

电子科技大学
UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

专业学位硕士学位论文
MASTER THESIS FOR PROFESSIONAL DEGREE



论文题目 基于 UVM 的 APB-UART 验证平台的设计与实现

专业学位类别 工程硕士

学 号 201722030622

作 者 姓 名 段一杰

指 导 教 师 王忆文 教授

分类号 _____ 密级 _____

UDC ^{注 1} _____

学 位 论 文

基于 UVM 的 APB-UART 验证平台的设计与实现

(题名和副题名)

段一杰

(作者姓名)

指导教师

王忆文

教 授

电子科技大学

成 都

(姓名、职称、单位名称)

申请学位级别 硕士 专业学位类别 工程硕士

工程领域名称 集成电路工程

提交论文日期 2020.05 论文答辩日期 2020.06

学位授予单位和日期 电子科技大学 2020 年 6 月

答辩委员会主席 _____

评阅人 _____

注 1: 注明《国际十进分类法 UDC》的类号。

Design and Implementation of APB-UART Verification Testbench Based on UVM

A Master Thesis Submitted to
University of Electronic Science and Technology of China

Discipline: **Master of Engineering**

Author: **Yijie Duan**

Supervisor: **Prof. Yiwen Wang**

School: **School of Electronic Science and Engineering**

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

作者签名： 段-杰

日期：2020年6月5日

论文使用授权

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

作者签名： 段-杰

导师签名： 王文

日期：2020年6月5日

摘 要

随着集成电路领域的不断发展以及 IP(Intellectual Property)集成技术和工艺制程的不断提升,当前芯片的结构愈加复杂,验证工作难度也越来越大,导致流片失败造成的损失越来越难以承受。这些都对芯片的验证工作提出了新的要求。

在当前阶段,常见的验证思想主要有定向激励验证、随机化验证、形式验证和 FPGA(Field Programmable Gate Array)验证等。经过综合对比各自的优劣,可以发现随机化验证方法拥有着明显的优势。在业界,能够很好支持受约束的随机化方法的 UVM(Universal Verification Methodology)验证方法学的应用越来越广泛,代表了验证领域的方向与潮流。

APB(Advanced Peripheral Bus)协议和 UART(Universal Asynchronous Receiver Transmitter)协议作为常见的低速总线协议,具有互连信号简单、功耗较低的特点,应用十分广泛。本文需要验证的待测模块(DUT, Design Under Test)为教研室开发的 APB-UART 模块。该模块是处理器和外围通信端口沟通的重要桥梁,是 SoC(System On Chip)领域中十分重要的模块。以往对此类模块的验证往往采用 FPGA 验证或定向激励验证,耗费大量人力和时间的同时,却不能完备高效的对其功能进行全覆盖。因此,基于 UVM 验证方法学为该模块搭建一款验证平台,具有十分重要的意义。

本文为该款 APB-UART 模块搭建了配套的 UVM 验证平台,并通过对各种功能场景的检测,完成了该模块的验证工作,主要内容如下:

(1) 深入研究了该 DUT 模块,包括其子模块结构、端口和寄存器信息,并在充分了解其功能点后,制定了相应的验证计划。

(2) 深入学习 UVM 验证方法学,为该模块搭建了配套的 UVM 验证平台。该平台包含以下结构:多个 agent 组件,分别负责 APB 端口、UART 收发端口、modem 模块端口数据的驱动和采集工作;多个 scoreboard 组件,对数据进行比对和检查;拥有预警机制的寄存器模型;验证顶层、验证环境以及其他必要的结构。

(3) 开发多组测试用例,实现随机激励的自动生成,对待测功能点进行覆盖。在验证后期,通过不断修改约束条件,使得测试用例对 DUT 的覆盖率达到 100%,标志这款验证平台设计合理并被成功实现。

关键词: 验证, UVM, APB, UART

ABSTRACT

With the development of the integrated circuit and the improvement of IP (Intellectual Property) integration technology, the functions of the chips are becoming more and more powerful, and the module structure can be very complicated. The loss caused by the failure of tape-out is increasingly difficult to bear. All these put forward new requirements for the verification of the chip.

At the current stage, common ideas of verification mainly include direct verification, randomization verification, formal verification, and FPGA (Field Programmable Gate Array) verification. After comparing the advantages and disadvantages of each verification idea comprehensively, we found that the randomization verification has distinct advantages. In the industry, UVM (Universal Verification Methodology) verification method, which supports the randomization verification better, is widely used, representing the direction and the trend of the verification field.

APB (Advanced Peripheral Bus) protocol and UART (Universal Asynchronous Receiver Transmitter) protocol, as common low-speed bus protocols, have the advantages of low power consumption and simple interconnection signals. In this article, DUT (Design Under Test) is a certain APB-UART module designed by our laboratory. This module is critical in the SoC (System on Chip) structure, which works as an essential bridge for communication between the processor and peripheral devices. In the past, the verification engineers often used FPGA verification or direct verification to test such modules, which consumed many resources but could not fully and efficiently verify the DUT. Therefore, it is of great significance to build a verification platform for this module based on UVM.

We built a UVM verification platform for this APB-UART module and completed the verification by verifying various test scenarios. The main work done in this article is as follows:

(1) We analyzed this APB-UART module, including protocol specifications, module architecture, the definition of ports, and the information of registers. Then, we formulate a corresponding verification plan.

(2) We in-depth researched the UVM and built a verification platform for this

module. That platform contains the following part: (a)multiple agent components, which are responsible for driving and monitoring the data of APB port, UART port, and modem port; (b)multiple scoreboard components, which are used for data comparison and inspection; (c)register model with warning mechanism; (d)verification top, verification environment, and other necessary structures.

(3) We developed multiple test cases to realize the automatic generation of random stimuli for covering the function points. In the later stage of verification, we changed the constraint conditions and reached the 100% coverage, indicating that the verification platform is designed reasonably and implemented successfully.

Keywords: Verification, UVM, APB, UART

目 录

第一章 绪论	1
1.1 研究背景	1
1.1.1 常见验证方法及各自优缺点	1
1.1.2 UVM 验证方法学的发展	3
1.1.3 UVM 方法学的现状与未来	4
1.2 主要工作及论文结构	5
1.3 论文研究创新点	6
第二章 UVM 验证技术概论	7
2.1 UVM 基础	7
2.1.1 UVM 验证平台架构	7
2.1.2 uvm_object 和 uvm_component 基类	8
2.1.3 UVM 树形结构	9
2.2 UVM 基本运行机制	9
2.2.1 Phase 机制	10
2.2.2 Objection 机制	11
2.2.3 Config_db 机制	11
2.3 寄存器模型	12
2.4 TLM 通信机制	13
2.5 覆盖率机制	14
2.5.1 代码覆盖率	15
2.5.2 功能覆盖率	16
2.6 本章小结	18
第三章 APB-UART 模块及验证计划	19
3.1 APB 总线协议	19
3.1.1 APB 总线与 SOC 系统	19
3.1.2 APB 总线传输的实现	19
3.2 UART 总线协议	20
3.2.1 UART 信号传输格式	21
3.2.2 UART 收发逻辑	21

3.3 APB-UART 模块介绍.....	22
3.3.1 模块架构及功能介绍	22
3.3.2 模块端口介绍	23
3.3.3 模块寄存器介绍	24
3.4 验证计划分类	25
3.4.1 测试计划	26
3.4.2 覆盖率计划	27
3.4.3 检查计划	28
3.5 验证计划制定	29
3.6 本章小结	30
第四章 UVM 验证平台设计与实现.....	31
4.1 APB-UART 验证平台总体架构.....	31
4.2 寄存器模型的实现	31
4.2.1 基本寄存器模型的实现	31
4.2.2 预警机制的实现	32
4.3 Apb_Agent 关键组件的实现.....	34
4.3.1 Apb_Driver 的实现	34
4.3.2 Apb_Monitor 的实现	35
4.3.3 其它类的实现	35
4.4 Uart_Agent 关键组件的实现	36
4.4.1 Uart_Driver 的实现.....	36
4.4.2 Uart_Monitor 的实现.....	37
4.4.3 其它类的实现	38
4.5 Modem_Agent 关键组件的实现.....	39
4.5.1 Modem_Driver 的实现	39
4.5.2 Modem_Monitor 的实现.....	39
4.5.3 其它类的实现	40
4.6 Scoreboard 组件的实现.....	40
4.6.1 Rx_Scoreboard 的实现	40
4.6.2 Tx_Scoreboard 的实现.....	41
4.6.3 Modem_Scoreboard 的实现	42
4.6.4 Div_Checker 的实现.....	42
4.7 其它组件及模块的实现	43

4.8 本章小结	44
第五章 验证计划实现与验证结果	45
5.1 验证计划实现	45
5.1.1 测试计划的实现	45
5.1.2 覆盖率计划实现	55
5.1.3 检查计划实现	57
5.2 验证结果分析	57
5.2.1 验证环境描述	58
5.2.2 典型测试用例结果分析	58
5.2.3 最终验证结果分析	60
5.3 本章小结	61
第六章 结论	63
6.1 工作总结	63
6.2 不足与展望	63
致谢	64
参考文献	65
攻读硕士学位期间取得的研究成果	67

第一章 绪论

1.1 研究背景

随着数字 IC 工艺的不断进步，数字芯片的规模和复杂度也日渐提升。同时，由于 SOC(System On Chip)设计技术的诞生，数字电路设计的效率和能力获得了极大的提升^[1]，这进一步导致了芯片功能和复杂度的增长。而由于工艺制程的提升和市场对产品缺陷越来越低的容忍度，每次流片失败后需要负担的成本也越来越高，这就使得流片前的验证工作越来越重要。现在业界的普遍共识是，验证工作在芯片设计流程中所占用的平均时间已经达到了整个设计周期的一半以上，在部分芯片验证领域，验证用时所占比例甚至达到了 80%以上^[2]。这说明验证工作在芯片的设计过程中发挥着越来越重要的作用。

1.1.1 常见验证方法及各自优缺点

目前常见的验证方法主要有四种：FPGA 验证、Formal 验证、定向激励验证以及受约束的随机化验证。每种验证方法都有其长处和不足，简要介绍如下：

(1) FPGA 验证属于较为传统的验证方式。FPGA 开发板上包含了大量的可编程逻辑单元、存储单元以及丰富的布线资源等。在模块的设计工作完成后，验证人员可以将 RTL 代码烧写到 FPGA 板子上，进行实物验证。这种验证方式由硬件实现，仿真速度快，灵活性高，但也拥有一定的缺点：不利于测试场景的构建；购买验证平台(FPGA 开发板)需要额外的资金投入；发现 bug 后无法快速的定位；开发板上可烧入的代码有大小限制等。这些缺点对复杂模块的验证工作有着很大影响。

(2) Formal 验证是基于已建立的形式化规格，采用数学的方法对目标模块的相关特性进行分析和验证，以判断其功能是否与预期一致^[3]。这种方法可以避免测试向量的枚举，实现无死角验证，但需要一定的数学功底和建模能力，所以还没有大规模应用在功能验证中。不过该方法在 IC 流程的其他环节已经得到了广泛应用。

(3) 定向激励测试一般用于低复杂度的模块级验证中。验证人员需要罗列出该模块的各个测试点和对应的激励，然后按照条目一项一项的去进行仿真测试。在仿真结束后，还需要人工去核对结果或者波形，确保待测模块在预期的工作状态下产生了正确的结果。

定向测试的方法并不需要搭建十分复杂的验证平台，验证人员所进行的操作只需要符合“产生激励-核对结果”原则就可以，所以这种测试方法可以较为快速

的取得阶段性成果，每一个待测项验证完毕后，都可以推动验证进度。

但定向测试有着先天的缺陷，主要体现在以下三个方面：(a)由于每个测试激励都对应着一条待测项，那么对于状态空间和功能组合比较复杂的模块，人工罗列出其所有的待测组合需要耗费极大的验证资源。使用定向测试时，验证进程的提升和时间成线性关系，这就意味着待测模块复杂度的提升也会导致测试时长的增加。(b)随着电路集成度和模块复杂性的增加，有时验证人员并不能够列出全部的待测项，而且只能想到预期中的电路漏洞，对于预料不到的错误无法进行检测，这往往会导致流片失败。(c)仿真结束后，需要人工去核对结果，进一步加大了验证资源的耗费。

(4) 受约束的随机化测试则要求验证平台能够在一定约束下产生随机的测试激励。采用这种方法搭建的验证平台往往更为复杂，可能在较长一段时间后才能够运行第一次测试，但是由于每个测试激励都可以共享验证平台，所以后续测试用例的实现速度会有很大的提升。同时结果自检和以覆盖率驱动等思想的提出，也弥补了之前定向测试的不足，使得验证过程更加自动和高效。

当然，在一个模块的测试过程中，并不是一直都要使用随机化的方法。这需要考虑投入和产出的比值，对于简单功能点的验证，可以使用定向激励来快速的完成任务。那些过度使用随机化方法的纯粹主义者，可能违背了引进这种方法的初衷。

另外，由于受约束的随机化方法具有随机性，或者由于某个激励的产生条件过于复杂，而导致这个激励总是不能够被产生出来时，就需要编写定向激励，去快速推进验证进程。

在分别采用定向测试和受约束化的随机测试验证同一个项目时，项目进度和时间的关系如图 1-1 所示^[4]：

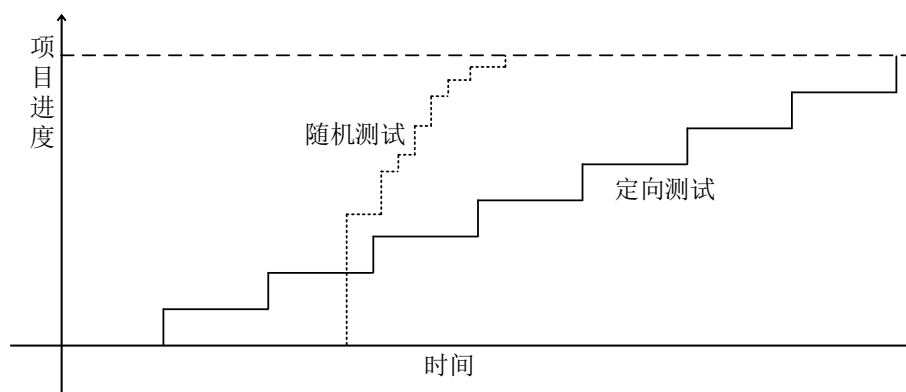


图 1-1 定向测试和受约束的随机测试时间-进度轴^[4]

综上所述，目前业界主流的验证方法为受约束的随机化测试，并且这种测试方

法有一套通用理论指导和库文件支持，即 UVM 验证方法学。

1.1.2 UVM 验证方法学的发展

验证是为设计服务的，被用来发现待测模块(DUT, Design Under Test)中存在的缺陷或错误，但验证又区别于设计，在实现层次上更偏向于软件。随着 IC 产业的发展，验证语言也在不断的发生着变化，目前常见的验证语言有 Verilog、SystemC 以及 SystemVerilog 等。其中 SystemVerilog 由于具有面向对象的特点，且能完美的兼容 Verilog，已经成为了现在验证领域的主流语言。

但是单独使用一门语言是难以完成验证工作的。因为验证人员往往需要搭建一个验证平台，将 DUT 嵌入到该平台中，通过平台给 DUT 施加激励，并检测结果是否符合预期。在这个过程中就容易出现两类问题：首先，由于工作经验和水平的不同，有些验证人员可能并不能够充分发挥验证语言的作用，搭建出足够优质的验证平台，完备的对 DUT 进行验证；再者，不同验证人员的验证思维与实现方法可能有很大区别，不利于人员之间进行技术交流，也会降低验证平台的重用性和移植性。

因此，验证方法学应运而生，用来指导不同的验证人员在特定的思维下编写验证平台。由图 1-2 可以了解到验证方法随着时间发展与更替的趋势^[5]。

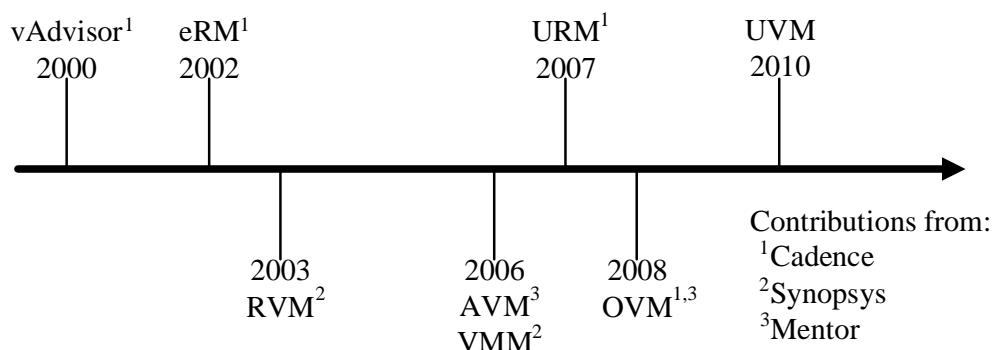


图 1-2 验证方法学发展时间轴^[5]

在 2000 年，Verisity Design 公司推出了一款名为 vAdvisor 的工具，该工具以 HTML 表格的形式介绍了多种多样的设计模式以及重要概念，包括激励产生、自检测功能以及覆盖率模型等。但很快，该公司便意识到仅仅有理论知识是不够的，如果想要实现验证平台的可重用化和自动化，必须要依靠共有的库文件支持。

在众多使用者的支持下，仅仅两年后，该公司便推出了第一套验证库文件 eRM，这套库文件及其背后的验证思想对后续各种验证方法学的诞生产生了深远影响，比如验证框架的结构、sequence 机制、objection 机制等等。

在 2003 年, Synopsys 公司推出了适用于验证领域的 RVM 库, 这套库文件基于 Vera 语言, 但其包含的理念和代码文件并没有 eRM 那样丰富多样, 所以在当时仅仅被当作 eRM 的附属或者替代。不过 RVM 提出了很重要的 callback 机制, 这一理念在 UVM 中依然存在。

后来随着验证语言的发展, 陆续涌现了一批优秀的验证方法学, 并拥有对应的库文件支持。2006 年由 Synopsys 公司提出的 VMM 方法学以及 2008 年由 Cadence 和 Mentor 公司提出的 OVM 方法学奠定了当前主流验证方法学的雏形, 这两套验证方法学各有特色: 其中 VMM 提出了针对寄存器/存储器的验证解决方案, 极大的简化了相关的验证过程; OVM 则在之前方法学的基础上引入了 factory 机制, 实现了对不同类(Class)的重载。

最终, 于 2010 年, Accellera 组织和三大巨头公司共同推出了 UVM(Universal Verification Methodology)验证方法学。该方法学继承了 OVM 和 VMM 的优点, 逐渐成为了当前验证领域的主流验证方法^[6]。

1.1.3 UVM 方法学的现状与未来

自 UVM 验证方法学诞生后, 国内外大量的专家学者都对该方法学进行了研究和使用的。

在国内业界, 与 UVM 相关的验证技术总是在不断发展: 徐金甫, 李森森等人在验证组件横向和纵向实现、寄存器模型的脚本自动化、仿真结果实时在线对比等方面提出了自己的看法, 提高了验证平台的可重用性, 快速完成了对 PCIe 模块的验证^[7]; 田晓旭等人利用特定格式的 Excel 表格以及脚本文件, 实现了 ralf 文件的自动产生, 再借助 ralgen 命令, 使得寄存器模型的生成更加自动化, 提高了验证效率, 减少了出错的概率^[8]; 张少真, 成丹, 刘学毅等人通过调用 Matlab 引擎的方法让 Matlab 和 UVM 进行联合仿真, 从而使 UVM 可以访问 Matlab 的算法模型, 加快了验证平台的搭建时间, 提高了仿真效率^[9]。

国内各个高校也加强了对 UVM 验证领域的探索: 张瑞使用基于断言语句的形式化验证方法和 UVM 验证方法分别对某款多核调试模块进行了综合验证^[10]; 刘魁玉设计了一款基于 UVM 的 1553B 总线协议验证平台, 切实提高了验证效率^[11]; 陈琳娜在现有 UVM 验证平台的基础上, 采用层次化建模的方法实现了一套共用库, 提高了验证平台的可重用性^[12]。

在国外, 各个专家对于 UVM 验证方法学的研究也是毫不停歇: John Aynsley 等人通过自定义的 register sequence, 解决了复杂寄存器的访问问题, 对于那些无法在 adapter 中直接访问, 而需要多个 transaction 组合才能实现访问的寄存器, 提

出了独特的访问方案^[13]；对于那些对错误输入有指定行为的健壮设计，Kurt Schwartz 和 Tim Corcoran 等人对错误输入的产生位置，以及如何高效注入错误激励并得到响应等问题提出了自己的看法^[14]；Jeff Vance 等人在 uvm harness 的基础上，针对因为模块配置或版本变更而需要耗费大量验证资源的问题，提出了一套较为新颖的解决方案，并且当设计模块的验证层级发生变化时，使用这种方法也可以减少相关资源的投入^[15]。

可以看到，国内外的专家学者都从理论或实践上开始重视 UVM 验证方法学，说明使用 UVM 验证方法学在验证领域已经是一种不可阻挡的趋势。

未来 UVM 的发展可能会着重体现在验证 IP 商用化^[16]、类库功能的不断增强以及在架构、测试等多个领域提供统一化解决方案等方面^[5]，这都需要广大验证人员、公司以及业界巨头的努力和沟通。

1.2 主要工作及论文结构

本文在基于 SystemVerilog 语言的基础上，研究了 UVM 验证方法学思想及通用库的使用方法，然后以教室内部的某款 APB-UART 模块为待测模块(DUT)，在充分掌握其结构，分析其待测功能点后，提出了验证计划，并在 UVM 验证方法学的基础上，为该模块搭建了配套的验证平台，实现了随机化激励生成、结果自动比较以及覆盖率收集等功能。最后，针对多种测试场景编写了对应的测试用例，在该验证平台上执行完全部的测试用例后，最终的覆盖率达到 100%，说明该验证流程充分完备的完成了对 DUT 模块的验证工作。

本论文一共分为六个章节，各个章节内容如下：

第一章为绪论，首先介绍了常用的验证方法及其优缺点，以此引入 UVM 验证方法学；然后以时间为单位进行纵向对比，详细阐述了 UVM 方法学的存在意义及发展历史；之后对该方法学进行横向对比，介绍了国内外的研究现状以及未来的发展趋势；最后简要介绍了本文的主要工作和论文结构。

第二章介绍了 UVM 验证方法学的基本知识，包括 UVM 验证平台的架构与基本运行机制、寄存器模型的意义与实现方法、TLM(Transaction Level Modeling)通信机制和覆盖率机制的实现等。

第三章首先对 DUT 模块进行了简要介绍，包括相关接口协议、DUT 的架构和功能、DUT 的寄存器定义等；然后引入验证计划的概念并制定了相关的验证计划表，这是整个验证流程的指导性计划，后续章节都与该计划表相关。

第四章主要介绍了验证平台的实现细节，包括寄存器模型的实现、预警机制的添加、各个 agent 组件及 scoreboard 组件的实现等，最后还简要介绍了其他结

构与模块的实现。本章描述了搭建完整 UVM 验证平台的过程，是实施测试计划的基石，为下一章各个激励的实现与结果的收集奠定了基础。

第五章阐述了测试用例的实现细节并分析了验证结果。本章前半部分根据验证计划表，介绍其主要测试场景和激励的实现方法，并对覆盖率计划和检查计划的实现进行了说明；后半部分对验证结果进行了分析，首先阐述了典型验证结果的构成与意义，然后分析了总的测试结果，包括仿真报告和覆盖率报表，得出了对该模块的验证工作已经充分完备的结论。

第六章对全文的工作进行了总结，分析了验证过程中的不足，并给出了改进建议。

1.3 论文研究创新点

在国内学术界，多数教研团队还在使用 FPGA 测试和定向激励测试的方法对数字芯片进行验证，而这些方法在复杂验证工作中的表现往往不尽人意。

本论文基于 UVM 验证方法学，为 APB-UART 模块搭建的验证平台，能够高效完备地完成复杂模块的验证工作，且因为 UVM 验证平台具有重用性和移植性，该代码资源还可以用来快速推动其他类似模块的验证进程。

此外，由于待测模块的特殊性，本验证平台在部分结构中采用了特殊的实现方法，如下所述：

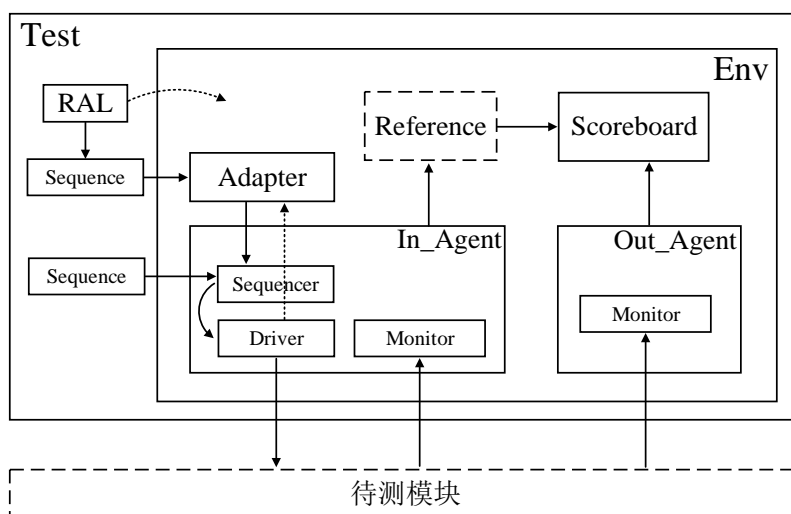
(1) 针对 DUT 中部分寄存器地址相同，必须依靠其他控制信号进行调控的情况，该验证平台使用钩子函数和回调函数实现了预警机制，当对特定寄存器进行非法访问时，会输出报错信息。

(2) 针对 modem 组件无统一时钟调控采样的问题，该验证平台在对应的组件中引入了 APB 接口，并通过事件(event)实现了定点采样操作。

(3) 针对 UART 串行传输特点以及波特率配置不同而导致的仿真结束时间不明确的问题，该验证平台使用旗语(semaphore)实现了传输数据个数的反馈，可以在测试用例中根据反馈情况及时结束仿真。

第二章 UVM 验证技术概论

2.1.1 UVM 验证平台架构



该验证平台中各个组件或类的功能阐述如下:

（2）**Monitor:** 与 **Driver** 功能相反，该组件负责监控 **DUT** 的行为。它可以将信号级的数据转化成事务级，并将其发送给后续组件，比如 **Reference_Model** 或者 **Scoreboard** 组件。

（4）**Agent**：将一些组件封装成一个容器，一个 Agent 往往对应 DUT 的某一个端口协议。Agent 可以通过配置特定的参数，实现内部组件的创建与连接的多样化，方便了验证平台的搭建。

(5) **Reference Model:** 用来模仿 DUT 的行为，实现和 DUT 相同的功能。该

组件的输出往往送到 Scoreboard 中，与 DUT 的实际输出进行比较。随着 DUT 复杂度的提升，Reference_Model 也会变得复杂，但该组件可以使用其他语言进行建模，通过接口函数与验证平台交互，更为灵活。

(6) Scoreboard: 是验证平台中负责数据比对的组件。它可以将预期数据和真实数据进行比较，从而实现验证结果的自动比对功能。

(7) Adapter: 负责实现寄存器模型(RAL, Register Abstract Layer)所使用的数据格式与 Driver 能够处理的格式之间的转换。

(8) Env: 该组件用来实现上述各个组件的实例化和连接工作。

(9) RAL: 寄存器模型，负责反应 DUT 中各个寄存器的值，给验证平台提供了一种更为便捷的访问 DUT 中寄存器的方式。

(10) Test: 该结构是验证平台的顶层，负责 DUT 和 Env 的实例化工作，并实现相关虚拟接口(virtual interface)的连接。

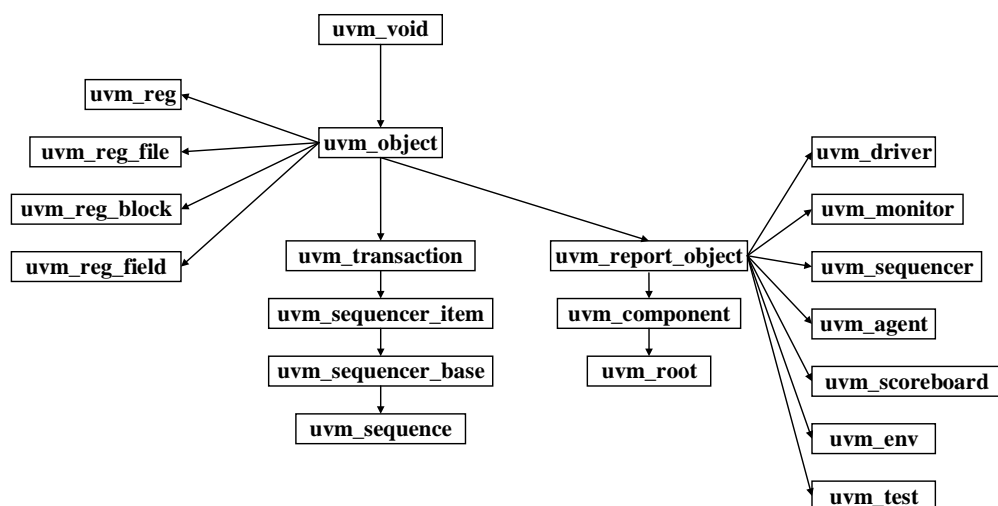
2.1.2 uvm_object 和 uvm_component 基类

验证人员在搭建 UVM 验证平台时，所使用的组件大部分都直接继承自 UVM 原有的基类，这些基类分为 uvm_object 和 uvm_component 两种。

其中 uvm_object 是最基本的类，UVM 平台中绝大多数的类都派生自 uvm_object，包括 uvm_component 也是如此。根据面向对象语言的特性，该类的扩展性最好，但是所具备的能力较差。直接派生自 uvm_object 的类一般不会对仿真过程中长久存在。经常使用到的派生自 uvm_object 的类有 uvm_sequence_item, uvm_sequence, uvm_reg_item, uvm_phase, 以及 uvm_reg 相关的类等。

uvm_component 类也派生自 uvm_object，但是它具有很多独有的特性。当一个 uvm_component 的类被实例化后，它将作为验证平台的一部分，长久的存在于仿真过程中。uvm_component 还可以通过指定 parent 参数，形成一种树形结构。另外，uvm_component 还具有 phase 自动执行的功能，这是 UVM 验证平台能够正常运转的关键。

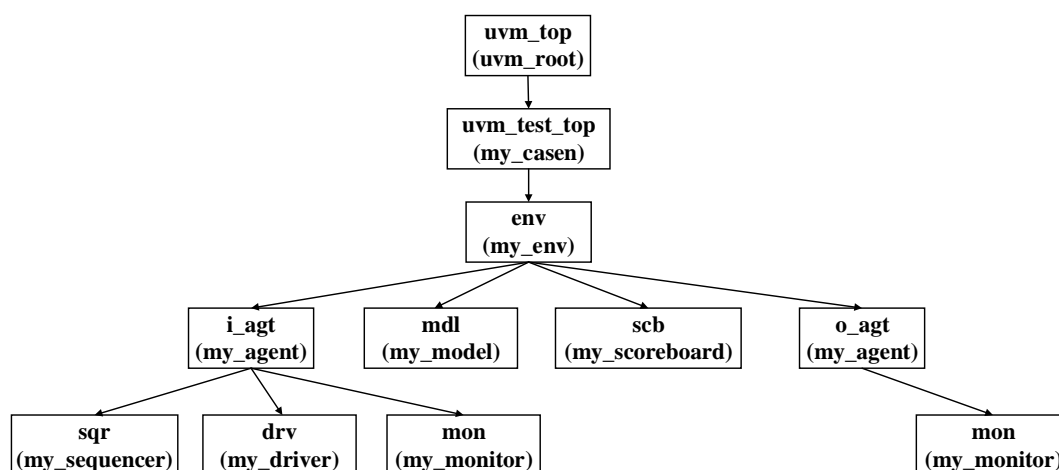
UVM 中常用类之间的派生关系如图 2-2 所示^[17]。

图 2-2 UVM 常用类的派生关系图^[17]

2.1.3 UVM 树形结构

在上一节中提到，由 `uvm_component` 例化出的组件可以长久的在仿真过程中存在，并且能够形成树状的组织结构从而方便管理。树形结构是 UVM 验证平台能够运行的基础，后续章节中提到的 `phase` 机制，`objection` 机制以及 `config_db` 机制等都与树形结构有着紧密关联。基本的 UVM 树状结构如图 2-3 所示^[18]。

其中 `uvm_top` 由 `uvm_root` 实例化而来，它是树结构的唯一根。唯一根的存在可以保证整个验证平台中只有一棵树状结构，这样就保证了所有的组件都是 `uvm_top` 的子节点。

图 2-3 UVM 基本树结构^[18]

2.2 UVM 基本运行机制

UVM 的运行除了需要基本的组件外，还要一定的机制支撑，才能够使得各个

组件有条不紊的相互配合、仿真的起始和结束时间明确。下面简要介绍 UVM 平台中最基本的几种运行机制。

2.2.1 Phase 机制

在 UVM 方法学的指导下搭建的验证平台拥有众多组件和类, 如何使各个组件能够在特定顺序下有序执行是十分重要的问题。UVM 通过 phase 机制实现这种操作。

UVM 中的 phase 按照功能可以分为 3 类: 建立阶段(build)phase, 在该阶段验证平台完成创建和配置工作; 运行阶段(run)phase, 在该阶段验证平台主要执行一些消耗仿真时间的工作, 比如运行各个测试用例; 收尾阶段(cleanup)phase, 该阶段主要实现对仿真结果的收集和报告工作。

UVM 中的 phase 如图 2-4 所示, 各个 phase 会按照从上到下的顺序自动执行。对于不同的 uvm_componnet 中含有相同 phase 的情况, UVM 采用了深度优先原则。

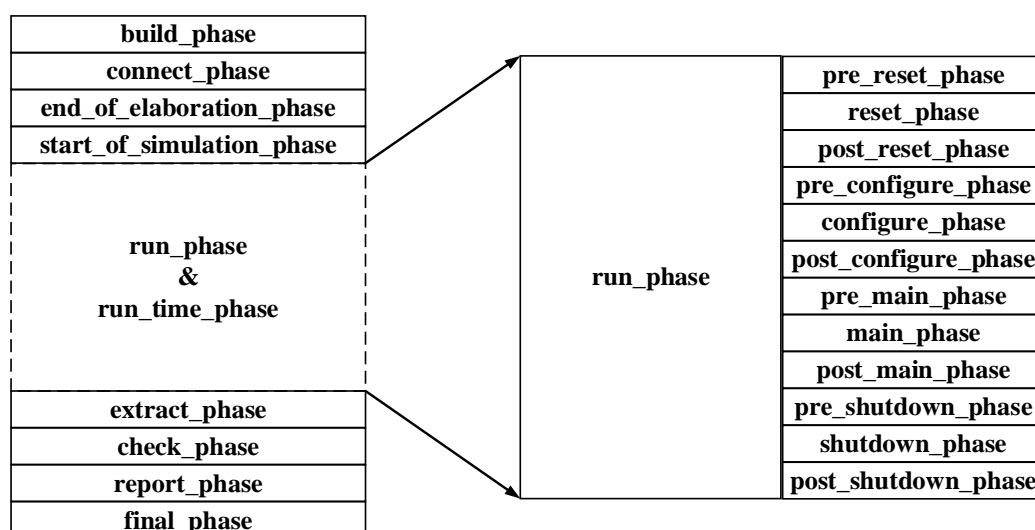


图 2-4 UVM 中的 phase 汇总

在 phase 依次执行的过程中, 从宏观上看, 各个 phase 之间没有时间间隔, 不过在微观上看, 单个组件对 phase 的执行是有一定间隔的。这种情况的发生主要是因为不同组件中同一个 phase 运行时间不同, 当某个组件中执行时间短的 phase 执行完毕后, 那么该组件尚不能立刻执行下一个 phase, 而是需要等待其他组件中的同一种 phase 都执行完毕后, 才能开始继续执行^[19]。

此外, 在验证功能较复杂的模块时, UVM 可以打破 phase 顺序执行的规则, 实现 phase 之间的跳转, 以便模拟对应的工作场景。比如当 DUT 在执行过程中突然收到了 reset 信号, 那么在 DUT 进入复位状态的同时, UVM 验证平台也应该直

接跳转到 `reset_phase`，进行一些清理工作，等待复位完成。

在实际的验证流程中，验证人员并不需要在所有的 `phase` 中都编写代码，按需求使用相关的 `phase` 即可。该套验证方法学之所以提供众多的 `phase`，一方面是为了充分细化仿真平台的执行流程，另一方面为了便于其他验证方法学向 UVM 迁移。

2.2.2 Objection 机制

UVM 平台使用 `objection` 机制来控制仿真的结束。在验证平台中可以配合使用 `raise_objection` 和 `drop_objection`，从而提起或撤销 `objection`。当所有被提起的 `objection` 全部被撤销后，仿真将会结束。需要注意的是，即便某些组件中有 `forever` 语句，仿真也并不会因此进入无限循环，`objection` 机制结束仿真的优先级高于 `forever`。

在 UVM 平台的执行过程中，每进入一个 `phase`，都会有专门的变量来统计该 `phase` 内所有组件提出的 `objection` 以及被撤销的 `objection` 数量，只有当全部的 `objection` 被撤消后，仿真平台才会进入下一个 `phase`。在全部的 `phase` 执行完毕后，验证平台会自动调用 `$finish` 系统函数结束仿真。

由于 `run_phase` 和 `run_time_phase` 是并行执行的，所以当 12 个 `run_time_phase` 中有 `objection` 被提起后，`run_phase` 即便不提起 `objection` 也会同时运转，直到之前的 `objection` 被全部撤销。

2.2.3 Config_db 机制

由于 UVM 验证平台是由多个组件或者类构成，这些组件之间有时需要进行参数传递，所以 UVM 库中引入了 `uvm_config_db` 机制。`Uvm_config_db` 提供了两个方法：`uvm_config_db::set` 和 `uvm_config_db::get`，分别用来发送和接收数据。

`Set` 函数和 `get` 函数的使用方法如下列代码所示。其中 `T` 表示待传递参数的类型，一般是虚拟接口或者配置文件(configuration object)；`cntxt` 和 `inst_name` 共同决定了要接收参数的对象的路径；`field_name` 表示待传参数的名字，`set` 函数和 `get` 函数中的 `field_name` 应该一致，才能保证正确的接收到参数数据；`value` 则是需要传递的参数或者数据，其类型应与 `T` 保持一致。

```
1. void uvm_config_db#(type T = int)::set(uvm_component cntxt, string inst_name, string field_name, T value);
2. bit uvm_config_db#(type T = int)::get(uvm_component cntxt, string inst_name, string field_name, ref T value);
```

2.3 寄存器模型

寄存器模型是 UVM 中至关重要的一部分，如果没有寄存器模型，那么验证平台对于 DUT 内寄存器的访问方式将十分有限，对 DUT 运行状态的把控也会变得更为复杂。在验证过程中，scoreboard 或者其他验证组件经常需要了解当前时间某个寄存器的值，以此来调控激励的输入或者进行数据的比对。如果不使用寄存器模型，那只能通过启动 sequence 的方式，给 DUT 的交互端口特定的地址和操作信号，获取寄存器的值。这种方法不仅十分繁琐，而且在某些情况下会提高测试用例的编写难度。

寄存器模型其实就是 DUT 内各个寄存器的“拷贝文件”，而且它还能够根据 DUT 的变化，及时更新内部属性值，达到和 DUT 内部寄存器同步的目的。这就要求寄存器模型有和真实寄存器类似的结构、相同的属性以及一定的访问方法，这样才方便理解和使用。

UVM 中的寄存器模型分为了三个级别^[20]：uvm_reg_field、uvm_reg 以及 uvm_reg_block，如图 2-5 所示。

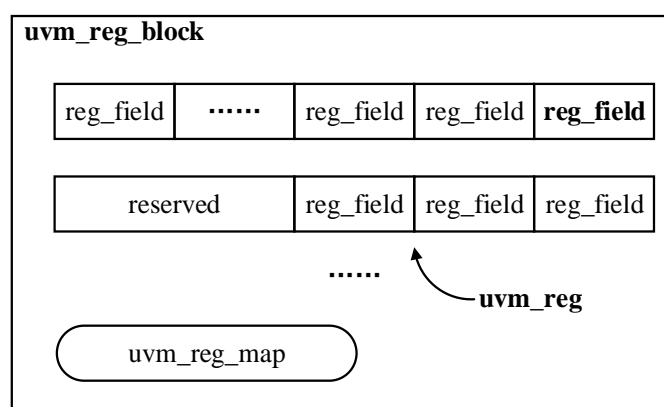


图 2-5 UVM 寄存器模型图

uvm_reg_field 是寄存器模型中最基本的单位，对应着真实寄存器中的域。uvm_reg_field 可以拥有不同的比特长度，定义不同的名称和属性；uvm_reg 与 DUT 中的寄存器相对应，每个 uvm_reg 至少包含一个 uvm_reg_field；uvm_reg_block 则属于比较大的单位，其中可以包含多个 uvm_reg，也可以包含多个 uvm_reg_block，block 一般可以涵盖模块级 DUT 中的全部寄存器。

每个寄存器模型除了拥有用来表征寄存器的三级结构外，还需要 uvm_reg_map 来实现地址上的映射。DUT 中的每个寄存器都会有对应的地址，用来给外部访问提供依据。同样，在寄存器模型中，验证人员要在 uvm_reg_map 里添加各个 uvm_reg

的访问地址，为寄存器模型前门访问提供相关的信息。

为了方便管理，UVM 在寄存器模型中规定了四种属性来描述寄存器信息。也就是说，对于 DUT 中的每个寄存器，寄存器模型都会使用四个变量从不同的角度对寄存器的值进行记录。这四种属性分别为 `m_reset`、`m_mirrored`、`value` 以及 `m_desired`。

其中 `m_reset` 用来储存寄存器的复位值，`m_mirrored` 用来存储寄存器的镜像值，`m_desired` 用来存储期望值，这三个变量都属于本地变量，不能够直接从外部进行访问，需要使用寄存器模型提供的方法，常用的访问方法如表 2-1 所示。而 `value` 属性属于 `public` 类型，可以从外部直接访问，主要用于功能覆盖率或者域的随机等方面。

表 2-1 寄存器属性常用方法

属性 方法	<code>m_reset</code>	<code>m_mirrored</code>	<code>m_desired</code>	<code>value</code>	DUT
<code>read ()</code>	N.A.	Set read value	Set read value	Set read value	Read the reg
<code>write ()</code>	N.A.	Set write value	Set write value	Set write value	Write the reg
<code>set ()</code>	N.A.	N.A.	Set value	Set value	N.A.
<code>get ()</code>	N.A.	N.A.	Return value	N.A.	N.A.

最后，由于寄存器模型中存储数据的格式和在验证平台中使用的格式不一致，所以需要一个 `adapter` 模块负责格式转换，使寄存器模型能够和 `sequencer` 实现数据交互。寄存器模型在验证环境中的应用结构如图 2-6 所示。

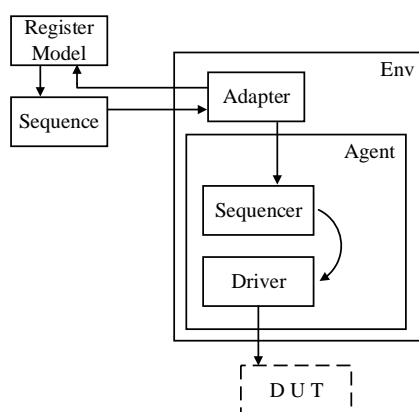


图 2-6 寄存器模型使用图

2.4 TLM 通信机制

为了实现 UVM 验证平台中各个组件之间的通信，SV 语言和 UVM 方法学提

供了很多方案。除了前述章节提到的 config_db 机制,还有 SV 语法中的全局变量、事件(event)、旗语(semaphore)和信箱(mailbox)^[21]。由于 UVM 验证平台的复杂性,这些方法都有其固有的弊端,因此需要一套更为先进的通信机制——TLM 机制。

TLM 机制通过通信端口、通信连接方式、通信操作方法三个部分,形成了完整的数据传输路径。其中常见的通信端口有 port、export 和 imp 三种;通信连接方式一般使用 connect 在上层结构中指定需要进行连接的端口;通信操作方法一般有 put、get、transport 三种,其中 put 操作对应放入数据, get 操作对应获取数据, transport 则相当于施加一次 put 操作后再进行一次 get 操作。

TLM 端口的常见连接方式如图 2-7 所示。

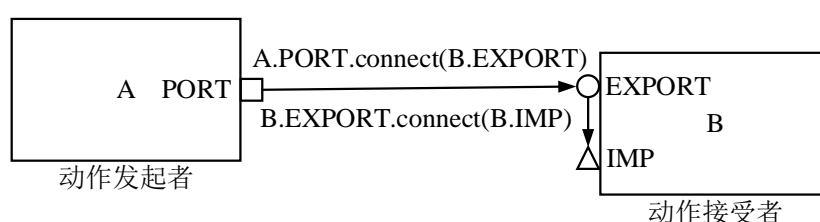


图 2-7 TLM 端口连接方式

此外,UVM 还提供了 analysis_port 端口、analysis_export 端口以及 uvm_analysis_fifo 等结构。这些结构对应的是广播操作,即一对多通信,不存在阻塞和非阻塞的区别^[22]。

2.5 覆盖率机制

由前述可知,在使用受约束的随机化方法时,由于打入的激励是在一定范围内的随机值,所以在对 DUT 进行验证的过程中,需要有特定的指标来表征验证进度。相关指标应该涵盖以下两方面:该 DUT 的所有功能是否都被验证完全,该 DUT 的代码是否被全部执行过。只有解决了这两个问题,才能够明确知道验证过程何时达到收敛。

在实际的验证结构中,验证平台需要给 DUT 打入各种激励,从而激活 DUT 的对应功能,并使其产生特定的输出。在这样的前提下,上述两个问题可以分别用功能覆盖率和代码覆盖率解决。功能覆盖率主要是检测 DUT 的预期功能是否被全部实现并能够被观测到;代码覆盖率主要是检测输入激励是否使 DUT 执行了全部的代码以及特定结构等。

常见的推动验证收敛的流程如图 2-8 所示。

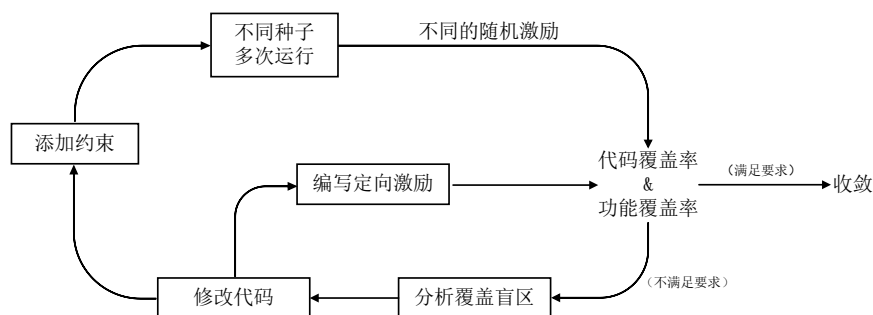


图 2-8 验证收敛流程图

2.5.1 代码覆盖率

1. 代码覆盖率的意义

代码覆盖率最初应用在系统软件测试领域，后来该方法被引入到验证领域，并且往往会被集成到验证工具中，不需要手动去编写相关代码。

代码覆盖功能会自动检测待测模块的源代码中有哪些代码被执行，又有哪些代码没有被现有激励覆盖到。

代码覆盖率达到 100% 只能表示 RTL 代码全部被执行过，并不能够代表该 DUT 没有错误^[23]。如图 2-9 所示，激励触发了 DUT 中的某处错误代码，但是由于缺少相应的选择信号配合，所以 DUT 的输出侧不能够显示出这个错误，从而造成了错误的遗漏。有研究表明，当验证平台达到了 90% 的代码覆盖率时，可能只有 54% 的功能对应的结果可以被观测到^[24]。

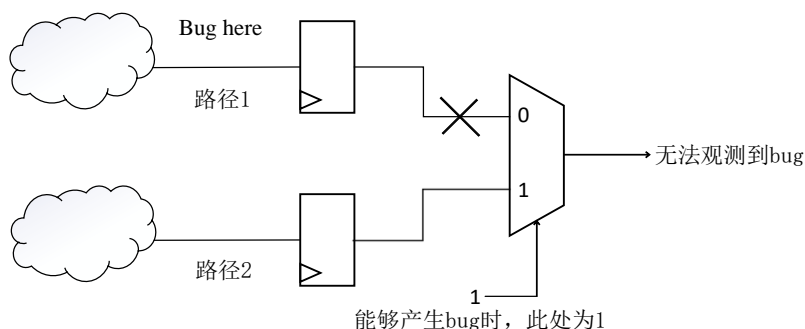


图 2-9 Bug 无法被观测到的情况

另外，代码覆盖率也无法表征有多少预期的功能被实现，如果 DUT 代码中根本没有实现某一项功能，即便代码覆盖率达到 100%，验证也是不充分的。

2. 代码覆盖率的种类

代码覆盖率包含不同的种类，分别对代码进行不同维度的覆盖检测。其中常见的代码覆盖率有行覆盖率(line coverage)、条件覆盖率(condition coverage)、翻转覆

盖率(toggle coverage)和状态机覆盖率(FSM coverage)。

行覆盖率用来表示在本次验证中,源代码中有哪些行的代码被执行过,哪些没有被执行。在执行完程序后得到的覆盖率报表中,未执行过的代码将会被标记出来,并参与到覆盖率的统计中。通过观看报表,一方面可以知道在现有的激励中是否存在了什么遗漏,从而造成了部分代码未被执行,另一方面也可以去分析是否因为代码错误导致某行代码不能够执行。此外,对于一个 IP 而言,在某些应用场景中可能并不会使用到它的全部功能,即不需要执行其全部代码,这种情况下就可以过滤掉无关代码,排除冗余项,从而提高代码覆盖率。

条件覆盖率用来统计多种条件共同作用下,判定语句对应的各种条件组合是否都被验证过。对于有多个条件的判定语句,使得判定语句结果为 True/False 的条件组合可能有很多种。表 2-2 所示为双重条件的判定组合,这种情况就需要去保证所有可能的组合都被激励触发过,才能说明验证足够充分。

表 2-2 双重条件判定组合

判定语句: $(x > y) \ \&\& \ (a > b)$	
条件组合	结果
$x > y \quad a > b$	T
$x > y \quad a < b$	F
$x < y \quad a > b$	F
$x < y \quad a < b$	F

在复杂的电路设计中,有些条件组合可能存在天然的互斥性。比如上述例子中,由于信号之间复杂的互连关系,可能在 $a < b$ 时 $x < y$ 永远不会成立,那么第四种组合永远不会出现。这种情况导致的条件覆盖率缺失是正常的,可以作为冗余项被排除。

翻转覆盖率用来统计寄存器或者信号线上每位翻转的次数,在验证过程中,一般需要检测到 0 到 1 翻转和 1 到 0 翻转才能够说明该比特被充分访问。翻转覆盖率往往用来检测 IP 互连时候信号线的连接关系,或者检测独热信号的完备性。

状态机覆盖率主要针对 DUT 源码中的状态机部分,它能够统计出进入每种状态的次数,某一状态向相邻状态转移的次数等。通过状态机覆盖率,可以直观的看出状态机运行过程是否有遗漏或者非预期的跳转。

2.5.2 功能覆盖率

1. 功能覆盖率的意义

功能覆盖率主要用来检测 DUT 是否正确的实现了预期的功能。由于受约束的随机化测试方法的使用,在验证过程中,验证平台可以轻易的产生大量的随机测试激励,省去了手工编写激励的繁琐步骤。但这种仿真方法的弊端也很明显,如果没有一个合适的表征系统,那么验证人员无法知道这大量的随机激励是否覆盖了所有预期的功能点,因此,功能覆盖率应运而生。

由于功能覆盖率相关的代码不会内嵌在工具中,所以验证人员需要人为制定覆盖计划并手动去编写相关的覆盖率组件,因此存在覆盖点与预期功能没有完全对应的风险。

2.功能覆盖率的种类与实现

在验证功能时,往往需要监测设计模块内部的状态或者总线上的数据,这时候经常使用覆盖组的方法。覆盖组一般用来监测总线或者寄存器的数值,并且能够在规定的采样点进行采样。根据采样结果,便可以知道仿真中是否出现了预期的状态,有没有实现预期的功能。验证平台中通常使用 SystemVerilog 的相关语句来实现该结构。

在 SystemVerilog 中,功能覆盖率通过覆盖组(cover group)、覆盖点(cover point)、仓(bin)实现。三者相互配合,共同表征覆盖模型,主要解决想要采样哪些数据、何时对数据采样以及对数据中的哪些特定值进行采样等问题。

其中覆盖组为最外围的结构,一个覆盖组中可以包含多个覆盖点,每个覆盖点又对应一定数量的仓。默认情况下,仓的数量为 2^N ,其中 N 为对应覆盖点的变量个数。在实际使用中,往往需要手动去设置仓的数量和值,一方面为了提高覆盖效率,同时也可以使用非法仓(illegal_bins)等方法对非预期的采样值进行一定的预警。

覆盖组的运行机制与 class 相似,在声明完覆盖组之后,还需要进行实例化才可以使用该覆盖组,并且一次声明之后可以多次实例化。

除此之外,覆盖属性(Cover Property)也是经常使用的方法之一。该方法常用于检测多个事件之间的运行关系是否正确。比如握手信号中的请求和应答是否合理,或者总线信号的变化是否符合协议等。验证平台中一般使用断言语句和 cover property 语句来实现属性的覆盖^[25]。

需要注意,功能覆盖率采样的数据应该是实际 DUT 所处的状态,而不是控制部分期望 DUT 所处的状态。这就对 cover group 的采样点提出了要求,即一般不在验证平台的控制部分进行采样,而是在 DUT 的输出侧进行采样。

当然,对于较复杂的设计,验证平台往往不能完全模拟 DUT 工作时的各种场景。越是复杂的 DUT 的验证工作,需要投入的资源就越多。因此需要在验证资源

和验证程度之间找到一个比较合适的临界点。验证是没有尽头的，只要做到有足够的概率保证芯片能够在工作场景中不出现错误，就可以认为这次芯片的验证工作是充分的。

2.6 本章小结

本章主要介绍了 UVM 方法学基础：首先讲述了 UVM 验证平台的架构、UVM 组件的树形结构等，这是验证平台能够运行的前提条件；然后讲述了 UVM 验证平台的运行机制，包括 `phase` 机制、`objection` 机制和 `config` 机制等；最后又对其他重要的概念进行了介绍，包括寄存器模型、TLM 机制和覆盖率机制等。这些概念在后续章节中会被多次使用。

第三章 APB-UART 模块及验证计划

3.1 APB 总线协议

APB(Advanced Peripheral Bus)总线协议属于 AMBA 总线协议的一种。与 AMBA 协议中的其他总线协议相比, APB 总线协议具有功耗低, 互连信号简单等特点, 适用于低性能的外设模块, 不支持流水线模式^[26]。

3.1.1 APB 总线与 SOC 系统

典型的基于 AMBA 总线的 SOC 系统结构如图 3-1 所示。这种 SOC 结构是在高速总线协议和低速总线协议的互连支持下, 将工作在不同频率的各个系统模块进行整合, 最终协同处理器完成运算工作。其中高速总线往往采用 AHB 或者 ASB 协议, 而低速总线一般使用 APB 协议, 两种总线之间使用转接桥(bridge)模块完成协议转换。APB 协议作为外围总线协议, 将为各个低速模块提供通信接口, 包括 UART、SPI、PIO 等。

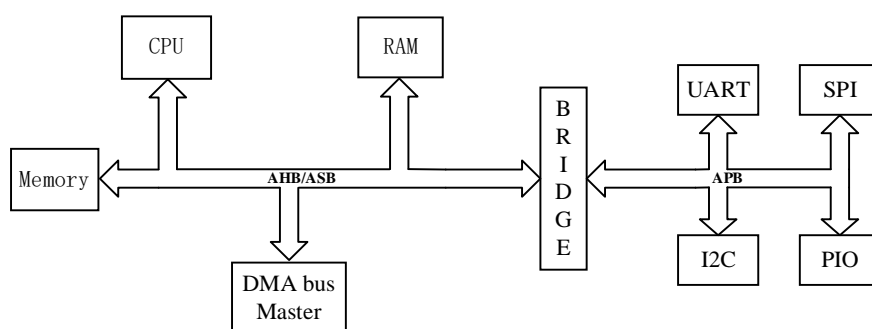


图 3-1 典型基于 AMBA 总线的体系架构

3.1.2 APB 总线传输的实现

该论文所用的 APB 协议属于 2.0 版本, 后续介绍也仅限于该版本。

APB 协议状态转移图如图 3-2 所示, 主要分为 3 个状态: IDLE 状态, SETUP 状态和 ENABLE 状态。其中 IDLE 态为 APB 状态机的初始状态; 当有数据需要进行传递时, 通过使能 PSEL、PADDR、PWRITE 等信号进入 SETUP 状态; 在 SETUP 状态停留一个时钟周期后, APB 总线进入 ENABLE 使能状态, 进行数据的读写操作。

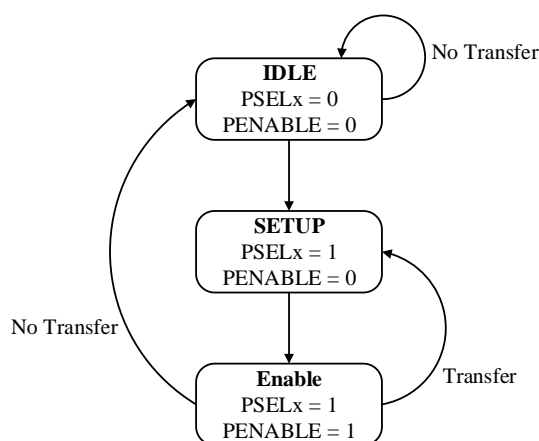


图 3-2 APB 协议状态转移图

APB 写传输时序如图 3-3 所示。在 T1 时，APB 内状态机进入 IDLE 状态。当 T2 时刻来临时，地址信号、读写控制信号和选择信号会在上升沿被触发，APB 状态机进入 SETUP 状态。一个周期后，PENABLE 信号被拉高，此时状态机进入 ENABLE 状态，进行数据的写入。如果没有后续的操作，一个周期后 PSEL 和 PNEABLE 都被拉低，代表此次传输结束。而 PADDR 和 PWRITE 等信号可以保持不变，以减少信号翻转从而降低功耗。

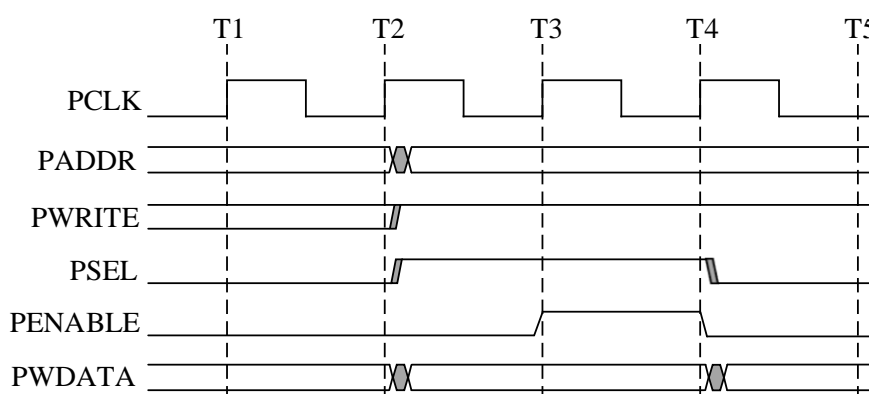


图 3-3 APB 写传输时序图

APB 总线的读传输操作与上述类似，在 T3 开始后，有效数据会被放到 PRDATA 信号线上，供 CPU 等模块读取。

3.2 UART 总线协议

UART(Universal Asynchronous Receiver-Transmitter)总线协议是串行异步通信中常见的协议。该协议规定使用单根的接收信号线和发送信号线将数据一位一位

的进行传输，成本较低，适用于远距离的低速通信。

3.2.1 UART 信号传输格式

UART 协议中数据的传输以**字符/帧**为单位，**每两个字符之间的时间间隔是可以配置的**。数据的传播速率由**波特率**决定，即**每秒钟传送的二进制数个数**。

在 UART 传输协议中，由于没有时钟信号线的参与，因此需要对每帧的数据格式进行严格的限制。收发双方对协议中可配置的部分应拥有共识，这样才能够避免数据的错误传输。

UART 协议规定的帧格式如图 3-4 所示，可以看到，每一帧的数据都可以划分成起始位、数据位、校验位、停止位等部分。

当 UART 器件没有传输数据时，信号线电平为高。当准备传输数据时，首先将信号线电平拉低，代表着数据开始传输，然后根据 UART 模块配置的不同，传输 5 至 8 位数据。为了提高远距离传输的纠错能力，还可以通过配置相关的寄存器，在数据位后面附带校验位。当所有数据传输完成后，信号线电平被拉高，代表着这一帧数据传输的结束。



图 3-4 UART 协议传输格式

3.2.2 UART 收发逻辑

UART 的**发送逻辑负责驱动发送端口**。当发送端口对应的移位寄存器中被载入数据后，发送逻辑首先拉低发送端口电平，表示起始位开始。经过一个周期后，发送逻辑将数据从移位寄存器中依次取出，并在发送端口以高低电平的形式表现出来。该逻辑还可以根据配置添加不同类型的奇偶校验位以及停止位。

UART 的**接收逻辑**往往工作在比波特率高的时钟频率下，**一般使用 8 倍频采样或者 16 倍频采样**。在每个时钟的上升沿，接收逻辑都会去检测接收端口的电平信息，如果发现**接收端口被输入了低电平且持续时间超过了半个波特率信号周期**，则认为检测到了起始位信号，并开始接收数据，否则认为起始位无效。当后续数据接收完后，接收逻辑应该将此次**接收的一帧数据**放到对应的**接收 FIFO** (First Input First Output) 中，并产生一定的信号告知系统已获得有效数据。

较复杂的接收逻辑还会对接收端口的信号进行监测，如果发现接收到的信号不符和协议规则，则产生警报。接受逻辑中也可以给接收 FIFO 设置水线功能，当接收到的有效数据达到一定数量时，产生中断信号告知 CPU 模块。

3.3 APB-UART 模块介绍

3.3.1 模块架构及功能介绍

该款 APB-UART 模块可以实现串行数据和并行数据的转换，并符合 UART16550 协议，发送和接收逻辑都分别拥有 16byte 深度的 FIFO 用来存储数据。CPU 可以通过 APB 总线访问该模块，进而间接的实现 UART 串行数据的访问。

该模块支持的功能有：(1)UART 数据格式可配置功能，包括数据长度、奇偶校验、停止位长度等；(2)波特率可配置功能；(3)字和半字访问功能；(4)有效起始位检测功能；(5)硬件流控功能；(6)全双工功能；(7)中断功能；(8)低功耗模式；(9)loop 模式；(10)DMA 和 IRDA 功能。

该模块的整体结构如图 3-5 所示。

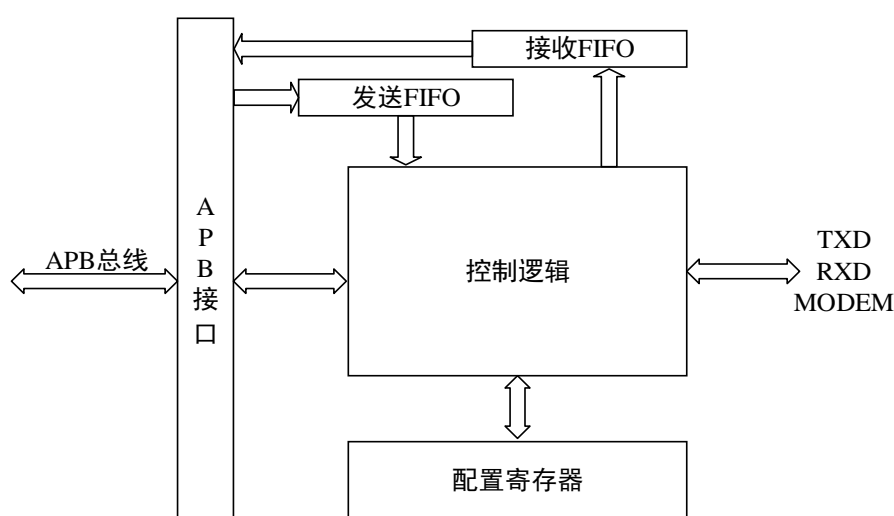


图 3-5 APB-UART 模块结构图

在进行数据发送操作时，需要发送的数据通过 APB 总线被写入到该模块后，会暂时存储在发送 FIFO(TFIFO)中。发送逻辑模块会在特定的时刻从 TFIFO 中取走数据，并添加上起始位、奇偶校验位以及停止位，形成一个完整的数据帧。最后发送逻辑会将该数据帧放入到发送端口一侧的移位寄存器中，按照特定的波特率将数据串行移位，实现数据发送的功能。当没有数据发送时，发送端口保持为高电平。

在进行数据接收操作时，串行数据通过接收端口后会进入到接收逻辑模块。该模块对数据进行格式检测后，会将其中的起始位、校验位以及停止位移除，并把剩下的数据暂时存储在接收 FIFO(RFIFO)中，等待 CPU 访问。

此外，该模块还提供了多种状态寄存器以及中断寄存器供外部访问，以便 CPU 能够充分了解模块的运行状态。

3.3.2 模块端口介绍

模块端口是该模块和外部进行信息交互的渠道，清楚掌握每个端口信号的作用及相互关系是分析待测模块的重要步骤。表 3-1 列出了该 APB-UART 模块的全部信号及对应功能，后续章节引用各个端口名称时均以此表为准。

表 3-1 APB-UART 模块端口信号列表

信号名称	输入/输出	功能描述
TXD	输入	UART 串行数据发送端口
RXD	输出	UART 串行数据接收端口
PCLK/PRESET	输入	APB 时钟和复位信号
PWRITE	输入	APB 读写控制信号
PSEL	输入	APB 从机选择信号
PENABLE	输入	APB 使能信号
PADDR[3:0]	输入	APB 地址信号
PRDATA[31:0]	输出	APB 读数据信号
PWDATA[31:0]	输入	APB 写数据信号
RTS	输出	(request to send)数据发送请求信号
DTR	输出	(data terminal ready)数据终端设备就绪信号
OUT1	输出	用户定义输出信号
OUT2	输出	用户定义输出信号
CTS	输入	(clear to send)数据发送中止信号
DSR	输入	(data set ready)数据通信设备就绪信号
DCD	输入	data carrier detect 信号
RI	输入	ring indicator 信号
Int	输出	中断信号

3.3.3 模块寄存器介绍

寄存器是设计模块中至关重要的部分，清楚掌握每个寄存器的功能才能够对该模块有更深入透彻的认知。该 APB-UART 模块所有的寄存器以及对应功能介绍如表 3-2 所示。

表 3-2 APB-UART 模块寄存器列表

名称	偏移地址	类型	复位值	功能介绍
RBR	0x00	RO	8'h0	该寄存器用于缓存接收到的数据
THR	0x00	WO	8'h0	该寄存器用于缓存待发送的数据
IER	0x04	RW	8'h0	该寄存器用于使能各种中断
IIR	0x08	RO	8'h01	该寄存器用于指示中断类型
FCR	0x08	WO	8'h0	控制 FIFO 开启、清零、水线等
LCR	0x0c	RW	8'h0	控制数据格式以及 DLAB 等功能
MCR	0x10	RW	8'h0	控制 modem 的四个输出信号值
LSR	0x14	RO	8'h60	该寄存器用于指示数据收发状态
MSR	0x18	RO	8'hf	该寄存器用于指示 modem 输入信号情况
SCR	0x1c	RW	8'h0	无功能
BRDL	0x00	RW	8'h0	该寄存器为波特率配置系数的低 8 位
BRDH	0x04	RW	8'h0	该寄存器为波特率配置系数的高 8 位
WA	0x20	RW	32'h0	该寄存器用于以字为单位访问
HWA	0x28	RW	16'h0	该寄存器用于以半字为单位访问
MISCC	0x24	RW	8'h0	该寄存器控制低功耗、IRDA 等功能
XY	0x2c	RW	32'h0	该寄存器控制时钟分频

该 DUT 的部分寄存器内包含了多个域，每个域用来实现不同的功能。这类功能较复杂的寄存器需要进行单独说明，如下所述：

(1) LSR 寄存器用来表示接收和发送数据的状态。LSR[0]有效表示 RFIFO 中有可读数据；LSR[1]有效表示 RFIFO 中发生了接收数据溢出情况；LSR[2]、LSR[3]以及 LSR[4]有效表示接收逻辑中的特殊情况，分别对应奇偶校验位错误、停止位错误和接收到 break 信号；LSR[5]有效表示 TFIFO 中没有数据；LSR[6]有效表示 TFIFO 以及移位寄存器中都没有数据；LSR[7]表示 RFIFO 中发生了接收错误。

(2) LCR 寄存器用来控制数据格式。LCR[0]和 LCR[1]共同控制每帧数据中的数据位长度，可在 5 到 8 之间选择；LCR[2]用来控制停止位的长度，可在 1 和

2 之间选择；LCR[3:5]用来控制奇偶校验位，分别代表 parity enable、even parity enable 以及 stick parity；LCR[6]用来调控 TXD 端口进行 break 信号输出；LCR[7]为 DLAB 位，当访问波特率相关的寄存器时，需要将其置位为 1。

(3) FCR 寄存器用来配置与 FIFO 功能相关的参数。FCR[0]用于使能 FIFO，决定了该 APB-UART 模块可存储数据的深度。当 FIFO 被使能后，该模块最多可存储 16 帧数据，否则只能存储 1 帧。FCR[1]和 FCR[2]分别用来复位 RFIFO 和 TFIFO。FCR[3]用来控制 DMA 的工作模式。FCR[6]和 FCR[7]用来设置 RFIFO 的水线。其余位保留。

(4) IER 寄存器用来使能不同类型的中断。IER[0]使能接收数据可得的中断。IER[1]使能发送 FIFO 空中断。IER[2]使能接收状态异常的中断。IER[3]调制解调的中断。

(5) IIR 寄存器用来指示中断类型。当 IIR[3:0]的值为 0001 时，表示当前没有中断发生。当该值为 0110 时，表示发生了接收数据状态异常的中断。当该值为 0100 时，表示发生了与 FIFO 水线相关的中断。当该值为 1100 时，表示在过去四个周期内没有收到数据。当该值为 0010 时，表示发生了 TFIFO 空的中断。当该值为 0000 时，表示发生了调制解调相关的中断。

3.4 验证计划分类

功能验证是确保 DUT 满足设计规范的一项重要步骤，在进行功能验证之前，需要根据验证需求制定验证计划。

验证需求往往来源于 DUT 的设计需求。在用 RTL 代码实现某个模块之前，需要将模块的原始需求进行两级的转化：首先需要根据功能、功耗、性能以及可靠性等方面将原始需求划分成各个功能模块的规范，然后再根据各模块的要求，制定出具体的实现方案，方案选择一般会涵盖到数据逻辑、控制逻辑、低功耗以及时钟复位等要素，如图 3-6 所示。

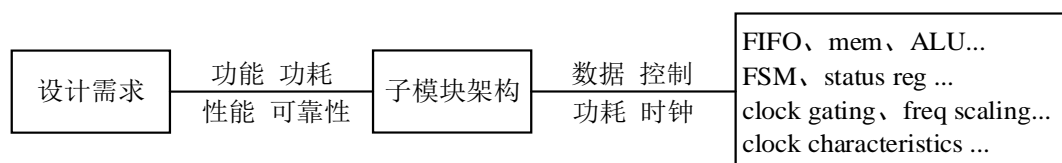


图 3-6 常见模块需求变化图

这两次转化的过程，就在原始需求的基础上，对验证需求进行了必要的扩充和约束：如果在数据逻辑中使用到了 FIFO 结构，就需要对 FIFO 的空满信号以及水线等相关特性进行验证；在控制逻辑中使用了状态机，就需要关注相关的代码覆盖

率，检测状态机状态转换是否正常；在使用 Verilog 语言时，由于其固有的特性，就需要关注模块之间连接关系是否正确。

根据上述思路，当充分掌握 DUT 的功能和结构，明确了验证需求之后，验证人员才能够制定出相关的验证计划。

在功能验证流程中，制定验证计划是十分关键的一步，在没有相对完善的验证计划前就盲目的开始验证流程是不被允许的。另外，验证计划能够决定验证平台的架构，计划中的任何纰漏都有可能对验证平台的架构产生影响，降低验证速度和验证质量。

在基本的验证思路中，制定验证计划至少需要考虑两方面，一方面是激励的产生，另一方面是与之相关的检测策略。但由于该项目采用的验证技术是覆盖率驱动的受约束的随机化验证，部分激励的产生是在某个范围内的不定值，所以还需要从功能覆盖率这个维度去证明需要检测的激励已经产生过了。

综上所述，该验证计划主要由三部分构成：测试计划(Test Plan)、覆盖率计划(Coverage Plan)以及检查计划(Check Plan)，下面分小节描述。

3.4.1 测试计划

测试计划往往和 DUT 的功能点相对应，在对 DUT 功能进行合理拆分后，便可以据此编写测试计划。测试计划可以使用表格进行简单直观的表述，每一项测试计划往往对应着相关的覆盖率计划和检查计划，是后续验证流程的指导性文件，也是验证人员在流程初期的重要产出。

在进行测试计划的编写前，需要对拟采用的验证技术有足够充分的了解：

(1) 随着设计模块复杂度的提升，当前流行的验证方法主要为覆盖率驱动的受约束的随机化验证方法。这种验证方法要求验证平台能够通过一定的组件实现数据结果的自动检测，并且通过覆盖率的收集来确定随机化的激励是否已经完备，即后续的覆盖率计划和检查计划。在制定验证计划时，应当尽量实现全局配置变量和一些测试激励的重用，从而提高验证平台的搭建速度。

受约束的随机化验证虽然大大解放了验证人员的工作量，但是对于某些验证场景，并不是必须使用这种方法。受约束的随机化验证方法主要适用于拥有以下特点的功能场景：有较大的连续状态空间；功能复杂且可能与其他功能有交互；有较多的排列和组合。而对于功能简单、只在特殊情况下发生、组合有限的验证场景，使用直接测试的方法更为简单有效。

此外，当需要对发现的错误进行定位时，受约束的随机化验证方法要比直接测试的方法更为繁琐，因为验证平台驱动的激励是随机数据，没有办法通过查阅代码

的方式去锁定特定激励，只能够通过打印信息或者检查波形来实现，这无疑会消耗更多的验证资源。

(2) 如果验证资源足够，不要仅对 DUT 进行黑盒验证，这有可能遗漏部分验证场景。在上节的叙述中，我们可以知道 DUT 内部功能模块的结构对于验证计划的编写是有指导意义的。只有充分掌握了各个结构之间是如何实现控制信号与数据信号的传输后，才能够清楚认识到这个 DUT 的弱点，从而避免验证计划的不完备。

此外，如果验证结果中出现了非预期的问题时，使用这种验证思路也能够及时的定位问题，对验证平台进行修改或迭代。

在编写测试计划时，应当严格遵循以下原则，这样才能够尽量减少计划的重复和遗漏，加快验证进程。

- (1) 测点完全，如果验证资源足够充分，则 DUT 的全部功能都应当被验证。
- (2) 描述精细，对于测试场景或者测试点的描述要清晰、准确。
- (3) 精细适度，测试点既要覆盖全面，又要避免过度验证。
- (4) 有先有后，在资源有限的情况下，需要对待测功能进行优先级划分。
- (5) 过程持续，验证计划也需要根据项目进度不断的进行迭代。

3.4.2 覆盖率计划

由第二章可知，覆盖率主要分为代码覆盖率和功能覆盖率。代码覆盖率的收集是由仿真软件自动完成的，所以在制定覆盖率计划时不需要考虑。但功能覆盖率是由验证人员手动实现的，因此需要提前制定相关计划。

验证人员需要基于测试计划，对每个测试用例的覆盖方式进行考虑，决定使用覆盖属性还是覆盖组去覆盖。

当使用覆盖组去覆盖时，还考虑以下问题^[27]：

(1) 哪些值是重要的，是采集全部数据还是部分数据。在编写覆盖组时，使用默认仓是比较方便的，但这有可能造成验证资源的大量浪费，尤其是和多位总线、时钟频率等相关的验证。

比如在该待测模块中，需要对波特率配置功能进行验证。虽然波特率产生模块可以根据相关寄存器的值产生多种波特率，但只需要覆盖几种常用的波特率即可。否则对不会在应用场景中出现的波特信号进行验证，将会大大滞后验证进度。

(2) 覆盖点/组之间是否有相互关系。要考虑覆盖点/组之间是否会相互影响，如果多个覆盖点/组之间不是完全独立的关系，在编写覆盖组或者进行交叉覆盖时，就需要将其考虑进去。

(3) **边界情况以及邻近边界情况**。在模块设计的过程中，边界情况往往容易出现非预期的问题，是**验证的重点考虑对象**。比如单笔数据读写时地址或者数据全 0 全 1 的情况，多笔数据传输时连续读写操作或者背靠背操作等，都是容易出问题的边界情况。同样，临近边界情况也需要特殊注意，因为在设计过程中，为了实现边界的安全性而采取的措施，可能会增加临近边界情况出错的风险。

(4) **是否有非法的情况**。某些情况下，如果覆盖组采集到了非预期的值，说明 DUT 发生了严重错误。验证人员在编写覆盖组时，需要添加对应的非法仓，有利于及时发现相关错误。

(5) **何时去采样**。除了少数情况下，可以用时钟边沿作为采样点自动采样，一般都需要手动编写采样点，在特定的时刻去采集覆盖率。通常可以在相关的测试用例执行完毕，且未发生错误时采样，采样时数据应该稳定有效。常见的情况有以下几种：**使用 scoreboard 组件检测之后再发送给覆盖率收集组件进行采样；引入相关的信号线，在信号有效时进行采样；使用断言语句判定采样点，当断言有效时进行采样。**

3.4.3 检查计划

如前所述，验证计划中的**测试计划主要侧重于激励的产生，覆盖率计划则是在随机化的基础上保证产生的激励覆盖了预期的场景，而检查计划则是测试运行期间，测试平台需要进行的检查项集合。**

检查计划可以分为即时检查和基于状态的检查。即时检查就是在收集到足够的数据后，将数据和预期结果进行对比；基于状态的检查则需要额外去获取 DUT 在过去一段时间内的状态，根据状态来决定是否检查。

检查计划要检查 DUT 期望实现的功能。在合适的时刻去比对采集到的数据和预期值是否一致，而当采集到的数据和预期值不符时，验证平台应当通过预警信息或者中断仿真来给予提醒。

常见的**检查计划**主要涉及到以下几个方面：

(1) **信号检查**：DUT 通过信号线和其他模块进行交互，所以信号相关的检查是最基本的且不用深入到 DUT 内部结构的检查。在进行信号级检查时，需要注意以下几点：信号线上是否存在非法值，比如 X 或 Z；信号拉高和拉低的时刻是否正确；信号之间的相互关系是否正确。信号检查一般在底层模块中进行，因为底层模块才有机会和具体的信号线产生交互。在**信号检查中最常用的方法就是断言语句**，而且往往涉及到一定的延时，可能是若干个周期或者固定时长的延迟。在固定时长延迟中，需要注意**尽量减少使用#延时符号**，因为这种操作在不同的编译设置

下可能会产生延迟时间单位紊乱的问题。

(2) **总线协议检查**：为了使系统中各个模块之间能够正确无误的传递信息，在信号线上进行数据传输时往往需要遵循一定的协议。由于信号线的数量、工作频率、串并行需求、全半双工等要求的不同，总线协议也是多种多样的。但总线协议中往往会划分出不同的阶段，用来分别传输控制信号和数据。对总线协议的检查可以在 `monitor` 或者 `interface` 中实现，一般需要对每个阶段对应的信号状态以及各个阶段之间的前后关系进行检查。

(3) **事务级数据检查**：DUT 可以认为是数据的搬运工，它**将一定的数据转换成不同的格式，并将其在一定的条件下发送出去**。在数据转化和发送过程中，需要对数据格式和内容进行检查。事务级数据检查一般不能简单通过在底层采集数据来实现，而需要在能够处理事务级数据的组件中，比如 `scoreboard` 中，采集一个或者多个接口的数据信息，进行相互比对或者和预期值比对。

(4) **寄存器检查**：在 DUT 的设计中，需要使用寄存器来实现控制逻辑或者状态逻辑。对于**寄存器的检查，可以分为与功能相关和与功能无关**两个部分。

与功能无关的部分一般指对寄存器的复位值以及读写属性的检查，这些检查与具体的 DUT 设计相独立，具有很强的移植性。

与功能相关的部分则必须结合具体的 DUT 功能，在仿真过程中实现。DUT 中的寄存器一般可分为控制寄存器和状态寄存器：其中控制寄存器常见类型为只写、可读可写等，通过对控制寄存器的配置，DUT 可以在不同工作模式间切换；而状态寄存器一般为只读或者读后清零等，通过访问状态寄存器可以了解 DUT 当前的工作状态。

3.5 验证计划制定

前述章节已提到，在验证工作初期，验证人员需要充分掌握 DUT 的结构，对 DUT 的功能点进行细致的划分，并根据功能点使用表格的形式将测试计划罗列出来，同时规划对应的覆盖率计划和检查计划。

本项目对 APB-UART 模块采用 UVM 验证方法学，进行模块级验证。对应的验证计划表如表 3-3 所示。

其中“测试项目”一栏罗列了将要实施的测试计划。后续的“检查”和“覆盖”栏简要描述了对应检查计划和覆盖计划。

在“检查”一栏中，使用具体组件名字标注的，代表该检查计划的相关工作将在对应组件中进行；其他的则是根据仿真结果或者断言语句进行判断。

在“覆盖”一栏中，Cover Group 代表该测试计划的覆盖将在对应的覆盖组中

进行，Cover Property 代表使用断言覆盖，N.A.则代表该测试计划不考虑功能覆盖率的统计情况。

表 3-3 验证计划表

测试项目	检查	覆盖
寄存器相关检测	Test result	N.A.
基本数据发送功能检测	Tx_Scoreboard	Cover Group
基本数据接收功能检测	Rx_Scoreboard	Cover Group
硬件流控功能检测	Modem_Scoreboard	Cover Group
字/半字读写功能检测	Rx/Tx_Scoreboard	Cover Group
波特率配置功能检测	Div_Checker	Cover Group
APB 协议检测	Assert property	Cover Property
接收数据溢出场景检测	Test Result	Cover Group
接收数据错误场景检测	Test Result	Cover Group
接收 break 信号场景检测	Test Result	Cover Group
发送数据空场景检测	Test Result	Cover Group
发送 break 信号功能检测	Test Result	Cover Group
DLAB 功能检测	Test Result	Cover Group
中断功能检测	Test Result	Cover Group
FIFO 相关功能检测	Test Result	Cover Group
全双工功能检测	Rx/Tx_Scoreboard	Cover Group
Loop 模式检测	Test Result	Cover Group
低功耗模式检测	Test Result	Cover Group

3.6 本章小结

本章主要对 APB-UART 模块以及对应的验证计划进行了介绍。首先介绍了 DUT 涉及的 APB 总线协议和 UART 总线协议；然后深入到 DUT 模块内部，分别讲解了模块架构及功能、模块端口、各个寄存器功能；最后介绍了验证计划中常见的三种计划，并给出了最终的验证计划表。

第四章 UVM 验证平台设计与实现

4.1 APB-UART 验证平台总体架构

该验证平台的整体架构如图 4-1 所示，图中展示了大部分组件及其之间的连接关系，本文中各个验证组件的名称均以下图为主。另外，为了使框图结构清晰，该图略去了覆盖率收集组件及部分互连细节。

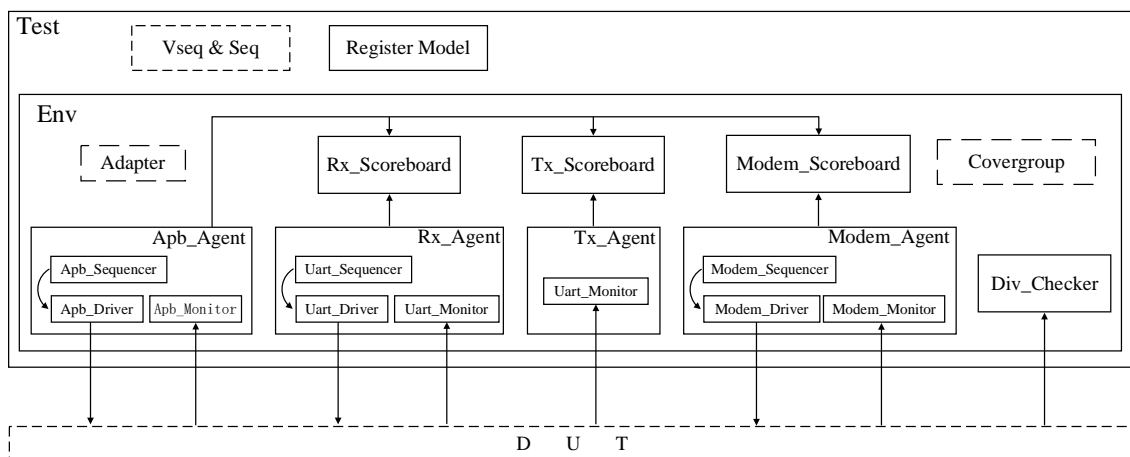


图 4-1 APB-UART 验证平台总体框架

4.2 寄存器模型的实现

4.2.1 基本寄存器模型的实现

在本验证平台中，使用寄存器模型对 DUT 内部的 15 个寄存器进行了建模，以方便后续验证工作使用。寄存器模型的搭建和使用过程主要分为三部分：(1)根据寄存器特征，对应完成 `uvm_reg` 的建模工作，(2)构建 `uvm_reg_block`，完成 `uvm_reg` 的实例化和配置，并将地址信息添加到 `map` 中，(3)在 `test` 中实例化寄存器模型，并将其指针赋给需要访问寄存器模型的组件。

下面以 IER 寄存器为例描述寄存器模型的实现过程。

首先按照规格书的要求对 `ier_reg` 完成建模，关键代码如下。

```

1. class ier_reg extends uvm_reg;
2.   `uvm_object_utils(ier_reg)
3.   ...
4.   rand uvm_reg_field RDI_E;
5.   function void build();
6.     RDI_E = uvm_reg_field::type_id::create("RDI_E");
7.     RDI_E.configure(this, 1, 0, "RW", 0, 0, 1, 1, 0);
8.     ...
9.   endfunction: build
10. endclass: ier_reg
  
```

然后将 `ier_reg` 模型添加到 `uvm_reg_block` 中，并配置相关的地址映射信息，关键代码如下：

```
1. class uart_reg_block extends uvm_reg_block;
2.   ...
3.   IER = ier_reg::type_id::create("IER");
4.   IER.build();
5.   IER.configure(this);
6.   ...
7.   map = create_map("map", 'h0, 4, UVM_LITTLE_ENDIAN);
8.   map.add_reg(IER, 8'h4, "RW");
9.   ...
10.  IER.add_hdl_path_slice("uart_cfg.IER_pool",0,8);
11. endclass: uart_reg_block
```

最后，在 `test` 基类中实例化寄存器模型，添加后门访问路径，并将指针赋给 `config` 文件中，供下层组件访问，关键代码如下：

```
1. uart_reg_block rm;
2. rm = uart_reg_block::type_id::create("rm");
3. rm.build();
4. rm.set_hdl_path_root("uart_tb.DUT.uart_module");
5. m_env_cfg.rm = rm;
```

4.2.2 预警机制的实现

通过本 DUT 的寄存器列表可知：**RBR、THR 和 BRDL 有相同的地址，IER 和 BRDH 有相同的地址。**但由于 `DLAB(LCR[7])` 和不同读写属性的存在，当我们通过总线对相同地址进行访问时，会操作不同的寄存器，如表 4-1 所示。当 `DLAB = 0` 时，APB 总线对 `0x0` 地址进行操作，那么实际访问的寄存器是 `THR` 与 `RBR`；当 `DLAB = 0` 时，对 `0x4` 地址进行操作，那么将会访问寄存器 `IER`；当 `DLAB = 1` 时，对 `0x0` 和 `0x4` 地址进行操作，那么对应访问的的寄存器是 `BRDL` 与 `BRDH`。

表 4-1 地址相同的寄存器列表

寄存器名称	属性	地址	DLAB
RBR	RO	0x 0	0
THR	WO	0x 0	0
IER	R/W	0x 4	0
BRDL	R/W	0x 0	1
BRDH	R/W	0x 4	1

如果验证人员按照经典的方法去构建寄存器模型，那么在使用时由于 `DLAB` 的错误设定，或者验证人员对 `DLAB` 的值有所疏忽，进而访问到非预期的寄存器，

就会对验证过程造成阻碍，浪费一定的验证资源。

本文提出了一种解决该问题的方法：为传统的寄存器模型配套一种预警机制，当验证人员想要访问的寄存器和 DLAB 的设定有冲突时，验证平台能够输出预警信息。本验证平台采用了 Hook(钩子)函数和 Callback(回调)机制来实现这套预警机制^[28]。

1. Hook 函数和 Callback 机制

Hook 函数可以理解为一位在特定地点处理特定事件的程序，各种各样预先设定好的钩子函数被挂载在 UVM 中。这些钩子函数可以加工处理事件，也可以不做任何事情，但是它们在验证平台的执行过程中总是会被调用到。

寄存器模型的基类提供了多种钩子函数(pre_write、post_write、pre_read、post_read 等)，这些函数会在寄存器操作的不同阶段被调用。如果验证人员不去赋给这些函数具体的任务和操作，那么它们不会处理任何事情。也就是说这些钩子函数在特定的时间和地点被调入到程序中且一定会被执行，但是执行内容由验证人员来填补。

Callback 机制是继承自 C 语言的概念，是指将某个函数的指针作为参数传递给另外一个函数，当这个指针被用来调用其所属的原函数时，就称其为回调函数。这种机制可以将调用者和被调用这分离，使得代码更为灵活。与钩子函数不同的是，回调函数并不会随着系统函数的执行而自动被执行。

寄存器模型的基类中也提供了多种回调函数：pre_write, post_write, pre_read, post_read, post_predict, encode 和 decode 等。验证人员可以通过继承 uvm_reg_cbs 来构建自定义的回调函数，并且一定要将该函数注册到对应的寄存器或者域中。一个寄存器/域可以对应多个回调函数，但只有经过构建并注册的函数才能够成功执行。

2. 使用 Hook 和 Callback 构建预警机制

本验证平台通过使用 Hook 函数中的 post_write、post_read 和 Callback 函数中的 post_write 为寄存器模型提供了一种预警机制，以便解决由于寄存器串扰而可能进行错误访问的问题。

首先，构造回调函数并将该函数注册到 LCR 寄存器中，并在函数中引入 RBR、THR、IER、BRDH 和 BRDL 寄存器的指针。然后，在寄存器模型中分别给这五个寄存器加入变量“EN”，并且通过 callback 中的 post_write 监视对 DLAB 写入的值并修改对应的 EN 值，从而在 LCR 和五个寄存器之间建立通信关系，让五个寄存器能够知道在当前 LCR 的配置下，自身是否可能被操作。

当 DLAB 被写入 1 时，如图 4-2 所示，BRDH 和 BRDL 寄存器中的 EN 信号

被改写为 1，表示当前使能这两个寄存器。反之则使能 IER、THR 和 RBR 寄存器。

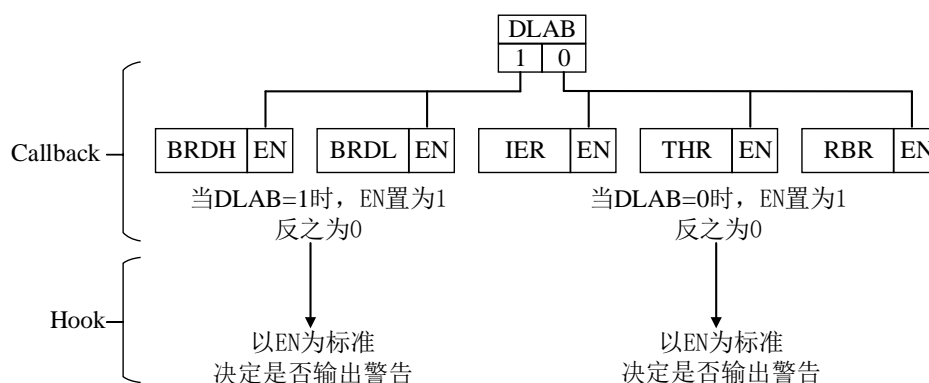


图 4-2 预警机制实现方案图

最后，分别在这五个寄存器模型中通过钩子函数(post_write 或 post_read)以及 EN 的值来决定是否需要输出预警信息。以 IER 寄存器为例，当 DLAB=1 时，IER 中的 EN 位为 0，这个时候如果对 IER 寄存器进行读/写操作，那么会紧跟着执行并打印 post_write 或 post_read 中定义的预警信息，进而实现通过验证平台来提醒验证人员误操作的目的，关键代码如下：

```

1. class ier_reg extends uvm_reg;
2.   bit EN = 1;
3.   virtual task post_write (uvm_reg_item rw);
4.   if (EN == 0) `uvm_error(...)
5.   endtask: post_write
6.   virtual task post_read (uvm_reg_item rw);
7.   if (EN == 0) `uvm_error(...)
8.   endtask: post_read
9. endclass: ier_reg

```

因为 DLAB 的复位值为 0，所以为了满足复位时的初始态，需要将 IER、THR、RBR 中的 EN 初始值设为 1，而 BRDH、BRDL 中 EN 的初始值设为 0。

4.3 Apb_Agent 关键组件的实现

Apb_Agent 组件封装并互连了与 APB 端口相关的结构，包括 Apb_Driver、Apb_Monitor 以及 Apb_Sequencer 等。该组件在 top 中通过 Apb_Interface 与 DUT 的 APB 端口信号进行连接，并使用 Apb_Seqitem 规定了传输的包格式。

4.3.1 Apb_Driver 的实现

该组件主要负责对 DUT 中的 APB 总线实现激励输入，且要保证对各个信号线的激励变化符合 APB 协议规定。在程序最开始，Apb_Driver 需要将 PSEL 和

PENABLE 驱动成低电平，等到从 Apb_Sequencer 接收到了 Apb_Seqitem 后，再将 item 中的数据按照一定的时序赋值给对应的信号线。该组件的 run_phase 执行流程如图 4-3 所示。

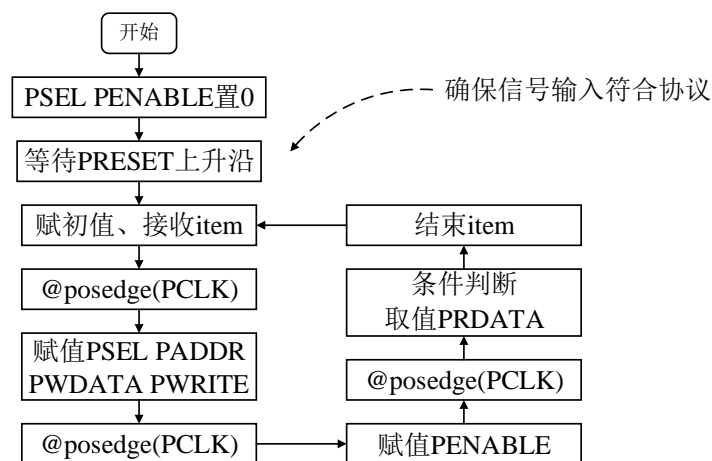


图 4-3 Apb_Driver 执行流程图

4.3.2 Apb_Monitor 的实现

Apb_Monitor 用来监测 APB 总线活动，在合适的时刻读取总线数据，并将信号级数据封装成事务级发送给其他组件。该组件 run_phase 的核心代码如下：

```

1. forever begin
2.   @(posedge APB.PCLK);
3.   if(APB.PREADY && APB.PSEL && APB.PENABLE)begin
4.     item.addr = APB.PADDR; item.we = APB.PWRITE;
5.     if(APB.PWRITE)
6.       item.data = APB.PWDATA;
7.     else
8.       item.data = APB.PRDATA;
9.     ap.write(item);
10.  end
11. end
  
```

由代码可知，该 monitor 组件将会监控 APB 总线信号，在 ENABLE 阶段捕获地址、读写操作以及数据信息并通过 uvm_analysis_port 发送出去。

4.3.3 其它类的实现

Apb_Interface 主要是将 APB 相关的信号封装起来，该类声明了 PADDR、PRDATA、PWDATA、PSEL、PENABLE、PWRITE 等 logic 类型的属性，并通过参数传递的方式获取 PCLK 以及 PRESET，是可以直接访问底层信号的类。

Apb_Seqitem 中声明了 addr、data、we 等 logic 类型的属性，分别对应存储 PADDR、PWDATA、PRDATA 以及 PWRITE 信号线的电平信息。

Apb_Sequencer 继承自 uvm_sequencer, 在 Apb_Agent 中需要将该组件和 Apb_Driver 进行连接。

4.4 Uart_Agent 关键组件的实现

Uart_Agent 负责与 UART 端口相关的协议处理工作, 其内部封装并互连了 Uart_Driver、Uart_Monitor 和 Uart_Sequencer 等组件, 分别负责数据的驱动和监测。该组件通过 Uart_Interface 与 DUT 中的 UART 端口进行了互连, 并使用 Uart_Seqitem 类规定了该组件内的数据包格式。

由于 RXD 端口和 TXD 端口都遵循一样的 UART 协议, 所以该 Uart_Agent 可以根据配置 cfg 的不同, 被分别例化为 Rx_Agent 以及 Tx_Agent。其中 Tx_Agent 由于其被动属性, 只需要 Uart_Monitor 负责采集数据即可, 不需要实例化 Uart_Driver 以及 Uart_Sequencer。

4.4.1 Uart_Driver 的实现

Uart_Driver 会接收从相关 sequence 中启动的 Uart_Seqitem, 并按照一定的规则将其转化成符合 UART 协议的数据流, 通过 RXD 端口输入到 DUT 中。

在本验证平台中, 该组件除了根据 LCR 寄存器的配置, 将正确的数据输入给 DUT 之外, 还需要拥有错误注入的功能, 即根据不同的需求将拥有特定错误的数据输入到 DUT 中, 进而观察 DUT 能否对该错误产生预期的反应。由于此 DUT 具有检测接收数据流是否有校验位和停止位错误的功能, 并将结果反映在 LSR[2:1] 寄存器中, 因此在 Uart_Driver 中需要能够产生这两种错误数据类型。

本验证平台将相关的配置参数 fe 和 pe 放在对应的 Uart_Seqitem 中, 这样就可以通过在顶层的 virtual sequence 中修改相关 sequence 的参数, 影响 Uart_Seqitem 中的 fe 和 pe 参数, 促使 Uart_Driver 按需产生拥有不同错误类型的数据, 并输入到 DUT 的 RXD 端口。

另外, 由于该 DUT 能够接收低电平信号(break 信号), Uart_Driver 需要在特定时刻将 RXD 端口驱动为低电平, 实现思路与上述类似。

该组件的 run_phase 执行流程如图 4-4 所示:

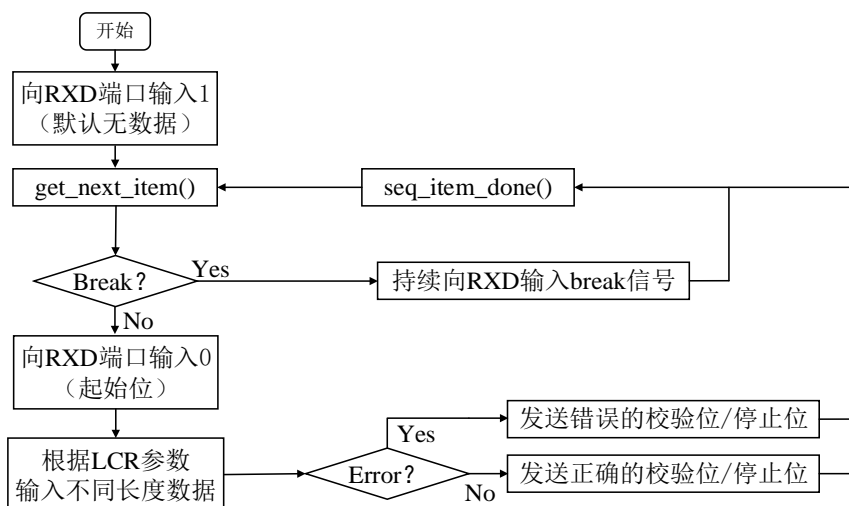


图 4-4 Uart_Driver 执行流程图

图中 LCR 参数的值由上层结构传递进来，确保了 LCR 参数与 DUT 内部的 LCR 寄存器的值保持一致。除此之外，也可以通过访问配置文件中的寄存器模型来获取 LCR 寄存器的值。

4.4.2 Uart_Monitor 的实现

由第三章可知，UART 协议的帧格式中包含了起始位、数据位、校验位、停止位等，这些数据按照一定的波特率由 UART 电路的发送端口 TXD 输出，被另一个 UART 电路的接收端口 RXD 获取。为了提高数据传输的可靠性，增强电路的抗干扰能力，UART 器件一般采用倍频采样的方法，以比数据流波特率高 N 倍(N 大于等于 1)的频率进行采样。本 DUT 电路采用 16 倍频采样策略，在 brg_16_en 使能信号的驱动下采样。

该 Uart_Monitor 可以根据 UART 协议，在正确的时刻对数据位以及校验位进行采样。此外，由于 DUT 具有注入错误检测功能，所以在 Uart_Monitor 中也应当能够实现 frame error 以及 parity error 的检测，并将检查结果放置到 Uart_Seqitem 级别的包中，供后续 scoreboard 使用。

当数据开始传输时，在波特率时钟下，如果检测到一个单位时长的低电平信号，则认为检测到了起始信号，数据流开始传输。但由于外部环境存在各种干扰，传输信号很容易产生毛刺，所以该 Uart_Monitor 还需要在 16 倍频采样下，对最开始接收到的 8 个电平进行检测。如果这 8 个电平是连续的低电平，就认为采集到了起始位，否则就认为是噪声干扰，需要重新进行采样。

起始位后面紧跟的是若干比特的数据位，对于每一个数据位，都在其中间部分进行采样。即在 16 倍频下，采样第 9 个倍频脉冲对应的信号值，以代表最终的采

样值。数据位的个数由 LCR[1:0] 的值来决定, Uart_Monitor 需要根据该值采样相应数量的数据位。

对起始位和数据位的采样示意如图 4-5 所示。

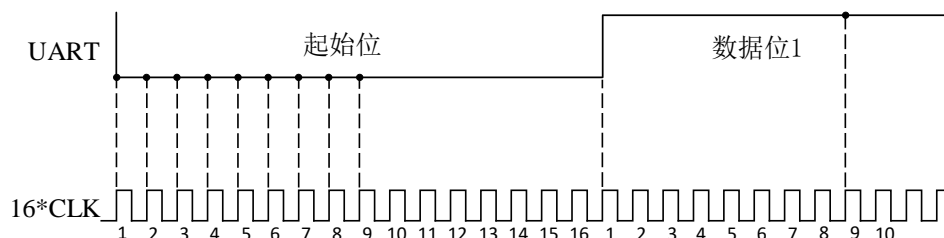


图 4-5 Uart_Monitor 采样波形图

奇偶校验位主要是作为预警信号, 用来判断数据在传输过程中是否发生了错误。在该 DUT 中, 当 LCR[3] 为 1 时, 则会在数据位后附加校验位, 校验位的采样方法与数据位类似。

在校验位采样完成后, 该组件可以检测采集到的校验位是否发生 parity error, 具体方法如下: (1) 读取寄存器模型中 LCR 寄存器的值。(2) 根据 LCR[5:3] 的值, 计算校验位的期望值: 当 LCR[5] 为 1 时, 如果 LCR[4] 为 1, 则校验位应该为 0, 否则为 1; 当 LCR[5] 为 0 时, 如果 LCR[4] 为 1, 则校验位应符合偶校验规则, 否则符合奇校验规则。(3) 将采样得到的校验位数据和预期值进行比对。

停止位标志着一串比特流的结束, 可能会出现 frame error, 其检测方法与检测起始位所使用的方法一致。

4.4.3 其它类的实现

Uart_Interface 主要封装了与 UART 端口相关的信号, 在顶层 Top 中与 DUT 的 UART 信号线相连, 并通过 config_db 的方式将该接口发送到验证环境中。由于 UART 信号线只有两条: RXD 和 TXD, 所以该 interface 只需要声明一个 logic 类型的属性 data, 在 Uart_Driver 和 Uart_Monitor 中重用即可。

Uart_Seqitem 中除了拥有必要的 data 属性用来存储对应的 UART 数据外, 还需要声明 sbe、fe、pe 以及 break_low 等参数, 分别用来传递起始位错误、停止位错误、校验位错误以及发送 break 低电平数据等信息。

Uart_Sequencer 继承自 uvm_sequencer, 在 Uart_Agent 中需要将该组件和 Uart_Driver 进行连接。

4.5 Modem_Agent 关键组件的实现

Modem_Agent 封装了与 modem 模块交互的组件和类，包括 Modem_Driver、Modem_Monitor、Modem_sequencer 等。该组件通过 Modem_Interface 虚拟接口与 DUT 中的 modem 模块的四个输入信号和四个输出信号相连，并实现了对 modem 模块的激励输入和监测功能。

4.5.1 Modem_Driver 的实现

Modem_Driver 组件主要负责对 DUT 中 modem 模块的输入端口进行激励输入，该组件从 Modem_Sequencer 中接收来自对应序列的数据包，并将 Modem_Seqitem 级别的数据转换成信号级，赋值给 Modem_Interface 中对应的属性，从而实现对输入数据的驱动。

此外，每次给 modem 输入端口施加激励后，Modem_Monitor 组件都需要对其进行采样。但由于 modem 输入信号的稳定时间不确定，所以在 Modem_Driver 运行结束后，需要触发一个 en_monitor 事件来与对应的 monitor 模块实现信息交互。实现代码如下：

```

1. event en_monitor;
2. ...
3. forever begin
4.   seq_item_port.get_next_item(req);
5.   ...
6.   seq_item_port.item_done();
7.   -> en_monitor;
8. end

```

4.5.2 Modem_Monitor 的实现

Modem_Monitor 主要用来采集 modem 模块输入输出端口的数据信息，并将采集来的信息以 Modem_Seqitem 的形式发送出去。如果只使用一个 analysis_port 端口进行数据传输，那么采集的输入和输出数据容易相互覆盖，所以在该组件中使用了两个 analysis_port 端口分别来传输两组 item，一组用来承载输入信号值，一组用来承载输出信号值。

在该 DUT 中，对 MCR 寄存器写入不同的值，会使 modem 输出不同的电平信号，而 Modem_Monitor 需要在每次对 MCR 写入新值后去采样 modem 的输出信号。因此该验证平台将 Apb_Interface 引入到 Modem_Monitor 中，这样就可以根据 APB 总线上的地址和操作信号来判断 MCR 中是否被写入了新值。如果 MCR 中被写入了新值，则 monitor 在下一个周期对 modem 模块的输出进行采样，将采到的数据通过 analysis_port 传到 Modem_Scoreboard 中。

如果发现 `en_monitor` 被激活, 则说明 `modem` 的四个输入信号已经被写入了新值。此时 `Uart_Monitor` 将这些值采样并打包后也通过 `analysis_port` 传到 `Modem_Scoreboard` 中。

4.5.3 其它类的实现

`Modem_Interface` 主要是将与 `modem` 有关的信号线封装起来, 以便于统一管理, 有利于提高验证平台的可移植性。在该接口中, 声明了 `RTS`、`CTS`、`DTR`、`DSR`、`RI`、`DCD`、`OUT1` 及 `OUT2` 等信号, 这些信号将会在顶层的 `Top` 中与 `DUT` 的对应信号一一相连, 并通过 `config_db` 的方式将该接口发送到验证环境中。

`Modem_Seqitem` 中声明了 8 比特位宽的 `logic` 变量, 用来存储 `modem` 的八根信号线上的值。

`Modem_Sequencer` 继承自 `uvm_sequencer`, 在 `Modem_Agent` 中需要将该组件和 `Modem_Driver` 进行连接。

4.6 Scoreboard 组件的实现

`Scoreboard` 主要用来实现数据的对比工作, 让 `UVM` 验证平台拥有结果自检功能。该验证平台中主要使用了四个组件进行数据的比对: `Rx_Scoreboard` 实现与 `RXD` 接收数据相关的数据比对; `Tx_Scoreboard` 实现与 `TXD` 输出数据相关的数据比对; `Modem_Scoreboard` 实现和与 `modem` 有关的数据比对; `Div_Checker` 实现了对时钟和分频相关的数据检测。这四个组件都继承自 `uvm_component`, 并使用一个或者多个 `uvm_tlm_analysis_fifo/uvm_analysis_port` 实现和其他组件的通信。

4.6.1 Rx_Scoreboard 的实现

该组件主要实现与接收功能相关的数据比对工作, 主要包括以下两方面: (1) 通过 `Uart_Driver` 写入到 `RXD` 端口的数据和由 `APB` 从 `RFIFO` 读取到的数据是否一致。(2) 在进行字和半字读写功能检测时, 对比通过 `APB` 总线读取出来的 2byte/4byte 数据和通过 `Uart_Driver` 写入的数据是否一致。

在该组件中, 通过 `uart_fifo.get()` 函数获取到 `Uart_Monitor` 发送来的数据后, 先将该数据复制到本地的队列中, 包括 `UART` 帧的数据位、表征检测到 `frame error` 时的指示信号 `fe` 和检测到 `parity error` 时的指示信号 `pe` 等数据。另外, 该组件通过执行另一个线程去接收由 `Apb_Monitor` 传来的数据, 通过其中的地址和读写信息, 进行相应的比对操作: 如果发现是对 `RBR` 寄存器的读操作, 那么便将读取的数据与队列中的数据依次进行比较, 比较的位数由 `LCR` 寄存器决定; 如果发现是对字

或半字寄存器的读操作，则将存储在队列中的数据依次取出 4byte 或者 2byte，并与通过 APB 读取出来的值进行比较。

此外，在收到 Rx_Monitor 送来的数据时，该组件还要判断此刻是否发生了 overrun 的情况：如果 LSR[1] 已经被拉高了，则说明发生了 overrun，后续数据不会被写入到 RFIFO 中，因此不需要比较后续数据。

需要注意的是，由于 LSR 和 RBR 寄存器都具有 volatile 属性，寄存器模型不能够及时的预测这两个寄存器的行为，所以不可以直接通过 reg.get() 的方式去获取这两个寄存器的值，需要通过前门访问的方式将这两个寄存器的值读取出来，再经过 Apb_Agent 结构送到 Rx_Scoreboard 中被使用。

Rx_Scoreboard 与其他组件的互连情况如图 4-6 所示：

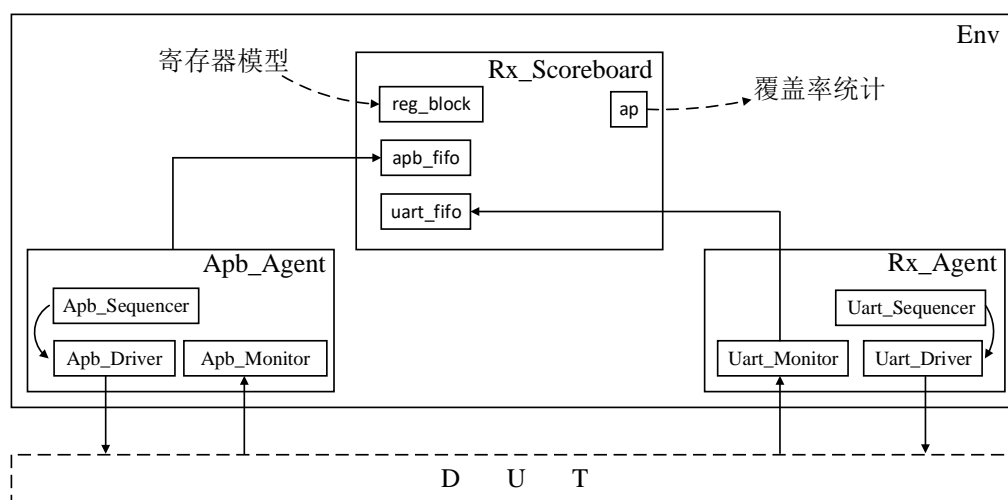


图 4-6 Rx_Scoreboard 与相关组件互连图

4.6.2 Tx_Scoreboard 的实现

Tx_Scoreboard 的主要功能是检测由 APB 总线向 THR 寄存器中写入的数据与最终通过该 DUT 的 TXD 端口串行传出的数据是否一致。该组件首先通过 apb_fifo.get() 函数获取 APB 侧的传输数据，通过地址信息过滤后，如果发现是对 THR 寄存器进行了写操作，那么将对应的数据放入到本地队列中。等从 uart_fifo 中收到 UART 数据后，依次将 APB 数据从队列中取出，与 UART 数据进行比对，比对的数据位数由当前 LCR 寄存器的值决定。

Tx_Scoreboard 和相关组件的互连关系与 Rx_Scoreboard 类似，不过 UART 数据需要从 Tx_Agent 中获取。

4.6.3 Modem_Scoreboard 的实现

Modem_Scoreboard 主要实现对 modem 功能正确性的检测，除了需要获取 modem 输入输出端口的数据外，还需要访问 MCR、MSR 寄存器。

该组件功能主要通过两个 task 实现：check_the_mcr 和 check_the_msr。其中 check_the_mcr 负责检测 MCR 值与 modem 输出值的关系，check_the_msr 负责检测 modem 输入值和 MSR 值的关系。

check_the_mcr 通过 apb_fifo.get() 获取从 Apb_Agent 传出的 APB 数据。当发现该数据是写入到 MCR 寄存器时，将数据中的 DTR、RTS、OUT1、OUT2 以及 loopback 等信息保存到本地，然后再通过 modem_fifo.get() 获取从 Modem_Agent 中传出的 modem 输出值，根据是否拥有回环模式，进行对应的数据比对。

check_the_msr 通过 modem_fifo_4in.get() 获取 modem 模块的输入值，并将该值与上一次的输入值进行对比。如果两次的值不一致，则说明对应的信号电平发生了变化，将对应信号标记之后与 MSR 寄存器的值进行比对。

Modem_Scoreboard 与相关组件的连接关系如图 4-7 所示。

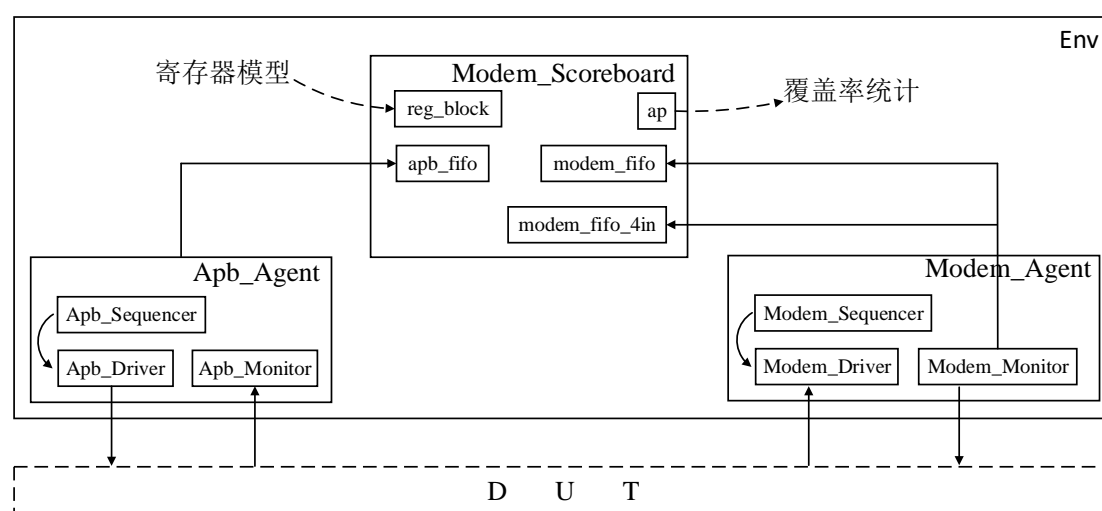


图 4-7 Modem_Scoreboard 与相关组件互连图

4.6.4 Div_Checker 的实现

该组件检测特定波特率下的时钟分频是否正确，主要指 uclk 在 BRDL 和 BRDH 的调控下是否正确的分频成 brg_en。

其中 BRDL 与 BRDH 寄存器的值和各个波形之间的频率关系如图 4-8 所示，通过验证在相邻两个 brg_en 时钟的上升沿之间有 $16 * \{BRDH, BRDL\}$ 个 uclk 时钟的上升沿，可以说明 BRDH 和 BRDL 的功能是正确的。

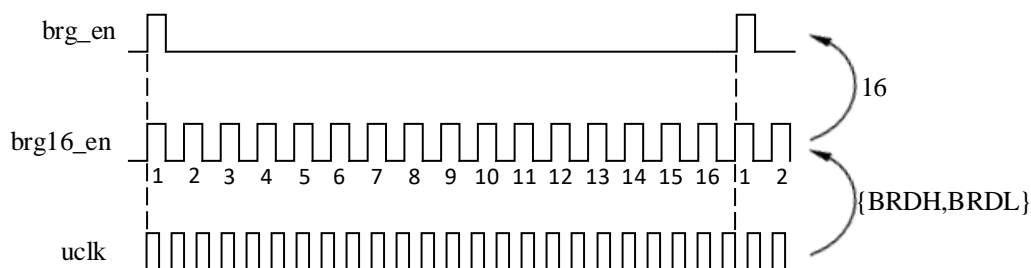


图 4-8 配置寄存器和波形关系图

该组件中使用了三个 task: `monitor_apb()`、`count_period()`和 `check_result()`。

任务 `monitor_apb()`用来监测 BRDH 或 BRDL 寄存器是否被写入新值，一旦发现两个寄存器之中有任何一个的值被更新之后，便将新值记录下来，并在名为 `sem_baud_updata` 的 semaphore 中放入一枚 key。该 semaphore 将用于判断是否重启线程。

任务 `count_period()`则是用来记录 uclk 上升沿的个数，每当一个上升沿来临时，其内部的 `clk_count` 就自增 1。

任务 `check_result()`则用来比对结果，内部的两个变量分别用来存储在 `brg_en` 前后两个上升沿时采样的 `clk_count` 数值，取差值后与 {BRDH,BRDL} 进行比对，从而得出检测结果。

每当发现 `sem_baud_updata` 中有 key 时，说明有新的寄存器值被写入，即波特率将会发生改变，此时需要重启三个进程，关键代码如下：

```

1. forever begin
2.   fork:check_fork
3.     monitor_apb();
4.     count_period();
5.     check_result();
6.   join_none
7.
8.   sem_baud_updata.get();
9.   disable check_fork;
10. end

```

4.7 其它组件及模块的实现

除了上述组件外，该 UVM 验证平台还有很多其他的组件和类支撑着平台的运作，下面对关键的模块进行简要介绍。

Env 组件继承自 `uvm_componnet`，主要在其 `build_phase` 和 `connect_phase` 中完成下层验证组件的实例化和互连工作。其中实例化工作主要使用 `create` 或者 `new` 语句，互连工作则有以下三种方式：使用 “=” 复制互连；使用 `connect` 互连；使用

`config_db::set` 函数实现数据共享。

`Env_cfg` 类继承自 `uvm_object`，主要负责封装配置参数及一些通用函数。该验证平台的 `env_cfg` 中声明了四个配置类型的指针，分别指向 `Apb_Agent`、`Tx_Agent`、`Rx_Agent` 和 `Modem_Agent` 的配置类。所以 `Env` 只需要从上层模块获得 `Env_cfg` 后，即可得到大部分配置信息，并将其传递下去。这种配置信息层层传递的方式提高了验证平台的灵活性和重用性。此外该类中还拥有指向寄存器模型的指针，供下层组件调用。`Env_cfg` 中还拥有一些常用的函数，比如用于等待固定周期时钟的函数 `wait_for_clock(int n = 1)`，获取中断信号电平高低的函数 `get_interrupt_level` 等。

`Test_base` 类为所有测试用例的基类，主要实现对下层模块的创建和互连工作。包括各个配置文件的创建以及 `Env_cfg` 指针的连接，寄存器模型的创建与指针连接，虚拟序列指针的连接等。后续验证流程所使用的绝大部分测试用例都继承于 `Test_base`，这大大减少了重复性工作，是面向对象语言的一大优势。

4.8 本章小结

本章主要介绍了该 UVM 验证平台的具体实现。采用总分思路，首先从宏观的角度给出了验证平台的总体架构框图，然后再对各个模块进行详细说明。主要包括寄存器模型，与 APB、UART 以及 modem 相关的 `agent` 组件，`scoreboard` 组件以及其他模块。这些组件和模块是支撑测试激励运行的重要结构，充分了解它们的构成和互连情况后，才能更透彻的理解下一章验证计划的实现过程。

第五章 验证计划实现与验证结果

5.1 验证计划实现

验证计划的实现和验证平台的搭建之间会相互影响，需要有一个不断迭代的过程。在第四章验证平台的基础上，本节重点讲述验证计划的实现细节，包括测试计划、覆盖率计划以及检查计划。

5.1.1 测试计划的实现

本节对一些重要测试计划的实现进行了说明，每个测试计划都会对应的一个测试用例，每个测试用例会实例化并启动一个虚拟测试序列(virtual sequence)，然后在该虚拟序列中对多个 sequence 进行操作。由于该验证平台使用的测试用例以及序列的作用可以通过其名称判断，故不在一一罗列。

这样操作一方面是为了使验证框架更具有层次性，方便一些特殊情况的实现；另一方面在需要多个 agent 以及 sequence 配合的场景中，虚拟序列是必须被使用的结构。

每个测试用例中除了调用虚拟序列外，还会添加适当的辅助性代码以调控仿真结束时间或者实现结果检测功能。

5.1.1.1 寄存器相关检测

数字电路往往是由组合逻辑和时序逻辑构成，而时序逻辑的核心是寄存器，寄存器类型和功能正确与否直接决定了一个模块能否正常工作。因此，寄存器相关的验证是数字验证中至关重要的一步，往往也是验证过程的第一步。

在标准的验证流程中，设计人员和验证人员会拿到两份相同的 Spec，分别从不同的角度对其进行实现，然后验证人员通过仿真结果来检测 DUT 的设计是否与 Spec 一致。寄存器检测的目的则是要保证 DUT 中对寄存器的实现符合 Spec 规范，以便及时发现问题，减少后续验证工作的负担。

本验证平台通过 reg_access_test 测试用例启动对应的虚拟序列，并在其中使用寄存器模型的前门/后门访问对寄存器的复位值、寄存器类型以及寄存器内串扰进行了检测。

1. 寄存器复位值检测

本文第二章已经讲述过 UVM 中寄存器模型的概念，验证人员通过寄存器模型记录各个寄存器的复位值，而设计人员一般通过判断复位信号的电平高低来给寄存器赋初值。

当对寄存器的复位值进行验证时，该测试用例首先通过对应序列给 DUT 打入复位信号以让各个寄存器的值变成复位值，然后读取 DUT 中各个寄存器的当前值，最后将该值与寄存器模型中的 `m_reset` 进行比较，如果两者一致，则说明 DUT 中寄存器复位值是正确的。

以 IER 为例，验证过程如图 5-1 所示。首先使用 `get_reset()` 函数来获取 `m_reset` 的值，然后调用前门访问的 `read` 函数读取 DUT 中 IER 寄存器的复位值，最后将两者进行比较，如果有差别则输出报错信息。

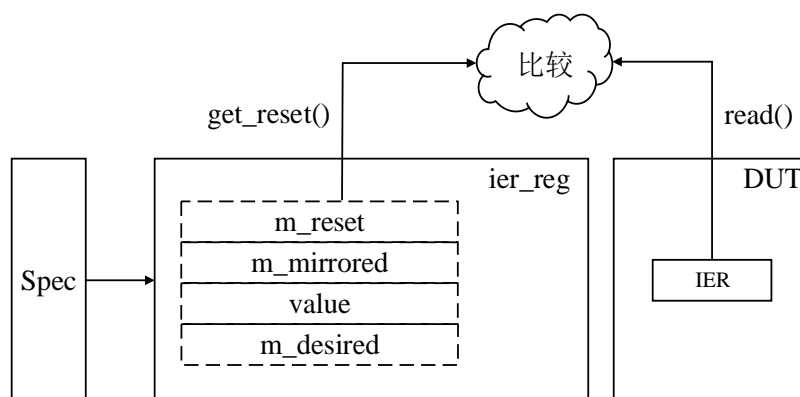


图 5-1 寄存器复位值检测示意图

对于 WO 类型的寄存器(比如 THR)，由于无法通过总线访问去获取该寄存器的值，所以该测试用例使用后门访问，直接通过硬件路径去获取此类寄存器的值。

2. 寄存器类型检测

为了模拟数字电路中多种多样的寄存器，UVM 寄存器模型提供了 25 种寄存器类型，以满足用户在各种场景下的需求。在该 DUT 中涉及到了以下几种类型：可读可写(RW)、只读(RO)、只写(WO)、读后清零(RC)。

(1) RW 类型寄存器检测

RW 类型即可读可写类型，对该类寄存器进行写操作会更新寄存器的值，进行读操作不会影响寄存器的值。

该测试用例采取如下方式检测此类寄存器：首先向寄存器写入特定的数值，然后读取寄存器内的值并与刚才写入的数值进行比较，最后将该数据取反，重复上述过程。如果该寄存器是 RW 类型，那么每次比较的结果均应一致。

(2) RO 类型寄存器检测

RO 类型的寄存器只可以进行读操作，且读操作不会影响该类寄存器内的值。

该测试用例采用如下方式检测此类寄存器：首先向寄存器内写入特定的数据，然后读取寄存器的值，最后将上次写入的数据取反，重复执行写入和读出操作。如

果该寄存器为 RO 类型，那么每次读出的值都是该寄存器的复位值，与写入的值无关。

（3）WO 类型寄存器检测

WO 类型即只可以进行写操作，对此类寄存器进行写操作会更新寄存器内的值。

该测试用例检测此类寄存器采用的方法和检测 RW 类型寄存器类似，但是由于不能通过总线读取 WO 类型的寄存器的值，所以使用后门访问的方式直接从该寄存器对应的硬件路径中获取。

（4）RC 类型寄存器检测

RC 类型寄存器即读清零类型，对该类寄存器进行写操作不会改变寄存器内的值，对其进行读操作会将该寄存器清零。

该测试用例采用如下方法检测此类寄存器：首先通过后门访问给该寄存器写入特定的数据，然后通过前门访问读取该寄存器的值，最后再次执行前门访问读取该寄存器的值。如果该寄存器为 RC 类型，那么第一次读出来的数据应该和之前写入的数据一样，第二次读出来的数据应该为 0。

3. 寄存器内串扰检测

当一个寄存器由多个比特构成时，就可能因为代码编写错误而导致寄存器内不同比特之间发生串扰，即在只想修改某一位值的同时也修改了其他位的值。

该测试用例采取以下方式检测此类错误：首先读取某寄存器的值，将值的第一位取反后，写入到该寄存器中，然后再次读取这个寄存器的值并和上一次写入的值进行比较，最后检测读取的值和写入的值是否一致。如果发现这两个值一致，则说明对该寄存器的第一位写入数据不会影响寄存器内其他位的值。

使用相同的方法遍历其他位，可以证明该寄存器中的各个比特之间没有相互串扰。需要注意的是，由于 DUT 中寄存器复杂度不高，没有对写入 0 和写入 1 进行区分。

5.1.1.2 基本数据发送功能检测

该 DUT 可以将通过 APB 总线写入的数据转化成串行数据后，按照 UART 协议从 TXD 端口输出。每一笔串行数据的格式可以通过 LCR 寄存器来设定，包括有效数据长度、停止位长度、奇偶校验格式等。输出数据流的频率与波特率有关，所以需要提前配置 BRDH 和 BRDL 寄存器来设定波特率。

发送功能的检测主要包括格式和内容两方面，需要保证任意写入 THR 寄存器的数据都可以按照固定格式正确的由 TXD 端口输出。由于部分格式的检测需要依

据 TXD 端口的信号变化，所以本验证平台在 Uart_Monitor 中检测奇偶校验位和停止位格式，在 Tx_Scoreboard 检测内容以及其他格式的正确性。

在检测过程中有以下几点需要注意：

(1) 因为 APB 端口侧和 UART 端口侧的数据流动速率有差别，连续写入到 THR 的数据会存储在 TFIFO 中，所以在通过 Apb_Driver 向 THR 打入数据的时候需要注意 TFIFO 的空满状态。该测试计划对应的测试用例通过特定序列读取 LSR[5] 的数值去判断 TFIFO 的空满状态，然后再使用 write_THR_seq 序列打入数据。

(2) 需要提前对相关寄存器进行配置：通过 write_BRDL_H_seq 序列来配置 BRDL 和 BRDH 寄存器产生特定波特率信号；通过 write_FCR_seq 序列配置 FCR 寄存器来使能 FIFO，使得连续多次写入到 THR 寄存器的数据可以暂时存储在 TFIFO 中。

另外，当验证平台采集完所有数据并实现了后续验证工作后，仿真才能结束。所以需要明确 Uart_Monitor 采集完所有数据的时刻。

该测试用例在 test 中提起 objection，并配合 semaphore 来决定何时撤销 objection 以结束仿真，如图 5-2 所示。实现过程如下所述：

(1) Uart_Monitor 每采集完一帧的数据后，便向其内部 semaphore 中放入一枚 key，来表征 TXD 端一共发送了多少帧的数据。

(2) 将该 semaphore 的指针复制到 test 中，并通过 config_db 得到由相应 virtual sequence 传递过来的参数 no_tx_char，该参数反映了传输数据的预期个数。

(3) 使用 semaphore.get(no_tx_char) 语句，只有当 Uart_Monitor 采集到全部数据后，该语句才会解除阻塞，进而执行后续程序，控制结束仿真的时间。

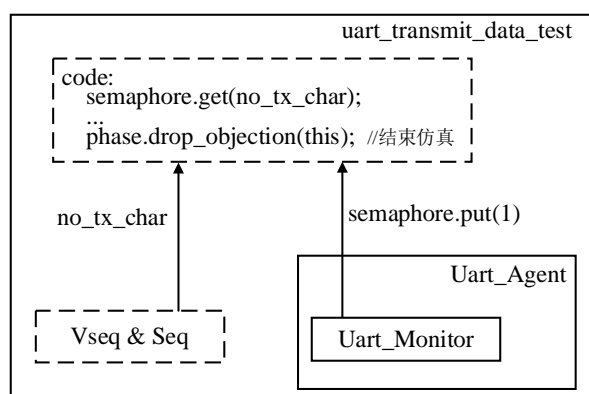


图 5-2 基本发送功能中控制仿真结束示意图

5.1.1.3 基本数据接收功能检测

该 DUT 通过 RXD 端口接收符合 UART 协议的串行数据，将这些串行数据按

照特定格式放入到 RFIFO 中，并通过相应的中断提醒 CPU 可以获取数据。而 CPU 再通过 APB 总线读取 RBR 寄存器的值，从而依次获取存储在 RFIFO 中的数据。

该测试计划对应的测试用例先调用了相关序列进行寄存器的配置，然后使用 Uart_Driver 向 DUT 的 RXD 端口打入若干帧的数据流，当 RFIFO 中被写入数据后，通过相关序列将数据读取出来，逐帧与 Uart_Driver 打入的数据进行对比，从而实现对接收数据内容正确性的检测。与上节发送功能检测一样，UART 接收数据格式的检测也在 Uart_Monitor 中实现。

此外，通过 driver 组件驱动的数据格式应与 DUT 中 LCR 寄存器规定的格式一致，这样才能够保证 DUT 中接收逻辑获得的数据拥有正确的格式。因此该测试用例在对应的 virtual sequence 中对相关参数进行随机化，将该参数的值写入到 LCR 寄存器中，同时将其复制到 Uart_Seqitem 里。这样在 Uart_Driver 获取到 item 数据时，便可以根据其中的格式参数驱动数据。

该测试用例调用 read_RBR_Seq 序列读取 RFIFO 中的数据。当 RFIFO 中的数据就绪后，LSR[0]将会拉高。因此在该序列会持续监测 LSR 寄存器的值，如果发现 LSR[0]为高电平，那么便通过 RBR 寄存器读出 RFIFO 中的数据。

另外，由于通过 Uart_Driver 写入的数据数量是由相关序列中的参数决定，因此该测试用例将相关参数发送给了 read_RBR_Seq 序列，用来判断是否从 RFIFO 中读取到了足够数量的数据，进而调控结束仿真的时间。

5.1.1.4 硬件流控功能检测

如第三章所述，该 DUT 的 modem 模块有四个输入端口和四个输出端口，分别为 CTS、DSR、DCD、RI 和 RTS、DTR、OUT1、OUT2。这些信号主要用于两个 UART 器件之间的握手通信。

RTS 为输出信号，用来表示设备是否准备好接收数据。CTS 为输入信号，一般与其他器件的 RTS 端相连，当该信号有效时，应停止发送数据。DTR 和 DSR 分别表示数据终端设备和数据通信设备准备就绪。在回环模式中，OUT1 和 RI 对应，OUT2 和 DCD 对应。

在该 DUT 中，MCR 寄存器可以控制四个输出端口的信号，MSR 寄存器可以反应四个输入端口信号的变化。当某个输入端信号改变后，MSR 寄存器会将对应比特置高。

对硬件流控功能的检测分为以下两部分：

1. 普通模式检测

在普通模式检测中，需要验证四个输入信号的变化能够影响 MSR 状态寄存器，

且 MCR 寄存器的值能够控制四个输出信号的电平。MSR 和 MCR 寄存器的功能如表 5-1 所示。

本验证平台在 modem_basic_test 测试用例中调用对应的序列，给 modem 的四个输入端口施加随机激励。每次打入激励后，Modem_Monitor 都会采集当前驱动的激励值并送到 Modem_Scoreboard 中。Modem_Scoreboard 会将最近两次的激励数据保存下来进行比对，并根据对比结果去检查 MSR 寄存器中对应比特值是否符合协议要求。

表 5-1 MSR 寄存器和 MCR 寄存器功能表

MSR 寄存器		MCR 寄存器	
Bit 0	CTS 输入信号变化时，置 1	Bit 0	该位为 1 时 DTR 输出 0，反之输出 1
Bit 1	DSR 输入信号变化时，置 1	Bit 1	该位为 1 时 RTS 输出 0，反之输出 1
Bit 2	RI 输入信号由低变高时，置 1	Bit 2	该位为 1 时 OUT1 输出 0，反之输出 1
Bit 3	DCD 输入信号变化时，置 1	Bit 3	该位为 1 时 OUT2 输出 0，反之输出 1
Bit 4	loop 模式下与 RTS 相同	Bit 4	loop 模式开启/关闭
Bit 5	loop 模式下与 DTR 相同	Bit 5	保留
Bit 6	loop 模式下与 RI 相同	Bit 6	保留
Bit 7	loop 模式下与 DCD 相同	Bit 7	保留

对于输出值检测，该验证平台调用 write_MCR_seq 改写 MCR 寄存器的值，那么对应的输出信号应该发生变化。然后 Modem_Scoreboard 模块会获取通过 APB 总线写入的 MCR 值以及 modem 模块的输出值，将两者进行对比从而完成该模式的验证。

2.loop 模式检测

该 DUT 模块拥有 loop 功能，由 MCR[4]来控制。当开启 loop 功能后，MSR 寄存器的后四位需要满足特定的关系：MSR[4]的值等于 RTS 端口的输出值，MSR[5]的值等于 DTR 端口的输出值，同理，MSR[6]和 OUT1、MSR[7]和 OUT2 也满足同样关系。

该验证平台首先将 MCR 的第四位配置为高电平以开启 loop 功能，然后通过 APB 总线随机写入 MCR 值，此时 modem 模块的输出和 MSR 寄存器的后四位应该相应发生变化，最后在 Modem_Scoreboard 中比较 modem 模块的输出值和 MSR 后四位值从而完成该模式的验证。

5.1.1.5 字/半字读写功能检测

该 DUT 通过访问地址 0x20 实现字读写功能。当 APB 总线对地址 0x20 进行一次写操作时，就可以向 TFIFO 中写入 32bit 数据，即占用深度为 4byte 的 TFIFO 空间。DUT 会将这 4byte 数据分为四帧，按照 UART 格式输出。同理，当 APB 总线对地址 0x20 进行一次读操作时，会读出 RFIFO 中 4byte 的并行数据。

该 DUT 通过访问地址 0x28 实现半字读写功能，该功能与字读写功能类似，只是读写的数据量为 2byte。

通过字读写功能读取数据，本质是将 RFIFO 中的多个寄存器的数据同时读出，这些数据是通过 RXD 端口写入到 DUT 中的。所以验证该功能的方法和验证基本接收功能类似。首先，该测试计划对应的测试用例启动若干个序列去配置相关的控制寄存器，然后启动 `uart_rx_seq` 序列向 RXD 端口写入足够多的数据，之后调用 `word_access_seq` 序列和 `half_access_seq` 序列分别读取 RFIFO 中连续 4byte 或者 2byte 的数据，最后在 `RX_Scoreboard` 中将读取出来的数据与通过 RXD 端口写入的数据进行比较。

通过字读写功能写入数据，本质是将数据写入到 TFIFO 中，这些数据会通过 TXD 端口发送出去。所以验证该功能的方法和验证基本发送功能类似。首先，该测试序列配置好 XY、BRDH、BRDL、DIV 等寄存器的值，然后启动对应的 `word_access_seq` 序列通过 APB 总线向地址 0x20 写入若干数据，最后 `Uart_Monitor` 将会采集 TXD 端口传出的数据并发送到 `Tx_Scoreboard` 中进行比较。

在上述过程中，同样需要控制仿真时间。该测试用例采用的方法与检测基本发送功能时使用的方法类似，只是 `Uart_Monitor` 中期望采集的帧的数量为 $4*w$ ， w 为写入的字数量。

同理，可以采用类似方法检测半字读写功能。

5.1.1.6 波特率配置功能检测

该 DUT 通过 `uart_baudrate` 模块和 `uart_divxy` 模块，产生满足一定波特率的使能信号。这两个模块对应的控制寄存器分别为 BRDL、BRDL 和 XY。模块框图如图 5-3 所示。

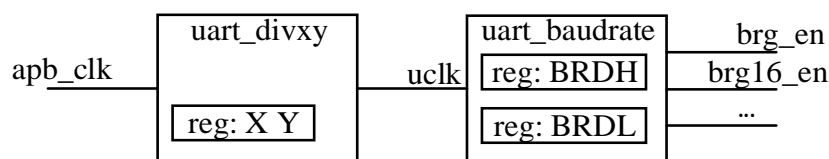


图 5-3 分频及波特率产生模块工作原理图

图中 `brg_en` 为满足波特率的信号，`brg16_en` 为 16 倍频采样信号。`BRDL` 和 `BRDH` 两个参数共同影响 `brg16_en` 的频率，`brg_en` 信号的周期是 `brg16_en` 信号周期的 16 倍。

为了检测波特率的配置功能是否正确，该测试用例给对应寄存器写入不同的值，然后在 `Div_checker` 中对分频关系进行检查。因为只需要对典型的波特率数值进行验证，所以在寄存器中写入的值为 9600、19200 和 115200 波特率所对应的分频系数。

5.1.1.7 APB 协议检测

为了确保该验证平台产生的激励符合协议规定，需要对 APB 信号以及 UART 信号进行协议检测。由于要访问的是信号级数据，APB 信号相关的检测将在 `interface` 级别的类中进行，而 UART 信号相关的检测在 `Uart_Monitor` 中进行。本小节主要阐述对 APB 协议的检测。

本验证平台使用断言语句检测 APB 端口信号，保证不同阶段各个信号的值符合 APB 协议。这主要包括下面三部分：

(1) 保证在特定时刻，各个信号线拥有有效值。对于 `PSEL` 和 `PRESET` 信号，因为这两个信号在仿真开始就被赋值，所以直接使用 `$isunknown` 语句检查其信号值是否为未知态。而其他信号则需要在 APB 总线执行完特定状态后再进行检测。以 `PRESET` 为例，代码如下：

```
1. property SIGNAL_VALID(signal);
2.   @(posedge PCLK)
3.   !$isunknown(signal);
4. endproperty: SIGNAL_VALID
5. RESET_VALID: assert property(SIGNAL_VALID(PRESETn));
```

(2) 保证信号在规定时间内维持稳定。如 APB 协议中规定，当 `PSEL` 拉高后，`PADDR`、`PWRITE`、`PWDATA` 等信号的值应该维持稳定。本验证平台使用 `$stable` 语句进行验证。

(3) 保证各个信号在不同阶段的值正确。如 APB 协议中规定，`PSEL` 拉高后代表进入 `SETUP` 状态；一个周期后，`PENABLE` 信号应该被拉高，代表进入了

ENABLE 状态；PENABLE 拉高一个周期后应该被拉低，代表一个周期的结束等。本验证给平台使用 non-overlapping ‘|=>’ 语句，样例代码如下：

```
1. property PENABLE_DEASSERTED;
2.   @(posedge PCLK)
3.     $rose(PENABLE) |=> !PENABLE;
4. endproperty: PENABLE_DEASSERTED
5.
6. PENABLE_DEASSERT: assert property(PENABLE_DEASSERTED);
```

5.1.1.8 接收数据溢出场景检测

该 DUT 可以通过 FCR[0]来决定是否开启 FIFO 功能。当 FCR[0]被配置为 1 时，表示开启了 FIFO 功能，即 16550 模式。此时该 DUT 最大可以存储 16 帧的数据。当 FCR[0]被配置为 0 时，表示关闭了 FIFO 功能，即 16450 模式。此时该 DUT 最多只能存储 1byte 的数据。

当接收数据的空间被占满时，如果继续通过 RXD 端口向 DUT 内写入数据，则 LSR[1]会拉高，表示此时空间已满，无法再接收数据。

本验证平台在 `receive_overrun_test` 中调用对应的虚拟序列对该功能进行验证。该虚拟序列首先调用 `write_FCR_seq` 序列将 FCR[0]的值配置为 1，此时 DUT 工作在 16550 模式下，FIFO 拥有 16byte 深度。然后调用 `uart_rx_seq` 序列，随机化其中的 `no_rx_chars` 参数并约束其随机值大于 16。这样 `Uart_Driver` 便会向 RXD 端口写入超过 16 帧的数据。当写入第 17 帧数据时，该测试用例会去读取 LSR 寄存器的值，并查看 LSR[1]是否被拉高。

此外，被写入的数据应该通过 `read_RBR_seq` 序列读取出来，并在 `Rx_Scoreboard` 中与 `Rx_Monitor` 采集的数据进行比较，保证在溢出的情况下，写入的这些数据依然能被正确读出。而发生溢出之后再从 RXD 端口被写入的数据，由于没有被真正写入到 RFIFO 中，所以不参与后续的比较。

当 FCR[0]被配置为 0，即 DUT 工作在 16450 模式下时，该测试用例使用相同的方法验证了相关功能。

5.1.1.9 接收数据错误场景检测

当 RXD 端口接收的数据拥有 parity error 或 frame error 时，该 DUT 可以检测到这些错误，并拉高 LSR 寄存器中的 LSR[2]和 LSR[3]。

本验证平台在 `receive_error_test` 中调用对应的虚拟序列对该功能进行检测。在该虚拟序列中，首先配置相关的寄存器，然后调用 `uart_rx_seq` 并将其 `no_error` 参数配置为 0，这样 `Uart_Seqitem` 中代表着 parity error 和 frame error 的参数 `pe` 和 `fe`

便会被随机置为 1。当 Uart_Driver 给 RXD 端口输入数据时，会根据 pe 和 fe 参数来产生具有对应错误类型的数据。最后该测试用例读取 LSR 寄存器的值，并对 LSR[3:2]进行检查。

5.1.1.10 接收 break 信号场景检测

UART 器件在与其他器件进行交互时，有时需要发送或者接收一段时长的低电平(break)信号，低电平的持续时间一般大于 UART 协议中一帧的时长。

该 DUT 的接收逻辑可以对 break 信号进行检测，当其收到一段时长的低电平信号后，就会拉高 LSR 寄存器中的 LSR[4]，表示此刻接收到的是一段的 break 信号，而不是有效的数据。

本验证平台在 receive_break_test 中调用对应的虚拟序列实现对该功能的检测。该虚拟序列首先对相关寄存器进行配置，然后实现 break 信号的输入，即将 uart_rx_seq 中 no_error 参数和 stop_low 参数置为 1，这样对应的 Uart_Driver 将会给 RXD 端口输入足够时长的低电平。经过一段时间后，该测试用例读取 LSR 寄存器的值，并判断 LSR[4]是否为 1，从而完成验证。

5.1.1.11 发送数据空场景检测

TFIFO 和发送逻辑中的移位寄存器关系如图 5-4 所示。当 TFIFO 中没有有效数据时，LSR[5]将会被拉高，当 TFIFO 和移位寄存器中都没有有效数据时，LSR[6]被拉高，这两个状态位可以提供发送数据空间的空满信息。

对该功能进行检测时，对应的测试用例先向 THR 寄存器写入数量为 no_tx_chars 帧的数据。当数据开始传输后，LSR[5]和 LSR[6]都应该为 0，表示此时 TFIFO 和移位寄存器中都有数据。随着 TXD 将数据逐帧传出，待 TFIFO 将最后一帧数据打入给移位寄存器中后，LSR[5]应该为 1 而 LSR[6]应该为 0。最后当移位寄存器中的数据也传输出去后，LSR[5]和 LSR[6]都变成 1。

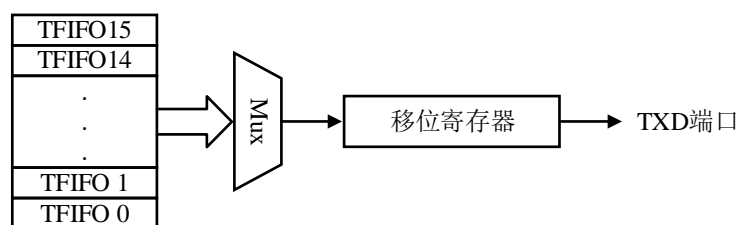


图 5-4 TFIFO 与移位寄存器关系图

在验证这两位寄存器时，Uart_Monitor 需要去采集已经通过 TXD 端口发送的

数据个数，并通过旗语告知相关序列。对应的序列根据发送的数据个数，在以下三个时间读取 LSR 寄存器的值，并与预期值进行比较：（1）刚刚写入数据时，（2）已经发送了 `no_tx_chars-1` 帧的数据时，（3）发送完全部的数据时。

5.1.1.12 其他功能检测

其他测试计划的实现与上述类似，同样是通过序列对 DUT 施加特定的激励，然后在恰当的时间去检测相关寄存器的值或者 DUT 的输出值，这些测试计划不再详细描述。

需要注意的是，当使用同一个验证平台运行多个测试用例时，由于不同测试用例所验证的功能不同，对验证组件的要求也不同，所以相关代码之间可能会出现冲突或干扰。

该验证平台使用以下方法解决该问题：（1）将这些代码写在对应的测试用例中。验证平台在启动不同测试用例时，便会分开执行这些代码。（2）在部分组件中，比如 `scoreboard` 组件，使用 `$test$plusargs` 语句和条件语句对不同的测试情况进行区分，有选择性的执行相关代码。（3）在部分分类中添加参数，通过对参数的调控而使组件执行不同的代码。

5.1.2 覆盖率计划实现

在本验证平台中，覆盖率的实现主要分为以下几种：信号覆盖、事务级数据覆盖以及寄存器覆盖。

1. 信号覆盖

信号级别的覆盖主要针对总线协议的检测，该验证平台在 `interface` 级别的类中使用 `Cover Property` 语句对 APB 总线信号进行了监测。

2. 事务级数据覆盖

该 DUT 模块可以将 APB 端口写入的数据转化成串行数据从 UART 的 TXD 端口发出，也可以将从 UART 的 RXD 端口接收到的串行数据转化成并行数据后被 APB 端口读出。

在对数据收发进行验证时，验证平台会将信号级的数据转换成事务级的数据。这些数据便是事务级数据覆盖的对象。该验证平台使用 `Rx_Coverage_Monitor` 和 `Tx_Coverage_Monitor` 去采样收发数据，并在这两个组件中构建对应的覆盖组，以验证收发数据的格式是否完备。

该验证平台将 `Rx_Scoreboard` 和 `Tx_Scoreboard` 作为前置组件，当确保收发数据的格式和内容无误后，再把相关寄存器的值传给 `Rx_Coverage_Monitor` 和

Tx_Coverage_Monitor 进行覆盖采样。

以 Rx_Coverage_Monitor 中采样接收数据的 rx_word_format_cg 覆盖组为例，关键代码如下：

```

1. covergroup rx_word_format_cg with function sample(bit[5:0] lcr);
2.   option.name = "rx_word_format";
3.   option.per_instance = 1;
4.
5.   WORD_LENGTH: coverpoint lcr[1:0]{ ...}
6.   STOP_BITS: coverpoint lcr[2]{ ...}
7.   PARITY: coverpoint lcr[5:3]{ ... }
8.
9.   WORD_FORMAT: cross WORD_LENGTH, STOP_BITS, PARITY;
10.
11. endgroup: rx_word_format_cg

```

Rx_Coverage_Monitor 和 Tx_Coverage_Monitor 继承自 uvm_subscriber。该验证平台在其中的 write 函数里编写采样语句，并在上层环境中将对应 scoreboard 组件的 port 端口连接到这两个组件的 analysis_export 端口。

3. 寄存器覆盖

对寄存器覆盖率的收集其实也属于事务级的操作，但由于和收发数据覆盖率的收集有一定区别，因此单独阐述。根据寄存器种类的不同，寄存器覆盖的实现主要包括以下几个方面：

（1）全部寄存器的访问覆盖。

这种覆盖率的采样不需要额外的条件，只需要在 APB 总线对 DUT 的寄存器进行操作时，将 Apb_Monitor 采集到的数据传递到对应的 subscriber 中，在 write 函数中进行采样即可。

由于部分寄存器拥有只读、只写属性，因此在编写覆盖组时需要注意一些无效仓的存在，关键代码如下：

```

1. covergroup reg_access_cg();
2.   option.name = "reg_access_cg";
3.   option.per_instance = 1;
4.
5.   WE: coverpoint we { ...}
6.   ADDR: coverpoint addr { ...}
7.   ACCESS: cross WE, ADDR {ignore_bins...}
8.
9. endgroup: reg_access_cg

```

（2）复杂寄存器中域的覆盖。

当寄存器中包含有多个域(field)时，需要对每个域的功能进行具体的覆盖率统计。以 LSR 寄存器为例，该寄存器具有多个域，可以分别代表不同的数据传输状

态。当发现 LSR 寄存器被访问时，验证平台会采取更为细致的覆盖策略，保证 LSR 中每个域都能被覆盖到，关键的覆盖组代码如下所示：

```

1. covergroup lsr_read_cg() with function sample(bit[7:0] LSR);
2.   option.name = "lsr_read_cg";
3.   option.per_instance = 1;
4.
5.   RECEIVE_DADA_READY: coverpoint LSR[0] {...}
6.   OVERRUN_ERROR: coverpoint LSR[1] {...}
7.   PARITY_ERROR: ...
8.   ...
9. endgroup: lsr_read_cg

```

寄存器覆盖组件在验证平台中实现方式如图 5-5 所示，可以看到 reg_access_cov_monitor 可以收集 Apb_Agent 中广播端口发出的操作信息，从而通过其中读写信号和地址进行相应的覆盖。

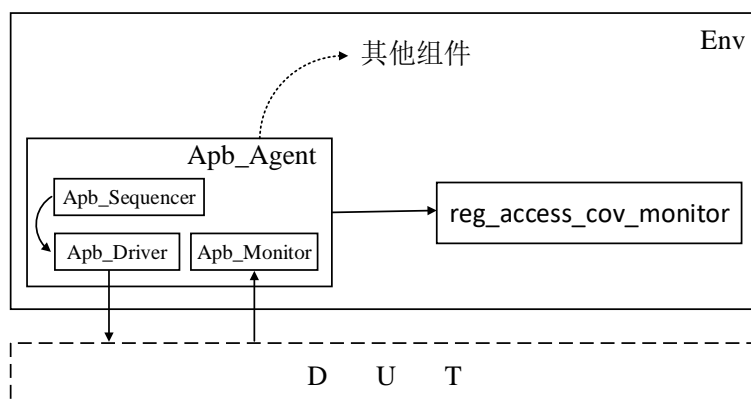


图 5-5 寄存器覆盖组件实现框图

5.1.3 检查计划实现

检查计划的实现会根据检查目标的不同而有所区别。对于事务级数据的检查一般在对应的 scoreboard 中实现；对于协议的检查一般在 monitor 组件或者 interface 中实现；对于寄存器数值的检查较为灵活，在 test 或者 scoreboard 中都有实现。

各个检查计划的实现已经在前面章节中被提及，此处不在赘述。

5.2 验证结果分析

在验证流程前期，验证人员需要根据仿真波形去查看激励的施加情况，并根据仿真报告中的打印信息修改验证平台、完善测试用例。在验证后期，随着验证平台的成熟，验证人员可以从繁杂的人工检查工作中脱离出来，根据仿真报告和覆盖率情况对验证的正确性和完备度进行评估，只有收到报错信息后，才会在波形图中对相关信号进行追踪和分析。

本节首先对验证环境进行描述，然后分析了两个典型测试用例的报告、波形以及覆盖率情况，最后对全部测试用例的验证结果进行了说明。

5.2.1 验证环境描述

本验证平台使用 QuestaSim 和 VCS 软件进行仿真。

QuestaSim 软件基于标准的单核验证引擎，内部集成了 HDL 模拟器、Constrain 求解器、判断引擎等，可以更为强力的支持覆盖率功能，在 windows 和 linux 系统中都有广泛应用^[29]。VCS 在仿真性能和高级仿真技术方面有一定优势，一般在 linux 系统中使用。

由于两个软件的厂商不同，对代码的解析存在一定差异，所以在 QuestaSim 中能够成功运行的代码并不一定可以直接应用在 VCS 中，有时需要一定的修改以规避某些错误。该 DUT 中部分端口名称与关键字重复，导致 VCS 无法识别，已对相关模块代码进行了修改。

5.2.2 典型测试用例结果分析

5.2.2.1 基本数据发送功能测试结果分析

图 5-6 为某一帧数据的发送波形图，波形中反应的信息与对应测试用例的操作一致。在配置寄存器阶段，APB 总线将 LCR 寄存器的值配置为 8'h27，代表着该帧 UART 数据应当包含 8 位数据位、0 位校验位、2 位停止位。由图可知，TXD 端口的输出波形符合预期。



图 5-6 基本数据发送测试波形图

在放宽约束条件并循环足够多的次数之后，最终验证报告如图 5-7 所示。可以看到在该功能的检测过程中，没有 UVM_ERROR 和 UVM_FATAL 的情况发生。报告中的 UVM_INFO 信息，可以辅助验证人员了解验证过程中的代码运行情况，而 UVM_WARNING 信息为使用回调函数时产生的一些警告，不会对验证结果产生影响。

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO : 5931
# UVM_WARNING : 3601
# UVM_ERROR : 0
# UVM_FATAL : 0
```

图 5-7 基本数据发送测试仿真报告

在功能覆盖率窗口中，可以看到相关的功能覆盖情况，如图 5-8 所示。图中各个覆盖点以及交叉覆盖的比率都达到了 100%，这代表着在多次随机化的过程中，产生的激励已经完全覆盖了预期功能点，证明对当前测试场景的验证已经足够完备。

Covergroups					
Name	Coverage	Goal	% of Goal	Status	Included
/uart_env_pkg/uart_tx_coverage_monitor					
TYPE tx_word_format_cg	100.0%	100	100.0%		✓
CVP tx_word_format_cg::WORD_LENGTH	100.0%	100	100.0%		✓
CVP tx_word_format_cg::STOP_BITS	100.0%	100	100.0%		✓
CVP tx_word_format_cg::PARITY	100.0%	100	100.0%		✓
CROSS tx_word_format_cg::CROSS_ALL	100.0%	100	100.0%		✓

图 5-8 发送数据格式的功能覆盖率结果

5.2.2.2 基本数据接收功能测试结果分析

图 5-9 为某一帧数据的接收波形图，与上一节的分析过程类似，在配置寄存器阶段，APB 总线将 LCR 寄存器的值配置为 8'h0f，代表该帧 UART 数据应该包含 8 位数据位、1 位奇校验位、2 位停止位。由图可知，RXD 端口接收的波形符合预期。

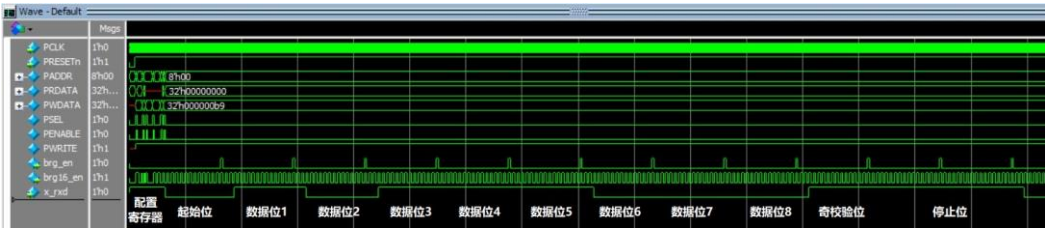


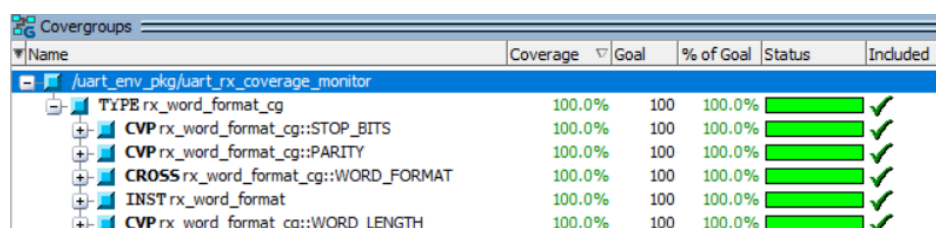
图 5-9 基本数据接受测试波形图

最终的验证报告如图 5-10 所示。由图可知，在基本数据接收功能的检测中没有出现 UVM_ERROR 和 UVM_FATAL 信息。而少量的 UVM_WARNING 信息为寄存器模型创建过程中产生的警告，不会影响验证结果。

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO : 5362
# UVM_WARNING : 7
# UVM_ERROR : 0
# UVM_FATAL : 0
```

图 5-10 基本数据接收测试仿真报告

最终的功能覆盖率情况如图 5-11 所示，图中各个覆盖点和交叉覆盖的比率均为 100%，代表随机测试激励已经充分的覆盖了各种可能出现的预期数据组合情况，证明当前测试场景的验证是完备的。



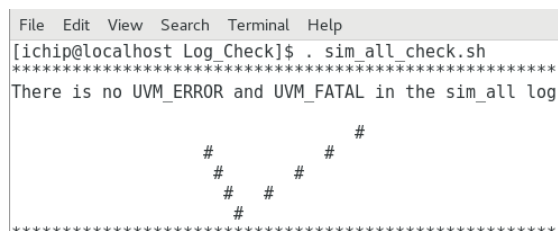
Name	Coverage	Goal	% of Goal	Status	Included
/uart_env_pkg/uart_rx_coverage_monitor					
TYPE rx_word_format_cg	100.0%	100	100.0%	✓	✓
CVP rx_word_format_cg::STOP_BITS	100.0%	100	100.0%	✓	✓
CVP rx_word_format_cg::PARITY	100.0%	100	100.0%	✓	✓
CROSS rx_word_format_cg::WORD_FORMAT	100.0%	100	100.0%	✓	✓
INST rx_word_format	100.0%	100	100.0%	✓	✓
CVP rx_word_format_cg::WORD_LENGTH	100.0%	100	100.0%	✓	✓

图 5-11 接收数据格式的功能覆盖率结果

5.2.3 最终验证结果分析

由上节可知，验证人员对各个测试用例的分析过程具有相似性。因为该验证平台运行的测试用例较多，本节不再逐个分析单个测试用例的仿真报告和波形图，而是通过脚本文件批量处理仿真报告中的关键数据，生成最终的打印信息，反应所有测试用例的运行情况。

该验证平台通过 Makefile 文件调用 VCS 工具将全部测试用例执行完后，把各个用例生成的仿真报告存放在对应的 log 文件里，并通过 shell 脚本抓取这些文件中的 UVM_ERROR 和 UVM_FATAL 信息，从而达到快速批量检查测试结果的目的。脚本文件的运行结果如图 5-12 所示，说明各个测试报告中没有 UVM_ERROR 和 UVM_FATAL 发生。



```
File Edit View Search Terminal Help
[ichip@localhost Log_Check]$ . sim_all_check.sh
*****
There is no UVM_ERROR and UVM_FATAL in the sim_all log

#
#
#
#
#
#
*****
```

图 5-12 shell 脚本运行结果

在全部测试用例被执行完后，各个测试用例的覆盖率结果，都会反映到最终的

覆盖率报表中。验证人员可以通过代码覆盖率和功能覆盖率去对验证流程的完备性进行评估。如果代码覆盖率较低而功能覆盖率较高，则说明有未被检测的功能，或者 DUT 中有冗余代码；如果代码覆盖率较高而功能覆盖率较低，则说明 DUT 可能未实现全部功能。

该 DUT 含有部分冗余代码，在去除冗余项后，最终的代码覆盖率达到 100%，如图 5-13 所示。其中被去除的冗余项包括不会被执行的 default 语句，不会发生的条件组合情况，部分不做要求的翻转覆盖率以及 DMA、IRDA 等在本次验证中不考虑的功能。

Name	Score ▾	Line	Condition	Toggle	FSM
uart_module	100.00%	100.00%	100.00%	100.00%	100.00%
uart_apbif	100.00%	100.00%	100.00%	100.00%	
uart_cfg	100.00%	100.00%	100.00%	100.00%	
uart_control	100.00%	100.00%	100.00%	100.00%	100.00%
uart_fifo	100.00%	100.00%	100.00%	100.00%	
uart_syn_inuart	100.00%	100.00%		100.00%	
uart_syn_outuart	100.00%	100.00%		100.00%	
uart_tfifo	100.00%	100.00%	100.00%	100.00%	

图 5-13 最终代码覆盖率结果

最终功能覆盖率结果如图 5-10 所示，可以看到全部覆盖组的覆盖率都达到了 100%，这说明此次测试产生的随机激励已经完全覆盖了预期的各种状态组合。

NAME	SCORE	INSTANCES
uart_env_pkg::uart_tx_coverage_monitor::tx_word_format_cg	100.00	100.00
uart_env_pkg::uart_rx_coverage_monitor::rx_word_format_cg	100.00	100.00
uart_env_pkg::uart_interrupt_coverage_monitor::int_enable_cg	100.00	100.00
uart_env_pkg::uart_modem_coverage_monitor::mcr_loop_inputs_cg	100.00	100.00
uart_env_pkg::uart_reg_access_coverage_monitor::reg_access_cg	100.00	100.00
uart_env_pkg::uart_reg_access_coverage_monitor::lsr_read_cg	100.00	100.00
uart_env_pkg::uart_reg_access_coverage_monitor::msr_read_cg	100.00	100.00
uart_env_pkg::uart_reg_access_coverage_monitor::mcr_write_cg	100.00	100.00
uart_env_pkg::uart_reg_access_coverage_monitor::ier_write_cg	100.00	100.00

图 5-14 最终功能覆盖率结果

综上，通过对仿真报告以及覆盖率结果的分析，可以知道该验证平台对 DUT 模块的验证是充分完备的，该 DUT 模块所实现的功能与 spec 文档一致，并拥有足够的流片成功率。

5.3 本章小结

本章主要分为两部分，第一部分主要介绍了在该 UVM 验证平台上实现验证计划的过程，首先详细讲解了重点测试场景的实现思路，然后阐述了覆盖率计划和检

查计划的实现方法。第二部分对验证结果进行了分析，首先以基本数据收发功能测试为例，对单个测试的验证结果进行了说明。然后使用脚本文件批量处理验证报告，并对最终的覆盖率结果进行了分析，得出了此次验证足够充分完备的结论，保证了DUT模块流片的成功率。

第六章 结论

6.1 工作总结

本文采用 UVM 验证方法学，对 APB-UART 模块进行了验证，具体工作总结如下：

(1) 深入学习了 SystemVerilog 语言和 UVM 验证方法学，了解验证平台的基本架构和运行机制，并对寄存器模型等内容进行扩展研究，可以在 UVM 库文件的基础上灵活熟练的搭建验证平台，并实现激励的输入。

(2) 深入研究了待测 APB-UART 模块，包括但不局限于 APB 协议、UART 协议的实现细节，模块架构和端口细节，模块寄存器和相关功能实现等，充分了解模块的各个功能点并在此基础上提出了验证计划。

(3) 独立搭建了完整的验证平台，该平台包括：处理不同协议端口的 Apb_Agent、Uart_Agent 以及 Modem_Agent 组件；进行数据比对和检测的 Rx_Scoreboard、Tx_Scoreboard、Modem_Scoreboard 以及 Div_Checker 组件；拥有预警机制的寄存器模型；验证顶层、验证环境以及配置文件等。

(4) 编写了多个测试用例和序列(sequence)，通过旗语(semaphore)和事件(event)解决了仿真结束时间与采样时间不明确的问题，对 DUT 模块施加激励并验证结果，覆盖了其常见功能。

(5) 编写覆盖率组件并收集了代码覆盖率和功能覆盖率，为仿真进度提供了判断依据。最终的覆盖率达到 100%，说明对 DUT 模块的验证工作已经足够完备。

该验证平台具有重用性和可移植性：其中 UART 协议相关的组件可以在内部通过配置信息直接重用；对于其他协议转换模块，该 UVM 验证平台可以提供借鉴意义并在一定程度上实现移植使用。

6.2 不足与展望

本文虽然实现了对 APB-UART 模块的验证工作，但是也存在着一些可以改进的地方。比如对覆盖率没有动态的实时把控，不清楚覆盖率在进行多少次循环测试后可以达到预期要求；部分 monitor 和 scoreboard 组件中使用了 \$test\$plusargs 来迎合测试用例，在移植使用时可能导致代码冗余；验证平台以及寄存器模型都是手工搭建，没有使用脚本语言实现自动化^[30]等，这些都是有待于提高的地方，可以从这些角度入手，进一步完善该 UVM 验证平台。

致谢

蓉城漫漫七载，尚不觉流年暗换，再看寒窗岁月，又岂止十年。谨在致谢之际，对曾经给予过帮助的人，诚表感恩和谢意。

感谢我的父母，于求学的道路上一路陪伴着我。在开心的时刻共享欢乐，在辛苦的时刻给予慰藉。你们总是支持我的决定和选择，让我能够自由飞翔。感谢你们，我爱你们。

感谢我的导师，从本科时期就开始在科研上提供帮助，多年来更是给予了丰富的资源让我茁壮成长。在教研室学习的时间里，导师平和包容的性格和严谨负责的精神更是影响了我，让我形成了正确的科研态度，有了步入社会的勇气和信心，谢谢您。

感谢教研室的同学，和你们在一起的日子里，我锻炼了很多能力，突破了很多难点，更得到了很多温暖。在这个屋檐下，大家和睦互敬，我不仅仅学习到了知识，更磨练了品行和性格，谢谢你们。

感谢我的朋友，是你们在日常生活中一直鼓励我、帮助我，并在科研之余一起培养有益的兴趣和爱好，让我的校园生活丰富多彩，谢谢你们。

感谢实习期间的前辈，是你们让我体会到了工作中的互助和温暖。在这段弥足珍贵的日子里，我不仅仅收获了知识，提升了能力，更踏出了步入社会的关键一步，谢谢你们。

最后感谢时敏，是你一直体会我的心情、共情我的感受、尊重我的选择，这在科研和生活中提供了重要的精神力量，让我能够审视自我，保持心态，愈战愈勇，谢谢你。

参考文献

- [1] 包日辉. SoC 设计平台中若干 IP 模块的设计[D]. 杭州: 杭州电子科技大学, 2019, 1-3
- [2] H. D. Foster. Trends in functional verification: a 2016 industry study [C]. DvCon, San Jose, 2017, 3-4
- [3] R. Drechsler. Formal System Verification[M]. Switzerland: Springer, 2018, 1-30
- [4] 张春, 麦宋平, 赵益新. SystemVerilog 验证: 测试平台编写指南[M]. 北京: 科学出版社, 2009, 5-6
- [5] S. Rosenberg. A Practical Guide to Adopting the Universal Verification Methodology[M]. North Carolina, LULU, 2013, 259-263
- [6] 刘斌. 芯片验证漫游指南[M]. 北京: 电子工业出版社, 2018, 257-314
- [7] 徐金甫, 李森森. 采用 UVM 方法学实现验证的可重用与自动化[J]. 微电子学与计算机, 2014, 31(11): 14-17
- [8] 田晓旭, 徐庆阳, 汤先拓, 等. 基于 UVM 的寄存器验证自动化方法[J]. 集成电路应用, 2020, 37(02): 18-21
- [9] 张少真, 成丹, 刘学毅. UVM 和 Matlab 的联合仿真方法及应用[J]. 中国集成电路, 2017, 26(09): 18-25
- [10] 张瑞. 基于断言的形式化验证与 UVM 的综合应用[D]. 西安: 西安电子科技大学, 2018, 47-63
- [11] 刘魁玉. 基于 UVM 的 1553B 总线协议验证平台设计[D]. 哈尔滨: 哈尔滨工业大学, 2019, 1-4
- [12] 陈琳娜. 基于 UVM 的层次化验证平台研究[D]. 浙江: 浙江大学, 2018, 1-5
- [13] J. Aynsley. Doing Funny Stuff with the UVM Register Layer: Experiences Using Front Door Sequences, Predictors, and Callbacks[C]. DvCon, California, 2017, 1-11
- [14] K. Schwartz, T. Corcoran. Error Injection: When Good Input Goes Bad[C]. DVCon, California, 2017, 1-13
- [15] J. Vance, J. Montesano, K. Vasconcellos. My Testbench Used to Break! Now it Bends: Adapting to Changing Design Configurations[C]. DVCon, California, 2018, 1-17
- [16] Accellera. Verification Intellectual Property (VIP) Recommended Practices [EB/OL]. https://ocpip.org/images/downloads/standards/uvm/VIP_1.0.pdf, Accellera, 2009
- [17] 张彦磊. 基于 UVM 的 APB-I~2C 验证 IP 的设计与实现[D]. 成都: 电子科技大学, 2019, 6-9
- [18] 张强. UVM 实战[M]. 北京: 机械工业出版社, 2014, 64-67
- [19] 潘玉茜. 基于 UVM 的 GMAC 高效验证平台设计[D]. 西安: 西安电子科技大学, 2017, 7-9
- [20] Accellera. Universal Verification Methodology (UVM) 1.2 User's Guide [EB/OL]. https://accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf, Accellera, 2015

- [21] IEEE, IEEE Std 1800™-2012, Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language[S]. New York: IEEE Standards Association Corporate Advisory Group, 2013
- [22] C. E. Cummings, H. Chambers. UVM Analysis Port Functionality and Using Transaction Copy Commands[R]. USA: Sunburst Design, 2018
- [23] 仇荔.基于 OVM 架构的 EPA 芯片验证的研究[D].浙江:浙江大学,2013,58-65
- [24] F. Fallah, S. Devadas, K. Keutzer. OCCOM-Efficient Computation of Observability -Based Code Coverage Metrics for Functional Verification[C]. Proceedings 1998 Design and Automation Conference, San Francisco, 1998,1-6
- [25] E. Cerny, S. Dudani, J. Havlicek,etc. SVA: The Power of Assertions in SystemVerilog[M]. Switzerland, Springer, 2015,3-28
- [26] ARM. AMBA Specification Rev 2.0 [EB/OL]. <https://developer.arm.com/docs/ih0011/latest/amba-specification-rev-20>, ARM, 2019
- [27] Mentor Graphics. UVM Cookbook [EB/OL]. <https://verificationacademy.com/cookbook/uvm>, Mentor Graphics, 2019
- [28] M. Litterick, M. Harnisch. Advanced UVM Register Modeling-There's More Than One Way to Skin a Reg[R]. Munich: DVCon, 2014
- [29] Mentor, Questa Advanced Simulator[EB/OL].<https://www.mentor.com/products/fv/questa/>,2020
- [30] 石轩.基于 UVM 的验证环境自动化生成和测试用例的标准化设计[D].西安:西安电子科技大学,2018,45-53

攻读硕士学位期间取得的研究成果

- [1] 段一杰.基于 UVM 的 APB-UART 验证平台的设计与实现[J].研究生学报,2020,(116)
- [2] 段一杰,王忆文.一种基于 UVM 的 APB-UART 模块的验证方法[P].中国,发明专利,202010360438.6, 2020 年 04 月 30 日