

SystemVerilog 与功能验证

钟文枫 编著



本书重点介绍硬件设计描述和验证语言 SystemVerilog 的基本语法及其在功能验证上的应用；书中以功能验证为主线，讲述基本的验证流程、高级验证技术和验证方法学，以 SystemVerilog 为基础结合石头、剪刀、布的应用实例，重点阐述了如何采用 SystemVerilog 实现随机激励生成、功能覆盖率驱动验证、断言验证等多种高级验证技术；最后，通过业界流行的开放式验证方法学 OVM 介绍如何在验证平台中实现可重用性。

封底无防伪标均为盗版

版权所有，侵权必究

本书法律顾问 北京市展达律师事务所

图书在版编目（CIP）数据

SystemVerilog 与功能验证/钟文枫编著. —北京：机械工业出版社，2010. 8

ISBN 978-7-111-31373-1

I . S… II . 钟… III . 硬件描述语言，SystemVerilog – 程序设计 IV . TP312

中国版本图书馆 CIP 数据核字（2010）第 140823 号

机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码 100037）

责任编辑：张少波

印刷

2010 年 10 月第 1 版第 1 次印刷

185mm × 260mm · 14 印张

标准书号：ISBN 978-7-111-31373-1

定价：33.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991；88361066

购书热线：(010) 68326294；88379649；68995259

投稿热线：(010) 88379604

读者信箱：hzjsj@hzbook.com

SystemVerilog | 前言

在读者深入阅读本书之前，我先对本书的主要结构和内容做个简要的介绍，以便不同背景的读者能够有选择地阅读，快速获取自己需要的知识。

本书的结构和内容

本书分为 11 章，系统论述了基于 SystemVerilog 的功能验证方法，重点关注以下三个方面的内容。

- 功能验证在整个 FPGA/ASIC 设计流程中的作用及主要的验证技术和方法学。
- SystemVerilog 的语法结构和在功能验证上的应用及基本原理。
- 如何采用 SystemVerilog 搭建验证平台。

第 1 章从 FPGA/ASIC 整个设计流程的角度介绍功能验证的地位和作用、验证的基本流程、验证的主要技术和方法学，最后引入硬件验证语言 SystemVerilog。

第 2 章介绍 SystemVerilog 相对于 Verilog 增加的数据类型、SystemVerilog 的编程结构、流程控制和方法（任务/函数）等基本语法。

第 3 章介绍 fork…join 结构、并行进程之间的通信方法：mailbox/semaphore/event 同步和互斥机制。

第 4 章介绍面向对象编程语言、类的基本概念、对象的创建、赋值与复制；如何采用类来封装事务处理器等验证组件和搭建验证平台。

第 5 章介绍虚接口，以及如何通过虚接口实现类的对象和设计模块的连接，实现事务处理器的可重用。

第 6 章介绍随机激励生成，其中重点讨论基于对象的约束随机激励产生机制、约束定义以及动态控制激励随机生成。

第 7 章介绍面向对象编程语言中的继承和多态。

第 8 章介绍覆盖率在验证流程中的作用、SystemVerilog 功能覆盖率的语法，包括覆盖组、覆盖点和交叉覆盖点，以及如何实现一个覆盖率驱动验证平台。

第 9 章介绍断言在验证流程中的作用、断言的采用策略、SystemVerilog 断言子集的语法结构以及如何通过 bind 结构实现断言与设计分离。

第 10 章介绍验证重用以及 OVM 验证方法学的核心技术：基于 Factory 的对象生成机制、动态参数配置、激励生成与验证架构分离以及测试用例在验证架构的顶层。

第 11 章介绍 SystemVerilog 和 C 语言的接口：DPI，重点介绍如何在 SystemVerilog 层面定义输入方法接口（SystemVerilog 调用外部 C 程序）和输出方法接口（SystemVerilog 程序输出供外部 C 调用），最后介绍 DPI 在验证中的作用。

从第 4~9 章，我们以石头、剪刀、布的仲裁器作为实例，并将各章中涉及的 SystemVerilog 重要语法和验证技术应用到验证平台搭建中，附有大量源代码供读者参考和练习。

如何阅读本书

本书的目标读者是 FPGA/ASIC 设计工程师和验证工程师、相关专业的在校本科生、研究生和老师。若具有一定的硬件描述语言（Verilog 或者 VHDL）和面向对象编程语言（如 C++）的基础，将有助于对本书的阅读。

想了解验证技术概况和验证方法学的读者，可以直接阅读第 1 章和第 10 章；对于普通读者，建议从第 1 章阅读到第 6 章；再根据自己的学习和工作需要，学习第 7 章以后的内容，这部分内容适合验证工程师和有一定验证经验的读者。有 Verilog 基础的读者，可以粗略浏览一下第 2 章的新增数据类型，SystemVerilog 其他语法结构与 Verilog 基本类似；没有 Verilog 基础的读者，这一章需要认真阅读；想了解覆盖率驱动验证、断言或者 DPI 的读者可以直接阅读对应内容的第 8 章、第 9 章和第 11 章。

学习一门新语言的第一步是学会读懂代码，本书为每个例子提供了详尽的解释，并且每个例子都可以在仿真平台上运行。第二步是能够将学到的语法应用到自己的项目中，编写自己的程序。第三步是学会调试代码，调试代码是最艰苦和最具挑战性的工作。和大多数编程语言一样，SystemVerilog 中基于面向对象编程结构的执行一般是不消耗物理时间的，是动态生成和析构的，为此在关键部位嵌入调试代码（如 \$display）将是最有效的方法。

EDA 联盟网站 www.edaunion.com 是专业的技术论坛，为广大的工程师和读者提供了一个技术讨论的平台；各位读者可以在验证板块发帖讨论和交流自己学习过程中遇到的问题，分享自己的学习体会。本书的所有例子也可以从论坛中下载，若发现任何错误请告知我们。

希望本书能够为对 SystemVerilog 和功能验证技术感兴趣的读者提供一个入门的指南。

致谢

本书在编写过程中，充分利用和吸收了业界最前沿的技术和信息，在此对书中参考引用文献和书籍的作者表示衷心的感谢。由于编写时间仓促，本人学识有限，若书中存在纰漏请各位读者给予谅解并指正，我将虚心听取并及时更正，并欢迎进行技术切磋和探讨（联系方式：edaunion.book@gmail.com）。

本书的完成有赖于朋友们的支持，其中赵立、萧路和王国平三位资深工程师审阅了全稿，并给出了很多有建设性的意见。对他们辛勤的工作，致以最高的敬意和最真挚的感谢。

在此，我还要特别感谢重庆邮电大学的邓亚平教授、吴慧莲教授、鲜继清教授和张宗琪教授对我在学业上的帮助和支持；感谢郑建宏教授提供了 TD – SCDMA 终端芯片项目的研发平台和锻炼机会；感谢入行以来帮助过我的朋友俞洋、钟信潮、王诚、薛小刚、庄永军和傅骏诚；感谢华为海思的宾兵，让我有幸加入了一个伟大的集体，参与了 GPON 项目；感谢明导电子乐于分享经验和技术的同事。

最后，感谢我敬爱的父母！

钟文枫
2010 年 7 月 15 日于上海

SystemVerilog | 目 录

前言

第1章 功能验证技术与方法学概要	1
1.1 功能验证与验证平台	1
1.1.1 专用芯片设计流程	1
1.1.2 什么是验证	2
1.1.3 验证平台可以做些什么	3
1.1.4 功能验证流程	5
1.2 验证技术和验证方法学	8
1.2.1 黑盒、白盒与灰盒验证	8
1.2.2 验证技术	9
1.2.3 验证存在的挑战	13
1.2.4 验证方法学	13
1.2.5 断言验证	15
1.2.6 覆盖率驱动验证	16
1.3 硬件验证语言	21
1.3.1 Open Vera	21
1.3.2 e 语言	21
1.3.3 PSL	22
1.3.4 SystemC	22
1.3.5 SystemVerilog	22
第2章 数据类型与编程结构	24
2.1 数据类型	24
2.1.1 两态数据类型	25
2.1.2 枚举类型和用户自定义类型	26
2.1.3 数组与队列	28
2.1.4 字符串	36
2.1.5 结构体和联合体	37
2.1.6 常量	39

2.1.7 文本表示	41
2.1.8 操作符和表达式	43
2.2 过程语句	45
2.2.1 赋值语句	45
2.2.2 控制结构	46
2.3 函数和任务	52
2.3.1 函数和任务的区别	52
2.3.2 子程序定义	53
2.3.3 子程序参数	53
2.3.4 子程序返回	56
2.3.5 自动存储	56
2.4 编程结构	57
2.4.1 模块	57
2.4.2 接口	59
2.4.3 过程块和语句块	60
2.4.4 数据对象	62
2.4.5 程序块	62
2.4.6 简单的验证架构	63
2.5 数据的生命周期和作用域	64
2.6 数据类型转换	65
2.6.1 静态类型转换	66
2.6.2 动态类型转换	66
第3章 并发进程与进程同步	68
3.1 fork...join	68
3.1.1 三种并发方式	69
3.1.2 进程与变量	72
3.1.3 进程控制	73
3.2 mailbox	74
3.2.1 mailbox 的基本操作	75
3.2.2 参数化 mailbox	77
3.2.3 mailbox 应用实例	77
3.3 semaphore	78
3.3.1 semaphore 的基本操作	79
3.3.2 semaphore 应用实例	80

3.4 event	81
3.4.1 事件触发	81
3.4.2 等待事件	81
3.4.3 事件的触发属性	81
第4章 面向对象编程入门	83
4.1 过程编程语言与面向对象编程语言	83
4.2 类	84
4.2.1 类的基本概念	85
4.2.2 构造函数	87
4.2.3 静态属性与方法	89
4.2.4 this 操作符	91
4.2.5 对象的赋值与复制	91
4.2.6 块外声明	94
4.3 石头、剪刀、布仲裁器实例（基于类的验证平台）	95
4.3.1 验证环境顶层	96
4.3.2 验证组件	99
第5章 虚接口	107
5.1 虚接口的基本概念及应用	107
5.1.1 虚接口的基本概念	107
5.1.2 虚接口的应用	109
5.2 端口模式和时钟控制块	113
5.2.1 端口模式	113
5.2.2 时钟控制块	114
第6章 随机测试	118
6.1 激励产生	118
6.1.1 什么是随机	119
6.1.2 潜在问题	119
6.2 随机生成机制	120
6.2.1 随机系统函数	120
6.2.2 randcase/randsequence	121
6.3 基于对象的随机生成	122
6.3.1 随机变量	123
6.3.2 约束定义	124

6.3.3 随机方法	130
6.3.4 随机使能控制	131
6.3.5 约束的动态修改	134
6.4 标准随机函数	134
6.5 随机激励的应用	135
第7章 继承与多态	137
7.1 继承和多态的基本概念	137
7.2 继承与子类	137
7.2.1 类的继承与重写	138
7.2.2 子类对象与父类对象的赋值	141
7.2.3 构造函数调用	142
7.3 虚方法与多态	144
7.3.1 虚方法	145
7.3.2 多态	148
7.4 虚类和参数化类	148
7.4.1 虚类	148
7.4.2 参数化类	149
7.5 约束重写	150
7.6 数据的隐藏与封装	151
第8章 功能覆盖率	153
8.1 覆盖率	153
8.1.1 目标覆盖率	153
8.1.2 代码覆盖率	154
8.1.3 功能覆盖率	154
8.2 SystemVerilog 的功能覆盖率	155
8.2.1 覆盖组 (covergroup)	155
8.2.2 覆盖点 (coverpoint)	157
8.2.3 交叉覆盖点 (cross)	159
8.3 覆盖率驱动的验证平台	162
第9章 断言	167
9.1 断言的概念及作用	167

9.2 SVA	169
9.2.1 SVA 的语法层次结构	170
9.2.2 SVA 应用实例	173
9.2.3 bind	175
第 10 章 验证重用与验证方法学	178
10.1 验证重用中存在的问题	178
10.2 验证方法学 OVM	179
10.3 OVM 的四大核心技术	180
10.3.1 基于 Factory 的验证平台动态构建	181
10.3.2 动态的配置机制	183
10.3.3 测试用例在验证架构的顶层	184
10.3.4 激励产生与验证架构分离	185
第 11 章 SystemVerilog 与 C 语言的接口	187
11.1 什么是 DPI	187
11.2 DPI 的应用	188
11.2.1 方法的导入	188
11.2.2 方法的导出	190
11.2.3 DPI 的数据类型映射	191
11.2.4 DPI 的具体应用	192
附录 A 覆盖率内置参数和方法列表	193
附录 B 断言重复操作符和序列操作符列表	195
附录 C QuestaSim 简要介绍	198
附录 D 常用术语中英文对照	205
参考文献	207
后记	208

源代码索引 | SystemVerilog

源代码 2-1 枚举类型实例: enum_example.sv	26
源代码 2-2 用户自定义类型实例: typedef_example.sv	28
源代码 2-3 多维数组和压缩数组实例: array_example.sv	30
源代码 2-4 动态数组实例: dy_array_example.sv	31
源代码 2-5 关联数组实例: as_array_example.sv	33
源代码 2-6 队列实例: queue_example.sv	35
源代码 2-7 结构体实例: struct_example.sv	38
源代码 2-8 类型参数化实例: para_type_example.sv	40
源代码 2-9 赋值语句实例: assign_example.sv	46
源代码 2-10 if 条件选择语句实例: if_example.sv	47
源代码 2-11 case 条件选择语句实例: case_example.sv	47
源代码 2-12 for 循环语句实例: for_example.sv	48
源代码 2-13 while 循环语句实例: while_example.sv	49
源代码 2-14 do...while 循环语句实例: dowhile_example.sv	49
源代码 2-15 repeat 循环语句实例: repeat_example.sv	50
源代码 2-16 forever 循环语句实例: forever_example.sv	51
源代码 2-17 foreach 循环语句实例: foreach_example.sv	51
源代码 2-18 ref 引用端口类型实例: ref_example.sv	54
源代码 2-19 模块实例: module_example.sv	58
源代码 2-20 接口实例: interface_example.sv	59
源代码 2-21 静态变量实例: static_auto_example.sv	65
源代码 3-1 fork...join 语句实例: fork_example.sv	70
源代码 3-2 fork...join_any 语句实例: fork_any_example.sv	71
源代码 3-3 fork...join_ none 语句实例: fork_none_example.sv	71
源代码 3-4 disable 语句实例: disable_example.sv	73
源代码 3-5 mailbox 应用实例: mailbox_example.sv	77
源代码 3-6 semaphore 应用实例: semaphore_example.sv	80
源代码 3-7 event 应用实例: event_example.sv	82
源代码 4-1 简单以太包的类实例: ether_packet.sv	84

源代码 4-2	类的构造函数实例：new_construct_example.sv	88
源代码 4-3	类的静态变量实例：static_var_class.sv	89
源代码 4-4	类的静态方法实例：static_method_class.sv	90
源代码 4-5	this 操作符实例：this_class_example.sv	91
源代码 4-6	类的浅复制实例：shallow_copy.sv	92
源代码 4-7	类的自定义深复制实例：deep_copy.sv	93
源代码 4-8	仲裁器设计（石头、剪刀、布）：rps_dut.sv	95
源代码 4-9	验证环境顶层（石头、剪刀、布）：top_class_based.sv	96
源代码 4-10	时钟复位模块（石头、剪刀、布）：rps_clock_reset.sv	97
源代码 4-11	验证环境库文件（石头、剪刀、布）——激励单元片段：rps_env_pkg.sv	98
源代码 4-12	验证环境库文件（石头、剪刀、布）——激励生成器片段：rps_env_pkg.sv	99
源代码 4-13	事务驱动器（石头、剪刀、布）：rps_driver.sv	100
源代码 4-14	监控器（石头、剪刀、布）：rps_monitor.sv	101
源代码 4-15	基于类的验证环境（石头、剪刀、布）：rps_env.sv	102
源代码 4-16	验证环境库文件（石头、剪刀、布）——记分板片段：rps_env_pkg.sv	103
源代码 5-1	虚接口例子：virtual_interface_example.sv	108
源代码 5-2	定义虚接口（石头、剪刀、布）：interface.sv	109
源代码 5-3	基于虚接口的事务驱动器（石头、剪刀、布）：rps_driver.sv	110
源代码 5-4	基于虚接口的监控器（石头、剪刀、布）：rps_monitor.sv	111
源代码 5-5	基于虚接口的验证环境（石头、剪刀、布）：rps_env.sv	111
源代码 5-6	基于虚接口的验证顶层（石头、剪刀、布）：rps_tb_top.sv	112
源代码 5-7	端口模式实例：interface_mode.sv	113
源代码 6-1	randcase 实例：randcase_example.sv	121
源代码 6-2	randsequence 实例：randsequence_example.sv	122
源代码 6-3	基于类的随机变量实例：class_random_example.sv	123
源代码 6-4	随机约束块实例：constraint_example.sv	125
源代码 6-5	inside 操作实例：inside_example.sv	125
源代码 6-6	dist 操作实例：dist_example.sv	126
源代码 6-7	foreach 操作实例：foreach_random_example.sv	128
源代码 6-8	solve...before 操作实例：solve_before_example.sv	129
源代码 6-9	随机变量使能模式实例：rand_mode_example.sv	132
源代码 6-10	随机约束使能模式实例：constraint_mode_example.sv	133
源代码 6-11	标准随机函数实例：std_randomize_example.sv	135
源代码 6-12	基于类的随机激励实例（石头、剪刀、布）：rps_env_pkg.sv	135
源代码 7-1	类的继承实例：class_extend_example.sv	139
源代码 7-2	类的重写实例 1：class_override_example.sv	139

源代码 7-3	基类与派生类实例：base_derived_example. sv	141
源代码 7-4	super 操作实例：super_example. sv	142
源代码 7-5	构造函数链实例：new_chain_example. sv	143
源代码 7-6	类的重写实例 2：base_override_example. sv	144
源代码 7-7	虚方法与多态实例：virtual_poly_example. sv	145
源代码 7-8	参数化类实例：parameterized_class. sv	149
源代码 7-9	约束块重写实例：constraint_override_example. sv	150
源代码 8-1	功能覆盖组实例：covergroup_example. sv	156
源代码 8-2	功能覆盖点实例：coverpoint_example. sv	156
源代码 8-3	功能分组柜实例：bin_example. sv	158
源代码 8-4	功能交叉覆盖点实例：cross_example. sv	159
源代码 8-5	功能覆盖模块（石头、剪刀、布）：rps_coverage. sv	162
源代码 8-6	基于功能覆盖率验证环境（石头、剪刀、布）：rps_env. sv	165
源代码 9-1	断言序列实例：sequence_example. sv	171
源代码 9-2	断言属性实例：property_example. sv	172
源代码 9-3	断言模块（石头、剪刀、布）：rps_sva. sv	173
源代码 9-4	基于断言的验证顶层（石头、剪刀、布）：rps_tb_top. sv	174
源代码 9-5	bind 操作实例：bind_example. sv	175
源代码 11-1	外部 Hello 程序 1（基于 C）：example 1/hello_c. c	190
源代码 11-2	DPI 导入方法实例：example 1/hello. v	190
源代码 11-3	外部 Hello 程序 2（基于 C）：example 2/hello_c. c	191
源代码 11-4	DPI 导出方法实例：example 2/hello. v	191

功能验证技术与方法学概要

摩尔定律指出，集成芯片可容纳的晶体管数目约每隔 18 个月便会增加一倍，性能也将提升一倍。随着半导体制造工艺的改进，大规模 SoC（System on Chip，片上系统）和多核设计的出现，专用集成芯片设计的复杂度以指数形式增长，这使得验证工作成为芯片设计流程中的瓶颈，数据表明，接近 70% ~ 80% 的设计时间花费在功能验证中。目前，专用集成芯片已经可以达到上亿门，设计上复杂度的提高迫切需要在功能验证方面有新的技术和方法学。

本章将从芯片设计流程入手，讨论功能验证在整个流程中的位置及其所涵盖的内容；介绍业界目前流行的各种验证技术和验证方法学，以及这些技术的优点和应用场景；最后，简要介绍业界常用验证语言 Vera、E、PSL、SystemC 和 SystemVerilog 的来历和特点。

1.1 功能验证与验证平台

在芯片设计过程中，验证是一个覆盖面比较广的课题，其中主要包括功能验证、物理验证、时序验证等内容。本书讨论的重点是功能验证，针对芯片设计对象的行为功能进行验证，以保证设计能够按照设计规范实现其应有的功能。在本章，我们会介绍功能验证中存在的多种验证技术，其中通过验证语言、传统的硬件描述语言或者 C 等其他语言，搭建验证平台（testbench，或译为测试平台）是业界最为重要的验证手段。

验证平台（testbench）需要提供更多的自动化机制来提高每一个测试用例（test case）的功能覆盖率和减少创建测试用例的时间。然而，我们也要折中考虑到验证平台的复杂度和投入在开发验证平台的时间。一方面，验证平台的开发不应该成为专用集成芯片开发的“制约因素”；另一方面，一定程度提高验证平台复杂性可以减少创建多个测试用例所消耗的时间，为此这是一个很难的抉择。

1.1.1 专用芯片设计流程

图 1-1 举例说明了一个 ASIC（专用集成芯片）从制定设计规范到投片的设计流程。

目前，整个流程中关键的部分在于功能验证。在图 1-1 中功能验证只作为一个独立的模块，但它是一个相当复杂的过程——包括定义测试用例，创建测试环境，运行测试用例，保证所有要求的用例被覆盖到。

验证活动在设计规范完成后就可以开始，而且持续到版图完成，在很多情况下，验证可能超出了版图完成的阶段。下一节，我们将进一步详细介绍功能验证和验证流程。

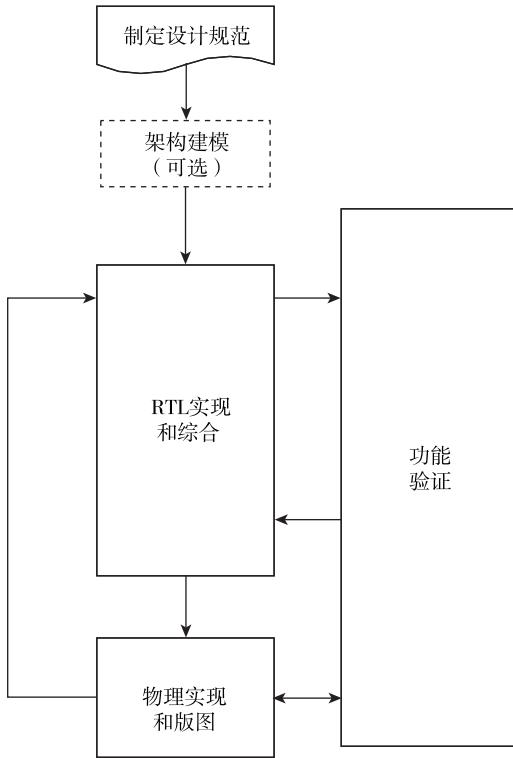


图 1-1 基本的 ASIC 设计流程

1.1.2 什么是验证

验证是确保设计和预定的设计期望一致（吻合）的过程。设计期望通常通过一个或者多个设计规范来定义的。对于专用集成芯片设计，在不同的阶段存在如下形式的验证。

- 1) 寄存器传输级（Register Transfer Level, RTL）的功能验证。
- 2) 门级的仿真，为了验证综合后网表和期望的功能是否一致。
- 3) 形式验证（等价性检查）来确保门级网表和 RTL 代码的一致性。
- 4) 时序验证，为了验证设计能否在特定的频率上运行，通常采用静态验证工具。

在本书下面，我们提到的验证（verification）特指寄存器传输级（RTL）的功能验证。其他形式的验证都不在讨论范围内。

功能验证在专用集成芯片设计流程中关注设计的行为，几乎所有的功能都可以在 RTL 层次被验证。

图 1-2 展示了一个高层次的功能验证视图。一个专用集成芯片设计可以被看成拥有一系列输入和输出的集合，设计的输出是基于设计的输入和当前的状态。在功能验证的过程中，工程师在被测设计（Design Under Test, DUT）外搭建验证平台。验证平台被用来应用一个或者多个测试激励，并将激励发送到设计的输入中。激励可以通过验证平台产生，或者通过手动创建。最后，输出将进行比较，看结果是否正确。结果检查可以通过验证平台、脚本或者手工来实现。

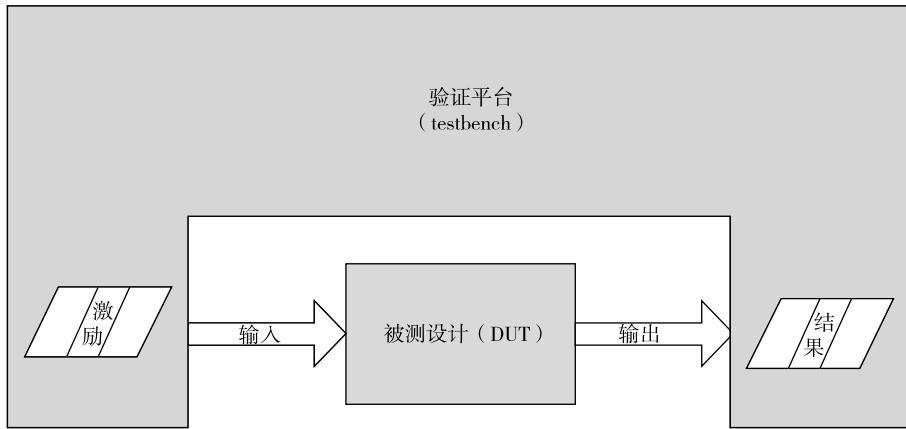


图 1-2 验证平台

1.1.3 验证平台可以做些什么

验证平台的主要功能如下。

- 1) 产生激励。
- 2) 把激励应用到被测设计中。
- 3) 检查结果和验证测试是否通过，也就是确保被测设计的输出和期望一致。

1. 激励产生的途径

如何区分二进制激励和用户激励很重要。二进制激励是驱动到被测设计的引脚中的 1 和 0 序列，如图 1-3 所示；用户激励是一个来自用户的指令（对信号赋值驱动或者通过对函数或者任务的调用）让验证平台可以运行某些操作。在传统的验证平台中，二进制激励和用户激励可以是一致的。

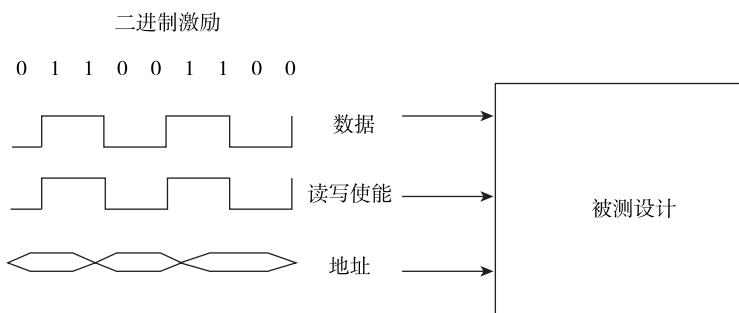


图 1-3 手工创建的二进制激励

产生用户激励的途径有很多种，可以是直接的，也可以是随机的。如果激励是从用户提供的确定输入，这就是所谓的直接测试（direct test）。如果激励是根据种子随机生成，这就是所谓的随机测试（random test）。

在很多传统的验证平台中，激励的过程就是把二进制序列，按照一定时间顺序送入到被测设计中。在高级的验证平台中，激励在更高的层次被用户抽象建模并封装。下面我们采用一个存储器的读写来解释直接测试和事务级激励产生，如图 1-4 所示。传统手工创建

直接测试，是通过驱动每个信号的数值来实现；而通过对激励和验证架构进行分层和高层抽象建模，也就是所谓的事务级的验证（transaction based verification），如本例中的 rd_mem，wr_mem 这种封装好的任务后，激励生成器可以将其生成的高层激励（可以是一组数据的集合）传递给它，这样可以大大提高激励产生的效率。

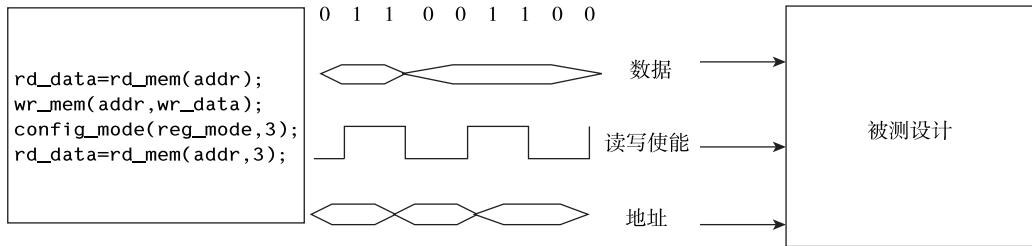


图 1-4 通过 Verilog 的任务（task）生成激励

在一个更加高级的验证平台中，激励可以根据用户指定的约束，完全通过验证平台自动产生，这就所谓的约束随机激励产生（constrained random stimulus generation），如图 1-5 所示。例如，用户要把包的大小约束在 64 比特和 1522 比特之间，并且在两者之间做一个权重配置方案。验证平台将可以自动生成和将随机包注入被测设计中。

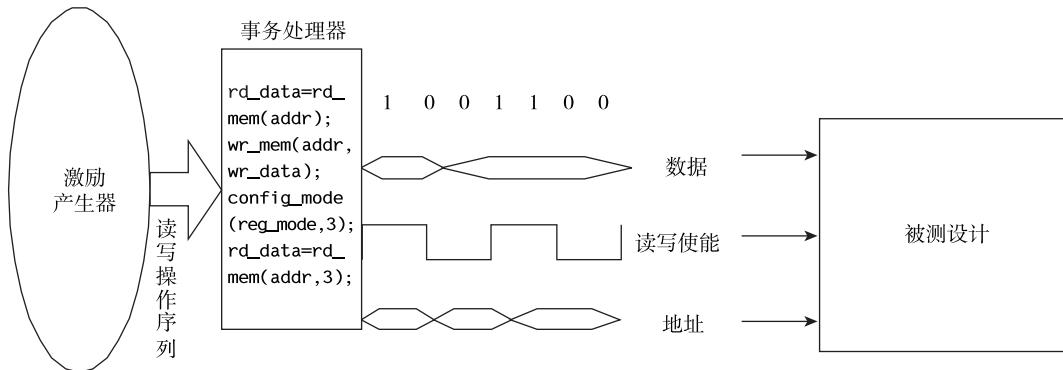


图 1-5 复杂的验证平台采用激励产生器生成激励

2. 应用二进制激励

最终，验证平台将把一个或者多个激励序列注入被测设计中。验证平台通过高层次的操作应用高层次的用户激励，二进制激励由验证平台生成并且通过对应的接口发送到被测设计中。在验证平台中，实现这种功能的验证组件（Verification Component，VC）被称为事务处理器（transactor）或者总线功能模型（Bus Function Model，BFM）。在本书中，我们采用事务处理器作为表述。

3. 结果检查

生成和应用二进制激励到被测设计中是验证平台的主要任务。此外，验证的目的在于检查被测设计的输出和期望是否一致，也就是结果检查。检查结果的办法和途径有以下三种。

1) 通过视图（波形窗口）检查。可视化（波形窗口）检查只适用于简单的设计。不需要额外投入，但是很可能出现人为的错误，而且不精确。如果这个设计需要回归测试，那么一定程度的自动化结果检查是必需的。

2) 通过自动化的后处理比较：记录被测设计的输出，通过运行脚本来比对结果。结果的后处理比较是常用的一种手段，因为它不需要添加太多的代码，而且不会影响仿真的性能（除非有过多的打印信息，因为对文件的 I/O 操作可能导致仿真速度的下降）。在复杂的设计中，必须小心的考虑打印信息。如果在每个时钟周期把所有输出都记录下来，日志文件将十分巨大。当然，需要考虑是否能够通过高层抽象来记录信息，以减少日志文件的大小。

后处理的另外一个好处是，由 HDL 开发的验证平台可以通过其他的语言来实现后处理比较，例如 perl、C/C++。但是，主要的问题是后处理只能在整个测试完成后来实现，为此会浪费很多仿真周期。另外，因为有些必要的状态信息很难被保存下来，所以调试也是很困难的事情。

3) 做一个实时的监察器可以及时自动检查。实时的监控器需要投入额外的开发，然而它们相当有用，特别是对于调试，一旦发生比对错误，它们可以通过设置错误标志或者打印信息提醒用户；在任何时刻，仿真器和验证平台的状态对用户完全可见，而且在发生错误的时刻，通过停止仿真可以缩短调试周期。

很多复杂设计，验证平台使用后处理和实时两种处理方法，这取决于设计的复杂度和接口的数目以及验证平台的应用层次。

1.1.4 功能验证流程

图 1-6 显示了功能验证流程。这个验证过程可以被分解成三个主要阶段：制定验证策略和验证计划；创建验证平台，运行和调试；覆盖率分析和回归测试。

虽然图 1-6 展示了特定顺序的活动，但是在制定验证策略和验证计划阶段，这个过程是交互的，可以在不同的阶段完成。

1. 制定验证策略和验证计划阶段

制定验证策略和验证计划阶段主要处理以下三个问题。

(1) 主要功能点和测试用例

当前，复杂设计的测试空间是非常大的。一个典型的百万门的设计可能有百万乃至上亿个需要测试的功能点，将测试空间缩小到一个可以管理的范围，或者说是一个有实际意义的集合并且没有损害其期望的功能，这是第一步。针对这些功能点，我们将根据具体情况拟定验证策略和测试用例，最后具体化到一个详细的、可执行的验证计划中，作为整个验证工作的指导。

(2) 验证平台的抽象层次

验证平台的抽象层次将决定它主要的处理对象：比特、包或者更高层次的数据类型。高层次的抽象建模可以让验证平台中低层次的功能自动化。

例如，一个以太网的验证平台运行在比特的抽象层次中，则要求验证工程师手工创建一个一个的数据包，从而组成一个测试用例。然而，若相同的验证平台在更高的层次上建

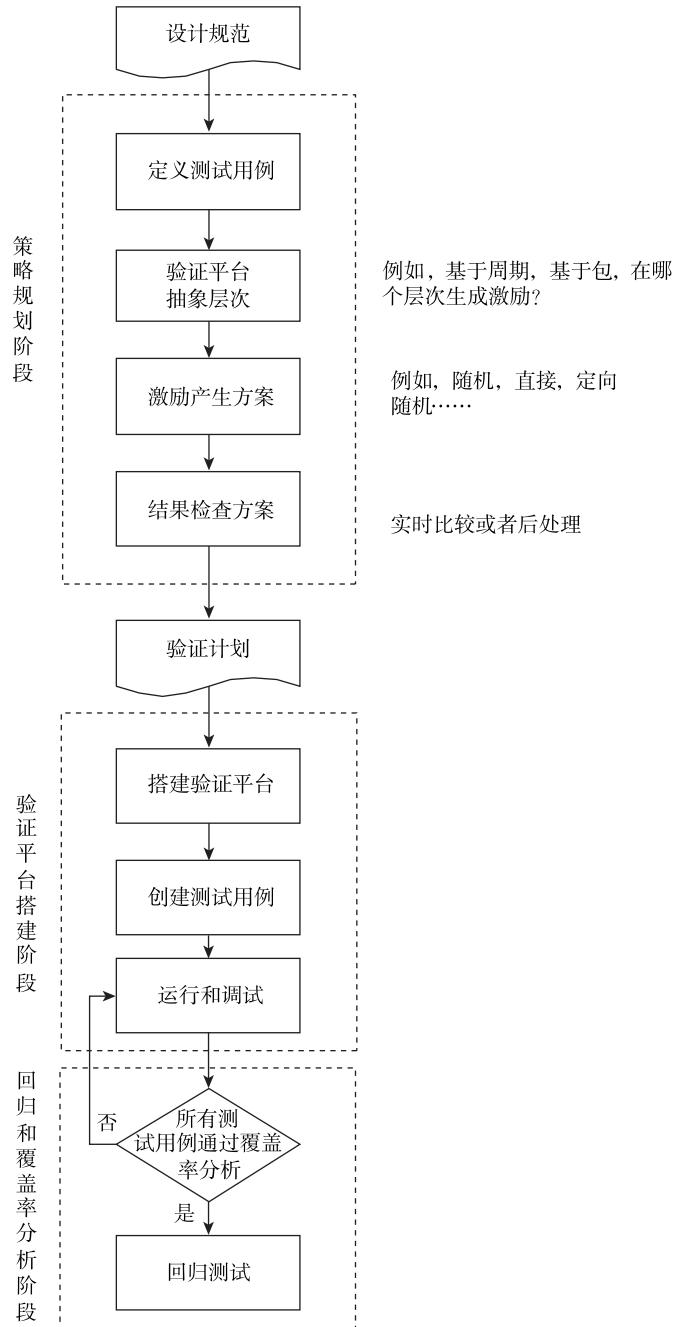


图 1-6 验证基本流程

模，例如包这个层次，它就能够自动的生成和注入数据包，而不需要验证人员去手工的创建每一个数据包。这个方法将极大地减少创建测试用例的时间，从而获取更高的测试覆盖率。这将大大解放验证工程师，使之可以专注在一些很难查找的边界情况（corner-case）上面。提高验证平台的抽象层次，也使得创建测试用例和检查结果更加容易。

然而，其中也存在需要折中考虑的地方，在一个抽象层次较低的验证平台中，验证人

员能够更有力地控制激励产生和测试用例执行的过程，从而可以创建特定的测试用例。但是，在高层次的抽象中，验证工程师就很难做到，例如在某个时刻修改某一个特定包的某些标志位或者对特定比特位注错。为击中某一个特定的情景，验证工程师必须添加额外的代码，这将导致很多不便。不管怎样，一个架构良好的验证平台要最大化的减少这种不便。

在哪个抽象层次建模，需要根据设计的具体情况而定。例如在一个 PCMCIA 接口中，抽象层次的建模可以在比特或者字节。在以太网媒体接入层（Ethernet MAC），如果验证平台在比特或者字节层次操作，每个以太包需要通过手工创建。如果在包这个层次抽象建模，使用者就能够对包头的数值和以太包的数量指定一个范围，验证平台可以自动生成包和检查媒体接入层的输出，验证以太包是否被正确处理。在一个更高层次的抽象，例如网络交互机，工程师可以构造各类以太包的组合序列，生成各种不同的激励流，从而创建一个拥塞的情景。在这种情况下，包头和负荷将通过验证平台随机产生，使用者将关注拥塞的情况。

（3）激励生成和结果检查原则

这些原则定义了输入到验证平台的激励是如何提供的，结果是如何检查的，并判断测试是通过还是失败。

2. 验证平台搭建阶段

这是第二个阶段，也就是验证平台的搭建，书写测试用例和调试阶段。在这个阶段，书写验证平台代码和测试用例。验证平台持续被扩展，因为测试用例不断被添加进来。其中，验证平台的搭建要以可重用为基本原则，而且能够方便设计工程师和验证工程师添加测试用例。

3. 回归测试和覆盖率收敛阶段

一旦几乎全部测试用例被成功运行，验证就进入了回归测试（regression test）和覆盖率收敛阶段。回归测试要求能够周期的批处理运行，二是激励必须能够容易得到重现，成功或者失败能够自动检查。覆盖率显示出该设计被测试的程度，是验证收敛的重要标准。

在这个阶段，所有的测试应该在每天或者每周做回归而且周期性的运行。设计或者验证工程师应该查看覆盖率，从而修改或者添加更多的测试用例。目标是达到 100%，或者尽可能达到 100%。

不同的设计有不同的验证流程。不存在一个对任何设计都为最好的验证流程。每个设计都有其不同的地方，而且有自身的验证流程。例如简单的设计，激励产生和结果检查策略可能将不会被显示的定义。它们可能运行在总线层次的事务交易中，验证平台使用传统的直接测试，但是它也可能不做检查。对于复杂的设计（例如交换机或者路由器），所有的激励生成方法或者结果检查规则都是预先定义好的。一般来说，对于这些设计的验证平台都是在一个较高的抽象层次上操作。例如，交换机的验证平台操作的可能不仅仅是包，也可能是不同包组成的序列。它可能创建一个有规律的包的数据流，或者创建一种拥塞的情景。

验证流程同样也可能取决于被验证设计的层次，也就是它将验证多大规模的代码。例如，一个芯片可以分解为几个模块。为此，可以为每个模块搭建一个独立的验证平台，同时可以有一个验证平台在芯片级做验证。某些设计，芯片可能是一个系统，可能存在几个芯片级或者系统级的验证平台。

每个模块级的验证平台主要的目的是帮助调试各个模块中的实现，使集成到芯片中更加容易，然而芯片级的验证平台主要测试的是各个模块之间的交互功能。

1.2 验证技术和验证方法学

本节将介绍三种常用验证手段：白盒、黑盒和灰盒验证；三种主要的验证技术：形式验证、仿真验证和硬件加速验证；三种重要的验证方法学：随机激励生成、断言验证和覆盖率驱动验证。

1.2.1 黑盒、白盒与灰盒验证

功能验证的最终目标是验证一个设计是否能够像预期的那样工作。然而，并非在设计中的任何一个设计缺陷都能通过激励在其输出边界上被检测到，为此可能会被遗漏。这样的错误可能包括下列几种情况。

- 1) 从来没有被激发过的错误。
- 2) 被激发的错误没有被传递出来。
- 3) 多个潜在的错误掩盖了其他错误。

1. 黑盒验证

黑盒验证（black-box verification）指的是只通过其边界信号来验证一个模块或者设计的功能。黑盒验证的一般的架构如图 1-7 所示。通过这种方法，激励可以被应用到设计和参考模型中；在某个抽象层次（例如事务级、指令级、时钟周期级等），通过被测设计和参考模型的输出被校对。

黑盒验证存在下列主要的缺点。

- 1) 很难验证和设计相关的特点。
- 2) 很难调试。
- 3) 要求一个精确的参考模型。

同一个设计规范，不可能所有的实现都做得等价。每一个实现都包含了很多和设计相关的细节，以达到性能或者效率上的差异，这是每个设计之间最大的区别（例如 FIFO 的读写阈值设定或者总线仲裁器的算法）。这都是通过黑盒验证方法很难验证的具体实现。

另外一个难题是，是否使用黑盒验证取决于被测的复杂度。黑盒验证需要通过多个时钟周期才能查看到传递到外部引脚上的内部错误。为此，要求更长的仿真时间来传递这个错误；即使这个错误在输出到外部检测到，也很难追溯这个问题的根源。

采用黑盒验证还有一个难题，在某种程度上要求有一个参考模型能够足够精确得可以检测所有的内部错误。对于这样一个模型，未必存在或者很难做得像设计那样，为此不可能完全依靠这种验证方法。

2. 白盒验证

白盒验证（white-box verification）和灰盒验证（grey-box verification）提供了另外一个途

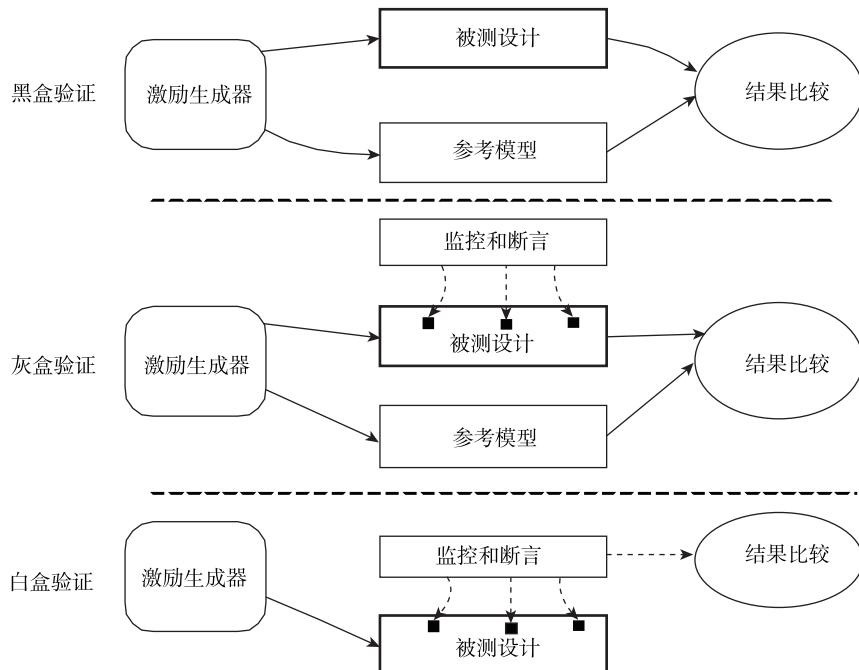


图 1-7 黑盒验证、灰盒验证和白盒验证

径来解决黑盒验证的局限性。在白盒验证中，可以不需要参考模型，因为可以通过在设计内部或者外部输出信号放置监控器和断言来保证设计操作的正确性。

3. 灰盒验证

灰盒验证是一个综合了白盒和灰盒的一个手段，也就是在设计内部添加监控器和断言来减少对参考模型的精确要求，在错误发现的时候也减少调试的压力。

1.2.2 验证技术

可以用来作为功能验证的技术主要可以分成以下三类。

- 1) 形式验证 (formal verification)。
- 2) 仿真验证 (simulation based verification)。
- 3) 硬件辅助加速验证 (hardware assisted verification/acceleration and emulation)。

下面对这些技术提供一个概述。

1. 形式验证

形式验证采用逻辑和数学公式的方法来证明或者否定硬件的特性。其中要重点强调的是，形式验证通过等式来描述和分析系统而不是测试向量。很多设计特性需要通过形式验证工具来证明，从而保证所有可能的测试向量能够应用到特定的行为中。

和其他验证技术相比，形式验证有如下两个主要的优点。

- 1) 形式验证技术可以是设计实现特性的完备声明，也就是全部可能输入流。

2) 形式验证技术不需要测试向量作为输入。

这些形式验证的优点使得这个验证方法可以让验证得更加全面，也就是设计的特性包含了所有的可能性。同样，第二个优点允许形式验证能够应用在当验证平台和测试向量还没有开发的时候。

形式验证技术主要有等价性检查（equivalence checking）和属性检查（property checking）两个分类，这些方法在下面详细讨论。

(1) 等价性检查

在这种方法中，做等价性检查的形式验证工具要证明：在所有可能的输入组合和序列下，两个硬件实现的功能是等价的。例如有限状态机，在所有可能性的输入序列和同样的初始条件下，会产生完全一样的输出序列。

等价检查工具的输入一般是两个设计的表达形式——在给定转换前后设计实现形式（例如，综合前的 RTL 代码和综合后的网表）。等价检查工具会为每个实现创建正则模型，在特定的假设条件下（例如变量的顺序），每个布尔功能有特定的正则模型表示。一旦在某个转换前后的实现存在其对应的正则表达式，那么在理想的条件下，证明两个表达式的等价性就很直接了。

实际操作中，通过一个等价性验证工具来比较一个实际的设计和该设计的变换形式，是一件困难的事情。一般来说，一个等价性验证工具要求存在转换前后一个正式的实现模型。例如，针对 RTL 代码和门级网表做等价性检查，也就是最普遍的等价性工具的应用。

然而，在没有一个完整的 RTL 代码阶段，也就是不存在一个完整的表达式的时候，等价检查工具是没有太多需要的；为此，等价性检查工具对功能验证前期遇到的挑战没有太多帮助，最好的应用模式是在整个设计流程的后期。对于形式验证，属性检查方法提供了一个强大的技术来解决验证的面临的困难。

(2) 属性检查

形式属性检查（断言验证）对设计的特定形式实现，例如 RTL 描述，属性检查验证给定属性描述的时序逻辑（temporal logic：时序逻辑表示一个系统的规律和符号，来代表和辨析两个逻辑变量在时间上的关系，例如一个时序表达式可以申明 A、B 两个变量之间在不同时钟周期上的期望关系）。

图 1-8 所示是一个属性检查的例子，用来证明或者否定硬件实现是否符合期望的属性。如图所示，描述的属性可以被重新形式化成一个等价表达式，用来评估对错，在属性变量和设计之间分析它们的关系。在这个例子中，变量之间的关系在正确的设计中是存在的，为此属性等式一致都被评估为真。同样，对于错误的设计，属性评估也就得出一个结论，错误的设计没有符合属性的要求。

另外，属性检查在功能验证中体现强大作用的还包括如下几个方面。

1) 属性能够在设计流程的任何阶段被描述出来，包括在系统设计阶段属性的定义、把属性指定到微观结构的设计模块中等。

2) 属性描述可在设计规范和开发进行的过程中，不断的搜集和添加。

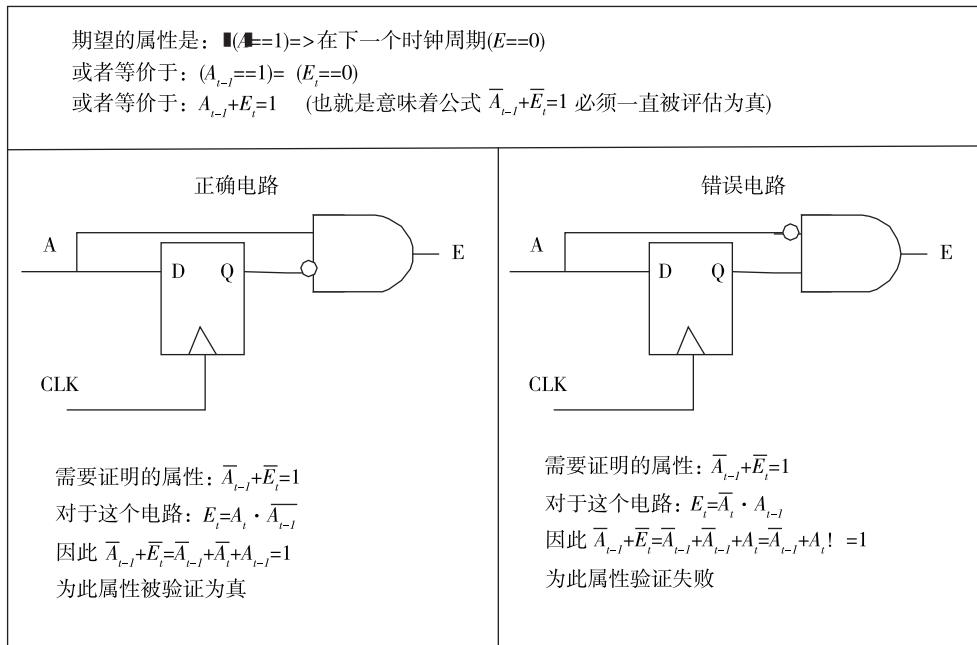


图 1-8 形式属性检查例子

3) 属性描述可以在设计的开始阶段, 当验证环境没有搭建好的时候, 为形式属性检查工具提供测试激励。

4) 属性间接地定义了覆盖率统计点, 可以用作检查验证过程的完备性。

目前, 已经开发出了多种语言来提供有效和强大的属性描述。PSL 语言是标准化语言, SystemVerilog 也提供了专门的语法可以定义断言的属性, 称为 SVA (SystemVerilog Assertion)。

2. 仿真验证

仿真验证中的基础步骤是在给定当前状态和输入的情况下, 通过计算下一个状态设计信号的值, 同时考虑信号的延迟, 把未来值赋给设计中的信号。在仿真工具中, 每次对下一个状态的求值就是一个 delta cycle (极短的时间片断)。仿真包含了一系列连续的 delta cycle 的求值计算, 也就是对信号在下一个时刻的赋值操作被预定。当没有未来的赋值操作被预定的时候 (例如, 没有任何设计信号会被改变), 仿真完成结束。仿真也可以显性的通过程序和工具控制机制来停止。

验证环境包括了验证平台和设计。在基于仿真的验证中, 最基本的操作是通过验证平台来应用激励数据到设计中, 计算设计下一个状态的值, 检查下一个状态是否符合设计期望。一个验证场景 (scenario) 是对应列在验证计划中, 连续将设计带入不同状态的一个执行过程。

图 1-9 展示了这样一个过程的表示形式。设计的所有状态空间在图中显示出来。一个仿真运行在一个初始状态, 而下一个状态是可以通过当前状态和输入到设计的值来确定的。一个仿真的追溯就是一次穿越, 对应着在给定路径和特定状态空间中, 一系列设计观测值的集合。在状态空间中, 一次穿越的路径可以通过验证场景在仿真器中的执行过程来确定。

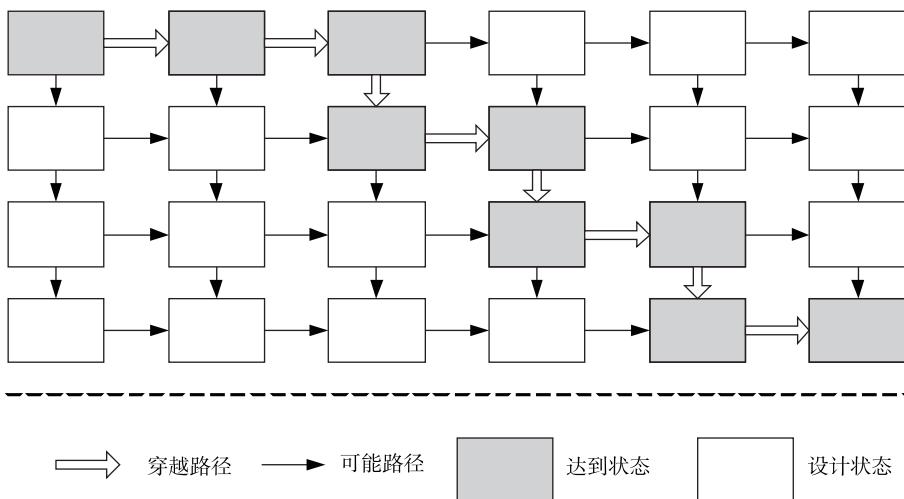


图 1-9 基于仿真的验证过程

3. 硬件辅助加速验证

基于形式和仿真的验证技术在设计流程的开始到中间阶段提供了很多便利。但是，当一个系统和应用软件要被验证的时候，这些技术的速度就远远落后于整个设计流程后段所要求的仿真速度。另外，对于需要通过仿真验证的大型设计，系统级仿真同样需要运行上万乃至百万个指令，以便应用软件的大量行为能够被验证。硬件辅助加速技术恰好满足这种需求。

硬件辅助加速最基本的思想就是把设计映射到可配置平台，例如 FPGA 或者通用运算单元，以便数字设计部分可以在接近最终产品的时钟速度下运行。

如图 1-10 所示，硬件加速（hardware acceleration）是指使用可配置平台来仿真被测设计，在硬件加速中，验证程序（testbench）在主机（host computer）中运行。在硬件模拟（hardware emulation）中，激励到设计的数据可以通过实际的接口获取，整个验证环境是真实的，可以通过软件和硬件的检测机制对验证过程进行监测和调试。

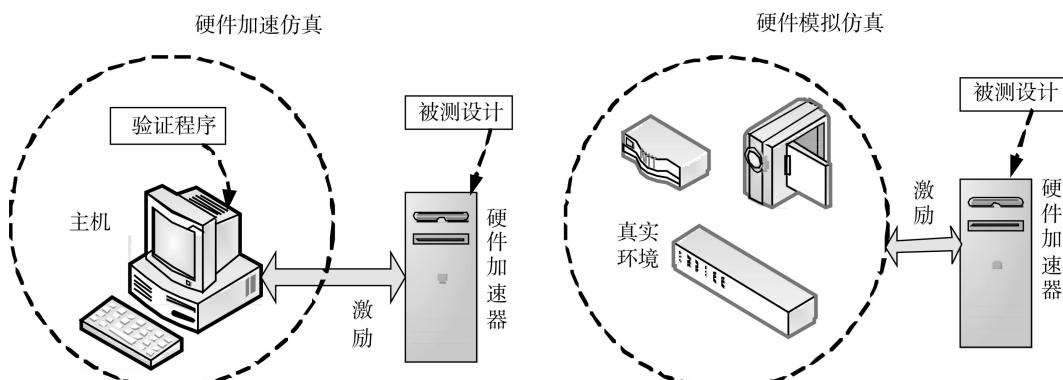


图 1-10 硬件辅助验证：硬件加速与硬件模拟仿真

1.2.3 验证存在的挑战

功能验证是芯片设计流程中的一个挑战。随着设计规模的变大和复杂性的提高，上市时间的缩短，这都意味着验证工程师要在更短的时间内验证一个更大，更复杂的设计。一个有效的验证方案必须解决以下挑战，来达到验证收敛。

(1) 完备性

验证完备性的挑战，就是如何最大限度地验证被测设计的行为。难点在于如何获取所有必须被验证的场景。这是手工操作的过程，可能存在错误而且冗长。在这个领域，最重要的进步是采用覆盖率驱动的验证方法学。覆盖率驱动验证方法学要求制定一个可以数量化衡量完备性，可追踪，有组织的验证计划。这对于验证计划的严格要求可以暴露出可能被遗漏的相关场景。

(2) 可重用性

验证可重用性的挑战是如何优化验证环境的架构，使之可以在不同的场合重用（下一个版本的项目或者同一个版本的不同层次或者一个类似的项目）。重用可以通过以下几种方式来实现：模块化验证组件、采用标准的接口、将激励产生机制和验证架构分离等；一个项目中，深层次的重用就是如何实现一个验证平台可以供多个测试用例使用。当然，验证平台的重用的程度取决于验证工程师投入多少人力和时间去规划和优化，其中也要折中考虑到项目的进程和投入回报。

(3) 可靠性

验证可靠性的挑战在于如何减少在完成一个验证项目中的手工操作。很明显，手工操作可能存在错误、冗长和耗时很大。相反，自动化系统可以在更短的时间内完成更多的工作。然而，自动化系统必须通过手动搭建。为此，在可靠性上的改进必须细心分析在搭建自动化系统上的投入和该系统的可靠性。约束随机验证是一个很好的方法，通过搭建一个自动化的系统来产生激励和自动比较，进而提高验证的可靠性。

(4) 效率

效率就是如何在给定的时间内使对验证工作投入的产出最大化（例如，调试失败场景的个数、验证环境模块的实现等）。提高验证效率已成为功能验证中的最重要的挑战。在设计流程中，重用等技术已经给予了设计工程师更高的效率。验证效率的提高已远落后于设计方面，这使得功能验证成为了完成一个项目中的瓶颈。高效的功能验证要求消除在设计和验证之间的这个鸿沟。提高验证效率可以通过提高验证平台的抽象层次和采用重用的。

(5) 性能

验证程序性能上的挑战就是如何最大化验证程序的效率。消耗在一个验证项目上的时间主要是由验证工程师投入的人工来决定的，为此，在设计和搭建验证环境中验证程序性能是第二个要考虑的因素。在运行回归测试的时候，迭代周期主要为验证平台运作的有效性来决定，验证程序的性能成为这个阶段的主要因素。对专业工具和语言的掌握是实现一个验证环境和改善验证性能的必要条件。

1.2.4 验证方法学

验证的活动渗透到设计的整个过程中，已有多个技术和多种方法可以应用到验证活动

中；所谓方法学就是：在某种科学、艺术或学科采用的方法、流程、运作概念规则和基本原理；功能验证的方法学就是验证电子系统的技术和科学。

验证一个设计需要回答两个问题：“Does it work?”（DUT 能够正常工作么？）（在验证计划中可以分解成：测试什么功能点？怎么测？最后功能对不对？）和“Are we done?”（我们做完了么？）。第一个问题属于最基本的验证概念，也就是说设计能否符合设计者的意图；第二个问题就是问我们的验证是否充分和完备，验证是否达到收敛。我们可以通过这两个问题来揭示整个设计和验证的流程。如图 1-11 所示。

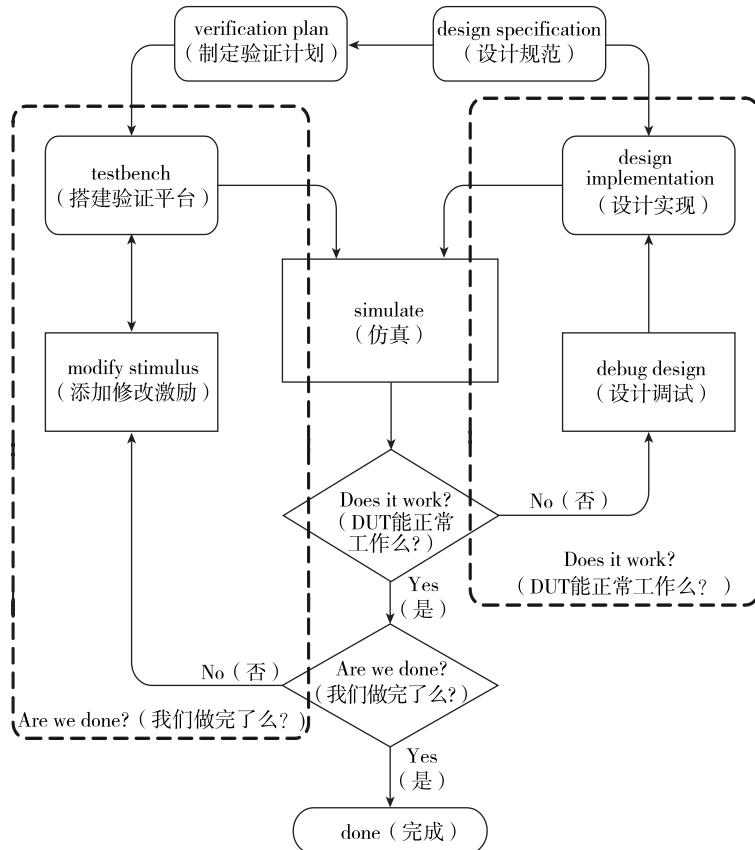


图 1-11 双环路流程

随着芯片规模的增大，基于事务交易和约束随机测试技术在验证中的广泛采用，验证调试和验证收敛的问题日益凸现，“Does it work？”和“Are we done？”两个问题充满挑战。基于断言验证和覆盖率驱动验证就是解决这两个问题主要的方法学。基于断言的验证关注断言如何被一致的使用在整个设计流程和不同的工具上，以协助工程师更快的定位错误。覆盖率驱动主要关注的是如何制定一个可衡量的标准和验证计划，以此作为指导更快地达到验证收敛。这些方法学存在交叉的地方，例如断言可以作为覆盖点成为覆盖分析的一个部分。

采用断言验证（ABV）、约束随机激励测试（CR TB）、覆盖率驱动验证（CDV）等技术将大大提高验证效率，为验证团队缩短验证周期，快速定位错误，加速激励生成和有效

的实现验证收敛，如图 1-12 所示。

这些方法学在下面做详细讨论。

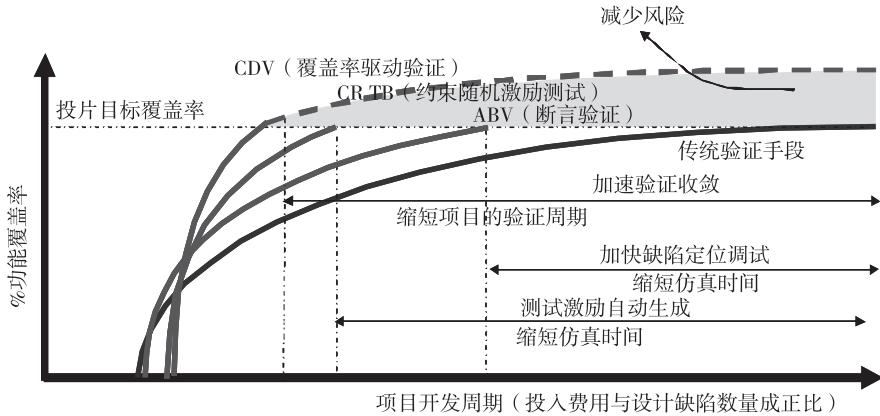


图 1-12 高级验证方法学和技术提高验证效率

1.2.5 断言验证

一般来说，断言是一个必须验证的，关于设计期望行为（也就是属性：property）的声明或者描述。断言验证（Assertion Based Verification, ABV）主要的目的是保证设计和设计期望之间是一致的。

- 1) 在属性评估过程中，如果一个被检查的属性和期望不符，那么这个断言就失败了。
- 2) 在属性评估过程中，如果一个被禁止在设计中出现的属性产生了，那么这个断言就失败了。

属性可以从设计的设计规范和功能描述中推知，并转化成为断言（属性描述）。这些断言可以在功能仿真过程中被监控评估，或者通过使用形式验证工具静态的分析。断言，又被称为监控器或者检验器，是一种在验证流程中使用多年的调试技术。传统上，它们是由过程语言，比如 Verilog 或者 VHDL 来实现的，也可以使用 PLI 和 C/C++ 的程序来实现。下面就是一个利用 Verilog 检查 a 和 b 两个互斥信号的断言，其中信号 a 和信号 b 不能同时为高电平，否则，打印错误信息。

```
`ifdef assertion
`ifndef a
`ifndef b
`endif
`endif
`ifndef a & b
`$display("Error:Mutually asserted check failed.\n");
`endif
`endif
```

断言还可以描述时序相关的设计属性，例如，当信号 req 在当前时钟为高电平之后，下面 1~3 个时钟周期内，信号 ack 应该变为高电平。

```
always @ (posedge req)
begin
repeat (1) @ (posedge clk);
fork :pos_pos
begin
```

```

    @ (posedge ack)
    $ display("Assertion Success", $ time);
    disable pos_pos;
end
begin
repeat (3) @ (posedge clk);
$ display ("Assertion Failure", $ time);
disable pos_pos;
end
join
end // always

```

断言可以嵌入在设计代码中，或者作为一个附属文件在必要的时候添加到验证平台中。目前，存在 PSL、OVA、SVA 等多种属性描述语言，这些语言可以更加完美和简练地描述时序相关的信息，提供对时间的卓越控制，精确而且易于维护。断言可以在不同的验证工具和技术中被评估使用：形式分析，仿真，硬件加速仿真。

断言的优点在于提高了验证平台对设计内部的可见性，断言验证的过程如图 1-13 所示，帮助工程师更快的定位问题的根源，缩短调试周期，提高功能覆盖率和加速系统集成。

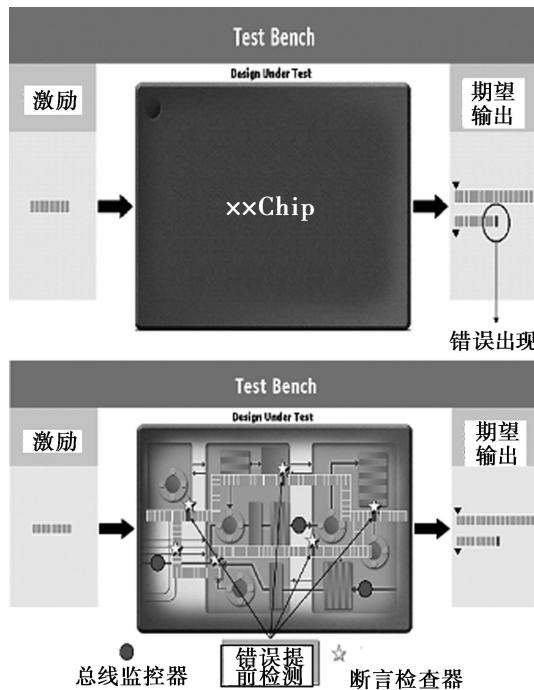


图 1-13 断言验证

1.2.6 覆盖率驱动验证

覆盖率驱动验证（Coverage Driven Verification, CDV）是一个基于仿真的验证方法，专门被开发用来解决当前功能验证项目面临的效率和完备性挑战。通过使用覆盖率驱动的验证手段，验证工程师能够在最短的时间内验证尽可能多的情景，达到更高的覆盖率，直接

的改善验证的完备性和正确性（完备性和效率）。

覆盖率驱动验证的方法最重要的特点是基于随机激励产生。随机激励生成是提高效率的最主要的原动力。

覆盖率驱动验证方法学把下面几个概念和技术融合到了一块：事务级验证、约束随机激励产生、自动化结果比较、覆盖率统计分析和直接测试。

事务级验证允许在一个更高的抽象层次来创建验证场景。约束随机激励产生可以获取更高的效率，为生成验证场景和自动化结果比较提供了充分的保证，设计能够在各种随机的情况下被验证。覆盖率统计是必须的，若没有覆盖率统计，就没有办法清晰的分析哪个场景被随机生成过。直接测试也是必须的，因为最终不可能全部的场景都能够随机生成。下面我们将分节介绍。

1. 事务级验证 (Transaction Based Verification, TBV)

仿真过程中，在设计中的所有的数据流，以低层次的信号比特或者比特向量的形式存在，在这个层次创建验证场景，通常效率很低。在事务级的验证中，低层次的信号活动被抽象成一个事务操作，以致能够可以通过这些高层次事务操作来描述各种验证场景。事务处理器就是把验证环境中抽象层次的活动转换成为低层次的信号活动，以至可以被 DUT 接受。如图 1-14 所示。

基于事务级的验证遵循下面一些原则。

- 1) 数据和数据流在较高的抽象层次定义（例如，帧、包）。
- 2) 验证场景在较高的抽象层次描述（例如，写存储器、执行指令）。
- 3) 事务处理器把这些抽象层次的数据和活动转换成低层次的操作和信号，以便应用到被测设计中。事务处理器可以是总线驱动器、接口驱动器或者就是一个层次到另一层次的数据协议转换器。

基于事务交易验证改善了验证的效率，可以让验证工程师在更高的抽象层次来处理。也大大方便了设计的架构模块在事务级的开发，也为验证环境中的验证组件之间提供了有效的通信方式。

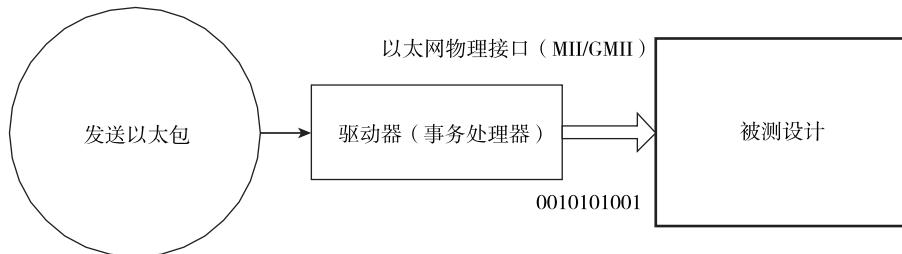


图 1-14 事务级验证

2. 约束随机激励生成 (Constrained Random Generation)

一个验证场景可以通过一系列的事务交易来组成，每个事务交易要求数据和参数值达到期望的效果。例如，在验证一个以太网包是否正确地通过以太网交换机时，事务交易包括了发包、收包，每个交易通过其中的数据和参数（带有目的地址和负荷的包）来识别。

随机激励生成指的是利用随机生成技术来产生一个事务交易中所有的数据内容，同样

产生一系列事务交易来形成一个特定的验证场景。在最理想的模式中，随机生成可以用来生成数据和场景。随机数据生成（例如，随机生成一个以太网包的源地址和目标地址）是可以简单的实现的，因为这只要求数据值被随机生成，通常对执行的场景没有太大影响。随机场景的生成，则要求验证计划需要的场景能够被随机生成，包括检查和覆盖率统计。为此，随机场景生成不仅仅需要更多的投入来实现，也要求相应的基础架构来支持这种验证方法。

随机生成提高了验证效率，因为一次随机仿真可以验证更多的场景和数据值的组合，因此减少了验证工程师搭建、执行和验证每个场景的时间。

图 1-15 比较了直接测试验证和随机激励产生验证的过程。每个验证步伐在这个图标中对应着需要使用的时间和每一步对于整体验证任务的贡献。在一个直接测试验证的方法中，验证过程通常是线性的，因为每一个新的直接测试对于整个验证流程的贡献是一致的。采用随机激励生成的方法，有一个初始的投入，因为它要花更多一点的时间来搭建一个随机的验证环境。

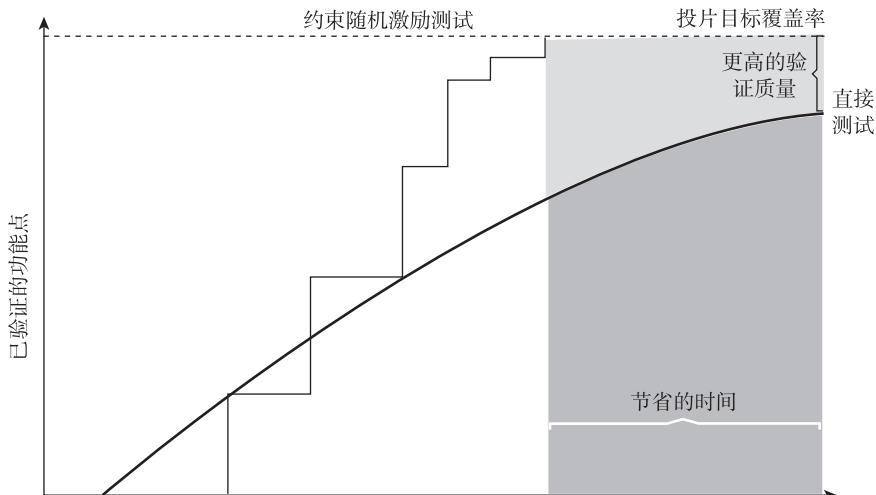


图 1-15 直接测试与随机测试比较

随机验证环境的另一个优点，是因为在验证环境中的各种随机组合交叉，使得在验证计划中没有指定的场景可能也会被生成。为此，随机激励产生不仅仅能够提高验证的效率，也有助于验证的完备性；为此，给定额外的时间和运算周期，更多的运行随机环境能够产生未制定的场景。

随机验证环境是不可以完全随机的，生成的数据和参数必须保持在一个合法的数值集合中；同样，随机场景生成需要每个仿真运行最终能够被引导到一个还没有生成过的场景。为此，在随机激励生成的机制中约束定义是一个很重要的环节。

搭建一个随机激励生成的环境不是一个实验性的任务，相对于传统的直接测试验证技术是革命性的转变。随机生成验证场景，要求每个随机生成的场景能够自动地被验证；为此，在任何随机生成环境中自动化检查是一个必要的集成部分。这也就意味着这一个随机激励生成的环境需要一个参考模型，这样各种验证场景的仿真结果能够被预测和自动检查。另外，一个随机的验证环境应该可以能够识别通过随机场景产生的所有类型的行为，这些

行为包括各种操作场景；也就是通过覆盖率统计分析机制，验证过程能够被自动的衡量。这是因为在仿真开始运行的时候，正在生成的确切场景是无法完全预知的；并且不同的仿真运行一个相同的测试将会产生不一样的场景和数据的组合。总而言之，一个包含了随机激励生成的验证环境，必须也有自动比较和覆盖率统计分析机制。

下面是随机激励产生应用的一个典型的例子，验证一个有限状态机：随机激励生成、自动比较、覆盖率统计和分析。整个过程分析如下。

(1) 随机激励产生

- 1) 初始化，把有限状态机初始化到一个随机合法的状态。
- 2) 生成随机输入。

(2) 自动化比较

- 1) 检查下一个仿真状态是否和预期的一致。
- 2) 检查仿真输出是否和预期的一致。

(3) 覆盖率统计

- 1) 统计所有覆盖的状态。
- 2) 统计状态的变迁。
- 3) 统计每个状态上的输入。

(4) 分析

- 1) 检查所有有限的状态是否被覆盖。
- 2) 检查所有的跳转是否被覆盖。

当然，尝试搭建一个可以随机生成所有可能验证场景或验证环境，在某种程度上是不切实际的。对于实际的设计来说，通常有多种使用模式，要求不同的验证环境拓扑结构和配置；某些边界情况出现概率极低，以致特殊的验证环境是要求来生成对应的场景；长时间运行一个单环境的一个单测试，可能导致重复的场景。

即使设计良好的验证环境架构和随机生成器也是受限在特定的场景类型中，为此，在同一个环境中添加遗漏的场景需要巨大的额外投入。另外，在一个单一的随机测试或环境中，对一个大的设计给定一定数量的参数和状态，它实际上不可能达到所有的边界情况。在这种情况下，最好的解决方法是，针对一些遗失的场景，创建特定约束版本的环境或者随机测试，以致这些场景可以被击中。

因为有这些限制，为了覆盖尽可能多的场景，一个随机环境通常将有很多数量的随机测试，可以覆盖验证计划的大部分；下一个阶段，随机测试将集中覆盖一些遗漏的场景，持续到最后验证结束；有些测试将是针对一个和两个特定场景的直接测试。

(1) 自动化结果检查

就如前面提及的，对于随机化的验证环境，自动化结果比较是一个必备条件。自动化结果比较可以采用监控器（monitor）和积分板（scoreboard）等技术。监控器是常常用来做协议检查和搜集设计数据流，然而积分板是用来做端到端的行为和数据比较的。

(2) 覆盖率统计和分析

随机激励生成潜在地提高了验证的效率，然而若没有一个清晰的策略来衡量验证过程，这些行为将很难被控制和实现。准确的衡量验证过程要求对于每个随机仿真的运行做下列

分析。

- 哪些场景在验证计划列出的，在一个仿真生成运行。
- 哪些没有在验证计划中列出的特定场景被生成了。
- 每次仿真运行对整个验证过程的贡献。
- 为了生成遗漏的场景，约束修改是必须的。

在每个随机仿真过程中，覆盖率信息统计是用来决定这次仿真对于整个验证过程的贡献；另外，若现存的测试用例运行更长的周期而没有任何更多的贡献时，对仿真环境进行修改或者添加新的测试用例是必要的。

对于每次仿真运行覆盖率结果同样可以用来排序测试用例，选择合适的回归集合。对覆盖率有较高贡献的测试用例将在回归环境中处于高优先级地位。

3. 直接测试

一个单一的随机环境无法创建所有的验证场景，为此通常存在多个环境或者测试用例；每个新的变化都关注在创建前面没有覆盖过的新场景。最后留下来的场景必须通过直接测试用例，这需要创建一个特殊的验证环境或者一个高度约束的测试用例。

添加直接测试在项目管理中存在很多困难，特别是一个需要定制的环境；例如直接测试用例需要手动的创建和验证，降低了全局的效率；另外，这样的特殊测试用例很难使用、维护和在项目之间重用。因此，搭建验证环境需要周密考虑以减少设计直接测试用例的数目。

覆盖率驱动验证项目的开发周期

一个覆盖率驱动的验证项目的开发周期如图 1-16 所示。

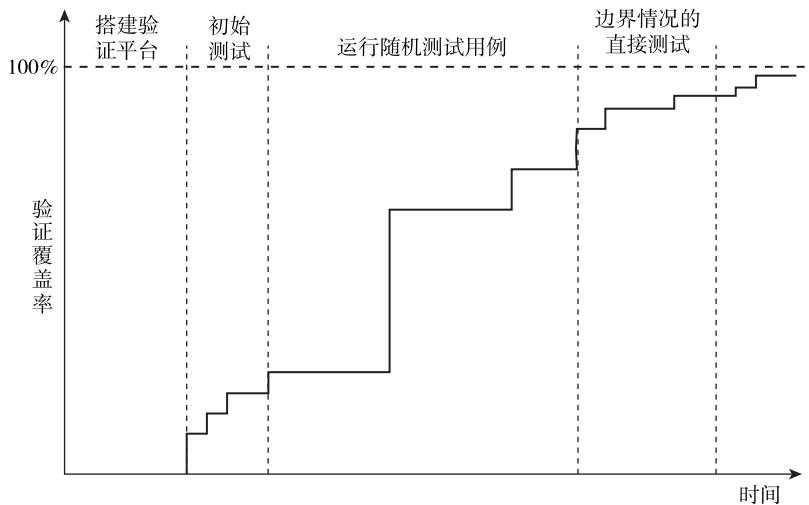


图 1-16 覆盖率驱动验证方法学的各个阶段

一个覆盖率驱动的验证项目的开发周期包括下面几个阶段。

(1) 开发验证计划

在第一个阶段，验证计划是根据设计规范和系统工程师、设计工程师、验证工程师的反馈来开发的。在这个验证开发周期，验证计划将被扩展和完善。为此，在这个阶段的主要目的是拥有一个足够详细的验证计划，以致可以创建一个稳定的验证环境的架构。

(2) 搭建验证环境

在第二个阶段，根据验证计划中制定出来的需求搭建验证环境。

(3) 测试运行环境

在第三个阶段，高度约束的验证场景被创建来检查验证环境是否正确运行，也可以调试一些已存在的错误；在这个阶段的目标是确保验证环境能够正确运行，验证过程在这个阶段是要考虑的次要问题。在环境足够稳定的使用到下个阶段之前，对阶段 2 和 3 多次测试是必须的。

(4) 运行随机测试用例

在第四个阶段的目标是通过仿真运行，尽可能多地覆盖验证计划列出的功能点。覆盖率结果将被用来决定连续的运行一个测试用例是否对整个验证过程有贡献。新的测试用例和环境将会被开发，用来针对还没有被验证过的特定的验证场景。这个阶段完成验证计划大部分的场景，除了一些必须独立考虑和通过高度的定制环境或者测试用例来测试的情况。

(5) 针对边界情况创建直接测试

在第五个阶段，通过直接测试用例生成遗漏的验证场景。就如前面提到的，目标应该是使定制测试用例的数量最少。

1.3 硬件验证语言

前面提到，功能验证已经成为制约高度复杂的电子系统和芯片设计发展的主要瓶颈。为了解决验证中存在的挑战，提高验证的完备性、可靠性、可重用性和验证效率，硬件验证语言（Hardware Verification Language, HVL）的重要作用也在业界引起了广泛的关注。早在 20 世纪 90 年代中期，业界就诞生了两门硬件验证语言：Verisity 公司的 e 语言和 Systems Science 公司的 Vera 语言；目前，SystemVerilog 已经逐渐取代两者成为了业界最流行的硬件验证语言。

1.3.1 Open Vera

Open Vera 来自 1995 年的 Vera（Systems Science 公司推出的私有语言），主要用于搭建 Verilog 的验证平台，后续也提供了对 VHDL 支持。Synopsys 公司在 1998 年收购了该公司，在 2001 年将该语言公开化并且重新命名为 Open Vera。在公开化的进程中，Synopsys 还把 Intel 开发的 ForSpec 语言添加到断言描述的部分当中，使其更加容易的在仿真过程中来检查某些信号的行为。

Open Vera 是一门容易掌握的语言，它包含了很多语言的特点：HDL、C++ 和 Java 等，提供了专门的语法结构来加速功能验证平台和断言验证的开发。

相对于传统的编程语言，Open Vera 有主要的三个优点：约束随机变量的生成，统计和报告功能覆盖率，定义和检查时序（temporal）断言。

1.3.2 e 语言

e 语言由 Verisity 公司在 1998 年开发，是 Specman 验证产品的一个部分。e 语言是一门

面向方面编程（Aspect Oriented Programming，AOP）语言。Verisity 公司在 2003 年向 IEEE 捐赠 e 语言，经多次修订后终于在 2006 年被标准化为 IEEE1647。2005 年，Cadence 收购了 Verisity。目前，e 语言主要由 Specman 这个产品支持，当然也有其他公司开始开放 e 语言的解析器。

e 语言的主要特点是：面向方面编程语言，方便修改和添加约束；通过 eVC（e - Verification Component）搭建的验证平台具高度的可重用性；能够产生约束随机激励，以提高验证效率；支持功能覆盖率定义和统计，提高验证的完备性；支持时序检查的断言，有助于作协议检查；具有高度的可扩展性。

1.3.3 PSL

PSL（Property Specification Language），是从 IBM 开发的 Sugar 这门属性语言演化而来。PSL 主要是用来定义硬件设计中需要检查的时序属性，规定严格而且有很多形式语法。

这种语言在 1994 年就被采用在 IBM 的 RuleBase 形式验证系统中，在 1997 年也被临时征用为仿真器的校验器。EDA 标准化组织 Accelera 在 2002 年将其采用作为组织的形式属性语言，在 2005 年该语言被标准化为 IEEE1850。

IEEE 认为 PSL 语言将有助于软件开发者节省验证时间、简化验证过程、降低验证成本，同时改善质量。它详细规范了电子系统设计中的相关内容、声明及方法。

1.3.4 SystemC

SystemC 是 20 世纪 90 年代后期由 Synopsys 推出的，目的是为了代替 Verilog 和 VHDL，作为可以综合的系统描述语言，它在电子系统级设计（Electronic System Level，ESL）上体现了其优点：事务级建模、行为建模和高层综合。目前，SystemC 由 Open SystemC Initiative（OSCI）负责支持、维护和发展。

SystemC 是基于 C++ 开发的系统级设计语言。实际上，SystemC 就是由 C++ 实现的一个库，其中包括了用来进行系统描述的函数和宏，主要是为使基于标准 C++ 的描述的程序能够模拟并行进程。SystemC 语言的发展大致经历了两个大的阶段：SystemC 1.0 和 SystemC 2.0。SystemC 1.0 可以用来进行硬件描述。SystemC 2.0 的推出使 SystemC 成为真正的系统级设计语言，能够用来对 SoC 体系结构进行更加自然和有效的描述。这样就使 SystemC 语言涵盖了从系统概念直到实现的针对系统软、硬件建模的能力。

SystemC 在现今验证领域的重要性，还体现在它有一个非常实用的验证库 SCV。OSCI 扩展 SystemC 具有强大的验证技术，包括新的数据结构、增强的并发功能以及约束随机化。并且 OSCI 接受 Cadence 捐献的基于 Cadence TestBuilder 开发的成熟的验证库，作为 SystemC 验证标准库，简称 SystemC 验证库（SCV）。它能够使用单一语言生成系统级的设计和全面的测试平台，包括验证单元，如激励生成器、监视器、检查器和事务处理器（transactor）。

1.3.5 SystemVerilog

SystemVerilog 是业界新兴的工程语言：硬件描述和验证语言（Hardware Description and

Verification Language, HDVL)；这个统一的语言使得工程师可以建模大型复杂的设计并且验证这些设计的功能是否正确。

SystemVerilog 是对 IEEE Std1364 – 2001 Verilog Standard 的一个扩展，其由 Accellera 标准组织维护并提交标准化，在 2005 年 12 月被标准化为 IEEE P1800—2005。SystemVerilog 对 Verilog 的扩展可以归纳为如下两个方面。

- 1) 对硬件建模的扩展，主要集成了 SUPERLOG 和 C 语言的很多特点。
- 2) 对验证和断言方面的扩展，主要集成了来自 SUPERLOG、VERA、C、C++ 和 VHDL 语言的特点，另外还有来自 OVA 和 PSL 的断言。

SystemVerilog 标准化的过程得到了 EDA 业界的鼎力支持，其主要技术来源于几个方面的贡献，其中包括：Co-Design Automation 公司的 SUPERLOG 扩展综合子集 (SUPERLOG ESS)，SYNOPSYS 公司的 Open Vera、PSL 断言（前身为 IBM Sugar），Mentor Graphics 公司的对 SVA 和 DPI 部分的捐赠，独立编译和 \$readmem 等的扩展。

SystemVerilog 有如下优点。

- 1) 单一，同时支持设计和验证的标准语言。
- 2) 支持约束随机的产生。
- 3) 支持覆盖率统计分析。
- 4) 支持断言验证。
- 5) 面向对象的编程结构，有助于采用事务级的验证和提高验证的重用性。

作为一门标准语言，SystemVerilog 得到了 Mentor Graphics、SYNOPSYS 和 Cadence 三大 EDA 厂家的一致支持，被业界各大公司广泛使用到实际项目中。QuestaSim、VCS 和 Incisive 等多个仿真器都支持 SystemVerilog 编译仿真，OVM 和 VMM 等基于 SystemVerilog 方法学已被广泛采用；SystemVerilog 已逐步取代 VERA 和 e 语言成为设计和验证工程师的首选语言。

在后续的章节中，我们重点讨论 SystemVerilog 在功能验证上的应用，其中包括新增数据类型，面向对象编程——类和对象，基于对象的随机产生及约束，验证组件的通信，以及断言验证。

第 2 章 | SystemVerilog

数据类型与编程结构

本章将介绍 SystemVerilog 的数据类型和编程结构，其中我们重点讨论 SystemVerilog 标准中新增的数据类型，例如：两态数据类型、枚举类型、用户自定义类型、压缩数组、动态数组、关联数组、队列、字符串等；另外，我们会介绍基本的过程语句、子程序和编程结构。

2.1 数据类型

除了 Verilog 已提供的硬件设计所需要的网线和变量等数据类型外，SystemVerilog 增加了很多新的数据类型来帮助描述更抽象的硬件行为，如图 2-1 所示，这些新增加的数据类型主要是与 C/C++ 语言的数据类型类似，这里介绍验证中主要使用到的部分。

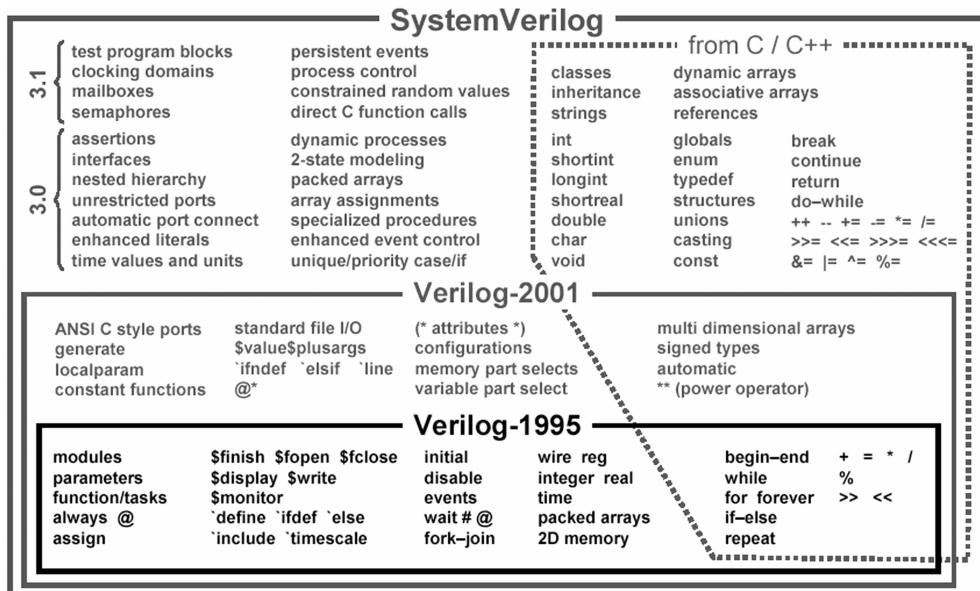


图 2-1 SystemVerilog 3.1a 数据类型

SystemVerilog 新增的主要数据类型主要有下面几种。

- 两态 (1/0) 数据类型
- 枚举类型和用户自定义类型
- 动态数组、关联数组和队列
- 联合体和结构体
- 字符串
- 类

2.1.1 两态数据类型

Verilog 1995 有两类基本的数据类型：变量（reg）和网线（wire），这是四态的数据类型（0、1、Z 和 X）。RTL 代码使用变量来存储组合逻辑和时序逻辑的数值，可以是标量或者是向量（reg [7:0] bus_addr）、有符号数 32 位变量（integer）、无符号数 64 位的变量（time）和浮点数（real）。变量也可以用来定义一个固定大小的数组。所有这些变量的存储是静态（static）的，也就意味着所有的变量在整个仿真过程中不能使用堆栈来保存参数和当前值。网线用来连接两个设计模块，如门级元件或者模块例化。

SystemVerilog 添加了很多新的数据类型来帮助硬件工程师和验证工程师。

两态 (1/0) 数据类型：SystemVerilog 引入了两态的数据类型（如图 2-2 所示）来减少仿真器对内存的使用和提高仿真的运行效率，其中 bit 是无符号数，其他四个是有符号数：short int、int、longint 和 byte。

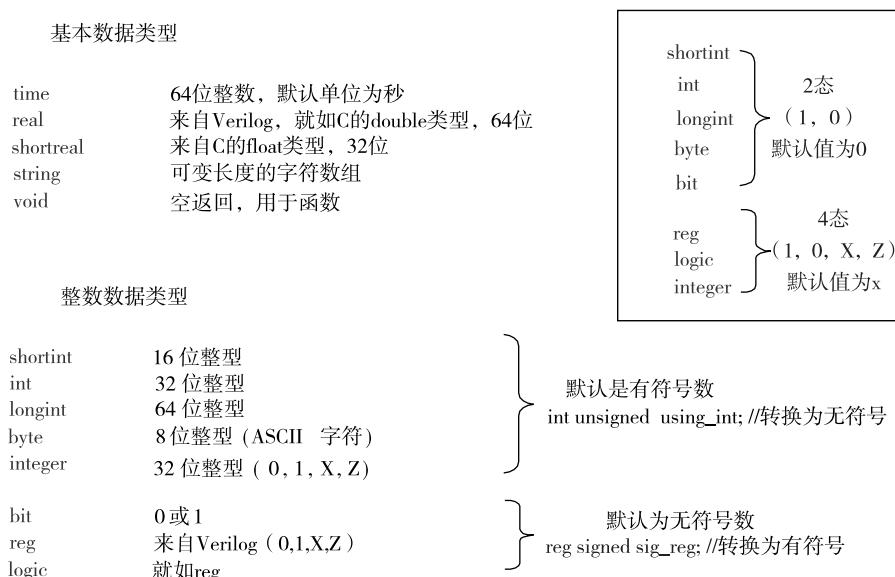


图 2-2 两态和有符号数据类型

例子：

```

bit      a;          //两态,单比特
bit [31:0] b32;    //两态,32比特无符号数
int      c32;        //两态,32比特有符号数
byte     d8;         //两态,8比特有符号数
  
```

```
shortint e16;      //两态,16 比特有符号数
longint f32;       //两态,32 比特有符号数
```

在数据算术运算操作的过程中，要时刻注意区分有符号（shortint/int/longint/byte/integer）和无符号数据类型（bit/reg/logic），上述这些数据类型都属于整型数据类型（integer type）。

SystemVerilog 对 reg 的数据类型作了改进，可以被连续赋值语句、门逻辑和模块直接驱动。另外，SystemVerilog 还引入了一个新的四态数据类型 logic，可以代替 reg；但是不能用在双向总线和多驱动的情况下，此时只能使用网线类型，例如 wire。

2.1.2 枚举类型和用户自定义类型

1. 枚举类型

枚举类型可以用来声明一组整型的命名常量，定义具有强类型的枚举变量。枚举类型还可以使用枚举名字而不是枚举值来方便地引用或显示。

一个枚举类型可以如下定义：

```
enum [data_type] {name1 = value1, name2 = value2..., nameN = valueN} var_name;
```

当没有指定数据类型的时候，缺省的数据类型是 int。在枚举类型中使用的任何其他数据类型都要求显式地声明。

枚举类型定义了一组具有名字的值。在下面的例子中，light1 和 light2 被定义成一个未命名的 int 枚举类型（没有具体的类型标识符，后面我们会提到如何使用用户自定义类型），它包含了三个成员：red、yellow 和 green。

```
enum {red, yellow, green} light1, light2;// anonymous int type
```

无论是枚举名（red/yellow…）还是它们的（整型）数值都必须是唯一的。它们的值可以被设置为任意整型常量值，或者从初始值 0 开始递增（默认情况）。如果将两个值设置到相同的枚举名，或者设置的值与递增值冲突都是非法的定义。

SystemVerilog 枚举类型是一种强类型，枚举变量在赋值、传递和关系操作符中进行类型检查，因此枚举变量不能被在枚举集合范围以外的数值直接赋值，除非使用强制类型转换或者该枚举变量是一个联合体的成员。枚举变量可以自动转换成整型值，在表达式中作为常量使用，并且运算结果也可以赋值到任何一个兼容的整型变量。

枚举类型实例代码如源代码 2-1 所示。

源代码 2-1 枚举类型实例

```
/Chapter2 enum_example.sv
module test_enum();
//默认值:red = 0,yellow = 1,green = 2;
enum {red, yellow, green} light1, light2;// 未命名的枚举类型(int 类型)

// 正确使用方法: IDLE = 0, S0 = 2, S1 = 3, S2 = x
enum integer {IDLE, S0 = 'b10,S1,S2 = 'x} state,next;

// 正确定义方法:bronze 和 gold 都没有指定大小
enum {bronze = 3, silver, gold} medal;// silver = 4, gold = 5
```

```

// c 被自动地指定为 8
enum {a = 3, b = 7, c} alphabet1;

// d = 0, e = 7, f = 8
enum {d, e = 7, f} alphabet2;

initial
begin
    light1 = red;
    light2 = yellow;
    $display("light1 is % 0d or % s", light1, light1);
    $display("light2 is % 0d or % s", light2, light2);
    state = S1;
    next = S2;
    $display("state is % 0d or % s", state, state);
    $display("next is % b or % s", next, next);
    medal = silver;
    $display("medal is % 0d or % s", medal, medal);
    alphabet1 = c;
    $display("alphabet1 is % 0d or % s", alphabet1, alphabet1);
    alphabet2 = d;
    $display("alphabet2 is % 0d or % s", alphabet2, alphabet2);
    //下面是错误的使用方法,需要做类型转换
    //light1 = 2;
end

endmodule

```

SystemVerilog 为枚举类型提供了如下内置方法（method）来方便操作。

- function enum first ()：返回枚举类型中第一个成员的值。
- function enum last ()：返回枚举类型中最后一个成员的值。
- function enum next (int unsigned N = 1)：以当前成员为起点，返回后续第 N 个成员的值，默认是下一个成员的值；若起点为最后一个成员，则默认返回第一个成员的值。
- function enum prev (int unsigned N = 1)：以当前成员为起点，返回前面第 N 个成员的值，默认是前面一个成员；若起点为第一个成员，则默认返回最后一个成员的值。
- function int num ()：返回该枚举类型的成员数目。
- function string name ()：以字符串的形式返回该成员名字。

2. 用户自定义类型

在 SystemVerilog 中，可以通过使用 `typedef` 关键字进行用户自定义类型的扩展。定义新的数据类型可以提高代码的可读性，复杂的数据类型（结构体、联合体等）和特定的数组可以通过使用一个简单易懂的名字（别名）被定义为一个新的数据类型，例如：

```
typedef int my_favorite_type;
```

这个新的类型就可以定义对应的变量：

```
my_favorite_type a, b;
```

其实你并未创建一个新的数据类型，而只是在做文本替换；将一个特定的数组定义为

新的数据类型，例如：

```
parameter OPSIZE = 8;
typedef reg [OPSIZE - 1:0] opreg_t;
opreg_t op_a, op_b;
```

如果使用空的 `typedef` 事先对一个数据类型作出标识，那么它就可以在其定义之前使用，例如：

```
typedef foo;
foo f = 1;
typedef int foo;
```

有时，一个用户自定义类型需要在类型的内容被定义之前声明。这对于由 `enum`、`struct`、`union` 和 `class` 派生出的用户定义类型很有用处，如下：

```
tyepdef enum type_declaratiion_identifier;
tyepdef struct type_declaratiion_identifier;
tyepdef union type_declaratiion_identifier;
tyepdef class type_declaratiion_identifier;
tyepdef type_declaratiion_identifier;
```

在某些时候，自定义一个新的数据类型也是必须的，因为在 SystemVerilog 中要通过数据类型的标识符才可以做类型转换，如源代码 2-2 所示。

源代码 2-2 用户自定义类型实例

```
//Chapter2 typedef_example.sv
module test_TYPEDEF();
    typedef enum {red, green, blue, yellow, white, black} colors ;
    colors my_colors;

    initial
    begin
        $display("my_color's default value is % s",my_colors);
        my_colors = green;
        //my_colors = 1;           // 错误使用方法,需要做数据类型转换
        my_colors = colors'(2);   // 通过 colors 数据类型标识符做类型转换
        $display("my_color is % s",my_colors.name);
    end
endmodule
```

2.1.3 数组与队列

SystemVerilog 提供了下面三种数组类型和队列。

- 静态数组（static array/fixed – size array）
- 动态数组（dynamic array）
- 关联数组（associative array）
- 队列（queue）

静态数组是指其数组的大小在定义时被显性地指定。动态数组有一维或者多维，其中非压缩部分的一维的大小在定义的时候未被指定，在使用前才分配。关联数组允许通过任何类型的索引来访问数组的成员。队列被用于定义一个顺序的数据集合。下面将逐个进行介绍。

1. 静态数组与压缩数组

在 SystemVerilog 中，静态数组扩展了数组的原始概念，允许编程者来定义每一维是如何存储的，这个扩展是基于存储的有效性和访问的灵活性考虑的。为此，SystemVerilog 引入了两种类型的数组：压缩数组（packed array）和非压缩数组（unpacked array）。

压缩数组指的是维数的定义在变量标识符之前，非压缩数组指的是维数的定义在变量标识符之后，如下所示：

```
bit [7:0] c1;      //压缩数组(比特类型)
real u [7:0];     //非压缩数组(实型)
```

静态数组的通用定义方法是：

```
element_data_type [PRange1]…[PRangeN] array_name [URange1]…[URangeM];
```

在 array_name 前面指定的维数是压缩部分，在 array_name 后面指定的维数是非压缩部分，下面是静态数组的使用方法。

- 1) 对单个压缩数组的引用可以用来访问一个整体的数组。
- 2) 数组成员访问的方式如下：

```
Array_name[URange1]…[URangeM][PRange1]…[PRangeN]
```

3) 静态数组的非压缩维数可以通过 [Number - 1: 0] 或者通过 [Number] 的方式来定义：

```
int Array[0:7][0:31];// 通过范围来定义数组
int Array[8][32];// 通过指定上限定义数组
```

4) 如果一个数组被定义为有符号数，那么其存储的所有压缩数组都被默认为有符号数，而每个压缩数组的成员是无符号数。

5) 压缩数组被指定为任何网线类型或者标量变量类型（如：reg、logic 和 bit）。具有预定义宽度的整数类型不能声明成压缩数组，这些类型包括：byte、shortint、int、longint 以及 integer。一个具有预定义宽度的整数类型可以被看作是一个一维的压缩数组。这些整数类型的压缩尺寸应该以趋向 0 的方向编号，并且最右边的索引是 0。非压缩数组可以被指定为任何数据类型。

```
integer i1;// 就如 logic signed [31:0] i1
byte c2[4:0];// 就如 bit signed [7:0] c2[4:0]
```

数组可以通过下列方式进行访问。

- 对数组做整体读写，例如：A = B。
- 对数组做部分读写，例如：A [i:j] = B [i:j]。
- 对数组的可变片断读写，例如：A [x+:c] = B [y+:c]。
- 对数组的一个成员读写，例如：A [i] = B [i]。

■ 对数组部分或者整体做比较操作，例如： $A == B$, $A [i:j] != B [i:j]$ 。

对于某些数据类型，你或许想通过整体访问或者把它分解成几个部分。例如，你可以把一个 32 比特寄存器，作为四个 8 比特的寄存器或者作为一个 32 比特的整体。那么压缩数组可以帮助你实现这个目的。

注意，压缩数组的维数只能通过 [hi: lo] 方式来定义。如下所示：变量 reg_32 是一个由四个 8 比特组成的压缩数组，作为一个 32 位的长字存储。你也可以将压缩数组和非压缩数组混合定义，如源代码 2-3 中的 mix_array。

源代码 2-3 多维数组和压缩数组实例

```

//Chapter2 array_example.sv
module test_array();
bit [3:0] [7:0] reg_32;                                // 4个字节压缩为32bits的向量
bit [3:0] [7:0] mix_array[3];

initial begin
    reg_32 = 32'hdead_beef;
    $display("%b", reg_32);                            // 打印所有的32 bits
    $display("%h", reg_32[3]);                          // 打印最高位的字节“de”
    $display("%b", reg_32[3][7]);                      // 打印最高位比特“1”

    mix_array[0] = 32'h0123_4567;
    mix_array[1] = mix_array[0];
    mix_array[2] = mix_array[1];

    if (mix_array[1][3] == 8'h01) $display("Access mix_array[1][3]");
    if (mix_array[2][1][6] == 1'b1) $display("Access mix_array[2][1][6]");

    //mix_array[0] = 64, mix_array[1] = 63, mix_array[2] = 62;
    mix_array = '{64, 63, 62};

end
endmodule

```

图 2-3 是源代码 2-3 中的 reg_32 和 mix_array 的存储结构。

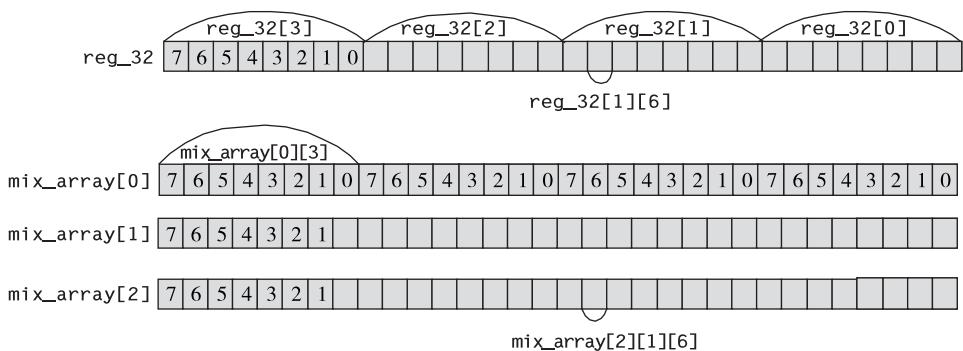


图 2-3 压缩数组的存储方式

SystemVerilog 还提供了下列系统内置方法来访问数组成员：\$left、\$right、\$low、\$high、\$increment、\$size、\$dimensions 和 \$unpacked_dimensions。

2. 动态数组

Verilog 的基本数组和向量就是前面提到的静态数组，数组的大小在编译的时候就确定下来了。但是，若在编译的时候不知道数组的大小，只有在运行的过程中才能确定。例如，事务交易的数量是随机的从 1000 到 100 000，但是你又不想使用静态数组，因为可能导致有一半的成员是空的。SystemVerilog 提供了动态数组，可以在仿真的过程中动态分配大小。动态数组可以是一维或者多维的，其中非压缩部分的一维的大小在定义的时候未被指定，其存储空间只有当数组在运行时被显式分配之后才会存在。

最简单的动态数组的声明语法如下：

```
data_type array_name [];
```

其中，data_type 是数组成员的数据类型，动态数组与静态数组支持相同的数据类型。空的“[]”意味着我们不需要在编译的时候指定数组的大小，在运行的过程中可以动态分配。动态数组初始定义时是空的，没有分配任何空间，使用前必须通过调用 new []，并在“[]”中输入期望的长度数值来分配空间。

动态数组的例子：

```
bit [3:0] nibble[]; // 4 比特向量的动态数组
integer mem[2][]; // 固定大小的非压缩数组中
// 带 2 个动态的整型子数组
```

系统函数 \$size() 可以返回一个静态数组或者动态数组的大小。另外，动态数组还有几个特殊的内置函数，例如 delete() 和 size()；这两个函数是不适用于静态数组的。如果一个动态数组与一个静态数组的维数及深度相同，那么这个动态数组可以被赋值为一个具有兼容类型的静态数组。一个动态数组或静态数组可以被赋值到一个具有兼容类型的动态数组。在这种情况下，赋值会自动分配一个新的动态数组，这个新的动态数组的长度等于静态数组的长度。动态数组实例的代码如源代码 2-4 所示。

源代码 2-4 动态数组实例

```
//Chapter2 dy_array_example.sv
module test_dy_array();
    int dyn1[],dyn2[]; // 动态数组

    initial begin
        dyn1 = new [100]; // 分配 100 个成员
        foreach (dyn1[i])
            dyn1[i] = i; // 初始化成员
        dyn2 = new [100](dyn1); // 复制一个动态数组
        dyn2[0] = 5; // 修改一个成员
        // 检查 dyn1[0] 和 dyn2[0] 的数值
        $display("dyn1[0] = % 0d, dyn2[0] = % 0d", dyn1[0], dyn2[0]);
        // 扩展大小，并复制初值
```

```

dyn1 = new [200](dyn1);
$display("The size of dyn1 is % 0d",dyn1.size());
//改变 dyn1 的大小
dyn1 = new [50];
$display("The size of dyn1 is % 0d",dyn1.size());
dyn1.delete;
$display("The size of dyn1 is % 0d",dyn1.size());
//复制一个动态数组
dyn1 = dyn2;
$display("dyn1[0] =% 0d,dyn2[0] =% 0d",dyn1[0],dyn2[0]);
end
endmodule

```

3. 关联数组

关联数组是对处理成员数目会动态改变的连续变量集合而言，动态数组非常有用。然而，当集合的大小是未知的或者数据空间缺紧的时候，关联数组则是更好的选择。关联数组在使用之前不会分配任何存储空间，并且索引表达式不仅仅是整型表达式，而且可以是任何数据类型。

动态数组可以允许在程序运行的过程中根据需求动态分配特定大小的数组。但是，若我们需要一个超大的数组呢？在具体的应用环境中，如处理器，它可以访问几个 GB 的地址空间，若我们全部分配，消耗的内存是非常巨大的；而实际测试中我们可能只需要访问某些范围或者离散的地址；关联数组是一种通过标号来分配空间和访问的数组，其好处就是只分配使用到的特定地址的空间，也就是当你访问某一个较大地址的数组时，SystemVerilog 只针对该地址分配空间。如图 2-4 所示，关联数组只分配 0~5、45、1000、4531 和 200 000 地址的数值。存储器分配也会比固定数组和动态数组小。

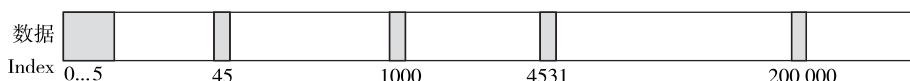


图 2-4 关联数组应用情景

关联数组实现了一个所声明类型成员的查找表，用索引的数据类型作为查找表的查找键值，并强制了其排列顺序。

关联数组的声明语法如下：

```
data_type array_id [index_type];
```

其中：

- **data_type** 是数组成员的数据类型。静态数组可以使用的任何类型都可以作为关联数组的数据类型。
- **array_id** 是关联数组的变量名。
- **index_type** 是用作索引的数据类型或者是 *。如果指定了 *，那么数组可以使用任意大小的整型表达式来索引。采用数据类型作为索引的关联数组将索引表达式限制为

一个特定的数据类型。

下面是关联数组的例子：

```
integer i_array[*];           // 整型关联数组(未指定索引类型)
bit [20:0] array_b[string];  // 21 比特向量的关联数组,索引为字符串
event ev_array[myClass];     // 事件的关联数组,索引为 myClass 的类
```

除了索引操作符外，SystemVerilog 提供了几个内建方法来让用户分析和处理关联数组，同时提供了对关联数组的索引或键值的迭代处理。关联数组实例代码如源代码 2-5 所示。

- function int num () / function int size ()：返回关联数组中成员的数目。如果数组是空数组，那么它返回 0。
- function void delete ([input index])：其中 index 是一个可选的适当类型的索引，删除特定索引的成员，如果没有指定索引，则删除数组的所有成员。
- function int exists (input index)：其中 index 是一个适当类型的索引，如果成员存在，则返回 1，否则返回 0。
- function int first (ref index)：其中 index 是一个适当类型的索引，将指定的索引变量赋值为关联数组中第一个（最小的）索引的值；如果数组是空的，则返回 0，否则返回 1。
- function int last (ref index)：其中 index 是一个适当类型的索引，将指定的索引变量赋值为关联数组中最后一个（最大的）索引的值。如果数组是空的，则返回 0，否则返回 1。
- function int next (ref index)：其中 index 是一个适当类型的索引，查找索引值大于指定索引的条目。如果存在下一个成员，索引变量被赋值为下一个成员的索引，并且函数返回 1。否则，索引不会发生变化，函数返回 0。
- function int prev (ref index)：其中 index 是一个适当类型的索引，查找索引小于指定索引的成员。如果存在前一个成员，索引变量被赋值为前一个成员的索引，并且函数返回 1。否则，索引不会发生变化，并且函数返回 0。

源代码 2-5 关联数组实例

```
//Chapter2 as_array_example.sv
module test_associate_array();
    bit [7:0] i_array[*];           // 整型关联数组(未指定索引类型)
                                    // 未指定索引(*)表明可以是任意整型
    bit [7:0] idx;
    bit [7:0] age [string];        // 8 比特向量的关联数组,索引为字符串
    string tom = "tom";
    int assoc_array[int unsigned] = '{1:1, 2:2, 3:3, 4:4, 5:5, 6:6, 7:7, 8:8, 9:9, 10:10};
    int unsigned idx_int, smallest, largest;

    initial begin
        idx = 1;
        repeat (6) begin
            i_array[idx] = idx;
            idx = idx < < 1;
        end
        if (i_array.first(idx))
```

```

begin
  do
    $display("i_array[% 0d]=% 0d", idx, i_array[idx]);
    while (i_array.next(idx));
  end

  age [tom]=21;
  age ["joe"]=32;
  $display("% s is % d years of age ", tom, age[tom], "[% 0d ages available]", age.num());
()

if (! assoc_array.first(smallest))
  $display("ERROR-Assoc array has no indexes!");

if (! assoc_array.last(largest))
  $display("ERROR-Assoc array has no indexes!");

$display("Smallest index is % 0d",smallest);
$display("Largest index is % 0d",largest);

idx_int=largest;
do
  $display("Index % 0d is set to % 0d", idx_int, assoc_array[idx_int]);
  while (assoc_array.prev(idx_int));
end
endmodule

```

4. 队列

队列是一个大小可变，具有相同数据类型成员的有序集合。队列能够在固定时间内访问它的所有元素，也能够在固定时间内对队列的尾部和头部插入和删除成员。队列中的每一个成员都通过一个序号来标识，这个序号代表了成员在队列内的位置，0 代表第一个成员，\$ 代表最后一个成员。队列类似于动态数组，是一个一维的非压缩数组，它可以自动地增长和缩减。因此，与数组一样，队列可以使用索引、串联、分片、比较操作符进行处理。队列适合于实现 FIFO 和堆栈之类的数据结构。

队列通过以下方法进行定义：

```

data_type queue_name[ $ ];
data_type queue_name[ $ :maxsize]

```

其中，`data_type` 是数据成员的数据类型，与静态数组和动态数组支持的一致；`queue_name` 是定义的队列变量，若给定 `max size`，那么队列成员的最大数将受到约束。

例如：

```

byte q1[ $ ];                                // 字节队列
string names[ $ ] = {"Bob"};                  // 字符串队列
integer Q[ $ ] = {3, 2, 7};                   // 初始化整型队列
bit q2[ $ :255];                            // 最大长度为 256 的比特队列

```

空队列文本 {} 可以用来指示一个空队列。如果声明时没有提供初始值，那么队列变量被初始化成一个空队列。

当创建一个队列的时候，SystemVerilog 实际上分配了额外的空间，为此我们可以很快地添加成员。注意，我们不用像动态数组那样调用 new [] 为队列分配空间；当我们添加到一定数量的成员之后，SystemVerilog 会自动分配额外的空间。为此，我们可以任意增加和缩减一个队列，而对性能没有太大影响。源代码 2-6 是一个队列实例。

源代码 2-6 队列实例

```
//Chapter2 queue_example.sv
module test_queue();
int queue1[$];      // $ 表示数组的上限
int n, m, item;
initial begin
queue1 = '{1,2,3,4,5};
n = 8;
m = 9;
// 使用拼接操作符把 n 放到 queue1 的最左边
queue1 = '{n, queue1};
// 使用拼接操作符把 m 放到 queue1 的最右边
queue1 = '{queue1, m};
// 将 queue1 左边第一个成员赋给 item
item = queue1[0];
// 将 queue1 右边最后一个成员赋给 item
item = queue1[$];
// 通过整数对 queue1 做遍历访问
for (int i = 0; i < queue1.size(); i++)
begin
$display("queue1[% 0d] = % 0d", i, queue1[i]);
end
// 求出 queue1 的长度
n = queue1.size();
$display("queue1 size is % 0d", n);
// 删除左边第一个成员
queue1 = queue1[1:$];
// 删除右边第一个成员
queue1 = queue1[0:$ - 1];
for (int i = 0; i < queue1.size(); i++)
begin
$display("queue1[% 0d] = % 0d", i, queue1[i]);
end
queue1 = '{}; // 清空队列
$display("queue1 size is % 0d", queue1.size());
end
endmodule
```

SystemVerilog 提供了下列预定义的方法来访问和操作队列。

- function int size ()：返回队列中成员的数目。如果队列是空的，它返回 0。
- function void insert (int index, queue_type item)：在指定的索引位置插入指定的成员。

`Q.insert (i, e)` 等效于 $Q = \{Q[0:i-1], e, Q[i, \$]\}$ 。

- `function void delete (int index):` 删除指定索引位置的成员。`Q.delete (i)` 等效于 $Q = \{Q[0:i-1], Q[i+1, \$]\}$ 。
- `function queue_type pop_front ():` 删除并返回队列的第一个成员。`e = Q.pop_front ()` 等效于 $e = Q[0]; Q = Q[1, \$]$ 。
- `function queue_type pop_back ():` 删除并返回队列的最后一个成员。`e = Q.pop_back ()` 等效于 $e = Q[\$]; Q = Q[0, \$-1]$ 。
- `function void push_front (queue_type item):` 在队列的前端插入指定的成员。`Q.push_front (e)` 等效于 $Q = \{e, Q\}$ 。
- `function void push_back (queue_type item):` 在队列的尾部插入指定的成员。`Q.push_back (e)` 等效于 $Q = \{Q, e\}$ 。

2.1.4 字符串

SystemVerilog 引入了一个字符串类型 (`string`)，它是一个大小可变、动态分配的字节数组。在 Verilog 中，字符串文本为一个具有宽度为 8 的整数倍的压缩数组。当一个字符串文本被赋值到一个大小不同、整型压缩数组变量的时候，它或者被截短到变量的大小或者在左侧填补 0。

在 SystemVerilog 中，字符串文本的表现行为与 Verilog 相同。然而，SystemVerilog 还支持字符串类型，我们可以将一个字符串文本赋值到这种数据类型。当使用字符串类型来替代一个整型变量的时候，字符串可以具有任意的长度并且不会发生截短现象。当字符串文本赋值到一个字符串类型或者在一个使用字符串类型操作数的表达式中使用的时候，它会被隐式地转换成字符串类型。

字符串类型变量的声明语法如下：

```
string variable_name [= initial_value];
```

其中，`variable_name` 是一个有效的标识符，可选的 `initial_value` 可以是一个字符串文本，也可以是一个空字符串 (" ")。例如：

```
string myName = "John Smith";
```

如果在声明中没有指定初始值，变量会被初始化成空字符串 ("")。

SystemVerilog 提供了一组操作符，这些操作符可以用来处理字符串变量和字符串文本。字符串类型的基本操作符在表 2-1 中给出，字符串类型的内置方法在表 2-2 中给出。

表 2-1 字符串操作符

操作符	描述
<code>Str1 == Str2</code>	检查两个字符串是否相等。如果相等则结果为 1，否则结果为 0
<code>Str1 != Str2</code>	检查两个字符串是否不等
<code>Str1 < Str2</code> <code>Str1 <= Str2</code>	比较、关系操作符。按字母表顺序对两个字符串做比较，如果对应的条件为真则返回 1
<code>Str1 > Str2</code> <code>Str1 >= Str2</code>	

(续)

操作符	描述
{ Str1 , Str2 , … , Strn }	将几个字符串串联，生成一个新的字符串
{ n { Str } }	对字符串 Str 复制 n 次，生成一个新的字符串
Str [index]	索引，返回一个字节
Str. method (…)	点 (.) 操作符用来调用字符串的内置方法

表 2-2 字符串内置方法

内置方法	描述
function int len ()	str. len () 返回字符串的长度，也就是字符串中字符的数目（不包括任何终结字符），如果 str 是 "", 那么 str. len () 返回 0
task putc (int i, byte c)	str. putc (i, c) 将 str 中的第 i 个字符替换成指定的整型值； putc 不会改变 str 的尺寸，如果 i < 0 或 i >= str. len ()，则 str 不会发生改变
function int getc (int i)	str. getc (i) 返回 str 中的第 i 个字符的 ASCII 码； 如果 i < 0 或 i >= str. len ()，则 str. getc (i) 返回 0
function string toupper ()	str. toupper () 将 str 中的字符转换成大写形式并返回该字符串；str 不会发生变化
function string tolower ()	str. tolower () 将 str 中的字符转换成小写形式并返回该字符串；str 不会发生变化
function int compare (string s)	str. compare (s) 将 str 与 s 进行比较，就像 ANSI C strcmp 函数一样
function int icompare (string s)	str. icompare (s) 将 str 与 s 进行比较，就像 ANSI C strcmp 函数一样，不区分大小写
function int substr (int i, int j)	str. substr (i, j) 返回一个由 str 中位置 i 到位置 j 之间的字符组成的一个新的字符串； 如果 i < 0、j < i 或者 j >= str. len ()，则 substr () 返回 ""（空字符串）
function integer atoi ()	str. atoi () 返回一个 str 中由 ASCII 码字符表示的十进制数
function integer atohex ()	str. atohex () 将字符串解释成十六进制数
function integer atooct ()	str. atooct () 将字符串解释成八进制数
function integer atobin ()	str. atobin () 将字符串解释成二进制数
function real atoreal ()	str. atoreal () 返回一个 str 中由 ASCII 码字符表示的实数
task itoa (integer i)	str. itoa (i) 将 i 的 ASCII 码十进制表示存储到 str（与 atoi 相反）
task hextoa (integer i)	str. hextoa (i) 将 i 的 ASCII 码十六进制表示存储到 str（与 atohex 相反）
task octtoa (integer i)	str. octtoa (i) 将 i 的 ASCII 码八进制表示存储到 str（与 atooct 相反）
task bintoa (integer i)	str. bintoa (i) 将 i 的 ASCII 码二进制表示存储到 str（与 atobin 相反）
task realtoa (real r)	str. realtoa (r) 将 r 的 ASCII 码实数表示存储到 str（与 atoreal 相反）

2.1.5 结构体和联合体

类似 C 语言，SystemVerilog 提供了结构体（struct）和联合体（union）两种结构。结构体的成员被连续地存储，而联合体的所有成员共享同一片存储空间，也就是联合体中最大成员的空间。

结构体和联合体的声明遵从 C 语言的语法，但在 “ { ” 之前没有可选的结构体标识符。

如下所示：

```
struct {
    bit[7:0] opcode;
    bit[23:0] addr;
} IR //未命名结构体,定义结构体变量 IR
IR. opcode=1;// 对 IR 的成员 opcode 赋值
```

其他一些声明结构体和联合体的例子如下：

```
typedef struct {
    bit[7:0] opcode;
    bit[23:0] addr;
} instruction;           // 命名为 instruction 的结构体类型
instruction IR;          // 定义结构变量

typedef union {
    int i;
    shortreal f;
} num;                  // 命名为 num 的联合体类型
num n;                  // 以浮点数格式设置 n
n. f = 0.0;
```

一个结构体可以作为一个整体赋值，并且可以作为一个整体作为接口参数在一个函数或任务中传递。结构体实例如源代码 2-7 所示。

源代码 2-7 结构体实例

```
//Chapter2 struct_example.sv
module struct_example();
    struct {
        bit [7:0] my_byte;
        int my_data;
        real pi;
    } my_struct; // my_byte、my_data 和 pi 是 my_struct 的成员

    initial begin
        my_struct. my_byte = 8'hab;
        my_struct = '{0, 99, 3.14};
    end
endmodule
```

通过使用 packed 关键字，结构体也可以类似压缩数组那样被定义为压缩结构体；压缩结构体的所有成员在存储器中被无缝地压缩在一起，如图 2-5 所示。这也就意味着，它们可以方便地转换成向量，或从向量转换过来。

与压缩数组类似，在使用算术和逻辑操作符时，一个压缩结构体可以当作一个整体使用。第一个指定的成员为最高位，后续的成员以降序排列。压缩结构体在声明的时候可以根据期望的算术行为，在 packed 关键字之后可以指定 signed 或 unsigned 关键字。缺省情况下，结构体是无符号的，例如：

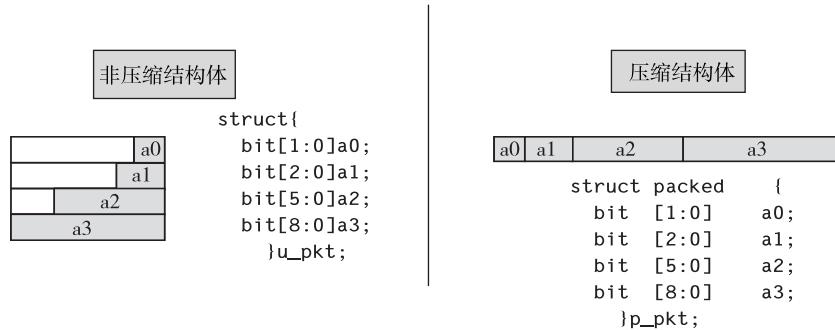


图 2-5 非压缩结构体与压缩结构体

```

struct packed signed {
    int a;
    shortint b;
    byte c;
    bit [7:0] d;
} pack1;// 有符号的 2 态值

struct packed unsigned {
    time a;
    integer b;
    logic [31:0] c;
} pack2;// 无符号的 4 态值

```

只要压缩结构体内的成员存在 4 态数据类型，那么整个结构体就被当作是 4 态数据类型，并且其中的任何 2 态成员都会使用强制类型转换进行转换。压缩结构体中的每一位数据可以使用 $[n-1:0]$ 的编号方法来访问，就如压缩数组一样。非整数数据类型（如 real 和 shortreal）以及非压缩数组不允许在压缩结构体或压缩联合体中使用。

联合体也可以被定义为压缩联合体，这时其内部所有成员的大小（位宽）必须一致。

2.1.6 常量

常量是一个永远都不会改变的命名数据变量，SystemVerilog 提供了三个 elaboration-time 常量（parameter、localparam、specparam）和一个 run-time 常量（const）。

通常，我们需要在程序中对同一个对象的不同的例化做定制（例如，存储器模块中的存储器的大小）。SystemVerilog 提供了 parameter、localparam 和 specparam 来指定参数常量（下面将 parameter 简称为参数）。这些参数可以在 SystemVerilog 中的 module、interface、program、class、package 结构中定义使用。每个参数必须在定义时给定初值；参数在 elaboration 的过程中设置（创建模块例化层次）并且保持该值直至整个程序执行结束。

参数可以定义成任何数据类型，例如：

```

parameter [data_type] param_name = default_value;
parameter string file_name = "testcase_1";

```

数据类型也可以定义为参数，这样使得 module、interface 或者 program 等可以定义一些数据类型参数化的端口和数据对象，在具体例化的时候才指定具体的数据类型。（注意，该

类参数不可以通过 defparam 来重写)，如源代码 2-8 所示。

源代码 2-8 类型参数化实例

```
//Chapter2 para_type_example.sv
module ma #(parameter p1 = 1, parameter type p2 = shortint )
  (input logic [p1:0] i, output logic [p1:0] o);
  p2 j=0;           // j 的类型通过参数来设置(若没有重定义,默认为 shortint)
  always @ (i) begin
    o = i;
    j++;
  end
endmodule
module mb;
  logic [3:0] i,o;
  ma#(.p1(3),.p2(int)) u1(i,o); //将 p2 重定义为 int 类型
endmodule
```

一个整型的参数可以使用“\$”赋值，例如 \$ 代表没有上限的边界，其最大值可以是任意整数，下面是一个例子：

```
parameter r2 = $;
property inq1(r1,r2);
  @ (posedge clk) a ##[r1:r2] b ##1 c |= > d;
endproperty
assert inq1(3);
```

本地参数（localparam）和参数类似，但是本地参数不能直接通过 defparam 和例化时修改。本地参数可以被带有参数的常量表达式赋值。本地参数有其特定的作用，可以在 generate 模块、package 和 class 内部定义使用。

const 常量与本地参数类似，但 const 常量在 elaboration 后才能确定。通过 const 关键字定义的静态常量可以通过文本表达式、参数、本地参数、genvar、枚举名等赋值，层次化引用也允许使用，因为 const 常量在 elaboration 之后才被确定下来，例如：

```
const logic option = a. b. c ;
```

通过 const 关键字定义的自动化常量可以通过任何不带 const 的合法表达式来赋值。一个类的例化（也就是类的句柄）也可以通过 const 来定义，例如：

```
const class_name object = new(5,3);
```

此时，类的对象（句柄）是不可重写的，注意，new 这个构造函数的参数必须是常量表达式；类的内部成员，除了通过 const 定义的以外，都可重写。

specparam 延时参数只能在延时说明块（specify block）中出现，而且只能定义延时参数。

所有的四种参数类型都可以通过文本表达式来赋值，例如：

```
parameter depth_array=8;
localparam byte colon1 = ":";
```

```
specparam delay = 10 ;// specparams are used for specify blocks
const logic flag = 1 ;
```

SystemVerilog 提供了四种方法来设置参数的值。每一个参数在声明时必须指定一个缺省值。我们可以使用下列方法之一来改变每一个模块实例的缺省参数值：

- 按顺序赋值（例如：`foo # (value, value) u1 (...);`）。
- 显式引用名字赋值（例如：`# (. name (value), . name (value)) u1 (...);`）。
- `defparam` 语句，使用层次化引用来重新定义每一个参数（这不是推荐的使用方法）。

2.1.7 文本表示

1. 数值表示

整数文本和逻辑文本可以是带有位宽的，也可以是不带位宽的。对于值的符号、截短和向左扩展，SystemVerilog 与 Verilog - 2001 遵守相同的规则。

指定位宽：

```
4'b1001      //固定长度的 4 比特的二进制表示
16'hdead     //固定长度的 16 比特的十六进制表示
```

不指定位宽：

```
4356         // 整型
'b0101       //二进制
```

SystemVerilog 可以使用撇号（'）作为前缀，来说明一个文本的所有位可使用相同的数值来填充，如'0、'1、'z 或 'x。此时，在撇号（'）与值之间没有基数限定符。这使得任何长度的向量不用显式地指定位宽就可以整体赋值，例如：

```
bit [63:0] data;
data = '1;      //设置数据的所有位为 1
```

2. 字符串表示

字符串文本由引号（“”）包围并且拥有自己的数据类型。对字符串文本的长度没有预定义的限制，一个字符串文本必须存在一个单行中，除非新的行紧跟着一个反斜杠（\）。在这种情况下，反斜杠和新行字符会被忽略。非打印字符或其他特殊字符都使用一个反斜杠（\），也就是转义符作为开始。一个字符串文本可以赋值给一个字符数组，就如 C 语言的用法一样在末尾会插入 null 结束符，若字符串长度与数组的大小不一致，则向左调整。例如：

```
char foo [0:12] = "hello world\n";
```

SystemVerilog 加入了表 2-3 中的特殊字符串字符。

表 2-3 特殊字符串及其描述

转义字符串	描述（产生的字符）
\n	换行符
\t	Tab 键
\\	符号 \
\ "	符号 "
\ v	垂直 Tab 键
\ f	换页
\ a	震铃
\ ddd	1 ~ 3 位八进制数所对应的 ASCII 码
\ xdd	1 ~ 2 位的十六进制数所对应的 ASCII 码

3. 结构体表示

结构体文本是结构体通过常量成员表达式的赋值方式或者表达式。结构体文本在语法上类似于 C 语言的初始化设置。结构体文本必须具有一个类型，无论是通过上下文关联还是通过强制类型转换。例如：

```
typedef struct {int a;shortreal b;} ab;  
ab c;  
c = '{0, 0.0};
```

嵌套花括号能够反映结构体的结构。下面是一个结构体数组被初始化的例子：

```
ab abarr[1:0] = '{'{1, 1, 0}, '{2, 2, 0}};
```

注意，C语言中采用的' {1, 1.0, 2, 2.0} 在SystemVerilog语言中是不允许的。结构体文本还可以使用成员名和值，或者是数据类型和缺省值，例如：

```
c = '{a:0, b:0.0}';//成员名及成员名的值  
c = '{default:0}';// 结构体 c 的所有元素均被设置成 0  
d = ab'{ int :1, shortreal :1.0};//数据类型以及这种类型的所有成员的缺省值
```

复制操作符“`{n}{{}}`”可以用来为完全相同数目的成员设置值，复制表达式中的内层括号对会被移除。例如：

```
struct {int X,Y,Z;} XYZ = '{3{1}}';//相当于'{1,1,1}'
```

4. 数组表示

数组文本在语法上与 C 语言的初始化设置类似，但数组文本允许使用复制操作符 (`{|n|}`)。例如：

```
int n[1:2][1:3] = {{ {0,1,2}, {3,4} }};
```

与 C 语言不同的是，括号的嵌套必须符合数组的维数。我们也可以嵌套复制操作符。复制嵌套中的内部括号对会被移除，复制表达式仅仅在一维空间上使用。例如：

```
int n[1:2][1:6] = '{2{'3{4, 5}}}; // 相当于 '{'{4,5,4,5,4,5},'{4,5,4,5,4,5}}
```

数组文本还可以使用它们的索引或类型作为操作数，并且可以使用一个缺省值。例如：

```
triple b = '{1:1, default :0}; // indices 2 and 3 assigned 0
```

5. 时间表示

时间文本使用整数或定点格式的数紧跟着一个时间单位来表示（fs、ps、ns、us、ms、s），在时间单位和数之间没有空格。例如：

2.1 ns

40 ps

时间文本被解释成 realtime 类型的值，按照当前的时间单位按比例缩放，并根据当前的时间精度四舍五入。

6. 注释方式

SystemVerilog 有两种方式的注释。单行注释以 “//” 开始并在新一行前结束。模块注释以 “/*” 作为开始并以 “*/” 作为结束。模块注释不可以嵌套，在模块注释中，单行注释符 “//” 不能有其他特殊含义。

2.1.8 操作符和表达式

SystemVerilog 支持以下三种操作符类型。

- 一元操作符 一个操作符在一个操作数前：

`a = ~b; // ~ 是一元操作符`

- 二元操作符 一个操作符在两个操作数之间：

`a = b || c; // “||” 是二元操作符`

- 三元操作符 一组操作符把三个操作数间隔开来：

`a = b ? c : d; // “?:” 是三元操作符`

SystemVerilog 提供了来自 Verilog 和 C 的操作符，如表 2-4 所示。

比较两个表达式的数值，从而确定指定的条件是否成立，有很多等价操作符可供选择，表 2-5 是不同操作符的一个比较。

逻辑比较操作逐位比较两个操作数。若任何一个操作数中含有 x 或者 z，它会返回一个 x；条件比较操作能够对操作数中的 x 和 z 也做一对一的比较，而通配比较操作能够将操作数中的任何 x 或者 z 作为通配，忽略该位上的比较操作。

表 2-4 操作符列表

	操作描述	语法	提示		操作描述	语法	提示
逻辑操作	逻辑非	$!E$		算术操作	取负	$- E$	
	逻辑与	$E_1 \&& E_2$			加	$E_1 + E_2$	
	相等	$E_1 == E_2$			减	$E_1 - E_2$	
	不相等	$E_1 != E_2$			乘	$E_1 * E_2$	
	全等	$I_1 === I_2$			除	E_1 / E_2	
	非全等	$I_1 !=!= I_2$			取模	$S_1 \% S_2$	
	通配等于	$I_1 ==? I_2$	sv		幂操作	$E_1 ** E_2$	
	通配不等于	$I_1 !=? I_2$	sv		自加	$++ J$	
位操作	取反	$\sim I$			自减	$-- J$	
	与操作	$I_1 \& I_2$		移位操作	无符号左移	$I_1 << I_2$	
	或操作	$I_1 + I_2$			无符号右移	$I_1 >> I_2$	
	异或操作	$I_1 \hat{+} I_2$			有符号左移	$S <<< I$	sv
	异或非操作	$I_1 \sim \hat{+} I_2$ $I_1 \hat{+} \sim I_2$			有符号右移	$S >>> I$	sv
归约操作	与操作	$\&I$		拼接	非重复拼接	$\{ I_1, \dots, I_n \}$	
	与非操作	$\sim \&I$			重复拼接	$\{ C \{ I_1, \dots, I_n \} \}$	
	或操作	$ I$			单个成员	$[C]$	
	或非操作	$\sim I$			固定范围	$[C_1 : C_2]$	
	异或操作	$\hat{+} I$			动态向右	$[I + : C]$	sv
	异或非操作	$\sim \hat{+} I$ $\hat{+} \sim I$			动态向左	$[I - : C]$	sv
关系操作	小于	$E_1 < E_2$		赋值操作	条件选择	$I? E_1 : E_2$	
	小于等于	$E_1 <= E_2$			阻塞赋值	$V = E$	
	大于	$E_1 > E_2$			非阻塞赋值	$V <= E$	
	大于等于	$E_1 >= E_2$			操作符	$+ =, - =, * =, / =, \% =, \& =, =, ^ =, <=, >=, <<=, >>=$	

I: 无符号整型的表达式

J: 整型变量

S: 有符号整型的表达式

V: 任意类型的变量

R: 实数类型的表达式

C: 无符号整型常数

E: I、S、R 中的一种

SV: SystemVerilog 中的新增操作符

表 2-5 比较操作符列表

	相等			不相等		
	相等 $a == b$	全等 $a === b$	通配全等 $a == ? b$	不相等 $a != b$	非全等 $a != b$	非通配全等 $a !=? b$
$a = 4'b0001$ $b = 4'b0001$	1	1	1	0	0	0
$a = 4'b0101$ $b = 4'bxx0x$	x	0	1	x	1	0
$a = 4'bxx0x$ $b = 4'bxx1x$	x	0	0	x	1	1
$a = 4'bxx0x$ $b = 4'b1x0x$	x	0	1	x	1	0
$a = 4'bxx0x$ $b = 4'b1xxx$	x	0	1	x	1	0
$a = 4'b0000$ $b = 4'b000z$	x	0	1	x	1	0

2.2 过程语句

一个过程模块有自己的执行进程，其内部的语句将顺序执行，就如 C 和 C++ 程序一样。过程模块由过程语句组成。SystemVerilog 提供了下列几种过程语句可以在过程模块中使用：

- 赋值语句
- 条件选择语句
- 循环语句
- 跳转语句
- 子程序调用
- 事件控制

下面我们将分节讨论介绍。

2.2.1 赋值语句

赋值语句用来对程序中的变量进行赋值。SystemVerilog 提供了下面几种赋值语句：

- 阻塞赋值
- 非阻塞赋值
- 自加/自减赋值
- 过程连续赋值语句

赋值语句实例如源代码 2-9 所示。

源代码 2-9 赋值语句实例

```
//Chapter2 assign_example.sv
module assign_example;
  wire [9:0] net_data;
  initial begin
    logic [9:0] data [10:0];
    logic [9:0] var_data;

    data[0] <= 'x;           //非阻塞赋值
    data[1] = 'z;          //阻塞赋值

    data[2] = 7;
    data[2] += ;
    data[2] -= ;

    assign var_data = data[2] //连续赋值语句将 7 赋值给 var_data
    var_data = 4;           //不会改变 var_data 的数值
    deassign var_data;      //去除连续赋值的作用
    var_data = 5;           //var_data 的值改变了

    force var_data = data[1];
    release var_data;

    force net_data = data[1];
    release net_data;
  end
endmodule
```

第七行和第八行为非阻塞赋值和阻塞赋值。这两类语句的区别和使用方法在 Verilog 的各种语法书中有详细的分析和介绍，在此不再赘述。SystemVerilog 引入了自加和自减操作符，允许变量自身可以将当前值做递增或递减。

过程连续赋值语句，包括对变量和网线的 assign、deassign 和 force、release。

2.2.2 控制结构

下面是 SystemVerilog 中支持的条件选择、循环、跳转等编程控制结构，其使用方法与 Verilog 和 C 类似。

1. 条件选择语句

(1) if...else 语句

if...else 语句根据不同的条件执行不同的分支。

```
if(expression)
  <begin> ... <end>
else
  <begin> ... <end>
```

if 条件选择语句实例如源代码 2-10 所示。

源代码 2-10 if 条件选择语句实例

```
//Chapter2 if_example.sv
module if_example(q,d,clear,clk);
    output q;
    input d,clear,clk;
    reg q;
    always @ (posedge clk)
        if (! clear) q <= 0;
        else         q <= d;
endmodule
```

(2) case 语句

case 语句为程序提供了分支选择控制的功能。case 要求分支表达式和 case 条件表达式做全等比较（`==`）而不是逻辑比较（`==`）。一个分支只有在其表达式完全匹配 case 条件表达式时，才被选中执行。

```
case(expression)
    constant_expression: < statement_block >;
    constant_expression: < statement_block >;
    ...
    default: < statement_block >;
endcase
```

case 条件选择语句实例如源代码 2-11 所示。

源代码 2-11 case 条件选择语句实例

```
//Chapter2 case_example.sv
module case_example(a);
    input a;
    always @ (a)
        case (a)
            1'b1: $display("input signal value is 1");
            1'b0: $display("input signal value is 0");
            1'bx: $display("input signal is in unknown state");
            1'bz: $display("input signal is in high impedance state");
        endcase
endmodule
```

SystemVerilog 也提供了两个有更多变化的 case 结构：casex 和 casez。

casex：case 条件表达式中所有的 x 值都不参与比较。

casez：case 条件表达式中所有的 x 和 z 值都不参与比较。

2. 循环语句

Verilog 提供了 for、while、repeat 以及 forever 循环。SystemVerilog 增强了 Verilog 的 for 循环，并加入了一个 do...while 循环和一个 foreach 循环。

(1) for 循环语句

for 循环语句可以实现条件循环，基本格式如下：

```

for( < initializing_expression > ; < terminating
expression > ; < loop_increment_expression > )
    < begin >
        //循环体
    < end >

```

在 Verilog 中，用来控制 for 循环的变量必须在循环体之前声明。如果两个或多个并行程序中的循环使用相同的循环控制变量，那么就有可能出现一个循环修改其他循环还在使用的循环控制变量的情况。在 for 循环中，SystemVerilog 添加了声明 for 循环控制变量的能力。这种方式会在循环内产生一个本地变量，其他并行循环不会偶然地影响这个循环控制变量。for 循环语句实例如源代码 2-12 所示。

源代码 2-12 for 循环语句实例

```

//Chapter2 for_example.sv
module for_example;
    initial begin
        loop1:for (int i = 0;i <= 255;i++)
            $display("loop1 i is %d",i);
            #1;
    end
    initial begin
        loop2:for (int i = 15;i >= 0;i--)
            $display("loop2 i is %d",i);
            #1;
    end
endmodule

```

在 for 循环内部声明的本地变量相当于在一个未命名的块中声明了一个自动变量，上述代码等价于：

```

module foo;
    initial begin
        begin
            automatic int i;
            for (i = 0;i <= 255;i++)
                ...
        end
        end
        initial begin
            begin :loop2
                automatic int i;
                for (i = 15;i >= 0;i--)
                    ...
            end
        end
    endmodule

```

Verilog 仅允许单个初始化语句以及在 for 循环中的单个步值赋值语句。SystemVerilog 允

许多初始声明或赋值语句可以是一个或多个用逗号分隔的语句。步值赋值也可以是一条或多条用逗号分隔的语句。例如：

```
for (int count = 0, done = 0, j = 0; j * count < 125; j++, count++)
    $display("Value j=%d\n", j);
```

(2) while 循环语句

while 循环语句执行循环体，直至条件表达式为假。如果一开始条件表达式就为假，那么循环语句永远都不会被执行。

```
while(<expression>)
    <begin>
        //循环体
    <end>
```

下面例子计算了向量 tempreg 中 1 的个数，代码如源代码 2-13 所示。

源代码 2-13 while 循环语句实例

```
//Chapter2 while_example.sv
module while_example();
    initial
        begin : count1s
            logic [7:0] tempreg;
            count = 0;
            tempreg = 8'b11110000;
            while (tempreg) begin
                if (tempreg[0])
                    count++;
                tempreg >>= 1;
            end
        end
    endmodule
```

(3) do…while 循环语句

do…while 循环语句和 while 循环语句有一点区别，也就是 do…while 循环语句在循环体的结束处评估循环条件，也就是 do…while 循环至少执行一次循环体。

```
do
    <begin>
        //循环体
    <end>
    while(<expression>);
```

使用 do…while 逐位打印字符串的实例如源代码 2-14 所示。

源代码 2-14 do…while 循环语句实例

```
//Chapter2 dowhile_example.sv
module dowhile_example();
```

```

int map[string];
map ["hello"]=1;
map["sad"] = 2;
map["world"] = 3;
string s;
initial begin
s = "hello!";
if (map.first( s ) )
do
$display( "%s : %d\n", s, map[ s ] );
while ( map.next( s ) );
end
endmodule

```

(4) repeat 循环语句

repeat 循环对循环体执行固定的次数。如果表达式被评估为未知或者高阻，那么应该认为是零次，不应执行循环体。

```

repeat( < expression > )
< begin >
//循环体
< end >

```

源代码 2-15 是一个 repeat 循环的例子，通过加法和移位实现了一个乘法器。

源代码 2-15 repeat 循环语句实例

```

//Chapter2 repeat_example.sv
module repeat_example();
parameter size=8, longsize=16;
logic [size:1] opa, opb;
input [size:1] opa, opb;
logic [longsize:1] result;

initial begin :mult
logic [longsize:1] shift_opa, shift_opb;
shift_opa = opa;
shift_opb = opb;
result = 0;
repeat (size) begin
if (shift_opb[1])
result = result + shift_opa;
shift_opa = shift_opa << 1;
shift_opb = shift_opb >> 1;
end
end
endmodule

```

(5) forever 循环语句

forever 循环语句持续执行一个循环体。

```

forever
  <begin>
    //循环体
  <end>

```

为了防止循环体在 delta 时间内产生无限循环，导致仿真挂起，`forever` 循环体内部必须带有时序控制或者 `disble` 语句。源代码 2-16 是一个利用 `forever` 循环生成周期性时钟的例子，时钟 `clock1` 周期为 20ns，时钟 `clock2` 周期为 10ns。

源代码 2-16 forever 循环语句实例

```

//Chapter2 forever_example.sv
module forever_example();
  logic clock1,clock2;
  initial begin
    clock1 <=0;
    clock2 <=0;
    fork
      forever #10 ns clock1 = ~clock1;
      #5 forever #5 ns clock2 = ~clock2;
    join
  end
endmodule

```

(6) foreach 循环语句

`foreach` 循环语句中指定数组后，程序会逐个遍历数组成员。

```
foreach( < array name > [ < loop variables > ] ) < statement >
```

它的自变量可以是一个指定的任意类型数组（固定尺寸的、动态的及联合数组），然后紧跟着一个包围在方括号内的循环变量的列表。每一个循环变量对应于数组的某一维。`foreach` 结构类似于一个 `repeat` 循环，它使用数组范围替代一个表达式来指定重复次数。`foreach` 循环语句实例如源代码 2-17 所示。

源代码 2-17 foreach 循环语句实例

```

//Chapter2 foreach_example.sv
module foreach_example();
  string words [2] = {"hello", "world"};
  int prod [1:8] [1:3];
  initial begin
    foreach ( words [ j ] )
      $display( j , words[j] );           // 打印每个索引和值
    foreach ( prod[ k, m ] )
      prod[k][m] = k * m;               // 初始化
    end
  endmodule

```

循环变量的数目必须匹配数组变量的维数。空循环变量可以用来表示在对应的数组维

数上没有迭代，并且处于尾部的连续空循环变量可以被忽略。循环变量是自动的、只读的，并且它们的作用范围对于循环来讲是本地的。每一个循环变量的类型被隐含地声明成与数组索引的类型一致。

3. 跳转语句

SystemVerilog 增加了 C 语言中的跳转语句：break、continue 和 return。

- break 像 C 语言一样跳出本层循环体。
- continue 像 C 语言一样跳转到本次循环的尾部。
- return (expression) 退出一个函数并返回函数值。
- return 退出一个任务或 void 函数。

continue 和 break 只能使用在循环体中。break 语句会中断最近一层循环并跳出此层循环；continue 语句结束本次循环，跳转到循环体的尾部，如果尾部存在循环控制语句，就执行这个循环控制语句。continue 和 break 语句不能在 fork...join 语句中用来控制位于 fork...join 块之外的循环。例如：

```
initial begin
    logic [127:0] text;
    integer file,result;
    file = $fopen("string.txt","r");
    while (! $feof(file))begin
        result = $fscanf(file,"%s",text);
        case (text)
            "" : continue;           //空操作,跳出本次循环进入下一次循环
            "done":break;           //跳出整个循环体,结束 while 循环
            //其他程序
            ...
    endcase
end
end
```

return 语句只能使用在一个任务或函数当中。在有返回值的函数中，return 必须具有一个正确类型的表达式。SystemVerilog 没有包含 C 语言中的 goto 语句。

2.3 函数和任务

SystemVerilog 提供了两种子程序调用（通称方法）：函数和任务。

2.3.1 函数和任务的区别

函数和任务的区别如下：

- 1) 它们拥有各自的命名空间，也就是可以存在两个名字完全相同而实现完全不一样的函数和任务。当然，我们应该避免这种情况出现，因为这会导致代码难以阅读和维护。
- 2) 任务没有返回值，而函数可以有返回值。
- 3) 任务可以带时序（timing consuming）语句，如#5ns，而函数不可以，也就是任务的

执行可以跨越时间 (advance time)，而函数不可以。

任务和函数可以用来实现复杂的操作和计算。任务的执行可以消耗仿真时间，也就是时间控制语句可以在内部使用（如：#、##、@、fork、wait、wait_order 或 expect）。函数不能带延时，如#100，阻塞语句如@ (posedge clock) 或者 wait (ready)，或者调用任务。

2.3.2 子程序定义

SystemVerilog 对任务和函数做了很多改进，使之更接近于 C 和 C++。

其中，最大的改进是在 SystemVerilog 中定义的子程序，begin…end 语句是可选的；但是在 Verilog1995 中，除了单个语句外，这是必须的。为此，在 SystemVerilog 中多条语句可以在 task/endtask 和 function/endfunction 之间使用，这些语句顺序地执行，可以不用 begin…end。任务和函数中没有任何执行语句也是合法的。

表 2-6 子程序定义

Verilog1995 coding style	SystemVerilog coding style
<pre>task multiple_lines; begin \$display("First line:1"); \$display("Second line:2"); end endtask</pre>	<pre>task multiple_lines; <begin> //optional \$display("First line:1"); \$display("Second line:2"); <end> //optional endtask</pre>

2.3.3 子程序参数

SystemVerilog 和 Verilog – 2001 语法能够更清楚和简练的定义子程序的参数。下面是一个例子，在 Verilog 中要求定义某些参数两次，一次是方向，一次是类型。例如：

```
//Verilog – 1995 coding style
task mytask;
  output [31:0] x;
  reg [31:0] x;
  input y;
  ...
endtask
```

在 SystemVerilog 中，任务/函数声明的形式参数除了像上面那样在任务/函数体内定义外，还可以像 ASCII-C 那样定义在圆括号中（参数缺省方向是 input，缺省类型是 logic），例如：

```
task mytask1(output int x, input logic y);
  ...
endtask
```

每一个形式参数可以选择下列方向属性之一：

input 在开始的时候复制值。

`output` 在结束的时候复制值。

`inout` 在开始的时候复制，在结束的时候输出。

`ref` 传递引用（句柄或者指针）。

在 SystemVerilog 中，如果没有指定参数的方向，那么它的缺省方向是输入。一旦指定了一个方向，那么它就成为后续参数的缺省方向。在下面的例子中，形式参数 `a` 和 `b` 缺省为输入，`u` 和 `v` 都是输出。

```
task mytask2(a, b, output bit [15:0] u, v);
  ...
endtask
```

每一个形式参数都有其自身的数据类型，可以是显式声明或者是缺省类型。在 SystemVerilog 中，任务/函数的参数缺省类型为 `logic`，上面 `mytask2` 中的 `a` 和 `b` 都是 `logic`。

Verilog 对于参数只有传值的方式：在子程序调用的时候，类型为 `input` 或者 `inout` 的实际参数的数值被复制到本地变量，在子程序退出的时候，类型为 `output` 或者 `inout` 的实际参数的数值被更新。

SystemVerilog 提供了两种方式来为函数和任务传递参数：值传递和引用传递。所谓引用传递（reference）就如 C++ 的句柄或者指针，也就是变量的入口地址。此时，参数的类型应定义为 `ref`，相对于 `input`、`output` 和 `inout`，`ref` 有其独特的地方。

首先，可以通过引用将数组传递到子程序中：

```
function void ary_sum(const ref int ary[]);
  int sum=0;
  for (int i=0;i<ary.size();i++)
    sum+=ary[i];
  $display("The sum of the arrays is % 0d",sum);
endfunction
```

当然，SystemVerilog 也允许通过传值的方式来传递数组参数，这样数组将被整体复制，这将消耗一定的内存和操作时间。而使用 `ref` 传递，只是获取该数组的入口地址（句柄/指针），操作速度快，减少内存使用。当然，在这个例子里面，我们使用到了 `const`。`const` 关键字可以防止一个函数或者任务改变一个通过 `ref` 类型传递的变量。这对于大型的数据结构通过 `ref` 传递，避免了整个结构被复制，而且在函数内部也不会被改变。一旦使用了这个 `const ref`，编译器就会检查子程序是否修改了该数组。

第二个好处是，在子程序修改 `ref` 参数变量的时候，其变化对于外部是立即可见的。这是很有用的，特别在几个进程并行地执行而想通过一个简单的方式来传递信息，源代码 2-18 是 `ref` 引用端口类型的例子，我们将在后面详细讨论这种应用。

源代码 2-18 `ref` 引用端口类型实例

```
//Chapter2 ref_example.sv
module ref_example();
task bus_read(input logic [31:0] addr,ref logic [31:0] data);
//request bus and drive address
bus.request=1'b1;
```

```

@ (posedge bus.grant) bus.addr = addr;
//wait for data from memory
@ (posedge bus.enable) data = bus.data;
//release bus and wait for grant
bus.request = 1'b0;
@ (negedge bus.grant);
endtask

logic [31:0] addr,data;
initial
  fork
    bus_read(addr,data);
  begin
    @ data;
    $display("data is % Od",data);
  end
  join
endmodule

```

参数可以通过名字或者位置来传递。任务和函数的参数还可以指定缺省值，这就使得调用任务或函数的时候可以不用传递参数。这是一种很有用的方法，在某些情况下，当验证平台开发到一定程度的时候，我们需要对某些验证组件的任务或者函数进行修改或者添加功能，有可能要增加接口参数，为了不影响验证平台其他地方对该任务或者函数的使用，我们需要对新添加的参数指定默认值。指定参数缺省值对于验证平台的重用有很大的帮助。例如：

```

function void print_sum(ref int a[]);
...
endfunction

function void calculate();
...
  print_sum(a);
...
endfunction

initial
  calculate();

```

若在后来需要修改 print_sum 函数添加参数，指定数组的开始地址和结束地址：

```

function void print_sum(ref int a[],input int start,input int last);
...

```

那么，原程序中的任何一个调用到 print_sum () 的代码，就需要改成：print_sum (a1,b1,c1);否则缺少了参数，程序将无法正常运行。这种改动将是很麻烦的。而若采用参数默认值的方法，则无需做任何改动：

```

function void print_sum(ref int a[],input int start=0,input int last=a.size()-1);
...

```

2.3.4 子程序返回

在 Verilog 中，当任务/函数运行到 endtask/endfunction 的时候退出。而在 SystemVerilog 中，可以使用 return 语句在任务体或函数体的任意一点退出子程序。

在 Verilog 中，函数必须具有返回值。返回值是通过对函数名赋值来完成的。

```
function [15:0] myfunc1(input [7:0] x,y);
    myfunc1 = x * y - 1; // 返回值赋值给函数名字
endfunction
```

SystemVerilog 允许将函数声明成 void 类型，它没有返回值。此外，其他函数像 verilog 一样，可以通过为函数名字赋值来返回一个值，或者使用 return 语句实现。

```
function [15:0] myfunc2(input [7:0] x,y);
    return x * y - 1; // 使用 return 语句指定返回值
endfunction
```

在 SystemVerilog 中，函数可以返回一个结构体或联合体。在这种情况下，函数内部可以使用以函数名为起始的层次化名，其被解释为返回值的一个成员。如果函数名在函数外使用，其表示整个函数的作用范围。如果函数名在一个层次化当中使用，也表示整个函数的作用范围。

void 类型的函数可以作为单个语句调用，其他情况下，函数都以表达式调用：

```
a = b + myfunc1(c, d);      // 将 myfunc1(在上面定义)作为一个表达式调用
myprint(a);                  // 将 myprint(在下面定义)作为一条语句调用
function void myprint(int a);
...
endfunction
```

在 Verilog – 2001 中，函数返回的值必须使用在赋值语句或使用在一个表达式中。在单个语句中调用非 void 类型的函数会导致一个警告信息，仿真器会将函数返回值强制转换成 void 类型，SystemVerilog 允许使用 void 数据类型来忽略一个函数的返回值，如下所示：

```
void '(some_function());
```

2.3.5 自动存储

当 Verilog 在 20 世纪 80 年代出现的时候，它主要的目的是用来描述硬件。为此，所有的对象都是静态分配的。特别是子程序参数和本地变量都是通过固定的位置存储的，而不是像其他程序那样通过堆栈的方式。那么，如何才可以描述一个有迭代子程序的硬件行为呢？熟悉基于堆栈，诸如 C 或者 C++ 语言的软件工程师对此感到束手无策，上述缺点大大限制了他们创建复杂验证平台的能力。

在 Verilog – 1995，如果在多个地方调用同一个任务，本地变量是共同而且静态分配的，为此，不同的进程相互访问同一个值。在 Verilog – 2001 中，可以通过使用 automatic 关键字，将任务、函数和模块声明为自动存储模式，这样，仿真器就能够对其所有形式的参数和内部

变量使用堆栈的形式来存储。automatic 使函数和任务的重入（reentry）成为可能，如下所示：

```
function automatic [63:0] factorial;
  input [31:0] n;
  if (n == 1)
    factorial = 1;
  else
    factorial = n * factorial(n - 1);
endfunction
```

在 SystemVerilog 中，子程序仍然默认使用静态存储方式，对于所有的模块（module block）和程序块（program block）也一样；对于声明在 class 中的子程序和变量默认是动态存储的。另外，SystemVerilog 还允许在一个静态任务中将特定的形式参数和本地变量声明成自动（automatic）的，也允许在一个自动任务内将特定的形式参数和本地变量声明成静态（static）的，后面我们将深入讨论。

2.4 编程结构

SystemVerilog 程序层次包括模块（module block）、程序块（program block）、过程块和数据对象 4 个部分。

模块是用来为层次结构的设计建模的。在基于模块的验证环境中，模块也可以用来搭建层次化的验证平台。模块可以包含其他模块的例化、过程化语句和数据对象。程序块也用来定义一个验证平台和被测设计的边界。程序块可以有类型和变量定义或者一个或者多个 initial 模块。过程化块可以包含其他过程化块或者数据对象。数据对象，指的不仅仅是前面提到的数据类型，还包括后面将讨论到的类（class）、接口（interface）、mailbox 等。下面我们将对 SystemVerilog 中的主要编程结构做简要的介绍，并着重介绍如何封装一个验证程序和设计，以及如何实现两者的连接和通信。

2.4.1 模块

一个模块（module）的简单定义格式如下：

```
module module_name <(module_port_list> ;
  <declaration_block;> // (include:port_list, parameter, data object)
  <function_task_declaration>
  <module_instance;>
  <procedure_block> // include(continuous assign, procedural block)
  ...
endmodule
```

在 Verilog 中，模块可以用来描述从简单的门元件到复杂的系统（例如一个微处理器）的任何一种硬件电路。一个复杂电路在使用 Verilog 建模后表现为一个顶层模块（如源代码 2-19 中的 top），该顶层模块的输入输出端口分别代表着电路的输入输出端；顶层模块可以由若干个子模块构成（如源代码 2-19 中的 mem_core 和 cpu_core），每个子模块代表着整个电路内特定功能的一个功能单元，各个子模块通过子模块的端口相互连接起来；子模块又

可以被分成若干个第二级子模块，代表着对上一级电路功能模块单元的进一步的细分；这种细分一直可以进行到基本逻辑单元（门级元件或者开关级元件）一级。这样，整个硬件电路在经过 Verilog 电路建模后表现为分层次的、相互联系的一组模块。模块实例如源代码 2-19 所示。模块例化连接如图 2-6 所示。

源代码 2-19 模块实例

```
//Chapter2 module_example.sv
module mem_core(input logic wen, input logic ren, output logic mrdy = 1, input logic [7:0] addr, input logic [7:0] mem_din, output logic [7:0] mem_dout, output logic status, input logic clk);
    logic [7:0] mem [7:0];
    task reply_read(input logic [7:0] data, integer delay);
        #delay;
        @ (negedge clk);
        mrdy = 1'b0;
        mem_dout = data;
        @ (negedge clk);
        mrdy = 1'b1;
    endtask
    always @ (negedge ren) reply_read(mem[addr],10);
endmodule

module cpu_core(output logic wen = 1, output logic ren = 1, input logic mrdy, output logic [7:0] addr=0, input logic [7:0] cpu_din, output logic [7:0] cpu_dout, output logic status = 0, input logic clk);
    task read_memory(input logic [7:0] raddr, output logic [7:0] data);
        @ (posedge clk);
        ren = 1'b0;
        addr = raddr;
        @ (negedge mrdy);
        @ (posedge clk);
        data = cpu_din;
        ren = 1'b1;
    endtask
    initial begin
        logic [7:0] read_data;
        read_memory(8'b00010000,read_data);
        $display("Read Result", $time,read_data);
    end
endmodule

module top
    logic mrdy,wen,ren;
    logic [7:0] addr,d1,d2;
    wire status; logic clk=0;
mem_core mem(. * ,. mem_din(d1),. mem_dout(d2));
cpu_core cpu(. * ,cpu_din(d2),. cpu_dout(d1));
initial for (int i=0 ;i <=255 ;i++ ) #1 clk = ! clk;
endmodule
```

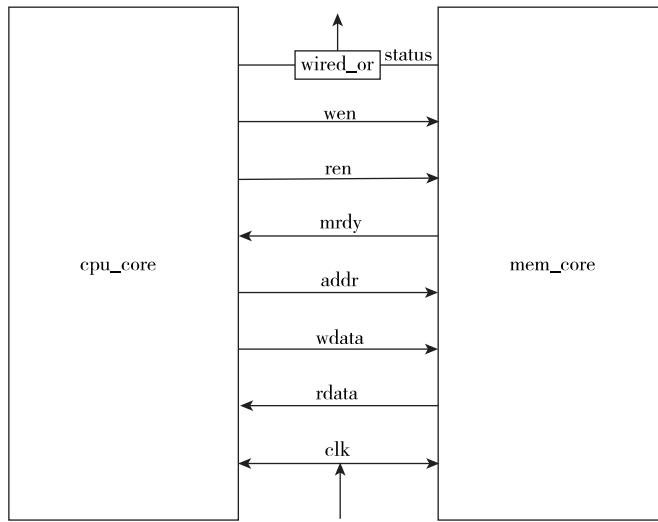


图 2-6 模块例化连接

如源代码 2-19 所示，模块例化存在三种方式：采用 * 对同名的信号做默认连接；端口名称关联；端口位置关联。

模块除了可以为硬件建模外，也可以用来封装验证平台，在模块内部，我们可以例化采用模块定义的 DUT 和采用 program 或者 class 封装的验证平台。

2.4.2 接口

在 SystemVerilog 中，在模块端口的声明中除了 input、output、inout 外，也像子程序那样增加了引用 (ref)。

另外，SystemVerilog 引入一个重要的数据类型：interface。其主要作用有两个：一是简化模块之间的连接；二是实现类和模块之间的通信；第二点我们后面再详细讨论。

接口 (interface) 为硬件模块的端口提供了一个标准化的封装方式。比如 PCI 或 ARM AHB 接口是标准的接口，而在不同的项目中，我们却可能要重复定义相同的接口。SystemVerilog 提供了一个新的接口描述方式，用 interface 来封装接口的信号和功能。interface 的定义是独立于模块的，通过关键字 interface 和 endinterface 包起来。此外，interface 里面还可以带时钟、断言、方法等定义。接口实例如源代码 2-20 所示，interface 简化模块的连接如图 2-7 所示。

源代码 2-20 接口实例

```

//Chapter2 interface_example.sv
interface membus(input logic clk,output wor status);
    logic mrdy;
    logic wen;
    logic ren;
    logic [7:0] addr ;
    logic [7:0] c2m_data;

```

```

logic [7:0] m2c_data;

task reply_read(input logic [7:0],integer delay);
    #delay;
    @ (negedge clk)
    mrdy = 1'b0;
    m2c_data = data;
    @ (negedge clk);
    mrdy = 1'b1;
endtask

task read_memory(input logic [7:0] raddr,output logic [7:0] data);
    @ (posedge clk);
    ren = 1'b0;
    addr = raddr;
    @ (negedge mrdy);
    @ (posedge clk);
    data = m2c_data;
    ren = 1'b1;
endtask

modport master(output wen,ren,addr,c2m_data,input mrdy,m2c_data);
modport slave(input wen,ren,addr,c2m_data,output mrdy,m2c_data);
endinterface

module mem_core(membus.slave mb);
    logic [7:0] mem [7:0];
    assign mb.status = 0;
    always @ (negedge mb.ren) mb.reply_read(mem[mb.addr],100);
endmodule

module cpu_core(membus.master mb);
    assign mb.status = 0;
    initial begin
        logic [7:0] read_data;
        mb.read_memory(7'b00010000,read_data);
        $display("Read Result", $time,read_data);
    end
endmodule

module top;
    wor status;
    logic clk=0;
    membusr mb(clk,status);

    mem_core mem(.mb(mb.slave));
    cpu_core cpu(.mb(mb.master));

    initial for (int i=0;i <= 255;i++) #1 clk = !clk;
endmodule

```

2.4.3 过程块和语句块

在模块中，除了连续赋值语句（assign）之外，其他主要由过程块组成。其中包括 initial 块、always 块的变型（always_comb, always_ff, always_latch）。SystemVerilog 还增加了 final 块。initial 块是在程序开始的时候运行，final 块是在程序结束前，所有执行进程结束后才执

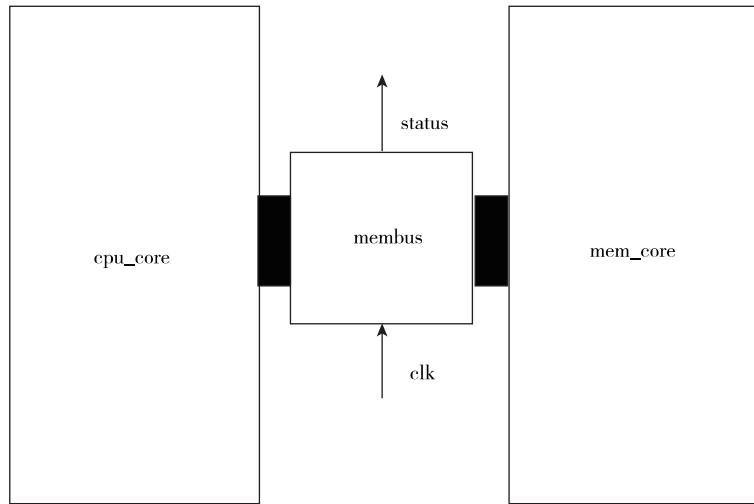


图 2-7 interface 简化模块的连接

行。always 块和其变形在其敏感条件满足的时候开始运行，在敏感条件不满足的时候被挂起。

过程块可以是串行执行或者并行执行；多条串行执行语句需要放置在 begin…end 串行块内。对于多条并行执行语句，可以使用 fork…join 并行块，SystemVerilog 还增加了两种变形 fork…join_any, fork…join_none（其使用方法我们在后面讨论）。串行块和并行块可以混合使用，如表 2-7 所示。

表 2-7 串行块与并行块

	initial	always @ (sensible list)
begin …end	<pre>initial begin d_out = 0; #1 d_out = 1; #2 d_out = 0; ... end</pre>	<pre>always @ (posedge clk) begin d_out <= reg_b; reg_b <= reg_a; reg_a <= d_in; end</pre>
fork…join	<pre>initial fork task_1 (); task_2 (); d_out = 1; ... join</pre>	<pre>always @ (negedge global_reset_n) fork reset_cpu (); reset_mem (); join</pre>
fork…join/begin…end	<pre>initial begin fork reset_n = 0; reset_cpu (); reset_mem (); join reset_n = 1; ... end</pre>	

(续)

	initial	always @ (sensible list)
begin…end/fork…join	<pre> initial begin fork begin reset_cpu (); reset_mem (); end reset_n = 0; join ... end </pre>	

2.4.4 数据对象

数据对象，除了普通的数据类型外，还可以是 interface、class 等。在 SystemVerilog 中，class 为搭建验证平台提供了很多高级的功能，如继承、随机、多态等。采用 class 搭建验证平台可以实现多种高级的验证技术和方法，如基于事务级的验证、随机激励生成、验证平台重用等。我们可以把事务处理器（transactor）、总线功能模型、参考模型、激励产生器等用 class 来封装。这都是推荐的使用方法，具体应用在后面讨论。

2.4.5 程序块

程序块（program）能够实现三个基本目标：

- 1) 它提供了验证程序执行的入口。
- 2) 它提供了一个封装验证程序数据、函数和任务的结构。
- 3) 它提供了一个语法来指定在 Reactive 区域中的调度执行。

程序块为设计和验证平台间建立了一个清晰的分隔；另外，它为程序块中声明的所有成员在 Reactive 区域中指定了特定的执行语义。与时钟控制块一起使用，程序块为设计和验证平台间提供了无竞争的交互，并实现了周期和事务级的抽象。

一个典型的程序块包含类型和数据声明、子程序、与设计的连接以及一个或多个过程化的代码。设计和验证平台间的连接使用相同的互连机制，采用端口（包括接口）。程序块的语法如下：

```
program test(input clk, input [16:1] addr, inout [7:0] data);
    initial ...
endprogram
```

或者

```
program test(interface device_ifc);
    initial ...
endprogram
```

程序块可以被看作是一个具有特殊执行语义的模块。一旦被声明，一个程序块可以在需要的层次位置（典型情况是顶层）中被实例化，并且它的端口可以像任何其他模块一样，

采用相同的方式做连接。

程序块可以嵌套在模块或接口内部。这就使得多个协同操作的程序块能够共享作用范围内的本地变量。没有端口的嵌套程序块，虽没有被显式实例化，当在顶层中其被隐含地实例化一次；被隐含地实例化的程序块具有相同的实例和声明名字。例如：

```
module test(...);
  int shared;      //变量 shared 被程序 P1 和 P2 共享
  program p1;
  ...
  endprogram
  program p2;
  ...
  endprogram      // p1 和 p2 隐式例化在模块 test 中
endmodule
```

一个程序块可以包含一个或多个 initial 块；但不能包含 always 块、UDP（用户自定义单元）、模块、接口或其他程序块的定义和例化。

程序块内的类型和数据声明对于程序块的作用范围来讲是本地的，并具有静态的生命周期。当然，我们也可以通过 automatic 将程序块声明为自动存储的。

程序块作为某公司推荐的语法，它主要的目的和使用方法如下。

- 1) 程序块是封装验证程序（特别是 class）的主要结构和入口。
- 2) 程序块可以消除验证平台和设计之间的竞争。
- 3) 程序块在 Reactive 区域调度执行。
- 4) 在程序块中要采用非阻塞赋值语句（<=）对时钟块（clocking_block）中的信号做驱动。
- 5) 在程序块中的本地变量进行赋值要采用阻塞赋值语句（=）。
- 6) 程序块中可以有任务、函数、类和 initial 块，但不能有 always 块。
- 7) 对封装验证程序的程序块采用 automatic 来定义。

另外，有的公司则同时支持在 module 和 program 封装验证程序（特别是 class）。为此，在验证程序的封装上存在两种方式：

- 将基于类的验证程序封装在程序块中，以程序块作为和设计的分隔处。
- 将基于类的验证程序封装在模块中。

2.4.6 简单的验证架构

如上面讨论到的，有的公司推荐将以 class 为单元的验证程序封装到程序块中，并以此作为验证平台和被测设计的分界点。下面是采用程序块的一种简单的验证架构。验证程序被封装在类 testbench_class 中，而类只能在程序块（program_test）中例化使用。最后，程序块和被测设计（design_under_test）在顶层（top）例化连接。程序块和被测设计通过端口或者接口作为通信，如图 2-8 所示。

另外，有的公司支持在模块（module）中例化类，如图 2-9 所示。为此我们可以采用模块来例化验证程序（testbench_class），并通过使用接口来实现类和被测设计之间的通信。后面我们将详细讨论类是如何通过接口和被测设计实现数据交互的。

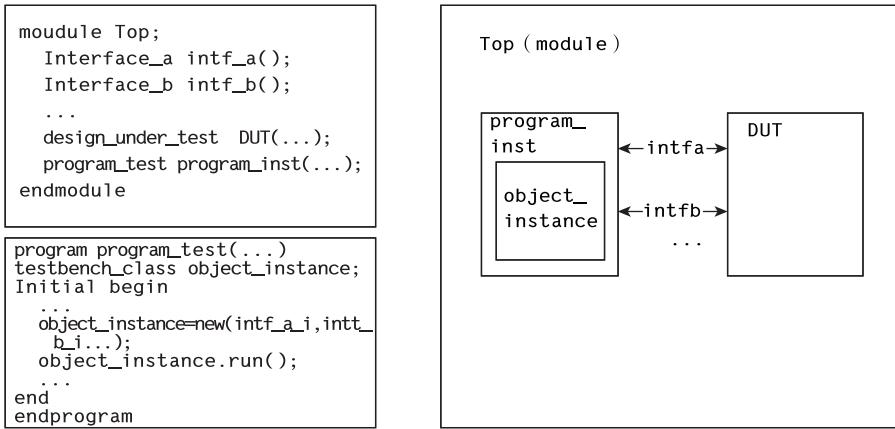


图 2-8 采用 program 程序块封装验证程序

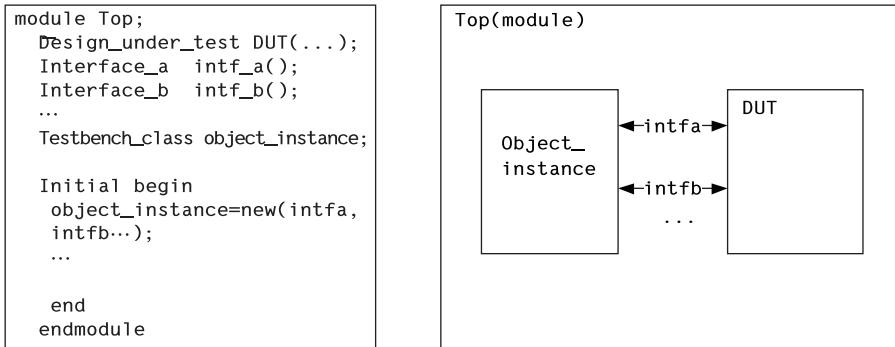


图 2-9 在 module 模块中例化验证程序

2.5 数据的生命周期和作用域

SystemVerilog 定义了数据的静态和动态这两种生命周期，说明如下。

一个动态变量在程序执行的时候在其定义作用范围创建生成。例如一个在函数内部定义的动态变量在程序执行进入该函数的时候创建，在程序执行返回退出该函数的时候被释放删除。静态变量在程序开始执行的时候被创建并且保持它数值直到被重新赋值，在整个程序执行过程中都存在。

在模块、接口、任务或函数体外声明的任何数据都具有全局的作用范围（可以在其声明后的任何地方使用），并且具有一个静态的生命周期（在整个确立和仿真时间内存在）。在模块或接口内，但在任务、进程或函数之外声明的变量具有本地的作用范围，并具有静态的生命周期（在模块或接口的生命周期内存在）。在自动任务、函数或块内声明的变量具有调用期或激活期内的生命周期，并且具有本地的作用范围。

在一个静态任务、函数或块内声明的变量默认情况下具有静态的生命周期并具有本地的作用范围。

Verilog – 2001 允许将任务和函数声明成自动的（automatic），这使得任务或函数内的所

有存储空间都是自动的。SystemVerilog 允许一个静态任务或函数内的特定数据被显式地声明成自动的（automatic）。声明成自动的变量具有调用块内的生命周期，并且在每次进入调用或块内的时候进行初始化。

SystemVerilog 也允许变量被显式地声明成静态的。在一个自动任务、函数或块内声明的静态数据具有静态的生命周期，并且对于块来说具有本地的作用范围。实例如源代码 2-21 所示。

源代码 2-21 静态变量实例

```
//Chapter2 static_auto_example.sv
module static_auto_example;
int st0;// static

initial begin
int st1;// static
static int st2;// static
automatic int auto1;// automatic
end

task automatic t1();
int auto2;// automatic
static int st3;// static
automatic int auto3;// automatic
endtask
endmodule
```

对于在模块、接口或程序中定义的任务、函数或块，SystemVerilog 加入了一个可选的限定符来指定其中声明的所有变量的缺省生命周期。生命周期限定符可以是 automatic 或 static，缺省的生命周期是 static。

```
program automatic test ;
int i; // 没有在过程语句中，静态
task t( int a ); // t 中的参数和变量属于动态
...
// 除非显式定义为静态
endtask
endprogram
```

类的方法以及声明在 for 循环内部的变量缺省是自动的，无论它们所在的范围的生命周期属性是什么。注意：自动或动态变量不能够使用非阻塞赋值（<=）或连续赋值语句（assign）来赋值。自动变量以及动态结构（对象句柄、动态数组、关联数组、字符串以及事件变量）应被限制到过程关联文中。

2.6 数据类型转换

Verilog 是一种弱类型语言，允许一个数据类型的值赋给另一个数据类型的变量或者网线。赋值时，按照 Verilog 标准中定义的规则进行数据类型的变换。

SystemVerilog 增加了强制数据类型转换的能力。强制类型转换不同于在赋值时自动转变；使用强制类型转换，不用赋值，在一个表达式内，一个数值就可以转换成一个新的类型。

Verilog - 1995 标准没有提供强制类型转换的方法。Verilog - 2001 提供了有限的类型转换——使用系统函数 \$signed 和 \$unsigned 实现有符号数和无符号数之间的转换。

2.6.1 静态类型转换

SystemVerilog 加入了一个强制类型转换操作符（'）来改变一个表达式的数据类型：

```
<type>'(<expression>);
```

需要进行强制类型转换的表达式必须包含在圆括号内，或者必须包含在串联或复制花括号内。例如：

```
int '(2.0* 3.0)
shortint '{8'hFA,8'hCE}
```

如果将一个正的十进制数作为数据类型，那么这意味着需要改变数据的位宽。例如：

```
16'(2)
```

数据的符号也可以改变。例如：

```
bit [7:0] x;
signed '(x)
```

也可以使用用户定义的数据类型：

```
mytype'(foo)
```

如果强制类型转换中的表达式需要改变位宽或符号，那么该表达式必须具有整型值。当改变位宽的时候，符号不会发生变化。当改变符号的时候，宽度不会发生变化。

当强制转换到一个预定义的数据类型的时候，强制类型转换操作符的前缀必须是预定义的类型关键字。当强制转换到一个用户定义的数据类型的时候，强制类型转换操作符的前缀必须是用户定义的类型标识符。

如果一个 shortreal 类型转换到一个 int 或 32 位数据类型，就像 Verilog 的规定一样，它的值会进行四舍五入，因此转换会丢失信息。压缩类型间的类型转换并不要求显式转换，因为它们都被当作是整型值，但工具可以使用强制类型转换来执行更严格的类型检查。

2.6.2 动态类型转换

上面我们介绍的 SystemVerilog 静态类型转换是一种编译时的转换，转换操作总会运行，而不会检查结果的有效性。如果需要进一步的检验，SystemVerilog 提供了一个新的系统函数 \$cast，这是动态的，在仿真运行时进行数据类型的检查，\$cast 可以作为任务或函数来调用。

\$case 的语法如下：

```
function int $cast(singular dest_var, singular source_exp);
```

或者

```
task $cast(singular dest_var, singular source_exp);
```

dest_var 是目标变量，source_exp 是源变量的表达式。

将 \$cast 作为任务还是函数调用，确定了无效赋值是如何处理的。当作为任务调用时，\$cast 试图将源表达式赋值给目的变量；如果赋值是无效的，会出现运行时错误并且目的变量保持不变。当作为函数调用时，\$cast 试图将源表达式赋值给目的变量，如果赋值成功则返回 1。如果赋值失败，\$cast 不会进行赋值操作而是返回 0。当作为函数调用时，不会出现运行时错误，并且目标变量保持不变。

需要注意：\$cast 执行的是仿真运行时检查。除了检查目的变量和源表达式是否为单一类型外，编译器不会进行类型检查。

例子：

```
typedef enum {red, green, blue, yellow, white, black} Colors;
Colors col;
$cast(col, 2+3);
```

这个例子将表达式 (5, black) 赋值给枚举类型。如果不使用 \$cast 或者下面提到的静态编译时强制类型转换，这种类型的赋值是无效的。

下面的例子显示了如何使用 \$cast 检查一个赋值操作是否成功：

```
if (!$cast(col, 2+8)) // 10: 无效的强制类型转换
    $display("Error in cast");
```

作为一种选择，前一个例子还可以使用静态强制类型转换来进行类型转换，例如：

```
col = Colors'(2+3);
```

然而，这是一种编译时的强制类型转换，也就是说，在静态强制类型转换操作时总是运行的，即使表达式不在枚举值合法的范围之内，静态强制转换也不会提供错误检查或警告。

这两种类型的强制类型转换为用户提供了完全的控制能力。如果用户知道将某个表达式赋值给一个枚举变量是安全的，那么可以使用静态强制类型转换。如果用户需要检查表达式是否位于枚举值范围内，那么用户没有必要手工编写一个冗长的 case 语句，通过使用 \$cast 函数，编译器自动提供了这种功能。通过这两种类型的强制类型转换，用户能够控制时间和安全性之间的平衡。

第3章 | SystemVerilog

并发进程与进程同步

硬件设计可以看成是由很多并发或者并行的进程组成的。例如，一个全速双工的以太网 MAC，一个模块从以太网接入端接受以太包，另外一个模块同时向以太网接入端发送以太包。当你仔细检查两个模块的具体实现的时候，会发现更多的并行进程：计数器、解码器、比较器等；并发在任何一个抽象层次存在，即使是晶体管级。

验证环境模拟硬件运行的能力越强，就越容易对硬件建模和验证。在验证平台中，每一个逻辑独立的并行操作都是采用独立的进程来实现的。每个接口或者事务处理都可以是一个进程。例如，在以太网 MAC 的验证平台中，一个主事务处理器把包送入 MAC，同时 MII 的监控器并行接受包和验证它们的正确性。在监控器和事务处理器之间，有很多个其他并行的进程来模拟和验证不同的 MAC 行为。验证平台中的这些并发进程要求验证语言要有高级的并发处理能力。

SystemVerilog 提供了下列处理并发进程的能力。

- fork...join 并发结构。
- 通过 mailbox 实现进程间的通信。
- 通过 semaphore 实现进程互斥和仲裁。
- 通过 event 实现进程之间的同步。

3.1 fork...join

fork...join 能够启动产生多个并发进程。如图 3-1 所示，fork...join 块可以指定一个或多个语句块，每一个语句块都应该作为并发进程执行。

fork...join 的语法结构如下：

```
fork
  code_block_1
  code_block_2
  ...
  code_block_n
join/join_any/join_none
```

fork...join 内的每个语句块称为子进程，执行这段 fork...join 代码的称为父进程。

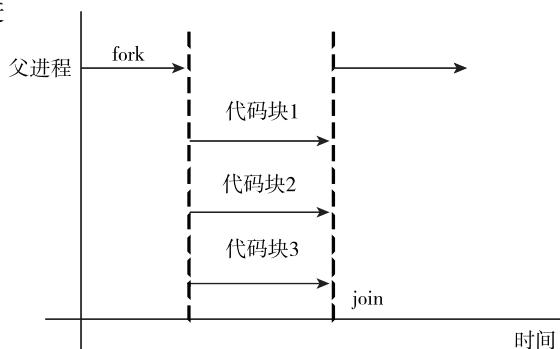


图 3-1 fork...join 并发进程

3.1.1 三种并发方式

Verilog 中的 fork...join 将执行 fork 语句的进程，并阻塞父进程的执行，直到 fork...join 中所有进程（子进程）中止。通过加入 join_any 和 join_none 关键字，SystemVerilog 提供了三种选择来指定父进程何时恢复执行。

fork...join 结构中，父进程会被阻塞直到所有的子进程结束；join_any 结构中，父进程会被阻塞直到其中任意一个子进程结束；join_none 父进程会立即与产生的所有进程并发执行。如图 3-2 所示。

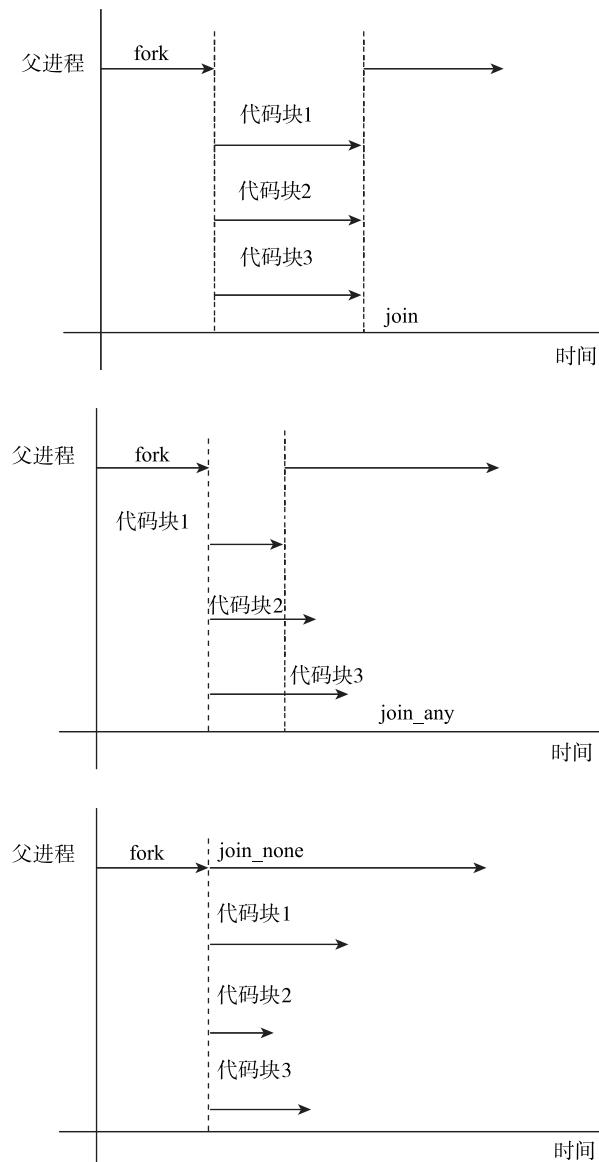


图 3-2 三种并发类型

下面的例子采用了 fork...join 结构。目标地址为接口 d 的发包进程只有在其他三个目标地址为 a/b/c 的发包进程结束后才可以运行。假如采用 fork...join_any 结构，目标地址为 d 的发包进程将会在三个接口中的其中一个发包进程结束后立即运行。最后，若采用 fork...join_none 结构，所有四个发包进程将同时运行。

```

fork
    send_packet(interface_a);
    send_packet(interface_b);
    send_packet(interface_c);
join
    send_packet(interface_d);

```

在定义一个 fork...join 块的时候，将整个子进程封装在一个 begin...end 块中会将整个块作为单个进程执行，其中每条语句顺序地执行。

```

fork
    begin
        statement1; // 一个带有 2 条语句的子进程
        statement2;
    end
join

```

源代码 3-1 所示包含两个子进程，第一个等待 20ns，第二个等待有名事件 eventA 被触发。因为指定了 join 关键字，父进程应该阻塞直到这两个子进程结束。也就是说，直到过了 20ns 并且 eventA 被触发，才结束运行整个 fork...join 的结构。

源代码 3-1 fork...join 语句实例

```

//Chapter3 fork_example.sv
module fork_example();
event eventA;
initial begin
    fork
        begin
            #20 ns ;
            $display("First Block finished!");
        end
        begin
            @ eventA;
            $display("Second Block finished!");
        end
    join
    $display("fork...join finish!");
end

initial begin
#30 ns ;
-> eventA;
end
endmodule

```

上面这个例子，在仿真器中的打印输出结果将会是：

```
First Block finished!
Second Block finished!
fork...join finish!
```

为了能够区分三种并行进程控制机制，我们再提供另外两个例子，源代码 3-2 是 fork…join_any 语句实例。fork…join_any 会在任何一个内部进程结束后，进入执行该并行结构之后的可执行语句。

源代码 3-2 fork…join_any 语句实例

```
//Chapter3 fork_any_example.sv
module fork_any_example();
event eventA;
initial begin
  fork
    begin
      #20 ns ;
      $display("First Block finished!");
    end
    begin
      #30 ns ;
      $display("Second Block finished!");
    end
    join_any
    #5 ns ;
    $display("fork...join_any finish!");
    #100 ns ;
  end
endmodule
```

本例在仿真器的运行输出会是：

```
First Block finished!
fork...join_ any finish!
Second Block finished!
```

源代码 3-3 是 fork…join_none 语句实例，该结构只是将内部的进程平行调用，而无须等待任何一个结束，就可以执行该结构后续的语句。

源代码 3-3 fork…join_none 语句实例

```
//Chapter3 fork_none_example.sv
module fork_none_example();
event eventA;
initial begin
  fork
    begin
      #20 ns ;
      $display("First Block finished!");
```

```

end
begin
#30 ns ;
$display("Second Block finished!");
end
join_none
#5 ns ;
$display("fork...join_none finish!");
#100 ns ;
end
endmodule

```

该例子在仿真器的运行输出会是：

fork...join_none finish!

First Block finished!

Second Block finished!

从上面三个例子及其运行结果，读者可以认识到这三种并发进程调用结构的运行机制的区别。

3.1.2 进程与变量

子进程可以在其内部定义本地变量，也可以访问在 fork 结构以外的变量。默认情况下，父进程和子进程都可以修改父进程定义的变量，而且都可以看到变量的变化。这类共享变量可以用来作为进程之间的通信。

采用共享变量作为通信是有效的，但也是很不方便的，因为可以导致进程之间的竞争。例如，下面的代码很方便的启动了十个并行进程，而且每个都有独立的 id 作为参数。

```

initial begin
for (id=0;id<10;id++)
  fork
    my_process(id);
  join_none

```

值得注意的是，每个并行进程都可能获取一个被修改了的 id，因为 id 随着循环的迭代变化，而每个 my_process 的进程访问同一个变量 id。

在 SystemVerilog 中会先执行循环体，将十个子进程展开后再启动并行执行。fork 操作会把每个子进程添加到一个进程管理器中，但是不会对其立即执行。当父进程结束了循环体或者挂起的时候，进程管理器才会执行其列表中的子进程。为此，当所有十个子进程开始执行的时候，作为参数传递的 id 的值将会是 10，因为父进程完成了整个循环体的迭代。

为了解决这个竞争，我们可以采用自动变量的方式来避免这种情况的出现，保证每个独立的进程可以保存其独立的 id。

```

automatic integer id;
for (id=0;id<10;id++)
  fork

```

```
my_process(id);
join_none
```

当拥有自动变量的子进程被 fork 操作调用的时候，一个新的 id 的复制会被创建并送入子进程中，这样，每个子进程都有一份独立 id 的复制，而不是访问同一个静态的共享变量，父进程中对 id 的修改不会影响各个子进程独立的 id。

3.1.3 进程控制

除了启动并发进程之外，在某些情况下我们还需要对某个进程进行控制，例如，停止或者等待所有进程。Verilog 中的 disable 语句同样适用于 SystemVerilog 的进程。另外，SystemVerilog 增加了一些结构，它们允许一个进程能够中止其他进程，或者等待其他进程的结束：wait fork 等待进程的结束，disable fork 结构停止进程的执行。

源代码 3-4 的例子通过 disable 来实现一个 time - out 的看门狗。

源代码 3-4 disable 语句实例

```
//Chapter3 disable_example.sv
module disable_example();
parameter time_out = 1000;
bit start_enable=0;
initial begin
  fork
    begin
      fork :time_out_block
        wait (start_enable==1);
        #time_out $display( "@ % 0d: Error - timeout: start_enable is not 1",
$time);
        join_any
        disable time_out_block;
      end
      join
    end
  endmodule
```

除了上面采用 disable 语句来中止有名块的进程之外，SystemVerilog 还提供了 disable_fork 用来中止调用进程的所有活跃的子进程，以及子进程内产生的所有子进程。也就是说，如果任何子进程还具有它们自己的子进程，那么 disable fork 语句也会将它们中止。

在下面的例子中，任务 get_first 产生了一个任务的三个版本，每一个都用来等待一个特定的设备（1、7 或 13）。任务 wait_device 等待一个特定的设备准备就绪，然后返回设备的地址。当第一个设备有效的时候，get_first 任务应该杀死正在执行的所有进程然后继续执行。

```
task get_first(output int adr);
fork
  wait_device(1, adr);
  wait_device(7, adr);
```

```

    wait_device(13, adr);
    join_any
    disable fork;
endtask

```

disable fork 语句与 disable 的不同之处在于， disable fork 会考虑动态的进程父子关系，而 disable 则使用静态的语法信息。因此， disable 中止执行一个特定块的所有进程，无论进程是否由 fork 调用产生分支；而 disable fork 则仅仅中止那些由 fork 调用产生的进程。

wait fork 语句被用来确保所有子进程（调用进程产生的进程）的执行都已经结束。

当没有进一步的动作时， Verilog 会中止仿真的运行。当所有的程序块和 initial 块停止执行的时候， SystemVerilog 会自动中止整个仿真，而不管任何子进程的状态如何。 wait fork 语句允许一个程序块在退出之前等待它的所有并发进程的结束。

对于下面的例子，在任务 do_test 中，前两个进程产生后，任务会阻塞直到其中一个进程结束（或者是 exec1 ，或者是 exec2 ）。接下来会在后台产生另外两个进程。 wait fork 语句能够确保任务 do_test 在返回调用之前能够等待产生的所有四个进程结束。

```

task do_test;
  fork
    exec1();
    exec2();
  join_any
  fork
    exec3();
    exec4();
  join_none
  wait fork ;// 等待直到 exec1 ... exec4 完成才退出该任务
endtask

```

3.2 mailbox

变量不是进程之间唯一的通信方式。 SystemVerilog 提供了 mailbox ，如图 3-3 所示，它可以看成是一个先进先出（ FirstIn-FirstOut ， FIFO ）的存储数组。通常有一个或者多个进程把数据送入一个 mailbox ，有一个或者多个进程从 mailbox 读出数据。客户进程可以被挂起，直至 mailbox 有可用的数据，也就是说， mailbox 可以实现生产进程和客户进程的同步。



图 3-3 mailbox 的基本行为

3.2.1 mailbox 的基本操作

mailbox 是一种通信机制，它使得数据可以在进程间传递和通信，数据被一个进程发送到一个 mailbox 中，而另外一个进程可以从中获得。

从概念上讲，mailbox 的行为就像一个真实的邮箱一样。当一封信被分发并且放入到邮箱的时候，另一个人可以重新取出这封信（以及存储在其中的任何数据）。然而，如果当这个人检查邮箱的时候这封信还没有被分发，那么这个人必须做出选择：要么等待这封信被分发，要么在下一次检查邮箱的时候取出这封信。与此类似，SystemVerilog 的 mailbox 以一个可控的方式来传输和接收数据。在创建 mailbox 的时候，它可以是有界的队列（bounded queue），也可以是无边界的队列（unbounded queue）。当一个有界 mailbox 存储的数据达到了其边界数目的时候，mailbox 会变满；试图向已经满了的 mailbox 放置数据的进程会被阻塞，直到在 mailbox 队列有足够的空间；无边界 mailbox 永远也不会阻塞一个发送操作进程。

定义一个 mailbox 的例子如下：

```
mailbox mbxRcv;
```

mailbox 是一个内建的类，它提供了下列方法。

- 创建一个 mailbox： new ()。
- 将一个数据放置到 mailbox 中： put ()。
- 尝试将一个数据无阻塞地放置到 mailbox 中： try_ put ()。
- 从 mailbox 中获取一个数据： get () 或者 peek ()。
- 尝试无阻塞地从 mailbox 中获取一个数据： try_ get () 或 try_ peek ()。
- 查询 mailbox 中数据的个数： num ()。

下面我们详细对各个方法进行解释。

(1) 方法 new()

原型： `function new(int bound = 0);`

`new ()` 方法用来创建 mailbox，返回 mailbox 句柄；如果不能创建，则返回 null（空句柄）。缺省情况下，`bound` 参数为 0，表示 mailbox 是无边界的，此时 `put ()` 操作永远不会被阻塞；如果 `bound` 为非 0，那么它表示 mailbox 队列的长度（或者理解为 FIFO 的深度）。`bound` 必须是正的；负的边界是非法的，并会导致不确定的行为。

(2) 方法 put()

原型： `task put(singular message);`

`put ()` 方法严格按 FIFO 的顺序将一个数据存储到 mailbox 中。如果 mailbox 是一个有界的队列而且队列已满，那么进程会被阻塞直到队列中有足够的空间。

(3) 方法 try_put()

原型： `function int try_put(singular message);`

`try_put ()` 方法尝试将一个数据放置在一个 mailbox 中。`try_put ()` 方法严格按 FIFO 的顺序将一个消息存储在 mailbox 中。这个方法仅仅对有界 mailbox 才有意义。如果 mailbox 没有满，那么指定的消息被放置在 mailbox 当中，并且函数返回 1。如果 mailbox 满了，那么

函数返回 0。

(4) 方法 get()

原型: `task get(ref singular message);`

`get()` 方法从 mailbox 中取出一个数据并从队列中将其删除。如果 mailbox 是空的，那么当前的进程阻塞直到一个数据被放置到 mailbox 中。如果在数据变量和 mailbox 中的数据间存在类型不匹配，那么会产生一个运行时错误。

(5) 方法 try_get()

原型: `function int try_get(ref singular message);`

`try_get()` 方法尝试无阻塞地从一个 mailbox 中取出一个数据。如果 mailbox 是空的，那么 `try_get()` 方法返回 0。如果数据变量和 mailbox 中的数据间存在类型不匹配，那么 `try_get()` 方法返回负数。如果一个数据是有效的，并且数据类型与数据变量的类型匹配，那么数据被取出并且 `try_get()` 方法返回 1。

(6) 方法 peek()

原型: `task peek(ref singular message);`

`peek()` 方法从一个 mailbox 中复制一个数据但不会将其从队列中删除。如果 mailbox 是空的，那么当前的进程会阻塞直到一个消息被放置到 mailbox 中。如果在数据变量和 mailbox 中的数据间存在类型不匹配，那么会产生一个运行时错误。

(7) 方法 try_peek()

原型: `function int try_peek(ref singular message);`

`try_peek()` 方法尝试从一个 mailbox 中复制一个数据但不会将其从队列中删除。如果 mailbox 是空的，那么 `try_peek()` 方法返回 0。如果在数据变量和 mailbox 中的数据间存在类型不匹配，那么 `try_peek()` 方法返回负数。如果一个数据是有效的，并且数据类型与数据变量的类型匹配，那么数据被复制并且 `try_peek()` 方法返回 1。

(8) 方法 num()

原型: `function int num();`

一个 mailbox 中存储数据的数目可以通过 `num()` 方法获得。在使用返回值的时候需要特别注意，因为它只有等到 `get()` 或 `put()` 在 mailbox 上执行后才有效。这些 mailbox 方法可能来自不同的进程，而不仅仅是执行 `num()` 方法的那个进程。因此，返回值的有效性应该依赖于其他方法启动和结束的时间。

对于 mailbox 的基本操作，分为写入 (`put/try_put`)、取出 (`get/try_get`) 和复制 (`peek/try_peek`)。这三种基本操作又分成两种不同的性质：阻塞和非阻塞，如表 3-1 所示。

带 `try_*` 前缀的为非阻塞，表示在操作条件不满足的时候，放弃该操作而进入下一进程的运行。

表 3-1 阻塞操作与非阻塞操作

方法	功能	阻塞	非阻塞
<code>put()</code>	送入一个数据	Yes: mailbox 满时阻塞该进程	No
<code>get()</code>	取走一个数据	Yes: mailbox 空时阻塞该进程	No
<code>peek()</code>	复制一个数据	Yes: mailbox 空时阻塞该进程	No

(续)

方法	功能	阻塞	非阻塞
try_put()	尝试送入一个数据	No	Yes: mailbox 满时放弃操作
try_get()	尝试取出一个数据	No	Yes: mailbox 空时放弃操作
try_peek()	尝试复制一个数据	No	Yes: mailbox 空时放弃操作

3.2.2 参数化 mailbox

缺省的 mailbox 是无类型的，也就是说，单个 mailbox 可以发送和接收任何类型的数据。这是一个非常强大的机制，然而致命的是，它也会因为存储数据与用来获取数据变量间的类型不匹配而导致运行错误。一个 mailbox 经常被用来传输一个特定类型的数据，在这种情况下，如果能够在编译时发现类型不匹配，那将是十分有用的。

参数化 mailbox 与参数化类、模块和接口使用相同的机制。其定义如下：

```
mailbox #(type = dynamic_type)
```

其中 dynamic_type 代表一个特殊的类型，它能够执行运行时的类型检查（缺省情况）。

一个特定类型的参数化的 mailbox 通过指定类型来声明，如下列所示：

```
typedef mailbox #(string) s_mbox;
s_mbox sm = new;
string s;
sm.put("hello");
...
sm.get(s); // s < - "hello"
```

参数化 mailbox 提供了所有与通用 mailbox 相同的标准方法：num()、new()、get()、peek()、put()、try_get()、try_peek()、try_put()。

通用（动态的）mailbox 与参数化 mailbox 之间唯一的不同之处是：对于参数化 mailbox，编译器能够确保 put()、try_put()、peek()、try_peek()、get() 和 try_get() 与 mailbox 指定的类型兼容，从而所有的类型不匹配都可以被编译器发现而不是在运行时发现。

3.2.3 mailbox 应用实例

源代码 3-5 所示是采用 mailbox 作为同步器，发送和接收数据的例子。

源代码 3-5 mailbox 应用实例

```
//Chapter3 mailbox_example.sv
module mailbox_example;
    mailbox #(int) my_mailbox = new(2);

    task send();
        for (int i=0;i<4;i++) begin
            my_mailbox.put(i);
            $display("Sent:% 0d",i);
        #5;
    end
endtask

task receive();
    int value;
    my_mailbox.get(value);
    $display("Received:% 0d",value);
endtask
endmodule
```

```

    end
endtask

task receive();
    int temp;
    while (1) begin
        my_mailbox.get(temp);
        $display("Received:% 0d",temp);
    end
endtask

initial begin
    fork
        send();
        receive();
    join
end
endmodule

```

上面的例子中，我们定义了一个参数化 mailbox：my_mailbox，其可以传递的数据类型是 int，深度为 2；任务 send 将一个整型数据 i 送入 my_mailbox 中；任务 receive 从 my_mailbox 中取出数据。后面我们还会详细介绍如何通过使用 mailbox 传递类的对象。

3.3 semaphore

当有多个进程进行通信时，就可能出现竞争。多个进程可能需要使用一个稀缺的资源，但是由于资源的约束，只有一定数量的进程才可以有访问权限。现实的例子就是：如果多个进程需要通过一个接口发送数据包，因为每次只能允许一个包在该接口上发送，每个进程需要等待自己获取权限才可以发送。

每当出现资源竞争的时候，获取一个共享资源就必须通过仲裁。SystemVerilog 通过 semaphore 提供了仲裁的功能。在操作系统上的术语就是“互斥接入”，为此，semaphore 也可以看成互斥量，来控制对共享资源的访问。

从概念上讲，semaphore 可以理解成一个存储桶。当为 semaphore 分配内存的时候，会创建一个带有一定数目 key 的存储桶。使用 semaphore 的进程在继续执行之前必须首先从存储桶中获得一个 key。如果一个特定的进程需要一个 key，那么同时可以执行的进程数目与 key 的数目一致。所有其他进程必须等待直到足够数目的 key 被放回到存储桶中。一般情况下，semaphore 被用来控制对共享资源的互斥访问和实现基本的同步。如图 3-4 所示。

创建一个 semaphore 的例子如下：

```
semaphore smTx;
```

semaphore 是一个内建的类，它提供了下列方法。

- 创建一个具有指定数目 key 的 semaphore：new()。
- 从存储桶中取出一个或多个 key：get()。

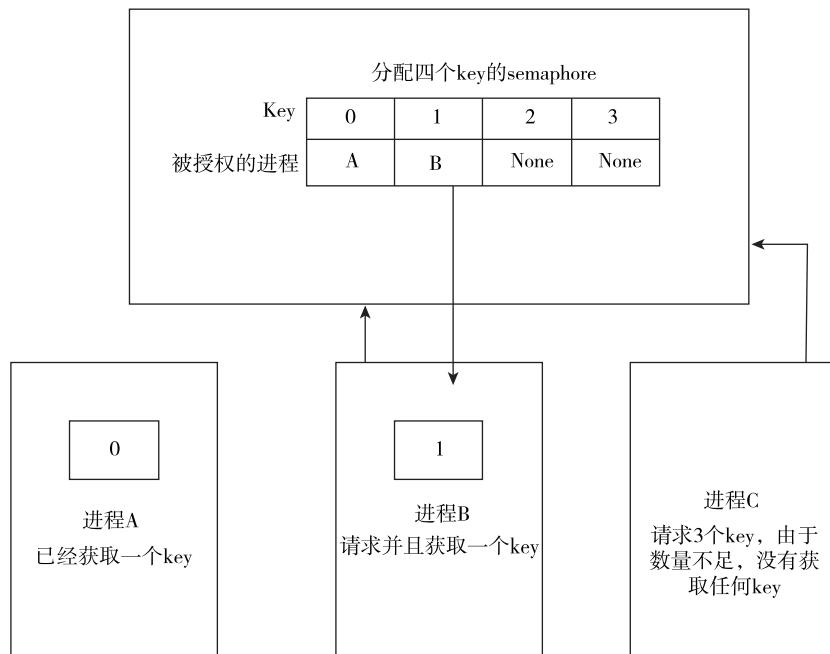


图 3-4 semaphore 的行为

- 向存储桶中放回一个或多个 key: put()。
- 尝试无阻塞地获取一个或多个 key: try_get()。

3.3.1 semaphore 的基本操作

semaphore 的基本操作和 mailbox 类似，主要有 new()、put()、get()、try_get() 四个方法，下面我们逐个讨论介绍其用法。

(1) 方法 new()

原型: `function new(int keyCount = 0);`

semaphore 使用 new() 方法创建。KeyCount 指定了最初被分配到 semaphore 存储桶中的 key 的数目。当放入存储桶中的 key 比取出的数目多的时候，存储桶中 key 的数目可以超过 KeyCount。KeyCount 的缺省值为 0。new() 函数返回 semaphore 的句柄，如果没有产生 semaphore，则返回 null。

(2) 方法 put()

原型: `task put(int keyCount = 1);`

keyCount 指定了放回到 key 的数目，它的缺省值为 1。

当调用 semaphore.put() 任务的时候，指定数目的 key 被放回到 semaphore 中。如果一个进程已经被挂起来等待一个 key，当返回了足够的 key 后这个进程可以继续执行。

(3) 方法 get()

原型: `task get(int keyCount = 1);`

get() 方法被用来从一个 semaphore 中获得指定数目的 key。keyCount 指定了需要获得的 key 的数目，它的缺省值为 1。如果指定数目的 key 存在，那么方法返回并且进程会继续执

行；如果不存在指定数目的 key，进程会阻塞直到对应数量的 key 出现。

(4) 方法 try_get()

原型：function int try_get (int keyCount = 1);

try_get()方法被用来无阻塞地获得指定数目的 key。keyCount 指定了需要获得 key 的数目，它的缺省值为 1。如果指定数目的 key 存在，那么 try_get()方法返回 1 并且进程会继续执行；如果指定数目的 key 不存在，那么 try_get()方法返回 0。

3.3.2 semaphore 应用实例

源代码 3-6 所示是一个 semaphore 应用的例子。

源代码 3-6 semaphore 应用实例

```
//Chapter3 semaphore_example.sv
module semaphore_example();
    semaphore s1 = new(1);

    task t1();
        for (int i = 0; i < 3; i++) begin
            s1.get(1);
            #5;
            $display("t1 owns semaphore");
            s1.put(1);
            #5;
        end
    endtask

    task t2();
        for (int i = 0; i < 3; i++) begin
            s1.get(1);
            #5;
            $display("t2 owns semaphore");
            s1.put(1);
            #5;
        end
    endtask

    initial begin
        fork
            t1();
            t2();
        join
    end
endmodule
```

该例子创建了 t1 和 t2 两个并行的任务，从只有一个 key 的 semaphore s1 中取出 key，然后放回去；一旦其中一个任务获得 key，另外一个任务就必须等待，直至该 key 被放回到存储桶中。

3.4 event

在 Verilog 中，命名的事件（event）是静态对象，它可以通过“`->`”操作符触发，并且进程通过“`@`”操作符可以等待一个事件被触发。SystemVerilog 中的事件支持相同的基本操作，但它从几个方面增强了 Verilog 事件。最为显著的增强是：Verilog 命名事件的触发状态没有持续时间，而在 SystemVerilog 中，这个触发状态在事件被触发的整个时间片内持续；SystemVerilog 的事件还可以作为同步队列的句柄；事件还可以作为参数传递给任务，而且可以相互赋值和比较。

SystemVerilog 的事件为底层的同步对象提供了一个操作句柄。当进程等待一个事件被触发的时候，进程被放入到一个在同步对象内部的维护队列当中。进程可以或者通过“`@`”操作符，或者通过使用 `wait()` 结构检查它们的触发状态来等待一个 SystemVerilog 事件被触发。事件通过使用“`->`”或“`->>`”操作符来触发。

```
event_trigger ::=  
    -> hierarchical_event_identifier;  
  |->> [delay_or_event_control] hierarchical_event_identifier;
```

3.4.1 事件触发

命名事件可以通过“`->`”操作符触发。

触发一个事件可以解除当前所有等待这个事件被阻塞的进程。当被触发以后，命名事件的行为就好像一次射击，也就是说，触发状态本身是不可观测的，我们仅仅能够观测到它的效果。这种方式类似于一个边沿（上升沿或者下降沿）触发一个触发器，但边沿的状态却不能永远持续和获取，也就是说，`if (posedge clock)` 是非法的。

非阻塞事件使用“`->>`”操作符触发。“`->>`”操作符的效果是语句会无阻塞地执行，并且在延时控制过期或事件控制发生的时候，会产生一个非阻塞的赋值更新事件。这个更新事件的效果应该是在仿真周期的非阻塞赋值区域触发被引用的事件。

3.4.2 等待事件

等待一个事件被触发的基本机制是通过事件控制操作符“`@`”：

```
@ hierarchical_event_identifier;
```

“`@`”操作符阻塞调用进程直到指定的事件被触发。

对于用来解除一个等待事件的进程的触发，等待进程在触发进程执行触发操作符“`->`”之前必须执行“`@`”语句。如果触发先执行，那么等待进程会永远处于阻塞状态。

3.4.3 事件的触发属性

SystemVerilog 能够区分事件触发（瞬时的）和事件触发状态——在整个时间片持续（也就是直到仿真时间继续）。事件的 `triggered` 属性允许用户检查这个状态。

triggered 属性使用一个类似于内置方法的语法来调用：

```
hierarchical_event_identifier.triggered
```

如果指定的事件在当前的时间片中已经被触发，那么事件的 triggered 属性为真，否则为假。如果事件标识符为 null，那么事件的 triggered 属性为假。

事件的 triggered 属性在一个 wait 结构中最为有用：

```
wait(hierarchical_event_identifier.triggered)
```

使用这种机制，一个事件的触发可以解除所有被阻塞的等待进程，无论它在触发操作之前执行还是在与触发操作相同的仿真时间上执行。因此，当触发和等待同时发生的时候，事件的 triggered 属性能够消除一个常见的竞争。根据等待和触发进程的执行顺序，一个等待事件触发的阻塞进程有可能（可以或者不能）解除阻塞。然而，无论等待和触发操作的顺序如何，一个等待触发状态的进程总是能够解除阻塞。

event 应用实例如源代码 3-7 所示。

源代码 3-7 event 应用实例

```
//Chapter3 event_example.sv
module event_example;
event done, blast;
event done_too = done;

task trigger(event ev);
    -> ev;
endtask

initial begin
fork
    @ done_too;           //等待事件(@ done_too)需要在触发事件(->)之前执行
    #1 trigger(done);
join
fork
    -> blast;
    wait(blast.triggered); //等待的事件blast的触发状态
join
end

endmodule
```

上面例子中的第一条并行分支语句展示了两个事件标识符（done 和 done_too）如何引用相同的同步对象，它还展示了一个事件怎样被传递到任务中并被触发。在这个例子中，一个进程通过 done_too 等待事件，而真实的触发是通过将 done 作为一个参数传递的 trigger 任务完成的。

在第二条并行分支语句中，等待进程（wait (blast.triggered)）有可能在另外一个激发进程（-> blast）之前执行并等待这个事件触发。该等待进程可以解除阻塞并且 fork...join 结构可以中止执行。这是因为进程等待事件的触发状态，它可以在一段时间段内持续保持它的触发状态。

面向对象编程入门

SystemVerilog 也有类的概念，这部分语法子集和 C++/Java 类似，属于面向对象编程语言，为此它具有面向对象编程语言的语法特征，如类的封装、对象的生成和使用、类的继承和扩展、虚方法和多态。这一章重点介绍 SystemVerilog 中类的基本概念、语法特点以及如何使用类去构建验证平台中的验证组件和整个验证平台。

4.1 过程编程语言与面向对象编程语言

对象就是封装数据和对数据操作（方法）的编程结构。它包括了数据成员和方法（Method），方法也就是一个任务或者函数，对对象内的数据进行操作。

过程编程语言就如 Verilog 和 C，数据和对数据的操作的函数和任务是各自独立的。例如在 C 语言中，函数和数据可以被独立地定义，没有任何形式的联系。一个 C 语言的函数可以对多个数据做操作，而且多个函数可以对同一个数据做操作。这种情况下，程序的功能会变得难以理解。

在面向对象编程语言中，数据和对数据的操作可以封装到一个独立的对象中。

从图 4-1 中，我们可以看出过程编程和面向对象编程的差异。在过程化编程模式中，三个函数操作三个不同的变量，没有中心主题和方法。为了在其他地方重复使用类似的功能，多个数据变量或者变量的数组要被重新定义，而且函数也需要做相应的修改。

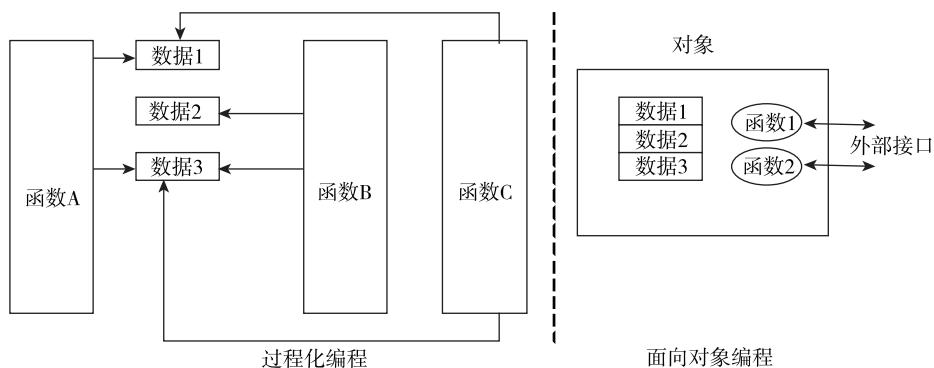


图 4-1 过程化编程与面向对象编程

在面向对象编程模式下，函数和数据被封装绑定到一个对象中。对象与外部的接口就是通过这些函数（或者任务，通称方法）来实现。如果相同的功能需要在多个地方应用，对象可以被例化多次，而内部无须改变。为此，对象是一个层次化的实现结构，我们可以将通用的操作和数据封装到一个对象中，以便重用。

另外一个很重要的特点是，在面向对象编程中，对象的使用者无须知道对象内部的实现细节，使用者只要知道如何通过对象的方法和对象进行交互即可。这使得对象内部实现细节的变动不会影响使用者。在编写验证平台中，这对于共享代码很有帮助。

在验证的过程中，采用对象可以提高验证平台的抽象层次，这使得开发者可以专注在验证平台和被测设计的交互过程。

在计算过程中，提高抽象层次可以让使用者更加容易的处理信息。在验证平台中，抽象可以将复杂的情况映射到一个简单的高层次的概念。例如，一个数字逻辑电路，其抽象层次可以从逻辑门电路到比较器到乘法器到算术运算单元或者 CPU。

在验证中，抽象层次定义了：

- 1) 验证平台和被测设计的交互方式。
- 2) 验证平台和其使用者——测试用例创建者的交互方式。

一旦抽象层次被确定之后，它可以映射到特定的用户自定义类型。在 SystemVerilog 中，这种用户自定义类型就是类（class），类可以将数据和对数据的操作封装在一起。

4.2 类

对象是如何定义的呢？一个对象通过它的类来定义，类的定义包括了一个对象的所有内容。类就是对象的一个模板，其内部定义了数据和方法。对象是类的一个例化和实现。

例如，你可以创建一个“银行卡”的类。这个银行卡类定义了一个公司的银行卡账号，而且定义了和这张卡相关的交易信息。

你可以定义一个称为“工行卡”的例化。工行卡现在就是一个对象了。你可以对这个类做多个例化，如：“交行卡”、“建行卡”、“招行卡”等。银行卡类定义了和一个通用银行卡账号的数据信息，例如账号用户的名字，账号；其中定义了所有的操作，包括“利息计算”、“添加交易”、“余额计算”等可能在银行卡对象上发生的操作。

方法，就是一个在对象中可以执行的操作。例如，上面我们提到“利息计算”、“添加交易”和“余额计算”等。

下面是一个简单的以太包的例子，数据成员包括前导码、帧头分隔码、目标地址、源地址、帧长和负荷；还有两个方法：打印函数和数据加载任务。

源代码 4-1 简单以太包的类实例

```
//Chapter4 ether_packet.sv
class ether_packet;
// 以太包字段
  bit [55:0] preamble = 'h5555555555555555;
  bit [7:0] sfd = 'hab;
  bit [47:0] dest, src;
```

```

bit [15:0] len;
bit [7:0] payld [ ];

function new (int i);
    payld = new [i]; len = i;
endfunction : new

function void print;
    $displayh("\t src: ", src);
    $displayh("\t dest: ", dest);
    $displayh("\t len: ", payld.size());
    for (int i=0;i < len;i++)
        $displayh("\t payld[% 0h]: % 0h", i, payld[i]);
    $displayh("");
endfunction : print

task load_frame(input [47:0] lsrc, ldest,
                input [7:0] start_dat);
    src = lsrc; dest = ldest;
    len = payld.size();
    if (start_dat > 0)
        for (int i=0;i < len;i++)
            payld[i] = start_dat + i;
endtask : load_frame

endclass : ether_packet
-----
```

你可以在 SystemVerilog 中的 program/module/package 内部或者外部定义一个类。类可以在 program 或者 module 内使用。

4.2.1 类的基本概念

在此我们顺便介绍一些面向编程语言中相关的一些术语。

- 属性 (property)：也就是类中定义的数据（变量）成员。
- 方法 (method)：也就是类中定义的函数或者任务。
- 句柄 (handle)：也就是指向对象的指针，即该对象的地址入口。

对于用户使用对象需要做如下的三个基本步骤。

- 1) 定义类。例如下面定义的类就是提供了一个包对象的模板：

```

class packet;
...
endclass

class long_packet;
...
endclass
```

- 2) 在 module 或者 program、class、function、task 等地方声明对象。

下面声明了三个对象，两个是 packet 的对象，一个是 long_packet 的对象。

```
packet my_packet;
packet packet_array[32];
long_packet my_l_packet;
```

对象的标识符（my_packet/packet_array/my_l_packet）是该例化对象的句柄。当该对象被创建的时候，该句柄才有效，默认情况下句柄将为空（null）。

3) 通过构造函数 new 创建对象的例化。

下面是通过 new 这个构造函数给对象分配内存空间，并且把入口地址赋给对象的句柄：

```
my_packet = new(168);
my_l_packet = new();
```

类、对象以及句柄和 new 之间的关系如图 4-2 所示，代码如下：

```
ether_packet e1_pkt,e2_pkt;
initial begin
e1_pkt = new(164);
e2_pkt = new(88);
end
```

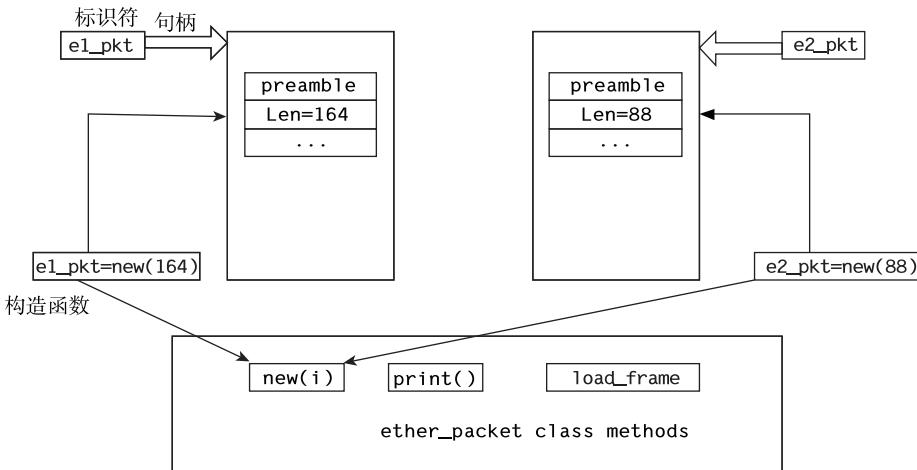


图 4-2 类、对象、句柄和构造函数

在验证平台中，类和对象对于激励的产生和数据的高层抽象很有帮助。类也可以用来封装一个通用的操作，从而在验证平台的不同地方重用。例如，以太网的媒体介入层有一个激励产生器在 MII 端和主机端。又如，在一个四个端口的以太网交换机中，验证平台需要四个事务处理器（transactor），全部都是一样的功能而且并行运行。我们可以定义一个 xactor 的类并且为每个端口例化一次。

对象可以和其他数据类型一样被声明。SystemVerilog 支持单个例化或者以数组的方式例化。下面就是一个包对象的动态数组：

```
ether_pkt pkt_list[];
```

创建了一个 pkt_list 的标识符（或称为句柄），在声明的过程中这个句柄是空值。

一个对象的例化是在它调用构造函数的时候被创建。构造函数为该对象分配内存空间

和并给句柄赋予有效值。

我们也可以在类的定义中声明对象：

```
class pktlist;
  packet pkt_queue[];
  ...
endclass
```

如上面所示，`pkt_queue` 中的每个包的对都要显式地构造。`pktlist` 的构造函数不会调用 `packet` 的构造函数。

一个类也可以在其定义中引用自身。下面是一个例子，`packet` 类中声明了 `next` 的 `packet` 对象。

```
class packet;
  packet next;
  ...
endclass
```

自我引用可以用来声明比较复杂的数据结构，例如双链表或者查找树。这种引用有别于使用 `this` 操作符。

对象的声明并不意味着对象的创建，声明没有实际分配空间。它只是简单的创建了一个标识符。在声明时，对象句柄为空。只有当为该对象分配空间时，句柄才被赋予有效值。分配空间需要通过构造函数来实现。在 SystemVerilog 中，这个构造函数就是 `new`。

在知道如何使用构造函数创建对象之后，我们如何使用对象呢？就如 Verilog 中的层次化引用一样，我们可以通过“.”操作符来访问对象中的成员：

```
initial begin
  ether_pkt e1_pkt;
  e1_pkt = new(64);
  e1_pkt.rst = 0;
  e1_pkt.dest = 1;
  e1_pkt.print;
  e1_pkt.load_frame(2,3,100);
end
```

4.2.2 构造函数

当定义了类和声明了对象，那么每个声明的标识符就对应这一个对象。然而此时，没有内存被分配，对象的句柄是一个空值。在调用该对象的构造函数之前，大部分的面向对象编程语言不会分配空间。一旦构造函数被调用，它就会为对象分配空间并给其句柄赋予有效值。构造函数也可以用来初始化对象的数据成员。

假如一个对象没有被构造，也就是说没有下列代码：

```
object_name = new();
```

那么空间不会被分配，句柄为空。如果该句柄被引用，就可能产生运行错误。当然我们可以通过下列代码来探测空句柄：

```
if (object_name == null) $stop;
```

SystemVerilog 中的对象可以通过构造函数 new 来初始化。如果一个数据成员没有在构造函数 new 中初始化，它将是系统的默认值。在构造函数中，数据成员可以被赋予初始值，如下所示：

```
function new ();
    addr = 3;
    foreach (data[i])
        data[i] = 5;
endfunction
```

也可以通过参数传递，使用参数对数据成员进行初始化，例如：

```
function new ( logic [31:0] addr = 3, d = 5 )
    this .addr = addr;
    foreach (addr[i])
        data[i] = d;
endfunction
```

当存储空间不再被某一个对象占用时，SystemVerilog，像 Java 一样，能够自动的回收对象例化（例如，内存分配）。为此，验证平台无须采用析构函数来释放空间，只要该例化没有被应用，系统就能自动析构。源代码 4-2 为类的构造函数实例。

实例如下：

源代码 4-2 类的构造函数实例

```
//Chapter4 new_construct_example.sv
module new_construct_example();
`include ether_packet.sv
ether_pkt e1_pkt, e2_pkt;
initial begin
e2_pkt = new(98);
e1_pkt = e2_pkt;
e1_pkt = new(118);
e1_pkt = new(119);
e2_pkt = new(200);
end
endmodule
```

源代码 4-2 的构造函数与句柄的关系如图 4-3 所示。SystemVerilog 的自动析构如图 4-4 所示。

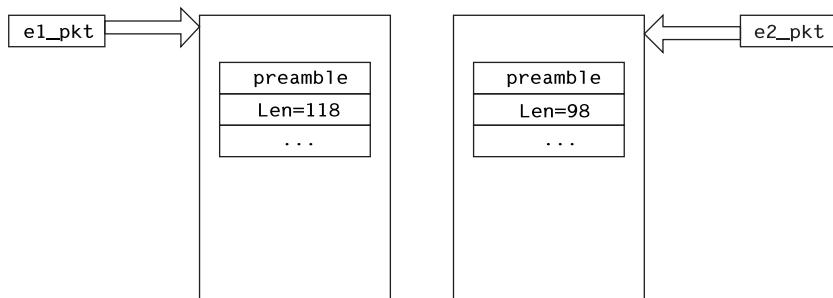


图 4-3 构造函数与句柄

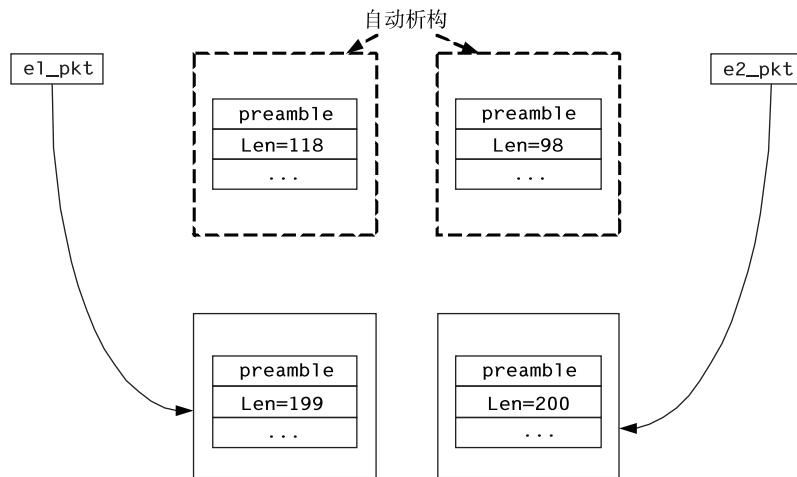


图 4-4 SystemVerilog 自动析构

在源代码 4-2 中，我们通过构造函数 new 给 e1_pkt 的句柄分配空间，并且传递了 98 这个实参给 len 变量；接着将句柄 e1_pkt 赋给 e2_pkt，此时系统没有分配任何新的空间给 e2_pkt（可参考图 4-5），而是将其指向 e1_pkt 的空间。后面通过连续的调用构造函数过程中，旧的空间将被自动析构收回。

当然，用户也可以通过将 null 赋值给一个对象，这样也可以将其分配的空间收回。

4.2.3 静态属性与方法

每个类的对象都拥有其自身的变量，自身的成员是不和其他对象共享的。假如有两个 ether_packet 的对象，它们有各自的目标地址、源地址和负载等。当需要在两个对象之间建立一种联系，可以让该类的所有对象都可以共享，例如，我们在激励产生器中会不断地产生一系列的以太包，在非面向对象的环境中，我们可能会创建一个全局的变量来统计包的个数；而在 SystemVerilog 中，我们可以采用静态的方式来声明属性和方法。

静态属性——每一个类的例化（也就是每一个对象）都拥有它自身每个变量的复制。有时要求所有的对象共享一个变量。我们可以使用关键字 static 来声明数据成员。源代码 4-3 展示了一个以太包的所有对象通过共享变量来统计 pkt_id。

源代码 4-3 类的静态变量实例

```
//Chapter4 static_var_class.sv
module static_var_class()
  class ether_packet;
    static int count = 0;
    int pkt_id;
    function new ;
      pkt_id = count + + ;
    endfunction
  endclass
  initial begin
```

```

ether_packet e1_pkt,e2_pkt;
e1_pkt = new();
e2_pkt = new();
$display("Second Packet,id=%d,count=%d",e2_pkt.id,e2_pkt.count);
end
endmodule

```

在源代码 4-3 中，静态变量 count 可以统计现存所有对象的个数，其在声明的时候被初始化为 0，而每当有新的对象产生的时候，它就会自增，对应着给该类的 pkt_id 赋值，来标识每一个独立的对象。对于类的静态属性，无需产生一个该类型的对象就可以直接使用；而对于所有类的例化，它们共同拥有该静态属性的一份复制。我们可以这样理解：类的静态属性是存放在类中的，而不是在每个类的对象中。

在激励产生的过程中，使用 pkt_id 对于我们调试追踪类的对象有很大的帮助。在调试验证平台的时候，我们期望能够在一个激励对象中获取唯一的标识值；SystemVerilog 无法打印输出每个对象在内存中的地址，但若我们能够通过定义一个 ID，那么对于辨别不同的对象就简单方便了。

静态方法——方法也可以被声明成静态的。静态方法对于类的所有对象都可以访问，它就像一个常规的方法一样，可以在类的外部被调用——即使没有该类的例化。静态方法不能访问非静态的成员（类的属性和方法），但它可以直接访问类的静态属性或调用同一个类的静态方法。在静态方法内部访问非静态成员或访问特殊的 this 句柄是不允许的，并且会导致一个编译错误。静态方法不能被声明为虚方法（Virtual）。类的静态方法实例如源代码 4-4 所示。

源代码 4-4 类的静态方法实例

```

//Chapter4 static_method_class.sv
module static_method_class();
  class id;
    static int current = 0;
    static function int next_id();
      next_id = ++current;
      $display("current id is %d",current);
    endfunction
  endclass
  initial begin
    id::next_id();
  end
endmodule

```

静态方法不同于一个具有静态生命周期的方法。前者指的是类内部方法的生命周期，而后者指的是方法内部的参数和变量的生命周期，如下所示：

```

class TwoTasks;
  static task foo();... endtask // 具有自动变量生命周期的静态方法
  task static bar();... endtask // 具有静态变量生命周期的非静态方法

```

```
endclass
```

默认情况下，类的数据成员以及方法的参数和变量具有自动的生命周期。

4.2.4 this 操作符

当使用一个变量时，SystemVerilog会在当前程序范围内查找，接着回到上一层范围查找，直至该变量被找到。假如在一个类的内部而且要求明确的指定引用的就是该类的成员，那该如何处理呢？SystemVerilog提供了 this 这个操作符（关键字）。关键字 this 被用来明确地引用当前类的属性或方法。关键字 this 对应着一个预定义的对象句柄，这个句柄可以在该对象内部使用，并可访问内部成员。关键字 this 只能在非静态方法中使用，否则会出现错误。源代码 4-5 定义了一个初始化任务的方法。

源代码 4-5 this 操作符实例

```
//Chapter4 this_class_example.sv
module this_example();
class this_class;
    integer x;
    function new(integer x)
        this.x=x;
    endfunction
endclass
...
endmodule
```

上述例子中，x 既可能是类的一个属性，也能可是构造函数 new 的一个自变量。在构造函数 new 中，对未限定范围的 x 会通过查看最内层的作用范围来解析，为此该 x 将是构造函数的自变量。为了访问类的属性，通过使用关键字 this 进行限定。

4.2.5 对象的赋值与复制

声明一个类的变量仅仅定义了对象的名字（标识符）。如：

```
Packet p1;
```

上述例子定义了一个变量 p1，它是一个类型为 Packet 类的对象的句柄，其初始值为空 (null)。在 Packet 类型的例化被创建之前，对象并不存在，并且 p1 为空句柄。

```
p1 = new;
```

如果声明了另外一个变量，并且将先前的句柄赋值给新的对象，如下所示：

```
Packet p2;
p2 = p1;
```

那么，仍然只有一个对象，它可以使用 p1 或 p2 引用，因为 new 只执行一次，所以只产生了一个对象。图 4-5 所示为两个句柄指向同一个对象。

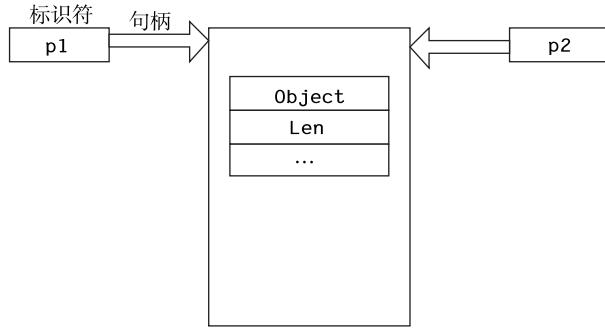


图 4-5 两个句柄指向同一个对象

如果将上面的例子按如下的方式改写，那么会产生一个 p1 的复制，如图 4-6 所示。

```
Packet p1;
Packet p2;
p1 = new;
p2 = new p1;
```

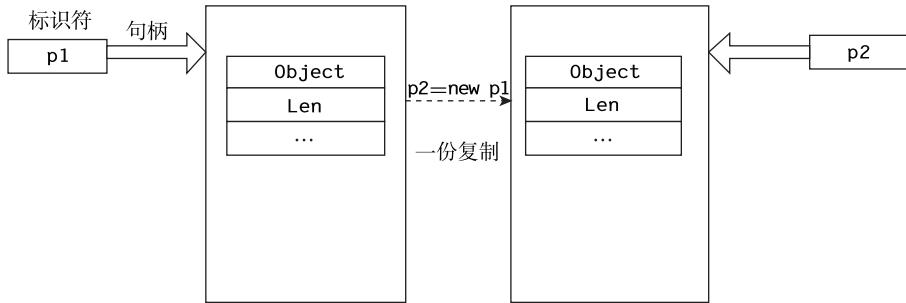


图 4-6 浅复制

最后一条语句是构造函数 new 的第二次执行，因此创建了一个新的对象 p2，它内部的属性全部复制自 p1 的内容；这种方式的复制称作浅复制（shallow copy，或称作浅拷贝）。所有的变量都被复制：整数、字符串、对象的句柄等。然而，其中包含的对象没有被复制，复制的只是它的句柄；与前面的例子一样，存在相同对象的两个名字；即使在类的声明中包含了构造 new 的例化，如源代码 4-6 所示。图 4-7 所示为只复制句柄不创建新的对象的浅复制示意图。

源代码 4-6 类的浅复制实例

```
//Chapter4_shallow_copy.sv
module shallow_copy();
class A;
    integer j = 5;
endclass
class B;
    integer i = 1;
    A a = new ;
endclass
```

```

function integer test;
    B b1 = new; // 创建一个 B 类的对象
    B b2 = new b1; // 创建一个 b1 对象的复制
    b2. i = 10; // i 在 b2 中改变, 然而在 b1 中没有改变
    b2. a. j = 50; // 改变 a. j, 而 a 为 b1 和 b2 共享的同一个对象
    test = b1. i; // test 被设置成 1(b1. i 没有改变)
    test = b1. a. j; // test 被设置成 50(a. j 已经改变了)
endfunction
...
endmodule

```

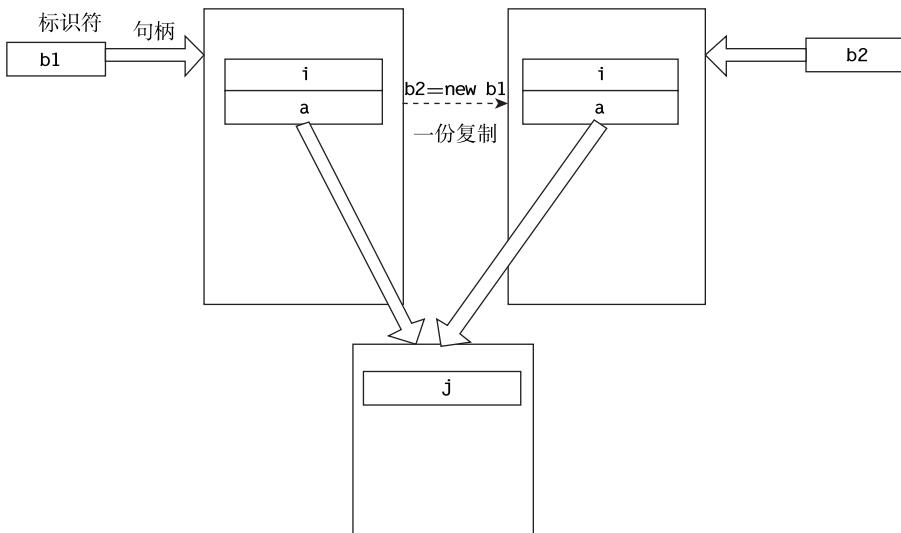


图 4-7 浅复制：只复制句柄不创建新的对象

另外需要注意：第一，类的属性和例化对象可以直接在类的声明中初始化；第二，浅复制不会复制对象（不会创建新的对象）；第三，在需要的时候实例限定可以被串接起来以便到达对象或穿越对象层次：

```

b1. a. j //访问到 a 对象
p. next. next. next. val //通过一系列句柄的串接来访问到 val

```

为了实现完整（深度）复制（或称为深拷贝），其中每一个数据成员（包括嵌套的对象）都要被复制，一般情况下需要使用者定制特殊的复制程序。如源代码 4-7 所示。自定义深复制的示意图如图 4-8 所示。

源代码 4-7 类的自定义深复制实例

```

//Chapter4 deep_copy.sv
module deep_copy();
...
class B;
    integer i=1;
    A a = new ;
    function copy(B in_b);

```

```

    this.i = in_b.i;
    this.a.j = in_b.a.j;
endfunction
endclass
initial begin
    b1 b1 = new ;
    b2 b2 = new ;
    b2.copy(b1);
end
endmodule

```

其中 copy (B in_b) 是一个定制的复制方法，它用来复制作为参数传递给它的对象实例化。

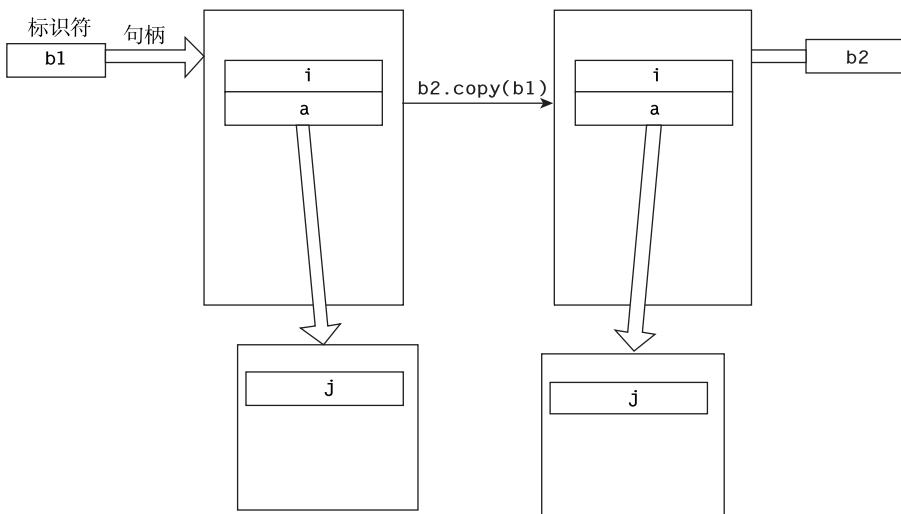


图 4-8 自定义深复制

4.2.6 块外声明

如果需要将方法定义放在类声明体的外部，那也是可以的。这可以通过两个步骤完成：首先，在类内声明方法的原型（无论它是任务还是函数）、任何限定符（local、protected 或 virtual）以及完整的参数说明加上 extern 限定符。extern 限定符指示方法体（它的实现）在类的外部定义。接着，在类声明体的外部定义完整的方法实现（与原型类似但没有限定符），并将这个方法绑定回它所属的类，通过使用类名和一对冒号（::）对方法名做限定。

```

class Packet;
    Packet next;
    function Packet get_next();
        get_next = next;
    endfunction
    // 块外(外部)声明

```

```

extern protected virtual function int send(int value);
endclass

function int Packet::send(int value);
// 将 protected virtual 关键字丢弃,用 Packet::做限定
...
endfunction

```

块外方法声明必须与它的原型声明完全一致；语法上的唯一不同点是方法名字前面加入了类名字和范围操作符（::）；块外方法声明必须在与类声明相同的作用范围内声明。

4.3 石头、剪刀、布仲裁器实例（基于类的验证平台）

首先，我们先介绍一个 rock-paper-scissor（RPS）arbitrator——石头、剪刀、布仲裁器的验证平台（paper 原意为纸，为了符合国人对该游戏的称谓，在此暂译成布，电影《非诚勿扰》中也有为此游戏创建的分歧终端机），如图 4-9 所示，该验证平台中有两个并行的 Stimulus Generator，分别代表 Player1/Player2，产生随机激励（rock/paper/scissor），Driver 接受到高层 Player1/Player2 发送过来的随机激励，通过物理信号（pin – level）按照一定的时序送入 DUT 中，其中 Checker 是断言模块，Monitor 是采集激励并将 DUT 的数据送往 Scoreboard 和 Coverage，Scoreboard 是记分板，Coverage 是做功能覆盖率统计模块；在这一章我们重点讨论如何采用类来封装激励单元、激励产生模块和其他验证组件，例如 BFM（总线功能模型）/Transactor（事务处理器）和记分板；Checker 和 Coverage 留待后续章节讨论。

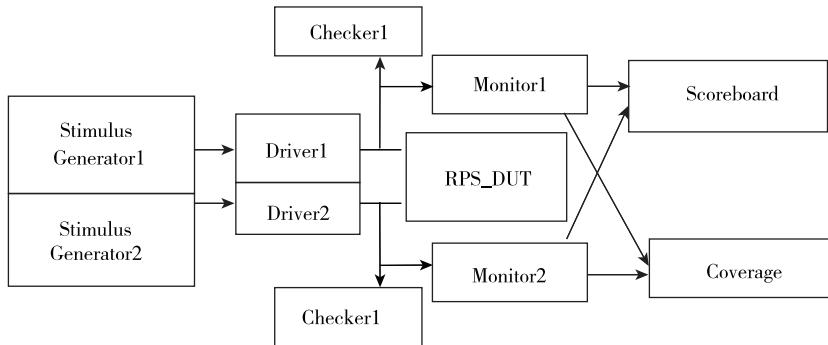


图 4-9 RPS 的验证平台

其中，DUT 的行为级源代码如源代码 4-8 所示，其主要实现接受两个 Stimulus Generator（Player）的激励并判断哪一个在本轮中取胜。score1/score2 表示 player1/player2 赢的次数。

源代码 4-8 仲裁器设计（石头、剪刀、布）

```

//Chapter4 rps_dut.sv
module rps_dut(
    input bit r1, p1, s1;           // r 为石头, p 为布(纸), s 为剪刀,三者互斥
    input bit r2, p2, s2;
    output int score1, score2;      //player1 或者 player2 赢的次数
    input bit go1, go2;
    input bit clk, rst, dut_busy);

```

```

bit win1, win2; // 哪位赢? 1 或者 2?
int tie_score;
assign both_ready = (go1 & go2); // 上升沿意味 DUT 开始计算
initial
    tie_score = 0;
always @ (posedge both_ready) begin
    #3;// Consume time
    win1 <= ((r1 & s2) |(s1 & p2) |(p1 & r2));
    win2 <= ((r2 & s1) |(s2 & p1) |(p2 & r1));
    if (win1)
        score1 <= score1 + 1;
    else if (win2)
        score2 <= score2 + 1;
    else
        tie_score <= tie_score + 1;
    dut_busy <= 1;
    @ (posedge c1k);
    dut_busy <= 0;
end
endmodule
-----
```

DUT 从输入的三个信号 r、p、s 中来判断该 player 发出的激励，r、p、s 在某一时刻只能有一个为高电平；player 1 和 player 2 通过剪刀、石头、布的游戏规则来判定哪位胜出，胜方 score 加一，打平 tie_score 加一，运算过程中 dut_busy 拉高，在一个时钟后计算结束拉低。

4.3.1 验证环境顶层

源代码 4-9 中用 top-down 的方式来研究整个验证平台的结构，top_class_based 是整个验证环境的顶层。

源代码 4-9 验证环境顶层（石头、剪刀、布）

```

//Chapter4 top_class_based.sv
module top_class_based;
import rps_env_pkg::*;
`include "rps_driver.sv"
`include "rps_monitor.sv"
`include "rps_env.sv"
bit r1,p1,s1;
bit r2,p2,s2;
int score1,score2;
bit go1,go2;
bit c1k,rst,dut_busy;

rps_clock_reset cr(.c1k(c1k),.rst(rst));

rps_dut
dut(.r1(r1),.p1(p1),.s1(s1),.r2(r2),.p2(p2),.s2(s2),.score1(score1),.score2(score2),
     .go1(go1),.go2(go2),.c1k(c1k),.rst(rst),.dut_busy(dut_busy));
```

```
// rps_dut dut(* );
rps_env env;

initial begin
    env = new();
    fork
        cr. run();
    join_none
    env. execute();
    $stop;
end
endmodule
```

如上所示，顶层模块内部例化了三个主要组件：rps_dut、rps_env 和 rps_clock_reset。其中，rps_dut 是石头、剪刀、布仲裁器设计（源代码 4-8）；rps_env 是一个用类定义的验证平台，封装了所有的验证组件；rps_clock_reset 是一个时钟和复位信号的产生模块。时钟复位模块代码如源代码 4-10 所示。

源代码 4-10 时钟复位模块（石头、剪刀、布）

```
//Chapter4 rps_clock_reset. sv
module rps_clock_reset(output bit clk,rst );
parameter bit ACTIVE_RESET=1;
task run(int reset_hold=4 ,int half_period=10 ,int count = 0 );
    clk = 0;
    rst = ACTIVE_RESET;

    for (int rst_i = 0;
        rst_i < reset_hold;rst_i++ ) begin
        #half_period;clk = ! clk;
        #half_period;clk = ! clk;
    end

    rst <= ~rst;
    // Run the clock
    for (int clk_i = 0;
        (clk_i < count || count == 0);
        clk_i++ ) begin
        #half_period;clk = ! clk;
    end
endtask
endmodule
```

从上面的代码可以看出，run 运行起来之后，rst 被复位为 1，在经过 reset_hold（默认值为 4）个时钟周期之后，撤销复位，rst 拉为 0；产生周期为 $2 \times$ half_period 的时钟；这个模块产生的信号可以供给 dut 和 rps_env 使用；这是一个相当独立的模块。

从顶层 top_class_based 模块中，我们可以看到程序起初就 import 了一个 rps_env_pkg 的 package；其内部用枚举定义了 rps_t 的用户自定义类型，分别代表着石头（ROCK）、剪刀

(SCISSORS)、布 (PAPER: 原意纸, 暂且翻译为布); 封装了一个 rps_c 的类, 这是激励单元, 内部包含 comp/clone/rps_randomize 三个方法; 封装了一个 stimulus_generator 的类, 是产生激励单元的验证组件。验证环境库文件 (石头、剪刀、布) 的代码如源代码 4-11 所示。

源代码 4-11 验证环境库文件 (石头、剪刀、布) —— 激励单元片段

```
//Chapter4 rps_env_pkg.sv
package rps_env_pkg;

typedef enum bit[1:0]
{IDLE, ROCK, PAPER, SCISSORS} rps_t;

class rps_c;
    rps_t rps;
    int score;

    function bit comp(input rps_c a);
        if(a.rps == this.rps)
            return 1;
        else
            return 0;
    endfunction

    function rps_c clone;
        clone = new();
        clone.rps = this.rps;
    endfunction

    task rps_randomize();
        bit [1:0] temp;
        temp = $urandom_range(3);
        $display("tmp is % b",temp);
        if(temp == 'b00) temp = 2'b01;
        case(temp)
            'b01:rps = ROCK;
            'b10:rps = PAPER;
            'b11:rps = SCISSORS;
        endcase
    endtask

endclass

function void report_the_play(
    string where, rps_c t1, rps_c t2);
    string str;
    $sformat(str,
        "(t1.rps,t2.rps) - Score1 =% 0d, Score2 =% 0d",
        t1.score, t2.score);
    $display("% s",str);
endfunction

class stimulus_generator;
    ...
endclass
```

```

class rps_scoreboard;
...
endclass

endpackage

```

从上面可以看出 rps_c 这个类中，采用 \$urandom_range 实现随机激励产生，在后面章节我们将不断优化该类，采用基于类的随机来实现同样的功能。

4.3.2 验证组件

验证组件（verification component）就是构造验证平台的基本功能单元，其中包括常见的激励生成器、事务处理器、记分板、控制单元、参考模型等。

1. 激励生成器

作为验证环境中的一个重要的验证组件，stimulus_generator 是一个激励产生器；从源代码 4-12 中我们可以看出，首先在 generate_stimulus 方法的内部是一个 forever 的循环体，每次产生一个激励都会运行一次 tmp = new（请大家记住这个方法）；若将 tmp = new 放在 forever 前面（外部），只执行一次 new，那将没有办法实现正常的功能；作为验证组件，其和外部的通信接口就是通过 mailbox 定义的 fifo，在调用 rps_randomize 对 tmp 做随机化之后，采用 fifo.put(tmp) 将其送入 mailbox 中，等待其他进程取出；后续我们可以看到与 stimulus_generator 通过 fifo 进行数据交互的就是 driver。

源代码 4-12 验证环境库文件（石头、剪刀、布）——激励生成器片段

```

//Chapter4 rps_env_pkg.sv
class stimulus_generator;
    mailbox #(rps_c) fifo;
    int id;
    bit stop=0;
    function new (int id_i);
        id=id_i;
    endfunction

    task generate_stimulus;
        rps_c tmp;
        forever begin
            if (stop == 0) begin
                tmp = new ;
                tmp.rps_randomize();
                $display("time:% 0d generator % 0d send out a stimulus:% s", $time,
                        id,tmp.rps);
                fifo.put(tmp);
            end
            else
                break;
        end
    endtask

```

```

task stop_stimulus_generation();
    stop = 1;
endtask

endclass

```

2. 事务处理器

介绍完了 rps_env_pkg 这个 package 中两个重要的类 rps_c 和 stimulus_generator 之后，我们来看在 top_class_based 模块中被 include (包含) 进来的 rps_driver.sv 文件；其中定义了两个类：rps_driver1 和 rps_driver2；两者的基本功能一致，区别在于 rps_driver1 驱动 rps_dut 中的 r1/p1/s1/go1 一组信号，而 rps_driver2 驱动了 r2/p2/s2/go2 一组信号。事务驱动器代码如源代码 4-13 所示。

源代码 4-13 事务驱动器（石头、剪刀、布）

```

//Chapter4 rps_driver.sv
class rps_driver1;
    mailbox #(rps_c) nb_get_port;
    rps_c transaction;

    task run;
        {top_class_based.r1, top_class_based.p1, top_class_based.s1} = 3'b000;
        top_class_based.go1 = 0;
        @ (negedge top_class_based.rst);
        forever @ (posedge top_class_based.clk)
            if (nb_get_port.try_get(transaction))
                begin
                    {top_class_based.r1, top_class_based.p1, top_class_based.s1}
                    = 3'b000;
                    $display("driver1 get a stimulus:%s",transaction.rps);
                    case (transaction.rps)
                        ROCK: top_class_based.r1 = 1;
                        PAPER: top_class_based.p1 = 1;
                        SCISSORS: top_class_based.s1 = 1;
                    endcase
                    top_class_based.go1 = 1;
                    @ (posedge top_class_based.clk);
                    top_class_based.go1 = 0;
                    @ (posedge top_class_based.clk);
                    {top_class_based.r1, top_class_based.p1, top_class_based.s1}
                    = 3'b000;
                end
            endtask
    endclass

    class rps_driver2;
        ...
        task run;
            ...
            case (transaction.rps)

```

```

    ROCK: top_class_based.r2 = 1;
    PAPER: top_class_based.p2 = 1;
    SCISSORS: top_class_based.s2 = 1;
endcase
...
endclass

```

driver 处于 stimulus_generator 和 rps_dut 之间，属于事务处理器（此处也可称为 BFM）；为了与 stimulus_generator 通信，其内部用 mailbox 定义了一个 nb_get_port；每次都尝试从该 mailbox 中取出一个数据放入 transaction 中；为了能够驱动 rps_dut，driver 采用层次化引用（top_class_based.xx）的方式对信号进行赋值，由于 driver1 和 driver2 驱动的信号不同，导致需要两份复制，后一章我们将介绍如何采用虚接口（virtual interface）实现 class 和 module 之间的连接，实现可重用化。

另外一个包含进来的是 rps_monitor.sv，其功能与 driver 恰恰相反，是从 rps_dut 的接口中搜集信号值，封装成一个 rps_c 的对象并送给记分板；由于监控的信号不同，其中包含了两个 monitor 的复制 monitor1 和 monitor2；和 driver 一样，monitor 与记分板的通信通过 mailbox 定义了 ap。监控器代码如源代码 4-14 所示。

源代码 4-14 监控器（石头、剪刀、布）

```

//Chapter4 rps_monitor.sv
class rps_monitor1;
    mailbox #(rps_c) ap;

    function rps_c pins2transaction;
        rps_c transaction = new ;
        case ({top_class_based.r1,top_class_based.p1,top_class_based.s1})
            3'b100: transaction.rps = ROCK;
            3'b010: transaction.rps = PAPER;
            3'b001: transaction.rps = SCISSORS;
        endcase
        transaction.score = top_class_based.score1;
        $display("time % 0d,monitor1 capture a transaction:% s", $time,transaction.rps);
        return transaction;
    endfunction

    task run;
        forever @ (posedge top_class_based.dut_busy)
            ap.put(pins2transaction());
        endtask
    endclass

    class rps_monitor2;
    ...
    case ({top_class_based.r2,top_class_based.p2,top_class_based.s2})
        3'b100: transaction.rps = ROCK;

```

```

3'b010: transaction.rps = PAPER;
3'b001: transaction.rps = SCISSORS;
endcase
...
endclass
-----
```

3. 验证平台和记分板

最后一个通过 include 进来的文件是 rps_env.sv，其中封装了所有验证组件的例化；两个 stimulus_generator 的例化 s1/s2；两个 driver 的例化 d1/d2；两个 monitor 的例化 m1/m2；一个记分板 rps_scoreboard 的例化 sb；另外还定义了四个传递 rps_c 数据类型的 mailbox：f1/f2/ap1/ap2，其中 f1/f2 深度为 2，分别作为 s1 → d1 和 s2 → d2 的数据同步；而 ap1/ap2 深度为无限，分别作为 m1 → sb 和 m2 → sb 的数据同步；采用 mailbox 作为两个进程之间同步的方法也有两种实现方式：stimulus_generator 和 driver 之间采用直接赋值的方式，而 monitor 和 scoreboard 之间，其中 scoreboard 采用在构造函数 new 中添加参数的方式传递。基于类的验证环境代码如源代码 4-15 所示。

源代码 4-15 基于类的验证环境（石头、剪刀、布）

```

//Chapter rps_env.sv
class rps_env;
    stimulus_generator s1, s2;
    mailbox #(rps_c) f1, f2, ap1, ap2;
    rps_driver1 d1;
    rps_driver2 d2;
    rps_monitor1 m1;
    rps_monitor2 m2;
    rps_scoreboard sb;

    function new ();
        s1 = new (1);
        f1 = new (2);
        d1 = new ;
        ap1 = new ;
        s2 = new (2);
        f2 = new (2);
        d2 = new ;
        ap2 = new ;

        m1 = new ;
        m2 = new ;

        sb = new (ap1, ap2);

        sb.limit = 10;

        s1 fifo = f1;
        s2 fifo = f2;
        d1.nb_get_port = f1;
```

```

d2. nb_get_port = f2;

m1. ap = ap1;
m2. ap = ap2;

endfunction

task execute;
    fork
        s1. generate_stimulus();
        s2. generate_stimulus();
        d1. run();
        d2. run();
        m1. run();
        m2. run();
        sb. run();
        terminate;
    join_any
endtask

task terminate;
    @ (posedge sb. test_done);
    s1. stop_stimulus_generation();
    s2. stop_stimulus_generation();
    $display("Test finished!");
endtask

endclass

```

从上述代码可以看出，任务 execute 并行地调用了 s1/s2/d1/d2/m1/m2/sb/terminate 八个进程，实现了从激励随机产生、驱动并激励 DUT、激励搜集，到记分板运算整个过程。在 sb. test_done 拉高之后整个运行程序结束，而 test_done 从源代码 4-16 中可以看出是由 limit 控制的。

记分板的工作原理（如源代码 4-16 所示）很简单，从 DUT 的输入输出引脚抽取信息：当前的激励信息和上次的设计运算结果；上次运算结果和记分板中的结果做比较，并报告是否运算正确，当前的激励信息供记分板做重新计算以获取预测和期望的结果。一旦 t1 或者 t2 获胜的次数达到 limit 之后把 test_done 拉高，实现程序的运行结束。

源代码 4-16 验证环境库文件（石头、剪刀、布）——记分板片段

```

//Chapter4 rps_env_pkg.sv
package rps_env_pkg
...
class rps_scoreboard;
    mailbox #(rps_c) fifo1, fifo2;
    rps_c t1, t2;
    int score1, score2, tie_score;
    int limit; //在配置的过程中初始化

```

```

reg test_done;//等待外部测试完毕

function new (mailbox #(rps_c) fifo1_i,fifo2_i);
    fifo1 = fifo1_i;
    fifo2 = fifo2_i;
    test_done = 0;
    score1 = 0;score2 = 0;tie_score = 0;
endfunction

task run;
    forever begin
        fifo1.get(t1);
        fifo2.get(t2);
        report_the_play("SBD", t1, t2);
        update_and_check_score();
    end
endtask

local function void update_and_check_score;
    string str;
    bit win1, win2;

    // 验证参考模型和设计对应得分
    if (score1 != t1.score) begin
        $sformat(str,
            "MISMATCH - score1 = % 0d, t1. score = % 0d",
            score1, t1.score);
        $display("SBD % s", str);
    end
    if (score2 != t2.score) begin
        $sformat(str,
            "MISMATCH - score2 = % 0d, t2. score = % 0d",
            score2, t2.score);
        $display("SBD % s", str);
    end

    // 计算新的得分
    win1 =
        ((t1.rps == ROCK && t2.rps == SCISSORS) |
        (t1.rps == SCISSORS && t2.rps == PAPER) |
        (t1.rps == PAPER && t2.rps == ROCK));

    win2 =
        ((t2.rps == ROCK && t1.rps == SCISSORS) |
        (t2.rps == SCISSORS && t1.rps == PAPER) |
        (t2.rps == PAPER && t1.rps == ROCK));

    if (win1)
        score1 += 1;
    else if (win2)

```

```

    score2 += 1;
else
    tie_score += 1;
// 检查是否测试完毕
if ((t1.score >= limit) ||
    (t2.score >= limit))
    test_done = 1;
$display("time:% 0d SBD compare sucessfully, score1: % 0d, score2:% 0d, tie_
score: % 0d", $time, score1,score2,tie_score);
endfunction
endclass
...
endpackage
-----
```

这个例子中我们没有涉及 checker 和 coverage，这两个部分设计断言和功能覆盖率，我们在后续的章节中再详细介绍。

通过本章的学习，有以下几点需要特别注意。

1) 类可以作为封装验证组件的载体。

通过类，我们可以封装激励单元（如 rps_c），激励产生器（如 stimulus_generator），事务处理器（如 driver/monitor），记分板（如 rps_scoreboard）和整个验证架构（如 rps_env）。

2) 并行进程之间可以通过 mailbox 来交互数据，实现同步。

验证平台的各个验证组件可以通过 fork...join 结构实现并行运行，并行进程之间可以通过 mailbox 来同步数据；而组件之间的 mailbox 可以有多种连接方式，这一点在 monitor -> rps_scoreboard 中可以看到 monitor 通过层次化对 ap 赋值，而对于 rps_scoreborad 采用构造函数进行传递。

3) 每个新的激励产生都要运行一次 new 构造函数，以避免同一个数据空间被随机多次，并且被多个对象的引用。

正确用法：

```

task generate_stimulus;
rps_c tmp;
forever begin
if (stop == 0) begin
    tmp = new ;
    tmp.rps_randomize ();
...

```

常见的错误用法：

```

task generate_stimulus;
rps_c tmp;
tmp = new ;
forever begin
if (stop == 0) begin
    tmp.rps_randomize ();
...

```

4) 时钟产生和复位可以独立在基于类的验证环境之外，为验证环境和 DUT 产生所需的复位信息和时钟。

剪刀、石头、布这个应用实例采用了最简单的基于类的结构来搭建验证平台，其中仍存在很多不完善的地方，例如：没有采用 virtual interface 导致 driver 和 monitor 的代码需要层次化引用，而且需要修改多份复制，而且缺乏重用性；没有采用基于类的随机，而是采用了 Verilog 传统的 \$urandom_range 的随机系统函数。

后面章节就是介绍虚接口和基于类的随机，介绍在基于类的验证环境中如何实现类和设计引脚的连接和可重用性；如何实现基于类的随机约束；最后我们也会简单介绍功能覆盖率、断言，把最后两个模块补齐。本书将围绕石头、剪刀、布仲裁器作为主线，通过逐步介绍以类为中心的 SystemVerilog 的各种语言特点，并应用到本例中，不断精练优化该验证环境。

虚 接 口

在第 2.4.2 节我们讨论了接口，其主要作用有两个：一是简化模块之间的连接；二是实现类和模块之间的通信。这两点都有助于设计和验证的重用。

本章我们讨论接口在验证方面的应用，重点讨论如何利用虚接口实现类和模块之间的通信，另外介绍一下接口中的几个重要的特性。

5.1 虚接口的基本概念及应用

设计中经常有多个相同功能或者标准协议的接口；这在网络设计中是常事，很多都是处理相同标准协议的接口，如以太包、SDH 等。上一章我们介绍的石头、剪刀、布这个例子中也出现了两个相同的接口，分别对应着 player1 和 player2。

在验证平台中，为两个相同功能的接口分别编写两个几乎一致的事务交易处理器（如 driver1 和 driver2 两个类），是一件烦琐的事情；但是，又只能如此处理，因为事务交易器中的方法采用了层次化应用的方式去访问对应端口的信号。

为了解决这个问题，SystemVerilog 提供了一个虚接口的语法结构，来实现方法（函数或者任务）可以对一个虚拟的端口进行操作。

虚接口和对应的方法可以将验证平台和设计分隔出来，保证其不受设计改动的影响。例如，假如设计引脚的名字发生改动，用户无须修改驱动这个端口的方法，而是只要在例化事务交易器的时候，给虚接口绑定对应连接的实体接口即可。

我们可以把虚接口看成对应接口类型的一个句柄（handle），可以在例化的时候执行具体的接口（接口的实体）。如图 5-1 所示为采用虚接口和没有采用虚接口的情况。

根据我们在前面学习到的类的基础知识可以知道，虚接口的句柄可以指向任意一个实体接口（非空对象），为此通用方法不会受到被测设计的任何约束，这样，我们可以把事务交易处理器的创建和设计分离出来，实现事务交易处理器的可重用性。

5.1.1 虚接口的基本概念

虚接口可以定义为类的一个成员，而且可以通过构造函数的参数或者过程初始化。这可以让相同的虚接口在不同的类中使用。虚接口的定义如下：

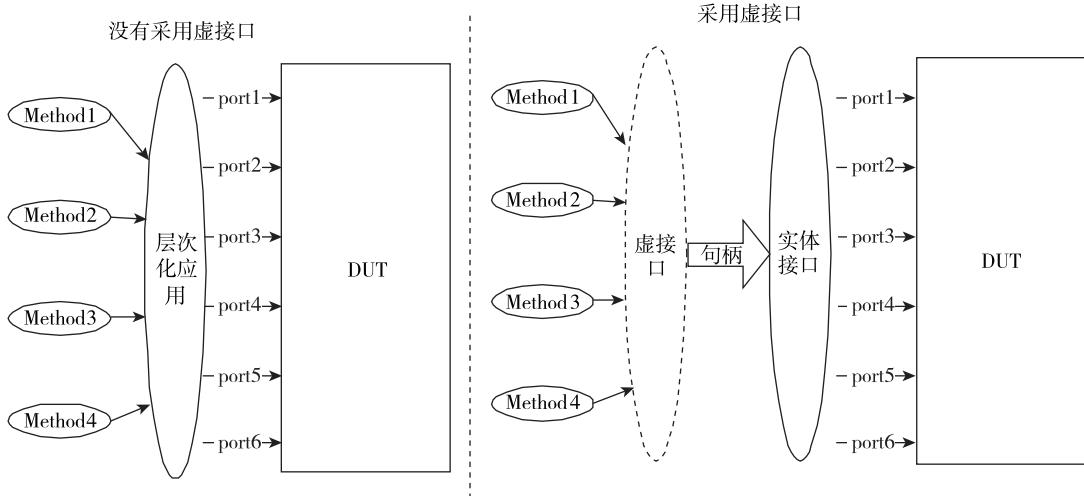


图 5-1 层次化应用与虚接口

```
virtual interface_type variable;
```

下面的例子使用虚接口的类和不同的器件实现连接和交互。如源代码 5-1 所示。

源代码 5-1 虚接口例子

```
//Chapter5 virtual_interface_example.sv
interface SBus;                                // 定义总线 SBus 接口——步骤 1
    logic req, grant;
    logic [7:0] addr, data;
endinterface

class SBusTransctor;                            // SBus 事务处理器
    virtual SBus bus;                          // SBus 类型的虚接口——步骤 2
function new(virtual SBus s);                  ——步骤 3
    bus = s;                                 // 初始化虚接口——步骤 4
endfunction

task request();                                // 请求获取总线控制权
    bus.req <= 1'b1;
endtask

task wait_for_bus();                           // 等待授权
    @(posedge bus.grant);
endtask
endclass

module devA( Sbus s ) ... endmodule // 使用 SBus 为接口的设计
module devB(input req,output grant,input [7:0] addr,data) ... endmodule

module top;
    SBus s[1:4]();                            // 例化 4 个接口——步骤 5
    devA a1( s[1] );                         // 例化 4 个设计——步骤 6A
    devB b1(.req(s[2].req),.grant(s[2].grant),.data(s[2].data),.addr(s[2].addr));
    //——步骤 6B
    devA a2( s[3] );
endmodule
```

```

devB b2(. req(s[4].req),. grant(s[4].grant),. data(s[4].data),. addr(s[4].addr) );

initial begin
    SbusTransactor t[1:4];
    t[1]= new( s[1] );           //——步骤 7
    t[2]= new( s[2] );
    t[3]= new( s[3] );
    t[4]= new( s[4] );
    // test t[1:4]
end
endmodule

```

在上面的例子中，事务处理器 SbusTransactor 是一个简单的可重用的验证组件。通过虚接口，方法可以不用全局或者层次化应用，不用关心具体哪个器件将会是交互的对象。另外，类也可以和任意数量的器件做交互，只要它们之间有一致的接口协议。

从上面例子，我们可以简单总结采用虚接口的七个步骤如下。

- 1) 定义接口（Sbus），该接口可供所有具有相同端口的模块或者类使用。
- 2) 在事务交易处理器的类中（Sbus_transactor）添加一个对应接口类型的（Sbus）虚接口成员（bus）。
- 3) 在事务交易处理器类的构造函数中，添加一个对应接口类型的虚接口的参数（s）。
- 4) 在事务交易处理器的构造函数体内将参数（s）赋值给虚接口成员（Sbus）。

注意：到此，我们就可以在事务交易处理器中，编写针对该接口的通用方法（如 request 和 wait_for_bus），只要针对虚接口进行操作即可，而该虚接口不针对特定的具体器件，只有在事务交易处理器的对象例化创建的时候，根据具体传给它的参数确定。

- 5) 在被测设计同一个层次例化实体接口（s[1:4]）。
- 6) 将被测设计（a1/b1/a2/b2）的例化端口和实体接口相连接（A/B 两种方式）。
- 7) 例化并创建事务处理器对象，并将实体接口作为参数传递给其例化。

在事务交易处理器中采用虚接口，遵循了其分离（类中的通用方法实现与具体端口分离）和可重用（例化并且传递不同的具体端口给对象的例化）的基本原则。

5.1.2 虚接口的应用

这一节，我们将虚接口的方法应用到上一章的石头、剪刀、布的例子中，我们将根据上一小节总结的七个步骤对程序做部分修改，使验证平台更加简练，更具可重用性。

步骤一：添加一个 interface.sv 的文件，定义两个 interface，如源代码 5-2 所示。

源代码 5-2 定义虚接口（石头、剪刀、布）

```

//Chapter5 interface.sv
interface rps_clk_if;
    bit clk, rst, dut_busy;
endinterface

interface rps_dut_pins_if(rps_clk_if clk_if);
    reg r, p, s;          // r 代表石头,p 代表布,s 代表剪刀
                           // 三者互斥

```

```

int score; // 该参与者胜出的次数
reg go; // go 的上升沿出现后表示可以进行仲裁
rps_t play;// 枚举类型,只在调试时使用

endinterface

```

步骤二、三、四：修改 driver 和 monitor 两个类的定义，采用虚接口作为方法的操作对象，如源代码 5-3 所示。

源代码 5-3 基于虚接口事务驱动器（石头、剪刀、布）

```

//Chapter5 rps_driver.sv
class rps_driver;
    virtual rps_dut_pins_if dut_vf;
    mailbox #(rps_c) nb_get_port;
    rps_c transaction;
    int id;
    function new(virtual rps_dut_pins_if dut_vf_i,int id_i);
        dut_vf=dut_vf_i;
        id=id_i;
    endfunction

    task run;
        {dut_vf.r, dut_vf.p,dut_vf.s}=3'b000;
        dut_vf.go=0;
        @ (negedge dut_vf.clk_if.rst);
        forever @ (posedge dut_vf.clk_if.clk)
            if(nb_get_port.try_get(transaction))
                begin
                    {dut_vf.r, dut_vf.p, dut_vf.s}
                    =3'b000;
                    $display("driver% 0d get a stimulus:% s",id,transaction.rps);
                    case(transaction.rps)
                        ROCK: dut_vf.r=1;
                        PAPER: dut_vf.p=1;
                        SCISSORS: dut_vf.s=1;
                    endcase
                    dut_vf.go=1;
                    @ (posedge dut_vf.clk_if.clk);
                    dut_vf.go=0;
                    @ (posedge dut_vf.clk_if.clk);
                    {dut_vf.r, dut_vf.p, dut_vf.s}
                    =3'b000;
                end
    endtask
endclass

```

源代码 5-4 是对 monitor 类的修改，过程与 driver 类似。

源代码 5-4 基于虚接口监控器（石头、剪刀、布）

```
//Chapter5 rps_monitor.sv
class rps_monitor;
    virtual rps_dut_pins_if dut_vf;
    mailbox #(rps_c) ap;
    int id;
    function new (virtual rps_dut_pins_if dut_vf_i,int id_i);
        dut_vf=dut_vf_i;
        id=id_i;
    endfunction
    function rps_c pins2transaction;
        rps_c transaction = new ;
        case ({dut_vf.r,dut_vf.p,dut_vf.s})
            3'b100: transaction.rps = ROCK;
            3'b010: transaction.rps = PAPER;
            3'b001: transaction.rps = SCISSORS;
        endcase
        transaction.score=dut_vf.score;
        $display("time %0d,monitor%0d capture a transaction:%s", $time, id, transaction.rps);
        return transaction;
    endfunction
    task run;
        forever @ (posedge dut_vf.clk_if.dut_busy)
            ap.put(pins2transaction());
    endtask
endclass
```

由于 driver 和 monitor 被封装在 rps_env 这个类中，为此我们需要把虚接口的参数逐层传递，如源代码 5-5 所示。

源代码 5-5 基于虚接口验证环境（石头、剪刀、布）

```
//Chapter rps_env.sv
class rps_env;
    stimulus_generator s1, s2;
    mailbox #(rps_c) f1, f2,ap1,ap2;
    rps_driver d1,d2;
    rps_monitor m1,m2;
    rps_scoreboard sb;

    task execute;
        fork
            s1.generate_stimulus();
            s2.generate_stimulus();
            d1.run();
            d2.run();
        join
    endtask
endclass
```

```

    m1. run();
    m2. run();
    sb. run();
    terminate;
    join_any
endtask

task terminate;
    @ (posedge sb. test_done);
    s1. stop_stimulus_generation();
    s2. stop_stimulus_generation();
    $display("Test finished!");
endtask

function new (virtual rps_dut_pins_if p1,p2);
    s1 = new (1);
    f1 = new (2);
    d1 = new (p1,1);
    ap1 = new ;
    s2 = new (2);
    f2 = new (2);
    d2 = new (p2,2);
    ap2 = new ;
    m1 = new (p1,1);
    m2 = new (p2,2);
    sb = new (ap1,ap2);
    sb. limit = 10;
    s1. fifo = f1;
    s2. fifo = f2;
    d1. nb_get_port = f1;
    d2. nb_get_port = f2;
    m1. ap = ap1;
    m2. ap = ap2;
endfunction
endclass

```

步骤五、六、七：在验证平台和被测设计的例化层次，实例化接口，做连接并通过参数传递给对应的构造函数，如源代码 5-6 所示。

源代码 5-6 基于虚接口的验证顶层（石头、剪刀、布）

```

//Chapter5 rps_tb_top.sv
module top_class_based;
import rps_env_pkg::*;
`include "interface.sv"
`include "rps_driver.sv"

```

```

`include "rps_monitor.sv"
`include "rps_env.sv"

rps_clk_if clk_if();
rps_clock_reset cr(.clk(clk_if.clk),.rst(clk_if.rst));

rps_dut_pins_if pins1_if(clk_if);
rps_dut_pins_if pins2_if(clk_if);

rps_dut
dut(.r1(pins1_if.r),.p1(pins1_if.p),.s1(pins1_if.s),.r2(pins2_if.r),.p2(pins2_if.p),.s2(pins2_if.s),.score1(pins1_if.score),.score2(pins2_if.score),.go1(pins1_if.go),.go2(pins2_if.go),.clk(clk_if.clk),.rst(clk_if.rst),.dut_busy(clk_if.dut_busy));

rps_env      env;

initial begin
  env = new(pins1_if,pins2_if);
  fork
    cr.run();
  join_none
  env.execute();
  $stop;
end

endmodule
-----
```

5.2 端口模式和时钟控制块

在此，我们顺便补充一些关于接口方面的知识，这一节讨论端口模式和时钟控制块。通过端口模式可以指定接口中信号的方向，而时钟控制块可以指定某些信号被某个时钟同步（采样和驱动）。

5.2.1 端口模式

为了限制在一个模块中对接口的访问，我们可以在接口内定义端口模式列表，通过采用关键字 modport，在其内部定义信号的方向。如源代码 5-7 所示。

源代码 5-7 端口模式实例

```

//Chapter5 interface_mode.sv
interface i2;
  wire a, b, c, d;
  modport master(input a, b, output c, d);
  modport slave(output a, b, input c, d);
endinterface
-----
```

在源代码 5-7 中，端口模式（master 或者 slave）可以被用来定义模块的端口，其定义了对应信号的方向。如下所示，模块 m 采用 i2. master，而模块 s 采用 i2. slave：

```

module m(i2.master i);
...
endmodule

module s(i2.slave i);
...
endmodule

```

在顶层连接的时候，我们可以通过实体接口作为传递对象，而例化可以自动查找自己对应的端口模式，如下所示，我们例化了实体端口 i () 并将其与 u1 和 u2 做连接：

```

module top;
  i2 i();
  m u1(.i(i));
  s u2(.i(i));
endmodule

```

interface_name. modport_name 的引用名可以作为层次化引用和访问，这个用法对于任何接口的端口模式都是通用的；而且实体接口的端口模式，其层次化引用也可以作为参数传递，如下例所示的 i. master 和 i. slave。

```

module m(i2 i);
...
endmodule

module s(i2 i);
...
endmodule

module top;
  i2 i();
  m u1(.i(i.master));
  s u2(.i(i.slave));
endmodule

```

当在模块的定义和例化都用端口模式的时候，要采用相同的端口模式的名字。注意，在端口模式中列出的信号必须和接口定义的信号名字一致，而不可以使用该接口未定义或者未引用的名字。

5.2.2 时钟控制块

在 Verilog 中，设计模块之间的通信通过端口来实现。SystemVerilog 添加了接口这个结构，接口能够封装模块之间的通信信号，用户可以方便地为模块间的通信抽象建模。接口定义了验证平台与被测设计进行通信的信号，然而接口并没有任何显式的时序规范、同步要求或时钟控制范例。

SystemVerilog 添加了时钟控制块（clocking block），它能够识别时钟信号，并能够获取该模块中指定的时序和同步要求。时钟控制块封装了同步于一个特定时钟的信号，并且为其显式地指定时序。时钟控制块是实现基于周期（cycle based）验证方法学中的一个关键元素，使得用户能够在一个更高的抽象层次上编写验证平台。除了关注于信号及其时间上的跳变外，

测试用例还可以在基于周期和事务交易的层次上去编写。根据应用环境的不同，一个验证平台可以包含一个或多个时钟控制块，每个都可以包含它自己的时钟和任意数目的信号。

时钟控制块将时序和同步细节与一个结构化、功能化的测试平台分隔开来。因此，采样和驱动时钟控制块信号的时序成为隐式的并且相对于时钟控制块的时钟。这使得我们能够以非常简洁的方式编写关键的操作，而无需显式地使用时钟或指定时序。这些操作是：

- 同步事件
- 输入采样
- 同步驱动

如下例所示，指定了与时钟 clk 相关的两个时钟控制模块：ck1 和 ck2。

```
clocking ck1 @ (posedge c1k);           //clock_event
    default input #1 step output negedge; //clock_skew
                                            //输出在 c1k 的下降沿被驱动
    input ...;
    output ...;
endclocking

clocking ck2 @ (clk); //没有指定沿
    default input #1step output negedge;
    input ...;
    output ...;
endclocking
```

@ (posedge clk) 和 @ (clk) 属于时钟事件，指定了作用于时钟控制块的触发条件。一般情况下，这个表达式是一个时钟信号的上升沿或下降沿。在一个指定的时钟控制块中说明的所有其他信号的时序都由这个时钟事件控制。时钟控制块中所有的输入或双向信号都在对应的时钟事件发生的时刻被采样。同样，时钟控制块中所有输出或双向信号都在对应的时钟事件发生的时刻被驱动。双向信号 (inout) 既被驱动也被采样。

时钟偏差指定了信号偏离时钟事件多少个时间单位被采样或驱动。输入偏差默认是负的，也就是说它们总是指向时钟事件之前的一个时间点，而输出则总是指向时钟事件之后的一个时间点，如图 5-2 所示。当时钟事件指定为一个简单的边沿而不是一个数值的时候，偏差可以采用该时钟信号的特定边沿。通过使用一个 default 来指定缺省为整个时钟控制块指定全局的时钟偏差。

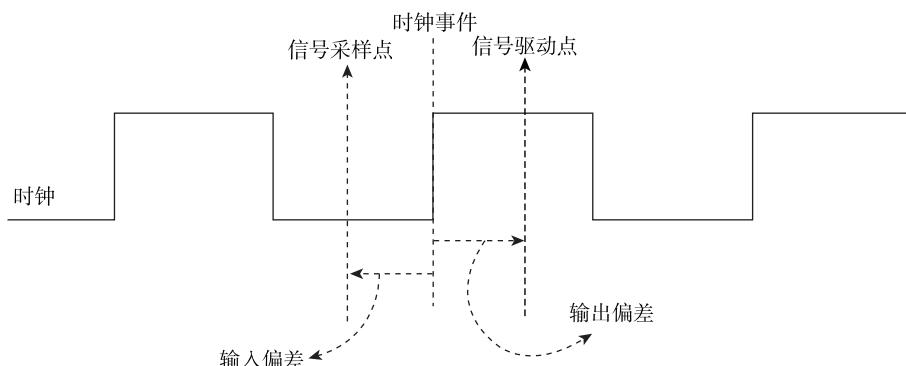


图 5-2 时钟事件与输入/输出偏差

在时钟控制块中，也可以采用层次化应用来说明与时钟控制块关联的信号的引用对象（跨模块引用），而不是一个本地端口。

```

clocking bus @ (posedge clock1);
  default input #10 ns output #2 ns ;
  input data, ready, enable = top.mem1.enable;
  output negedge ack;
  input #1 step addr;
endclocking

```

在上面的例子中，第一行代码声明了一个名为 bus 的时钟控制块，它以信号 clock1 的上升沿进行时钟控制。第二行代码通过 default 定义了：时钟控制块中的所有信号都应该使用相对于时钟事件 10ns 的输入偏差采样和 2ns 的输出偏差驱动。下一行代码为时钟控制块添加了三个输入信号：data、ready 和 enable；最后一个信号指向层次化的信号 top.mem1.enable。第四行代码为时钟控制块加入了一个 ack 信号，并修改了其缺省的输出偏差，从而使 ack 信号在时钟的下降沿被驱动。最后一行代码加入了 addr 信号并修改了其缺省的输入偏差，从而使 addr 在时钟上升沿之前的 1step 时被采样。

除非另有说明，缺省的输入偏差是 1step，缺省的输出偏差是 0。1step 是一个特殊的时间单位，它的值由时间精度来确定。1step 的输入偏差使得输入信号在时钟事件之前的时间步值中采样它们的稳定值（也就是在前一个 Postponed 区域）。与其他代表物理单位的时间单位不同，1step 不能被用来设置或修改时间精度或时间单位。

接口中可以采用一个时钟控制块来指定与某个时钟同步的信号。在时钟控制块中的信号会被同步采样或者驱动，以保证验证平台可以在正确的时间和被测设计进行交互。时钟控制块主要应用在验证平台，同时可以实现抽象层次的同步模型。

所有在端口模式中使用的时钟控制块必须在同一个接口中定义。和其他的端口模式定义一样，时钟控制块中的信号方向是从应用对象的角度来看的。下面的例子说明了如何采用端口模式来创建同步接口和异步接口，当和虚接口一起使用的时候，这种方法有助于同步抽象模型的创建。

```

interface A_Bus(input bit clk );
  wire req, gnt;
  wire [7:0] addr, data;
  clocking sb @ (posedge clk);
    input gnt;
    output req, addr;
    inout data;
    property p1;req ##[1:3] gnt;endproperty
  endclocking

  modport DUT(input clk, req, addr, // 被测设计的端口模式定义
              output gnt,
              inout data );

  modport STB(clocking sb); // 验证平台的同步端口模式定义
  modport TB(input gnt, // 验证平台的异步端口模式定义
             output req, addr,

```

```

    inout data );
endinterface

```

上面定义的接口 A_Bus 可以在下面的例子中例化，我们可以看出，program 中使用了时钟控制块定义的端口模式，也就是同步接口 b1 和 b2。除了根据时钟控制块的定义驱动和采样外，还对其中端口 b1 中的 p1 这个断言做检查，断言我们将在后续的章节介绍。

```

module dev1(A_Bus. DUT b); // 器件 1
...
endmodule

module dev2(A_Bus. DUT b); // 器件 2
...
endmodule

module top;
bit clk;
A_Bus b1( clk );
A_Bus b2( clk );
dev1 d1( b1 );
dev2 d2( b2 );
T tb( b1 , b2 );
endmodule

program T(A_Bus. STB b1 , A_Bus. STB b2 ); // 验证程序的两个同步接口
assert property (b1. p1); // 在程序内嵌入断言
initial begin
  b1. sb. req <= 1;
  wait ( b1. sb. gnt == 1 );
  ...
  b1. sb. req <= 0;
  b2. sb. req <= 1;
  wait ( b2. sb. gnt == 1 );
  ...
  b2. sb. req <= 0;
end
endprogram

```

随机测试

本章主要介绍随机激励产生和测试，我们将讨论如何产生随机激励以及基于随机测试的方法学；分析随机激励产生中和采用随机激励创建测试用例过程中潜在的问题。在介绍语法的同时，我们会将随机测试的方法应用到石头、剪刀、布这个验证平台中。

6.1 激励产生

激励产生是验证中很重要的一个组成部分，激励产生有下面几种可选的方法。

- 直接测试 (directed test)
- 直接随机测试 (directed random test)
- 随机测试 (random test)

每种激励产生策略所覆盖的测试空间的范围是不一样的。如图 6-1 所示，测试空间就如一个圆，其中任何一点或者一个区域代表被测设计中可能的功能或者参数。在一个复杂的设计中，测试空间可能由成千上万个功能点组成。一个直接测试击中的是测试空间中的某个特定的点，而随机测试则可以随机击中测试空间的任何一个点，直接随机测试可以随机的击中特定的某个区域。

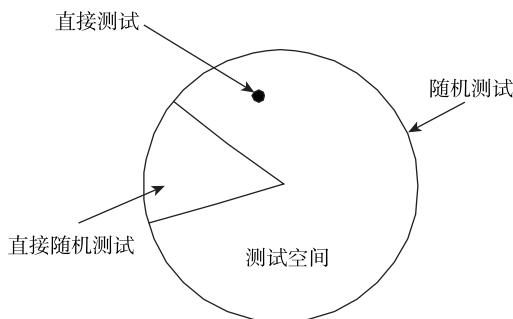


图 6-1 激励产生与测试空间覆盖率

直接测试比较适合简单的设计，因为其测试空间比较小，可以通过一定数目的测试即可覆盖到。直接测试也可以用来测试复杂设计中的简单功能点，特别是某些可能潜在缺陷的边

界条件；可以单独采用直接测试或者和其他测试混合在一起。例如，复杂设计中的奇偶错误检测的功能可以通过用直接测试检验，可以独立测试或者混合在顶层的随机测试中。

随机测试可以应用在任何类型的设计中，特别是在设计具有比较大的测试空间或者数量较多的交互情景。通过采用随机测试可以击中一些期望的功能点是很重要的一个功能，因为验证工程师未必都可以罗列出所有的功能点。随机测试可以创建一些异常并发或者异步的事件，从而测试设计中的某些独特、复杂的行为。

6.1.1 什么是随机

在计算机系统中，游戏算法和编码技术一直对随机有特殊的需求。一个顺序执行机器，如处理器，是一个确定的系统，它自动地执行指令。计算机行为的确定特性使得它们的数字序列是可预测的。对于真正的随机，无法凭借当前值来预测下一个值。计算机系统的确定性是很难实现一个随机的行为。

借助伪随机数产生器，计算机系统中引入随机。也就如其名字一样，伪随机数不是真正意义上的随机。然而通过数学公式或者一个预处理的列表，随机数是可以获取的。有很多研究伪随机算法的理论，现代算法对于生成伪随机数已经可以很接近真正的随机。

伪随机数是可以预测的，假如你知道序列的第一个数值。对于某些应用，可预测性也是很重要的特性。对于验证而言，可预测性是有好处的，因为它使得失败的测试用例可以在同样的条件下重新运行，从而重现特定的情景。对于仿真而言，一系列随机数可以在几个测试中用来重现，伪随机同样也可以实现。

复杂的芯片设计要求激励产生能够自动化，可以是确定的也可以是随机的。确定测试用例用来检验特定的功能或者参数。随机激励可以用来检查不同的功能覆盖点和减少测试用例的开发，但是它增加了验证平台的复杂性。SystemVerilog 采用伪随机数产生器来产生随机变量，从而产生随机激励。

6.1.2 潜在问题

采用随机测试过程中还存在需要折中考虑的方面。例如：

- 1) 相对于直接测试，随机测试很难调试。
- 2) 采用随机测试后，测试空间覆盖率需要一个可记录和衡量的标准。
- 3) 需要指定约束，从而随机生成的激励不会违反设计参数。
- 4) 相对于直接测试，采用随机测试的验证平台通常更加复杂，需要激励产生器、监控器等附加验证组件。

调试随机测试比调试直接测试要困难，其中有下面几种因素：验证平台更加复杂；随机测试一般都在长时间运行后发现设计缺陷，为此发现缺陷并重新运行测试增加了复杂度和延长了调试周期；随机激励生成器有其特定的实现机制。为了调试这些测试环境，验证工程师需要花额外的时间来理解整个交互过程是如何发生的。

对于随机测试，传统的调试过程可以通过日志文件或者监控器去追溯整个事务交易，也可以采用波形窗口等来作为辅佐的手段。

测试覆盖率时，因为随机测试是自动生成的，必须记录随机测试覆盖了哪些测试空间。

你可以在验证平台中通过自己的统计记录器或者 SystemVerilog 提供的覆盖率统计点来记录这些统计数据。在后面的章节我们将讨论如何采用 SystemVerilog 实现功能覆盖率的统计。

既然随机激励是自动生成的，它可能会违反设计参数，为了保证激励能够准确恰当的产生，SystemVerilog 提供了对随机产生进行约束的语法结构。约束为随机变量定义了有效范围。

一般来说，采用随机激励的验证平台比一般直接测试的要复杂。除了共同具有的功能之外，随机激励的验证平台还需要一些激励产生器、监控器和检查器等。

6.2 随机生成机制

SystemVerilog 有四种机制可以创建随机激励，实现随机数的产生和随机控制。

1) 采用 SystemVerilog 内置的系统函数来产生随机数，其中有 \$urandom 和 \$urandom_range 等，除此之外，还有一系列标准概率分布的系统函数：\$random、\$dist_uniform、\$dist_normal、\$dist_exponential、\$dist_poisson、\$dist_chi_square、\$dist_t 和 dist_erlang。

2) randcase 和 randsequence 结构来实现随机的分支选择。

3) 基于对象的随机生成，随机地初始化对象的数据成员的值。

4) 标准随机函数 std::randomize() 可以随时对任意变量进行随机化并添加约束。

上面每一种都可以用来作为随机的方法，这一节我们先介绍前面两种随机方法；在后面一节着重介绍基于对象的随机激励产生，这是 SystemVerilog 中随机激励产生的核心技术。

6.2.1 随机系统函数

系统函数 \$urandom 提供了生成伪随机数的机制。该函数返回一个 32 位的无符号数。其语法如下所示：

```
function int unsigned $urandom [(int seed)];
```

系统函数 \$urandom_range() 可以在指定的范围内返回一个无符号数。其语法如下所示：

```
function int unsigned $urandom_range( int unsigned maxval, int unsigned minval = 0 );
```

该函数会返回一个在 (maxval, minval) 之间的无符号整数，例如：

```
//Chapter6 random_sys_example.sv
val = $urandom_range(7,0);
```

在第 4 章的石头、剪刀、布仲裁器例子中，rps_c 激励单元类中的 rps_randomize 任务就是通过 \$urandom_range 的方式来生成随机数的。

系统函数 \$random 可以生成一个 32 位的伪随机数，该随机数是有符号数，为此可以是正数或者负数。

```
random_function ::= $random [(seed)]
```

除了上面提到的可以产生伪随机数的系统函数外，SystemVerilog 还提供了产生标准随机概率分布的系统函数：

```
dist_functions ::=
```

```
$dist_uniform( seed , start , end )
| $dist_normal( seed , mean , standard_deviation )
| $dist_exponential( seed , mean )
| $dist_poisson( seed , mean )
| $dist_chi_square( seed , degree_of_freedom )
| $dist_t( seed , degree_of_freedom )
| $dist_erlang( seed , k_stage , mean )
```

所有系统函数的参数都是整数，对于 exponential、poisson、chi-square、dist_t 和 dist_erlang 这几个系统函数，它们的参数 mean、degree_of_freedom 和 k_stage 要大于 0。每个系统函数将返回一个伪随机数，其遵从函数名所表示的随机分布，例如 \$dist_uniform 可以返回一个在指定范围内的均匀分布的随机数。详细的概率分布参数可以参考随机概率的书籍。

6.2.2 randcase/randsequence

randcase 关键字的引入，使得条件语句可以根据权重随机地选择执行某一条分支。每个分支的条目表达式是非负的整数，代表分支权重。每个分支的权重除以所有权重的和就是该分支被选中的概率。例如：

```
randcase
  3 : x = 1;
  1 : x = 2;
  4 : x = 3;
endcase
```

在这个例子中，所有权重的和为 8，所以第一条分支被选中的概率为 0.375，第二条分支被选中的概率为 0.125，第三条语句被选中的概率为 0.5。

如果一条分支的权重被指定了为 0，那么这条分支就不会被执行。如果所有的 randcase 分支的权重都被指定为 0，那么没有分支会被选中，仿真工具会发布一个警告信息。

randcase 的权重可以是任意表达式，而不仅仅是常量。randcase 实例如源代码 6-1 所示。

源代码 6-1 randcase 实例

```
//Chapter6 randcase_example.sv
...
byte a, b;
...
randcase
  a + b : x = 1;
  a - b : x = 2;
  a ^ ~ b : x = 3;
  12'b800 : x = 4;
endcase
...
```

除了随机分支选择（randcase）之外，SystemVerilog 还提供了随机序列生成（randsequence）。

程序解析器（例如 yacc），使用巴克斯范式（BNF）或类似的符号来描述被分析语言的语法；因此解析器能够检查程序是否采用该语言语法中正确的表达方式。SystemVerilog 的序列产生器（randsequence）从相反的方向描述了这个过程，它使用其语法来随机地产生正确的表达方式。随机序列产生器对于随机地产生结构化的激励序列（例如指令或网络流量模式）非常有用。

源代码 6-2 展示了 randsequence 的基本概念。

源代码 6-2 randsequence 实例

```
//Chapter6 randsequence_example.sv
randsequence (main)
  main : first second done;
  first : add | dec;
  second : pop | push;
  done : { $display("done"); };
  add : { $display("add"); };
  dec : { $display("dec"); };
  pop : { $display("pop"); };
  push : { $display("push"); };
endsequence
```

生成式 main 定义了三个非终端表达式（first、second 和 done 为非结束点）。当 main 被选中的时候，它产生序列 first、second 和 done。当 first 被产生的时候，它被分解成对应的生成式，在 add 和 dec 之间做随机选择。类似地，second 生成式在 pop 和 push 之间做选择。所有其他的生成式都是终端表达式（结束点），其代码块被完全指定，在这个例子中就是生成式名字。因此，该代码会导致下列可能的执行序列：

```
add pop done
add push done
dec pop done
dec push done
```

当 randsequence 语句被执行的时候，它产生一个基于语法驱动的随机生成式的序列。当每一个生成式被产生的时候，执行它的相关代码块并且产生了期望的激励。除了基本语法外，序列发生器提供了随机权重、交叉和其他的控制机制。尽管 randsequence 语句在本质上不会产生一个循环，然而一个递归的生成式会导致循环。如下面例子所示：

```
first : add := 3 | dec := (1 + 1);
```

上面的语句定义了一个 first 的生成式，由两个带权重的生成式 add 和 dec 组成。生成式 add 被选中的概率为 60%，而生成式 dec 被选中的概率为 40%。

randsequence 关键字可以跟着一个可选的生成式名字（在圆括号内）来指定顶层生成式。如果没有指定的话，第一个生成式即为顶层生成式。

6.3 基于对象的随机生成

本章前两节介绍了基于对象产生随机激励的基本概念和使用方法。SystemVerilog 使用一个

面向对象的方法来为对象的成员变量初始化随机值，它通过 rand 或者 randc 关键字来对数据成员进行定义。基于类的随机变量实例如源代码 6-3 所示。

源代码 6-3 基于类的随机变量实例

```
//Chapter6 class_random_example.sv
class Bus;
    rand bit[15:0] addr;
    rand bit[31:0] data;
    constraint word_align {addr[1:0]==2'b0;}
endclass
```

类 Bus 定义了一个简化的总线，它有两个随机变量：addr 和 data，分别代表总线上的地址和数据。以 constraint 关键字开头的 word_align 是约束块，其指定了：addr 的随机值必须是低两位为 0。

通过调用对象内置的 randomize() 方法便可以为一个总线对象产生新的随机值：

```
initial begin
    Bus bus_obj = new;
    repeat (50) begin
        if (bus_obj.randomize() == 1)
            $display("addr = % 16h data = % h\n", bus_obj.addr, bus_obj.data);
        else
            $display("Randomization failed.\n");
    end
end
```

每次调用 randomize() 都会重新为该对象中所有随机变量选择满足约束要求的随机值。在上面的程序中创建了一个总线对象 bus_obj，接着将该对象随机化 50 次。每一次随机化都会检查结果是否成功。如果随机化成功，那么会打印 addr 和 data 的新值；如果随机化失败，那么会打印一条错误信息。在这个例子中，仅仅约束了 addr 的值，而 data 的值未被约束。未被约束的变量被赋值为它们声明范围内的任意值。

从这个例子当中我们可以看出，基于对象的随机生成包含了三个部分：定义随机变量、指定约束条件（可选）、通过调用内置 randomize() 方法产生随机。下面我们将对各个主题进行详细介绍。

6.3.1 随机变量

类中的数据成员可以使用 rand 和 randc 关键字来声明为随机的变量。其语法如下：

```
class_property ::= {property_qualifier} data_declaraction
property_qualifier ::= rand | randc
```

使用 rand 关键字声明的变量是标准的随机变量。它们的值在其取值范围内均匀分布。例如：

```
rand bit [7:0] y;
```

这是一个 8 位的无符号整数，取值范围为 0 到 255。如果未被约束，该变量应该以相同的概率被赋值为 0 到 255 范围内的任意值。在本例中，连续地调用随机化产生相同值的概率为 1/256。

使用 randc 声明的变量是周期随机变量，它在声明范围内的一个随机排列中循环地选择所有的值。为了更好地理解 randc，我们来看看下面一个两位的随机变量 y：

```
randc bit [1:0] y;
```

变量 y 可以在 0、1、2 和 3 之间取值。随机化的过程会为 y 的取值范围计算一个初始的随机排列，然后再连续的调用中按顺序返回这些值。当返回一个排列的最后一个元素后，它会通过计算一个新的随机排列来重复这一过程。

randc 的基本思想是在取值范围内的所有值中随机地遍历，并且在一次遍历中不会有重复的值。当本次遍历结束后，会自动启动一个新周期的遍历。形象的例子就是扑克牌发牌的过程，每一轮都有 54 张随机序列的扑克牌，只有当 54 张牌发完以后才能进入下一轮的发牌过程，在这个过程中，同一张牌不能在本轮中重复出现两次。

在定义随机变量的时候需要注意的事项如下。

- 1) 解析器可以随机化任何整型类型的单一变量。
- 2) 数组可以声明成 rand 或 randc，在这种情况下，数组的所有成员元素都被当作 rand 或 rande 看待。
- 3) 动态数组和关联数组可以使用 rand 或 randc 声明。数组中的所有元素都被随机化，并会重写先前的任何数据。
- 4) 使用 rand 或 randc 声明的动态数组的大小也可以被约束。在这种情况下，数组应该根据大小的约束被重新调整，并且会接着随机化所有的数组元素。数组大小的约束如下所示：

```
rand bit [7:0] len;
rand integer data[];
constraint db {data.size == len;}
```

变量 len 具有 8 位的宽度。解析器会在 8 位的范围内（0 到 255）为 len 变量计算一个随机值，接着会随机化 data 数组的第一个 len 元素。如果一个动态数组的大小没有被约束，那么 randomize () 会随机化数组中的所有元素，而大小不会被改变。

- 5) 对象句柄可以使用 rand 声明，在这种情况下，该对象的所有变量和约束都被并发地求解，对象不能使用 randc 声明。

6.3.2 约束定义

在随机变量被定义后，该变量将在其数值范围内选取随机值。然而，有时候完全随机值是不可以的，例如一个合法的以太网的包长不能小于 64 字节，不能长于 1522 字节。SystemVerilog 可以通过约束块（constraint block）来限制随机变量的取值空间。随机变量的值由约束块中约束表达式来确定，像任务、函数和变量一样，约束块是类的成员，在一个类中，约束块的名字必须是唯一的。语法如下：

```
constraint constraint_identifier {{constraint_block_item}}
```

源代码 6-4 所示为随机约束块实例。

源代码 6-4 随机约束块实例

```
//Chapter6 constraint_example.sv
class ether_packet;
// 以太包字段
bit [55:0] preamble = 'h5555555555555555;
bit [7:0] sfd = 'hab;
rand bit [47:0] dest, src;
rand bit [15:0] len;
bit [7:0] payld [];

constraint packet_size_limit {len <=64;len <=1518};
...
endclass
```

SystemVerilog 的约束支持以下两种方式。

1) 指定取值范围，通过指定随机值的上限或者下限，或者某一个集合，或者某一特定值来限制其范围。

2) 分布式约束，为随机值的取值范围指定特定的权重，来影响其取值的概率分布。

下面我们讨论在约束块中使用的几种操作符和条件或者循环语句：inside，dist，->，if/else，foreach 及其应用。

1. inside 集合操作

约束支持使用 inside 操作符指定取值在整数值集合以及枚举集合。

如果没有其他的约束，inside 操作符在取值的时候，所有的可选值（包括单一值或范围值）都具有相等的概率。inside 操作符的否定形式如下所示：!(expression inside {set}) 表示其取值范围在表达式的集合范围之外。

inside 的应用例子如源代码 6-5 所示。

源代码 6-5 inside 操作实例

```
//Chapter6 inside_example.sv
...
rand integer x, y, z;
constraint c1 {x inside {3, 5, [9:15], [24:32], [y:2* y], z};}

rand integer a, b, c;
constraint c2 {a inside {b, c};}

integer fives[0:3] = {5, 10, 15, 20};
rand integer v;
constraint c3 {v inside fives;}
```

有一点我们必须注意：inside 操作符是双向的，因此，上述代码中的 {a inside {b, c};} 等价于 {a == b || a == c;}。

2. dist 分布操作

除了指定取值范围在某个集合内，约束也指定集合内不同的权重分布。关键字 dist 具有两个特性：对指定集合的范围进行关系测试；若为真，则为其指定统计分布权重。

```
expression dist {dist_list | value_range [: = |:/ dist_weight]};
```

如果表达式的值包含在集合中，那么分布操作符 dist 的计算结果为“真”；否则计算结果为“假”。分布集合为以逗号分隔的整型表达式和范围的列表。列表中的每一项都可以带有一个可选的权重，这个权重使用“:=”或“:/”操作符说明。如果没有为某一条目指定权重，那么缺省权重是：=1。权重可以是任意的整型表达式。

dist 的应用例子如源代码 6-6 所示。

源代码 6-6 dist 操作实例

```
//chapter6 dist_example.sv
...
int x;
constraint x_dist {x dist {100 := 1, 200 := 2, 300 := 5};}
```

上面的例子意味着 x 可以是 100、200 或 300，它们的权重比为 1:2:5。如果添加了另外的约束来说明 x 不能等于 200：

```
constraint x_dist {
    x != 200;
    x dist {100 := 1, 200 := 2, 300 := 5};
}
```

那么 x 只能等于 100 或 300，它们的权重比为 1:5。

当权重应用于范围的时候，它们可以应用到范围中的每一个值，或者它们可以将范围作为一个整体应用。例如：

```
constraint x_dist {x dist {[100:102] := 1, 200 := 2, 300 := 5} ;}
```

上面的例子意味着 x 可以等于 100、101、102、200 或 300 中的一个，并且它们的权重比为 1:1:1:2:5。

```
x dist {[100:102] :/ 1, 200 := 2, 300 := 5}
```

上面的例子意味着 x 可以等于 100、101、102、200 或 300 中的一个，并且它们的权重比为 (1/3):(1/3):(1/3):2:5。

当采用 := 操作符为一个条目指定一个权重时，如果条目是一个范围，它为范围中的每一个值指定权重。当采用 :/ 操作符为一个条目指定权重时，如果条目是一个范围，它会将范围作为一个整体指定权重；如果范围中有 n 个值，那么范围内每个值的权重为 range_weight/n。

采用 dist 为表达式或者变量指定分布的时候要注意下列限制。

- 1) dist 操作不可以应用到 randc 变量。
- 2) dist 表达式要求表达式至少包含一个 rand 变量。

3. 条件操作

约束提供了两种结构来声明条件操作：`->`（蕴涵操作）和`if...else`。

蕴涵操作符（`->`）可以用来声明隐含了一个约束的表达式。如下所示：

```
constraint a_b {a -> b;}
```

“`a -> b`” 蕴涵操作在布尔表达式上等价于“`! a || b`”。它表明：如果表达式为真，那么产生的随机数被其后的约束（或约束集合）所约束。否则，产生的随机数是未施加约束的。

具体应用如下所示：

```
constraint mode_len {
    mode == small -> len < 10;
    mode == large -> len > 100;
}
```

在这个例子中，`mode` 的值意味着：如果 `mode == small`，那么 `len` 的值应该被约束成小于 10；如果 `mode == large`，那么 `len` 的值应该被约束成大于 100；否则，`len` 的值是未约束的。

```
rand bit [3:0] a, b;
constraint c {(a == 0) -> (b == 1);}
```

无论是 `a` 还是 `b` 都是 4 位的，因此 `a` 和 `b` 的组合会有 256 种可能。约束块 `c` 表明，`a == 0` 意味着 `b == 1`，因此排除了 15 种组合：`{0, 0}`, `{0, 2}`, ..., `{0, 15}`。

SystemVerilog 还支持`if...else`类型的约束。

`if...else`类型的约束声明等同于蕴涵约束，例如：

```
constraint mode_len {
    if (mode == small)
        len < 10;
    else if (mode == large)
        len > 100;
}
```

等同于

```
constraint mode_len {
    mode == small -> len < 10;
    mode == large -> len > 100;
}
```

在这个例子中，`mode` 的值意味着 `len` 小于 10，或者大于 100，或者未约束。与蕴涵一样，`if...else`类型的约束也是双向的。在上面的声明中，`mode` 约束的值约束了 `len` 的值，并且 `len` 约束的值约束了 `mode` 的值。

4. foreach 迭代操作

迭代约束让我们能够使用循环变量和索引表达式以一种参数化的方式约束数组变量。具体使用语法在第 2 章我们详细介绍过。

`foreach` 结构指定了数组元素上的迭代。它的参数是一个标识符跟着一个包围在方括号

中的循环变量，这个标识符说明了任意类型的数组（固定尺寸、动态数组、联合数组或队列）。每一个循环变量均对应于数组的某一维。foreach 操作实例如源代码 6-7 所示。

源代码 6-7 foreach 操作实例

```
//chapter6 foreach_random_example.sv
class C;
    rand byte A[];
    ...
    constraint C1 {foreach(A[i]) A[i] inside {2,4,8,16};}
    constraint C2 {foreach(A[j]) A[j]>2*j;};
endclass
```

约束块 C1 将数组 A 的每一个元素约束成位于集合 [2, 4, 8, 16] 之内。约束块 C2 将数组 A 中的每一个元素约束成大于其索引值的两倍。

循环变量的数目不能超过数组变量的维数。

迭代约束可以包含判定条件。例如：

```
class C;
    rand int A[];
    constraint c1 {arr.size inside {[1:10]};}
    constraint c2 {foreach (A[k])(k < A.size - 1) -> A[k + 1] > A[k];}
endclass
```

第一个约束，c1 将数组 A 的尺寸约束在 1 到 10 之间。第二个约束，c2 将每一个数组值约束成大于前面一个的值，也就是数组按降序排列。

动态数组或关联数组的 size 方法可以被用来约束数组的大小（参见上面例子中的 c1）。如果一个数组既被大小约束所约束，也被迭代约束所约束，那么大小约束首先被解析，然后才是迭代约束。作为大小约束和迭代约束间隐含排序的结果，size 方法应该被当作对应数组的 foreach 块内的状态变量。例如，表达式 A.size 被当作约束 c1 中的一个随机变量，并被作为约束 c2 中的一个状态变量。

5. solve...before 操作

解算器必须确保选择的值在有效的值组合中具有均匀的值分布（也就是说，所有的有效值组合在求解过程中具有相等的概率）。这个重要的特性能保证所有的值组合的可能性是相等的，这也就使得随机化能够更好地探索整个设计空间。

然而，有时我们希望强制某种组合出现的频率更高一些。考虑下面一个例子，其中 1 位的控制变量 s 约束了一个 32 位的数据值 d：

```
class B;
    rand bit s;
    rand bit [31:0] d;
    constraint c {s -> d == 0;};
endclass
```

约束块 c 表明“s 蕴涵着 d 等于 0”。尽管表面上看好像由 s 确定 d，但事实上 s 和 d 是同时确定的。 $\{s, d\}$ 共有 2^{33} 种组合，但 s 仅在 $\{1, 0\}$ 这种组合时才为真。因此 s 为真的

可能性为 $1/2^{33}$ ，它近似于等于 0。

约束为排序变量提供了一种机制，这样 s 可以独立于 d 被选择。这种机制在变量的求解中定义了一种排序，并使用 solve 关键字说明。solve...before 操作实例如源代码 6-8 所示。

源代码 6-8 solve...before 操作实例

```
//Chapter6 solve_before_example.sv
...
class B;
  rand bit s;
  rand bit [31:0] d;
  constraint c {s -> d == 0;};
  constraint order {solve s before d;};
endclass
```

在这个例子中，order 约束块向解算器指示 s 在 d 被求解之前求解。这样处理的效果就是 s 具有 50% 的概率被选择为真，接下来 d 根据 s 的值被选择。相应地， $d == 0$ 的概率为 50%， $d != 0$ 的概率也为 50%。

变量排序可以被用来强制某些边界条件出现的机会更频繁一些。然而，一个“solve...before ...”约束不会改变求解空间，并且不会引起解算器失败。

下面是有关约束的几个重要特性。

- 约束可以是任何带有整型类型（bit、reg、logic、integer、enum、压缩结构体等）的变量和常量表达式。
- 约束解算器必须能够处理多种等式，例如代数因式分解、复杂的布尔表达式以及混合整数和位的表达式。
- 如果约束存在一个解的话，那么约束解算器必须找到这个解。只有在问题被过度约束并且没有随机值的组合能够满足约束的时候，解算器才可以失败。
- 约束双向的影响，并发求解。所有的表达式操作符都被双向地对待，包括蕴涵操作符 (\rightarrow)。
- 约束仅仅支持两态值。四态（X 或 Z）或四态操作符（例如， $==$ 、 $!=$ ）是非法的，并且会导致错误。

由于本章讨论的约束是基于对象的，为此，类的继承过程中涉及约束问题。具体继承的概念留待下一章详细解释，这里我们介绍约束在继承过程可能出现的问题。

对于继承而言，约束与类变量、任务和函数遵从相同的规则：

- 如果一个派生类（子类）中的约束与其基类（父类）中的约束具有相同的名字，那么它会改写基类约束。例如：

```
class A;
  rand integer x;
  constraint c {x < 0;};
endclass
class B extends A;
```

```

constraint c {x > 0;};
endclass

```

类 A (基类或父类) 的一个实例包含了 x 小于 0 的约束，而类 B (扩展类或子类) 的一个实例包含了 x 大于 0 的约束。扩展的类 B 改写了约束 c 的定义。为此，约束与虚方法具有相同的特性，所以将 B 的实例强制转换到 A 不会改变约束设置。

- randomize() 任务是虚方法。因此，它以一个虚方法的属性来对待类的约束。当一个命名的约束在扩展类中被重新定义的时候，基类中的定义会被子类中的改写。

下一章我们会详细讨论类的继承、重写 (override: 改写) 和虚方法等概念。

6.3.3 随机方法

对象中的变量使用 randomize() 方法进行随机化。每个类都有一个内建的 randomize() 虚方法，它的声明原型如下：

```
virtual function int randomize();
```

randomize() 方法是一个虚函数，它为对象中的所有激活的随机变量产生随机值，产生的随机值应该符合激活的约束。

如果 randomize() 方法成功地设置了所有的随机变量和对象的有效值，那么它返回 1；否则返回 0。

例子：

```

class SimpleSum;
  rand bit [7:0] x, y, z;
  constraint c {z == x + y;};
endclass

```

这个类定义声明了三个随机变量 x、y 和 z。下面是一个例子，展示了调用 randomize() 方法对类 SimpleSum 进行随机化的一个实例。

```

initial begin
  SimpleSum p = new ;
  int success = p. randomize();
  if (success == 1) ...
end

```

检查返回状态是很有必要的，因为状态变量或继承类中约束的增加都会导致表面上简单的约束不再满足。

每一个类都有内置的 pre_randomize() 和 post_randomize() 函数，它们在计算新的随机值之前和之后被 randomize() 自动调用。

pre_randomize() 的内建定义如下：

```

function void pre_randomize;
  if (super) super. pre_randomize(); // 检测 super 以便确定对象句柄是否存在
  // [可选]在此编写在随机化 randomize() 之前运行的程序
endfunction

```

`post_randomize()`的内建定义如下：

```
function void post_randomize();
    if (super) super.post_randomize(); // 检测 super 以便确定对象句柄是否存在
    // [可选]在此编写在随机化 randomize()之后运行的程序
endfunction
```

当调用 `obj.randomize()` 的时候，它首先调用 `obj` 的 `pre_randomize()` 方法以及它的所有被使能的随机对象成员。接下来 `pre_randomize()` 会调用 `super.pre_randomize()`。在新的随机值被计算并赋值后，`randomize()` 会调用 `obj` 的 `post_randomize()` 方法以及它的所有被使能的随机对象成员。接下来 `post_randomize()` 会调用 `super.post_randomize()`。

用户可以在任何类中重写 `pre_randomize()` 方法以便执行对象被随机化之前的初始化和预处理。用户可以在任何类中重写 `post_randomize()` 方法以便执行对象被随机化之后的清除、打印诊断以及后处理。

如果这些方法被重写，那么它们必须先调用对应的父类方法，否则它们在随机化之前和之后的处理步骤会被忽略。

`pre_randomize()` 和 `post_randomize()` 方法不是虚拟的。然而，因为它们会被虚拟的 `randomize()` 方法自动调用，所以它们看上去像是虚拟的。

使用随机化方法 `randomize()` 需要注意如下事项。

- 被声明成 `static` 的随机变量被随机变量在其中声明的类的所有实例所共享。每次调用 `randomize()` 方法的时候，在每一个类实例中的变量都会被改变。
- 如果 `randomize()` 失败，那么约束是不可实行的并且变量保持为原来的值。
- 如果 `randomize()` 失败，`post_randomize()` 方法不会被调用。
- `randomize()` 方法是内置并且不能被重写的。
- `randomize()` 方法实现了对象的随机稳定性。一个对象可以通过调用它的 `srandom()` 方法来设置种子。
- `pre_randomize()` 和 `post_randomize()` 内置方法是函数并且不能被阻塞。

6.3.4 随机使能控制

一般情况下，所有定义的随机变量和约束默认都是被激活。但是，在运行的过程中我们可以通过使用内置方法 `rand_mode()` 和 `constraint_mode()` 对随机变量的状态和约束进行控制、关闭或者激活。

1. 随机变量使能控制

`rand_mode()` 可以用来控制激活或关闭一个随机变量。当一个随机变量处于未激活状态的时候，这个随机变量就好像没有使用 `rand` 或 `randc` 声明一样。未激活的随机变量不会被 `randomize()` 方法随机化，并且它们会被解算器当作一般的状态变量。所有的随机变量最初都是激活的。

`rand_mode()` 方法的语法如下：

```
task object[.random_variable]::rand_mode(bit on_off);
```

或者：

```
function int object.random_variable::rand_mode();
```

object 是对象句柄，并且随机变量在其中定义。random_variable 是操作被应用的随机变量的名字。如果没有指定随机变量的名字（仅在作为任务调用时才允许出现这种情况），那么这个动作会被应用到对象内的所有随机变量。

当作为任务调用时，rand_mode 方法的参数确定了执行的操作。rand_mode 参数如表 6-1 所示。

表 6-1 rand_mode 参数

值	状态	描述
0	OFF	将指定的随机变量设置为未激活状态。这样，它们不会被后来调用的 randomize () 方法随机化
1	ON	将指定的随机变量设置为激活状态。这样，它们能够被后来调用的 randomize () 方法随机化

对于非压缩数组变量，random_variable 可以使用索引来指定数组中的单个成员。省略索引会导致数组中的所有成员都被调用影响。如果随机变量是一个对象句柄，那么仅仅该变量的模式被改变，而该句柄对象内的随机变量的模式不会改变。如果指定的变量在类层次中不存在或者它虽然存在但没有使用 rand 或 randc 声明，那么编译器应该发布一个错误信息。

如果作为一个函数调用，那么 rand_mode() 返回指定变量当前的激活状态。如果随机变量处于激活状态（ON），则返回 1；如果随机变量处于未激活状态（OFF），则返回 0。rand_mode() 方法的函数形式仅能接受单一变量，因此，如果指定的变量是一个非压缩数组，那么必须使用索引来选择单一的元素。

具体应用例子如源代码 6-9 所示。

源代码 6-9 随机变量使能模式实例

```
//Chapter6 rand_mode_example.sv
class Packet;
    rand integer source_value, dest_value;
    ... // 其他声明
endclass
...
initial begin
    int ret;
    Packet packet_a = new ;
    // 关闭对象中的所有随机变量
    packet_a.rand_mode(0);
    ... // 其他代码
    // 使能 source_value
    packet_a.source_value.rand_mode(1);
    ret = packet_a.dest_value.rand_mode();
    ...
end
```

这个例子首先关闭对象 packet_a 中的所有随机变量，然后仅仅使能 source_value 变量。最后它将 ret 变量设置为变量 dest_value 的激活状态。注意，rand_mode() 方法是内置的，并且不能被重写。

2. 随机约束使能控制

默认情况下，所有的约束都是激活的。constraint_mode() 方法可以用来控制激活或关闭一个约束。当约束处于未激活（关闭）状态时，它不会被 randomize() 方法采用。

constraint_mode() 方法的语法如下：

```
task object[. constraint_identifier]::constraint_mode(bit on_off);
```

或者

```
function int object. constraint_identifier::constraint_mode();
```

当作为任务调用的时候，constraint_mode 方法的参数确定了需要执行的操作，constraint_mode 参数如表 6-2 所示。

表 6-2 constraint_mode 参数

值	状态	描述
0	OFF	将指定的约束块设置成未激活状态，这样它就不会被后来对 randomize() 方法的调用所强制
1	ON	将指定的约束块设置成激活状态，这样它会被后来对 randomize() 方法的调用所考虑

如果指定的约束块在该类中不存在，那么编译器会发布一条错误信息。当作为函数调用的时候，constraint_mode() 方法返回指定约束块当前的激活状态。如果约束处于激活状态（ON），那么它返回 1；如果约束处于未激活状态（OFF），那么它返回 0。

应用例子如源代码 6-10 所示。

源代码 6-10 随机约束使能模式实例

```
//Chapter6 constraint_mode_example.sv
...
class Packet;
    int m = 100;
    rand integer source_value;
    constraint filter1 {source_value > 2 * m;};
endclass

function integer toggle_rand(Packet p);
    if (p.filter1.constraint_mode())
        p.filter1.constraint_mode(0);
    else
        p.filter1.constraint_mode(1);

    toggle_rand = p.randomize();
endfunction
```

在这个例子中，`toggle_rand` 函数首先检查指定的 `Packet` 对象 `p` 中约束 `filter1` 当前的状态。如果约束处于激活状态，函数会将它设置为处于非激活状态；如果约束处于非激活状态，那么函数会将它激活。最后，函数会调用 `randomize` 方法来为随机变量 `source_value` 产生一个新的随机值。

需要注意的是：`constraint_mode()` 方法是内建的并且不能被重写。

6.3.5 约束的动态修改

SystemVerilog 中提供了如下几种方法来动态地修改随机化方法。

- 蕴涵 (`->`) 以及 `if...else` 允许声明具有判决条件的约束。
- 可以使用 `constraint_mode()` 内建方法来激活或关闭约束块。初始情况下，所有的约束块均处于激活状态。未激活的约束会被 `randomize()` 函数忽略。
- 可以使用 `rand_mode()` 内建方法来激活或关闭随机变量。初始情况下，所有的 `rand` 和 `randc` 变量都处于激活状态。未激活的随机变量会被 `randomize()` 函数忽略。
- 在 `dist` 约束中的权重可以被改变，从而影响了被选定集合中特定值出现的概率。

另外，我们可以在调用 `randomize` 的时候通过 `with` 加以约束。

通过使用 `randomize() ...with` 结构，用户可以在 `randomize()` 方法的调用点上声明内联约束。这些额外的约束与对象约束共同起作用。

例如：

```
class SimpleSum
  rand bit [7:0] x, y, z;
  constraint c {z == x + y;};
endclass
task InlineConstraintDemo(SimpleSum p);
  int success;
  success = p.randomize() with {x < y;};
endtask
```

这个简单的例子在前面章节已经使用过，然而在这里我们使用 `randomize() ...with` 来引入一个额外的 `x < y` 的约束。

`randomize() ...with` 结构可以出现在表达式可以出现的任何地方。紧跟在 `with` 之后的约束块可以包含在类中声明的所有相同的约束类型和格式。`randomize() ...with` 约束块还可以引用本地变量以及任务和函数的参数。

6.4 标准随机函数

上面我们讨论了基于类的随机：从随机变量定义、随机约束到随机函数的调用。使用类对需要随机化的数据建模是一种强大的机制，它使得我们能够产生通用、可重用并带有随机变量和约束的对象，这些特性可以在后续进行扩展、继承、重写、使能、关闭以及与其他对象合并或分离。类的这一特性使得其非常适合用来描述和处理随机数据和约束。然

而，某些要求较少的问题并不需要采用类来建模，它可以使用一个较为简单的机制来随机化不属于一个类的数据。标准随机函数（`std::randomize()`）使得用户能够随机化当前范围内的数据，而无需定义一个类或实例化一个类对象。

标准随机函数的语法如下：

```
scope_randomize ::=  
[std::]randomize([variable_identifier_list]) [with constraint_block]
```

标准随机函数与类的随机方法的作用相同，只是它仅限于操作当前范围内的变量而不是类成员变量。函数的参数指定了那些需要赋值为随机值的变量，也就是随机变量。标准随机函数实例如源代码 6-11 所示。

源代码 6-11 标准随机函数实例

```
//Chapter6 std_randomize_example.sv  
module stim;  
    bit [15:0] addr;  
    bit [31:0] data;  
  
    function bit gen_stim();  
        bit success, rd_wr;  
        success = randomize(addr, data, rd_wr); // 调用 std::randomize  
        return rd_wr;  
    endfunction  
  
    ...  
endmodule
```

函数 `gen_stim` 调用 `std::randomize()` 函数并带有三个变量作为参数：`addr`、`data` 和 `rd_wr`。`std::randomize()` 将新的随机变量赋值到那些在 `gen_stim` 函数范围内可见的变量。注意，`addr` 和 `data` 具有模块级的作用范围，而 `rd_wr` 具有函数内的本地作用范围。

6.5 随机激励的应用

在学习完随机激励产生的基础知识后，我们把基于类的随机应用到前面石头、剪刀、布这个例子当中。

首先修改激励单元 `rps_c` 这个类，将其数据成员定义为随机变量，并且添加约束使其不产生无效的激励；删除 `rps_randomise` 这个任务，因为类自带随机函数。修改后的代码如源代码 6-12 所示。

源代码 6-12 基于类的随机激励实例（石头、剪刀、布）

```
//Chapter6 rps_env_pkg.sv  
class rps_c;  
    rand rps_t rps;  
  
    int score;
```

```

constraint illegal {rps != IDLE ;}

function bit comp(input rps_c a);
    if (a.rps == this.rps)
        return 1;
    else
        return 0;
endfunction

function rps_c clone;
    clone = new();
    clone.rps = this.rps;
endfunction

endclass
-----
```

下一步就是修改激励产生器 stimulus_generator 中的随机函数的调用，把 tmp.rps_randomize() 改为内置的 tmp.randomize()。

```

class stimulus_generator;
    mailbox #(rps_c) fifo;
    int id;
    bit stop = 0;
    function new(int id_i);
        id = id_i;
    endfunction

    task generate_stimulus;
        rps_c tmp;
        forever begin
            if (stop == 0) begin
                tmp = new();
                tmp.randomize();
                fifo.put(tmp);
            end
            else
                break;
        end
    endtask

    task stop_stimulus_generation();
        stop = 1;
    endtask

endclass
-----
```

从上面可以看出，采用基于类的随机激励生成可以方便地实现随机变量的定义、约束的添加和随机生成，从而加速了验证平台的搭建和测试用例的创建。

继承与多态

这一章主要介绍 SystemVerilog 这门面向对象编程语言中的继承和多态，其中涉及了类的扩展、类成员的重写、虚类、抽象类、虚方法和多态等概念。我们可以借助面向对象编程语言的这些功能实现代码的可重用性。本章将从基本概念开始介绍，并且赋以实例，让大家对其应用有比较深刻的了解。

7.1 继承和多态的基本概念

在验证的过程中，经常会有新的测试用例被添加进来，而且需要对原有的验证平台进行改进，特殊的测试用例经常会要求验证平台添加特定的功能。在过程化编程语言中，这要求修改原来的代码并且有可能修改原有的数据结构，这个过程可能会在原有的验证环境中引入意外的错误。面向对象编程语言提供了一个高层次的重用功能：通过对已有的基类引申或者扩展，生成新的子类。

从基类做扩展并创建新的派生类（子类）的过程，就是类的派生。当一个类被扩展并创建了派生类（子类）之后，该派生类继承了其基类的数据成员、属性和方法，这就是类的继承。

所谓多态，也就是可以呈现多个“形状”（功能）的能力。在面向对象语言中，一个新的派生类被创建后，其中基类中的某些方法可以通过重写（override）被重定义。这个过程就是重写。当一个对象调用了一个被重写的方法，对象的类型将决定调用方法的实现方式。通常，这是一个动态的过程，动态的选择方法的实现方式就是多态。

SystemVerilog 支持类的继承和多态。一个类可以被扩展、继承或者重写其方法，而被调用方法在多态的情况下能够动态的选择实现方式。后面我们将一一介绍各个基本概念。

7.2 继承与子类

当一个类被扩展产生一个派生类的时候，派生类（子类）继承了基类（父类）的所有属性。扩展同时可以实现下列功能。

1) 修改类中原有的方法。例如，一个队列基类有一个入队的函数，其实现了添加一个成员到队列的后面。一个派生类可以在扩展的时候通过重新定义（重写）该函数，例如入

队的时候通过排序的方式。这种重写定义和修改函数或者任务的过程就是重写。

2) 添加新的方法。例如给定一个队列基类，你可以在扩展的时候添加新的方法，以实现查看队列里面的所有成员。

3) 添加新的数据成员。例如给定一个长方形的基类，派生类中可以添加数据成员来表示内部颜色和各边的颜色。

我们采用一个多边形的例子来说明这些概念。其中基类是多边形，该类中有数据成员如面积、边的数量、每条边的长度和周长；另外有计算面积和周长的方法。在继承过程中，每个派生类都重写了面积和周长的计算公式，而且添加了如何绘制该类图形的方法。如图 7-1 所示。

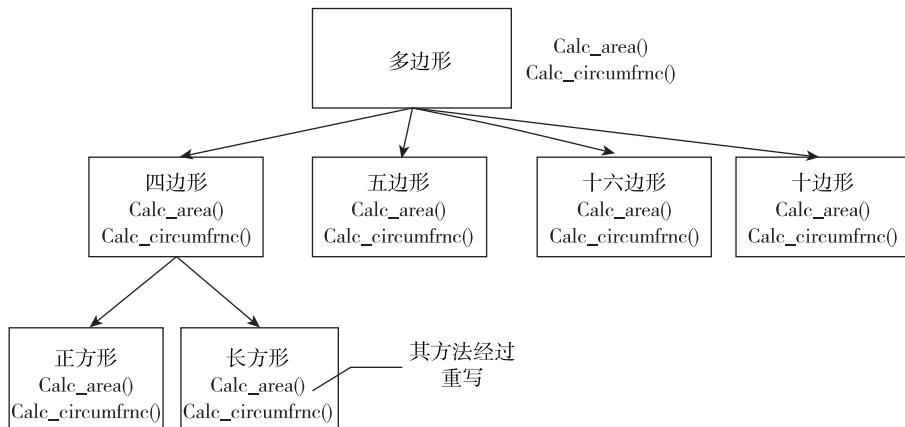


图 7-1 多边形的派生与继承

类似地，假如你开发了一个基本的验证平台（用类封装），你可以从该基类中引申出新的类，并添加或者修改其中的功能，从而创建了一个派生类——新的验证平台，但是没有影响原有测试用例在基类这个验证平台的操作。

7.2.1 类的继承与重写

当一个类从另外一个类继承其属性，原有的类就称为基类或者父类，而扩展引申出来的类称为派生类或者子类。

在如图 7-1 所示的继承层次中，多边形是一个基类，从它扩展出来的四边形、五边形、十六边形和十边形都是派生类，每个子类都继承了其父类的方法和数据成员，并且重写了面积和周长的计算方式，而且所有的子类还添加了一个可以绘制其形状的方法。四边形是一个父类，其派生扩展了正方形和长方形两个子类。

在继承的过程中，存在如下的一些基本规则。

- 1) 子类继承父类的所有数据成员和方法。
- 2) 子类可以添加新的数据成员或者方法。
- 3) 子类可以重写基类中的数据成员和方法，也就是重写。
- 4) 如果一个方法被子类重写，其必须保持和基类的原定义有一致的参数。
- 5) 子类可以通过 super 操作符引用父类中的方法和成员。
- 6) 被声明为 local 的数据成员或方法只能对自身可见，而对于外部和子类不可见；

被声明为 protected 的数据成员或方法，对外部不可见，对于自身和子类可见。

SystemVerilog 通过 extends 关键字来扩展类，语法如下：

```
class class_name extends base_class_name
...
endclass
```

下面通过源代码 7-1 来说明上述规则。

源代码 7-1 类的继承实例

```
//Chapter7 class_extend_example.sv
class Packet ;
  //属性
  integer status;

  // 方法
  task rst();
    status = 0;
  endtask
endclass

class DerivedPacket extends Packet ;
  //属性
  integer a,b,c;

  // 方法
  task showstatus();
    $display(status);
  endtask
endclass
```

基类为 Packet，派生类为 DerivedPacket；DerivedPacket 继承了 Packet 中的所有数据成员和方法，另外它还添加了 a、b、c 三个数据成员和一个 showstatus 方法。

SystemVerilog 所提供的机制被称为单一继承，也就是说，每一个子类都由唯一的父类继承而来。

父类的方法也可以重写，这样就可以改变原有定义的行为。源代码 7-2 是另外一个例子。

源代码 7-2 类的重写实例 1

```
//Chapter7 class_override_example.sv
class Packet ;
  //属性
  int status = 4;
  function int chkstat(int s);
    return (status == s);
  endfunction
endclass

class DerivedPacket extends Packet ;
  //属性
  int status = 15;
  function void chkstat(int s);
    $display(status);
  endfunction
endclass
```

在扩展的过程中，子类 DerivedPacket 重写了父类中原有的成员和方法。其中，将 status 的初值设置为 15，而 chkstat 改为打印信息而不是返回数值。

在实际应用过程中，我们有诸如此类的需要，从而要采用扩展的方式去定义派生类，例如在通信芯片的设计和应用中，可能存在下列需求。

- 1) 扩展一个基类 Packet 可以生成不同类型的 Packet (Ethernet/ATM…)。
- 2) 扩展一个基类 Monitor (监控器) 实现更多的功能。
- 3) 扩展一个基类 Transactor (事务处理器) 添加新的功能。
- 4) 扩展一个基类 Generator (激励产生器) 生成不同类型的激励数据。
- 5) 验证工程师通过建立共用的类库，通过扩展添加新的功能，实现在团队中共享代码，实现重用。

在第 5 章中，我们知道一个类的对象被定义的时候是一个空的句柄，当其构造函数被调用的时候才分配空间，其句柄也指向该片空间的入口地址。同样，当一个派生类的对象被定义的时候，它也是一个空句柄，当其构造函数被调用的时候，分配的空间不仅有来自父类的继承而来的数据和方法，还有自己添加的部分；其中新添加的数据成员和方法存储在独立的空间中，保持与继承成员和方法的独立性。上面 example_1 和 example_2 的对象可以通过图 7-2 来表示其空间分配过程中的关系，可以看出在派生类中，重写和新添加的数据成员和方法都保持独立的空间。

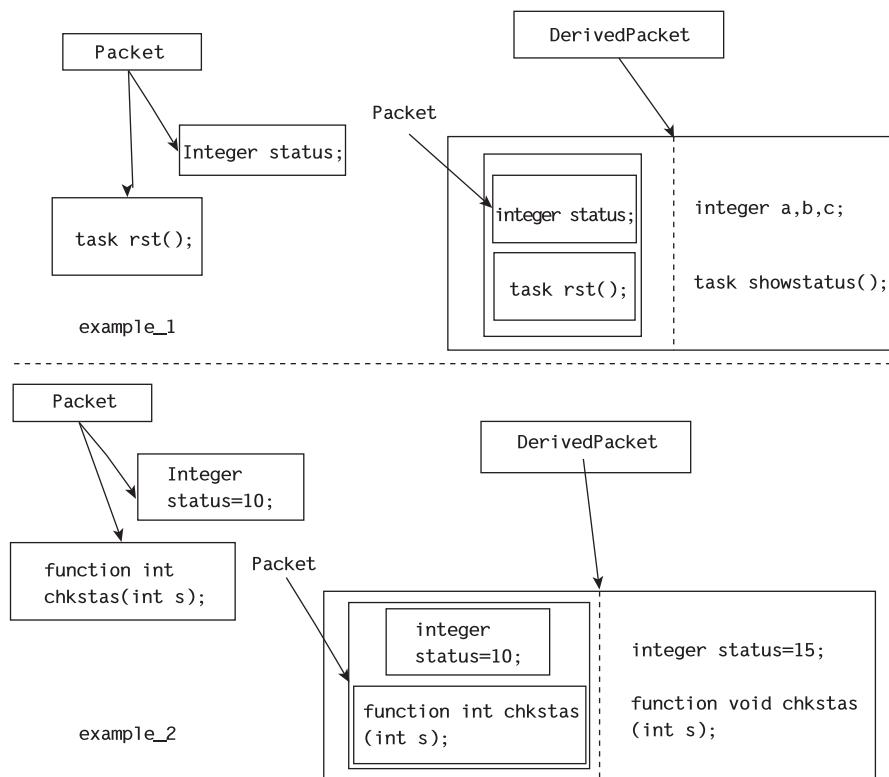


图 7-2 基类与派生类对象空间分配

7.2.2 子类对象与父类对象的赋值

子类对象也是它们的父类对象的有效表示。例如，每一个 DerivedPacket 对象都是一个完全合法的 Packet 对象。一个 DerivedPacket 对象可以赋值给一个 Packet 的对象变量：

```
DerivedPacket lp = new;
Packet p = lp;
```

为此，子类对象可以赋值给父类对象。在这个过程中有两个需要注意的问题。

- 1) 如何处理一个子类对父类的赋值过程？
- 2) 如何访问和引用重写和新增的数据成员和方法？

其中，赋值过程中遵循以下规则。

- 子类对象是父类对象的有效表示，可以赋值给父类对象。
- 父类对象可以通过 \$cast 的方式尝试给子类对象赋值，并判定是否赋值合法。

在访问对象的过程中遵循一下规则：通过父类的对象去引用在子类中重写的属性或方法，结果只会调用父类的属性或方法；通过子类对象可以直接访问重写的属性或方法；在子类扩展过程中新增的属性和方法对于父类对象不可见；子类可以通过 super 操作符访问父类中的属性和方法，以区分于本身重写的属性和方法。

我们通过源代码 7-3 所示的具体实例来解释上面的概念和规则。

源代码 7-3 基类与派生类实例

```
//Chapter7 based_derived_example.sv
module derived2base;
class Packet;
    integer i = 1;
    function integer get();
        get = i;
    endfunction
endclass

class DerivedPacket extends Packet;
    integer i = 2;
    function integer get();
        get = -i;
    endfunction
endclass

initial begin
    DerivedPacket ld,lp = new ;
    Packet p = lp;
    j = p.i; // j=1,而不是2
    j = p.get(); // j=1,不是-1或-2
    $cast(ld,p); // 将 p 赋值给 ld,做合法性检查。
end
endmodule
```

从 p 的角度看，DerivedPacket 中的 new 和所有重写的成员都是不可见的。为了能够通过父类对象（在本例中为 p）调用被重写的方法，这个方法需要使用 virtual 声明，也就是后面我们会介绍的虚方法。

前面我们已经介绍了，一个子类的对象赋值给一个父类的对象是合法的；而若一个父类对象直接赋值给子类对象就是不合法的，但是假如一个父类对象指向的是一个子类对象，那么把该父类对象赋值给另外一个子类对象，则是合法的。为了检查一个父类对象对一个子类对象赋值是否合法，我们可以使用动态的类型转换 \$cast。

```
task $cast( singular dest_handle, singular source_handle );
or
function int $cast( singular dest_handle, singular source_handle );
```

当和对象的句柄一起使用的时候，\$cast 会检查 source_handle 和 dest_handle 类型是否一致，如果是，而且合法，就赋值；否则就会发出类型不匹配的告警。

上面我们介绍了父类对象对子类对象的引用，另外在扩展的过程中由于重写，子类对象如何引用父类的属性和方法，而避免歧义呢？super 关键字可以在子类的内部使用，可以用来引用其父类的成员，特别是当父类成员被子类重写的时候。如源代码 7-4 所示。

源代码 7-4 super 操作实例

```
//Chapter7 super_example.sv
class Packet;                                // 父类
    integer value;
    function integer delay();
        delay = value * value;
    endfunction
endclass

class DerivedPacket extends Packet;           // 子类
    integer value;

    function integer delay();
        delay = super.delay() + value * super.value;
    endfunction
endclass
```

通过 super 访问的成员可以是在上一级声明的成员，或者是上一级类其自身继承得到的成员；但不能访问更高层次的成员（例如，super.super.count 是不允许使用的）。

7.2.3 构造函数调用

一个子类在实例化的时候会调用其构造函数 new() 为其分配空间。在该函数中定义的任何代码执行之前，new() 执行的第一个默认动作是调用其父类的构造函数 new()，并且会沿着继承关系按这种方式一直向上回溯调用。因此，所有的构造函数会按正确的顺序调用执行，它们都是起始于根基类并结束于当前的类。如图 7-3 所示。

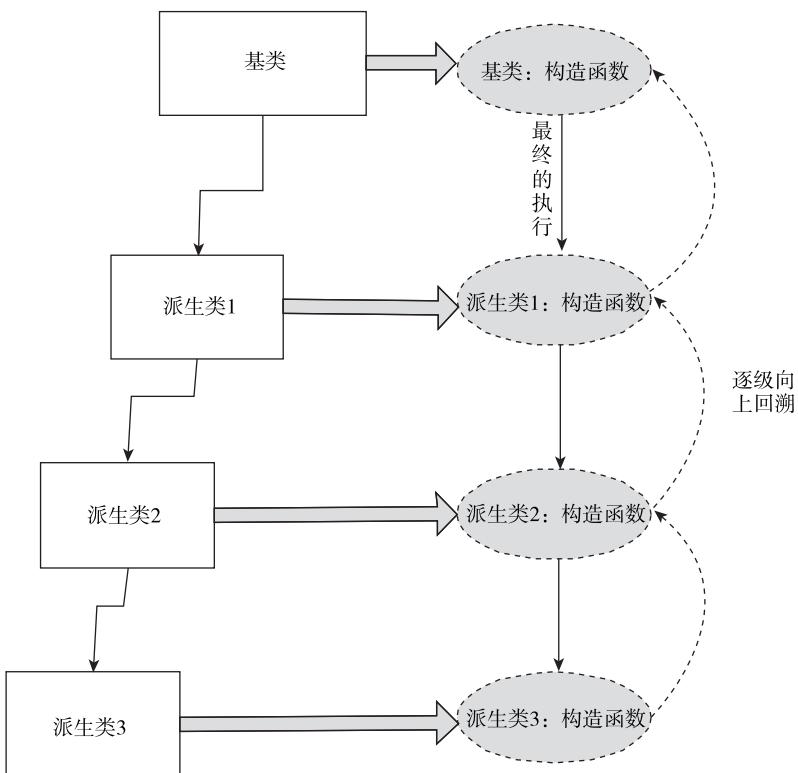


图 7-3 构造函数的调用和执行

如果父类的构造函数需要参数，那么可以有两种方法做参数传递：一种是在扩展的时候提供初始化参数，另外一种是在派生类中直接使用 `super` 关键字，调用父类的构造函数。

如果参数总是相同的，那么它们可以在被扩展的时候指定：

```
class EtherPacket extends Packet(5);
```

这个例子将 5 传递到与 `Packet` 中的 `new` 方法中。

一个更加通用的方法是直接使用 `super` 关键字来调用父类的构造函数：

```
function new();
    super.new(5);
endfunction
```

应该注意的是，`super.new(5)` 必须是 `new` 函数中执行的第一条语句。

我们通过源代码 7-5 来说明整个默认调用构造函数的过程。

源代码 7-5 构造函数链实例

```
//Chapter7 new_chain_example.sv
class base
    function new()
        $display("calling base constructor");
    endfunction
```

```

endclass
class derived1 extends base
    function new ()
        $display("calling derived1 constructor");
    endfunction
endclass
class derived2 extends derived1
    function new ()
        $display("calling derived2 constructor");
    endfunction
endclass
class derived3 extends derived1
    function new ()
        $display("calling derived3 constructor");
    endfunction
endclass
module constructor_chain;
    derived3 myderived3;
    initial begin
        myderived3. new();
    end
endmodule

```

在仿真器中的打印输出会是：

```

calling base constructor
calling derived1 constructor
calling derived2 constructor
calling derived3 constructor

```

7.3 虚方法与多态

在讨论多态之前，我们先来看源代码 7-6 所示的例子。

源代码 7-6 类的重写实例 2

```

//Chapter7 base_override_example.sv
class Packet ;
    task build_payload();
        $display("Packet payload");
    endtask

    task build_packet();
        ...
        build_payload();
        ...
    endtask
endclass
class DerivedPacket extends Packet;

```

```

task build_payload(); // 重写方法
    $display("DerivedPacket payload");
endtask
endclass

module poly1;
    DerivedPacket der = new();
initial
    der.build_packet();
endmodule

```

在派生类对象 der 调用其 build_packet 过程中，会调用到 build_payload 这个任务；而 build_payload 这个任务在扩展的过程中被重写过，那么哪一个会被调用呢？

答案是，父类中的方法会被调用，而不是重写的那个。

仿真器输出打印会是：

```
Packet payload
```

这说明，默认情况下子类中重写的方法对于父类是不可见的。如图 7-4 所示。

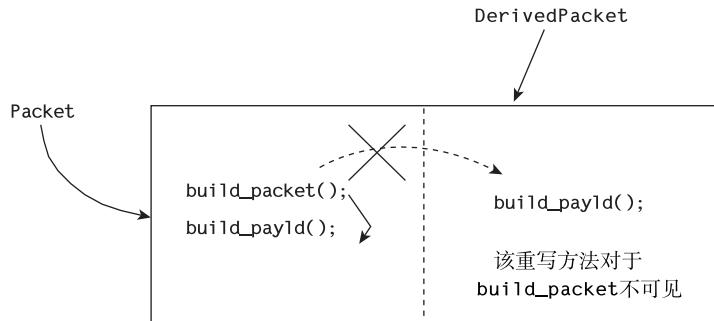


图 7-4 重写方法的可见度

与上面继承一节中讨论的空间分配类似，派生类对象的空间分配分为两个部分，一部分是来自父类的继承部分，一部分是重写或者新增的部分。默认情况下，父类的方法是无法访问派生类的重写和新增部分的。也就是说，重写后的 build_payload 对于继承的 build_packet 不可见，build_packet 只能调用父类中原有声明的 build_payload。

那么，假如我们希望重写的方法能够被父类看得到，该如何处理呢？也就是说，假如我们希望 build_packet 能够调用子类中重写的 build_payload，有实现的办法吗？

答案是，有的。也就是下面我们要讨论的虚方法和多态。

7.3.1 虚方法

类中的方法可以在定义的时候通过添加 virtual 关键字来声明一个虚方法，虚方法是一个基本的多态性结构。虚方法为具体的实现提供了一个原型，也就是说派生类中，重写该方法的时候必须采用一致的参数和返回值。

虚方法可以重写其所有基类中的方法，然而普通的方法被重写后只能在本身及其派生

类中有效。从另外一个角度解释，每个类的继承关系只有一个虚方法的实现，而且是在最后一个派生类中。

我们通过源代码 7-7 所示的例子来说明虚方法及其特性。

源代码 7-7 虚方法与多态实例

```
//Chapter7 virtual_poly_example.sv
class BasePacket;
    int A = 1;
    int B = 2;
    function void printA;
        $display("BasePacket::A is %d", A);
    endfunction : printA
    virtual function void printB;
        $display("BasePacket::B is %d", B);
    endfunction : printB
endclass : BasePacket
class My_Packet extends BasePacket;
    int A = 3;
    int B = 4;
    function void printA;
        $display("My_Packet::A is %d", A);
    endfunction : printA
    virtual function void printB;
        $display("My_Packet::B is %d", B);
    endfunction : printB
endclass : My_Packet
...
BasePacket P1 = new ;
My_Packet P2 = new ;
initial begin
    P1.printA;           // 打印'BasePacket::A is 1'
    P1.printB;           // 打印'BasePacket::B is 2'
    P1 = P2;             // P1 指向 My_Packet 对象
    P1.printA;           // 打印'BasePacket::A is 1'
    P1.printB;           // 打印'My_Packet::B is 4'
    P2.printA;           // 打印'My_Packet::A is 3'
    P2.printB;           // 打印'My_Packet::B is 4'
end
```

从上面我们可以看到，基类 BasePacket 定义了两个方法 printA 和 printB，其中 printB 为虚方法；在其派生类 My_Packet 中重写了这两个方法，而且还重写了 A 和 B 两个成员的初值。从内存分配的角度来看这两个类的对象 P1 和 P2，如图 7-5 所示。

在 P1 指向 P2 后，当 P1.printA 被调用的时候，程序会调用访问内存中类对象 P2 中 P1 的部分，而此时发现 printA 是一个普通方法，为此直接调用并结束访问；当 P1.printB 被调用的时候，程序同样找到类对象 P2 中 P1 的部分，此时发现 printB 是个虚方法，这时其会咨询系统，查看整个 P2 在定义的时候是否重写了该方法，系统发现在 P2 中（除了 P1 外）

确实重写了该方法，为此，程序会直接调用 P2 重写的实现；为此输出会是 My_Packet::B is 4。

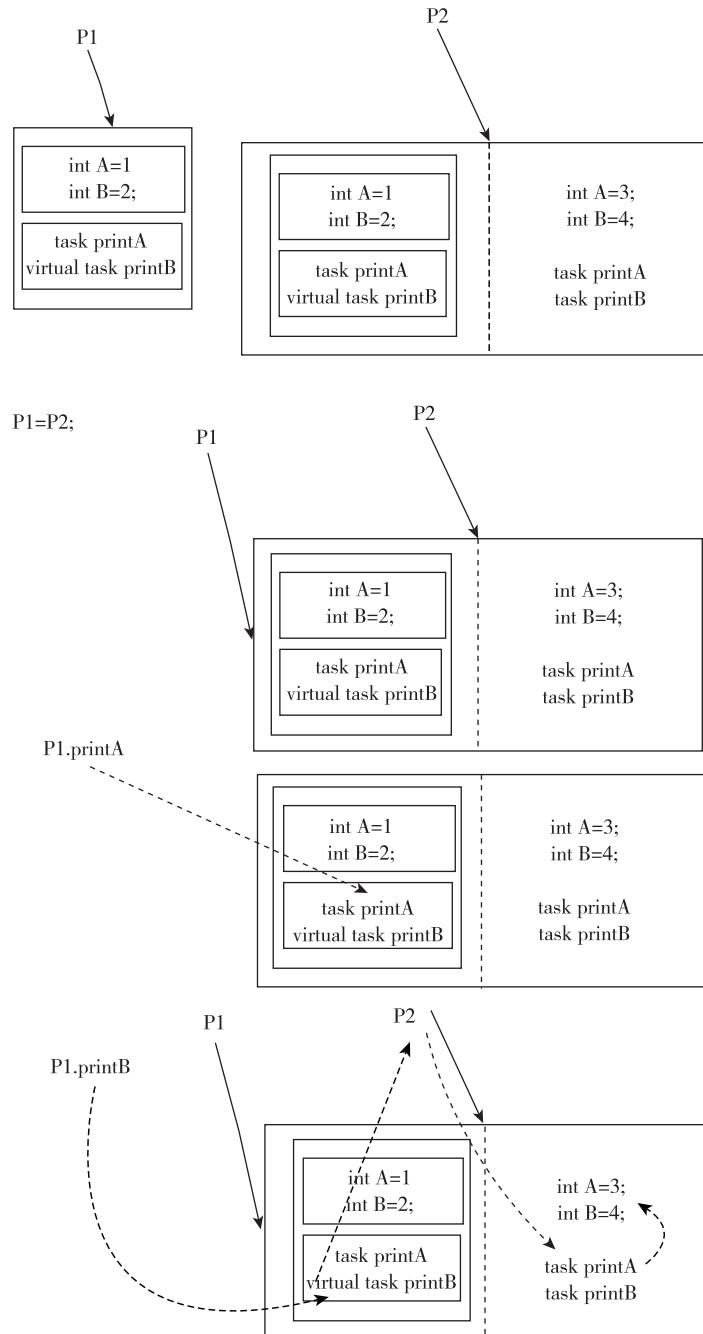


图 7-5 虚方法的具体实现

一个方法在被重写的时候可以声明为虚方法；一旦方法被声明为虚方法，它在后续的继承过程中就永远是一个虚方法，不管重写的时候是否使用 `virtual`。也就是说，虚方法在其子类扩展、重写的时候 `virtual` 是可选的而不是强制的。

7.3.2 多态

封装可以隐藏实现细节，使得代码模块化，继承可以扩展已存在的代码模块，它们的目的都是为了代码重用，而多态则是为了实现另外一个目的：接口重用。

虚方法是类中声明的非静态方法，在类中该方法声明带有 `virtual`，我们称带有虚方法的类为多态类。

多态类提供了相同的操作接口，但能适应于不同的应用要求。当基于父类所设计的程序在应用的时候需要适应子类操作变化时，就需要使用虚方法通知编译器这种可能的变化，使编译器为虚方法调用生成特别代码，以便运行时对虚方法调用采用动态绑定。

为此，我们说虚方法与重写的实现就是多态。

从前面 7.3.1 的例子（源代码 7-7 和图 7-5）中我们知道，实现了多态需要以下三个步骤：

- 1) 在父类中定义虚方法。如 `BasePacket` 类中的虚方法 `printB`。
- 2) 在子类中重写父类中的虚方法。如 `My_Packet` 中对 `printB` 的重写。
- 3) 申明父类对象的变量（如 `P1`），该变量既可以指向父类对象（如 `P1` 自身），也可以指向子类对象（如 `P2`）。当变量指向父类对象时，调用的是父类方法；当变量指向子类对象时，调用的是子类同名方法。为此，当父类的对象指向不同的子类对象的时候，虚方法（如 `printB`）可以表现出不同的实现方法，而它们都可以通过统一的父类对象实现访问。

7.4 虚类和参数化类

为了实现代码重用，SystemVerilog 还引入了虚类（抽象类）和参数化类两个特殊的类。虚类主要作为基类模板，不可以实例化；参数化类就如 C++ 中的类模板（template），可以根据需要修改其中参数的类型。

7.4.1 虚类

我们可以创建一组类，它们可以是从一个共有基类引申而来，如图 7-1 所示，多边形就是共有的基类。在网络通信的验证平台中，我们以 `BasePacket` 作为共有基类定义了一个包的初始结构，但这个结构并不完整，因此它永远也不会被实例化。然而从这个基类开始，我们可以派生许多子类，例如以太网包、令牌环包、GPSS 包等。每一个包可能看上去很相似，它们都有相同的方法，但其内部的实现细节上则可能存在明显的差别。

一个基类定义了其子类的原型，共有基类可采用 `virtual` 将其变得抽象化。为此，虚类也被称为抽象类。

虚类的对象是不可以被直接构造使用的，也就是虚类一般不用来定义对象。虚类的构造函数只能在其派生类的构造函数的调用过程中使用。虚类内也可以定义虚方法，虚方法是一个基本的多态性结构。虚类中的虚方法可以只提供原型而无须具体实现，也就是称为纯虚方法，只要 `virtual` 前面添加一个 `pure` 的关键字就可以，其扩展过程中可以被重写为纯虚方法或者普通虚方法，并提供一个具体的实现。

虚类也可以被进一步扩展为另外一个虚类，但是若要扩展为非虚类（普遍类）所有的纯虚方法必须被重写，并且提供一个具体的实现，以便在后续过程中使用。在提供了方法的实现后，普遍派生类就完整而且可以被构造使用了。任何类都可以被扩展为一个虚类，添加或者重写纯虚方法。

```
virtual class BasePacket;
    pure virtual function integer send(bit [31:0] data); // 没有实现
endclass
class EtherPacket extends BasePacket;
    virtual function integer send(bit [31:0] data);
        // 函数具体实现
    ...
endfunction
endclass
```

EtherPacket 是一个可以被实例化的普遍类。通常，如果一个虚类具有纯虚方法，为了使子类能够被实例化，所有这些纯虚方法都必须被重写（并且拥有一个方法的具体实现）。如果任何纯虚方法没有具体的实现，那么子类必须是抽象类。

需要注意的是，一个没有实现体的方法也是合法的，并且可以被调用。例如上面 BasePacket 中的纯虚方法 send 也可以如下声明：

```
virtual function integer send(bit [31:0] data); // Will return 'x
endfunction
```

简而言之，虚类是一个定义抽象概念的类模板，不可以实例化。只有虚类才可以定义纯虚方法（没有具体实现），在扩展过程中，纯虚方法必须要有具体实现的重写，才可以在普通派生类中使用。

7.4.2 参数化类

通用的类是很有用的，特别是在例化的时候可以修改数组的大小和数据类型。为了避免为每个特定的类编写类似的代码，SystemVerilog 也支持参数化类，就如 C++ 中的模板。

SystemVerilog 参数化机制可以用来参数化一个类，如源代码 7-8 所示。

源代码 7-8 参数化类实例

```
//Chapter7 parameterized_class.sv
class vector #(int size=1);
    bit [size-1:0] a;
endclass
```

接下来，这个类的对象就可以像模块或接口一样例化：

```
vector #(10) vten;           // size 为 10 的 vector 对象
vector #(.size(2)) vtwo;    // size 为 2 的 vector 对象
typedef vector #(4) Vfour;   // size 为 4 的类
```

当将类型作为参数的时候，这个功能特别有用：

```

class stack #(type T = int );
    local T items[];
    task push(T a);
    ...
    endtask
    task pop(ref T a);
    ...
    endtask
endclass

```

上面定义了一个通用的 stack 类，它可以使用任意类型进行实例化：

```

stack is;           // 缺省:int 类型的 stack
stack#(bit [1:10]) bs; // 10 位向量的 stack
stack#(real ) rs;   // 实数的 stack

```

任何类型都可以用作参数，包括用户定义的类型，例如类或结构体。

通用类和真实参数值的一个组合被称作一个特例。每个特例又具有一组单独的静态成员变量（这与 C++ 模板类一致）。为了在多个类特例间共享静态成员变量，这些变量必须放置在一个非参数化的基类中。

```

class vector #(int size=1);
    bit [size-1:0] a;
    static int count=0;
    function void disp_count();
        $display("count: %d of size %d", count, size);
    endfunction
endclass

```

上面例子中的变量 count 只能通过对应的 disp_count 方法访问。对于 vector 类的每一个特例，count 都具有它自己的唯一的复制。

为了避免必须在声明中或产生该类的参数中出现重复特例，我们可以使用 typedef：

```

typedef vector#(4) Vfour;
typedef stack#(Vfour) Stack4;
Stack4 s1, s2;           // 声明 Stack4 类型的对象

```

一个参数化的类可以扩展另外一个参数化的类，例如：

```

class C #(type T = bit);... endclass          // 基类
class D1 #(type P = real) extends C;           // T 的类型为 bit(缺省)
class D2 #(type P = real) extends C #(integer); // T 的类型为 integer
class D3 #(type P = real) extends C #(P);       // T 的类型为 P

```

D1 类使用基类的缺省类型（bit）参数扩展了基类 C。D2 类使用一个 integer 参数扩展了基类 C。D3 类使用参数化类型（P，它使得扩展类被参数化）扩展了基类 C。

7.5 约束重写

类在继承的过程中除了可以重写数据成员和方法外，其约束也可以被重写。这对于激励的随机生成和代码重用有重要的意义。

约束与类数据成员和方法遵从相同的规则：如果一个继承类中的约束与它父类中的约束具有相同的名字，那么它会重写基类约束。如源代码 7-9 所示。

源代码 7-9 约束块重写实例

```
//Chapter7 constraint_override_example.sv
class A;
    rand integer x;
    constraint c {x < 0;};
endclass
class B extends A;
    constraint c {x > 0;};
endclass
```

类 A 中包含了 x 小于 0 的约束，而类 B 中包含了 x 大于 0 的约束。派生类 B 重写了约束 c 的定义。约束与虚方法类似，所以若将 B 的对象强制赋值给到 A 的句柄不会改变约束设置。例如：

```
initial begin
    A a1;
    B b1;
    b1.randomize() //x 小于 0
    a1 = b1;
    a1.randomize() //x 大于 0
end
```

在约束的定义过程中，我们也可以把约束分为两个类型：硬约束和软约束。所谓硬约束也就是上面的例子，限制在一个确定的范围之内，例如，以太包的包长大于 64 字节小于 1522 字节；软约束就是通过变量来限制范围，这样我们通过变量来动态地改变需要的约束，例如 $x < y$ ，而 y 是一个变量，所以我们可以控制 y 进而动态地对 x 进行控制。

7.6 数据的隐藏与封装

到目前为止，所有的类属性和方法都毫无限制地在类外可见。然而，我们有时希望通过隐藏类属性和类方法的名字来限制在类的外部访问类属性和类方法。这就使得其他程序员能够不依赖于一个特定的实现，而且它还防止仅对类内部有效的类属性被偶然地修改。当所有的数据都变成隐藏的（仅能被公共方法访问），代码的维护和测试都变得更加容易。

在 SystemVerilog 中，未被限定的类属性和方法默认是公共的（public），它们对访问对象的任何人都可见的。

被标识成 local 的成员仅对类内的方法可见，而且这些本地成员在派生类内是不可见的，即是无法访问的。当然，访问本地类的属性或方法的非本地方法可以被继承，并且作为子类的方法它可以正常工作。

除了可以被继承以及对子类可见外，被标识成 protected 的类属性或方法具有本地成员的所有特性。

需要注意的是，在类的内部，本地方法或类属性可以被引用，即使它属于一个不同的实例对象。例如：

```
class Packet;
    local integer i;

    function integer compare(Packet other);
        compare = (this.i == other.i);
    endfunction
endclass
```

关于封装的一个严格的解释可能是：other.i 在当前的 Packet 中不应该是可见的，因为它在实例之外引用类的本地属性。然而，在同一个类内这样的引用是被允许的。在这个例子中，this.i 可以与 other.i 比较，并返回逻辑比较的结果。

类成员可以被标识成 local 或 protected；类属性可以进一步定义成 const，方法可以进一步定义成 virtual。在指定这些修饰符的时候没有预定义的顺序；然而，对于每一个成员它们只能出现一次，也就是不可以将成员同时定义为 local 和 protected。

继承和多态在面向对象编程语言 SystemVerilog 中是很容易实现的，但是硬件验证和传统编程有很大区别。对继承和多态总结如下。

- 1) 通过扩展来为原有的类添加新的功能或者实现动态的绑定方法。

如果一个类已经被多个部分共享和扩展使用，直接修改该类的功能可能会影响其他部分的功能实现，通过简单的扩展而且使用其派生类是最好的选择。若这个类没有被其他部分使用，如何修改取决于需要添加多少功能。

假如修改只是一两行代码，你可以不需要为此而扩展一个派生类。假如这个改动比较大，那么就应该考虑扩展一个派生类了。需要注意的是，通过扩展来实现重用，隐藏的功能缺陷可能会在共享代码扩展的过程中传递到整个验证环境中。

- 2) 不要跨多个层次引用方法，一般情况下只引用上一层。

- 3) 利用构造函数链来实现对基类数据成员的初始化。

这样可以保证在上一层的类发生改动的时候，对整体的初始化没有影响。

- 4) 假如类中的方法需要多个层次嵌套 if…else 或者 case 语句，可以考虑定义一个抽象类，再从该抽象类（虚类）中根据需求和条件扩展不通用的派生类。

- 5) 假如使用虚类，确保在每个层次都使用虚方法，这样可以留出足够的空间供后续继承和重写。

- 6) 尽量把派生类的层次控制在 6 层以内，以保证结构清晰和方便调试。

- 7) 为抽象类、基类和派生类使用统一的命名方案。

例如，基类可以使用大写：BASE_PACKET，抽象类可以使用 v_作为前缀：V_PACKET，派生类可以使用_ext 后缀：ether_packet_ext。

多态和继承可以在不影响现有验证环境的情况下，扩展不同的功能。多态可以支持不同的实现功能，继承可以扩展功能、共享类库和创建新的测试用例。

验证平台可以通过抽象类来支持多态，其中抽象类可以用来定义激励生成和事务处理器。虽然多态和继承可以提供特定的边界条件测试用例，通过控制生成大量的激励，但是验证平台也应该考虑到可以支持用户输入，例如通过数据文本或者调用参数来输入。

功能覆盖率

随着设计的日益复杂，验证面临一系列挑战：上市时间、调试周期、功能完整性等。最有效而且全面的验证方法就是使用约束随机激励测试，这种方法相对于针对单个功能点的直接测试，大大提升了验证效率。但是，随机激励在测试整个设计的各种状态的时候，我们怎么才能知道哪个状态或者功能点已经被验证过了呢？进一步，我们可以问哪些功能点在随机激励的情况下，没有被覆盖到呢？这就需要有一个衡量的标准——功能覆盖率就是很好的标准。

基于覆盖率驱动的验证方法学中包含了代码覆盖率和功能覆盖率两个概念。这两个覆盖率都是供验证工程师分析并回答一个终极问题：验证是否收敛？

8.1 覆盖率

一般来说，覆盖率就是指设计中的哪些部分被测试过了。其中，有验证计划目标覆盖率、代码覆盖率和功能覆盖率三类覆盖率。下面我们将对各个概念一一做详细讨论。

8.1.1 目标覆盖率

目标覆盖率是指在验证计划中规定的需要验证点的目标值。在验证计划中，当验证点实际覆盖率没有达到 100% 的时候，说明验证工作还未完成目标方案。没有达到 100% 的项目需要通过添加测试用例或者修改约束等来对其进行充分的验证。

验证计划中列出的项目都要一一被测试，当然这需要一个比较全面和完整的验证计划。为此，在验证环境搭建的前期，制定验证计划，明确验证点并确定目标覆盖率是一项艰巨而且细致的工作。

制定验证计划中的功能点的时候，需要考虑如下三个问题。

- 1) 哪些功能点需要检查？
- 2) 这个功能点的哪些数据需要检查？
- 3) 如何对这些数据进行采样？

哪些功能点需要检查呢？这要根据设计的具体情况而定，一般情况下，以下几类是参考的对象：功能要求、接口要求、系统规范、协议规范等。具体验证计划中可能表现为：FIFO 是否溢出和空读、外部接口是否遵从以太网物理层的传输协议、是否满足系统规范要求的支持发送超长包、内部的 AMBA 总线是否符合协议要求等。

在把这些功能点具体化到设计代码中的时候，我们可能会做一些折中的考虑，如何划

分数据类别，例如，对于有一个以太包的包长的功能点，我们当然希望能够把所有的包长都测试一遍，假如时间有限，我们也可以把包长划分为最短包长、最长包长、合法包长（在最长和最短之内）、非法包长（在最长和最短以外），这就具体化了，可以容易得知哪些数据需要检查。因为验证空间是无穷大的，抽象并且量化验证功能点是必须引起我们关注的问题。

在解决如何分类和量化功能点之后，就是什么时候对该功能点的数据进行采样。这要根据具体情况做详细分析。例如，在 mailbox 接送和发送包的过程中我们可以统计包的相关情况，我们也可以在每个时钟边沿去检查计数器的值，激发条件要根据功能点的性质来确定。

8.1.2 代码覆盖率

代码覆盖率是从软件编程的角度来分析设计代码是否被充分的验证。代码覆盖率可以通过仿真器在运行的过程中自动统计数据并生成报告。代码覆盖率只是一个统计报告，并不能指出设计行为是否正确，它只是表明设计代码中的哪些部分已经测试或者没有被测试过，验证工程师或者设计工程师通过分析报告，了解是否存在冗余的代码，是否需要增加测试激励以保证代码能够被充分的测试。

代码覆盖率可以通过如下方式来统计，这些方法也适用于一般的 C 或者 C++ 等编程语言的设计。

- 行覆盖率：检查某行代码是否被执行过。
- 分支覆盖率：检查条件分支是否都被执行过。
- 条件覆盖率，表达式覆盖率：通过真值表分析表达式各种逻辑组合。
- 有限状态机覆盖率：检查每个状态是否被覆盖，状态之间的跳转是否被执行。

具体的使用方法可以参考仿真器的使用文档。代码覆盖率无法从硬件实现的功能点这个角度来提供更多的信息；代码覆盖率一般是在测试回归的时候，在所有的测试用例仿真结束后做一个总结报告，然后进行分析。

8.1.3 功能覆盖率

就如前面提到的，功能覆盖率在随机激励测试中尤其重要。一般情况下，不论直接测试还是随机测试，验证计划中都会规定哪些功能点是需要测试的。在直接测试中，可能一个测试用例就对应测试某一个功能点，所以很多情况下，回归测试所有用例全部通过就可以表示达到了验证计划的要求；而在随机激励测试中，一个测试用例可能会测试到不同的功能点，或者是这些功能点的局部，为此需要有一个具体的衡量标准。

在测试用例运行过程中如何做功能覆盖率统计呢？这需要根据验证计划并借助 SystemVerilog 的专门语法在验证环境中具体实现。

SystemVerilog 中提供了如下两种类型的功能覆盖率表达形式。

1) 面向控制的功能覆盖率。通过 cover 对断言中的 sequence 或者 property 做统计（在第 9 章我们再详细讨论）。

2) 面向数据的功能覆盖率。在特定的时间点对某些数据值使用 covergroup 做采样统计分析。

这一章，我们重点讨论如何使用 covergroup 做功能点覆盖率的统计和分析，最后我们把该功能应用到剪刀、石头、布例子中，并搭建一个覆盖率驱动的验证平台。

8.2 SystemVerilog 的功能覆盖率

SystemVerilog 的功能覆盖率通过提供简洁的语法定义功能覆盖率模型，从而解决了 Verilog 中原有语法上的不足之处。功能覆盖率模型在执行的过程中，通过覆盖率统计分析工具的反馈来加速开发更多高效的测试用例，进而可以测试到更多的边界情况和特定情形。

SystemVerilog 功能覆盖率结构可以实现如下功能。

- 定义和统计变量和表达式的覆盖率及其交叉覆盖率。
- 支持自动化创建覆盖点计数器和用户自定义覆盖点计数器。
- 覆盖点计数器可以同时包括多种类型，例如数值集合、跳转、交叉项等。
- 可以在不同层次设置过滤条件。
- 可以通过 events 和 sequences 来触发覆盖点采样。
- 过程化覆盖点采样激活和覆盖率实时查询。
- 提供属性参数来管理覆盖率统计和计算。

下面我们重点讨论 covergroup（覆盖组）/coverpoint（覆盖点）/cross（交叉覆盖点）三个语法结构及其应用，对于其他的配置参数和内置方法，请参阅附录 A。

8.2.1 覆盖组 (covergroup)

SystemVerilog 中提供了 covergroup 这个关键字来定义一个覆盖组。每个功能覆盖组内可以定义以下信息。

- 该覆盖组采样的时钟事件。
- 一系列的覆盖点。
- 覆盖点的交叉覆盖。
- 可指定形式参数。
- 覆盖组的属性参数。

covergroup 是一个用户自定义的类型，其类似一个 class，需要通过 new 来构造而且可以多次例化；可以在 module/program/interface/class/package 等结构体内定义覆盖组，其语法如下：

```
covergroup name [( <list_of_args> )] [<clocking_event>] ;
  <cover_option> ;
  <cover_type_option> ;
  <cover_point> ;
  <cover_cross> ;
endgroup [ : identifier ]
```

覆盖组可以定义参数，在其例化时需传递实参，实参在 new 的过程中采样；时钟事件定义了功能覆盖组的采样条件，若不定义时钟采样条件，则需要通过调用内置的采样方法 sample ()。其中，最重要的是通过 coverpoint 和 cross 的定义覆盖点和交叉覆盖点，其能够指定采样变量的对象；源代码 8-1 是一个覆盖组的应用例子。

源代码 8-1 功能覆盖组实例

```
//Chapter8 covergroup_example.sv
module cover_group;
bit clk;
enum {sm1_pkt, med_pkt, lrg_pkt} ether_pkts;
bit [1:0] mp3_data, noise, inter_fr_gap;

covergroup net_mp3() @ (posedge clk);
    Mp3: coverpoint mp3_data;
    Junk:coverpoint noise;
    epkt: coverpoint ether_pkts;
    Traffic:cross epkt, Mp3; // 两个覆盖点
endgroup

net_mp3 mp3_1 = new();
net_mp3 mp3_2 = new();
...
endmodule
```

从上述例子我们可以看出，覆盖组 net_mp3 的采样事件是时钟 clk 的上升沿，其中定义了三个覆盖点 Mp3、Junk、epkt 和一个交叉覆盖点 Traffic；这三个覆盖点分布对应着变量 mp3_data、noise、either_pkts；交叉覆盖点 Traffic 是 epkt 和 Mp3 两项的所有组合。mp3_1 和 mp3_2 是覆盖组的两个例化，通过 new 构造分配空间。

covergroup 也可以在 class 内定义使用，可以对该 class 的属性定义对应的功能覆盖率模型，无论该属性是否是 protected 或者是 local；在类中可以不使用例化，但仍需在类的构造函数中调用 new 对覆盖组进行分配初始化。如源代码 8-2 所示。

源代码 8-2 功能覆盖点实例

```
//Chapter8 coverpoint_example.sv
class packet;
// 以太包字段
rand bit [7:0] dest, src;
bit [15:0] len;
bit [47:0] payld [];
bit valid;

covergroup cov1 @ (valid); // 嵌入覆盖组
    cp_dest : coverpoint dest;
    cp_src : coverpoint src;
endgroup

function new (int i);
    payld = new [i]; len = i;
    cov1 = new ();
endfunction :new

endclass : packet
```

覆盖组也可以定义一些属性参数去控制和管理其数据统计和计算的过程，具体的控制参数可参见附录 A。

8.2.2 覆盖点 (coverpoint)

覆盖组内可以通过关键字 coverpoint 定义一个或者多个覆盖点。覆盖点可以针对一个整型的变量或者表达式，每个覆盖点内会有对应的一组分组柜（bin），计数器可以是用户自定义的或者是自动创建的。下面是 coverpoint 的语法：

```
[label :] coverpoint < expr > [ iff( < expr > )]
{
  [< coverage_option > ]
  bins name [ [ ] ]={value_set} [ iff( < expr > )];
  bins name [ [ ] ]=( transitions )[ iff( < expr > )];
  bins name [ [ ] ]=default [ sequence ][ iff( < expr > )];
  ignore_bins name = {value_set};
  ignore_bins name =( transitions );
  illegal_bins name = {value_set};
  illegal_bins name =( transitions );
}
```

label 是可选的覆盖点的名称，用户可指定具体的名字以便通过层次化引用来访问该覆盖点；若不指定，则由仿真器自动分配。该覆盖点在采样事件激发或者内置 sample 被调用时采样统计。iff 结构可以指定采样条件，若指定条件不成立时，则不作覆盖率统计，如下所示：

```
bit [1:0] s0;
covergroup g4;
  cover1:coverpoint s0 iff(! reset);
endgroup
```

仅当 reset 为低电平，也就是！rest 为高电平的时候，表达式为真，覆盖点 cover1 才对变量 s0 做统计。而且，系统自动为覆盖点 s0 分配 4 个分组柜（bins），分别对应其四种可能的取值。每个覆盖点的分组柜可以自动创建（由仿真器自动分配），或者通过各种分组柜语法来显示定义。如果没有显示定义，自动创建分组柜的最大数目将由 auto_bin_max 这个内置参数决定，默认值是 64（其他参数详见附录 A）。如前面例子中的覆盖点 cp_dest 是 bit [7:0] 的数据类型，总共有 256 个数值，若自动生成分组柜，共有 64 个分组柜，每个分组柜分配 4 个数值：<0, 1, 2, 3>, <4, 5, 6, 7>, … <252, 253, 254, 255>。

因为针对一个变量或者表达式，可能我们关心的不是这个变量的所有范围的值，关心的可能只是其取值范围内的某些区域或者跳转点，那么我们可以在覆盖点中显性定义的各种分组柜（bins/ignore_bins/illegal_bins）分别对应一个名字和一个计数器，每个分组柜可以指定一组数值（如 3, 5, 6…）或者数值的跳转序列（如 3 -> 5 -> 6），在仿真过程中，覆盖点指定的情况一旦出现，该计数器加 1。一旦用户显示定义分组柜，系统就不再对其覆盖点对象（变量或者表达式）取值范围内的可能数值自动分配分组柜了，而是只生成用户

定义的分组柜。

不管是显示定义还是自动生成，若分组柜中各种情况的总数除以分组柜的个数，其商就是每个组的平均个数，余数部分会归入最后一个分组柜中，如下例所示：

```
bins fixed [3] = {1:10};
```

其中有 11 个数值，分别分成三个分组柜：<1, 2, 3>，<4, 5, 6>，<7, 8, 9, 10>；其中第三个，也就是最后一个分组柜比其他分组柜多一个数值。如果分组柜的个数超过数值总数，就有可能存在一些空的分组柜。

在显示指定分组柜的时候，可以通过关键字 default 将其他未分配到特定分组柜中的数值进行分配。如源代码 8-3 所示。

源代码 8-3 功能分组柜实例

```
//Chapter8 bin_example.sv
bit [9:0] v_a;
covergroup cg @ (posedge clk);
    coverpoint v_a
    {
        bins a = {[0:63],65};
        bins b[] = {[127:150],[148:191]};// 注意重叠数值
        bins c[] = {200,201,202};
        bins d = {[1000:$]};
        bins others[] = default ;
    }
endgroup
```

这个例子中的第四个分组柜 d 定义了数值 1000 到 1023（\$ 代表 v_a 的最大值）。其他没有分配的会通过关键字 default 归入 others 分组柜（others 数组）中。

除了在分组柜中定义数值，也可以定义数值之间的跳转，如下例所示：

```
bit [2:0] v;
covergroup sg @ (posedge clk);
    coverpoint v
    {
        bins b2 = (3 => 4 => 5); // 3 to 5
        bins b3 = (1,5 => 6,7); // (1 => 6),(1 => 7),(5 => 6),(5 => 7)
        bins b5 = (5 [* 3]); // 3 consecutive 5's
        bins b6 = (3[* 3:5]); // (3 => 3 => 3),(3 => 3 => 3 => 3),(3 => 3 => 3 => 3 => 3 )
        bins b7 = (4[ -> 3] => 5); // ... => 4 ... => 4 ... => 4 => 5
        bins b8 = {2[=3] = > 5}; // ... => 2 ... => 2 ... => 2 ... => 5
        bins anthoers = default_sequence ;
    }
endgroup
```

如上所示，其中分组柜 b2 指定了 $3 \Rightarrow 4 \Rightarrow 5$ 这种数值序列变化，通过“ \Rightarrow ”操作符我们可以指定数值之间的跳转；分组柜 b5 指定了 5 的三次连续重复， $5[* 3]$ 等效于出现了

$5 \Rightarrow 5 \Rightarrow 5$ 这种数值序列；同时我们也可以为连续重复数值序列指定一个窗口，如 b6 所示，其指定了 3 连续重复的次数为 3 次到 5 次 ($[*3:5]$)，每种可能情况都是一个被统计的数字序列；另外，我们也可以定义跟随重复 (goto repetition) 如 b7，在最后一个 4 (第三个 4) 出现后，下一个数值应为 5；另外，非连续重复如 b8 所示，在第三个 2 出现后，只要后续出现 5 就可以（不必像跟随重复那样强制要求）。

`default_sequence` 关键字可以把其他没有归入已定义分组柜中的序列放入 `another`s 这个分组柜中，具体实现算法和仿真器相关，一般情况只记录一次跳变的情况。

除了可以通过 `bins` 定义普通的分组柜外，SystemVerilog 也提供了非法分组柜 (`illegal_bins`) 和可忽略分组柜 (`ignored_bins`)，其中归入非法分组柜的数值或者跳转，若在统计采样的时候出现，仿真器会报告错误，归入可忽略分组柜的数值或者跳转，若在统计采样的时候出现，可以忽略而不做统计，也就是计数器不会递增。

8.2.3 交叉覆盖点 (cross)

覆盖组内可以通过 `cross` 这个关键字对两个或者多个覆盖点指定交叉覆盖点。若 `cross` 指定的交叉覆盖点中的成员是一个变量（未通过 `coverpoint` 指定的覆盖点），则系统会自动为该变量创建对应的默认的覆盖点；但是成员中不能是未通过 `coverpoint` 指定为覆盖点的表达式，这种情况必须事先为其定义覆盖点。

交叉覆盖点的语法结构如下：

```
[label :] cross < coverpoint list > [ iff( < expr > ) ]
{
    bins name = binof(binname) op binof(binname) op ... [ iff(expr) ];
    bins name = binof(binname) intersect { value | [ range] } [ iff(expr) ];
    ignore_bins name = binof(binname) ...;
    illegal_bins name = binof(binname) ...;
}
```

和 `coverpoint` 类似，`label` 可以指定交叉覆盖点的名字，`iff` 可以定义采样保护条件，`bins` 可以定义普通的分组柜，`ignore_bins` 可以定义忽略的交叉覆盖点，该类交叉覆盖点出现的时候不列入统计的范围，计数器不做自加；`illegal_bins` 可以定义非法的交叉覆盖点，一旦出现该类交叉覆盖点，系统会报告出错。

交叉覆盖点是其定义中覆盖点成员的分类柜的所有组合情况。如源代码 8-4 所示。

源代码 8-4 功能交叉覆盖点实例

```
//Chapter cross_example.sv
bit [3:0] a, b;
covergroup cov @ (posedge clk);
    aXb:cross a, b;
endgroup
```

覆盖组 `cov` 中定义了两个 4 比特变量 `a` 和 `b` 的交叉覆盖点 `aXb`。系统会自动为 `a` 和 `b` 两个变量分别创建两个覆盖点。每个覆盖点默认情况下是 16 个分组柜，那么因为 `a` 和 `b` 的分

组柜之间存在 256 种组合，所以交叉覆盖点 aXb 就会自动产生 256 个分组柜。

而对于表达式，我们上面提到，必须事先对其定义覆盖点，如下所示：

```
bit [3:0] a, b, c;
covergroup cov2 @ (posedge clk);
    BC: coverpoint b + c;
    aXb: cross a, BC;
endgroup
```

其中 BC 就是事先定义好的对应表达式 $b + c$ 的覆盖点，因为 $b + c$ 应该是一个 4 比特的数值，所以可以为该覆盖点分配 16 个分组柜；而 a 可以分配 8 个分组柜，最后交叉覆盖点 aXb，总共可以分配 128 个分组柜，来表示 a 和 $b + c$ 的各种组合情况。

交叉覆盖点中交叉分组柜的总数就是其定义的成员覆盖点分组柜的所有组合情况。如下所示：

```
bit [31:0] a_var;
bit [3:0] b_var;
covergroup cov3 @ (posedge clk);
    A: coverpoint a_var {bins yy[] = {[0:9]};};
    CC: cross b_var, A;
endgroup
```

这个例子中，我们对于变量 a_var 只关心 0 到 9 是个数值，为此在覆盖点 A 中显性的定义了 yy [0] …yy [9] 是个分组柜，作为交叉覆盖点 CC 中定义的成员分别是 b_var 和 A。系统会为变量 b_var 自动创建 16 个分组柜 (auto [0] …auto [15])。由于没有显示定义分组柜，系统会自动为交叉覆盖点 CC 分配交叉分组柜，其总数为： $16 * 10 = 160$ ；如下所示：

```
< auto[0], yy[0] >
< auto[0], yy[1] >
...
< auto[0], yy[9] >
< auto[1], yy[0] >
...
< auto[15], yy[9] >
```

为此，我们在定义的时候要小心使用交叉覆盖点，以免系统自动创建大量冗余的分组柜，因为每个交叉分组柜都是一个计数器，频繁采样统计，会对仿真速度带来影响。另外，我们也可以像覆盖点那样显示定义期望的分组柜。

其中关键字 binsof 可以为其指定的表达式（覆盖点或者变量）生成对应的分组柜，其结果可以通过和其他分组柜做进一步的选择操作，从而生成期望的交叉分组柜。如下所示：

```
binsof( x ) intersect {y}
```

表示覆盖点 x 中和给定 y 这个表达式的交集合，其反向表达式是：

```
! binsof( x ) intersect {y}
```

表示覆盖点 x 中和给定 y 这个表达式交集以外的范围。

下面我们通过一个具体例子来理解 binsof 是如何使用的。

```

bit [7:0] v_a, v_b;
covergroup cg @ (posedge clk);
a: coverpoint v_a
{
    bins a1 = {[0:63]};
    bins a2 = {[64:127]};
    bins a3 = {[128:191]};
    bins a4 = {[192:255]};
}
b: coverpoint v_b
{
    bins b1 = {0};
    bins b2 = {[1:84]};
    bins b3 = {[85:169]};
    bins b4 = {[170:255]};
}
c:cross a, b
{
    bins c1 = ! binsof (a) intersect {[100:200]};// 4 cross products
    bins c2 = binsof (a.a2) || binsof (b.b2); // 7 cross products
    bins c3 = binsof (a.a1) && binsof (b.b4); // 1 cross product
}
endgroup

```

这个例子定义了覆盖组 cg，其在时钟 clk 的上升沿采样。覆盖组中有两个覆盖点 v_a 和 v_b，分别对应着两个 8 比特的变量。覆盖点 a 和 b 分别自定义了四个分组柜，而交叉覆盖点恰好就是针对覆盖点 a 和 b 做交叉覆盖。若用户在交叉覆盖点 c 中不作自定义交叉分组柜，那么系统会自动为其生成 16 个交叉分组柜，分别代表着每个交叉组合： $\langle a1, b1 \rangle$ ， $\langle a1, b2 \rangle$ ， $\langle a1, b3 \rangle$ ， $\langle a1, b4 \rangle$ … $\langle a4, b1 \rangle$ ， $\langle a4, b2 \rangle$ ， $\langle a4, b3 \rangle$ ， $\langle a4, b4 \rangle$ 。

而实际上，在这个例子中用户自定义了三个分组柜。第一个用户自定义的交叉分组柜是 c1，其指定了覆盖点 a 中没有和数值范围 100 到 200 有交集的分组柜；首先有交集的分组柜是 a2, a3 和 a4；为此，c1 就应该是分组柜 a1 和 b 中四个分组柜的四种组合情况： $\langle a1, b1 \rangle$ 、 $\langle a1, b2 \rangle$ 、 $\langle a1, b3 \rangle$ 和 $\langle a1, b4 \rangle$ ；也就是说，其中任何一个组合被击中，分组柜 c1 的计数器就加一。

第二个自定义交叉分组柜 c2 指定了交叉项中或者存在分组柜 a2，或者存在分组柜 b2；那么总共可能有 7 个交叉分组柜： $\langle a2, b1 \rangle$ 、 $\langle a2, b2 \rangle$ 、 $\langle a2, b3 \rangle$ 、 $\langle a2, b4 \rangle$ 、 $\langle a1, b2 \rangle$ 、 $\langle a3, b2 \rangle$ 和 $\langle a4, b2 \rangle$ ；7 个交叉分组柜的集合就是 c2，只要上述任何一种情况出现，交叉分组柜 c2 的计数器加一。

最后一种用户自定义交叉分组柜 c3，其定义了交叉项必须同时包括 a1 和 b4，也就是 $\langle a1, b4 \rangle$ 。

另外，其他可能在自动生成交叉分组柜中出现的、但在本例中没有显示定义出来的 $\langle a3, b1 \rangle$ 、 $\langle a4, b1 \rangle$ 、 $\langle a3, b3 \rangle$ 、 $\langle a4, b3 \rangle$ 、 $\langle a3, b4 \rangle$ 和 $\langle a4, b4 \rangle$ 这 6 种交叉项不是交叉覆盖点关心的对象，其是否被击中不会在任何统计中影响交叉覆盖点 c。

由此可见，通过 binof 的结构可以筛选出我们关心的交叉项的组合，缩减交叉分组框的产生，更好地反映用户关心的对象，提高仿真效率。

8.3 覆盖率驱动的验证平台

在学习完 SystemVerilog 中功能覆盖率的语法结构之后，我们将把基于覆盖率驱动的方法学应用到石头、剪刀、布这个例子中。

首先，我们需要分析设计，拟定验证计划；针对这个仲裁器，两个 Player 的发出信息需要遍历不同的情况，而且要考虑两个 Player 发出信息的不同组合（9 种情况）；另外，就是要确定在哪个地方，什么时候做统计和采样。在这里，我们重新看看如图 8-1 所示的验证平台的结构。

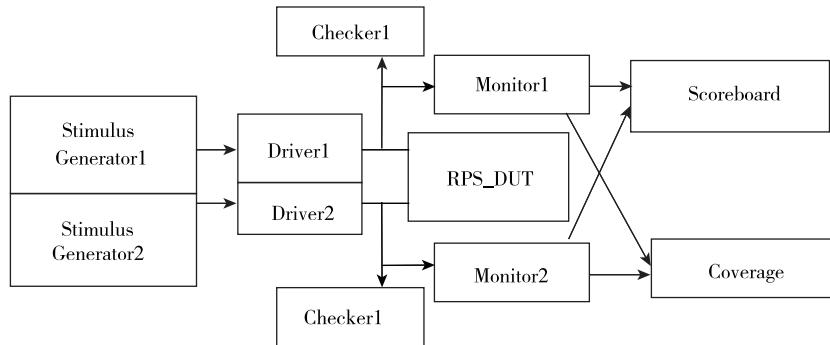


图 8-1 RPS 的验证平台

显然，在 Monitor 中我们搜集了两个 Player 发出的信号，为此我们可以将 Coverage 模块放在 Monitor 模块之后，并集成到 Scoreboard 中。

第一步：定义覆盖率统计模块，如源代码 8-5 所示。

源代码 8-5 功能覆盖模块（石头、剪刀、布）

```

//Chapter8 rps_coverage.sv
class rps_coverage;
  mailbox #(rps_c) fifol, fifo2;
  rps_c t1, t2;
  rps_t t1rps, t2rps;

  covergroup rps_cover;
    coverpoint t1rps {
      ignore_bins illegal = {IDLE};
    }
    coverpoint t2rps {
      ignore_bins illegal = {IDLE};
    }
    cross t1rps, t2rps;
  endgroup

  function new();
  endfunction

```

```

rps_cover = new ;
endfunction

task run;
forever begin
    fifo1.get(t1);
    fifo2.get(t2);
    t1rps = t1.rps;
    t2rps = t2.rps;
    rps_cover.sample;
end
endtask
endclass

```

从 run 这个任务中我们可以看到，首先我们将从 fifo1 和 fifo2 中取出对应 Player 的信息，赋给 t1 和 t2；接着将信息中的 rps 这个域的值赋给 t1rps 和 t2rps；最后直接调用覆盖组 rps_cover 的内置函数 sample 进行采样。为此，只有在从 mailbox 中可以取到包数据就可以做覆盖率的统计。

第二步：将覆盖率统计模型集成到 Scoreboard 中，并在任务 run 中调用 cov.run 进行覆盖率统计。

```

package rps_env_pkg;
...

`include "rps_coverage.sv"

class rps_scoreboard;
    mailbox #(rps_c) fifo1, fifo2;

    rps_c t1, t2;
    int score1, score2, tie_score;
    rps_coverage cov;

    int limit; //在配置的过程中初始化
    reg test_done; //等待外部测试完毕

    function new(mailbox #(rps_c) fifo1_i, fifo2_i);
        fifo1 = fifo1_i;
        fifo2 = fifo2_i;
        test_done = 0;
        score1 = 0; score2 = 0; tie_score = 0;
        cov = new();
    endfunction

    task run;
        fork
            forever begin
                fifo1.get(t1);
                fifo2.get(t2);
                update_and_check_score();
                cov.fifo1.put(t1); //发送到覆盖率统计模块 cov
            end
        join
    endtask
endclass

```

```

        cov.fifo2.put(t2); //发送到覆盖率统计模块 cov
    end
    cov.run();
join
endtask

local function void update_and_check_score;
    string str;
    bit win1, win2;

    // 验证参考模型和设计对应得分
    if(score1 != t1.score) begin
        $sformat(str, "MISMATCH - score1 = % 0d, t1. score = % 0d", score1, t1. score);
        $display("SBD % s", str);
    end
    if(score2 != t2.score) begin
        $sformat(str, "MISMATCH - score2 = % 0d, t2. score = % 0d", score2, t2. score);
        $display("SBD % s", str);
    end

    // 计算新的得分
    win1 =
        ((t1.rps == ROCK && t2.rps == SCISSORS) |
        (t1.rps == SCISSORS && t2.rps == PAPER) |
        (t1.rps == PAPER && t2.rps == ROCK));

    win2 =
        ((t2.rps == ROCK && t1.rps == SCISSORS) |
        (t2.rps == SCISSORS && t1.rps == PAPER) |
        (t2.rps == PAPER && t1.rps == ROCK));

    if(win1)
        score1 += 1;
    else if(win2)
        score2 += 1;
    else
        tie_score += 1;

    // 检查是否测试完毕
    if((t1.score >= limit) ||(t2.score >= limit))
        test_done = 1;
    $display (" time:% 0d SBD compare sucessfully, score1:% 0d, score2:% 0d, tie_
score:% 0d", $time, score1, score2, tie_score);
endfunction
endclass
endpackage

```

从上面的代码中可以看出，从 monitor 送过来的数据除了做比较以外还送入 cov 中做覆盖率统计。覆盖组的使用和类一样需要例化，用 new 创建分配。

第三步：修改验证环境的控制流程，把覆盖率作为仿真运行中止的标准之一。如源代码 8-6 所示。

源代码 8-6 基于功能覆盖率验证环境（石头、剪刀、布）

```
//Chapter8 rps_env.sv
class rps_env;

stimulus_generator s1, s2;
mailbox #(rps_c) f1, f2, ap1, ap2;
rps_driver d1, d2;
rps_monitor m1, m2;
rps_scoreboard sb;

...
task terminate;
  fork
    begin
      @ (posedge sb. test_done);
      s1. stop_stimulus_generation();
      s2. stop_stimulus_generation();
      $display("Test finished:reach 1000 times!");
    end
    begin
      check_coverage();
    end
  join
endtask

task check_coverage;
  forever
    begin
      if ( $get_coverage() == 100 )
        begin
          repeat (3) @ (posedge d1. dut_vf. clk_if. clk);
          $display("Final coverage report is %d %% ", sb. cov. rps_cover. get_inst_coverage());
          break;
        end
      else
        begin
          @ (posedge d1. dut_vf. go);
          $display("coverage report is %d", sb. cov. rps_cover. get_inst_coverage());
        end
    end
  //@ (posedge sb. test_done2;
  s1. stop_stimulus_generation();
  s2. stop_stimulus_generation();
  $display("Test finished:reach 100% % coverage!");
endtask
...
endclass
```

其中，我们添加了一个覆盖率查询的任务 `check_coverage`，这个任务中我们通过系统函数 `$get_coverage()` 查询整个验证环境中的覆盖率（其他与覆盖率相关的内置函数可参见附录 A），一旦达到 100% 就停止运行。同时，我们修改任务 `terminate`，将 `check_coverage` 集成进去，那么系统可能在两种情况下停止仿真：一是运行到 1000 次比较，`sb.test_done` 为 1，退出仿真；二是运行达到覆盖率为 100%，退出仿真。

断 言

本章重点介绍什么是断言、断言的应用以及如何将 SVA (SystemVerilog Assertion) 应用到具体项目中。本章不对 SVA 的语法做深入讨论，重点关注在验证流程中如何借助断言 (SVA) 去提高验证环节中的调试效率。其中，会介绍 OVL (Open Verification Library)，这是由 Accellera 组织提供的断言库。

9.1 断言的概念及作用

断言就是一段描述设计期望行为的代码。目前，对断言的使用主要在于仿真，但断言的能力不仅仅如此。断言是基于一些更加基础的信息，我们称之为属性 (Property)，属性可以用来作为断言、功能覆盖点、形式检查和约束随机激励生成。

从而，我们可以说断言就是对设计属性的一种描述。如果在模拟分析或者真实运行中，被检查的属性不符合我们的期望行为，那么这个断言就被宣告失败，若一个被认为是非法的属性，也就是不期望出现的属性发生了，那么断言也会宣告失败。

属性可以用在仿真器中，在硬件辅助加速平台中和形式分析工具中运行，可以是动态的或者是静态的，也可是两者的混合。随着在这个领域标准语言的出现，采用属性作为输入的工具，方法学和应用将越来越多。

断言可以嵌入到设计当中，也可以在设计以外通过绑定链接到不同的设计点中。断言查找期望的特定事件序列，或者说是在特定时钟周期内的事件。这些操作其实可以通过一个状态机来实现，设计工程师也可以通过硬件设计语言来实现，但是这种方法不利实现，难以调试和维护。因此，为了提高断言的效率，针对断言描述的标准语言（如 PSL 和 SVA），被用户广泛采用。

断言在验证过程的作用在于帮助客户更快的定位问题，帮助调试。就如前面提到的，验证平台存在一个局限性，就是对设计的可见度。所谓可见度就是，设计中异常的行为出现后，需要着重注意两点：一是这个错误是否能够被传递到顶层的输出；二是期望值和实际输出是否能够在比较后发现差异。对于很多验证平台，顶层的输出接口是有限的，为此不可能任何错误都可以传递到顶层并被捕获，另外，即使被传递出来，如何定位错误的根源也是十分麻烦的事情。断言有一个重要的作用就是可以提高对设计的可见度。它能够帮

助定位错误的根源，以保证调试能更加容易和快速地进行。这是因为断言可以分布在设计的各个重要部位，能及时捕捉与设计属性不一致的行为。

断言并不是一种新的技术，在软件行业已有 50 年的应用历史，同时在 VHDL 中也支持部分的断言功能。那么，断言是否适合硬件设计呢？特别是目前大型的 SOC 设计。从各个公司的统计来看，50% ~ 70% 的设计缺陷可以通过断言捕获，而且可以减少一半以上的调试周期。

断言可以应用在白盒调试或者黑盒调试两种方式中，一般情况下，白盒调试会被设计工程师采用，而黑盒调试会与验证工程师比较相关。作为设计工程师，他可以借助对设计的认识，对某些关键的设计部分添加断言，例如 fifo 的读写是否溢出，数据传输时握手协议是否符合约定等，通常在采用第三方的设计 IP，都会要求 IP 提供商附带提供该 IP 接口的断言模块，以保证在系统集成的时候，在对接部分的设计能够符合 IP 接口的要求。作为验证工程师，主要是在更高一个层次架构验证平台和验证策略，其未必对设计的细节很了解；为此，设计的大部分模块对他们来说是黑盒子，验证工程师关注的是各个模块之间的连接，以及设计顶层的接口是否符合设计规范，例如：AMBA 接口设计是否符合规范，顶层以太网接口是否符合规范，上电复位的过程是否正常等。

前面我们讨论了，断言既可以嵌入在设计代码中，也可以独立在设计以外。那么，我们有哪些方式可以支持断言呢？第一，我们可以采用预建的断言库，如 OVL；第二，用户可以自定义断言，例如采用 SVA 或者 PSL。在此，我们优先推荐采用断言库，而不建议用户学习断言详尽的语法，从而自定义断言。为什么呢？

1) SVA 和 PSL 虽然是专门提供给用户定义断言的编程语言，其优点是语法简练；其缺点是在简练的语法下全面地表达特定的、复杂的电路行为，对语法或者电路行为理解不深刻的人，很难写出比较有实用价值的断言。

2) 断言是设计属性的表现，也就是采用 SVA 和 PSL 表示出来的是设计的意图，这就存在证明断言代码是否正确的过程；断言没有办法自我证明，那么只能通过检视（review）和设计仿真过程中的相互验证，也就是在这个调试过程中，可能是设计上的错误，也可能是断言写错了，或许你大部分的时间不是在调试设计，而是在调试断言本身。

3) 验证对于调试周期有很高的要求，为了缩短验证周期，首先，我们需要能更快地暴露错误的行为，这个可以通过随机激励生成，通过大量的随机数据测试尽可能多地设计空间；其次，在错误暴露的时候，我们如何定位到这个错误，断言就是在这个阶段起作用，若我们能够采用已经被验证过的断言库，我们就可以遵从验证重用的思想，做到实实在在的提高验证效率。

综上所述，为了缩短调试周期，我们要有采用断言的思想，而不是陷入在长篇累牍的断言语法中。

Accellera 是一个电子行业组织，其主要的任务是推动行业在世界范围的发展，推广使用业界中系统提供商、半导体公司和设计工具公司共同制定的标准，从而提高设计自动化的流程。Accellera 组织主要负责新标准的认定，开发新的设计格式，并且推广新的方法学的采用。Accellera 已经批准了几个断言的标准，其中包括 PSL，OVL 和 SystemVerilog。

OVL (Open Verification Library) 是一个统一格式，能够捕获功能行为规范的标准库，

并且可以在不同厂家的工具中使用。OVL 定义了大概 30 个检查器，采用 Verilog、VHDL、PSL-Verilog、PSL-VHDL 和 SVA 等各种版本，每种版本都有相同类型的检查器。OVL 定义了一些白盒验证经常用到的检查器，为此十分实用。虽然这个标准库的内容是有限的，但是用户可以组合不同的检查器，扩展自己需要的断言。用户只要例化、配置，就可以构建自己的需要的断言，而不需学习一门新的语言。

SystemVerilog Assertion 提供了定义序列（sequence）、属性（property）等能力，后面我们会深入讨论。PSL 也是一门标准化的说明性语言，其来自于 IBM 开发的 Sugar 属性语言。后来，被 Accellera 采用并开发为 PSL，而后被标准化为 IEEE1850。

在后面的章节中，我们将对 SVA 做简要的介绍，并且重点介绍如何采用 SystemVerilog 中的 bind 结构，使预定好的断言模块可以方便的绑定到特定模块中。

9.2 SVA

SVA（SystemVerilog Assertion）是 SystemVerilog 中一个重要的语法子集，其简练的语法可以描述复杂的时序行为。

我们先通过一个例子来说明，SVA 断言为什么要能够比普通的 Verilog 和 VHDL 更加简练的表达设计的时序行为。

期望行为就是我们常见的握手信号，在请求信号 req 拉高之后，确认信号 ack 在一个到三个时钟后拉高。如图 9-1 所示。

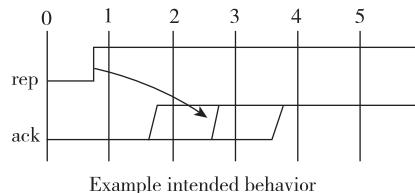


图 9-1 握手协议的期望行为

若采用 Verilog 来描述这个期望行为，会是 16 行代码，而且还要用到 fork...join 这种并行结构：

```
always @ (posedge req)
begin
  repeat (1) @ (posedge clk);
  fork : pos_pos
    begin
      @ (posedge ack)
      $display("Assertion Success", $time);
    disable pos_pos;
    end
  begin
    repeat (3) @ (posedge clk);
    $display("Assertion Failure", $time);
    disable pos_pos;
  end
end
```

```
join
end
```

若采用 SVA，其将十分简单，只要短短的四行代码，可见其简练和高效。核心部分（第二行）只用了一条语句就表示了重要的期望行为。

```
property req_ack;
  @ (posedge clk) req ##[1:3] $ rose(ack);
endproperty
as_req_ack: assert property (req_ack);
```

在此，我们先区分一下断言的两个分类：立即断言和并发断言。

```
always @ (posedge clk)
  traf_light:assert (green && ! red && ! yellow && go );
```

如上例所示，在过程化代码（initial/program/function/task）中使用的 assert，其中没有任何时序概念的为立即断言。握手协议的例子就是并发断言，是基于周期采样的，有一定时间跨越的设计行为。我们重点讨论并发断言。在学习 SVA 的语法结构之前，需要提醒大家，并发断言是一种基于周期的语法结构，也就是需要采样事件去激发周期性的数值采样，为此对同步时序电路比较有效。如上面的例子所示，该断言采用时钟 clk 的上升沿采样。既然如此，在一个时间间隙中，进入前将做上一个时间间隙采样的断言评估，在这个时间间隙退出的时候做这个间隙的数值采样。也就是说，若采用周期性时钟做采样，断言告警相对实际的设计行为会有一个周期的滞后。

9.2.1 SVA 的语法层次结构

这一节我们主要介绍 SVA 的语法结构，熟悉了语法结构之后，用户就可以直接在设计或者验证中采用 SVA 了。

SVA 的语法结构主要分五个部分，如图 9-2 所示。最底层是布尔表达式，这个和 Verilog 中没有差别；第二层是序列（sequence），其中可以包含一些新的操作符，如##时隙延迟、重复操作符、序列操作符等，序列是一个封装格式，采用序列封装后可以在不同地方使用，一个序列会被评估为真或者假；第三层是属性（property），这是重要的封装方式，其中最重要的特点是属性内部可以定义蕴涵操作符（ \rightarrow 、 \Rightarrow ）。第四层是断言指示层，也就是采用 assert 对特定属性或者序列做行为检查，或者采用 cover 做统计等。第五层是断言的最后封装，只有通过最后封装成一个单元的断言才可以在不同的地方重用，就如同一个可以例化模块或者类，通常这一层可以通过 module 或者 program、interface 来封装。

首先，我们介绍两个断言中常见的操作符：##和 [m:n]。##n 表示延时 n 个时钟周期，如图 9-3 所示，其中 b 为真之后，隔两个周期 c 也要为真。



图 9-2 SVA 的语法层次结构

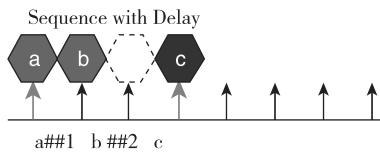


图 9-3 ##时延操作符

[m:n] 可以指定窗口，假如和##一起使用，如下面的##[1:5] 表示，后续事件在[1:5]个周期的这个窗口内出现都可以判定为真。如图 9-4 所示，##[1:5] z 等效于：##1 z 或##2 z 或##3 z 或##4 z 或##5 z，只要其中一种情况为真，则##[1:5]z 为真。

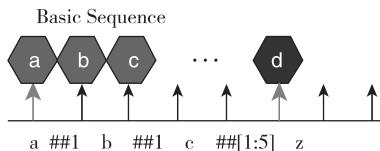


图 9-4 [m:n] 指定窗口

在此，我们讨论一下 sequence 和 property 在断言中的一个重要区别，以及蕴涵操作符的作用。

我们仍然采用前面提到的前一节的握手协议作为例子，分两种断言方式来实现。我们首先把期望行为用 sequence 来封装，如源代码 9-1 所示。

源代码 9-1 断言序列实例

```
//Chapter9 sequence_example.sv
sequence s1;
  @ (posedge clk) req ##[1:3] $rose(ack);
endsequence
sva_1:assert property (s1);
```

上述断言 sva_1 描述了什么呢？它描述的是：在每个时钟周期必须采样到 req 为 1，若成立就会激发一个进程，这个进程就会在 1 到 3 个时钟周期内检查是否出现 ack 的上升沿，从前面的图 9-1 中我们可以看到，req 在时钟时隙 1, 2, 3, 4, 5…都是为 1 的，那么也就是，断言 sva_1 会在对应时刻激发对应的检查进程，分别对应于 1, 2, 3, 4, 5…；这是我们期望的么？应该说，超出了我们的期望，其实我们只需要在一次 req 拉高之后检测一次对应的 ack 确认信号就可以。若采用 sva1 作为检查，就会产生很多虚假的错误（vacuous failure），因为时隙 2, 3, 4, 5…之后的检查进程是没有必要激活的，出现断言报告行为错误也是正常的。同时存在另外一种情形是，若 req 没有被置 1，sva_1 也会发出虚假的错误，因为 req 不被满足。注意，若 s1 采用 property 来封装，则会有类似的虚假结果。

property 有一个很大的特点就是可以定义蕴涵操作符，如源代码 9-2 所示。

源代码 9-2 断言属性实例

```
//Chapter9 property_example.sv
property p1;
  @ (posedge clk) req |> ##[1:3] $rose(ack);
endproperty
sva_2:assert property(p1);
```

蕴涵操作符号的作用是，它首先对其前置表达式（req）做检查，不成立则不激发后置检查进程（##[1:3] \$rose(ack);）；若成立，则激发后置表达式的检查。这样就避免了上面我们提到的在每个采样时钟，若 req 不为 1，总会产生虚假的错误信息的情况。但是，仍然没有消除前面提到的第一种情况：产生冗余进程而导致虚假的错误信息。我们再进一步优化，如下：

```
property p2;
  @ (posedge clk) $rose(req) |> ##[1:3] $rose(ack);
endproperty
sva_3:assert property(p2);
```

断言 sva3 的描述是，在检测到一个 req 的上升沿之后的 1~3 个时钟周期内会有一个 ack 的上升沿。

至此，我们已经分析了采用 sequence 和 property 的不同之处，以及 property 中的蕴涵操作符。其中，“|>”表示当前置表达式成立后同一时钟检查后置表达式，“|=>”表示要延迟一个时钟，等效于 |> #1。

在我们的断言定义完毕之后，如何将其封装重用呢？首先，sequence 和 property 都是可以定义接口信号的，为此，sequence 和 property 中定义的信号可以在例化的时候做具体连接。现在我们采用 module 对上述断言 sva_3 做如下封装。

```
module sva_test(input clk,req,ack);
  property p2;
    @ (posedge clk) $rose(req) |> ##[1:3] $rose(ack);
  endproperty
  sva_3:assert property(p2);
endmodule
```

在不同的设计中，我们通过模块例化对特定的接口做分析，如下面所示：

```
module testbench();
  ...
  dut(req,ack...);
  sva_test sva_ins(clk,req,ack);
  ...
endmodule
```

这就完成了断言的顶层封装和例化。从这个例子可以看到，这种例化方式需要嵌入在设计当中，也就是端口对于断言是同一个层次或者说是直接可见。这会影响设计的独立性，因为我们总希望可以将设计代码和验证代码分离，保持独立性，也是可重用性和可维护性

的一个重要要求。从另外一个角度来说，在验证平台的顶层，像 sva_test 这样用模块封装的断言能否不影响设计结构，直接探测到设计内部某个例化层次的接口上呢？答案是可以的。下面我们将通过一个石头、剪刀、布的例子来讲述，如何采用 bind 这个结构，让 SVA 断言和设计分离。

9.2.2 SVA 应用实例

首先我们来看看这个石头、剪刀、布仲裁器的接口，其中分 play1 和 play2 两组，如图 9-5 所示。

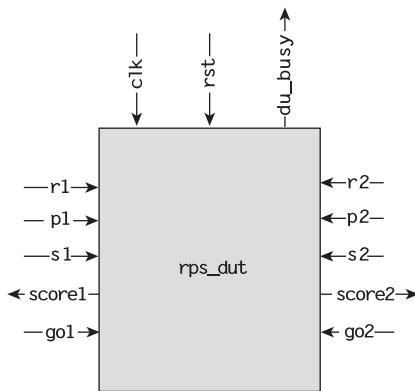


图 9-5 RPS 的设计

其中一个重要的功能就是在 go1 和 go2 同时为 1 的时候，rps_dut 需要把 dut_busy 置为 1，以防止后续数据改变信号 r/p/s，影响本次的比较。

我们把这个属性用断言来描述，并封装在 module 中，如源代码 9-3 所示。

源代码 9-3 断言模块（石头、剪刀、布）

```

//Chapter9 rps_sva.sv
module sva_unit(input sva_clk, sva_go1, sva_go2, sva_dut_busy);
    property rps_busy;
        @ (posedge sva_clk) (sva_go1&&sva_go2) |-> sva_dut_busy;
    endproperty
    rps_sva: assert property (rps_busy);
endmodule

```

为了更能显示 bind 的功能，我们把 rps_dut 再做一个封装，增加其层次结构。

```

//Chapter9 rps_dut_wrapper.sv
module rps_dut_wrapper(input bit r1, p1, s1, input bit r2, p2, s2, output int score1,
    score2, input bit go1, go2, input bit clk, rst, output bit dut_busy);
    rps_dut dut(. *);
endmodule

```

在验证环境顶层 rps_class_based 中，我们采用 bind 结构将 sva_unit 绑定到两个层次以下

的 rps_dut 中，如源代码 9-4 所示。

源代码 9-4 基于断言的验证顶层（石头、剪刀、布）

```
//Chapter9 rps_tb_top.sv
module top_class_based;
  import rps_env_pkg::*;
  `include "interface.sv"
  `include "rps_driver.sv"
  `include "rps_monitor.sv"
  `include "rps_env.sv"

  rps_clk_if clk_if();
  rps_clock_reset cr(.clk(clk_if.clk),.rst(clk_if.rst));

  rps_dut_pins_if pins1_if(clk_if);
  rps_dut_pins_if pins2_if(clk_if);

  rps_dut_wrapper
  dut_wrapper(.r1(pins1_if.r),.p1(pins1_if.p),.s1(pins1_if.s),.r2(pins2_if.r),
  .p2(pins2_if.p),.s2(pins2_if.s),.score1(pins1_if.score),.score2(pins2_if.score),
  .go1(pins1_if.go),.go2(pins2_if.go),.clk(clk_if.clk),.rst(clk_if.rst),.dut_busy
  (clk_if.dut_busy));

  rps_env env;
  bind dut_wrapper.dut sva_unit sva_instance(clk,go1,go2,dut_busy);
  initial begin
    env = new(pins1_if,pins2_if);
    fork
      cr.run();
    join_none
    env.execute();
    $stop;
  end
-----
```

从上面的 bind 语句可以看到，bind 可以把 sva_unit 绑定到层次化结构 dut_wrapper.dut 这个层次，其绑定的例化名为 sva_instance，而且 sva_unit 的接口可以从 dut_wrapper.dut 这个例化中获取 clk/go1/go2/dut_busy 的信号。其实，bind 的能力不仅可以捕获 dut_wrapper.dut 这个例化接口上的信号，还可以探测到这个例化内部的信号。

为了能够体现断言的作用，大家可以修改 rps_dut 这个模块 dut_busy 信号的生成，故意制造随机错误，以便观察到断言监控功能。

将

```
assign both_ready = (go1 & go2);
```

修改为

```
assign both_ready = (go1 & go2 & ( $random() % 1 ));
```

通过本例了解 bind 的应用之后，下一节我们将学习 bind 的基础知识。

9.2.3 bind

为了能够实现验证和设计分离，SystemVerilog 支持通过 bind（绑定）将断言属性封装模块绑定到任意的设计模块或者其特定例化中。这个功能可以实现以下几个目的：

- 验证工程师可以最少的改动原有的设计代码和文件结构。
- 验证 IP 可以方便的绑定到特定的设计模块或者例化中。
- 对现有的断言没有任何语法上的影响，断言可以通过这种方式实现层次化的访问。

通过这个方法，用户可以将一个 module、interface 或者 program 绑定到一个模块或者其例化中。bind 的语法结构如下：

```
bind hierarchical_identifier container_select bind_instantiation;
```

对该语法，我们可以用图 9-6 来说明，target 一般是设计对象，可以是模块名或者是例化名，也就是我们可以通过层次性的例化名访问设计内部例化。container 是断言的封装载体，可以使用 module/interface/program 来封装；例化名就是本次绑定的例化名。



图 9-6 bind 语法结构

绑定语句可以在哪些位置定义呢？可以在 module/interface 或者在编译层次内定义。我们再看一个简单的 bind 的例子，以便进一步理解 bind 的使用方法。

```
bind cpu fpu_props fpu_rules_1(a,b,c);
```

其中：

- cpu 是设计的模块名。
- fpu_props 是一个封装了属性/断言的 program/interface/module。
- fpu_rules_1 是例化名。
- 例化端口 (a, b, c) 是获取来自 cpu 模块的信号，采用位置关联的例化方式。
- cpu 的每个例化都被绑定了这个断言。

当然，我们可以针对某一个例化做绑定，这时需要指定层次化的例化名，如下例所示：

```
bind top.cpu_instance1 fpu_props fpu_rules_1(a,b,c);
```

最后，我们再看看对一个计数器实例是如何使用 bind 实现 sva 绑定的，如源代码 9-5 所示。

源代码 9-5 bind 操作实例

```
//Chapter9 bind_example.sv
module counter #(parameter WIDTH=8)(
    input clk , rst_n, ld_n, up_dn, cen,
    input [WIDTH-1:0] din,
    output tc, zero,
```

```

output reg [WIDTH-1:0] dout
);
...
endmodule
module testcounter;
...
endmodule
module counter_props #(parameter WIDTH=8)(
  input clk, rst_n, ld_n, up_dn, cen, tc, zero,
  input [WIDTH-1:0] din, dout
);
...
assert property (verify_reset) else $error( "The counter reset functionality failed");
assert property (verify_load) else $error( "The counter load functionality failed");
assert property (verify_hold) else $error( "The counter hold functionality failed");
assert property (verify_count) else $error( "The counter counting functionality failed");
assert property (verify_count_lv) else $error( "The counter counting functionality failed");
endmodule

bind counter counter_props #(8) counter1_bind
  (.clk(clk),
   .rst_n(rst_n),
   .ld_n(ld_n),
   .cen(cen),
   .up_dn(up_dn),
   .din(din),
   .tc(tc),
   .zero(zero),
   .dout(dout)
 );

```

通过本章的学习，有如下几点要特别注意。

- 1) 断言描述的是设计的属性，可以表达设计意图，断言可以采用 VHDL、Verilog、PSL、SVA 等多种语言来描述，可以应用在动态仿真或者静态分析中。
- 2) 断言主要的作用是帮助定位错误，缩短调试时间，为此，断言本身应该是正确的，否则有可能会花很多时间去调试断言，而非设计。
- 3) PSL、SVA 这些专门用来描述断言的语言，其特点是可以更加简练的描述设计行为，特别是跨越多个时隙的行为；但是要注意的是，这些简洁的语法未必都可以综合成网表；在某些特殊的情况，需要在真实硬件原型内做调试的时候，我们也可以采用 Verilog 或者

VHDL 去写一些可综合的断言。

4) 为了保证验证和设计分离，也就是验证过程中不要对原有设计对象代码和层次结构有影响。SystemVerilog 提供了 bind 这个语法，用户可以将封装好的断言绑定到任意的设计模块或者其某一个例化层次。

5) 设计工程师和验证工程师对断言有不同层面的视角，设计工程师对自己的设计细节比较熟悉，他们可以为设计内部的关键模块或者模块接口编写断言；作为验证工程师，面对的是系统级的设计，他们关心的是一些标准的片内总线结构、顶层接口和 IP 核之间的接口。

6) 重用是验证的核心思想，采用预建的断言库，例如 OVL，由于这些库都是经过验证，可以缩短断言开发周期，提高验证效率。

验证重用与验证方法学

这一章我们讨论实际项目中验证工作所面临的挑战，重用对验证的意义，以及介绍由 Mentor Graphics 公司和 Cadence 公司共同推出的 OVM (Open Verification Methodology)，着重介绍该方法学中提供的高级技术是如何解决验证挑战，实现一个重用验证平台的。

10.1 验证重用中存在的问题

第 1 章我们简单介绍了一系列验证中存在的挑战，对于刚刚入门从事验证工作的工程师或许有些抽象，为此，这一章我们将列举实际项目中可能出现的具体问题。

案例一：在验证的后期，设计规范发生更改，设计代码有部分改动，要求验证平台中的某一个接口的驱动器（transactor）要添加或者修改一些功能。

传统的解决方案：直接修改验证环境中该驱动器的代码。可能出现的问题，就是在修改的代码时候会潜在的引入其他错误；另外，修改的代码可能无法兼容原环境中各个已开发的测试用例；若需要添加附加参数还可能会需要修改整个验证环境的连接关系、架构和部分代码，扩大了牵涉范围。

挑战一：对某个验证组件进行修改或者添加功能的时候，如何确保对该组件的修改不会引入其他错误，不会对原有测试用例、测试平台有影响？

我们可以用类的扩展继承，也可以采用 callback 或者 factory 的方式来解决这个问题。OVM 提供了 Factory 的功能，基于 ovm_component 和 ovm_sequence_item 的类都可以被注册，灵活的按需生成特定的对象。

案例二：验证组件新增的某个功能，需要从顶层某个控制模块配置参数。

传统的解决方案：直接修改代码，添加验证组件的接口信号，以便接受来自其他模块的配置参数。这会导致原有验证平台的需要添加和修改连接关系。

挑战二：如何应对特殊情况下的参数配置，而不影响原有整个验证平台的结构？

OVM 验证方法学提供了跨层次的参数配置机制，可以避免对原有验证平台架构的修改。

案例三：验证过程中，关键的就是测试激励的生成，只有生成更多的激励和激励序列才能测试更多的功能点，覆盖到更大的验证空间。在激励生成算法开发的过程中，如何才能重用原有激励的算法，组合成不同的激励序列，产生更多的激励情景呢？具体来说就是，假如我们测试的是一个存取器（memory），已开发了对存取器的读操作和写操作，那么我们如何在两个操作的基础上组成读-写-读、读-读-写、写-读-写-读等不同的激励序列（测试情景）？

挑战三：激励生成的算法和机制如何实现组合重用？

OVM 验证方法学中提供了 `ovm_sequence` 的基类让用户可以定义激励生成算法和机制，灵活生成各种激励序列，并且通过 `ovm_sequencer` 的基类实现对各种激励的管理和仲裁。

案例四：在验证工程师搭建完验证平台后，多个工程师能否基于这个公共的平台开发自己的测试用例？这是最基本的，而且是验证工程师需要关注和思考的问题。测试用例的开发者，在创建测试用例的时候主要关心的就是能否在顶层配置环境，按照测试需求配置对应的激励，或者添加特殊的激励生成机制，而对原有平台不会有太大影响。

挑战四：测试用例开发者能否在验证平台上控制和配置激励的产生？能否做到测试用例和测试平台分离，使多个测试用例能共享一个验证平台？

OVM 验证方法学提供了测试用例和验证平台分类的结构，验证工程师可以轻松地实现这个功能。

目前，设计重用已经广泛应用；一个 SoC 设计可以利用成熟 IP 核的实现快速系统集成，例如，一部手机基带芯片中可能集成 ARM 核、DSP 核、RF 模块、外设接口 IP 和各种存储器模块。通过基于 IP 重用的方法，设计周期大大缩短，而验证却面临巨大的时间压力。如何在最短的时间内搭建一个可重用的验证平台是极具挑战的任务。

验证重用沿用了设计重用的思想，其中在一个项目中我们需要重点考虑三大问题：验证组件是否能重用（易于扩展，方便集成，灵活配置），激励生成算法和机制是否可以重用，验证平台是否可以被多个测试用例重用。

10.2 验证方法学 OVM

Merriam-Webster 是这样定义方法学（methodology）的：“在某种科学、艺术或学科采用的方法、流程、运作概念、规则和基本原理的个体”。功能验证方法学也就是验证电子系统的艺术和科学。一般情况下，它提供了一个开发验证平台的架构，可以在构造验证平台过程中使用的、带有基类和应用函数的库，以及解决常见问题的策略和手段。

业界中现有 OVM 和 VMM 两个主要的验证方法学，这里我们对业界第一个开放式的 OVM 验证方法学做简要的介绍。如图 10-1 所示，OVM 类库提供了搭建一个验证平台的所有构建模块，组件可以被封装，层次性例化和通过一系列可扩展的阶段来初始化，最后运行并完成一个测试。

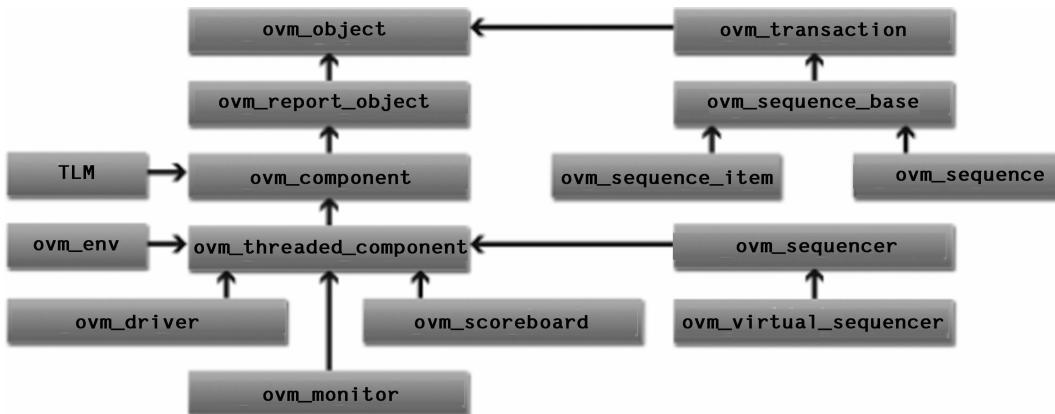


图 10-1 OVM 的类库层次结构

10.3 OVM 的四大核心技术

搭建一个高级的验证平台有很多要求，其中一个重要的目标就是验证平台可重用。验证平台的可重用性体现在以下几个方面：模块级验证到系统级验证的重用，同一个项目下不同测试环境的重用，在公司内部不同项目之间的重用，在不同公司之间的重用；从重用对象的角度来看，存在着验证组件的可重用、事务激励和产生算法的可重用等。

可以说，任何验证平台都是可重用的，这取决于我们准备投入多少精力来使之可重用。其中，有如下几个概念可以帮助我们通过适当的投入就可以实现验证平台的可重用。

- 抽象并标准化验证组件的通信接口
- 具有标准 API 的验证组件
- 标准化的事务交易
- 封装
- 验证平台架构的动态构建
- 动态的配置机制
- 测试作为验证的顶层
- 激励产生与验证架构分离

OVM 提供了事务交易级的架构，借助这些事务交易级的接口，可以搭建模块化、可重用的验证组件；它的类库可以帮助使用者创建约束随机的激励和序列，搜集和分析功能覆盖信息，包括断言在内的可配置和层次性管理的验证环境；其中，基于 Factory Pattern 的对象生成方式、验证平台架构的动态构建、动态的配置机制、激励产生与验证架构分离和测试作为验证的顶层四大核心技术使得可重用的验证平台更易实现。这四大核心技术将分节介绍。

1. 抽象并标准化验证组件的通信接口

OVM 有一个 TLM 子库，这是基于 OSCI（Open SystemC Initiative）的 TLM 标准的。这个标准定义了一系列的接口及其操作方法和通信机制，并把接口定义和功能的实现分离开来。

TLM之间的通信是在 port 和 export 之间进行的（如图 10-2 所示），两者的连接在上层完成，为此上层的 connect（）阶段代码可以被重用来完成相同通信接口的连接，在 end_of_elaboration（）阶段所有的连接绑定会被自动检查和确认。采用 OVM 的 TLM 接口进行通信，可以让验证平台的组件之间有统一的、标准的接口，具有相同 TLM 接口的验证组件可以相互替换而不影响整个验证环境，从而实现可重用性。另外，基于事务交易级的验证组件可以提高仿真速度而且容易调试和维护。



图 10-2 port 到 export 的 TLM 连接

2. 具有标准 API 的验证组件

验证组件是验证平台中的重要组成部分。例如，验证组件可能是一个记分板或者事务处理器。OVM 提供了基类 ovm_component，所有的验证组件可以从中扩展出来。该基类提供了应用编程接口 API，验证组件可以继承和扩展。API 包括例如报告、层次化信息管理和仿真阶段。API 提供了标准的接入方法，可以供所有的验证组件使用。

3. 标准化的事务交易

事务交易也就是激励对象，不是验证平台架构中的组成部分，而是对 DUT 激励的一个载体，如以太网的数据包对象。OVM 提供了可以被扩展定义事务交易的基类 ovm_transaction 和 ovm_sequence_item。这些基类为激励提供了标准的 API，为此验证组件可以对所有类型的事务交易做类似的操作。API 包括了激励对象的创建、克隆、复制和信息打印等。

4. 封装

OVM 提供了层次化的验证组件和一个标准 API 可以访问和管理层次化信息。这使得几个组件可以被封装为一个整体，从而实现可重用性。例如，一个层次化组件可能封装了所有的分析组件或者激励生成组件。封装可以是一组共有基类的验证组件的实现多态，而不仅是一个组件。

10.3.1 基于 Factory 的验证平台动态构建

在验证中经常遇到可重用性的难题就是需要及时调整特定的验证平台环境，对 DUT 应用一系列新的功能测试。通常，我们通过修改已有验证平台中特定的验证组件源代码，得到一个新的验证环境。例如，通过用一个有注错功能的驱动器去替换通用的驱动器。这很容易就可以通过面向编程语言的技术，在例化原驱动器的位置上修改代码，例化一个有注错功能的驱动器，从而扩展基类环境到一个新的环境。假如两个驱动器接口都是兼容的，那么验证环境的其他部分的代码便可以保持不变。

传统上，层次化基于类的对象产生创建是通过一个对象的构造函数 new（）来实现的。高层次的组件通过调用低层次组件的构造函数，创建低层次组件的对象。因为对象的类型

在编译的时候就已经确定了，所以这种方法限制了创建对象的灵活性。另外，我们需要维护两个不同的验证环境，虽然我们可以通过面向编程技术使原有的代码得到重用，但是我们更加希望整个验证架构也能够重用。也就是说，我们希望可以不改动任何原有的代码，但可以通过外部配置调整其内容从而实现一个验证架构的重用。在 OVM 中我们可以通过 Factor Pattern 的方法来实现。

Factory Pattern 是一个很出名的面向对象的编程技巧，Factory 是一个可以动态创建对象的类。它的主要好处是可以在特定的时刻创建特定的对象。Factory 不是验证组件层次中的一部分，而是处于层次化结构之外。OVM 提供的 Factory 用来创建任何类型的事务交易或者任何验证组件，只要事前它们在 Factory 做过注册。Factory 提供的类型重写可以动态地改变所创建对象的类型。在 OVM 中我们通过 ovm_factory 来具体实现，例如：

```
class my_env extends ovm_env;
    drv d1,d2;
    ...
    function void build();
        ... //搭建环境的其他部分
        d1 = new ("d1",this); //显示使用构造函数 new()
        assert ($cast(d2,create_component("drv","d2"))); // 使用 factory 方法 create_component
                                                // 创建 drv 类型的对象
    endfunction
    ...
endclass
```

我们从上面的代码可以看到，d1 和 d2 采用了不同的例化和构造方式。d1 这个对象是通过调用其构造函数来实现的，这限制了可重用性。相反，Factory 的 create_component() 方法返回了一个 drv 类的对象并且赋给了 d2。这两段代码都是一个 drv 的例化对象被创建，但是 Factory 提供了额外的灵活性，因为它可以在 my_env 类以外对其进行控制，也就是在其上层的 ovm_test 例化中进行操作，从而可以从 Factory 中返回一个期望类型的组件给 driver 的例化对象，例如：

```
class err_test extends ovm_test;
    my_env env;
    function void build();
        ovm_facotry::set_type_override("drv","err_drv"); //factory 的类型重写方法
        env = new ("env");
        ...
    endfunction
    ...
endclass
```

set_type_override() 方法告诉 Factory：一旦验证环境通过 create_component() 要求一个 drv 基类的对象时，请为其返回一个 err_drv 类的例化对象。在 Factory 中也可以针对特定的例化对象做类型修改。这个机制使得一个相同的验证环境类可以被例化到多个测试中，每一个测试都可能要求一个不同类型的 driver 的继承，但是环境的代码没有改变。这个环境本身是一个可重用和上下文相关的（取决于测试如何控制 Factory 去生成相应类型的对象）。

当在各个层次的 build() 阶段都采用这种方式时，每个上层的组件通过 Factory 去创建一个子组件的例化对象，那么任何一个组件就可以通过类似的方式来指定需要的类型。

10.3.2 动态的配置机制

传统的层次化是在基于类的环境中使用构造函数的参数来配置验证平台。例如验证平台的架构、参数设置（数组的大小和常数）、操作模式（错误注入和调试）是可以通过这个方法来配置的。但是在多层次的结构中，这种方法使用起来变得困难而且很难添加新的参数。

OVM 支持内置动态的配置机制，可以实现结构化属性和运行时参数的配置，从而避免了通过构造函数的参数来传递信息。一个高层次的组件可以设置配置信息，这些信息被对应的低层组件获取后使用。每个可配置的组件负责在合法的时刻去获取自己的配置信息：结构化配置信息，例如多少个子模块可以被例化，可以通过 build() 这个阶段来控制；运行时的信息，例如在总线周期之间需要等待多少个状态才注错，可以通过组件的 build() 或者 configure() 阶段来实现，它们也可以在 run() 这个阶段来实现。OVM 提供了一个 API 来实现这个配置信息的设置和获取的过程。

例如：

```
class drv extends ovm_threaded_component;
    local int delay;
    virtual function void build();
        if (get_config_int("numdly", numdly)) //从全局配置表中获取配置信息
            set_delay_length(numdly);
    endfunction

    virtual task run();
    ...
    case (state)
        DONE:begin
            if (get_config_int("numdly", numdly))
                if (numdly <= 10)
                    set_delay_length(numdly);
                else ovm_report_error("Driver", "ILLEGAL length sepcification");
            state = IDLE;
        IDLE:begin
            if (delay == 0)
                state = GO;
            ...
        endcase
        ...
    endtask
endclass
```

通过控制自身的配置设置，一个组件能够保证它只有在合法的途径下才可以获取信息；而差的编程方法则会在一个总线交易的过程中，让其他组件来改变运行时间参数。

配置信息可以通过高层的组件来指定，而常常由顶层的验证环境或者顶层的测试来决

定；通过 `set_config_*`()，配置也可以在一个特定例化上实现；配置信息可以直接被制定为整数或者字符串的值，但是有些比较复杂的需要封装在一个 `ovm_object` 的对象中，通过使用 `set_config_object()` 方法来实现。下面是一个例子，测试将配置 driver 中的延迟时间：

```
class dly_test extends ovm_test;
    virtual function void build()
        set_config_int("env.d1","numdly",5);
        ...
    endfunction
    ...
endclass
```

在 OVM 中这些配置是通过一个全局的查找表来实现的，这为验证平台提供了可重用性。第一，对组件的配置信息独立于自身的构造函数，这使得测试可以更灵活地根据其他配置信息或者随机去为还未被例化的组件配置信息。使用者还可以通过使用通配符来在多个组件中对多个参数做配置。组件自主获取其配置信息可以让组件保证无论在哪种情况下都要被合法的配置。如果其中有不匹配的地方，组件作为一个仲裁者，会要求恰当的配置。

10.3.3 测试用例在验证架构的顶层

标准验证平台的结构中有一个顶层的模块 (`top`)，在顶层模块中例化了 DUT (`alu`)，DUT 接口 (`alu_if`) 和一个顶层的类 (`test_env`)；顶层的类 (`test_env`) 即验证环境中包含了验证平台的所有组件，可以在这个架构中应用 SystemVerilog 技术，例如约束随机数的产生和功能覆盖率。

在如图 10-3 所示的标准验证架构中，顶层的对象是一个验证环境类，其中嵌入了激励产生器。这限制了添加和修改测试用例的灵活性。

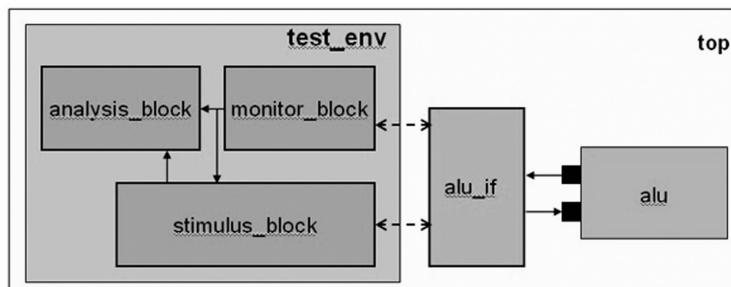


图 10-3 标准验证平台的顶层结构

将激励生成算法从验证平台的结构中分离出来，这样可以让我们将一个测试用例的类 (`test`) 作为顶层的对象而不是一个验证环境的类 (`env`)。如图 10-4 所示，`test_MAC` 是我们的测试用例，它是一个类，其中包含如下四个成员。

- 1) 一个 `sequence` (`MAC_sequence`)，其可以生成一系列事务交易；在这个例子中，事务交易通过 `sequence` 生成，是一个实现了乘累加算法的激励序列。
- 2) 一个验证环境 (`t_env`)，其例化包含了各种验证组件。
- 3) Factory 的重写，可以为 MAC 的测试动态地创建一个验证环境。

4) 配置信息，可以为 MAC 的测试动态地配置验证环境。

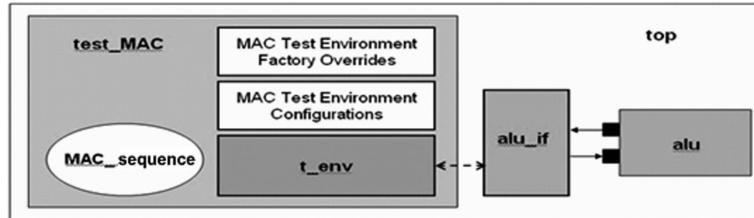


图 10-4 测试作为验证的顶层

相对于标准的验证平台，测试作为验证的顶层在添加和修改测试上提供了很大的灵活性。每个测试用例可以定义它自己特定的配置信息，在编译完所有的测试和验证组件之后，因为每个测试在它运行的时候可以动态地创建和配置验证架构，所以每个测试可以不用重新编译就能运行。

采用了这种方法，上述例子可以根据不同的应用被配置到特定的测试中，包括选择特定的记分板，给测试指定一个合适的衡量机制，选择一个特殊的事务处理器或者配置一个可预测的结果。在某些情况下，整个层次化的模块可能被代替，例如激励生成模块、分析模块和监视模块。从而，同一个验证环境 (test_env) 能够被不同的测试 (test) 多次重用、动态创建和配置。

10.3.4 激励产生与验证架构分离

就如我们前面所说的激励产生、事务交易在验证平台中的验证组件——激励产生器中创建生成；如图 10-5 所示，把这些激励送到 DUT 中需要事务处理器，也就是需要把一个抽象层次的数据转换为另一个层次的数据格式。在我们的例子中 driver 是一个事务处理器，可以接受 ALU 的事务交易，例如加、减操作，将其分解送入到 ALU 的管脚级的端口中。在 DUT 和事务处理器之间通过虚接口 (virtual interface) 来实现。

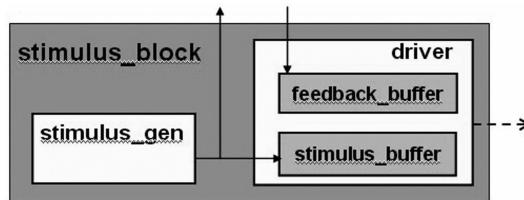


图 10-5 标准的激励产生模块

激励的生成算法被嵌入在激励生成器的类中：stimulus_gen，这种接口限制了修改测试的灵活性。为了添加或者修改测试，产生器的对象需要被另外一个产生器代替，从而重新配置和重新编译是需要的。

除了支持上述方法，OVM 还推荐了另外一种方法：把激励生成的算法模块从验证平台的结构中分离出来，从而在添加和修改测试上提供了更大的灵活性。

在图 10-6 中，生成事务交易的算法包含在一个 sequence 对象中：MAC_sequence，这不是一个结构化的验证组件，而是存在于验证架构以外。OVM 提供了统一的方法来封装激

励: sequence_item, 通过 sequencer 这个仲裁器, 不同的 sequence 能和对应的 sequencer 进行对话从而接入 transactor 中。本例中有一个 sequence (MAC_sequence)、一个 sequencer 和一个事务交易器 driver。sequencer 同步了 MAC_sequence 和 driver 之间的通信。

OVM 不仅仅是一个类库, 如图 10-7 所示, 它还为验证工程师在开发高级验证平台上提供了系统、开放的方法学。这个业界领先的方法学让使用者在更高的层次实现可重用和灵活性。此外, OVM 提供了良好的互用性机制, 使得基于 OVM 开发的验证平台和 VIP 可以在多个验证平台上运行。

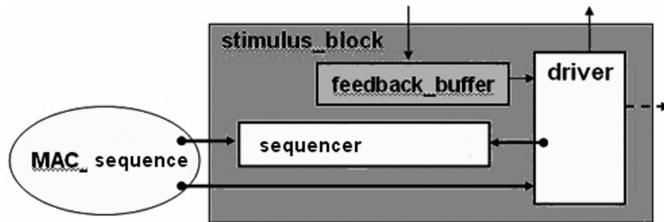


图 10-6 层次化的激励产生模块

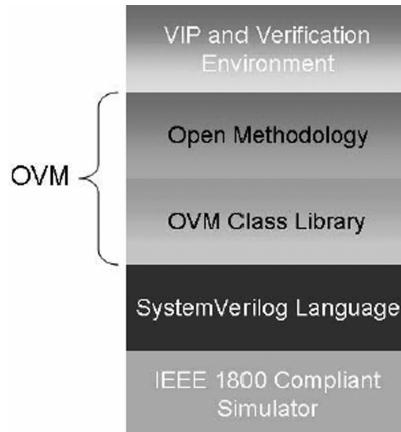


图 10-7 OVM 的验证层次

SystemVerilog 与 C 语言的接口

本章我们简要介绍 SystemVerilog 中一个强大的外部语言接口——DPI。DPI 使得混合使用 C 和 SystemVerilog 更加方便和快捷。下面我们将从三个方面介绍 DPI：DPI 的发展历史及其在工程应用中的地位，如何采用 DPI 实现 SystemVerilog 和 C 程序之间的相互调用和通信，SystemVerilog 和 C 之间的数据类型映射。

11.1 什么是 DPI

什么是 DPI？DPI 就是 Direct Programming Interface（直接编程接口）的缩写。在详细介绍 DPI 之前，我们先介绍一下 Verilog 中 PLI（Programming Language Interface）的发展历史。

Verilog 的 PLI 为 Verilog 提供了一个可以调用 C 程序的机制。倍受关注的 Verilog PLI 是 Verilog 语言被业界广泛采用的重要原因之一。通过使用 PLI，第三方和终端用户可以扩展商用 Verilog 仿真器的功能。实际上，每个使用 Verilog 作为设计语言的大项目都会使用到 Verilog 的 PLI。在过去的二十几年中，Verilog 的 PLI 发布了两个版本：PLI 1.0 和 PLI 2.0（官方称为 PLI – VPI）。Accellera 组织在将 SystemVerilog 作为 Verilog 的扩展写入 IEEE 1800 – 2005 时，引入了一个新的对 C/C++ 调用的接口。这个接口就是 DPI。

通过 DPI，SystemVerilog 的任务和函数可以调用 C 函数，反之亦然。在 SystemVerilog 中调用一个 C 函数，就像调用一个 SystemVerilog 的任务或函数一样简便；同样，在 C 中调用 SystemVerilog 的任务和函数，和调用其他 C 的函数没有太多区别，如图 11-1 所示。为此，我们可以不用去关心和学习相当繁琐的 PLI 或者 VPI。

<u>SystemVerilog</u>	<u>C</u>
<pre> ... if(p==1'b1) t=hello_sv(); ... </pre>	<pre> void hello_sv(){ printf("Hello from C/n"); } </pre>

图 11-1 DPI 调用的简单例子

从理论上说，DPI 不仅是 SystemVerilog 和 C 的接口方式，它也是 SystemVerilog 和其他编程语言的一个可选接口。目前，DPI 只描述了 SystemVerilog 和 C 接口的一些语法结构。

和 PLI 相比，DPI 使得对 C 函数的调用更加简便。使用 PLI，用户必须定义一个系统函数或者任务。用户自定义的系统函数或任务将会以 \$ 作为名字的开头。例如一个 sine 函数在 Verilog 中可能会被表示为 \$sine，这个函数和一个用户提供的 C 函数（calltf routine）产生关联；这个 calltf 子程序可以调用 C 算术库（math library）中的 sine 函数。calltf 函数和它调用的其他函数，都必须编译和链接到 Verilog 仿真器中。一旦这些步骤完成之后，\$sine 系统函数可以在 Verilog 中如同普通函数一样使用。在仿真执行的过程中，它会调用 C 的 calltf 函数，接着调用 C 的 sin 函数。

```
always @ (posedge clock) begin
    slope <= $sine(angle); // 调用 sine 函数
end
```

从 Verilog 的角度看，这似乎没有太多区别，其实最大的区别在于它在定义 PLI 的用户自定义系统函数和任务的过程中。而 DPI 无须如此繁琐，只要通过简单的语法就可以直接实现调用。

DPI 的出现并不意味着就可以取代 PLI（或者 VPI），它们之间应该是互补的关系。其中，有下列两个主要原因可以解释为什么 PLI 和 VPI 将仍然存在和发挥其重要作用。

- 1) PLI 和 VPI 的时间测试方法可以保护仿真器的数据库。PLI 和 VPI 将继续提供这样一个保护机制来访问设计数据，同时保护仿真器数据库的完整性。

- 2) 对于很多公司，PLI 仍是未来数据接口一种可行的选择。有很多应用程序都是通过 PLI 和 VPI 来描述的，这些遗留下来的应用程序仍会被维护并且更新，从而可以在新的项目中使用。

另外，最重要的一个因素是 DPI 不能直接访问仿真数据的内部，而 PLI 可以全面地访问内部仿真数据结构，特别在波形显示、图像化调试等其他需要访问和分析仿真内部数据结构的时候，PLI 是唯一一个过程化的接口。

11.2 DPI 的应用

DPI 作为一个接口，它连接着两个层面（layer）：SystemVerilog 层面和 C 层面。就如前面提到的，理论上 SystemVerilog 可以通过 DPI 接口调用任意的外部语言，但是在目前，只有 C 接口的语法被定义。

接口的两个层面保持各自的独立性。例如数据如何在接收端解释处理和发送端没有关系，一个层面内部对数据的解析和数据的内部转换是和另外一个层次也不相关的。

下面我们将讨论在采用 DPI 接口的过程中，如何在 SystemVerilog 层面中对函数或任务进行的两种基本操作：导入（import）和导出（export）。

基本上，我们可以从 SystemVerilog 这个层面（作为主端）去看 DPI 接口，导入就是 SystemVerilog 通过 DPI 引入一个外部程序；导出就是 SystemVerilog 通过 DPI 向外输送一个 SystemVerilog 的程序。无论导入还是导出，都是在 SystemVerilog 这个层面进行定义的。

11.2.1 方法的导入

当 SystemVerilog 代码需要调用一个外部的 C 函数时，该 C 函数就被称为导入函数（im-

ported function)。类似，若是一个 C 的任务被 SystemVerilog 调用，就称为导入任务（imported task）。在 SystemVerilog 中，我们把任务和函数统称为方法（method），为此我们也统称为导入方法（imported method）。

很重要的一点是，任何 C 函数都可以被引入到 SystemVerilog 中，包括标准的 C 的函数库，例如 malloc()、free()、strlen()。DPI 使用户可以采用强大的 C 库，而无需额外的投入。

从 SystemVerilog 层调用一个外部的 C 程序需要如下三个步骤。

- 1) 在 SystemVerilog 中采用关键字 import，定义一个导入方法（imported method）。
- 2) 被调用 C 函数或者任务的实现。
- 3) 在 SystemVerilog 代码中调用该方法。

下面，我们从相反的顺序讨论这个过程。

在 SystemVerilog 中调用一个导入方法，就如同调用 SystemVerilog 的本地方法一样。如下所示，我们无法从调用代码上区分 my_function 和 my_funtion2 是采用 SystemVerilog 或者 C 实现的。

```
integer t1, t2;
logic [31:0] q;
...
if (p == 1'b1) begin
    t1 = my_function();
    t2 = my_function2(q);
end
```

C 程序的实现，就如前面提到的，SystemVerilog 层和 C 层的实现是各自独立的。后面我们会重点讨论 SystemVerilog 数据类型和 C 的数据类型如何映射的问题。需要注意的是，导入方法也可能产生时间延迟，会潜在导致某些行为在时序上的不一致，另外需要注意的是迭代函数。

方法导入的定义，就是在 SystemVerilog 层定义了一个完整的被引入方法的对象，是 SystemVerilog 层最重要的部分。

在 SystemVerilog 中调用的每个外部程序，都要通过 import 这个关键字来定义其导入，说明如下。

1) 外部函数导入的定义

```
import "DPI-C" [context |pure] [c_name = ] function data_type fname([params]);
```

2) 外部任务导入的定义

```
import "DPI-C" [context] [c_name = ] task tname([params]);
```

其中，context 表示被调用的 C 函数可能会访问其接口参数以外的 SystemVerilog 内部对象，若 C 函数只是处理其接口参数，而对 SystemVerilog 其他对象没有任何操作的话，可以采用 pure 来定义。

如前面的例子所示，一个输入函数 my_funtion 需要下面一个类似的定义：

```
import "DPI-C" function integer my_function();
```

DPI 对导入函数的返回值有特殊的限制，说明如下。

- 返回值的数据类型只能是 void、byte、shortint、int、longint、real、shortreal、chandle 和 string。
- 32 位的压缩向量或者等价类型。
- bit 和 logic 标量。

源代码 11-1 和源代码 11-2 所示是一个简单的应用实例，通过 DPI 接口调用一个用 C 写的 Hello 程序。

源代码 11-1 外部 Hello 程序 1（基于 C）

```
//Chapter11 example1/hello_c.c:
#include "dpiheader.h"
int c_Hello()
{
    printf("Hello from c_Hello()\n");
    return (0);/* 结束返回 */
}
```

源代码 11-2 DPI 导入方法实例

```
//Chapter11 example1/hello.v:
module hello_top;
    int ret;
    import "DPI-C" context task c_Hello();
    initial
    begin
        c_Hello(1, ret); // 调用 c 任务 c_Hello()
    end
endmodule
```

11.2.2 方法的导出

当我们需要从 SystemVerilog 层面将一个 SystemVerilog 内部定义的方法导出，以便在外部 C 程序中可以调用时，我们需要采用关键字 export 对每个输出的方法做导出定义。其语法如下：

函数导出的定义：

```
export "DPI-C" [c_name = ] function data_type fname([params]);
```

任务导出的定义：

```
export "DPI-C" [c_name = ] task tname([params]);
```

所有的导出方法都具有 context 的属性，也就是 C 程序通过这个接口需要访问到 SystemVerilog 中与例化相关的一些数据。SystemVerilog 层面输出的任务不可以被 SystemVerilog 层面作为导出函数的 C 程序调用。导出函数或者任务的输出参数（output）会通过 C 的指针的方式来访问。

源代码 11-3 和源代码 11-4 所示是一个采用 DPI 的导出 SystemVerilog 函数的例子。

源代码 11-3 外部 Hello 程序 2（基于 C）

```
//Chapter11 example2/hello_c.c:
#include "svdpi.h"
#include "dipiheader.h"
int c_Hello(int i, int * o)
{
    printf("Hello from c_Hello()\n");
    verilog_task(i, o); /* 回调 Verilog 任务 */
    * o = i;
    return (0); /* 结束返回 */
}
```

源代码 11-4 DPI 导出方法实例

```
//Chapter11 example2/hello.v:
module hello_top;
int ret;
export "DPI-C" task verilog_task;
task verilog_task(input int i, output int o);
#10;
$display("Hello from verilog_task()");
endtask
import "DPI-C" context task c_Hello(input int i, output int o);
initial
begin
c_Hello(1, ret); // 调用 c 任务 c_Hello()
$display("ret is %d", ret);
end
endmodule
```

11.2.3 DPI 的数据类型映射

通过 DPI，SystemVerilog 需要和 C 做数据上的交互，表 11-1 是 SystemVerilog 数据类型和对于 C 类型的一个映射表。

表 11-1 数据类型映射

SystemVerilog Type	C Type
byte	char
shortint	short int
int	int
longint	long long
real	double
shortreal	float
chandle	void *
string	const char *
bit	unsigned char
logic/reg	unsigned char

具体使用 DPI 的过程中，还需参考仿真器的实现方式和推荐的代码风格。在参数类型上，SystemVerilog 具体通过 DPI 接口是如何将数据传递到 C 程序的？图 11-2 是一个简单的规则，供大家参考。

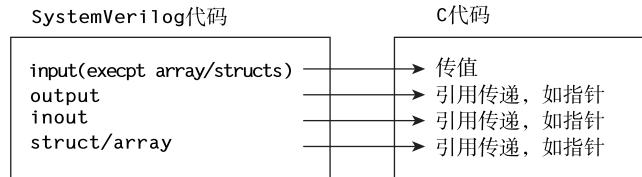


图 11-2 参数传递规则

11.2.4 DPI 的具体应用

前面我们提到，DPI 相对 PLI 在语法和使用上比较简练。采用 DPI 还有其他优点，例如同样 C 程序采用 DPI 要比 PLI 调用运行速度要快，而且容易调试。DPI 还能给验证工作提供哪些好处呢？这就涉及一个最根本的问题：为什么验证平台中需要用到外部程序？

在验证平台中集成外部程序，如 C/C++ 程序，其重要的目的在于重用一些现成的、稳定的模型和机制，例如 SDH 通信芯片可以通过外部 C/C++ 程序产生数据激励；经验证过的物理层编解码模块的 C/C++ 算法模型，可以作为 RTL 验证平台中的参考模型，提供期望的数据结果。在一个项目开始的时候，很多模块会通过 C/C++ 去搭建其原型，分析性能和功能的正确性。为此，在设计的硬件实现（RTL 实现）完成之后，若可以借用系统分析时完成的、稳定健壮的 C/C++ 代码提高验证效率，这也是验证重用的一个重要思想。

当然，在某些特殊情况下，比如大量的文本处理、数据比较等操作，利用 C/C++ 程序也是很容易而且快速实现的，要比在 SystemVerilog 中采用类似的程序更具有通用性和更高的效率。

DPI 接口为 SystemVerilog 调用提供了一个便捷的方法。图 11-3 就是一个通过 DPI 采用 C 实现数据激励和参考模型的例子。

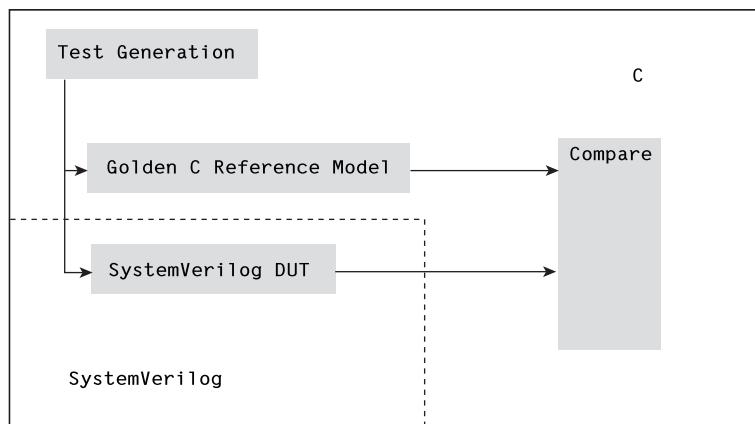


图 11-3 DPI 的典型应用

SystemVerilog | 附录 A

覆盖率内置参数和方法列表

表 A-1 基于特定例化覆盖组的内置参数

选项名称	默认值	描述
weight = number	1	设置每个覆盖组的权重，非负数
goal = number	100	指定目标覆盖率
name = string	unique name	指定名字
comment = string	""	说明
at_least = number	1	每个分组柜至少击中几次才算被覆盖过
detect_overlap = boolean	0	若被设置为 1，分组柜之间若出现数值域的重叠，会发出告示
auto_bin_max = number	64	默认情况下覆盖点最多可以创建分组柜的数目
cross_num_print_missing = number	0	交叉覆盖点中没有被划入特定交叉分组柜的情况是否被报告
per_instance = boolean	0	是否每个例化都对整体的覆盖率有影响
get_inst_coverage = boolean	0	只有 merge_instance 这个类型选项被设置时才有用，使能该选项后，每个例化对应的 get_instance_coverage 返回自身的数据；默认情况下，其返回数值等于 get_coverage

使用方法：option.option_name = expression；每个覆盖组的例化都可以修改并初始化其内置的这些参数选项。

表 A-2 基于特定例化覆盖组内置参数的应用层次

选项名称	可应用的语法层次		
	covergroup	coverpoint	cross
name	Yes	No	No
weight	Yes	Yes	Yes
goal	Yes	Yes	Yes
comment	Yes	Yes	Yes
at_least	Yes (default for coverpoints & crosses)	Yes	Yes
detect_overlap	Yes (default for coverpoints)	Yes	No
auto_bin_max	Yes (default for coverpoints)	Yes	No
cross_num_print_missing	Yes (default for crosses)	No	Yes
per_instance	Yes	No	No

表 A-3 基于覆盖组类型的内置参数

选项名称	默认值	描述
weight = constant_number	1	如果设置在覆盖率组这个层次，表示该覆盖率组在全部统计中的权重，若设置在覆盖率点，则表示该覆盖率点在覆盖率组内的权重，非负数
goal = constant_number	100	指定某个覆盖组类型中，整体的目标或者某个覆盖点或者交叉覆盖点的目标值
comment = string_literal	""	说明
strobe = boolean	0	若被设置为真时，在时刻间隙的最后点发生采样，如同 \$strobe 系统函数
merge_instances = boolean	0	若被设置为真，该类型的累计覆盖率会将所有的例化覆盖率做合并后才计算。若为假，则根据每个例化权重计算

使用方法：type_option. option_name = expression；上述参数就如类中的静态数据成员，为所有例化所共有。

表 A-4 基于类型内置参数的应用层次

参数选项	可应用的层次		
	covergroup	coverpoint	cross
weight	Yes	Yes	Yes
goal	Yes	Yes	Yes
comment	Yes	Yes	Yes
strobe	Yes	No	No

表 A-5 预定义覆盖率的内置方法

方法	可被调用层次			描述
	covergroup	coverpoint	cross	
void sample ()	Yes	No	No	触发覆盖组采样
real get_coverage () real get_coverage (ref int, ref int)	Yes	Yes	Yes	计算覆盖组的覆盖率
real get_inst_coverage () real get_inst_coverage (ref int, ref int)	Yes	Yes	Yes	计算例化覆盖率
void set_inst_name (string)	Yes	No	No	设置例化名字
void start ()	Yes	Yes	Yes	开始统计覆盖率信息
void stop ()	Yes	Yes	Yes	停止统计覆盖率信息

SystemVerilog | 附录 B

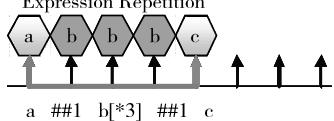
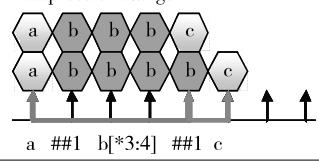
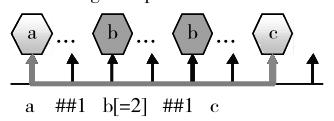
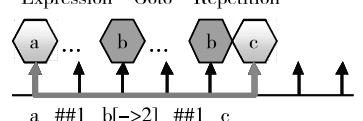
断言重复操作符和序列操作符列表

```
sequence sequence_identifier [( [ sequence_port_list ] )] ;
{assertion_variable_declaration}
sequence_expr ;
endsequence [ : sequence_identifier ]

sequence_expr ::= =
  cycle_delay_range sequence_expr {cycle_delay_range sequence_expr}
  | sequence_expr cycle_delay_range sequence_expr {cycle_delay_range sequence_expr}
  | expression_or_dist [boolean_abbrev]
  | (expression_or_dist {, sequence_match_item}) [boolean_abbrev]
  | sequence_instance [sequence_abbrev]
  | (sequence_expr {, sequence_match_item}) [sequence_abbrev]
  | sequence_expr and sequence_expr
  | sequence_expr intersect sequence_expr
  | sequence_expr or sequence_expr
  | first_match(sequence_expr {, sequence_match_item})
  | expression_or_dist throughout sequence_expr
  | sequence_expr within sequence_expr
  | clocking_event sequence_expr

property property_identifier [( [ property_port_list ] )] ;
{assertion_variable_declaration}
property_spec ;
endproperty [ : property_identifier ]
property_expr ::= =
  sequence_expr
  | (property_expr)
  | not property_expr
  | property_expr or property_expr
  | property_expr and property_expr
  | sequence_expr | -> property_expr
  | sequence_expr |= > property_expr
  | if(expression_or_dist) property_expr [else property_expr]
  | property_instance
  | clocking_event property_expr
```

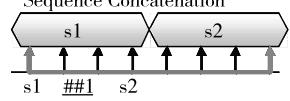
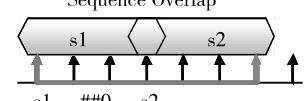
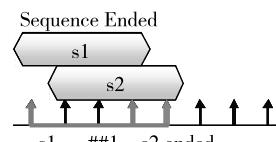
表 B-1 重复操作符列表

操作符	描述	例子	图示
$[^n]$	连续重复 n 次	a [*2] a 在连续两个周期内为真	Expression Repetition 
$[m:n]$	连续重复 m 到 n 次（最少 m 次，最多 n 次）	c[*2:5] c 在连续 2 到 5 个周期内为真 (连续 2/3/4/5 次都满足条件)	Expression Range 
$[=n]$	非连续重复操作	data [=8] data 连续 8 次被采样到为真，其中每次之间无须是相邻周期	Expression Non-Consecutive “Counting” Repetition 
$[>n]$	跟随重复操作	data [->4] data 连续被采样到 4 次为真，其中后续表达式在第四次采样后马上生效	Expression “Goto” Repetition 

注意： $[>n]$ 和 $[=n]$ 只有给操作符后面跟其他表达式才可以看出两者的区别。满足 $[>]$ 必定满足 $[=n]$ ，也就是 $[>n]$ 成立的条件更加严格；而满足 $[^n]$ 必定满足 $[>n]$ 和 $[=n]$

序列操作符：序列之间除了可以使用##n、##[m:n] 和表 B-1 中提到的重复操作之外，还有其他的操作。

表 B-2 序列操作符

操作符	图示	描述
$s1 \underline{\#1} s2$	Sequence Concatenation 	序列 “s1” 结束后的下一个周期就是序列 “s2”的开始
$s1 \underline{\#0} s2$	Sequence Overlap 	序列 “s1” 结束点和序列 “s2” 的开始点在同一个时钟周期
$s1 \underline{\#1} \underline{s2. ended}$	Sequence Ended 	序列 “s2”的结束点必须出现在序列 “s1”的结束点的后一个周期。换句话说，序列 “s1” 结束后的下一个周期序列 “s2” 也结束

(续)

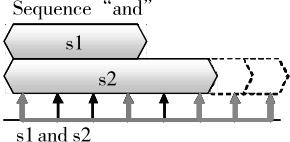
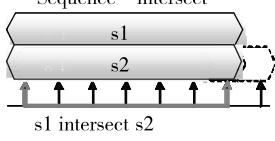
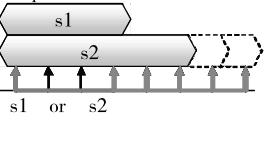
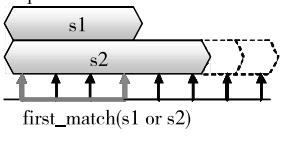
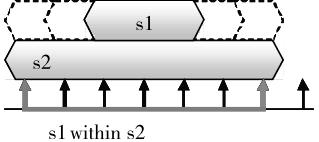
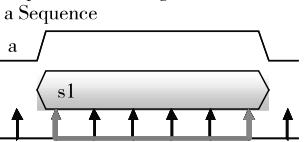
操作符	图示	描述
and	Sequence “and” 	序列“s1”和“s2”在同一个时刻开始，在其中一个序列成立之后，等待另外一个序列成立。一旦两个序列都成立，该操作的结果判定为真
intersect	Sequence “intersect” 	intersect 是 and 的一个特例。它要求两个序列同时开始，同时结束
or	Sequence “or” 	只要序列“s1”或“s2”其中一个成立，该操作的结果判定为真
first_match	Sequence First Match 	只要“s1”或者“s2”中第一序列被判定为真，则该操作为真。下列窗口延迟，可能每一个可能出现都可以被判定为真： a ##[2:9] b 然而，first_match (a ##[2:9] b) 只有第一符合要求序列出现的时候被判定为真，其他作不判定
within	Sequence “within” another Sequence 	序列“s1”在序列“s2”发生的期间内出现。可以看成是： (1 [*0:\$] ##1 s1 ##1 1 [*0:\$]) intersect s2
throughout	Expression “throughout” a Sequence 	表达式‘a’必须在序列“s1”发生期间保持为真。可以看成是： (a) [*0:\$] intersect s1

表 B-3 内置采样方法

方法	描述
\$rose (expr [, clk])	指定表达式从低到高发生跳转
\$fell (expr [, clk])	指定表达式从高到低发生跳转
\$past (expr1 [, num_ticks] [, expr2] [, clk])	表达式1在第N个时钟周期前的值；表达式2是保护条件
\$stable (expr [, clk])	上个时钟周期表达式的值没有变化
\$sampled (expr [, clk])	上个时钟周期表达式的值

附录 C | SystemVerilog

QuestaSim 简要介绍

目前，半导体行业有三大 EDA 公司，分别是 SYNOPSYS、Mentor Graphics 和 Cadence。这三家公司分别提供了 VCS、QuestaSim 和 NC_Sim 仿真软件，分别占领 1/3 的市场份额。下面我们将简要的介绍 QuestaSim 的使用流程。

1. QuestaSim Verification Platform

QuestaSim 是 Mentor Graphics 公司提供的全面功能验证解决方案，也是业界唯一能够支持所有验证流程并提高质量和生产力的数字电路功能验证集成化平台。

作为完整的可扩展验证解决方案的一部分，Questa 是第一个基于标准的单内核的验证平台，它统一的用户界面集成了 HDL 仿真器（modelsim）、约束解算器（constraint solver）、断言（assertion）引擎和功能覆盖率统计。该平台内置高级的验证方法，如测试平台自动化（Testbench Automation, TBA）、覆盖率驱动验证（Coverage-driven Verification, CDV）、基于断言的验证（Assertion-based Verification, ABV）和事务级建模（Transaction-level Modeling, TLM），是用户验证复杂 ASIC/FPGA 的最佳选择。

QuestaSim 有以下主要特性。

- 内建单内核仿真器支持 SystemVerilog、SystemC、PSL、VHDL、Verilog。
- 内建约束解释器支持 Constrained – random 激励生成，以实现 Testbench – Automation。
- 支持基于 PSL、SystemVerilog 语言断言的功能验证，支持业界 OVL 和 QVL 断言库功能验证。
- 集成化支持功能覆盖率检查与分析。
- 高性能的 RTL 和 Gate-level 仿真速度。
- 支持用 SystemVerilog 和 SystemC 实现高层次 testbench 设计与调试。
- 高性能集成化的混合语言调试环境加速对混合验证语言（SystemVerilog、SystemC、PSL、VHDL、Verilog）的交叉调试与分析。
- 基于标准的解决方案能支持所有的流程，便于保证验证上的投入。
- 提供最高性价比的功能验证解决方案。

QuestaSim 的用户界面如图 C-1 所示。

2. QuestaSim 的使用基本流程

用户安装好 QuestaSim 之后，进入安装目录下的 examples/tutorials/verilog/basicSimulation 目录（在 QuestaSim 提示符下键入 cd ...examples/tutorials/verilog/basicSimulation）。

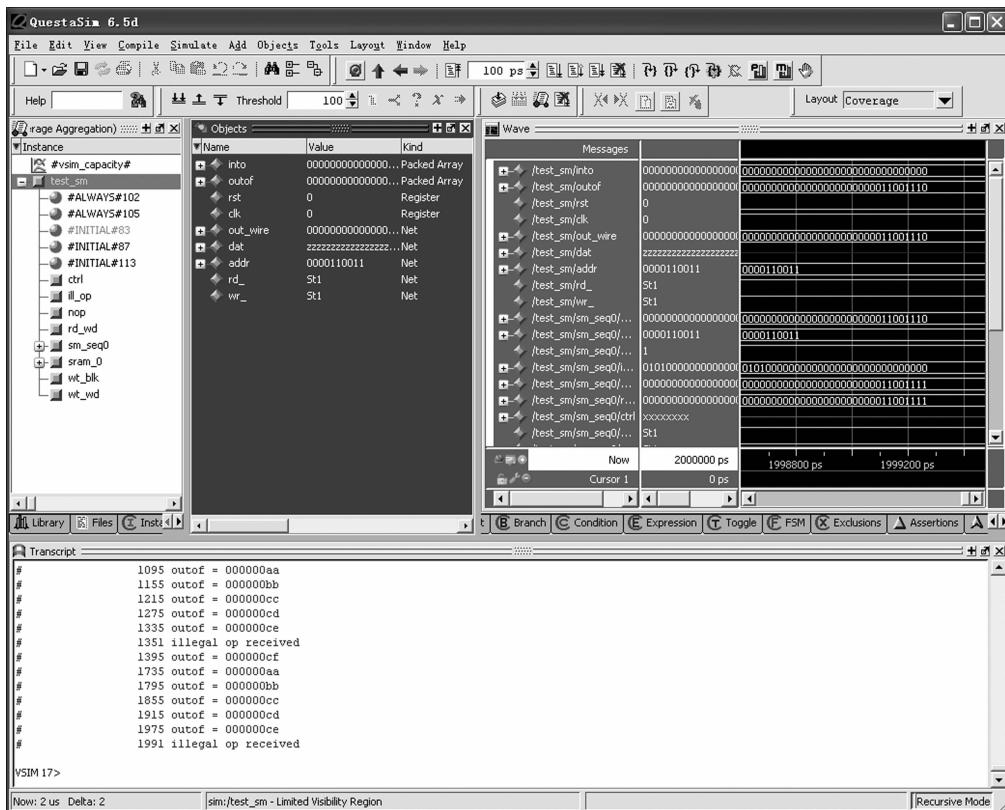


图 C-1 QuestaSim 用户界面

第一步：创建库。

```
vlib work
```

第二步：编译代码（Transcript 窗口如图 C-2 所示）。

```
vlog -novopt counter.v tcounter.v
```

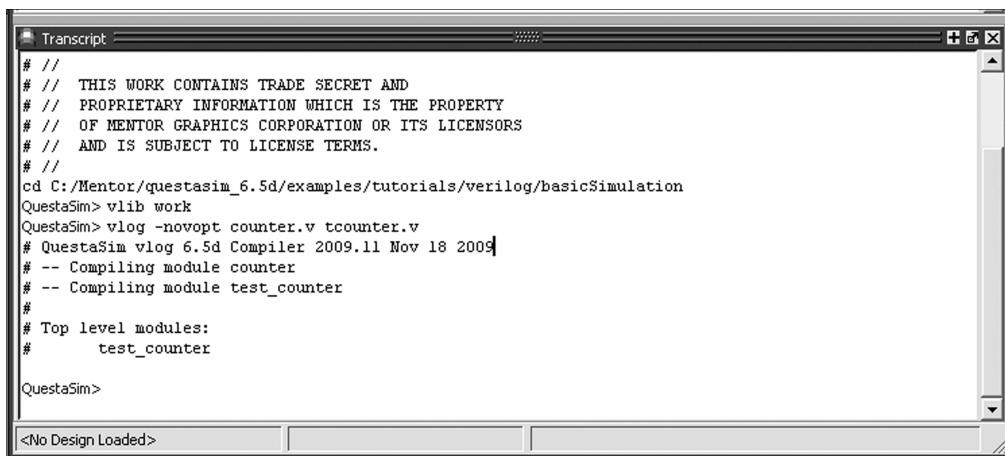


图 C-2 Transcript 窗口

第三步：调用仿真。

```
vism test_counter -novopt
```

第四步：添加波形。

```
add wave -r /*
```

第五步：仿真运行（如图 C-3 所示）。

```
run 1000 ns
```

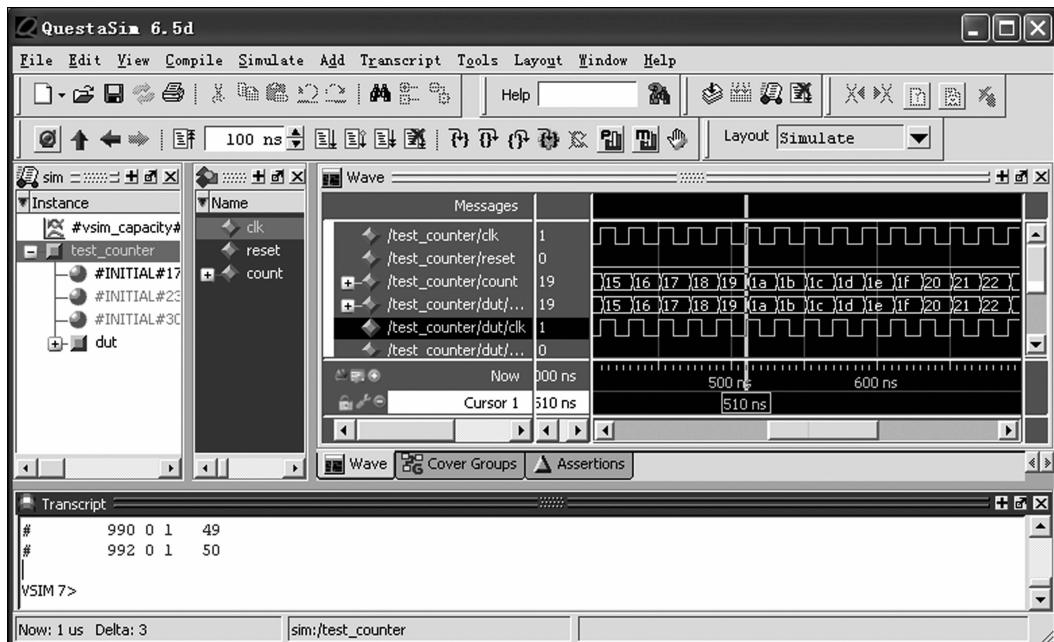


图 C-3 Wave 运行波形窗口

第六步：退出仿真。

```
quit-sim
```

对于 QuestaSim 的基本使用流程，请参考安装目录下的 docs/pdfdoc/questa_afv_tut.pdf 文档 (Questa SV/AFV Tutorial)。用户使用该文档并配合 examples 下的程序实例，可以基本熟悉 QuestaSim 的使用流程。请入门学习使用 QuestaSim 或者 Modelsim 的读者务必认真通读该份文档 (Questa SV/AFV Tutorial)。

另外两份重要的参考文档：questa_afv_user.pdf (Questa SV/AFV User's Manual) 是 QuestaSim 各种功能的用户使用手册，对于特定功能的实现请用户在需要的时候参考对应章节，以了解详细的使用流程和注意事项；questa_afv_ref.pdf (Questa SV/AFV Reference Manual) 是 QuestaSim 的命令参考手册，用户可以在此查阅所有用户命令的详细说明和使用方法。

3. GUI 操作的基本流程

首先，改变访问目录到安装目录下的 examples/tutorials/verilog/basicSimulation，操作如

下：单击 File -> Change Directory，如图 C-4 所示。在弹出的对话框中，可以选择对应的文件目录并单击“确定”按钮，如图 C-5 所示。

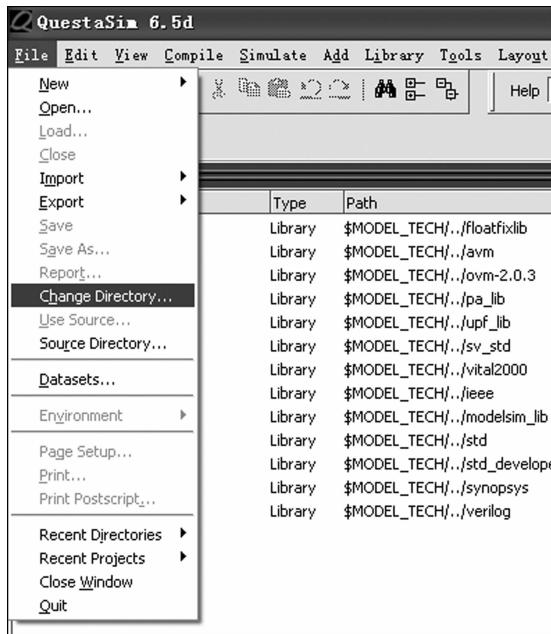


图 C-4 File 菜单



图 C-5 文件夹对话框

步骤一：创建一个工作库。首先，单击 File -> New -> Library，选中 Library，会出现一个对话框。在对话框中，可以看到有 Library Name，这是库的逻辑名，有 Library Physical Name 这是库的文件夹名字（物理存在的）。默认情况下，两者均为 work，我们称之为 work 库。如图 C-6 所示，单击 OK 按钮，就完成了库的创建。

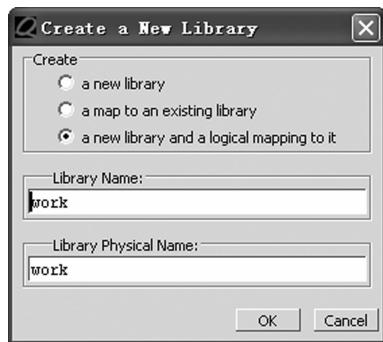


图 C-6 创建新库对话框

步骤二：编译源文件。单击下拉菜单 Compile -> Compile，就会出现一个对话框。选中 BasicSimualtion 下面的两个源文件 counter. v 和 tcounter. v；单击 Compile 按钮，如图 C-7 所示。

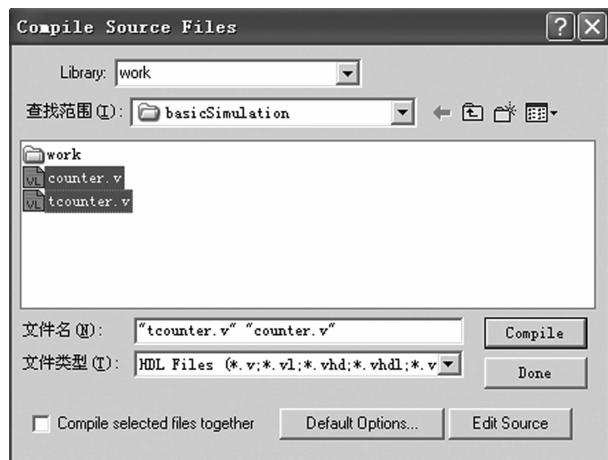


图 C-7 编译文件对话框

可以看到 Transcript 窗口中输出编译日志，最后单击 done 按钮结束本次编译。

```
vlog      - reportprogress      300      - work      work
C:/Mentor/questasim_6.5d/examples/tutorials/verilog/basicSimulation/tcounter. v
# QuestaSim vlog 6.5d Compiler 2009.11 Nov 18 2009
# -- Compiling module test_counter
#
# Top level modules:
# test_counter
vlog      - reportprogress      300      - work      work
C:/Mentor/questasim_6.5d/examples/tutorials/verilog/basicSimulation/counter. v
# QuestaSim vlog 6.5d Compiler 2009.11 Nov 18 2009
# -- Compiling module counter
#
# Top level modules:
# counter
```

步骤三：调用仿真。在 library 窗口中，单击 work 库展开其内容，选中 test_counter，单击右键，出现一个快捷菜单；单击 Simulation without Optimization，如图 C-8 所示。

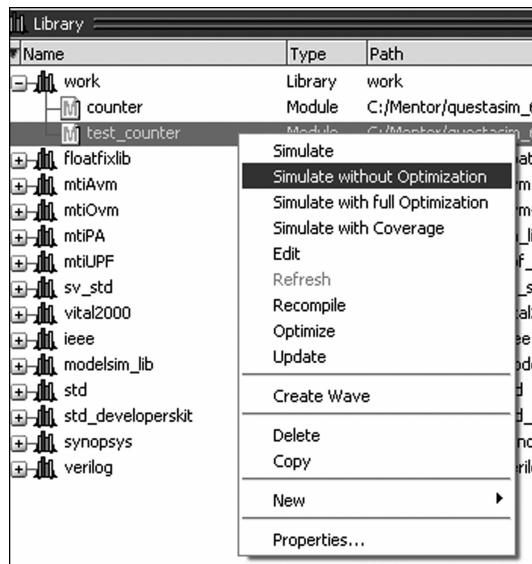


图 C-8 库窗口

此时，Transcript 窗口会打印本次仿真加载的信息，并进入 vsim 的提示符下。

```
vsim -novopt work.test_counter
# vsim -novopt work.test_counter
# Refreshing C:\Mentor\questasim_6.5d\examples\tutorials\verilog\basicSimulation\
work.test_counter
# Loading work.test_counter
# Refreshing C:\Mentor\questasim_6.5d\examples\tutorials\verilog\basicSimulation\
work.counter
# Loading work.counter
```

步骤四：添加信号到波形窗口。首先，单击菜单 View->Objects 调用 Objects 窗口，单击 View->Wave 调用波形窗口。之后在 Instance 窗口中选中 dut（counter 的例化名），Objects 窗口会出现该例化层次下面的所有信号。选中所有信号并拖曳到波形窗口中。或者选中后单击右键，选择 Add->To Wave->Selected Signals 把信号添加到波形窗口中，如图 C-9 所示。

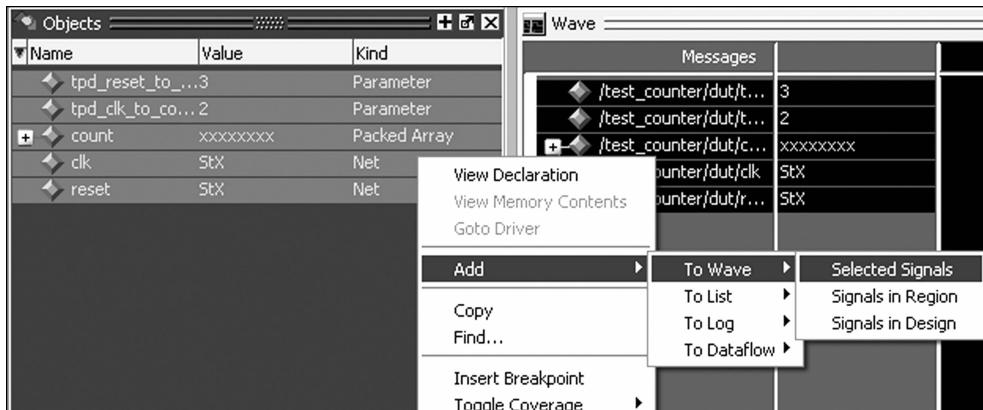


图 C-9 从 Object 窗口添加信号到 Waveform 窗口

步骤五：仿真运行。单击菜单 Simulate -> Run -> Run 100，如图 C-10 所示，默认运行 100ns，也可以通过  来运行 Run 命令。

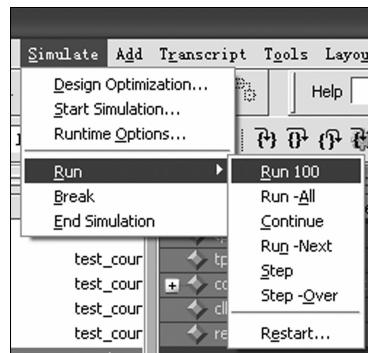


图 C-10 Simulate 菜单

步骤六：结束本次仿真，单击菜单 File -> Quit 或者直接关闭 QuestaSim。

SystemVerilog | 附录 D

常用术语中英文对照

testbench	验证平台或测试平台
testcase	测试用例
directed test	直接测试
random test	随机测试
stimulus	激励, 测试激励
transactor	事务处理器
transaction	事务交易
BFM	总线功能模型
verification component	验证组件
scoreboard	记分板
verification plan	验证计划
black-box verification	黑盒验证
white-box verification	白盒验证
formal verification	形式验证
simulation based verification	仿真验证
hardware assisted verification	硬件辅助验证
hardware acceleration and emulation	硬件加速和模拟验证
event based simulation	基于事件的仿真
cycle based simulation	基于时钟周期的仿真
functional verification methodology	功能验证方法学
Assertion Based Verification (ABV)	基于断言的验证技术
Coverage Driven Verification (CDV)	基于覆盖率驱动的验证技术
Constrained Random Verification (CRV)	约束随机验证
reuse	重用
transaction-level interface	事务层接口
Transaction-Level Modeling (TLM)	事务级建模
dynamic array	动态数组
associative array	关联数组
queue	队列
packed array	压缩数组

unpacked array	非压缩数组
Object Oriented Programming (OOP)	面向对象编程
class	类
property	属性（类中的变量或者断言的一种封装结构）
method	方法（包括函数和任务）
object	对象，类的一个例化
handle	句柄，对象的入口地址
shallow copy	浅复制（浅拷贝）
deep copy	深复制（深拷贝）
interface	接口
virtual interface	虚接口
inheritance	继承
parent class	父类
child class	子类
base class	基类
extended class	派生类
override	重写/过载
constructor	构造函数：new()
polymorphism	多态
virtual method	虚方法
functional coverage	功能覆盖率
covergroup	覆盖组
coverpoint	覆盖点
cross	交叉覆盖点
bin	分组柜
assertion	断言
immediate assertion	立即断言
concurrent assertion	并发断言
consecutive repetition	连续重复
non-consecutive repetition	非连续重复
goto repetition	跟随重复
DPI	直接编程接口

参 考 文 献

- [1] Faisal Haque, Khizar Khan, Jonathan Michelson. The Art of Verification with VERA [M]. California: Verification Central, 2001.
- [2] Sasan Iman. Step-by-Step Functional Verification with SystemVerilog and OVM [M]. US: Hansen Brown Publishing, 2008.
- [3] Chris Spear. SystemVerilog for Verification [M]. US : Springer , 2006.
- [4] SystemVerilog 3. 1a 语言参考手册, 2005. www. fpgatech. net.
- [5] IEEE. IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language-IEEE1800-2005 [S]. US: IEEE, 2005.
- [6] IEEE, IEEE P1800/D6 Draft Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language [S]. US: IEEE, 2009.
- [7] Janick Bergeron. Writing Testbenches Using SystemVerilog [M]. US: Springer, 2006.
- [8] Ben Cohen. SystemVerilog Assertions Handbook [M]. US: vhdlcohen Publishing, 2005.
- [9] Mikhail Noumerov, Lyubov Zhivova. Using SystemVerilog for IC Verification [M]. SNUG, 2005 .
- [10] Stuart Sutherland. The Verilog PLI is Dead (maybe)—Long Live of the SystemVerilog DPI! [M]. SNUG, 2004.
- [11] Willamette HDL. SystemVerilog for Verification Training [M]. US: Willamette HDL, 2008.
- [12] Willamette HDL. Open Verification Methodology Training [M]. US: Willamette HDL, 2008.
- [13] 钟文枫. 下一代芯片设计与验证语言：SystemVerilog（验证篇） [J]. 电子设计应用, 2008 (12).
- [14] 钟文枫. OVM 实现了可重用的验证平台 [J]. 电子工程专辑, 2009 (3).
- [15] 钟文枫, 耿彦莉. AMBA 片上总线在 SoC 芯片设计中的应用 [J]. 电子设计应用, 2006 (3).
- [16] Mark Glasse. 高级验证方法学 [M]. 钟文枫, 王欣, 等译. 成都: 电子科技大学出版社, 2007.
- [17] 牛风举. 基于 IP 复用的数字 IC 设计技术 [M]. 北京: 电子工业出版社, 2003.
- [18] 袁俊泉. Verilog HDL 数字系统设计及其应用 [M]. 西安: 西安科技大学出版社, 2002.
- [19] Mentor Graphics. Questa 6. 5d SV/AFV Tutorial.
- [20] Mentor Graphics. Questa 6. 5d SV/AFV User Guide.
- [21] David Jones, Rich Edelman. A Reusable SystemVerilog Testbench in Only 300 Lines of Code. US: Mentor Graphics U2U, 2006.

后记

过去的五年，我一直都在从事与功能验证相关的工作，与不少公司有交流合作，深深体会到国内验证技术和国外的差距，也是我一直决心编写这本书的主要原因。在国内开始学习采用 Vera 和 SpecmanE 的时候，国外已经开始将 SystemVerilog 标准化，大规模推广这门硬件描述和验证语言。近年来，随着 EDA 工具对 SystemVerilog 的广泛支持，多数设计公司已经逐步从 Vera 和 SpecmanE 转向 SystemVerilog；国内却只有在大公司才有机会接触和学习 SystemVerilog，原因在于许多高校缺乏合适的教材，而国外的书籍面向的对象都是专业的工程师，如何弥补这当中的鸿沟呢？希望本书给初学者架起一个桥梁，以缩短国内技术普及教育和国外的差距。

国内的半导体行业发展迅速，主要是我国的电子产品销往了全球各地，而对于芯片设计行业而言，却是刚刚起步，还有很大差距。但是，每个行业都有一个积累的过程，需要从技术的点点滴滴做起，就算是国外很成熟的技术，我们仍要保留一份学习的心态，认真分析透彻。就这点而言，山寨也算是成功模仿的典范！

在这里，我简要介绍一下关于功能验证方面的一些概念和准备。

在功能验证方面重要的有几个步骤，其中第一个步骤就是验证计划的制定，也就是解决验证的目标和策略，这通常需要系统工程师还有设计工程师的共同参与。假如一个验证项目组或团队对将要验证的对象没有一点认识和理解的话，就无从制定验证方案。为此，特定的专业知识是必须的，例如通信协议、图像处理算法等，可以根据各自工作的需求学习。

验证计划要回答几个问题，第一个就是验证什么功能，对功能点的描述需要明确；第二个问题是怎们验证，对该功能点的验证方法和策略要清楚，这样才有可能通过具体的手段，例如验证语言、FPGA 原型等去实现；第三个问题是如何才算验证收敛，其中涉及两个方面，每个功能点是否被测试通过的衡量标准，另外就是整个验证工作完成的衡量标准。

只有清楚地回答了这三个问题，验证的方向明确了，才有可能开展后续的具体实施工作。

制定验证计划就如制定设计方案，需要投入大量的前期工作和规划，是必不可少的一个步骤。

第二个步骤就是搭建验证环境，这个验证环境可以是基于验证语言的仿真环境，也可以是其他的，例如 FPGA 的原型验证环境，这里我们指前者。其中需要注意的一点是，无论 SystemVerilog、SystemC 或者 SpecmanE，验证语言只是一种工具，是表达验证计划中验证策略的一种方式。验证语言无法回答设计是否正确的问题。也就是说，是不是采用了 SystemVerilog，验证结果和被验证对象就一定是正确的？不是。只有通过 SystemVerilog 正确描

述了一个正确的测试意图，这种测试结果才算是正确的。

在搭建验证环境的过程中，需要解决的几个问题是：如何实现验证平台的可重用性、提高验证效率和加速验证收敛。

针对这些问题，我们在验证方法学中已经介绍了很多具体的技术，简单总结如下：基于事务交易级的验证，提高激励生成和应用的层次；验证模块封装和动态配置；随机激励生成提高测试激励的覆盖范围，减少测试用例的开发；采用断言快速定位问题，缩短调试周期；采用基于覆盖率驱动的验证技术，加速验证收敛。

第三个步骤就是验证调试阶段，其中最重要的问题是提高验证效率和缩短调试周期。大家都知道软件仿真的速度慢，但是可调试性强；而 FPGA 原型平台速度快，但是可调试性差。目前，业界很多大的公司都采用硬件辅助验证平台，俗称硬件加速器。其速度比软件仿真器快，比 FPGA 原型平台慢，一般情况下可以满足 10 ~ 1000 倍的加速。

功能验证只是验证领域中的一部分，广义的验证还包括时序验证、物理验证。作为验证工程师，应该要有所专长，有所专注。总的来说，就是要具有一定的专业技术背景，懂得如何使用验证语言，知道如何采用各种不同的验证技术和策略去解决验证过程中存在的问题。

另外，若验证工程师有一定的设计经验，这将有助于在项目中和设计工程师、系统工程师进行良好的沟通和交流。

希望本书能够给国内有志于在 FPGA/IC 领域发展的朋友带来帮助，祝愿你们学习进步，事业有成！

钟文枫

2010 年 8 月 8 日