高级验证方法学
–AVM

Mark Glasser, Editor
Adam Rose
Tom Fitzpatrick
Dave Rich
Harry Foster

Contractor/manufacturer is:

Mentor Graphics Corporation

8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777.

Telephone: 503.685.7000

Toll-Free Telephone: 800.592.2210

Website: www.mentor.com

SupportNet: www.mentor.com/supportnet

Contact Your Technical Writer:

www.mentor.com/supportnet/documentation/reply_form.cfm

Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code,
generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations,elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition,

"submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."
"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
   a. You must give any other recipients of the Work or Derivative Works a copy of this License; and
   b. You must cause any modified files to carry prominent notices stating that You changed the files; and
   c. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding

those notices that do not pertain to any part of the Derivative Works; and

d. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places:
within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or,within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License.

You may add Your own attribution notices within Derivative Works that You distribute,alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License. You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use,reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. Trademarks. This License does not grant permission to use the trade names, trademarks,service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing,Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the

appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

目录

第 1 章 使用手册指南

序

当我要学习一个新的软件时，我会去发现其应用领域存在的一些问题， 然后用这个新的工具去解决这些问题。 在解决问题的过程中我学会了如何使用这个工具，同时也获得了一些有实际意义的观点，即这个工具的哪些特性是有用的，哪些特性是没有用的。最简单的做法就是去拷贝一些别人做过的程序并做修订。 许多年以前，当我刚开始学习用C编程时，我就从拷贝“Hello World”这个程序开始看它会如何工作，当然，当我编译后运行时，就出现如下字符串：

```
hello world
```

在我的显示器上闪烁。这个小程序使我明白基本的C程序看起来是什么样，以及如何去使用Unix应用程序，如cc和ld。下一步就是使用变量并用scanf从命令行读取。从这里，我学会了循环，函数，不久我就可以使用C来写真正的程序了。每次我想尝试一下新的方法，我会写一个简单的程序来编译和运行。

本书假定大部分工程师用相似的途径学习新技术。他们都想把这个做出来看看感觉如何，了解由它引出的各种问题的情况，并提高实际经验。这就是为什么quickstart指南和在线帮助系统受欢迎的原因。通常，工程师们不想首先读长篇的用户手册并学习操作理论。他们宁愿先使用，然后当遇到问题时再参考用户手册。同时，他们对这个技术是什么和如何进行基本操作有了大体的了解。本手册中的详细介绍会在对这门技术是什么有了基本了解之后体现更高有价值了。

考虑到这本书是功能验证，尤其是测试平台结构的启蒙指导。构建测试平台进行数字电路设计的功能性验证是一个难题。从哪里开始？许多关于这方面的书都是引领其读者了解验证组件的软件结构和发展的理论知识。这是很有价值的，我们也鼓励读者去阅读并钻研这方面的书籍。附录提供了这方面的

参考书。我们也知道工程师们想了解一个完整的测试平台是什么样，如何构建一个好的接口，并无需预先花费一周的时间来构建模型的情况下理解随机数据流。

这本书给学习测试平台的结构提供了实用的途径。我们提供了一系列的实例，每个例子都解决一个特殊的验证问题。所有的实例有完整的文件，包括构造和运行脚本，这样可以在仿真器中执行它们并观察其行为。这些实例通常都较小并集中在所阐述的问题上，这样读者就不必通过阅读许多的辅助材料来了解整个实例的核心。

所有的实例都是采用Mentor Graphics 高级验证方法学（Advanced Verification Methodology －AVM)来构建的。该方法学被充分验证，可通过结合固定的软件结构实例和事务交易级建模（Transaction Level Modeling－TLM）技术来构建复杂的测试平台。

之所以将本书的书名定为验证手册（verification cookbook），是因为它模仿食谱（cookbook）的结构来组织内容。每个实例就是一种“烹饪方法”。它包括将AVM用到一个特定问题的可执行代码。所有方法按章节组织，从简单到复杂。

这里提供的实例可有多种用途。实例是线性推进出现的—从只有一个管脚级激励发生器，监视器和DUT的最基本的测试平台，到非常复杂的用途，包括堆栈协议，覆盖率和自动测试控制。每个实例介绍新的概念，并告诉你如何用直接的方法来应用这些概念。我们建议你从第一个实例开始练习。当你对它很熟悉时，再转向第二个例子。依此进行，掌握每个实例。

这里的实例都是让你去探索的。在你运行了这些实例以后，再学习代码，真正理解这些实例的结构。每个实例提供的代码文件可作为路标，告诉你重要的特性。从这里开始学习编码的组织，风格和其它没有明确讨论的实现的细节。你会发现模块化设计更易于理解每个模块和了解整个设计和流程。

在运行这些实例时，可以通过改变仿真的总时间，改变激励，增加或减少组件等等以看到更多的结果。你所尝试的每个新事件都会帮助你更为完整了理解实例以及它们是如何工作的。

你可以在的工作中自由的使用这些实例代码作为模板[1]。将你认为有用的代码

1.The Verification Cookbook is delivered under an open source license. See LICENSE.txt in the cookbook kit or refer to http://opensource.org/licenses/apache2.0.php for full text of the Apache-2.0 license.

剪切并粘贴成你自己的，或者将它们作为你研发自己的验证框架的开端。总的说来，好好享用。

绪论

当硬件设计师和验证工程师考虑他们的工作时，软件结构并不是一个经常出现的话题。设计师和验证工程师，尤其是那些电子工程专业的人们，自然而然的将设计和验证工作看成是“硬件问题”，就是说构造和验证系统需要硬件设计理论。当然，这在很大程度上是对的。电子设计需要对硬件有深入的认识：每一样都是基本的DC和AC电流分析，晶体管操作，到通讯协议和计算机架构。对于在硅片上设计的实现（当今大部分是这样的），还需要略知物理结构。但是，构造测试平台来验证硬件设计则是另外一种问题。

当前，随着可靠综合器和同步设计技术的应用，设计师必须考虑的最底层情况就是RTL（寄存器传输级）。正如其名字所说明的，RTL级表现的基本设计原理就是寄存器，寄存器之间的连接以及更改数值时所需要的计算。既然每个寄存器只有在时钟脉冲到达时接收新值，计算寄存器值所需要的组合逻辑都可以抽象为布尔和代数表达式。

RTL跨在硬件和软件世界之间。RTL设计的组件可通俗的确认为硬件，如寄存器，连线，时钟。但是，组合表达式和控制逻辑看起来就像典型的程序设计语言，如C。构造RTL设计的过程就像一个编程过程。例如，你使用编译器，链接器和调试器就像你在用C编程。当然，它们也有区别。当用C编程时，有关时序，并行发生的事情和同步方面的问题就不需考虑（除非你在写内嵌式软件，这将更加模糊硬件和软件之间的界线）。

测试平台同样存在于软件世界。一个测试平台的组件正如在所有软件系统中看到的组件一样—数据结构和算法。测试平台是关注硬件的，因为它的工作就是去控制，响应和分析硬件。然而，其中仍然有大量的结构和操作属于软件的范畴。其原因有如下几点：

- 测试平台在比RTL更高的抽象级别运作。
- 激励发生以及结果的收集和分析不需要时序，寄存器，电线或其它硬件组件。
- 测试平台无需可综合，因此就不受可综合化[1]设计隐含约束的限制。

1. 在一些流程中，必须可综合所有的或部分的测试以便在硬件仿真器上运行。

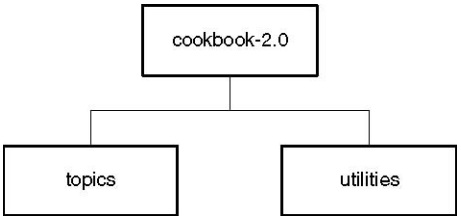
软件架构是当代验证技术的核心和方法。验证是一个软件问题。高级验证方法的使用是将软件架构技术应用到硬件验证问题中。软件架构本身是一个非常热点的主题，也有很多文献来描写它。我们没有必要深入了解诸如面向对象的编程，库的组织，代码分解，测试策略等问题。但是，在应用过程中我们会接触到这些概念，告诉你如何应用软件技术去构建测试平台。我们依赖这些例子来说明讨论的原理。

手册的使用

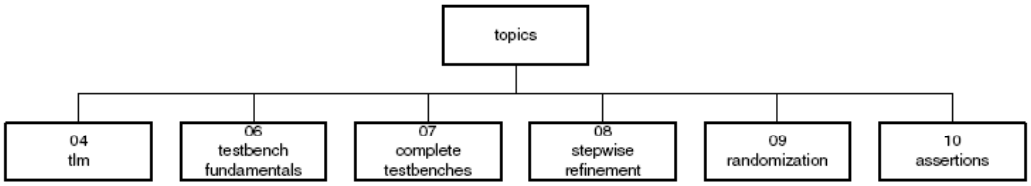
该书分成2部分：文件和实例树结构。在这一节里，我们告诉你如何使用说明书的树结构以及如何构建和运行实例。

本书的结构

本书包括2个基本目录：主题（topics）和应用程序（ulitities）。主题包括实例，应用程序包括用于SystemC 和System－ Verilog 的AVM库。

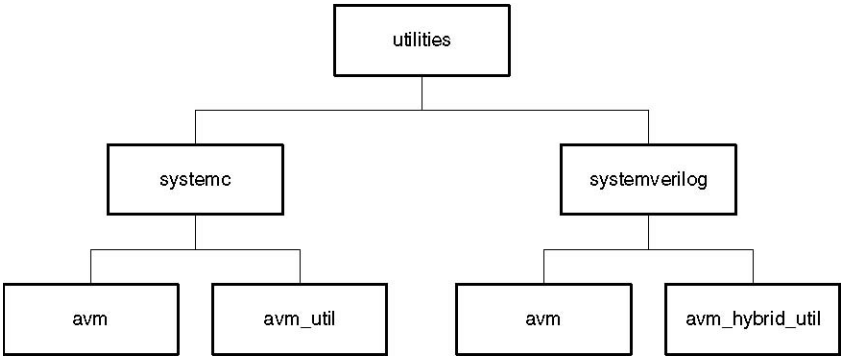


主题是树根，包含本书中提供的所有实例。目录名的前缀有数字表示在本书中所讨论的主题出现在哪一章。并不是本书的每一章都有相应的代码实例。



应用库有2部分,一个用于SystemC，一个用于SystemVerilog。在应用程序的

目录树上没有可运行的实例。应用库包括基类和应用函数，其采用SystemC和SystemVerilog来实现AVM。应用程序被主题目录树下的在大部分的实例中引用。SystemC 的应用库在附录A中进行了全面的解释。



形式

本书的实例以几个不同的形式存在中。形式与实例所采用的语言有一定的关系，每种形式都有一个唯一的ID标签。这些形式是：

1. sc - SystemC
2. svc - SystemVerilog class-based
3. svm - SystemVerilog module-based

vhdl - VHDL

采用形式名称是为了当同样的实例放在多种形式下时，消除原本名称和目录名称的歧义。

构造和运行实例

安装该手册的方法就是在适当的位置解压手册的压缩包。不需要额外的安装脚本或过程。

每个实例目录包括一个“all“脚本和一个”compile“脚本。“all“脚本按全名运行实例all_<form_id>.do（如all_svc.do）。”compile“脚本是一个提供诸如判断编译命令行中的－f选项的文件。每个实例都有一个vsim.do文件，该文件包括需要运行该实例的仿真器的命令。

运行实例最简单的方式就是执行“all“脚本。

```
% ./all_svc.do
```

下面是编译，连接和运行实例。你也可手动运行这些步骤：

```
% vlib work
% vlog -f compile_svc.f
% vsim -c top -do vsim.do
```

SystemC实例要求额外的连接步骤。运行SystemC的各个步骤的实例是：

```
% vlib work
% sccom -f compile_sc.f
% sccom -g -scv -link
% vsim -c top -do vsim.do
```

实例代码

本文中用于举例说明的大部分代码是直接从实例和应用库代码引用来的。在需要时看与代码相关的行号，在许多情况下，文件名插入在代码的顶部或底下。文件名识别实例树中的特殊文件，在本文中实例树包括代码。行号与文件有关。下面是一个代码的实例：

```
44    class producer : public sc_module
45    {
46     public:
47      producer(sc_module_name nm) :
48        sc_module(nm),
49        put_port("put_port")
50      {
51        SC_THREAD(run);
52      }
53      SC_HAS_PROCESS(producer);
54
55      sc_export<put_if> put_port;
56
57      void run()
58      {
59        int randval;
60
61        for(int i=0; i<10; i++)
62        {
63          randval = rand() % 100;
64          cout << "producer: putting " << randval << endl;
65          put_port->put(randval);
66        }
67      }
68    };
file: topics/04_tlm/01_put_sc/put.cc
```

底部的文件名告诉你在哪里去寻找从实例树目录引用这些代码的相关文件，在本例中，它是topics/04_tlm/01_put_sc/put.cc.。行号与文件名有关。

获取手册的套件

AVM验证手册是一个实时的文件，全部都可从Mentor Graphics网站 *http://www.mentor.com/products/fv/_3b715c*.上找到。手册和实例都会定期更新。请检查并更新。。

问题和意见

我们非常希望收到你的来信。请告诉我们你是如何使用该手册，为了未来的版本，也请您提出宝贵的意见和建议。我们也会回答相关材料的问题。你可以将问题和意见email给我们：

cookbook_register@mentor.com

符号说明

本手册有一些带图表的例子用来说明各个验证器件以及它们之间的连接关系。这些采用流程图的形式将数据流和控制流的概念放在一起说明。

传统的符号表是数据流形式。在传统的原理图表示中，器件带有和网线连接管脚。管脚是有方向性的，他们可以是输入，输出或双向的，而且必须和别的管脚相连。例如，一个组件的输出一定是另一个组件的输入。在事务交易级的系统中，我们要同时描述控制流和数据流。事务交易级模型就是基于函数调用的，这在后文中会详细说明。一个组件的函数调用另一个组件的函数时就会产生动作。控制流就指谁调用谁。

将独立的组件通过定义好的接口连接起来是AVM的一个重要原则， 这些观点也反映在我们的符号中。图文符号有三部分：组件，接口接口和连接。

组件

组件用方框表示。

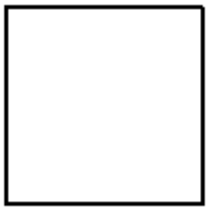


图1-1 组件符号

组件是例化对象，如（在SystemC中）模块，或（在SystemVerilog中）模块，接口，程序块或类。组件总有自己的运行进程。有时，进程的在设计或测试平台中的位置对于理解设计是很重要的。为了表明一个组件有一个或多个进程，我们用圆弧箭头表示。

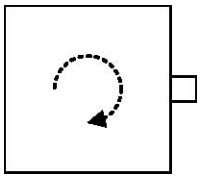


图1-2 有thread的组件

接口

接口是与组件相连的外部可见的连接。所有组件的行为只有通过接口才能访

问和可见。首先就是熟悉的管脚接口。

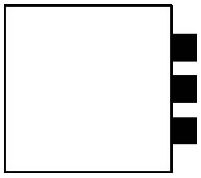


图1-3 带管脚接口的组件

右侧的黑色方框子表示管脚

尽管管脚接口在组件之间以比特传送数据，事务交易接口则在组件之间进行高层次数据传输。

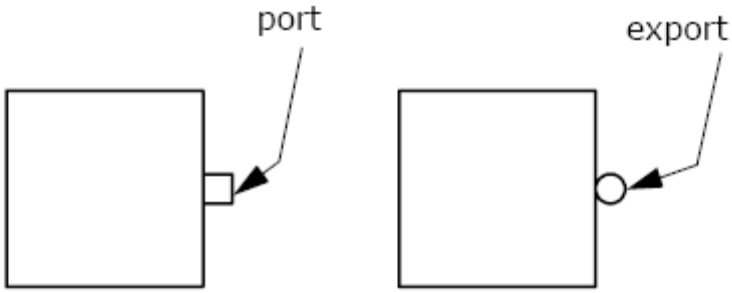


图1-4 事务级界面

图1-4表示事务交易接口的两种变化：一个端端口(port)，一个输出端口（export－－ 在SystemC中用法）。左边的组件有一个事务交易端端口(port)，右边的组件有一个输出。输出端口(export)使得接口变得可见，而必须有一个端端口(port)是与输出端口(export)端口相连。将处理端端口(port)想象为一套还没有解决的函数调用，函数调用一般由输出完成。 端端口(port)和输出端口(export)互为补充，端端口(port)与输出端口(export)相连。而输出端口(export)和输出端口(export)之间或端端口(port)和端端口(port)之间是不能直接相连的。

端端口(port)/输出端口(export)符号表明了组件之间的控制流。是端端口(port)接口调用输出端口(export)上的函数，控制流就是从端端口(port)到输出端口(export)。

互连

就像传统的示意图一样，我们用接口间的线条来表示组件之间的相互连接.带箭头的表示是数据流。

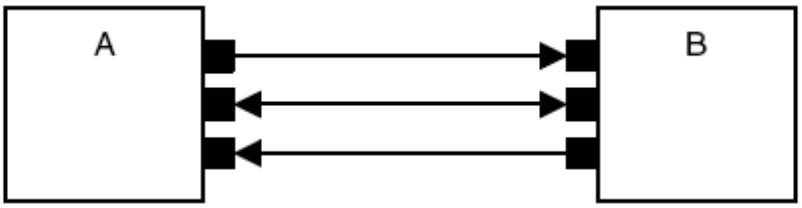


图1-5 管脚级数据流

管脚间的箭头表示组件间数据流的方向。上图表明是从上到下，从A—B，在A和B之间是双向的，也从B—A。

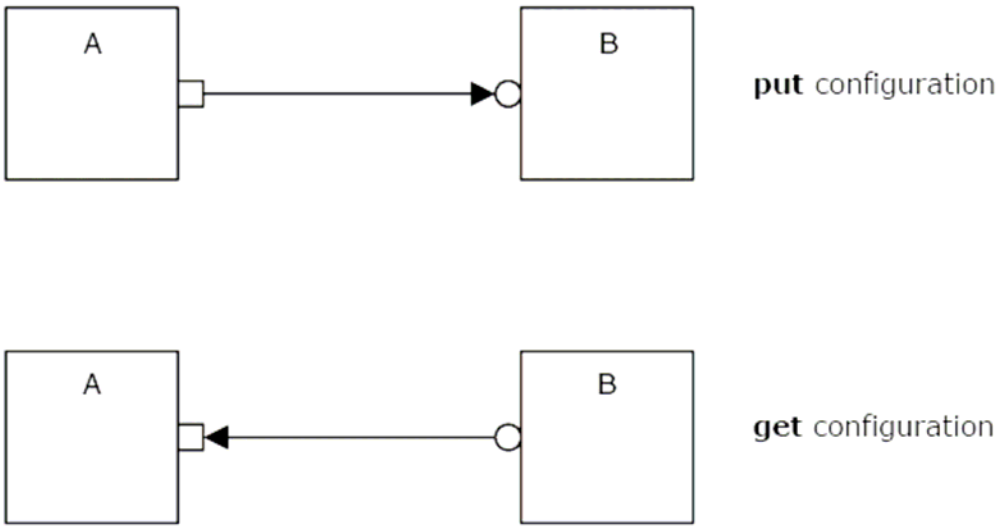


图1-6 事务数据流

图1-6举例说明了两种配置，每一个都具有相同的事务界面和不同的数据流。在这两个配置中，B中的函数包含于A中。A引起B动作。A是施动者，B是对象。在上面的那个配置，A将数据送到B。这就是put动作。在下面的配置中，A将数据传送到B然后又返回，者就是get动作。

通道

事务交易级组件一般通过通道进行通讯。通道就是一个组件，它定义了通讯的语义。最常见的通道就是FIFO。FIFO用于在两个事务交易级组件中节点通讯。为了以图表的形式表示这一点，在组件之间以有个小方框表示FIFO。

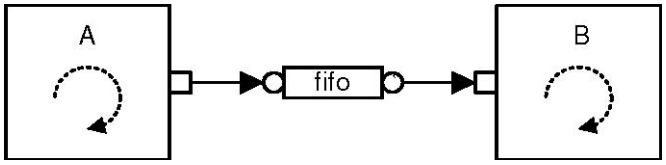


图1-7 通过FIFO通讯的两个组件

FIFO与其它通讯通道一样，输出一个接口。但是为了保持示意图整洁，输出通道上的圆是可选的，一般就省略了。就如希伯来人的元音一样，通道上的输出接口对读者是显而易见的。

图1-7表示两个组件，每一个都有自己的进程，每一个都有事务交易级端端口(port)接入到相应的通道中。组件A将事务交易输送到FIFO通道，组件B则从这个通道中取出事务交易。信息上的数据流箭头表明了哪些组件在读取数据，哪些组件在输入数据。A有一个进程，一个事务交易端端口(port)（与输出端口export相反）和一个指向外的箭头。这说明A是往通道输入事务交易。B也有一个进程和一个事务交易端端口(port)，但是他的数据流箭头是指向组件的，这说明B是从通道读取事务交易。

总结

我们的符号是对传统的RTL原理图符号的扩展。这些扩展有利于我们说明处理接口，通道，组件之间的数据流。利用这些符号，我们可以将事务交易（Transaction）级和RTL组件联系在一个示意图中，这对建立测试平台图表是很重要的。

命名惯例

高质量的代码看起来和感觉起来有一致性。要想获得一致性，就要使用一致

的命名方案。这一节就用我们的例子来说明这一点。

在两个语言SystemC 和 SystemVerilog中创建一个保持命名的一致性的规则存在一点非同寻常的问题。在大部分情况下，对其命名时两种语言都能很好的使用，所以可以构建一个适用于两种语言的规则，其中只存在略微的差异。所有适用的规则对两种语言都是均等的，只有那些明确注明的例外。

一个名字由三部分组成，前缀，主体和后缀。名字的主体部分可以由一个或多个单词组成。各部分－前缀，后缀和主体单词－则由下划线分开。下面是一些名字的举例：

avm_fifo

m_parent_p

finite_state_machine

top

第一个名字的主体是“fifo”并带有后缀“avm”。第二个名字三部分俱全：前缀m_，主体是parent，后缀是_p。第三个名字主体部分由三个单词组成，但是没有前、后缀。最后一个名字也是没有前后缀，其主体是top一个单词。

我们提倡避免使用缩写，如果可能就使用完整的单词。

我们称之为“常规命名方案”，这也是其他命名－特别命名方案的基础。

名字也应当能表明它是哪种事物。以下特殊的命名的规则。

- 类名称

类名称使用常规命名流程。类如果是一个特殊包或库的一部分，他们就应当具有和这个包或库种所有成员相同的前缀。

avm_analysis_port

avm_fifo

- 局部变量

局部变量使用通常的命名流程，但是没有前缀。依据命名对象的类型，它可能由后缀。

- 整数指针

用i，j，k作为整数指针。这是允许使用单一字母作为变量名的一个地方。

```
int i;
int j;
for(i = 0; i < last; i++)
{
for(j = i; j < last; j++)
   matrix[i,j] = compute_entry(i,j);
   }
```

- **类成员**

类成员是局部变量的另外一种形式。它不是存在于函数或任务中，而是存在类中。为了在一个函数，任务或方法中区别类成员和局部变量，采用局部变量规则并加前缀m_表示。

```
class bus_request
   addr_t m_address;
   data_t m_data;
   request_t m_request_type;

endclass : bus_request
```

- **带后缀的局部变量**

它大大提高了程序的可读性，以便很快理解你所阅读表达式中的对象的类型而不必去看其的定义。

o 指针

指针采用局部变量命名规则加后缀_p 表示。

o 句柄

SystemVerilog有句柄，SystemC没有句柄。采用局部变量命名规则加后缀_h表示。

o 类型名称

类型名称采用typedef来创建，采用局部变量命名规则加后缀_t表示。

```
typedef unsigned long int addr_t;

typedef sc_lv<16> bus_t;

typedef sc_port< sc_signal_in_if< sc_uint<32> > > bus_in_port_t;

typedef struct {bit [7:0] value} data_t;
```

- **函数/任务/方法名称和形式参数**

函数，任务，方法及其形式参数采用局部变量同样的命名规则－无需前后缀。形式参数可以用缩写。

```
function send(trasanction_t t, string parent);;
```

- **宏**

宏都采用大写字母，且每个单词都用下划线分开，从而区别SystemC中的宏和有名的常量。宏仅仅是文本，通过预处理器在程序中的适当位置做替代。有名的常量是一个有常数的名称，并且是编译器和调试器都认可的名字。

```
#define MAX_SIZE 100

#define TRANSPORT(req,rsp) send(req);rsp=recv();
```

- **参数**

参数都采用大写字母，且每个单词都用下划线分开。这里可以使用缩写。在SV中，参数或局部参数优先于宏，这样可以减少编译的次数。

```
parameter type T = int;

localparam MAX_SIZE = 100;
```

- **枚举类型和枚举成员**

只有定义枚举类型时，才需要后缀。这时要使用_t。如

```
typedef {mode_unidir, mode_bidir, mode_off} mode_t;
```

枚举类型中包含的所有成员应当具有相同的前缀，表明它们属于哪种类型。

```
enum {color_red, color_blue, color_green, color_purple} color;
typedef enum {req_read, req_write, req_idle} req_t;
```

- **接口**

  modports用后缀_mp

  接口用后缀_if

```
interface bus_if;
   ...
   endinterface : bus_if

class bus_if : public sc_interface

{
   ...

};
```

- **包**

  包用后缀_pkg。

- **端口**

管脚级的端端口(port)应当使用与一个函数的形式参数相同的命名规则。事务交易级端端口(port)应当适当地采用后缀_port 或者 _export。 分析端口（analysis port）采用前缀_ap。如果在一个模块中只有一个分析端口，这是很常见的，那就用ap，无需前后缀。

```
sc_export<control_if> ctrl_export;
analysis_port error_ap, good_ap;
```

第 2 章 验证原理

整个验证过程就是将设计者的意图与观察到设计的行为进行比较来确定他们是否一致。这个基本原理在讨论测试平台，断言，仿真器和所有其它的用于现代验证流程的工具时经常被忽略掉。在阅读下文时一定要牢记这一点。无论何时我们提出观点或举例说明测试平台架构技术，我们都会清晰的指明“参考设计”（Reference Model）和“被测设计（DUT－Design Under Test）”。DUT在某种形式上符合产品一种设计：也就是说，它可以转化为硅片（结合手动和自动方式）。参考设计是抓住设计者的意图，也就是他的设计是打算干什么。参考设计有许多形式，如用一个文件来描述DUT的操作，这是一个

golden model，它包括一种独特的算法，或断言来描述协议。

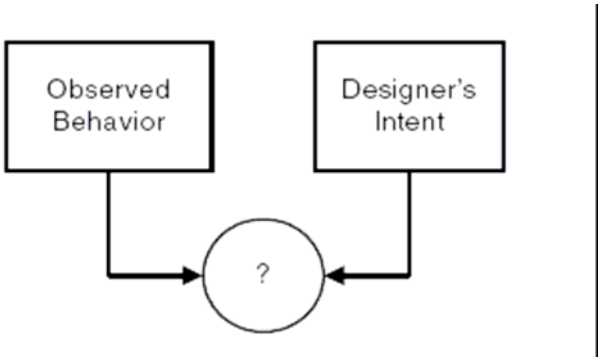


图2-1 比较设计和意图

一个设计的产生，就如问题本身一样，应当被设计工程师正确的理解，它在本文讨论之外。这里我们将讨论限定在如何抓住设计意图，通过比较设计和意图以显示其功能的等价性这些问题上。

两个问题

完成一个验证方案要回答两个问题：它能工作么？我们做完了么？尽管这是非常基础和显而易见的问题，但他们能激发每一个验证流的结构。第一个问题是：它能工作么？来源于基本的求证想法，这在前面一节中已经讨论过。也就是说，设计能否满足设计者的意图。第二个问题是：我们做完了么？就是问是否将设计和意图进行了充分的比较以断定设计是否（或确实）满足意图：如果不是，为什么。

现在看一下如何利用这些问题。图2-2普通流程图表明验证流程的主要元素和在哪些步骤问关键的问题。圆圈里的数字是流程的参考点。

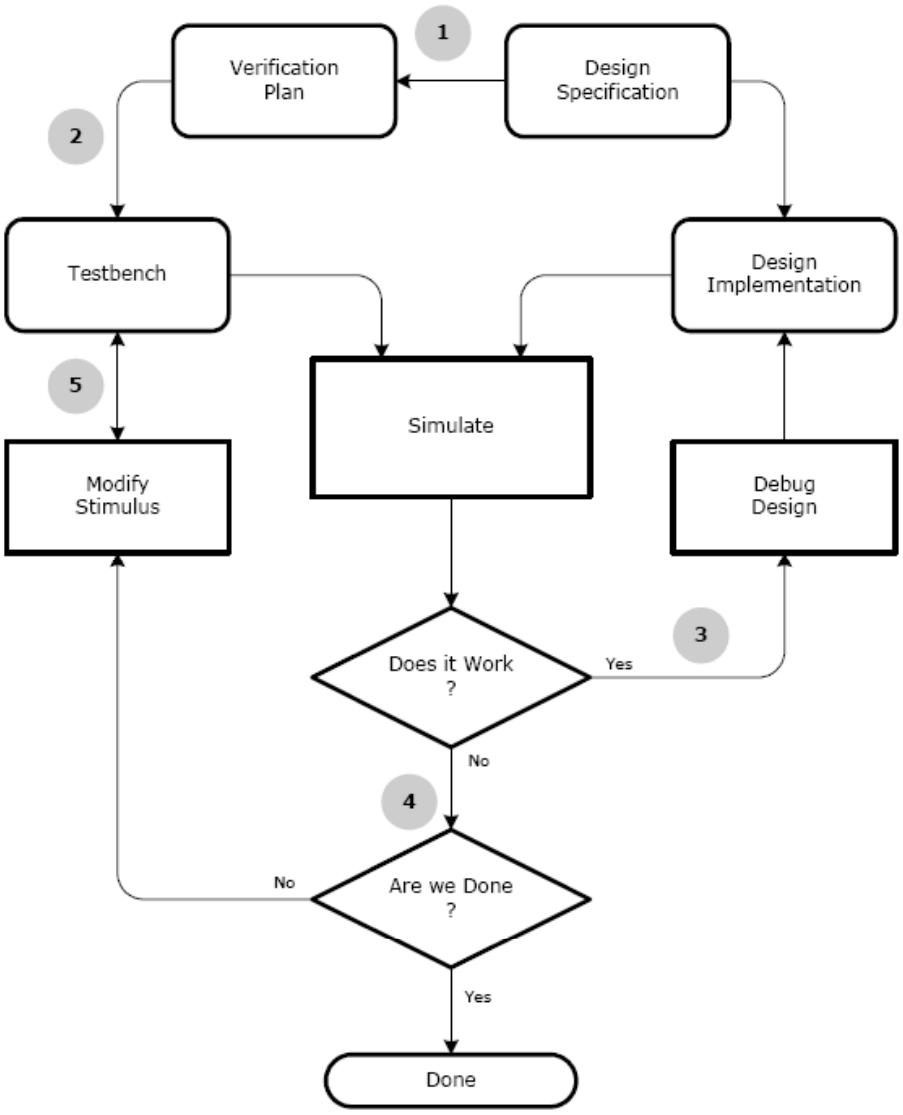


图2-2 通用验证流程

每一个验证方案都是从设计规范开始。设计规范包含对设计解释和意图的详细说明。验证团队用设计规范来构建验证方案。验证方案有一个问题清单，列举所有验证过程需要回答的问题，和说明如何被回答的机制的描述。此外还有一个checklist，列举所有需要回答的问题，它也是测试平台的功能规范。

一旦建立起测试平台，下一阶段就是在测试平台中仿真设计。仿真结果为我们提供了回答上述两个问题的信息。第一个问题是：它能工作吗？如果回答

是“否”则需要调试设计，这就要调整设计来修正遇到的所有的错误，瑕疵或不足。如果设计运行正常就要问第二个问题：我们做完了吗？这个可以通过看覆盖率来回答，进行了多少次仿真。覆盖率的目标在验证计划中应有规定。如果覆盖率不够，则激励或测试平台需要调整以便能实现更多的仿真。在理想情况下，设计没有bugs，覆盖率也总是很充分，以致你就只需看看一次循环，确认对上述两个问题的回答都是“是”。在实际情况下，要做很多次迭代才能得到两个“是”。一个好的验证流程的目标是尽可能减少迭代次数，使得整个验证过程用时较少并且使用较少的资源。

测试平台

这一节里主要讨论测试平台（testbench）的基本特点。一个测试平台就是一个由一套相互相连的验证组件组织得到的软件。它的任务就是回答两个问题：它能工作吗？我们做完了吗？

测试平台的最基本的形式就是将DUT运行的结果与期待的结果进行比较，如图2-3所示。这个过程回答了一个问题“它能工作吗？”如果实际结果与期待的结果相匹配，那这个被测试的设计就能工作了，至少是达到了一部分预期结果。

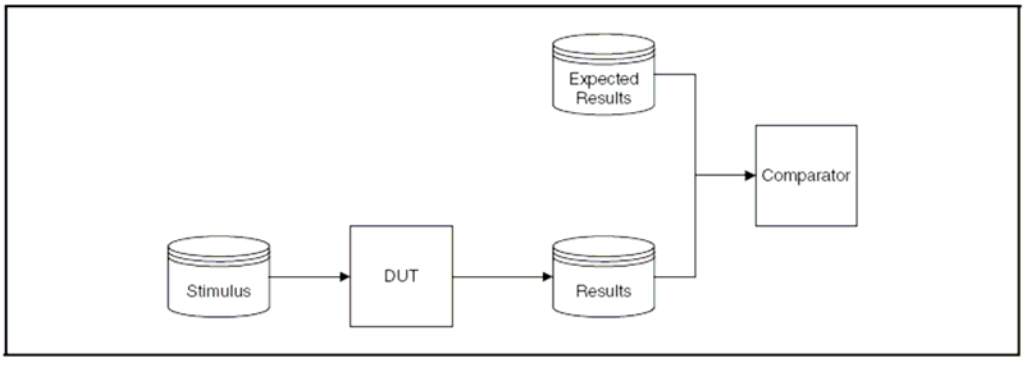


图2-3 非常基本的测试

我们的基本测试平台是有一个激励文件，这个文件将激励应用到设计中并收集结果。一旦出现结果，就将其与预期结果进行比较。

对一个只有几个输入管脚和状态的简单设计，就可以手动直接构建一套预期结果。对任何稍微复杂的设计，这就会是一个费时的任务。最好的办法是构

建一个参考模型来产生预期结果。图2-4展示了从同一个激励中产生预期结果和DUT运行结果的基本结构。如果DUT运行正常，比较器就会显示参考模型和DUT的运行是相同的。

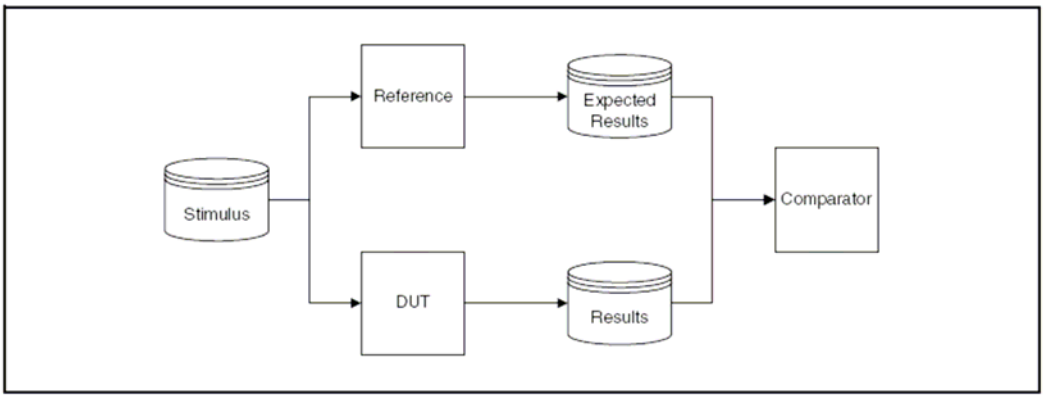


图2-4 带参考模块的基本测试

正如参考模型一样，不可能直接将激励作为一系列向量写出。通常，写一个程序来产生激励更容易。而且也没有必要在文件中存储激励或其结果。相反，激励发生器能很快地产生激励，而比较器则比较这些结果。这个过程更为自动化，此时激励产生过程和结果比较过程就成为测试平台的一部分。

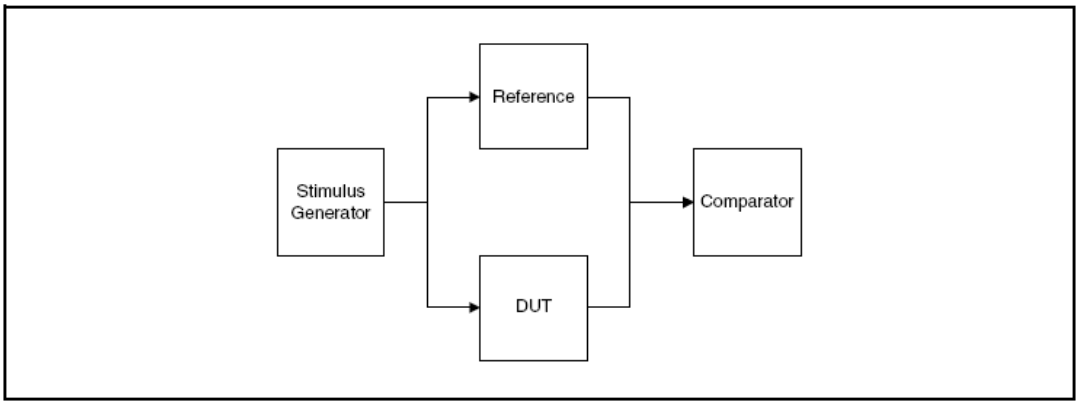


图2-5 自动测试

这里是在普遍的层次进行讨论。当我们将“比较器”用“问题回答器”取代时，就可以看到我们如何收集更深入的，复杂的数据。一个最常见和最有力的问题是“DUT的功能与参考模型的功能匹配吗？”除了那些简单地将预期结果与实际相比较外，许多类似的问题也可以实现。

回答问题“它能验证吗？”的组件通常被称为“记分板（Scoreboard）”。

记分板一词来源于计算机体系结构领域，这里的结构，即所谓的记分板，是用来跟踪系统行为（如一个指令经过通道或高速缓冲存储器，等），藉此得出结论。在验证过程中，记分板指的是一个跟踪验证仿真过程中产生信息的组件。这里收集的信息也可用于下结论：将它输入到我们的参考框架中来回答验证问题。

另一个需要回答的问题是“我们做完了吗？”回答这个问题的信息被收集在“覆盖率收集器”中。它从激励发生器，记分板以及DUT中获得信息。

带记分板和覆盖率收集器的测试如图2-6所示。

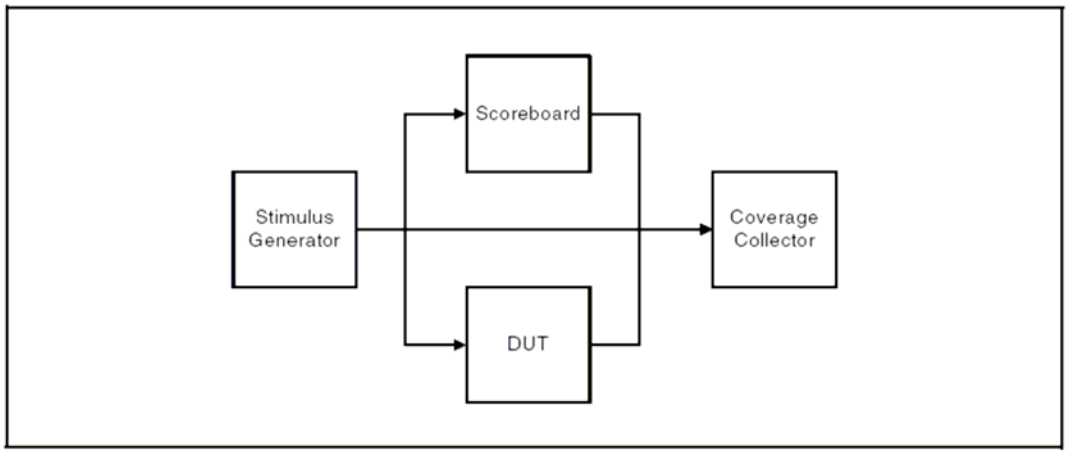


图2-6 通用测试

总结：

- 激励发生器在系统中产生激励。这个激励同时被送到设计和记分板或参考模型上。
- 记分板观察DUT的行为并回答问题“它可以工作吗”。 记分板是参考模型的一种形式。
- 覆盖率收集器收集模拟过程中有关DUT运行的信息。覆盖率收集器中的信息可以回答问题“我们做完了么？”

第一个测试平台

让我们通过一个如何验证数字电子设计中最基本的器件(一个“与”门)的例子进行说明。大家都知道“与”门的工作原理[1]－输出是输入的逻辑“与”。这个器件的这个功能是很简单，几乎不值得做测试平台。正因为其简单，我们可以用它来说明验证过程的一些基本原理，而不需要详细钻研更为复杂的设计。

图2-7 一个2输入的“与”门

我们的任务是证明我们的设计－一个“与”门－能正确工作。为了验证这一点，实际上需要正确地运行一个“与”的功能，我们需要做如下事情：

1. 一个表示DUT的模块－被测试的设计（在本例中，这个DUT是一个“与”门）。
2. 知道本设计打算干什么，它能被编码为一个参考
3. 一些可驱动设计的激励。
4. 一种运用激励的结果与已知正确输出进行比较的方法。
5. 这些都被组合在一个测试平台中，即围绕DUT的组件，它们执行设计并确定该设计是否在正确的工作。图2-8是一个“与”门的测试。

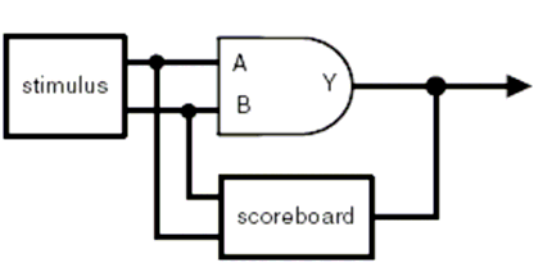


图2-8 测试平台

尽管这个小测试非常简单，但它包含了大部分复杂测试中的关键元素。这些关键元素就是：

1. 如果你不知道“与”门是怎么工作的，或什么是“与”门，那你最好从布尔逻辑和数字设计的入门课开始。

- DUT，被测设计
- 激励发生器：为DUT产生一系列的激励
- 记分板：包括参考模型

记分板观察DUT的输入和输出，执行DUT相同的功能，除非在更高级的抽象过程中，并确定DUT和参考模型是否匹配。

第一个测试先用SystemC进行说明。SystemC未必是构建一个低层次测试平台的理想语言。但是，在这个例子中选取SystemC，并说明测试平台设计是未限定某种语言的。我们可以用VHDL 或 erilog很容易的构建相同的测试平台。尽管这里没有列出，在这个设计和测试的实例树目录中还是包含一个SystemVerilog版本。

这里的DUT是一个具有2输入的“与”门。在SystemC，它表示如下：

```
29    class and2 : public sc_module
30    {
31     public:
32      sc_in<bool> A;
33      sc_in<bool> B;
34      sc_out<bool> Y;
35
36      and2(sc_module_name nm):
37        sc_module(nm),
38        delay_time(1.0)
39      {
40        SC_METHOD(delay);
41        sensitive << A << B;
42        dont_initialize();
43
44        SC_METHOD(compute);
45        sensitive << compute_event;
46        dont_initialize();
47      }
48      SC_HAS_PROCESS(and2);
49
50      void delay()
51      {
52        compute_event.notify_delayed(delay_time, SC_NS);
53      }
54
55      void compute()
56      {
57        Y = A && B;
58      }
59
60     private:
61      float delay_time;
62      sc_event compute_event;
63    };
```

这里的“与”门是一个有2个输入（A和B），一个输出（Y）的模块。运行（run()）计算来自A和B的Y值，该值对A和B都敏感，因此，无论何时，任何一个输入值的改变就会计算出一个新的Y值。在构造函数中调用Dont_initialize()，以告诉仿真器在0次时不要自动调用run()。 这样，执行运行完全取决于输入值A和B的变化。

改变输入A或B并不会直接引起重新计算输出值Y。相反，后面的某个时间有一个预定事件。当该事件发生时才会重新计算一个新的Y值。组合逻辑延迟在SystemC中就是这样实现的。

激励发生器产生直接激励。激励发生器发出的每一个新值都按照一个的规定顺序进行明确计算。然后，就可以看看随机激励发生器，它产生随机值。

```
71    class stim_gen : public sc_module
72    {
73     public:
74      sc_out<bool> A;
75      sc_out<bool> B;
76
77      stim_gen(sc_module_name nm) :
78        sc_module(nm),
79        stimulus(0)
80      {
81        SC_METHOD(run);
82      }
83      SC_HAS_PROCESS(stim_gen);
84
85      void run()
86      {
87        A = stimulus[0];
88        B = stimulus[1];
89        stimulus++;
90        next_trigger(3, SC_NS);
91      }
92
93     private:
94      sc_int<2> stimulus;
95    };
```

激励发生器的要旨就是产生数值用于输出，A和B就输入到DUT中。一个2比特的 sc_int命名为stimulus，包含了要赋给A和B的值。该值在每一次迭代中都会增加， 将低次比特赋给A，高次比特赋给B。

激励发生器在一个无限循环中工作，但在代码里并不出现循环结构！这是怎么工作的？答案包括两部分－开始循环和连续循环。注意run()对任何输入都不敏感（主要是因为激励发生器没有任何输入，只有输出）。列举了run()的敏感清单后，在回顾一下“与”门模块，调用dont_initialize()，它通知仿真器在运行次数为0时不要调用run()。这里，为了开始循环，从调用dont_initialize()中跳出，这就暗中通知了仿真器在运行次数为0时去调用run()。

好了，现在程序开始了，现在是如何让它重复。调用函数next_trigger()可以达到这个功能。它确定再次调用当前方法的时间表。调用next_trigger()的参数就是到下次激励的时间延迟。在本例中取1ns，但这并不是临界值。唯一的约束就是它必须大于0。延迟为0就不能使“与”门动作，这就很容易引起激励发生器进行无限循环直到手动停止仿真器，或它自己最终放弃（一般情况下，如果在同一时刻中执行很多的delta cycle，它就会自动放弃循环）。

在这个小测试平台中最有趣的模块是记分板。它观察DUT的行为，并且无论它是否正确运行都会反馈信息[1]。值得注意的一件重要事情是记分板的结构与DUT的有惊人的相似。当你考虑到记分板的任务是跟踪DUT的行为并确定DUT是否按预期进行工作时，这一点就不难理解了。

```
100   class scoreboard : public sc_module
101   {
102    public:
103     sc_in<bool> A;
104     sc_in<bool> B;
105     sc_in<bool> Y;
106
107     sc_event input_change_event;
108
109     scoreboard(sc_module_name nm) :
110       sc_module(nm)
111     {
112       SC_METHOD(input);
113       sensitive << A << B;
114
115       SC_METHOD(checker);
116       sensitive << check_event;
117       dont_initialize();
118
119       truth_table[0]= 0;
120       truth_table[1]= 0;
121       truth_table[2]= 0;
122       truth_table[3]= 1;
123
124       cout << "\t\tA B : Y" << endl;
125     }
126     SC_HAS_PROCESS(scoreboard);
127
128     void input()
129     {
130       cout << sc_time_stamp() << "\t\t" << A << " " << B
131            << " : applying stimulus" << endl;
132       check_event.notify_delayed(2, SC_NS);
133     }
134
```

1. 对任何比“与”门复杂的事情，监视器和反应寄存器将会成为测试中的独立组件。对一个小的“与”门测试，这一阶段就很麻烦，而且会影响我们要阐明的基本原理。

```
135     void checker()
136     {
137       unsigned truth_index = (A << 1) | B;
138       // check Y against truth table
139       bool match = (Y == truth_table[truth_index]);
140       cout << sc_time_stamp() << "\t\t" << A
141    << " " << B << " : " << Y;
142       cout << "   " << (match ? "yes!" : "<<--no") << endl;
143     }
144
145    private:
146     bool truth_table[4];
147     sc_event check_event;
148   };
```

对记分板要注意的第一件事情就是所有的管脚都是输入。记分板对设计不会产生动作，它只是被动的观察DUT的输入和输出。这里的记分板有两个方法，input_change()和checker ()。Input_change()对输入A和B都敏感，指向DUT中的输入。. Checker()则对一些稍微神秘的check_event敏感。这都是哪些事件呢？为了回答这个问题，首先来看看记分板的目的是作什么。我们希望记分板告诉我们与DUT的的输入相关的输出是否正确。其最简单的办法就是每次输出改变时记分板都工作，检查并看看输出是否正确。但有时A和B改变而Y并不改变，这是为什么？仿真器执行了一个简单的最优化：如果一个信号的新值和当前值相同，则更新的事件就不是预计的。为什么要不厌其烦的将a 1改为a 1或将a 0改为a 0呢？这样做仅仅是运行计算循环而不会达到任何结果。因此，例如如果A是0，B是1，则改变输入值，这样A现在是1， 而B则是0，Y值不会改变。这是因为与（0，1）等于与（1，0），他们都等于0。在这种情况下，或其它输入改变并不引起输出Y改变的情况下，我们还想知道答案是否正确。仅仅因为数值不变并不能成为相信结果正确的理由。我们需要证实Y的正确性。

用input()得到仿真器的最优化方法。任何时候调用input()，A或B就会改变。这需要做一件简单的事情：在一段时间以后引起checker()动作。后面将要讨论具体是多少时间以后。它通过预计一件事情check_event来作为将来的时间。Checker()只对check_event敏感，也就是说检查Y = A & B是否正确将不再取决于Y的改变。

“与”门的延迟时间是1ns。任何时候输入改变，输出将在1ns后出现。为了确保记分板只检查有效值，（那些固定在时间继电器后的值）记分板输入改变2ns后调用checker()，这就保证了正在检查的Y值确实是由于A或B的改变引起的真值，而不是由于中间计算产生的假值。为此，要确定check_event，在输入改变1ns后来触发。但是，如果checker()和Y值改变正好在同一时间发生，就没有办法预计是哪一个先开始。如果checker()在Y更新之前运行就会查到一件错误的事情。为了避免这个问题，对于1ns就跳出检查，这样就使得Y在执行checker()之前就进行强制更新。激励发生器每3ns给A和B产生一个新输入，

这样就给“与”门充分得时间去稳定，也给记分板充分的时间去检查。

下面是一个顶层模块，它完全结构化。它包括DUT，记分板，激励发生器以及与它们相连得必要代码的实例。

```
153   class top : public sc_module
154   {
155    public:
156     sc_signal<bool> A;
157     sc_signal<bool> B;
158     sc_signal<bool> Y;
159
160     top(sc_module_name nm):
161       sc_module(nm),
162       a("and"),
163       s("stim"),
164       b("score")
165     {
166       a.A(A);
167       a.B(B);
168       a.Y(Y);
169
170       s.A(A);
171       s.B(B);
172
173       b.A(A);
174       b.B(B);
175       b.Y(Y);
176     }
177
178     and2 a;
179     stim_gen s;
180     scoreboard b;
181   };
```

对这个仿真运行50ns可以得到如下结果：显示的信息成对出现。每对中的第一个说明采用了新的激励。第二个检查所采用的激励是否导致了正确的反应。

```
                    A B : Y
# 0 s           0 0 : applying stimulus
# 2 ns          0 0 : 0  yes!
# 3 ns          1 0 : applying stimulus
# 5 ns          1 0 : 0  yes!
# 6 ns          0 1 : applying stimulus
# 8 ns          0 1 : 0  yes!
# 9 ns          1 1 : applying stimulus
# 11 ns         1 1 : 1  yes!
# 12 ns         0 0 : applying stimulus
# 14 ns         0 0 : 0  yes!
# 15 ns         1 0 : applying stimulus
# 17 ns         1 0 : 0  yes!
# 18 ns         0 1 : applying stimulus
# 20 ns         0 1 : 0  yes!
# 21 ns         1 1 : applying stimulus
# 23 ns         1 1 : 1  yes!
```

这个简单的测试说明了激励发生器，参考模型和记分板的用途。尽管DUT是一个简单的“与”门，但构成一个完整测试平台的所有基本部件都在这里。

第二个测试平台

前面的例子采用一个组合逻辑设计（一个“与”门），说明了基本的验证概念。很自然，组合逻辑设计并不提供任何状态。在这个例子里，我们看一个稍微复杂的设计，它提供状态值并用了一个时钟去引起状态之间的变换。自带状态的同步（时钟）设计在大大小小的设计中都非常常见。

图2-9所示的设计，是一个3比特的带有异步置位的计数器。每次时钟脉冲为高时，计数器就增加。这个设计由3个触发器组成，每个振荡器提供一个单比特计数器。触发器由一些组合逻辑连接在一起形成一个计数器。当T输入高时，每个触发器就翻转，当T低时，触发器就保持一个当前状态。当把异步置位设置到0，则触发器恢复到0状态。

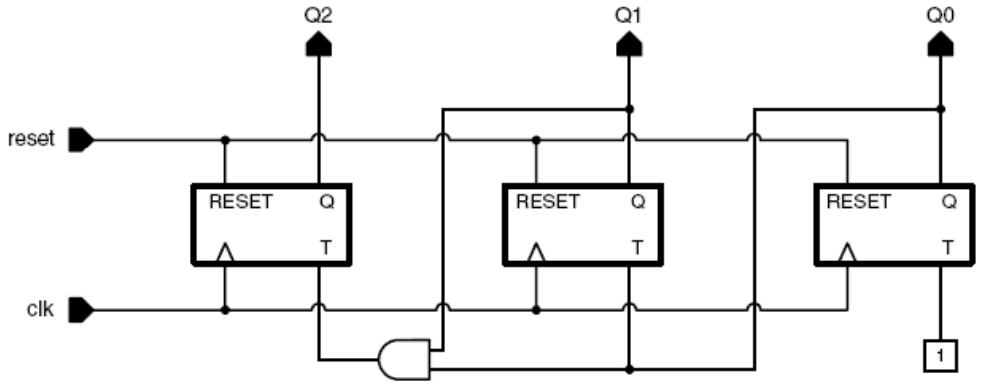


图2-9 3位计数器

计数器的代码包含2个模块：一个是简单的触发器，另一个用必要的粘连逻辑（Glue Logic）将触发器连接在一起形成一个计数器。

```
23    module toggle_ff(t, q, reset, clk);
24      input t;
25      input reset;
26      input clk;
27      output q;
28
29      wire t;
30      wire reset;
31      wire clk;
32      bit q;
33
34      always @ (posedge clk)
35        begin


36          if(t == 1)
37            begin
38              q = !q;
39            end
40        end
41
42      always @ (negedge reset)
43        begin
44          q = 0;
45        end
46    endmodule
```

计数器包括3个触发器和一个“与”门。

```
51    module counter(q, reset, clk);
52      input reset;
53      input clk;
54      output q;
55
56      wire clk;
57      wire reset;
58      wire [2:0] q;
59
60      toggle_ff ff0 (1'b1, q[0], reset, clk);
61      toggle_ff ff1 (q[0], q[1], reset, clk);
62      toggle_ff ff2 (t2,   q[2], reset, clk);
63      and a1 (t2, q[0], q[1]);
64    endmodule
```

这个设计非常直接，但却具有实际设计中的一般特点，为了正确验证必须考虑这些特点。最主要的特点就是这个设计是由时钟驱动并寄存内部状态。而前面那个例子中的“与”门则不寄存任何状态，所有关于设计正在是做什么的信息都可以从管脚收集。但带内部数据的设计就不是这样。这个差别反应在记分板的设计上。图2-10表明3字节计数器测试的结构。

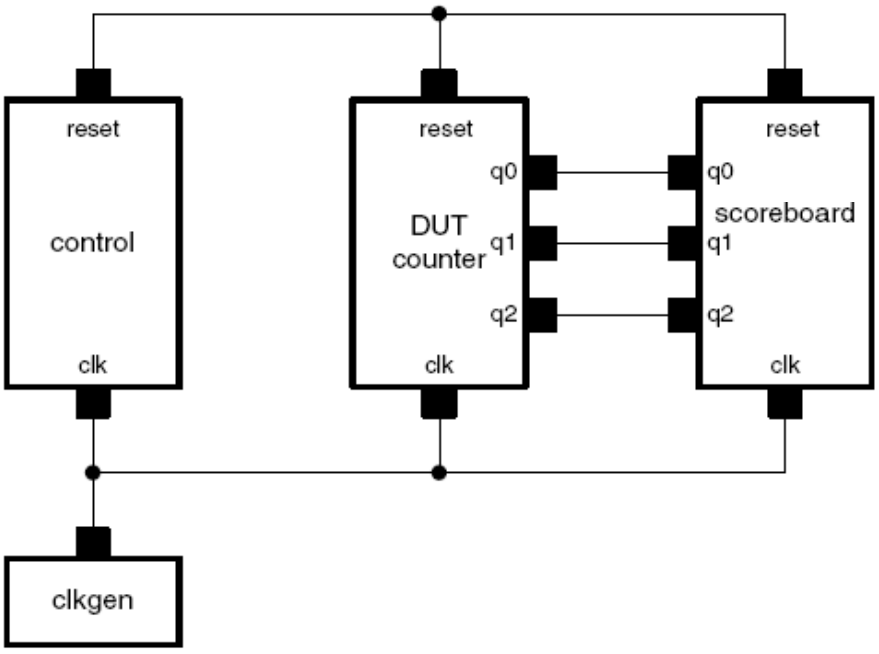


图2-10 3位计数器的测试结构

在许多方面，对3比特计数器的测试平台类似与对“与”门的测试平台。两者都有记分板，用来观察设计是作什么的并确定其工作是否正确。两者都有一个驱动DUT的器件。对“与”门是激励发生器，对3比特计数器则是控制器。3比特计数器是一个自由运行的器件。只要它与时钟相连，它就会一直计数。

因此无需激励发生器，而是由控制器管理DUT和测试的运行。控制器提供初始复位让它可以在一个确定的数值开始计数。它也能够在适当的时候停止仿真。

```
module control(clk, reset);
   input clk;
   output reset;
   bit reset;

   initial
      begin
         @clk;
         reset = 0;
         @clk;
         reset = 1;
         #200;
         $finish;
      end
endmodule
```

记分板必须跟踪DUT的内部状态。这是由变量count完成的。就如DUT一样，当进行重置时，计数设定为0。每一个时钟循环计数就增加，新的计数就与来自DUT的计数进行比较。

```
module scoreboard(q, reset, clk);
   input q;
   input reset;
   input clk;

   wire reset;
   wire clk;
   wire [2:0] q;

   int count;

   always@(negedge reset)
      begin
         count = 0;
      end

   always @(posedge clk)
      begin
         count = count + 1;
            if(count > 7)
               count = 0;
         #0;
         if (count == q)
         $display("time=%t q=%b count=%d match!", $time, q, count);
            else
         $display("time=%t q=%b count=%d <-- no match", $time, q, count);
      end
endmodule
```

记分板有一个计数器的高层次的模型。它用整型变量和加号(+)运算符组成

计数器，取代触发器和“与”门。每次时间脉冲，计数增加，就像RTL计数器。它也进行比较，看其内部计数与计数器DUT的输出是否一致。

为了全面了解，我们给出了时钟发生器和顶层模块。时钟发生器简单的将时钟初始化为0，然后每5ns翻转一次。

```
module clkgen(clk);
   output clk;
   bit clk;

   initial
      begin
         clk = 0;
      end

   always
      begin
         #5;
         clk = !clk;
      end
endmodule
```

The top level module is typical of most testbenches. It connects the DUT and the testbench components together.

```
module top;
   wire [2:0] q;

   clkgen  ckgn (clk);
   counter cntr (q, reset, clk);
   control ctrl (clk, reset);
   scoreboard score (q, reset, clk);
endmodule
```

我们说明了一个简单的测试平台，它包含使用在更为复杂的测试平台中的基本部件。自带状态的时序逻辑设计要求一个与DUT同时运行的记分板。它和DUT执行相同的计算，但是是在更高的抽象层次上。通过记分板，记分板也将它自己的计算结果与来自于DUT的输入进行比较。

第 3 章 AVM 综述

Merriam-Webster是这样定义方法学（Mothodology）的：“由方法，程序，运作概念，规则和假定构成的，在某种科学，艺术或学科采用的的个体[1]”。在我们的案例中，科学/艺术/学科就是指功能验证－验证电子系统的艺术和科学。AVM是一种构建软件的方式，称为测试平台，它的功能就是验证电子设计。

验证构件

就如设计是一个设计构成的网络，测试平台是一个验证构成的网络。不仅要构建判断组件，我们推荐构建一个由在这一节说明之外的组件构成的测试。用定义好的行为和接口来调试更为容易。用这种组件也可提高再利用性。

同心圆的测试平台架构

AVM测试是分层组合的。各层定义如图3-1所示。

1. “methodology.” *Webster’s Third New International Dictionary, Unabridged*. Merriam-Webster, 2002. http://unabridged.merriam-webster.com (7 May 2006).

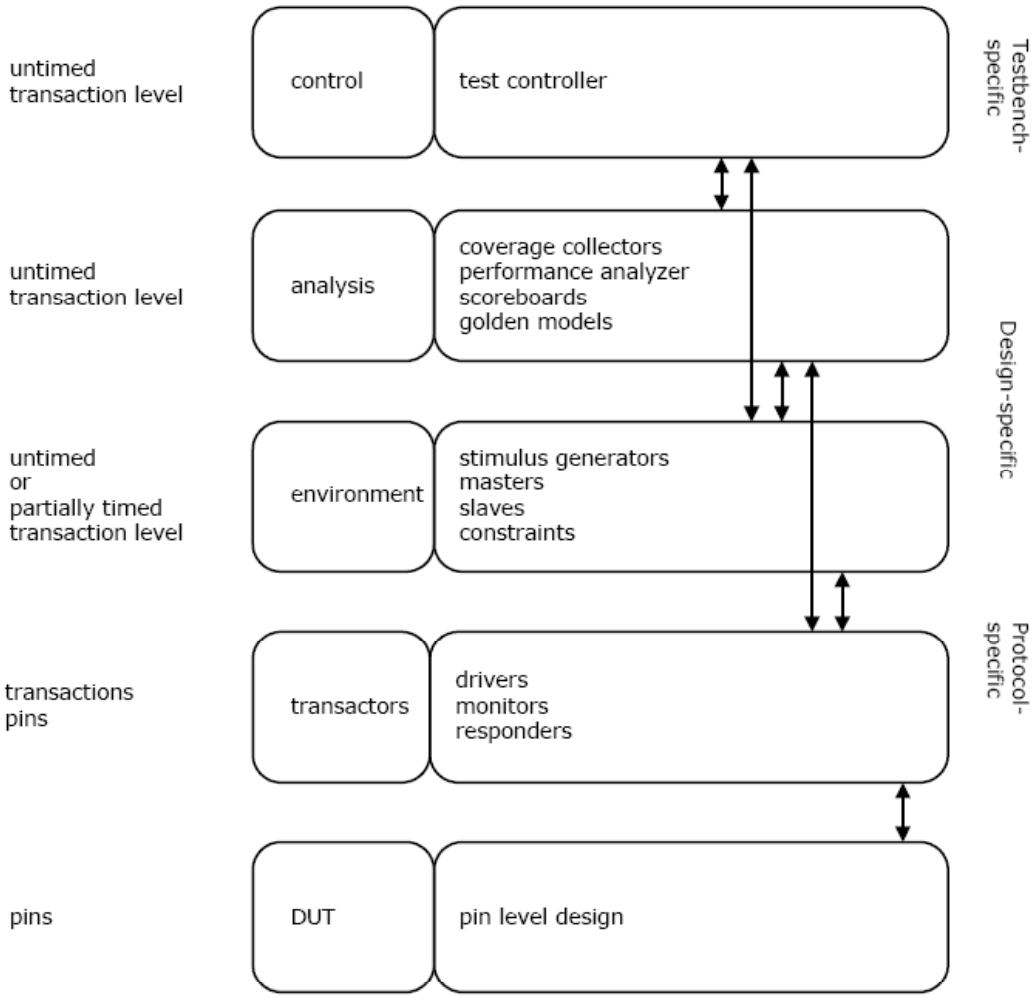


图3-1 AVM测试平台架构的各层

底层是管脚级的DUT。其上一层是一组处理器（Transactor）层，用于在管脚层行为和交易事务流之间的转变。处理器层以上的组件都在事务交易级。你也可以将测试平台看成是一套同心圆组合的结构。

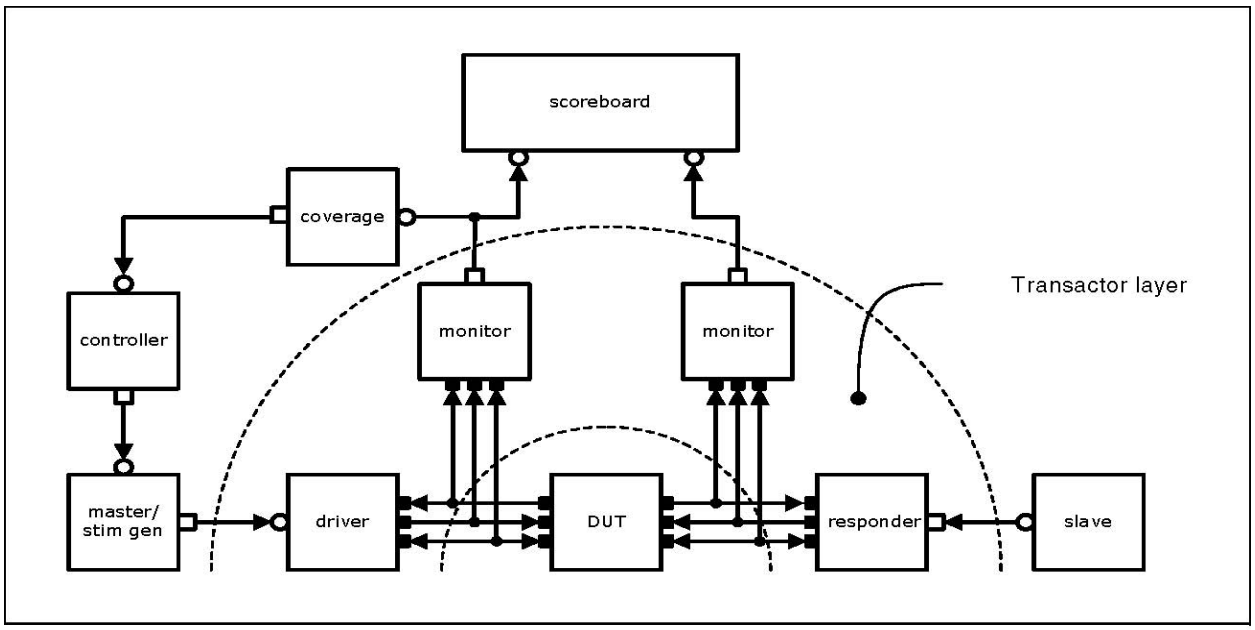


Figure 3-2. Concentric Testbench Organization

处理器（Transactor）

测试中处理器的任务即使将事务交易转化为管脚级行为或反过来，将管脚级行为转换为事务交易。处理器的特点是至少有一个管脚级的接口和一个事务交易级的接口。处理器有许多种形状，颜色和类型，这里主要讨论监视器，驱动器和响应器。

监视器（Monitor）

监视器就是监控总线。它监视针并将其摆动转换为事务交易。监视器是被动的，也就是说它不影响DUT的运行。

驱动器(Driver)

驱动器将事务交易转变为管脚级的动作。

响应器（Responder）

响应器很象驱动器，但他是响应管脚的动作而不是驱动动作。

环境组件

环境是提供给DUT运行所需要的一套组件。环境组件对为DUT产生的交易事务。它们都是事务交易级组件，也只有事务交易级接口。其产生激励的方式依器件的不同而有差异。下面会看到三种类型的环境组件：激励发生器，主器件Mastors）和从器件（Slaves）。

激励发生器（Stimulus Generator）

激励发生器为受激励的DUT产生交易事务流。激励发生器可以是随机的，定向的或定向随机的。它们可以自由运行也可以被控制。它们可以独立也可以同步使用。最简单的激励发生器是随机化请求对象的内容，然后将该对象传送到驱动器。

特定发生器是激励发生器的一种形式。它产生定向的或定向随机顺序以便在DUT上执行一个特定的功能或运行一个特殊的情节，而不仅仅是产生一个随机请求信息流。

主器件（Mastors）

主器件是一个双向组件，它发出请求并接收响应。主器件引发行为。正如特定发生器一样，它们可以发送定向的或定向随机处理的独立的随机处理信息或顺序。主器件依据响应来确定其下一阶段的动作。

从器件（Slaves）

从器件向主器件一样，也是双向组件。它们接受请求并返回响应（而主器件是发送请求和接收响应）。

分析组件

分析组件接收关于测试所进行的内容的信息，并利用这些信息来确定被测试设计的正确性和完整性。两种常见的分析组件是记分板和覆盖率收集器

记分板（Scoreboard）

记分板用来确定DUT的正确性。记分板监控进/出DUT的信息，并确定DUT对激励是否发出了正确响应。

覆盖率收集器（Coverage Collector）

覆盖率收集器是对事件计数。它深入交易事务流，计算交易事务或交易事务的的各个方面。其目的就是确定仿真的完整性。覆盖率收集器中对特定事件计数取决于设计和特定的测试。覆盖率收集器计数的一般时间包括：处理次数，在地址特定段处理的次数，错误数等，不胜枚举。

覆盖率收集器的计数也可作为完整性检查的一部分。例如，覆盖率收集器可以记录一个开始跟踪数据的运行平均和标准偏差。它也能记录错误/正确处理的比例。

控制器

控制器是测试和组合行为的主要进程。典型的控制器接收来自记分板和覆盖率收集器的信息，并将信息传送到环境组件。

例如，一个控制器可以使激励发生器开始运行，然后等待来自于覆盖率收集器的信号，该信号通知它测试何时结束。然后控制器就以此控制停止激励发生器。这个主体可能会有很多具体的变化形式。控制器给激励发生器提供一套初始约束然后使激励发生器开始运行。当信息包类型比例达到某一特定值时，覆盖率收集器给控制器发送信号。控制器会给激励发生器发送一套新的约束而不是终止其运行。

两个域

我们可以看到测试平台中的一套组件属于两个不同的域。操作域的组件包括DUT，它运行DUT。有激励发生器，BFM（Bus Function Model）和一些类似组件产生激励提供响应来驱动仿真。分析域的一套组件是用来监视和分析操作。DUT，响应器，驱动交易处理以及环境组件直接传送或响应驱动器和响应器，它们组成操作域。测试平台组件的其它部分，监视器，处理器，记分板，覆盖率收集器和控制器共同组成分析域。

数据必须以一定的方式从操作域传送到分析域而不受DUT运行的影响以便保留事件时序。这是由一个专门的称为分析端口的通道来完成的。分析端口是一种特殊的处理端口，这里，发布者将数据广播给一个或多个注册者[1]。一旦有新数据，发布者就立即给注册者信号。

分析端口的一个重要特点就是它们有一个独立的接口功能write()。

分析fifo是联系分析端口和分析组件的通道，它非常庞大。这就保障了发布者的通畅并将数据以与数据产生的迭代循环一样的精度存放到分析fifo。分析端口和分析fifo将在第7章进行详细说明。

1. 采用观察面向对象的设计方式来运行分析端口。设计方式包括观察器方式，在《设计方式：可再利用的面向对象软件组件》一文中有详细的讨论。文章作者是 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (the gang of four), Addison –Wesley，发表于 1994 年 10 月。

面向对象编程风格

作为组件的对象

软件工程师不受电磁物理学的约束，一直在寻求构建一个可再复用，可互换，健壮的组件。关于这个问题的思路和技术的主体就是所谓的面向对象编程技术。面向对象编程（oop）其中心观点就是程序是按照一套相互作用的对象组织在一起，每一个都有自己的数据空间和一套功能性函数。

对象的一个基本特点是接口，即对象的使用者可以使用的方法的集合。方法（Method）是运行对象的函数，也是唯一访问对象里的数据的途径。一个完整的对象是不允许访问其内部数据的，除非是通过其接口。

有趣的是，HDL使用类似的概念操作，如Verilog 和VHDL。例如在Verilog 中的模块，也是互为对象，有自己的数据空间和自己的任务及函数。正如面向对象程序中的对象一样，模块的每个实例就是一个独立的副本，所有实例都共享一套相同的任务和和函数，也共享同一套接口，但其包含的数据是独立

的。模块由接口控制。Verilog不支持继承，类型参数化而且是静止的，所以它不适合真正的面向对象编程。表3-1比较了Verilog, SystemVerilog, 和C++的类特性。

Table 3-1. Comparison of Classes and Modules

Feature	Verilog Modules	C++ Classes	SystemVerilog Classes
local data space	yes	yes	yes
function interface	kind of	yes	yes
port interface	yes	no	yes
inheritance	no	yes/multiple	yes/single
type parameterization	no	yes	yes
dynamic	no	yes	yes

对象和模块之间的相似使我们可以在硬件之间采用类对象。我们可以创造验证组件作为类的实例，这给类和硬件元件的连接提供了可操作性。SystemVerilog的设计师在用类扩展来Verilog时利用了这层关系，使得类可以象模块一样工作。

SystemVerilog的这一特点使得可以创建虚界面接口。虚拟接口是接口的指针（这里的接口指的是SystemVerilog的接口结构）。类可以在它内部包括仍未存在的接口的指针。这就使得类对象既可以驱动管脚动作，又可以对其动作产生响应。SystemC模块可以像类一样操作，并允许管脚排成端口列，提供同类结构。

工具条：Simula 67

类对象和硬件仿真之间长期存在着一定的关系。Simula 67[1]是最早的面向对象的程序语言之一，开发它的目的就是构建离散时间的模型。模拟语言具有类对象的概念和模拟内核。它甚至有一种PLI来与外部的Fortran程序相连。模拟语言提供关键字DETACH 和 RESUME，这样就可以产生程序并重新连接程序，一类分岔汇接。它有一个特殊的内置类称作SIMULATION，具有事件清单的特点。

尽管在Simula 67没有用到对象或面向对象这样的用语，但是所有现代的面向对象的程序都可以追溯到这个早期的程序语言。离散时间模拟语言也可以追溯到Simula 67。在许多人看来，将面向对象的程序和硬件模拟放到一起似乎是个新想法，其实这两个概念本来就是在一起的，只是在后来分道扬镳了而已。将面向对象的程序和离散时间仿真器放在一起便构成了一个完整的循环。

Oslo大学信息学院的Johan Dahl and Kristen Nygaard说道[2]：

Simula 67在世界各地仍被广泛使用，但是，它的主要影响是通过引入程序的一个主要目录，更广泛的影响着面向对象的程序设计。Simula概念的重要性始于19世纪八十年代早期，在讨论抽象数据类型和执行同时发生的程序模块时。Simula 67和修订的Simula都曾被用于VLSI电路(Intel, Caltech, Stanford)设计中。Xerox PARC的Alan Kay 工作组将Simula作为他们研发Smalltalk（19世纪八十年代第一个语言版本）的平台，通过将绘图客户接口和交互程序执行结合起来，大大的拓展了面向对象的程序设计。Bjarne Stroustrup通过将Simula的关键概念用于C程序语言，从而开始开发C++（19世纪九十年代）。Simula在程序构成再利用领域和构建程序库领域也激起了大量的工作。

1 Lamprecht, Gunther, "Introduction To Simula 67," Vieweg, 1983

2 http://heim.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_OO_start.html

继承

在一个面向对象的程序中，对象之间是互相关联的。许多种关联是有用的。其中，两种最常见的关联是IS-A 和 HAS-A。

当一个对象是另一个对象的引用（reference）或指针（pointer）时，这两个对象就通过HAS-A关联。在下面的例子中，A有一个B的指针。

由继承产生的对象通过IS-A构成。继承对象就认为是子类，或者是父对象的一个特殊版本。

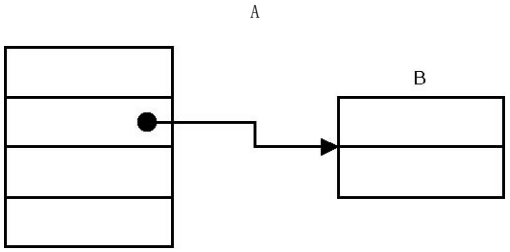


图3-3 HAS-A 关联实例

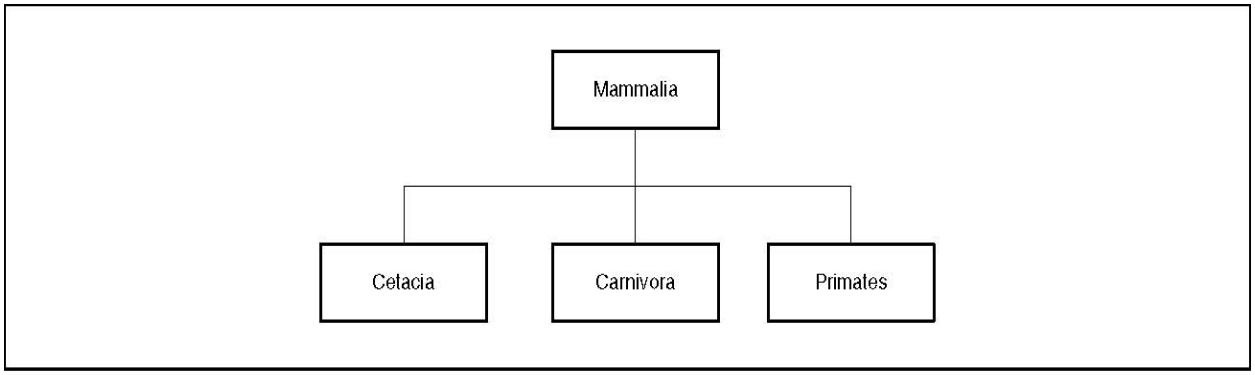


图3-4 IS－A：哺乳动物分类

为了说明继承概念，来看看哺乳动物分类的一部分。鲸类动物，食肉动物或灵长类动物都是哺乳动物。这些不同的动物都具有哺乳动物的共有特点。然而鲸类动物（鲸，海豚），食肉动物（狗，熊，浣熊）和灵长类动物（猴，人类）又都有其各自独特和明确的特点。熊和鲸都是哺乳动物，但各自又属于它们自己的类别。用面向对象的术语来说，熊IS-A食肉动物，食肉动物IS-A哺乳动物。熊这个对象具有哺乳动物和食肉动物的特性，也有它区别于其它食肉动物的特性。

当用继承将两个对象放到一个计算机程序中时，子集对象具有父类父类对象的特点，通常还有它自己额外的特点。在下面的例子中，对象B源于对象A。

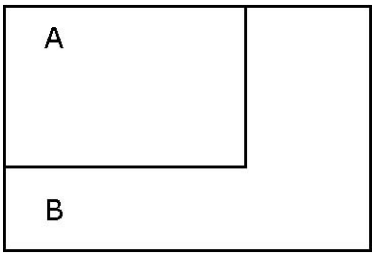


图3-5 IS-A关联实例

下面是在SystemVerilog中，这个组合的结构：

```
class A;
   int i;
   float f;
endclass

class B extends A;
   string s;
endclass
```

既然B源于A，它就具有A的所有特征。任何B的实例不仅包含字符串s，而且还包含浮点数f和整数i。

这只是对面向对象的程序中继承概念的简要介绍。对于继承和其它面向对象的概念和技术的更为深入的讨论，请指针附录B中的指针书目中的文章。

AVM使用继承为验证组件提供一致的函数。当你构建一个从基础类中导出的对象，你就可以利用基础类中的所有函数。AVM套件包含了一个基础类的库。你可以从库中的类来创建自己的类。通过继承，你的新类就可以利用基础类的所有函数。

接口

为了使得对象或模块可复用和可继承，通过定义好的接口来操作就显得很重要。这可以保证其内部的状态保持一致并允许内部函数产生关于状态的假设，这样，即使这个状态被外部函数改变，但每一个内部函数还能同样的执行一致性检查。

就如花园的软管有外螺纹和内罗纹管件来连接水龙头，台灯有与墙上的电源插座相匹配的插头一样－对象接口也有两部分，且必须匹配（用程序语言来说，又叫绑定）以便两个对象能联系起来。接口的一面提供函数，而另一面则请求这个函数。只要这两个匹配，就可以在两个对象之间交换数据和控制。requires/provides的范例是连接在事务交易级和硬件（管脚）级的验证组件的基础。首先来看看一个面向硬件的例子。

```
module requires(input clk, input request, input addr, output data);
...
endmodule
```

requires模块需要一个时钟，请求比特和地址，如果在正确的时间，以正确的顺序接收了这些事件，这个模块就会用一些数据做出响应。

```
module provides(output clk, output request, output addr, input data);
...
endmodule
```

管脚级接口给provides模块提供所有需要绑定到requires模块上的所有信息。

事务交易级接口工作的机理就有点不同，但原理是一样的。

```
class interface : public sc_interface
{
   public:
      virtual data_t fcn(addr_t addr) = 0;
};
```

接口类是“纯虚”，也就是说它不做任何操作，也不占用存储器。它提供定义了数，顺序，参数类型和返回值的类型的函数原型。virtual这个关键字指这个函数在目标对象中可以重载。virtual这个关键字在SystemVerilog也有同样的意义。接口类提供所有函数的函数原型，这些函数组成接口。在我们的例子里，有一个称作fcn的接口函数。提供类是用来提供接口函数的执行结果，而命令类则调用它，这样，在某处就可以命令执行。一个顶级的类约束命令和供给对象。

```
class provides : public sc_module, public interface
{
   public:
   ...
      data_t fcn(addr_t addr)
      { ... }
};
```

Here we provide an implementation of fcn

provides类提供fcn()的执行，requires类通过端口（实际上是一个输出口）调用fcn()。

```
class requires : public sc_module
{
   public:
   ...
      sc_export<interface> port;

      void run()
      {
         ...
         data = port->fcn(addr);
      }
};
```

Here we establish the requirement for fcn via the interface

Here we invoke fcn() through the port which owns the requirement that such a function exist

requires/provides在一个对象中是结合在一起的，这样就形成了它们的父类。

```
class top : public sc_module
{
   public:
   top(sc_module_name nm) : sc_module(nm)
   {
      r.port(p);
   };

   private:
      requires r;
      provides p;
};
```

Bind the *requires* port to the *provides* object.

requires/provides范例是AVM的基础。后面我们将看到在SystemVerilog中，它是如何在事务交易级将对象连接在一起的。

总结

当你可以把验证架构的构件采用许多小的，带有定义好的接口的的组件来实现时，构建复杂验证架构的问题就能简单很多。当你使得这些组件可以再复用时，这个问题就能更进一步简单化。AVM通过使用标准TLM接口和面向对象的编程技术来支持这些组件的构建。

第 4 章 TLM 介绍

设计一个电子系统的过程包括抽象概念，并用具体的详细的资料（它们可以制作在硅片上）来取代抽象的概念。有些抽象的概念已经被仔细的定义和编码并成为设计实施的方法。RTL是一个非常常见的用于创建设计的抽象概念。有许多基于RTL的工具，使得RTL成为引发设计和验证过程的一个方便途径。随着设计变得越来越大，越来越复杂，使用高级抽象（具有更少细节的方法）就变得更为方便。事务级建模（TLM）作为一种创建一个设计的最初的具体的结构（它可以被仿真和分析）的方式也变得越来越受欢迎。

RTL模块在管脚级互相通讯。通过wire传送字节。组件之间通讯的媒介就是net。Net有定义好的语言：它在任何一次都能保留单一比特位的值，当一个驱动组件要改变时，这个值也随之改变。当net改变数值时，由net驱动的所有组件必须重新赋值，以便在它们的一个输入中反应这个新的变化。

事务级模块由互相通讯的许多程序构成，它们通过通道来交互事务信息。

在Grötker et. al.编的书《System Design with SystemC》中，讨论了计算模块。他们定义计算模型有3个构件：

- 一个时序模型
- 在同时发生的程序件通讯的方法
- 激活程序的规则

RTL模块有一个离散事件模型，而程序之间的通讯则通过net完成，当程序的一个输入改变数值时，程序就被激活。

事务级模块可以计时也可以不计时，通过通道在程序间通讯。其程序间的通讯是通过互相发送事务信息完成的，而不是发送单个的比特位。

事务级模块包括一系列具有不同时间的计算模型，通讯和程序激活模型。然而，在每一种情况下，通讯的内容与各自的比特位相比，具有更高抽象级别。在各个进程之间进行交互的就是事务。在最普遍的情况下，你可以很容易地将单个比特位的转移做成事务信息。因此，尽管TLM是RTL模块（所有的RTL模块也是事务级模块）的一个固有扩展集，在讨论事务信息时，我们还是仅限于比比特位更高级别地抽象。

事务的定义

为了讨论事务级模块，首先要定义事务

事务就是在一个设计中受到时间限制的活跃的量子。

这是事务最一般的定义。它是说，在一个设计（或模块，或一个设计的子系统）中发生在两个时间点之间的所有事件都可以称作事务。尽管这个定义是精确的和普遍的，但并不是很有用。一个更有用的定义是：

事务是在两个实体之间单向转移控制或数据。

这就是事务的面向硬件的概念。当看硬件时，你可以很容易的区别转移控制或数据的实体。在一个基于总线的设计中，在总线上进行读写可以认为是事务。当你在看一个基于总线的设计或通讯设计时，也很容易辨别在两个实体之间，哪个是转移控制的操作，哪个是转移数据的操作。

一个事务就是一个函数调用。

这是事务的面向软件的概念。在事务级模块中，通过函数调用开始动作。函数调用包括参数，它们有被送到被调用函数中的参数，也有函数的返回值，包括从被调用的函数中返回的数据。被调用的函数可能被堵塞从而引起时间终止（在计时系统），或者它立即返回。

表示事务

为了构建事务级模块，需要表示事务以便进行操作。有三种方式表示事务：

1 函数调用

2 事务对象

3 事务记录

每一个都很好，在不同的场合，每一个都是最合适的。

函数调用

函数调用是用来在事务中发起动作。简单的说，函数调用是一个事务或一个事务就是一个函数调用。例如，考虑一个脉冲读入操作的代码片断。

```
void burst_read(addr_t base_addr, data_t *data, int length)
{
   for(int i=0; i<length; i++)
   {
      data[i] = bus_read(base_addr + i);
   }
}
```

函数burst_read表示一个事务，它是单独读入事务的有序集合。在两种情况下，即burst_read 和read， 函数发起一些动作从而暗自完成。不管是在多低的级别操作，它们都执行集合里的read函数，它们形成一个读入事务。

事务对象

本质上，事务级模块是关于模块通讯的程序。事务级程序之间相互传送的项目就是事务。以burst_read为例，程序执行代码片断，它引起的读入不一定是保存已经读入的存储器的程序。参见图4-1看系统组织情况。，

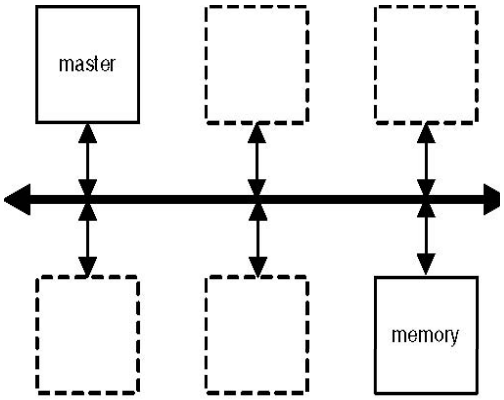


图4-1 基于总线的系统

许多主机和从机都连接在一个单一总线上。主机引起脉冲读入，存储从机则发生响应。如果主机和存储器从机是独立的程序模块，就需要采取一定的方式组织读入请求和存储响应。可以采用如下结构：

```
class request
{
   addr_t address;
   data_t data;
   bus_op_t op;
};

class response
{
   addr_t address;
   data_t data;
   bus_op_t op;
   status_t status;
};
```

请求和响应对象可以认为是事务，因为它们是总线主机和存储器通讯进行的媒介。当总线主机要从存储器中读入数据是，它就发送事务信息给存储器从机。储器从机返回响应。

现在，上面的burst_read采用的bus_read函数可以进行如下操作：

```
data_t bus_read(addr_t addr)
{
   request req;
   response rsp;

   req.address = addr;
   req.op = OP_READ;
   bus.put(req);

   rsp = bus.get();
   return rsp.data;
}
```

事务记录

有时需要进行详细的RTL设计，使得它看起来像一个TLM。有时你需要观察它在事务级的行为，原因在于：

- 在抽象的更高级别观察其行为可以更好的理解设计，而不仅仅是解码。
- 用对应的事务级组件与RTL设计进行操作上的比较。
- 为了计算函数指针，将其作为事务信息，即计算功能覆盖率。

用于SystemC的验证库SCV提供了事务记录设施。在没有深入了解其工作原理的时候，其本质的观点就是标记一条事务信息的起点和终点。当控制的轨迹到达起点，就说事务开始。当控制的轨迹到达终点，就说事务结束。事务都被写入信息流中，这些信息流都大体等价于波形轨迹。事务按照时间顺序输入信息流，正如按时间顺序存入线上的变化以形成波形轨迹一样。

下面是一个基本例子：

```
void thread()
{
   scv_tr_stream stream("stream");
   scv_tr_generator<data_t> tr_gen("read",stream","data");

   while(1)
   {
      wait(synch_event);
      tr_gen.begin_transaction();
      // ... do stuff
      tr.gen.end_transaction();

   }
}
```

每次循环迭代，就记录一个新的事务。循环的开始是调用函数begin_transaction()，它记录一个新事务的开始。循环的结尾是调用函数end_transaction,它结束已经开始的函数begin_transaction()的事务从而而完成函数。在调用函数begin_transaction()和end_transaction之间的所有行为组成了一个事务。

这就给SCV事务记录设施能做什么提供了线索。这里只是给出了一个基本的例子来说明事务作为一个被记录的实体的概念。采用SCV，你可以将按照各自的线程创建的事务信息联系起来放在一起。对于高级操作，详见SCV文件。

事务级模块和验证

事务级模块和功能验证并不常放在一起讨论。下面给出应用事务级模块技术来构建功能验证的测试有许多的有点。

任何在仿真器上运行的代码就是模型。我们的目的是讨论参考模型和实现模型。实现模型是准备用于制作的设计模型。目前，实现模型通常是综合的RTL代码。参考模型是在更高的级别表示实现，省略了许多细节，但保留了基本功能。

简介

参考模型比实现模型更为抽象。创建抽象模型就是抽象时间，数据和函数。

抽象时间：仿真器中的时间抽象就是指真个设计状态有多少是一致的。在事件中运行的模型通过离散的事件概念驱动仿真器（如逻辑仿真器），对在特定的时间点发生的事件求平均。事件通常（尽管不是常常）引起调用某种程序。

在一个仿真中发生的事件越多，需要调用的程序就越多，仿真运行得就越慢。抽象时间就是减少设计必须一致的点的数量，因此，必须激活所有事件和程序。例如，在一个RTL模型中，每次变化后每个网络必须一致。在循环精确抽象中，只在时钟边缘的设计必须一致，消除发生在时钟边缘之间的所有事件。在事务级模型中，每个事务结束的设计状态必须一致，每个都可能消耗许多时钟周期。

抽象数据：数据指组件之间通讯的对象。在RTL模型中，数据指通过组件之间的net传送的单个比特位。在TLM中，数据存在于事务信息以及包含原理判断集合的不同类的结构的形式中。

例如，考虑一台通讯设施的信息包。详情的最低级别，这个信息包包括起始和结束比特位，包头，错误校正信息，有效载荷大小，有效载荷和包尾。在一个抽象模型中，可能只有有效载荷和大小是必须的，其它数据对于已经执行的计算则不必要。

抽象功能： 一个模型的函数是一系列它必须做的事情。抽象功能就是要减少这个系列或用更简单的计算代替这些事情。例如，在一个ALU中，对一个加减乘法器，你可以选择本征的乘法运算操作（如*）应用到你的建模语言中，而不用对所有运算用移位加的算法来进行编码。后者就是实现的一部分，但在更高的级别，移位加算法的详情就不再重要了。

抽象时间，数据和函数的特殊方式就成为抽象级。RTL就是一个抽象级，它的功能比门级模型或晶体管级模型要少的多。为了进行功能验证，RTL是我们需要考虑的最低级别的抽象。因为综合器能非常有效地将RTL转换为门级，我们就不需要考虑更低级别地详情。此外，更低级进入电子领域地在设计功能之外的事件。

参考模型

验证工程师和设计工程师建立了参考模型，也叫做黄金模型，该模型在C 或C++中用了很长时间。为了验证他们的设计，他们用某种方式将黄金模型与DUT连接起来来比较两者的输出。他们这样做的原因就在于参考模型具体化了设计意图。它们包含的基本功能－运算法则，数据结构和程序－这使得驱动程序在不实现细节时就可工作。将执行结果与参考比较是功能验证的根本任务。

参考模型是比RTL实现更高级的抽象。原因有许多。参考模型必须要可以构建成比执行快得多和成本低得多。如果它就像DUT一样在相同的抽象级，在设计中就没有必要比较DUT和参考模型了。低级抽象的参考是没有用的，因为你已经建立了RTL模型以便进入门或晶体管。这就要求我们用事务级来表示黄金模型。事务级是高于RTL的所有事件，可以是时序精确的，计时的或不计时的，使用抽象数据或字节精确的寄存器。

用事务级技术时构建参考模型的一个好方式，也是AVM提倡的方式。事务级模型是更高级的抽象，所以它们可以快速构建，快速运行。另外，使用定义好的TLM接口和通道使得构建与其它TLM相互作用的模型更为容易。本章后面

将介绍基本的TLM概念，以作为构建基于AVM验证组件的基础。

输入

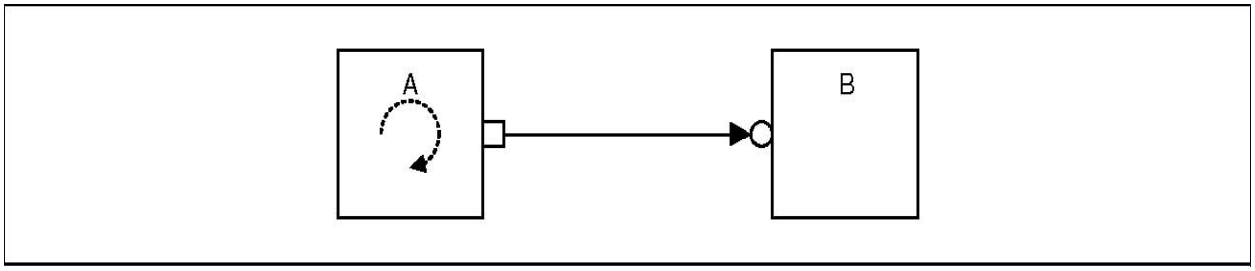


图4-2 TLM输入图

说明

方框A将事务信息输入方框B。方框A是一个有自己线程的自由运行的程序。方框B是一个从机，它没有自己的线程，只有被方框A调用时才运行。A发起事务信息，B接收并事务。

主要概念

- 一条事务信息是一串数据，它表示系统中的一些行为。数据可以是任何类型，但通常包括一个驱使组件执行动作的命令或已经完成的执行结果。
- 一个输入结构将事务信息从激励器传送到目标。激励器调用函数put()，该函数调用输入的执行结果放到目标中。数据则通过参数传送到put()中。
- 这里的输入操作是模块化操作。目标B模块是说，当A发出输入信号，它必须等待，直到B完成这个操作。B可以立即返回否则会消耗时间。
- put操作基本不需要时间，整个操作是通过函数调用来实现，不需要消耗时间。
- 本例就是说明什么是常被称为生产者/消费者对的。左侧的激励器产生一个对象，它被右侧的目标消耗。

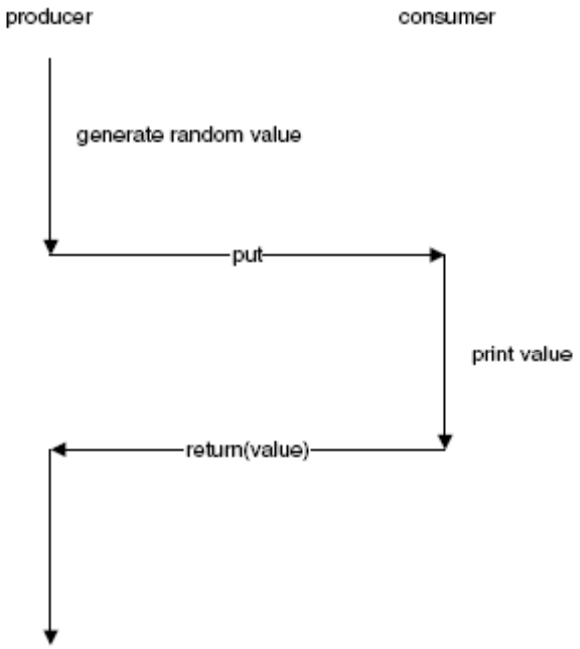


图4-3 模块化输入操作的控制流程图

就如上面的流程图所说明的，一个put包括两个程序，一个产生程序，一个消耗程序。本例中的产生程序是一个很简单的程序，它只是产生一个随机值。put将产生的值发送给消耗程序。消耗程序阻塞，阻塞表示函数调用器等待到被调用函数完成。在一个计时系统中，如硬件仿真器，消耗程序可能会引起时间消逝。当消耗程序结束并返回时，产生起就会从停止的地方拾起，只是时间会提高。

SystemVerilog 实现

首先来看一下产生器。

```
34    class producer;
35
36      put_if put_port;
37
38      task run;
39
40        int randval;
41
42        for(int i=0; i<10; i++)
43          begin
44            randval = $random %100;
45            $display("producer: sending   %4d", randval);
46            put_port.put(randval);
47          end
48
49      endtask
50
51    endclass : producer
file: topics/04_tlm/01_put_svc/put.sv
```

发生器在0-99范围内产生10个随机整数，每次它产生一个新的随机值，它就调用chan.put()发送该值给目标。

put_if类将producer和consumer连结在一起。它给名为put的任务提供必要条件，该任务有一个必须由消耗器提供的整型输入。

```
26    class put_if;
27      virtual task put(int val);
28      endtask
29    endclass : put_if
```

目标包括put任务的实现，在本例中，消耗一个值只是简单的显示该值以便确认该值是正确的来自于发生器。

```
56    class consumer extends put_if;
57      task put(int val);
58        $display("consumer: receiving %4d", val);
59      endtask : put
60    endclass : consumer
file: topics/04_tlm/01_put_svc/put.sv
```

发生器控制整个操作，它由一个程序组成，该程序以整数的形式将事务信息发送给消耗器。为此，产生器调用消耗器中的一个函数称作put。Put函数的目的就是将数据从发生器的范围传送到consumer的范围。发生器以实际参数的形式给put函数提供数据。Put函数在consumer的范围内运行，因此一旦调用put()，其参数中的数据就传送到consumer的范围。

咋一看，这像一个小魔术－将数据从一个范围传送到另一个，而仅仅是通过调用函数进行。如果看得更深入，连接是如何形成的这一点就会变得更清晰。

顶层模块将发生器和消耗器连接起来。

```
65    module top;
66
67      producer p;
68      consumer c;
69
70      initial begin
71
72        // instantiate producer and consumer
73        p = new;
74        c = new;
75
76        // connect producer and consumer
77        // through the put_if interface class
78        p.put_port = c;
79        p.run;
80
81      end
82
83    endmodule : top
file: topics/04_tlm/01_put_svc/put.sv
```

在initial模块中，顶部通过调用一个新的操作符，创建发生器和消耗器的实例化。通过赋值完成这二者之间的连接。

```
78 p.put_port = c;
```

赋值是连接中的最后一阶段。第一阶段是接口类，它包括put函数的定义，该函数是用来在发生器和消耗器之间通讯的。这个定义是虚拟的，也就是说这个函数不在这个类中运行，而是必须在一个扩展类中运行。第二阶段就是消耗器来源于put_if类。这一阶段产生消耗器执行put函数的请求。第三阶段是判断发生器的一个端口。该端口的类型就是put_if类的类型。put_port是对象put_if的指针。最后一阶段，分配，将put_port指针的值分配给消耗器。因为消耗器来源于put_if，其类型适合于分配目的。发生器现在有一个指向消耗器接口put()的指针。消耗器除了put()外，可能还有许多其它的成员和方法，但是发生器只能访问put()，因为它的指针是指向消耗器的基类的。

SystemC 实现

SystemC的发生器极像SystemVerilog的发生器。两者都有一个在0-99之间产生10个随机整数的循环。SystemC变量在发生器中使用线程（SC_THREAD）。

这样组织有许多理由。其中之一就是一个方法（SC_METHOD)要求时钟或一些其它的信号来触发它。我们希望发生器在没有受到任何外界触发的情况下自由运行。另一个原因就是put()调用可能被堵塞。也就是说，消耗器可能导致时间消逝。线程可以阻塞或重新开始，方法则不能。

```
44    class producer : public sc_module
45    {
46     public:
47      producer(sc_module_name nm) :
48        sc_module(nm),
49        put_port("put_port")
50      {
51        SC_THREAD(run);
52      }
53      SC_HAS_PROCESS(producer);
54
55      sc_export<put_if> put_port;
56
57      void run()
58      {
59        int randval;
60
61        for(int i=0; i<10; i++)
62        {
63          randval = rand() % 100;
64          cout << "producer: putting " << randval << endl;
65          put_port->put(randval);
66        }
67      }
68    };
file: topics/04_tlm/01_put_sc/put.cc
```

消耗器也具有和SystemVerilog副本一样的结构。其初始成分是一个put()函数的执行。

```
78    class consumer : public sc_module, public put_if
79    {
80     public:
81      consumer(sc_module_name nm) :
82        sc_module(nm)
83      {}
84
85      void put(int val)
86      {
87        cout << "consumer: receiving " << val << endl;
88      }
89    };
file: topics/04_tlm/01_put_sc/put.cc
```

与消耗器相连的是put_if，一个纯虚拟类，它定义了接口，外部模块可以使用这个接口发送要消费的项目。

```
28    class put_if : public sc_interface
29    {
30     public:
31      virtual void put(int) = 0;
32    };
```

发生器和消耗器之间的连接，尽管在SystemC 和 SystemVerilog中的概念是相同的，但其机理却不一样。概念上，消耗器提供函数put()，该函数被发生器调用。发生器通过调用消耗器可利用的put()函数经过对象到消耗器。消耗器的输出则跳到consumer_if接口。

```
94    class top : public sc_module
95    {
96     public:
97      top(sc_module_name nm) :
98        sc_module(nm),
99        p("p"),
100       c("c")
101     {
102       p.put_port(c);
103     }
104
105     producer p;
106     consumer c;
107   };
file: topics/04_tlm/01_put_sc/put.cc
```

上面的顶层模块直接将发生器和消耗器相连，而不需通过通道。就如SystemVerilog中一样，消耗器通过一个对象调用输出put()执行。这里称作调出。顶层模块将发生器上的端口与消耗器的输出口相连，将发生器的put()调用与消耗器的put()执行绑在一起。

尽管在本例中没有用明确的通道，SystemC在许多应用领域广泛使用了通道。在TLM的下一节将探究通道是如何工作，以及为什么采用通道。

Get

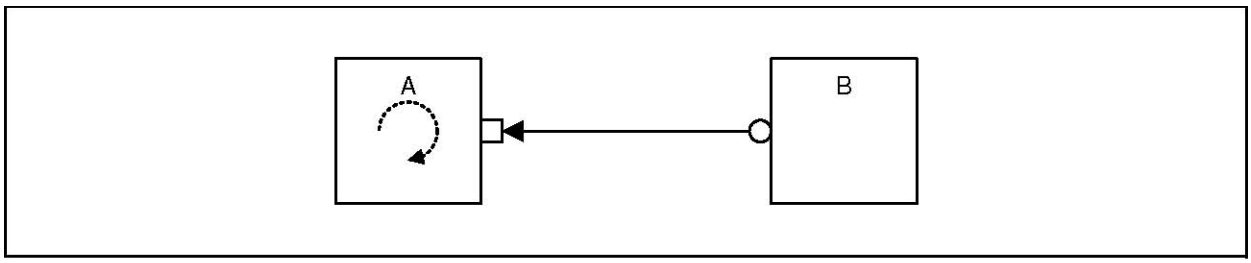


Figure 4-4. TLM get Configuration

说明

获取示意图是put示意图的补充。就如put示意图一样，get示意图有两块，一个激发器和一个目标。激发器从目标处获取事务信息。在本例中，激发器是消耗器，目标是发生器。激发器/消耗器请求目标/发生器产生新值。

主要概念

- 获取示意说明就是将事务信息从目标传送到激发器。
- 就如事务示意图一样，获取激发器接口是一个请求/提供接头。
- 这里的事务超作是模块操作。目标B模块，也就是说，当A发出获取时，它必须等待直到B完成这个操作。B应当立即返回获消耗时间

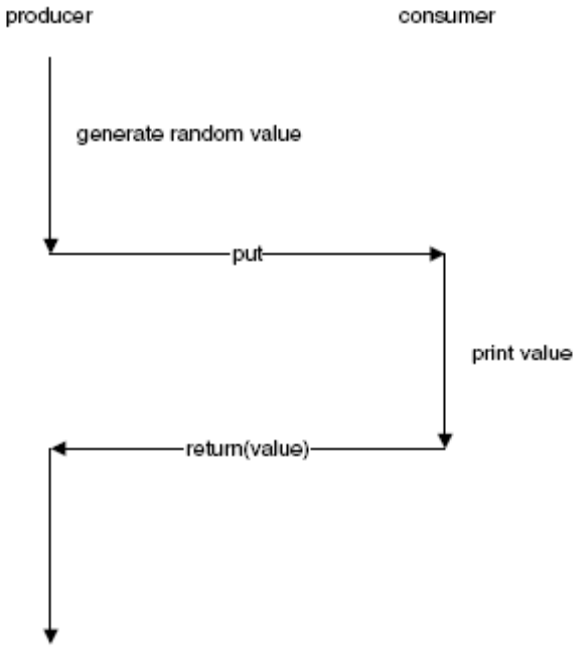


图4-5 模块获取操作的控制流程图

在获取操作过程中，消耗器负责，它告诉发生器产生新对象，并将它与发生器负责的放入操作进行对照。获取操作是阻滞，也就是说发生器在产生新值的过程中会引起时间消逝。发生器通汇返回get()函数的值将该值发送给消耗器。

SystemVerilog 实现

在获取操作中，因为消耗器包含了驱使本例操作的线程，就可以从消耗器开始。循环调用chan.get() 10次。每次调用chan.get请求发生器产生新值并将其返回。

```
50    class consumer;
51
52      get_if get_port;
53
54      task run;
55        int i;
56        int randval;
57        for(i=0; i<10; i++)
58         begin
59           randval = get_port.get();
60           $display("consumer: receiving %4d", randval);
61         end
62      endtask
63    endclass : consumer
file: topics/04_tlm/02_get_svc/get.sv
```

发生器执行获取函数。每次它被调用它就产生一个0-99之间的随机整数，并将该值返回给调用器。

```
36    class producer extends get_if;
37
38      function int get();
39        int randval;
40        randval = $random % 100;
41        $display("producer: sending   %4d", randval);
42        return randval;
43      endfunction
44
45    endclass : producer
file: topics/04_tlm/02_get_svc/get.sv
```

将发生器和消耗器连接起来的接口如放入例中一样的工作。get_if类有一个虚拟函数，该函数建立请求，要求发生器执行函数get()。

```
28    class get_if;
29      virtual function int get();
30      endfunction
31    endclass : get_if
file: topics/04_tlm/02_get_svc/get.sv
```

发生器和消耗器之间的连接也如put结构中一样。

```
68    module top;
69
70      producer p;
```

```
71      consumer c;
72
73      initial begin
74        // instantiate producer and consumer
75        p = new;
76        c = new;
77
78        // connect producer and consumer through the get_if
79        // interface class
80        c.get_port = p;
81        c.run;
82      end
83    endmodule : top
file: topics/04_tlm/02_get_svc/get.sv
```

SystemC 实现

SystemC消耗器的的结构和SystemVerilog消耗器的一样。它包括一个线程，该线程运行一个循环从发生器中获取10个值。

```
52    class consumer : public sc_module
53    {
54     public:
55      consumer(sc_module_name nm) :
56        sc_module(nm),
57        get_port("get_port")
58      {
59        SC_THREAD(run);
60      }
61      SC_HAS_PROCESS(consumer);
62
63      sc_export<get_if> get_port;
64
65      void run()
66      {
67        for(int i=0; i<10; i++)
68        {
69          int val = get_port->get();
70          cout << "consumer: receiving " << val << endl;
71        }
72      }
73    };
file: topics/04_tlm/02_get_sc/get.cc
```

发生器的结构也如SystemVerilog发生器的一样。把包括一个get()函数的执行。每次get()被调用时，就产生一个0-99之间的随机值并将该值返回给调用器（消耗器）。

```
34    class producer : public sc_module, public get_if
35    {
36     public:
37      producer(sc_module_name nm) :
38        sc_module(nm)
39      {}
40
41      int get()
42      {
43        int randval = rand() % 100;
44        cout << "producer: sending " << randval << endl;
45        return randval;
46      }
47    };
file: topics/04_tlm/02_get_sc/get.cc
```

发生器通过producer_if接口使得可以在外部利用get()函数。

```
25    class get_if : public sc_interface

26    {
27     public:
28      virtual int get() = 0;
29    };
file: topics/04_tlm/02_get_sc/get.cc
```

顶部模块将发生器和消耗器绑在一起。它用发生器接口get_if连接消耗器输出口，即get_port。

```
78    class top : public sc_module
79    {
80     public:
81      top(sc_module_name nm) :
82        sc_module(nm),
83        p("p"),
84        c("c")
85      {
86        c.get_port(p);
87      }
88
89      producer p;
90      consumer c;
91    };
file: topics/04_tlm/02_get_sc/get.cc
```

请求/响应

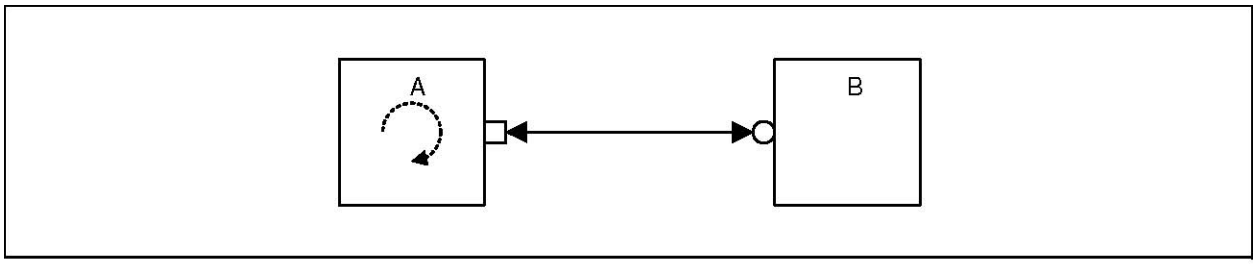


Figure 4-6. TLM request/response Configuration

说明

请求/响应结构本质上是结合了获取和放入结构。模块A在自己的线程下操作，将请求发送给B。B（从机），接受请求并发出响应。A阻塞直到B完成其操作并返回一个响应。

主要概念

- 请求/响应结构将请求事务信息从激发器传送到目标，并将响应事务信息从目标返回给激发器。
- Put通过函数判断给目标发送一个值。Get收到一个值并通过函数返回值反给激发器。请求/响应结构用传送函数来做这两件事。

```
put(val)
val = get()
response = transport(request)
```

- 发生器和消耗器是用来描述一个单向通讯中的过程。在双向结构中，两个过程都产生值也消耗值。因此，请求/响应术语就不再起作用。请求/响应结构中所包含的程序叫做主机和从机。一个主机激发通讯，从机则响应之。
- 请求/响应操作是阻塞的。主机阻塞直到从机返回请求。从机可以消耗仿真时间。

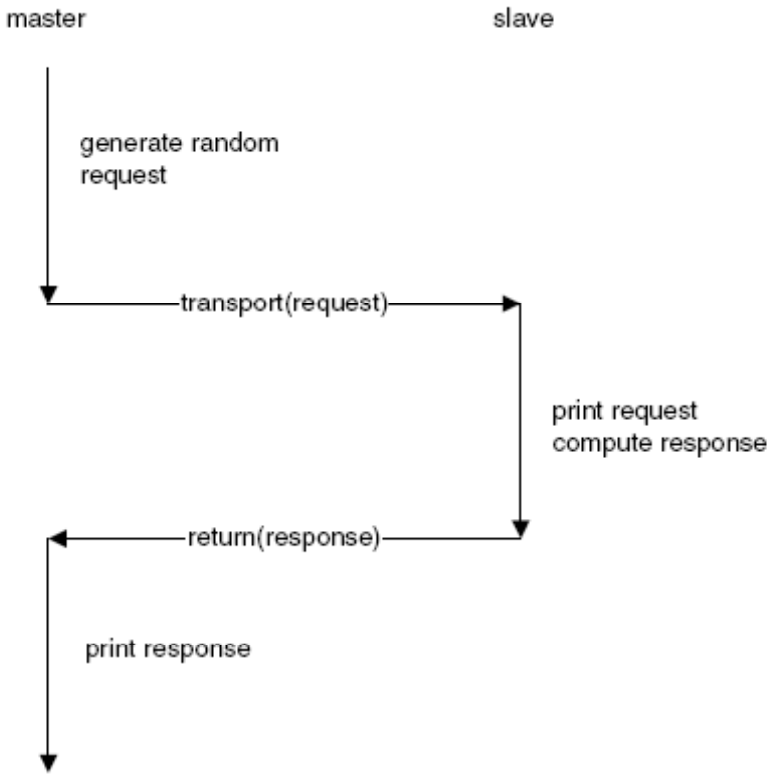


图4-7 传送操作控制流程图

就如put() 和 get()， transport()是一个阻塞函数。主机阻塞直到传送函数完成并返回响应。从机可以消耗仿真时间。

SystemVerilog 实现

毫不奇怪的是，请求/响应结构执行合并了put和get结构。它通过transport_if给从机发送请求。不像发生器一样，这里用主机来驱动所有操作。

```
31    class master;
32
33      transport_if port;
34
35      task run;
36
37        int request;
38        int response;
39
40        for(int i=0; i<10; i++)
41          begin
42            request = $random % 100;
43            $display("master: sending   request  %4d", request);
44            response = port.transport(request);
45            $display("master: receiving response %4d", response);
46          end
47
48      endtask
49    endclass : master
file: topics/04_tlm/03_req_rsp_svc/req_rsp.sv
```

transport()调用给从机发送请求并接收响应。从机获取请求，表示响应并将起返回。本例中的响应是请求的一个微小信号改变。

```
54    class slave extends transport_if;
55
56        function int transport(int request);
57            int response;
58            $display("slave:  receiving request  %4d", request);
59            response = -request;
60            $display("slave:  sending   response %4d", response);
61            return response;
62        endfunction
63
64    endclass : slave
file: topics/04_tlm/03_req_rsp_svc/req_rsp.sv
```

这个连接通道看起来也很熟悉。它规定主机命令（输入）一个transport()函数的执行，从机则提供（输出）该函数。

```
23    class transport_if;
24      virtual function int transport(int request);
25      endfunction
26    endclass : transport_if
file: topics/04_tlm/03_req_rsp_svc/req_rsp.sv
```

最后，顶层模块将所有的片断连接在一起。

```
69    module top;
70
71      master m;
72      slave s;
73
74      initial begin
75        // instantiate the master and slave
76        m = new;
77        s = new;
78
79        // connect the master and slave through
80        // the port interface
81        m.port = s;
82        m.run;
83      end
84
85    endmodule : top
file: topics/04_tlm/03_req_rsp_svc/req_rsp.sv
```

SystemC 实现

在本例的请求/响应结构中，主机产生有10个请求的一个系列。每个请求就是0–99之间的一个随机整数。

```
37 class master : public sc_module
38 {
39 public:
40 master(sc_module_name nm) :
41 sc_module(nm),
42 port("port")
43 {
44 SC_THREAD(run);
45 }
46 SC_HAS_PROCESS(master);
47
48 sc_export<transport_if> port;
49
50 void run()
51 {
52 int request;
53 int response;
54
55 for(int i=0; i<10; i++)
56 {
57 request = rand() % 100;
58 cout << "master: requesting " << request << endl;
```

```
59 response = port->transport(request);
60 cout << "master: receiving response " << response << endl;
61 }
62 }
63 };
```
file: topics/04_tlm/03_req_rsp_sc/req_rsp.cc

从机接收这个请求，计算响应，并将该响应返给主机。

```
68 class slave : public sc_module, public transport_if
69 {
70 public:
71 slave(sc_module_name nm) :
72 sc_module(nm)
73 {}
74
75 int transport(int request)
76 {
77 cout << "slave: receiving request " << request << endl;
78 int response = -request;
79 cout << "slave: responding with " << response << endl;
80 return response;
81 }
82 };
```
file: topics/04_tlm/03_req_rsp_sc/req_rsp.cc

纯虚类transport_if提供了主机用于绑定从机的接口。

```
28 class transport_if : public sc_interface
29 {
30 public:
31 virtual int transport(int) = 0;
32 };
```
file: topics/04_tlm/03_req_rsp_sc/req_rsp.cc

顶层模块将主机和从机连接起来。

```
87 class top : public sc_module
88 {
89 public:
90 top(sc_module_name nm) :
91 sc_module(nm),
92 m("m"),
93 s("s")
```

```
94 {
95 m.port(s);
96 }
97
98 master m;
99 slave s;
100 };
```

file: topics/04_tlm/03_req_rsp_sc/req_rsp.cc

FIFO

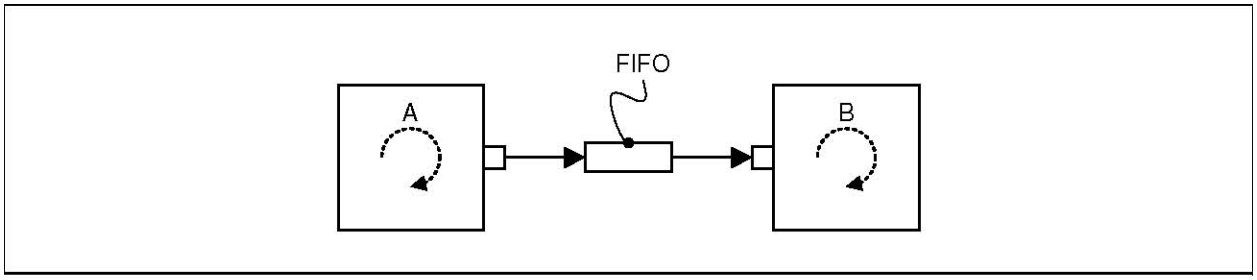


Figure 4-8. TLM FIFO Configuration

说明

模块A和模块B都在各自的线程下操作。激发器A发送事务信息给目标B。但是，并不是直接将事务信息发送给目标，激发器将它们放入fifo,目标则从fifo中获得它们。在激发器和目标之间插入fifo使得它们可以独立操作。

主要概念

fifo插入激发器和目标之间。Fifo节流这两个程序间的通讯。激发器将事务信息放入fifo，目标则从fifo获取事务信息。

A和B仍然使用阻塞操作，但其中一个等待另一个的操作被fifo消除了。阻塞以fifo的形式出现。如果当A在执行放入时fifo满了，A将阻塞直到fifo中有放置另一条事务信息的空间。同样地，当B执行获取时fifio空了，B阻塞直到有事务信息出现在fifo中。

在通讯组件之间采用fifo，可以使得可以独立的构建和测试这些组件，也可以使得它们独立的操作。每个组件只需要知道它发送或接收的事务目标的类型，没有必要知道它将通讯的是哪个特定的组件。

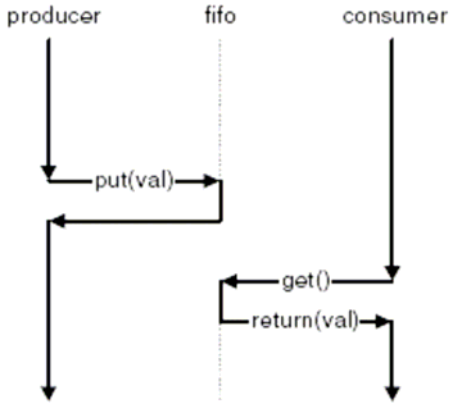


图4-9 带fifo的发生器/消耗器的控制流程图

发生器使用我们在本章第1结讨论的同样的放入接口。发生器调用put()。它将事务信息堆栈在fifo中，而不是直接将事务信息发送给消耗器。一段时间以后，消耗器调用get从fifo中找回事务信息。Fifo将发生器和消耗器独立起来，以便它们互相之间不再直接依赖。

SystemVerilog 实现

发生器和消耗器的执行都是直接的。发生器执行put的一个序列，每次将一个项目堆栈到fifo中。消耗器则通过调用get从fifo中找回事件。

```
25 class producer extends avm_verification_component;
26
27 tlm_blocking_put_if#(int) put_port;
28
29 task run;
30
31 int randval;
32
33 for(int i = 0; i < 10; i++)
34 begin
35 randval = $random % 100;
36 $display("producer: sending %4d", randval);
37 put_port.put(randval);
38 end
39 endtask
40
41 endclass : producer
```

file: topics/04_tlm/04_fifo_svc/fifo.sv

在前面的发生器和消耗器的例子里，它们称为chan.put()和chan.get()。在本例中，则叫做fifo.put()和fifo.get()。Fifo是在发生器和消耗器之间的通讯通道。放入和获取都是在fifo上的操作而不是在发送或接收组件上的操作。

```
46 class consumer extends avm_verification_component;
47
48 tlm_blocking_get_if#(int) get_port;
49
50 task run;
51
52 int val;
53
54 forever
55 begin
56   get_port.get(val);
57    $display("consumer: receiving %4d", val);
58 end
59
60 endtask
61
62 endclass : consumer
```

file: topics/04_tlm/04_fifo_svc/fifo.sv

tlm_fifo(#int)是指向一个参数化类的指针，这个类停留在AVM库中。端口的方向则按照输入发生器和消耗器来规定。其理由是指向类对象的指针在仿真运行前是不知道的。输入例化时允许在运行时输入新值。

顶层模块将发生器和消耗器绑在一起，将它们通过fifo通道连接方式就如两个RTL组件被net连接的方式一样。

调用新的操作符来创建一个fifo例化。变量fifo是新产生的tlm_fifo的指针。该指针通过发生器和消耗器的端口列表。因为这两个组件上的端口都是输入端口，组件会接收指针放到新的fifo通道。

```
67 module top;
68
69 producer p;
70 consumer c;
71 tlm_fifo#(int) f;
72
73 initial
74 begin
```

```
75 // instantiate the producer, consumer,
76 // and the fifo channel
77 p = new;
78 c = new;
79 f = new;
80
81 // connect the producer and consumer
82 // through the fifo channel
83 p.put_port = f.blocking_put_export;
84 c.get_port = f.blocking_get_export;
85
86 // kick off the run processes in each
87 // verification component
88 avm_verification_component::run_all();
89 end
90
91 endmodule : top
```

file: topics/04_tlm/04_fifo_svc/fifo.sv

SystemC 实现

发生器用一个端口与fifo通道相连。这个端口就建立必要条件，与之相连的通道必须满足这个条件。这个条件被封装在我们用来指定端口类型的接口中。在本例中，这个接口就是tlm_put_if<int>。角括号里的类型int就是由发生器发出的对象的类型。

```
30 class producer : public sc_module
31 {
32 public:
33 producer(sc_module_name nm) :
34 sc_module(nm),
35 put_port("put_port")
36 {
37 SC_THREAD(run);
38 }
39 SC_HAS_PROCESS(producer);
40
41 sc_port<tlm_put_if<int> > put_port;
42
43 void run()
44 {
45 int randval;
46
47 for(int i = 0; i < 10; i++)
48 {
49 randval = rand() % 100;
50 cout << "producer: sending " << randval << endl;
51 put_port->put(randval);
52 }
53 }
54 };
```

file: topics/04_tlm/04_fifo_sc/fifo.cc

消耗器也有一个端口与fifo通道相连。消耗器端口使用的接口是59 class consumer : public sc_module

```
60 {
61 public:
62 consumer(sc_module_name nm) :
63 sc_module(nm)
```

```
64 {
65 SC_THREAD(run);
66 }
67 SC_HAS_PROCESS(consumer);
68
69 sc_port<tlm_get_if<int> > get_port;
70
71 void run()
72 {
73 int val;
74 while(1)
75 {
76 val = get_port->get();
77 cout << "consumer: receiving " << val << endl;
78 }
79 }
80 };
```

file: topics/04_tlm/04_fifo_sc/fifo.cc

顶层模块包含fifo，通过它发生器和消耗器之间进行通讯。Fifo是一个模板对象，它的模板参数就是fifo将要储存的对象的类型。Fifo与发生器的输出端口和消耗器的输入端口相连。编译器进行类型检查以确保与fifo相连的端口与fifo类型的数据类型相匹配。

```
85 class top : public sc_module
86 {
87 public:
88 top(sc_module_name nm) :
89 sc_module(nm),
90 p("p"),
91 c("c")
92 {
93 p.put_port(fifo);
94 c.get_port(fifo);
95 }
96
97 producer p;
98 consumer c;
99
100 private:
101 tlm_fifo<int> fifo;
102 };
```

file: topics/04_tlm/04_fifo_sc/fifo.cc

双向通讯

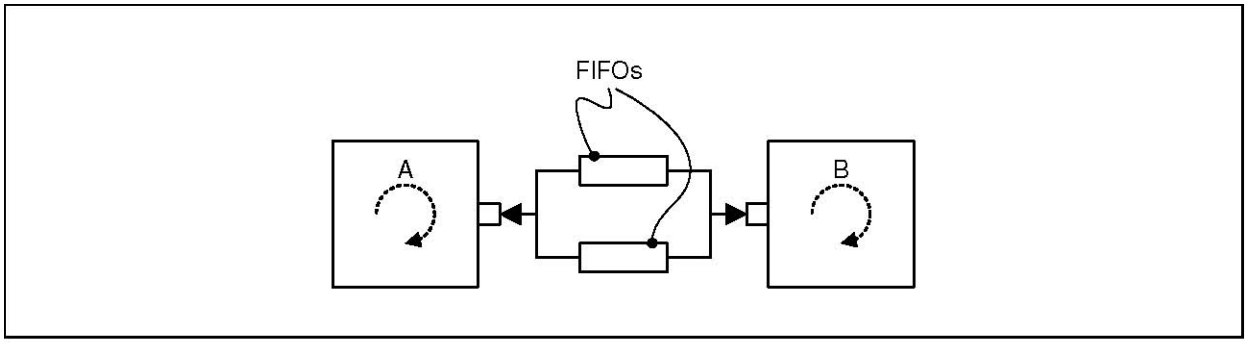


Figure 4-10. Bi-directional Communication

说明

就如单向通讯一样，经常需要双向通讯，这里通讯组件之间是相互独立的。这个结构说明了独立的请求和响应。激发器A发送请求给B，B返回响应。这里不像请求/响应配置，A直到请求被返回时才阻塞。

主要概念

一对fifo被插入两个通讯组件之间来节流请求和响应信息流。

有了fifo结构，A和B仍然使用阻塞操作，但是是以fifo的形式阻塞，而不是其它的组件。当发出一个请求时，如果fifo是满的，A会阻塞直到fifo中有放置另一条事务信息的空间。同样的，当B获取数据时，如果请求序列是空的B阻塞。响应通讯从目标B返回给激发器A也是以同样的方式工作的。

在组件之间使用fifo作为双向通讯，它们互相之间就不需要知道。唯一需要考虑的是发送和接收请求和响应。

因为请求和响应是独立发送的，就引起了如何匹配特定的请求与响应的问题。请求是否必要和如何去做完全由应用软件来决定。

一个双向通讯接口需要4个接头，两个给主机，两个给从机。

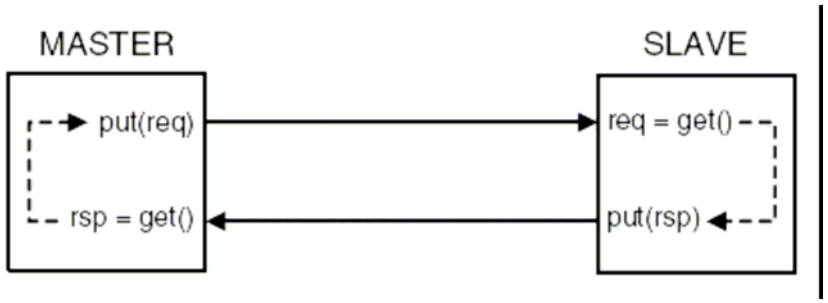


图4-11 双向通讯的操作顺序

双向通讯中的事件发生顺序如下：主机通过使用put()发送请求给从机发起通讯。从机用get()找回请求。从机发送请求并表示响应。一旦形成响应，从机也用put()将响应返回给主机。最后，响应被主机用get()找回。

SystemVerilog 实现

环境类包括通道并组成四个接头从而可以进行双向通讯。

```
104 class bidir_env extends avm_env;
105
106 master m;
107 slave s;
108 tlm_req_rsp_channel #( int ) req_rsp;
109
110 function new;
111 m = new("master");
112 s = new("slave");
113 req_rsp = new("req_rsp_channel");
114 endfunction
115
116 function void connect;
117 m.req_port = req_rsp.blocking_put_request_export;
118 m.rsp_port = req_rsp.blocking_get_response_export;
119 s.req_port = req_rsp.blocking_get_request_export;
120 s.rsp_port = req_rsp.blocking_put_response_export;
121 endfunction
122
123 task execute;
124 #10;
125 endtask
126
127 endclass // bidir_env
```

函数new()例化了请求/响应通道。函数将主机和从机上的请求和响应端口连接到请求/响应通道。

主机有两个端口，req_port和rsp_port。两个端口都是接口的指针。在环境类中，指针的值有函数connect()赋给。

```
24 class master extends avm_verification_component;
25
26 tlm_blocking_put_if #( int ) req_port;
27 tlm_blocking_get_if #( int ) rsp_port;
28
```

```
29 function new( string name , avm_named_component parent = null );
30       super.new( name , parent );
31 endfunction // new
32
33 task run;
34     fork
35       request_process;
36       response_process;
37     join
38 endtask
39
40 task request_process;
41
42 string request_str;
43
44 for( int i = 0; i < 10; i++ ) begin
45     $sformat( request_str , "%d" , i );
46     avm_report_message("sending request" , request_str );
47      req_port.put( i );
48 end
49
50 endtask // request_process
51
52 task response_process;
53
54 int response;
55 string response_str;
56
57 forever begin
58     rsp_port.get( response );
59     $sformat( response_str , "%d" , response );
60      avm_report_message("recieving response" ,
response_str );
61     end
62 endtask
63
64 endclass // master
```

运行任务fork两个程序，一个是发送请求，另一个是事务响应。在这个简单的实例中，请求和响应都是整数。如果请求和响应是更为复杂的数据包，这个组合就会是相同的。请求程序发送整数请求信息流。响应管理器找回响应

并打印它。

从机组织成单一的程序，它找回请求，事务请求并表示响应，然后将响应返回给主机。

```
70 class slave extends avm_verification_component;
71
72 tlm_blocking_get_if #( int ) req_port;
73 tlm_blocking_put_if #( int ) rsp_port;
74
75 function new( string name , avm_named_component parent = null );
76 super.new( name , parent );
77 endfunction // new
78
79 task run;
80
81 int request , response;
82 string request_str , response_str;
83
84 forever begin
85 req_port.get( request );
86 $sformat( request_str , "%d" , request );
87 avm_report_message("recieving request" , request_str
);
88
89 response = request;
90
91 $sformat( response_str , "%d" , response);
92 avm_report_message("sending response" , response_str );
93
94 rsp_port.put( response );
95
96 end // forever begin
97 endtask
98
99 endclass
```

SystemC 实现

主机有两个程序：一个用于发送请求，另一个用于异步接收响应。相应程序与前面的例子中的单向发生器非常类似。

```
46 void request_process()
47 {
48 int request;
49
50 for(int i=0; i<10; i++)
51   {
52      request = i;
53      cout << "master: sending request " << request << endl;
54      req_port->put(request);
55      wait(SC_ZERO_TIME);
56    }
57 }
```

file: topics/04_tlm/05_bidir_sc/bidir.cc

response程序用一个阻塞get操作来找回响应。因为它阻塞，它要等到一个响应出现后才能做任何事件。

```
59 void response_process()
60 {
61 int response;
62
63 while(1)
64 {
65 response = rsp_port->get();
66 cout << "master: receiving response " << response << endl;
67 wait (SC_ZERO_TIME);
68 }
69 }
```

file: topics/04_tlm/05_bidir_sc/bidir.cc

从机只有一个程序，该程序接受请求并返回响应。

```
90 void run()
91 {
92  int request;
93  int response;
94
95  while(1)
```

```
96 {
97     request = req_port->get();
98     cout << "slave: receiving request " << request << endl;
99     response = request;
100        cout << "slave: sending response " << response << endl;
101     rsp_port->put(response);
102    wait(SC_ZERO_TIME);
103    }
104 }
```

顶层模块表明了每一样是如何连接在一起的。req_rsp_channle<int>内有两个fifo，一个请求fifo，一个响应fifo。请求将事件放入请求fifo并从响应fifo中获取事件。从机则相反，它是获取请求放入响应。

```
110 class top : public sc_module
111 {
112  public:
113  top(sc_module_name nm) :
114  sc_module(nm),
115  m("m"),
116  s("s"),
117 req_rsp("req_rsp")
118 {
119         m.req_port(req_rsp.put_request_export);
120         m.rsp_port(req_rsp.get_response_export);
121         s.req_port(req_rsp.get_request_export);
122         s.rsp_port(req_rsp.put_response_export);
123  }
124
125  master m;
126  slave s;
127
128  private:
129    tlm_req_rsp_channel<int, int> req_rsp;
130  };
file: topics/04_tlm/05_bidir_sc/bidir.cc
```

我们只是简单的用了两个fifo，一个用于请求，一个用于响应。req_rsp_channel是一个方便的给两个fifo配对的通道，并提出正确的请求和响应接口。它有助于避免犯低级错误，如连接从机，因此它从响应fifo中获取并放入请求fifo中（这一点正好相反）。

事务级总线

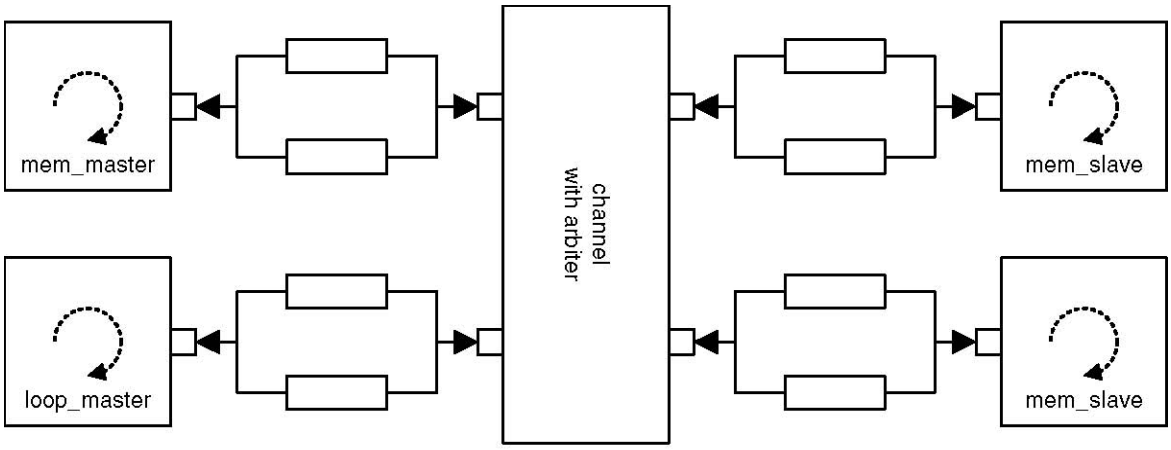


Figure 4-12. Transaction Bus with Four Masters/Slaves

说明

现在就该使用已经学会的单向和双向事务级通讯了。本例是一个有2个主机2个从机的事务级总线。loop_master既是主机又是从机。它在自己的地址空间产生事务信息。mem_master产生存储器从2个存储器从机中读出或写入。尽管这个简单的总线模块远不如真正的系统，但到目前所看到的例子为止，它却要现实的多。

通道是一个非常简单的总线模块。它包括一个寄存器分配图，该图画出了与总线相连的特定驱动程序的地址。它用循环判断程序在请求中判断。它不断地以一定的方式检查请求fifo。当出现请求时，它将请求传送到能为它服务的驱动程序，然后再到下一个请求fifo。

主要概念

- 独立操作的驱动程序通过总线相互通讯。
- 所有通讯都在事务级上进行。
- 前面实例中讨论的相同的双向接口和通道这里也可以用。本例中许多的驱动程序通过总线进行通讯，而不是两个驱动程序之间的点对点通讯。

SystemC 实现

该模块包括一个总线和一套主/从机。总线负责地址分配，这是通过从每个主/从机获取地址分配图入口构建起来的。主机有可以驱使系统操作的线程。当最后一个主机的线程终止时，整个系统就终止了。

主机和从机

与总线相连的每一个驱动程序都具有相同的结构。这是通过将它们都从同一个基本类中导出而得到的。

```
38 class master_slave_base : public addr_config_if
39 {
40 public:
41 master_slave_base();
42
43 sc_port<tlm_master_if<request,response> > master_port;
44 sc_port<tlm_slave_if<request,response> > slave_port;
45
46 addr_map_entry request_addr_space() { return entry; }
47 void set_addr_space(bus_addr_t low, unsigned size);
48
49 private:
50 virtual void master_thread() = 0;
51 virtual void slave_thread() = 0;
52
53 protected:
54 addr_map_entry entry;
55
56 };
file: topics/04_tlm/06_tlm_bus_sc/ms.h
```

master_slave_base类提供所有与我们的总线相连的主机和从机所需要的框架。它有一个主机端口和一个从机端口。它也指定了两个虚拟函数，master_thread和slave_thread。最后，它有一个地址分配图入口。

主机和从机的端口是以请求和响应对象的形式定义的。tlm_req_rsp_channel输出tlm_master_if和tlm_slave_if，由2个fifo组成，一个用于请求一个用于响应。主机端口与通道相连因此它能放入请求获取响应。从机端口则以相反的方式

相连，所以它获取请求放入响应。每个主/从机都有2个req_rsp_channels，这在总线模块中可以看到。每个独特驱动程序都可以用作主机，发送请求找回响应，也可以作为从机，接收请求，履行请求并返回响应。或者一个单独的驱动程序可以在两种方式下工作。

回路的主机是一个驱动程序，它的唯一目的就是在总线上产生一定的交通（来说明主机和从机的构造）。主机线程包括一个循环，该循环将请求发送到自己的地址空间然后等待响应。因为这个请求是被发送到自己的地址空间，所以可以用同一个器件服务，换句话说，它给自己发请求。

```
28 void
29 loopback::master_thread()
30 {
31 addr_map_entry entry = request_addr_space();
32 bus_addr_t low = entry.low_addr();
33 unsigned size = entry.size();
34
35 request req;
36 response rsp;
37
38 for(int i = 0; i < 256; i++)
39 {
40 bus_addr_t target_addr = (rand() % size) + low;
41 req.addr = target_addr & bus_addr_mask;
42 req.data = 0xffffL;
43 req.op = OP_READ;
44
45 cout << "loop master: " << name() << ": ";
46 cout << "sending : " << req << endl;
47 master_port->put(req);
48
49 // wait for response
50 rsp = master_port->get();
51 cout << "loop master: " << name() << ": ";
52 cout << "receiving: " << rsp << endl;
53 }
54 }
file: topics/04_tlm/06_tlm_bus_sc/loopback.cc
```

从机线程是对主机线程的补充。它等待响应，表示响应，在本例中是对产生

响应的请求成分的简单复制，并返回响应。

```
59 void
60 loopback::slave_thread()
61 {
62 request req;
63 response rsp;
64
65 while(1)
66 {
67 // get request
68 req = slave_port->get();
69 cout << "loop slave : " << name() << ": ";
70 cout << "receiving: " << req << endl;
71
72 // formulate response
73 rsp.addr = req.addr & bus_addr_mask;
74 rsp.data = req.data & data_mask;
75 rsp.op = req.op;
76 rsp.status = STATUS_SUCCESS;
77 cout << "loop slave : " << name() << ": ";
78 cout << "sending : " << rsp << endl;
79 slave_port->put(rsp);
80 }
81 }
file: topics/04_tlm/06_tlm_bus_sc/loopback.cc
```

注意到在发送和接收回路事务信息过程中，接口的4部分都在工作：

1 主机发送请求，它将请求放入master_if的请求fifo

2 从机接收请求。它从slave_if的请求fifo中获取请求。

3 从机发送响应。它将响应放入slave_if的响应fifo中。

4 主机接收响应。他从master_if的响应fifo中获取响应。

mem_master只有一个线程，即master_thread。判断模块满足虚拟函数slave_thread()必要条件的判据是通过提供一个空函数。

主机执行三个测试。第一个测试写入一系列随机产生的数值放入mem_slave(s)中。第二个测试读回这些数值。第三个测试是复制1.5倍mem_slave的内存空间到另一半。

写入测试产生一列总线写入事务信息。

```
36 req.op = OP_WRITE;
37 for(unsigned idx = 0; idx < 128; idx++)
38 {
39 req.addr = (start_addr + idx) & bus_addr_mask;
40 req.data = rand() & data_mask;
41
42 cout << "mem master : " << name() << ": ";
43 cout << "sending : " << req << endl;
44 master_port->put(req);
45
46 rsp = master_port->get();
47 cout << "mem master : " << name() << ": ";
48 cout << "receiving: " << rsp << endl;
49 }
file: topics/04_tlm/06_tlm_bus_sc/mem_master.cc
```

读入测试产生一序列总线读入事务信息。

```
52 req.op = OP_READ;
53 req.data = 0L;
54 for(unsigned idx = 0; idx < 128; idx++)
55 {
56 req.addr = (start_addr + idx) & bus_addr_mask;
57 master_port->put(req);
58
59 rsp = master_port->get();
60 }
file: topics/04_tlm/06_tlm_bus_sc/mem_master.cc
```

复制测试则将地址空间分成两半。它在低的一半空间读出一个字节，然后将它写入高的一半空间的相应位置。

```
63 for(unsigned idx = 0; idx < 64; idx++)
64 {
65 req.op = OP_READ;
66 req.addr = (start_addr + idx) & bus_addr_mask;
67 req.data = rand() & data_mask;
68
69 cout << "mem master : " << name() << ": ";
70 cout << "sending : " << req << endl;
71 master_port->put(req);
72
73 rsp = master_port->get();
```

```
74 cout << "mem master : " << name() << ": ";
75 cout << "receiving: " << rsp << endl;
76
77 req.op = OP_WRITE;
78 req.addr = (start_addr + idx + 64) & bus_addr_mask;
79 req.data = rsp.data & data_mask;
80
81 cout << "mem master : " << name() << ": ";
82 cout << "sending : " << req << endl;
83 master_port->put(req);
84
85 rsp = master_port->get();
86 cout << "mem master : " << name() << ": ";
87 cout << "receiving: " << rsp << endl;
88
89 }
```

file: topics/04_tlm/06_tlm_bus_sc/mem_master.cc

mem_slave是一个内存器件。它接受有成功状态信息的读/写请求和响应。对于读入，它也对来自内存的数据产生响应。

```
25 void
26 mem_slave::slave_thread()
27 {
28 request req;
29 response rsp;
30
31 bus_addr_t addr;
32
33 while(1)
34 {
35 // get request
36 req = slave_port->get();
37 cout << "mem slave : " << name() << ": ";
38 cout << "receiving: " << req << endl;
39
40 // adjust address to local memory space
41 addr = req.addr - entry.low_addr();
42
43 switch(req.op)
44 {
45 case OP_READ:
46 rsp.op = req.op;
47 rsp.addr = req.addr;
```

```
48 rsp.data = memory.read(addr);
49 rsp.status = STATUS_SUCCESS;
50 break;
51
52 case OP_WRITE:
53 rsp.op = req.op;
54 rsp.addr = req.addr;
55 rsp.data = req.data;
56 memory.write(addr, req.data);
57 rsp.status = STATUS_SUCCESS;
58 break;
59 };
60
61 cout << "mem slave : " << name() << ": ";
62 cout << "sending : " << rsp << endl;
63 slave_port->put(rsp);
64 }
65 }
file: topics/04_tlm/06_tlm_bus_sc/mem_slave.cc
```

内存从机的主体是一个回路，该回路不断的发出请求。实质上，这个回路是一个小型的命令解释程序，看请求对象的内容，然后进行适当的处理。从机有一个对象称作内存，它是一排数据单元所形成的可设定地址的内存。所有的请求最终都编程访问内存对象。

总线

总线是tlm_req_rsp_channels的集合，每个主机一个，每个从机也有一个，以及从通道到主机和从机本身的连接。它还有全部的地址分配图，将地址分配给总线单元数。总线的任务就是将请求分配给与总线相连的单元，将请求寄给这个单元，并将响应返回到请求程序。

```
43 class bus : public sc_module
44 {
45 public:
46
47 bus(sc_module_name nm);
48 SC_HAS_PROCESS(bus);
49
50 sc_export<tlm_master_if<request,response> >
master_export[bus_masters];
```

```
51 sc_export<tlm_slave_if<request,response> >
slave_export[bus_masters];
52 sc_port<addr_config_if, bus_masters> addr_map_port;
53
54 private:
55 // internal channels
56 addr_map amap;
57 tlm_req_rsp_channel<request,response>
master_channel[bus_masters];
58 tlm_req_rsp_channel<request,response> slave_channel[bus_masters];
59
60 private:
61 void build_addr_map();
62 void run();
63 };
file: topics/04_tlm/06_tlm_bus_sc/bus.h
```

与总线相连的主机数固定为一个常数。

```
38 const int bus_masters = 4;
```

file: topics/04_tlm/06_tlm_bus_sc/bus.h

总线模块本身的结构是一个 while(1)循环，该循环不断的事务请求。While循环中是一个for循环，该循环推选出每个主机/从机来看是否还有请求需要事务。这就形成了一个非常简单的判断机制。下面是一个主要循环。以下将它们分开以便进一步检查。

```
53 while(1)
54 {
55 masters_serviced = 0;
56 for(unsigned portidx = 0; portidx < bus_masters; portidx++)
57 {
58 wait(SC_ZERO_TIME);
59
60 // get the next request from a master. If there is
61 // nothing to get from this master then continue on to
62 // the next master.
63 bool ok_to_get = master_channel[portidx].get_request_export-
>nb_can_get();
64 if(!ok_to_get)
65 continue;
```

```
66
67 req = master_channel[portidx].get_request_export->get();
68 masters_serviced++;
69
70 int reqidx = amap.lookup(req.addr);
71 if(reqidx < 0)
72 continue;
73
74 slave_channel[reqidx].put_request_export->put(req);
75 rsp = slave_channel[reqidx].get_response_export->get();
76 master_channel[portidx].put_response_export->put(rsp);
77 }
78
79 // masters_serviced == 0 means there were no requests or
80 // responses to process. When that becomes the case
81 // then we're done.
82  if(masters_serviced == 0)
83  break;
84 }
file: topics/04_tlm/06_tlm_bus_sc/bus.cc
```

整数变量端口指当前正在运行的主机端口。对每一个主机，总线模块询问是否有请求。如果在请求fifo中有请求的话，而且该请求可以通过调用get()找回，函数nb_can_get()返回“真”。如果没有什么可获取的，就进行下一个主机。

```
63   bool ok_to_get = master_channel[portidx].get_request_export-
>nb_can_get();
64   if(!ok_to_get)
65   continue;
66
67 req = master_channel[portidx].get_request_export->get();
file: topics/04_tlm/06_tlm_bus_sc/bus.cc
```

放在变量图中的地址分配图被封装用来分配请求的地址给驱动程序，该程序将事务请求。如果地址分配图中出现错误，请求指针返回一个负值。在这种情况下，地址分配图发出一个出错信息，我们就将请求置底。

```
70 int reqidx = amap.lookup(req.addr);
71 if(reqidx < 0)
72    continue;
file: topics/04_tlm/06_tlm_bus_sc/bus.cc
```

一旦确定要事务请求和使用哪个驱动程序，我们就将请求转寄给驱动程序并等待响应。一旦接到响应就可以将它返回给请求程序。

```
74 slave_channel[reqidx].put_request_export->put(req);
75 rsp = slave_channel[reqidx].get_response_export->get();
76 master_channel[portidx].put_response_export->put(rsp);
file: topics/04_tlm/06_tlm_bus_sc/bus.cc
```

注意到放入和获得调用都是阻塞。这并不是说事件会提前，只是说事件可能提前。

这个小总线设计的方式就是一次只能复制一件事到总线上。一旦做好地址分配图，请求就要被处理，当驱动程序事务请求完成操作并返回响应后，请求才会承认总线。所有其它驱动程序都必须阻塞直到总线可以处理请求。

第 5 章　SystemVerilog 中的 AVM 机制

本章讨论如何构使用AVM类在SystemVerilog 中构建层次化的测试平台。特别要讨论在平面和分层结构中如何用事务级将组件联接在一起。

接口

将事件连接在一起需要接口。但是，接口有几个不同的用途。在AVM的SystemVerilog执行中，接口有三个意思。按照抽象逐渐增加的顺序，它们是：

- SystemVerilog 接口
- SystemVerilog 虚拟接口
- 纯虚接口类

SystemVerilog 接口

SystemVerilog中的一个接口就是一种即时阻塞。它基本上是一束组在一起的wire。接口用来连接DUT，也可以用来把DUT的子模块连接在一起。例如：

```
23 interface mem_pins_if;
24
25 parameter int ADDRESS_WIDTH = 8;
26 parameter int DATA_WIDTH = 8;
27
28 typedef bit[ADDRESS_WIDTH-1:0] address_t;
29 typedef bit[DATA_WIDTH-1:0] data_t;
30
31 address_t address;
32 data_t wr_data;
33 data_t rd_data;
34 bit clk , rst;
35 bit req , rw;
```

```
36 bit ack , err;
37
38 //...
39
60 endinterface : mem_pins_if
```

你可以按照例示用接口将驱动程序连接起来，而不是单独规定每一个端口。

```
27 module top;
28
29 mem_pins_if #( .ADDRESS_WIDTH( 8 ) , .DATA_WIDTH( 8 ) ) pins_if();
30
31 mem_master master( pins_if.master_mp );
32 mem_dut dut( pins_if.slave_mp );
33
51 endmodule
```

SystemVerilog 虚拟接口

一个虚拟接口是一个接口的指针。AVM事务器使用虚拟接口与DUT通话。通过虚拟接口指向真正的接口使得事务器能够再利用。事务器并不需要设计的是什么，只需要知道接口即可。如：

```
class mem_monitor #( int ADDRESS_WIDTH = 8 , int DATA_WIDTH = 8 )
extends avm_verification_component;
    virtual mem_pins_if #(
        .ADDRESS_WIDTH( ADDRESS_WIDTH ) ,
        .DATA_WIDTH( DATA_WIDTH )
    ) pins_if;
    analysis_port #( transaction_t ) ap;
    task run;
    forever begin
        @( posedge pins_if.monitor_mp.clk );
        if( pins_if.monitor_mp.rst ) begin
        state = WAIT_FOR_REQ;
        continue;
        end
    case( state )
    WAIT_FOR_REQ : begin … end
    WAIT_FOR_ACK : begin … end
    endcase
end
```

```
endtask
endclass
```

上面的代码用虚拟接口pins_if来监视类型为mem_pins_if的总线。但是，它并不了解接口所采用的设计，因此可以用它来监视任何设计中的这一类型的总线。

纯虚接口类

接口类是基类的抽象。通常，它的所有办法都是纯虚的。在AVM中，一个接口类通常是TLM接口中的一种，而且被用来在测试平台上转移事务信息。三个TLM put接口表示如下：

```
virtual class tlm_blocking_put_if #( type T = int );
   pure virtual task put( input T t );

endclass

virtual class tlm_nonblocking_put_if #( type T = int );
    pure virtual function bit try_put( input T t );
   pure virtual function bit can_put();

endclass

virtual class tlm_blocking_put_if #( type T = int );
    pure virtual task put( input T t );
    pure virtual function bit try_put( input T t );
    pure virtual function bit can_put();

endclass
```

这些接口类可以被认为是在消耗器和服务提供者之间指定协议。一方面，消耗器说“这就是我所需要的服务－但我并不在乎你是怎么做的”。服务提供者则说“这就是我所提供的－应该由你来决定用它做什么”。如果其它组件需要服务提供者提供的接口，就要同意协议。通常，消耗器要求尽可能少以避免多付，而服务提供者则提供得尽可能多以取得更大的市场份额。

端口和输出

端口和输出都是AVM库中的对象，这推动匹配请求和提供接口以形成连接。

端口

端口就是接口，它请求一个由外部提供的执行。端口的请求应当是越少越好：例如有一个端口tlm_blocking_put_if，而不是tlm_put_if，因为它不需要tlm_put_if中的非阻塞函数。

```
class avm_stimulus #( type trans_type = avm_transaction )
    extends avm_named_component;

  tlm_blocking_put_if #( trans_type ) blocking_put_port;

endclass
```

输出端口

另一方面，一个通道应该提供尽可能多的接口。例如，Tlm_fifo提供12个接口－在阻塞和非阻塞中的put, get, peek 和 get_peek ，以及它们的组合

形式。

```
class tlm_fifo #( type T = int , type CLONE = avm_built_in_clone #( T ) )
extends avm_named_component;
    typedef tlm_fifo #( T , CLONE ) this_type;
    tlm_put_imp #( this_type , T ) put_export;
    tlm_blocking_put_imp #( this_type , T ) blocking_put_export;
    tlm_nonblocking_put_imp #( this_type , T ) nonblocking_put_export;
// and similar for get, peek and get_peek
    function new( string name = "" ,
    avm_named_component parent = null ,
    int size = 1 );
    super.new( name , parent );
    put_export = new( this );
    blocking_put_export = new( this );
    nonblocking_put_export = new( this );
// and similar for get, peek and get_peek
    endfunction
    task put( input T t ); … endtask
    function bit try_put( input T t ); … endfunction
    function bit can_put(); … endfunction
// and similar for get, peek and get_peek
endclass
```

环境类

在AVM的SystemVerilog Implementation的基于类的版本中使用环境类。在SystemVerilog测试平台中的所有基于类的验证组件都包含在环境类中。通常，任何环境类的仅有的2个公开可视的方法是constructor 和 do_test 任务。

```
module top;
    mem_pins_if #( .ADDRESS_WIDTH( 8 ) , .DATA_WIDTH( 8 ) ) pins_if();
    mem_master master( pins_if.master_mp );
    mem_dut dut( pins_if.slave_mp );
    clock_reset cr( pins_if );
    mem_env #( 8 , 8 ) env;

    initial begin
        env = new( pins_if );
        env.do_test;
        $finish;
```

```
    end
endmodule
```

在上面的代码中，我们用mem_pins_if将主机和从机连接起来。这个接口被传送到环境类的构造器，这样，一个嵌套在环境类中的基于类的组件就可以监视总线的行为。

do_test方法有5个阶段。该方法中的每一阶段都是函数，除了执行阶段。这就是说，在执行阶段一定会消耗时间，或甚至调用@, wait or fork ... join。这5个阶段分述如下：

1 构造

构造阶段就是在测试平台中构建所有的基于类的验证组件。它也将虚拟接口的当地复制件从顶层模块或程序块中传送到构造器。

2 连接

连接阶段就是将基于类的验证组件连接在一起。详情参见连接阶段。

3 配置

配置阶段在仿真时间0时刻配置。这包括后门存储器初始化，或配置驱动器的参数以确定错误方式率，或配置报告组件。每个组件都有一个称为configure()的函数，这个阶段就是调用这个函数。

4 执行

执行阶段是唯一的一个在这个阶段测试组件可以通过调用@, wait or fork... join与SystemVerilog进程相互作用的阶段。执行阶段用了两个方法，avm_env::execute 和 avm_verification_component::run.

do_test任务并行运行在所有avm_verification_components中的run任务（即从avm_verification_component派生的组件）。这些都用fork…join_none，因此它们都是独立运行的。构造有run任务（在do_tests中被调用）的组件通常是事务器，它们必须从时刻0开始。一旦任务被执行，就调用avm_env类中的execute。Execute是一个很好的放置测试控制功能性的地方。例如，也许你希望用execute任务去启动激励发生器，该发生器的操作是由外部的记分板或覆盖率采集器控制的

5 报告

每个测试组件的report方法都会被调用。report方法放置仿真信号的地方，通常报告事件，比如全部的覆盖率统计表和记分板的成功/失败状态。

连接阶段

在连接阶段，你要指定测试组件之间的相互联系。组件可以被连接起来而不分层，所有的组件在一个级的不存在分层，或分层次体系（它们的组件在不同的级）。

不分层次的绑定

当在一个连接函数中将输出口分配给端口时，它们之间就有了协议。假设我们想将下面的发生器和消耗器连接到tlm_fifo。

```
class producer extends avm_verification_component;

  tlm_blocking_put_if #( int ) put_port;

  task run();

    for( int i = 0; i < 10; i++ ) begin
        $display("about to put %d" , i );
        put_port.put( i );

    end
 endtask;
endclass
```

```
class consumer extends avm_verification_component;

  tlm_blocking_get_if #( int ) get_port;

   task run();
  int i;
     forever begin

      get_port.get( i );
      $display( "Just got %d" , i );
 end
 endtask
endclass
```

上面的两个运行方法假定put_por和get_port都不为空－它们请求执行由外部提供的阻塞put和get 接口，如果没有执行，运行方法会引起运行时间错误。

```
class my_env extends avm_env;
   producer p;
   consumer c;
   tlm_fifo #( int ) f;
   function new;
      p = new("producer");
      c = new("consumer");
      f = new("fifo");
   endfunction
   function void connect;
      p.put_port = f.blocking_put_export;
      c.get_port = f.blocking_get_export;
   endfunction
   task execute;
      #100;
   endtask
endclass
```

This pair of assignments seals the contract between *requires* side and *provides* side of each interface

tlm_fifo给发生器和消耗器提供阻塞接口。当输出口分配给端口时，在上面的连接方法中同意协议。基类avm_env在发生器和消耗器中开始运行方法以前自动调用连接，这样，当运行方法开始时，端口就不为0，发生器和消耗器就会按照预期工作。

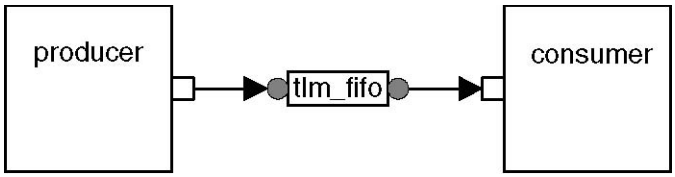


图5-1 调用连接之前的端口和输出口

图5-1的阴影表示在调用connect方法之前，输出口不为0，而图5-2则表示在执行运行任务之前端口和输出口不为0。

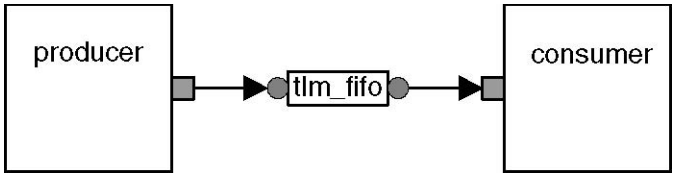


图5-2调用连接之后的端口和输出口

端口，输出口和继承

简单的测试可以用平面结构完成。对于这些简单的测试，采用上一节讲的无继承端口和输出口绑定就足够了。

但是，有大量的再利用的，更为复杂的测试要求在测试中采用继承。考虑将驱动器中的tlm_fifo通过继承与发生器相连。

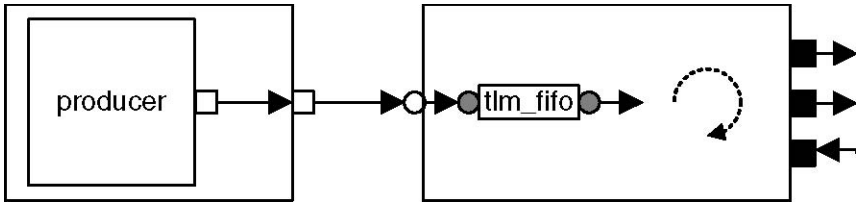


图5-3 调用连接之前的层次

在顶层，将出口和输出口连接在平面上。但是，这里引入了两种新的连接类

型：在驱动器中，有一个输出对输出的连接，在分级发生器中，有端口对端口的连接。

为了在顶层调用连接之前得到驱动器的export，就在称为export_connections的方法中进行输出对输出的分配。

```
class driver extends avm_verification_component;
    tlm_blocking_put_if #( my_transaction ) blocking_put_export;
    virtual interface m_bus_if;

  local tlm_fifo #( my_transaction ) fifo;

  function new( string name , avm_named_component parent = null );
      super.new( name , parent );
      fifo = new("fifo");

  endfunction

  function void export_connections;

blocking_put_export = fifo.blocking_put_export;

  endfunction

 task run;
      // implement state machine, stimulate bus
   endtask
  endclass
```

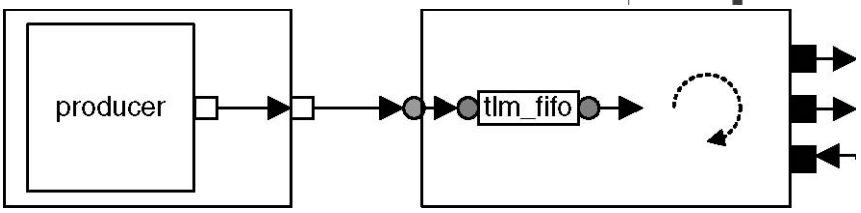


图5-4 分配给分级输出

在顶层的connect方法将顶层端口分配给顶层输出端口。

```
class my_env extends avm_env;
   hierarchical_producer hp;
   driver d;
   tlm_fifo #( int ) f;
   // virtual interface omitted

 function new;
   hp = new("producer");
   d = new("driver");

 endfunction

 function void connect;

hp.blocking_put_port = d.blocking_put_export;

 endfunction

 task execute;
    #100;
    endtask
   endclass
```

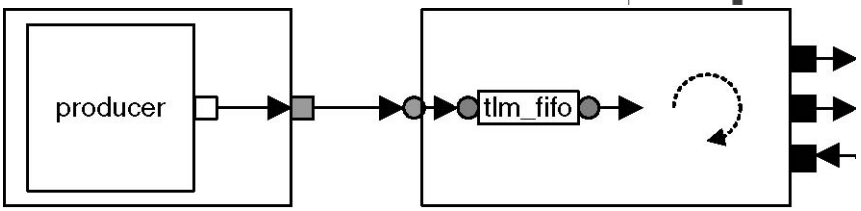


图5-5 分配给分级端口

最后，需要在分级发生器中将顶层端口传送到低级端口。通过采用import_connections方法来完成这一步：

```
class hierarchical_producer extends avm_named_component;

    tlm_blocking_put_if #( my_transaction ) blocking_put_port;

   local producer p;


  function new( string name , avm_component parent = null );

   super.new( name , parent );

   p = new("producer");


  endfunction

  function void import_connections;

  p.blocking_put_port = blocking_put_port;


    endfunction

endclass
```

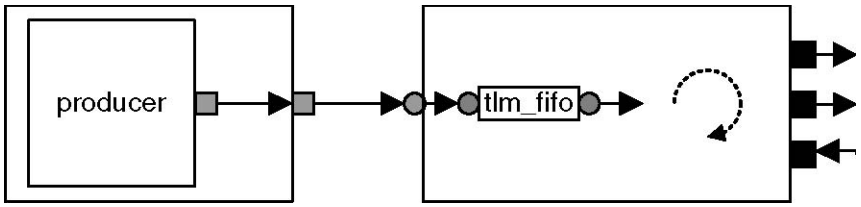


图5-6 分配给内部端口

测试设计师并不需要制定出按照什么样的顺序来调用这些方法。验证工程师所要做的工作就是确定一个连接是端口对输出口的连接，输出口对输出口的连接或端口对端口的连接，并将它放在适当的方法中。AVM库则确认是否按

照正确的顺序调用了正确的方法。

表5-1

连接类型	方法	方向
输出口-输出口	export_connections	上部，出
端口-输出口	connect	平连
端口-端口	import_connections	下面 进

连接分析端口

一个分析端口就是一个面向对象设计模式的执行，称作观察器。它在分析接口目录周围的一层执行。因为这里可能有多个接口，我们不能简单的采用分配来进行连接。相反，我们采用寄存器函数给目录增加接口。

考虑给两个覆盖率对象连接一个监视器：

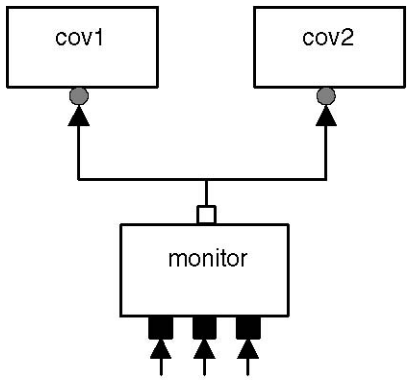


Figure 5-7. Analysis Ports Before Calling connect

```
class my_env extends avm_env;
monitor m;
cov1 c1;
cov2 c2;
function new;
    m = new("monitor");
    c1 = new("cov1");
    c2 = new("cov2");
endfunction
function void connect;
```

```
  m.ap.register( c1.analysis_export );
  m.ap.register( c2.analysis_export );
endfunction
endclass
```

上面代码中的register函数给分析端口中的接口目录增加了两个分析接口，这个目录的结果就不再是空。当监视器调用write时，就调用了两个覆盖率对象中的write方法。

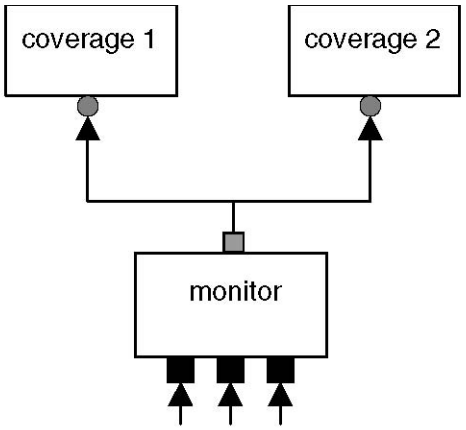


Figure 5-8. Analysis Ports After connect Finishes

分析端口和层次

用export_connections中的赋值，在继承的上部输出分析输出，在其它接口也是这样操作。之所以能进行这样的操作是因为一个分析输出口只是一个接口。

但是，分析端口是接口列表，它们需要用寄存器函数进行输入。考虑在顺序执行的总线上的一个分层次的监视器：

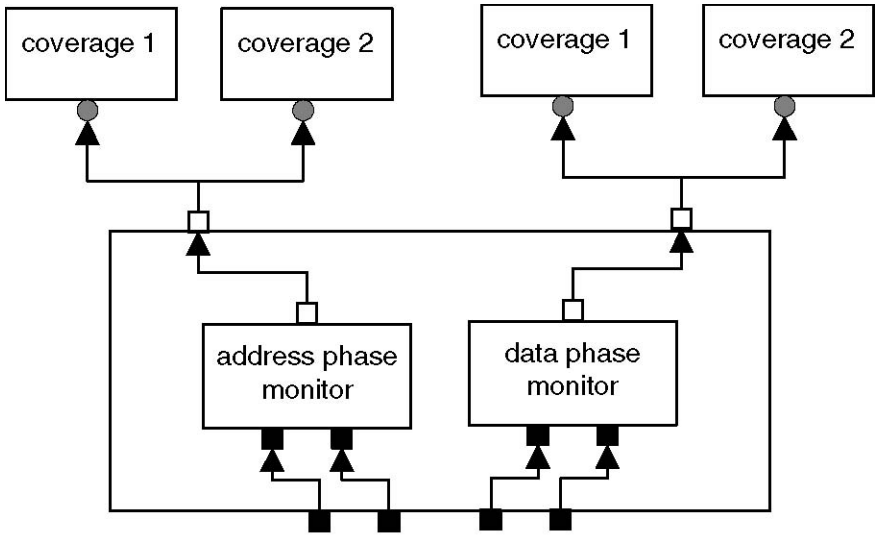


Figure 5-9. Hierarchical Monitor on Pipelined Bus

在顶层，采用常规方法使用connect方法中的register。

```
class my_env extends avm_env;
  monitor m;
  cov1 c1;
  cov2 c2;
  cov3 c3;
  cov4 c4;
  function new;
      m = new("monitor");
      c1 = new("cov1");
      c2 = new("cov2");
      c3 = new("cov3");
      c4 = new("cov4");
    endfunction
  function void connect;
      m.address_ap.register( c1.analysis_export );
      m.address_ap.register( c2.analysis_export );
      m.data_ap.register( c3.analysis_export );
      m.data_ap.register( c4.analysis_export );
  endfunction
endclass
```

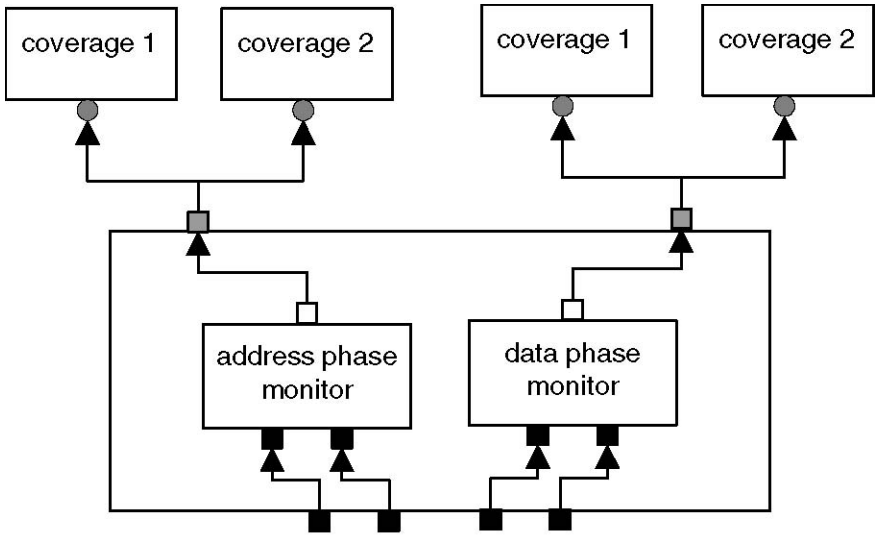


Figure 5-10. Registering Listeners to Analysis Ports

为了将连接输入到次级模块，将父级端口和低级端口登记在一起（记住首先构造父级分析端口）。

```
class pipelined_monitor extends avm_named_component;
   analysis_port #( address_transaction ) address_ap;
   analysis_port #( data_transaction ) data_ap;
   local address_monitor m_address_monitor;
   local data_monitor m_data_monitor;
   function new( string name , avm_named_component parent = null );
      super.new( name , parent );
      address_ap = new;
      data_ap = new;
      m_address_monitor = new("address_phase");
      m_data_monitor = new("data_phase");
   endfunction
   function void import_connections;
      m_address_monitor.ap.register( address_ap );
      m_data_monitor.ap.register( data_ap );
   endfunction
endclass
```

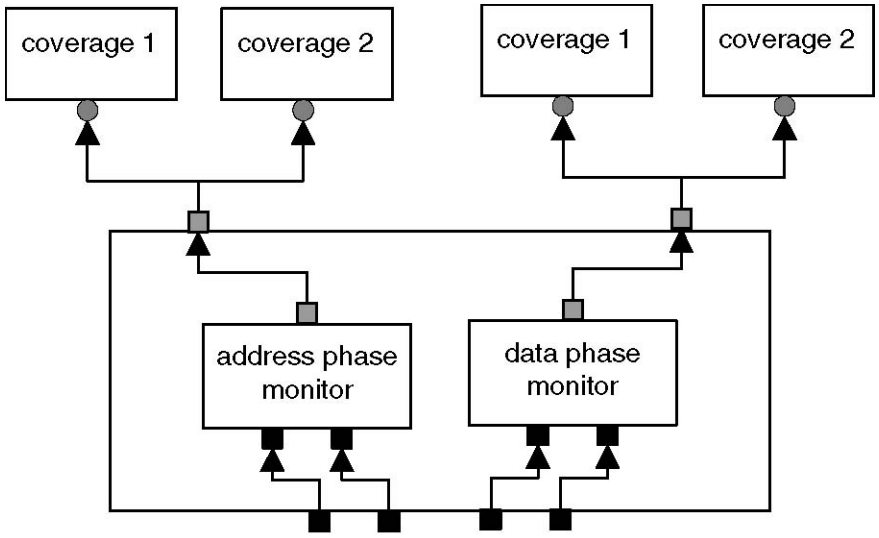


Figure 5-11. Registering Hierarchical Analysis Ports

在继承的每一层次构造一个新的分析端口，然后用寄存器将它们连接起来，这对于我们在层次化的任意一层增加覆盖率或记分板是非常有用的，如图5-12所示。

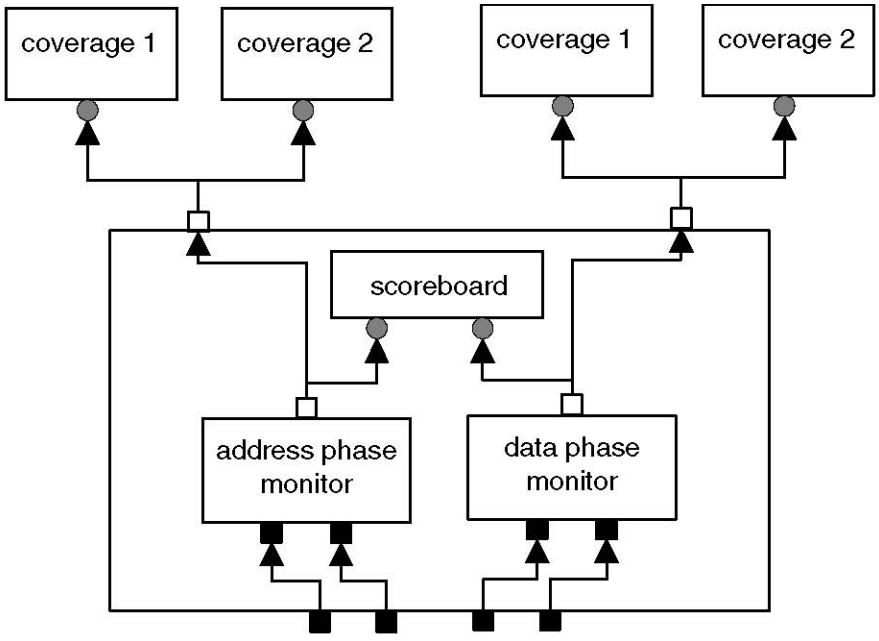


Figure 5-12. Using Analysis Ports at Multiple Levels of Hierarchy

虚拟接口和 avm_env

可是接口只是一种端口。它们是由枝叶上的事务器要求并由DUT本身提供的

一些指针。

一个环境类对每一个它想连接的接口必须有一个虚拟接口：在这个意义上，它与一个端口列表avm_named_component没什么区别。但是，虚拟接口是通过创建函数，从顶层模块或program模块输入的。而不是用import_connections函数连接事务器。

例如，假设有一个监视器在检测总线，如图5-13所示：

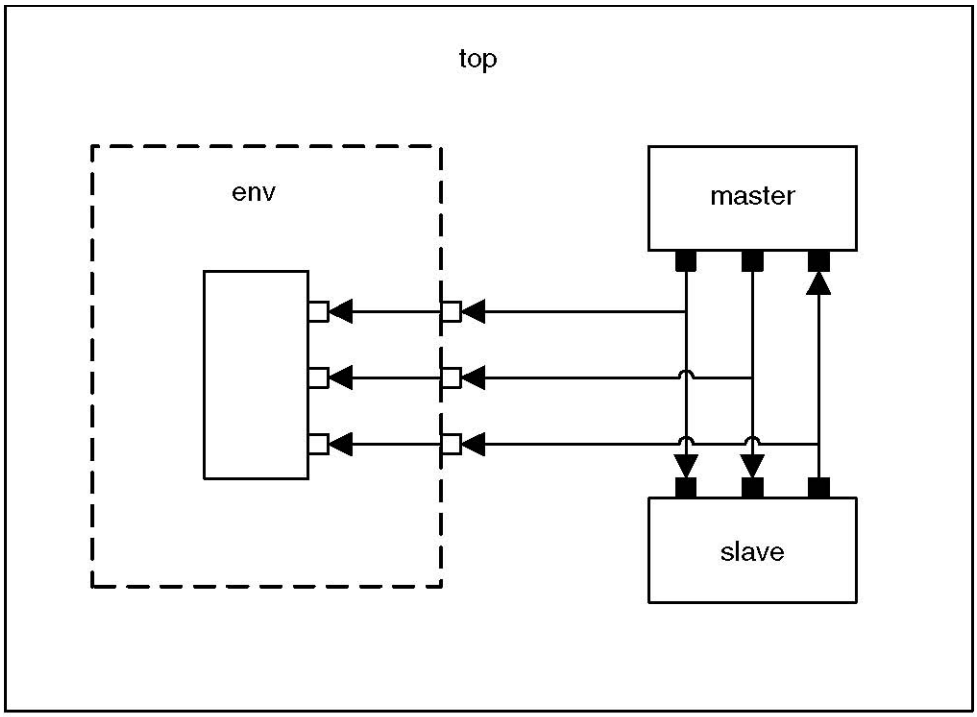


Figure 5-13. Monitor Connected to a Bus

在顶层模块，我们创建环境，通过构造器传入接口：

```
module top;
    mem_pins_if pins_if();
    mem_master master( pins_if.master_mp );
    mem_dut dut( pins_if.slave_mp );
    mem_env env;
    initial begin
        env = new( pins_if );
        env.do_test();
        $finish;
    end
endmodule
```

在这个环境里，首先在构造器里存放虚拟接口的当地副本：

```
class mem_env extends avm_env;
    local virtual mem_pins_if m_bus_if;
    local mem_monitor #( ADDRESS_WIDTH , DATA_WIDTH ) m_monitor;
    function new( virtual mem_pins_if bus_if );
        m_bus_if = bus_if;
        m_monitor = new("monitor");
    endfunction
    …
endclass
```

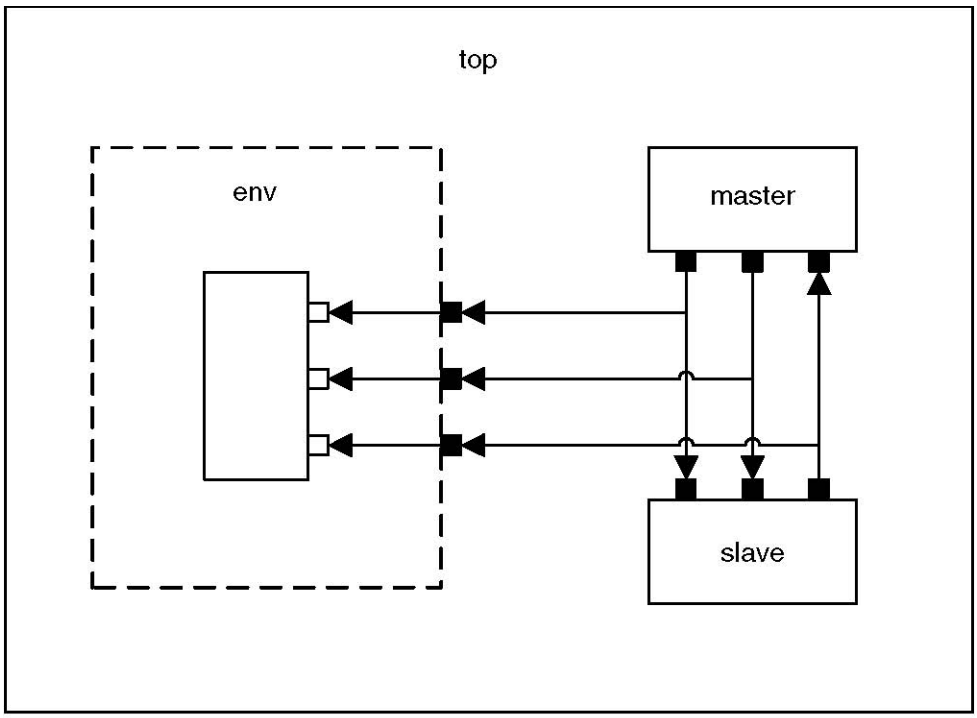


Figure 5-14. Connecting Environment to an Interface

最后，我们将虚拟接口的当地副本赋给环境类连接函数里的监视器：

```
class mem_env extends avm_env;
    function void connect;
        m_monitor.pins_if = m_bus_if;
    endfunction
    task execute;
        # 1000;
    endtask
endclass
```

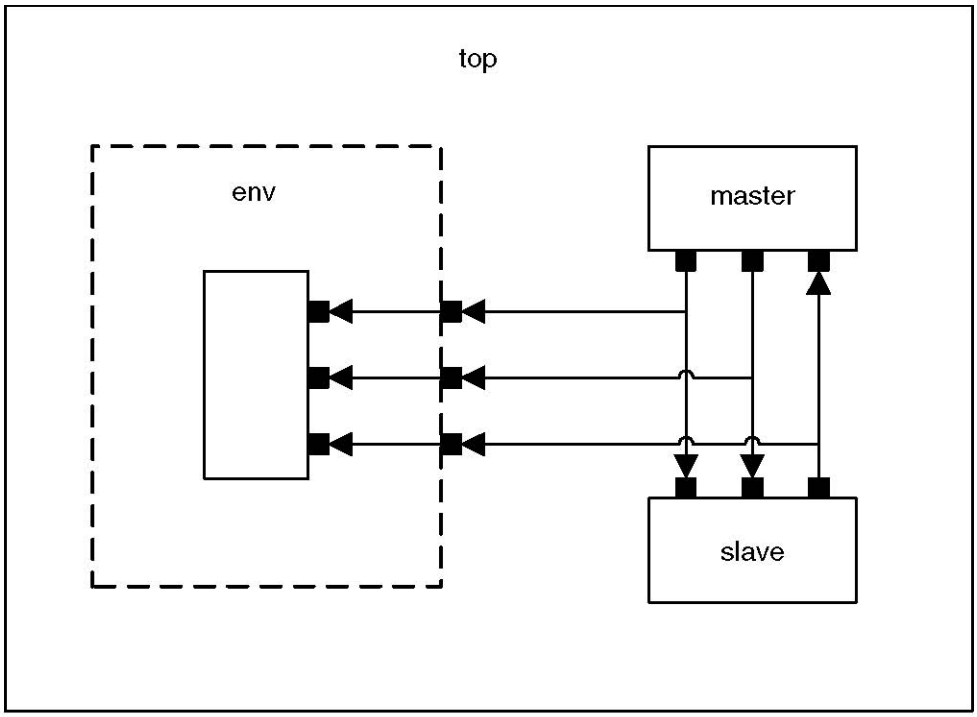


Figure 5-15. Passing the Virtual interface to Components in the Environment

在更为复杂的测试平台中，也许有多个驱动程序，因此就需要多个接口来与测试平台通讯。我们需要将所有的接口传送到构造器，在环境级创建它们的副本，并将它们都分配给连接函数中的相应事务器。对于层次化的测试平台，我们必须用import_connections将虚拟接口传送到测试平台的更深处，就像在层次中传送其它的端口一样。

总结

AVM提供了事务级接口和连接器，因为它们都没有嵌入到AVM语言中。事务接口有两面：请求面和提供面。一个port端口表示请求一半接口，一个export输出口表示提供的另一半接口。连接函数中的一个简单分配将这两半连接起来形成一个完整的连接。avm_env类提供连接函数并调用形成连接的程序。

第 6 章　测试基本原理

一个测试至少要做两件事情。它必须提供一种途径用于：

产生激励并将它用到DUT。

在仿真过程中，使用激励时，观察DUT的行为。

在这一章里，要讨论激励发生器，它用于创造和应用激励，以及监视器，它用来观察系统行为。

一个存储器的测试平台

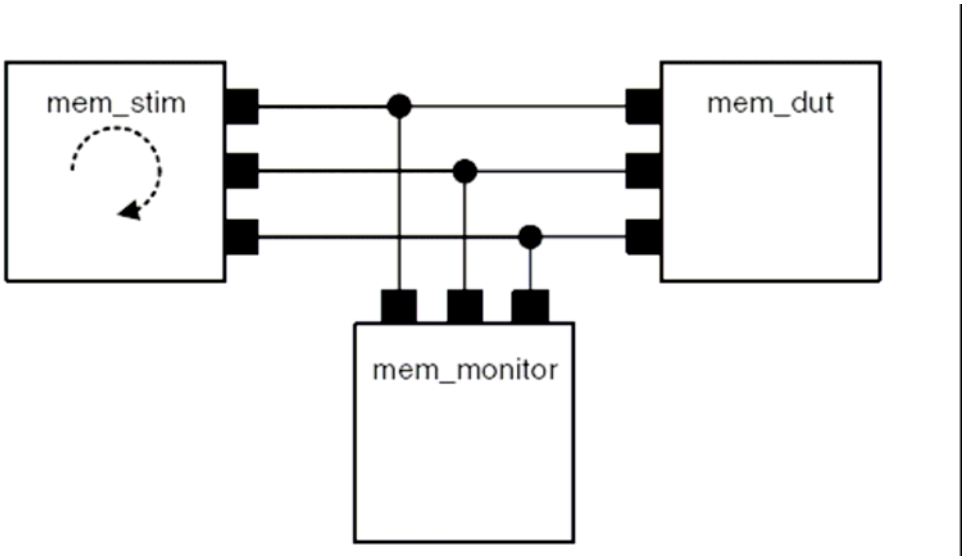


图6-1 存储器测试

说明

简单的测试只包括三个组件：一个RTL主机，一个RTL从机和一个监视器。

主要概念

- 一个监视器是一个被动装置，它并不影响DUT的操作。它是用一个类似于

DUT中的状态机构成的。监视器的任务就是监视总线上的行为并报告它所看到的事情。

- 监视器有一个引脚接口来监视总线上的行为。它用这个引脚的行为来驱动识别事务信息的内部状态机。
- 一个监视器将引脚级行为转变为事务信息流。

监视器结构

监视器的基本结构是一个时钟驱动的状态机。这个状态机是用case语句实现的。case语句中的每一个选择都代表了一个特定状态的行为。这个行为代码读入引脚信号，还可能引起状态的改变，并引起一组不同的行为来在下一个时钟周期运行。

监视器的主要目的就是将信号级行为转变为一系列的事务信息。状态行为代码做两件事情来进一步完成这个目的。在每一个时钟周期中，它：

- 观看引脚的状态来评估总线的状态。
- 收集数据并组装起来给事务对象。

SystemVerilog 实现细节

这个代码在时钟的上升沿触发。一旦复位，状态机就返回到WAIT_FOR_REQ。在WAIT_FOR_REQ中，它产生请求，在WAIT_FOR_ACK中，它产生相匹配的响应并将请求和响应联结为一个事务信息。

```
51          forever begin
52
53            @( posedge pins_if.monitor_mp.clk );
54
55            if( pins_if.monitor_mp.rst ) begin
56              state = WAIT_FOR_REQ;
57              continue;
58            end
59
60            case( state )
61            WAIT_FOR_REQ : begin
62
63              if( pins_if.monitor_mp.req ) begin
64
65                request = read_request_from_bus();
66
67                avm_report_message( "Saw Mem Request" ,
68                                    request.convert2string() );
69
70                state = WAIT_FOR_ACK;
71              end
72
73            end
74            WAIT_FOR_ACK : begin
75
76              if( pins_if.monitor_mp.ack ) begin
77
78                response = read_response_from_bus();
79
80                avm_report_message("Saw Mem Response" ,
81                                   response.convert2string() );
82
83                transaction = new( request , response );
84
85                avm_report_message("Saw Mem Transaction" ,
86                                   transaction.convert2string() );
87
88                ap.write( transaction );
89
90                state = WAIT_FOR_REQ;
91              end
92
93            end
94            endcase
file: topics/06_testbench_fundamentals/mem_svc/mem_monitor.svh
```

为了方便起见，将函数read_request_from_bus和read_response_from_bus的代码表示如下：

```
99      function request_t read_request_from_bus;
100
101       request_t request;
102
103       request = new( pins_if.monitor_mp.address ,
104                      ( pins_if.monitor_mp.rw ? MEM_WRITE : MEM_READ ) ,
105                      pins_if.monitor_mp.wr_data );
106
107       return request;
108
109     endfunction
110
111     function response_t read_response_from_bus;
112
113       response_t response;
114
115       response =
116         new( ( pins_if.monitor_mp.rw ? MEM_WRITE : MEM_READ ) ,
117              ( pins_if.monitor_mp.err ? MEM_ERROR : MEM_SUCCESS ) ,
118              pins_if.monitor_mp.rd_data );
119
120      return response;
121
122     endfunction
file: topics/06_testbench_fundamentals/mem_svc/mem_monitor.svh
```

在这两种情况下，我们询问引脚级信号的值并将合适的值传送给事务构造器。然后返回新创建的事务信息给状态机用。

并没有必要使用便捷函数，但便捷函数确实可以使我们简化状态机的设计。

SystemC 实现细节

监视器状态机的SystemC执行很像SystemVerilog执行。

在构造器中方法run()是对时钟的上升沿敏感的：

```
23    mem_monitor::mem_monitor( sc_module_name nm ) :
24      sc_module( nm ) {
25
26      SC_METHOD( run );
27      sensitive << clk.pos();
28      dont_initialize();
29
30    }
file: topics/06_testbench_fundamentals/mem_sc/mem_monitor.cc
```

run 的实现方式与SystemVerilog的实现方式一致:

```
32    void mem_monitor::run() {
33
34      if( rst ) {
35        m_state = WAIT_FOR_REQ;
36      }
37
38      else {
39
40        switch( m_state ) {
41        case WAIT_FOR_REQ :
42
43          if( req ) {
44
45            read_request_from_bus();
46
47            cout << name() << " Saw Mem Request " << m_request
48                 << " at " << sc_time_stamp() << endl;
49
50            m_state = WAIT_FOR_ACK;
51          }
52
53          break;
54
55        case WAIT_FOR_ACK :
56
57          if( ack ) {
58
59            read_response_from_bus();
60
61            m_transaction.m_request =  m_request;
62            m_transaction.m_response = m_response;
63
64            ap.write( m_transaction );
65


66             cout << name() << " Saw Mem Transaction " << m_transaction
67                  << " at " << sc_time_stamp() << endl;
68
69             m_state = WAIT_FOR_REQ;
70           }
71
72           break;
73
74         }
75       }
76     }
file: topics/06_testbench_fundamentals/mem_sc/mem_monitor.cc
```

便捷函数从引脚读入数据给请求和响应数据结构：

```
78    void mem_monitor::read_request_from_bus() {
79      m_request.m_address = address.read();
80      m_request.m_type = rw ? MEM_WRITE : MEM_READ;
81      m_request.m_data = wr_data.read();
82    }
83
84    void mem_monitor::read_response_from_bus() {
85      m_response.m_type =  rw ? MEM_WRITE : MEM_READ;
86      m_response.m_err = err ? MEM_ERROR : MEM_SUCCESS;
87      m_response.m_rd_data = rd_data.read();
88    }
file: topics/06_testbench_fundamentals/mem_sc/mem_monitor.cc
```

SystemC 和 SystemVerilog 实现之间的唯一明显的区别就是，既然我们在SystemC中可以用值传递的方式，那我们每次在确定一个新的事务时就没有必要明确地调用new函数。

带独立驱动器的存储器测试

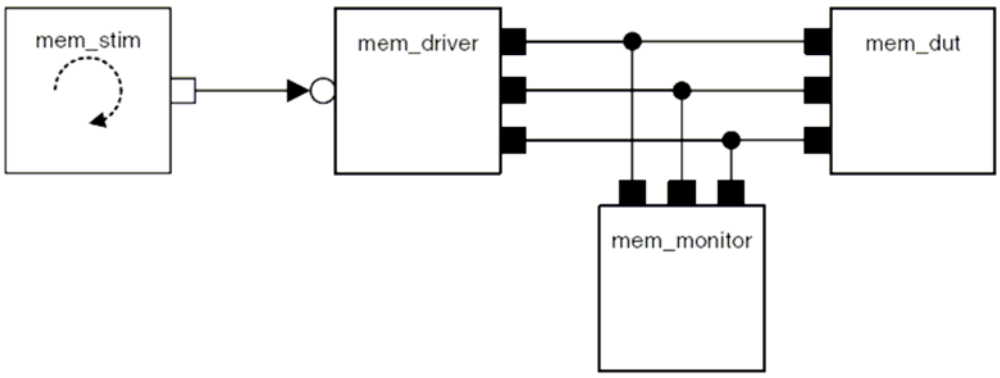


图6-2 存储器测试

说明

本例中，我们将上例中的RTL主机用一个事务级激励发生器和一个驱动器来代替。

主要概念

- 激励发生程序独立于驱动器，这使得它们可以重复利用。我们可以将不同的激励发生器连接到同一个驱动器，我们也可以通过将激励发生器连接到不同的驱动器而在系统级使用同一个激励发生器。
- 激励发生器的任务就是产生事务信息流。

- 驱动器的任务就是将这个事务信息流转化为引脚级行为。
- 采用发生器模式来精心调整随机激励发生程序。

事务级激励发生器

事务类代表了在主机和从机之间进行通讯的数据传递，而不需要详细描述发生在实际总线上的时序和握手信号。

一个激励发生器产生一个事务信息流。在本例中，这个信息流是随机产生的，但它也可能需要产生定向事务信息流－要么是用来在随机测试开始前执行一个初始化序列，要么是用来执行测试本身。

在我们简单的存储器协议中，激励发生器产生操作（读或写），地址和用于写的数据。存储器协议的细节留给了驱动器。

驱动器结构

驱动器将抽象的激励转化为具体的激励。为此，它必须是总线上的引脚级协议操作。

在这个简单例子中，驱动器是一个阻塞驱动器：它直接执行有激励发生器调用的阻塞put方法。如果总线处在还不能接受一个新的请求或提供新的响应的状态，这个put方法就阻塞。

在后面的各章中，我们会看到其它的使用fifo来管理激励发生器和驱动器之间通讯的驱动器，以及使用状态机来驱动总线的驱动器。这些驱动器更适用于更为复杂的总线协议并能处理许多激励发生器而不仅仅是如本例所示的一个。

SystemVerilog 实现细节

激励发生器执行

执行方法是用来控制激励的产生，它会依次控制其余的测试。测试平台的控制是通过发生器模式来实现的。

首先，定义请求事务类型，它们的所有属性都是随机的：

```
26    class mem_request #( int ADDRESS_WIDTH = 8 , int DATA_WIDTH = 8 )
27      extends avm_transaction;
28
29      typedef bit[ADDRESS_WIDTH-1:0] address_t;
30      typedef bit[ADDRESS_WIDTH-1:0] data_t;
31
32      typedef mem_request #( ADDRESS_WIDTH , DATA_WIDTH ) request_t;
33
34      rand address_t m_address;
35      rand data_t    m_wr_data;
36      rand mem_transaction_type_e m_type;
37
38      function new( address_t address = 0  ,
39                    mem_transaction_type_e transaction_type = MEM_REAJ
40                    data_t wr_data = 0 );
41
42        m_address = address;
43        m_wr_data = wr_data;
44        m_type = transaction_type;
45
46      endfunction
47
48      function request_t clone();
49        request_t request;
50
51        request = new( m_address , m_type , m_wr_data );
52        return request;
53      endfunction
54      // ...
file: topics/06_testbench_fundamentals/mem_tr_sv/mem_transaction.svh
```

特别重要的一点是克隆方法，激励发生器用这个方法给每一个事务创建新的副本。在SystemVerilog中，必须防止当前事务被下一个事务的随机化意外地覆盖。只要我们有事务器处理mem协议，就要使用请求类。在大型组织中，这些事务类是有一个专业的验证基础团队一次写成的，并将它交给组织的其它团队。

下面，我们定义men请求类的本地扩展。我们将men请求类扩展为带有新的约束，这些约束现在只在这个特殊的测试平台中可以用。

28

```
class write_request #( int ADDRESS_WIDTH = 8 , int DATA_WIDTH = 8 )
29 extends mem_request #( ADDRESS_WIDTH , DATA_WIDTH );
```

```
30
31 constraint write_only { this.m_type == MEM_WRITE; }
32
33 endclass
34
35  class read_request #( int ADDRESS_WIDTH = 8 , int DATA_WIDTH =
8 )
36 extends mem_request #( ADDRESS_WIDTH , DATA_WIDTH );
37
38 constraint read_only { this.m_type == MEM_READ; }
39
40 endclass
```

file: topics/06_testbench_fundamentals/02_mem_svc/mem_env.sv

最后，在测试平台的执行方法中，我们告诉激励发生器产生：10个随机写，10个随机读，最后，一个受时间限制的但不知道多少数量的完全随机事务。

```
90          task execute;
91
92            m_stimulus.generate_stimulus( m_write_gen , 10 );
93            m_stimulus.generate_stimulus( m_read_gen , 10 );
94
95            fork
96              m_stimulus.generate_stimulus;
97              terminate;
98            join
99
100         endtask
101
102         task terminate;
103           #100;
104           m_stimulus.stop_stimulus_generation;
105         endtask
```

file: topics/06_testbench_fundamentals/02_mem_svc/mem_env.sv

SystemC实现细节

激励发生器执行

在SystemC中，所有的激励发生器的组织类似于SystemVerilog中的组织，尽管不是完全一样。因为有了值传递概念和操作符重载，实际的事务定义就简单多了，但是用SystemC验证库(SCV)产生激励的机制却麻烦多了。

首先，我们定义一个事务类：

```
28    class mem_request {
29     public:
30      ADDRESS_TYPE m_address;
31      DATA_TYPE m_data;
32      mem_transaction_type_e m_type;
35    };
file: topics/06_testbench_fundamentals/mem_sc/mem_transaction.h
```

这个事务类的定义比SystemVerilog中相当的定义要简单一些，因为我们只是利用了内嵌的构造器，分配操作符，数据流操作符和比较操作符，而不是定义我们自己的方法。

下面，我们需要为这个类产生一个scv 扩展名：

```
45    template<>
46    class scv_extensions<mem_request> : public
scv_extensions_base<mem_request>
47    {
48     public:
49      scv_extensions< ADDRESS_TYPE > m_address;
50      scv_extensions< DATA_TYPE> m_data;
51      scv_extensions<mem_transaction_type_e> m_type;
52
53      SCV_EXTENSIONS_CTOR(mem_request)
54      {
55        SCV_FIELD(m_address);
56        SCV_FIELD(m_data);
57        SCV_FIELD(m_type);
58      }
59    };
file: topics/06_testbench_fundamentals/mem_sc/mem_transaction_ext.h
```

这个扩展名告诉scv我们想要随机化的每一个领域。

通过scv_smart_ptr这个扩展名用来创建一个基本约束类，产生不受约束的请求：

```
64 class mem_request_constraint_base : public scv_constraint_base
65 {

66     public:
67     scv_smart_ptr< mem_request > req;
68
69     SCV_CONSTRAINT_CTOR( mem_request_constraint_base )
70     {}
71     };
```

file:

topics/06_testbench_fundamentals/mem_sc/mem_transaction_ext.h

智能指针req包含随机化请求。

在一个大组织中，所有这些艰巨的工作都是由验证基础团队完成的，并公布给组织的其它成员。测试书写人员的任务就要简单多了：他们既可以使用基本约束类，也可以自己添加。

```
22 class mem_read_request : public mem_request_constraint_base
23 {
24 public:
25 SCV_CONSTRAINT_CTOR( mem_read_request )
26 {
27 SCV_BASE_CONSTRAINT( mem_request_constraint_base );
28
29 SCV_CONSTRAINT( req->m_type() == MEM_READ );
30 }
31 };
32
33 class mem_write_request : public mem_request_constraint_base
34 {
35 public:
36 SCV_CONSTRAINT_CTOR( mem_write_request )
37 {
38 SCV_BASE_CONSTRAINT( mem_request_constraint_base );
39
40 SCV_CONSTRAINT( req->m_type() == MEM_WRITE );
41 }
42 };
```

file: topics/06_testbench_fundamentals/02_mem_sc/top.cc

在上面的代码中，我们定义了两个本地约束类，它们只产生读和写。然后，我们在称作run的SC_THREAD中产生10个读，10个写，20个不受约束的请求：

```
90 void top::run() {
91
92 mem_write_request write_gen(“write_gen”);
93 mem_read_request read_gen(“read_gen”);
94
95 m_stimulus.generate_stimulus( write_gen , 10 );
96 m_stimulus.generate_stimulus( read_gen , 10 );
97
98 m_stimulus.generate_stimulus( 20 );
99
100 }
```

file: topics/06_testbench_fundamentals/02_mem_sc/top.cc

带独立驱动器和激励发生器的存储器TB

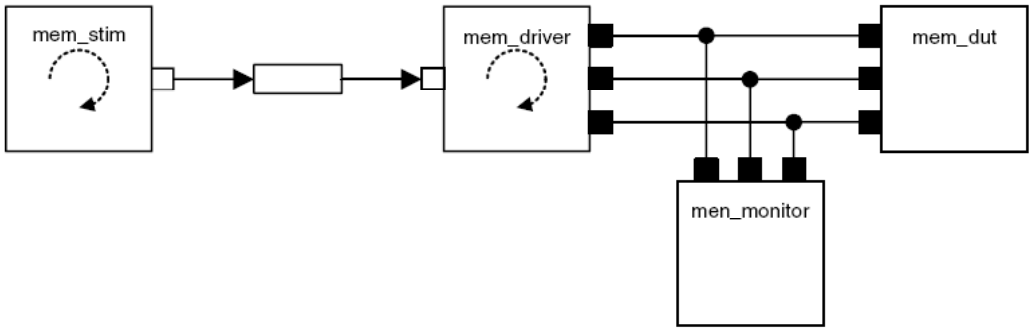


图6-3 存储器测试

说明

激励发生器将事务放入fifo中。驱动器执行状态机来控制总线。在状态机的合适点，驱动器用不阻塞get来看在fifo中是否由事务信息。

主要概念

- 复用前面例子中的具有不同驱动器的激励发生器。
- 用fifo在激励发生器和驱动器之间进行缓冲和同步。
- 使用状态机模拟总线
- 执行可重置的驱动器。

驱动器设计

因为本例中的tlm_fifo和前面的例子中的阻塞驱动器都在执行同一个阻塞put界面，我们可以再次利用前面例子中的激励发生器，不需作任何修改。本例中的驱动器不同于前面的例子，那里激励发生器和驱动器各自在自己的thread中操作。它们通过tlm_fifo通讯。

使用这个建模类型有许多优点。在SystemC中，它表示我们可以将同步点（如等待语句）降低到最少，这反过来也可以影响仿真效果。总线协议可以写成状态机，通常和说明书中描述的一样。给程序使用独立的thread，你可以用写代码的方式来给中断和复位建模，这样既容易理解也便于维护。

另外，一个将驱动器和监视器分开的更为微妙的优点是不需要额外的机器来处理由于重新进入引起的问题。Fifo有一个内嵌的互斥体，因此状态机一次只需要处理一个事务。这使得我们可以在一个驱动器上连接多个激励发生器，而不必明确的增加旗语或互斥体。

SystemVerilog 实现细节

驱动器执行

```
20    class mem_driver #( int ADDRESS_WIDTH = 8 ,
21                        int DATA_WIDTH = 8 )
22      extends avm_verification_component;
23
24      typedef mem_request #( ADDRESS_WIDTH , DATA_WIDTH ) request_
25      typedef mem_response #( DATA_WIDTH )    response_t;
26
27      typedef enum { WAIT_FOR_REQ , WAIT_FOR_ACK } state_e;
28
29      virtual mem_pins_if #(
30        .ADDRESS_WIDTH( ADDRESS_WIDTH ) ,
31        .DATA_WIDTH( DATA_WIDTH )
32      ) pins_if;
33
34      tlm_nonblocking_get_if #( request_t ) request_port;
35
36      function new( string nm , avm_named_component p = null );
37        super.new( nm , p );
38      endfunction
file: topics/06_testbench_fundamentals/mem_svc/mem_driver.svh
...
endclass
```

上面概述了mem_driver的外部连通性。它有一个不阻塞get端口，该端口用来从fifo中抽取事务信息，还有一个虚拟接口，它用来管理总线上的引脚。构造器只是简单的将名称和指针与基类记录在一起。

状态机描述如下：

```
46        forever begin
47          @( posedge pins_if.master_mp.clk );
48
49          if( pins_if.master_mp.rst ) begin
50            avm_report_message("mem_driver" , "doing reset");
51            state = WAIT_FOR_REQ;
52            continue;
53          end
54
55          case( state )
56          WAIT_FOR_REQ : begin
57
58            if( request_port.try_get( request ) ) begin
59
60              avm_report_message( "Sending Request" ,
request.convert2string() );
61              write_request_to_bus( request );
62
63              state = WAIT_FOR_ACK;
64            end
```

```
65          end
66
67          WAIT_FOR_ACK : begin
68
69            pins_if.master_mp.req <= 0;
70
71            if( pins_if.master_mp.ack ) begin
72
73              response = read_response_from_bus();
74              state = WAIT_FOR_REQ;
75            end
76
77          end
78          endcase
file: topics/06_testbench_fundamentals/mem_svc/mem_driver.svh
```

该状态机有两个状态，WAIT_FOR_REQ和WAIT_FOR_ACK。WAIT_FOR_REQ用try_get来get下一个未解决的事务信息。既然这是一个不阻塞接口，（即try_get是一个函数）我们就可以保证不会消耗时间。如果这里有事务信息，我们将它传送给总线并改变它的状态。如果这里没有事务信息，我们就停在这个状态。WAIT_FOR_ACK只是简单地等着来自从机的询问并返回给WAIT_FOR_REQ。

复位的处理非常简单：无论我们在哪种状态，如果看到复位我们就返回到WAIT_FOR_ REQ。

最后，便捷方法write_req_to_bus实际上是发送请求给总线。我们使用这些便捷函数部分是使得状态机的设计更为容易，部分是因为我们也想在后面做错误注入。

```
84      virtual task write_request_to_bus( input request_t request );
85
86        pins_if.master_mp.req <= 1;
87        pins_if.master_mp.address <= request.m_address;
88
89        if( request.m_type == MEM_WRITE ) begin
90          pins_if.master_mp.rw = 1;
91          pins_if.master_mp.wr_data = request.m_wr_data;
92        end
93        else begin
94          pins_if.master_mp.rw = 0;
95        end
file: topics/06_testbench_fundamentals/mem_svc/mem_driver.svh
```

SystemC 实现细节

驱动器执行

SystemC的实现与SystemVerilog的实现多少有些相同。它在事务级也有一个

tlm_nonblocking _get_port，在RTL级有一个主机引脚接口。

```
34    class mem_master_if
35    {
36     public:
37      sc_in<bool> clk;
38      sc_in< bool > rst;
39
40      sc_out< ADDRESS_TYPE > address;
41      sc_out< DATA_TYPE > wr_data;
42      sc_in< DATA_TYPE > rd_data;
43      sc_out<bool> rw;
44      sc_out<bool> req;
45      sc_in<bool> ack;
46      sc_in<bool> err;
47    };
file: topics/06_testbench_fundamentals/mem_sc/mem_pin_if.h
```

在这个特例中，我们继承了引脚接口：

```
29    class mem_driver :
30      public sc_module ,
31      public mem_master_if
32    {
33     public:
34      sc_port< tlm_nonblocking_get_if< mem_request > > request_port;
35
36      SC_HAS_PROCESS( mem_driver );
37
38      mem_driver( sc_module_name );
file: topics/06_testbench_fundamentals/mem_sc/mem_driver.h
...
};
```

这里的状态机和便捷方法与SystemVerilog中的非常类似：

```
32    void mem_driver::run() {
33
34      if( rst ) {
35
36        cout << name() << " mem_driver : doing reset "
37             << " at " << sc_time_stamp() << endl;
38
39        m_state = WAIT_FOR_REQ;
40        return;
41      }
42
```

```
43      switch( m_state ) {
44      case WAIT_FOR_REQ :
45
46        if( request_port->nb_get( m_request ) ) {
47
48          cout << name() << " Sending Request " << m_request
49               << " at " << sc_time_stamp() << endl;
50
51          write_request_to_bus( m_request );
52
53          m_state = WAIT_FOR_ACK;
54        }
55        break;
56
57      case WAIT_FOR_ACK :
58
59        req = 0;
60
61        if( ack ) {
62
63          read_response_from_bus( m_response );
64          m_state = WAIT_FOR_REQ;
65        }
66
67        break;
68      }
69
70    }
71
72    void mem_driver::
73    write_request_to_bus( const mem_request &request ) {
74
75       req = 1;
76       address = request.m_address;
77
78       if( request.m_type == MEM_WRITE ) {
79         rw = 1;
80         wr_data = request.m_data;
81       }
82       else {
83         rw = 0;
84       }
85
86    }
file: topics/06_testbench_fundamentals/mem_sc/mem_driver.cc
```

测试平台中的双向通讯

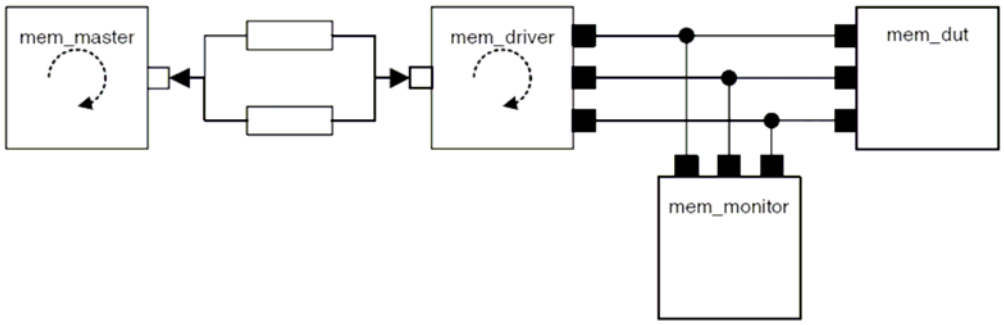


图6-4 测试中的双向通讯

说明

许多总线协议都包括请求和响应。主机发送请求然后等待从从机返回的响应。这个例子说明了在激励发生器和驱动器之间的双向通讯。激励发生器是定向的而不是像前面例子中是随机的，驱动器处理请求和响应。

主要概念

- 定向双向激励产生。
- 双向驱动器，它给总线发送请求，然后将响应返回给激励发生器。
- 在激励发生器中使用便捷层隐藏来自测试书写器的通讯细节。
- 用tlm_transport_if来模拟非流水线的双向通讯。
- 用tlm_transport_channel作为便捷方式取代两个独立的fifo。

SystemVerilog 实现细节

构造一个主机

```
20    class mem_bidirectional_stimulus
21      #( int ADDRESS_WIDTH = 8 ,
22         int DATA_WIDTH = 8 ) extends avm_named_component;
23
24      typedef mem_request #( ADDRESS_WIDTH , DATA_WIDTH ) request_t;
25      typedef mem_response #( DATA_WIDTH ) response_t;
26
27      typedef bit[ADDRESS_WIDTH - 1:0] address_t;
28      typedef bit[DATA_WIDTH-1:0] data_t;
29
30      tlm_transport_if #( request_t , response_t )  initiator_port;
31
32      function new( string name , avm_named_component parent = null );
33        super.new( name , parent );
34      endfunction
file:
topics/06_testbench_fundamentals/mem_svc/mem_bidirectional_stimulus.svh
```

上面勾画出mem_bidirectional_stimulus的外部连通性。实际上，这个激励发生器的外部连通性非常简单：它只有一个tlm_transport_if端口。这是一个任务，它将请求作为输入，将响应作为输出。既然传送是一个阻塞调用（例如一个任务），它需要花费一定的时间来执行全部的事务。

为了使这个测试易于书写，我们定义了一个便利层，它隐藏从测试书写器来的传输调用。

```
51      task write( input address_t address , input data_t data );
52
53        request_t request = new( address , MEM_WRITE , data );
54        response_t response;
55        string write_str;
56
57        $sformat( write_str , "%d %d" , address , data );
58
59        avm_report_message("about to do write" , write_str );
60        initiator_port.transport( request , response );
61        avm_report_message("just done write" , write_str );
62
63      endtask
64
65      task read( input address_t address , output data_t data );
66
67        request_t request = new( address , MEM_READ );
68        response_t response;
69        string read_str;
70
71        $sformat( read_str , "ADDR = %d" , address);
72        avm_report_message("about to do read", read_str );

73
74        initiator_port.transport( request , response );
75        data = response.m_rd_data;
76
77        $sformat( read_str , "%d %d" , address , data );
78        avm_report_message("just done read" , read_str );
79
80      endtask
file:
topics/06_testbench_fundamentals/mem_svc/mem_bidirectional_stimulus.svh
```

便利层可能是一套函数用来理解所讨论的协议。对这个特定的协议，我们所需要定义的是读和写，但是对于其它的协议，我们可能需要一套更大的包含象burst_read和burst_write的任务函数。

然后，我们就可以写测试本身了。

```
36      task generate_stimulus( int write_count = 16 , int read_count = 16
);
37
38        address_t address;
39        data_t    data;
40
41        for( address = 0; address < write_count; address++ ) begin
42          write( address , address + 17 );
43        end
44
45        for( address = 0; address < read_count; address++ ) begin
46          read( address , data );
47        end
48
49       endtask
file:
topics/06_testbench_fundamentals/mem_svc/mem_bidirectional_stimulus.svh
```

这个特定的测试进行一个确定数量的写操作，紧跟着对同一个存储器位置进行同样数量的读操作。

在一个大组织中，可能是有验证基础团队来写有传输端口和便利层的基类，而测试书写者则利用这个基类的一个或多个子类中的便利层来开发测试。

tlm_transport_channel 的内部操作

tlm_transport_channel包含一个请求和响应fifo。它输出这两个fifo的所有put，get和peek接口，以及主机接口，该接口将put请求与get和peek响应以及它们的镜像从机接口（它将get和peek请求与put响应连接起来）连接起来。

它与它的基类tlm_req_rsp_channel共享这些功能。使得它

tlm_transport_channel唯一的原因是它也能实现传送接口。

```
209 task transport( input REQ request , output RSP response );
211 this.m_request_fifo.put( request );
212 this.m_response_fifo.get( response );
213 endtask
```

当调用transport(req,rsp)时，它将请求放入put fifo并阻塞，然后等待get

响应。这个在fifo中的请求就被驱动器找回并应用到总线。过一会，驱动器将响应放入响应fifo。现在，传输方法可以不阻塞，从响应fifo中get响应，然后将它发送会激励发生器。

构造一个双向驱动器

阻塞驱动器需要在状态机的合适点从请求fifo获得请求，并将它们发送给总线。它也需要识别发生在总线上的响应，并将这些响应放入响应fifo中。

```
20    class mem_bidirectional_driver #( int ADDRESS_WIDTH = 8 ,
21                                      int DATA_WIDTH = 8 )
22      extends avm_verification_component;
23
24      typedef mem_request #( ADDRESS_WIDTH , DATA_WIDTH ) request_t;
25      typedef mem_response #( DATA_WIDTH )    response_t;
26
27      typedef enum { WAIT_FOR_REQ , WAIT_FOR_ACK } state_e;
28
29      virtual mem_pins_if #(
30        .ADDRESS_WIDTH( ADDRESS_WIDTH ) ,
31        .DATA_WIDTH( DATA_WIDTH )
32      ) pins_if;
33
34      tlm_nonblocking_get_if #( request_t ) request_port;
35      tlm_nonblocking_put_if #( response_t ) response_port;
36
37      function new( string nm , avm_named_component p = null );
38        super.new( nm , p );
39      endfunction
40    // ...
file:
topics/06_testbench_fundamentals/mem_svc/mem_bidirectional_driver.svh
```

阻塞驱动器的外部连通性如上面所示。引脚级接口，状态机和请求端口都与单向驱动器中的一样。唯一的区别就是我们现在加入了一个响应端口。修改后的状态机如下所示：

```
48        forever begin
49          @( posedge pins_if.master_mp.clk );
50
51          if( pins_if.master_mp.rst ) begin
52            avm_report_message("mem_driver" , "doing reset");
53            state = WAIT_FOR_REQ;
54            continue;
55          end
56
57          case( state )
58          WAIT_FOR_REQ : begin
```

```
59
60
61            if( request_port.try_get( request ) ) begin
62
63              avm_report_message( "Sending Request" ,
64                                  request.convert2string() );
65
66              write_request_to_bus( request );
67              state = WAIT_FOR_ACK;
68            end
69          end
70
71          WAIT_FOR_ACK : begin
72
73            pins_if.master_mp.req <= 0;
74
75            if( pins_if.master_mp.ack ) begin
76
77              response = read_response_from_bus();
78
79              if( !response_port.try_put( response ) ) begin
80                avm_report_error("mem_bidirectional_driver" ,
81                                 "Cannot put reponse" );
82              end
83
84              state = WAIT_FOR_REQ;
85            end
86
87          end
88          endcase
file:
topics/06_testbench_fundamentals/mem_svc/mem_bidirectional_driver.svh
```

在WAIT_FOR_ACK中，我们读出来自总线的响应并将它放入响应fifo中而不是抛弃它。如果响应fifo满了，我们就报告一个错误，因为激励发生器在没有获得响应时已经发送了一个请求。便捷方法与前面的例子中说明的一样。

SystemC 实现细节

通常，SystemC实例的构造和实现与SystemVerilog代码非常相似。

构造主机

双向激励发生器的头文件显示如下：

```
28    class mem_bidirectional_stimulus : public sc_module {
29     public:
30      sc_port<
31        tlm_transport_if< mem_request , mem_response >
32        > initiator_port;
33
34      mem_bidirectional_stimulus( sc_module_name );
35
36      void generate_stimulus( int write_count = 16 , int read_count = 16
);
37
38     private:
39      void read( const ADDRESS_TYPE & , DATA_TYPE & );
40      void write( const ADDRESS_TYPE & , const DATA_TYPE & );
41
42    };
file:
topics/06_testbench_fundamentals/mem_sc/mem_bidirectional_stimulus.h
```

sc_port定义了传输端口。Generate_stimulus是测试控制器开始测试的访问点。Read和write是便捷函数。

关于SystemC执行需要注意的一点是我们可以将便利层移到sc_port的子集。在TLM组件的例子中也是这么做的，参见*http://www.systemc.org*.

tlm_transport_channel 的内部操作

tlm_transport_channel在SystemC执行中的传输方法与SystemVerilog的类似：

```
RSP transport( const REQ &req )
{
    mutex.lock();

    request_fifo.put( req );
    rsp = response_fifo.get();

    mutex.unlock();
    return rsp;

  }
```

它像单一的元操作一样执行put和get。为了保证传输操作中的单元数，这个函数使用互斥体。互斥体防止另一个驱动器访问request_fifo或response_fifo，除非完成了put和get。

构造一个双向驱动器

```
30    class mem_bidirectional_driver :
31      public sc_module ,
32      public mem_master_if
33    {
34     public:
35      sc_port< tlm_nonblocking_get_if< mem_request > > request_port;
36      sc_port< tlm_nonblocking_put_if< mem_response > > response_port;
37
38      SC_HAS_PROCESS( mem_bidirectional_driver );
39
40      mem_bidirectional_driver( sc_module_name );
41
42     private:
43
44      enum state_e {
45        WAIT_FOR_REQ , WAIT_FOR_ACK
46      };
47
48      void run();
49
50      state_e m_state;
51      mem_request m_request;
52      mem_response m_response;
53
54      void write_request_to_bus( const mem_request & );
55      void read_response_from_bus( mem_response & );
56
57    };
file: topics/06_testbench_fundamentals/mem_sc/mem_bidirectional_driver.h
```

SystemC实现也有一个用于请求的阻塞get端口和用于响应的阻塞put端口。其状态机与SystemVerilog中的非常类似。运行nb_get来get响应，运行nb_put来发送回请求：

```
33    void mem_bidirectional_driver::run() {
34
35      if( rst ) {
36
37        cout << name() << " mem_driver : doing reset "
38             << " at " << sc_time_stamp() << endl;
39
40        m_state = WAIT_FOR_REQ;
41        return;
42      }
43
44      switch( m_state ) {
45      case WAIT_FOR_REQ :
46
47        if( request_port->nb_get( m_request ) ) {
48
```

```
49          cout << name() << " Sending Request " << m_request
50               << " at " << sc_time_stamp() << endl;
51
52          write_request_to_bus( m_request );
53
54          m_state = WAIT_FOR_ACK;
55        }
56        break;
57
58      case WAIT_FOR_ACK :
59
60        req = 0;
61
62        if( ack ) {
63
64          read_response_from_bus( m_response );
65
66          if( !response_port->nb_put( m_response ) ) {
67            cout << name() << " Cannot put response  " << m_response
68                 << " at " << sc_time_stamp() << endl;
69
70          }
71
72          m_state = WAIT_FOR_REQ;
73        }
74
75        break;
76      }
77
78    }
file: topics/06_testbench_fundamentals/mem_sc/mem_bidirectional_driver.cc
}
```

第 7 章 完成测试

我们已经知道如何构造TLM组件以及如何构造驱动器和监视器。真正的测试需要更多。在这一章里，我们把事情放在一起构造更有现实意义的测试。构造一个可用于产品的测试平台，我们需要分析组件（组件的任务是分析在DUT中发生的事情）、与分析组件通讯的方式和整合所有操作的控制器。

至今为止我们讨论过的测试结构是用于测试平台的工作域。这些通过采用激励和响应来控制DUT的操作。为了使得操作有用，我们需要提供测试组件来分析在分析域发生了什么以便回答两个问题：它有用吗？和我们做完了吗？这些组件属于测试平台的分析域。

记分板是测试组件，它的任务就是回答这个问题：它有用吗？记分板监视DUT的激励和响应并进行必要的计算来决定这个激励和响应是否正确。

覆盖率采集器也是分析组件。它们回答问题：做好了吗？覆盖率采集器通过计算传入和发出DUT的激励和响应的数量来计算覆盖率。

记分板

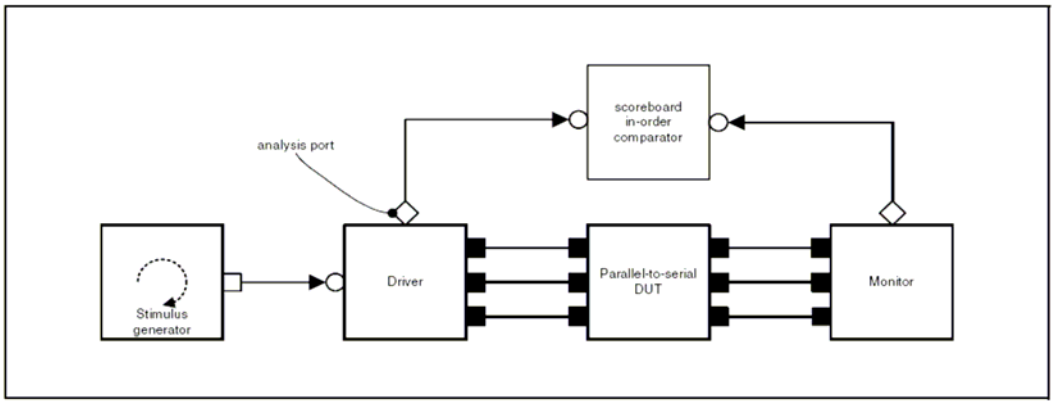


图7-1 有记分板的测试

说明

记分板是验证组件，它的主要任务就是确定DUT是否工作正常。记分板好，可以记分。然而， 一个黄金模型必须完全模仿DUT的操作，记分板则通过收集必要的数据来计算指标或记录行为。例如，在一个基于包的通讯系统，记分板可用来将输出包与输入包匹配。记分板回答问题“它能工作吗？”

并到串的设计，即所谓的p2s DUT，取并行字并将它转化为一系列的比特流。激励发生器发出字流给驱动器，驱动器则将它们送入p2s DUT。 驱动器也通过分析端口给记分板提供字流。与输出比特流相连的比特位监视器重构比特流成字流。顺序比较器将来自于驱动器的输入字流与来自于监视器的输出字流相比较。如果一切都工作正确，这两个字流就应该相同。

主要概念

- 事务处理器（驱动器和监视器）利用分析端口与记分板通讯。
- 记分板是一个不计时的事务级组件，它作为一个参考使用。它负责通过检查输入和输出信息流来确定DUT是否工作正常。

分析端口

在测试过程中，我们需要捕捉DUT，事务处理器，和操作域中的环境对象的行为，以便分析它。我们不希望中断或打扰DUT或环境组件的操作，我们希望确认在数据产生时就能收集到它们。分析端口就是实现这一点的途径。

分析端口像一个发行对象一样组织，有一组签约用户对象。用户对象将自己向发行对象注册。当发行对象有新数据要发行时，它通知所有用户对象。

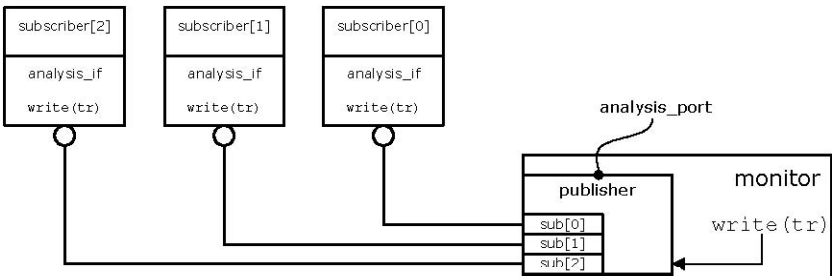


图7-2 分析端口组织

在测试进行前，每个用户对象必须在发行对象处注册，发行对象则保留一个用户对象的清单。有时在它的操作过程中，拥有分析端口的设备调用write()，传给事务对象。分析端口向前给每个用户传送write调用，传递事务对象的副本。

当调用时write()，其接口功能一定不能被堵塞。如果write()被堵塞，它就会干扰监视器的操作。而且分析设备在write()调用的同一个delta周期内接收数据也非常重要。分析fifo确保支持这些性能。一个分析fifo就是一个TLM fifo，它的大小是无限的，且有一个write()接口，而不是通常的put和get接口。通过使得分析端口的用户成为一个极大的fifo，我们就可以确保它绝对不会在单个的delta周期被占满或堵塞，且分析组件能够访问同一个delta周期内发送的所有事务。

分析端口的典型用法，就如我们在本例中看到的，就是从监视器到记分板之间的通讯。监视器为了实现完整的事务，发送事务对象给任何一个通过分析端口与它相连的分析组件。

记分板

记分板的作用就是确定DUT是否功能正确。记分板收集形成DUT输入和输出的事务流。这样它就可以计算来自输入的输出并确定它计算得到的输出与从DUT收到的事务输出是否一致。

本例中的记分板是一个顺序比较器。它接受两条事务信息流，前面的和后面的。对每一个来自后面的事务信息流，它从前面的信息流中取出一条进行比较来看它们是否相等。之所以它被称作顺序比较器，就是因为它假定两条信息流的事务是以同样的顺序比较的。

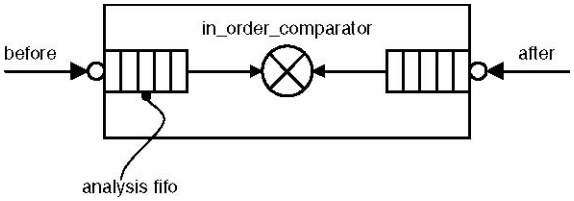


图7-3 顺序比较器的组织

顺序比较器在每一个输入上都有一个分析fifo。等到后fifo里出现事务后比较器开始工作。当它开始比较时，他从前fifo获得下一个事务。SystemC使用operator==()来比较这两个事务（当然，它们应该是同一类型的）。SystemVerilog则使用比较法则，可以是内嵌的（内嵌的类型如float和int），或者是类比较器（用户自定义的类对象）。

SystemVerilog 基于类的实现细节

在本章中，分析端口和顺序比较器都是新内容，因此我们集中在这些组件的实现细节上。

分析端口

在本节，将看看在监视器中是如何使用分析端口的，以及它是如何与记分板相连的，在这种情况下就是顺序比较器。

```
23      analysis_port #( p2s_transaction ) ap;
24      virtual p2s_pins_if #( .DATA_SIZE( 1 ) ) m_bus_if;
25
26      function new( string name , avm_named_component parent = null );
27        super.new( name , parent );
28        ap = new;
29      endfunction
file:
topics/07_complete_testbenches/p2s_transactors_svc/p2s_bit_monitor.svh
```

这个代码片断表明这两个界面在位监视器上的声明以及它的构造器。第一个声明有一个p2s_transaction型的分析端口。也就是说这个分析端口可用来发送类型为p2s_transaction的对象。第二个声明是一个虚拟接口，它与包含该设备监视的引脚的真实接口绑定。构造器用new操作符来创建一个分析端口的实例。

```
53            if( byte_index == 8 ) begin
54              transaction = new;
55
56              transaction.data = b;
57
58              ap.write( transaction );
59
60            avm_report_message("seen byte" , transaction.convert2string
);
61              byte_index = 0;
62            end
file:
topics/07_complete_testbenches/p2s_transactors_svc/p2s_bit_monitor.svh
```

如上面的片断所示，监视器的核心部分把比特数据收集成有效的字节，将该字节放入一个新构造的事务对象中，并通过分析端口将它发送到顺序比较器。

顺序比较器

顺序比较器是AVM库中的标准组件，它是在假定两条事务信息处在同样顺序的前提下对它们进行比较。它是一个有几个参数的参数化组件。第一个参数（在大部分情况下也是你需要提供的唯一参数）就是T，即要比较的对象的类型。另外的三个参数：comp，convert和 pair_type都是policy 类，它们给in_order_comparator提供额外的功能。Comp定义比较是如何进行的；convert则把类型为T的对象转换为可打印的字符串；pair_type则定义一对包含两个截然不同的T类型的实例。对大部分的应用软件，你可以使用缺省方针，就如则本例中所示：

```
42    class avm_in_order_comparator
43      #( type T = int ,
44         type comp = avm_built_in_comp #( T ) ,
45         type convert = avm_built_in_converter #( T ) ,
46         type pair_type = avm_built_in_pair #( T ) )
47        extends avm_named_component;
48
49      //  The two exports. Actually, there are no assumptions made about
50      // ordering, so it doesn't matter which way around you make the
51      // connections
52
53      analysis_if #( T ) before_export , after_export;
54
55      analysis_port #( pair_type ) pair_ap;
56
57      local analysis_fifo #( T ) before_fifo , after_fifo;
file: utilities/systemverilog/avm/utils/avm_in_order_comparator.svh
```

在组件头的后面就是用在这个组件里的主要对象的声明。Before_export和after_export是分析接口，是从analysis_fifo通道到外部世界的联接。Before_fifo 和 after_fifo 是分析 fifos，用来缓冲进来的信息流。Pair_ap是一个分析端口用来发出被比较过的每一对对象。比较器从

after_fifo和before_fifo中各拉出一个对象构造成一对。然后这对对象通过pair_apanalysis端口被送到下游组件。拥有成对的信息流对下游组件访问成对对象以便进一步分析或显示是很有用的。例如，如果对对象中的元件不相等，你可能希望打印一条信息或进行进一步的分析来确定哪个特定的成员是不等的。

SystemVerilog 基于模块的实现细节

在基于本例的变种模块中，组件－DUT，驱动器，监视器等都被构造成模块。通道仍然采用类来实现。

分析fifo是从avm_verification_component派生的，因此，当调用构造器时，我们需要给每一个提供名称。

```
59      analysis_fifo #( p2s_transaction ) before_fifo =
new("before_fifo");
60      analysis_fifo #( p2s_transaction ) after_fifo = new("after_fifo");
61
62      analysis_fifo #(
63        avm_class_pair #( p2s_transaction )
64      ) pair_fifo = null;
65
66      tlm_fifo #( p2s_transaction ) request_fifo = new("request_fifo");
file: topics/07_complete_testbenches/01_scoreboard_svm/top.sv
```

模块实例采用熟悉的Verilog语法。SystemVerilog使我们将类指针放入一个端口列表。这样就提供了模块和通道之间的连接，它就是类。

```
69      p2s_stimulus_mod stimulus( .request_fifo( request_fifo ) );
70
71      p2s_driver_mod driver( .request_fifo( request_fifo ) ,
72                             .master_mp( input_bus.master_mp ) );
73
74
75      p2s_bit_monitor_mod bit_monitor( .af( after_fifo ) ,
76                                       .monitor_mp(
output_bus.monitor_mp ) );
77
78      avm_in_order_class_comparator_module #( p2s_transaction )
79        comp( .before_fifo( before_fifo ) ,
80              .after_fifo( after_fifo ) ,
81              .pair_fifo( pair_fifo ) );
file: topics/07_complete_testbenches/01_scoreboard_svm/top.sv
```

本例中，我们没有使用固有的分析端口，而只是使用的分析fifo。当我们知道这里只有一个用户时就可以这样做。对象avm_analysis_port是基于观测器模板的，面向对象的设计模板允许发行对象通知和传送数据给多用户。在只

有一个用户的情况下，额外的机制用于注册用户，并且当有新数据等待发布时单独通知它们是不必要的。

相反，我们只是使用分析fifo，它是极大的且具有和分析端口一样的write()接口。我们在组件间连接分析fifo，正如我们在其它通道间连接一样。在进行连接时，有一点需要稍加注意：

```
57  if( af != null ) af.write( transaction );
```

在给分析fifo写入时，我们需要确认有个分析fifo确实被连接到我们的模块中。在这个任务中完成了简单的查看分析fifo指针是否为0。

SystemC 实现细节

分析端口

SystemC中的分析端口和SystemVerilog中的分析端口实现同样的功能，但它们的实现有一小点区别。Analysis_port<T>是带有T = analysis_if的特殊的sc_port<T>。Analysis_if是一个简单的纯虚拟接口类，它包括一个函数write()：

```
33      template < typename T >
34      class analysis_if : public virtual sc_interface
35      {
36       public:
37         virtual void write( const T & ) = 0;
38      };
```

特殊的对象sc_port包含一个分析代理列表，这些对象也继承了analysis_if，并提供执行write()。当你绑定到一个分析端口时，它生成一个新的代理对象并将它加入列表中。这取代了SystemVerilog中分析端口需要的明确的注册功能。

当你调用分析端口上的write()时，它将这个调用向前传送给它列表中的每个代理。每个代理代表一个用户，调用和它绑定的接口上的write()。结果就是同一个观测模板被用于SystemVerilog上。相反，分析端口使sc_port<T>和超载bind()专用于在每次调用代理时给它的列表增加入口，而不是一个专门的注册函数。用户使用通常的绑定语法来连接分析端口，而不是一个专门的注

册函数。

顺序比较器

顺序比较器，就如SystemVerilog中的配对物一样，有两个分析fifo：一个前fifo，一个后fifo。它从前fifo和后fifo中各拉出一条信息，用operator==()对它们进行比较。如果比较器的对象类型是标量，int float，等，其缺省operator==就可以。如果比较器类型是类，包含了除标量以外的事件（包括指针，矢量等），我们建议你提供一个明确的operator==值，而不要依赖于编译器提供的缺省值。

控制器和激励发生器

在这些例子的SystemC版本中，我们使用独立的控制器组件。控制器拥有控制整个测试平台的主thread。在本例中，它启动和停止激励发生器。控制器是一个简单设备，它有一个thread和一个start_stop_port。start_stop_port是绑定在激励发生器上的，提供启动和停止激励发生器的方式。

```
12    class controller : public sc_module
13    {

14     public:
15      controller(sc_module_name nm) :
16        sc_module(nm)
17      {
18        SC_THREAD(main_thread);
19      }
20      SC_HAS_PROCESS(controller);
21
22      sc_port<start_stop_config_if> start_stop_port;
23
24     private:
25      void main_thread();
26    };
file: topics/07_complete_testbenches/01_scoreboard_sc/controller.h
```

控制器事务类start()使得激励发生器开始工作，等1000ns后，最后调用stop()，使得激励发生器停止操作。

```
25    void controller::main_thread()
26    {
27      start_stop_port->start();
28      wait(1000, SC_NS);
29      start_stop_port->stop();
30    }
file: topics/07_complete_testbenches/01_scoreboard_sc/controller.cc
```

start_stop_config_if是一个纯虚拟接口，它规定启动和停止函数。控制器调用这些函数，激励发生器提供执行。

```
27    class start_stop_config_if : public virtual sc_interface
28    {
29    public:
30      virtual void start() = 0;
31      virtual void stop() = 0;
32    };
file: utilities/systemc/avm/avm_config/start_stop_if.h
```

激励发生器与外界有两个连接：一是start_stop_export，它将被绑定到控制器上的start_stop_port，另一个是事务put端口p，put端口被用来将事务发送到驱动器。

```
33    class random_stimulus :
34      public sc_module,
35      public start_stop_config_if
36    {
37     public:
38      random_stimulus(sc_module_name);
39
40      sc_port< tlm_blocking_put_if<word_t> > p;
41
42      sc_export<start_stop_config_if> start_stop_export;
43
44      SC_HAS_PROCESS(random_stimulus);
45


46     private:
47      void run();
48      void start();
49      void stop();
50
51      bool go;
52      sc_event start_event;
53    };
file: topics/07_complete_testbenches/p2s_sc/random_stimulus.h
```

注意激励发生器有一个称作start_event的私有事件。该函数使用SystemC thread并且在时间为0时立即开始。我们可能希望激励发生器在这个时候开始，也可能不希望这样。我们希望激励发生器在start和stop函数的控制下。

```
64      cout << sc_time_stamp() << ": waiting for start event..." << endl;
65      wait(start_event);
66
67      while(go)
68      {
69        w->next();
70        word_t val = *w.get_value();
71        p->put(val);
72        cout << sc_time_stamp() << ": " << name()
73             << " done sent : " << setw(2) << hex << val << endl;
74      }
file: topics/07_complete_testbenches/p2s_sc/random_stimulus.cc
```

程序开始时，它首先等待激发start_event，start()函数调用start_event.notify()，该函数释放等待并开始激励发生循环。 Stop()则在它完成当前迭代后停止计算字节，它又会停止激励发生器

覆盖率

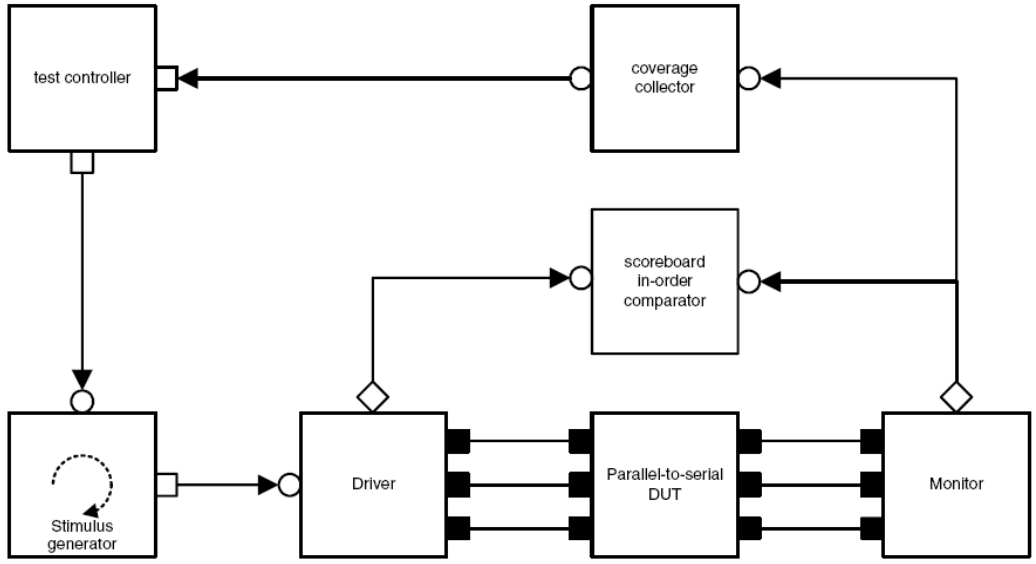


图7-4 测试中的覆盖率

说明

在移向真实测试平台过程中的这一步介绍覆盖率和覆盖率反馈。在前面的实例中，我们看了一个小的测试控制器。这里，我们要给他更大的力量。

覆盖率采集器是通过分析端口与监视器相连的，就如记分板一样。有监视器产生的每一个事务都被发送到覆盖率采集器以及记分板。覆盖率采集器计算它所看到的事务。覆盖率采集器的目的就是回答这个问题“做好了吗？”的。

主要概念

- 覆盖率是回答“我们做好了吗？”这个问题的一种方式。覆盖率采集器分析来自事务器的数据，产生能回答这个问题的信息。
- 激励发生器有一个配置接口，用来启动和停止它的操作。

- 测试控制器是测试中的决策者。测试控制器启动和停止模拟器并利用来自覆盖率采集器的信息来决定要做什么。

覆盖率和覆盖率采集器

覆盖率后面的关键问题就是计算在有限的空间内各种事件的发生。独特事件与一定范围内基本数量的比率就是覆盖率。也就是说，比如，我们希望覆盖一个地址空间，为了计算覆盖率，我们需要一个在每次访问一个以前还没有访问过的地址时增加计数的方式。测试结束时，我们通过将地址空间划分到计数器来计算覆盖率。如果地址空间有255个独立的地址，而我们在测试过程种用到过其中的147个，那么覆盖率就是147/256，大约是68％。

覆盖率采集器就是存储计数器计算覆盖率的途径。它们所计算的内容是有测试计划决定的，你想知道什么决定了你是否做了足够的测试。如果你是在验证一个微处理器，你可能想要计算指令。如果你在验证一台路由器，你可能想知道什么时候用至少一个包会走完所有可能的路径。等等。获得新数据进行计算的行为就称作取样。

除了计算事件外，你可能想要通过目录或bin计算事件，这就叫binnng。为了执行binning，你需要创建多个计数器，每个bin一个。在模拟一个函数的过程中，每次采样后要确定应当增加哪个bin。

SystemVerilog 实现细节

覆盖率采集器使用内嵌在SystemVerilog语言中的覆盖功能。覆盖点和覆盖群都是SystemVerilog的功能，设计用来收集和事务覆盖数据。在我们的覆盖率采集器中心的覆盖群被称作byte_cov

```
24 covergroup byte_cov;
25 top_four : coverpoint data[7:4];
26 bottom_four : coverpoint data[3:0];
```

```
27 endgroup
```

file:

topics/07_complete_testbenches/p2s_transactors_svc/p2s_byte_coverage.svh

它有两个覆盖点，top_four和bottom_four。每一个都是一个4位的半字节。覆盖点占据一个字节的顶上的4位，bottom_four而则占据下面的4位。每个覆盖点就是一个bin。当取样了覆盖群时，样本数据就被binning入这两个bin中。

```
29 function new( string name , avm_named_component 指针 = null );
30 super.new( name , 指针 );
31
32 byte_cov = new;
33 m_is_covered = 0;
34 endfunction
```

file:

topics/07_complete_testbenches/p2s_transactors_svc/p2s_byte_coverage.svh

构造器用new操作符产生一个覆盖群实例。

监视器使用分析端口将信息传送到覆盖率采集器。分析端口会调用每个用户的write()，并传送给事务器。就如用户对分析端口一样，覆盖率采集器必须提供一个write()的执行。这个覆盖率采集器的write()函数做如下三件事情：

1. 它将事务器中的数据复制到覆盖群知道的数据对象。
2. 它调用样本
3. 它计算覆盖率是否达到了完成的极限值。

如果达到了覆盖率极限值，则将m_is_covered置为1。

```
36 function void write( input p2s_transaction t );
37 data = t.data;
38 byte_cov.sample;
```

```
39
40 if( byte_cov.get_inst_coverage > 95 ) begin
41 m_is_covered = 1;
42 end
43 endfunction
```

file:
topics/07_complete_testbenches/p2s_transactors_svc/p2s_byte_coverage.svh

通过观看m_is_covered，控制器知道什么时候达到了覆盖率极限值。

SystemVerilog 基于模块的实现细节

覆盖率采集器基于模块的执行采用覆盖群和覆盖点，就如基于类的执行一样。

```
32 covergroup byte_cov;
33 top_four : coverpoint data[7:4];
34 bottom_four : coverpoint data[3:0];
35 endgroup
```

file:
topics/07_complete_testbenches/p2s_transactors_svm/p2s_byte_coverage_mod.
sv

覆盖率采集器的主循环运行稍微有点不同。循环不是直接执行函数write()，而是从一个分析fifo的af.读取。

```
39 initial begin
40 m_byte_cov = new;
41 m_is_covered = 0;
```

```
42
43 forever begin
44 af.get( t );
45
46 data = t.data;
47 m_byte_cov.sample;
48
49 if( m_byte_cov.get_inst_coverage > 95 ) begin
50 m_is_covered = 1;
51 end
52 end
53 end
```

file:
topics/07_complete_testbenches/p2s_transactors_svm/p2s_byte_coverage_mod.
sv

调用af.get(t)被堵塞。也就是说，如果在分析fifo中没有什么可以获得，它将等到有为止。一旦fifo中出现了一个项目，循环的其余部分执行相同的计算，就像它的基于类的副本一样。

测试平台是由在模块top的初始化模块里一对交叉的进程控制的。

```
101 fork
102 begin
103 stimulus_process = process::self;
104 stimulus.generate_stimulus;
105 end
106 terminate_when_covered;
107 join
108
```

```
109 $finish;
```

file: topics/07_complete_testbenches/02_coverage_svm/top.sv

第一个事务等于begin/end。它首先请求句柄，然后启动激励发生器运行。激励发生器是自由运行：它不会自动停止。因此，第一个事务会一直运行，直到被明确停止。第二个事务，终止，会等待直到覆盖率达到预先确定的极限值然后停止激励发生器。

```
113 task terminate_when_covered;
114 wait( byte_coverage.m_is_covered );
115 stimulus_process.kill;
116 avm_report_warning(“terminate_when_covered” , “ “ );
117 endtask
```

file: topics/07_complete_testbenches/02_coverage_svm/top.sv

语句stimulus_process.kill用精密的程序控制来停止激励发生器。终止进程的调用句柄是stimulus_process，就是在fork/join模块中首先产生的两个进程。当在覆盖率采集器中的m_is_covered改变状态时，终止任务知道已经达到了覆盖率收集极限。

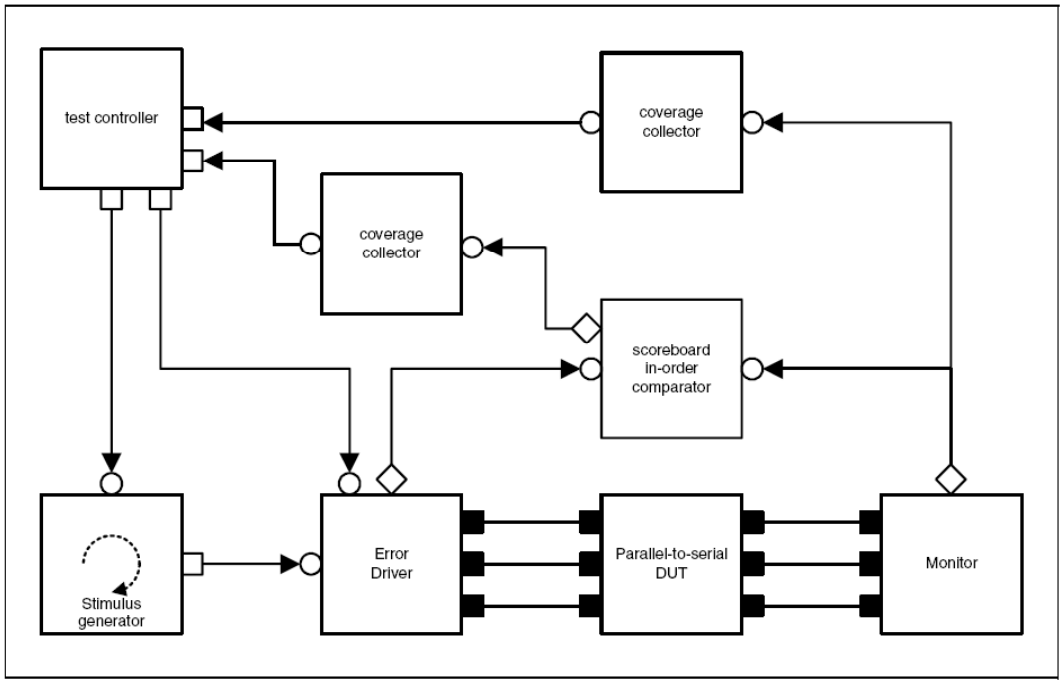
SystemC 实现细节

在SystemC中，覆盖率采集器时基于AVM体统的覆盖点<T,N>类。覆盖点模板有两个判据。第一个T是要覆盖的对象的类型，第二个N是在覆盖点中产生的bin的数量。

```
12    class word_coverpoint : public coverpoint<word_t, 4>
13    {
14     public:
15      word_coverpoint(sc_module_name nm) :
16        coverpoint<word_t, 4>(nm)
17      {}
18
19     private:
20      int sort(const word_t& w) const
21      {
22        // return low order 2 bits
23        return  (w & 0x3);
24      }
25    };
file: topics/07_complete_testbenches/p2s_sc/word_coverage.h
```

覆盖点是通过分析端口连接的，因此必须提供一个write()函数。Write函数采用类型为T的对象，并调用其类别来决定它们属于哪个bin。分类函数的判据也必须是T型，它返回一个在0和N－1的整数，表明哪个bin要增加。

产生错误



带错误驱动器测试

说明

适当验证一个设计的一部分就可以说明它是否正确的处理错误。测试必须在适当的时间和地点加入错误，并确定这个错误是否被DUT按照预期的方式进行处理。

在本例中，我们采用了两个覆盖率采集器：一个计算字数，就如前面例子中说明的，另一个计算错误。当错误驱动器注入一个错误，它就将正确的数据发送到记分板，而错误的数据则发送到DUT。当错误的数据和正确的数据进行比较时，记分板就登记一个不成功的比较。当发生这种事情时，记分板通知计算错误的错误覆盖率采集器。

主要概念

- 错误驱动器是通过继承（或扩展）无错误驱动器构成的。
- 修正测试控制器给控制如何和何时产生错误的错误驱动器设置参数。
- 既然错误驱动器会给DUT发送坏数据，DUT就可能以一定的方式响应来表明它认出了和/或处理了这个错误。记分板必须知道这一点并与收到错误的测试控制器通讯。
- 测试控制器对两个覆盖率采集器做出响应，一个计算事务，另一个计算错误。

构造错误驱动器

构造错误驱动器的基本技术就是继承无错误驱动器和再从新实现某一个虚拟方法，这样它们就可以注入错误。使用继承避免给协议状态机重写复杂代码。另外，错误驱动器输出一个配置接口，该接口给错误注入提供外部控制。

考虑一个典型驱动器的伪代码：

```
do forever
{
transaction = fifo.get();
```

```
send_to_bus(transaction);
}
```

驱动器进行无限循环，从输入fifo中拉出事务。处理每个事务来运行引起引脚改变的协议状态机。这个有点简单，在真正的协议中，FSM对输入引脚的改变也很敏感，需要做出相应的响应。尽管如此，这个简单的驱动器结构能用来说明从无错误驱动器创建错误驱动器的自然方式。一个办法就是插入一个函数，在调用send_to_bus()之前通知事务。

```
do forever
{
transaction = fifo.get();
inject_error(transaction);
send_to_bus(transaction);
}
```

inject_error是虚拟函数。在无错误驱动器中，它什么也不做；在错误驱动器中，它被重载以便在一些错误注入途中改变事务。问题就在于对于无错误应用，是不使用注错函数的，它就有点过头，混乱了驱动器代码。

另一种错误注入的解决办法就是将编码错误修改到事务对象本身。

```
do forever

{

transaction = fifo.get();

transaction.inject_error();

send_to_bus(transaction);

}
```

用这个办法的缺点就是它要求驱动器的主循环在错误驱动器中被重载。就拿本例中这个简单的循环来说，这并不是一个真正的结果。在更为复杂的情况下，它就会成为大问题：需要复制代码，包含细微差别、缺陷和特异性。这会降低驱动器的生存能力。没有办法辨别，在“干净的”仿真和有错的仿真之间的区别是因为注入了错误还是由于驱动器本身在操作过程中的细微变化。

第三种办法，也是在本例中我们选择的办法，就是重载send_to_bus()。派生的错误驱动器中的send_to_bus版本在准备给总线发送信息时会修改数据。为了确保保留了send_to_bus的语义并避免重复编码一个潜在的复杂函数，send_to_bus的错误版本调用父类中的同一函数。在SystemVerilog中，它是通过使用超级引用来完成的，在SystemC中，则是使用完整的有条件的名称（带有:: operator）绑定到正确的函数。

SystemVerilog 基于类的实现细节

在无错误驱动器中，send_transaction_to_bus函数做了繁重的提升工作。它将单个的事务转换为一系列引脚信号的变动。在错误驱动器中也是这样。错误驱动器用一些额外的信息来确定什么时候注入错误。Error_length规定了在错误之间事务的数量。Error_count则记录在上次错误发生以来事务的数量。当Error_count等于error_length时，就该注入另一个错误了。

```
40      virtual task send_transaction_to_bus(
41        input p2s_transaction transaction
```

```
42      );
43
44        string error_str;
45        p2s_transaction temp_transaction = transaction.clone;
46
47        if( m_error_length != 0 ) begin
48          m_error_count++;
49
50          if( m_error_count == m_error_length ) begin
51
52            temp_transaction.data[0] = !temp_transaction.data[0];
53
54            $sformat( error_str ,"%s -> %s" ,
55                      transaction.convert2string ,
56                      temp_transaction.convert2string );
57
58            avm_report_warning("error injection" , error_str );
59            m_error_count = 0;
60          end
61        end
62
63        super.send_transaction_to_bus( temp_transaction );
64      endtask
file:
topics/07_complete_testbenches/p2s_transactors_svc/p2s_error_driver.svh
```

注意任务send_transaction_to_bus是虚拟的。如果需要，它可以被下一个错误驱动器进一步重载。

该函数的第一部分是询问是不是到了注入错误的时间。如果是，它就对发送的字进行简单的修改，反转它。

在任务的最后，调用send_transaction_to_bus的父类版本。这将提供将事务转变为引脚行为的功能。通过调用父类中的这个任务，我们就可以避免在无驱动器中重新编码这个函数。

SystemVerilog 基于模块的实现细节

在基于模块的形式中是不能使用继承的，至少不能使用在类中的那种继承。我们希望复用错误驱动器来避免重新编码。这可以通过在错误驱动器中例化无错误驱动器并增加一些修改事务激励的片断来完成。

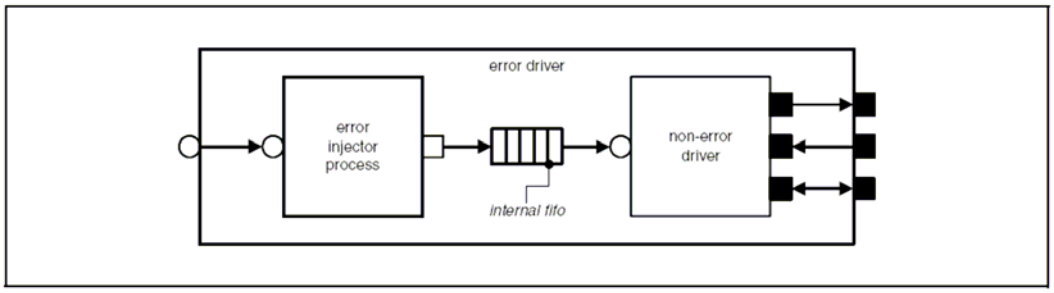


图7-6 从无错误驱动器构造的错误驱动器

这是一种静态的继承形式。我们通过使用HAS-A关联取代IS-A关联，继承了无错误驱动器的性能来构造错误驱动器。

在驱动器前端的前面有一个附加的进程，它从输入get事务，并改变它们的内容：

```
35      initial begin
36        m_error_count = 0;
37        set_error_length( 0 );
38
39        forever begin
40          request_fifo.get( transaction );
41          temp_transaction = transaction.clone();
42
43          if( m_error_length != 0 ) begin
44            m_error_count++;
45
46            if( m_error_count == m_error_length ) begin
47
48              temp_transaction.data[0] = !temp_transaction.data[0];
49
50              $sformat( error_str ,"%s -> %s" ,
51              transaction.convert2string ,
52              temp_transaction.convert2string );
53
54              avm_report_warning("error injection" , error_str );
55              m_error_count = 0;
56            end
57          end
58
59          internal_fifo.put( temp_transaction );
60        end
61      end
file:
topics/07_complete_testbenches/p2s_transactors_svm/p2s_error_driver_mod.s
v
```

初始化模块的结构与我们在基于类的实现中的任务 send_transaction_to_bus 很像。其主要区别就在于在无限循环的最后调用 internal_fifo.put()。这个调用发送修改过的事务给下级无错误驱动器。

SystemC 实现细节

在本质上，SystemC实现情况与SystemVerilog基于类的实现情况很像，当然，有一些句法差别。

```
28    class error_word_driver :
29      public word_driver,
30      public error_word_driver_config_if
31    {
32     public:
33      error_word_driver(sc_module_name nm, int b = 1, int w = 4, int e =
0) :
34        word_driver(nm, b, w),
35        m_error_length(e),
36        m_error_count(0)
37       {}
38
39       void set_error_length(int);
40       void get_error_length(int&);
41
42     private:
43       virtual void send_transaction_to_bus(word_t t);
44
45       int m_error_length;
46       int m_error_count;
47    };
file: topics/07_complete_testbenches/p2s_sc/error_word_driver.h
```

错误驱动器声明从无错误驱动器的word_driver继承。它也从error_word_ driver_ config_if继承，这包括声明错误控制函数set_error_length()和get_ error_length()。错误驱动器也给虚函数send_transaction_to_bus()提供执行。

```
41    void error_word_driver::send_transaction_to_bus(word_t t)
42    {
43      if(m_error_length != 0)
44      {
45        m_error_count++;
46
47        if(m_error_count == m_error_length)
48        {
49          word_t u = t;
50          t = ~t;
51
52          cout << sc_time_stamp() << ": " << name()
53                << ": injecting error: changed from "
54                << u << " to " << t << endl;
55
56          m_error_count = 0;
57        }
58      }
59
60      word_driver::send_transaction_to_bus(t);
```

确切地说，当send_transaction_to_bus注入一个错误，而这个错误是有错误

长度控制的，错误长度由set_error_length设置。Error_length是错误之间事务的数量。而Error_count则是上次发生错误以来事务的数量。当Error_count等于error_ length时，就该注入一个新错误。

无论一个错误是否是注入的，循环的最终行为就是调用word_driver::send_transaction_to_bus。Word_driver是error_word_driver的父类（即，error_word_driver是从这个类派生的）。send_transaction_to_bus的父类版本取得事务，并将它转换成引脚级行为。通过复用父 类中的函数，我们避免重新编码，这会导致语义中的潜在的（和无意中的）细微差别。

第 8 章　逐步替换

逐步替换是一个使用高级设计的过程，而且通过各种转换和替代，进入具体的实现。在一个从上到下的流程中，设计的第一个具体化就是在事务级构造。TLM使得设计师理解驱动器的一般属性，如性能和产量。设备将有一个测试平台来捕捉和分析必要的数据以便确定这些性能。

在一个大的设计中，经常所有RTL并不能一次完成。所以希望用可以获得的RTL组件替代TLM部分。进行这种替代要求插入事务器来连接事务级组件和管脚级组件。

TLM的测试会含有测试中的“你做好了吗？”方面，但是没有“它有用吗？”方面。原因就在于这里没有另一个执行来进行比较。要通过采用除了比较结果和RTL模型以外的方式来确定它是否有用。

在事务级产生激励发生器（或主机和从机）是一个非凡的工作，所以无论在什么可能的程度复用它们是非常重要的。在这一系列的实例中，我们将展示如何做这些事情。我们将展示在从TLM移向RTL执行时，如何复用一个激励发生器。我们也将展示如何复用TLM作为一个golden的模板，并在运行时将它的结果与RTL模板的结果进行比较。

事务级 FPU

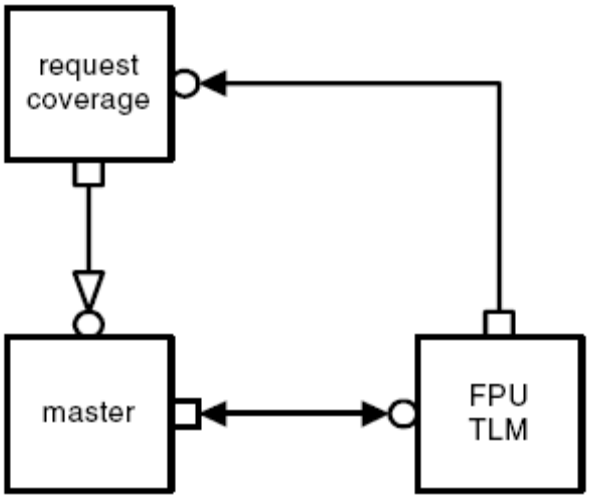


图8-1 测试事务级FPU

说明

浮点单元(FPU)的事务级模板非常简单。它从主机接收事务，包括操作数，操作符和圆整模式。TLM使用这些信息进行计算。FPU用公式表示一个有计算的答案的响应，并将答案返回给主机。

在这个设计中，覆盖率就是请求。当看到某种类型的大量请求时，覆盖率门限就算达到。

主要概念

- 全部设计都在事务级进行了模块化。
- 主机产生事务信息流，它驱动DUT操作，在本例中时FPU TLM。
- 覆盖率采集器累计来自DUT的覆盖率信息。
- 覆盖率采集器形成反馈，当达到覆盖率门限值时停止主机。

SystemVerilog 实现细节

除了说明从TLM到RTL逐步替换过程的概念，本例还阐明了如何给TLM通讯使用主机和从机双向端口。主机端口是用来将事务（在本例中，是fpu_request事务）放入从机，并获得相应的响应(fpu_response)事务。相应的从机端口是用来get请求并放回响应。在这个特例中，fpu_master直接与事务级模块fpu_tlm通讯。

主机

fpu_master有一个专门的tlm_blocking_master_if端口，叫做master_port，如下所示：

```
20    class fpu_master extends avm_named_component;
21
25
26      tlm_blocking_master_if #(.REQ(fpu_request),
27                               .RSP(fpu_response)) master_port;
file: topics/08_stepwise_refinement/fpu_sv/fpu_master.sv
```

既然本例中没有流水线操作，每次紧随master_port.get()调用master_port.put()。每个这样的调用模块，因此主机会停止直到从机完全事务了请求并提供了相应的响应。

TLM

为了与fpu_master通讯，且与后面的golden模型匹配，我们定义fpu_tlm有一个fpu_tlm（叫做master_channel），该通道将fpu_tlm与主机的master_port连接起来。Fpu_tlm也有一个分析端口（叫做response_ap），可用来做覆盖率采集。

```
20    class fpu_tlm extends avm_verification_component;
21      tlm_req_rsp_channel #(.REQ(fpu_request),
22                            .RSP(fpu_response)) master_channel;
23
26
27      tlm_blocking_master_if #(.REQ(fpu_request),.RSP(fpu_response))
28        blocking_master_export;
29
30      protected tlm_blocking_get_if #(fpu_request) slave_get_export;
31      protected tlm_blocking_put_if #(fpu_response) slave_put_export;
32
33      analysis_port #(fpu_response) response_ap;
file: topics/08_stepwise_refinement/fpu_sv/fpu_tlm.sv
```

为了简化指向master_channel的内部fifo的指针，我们声明接口*slave_get_*

*export*和*slave_put_export*，它们与master_channel的响应get和put执行关联。

```
42 function void export_connections;
43 blocking_master_export = master_channel.blocking_master_export;
44 slave_get_export = master_channel.blocking_get_request_export;
45 slave_put_export = master_channel.blocking_put_response_export;
46 endfunction // void
```

file: topics/08_stepwise_refinement/fpu_sv/fpu_tlm.sv

这些连接到位后，FPU TLM的行为就包含于compute()方法中。TLM只是等待请求，调用这个方法，返回响应：

```
53 slave_get_export.get(m_req);
56
58 m_rsp = compute(m_req);
59
62 slave_put_export.put(m_rsp);
```

file: topics/08_stepwise_refinement/fpu_sv/fpu_tlm.sv

SystemC实现细节

本例中的控制器非常简单，但它的结构值得研究。在控制器的开头，你可以看到它有两个连接，start_stop_port和op_cov。Sc_start_stop是和start_stop_config_if绑定的端口。控制器用这个接口来启动和停止在主机的激励产生。

```
31    class controller : public sc_module
32    {
33     public:
34      controller(sc_module_name nm) :
35        sc_module(nm),
36        start_stop_port("start_stop_port")
37      {
38        SC_METHOD(operator_coverage_notification_handler);
39        sensitive << op_cov.covered_event();
40
41        SC_THREAD(main_thread);
42      }
43      SC_HAS_PROCESS(controller);
44      sc_port<start_stop_config_if> start_stop_port;
45
46      coverpoint_notify_port op_cov;
47
48     private:
49      void main_thread();
50      void operator_coverage_notification_handler();
file: topics/08_stepwise_refinement/01_fpu_tlm_sc/controller.h
```

Op_cov是一个coverpoint_notify_port，这是一种在覆盖率采集器上的端口。一个coverpoint_notify_port有一个当覆盖率采集器中达到了覆盖率门限值时被触发的事件。方法operator_coverage_notification_handler()对由coverpoint_notify_port提供的covered_event触发。当达到覆盖率门限值时，该事件被触发，这将引起调用operator_ coverage_notification_handler()。

正如其名字所暗示的，Main_thread是控制器的主要thread。既然控制器整合了测试，它就也是整个测试的主thread。

```
25    void controller::main_thread()
26    {
27      start_stop_port->start();
28    }
file: topics/08_stepwise_refinement/01_fpu_tlm_sc/controller.cc
```

控制器调用主机上的start()来启动激励操作流。

主机

主机也有两个连接。Master_port是主机发送请求，接收响应的端口。Start_stop_ export是一个允许它绑定到控制器上的接口。

```
49 sc_port<tlm_master_if<fpu_request,fpu_response> > master_port;
50
```

```
51 sc_export<start_stop_config_if> start_stop_export;
```

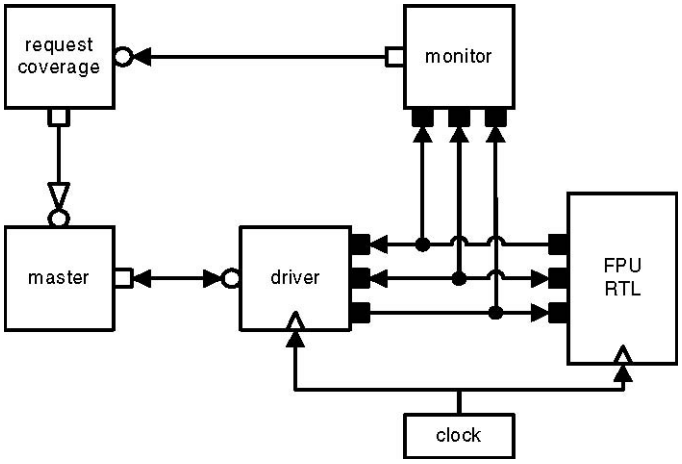
file: topics/08_stepwise_refinement/fpu_sc/fpu_master.h

回忆在SystemC中，输出口在一个请求/响应连接的“提供”侧。start_stop_export所提供的函数是start()和stop()，这两个函数都是fpu_master的方法。在构造器中，我们将start_stop_export绑定到这里，从而使得这些接口函数可以通过输出进行外部访问。现在，任何绑定到start_stop_export上的组件都可以调用start()和stop()。

```
39 fpu_master(sc_module_name nm) :
40 sc_module(nm),
41 master_port(“master_port”),
42 start_stop_export(“start_stop_export”)
43 {
44 start_stop_export(*this);
45 SC_THREAD(run);
46 }
```

file: topics/08_stepwise_refinement/fpu_sc/fpu_master.h

FPU RTL



说明

本实例是上面例子的逐步替换。一个FPU的TLM模板已经被FPU的RTL模板以及一个驱动器和一个监视器所取代。来自TLM测试平台中主机，覆盖率采集器，覆盖率反馈循环都被复用。

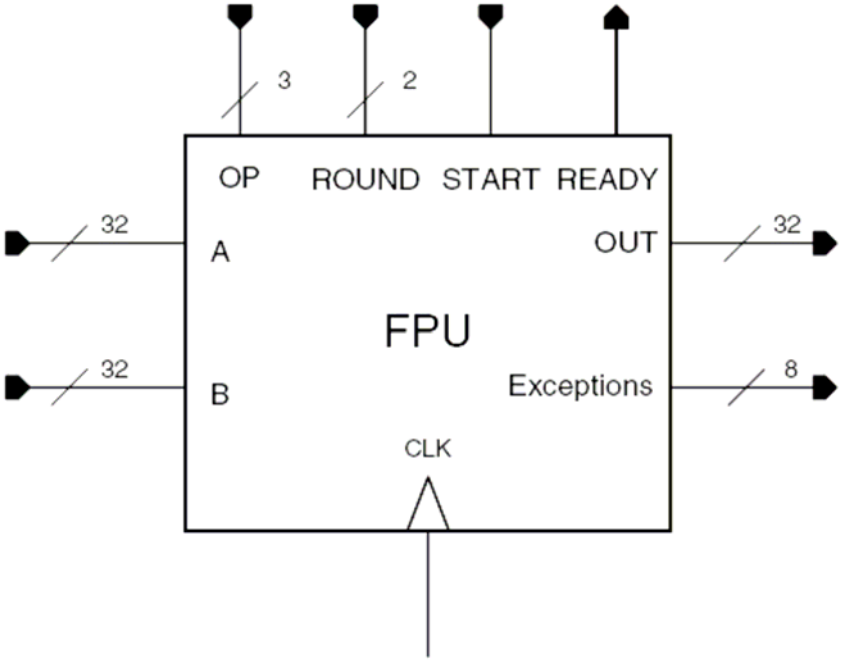


图8-2 FPU的RTL模板的输出管脚

FPU的RTL模板[1]有三套管脚，输入，输出和控制。本操作手册包包括了关于FPU模板的完整的文件。

输入

- A - 32 位浮点操作数
- B - 32 位浮点操作数

输出

- Out - 32 位浮点数
- Exceptions. FPU可以用信号通知8个例外

  Inexact不精确

  overflow 上溢

  underflow 下溢

  divide-by-zero 被0除

  infinity 无穷大

  zero 0

  QNaN

  SNaN

控制

操作数是3位，其支持的值为:

000 = add 加

001 = subtract 减

010 = multiply 乘

011 = divide 除

100 = square root 平方根

101 = unused 没用

110 = unused没用

111 = unused没用

圆整模式是2位。其可能的值为：

1. FPU 的 RTL 模板是被写入 VHDL 的 fpu100 模板。该模块的来源为 OpenCores.org. (http://www.opencores.org)

   a. 00 = round to nearest even 圆整为相近偶数
   b. 01 = round to zero 圆整为0
   c. 10 = round up 向上圆整
   d. 11 = round down 向下圆整

- Start. 在开始信号的上升边开始计算
- Ready. 当完成了一个计算时就插入这个信号

例外

Inexact不精确

overflow 上溢

underflow 下溢

divide-by-zero 被0除

infinity 无穷大

zero 0

QNaN

SNaN

关于VHDL FPU的设计详情在手册包的如下文件中：

topics/08_stepwise_refinement/fpu_vhd/doc/fpu.pdf

主要概念

- 一个TLM模板可以被一个RTL模板以及一套合适的事务器取代。事务器允许通过在管脚级action和事务流之间的转换，将RTL模板与事务级环境相

连。

- 复用事务级环境。在本例中，这包括了主机和覆盖率采集器。

FPU模板和驱动器共享一个时钟。这意味着由监视器收集的事务事件被计时了：能知道每个事务的开始和结束时间。我们在前面的例子中看到，在事务级FPU模板中，其事务是不计时的。它们的顺序和内容是重要的，但是它们发生的时间并不重要。然而，在RTL执行过程中，就要考虑时间。

即使我们有计时模板，我们对使用它来收集覆盖率时间也没兴趣。无论合适事务过来，覆盖率采集器都接收它们。

数据从没有计时的域移到计时域，然后返回到没有计时的域。激励发生器是不计时的，它给驱动器发送事务，驱动器将没有计时的事务转变为计时的管脚action。监视器将计时的管脚action转换为不计时的事务流。

事务器转换事务级和管脚级之间的数据，控制和计时，以授权不计时的事务域和计时的RTL域之间的接口。

SystemVerilog 实现细节

驱动器

既然在本例中我们使用同一个fpu_master组件，与fpu_driver之间的通讯就肯定是兼容的。因此，驱动器有一个tlm_req_rsp_channel master_channel和一个blocking_master_export，就如fpu_tlm一样。

```
20    class fpu_driver extends avm_verification_component;
21      tlm_req_rsp_channel #(.REQ(fpu_request),
22                            .RSP(fpu_response)) master_channel;
23
26
27      virtual fpu_pin_if m_fpu_pins;
28
29      tlm_blocking_master_if #(.REQ(fpu_request),
30                               .RSP(fpu_response)) blocking_master_export;
31      tlm_blocking_slave_if #(.REQ(fpu_request),
32                              .RSP(fpu_response)) master_chan_slave;
33
34      analysis_port #(fpu_response) response_ap;
35
file: topics/08_stepwise_refinement/fpu_sv/fpu_driver.sv
```

注意，fpu_tlm模板用来与tlm_req_rsp_channel通讯的不是独立的put和get输出口，而是使用slave_export::

```
44      function void export_connections;
45        blocking_master_export = master_channel.blocking_master_export;
46        master_chan_slave = master_channel.blocking_slave_export;
47      endfunction // void
file: topics/08_stepwise_refinement/fpu_sv/fpu_driver.sv
```

语义上，master_chan_slave接口与独立的get和put接口是一样的。这个单一的连接简化了代码。

fpu_driver的操作与fpu_tlm相似，即它get请求事务并将响应放回主机，主机阻塞等到结束的时钟单元。

```
69        forever begin
71          master_chan_slave.get(m_req);
74          issue_request(m_req);
77          wait(m_fpu_pins.ready == 1);
78
79          m_rsp = collect_response(m_req);
80
82          master_chan_slave.put(m_rsp);
83
84          @(posedge m_fpu_pins.clk);
85
86        end
87      endtask // run
```

监视器

监视器与fpu_pin_ifso相连，它可以监视与DUT相连的物理信号。它有两个analysis_ports来通讯请求和响应事务。

```
20    class fpu_monitor extends avm_verification_component;
21
24      virtual fpu_pin_if m_fpu_pins;
25
26      analysis_port #(fpu_request) request_ap;
27      analysis_port #(fpu_response) response_ap;
28
file: topics/08_stepwise_refinement/fpu_sv/fpu_monitor.sv
```

监视器本身对clk信号必须敏感，同时寻找总线上的请求和响应事务。

```
37      task run();
38        forever
39          @(posedge m_fpu_pins.clk) fork
40            if(m_fpu_pins.start == 1'b1) begin
42              monitor_request();
43            end
44            if(m_fpu_pins.ready == 1'b1) begin
46              monitor_response();
47            end
48          join
49      endtask // run
file: topics/08_stepwise_refinement/fpu_sv/fpu_monitor.sv
```

当看到一个请求事务时，监视器组合一个fpu_request对象，并将它复制给m_req_in_process。然后将m_req_in_process发出分析端口。

```
51      task monitor_request();
52        fpu_request req;
67        req.a = $bitstoshortreal(m_fpu_pins.opa);
68        req.b = $bitstoshortreal(m_fpu_pins.opb);
69        req.op = op_t'(m_fpu_pins.fpu_op);
70        req.round = round_t'(m_fpu_pins.rmode);
71
72        m_req_in_process = req.clone();
73
77      endtask // monitor_request
file: topics/08_stepwise_refinement/fpu_sv/fpu_monitor.sv
```

当看到一个响应事务时，监视器通过捕捉来自相应的请求事务的操作数和操作符值并从总线抓住结果值来组合一个fpu_response对象。然后将响应和相应的请求事务分别写给各自的analysis_ports。

```
79 task monitor_response();
80 fpu_response rsp;
90 rsp = new();
91
92 rsp.a = m_req_in_process.a;
93 rsp.b = m_req_in_process.b;
94 rsp.op = m_req_in_process.op;
95 rsp.round = m_req_in_process.round;
96
97 rsp.result = $bitstoshortreal(m_fpu_pins.outp);
109 request_ap.write(m_req_in_process);
110 response_ap.write(rsp);
111
115 endtask // monitor_response
file: topics/08_stepwise_refinement/fpu_sv/fpu_monitor.sv
```

SystemC实现细节

驱动器

有三个组件：监视器，驱动器和RTL FPU，它要取代事务级FPU，就必须要具有和它将取代的事务级模板相同的接口。对于驱动器，这意味着它必须具有与事务级FPU相同的事务级接口。

```
38    class fpu_driver : public sc_module, public fpu_driver_pin_if
39    {
40     public:
41      fpu_driver(sc_module_name nm) :
42        sc_module(nm),
43        req_rsp_channel("req_rsp_channel"),
44        master_export("master_export"),
45        slave_export("slave_export")
46      {
47        SC_THREAD(run);
48
49        master_export(req_rsp_channel.master_export);
50        slave_export(req_rsp_channel.slave_export);
51      }
52      SC_HAS_PROCESS(fpu_driver);
53
54      sc_export<tlm_master_if<fpu_request,fpu_response> > master_export;
55
56
57     private:
58      void run();
59      void issue_request(const fpu_request& req);
60      fpu_response collect_response(const fpu_request& req);
61
62      tlm_req_rsp_channel<fpu_request,fpu_response> req_rsp_channel;
63      sc_export<tlm_slave_if<fpu_request,fpu_response> > slave_export;
64    };
file: topics/08_stepwise_refinement/fpu_sc/fpu_driver.h
```

Req_rsp_channel是一个双向通道，通过它请求和响应流，并且正好与在事务级FPU中同一个名字的通道具有同样的类型。请求/响应通道与主机和从机输出口相连，这里给找回的请求和记入的响应提供接口。

驱动器获得来自激励发生器的请求，将它转换为管脚级的活动，然后等待管脚接口上响应的出现。当ready为高时就可以获得响应。

```
53      while(1)
54      {
55        req = slave_export->get();
56        issue_request(req);
57
58        while(ready != 1)


59        {
60          wait(ready.default_event());
61        }
62        rsp = collect_response(req);
63        slave_export->put(rsp);
64
65        wait(clk.default_event());
66      }
file: topics/08_stepwise_refinement/fpu_sc/fpu_driver.cc
```

最后，它收集响应信息，构造响应对象，将响应反送给主机。

监视器

监视器是三个组件群的成员之一，它用一个RTL模板来取代事务级 FPU模板。

因此，就如驱动器一样，它也必须提供与事务级FPU模板相同的接口。对于驱动器，相关的接口是给激励请求和响应的接口。对于监视器，相关的接口就是请求分析端口。

```
34    class fpu_monitor : public sc_module, public fpu_monitor_pin_
35    {
36     public:
37      fpu_monitor(sc_module_name nm) :
38        sc_module(nm)
39      {
40        SC_METHOD(monitor_request);
41        sensitive << clk.pos();
42
43        SC_METHOD(monitor_response);
44        sensitive << clk.pos();
45      }
46      SC_HAS_PROCESS(fpu_monitor);
47
48      analysis_port<fpu_request> request_ap;
49      analysis_port<fpu_response> response_ap;
50
51      void monitor_request();
52      void monitor_response();
53
54     private:
55      fpu_request req_in_process;
56    };
file: topics/08_stepwise_refinement/fpu_sc/fpu_monitor.h
```

监视器有两个分析端口，一个用来发送请求信号流，另一个用来发送响应信号流。请求分析端口与事务级FPU模板上的分析端口相匹配。

监视器通过两个方法操作，一个用来识别请求，另一个用来识别响应。识别请求的方法首先看是否开始位被确认。如果已经被确认，则在总线上就激活了一个请求，这样，它收集来自管脚的信息并形成请求对象。

```
25    void fpu_monitor::monitor_request()
26    {
27      fpu_request req;
28
29      if(start != 1)
30    return;
31
32      req.a = bits_to_float(opa);
33      req.b = bits_to_float(opb);
34      req.op = bits_to_op(fpu_op);
35      req.round = bits_to_round(rmode);
36
37      // put the request into the pipeline
38      req_in_process = req;
39
40      cout << sc_time_stamp() << ": monitor request: " << req << endl;
41    }
```

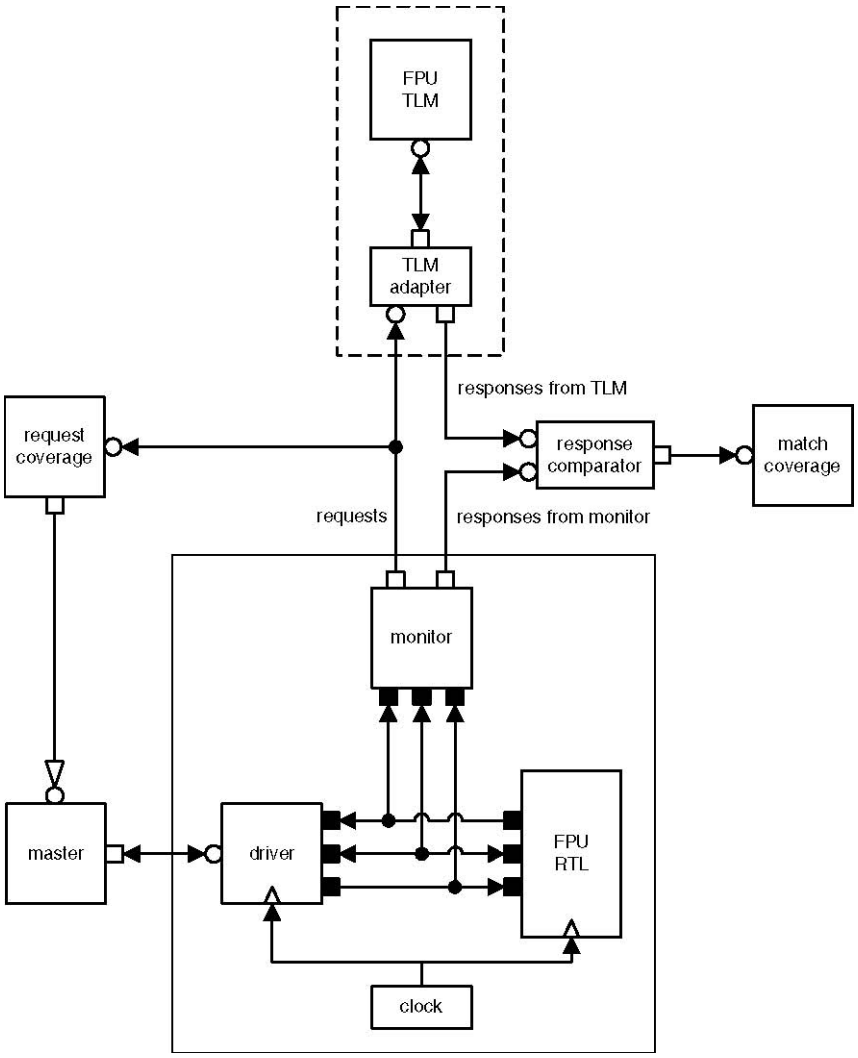
FPU DUT是流水线的，在接收到一个请求一个时钟单元以后，就出现一个响应。为了正确的跟随DUT的action，监视器类似于pipelined。请求方法不是立即

将请求发出分析端口，而是储存在用变量req_in_process表示的流水线中。

```
46    void fpu_monitor::monitor_response()
47    {
48      fpu_response rsp;
49
50      if(ready != 1)
51    return;
52
53      rsp.a = req_in_process.a;
54      rsp.b = req_in_process.b;
55      rsp.op = req_in_process.op;
56      rsp.round = req_in_process.round;
57
58      rsp.result = bits_to_float(output);
59
60      // collect up status information
61      rsp.status[STATUS_INEXACT] = ine;
62      rsp.status[STATUS_OVERFLOW] = overflow;
63      rsp.status[STATUS_UNDERFLOW] = underflow;
64      rsp.status[STATUS_DIV_ZERO] = div_zero;
65      rsp.status[STATUS_INFINITY] = inf;
66      rsp.status[STATUS_ZERO] = zero;
67      rsp.status[STATUS_QNAN] = qnan;
68      rsp.status[STATUS_SNAN] = snan;
69
70      // send out the pipelined request and response
71      request_ap.write(req_in_process);
72      response_ap.write(rsp);
73
74      cout << sc_time_stamp() << ": monitor response: " << rsp << endl;
75    }
```

响应方法首先检查准备位是否被确认。如果已经被确认，则在总线上就激活一个响应。因此它收集用于响应的信息。有些响应信息用于pipeline中的当前请求。当用于响应的所有数据都被组合起来时，监视器在请求分析端口发出请求并在响应分析端口发出响应。

FPU golden 模型



说明

本例中测试组织的要旨是比较TLM和RTL运行响应。RTL和事务级模块是平行运行，并比较它们对同一激励的响应。

为了构造这个测试，又复用了事务级模型作为golden模型。监控器产生两个事务信息流-请求和响应。请求信息流被输送到TLM模块（未计时）。响应信息流则输入到响应器。同样的，来自TLM的响应也被送到比较器。比较器比较由同一个请求产生的TLM响应和RTL响应。

主要概念

- 最初构造TLM是用来在事务级设计开发的，现在则又被用作RTL测试golden模块。在运行过程中比较RTL和事务级的运行结果来创建一个自我检测的测试平台。
- 监视器产生两个事务信息流，一个包含请求，一个包含响应。
- 本测试中的大部分组件都要被再次利用。在新组件中，即TLM适配器，响应比较器和比较覆盖率采集器，只有TLM适配器是真正的新组件。其它都可以取自库组件。

逐步细化的程序包括用RTL模块替换事务级模块。在做替换时，你要确认他们的运行是相同的。本例就是说明如何通过增加一个给TLM传送请求和找回响应的适配器，而将TLM用作一个golden模块。

SystemVerilog 实现细节

本例中真正的新组件是TLM适配器。该组件有三个端口：主机端口，它与TLM相连；请求端口，用来找回请求和响应端口，它用来发送响应。

```
20    class fpu_tlm_adapter extends avm_verification_component;
21
25      tlm_blocking_master_if #(.REQ(fpu_request),
26                               .RSP(fpu_response)) master_port;
27
28      analysis_fifo #(fpu_request) request_fifo;
29      analysis_if #(fpu_request) request_export;
30
31      analysis_port #(fpu_response) response_ap;
file: topics/08_stepwise_refinement/fpu_sv/fpu_tlm_adapter.sv
```

request_export就是一个 analysis_if用于将其与监视器的输出相连。这是一个非常简单的装置，它接收来自fifo输入的请求，然后发送到TLM。然后它就等待来自TLM的响应。一旦它接收到响应，就将其送出分析端口，送到in_order_comparator。

```
45      task run;
47        forever begin
48          request_fifo.get(m_req);
51          master_port.put(m_req);
52
53          master_port.get(m_rsp);
56          response_ap.write(m_rsp);
57        end
58      endtask // run
file: topics/08_stepwise_refinement/fpu_sv/fpu_tlm_adapter.sv
```

TLM FPU设计与tlm_master_if相连而不是一个分析端口。该适配器是用来给监视器，TLM FPU模块和in_order_comparator之间提供联系的。

SystemC 实现细节

本例中真正的新组件是TLM适配器。该组件有三个端口：主机端口，它与TLM相连；请求端口，用来找回请求和响应端口，它用来发送响应。

```
47      sc_port<tlm_master_if<fpu_request,fpu_response> > master_port;
48      sc_export< analysis_if<fpu_request> > request_export;
49      analysis_port<fpu_response> response_ap;
file: topics/08_stepwise_refinement/fpu_sc/fpu_tlm_adapter.h
```

request_export就是一个 analysis_if用于将其与监视器的输出相连。这是一个非常简单的装置，它接收来自fifo输入的请求，然后发送到TLM。然后它就等待来自TLM的响应。一旦它接收到响应，就将其送出分析端口，送到in_order_comparator。

```
51      void run()
52      {
53        fpu_request req;
54        fpu_response rsp;
55
56        while(1)
57        {
58          req = request_fifo.get();
59          master_port->put(req);
60          rsp = master_port->get();
61          response_ap->write(rsp);
62        }
63      }
file: topics/08_stepwise_refinement/fpu_sc/fpu_tlm_adapter.h
```

TLM FPU设计与tlm_master_if相连而不是一个分析端口。该适配器是用来给监视器，TLM FPU模块和in_order_comparator之间提供联系的。

第 9 章　有约束的随机验证

验证工程师写的测试为一个设计提供激励以便测试不同的功能性函数并检查它是否按照预期进行响应。每个这样的测试都需要时间来写入，调试和运行。一旦测试被证明有效，它就成为回归测试的一部份，然而客户也会产生一些新的测试，这也是需要时间来写入，调试和运行的。这个方法要求写测试的人为设计的每一个特点创建明确的测试，这样当设计中加入更多的特点时，只需要增加测试（增加时间）。

随机约束验证（CRV）可有效的允许单一的测试来检查多个功能点，而不是要求验证工程师写入测试来独立地检查每一个功能点。有了这个方法，每个测试实际上就要描述一系列可能发生的事件。这是一个更为有力的验证方法，但是这个方法却得不到Verilog 或 VHDL 的有力支持。SystemVerilog正是设计用来支持这个方法的。

CRV 方法概述

定向测试

当验证工程师设计一个单一的时钟单元进行那些可能跨越多个时钟单元的激励时就使用定向测试。每个时钟单元或单元群群定向为验证一个由验证工程师选出的特殊的功能点。

正是由于定向测试的直接性的特点，它们就很容易写入。但是，明确的来说，它们只能表述验证工程师预计的明确的事件。随着设计变得越来越复杂，写这些定向测试来覆盖所有可能发生的边际情况也变得越来越难，这有两方面的原因：一是预计的响应变得越来越难，二是很难碰到边际情况，即使它们能被预料到。这种测试可以通过增加随机性来提高，比如给存储器写入随机值，除了（或代替） walking-ones pattern。这些测试仍然是定向的。

与功能点无关的被测数据可能仍然采用随机数发生器作为内容的填充。

Verilog系统的$random 或 $dist_uniform 功能提供了用随机数填字节的简单方式。它们还可用来随机化延迟或时钟单元计数。

在有些时候，给一个定向测试增加大量的随机数并不足以说明测试问题。测试代码本身的程序化特点限制了它的有效性，也限制了能引出的新测试的数量。开始采用这种方法就排除了构造一个真正的随机化环境，在这样的环境里边际情况会被执行到，从这些事件中可以以最小的代码变化引出新的测试。

有约束的随机验证

咋一看，如果发生器允许发生的随机数没有受到约束，将所有的随机激励输入到设计中的想法好像效率很低，事实上也是这样。CRV背后的观点就是数据和有测试产生的事务信息是从一套有效的，或受约束的，可能性中随机选择的。

例如，对于每一个操作不需要写入很多不同的任务，而只需要创建一个命令任务，操作就能象数据一样进行编码，它就可以作为判据被传送到一个任务中，如下面的例子所示。

```
begin
   write(address[1],data1);
   write(address[2],data[2]);
   read(address[3],data[3]);
   write(address[4],data[4]);
end

enum {READ, WRITE} op[N];
for (int i=0;i<N;i++)
   command(op[i],address[i],data[i]);
```

最后，命令任务中的所有参数都被封装到一个具有不同领域的，所有选择可随机化的对象中。

验证工程师的任务就是将那些已被证明合法的输入带到设计中，并将这些信息变成一系列约束。注意如果设计说明书指出设计是可以处理这些错误的话，在输入激励中产生的错误就是合法的。

写被约束的随机测试和测试设计只是一个完整的验证方法的一部分。功能覆盖率报告一个测试计划中随机发生的测试也是必要的。

约束随机中的定向测试

在一个约束的随机环境中，必须严谨的约束条件以便执行某一单独事件就可以得到定向测试。因此，在这个环境中，通过约束测试，给一个规定的地址产生一个写入在从同样的地址中读出后，就是一个完整的测试。一旦这个全面的测试有效，那就证明读/写接口工作正常。此时就可以去掉约束，产生全范围的测试。在本测试中，所有的读/写都是随机的，

就可以去掉约束，产生全范围的测试。在本测试中，所有的寄存器都是按照随机顺序，以不同的方式进行读/写的。当出现问题时就很容易给测试增加新的约束以便集中引起问题的功能点，使得它能被调试。

这并不是说在有些环境中定向测试没有用。相反，对于一定的事件，写定向测试来保证设计迅速达到一定的状态更为容易，而不是依赖于随机行为达到预期结果。例如，当验证一个中断信号的计时寄存器时，测试必须一次性设定每一个计时字节，产生相应的中断信号，然后清除这个计时字节。随机获得这种行为的几率实际上为0。因此，在这种情况夏，写一个定向测试就更有效。

技术基础

约束的随机验证背后的技术可分为两个基本部分。一部分是实际随机数产生（RNG）程序。尽管这在很大程度上是一个学术讨论，客户还是应当知道几点的，比如种子的选择和稳定性。另一部分是实际的约束求解器。求解器确定是哪一套值可能满足给定的约束，如果有的话。这些值就称为求解空间。实际上，求解器首先计算求解空间，然后利用随机数发生器选择求解中的一个。

产生随机数

计算机中产生随机数的程序并不是真正完全随机的。实际上，它是一个可预言的数的序列，称做伪随机数。发生器看起来是随机的，因为这个序列需要很长时间才会重复。执行整数RNG表示一个带有内部状态寄存器，典型的是96字节的序列。因此，在产生2^{96}个32字节的之后，该序列才会重复。给一个字

节数比内部状态寄存器更大的变量产生随机数就意味着这个序列要在达到所有可能的数以后才会重复。RNG程序掌握测试中大量的随机方面，包括约束和和随机过程语句。现在来看看第4行$urandom的系统函数，该函数是由SystemVerilog提供，可直接访问RNG。下面的例子给出了有5个伪随机数的序列。

```
1   int A;
2   initial repeat (5)
3     begin
4       A =$urandom;
5       $display(A);
6     end
```

无需对代码进行任何修改，RNG在每次执行模拟时发送同样的随机数序列。这是一个重要性能，因为如果一旦在正在测试的设计中发现问题，就可以用同样的测试来修订，校验设计。

随机种子

随机数发生器可以通过接收一个给定初值来改变它们所产生的数值序列。该初值本来是一个代码，它在RNG的内部状态中被转换为值，代表一个特定的序列。

```
1   int A;
2   initial
3     begin
4       A=$urandom(`mySEED);
5       repeat (5)
6       begin
7         A =$urandom;
8         $display(A);
9       end
10    end
```

第4行中的值mySEEDon可以按照要求改变，以产生不同的有有5个伪随机数的序列。

随着设计变得越来越复杂，产生随机数的序列就会被其它产生随机数的序列干扰。A的随机数序列会受到在下面第5行的，称作$urandom的干扰。

```
1 int A, B;
2 initial repeat (5)
3 begin
```

```
4     A =$urandom;
5     B =$urandom;
6     $display(A);
7  end
```

如果最初的例子为A产生了序列6, 35, 41, 3, 27，新的序列就会安6, 41, 27, 等开始，因为产生的数35和3将成为序列B中的一部分。

避免这种干扰发生的方法是对每一次调用$urandom用不同的变量手动赋初值。每个变量必须有一个唯一的值，否则每个RNG会产生相同的随机值序列。另一个解决这个问题的方法是给每一个RNG定义一个程序，自动产生独立的初始值。这个程序和可复写的随机值序列一起给系统提供宿疾稳定性。

随机稳定性

必须知道修改激励会影响随机稳定性，这对调试和回归测试是重要的。在SystemVerilog中，每个分级子系统（一个指令舱，程序或界面）的实例都是从自己的根种子开始的，然后用RNG选取本子系统中的其它初值。这样，一个子系统产生的随机数就不会受到增加或移走其它子系统的影响。

在每一个子系统中，每个静态线程（一个初始或常阻塞）都按照子系统中线程声明的顺序依次从根种子中获得派生的种子。在这个静态线程中，动态子线程按照线程程序开始的顺序获得派生种子。因此，每个线程，无论是静态的还是动态的，都有自己的种子。在每个静态线程中调用$urandom，就产生一个稳定的随机值序列，只要维持着相关的线程关连：给静态线程声明顺序，给动态线程程序上的顺序。

在下面的实例中，为A产生的随机数序列没有被干扰，因为第7行的初始块是在第2行公告的静态线程以后加入的静态线程。

```
 1  int A, B;
 2  initial repeat (5)
 3    begin
 4      A =$urandom;
 5      $display(A);
 6    end
 7  initial repeat (5)
 8    begin
 9      B =$urandom;
10    end
```

实际上，产生随机数至少要有一个约束，随机化其大小（或字节数）。大小决定了随机变量可能有的数值总量，即，求解空间的大小。一个约束基本上是一个布尔表达式，要使求解器选取这些数值，这个表达式的值应为真。一个约束通常会减小求解空间的大小。

一个约束表达式可以是随机和非随机变量混合而成。非随机变量使得约束状态独立，也就是说在测试期间可以动态改变约束。如果随机变量没有约束，或看起来有约束而没有其它随机变量，就称作标量随机变量。其求解空间可以分开并独立求解。

求解空间的大小是所有连带的受约束随机变量的可能值的排列。求解器对所有可能的求解创建顺序，并用RNG选取其中一个有序的求解结果。

例如，一个8字节的标量随机变量，A就有约束条件2 < A && A<9。它就有6个可能的求解结果（3－8），这个随机变量的大小就会减小为4字节。

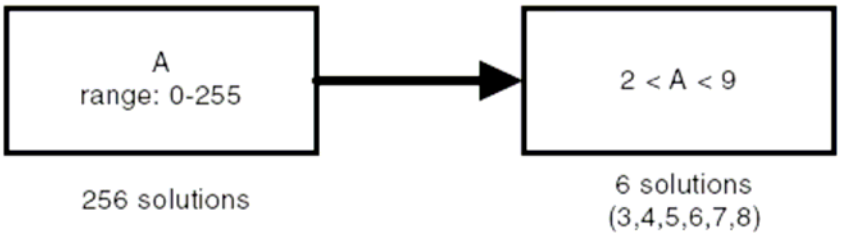


图9-1 单一变量约束

现在假定有2个8位随机变量，A和B。在没有任何约束条件的情况下，有2个求解空间，对A和B每个都有256个可能的值。然后加入约束条件A!=B。有了这个约束条件，现在就只有1个求解空间，有65280个解。（A和B的排列是2^{8+8} = 65,536个解，减去A==B.时的256个解）。它就被认为是一个16字节的随机变量。

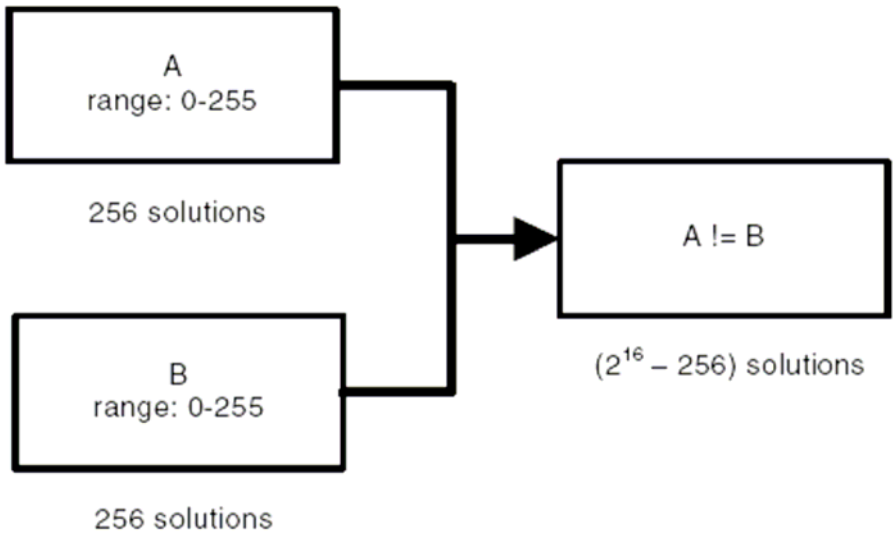


图9-2 多变量约束

求解一个变量所需要的时间与在一个求解空间必须求解的随机变量总的关联数成正比。将关联随机变量的总位数降低到求解最小值。尽可能的消除在每一步的随机化过程中由约束条件引起的随机变量之间的相互依赖。例如，如果你想产生一列具有100个随机变量的数，可以按照$A_i < A_{i+1}$创建迭代约束条件。但是这样产生的单个随机变量为3200位，这在任一个合理的cpu时间内，很可能是不能求解的。一个更好的办法是产生100个独立的32位的随机数，然后按程序要求对它们分类。

以对象为导向的随机化

在SystemVerilog中，随机变量，随机数发生器和约束器都被整合在以对象为导向的系统中。你不需要了解以对象为导向的系统所有方面就可以开始使用CRV。下面是一些重要的概念。

以对象为导向的基础

简单的来说，类就如一个结构，是一个封装的数据。在SystemVerilog中，类是动态产生的，而结构是在它们被声明时产生的。

class Packet;
  bit [7:0] m_address; // properties
  bit [31:0] m_data;
endclass : Packet
	typedef struct {
	  bit [7:0] m_address; // members
	  bit [31:0] m_data;
	} Packet_t;

Packet P; //declares a handle to Packet
P = new(); //Constructs an instance of Packet
// P now has a reference to a Packet class
P.m_data = 1234;
	Packet_t P; //declares an instance of Packet
	
	P.m_data = 1234;

用C++术语来说，一个数据变量是一个类说明中的一部分，称作属性。加入任务和功能的定义称作方法，类将数据和功能性联系在一个单一的目标中。这就使得执行类的详细情况可以被从类客户中抽象出来。本例中，任务的书写者并不知道第10行是在访问数据还是函数。

```
 1  class Packet;
 2    bit [7:0] m_address;
 3    bit [31:0] m_data;
 4    function bit parity;
 5      return ^m_data;
 6    endfunction
 7  endclass : Packet
 8
 9  task command( input Packet P);
10  if (P.m_parity)
11  ...
12  endtask
```

面向对象编程的真正作用是通过使用继承获得的。一个新类可以定义为当前定义的基类的派生类。派生类的定义可以按照基类数据的性能或方法加入也可搜索优先顺序。下面的例子就通过增加一个新域m_id扩展了原来的信息包类，不考虑奇偶法。

```
1  class Bad_Packet extends Packet;
2    int m_id;
3    function bit parity;
4      return ~^m_data;
5    endfunction
6  endclass : Bad_Packet
```

扩展类创建了有层次的类型，它使得单一的基类在其根和子类处依次扩展。在创建子集时，默认情况下，所有的高级类或超级类是同时创建在基类的根下。

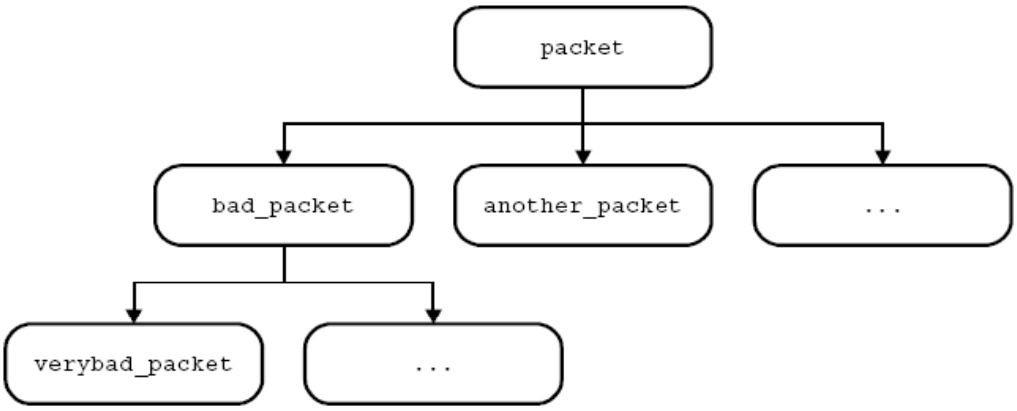


图9-3 包的类层级

在图9-3中，在创建Really_Bad_Packet时，同时创建了Bad_Packe和Packet的性能，但它们在扩展集中时隐藏的，不能直接访问。但我们讨论随机变量和约束条件时，这个特点就很重要，因为约束求解器对超级类中的所有性能都进行求解。

```
1  class Really_Bad_Packet extends Bad_Packet;
2    bit [33:0] m_data;
3    function bit parity;
4      return ~^super.m_data;
5    endfunction
6  endclass : Bad_Packet
```

第4行super.m_data的指针指向Packet中m_data的性能。超级前缀在高级类级别上开始搜索对象。

给对象增加随机性

SystemVerilog用rand modifier来区别随机变量和非随机变量。增加一个带有名字的公式列表的约束，用约束关键字声明约束。

```
1  class Packet;
2    rand bit [7:0] m_address;
3    rand bit [31:0] m_data;
4    constraint addr_range {
5       m_address < 132;
6    }
7  endclass : Packet
```

通过采用类的内嵌式方法来调用RNG和约束求解器， 在构造了类以后就可以调用randomize()。在本例中， 调用P.randomize将用约束addr_range随机化m_address，它也会在没有约束的情况下随机化m_data。

```
1  Packet P;
2  initial begin
3     P = new();
4     if (!P.randomize) $stop;
5     $display(P.m_address);
6     end
```

一直检查随机化的返回值。如果放在一个类中的随机变量约束没有解，randomize()返回0。检查返回值是严格的，这样任何问题都会报告给用户。为了报告这些问题，立即声明就很有用。

assert (P.randomize) else $error(“No solutions for P.randomize”);

用继承法的层次约束

在扩展类时，可以增加约束，也可以改写约束，就如属性和方法一样。基类中的所有现有的约束仍然有效，除非它们被改写。例如，假设前面例子里的里的Packet类被扩展：

```
1 class Word_Packet extends Packet;
2 constraint word_align {
3     m_address[1:0] ==  ‘0;
4}
5 endclass : Word_Packet
```

在随机化Word_Packet类中的一个情况时，此约束对addr_range和word_align同样适用。当改写一个随机属性时，要确保父集属性的现有约束也已经被改写。

在扩展一个类时，现有的约束对所有的扩展加入属性没有任何影响。约束只适用于类继承范围内的被定义过的属性。例如，假设前面的类Word_Packet被定义为：

```
1 class Word_Packet extends Packet;
2 rand addr_t [0:3] m_address; // now 32 bits
3 constraint word_align {
4 m_address[1:0] ==  ‘0;

5}
6 endclass : Word_Packet
```

对派生类中的m_address的唯一约束就是word_align。super.m_address仍然在基类中，且约束addr_range有效。

有时，考虑通过基类层级，将扩展类名称定义在前面，然后利用基类。从上面的例子中，类名为的Word_Packet就是公用的。如果不需要额外的约束条件，就不需要提供内容来扩展类。

```
class Word_Packet extends Packet; endclass
```

管理约束

动态修改约束

约束可以指向非随机变量，它也可以是类的属性或外部的所有变量。在执行测试是，动态修改变量就可以修改约束。

```
 1  typedef bit [7:0] addr_t;
 2  typedef bit [31:0] data_t;
 3
 4  class Packet;
 5    rand addr_t m_address;
 6    rand data_t m_data;
 7    addr_t high_address = `1;
 8    addr_t low_address = 0;
 9    constraint addr_range {
10       m_address <= high_address;
11       m_address >= low_address;
12    }
13  endclass : Packet
14  Packet P;
15  initial begin
16            P = new();
17            P.high_address = 10;
18            assert (P.randomize) else $stop; // range is 0-10
19            $display(P.m_address);
20            P.high_address = `1
21            P.low_address = 250;
22            assert (P.randomize) else $stop; // range is 250-255
23            $display(P.m_address);
24          end
```

约束也可以通过约束名用程序打开或关闭。将P.addr_range.constraint_mode的值指定为0就可以关上约束addr_range。

过度约束

如果给足够的时间，约束求解器总能找到求解空间，如果存在求解空间的话。如前面所提到的，CPU的时间增加与一个求解空间中必须求解的随机变量的关联位的总量成正比。

可以将随机变量分隔在不同的类中，用程序随机化每一个类，然后用第一次随机化的结果作为第二次的状态变量。但这样就削弱了面向对象模块的意图，而且使得更加难以维持。SystemVerilog提供了几种可选择性来保持所有的随机变量在一个类中。

那些只是其它随机变量的派生变量可用函数调用。内嵌在约束中的函数调用提供了将随机变量的求解空间分开的途径。也就是说，在调用函数以前，先求解和选择作为函数判据的随机变量。

```
 1   class Packet;
 2     rand bit [7:0] m_address;
 3     rand bit [31:0] m_data;
 4     rand bit m_data_parity;
 5     constraint addr_range {
 6        m_address < 132;
 7     }
 8     constraint gen_data_parity {
 9        m_data_parity == even_parity(m_data);
10     }
11     function bit even_parity(bit [31:0] d);
12       return (~^d);
13     endfunction
14   endclass : Packet
```

既然函数调用嵌套在约束内，约束就可以被打开或关上。如果被关上，m_data_parity返回成为一个独立的随机变量。

仅仅是其它随机变量的派生变量应采用post_randomize()方法。

SystemVerilog自动调用用户定义的pre_randomize()和post_randomize()方法作为执行randomize()方法的一部分。在下面的例子中，m_data_parity不再是随机变量，而是被post_randomize方法放在第9行。

```
1 class Packet;
2 rand bit [7:0] m_address;
3 rand bit [31:0] m_data;
4 bit m_data_parity;
5 constraint addr_range {
6 m_address < 132;
7}
8 function void post_randomize();
9 m_data_parity = ~^m_data;

10 endfunction

11 endclass : Packet
```

隐含

隐含表达式A -> B有条件的应用了约束B，如果表达式A是真的话。隐含执行器在本质上倾向于显示它们是单向的，但是，这里有一个双向效应，即如果约束B为真的可能性很小的话，则A为真的可能性也非常小。

如果隐含的两面都包含随机变量，就要检查两面的双向可能性是否为真。在下面的实例中，在第7行的变量enum中有一个隐含约束，即m_op必须是READ, WRITE, 或 NOP中的一个。

```
1 typedef bit [7:0] addr_t;
2 typedef enum {READ,WRITE,NOP} kind;
3
4 class Packet;
5 rand addr_t m_address;
6 rand bit [31:0] m_data;
7 rand kind m_op;
8 constraint data_range {
9 (m_op == READ) -> m_data inside {[1:100]};

10(m_op == WRITE) -> m_data inside {[101:255]};
11(m_op == NOP) -> m_data inside {0};
12 }
13 endclass : Packet
```

如果m_op等于READ，m_data就有100个可能值满足第一个隐含。如果m_op等于WRITE，m_data就有155个可能值满足第二个隐含。但是，如果m_op等于NOP，m_data就只有1个可能值满足第三个隐含。因为m_data只有1/256可能值满足第三个隐含，所以m_op的值为NOP的机率仅为0.004。

分配和求解顺序

实际上，随机化过程包括计算求解空间，然后随机选取一个解。这个解就作为给定值写给随即变量。通常，求解器用相同的机率来选取每个解。也可以建议求解器在选取其它变量之前选取特定的随机变量的值。所有随机变量的求解空间保持不变；但是，只在最早选取的随机变量选择中存在机率一致。如果将前面的例子改为先求解m_op，后求解m_data，则在为m_data选择值之前READ, WRITE, 和 NOP具有相同的入选机率。

```
1 class Packet;
2 rand addr_t m_address;
3 rand bit [31:0] m_data;
4 rand kind m_op;
5 constraint data_range {
6                         (m_op == READ) -> m_data inside {[1:100]};
7                         (m_op == WRITE) -> m_data inside {[101:300]};
8                         (m_op == NOP) -> m_data inside {0};
9 }

10 constraint order {solve m_op before m_data;}
11 endclass : Packet
```

除了建议首先为哪个随机变量选择值以外，分配约束为选择值增加了权重因子。这不需要修改求解空间，除非有一个权重为0。在下面的实例中，m_op选择READ的机率为2/5（40％），选择WRITE的机率为2/5（40％）选择NOP的机率为1/5（20％）。

```
1 class Packet;
2 rand addr_t m_address;
3 rand bit [31:0] m_data;
4 rand kind m_op;
```

```
5 constraint data_range {
6 (m_op == READ) -> m_data inside {[1:100]};
7 (m_op == WRITE) -> m_data inside {[101:300]};
8 (m_op == NOP) -> m_data inside {0};
9 }

10 constraint op_dist { m_op dist {READ := 2, WRITE := 2 NOP := 1};}
11 endclass : Packet
```

在一系列交错的随机变量中只针对一个随即变量使用分配约束。当出现多重的，相冲突的约束时，就很难计算出选择随机变量值的机率。在创建了求解空间后采用分配约束，且不能保证满足分配约束。

约束中的有用操作

设定成员资格

inside操作符是一个用来动态创建一套约束值的强大结构。Inside操作符测试给定的成员。一定范围的值就是一套，这里使用inside操作符来测试一个值是否在有效的范围内。有两种方法来使用它。如果内存排列的大小为0(ad_q.num == 0)，则将m_op约束为NOP 或 WRITE。如果内存排列的大小不是0，则将m_op约束为READ ，而且m_addr在内存的有效地址范围内。下面的例子说明如何将read事务约束为只针对那些已经写入的地址。

```
1  module memq;
2     typedef bit [7:0] addr_t;
3     typedef enum {READ,WRITE,NOP} op_t;
4
```

```
 5      class Packet_Xaction extends Packet;
 6
 7      rand op_t m_op;
 8      rand addr_t m_addr;
 9      addr_t m_addr_q[addr_t];
10
11      constraint read_only_written {
12                                     if (m_addr_q.num == 0)
13                                       m_op inside {NOP,WRITE};
14                                     else
15                                       m_op == READ -> m_addr inside {ad_q};
16                                    }
17      function void post_randomize();
18         if (m_op == WRITE) m_addr_q[m_addr] = m_addr;
19      endfunction // void
20
21      endclass
22
23    Packet_Xaction X = new;
24
25    initial repeat (100)
26      begin
27         if (X.randomize);
28         $display("%s %0d",X.m_op,X.m_addr);
29      end
30    endmodule : memq
```

动态数组

通常，当动态数组被声明为随机变量时，它们就被认为是固定大小的数组。在某些情况下，如果在一个约束中引用动态数组的大小，这个大小就作为随机变量处理。在给这个数组重新分配大小时，约束求解器会求解其大小，这个求解器会在个别元素上应用其余的约束，在那一点上将这个大小作为状态变量处理。

如果有元素包含了类句柄，你就不能将数组的大小当成随机变量来处理，因为SystemVerilog 不会构造一个类对象作为约束求解过程的一部分。一个类句柄的动态数组必须至少在两个阶段进行随机化以确保正确的构造类对象。通过使用方法pre_randomize 或post_randomize，一个类句柄的动态数组就可以在一个随机化操作中被随机化。按照类中元素和其它随机变量之间的约束关系，你可以使用pre_randomize来求解大小也可以用主要的随机化方法来求解元素。你也可以使用主要随机化方法来求解大小，用方法post_randomize来求解元素。在下面的实例中，方法pre_randomize用来选取大小并构造动态数组中的元素。

```
1   class array_pre;
2     rand Packet array[];
3     int m_array_size; // Note: not rand!
4     constraint c1
```

```
 5     {
 6       foreach (array[i]) array[i].m_data > i;
 7     }
 8
 9     function void pre_randomize();
10        assert (std::randomize (m_array_size) with {m_array_size inside
    {[12:16]};});
11        array = new[m_array_size];
12        foreach (array[i]) array[i] = new();
13     endfunction
14
15   endclass : array_pre
```

另外，这个例子也可用post_randomiz方法来构造动态排列：

```
 1   class array_post;
 2     Packet array[]; // Note: not rand!
 3     rand int m_array_size;
 4     rand enum {kind_LITTLE, kind_MEDIUM, kind_BIG} m_kind;
 5
 6     constraint data_range
 7     {
 8       (m_kind == kind_LITTLE) -> m_array_size inside {[1:10]};
 9       (m_kind == kind_MEDIUM) -> m_array_size inside {[11:19]};
10       (m_kind == kind_BIG)    -> m_array_size inside {[20:23]};
11     }
12
13     function void post_randomize();
14        array = new[m_array_size];
15        foreach (array[i]) array[i] = new();
16        assert (std::randomize (array));
17     endfunction
18
19   endclass : array_post
```

约束架构

预设计/预测试结构

信息包为收集所有的很少改变的定义和在每一个测试的基础上变换一系列常量提供了便捷的机制。

```
 1   Example of using a package to set knobs
 2   package Config; // per test package
 3     parameter int width=16;
 4     parameter int num_tests=20;
 5   endpackage
 6   package protocol; // per design package
 7     class Packet;
 8       rand byte m_header;
 9       rand bit [Config::width-1:0] m_data;
10       static Packet scoreboard[$];
11       static int m_id;
12       function new(int d,byte h=m_id++);
13       m_header=h; m_data =d;
14       endfunction
15     endclass
16   endpackage
```

如下所示的另一种结构包可用来替换上面的结构包。

```
1  package Config; // per test package
2    parameter int width=32;
3    parameter int num_tests=512;
4  endpackage
```

设计约束

设计中要求一定的设计和有效的约束来保证正确的发挥功能。这些约束是不能关掉或忽略的。它们表示物理限制或设计不能处理的错误。例如一个补充输入或紧急编码输入就是这种情况。

给设计约束指定一个唯一的前缀来鉴别它们。建议约束块的前缀为：

assert_*<constraint_name>*.

错误注入

在约束块中用错误模式来组织约束以便它们能被忽略。尽管将约束组织在一个块中便于容易的打开或关闭，如果你在这个模块中只需修改一个约束，你就要重写剩下的所有约束。

高级话题

类群

大部分的测试环境需要生成大量的随机数并重复调用约束求解器。重复随机化一个类中的一个事件，然后复制它有一些重要的意义。每次构成一个新的类事件，则在开始调用randomize()之前也要构造其求解空间。当重复随机化一个类时，在那个类的一个事件上调用randomize()，然后创建该事件的副本，传送到测试的另一部分。在同一事件上重复调用randomize()也可以节约大量的额外计算。创建类事件的程序方法时面向对象概念的一部分，称作类群。使指向类的单一句柄静态随机化并可为测试环境的其它部分可用。一个静态句柄对整个测试通常是可视的。如果一个类有动态修改，或有状态约束，可在测试的任何地方修改句柄。通常，沿测试布置了随机发生器。所有的发生器都可以从中央位置进行控制。在下面的实例中，stimgen模块在第12行创建一个静态句柄，这个模块在根模块的顶部示例了两次。另一个根模块测试控

制了从第25行开始的块中的所有的激励发生器

```
 1  module top;
 2     addr_t a1,a2;
 3     data_t d1,d2;
 4
 5     stimgen t1(a1,d1);
 6     stimgen t2(a2,d2);
 7     DUT U1(a1,a2,d1,d2); // definition not shown
 8  endmodule : top
 9
10  module stimgen(output addr_t address, data_t data);
11
12  Packet handle =new();
13
14  initial forever #1 begin : stimulus
15     Packet p;
16     assert (handle.randomize);
17     p = handle.clone();
18     put(p); // task to wiggle ports definition not shown
19  end
20
21  endmodule : stimgen
22
23  module test;
24
25     initial begin
26        top.t1.handle.high_address = 10;
27        top.t1.handle.low_address = 5;
28        top.t2.handle.high_address = 5;
29        top.t2.handle.low_address = 1;
30        #10 $finish;
31     end
32
33  endmodule : test
```

类群以及继承使得激励发生器成为可以再利用的码。调用一个函数来构造类，而不是简单地调用静态新构造器；这就是一个抽象工厂。这个功能可以返回基类或派生类地一个事件。前面例子中第12行可以重写来调用一个抽象工厂在Packet和Word_Packet之间进行选择。

```
1  Packet handle;
2
3  initial stimulus_generator(Abstract_Packet1());
4
5  function automatic Packet Abstract_Packet1();
6    Packet p1;
7    Word_Packet p2;
8    randcase
9      1: begin p1 = new; return p1; end

10      1: begin p2 = new; return p2; end
11   endcase
12   endfunction
```

non-OO function, Abstract_Packet1也可以被重写，作为来自Packet的不同

类继承中一个类的方法。它也可以被其它的抽象工厂进行构造。抽象工厂返回的类事件称作工厂模式，因为它作为由那个类群创建的所有类的副本的蓝图。这种结构也可用来分别发生的模式和发生的激励，使得两者都可以再利用。

状态决定约束实例

尽管约束是可以声明的，也不要将它们看成是静态的。因为约束可以基于表达式，所以可以声明一个能改变整个基于特定变量的值的模拟的约束。考虑：

```
 1  class Xaction;
 2    int m_counter = 0;
 3    rand enum {READ, WRITE, NOP} m_busop;
 4    constraint startwriting {
 5                              if (m_counter < 10)
 6                                m_busop == WRITE;
 7                              else
 8                                m_busop dist {1:=NOP, 2:=READ};
 9                              }
10  endclass : Xaction
```

这里，busop变量被约束为READ或NOP，除非计数器变量达到10，这时busop被约束为WRITE。

同样的，约束也可以象限定的状态机一样被写入来更明确的改变被约束的值。状态变量用下面实例中第12行的pre_randomize方法进行更新。

```
 1  class myState;
 2    typedef enum {INIT, REQ, RD, WR,..} State_t;
 3    rand State_t m_state = INIT;
 4    State_t prev_state;
 5    bit req;
 6    constraint fsm {if (prev_state == INIT)
 7                      {state == REQ; req == 1};
 8                    if (prev_state == REQ && rdwr == 1)
 9                      state == RD;
10                    ...};
11    function void pre_randomize();
12     prev_state = m_state; // copy state value before randomizing
13    endfunction
14  endclass
15
16  myState ms1 = new;
17  ...

18   begin
19     st = ms1.randomize();
20   ...
21   end
```

第 10 章　基于断言的监视器

断言不仅是一个强有力的工具用来检查一个系统的行为，而且在收集已经发生的事务的覆盖率信息时也是非常有用的。至此，我们已经知道如何利用 SystemC 和 SystemVerilog的命令程序技术构造测试验证组件。在本章里，将集中讨论一个新的SystemVerilog断言结构，并举例说明如何创建断言监视器，它将命令和语句编程技术集成起来。

基于断言的监视器

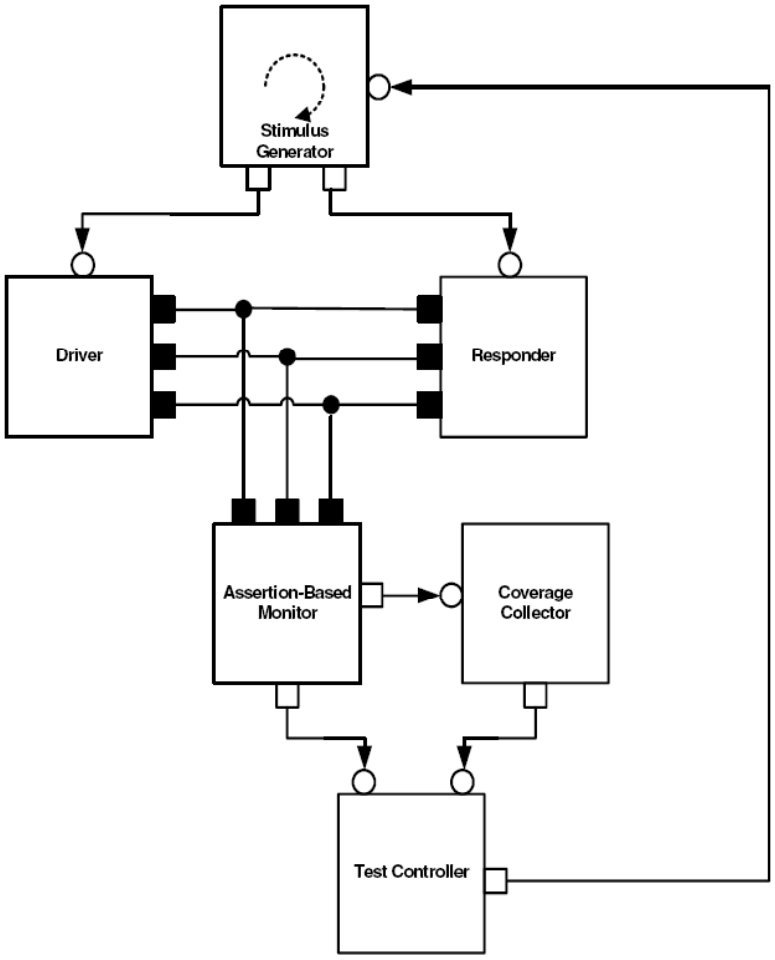


图10-1 基于断言监视器的测试

说明

一个基于断言的监视器有两个任务。首先它是一个将信号级别的行为转化成一个事务序列的验证组件，其次，基于断言的监视器是一个验证组件来确定DUT是否按照预想工作。使用SystemVerilog的properties，sequences，和cover constructs与发生在总线上的事件进行匹配是我们基于断言的监视器的独特价值所在。基于断言的监视器利用分析端口将信息传递给覆盖率采集器。覆盖率采集器统计各种发生的事件，其统计结果可以被测试控制器用来要么中断一个给定的测试，要么对一个给定的激励发生器调整参数。

我们已经知道，Scoreboard也是一个用来证实DUT的端到端行为的验证组件，比较而言，一个基于断言的监视器则通常用来证实DUT固有的界面行为。例如，在一个基于总线的SoC设计中，可能会在测试中采用一个基于断言的监视器来证实DUT界面是否满足特定的总线协议要求。基于断言的监视器通过一个分析端口将状态信息传递给测试控制器，由它处理通过基于断言的监视器识别出来的错误。

一个基于断言的监视器是一个被动的验证组件。通常，通过对比实测到的总线界面控制信号和特定协议的概念状态来构造一个基于断言的监视器。然后创建一套声明的Properties或Sequences，用来监视所观测到的接口行为。。

主要概念

- 基于断言的监视器是一个带时序的事务级组件，它监视发生在总线上的行为，并将信号级行为转化成事务信息流。
- 基于断言的监视器是一个带时序的事务级组件，它通过一套断言用作参照对象。

基于断言的协议监视器实例

本例中，从介绍一个简单的，非流水非流水的并行总线协议开始。下图示意了一个这样的设计。

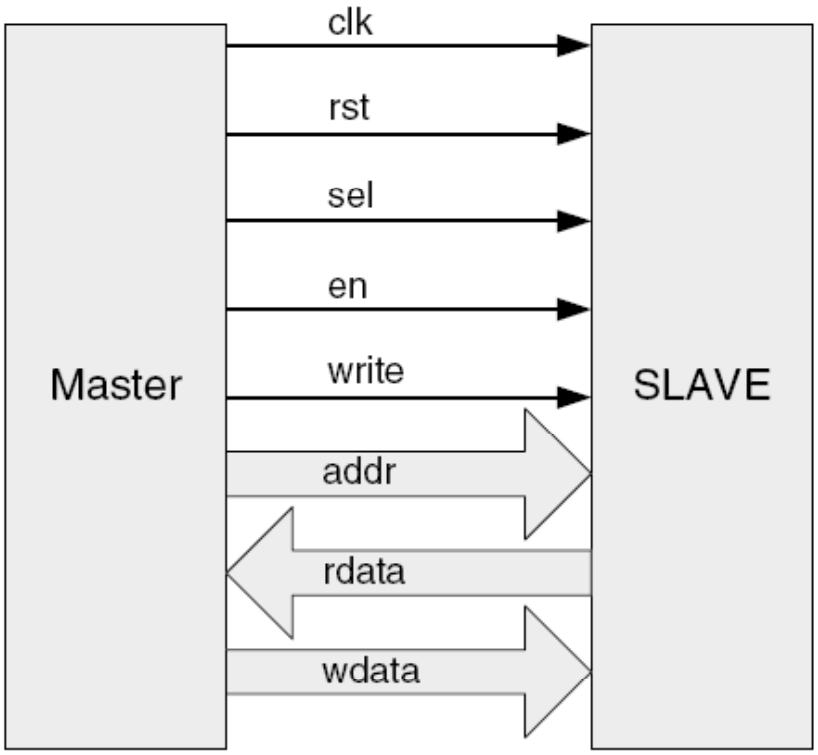


图10-2 简单的，非传递总线设计实例

表10-1中，，我们为本例的设计定义了非流水的总线信号。

表10-1 简单的非传递总线信号说明

名称	概述	说明
clk	总线时钟	用Clk的上升沿来同步所有总线传输的时间点
rst	总线复位复位信号	复位高有效总线复位信号
sel	从设备选择信号	这个信号表明选择了一个从设备
en	使能信号	用于控制总线访问时间

write	读写控制信号	高时，写访问；低时，读访问
addr[7:0]	地址	地址总线
rdata[7:0]	读数据总线	Write低时，驱动读数据
wdata[7:0]	写数据总线	Write高时，驱动写数据

我们用一个状态机图来描述总线协议的有效操作的过程。图10-3表明了它的状态流程图。

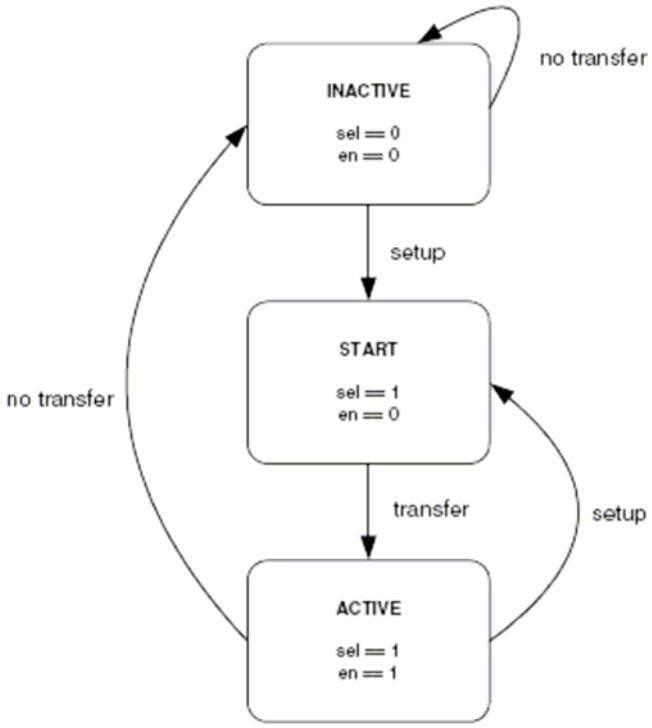


图10-3 简单的，非流水的总线设计实例

复位后，(即rst == 1)，这个简单的并行总线就被初始化为其默认的INACTIVE状态，也就是说sel和en都处于无效状态。为了开始一个数据传送，总线移入START状态，这里响应器（从机）选择信号sel由驱动器（主机）来置为有效，。总线在该状态只停留一个时钟周期，然后在下一个时钟上升沿移到ACTIVE状态。ACTIVE状态只停留一个时钟循环用于数据传输。然后，如果有另一个数据传输请求，sel信号保持有效（例如，一个突发操作），总线会移回到START状态。

另外一种情况，如果没有请求另外的数据传输请求，当驱动器取消从设备选

择和总线使能信号时，总线移回到INACTIVE状态。

非流水总线请求

在为我们简单的非流水并行总线创建基于断言的监视器之前，我们必须首先确认一个由自然语言实现的功能要求全面列表。我们从将这些要求分类成目录来开始，如表10-2所示

表 10-2. 非流水总线界面的必要条件

断言的名称	描述
总线合法转换	
a_state_reset_inactive	Reset为inactive后的初始状态
a_valid_inactive_transition	ACTIVE 状态不会紧跟 INATIVE
a_valid_start_transition	INACTIVE状态不会紧跟START
a_valid_active_transition	ACTIVE 状态不会紧跟ACTIVE
a_no_error_state	总线状态必须有效
总线稳定信号	
a_sel_stable	从START 到 ACTIVE从设备选择信号保持稳定
a_addr_stable	从START 到 ACTIVE地址保持稳定
a_write_stable	从START 到 ACTIVE读写控制信号保持稳定
a_wdata_stable	从START 到 ACTIVE数据保持稳定

SystemVerilog 实现细节

为了给我们简单的非流水总线创建一个我们的基于断言的监视器，我们开始创建一些模型代码来对比sel和en的控制信号的当前值（由总线驱动器驱动）和概念总线状态。然后我们写一套断言通过监控违法总线状态转换来检测协议违反情况。

```
60
61        if (monitor_mp.rst) begin
62          bus_reset    = 1;
63          bus_inactive = 1;
64          bus_start    = 0;
65          bus_active   = 0;
66          bus_error    = 0;
67        end
68        else begin
69          bus_reset    = 0;
70          bus_inactive = ~monitor_mp.sel & ~monitor_mp.en;
71          bus_start    =  monitor_mp.sel & ~monitor_mp.en;
72          bus_active   =  monitor_mp.sel &  monitor_mp.en;
73          bus_error    = ~monitor_mp.sel &  monitor_mp.en;
74        end
75
file:
topics/10_assertions/protocol_transactors_svm/protocol_monitor_mod.sv
```

现在，我们为总线接口要求写断言。

```
79
80      // ---------------------------------------------
81      // REQUIREMENT: Bus legal states
82      // ---------------------------------------------
83
84      property p_state_reset_inactive;
85        @(posedge monitor_mp.clk) disable iff (bus_reset)
86           $past(bus_reset) |-> (bus_inactive);
87      endproperty
88      assert property (p_state_reset_inactive) else begin
89        status = new();
90        status.set_err_trans_reset();
91        if (status_af != null) status_af.write(status);
92      end
93
94      property p_valid_inactive_transition;
95        @(posedge monitor_mp.clk) disable iff (bus_reset)
96          ( bus_inactive) |=>
97            (( bus_inactive) || (bus_start));
98      endproperty
99      assert property (p_valid_inactive_transition) else begin
100       status = new();
101       status.set_err_trans_inactive();
102       if (status_af != null) status_af.write(status);
103     end
104
```

```
105     property p_valid_start_transition;
106       @(posedge monitor_mp.clk) disable iff (bus_reset)
107         (bus_start) |=> (bus_active);
108     endproperty
109     assert property (p_valid_start_transition) else begin
110       status = new();
111       status.set_err_trans_start();
112       if (status_af != null) status_af.write(status);
113     end
114
115     property p_valid_active_transition;
116       @(posedge monitor_mp.clk) disable iff (bus_reset)
117         (bus_active) |=>
118           (( bus_inactive ) || (bus_start));
119     endproperty
120     assert property (p_valid_active_transition) else begin
121       status = new();
122       status.set_err_trans_active();
123       if (status_af != null) status_af.write(status);
124     end
125
126     property p_valid_error_transition;
127       @(posedge monitor_mp.clk) disable iff (bus_reset)
128         (~bus_error);
129     endproperty
130     assert property (p_valid_error_transition) else begin
131       status = new();
132       status.set_err_trans_error();
133       if (status_af != null) status_af.write(status);
134     end
135
file:
topics/10_assertions/protocol_transactors_svm/protocol_monitor_mod.sv
```

我们的第一个要求(p_state_reset_inactive)表示，在复位以后，总线必须初始化到INACTIVE状态（这意味着sel信号和en都不被置位）。这定义了在复位以后有效转换的总线状态。同样，我们也把所有有效总线转换要求写入断言，这是基于我们前面定义的协议概念上的状态机的。

我们的最后一套断言规定当总线在START 和ACTIVE状态之间转换时，总线控制，地址和写入的数据信号必须保持稳定。。

```
137
138     // ---------------------------------------------
139     // REQUIREMENT: Bus must remain stable
140     // ---------------------------------------------
141
142     property p_bsel_stable;
143       @(posedge monitor_mp.clk) disable iff (bus_reset)
144         (bus_start) |=> $stable(bus_sel);
145     endproperty
146     assert property (p_bsel_stable) else begin
147       status = new();
148       status.set_err_stable_sel();
```

```
149       if (status_af != null) status_af.write(status);
150     end
151
152     property p_baddr_stable;
153       @(posedge monitor_mp.clk) disable iff (bus_reset)
154         (bus_start) |=> $stable(bus_addr);
155     endproperty
156     assert property (p_baddr_stable) else begin
157       status = new();
158       status.set_err_stable_addr();
159       if (status_af != null) status_af.write(status);
160     end
161
162     property p_bwrite_stable;
163       @(posedge monitor_mp.clk) disable iff (bus_reset)
164         (bus_start) |=> $stable(bus_write);
165     endproperty
166     assert property (p_bwrite_stable) else begin
167       status = new();
168       status.set_err_stable_write();
169       if (status_af != null) status_af.write(status);
170     end
171
172     property p_bwdata_stable;
173       @(posedge monitor_mp.clk) disable iff (bus_reset)
174         (bus_start) && (bus_write) |=> $stable(bus_wdata);
175     endproperty
176     assert property (p_bwdata_stable) else begin
177       status = new();
178       status.set_err_stable_wdata();
179       if (status_af != null) status_af.write(status);
180     end
181
file:
topics/10_assertions/protocol_transactors_svm/protocol_monitor_mod.sv
```

如果检测到错误，则通过anaylsis_fifo（例如：status_af）将故障信息反馈给测试控制器，测试控制器则打印错误信息并中止仿真进程。

对于我们的覆盖率Property，我们写入一个sequence（在property内）来跟踪总线读写突发的大小，如下所示：

```
183
184     property p_burst_size;
185       int psize;
186
187       @(posedge monitor_mp.clk)
188           ({bus_inactive}, psize=0)
189       ##1 ({bus_start, psize++, build_transaction(psize)}
190            ##1 (bus_active))[*1:$]
191       ##1 (bus_inactive);
192     endproperty
193
194     cover property (p_burst_size);
195
```

file:

topics/10_assertions/protocol_transactors_svm/protocol_monitor_mod.sv

局部变量psize，已经在上面的Sequence定义，每增加一次新的数据字传输，就会转换为一个给定的总线事务突发。该突发的大小和类型（如读或写）就被传递给build_transaction函数，如下所示：

```
197
198     function void build_transaction(int psize);
199       protocol_transaction tr;
200
201       tr = new();
202       if (bus_write) begin
203         tr.set_write();
204         tr.data = bus_wdata;
205       end
206       else begin
207         tr.set_read();
208         tr.data = bus_rdata;
209       end
210       tr.burst_count = psize;
211       tr.addr        = bus_addr;
212
213       if (trans_af != null) trans_af.write(tr);
214     endfunction
215
file:
topics/10_assertions/protocol_transactors_svm/protocol_monitor_mod.sv
```

build_transaction函数传递捕捉到的突发的大小和类型信息给analysis fifo，该analysis fifo在我们测试的顶部模块中已经被定义过：

```
43      // channels
44
45      analysis_fifo #( protocol_transaction )
46                                  trans_fifo = new("trans_fifo");
47      analysis_fifo #( protocol_status )
48                                  status_fifo = new("status_fifo");
49
50      tlm_fifo #( protocol_transaction ) m_response_fifo =
51                                          new("m_response_fifo");
52      tlm_fifo #( protocol_transaction ) s_response_fifo =
53                                          new("s_response_fifo");
file: topics/10_assertions/01_monitor_svm/top.sv
```

覆盖率采集器就测量各种突发事务的大小，当各种突发的大小达到极限值时，覆盖率采集器就中止激励发生器，并完成整个仿真过程。

基于断言检查器的测试

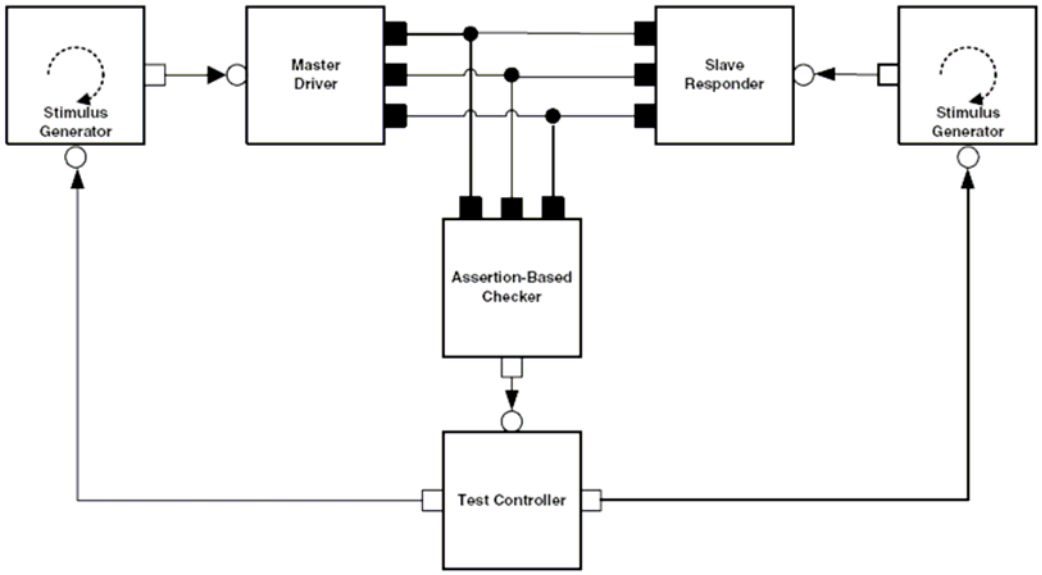


图10-4基于断言检查器的测试

说明

创建一个基于断言的检查器提供了一个有效的定位总线界面故障的途径。在一个测试中，一个基于断言的检查器可以用来警报一个违反协议给测试控制器，并使测试控制器采取适当措施。

在本例中，我们采用了一个驱动器和一个响应器来模拟我们的无流水总线活动。注意在我们的测试例中没有DUT。等以后驱动器或响应器成为可得到的话，它就会被一个真正的DUT取代。但是本例说明如何在提供真正的RTL设计之前构造和调试一个测试。

主要概念

- 将基于断言的检查器内嵌到完整的测试中。
- 驱动器和响应器可以联合起来构造一个可以工作的测试，该测试允许调试我们的基于断言的检查器（不需要一个真正的RTL设计）。

SystemVerilog 实现细节

对于我们基于模块的测试实例，所有的验证组件都被构造成模块。通道是定义通讯语义的组件，它被用Class类来实现，并用来连接响应发生器到驱动器和响应检查器。

在本例中，我们在top.sv模块中用了2个analysis_fifo和tlm_fifo来实现通道，如下所示：

```
43      // channels
44
45      analysis_fifo #( protocol_transaction )
46                                  trans_fifo = new("trans_fifo");
47      analysis_fifo #( protocol_status )
48                                  status_fifo = new("status_fifo");
49
50      tlm_fifo #( protocol_transaction ) m_response_fifo =
51                                          new("m_response_fifo");
52      tlm_fifo #( protocol_transaction ) s_response_fifo =
53                                          new("s_response_fifo");
file: topics/10_assertions/01_monitor_svm/top.sv
```

然后，通道tlm_fifo将激励发生器和驱动器及响应器连接起来，如下所示：

```
55      // verification components
56
57      protocol_stimulus_mod
58                  stimulus( .m_response_fifo( m_response_fifo ) ,
59                            .s_response_fifo( s_response_fifo ) );
60
61      protocol_driver_mod
62                  master ( .response_fifo(m_response_fifo),
63                           .driver_mp( nonpiped_bus.driver_mp ) );
64
65      protocol_responder_mod
66                  slave ( .response_fifo(s_response_fifo),
67                          .responder_mp( nonpiped_bus.responder_mp ) );
file: topics/10_assertions/01_monitor_svm/top.sv
```

接口：

为了创建一个可以把测试驱动器和响应器及基于断言的监视器连接起来的总线，我们使用SystemVerilog的Interface，如下所示：

```
24    interface protocol_pins_if( input clk , input rst );
25
26      parameter int DATA_SIZE = 8;
27      parameter int ADDR_SIZE = 8;
```

```
28
29      bit       sel;
30      bit       en;
31      bit       write;
32      bit [DATA_SIZE-1:0] wdata;
33      bit [DATA_SIZE-1:0] rdata;
34      bit [ADDR_SIZE-1:0] addr;
35
36      modport driver_mp (
37       input        clk , rst ,
38       output       sel , en , write , addr ,
39       output       wdata ,
40       input        rdata
41            );
42
43      modport responder_mp (
44       input        clk , rst ,
45       input        sel , en , write , addr ,
46       input        wdata ,
47       output       rdata
48           );
49
50      modport monitor_mp (
51       input        clk , rst ,
52       input        sel , en , write , addr ,
53       input        wdata ,
54       input        rdata
55             );
56
57    endinterface
file: topics/10_assertions/protocol_transactors_svm/protocol_pins_if.sv
```

有了这个单独的Interface描述，我们就可以越过多个事务级转换器复用总线定义。这意味着如果我们对Interface进行了改变，它会自动地被反映到所有相连的共享这个Interface的事务级转换器。

例如，在我们的测试驱动器事务级转换器中，我们用SystemVerilog 的driver_mp modport，如下所示：

```
23    module protocol_driver_mod(
24      input tlm_fifo #( protocol_transaction ) response_fifo ,
25      interface.driver_mp driver_mp
26    );
file:
topics/10_assertions/protocol_transactors_svm/protocol_driver_mod.sv
```

这样，所有我们总线接口的指向都通过driver_mp modport建立。例如，驱动器会使用modport分配器驱动总线interface的sel和en信号，如下所示：

```
145   // Setup bus controls for write
146
147                 if ( m_transaction.is_write() ) begin
148                   driver_mp.write <= 1;
149                   driver_mp.wdata <= m_transaction.get_data() ;
```

```
150                   end
151                   else begin
152                     driver_mp.write <= 0;
153                   end
file:
topics/10_assertions/protocol_transactors_svm/protocol_driver_mod.sv
```

同样，我们的响应器事务级转换器也可以通过我们的interface使用modport responder_mp

```
23
24    module protocol_responder_mod(
25      input tlm_fifo #( protocol_transaction ) response_fifo ,
26      interface.responder_mp responder_mp
27    );
28
file:
topics/10_assertions/protocol_transactors_svm/protocol_responder_mod.sv
```

响应器内部所有的总线界面的指向都通过responder_mp modport构造.

最后，所有基于断言的监视器都从我们Interface使用monitor_mp modport，如下所示：

```
23
24    module protocol_monitor_mod(
25      interface.monitor_mp monitor_mp ,
26      input analysis_fifo #(protocol_transaction) trans_af ,
27      input analysis_fifo #(protocol_status) status_af
28    );
29
file:
topics/10_assertions/protocol_transactors_svm/protocol_monitor_mod.sv
```

驱动器和响应器模拟总线通讯

驱动器(protocol_driver_mod.sv)和响应器(protocol_responder _mod.sv)，每一个都对前面定义的用于无流水总线协议的概念状态机进行建模，并按照它自己对协议的理解，产生和驱动相应的数据和控制信号。我们的基于断言的监视器确保驱动器和响应器之间的通讯协议定义的一致，并将信号级和Cycle精确的总线操作转换成事务变化，然后通过一个analysis port输出。激励发生器事务器给驱动器（对于总线写入）和响应器（对于总线读出）提供事务。驱动器和响应器则提供必要的、由我们的协议定义的时序的和适当的控制信号。

作为实例，下面的代码演示了驱动器通过接收来自激励发生器（通过通道）和总线控制的数据和地址来实现一个总线写入的操作。

```
62 // Get stimulus generator output for transaction
63
64              if( response_fifo.try_get( m_transaction ) ) begin
65
66    // If not idle, setup bus controls to transition to a START state
67
68                driver_mp.write <= 0;
69                driver_mp.en <= 0;
70
71                if ( ~m_transaction.is_idle() ) begin
72                  driver_mp.sel <= 1;
73                  m_state <= START;
74                end
75                else begin
76                  driver_mp.sel <= 0;
77                  m_state <= INACTIVE;
78                end
79
80                driver_mp.addr  <= m_transaction.get_addr() ;
81
82    // Setup bus controls for write
83
84                if ( m_transaction.is_write() ) begin
85                  driver_mp.write <= 1;
86                  driver_mp.wdata <= m_transaction.get_data() ;
87                end
88              end
file:
topics/10_assertions/protocol_transactors_svm/protocol_driver_mod.sv
```

同样，下面的代码则演示了响应器通过对来自器激励发生器的数据接收来完成一个总线读出的操作。

```
54
55    // If read transaction, get generator output for transaction
56
57            if (~responder_mp.write) begin
58
59    // Get stimulus generator output for read transaction
60
61              if( response_fifo.try_get( s_transaction ) ) begin
62
63                responder_mp.rdata <= s_transaction.data;
64
file:
topics/10_assertions/protocol_transactors_svm/protocol_responder_mod.sv
```

附录 A SystemVerilog AVM 库

引言

本手册中的实例都使用了一个内在的类和效用的效用库来构造验证组件（就是所谓的VIP）。本附录详细介绍了SystemVerilog 库，所有的基本类，它们的成员和如何使用它们。

报告

报告工具提供了在各个可变级别显示信息并且当这些信息被显示后进行控制的途径。

基本报告方法

报告是由VIP通过使用如下报告方法中的一种发布的：

```
avm_report_message( string id, string mess, int verbosity = 3 );
avm_report_warning(string id, string mess, int verbosity = 2 );
avm_report_error( string id, string mess, int verbosity = 1 );
avm_report_fatal( string id, string mess, int verbosity = 0 );
```

每个报告调用的报告句柄决定了要发生的特定行为。顶级报告句柄给从模块，接口或avm_env子集发布的报告提供缺省设置。另外，每个avm_named_component都有自己的报告句柄。通过缺省，给每个报告句柄一个标准结构。

冗余级

一个报告句柄都有一个最大的冗余级。如果一个报告的冗余级比最大冗余级要大，它就会被忽略掉。一个报告句柄查看与成对(severity id)关联的动作，然后处理这些报告。

任何句柄的最大的冗余级可用如下方法改变：

```
function void
avm_set_verbosity_level( int verbosity_level ,
input avm_recursion_e recurse = THIS_LEVEL_ONLY );
```

在这里以及后来的“configuration setting”方法中，有一个可操作的最终参数，该参数有两个值：THIS_LEVEL_ONLY (缺省值)或 THIS_LEVEL_AND_BELOW。前者只将设置用到这个对象的报告句柄，而后者还将设置用于所有对象的子集，按层次一直回归下去。

动作

除了简单的显示信息，调用报告方法可能会调用另外的动作。可能发生的动作都通过列举型action_type定义。

```
typedef enum action

{
   NO_ACTION= 4'b0000 ,
 DISPLAY = 4'b0001 ,
  LOG = 4'b0010 ,
  COUNT = 4'b0100 ,
  EXIT = 4'b1000
```

} action_type;

表 A-1.报告action

动作标识符	相应的动作
DISPLAY	在标准输出上显示报告
LOG	发送报告给相关文件
COUNT	在激活前计数到极值
EXIT	立即退出模拟

这些action_type一起使用字节或操作符‘|’来产生关联动作。与每个severity相关的缺省action为：

```
severity_actions[MESSAGE] = DISPLAY | LOG;
severity_actions[WARNING] = DISPLAY | LOG;
severity_actions[ERROR] = DISPLAY | LOG | COUNT;
severity_actions[FATAL] = DISPLAY | LOG | EXIT;
```

设置动作

可以用如下三种方法中的一种改变行为：

```
function void
avm_set_severity_action( input severity s , input action a ,
input avm_recursion_e recurse = THIS_LEVEL_ONLY );

function void
avm_set_id_action( input string id , input action a ,
input avm_recursion_e recurse = THIS_LEVEL_ONLY );

function void
avm_set_severity_id_action( input severity s , input string id , input
action a ,
```

```
input avm_recursion_e recurse = THIS_LEVEL_ONLY );
```

第一个与一个特殊级的action关联，第二个与id关联，第三个与对(severity,id) 关联。与id关联的action覆盖与severity关联的action，与pair(severity,id)关联的action覆盖与severity或id关联的action。

文件输出

如果一个报告有与它关联的储存的action，报告句柄就会看这里是否有关联的文件句柄来储存这个报告。你可以用severity，id或pair（severity,id）将一个文件句柄与报告句柄联系起来成为一个整体（缺省文件句柄）。报告句柄不打开或关闭文件：是你来做这些。既然文件句柄是普通的verilog文件句柄，如果你想同一个报告进入许多不同的文件，你可以将同一个句柄里许多文件结合起来。

将文件句柄与报告句柄关联的方法如下：

```
function void
avm_set_default_file( input FILE f,
input avm_recursion_e recurse = THIS_LEVEL_ONLY );

function void
avm_set_severity_file( input severity s, input FILE f,
 input avm_recursion_e recurse = THIS_LEVEL_ONLY  );

function void
avm_set_id_file( input string id , input FILE f,
input avm_recursion_e recurse = THIS_LEVEL_ONLY );

function void
avm_set_severity_id_file( input severity s, input string id,
```

```
input FILE f,
input avm_recursion_e recurse = THIS_LEVEL_ONLY );
```

与severity关联的文件覆盖这个报告句柄的缺省文件，与id关联的文件覆盖与severity关联的文件，与pair（severity,id）关联的文件覆盖任何其它的关联。如果给这个报告句柄没有调用任何方法，就不会有文件输出，因为缺省文件句柄“default”是0。

报告格式化程序

一个报告格式化程序确定action是如何进行的和报告是什么格式。它还储存max_quit_count，这里控制在退出仿真时，看到了多少COUNT事件。

使用格式

通常，你可以使用报告格式化程序设置max_quit_count变量，用总结方法获得有用的统计表：

```
class avm_report_formatter;
    static int max_quit_count = 1;
    static function void summarize( FILE f = 0 );
   …

endclass
```

高级话题

本节说明快捷工具组合的特点。

你也可以用自己的方法，写入自己的报告格式化程序来覆盖标准的报告格式化程序：

```
function void set_report_formatter(avm_report_formatter rf);
```

新的报告格式化程序必须继承avm_report_formatter，并可以用来覆盖两个虚方法中的任何一种：

```
class avm_report_formatter;
   ..
    virtual function
    string compose_message( severity s , string name , string id , string

  mess );

    virtual function void
    process_report( action a , FILE f , string m );

  endclass
```

在执行报告解码action和正确action时，Compose_message是实际的从其元素部分格式化报告的方法。

构造模块

构造测试的三个主要AVM类分别是：avm_env, avm_named_component 和 avm_verification_component。在本节中，我们解释每一个类，以及如何运用这些类来构造验证组件和测试平台。

avm_named_component

在AVM中，所有验证组件都是从指定的组件派生出来的。也可以这样说，所有的验证组件都是指定的组件。一个指定的组件管理测试平台的分层。它有一个层次化的名字，一个可选择的父类，一个（也可能是空）子类列表，一个本地报告句柄，以及管理其子类连接性的方法。

构造器

```
function new(string name, avm_named_component parent= null);
```

构造器有两个判据：名称和可选择的父类。传递给构造器的字符串是一个本地实例名（它必须是唯一的）：avm_named_component 自动地体现了完整的

层次名。

只有avm_env的子集（它是顶级组件，或没有父类的组件）可以省略父类参数，否则就要用“this”：

```
class my_env extends avm_env;
    pipelined_bus_hiearchical_component m_component;
   function new;

        m_component = new(“hierarchical_component”); // no parent
  specified
   endfunction
  endclass

class pipelined_bus_hierarchical_component extends avm_named_component;
    address_phase_component m_address_component;
   data_phase_component m_data_component;

 function new( string name , avm_named_component parent = null );
        super.new( name , parent); // register name and parent with named
  component
        m_address_component = new( “address_component” , this );
        m_data_component = new( “data_component” , this );

    endfunction
  endclass
```

这将产生两个子级实例，叫做*hierarchical_component. address_component* 和 *hierarchical_component.data_component*.

当这些指定的对象被创建后，它们就自动的插入一个任何时候都可以被访问的库。这个库叫做s_named_object_repository，且定义如下：

```
class avm_named_component;

  …
  static avm_named_component s_named_object_repository[string];
 …

endclass
```

这个库对于从环境类(avm_env)的方法来形成特定的组件特别有用。

报告方法

Avm_named_component 有一个可在任何组件中被覆盖的report_ method，这个报告方法可以在执行测试的任何时候调用，但是当环境类完成了执行阶段后，它往往被自动调用。一个典型的用途就是在测试的最后报告记分板的状态。

```
class avm_named_component;
    …
     virtual function void report;
    return;
     endfunction
    …
   endclass

 class my_scoreboard extends avm_named_component;
     local int m_successes , m_failures;
    …
```

```
  virtual function void report;

    string report_str;

          $sformat( report_str , “%d success and %d failures” ,
m_successes ,
m_failures );
     avm_report_message(“my scoreboard report” , report_str );
  endfunction
endclass
```

报告处理

每个avm_named_component都有自己的报告句柄和构造器一节介绍所有方法。发送到报告句柄的名称是实例的全部层级名称，如在构造器一节中的一样，THIS_LEVEL_AND_BELOW设置被往前发送给子类，在构造过程中，它将这个指定的组件登记为它的父类。

各层的连通性

本节假定熟悉纯虚拟接口，端口和输出口的概念。这些概念都在本附录的包装，多层级，端口/输出口译码惯例一节中进行了解释。

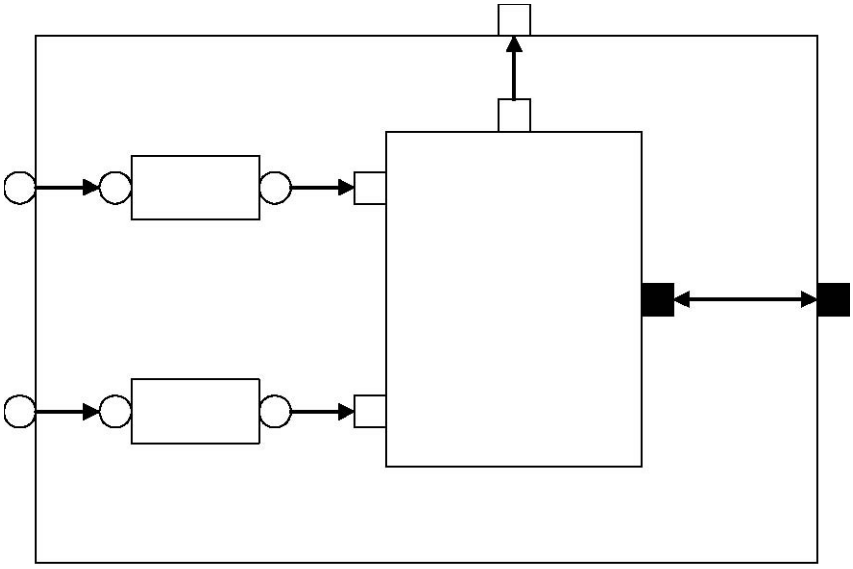


Figure A-1. Generic Hierarchical Component

图A-1给出了通用层级组件。它包括一个“inner core”和两个通道（可能是tlm_fifos）。内部核心有一个通过层次与DUT相连的虚拟接口。它从两个通道中引入事务，put接口被输出给测试平台的其它部分，以便其它的VIP可以发送事务给这个VIP。它也有一个端口用来放感兴趣的事务给外部环境。

有三种关联，如下所示：

- 对象子类之间的内部关联，它是从端口到输出口
- VIP子类提供的接口，它必须被输出给外部
- VIP子类请求的接口，它必须从外部输入。

avm_named_component提供了三种独立的方法，在如下三类连通性中必须使用它们：

```
protected virtual function void connect;
protected virtual function void export_connections;
protected virtual function void import_connections;
```

上图代码的构成实例如下所示：

```
class generic_vip extends avm_named_component;
   tlm_put_if #( transaction ) put_port;
   tlm_put_if #( transaction ) put_export_1 , put_export_2;
   virtual bus_if m_bus_if;
   local inner_core m_inner_core;
   local tlm_fifo #( transaction ) m_channel_1 , m_channel_2;
```

```
    function new( string name , avm_named_component parent = null );
        super.new( name , parent );
    // create all the child components, with parent == this
        m_inner_core = new("inner_core" , this );
        m_channel_1 = new("channel_1" , this );
        m_channel_2 = new("channel_2" , this );
    endfunction
    function void connect; // internal connections : child.port =
            child.export
        m_inner_core.get_port_1 = m_channel_1.get_export;
        m_inner_core.get_port_2 = m_channel_2.get_export ;
    endfunction
    function void export_connections; // export connections : export =
            child.export
        put_export_1 = m_channel_1.put_export;
        put_export_2 = m_channel_2.put_export;
    endfunction
    function void import_connections; // import connections : child.port
=port
        m_inner_core.put_port = port_port;
        m_inner_core.m_bus_if = m_bus_if ;
    endfunction
endclass
```

连接export_connections 和 import_connections调用的顺序是用户看不见的方法avm_env 和 avm_named_component控制的。其顺序为 export_connections, connect, import connections。Export_ connections 和 connect是从下往上执行的，而import_connections则是从上往下执行的。

avm_verification_component

Avm_verification_component扩展了avm_named_component。它增加到指定组件上的唯一特点就是能在仿真开始时，自动地停止一个运行任务。它也给挂起，复用和取消运行方法提供可重载的方法以及它可能并行运行的任何事务。验证组件的一个典型用途就是一个事务器，它需要在时间为0时启动，并继续到仿真结束，尽管一些记分板可能也需要这个容量。一般来说，激励发生器

不是验证组件，因为它们需要在时间不为0的时候启动，而且经常有一个限定的结束点。

```
virtual class avm_verification_component extends avm_named_component;
    …
     local process m_main_process;

   pure virtual task run;

  virtual function void suspend;
    m_main_process.suspend;
    endfunction

  virtual task resume;
    m_main_process.resume;
    endtask

  virtual function void kill;
    m_main_process.kill;
    endfunction

   endclass
```

一个事务器（或其它的扩展avm_verification_component的组件）必须提供一个运行任务，因为这是它的定义属性。典型地，这是一个无限循环，它会等待在一个时钟的边缘，并执行一个状态机。

M_main_process是这个运行任务的过程id：它有用户的非可视方法设置。在这个过程中挂起，复用和取消方法操作是缺省实现的。如果运行方法与多个过程同时执行，这些过程也将用这三个方法中非缺省实现。

avm_env

顶层环境对象必须扩展avm_env。测试的创建者在avm_env的子类中声明了一套组件，并在这个子类构造器中创建它们。构造完成后，唯一的用户可视的公共方法就是任务do_test，它调用测试平台各阶段的顺序。

avm_env阶段为：

1. 创建（在子类的构造器中）
2. 连接
3. 配置
4. 执行
5. 报告

avm_env的子类必须提供连接和执行方法；它可能提供配置和报告方法。在正确的时间，用do_test方法发出调用各种连接，运行，报告和取消方法。

```
virtual class avm_env;
virtual task do_test;
avm_named_component::export_top_level_connections;
connect();
avm_named_component::import_top_level_connections;
configure;
avm_verification_component::run_all;
execute;
report;
avm_named_component::report_all;
avm_verification_component::kill_all;
endtask
pure virtual function void connect;
virtual function void configure; // calls to config interfaces of
testbench components
return;
endfunction
pure virtual task execute; // execute phase
virtual function void report; // report
avm_report_message("avm_env" , "Finished Test");
return;
endfunction
```

```
endclass
```

核心 AVM 类和组件

avm_transaction

avm_transaction类是一个基类，但是avm_transaction的大部分需求都不受语法所限。为了允许avm和tlm库工作，我们需要方法来打印，比较和复制事务。只有第一项是在基类中用方法进行实际的事务。

对任何给定的事务T，我们需要做四件事情：

- 从avm_transaction继承
- 执行虚函数字符串convert2string
- 执行函数位comp(input T t)
- 执行函数T clone();[1]

1. *在C++中，对于<<, ==和复制构造器，这三种方法是相等的。这三种方法都按照需要可深可浅。*

方法convert2string返回一个描述事务的字符串。它经常与报告库一起使用。Comp方法是用来确定两个对象是否相等。它返回一个位表明事务t和this是否现等。Comp方法经常用到记分板上，这里你可以对两个数据流进行一些比较。比如：

```
if( a.comp( b ) ) begin
avm_report_message("transactions match" , a.convert2string() );
end
else begin
$sformat( error_str , "%s does not match %s" , a.convert2string() ,
b.convert2string() );
avm_report_warning("transaction mismatch" , error_str );
end
```

Clone方法给新分配的this复制返回句柄。Clone方法在激励发生器中尤其有用，在将事务发送到验证环境的其余组件时，我们需要在激励发生器中创建一个新复制的事务。如：

```
function void generate_stimulus( input transaction generator );
transaction t;
for( int i = 0; I < 10; i++ ) begin
assert( generator.randomize() );
t = generator.clone();
stimulus_port.put( t );
end
endfunction
```

如果每次我们打乱发生器时不Clone它，每个新的事务就会覆盖前面的那个。如果前一个事务仍然在验证环境的某处被利用，－如，在用到总线上以前的事务器中，或在记分板中等待与其它事务进行比较－这将非常糟糕。

convert2string, comp和复制方法都假定用tlm库中的其它组件存在。

avm_stimulus

Avm_stimulus既是一个普通的随机单向激励发生器，又是一个可以用作其它激励发生器的基础的原型。它假定两个参数trans_type以及发生器类都有复制方法。

在设计avm_stimulus时有许多关键点：

- 它不是一个简单的类群：它是一个具有put端口的验证组件，准备与通道相连，如tlm_fifo。
- trans_type参数是发出put端口的事务的类型。
- 在trans_type被发出put端口前，复制了一个新的trans_type副本。
- trans_type的约束应尽可能少。
- trans_type事务由trans_type的子类产生，它通常会有更多的测试细节约束加到这个子类中。因此，一般说来，trans_type与做激励发生的类不是同一个。
- generate_stimulus方法是虚的以允许在约束随机发生以前或以后加入直接测试。

一个 avm_stimulus 例子:

```
class write_request extends mem_request
    constraint write_only { this.m_type == MEM_WRITE; }
endclass
class read_request extends mem_request
    constraint read_only { this.m_type == MEM_READ; }
endclass
avm_stimulus #( request_t ) m_stimulus;
task execute;
    m_stimulus.generate_stimulus( m_write_gen , 10 );
    m_stimulus.generate_stimulus( m_read_gen , 10 );
    fork
        m_stimulus.generate_stimulus;
        terminate;
    join
endtask
task terminate;
    #100;
    m_stimulus.stop_stimulus_generation;
endtask
```

上例中的Mem_request没有任何约束，因为请求中的所有值都是有效的。我们提供两个本地约束类，只分别产生写和读的事务。然后产生一串10个写，10读以及一串未被绑定的受约束的请求，除非在中止任务中规定了推出时间。在上例的代码中，我们只是利用了SystemVerilog中的过程控制能力来产生串行或并行的激励。也可以通过使用很好的过程事务方法挂起，复用和杀掉来获得更好的激励控制，这一点没有举出实例。

analysis_if 和 analysis_port

分析接口是一个参数化的虚类，它只有一个纯虚函数：

```
virtual class analysis_if #( type T = int );
   pure virtual function void write( input T t );
endclass
```

接口用来广播来自组件的感兴趣的事务，如监视器。因为监视器什么时候都

不能阻塞，写入方法就是一个函数。通常说来，因为我们需要将0，1或许多用户与一个专门的监视器相连，所以提供了分析端口。一个分析端口只是围绕分析端口列表的一薄层。实际上，它是一个观测模式的验证特定实现。

```
class analysis_port #( type T = int ) extends analysis_if #( T );
    local analysis_if #( T ) if_list[$];
    function void register( input analysis_if #( T ) i );
        if_list.push_back( i );
    endfunction
    function void write( input T t );
        analysis_if #( T ) temp;
        for( int i = 0; i < if_list.size; i++ ) begin
            if_list[i].write( t );
        end
    endfunction
endclass
```

一个监视器例示和创建了一个或多个分析端口。然后一个或多个用户可能与这些分析端口的每一个去注册来接收广播的事务。

```
class monitor extends avm_verification_component;
analysis_port #( transaction ) ap;
function new( string name , avm_named_component parent = null );
super.new( name , parent );
ap = new;
endfunction
task run;
transaction t;
forever begin
t = new;
…
ap.write( t );
endclass endtask
class my_env extends avm_env;
monitor m_monitor_1 , m_monitor_2;
avm_in_order_class_comparator #( transaction ) m_comparator;
coverage_object m_coverage;
// constructor omitted
function void connect;
m_monitor_1.ap.register( m_comparator.before_export );
m_monitor_1.ap.register( m_coverage.analysis_export );
m_monitor_2.ap.register( m_comparator.after_export );
```

```
endfunction
endclass
```

上面的代码将比较器的一侧与一个监视器相连，另一侧与另一个监视器相连。它还将第一个监视器与覆盖率对象相连。分析端口有一个分析接口列表和一个执行分析接口本身。但对分析端口进行层次化绑定时，这是很有用的。

```
class hierarchical_monitor extends avm_named_component;
    analysis_port #( transaction ) ap;
    local internal_monitor m_internal_monitor;
    function new( string name , avm-named_component parent = null );
        super.new( name , parent );
        ap = new;
    m_internal_monitor = new(‘internal monitor” , this );
    endfunction
    function void import_connections;
        m_internal_monitor.ap.register( ap );
    endfunction
endclass
```

avm_in_order_comparator

avm_in_order_comparator组件比较两个事务流。这些事务被一个analysis_port广播，因此avm_in_order_comparator提供两个分析输出口将这两个数据流连接起来。在内部，这两个数据流被储存在分析fifo中，只要两个fifo中有数据，就将这两个数据流成对的进行比较。

avm_in_order_comparator的报告方法报告匹配数和不匹配对。实际上，这里有大量的avm_in_order_comparator变量：

- avm_in_order_class_comparator 比较类流，并依赖现有的事务类型中的comp方法。
- avm_in_order_built_in_comparator用由SystemVerilog 定义的== operator来比较内嵌类型流。
- avm_in_order_class_comparator_external比较类流，但是假定fifo是比较器的外部。
- avm_in_order_built_in_comparator_external 比较内嵌类型流，但是

假定fifo是比较器的外部。

- avm_in_order_comparator_module 是一个混合模块，它在它的端口列表中有两个分析fifo。
- avm_in_order_class_comparator_module是一个混合模块，它在它的端口列表中有两个分析fifo，它用它们的comp方法比较两个类流。
- avm_in_order_built_in_comparator_module 是一个混合模块，它在它的端口列表中有两个分析fifo，它用由SystemVerilog 定义的== operator来比较内嵌类型流。

avm_subscriber

avm_subscriber组件是一个参数化的，被命名的组件，它提供一个分析输出端口。通常，它被用来传送事务的内容，或用作覆盖率。

```
virtual class avm_subscriber #( type T = int ) extends
    avm_named_component;
    typedef avm_subscriber #( T ) this_type;
    analysis_imp #( this_type , T ) analysis_export = new( this );
    function new( string name , avm_named_component p = null );
        super.new( name , p );
    endfunction
    pure virtual function void write( input T t );
endclass
class my_transaction_dump extends avm_subscriber #( transaction );
    function void write( input transaction t );
        avm_report_message("dump" , t.convert2string );
    endfunction
endclass
class my_coverage extends avm_subscriber #( transaction );
    local bit [7:0] data;
    covergroup byte_cov;
        top_four : coverpoint data[7:4];
        bottom_four : coverpoint data[3:0];
    endgroup
    function new( string name , avm_named_component parent = null );
        super.new( name , parent );
        byte_cov = new;
    endfunction
    function void write( input p2s_transaction t );
```

```
        data = t.data;
        byte_cov.sample;
    endfunction
endclass
monitor m_monitor;
my_transaction_dump m_dump;
my_coverage m_coverage;
…
m_monitor.ap.register( m_dump.analysis_export );
m_monitor.ap.register( m_coverage.analysis_export );
```

上例注册了一个覆盖率类和一个带有单一分析端口的事务“dumper”

TLM 库

AVM的基础是用来在验证组件中进行通讯的事务级通道和接口。我们已经在OSCI TLM-1.0标准上模拟了TLM接口和通道的SystemVerilog执行。我们之所以选择OSCI TLM标准，是因为它考虑周全了事务级接口和通道的语义。

TLM 接口

单向接口

TLM接口的核心是put，get和peek。每一个都有阻塞，非阻塞和组合接口。也有一些接口将get和peek又组合为阻塞，非阻塞和组合形式。

单向接口假定对象可以商讨执行事务。阻塞接口就不允许对象商讨事务是否结束，但是它允许对象等待，除非它已经准备了。所以，用任务实现阻塞接口。非阻塞方法不允许商谈期限：它们是即时的，正因为这个原因，它们被实现为函数。但是，它们确实允许对象商谈是否接受由发动器请求的事务。然后，它们返回一个位表明这个事务是否被成功事务。

如下显示了put接口。Get，peek和get_peek的接口和它的形式相同。

```
virtual class tlm_blocking_put_if #( type T = int );
pure virtual task put( input T t );
endclass
virtual class tlm_nonblocking_put_if #( type T = int );
pure virtual function bit try_put( input T t );
pure virtual function bit can_put();
```

```
endclass
virtual class tlm_put_if #( type T = int );
pure virtual task put( input T t );
pure virtual function bit try_put( input T t );
pure virtual function bit can_put();
endclass
```

就如从上面的代码中看到的，所有的TLM接口都只用纯虚方法与抽象类一起被执行。尽管我们确实提供了一些通道来执行这些接口，但是接口仍然是TLM的基础，而不是通道。

双向接口

所有TLM双向接口都面有两个参数：一个用作请求类型，一个作为响应类型。主机接口将请求上的put和响应上的get和peek组合起来，而从机接口则将响应上的put与请求上的get和peek组合起来。

阻塞主机和从机接口如下所示。这里也有非阻塞的和组合变量。

```
virtual class tlm_blocking_master_if #( type REQ = int , type RSP = int );
    pure virtual task put( input REQ req );
    pure virtual task get( output RSP rsp );
    pure virtual task peek( output RSP rsp );
endclass
virtual class tlm_blocking_slave_if #( type REQ = int , type RSP = int );
    pure virtual task put( input RSP rsp );
    pure virtual task get( output REQ req );
    pure virtual task peek( output REQ req );
endclass
```

当请求和响应被紧紧的绑定在一个non pipelined 中，形成一一对应的关系时，就可以将调用put和get组合到单一的传送调用中：

```
virtual class tlm_transport_if #( type REQ = int , type RSP = int );
    pure task transport( input REQ request , output RSP response );
endclass
```

TLM 通道

tlm_fifo

tlm_fifo是一个参数化的类，它执行和输出所有在本节中描述的tlm接口。事

务提供基本的fifo性能，它还有两个分析端口：put_ap和get_ap。只要成功的完成了一个put，就写入一个事务到put_ap，只要成功的完成了一个get或peek，就写入一个事务到get_ap。

Tlm_fifos可大可小或不被绑定。构造器中规定了它的大小，如果它是0，fifo就认为未被绑定。

```
class tlm_fifo #( type T = int , type CLONE = avm_built_in_clone #( T ) )
    extends avm_named_component;
    tlm_put_imp #( this_type , T ) put_export;
    tlm_blocking_put_imp #( this_type , T ) blocking_put_export;
    tlm_nonblocking_put_imp #( this_type , T ) nonblocking_put_export;
    tlm_get_imp #( this_type , T ) get_export;
    tlm_blocking_get_imp #( this_type , T ) blocking_get_export;
    s tlm_nonblocking_get_imp #( this_type , T ) nonblocking_get_export;
    tlm_peek_imp #( this_type , T ) peek_export;
    tlm_blocking_peek_imp #( this_type , T ) blocking_peek_export;
    tlm_nonblocking_peek_imp #( this_type , T )
nonblocking_peek_export;
    tlm_get_peek_imp #( this_type , T ) get_peek_export;
    tlm_blocking_get_peek_imp #( this_type , T )
    blocking_get_peek_export;
    tlm_nonblocking_get_peek_imp #( this_type , T )
    nonblocking_get_peek_export;
    analysis_port #( T ) put_ap , get_ap;
    function new( string name = "" ,
        avm_named_component parent = null ,
            int size = 1 );
            ...
```

analysis_fifo

Analysis_fifo是一个没有界线的tlm_fifo的扩展，以及执行和输出所有的tlm接口，它执行分析接口。这表明它被经常用来订阅分析端口－可能是直接的，与层级的顶级分析fifo一起订阅，也可能是间接的，其分析fifo的分析输出口被从层次的低级层输出。

```
class analysis_fifo #( type T = int ) extends tlm_fifo #( T );
    analysis_imp #( analysis_fifo #( T ) , T ) analysis_export;
    function new( string name , avm_named_component parent = null );
        super.new( name , parent , 0 );
        analysis_export = new( this );
```

```
    endfunction
    function void write( input T t );
        this.try_put( t );
    endfunction
endclass
```

tlm_req_rsp_channel

tlm_req_rsp_channel是一个参数化类，它由请求和响应fifo组成。所有优先的fifo的已经输出的接口都由tlm_req_rsp_channel输出，尽管只输入了两个put_aps。这两个fifo的大小可以在构造器中任意规定。

除了这两个优先的fifo的原始put，get和peek接口，这个通道也输出双向主机和从机接口。

tlm_transport_channel

Tlm_transport_channel是一个tlm_req_rsp_channel，这里两个fifo的大小都为1。除了tlm_req_rsp_channel所有已经输出的接口，它也输出传输接口，它假定请求和响应之间有一个紧密的，一对一的，non pipelined 的连接。

```
class tlm_transport_channel #( type REQ = int , type RSP = int )
    extends tlm_req_rsp_channel #( REQ , RSP );
    tlm_transport_imp #( this_type , REQ , RSP ) transport_export;
    function new( string name = "" , avm_named_component parent = null );
        super.new( name , parent , 1 , 1 );
        transport_export = new( this );
    endfunction
    task transport( input REQ request , output RSP response );
        this.m_request_fifo.put( request );
        this.m_response_fifo.get( response );
    endtask
endclass
```

当将直接的双向激励转换成独立的，能被双向总线理解的请求和响应时，这个通道就非常有用。

附加的 AVM 组件

avm_algorithmic_comparator

Avm_algorithmic_comparator比较两个不同类型的数据流。这要求在这两个

数据流之外有一个第三个参数，它规定两个事务类型。第三个参数规定了一个变换器，它将一个数据流转换为另一个。变换器类将“BEFORE”对象变换为“AFTER”对象，然后将它们输入一个正常的顺促比较器。如果需要的话，TRANSFOMER中的变换方法会调用一个与DPI输入的C函数。

```
class avm_algorithmic_comparator #( type BEFORE = int ,
    type AFTER = int ,
    type TRANSFORMER = int_transform )
    extends avm_named_component;
    analysis_if #( AFTER ) after_export;
    analysis_imp #( this_type , BEFORE ) before_export;
    local avm_in_order_class_comparator #( AFTER ) comp;
    local TRANSFORMER m_transformer;
    function new( TRANSFORMER transformer ,
        string name ,
        avm_named_component parent = null );
        super.new( name , parent );
        m_transformer = transformer;
        comp = new("comp" , this );
        before_export = new( this );
    endfunction
    function void export_connections;
        after_export = comp.after_export;
    endfunction
    function void write( input BEFORE b );
        comp.before_export.write( m_transformer.transform( b ) );
    endfunction
endclass
```

TRANSFORMER必须有一个函数调用签名为*AFTER transform(输入BEFORE b)*变换；我们创建一个变换器的实例（而不是使它成为一个带有静态变换方法的真正的方针类），因为我们可能需要重置并在变换器自身上面配置。

为了使用代数比较器，我们必须：

- 创建一个变换器例化
- 将变换器实例传送到比较器构造器
- 注册带有一个analysis_port #的before_export(BEFORE)
- 注册带有一个analysis_port #的after_export(AFTER)

当第一个分析端口发出一个类型BEFORE为的事务，它就被变换器变换为AFTER型的，并被放入顺序比较器中的一个fifo中。一个AFTER型事务立即进入另一个fifo中。只要在这两个fifo中有一对事务，它们就从fifo中被取出并用AFTER.comp进行比较。

avm_global_analysis_ports

一个通用分析端口是一个多对多的广播。有给定名称和类型的通用分析端口在任何仿真中都可以用（只有一个，这样的分析端口，它是全局变量）。
它可以用来获取深埋在测试平台层级中的测试组件信息，而无需通过层级将分析端口连接起来。
但是，正因为它们是全局的，就要特别注意。对于任何一个全局分析端口，“writers”数可能为0，1或多，我们也没有任何控制来知道谁在听。

在global_analysis_ports #(type T = int)中只有一个方法。就是静态方法。

```
static function analysis_port #( T ) get_analysis_port( string name );
```

这可以获取这个名称和类型的分析端口，如果它已经存在的话。如果它还不存在，它就创建它。这个方法总是被用来访问全局分析端口。
如：

```
typedef global_analysis_ports #( transaction ) gap_t;
analysis_port #( transaction ) global_ap;
global_ap = gap_t::get_analysis_port("fred");
```

使用模板问题

本节讨论各种AVM库使用过程中要注意的事项。

使用完好的过程控制

在SystemVerilog中有完好的过程控制性能。你可以利用这些性能来找出当前过程的过程id并将它储存在某处。储存后，另一个过程会挂起，复用或杀掉那个过程。

可用这个性能来控制激励和事务器，尽管各种对象的详细内容有所不同。

激励控制

你经常想要在一个测试中执行许多不同的激励发生器。这些测试的顺序通常受到一定的约束。你可能希望串行运行一些测试，另外的测试则并行。当一些测试获得了它们的覆盖率结果时，你可能想要销毁这些测试，而让其它的测试继续，除非它们完成了执行。

你可以使用完好的过程控制来达到这一点。

例如：

```
process p_array[2:0];

fork
  begin p_array[0] = process::self; stimulus0.generate_stimulus( );
end
  begin p_array[1] = process::self; stimulus1.generate_stimulus( );
end
  begin p_array[2] = process::self; stimulus2.generate_stimulus( );
end
  do_stimulus_control_a( p_array );

join

stimulus3.generate_stimulus( );
```

```
fork
    begin p_array[0] = process::self; stimulus4.generate_stimulus( );
  end
    begin p_array[1] = process::self; stimulus5.generate_stimulus( );
  end
    begin p_array[2] = process::self; stimulus6.generate_stimulus( );
  end
    do_stimulus_control_b( p_array );

  join
```

上述代码：

- 并行的执行了stimulus 0，stimulus 1和stimulus 2，直到它们要么自己停止，要么由do_stimulus_control_a停止下来。
- 当这三个都完成，它执行stimulus 3直到结束。
- 并行的执行了stimulus 4，stimulus5和stimulus6，直到它们要么自己停止，要么由do_stimulus_control_b停止下来。

事务器中的线程控制

事务器中的线程控制略有差别，因为事务器是avm_verification_components，它的运行任务始于激励的开端。就如在avm_verification_component一节中所讲的，这里有缺省函数来挂起，复用和杀掉这个运行任务。但是当运行任务自己调用了自己的任务时，我们需要多做一点工作。

例如：

```
class pipelined_monitor extends avm_verification_component;
    analysis_port #( request_t ) request_ap;
    analysis_port #( response_t ) response_ap ;
    analysis_port #( transaction_t ) transaction_ap ;
// constructor omitted
    process m_request_process , m_response_process ;
    task run ;
        fork
            handle_request ;
```

```
            handle_response;
        join
    endtask
    task handle_request ;
        request_t req ;
        m_request_process = process ::self;
        forever begin
        ...
        req = new;
        request_ap.write( req );
        end
        endtask
    task handle_response ;
        response_t rsp ;
        transaction_t transaction;
        m_response_process = process ::self;
        forever begin
            ...
            rsp = new;
            transaction = new( req , rsp );
            request_ap.write( rsp );
        transaction_ap.write( transaction );
    end
    endtask
    virtual function void suspend;
        super.suspend;
        m_response_process.suspend;
        m_request_process.suspend;
    endfunction
    virtual task resume;
        super.resume;
        m_response_process.resume;
        m_request_process.resume;
    endtask
    virtual function void kill;
        super.kill;
        m_response_process.kill;
        m_request_process.kill;
    endfunction
endclass
```

上面的pipeline实例并行运行了两个过程：

handle_response 和 handle_request。这两个额外的过程在运行任务的同时必须被挂起，复用和杀掉。

事务，便捷方法和定向测试

在做一些诸如受约束的随机激励发生，功能覆盖率和事务器设计时，使用事务有许多的优点。将事务用一个在组件间移动的类（而不是作为带有不同判据的许多方法的调用）表示使得所有这些技术的机制容易理解得多。

然而，交付写入的定向激励的自然方式是用一系列的方法调用，这看起来就更像软件。为了弥补这个缺陷，我们使用了一套称作便捷层的方法。它由一个或多个方法组成，这些方法看起来项软件调用，但是它们创建事务并将事务通过tlm端口发出，也有可能接收事务并将它们发回给用户层。

只要定向激励发生器的外部连通性运行，在定向和约束随机激励发生器如 avm_stimulus (参见 avm_stimulus 节)之间都没有差别。只要激励写入器在运行，它们所看到的就都是便捷层，并且完全不知道tlm端口的存在，tlm端口是将激励发生器和其余验证环境连接起来的必要端口。

```
class mem_bidirectional_stimulus extends avm_named_component;
    tlm_transport_if #( request_t , response_t ) initiator_port;
    function new( string name , avm_named_component parent = null );
        super.new( name , parent );
    endfunction

    task generate_stimulus( int write_count = 16 , int read_count = 16 );
        address_t address;
        data_t data;
        for( address = 0; address < write_count; address++ ) begin
            write( address , address + 17 );
        end
        for( address = 0; address < read_count; address++ ) begin
            read( address , data );
        end
    endtask
    task write( input address_t address , input data_t data );
        request_t request = new( address , MEM_WRITE , data );
        response_t response;
        initiator_port.transport( request , response );
```

```
    endtask
    task read( input address_t address , output data_t data );
        request_t request = new( address , MEM_READ );
        response_t response;
        initiator_port.transport( request , response );
        data = response.m_rd_data;
    endtask
endclass
```

上面的实例在运行任务里有两个循环，这看起来相当于软件。是便捷层（它可能被放在基类中）事务通过tlm_transport_if port来传送请求和响应。而且，如果我们希望激励发生器在自然停止前挂起，复用或杀掉它，我们可以使用“激励控制”那一节所采用的技术。

可复制的随机激励

avm_stimulus类的构造器写出get_randstate()的结果。这个不可读的字符串包含在激励发生器在产生激励以前复制它的的准确状态所需要的所有信息。这意味着我们可以用set_randstate方法来产生同样的一列约束随机激励，无论我们是否已经设定了一个不同的通用值，也无论我们是否已经将这个激励发生器移到测试平台的不同地方（或完全移到了一个不同的测试中）。

例如：

```
avm_stimulus #( transaction ) m_stimulus;
...
m_stimulus = new(“stimulus”);
…
read_randstate_from_file( previous_randstate );
m_stimulus.set_randstate( previous_randstate );
…
m_stimulus.generate_stimulus( … );
```

编码技术

在执行AVM的过程中有两种编码技术需要进行解释。第一种就是利用包来减少层次，第二种就是在标准组件中，如avm_in_order_ comparator 和tlm_fifo.[1]

利用策略类。

包和多级继承

分开描述的抽象接口，如TLM接口（见“TLM接口”一节）或分析接口（见analysis_if 和 analysis_port）提供了事务级传送启动器和目标之间的清楚的分别。这个接口规定了启动器和目标之间的协议（用基于对象的术语，就是客户和服务器），但是由目标来确定如何实现这个接口。将启动器和目标通过接口分开的观点是TLM和AVM的核心。

大部分的通道执行多个接口。例如，tlm_fifo以blocking，non，blocking和组合形式执行put，get，peek和get_peek，加起来有12各接口。在程序语言如C++ 和 Java中，我们可以用多级继承来实现这个模式：以及从一个基本组件如avm_named_component中继承，它也可以继承这12个抽象接口。

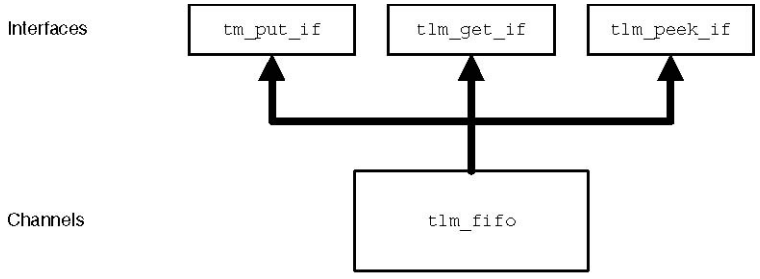


Figure A-2. Multiple Inheritance

但是，SystemVerilog没有这样的内嵌到语言中的工具，因此我们需要采用另外的方式。下面就是包模式采用的方式：

1. 在缺乏C++ 或 Java风格的多级继承和操作符重载时，这些可以看成是工作区。

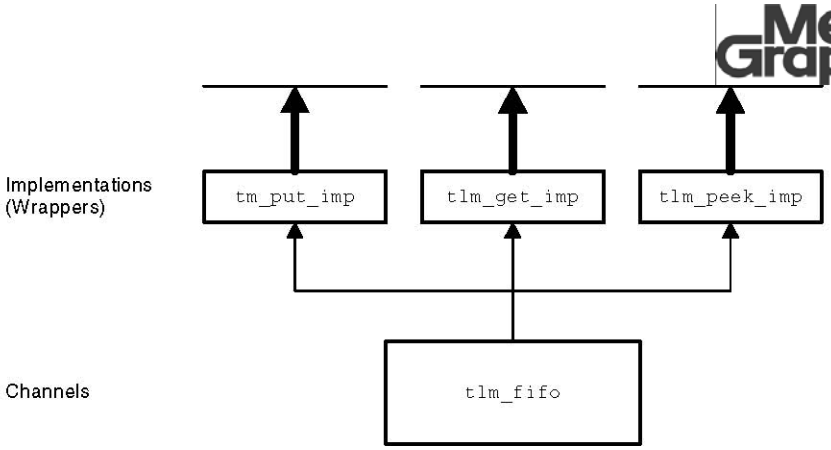


Figure A-3. Multiple Inheritance in SystemVerilog using Wrappers

如上所示的包模式中，通道有许多实现或包。这些包从它们的接口继承并代表调用这个接口到通道的每一个方法。

blocking put包的结构和tlm_fifo代码的一部分如下所示：

```
class tlm_blocking_put_imp #( IMP = tlm_fifo, T = int )
    extends tlm_blocking_put_if #( T );
    local IMP m_imp;
    function new( IMP I );
        m_imp = i;
    endfunction
    task put( input T t );
        m_imp.put( t );
    endtask
endclass

class tlm_fifo #( T = int )
```

现在，我们只需要知道生产者和用户有blocking put 和 blocking get 端口，就可以写它们了－实际上，实现blocking put 和 get函数是在tlm_fifo中进行的，这与生产者和用户无关。基本TLM模式的SystemVerilog执行，在第91页使用fifo就是一个这样的生产者和用户的实例。

Port / Export 编码惯例

从上面的代码中可以注意到，生产者和用户都有port，而tlm_fifo有exports。Port和export都只是对接口的句柄，但这些句柄是用不同的方式使用的。

对于一个export，接口的实现包括within这个组件以及用输出口provided to外部环境。在构造器被调用后，这些句柄有非0值。

对于port，接口的实现是external到这个组件。这个组件require一个非0值，并在构造一些时间后将该值提供给这个句柄。

如果讨论的组件在环境类的顶部，则提供接口的输出口应当分配给需要接口与函数相连的端口，就如上例所示。关于各层组件联通性的详情请参见AVM Library Documentation中的相关章节。

策略类

在avm_transaction一节中，我们介绍了一个类称为有效AVM事务事件的必要条件，以及实现纯虚convert2class方法，它也必须提供T clone ()和comp(input T t)方法，这是因为这些方法被假定是存在于参数化的验证组件中，如avm_in_order_comparator。

但是，在SystemVerilog中，有两个独立类型的系统。有些类被给出了用户定义的方法，有些则内嵌了静态类型，如int，bit，enums和structs，它们主要被操作，或者专门内嵌，而且没有重载操作符，如“==”。

我们希望我们的标准组件，如tlm_fifo 或 avm_in_order_ comparat or可以与基于类的事务事件和内嵌型进行工作。为此，我们引入了策略类概念。一个策略类是一个类，它只是告诉我们如何执行某个操作。被定义和用于avm中的指针有*comp*, *convert* 和 *clone*。这三个方法被定义用于类和内嵌型。

下面是一个用来比较两个带内嵌型对象的策略类。

```
34 class avm_built_in_comp #( type T = int );
35
36 static function bit comp( input T a , input T b );
37 return a == b;
38 endfunction
39
40 endclass
```

下面是一个用来比较两个带类对象的策略类。

```
78 class avm_class_comp #( type T = int );
79
80 static function bit comp( input T a , input T b );
```

```
81 return a.comp( b );
82 endfunction
83

   84 endclass
```

内嵌策略依赖于用于内嵌型的==操作符的存在，类操作符依赖于类T中comp()方法的存在。

如何使用策略类的实例在in_order_comparator组件中。它被设计成要么内嵌型工作，要么与类类型工作。

```
class avm_in_order_comparator #( type T = int ,
type comp = avm_built_in_comp #( T ) ,
 type convert = avm_built_in_converter #( T ) )
```

在内部，我们用策略类来进行比较和信息构造：

```
 forever begin

 before_port.get( b );
 after_port.get( a );

 if( !comp::comp( b , a ) ) begin

$sformat( s , “%s differs from %s” ,
                        converter::convert2string( b ) ,
                        converter::convert2string( b ) );

   m_report_object.avm_report_warning("Comparator Mismatch" , s );

  m_mismatches++;
```

```
    end
    else begin

            m_report_object.avm_report_message("Comparator Match" ,

     converter::convert2sting( b ) );
    m_matches++;
    end
    end
    endtask
```

现在，我们可以简单地给方便的类规定不同地策略：

```
  class avm_in_order_built_in_comparator #( type T = int )
      extends avm_in_order_comparator #( T );
    …
  endclass

class avm_in_order_class_comparator #( type T = int )
     extends avm_in_order_comparator #( T , avm_class_comp #( T ) ,
  avm_class_converter #( T ) );
    …
  endclass
```

而执行比较和打印的代码保持不变。

附录 B

参考书目

标准：

1　IEEE standard 1800-2005, “IEEE Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language”, November 2005.

2　IEEE, standard 1666-2005, “IEEE Standard SystemC Language Reference Manual”, March 2006.

3　OSCI TLM-1.0 Transaction Level Modeling Standard. SystemC kit with whitepaper available on *http://www.systemc.org*.

功能验证

1　Janick Bergeron, “Writing Testbenches: Functional Verification of HDL Models”, Second edition, Kluwer Academic Publishers, 2003.

2　Andreas S. Meyer, “Principles of Functional Verification”, Elsevier Science, 2004.

3　Harry D. Foster, Adam C. Krolnik, David J. Lacey, “Assertion-Based Design”, 2nd Edition, Kluwer Academic Publishers, 2004.

SystemC

1　Thorsten Grotker, Stan Liao, Grant Martin, Stuart Swan, “System Design with SystemC”, Kluwer Academic Publishers, 2002.

2　David C. Black and Jack Donovan, “SystemC: From the Ground Up”, Kluwer Academic Publishers, 2004.

3　J. Bhasker, *A SystemC Primer*, Star Galaxy Publishing, 2002.

4 Frank Ghenassia (ed.), “Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems”, Springer, 2005.

C++及面向对象编程

1 Bjarne Stroustrup, “The C++ Programming Language”, Third Edition, Addison-Wesley, 1997.

2 Gregory Satir, Doug Brown, “C++: The Core Language”, O’Reilly & Associates, Inc., 1995.

3 Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995.

4. Andrei Alexandrescu, “Modern C++ Design: Generic Programming and Design Patterns Applied”, Addison-Wesley, 2001.

编程类型

1 Steve McConnell, “Code Complete”, Second Edition, Microsoft Press, 2004.

2 Herb Sutter, Andrei Alexandrescu, “C++ Coding Standards: 101 Rules, Guidelines, and Best Practices”, Addison-Wesley, 2005.