



普通高等教育“十一五”
国家级规划教材



北京高等教育精品教材
BEIJING GAODENG JIAOYU JINGPIN JIAOCAI

Verilog 数字系统设计

Digital System Design

[第3版]

夏宇闻 编著

教程

Tutorial



北京航空航天大学出版社
BEIHANG UNIVERSITY PRESS

策划编辑：金友泉
封面设计：艺彩书装

上架建议：计算机应用

ISBN 978-7-5124-1186-9



9 787512 411869 >

定价：45.00元



普通高等教育“十一五”
国家级规划教材



北京高等教育精品教材
BEIJING GAODENG JIAOYU JINGPIN JIAOCAI

Verilog 数字系统设计教程 (第3版)

夏宇闻 编著

北京航空航天大学出版社

内 容 简 介

本书讲述利用硬件描述语言(Verilog HDL)设计复杂数字系统的方法。这种方法源自 20 世纪 90 年代的美国,取得成效后迅速在其他先进工业国得到推广和普及。利用硬件描述语言建模、通过仿真和综合技术设计出极其复杂的数字系统是这种技术的最大优势。

本书从算法和计算的基本概念出发,讲述如何用硬线逻辑电路实现复杂数字逻辑系统的方法。全书共分三部分。第一部分内容共 18 章;第二部分共 12 个上机练习实验范例;第三部分是 Verilog 硬件描述语言参考手册,可供读者学习、查询之用。本书第 2 版后,在语法篇中增加了 IEEE Verilog1364-2001 标准简介,以反映 Verilog 语法的最新变化。

本书的讲授方式以每 2 学时讲授一章为宜,每次课后需要花 10h 复习思考。完成 10 章学习后,就可以开始做上机练习,由简单到复杂,由典型到一般,循序渐进地学习 Verilog HDL 基础知识。按照书上的步骤,可以使大学电子类及计算机工程类本科及研究生,以及相关领域的设计工程人员在半年内掌握 Verilog HDL 设计技术。

本书可作为电子工程类、自动控制类、计算机类的大学本科高年级及研究生教学用书,亦可供其他工程人员自学与参考。

图书在版编目(CIP)数据

Verilog 数字系统设计教程 / 夏宇闻编著. --第 3 版

--北京 : 北京航空航天大学出版社, 2013.7

ISBN 978-7-5124-1186-9

I. ①V… II. ①夏… III. ①硬件描述语言—程序设计—教材 IV. ①TP312

中国版本图书馆 CIP 数据核字(2013)第 147932 号

版权所有,侵权必究。

Verilog 数字系统设计教程 (第 3 版)

夏宇闻 编著

责任编辑 金友泉

*

北京航空航天大学出版社出版发行

北京市海淀区学院路 37 号(邮编 100191) <http://www.buaapress.com.cn>

发行部电话:(010)82317024 传真:(010)82328026

读者信箱: goodtextbook@126.com 邮购电话:(010)82316936

涿州市新华印刷有限公司印装 各地书店经销

*

开本: 787 mm×1 092 mm 1/16 印张: 30.75 字数: 787 千字

2013 年 7 月第 3 版 2013 年 7 月第 1 次印刷 印数: 10 000 册

ISBN 978-7-5124-1186-9 定价: 45.00 元

若本书有倒页、脱页、缺页等印装质量问题,请与本社发行部联系调换。联系电话:(010)82317024

前　　言

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

半个世纪来,我国在复杂数字电路设计技术领域与国外的差距越来越大。作为一名在大学讲授专用数字电路与系统设计课程的老师深深感到责任的重大。作者认为,我国在这一技术领域的落后与大学的课程设置和教学条件有关。因为我们没有及时把国外最先进的设计方法和技术介绍给学生,也没有给他们创造实践的机会。1995年我受学校和系领导的委托,筹建世行贷款的电路设计自动化(EDA)实验室。通过二十年的摸索、实践,逐步掌握了利用Verilog HDL设计复杂数字电路的仿真和综合技术。在此期间我们为航天部等有关单位设计了卫星信道加密用的复杂数字电路,提供给他们经前后仿真验证的Verilog HDL源代码,得到很高的评价。在其后的几年中又为该单位设计了卫星下行信道RS(255,223)编码/解码电路和卫星上行信道BCH(64,56)编码/解码电路,这几个项目已先后通过有关单位的验收。1999年到2000年期间,我们又成功地设计了用于小波(Wavelet)图像压缩/解压缩的小波卷积器和改进的零修剪树算法(即SPIHT算法)的RTL级Verilog HDL模型。不但成功地对该模型进行了仿真和综合,而且制成的可重新配置硬线逻辑(采用ALTERA FLEX10K系列CPLD/10/30/50各一片)的PCI线路板,能完成约2000条C语句程序才能完成的图像/解压缩算法。运算结果与软件完成的完全一致,而且速度比用微型计算机快得多。2003年由作者协助指导的JPEG2000算法硬线逻辑设计,在清华同行的努力下完成了FPGA验证后并成功地投片,该芯片目前已应用于实时监控系统,可见这种新设计方法的潜力。近年来作者带领的研究生分别为日本某公司、香港科技大学电子系、革新科技公司和神州龙芯集成电路设计公司完成多项设计,其中包括SATA接口、AMBA总线接口、LED控制器和USB控制器等在内的多项IP设计,取得了良好的社会效益和声誉。2006年秋起,正式受聘于神州龙芯等集成电路设计公司担任技术顾问,目前在至芯科技公司担任FPGA设计培训顾问。

本书是在1998年北京航空航天大学出版社出版的《复杂数字电路与系统的Verilog HDL设计技术》、2003年《Verilog数字系统设计教程》和2008年《Verilog数字系统设计教程(第2版)》基础上修订的,是一本既有理论又有实践的设计大全。由于教学、科研、技术资料翻译和实验室的各项工作很忙,只能利用零碎时间,一点一滴地把积累的教学经验和新收集到的材料

补充输入到计算机中,抽空加以整理。我们使用 Verilog 设计复杂数字逻辑电路虽然已经有 18 年的时间,但仍旧在不断地学习提高之中,书中难免存在疏忽、错误之处,敬请细心的读者不吝指教。作者之所以在原版基础上把这本书再版,是想把原教材中一些不足的地方作一些必要的补充和修改,在大学生和研究生中加快 Verilog 设计技术的推广,尽快培养一批掌握先进设计技术的跨世纪的人才。期望本书能在这一过程中起到抛砖引玉的作用。

回想起来,这本书实质上是我们实验室全体老师和同学们多年的劳动成果,其中在 EDA 实验室工作过的历届研究生张琰、山岗、王静璇、田玉文、冯文楠、杨柳、傅红军、龚剑、王书龙、胡瑛、杨雷、邢伟、管丽、刘曦、王进磊、王煜华、苏宇、张云帆、杨鑫、徐伟俊、邢小地、霍强、宋成伟、邢志成、李鹏、李琪、陈岩、赵宗民等都帮我做了许多工作,如部分素材的翻译、整理、录入和一些 Verilog HDL 模块的设计修改和验证。

而我做的工作只是收集全书的素材、翻译、理解素材中一些较难的概念,结合教学经验编写一些章节和范例,以及全书文稿的最后组织、整理和补充,使其达到出版的要求。趁此机会让我衷心地感谢在编写本书过程中所有给过我帮助和鼓励的老师和同学们。本书是在第 2 版第 16 次印刷之后,受北航出版社之托进行的,虽然被称为第 3 版,然而本人在至芯科技的 FPGA 培训工作繁忙,没有时间对本书做大幅度的修改,望各位读者谅解。

教学中使用的多媒体课件已交付给出版社,有需要者可向北航出版社发行部或北京至芯科技公司索取,可以免费提供给有关教师指导教学和备课演示之用。

作者的通信邮箱是 xyw46@263.net,有什么问题可与作者商讨,谢谢!

编 者

2013 年 5 月 14 日

于北京至芯科技

目 录

绪 论	1
-----------	---

第一部分 Verilog 数字设计基础

第1章 Verilog的基本知识	10
1.1 硬件描述语言 HDL	10
1.2 Verilog HDL的历史	11
1.2.1 什么是Verilog HDL	11
1.2.2 Verilog HDL的产生及发展	11
1.3 Verilog HDL和VHDL的比较	12
1.4 Verilog的应用情况和适用的设计	13
1.5 采用Verilog HDL设计复杂数字电路的优点	13
1.5.1 传统设计方法——电路原理图输入法	13
1.5.2 Verilog HDL设计法与传统的电路原理图输入法的比较	14
1.5.3 Verilog 的标准化与软核的重用	14
1.5.4 软核、固核和硬核的概念及其重用	14
1.6 采用硬件描述语言(Verilog HDL)的设计流程简介	15
1.6.1 自顶向下(Top_Down)设计的基本概念	15
1.6.2 层次管理的基本概念	16
1.6.3 具体模块的设计编译和仿真的过程	16
1.6.4 具体工艺器件的优化、映像和布局布线	16
小 结	17
思 考 题	18
第2章 Verilog语法的基本概念	19
概 述	19
2.1 Verilog 模块的基本概念	20
2.2 Verilog 用于模块的测试	23
小 结	24
思 考 题	25

第3章 模块的结构、数据类型、变量和基本运算符号	26
概 述	26
3.1 模块的结构	26
3.1.1 模块的端口定义	26
3.1.2 模块内容	27
3.1.3 理解要点	28
3.1.4 要点总结	28
3.2 数据类型及其常量和变量	29
3.2.1 常 量	29
3.2.2 变 量	32
3.3 运算符及表达式	35
3.3.1 基本的算术运算符	35
3.3.2 位运算符	36
小 结	37
思 考 题	38
第4章 运算符、赋值语句和结构说明语句	39
概 述	39
4.1 逻辑运算符	39
4.2 关系运算符	40
4.3 等式运算符	40
4.4 移位运算符	41
4.5 位拼接运算符	41
4.6 缩减运算符	42
4.7 优先级别	42
4.8 关 键 词	43
4.9 赋值语句和块语句	43
4.9.1 赋值语句	43
4.9.2 块语句	45
小 结	48
思 考 题	49
第5章 条件语句、循环语句、块语句与生成语句	50
概 述	50
5.1 条件语句(if_else 语句)	50
5.2 case 语句	53
5.3 条件语句的语法	57
5.4 多路分支语句	58

5.5 循环语句.....	60
5.5.1 forever 语句	60
5.5.2 repeat 语句	60
5.5.3 while 语句	61
5.5.4 for 语句	61
5.6 顺序块和并行块.....	63
5.6.1 块语句的类型.....	63
5.6.2 块语句的特点.....	65
5.7 生成块.....	67
5.7.1 循环生成语句.....	68
5.7.2 条件生成语句.....	70
5.7.3 case 生成语句.....	71
5.8 举 例.....	72
5.8.1 四选一多路选择器.....	72
5.8.2 四位计数器.....	73
小 结	74
思 考 题	75
第6章 结构语句、系统任务、函数语句和显示系统任务	78
概 述	78
6.1 结构说明语句.....	78
6.1.1 initial 语句	78
6.1.2 always 语句	79
6.2 task 和 function 说明语句	82
6.2.1 task 和 function 说明语句的不同点	82
6.2.2 task 说明语句	83
6.2.3 function 说明语句	84
6.2.4 函数的使用举例.....	86
6.2.5 自动(递归)函数.....	88
6.2.6 常量函数.....	89
6.2.7 带符号函数	90
6.3 关于使用任务和函数的小结.....	90
6.4 常用的系统任务.....	91
6.4.1 \$ display 和 \$ write 任务	91
6.4.2 文件输出.....	94
6.4.3 显示层次.....	96
6.4.4 选通显示	96
6.4.5 值变转储文件.....	97
6.5 其他系统函数和任务	98

小结	98
思考题	99
第7章 调试用系统任务和常用编译预处理语句	100
概述	100
7.1 系统任务 \$monitor	100
7.2 时间度量系统函数 \$time	101
7.3 系统任务 \$finish	102
7.4 系统任务 \$stop	102
7.5 系统任务 \$readmemb 和 \$readmemh	103
7.6 系统任务 \$random	105
7.7 编译预处理	106
7.7.1 宏定义 `define	106
7.7.2 “文件包含”处理 `include	108
7.7.3 时间尺度 `timescale	111
7.7.4 条件编译命令 `ifdef、`else、`endif	112
7.7.5 条件执行	114
小结	115
思考题	116
第8章 语法概念总复习练习	117
概述	117
小结	128

第二部分 设计和验证部分

第9章 Verilog HDL 模型的不同抽象级别	130
概述	130
9.1 门级结构描述	130
9.1.1 与非门、或门和反向器及其说明语法	130
9.1.2 用门级结构描述 D 触发器	131
9.1.3 由已经设计成的模块构成更高一层的模块	132
9.2 Verilog HDL 的行为描述建模	133
9.2.1 仅用于产生仿真测试信号的 Verilog HDL 行为描述建模	134
9.2.2 Verilog HDL 建模在 Top-Down 设计中的作用和行为建模的可综合性问题	136
9.3 用户定义的原语	137
小结	138

思 考 题	139
第 10 章 如何编写和验证简单的纯组合逻辑模块	140
概 述	140
10.1 加法器	140
10.2 乘法器	142
10.3 比较器	145
10.4 多路器	146
10.5 总线和总线操作	148
10.6 流水线	149
小 结	154
思 考 题	155
第 11 章 复杂数字系统的构成	156
概 述	156
11.1 运算部件和数据流动的控制逻辑	156
11.1.1 数字逻辑电路的种类	156
11.1.2 数字逻辑电路的构成	156
11.2 数据在寄存器中的暂时保存	158
11.3 数据流动的控制	160
11.4 在 Verilog HDL 设计中启用同步时序逻辑	162
11.5 数据接口的同步方法	164
小 结	165
思 考 题	165
第 12 章 同步状态机的原理、结构和设计	166
概 述	166
12.1 状态机的结构	166
12.2 Mealy 状态机和 Moore 状态机的不同点	167
12.3 如何用 Verilog 来描述可综合的状态机	168
12.3.1 用可综合 Verilog 模块设计状态机的典型办法	168
12.3.2 用可综合的 Verilog 模块设计、用独热码表示状态的状态机	170
12.3.3 用可综合的 Verilog 模块设计、由输出指定的码表示状态的状态机	171
12.3.4 用可综合的 Verilog 模块设计复杂的多输出状态机时常用的方法	173
小 结	175
思 考 题	175
第 13 章 设计可综合的状态机的指导原则	177
概 述	177

13.1 用 Verilog HDL 语言设计可综合的状态机的指导原则	177
13.2 典型的状态机实例	178
13.3 综合的一般原则	180
13.4 语言指导原则	180
13.5 可综合风格的 Verilog HDL 模块实例	181
13.5.1 组合逻辑电路设计实例	181
13.5.2 时序逻辑电路设计实例	187
13.6 状态机的置位与复位	189
13.6.1 状态机的异步置位与复位	189
13.6.2 状态机的同步置位与复位	191
小 结	192
思 考 题	193
第 14 章 深入理解阻塞和非阻塞赋值的不同	194
概 述	194
14.1 阻塞和非阻塞赋值的异同	194
14.1.1 阻塞赋值	195
14.1.2 非阻塞赋值	196
14.2 Verilog 模块编程要点	196
14.3 Verilog 的层次化事件队列	197
14.4 自触发 always 块	198
14.5 移位寄存器模型	199
14.6 阻塞赋值及一些简单的例子	203
14.7 时序反馈移位寄存器建模	203
14.8 组合逻辑建模时应使用阻塞赋值	205
14.9 时序和组合的混合逻辑——使用非阻塞赋值	207
14.10 其他阻塞和非阻塞混合使用的原则	208
14.11 对同一变量进行多次赋值	209
14.12 常见的对于非阻塞赋值的误解	210
小 结	212
思 考 题	212
第 15 章 较复杂时序逻辑电路设计实践	213
概 述	213
小 结	224
思 考 题	224
第 16 章 复杂时序逻辑电路设计实践	226
概 述	226

16.1 二线制 I ² C CMOS 串行 EEPROM 的简单介绍	226
16.2 I ² C 总线特征介绍	226
16.3 二线制 I ² C CMOS 串行 EEPROM 的读写操作	227
16.4 EEPROM 的 Verilog HDL 程序	228
总 结	251
思 考 题	251
第 17 章 简化的 RISC_CPU 设计	252
概 述	252
17.1 课题的来由和设计环境介绍	252
17.2 什么是 CPU	253
17.3 RISC_CPU 结构	253
17.3.1 时钟发生器	255
17.3.2 指令寄存器	257
17.3.3 累加器	258
17.3.4 算术运算器	259
17.3.5 数据控制器	260
17.3.6 地址多路器	261
17.3.7 程序计数器	261
17.3.8 状态控制器	262
17.3.9 外围模块	268
17.4 RISC_CPU 操作和时序	269
17.4.1 系统的复位和启动操作	269
17.4.2 总线读操作	270
17.4.3 总线写操作	271
17.5 RISC_CPU 寻址方式和指令系统	271
17.6 RISC_CPU 模块的调试	272
17.6.1 RISC_CPU 模块的前仿真	272
17.6.2 RISC_CPU 模块的综合	286
17.6.3 RISC_CPU 模块的优化和布局布线	292
小 结	302
思 考 题	303
第 18 章 虚拟器件/接口、IP 和基于平台的设计方法及其在大型数字系统设计中的作用	304
概 述	304
18.1 软核和硬核、宏单元、虚拟器件、设计和验证 IP 以及基于平台的设计方法	304
18.2 设计和验证 IP 供应商	306
18.3 虚拟模块的设计	307
18.4 虚拟接口模块的实例	311
小 结	312

思考题	312
-----------	-----

第三部分 设计示范与实验练习

概述	313
练习一 简单的组合逻辑设计	314
练习二 简单分频时序逻辑电路的设计	316
练习三 利用条件语句实现计数分频时序电路	318
练习四 阻塞赋值与非阻塞赋值的区别	320
练习五 用 always 块实现较复杂的组合逻辑电路	322
练习六 在 Verilog HDL 中使用函数	324
练习七 在 Verilog HDL 中使用任务(task)	326
练习八 利用有限状态机进行时序逻辑的设计	329
练习九 利用状态机实现比较复杂的接口设计	332
练习十 通过模块实例调用实现大型系统的设计	337
练习十一 简单卷积器的设计	343
附录一 A/D 转换器的 Verilog HDL 模型机所需要的技术参数	357
附录二 2K * 8 位 异步 CMOS 静态 RAM HM - 65162 模型	361
练习十二 利用 SRAM 设计一个 FIFO	366

第四部分 语 法 篇

语法篇 1 关于 Verilog HDL 的说明	376
一、关于 IEEE 1364 标准	376
二、Verilog 简介	377
三、语法总结	377
四、编写 Verilog HDL 源代码的标准	379
五、设计流程	381
语法篇 2 Verilog 硬件描述语言参考手册	382
一、Verilog HDL 语句与常用标志符(按字母顺序排列)	382
二、系统任务和函数(System task and function)	448
三、常用系统任务和函数的详细使用说明	452
四、Command Line Options 命令行的可选项	463
五、IEEE Verilog 1364 - 2001 标准简介	464
参考文献	478
再版者的话	479

绪论

我们知道构成数字逻辑系统的基本单元是与门、或门和非门，它们都是由三极管、二极管和电阻等器件构成，并能执行相应的开关逻辑操作；与门、或门和非门又可以构成各种触发器，实现状态记忆。在数字电路基础课程中，在了解这些逻辑门和触发器的构成和原理后，把它们作为抽象的理想器件来考虑，学习如何用布尔代数和卡诺图简化方法来设计一些简单的组合逻辑电路和时序电路。这些基础知识从理论上了解一个复杂的数字系统，例如 CPU 等都可以由这些基本单元组成。但真正如何来设计一个极其复杂的数字系统，如何验证设计的逻辑系统功能是否正确，本教程就是讲解如何利用 Verilog 硬件描述语言来设计和验证这样一个复杂数字系统的方法。下面就复杂数字系统的概念、用途和几个有关的基本问题做一些说明。

1. 为什么要设计专用的复杂数字系统

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

在数字信号处理的领域内有相当大的一部分工作是可以事后处理的，即利用通用的计算机系统来处理这类问题。如在石油地质调查中，通过钻探和一系列的爆破，记录各种地层的回波数据，然后去除噪声等无用信息，并用计算机对这些数据进行处理，最后得到地层的构造，从而找到埋藏的石油。因为地层不会在几年内有明显的变化，因此花数十天乃至更长的时间把地层的构造分析清楚也能满足要求。这种类型的数字信号处理是非实时的，在通用的计算机上通过编写、修改和运行程序，分析程序运行的结果就能满足需要。

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

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

本教程的目的是想通过对数字信号处理、计算、算法和数据结构、编程语言和程序、体系结

构和硬线逻辑等基本概念的介绍,了解算法与硬线逻辑之间的关系,从而引入利用 Verilog HDL 硬件描述语言设计复杂的数字逻辑系统的概念和方法。向读者展示一种 20 世纪 90 年代才真正开始在美国等先进的工业化国家逐步推广的数字逻辑系统的设计方法,借助于这种方法,在电路设计自动化仿真和综合工具的帮助下,只要对并行计算微体系结构有一定程度的了解,对有关算法有深入的研究,就完全有能力设计并制造出具有自己知识产权的 DSP(数字信号处理)类和任何复杂的数字逻辑集成电路芯片,为我国的电子工业和国防现代化做出应有的贡献。

2. 数字信号处理

大规模集成电路设计制造技术和数字信号处理技术,30 多年来各自得到了迅速的发展。这两个表面上看来没有什么关系的技术领域实质上是紧密相关的。因为数字信号处理系统往往要进行一些复杂的数学运算和数据处理,并且又有实时响应的要求,它们通常是由高速专用数字逻辑系统或专用数字信号处理器所构成,通常包括高速数据通道接口和高速算法电路,其电路是相当复杂的。因此,只有在高速大规模集成电路设计制造技术进步的基础上,才有可能实现真正有意义的实时数字信号处理系统。对实时数字信号处理系统的要求不断提高,也推动了高速大规模集成电路设计制造技术的进步。现代专用集成电路的设计是借助于电子电路设计自动化(EDA)工具完成的。学习和掌握硬件描述语言(HDL)是使用电子电路设计自动化(EDA)工具的基础。

3. 计 算

说到数字信号处理,自然会想到数学计算。现代计算机和通信系统中广泛采用了数字信号处理的技术和方法。其基本思路是先把信号用一系列的数字来表示,把连续的模拟信号,通过采样和从模拟量到数字量的转换,把信号转换成一系列的数字信号,然后对这些数字信号进行各种快速的数学运算。其目的是多种多样的:有的是为了加密;有的是通过编码来减少误码率以提高信道的通信质量;有的是为了去掉噪声等无关的信息;有的是为了数据的压缩以减少占用的频道等。有时也把某些种类的数字信号处理运算称为变换,如离散傅里叶变换(DFT)、离散余弦变换(DCT)和小波变换(Wavelet T)等。

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

本书中凡提到“计算”这个词处,指的就是上面一段中 computing 所包含的意思。由传统的观点出发,可以从三个不同的方面来研究计算,即从教学、科学和工程的不同角度。由比较现代的观点出发,可以从四个主要的方面来研究计算,即从算法和数据结构、编程语言、体系结构、软件和硬件设计方法来研究。本绪论的目的是想让读者对设计复杂数字系统有一个全面的了解,从而加深对掌握 Verilog HDL 设计方法必要性的认识。一个复杂的数字系统设计往往是一个从算法到由硬线连接的门级逻辑结构,再映射到硅片的逐步实现的过程。因此,我们将从算法和数据结构、编程语言和程序、微体系结构和硬线逻辑以及设计方法学等方面的基本概念出发来研究和探讨用于数字信号处理等领域的复杂硬线逻辑电路的设计技术和方法,特

* 摘自 Denning et al, “Computing as a Discipline,” Communication of ACM, January, 1989。

别强调利用 Verilog 硬件描述语言的 TopDown 设计方法的介绍。

4. 算法和数据结构

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

5. 编程语言和程序

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

6. 系统的微体系结构和硬线连接的门级逻辑

计算电路究竟是如何构成的?为什么它能有效和正确地执行每一步程序?它能不能用另外一种结构方案来构成?运算速度还能不能再提高?所谓计算微体系结构就是回答以上问题,并从硬线逻辑和软件两个角度一起来探讨某种结构的计算机的性能潜力。比如, Von Neumann(冯·诺依曼)于 1945 年设计的 EDVAC 电子计算机,它的结构是一种最早的顺序机,该机执行标量数据的计算机系统结构。顺序机是从位串行操作到字并行操作,从定点运算到浮点运算逐步改进过来的。由于 Von Neumann 系统结构的程序是顺序执行的,所以速度很慢。随着硬件技术的进步,不断有新的计算机体系结构产生,其计算性能也在不断提高。计算机体系结构是一门讨论和研究通用计算机的中央处理器如何提高运算速度性能的学问。对计算机中央处理器微体系结构的了解是设计高性能的专用的硬线逻辑系统的基础,因此本书将通过一个简化的 RISC_CPU 的设计实例对系统结构的基本概念加以初步的介绍。但由于本书的重点是利用 Verilog HDL 进行复杂数字电路的设计技术和方法,大量的篇幅将介绍利用 HDL 进行设计的步骤、语法要点、可综合的风格要点、同步有限状态机和由浅入深的设计实例。至于有关处理器微体系结构的深入了解和高速标量计算逻辑的微结构等专门知识和设计诀窍,将在以后推出的新书中介绍。

7. 设计方法学

复杂数字系统的设计是一个把思想(即算法)转化为实际数字逻辑电路的过程。我们知道,同一个算法可以用不同结构的数字逻辑电路来实现,这从运算的结果来说可能是完全一致的,但其运算速度和性能价格比可以有很大的差别。可用许多种不同的方案来实现实时完成算法运算的复杂数字系统电路。下面列出了常用的四种方案:

第一种,以专用微处理机芯片为中心来完成算法所需的电路系统;

第二种,用高密度的 FPGA(从几万门到几百万门);

第三种,设计专用的大规模集成电路(ASIC);

第四种,利用现成的微处理机的 IP 核并结合专门设计的高速 ASIC 运算电路。

究竟采用什么方案要根据具体项目的技术指标、经费、时间进度和批量综合考虑而定。

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

8. 专用硬线逻辑与微处理器的比较

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

用高密度的 FPGA(从几万门到几百万门)来构成完成算法所需的电路系统也是一种较好的办法。必须购置有关的 FPGA 开发环境、布局布线和编程工具。有些 FPGA 厂商提供的开发环境不够理想,其仿真工具和综合工具性能不够完善,还需要利用性能较好的硬件描述语言仿真器、综合工具,才能有效地进行复杂的 DSP 硬线逻辑系统的设计。由于 FPGA 是一种通用的器件,它的基本结构决定了只对某一种特殊的应用,其性能不如专用的 ASIC 电路。

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

以上所述算法的专用硬线逻辑的实现都需要对算法和数据接口协议有深入的了解,还须掌握硬件描述语言和相关的 EDA 仿真、综合和布局布线工具。

9. C 语言、Matlab 与硬件描述语言在算法运算电路设计的关系和作用

数字电路设计工程师一般都学习过编程语言、数字逻辑基础、各种 EDA 软件工具的使用。就编程语言而言,国内外大多数学校都以 C 语言为标准,目前很少有学校使用 Pascal 和 Fortran;而 Matlab 则是一个常用的数学计算软件包,有许多现成的数学函数可以利用,大大节省了复杂函数的编程时间;Matlab 也提供手段可以与 C 程序模块方便地接口。因此用 Matlab 来做数学计算系统的行为仿真常常比直接用 C 语言方便,能很快生成有用的数据文件和表格,直接用于算法正确性的验证。

基础算法的描述和验证常用 C 语言来做。例如要设计 Reed Solomen 编码/解码器,必须先深入了解 Reed Solomen 编码/解码的算法,再编写 C 语言的程序来验证算法的正确性。运行描述编码器的 C 语言程序,把在数据文件中的多组待编码的数据转换为相应的编码后将数据并存入文件;再编写一个加干扰用的 C 语言程序,用于模拟信道。它能产生随机误码位(并

把误码位个数控制在纠错能力范围内)将其加入编码后的数据文件中。运行该加扰程序,产生带误码位的编码后的数据文件;然后再编写一个解码器的 C 语言程序,运行该程序把带误码位的编码文件解码为另一个数据文件。只要比较原始数据文件和生成的文件便可知道编码和解码的程序是否正确(能否自动纠正纠错能力范围内的错码位)。用这种方法就可以来验证算法的正确性。但这样的数据处理其运行速度只与程序的大小和计算机的运行速度有关,也不能独立于计算机而存在。如果要设计一个专门的电路来进行这种对速度有要求的实时数据处理,除了以上介绍的 C 程序外,还须编写硬件描述语言(如 Verilog HDL 或 VHDL)程序,进行仿真以便从电路结构上保证算法能在规定的时间内完成,并通过与前端和后端的设备接口正确无误地交换(输入/输出)数据。

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

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

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

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

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

(1) 为什么选择 C 语言与 Verilog HDL 配合使用?首先,C 语言很灵活,查错功能强,还可以通过 PLI(编程语言接口)编写自己的系统任务,并直接与硬件仿真器(如 Verilog—XL)结合使用。C 语言是目前世界上应用最为广泛的一种编程语言,因而 C 程序的设计环境比 Verilog HDL 更完整。此外,C 语言有可靠的编译环境,语法完备,缺陷较少,可应用于许多领域。比较起来,Verilog 语言只是针对硬件描述的,在别处使用(如用于算法表达等)并不方便。而且 Verilog 的仿真、综合、查错工具等大部分软件都是商业软件,与 C 语言相比缺乏长期大量的使用,可靠性较差,亦有很多缺陷。所以,只有在 C 语言的配合使用下,Verilog 才能更好地发挥作用。

面对上述问题,最好的方法是 C 语言与 Verilog HDL 语言相辅相成,互相配合使用。这就是既要利用 C 语言的完整性,又要结合 Verilog 对硬件描述的精确性,来更快更好地设计出符合性能要求的硬件电路系统。利用 C 语言完善的查错和编译环境,设计者可以先设计出一个功能正确的设计单元,以此作为设计比较的标准。然后,把 C 程序一段一段地改写成用并型结构(类似于 Verilog)描述的 C 程序,此时还是在 C 的环境里,使用的依然是 C 语言。如果运行结果正确,就将 C 语言关键字用 Verilog 相应的关键字替换,进入 Verilog 的环境。将测

试输入同时加到 C 与 Verilog 两个单元,将其输出做比较。这样很容易发现问题的所在,然后更正,再做测试,直至正确无误。剩下的工作就交给后面的设计工程师。

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

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

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

(3) 如何利用 C 语言来加快硬件的设计和查错:表 0.1 中列出了常用的 C 语言与 Verilog 语言相对应的关键字与控制结构。

表 0.1 C 语言与 Verilog 语言的比较

C 语言	Verilog 语言
sub - function	module, function, task
if - then - else	if - then - else
case	case
{,}	begin, end
for	for
while	while
break	disable
define	define
int	int
printf	monitor, display, strobe

表 0.2 中,列出了 C 语言与 Verilog 语言相对应的运算符和功能。

从上面的讨论可以总结如下:

① C 语言与 Verilog 硬件描述语言可以配合使用,辅助设计硬件。

② C 语言与 Verilog 硬件描述语言类似,只要稍加限制,C 语言的程序很容易转成 Verilog 的行为程序。

表 0.2 C 语言与 Verilog 语言相对应的运算符

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

美国和中国台湾地区逻辑电路设计和制造厂家大都以 Verilog HDL 为主,中国大陆地区目前学习使用 Verilog HDL 已经超过 VHDL。到底选用 Verilog 或是 VHDL 来配合 C 一起用,就留给各位同学自行去决定。但从学习的角度来看,Verilog HDL 比较简单,也与 C 语言较接近,容易掌握;而从使用的角度看,支持 Verilog 硬件描述语言的半导体厂家也较支持 VHDL 的多,从发展趋势看 Verilog 也比 VHDL 有更宽广的前途。

总 结

绪论全面介绍了信号处理与硬线逻辑设计的关系,以及有关的基本概念,引入了 Verilog 硬件描述语言,向读者展示一种 20 世纪 90 年代才真正开始在美国等先进的工业化国家逐步推广的数字逻辑系统的设计方法。在下面的各章里将分步骤地详细介绍这种设计方法。

思 考 题

1. 什么是信号处理电路？它通常由哪两大部分组成？
2. 为什么要设计专用的信号处理电路？
3. 什么是实时处理系统？
4. 为什么要用硬件描述语言来设计复杂的算法逻辑电路？
5. 能不能完全用 C 语言来代替硬件描述语言进行算法逻辑电路的设计？
6. 为什么在算法逻辑电路的设计中需要用 C 语言和硬件描述语言配合使用来提高设计效率？

第一部分 Verilog 数字设计基础

数字通信和自动化控制等领域的高速发展和世界范围内的高技术竞争对数字系统提出了越来越高的要求,特别是需要设计具有实时信号处理能力的专用集成电路,要求把包括多个 CPU 内核在内的整个电子系统综合到一个芯片(SOC)上。设计并验证这样复杂的电路及系统已不再是简单的个人劳动,而需要综合许多专家的经验和知识才能够完成。近 10 年来电路制造工艺技术进步非常迅速,目前国际上 60 nm 的制造工艺,已达到工业化生产的规模,而电路设计能力远远落后于制造技术的进步。在数字逻辑设计领域,迫切需要一种共同的工业标准来统一对数字逻辑电路及系统的描述,这样就能把系统设计工作分解为逻辑设计(前端)、电路实现(后端)和验证三个互相独立而又相关的部分。由于逻辑设计的相对独立性就可以把专家们设计的各种常用数字逻辑电路和组件(如 FFT 算法、DCT 算法部件,DDRAM 读写控制器等)建成宏单元(megcell)或软(固/硬)核,也称作 Soft(firm/hard)Core,即 IP(知识产权内核的英文缩写)库供设计者引用,设计者可以直接利用它们的行为模型设计并验证其他电路,以减少重复劳动,提高工作效率。电路的实现则可借助于综合工具和 IP 的重复利用,以及布局布线工具(与具体工艺技术有关)自动地完成。

Verilog HDL 和 VHDL 这两种工业标准的产生顺应了历史的潮流,因而得到了迅速的发展。美国、日本等国由于高级设计工程师人力资源成本远高于中国,所以,近年来把许多设计工作转移到中国大陆,以降低设计成本。作为新世纪的中国大学生和年轻的电子工程师应该尽早掌握这种新的设计方法,使我国在复杂数字电路及系统的设计竞争中逐步缩小与美国等先进的工业发达国家的差距。

第1章 Verilog 的基本知识

1.1 硬件描述语言 HDL

硬件描述语言(HDL, hardware description language)是一种用形式化方法来描述数字电路和系统的语言。数字电路系统的设计者利用这种语言可以从上层到下层(从抽象到具体)逐层描述自己的设计思想,用一系列分层次的模块来表示极其复杂的数字系统。然后利用电子设计自动化(以下简称为 EDA)工具逐层进行仿真验证,再把其中需要变为具体物理电路的模块组合经由自动综合工具转换到门级电路网表。接下去再用专用集成电路(ASIC)或现场可编程门阵列(FPGA)自动布局布线工具把网表转换为具体电路布线结构的实现。在制成物理器件之前,还可以用 Verilog 的门级模型(原语元件或 UDP)来代替具体基本元件。因其逻辑功能和延时特性与真实的物理元件完全一致,所以在仿真工具的支持下能验证复杂数字系统物理结构的正确性,使投片的成功率达到 100%。目前,这种称为高层次设计(high-level-design)的方法已被广泛采用。据统计,目前在美国硅谷约有 90%以上的 ASIC 和 FPGA 已采用 Verilog 硬件描述语言方法进行设计。

硬件描述语言的发展至今已有近 30 年的历史,并成功地应用于设计的各个阶段:建模、仿真、验证和综合等。到 20 世纪 80 年代,已出现了上百种硬件描述语言,并对设计自动化曾起到了极大的促进和推动作用。但是,这些语言一般各自面向特定的设计领域与层次,而且众多的语言使用户无所适从。因此急需一种面向设计的多领域、多层次、并得到普遍认同的标准硬件描述语言。进入 20 世纪 80 年代后期,硬件描述语言向着标准化的方向发展。最终,VHDL 和 Verilog HDL 语言适应了这种趋势的要求,先后成为 IEEE 标准。把硬件描述语言用于自动综合还只有 10 多年的历史。最近 10 多年来,用综合工具把可综合风格的 HDL 模块自动转换为具体电路发展非常迅速,大大地提高了复杂数字系统的设计生产率。在美国和日本等先进电子工业国,Verilog 语言已成为设计数字系统的基础。本书第一部分将通过具体例子,由浅入深地帮助同学们学习以下内容:

- (1) Verilog 的基本语法;
- (2) 简单的可综合 Verilog 模块与逻辑电路的对应关系;
- (3) 简单的 Verilog 测试模块和它的意义。

书中第二部分将通过较复杂的设计实例,帮助同学们掌握以下内容:

- (1) 如何编写复杂的多层次的可综合风格的 Verilog HDL 模块;
- (2) 如何用可综合的 Verilog 模块构成一个可靠的复杂 IP 软核和固核模块;
- (3) 如何借助于 Verilog 语言,并利用已有的虚拟行为模块对所设计的系统模块(由可综合的自主和商业 IP 模块组成)进行全面可靠的测试和验证(包括软/硬件协同测试的基本概念)。

1.2 Verilog HDL的历史

1.2.1 什么是Verilog HDL

Verilog HDL是硬件描述语言的一种,用于数字电子系统设计。该语言允许设计者进行各种级别的逻辑设计,进行数字逻辑系统的仿真验证、时序分析、逻辑综合。它是目前应用最广泛的一种硬件描述语言。据有关文献报道,目前在美国使用Verilog HDL进行设计的工程师大约有10多万人,全美国有200多所大学教授用Verilog硬件描述语言的设计方法。在我国台湾地区几乎所有著名大学的电子和计算机工程系都讲授Verilog有关的课程。

1.2.2 Verilog HDL的产生及发展

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

随着Verilog-XL算法的成功,Verilog HDL语言得到迅速发展。1989年,Cadence公司收购了GDA公司,Verilog HDL语言成为Cadence公司的私有财产。1990年,Cadence公司决定公开Verilog HDL语言,于是成立了OVI(Open Verilog International)组织来负责促进Verilog HDL语言的发展。基于Verilog HDL的优越性,IEEE于1995年制定了Verilog HDL的IEEE标准,即Verilog IEEE1364-1995;2001年发布了Verilog IEEE1364-2001标准;2005年SystemVerilog IEEE 1800-2005标准的公布,更使得Verilog语言在综合、仿真验证和模块的重用等性能方面都有大幅度的提高。

图1.1展示了Verilog的发展历史和未来。

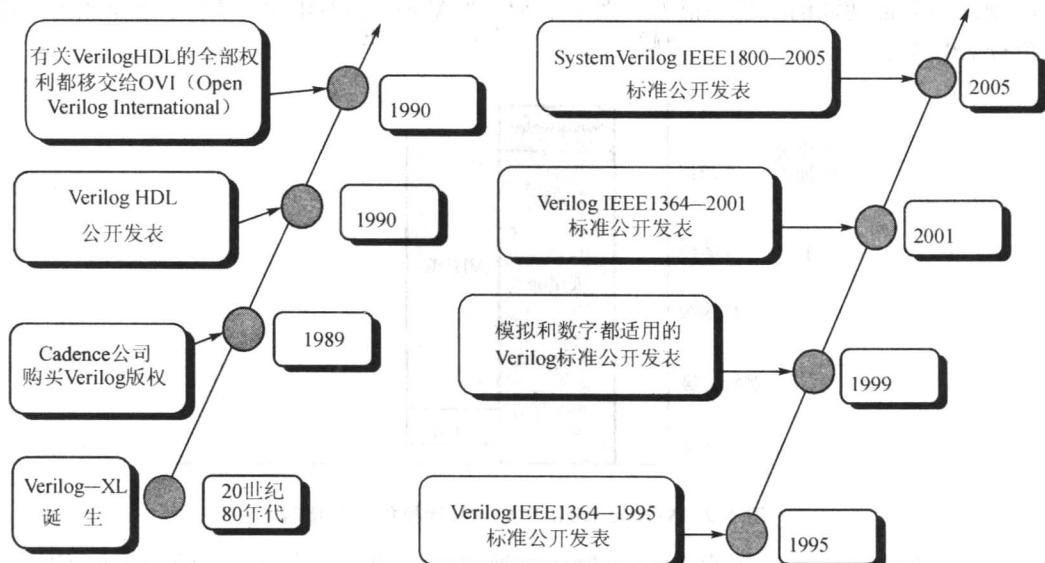


图1.1 Verilog HDL的发展历史和未来

1.3 Verilog HDL 和 VHDL 的比较

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

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

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

但是 Verilog HDL 和 VHDL 又各有其自己的特点。由于 Verilog HDL 早在 1983 年就已推出,至今已有 30 年的应用历史,因而 Verilog HDL 拥有更广泛的设计群体,成熟的资源也远比 VHDL 丰富。与 VHDL 相比 Verilog HDL 的最大优点是:它是一种非常容易掌握的硬件描述语言,只要有 C 语言的编程基础,通过 20 学时的学习,再加上一段实际操作,一般同学可在 2~3 个月内掌握这种设计方法的基本技术。而掌握 VHDL 设计技术就比较困难。这是因为 VHDL 不很直观,需要有 Ada 编程基础,一般认为至少需要半年以上的专业培训,才能掌握 VHDL 的基本设计技术。2005 年,SystemVerilog IEEE1800-2005 标准公布以后,集成电路设计界普遍认为 Verilog HDL 将在 10 年内全面取代 VHDL 成为 ASIC 设计行业包揽设计、测试和验证功能的唯一语言。图 1.2 所示为 Verilog HDL 和 VHDL 建模能力的比较图,供读者参考。

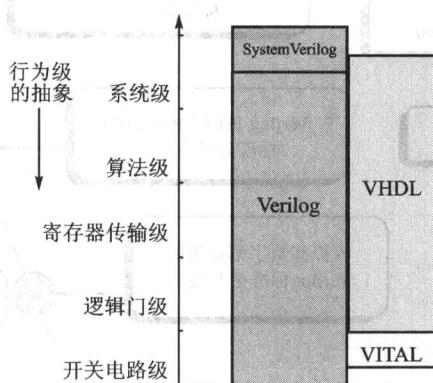


图 1.2 Verilog HDL 与 VHDL 建模能力的比较

2001 年公布的 VerilogIEEE1364-2001 标准和 2005 年公布的 SystemVerilog IEEE1800

-2005 标准,不但使 Verilog 的可综合性能和系统仿真性能方面有大幅度的提高,而且在 IP 的重用方面(包括设计和验证模块的重用)也有重大的突破。因此,Verilog HDL 不但作为学习 HDL 设计方法的入门和基础是比较合适的,而且对于 ASIC 设计专业人员而言,也是必须掌握的基本技术。学习掌握 Verilog HDL 建模、仿真、综合、重用和验证技术不仅可以使同学们对数字电路设计技术有更进一步的了解,而且可以为以后学习高级的行为综合、物理综合、IP 设计和复杂系统设计和验证打下坚实的基础。

1.4 Verilog 的应用情况和适用的设计

近 10 多年以来,EDA 界一直对在数字逻辑设计中究竟采用哪一种硬件描述语言争论不休。近 10 多年来,美国、日本和我国台湾地区电子设计界的情况已经清楚地表明,在高层次数字系统设计领域,Verilog 已经取得压倒性的优势;在中国大陆,近 10 多年来,Verilog 应用的比率已有显著的增加。据笔者了解,国内大多数集成电路设计公司都采用 Verilog HDL。Verilog 是专门为复杂数字系统的设计仿真而开发的,本身就非常适合复杂数字逻辑电路和系统的仿真和综合。由于 Verilog 在其门级描述的底层,也就是在晶体管开关的描述方面比 VHDL 有更强的功能,所以,即使是 VHDL 的设计环境,在底层实质上也是由 Verilog HDL 描述的器件库所支持的。1998 年通过的 Verilog HDL 新标准,把 Verilog HDL-A 并入 Verilog HDL 新标准,使其不仅支持数字逻辑电路的描述还支持模拟电路的描述,因此在混合信号的电路系统的设计中,它也有很广泛的应用。在深亚微米 ASIC 和高密度 FPGA 已成为电子设计主流的今天,Verilog 的发展前景是非常远大的。2001 年 3 月,Verilog IEEE1364-2001 标准的公布,以及 2005 年 10 月 SystemVerilog IEEE 1800-2005 标准的公布,使得 Verilog 语言在综合、仿真验证和 IP 模块重用等性能方面都有大幅度的提高,更加拓宽了 Verilog 的发展前景。作者本人的意见是:学习硬件描述语言的设计方法,则应首选 Verilog HDL。

Verilog 适合系统级(system)、算法级(algorithm)、寄存器传输级 RTL、逻辑级(logic)、门级(gate)、电路开关级(switch)设计,而 SystemVerilog 是 Verilog 语言的扩展和延伸,更适用于可重用的可综合 IP 和可重用的验证用 IP 设计,以及特大型(千万门级以上)基于 IP 的系统级设计和验证。

1.5 采用 Verilog HDL 设计复杂数字 电路的优点

1.5.1 传统设计方法——电路原理图输入法

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

近年来,FPGA 和 ASIC 的设计在规模和复杂度方面不断取得进展,而对逻辑电路及系统的设计时间要求却越来越短。这些因素促使设计人员采用高水准的设计工具,如:硬件描述语言(Verilog HDL 或 VHDL)来进行设计。

1.5.2 Verilog HDL 设计法与传统的电路原理图输入法的比较

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

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

1.5.3 Verilog 的标准化与软核的重用

Verilog 是在 1983 年由 GATEWAY 公司首先开发成功的,经过诸多改进,于 1995 年 11 月正式被批准为 Verilog IEEE1364-1995 标准,2001 年 3 月在原标准的基础上经过改进和补充又推出 Verilog IEEE1364-2001 新标准。2005 年 10 月又推出了 Verilog 语言的扩展,即 SystemVerilog (IEEE 1800-2005 标准)语言,这使得 Verilog 语言在综合、仿真验证和 IP 模块重用等性能方面都有大幅度的提高,更加拓宽了 Verilog 的发展前景。

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

1.5.4 软核、固核和硬核的概念及其重用

在 1.1.3 节中我们已介绍了软核的概念,下面再介绍一下固核(firm core)和硬核(hard core)的概念。把在某一种现场可编程门阵列(FPGA)器件上实现的、经验证是正确的、总门数在 5 000 门以上电路结构编码文件称为“固核”。把在某一种专用集成电路工艺的(ASIC)器件上实现的、经验证是正确的、总门数在 5 000 门以上的电路结构版图掩膜称为“硬核”。

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

1.6 采用硬件描述语言(Verilog HDL)的设计流程简介

1.6.1 自顶向下(Top_Down)设计的基本概念

现代集成电路制造工艺技术的改进,使得在一个芯片上集成数十万乃至数千万个器件成为可能。但很难设想仅由一个设计师独立设计如此大规模的电路而不出现错误。利用层次化、结构化的设计方法,一个完整的硬件设计任务首先由总设计师(Architect)划分为若干个可操作的模块,编制出相应的模型(行为的或结构的),通过仿真加以验证后,再把这些模块分配给下一层的设计师。这就允许多个设计者同时设计一个硬件系统中的不同模块,其中每个设计者负责自己所承担的部分;而由上一层设计师对其下层设计者完成的设计用行为级上层模块对其所做的设计进行验证。为了提高设计质量,如果其中有一部分模块可由商业渠道得到,用户可以购买它们的知识产权的使用权(IP核的重用),以节省时间和开发经费,图1.3为自顶向下(Top-Down)的示意图,以设计树的形式绘出。

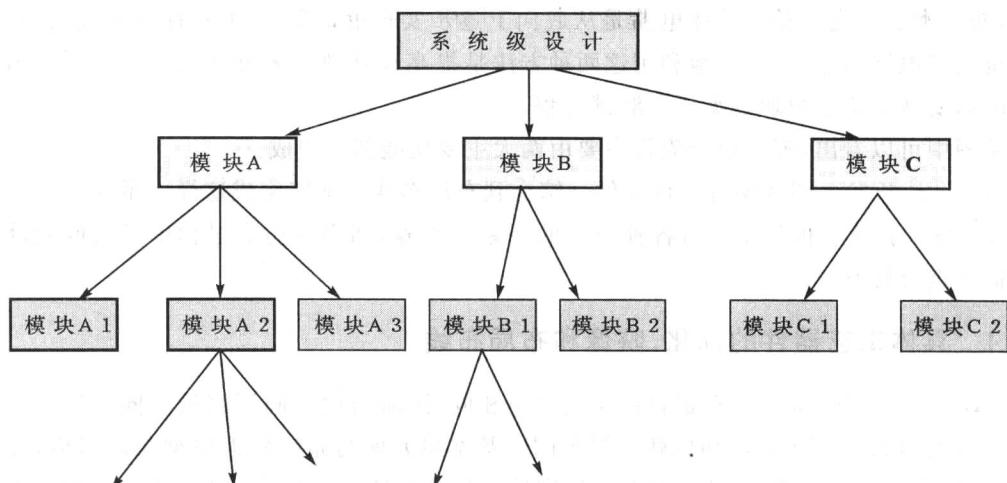


图 1.3 Top_Down 设计思想

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

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

1.6.2 层次管理的基本概念

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

1.6.3 具体模块的设计编译和仿真的过程

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

从图中可以看出,模块设计流程主要由两大主要功能部分组成:

- (1) 设计开发 即从编写设计文件→综合到布局布线→电路生成这样一系列步骤。
- (2) 设计验证 也就是进行各种仿真的一系列步骤,如果在仿真过程中发现问题就返回设计输入进行修改。

1.6.4 具体工艺器件的优化、映像和布局布线

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

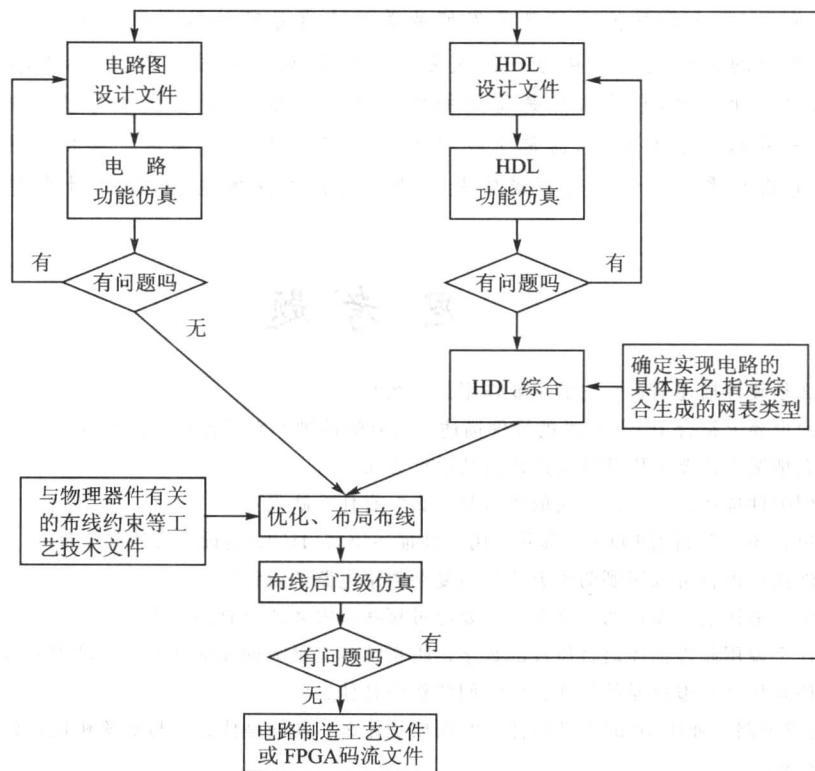


图 1.4 HDL 设计流程图

的库和模型进行准确的处理,从而大大提高设计效率。

小结

采用 Verilog HDL 设计方法比采用电路图输入的方法更有优越性,这就是为什么美国等国家在进入 20 世纪 90 年代以后纷纷采用 HDL 设计方法的原因。在两种符合 IEEE 标准的硬件描述语言中,Verilog HDL 与 VHDL 相比更加基础、更易学习,掌握 HDL 设计方法应从学习 Verilog HDL 设计方法开始。Verilog HDL 适用于复杂数字逻辑电路和系统的总体仿真、子系统仿真和具体电路综合等各个设计阶段。

由于 Top_Down 的设计方法是首先从系统设计入手,从顶层进行功能划分和结构设计。系统的总体仿真时顶层进行功能划分的重要环节,这时的设计是与工艺无关的。由于设计的主要仿真和调试过程是在高层次完成的所以能够早期发现结构设计上的错误,避免设计工作的浪费,同时也减少了逻辑仿真的工作量。自顶向下的设计方法方便了从系统级划分和管理整个项目,使得几十万门甚至几千万门规模的复杂数字电路的设计成为可能,并可减少设计人员,避免不必要的重复设计,提高了设计的一次成功率。

从底向上的设计在某种意义上讲可以看作上述 Top_Down 设计的逆过程。虽然设计也是从系统级开始,即从设计树的树根开始对设计进行逐次划分,但划分时首先考虑的是单元是

否存在,即设计划分过程必须从已经存在的基本单元出发,设计树最末枝上的单元要么是已经制造出的单元,要么是其他项目已开发好的单元或者是可外购得到的单元。

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

思 考 题

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

第 2 章 Verilog 语法的基本概念

概 述

Verilog HDL 是一种用于数字系统设计的语言。用 Verilog HDL 描述的电路设计就是该电路的 Verilog HDL 模型,也称为模块。Verilog HDL 既是一种行为描述的语言也是一种结构描述的语言。这就是说,无论描述电路功能行为的模块或描述元器件或较大部件互联的模块都可以用 Verilog 语言来建立电路模型。如果按照一定的规则和风格编写,功能行为模块可以通过工具自动地转换为门级互联的结构模块。Verilog 模型可以是实际电路的不同级别的抽象。这些抽象的级别和它们所对应的模型类型共有以下 5 种,现分别给以简述。

- (1) 系统级(system-level):用语言提供的高级结构能够实现待设计模块的外部性能的模型。
- (2) 算法级(algorithm-level):用语言提供的高级结构能够实现算法运行的模型。
- (3) RTL 级(register transfer level):描述数据在寄存器之间的流动和如何处理、控制这些数据流动的模型。

以上三种都属于行为描述,只有 RTL 级才与逻辑电路有明确的对应关系。

- (4) 门级(gate-level):描述逻辑门以及逻辑门之间连接的模型。

与逻辑电路有确定的连接关系,以上 4 种数字系统设计工程师必须掌握。

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

这与具体的物理电路有对应关系,工艺库元件和宏部件设计人员必须掌握,将在高级教程中介绍。

一个复杂电路系统的完整 Verilog HDL 模型是由若干个 Verilog HDL 模块构成的,每一个模块又可以由若干个子模块构成。其中有些模块需要综合成具体电路,而有些模块只是与用户所设计的模块有交互联系的现存电路或激励信号源。利用 Verilog HDL 语言结构所提供的这种功能就可以构造一个模块间的清晰层次结构,以此来描述极其复杂的大型设计,并对所作设计的逻辑电路进行严格的验证。

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

- 可描述顺序执行或并行执行的程序结构;
- 用延迟表达式或事件表达式来明确地控制过程的启动时间;
- 通过命名的事件来触发其他过程里的激活行为或停止行为;
- 提供了条件如 if-else,case 等循环程序结构;
- 提供了可带参数且非零延续时间的任务(task)程序结构;
- 提供了可定义新的操作符的函数结构(function);
- 提供了用于建立表达式的算术运算符、逻辑运算符、位运算符;
- Verilog HDL 语言作为一种结构化的语言非常适用于门级和开关级的模型设计。因

其结构化的特点又使它具有以下功能：

提供了一套完整的表示组合逻辑的基本元件的原语(primitive)；

提供了双向通路(总线)和电阻器件的原语；

可建立 MOS 器件的电荷分享和电荷衰减动态模型。

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

Verilog HDL 作为一种高级的硬件描述编程语言，与 C 语言的风格有许多类似之处。其中有许多语句，如 if 语句、case 语句和 C 语言中的对应语句十分相似。如果读者已经掌握 C 语言编程的基础，那么学习 Verilog HDL 并不困难，只要对 Verilog HDL 某些语句的特殊方面着重加以理解，并加强上机练习就能很好地掌握它，就能利用它的强大功能来设计复杂的数字逻辑电路系统。下面将对 Verilog HDL 中的基本语法通过实例逐一加以初步的介绍。

2.1 Verilog 模块的基本概念

下面先介绍几个简单的 Verilog HDL 程序，从中了解 Verilog 模块的特性。

【例 2.1】 图 2.1 所示的二选一多路选择器的 Verilog HDL 程序如下：

```
module muxtwo (out, a, b, sl);
    input a,b,sl;
    output out;
    reg out;
    always @ (sl or a or b)
        if (! sl) out = a;
        else out = b;
endmodule
```

从[例 2.1]中很容易理解模块 muxtwo 的作用。它是一个如图 2.1 所示的二选一多路器，输出 out 与输入 a 一致，还是与输入 b 一致，由 sl 的电平决定。当控制信号 sl 为非(低电平 0)时，输出 out 与输入 a 相同，否则与 b 相同。always @(sl or a or b) 表示只要 sl 或 a 或 b，其中若有一个变化时就执行下面的语句。人们并不关心它的电路结构，关心的是如何从逻辑功能上来描述它。Verilog 的语法支持这种逻辑行为的描述。

为了实现这个电路的逻辑功能，也能用布尔表达式来描述，Verilog 语言中，可以用“~”、“&”、“|”操作符分别表示：求反、相与和相或运算操作。所以[例 2.1]的 muxtwo 模块实现的逻辑功能也能用[例 2.2]形式的 Verilog 代码来表示。

【例 2.2】 图 2.2 所示的带有与非门的二选一多路选择器的 Verilog HDL 程序如下：

```
module muxtwo (out, a, b, sl);
    input a,b,sl;          //输入信号名
    output out;           //输出信号名
```

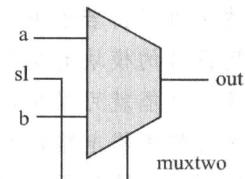


图 2.1 二选一多路器(一)

```

wire ns1,sela,selb;          //定义内部连接线
assign ns1=~sl;               //求反
assign sela=a&.ns1;           //按位与运算
assign selb=b&.sl;            //按位与运算
assign out=sela|selb;          //按位或运算
endmodule

```

上面例子中实现的逻辑功能是非常容易理解的,它从本质上就是一个逻辑表达式,表达的是一个二选一多路选择器,如[例 2.3]所示。

【例 2.3】 图 2.3 所示多路选择器的 Verilog HDL 程序如下:

```

module muxtwo (out,a,b,sl);
input a,b,sl;
output out;
not      u1(ns1,sl);
and #1 u2(sela,a,ns1);
and #1 u3(selb,b,sl);
or   #1 u4(out,sela,selb);
endmodule

```

从[例 2.3]中很容易理解模块 muxtwo 的作用。它也是一个二选一多路器,输出 out 与输入 a 一致,还是与输入 b 一致,由 sl 的电平决定。当控制信号 sl 为非(!)(低电平 0),输出 out 与输入 a 相同,否则与 b 相同。模块的描述用基本的与门、或门和非门的互联来描述。在程序模块中出现的 and、or 和 not 都是 Verilog 语言的保留字,由 Verilog 语言的原语(primitive)规定了它们的接口顺序和用法,分别表示与门、或门和非门,其中元件的输出口都规定在第一个端口,#1 和 #2 分别表示门输入到输出的延迟为 1 和 2 个单位时间;模块程序中的 u1、u2、u3、u4 与逻辑图中的逻辑元件对应,表示逻辑元件的实例名称。模块表示的是电路结构,跟程序右面的电路逻辑图表示完全一致的。Verilog 的语法也支持这种基于逻辑单元互联结构的描述。

如果在编写 Verilog 模块时,不但符合语法,还符合一些基本规则,就可以通过计算机上运行的工具把[例 2.1]通过[例 2.2]的中间形式自动转换为[例 2.3]形式的模块,这个过程称为综合(synthesis)。我们知道[例 2.3]模块很容易与某种工艺的基本元件逐一对应起来,再通过布局布线工具自动地转变为某种具体工艺的电路布线结构。在第一部分里除了讲解基本语法外,主要讲解符合何种风格的 Verilog 模块是可以综合的;何种风格的模块是不可以综合的;不可综合的 Verilog 模块有些什么作用等。

下面再看几个简单的模块,目的是初步了解 Verilog 语法规最重要的几个基本概念:并行性、层次结构性、可综合性,并了解测试平台(testbench)。

【例 2.4】 通过连续赋值语句描述一个 3 位加法器的 Verilog HDL 程序如下:

```

module adder ( cout,sum,a,b,cin );
input [2 : 0] a,b;
input      cin;

```

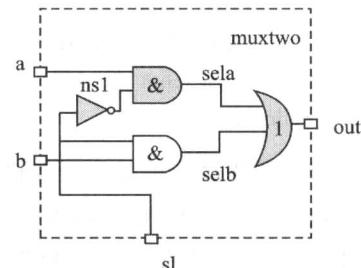


图 2.2 带有与非门的二选一多路器

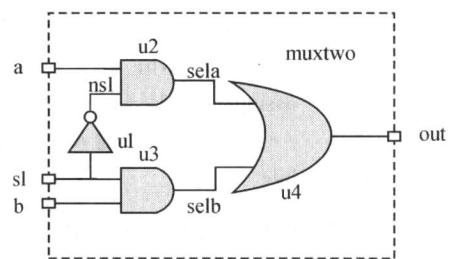


图 2.3 多路选择器(二)

```

    output cout;
    output [2 : 0] sum;
    assign {cout,sum} = a + b + cin;
endmodule

```

这个例子通过连续赋值语句描述了一个名为 adder 的 3 位加法器。它可以根据两个 3 比特数 a、b 和进位(cin)计算出和(sum)及向上进位(cout)。从例子中可以看出整个 Verilog HDL 程序是位于 module 和 endmodule 声明语句之间的。

【例 2.5】 通过连续赋值语句描述一个比较器的 Verilog HDL 程序如下：

```

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

```

此程序通过连续赋值语句描述了一个名为 compare 的比较器。对 2 比特数 a、b 进行比较,如 a 与 b 相等,则输出 equal 为高电平,否则为低电平。在这个程序中,/* */ 和//..... 表示注释部分。注释只是为了方便程序员理解程序,对编译是不起作用的。

【例 2.6】 图 2.4 所示的三态门选择器,其 verilog HDL 程序如下:

```

module trist2(out,in,enable);
    output out;
    input in, enable;
    bufif1 mybuf(out,in,enable);
endmodule

```

该程序描述了一个名为 trist2 的三态驱动器。程序通过调用一个在 Verilog 语言提供的原语(primitive)库中现存的三态驱动器元件 bufif1 来实现其逻辑功能。在 trist2 模块中所用到的三态驱动器元件 bufif1 的具体名字叫做 mybuf,这种引用现成元件或模块的方法叫做实例化或实例引用,这表示电路构造的一种常用语法现象。

【例 2.7】 采用二个模块的三态门选择器,其 Verilog HDL 程序如下:

```

module trist1(sout,sin,ena);
    output sout;
    input sin, ena;
    mytri tri_inst(.out(sout),.in(sin),.enable(ena));
    //引用由 mytri 模块定义的实例元件 tri_inst
endmodule

```

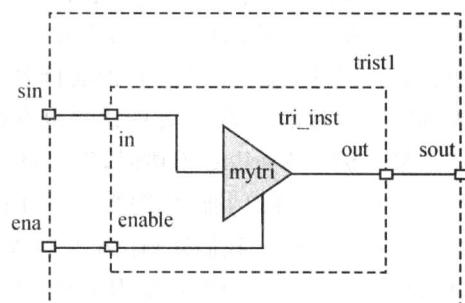


图 2.4 三态门选择器

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

该程序通过另一种方法描述了一个三态门。在这个例子中存在着两个模块：模块 trist1 引用由模块 mytri 定义的实例部件 tri_inst，模块 trist1 是上层模块；模块 mytri 则被称为子模块。在实例部件 tri_inst 中，带“.”表示被引用模块的端口，名称必须与被引用模块 mytri 的端口定义一致，小括号中表示在本模块中与之连接的线路。

上面这些例子都是可以综合的，通过综合工具可以自动转换为由与门、或门和非门组成的加法器、比较器和三态门等组合逻辑。在数字电路基础中已经学习过怎样用组合逻辑来实现 1 位或 2 位整数的加法和比较，而带超前进位链的多位整数加法器和多位比较器的逻辑图相当复杂，很难即时辨明。但这些也是已经成熟的电路结构，对于计算机支持的 EDA 工具来说，这只是一个映射的过程，系统设计人员就不必过于关心它们逻辑构成的细节，而把主要精力集中在系统结构的考虑上，从而大大提高了设计效率。

2.2 Verilog 用于模块的测试

Verilog 还可以用来描述变化的测试信号。描述测试信号的变化和测试过程的模块也称为测试平台 (testbench 或 testfixture)，它可以对上面介绍的电路模块（无论是行为的或结构的）进行动态的全面测试。通过观测被测试模块的输出信号是否符合要求，可以调试和验证逻辑系统的设计和结构正确与否，并发现问题及时修改。图 2.5 为 Verilog 时用于模块测试的原理图。

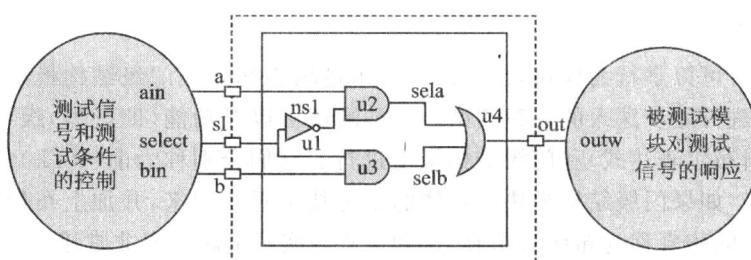


图 2.5 Verilog 用于模块测试

下面来看一个 Verilog 的测试模块，它可以对 [例 2.1] 到 [例 2.3] 的多路器模块进行逐步深入的全面测试。

【例 2.8】 对 [例 2.1]~[例 2.3] 多路器模块进行 Verilog HDL 程序如下：

```
'include "muxtwo.v"
module t;
reg ain, bin, select;
reg clock;
```

```

    wire    outw;
initial           //把寄存器变量初始化为一的确定值
begin
    ain = 0;
    bin = 0;
    select = 0;
    clock = 0;
end

always #50 clock = ~clock; //产生一个不断重复的周期为 100 个的时钟信号 clock

always @(posedge clock)
begin
    #1ain= { $random} %2; //产生随机的位信号流 ain 和 bin,%2 为模 2 运算
    #3bin= { $random} %2; //分别延迟 1 和 3 个时间单位后产生随机的位信号流 ain 和 bin
end

always #10000   select= ! select; //产生周期为 10 000 个单位时间的选通信号变化
//.....
    muxtwo m (.out(outw), .a(ain), .b(bin), .sl(select));
/* 实例引用多路器,并加入测试信号流,以观察模块的输出 out。其中,muxtwo 是已经
定义的(行为的或结构的)模块,m 表示在本测试模块中有一个名为 m 的 muxtwo 的模
块,其四个端口分别为:
    .out(), .a(), .b(), .sl(),
“.”表示端口;后面紧跟端口名,其名称必须与 muxtwo 模块定义的端口名一致;小括号内
的信号名为与该端口连接的信号线名,可以用别的名,但必须在本模块中定义,说明其类
型。 */
endmodule

```

其中 muxtwo 可以是行为模块,也可以是布尔逻辑表达式或门级结构模块。模块 t 可以对 muxtwo 模块进行逐步深入的完整测试。这种测试可以在功能(即行为)级上进行,也可以在逻辑网表(逻辑布尔表达式)和门级结构级上进行。它们分别称为前(RTL)仿真、逻辑网表仿真和门级仿真。如果门级结构模块与具体的工艺技术对应起来,并加上布局布线引入的延迟模型,此时进行的仿真称为布线后仿真,这种仿真与实际电路情况非常接近。可以通过运行仿真器,并观察输入/输出波形图来分析设计的电路模块的运行是否正确。

小 结

通过上面众多例子可以看到:

- (1) Verilog HDL 程序是由模块构成的。每个模块的内容都是位于 module 和 endmodule 两个语句之间。每个模块实现特定的功能。
- (2) 模块是可以进行层次嵌套的。正因为如此,才可以将大型的数字电路设计分割成不同的小模块来实现特定的功能。

(3) 如果每个模块都是可以综合的,则通过综合工具可以把它们的功能描述全都转换为最基本的逻辑单元描述,最后可以用一个上层模块通过实例引用把这些模块连接起来,把它们整合成一个很大的逻辑系统。

(4) Verilog 模块可以分为两种类型:一种是为了让模块最终能生成电路的结构,另一种只是为了测试所设计电路的逻辑功能是否正确。

(5) 每个模块要进行端口定义,并说明输入、输出口,然后对模块的功能进行描述。

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

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

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

思 考 题

1. Verilog 语言有什么作用?
2. 构成模块的关键词是什么?
3. 为什么说可以用 Verilog 构成非常复杂的电路结构?
4. 为什么可以用比较抽象的描述来设计具体的电路结构?
5. 是否任意抽象的符合语法的 Verilog 模块都可以通过综合工具转变为电路结构?
6. 什么叫综合?
7. 综合是由什么工具来完成的?
8. 通过综合产生的是什么?产生的结果有什么用处?
9. 仿真是什么?为什么要进行仿真?
10. 仿真可以在几个层面上进行?每个层面的仿真有什么意义?
11. 模块的端口是如何描述的?
12. 在引用实例模块的时候,如何在主模块中连接信号线?
13. 如何产生连续的周期性测试时钟?
14. 如果不用 initial 块,能否产生测试时钟?
15. 从本讲的简单例子,是否能明白 always 块与 initial 块有什么不同?
16. 为什么说 Verilog 可以用来设计数字逻辑电路和系统?

第3章 模块的结构、数据类型、变量和基本运算符号

概 述

在本章中将详细地学习 Verilog 语法中关于模块的结构、数据类型、变量和基本运算符号等语法要素。这些内容看起来简单,有许多语法现象和 C 语言也很类似,但有许多地方则是完全不同的。在学习中要注意不同点,并有意识地把新概念与硬件结构和测试联系起来,再通过理解物理意义,牢牢地记住。

3.1 模块的结构

Verilog 的基本设计单元是“模块”(block)。一个模块是由两部分组成的,一部分描述接口,另一部分描述逻辑功能,即定义输入是如何影响输出的。图 3.1 是模块结构组成图。下面举例说明。

图中的程序模块(见图 3.1(a))旁边有一个电路图的符号(见图 3.1(b))。在许多方面,程序模块和电路图符号是一致的,这是因为电路图符号的引脚也就是程序模块的接口。而程序模块描述了电路图符号所能实现的逻辑功能。上面的 Verilog 设计中,模块中的第二、第三行说明接口的信号流向,第四、第五行说明了模块的逻辑功能。以上就是设计一个简单的 Verilog 程序模块所需的全部内容。

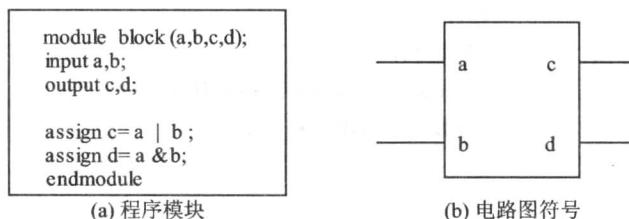


图 3.1 模块结构的组成

从这一例子可以看出,Verilog 结构位于在 module 和 endmodule 声明语句之间,每个 Verilog 程序包括 4 个主要部分:端口定义、I/O 说明、内部信号声明和功能定义。

3.1.1 模块的端口定义

模块的端口声明了模块的输入输出口。其格式如下:

```
module 模块名(口 1, 口 2, 口 3, 口 4, .....);
```

模块的端口表示的是模块的输入和输出口名,也就是说,它与别的模块联系端口的标识。

在模块被引用时，在引用的模块中，有些信号要输入到被引用的模块中，有的信号需要从被引用的模块中取出来。在引用模块时其端口可以用两种方法连接：

(1) 在引用时，严格按照模块定义的端口顺序来连接，不用标明原模块定义时规定的端口名，例如：

模块名(连接端口 1 信号名，连接端口 2 信号名，连接端口 3 信号名，……)；

(2) 在引用时用“.”符号，标明原模块是定义时规定的端口名，例如：

模块名(.端口 1 名(连接信号 1 名)，端口 2 名(连接信号 2 名)，……)；

这样表示的好处在于可以用端口名与被引用模块的端口相对应，而不必严格按端口顺序对应，提高了程序的可读性和可移植性。

例如：

.....

MyDesignMK M1(.sin(SerialIn), .pout(ParallelOut),……);

.....

其中，.sin 和.pout 都是 M1 的端口名，而 M1 则是与 MyDesignMK 完全一样的模块。MyDesignMK 已经在另一个模块中定义过，它有两个端口，即 sin 和 pout。与 sin 口连接的信号名为 SerialIn，与 pout 端口连接的是信号名为 ParallelOut。

3.1.2 模块内容

模块的内容包括 I/O 说明、内部信号声明和功能定义。

1. I/O 说明的格式

输入口： input [信号位宽-1:0] 端口名 1;

input [信号位宽-1:0] 端口名 2;

⋮

input [信号位宽-1:0] 端口名 i; // (共有 i 个输入口)

输出口： output [信号位宽-1:0] 端口名 1;

output [信号位宽-1:0] 端口名 2;

⋮

output [信号位宽-1:0] 端口名 j; // (共有 j 个输出口)

输入/输出口： inout [信号位宽-1:0] 端口名 1;

inout [信号位宽-1:0] 端口名 2;

⋮

inout [信号位宽-1:0] 端口名 k; // (共有 k 个双向总线端口)

I/O 说明也可以写在端口声明语句里。其格式如下：

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

2. 内部信号说明

在模块内用到的和与端口有关的 wire 和 reg 类型变量的声明。

如： reg [width-1 : 0] R 变量 1， R 变量 2…；

wire [width-1 : 0] W 变量 1， W 变量 2…；

:

3. 功能定义

模块中最重要的部分是逻辑功能定义部分。有3种方法可在模块中产生逻辑。

(1) 用“assign”声明语句 如:assign a = b & c;

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

(2) 用实例元件 如:and #2 u1(q, a, b);

采用实例元件的方法像在电路图输入方式下调入库元件一样,键入元件的名字和相连的引脚即可。这表示在设计中用到一个跟与门(and)一样的名为u1的与门,其输入端为a、b,输出为q。输出延迟为2个单位时间。要求每个实例元件的名字必须是唯一的,以避免与其他调用与门(and)的实例混淆。

(3) 用“always”块 如:always @ (posedge clk or posedge clr);

```
begin
    if(clr) q <= 0;
    else if(en) q <= d;
end
```

采用“assign”语句是描述组合逻辑最常用的方法之一。而“always”块既可用于描述组合逻辑,也可描述时序逻辑。用“always”块的例子生成了一个带有异步清除端的D触发器。“always”块可用很多种描述手段来表达逻辑,例如上例就用了if...else语句来表达逻辑关系。如按一定的风格来编写“always”块,可以通过综合工具把源代码自动综合成用门级结构表示的组合或时序逻辑电路。

3.1.3 理解要点

如果用Verilog模块实现一定的功能,首先应该清楚哪些是同时发生的,哪些是顺序发生的。上面3.1.2节中的(1)~(3)的3个例子分别采用了“assign”语句、实例元件和“always”块。这3个例子描述的逻辑功能是同时执行的。也就是说,如果把这3项写到一个Verilog模块文件中去,它们的顺序不会影响实现的功能。这3项是同时执行的,也就是并发的。

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

3.1.4 要点总结

Verilog的初学者一定要深入理解并记住:

(1) 在Verilog模块中所有过程块(如:initial块、always块)、连续赋值语句、实例引用都是并行的;

(2) 它们表示的是一种通过变量名互相连接的关系;

- (3) 在同一模块中这三者出现的先后秩序没有关系；
- (4) 只有连续赋值语句 assign 和实例引用语句可以独立于过程块而存在于模块的功能定义部分。

以上 4 点与 C 语言有很大的不同。许多与 C 语言类似的语句只能出现在过程块中，而不能随意出现在模块功能定义的范围内。

3.2 数据类型及其常量和变量

Verilog HDL 中总共有 19 种数据类型。数据类型是用来表示数字电路硬件中的数据储存和传送元素的。在本教材中先介绍 4 个最基本的数据类型，它们是：

reg 型、wire 型、integer 型和 parameter 型

其他数据类型在后面的章节里逐步介绍，也可以查阅第四部分中 Verilog 硬件描述语言参考手册的有关内容，以利逐步掌握。其他的类型是：

large 型、medium 型、scalared 型、time 型、small 型、tri 型、trio 型、tril 型、triand 型、trior 型、trireg 型、vectored 型、wand 型和 wor 型。

这 14 种数据类型除 time 型外都与基本逻辑单元建库有关，与系统设计没有很大的关系。在一般电路设计自动化的环境下，仿真用的基本部件库是由半导体厂家和 EDA 工具厂家共同提供的。系统设计工程师不必过多地关心门级和开关级的 Verilog HDL 语法现象。

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

3.2.1 常量

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

1. 数字

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

- 1) 二进制整数(b 或 B)；
- 2) 十进制整数(d 或 D)；
- 3) 十六进制整数(h 或 H)；
- 4) 八进制整数(o 或 O)。

数字表达方式有以下 3 种：

1) <位宽><进制><数字>，这是一种全面的描述方式。

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

3) 在<数字>这种描述方式中，采用默认进制(十进制)。

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

8'b10101100 //位宽为 8 的数的二进制表示，'b 表示二进制

8'ha2 //位宽为 8 的数的十六进制表示,'h 表示十六进制

(2) x 和 z 值 在数字电路中,x 代表不定值,z 代表高阻值。一个 x 可以用来定义十六进制数的 4 位二进制数的状态,八进制数的 3 位,二进制数的 1 位。z 的表示方式同 x 类似。z 还有一种表达方式是可以写作“?”。在使用 case 表达式时建议使用这种写法,以提高程序的可读性。见下例:

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

(3) 负数 一个数字可以被定义为负数,只需在位宽表达式前加一个减号,减号必须写在数字定义表达式的最前面。

注意:减号不可以放在位宽和进制之间,也不可以放在进制和具体的数之间。见下例:

```
-8'd5        //这个表达式代表 5 的补数(用八位二进制数表示)
8'd-5       //非法格式
```

(4) 下画线(underscore_) 下画线可以用来分隔开数的表达以提高程序可读性。它不可用在位宽和进制处,只能用在具体的数字之间。见下例:

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

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

```
10=32'd10=32'b1010
1=32'd1=32'b1
-1=-32'd1=32'hFFFFFFF
'BX=32'BX=32'BXXXXXXX...X
"AB"=16'B0100001_01000010 //字符串 AB,为十六进制数 16'h4142
```

2. 参数(parameter)型

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

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

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

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

```
parameter average_delay = (r+f)/2; //用常数表达式赋值
```

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

【例 3.1】 模块 Decode 定义时用了两个参数类型变量:Width 和 Polarity,且都为 1。在 Top 模块中引用 Decode 实例时,可通过参数的传递来改变定义时已规定的参数值。即通过 #(4,0),实例 D1 实际引用的是参数 Width 和 Polarity 分别为 4 与 0 时的 Decode 模块;通过 #(5),实例 D2 实际引用的是参数 Width 为 5,而 Polarity 仍为 1 时的 Decode 模块。这种利用参数编写模块的方法使得已编写的底层模块具有更大的灵活性。参数常用在表示门级模型的延迟,因此可通过参数传递表示不同的延迟。

```
module Decode(A,F);
    parameter Width=1, Polarity=1;
    :
endmodule

module Top;
    wire[3:0] A4;
    wire[4:0] A5;
    wire[15:0] F16;
    wire[31:0] F32;
    Decode #(4,0) D1(A4,F16);
    Decode #(5) D2(A5,F32);
endmodule
```

【例 3.2】 下面是一个由多层次模块构成的电路(见图 3.2)。在一个模块中改变另一个模块的参数时,需要使用 defparam 命令。在做布线后仿真时,就是利用这种方法把布线延迟通过布线工具生成的延迟参数文件反标注(Back-annotate)到门级 Verilog 网表上。

在模块 Annotate 中定义的参数值 2 和 3 可以通过模块 Test 中,经实例 T 对模块 Top 的引用,而在模块 Top 中,实例 B1 和 B2 对模块 Block 的引用,分别将参数值 2 和 3 传递到模块 Block 中用参数定义的地方(即原来在模块 Block 中定义的 P=0,在实例 B1 和 B2 中分别被 P=2 和 P=3 替代)。

```
'include "Top.v"
#include "Block.v"
#include "Annotate.v"

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

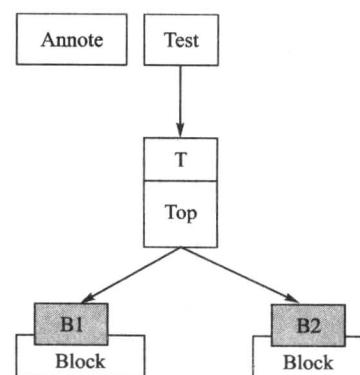


图 3.2 多层次模块构成的电路

```

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

module Block;
    Parameter P=0;
endmodule

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

```

3.2.2 变量

变量是一种在程序运行过程中其值可以改变的量,在 Verilog HDL 中变量的数据类型有很多种类,这里只对常用的几种进行介绍。

网络数据类型表示结构实体(例如门)之间的物理连接。网络类型的变量不能储存值,而且它必须受到驱动器(例如门或连续赋值语句,assign)的驱动。如果没有驱动器连接到网络类型的变量上,则该变量就是高阻的,即其值为 z。常用的网络数据类型包括 wire 型和 tri 型。这两种变量都是用于连接器件单元,它们具有相同的语法格式和功能。之所以提供这两种名字来表达相同的概念,其目的是为了与模型中所使用的变量的实际情况相一致。wire 型变量通常是用来表示单个门驱动或连续赋值语句驱动的网络型数据,tri 型变量则用来表示多驱动器驱动的网络型数据。如果 wire 型或 tri 型变量没有定义逻辑强度(logic strength),在多驱动源的情况下,逻辑值会发生冲突,从而产生不确定值。表 3.1 为 wire 型和 tri 型变量的真值表。

注意:表中假设两个驱动源的强度是一致的。关于逻辑强度建模可参阅第四部分中 Verilog 硬件描述语言参考手册。

表 3.1 wire/tri 变量的真值

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

1. wire 型

wire 型数据常用来表示用以 assign 关键字指定的组合逻辑信号。Verilog 程序模块中输

入、输出信号类型默认时自动定义为 wire 型。wire 型信号可以用做任何方程式的输入,也可以用做“assign”语句或实例元件的输出。

wire 型信号的格式同 reg 型信号的格式很类似。其格式如下:

wire [n-1:0] 数据名 1, 数据名 2, … 数据名 i; //共有 i 条总线,每条总线内有 n 条线路,
或 wire [n:1] 数据名 1, 数据名 2, … 数据名 i。

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

```
wire a;           //定义了一个 1 位的 wire 型数据  
wire [7 : 0] b;   //定义了一个 8 位的 wire 型数据  
wire [4 : 1] c, d; //定义了二个 4 位的 wire 型数据
```

2. reg 型

寄存器是数据储存单元的抽象。寄存器数据类型的关键字是 reg。通过赋值语句可以改变寄存器储存的值,其作用与改变触发器储存的值相当。Verilog HDL 语言提供了功能强大的结构语句,使设计者能有效地控制是否执行这些赋值语句。这些控制结构用来描述硬件触发条件,例如时钟的上升沿和多路器的选通信号。在行为模块介绍这一节中,还要详细地介绍这些控制结构。reg 类型数据的默认初始值为不定值 x。

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

reg 型数据的格式如下:

或
reg [n-1:0] 数据名 1, 数据名 2, …, 数据名 i;
reg [n:1] 数据名 1, 数据名 2, …, 数据名 i;

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

```
reg rega;           //定义了一个 1 位的名为 rega 的 reg 型数据  
reg [3:0] regb;    //定义了一个 4 位的名为 regb 的 reg 型数据  
reg [4:1] regc, regd; //定义了二个 4 位的名为 regc 和 regd 的 reg 型数据
```

对于 reg 型数据,其赋值语句的作用就如同改变一组触发器的存储单元的值。在 Verilog 中有许多构造(construct)用来控制何时或是否执行这些赋值语句。这些控制构造可用来描述硬件触发器的各种具体情况,如触发条件时用时钟的上升沿,或用来描述判断逻辑的细节,如各种多路选择器。

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

注意: reg 型只表示被定义的信号将用在“always”模块内,理解这一点很重要。并不是说 reg 型信号一定是寄存器或触发器的输出,虽然 reg 型信号常常是寄存器或触发器的输出,但并不一定总是这样。在本书中还会对这一点做更详细的解释。

3. memory 型

Verilog HDL通过对 reg 型变量建立数组来对存储器建模,可以描述 RAM 型存储器、ROM 存储器和 reg 文件。数组中的每一个单元通过一个数组索引进行寻址。在 Verilog 语言中没有多维数组存在。memory 型数据是通过扩展 reg 型数据的地址范围来生成的。其格式如下:

```
reg [n-1:0] 存储器名[m-1:0];
```

或 reg [n-1:0] 存储器名[m:1];

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

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

这个例子定义了一个名为 mema 的存储器,该存储器有 256 个 8 位的存储器。该存储器的地址范围是 0 到 255。

注意: 对存储器进行地址索引的表达式必须是常数表达式。

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

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

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

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

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

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

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

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

进行寻址的地址索引可以是表达式,这样就可以对存储器中的不同单元进行操作。表达式的值可以取决于电路中其他的寄存器的值。例如可以用一个加法计数器来做 RAM 的地址索引。本小节里只对以上几种常用的数据类型和常数进行了介绍,其余的在以后讲解

的示例中用到之处再逐一介绍。有兴趣者还可以参阅第四部分中 Verilog 硬件描述语言参考手册。

3.3 运算符及表达式

Verilog HDL 语言的运算符范围很广,其运算符按其功能可分为以下几类:

- (1) 算术运算符($+$, $-$, \times , $/$, $\%$);
- (2) 赋值运算符($=$, $<=$);
- (3) 关系运算符($>$, $<$, $>=$, $<=$);
- (4) 逻辑运算符($\&$, $\&$, $\|$, $!$);
- (5) 条件运算符($?:$);
- (6) 位运算符(\sim , $|$, \wedge , $\&$, $\sim\wedge$);
- (7) 移位运算符($<<$, $>>$);
- (8) 拼接运算符($\{ \}$);
- (9) 其他。

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

- (1) 单目运算符(unary operator):可以带一个操作数,操作数放在运算符的右边。
- (2) 双目运算符(binary operator):可以带两个操作数,操作数放在运算符的两边。
- (3) 三目运算符(ternary operator):可以带三个操作数,这三个操作数用三目运算符分隔开。见下例:

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

下面对常用的几种运算符进行介绍。

3.3.1 基本的算术运算符

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

- (1) $+$ (加法运算符,或正值运算符,如 $reg_a + reg_b$, $+ 3$);
- (2) $-$ (减法运算符,或负值运算符,如 $reg_a - 3$, $- 3$);
- (3) \times (乘法运算符,如 $reg_a * 3$);
- (4) $/$ (除法运算符,如 $5 / 3$);
- (5) $\%$ (模运算符,或称为求余运算符,要求%两侧均为整型数据。如 $7 \% 3$ 的值为 1)。

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

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

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

3.3.2 位运算符

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

- (1) ~ //取反
- (2) & //按位与
- (3) | //按位或
- (4) ^ //按位异或
- (5) ^~ //按位同或(异或非)

说 明:

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

(2) 位运算符中的双目运算符要求对两个操作数的相应位进行运算操作。

下面对各运算符分别进行介绍。

(1) “取反”运算符~: ~是一个单目运算符, 用来对一个操作数进行按位取反运算。其运算规则如表 3.2(a)所列。

举例说明:

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

(2) “按位与”运算符 &: “按位与”运算就是将两个操作数的相应位进行与运算。其运算规则如表 3.2(b)所列。

(3) “按位或”运算符 |: “按位或”运算就是将两个操作数的相应位进行或运算。其运算规则如表 3.2(c)所列。

(4) “按位异或”运算符 ^ (也称为 XOR 运算符): “按位异或”运算就是将两个操作数的相应位进行异或运算。其运算规则如表 3.2(d)所列。

(5) “按位同或”运算符 ^~: 按位同或运算就是将两个操作数的相应位先进行异或运算再进行非运算。其运算规则如表 3.2(e)所列。

表 3.2(a) 取反运算符的运算规则

\sim	结果
1	0
0	1
x	x

表 3.2(b) “按位与”运算规则

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

表 3.2(d) “按位异或”运算规则

\wedge	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

表 3.2(c) “按位或”运算规则

	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

表 3.2(e) “按位同或”运算规则

$\wedge\sim$	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

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

小 结

本章学习了基本语法的第一部分，回想一下，需要记住的基本概念是什么？Verilog 的初学者一定要深入理解并记住：

(1) 在 Verilog 模块中所有过程块(如：initial 块、always 块)、连续赋值语句、实例引用都是并行的。

(2) 它们表示的是一种通过变量名互相连接的关系。

(3) 在同一模块中各个过程块、各条连续赋值语句和各条实例引用语句这三者出现的先后次序没有关系。

(4) 只有连续赋值语句(即用关键词 assign 引出的语句)和实例引用语句(即用已定义的模块名引出的语句)，可以独立于过程块而存在于模块的功能定义部分。

(5) 被实例引用的模块，其端口可以通过不同名的连线或寄存器类型变量连接到别的模块相应的输出、输入信号端。

(6) 在“always”模块内被赋值的每一个信号都必须定义成 reg 型。

以上 6 点与 C 语言有很大的不同。许多与 C 语言类似的语句只能出现在过程块中，而不能随意出现在模块功能定义的范围内。

思 考 题

1. 模块由几个部分组成?
2. 端口分为几种?
3. 为什么端口要说明信号的位宽?
4. 能否说模块相当于电路图中的功能模块,端口相当于功能模块的引脚?
5. 模块中的功能描述可以由哪几类语句或语句块组成? 它们出现的顺序会不会影响功能的描述?
6. 这几类描述中哪一种直接与电路结构有关?
7. 最基本的 Verilog 变量有几种类型?
8. reg 型和 wire 型变量的差别是什么?
9. 由连续赋值语句(assign)赋值的变量能否是 reg 类型的?
10. 在 always 模块中被赋值的变量能否是 wire 类型的? 如果不能是 wire 类型,那么必须是什么类型的? 它们表示的一定是实际的寄存器吗?
11. 参数类型的变量有什么用处?
12. Verilog 语法规定的参数传递和重新定义功能有什么直接的应用价值?
13. 逻辑比较运算符小于等于“ $<=$ ”和非阻塞赋值大于等于“ $<=$ ”的表示是完全一样的,为什么 Verilog 在语句解释和编译时不会搞错?
14. 是否可以说实例引用的描述实际上就是严格意义上的电路结构描述?

第4章 运算符、赋值语句和结构说明语句

概 述

在本章中我们将学习 Verilog 语法中关于各种运算符、赋值语句、结构说明语句等基本语法要素。这些内容看起来简单,有许多语法现象和 C 语言也很类似,但有许多地方则是完全不同的,例如拼接运算符、缩减运算符、阻塞和非阻塞赋值运算符和结构说明语句中的并行块等。在学习中要注意这些不同点,有意识地把新概念与硬件结构与测试联系起来,通过理解物理意义,牢牢地掌握。

4.1 逻辑运算符

在 Verilog HDL 语言中存在 3 种逻辑运算符:

- (1) $\&\&$ 逻辑与;
- (2) \parallel 逻辑或;
- (3) $!$ 逻辑非。

“ $\&\&$ ”和“ \parallel ”是双目运算符,它要求有两个操作数,如 $(a>b)\&\&(b>c)$, $(a<b)\parallel(b<c)$ 。“!”是单目运算符,只要求一个操作数,如 $!(a>b)$ 。表 4.1 为逻辑运算的真值表,它表示当 a 和 b 的值为不同的组合时,各种逻辑运算所得到的值。

表 4.1 逻辑运算的真值

a	b	$! a$	$! b$	$a \& \& b$	$a \parallel b$
真	真	假	假	真	真
真	假	假	真	假	真
假	真	真	假	假	真
假	假	真	真	假	假

逻辑运算符中“ $\&\&$ ”和“ \parallel ”的优先级别低于关系运算符,“!”高于算术运算符。见下例:
 $(a>b)\&\&(x>y)$,可写成: $a>b \&\& x>y$;
 $(a==b)\parallel(x==y)$,可写成: $a==b \parallel x==y$;
 $(! a)\parallel(a>b)$,可写成: $! a \parallel a>b$ 。

为了提高程序的可读性,明确表达各运算符元间的优先关系,建议使用括号。

4.2 关系运算符

关系运算符共有以下4种：

- (1) $a < b$, 读作 a 小于 b;
- (2) $a > b$, 读作 a 大于 b;
- (3) $a \leq b$, 读作 a 小于或等于 b;
- (4) $a \geq b$, 读作 a 大于或等于 b。

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

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

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

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

4.3 等式运算符

在 Verilog HDL 语言中存在4种等式运算符：

- (1) == (等于);
- (2) != (不等于);
- (3) === (等于);
- (4) !== (不等于)。

注意：求反号、双等号、三个等号之间不能有空格。

这4个运算符都是双目运算符,它要求有两个操作数。“==”和“!=”又称为逻辑等式运算符,其结果由两个操作数的值决定。由于操作数中某些位可能是不定值x和高阻值z,结果可能为不定值x。而“==”和“!=”运算符则不同,它在对操作数进行比较时对某些位的不定值x和高阻值z也进行比较,两个操作数必须完全一致,其结果才是1,否则为0。“==”和“!=”运算符常用于case表达式的判别,所以又称为“case等式运算符”。这4个等式运算符的优先级别是相同的。表4.2列出“==与===”的真值表,帮助理解两者间的区别。

下面举一个例子说明“==”和“===”的区别：

```
if(A == 1'b1) $display("A is x");    (当 A 等于 x 时,这个语句不执行)
```

if(A==1'b1) \$display("A is x"); (当 A 等于 x 时,这个语句执行)

表 4.2 等式运算符的真值

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

4.4 移位运算符

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

$a >> n$ 或 $a << n$

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

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

从这一例子可以看出, $start$ 在移过两位以后, 用 0 来填补空出的位。

进行移位运算时应注意移位前后变量的位数, 下面将给出一例:

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

4.5 位拼接运算符

在 Verilog HDL 语言中有一个特殊的运算符:位拼接运算符(Concatation){}。用这个运算符可以把两个或多个信号的某些位拼接起来进行运算操作。其使用方法如下：

{信号 1 的某几位, 信号 2 的某几位, …, 信号 n 的某几位}

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

号。见下例：

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

也可以写成为

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

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

位拼接可以用重复法来简化表达式。见下例：

```
{4{w}} //这等同于{w,w,w,w}
```

位拼接还可以用嵌套的方式来表达。见下例：

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

用于表示重复的表达式,如上例中的4和3,必须是常数表达式。

4.6 缩减运算符

缩减运算符(reduction operator)是单目运算符,也有与、或、非运算。其与、或、非运算规则类似于位运算符的与、或、非运算规则,但其运算过程不同。位运算是对操作数的相应位进行与、或、非运算,操作数是几位数,其运算结果也是几位数。而缩减运算则不同,缩减运算是对单个操作数进行或、与、非递推运算,最后的运算结果是1位的二进制数。缩减运算的具体运算过程是这样的:第一步先将操作数的第1位与第2位进行或、与、非运算;第二步将运算结果与第3位进行或、与、非运算,依次类推,直至最后1位。例如:

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

相当于:

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

由于缩减运算的与、或、非运算规则类似于位运算符与、或非运算规则,这里不再详细讲述,可参阅位运算符的运算规则介绍。

4.7 优先级别

下面对各种运算符的优先级别关系作一总结,如表4.3所列。

表 4.3 各运算符的运算级别

优先级别	
!	最高优先级别
~	
*	
/	
%	
+	
-	
<<	
>>	
< <= > >=	
== ! = === !=	
&	
^	
^~	
&&	
:	
	最低优先级别

4.8 关键词

在 Verilog HDL 中,所有的关键词是事先定义好的确认符,用来组织语言结构。关键词是用小写字母定义的,因此在编写源程序时要注意关键词的书写,以避免出错。下面是 Verilog HDL 中使用的关键词(请参阅第四部分的 Verilog 硬件描述语言参考手册):

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

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

4.9 赋值语句和块语句

4.9.1 赋值语句

在 Verilog HDL 语言中,信号有两种赋值方式:

1. 非阻塞(Non_Blocking)赋值方式(如 `b<= a;`)

(1) 在语句块中,上面语句所赋的变量值不能立即就为下面的语句所用;

(2) 块结束后才能完成这次赋值操作,而所赋的变量值是上一次赋值得到的;

(3) 在编写可综合的时序逻辑模块时,这是最常用的赋值方法。

注意:非阻塞赋值符“ $<=$ ”与小于等于符“ \leq ”看起来是一样的,但意义完全不同,小于等于符是关系运算符,用于比较大小,而非阻塞赋值符用于赋值操作。

2. 阻塞(blocking)赋值方式(如 $b = a;$)

(1) 赋值语句执行完后,块才结束;

(2) b 的值在赋值语句执行完后立刻就改变的;

(3) 在时序逻辑中使用时,可能会产生意想不到的结果。

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

$b <= a;$

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

$b = a;$

这种赋值方式是马上执行的,也就是说执行下一条语句时, b 已等于 a 。尽管这种方式看起来很直观,但是可能引起麻烦。下面举例说明。

【例 4.1】

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

【例 4.1】的“always”块中用了非阻塞赋值方式,定义了两个 reg 型信号 b 和 c 。 clk 信号的上升沿到来时, b 就等于 a , c 就等于 b ,这里用到了两个触发器。

注意:赋值是在“always”块结束后执行的, c 应为原来 b 的值。这个“always”块实际描述的电路功能如图 4.1 所示。

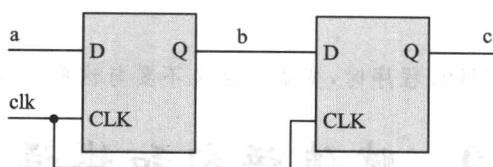


图 4.1 非阻塞赋值方式的“always”电路图

【例 4.2】

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

```
c=b;
```

```
end
```

[例 4.2]中的“always”块用了阻塞赋值方式。clk 信号的上升沿到来时,将发生如下的变化:b 马上取 a 的值,c 马上取 b 的值(即等于 a),生成的电路如图 4.2 所示,图中只用了一个触发器来寄存 a 的值,又输出给 b 和 c。这大概不是设计者的初衷,如果采用[例 4.1]所示的非阻塞赋值方式就可以避免这种错误。图 4.2 为阻塞值方式的“always”块图。

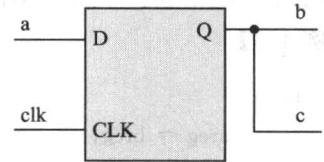


图 4.2 阻塞值方式的“always”电路图

关于赋值语句更详细的说明请参阅第二部分第 14 章中“深入理解阻塞和非阻塞赋值”一节。

4.9.2 块语句

块语句通常用来将两条或多条语句组合在一起,使其在格式上看更像一条语句。块语句有两种:一种是 begin_end 语句,通常用来标识顺序执行的语句,用它来标识的块称为顺序块;另一种是 fork_join 语句,通常用来标识并行执行的语句,用它来标识的块称为并行块。下面进行详细的介绍。

1. 顺序块

顺序块有以下特点:

- (1) 块内的语句是按顺序执行的,即只有上面一条语句执行完后下面的语句才能执行。
- (2) 每条语句的延迟时间是相对于前一条语句的仿真时间而言的。
- (3) 直到最后一条语句执行完,程序流程控制才跳出该语句块。

顺序块的格式如下:

```
begin
```

```
语句 1;
```

```
语句 2;
```

```
:
```

```
语句 n;
```

```
end
```

或

```
begin:块名
```

块内声明语句

```
语句 1;
```

```
语句 2;
```

```
:
```

```
语句 n;
```

```
end
```

其中:

(1) 块名即该块的名字,一个标识名,其作用后面再详细介绍。

(2) 块内声明语句可以是参数声明语句、reg型变量声明语句、integer型变量声明语句和real型变量声明语句。下面举例说明。

【例 4.3】

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

从该例可以看出,第1条赋值语句先执行,areg的值更新为breg的值,然后程序流程控制转到第2条赋值语句,creg的值更新为areg的值。因为这两条赋值语句之间没有任何延迟时间,creg的值实为breg的值。当然可以在顺序块里延迟控制时间来分开两个赋值语句的执行时间,见例4.4。

【例 4.4】

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

【例 4.5】

```
parameter d=50; //声明 d 是一个参数
reg [7:0] r; //声明 r 是一个 8 位的寄存器变量
begin
    //由一系列延迟产生的波形
    #d r = 'h35;
    #d r = 'hE2;
    #d r = 'h00;
    #d r = 'hF7;
    #d -> end_wave; //->表示触发事件 end_wave 使其翻转
end
```

这个例子中用顺序块和延迟控制组合可以产生一个时序波形。

2. 并行块

并行块有以下4个特点:

- (1) 块内语句是同时执行的,即程序流程控制一进入到该并行块,块内语句则开始同时并行地执行。
- (2) 块内每条语句的延迟时间是相对于程序流程控制进入到块内的仿真时间的。
- (3) 延迟时间是用来给赋值语句提供执行时序的。
- (4) 当按时间时序排序在最后的语句执行完后或一个 disable 语句执行时,程序流程控制跳出该程序块。

并行块的格式如下：

fork 块名 begin
语句 1;

语句 2;

⋮

语句 n;

join

或

fork; 块名 begin
块内声明语句

语句 1;

语句 2;

⋮

语句 n;

join

其中：

(1) 块名即标识该块的一个名字，相当于一个标识符。

(2) 块内说明语句可以是参数说明语句、reg型变量声明语句、integer型变量声明语句、real型变量声明语句、time型变量声明语句和事件(event)说明语句。下面举例说明。

【例 4.6】

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

在这个例子中用并行块替代[例 4.3]~[例 4.5]的顺序块来产生波形，用这两种方法生成的波形是一样的。

3. 块 名

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

(1) 可以在块内定义局部变量，即只在块内使用的变量。

(2) 可以允许块被其他语句调用，如 disable 语句。

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

基于以上原因，块名就提供了一个在任何仿真时刻确认变量值的方法。

4. 起始时间和结束时间

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

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

在 fork_join 块内,各条语句不必按顺序给出,因此在并行块里,各条语句在前还是在后是无关紧要的。见例 4.7。

【例 4.7】

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

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

小结

在本章中要注意几个问题:

(1) 无论是逻辑运算、逻辑比较还是逻辑等式等逻辑操作一般发生在条件判断语句中,其输出只有 1 或 0,也可以理解为成立(真)或不成立(假)。

(2) 位拼接运算符 { } 在 C 语言中没有定义,但在 Verilog 中是一种很有用的语法。可以借助于拼接符用一个信号名来表示由多位信号组成的复杂信号,其中每个功能信号可以有自己独立的名字和位宽。例如控制信号,可以用如下的位拼接来表示:

```
assign control={ read, write, sel[2:0], halt, load_instr, ...};
```

这样可以大大提高程序的可读性和可维护性。

(3) 缩减运算符(reduction operator)也是 C 语言所没有的,合理地使用缩减运算符可以使程序简洁、明了。

(4) 阻塞和非阻塞赋值也是 C 语言所没有的。我们应当理解这是非常重要的概念,特别是在编写可综合风格的模块中要加以注意。阻塞语句,如果没有写延迟时间看起来是在同一时刻运行,但实际上是有先后的,即在前面的先运行,然后再运行下面的语句,阻塞语句的次序与逻辑行为有很大的关系。而非阻塞的就不同了,在 begin end 之间的所有非阻塞语句都在同一时刻被赋值,因此逻辑行为与非阻塞语句的次序就没有关系。在硬件实现时这两者有很

大的不同。

(5) begin end 块语句与 C 语言中的大括号对(即{})类似,而 fork join 语句在 C 语言中没有定义,但其语义并不难理解。在测试模块中,描述测试信号常在 initial 和 always 过程块中使用并行块。这种描述方法,由于时间关系只与起点比较,有时这样表达比较容易和清楚。

思 考 题

1. 逻辑运算符与按位逻辑运算符有什么不同,它们各在什么场合使用?
2. 指出两种逻辑等式运算符的不同点,解释书上的真值表。
3. 拼接符的作用是什么?为什么说合理地使用拼接符可以提高程序的可读性和可维护性?拼接符表示的操作其物理意义是什么?
4. 如果都不带时间延迟、阻塞和非阻塞赋值有什么不同?举例说明它们的不同点?
5. 举例说明顺序块和并行块的不同。
6. 如果在顺序块中,前面有一条语句是无限循环,下面的语句能否进行?
7. 如果在并行块中,发生上述情况,会如何呢?

第 5 章 条件语句、循环语句、块语句与生成语句

概 述

在本章中将详细地学习 Verilog 语法中的条件语句、循环语句、块语句和生成语句的写法。其中生成语句是 Verilog1364-2001 版标准新添加的语句。这些语句中有些与 C 语言很类似，所以比较容易理解。但也有些与 C 语言不同，如块语句、生成语句、casex 和 casez 等。在学习中要注意这些不同点，有意识地把新概念与硬件结构与测试联系起来。只有通过大量的练习，深刻理解物理意义，才能牢牢地记住这些语法。

5.1 条件语句(if_else 语句)

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

1) if(表达式)语句。例如：

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

2) if(表达式)。

 语句 1

 else

 语句 2

例如： if(a>b)

```
        out1 = int1;  
        else  
            out1 = int2;
```

3) if(表达式 1)

 语句 1；

 else if(表达式 2) 语句 2；

 else if(表达式 3) 语句 3；

 ⋮

 else if (表达式 m) 语句 m；

 else 语句 n；

注意：条件语句必须在过程块语句中使用。所谓过程块语句是指由 initial 和 always 语句

引导的执行语句集合。除这两种块语句引导的 begin end 块中可以编写条件语句外,模块中的其他地方都不能编写。

例如: always@(*some_event*) //斜字体表示块语句

```
begin
    if(a>b)      out1 = int1;
    else          if (a==b)    out1 = int2; ←块语句
    else          out1 = int3;
end
```

6 点说明:

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

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

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

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

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

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

注意:在 end 后不需要再加分号。因为 begin_end 内是一个完整的复合语句,无须再附加分号。

(4) 允许一定形式的表达式简写方式。例如:

if(*expression*) 等同与 if(*expression* == 1)

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

(5) if 语句的嵌套。在 if 语句中又包含一个或多个 if 语句称为 if 语句的嵌套。一般形式

如下：

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

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

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

这时,begin_end 块语句限定了内嵌 if 语句的范围,因此 else 与第一个 if 配对。

注意:begin_end 块语句在 if_else 语句中的使用,因为有时 begin_end 块语句的不慎使用会改变逻辑行为。例如:

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

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

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

```
$ display("error-indexiszero");
```

(6) if_else 的例子。该例子是取自某程序中的一部分。这部分程序用 if_else 语句来检测变量 index, 以决定 3 个寄存器 modify_seg 中哪一个的值应当与 index 相加作为 memory 的寻址地址, 并且将相加的值存入寄存器 index 以备下次检测使用。程序的前 10 行定义寄存器和参数。

```
//定义寄存器和参数
reg [31:0] instruction, segment_area[255:0];
reg [7:0] index;
reg [5:0] modify_seg1, modify_seg2, modify_seg3;
parameter
    segment1=0,    inc_seg1=1,
    segment2=20,   inc_seg2=2,
    segment3=64,   inc_seg3=4,
    data=128;

//检测寄存器 index 的值
if(index<segment2)
begin
    instruction = segment_area[index + modify_seg1];
    index = index + inc_seg1;
end
else if(index<segment3)
begin
    instruction = segment_area[index + modify_seg2];
    index = index + inc_seg2;
end
else if (index<data)
begin
    instruction = segment_area[index + modify_seg3];
    index = index + inc_seg3;
end
else
instruction = segment_area[index];
```

5.2 case 语句

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

- (1) case(表达式) <case 分支项> endcase
- (2) casez(表达式) <case 分支项> endcase

(3) casex(表达式) <case 分支项> endcase

case 分支项的一般格式如下：

分支表达式： 语句；

默认项(default 项)： 语句；

说 明：

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

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

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

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

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

(4) 每一个 case 分项的分支表达式的值必须互不相同, 否则就会出现问题, 即对表达式的同一个值, 将出现多种执行方案, 产生矛盾。

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

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

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

下面将给出 case, casez, casex 的真值, 如表 5.1 所列。

表 5.1 case, casez 和 casex 的真值

case	0	1	x	z	casez	0	1	x	z	casex	0	1	x	z
0	1	0	0	0	0	1	0	0	1	0	1	0	1	1
1	0	1	0	0	1	0	1	0	1	1	0	1	1	1
x	0	0	1	0	x	0	0	1	1	x	1	1	1	1
z	0	0	0	1	z	1	1	1	1	z	1	1	1	1

case 语句与 if_else_if 语句的区别主要有两点：

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

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

【例 5.1】

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

【例 5.2】

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

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

【例 5.3】

```
reg[7:0] ir;
casez(ir)
  8'b1???????: instruction1(ir);
```

```

8'b01?????: instruction2(ir);
8'b00010???: instruction3(ir);
8'b000001???: instruction4(ir);
endcase

```

【例 5.4】

```

reg[7:0] r, mask;
mask = 8'b0x0x0x0x0;
casex(r掩码)
8'b001100xx: stat1;
8'b1100xx00: stat2;
8'b00xx0011: stat3;
8'bxx001100: stat4;
endcase

```

以下使用条件语句不当在设计中生成了原本没想到有的锁存器及其解决办法。

Verilog HDL 设计中容易犯的一个通病是由于对语言理解不全面, 使用不准确, 从而生成了并不想要的锁存器。图 5.1 给出了一个在“always”块中不正确使用 if 语句, 造成这种错误的例子。

<pre> always @ (al or d) begin if (al) q=d; end </pre> <p>有锁存器</p>	<pre> always @ (al or d) begin if (al) q=d; else q=0; end </pre> <p>无锁存器</p>
--	--

图 5.1 不正确使用 if 语句造成的错误

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

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

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

Verilog HDL 程序的另一种综合后生成没有预料到的锁存器是在使用 case 语句时缺少 default 项的情况下发生的。

case 语句的功能是: 在某个信号(本例中的 sel)取不同的值时, 给另一个信号(本例中的 q)赋不同的值。注意看如图 5.2 左边框内的例子, 如果 $sel=2'b00$, q 取 a 值; 而若 $sel=2'b11$, q 取 b 的值。这个例子中, 代码中并没有清楚地说明: 如果 sel 取 00 和 11 以外的值时 q 将被赋予什么值? 在图左边框内的这个用 Verilog HDL 写的例子中, 因为没有 default 分支语句, 所以默认为 q 保持原值, 因此, 在综合后的电路中就会自动生成锁存器。

<pre>always @ (sel [1 : 0] or a or b) case (sel [1 : 0]) 2'b00: q<=a; 2'b11: q<=b; endcase</pre> <p style="margin-top: 10px;">有锁存器</p>	<pre>always @ (sel [1 : 0] or a or b) case (sel [1 : 0]) 2'b00: q<=a; 2'b11: q<=b; default: q<='b0; endcase</pre> <p style="margin-top: 10px;">无锁存器</p>
--	--

图 5.2 缺少 default 项生成锁存器

右边例子的代码中, case 语句有 default 项, 明确指明了如果 sel 不取 00 或 11 时, 应赋予 q 的值。如图右框内程序所示, q 赋值为 0, 因此综合后不会生成锁存器。

以上介绍的方法可以避免 Verilog 代码处综合后的电路中生成锁存器。如果用到 if 语句, 最好写上 else 项; 如果用 case 语句, 最好写上 default 项。遵循上面两条原则, 就可以避免发生这种错误, 使设计者更加明确设计目标, 同时也增强了 Verilog 程序的可读性。

5.3 条件语句的语法

条件语句用于根据某个条件来确定是否执行其后的语句, 关键字 if 和 else 用于表示条件语句。Verilog 语言共有 3 种类型的条件语句, 条件语句的用法如下所示。

```
//第一类条件语句:没有 else 语句
//其后的语句执行或不执行
if (< expression >)  true_statement;

//第二类条件语句:有一条 else 语句
//根据表达式的值,决定执行 true_statement 或者 false_statement
if (< expression >)  true_statement ;  else false_statement;

//第三类条件语句:嵌套的 if_elsif 语句
//可供选择的语句有许多条,只有一条被执行
if (< expression1 >)  true_statement1 ;
else  if (< expression2 >)  true_statement2 ;
else  if (< expression3 >)  true_statement3 ;
else  default_statement ;
```

条件语句的执行过程为: 计算条件表达式<expression>, 如果结果为真(1 或非零值), 则执行 true_statement 语句; 如果条件为假(0 或不确定值 x), 则执行 false_statement 语句。在条件表达式中可以包含任何操作符。true_statement 和 false_statement 可以是一条语句, 也可以是一组语句。如果是一组语句, 则通常使用 begin、end 关键字将它们组成一个块语句。具体的使用方法如下:

[例 5.5] 条件语句举例。

```

//第一类条件语句
if ( ! lock ) buffer = data ;
if ( enable ) out = in ;

//第二类条件语句
if ( number_queued<MAX_Q_DEPTH )
begin
    data_queue = data ;
    number_queued = number_queued + 1 ;
end
else
    $ display ("Queue Full. Try again") ;

//第三类条件语句
//根据不同的算术逻辑单元的控制信号 alu_control 执行不同的算术运算操作
if ( alu_control == 0 )
    y = x + z ;
else if ( alu_control == 1 )
    y = x - z ;
else if ( alu_control == 2 )
    y = x * z ;
else
    $ display ("Invalid ALU control signal");

```

5.4 多路分支语句

5.3节所讲述的第三类条件语句使用 `if_elsif_if` 的形式从多个选项中确定一个结果。如果选项的数目很多,那么使用起来很不方便。而使用 `case` 语句来描述这种情况是非常简便的。

`case` 语句使用关键字 `case`、`endcase` 和 `default` 来表示。

```

case (expression)
    alternative1 : statement1 ;
    alternative2 : statement2 ;
    alternative3 : statement3 ;
    :
    default : default_statement
endcase

```

`case` 语句中的每一条分支语句都可以是一条语句或一组语句。多条语句需要使用关键字 `begin_end` 组合为一个块语句。在执行时,首先计算条件表达式的值,然后按顺序将它和各个候选选项进行比较:如果等于第一个候选选项,则执行对应的语句 `statement1`;如果和全部候选项

都不相等，则执行 default_statement 语句。

注意：default_statement 语句是可选的，而且在一条 case 语句中不允许有多条 default_statement。另外，case 语句可以嵌套使用。

下面的 Verilog 代码实现[例 5.5]中的第三类条件语句。

```
//根据不同的 alu_control 信号,执行不同的语句
reg [1:0] alu_control ;
:
:
case (alu_control)
 2'd0 : y = x + z ;
 2'd1 : y = x - z ;
 2'd2 : y = x * z ;
default : $display ("Invalid ALU control signal") ;
endcase
```

case 语句的行为类似于多路选择器。为了说明这一点，可使用 case 语句来对 5.8.1 节中的四选一多路选择器建模。由于只是实现方式的改变，因此模块的 I/O 端口保持不变。从该例子中可以看到，八选一或十六选一多路选择器也很容易用 case 语句来实现。

[例 5.6] 使用 case 语句实现四选一多路选择器。

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

//根据输入/输出图的端口声明
output out;
input i0, i1, i2, i3;
input s1, s0;

//把输出变量声明为寄存器类型
reg out;

//任何输入信号改变,都会引起输出信号的重新计算
//使输出 out 重新计算的所有输入信号必须写入 always @(...)的变量列表中

always @(s1 or s0 or i0 or i1 or i2 or i3)
begin
  case ({s1, s0})
    2'b00: out = i0;
    2'b01: out = i1;
    2'b10: out = i2;
    2'b11: out = i3;
    default: out = 1'bx;
  endcase
end

endmodule
```

5.5 循环语句

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

- (1) forever 语句: 连续的执行语句。
- (2) repeat 语句: 连续执行一条语句 n 次。
- (3) while 语句: 执行一条语句直到某个条件不满足。如果一开始条件即不满足(为假), 则语句一次也不能被执行。
- (4) for 语句: 通过以下 3 个步骤来决定语句的循环执行。

① 先给控制循环次数的变量赋初值。

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

③ 执行一条赋值语句来修正控制循环变量次数的变量值,然后返回第②步。

下面对各种循环语句详细的进行介绍。

5.5.1 forever 语句

forever 语句的格式如下:

forever 语句;

或 forever begin 多条语句 end

forever 循环语句常用于产生周期性的波形,用来作为仿真测试信号。它与 always 语句不同之处在于不能独立写在程序中,而必须写在 initial 块中。其具体使用方法将在“事件控制”这一小节里详细地加以说明。

5.5.2 repeat 语句

repeat 语句的格式如下:

repeat(表达式)语句

或 repeat(表达式) begin 多条语句; end

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

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

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

```

begin
    if(shift_opb[1])
        result = result + shift_opa;
    shift_opa = shift_opa <<1;
    shift_opb = shift_opb >>1;
end

```

5.5.3 while 语句

while 语句的格式如下：

while(表达式)语句；或用如下格式：

while(表达式) begin 多条语句;end

下面举一个 while 语句的例子，该例子用 while 循环语句对 rega 这个八位二进制数中值为 1 的位进行计数。

```

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

```

5.5.4 for 语句

for 语句的一般形式为：

for(表达式 1; 表达式 2; 表达式 3)语句；

它的执行过程如下：

- (1) 先求解表达式 1。
- (2) 求解表达式 2, 若其值为真(非 0), 则执行 for 语句中指定的内嵌语句, 然后执行下面的第(3)步。若为假(0), 则结束循环, 转到第(5)步。
- (3) 若表达式为真, 在执行指定的语句后, 求解表达式 3。
- (4) 转回上面的第(2)步骤继续执行。
- (5) 执行 for 语句下面的语句。

for 语句最简单的应用形式是很易理解的, 其形式如下：

for (循环变量赋初值; 循环结束条件; 循环变量增值)

 执行语句;

for 循环语句实际上相当于采用 while 循环语句建立以下的循环结构:

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

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

下面分别举两个使用 for 循环语句的例子。[例 5.7]用 for 语句来初始化 memory。

[例 5.8]则用 for 循环语句来实现前面用 repeat 语句实现的乘法器。

[例 5.7]

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

[例 5.8]

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

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

在 for 语句中, 循环变量增值表达式可以不必是一般的常规加法或减法表达式。下面是对 rega 这个八位二进制数中值为 1 的位进行计数的另一种方法。见下例:

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

```

if (tempreg[0])
    count = count+1;
end

```

5.6 顺序块和并行块

块语句的作用是将多条语句合并成一组,使它们像一条语句那样。在前面的例子中,可使用关键字 begin 和 end 将多条语句合并成一组。由于这些语句需要一条接一条的顺序执行,因此常称为顺序块。在本节中,将讨论 Verilog 语言中的块语句:顺序块和并行块;还将讨论 3 种有特点的块语句:命名块、命名块的禁用以及嵌套的块。

5.6.1 块语句的类型

块语句包括两种类型:顺序块和并行块。

1. 顺序块(也称过程块)

关键字 begin – end 用于将多条语句组成顺序块。顺序块具有以下特点:

(1) 顺序块中的语句是一条接一条按顺序执行的,只有前面的语句执行完成之后才能执行后面的语句(除了带有内嵌延迟控制的非阻塞赋值语句)。

(2) 如果语句包括延迟或事件控制,那么延迟总是相对于前面那条语句执行完成的仿真时间的。

在前面的各章节中,已经使用了许多顺序块的例子。在 [例 5.9] 中进一步给出了两个顺序块语句的例子。顺序块之中语句按顺序执行,[例 5.9] 的说明 1 中,在仿真 0 时刻 x,y,z,w 的最终值分别为 0,1,1,2。执行这 4 个赋值语句有顺序,但不需要执行时间。在说明 2 中,这 4 个变量的最终值也是 0,1,1,2,但块语句完成时的仿真时刻为 35,因为除第一句外,以后每执行一条语句都需要等待。

[例 5.9] 顺序块。

```
//说明 1
reg x, y;
reg [1 : 0] z, w;
```

initial

```
begin
    x = 1'b0;
    y = 1'b1;
    z = {x, y};
    w = {y, x};
end
```

```
//说明 2:带延迟的顺序块
```

```
reg x, y;
reg [1 : 0] z, w;
```

```

initial
begin
    x = 1'b0;           //在仿真时刻 0 完成
    #5 y = 1'b1;        //在仿真时刻 5 完成
    #10 z = {x, y};     //在仿真时刻 15 完成
    #20 w = {y, x};     //在仿真时刻 35 完成
end

```

2. 并行块

并行块由关键字 fork – join 声明,它的仿真特点是很有意思的。并行块具有以下特性:

(1) 并行块内的语句并发执行;

(2) 语句执行的顺序是由各自语句内延迟或事件控制决定的;

(3) 语句中的延迟或事件控制是相对于块语句开始执行的时刻而言的。

注意:顺序块和并行块之间的根本区别在于:当控制转移到块语句的时刻,并行块中所有的语句同时开始执行,语句之间的先后顺序是无关紧要的。请考虑[例 5.9]中带有延迟的顺序块语句,并且将其转换为一个并行块。转换后的 Verilog 代码见[例 5.10]。除了所有语句在仿真 0 时刻开始执行以外,仿真结果是完全相同的。这个并行块执行结束的时间第 20 个仿真时间单位,而不再是 35 个。

[例 5.10] 并行块。

```

//举例 1: 带延迟的并行块
reg x, y;
reg [1:0] z, w;

initial
fork
    x = 1'b0;           //在仿真时刻 0 完成
    #5 y = 1'b1;        //在仿真时刻 5 完成
    #10 z = {x, y};     //在仿真时刻 10 完成
    #20 w = {y, x};     //在仿真时刻 20 完成
join

```

并行块为我们提供了并行执行语句的机制。不过在使用并行块时需要注意,如果两条语句在同一时刻对同一个变量产生影响,那么将会引起隐含的竞争,这种情况是需要避免的。下面给出了[例 5.9]中说明 1 的并行块描述。在这段代码中故意引入了竞争。所有的语句在仿真 0 时刻开始执行,但是实际的执行顺序是未知的。在这个例子中,如果 $x = 1'b0$ 和 $y = 1'b1$ 两条语句首先执行,那么变量 z 和 w 的值为 1 和 2;如果这两条最后执行,那么 z 和 w 的值都是 $2'bxx$ 。因此执行这个块语句后 z 和 w 的值不确定,依赖于仿真器的具体实现方法。从仿真的角度来讲,并行块中的所有语句是一起执行的,但是实际上运行仿真程序的 CPU 在任一时刻只能执行一条语句,而且不同的仿真器按照不同的顺序执行。因此无法正确的处理竞争是目前所使用的仿真器的一个缺陷,这一缺陷并不是并行块所引起的。例如:

```
//故意引入竞争条件的并行块
```

```
reg x, y;
reg [1:0] z, w;
```

initial
fork
 x = 1'b0;
 y = 1'b1;
 z = {x, y};
 w = {y, x};
join

可以将并行块的关键字 fork 看成是将一个执行流分成多个独立的执行流;而关键字 join 则是将多个独立的执行流合并为一个执行流。每个独立的执行流之间是并发执行的。

5.6.2 块语句的特点

下面将讨论块语句所具有的 3 个特点:嵌套块、命名块和命名块的禁用。

1. 嵌套块

块可以嵌套使用,顺序块和并行块能够混合在一起使用,如[例 5.11]所示。

[例 5.11] 嵌套块。

```
//嵌套块
initial begin
    begin
        x = 1'b0;
        fork
            #5 y = 1'b1;
            #10 z = {x, y};
        join
        #20 w = {y, x};
    end
endmodule
```

2. 命名块

块可以具有自己的名字,这称为命名块。命名块的特点是:

- (1) 命名块中可以声明局部变量;
- (2) 命名块是设计层次的一部分,命名块中声明的变量可以通过层次名引用进行访问;
- (3) 命名块可以被禁用,例如停止其执行。

[例 5.12] 显示命名块和命名块的层次名引用。

```
//命名块
module top;
```

```

initial
begin : block1      //名字为 block1 的顺序命名块
    integer i;        //整型变量 i 是 block1 命名块的静态本地变量
                        //可以用层次名 top.block1.i 被其他模块访问
    ...
    ...
end

initial
fork : block2      //名字为 block2 的并行命名块
    reg i;           //寄存器变量 i 是 block2 命名块的静态本地变量
                        //可以用层次名 top.block2.i 被其他模块访问
    ...
    ...
join

```

3. 命名块的禁用

Verilog 通过关键字 disable 提供了一种中止命名块执行的方法。disable 可以用来从循环中退出、处理错误条件以及根据控制信号来控制某些代码段是否被执行。对块语句的禁用导致紧接在块后面的那条语句被执行。对于 C 程序员来说,这一点非常类似于使用 break 退出循环。两者的区别在于 break 只能退出当前所在的循环,而使用 disable 则可以禁用设计中任意一个命名块。

让我们来考虑[例 5.13]中的说明,这段代码的功能是在一个标志寄存器中查找第一个不为零的位。[例 5.13]中的 while 循环可以使用 disable 来进行改写,使得在找到不为零的位后马上退出 while 循环。

[例 5.13] 命名块的禁用。

```

//在(矢量)标志寄存器的各个位中从低有效位开始找寻第一个值为 1 的位
//从矢量标志寄存器的低有效位开始查找第一个值为 1 的位
reg [15:0] flag;
integer i; //用于计数的整数

```

```

initial
begin
    flag = 16'b 0010_0000_0000_0000;
    i = 0;
begin: block1      //while 循环声明中的主模块是命名块 block1
    while(i < 16)
begin
    if (flag[i])
begin
        $display("Encountered a TRUE bit at element number %d", i);
        disable block1; //在标志寄存器中找到了值为真(1)的位,禁用 block1
end
end

```

```
i = i + 1;
end
end
```

5.7 生成块

生成语句可以动态地生成 Verilog 代码。这一声明语句方便了参数化模块的生成。当对矢量中的多个位进行重复操作时,或者当进行多个模块的实例引用的重复操作时,或者在根据参数的定义来确定程序中是否应该包括某段 Verilog 代码的时候,使用生成语句能够大大简化程序的编写过程。

生成语句能够控制变量的声明、任务或函数的调用,还能对实例引用进行全面的控制。编写代码时必须在模块中说明生成的实例范围,关键字 generate – endgenerate 用来指定该范围。

生成实例可以是以下的一个或多种类型:

- (1) 模块;
- (2) 用户定义原语;
- (3) 门级原语;
- (4) 连续赋值语句;
- (5) initial 和 always 块。

生成的声明和生成的实例能够在设计中被有条件地调用(实例引用)。在设计中可以多次调用(实例引用)生成的实例和生成的变量声明。生成的实例具有唯一的标识名,因此可以用层次命名规则引用。为了支持结构化的元件与过程块语句的相互互联,Verilog 语言允许在生成范围内声明下列数据类型:

- (1) net (线网)、reg (寄存器);
- (2) integer (整型数)、real(实型数)、time(时间型)、realtime (实数时间型);
- (3) event (事件)。

生成的数据类型具有唯一的标识名,可以被层次引用。此外,究竟是使用按照次序或者参数名赋值的参数重新定义,还是使用 defparam 声明的参数重新定义,都可以在生成范围内定义。

注意:生成范围内定义的 defparam 语句所能够重新定义的参数必须是在同一个生成范围内,或者是在生成范围的层次化实例当中。

任务和函数的声明也允许出现在生成范围之中,但是不能出现在循环生成当中。生成任务和函数同样具有唯一的标识符名称,可以被层次引用。

不允许出现在生成范围之中的模块项声明包括:

- (1) 参数、局部参数;
- (2) 输入、输出和输入/输出声明;
- (3) 指定块。

生成模块实例的连接方法与常规模块实例相同。

在 Verilog 中有 3 种创建生成语句的方法, 它们是:

- (1) 循环生成;
- (2) 条件生成;
- (3) case 生成。

在 5.7.1~5.7.2 节中, 将对这 3 种方法进行详细说明。

5.7.1 循环生成语句

循环生成语句允许使用者对下面的模块或模块项进行多次实例引用:

- (1) 变量声明;
- (2) 模块;
- (3) 用户定义原语、门级原语;
- (4) 连续赋值语句;
- (5) initial 和 always 块。

[例 5.14]说明了如何使用生成语句对两个 N 位的总线用门级原语进行按位异或。在这里其目的在于说明循环生成语句的使用方法, 其实这个例子如果使用矢量线网的逻辑表达式比用门级原语实现起来更为简单。

[例 5.14] 对两个 N 位总线变量进行按位异或。

```
//本模块生成两条 N 位总线变量的按位异或
module bitwise_xor ( out , i0 , i1 ) ;
  //参数声明语句, 参数可以重新定义
  parameter N = 32 ; //默认的总线位宽为 32 位
  //端口声明语句
  output [ N-1 : 0 ] out ;
  input [ N-1 : 0 ] i0 , i1 ;
  //声明一个临时循环变量
  //该变量只用于生成块的循环计算
  //Verilog 仿真时该变量在设计中并不存在
  genvar j ;
  //用一个单循环生成按位异或的异或门(xor)
  generate
    for ( j = 0 ; j < N ; j = j + 1 )
      begin : xor_loop
        xor g1 (out [j] , i0 [j] , i1 [j]) ;
      end      //在生成块内部结束循环
  endgenerate //结束生成块
  //另外一种编写形式
  //异或门可以用 always 块来替代
```

```

// reg [ N-1 : 0 ] out ;
// generate
//   for ( j = 0 ; j < N ; j = j + 1 )
//     begin : bit
//       always @ ( i0 [ j ] or i1 [ j ] ) out [ j ] = i0 [ j ] ^ i1 [ j ] ;
//     end
//   endgenerate
endmodule

```

从[例 5.14]中可以观察到下面几个有趣的现象：

- (1) 在仿真开始之前,仿真器会对生成块中的代码进行确立(展平),将生成块转换为展开的代码,然后对展开的代码进行仿真。因此,生成块的本质是使用循环内的一条语句来代替多条重复的 Verilog 语句,简化用户的编程。
- (2) 关键词 genvar 用于声明生成变量,生成变量只能用在生成块之中;在确立后的仿真代码中,生成变量是不存在的。
- (3) 一个生成变量的值只能由循环生成语句来改变。
- (4) 循环生成语句可以嵌套使用,不过使用同一个生成变量作为索引的循环生成语句不能够相互嵌套。
- (5) xor_loop 是赋予循环生成语句的名字,目的在于通过它对循环生成语句之中的变量进行层次化引用。因此,循环生成语句中各个异或门的相对层次名为: xor_loop[0].g1, xor_loop[1].g1, ……, xor_loop[31].g1。

循环生成语句的使用是相当灵活的。各种 Verilog 语法结构都可以用在循环生成语句之中。对于读者来说,重要的是能够想像出循环生成语句被展平之后的形式,这对于理解循环生成语句的作用是很有必要的。[例 5.15]给出了使用生成语句描述的脉动加法器,并且在循环生成语句中声明了线网变量。

[例 5.15] 用循环生成语句描述的脉动加法器。

```

//本模块生成一个门级脉动加法器
module ripple_adder ( co , sum , a0 , a1 , ci ) ;
//参数声明语句,参数可以重新定义
parameter N = 4 ; //默认的总线位宽为 4
//端口声明语句
output [ N-1 : 0 ] sum ;
output co ;
input [N-1 : 0 ] a0 , a1 ;
input ci ;
//本地线网声明语句
wire [N-1 : 0 ] carry ;

```

```

//指定进位变量的第 0 位等于进位的输入
assign carry [0] = ci ;

//声明临时循环变量,该变量只用于生成块的计算
//由于在仿真前,循环生成已经展平,所以用 Verilog 对设计进行仿真时,该变量已经不再存在
genvar i ;

//用一个单循环生成按位异或门等逻辑
generate for (i = 0 ; i < N ; i = i + 1) begin : r_loop
  wire t1 , t2 , t3 ;
  xor g1 (t1 , a0 [i] , a1 [i]) ;
  xor g2 (sum [i] , t1 , carry [i]) ;
  and g3 (t2 , a0 [i] , a1 [i]) ;
  and g4 (t3 , t1 , carry [i]) ;
  or g5 (carry [i + 1] , t2 , t3) ;
end //生成块内部循环的结束
endgenerate //生成块的结束

//根据上面的循环生成,Verilog 编译器会自动生成以下相对层次实例名

// xor : r_loop[0].g1 , r_loop[1].g1 , r_loop[2].g1 , r_loop[3].g1,
//       r_loop[0].g2 , r_loop[1].g2 , r_loop[2].g2 , r_loop[3].g2;
// and : r_loop[0].g3 , r_loop[1].g3 , r_loop[2].g3 , r_loop[3].g3,
//       r_loop[0].g4 , r_loop[1].g4 , r_loop[2].g4 , r_loop[3].g4;
// or  : r_loop[0].g5 , r_loop[1].g5 , r_loop[2].g5 , r_loop[3].g5,

//根据上面生成的实例用下面这些生成的线网连接起来
// Nets : r_loop[0].t1 , r_loop[0].t2 , r_loop[0].t3,
//        r_loop[1].t1 , r_loop[1].t2 , r_loop[1].t3,
//        r_loop[2].t1 , r_loop[2].t2 , r_loop[2].t3,
//        r_loop[3].t1 , r_loop[3].t2 , r_loop[3].t3;

assign co = carry [N] ;
endmodule

```

5.7.2 条件生成语句

条件生成语句类似于 if_else_if 的生成构造,该结构可以在设计模块中根据经过仔细推敲并确定表达式,有条件地调用(实例引用)以下这些 Verilog 结构:

- (1) 模块;
- (2) 用户定义原语、门级原语;
- (3) 连续赋值语句;

(4) initial 或 always 块。

[例 5.16]说明如何用条件生成语句实现参数化乘法器。如果参数 a0_width 或 a1_width 小于 8(生成实例的条件),则调用(实例引用)超前进位乘法器;否则调用(实例引用)树形乘法器。

[例 5.16] 使用条件生成语句实现参数化乘法器。

```
//本模块实现一个参数化乘法器
module multiplier ( product , a0 , a1 ) ;
  //参数声明,该参数可以重新定义
  parameter a0_width = 8 ;
  parameter a1_width = 8 ;

  //本地参数声明
  //本地参数不能用参数重新定义(defparam)修改
  //也不能在实例引用时通过传递参数语句,即 #(参数 1,参数 2,...)的方法修改
  localparam product_width = a0_width + a1_width ;

  //端口声明语句
  output [ product_width - 1 : 0 ] product ;
  input [ a0_width - 1 : 0 ] a0 ;
  input [ a1_width - 1 : 0 ] a1 ;

  //有条件地调用(实例引用)不同类型的乘法器
  //根据参数 a0_width 和 a1_width 的值,在调用时引用相对应的乘法器实例
  generate
    if ( a0_width < 8 ) || ( a1_width < 8 )
      cal_multiplier # ( a0_width , a1_width ) m0 ( product , a0 , a1 );
    else
      tree_multiplier # ( a0_width , a1_width ) m0 ( product , a0 , a1 );
  endgenerate //生成块的结束

endmodule
```

5.7.3 case 生成语句

case 生成语句可以在设计模块中,根据仔细推敲确定多选一 case 构造,有条件地调用(实例引用)下面这些 Verilog 结构:

- (1) 模块;
- (2) 用户定义原语、门级原语;
- (3) 连续赋值语句;
- (4) initial 或 always 块。

[例 5.17]说明如何使用 case 生成语句实现 N 位加法器。

[例 5.17] case 生成语句举例。