中国科学院大学

University of Chinese Academy of Sciences

硕士学位论文

基于 UVM 的高可重用验证平台设计与实现

作者姓名：魏 聪

指导教师：张浩 副研究员 中国科学院微电子研究所

杨赛 高级工程师 北京昂瑞微电子技术股份有限公司

学位类别：工程硕士

学科专业：集成电路工程

培养单位：中国科学院大学微电子学院

2021 年 6 月

Design and Implementation of Highly Reusable Verification Platform Based on UVM

A thesis submitted to

University of Chinese Academy of Sciences

in partial fulfillment of the requirement

for the degree of

Master of Engineering

in Integrated Circuit Engineering

By

Cong Wei

Supervisor: Professor Hao Zhang

School of Microelectronics, University of Chinese Academy of Sciences

June 2021

中国科学院大学

研究生学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明或致谢。

作者签名：魏聪

日　　期：2021年4月15日

中国科学院大学

学位论文授权使用声明

本人完全了解并同意遵守中国科学院有关保存和使用学位论文的规定，即中国科学院有权保留送交学位论文的副本，允许该论文被查阅，可以按照学术研究公开原则和保护知识产权的原则公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：魏聪　　导师签名：

日　　期：2021年4月15日　　日　　期：2021年4月15日

摘 要

随着集成电路技术的不断发展，芯片的规模和复杂度持续增加，使得芯片验证工作的难度呈爆炸式增长，传统的验证方法已经难以满足当前的验证需求，芯片验证逐渐成为影响芯片研发速率的关键因素。因此，为了提高芯片验证的效率，如何搭建具有更高可重用性的验证平台成为解决问题的关键。

本论文针对芯片验证展开研究，选择 SystemVerilog 作为验证语言，以 UVM 作为指导方法学搭建具有高可重用性的验证平台。主要工作如下：

(1)选择 SPI 作为待测设计，搭建了基于 UVM 的验证平台。首先根据 SPI 的具体功能提取出验证功能点，然后对验证平台进行层次化处理，对底层各个组件分别建模。为了保障验证工作的完备性，定义功能覆盖率组件收集覆盖率信息。过程中借鉴软件设计模式的内容，运用随机验证方法和 UVM 中的多种机制，实现验证平台的高可重用性。

(2)基于 SPI 验证平台，以 UART 作为待测设计，完成验证平台的搭建工作。过程中重用 SPI 验证平台内部可重用组件，按照 UART 协议内容完成验证平台中其他底层组件的定义。测试用例的构建运用 sequence 机制，然后通过脚本完成仿真。为了说明在系统级验证的可重用性，本文尝试性地对 SPI 和 UART 进行了综合验证，并且得到了正确的验证结果。

本文创新性的将 UVM 验证方法学与软件设计模式相结合，同传统的验证平台相比，进一步提升了内部各组件的可重用性，加快了验证工作的效率，同时对系统级的验证也具有一定的指导意义。

关键词：UVM，软件设计模式，SystemVerilog，SPI，UART

Abstract

With the continuous development of integrated circuit technology, the scale and complexity of chips continue to increase, making the difficulty of chip verification explode. Traditional verification methods have been unable to meet the current verification needs, and chip verification has gradually become a key factor affecting the rate of chip development. Therefore, in order to improve the efficiency of chip verification, how to build a verification platform with higher reusability has become the key to solving the problem.

This thesis focuses on chip verification, choosing SystemVerilog as the verification language, and UVM as the guiding methodology to build a highly reusable verification platform.The main work is as follows:

(1) Choose SPI as the design under test and build a verification platform based on UVM. First, the verification function points are extracted according to the specific functions of the SPI, and then the verification platform is hierarchically processed, and each component of the bottom layer is modeled separately. In order to ensure the completeness of the verification work, a functional coverage component is defined to collect coverage information. In the process, the content of the software design pattern was used for reference, and the random verification method and various mechanisms in UVM were used to realize the high reusability of the verification platform.

(2) Based on the SPI verification platform, the UART is used as the design under test to complete the construction of the verification platform. In the process, reusable components within the SPI verification platform are reused, and the definition of other underlying components in the verification platform is completed according to the content of the UART protocol. The construction of test cases uses the sequence mechanism, then complete the simulation through scripts. In order to illustrate the reusability of verification at the system level, this article tries to comprehensively verify SPI and UART, and get the correct verification results.

This thesis innovatively combines the UVM verification methodology with the software design mode. Compared with the traditional verification platform, it further improves the reusability of internal components, speeds up the efficiency of verification work, and also has guiding significance for system-level verification.

Key Words：UVM,Software Design Mode,SystemVerilog,SPI,UART

目 录

图目录

表目录

符号说明

缩略语	英文全称	中文对照
UVM	Universal Verification Methodology	通用验证方法学
SOC	System on Chip	片上系统
DPI	Direct Programming Interface	直接编程接口
VMM	Verification Methodology Manual	验证方法手册
OVM	Open Verification Methodology	开源验证方法
SPI	Serial Peripheral Interface	串行外设接口
UART	Universal Asynchronous Receiver/Transmitter	通用异步收发传输器
AMBA	Advanced Microcontroller Bus Architecture	微控制器总线架构
AHB	Advanced High performance Bus	高性能总线
ASB	Advanced System Bus	系统总线
APB	Advanced Peripheral Bus	外围总线
DMA	Direct Memory Access	直接存储器访问
VHDL	VHSIC hardware description language	超高速集成电路硬件描述语言
RTL	Register Transfer Level	寄存器传输级
TLM	Transaction Level Modeling	事务级模型

第1章 绪论

1.1 课题背景及意义

自第一块集成电路问世以后，集成电路的规模就越来越大，集成度也会越来越高，这都对验证提出了很高的要求。在实际工作中，经常会出现由于验证不充分导致芯片流片之后不能按照预期工作，从而给公司带来巨大损失的情况，而UVM验证方法学的出现恰好解决了这一问题，它的可重用性及内部的一些机制、方法都能够极大提高验证效率[1]，而且更能保证验证工作的完备性，从而可以缩短设计周期并降低成本。

随着集成电路设计层次的逐渐提高，在这种情况下，集成电路所能实现的功能越来越强大，集成的IP核也越来越多。但这也带来一个问题，就是加剧了芯片验证的复杂度[2]，使验证工作的工作量急剧攀升，这也是为什么芯片验证所花费的时间可以占到芯片研发过程整体时间的大部分，因此提高验证效率成为亟待解决的关键问题[3]。

业界目前使用的传统验证平台具有一些显著的问题，比如需要手动提供激励，不能产生随机的激励[4]，对于验证的结果只能通过观察输出波形来判断，而且对于验证工作进行到何种程度缺乏一个量化标准，除此之外，在针对系统其它IP模块进行验证时代码的可重用性也很低，这些问题在集成电路规模比较小的时候体现的并不明显，但是随着芯片规模的不断增大，这些问题越发成为制约验证效率提高的主要因素。

由于上面提到的这些制约因素，在对集成有大量IP的SOC进行验证时，传统的验证方法已经不能满足现在对验证的要求。本文的研究意义在于设计出一种基于UVM的具有高可重用性的验证平台，代码的重用避免了重复编写功能相同的组件，所以能够在完成本次验证目标的同时，节省下一次验证工作的时间。

1.2 国内外研究现状

在1990年末，Verilog逐渐成为了集成电路领域应用范围最广的设计语言[5]。因为设计工作是使用Verilog来完成的，所以使用Verilog编写testbench进行验证工作很方便，但前提是设计的复杂度不是很高，而且使用Verilog编写的代码可重用性也不高，只要设计做了一点修改，testbench也要做出相应的修改。

随着设计规模的不断增长，继续使用 Verilog 语言来完成验证工作就显得有些捉襟见肘。虽然后来出现了一些专用于验证的语言，但因为其是出于盈利的目的，所以使用价格昂贵，并没有在业内得到推广。为了解决这一问题，在集成电路领域内催生出了 Accellera 组织，该组织由业内企业及用户组成[6]，致力于创建一种更高效的验证语言。终于在 2005 年，该组织的目标得以实现，IEEE 采纳了该组织提出的 SystemVerilog 语言作为业内的标准。

SystemVerilog 相当于 Verilog 的一个拓展，完全兼容 Verilog，使得习惯于使用 Verilog 的用户可以更快上手。SystemVerilog 作为一种面向对象语言，其同样包含封装、继承和多态等特征[7]，除此之外，其还具有一些非常适合用于验证的特征，比如受约束的随机方法和覆盖率驱动方法等，对于算法类的设计，SystemVerilog 可以通过 DPI 接口调用 C/C++ 中的函数[8]，这些就使得 SystemVerilog 成为了验证语言的主流。

下图所示为 2008 年针对不同验证语言的使用情况，对某设计与验证会议的与会人员进行调查所得到的结果，从图中可以看出，对于前文介绍的三种验证语言，其中 SystemVerilog 的使用频率最高，有接近一半的业内人员将其选为自己工作中的验证语言，然后依次是其他验证语言、Verilog 和 SystemC 语言。从那时起，SystemVerilog 就逐渐占据了验证语言中的主导地位。

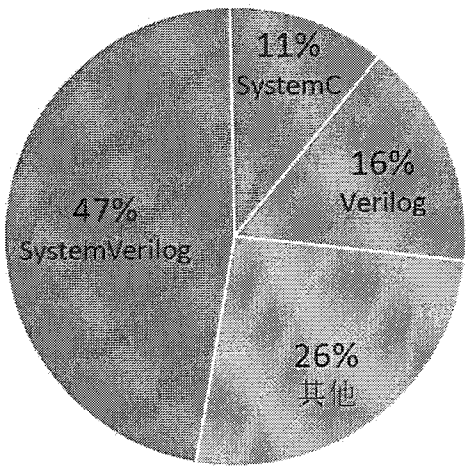

图 1.1 验证语言使用占比

Figure 1.1 Proportion of Verification Language Usage

虽然 SystemVerilog 作为验证语言具有很强的灵活性，但是对于如何搭建验

证平台却没有一个统一的标准，所以业界需要出现一种验证方法学对这个问题进行指导。

Synopsys 在 2006 年提出 VMM 验证方法学，该方法学内部有一套基本类，而且也有用于线程间交互的机制，十分擅长处理复杂问题[9]。该方法学在初期并非是开源的，不过受到来自之后出现的方法学的压力，最终选择了开源。

Candence 和 Mentor 两家公司联合在一起，于 2011 年提出了 OVM 验证方法学[10]。该方法学引入了 factory 机制，但是其内部并不支持寄存器模型，虽然之后推出了相应的补救措施，但是还是没能挽救回来，目前已经停止更新。

Accellera 组织在 2011 年提出 UVM 验证方法学，并且得到 EDA 领域三大厂商的支持。下表所示为三种验证方法学的比较，从中可以看出，UVM 虽然出现的时间最晚，但其是吸取了另外两种方法学优秀之处的产物，所以越来越成为验证方法学的主流[11]，代表了未来的前进方向。

表 1.1 三种验证方法学对比

Table 1.1 Comparison of three verification methodologies

特征	VMM	OVM	UVM
Callback	✔	✔	✔
Phase	✔		✔
Factory	✔	✔	✔
Agent		✔	✔
Configuration		✔	✔
Sequence		✔	✔
TLM		✔	✔
RAL	✔		✔

UVM 验证方法学被推出的第二年，国内相关期刊中就已经出现 UVM 相关的文章，由此可见，不仅是在国外，UVM 在国内的推广也非常迅速。目前，国内规模比较大的芯片设计企业也逐渐改变之前设计人员同时负责验证工作的情况，开始转变为由专门的验证人员负责验证工作。由于 UVM 的一大优点就是可重用性高，所以国内企业大多选择此方法学作为指导，逐步完成本公司验证平台

库的搭建工作。

1.3 论文主要工作

本文选用当前应用范围最广泛的 SystemVerilog 作为验证语言，以 UVM 作为方法学指导，借鉴部分软件设计模式，搭建一个具备高可重用性的验证平台，并基于此平台完成模块级验证。主要工作如下：

(1)选择 SPI 作为待测设计，搭建基于 UVM 的验证平台。首先根据 SPI 的功能提取验证功能点，然后对验证平台进行层次化处理，对底层各个组件分别建模。为了保障验证工作的完备性，定义功能覆盖率组件收集覆盖率信息。过程中借鉴软件设计模式的内容，运用随机验证方法和 UVM 中的多种机制，实现验证平台的高可重用性。在验证运行阶段，根据验证功能点，运用 sequence 机制完成测试用例的构建，通过脚本完成测试用例的运行。

(2)基于 SPI 验证平台，以 UART 作为待测设计，完成验证平台的搭建工作。过程中重用 SPI 验证平台内部可重用组件，按照 UART 协议内容完成验证平台中其他底层组件的定义。测试用例的构建运用 sequence 机制，然后通过脚本完成仿真。为了说明在系统级验证的可重用性，本文尝试性的对 SPI 和 UART 进行了综合验证，并且得到了正确的验证结果。

1.4 论文主要内容

每个章节内容大致如下所示：

第一章：绪论。主要介绍了目前验证工作所面临的主要问题，然后分析了国内外相关研究的发展方向，并对此次研究内容作了简要阐述。

第二章：验证语言及设计模式介绍。主要对几种验证语言的优势与劣势进行了分析，并对设计模式的内容作了简要的说明。

第三章：UVM 验证方法学。主要介绍了进行验证的流程和两种常用的验证方法，最后着重介绍了 UVM 验证方法学的内容，包括验证平台的组成及其内部的各种机制等。

第四章：SPI 验证平台的设计与实现。首先对 SPI 的相关内容作了介绍，然后结合软件设计模式，以 SPI 作为待测设计，为其搭建验证平台，编写测试用例，并且对最后的运行结果展开分析。

第五章：验证平台可重用性应用。重用 SPI 验证平台内部部分可重用组件，搭建一个 UART 的验证平台，之后运行此验证平台并分析仿真结果，以此证明验证平台的高可重用性，最后对系统级的重用进行尝试。

第六章：总结与展望。总结本次课题所做的工作，并对其中的不足加以反思。

第 2 章　验证语言及设计模式介绍

验证工作一般分为系统级的验证和模块级的验证，本课题的研究内容主要涉及模块级的验证。在开始验证工作之前，验证人员需要熟悉各种验证语言的优缺点，除此之外，还可以了解一些可以提升验证效率的其他方法，本章将会对这些内容进行详细介绍。

2.1　验证语言

可以用来进行验证的语言不止一种，但归根结底验证是依赖于设计的。可以用来对 Verilog 设计进行验证的语言一般有三类：Verilog，SystemC 和 SystemVerilog。下面将对其展开具体介绍。

2.1.1　Verilog 语言

Verilog 语言是专门用于芯片设计的语言，但是在发展之初没有专门的验证工程师，通常都是由设计工程师同时完成验证的工作。为了适应这种情况，Verilog 中内置了一些用于验证的内容，其中最典型的就是 initial 语句，initial 语句是不可综合语句，在设计代码中是不允许出现的，其存在意义就是为了方便进行验证工作。除此之外还有 task 和 function 语句，task 和 function 语句在设计中用到的频率也不是很高，但是当用到验证工作中时，却可以在很大程度上缩减验证的代码量，同时提高验证代码的可读性。

Verilog 用来进行验证工作时也存在一些问题，其中问题最明显的就是在验证环境的功能模块化以及受约束的随机验证方面存在不足，这些问题就导致在进行验证工作时只能采用定向激励的方法，而不能使用随机测试方法，这两种方法在使用测试向量的数量上存在非常巨大的差距，同时由于功能模块化不足，导致整个验证环境几乎不可重用。随着芯片中的逻辑越来越复杂，单纯的 Verilog 验证已经不能满足如今的验证需求[12]。

2.1.2　SystemC 语言

SystemC 是基于 C++的一种语言，也可以说其实是 C++的一个库，是专门为了熟悉 C\C++的工程师设计的，具有软硬件协同设计的特性[13]。SystemC 除了具

有比较高的抽象性，同时还可以体现出硬件设计中的时钟延迟、信号同步等信息，这就为工程师提供了一个公共平台。一般来讲，集成电路大致可以分为两类，一类是对算法要求比较低的，比如 SPI 和 UART 等通信协议，另一类是对算法要求比较高的，比如视频处理、图像处理等。那些对算法需求比较复杂的设计在编写 Verilog 代码之前，工程师会使用更高级的语言，比如 C 或 C++搭建一个参考模型，该参考模型可以实现和 Verilog 代码相同的功能，然后将其集成到验证环境中，在进行验证工作时比较参考模型和 Verilog 代码的输出信息，以此来验证 Verilog 代码的正确性。所以 SystemC 基于 C++的这一特性使它非常适应此类对算法要求比较高的设计，同时，在对算法要求比较低的设计中表现得也不错。

基于 C++这一特性是 SystemC 最大的优点，但同时也是它最大的缺点。在 C++中，工程师在使用动态存储变量时，就面临需要自主管理设计中内存的问题，在这个过程中一旦出错，通常就会造成比较严重的后果[14]。除此之外，在对设计进行验证的过程中，往往需要考虑一些异常情况，而使用 SystemC 进行异常测试用例的编写并不方便。

2.1.3 SystemVerilog 语言

SystemVerilog 扩展自 Verilog，可以向下完全兼容 Verilog，同时又扩充了 C 语言的接口、结构体和数据类型等内容，在此基础之上又加入了断言等内容，其体系结构图如图 2.1 所示。除此之外，作为一种面向对象语言，其具有面向对象语言的所有特征，进一步提升了作为验证语言的实用性[15]。

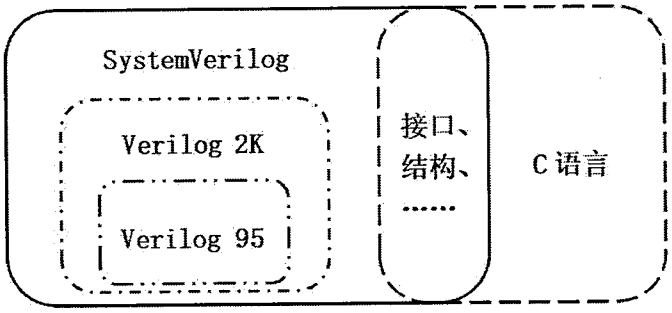

图 2.1 SystemVerilog 体系结构图

Figure 2.1 SystemVerilog architecture diagram

在 SystemVerilog 中，面向对象是通过类(class)来实现的。类是通过将数据结

构和相应的函数或者任务组合在一起形成的一种新的组织结构，在类的内部有两个部分，一个部分是数据结构，又被叫做类内的成员变量，另一部分是相应的函数或任务等方法，又被叫做类的接口。把一个类定义好之后，在使用类内部的方法之前，必须先将该类实例化，否则就会发生错误。

封装是面向对象编程的特征之一，其本质实际上就是信息隐藏，在 SystemVerilog 中指的就是通过类将一些变量和对应的方法封装起来，构成完整的一个个体[16]。在类的内部，对于一些不想被外界访问或者调用的的数据结构和相应的方法，可以将其设置为私有类型，私有类型的数据和方法只可以在类的内部被访问和调用，如果在外部对其访问和调用则会出错。所以封装可以有效降低类之间的耦合，并且可以对类内部的数据结构进行更加精准的控制，当需要对成员变量和方法进行适当的修改时，不会对外部造成大的影响。

继承是面向对象编程的又一大特征，指的是在已经定义好的类的基础之上定义一个新类[17]，其中已经定义好的类被称作父类，在父类基础上定义的新类被称作子类。子类会继承父类除私有内容外所有的内容，并且是不可选择的继承所有内容。当然继承也有其不好的地方，就是继承会极大地加强类与类之间的耦合，当父类内容改变时，子类内容也会被动改变。

多态是面向对象编程的第三个特性，也是最令人难以理解的一个特性，指的是程序中定义的引用变量所调用的方法具体是在哪个类中定义的，必须在程序真正运行时才确定[18]。这样一来，不需要改动调用程序的源代码，而只是改动被调用程序的代码，就可以让源代码中的引用变量表达出不同的实现。多态的实现也是需要条件的，运用多态的类之间必须是父类和子类的关系，并且父类中定义的方法必需是虚方法，然后在子类重新定义该方法。

SystemVerilog 作为一种面向对象的编程语言，集合了多种语言的优点，并且增加了一些便于验证的内容，无论是对于算法类设计还是非算法类设计都有不错的表现，这就使得 SystemVerilog 越来越成为三种验证语言中的主流。

2.2　设计模式

设计模式的概念在上世纪九十年代被首次提出，其代表的是经过实践验证过后在软件设计过程中最有利的模式，如今已经被广泛应用到软件开发的过程中。设计模式一般而言有 23 种，其中每一种模式都有与其相对应的原理，都代表了

一种在实践过程中经常遇到的问题，在进行软件设计的过程中合理使用设计模式可以方便的解决遇到的很多问题。

设计模式有六大原则，23 种设计模式都是基于六大原则提出来的。这六大原则都是为了提高程序的易用性，基于此提出的设计模式自然也是出于相同的目的，当面向对象编程与设计模式相结合时，可以更好的发挥面向对象的优势。下面将对部分设计模式进行详细介绍。

2.2.1 中介者模式

中介者模式指的是通过定义一个中介对象将多个有通信需求的对象封装在一起，各个对象只需要与中介对象进行通信[19]，使得有通信需求的对象不再需要通过显示调用就可以完成通信过程，从而达到了降低对象之间耦合度的目的。

当两个对象之间存在频繁的交流时，如果不使用中介者模式，会使得整个系统的结构显得非常复杂，形成一种网状结构，如图 2.2 所示，此时如果系统中一个类发生改变，那么与之相关联的类也需要被检查是否需要做出相应的更改。

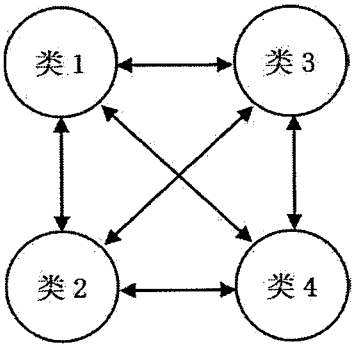

图 2.2 无中介对象通信过程

Figure 2.2 No mediation object communication process

当引入中介者模式之后，对象间通信过程如图 2.3 所示，会形成一种星型结构，此时系统中引入了一个中介对象，类与类之间的通信全都通过这个中介对象来进行，使得一个类与多个类之间的耦合关系转变成类与中介对象之间的耦合关系，即使对其中某一个类进行适当修改，系统中其他的类也不会受此影响。中介者模式也有其不足之处，就是封装的类越多，中介对象就会越复杂，这就在一定程度上增加了其维护的难度。

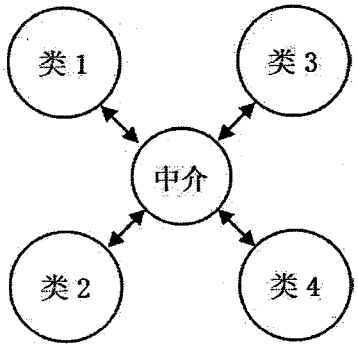

图 2.3 引入中介对象通信过程

Figure 2.3 Introducing the mediation object communication process

2.2.2　模板方法模式

模板方法模式属于行为型模式的一种，指的是只在一个模板对象中定义执行某种操作的框架，而具体的实施过程延迟到其他对象中进行定义[20]。模板方法模式使得在不改变模板类的情况下，就可以重新定义某些操作的具体内容。

在实际应用中，当需要完成某种操作，该操作具有多个固定的步骤，而其中的步骤又根据操作对象而有所不同时，可以考虑引入模板方法模式。假如某种操作需要三个方法来完成，但是针对不同对象，每个方法的具体内容会有所变化，如果此时没有引入模板方法模式，那么当操作对象改变时，就需要重新检视整个程序并作出相应的修改，这在很大程度上降低了代码的可重用性。

当引入模板方法模式之后，如图 2.4 所示，在系统中定义一个模板类，在模板类中只定义完成此种操作所需要三个方法的框架，而三个方法的具体内容下延到具体类中定义，当操作对象发生变化时，只需要修改具体类，而模板类不需要进行任何修改，从而提高了程序的可维护性和可重用性。当然模板方法模式也有其缺点，就是每一个操作对象可能都需要对应一个具体类，这样就导致类的数量增加，使系统变得更复杂。

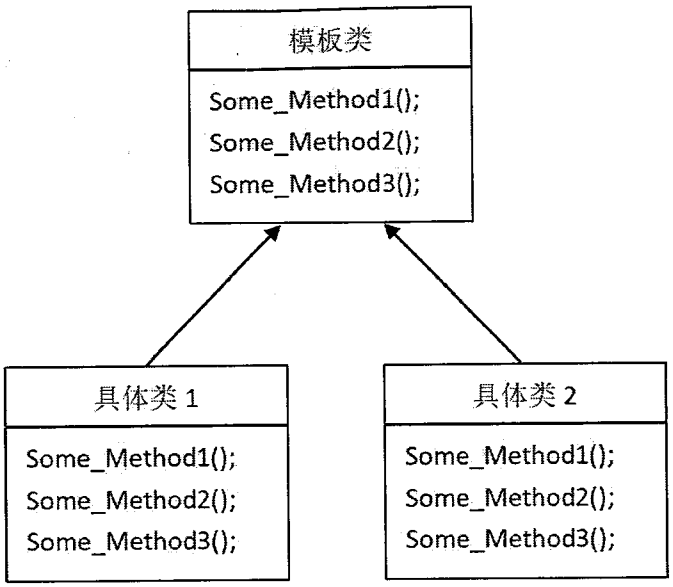

图 2.4 程序中引入模板类

Figure 2.4 Introduce a template class into the program

2.2.3 外观模式

外观模式的本质就是将多个子系统隐藏起来，只保留一个统一的外观对象，对子系统的访问经由外观对象来实现[21]，这样就隐藏了每个子系统的具体实现细节，降低了对子系统进行访问时的复杂度，为子系统的访问提供了一个更简便的接口。

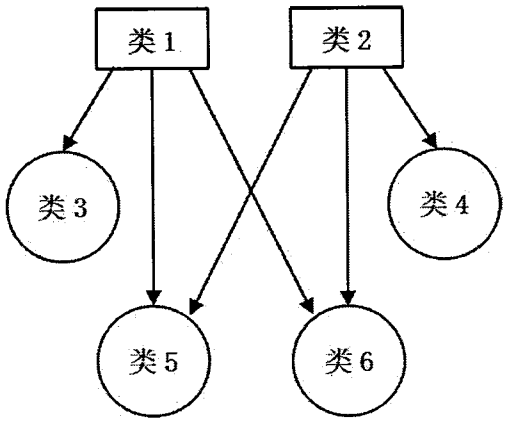

图 2.5 无外观对象访问过程

Figure 2.5 No facade object access procedure

在实际应用中，当需要进行某种操作，而完成该操作需要访问多个子系统时，

为了简化整个流程，可以考虑引入外观模式。如图 2.5 所示，类 1 和类 2 为了完成某项操作，需要对类 3、类 4、类 5 和类 6 进行访问，如果没有引入外观类，则类 1 和类 2 需要逐个对它们进行访问，这就增强了类与类之间的耦合，并在一定程度上增加了系统的复杂度。

当引入外观模式之后，如图 2.6 所示，在系统中定义一个外观类，类 1 和类 2 不再直接对其他类进行访问，而是通过外观类来完成这一过程，这样就避免了不同类之间直接进行通信的情况，使子系统的访问变得更加简洁。

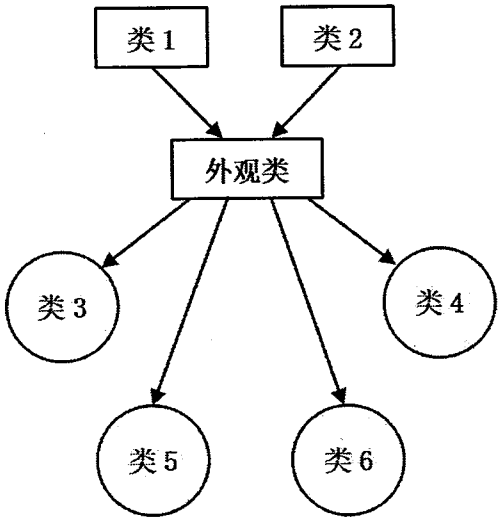

图 2.6 引入外观对象访问过程

Figure 2.6 Introduce a facade object access procedure

2.2.4　适配器模式

适配器模式也属于结构型模式的一种，在一个系统中，其主要作用就是充当两个不兼容的对象之间进行沟通的桥梁，使两个原本不能在一起工作的对象能够进行正常的交互[22]。

在软件设计过程中，有时会想要将一个对象添加到现有的系统中，但是该对象所需要的接口当前系统不一定可以满足，此时就需要引入适配器模式。如图 2.7 所示，如果不引入适配器，类 1 和类 2 由于接口不匹配导致无法进行正常的交互，而当在系统中引入适配器之后，该适配器担当起了两个类之间沟通桥梁的角色，使得它们能够进行正常的交互。这就在一定程度上提高了类的可重用性，同时也使系统的灵活性得到了提高。除此之外，适配器模式也有一个很大的缺点，就是如果系统中引入适配器过多，会导致系统结构变得很混乱，所以如果非必要，

可以尽量不引入适配器模式。

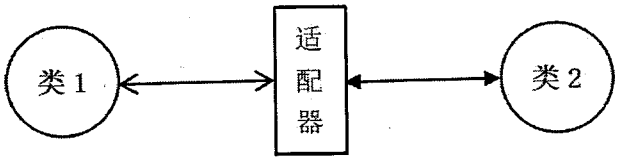

图 2.7 适配器模式示意图

Figure 2.7 Schematic diagram of adapter mode

第 3 章　UVM 验证方法学

验证与设计之间有着密不可分的关联，只有两者紧密合作才能更好地完成芯片前端设计的工作，但是随着芯片规模越来越大，验证工作的工作量呈几何倍数增加，此时就需要一种方法论来指导验证工作的进行。本章将对验证的内容，验证过程中使用的方法和 UVM 验证方法学进行详细介绍。

3.1　验证概述

芯片验证是为芯片设计服务的，验证的目的就是确保芯片的设计能够在满足所有设计需求的同时没有设计漏洞，而且验证贯穿于芯片前端设计的各个流程之中。现代数字芯片的设计流程一般都是由一份需求说明开始，其大致过程如图 3.1 所示。

拿到需求说明书之后，应该将其整理为特性列表。设计人员就参考其中的内容，制订设计规格说明书，在这份规格说明书中介绍本次设计的一些具体内容。与此同时，验证人员也会根据这份特性列表制定一份验证规格说明书，然后在这份说明书中介绍本次验证工作的一些内容，比如如何搭建验证环境，如何编写测试用例才能将特性列表完全覆盖。

设计工程师在完成设计规格说明书之后，就要使用设计语言，比如 Verilog，将特性列表的内容按照规格说明书内的方法翻译成 RTL 代码。验证工程师在完成验证规格说明书之后也要使用验证语言，比如 SystemVerilog，根据规格说明书的内容进行验证环境的搭建，并且开始编写测试用例[23]。

当设计人员完成 RTL 代码的编写之后，验证人员就要使用搭建好的验证环境开始对设计代码进行验证，主要是验证设计代码是否满足了特性列表的所有需求功能，设计代码的输出结果是否与特性列表的结果相同，当出现异常情况时设计代码能否按照特性列表的要求进行处理等等。

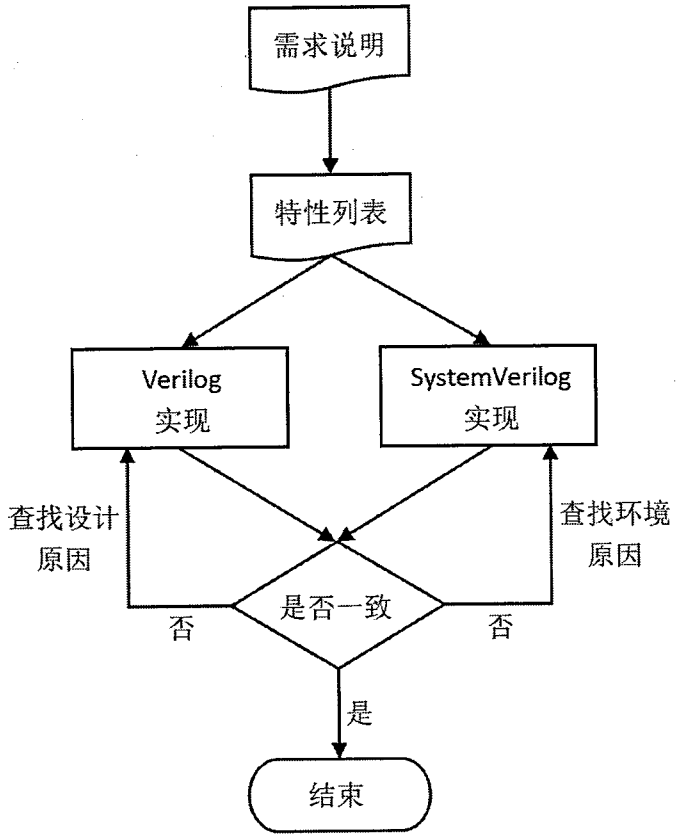

图 3.1 芯片前端设计流程图

Figure 3.1 Chip front-end design flow chart

如果经过验证之后发现，设计代码的所有行为表现都与预期一致，那么此时就代表验证工作完成，可以继续进行芯片设计后面的流程。如果验证结果显示设计代码的行为表现与预期不完全一致，那么此时就需要检查是搭建的验证环境存在问题还是设计代码存在问题，找到问题并修改之后继续进行验证，直到结果一致就说明验证工作可以结束了。

3.2 验证方法

现如今，随着芯片复杂度的不断提高，验证所花费的时间占整个芯片设计周期的比重越来越大，缩短芯片研发周期的关键之一就是提升验证的效率，所以为了提升验证效率，在进行验证工作时也需要采取一些方式方法。下面将会对验证工作中采用的几个验证方法做具体介绍。

3.2.1　约束随机方法

在验证工作中，验证平台需要为待测设计提供激励，激励一般分为定向激励和受约束的随机激励。定向激励只针对待测设计的某个具体特性，而受约束的随机激励覆盖的范围更大，一次可以完成对多个特性的验证[24]。

定向激励的创建方法更加简单，而且在一个较短的时间内就可以得到结果，所以如果项目时间充足，使用定向激励完全可以完成验证任务。但是定向激励也有一个缺点，因为其只针对某个具体特性，所以定向激励只能发现待测设计中预期的漏洞。

如果项目时间紧迫，此时再使用定向激励就有可能会影响到项目进度，所以这时应该考虑使用随机激励。随机激励也需要受到约束，否则不仅不能提升验证效率，还会因为进行了一些无用甚至错误的仿真而拖慢了项目进度。受约束的随机激励创建方法要比定向激励复杂，而且随机激励不针对某一具体特性，所以为了衡量项目进度而引入了功能覆盖率来评估验证工作的进展状况，为了能够对仿真结果进行比对而引入了计分板。由于随机激励的随机性，在运行仿真时还有可能发现意料不到的漏洞，这是定向激励所做不到的。

由于随机测试需要搭建验证平台和创建测试激励，所以其前期准备过程所消耗的时间要比定向测试多的多。当准备工作完成开始运行仿真的时候，随机测试会以一个比较快的速度完成验证任务，而反观定向测试，虽然运行第一次仿真的时间要比随机测试早，但是项目推进速度要比随机测试慢的多，所以总的来看，其完成验证任务所需要的时间比随机测试还要长。

3.2.2　覆盖率驱动方法

在验证平台中，覆盖率被用来判断验证工作的进度，覆盖率主要分为功能覆盖率和代码覆盖率。

功能覆盖率代表了已经验证的特性占总特性的比重，其信息的收集需要在验证环境中编写额外的代码，根据特性列表定义覆盖组，在覆盖组中又定义覆盖点[25]。等到一次仿真完成之后查看覆盖组和覆盖点的覆盖情况，并以此为根据，判断下一步测试激励的修改方向，以期达到更高的覆盖率。代码覆盖率代表了设计代码的运行程度，其信息的收集不需要额外的操作，只需要在编译仿真命令中增加具体的命令就可以完成相应信息的收集。完成仿真之后，通过代码覆盖率可以

非常直观的看到设计代码的覆盖情况，并且指明了下一步的验证方向。

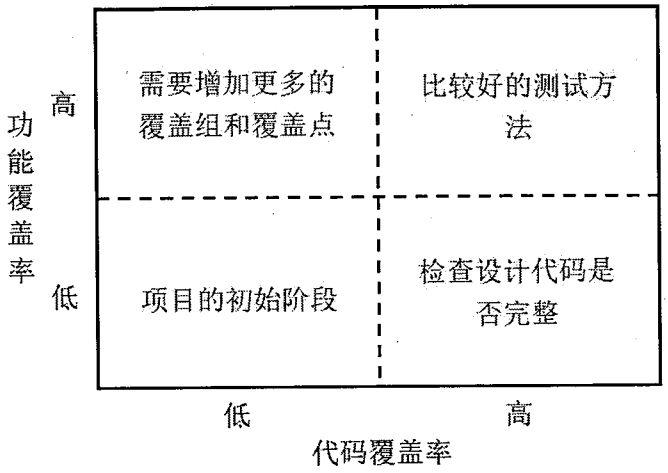

图 3.2 覆盖率情况分析

Figure 3.2 Coverage analysis

验证工作的完备性通常需要综合考虑功能覆盖率和代码覆盖率信息。从图 3.2 可以看出，在验证工作开始的初期，两种覆盖率都比较低，需要运行更多的测试激励；随着项目逐渐推进，如果功能覆盖率和代码覆盖率都达到了比较高的水平，说明这是比较好的测试方法；如果功能覆盖率达到了比较高的水平，而代码覆盖率还处于比较低的水平，说明需要根据特性列表检查是否需要增加更多的覆盖组和覆盖点，又或者是设计代码有冗余需要删减；如果代码覆盖率达到了比较高的水平，而功能覆盖率还处于比较低的水平，说明需要根据特性列表检查是否定义了多余的覆盖组和覆盖点，又或者是设计代码并没有实现所有需求说明内要求的功能。只有当两者都达到 100%或者是很高的水平，而且没有漏洞再出现，此时才能说明验证工作的完成。

3.3 UVM 方法学介绍

SystemVerilog 是前文介绍的三种验证语言中使用频率最高的，用来搭建验证平台是完全可行的，但是会有一定难度，所以为了提高工作效率和验证平台的易用性，在搭建验证平台的过程中需要使用一些方法学内容。UVM 因为结合了另外两种验证方法学的优点，所以现在逐渐成为了验证方法学的主流。下面将对 UVM 验证方法学进行详细介绍。

3.3.1　验证平台组成

为了提高搭建验证平台的效率，UVM 验证方法学为用户提供了一个文件库，库内提供了一些常用的基类还有方法，这就为验证人员在搭建验证平台时省去了很多底层工作。在验证平台搭建好之后需要将待测设计放入此平台，然后为其提供适当的激励，通过输出结果来判断待测设计的功能是否正确[26]。

验证平台通常由多个组件共同构成，每个组件都在其中承担不同的任务，典型的 UVM 验证平台结构如图 3.3 所示。

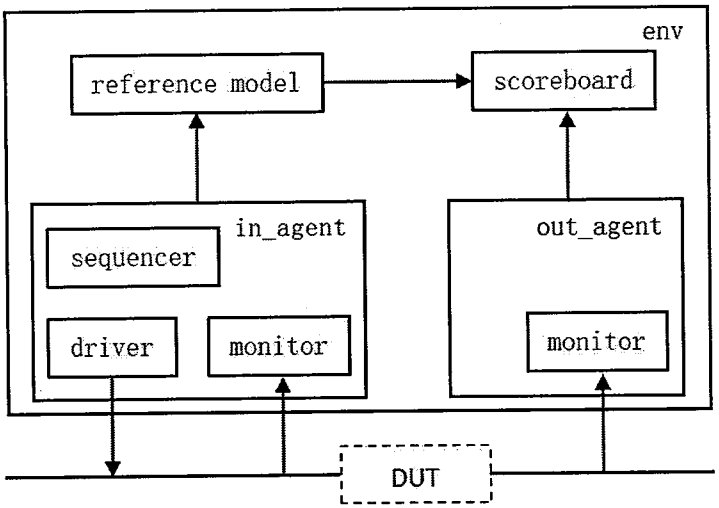

图 3.3 典型 UVM 验证平台结构

Figure 3.3 Typical UVM verification platform structure

图中各组件的功能描述如下：

（1）driver 组件作为验证平台中的驱动器，负责为激励数据按照具体协议增加时序信息，然后通过接口将其驱动给待测设计；

（2）monitor 组件作为验证平台中的监视器，上图典型 UVM 验证平台中定义了两个 monitor，其中左边的 monitor 负责监视待测设计的输入数据，右侧的 monitor 负责监视待测设计的输出数据，之后就将得到的数据发送给验证平台中的其他组件；

（3）sequencer 组件作为验证平台中的序列器，主要负责将激励数据发送给 driver 组件，当有不止一笔激励数据请求同时发送时，sequencer 还负责对其进行仲裁；

（4）agent 组件作为验证平台中的代理器，其主要作用就是将处理相同协议

的组件封装起来，图中的 agent 组件在内部封装了上述三个组件，并实例化了两个对象：in_agent 和 out_agent，因为 out_agent 不需要为待测设计提供激励，所以在其内部只实例化了 monitor 组件。

（5）reference model 组件作为验证平台中的参考模型，其内部实现的功能在理想情况下和待测设计是一样的，其输入数据来自 in_agent 内的 monitor 组件，然后产生一份预期的输出数据用于与待测设计的输出进行比较；

（6）scoreboard 组件作为验证平台中的计分板，它会得到两份数据，一份来自参考模型，另一份来自 out_agent 内的 monitor 组件，之后对这两份数据进行比较，如果比对结果不同，就说明验证平台或者待测设计存在漏洞。

（7）env 组件作为验证平台中的环境，其主要作用就是将验证平台中的其他组件封装成一个整体，将测试激励和验证平台分开。

3.3.2 objection 机制

在 UVM 验证平台中，objection 机制在使用时需要被提起和撤销，分别通过 raise_objection 和 drop_objection 实现[27]。另外，objection 机制只存在于 task phase 中，function phase 中不存在 objection 机制。

在每一个 task phase 中，只有使用 raise_objection 将 objection 提起，该 task phase 的内容才会被运行，如果 UVM 没有检测到 objection 被提起，则当前 task phase 会被跳过，转而执行下一个 phase，如果下一个 task phase 中 objection 仍然没有被检测到提起，则继续向下检测，直到有 objection 被提起。task phase 运行结束后，还需要通过 drop_objection 将 objection 撤销，否则验证平台就会被卡在当前 task phase 无法继续执行后面的 phase，只有当 UVM 检测到当前 task phase 中不存在 objection 被提起还没被撤销之后，才会按照前文介绍的顺序执行下一个 phase。所以，通过 objection 机制实现对 phase 机制的控制，可以完全决定验证平台的整个运行流程。

task phase 中的 run_phase 是一个比较特殊的 phase，有两种方式可以控制其开始和结束。第一种方式是由其他 task phase 决定，因为 run phase 和其他 12 个 task phase 是并行运行的，所以在其内部可以不用提起 objection，这种情况下其内部程序也会与其他 task phase 并行执行；另一种方式就是在 run_phase 内部通过 raise_objection 和 drop_objection 来提起和撤销 objection，这样其开始和结束

都由 run_phase 自己决定。

3.3.3　TLM 通信机制

在 UVM 验证平台中，各个组件的功能划分比较独立，因此 TLM 机制通常被用来实现两个组件之间的通信。TLM 依赖于事务(transaction)进行传输，所谓事务就是将多个数据封装在一起的一个类，组件之间的通信一般都是基于事务进行的[28]。

通信过程涉及通信的主动方和被动方，其中在主动方内部定义的端口称作 port，在被动方内部定义了两个端口：export 和 imp。端口之间通信常用的操作有三种：put、get 和 transport，而且这三种操作都由阻塞和非阻塞之分。put 操作是指通信的主动方向被动方发送一个事务信息，其中数据是从主动方流向被动方，如图 3.4(a)所示；get 操作是指通信的主动方向被动方请求一个事务信息，其中数据是从被动方流向主动方，如图 3.4(b)所示；transport 操作相当于先进行一次 put 操作，再进行一次 get 操作，如图 3.4(c)所示。在使用 TLM 进行通信时，需要根据具体操作，在被动方内部定义相应的 put、get 和 transport 函数，否则系统运行时会报错。

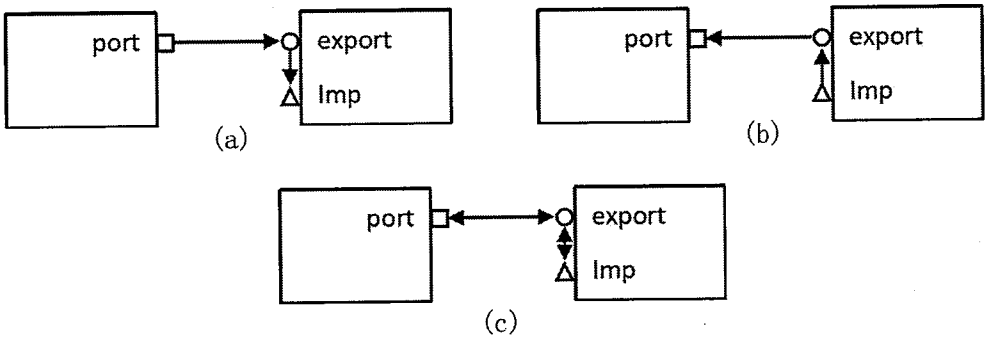

图 3.4 TLM 中三种操作

Figure 3.4 Three operations in TLM

TLM 中除了以上三类端口外，还有另外两类：analysis_port 与 analysis_export。这两类端口的传输也是基于事务进行的，但是其与另外三种端口有两点区别：第一点是前者可以实现一对多进行传输，而另外三种端口只可以进行一对一通信；第二点就是前者没有阻塞与非阻塞的概念，不需要等待被动方的响应。

在使用 analysis_port 和 analysis_export 进行通行时，和其他三个端口类似，

也需要在被动方内部定义一个 write 函数。UVM 提供了另外一种方法可以省略 write 函数的定义，就是在通信过程中加入一个 FIFO，如图 3.5 所示，这样就使得原来的被动方也变成了主动方，整个通信过程中只有 FIFO 是唯一的被动方。

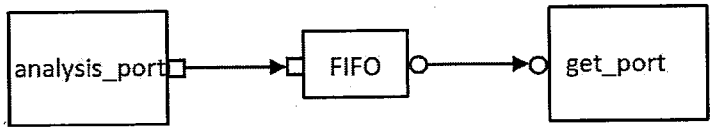

图 3.5 TLM 使用 FIFO 通信

Figure 3.5 TLM uses FIFO communication

使用 FIFO 和使用 imp 两种方式各有优势和劣势，在具体的使用过程中，需要根据具体情况做出最优的选择。

3.3.4 sequence 机制

UVM 验证方法学的一个优势就是可以将验证平台的搭建和测试激励的编写完全分开，其中测试激励的编写就是通过 sequence 机制完成的，不同的 sequence 对应不同的激励，充分体现了其可重用性。sequence 机制包括 sequencer 和 sequence 两部分，sequence 负责产生不同的事务信息，sequencer 负责将事务信息通过 TLM 的方式发送给 driver[29]。

sequencer 就像 sequence 和 driver 之间的指挥官。sequence 在向 sequencer 发送事务信息之前会先向 sequencer 提交一个申请，sequencer 会将这个申请放入其内部一个队列，driver 在向 sequencer 请求事务信息时，若此时队列内有申请则 sequencer 就会将此事务信息发送给 driver。如果 driver 没有向 sequencer 请求信息，或者其内部队列内没有 sequence 提交的申请，则 sequencer 此时处于等待状态。

sequence 的启动方法通常是是通过将其设置为某一个 sequencer 的 default_sequence 来实现。这其中又存在两种不同的方式，第一种方式就是直接将 sequence 设置为 sequencer 的 default_sequence，这种方式会自动实例化该 sequence；第二种方式就是先将 sequence 实例化，然后此时可以对该 sequence 做一些适当修改，之后再将其设置为 sequencer 的 default_sequence。

所有 sequence 内部都有且仅有一个名为 body 的任务，sequence 被启动之后，其内部的 body 任务会自行执行。事务信息的生成一般都在 body 任务中进行，其

中使用频率最高的方式就是通过调用 uvm_do 宏来完成，uvm_do 宏主要完成三个动作：首先是将事务信息实例化，然后为内部随机变量赋随机值，最后将事务信息发送给指定的 sequencer。除了 uvm_do 宏外常用的还有 uvm_do_with 宏和 uvm_do_on 宏等，其中 uvm_do_with 宏可以对事务信息内部的随机变量做一些约束，uvm_do_on 宏可以指定将生成的事务信息发送给特定的 sequencer。

当验证平台结构比较简单，只需要一个 sequence 提供事务信息，或者需要多个 sequence 但是其同步关系比较简单时，上述内容就足够验证人员比较方便的完成测试激励的编写，但是如果多个 sequence 之间的同步关系比较复杂，此时就需要考虑引入 virtual sequence。顾名思义，virtual sequence 即虚拟的 sequence，其不需要生成事务信息，只需要对其他 sequence 进行调度。与 virtual sequence 搭配工作的还有一个 virtual sequencer，如图 3.6 所示，virtual sequencer 内部包含有指向验证平台中其他 sequencer 的指针。在实际运用时，可以将 virtual sequence 设置为 virtual sequencer 的 default_sequence，然后在 virtual sequence 的 body 任务中通过 uvm_do_on 宏来启动其他 sequence 并为其指定 sequencer。

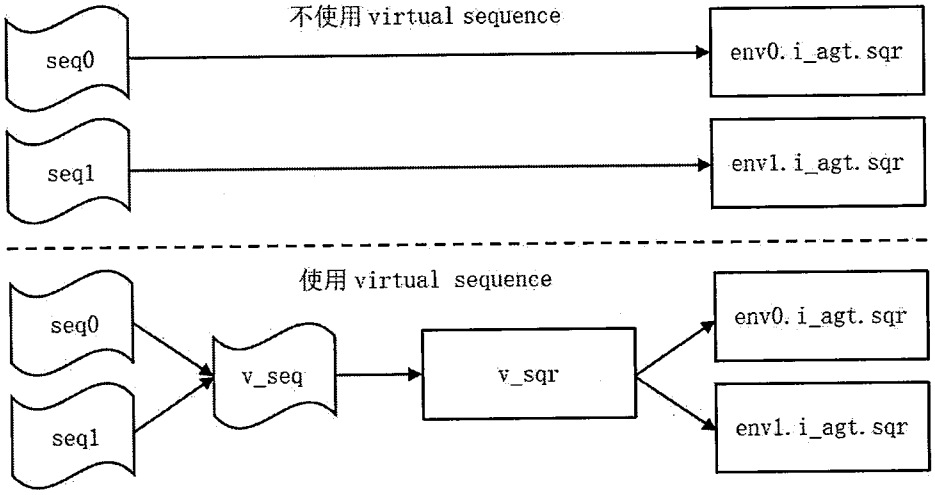

图 3.6 virtual sequence 示意图

Figure 3.6 virtual sequence diagram

3.3.5　config_db 机制

平台中各组件之间基于事务的传输是通过 TLM 机制完成的，而当需要传递参数时则是通过 config_db 机制完成的。通过 config_db 机制进行参数传递时需要使用 set 函数和 get 函数相互配合，其中 set 函数用来寄送数据，get 函数用来接

收数据[30]。

set 函数有四个参数，其中第一个参数必须是验证平台中某组件的实例，第二个参数是目标路径相对于第一个参数的路径，第三个参数是一个标记，用于进行适当的说明，第四个参数是具体要进行传递的参数名；get 函数也有四个参数，第一、二个参数用来指明路径，第三个参数要求和 set 函数的第三个参数完全一致，第四个参数是要接收的参数名。

set 函数和 get 函数往往是成对出现的，如果同一个参数使用 set 函数寄送了两次，而只用 get 函数接收了一次，则此时需要比较两次 set 操作发起的优先级，即比较 set 前两个参数的层次，所谓层次就是其在 UVM 树中的位置高低，层次越高的 set 操作具有越高的优先级，才可以被 get 函数成功接收。如果两次 set 操作的层次相同，则第二次 set 操作可以被 get 函数成功接收。

其实，config_db 机制不仅可以用来传递参数，还可以用来传递接口等，除此之外，上文提到的设置 default_sequence 也是通过 config_db 完成的。

3.3.6 factory 机制

在 UVM 验证方法学中，factory 机制能够根据类名创建该类的一个实例，并自动调用该类内部的方法[31]，而一个更重要的功能就是可以实现重载，这里的重载是不同于一般面向对象语言中重载的另一种重载。

factory 机制的实现首先需要将类注册到 UVM 内部一个表中，这个过程通过调用 uvm_component_utils 宏或者 uvm_object_utils 宏来实现，这时就可以实现根据类名创建实例并调用内部方法的功能了。要想实现重载的功能还需要额外的几点要求，第一点是类需要使用 type_id::create 的方式构建对象；第二点是两个类之间不能是毫无关系的两个类，而是需要有继承关系，被重载类是重载类的父类；第三点是重载类和被重载类需要派生自 uvm_component 类或 uvm_object 类或者其派生类，即不可以派生自不同类型的基类。满足这几点条件之后，就可以应用 factory 机制的重载功能了。

在具体应用重载功能时，还需要用到 set_type_override_by_type 等函数，该函数有三个参数，其中第一个是被重载类的类型，第二个是重载类的类型，第三个参数只有在同一个类被多次重载时才有意义。此时，如果系统想要实例化一个类，就会先查找该类的是否存在重载记录，如果有，则会创建一个重载类的对象，

如果没有查找到重载记录，则会创建一个原本类的对象。

3.3.7　寄存器模型

UVM 验证方法学中的寄存器模型借鉴自 VMM 方法学，因为待测设计中一般都会存在数量非常多的寄存器类型变量，通过寄存器模型不仅可以实现对内部寄存器的配置，还可以在验证平台中的其他组件，比如 reference model 中实现对寄存器数据的实时读取，极大地简化了对待测设计中寄存器的读写操作。具体过程如图 3.7 所示，其中左图为不引入寄存器模型的方式，右图为引入寄存器模型的方式。

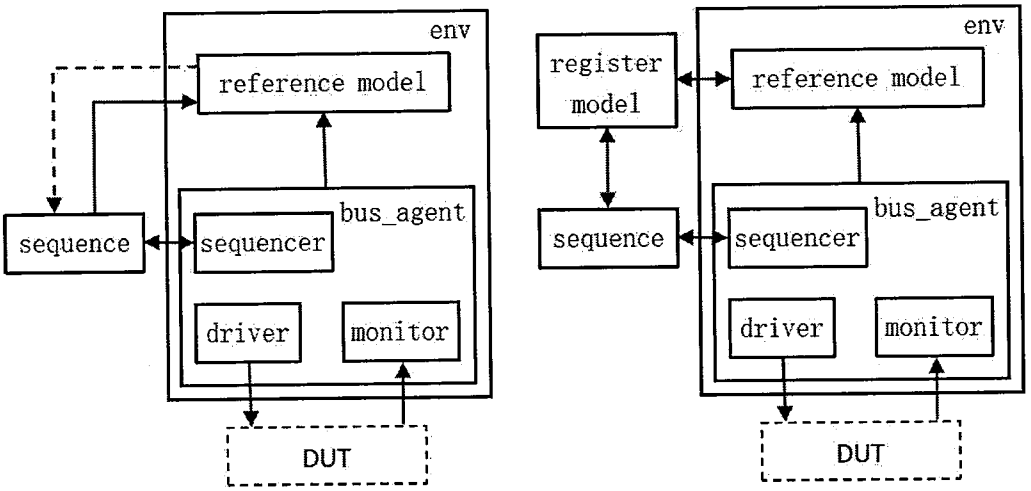

图 3.7 读取寄存器两种方式

Figure 3.7 Two ways to read the register

对寄存器模型访问有两种方式，其中前门访问就是通过总线完成对寄存器的读写访问，后门访问就是通过寄存器的路径信息完成对其的读写访问[32]。这两种方式各有优缺点，前门访问会占用仿真时间，如果需要对大量的寄存器进行配置时，由于后门访问不占用仿真时间，所以后门访问的速度要比前门访问快得多，但也正因如此，后门访问过程在仿真波形中得不到体现，如果遇到需要调试的情况会很不方便，除此之外，对于一些只读寄存器，使用前门访问的方式对其进行写操作无效，而使用后门访问的方式却可以将数据写进去。

UVM 提供了几种访问寄存器模型的函数。除了常用的 write 和 read 函数之外，还有 poke 和 peek 函数，其中后者只能用于后门访问，而且也有多个参数，常用的只有前两个，第一个参数用于表示此次读写操作是否成功，第二个参数是

此次操作要读取的变量或要写入的值。write、read 函数和 poke、peek 函数还有一个非常显著的不同，就是前者会考虑寄存器的具体属性，但是后者则不会考虑这些，即使是只读寄存器，使用 poke 函数也可以将数据写入。

除了上文介绍的读写函数之外，寄存器模型还提供了 set、get 和 update 等函数。set 函数用来设置寄存器模型的期望值，get 函数用来提取寄存器模型的期望值，update 函数用来检查寄存器模型的期望值与镜像值是否相同，若不相同则将期望值写入待测设计中，同时更新镜像值。其中期望值和镜像值是为了保持寄存器模型的值与待测设计中寄存器值的一致性而引入的，write、read 函数和 poke、peek 函数会同时更新期望值和镜像值。

因为寄存器的检查方法大同小异，所以 UVM 提供了多个内建的 sequence 用来检查寄存器模型和待测设计中寄存器的值。这些 sequence 的存在，在一定程度上缩减了验证工程师的工作量。

第 4 章　SPI 验证平台的设计与实现

SPI 因其数据传输速度比较快，节省资源等优点，被广泛应用在 SOC 的设计中。鉴于其超高的使用频率，搭建一个 SPI 的验证平台是非常有必要的。本章将会对 SPI 的具体功能和 SPI 验证平台的搭建及运行过程进行详细介绍。

4.1　SPI 协议介绍

SPI 是摩托罗拉公司提出的一种同步通信接口，不仅可以实现并行数据到串行数据的转换，工作于全双工模式，而且还具有多种工作方式，具有很高的灵活性[33]。由于以上的各种优点，SPI 非常适合集成到 SOC 内部，下面主要对 SPI 展开具体介绍。

4.1.1　SPI 接口信号

SPI 接口只有四个信号用于外部的通信，如图 4.1 所示。

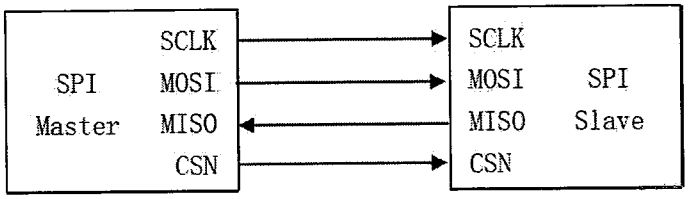

图 4.1 SPI 接口示意图

Figure 4.1 SPI interface diagram

上图 4.1 中 SCLK 信号的时钟频率由 SPI Master 内部的控制寄存器控制，用来同步 SPI Master 和 SPI Slave 之间的数据传输；MOSI 信号即 Master Out Slave In，负责在 SPI Master 的控制下按照串行的方式将数据发送给 SPI Slave；MISO 即 Master In Slave Out，负责信号在 SPI Slave 的控制下按照串行的方式将数据发送给 SPI Master，其中 SPI Slave 在时钟边沿发送数据，SPI Master 在时钟边沿采样数据；CSN 信号作为选择信号，也是受到 SPI Master 内部寄存器的控制，SPI Slave 只有在 CSN 信号为低时才会工作。

SPI 有两种工作状态，分别是主工作状态和从工作状态，由内部控制寄存器控制。当 SPI 工作于主状态时，其负责驱动 SCLK、MOSI 和 CSN 信号，MISO

信号是其输入信号；当 SPI 工作于从状态时，其只负责驱动 MISO 信号，SCLK、MOSI 和 CSN 信号都是其输入信号。

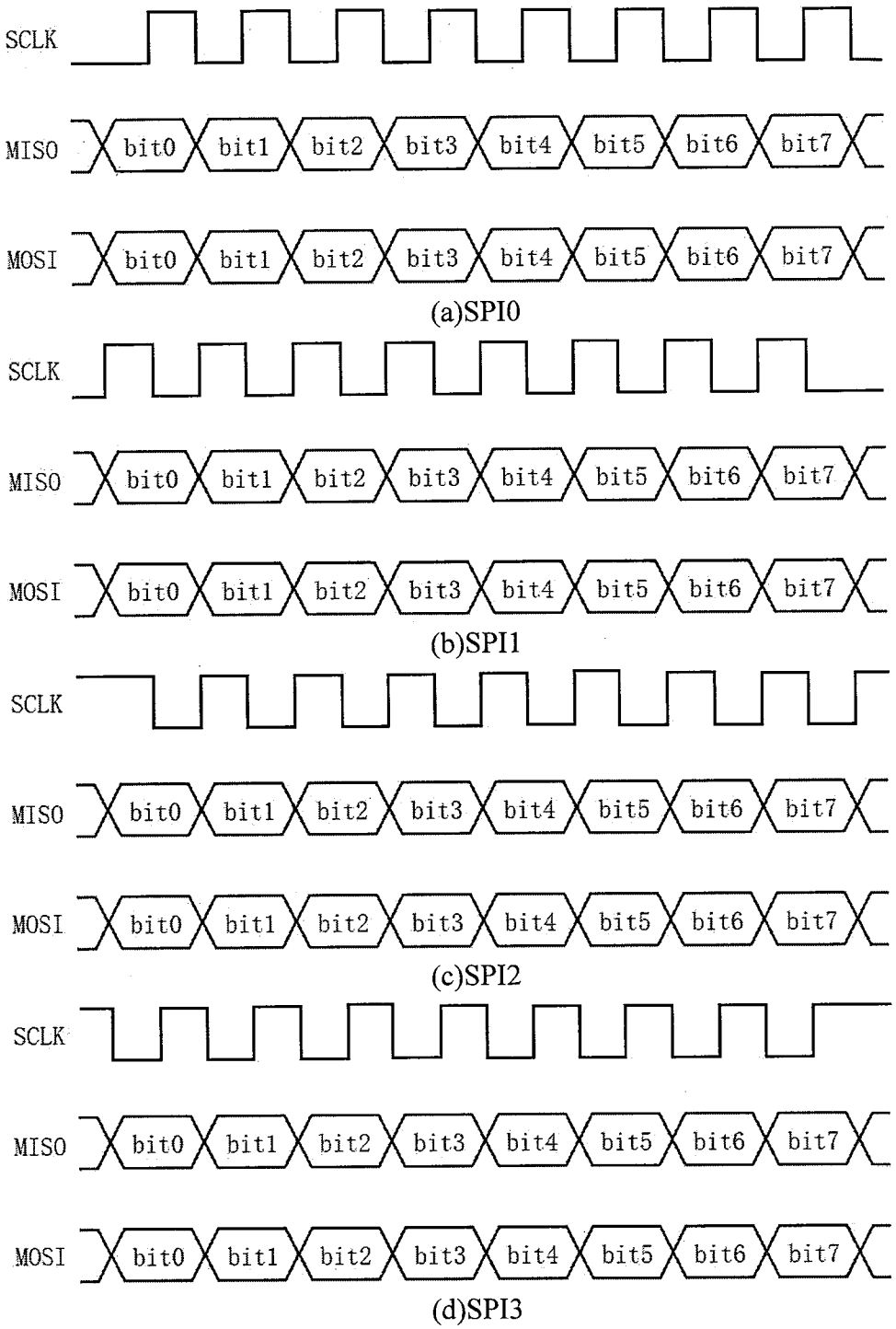

图 4.2 SPI 传输时序

Figure 4.2 SPI transmission timing

SPI 协议有两个要素：时钟极性(CPOL)和时钟相位(CPHA)[34]，这两个因素

的搭配组合使得 SPI 可以有四种工作方式，具体的如下所示：

（1）SPI0：时钟相位和时钟极性都为 0；

（2）SPI1：时钟相位为 1，时钟极性为 0；

（3）SPI2：时钟相位为 0，时钟极性为 1；

（4）SPI3：时钟相位和时钟极性都为 1。

上图 4.2 所示分别为 SPI 四种工作方式的传输时序，其中 SPI0 和 SPI1 的 SCLK 信号在空闲状态下维持低电平，SPI2 和 SPI3 则相反；SPI0 和 SPI2 在第一个时钟边沿采样数据，SPI1 和 SPI3 则相反。

在实际应用中，SPI Master 和 SPI Slave 需要配置为同一种工作方式，即两者在同一个时钟沿驱动数据，在同一个时钟沿采样数据，否则会存在数据传输错误的可能性。

4.1.2　SPI 寄存器

SPI 内部主要有四个 32 位寄存器，分别为 SPI_CTL、SPI_WDATA、SPI_RDATA 和 SPI_STAT。其中 SPI_CTL 寄存器用来配置 SPI 的工作状态，SPI_WDATA 寄存器只使用了低 8 位，用来存放 SPI 要向外发送的数据，SPI_RDATA 寄存器也只使用了低 8 位，用来存放 SPI 接收的数据，SPI_STAT 寄存器用来表示 SPI 当前所处的状态。下面将对其中部分寄存器进行详细介绍。

SPI_CTL 寄存器控制着 SPI 的工作状态，其详细内容如下：

（1）SPI_CTL[15:0]：只在 SPI 被配置为主模式时有效，用来配置 SCLK 信号的输出时钟频率；

（2）SPI_CTL[16]：配置 SPI 工作于主模式或者从模式；

（3）SPI_CTL[17]：配置 SPI 工作时的时钟相位；

（4）SPI_CTL[18]：只在 SPI 被配置为主模式时有效，用来配置 CSN 信号的输出模式；

（5）SPI_CTL[19]：配置 SPI 进行软复位；

（6）SPI_CTL[20]：配置 SPI 进行数据传输时的高低位优先级；

（7）SPI_CTL[21]：配置 SPI 工作时的时钟极性；

（8）SPI_CTL[22]：配置 CSN 信号是否使能；

（9）SPI_CTL[23]：配置 SPI 为三线或四线；

（10）SPI_CTL[25:24]：配置 SPI 输出接口的状态；

（11）SPI_CTL[27:26]：配置 SPI 内部数据接收 FIFO 的中断阈值；

（12）SPI_CTL[28]：复位接收数据 FIFO，其中包括 FIFO 地址指针、字节计数和溢出状态;

（13）SPI_CTL[29]：复位发送数据 FIFO，其中包括 FIFO 地址指针、字节计数和溢出状态;

（14）SPI_CTL[30]：使能接收数据 FIFO；

（15）SPI_CTL[31]：使能发送数据 FIFO。

SPI_STAT 寄存器表示 SPI 当前工作状态，其详细内容如下：

（1）SPI_STAT[0]：表示接收数据 FIFO 内部是否存在数据，当 FIFO 为空时此位置“0”；

（2）SPI_STAT[1]：表示接收数据 FIFO 是否溢出，当 FIFO 为满并且 SPI 内部移位寄存器内存有数据时置“1”；

（3）SPI_STAT[2]：表示发送数据 FIFO 是否溢出，当 FIFO 为满并且 SPI 内部移位寄存器内存有数据时置“1”；

（4）SPI_STAT[4]：表示发送数据 FIFO 空中断是否使能，此位为“1”时使能；

（5）SPI_STAT[5]：表示发送数据 FIFO 是否为空，当 FIFO 为空时此位置“1”；

（6）SPI_STAT[6]：表示接收数据 FIFO 中断是否使能，此位为“1”时使能；

（7）SPI_STAT[7]：表示接收数据 FIFO 内部数据是否达到阈值，当达到阈值时此位置“1”；

（8）SPI_STAT[15:8]：表示接收数据 FIFO 内部字节计数；

（9）SPI_STAT[23:16]：表示发送数据 FIFO 内部字节计数；

（10）SPI_STAT[24]：表示当前 SPI 正在进行数据传输，为只读寄存器；

（11）SPI_STAT[25]：表示从 CSN 信号读取的值，为只读寄存器；

（12）SPI_STAT[26]：表示驱动到 CSN 信号的值，为只读寄存器；

（13）SPI_STAT[27]：表示 SPI 正在等待 CSN 信号拉高，为只读寄存器；

（14）SPI_STAT[30:28]：表示当前正在传输的字节已经读写的比特数，为

只读寄存器；

（15）SPI_STAT[31]：此位被读取时表示当前 SPI 的中断状态，被写“1”时清除 SPI 的中断状态。

4.1.3　功能点提取

功能点即验证工作需要进行验证的点，测试激励的定义就需要以其为指导，除此之外，功能覆盖组和功能覆盖点的定义也需要覆盖所有功能点，并根据仿真结束之后的覆盖情况，以此为依据，对测试激励进行适当的修改。

在实际的验证工作中，功能点的提取需要经过验证人员和设计人员的共同讨论提出，争取考虑到所有需要验证的功能点。本此验证的功能点是基于上文介绍的寄存器提出的，主要有以下几点：

（1）寄存器的读写功能检查；

（2）主从模式基本功能检查；

（3）SPI 四种工作方式检查；

（4）数据传输时高位或低位优先功能检查；

（5）CSN 信号输出方式检查；

（6）接收和发送数据 FIFO 使能检查；

（7）三线模式检查。

4.2　验证平台层次设计

本论文所搭建的验证平台可以模拟待测设计的真实工作环境，为其提供适当激励，不仅每次仿真结束后可以自动比对输出结果，而且可以通过对激励信息进行采样衡量目前项目进度。

功能模块化是 UVM 验证方法学的优势之一。在搭建验证平台之前，首先应该根据功能将整个验证平台模块化，如图 4.3 所示，然后运用自顶向下的方法完成对底层组件的定义。

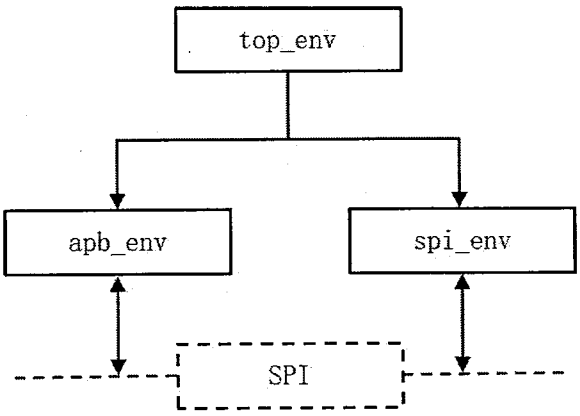

图 4.3 SPI 验证平台层次结构

Figure 4.3 SPI verification platform hierarchy

apb_env：模拟 apb 总线的行为，完成对 SPI 内部寄存器的读写操作，为 SPI 提供激励信息，同时可以将激励信息通过 TLM 通信机制发送给验证平台中的其他组件。

spi_env：模拟 SPI Master 或者 SPI Slave 的行为，当 SPI 被配置为主模式时模拟 SPI Slave 的行为，完成对 MOSI 信号的采样，同时驱动 MISO 信号；当 SPI 被配置为从模式时模拟 SPI Master 的行为，完成对 MISO 信号的采样，同时驱动 SCLK 信号、MOSI 信号和 CSN 信号。

top_env：内部包括计分板和功能覆盖率信息采集等组件，用于判断数据收发的正确性，同时为衡量项目进度提供依据。

4.3 验证组件设计

验证平台由众多验证组件构成，各组件之间高效配合，才能的更有效率地完成验证工作，所以对平台中各验证组件的设计至关重要。

本节运用层次化的结构设计方法，以上文介绍的结构框图为指导，基于 UVM 验证方法学搭建了如下图 4.4 所示的 SPI 验证平台，下面将对验证平台内部各组件的设计过程进行详细介绍。

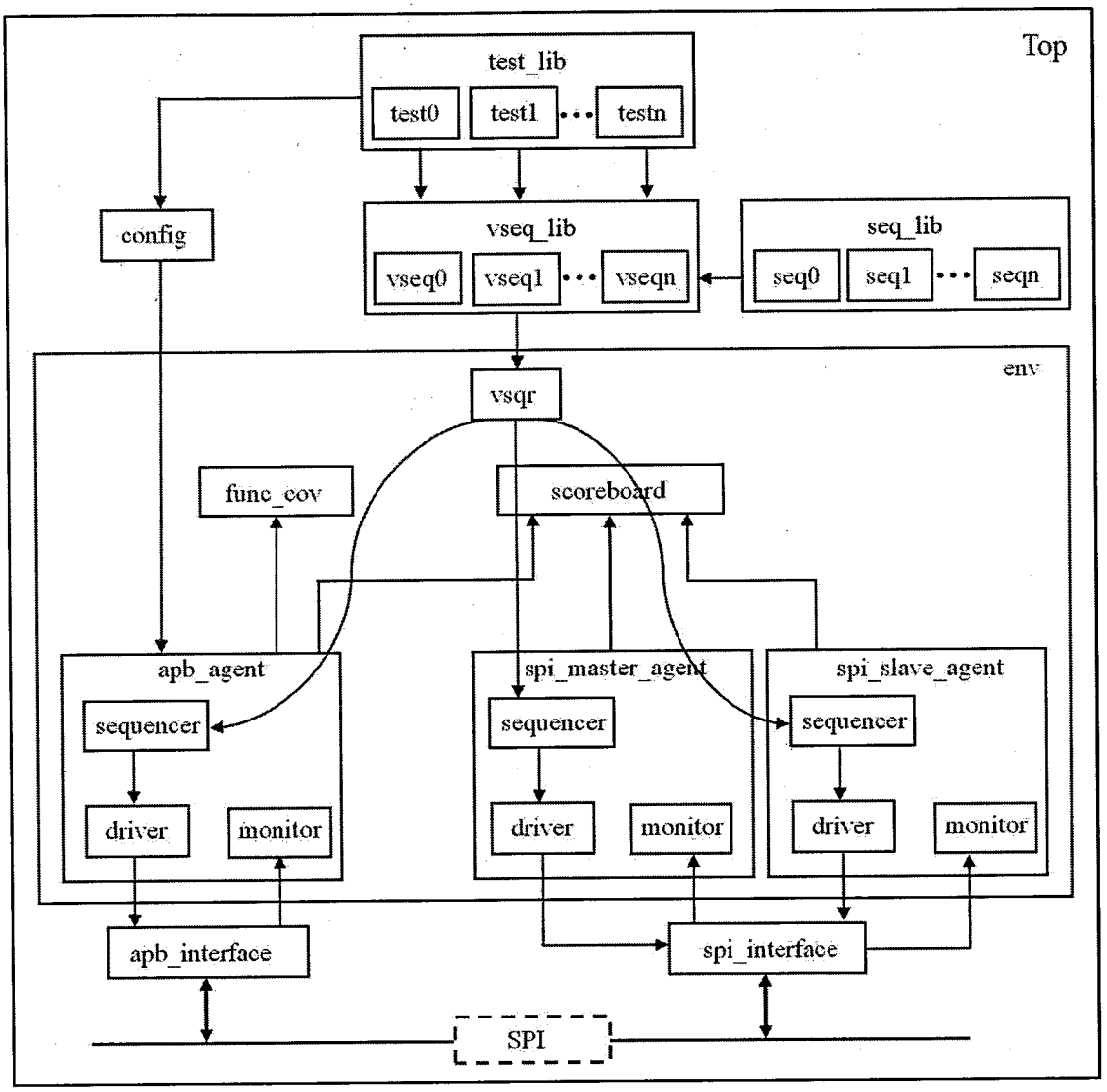

图 4.4 SPI 验证平台结构图

Figure 4.4 SPI verification platform structure diagram

4.3.1　事务 transaction

在待测设计的各个 module 之间，信息传递是以单笔数据为单位进行的，而在验证平台中，各组件之间的通信都是以 transaction 为单位进行的，每个 transaction 内部都封装了多笔数据，不同的 transaction 代表了不同的传输协议[35]。transaction 就像是一个数据包，将传输协议中的数据抽象出来封装成一个整体，有利于对 transaction 进行重用。

在典型 UVM 验证平台中 transaction 的传输路径如图 4.5 所示。sequence 在生成 transaction 之后会发送给 sequencer，driver 在需要 transaction 时会向 sequencer

提出请求，sequencer 在中间起到一个指挥的作用。monitor 与 reference model、reference model 与 scoreboard、monitor 与 scoreboard 之间的信息传递都是基于 transaction 进行的，如果验证平台中存在功能覆盖率收集模块，其数据也是来自于 monitor 发送的 transaction。

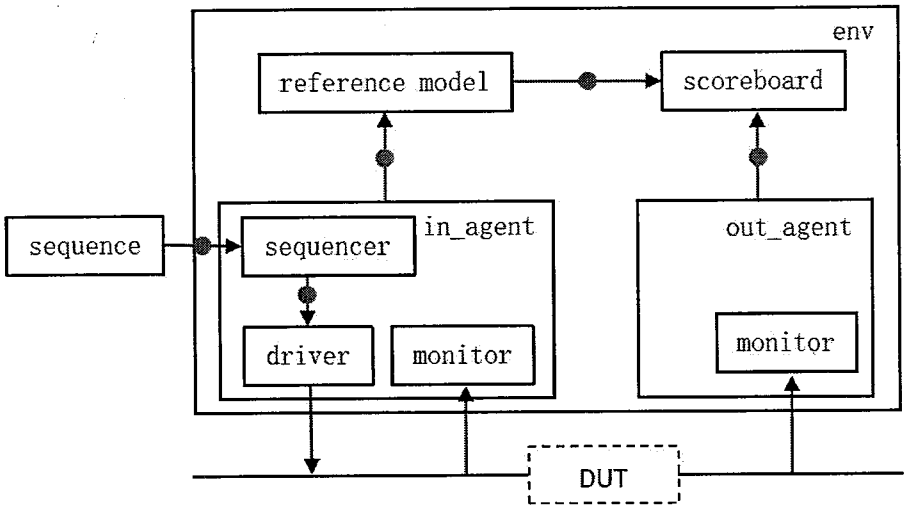

图 4.5 transaction 传输路径

Figure 4.5 transaction transmission path

在本次验证平台的搭建过程中，一共定义了两个不同的 transaction，一个是抽象自 APB 协议的 apb_transaction，一个是抽象自 SPI 协议的 spi_transaction。其中 apb_transaction 的定义如下：

```
class apb_transaction extends uvm_sequence_item;
rand bit [31:0]              addr;
  rand bit                   direction;
  rand bit [31:0]            data;
  `uvm_object_utils_begin(apb_transaction)
    `uvm_field_int(addr, UVM_DEFAULT)
    `uvm_field_int(direction, UVM_DEFAULT)
    `uvm_field_int(data, UVM_DEFAULT)
  `uvm_object_utils_end
  function new (string name = "apb_transaction");
    super.new(name);
```

```
    endfunction
endclass : apb_transaction
```

其中 addr、direction 和 data 表示 APB 协议中最基本的信号，被声明为 rand 类型是因为只有 rand 类型的变量才能被 randmize 函数赋随机值，然后使用 uvm_object_utils 对三个变量进行注册，注册过后就可以使用 copy、prepare 和 print 等函数而无需自己定义。new 函数为构造函数，用于实例化一个 transaction 的对象。

spi_transaction 的定义与 apb_transaction 的内容相似，只是具体的 rand 类型变量不同，故此处不再过多介绍。

4.3.2　接口 interface

传统的验证方法中，验证环境与待测设计之间都是基于信号进行连接的，在运用 UVM 验证方法学搭建验证平台之后，与待测设计之间的连接可以通过 interface 来完成[36]。根据接口协议的不同，interface 内部声明的信号也不同。

在本次搭建验证平台的过程中，一共定义了两个不同的 interface，一个是用于 apb_agent 与待测设计进行连接的 apb_interface，一个是用于 spi_agent 与待测设计进行连接的 spi_interface，两者内容非常类似，只是具体信号不同。其中 apb_interface 内部声明的即 apb 协议的内部信号，如下所示：

```
logic [31:0]   paddr;
logic          prwd;
logic [31:0]   pwdata;
logic [31:0]   prdata;
logic          penable;
logic          psel;
logic          pready;
```

使用 interface 最大的优势就是可以使连接变得更加简单，如果需要在接口内添加新的信号，只需要修改 interface 的定义及使用这个 interface 的模块即可，极大地降低了因修改连接信号而出错的概率。

在使用 interface 时，需要在 top_tb 中通过 config_db 机制将 interface 传送给验证平台内部组件。在本验证平台中，需要将 apb_interface 传送给 apb_agent 内

的 driver 和 monitor 组件，由于平台内存在两个 spi_agent，所以需要为两个 spi_agent 内的 driver 和 monitor 组件分别传送 spi_interface。

4.3.3 序列 sequence

sequence 在验证平台中被用来产生 transaction，每个 sequence 内部可以生成一个或多个 transaction，这些 transaction 可以通过 uvm_do_on 宏被发送给同一个或者多个不同的 sequencer。

在本验证平台的搭建过程中，为了生成不同的 transaction 定义了多个不同的 sequence，其中有的用来生成 SPI 内部寄存器的配置信息，有的用来生成 SPI 向外发送的数据，有的用来生成 SPI 从外部接收的数据。在 sequence 的定义过程中应用了继承，其继承关系如图 4.6 所示。

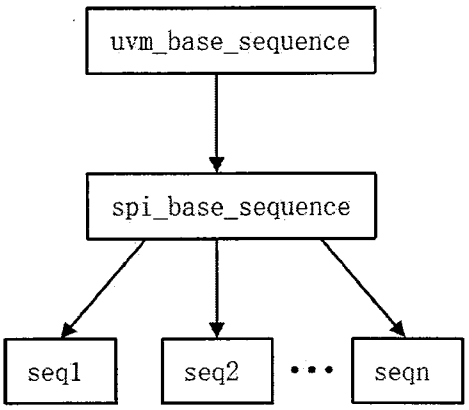

图 4.6 sequence 的继承

Figure 4.6 Inheritance of sequence

其中 uvm_base_sequence 中定义了一些通用的 task，比如 apb_write_data 和 apb_read_data 等，以此为基类，继承出 spi_base_sequence 派生类，在内部定义一些专用于 SPI 验证的 task，并由此继承出多个 sequence，在内部通过调用上述 task 产生符合要求的 transaction，最后形成一个 sequence 库。通过继承，使得 uvm_base_sequence 可以在不同的验证平台中重用。上述 sequence 内部定义如下所示：

```
class uvm_base_sequence extends uvm_sequence #(apb_transaction);
...
extern task apb_read(input [31:0] ad, output [31:0] rd);
```

```
extern task apb_write(input [31:0] ad, input [31:0] wd);
...
endclass:uvm_base_sequence
class spi_base_sequence extends uvm_base_sequence;
...
   extern task apb_write_item();
...
endclass : spi_base_sequence
class sequence1 extends spi_base_sequence;
...
apb_write_item();
...
endclass : sequence1
```

为了进一步提高验证平台的可重用性，此处借鉴软件设计模式中的外观模式，引入 virtual sequence 完成对其它 sequence 的调用，在验证平台中启动 sequence 时实际上启动的是 virtual sequence。virtual sequence 与 sequence 的调用关系如图 4.7 所示。

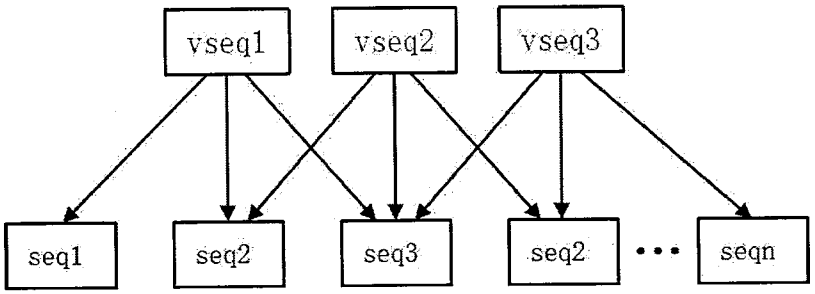

图 4.7 vseq 与 seq 调用关系图

Figure 4.7 vseq and seq call relationship diagram

在本验证平台中，每个 virtual sequence 内部调用了 3 个 sequence，分别用来生成配置信息、发送数据和接收数据，其中生成配置信息的 sequence 被首先启动，由于 SPI 发送和接收数据是同步的，所以另外两个 sequence 通过 fork…join_any 同时启动，其内部具体实现细节如下所示：

```
class spi_master_vsequence extends uvm_sequence;
```

```
  apb_config_seq          apb_config_seq;
  apb_transfer_seq        apb_transfer_seq;
  spi_transfer_seq        spi_transfer_seq;
...
  task body();
...
`uvm_do_on(apb_config_seq, p_sequencer.apb_master_sqr)
...
  fork
    `uvm_do_on(apb_transfer_seq, p_sequencer.apb_master_sqr)
    `uvm_do_on(spi_transfer_seq, p_sequencer.spi_slave_sqr)
join_any
...
  endtask
endclass:spi_master_vsequence
```

4.3.4 序列器 sequencer

sequencer 在验证平台中主要负责 driver 和 sequence 之间 transaction 的调配，其内部定义非常简单，通常只需要继承自 uvm_sequencer 基类，然后使用 uvm_component_utils 注册一下，再通过两个 function 执行一下父类的构造函数和 build_phase 即可。

本验证平台中一共定义了 3 个 sequencer：apb_sequencer、spi_master_sequencer 和 spi_slave_sequencer，分别与前文介绍的 3 类 sequence 相对应，负责将 transaction 通过 TLM 通信机制发送给相应的 driver。

为了配合 virtual sequence，此验证平台中引入了 virtual sequencer，其内部定义与一般的 sequencer 相似，只是增加了对验证平台中其他 sequencer 的声明。

4.3.5 驱动器 driver

driver 在验证平台中主要负责将接收到的 transaction 按照相应协议的时序，通过接口驱动给待测设计[37]。本验证平台中一共定义了 3 个 driver：apb_driver、spi_master_driver 和 spi_slave_driver，其中 apb_driver 负责驱动 apb 协议信号，在

整个仿真流程中一直工作，另外两个 driver 负责驱动 spi 协议信号，分别在 SPI 被配置为主或从模式时工作。

上述 3 个 driver 内部主要定义了两个 phase：build_phase 和 run_phase。build_phase 内部主要是通过 config_db 机制完成对一些参数的接收，其中包括对接口的接收，而 run_phase 则是 3 个 driver 的主要不同之处。

在 apb_driver 内部的 run_phase 中，调用了一个任务：get_and_drive，在此任务中完成对 transaction 的请求及驱动。为了提高验证平台的可重用性，此处借鉴软件设计模式中的模板方法模式，在驱动 transaction 之前引入 callback 机制，这就使得可以在驱动 apb 总线信号之前，按照不同的需求对 transaction 做出不同的修改，而无需对验证平台中的其他模块进行修改。其主要内容如下所示：

```
class apb_driver extends uvm_driver #(apb_transaction);
...
task run_phase(uvm_phase phase);
    get_and_drive();
endtask : run_phase
task get_and_drive();
  while (1) begin
    forever begin
      @(posedge vif.pclk iff (vif.preset))
      seq_item_port.get_next_item(req);
      `uvm_do_callbacks(apb_driver, A, pre_tran(this, req))
      drive_transfer(req);
      seq_item_port.item_done();
    end
  end
endtask : get_and_drive
...
endclass:apb_driver
```

只要有 transaction 被传送过来，apb_driver 就要将其驱动给待测设计，所以

此处使用了 while(1)循环，seq_item_port 与 apb_sequencer 的端口相连，用于接收 transaction，在驱动数据之前，只要按需求重定义 pre_tran 的内容，就可以实现对 transaction 内容的修改。

spi_master_driver 和 spi_slave_driver 的内容和 apb_driver 的内容相似，只是具体的驱动任务会因协议不同而存在差别。其中 spi_master_driver 只在 SPI 被配置为从模式时工作，负责驱动 SCLK、MOSI 和 CSN 信号，spi_slave_driver 只在 SPI 被配置为主模式时工作，负责驱动 MISO 信号。其内部具体内容的定义不在此处做过多介绍。

4.3.6 监视器 monitor

monitor 在验证平台中主要负责将接口信号按照具体协议重新打包成 transaction 的形式，然后将其发送给平台中的其他组件[38]。本验证平台中一共定义了 3 个 monitor：apb_monitor、spi_master_monitor 和 spi_slave_monitor，其中 apb_monitor 负责打包 apb 接口信号，在整个仿真流程中一直工作，另外两个 monitor 负责打包 spi 接口信号，分别在 SPI 被配置为主或从模式时工作。

和 driver 类似，上述 3 个 monitor 内部也定义了两个 phase：build_phase 和 run_phase。build_phase 内部除了要通过 config_db 机制完成对一些参数及接口的接收之外，还需要调用构造函数完成对 analysis_port 端口的实例化，在 run_phase 中打包 transaction 之后通过此端口向外发送。在 run_phase 中，主要就是调用了一个任务：collect_transactions，在此任务中完成对接口信息的收集，同时负责向外发送。其内部主要内容如下所示：

```
class apb_monitor extends uvm_monitor;
  uvm_analysis_port #(apb_transaction) ap;
...
task run_phase(uvm_phase phase);
    collect_transactions();
endtask : run_phase
task collect_transactions();
apb_transaction tr;
  forever begin
```

```
    …   //do something to collect transaction
    ap.write(tr);
  end
endtask : collect_transactions
endclass: apb_monitor
```

monitor 有向外传输数据的需求，所以声明了一个 uvm_analysis_port 类型的接口。同 apb_driver 一样，monitor 在整个仿真过程中都工作，所以此处使用了 forever 语句，在收集完一笔 transaction 之后，调用端口的 write 函数将 transaction 发送给平台中的其他组件。

spi_master_monitor 和 spi_slave_monitor 的内容和 apb_monitor 的内容相似，只是在收集 transaction 时会根据协议不同而存在差别。其中 spi_master_monitor 只在 SPI 被配置为从模式时工作，spi_slave_monitor 只在 SPI 被配置为主模式时工作。其内部具体内容的定义不在此处做过多介绍。

4.3.7　代理器 agent

agent 在验证平台中主要起到封装的作用[39]，一般封装有 driver、monitor 和 sequencer 三个组件，其内部存在一个成员变量：is_active，在实际应用中设置此变量的值，可以选择性的对组件实例化。本验证平台中一共定义了 3 个 agent：apb_agent、spi_master_agent 和 spi_slave_agent，其中 apb_agent 负责处理 apb 接口的相关信号，在整个仿真流程中一直工作，另外两个 agent 负责处理 spi 接口的相关信号，分别在 SPI 被配置为主或从模式时工作。

上述 3 个 agent 内部均封装有三个组件，只是组件的具体内容不同，其内部主要定义了两个 phase：build_phase 和 connect_phase。build_phase 内部主要完成驱动器、监视器和序列器三个组件的实例化工作，connect_phase 完成内部封装的三个组件中各端口的连接工作。其内部主要内容如下所示：

```
class apb_agent extends uvm_agent;
  apb_sequencer apb_sqr;
  apb_driver    apb_drv;
  apb_monitor   apb_mon;
  …
```

```
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        apb_sqr = apb_sequencer::type_id::create("apb_sqr",this);
        ...
    endfunction : build_phase
    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        apb_drv.seq_item_port.connect(apb_sqr.seq_item_export);
        ap=apb_mon.ap;
    endfunction : connect_phase
    endclass : apb_agent
```

由于内部三个组件都使用 uvm_component_utils 宏注册过，所以在 build_phase 中要使用 type_id::create 来进行实例化，在 connect_phase 中通过 connect 将 apb_drv 和 apb_sqr 的端口连接起来，同时将 agent 内部端口指针指向 apb_mon 内部的端口。这里借鉴了软件设计模式中的中介者模式，使用 agent 将多个组件封装在一起，内部组件与外部组件之间的通信全部通过 agent 组件来完成，这就降低了验证平台组件之间的耦合，提高了其可重用性。

spi_master_agent 和 spi_slave_agent 的作用及内部定义都与 apb_agent 类似，故不再对其做具体介绍。

4.3.8 计分板 scoreboard

scoreboard 主要负责对每次仿真结束之后的仿真结果进行比较，比对的数据一部分是预期数据，一部分是实际数据[40]。在一般的验证平台中，预期数据来自于参考模型，但此次验证的待测设计只负责数据传输，功能并不复杂，所以并没有引入参考模型，预期数据直接由 monitor 在输入端口进行收集。

结果比对一般有两种方式，分别是即时比对和结果比对，当仿真时间比较长时采用即时比对方式可以在一定程度上提高验证效率，但本次验证所用 test 仿真时间比较短，所以采用的是结果比对。其内部主要定义了一个 run_phase，在 run_phase 中完成对 transaction 的保存及比对工作。其内部主要内容如下所示：

```
class spi_scoreboard extends uvm_scoreboard;
```

```
    uvm_blocking_get_port #(apb_transaction) apb_port;
...
    spi_transaction spi_send_data1[$];
...
task run_phase(uvm_phase phase);
...
        fork
          while(1) begin
            apb_port.get(apb_tr);
            write_master_apb_ap(apb_tr);
        ...
          end
          while(1) begin
            spi_slave_port.get(spi_slave_tr);
            write_master_spi_ap(spi_slave_tr);
          end
          while(1) begin
            spi_master_port.get(spi_master_tr);
            write_slave_spi_ap(spi_master_tr);
          end
        join
endtask : run_phase
...
endclass : spi_scoreboard
```

由于 SPI 有主从两种工作模式，所以需要声明 3 个端口，分别用来接收 apb_monitor、spi_master_monitor 和 spi_slave_monitor 发送的 transaction，为了保存所有 SPI 发出的数据和接收的数据，一共声明了六个队列。在 run_phase 中通过 fork...join 同时启动三个 while 循环，每一个 while 循环内部接收一个 monitor 发送的 transaction 并进行保存，在数据传输结束之后调用其他自定义任务完成比

对工作。

4.3.9 覆盖率 coverage

覆盖率分为功能覆盖率和代码覆盖率，其中代码覆盖率只需要在命令行添加适当命令就可以完成收集，而功能覆盖率就必须自定义覆盖率信息采样组件才能完成收集[41]。最后覆盖率的统计结果作为修改测试激励的指导，同时也作为衡量验证工作进度的标准。

本验证平台中负责功能覆盖率信息收集的组件为 func_cov，其内部定义有覆盖组(cover_group)和覆盖点(cover_point)，要求覆盖所有功能点，之后在 run_phase 中调用其他任务完成采样工作。其内部主要内容如下所示：

```
class spi_func_cov extends uvm_component;
  uvm_blocking_get_port #(apb_transaction) apb_port;
...
  task run_phase (uvm_phase phase);
    apb_transaction tr;
    while(1) begin
      apb_port.get(tr);
      sample_spi_register(tr);
    end
  endtask
  function void sample_spi_register(apb_transaction tr);
...
    cov_spi_ctrl.sample();
  endfunction
...
endclass
```

func_cov 需要与 apb_monitor 进行数通通信，所以需要声明一个端口用于接收 transaction。在 run_phase 中使用了一个 while 循环，在循环内部调用端口的 get 函数，因为 get 具有阻塞性质，所以当没有接收到事务时程序会阻塞在该语句，当该端口 get 到一笔 transaction 时会调用 sample_spi_register 任务，在该任务

中再调用覆盖组 cov_spi_ctrl 的 sample 函数完成最终的采样工作，覆盖组中存在多个覆盖点，每个覆盖点包含多个仓(bin)，被覆盖的仓与仓总数的比值即为功能覆盖率。

4.3.10　环境 env

env 的存在便于对整个验证平台进行管理，在其内部主要完成验证平台各组件的实例化工作，以及完成对各组件必要的连接工作[42]。其内部主要内容如下所示：

```
class spi_env extends uvm_env;
    uvm_tlm_analysis_fifo #(apb_transaction) apb_master_scb_fifo;
…
    function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      apb_master=apb_agent::type_id::create("apb_master",this);
…
      apb_master_scb_fifo=new("apb_master_scb_fifo",this);
…
    endfunction
    function void connect_phase(uvm_phase phase);
      super.connect_phase(phase);
      spi_vsqr.apb_sqr = apb_master.apb_sqr;
…
     apb_master.ap.connect(apb_master_scb_fifo.analysis_export);
…
   endfunction
endclass:spi_env
```

验证平台中 monitor、scoreboard 和 func_cov 之间的通信都是通过 TLM 机制完成的，此处选择 FIFO 的方式，一共声明了四个 FIFO 用于上述组件的通信，然后在 build_phase 中完成对其他组件和四个 FIFO 的实例化，在 connect_phase 中完成四个 FIFO 与监视器等组件的端口连接，同时也完成了 virtual sequencer

中三个 sequencer 的指针与验证平台中三个 sequencer 的连接。

4.3.11 寄存器模型 reg_model

待测设计内往往存在大量的寄存器，如果手动编写寄存器模型的话需要花费大量的时间，所以在实际的验证工作中，一般都是使用脚本文件来完成寄存器模型的定义[43]。

在对寄存器模型进行读写访问时，都会通过 sequence 机制生成一个类似于 transaction 类型的 uvm_reg_bus_op 变量，寄存器模型只能识别此种类型的数据，但是验证平台中的其他组件只能处理 apb_transaction 或 spi_transaction 类型的数据，所以这里借鉴软件设计模式中的适配器模式，引入了一个转化器(adapter)，该转换器可以将 uvm_reg_bus_op 变量转换成目标组件可以处理的类型，同时也可以反方向转换。

内部主要定义了两个 function：reg2bus 和 bus2reg，在对寄存器模型进行读写访问操作时，reg2bus 可以将 uvm_reg_bus_op 变量转换成 apb_transaction 类型，bus2reg 可以将 apb_transaction 类型转换成 uvm_reg_bus_op 变量。

4.4 验证平台运行

在验证平台搭建完成之后，就需要根据功能点编写测试激励，这个过程中需要用到 sequence 机制，之后运行所有的 test case，以期能够覆盖所有的功能点。仿真结束之后，根据内部打印信息查看仿真结果或者根据打印内容进行 debug，最后查看覆盖率信息，判断此次验证工作的完备性。下面将对上述整个流程作一个详细介绍。

4.4.1 构建测试用例

为了验证所有的功能点，一共构建了九个测试用例，这里应用了面向对象语言三大特征之一的继承，即首先定义了一个 base_test 基类，在此基类中完成一些公共的设置，然后基于此类派生出其他测试用例。平台中测试用例介绍如下：

（1）spi_master_test：此 test 作为冒烟测试，只验证 SPI 最基本的功能，证明待测设计没有非常明显的错误；

（2）spi_master_burst_test：此 test 测试 SPI 的 burst 传输功能，此时 SPI 被配置为只有当其内部发送数据的 FIFO 为空时，CSN 信号才会被拉高；

（3）spi_master_disfifo_test：此 test 测试 SPI 内部的接收数据 FIFO 和发送数据 FIFO 能否正常使能与关闭使能；

（4）spi_master_nocsn_test：此 test 测试 SPI 不使用 CSN 作为选择信号时，数据传输能否正常进行；

（5）spi_master_3wire_test：此 test 测试 SPI 的三线模式能否正常工作；

（6）spi_master_3burst_test：此 test 测试 SPI 被配置为三线模式时，其 burst 传输模式能否正常运行；

（7）spi_slave_test：此 test 测试 SPI 被配置为从模式时能否正常工作；

（8）spi_slave_burst_test：此 test 测试 SPI 被配置为从模式时，其 burst 传输模式能否正常运行；

（9）spi_slave_3wire_test：此 test 测试 SPI 被配置为从模式时，其三线模式能否正常运行。

与上述 test 对应，平台中也定义了九个 virtual sequence，每个 test 内部都会将一个 virtual sequence 设置为 default_sequence，然后通过嵌套 sequence 的方式[44]，由 virtual sequence 从 sequence 库中选择合适的 sequence，并使用前文介绍的 fork...join 的方式对其进行同步。

对于寄存器功能的验证可以直接使用前文介绍的 UVM 中内建 sequence 来完成，而无需构建专门的测试用例。

4.4.2　运行测试用例

整个验证平台通过在 tb_top 中调用 run_test 任务启动，该任务有一个字符串类型的参数，验证平台根据此参数可以运行相应的测试用例[45]，为了提高验证平台的灵活性，在实际应用中并不直接为 run_test 任务提供参数，一般都是使用 UVM_TESTNAME 为其规定测试用例的名称[46]。

为了提高验证工作的自动化程度，整个仿真过程都通过脚本文件来完成。仿真和编译过程使用 Makefile 脚本完成，在其中添加收集代码覆盖率信息的命令。之后编写一个 shell 脚本，在 shell 脚本中使用 make 指令，同时指定测试用例名和种子进行仿真，一个测试用例仿真结束之后直接开始下一次仿真，通过此脚本文件一次性完成九个测试用例的仿真，最后将九次仿真的覆盖率数据 merge 到一起形成一个 vdb 文件[47]，方便以后查看覆盖率信息。

本次验证工作选择 VCS 作为仿真工具，使用 DVE 查看波形和覆盖率信息。

在运行测试用例时，只有第一个测试用例需要按照先编译后仿真的顺序进行，其余八个测试用例沿用第一次的编译结果，在运行时只需修改 UVM_TESTNAME 参数即可进行仿真。

4.4.3 仿真结果

上述九个测试用例每运行结束一个，都会生成一个.log 的记录文件，内部会记录此次仿真过程的信息，其中包括 UVM_INFO、UVM_WARNING、UVM_ERROR 和 UVM_FATAL 等信息，而且还会显示上述信息具体来自验证平台中哪一组件。

```
UVM_INFO ../verif/spi_master_vsequence.sv(20) @ 243750:
uvm_test_top.env.spi_vsqr@@spi_master_vsequence [spi_master_vsequence]
Configuration is done!
UVM_INFO ../verif/spi_scoreboard.sv(118) @ 35668750: uvm_test_top.env.scb
[scb] Write finish data
UVM_INFO ../verif/spi_scoreboard.sv(265) @ 35668750: uvm_test_top.env.scb
[scb] APB compare SUCCESSFULLY
UVM_INFO ../verif/spi_scoreboard.sv(202) @ 35668750: uvm_test_top.env.scb
[scb] SPI compare SUCCESSFULLY
UVM_INFO /home/uncle_root/Desktop/synopsys_softwares/vcs2016.06/etc/uvm-1.1/
base/uvm_objection.svh(1267) @ 1035668750: reporter [TEST_DONE] 'run' phase
is ready to proceed to the 'extract' phase
UVM_INFO ../verif/base_test.sv(75) @ 1035668750: uvm_test_top
[spi_master_test] this test case passed!

--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :    7
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
[TEST_DONE]     1
[scb]     3
[spi_master_test]     1
[spi_master_vsequence]     1
```

图 4.8 SPI 仿真记录文件

Figure 4.8 SPI simulation record file

图 4.8 为测试用例 spi_master_test 的仿真记录文件，从中可以看出本次仿真产生了七个 UVM_INFO 信息，没有产生 UVM_WARNING、UVM_ERROR 和 UVM_FATAL 信息，七个 UVM_INFO 信息中有三个来自 scoreboard 组件，一个

来自 spi_master_test 组件，还有一个来自 spi_master_vsequence。其中第一个来自 spi_master_vsequence 的信息表示当前已经完成对 SPI 内部寄存器的配置，接下来三个来自 scoreboard 的信息中，第一个表示当前已经完成数据传输任务，可以开始结果比对，第二个表示通过 SPI 向外发送的数据比对成功，第三个表示通过 SPI 接收的数据比对成功，最后一条来自 spi_master_test 的信息表示本次仿真没有发生任何错误。

图 4.9 为测试用例 spi_master_test 的仿真波形，从中可以看出首先进行的是对 SPI 内部地址为 32'h00 和 32'h0c 的 SPI_CTL 和 SPI_STAT 寄存器进行配置，其配置数据由前文介绍的 virtual sequence 中三个 sequence 的第一个产生；接下来对地址为 32'h04 的 SPI_WDATA 寄存器进行写操作，当存在字节被写入完成之后，SPI 接口开始将该字节以串行的方式通过 sig_mosi 向外发送；写完 4 字节数据之后开始对地址为 32'h0c 的 SPI_STAT 寄存器进行读操作，当读取到 SPI_STAT[31]为“1”，即因为发送数据 FIFO 为空而产生中断时，对此中断标志位写“1”清零，然后对地址为 32'h04 的 SPI_RDATA 寄存器进行读操作，用来将 SPI 接收的 4 字节数据读取出来，之后继续向 SPI_WDATA 寄存器内写入数据，并重复上面的过程。

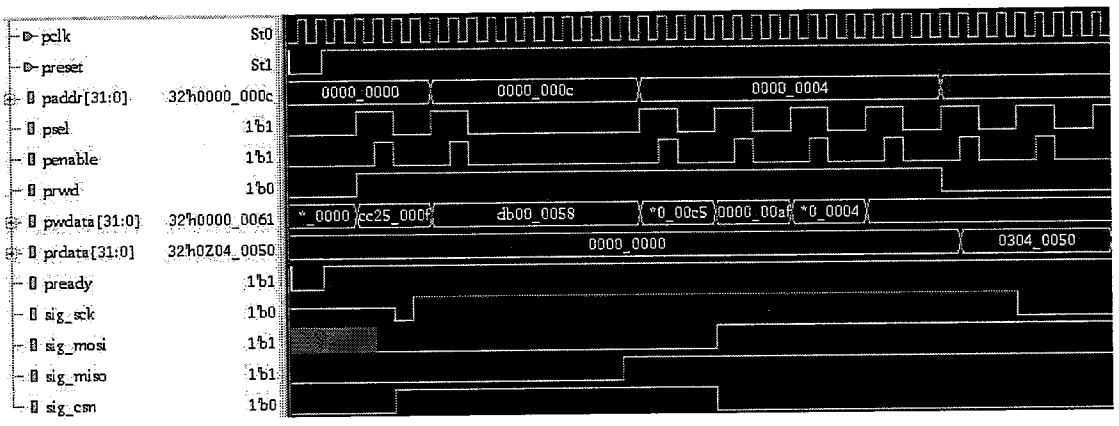

图 4.9 spi_master_test 仿真波形图

Figure 4.9 spi_master_test simulation waveform

通过脚本文件完成九个测试用例的仿真并且没有发生错误之后，就可以通过 DVE 查看最后的覆盖率信息，如果覆盖率信息还没有达到要求，则可以考虑继续多次运行以上测试用例，或者可以构建定向测试用例。

下图 4.10 所示为仿真结束后的代码覆盖率情况，其中第一列为总的代码覆

盖率信息。从图中可以看出，本次验证工作只有行覆盖率和分支覆盖率没有达到100%，其余的翻转覆盖率、状态及覆盖率以及条件覆盖率这三个方面均达到了100%覆盖率。

Name	Score	Line	Toggle	FSM	Condition	Branch
tb_top	99.60%	99.31%	100.00%	100.00%	100.00%	98.72%
spi_vif_s	100.00%		100.00%			
spi_vif	100.00%		100.00%			
regi_spi_inst	99.56%	99.23%	100.00%	100.00%	100.00%	98.57%
apb_vif	100.00%		100.00%			

图 4.10 SPI 验证代码覆盖率

Figure 4.10 SPI verification code coverage

下图 4.11 所示为仿真结束后的功能覆盖率情况，第一行表示总的覆盖率信息，中间部分表示覆盖点信息，后面的部分表示交叉覆盖率信息。其中覆盖点信息表示的是单个变量所有可能值出现的情况，确保所有感兴趣的值被观测到，而交叉覆盖率信息表示的是多个覆盖点各种交叉组合出现的情况。从图中可以看出，本次验证工作所有的覆盖点和交叉覆盖都达到了 100%覆盖率。

Cover Group Item	Definition	Score	Goal
tb_top.me.obj...	$unit::spi...	100.00%	100%
cov_active...		100.00%	100%
cov_bidire...		100.00%	100%
cov_clk_di...		100.00%	100%
cov_inacti...		100.00%	100%
cov_maste...		100.00%	100%
cov_maste...		100.00%	100%
cov_rx_fif...		100.00%	100%
cov_rx_tri...		100.00%	100%
cov_soft_r...		100.00%	100%
cov_spi_ba...		100.00%	100%
cov_tx_fifo...		100.00%	100%
cov_use_rd...		100.00%	100%
spi_master		100.00%	100%
spi_master...		100.00%	100%
spi_master...		100.00%	100%
spi_master...		100.00%	100%
spi_master...		100.00%	100%
spi_master...		100.00%	100%
spi_slave		100.00%	100%
spi_slave_...		100.00%	100%
spi_slave_...		100.00%	100%

图 4.11 SPI 验证功能覆盖率

Figure 4.11 SPI verification function coverage

4.5　重用性比较

下表 4.1 所示为本文所搭建的验证平台内部组件与传统验证平台内部组件的可重用性比较，从中可以看出，本论文所搭建的验证平台内部组件的确比传统的验证平台具有更高的可重用性。

表 4.1 重用性比对

Table 4.1 Reusability comparison

验证组件	重用性对比	
	本文验证平台	传统验证平台
env	✔	
apb_agent	✔	
apb_driver	✔	
apb_monitor	✔	✔
apb_sequencer	✔	✔
apb_interface	✔	✔
sequence	✔	
adapter	✔	
apb_transaction	✔	✔
virtual sequencer	✔	

注：“✔”表示可重用性更高。

其中 env 需要根据不同的待测设计作适当的修改，apb_agent 借鉴中介者模式降低了组件之间的耦合度，apb_driver 借鉴模板方法模式在内部引入了 callback 机制，进一步提高了可重用性，sequence 借鉴了外观模式，通过 virtual sequence 的调用生成不同的激励，adapter 则借鉴适配器模式，实现了不同 transaction 之间的转换。

第 5 章　验证平台可重用性应用

在搭建完 SPI 的验证平台之后，为了证明其可重用性，需要对其进行应用。本章将要完成 UART 验证平台的搭建工作，过程中会重用前文介绍的 SPI 验证平台内部组件，从而使得验证平台的高可重用性得到证明。

5.1　UART 协议介绍

UART 的传输速度要慢于 SPI，但其具有通信线路简单的特点，非常适合应用于传输距离远但对速度要求不高的场合，而且 UART 可以实现将并行输入转换成串行输出，所以其应用范围也非常广[48]。下面主要对 UART 展开具体介绍。

5.1.1　UART 接口信号

UART 接口只有两个信号用于外部的通信，如图 5.1 所示，两个信号分别为 TX 和 RX，其中 TX 负责向外发送数据，RX 负责向内接收数据[49]。

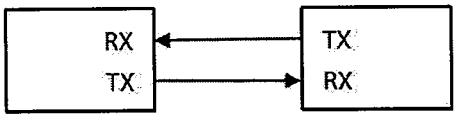

图 5.1 UART 接口示意图

Figure 5.1 UART interface diagram

UART 没有所谓的主从模式，在工作时需要配置好波特率，每笔传输数据就会以按照期望的形式进行传输，其具体传输方式都由内部寄存器控制。

UART 接口信号不包括时钟信号，所以对每笔数据都有严格的要求，为了不引起传输错误，通信双方需要在寄存器的配置上达成一定的共识。

5.1.2　UART 工作方式

UART 的工作传输时序如图 5.2 所示，接口信号在空闲状态保持高电平，当有传输需求时，信号线拉低到低电平进入起始位，之后根据内部寄存器配置，传输 5-8 位数据位，奇偶校验位和停止位，停止位保持高电平，此时就表示一笔数据传输结束。

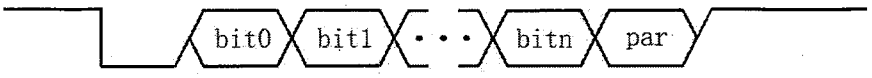

图 5.2 UART 传输时序图

Figure 5.2 UART transmission timing diagram

UART 存在一个波特率时钟输出，图示的每一位都对应波特率时钟的 16 个周期，而波特率时钟与 apb 接口时钟的对应关系由内部寄存器控制[50]。

5.1.3 UART 寄存器

UART 内部主要有九个 8 位寄存器，这些寄存器用来配置 UART 的工作模式，保存发送和接收的数据以及表示 UART 当前的工作状态。模块内部寄存器接收如下表 5.1 所示：

表 5.1 UART 寄存器

Table 5.1 UART register

名称	偏移地址	位宽	R/W	描述
RXD	0X00	8	RO	接收 FIFO 输出寄存器
TXD	0X00	8	WO	发送 FIFO 输入寄存器
IER	0X04	8	R/W	中断使能寄存器
IIR	0X08	8	RO	中断标志寄存器
FCR	0X08	8	WO	FIFO 控制寄存器
LCR	0X0C	8	R/W	传输控制寄存器
LSR	0X10	8	RO	传输状态寄存器
DIV1	0X14	8	R/W	波特率控制寄存器 1
DIV2	0X18	8	R/W	波特率控制寄存器 2

5.2 验证平台重用

前文介绍了 SPI 验证平台搭建过程，在此过程中充分考虑了如何提高验证平台的可重用性。下面将基于搭建好的 SPI 验证平台，重用内部部分组件，完成 UART 验证平台的搭建工作，以此证明验证平台的高可重用性。

5.2.1　验证平台结构设计

UART 验证平台的设计与 SPI 的类似，采用了图 5.3 所示的层次化设计，主要包括顶层环境、APB 环境和 UART 环境三部分，验证平台的搭建就是基于此框图完成底层组件的设计。

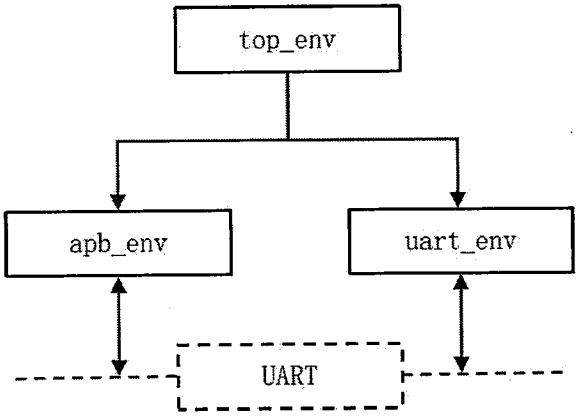

图 5.3 UART 验证平台层次结构

Figure 5.3 UART verification platform hierarchy

基于 UVM 的 UART 验证平台结构如图 5.4 所示。从图中可以看出，该平台结构与 SPI 的平台结构及其相似，只是由于 UART 和 SPI 不同，没有所谓的主从工作模式，所以验证平台中只有一个 uart_agent。

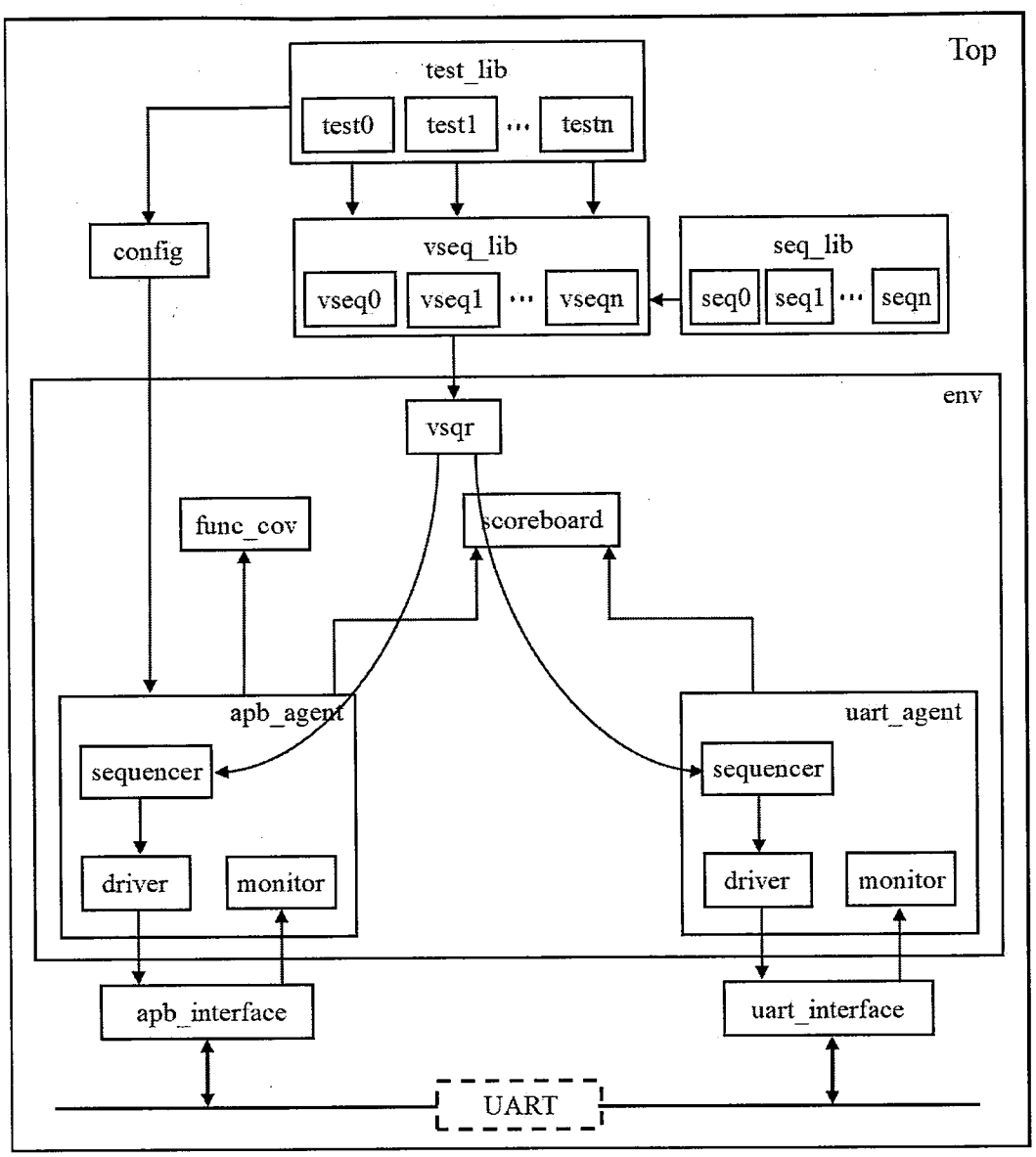

图 5.4 UART 验证平台结构图

Figure 5.4 UART verification platform structure diagram

5.2.2 验证组件重用

前文已经介绍了 SPI 验证平台内部各个组件的设计过程，在此过程中对于如何提高可重用性作了相关设计，之后就可以重用这些验证组件，节省 UART 验证平台的搭建时间。

此次验证工作的验证目标 UART 与 SPI 都是挂载在 APB 总线上的外设，APB 作为通用的总线协议，SPI 验证平台中的 apb_agent 组件及其内部封装的 driver、monitor 和 sequencer 组件可以直接重用到 UART 验证平台，而且 driver 组件内部还引入了 callback 机制，进一步加强了对 transaction 的控制。除此之外，SPI 验

证平台中的 apb_transaction 和 apb_interface 也都是抽象自 APB 总线协议，所以也可以直接进行重用。

此 UART 验证平台引入了寄存器模型，所以为了方便验证平台对寄存器模型进行访问，有必要引入 adapter 组件，用于在不同类型 transaction 之间进行转换。SPI 验证平台中的 adapter 可以对 uvm_reg_bus_op 类型和 apb_transaction 类型进行转换，而此 UART 验证平台有相同的需求，所以此 adapter 组件也可以直接进行重用。

UART 和 SPI 验证平台的层次结构十分类似，对于 env 组件虽然做不到完全重用，但内部定义相似，都需要对 agent、virtual sequencer 和 scoreboard 等组件进行声明并实例化，所以只需对 SPI 验证平台中 env 组件内部定义作适当的修改，也可重用到 UART 的验证平台中。

对于 uart_agent 和 scoreboard 等组件，与 UART 协议的依赖程度较高，重用的难度较大，所以在这里选择重新定义。

5.3　验证平台运行

运行验证平台的过程和 SPI 的一样，也需要引入 sequence 机制来构建测试用例，然后通过 virtual sequencer 和 virtual sequence 的配合来生成适当的激励数据。最后查看覆盖率信息，判断此次验证工作的完备性。下面将对上述整个流程作一个详细介绍。

5.3.1　测试用例

为了尽可能多的覆盖测试功能点，一共构建了三个测试用例，分别用来进行 UART 基本功能测试、中断功能测试和波特率测试。和前文介绍的 SPI 的测试用例类似，为了与上述三个 test 对应，平台中也定义了三个 virtual sequence，每个 test 内部都会将一个 virtual sequence 设置为 default_sequence，然后通过嵌套 sequence 的方式，由 virtual sequence 从 uart 验证平台的 sequence 库中选择合适的 sequence，并使用 fork...join 的方式对其进行同步。

在定义 sequence 时，可以重用 SPI 验证平台中的 uvm_base_sequence，因为其内部只包含基于 APB 总线的读写任务，与 SPI 没有任何依赖关系，所以可以在此处进行直接重用，然后基于此类扩展出其他派生类用于 UART 的验证。

由于这里也引入了寄存器模型，所以对于寄存器功能的验证可以直接使用 UVM 中内建 sequence 来完成，而无需构建专门的测试用例。

对于测试用例的运行仍然通过脚本来完成，将 SPI 验证时使用的脚本文件作适当修改，更换测试用例的名称即可用于运行 UART 的测试用例。

5.3.2 仿真结果

图 5.5 为测试用例 uart_basic_test 的仿真记录文件，从中可以看出本次仿真产生了五个 UVM_INFO 信息，没有产生 UVM_WARNING、UVM_ERROR 和 UVM_FATAL 信息，五个 UVM_INFO 信息中有一个来自波特率检查组件，两个来自 scoreboard 组件，还有一个来自 uart_basic_test 组件。其中第一个来自波特率检查组件的信息表示当前测试用例没有发生波特率错误，接下来两个来自 scoreboard 的信息表示通过 UART 向外发送的数据和接收的数据均比对成功，最后一条来自 uart_basic_test 的信息表示本次仿真没有发生任何错误。

```
UVM_INFO ../verif/env/baud_rate_checker.svh(122) @ 8555800: uvm_test_top.m_env.br_sb
[baud_rate_checker] This's no baud rate errors
UVM_INFO ../verif/env/uart_rx_scoreboard.svh(133) @ 8555800: uvm_test_top.m_env.rx_sb
[uart_rx_scoreboard] Characters received with no errors
UVM_INFO ../verif/env/uart_tx_scoreboard.svh(114) @ 8555800: uvm_test_top.m_env.tx_sb
[uart_tx_scoreboard] Characters transmitted with no errors
UVM_INFO ../verif/tests/word_format_poll_test.svh(30) @ 8555800: uvm_test_top
[uart_basic_test] this test case passed!

--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :    5
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
[baud_rate_checker]     1
[uart_basic_test]     1
[uart_rx_scoreboard]     1
[uart_tx_scoreboard]     1
```

图 5.5 UART 仿真记录文件

Figure 5.5 UART simulation record file

图 5.6 为测试用例 uart_basic_test 的仿真波形，从中可以看出首先进行的是对 UART 内部地址为 32'h0c 和 32'h08 的 LCR 和 FCR 寄存器进行配置，对地址为 32'h14 和 32'h18 的波特率控制寄存器进行配置，其配置数据由前文介绍的

virtual sequence 中第一个 sequence 产生；然后对地址为 32'h10 的 LSR 寄存器进行读操作，确定 UART 当前的状态，之后就开始对地址 32'h00 的 TXD 寄存器进行写操作，写入的就是需要通过 UART 向外发送的数据，当这笔数据写入之后 TX 接口将信号拉低进入起始位，开始接下来的传输工作，与此同时，RX 接口也在传输 UART 要接收的数据。

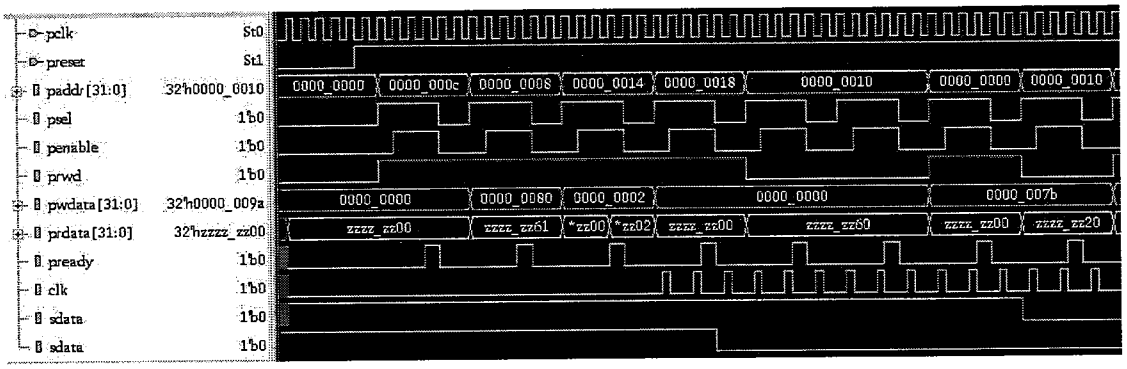

图 5.6 uart_basic_test 仿真波形图

Figure 5.6 uart_basic_test simulation waveform

在使用脚本文件完成测试用例的运行之后，就可以通过 DVE 查看最后的覆盖率信息，包括代码覆盖率和功能覆盖率。

下图 5.7 所示为仿真结束后的代码覆盖率情况，从图中可以看出，本次验证工作只有行覆盖率和分支覆盖率没有达到 100%，其余三方面均达到 100%覆盖率。

Name	Score	Line	Toggle	FSM	Condition	Branch
uart_tb	99.39%	98.92%	100.00%	100.00%	100.00%	98.03%
DUT	99.38%	98.85%	100.00%	100.00%	100.00%	98.03%
control	99.00%	98.20%	100.00%	100.00%	100.00%	96.83%
rx_channel	99.56%	99.19%	100.00%	100.00%	100.00%	98.61%
tx_channel	99.53%	99.12%	100.00%	100.00%	100.00%	98.53%
IRQ	100.00%		100.00%			
RX_UART	100.00%		100.00%			
TX_UART	100.00%		100.00%			
apb_vif	100.00%		100.00%			

图 5.7 UART 验证代码覆盖率

Figure 5.7 UART verification code coverage

下图 5.8 所示为仿真结束后的功能覆盖率情况，从图中可以看出，本次验证工作所有的覆盖点和交叉覆盖都达到了 100%覆盖率。

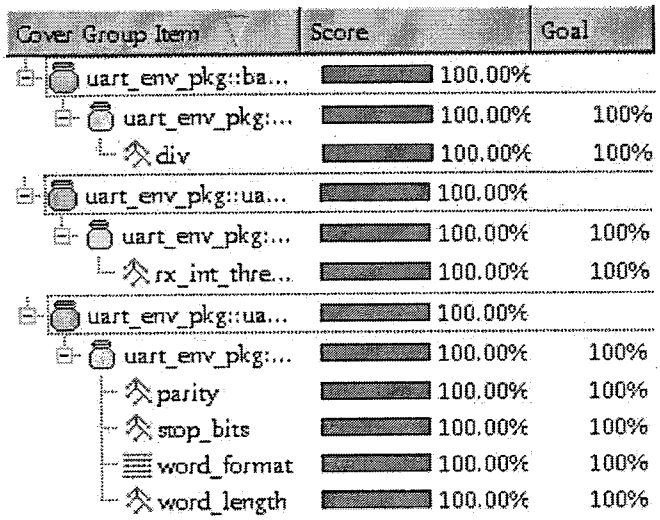

图 5.8 UART 验证功能覆盖率

Figure 5.8 UART verification function coverage

根据以上仿真结果可以看出，SPI 验证平台内的部分组件完全可以重用于 UART 的验证平台，并且此 UART 验证平台也完全可以用来完成 UART 的验证工作。

5.4 重用性分析

为了对本论文所搭建的验证平台的可重用度进行分析，所以对 UART 验证平台中各组件的代码量进行统计。因为图 5.3 UART 验证平台层次结构中 uart_env 内部组件完全依赖于待测设计，属于完全不可重用的组件，所以在这里不再分别统计，其他组件的统计结果如表 5.2 所示。

由下表分析可得，整个验证平台中各组件的总代码量为 1080 行，其中重用的代码量为 311 行，非重用的代码量为 769 行，所以计算可得整个验证平台的重用率达到 28.8%。

在验证平台的搭建过程中，如果有超过 1/4 的代码可以重用，就可以在很大程度上提高验证平台的搭建速度，加快验证工作的效率，从而证明本论文搭建的验证平台确实具有更高的可重用性。

表 5.2 代码量统计

Table 5.2 Code volume statistics

验证组件	总代码量	重用代码量
apb_agent	38	38
apb_driver	71	71
apb_monitor	41	41
apb_sequencer	13	13
apb_interface	19	19
apb_transaction	28	28
virtual sequencer	15	11
env	71	31
adapter	32	32
base_sequence	27	27
uart_env	725	0

5.5　系统级可重用探究

验证工作一般分为系统级的验证和模块级的验证[51]。一个大的芯片内部通常存在多个小的模块，在进行验证工作时需要先对这些子模块进行模块级的验证，这些子模块的验证平台的搭建工作一般由不同的人员负责[52]，当所有模块的验证工作完成，就要开始进行系统级的验证工作。

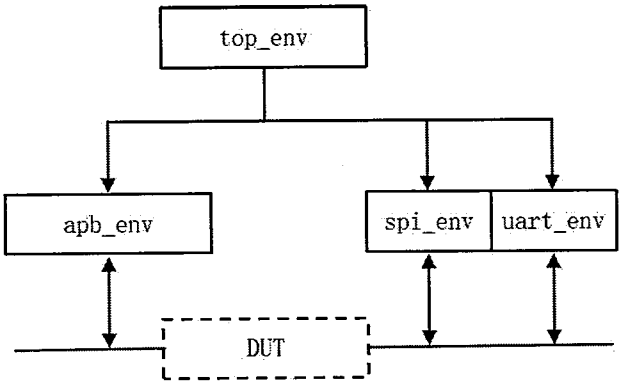

图 5.9 系统级验证平台层次设计

Figure 5.9 System-level verification platform level design

前文已经完成了对 SPI 和 UART 模块级的验证，可以尝试重用上述模块级验证平台进行系统级的验证工作。图 5.9 为系统级验证平台的层次结构，其中重用了 SPI 和 UART 验证平台中的 apb_agent、spi_agent 和 uart_agent，以及 scoreboard 和 func_cov 等组件，各组件的具体来源如表 5.3 所示，其中各 agent 组件同时代表了其内部封装的其他组件。对于系统级验证的测试用例，可以通过 virtual sequence 使用嵌套的方法，从 SPI sequence 库和 UART sequence 库中选择适当的 sequence，使用 fork…join 的方式进行同步。

表 5.3 系统级验证平台组件来源

Table 5.3 Source of system-level verification platform components

验证组件	重用组件来源	
	SPI 验证平台	UART 验证平台
apb_agent	✔	✔
spi_env	✔	
uart_env		✔
spi_master_agent	✔	
spi_slave_agent	✔	
uart_agent		✔
spi_scoreboard	✔	
uart_scoreboard		✔
spi_func_cov	✔	
uart_func_cov		✔

图 5.10 所示为系统级仿真的记录文件，此次仿真的测试用例是由 spi_master_test 和 uart_basic_test 组合而成。从图中可以看出本次仿真产生了六个 UVM_INFO 信息，其中前两个信息表示系统中的 SPI 工作正常，后三个信息表示系统中的 UART 工作正常。

```
UVM_INFO ../verif/scoreboard&cov/spi_scoreboard.sv(264) @ 312868750:
uvm_test_top.env.scb [scb] SPI Receive compare SUCCESSFULLY
UVM_INFO ../verif/scoreboard&cov/spi_scoreboard.sv(201) @ 312868750:
uvm_test_top.env.scb [scb] SPI Transmit compare SUCCESSFULLY
UVM_INFO ../verif/scoreboard&cov/baud_rate_checker.svh(122) @ 1312868750:
uvm_test_top.m_env.br_sb [br_sb] BAUD RATE check SUCCESSFULLY
UVM_INFO ../verif/scoreboard&cov/uart_rx_scoreboard.svh(134) @ 1312868750:
uvm_test_top.m_env.rx_sb [rx_sb] UART Receive compare SUCCESSFULLY
UVM_INFO ../verif/scoreboard&cov/uart_tx_scoreboard.svh(115) @ 1312868750:
uvm_test_top.m_env.tx_sb [tx_sb] UART Transmit compare SUCCESSFULLY

--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :    6
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
[br_sb]     1
[rx_sb]     1
[scb]     2
[tx_sb]     1
```

图 5.10 系统级仿真记录文件

Figure 5.10 System-level simulation record file

第 6 章　结论与展望

6.1　结论

本文详细介绍了 UVM 验证方法学，并与软件设计模式相结合，搭建了以 SPI 作为待测设计的验证平台，之后重用该平台内部组件完成了 UART 验证平台的搭建工作，以此证明此验证平台的高可重用性。本文主要完成工作如下：

(1)选择 SPI 作为待测设计，搭建基于 UVM 的验证平台。首先根据 SPI 的功能提取验证功能点，然后对验证平台进行层次化处理，对底层各个组件分别建模。过程中借鉴软件设计模式的内容，运用随机验证方法和 UVM 中的多种机制，实现验证平台的高可重用性。在验证运行阶段，根据验证功能点，运用 sequence 机制完成测试用例的构建，通过脚本完成测试用例的运行。为了保障验证工作的完备性，本文引入覆盖率作为衡量依据，经过不断测试，最后功能覆盖率达到 100%，代码覆盖率达到 99.6%。

(2)基于 SPI 验证平台，以 UART 作为待测设计，完成验证平台的搭建工作。过程中重用 SPI 验证平台内部可重用组件，按照 UART 协议内容完成验证平台中其他底层组件的定义。测试用例的构建运用 sequence 机制，然后通过脚本完成仿真，最终功能覆盖率达到 100%，代码覆盖率达到 99.39%，以此说明验证平台在模块级验证的可重用性。为了说明本论文所搭建的验证平台在系统级验证的可重用性，本文尝试性的对 SPI 和 UART 进行了综合验证，并且得到了正确的验证结果。

6.2　展望

作为验证方法学中的后起之秀，本文基于 UVM 验证方法学，结合面向对象语言的相关特征以及软件设计模式的部分模式搭建验证平台，进一步提高了其可重用性。

本文虽然搭建了 SPI 的验证平台，并且基于此又搭建了 UART 的验证平台，但是还存在部分不足与需要改进的地方：

（1）断言(assertion)作为 SystemVerilog 语言的一部分，可以被引入到此次的验证工作中，对接口时序进行检查，当仿真出现错误时，断言可以快速地对错误

进行定位；

（2）本文最后只对系统级的可重用作了尝试性的探究，但是实际上系统级验证要比这复杂得多，在下一步的工作中，可以对系统级的可重用进行更深入的探究；

（3）对于功能点的选取，实际工作中需要与设计人员共同讨论确定，但本文更倾向于对方法的探究，所以在功能点的选取上难免存在不足，在下一步的工作中，可以选取更多的功能点并构建相应的测试用例，完善整个验证工程。