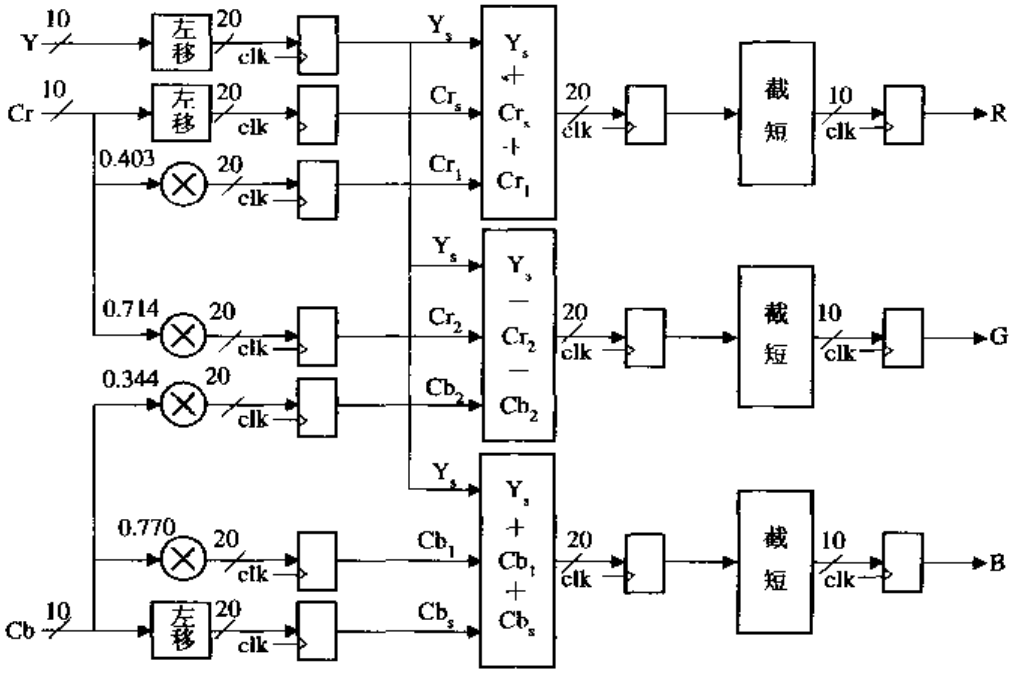


图 6.2　YCbCr-RGB 转换模块数据流图

6.2　DCT(离散余弦变换)

6.2.1　DCT 原理

在视听产品领域，人们总是过于贪婪：一方面要求更大的解析度、更丰富的音质，同时又要求更快的响应速度。无压缩情形下传输音频、视频流只是妄想。尽管有各种各样的技术来增加带宽，但对于视频与音频应用来说，带宽总是不够用的。视频与音频数据流包含的比特非常庞大，轻而易举就会把带宽占满，所以种种压缩方法应运而生。

任何压缩方法都有赖于数据的相关性。在一幅图像中格点通常是与高度相关的。因此可以用可逆转换将随机数转换为更少的、不相关的参数。离散余弦变换是将信号在频域分解，再更进一步地处理，以进行压缩(然后再用反余弦变换恢复)。

下面说明离散余弦变换的原理。

一维离散余弦变换公式如下：

$$X(u)=\sqrt{\frac{2}{N}}C(u)\sum_{i=0}^{N-1}x(i)\cos\left(\frac{(2i+1)u\pi}{2N}\right)$$

这里，$C(0)=1/\sqrt{2}$，$C(u)=1$，$u\neq 0$。

二维离散余弦变换公式如下：

$$X(u,v)=\frac{2}{N}C(u)C(v)\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}x(i,j)\cos\left(\frac{(2i+1)u\pi}{2N}\right)\cos\left(\frac{(2j+1)v\pi}{2N}\right)$$

这里，$C(0)=1/\sqrt{2}$，$C(u)=C(v)=1$，$u,v\neq 0$。

二维离散反余弦变换公式如下：

$$x(i,j)=\frac{2}{N}\sum_{u=0}^{N-1}\sum_{v=0}^{N-1}C(u)C(v)X(u,v)\cos\left(\frac{(2i+1)u\pi}{2N}\right)\cos\left(\frac{(2j+1)v\pi}{2N}\right)$$

6.2.2 DCT 模块设计

为节省篇幅，这里仅仅给出 DCT 模块的设计思想。

从以上公式中可以看出，DCT 变换和反向 DCT 变换的实现是很类似的，只是系数不同。因此，对于 DCT 模块，只需改变系数表就可以完成反 DCT 变换。

DCT 模块框图如图 6.3 所示，其端口说明如表 6.1 所示。

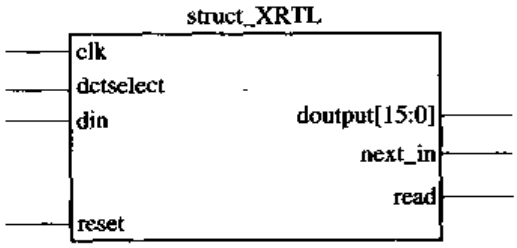


图 6.3 DCT 模块的框图

表 6.1 DCT 模块的端口说明

输入	端口信号说明
clk	时钟
dctselect	选择是前向或是反向 DCT "1"：正向 DCT；"0"：反向 DCT
din	串行时钟输入
reset	同步复位
输出	**端口信号说明**
doutput[15:0]	数据输出端口(并行输出)
next_in	触发输入下一个字
read	触发读下一个输出的字

DCT 系数表可采用 case 语句用查找表结构实现。一种实现代码如下所示，这里实现的是 8×8 的 DCT 变换，即公式中的 N 为 8。

```
function [31:0] dct_ table;
    input [2:0] i,j,u,v;        // 均在 000~111 之间
begin
    // DCT 函数：cos( (2i +1) * u * pi)/16) * cos( (2j +1) * v * pi)/16)
    case ( {v,u} )
        6'h00:
            case ( {j,i} )
                6'h00: dct_table = 32'h20000000;
                6'h01: dct_table = 32'h20000000;
```

```
                6'h02: dct_table = 32'h20000000;
                6'h03: dct_table = 32'h20000000;
                6'h04: dct_table = 32'h20000000;
                6'h05: dct_table = 32'h20000000;
                6'h06: dct_table = 32'h20000000;
                6'h07: dct_table = 32'h20000000;
                6'h08: dct_table = 32'h20000000;
                ......
                6'h3f: dct_table = 32'h20000000;
            endcase
        6'h02:
        ......
         6'h3f:
            case ( {j,i} )
                6'h00: dct_table = 32'h026f9430;
                6'h01: dct_table = 32'hf9103298;
                ......
            endcase
    endcase
end
endfunction
```

DCT 变换的实现过程为：串行的数据首先放到输入缓冲中(采用环形寄存器来构造)，然后进行求积、求和运算，最终以并行数据的形式输出(和滤波器有些类似)。这些操作要在控制模块下完成，以保证时序的正确。

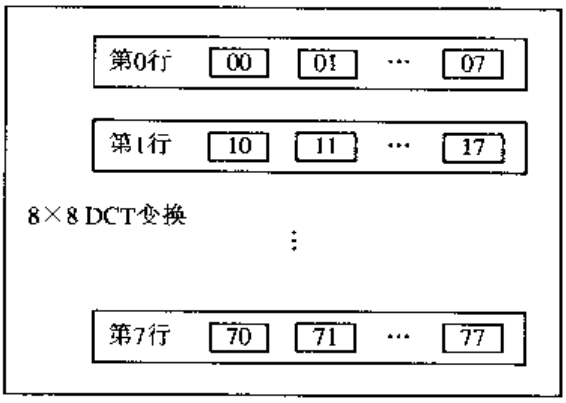


图 6.4　DCT 变换模块框图

DCT 变换模块的框图如图 6.4 所示。

8×8DCT 变换实际上就是 64 个像素点的并行运算。

对每个像素点所作的运算为：输入数据和 DCT 系数相乘，而后相加得到最后结果。流程示意图如图 6.5 所示。

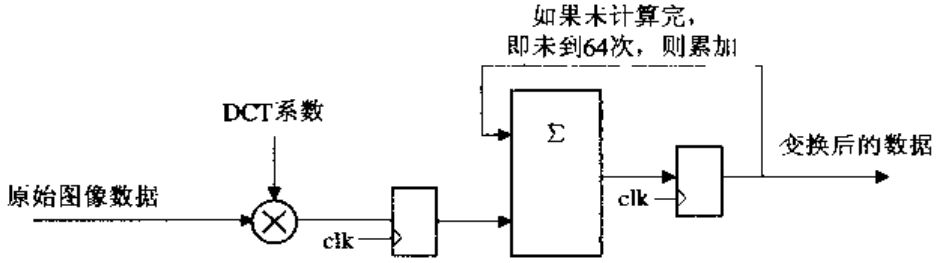


图 6.5　DCT 变换单个像素点的运算流程图

由公式可知，需要计算 8×8=64 次，每计算一次后，将 i、j、u、v 作相应变化，并重新在系数表中找到相应的系数，和新的数据相乘，作下一次运算。

原始图像数据是串行输入的，每个时钟输入一个像素点的数据。DCT 系数是通过 DCT 系数表查找到的，随 i、j、u、v 的不同而不同。采用先相乘，后累加的结构得到 DCT 变换的结果。这里采用了两级流水结构。

6.3 zigzag 扫 描

6.3.1 zigzag 概念

DCT 系数是二维数据，需要将二维数据按一定的次序排成一维数据，才能进行后面的游程编码。有一种有效的排序方法被称为 zigzag 扫描法，即 z 字形转弯，如图 6.6 所示。由于 DCT 变换后高频系数较小，且集中在块的右下角，利用 zigzag 扫描方法可以使块中 0 系数连续的长度比较长，提高后面游程编码的效率。

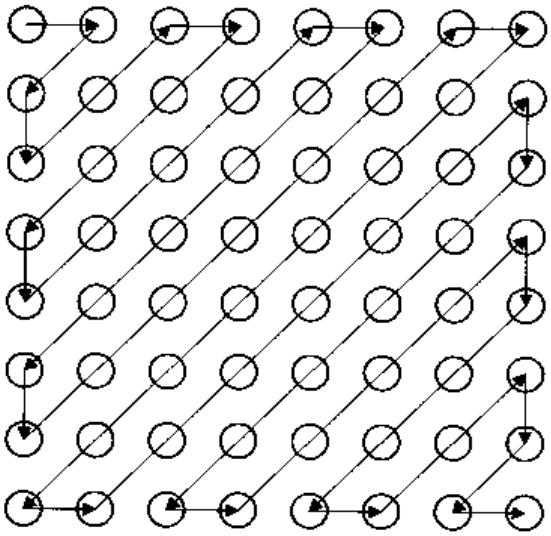


图 6.6 zigzag 扫描次序示意图

6.3.2 zigzag 模块设计

zigzag 模块的设计比较简单，将输入的二维图像数据按图 6.6 的次序赋给输出数据，然后利用移位寄存器将块图像数据从最高位到最低位依序移出，即成为 zigzag 扫描后的一维数据序列。

表 6.2 为针对 8×8 图像块输入和输出数据的一种匹配方法。

表 6.2　zigzag 扫描中输入和输出数据的匹配

输入数据	输出数据
00, 01, 02, 03, 04, 05, 06, 07,	63, 62, 58, 57, 49, 48, 36, 35,
10, 11, 12, 13, 14, 15, 16, 17,	61, 59, 56, 50, 47, 37, 34, 21,
20, 21, 22, 23, 24, 25, 26, 27,	60, 55, 51, 46, 38, 33, 22, 20,
30, 31, 32, 33, 34, 35, 36, 37,	54, 52, 45, 39, 32, 23, 19, 10,
40, 41, 42, 43, 44, 45, 46, 47,	53, 44, 40, 31, 24, 18, 11, 09,
50, 51, 52, 53, 54, 55, 56, 57,	43, 41, 30, 25, 17, 12, 08, 03,
60, 61, 62, 63, 64, 65, 66, 67,	42, 29, 26, 16, 13, 07, 04, 02,
70, 71, 72, 73, 74, 75, 76, 77,	28, 27, 15, 14, 06, 05, 01, 00,

用下面的代码可以实现这种匹配。

```
always @(posedge clk)
    ......
    begin
        zigzag[63] <= #1 din_00;
        zigzag[62] <= #1 din_01;
        zigzag[61] <= #1 din_10;
        zigzag[60] <= #1 din_20;
        zigzag[59] <= #1 din_11;
        zigzag[58] <= #1 din_02;
        zigzag[57] <= #1 din_03;
            ......
        zigzag[08] <= #1 din_56;
        zigzag[07] <= #1 din_65;
        zigzag[06] <= #1 din_74;
        zigzag[05] <= #1 din_75;
        zigzag[04] <= #1 din_66;
        zigzag[03] <= #1 din_57;
        zigzag[02] <= #1 din_67;
        zigzag[01] <= #1 din_76;
        zigzag[00] <= #1 din_77;
    end
        ......
```

其中，zigzag 定义为一组寄存器，存放 zigzag 扫描后的图像块数据。

```
reg [9:0] sresult [63:0];
```

然后，用移位寄存器将二维数据变换为一维序列输出，用下面的代码可以实现。

```
always @(posedge clk)
    ......
    for (i=63; i<=1; i=i-1)
        zigzag[i] <= #1 zigzag[i-1];
        ......
    assign data_out = zigzag [63];
```

data_out 为扫描排序完成后得到的一维数据序列。

6.4 量　　化

6.4.1 量化的概念

量化，就是将图像数据的取值范围分成 n 个区间，每个区间内的所有取值用一个值来代替。一般情况下，$n=2^k$，k=6，7，或 8，即可以将图像的数据分为 64，128，或 256 个区间。

量化可以减少图像的数据量，显然，这是以量化后图像的质量为代价的，所分的区间越少，则量化后的数据量越小，而复原图像与原始图像的误差就越大。在图像压缩系统中，一般在 DCT 变换之后，再进行量化，很多值会量化为 0，这大大降低了后面计算的数据量。

量化可以分为均匀量化和非均匀量化。均匀量化就是将图像数据的取值范围均匀地分为 n 个区间，这是比较简单的做法。但如果图像中数据分布并不均匀，在某个取值范围内频繁出现，在其它范围内却很少出现，则可以采用非均匀量化，在频繁出现的取值范围内采用小区间的密集量化，在很少出现的取值范围内采用大区间的量化，即量化区间的长度不同。非均匀量化虽然比较复杂，但是可以在不增加量化层次的条件下，减少量化所带来的误差。

例如，取值范围为 0～63 的数据，均匀量化为 0～7，其对应关系如表 6.3 所示。

表 6.3　均匀量化举例

原始数据	0～7	8～15	16～23	24～31	32～39	40～47	48～5	56～63
量化后的数据	7	6	5	4	3	2	1	0

如果要对数据进行非均匀量化，则其中的一种对应关系可以如表 6.4 所示。

表 6.4　非均匀量化举例

原始数据	0～15	16～23	24～28	29～31	32～34	35～39	40～47	48～63
量化后的数据	7	6	5	4	3	2	1	0

6.4.2 量化模块设计

非均匀量化的硬件实现有一定难度，一般的图像系统都采用相对较简单的均匀量化。

根据量化的概念，很容易想到用嵌套的 if 语句来实现，但实际上对于均匀量化，可以利用除法完成。例如表 6.3 中的例子，可通过直接将数据除以 8，然后将小数舍去来完成量化。而除法器的设计在 4.2.3 节中已经作了介绍，这里不再详述。

6.5 霍夫曼编码/解码

6.5.1 霍夫曼码原理

图像的数据量非常大，为了能有效地传输和存储图像，有必要对图像进行压缩。以恢

复的图像和原始图像是否一致来分，可以将图像压缩编码分为无失真编码和限失真编码两种。霍夫曼码属于无失真编码，它要求经过编解码后得到的图像应与原图完全一致，编码过程不丢失任何信息。

霍夫曼码的思想是：当所有数据出现的概率不相等时，如果采用不等长码字，将概率大的数据编码为较短的码字，而概率相对较小的数据可以用相对较长的码字，这样可以减少平均码长，提高编码效率。

霍夫曼码可实现的冗余度最小的编码，属于不等长码，在收端可以得到惟一的解码，且无失真。

霍夫曼码的编码过程：

(1) 将数据按出现概率排序，概率较大的在前。将概率最小的两个数据编码为相同的码长，仅最后一位不同(分别为 0，1)，其他各位相同(具体是 0 或 1 要等编码完成后才确定)。

(2) 将最后两个数据的概率相加，合为一个，再按概率重新排序。然后依(1)中的方法对概率最小的两个数据编码，仍然只确定最后一位(相对第 1 步来讲是倒数第二位)。

(3) 重复(2)，直到只剩下两个数据为止。

霍夫曼码编码的过程如图 6.7 所示。

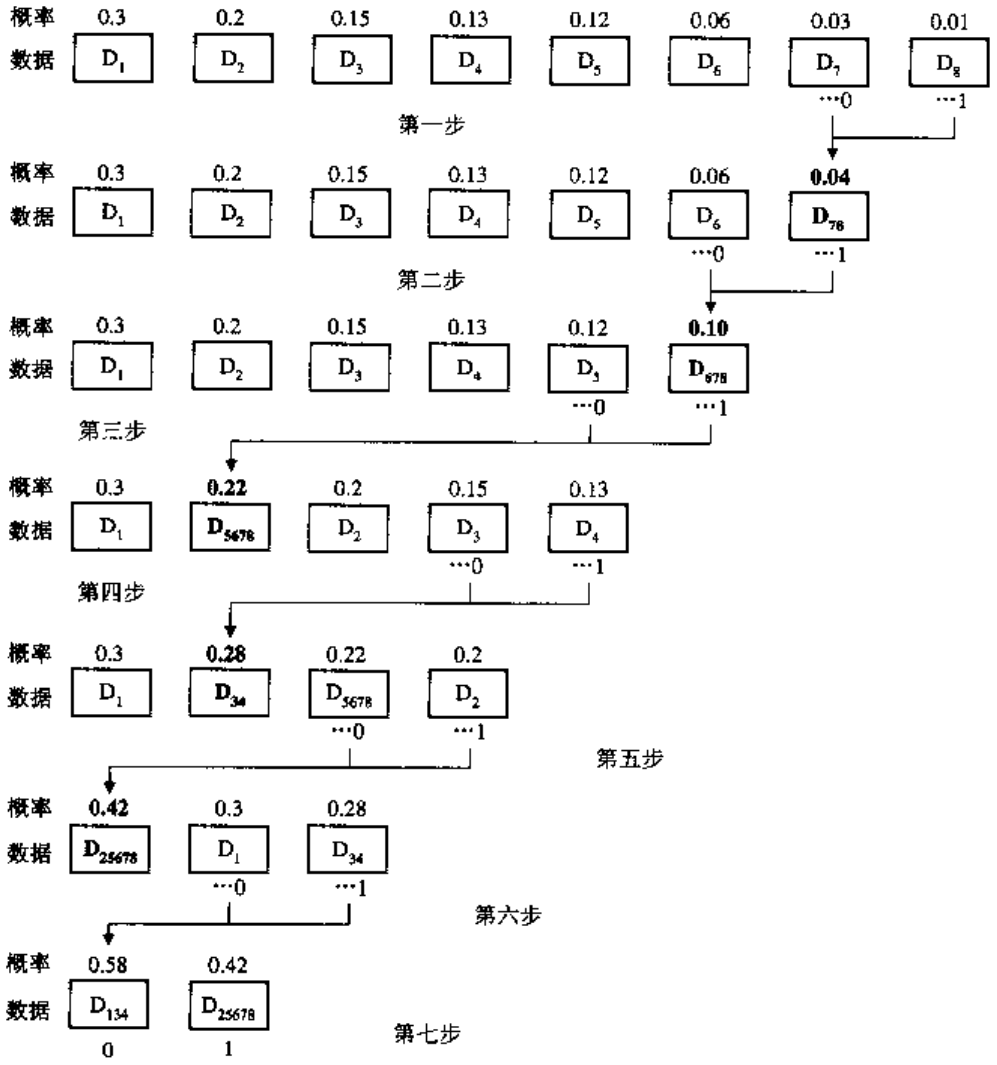


图 6.7　霍夫曼码的编码过程示意图

由图 6.7 中得到的码字如下：

D_1：00；D_2：11；D_3：010；D_4：011；D_5：100；D_6：1010；D_7：10110；D_8：10111。

霍夫曼码的任何一个码字都不是其它码字的前缀，所以解码结果是唯一的。

霍夫曼码的解码过程可用树形来形象地表示，如图 6.8 所示。

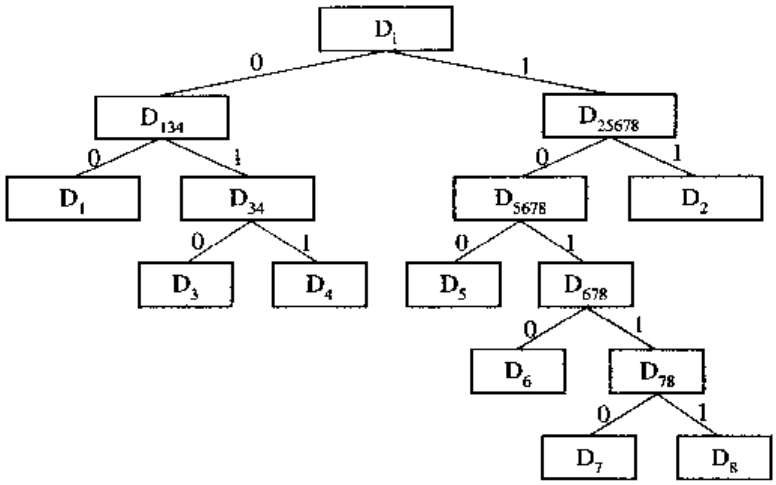


图 6.8　霍夫曼码的解码过程

例如接收到的 11101010111，第 1 位码为 1，则取右分支，第 2 位为 1，又取右分支，为 D_2，第 3 位为新数据的起始位，可得到 1010 为 D_6，10111 为 D_8，得到 $D_2D_6D_8$，从解码过程中可以看出，虽然霍夫曼码是不等长码，但在解码过程中能自己确定码字的起止位置，解码结果是唯一的，不会造成任何混淆。

6.5.2　霍夫曼码编码/解码模块设计

霍夫曼码设计好之后，其码表的硬件实现可利用 case 语句用查找表结构来完成。如 6.5.1 节所示的霍夫曼码，其码表的一种实现代码如下：

```
function [7:0] huffman_enc;
  input [ 3:0] data;
begin
  case(data)
    4'h1: huffman_enc = {3'h2, 5'b0_0000};
    4'h2: huffman_enc = {3'h2, 5'b0_0011};
    4'h3: huffman_enc = {3'h3, 5'b0_0010};
    4'h4: huffman_enc = {3'h3, 5'b0_0011};
    4'h5: huffman_enc = {3'h3, 5'b0_0100};
    4'h6: huffman_enc = {3'h4, 5'b0_1010};
    4'h7: huffman_enc = {3'h5, 5'b1_0110};
    4'h8: huffman_enc = {3'h5, 5'b1_0111};
  endcase
end
endfunction
```

其中，huffman_enc 的前三位表示该码字的长度，后五位表示数据所对应的码字。例如 data=4，则 huffman_enc=011 00011，即对应的码字为 011=3 位，则截取码字 00011 的后三位，为 011。

同样，霍夫曼码的解码码表也可以利用 case 语句通过查找表结构实现。一种实现代码如下：

```
function [6:0] huffman_dec;
  input [4:0] codeword;
begin
  casex(codeword)
    5'b0_0???: huffman_dec = {3'h2, 4'h1};
    5'b1_1???: huffman_dec = {3'h2, 4'h2};
    5'b0_10??: huffman_dec = {3'h3, 4'h3};
    5'b0_11??: huffman_dec = {3'h3, 4'h4};
    5'b1_00??: huffman_dec = {3'h3, 4'h5};
    5'b1_010?: huffman_dec = {3'h4, 4'h6};
    5'b1_0110: huffman_dec = {3'h5, 4'h7};
    5'b1_0111: huffman_dec = {3'h5, 4'h8};
  endcase
end
endfunction
```

其中，huffman_dec 的前三位表示该码字的长度，后四位表示解码得到的数据。例如 codeword=1010，则 huffman_dec=100 0110，即码字长度为 100=4，解码得到的数据为 0110=6。

在代码中直接调用 huffman_enc 函数，即可完成霍夫曼编码。如下面代码所示：

```
assign codeword = huffman_enc( data[3:0] );
```

在代码中直接调用 huffman_dec 函数，即可完成霍夫曼解码。如下面代码所示：

```
assign data = huffman_dec( codeword[3:0] );
```

注意：霍夫曼码为不等长码，原始数据和编码后的数据位数不一致，因此，对于数据的输入输出可考虑采用移位寄存器实现。

6.6 JPEG

JPEG 标准是由国际标准化组织(ISO)下属的联合图片专家组(JPEG，Joint Photographic Experts Group)制定的，对彩色和灰度图像进行压缩的公用标准。

JPEG 标准中压缩算法的基本步骤包括 DCT、量化、编码等。其编解码过程的框图如图 6.9 所示。

图中，FDCT 为正向 DCT，IDCT 为逆向 DCT。扫描排序一般用 zigzag 扫描方法，熵编码一般用霍夫曼编码，对 DCT 变换后的直流系数和交流系数分别进行编/解码。

JPEG 的具体设计可由上面所介绍的 DCT、量化、编解码等模块组成，并加上控制模块对时序进行控制。这里不再详细说明。

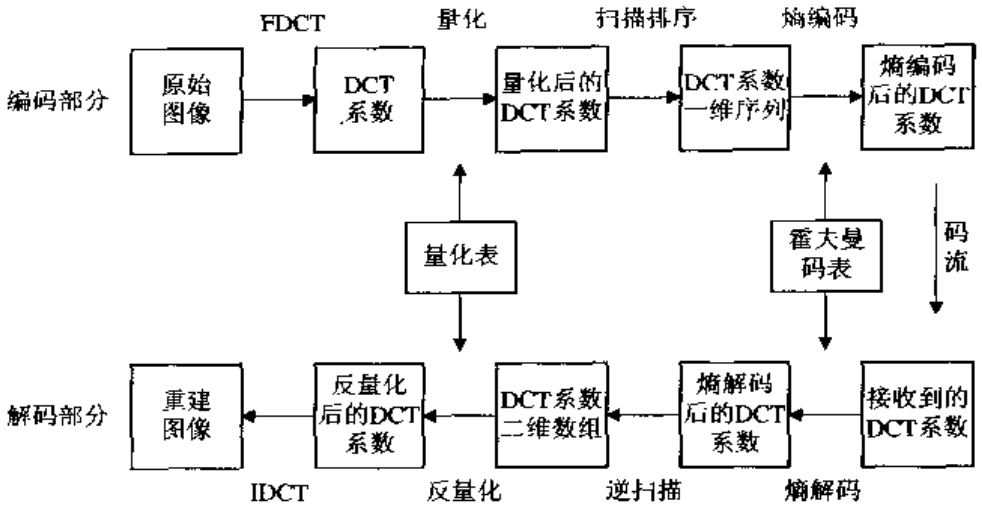


图 6.9 JPEG 系统框图

6.7 MPEG

MPEG 标准是运动图像专家组(MPEG，Moving Pictures Experts Group)制定的，用于运动图像及相应音频压缩编码的标准系列。它包括 MPEG-1，MPEG-2，和 MPEG-4 等。

目前应用比较广泛的是 MPEG-2 技术。MPEG-2 技术综合采用了三种基本编码技术，即预测编码、变换编码和统计编码。该压缩技术采用了多种编码手段来消除系统的冗余信息，归纳起来有以下四个方面：

(1) 利用二维 DCT 来减少图像的空间冗余度；

(2) 利用运动补偿预测来减少图像的时间域冗余度；

(3) 利用视觉加权量化来减少图像的“灰度域”冗余度；

(4) 利用熵编码来减少图像“频率域”上统计特性方面的冗余度。

MPEG-2 视频编码器的系统框图如图 6.10 所示。

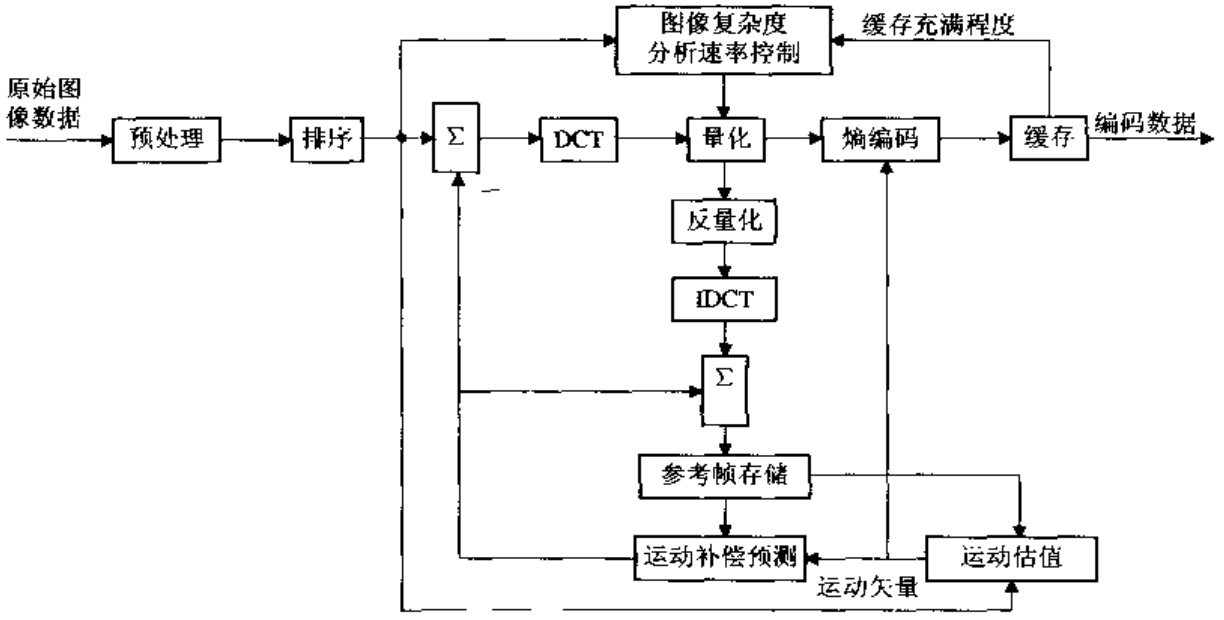


图 6.10 MPEG-2 编码器系统框图

MPEG-2 视频解码器的系统框图如图 6.11 所示。

MPEG 系统中的 DCT、量化、编解码等部分仍可以利用前面所介绍的模块。系统中运动估计和运动补偿部分是最重要也是最有难度的地方。限于篇幅，这里不再详述。

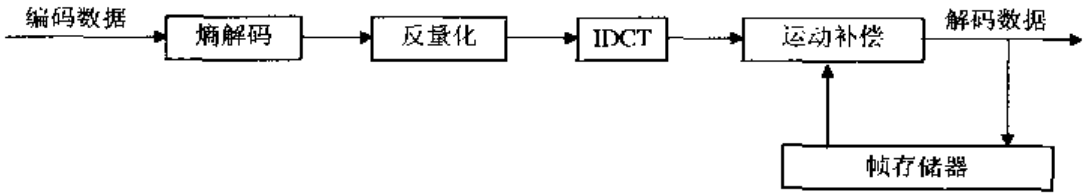


图 6.11　MPEG-2 解码器系统框图

6.8　VGA 控制器

6.8.1　视频基础知识

VGA(视频图形阵列)是计算机的一种图像显示模式，下面对 VGA 中的时序进行说明。

在 VGA 中，水平同步脉冲在光栅扫描线需要回到水平开始位置(即屏幕的左边)时插入，垂直同步脉冲在光栅扫描线需要回到垂直开始位置(屏幕的上方)时插入，而复合同步脉冲则是水平同步信号和垂直同步信号的组合。在没有图像投影在屏幕上时插入消隐信号。r,g,b 是像素的数据，当消隐信号有效时，RGB 的值是无效的。

1) 水平时序

图 6.12 给出了水平视频时序示意图。

在水平时序中，包括以下时序参数：

Thsync：水平同步脉冲的宽度。

Thgdel：水平同步脉冲的结束和水平门的开始之间的宽度。

Thgate：一个视频行可视区域的宽度。

Thlen：一个完整的视频行的宽度，从水平同步脉冲的开始直到下一个水平同步脉冲的开始。

说明：以上均以像素时钟来计算。

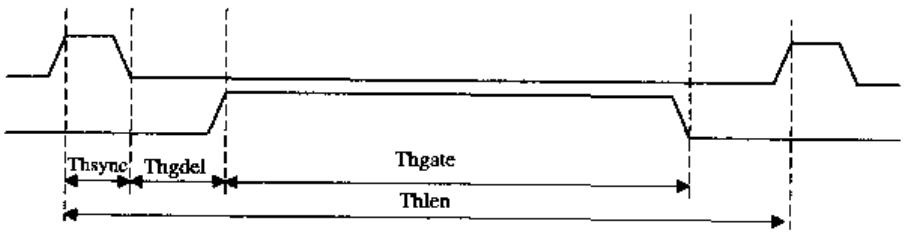


图 6.12　水平视频时序示意图

2) 垂直时序

在图 6.13 中给出了垂直视频时序的示意图。

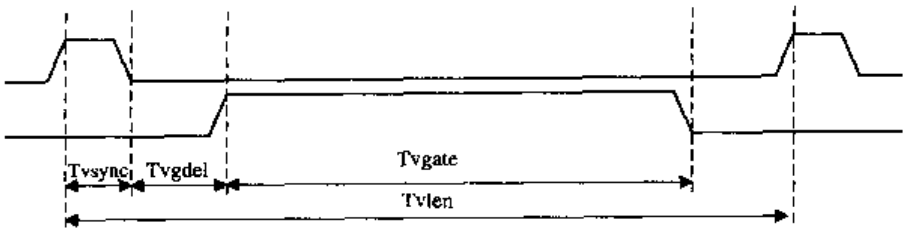


图 6.13　垂直视频时序示意图

在垂直时序中，包括如下时序参数：

Tvsync：垂直同步脉冲的宽度。

Tvgdel：垂直同步脉冲的结束和垂直门的开始之间的宽度。

Tvgate：一个视频帧可视区域的宽度。

Tvlen：一个完整的视频帧的宽度，从垂直同步脉冲的开始直到下一个垂直同步脉冲的开始。

说明：以上均以视频行来计算。

3) 组合视频帧时序

组合视频帧时序示意图如图 6.14 所示。

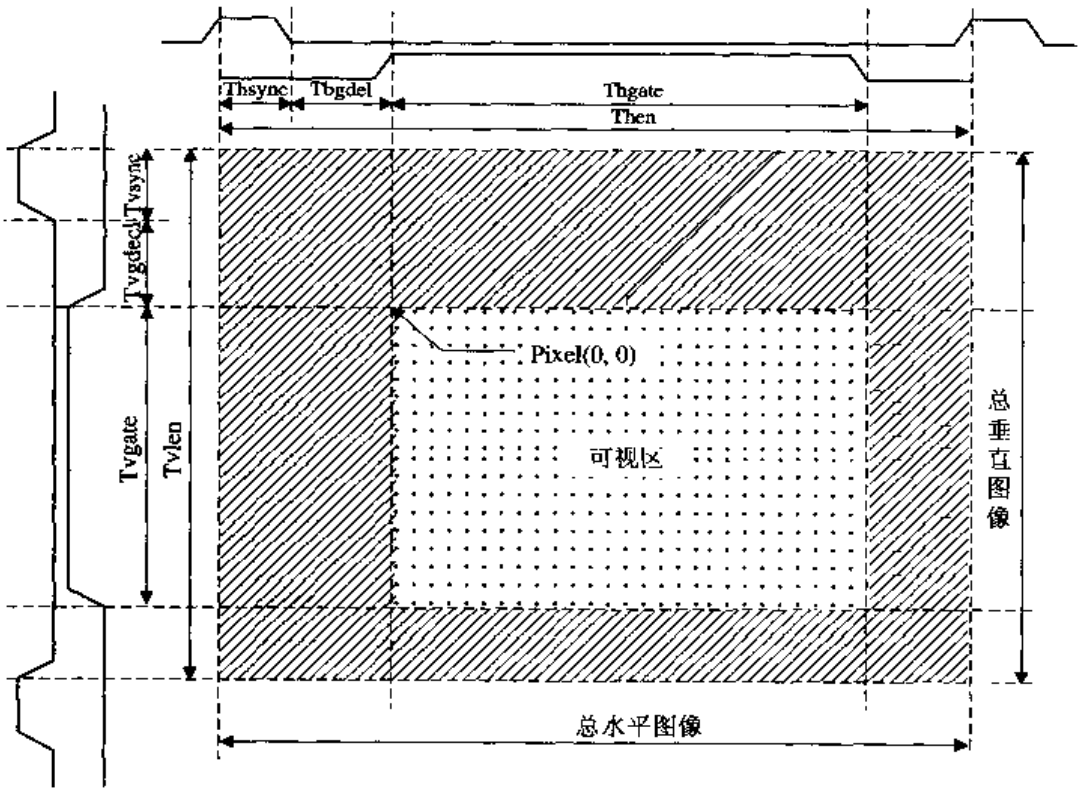


图 6.14　组合视频时序示意图

视频帧由 Tvlen 个视频行组成，每行有 Thlen 个像素。水平门和垂直门的“与”函数即为可视区域，图像的其它区域是消隐区。

目前存在很多常用的 VGA 模式，表中各参数的值随模式的不同而变化。表 6.5 和表 6.6 分别给出了一些普通 VGA 模式中的垂直时序和水平时序信息。

表 6.5　一般 VGA 模式的垂直时序信息

模式	分辨率	刷新速率/Hz	行宽/μs	同步脉冲 Tvsync	后沿 Tvgdel	有效时间 Tvgate	前沿	帧长 Tvlen
VGA	640×480	60	31.78	63 μs 2 行	953 μs 30 行	15 382 μs 484 行	285 μs 9 行	16 683 μs 525 行
VGA	640×480	72	26.41	79 μs 3 行	686 μs 26 行	12 782 μs 484 行	184 μs 7 行	13 735 μs 520 行
VGA	800×600	56	28.44	56 μs 1 行	568 μs 20 行	17 177 μs 604 行	1*	17 775 μs 625 行
VGA	800×600	60	26.40	106 μs 4 行	554 μs 21 行	15 945 μs 604 行	1*	16 579 μs 628 行
VGA	800×600	72	20.80	125 μs 6 行	436 μs 21 行	12 563 μs 604 行	728 μs 35 行	13 853 μs 666 行

说明：

有效时间包含 4 行过扫描边界行。有些时序表中将这几行加在后沿和前沿中。

*当有效时间增加时，它超过了 vsync 信号的上升沿，因此，前沿为-1。

表 6.6　一般 VGA 模式的水平时序信息

模式	分辨率	刷新速率/Hz	像素时钟/MHz	同步脉冲 Thsync	后沿 Thgdel	有效时间 Thgate	前沿	行总长 Thlen
VGA	640×480	60	25.175	3.81 μs 96 像素	45 像素	646 像素	13 像素	800 像素
VGA	640×480	72	31.5	1.27 μs 40 像素	125 像素	646 像素	21 像素	832 像素
VGA	800×600	56	36	2 μs 72 像素	125 像素	806 像素	21 像素	1024 像素
VGA	800×600	60	40	3.2 μs 128 像素	85 像素	806 像素	37 像素	1056 像素
VGA	800×600	72	50	2.4 μs 120 像素	61 像素	806 像素	53 像素	1040 像素

说明：

有效时间包含 6 列过扫描边界列，有些时序表将这几列加在后沿和前沿中。

6.8.2　VGA 控制器的设计

1. 功能描述

这里较粗略地给出 VGA 控制器的功能：

与 VGA 标准兼容，产生符合 VGA 标准的时序控制信号。

支持 24 位和 16 位的色彩模式；8 位灰度等级和 8 位伪色彩模式(伪色彩模式使用额外的存储器作为色彩查找表)。

支持视频存储器 bank 切换。

2. 时序

VGA 控制器中的时序包括：水平视频时序、垂直视频时序和组合视频时序。

VGA 控制器中较重要的时序信号包括 clk(像素时钟)、hsync(水平同步脉冲)、vsync(垂直同步脉冲)、csync(组合同步脉冲)和 blank(消隐信号)等。

3. 系统框架

在该系统中，主要包括如下模块：

(1) 视频时序生成器：能产生水平同步脉冲(hsync)、垂直同步脉冲(vsync)、组合同步脉冲(csync)、消隐信号(blank)，以及对线性 FIFO 的读请求等时序控制信号。

(2) 色彩处理器：能将收到的像素数据转换为 8 位 RGB 色彩信息。在输入为 24 位色彩时无需转换，直通输出。在输入为 16 位(5 位 R，6 位 G，5 位 B)模式时，只需进行线性转换。在输入为 8 位灰度模式时，R，G，B 的值是相同的，因而得到的是黑白图像。该色彩处理器还可以工作在 8 位伪色彩模式下，此时输入的数据为 8 位，将收到的像素送到外部色彩查找表中，输出 24 位色彩数据。该模块还包括一个线性 FIFO，用以保证送到显示器的数据流是连续的。

VGA 控制器的系统框架结构如图 6.15 所示。

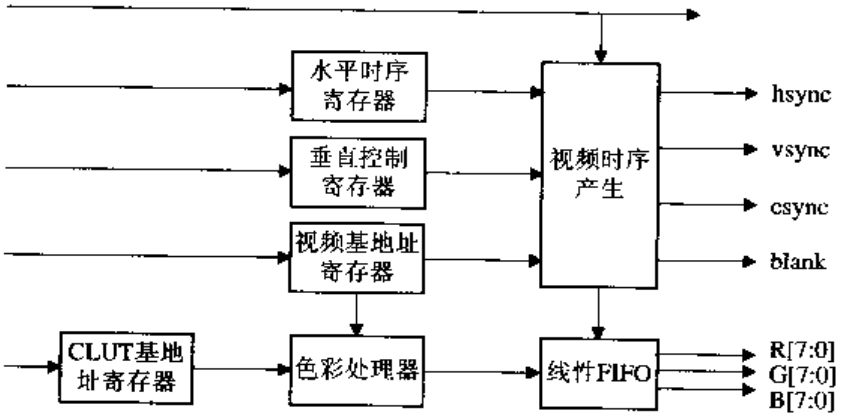


图 6.15　VGA 控制器的框架结构。

4. 寄存器

在这个 VGA 控制器中，需要用到的寄存器有：水平时序寄存器、垂直时序寄存器、水平和垂直长度寄存器、视频存储器基地址寄存器和色彩查找表基地址寄存器。

其中，视频存储器基地址寄存器包括外部视频存储器组的开始地址。色彩查找表基地址寄存器用于在 8 位伪色彩模式下给出输出的 RGB 值。

表 6.7 给出这些寄存器的一般描述。

表 6.7　VGA 中的寄存器

名称	存取	描述
HTIM	读/写	水平时序寄存器
VTIM	读/写	垂直时序寄存器
VBAR	读/写	视频存储器基地址寄存器
CBAR	读/写	色彩查找表基地址寄存器

这些寄存器的结构如表 6.8 所示。

表 6.8　寄存器的结构

寄存器类别	名称	描述	备注
水平时序寄存器	Thsync	水平脉冲宽度	像素点数
	Thgdel	水平门延迟时间	像素点数
	Thgate	水平门时间	像素点数
	Thlen	水平长度	像素点数
垂直时序寄存器	Tvsync	垂直脉冲宽度	视频行数
	Tvgdel	垂直门延迟时间	视频行数
	Tvgate	垂直门时间	视频行数
	Tvlen	垂直长度	视频行数
视频存储器基地址寄存器	VBA	视频基地址	定义了视频存储器的开始位置。图像就存于存储器中由该视频基地址开始的连续位置
色彩查找表基地址寄存器	CBA	色彩查找表基地址	定义了外部色彩查找表的地址

5. 视频时序生成器的设计

视频时序生成器可以生成 VGA 相应的时序。其框图如图 6.16 所示。表 6.9 给出了它的端口描述及信号说明。

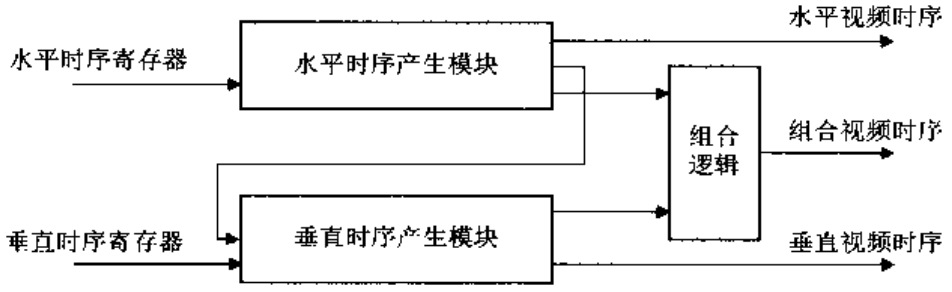


图 6.16　VGA 模块框图

时序生成模块的 module 定义如下：

module vga_pgen (clk, ena, hsync_l, Thsync, Thgdel, Thgate, Thlen, vsync_l, Tvsync, Tvgdel, Tvgate, Tvlen, csync_l, blank_l, eoh, eov, gate, hsync, vsync, csync, blank);

表 6.9　端口描述及信号说明

信号名	I/O	说明
mclk	I	主时钟
pclk	I	像素时钟
ena	I	视频使能信号
hsync_l	I	水平同步脉冲极化电平(正/负)
Thsync[7:0]	I	水平同步脉冲宽度(像素数目)
Thgdel[7:0]	I	水平门延迟(像素数目)
Thgate[15:0]	I	水平门长度(每行可视像素的数目)
Thlen[15:0]	I	水平长度(每行像素的数目)
vsync_l	I	垂直同步脉冲极化电平(正/负)
Tvsync[7:0]	I	垂直同步脉冲宽度(行数)
Tvgdel[7:0]	I	垂直门延迟(行数)
Tvgate[15:0]	I	垂直门长度(每帧可视行的数目)
Tvlen15:0]	I	垂直长度(每帧的行数目)
sync_l	I	组合同步脉冲极化电平
blank_l	I	消隐信号极化电平
eoh	O	水平的结束
eov	O	垂直的结束
gate	O	垂直 gate 和水平 gate 的逻辑“与”
hsync	O	水平同步脉冲
vsync	O	垂直同步脉冲
Csync	O	组合同步：水平同步和垂直同步逻辑“或”
Blank	O	消隐信号

该模块用到的寄存器包括水平时序寄存器、垂直时序寄存器、水平与垂直长度寄存器。

水平和垂直时序信号可以用相似的模块来生成。我们将这个模块称为 vga_tim。该模块的描述如下：

module vga_vtim(clk, ena, rst, Tsync, Tgdel, Tgate, Tlen, Sync, Gate, Done);

端口信号说明如表 6.10 所示。

表 6.10　端口描述及信号说明

信号名	I/O	说明
clk	I	时钟
ena	I	计数使能
rst	I	同步高有效复位信号
Tsync[7:0]	I	同步宽度 1
Tgdel[7:0]	I	门延迟-1
Tgate[15:0]	I	门长度-1
Tlen[15:0]	I	行长/帧长
Sync	O	同步脉冲
Gate	O	门限信号
Done	O	行/帧结束

要产生的时序信号包括 done、gate 和 sync。其中，sync 是同步脉冲，用于行/帧同步。gate 用于标志像素的有效区域(可视区域)。done 用于标志行/帧的结束。

在这个模块中，输入信号包括：Tsync(同步信号的长度)、Tgdel(同步信号与门限信号之间的间隔)、Tgate(门限信号的长度)、Tlen(总长度)。根据这些信号，可以很容易画出行/帧的时序图，请参见图 6.12~6.14。

时序信号生成过程如下：

- 每一行的结束信号 eol：表示每一行结束时，该信号有效。

  assign eol = hdone;

- 门限信号 gate：用来表示像素的有效区域。hgate 表示水平有效区域，vgate 表示垂直有效区域，hgate & vgate 表示整个可视区域，如图 6.14 所示。

  assign gate = hgate & vgate;

- 水平同步脉冲信号 hsync：由生成的水平同步脉冲与水平同步脉冲极化电平异或得到。

  assign hsync = ihsync ^ hsync_l;

- 垂直同步脉冲信号 vsync：由生成的垂直同步脉冲与垂直同步脉冲极化电平异或得到。

  assign vsync = ivsync ^ vsync_l;

- 组合同步信号 csync：由水平同步或场同步信号，与组合同步电平异或得到。

  assign csync = (ihsync | ivsync) ^ csync_l;

- 消隐信号 blank：由门限信号与消隐电平异或得到。

  assign blank = !(gate ^ blank_l);

在 4.3.3 节中，介绍了用线性移位寄存器产生定时标志信号的方法，这里不再赘述。图 6.17 给出了该模块的仿真波形。

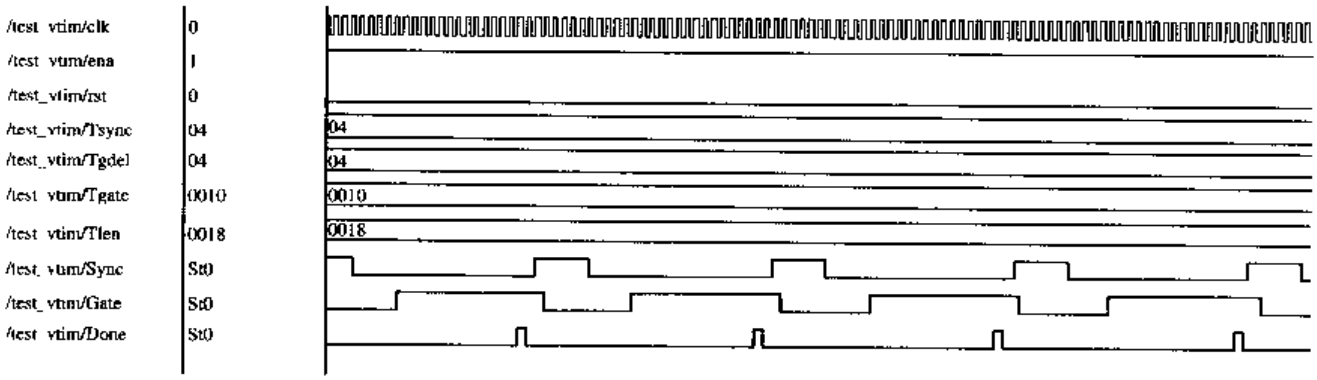


图 6.17　VGA 模块的仿真波形

另一种较简单的实现方法是利用计数器来生成有关的时序信号，读者可以完成用计数器生成时序的代码。

6. 色彩处理器的设计

1) 模块框图

色彩处理器的结构如图 6.18 所示。

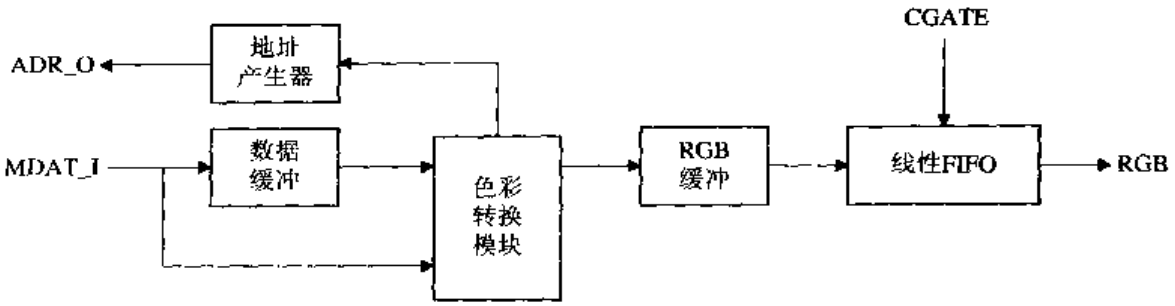


图 6.18　色彩处理器结构

首先将从外部输入的数据放到数据缓冲中，再放到色彩转换模块中(将数据转化为色彩信息)，最后将色彩信息送到线性 FIFO 中，由 FIFO 送到显示器。

色彩转换模块与视频存储器和色彩查找表进行交互：

色彩转换模块产生视频存储器的地址，对视频存储器进行控制，对读入的数据进行处理，转换成所用的色彩数据。如果需要用到色彩查找表(在伪 8 位色彩模式下)，则色彩转换模块还要产生色彩查找表的地址，并进行相应的控制，从中得到所需的数据。

图 6.18 中还包括一个数据缓冲，用于临时存储从视频存储器读出来的数据；包括一个 RGB 缓冲，临时存放由色彩转换模块生成的 RGB 色彩值。

色彩处理器要用到视频基地址寄存器与色彩查找表基地址寄存器。

2) 实现代码

在色彩处理模块中，要对输入的色彩数据进行处理。这在前面已经提到过。

下面给出输入像素为 24 位数据时的一种实现代码。数据不做任何操作，直接输出。但由于数据缓冲器设计为 32 位，所以要用代码控制缓冲的各位数据，以正确地输出数据。

```
//每像素为 24 bit
24:
begin
  if (!rgb_fifo_full)
    case (colcnt)
        2'b11:
        begin
          iR  = DataBuffer[31:24];
          iG  = DataBuffer[23:16];
          iB  = DataBuffer[15: 8];
          iRa = DataBuffer[ 7: 0];
        end

        2'b10:
        begin
          iR  = Ra;
          iG  = DataBuffer[31:24];
          iB  = DataBuffer[23:16];
```

```
            iRa = DataBuffer[15: 8];
            iGa = DataBuffer[ 7: 0];
        end

        2'b01:
        begin
            iR  = Ra;
            iG  = Ga;
            iB  = DataBuffer[31:24];
            iRa = DataBuffer[23:16];
            iGa = DataBuffer[15: 8];
            iBa = DataBuffer[ 7: 0];
        end

        default:
        begin
            iR = Ra;
            iG = Ga;
            iB = Ba;
        end
      endcase
  end
```

下面给出输入为 16 位数据时代码的实现。在该代码中，将 16 位(R5G6B5)数据直接输出，并且在后面补 0。由于数据缓冲为 32 位，所以代码实现分为两部分，一部分是缓冲中前 16 位数据的输出，另一部分是后 16 位的输出。

```
// 每像素为 16 bit
16a: //数据缓冲的前 16 位输出
begin
 if (!rgb_fifo_full)
   RGBbuf_wreq = 1'b1;
       iR[7:3] = DataBuffer[31:27];
       iG[7:2] = DataBuffer[26:21];
       iB[7:3] = DataBuffer[20:16];
end

16b: //数据缓冲的后 16 位输出
begin
 if (!rgb_fifo_full)
   begin
     RGBbuf_wreq = 1'b1;
```

```
            if (!vdat_buffer_empty)
              ivdat_buf_rreq = 1'b1;
        end

        iR[7:3] = DataBuffer[15:11];
        iG[7:2] = DataBuffer[10: 5];
        iB[7:3] = DataBuffer[ 4: 0];
    end
```

色彩处理器还可以处理输入为 8 位数据的情况，这有两种处理方式：一种处理是直接输出，RGB 值都相同；另一种是伪色彩模式，将输入的 8 位数据通过色彩查找表转为 24 位彩色数据 RGB。希望读者可以完成这部分代码。

6.9 练　　习

1. 对 6.5 节中所举的霍夫曼码的例子，完成其编码模块和解码模块的 Verilog 代码，并进行仿真。

2. 用计数器实现 VGA 时序的生成，用综合工具进行综合，并与用移位寄存器实现代码的综合结果进行比较。

3. 试完成一个简单的 JPEG 编码系统，包括 DCT 变换、量化、zigzag 扫描及霍夫曼编码。

第7章 CPU的设计

CPU是许多数字系统的核心。CPU控制复杂，对健壮性、面积、性能都有较高要求，所以，设计一个CPU对设计者来说是一个很好的锻炼。

CPU的设计，首要问题是架构设计。设计者要考虑：指令集是怎样的？CPU内采用几条总线？用多少个寄存器？多少缓存？采用几级流水线？怎样进行分支预测、指令预取(Prefetch)和推测执行(Speculative Excution)？本书以8/16位RISC和8051为例，说明CPU的设计过程。这些示例的复杂性跟Pentium商用高性能CPU当然不可相提并论，但“麻雀虽小，五脏俱全”。

建议读者在阅读本章之前，应具备一些计算机体系结构方面的知识。这方面有许多很经典的书，像《计算机组织与设计：硬件/软件接口》、《结构化计算机组成》、《计算机体系结构》等，都值得仔细研读。

7.1 基础知识

1. CPU基本结构

一个基本的CPU要包括三部分功能：数据的存储、数据的运算和控制部分。与之相对应的硬件结构也分为三部分：存储器、数据通路和控制器。**存储器**存放指令和数据。**数据通路**包括ALU、程序计数器等，主要功能是对操作数进行运算，得到结果，并产生程序计数器的值，作为要执行的下一条指令的地址。**控制器**内有指令寄存器，它对指令进行译码，产生相应的控制信号，完成对存储器和数据通路部分的控制。图7.1画出了三者之间的关系。

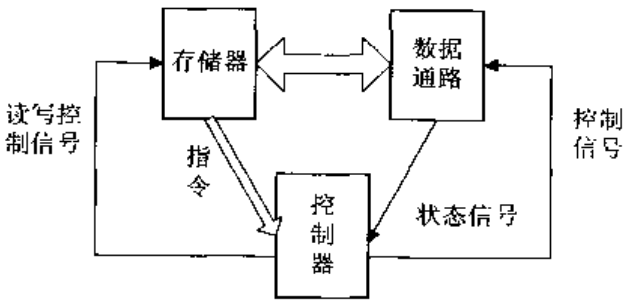


图7.1 CPU的基本结构

2. 冯·诺依曼结构和哈佛结构

计算机体系结构可分为两种类型：冯·诺依曼结构及哈佛结构。大多数CPU都采用冯·诺依曼结构。

冯·诺依曼结构有如下特点：

- 存储器是字长固定的、顺序线性编址的一维结构。

- 存储器提供可按地址访问的一级地址空间，每个地址是惟一定义的。
- 由指令形式的低级机器语言驱动。
- 指令顺序执行，即一般按照指令在存储器中存放的顺序执行，程序分支由转移指令实现。
- 以运算器为中心，输入输出设备与存储器之间的数据传送都途经运算器。运算器、存储器、输入输出设备的操作以及它们之间的联系都由控制器集中控制。

哈佛结构多用于 DSP 中。在哈佛结构中，数据存储与程序存储分为两块存储器，允许有两个同时的取存储器操作。这样，以面积的增加为代价就获得了较高的速度。

3. 指令

CPU 所能执行的基本操作是指令。指令由一个位串来表示。这些位串可划分成几个字段，指令的这种格式称为指令格式。图 7.2 给出了一种简单的指令格式。

指令通常要包括如下信息：

- 操作码，指定将要完成的操作(例如 ADD)。
- 操作数，给出操作数来源，并说明如何存放结果。
- 下一指令地址，告诉 CPU 去哪里取下一条指令。

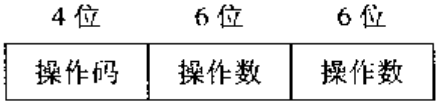


图 7.2　一种简单的指令格式

CPU 执行指令的一般步骤为：

- 对每种指令，首先要将程序计数器 pc 的值送到包含代码的存储器中，然后将指令从存储器中取出(指令与数据可以一起存放，也可以分开存放)。
- 从一组寄存器中将操作数读入寄存器。寄存器的地址由指令的某一部分指定。
- 接下来是具体执行，执行过程跟具体指令的类型有关。通常要用到 ALU。其中，load 和 store 指令使用 ALU 来计算地址，然后将数据从存储器中读出或者将数据写入到存储器；算术逻辑指令使用 ALU 完成操作，然后写入到寄存器；分支指令使用 ALU 完成比较，然后改变下一个指令的地址。

4. 指令集

CPU 能执行的各种不同的指令的集合称为指令集。指令集必须能够映射通常的高级语言语句。指令一般可以分为如下几类：

- 数据处理：算术和逻辑指令。
- 数据存储：存储器指令。
- 数据传送：I/O 指令。
- 控制：测试和转移指令。

其中，算术和逻辑指令提供了处理数值型数据和对数据位进行逻辑操作的能力；存储器指令用于在寄存器和存储器间传递数据；I/O 指令用于将程序和数据传递到存储器，并将计算结果返回给用户；测试指令用于测试数据字的值或计算的状况；转移指令根据判定条件跳转到另一些指令上。

表 7.1 给出常见指令类型的说明。

表 7.1　常见指令类型说明

指令类型	CPU 的动作	常见指令
数据处理	在 ALU 内完成功能，设置条件代码和标志信息	add，multiply，and.
数据存储	将数据从一个位置送到另一个位置	store，load
数据传送	向 I/O 模块发命令	input(read)，output(write)
控制	修改程序计数器。对于子程序的调用和返回，管理参数传送和链接	jump，skip，wait

指令集的设计是一件很复杂的事情。设计者需要考虑要处理的数据类型、所提供的指令类型、指令的长度、地址数目、各字段的大小、指令能访问的寄存器的数目及它们的用途、寻址方式等等。

数据类型通常可以分为数值、字符和逻辑数据三类。计算机使用的数值数据包括整数、浮点数两类。字符一般用 ASCII(American Standard Code for Information Interchange)码表示。逻辑数据可以看作是由 n 个 1 位项组成，每一项取值为 0 或 1。

下面以一个简单的 8 位 RISC(精简指令集计算机)为例，说明 CPU 设计的过程。

7.2　8 位 RISC 的设计

1．架构

对图 7.1 中的结构进行细化，可以得到一个简单 CPU 的架构。这个 CPU 是 8 位的，它的寄存器宽度、总线宽度都是 8 位。

CPU 采用总线结构，即控制器所需的指令、数据通路所需的数据都是从总线上得到的。相应的结构如图 7.3 所示。

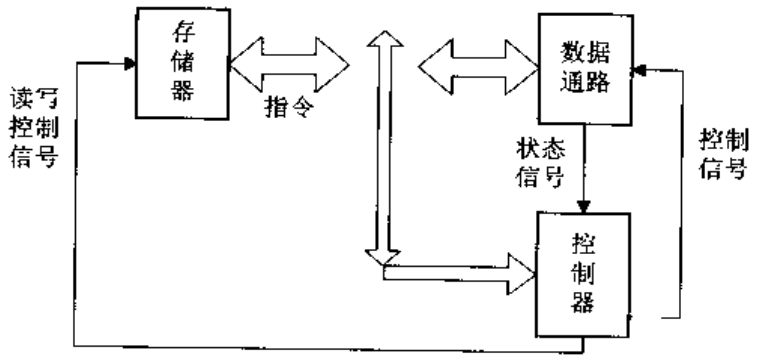


图 7.3　基于总线的 CPU 结构

2．指令集

定义指令格式如图 7.4 所示。其中，操作码 3 位，最多可实现 8 条指令；地址 5 位，可寻址 32 字节的存储器。

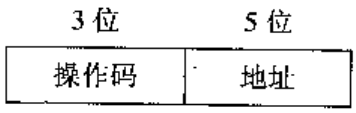


图 7.4　简单指令格式

在这个设计中，共实现了 6 条指令。指令集如表 7.2 所示。

表 7.2　基本 CPU 的指令集

7~5 位	4~0 位	指令	结果
000	操作数地址	add [addr]	A = a + M[addr];
001	操作数地址	and [addr]	A=a&M[addr];
010	操作数地址	xor [addr]	A = a ⊕ M[addr];
011	操作数地址	load [addr]	A=M[addr];
100	存放地址	store [addr]	M[addr]=a;
101	下个 pc 地址	Jump [addr]	Pc=addr;
110	保留		
111	保留		

3. 存储器

存储器中存放了要执行的指令和相应数据。存储器采用 SRAM 来实现。实现代码参考第 4 章对 RAM 的描述。存储器的大小是 32×8，即可以存放 32 个字节。

存储器的读写信号由控制器给出。存储器的地址来源有两个：程序计数器和指令(在本例中，每个指令都包含地址部分，用于指示存取数据的位置，如图 7.4 所示)。在取新指令时，用程序计数器的值作为存储器地址；在执行指令时，用指令中的地址部分作为存储器地址。

4. 数据通路

数据通路主要包括累加器、程序计数器和算术逻辑单元(ALU)。

累加器可以直接用寄存器来完成。图 7.5 给出累加器 A 中的某一位的电路结构。其中，en 信号是由解码器给出的使能信号；输入端口 d 的数据来自数据总线；输出端口 qout 的数据输出到数据总线上。

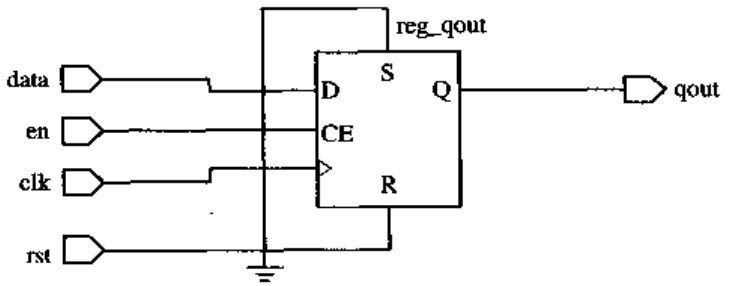


图 7.5　累加器的实现

程序计数器给出下一条要执行的指令的地址(用 pc 表示)。pc 的来源有两个：在正常执行时，每次 pc 地址加 1；在执行跳转指令“jump”时，pc 的值由指令给出。相应的框图如图 7.6 所示。

当 selpc 为 1 时，表明当前指令是跳转指令，pc 的值由指令寄存器给出；当 incpc 为 1

时、pc 的值加 1。

ALU 的框图如图 7.7 所示。ALU 根据指令类型，执行加法、与、异或等操作。

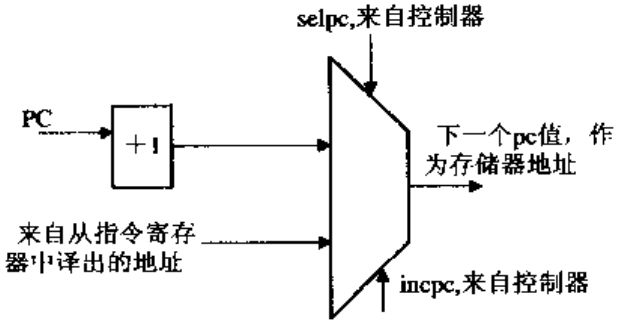


图 7.6　程序计数器的实现

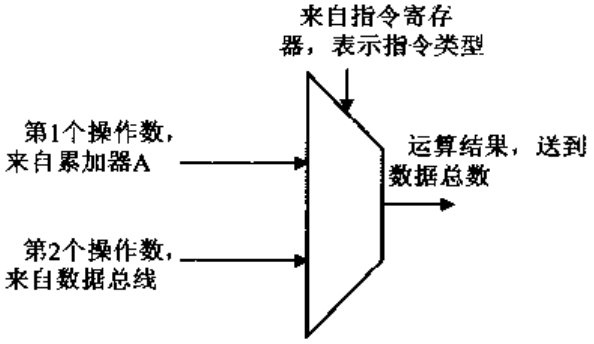


图 7.7　ALU 的框图

ALU 的实现请参见第 4 章的有关描述。

5. **控制器**

控制器是 CPU 中最核心的部分。

在这个例子中，控制器主要包括两部分：指令寄存器和控制信号生成逻辑。

1) 指令寄存器

指令寄存器用一个 8 位寄存器来实现。

2) 控制信号生成逻辑

控制信号生成逻辑的功能包括：

- 生成存储器的读写信号。
- 控制 pc 的赋值。在跳转指令时，pc 的值由指令寄存器给出；其它情况下，由上一个值累加得到。
- 将解码出的操作码送给数据通路的 ALU，控制 ALU 的运行。
- 控制累加器是否要从总线上得到数据。

控制信号跟当前的指令和时序有关。如下式所示：

$$C = T1*(INS1 + INS2 + \cdots) + T2*(INS1 + INS2 + \cdots) + \cdots$$

设计者要根据当前指令的类型和所处的周期来决定哪些控制信号有效，哪些无效。控制信号的时序必须小心定义。这里，我们将一个完整的操作分成 8 个状态。这 8 个状态分别由时钟 clk、clk2x、clk4x 的组合来表示，其中 clk2x 是 clk 的 2 分频，clk4x 是 clk 的 4 分频。相应的分频器模块可用如下方式来实现：

```
module divclock(clk,clk2x,clk4x);
input clk;
output clk2x,clk4x;
reg clk2x,clk4x;

always @(negedge clk)
begin
```

```
{clk4x,clk2x}={clk4x,clk2x}+1;
end

endmodule
```

这个模块产生的波形如图 7.8 所示。在前 4 个状态，主要完成取指令，以程序计数器的值作为存储器的地址。在后 4 个状态，主要完成指令的执行，以指令寄存器的地址部分作为存储器地址。需要说明的是，第 1 个状态开始于时钟的下边沿，第 2 个状态开始于时钟的上边沿……依次类推。

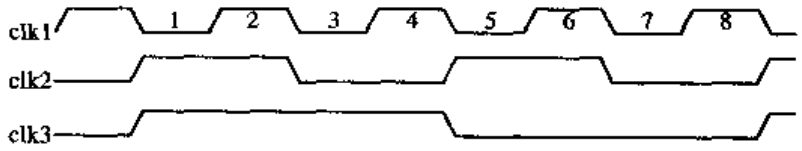


图 7.8　时钟分频

各个状态的作用如表 7.3 所示。

表 7.3　状态时序和控制信号

状态	作用	相应的控制信号
1	清除控制器产生的所有控制信号	控制信号清 0
2	取指令	存储器读信号有效
3	将指令从总线读到指令寄存器中	指令寄存器使能信号有效
4	空闲状态	各控制信号保持原状态
5	使程序计数器的值加 1	程序计数器的使能信号有效
6	对于 add，and，or 和 load 指令，从存储器中取出相应的操作数	存储器读信号有效
7	ALU 进行运算	对于 add、and、xor、load 指令，将第 2 个操作数载入； 对于跳转指令，使得 selpc 信号有效
8	结果写回	对于跳转指令，使得程序计数器使能信号有效，得到下一条指令的地址； 对于存储指令，使得存储器写信号有效，将值写回存储器； 对于 add，and，xor 和 load 指令，使得累加器使能信号有效，将结果存放于累加器中

下面以 add 指令、jump 指令、store 指令为例，对相关控制信号的时序进行解释。

(1) add 指令。对于 add 指令，控制信号时序如下：

第 1 个状态，清除所有控制信号，作用是防止以前指令的影响。

第 2 个状态，mem_rd 信号有效，将指令放到数据总线上。

第 3 个状态，指令寄存器使能信号有效。

第 4 个状态，将指令从数据总线上载入到指令寄存器中(因为所有寄存器是在上升沿采样)。

第 5 个状态，selpc=0，指示 pc 地址加 1。

第 6 个状态，mem_rd 信号有效，此时存储器地址取指令寄存器的地址，因此从存储器中读出的是 M[add]。

第 7 个状态，ALU 计算 a+M[addr]，并将结果送到输出端。这时候，累加器使能信号有效，准备好将输出结果送到累加器中存放。

第 8 个状态，在时钟上升沿来临后，将 ALU 的结果锁存到累加器中。

对 and 与 xor 指令的分析，类似于 and 指令。

(2) jump 指令。前 5 个状态时的控制信号与 add 指令一样。

由于 jump 指令不需要从存储器中读操作数，因此第 6 个状态控制信号无效。

第 7 个状态，使得程序计数器的 selpc 信号有效，用指令寄存器的地址部分更新 pc 的值。

第 8 个状态，inc 信号有效，将 pc 地址加 1。

(3) store 指令。前 5 个状态的控制信号与 add 指令一样。

由于 store 指令不需要从存储器中读操作数，因此第 6 个状态控制信号无效。

第 7 个状态，将累加器的值送到 ALU 的输出端。所有控制信号都无效。

第 8 个状态，mem_wr 有效，将 ALU 的结果送到存储器中。

至此，一个 CPU 就构造成功了。虽然它很简单，但涵盖了 CPU 设计的主要方面。下面我们要将这个 CPU 扩展为 16 位，并增加更多的功能，使它更接近于实用。

7.3 8 位 CPU 的扩展

将 8 位 CPU 扩展为 16 位，除了将 8 位的总线与寄存器变为 16 位，还包括如下几个方面：

- 存储器的扩展；
- 指令集的扩展；
- 数据通路的扩展；
- 控制器的扩展。

下面分别进行说明。

1. 存储器的扩展

前面介绍的 8 位 CPU 中，数据和程序是统一存放的，这种结构称为冯·诺依曼结构。在 16 位 CPU 中，我们可以考虑采用哈佛结构，相应的架构如图 7.9 所示。

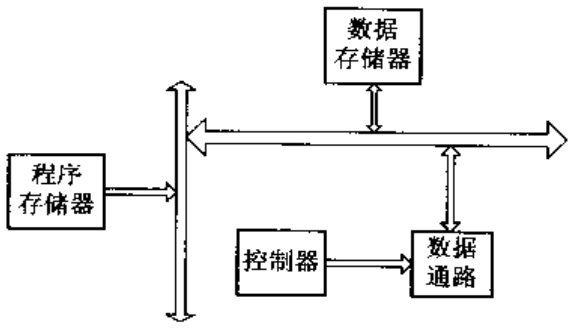


图 7.9 哈佛结构的 16 位 CPU

2. 指令集的扩展

指令集的扩展包括两方面：增加指令类型，增加指令功能。

1) 增加指令类型

在前面设计的 8 位 CPU 中，每个指令都调用存储器，即每种指令都属于寄存器—存储器指令。对存储器的访问要比对寄存器的访问慢 1000 倍。因此，为了提高这个 CPU 的指

令执行速度，需要对原有的指令进行改动，尽可能地从寄存器中取操作数。

我们将指令分为两类：寄存器－寄存器指令和寄存器－存储器指令。寄存器－存储器指令只有两种，一种是 load 指令，一种是 store 指令。load 指令将特定位置的存储器数据读入到寄存器中；store 指令将数据写入到特定位置的存储器中。其它指令都属于寄存器－寄存器指令。这类指令中，数据的来源与目的都是寄存器。需要说明的是，寄存器也可以跟立即数进行操作。立即数是一个确定的数值。

为了实现寄存器－寄存器指令，我们增加了一个寄存器 b，以及寄存器阵列 aregs 和 bregs。寄存器 b 存放操作的第二个操作数。寄存器阵列给寄存器 a 和 b 提供操作数。相关的运算结果要同时写到 aregs 与 bregs 中，所以这两个寄存器阵列所存的内容是一样的。构造两个寄存器阵列的原因，是为了能够同时读取两个操作数(另一种方法是构造多端口的寄存器阵列)。

2) 增加指令功能

在 16 位 CPU 中，指令集的规模增加许多。指令可以分为算术运算指令、逻辑运算指令、移位指令、载入和存储指令、跳转指令、分支指令、调用指令和立即数指令。它们分别在表 7.4～7.11 中进行了说明。

表 7.4　算术运算指令

15～12 位	11～8 位	7～4 位	3～0 位	指令	结果
0	Rd	Ra	Rb	Add rd,ra,rb	R[rd] = r[ra] + r[rb];
1	Rd	Ra	Rb	Sub rd,ra,rb	R[rd] = r[ra] − r[rb];
2	Rd	Ra	Imm	Addi rd,ra,imm	R[rd] = r[ra] + immed;
3	rd	4	Rb	Adc rd,rb	R[rd] = r[rd] + r[rb] + t;
3	rd	5	Rb	Sbc rd,rb	R[rd] = r[rd] − r[rb] − t;
4	rd	4	Imm	Adci rd,imm	R[rd] = r[rd] + immed + t;
4	rd	5	Imm	Sbci rd,imm	R[rd] = r[rd] − immed − t;

其中，immed 表示立即数。t 表示上次操作的进位。

表 7.5　逻辑运算指令

15～12 位	11～8 位	7～4 位	3～0 位	指令	结果
3	rd	0	Rb	And rd,rb	R[rd] = r[rd] & r[rb];
3	Rd	1	Rb	Or rd,rb	R[rd] = r[rd] | r[rb];
3	Rd	2	Rb	Xor rd,rb	R[rd] = r[rd] ⊕ r[rb];
3	Rd	3	Rb	Andn rd,rb	R[rd] = r[rd] & ~r[rb];
4	rd	0	Imm	Andi rd,imm	R[rd] = r[rd] & immed;
4	rd	1	Imm	Ori rd,imm	R[rd] = r[rd] | immed;
4	rd	2	Imm	Xori rd,imm	R[rd] = r[rd] ⊕ immed;
4	rd	3	Imm	Andni rd,imm	R[rd] = r[rd] & ~immed;

表 7.6　移位指令

15～12 位	11～8 位	7～4位	3～0位	指令	结果
4	rd	6	imm	Srli rd,imm	R[rd] = r[rd] >> immed;
4	rd	7	imm	Srai rd,imm	R[rd] = (signed)r[rd] >> immed;
4	rd	8	imm	Slli rd,imm	R[rd] = r[rd] << immed;
4	rd	9	imm	Srxi rd,imm	R[rd] = (t << 16-immed) | (r[rd] >> immed);
4	rd	A	imm	slxi rd,imm	R[rd] = (r[rd] << immed) | (t << (immed-1));

表 7.7　载入和存储指令

15～12位	11～8 位	7～4 位	3～0 位	指令	结果
5	rd	ra	imm	Lw rd,imm(ra)	R[rd] = load_word(r[ra] + immed);
6	rd	ra	imm	Lb rd,imm(ra)	R[rd] = $0_{15\,8}$ || load_byte(r[ra] + immed);
8	rd	ra	imm	Sw rd,imm(ra)	Store_word(r[rd], r[ra] + immed);
9	rd	ra	imm	Sb rd,imm(ra)	Store_byte((Byte)r[rd], r[ra] + immed);

表 7.8　跳转指令

15～12 位	11～8位	7～4位	3～0位	指令	结果
A	rd	ra	imm	jump rd,imm(ra)	target = r[ra] + immed; r[rd] = pc; pc = target;

表 7.9　分支指令

15～12 位	11～8 位	7～4 位	3～0 位	指令	结果
B	0	disp		Branch label	Pc += 2×sign_ext(disp) + 2;
B	2	disp		Branch_eq label	If (==) pc += 2×sign_ext(disp) + 2;
B	3	disp		Branch_ne label	If (!=) pc += 2×sign_ext(disp) + 2;
B	4	disp		Bc label	If (carry_sum(,)) pc += 2×sign_ext(disp) + 2;
B	5	disp		Bnc label	If (!carry_sum(,)) pc += 2×sign_ext(disp) + 2;
B	6	disp		Bv label	If (overflow_sum(,)) pc += 2×sign_ext(disp) + 2;
B	7	disp		Bnv label	If (!overflow_sum(,)) pc += 2×sign_ext(disp) + 2;
B	8	disp		Blt label	If ((signed)<) pc += 2×sign_ext(disp) + 2;
B	9	disp		Bge label	If ((signed)>=) pc += 2×sign_ext(disp) + 2;
B	A	disp		Ble label	If ((signed)<=) pc += 2×sign_ext(disp) + 2;
B	B	disp		Bgt label	If ((signed)>) pc += 2×sign_ext(disp) + 2;
B	C	disp		Bltu label	If ((unsigned)<) pc += 2×sign_ext(disp) + 2;
B	D	disp		Bgeu label	If ((unsigned)>=) pc += 2×sign_ext(disp) + 2;
B	E	disp		Bleu label	If ((unsigned)<=) pc += 2×sign_ext(disp) + 2;
B	F	disp		Bgtu label	If ((unsigned)>) pc += 2×sign_ext(disp) + 2;

注：其中，disp 表示偏移地址。

表 7.10　调用指令

15～12 位	11～8 位	7～4 位	3～0 位	指令	结果
C	Imm12			Call function	R[15] = pc; pc = $imm12_{11:0} \| \| o_{3:0}$;

表 7.11　立即数指令

15～12 位	11～8 位	7～4 位	3～0 位	指令	结果
D	Imm12			Imm imm12	$Immed(next)_{15:4}$ = imm12;

说明：

有许多指令要求处理立即数，例如 addi，该指令里的立即数位数是 4 位。如果要求处理的立即数是 16 位，则在调用该指令之前，要先调用立即数指令 imm，将 12 位立即数先送到寄存器 b 的高 12 位。addi 指令中的 4 位立即数作为整个立即数的低 4 位。

3．数据通路的扩展

在典型的数据通路中，包括寄存器(一般是 1～32 个)、算术逻辑部件 ALU 和连接它们的总线，如图 7.10 所示。寄存器给 ALU 的两个输入寄存器(图中的 A 和 B)提供输入。寄存器的功能是指在 ALU 进行计算时维持 ALU 的输入数据。

ALU 本身对输入数据进行加、减等简单运算，然后将产生的运算结果载入到输出寄存器，经输出寄存器存回到某个寄存器中。以后再需要时还可以从寄存器写回到内存中。

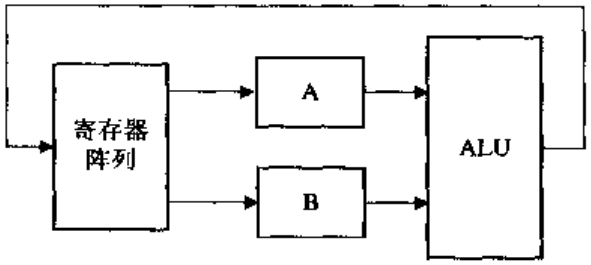


图 7.10　数据通路的一般结构

4．控制器的扩展

在 16 位 CPU 中，我们将采用流水线结构。

流水线的原理在第 1 章已作了介绍。RISC 中的指令都是流水化的。在复杂的 CISC 中，也采用了流水线的结构，以加快指令的执行速度。

图 7.11 给出了一种 5 流水线的 RISC 的流水线结构。图 7.12 给出了在这种 CPU 中指令的执行方式。

而在 16 位 CPU 中，我们将采用三级流水线，即将指令的执行分成取指、解码、执行三个阶段。

流水线的优点是大大提高了指令的执行速度，缺点是使控制变得复杂。

流水线的主要困难是对条件转移(包括分支和跳转)的处理。例如，如果所取的指令是一个条件分支指令 beq，该指令只有在执行阶段才能判断两个操作数是否相同，从而决定程序要跳到何处，下面要执行哪一条指令。由于采用了流水线结构，在分支指令的执行阶段，第 2 条指令已经被取入，正在进行译码；第 3 条指令被读入。如果发生了转移，这时候控制器需要清除那些已取来的无用的指令。解决办法是，增加两个控制信号 ex_annul 和 dc_annul，分别将处于解码阶段的第 2 条指令和处于取指阶段的第 3 条指令取消。

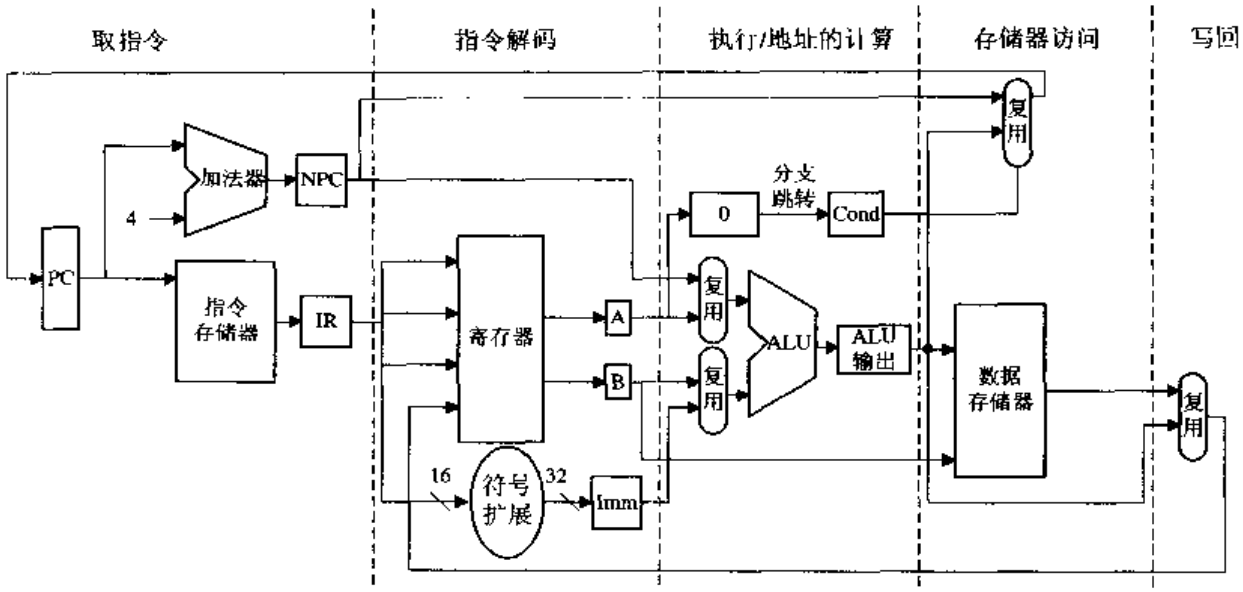


图 7.11　5 流水线的 RISC

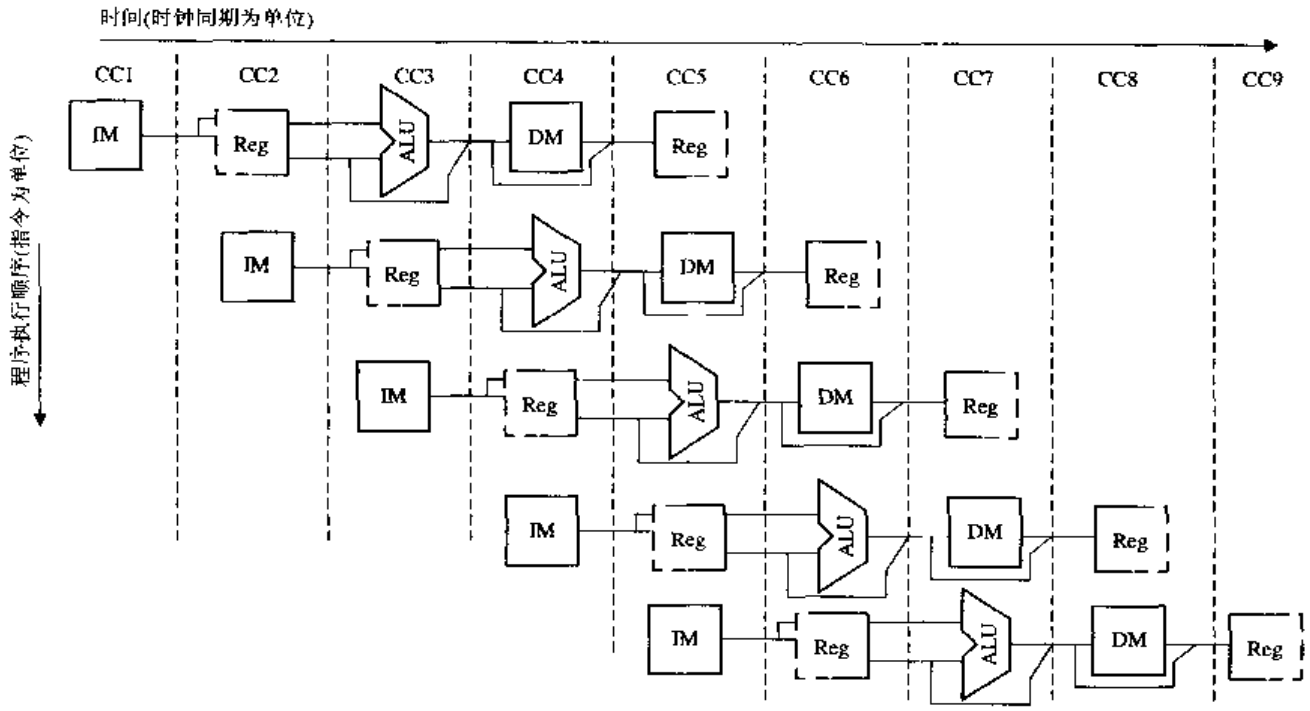


图 7.12　流水线指令的执行

流水线的另一个问题是：如果第 1 条指令对某个寄存器的内容进行了改变，而紧跟其后的第 2 条指令需要用到这个寄存器的内容，这时候可能会发生错误。例如，假设在第 1 个周期开始执行第 1 条指令，将寄存器阵列中的第 3 个寄存器的值加 1，然后将结果送回。寄存器 3 的内容只有在第 3 个周期才被更新。与此同时，在第 2 个周期，第 2 条指令被读入。在第 3 个周期，第 2 个指令要读入操作数。这时候寄存器 3 的内容并未被更新，第 2

条指令读入的是旧值，这会导致错误。

针对这种情况，可采取的解决措施是：如果第 1 条指令的目的寄存器等于第 2 条指令的源寄存器，则在第 3 个周期，一旦得到计算结果，立即将它送到操作数寄存器 a 中，而不必先写回寄存器阵列中。这样，第 2 个指令读入的操作数是更新后的操作数。

7.4 16 位 RISC 的设计

下面给出扩展的 16 位 CPU 的实现，包括架构、数据通路和控制器的实现。

7.4.1 架构

16 位 CPU 的架构如图 7.13 所示，它采用了总线结构。程序存储器位于 CPU 外部，通过外部总线将数据送到内部总线。内部总线连接 CPU 的主要模块(指令预取寄存器、寄存器阵列、ALU、数据存储器)。

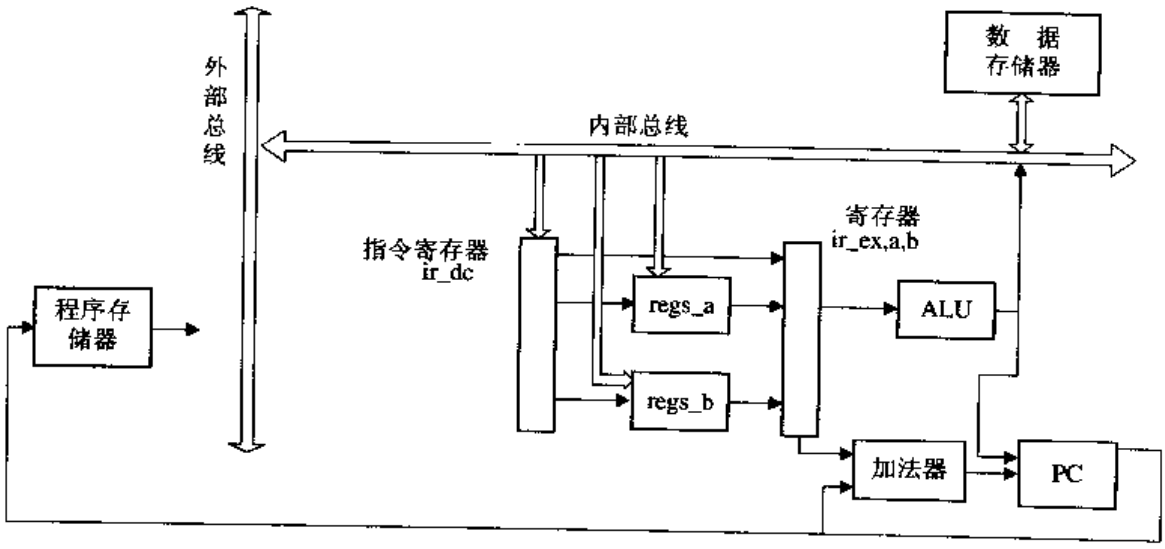


图 7.13 16 位 CPU 的架构

16 位 CPU 中的数据流向如图 7.14 所示。

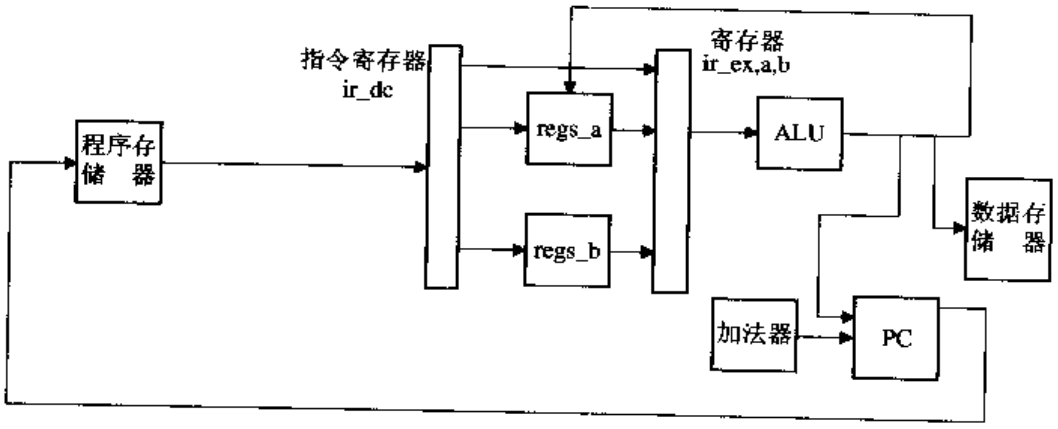


图 7.14 16 位 CPU 中的数据流向

在图 7.14 中，指令执行的过程是：

- 首先，指令从程序存储器中送到指令寄存器 ir_dc 中。这个阶段即取指阶段。
- 然后，根据 ir 的内容，解码出相应的控制信号，例如，rna 与 rnb 的地址选择信号。这个阶段即解码阶段。在此阶段，还要将 ir_dc 的内容送到执行阶段的指令寄存器 ir_ex。
- 最后，根据 ir_ex 的内容，产生执行阶段的控制信号，控制操作数的运算、PC 的更新等(操作数由寄存器 a 和 b 将操作数送到 ALU 单元)。ALU 的计算结果存放到寄存器阵列中；更新的 PC 作为程序存储器的地址。load 指令与 store 指令的执行稍微复杂一些。对于 load 指令，在执行阶段，ALU 得到要读的数据存储器的地址，然后将存储器的数据读到寄存器中。对于 store 指令，在执行阶段，ALU 得到要写的数据存储器的地址，然后将寄存器中的数据写入到数据存储器中。这个阶段即执行阶段。

7.4.2 数据通路的实现

数据通路部分可实现如下功能：

- 加法、逻辑运算、移位等的实现。
- 结果的选择输出。
- 在输出结果时，可以对一些标志位进行设置。例如，结果是否为 0，是否溢出，是否有进位等。
- 更新程序计数器的值。一般情况下，每个指令执行一次 PC+2(因为存储器是 8 位的，2 个 Byte 放一个指令，所以 PC 是偶数的)。如果发生跳转等操作，则需要用 ALU 计算目的地址，然后再送给 PC。如果发生分支操作，则需要将现在的 PC 加偏移地址。

数据通路模块的端口描述如下所示：

```
module datapath(
clk, rst,                    //全局信号
rf_we, rna, rnb,             //寄存器阵列的输入信号
fwd, imm, sextimm4, zextimm4, wordimm4, imm12, pipe_ce, h15_4_ce, //操作数寄存器的输入信号
    add, ci, logicop, sri,   //ALU 及逻辑运算单元的输入
sum_t, logic_t, shl_t, shr_t, zeroext_t, ret_t, ld_t, ud_t, udlt_t,  //输出多路选择器的输入
branch, brdisp, selpc, zeropc, dmapc, pc_ce, ret_ce, //地址/PC 单元的输入
a15, z, n, co, v, addr_nxt, res);          //输出结果
```

表 7.12 对端口信号进行了说明。

表 7.12　数据通路模块的端口信号描述

信号名	I/O	信号流向	说明
Clk	I	顶层模块	时钟
Rst	I	顶层模块	复位号
Rf_we	I	控制模块	寄存器阵列的写使能信号
Rna	I	控制模块	寄存器阵列端口 A 寄存器号
Rnb	I	控制模块	寄存器阵列端口 B 寄存器号
Fwd	I	控制模块	将结果提前送到 A 操作数寄存器的控制信号(在流水线处理器中，指令的运算结果通常写回到寄存器阵列中。如果下一指令要用到上一条指令的运算结果，则需要从寄存器阵列中读出。为加快速度，可以将运算结果直接送到操作数寄存器中)
Imm	I	控制模块	12 位的立即数域
Sextimm4	I	控制模块	4 位立即数(带符号扩展位)
Zextimm4	I	控制模块	4 位立即数(扩展位为 0)
Wordimm4	I	控制模块	字偏移 4 位立即数
Pipe_ce	I	控制模块	流水线时钟使能
B15_4_ce	I	控制模块	B[15:4]时钟使能
Add	I	控制模块	选择加法或减法
Ci	I	控制模块	进位
Logicop[1:0]	I	控制模块	表示各种逻辑操作
Sri	I	控制模块	右移时输入的最高有效位
Sum_t	I	控制模块	加法器输出使能，低电平有效
Logic_t	I	控制模块	逻辑单元输出使能，低电平有效
Shl_t	I	控制模块	左移单元输出使能，低电平有效
Shr_t	I	控制模块	右移单元输出使能，低电平有效
Zeroext_t	I	控制模块	0 扩展输出使能，低电平有效
Ret_t	I	控制模块	返回地址输出使能，低电平有效
Ld_t	I	控制模块	数据低字节输出使能，低电平有效
Ud_t	I	控制模块	数据高字节输出使能，低电平有效
Udlt_t	I	控制模块	数据从高字节输出到低字节的输出使能，低电平有效
Branch	I	控制模块	表示是否有分支发生
Brdisp	I	控制模块	8 位的分支偏移值
Selpc	I	控制模块	地址复用器选择信号，用于选择下一个 PC 值
Zeropc	I	控制模块	使下一个 PC 值为 0
dmapc	I	控制模块	在 PC 寄存器阵列中，使用 DMA 寄存器
Pc_ce	I	控制模块	PC 使能
Ret_ce	I	控制模块	返回地址使能
A15	O	控制模块	A 操作数的最高有效位
Z	O	控制模块	表示结果是否为 0

续表

信号名	I/O	信号流向	说明
N	O	控制模块	表示结果是否为负数
Co	O	控制模块	输出进位
V	O	控制模块	表示结果是否溢出
Addr_nxt	O	顶层模块，最终送到存储控制器中	下一个存储器存取地址
Res	O	片上数据总线	片上数据总线

数据通路模块主要包括寄存器阵列、操作数寄存器、ALU 和程序计数器等部分。下面具体说明各子模块的实现。

1. 寄存器阵列与操作数寄存器

寄存器阵列 aregs 和 bregs。两个寄存器阵列可以用 SRAM 来实现，相应的结构如图 7.15 所示。

其中，Addr 是地址端，用于选择寄存器。WE 是读写控制端，为 1 时，写有效；为 0 时，读有效。

寄存器阵列的控制信号有：寄存器地址，读写控制信号。我们分别用 rna 与 rnb 表示寄存器地址，用 rf_we 表示读写控制信号。

寄存器阵列给 a 和 b 寄存器提供操作数，相应的连接如图 7.16 所示。

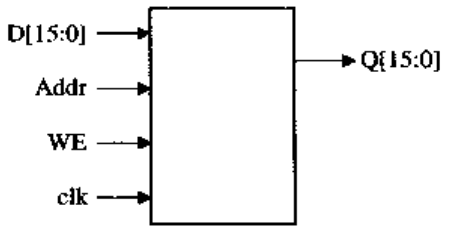


图 7.15　寄存器阵列

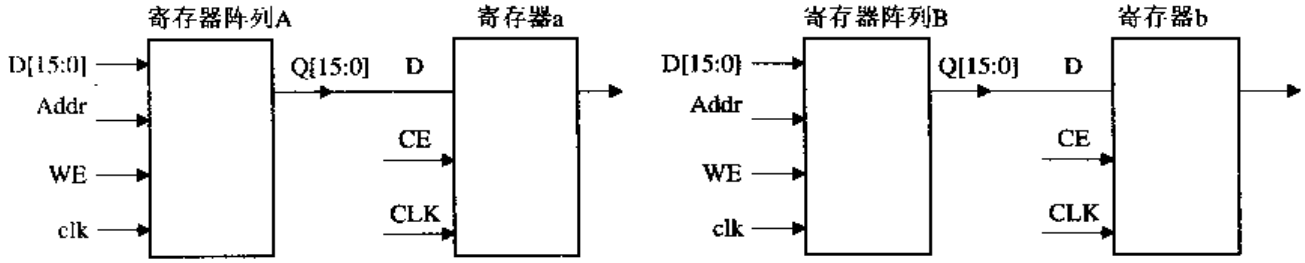


图 7.16　寄存器阵列与操作数寄存器

寄存器阵列的实现代码如下所示：

```
ram16 aregs(.clk(clk), .addr(rna), .wc(rf_wc), .d(result), .o(areg_nxt));
ram16 bregs(.clk(clk), .addr(rnb), .we(rf_we), .d(result), .o(breg_nxt));
always @(negedge clk or posedge rst) begin
    if (rst) begin
        a <= 0;
        b <= 0;
```

```
        end
        else begin
              if(ce)
            a <= areg_nxt;
            b <= breg_nxt;
        end
    end
```

2. ALU

寄存器 a 与 b 将操作数送到 ALU 单元。ALU 可以分成两部分：算术单元与逻辑单元。ALU 的输入包括输入的操作数(来自寄存器 a 与 b)和相应的控制信号(由控制器解码产生)。ALU 根据解码出来的操作码，进行相应运算，将结果送到数据总线上。

在这个扩展的 CPU 中，增加了带进位的加减运算 adc 与 sbc，如下所示：

Adc rd,rb，含义是 r[rd] = r[rd] + r[rb] + t

Sbc rd,rb，含义是 r[rd] = r[rd] – r[rb] - t

其中，t 是上次运算结果所得到的进位。

相应的结构如图 7.17 所示。

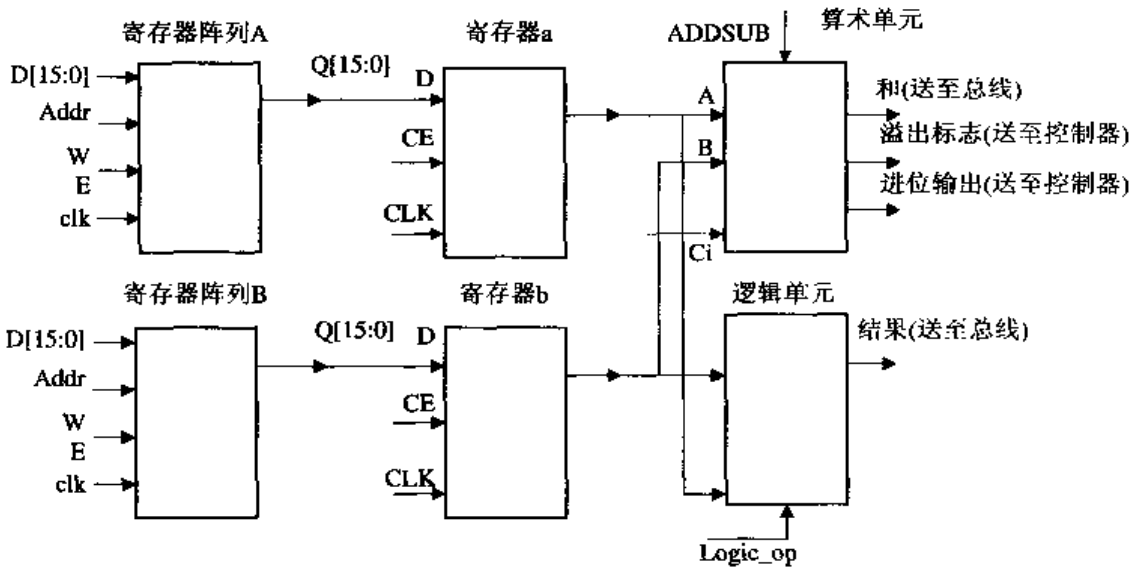


图 7.17　寄存器阵列、寄存器与 ALU

前一次运算的进位输出 cout 送到控制器中。控制器判断当前指令是否是 adc 与 sbc。如果是这两条指令，控制器再根据是加法还是减法，将进位或进位取反后，送到数据通路中，作为本次运算的进位。

在 ALU 中，除了算术和逻辑运算，还要完成逻辑左移、逻辑右移、算术左移、算术右移的功能。逻辑移位与算术移位的差异是：在逻辑右移时，寄存器最高位加 0；在进行算术右移时，符号位跟着移位，并且符号位一直保持原值。

3. **程序计数器**

在 8 位 CPU 中，PC 的取值有两种情况：在正常情况下，PC 地址加 1；在 jump 指令时，

PC 的值由指令给出。

在 16 位 CPU 的指令集中，增加了调用指令和一些分支跳转指令，并对原来的 jump 指令进行了扩展，因此程序计数器的实现变得更复杂了。与程序计数器有关的新指令主要包括 jump 与 call 指令和分支指令。

1) jump 与 call 指令

新的 jump 指令形式是：jump rd,imm(ra)。该指令的功能如下所示：

target = r[ra] + immed; r[rd] = PC; PC = target;

即目标地址由寄存器 a 与立即数相加得到。

指令 call 也会影响 PC 的值。该指令的功能如下所示：

r[15] = PC;　PC = imm||0000;

即将当前的 PC 值赋给第 15 个寄存器，并且将 12 位立即数左移 4 位，作为 PC 的新值。

执行 call 和 jump 指令时，PC 值由加法器或立即数移位得到。也就是说，PC 值来自 ALU 单元。

2) 分支指令

新指令集中增加了许多分支指令，例如 branch 指令。该指令功能如下所示：

PC += 2×sign_ext(disp) + 2;

执行该指令时，当前 PC 值加上偏移值，即可得到下次指令的 PC 值。

执行分支指令时，需要对当前 PC 值进行算术运算。这可以由 ALU 单元的加法器来完成。为了简单起见，也可以为 PC 单独构造加法器。我们采用后一种方法。

综上所述，PC 的来源有三个：jump 与 call 指令来自 ALU；分支类指令由当前 PC 和偏移值相加得到；其它指令由当前指令累加得到。相应的实现结构如图 7.18 所示。

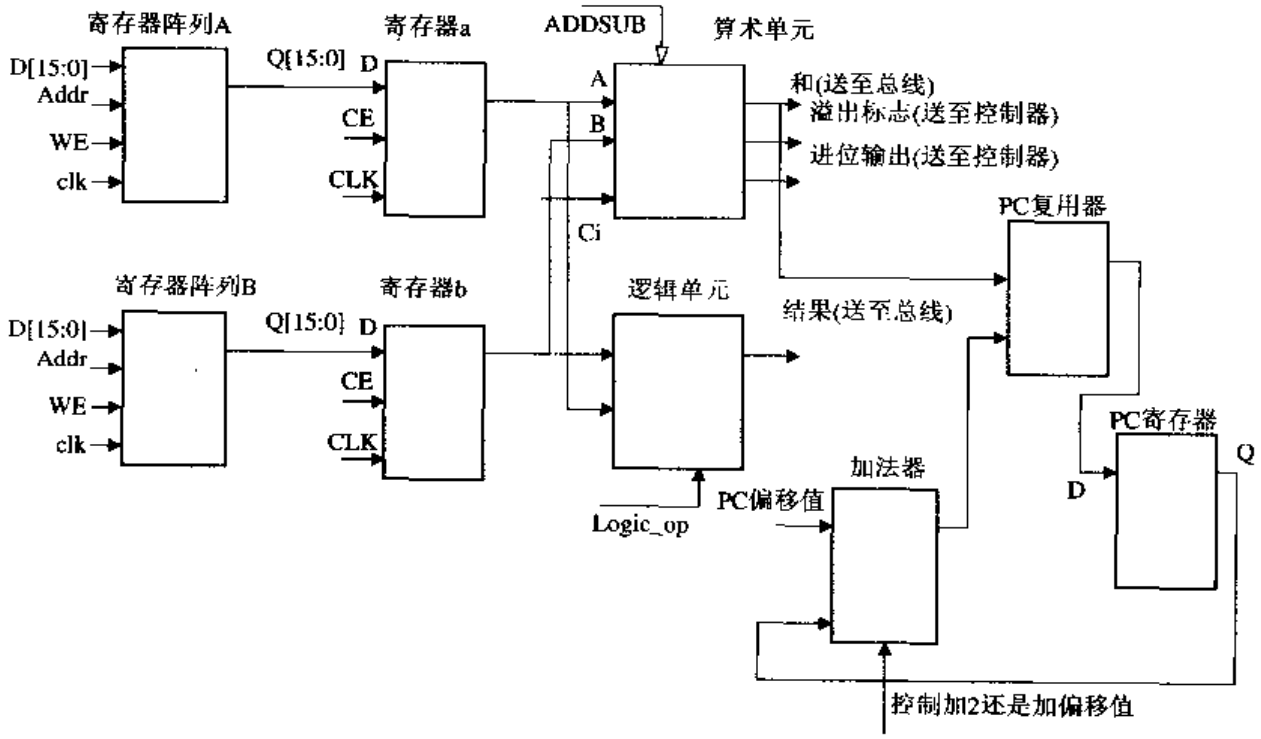


图 7.18　寄存器阵列、操作数寄存器、ALU 和 PC 单元

注意：

需要说明的是，由于现在 CPU 是 16 位的，即指令要占用存储器的 2 个字节，因此每执行一条指令，PC 的地址要加 2，而不是加 1。

程序计数器的实现代码如下所示：

```
// address/PC unit
    always @(branch or pc or brdispext or brdisp or dmapc) begin
        if (branch & ~dmapc)
            pcnext <= pc + {brdispext[6:0],brdisp[7:0],1'b0};
        else
            pcnext <= pc + 2;
    end
    always @(sum or pcnext or selpc or zeropc) begin
        if (zeropc)
            addr_nxt <= 0;
        else if (selpc)
            addr_nxt <= pcnext;
        else
            addr_nxt <= sum;
    end
```

4. 数据通路的输出部分

现在，我们已经能够用数据通路来完成计算，还需要将计算结果(包括算术单元结果 sum、逻辑单元结果 logout、移位操作结果、PC 值)送到总线上，输出控制信号由控制器给出。实现代码如下所示：

```
inout [N:0]          result;          // on-chip data bus
tri                  [N:0] result;
assign result        = sum_o      ? 16'bz : sum;
assign result        = logic_o    ? 16'bz : logic;
assign result        = shl_o      ? 16'bz : { a[14:0],1'b0 };
assign result        = shr_o      ? 16'bz : { sri, a[15:1] };
assign result        = pc_ret     ? 16'bz : pc_add;
……
```

7.4.3 控制器的实现

控制模块是 CPU 的“大脑”，它为系统的其它部分提供控制信号以控制数据的存储及流动。下面对指令流水化、指令解码与控制信号的时序进行说明。

1. 指令流水化

在 16 位 CPU 中，指令执行采用三级流水，即取指、解码和执行。原来单一的指令寄存器现在变成两个：指令寄存器 ir_dc 和指令执行寄存器 ir_ex。

在取指阶段，由程序存储器送来的指令放入到 ir_dc 中。在解码阶段，将 ir_dc 的内容送到 ir_ex 中，并进行解码，将操作数取出。在执行阶段，进行运算。这样就完成了流水化操作。

相应的实现代码如下所示：

```
parameter       IN    = 15; // instruction word msb index
reg [IN:0]  ir_dc, ir_ex;
always @(posedge clk or posedge rst) begin
    if (rst) begin
        ir_dc <= 0;
        ir_ex <= 0;
    end
    else begin
        if (ifetch && rdy&&pipe_ce)
            ir_dc <= insn;
        if (pipe_ce)
            ir_ex <= ir_dc;
        end
end
```

2. 指令解码

在解码阶段，根据指令寄存器的内容，给出指令类型、操作数寄存器地址、目的寄存器地址，并产生相应的控制信号。实现代码如下所示：

```
wire [3:0]  op      = ir[15:12];    // 操作码
wire [3:0]  rd      = ir[11:8];     // 目的寄存器
wire [3:0]  ra      = ir[7:4];      // 源寄存器 A
wire [3:0]  rb      = ir[3:0];      // 源寄存器 B
wire [3:0]  cond    = ir[11:8];     // 分支条件
wire [3:0]  fn      = ir[7:4];      // rr or ri 指令的操作子码
```

对某些指令，子操作码的解码可以在执行阶段完成。例如，对逻辑运算，子操作码的译码可以用如下方式实现：

```
assign    logicop   = ex_ir[5:4];
```

分支操作中的偏移地址的译码可以用如下方式实现：

```
assign    brdisp    = ex_ir[7:0];   // 8 位分支偏移地址
```

3. 控制信号时序

控制信号的生成是 CPU 设计中最核心的部分，也是最困难的地方。只有规划好时序，才能够正确地生成控制信号。

表 7.13 给出了控制器送给数据通路的控制信号。下面对这些控制信号的时序进行讨论。

表 7.13　控制信号说明

控制信号	功能
rf_we	寄存器阵列写使能
rna[3:0]	寄存器阵列端口 A 的寄存器号
rnb[3:0]	寄存器阵列端口 B 的寄存器号
Fwd	将结果总线上的值提前放入 a 操作数寄存器中
Imm[11:0]	ir 的 0~11 位
sextimm4	符号扩展的 4 位立即操作数。当标志操作数包含立即数的逻辑、算术和移位指令时，4 位立即数将送到寄存器中
zextimm4	0 扩展的 4 位立即操作数。指令为 LB 或 SB 时有效
imm12	指令为 call 或 imm 时有效
pipe_ce	流水线使能
ifetch	决定下个周期是否读入新指令
B15_4_ce	1 表示正在执行的指令不是 imm 指令
add	标志加减法。为 1 时，加法；为 0 时，减法
ci	进位
logicop[1:0]	逻辑单元的操作码
sri	移位寄存器的最高有效位输入
Sum_o	低电平有效，加法器输出使能
Logic_o	低电平有效，逻辑单元输出使能
Shl_o	低电平有效，左移输出使能
Shr_o	低电平有效，右移输出使能
Zeroext	低电平有效，0 扩展输出使能
Pc_ret	低电平有效，返回地址输出使能
Ud_t	将数据低字节输出到存储器的使能信号，低电平有效
Ld_t	将数据高字节输出到存储器的使能信号，低电平有效
Udlt_t	将数据从高字节输出到存储器低字节的输出使能，低电平有效
Branch	标志分支的发生
Brdisp[7:0]	8 位的分支偏移量
Selpc	1 表示指令不是 jump 指令
Pc_ce	PC 计数器使能信号
Ret_ce	返回地址使能

1) 算术指令

在流水线结构中，控制信号跟前后指令有关，也就是说指令的序列对最终结果会有影响。设一个指令序列为 1020、b305、2201、2020、1000，由指令集可知，

1020 表示 sub rd,ra,rb，其中，rd 为 0，ra 为 2，rb 为 0。

b305 表示 bne 指令。

2201 表示 addi rd,ra,imm，其中，rd 取 2，ra 取 0，imm 取 1。

2020 表示 addi rd,ra,imm，其中，rd 为 0，ra 为 2，imm 为 0。

1000 表示 sub rd,ra,rb，其中，rd 为 0，ra 为 0，rb 为 0。

下面以 2201 指令为例，说明算术运算指令的控制信号时序。时序如表 7.14 所示。

表 7.14　addi 指令的控制信号时序

控制信号	取指周期	解码周期	执行周期
rf_we	0	0	1
rna[3:0]前半周期	0	0(2201 的源寄存器地址)	2(目的寄存器地址)
rna[3:0]后半周期	0	3(b305 的目的寄存器地址)	2(目的寄存器地址)
rnb[3:0]前半周期	5(b305 的低 4 位)	1(2201 的低 4 位)	0(2020 的低 4 位)
rnb[3:0]后半周期	0(1020 的目的寄存器地址)	3(b305 的目的寄存器地址)	2(2201 的目的寄存器地址)
Fwd	0	0	1
Imm[11:0]	305	201	020
sextimm4	0	1	1
zextimm4	0	0	0
imm12	0	0	0
pipe_ce	1	1	1
ifetch	1	1	1
B15_4_ce	1	1	1
add	0	1	1
ci	1	0	0
logicop[1:0]	10	00	00
sri	0	0	0
Sum_o	0	1	0
Logic_o	1	1	1
Shl_o	1	1	1
Shr_o	1	1	1
Zeroext	1	1	1
Pc_ret	1	1	1
Ud_t	1	1	1
Ld_t	1	1	1
Udlt_t	1	1	1
Branch	0	0	0
Brdisp[7:0]	当前 ex_ir 的 0～7 位，这里是 20	05	01
Selpc	1	1	1
Pc_ce	1	1	1
Ret_ce	1	1	1

2) 分支指令

下面以 Branch 指令“b004”指令为例，说明分支指令相应的控制信号时序。设指令执行序列为 103f、b303、b004、0000、0000。时序如表 7.15 所示。由表可见，在执行周期，控制信号 Branch 变为有效。

表 7.15 Branch 指令下控制信号时序

控制信号	取指周期	解码周期	执行周期
rf_we	0	0	0
Rna[3:0]前半周期	0	0	0
Rna[3:0]后半周期	0	3	0
Rnb[3:0]前半周期	3	4	0
Rnb[3:0]后半周期	0	3	0
Fwd	0	0	0
imm[11:0]	303	004	000
sextimm4	0	0	0
zextimm4	0	0	0
imm12	0	0	0
pipe_ce	1	1	1
ifetch	1	1	1
B15_4_ce	1	1	1
add	0	1	1
ci	1	0	0
logicop[1:0]	11	00	00
Sri	0	0	0
Sum_o	0	1	1
Logic_o	1	1	1
Shl_o	1	1	1
Shr_o	1	1	1
Zeroext	1	1	1
Pc_ret	1	1	1
Ud_t	1	1	1
Ld_t	1	1	1
Udlt_t	1	1	1
Branch	0	0	1
Brdisp[7:0]	3f	03	04
Pc_ce	0	0	0
Ret_ce	1	1	1

3) Store 指令

执行 Store 指令时，数据从 regsb 中取出，通过总线送到数据存储器中。存储器的地址由 ALU 给出。由于存储器执行指令要比寄存器慢许多，所以 Store 指令无法在 3 个周期内

完成。

解决方法是：停掉流水线，将已读入的新指令送到另一个寄存器(ir_fetch)中暂存起来，并且不再从程序存储器中读指令，直到 Store 指令执行完毕，并且 ir_fetch 中的指令也已送到 ir 中，才继续从程序存储器中读新指令。

控制流水线开关的控制信号是 pipe_ce。控制是否读取新指令的控制信号是 ifetch。

设指令序列为 210d、d000、810c、d000、520c。其中，210d 是 addi 指令，d000 是 imm 指令，810c 是 sw 指令，520c 是 lw 指令。Store 指令执行过程如图 7.19 所示。

由图 7.19 可见，在执行 sw 指令(810c)时，总共需要 6 个周期。在周期 3，下一条指令 d000 一直放在 ir_dc 中，在周期 7 才执行。而指令 520c 在第 3 周期放入到 ir_fetch 中，直到周期 7 才送入 ir_dc 中，并进行解码。

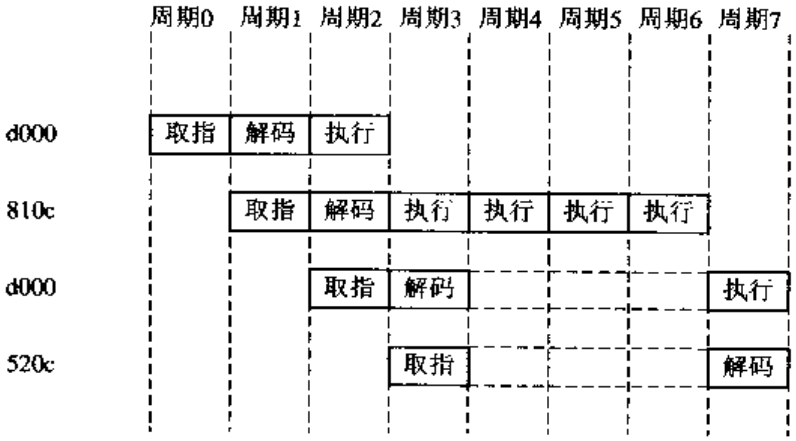


图 7.19　Store 指令执行过程

对于 810c 指令，相应控制信号的时序如表 7.16 所示。

表 7.16　Store 指令的控制信号时序

控制信号	取指	解码	执行				
	1	2	3	4	5	6	7
rf_we	1	0	0	0	0	0	0
rna[3:0](前半周期)	0 1	0	0 1	0 1	0 1	0 1	0 0
Rna[3:0](后半周期)	1	0	1	1	1	1	0
rnb[3:0](前半周期)	0	1	0	0	0	0	C
rnb[3:0](后半周期)	1	0	1	1	1	1	0
Fwd	0	0	0	0	0	0	0
imm[11:0]	000	10c	000	000	000	20c	000
sextimm4	0	0	0	0	0	0	0
zextimm4	0	0	0	0	0	0	0
imm12	1	0	1	1	1	1	0
pipe_ce	1	1	0	0	0	1	1
ifetch	1	1	1	0	0	0	1

续表

控制信号	取指	解码	执行				
	1	2	3	4	5	6	7
B15_4_ce	1	0	0	0	0	1	0
add	1	1	1	1	1	1	1
ci	0	0	0	0	0	0	0
logicop[1:0]	0	0	0	0	0	0	0
Sri	0	0	0	0	0	0	0
Sum_o	0	1	1	1	1	1	1
Logic_o	1	1	1	1	1	1	1
Shl_o	1	1	1	1	1	1	1
Shr_o	1	1	1	1	1	1	1
Zeroext	1	1	1	1	1	1	1
Pc_ret	1	1	1	1	1	1	1
Ud_t	1	1	1	0	0	0	1
Ld_t	1	1	1	0	0 1	1	1
Udlt_t	1	1	1	1	1 0	0	1
Branch	0	0	0	0	0	0	0
Brdisp[7:0]	0d	00	0c	0c	0c	0c	00
Selpc	1	1	0	1	1	1	1
Pc_ce	1	1	0	0	0	1	1
Ret_cc	1	1	0	0	0	1	1

4) Load 指令

执行 Load 指令时，从存储器读出数据，存放到 rd 中。存储器的地址由 ALU 给出。

Load 指令也无法在 3 个周期内完成。在执行时，可以通过使 pipe_ce 和 ifetch 信号变低一个周期，从而使执行周期延长为 2 个周期来完成。

假设指令序列为 b34e、d000、620c、dff5 和 2025。其中，b34e 是 bne 指令，d000 是 imm 指令，620c 是 lb 指令，dff5 是 imm 指令，2025 是 addi 指令。指令序列 d000、620c、dff5 的执行情况如图 7.20 所示。

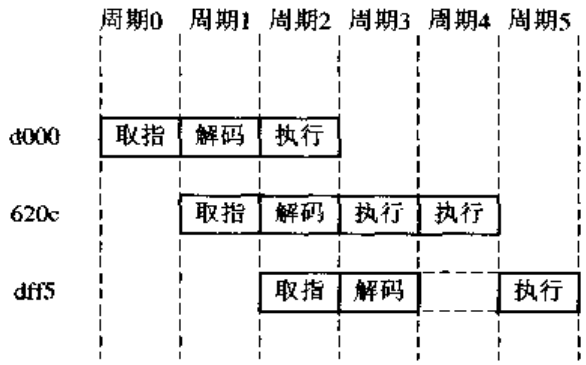


图 7.20　Load 指令的执行序列

620c 指令的相应控制信号时序如表 7.17 所示。

表 7.17　Load 指令相关的控制信号时序

控制信号	取指	解码	执行				
	1	2	3	4	5	6	7
rf_we	0	0	0	1	0	0	0
rna[3:0](前半周期)	0	0	f	f	2	4	0
Rna[3:0](后半周期)	3	0	2	2	f	0	3
rnb[3:0](前半周期)	0	C	5	5	5	9	0
rnb[3:0](后半周期)	3	0	2	2	f	0	3
fwd	0	0	0	0	0	0	0
imm[11:0]	000	20c	Ff5	Ff5	025	349	000
sextimm4	0	0	0	0	1	0	0
zextimm4	0	1	0	0	0	0	0
imm12	1	0	1	1	0	0	1
pipe_ce	1	1	0	1	1	1	1
ifetch	1	1	1	0	1	1	1
B15_4_ce	1	0	0	1	0	1	1
add	1	1	1	1	1	1	1
ci	0	0	0	0	0	0	0
logicop[1:0]	0	0	0	0	3	2	0
Sri	0	0	0	0	0	0	0
Sum_o	1	1	1	1	1	0	1
Logic_o	1	1	1	1	1	1	1
Shl_o	1	1	1	1	1	1	1
Shr_o	1	1	1	1	1	1	1
Zeroext	1	1	0	0	1	1	1
Pc_ret	1	1	1	1	1	1	1
Ud_t	1	1	1	1	1	1	1
Ld_t	1	1	1	1	1	1	1
Udlt_t	1	1	1	1	1	1	1
Branch	0	0	0	0	0	0	0
Brdisp[7:0]	4e	00	0c	0c	F5	25	49
Selpc	1	1	0	1	1	1	1
Pc_ce	1	1	0	1	1	1	1
Ret_ce	1	1	0	1	1	1	1

5) Call 指令

设 Call 指令序列为 a200、2f00、c131、b01f、0000。其中，a200 为 jal 指令，2f00 为 addi 指令，c131 为 Call 指令，b01f 为 Branch 指令，0000 为 add 指令。c131 指令相应的控制信号的时序如表 7.18 所示。

表 7.18 Call 指令的控制信号时序

控制信号	取指	解码	执行
rf_we	0	1	1
rna[3:0](前半周期)	0	0	1
Rna[3:0](后半周期)	2	F	f
rnb[3:0](前半周期)	0	1	f
rnb[3:0](后半周期)	2	F	f
fwd	0	0	0
imm[11:0]	f00	131	01f
sextimm4	1	0	0
zextimm4	0	0	0
imm12	0	1	0
pipe_ce	1	1	1
ifetch	1	1	1
B15_4_ce	1	1	1
add	1	1	1
ci	0	0	0
logicop[1:0]	0	0	3
Sri	0	0	0
Sum_o	1	0	1
Logic_o	1	1	1
Shl_o	1	1	1
Shr_o	1	1	1
Zeroext	1	1	1
Pc_ret	0	1	0
Ud_t	1	1	1
Ld_t	1	1	1
Udlt_t	1	1	1
Branch	0	0	0
Brdisp[7:0]	00	00	31
Selpc	1	1	0
Pc_ce	1	1	1
Ret_ce	1	1	1

针对 CPU 所执行的所有指令，给出详细的控制信号时序表，这样就可以很容易地构造出控制器。下面给出部分控制信号的生成代码。

```
assign imm12     = op==`CALL || op==`IMM;
assign sextimm4  = op==`ADDI || op==`RI;
assign zextimm4  = op==`LB || op==`SB;
assign wordimm4  = op==`LW || op==`SW || op==`JAL;
assign           imm        = ir[11:0];        // 12-bit immediate field
```

```
assign          brdisp      = ex_ir[7:0];      // 8-bit branch displacement
assign pipe_ce  = rdy&if_nxt;
assign pc_ce    = rdy&if_nxt;
assign ret_ce   = rdy&if_nxt;
assign selpc    = if_nxt&~jump;
```

7.5 商业 RISC 介绍

RISC 处理器中，最著名的是 ARM 和 MIPS。下面分别进行介绍。

7.5.1 ARM 体系结构及实现

ARM 是应用最普遍的 RISC 结构之一。它体现了 RISC 典型的技术特点，如大的寄存器阵列、硬布线方式实现译码及控制逻辑、简单的寻址方式和采用多级流水技术等。当前，用得较多的 ARM 结构为 ARM7 与 ARM9。

图 7.21 给出了 ARM7 的结构。

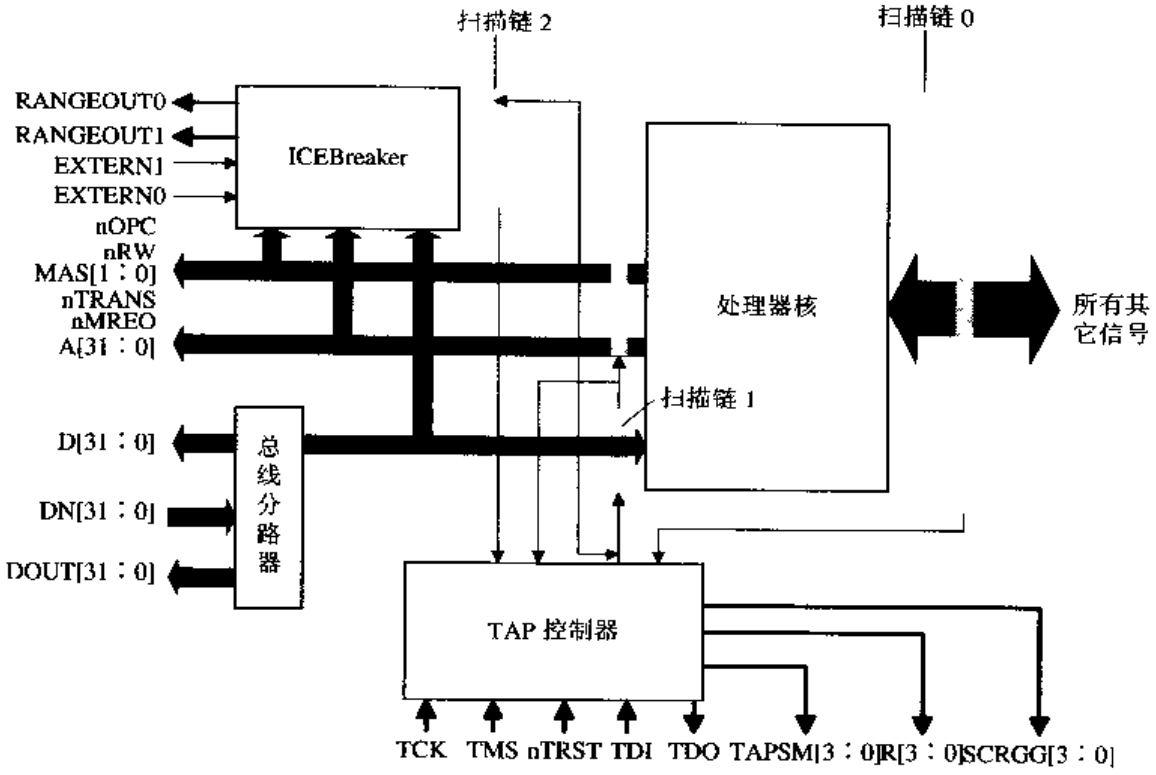


图 7.21 ARM7 的框图

ARM 中的指令可分为六大类：

- 数据处理指令。这类指令的第一操作数为寄存器，第二操作数是寄存器或立即数，可通过 ALU 对操作数完成加、减等算术运算和与、或等逻辑运算。另外，还有无符号数和有符号数的乘法和乘加运算指令。

- 状态寄存器的传送指令。这类指令可以在状态寄存器和通用寄存器之间传送数据，以改变通用寄存器的值或影响状态寄存器的一些标志位。
- 跳转指令。这类指令可实现 PC 指针的跳转，如普通的跳转指令(B)、跳转连接指令(BL)和跳转交换指令(BX)。BL 指令在跳转同时可保存返回地址值，BX 指令则在跳转的同时完成 ARM 指令集和 16 位 THUMB 指令集之间的切换。
- 存储器存取指令。这类指令可实现单个或多个存储器和寄存器之间的数据读写操作，或单个寄存器和存储器单元的值的数据交换(SWP)。
- 协处理器指令。这类指令包括可实现协处理器内部数据处理的指令，协处理器和主处理器的寄存器之间或与存储器的数据传送指令。
- 中断处理指令。这类指令包括软件中断指令(SWI)和对外部复位、中断请求、存储器访问失败、未定义指令等进行的相应处理。

ARM 中采用了两相非交迭时钟的时钟方案，以避免电路中的竞争。

在 ARM 中，对数据通路部分都进行了优化，以提高性能。例如，加法采用进位判决加法器；移位采用交叉开关矩阵来实现桶式移位器；乘法采用 Booth 乘法器。

下面给出 ARM7 的一个行为模型。

```
`define EACH_CLOCK
`define PROGRAM4
`include "clock.v"
// *************** STATE MNEMONICS *************
`define NUM_STATE_BITS 9

`define IDLE       9'b000000000
`define INIT       9'b111111111

`define F1         9'b000100000

`define PC         r[15]
`define RD         r[ir2[15:12]]
`define OPA        r[ir2[19:16]]
`define OPB        (ir2[25] ? ir2[7:0] : r[ir2[3:0]])
`define OFFSET4 {ir2[23],ir2[23],ir2[23],ir2[23],ir2[23],ir2[23],ir2[23:0],
                 2'b00}
module   arm7_system(cont,sysclk);
  input cont,sysclk;
  wire cont;
  wire sysclk;

  reg [31:0] ir1,ir2,psr;
  reg halt;
```

```
reg [31:0] m[31:0];
reg [31:0] r[15:0];

reg [`NUM_STATE_BITS-1:0] present_state;

function condx;
  input [3:0] condtype;
  input [31:0] psr;
  begin
    if (condtype == 4'b1110)
      condx = 1;
    else if (condtype == 4'b0100)
      condx = psr[31];
    else if (condtype == 4'b0101)
      condx = !psr[31];
    else
      condx = 0;
  end
endfunction

function [31:0] dp;
  input [3:0] opcode;
  input [31:0] opa,opb;
  begin
    if      (opcode == 4'b0010)
      dp = opa - opb;
    else if (opcode == 4'b0100)
      dp = opa + opb;
    else if (opcode == 4'b1101)
      dp = opb;
    else
      begin
        dp = 0;
        $display("other DP instructions...");
      end
  end
endfunction
```

```
function [31:0] f;
  input [31:0] dpres;
  begin
    f = dpres & 32'h80000000;
  end
endfunction


always
 begin
   @(posedge sysclk) enter_new_state(`INIT);
    `PC <= @(posedge sysclk) 0;
    halt <= @(posedge sysclk) 1;
    psr <= @(posedge sysclk) 0;
    forever
      begin
        @(posedge sysclk) enter_new_state(`F1);
          if (halt)
           begin
            $stop;
            while (~cont)
              begin
                @(posedge sysclk) enter_new_state(`IDLE);
                halt <= @(posedge sysclk) 0;
                ir1 <= @(posedge sysclk) 32'hf0000000;
                ir2 <= @(posedge sysclk) 32'hf0000000;
              end
           end
          else
           begin
              if (condx(ir2[31:28],psr) &&
                    ((ir2[27:25] == 3'b101)
                      || (ir2[27:26] == 2'b00 && ir2[15:12] == 4'b1111)))
                begin
                  ir1 <= @(posedge sysclk) 32'hf0000000;
                  ir2 <= @(posedge sysclk) 32'hf0000000;
                end
              else
                begin
```

```
                `PC <= @(posedge sysclk) `PC + 4;
                ir1 <= @(posedge sysclk) m[`PC>>2];
                ir2 <= @(posedge sysclk) ir1;
              end

            if (condx(ir2[31:28],psr))
              begin
                if        (ir2[27:26] == 2'b00)
                  begin
                    `RD <= @(negedge sysclk) dp(ir2[24:21],`OPA,`OPB);
                    if (ir2[20])
                     psr <= @(posedge sysclk) f(dp(ir2[24:21],`OPA,`OPB));
                  end
                else if (ir2[27:25] == 3'b101)
                  begin
                    `PC <= @(posedge sysclk) `PC + `OFFSET4;
                  end
                else if (ir2[27:24] == 4'b1111)
                  begin
                      halt <= @(posedge sysclk) 1;
                  end
                else
                  $display("other instructions...");
              end
          end
      end
  end

  task enter_new_state;
      input [`NUM_STATE_BITS-1:0] this_state;
      begin
          present_state = this_state;
          #1;
      end
  endtask

  `ifdef EACH_CLOCK
    always @(posedge sysclk) #1
      begin
```

```
            $display("PC=%h IR1=%h IR2=%h N=%b    %d",`PC,ir1,ir2,psr[31],$time);
            $display("    r0=%h r1=%h r2=%h r3=%h r4=%h",r[0],r[1],r[2],r[3],r[4]);
        end
    `endif
endmodule


module top;
    wire sysclk;
    reg cont;

    cl clock(sysclk);
    arm7_system arm7_machine(cont,sysclk);

    reg [31:0] mi,i;
    initial
      begin
        `ifdef PROGRAM1
          //tests R15 as source and destination
          arm7_machine.m[0] = 32'he3b00000;    //e3b00000    MOVS    R0,0
          arm7_machine.m[1] = 32'he08f1000;    //e08f1000    ADD     R1,R15,R0
          arm7_machine.m[2] = 32'he080200f;    //e080200f    ADD     R2,R0,R15
          arm7_machine.m[3] = 32'he08f3000;    //e08f3000    ADD     R3,R15,R0
          arm7_machine.m[4] = 32'he080400f;    //e080400f    ADD     R4,R0,R15
          arm7_machine.m[5] = 32'he3b0e0ff;    //e3b0e0ff    MOVS    R14,0xff
          arm7_machine.m[6] = 32'he2400008;    //e2400008    SUB     R0,R0,8
          arm7_machine.m[7] = 32'he1a0f001;    //e1a0f001    MOV     R15,R1
          arm7_machine.m[8] = 32'heafffffe;    //eafffffe    B       0xfffffe
        `endif

        `ifdef PROGRAM2
          //another R15 test
          arm7_machine.m[0] = 32'he3b01000;
          arm7_machine.m[1] = 32'he08f1001;
          arm7_machine.m[2] = 32'he3a02000;
          arm7_machine.m[3] = 32'he082200f;
          arm7_machine.m[4] = 32'he3b00000;
          arm7_machine.m[5] = 32'he08f3000;
          arm7_machine.m[6] = 32'he080400f;
```

```
    arm7_machine.m[7] = 32'heafffffe;
`endif


`ifdef PROGRAM3
    //division with conditional ADD SUB
    //div8

    arm7_machine.m[0] = 32'he3a0100e;    //e3a0100e    MOV     R1,0x0e
    arm7_machine.m[1] = 32'he3a04007;    //e3a04007    MOV     R4,0x07
    arm7_machine.m[2] = 32'he3a02000;    //e3a02000    MOV     R2,0x00
    arm7_machine.m[3] = 32'he0511004;    //e0511004    L1 SUBS R1,R1,R4
    arm7_machine.m[4] = 32'h52822001;    //52822001    ADDPL   R2,R2,0x01
    arm7_machine.m[5] = 32'h50511004;    //50511004    SUBPLS  R1,R1,R4
    arm7_machine.m[6] = 32'h52822001;    //52822001    ADDPL   R2,R2,0x01
    arm7_machine.m[7] = 32'h5affffa;     //5affffa     BPL     L1
    arm7_machine.m[8] = 32'hef000000;    //ef000000    L2      SWI
`endif


`ifdef PROGRAM4
    //division with cond branch at top and bot
    //analogous to ch9 on pdp8 (R0 act like AC)
    //div7

    arm7_machine.m[0] = 32'he3a0100e;    //e3a0100e    MOV     R1,0x0e
    arm7_machine.m[1] = 32'he3a02000;    //e3a02000    MOV     R2,0x00
    arm7_machine.m[2] = 32'he3a04007;    //e3a04007    MOV     R4,0x07
    arm7_machine.m[3] = 32'he0510004;    //e0510004    SUBS    R0,R1,R4
    arm7_machine.m[4] = 32'h4a000003;    //4a000003    BMI     L2
    arm7_machine.m[5] = 32'he1a01000;    //e1a01000    L1 MOV  R1,R0
    arm7_machine.m[6] = 32'he2822001;    //e2822001    ADD     R2,R2,0x01
    arm7_machine.m[7] = 32'he0510004;    //e0510004    SUBS    R0,R1,R4
    arm7_machine.m[8] = 32'h5affffb;     //5affffb     BPL     L1
    arm7_machine.m[9] = 32'hef000000;    //ef000000    L2      SWI
`endif


`ifdef PROGRAM5
```

```
    //div3
    //division with cond branch at top, uncond at bot
    //analogous to ch8 on pdp8

    arm7_machine.m[0] = 32'he3a0100e;    //e3a0100e    MOV     R1,0x0e
    arm7_machine.m[1] = 32'he3a04007;    //e3a04007    MOV     R4,0x07
    arm7_machine.m[2] = 32'he3a02000;    //e3a02000    MOV     R2,0x00
    arm7_machine.m[3] = 32'he0510004;    //e0510004    L1 SUBS R0,R1,R4
    arm7_machine.m[4] = 32'h4a000002;    //4a000002    BMI     L2
    arm7_machine.m[5] = 32'he1a01000;    //e1a01000    MOV     R1,R0
    arm7_machine.m[6] = 32'he2822001;    //e2822001    ADD     R2,R2,0x01
    arm7_machine.m[7] = 32'heafffffa;    //eafffffa    B       L1
    arm7_machine.m[8] = 32'hef000000;    //ef000000    L2      SWI
  `endif

  cont = 0;
  #200 cont = 1;
  #100 cont = 0;
  #400 wait(arm7_machine.halt);
  for (i=0; i<=8; i = i+1)
    begin
      mi = arm7_machine.m[i];
      $display("%h/%h",i,mi);
    end
  $stop;
end

endmodule
```

7.5.2 MIPS 体系结构

MIPS 也是一种很著名的 RISC。其体系结构有如下特点：

(1) 指令系统简单。MIPS 所有的指令都是 32 位宽度。指令数总共 6 类。指令格式共三种，即立即数型、转移型和寄存器型。操作数寻址方式有基址加 16 位位移量的访存寻址、立即数寻址及寄存器寻址三种。

(2) 不采用硬件流水互锁。MIPS 依靠优化编译器进行指令序列的重新安排，以防止流水线中出现的相互冲突。

(3) 使用较多寄存器。MIPS 中有 32 个通用寄存器、一对存储 64 位数据的寄存器 Hi,Lo 以及异常 PC 寄存器 epc。32 个通用寄存器表示为$0 到$31，其中$0 固定为 0。Hi,Lo 寄存

器用于存放定点乘法的结果。

(4) 采用“比较与转移”指令。比较和转移这两个动作在一条指令内便可完成，如 beq $1, $2, 1000。MIPS 没有状态寄存器。

图 7.22 给出了一个 MIPS 的体系结构。

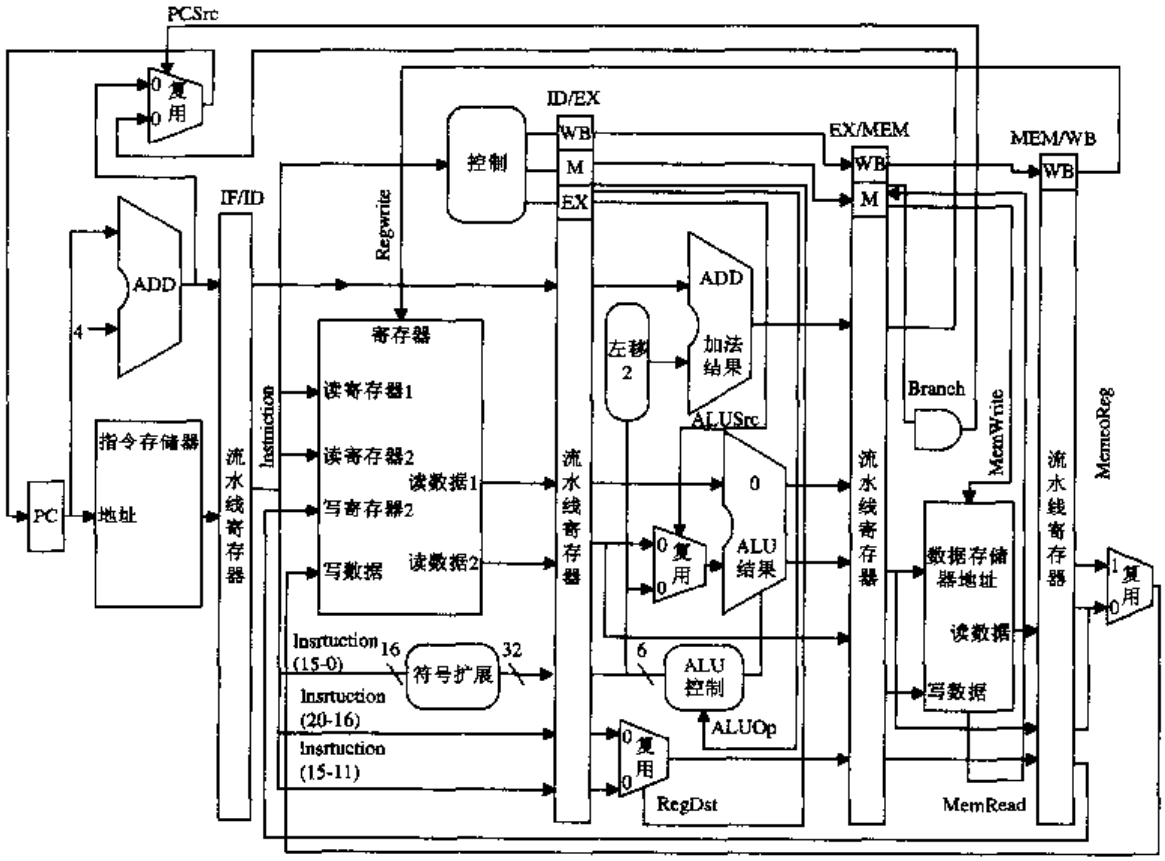


图 7.22 MIPS 处理器的系统结构

在该体系结构中，流水线分为五个阶段，以加(ADD)指令为例进行说明：

第一阶段，取指令。

取指令时，将 PC(程序计数器的内容)所指的指令存储器的内容取出，送到指令寄存器中，并将 PC 加 4(因为这是 32 位的系统，存储器是 8 位的，32=4×8)。

IR = Memory[PC]

PC = PC + 4

在第二个阶段，完成如下操作：

A = R[IR[25:21]]

B = R[IR[20:16]]

在第三个阶段，ALUoutput = A + B：

ALUoutput = A + B

在第四个阶段，执行空操作(对于读指令，要在这个阶段完成读存储器的操作；一般指令在此阶段执行空操作)。

在第五个阶段，将运算结果由 MemtoReg 寄存器写回到数据寄存器：

R[IR[15:11]] = ALUoutput

在介绍过 RISC 的设计之后，我们要涉足另一个领域：CISC(复杂指令集计算机)的设计。

7.6 RISC 与 CISC

按指令类型来分，微处理器可分为 CISC 与 RISC 两类。前者如 Intel 的 x86 系列，后者如 ARM，MIPS 等等。

CISC 即复杂指令集计算机，有如下特点：

- 指令系统复杂。具体表现在指令数多、寻址方式多、指令格式多。
- 绝大多数指令需要多个时钟周期才能执行完成。
- 各种指令都可访问存储器。
- 采用微程序控制。
- 有专用寄存器。
- 难以用优化编译生成高效的目标代码程序。

RISC 即精简指令集计算机，具有如下特点：

- 简化的指令系统。表现在指令数较少、基本寻址方式少、指令格式少、指令字长度一致。
- 以寄存器—寄存器方式工作。
- 以流水方式工作，从而可在一个时钟周期内执行完毕。
- 使用较多的通用寄存器以减少访存，不设置或少设置专用寄存器。
- 采用由阵列逻辑实现的组合电路控制器，不用或少用微程序。
- 采用优化编译技术，保证流水线畅通，对寄存器分配进行优化。

几乎所有讨论体系结构的书都证明了 RISC 要优于 CISC，但这种性能上的优势为什么没有导致市场上的优势呢？

事实上，技术优势常常并不意味着市场份额的优势。由于许多公司的软件都是基于 Intel 的 x86 系列的，因而要全部移植到 RISC 上，代价太高。

这是否意味着 RISC 就不重要了呢？并非如此。实际上，现代的 CISC 融合了许多 RISC 的思想。从 486 开始，Intel 的 CISC 就开始包含一些 RISC 特性，常用的简单指令用 RISC 来实现，而复杂指令用 CISC 方法来实现。这种混合方案不如纯 RISC 快，但却能在不加修改地运行旧软件的前提下给出极具竞争力的整体性能。

现代处理器设计的一些基本思想包括：

- 所有指令由硬件直接执行(而不再由微指令解释一遍)。
- 最大限度地提高指令启动速度。
- 指令应易于译码。

- 只允许读写内存指令访问内存(从内存中读取指令是速度的瓶颈)。
- 提供了足够多的寄存器(寄存器速度远大于存储器)。

7.7 8051 基础

微处理器分为通用微处理器和嵌入式微处理器两种。嵌入式微处理器通常又称为微控制器(Microcontroller)。微控制器的应用非常广泛。其中，最知名的是 Intel 20 年前推出的 MCS 51(这是一种应用很广的 8 位微控制器系列，8051 是其中的一种)。在这款微控制器芯片上，集成了中央处理器单元 CPU、数据存储器 RAM、只读存储器 ROM、定时器/计数器和多种输入输出 I/O(如并行 I/O，串行 I/O)等部分。

设计 8051 的前提是熟悉 8051 的结构。这方面有许多参考书(当然，这些参考书主要讲述 8051 的应用，而不是设计)。下面介绍一下 8051 的存储器结构和指令集。

7.7.1 8051 的存储结构

8051 中的存储器结构，与通常的微处理器不同。它具有分离的程序存储器和数据存储器。

1) 程序存储器

8051 中，程序存储器使用 16 位地址总线，由 16 位程序计数器(PC)进行寻址。寻址空间是 64 KB。

程序存储器有内部和外部之分，8051 内部有 4 KB 的 ROM/EPROM 程序存储器，地址为 0000H～1fffH。8052 内部有 8 KB 的内部程序存储器，地址为 0000～1fffH。正常情况下，程序从内部 ROM 开始运行。当 PC 值超过内部程序存储器范围时，自动转到外部扩展的存储器。在 8031/8032/80c31/80c32 中，片内无 ROM，可扩展 64 KB 外部程序存储器。

程序存储器可采用**立即寻址和基址加变址寻址方式**。

64 KB 的程序存储器中有几个特殊的地址。一个是 0000H，这是系统程序的启动地址。此外，其它特殊地址分别对应于中断源的中断服务子程序入口地址。

2) 数据存储器

数据存储器也分片内和片外两种。两者的地址空间是彼此独立的。片内数据存储器的地址范围为 00H～FFH，片外数据存储器的地址空间为 0000H～FFFFH。访问片内 RAM 用“MOV”指令，访问片外 RAM 用“MOVX”指令。

在 8051 的内部数据存储器中，有 128 B 的 RAM 块(地址为 00H～7FH)和 128 B 的特殊功能寄存器块(地址为 80H～FFH)。这两块地址空间是相连的。

8051 的外部数据存储器寻址空间是 64 KB。对外部数据存储器的访问采用寄存器间接寻址方式。间接寻址寄存器有 R0、R1 和数据指针 DPTR。

3) 特殊功能寄存器

在 8051 中，I/O 口锁存器、定时器/计数器、串口数据缓冲器以及各种控制寄存器(PC 除外)都以特殊功能寄存器的形式出现。特殊功能寄存器能实时地反映整个微控制器内部的工作状态及工作方式。

7.7.2 8051 的指令集

8051 共有 111 条指令。这些指令按功能可分为 5 类：数据传送类、算术运算类、逻辑运算类、控制转移类、位操作类。各类的描述如表 7.19～7.23 所示。

表 7.19 数据传送类指令

指令	描述	字节	执行周期
MOV A, Rn	寄存器内容送入累加器	1	12
MOV A, direct	直接地址单元数据送入累加器	2	12
MOV A, @Ri	间接 RAM 单元中数据送入累加器	1	12
MOV A, #data	立即数送入累加器	2	12
MOV Rn, A	累加器内容送入寄存器	1	12
MOV Rn, direct	直接地址单元数据送入寄存器	2	24
MOV Rn, #data	立即数送入寄存器	2	12
MOV direct, A	累加器内容送入直接地址单元	2	12
MOV direct, Rn	寄存器内容送入直接地址单元	2	24
MOV direct, direct	直接地址单元数据送入另一个直接地址单元	3	24
MOV direct, @Ri	间接 RAM 单元中数据送入直接地址单元	2	24
MOV direct, #data	立即数送入直接地址单元	3	24
MOV @Ri, A	累加器内容送入间接 RAM 单元	1	12
MOV @Ri, direct	直接地址单元送入间接 RAM 单元	2	24
MOV @Ri, #data	立即数送入间接寻址 RAM	2	12
MOV DPTR, #data16	将 16 位的立即数载入地址寄存器	3	24
MOV A, @A+DPTR	以 DPTR 为基地址，变址寻址单元中的数据送入累加器	1	24
MOV A, @A+PC	以 PC 为基地址，变址寻址单元中的数据送入累加器	1	24
MOV A, @Ri	外部 RAM(8 位地址)送入累加器	1	24
MOV A, @DPTR	外部 RAM(16 位地址)送入累加器	1	24
MOV @Ri, A	累加器内容送入外部 RAM(8 位地址)	1	24
MOV @DPTR, A	累加器内容送入外部 RAM(16 位地址)	1	24
PUSH direct	直接地址单元中的数据入栈	2	24
POP direct	直接地址单元中的数据出栈	2	24
XCH A, Rn	寄存器与累加器交换内容	1	12
XCH A, direct	直接地址单元与累加器交换内容	2	12
XCH A, @Ri	间接 RAM 单元与累加器交换内容	1	12
XCHD A, @Ri	间接 RAM 的低半字节与累加器交换内容	1	12

表 7.20　算术运算类指令

指令	描述	字节	执行周期
ADD　A, Rn	寄存器内容加到累加器	1	12
ADD　A, direct	直接地址单元的数据加到累加器	2	12
ADD　A, @Ri	间接 RAM 内容加到累加器	1	12
ADD　A, #data	立即数加到累加器	2	12
ADDC　A, Rn	寄存器内容加到累加器，带进位	1	12
ADDC　A, direct	直接地址单元数据加到累加器，带进位	2	12
ADDC　A, @Ri	间接 RAM 内容加到累加器，带进位	1	12
ADDC　A, #data	立即数加到累加器，带进位	2	12
SUBB　A, Rn	从累加器减去寄存器内容，带借位	1	12
SUBB　A, direct	从累加器减去直接地址单元的内容，带借位	2	12
SUBB　A, @Ri	从累加器减去间接 RAM 中的内容，带借位	1	12
SUBB　A, #data	从累加器减去立即数，带借位	1	12
INC　A	累加器加 1	1	12
INC　Rn	寄存器加 1	1	12
INC　direct	直接地址单元加 1	2	12
INC　@Ri	间接 RAM 单元加 1	1	12
DEC　A	累加器减 1	1	12
DEC　Rn	寄存器减 1	1	12
DEC　direct	直接地址单元减 1	2	12
DEC　@Ri	间接 RAM 单元减 1	1	12
INC　DPTR	地址寄存器加 1	1	24
MUL　AB	A 和 B 相乘	1	48
DIV　AB	A 被 B 除	1	48
DA　A	累加器十进制调整	1	12

表 7.21　逻辑运算类指令

指令	描述	字节	执行周期
ANL　A, Rn	寄存器与累加器相“与”	1	12
ANL　A, direct	直接地址单元与累加器相“与”	2	12
ANL　A,@Ri	间接 RAM 单元与累加器相“与”	1	12
ANL　A, #data	立即数与累加器相“与”	2	12
ANL　direct, A	累加器与直接地址单元相“与”	2	12
ANL　direct, #data	立即数与直接地址单元相“与”	3	24
ORL　A, Rn	寄存器与累加器相“或”	1	12
ORL　A, direct	直接地址单元与累加器相“或”	2	12

续表

指令	描　述	字节	执行周期
ORL　A, @Ri	间接 RAM 单元与累加器相“或”	1	12
ORL　A, #data	立即数与累加器相“或”	2	12
ORL　direct, A	累加器与直接地址单元相“或”	2	12
ORL　direct, #data	立即数与直接地址单元相“或”	3	24
XRL　A, Rn	寄存器与累加器相“异或”	1	12
XRL　A, direct	直接地址单元与累加器相“异或”	2	12
XRL　A, @Ri	间接 RAM 单元与累加器相“异或”	1	12
XRL　A, #data	立即数与累加器相“异或”	2	12
XRL　direct, A	累加器与直接地址单元相“异或”	2	12
XRL　direct, #data	立即数与直接地址单元相“异或”	3	24
CLR　A	累加器清 0	1	12
CPL　A	累加器取反	1	12
RL　A	累加器循环左移	1	12
RLC　A	累加器循环左移，带进位	1	12
RR　A	累加器循环右移	1	12
RRC　A	累加器循环右移，带进位	1	12
SWAP　A	累加器半字节交换	1	12

表 7.22　控制转移类指令

指令	描述	字节	执行周期
ACALL　addr11	绝对(短)调用子程序	2	24
LCALL　addr16	长调用子程序	3	24
RET	从子程序返回	1	24
RETI	从中断返回	1	24
AJMP　addr11	绝对(短)跳转	2	24
LJMP　addr16	长跳转	3	24
SJMP　rel	短跳转(相对地址)	2	24
JMP　@A+DPTR	相对 DPTR 的间接跳转	1	24
JZ　rel	累加器为 0，则跳转	2	24
JNZ　rel	累加器非 0，则跳转	2	24
CJNE　A, direct, rel	比较直接地址单元和累加器，不相同则跳转	3	24
CJNE　A, #data, rel	比较立即数和累加器，不相同则跳转	3	24
CJNE　Rn, #data, rel	比较立即数和寄存器，不相同则跳转	3	24
CJNE @Ri, #data, rel	比较立即数和间接寄存器，不相同则跳转	3	24
DJNZ　Rn, rel	寄存器减 1，非 0 则跳转	3	24
DJNZ　direct, rel	直接地址单元减 1，非 0 则跳转	3	24
NOP	空操作	1	12

表 7.23　位操作指令

指令	描述	字节	执行周期
CLR C	进位清 0	1	12
CLR　bit	直接地址位清 0	2	12
SETB　C	进位位设为 1	1	12
SETB　bit	直接地址位设为 1	2	12
CPL　C	进位位取反	1	12
CPL　bit	直接地址位取反	2	12
ANL　C, bit	进位位与直接地址位相“与”	2	24
ANL　C, /bit	进位位与直接地址位的反相“与”	2	24
ORL　C, bit	进位位与直接地址位相“或”	2	24
ORL　C, /bit	进位位与直接地址位的反相“或”	2	24
MOV　C, bit	将直接地址位送入进位位	2	12
MOV　bit, C	将进位位送入直接地址位	2	24
JC　rel	进位位为 1，则跳转	2	24
JNC　rel	进位位为 0，则跳转	2	24
JB　bit, rel	直接地址位为 1，则跳转	3	24
JNB　bit, rel	直接地址位为 0，则跳转	3	24
JBC　bit, rel	直接地址位为 1，则跳转，并将该位清 0	3	24

7.8　8051 的设计

7.8.1　设计要求

下面给出 8051 设计的一些指标：

- 8 位 CPU；
- 128 B 的片内数据 RAM；
- 4 KB 的片内程序 ROM；
- 64 KB 的程序 RAM；
- 64 KB 的数据 RAM；
- 两个 16 位可编程定时器/计数器；
- 可嵌套的中断处理结构；
- 全双工的 UART；
- 四个 8 位双向 I/O 端口，每一位均可单独寻址；
- 支持 IDLE、POWER-DOWN 模式。

7.8.2 架构规划

系统的设计大致可以划分为外部 ROM、外部 RAM、时钟产生电路及 CPU 核这几个部分，如图 7.23 所示。

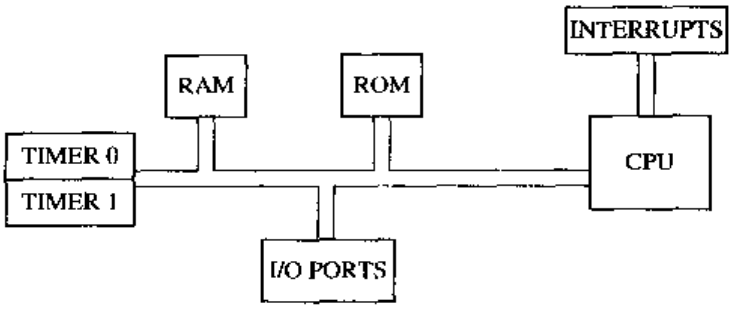


图 7.23 系统的整体划分

图 7.24 给出了 MCS-51 的内部结构。其中 TH2 与 TL2 仅 8052/8032 中才有。

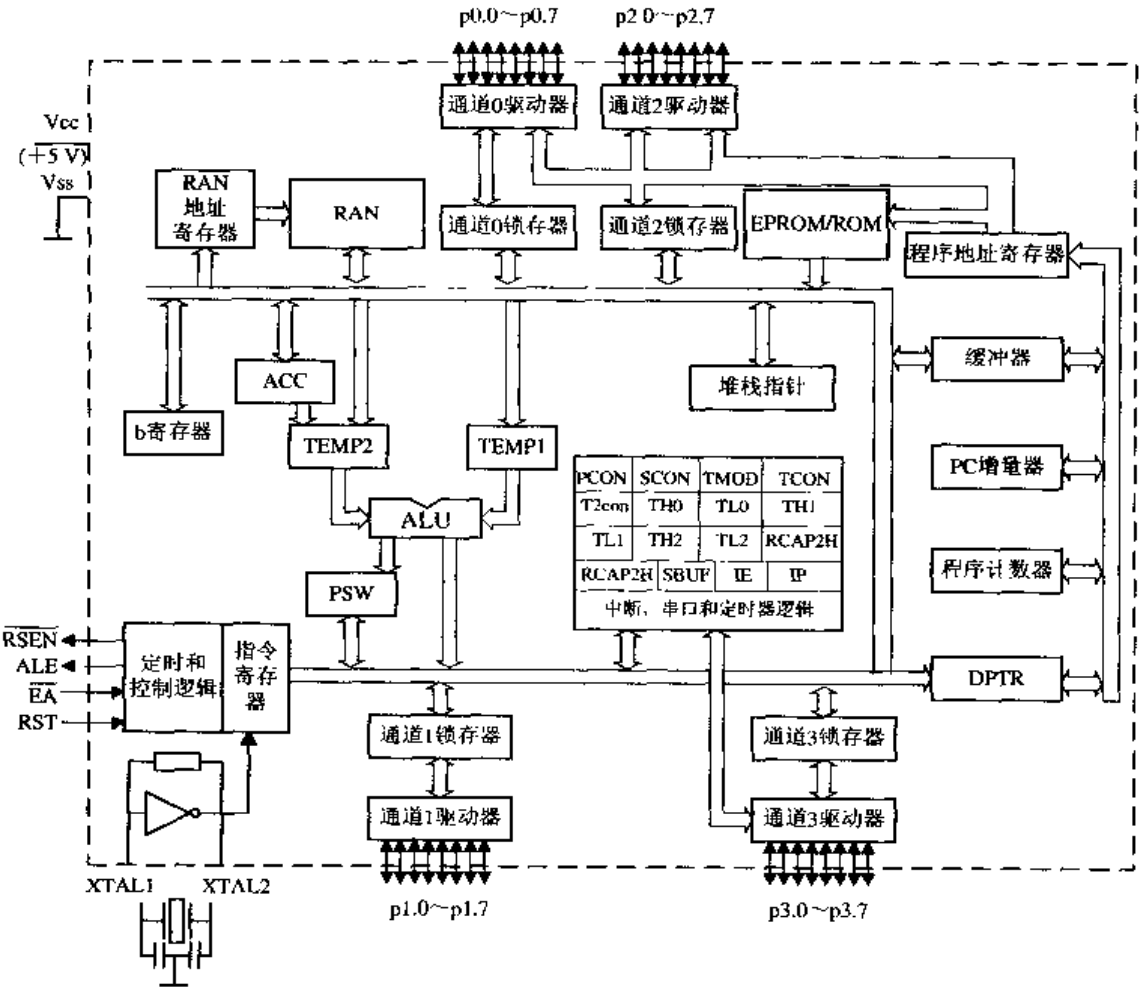


图 7.24 MCS 51 的架构

根据 8051 的架构，可以将 CPU 分成如下模块：

- 时钟分频模块：进行时钟分频，生成系统所用的时钟。

- 存储器控制模块：实现 RAM 读写使能、选通、RAM 地址信号的译码。
- 控制模块：包括时序控制模块(生成指令执行时的状态信号)，指令译码模块(完成指令的译码)。
- ALU 模块：完成各种逻辑和算术运算，并实现了 ACC、B、TMP1、TMP2、PSW 这些寄存器。
- Timer 模块：实现定时器/计数器。
- 中断处理模块：进行中断仲裁。

在 8051 中，程序存储器是这样划分的：

地址 0000～0002 用于初始化程序；

地址 0003～0042 是中断向量区，放置跳向中断服务程序的长跳转指令。

内部数据存储器可分为 128 B 的内部数据 RAM 和 128 B 的特殊寄存器空间。在内部数据存储器中有四个寄存器阵列(每个寄存器阵列由 8 个寄存器组成)，128 个可寻址的位以及堆栈。堆栈的最大深度由可利用的内部数据存储器的大小决定。堆栈的起始地址由堆栈指针寄存器指定。8051 中所有的寄存器(除了程序计数器 PC 和四个寄存器阵列)都位于特殊功能寄存器 SFR 的地址空间中。这些存储器映射的寄存器包括 ALU 单元、I/O 单元、中断系统、定时器/计数器单元和串行输入/输出单元中的所有寄存器。

内部数据 RAM 的地址空间为 0～255，其中四个寄存器阵列的地址空间为 0～31。另外，内部数据 RAM 有 128 位可以通过直接寻址方式来访问。这些位的字节地址为 32～47。特殊功能寄存器的地址空间为 128～255。除了程序计数器 PC 和四个寄存器阵列，其它所有的寄存器都处于这个地址范围内。存储器映射的特殊功能寄存器使得它们可以像访问内部 RAM 一样方便地被访问。

这样，特殊功能寄存器可以由大多数指令来操作，而无需增加专门的指令。

关于 8051 中存储器的结构，请参看有关 8051 的参考书。

7.8.3 时序规划

80851 中，典型的指令周期是一个机器周期。一个机器周期包括 6 个状态 S1～S6，每个状态又分为两部分：相位 1(P1)和相位 2(P2)，也就是说，1 个机器周期=6 个状态=12 个振荡周期，如图 7.25 所示。

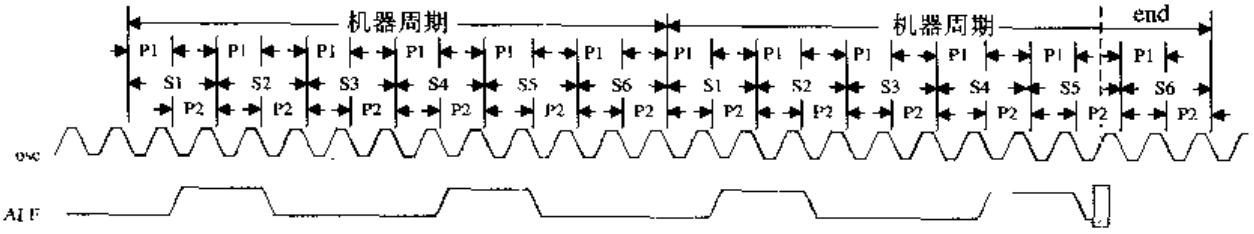


图 7.25 机器周期与 ALE 信号

在 8051 中，指令长度从 1B 到 3B 不等，指令周期从 1 到 4 个机器周期不等。因此，指令的时序较为复杂。图 7.26 给出了典型的单字节周期指令的时序。图 7.27 给出了典型的双字节单周期指令的时序。图 7.28 给出了典型的单字节双周期指令的时序。

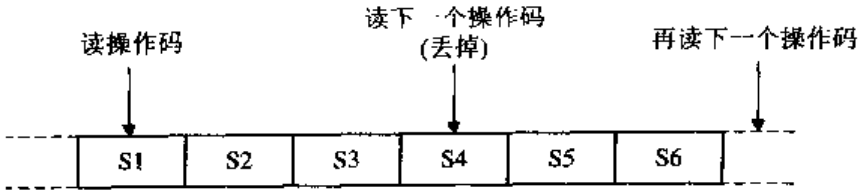


图 7.26　单字节周期指令的时序

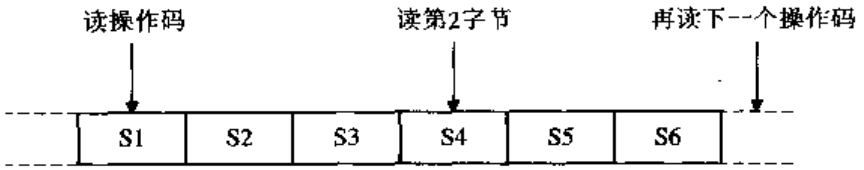


图 7.27　双字节单周期指令的时序

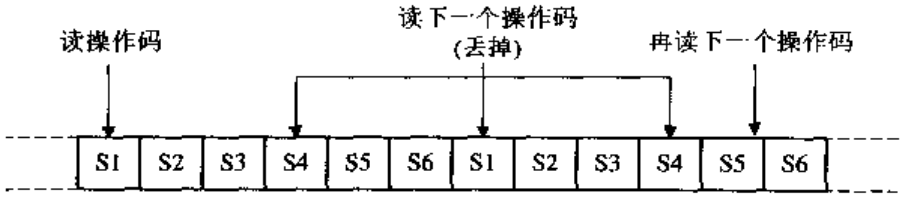


图 7.28　单字节双周期指令的时序

7.8.4　低功耗方式和时钟生成模块

8051 支持低功耗模式。

实现低功耗的原理是：在空闲状态，关闭某些模块的时钟，使这些模块不工作，不产生功耗。实现低功耗的方法有两种，一种是将所有时钟都关掉的掉电工作方式，一种是保留中断逻辑/串口/定时器的时钟，而关闭其它模块时钟的待机方式。

图 7.29 给出了待机和掉电硬件的结构。

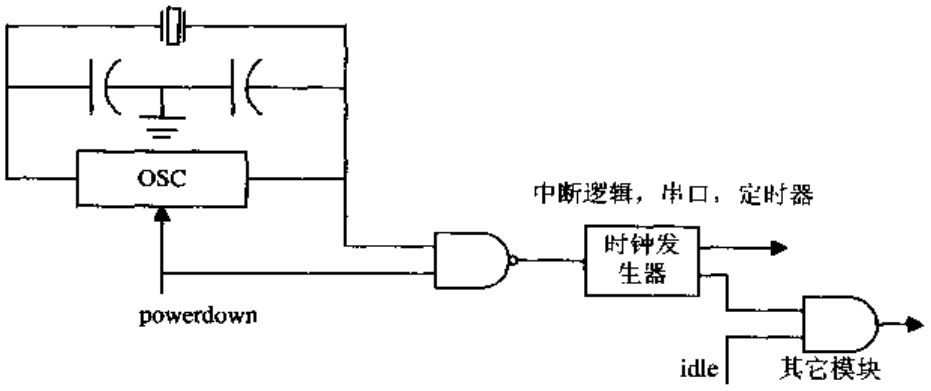


图 7.29　待机和掉电硬件结构

在 8051 中，有两个时钟，即 cka 与 ckb，两个时钟是双相非交迭时钟。在低功耗状态下 ckb 被关掉，而 cka 不能被关掉。

时钟生成模块的实现代码如下所示：

```
module clkgen (osc, power_t, idle_t, cka1, cka2, cka2n, ckb1, ckb2, ckb2n);
//端口描述
    input osc;
    input power_t;
    input idle_t;
    output cka1;
    output cka2;
    output cka2n;
    output ckb1;
    output ckb2;
    output ckb2n;
//信号声明
    wire cka1;
    wire cka2;
    wire cka2n;
    wire ckb1;
    wire ckb2;
    wire ckb2n;

    reg osc_2;
    reg cka1_tmp;
    reg cka2_tmp;

    initial
    begin
        osc_2 <= 1'b0;
    end

    always @(osc)
    begin : divide_unit
        if (osc)
        begin
            osc_2 <= ~osc_2 ;
        end
    end

    always @(osc_2)
    begin : cka1_unit
```

```
        if (osc_2)
        begin
            cka2_tmp <= ~osc_2 ;
            cka1_tmp <= #10 osc_2 ;
        end
        else
        begin
            cka1_tmp <= osc_2 ;
            cka2_tmp <= #10 ~osc_2 ;
        end
    end
    assign cka1 = (!idle_t) ? cka1_tmp : 1'b0 ;
    assign cka2 = (!idle_t) ? cka2_tmp : 1'b1 ;
    assign cka2n = (!idle_t) ? ~cka2_tmp : 1'b1 ;
    assign ckb1 = cka1_tmp ;
    assign ckb2 = cka2_tmp ;
    assign ckb2n = ~cka2_tmp ;
endmodule
```

时钟生成电路的综合结果如图 7.30 所示。

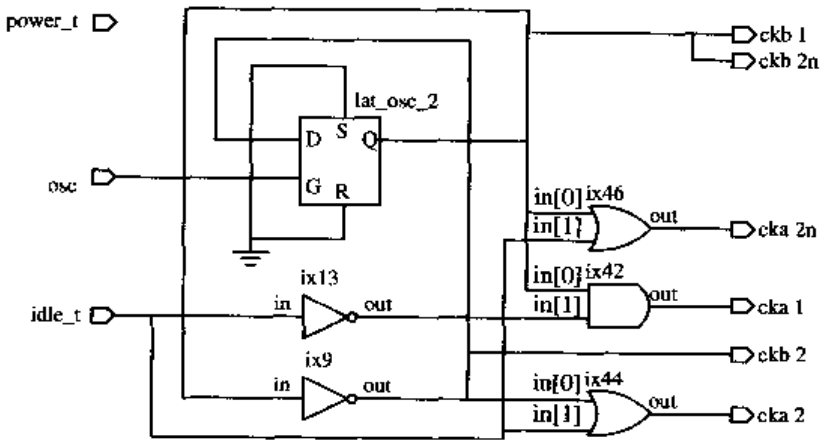


图 7.30　时钟生成电路的综合结果

7.8.5　控制模块的设计

控制器的实现有两种方式，一种是微程序方式，一种是硬布线方式。

1）微程序控制器

简单地说，微程序控制器是把指令执行所需要的所有控制信号存放在控制存储器中，需要时再从这个存储器中读取。

它的基本思想是：模仿一般的程序设计方法，把操作控制信号编写成所谓的“微指令”，并把它们存放在一个只读存储器内。当控制器工作时，这些微指令被一条条地逐次读出，

从而产生了各种操作控制信号，使被控制的部件执行所需要的操作。

CPU 的指令功能是由许多微指令组成的序列来完成的。这个微指令序列通常称为微程序。在执行一条微指令时必须给出下一条微指令的地址，以便当前的微指令执行完毕后，取出下一条指令。

微程序控制器的原理图如图 7.31 所示，它主要由控制存储器、微指令寄存器和地址转移逻辑三部分组成。其中，微指令寄存器分为微地址寄存器和微命令寄存器两个部分。

微程序控制的特点是灵活性好，速度慢。

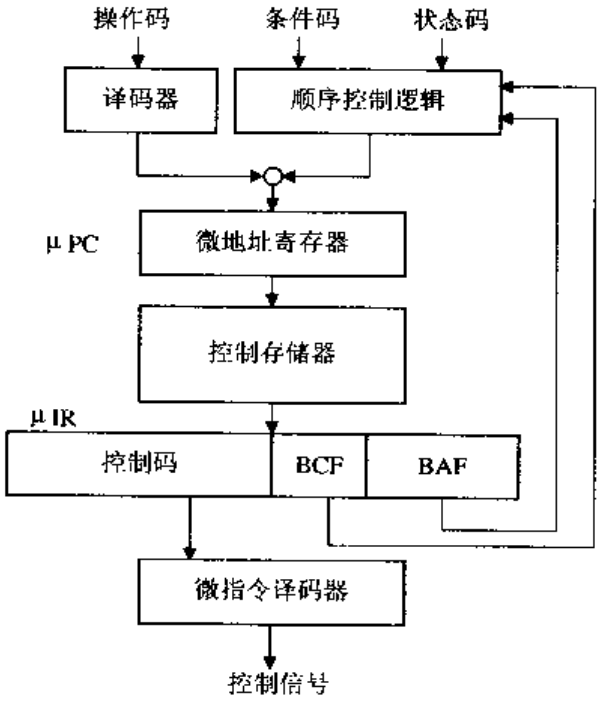


图 7.31　微程序控制器

2) 硬布线逻辑控制器

硬布线逻辑控制器的组成如图 7.32 所示。

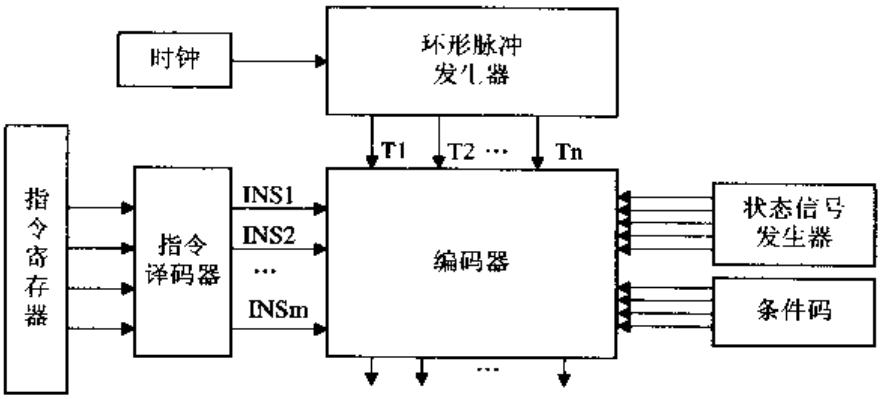


图 7.32　硬布线逻辑控制器的组成

说明：条件码是指算术运算的异常、内部中断等。

编码器电路的一般逻辑表达形式是:

C = T1*(INS1 + INS2 + …) + T2*(INS1 + INS2 + …) + …

控制器的主要功能是对指令译码，并产生相应的控制信号。控制器针对不同的指令和时序，译码出相应的微操作指令，用组合逻辑产生针对各特殊功能寄存器、输入输出端和存储器的读写控制信号，各种算术逻辑微操作控制信号以及各种操作状态的标志信号。这是整个 8051 设计的难点所在。一般而言，对于 RISC，控制时序的产生较容易，常采用硬布线控制器；对于 CISC，控制时序的产生较复杂，常采用微程序控制器。8051 虽然也属于 CISC，但由于它的控制相对简单，因此也可以用硬布线控制器的方式。

8051 的机器周期由包含 12 个时钟振荡周期的 6 个状态(S1，S2，…，S6)组成。每个状态可分成两部分：相位 1(P1)和相位 2(P2)。相位 1 在每个状态的前半周期有效。相位 2 在每个状态的后半周期有效。这样一个机器周期包括 12 个时钟状态。

对于单周期指令，指令周期开始于 S1P2。此时把指令码读入指令寄存器，并开始执行。对于双字节单周期指令，在同一机器周期的 S4 状态读入第二个字节。如果是单字节单周期指令，在 S4 状态仍进行读操作，但不理会读的内容，并且 PC 不加 1。所有的指令都在 S6P2 结束。

在控制模块中，要生成状态信号，用来表示状态 S1～S6。

由于不同的指令占用的机器周期不同。有单周期、双周期和四周期之分。针对不同的指令，要产生相应的周期信号，以进行区别。

要设计一个 8051 微控制器，必须了解每个指令执行时的细节。要清楚何时译码，何时取操作数，何时将指令的执行结果写回到存储器单元。例如，

```
casex (op)
    `ACALL :begin
        ram_rd_sel = 2'b00 ;
        ram_wr_sel = 3'b011;
        operand_sel1 = 2'b11;
        operand_sel2 = 2'b00;
        alu_op = 4'b0000;
        immediate_sel = 3'b010;
        wr = 1'b1;
        psw_set = 2'b00;
        c_sel = 2'b00;
        pc_wr = 1'b0;
        pc_sel = 2'b00;
        operand_sel3 = 1'b0;
        comp_sel = 2'b00;
        rmw = 1'b0;
        bit_addr = 1'b0;
    rom_addr_sel = 1'b0;
```

```
        end
    `AJMP : begin
    ...
```

7.8.6 数据通路部分的设计

在数据通路中，累加器 A 是使用最频繁的寄存器。许多指令都要通过累加器 A 来进行。累加器 A 是许多指令缺省的源操作数或目的操作数，这对 A 的读写控制与具体的指令有关，需要由指令译码电路来产生专门的控制信号。

寄存器 B 一般用于乘、除法指令。在进行乘、除法时，寄存器 B 中存放第二个操作数、乘积的高位字节或除法的余数部分。

数据通路中，算术逻辑单元主要完成如下功能：

- 加、减、乘、除；
- 递增(加 1)、递减(减 1)运算；
- 移位；
- 按字节的逻辑操作(与，或，异或，取反)；
- 按位的逻辑操作(与，或，异或，取反)；
- 按字节的置位、清零操作；
- 按位的置位、清零操作；
- 数据传送操作。

算术逻辑单元由三个部分构成：加减法部分、按字节的逻辑操作电路、位处理电路。

加减法电路完成 ADD、ADDC、INC、DA、SUBB、DEC 等指令。按字节的逻辑操作电路完成 ANL、ORL、XRL、CLR、CPL、RL、RLC、RR、RRC、SWAP、XCHD 等指令的操作。位处理电路完成 SETB、CLR、CPL、ANL、ORL、MOV、JB、JNB、JBC 等位处理指令。

下面给出 ALU 模块的部分实现代码。

```
////////////////////////////////////////////////////////////
//// 乘法器与除法器的实例化////
multiply
mul1(.clk1(clk1), .clk2(clk2),.rst(rst), .enable(enable_mul), .operand_1(operand_1), .operand_2(opera
nd_2), .Output1(multiplicand), .Output2(mulitiplier), .Overflow_out(mulOv));
divide
div1(.clk1(clk1), .clk2(clk2),.rst(rst), .enable(enable_div), .operand_1(operand_1), .operand_2(operand
_2), .Output1(dividend), .Output2(divisor), .Overflow_out(divOv));
```

下面给出 ALU 实现的部分代码：

```
    case (op_code)
//加法
        ...
//减法
```

```
    ...
//乘法
    `ALU_MUL: begin
      Output1 = multiplicand;
      Output2 = mulitiplier;
      Overflow_out = mulOv;
      C_Out = 1'b0;
      AC_Out = 1'bx;
      enable_mul = 1'b1;
      enable_div = 1'b0;
    end
//除法
    `ALU_DIV: begin
      Output1 = dividend;
      Output2 = divisor;
      Overflow_out = divOv;
      AC_Out = 1'bx;
      C_Out = 1'b0;
      enable_mul = 1'b0;
      enable_div = 1'b1;
    end
//取反操作
    `ALU_NOT: begin
      Output1 = ~operand_1;
      Output2 = 8'h00;
      C_Out = !C_in;
      AC_Out = 1'bx;
      Overflow_out = 1'bx;
      enable_mul = 1'b0;
      enable_div = 1'b0;
    end
//与操作
    `ALU_AND: begin
      Output1 = operand_1 & operand_2;
      Output2 = 8'h00;
      C_Out = C_in & bit_in;
      AC_Out = 1'bx;
      Overflow_out = 1'bx;
      enable_mul = 1'b0;
```

```
            enable_div = 1'b0;
        end
    //异或操作
        `ALU_XOR: begin
            Output1 = operand_1 ^ operand_2;
            Output2 = 8'h00;
            C_Out = C_in ^ bit_in;
            AC_Out = 1'bx;
            Overflow_out = 1'bx;
            enable_mul = 1'b0;
            enable_div = 1'b0;
        end
    //或操作
        `ALU_OR: begin
            Output1 = operand_1 | operand_2;
            Output2 = 8'h00;
            C_Out = C_in | bit_in;
            AC_Out = 1'bx;
            Overflow_out = 1'bx;
            enable_mul = 1'b0;
            enable_div = 1'b0;
        end
    //循环左移
        `ALU_RL: begin
            Output1 = {operand_1[6:0], operand_1[7]};
            Output2 = 8'h00;
            C_Out = C_in | !bit_in;
            AC_Out = 1'bx;
            Overflow_out = 1'bx;
            enable_mul = 1'b0;
            enable_div = 1'b0;
        end
    //循环左移，进位跟着移
        `ALU_RLC: begin
            Output1 = {operand_1[6:0], C_in};
            Output2 = {operand_1[3:0], operand_1[7:4]};
            C_Out = operand_1[7];
            AC_Out = 1'b0;
            Overflow_out = 1'b0;
```

```
            enable_mul = 1'b0;
            enable_div = 1'b0;
        end
    //右移
        `ALU_RR: begin
            Output1 = {operand_1[0], operand_1[7:1]};
            Output2 = 8'h00;
            C_Out = C_in & !bit_in;
            AC_Out = 1'b0;
            Overflow_out = 1'b0;
            enable_mul = 1'b0;
            enable_div = 1'b0;
        end
    //右移，进位跟着移动
        `ALU_RRC: begin
            Output1 = {C_in, operand_1[7:1]};
            Output2 = 8'h00;
            C_Out = operand_1[0];
            AC_Out = 1'b0;
            Overflow_out = 1'b0;
            enable_mul = 1'b0;
            enable_div = 1'b0;
        end
    //其它操作
      ...
        default: begin
            Output1 = operand_1;
            Output2 = operand_2;
            C_Out = C_in;
            AC_Out = AC_in;
            Overflow_out = 1'bx;
            enable_mul = 1'b0;
            enable_div = 1'b0;
        end
      endcase
    end

    endmodule
```

7.8.7 定时器/计数器的设计

定时器/计数器是 8051 中重要的外围设备，可用于定时控制，或者用于对外部信号的计数。8051 提供了两个 16 位可编程定时器/计数器。这里所说的可编程，是指编程者可以控制如下内容：

- 可以确定其工作方式是定时还是计数；
- 可以预置定时或计数初值；
- 当达到定时时间或计数终止时，确定要不要产生中断请求；
- 如何启动定时或计数器工作。

图 7.33 给出了定时器/计数器的逻辑结构。由图 7.33 可见，两个 16 位定时器/计数器 T0 和 T1，分别由 8 位计数器 TH0、TL0、TH1 和 TH0 构成。特殊功能寄存器 TMOD 用于控制定时器/计数器的工作方式，TCON 用于控制定时器/计数器的启动运行，并记录 T0、T1 的溢出标志。通过对 TH0、TL0 和 TH1、TL1 的初始化编程，可以预置 T0、T1 的计数初值。通过对 TMOD 和 TCON 的初始化编程，可以指定工作方式并控制 T0、T1 按规定的工作方式进行计数。

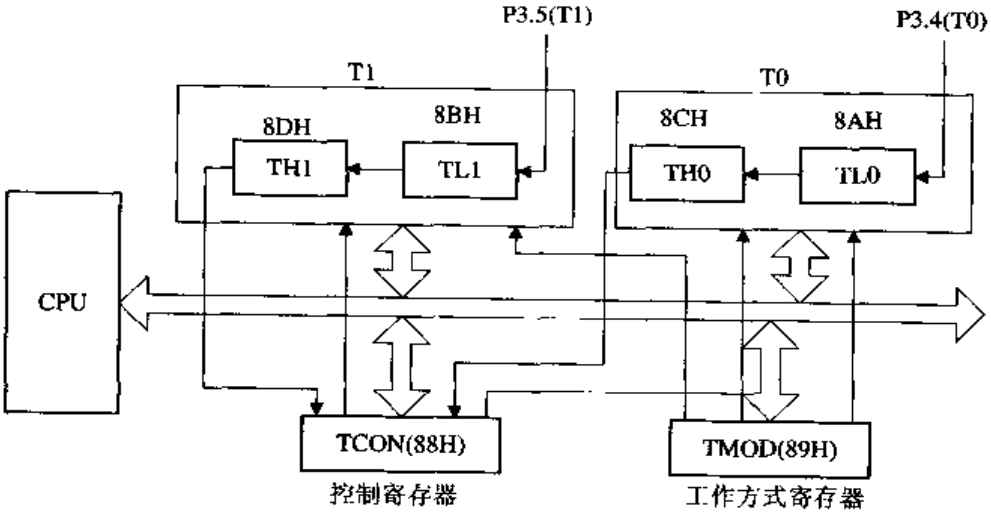


图 7.33 定时器/计数器逻辑结构

当选择定时器工作方式时，计数输入信号来自内部振荡信号。在每个机器周期内，计数器作“加 1”运算。

当选择计数器工作方式时，计数输入信号来自外部引脚 T0(P3.4)、T1(P3.5)上的计数脉冲。外部每输入一个脉冲，计数器 TH0、TL0 作一次加 1。在对脉冲进行计数时，每个机器周期采样一次。若前一次采样到低电平，后一次采到高电平，则计数器加 1。

定时器/计数器的用法是可以控制的。在开始定时或开始计数之前，必须要对特殊功能寄存器 TMOD 和 TCON 写入一个方式字或控制字，即要有一个初始化过程。

工作方式寄存器 TMOD 的格式如图 7.34 所示。

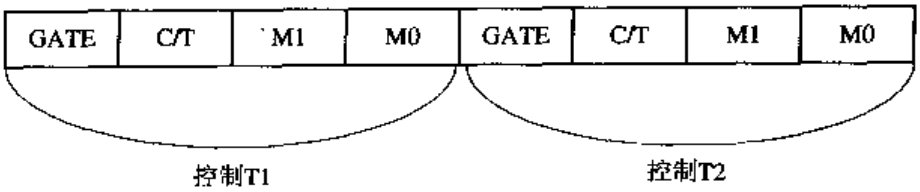


图 7.34　TMOD 寄存器的结构

其中，GATE 是门控位。计数器受外部中断信号的控制(int0 控制 T0 计数，int1 控制 T1 计数)。当运行控制位 TR0(或 TR1)为“1”时，开始计数；为“0”时，停止计数。当 GATE 为“0”时，外部中断信号不参与控制。此时，只要运行控制位 TR0(或 TR1)为“1”，计数器就开始工作。

C/T 决定采用的是计数器方式还是定时器方式。“0”为定时器方式；“1”为计数器方式。

M0 与 M1 是操作方式选择位，功能如表 7.24 所示。

表 7.24　计数器操作方式

M1M0	操作方式	功能
00	方式 0	13 位计数器
01	方式 1	16 位计数器
10	方式 2	可自动再装载的 8 位计数器
11	方式 3	T0 分为两个独立计数器，T1 停止计数

控制寄存器 TCON 的结构如图 7.35 所示。

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

图 7.35　TCON 的结构

其中，TF1 是 T1 溢出标志。当 T1 产生溢出时，由硬件置 1，可向 CPU 发中断请求。CPU 响应中断后，该标志位被硬件自动清 0，也可以由程序查询后清 0。

TR1 是 T1 运行控制位。由软件置 1 启动 T1 工作，置 0 关闭 T1 工作。

TF0：T0 溢出标志。

TR0：T0 运行控制位。

IE1：外部中断 INT1 请求标志。

IT1：外部中断 INT1 触发方式选择位。

IE0：外部中断 INT0 触发方式选择位。

IT0：外部中断 INT0 触发方式选择位。

下面给出 TMOD 寄存器的实现代码：

```
reg [7:0] tomd;
always @(posedge irst or negedge ckb2)
  begin
    if(irst)
      tmod <= 8'b0;
```

```
    else
      if(tmodwr_en)
        tmod <= dbusin;
    if(
end
assign gate1 = tmod[7];
assign c_tbar1 = tmod[6];
assign mode1 = {tmod[5],tmod[4]};
assign gate0 = tmod[3];
assign c_tbar0 = tmod[2];
assign mode0 = {tmod[1],tmod[0]};
```

7.8.8 串口的设计

8051 的串口 I/O 是一个全双工的异步串行通信接口，由发送缓冲和接收缓冲构成。与串口有关的特殊功能寄存器是 SCON 和 PCON。

1) 串口控制寄存器 SCON

串口包括一个控制寄存器 SCON，其格式如图 7.36 所示。

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

图 7.36 SCON 寄存器的结构

下面对寄存器的各标志位进行说明。

SM0、SM1：串口方式选择位。串口工作方式如表 7.25 所示。

表 7.25 串口工作方式

SM0	SM1	方式	功能说明
0	0	0	移位寄存器方式(用于 I/O 扩展)
0	1	1	8 位 UART，波特率可变
1	0	2	9 位 UART，波特率为 fosc/64 或 fosc/32
1	1	3	9 位 UART，波特率可变

SM2：允许方式 2 和方式 3 多机通信控制位。

REN：允许接收控制位。当软件置 1 时，允许接收；当软件置 0 时，禁止接收。

TB8：在方式 2 和方式 3 中要发送的第 9 位数据。需要时由软件置位或复位。

RB8：在方式 2 和方式 3 中要接收到的第 9 位数据。在方式 1 时，如 SM2=0，RB8 是接收到的停止位。在方式 0 中，不使用 RB8。

TI：发送中断标志。

RI：接收中断标志。

SIO 共有四种工作方式，可以通过对特殊功能寄存器 SCON 的编程进行选择。其中，

工作方式 0：作为移位寄存器来使用，可以用来扩展 I/O 接口。在输出时，SIO 将数据

发送缓冲器中的内容串行地移出到外部的移位寄存器。在输入时，SIO 将外部移位寄存器的内容移入内部的输入移位寄存器，然后再写入数据接收缓冲器。

工作方式 1：一个 8 位异步通信口。

工作方式 2 或 3：一个 9 位异步串行通信口。

SCON 的实现代码如下所示：

```
reg [7:0] scon;
//串行端口控制寄存器的实现
always @(posedge clk or posedge rst)
begin
  if (rst)
    scon <= #1 8'b00000000;
  else if ((wr) & !(wr_bit) & (wr_addr==8'h98))
    scon <= #1 data_in;
  else if ((wr) & (wr_bit) & (wr_addr[7:3]==5'b10011))
    scon[wr_addr[2:0]] <= #1 bit_in;
  else if ((trans_buf) & !(trans))
    scon[1] <= #1 1'b1;
  else if ((receive_buf) & !(receive) & !(sbuf_rxd_tmp[0])) begin
    case (scon[7:6])
      2'b00: scon[0] <= #1 1'b1;
      default: begin
        if ((sbuf_rxd_tmp[9]) | !(scon[5])) scon[0] <= #1 1'b1;
scon[2] <= #1 sbuf_rxd_tmp[9];
        end
    endcase
  end
end
```

串行口内部有数据接收缓冲器 SBUFI 和数据发送缓冲器 SBUFO。这两个缓冲器公用一个地址 99H，发送缓冲器只写不读，接收缓冲器只读不写。

2) 发送缓冲

发送缓冲实现代码如下所示：

```
always @(posedge clk or posedge rst)
begin
  if (rst) begin
    sbuf_txd <= #1 8'b0000_0000;
    tr_start <= #1 1'b0;
  end
else if ((wr_addr==8'h99) & (wr) & !(wr_bit)) begin
    sbuf_txd <= #1 data_in;
```

```
        tr_start <= #1 1'b1;
    end
else tr_start <= #1 1'b0;
end
//
// 发送功能的实现
always @(posedge clk or posedge rst)
begin
    if (rst) begin
        txd <= #1 1'b1;
        tr_count <= #1 4'd0;
        trans <= #1 1'b0;
        smod_cnt_t <= #1 6'h0;
//开始发送
    end
else if (tr_start) begin
        case (scon[7:6])
            2'b00: begin    // mode 0
                txd <= #1 sbuf_txd[0];
tr_count <= #1 4'd1;
            end
            2'b10: begin
                txd <= #1 1'b0;
tr_count <= #1 4'd0;
            end
            default: // 模式 1 与模式 3
            begin
tr_count <= #1 4'b1111;
            end
        endcase
        trans <= #1 1'b1;
        smod_cnt_t <= #1 6'h0;
    end
//发送中
else if (trans)
    begin
        case (scon[7:6])
            2'b00: begin //mode 0
                if (smod_cnt_t == 6'd12) begin
```

```
            if (tr_count==4'd8)
            begin
                trans <= #1 1'b0;
                txd <= #1 1'b1;
              end else begin
              txd <= #1 sbuf_txd[tr_count];
                tr_count <= #1 tr_count + 4'b1;
              end
            smod_cnt_t <= #1 6'h0:
            end else smod_cnt_t <= #1 smod_cnt_t + 6'h01;
        end
        2'b01: begin // mode 1
          if ((t1_ow) & !(t1_ow_buf))
            begin
          if (((pcon[7]) &    (smod_cnt_t == 6'd15))| (!(pcon[7]) & (smod_cnt_t==6'd31)))
              begin
              case (tr_count)
                4'd8: txd <= #1 1'b1;   // stop bit
                  4'd9: trans <= #1 1'b0;
                  4'b1111: txd <= #1 1'b0; //start bit
                  default: txd <= #1 sbuf_txd[tr_count];
                endcase
              tr_count <= #1 tr_count + 4'b1;
                smod_cnt_t <= #1 6'h0;
              end else smod_cnt_t <= #1 smod_cnt_t + 6'h01;
            end
        end
        2'b10: begin // 模式 2
// 如果 smod (pcon[7]) 为 1，则计到 4，否则计到 6
          if (((pcon[7]) & (smod_cnt_t==6'd31)) | (!(pcon[7]) & (smod_cnt_t==6'd63))) begin
              case (tr_count)
              4'd8: begin
                  txd <= #1 scon[3];
                end
              4'd9: begin
                  txd <= #1 1'b1; //stop bit
                end
              4'd10: begin
                  trans <= #1 1'b0;
```

```
                end

                default: begin
                  txd <= #1 sbuf_txd[tr_count];
                end
              endcase
            tr_count <= #1 tr_count+1'b1;
              smod_cnt_t <= #1 6'h00;
             end
           else begin
            smod_cnt_t <= #1 smod_cnt_t + 6'h01;
            end
        end
        default: begin // 模式 3
          if ((t1_ow) & !(t1_ow_buf))
      begin
        if (((pcon[7]) &    (smod_cnt_t == 6'd15))| (!(pcon[7]) & (smod_cnt_t==6'd31)))
         begin
           case (tr_count)
                 4'd8: begin
                txd <= #1 scon[3];
             end
                 4'd9: begin
                txd <= #1 1'b1; //stop bit
             end
                 4'd10: begin
            trans <= #1 1'b0;
          end
             4'b1111: txd <= #1 1'b0; //start bit
             default: begin
                txd <= #1 sbuf_txd[tr_count];
             end
           endcase
              tr_count <= #1 tr_count+1'b1;
           smod_cnt_t <= #1 6'h00;
         end else smod_cnt_t <= #1 smod_cnt_t + 6'h01;
       end
        end
      endcase
```

```
    end else
      txd <= #1 1'b1;
  end
```

3) 接收缓冲

接收缓冲实现代码如下所示：

```
  always @(posedge clk or posedge rst)
  begin
    if (rst) begin
      re_count <= #1 4'd0;
      receive <= #1 1'b0;
      sbuf_rxd <= #1 8'h00;
      sbuf_rxd_tmp <= #1 11'd0;
      smod_cnt_r <= #1 6'h00;
      r_int <= #1 1'b0;
    end else if (receive) begin
      case (scon[7:6])
        2'b00: begin // mode 0
          if (smod_cnt_r==6'd12) begin
            if (re_count==4'd8) begin
                receive <= #1 1'b0;
                r_int <= #1 1'b1;
                sbuf_rxd <= #1 sbuf_rxd_tmp[8:1];
             end
              else begin
              sbuf_rxd_tmp[re_count + 4'd1] <= #1 rxd;
                r_int <= #1 1'b0;
               end
            re_count <= #1 re_count + 4'd1;
            smod_cnt_r <= #1 6'h00;
          end
           else smod_cnt_r <= #1 smod_cnt_r + 6'h01;
        end
        2'b01: begin // mode 1
          if ((t1_ow) & !(t1_ow_buf))
          begin
            if (((pcon[7]) &    (smod_cnt_r == 6'd15))| (!(pcon[7]) & (smod_cnt_r==6'd31)))
              begin
              r_int <= #1 1'b0;
               re_count <= #1 re_count + 4'd1;
             smod_cnt_r <= #1 6'h00;
```

```
            sbuf_rxd_tmp[re_count_buff] <= #1 rxd;
            if ((re_count==4'd0) && (rxd))
              receive <= #1 1'b0;

        end else smod_cnt_r <= #1 smod_cnt_r + 6'h01;
    end
         else begin
        r_int <= #1 1'b1;
            if (re_count == 4'd10)
        begin
                sbuf_rxd <= #1 sbuf_rxd_tmp[8:1];
          receive <= #1 1'b0;
                r_int <= #1 1'b1;
        end
        else r_int <= #1 1'b0;
      end
    end
    2'b10: begin // mode 2
      if (((pcon[7]) & (smod_cnt_r==6'd31)) | (!(pcon[7]) & (smod_cnt_r==6'd63))) begin
        r_int <= #1 1'b0;
            re_count <= #1 re_count + 4'd1;
        smod_cnt_r <= #1 6'h00;
        sbuf_rxd_tmp[re_count_buff] <= #1 rxd;
        re_count <= #1 re_count + 4'd1;
         end else begin
        smod_cnt_r <= #1 smod_cnt_r + 6'h1;
            if (re_count==4'd11) begin
              sbuf_rxd <= #1 sbuf_rxd_tmp[8:1];
              r_int <= #1 sbuf_rxd_tmp[0] | !scon[5];
              receive <= #1 1'b0;
            end
                else
              r_int <= #1 1'b0;
      end
    end
    default: begin // mode 3
      if ((t1_ow) & !(t1_ow_buf))
      begin
        if (((pcon[7]) &    (smod_cnt_r == 6'd15))| (!(pcon[7]) & (smod_cnt_r==6'd31)))
            begin
```

```
                sbuf_rxd_tmp[re_count] <= #1 rxd;
                    r_int <= #1 1'b0;
                 re_count <= #1 re_count + 4'd1;
                    smod_cnt_r <= #1 6'h00;
                 end
                else smod_cnt_r <= #1 smod_cnt_r + 6'h01;
              end
             else begin
             if (re_count==4'd11) begin
                sbuf_rxd <= #1 sbuf_rxd_tmp[8:1];
                receive <= #1 1'b0;
                    r_int <= #1 sbuf_rxd_tmp[0] | !scon[5];
                 end else begin
                r_int <= #1 1'b0;
             end
              end
        end
     endcase
end
     else begin
   case (scon[7:6])
      2'b00: begin
         if ((scon[4]) && !(scon[0]) && !(r_int)) begin
            receive <= #1 1'b1;
            smod_cnt_r <= #1 6'h6;
         end
      end
      2'b10: begin
         if ((scon[4]) && !(rxd)) begin
            receive <= #1 1'b1;
            if (pcon[7])
               smod_cnt_r <= #1 6'd15;
            else smod_cnt_r <= #1 6'd31;
         end
      end
      default: begin
         if ((scon[4]) && (!rxd)) begin
            if (pcon[7])
               smod_cnt_r <= #1 6'd7;
            else smod_cnt_r <= #1 6'd15;
```

```
            receive <= #1 1'b1;
          end
        end
      endcase

      sbuf_rxd_tmp <= #1 11'd0;
      re_count <= #1 4'd0;
      r_int <= #1 1'b0;
    end
  end
```

7.8.9 中断控制系统

中断能对异步发生的外界事件作出及时的处理，从而可以大大提高处理效率。

8051 的中断系统有 5 个中断源，具有两个中断优先级，可以实现中断服务程序的两级嵌套。与中断控制系统有关的特殊功能寄存器有中断标志寄存器(由 TCON 和 SCON 中的有关位组成)，中断允许寄存器 IE 和中断优先级寄存器 IP。

当 CPU 响应一个中断请求时，CPU 会发出一个 INTACK 信号，且中断向量在某一时刻会出现在内部数据总线上，于是可以通过对中断向量的译码来识别当前 CPU 响应的是哪一个中断请求，从而清除相应的请求标志。

表 7.26 给出了 8051 中断系统的端口描述。

表 7.26 中断模块端口说明

Ckb1	输入	时钟
Ckb2	输入	时钟
Reset	输入	复位信号
Wr_addr[7:0]	输入	写寄存器的地址信号
Rd_addr[7:0]	输入	读寄存器的地址信号
Data_in	输入	输入数据
Bit_in	输入	位输入
Uart	输入	标志中断来自串口
Wr_rd	输入	读/写信号
Tf0	输入	定时器中断 0
Tf1	输入	定时器中断 1
Ie0	输入	外部中断 0
Ie1	输入	外部中断 1
Ret_interrupt	输入	中断信号的返回
Int_vec	输出	中断向量
Ack	输出	中断应答
Bit_out	输出	位输出

下面给出中断模块的定义，具体实现这里略去。

module oc0851_int (clk, wr_addr, rd_addr, data_in, bit_in, data_out, bit_out, wr, wr_bit, tf0, tf1, intr, ie0, ie1, rst, reti, int_vec, tr0, tr1, uart, ack);

7.9 练　　习

1．某计算机有 10 条指令，它们的使用频率分别为

0.35, 0.20, 0.11, 0.09, 0.08, 0.07, 0.04, 0.03, 0.02, 0.01

试用霍夫曼编码对它们的操作码进行编码，并计算平均代码长度。

2．对于图 7.37 所示的单总线的 CPU 结构，写出执行带有间接访存寻址方式的指令“STORE (mem),R1”的执行过程；然后设计一个可实现下列指令操作的硬连线控制器，画出控制器逻辑图并写出各控制信号的逻辑表达式

ADD R3,R1,R2　　;R1+R2->R3

LOAD mem,R1　　;(mem)->R1

STORE mem,R1　　;R1->(mem)

JMP #A　;PC+1+A->PC

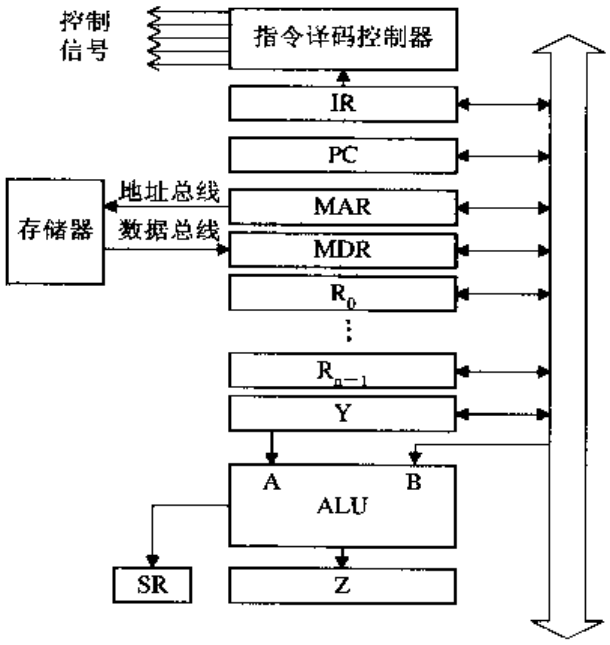


图 7.37　单总线的 CPU 结构

3．完成 8051 中的中断处理模块。

参 考 文 献

1 Rabaey Jan M. Digital Integrated Circuits: a design perspective. Prentice-Hall International, Inc. 1996

2 Hennessy John L. Patterson David A. Computer Organization & Design, the hardware/software interface. Morgan Kaufmann Publishers, Inc. 1998

3 Bhasker J. Verilog HDL Sythesis: A Praetical Primer. Star Galaxy Publishing, 1998

4 Michael John Sebnstian Smith. Application-Specific Integrated Circuits. Pearson Education North Asia Limited Inc.,1997

5 SYNOPSYS.Design Compiler User Guide

6 ALTERA. Quartus User Guide

7 XILINX. ISE User Guide

8 Mentor Graphics. ModelSim User Guide

9 Bruce Schneier 著. 应用密码学，协议、算法与 C 源程序. 吴世忠，祝世雄，张文政等译. 北京：机械工业出版社，2002

10 William Stallings 著. 计算机组织与结构——性能设计（第五版）. 张昆藏等译. 北京：电子工业出版社，2001

11 Steve Furber 著. ARM SoC 体系结构. 田泽，于敦山，盛世敏译. 北京：北京航空航天出版社，2002

12 余松煜，周源华，吴时光编著. 数字图像处理. 北京：电子工业出版社，1989

13 余松煜，张文军，孙军编著. 现代图像信息压缩技术. 北京：科学出版社，1998

14 朱正涌编著. 半导体集成电路. 北京：清华大学出版社，2001