

The UVM Primer

ajzcr@qq.com

v0.0.2

2016.8.17

目录

目录	1
第一章 引言	6
短而美	6
你说的是些什么"概念"呢?	6
网上资源	6
示例设计	7
代码字体说明	8
第二章 用于 TinyALU 的一个常规验证平台	10
TinyALU 的功能覆盖率模型	10
验证平台文件	10
验证平台变量定义	10
covergroup 模块	11
tester 模块	13
用 Scoreboard 循环来做自检	14
我们有一个验证平台了	14
第三章 SystemVerilog interfaces 和 BFM	15
TinyALU 的 BFM	15
创建一个模块化验证平台	17
总结	18
第四章 面向对象编程	19
为什么人人都爱 OOP	19
代码重用	19
代码维护	19
内存管理	20
总结	20
第五章 类和继承	21
从结构体开始	21
定义一个类	22
继承类	24
总结	25
第六章 多态	26
虚函数	28
抽象类和纯虚函数	29
总结	30
第七章 静态变量和方法	31
声明静态变量	31
定义静态方法	32
总结	33
第八章 类的参数化定义	35
造一个动物笼子	35

用参数来定义变量.....	37
总结.....	38
第九章 Factory 模式.....	40
为什么用 Factory 模式?.....	40
创建动物 Factory.....	41
使用动物 Factory.....	42
总结.....	43
第十章 面向对象的验证平台.....	44
TinyALU 验证平台对象.....	44
基于对象的顶层 module.....	44
引入类定义.....	45
例化 DUT 和 BFM, 定义验证平台类变量.....	45
例化并启动 testbench 对象.....	45
testbench 类.....	46
声明.....	46
execute()方法.....	47
tester 类.....	47
scoreboard 类.....	48
coverage 类.....	49
模块集合.....	49
总结.....	49
第十一章 UVM Test.....	51
使用 Factory 模式创建测试用例.....	51
用 UVM 启动仿真.....	51
定义并注册 UVM Test.....	53
run_test()方法.....	53
理解 objection.....	54
编写 add_test 类.....	55
总结.....	55
第十二章 UVM Component.....	57
第一步: 从 uvm_component 类继承创建模块.....	57
第二步: 用 uvm_component_utils()宏注册类到 Factory.....	57
第三步: 提供最简 uvm_component 构造器.....	57
第四步: 有必要的话重写 UVM phase.....	58
重载比较器类中的方法.....	58
用 build_phase()方法来建立验证平台.....	59
总结.....	60
第十三章 UVM Env.....	62
复杂性编程 VS 适应性编程.....	62
构建适应性代码.....	63
结构与激励分离.....	65
env 类.....	67
用 UVM Factory 来创建 UVM 组件.....	67
Factory 重载.....	68

总结	69
第十四章 一个新思路	70
对象和脚本	70
两种对象通信的类型	71
第十五章 与多个对象通信	72
观察者设计模式	75
UVM 和观察者设计模式	75
uvm_analysis_port	75
uvm_subscriber	75
实现我们的订阅者	76
在 dice_roller 里用 uvm_analysis_port	76
connect_phase()方法	77
得 A	78
总结	78
第十六章 在验证平台里使用 analysis port	80
重复代码问题	80
验证平台图解	80
BFM 中的对象句柄	81
监控 TinyALU 的指令	82
用 command_monitor_h 操作指令	83
TinyALU 的 coverage 类用作订阅者	83
订阅多个 analysis port	85
订阅到 monitor	86
总结	87
第十七章 线程间通信	88
等一下, 没有对象端口吗?	89
producer 作为对象	89
consumer 作为对象	90
连接端口	91
连接端口到 TLM FIFO	91
非阻塞通信	92
图解 put 端口和 get 端口	93
总结	93
第十八章 put 和 get 端口的实践	94
base_tester 类	95
TinyALU 的 driver	96
总结	97
第十九章 UVM 报告	98
UVM 报告宏	98
UVM verbosity 等级	99
设置 verbosity 门限(级别)	100
设置全局 verbosity 门限	100
在 UVM 层级中设置 verbosity	100
禁用 warning, error 和 fatal 信息	102

总结	103
第二十章 类层级和深度操作	104
还是回到狮子的例子	104
深度拷贝	106
总结	109
第二十一章 UVM Transaction	110
用对象存放数据的益处	110
transaction 启动	110
定义 transaction	111
创建随机化数据域	111
uvm_object 构造器	112
do_copy()方法	112
clone_me()方法和 MOOCOW	113
do_compare()方法	113
convert2string()方法	114
使用 transaction	114
第1步: 创建 result_transaction 类操作结果	115
第2步: 从 command_transaction 继承创建 add_transaction 类来生成加法指令	115
第3步: 将 base_tester 命名为 tester, 因为现在可以用来做任意的 Test 了	116
第4步: 修改 command_monitor 来创建 command_transaction	116
第5步: 修改 result_monitor 来创建 result_transaction	116
第6步: 修改 scoreboard 来使用 result_transaction 的 compare()方法	117
第7步: 修改 add_test 来使用 add_transaction	117
总结	118
第二十二章 UVM Agent	119
编写 TinyALU agent	121
使用 UVM Agent	123
顶层 Module	123
dual_test 类	124
TinyALU 环境	126
总结	126
第二十三章 UVM Sequence	128
第一步: 创建携带数据的 TinyALU sequence 项目	129
第二步: 把 tester 替换成 uvm_sequencer	129
第三步: 对 driver 进行支持 sequence 的升级	130
第四步: 在环境里例化 driver 和 sequence 然后相互连接	130
第五步: 编写 UVM sequence	132
第六步: 编写用 sequencer 启动 sequencer 的 Test	133
斐波那契 Test	134
Virtual Sequence	135
不用 Sequencer 启动 Virtual Sequence	136
多线程里的 Virtual Sequence	137
总结	138
第二十四章 UVM 伴我同行	140

说明

原文中的以下内容不做翻译:

1. 使用等宽字体排版的, 直接与代码对应的词汇, 本文中使用 consolas 字体排版直接对应不翻译了.
2. SystemVerilog 中的关键词术语 interface, module 等.
3. UVM 中的专用术语 agent, driver, transaction 等.
4. 其他的专用术语 TLM(Transaction Level Model), OOP(Object Oriented Programming)和 BFM(Bus Functional Model)等.

第一章

引言

Primer--An elementary textbook that serves as an introduction to a subject of study. (New Oxford American Dictionary)

作为验证咨询师和通用验证方法学(UVM)应用专家的我经常接到的提问是, "我应该读什么书来学习 UVM 呢?"

这个问题搞的我很为难, 因为虽然网上有很多 UVM 的参考资料了, 但是市面上甚至还没有教读者 UVM 的基础书籍.

直到现在.

《The UVM Primer》是你决定开始学习 UVM 的时候可以读的书. 这本书假定你已经有了 SystemVerilog 的基础知识和验证的基本概念, 然后引领你一步一步的理解和编写基于 UVM 的验证平台.

短而美

UVM Primer 是 UVM 的入门. 我编写的这本书可以让一个普通的验证工程师经过短暂的阅览之后就能准备去面试了. 这本书的特点是篇幅短, 样例多, 容易阅读.

我避免了入门书通常会有的无聊的玄学问题("UVM的历史是怎样的?" "谁写了 UVM?" "我能用 UVM 来做个美味的煎饼吗?").

UVM Primer 提供了你所需要理解和编写 UVM 验证平台一切, 不过并没有深陷 UVM 的所有细枝末节. UVM 是庞大的, 一本无所不包的书是很大的, 取而代之的是, UVM Primer 为你提供了探究 UVM 特性所需要的概念.

你说的是些什么"概念"呢?

UVM 构建于以下的简单概念:

- SystemVerilog 的 OOP
- 让你可以不用重编译来指定 test 和验证平台结构的动态生成对象
- 由 Agent, Driver, Monitor 和 BFM 组成的层级化验证平台
- 对象间的 transaction 级别通信
- 验证平台激励(UVM sequence)和验证平台结构的分离

到本书结尾的时候, 你就会对这些概念和它们对验证平台的作用有坚实的掌握.

网上资源

The UVM Primer 通过代码来探讨 UVM. 这对读者和作者在代码细节方面提出了挑战. 一方面, 一行行的描述代码是枯燥的. 另一方面, 代码片段之上的讨论, 会让读者纠结于诸如代码的实现和代码的作用这些问题上. 这些细节也不应该留作读者的作业.

在提供本书的代码行为的探讨之外, 我还在 www.uvmprimer.com 上提供了视频. 每一章的代码示例都有配套的视频.

入门读物的读者通常会有很多相关的问题, 读者们可以在 UVM Primer 的 Facebook 主页(这是读者进行书中的概念和代码讨论的中心)上提问. 你可以在 www.uvmprimer.com 下载 gzip tar 文件格式的示例代码或者在 www.git-hub.com 中 pull 下代码的 GIT 仓库.

示例设计

我们会通过验证一个简单的设计---TinyALU, 来展开讨论 UVM. 这样一来, 你无需受到 DUT 复杂度的干扰, 可以投入精力到验证平台上.

TinyALU 是 VHDL 编写的一个简单的 ALU. 它接收两个 8 位数据(A 和 B), 并产生 16 位的结果. 这里是 TinyALU 的顶层:

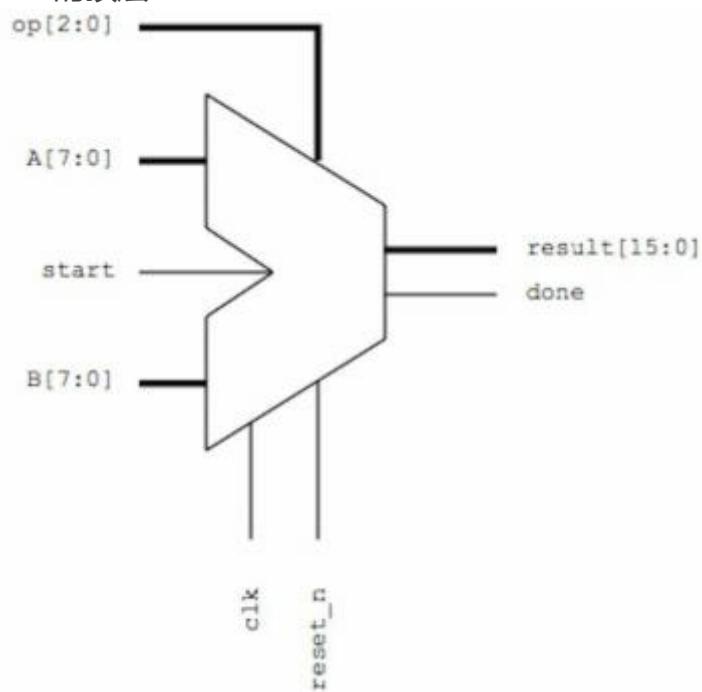


图 1. TinyALU 模块图

这个 ALU 在时钟的上升沿工作. 当 start 信号有效时, TinyALU 从 A, B 总线上读取操作数, 从 op 总线上读取指令, 然后根据指令生成结果. 指令可以是任意长度时钟周期的, TinyALU 在指令完成的时候拉高 done 信号.

reset_n 信号是低有效, 同步的复位信号.

TinyALU 有 5 个指令: NOP, ADD, AND, XOR 和 MULT. 用户在需要计算的时候对 3 位总线 op 进行编码, 编码表如下:

Operation	Opcode
no_op	3'b000
add_op	3'b001
and_op	3'b010
xor_op	3'b011
mul_op	3'b100
unused	3'b101-3'b111

图 2. TinyALU 指令和操作数

这里是 TinyALU 的波形图:

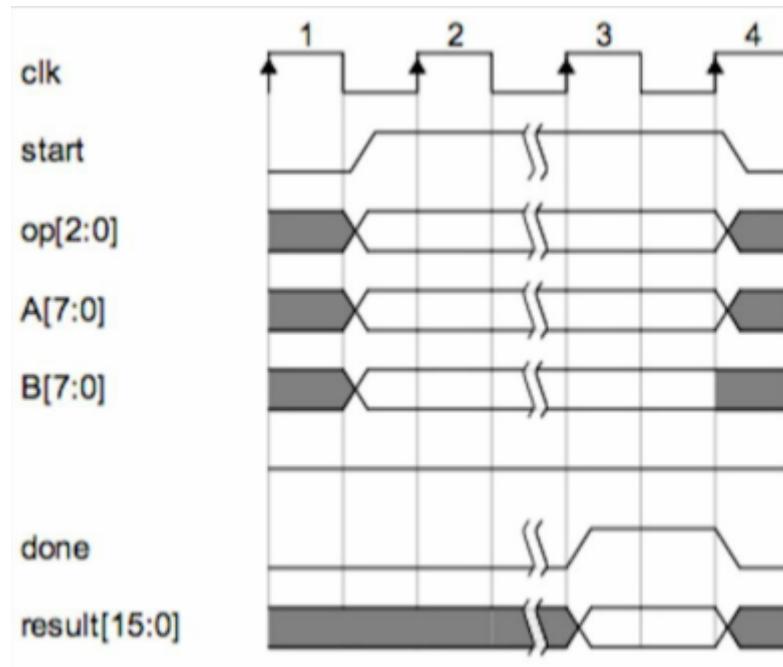


图 3. TinyALU 协议波形

start 信号需保持高, 操作码和操作数在 TinyALU 拉高 done 信号之前都要保持稳定. done 信号只拉高一个时钟长度. NOP 指令没有 done 信号. 在 NOP 中, 应答器在 start 信号为高一个周期后将其拉低.

我们将从创建传统的 TinyALU 验证平台开始我们的 UVM 之旅. 然后每一章我们都会改动验证平台, 让它变得越来越 UVM, 然后在转变验证平台的时候讨论使用 UVM 的好处.

代码字体说明

说到代码就得提到变量名的排版了, 由于这些会在文句中引起歧义(比如"waiting for start to go high"), 我会用 **this code font** 等宽字体来标识变量还有 SystemVerilog 关键字(比如

"waiting for **start** to go high").

第二章

用于 TinyALU 的一个常规验证平台

从我们熟悉的东西开始去学新的东西会更容易。所以在这里，我们从传统的 SystemVerilog 验证平台开始。完成这个传统验证平台之后，我们再一步步的改进它，直至变成一个完全的 UVM 验证平台。

TinyALU 对验证平台的需求如下：

- 完整的功能测试
- RTL 的所有代码行的运行，和之上形成的路径的仿真

这是一个“覆盖率导向”方法学。我们先定义需要覆盖的内容，然后创建验证平台去覆盖。我们将创建一个自检的验证平台，这样我们就可以进行回归运行而不用去手动检查结果了。

TinyALU 的功能覆盖率模型

TinyALU 的功能覆盖模型用 SystemVerilog 的 covergroup 来实现。本文不再深入讨论 covergroup 的细节，覆盖率目标如下：

- 测试所有指令
- 所有指令的全“0”输入仿真
- 所有指令的全“1”输入仿真
- 所有指令的复位后运行
- 单周期指令之后运行乘法指令
- 乘法指令之后运行一个单周期指令
- 指令连续两次运行的仿真

所有的上述情景都测试并通过之后，我们就能确认这个 TinyALU 是可以用的。是否达到 100% 代码覆盖率也是需要检查的。这章之外我们将不会再讨论覆盖率的细节，所以需要的话读者可以自行跳过 covergroup 的定义。

验证平台文件

我们把验证平台放在同一个仿真文件里。该文件包含三个部分：激励，自检，覆盖率。我们在文件里例化 DUT，为其施加激励并使用自检和覆盖采集的 always 模块来监控它。

验证平台变量定义

首先，我们用一种便于定义激励的形式来定义 TinyALU 的指令，我们的验证平台信号定义如下：

```

1 module top;
2
3     typedef enum bit[2:0] {no_op    = 3'b000,
4                             add_op   = 3'b001,
5                             and_op   = 3'b010,
6                             xor_op   = 3'b011,
7                             mul_op   = 3'b100,
8                             rst_op   = 3'b111} operation_t;
9     byte          unsigned      A;
10    byte          unsigned     B;
11    bit           clk;
12    bit           reset_n;
13    wire [2:0]    op;
14    bit           start;
15    wire          done;
16    wire [15:0]   result;
17    operation_t  op_set;
18
19    assign op = op_set;
20
21    tinyalu DUT (.A, .B, .clk, .op, .reset_n, .start, .done, .result);
22
23

```

图 4. 验证平台声明和例化

我们使用 SystemVerilog 的枚举类型将 TinyALU 的指令定义为枚举变量。为了方便验证平台的使用，我们将 DUT 中没有的 `rst_op` 指令加入指令枚举类型。`assign` 语句用于将指令输入到 DUT 的指令总线上。

我们使用 SystemVerilog 的 `byte` 和 `bit` 来定义激励变量。最后，我们使用端口顺序映射的形式(这样就不用把信号名写两次，这是个很好的特性)来例化 DUT。

covergroup 模块

我们之前已经从测试的角度定义了功能覆盖。所有的激励都运行完之后，TinyALU 就达到了完全测试。

我们要用 `covergroup` 来捕获功能覆盖。先定义 `covergroup`，再例化并用其进行采集。先来看看定义：

```

25    covergroup op_cov;
26
27        coverpoint op_set {
28            bins single_cycle[] = {[add_op : xor_op], rst_op,no_op};
29            bins multi_cycle = {mul_op};
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45    covergroup zeros_or_ones_on_ops;
46
47        all_ops : coverpoint op_set {
48            ignore_bins null_ops = {rst_op, no_op};}
49
50        a_leg: coverpoint A {
51            bins zeros = {'h00};
52            bins others= {'h01:'hFE}};
53            bins ones  = {'hFF};
54        }

```

图 5. TinyALU covergroup 定义

这些定义展示了覆盖模型中的一些“覆盖仓”. op_cov 这个 covergroup 用来保证我们覆盖了所有的指令以及这些指令间可能的组合. zeros_or_ones_on_ops 这个 covergroup 用来检查我们是否在所有的指令上用全 0 和全 1 操作数进行测试.

定义完这些 covergroup 之后, 我们需要将其声明, 例化并采集:

```

111      initial begin : coverage
112
113          oc = new();
114          c_00_FF = new();
115
116          forever begin @(negedge clk);
117              oc.sample();
118              c_00_FF.sample();
119          end
120      end : coverage
121

```

图 6. 在 coverage 块里声明, 例化并收集 covergroup

这是个非常简单的覆盖模型. 我们在每个时钟的下降沿检查 TinyALU 的指令和数据输入并进行记录.

先跳到最后看看结果, 这个验证平台达到了 100% 功能覆盖率:

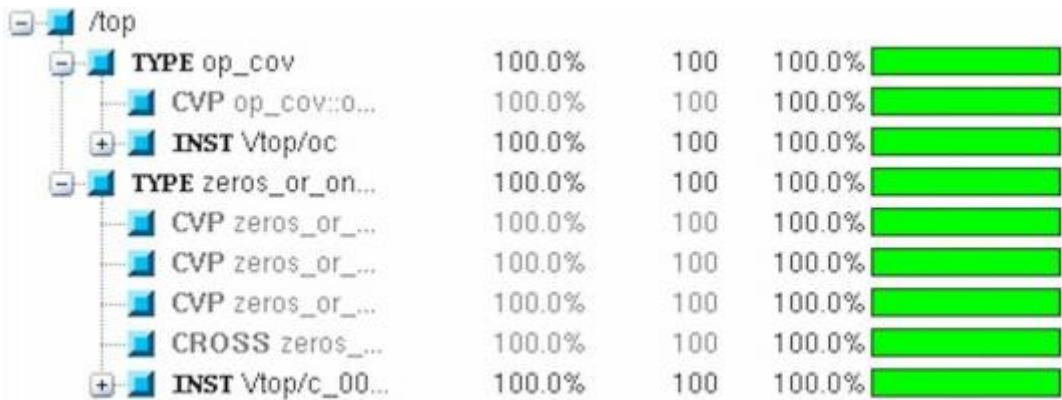


图 7. 完成 functional coverage

完全覆盖是借助带约束随机激励达成的.

tester 模块

在理想情况下, 我们可以为每个覆盖目标写一个直接测试用例. 对 TinyALU 来讲, 这个是可行的, 就是需要写更多的代码. 不过我们还是用随机的激励来吧. 由于纯随机大多是无效和错误的, 我们要将激励约束到正确和有效的值.

我们用 `get_op()` 和 `get_data()` 这两个函数来实现带约束随机激励:

```

168      initial begin : tester
169          reset_n = 1'b0;
170          @(negedge clk);
171          @(negedge clk);
172          reset_n = 1'b1;
173          start = 1'b0;
174          repeat (1000) begin
175              @(negedge clk);
176              op_set = get_op();
177              A = get_data();
178              B = get_data();
179              start = 1'b1;
180              case (op_set) // handle the start signal
181                  no_op: begin
182                      @(posedge clk);
183                      start = 1'b0;
184              end

```

图 8. tester 块生成带约束随机激励

这个循环为 TinyALU 生成了 1000 个 transaction. 每次循环, 我们从 `get_op()` 里得到随机的指令, 从 `get_data()` 里得到随机操作数, 将他们驱动到 TinyALU, 并且执行了 `start` 信号的协议. 这两个函数保证了我们得到的是合法的指令和有效的数据. 这些过程我们会在后面的章节做细节展开讨论.

要注意上文中我们做了哪些事情: 我们选择了指令的数量, 选择了数据, 处理了信号级的协议. 后面的内容中我们还会使用 TinyALU 来产生斐波那契序列. 想象一下把现在的验证平台修改了去做这个测试用例有多困难. 我们怕是得先完全重写这本书了.

用 Scoreboard 循环来做自检

Scoreboard 循环会用预期结果来检查 TinyALU 的结果. 循环会监控 done 信号, 每当 done 信号拉高, scoreboard 根据 TinyALU 的输入预测其结果, 并检查其是否正确:

```
149      always @(posedge done) begin : scoreboard
150          shortint predicted_result;
151          case (op_set)
152              add_op: predicted_result = A + B;
153              and_op: predicted_result = A & B;
154              xor_op: predicted_result = A ^ B;
155              mul_op: predicted_result = A * B;
156          endcase // case (op_set)
157
158          if ((op_set != no_op) && (op_set != rst_op))
159              if (predicted_result != result)
160                  $error ("FAILED: A: %0h B: %0h op: %s result: %0h",
161                         A, B, op_set.name(), result);
162
163      end : scoreboard
```

图 9. scoreboard 块

我们有一个验证平台了

我们已经成功的完成了 TinyALU 的验证平台. 我们的验证平台可以传递功能覆盖率, 检查所有的指令是否工作正常, 并且只用很少的工作来产生激励. 这些工作用一个模块和一个文件就完成了.

然而这对验证平台来说是一个很不好的例子. 所有的行为都在一个文件里混到一起了. 这些混杂使得这个验证平台难以重用.

那我们怎么去改进这个验证平台呢? 我们最先注意到的是 scoreboard 模块, 覆盖率模块和 tester 模块是在完全不同的功能之上定义的. 这些不应该出现在同一个文件里. 把他们分开之后, 我们就可以把这个验证平台中的模块重用到其他验证平台, 然后方便的修改其他 Test 模块中的其他模块, 比如激励.

在下一章, 我们会把验证平台拆分成多个模块. 我们也将学习如何用 SystemVerilog 的 interface 来实现 BFM.

第三章

SystemVerilog interfaces 和 BFM

在前一章我们为 TinyALU 创建了一个还不错的验证平台. 它有着一个现代验证平台的所有特性元素:

- 功能覆盖率目标--验证平台会衡量测试过的内容而不是依赖于一系列的 Test.
- 自检--我们不需要再去检查波形或者查看文本输出. 想运行多少测试用例都可以.
- 带约束的随机激励--不需要为所有的指令写 test; 只用为满足覆盖率要求写随机激励.

我们的验证平台的不足之处在于缺少模块化. 所有的功能都塞在一个文件, 一个 SystemVerilog 模块里面. 这让它的修改, 重用和调试都变得困难了. 这是个设计缺陷.

验证团队有两个选择. 随着项目和设计的演化, 他们可以让验证平台变得越来越脆弱, 越来越统一发生 bug, 或者将验证平台设计得越来越灵活越来越强大.

我们在前面章节写的验证平台属于前者. 因为所有的功能都塞在了一个文件里, 这样验证平台不会优雅的随着设计而演化. 反之, 我们不得不不断的放新的东西进去, 不断的拷贝修改, 或者是最次的情况, 用#ifndef 宏来控制不同仿真之间需要编译的内容.

等到我们的项目到了关键时刻, 我们的验证平台会变得很脆弱, 没人敢动它了因为小小的改动就会让其崩溃.

这样, 即使我们没等项目做完, 就已经发现这个脆弱的 bug 丛生的验证平台是不可能用到下个设计的.

谁会需要它呢?

与之相对的, 工程师们需要的是使用一种标准化方法来创建模块化的, 会越来越强大的验证平台. 我们需要的是每次我们加入新代码, 其功能就更强大, 并且之后的开发者可以很容易的加入新功能的一个验证平台.

幸运的是, SystemVerilog 给了我们这样的工具. interface 是其中首要的工具. SystemVerilog 的 interface 封装了验证平台中的 port 信号, 让其在 modules 间, 以及我们后面会见到的 objects 间共享这些信号.

我们通向 UVM 的第一步是使用 SystemVerilog 的 interface 来模块化我们的验证平台. 我们会发现 interface 可以让我们很方便的共享信号. 我们可以用它们来创建 BFM, 这样就可以对简单的总线访问协议进行封装.

TinyALU 的 BFM

`tinyalu_bfm` 封装了所有 TinyALU 验证平台用到的信号, 并提供了一个时钟, 一个 `reset_alu()` task 还有一个向 DUT 发送指令的 `send_op()` task. 我们用定义 module 的方式来定义一个 SystemVerilog interface. 我们从使用 interface 关键字来定义 interface 中的信号开始:

```

1   interface tinyalu_bfm;
2     import tinyalu_pkg::*;
3
4     byte      unsigned      A;
5     byte      unsigned      B;
6     bit       clk;
7     bit       reset_n;
8     wire [2:0] op;
9     bit       start;
10    wire     done;
11    wire [15:0] result;

```

图 10. TinyALU BFM: 信号和时钟

BFM 为我们提供了模块化的第一步. 它可以处理所有的信号级的激励问题, 所以验证平台中的其他部分都可以忽略这个问题了. 比如, `tinyalu_bfm` 生成了时钟. 这个模块化为我们将来验证平台拆分成多个 module 提供了直接的便利.

BFM 提供了两个 task: `reset_alu()` 和 `send_op()`. 这里是 `reset_alu()`:

```

25   task reset_alu();
26     reset_n = 1'b0;
27     @(negedge clk);
28     @(negedge clk);
29     reset_n = 1'b1;
30     start = 1'b0;
31   endtask : reset_alu

```

图 11. 复位 task

这个 task 用来拉低 `reset` 信号, 等待几个时钟然后再次拉高.

`send_op()` task 用来向 ALU 发送指令并返回结果:

```

33   task send_op(input byte iA, input byte iB, input operation_t iop,
34             output shortint alu_result);
35
36     op_set = iop;
37
38     if (iop == rst_op) begin
39       @(posedge clk);

```

图 12. `send_op()` task 定义

`operation_t` 是一个 `tinyalu_pkg` 包中定义的枚举类型. 我们在 `interface` 的顶部引入 `tinyalu_pkg` 包.

`send_op()` task 展示了 BFM 是如何封装 DUT 相关的协议:

```

45      end else begin
46          @ (negedge clk);
47          A = iA;
48          B = iB;
49          start = 1'b1;
50          if (iop == no_op) begin
51              @ (posedge clk);
52              #1;
53              start = 1'b0;
54          end else begin
55              do
56                  @ (negedge clk);
57                  while (done == 0);
58                  start = 1'b0;

```

图 13. 对 TinyALU 信号级协议进行封装

在这个例子中, task 将指令和数据放在 op 总线和操作数总线上. 之后它拉高 start 信号并根据指令的要求对其进行拉低. 对这个行为的封装有两个好处:

- 不必再分散代码来处理协议级行为; 调用这个 task 的代码比直接操作这些信号的代码要简单.
- 可以在一个地方修改所有协议级行为; 一处修改可以传递到整个 design.

有了 TinyALU 之后, 我们可以模块化 design 的其他部分了.

创建一个模块化验证平台

既然我们有了一个 BFM, 我们可以通过 BFM 创建实现我们三个验证平台操作(测试, 对比和覆盖)并将其连接至 DUT:

```

1 module top;
2     tinyalu_bfm    bfm();
3     random_tester  random_tester_i    (bfm);
4     coverage       coverage_i   (bfm);
5     scoreboard     scoreboard_i(bfm);
6
7     tinyalu DUT (.A(bfm.A), .B(bfm.B), .op(bfm.op),
8                 .clk(bfm.clk), .reset_n(bfm.reset_n),
9                 .start(bfm.start), .done(bfm.done), .result(bfm.result));
10
11 endmodule : top

```

图 14. 使用 BFM 连接验证平台 module

将验证平台拆分成模块的益处马上就出现了. 不仅程序员的意图更明显了, 我们想要修改或增强验证平台也知道从何入手了. 验证平台包括 4 个部分:

- `tinyalu_bfm`--一个包含所有信号级行为的 SystemVerilog interface
- `tester`--一个生成测试激励的模块
- `coverage`--一个提供功能覆盖的模块
- `scoreboard`--一个测试结果的模块

在模块例化的时候, 我们将 SystemVerilog interface 传入. 验证平台只有在 DUT 里有信号级的连接. 你可以看到 DUT 的例化引用了 BFM 内部的信号.

在本书中我们将会实现这 4 个基本结构并将把这个验证平台转换为 UVM 验证平台. 我们已经把之前的验证平台中的每个 loop 提取出来并封装进 module 中.

这里是把 scoreboard 的 loop 换成 module 的例子:

```
1  module scoreboard(tinyalu_bfm bfm);
2      import tinyalu_pkg::*;
3
4      always @(posedge bfm.done) begin
5          shortint predicted_result;
6          case (bfm.op_set)
7              add_op: predicted_result = bfm.A + bfm.B;
8              and_op: predicted_result = bfm.A & bfm.B;
9              xor_op: predicted_result = bfm.A ^ bfm.B;
10             mul_op: predicted_result = bfm.A * bfm.B;
11         endcase // case (op_set)
12
13         if ((bfm.op_set != no_op) && (bfm.op_set != rst_op))
14             if (predicted_result != bfm.result)
15                 $error ("FAILED: A: %0h B: %0h op: %s result: %0h",
16                         bfm.A, bfm.B, bfm.op_set.name(), bfm.result);
17
18     end
19 endmodule : scoreboard
```

图 15. 完成的 scoreboard module

我们通过用 bfm 的层次化的引用来访问 BFM 中的信号. 剩余的逻辑跟大一统的验证平台中的 loop 是一样的.

由于不需要处理信号级协议了, 测试激励 module 现在变得简单了. 它现在只用调用 BFM 中的 task:

```
36      bfm.reset_alu();
37      repeat (1000) begin : random_loop
38          op_set = get_op();
39          iA = get_data();
40          iB = get_data();
41          bfm.send_op(iA, iB, op_set, result);
42      end : random_loop
43      $stop;
44  end // initial begin
45 endmodule : tester
```

图 16. 在 tester module 里使用 BFM

总结

在本章中, 我们把整体的验证平台拆分成便于管理的逻辑块. 这个验证平台现在有 4 个编译单元: tinyalu_bfm interface 以及 tester, scoreboard 和 coverage module.

在后续章节, 我们将用 OOP 来实现这个基本的四部分架构. 为什么呢? 因为 OOP 为编写和维护复杂验证平台提供了强力的工具.

我们将用接下来的 6 个章节来学习 OOP, 之后我们就要开始使用 UVM 了.

第四章

面向对象编程

一说到史蒂夫·乔布斯在 1970 年底在施乐 PARC 进行的"宿命拜访", 人们会想象着他摆弄着"皇冠上的珠宝"(让 Mac 闻名的窗口, 键盘, 鼠标交互界面)的情形. 乔布斯后来回想起 PARC 展示给他的两件东西, 一个是计算机网络, 另外一个是 OOP, 后来被证明是改变游戏规则的家伙. 当 Scully 将乔布斯逐出苹果, 乔布斯开了一家叫 NeXT 电脑的公司, 想要将 OOP 推向市场. 当世首批 web 服务器和浏览器的编写都使用了 NeXT 的库.

OOP 在 1990 年代伴随着 C++ 和 Java 的面世而腾飞. 很快大家都学着使用 OOP 来编程(你要教小朋友编程的话, 就从教 OOP 开始吧).

验证工程师也在 1990 年代开始使用 OOP. 在一开始, 这个是通过把 C++ 链接到 Verilog 仿真器来实现的. 验证平台可以用 OOP 编写, 然后通过 PLI 来驱动 DUT. 同时, Verisity 推出了 e 语言. e 基于 OOP, 提出了面向方向编程并且很快成为了主流的验证语言.

此外在 1997, Co-Design 的同仁把面向对象扩展加入 Verilog, 创造了一个叫 Superlog 的新语言. Co-Design 把 Superlog 捐赠给业界标准组 Accelera. Accellera 将 Superlog 跟另外一个捐赠的语言 Vera 进行合并, 推出了 SystemVerilog.

为什么人人都爱 OOP

三个原因使得 OOP 深受软件工程师和验证工程师的喜爱:

- 代码重用
- 代码维护
- 内存管理

我们轮流来看这三条.

代码重用

举一个简单的微控制器的例子. 一个芯片包含一个微控制器, 寄存器组, 一个 UART 和其他接口. 你不需要知道设计人员怎么实现的, 你只需要知道管脚的信号定义和指令使用. 实际上, 无论给你的微控制器是原厂封装的, 芯片中的一部分或者甚至是编在一个 FPGA 中的, 你都可以用一样的代码.

OOP 中的对象也是一样的道理. 对象包含了数据以及操作该数据的 task 和 function. 一旦你定义了一个对象, 你不需要紧张里面有什么; 你只需要按照文档的指示来使用.

重用意味着你每加一个功能, 你的验证平台就更强大了. 如果你的新功能编写正确了, 你可以在之前的工作之上创建更多新功能.

代码维护

每次你把验证平台中的一部分代码拷到另外的部分，你就是在制造潜在的代码维护灾难。你希望代码在两个地方是一样的，但是你在一个地方发现了一个 bug，你就需要在在程序里搜索并修复另外一个地方。

如果你拷的是别人的代码，这个问题会更严重。在这种情况下，他们修改了自己的 bug，但是很可能不会跟你讲起这个修改。你就不同步了。

OOP 可以解决这个问题。你在一个地方写通用的代码，然后在整个验证平台中都可以访问。如果你使用了别人的代码，别人的修改你也会自动从中受益。

好的面向对象代码是易于维护的。

内存管理

在学习 OOP 的时候，我们会发现自己在创建并向验证平台传递对象。实际上执行的是在代码里分配内存并在多个线程里共享。这个在像是 C 语言这样的传统语言里是一个让人头痛的 bug 易发过程，但是在像 SystemVerilog 或 Java 这样的纯 OOP 语言里是简单的。这太简单了以至于我们都不知道我们干了这些。

总结

OOP 是一个大课题。OOP 的书可以塞满整个书柜。OOP 尽管很酷，不过本书也不会深入到每个细枝末节了。我们把 OOP 学到能利落的使用 UVM 的程度就好了。

当然，这已经包含了很多强大的 OOP 概念了。我们会从类和继承开始。

第五章

类和继承

假设我们在一辆跑长途的轿车上，决定玩游戏，玩问 20 个问题猜谜，我先来。我跟你说我想到的是一个动物。你说这很傻因为完全不用 20 个问题就猜到了，但是我还是坚持要玩。你翻了翻白眼，跟我赌一罐啤酒说可以 10 个问题内答出，然后就开始了。

"是属于什么门的？" 你这样问道，我就发现我被"斯诺克"了。

"这不公平"，我说，想试着小小的耍赖一下。不过打赌就是打赌，后来还是说了是脊索动物门的。这样你就知道这个动物是有脊椎骨的。

"什么纲？"

"哺乳纲。"

这样你就知道我说的动物有毛，有乳腺，有锤骨和砧骨，有耳中的镫骨。

"什么目？"

"啮齿目。"

它有着持续生长的大门牙。

"什么科？"

"科？" 我回答说，"我不知道什么科。"

"你自己都回答不出来问题还怎么玩？"

我用智能手机查了查，找到维基百科上了，然后读道，"鼠科。不过现在开始，你得马上就猜了。"

你知道它是个尖鼻子，长尾巴，小身子的小啮齿动物。

你猜道，"松鼠？"

"不是。"

"田鼠？"

"不是。"

"家鼠？"

"是的。我现在欠你啤酒了。"

我们的猜谜游戏展现了生命科学中一个很重要的概念，那就是分类。分类是从我们星球上无穷无尽的生命从中获取信息的捷径。

在软件里的对象也是这样的。

OOP 使用分量来强制代码重用。就像我跟你说一个动物是啮齿动物，你了解啮齿动物，你就知道很多关于这个动物的东西了。同样的，我跟你说一个软件对象是某个类的，你就知道它能干什么了。

从结构体开始

从熟悉的(结构体)再到不熟悉的(类)来向类探索吧。看看我们用来保存长方形和正方形几何形状数据的两个结构体：

```

1  |     typedef struct {
2  |         int          length;
3  |         int          width;
4  |     } rectangle_struct;
5
6  |     typedef struct {
7  |         int          side;
8  |     } square_struct;

```

图 17. 正方形和长方形结构体

可见结构体可以很好的获取长方形和正方形的信息. 我们用这些的信息来保存展示这些图形的信息:

```

10 | module top_struct ;
11 |     rectangle_struct rectangle_s;
12 |     square_struct      square_s;
13 | initial begin
14 |     rectangle_s.length = 50;
15 |     rectangle_s.width  = 20;
16 |     $display("rectangle area: %0d", rectangle_s.length * rectangle_s.width);
17 |     square_s.side = 50;
18 |     $display("square area: %0d", square_s.side ** 2);
19 |
20 | end
endmodule

```

图 18. 使用结构体

我用 `rectangle_struct` 和 `square_struct` 来定义变量, 然后用这些变量来保存大小并计算面积.

这些结构体的一些情况如下:

- 尽管我们知道长方形跟正方形是有关系的, 不过程序里没有体现.
- 定义变量的时候, 仿真器就为其分配内存了.
- 如果用户不知道公式的话, 就算不了长方形跟正方形的面积了. 等到了我们需要扩展到梯形和菱形时, 这个问题可能就更大了.

定义一个类

这里是一个长方形类的定义, 我们用这个例子来说明 OOP 的一些术语:

```

1   class rectangle;
2       int length;
3       int width;
4
5       function new(int l, int w);
6           length = l;
7           width = w;
8       endfunction
9
10      function int area();
11          return length * width;
12      endfunction
13  endclass

```

图 19. 定义长方形类

首先要注意到我们用 `class` 关键字来定义类. 不像图 17 中的结构体, 我们不用打大括号来包围这个类, 而是用 `endclass` 关键字.

我们定义了 `length` 和 `width` 类变量. 在 OOP 中这个叫做类的数据成员. 大多数类都是有数据成员的.

我们定义了一个用数据成员返回长方形面积的 `area()` 函数. 这个函数是任何可以访问长方形类的用户都可以使用的. 我们把定义在类中的函数或者任务叫做方法. 这一步可以描述为, "长方形类有两个数据成员 `length` 和 `width`, 还有一个方法 `area`."

在分配内存方面, 仿真器对结构体和类是区别对待的. 结构体一声明就分配内存了, 类不是这样的. 用户必须调用类的 `new()` 构造函数来为对象显式分配内存.

在这个例子中, 我们定义了 `new()`, 需要用户提供长度和宽度. 创建长方形的时候不提供这些信息是没道理的, 这也会让我们在编译时而不是运行时得到这个错误.

在 module 里面使用类是这样的:

```

23  module top_class ;
24      rectangle rectangle_h;
25      initial begin
26          rectangle_h = new(.l(50),.w(20));
27          $display("rectangle area: %0d", rectangle_h.area());
28      end
29  endmodule

```

图 20. 使用长方形类

我们首先声明了一个 `rectangle` 类型的变量. 我们把它命名为 `rectangle_h` 因为它是个指向长方形类的句柄. 句柄类似于内存指针, 不同的是句柄不能拿来运算. 这句定义为句柄分配了足够的内存了, 不过还没为对象分配内存.

我们用 `new()` 函数创建一个长度 50 宽度 20 的长方形, 是在这里为对象分配的内存. (函数调用中的点号表达式是 Verilog 中的用法. 根据参数的顺序在构造函数中进行赋值.) 这个叫做例化对象. 我们可以描述这一步为, "例化 `rectangle` 类的一个对象, 并用 `rectangle_h` 变量来保存."

可创建的对象的数量, 仅仅受制于我们使用的计算机内存大小. 我们可以将这些长方形的句柄复制到其他变量上, 可以把句柄保存在数组中, 也可以传入验证平台来控制仿真.

我们操作对象的时候, SystemVerilog 会处理所有的内存分配. 调用 `new()` 的时候为我们的新对象分配内存, 我们不再引用的时候回收内存. 比如, 我们创建了第二个长方形对象并保存

在 rectangle_h 变量中, 仿真器会回收第一个长方形对象的内存.

有了保存在 rectangle_h 变量中的 rectangle 对象之后, 我们可以用 area() 来计算长方形的面积了. 我们在内部使用 \$display():

```
23 module top_class ;
24     rectangle rectangle_h;
25     initial begin
26         rectangle_h = new(.l(50),.w(20));
27         $display("rectangle area: %0d", rectangle_h.area());
28     end
29 endmodule
```

图 21. 使用 Rectangle 计算面积

这个小小的例子展示了 OOP 是如何让工程师们在不了解实现细节的情况下使用其他人的代码. 使用长方形类的工程师只需要在创建时提供长度和宽度然后调用 area() 方法. 对象之后的考虑都由定义这个类的人员完成.

rectangle 类展示了定义类的一些基础. 接下来我们来看看 OOP 中另外一个重要的特性: 继承.

继承类

在之前的结构体例子中, 我们定义了一个长方形结构体和一个正方形结构体. 这两个结构体里没有重用的可能性. 如果我们写了一个用 rectangle_struct 做参数的函数 area(), 就不能用 square_struct 做参数了. 如果这个函数用长度和宽度做参数, 就又破坏了封装的目的了.

OOP 解决这个重用问题就像上面讲的那个猜老鼠游戏一样: 先进行细致的分类. 像啮齿目基于哺乳纲一样, 我们可以基于其他类来创建 SystemVerilog 类.

在这个例子中, 我们注意到正方形就是边长相等的长方形. 这意味着我们可以重用长度, 宽度和面积的概念. 我们用 extends 关键字来获得重用:

```
15 class square extends rectangle;
16
17     function new(int side);
18         super.new(.l(side), .w(side));
19     endfunction
20
21 endclass
```

图 22. 定义正方形类

上面的代码中定义了完全的 square 类. extends 关键字向编译器指示了 square 类默认使用所有 rectangle 类中定义的数据成员和方法.

不同之处是在这个类中重写的数据成员和方法. 在这个例子中, 我们重载了 new() 构造函数. 这个 square 版的 new() 只有一个参数: 边长. 这个函数使用这个边长, 然后使用了 OOP 中的独门绝技, 就是 super.new().

super 关键字指示编译器来显式的引用父类中定义的数据成员和方法. 在 square 这个例子中, new() 函数中 super 关键字指示编译器调用 rectangle 中的构造函数并使用 side 传入长度和宽度参数.

我们在不用拷贝 rectangle 类代码的情况下实现了完全的重用. 我们利用了 square 是 rectangle 的一个特例的性质, 然后让编译器来做这个工作. 下面是使用的代码:

```
27     initial begin
28
29         rectangle_h = new(.l(50),.w(20));
30         $display("rectangle area: %0d", rectangle_h.area());
31
32         square_h = new(.side(50));
33         $display("square area: %0d", square_h.area());
34
35     end
36 endmodule
```

图 23. 使用 square_h 对象

我们声明了 square 的一个变量然后用 new() 来创建对象. 我们调用了 new() 两次, 但是两次运行的是不同的版本: 一个是 rectangle 的一个是 square 的. 这个是 SystemVerilog 的约定(很多其他的语言也是这样的). 编译器知道所有 new() 的调用跟表达式左边的变量类型都是相关的.

总结

在其他类之上定义类并且建立功能拓扑的能力是 OOP 和传统编程范式的区别的关键的区别, 当你把这个能力用在创建多个对象时, 你就拥有了一个可以创建软件和验证平台的强力工具平台.

然而这仅仅是一个开始. 更深入的思考对象和分类会引领你进入一个强大的编程技术新世界. 想一想: 上面这个例子中有一个 rectangle_h 变量和一个 square 对象. 既然 square 是一个 rectangle, 那我们可以用 rectangle_h 变量来存 square 吗?

这是可以的. 这个叫做多态, 我们将在下一章来探讨.

第六章

多态

OOP 带来了很多大概念. 通常这些概念可以用更简单的词来替换. "例化"可以换成"创建", "设计模式"可以换成"编程技巧". "多态"可以换成... 好吧, 可能没别的换的, 这是 OOP 中独有的.

多态这个词源于面向对象的类是从其他类里继承的. 一个正方形就是一个长方形, 一个平行四边形, 一个梯形, 一个多边形. 这就让人要发问了, "我声明了一个多边形类型的变量, 然后例化了一个正方形对象, 那我可以用这个多边形变量来存放正方形对象吗?" 答案是肯定的, 这种语言特性叫做多态.

用动物的例子来解释多态吧. 下面是一个说明类结构的 UML 图:

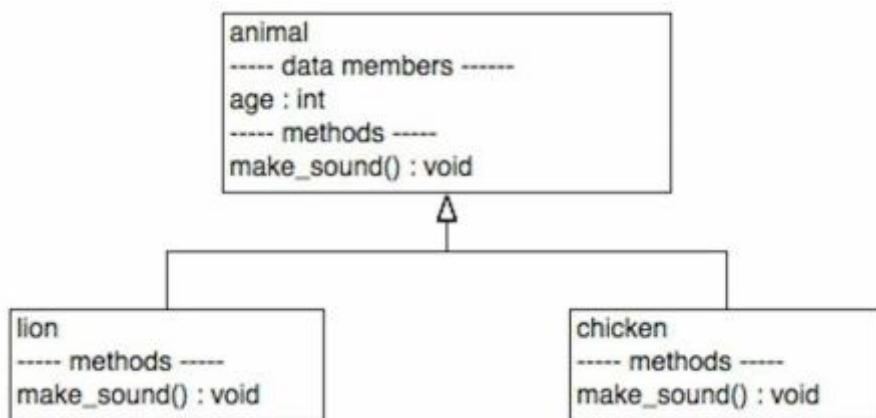


图 24. Animal UML 图

UML 图可以让我们用图像来设计和描述面向对象的类层次. 在上面的例子中, 我们有一个 `animal` 基类和两个基类的继承类 `lion` 和 `chicken`.

所有的动物类都有一个数据成员 `age` 和一个方法 `make_sound()`. 这个方法将动物的叫声打印到屏幕上.

下面是实现 `animal` 类的代码:

```
1  class animal;
2      int age=-1;
3
4      function new(int a);
5          age = a;
6          endfunction : new
7
8      function void make_sound();
9          $fatal(1, "Generic animals don't have a sound.");
10         endfunction : make_sound
11
12     endclass : animal
13
```

图 25. animal 类代码

上面的代码用动物的年纪做参数来创建一个新动物. 这个对象有 `make_sound()`方法, 不过这个不能用在一个泛指的动物上, 所以我们返回了一个致命错误. 你要在一个泛指的动物上使用 `make_sound()` 来得到叫声的话, `$fatal` 会停止这个动作.

现在我们从 `animal` 继承得到 `lion` 类:

```
15 class lion extends animal;
16
17     function new(int age);
18         super.new(age);
19         endfunction : new
20
21     function void make_sound();
22         $display ("The Lion says Roar");
23         endfunction : make_sound
24
25 endclass : lion
```

图 26. lion 类代码

`lion` 从 `animal` 里继承得到了 `age` 数据成员. `new()` 方法将 `age` 传递到 `animal` 的构造函数. 这里重载了 `make_sound()` 函数来弄出狮子的叫声.

`chicken` 代码也做了同样的事情, 只是叫声不一样. 两个类中的 `new()` 方法都调用了父类构造函数并传入 `age`.

这个代码有一个问题: "如果我只是调用 `super.new()` 构造函数的话, 为什么不能直接从基类里面继承这个构造函数呢?" 答案是, 我们的构造函数是有参数的, SystemVerilog 要求我们显式的写出带参数的构造参数, 即使我们只是调用了父类构造函数.

让我们在一个小程序里用一下这些类. 从一个常用模型开始:

```
68     lion_h = new(15);
69     lion_h.make_sound();
70     $display("The Lion is %0d years old", lion_h.age);
71
72     chicken_h = new(1);
73     chicken_h.make_sound();
74     $display("The Chicken is %0d years old", chicken_h.age);
```

图 27. 使用 animal 类的常规方式

这里我们创建了一个 15 岁的狮子和一个 1 岁的小鸡. 然后让他们叫了一下.

结果输出是这样的:

```
# The Lion says Roar
# The Lion is 15 years old
# The Chicken says BECAWW
# The Chicken is 1 years old
```

图 28. 运行代码

目前为止还不错. 我们创建了 `lion` 和 `chicken` 类型的变量, 然后用对象填充. 那要是我们用 `animal` 定义变量 `animal_h` 呢? 我们看看会发生什么. 这里是代码:

```

76     animal_h = lion_h;
77     animal_h.make_sound();
78     $display("The animal is %0d years old", animal_h.age);
79
80     animal_h = chicken_h;
81     animal_h.make_sound();
82     $display("The animal is %0d years old", animal_h.age);

```

图 29. 将 lion 和 chicken 放入 animal_h 变量

我们用 animal_h 变量来存放 lion 和 chicken 对象。这个是合法的，因为 lion 和 chicken 是从 animal 派生的。然后我们调用了 make_sound() 方法，得到的结果是：

```

# The Chicken is 1 years old
# ** Fatal: Generic animals don't have a sound.

```

图 30. 运行时的 fatal error

致命错误的发生是因为 animal_h 调用的是 animal 类中定义的 make_sound()。

某个角度来说，这个也是有道理的。变量调用的是自身的类的方法。不过从另外的角度来看，这又是没道理的。animal_h 变量操作的是 lion 对象。现实中，你把狮子放进标着“动物”标签的笼子里，狮子还是可以吼的。所以为什么我们不能用 animal 类型的变量操作 lion 对象并且让它可以“吼”呢？

其实是可以的，不过我们需要告诉 SystemVerilog 我们通过变量的类型还是存储的对象的类型来使用引用的函数。我们后面就要这么弄了。

虚函数

SystemVerilog 在很多地方都用了 virtual 关键字（在我看来是滥用了）。这个关键字暗示着，用“virtual”定义的东西其实是一个之后才会实现的占位符，或是在其他地方才有效的。

你把一个方法定义为虚方法时，你是在指示 SystemVerilog 忽略变量类型的表面限制，去更深入的查看存放在变量里的对象的类型。然后，你就可以找出真正的方法了。

下面就是定义虚函数的方法：

```

1  class animal;
2      int age=-1;
3
4      function new(int a);
5          age = a;
6          endfunction : new
7
8          virtual function void make_sound();
9              $fatal(1, "Generic animals don't have a sound.");
10             endfunction : make_sound
11
12         endclass : animal

```

图 31. 使用 virtual function 的 animal 类

跟原先的 animal 代码唯一的不同是关键字 virtual。

lion 和 chicken 里的 make_sound() 定义是不需要 virtual 关键字的。他们从基类中继

承了这个行为, 所以他们的代码没有变化. 现在的代码在图 29 中的作用符合我们的预期:

```
# The Chicken says BECAWW
# The Chicken is 1 years old
# The Lion says Roar
# The animal is 15 years old
# The Chicken says BECAWW
# The animal is 1 years old
```

图 32. 运行结果

animal_h.make_sound() 的调用根据变量存放的对象类型返回了不同的结果.

抽象类和纯虚函数

我们之前版本的动物类, 为了解决一个简单的问题使用了一个厚重的方案. 我们希望强制用户去重载 make_sound() 方法, 于是我们编写了调用 \$fatal 的方法版本. 这意味着我们的用户可能会在跑整个仿真时最后得到一个致命错误. 这个在一个大型设计里可能得花上个半小时或者更长时间.

要是能更早的捕捉到这个错误不是更好吗? 这个实际上也是可以的.

我们可以在 SystemVerilog 里定义一种叫抽象类的东西. 抽象类只能用来做基类. 你不能用抽象类来例化一个对象. 不然你会得到一个运行时错误.

在定义抽象类时, 你可以把方法定义为纯虚方法. 这些方法没有实体, 而且在重载类时需要强制重载这些方法. 如果不重载的话会得到编译错误.

抽象类和纯虚函数对继承抽象基类的工程师进行了强制规范. 纯虚函数说, "你想用我的基类吗? 可以的. 这里是写了你需要做什么事情的合同条款."

在这个例子里, animal 类是一个抽象类的绝好例子. 世上没有"泛指动物"这样的事物.

下面是 animal 的抽象版本:

```
1  virtual class animal;
2      int age=-1;
3
4      function new(int a);
5          age = a;
6      endfunction : new
7
8      pure virtual function void make_sound();
9
10 endclass : animal
```

图 33. animal 虚基类

好简单! 我们定义了抽象类 animal, 纯虚函数 make_sound(). 我们不用为 make_sound() 写实体, 因为重载类会去处理.

这是个新方法. 我们首先试着直接例化一个动物对象吧:

```
46      animal animal_h;
47
48      animal_h = new(3);
49
50      lion_h = new(15);
```

图 34. 从虚类继承

跟之前的是一样的, 不过现在我们为 SystemVerilog 引入了一个捕捉错误的方法. 在上面我们尝试例化一个抽象动物, 发生了什么呢:

```
22 # Loading work.top(fast)
23 # ** Fatal: (vsim-8250) Class allocator method 'new' called on Abstract Class.
24 #   Time: 0 ns Iteration: 0 Process: /top/#INITIAL#42(#ublk#0#44) File: pure_virtual.sv
25 # Fatal error in Module top at pure_virtual.sv line 48
26 #
27 # HDL call sequence:
28 # Stopped at pure_virtual.sv 48 Module top
```

图 35. 错误 animal

Questa 仿真器在我们试着创建抽象动物这件无意义的事情的时候正确的给出了一个致命运行错误.

总结

在这一章我们试了一下 OOP 的多态. 这个是我们灵活运用 UVM 的重要武器.

在下一章, 我们将学习编写可以用于整个验证平台的通用方法库. 我们将定义包含静态变量和静态方法的类来达到这个目的. 这个是我们使用 UVM 的另外一个重要工具.

第七章

静态变量和方法

我讨厌全局变量. 为什么呢? 一方面是因为所有的编程书都教导我不要用全局变量, 不过主要还是因为我调试过有大量全局变量的代码. 这是个灾难.

首要的问题在于你难以发现全局变量的类型. 翻到你函数的头, 没发现定义. 翻到程序的开头, 还是没发现定义. 最后你翻遍了代码才找到了这个变量, 然后你才能看到它的类型.

第二个问题在于, 程序的其他部分可以很轻易的修改全局变量的值, 而这个是预期之外的. 你存了 1, 后来发现它变成别的了. 这两个原因结合起来就让我很讨厌全局变量.

其实这个也说明了, 在很多情况下, 特别是在一个验证平台设计里, 全局的数据结构还是很有用的. 比如, 你会在一个地方为 DUT 输入数据, 然后在其他地方读取结果. 你可能会想着把这两部分都存到一个全局的位置然后再用一个另外的自检工具来检查结果. 这是全局变量的合理用法.

我们需要的是易用易维护的全局存储的用法. OOP 通过类定义中的静态变量和静态方法提供了这种功能. 这简化了全局变量和方法的定义, 使用和维护.

声明静态变量

在我们之前讨论类定义的时候, 我说过调用 new() 函数的时候才会为一个对象分配内存. 其实不完全是这样. 如果我们在一个类的成员变量前放上 static 关键字, SystemVerilog 是会区别对待的. 如果我们这么做了, 这个变量的内存在类定义的时候就被分配了.

此外, 无论我们用这个类例化了多少新对象, 这个数据都只有一份拷贝. 所有对象看到的都是同一个成员. 重要的是, 我们就这样可控的创建了一个全局变量.

回到狮子的例子来看看怎么创建使用静态变量. 假如我们决定把狮子都关在笼子里, 这个例子里的"笼子"是指 SystemVerilog 中的队列, 用来把所有的狮子都存放在一个数组里, 然后再读取出来.

我们只能有一个放狮子的笼子, 所以我们不能在 lion 类里创建笼子变量. 我们需要一个带静态变量的新类 lion_cage 来装所有的狮子. lion_cage 是这样的:

```
50 class lion_cage;
51
52     static lion cage[$];
53
54 endclass : lion_cage
```

图 36. lion cage

lion_cage 包含了一个 SystemVerilog 队列并定义为静态变量. 这个队列用于存放 lion 对象.

定义前的 static 关键字意味着我们不用例化这个类就可以访问这个变量. 这个变量只有一份拷贝, 程序一运行, 这个变量就得到了分配的空间. 这样我们就可以任意访问这个狮笼了.

下面是把狮子放进笼子的例子:

```
58      initial begin
59          lion    lion_h;
60          lion_h = new(2, "Kimba");
61          lion_cage::cage.push_back(lion_h);
62          lion_h = new(3, "Simba");
63          lion_cage::cage.push_back(lion_h);
64          lion_h = new(15, "Mustafa");
65          lion_cage::cage.push_back(lion_h);
66          $display("Lions in cage");
67          foreach (lion_cage::cage[i])
68              $display(lion_cage::cage[i].get_name());
69      end
```

图 37. 将 lion 放入笼子

这里我们创建了新的狮子, 然后用静态变量把它们放进了笼子队列. 注意到"::"操作符. 我们通过把::置于类名之后来访问 lion_cage 的变量. 这告诉了编译器我们在类的命名空间下访问静态变量.

我们用 lion_cage::cage 访问变量, 用 push_back()方法在队列里存放狮子.

我们用 foreach 操作符来对队列进行遍历访问. 由于每个节点都存放着一个 lion 对象, 我们在循环里遍历队列实体并用 get_name()方法来打印出狮子的名字.

这里是输出:

```
# Lions in cage
# Kimba
# Simba
# Mustafa
```

图 38. lion 目录

这个例子展示的是代码中全局变量的创建和访问. 这个机制的妙处在于我们可以任意访问变量并随时看到变量是在哪里定义的. 类名就在::操作符前, 到我们想要进行改动的时候, 就知道要从类定义里找了.

不过直接访问原始的静态变量不是好的编程习惯. 因为我们把内容都暴露了, 之后的工程师会有使用困难. 比如, 你可能不知道怎么用 SystemVerilog 的队列, 于是你得去 google 了才会用狮子笼.

我们实际需要的其实是操作静态变量的方法, 掩藏了实现细节, 更便于后来者使用的静态方法.

定义静态方法

见识过我们定义 SystemVerilog 队列为静态变量并对其进行访问的成功共享之后, 我们还是决定对用户隐藏狮子笼的实现细节, 取而代之的是提供笼子的接口. 这会简化对象的使用.

这里就是实现代码:

```

44   class lion_cage;
45
46     protected static lion cage[$];
47
48     static function void cage_lion(lion l);
49       cage.push_back(l);
50     endfunction : cage_lion
51
52     static function void list_lions();
53       $display("Lions in cage");
54       foreach (cage[i])
55         $display(cage[i].get_name());
56     endfunction : list_lions
57
58   endclass : lion_cage

```

图 39. 带接口的 lion cage

新版本比老版本更安全更易用。首先，我们用了 `protected` 关键字来保护静态变量。这个关键字阻止了用户对我们的 SystemVerilog 队列的直接访问，保证了良好的重用。有任何直接访问笼子变量的行为，SystemVerilog 编译器都会报错。这样，你不用修改其他人的代码就能改变这个实现。

我们提供了两个静态可访问方法。

`cage_lion()` 使用一个 `lion` 对象参数并将其放入 SystemVerilog 队列。

`lion_list()` 使用 SystemVerilog 的 `foreach` 循环来列出存放的狮子。

这个代码比先前的简单，因为用户不需要直接去访问笼子(里面放的狮子的时候这很危险的)：

```

65   initial begin
66     lion  lion_h;
67     lion_h = new(2, "Kimba");
68     lion_cage::cage_lion(lion_h);
69     lion_h = new(3, "Simba");
70     lion_cage::cage_lion(lion_h);
71     lion_h = new(15, "Mustafa");
72     lion_cage::cage_lion(lion_h);
73     lion_cage::list_lions();
74   end

```

图 40. 使用静态方法装狮子

这个代码里我们新建了狮子对象并用 `lion_cage::cage_lion()` 方法将其放进笼子。然后我们用 `lion_cage::list_lions()` 列出了所有的狮子。

所有这些方法的调用都访问了一样的地址空间。我们可以在程序的任何地方用全局引用 `lion_cage` 类来访问狮笼。

总结

在本章，我们用类中的静态变量来创建全局资源。我们看到 SystemVerilog 可以让我们可控的创建全局资源，我们也学习了怎么定义访问这些全局资源。

但是 `lion_cage` 有一个问题：只限狮子。如果我们要关小鸡，我们需要把这个代码拷一份。

然后把静态变量的类型改掉。这不符合代码重用规范。

SystemVerilog 用带参数的类来解决这个问题。这个是学会用 UVM 的另外一个关键。下一章来看吧。

第八章

类的参数化定义

在上一章，我们创建了使用静态变量的类，用来装狮子。这个静态类很不错，我们可以合理的运用全局资源了。然而，问题还是出现了。

我们要一个小鸡笼怎么办呢？看 `lion_cage` 的代码，我们会发现需要的 `chicken_cage` 类跟 `lion_cage` 基本是一样的，除了把队列的类型换成 `chicken`。

直觉是把狮子笼的代码拷出来弄一个小鸡笼。不过这样犯了一个错误。一个动物园要多烂才会只有狮子跟小鸡啊。我们肯定是要为其他的动物准备不同的笼子。然后我们还是拷代码？

可以想见的是很多情况下我们都会对笼子进行通用性的改动（比如添加一个 `feed()` 方法），如果我们用的是代码拷贝，这样要改动的话就需要每个地方都改。这个很快就会失控，之后我们不停的因为这些 bug 被问责。

幸运的是，SystemVerilog 的“参数化类定义”可以解决这个问题。更重要的是 UVM 里面大量的使用了这一套，所以我们要把这个搞懂才行。

来创建一个参数化类定义吧。

造一个动物笼子

都还记得原来 Verilog 中的参数吧。参数是我们在例化 module 的时候可以设置的量。最典型的例子是存储器。我们为数据总线和地址总线配置参数，然后用户就可以在同一个模块里例化任意容量的存储器了。这里是个例子：

```
1 module RAM #(awidth, dwidth) (
2     input wire [awidth-1:0] address,
3     inout wire [dwidth-1:0] data,
4     input we);
5
6     initial $display("awidth: %0d  dwidth %0d",awidth, dwidth);
7     // code to implement RAM
8 endmodule // RAM
```

图 41. module 中的 parameter

我们要用同样的方法来解决笼子的问题，不过我们用参数指定类型。这里是代码：

```

60   class animal_cage #(type T);
61       protected static T cage[$];
62
63       static function void cage_animal(T l);
64           cage.push_back(l);
65       endfunction : cage_animal
66
67       static function void list_animals();
68           $display("Animals in cage:");
69           foreach (cage[i])
70               $display(cage[i].get_name());
71       endfunction : list_animals
72
73   endclass : animal_cage

```

图 42. 通用 animal cage

animal_cage 的代码跟 lion_cage 基本一样，除了前者带有一个参数。在第一行可以看见我们定义的 type 参数 T，然后我们用这个参数来定义队列的类型，并用作 cage_animal 的参数类型。

现在我们可以用同样的代码为基础，为不同的动物创建不同的笼子类，在这里的修改会影响到所有的动物笼子。

这个笼子的用法是在用静态方法的时候提供类型：

```

81   initial begin
82       lion    lion_h;
83       chicken chicken_h;
84       lion_h = new(15, "Mustafa");
85       animal_cage #(lion)::cage_animal(lion_h);
86       lion_h = new(15, "Simba");
87       animal_cage #(lion)::cage_animal(lion_h);
88
89       chicken_h = new(1, "Clucker");
90       animal_cage #(chicken)::cage_animal(chicken_h);
91       chicken_h = new(1, "Scratchy");
92       animal_cage #(chicken)::cage_animal(chicken_h);
93
94       $display("--- Lions ---");
95       animal_cage #(lion)::list_animals();
96       $display("--- Chickens ---");
97       animal_cage #(chicken)::list_animals();
98   end

```

图 43. 小鸡和狮子的笼子

这个例子展示了访问狮子笼和小鸡笼的方法。我们把动物的类型方法#之后的括号里面。狮子的话我们用 lion 类作为参数来访问 cage_animal 静态方法。小鸡的话就用 chicken 类作为参数来访问 cage_animal 静态方法。

最后我们得到两个静态队列，一个存着狮子，一个存着小鸡。这推出了参数化类的一个重要性质。每当例化一个参数化类，你用不同的参数创建的类都是不同的。animal_cage#(lion) 和 animal_cage#(chicken) 是完全不同的命名空间下的类，只不过代码是一样的。

两次 list_animals() 的调用输出显示了这个区别：

```
# --- Lions ---
```

```

# Animals in cage:
# Mustafa
# Simba
# -- Chickens --
# Animals in cage:
# Clucker
# Scratchy

```

图 44. 所有的动物

你会注意到我们看到的不是一个笼子 4 个动物，而是两个笼子，每个笼子里有两个动物。狮子跟小鸡是分开的。

读到这里，你们可能会觉得我们只能把参数化的类用在静态变量和静态方法上。先不看这个，我们来看另外一个造笼子的方法。

用参数来定义变量

UVM 里会用静态方法和参数，或者两个混用，不过 UVM 里面参数化类型用途最多的地方还是变量定义。我们来改一下动物笼子来展示一下。

这个例子中，我们不用静态方法来访问动物笼子了，我们要开始用例化的动物笼子对象来存放其他的动物。这里是新的 animal_cage 代码：

```

60   class animal_cage #(type T);
61
62     protected T cage[$];
63
64     function void cage_animal(T animal_h);
65       cage.push_back(animal_h);
66     endfunction : cage_animal
67
68     function void list_animals();
69       $display("Animals in cage:");
70       foreach (cage[i])
71         $display(cage[i].get_name());
72     endfunction : list_animals
73
74   endclass : animal_cage

```

图 45. animal cage

跟之前的唯一区别是上面的代码去掉了变量和方法之前的 static 关键字。我们需要声明例化操作对象的变量。看下面：

```

78   module top;
79
80     lion  lion_h;
81     chicken  chicken_h;
82
83     animal_cage #(lion)  lion_cage;
84     animal_cage #(chicken)  chicken_cage;
85

```

图 46. 声明 cage

上面的代码跟之前的也很像，不过我们有了两个新变量：lion_cage 和 chicken_cage。我们在声明变量的时候传入参数。编译器根据传入的参数类型来生成一个新的类，然后用来声明变量。如果你用了跟动物笼子不匹配的类型，比如说一个没有 get_name()方法的对象，这样会得到语法错误。

现在我们有动物对象和动物笼子对象的变量了，接着来用 new() 来例化变量吧：

```
88     initial begin
89         lion_cage = new();
90         lion_h = new(15, "Mustafa");
91         lion_cage.cage_animal(lion_h);
92         lion_h = new(15, "Simba");
93         lion_cage.cage_animal(lion_h);
94
95         chicken_cage = new();
96         chicken_h = new(1, "Little Red Hen");
97         chicken_cage.cage_animal(chicken_h);
98
99         chicken_h = new(1, "Lady Clucksalot");
100        chicken_cage.cage_animal(chicken_h);
101
102
103        $display("-- Lions --");
104        lion_cage.list_animals();
105        $display("-- Chickens --");
106        chicken_cage.list_animals();
107    end
```

图 47. 使用 animal cage

上面的代码例化了一个新的狮子笼对象，新建了两个狮子（木莎法和辛巴），然后关进笼子。接着是对小鸡进行一样的操作，例化新的 chicken_cage 然后关进两只小鸡。

打印出来的动物是：

```
# -- Lions --
# Animals in cage:
# Mustafa
# Simba
# -- Chickens --
# Animals in cage:
# Little Red Hen
# Lady Clucksalot
```

图 48. 查看 animal

这个代码跟之前的效果是一样的，一个大区别是我们不能在验证平台的任意位置访问笼子了。我们创建了新的笼子对象然后放进动物，task 结束时，对象就消失了。

总结

在这章我们学习了 UVM 的基础：参数化的类。我们对 SystemVerilog 中的 OOP 已经了解的差不多了。

在下一章，我们要把这些 OOP 特性集合起来，实现一个 Factory 类。因为 Factory 设计模式是 UVM 的一个关键部分，这样的话我们就能了解这是怎么回事了。

此外，这是很简洁的。

第九章

Factory 模式

在很多方面 OOP 都让我联想到国际象棋. 学象棋分为两步. 首先学步法: 车怎么走, 皇后怎么走, 怎么吃过路兵. 然后是用步法来组合: (应该是很多阵形啦, 自己体会吧).

学 OOP 也是一样. 我们已经学了步法: 类继承, 多态, 静态方法, 参数化类. 现在我们要学习一些组合和编程技巧了.

OOP 工程师喜欢发明词汇. 他们用"设计模式"来指代"编程技巧". 就像象棋中的阵形一样, 设计模式是用来描述程序员的解决方案架构的词汇. 来看看 Factory 模式.

Factory 模式是 UVM 中最显眼的设计模式. 之后我们会用它来创建动态的灵活的验证平台. 由于 Factory 模式的广泛应用, 我们将在这章创建一个简单的 Factory 来展示这个模式是如何工作的.

为什么用 Factory 模式?

Factory 模式强调了我们先前例化对象的限制. 看看我们之前例化对象的例子:

```
90     lion_h = new(15, "Mustafa");
91     lion_cage.cage_animal(lion_h);
92     lion_h = new(15, "Simba");
93     lion_cage.cage_animal(lion_h);

94
95     chicken_cage = new();
96     chicken_h = new(1, "Little Red Hen");
97     chicken_cage.cage_animal(chicken_h);

98
99     chicken_h = new(1, "Lady Clucksalot");
00     chicken_cage.cage_animal(chicken_h);
```

图 49. 使用 new() 方法例化对象

创建所有的对象都用了 new(). 这些调用都写在了代码里. 要创建不同的对象集合或者是增加新的动物类型, 我们都需要修改源码. 这是个严重的限制.

我们称这个为"Hardcoding". 你可能会说, "我看到你编写这些动物的例化了, 要是我想改变这些动物怎么办?"

好问题.

考虑一个情况吧, 我们需要从一个文件里读取动物的列表, 然后例化对象来代表它们. 这是个选择动物的动态方式, 而这个基本是不能用 Hardcoding 来实现的. 我们得用脚本来把文件转换成代码, 然后编译运行. 这样每次数据有更改的时候我们就要重新编译. 这样不好.

需要写程序来动态生成数据的话就更糟了. 比如, 我们要随机选个动物, 或者根据其他动物对动物的数量进行一个平衡. 如果我们用 Hardcoding 来写构造器的话是弄不出来的.

Factory 模式就是用来解决这个问题的.

创建动物 Factory

在 Factory 模式中, 我们希望传递一个参数到方法里然后就得到了我们指定的对象. 在我们的简单例子里用字符串作为参数. 我们要创建生成动物的 Factory. 在此之前, 回忆一下动物类的层次结构:

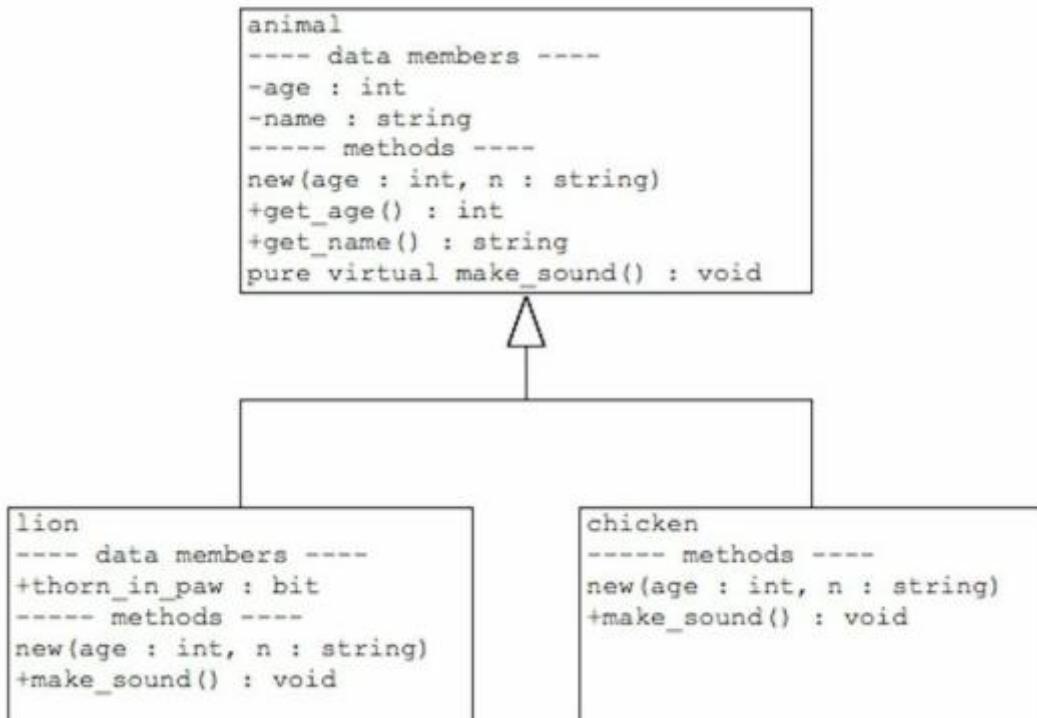


图 50. animal 类层级

我们看到这里 `lion` 类和 `chicken` 类都继承了 `animal` 类, 回想一下前面的多态一章里两个类重载了纯虚函数 `make_sound()` 和构造函数 `new()`. 这些是跟基类唯一的不同.

既然我们可以用 `animal` 类型来存放任意的动物, 我们就可以写一个返回一个 `animal` 变量的函数. 我们以此来创建 Factory:

```
50 |     class animal_factory;
51 |
52 |         static function animal make_animal(string species,
53 |                                         int age, string name);
54 |             chicken chicken;
55 |             lion lion;
56 |             case (species)
57 |                 "lion" : begin
58 |                     lion = new(age, name);
```

图 51. 声明 animalFactory

我们新建了 `animal_factory` 类. 这个类包含 `make_animal` 方法. `make_animal` 方法用种类名字符串, 年龄和动物名称做参数. 它会创建种类名指定的动物并返回为 `animal`.

`make_animal` 方法里有两个 `chicken` 和 `lion` 变量. 我们会用这两个来调用构造函数来得到正确的动物类型:

```

54     chicken chicken;
55     lion lion;
56     case (species)
57         "lion" : begin
58             lion = new(age, name);
59             return lion;
60         end
61
62         "chicken" : begin
63             chicken = new(age, name);
64             return chicken;
65         end
66
67         default :
68             $fatal (1, {"No such animal: ", species});
69
70     endcase // case (species)

```

图 52. 创建新 animal

Factory 的关键就在于 case 语句, 这个语句根据 species 字符串来创建动物(如果是不存在的动物就返回致命错误). 由于函数的返回值是 animal, 所以返回 lion 对象和 chicken 对象都是可以的.

有了这个动物 Factory, 我们就可以动态的创建不同的动物了.

使用动物 Factory

animal_factory 有一个静态方法, 我们可以在验证平台中任意使用. 先来用 initial 块来测试一下:

```

99     initial begin
100        animal animal_h;
101        lion   lion_h;
102        chicken chicken_h;
103
104        animal_h =
105            animal_factory::make_animal("lion", 15, "Mustafa");
106        animal_h.make_sound();
107        if (animal_h.thorn_in_paw) $display("He looks angry!");

```

图 53. 创建 animal

我们声明了三个变量: animal_h, lion_h 和 chicken_h.

我们在 Factory 中调用 make_animal() 来生成一个 15 岁的木莎法. Factory 用多态来创建了一个狮子并保存在 animal_h 里. 让我们来花些时间来用用多态的特性吧.

首先我们用 animal_h 变量调用 make_sound(). 这个是可以的, 因为 make_sound() 是个虚函数. 仿真器会调用 lion 类的方法, 因为 animal_h 中保存的是狮子对象.

然而, 我们想用 lion 类中的 thorn_in_paw 数据时, 就感受到多态的限制了. 编译器给出了错误:

```

5 # ** Error: factory.sv(107): Field/method name (thorn_in_paw) not in 'animal_h'
6 # ** Error: /tools/mentor/questa/10.1c_1/questasim/linux/vlog failed.

```

图 54. 语法错误

编译器给出的错误解释了问题所在. 我们是在 lion 里定义的 thorn_in_paw 变量, animal 中没有. 能访问 make_sound() 是因为两个类中都定义了.

要访问 lion 类独有的数据成员, 我们需要把动物成员转换成狮子成员. 这个叫做"casting", 操作如下:

```
105     animal_h =
106         animal_factory::make_animal("lion", 15, "Mustafa");
107     animal_h.make_sound();
108
109     cast_ok = $cast(lion_h, animal_h);
110     if ( ! cast_ok )
111         $fatal(1, "Failed to cast animal_h to lion_h");
```

图 55. 将 Factory 的生成结果转换为 lion

系统调用\$cast 指示仿真器把第二个参数的类型(animal 类的 animal_h)转换为第一个参数的类型(lion 类的 lion_h), 同时进行拷贝. 不过这个只在目标类是被转换类的子类时有效. 就是说你不能把 lion 转换成 chicken.

\$cast 转换成功时返回 1'b1, 失败时返回 1'b0. 所以最好每次都检查一下返回值. 在这个例子中, 我把返回状态存到 cast_ok 中, 然后再检查. 不过这个是可以合并的.

我们可以把转换跟 Factory 调用合并起来然后进行错误检查:

```
116     if (!$cast(lion_h, animal_factory::make_animal("lion", 2, "Simba")))
117         $fatal(1, "Failed to cast animal from factory to lion_h");
118
119     animal_cage#(lion)::cage_animal(lion_h);
120
121     if (!$cast(chicken_h ,animal_factory::make_animal("chicken", 1, "Clucker")))
122         $fatal(1, "Failed to cast animal factory result to chicken_h");
123
124     animal_cage #(chicken)::cage_animal(chicken_h);
```

图 56. 将 animal 转换为正确的变量

这样我们可以用 Factory 成功的创建动物了.

总结

在这个 Factory 例子中, 我们学会了如何在不用修改源码的情况下动态的创建不用类型的对象. 实际上我们还是用 Hardcoding 向 Factory 传入了字符串"lion"和"chicken", 但是这个是很容易从一个文件中读取出来的.

在我们开始用 UVM 搭建验证平台的时候, 我们就会用到 Factory 模式来动态创建验证平台组件. UVM 的 Factory 可以让我们重载对象, 就是你要的是个普通的"狮子", 结果得到了"东山狮".

我们现在已经了解了用 OOP 搭建验证平台所有的概念, 现在回到 TinyALU 然后开工吧.

第十章

面向对象的验证平台

我们等候许久的时刻到来了：创建面向对象的验证平台。

就搭建一个平台而已嘛，为什么要先学 OOP 然后用这个范式来写这么大费周章呢？老的方法不好吗？

是这样的。

传统的 Verilog 对付像 TinyALU 这样简单的设计还是可以的。更复杂一点设计的比如 Wishbone 转 I2C 模块也还行。再复杂一点的，像是上百个端口的以太网交换机或者带三层缓存的 CPU 这样的设计就不行了。

到那个时候，你会需要实现复杂的预测器，可扩展的覆盖率工具还有强大的激励生成器。你需要一个通用的框架执行从模块到系统的测试，这需要一个大团队。最后，这些东西都得是可重用的，不然每个项目都写一个验证平台工作量太大了。

有着清晰的封装，强制的重用规则和灵活的内存管理的 OOP 是处理这种程度的复杂度的唯一工具。

TinyALU 验证平台对象

在本书里，我们会把这个简单的 TinyALU 验证平台转换成完全的 UVM 验证平台。我们会用一个具体的例子来讨论 UVM 中的各种概念。

我们第一步要做的是把第三章中 module 组成的验证平台换成基于对象的验证平台。这个面向对象验证平台由一个 module 和 4 个类组成：

- top--例化 testbench 类的顶层模块
- testbench--顶层类
- tester--激励驱动
- scoreboard--检查器
- coverage--收集功能覆盖信息

这些类会随着课程的进行，更多 UVM 特性的引入而变化，不过它们的基本职能是不变的。我们先来看顶层 module 把。

基于对象的顶层 module

在这个基于 module 的验证平台中，我们用 SystemVerilog 的 interface 来连接各个模块。在创建基于对象的验证平台时我们也要这么弄。每个类都会得到一个 interface 的拷贝，这样它就可以操作信号了。

顶层验证平台做了三件事情：

- 引入类定义

- 例化 DUT 和 BFM, 定义验证平台类变量
- 例化并启动验证平台类

这是所有基于对象的验证平台都要完成的事情, 尽管实现细节不同. 来看看这个顶层 module.

引入类定义

在搭建基于对象的验证平台的时候, 我们把所有的类定义和共享资源都放在 SystemVerilog 包里. 用了 package 就可以在多个 module 中共享类和变量定义了. 当你引入 package 时, 你可以访问 package 中所有的定义和声明的数据.

我们的 package 叫 tinyalu_pkg. 里面定义了所有的类. 我们通过在 module 中引入 package 来访问所有的定义:

```
module top;
  import tinyalu_pkg::*;
  `include "tinyalu_macros.svh"
  tinyalu DUT (.A(bfm.A), .B(bfm.B), .op(bfm.op),
```

图 57. 引入 tinyalu_pkg 并 include TinyALU 宏

tinyalu_pkg 定义了四个验证平台中的类, tinyalu_macros.svh 文件定义了一些有用的宏. 我们后面会看到 UVM 也是用了这种 package/宏的方式来传递功能特性.

例化 DUT 和 BFM, 定义验证平台类变量

我们已经在 package 里定义了验证平台变量了. 在顶层 module, 我们例化了 DUT 和 BFM 并声明验证平台变量:

```
tinyalu DUT (.A(bfm.A), .B(bfm.B), .op(bfm.op),
             .clk(bfm.clk), .reset_n(bfm.reset_n),
             .start(bfm.start), .done(bfm.done), .result(bfm.result));

tinyalu_bfm      bfm();
testbench      testbench_h;
```

图 58. 声明类变量, 例化 DUT 和 BFM

上面的代码跟基于 module 的代码有相似之处, 除了我们把激励, 自检, 功能覆盖模块换成了 testbench 类. 我们现在有存放 testbench 对象的 testbench_h 变量了.

例化并启动 testbench 对象

现在我们需要例化验证平台对象并传入 BFM 句柄. 之后, 我们用 testbench_h 的一个方法启动验证 TinyALU:

```

14      initial begin
15          testbench_h = new(bfm);
16          testbench_h.execute();
17      end
18
19  endmodule : top

```

图 59. 例化并启动验证平台

我们用 new() 函数传入 BFM 句柄来新建一个验证平台。这个跟例化 module 的时候把 BFM 放进端口列表里相似。然后我们调用 testbench_h.execute() 来验证 TinyALU。

由于验证平台有一个 BFM 的拷贝，里面就可以用 BFM 的 task 来驱动激励，然后我们就可以观察这些信号来检查输出和覆盖率了。

下面，我们来看看验证平台中的类。

testbench 类

面向对象的验证平台有一个例化其他对象并把这些对象连接起来然后调用方法的顶层类。下一章我们会用 UVM 来处理这些，不过现在我们要自己做。testbench 就是做这个的。

声明

testbench 类位于验证平台类结构的顶层。里面定义声明了其他验证平台模块的变量，然后例化他们并启动他们。首先要处理的是声明。做这个的时候我们会学到无处不在的 virtual 的另外的用法。

```

1  class testbench;
2
3      virtual tinyalu_bfm bfm;
4
5      tester    tester_h;
6      coverage  coverage_h;
7      scoreboard scoreboard_h;
8
9      function new (virtual tinyalu_bfm b);
10         bfm = b;
11     endfunction : new

```

图 60. 声明和构造函数

testbench 的类定义包含了神秘的 "virtual tinyalu_bfm bfm"。这个 virtual tinyalu_bfm 是什么呢？

在 OOP 里这个相当于 module 里的端口列表。

回顾基于 module 的代码，我们知道 SystemVerilog 接口是在验证平台的 module 间传递信号的独立的编译单元。tester, scoreboard 和 coverage module 通过模块的端口列表得到 BFM 的拷贝。

对象可以通过 SystemVerilog 接口的句柄来用同样的方式访问信号。virtual 声明指示编译器这个变量会得到一个接口的句柄。virtual 关键字指示了编译时接口句柄是不是在 bfm 变量里

的, 后面有人要仿真这些信号的时候会把句柄放进变量里.

有很多方式可以把接口的句柄存入 bfm 变量里. 这个例子里, 我们用 new()方法来做. 后面的类版本会用其他方式.

一个面向对象的验证平台使用类和对象而不是 module 来验证 DUT. 我们声明了三个变量 (tester_h, coverage_h 和 scoreboard_h) 来存放三个验证平台对象.

execute()方法

我们声明了测试器, 覆盖率收集器和检查器的变量之后, 需要例化并启动这些对象. 所有的验证任务都用 execute()这个 task 完成. 这是我们在顶层 module 的 initial 块里执行的.

execute() task 例化验证平台的对象然后调用他们的 execute()方法:

```
13   task execute();
14     tester_h    = new(bfm);
15     coverage_h  = new(bfm);
16     scoreboard_h = new(bfm);
17
18   fork
19     tester_h.execute();
20     coverage_h.execute();
21     scoreboard_h.execute();
22   join_none
23 endtask : execute
24 endclass : testbench
```

图 61. 启动验证平台对象

execute()方法例化了三个验证平台对象, 每个都传入了 bfm 拷贝. 然后用 fork-join_none 结构来为每个对象生成线程来启动对象内的 execute(). 这跟在基于 module 的验证平台中例化三个 module, 然后每个里用 initial 和 always 模块是一样的.

tester 类

测试器用随机指令对 TinyALU 进行激励, 最后得到全部的功能覆盖. 基于类的测试器跟基于 module 的测试器是一样的, 不过有三处不同: 用的是类的定义; 用变量来访问 BFM 而不是端口列表; 用 execute()方法而不是 initial 块.

这里是类定义:

```
1  class tester;
2
3   virtual tinyalu_bfm bfm;
4
5   function new (virtual tinyalu_bfm b);
6     bfm = b;
7   endfunction : new
8
```

图 62. tester 定义

上面的代码定义了 tester 类, 定义了操作 BFM 的变量, 这个变量在构造函数里赋值.
接着我们创建 execute()方法. 我们用顶层类来调用这个方法:

```
task execute();
    byte      unsigned      iA;
    byte      unsigned      iB;
    shortint  unsigned      result;
    operation_t          op_set;

    bfm.reset_alu();
    op_set = rst_op;
    iA = get_data();
    iB = get_data();
    bfm.send_op(iA, iB, op_set, result);
    op_set = mul_op;
    bfm.send_op(iA, iB, op_set, result);
    bfm.send_op(iA, iB, op_set, result);
    op_set = rst_op;
    bfm.send_op(iA, iB, op_set, result);
    repeat (10) begin : random_loop
        op_set = get_op();
        iA = get_data();
        iB = get_data();
        bfm.send_op(iA, iB, op_set, result );
        $display("%2h %6s %2h = %4h",iA, op_set.name(), iB, result);
    end : random_loop
    $stop;
endtask : execute
```

图 63. execute()方法

这个 execute()方法跟原来的 module 中的 initial 块的内容是一样的, 生成 1000 个随机 transaction 然后调用\$stop 结束仿真.

scoreboard 类

scoreboard 类跟 module 版的几乎是一样的. 唯一的不同是 class 关键字和 BFM 的传入方式.

```
class scoreboard;
    virtual tinyalu_bfm bfm;
function new (virtual tinyalu_bfm b);
    bfm = b;
endfunction : new
task execute();
    shortint predicted_result;
    forever begin : self_checker
        @(posedge bfm.done)
        case (bfm.op_set)
            add_op: predicted_result = bfm.A + bfm.B;
            and_op: predicted_result = bfm.A & bfm.B;
            xor_op: predicted_result = bfm.A ^ bfm.B;
            mul_op: predicted_result = bfm.A * bfm.B;
        endcase // case (op_set)
        if ((bfm.op_set != no_op) && (bfm.op_set != rst_op))
            if (predicted_result != bfm.result)
                $error ("FAILED: A: %0h  B: %0h  op: %s result: %0h",
                        bfm.A, bfm.B, bfm.op_set.name(), bfm.result);
    end
endtask : execute
```

```
end : self_checker  
endtask : execute  
endclass : scoreboard
```

图 64. scoreboard 类

module 版的比较器使用 always 块, 敏感列表中是 done 的上升沿. 我们在 execute() 中用 forever 和 wait 表达式来重新实现这个功能.

其他的代码是一样的.

coverage 类

coverage 类跟之前 module 里一样, 都是定义了两个覆盖组然后进行收集. 唯一的不同就是一个在类里做, 一个在 module 里.

在 module 里, 我们定义了 covergroup, 然后声明了变量, 把 covergroup 当作类型了. 变量声明的时候 covergroup 就创建了, 然后我们可以用变量的 sample() 方法.

在类中我们不用声明 covergroup 变量, 不过我们要用构造器来创建:

```
82     function new (virtual interface tinyalu_bfm b);  
83         op_cov = new();  
84         zeros_or_ones_on_ops = new();  
85         bfm = b;  
86     endfunction : new
```

图 65. coverage 构造器

这里的构造器跟用与其他构造器一样的方式处理 BFM, 另外还用 new() 来例化 op_cov 和 zeros_or_ones_on_ops 两个 covergroup.

模块集合

我们已经见证了整个基于对象的验证平台了. 顶层 module 例化 DUT 和作为 BFM 的 SystemVerilog 接口, 在例化中连接 DUT 和 BFM.

顶层的 initial 块创建了验证平台对象并在 new() 方法中传入 BFM 句柄. 这个顶层验证平台对象创建验证模块对象并为每个对象传入 BFM 句柄的拷贝.

之后验证平台对象用 fork-join-none 块来多线程启动模块对象的 execute(). 我们的验证平台跟之前的 module 运行起来是一样的. 不同之处在于我们获得了 OOP 中的灵活性和重用性.

总结

本章我们学习了使用对象而不是 module 来构建验证平台. 我们知道顶层 module 需要在验证平台中声明例化对象, 然后在多个进程中启动这些对象.

我们已经可以开始用 UVM 了. 在本书的剩下内容里, 我们将根据本章的基本思想来构建更强大的基于对象的验证平台.

这就是我们期待的, 开始学习 UVM Test 吧.

第十一章

UVM Test

我们已经了解了很多验证平台设计, SystemVerilog 接口, OOP 了, 可以开始用 UVM 创建验证平台了. 先从 UVM Test 开始.

验证团队需要在不重新为每个测试用例编译验证平台的情况下, 对一个设计运行上千个测试用例. 假如一个团队要运行 1000 个测试用例, 每次测试用例的验证平台编译要用 5 分钟, 那就是 5000 分钟或者说 3 天半的编译时间, 这意味着我们要花半个星期编译这个设计.

我们需要的, 是编译一次验证平台, 然后可以用不用的参数运行来构造上千个测试用例.

UVM 就可以让你实现动态配置的验证平台. 它可以让你通过定义类, 为不同的测试用例化不同对象的方式来构造验证平台.

上一章中的 TinyALU 验证平台基本都是用 Hardcoding 写出来的. 如果我们要用我们的测试器的随机激励作出不同的测试激励来运行测试用例的话, 我们需要重写测试器对象然后重新编译.

我们从把 Hardcoding 的 TinyALU 验证平台换成动态验证平台来开始 UVM 的学习.

先从一次编译运行多个测试用例这个问题开始吧.

使用 Factory 模式创建测试用例

在第 9 章, 我们探讨了 Factory 模式. 这是个让我们可以根据传入的数据(比如字符串)在运行时动态的创建对象的 OOP 技巧.

我们对 Factory 模式进行探讨是因为这是 UVM 中的一个重要特性. UVM 可以让我们用它的 Factory 来创建任何东西, 这样就可以搭建灵活动态的验证平台了.

这个例子里, 我们将用 UVM 的 Factory 来启动不同的测试用例. 我们希望可以用下面的方式来运行命令:

```
vsim testbench -coverage +UVM_TESTNAME=add_test  
vsim testbench -coverage +UVM_TESTNAME=random_test
```

图 66. 一次编译多次运行

在上面的例子中, 我们编译这个验证平台一次, 然后用不同的测试用例名传入来运行不同的测试用例. 这个验证平台可以根据这些不同的名字来启动不同的测试用例.

我们通过用 UVM 定义两个类: add_test 和 random_test. UVM 就用 UVM_TESTNAME 字符串来调用 Factory 来创建这些类的实例. 例化完成之后就启动仿真了.

用 UVM 启动仿真

"UVM 读出+UVM_TESTNAME 参数, 然后创建对象并运行仿真"说起来很容易, 不过直接根据这个描述是写不出代码来的. 我们需要知道 UVM 是怎么做这个的, 然后我们才可以写出合适的类. 我们也得知道怎么用 UVM 启动测试用例.

在用 UVM 的时候, 我们会发现它处理了大多数普通的跟仿真验证平台相关的任务. 比如,

基于对象的验证平台都需要例化 testbench 的等价顶层类, UVM 就自动帮我们完成了. 我们会发现它也自动完成了 testbench 类中的对象例化和多线程并行启动.

不过现在的话, 我们还是先看看 UVM 怎么自动创建 testbench 的等价对象吧.

在前文中的基于对象的验证平台里, 我们仿真了一个例化顶层类(testbench), 然后调用 testbench.execute()方法来启动测试用例的 module. UVM 会为我们做这些事情的.

在新 module 中, 我们做了两件事: 保存一个 BFM 的句柄然后调用 run_test()方法:

```
1  module top;
2      import uvm_pkg::*;
3      `include "uvm_macros.svh"
4
5      import tinyalu_pkg::*;
6      `include "tinyalu_macros.svh"
7
8      tinyalu_bfm      bfm();
9      tinyalu DUT (.A(bfm.A), .B(bfm.B), .op(bfm.op),
10                  .clk(bfm.clk), .reset_n(bfm.reset_n),
11                  .start(bfm.start), .done(bfm.done), .result(bfm.result));
12
13     initial begin
14         uvm_config_db #(virtual interface tinyalu_bfm)::set(null, "*", "bfm", bfm);
15         run_test();
16     end
17
18 endmodule : top
```

图 67. 顶层 module 中使用 UVM

这个顶层 module 展示了 UVM 的典型用法. 我们首先引入 uvm_pkg 和相关的宏, 里面包含了所有的 UVM 类定义, 方法和对象. 接下来我们引入 tinyalu_pkg 和相关的宏, 这里面包含了我们的类定义和 BFM 变量.

这种验证平台的 package/macro 组合在业界很是典型.

例化 BFM 和 DUT, 跟之前一样.

启动测试用例的 initial 块, 跟之前不一样. 第一个要注意的地方是传递 BFM 句柄的方式.

在之前的例子中, 我们用构造器来传入 BFM 句柄. 这里不行了, 因为 UVM 需要在构造器里指定其他参数. 我们用的是 UVM 的 uvm_config_db 类.

UVM 开发者用参数化类和静态类来简化整个验证平台的全局信息存储和组织. 这个系统跟动物笼类一样, 不过这里调用的是 set.

set 的前两个参数是 null 和*, 它们指示 set 这个数据在整个验证平台都有效. 第三个参数是存储在数据库里为数据命名的字符串, 这里是 tinyalu_bfm 的句柄.

把 BFM 存入全局位置之后, 我们就可以开始测试用例了. 在文件上方引入的 uvm_pkg 里定义了 run_test() task, 我们这个来启动测试用例.

run_test() task 从仿真器的命令行+UVM_TESTNAME 里读出参数, 然后对这个类名例化对象. 不用+UVM_TESTNAME 的话可以用测试用例名字符串作为 run_test()的参数传入, 不过这个就违背了一次编译运行多项测试用例的原则了.

现在我们完成了一次编译支持运行多项测试用例的第一步. 上面的 run_test()方法从命令行得到一个(类名)字符串, 然后用 UVMFactory 来创建这个测试用例类的对象, 然后这个对象运行了测试用例.

我们还需要定义 random_test 和 add_test 类才能让这个平台工作, 后面来做这个吧.

定义并注册 UVM Test

上面我们看到了 UVM 用 Factory 和命令行中的字符串来创建顶层的 Test 对象然后启动这个对象. 我们需要定义这个 Test 类. 先从 random_test 开始:

```
class random_test extends uvm_test;
  `uvm_component_utils(random_test);

  virtual interface tinyalu_bfm bfm;

  function new (string name, uvm_component parent);
    super.new(name,parent);
    if(!uvm_config_db #(virtual interface tinyalu_bfm)::get(null, "*", "bfm", bfm))
      $fatal("Failed to get BFM");
  endfunction : new
```

图 68. random_test 类的顶部

首先, random_test 继承了 uvm_test 类. 这个 uvm_test 是 uvm_component 类的子类, 所以 random_test 也是 uvm_component 的子类.

从 uvm_component 继承的子类的 new()方法要遵从限制条件:

- 构造器必须按顺序定义 name 和 parent 两个参数, name 在前, parent 在后.
- name 参数必须是字符串. parent 参数必须是 uvm_component 类型.
- 构造器的第一个可执行行必须调用 super.new(name, parent), 然后 UVM 才能正常工作.

对父类完成上面的工作之后, 构造器剩下的部分就可以自由发挥了. 我们调用了 uvm_config_db 的 get()方法来取得 bfm 的句柄. 注意我们传入了 set()方法中用过的"bfm"字符串和 BFM 句柄变量. get()方法接收失败的话会返回 0, 我们对返回状态进行测试, 失败的话就退出仿真.

定义类还需要做最后一件事情, 注册到 Factory. 回顾我们的动物 Factory 的话你会发现我们是用 Hardcoding 编写出来的, Factory 能处理的动物种类. 要增加动物到 Factory 的话只能改 Factory 的源码.

UVM 开发者不希望我们改他们的代码, 于是他们创建了用 uvm_component_utils 宏的机制来解决这个问题. 我们在 class 表达式之后就使用了这个宏, 这个宏在 Factory 里注册了 random_test 类. 现在 Factory 可以创建 random_test 对象了.

run_test()方法

回顾 OOP 章节, 你会看到 module 例化了 test 对象并调用 tb.execute()启动测试. UVM 也做了相似的事情. 我们在顶层 module 调用 run_test(), UVM 用 Factory 创建并通过调用 run_phase()方法来启动 Test 对象. UVM 的 uvm_test 定义里有 run_phase()方法, UVM 用 run_phase()来执行我们的 Test.

我们的 Test 要工作的话必须重载 run_phase()方法. 这个方法名必须是 run_phase()然后必须有一个 uvm_phase 类型的 phase 参数. 这里是我们的 run_phase()定义:

```

task run_phase(uvm_phase phase);
    random_tester    random_tester_h;
    coverage   coverage_h;
    scoreboard  scoreboard_h;

    phase.raise_objection(this);

    random_tester_h = new(bfm);
    coverage_h = new(bfm);
    scoreboard_h = new(bfm);

    fork
        coverage_h.execute();
        scoreboard_h.execute();
    join_none

        random_tester_h.execute();
    phase.drop_objection(this);
endtask : run_phase

endclass

```

图 69. run_phase()方法

UVM 在创建对象后调用 run_phase()方法. 这个代码跟我们前面的面向对象例子相似. 我们加了两行来指示验证平台什么时候停止, 这个叫"*raising an objection*".

理解 objection

验证平台的启动比停止要容易. 启动验证平台只需要调用 run_test(), 不过这个可能会生成上百个对象, 每个对象都运行在自己的线程上. 有些验证平台会在不同的端口上用好几个对象施加激励, 这些对象都终止了你才能停止测试用例.

你怎么知道什么时候停止测试用例呢?

一个粗暴的方案是简单的运行验证平台一段时间然后假设所有的对象在这段时间里都完成各自的工作了. 这个方案很差, 最好的情形已经是浪费了时间了, 最坏的情况是激励还没完成就运行超时了.

仔细考虑的话, 验证平台停止测试用例的问题跟办公室里什么时候关灯的问题是类似的. 留着灯的话费电, 不过人还在办公室就关灯他们会怒的. 这个问题的解决方案是人走完了就关灯. (很多办公室用运动传感器实现了这个方案)

UVM 也是这样解决退出问题的. 验证平台中只要有对象反对终止, 测试用例就会继续运行. UVM 中的对象可以对测试用例的终止"提出反对意见", 有一个对象反对终止, 测试用例就会继续运行.

作为开发者, 你需要在产生激励前提出 objection, 然后完成之后撤销 objection. 这个时候就相当于你通知 UVM, "终止测试用例我是没意见的." 所有的对象都撤销 objection 的时候 Test 就终止了.

我们说过 run_phase()中必须有 uvm_phase 类型的参数 phase. 我们用 phase 对象来提出和撤销 objection.

调用完 raise_objection()之后, 我们例化了验证平台对象, 在他们各自的线程里启动

coverage_h.execute()和scoreboard_h.execute()方法, 然后调用tester.execute(). tester_h.execute()返回的时候我们的Test就结束了, 所以我们调用了phase.drop_objection()方法. 这个时候因为没有其他提出的objection仿真就结束了.

编写 add_test 类

这个例子开始的时候, 我们的目标是不用重新编译来启动不同的测试用例. 我们创建的random_test类是第一项测试用例. 现在我们要创建add_test类.

add_test类跟random_test类基本一样, 除了不同的测试器对象:

```
class add_test extends uvm_test;
  `uvm_component_utils(add_test);

  virtual interface tinyalu_bfm bfm;

  function new (string name, uvm_component parent);
    super.new(name,parent);
    if(!uvm_config_db #(virtual interface tinyalu_bfm)::get(null, "*", "bfm", bfm))
      $fatal("Failed to get BFM");
  endfunction : new

  task run_phase(uvm_phase phase);
    add_tester add_tester_h;
    coverage coverage_h;
```

图 70. add_test 类

在第1行和第2行, 我们的add_test类继承了uvm_test然后我们用uvm_component_utils()宏将add_test注册到Factory.

random_test跟add_test的唯一区别是测试器对象. 新的Test对象是add_tester类而不是tester类的对象.

现在我们可以运行我们的两个测试用例了:

```
1 # vsim +UVM_TESTNAME=random_test top
2 ...
3 # UVM_INFO @ 0: reporter [RNTST] Running test random_test...
4 ...
5 # vsim +UVM_TESTNAME=add_test top
6 ...
7 # UVM_INFO @ 0: reporter [RNTST] Running test add_test...
8
```

图 71. 一次编译运行两项 test

总结

在本章我们第一次使用了UVM. 我们定义顶层类为uvm_test的子类, 然后用这些类在一次编译下运行多项测试用例.

本章中构建的 Test 例化了诸如 scoreboard 这样的验证平台对象然后在他们各自的线程里启动。这种在对象里例化对象来构建 Test 叫做创建验证平台层级。UVM 支持用 uvm_component 类来创建验证平台层级。下一章我们把验证平台组件类换成 uvm_component 的子类的时候会讨论这个类。

第十二章

UVM Component

验证平台的设计可以分成三个部分：结构，序列和数据。结构描述了验证平台的各个部分还有它们的配合方式。序列描述了向 DUT 发送的指令和指令的顺序，数据描述了指令中的激励数据。

下面的几章里我们要讨论如何用 UVM 来搭建验证平台的结构。UVM 描述了一个层次化的对象组成的验证平台，让我们能够在验证平台设计里持续的例化，启动和终止所有的对象。

`uvm_component` 类是验证平台结构的基础。比如，上一章中我们继承的 `uvm_test` 是一个 `uvm_component`。后面我们会用到的 `uvm_subscriber` 和 `uvm_driver` 都继承自 `uvm_component`。

要用 UVM 干活，你需要自如的定义和例化 UVM 组件。这里是步骤：

- 第 1 步：从 `uvm_component` 类或其子类继承定义你的组件。
- 第 2 步：用 `uvm_component_utils()` 宏注册这个类到 Factory。
- 第 3 步：提供 `uvm_component` 构造器。
- 第 4 步：在必要时重写 UVM 的 phase 方法。

在前面的章节，我们继承了 `uvm_test` 然后例化了三个验证平台的普通对象。在本章，我们要把测试器，覆盖器和比较器类转换成 `uvm_component`，然后用 UVM 的标准方法在 Test 里例化。

让我们用下面的步骤把 scoreboard 转换成 `uvm_component`：

第一步：从 `uvm_component` 类继承创建模块

继承 `uvm_component` 类来分享 UVM 开发者的成果：

```
class scoreboard extends uvm_component;
  uvm_component_utils(scoreboard);
  virtual tinyalu_bfm bfm;
```

图 72. 定义 scoreboard 为一个 UVM component

第二步：用 `uvm_component_utils()` 宏注册类到 Factory

`uvm_component_utils()` 宏把这个类注册到 Factory，现在我们可以用 Factory 来例化比较器了。这跟我们在 `random_test` 里用的宏是一样的。

第三步：提供最简 `uvm_component` 构造器

所有 `uvm_component` 类的 `new()` 方法都需要 `name` 和 `parent` 两个参数。这里是比较器类的构造器：

```

function new (string name, uvm_component parent);
    super.new(name, parent);
endfunction : new

```

图 73. uvm_component 构造器

random_test 的构造器提供了两个所需的参数, 并且把 BFM 句柄拷入局部变量. UVM 的 component 构造器可以做很多调用 super.new() 之外的事情, 不过这个 super.new() 的调用是必需的.

第四步: 有必要的话重写 UVM phase

写 UVM Test 的时候, 我创建了 run_phase() 方法来解释 UVM 会自动调用 run_phase() 来启动仿真. run_phase() 方法是 UVM phase 方法中的一个.

所有 UVM component 都继承了这些 phase 方法. UVM 在所有的 component 里以设定的顺序调用这些 phase 方法. 你可以在你的 component 里重写这些 phase 方法, 然后 UVM 会按顺序调用.

phase 方法不用强制重写, 不过在重写的时候, 你必须在第一步调用 super.<phase_name> 函数. 这能保证你使用 UVM 开发者的工作成果.

所有的 phase 方法都必须有一个 uvm_phase 类型名字是 phase 的参数. UVM phase 有很多, 不过本书中我们将只用其中 5 个而已.

UVM 按照以下顺序调用 phase 方法:

- function void build_phase--UVM 用这个方法自顶向下建立你的验证平台. 你要在这个方法里例化你的 uvm_component, 如果在别的方法里例化的话, 你会得到 UVM 的致命报错.
- function void connect_phase--connect phase 把各个模块连接到一起. 我们会在后面的章节学习连接.
- function void end_of_elaboration_phase--UVM 在所有 component 都就位并连接好以后调用这个方法, 需要在 UVM 层次设定完之后再对验证平台进行调整的话可以用这个 phase.
- task run_phase--UVM 会在各自的线程调用这个 task. 验证平台中的 run_phase() 是 "同时" 运行的, 所以你不知道那个会最先启动.
- function void report_phase--这个 phase 会在最后一个 objection 撤销后运行, 然后测试用例就结束了. 你可以用来报告结果.

重载比较器类中的方法

scoreboard 类重载了 build_phase() 和 run_phase() 两个类方法.

这里是 scoreboard build_phase() 方法:

```

function void build_phase(uvm_phase phase);
    if(!uvm_config_db #(virtual tinyalu_bfm)::get(null, "*", "bfm", bfm))
        $fatal("Failed to get BFM");
endfunction : build_phase

```

图 74. scoreboard 的 build_phase() 方法

所有的 UVM phase 都有同样的 phase 参数. 如果你不提供的话会得到编译报错.

在之前的验证平台版本中, `uvm_test` 对象在创建验证对象时传递了 BFM. 这是不好的设计, 因为这会强迫用比较器的人仅仅为了把 BFM 传给其他对象就接收 BFM. 每个类都自给自足更好一些.

在 `scoreboard` 类中, 我们在 `build_phase()` 方法里自行从 `uvm_config_db` 里得到 BFM. 下面, 我们来重载 `scoreboard` 的 `run_phase()` 方法:

```
task run_phase(uvm_phase phase);
    shortint predicted_result;
    forever begin : self_checker
        @(posedge bfm.done)
        case (bfm.op_set)
            add_op: predicted_result = bfm.A + bfm.B;
            and_op: predicted_result = bfm.A & bfm.B;
            xor_op: predicted_result = bfm.A ^ bfm.B;
            mul_op: predicted_result = bfm.A * bfm.B;
        endcase // case (op_set)
        if ((bfm.op_set != no_op) && (bfm.op_set != rst_op))
            if (predicted_result != bfm.result)
                $error ("FAILED: A: %0h B: %0h op: %s result: %0h",
                        bfm.A, bfm.B, bfm.op_set.name(), bfm.result);
    end : self_checker
endtask : run_phase
```

图 75. `scoreboard` 的 `run_phase()` 方法

其他 UVM phase 方法都是函数, 而 `run_phase()` 是 task. 这意味着 `run_phase()` 是唯一可以消耗仿真时间来等待时钟, 插入延时的. 其他的方法都需要立即返回.

这样普通比较器到 `uvm_component` 比较器的转换就完成了. 其他的三个部分的转换也是类似的, 详情参见 www.uvmprimer.com. 现在我们要用它们来搭建验证平台了.

用 `build_phase()` 方法来建立验证平台

我们已经见识了怎么定义一个 `uvm_component`. 现在回到 `random_test`, 如果 `random_test` 继承自 `uvm_test` 而 `uvm_test` 继承自 `uvm_component`, 那我们不是要用 `build_phase()` 来例化三个验证平台 component 吗?

是这样的:

```

16   class random_test extends uvm_test;
17     `uvm_component_utils(random_test);
18
19     random_tester tester_h;
20     coverage      coverage_h;
21     scoreboard    scoreboard_h;
22
23     function void build_phase(uvm_phase phase);
24       tester_h      = new("tester_h", this);
25       coverage_h    = new("coverage_h",   this);
26       scoreboard_h = new("scoreboard_h",  this);
27     endfunction : build_phase
28
29     function new (string name, uvm_component parent);
30       super.new(name,parent);
31     endfunction : new
32
33   endclass

```

图 76. 例化验证平台组件

这个小类不是挺灵活吗？它只是重载了 build phase 然后例化了三个验证平台组件，其他的活就不用操心了，各个模块的 run_phase() 方法会去做的。

事实上 random_test 连 run_phase() 都没有，它建立完这个测试用例就完工了。

add_test 继承了 random_test 类，然后把 tester_h 数据成员重载(数据成员可以直接重载吗？变量名不变，直接把类型改了?)为另外的类型。它继承了 build_phase() 方法，所以我们只用写一次 build_phase()：

```

1  class add_test extends random_test;
2   `uvm_component_utils(add_test);
3
4   add_tester tester_h;
5
6   function new (string name, uvm_component parent);
7     super.new(name,parent);
8   endfunction : new
9
10  endclass

```

图 77. 使用派生创建 add test

总结

在本章，我们学习了 UVM 的主力类：uvm_component。我们按照 uvm_component 的创建步骤操作并定义了多个组件类。

每次定义 uvm_component 类我们都用 uvm_component_utils() 宏来“注册这个类到 Factory”。

之后我们没有用 Factory 来创建类，这是为啥？

下一章，我们会引入一个另外的层级：uvm_env。然后我们会看到 uvm_test 和 uvm_env 怎么一起用 Factory 动态改变验证平台结构来进行不同的 Test。

第十三章

UVM Env

前面两章我们学习了用 Factory 创建 uvm_test 顶层对象然后来例化 uvm_component. 我们还了解了 UVM 可以自动调用组件的 phase 方法, 组件会在各自的线程里自动启动 run_phase()方法.

我们在之前直接把 component 例化到 test 里. 这种固定的形式验证平台创建容易理解, 但是难以重用. 实际上, 这引发了复杂性编程和适应性编程的讨论.

复杂性编程 VS 适应性编程

这是一道送分题:"你来选的话, 会选择写随着规模变大更容易改动的验证平台, 还是更难改动的验证平台?"

答案是显然的. 我们希望在不破坏验证平台的情况下增加功能和测试用例, 并希望这些组件可以用在下个验证平台上.

人们经常会讨论可重用编程是"把功能写对" 还是"不要把自己逼到绝境". 这些问题是好问题, 不过还没有完全表达出这个意思, 真正的问题是你要写复杂的代码还是适应性强的代码.

复杂代码就是你不用想太多就写出的代码, 不过需要长时间配置, 而且会变得越来越难改. 我见过的最坏的情况来自于一个工程团队, 他们创建新 Test 的方法是把原先的验证平台代码全部拷到另外一个目录, 然后改很多文件把这个验证平台改成另外的 Test.

这是一个典型的复杂编程例子, 这个验证平台明显会变得越来越难以管理. 如果这个验证平台出了 bug 的话, 维护者需要在很多路径里对同一个文件做改动. 改一个文件然后拷到所有其他的目录里也可以, 不过如果里面的一个文件的某个地方被另外的人改了, 这个验证平台就被破坏了.

每次你一改动验证平台就崩了, 这好惨的, 然后因为需要把改动复制到所有的目录, 这个是完全重用不了的.

适应性编程就是所有你编写的内容都变成可以使用的资源, 扩展验证平台的时候也可以使用. 适应性的验证平台会越来越强大, 因为每个工具都是可以在将来使用的.

我们的 TinyALU 验证平台一开始把所有 Verilog 代码放在一个文件里的时候是复杂的. 我们根据一下的三个原则把它改造的更灵活:

- 创建高内聚的类来构造方案.
- 尽可能避免 Hardcoding.
- 不关注实现, 只使用接口.

因为遵守了这些原则, 我们现在的验证平台得以在不重新编译的情况下运行不同的测试用例. 我们在命令行指定测试用例名, factory 就会根据这个测试用例名称来创建对应的 uvm_test 变量.

然而, 我们目前的 random_test 和 add_test 对象违背了上面的第一个原则:

- 它们用例化组件的形式创建验证平台.

- 它们用声明测试器的方式改动验证平台.

在本章, 我们会把验证平台的结构跟功能进行分离. 我们要添加另外一个用于搭建验证平台的类 `uvm_env`. 然后我们会看到测试用例怎么跟这个类通过 `factory` 来通信.

构建适应性代码

目前的 TinyALU 验证平台通过定义 `random_tester` 和 `add_tester` 类来得到随机测试用例和加法测试用例. 我们就用这两个类来理解复杂编程和适应性编程的区别以及怎样进行适应性编程.

我们先来看看这两个类是做什么的:

- `random_tester`: 发送 1000 个随机指令和 1000 组带约束随机操作数到 DUT.
- `add_tester`: 发送 1000 个加法指令和 1000 组带约束随机操作数到 DUT.

这里有三种实现这两个类的方式, 从最粗暴到最灵活.

最粗暴的方案是这样的:

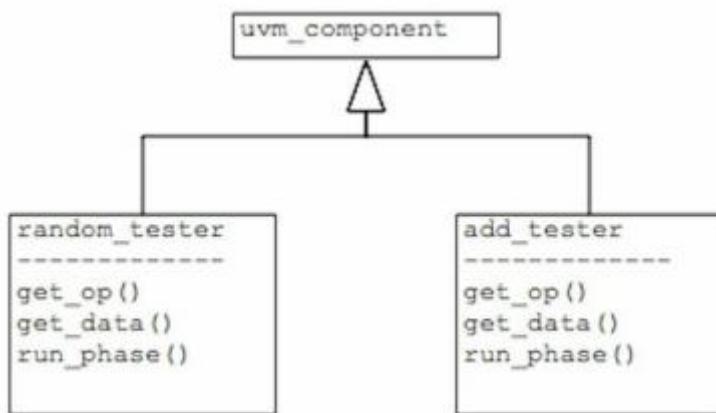


图 78. tester 问题的粗暴方案

在这个方案里我们各从 `uvm_component` 类里继承了一次. 每个类都有提供指令的 `get_op()` 方法, 提供操作数的 `get_data()` 方法和在 DUT 上运行 1000 个指令的 `run_phase()` 方法.

是什么让这个方案变的难搞, 然后还因此倾向于越变越难以维护呢? 是我们在两个类里拷贝了 `run_phase()`. 如果我们发现 `run_phase()` 里搞错了, 或者想做下改动, 就需要进行多处改动. 我们可以用 `include` 文件来解决这个问题, 不过还有更好的办法.

下面的是适应性更好的方案:

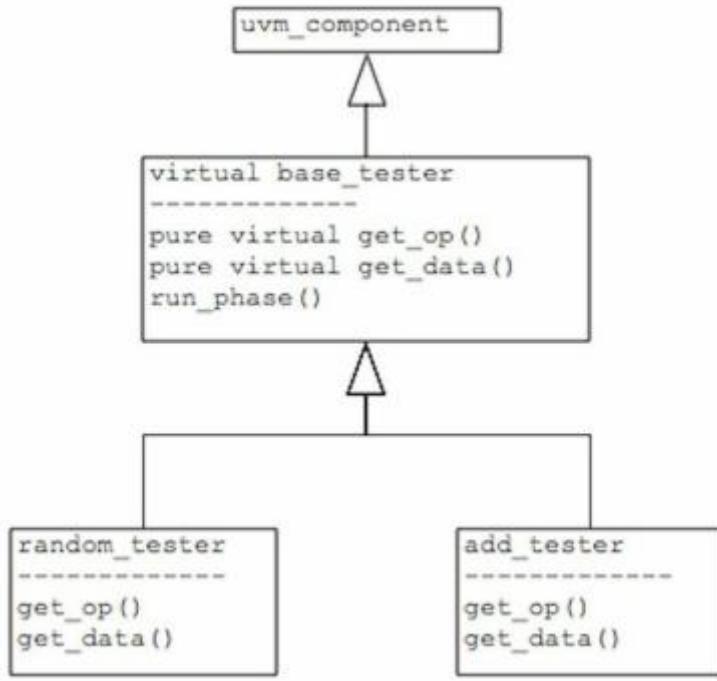


图 79. 使用 virtual class

在这个方案中我们创建了 `base_tester` 类, 它提供了 `run_phase()` 方法. 这个 `run_phase()` 方法用 `get_op()` 和 `get_data()` 方法进行 1000 次指令. `base_tester` 是一个虚类, 不能实例化, 不过可以继承 `base_tester` 来创建其他 Test 类.

`run_phase()` 方法假定所有的 `base_tester` 子类都重载 `get_op()` 和 `get_data()` 方法, 用 `pure virtual` 关键字来强制 `random_tester` 和 `add_tester` 来完成.

这是个更好的方案, 不过还是有改进的余地. 我们的目标是在验证平台壮大的时候更易于改动, 因为我们给程序员更多的工具了. 我们已经通过编写 `base_tester` 创造了一个工具, 这个类可以发送 1000 次 transaction 给 DUT. 现在我们用派生类来创造其他的工具.

`random_tester` 可以生成随机指令和操作数. `add_tester` 可以生成随机操作数的加法指令. 我们发现 `random_tester` 可以成为其他需要随机指令和操作数的类的资源. 我们用这个来创建更简单的 `add_tester` 吧.

下面的是适应性最好的方案:

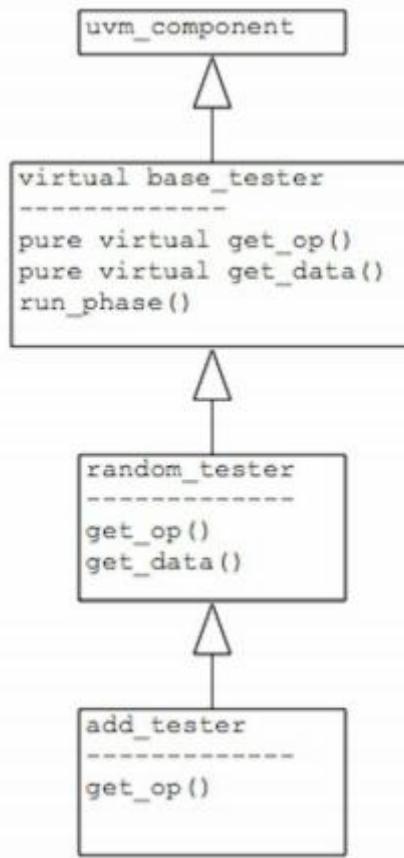


图 80. 一种灵活的架构

我们现在把 `random_tester` 变成了其他 Test 可以用的资源. 可以想见再添加 `mul_tester` 和 `xor_tester` 类到这个架构里是很容易的.

最后我们的 `add_tester` 代码是很简洁的:

```

1  class add_tester extends random_tester;
2      `uvm_component_utils(add_tester)
3
4      function operation_t get_op();
5          bit [2:0] op_choice;
6          return add_op;
7          endfunction : get_op
8
9      function new (string name, uvm_component parent);
10         super.new(name, parent);
11         endfunction : new
12
13     endclass : add_tester

```

图 81. 站在巨人肩上的 `add tester`

我们在写的 `add_tester` 里创建了一个简单的返回 `add_op` 值的 `get_op()` 方法. 我们从继承线上得到其他的功能, 这是个很长的 Test 继承谱线呢.

结构与激励分离

我们的 `base_tester` 是架构里很好的一个模块, 它为可以发送 1000 个 transaction 到验证平台的 `test object` 提供了基础. 我们已经用这个为基础, 创建了 `random_tester` 和 `add_tester` 类. 现在问题来了:"我们怎么用这些类呢?"

一个简单的方案是跟我在之前的章节里做的一样, 创建一个 `uvm_test` 类然后声明各个类型的 `tester` 变量. 这个方案可行, 但是造成了粗暴代码.

这个代码是粗暴的, 因为它用 Hardcoding 写 `random_tester` 验证平台的结构. 每个从 `random_test(z: 这里写错了呢)` 继承的 `Test` 都包含 `random_tester`, `coverage` 和 `scoreboard` 对象. 如果我们要创建一个有不同验证平台对象配置的验证平台呢? 这样就麻烦了.

这个验证平台的棘手之处在于违背了每个类只做"一件事"的原则. 这里的 `random_test` 类族指定了 `tester_h` 对象的类型还有验证平台的结构. 我们要把这两个功能分到两个类里.

UVM 提供了 UVM 环境(`uvm_env`)来解决这个问题. `uvm_env` 类派生自 `uvm_component` 类并提供了配置结构性代码的标准类. `uvm_env` 通常只包含 `build_phase()` 和 `connect_phase()` 方法.

我们用 `uvm_env` 来操作验证平台的结构, 然后用 `test` 来指定完成这个结构的变量. 第一步是在 `env` 类里例化我们的验证平台组件, 然后在 `random_test` 类里例化 `env` 类.

这里是我们的用环境对象之前和之后的 UVM 层次图:

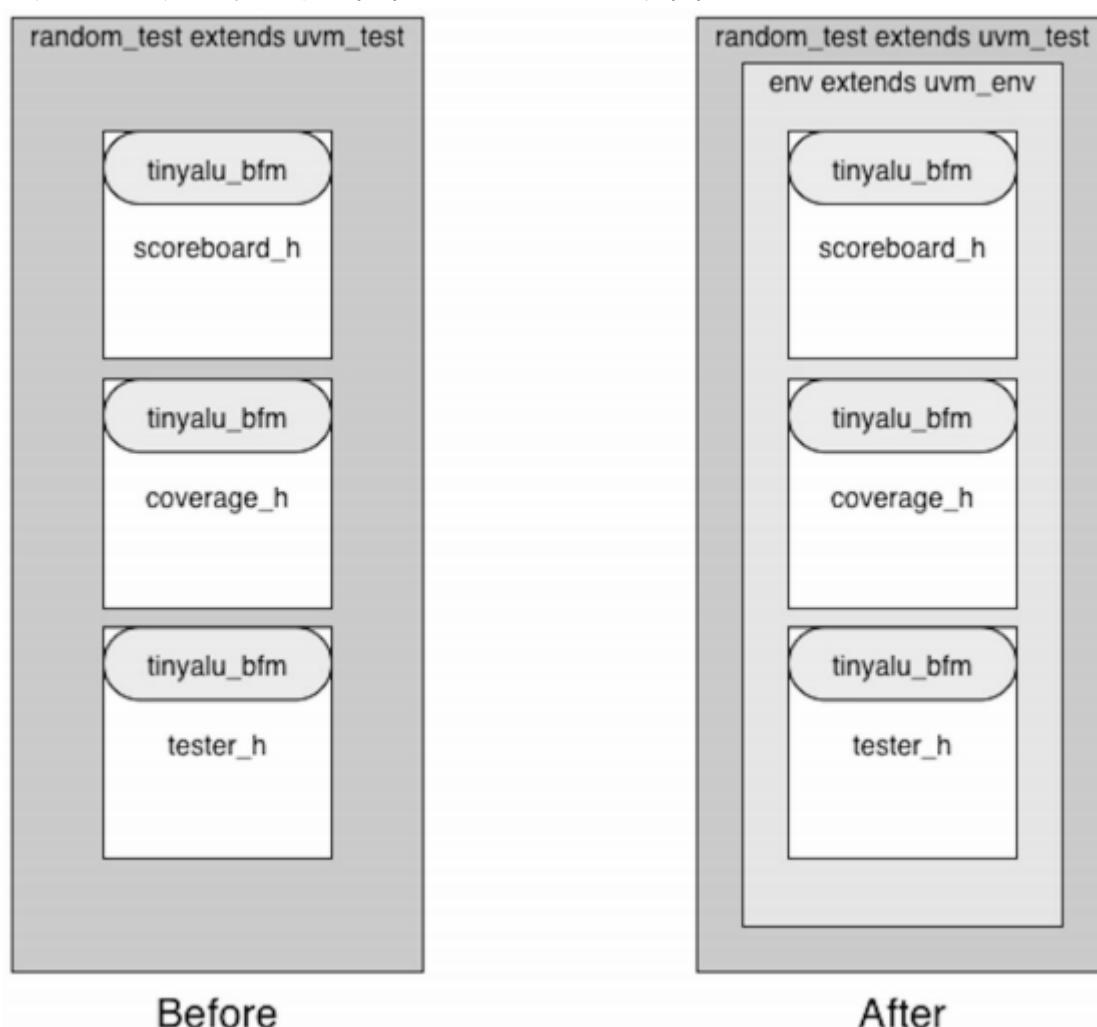


图 82. 增加一层结构

上面的结构从 `uvm_test` 中分离出了我的验证平台结构, 所有我们可以在多个 `test` 或者甚

至其他验证平台里用一样的结构了.

env 类

env 类定义了验证平台的结构. 它例化了验证平台中的对象, 后面的版本会把它们连接起来.

这里是 env 类的定义:

```
class env extends uvm_env;
  `uvm_component_utils(env);

  base_tester  tester_h;
  coverage     coverage_h;
  scoreboard   scoreboard_h;

  function void build_phase(uvm_phase phase);
    tester_h      = base_tester::type_id::create("tester_h",this);
    coverage_h    = coverage::type_id::create ("coverage_h",this);
    scoreboard_h = scoreboard::type_id::create("scoreboard_h",this);
  endfunction : build_phase

  function new (string name, uvm_component parent);
    super.new(name,parent);
  endfunction : new

endclass
```

图 83. env 中例化验证平台组件

env 类展现了 OOP 的精华, 把问题分解到多个功能单一的类, 你的代码会越来越简单. 一个好的验证平台会有很多简单的类而不是少量的复杂的类. 这也简化了调试.

env 类定义了 tester_h, coverage_h 和 scoreboard_h 变量来操作三个 uvm_component, 在 build phase 例化这些组件.

另一个角度来讲, build_phase() 比上一章中要复杂. 之前的 build_phase() 调用 new() 方法来创建组件, 这里的调用的复杂的静态方法, 这个是什么呢?

用 UVM Factory 来创建 UVM 组件

在 Factory 模式这一章, 我们学习了不用 hardcoding 的构造器动态的创建对象.

我们的简化动物 Factory 案例有严重的重用限制, 需要添加新动物需要改动代码. 而且, 你需要从结果转换得到需要的对象类型.

UVM Factory 更高明, 它解决了这两个问题:

- 用 uvm_component_utils() 和 uvm_object_utils() 宏来添加新类到 factory.
- Factory 不用我们去转化就返回正确的对象类型.

就像神秘的魔力, UVM Factory 通过精确的"咒语"来使用. 一些"咒语"跟对象的类型相关, 另外一些是不变的:

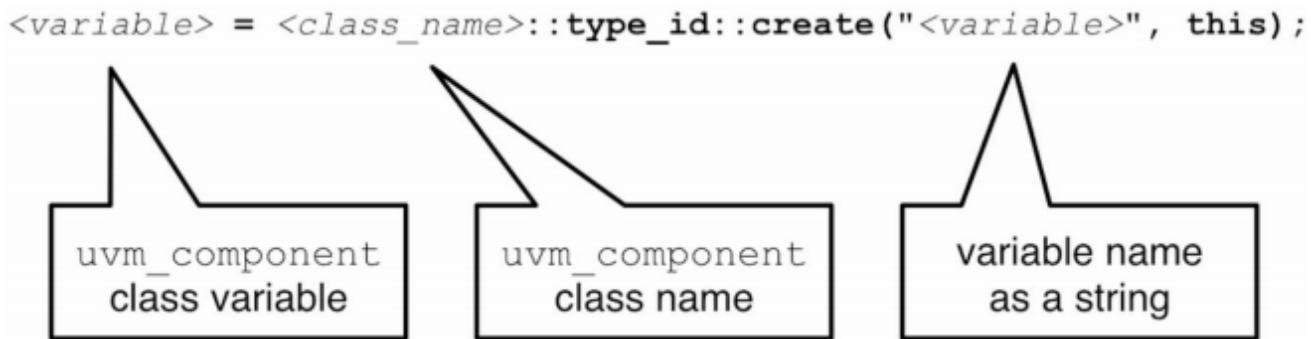


图 84. UVM Factory 使用原型

Factory 用使用我们之前讨论过的静态成员/方法来创建 uvm_component. 这有很多好处:

- 不用转换输出的类型; 这是自动完成的.
- 编译器可以捕捉类没定义或者类名的拼写错误.
- 编译器可以捕捉 uvm_component_utils() 宏的遗漏.

编译器对小错误的捕获可以节省时间, 特别是你有一个大型的验证平台需要长时间初始化的时候.

在我们的例子里, 我们用 Factory 创建了三个验证组件. 这里是 build phase:

```

function void build_phase(uvm_phase phase);
    tester_h      = base_tester::type_id::create("tester_h",this);
    coverage_h    = coverage::type_id::create ("coverage_h",this);
    scoreboard_h = scoreboard::type_id::create("scoreboard_h",this);
endfunction : build_phase

```

图 85. build phase 里创建 base tester

上面的代码用 Factory 格式创建三个对象. 然而, 我们发现了一个奇怪的事情. tester_h 这行创建了 base_tester 类的对象. 我们在类图上看到 base_tester 类是个抽象类. 你不能创建 base_tester 对象, 只能从中派生出子类. 这个代码是怎么工作的呢?

原来 env 类把 base_tester 变量作为一个占位符. 这个代码推断 Factory 会返回 base_tester 的子类, 但是不知道是哪一个. 因此, 调用 build_phase() 之前需要有另外的代码在 Factory 里重载 base_tester 类. 在本例中, Factory 的重载在例化这个环境的 Test 类里完成.

Factory 重载

第一次使用 UVM Test 的时候, 我们从 random_test 派生出 add_test. add_test 用 add_tester 替换了 tester.

我们复制了 random_test 然后把 tester 对象替换成 add_tester 对象, 这样创建 add_test 是可行的, 不过违犯了如下的规则:

"你复制代码来做改动的时候, 你就做错了."

复制代码会造成棘手代码. 要是有人在某天改了其中一个忘了改掉另外一个的话, 就会出 bug.

UVM Factory 的重载特性解决了这个问题.

由于 add_tester 派生自 base_tester 类, 我们可以在需要用 base_tester 用 add_tester 替代. 我们可以让 UVM Factory 在生成 base_tester 的时候生成 add_tester, 这个代码是可以编译运行的. 这就叫做 Factory 重载.

我们用另外的例化来重载:

```
<base_class_name>::type_id::set_type_override(<child_class_name>::get_type());
```

图 87. 重载原型

静态方法 set_type_override()告诉 Factory, "在看到 base_class_name 的请求时, 返回 child_class_name 类型的对象."

这个特性允许我们分离出验证平台的结构(由 uvm_env 控制)和激励的选择(由 uvm_test 控制).

这里是 random_test:

```
class random_test extends uvm_test;
  `uvm_component_utils(random_test);

  env      env_h;

  function void build_phase(uvm_phase phase);
    base_tester::type_id::set_type_override(random_tester::get_type());
    env_h = env::type_id::create("env_h",this);
  endfunction : build_phase

  function new (string name, uvm_component parent);
    super.new(name,parent);
  endfunction : new

endclass
```

图 87. 重载 tester

这是编写 Test 的适应性方式. 所有的 Test 根据需要的激励重写 base_tester 然后用 env 类创建对象并启动. 现在 Test 类做好了一个事情, env 类做好了另外一件事情.

总结

我们在本章通过继承 uvm_env 增加了另外一个 UVM 验证平台层次. 这个新层次分离了 Test 功能(选择激励)和环境功能(提供结构).

我们用定义 base_tester 类族然后用 Factory 和环境类来为每个测试用例选择测试器的方式来创建适应性代码. Factory 可以让我们在环境里选择激励.

在这个验证平台里, 所有的组件都各自访问 BFM. 这在一个简单的验证平台里是可行的, 但是复杂验证平台的话就不行了. 如果验证平台的对象可以互相通信, 我们就可以更容易写出复杂的验证平台.

UVM 提供了对象间通信, 让我们得以连接对象用以创建新的行为.

我们将在后面四章探讨 UVM 通信, 不过我们还是先来消化和讨论 OOP 程序员的思想吧.

第十四章

一个新思路

我在 Sun Microsystems 工作的时候公司引入了 Java, 就这样我第一次接触到了 OOP. 我还记得我第一次读 Java 的文献, 然后尝试一些简单的程序, 不过真的没怎搞明白.

我还记得当时坐在工位上想, "我需要一个思维转换." 遗憾的是, 自发的思维转换几乎是不可能的. 你必须坚持学习然后等待顿悟, 然后你会对这个世界有全新的视角.

就是说, 我希望在本章可以给你一些面向对象的思路, 然后让你更快领悟.

对象和脚本

大多数人都是从写代码执行算法和流程的角度来学习编程的. 我们学到的是把程序看成一系列的步骤, 然后我们会认为解决问题就是调用一系列的命令和子程序把数据从一种形式转换成另外一种. 这叫做过程式编程.

过程式编程在脚本界(PERL, Python 等)和 RTL 开发界是主流. 我们等待时钟上升沿, 然后执行一系列的数据转换并把结构存储到一系列的寄存器里.

TinyALU 的 VHDL 代码展示了程序思维的操作单周期指令的例子:

```
begin
    if (clk'event and clk = '1') then
        -- Synchronous Reset
        if (reset_n = '0') then
            -- Reset Actions
            result_aax      <= "0000000000000000";
        else
            if START = '1' then
                case op is
                    when "001"  =>
                        result_aax <= ("00000000" & A) +
                                    ("00000000" & B);
                    when "010"  =>
                        result_aax <= unsigned(std_logic_vector("00000000" & A) and
                                    std_logic_vector("00000000" & B));
                end case;
            end if;
        end if;
    end if;
end process;
```

图 88. 过程代码

上面的代码等待时钟上升沿, 对指令进行译码然后执行正确的指令. 过程式编程程序员总是在问:"我下一步做什么呢?"

而一个 OOP 程序员有部分的过程式思维, 不过这不是他们处理问题的主要方式. 他们问的是:"我要怎么创建并连接对象来解决这个问题呢?"

这个是编程的新思路, 不过对 RTL 开发者来说也不是完全陌生的. 我们一直在把对象组合起来解决问题, 只是我们把他们叫做 VHDL 里的组件和 Verilog 里的 module. 例如, 这里是 TinyALU 的顶层中的代码片段:

```

add_and_xor : single_cycle
port map (
    A          => A,
    B          => B,
    clk        => clk,
    op         => op,
    reset_n    => reset_n,
    start      => start_single,
    done_aax   => done_aax,
    result_aax => result_aax
);
mult      : three_cycle
port map (
    A          => A,
    B          => B,
    clk        => clk,
    reset_n    => reset_n,
    start      => start_mult,
    done_mult  => done_mult,
    result_mult => result_mult
);

```

图 89. VHDL 中的例化

这里我们有两个组件, 一个操作单周期指令, 另外一个操作 3 周期指令. 我们一样在用 OOP 思想来创建 TinyALU 了---"我要怎么组合模块来解决问题呢?"

这个例子里我们用 TinyALU 的顶层信号把两个模块连接到一起, 这引出了我们尚未讨论过的 OOP 的话题.

我要怎么在 UVM 里把对象连接到一起呢?

我们的 TinyALU 验证平台目前有 `tester_h`, `coverage_h` 和 `scoreboard_h` 三个对象. 不过, 这些对象不能相互交流. 实际上, 它们是无视对方的, 只是通过 BFM 的拷贝跟 DUT 交流.

如果我们要用 UVM 设计更复杂的验证平台的话, 这是需要改变的.

两种对象通信的类型

在后面四章, 我们将讨论两种对象通信模型:

- 单线程通信---这个情景中, 一个对象在一个线程中运行, 简单的调用另外一个对象的方法.
- 双线程通信---这个情景中, 两个对象在不同线程中运行. 我们需要两个对象彼此通信并进行线程间的时序合作.

在面向对象的设计思想里, UVM 给我们提供了解决这些问题的类.

我们先从讨论单线程通信, OOP 设计和 `uvm_analysys_port` 类开始吧.

第十五章

与多个对象通信

假设我们在与一个 UVM/OOP 类通话, 然后任务是这样的: 写一个程序, 掷 2 个 6 面骰子 20 次, 然后打印如下的信息:

- 平均点数
- 点数频率直方图
- 一个覆盖率统计展示我们是否得到了 2-12 的所有点数值

我们的目的是用 OOP 来写这个程序. 我们希望编写的对象功能单一, 易于重用. 我们要用 UVM 来构造这个程序.

让我们回想一下 OOP 思想要解决的问题:

怎么创建并连接对象来解决这个问题?

我们需要编写类来解决这个问题. 我们先从创建三个 uvm_component 来控制这三个报告. 这里我们只关注 average 类, 其他的详见 www.uvmprimer.com.

average 类是一个 uvm_component, 它用 write() 方法来收集数据, 然后在仿真结束的时候用 report_phase() 打印结果:

```
1 class average extends uvm_component;
2   `uvm_component_utils(average);
3
4   protected real dice_total;
5   protected real count;
6
7   function new(string name, uvm_component parent = null);
8     super.new(name,parent);
9     dice_total = 0.0;
10    count = 0.0;
11  endfunction : new
```

图 90. average 类数据成员和构造器

average 有 count 和 dice_total 两个数据成员. 我们将这两个成员设为 protected, 因为它们不应该被直接访问. 现在我们要集合数据算出平均值了:

```
13   function void write(int t);
14     dice_total = dice_total + t;
15     count++;
16   endfunction : write
17
18   function void report_phase(uvm_phase phase);
19     $display ("DICE AVERAGE: %2.1f",dice_total/count);
20   endfunction : report_phase
21 endclass : average
```

图 91. 计算平均值并报告

我们创建了 write() 方法收集骰子的数值, 把它加到 dice_total 里, 同时增加 count.

UVM 在仿真完成时调用 `report_phase()`方法, 我们用这个来打印出平均值.
我们也完成了 `histogram` 类和 `coverage` 类, 它们跟 `average` 类似, 都有收集数据的 `write()`方法和输出数据的 `report_phase()`.

根据 UVM 的良好实践, 我们创造了派生自 `uvm_test` 的 `dice_test` 类, 然后依靠 UVM 来例化测试用例并运行.

要记住, 我们在寻找把对象连接到一起解决问题的方法. 所以我们在 `dice_test` 类的上方声明变量然后例化这些对象:

```
1 class dice_test extends uvm_test;
2   `uvm_component_utils(dice_test);
3
4   dice_roller dice_roller_h;
5   coverage coverage_h;
6   histogram histogram_h;
7   average average_h;
8
9   function void build_phase(uvm_phase phase);
10    coverage_h = new("coverage_h", this);
11    histogram_h = new("histogram_h", this);
12    average_h = new("average_h", this);
13    dice_roller_h = new("dice_roller_h", this);
14 endfunction : build_phase
```

图 92. 在 dice test 类里声明例化组件

目前为止还不错. 我们定义了解决掷骰子的问题的四个类, 在 `build_phase()`里例化它们. 这都是根据 UVM 和 OOP 的良好设计原则来的. 现在我们只需要运行测试用例. 这里是 `run_phase()`:

```
16 task run_phase(uvm_phase phase);
17   int the_roll;
18   phase.raise_objection(this);
19   repeat (20) begin
20     the_roll = dice_roller_h.two_dice();
21     coverage_h.write(the_roll);
22     histogram_h.write(the_roll);
23     average_h.write(the_roll);
24   end
25   phase.drop_objection(this);
26 endtask : run_phase
```

图 93. 运行 dice test

这个程序看起来很简单, 我们循环了 20 次, 每次调用 `dice_roller` 中的 `two_dice()`方法, 然后把结果送入 4 个报告对象. 结果看起来是这样的:

```

59  # DICE AVERAGE: 6.5
60  #
61  # COVERAGE: 82%
62  #
63  # 2: ##
64  # 3:
65  # 4: ###
66  # 5: ##
67  # 6: ##
68  # 7: ####
69  # 8: ###
70  # 9: ##
71  # 10: #
72  # 11:
73  # 12: #
74  #

```

图 94. dice roller 结果

我们的任务是简单的, 我们自豪的提交了我们的源码和结果. 分数下来的时候, 我们得到的是 B. B? 我们震惊了, 为什么不是 A?

我们一般是不强求分数的, 不过这是原则问题. 我们闯进教授的办公室, 挥舞着代码.

"我觉得我们应该得 A." 我们说.

"显然, 我不这么认为." 傲慢的教授说道.

"我们为什么不能拿 A?"

教授拿过我们的代码, 抽出红笔然后做了下面的事情:

```

17  task run_phase(uvm_phase phase);
18    int the_roll;
19    super.run_phase(phase);
20    phase.raise_object_in_mixin();
21    repeat (20) begin
22      the_roll = dice_roller_h.two_dice(),
23      coverage_h.write(the_roll);
24      histogram_h.write(the_roll);
25      average_h.write(the_roll);
26    end
27    phase.drop_object_in_mixin();
28  endtask : run_phase

```

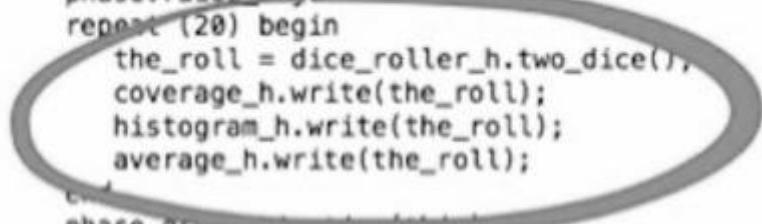


图 95. 不好的代码, 重来吧

教授说, "这是脚本而已, 甚至算不上好脚本. 如果你要这么写, 原来就不应该浪费时间创建类."

然后我们怒了.

"我们创建类只是因为你叫我们这么做."

"是的."

"那我们应该怎么做?"

"你们应该写出完成单一任务的类, 然后在顶层把它们连接到一起. 记住这句话: '我要怎么把这些对象连接到一起来解决这个问题?'"

我们面面相觑.

教授继续说, "回去研究一下观察者设计模式和 UVM 的 Analysis Port, 重新写你们的代码,

然后我再给你们 A."

所以现在是时候开始学观察者模式了.

观察者设计模式

观察者设计模式可以命名为推特设计模式.

在推特上, 你写了一个推特然后向 follow 你的人发布. 你不需要知道谁 follow 了你, 也不知道你的粉丝会对这条推怎么做. 甚至你没有粉丝也可以发推, 一写然后点回车就是了.

观察者设计模式也是这样的, 在观察者设计模式里, 一个对象创建数据然后进行共享, 不用管多少粉丝(观察者). 无论有多少观察者, 都是可以 follow 这个发送对象, 他们得到一样的数据拷贝, 然后他们可以用来做任何事情.

这个怎么应用于我们的骰子问题呢? 我们有 `dice_roller_h` 对象, 它创建其他对象需要的数据. `dice_roller_h` 是被观察的对象, `coverage_h`, `histogram_h` 和 `average_h` 对象是观察者. 不幸的是, UVM 把这些对象叫做"订阅者"而不是"观察者".(好多同义词啊...)

UVM 和观察者设计模式

UVM 为简化观察者模式的实现提供了两个类:

- `uvm_analysis_port`--给任意的对象提供复发送数据到订阅者(观察者)的途径
- `uvm_subscriber`--派生自 `uvm_component`, 可以让组件订阅 `uvm_analysis_port`

`uvm_analysis_port`

`uvm_analysis_port` 得以让我们向任意数量的订阅者发送数据. 我们用一个三步的流程来用这个 port:

- 声明 analysis port 变量, 然后定义待传输数据的类型.
- 在 build phase 里例化 analysis port.
- 用 `write()`方法写数据到 analysis port, 一旦写入数据到 port, 订阅者就全都收到了.

`uvm_analysis_port` 有另外一个方法: `connect()`, 用来连接订阅者到 port. `connect()` 有一个 `analysis_export` 对象参数.

`uvm_subscriber`

派生自 `uvm_component` 的 `uvm_subscriber` 类可以让我们连接到 analysis port. 这个类给了你好处也要从你这里得到回报:

- 这个类了你 `analysis_export` 对象, 你是从派生类里得到的.
- 这个类需要你编写控制接收数据的 `write()`方法.

我们来用这些类重写掷骰子对象吧.

实现我们的订阅者

我们有 average, coverage 和 histogram 三个 uvm_subscriber. 这里是写成 uvm_subscriber 的 average 类:

```
1  class average extends uvm_subscriber #(int);
2    `uvm_component_utils(average);
3
4    real dice_total;
5    real count;
```

图 96. 派生 uvm_subscriber 类

跟之前的代码唯一的区别是我们现在从 uvm_subscriber 而不是 uvm_component 继承, uvm_subscriber 类是参数化的, 要求我们提供操作数据的类型. 这里是 int.

我们的 write()方法跟之前是一样的, 这个方法必须要有一个跟继承uvm_subscriber 的时候的参数类型一样的参数 t:

```
13   function void write(int t);
14     dice_total = dice_total + t;
15     count++;
16   endfunction : write
```

图 97. write 方法

我们在 histogram 和 coverage 类里也做了一样的改动, 把它们都转换成了 uvm_subscriber.

现在我们需要修改 dice_roller 并把这些对象连接起来.

在 dice_roller 里用 uvm_analysis_port

之前我们讨论观察者模式的时候, 说这个跟推特很像. 我们有一个发布数据的对象, 还有其他的对象来得到数据的拷贝. 我们也了解了 uvm_analysis_port 可以让我们的类实现观察者模式.

我们通过在 dice_roller 类里例化 uvm_analysis_port 来使用它:

```
1  class dice_roller extends uvm_component;
2    `uvm_component_utils(dice_roller);
3
4    uvm_analysis_port #(int) roll_ap;
5
6    function void build_phase (uvm_phase phase);
7      roll_ap = new("roll_ap",this);
8
9    endfunction : build_phase
```

图 98. 例化 UVM 的 analysis port

我们声明了操作 analysis port 的变量 roll_ap. 这个声明显示了 analysis port 发送的是 int 变量. 定义完之后, 我们在 build phase 里例化 roll_ap.

我们现在生成掷骰子然后把结果写到没定义数量的对象上, 像这样:

```
19      task run_phase(uvm_phase phase);
20          int the_roll;
21          phase.raise_objection(this);
22          void'(randomize());
23          repeat (20) begin
24              void'(randomize());
25              the_roll = die1 + die2;
26              roll_ap.write(the_roll);
27          end
28          phase.drop_objection(this);
29      endtask : run_phase
```

图 99. 使用 analysis port 来发送数据

我们调用 analysis port 的 write()方法来发送数据, 然后 analysis port 会调用所有订阅者的 write()方法来传递数据. 这是 analysis port 的精华---我们不用关心谁在用我们的数据, 或是怎么用我们的数据. 这是我们的设计的另外一个部分操纵的.

connect_phase()方法

我们说过观察者模式可以叫做推特模式, 对象 follow(订阅到)其他对象来获得它们的动态. 我们用 analysis port 创建了数据源, 同时创建了几个 UVM 订阅者. 现在我们需要让订阅者 follow 数据源.

这个过程叫做对象连接, UVM 提供了一个 phase 方法来做这个工作. UVM 在所有的 component 里完成所有 build_phase()之后调用 connect_phase().

UVM 自顶向下调用 build_phase()方法, 首先调用顶层 Test 的, 然后在顶层创建的对象中调用 build_phase(). 最后调用到为所有组件运行 build_phase()的层次.

一旦 UVM 完成所有 build_phase()的调用, 就开始自底向上调用 connect_phase()直至所有的对象都调用了 connect_phase(). 我们用 connect_phase()来连接 UVM 订阅者到 analysis port.

连接过程有两个部分:

- uvm_subscriber 中包含 analysis_export 对象, 我们不需要例化这个对象, 这个过程在继承 uvm_subscriber 的时候就完成了.
- uvm_analysis_port 类提供了 connect()方法.
- 我们通过调用 analysis port 的 connect()方法并传入 analysis_export 对象让订阅者订阅到 analysis port.

这里是用 connect_phase()把订阅者连接到 dice roller 的 dice_test:

```
class dice_test extends uvm_test;
  uvm_component_utils(dice_test);
  dice_roller dice_roller_h;
  coverage coverage_h;
  histogram histogram_h;
  average average_h;
  function void connect_phase(uvm_phase phase);
    dice_roller_h.roll_ap.connect(coverage_h.analysis_export);
    dice_roller_h.roll_ap.connect(histogram_h.analysis_export);
    dice_roller_h.roll_ap.connect(average_h.analysis_export);
```

```
| endfunction : connect_phase
```

图 100. 连接订阅者到 analysis port

我们有三个订阅者，每个都有 `analysis_export` 对象。我们在 `dice_roller_h` 对象里有一个 `analysis port roll_ap`。我们对 `roll_ap` 调用 `roll_ap` 对象并传入每个订阅者的 `analysis_export` 对象。

得 A

我们现在有了连接对象到一起解决问题的设计了。dice roller 掷骰子然后把数值写给订阅者。订阅者创建不同的骰子统计。我们可以把新的统计加到这个程序里来，只用写一个新的订阅对象然后连接到 dice roller。其他的都不需要改。

我们用对象连接图画出了这个设计，然后完成了作业提交。我们用这个图来显示怎么把类连接到一起的：

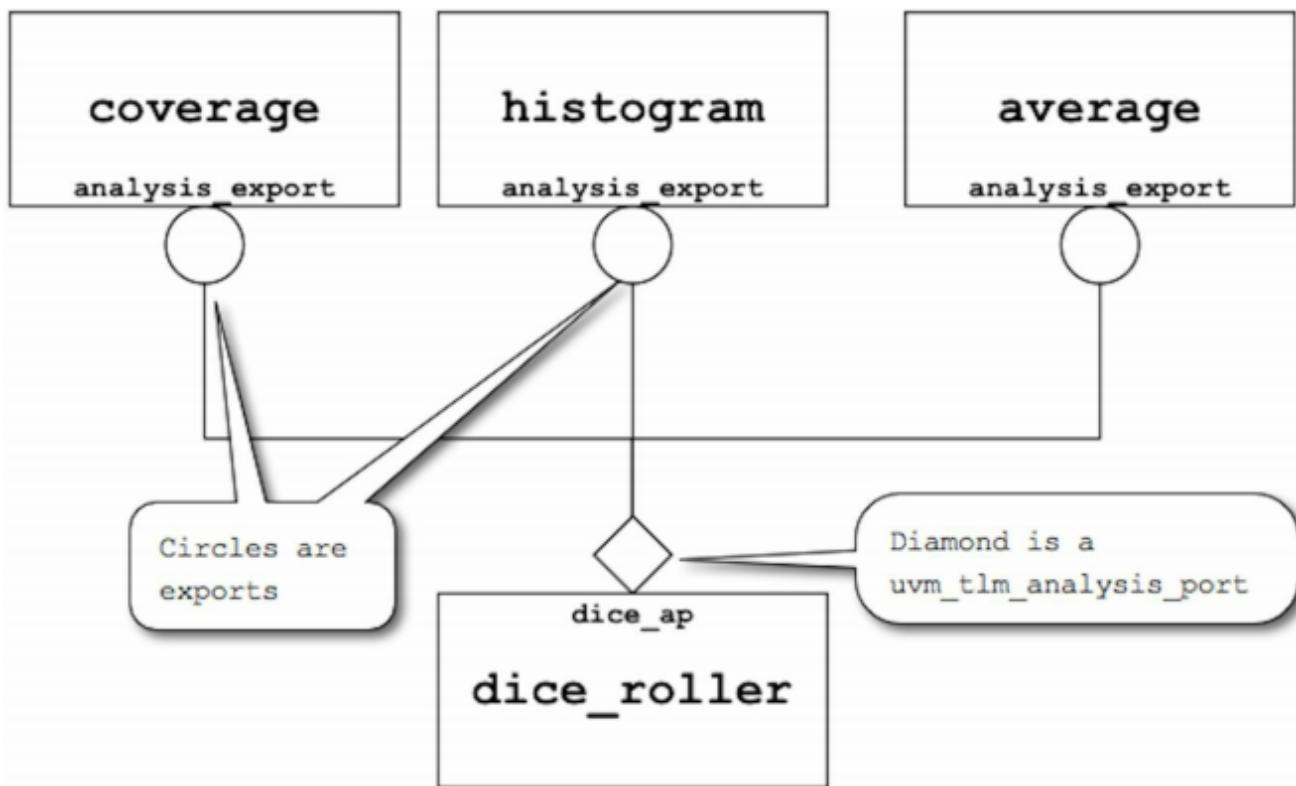


图 101. 对象连接图

这个图显示了我们的 `dice_roller` 里包含了 `dice_ap` 这个 `uvm_analysis_port`，然后每个订阅者都包含 `analysis_export` 对象。这些 `export` 对象是 `port` 的 `connect()` 方法的参数。

总结

在本章我们用 UVM 来让对象分享数据。我们学习了观察者模式，并用

`uvm_analysis_port` 和 `uvm_subscriber` 类来实现这个模式. 我们用这些类来完成掷骰子分析程序, 拿到了 A.

下一章, 我们会把 analysis port 和订阅者应用到 UVM 验证平台里.

第十六章

在验证平台里使用 analysis port

UVM 开发者实现观察者模式(用 `uvm_subscriber` 和 `uvm_analysis_port`)一定是有原因的. 不是为了骰子和直方图, 而是观察者模式非常适合用来监控一个 DUT.

所有的验证平台都做了两件事情:

- 驱动激励到 DUT
- 观测输出

我们的验证平台里一个对象负责驱动激励, 两个对象负责观测响应. `tester_h` 对象驱动激励 `scoreboard_h` 和 `coverage_h` 对象观测响应. 用 UVM 术语来描述的话, `scoreboard_h` 和 `coverage_h` 组成了验证平台的分析层, 这就是 `uvm_analysis_port` 这个名字的由来.

在本章, 我们会看到怎样用 `uvm_analysis_port` 类来创建 TinyALU 验证平台中的分析层.

重复代码问题

编写两段完成几乎相同事情的代码跟在露营时打开帐篷一样, 你这是在招 bug 呢.(双关么, 有点尴尬的)

在我们的 TinyALU 验证平台中就有代码重复的问题, 验证平台中的 `coverage_h` 和 `scoreboard_h` 两个类需要观测 TinyALU 的命令. 在验证平台的之前版本中, 我们给两个类都传递了 BFM, 然后写代码来从信号中提取出指令. 这个是会引入 bug 的重复.

BFM 检测 TinyALU 管脚上的命令再传递给 scoreboard 和 coverage 对象要更好一点. scoreboard 还需要运算结果, 所以我们需要 BFM 检测结果然后发送给验证平台.

验证平台图解

从大份完整代码, 到连接在一起的分散对象时, 在动手写代码前先画好连接关系是很有用的. 这里是我们的用了 analysis port 的验证平台图:

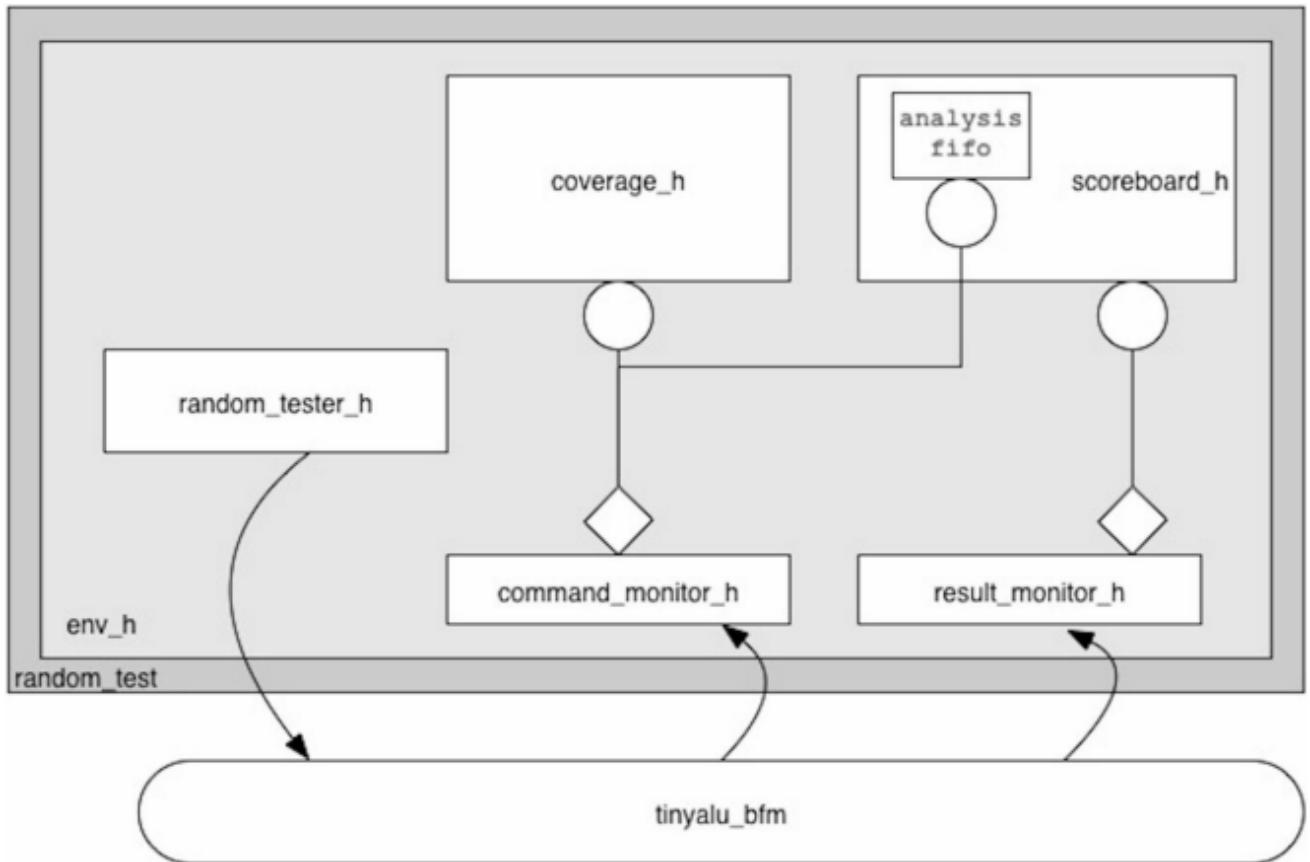


图 102. 带 analysis port 的 TinyALU 验证平台

在之前的验证平台中, tester_h 有一个 BFM 句柄. 然而, 在现在这个验证平台中, tinyalu_bfm 里有 command_monitor_h 和 result_monitor_h 两个对象的句柄. BFM 识别出 TinyALU 的命令和运算结果然后把它们传给这些监控器. 这些监控器反过来把这些数据写入通向 coverage_h 和 scoreboard_h 对象的 analysis port.

scoreboard_h 对象实际上订阅了两个 analysis port: 一个用来取得 TinyALU 命令, 另一个用来取得 TinyALU 运算结构.

在本章, 我们会看到实现这些连接的代码.

BFM 中的对象句柄

OOP 的一大好处, 是你可以通过为一个对象共享句柄来为你的验证平台增加功能. 在这个例子裡, 我要用对象的句柄来把 tinyalu_bfm 连接到其他的验证平台模块上.

我们首先在 BFM 里声明操作这些对象的变量:

```

1  interface tinyalu_bfm;
2      import tinyalu_pkg::*;
3
4      command_monitor command_monitor_h;
5      result_monitor  result_monitor_h;

```

图 103. 声明 BFM 中的类句柄

我们在 tinyalu_pkg 里定义 command_monitor 和 result_monitor 类以及其他类. 我

们已经在 tinyalu_bfm 里引入了 tinyalu_pkg, 现在可以声明对象句柄了.

我们在监控器各自的 build_phase() 方法里设定 command_monitor_h 和 result_monitor_h 变量的值:

```
class command_monitor extends uvm_component;
  `uvm_component_utils(command_monitor);

  uvm_analysis_port #(command_s) ap;

  function void build_phase(uvm_phase phase);
    virtual interface tinyalu_bfm bfm;
      if(!uvm_config_db #(virtual interface tinyalu_bfm)::get(null, "*", "bfm", bfm))
        $fatal("Failed to get BFM");

      bfm.command_monitor_h = this;
      ap = new("ap", this);
    endfunction : build_phase
```

图 104. 复制 command monitor 句柄到 BFM

上面的代码从 UVM 配置数据库里得到 BFM 的句柄, 然后把自己的句柄(this)拷贝到 BFM 的变量里. 现在 BFM 可以通过 command_monitor_h 句柄向验证平台传送数据了.

result_monitor 对 result_monitor_h 句柄做了一样的处理.

监控 TinyALU 的指令

我们用 cmd_monitor 和 rslt_monitor 两个 always 块来监测 TinyALU 的指令和结果. 这里是 cmd_monitor 循环:

```
76   always @(posedge clk) begin : cmd_monitor
77     bit new_command;
78     if (!start)
79       new_command = 1;
80     else
81       if (new_command) begin
82         command_monitor_h.write_to_monitor(A, B, op);
83         new_command = (op == 3'b000); // handle no_op
84       end
85     end : cmd_monitor
```

图 105. BFM 中的 command monitor

cmd_monitor 循环实现了一个简单的状态机. 这个循环在时钟的上升沿检查 start 信号, 信号为真时检查这是否是新命令, 如果是新指令就调用 command_monitor 的 write_to_monitor() 方法向验证平台发送这个命令.

result monitor 循环也类似:

```

96      always @(posedge clk) begin : rslt_monitor
97          if (done)
98              result_monitor_h.write_to_monitor(result);
99      end : rslt_monitor

```

图 106. BFM 中的 result monitor

用 command_monitor_h 操作指令

command_monitor 类从 BFM 里获得指令, 封装到结构体里然后传送给 analysis port. 存储指令数据的结构体 command_s 如下:

```

15     typedef struct {
16         byte unsigned           A;
17         byte unsigned           B;
18         operation_t op;
19     } command_s;

```

图 107. command_s 结构体

write_to_monitor()方法填写了这个结构体然后通过 analysis port 把它发送到验证平台:

```

class command_monitor extends uvm_component;
    `uvm_component_utils(command_monitor);

    uvm_analysis_port #(command_s) ap;

    function void build_phase(uvm_phase phase);
        tinyalu_pkg::bfm_g.command_monitor_h = this;
        ap = new("ap",this);
    endfunction : build_phase

    function void write_to_monitor(byte A, byte B, bit[2:0] op);
        command_s cmd;
        cmd.A = A;
        cmd.B = B;
        cmd.op = op2enum(op);
        $display("COMMAND MONITOR: A:0x%2h B:0x%2h op: %s", A, B, cmd.op.name());
        ap.write(cmd);
    endfunction : write_to_monitor

```

图 108. 使用 analysis port 发送 command

这个类声明了接收 command_s 结构体的 analysis port 并在 build_phase()方法里例化然后用 write_to_monirot()方法向验证平台发送指令.

在我们不断把操作拆分到更小的单元之后, 我们的代码变得越来越简洁清晰了. 这个简单的管道代码可以让我们创建更简单的 coverage 类, 我们接着来看看这个类吧.

TinyALU 的 coverage 类用作订阅者

我们现在有了一个任何类都可以观测 TinyALU 指令的源了, 这要谢谢 command_monitor 类. 这些类只需要订阅到 command_monitor 的 analysis port, 然后实现 write()方法. 这让我们可以创建简单的易于调试和重用的类. 这个例子里, 我用创建了 coverage 类:

```

class coverage extends uvm_subscriber #(command_s);
  uvm_component_utils(coverage)
  byte      unsigned      A;
  byte      unsigned      B;
  operation_t op_set;
  covergroup op_cov;
    coverpoint op_set {
      bins single_cycle[] = {[add_op : xor_op], rst_op,no_op};
      bins multi_cycle = {mul_op};
      bins opn_rst[] = ([add_op:no_op] => rst_op);
      bins rst_opn[] = (rst_op => [add_op:no_op]);
      bins sngl_mul[] = ([add_op:xor_op],no_op => mul_op);
      bins mul_sngl[] = (mul_op => [add_op:xor_op], no_op);
      bins twoops[] = ([add_op:no_op] [* 2]);
      bins manymult = (mul_op [* 3:5]);
      bins rstmulrst  = (rst_op => mul_op [= 2] => rst_op);
      bins rstmulrstim = (rst_op => mul_op [-> 2] => rst_op);
    }
  endgroup
  covergroup zeros_or_ones_on_ops;
    all_ops : coverpoint op_set {
      ignore_bins null_ops = {rst_op, no_op};}
    a_leg: coverpoint A {
      bins zeros = {'h00};
      bins others= {'h01:'hFE};
      bins ones  = {'hFF};
    }
    b_leg: coverpoint B {
      bins zeros = {'h00};
      bins others= {'h01:'hFE};
      bins ones  = {'hFF};
    }
    op_00_FF: cross a_leg, b_leg, all_ops {
      bins add_00 = binsof (all_ops) intersect {add_op} &&
                   (binsof (a_leg.zeros) || binsof (b_leg.zeros));
      bins add_FF = binsof (all_ops) intersect {add_op} &&
                   (binsof (a_leg.ones) || binsof (b_leg.ones));
      bins and_00 = binsof (all_ops) intersect {and_op} &&
                   (binsof (a_leg.zeros) || binsof (b_leg.zeros));
      bins and_FF = binsof (all_ops) intersect {and_op} &&
                   (binsof (a_leg.ones) || binsof (b_leg.ones));
      bins xor_00 = binsof (all_ops) intersect {xor_op} &&
                   (binsof (a_leg.zeros) || binsof (b_leg.zeros));
      bins xor_FF = binsof (all_ops) intersect {xor_op} &&
                   (binsof (a_leg.ones) || binsof (b_leg.ones));
      bins mul_00 = binsof (all_ops) intersect {mul_op} &&
                   (binsof (a_leg.zeros) || binsof (b_leg.zeros));
      bins mul_FF = binsof (all_ops) intersect {mul_op} &&
                   (binsof (a_leg.ones) || binsof (b_leg.ones));
      bins mul_max = binsof (all_ops) intersect {mul_op} &&
                   (binsof (a_leg.ones) && binsof (b_leg.ones));
      ignore_bins others_only =
                     binsof(a_leg.others) && binsof(b_leg.others);
    }
  endgroup

```

```

function new (string name, uvm_component parent);
    super.new(name, parent);
    op_cov = new();
    zeros_or_ones_on_ops = new();
endfunction : new
function void write(command_s t);
    A = t.A;
    B = t.B;
    op_set = t.op;
    op_cov.sample();
    zeros_or_ones_on_ops.sample();
endfunction : write
endclass : coverage

```

图 109. 更简单的新 coverage 类

coverage 类不用再委身来处理信号, 时钟和位数据了, 取而代之的是可以从 command_monitor 的 analysis port 接收简洁的 command_s 结构体数据.

write()方法中的参数类型必须与类声明中的类型参数一致. write()方法中的参数叫做 t, 在我们的例子里, t 是 command_s 结构体类型, 所以我们从结构体里拷贝 A, B 和 op 出来然后收集 coverage.

我们从 uvm_subscriber 继承得到可以操作 command_s 结构体类型的 analysis_export 对象, 我们要用这个对象来订阅 analysis port. 这是最简单的用法.

scoreboard 类因为需要订阅到两个不同的 analysis port, 所以会更复杂一点.

订阅多个 analysis port

UVM 的基本 analysis port 机制允许一个 uvm_subscriber 订阅一个 analysis port. 不过在一些情况下, 比如 scoreboard 里, 我们需要 uvm_subscriber 从两个 analysis port 里取得数据.

这个问题最简单的解决方案是在类里例化另外一个 subscriber 对象然后让这个对象订阅到第二个 analysis port. UVM 提供了 uvm_tlm_analysis_fifo 类来解决这个问题.

要想订阅两个不同的 analysis port, 我们需要两个不同的 analysis_export 对象来传递给 analysis port 的 connect()方法. 在这个例子里, scoreboard 需要为 command_s 和 shortint 配置两个 analysis_export. 这里是 analysis_export 的创建方式:

```

1 class scoreboard extends uvm_subscriber #(shortint);
2     `uvm_component_utils(scoreboard);
3
4     uvm_tlm_analysis_fifo #(command_s) cmd_f;
5
6     function void build_phase(uvm_phase phase);
7         cmd_f = new ("cmd_f", this);
8     endfunction : build_phase

```

图 110. 为两个类创建订阅器

scoreboard 以 shortint 类型继承 uvm_subscriber 类, 这意味着我们有了一个 shortint 类型的 analysis_export, 同时 scoreboard 的 write()方法必须接收 shortint 数据.

但是我们的 scoreboard 必须同时接收 shortint 和 command_s 结构体, 处理了 shortint

之后还需要处理 command_s. 我要用 uvm_tlm_analysis_fifo.

uvm_tlm_analysis_fifo 是一个参数化类, 一端提供 analysis_export, 另外一端提供 try_get()方法. 我们用 analysis_export 订阅需要的 analysis_port 然后用 try_get()方法来从 FIFO 中取出数据.

这个例子里, 我们声明了接收 command_s 结构体的 uvm_tlm_analysis_fifo 然后在 run_phase()方法里取出数据. 这让我们有了在一个类里订阅两种不同数据的方法.

scoreboard 的 write()方法假定了指令输入系统之后才会输出 1 个结果. 当 result_monitor 发送了一个结果, 我们从 FIFO 里取出指令, 直到我们找到 no_op 和 rst_op 之外的指令. 我们用这个指令来计算预期值并与结果比较, 代码如下:

```
function void write(shortint t);
    shortint predicted_result;
    command_s cmd;
    cmd.op = no_op;
    do
        if (!cmd_f.try_get(cmd)) $fatal(1, "No command in self checker");
        while ((cmd.op == no_op) || (cmd.op == rst_op));

    case (cmd.op)
        add_op: predicted_result = cmd.A + cmd.B;
        and_op: predicted_result = cmd.A & cmd.B;
        xor_op: predicted_result = cmd.A ^ cmd.B;
        mul_op: predicted_result = cmd.A * cmd.B;
    endcase // case (op_set)

    if (predicted_result != t)
        $error (
            "FAILED: A: %2h B: %2h op: %s actual result: %4h expected: %4h",
            cmd.A, cmd.B, cmd.op.name(), t, predicted_result);
endfunction : write
```

图 111. write()方法中的 scoreboard 任务

FIFO 的 try_get()方法从 FIFO 里读出一个指令, FIFO 为空的话返回 0, 由于 FIFO 不应该为空, 所以这个情况发生的话我们就抛出一个致命异常.

我们用 do...while()循环读取指令, 直到读取出有效指令. 这必须是产生结果的指令, 我们用这个来计算预测值并进行比较.

现在我们有了 2 个 monitor 类和 2 个 analysis 类, 剩下的就是把这些 monitor 和 analysis 工具连接到一起.

订阅到 monitor

我们在 env 里用 connect_phase()来连接 analysis 对象和 monitor:

```
24     function void connect_phase(uvm_phase phase);
25         result_monitor_h.ap.connect(scoreboard_h.analysis_export);
26         command_monitor_h.ap.connect(scoreboard_h.cmd_f.analysis_export);
27         command_monitor_h.ap.connect(coverage_h.analysis_export);
28     endfunction : connect_phase
```

图 112. 订阅到 monitor

`coverage_h` 对象和 `scoreboard_h` 对象各有一个 `analysis_export`, 我们调用 `connect()`方法将订阅者连接到 port. 我们用 FIFO 的 `analysis_export` 来连接 scoreboard 到 `command_monitor`.

总结

在本章中, 我们用了 `uvm_analysis_port` 实现验证平台中的观察者模式. 我们探讨了在 BFM 中聚合所有信号级的观测, 用 monitor 类和 analysis port 将 monitor 数据发送给任意数量的 analysis 类.

`uvm_analysis_port` 的对象间通信是一种线程内通信, 所有的函数调用都在同一个线程内. `cmd_monitor` always 块调用 `write_to_monitor()` 的时候其实是在调用所在线程内所有订阅者的 `write()`.

线程内通信适合用于这种情况下的分析, 不过我们往往需要在线程间传递信息. 这个看起来会很复杂, 不过 UVM 给我们提供了轻松完成这个任务的工具.

第十七章

线程间通信

"线程间通信"这个词像是恐怖的 Java 词汇, 不过这个是我们 Verilog 和 VHDL 工程师从第一天就开始做的. 我们所有写的 HDL 代码都是基于多线程(initial, always 和 process 块)和它们之间的通信的.

假定有一个 SystemVerilog 写的 Producer/Consumer 程序, Producer 产生数据, Consumer 读取数据:

```
module producer(output byte shared, input bit put_it, output bit get_it);
    initial
        repeat(3) begin
            $display("Sent %0d", ++shared);
            get_it = ~get_it;
            @(put_it);
        end
    endmodule : producer

module consumer(input byte shared, output bit put_it, input bit get_it);
    initial
        forever begin
            @(get_it);
            $display("Received: %0d", shared);
            put_it = ~put_it;
        end
    endmodule : consumer

module top;
    byte shared;
    producer p (shared, put_it, get_it);
    consumer c (shared, put_it, get_it);
endmodule : top
```

图 103. 基于 module 的线程间通信

producer 和 consumer 中的两个 initial 块是两个线程. 线程间通信可以简单的描述为, 我们可以通过 module 的端口从一个线程发送数据到另外的线程. 我们用 shared 总线来传递数据, 用 put_it 和 get_it 信号来挂起各自的线程让对方线程运行.

producer 这个 module 线程写入数据到 shared 变量然后翻转 get_it 信号来通知 consumer module 数据已经准备完毕. 然后 producer 阻塞在 put_it 信号上让 consumer 来取得数据. consumer 开始时阻塞在 get_it 信号上, 等待从 producer 来的命令, 当 producer 翻转 get_it 信号的时候, consumer 唤醒了, 然后从 shared 信号上读取数据接着翻转 put_it 来释放 producer. 之后 consumer 再一次阻塞在 get_it 信号上等待下一次数据.

通信的结果是:

```

22 # Loading work.top(fast)
23 # Sent 1
24 # Received: 1
25 # Sent 2
26 # Received: 2
27 # Sent 3
28 # Received: 3

```

图 114. 通信成功

本章中我们将学习怎么用 UVM 的对象来做这种事情.

等一下, 没有对象端口吗?

在前面的例子中, 线程通过 producer 和 consumer module 的端口通信, 我们用端口来传递 module 间的 shared 变量和通信信号.

面向对象的 SystemVerilog 没有 module 端口这样的内建语言结构, 不过, SystemVerilog 提供了对象间共享句柄的功能, 这个包括信号量和邮箱这样的线程协调结构.

大家可以用这些块来实现线程通信系统, 不过这样每个人的方案都会不尽相同. UVM 通过提供一套通用的, 隐藏了细节的, 线程间通信方案来解决这个问题. 这个方案有两个基本部分:

- 端口---在 uvm_component 中例化的, 可以让 run_phase() task 跟其他线程通信的对象. 我们用 put 管道和 get 管道来向其他线程收发数据.
- TLM FIFO---连接 put 端口和 get 端口的对象. TLM 表示 transaction level modeling, 这个是早期 UVM 版本的概念. TLM FIFOs 可以传递 transaction(我们后面会看到)以及其他任意类型数据.

TLM FIFO 只操作一个对象, 这对一个 FIFO 来讲是很烂的容量, 不过在线程间通信上这个特点是很有用的.

在本章, 我们将用端口和 FIFO 来重新创建我们的基于 module 的 producer/consumer.

producer 作为对象

在之前的基于 module 的例子里, Producer 创建了三个数据然后用 module 的端口来发送数据并协调信号(put_it 和 get_it). 为了用对象来做一样的事情, 我们需要声明例化 uvm_put_port 对象:

```

1 class producer extends uvm_component;
2   `uvm_component_utils(producer)
3
4   int shared;
5   uvm_put_port #(int) put_port_h;
6
7   function void build_phase(uvm_phase phase);
8     put_port_h = new("put_port_h", this);
9   endfunction : build_phase

```

图 115. 声明例化 put 端口

uvm_put_port 是参数化类, 我们需要指定传送的数据类型, 这个例子里我们发送的是

int 数据.

put_port_h 对象现在可以用 int 数据然后发送到 FIFO 里了. 这里是 producer 用端口发送三个数字的代码:

```
17      task run_phase(uvm_phase phase);
18          phase.raise_objection(this);
19          repeat (3) begin
20              put_port_h.put(++shared);
21              $display("Sent %0d", shared);
22          end
23          phase.drop_objection(this);
24      endtask : run_phase
25  endclass : producer
```

图 116. producer 对象中发送数据

这个跟基于 module 的代码很像, 不过更简单. 基于 module 的代码要用 get_it 和 put_it 信号来处理线程间通信, 这里 put_port_h 把这些都处理了. 循环里第一次调用 put() 把数据送到端口里, 数据就送到 FIFO 里了. 现在 FIFO 已经满了, 再次循环到调用 put() 存放下一个数据的时候, 由于 FIFO 已经满了, put() 产生了阻塞. 阻塞会持续到 consumer 从 FIFO 里得到数据.

consumer 作为对象

consumer 声明例化 uvm_get_port 对象来从 FIFO 里取出数据:

```
1  class consumer extends uvm_component;
2      `uvm_component_utils(consumer);
3
4      uvm_get_port #(int) get_port_h;
5      int shared;
6
7      function void build_phase(uvm_phase phase);
8          get_port_h = new("get_port_h", this);
9      endfunction : build_phase
```

图 117. 声明例化 get 端口

uvm_get_port 也是一个参数化类, 我们可以用与 producer 的 put 端口类一样的类型来声明 get_port_h 变量.

有了端口之后, 我们在 run_phase() 线程里从 producer 里取出数据:

```
16      task run_phase(uvm_phase phase);
17          forever begin
18              get_port_h.get(shared);
19              $display("Received: %0d", shared);
20          end
21      endtask : run_phase
22  endclass : consumer
```

图 118. 从 producer 接收数据

跟在 producer 中一样, `uvm_get_port` 处理了所有线程间协调. 这个 task 开始的时候, 我们调用 `get()` 从端口取出数据, 由于 FIFO 是空的, 这里就阻塞了. producer 填充 FIFO 然后阻塞, 这个时候这边的唤醒然后从 FIFO 里取出数据. 这个过程循环从 FIFO 里取出数据, 取出来之后 FIFO 又空了, 于是又进入阻塞状态.

简而言之, 这个系统是这样运行的:

- producer 填充 FIFO 然后挂起
- consumer 清空 FIFO 然后挂起
- producer 填充 FIFO 然后挂起
- 如此循环

剩下要做的就是用 FIFO 连接端口.

连接端口

在 module 的例子里, 我们在顶层 module 例化 producer 和 consumer 然后连接它们的端口. 这里要做的事情类似, 我们在 `uvm_test` 里例化 producer, consumer 和 `tlm_fifo` 然后连接端口到 `tlm_fifo`.

这里是例化:

```
1  class communication_test extends uvm_test;
2    `uvm_component_utils(communication_test)
3
4    producer producer_h;
5    consumer consumer_h;
6    uvm_tlm_fifo #(int) fifo_h;
7
8    function void build_phase(uvm_phase phase);
9      producer_h = new("producer_h", this);
10     consumer_h = new("consumer_h", this);
11     fifo_h = new("fifo_h", this);
12   endfunction : build_phase
```

图 119. 例化 producer, consumer 和 TLM FIFO

注意到 `uvm_tlm_fifo` 有着 producer 的 `put` 端口和 consumer 的 `get` 端口一样的类型参数.

UVM 自顶向下调用所有对象的 `build_phase()`, 所以最先例化的是 `producer_h`, `consumer_h` 和 `fifo_h` 对象, 然后调用它们的 `build` 方法. `producer_h` 和 `consumer_h` 的 `build_phase()` 方法在各自的对象里例化了这些端口.

`build phase` 方法全都调用完之后, UVM 就开始自底向上调用各组件的 `connect_phase()` 方法.

连接端口到 TLM FIFO

在前面章节中我们了解了 `uvm_analysis_port` 对象有 `connect()` 方法, `connect()` 方法用 `analysis_export` 对象作为参数. 此外 `uvm_subscriber` 类提供了 `analysis_export` 对象.

这个方法也适用于 put 端口, get 端口和 uvm_tlm_fifo. uvm_tlm_fifo 提供了两个对象: put_export 和 get_export. 通过在这些端口的 connect 方法里传入这些对象可以连接到 uvm_tlm_fifo:

```
18     function void connect_phase(uvm_phase phase);
19         producer_h.put_port_h.connect(fifo_h.put_export);
20         consumer_h.get_port_h.connect(fifo_h.get_export);
21
22     endfunction : connect_phase
23 endclass : communication_test
```

图 120. 连接端口到 FIFO

connect_phase()方法从producer和consumer里引用了端口然后调用了它们的connect()方法. fifo_h 对象里有 put_export 和 get_export 对象, 我们传入这些对象到端口来创建连接.

通信的结果如下:

```
58 # Sent 1
59 # Received: 1
60 # Sent 2
61 # Received: 2
62 # Sent 3
63 # Received: 3
```

图 121. 使用对象的通信

非阻塞通信

上面用到的阻塞通信在你不关注时钟和时间的时候是很好的, 需要在两个空闲线程通信的时候, 你可以用 get() 和 put() 调用来阻塞线程而不会引发 bug.

不过, 有其他像是时钟沿这样的阻塞事件时, 混合 get() 和 put() 的调用可能就不能正常工作了. 因为时钟驱动的系统假定你在每个时钟沿都没有阻塞在其它事件上.

UVM 提供了非阻塞版本的 put() 和 get(), try_put() 和 try_get(), 来解决这个问题.

try_put() 和 try_get() 跟 put() 和 get() 的作用是一样的, 不过它们不会阻塞, 反之它们会返回一个位来通知你是否 put 或 get 成功了. 根据这个位来做正确的处理就是你的任务了. 我们来小小的改动一下 producer 和 consumer 来展示 try_get 的用法. 在这个例子里, 我们要用 14ns 的时钟沿工作, 同时我们的 producer 每 17ns 进行一次数据的写入.

下面是 consumer 里在时钟下用 try_get():

```
14 task run_phase(uvm_phase phase);
15     super.run_phase(phase);
16     forever begin
17         @(posedge clk_bfm_i.clk);
18         if(get_port_h.try_get(shared))
19             $display("%0tns Received: %0d", $time,shared);
20     end
21 endtask : run_phase
```

图 122. 带 clock 的 consumer

这个 consumer 有一个 forever 循环, 在每个时钟上升沿阻塞. 在每个时钟上升沿, consumer 尝试从 FIFO 里. 如果 try_get() 返回 1(意外着得到了数据), 就打印出数据; 如果返回 0, 就继续循环等待下一个时钟上升沿.

这里是带时间的通信过程:

```

59 # 17ns Sent 1
60 # 21ns Received: 1
61 # 34ns Sent 2
62 # 35ns Received: 2
63 # 51ns Sent 3
64 # 63ns Received: 3

```

图 123. 非阻塞通信

我们看到 consumer 的时钟在 7ns 启动, 然后每 14ns 触发一次; producer 每 17ns 产生一次数据. consumer 只在时钟沿取出数据, 只在取得数据的时候打印出来.

图解 put 端口和 get 端口

在之前的章节我们画出里验证平台里, 用菱形块来表示 analysis port, 用圆圈来表示 analysis export. 这里我们可以画出类似的 put 端口和 get 端口图, 不过用正方形来表示 port, export 还是用圆圈:

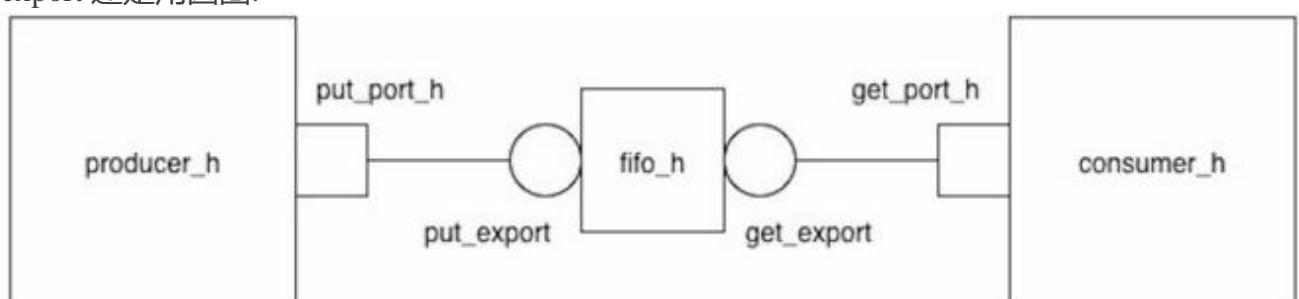


图 124. producer/consumer 图

我们用正方形来画出 put/get port, 用菱形来画 analysis port, 为什么要这么来我们在下一章解释. 我们不区分阻塞和非阻塞的 put 和 get 函数.

总结

本章我们了解到了对象的线程间通信跟 module 的线程间通信是相似的, UVM 用 uvm_put_port, uvm_get_port 和 uvm_tlm_fifo 提供了线程间通信机制. 我们看到了, 任何想要跟另外的线程通信的对象需要例化一个端口, 而这个端口需要连接到 FIFO.

在下一章, 我们要用线程间通信来把我们的验证平台拆分成更小的块. 我们要把测试激励的生成和 DUT 的驱动分离.

第十八章

put 和 get 端口的实践

前面的章节中, 我们学习了用 put 和 get 端口来进行线程间通信. 在本章我们会为 TinyALU 验证平台添加功能.

整本书中我们都在寻找单个对象处理多项任务的地方, 然后把这些任务拆分到多个类中. 处理单个任务的类要更容易调试和复用.

在本章, 我们要来探讨 `base_tester` 类(就是派生出来的 `random_tester` 和 `add_tester` 类). 我们的 `base_tester` 做了两件事情, 选择了发送给验证平台的指令类型, 然后给 BFM 应用. 我们要把这个操作拆分成两个类, 一个用来选择指令, 另外一个跟 BFM 交互. 我们后面会看到这样的拆分让验证平台的操作更灵活了.

新的验证平台看起来是这样的:

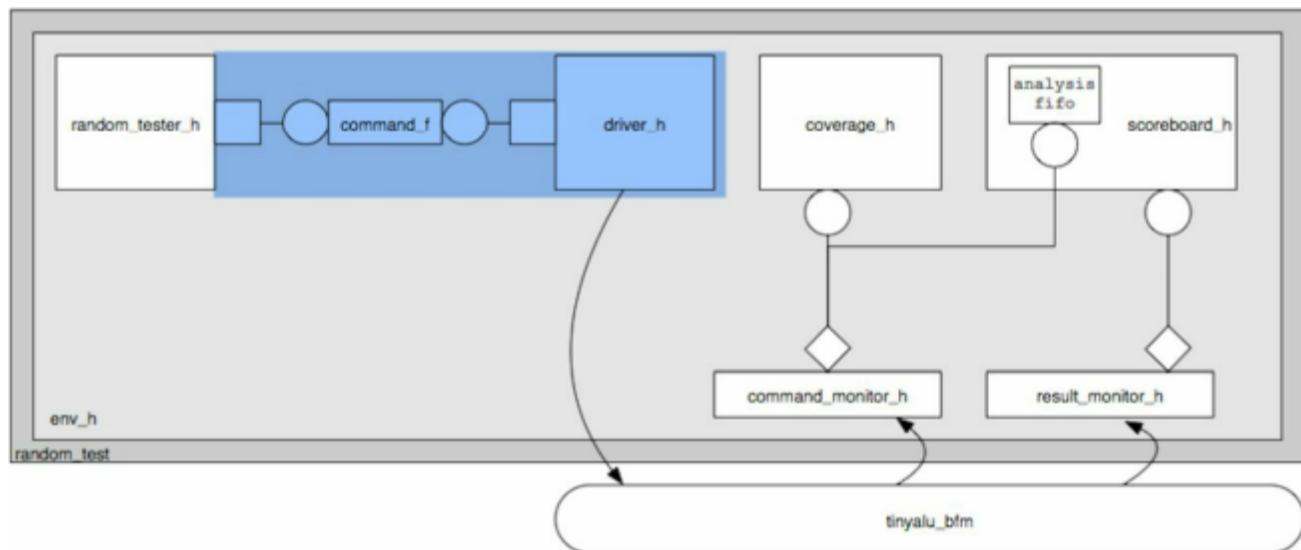


图 125. 从 BFM 接口中剥离激励的生成

验证平台里加色的部分是新的, `driver` 是指从验证平台中取出数据然后把它转化到 BFM 的信号中的对象. 我们在验证平台里添加了 `driver_h` 对象然后用 FIFO 连接到 `tester_h` 对象上. 现在我们的验证平台有了专门选择激励的类和专门传送激励的类.

这里是连接这些对象的 `env_h` 对象:

```

1 class env extends uvm_env;
2   `uvm_component_utils(env);
3
4   random_tester      random_tester_h;
5   driver            driver_h;
6   uvm_tlm_fifo #(command_s) command_f;
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25   function void connect_phase(uvm_phase phase);
26     driver_h.command_port.connect(command_f.get_export);
27     random_tester_h.command_port.connect(command_f.put_export);
28
29     result_monitor_h.ap.connect(scoreboard_h.analysis_export);

```

图 126. 声明并连接 tester 和 driver

我们现在有了 tester, driver 和一个接收 command_s 结构体的 uvm_tlm_fifo. 我们已经在 tester 里添加了 command_port 这个 put 端口, 在 driver 里添加了 command_port 这个 get 端口. 然后我们在 FIFO 里把它们的 get_export 和 put_export 对象传入各自的 connect() 方法.

有时候完全记住连接 port 跟 export 的接法是困难的. 要记住"port 主动连接 export", 就是传送 export 到 port 的 connect()方法.

我们看一看对象内部.

base_tester 类

之前创建 base_tester 类的先见之明就要显露出来了. base_tester 跟 driver 之间的连接不影响 get_op() 和 get_data() 方法, 所以 random_tester 和 add_tester 还是不用改变, 它们会继承到新的功能.

base_tester 跟之前一样, 不同之处在于现在不包含 BFM 句柄了, 而是有 command_port 这个 uvm_put_port. 我们声明例化 port 端口如下:

```

1 virtual class base_tester extends uvm_component;
2   `uvm_component_utils(base_tester)
3   virtual tinyalu_bfm bfm;
4
5   uvm_put_port #(command_s) command_port;
6
7   function void build_phase(uvm_phase phase);
8     command_port = new("command_port", this);
9   endfunction : build_phase

```

图 127. 在 tester 中声明例化 port 端口

现在有了 port 端口, 我们可以用来发送数据到验证平台了:

```

36      task run_phase(uvm_phase phase);
37          byte         unsigned         iA;
38          byte         unsigned         iB;
39          operation_t               op_set;
40          command_s    command;
41
42          phase.raise_objection(this);
43          command.op = rst_op;
44          command_port.put(command);
45          repeat (1000) begin : random_loop
46              command.op = get_op();
47              command.A =  get_data();
48              command.B =  get_data();
49              command_port.put(command);
50          end : random_loop
51          #500;
52          phase.drop_objection(this);
53
54      endtask : run_phase

```

图 128. tester 使用 put 端口

base_tester 声明了 command_s 变量, 用指令和数据填充它然后通过 put 端口发送到验证平台. 发送完数据到 put 端口之后, 我们就无须再担心了; 这是其他人的任务.

在大型的复杂验证平台里, 这个拆分行为有大益处, 因为生成合适的数据是麻烦事, 做这个的人不想去关心信号级的考量, 这个是 driver 类的任务.

TinyALU 的 driver

driver 是从 get 端口取出指令然后用 BFM 的 send_op 发送指令给 BFM 的简单类. 下面是 driver 的代码:

```

class driver extends uvm_component;
  `uvm_component_utils(driver)

  virtual interface tinyalu_bfm bfm;

  uvm_get_port #(command_s) command_port;

  function void build_phase(uvm_phase phase);
    if(!uvm_config_db #(virtual interface tinyalu_bfm)::get(null, "*", "bfm",
      $fatal("Failed to get BFM"));
    command_port = new("command_port",this);
  endfunction : build_phase

  task run_phase(uvm_phase phase);
    command_s    command;
    shortint     result;

    forever begin : command_loop
      command_port.get(command);
      bfm.send_op(command.A, command.B, command.op, result);
    end : command_loop
  endtask : run_phase

```

图 129. 带 get 端口的 TinyALU driver

driver 不停循环从 get 端口取出指令然后发送到 BFM. 如果 get_port 里没有指令了, driver 就阻塞直到 tester 发送新指令.

总结

本章我们使用了 UVM 线程间通信类来进一步提炼我们的 TinyALU 验证平台. 我们现在有了分离激励和分析功能的验证平台, 然后我们进一步把激励功能拆分成激励生成(tester 类)和激励应用(driver 类).

我们现在有了轻易驱动大量激励到验证平台并在分析层捕捉的技术. 这个引出了怎么处理这些数据的问题. 在这点上, 我们已经使用了 SystemVerilog 的 \$display, \$error 和 \$fatal 系统调用, 不过这些对处理现代验证平台数据来讲都太粗糙了.

我们需要更强大的报告工具, UVM 提供了. 在下一章, 我们会探讨 UVM 的报告工具.

第十九章

UVM 报告

伊万·巴甫洛夫最出名的就是他一摇铃就让狗流口水，不过这个没有让他得到诺贝尔奖，让他得诺贝尔奖的是他在消化领域的研究。他喂狗，食物在狗的体内流动的时候他从它们的消化系统里取样。

我们工程师也经常做同样的事情，不过是对数据。我们穿过验证平台把数据传送到 DUT 然后从出来的一端观察。当出现问题的时候，我们需要打印数据采样到屏幕。

验证平台生成了大量数据，如果我们用 \$display 这种没过滤的打印工具的话，我们很快就会被信息淹没。太多信息了会让我们不知道发生了什么。

UVM 提供了一套报告系统来解决这个问题。所有的 uvm_component 都提供了打印方法。我们可以用 3 个 UVM 报告宏来打印信息到屏幕，然后我们用 end_of_elaboration_phase() 来控制输出。

先看看输出宏吧。

UVM 报告宏

UVM 提供了四种展示不同严重度信息的报告宏：

```
uvm_info(<Message ID String>, <Message String>, <Verbosity>)
uvm_warning(<Message ID String>, <Message String>)
uvm_error(<Message ID String>, <Message String>)
uvm_fatal(<Message ID String>, <Message String>)
```

在查看这些宏的参数之前，我们先看看其中两个的应用。这会让我们在讨论细节之前先得到这些宏工作机制的感受。下面的代码来自与 scoreboard 类：

```
data_str = $sformatf("%2h %0s %2h = %4h (%4h predicted)",
                      cmd.A, cmd.op.name(), cmd.B, t, predicted_result);

if (predicted_result != t)
  `uvm_error("SCOREBOARD", {"FAIL: ",data_str})
else
  `uvm_info ("SCOREBOARD", {"PASS: ",data_str}, UVM_HIGH)
```

图 130. scoreboard 里使用 UVM 报告

上面的代码展示了 UVM 报告宏和参数填充的使用。要注意四个地方：

- 严重度---显然错误要比信息要比通知信息更严重。UVM 提供了四种不同的宏名：info, warning, error, fatal 来识别不同的严重度。
- Message ID 字符串---"SCOREBOARD"字符串标识了信息的类型，UVM 会记录每个 ID 发送了多少次信息。我们还可以用 ID 来控制 UVM 相关操作。
- Message 字符串---Message 字符串包含了读者读到的信息。这个例子里，我们连接了 "PASS" 或 "FAIL" 字符串到 data_str 字符串。创建 data_str 字符串用的是 \$sformatf()。

- Verbosity---只有 uvm_info()宏有 verbosity 参数. 这个参数控制了信息是否打印. 我们后面会来探讨这个.

运行 10 次随机指令之后, 得到如下输出:

```
# UVM_ERROR tb_classes/scoreboard.svh(38) @ 510:
uvm_test_top.env_h.scoreboard_h [SCOREBOARD] FAIL: 4f add_op ff = 014e (014f
predicted)
```

图 131. 出错了

预期响应跟实际不符, 所以我们调用了 uvm_error()宏来生成了上面的输出. 这个例子显示了 UVM 的典型信息域:

- UVM_ERROR---信息严重度
- tb_classes/tinyalu...---调用对象和所在的行号
- @510---得到错误的仿真时间标
- uvm_test_top.env_h.scoreboard_h---调用在 UVM 层级中的位置
- [SCOREBOARD]---我们提供的信息 ID
- FAIL: 4f add_op....---我们用\$sformatf 创建的字符串

所有的 UVM 报告宏创建的信息都符合这个格式, 严重度各有不同.

运行这个测试用例的时候, 我们说过我们执行了 10 次指令, 而在图 130 中我们看到 uvm_info()宏会在测试用例通过的情况下调用. 不过我们只看到了错误信息, 这是为什么呢?

这是因为 uvm_info()信息的严重度为 UVM_HIGH. 这个阻止了它的打印. verbosity 的控制是 UVM 报告系统的关键特性. 我们来看一下.

UVM verbosity 等级

uvm_info 宏有三个参数, 第三个叫做 verbosity, 用来处理我们 debug 时遇到的问题: 信息太多了.

把 debug 信息放在代码里是常见的, 这些通常是 SystemVerilog 里的\$display()语句, 用来打印数据帮助我们 debug. 然而, 在调试结束时我们要么注释掉这些语句, 要么就得忍受好几大页的伪造信息. UVM 的 verbosity 解决了这个问题.

UVM 用两步来控制信息:

- 在 uvm_info()里提供 verbosity.
- 在仿真的时候设置 verbosity 门限. 我们通过为仿真设置门限来控制输出, 无论是全局的还是在 UVM 层级的特定部分.

UVM 自带了六种 verbosity 级别:

```

305 |     typedef enum
306 |     {
307 |         UVM_NONE    = 0,
308 |         UVM_LOW     = 100,
309 |         UVM_MEDIUM  = 200,
310 |         UVM_HIGH    = 300,
311 |         UVM_FULL    = 400,
312 |         UVM_DEBUG   = 500
313 |     } uvm_verbosity;

```

图 132. UVM verbosity 级别

UVM 默认的门限是 UVM_MEDIUM, 如果没改变门限的话, 任何 verbosity 大于 UVM_MEDIUM 的信息都不会被打印. 现在就知道为什么通过信息没有打印出来了.

```

else
`uvm_info ( "SCOREBOARD", {"PASS: ",data_str}, UVM_HIGH)

```

图 133. debug 信息示例

这个信息的 verbosity 设置为 UVM_HIGH, 也就是 300. 验证平台中默认的 verbosity UVM_MEDIUM 是 200. 由于 300 比 200 大, 于是信息没有在门限之下就不打印了.

如果我们要在指令正常执行时看到信息, 就需要把 verbosity 门限提升到 UVM_HIGH.

设置 verbosity 门限(级别)

我们用设定 verbosity 门限来控制 `uvm_info()` 信息的可见数量. 如果我们调低了门限, 门限之下的信息会更少. 如果我们调高门限则反之.

控制门限有两种方式: 整个仿真全局性的, 还有 UVM 层级中一个特定部分的.

设置全局 verbosity 门限

我们用 `+UVM_VERBOSITY` 加号参数来控制全局的 verbosity. 如果我们想要看到 scoreboard 的传送信息, 就像这样提升 verbosity:

```

vsim top_optim -coverage +UVM_TESTNAME=random_test +UVM_VERBOSITY=UVM_HIGH

```

图 134. 设置全局 verbosity

`+UVM_VERBOSITY` 加号参数为这个仿真设置了 verbosity 门限. `UVM_HIGH` 字符串来自 `uvm_verbosity` 枚举类型中的值, 那里可以看到我们信息的 verbosity.

在 UVM 层级中设置 verbosity

全局设置 verbosity 门限是简单的, 不过这个可能会导致一个大验证平台中的信息过载. 如果你跟其他 20 个工程师合作项目, 他们有自己的高 verbosity 的 debug 语句, 提升门限就会像开了泄洪闸.

好在 UVM 开发者考虑了这个. 所有的 `uvm_component` 都提供了报告控制方法, 包括控制

verbosity 门限的。有两种方式，一种是控制单个组件，一种是控制一个组件和这个组件层级以下的组件。

我说到层级的时候，不是在说 DUT 的 module 层级，而是在说 UVM 在所有组件中用 build_phase()方法创建的层级。比如，这个是本章例子中的 UVM 层级：

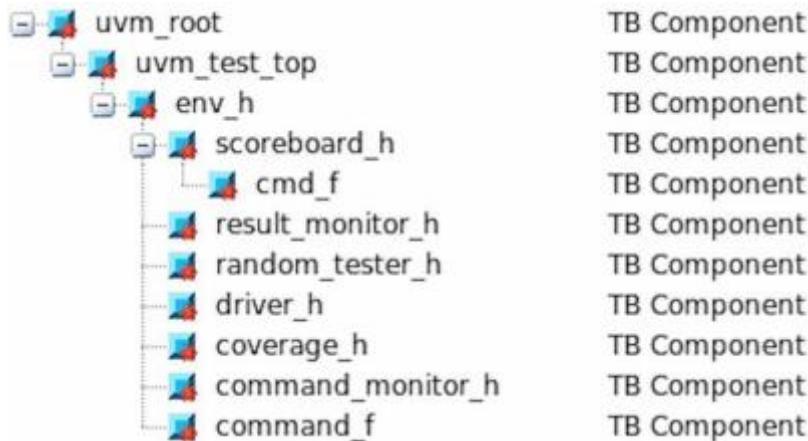


图 135. TinyALU 验证平台的 UVM 层级

如果我们想用 UVM 层级来控制 verbosity 门限，需要在层级建立之后仿真运行之前调用报告方法。我们用刚好在 run_phase()方法之前运行的 end_of_elaboration_phase()这个 UVM 方法来处理这个事情。

在下面的例子中，我们用 env 中的 end_of_elaboration_phase()方法来为 scoreboard 和在它之下的组件设置 verbosity 门限：

```
24 |     function void end_of_elaboration_phase(uvm_phase phase);
25 |         scoreboard_h.set_report_verbosity_level_hier(UVM_HIGH);
26 |     endfunction : end_of_elaboration_phase
```

图 136. 在 scoreboard 里设置 verbosity

上面的 set_report_verbosity_level_hier()方法为 scoreboard 和在它层级之下的组件设置 verbosity 门限。如果我们想单单为 scoreboard_h 设置 verbosity，就用一样的去掉 "_hier" 后缀的调用(set_report_verbosity_level(UVM_HIGH))。

这里是使用了更高 verbosity 之后的输出：

```
# UVM_INFO @ 0: reporter [RNTST] Running test_random_test...
# UVM_INFO tb_classes/scoreboard.svh(37) @ 130:
uvm_test_top.env_h.scoreboard_h [SCOREBOARD] PASS: 09 mul_op ff = 08f7 (08f7
predicted)
# UVM_ERROR tb_classes/scoreboard.svh(35) @ 210:
uvm_test_top.env_h.scoreboard_h [SCOREBOARD] FAIL: c5 add_op e5 = 01aa (01ab
predicted)
# UVM_INFO verilog_src/uvm-1.1c/src/base/uvm_objection.svh(1121) @ 791:
reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
```

图 137. scoreboard 输出显示 pass

现在我们的 scoreboard_h 组件的通过和失败结果都输出了。我们可以看到乘法指令结果正常通过，我们的 16 进制计算器告诉我们这的确是正确的结果。

scoreboard 组件的 add_op 做的没这么好，我们可以看到预期结果跟实际结果不相符，但是我们的计算器显示实际结果是正确的。我们的 scoreboard 出 bug 了。

在我们的小验证平台里我们可以直接简单的进入 scoreboard 然后改 bug, 不过在有着复杂的 scoreboard 的大型验证平台, 我们不能改动 scoreboard. 我们得提交 bug, 之后禁用它直至修复完成再继续使用.

我们怎么禁用 scoreboard_h 错误信息呢? 第一种想法是把 verbosity 门限降到 0, 这样会阻断所有信息. 然后这个只对 uvm_info 信息有效, 阻断 uvm_error() 信息需要更猛的药.

禁用 warning, error 和 fatal 信息

通过 verbosity 门限来对 warning, error 和 fatal 信息消声是无效的. 为了禁用这些信息, 我们需要用 UVM 报告这的另外一种控制手段, action.

UVM 报告宏可以做打印到屏幕之外的事情, 实际上, 它们可以做包括打印到屏幕在内的 6 件事情:

```
277     typedef enum
278     {
279         UVM_NO_ACTION = 'b000000,
280         UVM_DISPLAY   = 'b000001,
281         UVM_LOG        = 'b000010,
282         UVM_COUNT      = 'b000100,
283         UVM_EXIT       = 'b001000,
284         UVM_CALL_HOOK  = 'b010000,
285         UVM_STOP        = 'b100000
286     } uvm_action_type;
```

图 138. UVM 报告操作

我们可以控制哪种严重度级别使用哪种动作, 我们也可以用"或"来创建多种复合操作. action 的细节在本书之外了. 在本例中, 我们只是简单的用 UVM_NO_ACTION 来禁用 scoreboard 中的 error:

```
function void end_of_elaboration_phase(uvm_phase phase);
    scoreboard_h.set_report_severity_action_hier(UVM_ERROR, UVM_NO_ACTION);
endfunction : end_of_elaboration_phase
```

图 139. scoreboard 中禁用 error

上面的代码控制了 scoreboard 和其内部的组件 . set_report_severity_action_hier() 指示系统不去处理严重度为 ERROR(uvm_error()) 的信息.

结果就清静了:

```
# UVM_INFO @ 0: reporter [RNTST] Running test_random_test...
# UVM_INFO verilog_src/uvm-1.1c/src/base/uvm_objection.svh(1121) @ 791:
reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO : 4
# UVM_WARNING : 0
```

图 140. 你若安好

仿真运行的很干净, 一旦 scoreboard 修复了, 我们就可以重新打开 error 信息了.

总结

本章我们用 UVM 报告系统来生成信息. 在大型验证平台中报告系统对信息的管理是很重要的, 报告系统得以让我们创建 debug 信息并把它们保留在代码里.

通过把 TinyALU 验证平台从一个完全的 HDL 式验证平台转换成模块化的基于对象验证平台, 我们探索了 UVM, 我们学习了用不同的类操作, 展示数据. 然而, 数据本身还没有转换成类.

这个需要改变了. 把数据表现为类和对象有很大的好处, 所以我们现在要开始进入到 transaction 级别建模了. 不过我们先来探讨一下另外一个 OOP 概念: 深度操作.

第二十章

类层级和深度操作

生活中有一些简单粗暴的规则，包括“不要吃比你的头还大的东西”，“不要吃黄色的雪”还有“不要把一样的代码拷很多份。”

另外一种说法，“如果你发现你拷贝了代码然后做了很小的改动，你就做错了。”

我们可以把 OOP 看成是致力于阻止人们拷贝代码的一种成果。我们把拷代码来重用换成继承类(就像我们从 `uvm_component` 继承)或者创建对象的实例(就像我们创建 `uvm_put_port`)。

即使有了这些工具，我们还是要小心不要破坏了使用其他人的代码的功能。这个在深度操作里特别重要。我们看看拷贝。

还是回到狮子的例子

狮子回来了，这次它们要来帮我们理解深度操作。假如现在有这样的类层级：

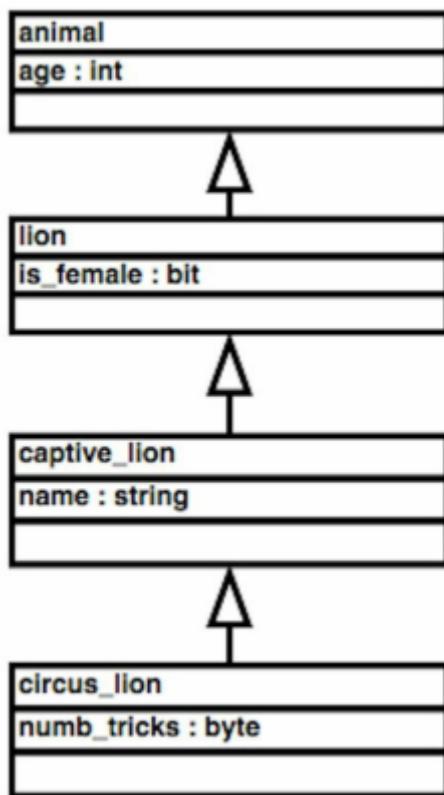


图 141. lion 类层级

这个层级从最通用(`animal`)一直演化到最个别(`circus_lion`)。每个类都在本层加入了一点合理的细节。我们想要使用全部的信息的时候，挑战就来了。

考虑一个 `convert2string()` 方法，我们会在本书的剩下部分多次使用这个方便的方法。这个方法使用对象中的值然后转换成为适合打印的字符串。

这个简单的函数显示了我们会轻易掉进拷贝代码的陷阱。这里是 animal 类中的 convert2string()方法：

```
8     virtual function string convert2string();
9         return $sformatf("Age: %0d", age);
10    endfunction : convert2string
```

图 142. 黑科技之化 animal 为字符串

animal 类只有一个变量，age，所有我们把它转换成字符串并返回。目前为止还不错，不过我要把 lion 转换成 string 要怎么做呢？下面是一种方法：

```
27     function string convert2string();
28         string gender_s;
29         gender_s = (is_female) ? "Female" : "Male";
30         return sformat("Age: %0d Gender: ", age, gender_s);
31     endfunction : convert2string
```

图 143. 在 lion 里先行设置性别

convert2string()方法为我们把 is_female 为转换成"Female"或"Male"。然后它还打印了年龄。看起来可以。到 captive_lion 的时候怎么样呢？

```
52     function string convert2string();
53         return {$sformatf("age: %0d is_female: %0b name: %s",
54             age, is_female, name) };
55     endfunction : convert2string
```

图 144. 拷贝代码的弊端

这里有点尴尬了。定义 captive_lion 类的人选择不直接给我们好用的"Female"和"Male"，我们只得到了一个位。现在我们有了跟父类动作不一致的继承类。

问题在于我们在类与类之间拷贝代码，每份拷贝的代码都有可能会引入错误或者不一致。不过一个更严重的问题要显露了。

考虑一下如果我们像这样改变类层级会发生什么：

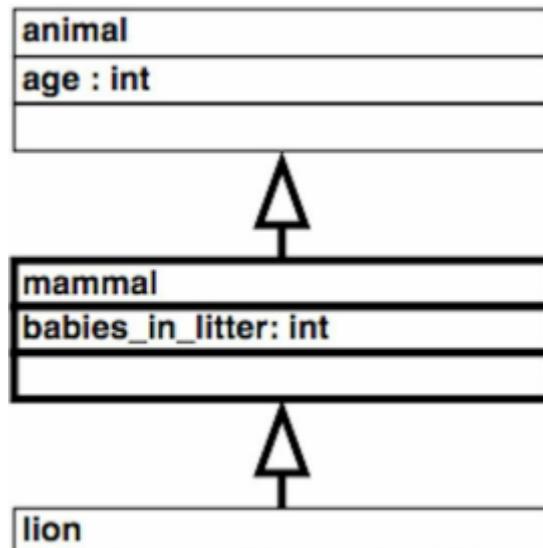


图 145. 被改变的 lion 层级

有人引入了带有一个新变量的另外一个层级。captive_lion 的 convert2string()方法

现在崩溃了，它未能打印所有的变量。我们的代码里出现了 bug。

看起来不公平，不过我们拷了代码，出这个 bug 是活该。唉~

我们本来应该怎么做呢？我们本来应该用一个深度方法来写我们的 `convert2string()` 方法，然后让每个类都只关心自身的变量。用 `super` 关键字来引用父类的变量和方法，而不是在本类里瞎搞可以做到这个。我们来用深度方法重新写 `convert2string()` 方法。

`animal` 的 `convert2string()` 不用变，在 `mammal` 的 `convert2string()` 里做些其他的：

```
function string convert2string();
    return {super.convert2string(),
        $sformatf("\nbabies in litter: %0d",babies_in_litter)};
endfunction : convert2string
```

图 146. 字符串转换的深度策略

现在我们不用关心 `animal` 里有什么了，因为 `super.convert2string()` 处理了。我只用关心 `mammal` 的 `babies_in_litter` 变量。这条链延伸到 `lion` 类：

```
49     function string convert2string();
50         string gender_s;
51         gender_s = (is_female) ? "Female" : "Male";
52         return {super.convert2string(), "\nGender: ",gender_s};
53     endfunction : convert2string
```

图 147. 使用 super 打印 mammal 和 animal 信息

`super` 的调用调用了 `mammal` 中的 `convert2string()` 方法，然后又调用了 `animal` 中的 `convert2string()` 方法。这个代码一直沿着类层级向上来创建字符串。得到的好处就是，继承 `lion` 类的人可以自动的获得我们在这个方法里做的位到字符串转换。

这里是结果：

```
112     initial begin
113         circus_lion1_h = new(.age(2), .is_female(1),
114                             .babies_in_litter(2), .name("Agnes"),
115                             .numb_tricks(2));
116         $display("\n--- Lion 1 ---\n",circus_lion1_h.convert2string());
117         circus_lion2_h = new();
118
119
120
121
122
123
124     # --- Lion 1 ---
125     # Age: 2
126     # babies in litter: 2
127     # Gender: Female
128     # Name: Agnes
129     # numb_tricks: 2
```

图 148. 打印马戏团狮子

由于 `convert2string()` 方法，任何 `lion` 层中的类都是可以正常打印信息的。

深度拷贝

拷贝对象没有你想象的这么简单。`obj2_h = obj1_h` 这样简单的赋值并没有为 `obj2_h` 创建

一个新的对象拷贝, 只是这两个句柄共享了同样的对象. **如果你用一个句柄改变了对象, 另外一个句柄指向的对象也改变了, 因为它们指向同个对象.**

为了创建真正的对象拷贝, 你需要例化一个新的对象然后像下面这样把一个类的数据拷贝到另一个类:

```
112 initial begin
113     circus_lion1_h = new(.age(2), .is_female(1),
114                         .babies_in_litter(2), .name("Agnus"),
115                         .numb_tricks(2));
116     $display("\n--- Lion 1 ---\n",circus_lion1_h.convert2string());
117     circus_lion2_h = new();
118     $display("\n--- Lion 2 before copy ---\n",
119             circus_lion2_h.convert2string());
120     circus_lion2_h.do_copy(circus_lion1_h);
121     $display("\n--- Lion 2 after copy ---\n",
122             circus_lion2_h.convert2string());
123 end
```

图 149. 复制狮子

上面的代码中我们创建了 `circus_lion1_h` 然后传入数据. 接着我们创建了 `circus_lion2_h`, 最后用 `do_copy()` 方法把数据从 `circus_lion1_h` 拷贝到 `circus_lion2_h`.

`do_copy()`方法遭受着跟 `convert2string()`方法一样的挑战. 每个类只能拷贝本类中定义的变量. 必须依赖于 `super.do_copy()`来拷贝父类变量.

创建 `do_copy()`方法的难度在于, 层级中的每个类必须调用其上的类, 所以所有的 `do_copy()`方法需要传入同样类型的参数. 不过根据定义, `circus_lion` 的 `do_copy()`需要一个 `circus_lion` 变量输入, `captive_lion` 类 `do_copy()`需要一个 `captive_lion` 变量输入.

这题好难.

我们可以用多态来解决这个问题. 这个类族中的 `do_copy()`方法传入的是派生的 `animal` 类, 我们要利用好这个, 然后写出的 `circus_lion` 和 `captive_lion` 的 `do_copy` 方法是这样的:

```

64 class captive_lion extends lion;
65     string name;
66
67
68
69
70
71
72
73
74
75
76     function void do_copy(animal copied_animal);
77         captive_lion copied_captive_lion;
78         super.do_copy(copied_animal);
79         $cast(copied_captive_lion, copied_animal);
80         this.name = copied_captive_lion.name;
81     endfunction : do_copy
82 endclass : captive_lion
83
84 class circus_lion extends captive_lion;
85     byte numb_tricks;
86
87
88
89
90
91
92
93
94
95
96
97
98     function void do_copy(animal copied_animal);
99         circus_lion copied_circus_lion;
100        super.do_copy(copied_animal);
101        $cast(copied_circus_lion, copied_animal);
102        this.numb_tricks = copied_circus_lion.numb_tricks;
103    endfunction : do_copy
104 endclass : circus_lion

```

图 150. 使用多态实现深拷贝

`circus_lion` 和 `captive_lion` 的 `do_copy` 方法都用了 `animal(copied_animal)` 作为参数, 里面也声明了各自类型的变量(`copied_captive_lion` 和 `copied_circus_lion`.)

两个方法的工作方式是一样的, 马上把 `copied_animal` 传入父类的 `do_copy()` 方法. 因为所有的 `do_copy()` 方法都接收 `animal`, 所以这是可行的.

调用父类方法后, 再把 `copied_animal` 转换成各自类型的变量. 所以 `circus_lion` 的 `do_copy()` 把 `copied_animal` 转换成 `copied_circus_lion`, 而 `captive_lion` 的 `do_copy()` 把 `copied_animal` 转换成 `copied_captive_lion`.

这是可行的, 因为 `circus_lion` 属于 `captive_lion`, 所以 `circus_lion` 可以存到 `captive_lion` 变量里.

在 `do_copy()` 把 `animal` 转换成合适的 `lion` 之后, 它从拷贝的 `lion` 里的数据拷贝到 `this` 对象. 只要保持 `do_copy()` 的这个形式, 我们就可以不断的在这个类层级里加入层级.

最后的结果:

```
24 # --- Lion 1 ---
25 # Age: 2
26 # babies in litter: 2
27 # Gender: Female
28 # Name: Agnus
29 # numb_tricks: 2
30 #
31 # --- Lion 2 before copy ---
32 # Age: 0
33 # babies in litter: 0
34 # Gender: Male
35 # Name:
36 # numb_tricks: 0
37 #
38 # --- Lion 2 after copy ---
39 # Age: 2
40 # babies in litter: 2
41 # Gender: Female
42 # Name: Agnus
43 # numb_tricks: 2
```

图 151. 复制出来的狮子

我们打印出了拷贝前和拷贝后的 Lion 2，因为我们要验证工程师所以我们要确保拷贝确实完成了。

总结

本章我们学习了深度操作在类层级中的实现。我们了解了在类之间拷贝代码会很容易引入 bug，然后了解了很多用 super 来实现深度操作的方法。

我们在创建接收整个类族中相同的参数的 do_copy() 方法时，也发现了多态的另外一个用法。

在下一章，我们会看到 UVM 用深度操作的用法，用来移动验证平台中 UVM transaction 对象中的数据。

第二十一章

UVM Transaction

在整篇这本入门教程, 我把 TinyALU 验证平台拆分成小块来提升它的可维护性和灵活性. 每个类完成一个任务, 通过数据的传递来完成工作, 所以易于理解类, 发现问题.

通过为类提供本类的方法并防止类暴露各自的工作细节, 我们完成了模块化任务.

这些都是很好的工作, 除了一件事情. 我们还没有在数据上应用 OOP 的好处. 我们还在用 command_s 结构体向验证平台发送指令, 然后用 shortint 类型读取结果. 是时候做出改变了, 用了对象之后会好很多的.

用对象存放数据的益处

你可以(也很有可能)争论说把数据存进对象里没有意义. command_s 结构体已经模块化的很好了, 而且 shortint 也只是个数字, 为什么要把这些弄乱呢?

答案还是: 灵活性, 可维护性和重用性. 假如需要打印指令数据, 任何想要用 command_s 结构体的类都需要知道它包含了三个域: A, B 和 op, 然后 A 和 B 是 byte unsigned 类型, op 是 operation_t 类型.

即使我们的验证平台的 tester, driver, coverage 和 scoreboard 都用了它, 这意味着我们的代码有了软肋. command_s 结构体里的改动可能会需要整个验证平台随之改动.

而且, 记住 TinyALU 指令跟其他的设计相关. 一个以太网指令结构可能会更复杂, 如果你想把你的验证平台复用到另外的以太网验证平台, 你会有很多的工作要做.

你可能会争论说在类里存放指令数据跟在结构体里存放是一样的, 都有数据域和封装. 然而类和例化的对象相对有两个结构体没有的好处:

类有向用户隐藏的跟数据交互的方法. 我们用句柄来操作对象, 我们可以在验证平台里传递这些句柄, 因此很多对象都可以共享代码. 这在我们有像是视频帧这样的大数据块的时候是很重要的.

我们用 UVM 类库来兑现这些优势, 来获得搭建更好验证平台的能力.

transaction 启动

是时候停止用"存储数据的类和对象"这种言论了, 开始用业界术语 transaction 吧. 当我们定义一个存储数据的类时, 我们就是在"定义 transaction", 当我们传送存储数据的对象, 我们是在"传送 transaction."

说实话, "transaction"是数字验证词典中的一个祸害(其他的还有"scoreboard"和"virtual"). 向你保证, 读了我对"transaction"的定义会有人说, "这不是 transaction, transaction 是..." 然后它们会说出一些稍微有些区别的定义. 我说让他们玩去吧, 我们继续.

Transaction 封装了数据和所有对数据的操作. 这些操作包括:

- 提供数据的字符串值(convert2string)
- 拷贝 transaction(do_copy)

- 比较 transaction(do_compare)
- 随机化数据域(用 SystemVerilog 内建的 randomize()方法)

把所有的数据封装进 transaction 让验证平台的其他部分变的简单了. 比如, tester 无需再挑出合法的值来驱动验证平台了, 只需要简单的让 transaction 自行随机化.

定义 transaction

我们通过继承 uvm_transaction 基类然后实现下列方法来定义 transaction:

- do_copy()
- do_compare()
- convert2string()

我们将通过把旧的 command_s 结构体转换成 command_transaction 类来遍历这几个步骤. 先从数据域开始.

创建随机化数据域

在我们的第一代验证平台里, 我们创建了两个测试函数, get_op()和 get_data(). 我们需要这些函数来得到合理的随机输入数据分布. get_op()函数利用了指令总线有 8 种可能值, 其中 6 个是合法指令这个事实. get_data()函数给出了各 1/3 的概率得到全 1, 全 0 和随机的 A, B 总线. 如果没有这个的话, 我们只有 1/256 的概率得到全 0, 这会让我们的 coverage 目标很难达到.

实际上, 这些函数都是不必要的, 因为 SystemVerilog 在我们用 rand 关键字前置声明变量的时候会处理好这些随机.

这里是 command_transaction 类的顶部:

```

1 class command_transaction extends uvm_transaction;
2   `uvm_object_utils(command_transaction)
3   rand byte unsigned      A;
4   rand byte unsigned      B;
5   rand operation_t        op;
6
7   constraint data { A dist {8'h00:=1, [8'h01 : 8'hFE]:=1, 8'hFF:=1};
8     B dist {8'h00:=1, [8'h01 : 8'hFE]:=1, 8'hFF:=1}; }
9

```

图 152.

我们的 transaction 使用 rand 关键字声明了 A, B 和 op 随机数据成员. 所有 SystemVerilog 类都隐式提供了 randomize()方法来从类的随机变量里选择随机值.

SystemVerilog 让我们得以摆脱 get_op()和 get_data()函数. 我们不用再需要 get_op()了, 因为 SystemVerilog 会把 op 变量的值设置为 6 个枚举值中的一个, 每个都是合法的.

我们用 SystemVerilog 的约束来替换 get_data()函数. 这个 constraint 表达式给了我们均等的概率得到全零, 全 1 和之间的随机数.

uvm_object 构造器

uvm_transaction 类派生自 uvm_object 而不是 uvm_component. 所有它的构造器更简单. 由于 UVM object 不在 UVM 层级中, 构造器就不需要 parent 句柄, 只需要一个名字.

所以我们的 uvm_transaction 构造器是这样的:

```
65     function new (string name = "");
66         super.new(name);
67     endfunction : new
68
69 endclass : command_transaction
```

图 153. 简单的 uvm object 构造器

为对象设置名字是好习惯, 不过没有的话也没关系.

do_copy()方法

uvm_object 类和所有它的后继都有 copy()方法来拷贝同类型的对象数据到另外一个对象, 还有 clone()方法来返回带相同数据的对象的新实例.

这些方法只有在我们重载了 do_copy()的时候有效. do_copy()方法跟之前狮子的方法是一样的, 不同之处在于 UVM 要求我们把参数命名为 rhs¹, 就是"右手边", 表现了类似 lhs=rhs 这种赋值语句中的数据位置:

```
function void do_copy(uvm_object rhs);
    command_transaction copied_transaction_h;

    if(rhs == null)
        `uvm_fatal("COMMAND TRANSACTION", "Tried to copy from a null pointer")

    if(!$cast(copied_transaction_h,rhs))
        `uvm_fatal("COMMAND TRANSACTION", "Tried to copy wrong type.")

    super.do_copy(rhs); // copy all parent class data

    A = copied_transaction_h.A;
    B = copied_transaction_h.B;
    op = copied_transaction_h.op;

endfunction : do_copy
```

图 154. 在 transaction 里实现 do_copy()

一开始我们要检查是否传入了空句柄, 如果是的话就要发送 fatal 信息然后退出.

接下来我们检查 rhs 变量的对象类型是否跟我们一致, 将 rhs 转换到 copied_transaction_h. 如果这个转换返回 0, 意味着我们把其他类型对象传进来了, 这个不能拷贝的, 所以发送 fatal 退出. 这些检查可以简化 debug.

¹ 译注: UVM 其实并没有这个要求, do_copy 作为一个普通的方法, 参数的名字可以是任意的, 只是 UVM 源码里给的参考实现里的参数名是 rhs 而已.

现在我们知道传入的对象类型是正确的, 然后调用 `super.do_copy` 传递对象到父类拷贝父类的数据. 完成之后, 我们从 `copied_transaction_h` 对象拷贝 A, B 和 op 数据成员.

clone_me()方法和 MOOCOW

用 `transaction` 的一大好处是, 所有拥有同样对象句柄的对象都可以得到相同的数据. 这个在 TinyALU 看起来没什么, 不过当你设计的是一个大型的 250 端口的交换机, 你不会想着去把数据拷贝 250 份的.

然而, 这个用多个句柄来共享单个对象的想法, 只适用于你的团队严格的实现了 MOOCOW(Mandatory Obligatory Object Copy On Write 对象写时强制拷贝)的时候. MOOCOW 意味着你只有在不去改动对象的数据的时候才共享对象的句柄(不然改动了队友就不知晓了). 一旦你想对对象执行写操作, 你必须严格的手动拷贝一份这个对象, 这就是 Manual Obligatory Object Copy On Write(另一种 MOOCOW)..

UVM 用 `clone()` 方法来支持 MOOCOW. `clone()` 方法借用你的 `do_copy` 方法来创建第二个本对象的拷贝. `clone()` 方法返回的是 `uvm_object`, 所以你需要把返回的对象转换成你的目标类.

这造成了每个需要克隆这个对象的人都需要提供自己的 `$cast` 调用, 所以我会创建一个好用的 `clone_me()` 来拷贝对象并转换:

```
29      function command_transaction clone_me();
30          command_transaction clone;
31          uvm_object tmp;
32
33          tmp = this.clone();
34          $cast(clone, tmp);
35          return clone;
36      endfunction : clone_me
```

图 155. `clone_me()` 方法

这个方法简单的克隆了当前的对象然后转换成当前对象类型, 最后返回对象. 我们不需要在 TinyALU 验证平台里用这个方法, 因为这里我们不需要关心 MOOCOW.

do_compare()方法

在验证平台里我们会对比大量的数据, 所以 `uvm_transaction` 之间可以进行对比是合理的. UVM 提供了 `compare()` 方法来做这个(结果相同的话返回 1).

我们通过提供 `do_compare()` 方法来实现 `uvm_transcation` 的 `compare()` 方法. `do_compare()` 方法用一样的方式比较 `transaction`. 这是个深度比较, 所以我们要把参数传入父类进行比较再把结果连到自身的比较.

下面是 `command_transaction` 的 `do_compare()` 方法:

```

function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    command_transaction compared_transaction_h;
    bit same;

    if (rhs==null) `uvm_fatal("RANDOM TRANSACTION",
                           "Tried to do comparison to a null pointer");

    if (!$cast(compared_transaction_h,rhs))
        same = 0;
    else
        same = super.do_compare(rhs, comparer) &&
               (compared_transaction_h.A == A) &&
               (compared_transaction_h.B == B) &&
               (compared_transaction_h.op == op);

    return same;
endfunction : do_compare

```

图 156. uvm 对比操作

`do_compare()`方法有两个参数。第一个 `rhs` 是用来做比较的对象；第二个叫做 `uvm_comparer`，这个是本书之外的内容了，我们只要简单的把这个跟 `rhs` 在调用 `super.do_compare()` 的时候一起传入父类就好了。

`transaction` 相同的话，`same` 位置 1. 我们先检查接收的是否为空句柄，这是个致命的错误。接着，我们把 `rhs` 转换成 `command_transaction` 类型的 `compared_transaction_h` 变量。这是第一级的比较，如果转换失败了说明 `transaction` 是不一致的，我们为 `same` 置 0。对象中三个数据域和被测试对象中的相同而且 `super.do_compare()` 返回 1 的话，我们就把 `same` 位置为 1。
最后我们返回 `same` 位。

convert2string()方法

像之前的狮子类一样，我们的 `transaction` 需要 `convert2string()` 方法才好用来打印：

```

58     function string convert2string();
59         string s;
60         s = $sformatf("A: %2h B: %2h op: %s",
61                         A, B, op.name());
62         return s;
63     endfunction : convert2string

```

图 157. 将 command transaction 转换为字符串

`$sformatf()` 方法用了 SystemVerilog 的格式配置来把我们的数据转换成字符串。枚举类型的值有 `name()` 方法可以返回值的字符串，像“`add_op`”和“`no_op`”这样的。

使用 transaction

向基于 `transaction` 数据的迁移需要我们改动几乎每个 TinyALU 验证平台的部分。不过这些

都是让我们的代码更简单更易于重用的修改.

以下是改动清单:

- 第1步: 创建 result_transaction 类操作结果.
- 第2步: 创建 add_transaction 类继承 command_transaction 来生成加法指令.
- 第3步: 将 base_tester 命名为 tester, 因为现在可以用来做任意的 Test 了.
- 第4步: 修改 command_monitor 来创建 command_transaction
- 第5步: 修改 result_monitor 来创建 result_transaction.
- 第6步: 修改 scoreboard 来使用 result_transaction 的 compare()方法.
- 第7步: 修改 add_test 来使用 add_transaction.

显然, 如果我们从一开始就用 transaction 来设计我们的验证平台就容易了. 不过这些是简单的修改. 我们一步一步来吧.

第1步: 创建 result_transaction 类操作结果.

result_transaction 类跟 command_transaction 类相似, 不过它只有一个数据元素:

```
1 class result_transaction extends uvm_transaction;
2   `uvm_object_utils(result_transaction)
3
4   shortint result;
5
6   function bit do_compare(uvm_object rhs, uvm_comparer comparer);
7     result_transaction tested;
8     bit same;
9
10    if (!$cast(tested,rhs))
```

图 158. result transaction 类

scoreboard 会用 do_compare()方法来对比预期结果和实际结果.

第 2 步 : 从 command_transaction 继承 创建 add_transaction 类来生成加法指令.

add_transaction 派生自 command_transaction, 提供了一个随机约束:

```
1 class add_transaction extends command_transaction;
2   `uvm_object_utils(add_transaction)
3
4   constraint add_only {op == add_op;}
5
6   function new(string name="");super.new(name);endfunction
7 endclass : add_transaction
```

图 159. add transaction

add_transaction 让我们得以在不改变 tester_h 对象的情况下生成不同的激励. tester_h 对象会用同样的方式来使用 add_transaction, 不过收到的只有加法指令.

第3步: 将base_tester命名为tester, 因为现在可以用来做任意的 Test 了.

我们之前的设计中有两个 tester 类, 实现了两种测试用例. base_tester 生成了随机激励, add_tester 生成加法指令和随机操作数. 现在我们有了一个 add_transaction 来把两个类压缩成一个. tester 类用 randomize()方法来生成指令:

```
23     command = command_transaction::type_id::create("command");
24     repeat (10) begin
25       assert(command.randomize());
26       command_port.put(command);
27     end
```

图 160. tester 类的

这个代码有两个关键点. 第一是我们用 Factory 来创建 command_transaction, 所以这个 transaction 的类型是可以重载的. 第二点是我们随机化了这个 transaction, 约束控制了这个结果. 当我们用 add_transaction 重载 command_transaction 时, 我们会只得到加法指令.

第 4 步 : 修 改 command_monitor 来 创 建 command_transaction

command_monitor 从 BFM 中取得数据, 进行封装然后发送到验证平台. 现在它发送的是 command_transaction:

```
function void write_to_monitor(byte A, byte B, operation_t op);
  command_transaction cmd;
  `uvm_info("COMMAND MONITOR",$sformatf("MONITOR: A: %2h B: %2h op: %s",
    A, B, op.name()), UVM_HIGH);
  cmd = new("cmd");
  cmd.A = A;
  cmd.B = B;
  cmd.op = op;
  ap.write(cmd);
endfunction : write_to_monitor
endclass : command_monitor
```

图 161. 监测 command

这个代码跟之前的是类似的, 不过现在发送的是一个对象.

第 5 步: 修改 result_monitor 来 创建 result_transaction.

就是一个简化版本的 command monitor:

```

17     function void write_to_monitor(shortint r);
18         result_transaction result_t;
19         result_t = new("result_t");
20         result_t.result = r;
21         ap.write(result_t);
22     endfunction : write_to_monitor

```

图 162. 用 result transaction 监测结果

我们把从 BFM 得到的数据存放进 result_transaction 里然后发送这个 transaction 到验证平台.

第 6 步: 修改 scoreboard 来使用 result_transaction 的 compare()方法.

transaction 级的仿真简化了预期结果和实际结果的比较. result monitor 和预测器都创建了 result_transaction 对象. 如果这些是相同的, 那结果就是正确的:

```

38     do
39         if (!cmd_f.try_get(cmd))
40             $fatal(1, "Missing command in self checker");
41         while ((cmd.op == no_op) || (cmd.op == rst_op));
42
43         predicted = predict_result(cmd);
44
45
46         if (!predicted.compare(t))
47             `uvm_error("SELF CHECKER", {"FAIL: ",data_str})
48         else
49             `uvm_info ("SELF CHECKER", {"PASS: ", data_str}, UVM_HIGH)
50
51         endfunction : write
52     endclass : scoreboard

```

图 163. transaction 级的 scoreboard

result_monitor 给我们传递了实际结果 t, 同时我们从 command_monitor 里得到对应的指令, 然后用 predict_result()方法来创建预期的 result_transaction.

我们用 compare()来检查结果是否正确. scoreboard 现在简单很多了.

第 7 步: 修改 add_test 来使用 add_transaction.

add_test 和 random_test 是一样的, 除了我们用 add_transaction 来重载了 command_transaction:

```
class add_test extends random_test;
  `uvm_component_utils(add_test);

function void build_phase(uvm_phase phase);
  tinyalu_transaction::type_id::set_type_override(add_transaction::get_type());
  super.build_phase(phase);
endfunction : build_phase
```

图 164. 用 add transaction 来重载 command transaction

这个重载使得 tester 创建了 add_transaction 而不是 command_transaction, 不需要修改 tester 代码了.

总结

本章中, 我们用 uvm_transaction 在验证平台中传送数据. 这让我们得以简化 Test 组件, 甚至去掉了验证平台中的类(add_test).

下一章, 我们会基于重用的需求再一次转向 UVM 层级. 我们会意识到由给定的模块或接口联系到一起的类都是可以组合到一起, 在其他使用这个接口的验证平台中就可以重用了. 我们要接着来学习 uvm_agent.

第二十二章

UVM Agent

模块化是个好东西, 无论你是用来封装硬件中的模块, 固件中的子程序还是软件中的对象, 模块化隐藏了复杂实现, 让我们得以复用测试过的硬件 IP, 固件的库还有软件中的类库.

模块化成功使用的关键在于提供一个严格固定的接口. 定义保存接口使得用户(以及未来的我们自己)可以放心的使用一个模块组件而不用担心它内在的运作过程.

UVM 支持的模块化包括 agent, configuration object 和 configuration database 这些概念. Agent 允许我们封装相似的对象, configuration object 允许我们存放 agent 需要的信息, configuration database 允许我们在通过 agent 的多个实例间传送配置信息.

考虑下我们的验证平台:

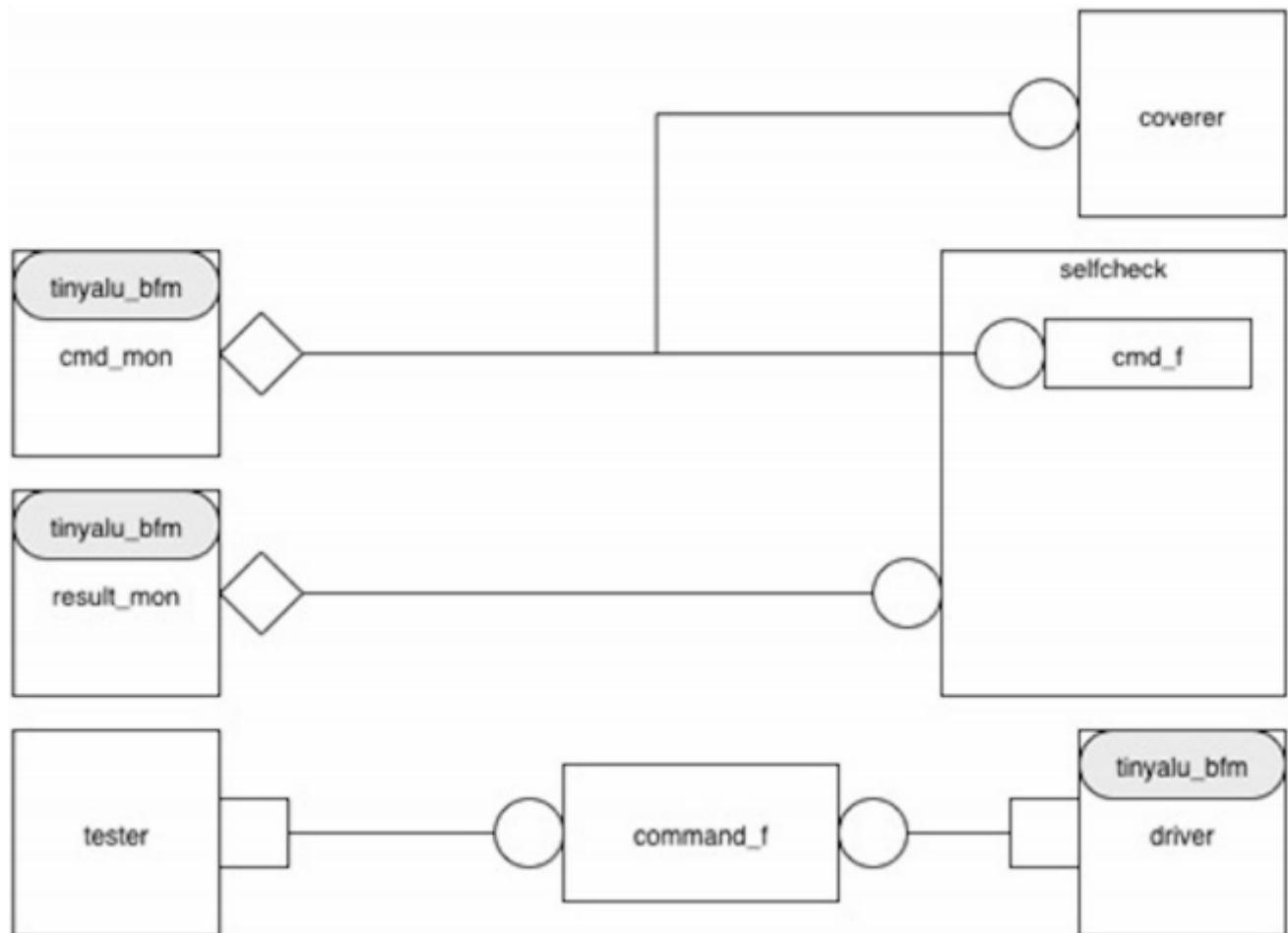


图 165. 当前的 TinyALU 验证平台

这个验证平台生成激励, 检查结果, 然后监控覆盖率. 不过这个是很水的. 假如我们在设计里有多个 TinyALU 的话, 我们得拷贝这些对象和连接关系来复用它们, 这可不是好办法.

我们识别出验证平台中所有监控或驱动 TinyALU 的对象来解决这个问题. 把这些 TinyALU 相关的对象封装到一个 uvm_agent 类里, 创建 TinyALU agent 会更好.

下面是我们的 agent:

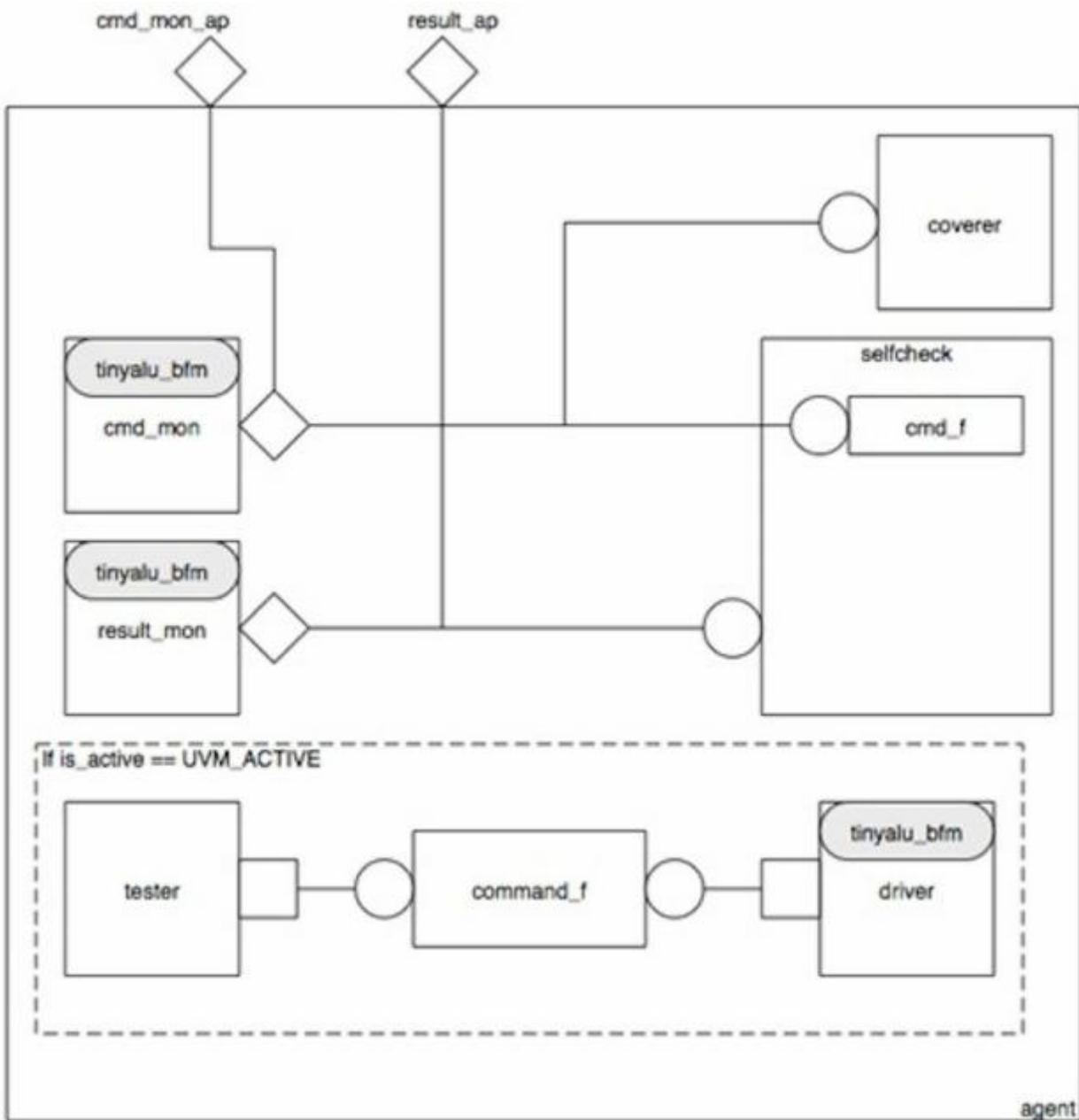


图 166. TinyALU agent

我们用的是一样的对象，把它们封装进另外一种 `uvm_component` 类型的 `uvm_agent` 里。这些验证平台中的对象是一样的，连接方式也是一样的。这样需要处理另外一个 TinyALU 的时候，我们就不用每次都重新做这个连接了。

我们还创建了一对顶层 `analysis port` 来让更高层级的组件来监测 agent 中的信息流。同时，我们的 `scoreboard_h` 和 `coverage_h` 的订阅是一样的。

`tester_h`, `command_f` 和 `driver_h` 对象周围的虚线表示这些只在用户需要产生激励的时候例化。没有的这些对象的话，你还是可以用来监控 TinyALU。

从外面看我们的 TinyALU agent 是这样的：

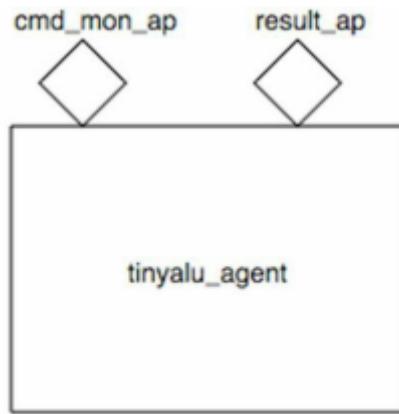


图 167. TinyALU agent 隐藏的细节

编写 TinyALU agent

`tinyalu_agent` 类看起来跟前面章节的 `env` 很像, 不同之处在于我们引入了配置类来传递 BFM, 控制 agent 是否产生激励.

TinyALU agent 包含了之前验证平台中所有的组件和 FIFO. 另外还有一个配置类:

```

1  class tinyalu_agent extends uvm_agent;
2      `uvm_component_utils(tinyalu_agent)
3
4      tinyalu_agent_config tinyalu_agent_config_h;
5
6      tester          tester_h;
7      driver          driver_h;
8      scoreboard      scoreboard_h;
9      coverage         coverage_h;
10     command_monitor command_monitor_h;
11     result_monitor   result_monitor_h;
12
13     uvm_tlm_fifo    #(command_transaction) command_f;
14     uvm_analysis_port #(command_transaction) cmd_mon_ap;
15     uvm_analysis_port #(result_transaction) result_ap;

```

图 168. agent 中的声明

`uvm_agent` 提供了 `is_active` 数据成员和 `get_is_active()` 方法. `is_active` 数据成员是 `uvm_active_passive_enum` 类型的变量. 这个枚举类型有两个值, `UVM_ACTIVE` 和 `UVM_PASSIVE`, 默认值是 `UVM_ACTIVE`.

由于我们会例化多个同样的 agent, 这样就需要把 TinyALU 的 BFM 的句柄和 `is_active` 值传入 agent. 我们用 `tinyalu_agent_config` 类来处理:

```

1 class tinyalu_agent_config;
2
3     virtual interface tinyalu_bfm bfm;
4         protected uvm_active_passive_enum      is_active;
5
6     function new (virtual tinyalu_bfm bfm, uvm_active_passive_enum
7                 is_active);
8         this.bfm = bfm;
9         this.is_active = is_active;
10    endfunction : new
11
12    function uvm_active_passive_enum get_is_active();
13        return is_active;
14    endfunction : get_is_active
15
16 endclass : tinyalu_agent_config

```

图 169. TinyALU agent

TinyALU agent 在 build_phase()方法里获取这个配置对象:

```

function void build_phase(uvm_phase phase);

    if(!uvm_config_db #(tinyalu_agent_config)::get(this, "", "config",
                                                    tinyalu_agent_config_h))
        `uvm_fatal("AGENT", "Failed to get config object");
    is_active = tinyalu_agent_config_h.get_is_active();

    if (get_is_active() == UVM_ACTIVE) begin : make_stimulus
        command_f = new("command_f", this);
        tester_h   = tester::type_id::create( "tester_h",this);
        driver_h   = driver::type_id::create("driver_h",this);
    end

    command_monitor_h = command_monitor::type_id::create("command_monitor_h",this);
    result_monitor_h = result_monitor::type_id::create("result_monitor_h",this);

    coverage_h = coverage::type_id::create("coverage_h",this);
    scoreboard_h = scoreboard::type_id::create("scoreboard_h",this);

    cmd_mon_ap = new("cmd_mon_ap",this);
    result_ap  = new("result_ap", this);

endfunction : build_phase

```

图 170. TinyALU agent 构造器

这里 uvm_config_db 给我们传递了 tinyalu_agent_config 对象的句柄而不是虚拟接口的句柄。我们从配置对象里得到 is_active 值。根据这个名字的暗示，UVM 设计者期望 uvm_config_db 在整个验证平台里传送配置对象，uvm_config_db 通常是传送配置对象而不是 BFM 的。

这里在创建一个类的多个对象时有一个问题。如果我用"config"字符串从配置对象里得到 uvm_config_db，那怎么和其他用"config"字符串的对象区分开来呢？后面我们会看到 uvm_config_db 是解决了这个问题的。

有了配置数据之后，我们创建了 agent。创建 agent 跟之前创建 env 很像，除了我们根据 is_active 变量的值进行选择性的例化。

例化了 TinyALU agent 的任何人都得到了驱动 TinyALU, 监控结果, 计算覆盖率的所有工具. 接着我们来连接这些对象:

```
47 function void connect_phase(uvm_phase phase);
48     if (get_is_active() == UVM_ACTIVE) begin : make_stimulus
49         driver_h.command_port.connect(command_f.get_export);
50         tester_h.command_port.connect(command_f.put_export);
51     end
52
53     command_monitor_h.ap.connect(cmd_mon_ap);
54     result_monitor_h.ap.connect(result_ap);
55
56     command_monitor_h.ap.connect(scoreboard_h.cmd_f.analysis_export);
57     command_monitor_h.ap.connect(coverage_h.analysis_export);
58     result_monitor_h.ap.connect(scoreboard_h.analysis_export);
59
60 endfunction : connect_phase
```

图 171. TinyALU connect phase

这个方法连接了这些对象, 然后像我们看到的把 analysis port(command_ap 和 result_ap)延伸到了顶层. 上面带阴影的行通过调用较底层的 connect()方法连接底层的 analysis port(如 command_monitor_h.ap)到顶层的 analysis port(像 cmd_mon_ap).

使用 UVM Agent

可以的话想像一下这个场景, 你的老板发现你把 tester module 替换成了 tester UVM 组件, 他很不高兴. 原来是他写的 tester module, 然后觉得自己很厉害, 特别是那个 get_op()方法, 简直是一直在说这个说个不停.

你争论说新的系统运行的很好, 甚至你都不需要 get_op()方法(你真是哪壶不开提哪壶!). 现在他要你同时运行两个 TinyALU 仿真看看哪个激励运行的好一点. 你一边安抚他一边在更新简历...

顶层 Module

为了实现这个任务, 我们例化了两个 TinyALU module 和两个 TinyALU BFM. 一个 BFM 我们用 TinyALU agent 来驱动, 另外一个用 tester_module 驱动. 我们会用一个 agent 来监控第二个 TinyALU 的结果.

这里是顶层 module 的例化:

```

module top;
  import uvm_pkg::*;
  import tinyalu_pkg::*;
`include "tinyalu_macros.svh"
`include "uvm_macros.svh"

tinyalu_bfm      class_bfm();
tinyalu class_dut (.A(class_bfm.A), .B(class_bfm.B), .op(class_bfm.op),
                   .clk(class_bfm.clk), .reset_n(class_bfm.reset_n),
                   .start(class_bfm.start), .done(class_bfm.done),
                   .result(class_bfm.result));

tinyalu_bfm      module_bfm();
tinyalu module_dut (.A(module_bfm.A), .B(module_bfm.B), .op(module_bfm.op),
                     .clk(module_bfm.clk), .reset_n(module_bfm.reset_n),
                     .start(module_bfm.start), .done(module_bfm.done),
                     .result(module_bfm.result));

tinyalu_tester_module stim_module(module_bfm);

```

图 172. 两个 TinyALU 顶层

我们有两个 DUT(class_dut 和 module_dut)和两个 tinyalu_bfm 接口(class_bfm 和 module_bfm). 我们还有一个 tinyalu_tester_module 连接到 module_bfm.

接下来我们来连接 BFM 到验证平台.

用户不知道模块和对象里怎么运作靠着接口来使用的这种模块可是最好的. 接口的清晰定义可以让用户从细节实现中解脱出来, 让它们更关注更高层次的思考.

在这个双 TinyALU 验证平台里, 我们不想让用户考虑我们是否使用了 agent 来实现这个测试用例, 我们想让用户用简单的步骤来就可以了.

下面是 TinyALU 验证平台的用户接口:

```

initial begin
  uvm_config_db #(virtual tinyalu_bfm)::set(null, "*", "class_bfm", class_bfm);
  uvm_config_db #(virtual tinyalu_bfm)::set(null, "*", "module_bfm", module_bfm);
  run_test("dual_test");
end

endmodule : top

```

图 173. 验证平台里的实现

这个代码通过 uvm_config_db 来用 class_bfm 传送 class_bfm, 用 module_bfm 来传送 module_bfm. 之后代码调用了 run_test("dual_test"), 剩下的我们处理了. 代码的编写者不需要担心 BFM 传送到哪里了这些细节. 我们处理了这些, 我们会在设计里保持模块化的.

dual_test 类

验证平台的顶层模块存储 class_bfm 和 module_bfm 在 uvm_config_db 里, 没有告知这些接口要怎么在 Test 里面用. dual_test 类的任务就是将这些原始的 BFM 保存到环境的配置

对象里.

在每个层级都使用配置对象的这一个理念使得我们验证平台中类的重用成为可能, 因为配置对象就是一个清晰的接口. 我们例化使用一个配置对象的时候, 就知道使用这个对象的类是开心的.

dual_test 只例化了一个 env 对象. 所有我们只需要例化 env 的配置对象, 然后用 uvm_config_db 传给 env. 这里是 env 的配置类:

```
class env_config;
    virtual tinyalu_bfm class_bfm;
    virtual tinyalu_bfm module_bfm;

    function new(virtual tinyalu_bfm class_bfm, virtual tinyalu_bfm module_bfm);
        this.class_bfm = class_bfm;
        this.module_bfm = module_bfm;
    endfunction : new
endclass : env_config
```

图 174. env 的配置类

env_config 类在验证平台里存储了两个 tinyalu_bfm 接口. 我们确保了用户会在构造器里放入类的数据成员, 如果用户忘了做的话, 仿真器会报出语法错误.

dual_test 测试用例只包含了 build_phase():

```
function void build_phase(uvm_phase phase);
    virtual tinyalu_bfm class_bfm, module_bfm;
    env_config env_config_h;

    if(!uvm_config_db #(virtual tinyalu_bfm)::get(this, "", "class_bfm", class_bfm))
        `uvm_fatal("DUAL TEST", "Failed to get CLASS BFM");
    if(!uvm_config_db #(virtual tinyalu_bfm)::get(this, "", "module_bfm", module_bfm))
        `uvm_fatal("DUAL TEST", "Failed to get MODULE BFM");

    env_config_h = new(.class_bfm(class_bfm), .module_bfm(module_bfm));

    uvm_config_db #(env_config)::set(this, "env_h*", "config", env_config_h);

    env_h = env::type_id::create("env_h",this);
endfunction : build_phase
```

图 175. 使用 config 对象来 build

build phase 从 uvm_config_db 里得到 class_bfm 和 module_bfm 然后用它们来例化 env_config_h 对象. 接着它又把 env_config_h 对象存进 uvm_config_db 里.

这里的 uvm_config_db::set 调用跟之前看到的都不一样. 所有之前的 set() 调用都在 module 的 initial 模块里. 这个调用是在 UVM 的层级里. 这样一来我们就可以用 uvm_config_db 的过滤功能了.

这个调用有两个不同之处. 第一个是我们传入了 this 变量来为存储的信息设置上下文. 接着我们传入字符串指示 uvm_config_db 哪些可以访问这个数据. 这里, 我们限制 env_h 类的可访问者为这个 Test 内例化的对象. 这个在这里是多余的啦, 因为我们只例化了一个对象. 我们会在 env 里用到更好的过滤功能.

TinyALU 环境

在前面的验证平台中, env 类只是简单的用来例化组件. 现在我们有了多个 agent, env 要做更多的工作了; 它要为这些 agent 创建配置对象, 还要把这些配置对象放到 agent 可以取得的地方.

这个类充分运用了 uvm_config_db 里的层级过滤功能. 还记得 tinyalu_agent 类用 config 字符串从数据库里获取配置对象吗? 上次的问题是, 在同个 agent 的多个实例都要用同一个字符串访问数据库的时候要怎么做?

答案就是 uvm_config_db 允许我们在 UVM 层级的不同部分使用不同的配置对象. env 从 uvm_config_db 里得到配置数据, 为每个 tinyalu_agent 实例创建配置对象, 然后存储这些配置对象, 这样每个实例都能访问到正确的对象:

```
if(!uvm_config_db #(env_config)::get(this, "", "config", env_config_h))
  `uvm_fatal("RANDOM TEST", "Failed to get CLASS BFM");

class_config_h = new(.bfm(env_config_h.class_bfm), .is_active(UVM_ACTIVE));
module_config_h = new(.bfm(env_config_h.module_bfm), .is_active(UVM_PASSIVE));

uvm_config_db #(tinyalu_agent_config)::set(this, "class_tinyalu_agent_h*", "config", class_config_h);

uvm_config_db #(tinyalu_agent_config)::set(this, "module_tinyalu_agent_h*", "config", module_config_h);
```

图 176. 在层级中存储配置对象

我们为两个 agent 创建了两个配置对象. class_config_h 持有 class_bfm, 然后把 is_active 设置为 UVM_ACTIVE 因为这个 agent 需要生成激励. module_config_h 对象持有 module_bfm, 然后把 is_active 设置为 UVM_PASSIVE 因为老板的 module 会提供激励.

我们最后用适配上面 set() 调用的名字来例化 agent:

```
32  class_tinyalu_agent_h = new("class_tinyalu_agent_h",this);
33  module_tinyalu_agent_h = new("module_tinyalu_agent_h",this);
```

图 177. 创建适配 config 的 agent

现在我们有了会得到不同 tinyalu_agent_config 对象的 tinyalu_agent 的不同实例了.

总结

本章中我们学习了如果把组件封装进一个 UVM agent 类. 这种封装让我们得以简单的重用验证平台组件, 或为同样的组件创建多个实例. 我们还学习了怎么创建生成激励的主动型 agent 和仅仅监控接口的被动型 agent. 模块化让生活更美好.

如果层级中的每一级都有一个定义清晰的配置对象, 这种对象中的封装行为就是递归的. 我们可以例化两个 env 类来仿真四个 TinyALU, 只用很小的改动.

创建真正可重用的模块化验证平台还剩最后一件事. 你或许注意到 agent 类的一个奇怪的地方. 尽管我们需要通过 agent 来运行多种不同的激励, 不过内置的 tester 值提供了默认的

一种激励. 我们可以修改 `command_transaction` 来稍作控制, 不过这个还是不怎么行.

我们真正需要的是让 `agent` 根据不同的命令来产生不同类型的激励. 我们要把激励从验证平台里剥离. 我们将在下一章的用 UVM sequence 实现.

第二十三章

UVM Sequence

UVM Primer 通篇都在把验证平台的功能拆分来创建更小更简单的设计单元。这个过程的代码是灵活的，随着代码的增长，它会让验证平台更加强大，也可以在未来的验证平台中重用。

前面的 UVM agent 章节中，我们关注于结构。我们把所有与 TinyALU 相关的组件取出来封装到一个我们可以轻易重用的组件。

在 UVM transaction 这章，我们关注于数据。我们创建了类和对象来更方便的生成，对比并传输数据。我把数据类从结构类中分离。

在本章，我们要关注数据和结构之间的终极混合点：测试激励。因为尽管我们已经分离了数据和结构，我们还没有把数据激励从结构中分离出来。这个验证平台设计是不好的，我们的 tester 类就有这个问题。

tester 创建了新的 transaction 并发送到验证平台。这表示 tester 做了两件事：选择 transaction 的顺序，把 transaction 发送到验证平台。这会引起重用的问题，后来的设计者（或是后来的你）可能会认为 tester 是创建新 transaction 的理想方案，但是因为 tester 有着终止激励的副作用，很可能这个 tester 是用不了的。

我们可以重载 transaction 类型来控制数据随机化，但是要改变 transaction 的数量和发送方式的话得重载整个 tester 类。这就像你要开车到不同目的地需要换方向盘。

好的验证平台把 transaction 的序列（测试激励）从验证平台的结构中分离。transaction 的序列改变，结构应该不受影响。我们会展示怎么在 TinyALU 里做到这点，我们将创建三个 Test：

- 斐波那契 Test，用 TinyALU 来计算斐波那契数列
- 完全随机 Test，用带约束随机实现覆盖率目标
- 混合斐波那契 Test 和完全随机 Test 到一个激励流里

这些 Test 表明了我们为何要把激励生成从结构中分离。当我们创建斐波那契 Test 和完全随机 Test 的时候，我们得重新编写包含它们的 tester。如果我们加入更多的测试用例和组合，tester 类会爆炸性的增多。这是很难维护的。

UVM sequence 把激励从验证平台结构中分离出来，让我们得以创建同个验证平台来运行不同的数据，这样为我们的 UVM 旅程画上句号。

我们将通过把当前的 transaction 级验证平台转换成使用 sequence 的验证平台来学习 UVM sequence。这个过程分为 7 步：

- 第一步：创建携带数据的 TinyALU sequence 项目。
- 第二步：把 tester 替换成 uvm_sequencer。
- 第三步：对 driver 进行支持 sequence 的升级。
- 第四步：在环境里例化 driver 和 sequence 然后相互连接。
- 第五步：编写 UVM sequence。
- 第六步：编写用 sequencer 启动 sequence 的 Test。

每一步都是简单的，到我们进行到最后的时候，我们就会有一个完整的灵活的验证平台啦。

第一步：创建携带数据的 TinyALU sequence 项目.

两章之前我们学习了 `uvm_transaction` 类. UVM 从 `uvm_transaction` 继承创建了 `uvm_sequence_item` 类. `uvm_sequence_item` 类携带了 `uvm_sequence` 的从 `uvm_sequence` 到 `uvm_driver` 的数据.

首先, 我们把 `command_transaction` 转换成 `sequence_item`:

```
1 class sequence_item extends uvm_sequence_item;
2   `uvm_object_utils(sequence_item);
3
4   function new(string name = "");
5     super.new(name);
6   endfunction : new
7
8   rand byte unsigned      A;
9   rand byte unsigned      B;
10  rand operation_t        op;
11  shortint unsigned      result;
12
13  constraint op_con {op dist {no_op := 1, add_op := 5, and_op:=5,
14    xor_op:=5,mul_op:=5, rst_op:=1};}
```

图 178. TinyALU sequence item

`sequence_item` 类跟 `command_transaction` 类的区别在以下两点:

- 从 `uvm_sequence_item` 而不是 `uvm_transaction` 继承.
- 加入了 `result` 到 `tinyalu_item`, 原因后面再解释.

剩下的类都是一样的, 然后我们进入第二步.

第二步：把 `tester` 替换成 `uvm_sequencer`.

在之前的例子里, 我们把所有 TinyALU 组件封装进一个 TinyALU agent 组件. 由于本章跟 agent 无关, 我们还是回到所有组件在 env 类里例化的验证平台.

当前版本的 env 例化了 `tester` 类的 `tester_h` 对象. 我们之前说过, `tester` 是有毛病的类. 它做了太多的事情, 是难以维护和重用的. 最坏的情况是我们不得不重写.

我们需要 `tester` 做的只是把 `sequence` 传送到 `driver`, `sequence` 项目的顺序选择由验证平台的其他部分来做. 所以再把这个类叫做 `tester` 是不对的, 我们来用 `sequencer` 这个名字来代替.

`sequencer` 类从 `sequence` 里取出 `sequence_item`, 传给 `driver`. UVM 提供了 `uvm_sequencer` 基类. 我们用下面的方法来使用:

```
13 `include "sequence_item.svh"
14 typedef uvm_sequencer #(sequence_item) sequencer;
15 sequencer sequencer_h;
```

图 179. 声明 sequencer

我在 `tinyalu_pkg` 包里定义所有的类和类型. 上面的代码行中我们定义 `sequence` 为带 `sequence_items` 类型的 `uvm_sequencer`. 用 `typedef` 设定参数类可以简化这个类名在设计

中的后续使用.

第三步: 对 driver 进行支持 sequence 的升级.

在整篇教程中, 我们使用泛指的 "driver" 来指代跟 BFM 交互的对象. 我们从 `uvm_component` 派生出 `driver` 然后在环境里例化.

UVM 有一个比我们更专业的"driver"的概念, 它定义了派生自 `uvm_component` 的跟 `uvm_sequencer` 交互的 `uvm_driver`. 我们通过继承 `uvm_driver` 并改变它的 `run_phase()` 方法来升级 `driver`.

首先我们从 `uvm_driver` 派生, 把工作参数设为 `sequence_item`:

```
class driver extends uvm_driver #(sequence_item);
  `uvm_component_utils(driver)

  virtual tinyalu_bfm bfm;

  function void build_phase(uvm_phase phase);
    if(!uvm_config_db #(virtual tinyalu_bfm)::get(null, "*", "bfm", bfm))
```

图 180. 派生 `uvm_driver` 类

在派生 `uvm_driver` 的时候我们继承了 `seq_item_port` 对象及其所有的功能. 现在我们修改 `run phase` 来使用新的 `seq_item_port`:

```
11   task run_phase(uvm_phase phase);
12     sequence_item cmd;
13
14     forever begin : cmd_loop
15       shortint unsigned result;
16       seq_item_port.get_next_item(cmd);
17       bfm.send_op(cmd.A, cmd.B, cmd.op, result);
18       cmd.result = result;
19       seq_item_port.item_done();
20     end : cmd_loop
21   endtask : run_phase
```

图 181. `driver` 里的 `run phase`

`run phase` 调用了 `seq_item_port` 的 `get_next_item()`方法. 这个方法是阻塞性的, 它从 port 里取得 sequence 发送数据, 然后我们从 `sequence_item` 对象里取得 `cmd`.

取得指令之后, 我们调用 BFM 中的 `send_op` 并得到结果. 然后, 我们把结果存到 `cmd` 对象里, 来把结果返回给调用者. 我们假定传给我们 `cmd` 的代码还保留着 `driver` 的句柄, 然后把数据存进 `cmd` 的话就能把结果返回给调用者了.

我们调用 `seq_item_port` 对象的 `item_done()`方法来指示 `sequencer` 可以发送下一个 sequence 项目了.

等一下, `seq_item_port` 是哪里来的?

这引出了我们的下一步: 升级 `driver`.

第四步: 在环境里例化 `driver` 和 `sequence` 然后相互连接.

我们在 env 用 sequencer 替换 tester 然后连接到 driver 类:

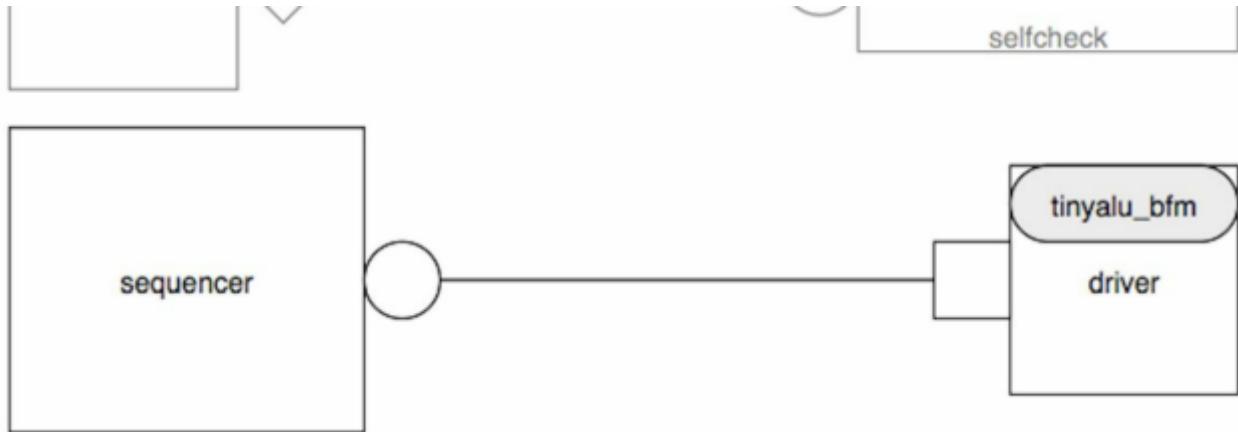


图 182. 用 sequencer 替换 tester

新的 environment 有两处改变:

- tester 替换成 uvm_sequencer.
- sequencer 不需要 FIFO 来连接 driver. driver 本来就是连接到 sequencer 的, 可以直接连接, 我们后面会看到.

下面是 env 中的关键部分代码, 声明了 sequencer 和其他组件:

```
1 class env extends uvm_env;  
2   `uvm_component_utils(env);  
3  
4   sequencer      sequencer_h;  
5   coverage       coverage_h;  
6   scoreboard     scoreboard_h;  
7   driver         driver_h;  
8   command_monitor command_monitor_h;  
9   result_monitor result_monitor_h;  
10
```

图 183. TinyALU env 里的声明

例化 sequencer 要注意构造器中的名字要正确:

```
15 function void build_phase(uvm_phase phase);  
16   // stimulus  
17   sequencer_h = new("sequencer_h",this);  
18   driver_h   = driver::type_id::create("driver_h",this);
```

图 184.

连接 sequencer 到 driver:

```
28 function void connect_phase(uvm_phase phase);  
29  
30   driver_h.seq_item_port.connect(sequencer_h.seq_item_export);
```

图 185. 连接 sequencer 到 driver

我们不需要用 TLM FIFO 来连接 sequencer 和 driver. uvm_sequencer 自带了 seq_item_export 对象(类似之前见过的 put_export 和 analysis_export). 通过调用 driver 的 seq_item_port 的 connect() 方法, 我们将 driver 连接到 sequencer.

之后所有的事情就是升级验证平台来处理 sequence. 我们已经准备好创建 UVM sequence 来发送 sequence 项目了.

第五步: 编写 UVM sequence.

uvm_sequence 类在 UVM 层级之外(构造器里没有 parent 参数), 不过可以通过 uvm_sequencer 向 UVM 层级发送数据. 从 uvm_sequence 派生的类都继承了三个东西来使得它们可以向 sequencer 发送数据:

- m_sequencer---这个数据成员保存着接收 sequence 项目的 sequencer 句柄.
- body() task---UVM 在开始发送 sequence 的时候启动这个 task.
- start_item()/finish_item()方法对---这两个方法得到对 sequencer 的控制并向其发送 sequence 项目.

我们通过派生 uvm_sequence 定义 uvm_sequence, 将其参数化为我们需要的 sequence_item, 然后编写 body() 方法来创建 sequence_itrm 并发送给 sequencer. 我们来通过生成著名的斐波那契数列来学习吧.

斐波那契数列通过对前两个数字相加生成后面的数字序列. 这个序列中的数字在大自然中普遍存在, 比如行星到太阳的距离, 比如雏菊中每一圈的花瓣数量. 下面是 8 位数据可以存储的斐波那契数值:

0 1 1 2 3 5 8 13 21 34 55 89 144 233

我们的 sequence 用 TinyALU 中的加法器生成这些数字.

下面是代码:

```
1 class fibonacci_sequence extends uvm_sequence #(sequence_item);
2   `uvm_object_utils(fibonacci_sequence);
3
4   function new(string name = "fibonacci");
5     super.new(name);
6   endfunction : new
```

图 186. 派生 uvm_sequence 来创建斐波那契数列

我们派生了 uvm_sequence 并将其参数化使之可以作用于 sequence_item. 我们用 uvm_object_utils() 宏而不是像我们用 uvm_component 一样用 uvm_component_utils() 宏, 还用了一个单参数的构造器.

我们所有的工作在 body() task 里完成. UVM 在有人启动 sequence 的时候调用 body() task:

```

9      task body();
10     byte unsigned n_minus_2=0;
11     byte unsigned n_minus_1=1;
12     sequence_item command;
13
14     command = sequence_item::type_id::create("command");
15
16     start_item(command);
17     command.op = rst_op;
18     finish_item(command);

```

图 187. 斐波那契 command 对象的操作

上面的片段展示了 uvm_sequence 的 body() task 的基础. 在 12 行, 我们声明了 sequence_item 对象句柄, 然后例化了指令对象.

start_item()方法在 uvm_sequencer 准备好接收 sequence 项目前会阻塞. 解除阻塞的时候, 我们就知道验证平台已经准备好接收指令了. 这里的指令是简单的复位.

finish_item()方法在 driver 完成指令前是阻塞的, 运行通过 finish_itrm()时, 我们就地址 TinyALU 复位完成, 准备好了斐波那契操作了:

```

`uvm_info("FIBONACCI", " Fib(01) = 00", UVM_MEDIUM);
`uvm_info("FIBONACCI", " Fib(02) = 01", UVM_MEDIUM);
for(int ff = 3; ff<=14; ff++) begin
    start_item(command);
    command.A = n_minus_2;
    command.B = n_minus_1;
    command.op = add_op;
    finish_item(command);
    n_minus_2 = n_minus_1;
    n_minus_1 = command.result;
    `uvm_info("FIBONACCI", $sformatf("Fib(%02d) = %02d", ff, n_minus_1),
              UVM_MEDIUM);
end
endtask : body
endclass : fibonacci_sequence

```

图 188. 输出斐波那契数列

上面的斐波那契循环展示了怎么从 DUT 中读取结果. 我们调用 start_item() 来等待 sequencer, 之后我们载入两个之前的斐波那契数值跟 add_op 指令一起到 ALU. 然后我们调用 finish_item() 方法来等待 driver 调用 item_done(), 在指令完成前都等待.

记得前面的 driver 把结果写回指令 transaction, 我们保留着 transaction 的句柄, 所有可以在 finish_item() task 完成返回之后读取结果.

这就是编写 sequence 的全部过程. 下面我们来探讨怎么在 test 里启动 sequence.

第六步: 编写用 sequencer 启动 sequencer 的 Test.

所有我们验证平台中的 test 都有着同样的 build_phase() 和 end_of_elaboration_phase() 方法, 我们要用这些方法来创建基类, 然后派生出我们的 test:

```

1 virtual class tinyalu_base_test extends uvm_test;
2
3     env      env_h;
4     sequencer sequencer_h;
5
6     function void build_phase(uvm_phase phase);
7         env_h = env::type_id::create("env_h",this);
8     endfunction : build_phase
9
10    function void end_of_elaboration_phase(uvm_phase phase);
11        sequencer_h = env_h.sequencer_h;
12    endfunction : end_of_elaboration_phase
13
14    function new (string name, uvm_component parent);
15        super.new(name,parent);
16    endfunction : new
17
18 endclass

```

图 189. TinyALU base test

`end_of_elaboration_phase()`方法从 `environment` 里拷贝了 `sequencer`. 现在任何派生 `base test` 的 `test` 都有 `sequencer` 的句柄了.

斐波那契 Test

斐波那契 `test` 例化了 `fibonacci_sequence` 然后用 `sequencer` 启动了它:

```

1 class fibonacci_test extends tinyalu_base_test;
2     `uvm_component_utils(fibonacci_test);
3
4     task run_phase(uvm_phase phase);
5         fibonacci_sequence fibonacci;
6         fibonacci = new("fibonacci");
7
8         phase.raise_objection(this);
9         fibonacci.start(sequencer_h);
10        phase.drop_objection(this);
11
12     endtask : run_phase

```

图 190. 斐波那契 `test`

所有的 `uvm_sequence` 都用 `start()`方法, 这个方法以 `uvm_sequencer` 为参数并在 `sequence` 完成之后返回. 这个例子里, `start()`方法在我们生成斐波那契数列之后就返回了. 这个 `Test` 的输出如下:

```

103 # UVM_INFO @ 0: reporter [RNTST] Running test fibonacci_test...
104 # UVM_INFO <snipped> [FIBONACCI] Fib(01) = 00
105 # UVM_INFO <snipped> [FIBONACCI] Fib(02) = 01
106 # UVM_INFO <snipped> [FIBONACCI] Fib(03) = 01
107 # UVM_INFO <snipped> [FIBONACCI] Fib(04) = 02
108 # UVM_INFO <snipped> [FIBONACCI] Fib(05) = 03
109 # UVM_INFO <snipped> [FIBONACCI] Fib(06) = 05
110 # UVM_INFO <snipped> [FIBONACCI] Fib(07) = 08
111 # UVM_INFO <snipped> [FIBONACCI] Fib(08) = 13
112 # UVM_INFO <snipped> [FIBONACCI] Fib(09) = 21
113 # UVM_INFO <snipped> [FIBONACCI] Fib(10) = 34
114 # UVM_INFO <snipped> [FIBONACCI] Fib(11) = 55
115 # UVM_INFO <snipped> [FIBONACCI] Fib(12) = 89
116 # UVM_INFO <snipped> [FIBONACCI] Fib(13) = 144
117 # UVM_INFO <snipped> [FIBONACCI] Fib(14) = 233

```

图 191. 生成斐波那契数

我给 fibonacci_sequence 的 start()方法传递了一个 sequencer 然后就得到了斐波那契数字. 由于我们验证平台里有一个 sequencer, 我们就可以在不改变结构的情况下随意的运行组合激励.

指定了 sequencer 的句柄之后, 我们可以通过调用 start()方法传入 sequencer 参数来随意启动任何 sequence. 不过传入 sequencer 到 start()方法也不是必需的. UVM 给我们提供了另外的使用"virtual"的机会, 就是将不用 sequencer 的 sequence 取名为 virtual sequence.

Virtual Sequence

我们的斐波那契 sequence 是简单的一次性的 sequence, 它运行它的激励然后结束. 不过, 我们经常会需要通过顺序或并行的运行多个 sequence 来得到混合的行为. 我们来用 runall_sequence 对 TinyALU 进行完全 Test 吧. runall_sequence 是一个 virtual sequence, 没有通过 sequencer 调用, 是通过得到获取 sequencer 的句柄然后用这个句柄来调用其他 sequence.

uvm_pkg 提供了 uvm_top 对象解决这个问题. uvm_top 对象的类型是 uvm_root, 它提供了向其他功能组件的访问. 其中一个功能组件是 find()方法.

uvm_top.find()方法以组件实例名的字符串为输入返回这个组件的句柄. 我们可以在 find 字符串里用通配符来在不知道整个层级的情况下查找组件.

uvm_top.find()方法返回的是 uvm_component 类型的对象, 所以我们需要把返回的基本类转换成需要的类. 这个例子里, 我们要通过将".env_h.sequencer_h"传入 uvm_top.find()方法得到 sequencer 的句柄然后转换返回的值到 sequencer 类型变量.

下面是 runall_sequence 类:

```

class runall_sequence extends uvm_sequence #(uvm_sequence_item);
  `uvm_object_utils(runall_sequence);

  protected reset_sequence reset;
  protected maxmult_sequence maxmult;
  protected random_sequence random;
  protected sequencer sequencer_h;
  protected uvm_component uvm_component_h;

  function new(string name = "runall_sequence");
    super.new(name);

    uvm_component_h = uvm_top.find("*.env_h.sequencer_h");

    if (uvm_component_h == null)
      `uvm_fatal("RUNALL SEQUENCE", "Failed to get the sequencer")

    if (!$cast(sequencer_h, uvm_component_h))
      `uvm_fatal("RUNALL SEQUENCE", "Failed to cast from uvm_component_h.")

```

图 192. virtual runall sequence 的顶部

上面的代码为了教学目的将获取句柄的流程拆分为三步。首先我们用 `uvm_top.find()` 来得到 sequencer 的 `uvm_component` 句柄。然后我们判断是否获取成功，如果没成功，表示字符串中有拼写错误。得到句柄之后我们将其转换成 sequencer 句柄，并判断是否转换成功。转换失败的话表示我们得到的句柄不是 sequencer 组件。

`runall_sequence` 使用三个其他的 `sequence` 来完成工作：`reset_sequence`, `maxmult_sequence` 和 `random_sequence`。我们用构造器例化它们，然后在 `body()` 方法里启动：

```

22   reset = reset_sequence::type_id::create("reset");
23   maxmult = maxmult_sequence::type_id::create("maxmult");
24   random = random_sequence::type_id::create("random");
25 endfunction : new

26
27 task body();
28   reset.start(sequencer_h);
29   maxmult.start(sequencer_h);
30   random.start(sequencer_h);
31 endtask : body
32
33 endclass : runall_sequence

```

图 193. virtual sequence 的 body 方法

`runall_sequence` virtual sequence 按顺序启动其他的 `sequence`。先复位 TinyALU，然后进行最大值的乘法，最后运行随机 `sequence`。

不用 Sequencer 启动 Virtual Sequence

UVM 开发者称 virtual sequence 为 "virtual" 是因为它们可以是不用 sequencer 启动的。这里是一个 `full_test` 的不用 sequencer 启动 sequence 的例子：

```

1 class full_test extends tinyalu_base_test;
2   `uvm_component_utils(full_test);
3
4   runall_sequence runall_seq;
5
6   task run_phase(uvm_phase phase);
7     phase.raise_objection(this);
8     runall_seq.start(null);
9     phase.drop_objection(this);
10    endtask : run_phase

```

图 194. 不用 sequencer 启动 sequence

runall_seq 用 null 传入 start()方法. 如我们所见, runall_sequence 有自己的方式来得到 sequencer.

多线程里的 Virtual Sequence

我们经常会需要在多线程里运行 sequence. 可以通过这个检查数据从不同端口进入是否会造冲突. 我们用 SystemVerilog 的 fork/join 结构来创建多线程. fork 表达式启动不同的线程, join 表达式在所有任务完成之前阻塞. 这里的"任务可以是 SystemVerilog 表达式, task 调用或者 sequence 启动.

我们来创建同时运行斐波那契 sequence 和随机 sequence 的一个 sequence.

这里是启动并行运行的 test:

```

1 class parallel_test extends tinyalu_base_test;
2   `uvm_component_utils(parallel_test);
3
4   parallel_sequence parallel_h;
5
6   function new(string name, uvm_component parent);
7     super.new(name,parent);
8     parallel_h = new("parallel");
9     endfunction : new
10
11   task run_phase(uvm_phase phase);
12     phase.raise_objection(this);
13     parallel_h.start(sequencer_h);
14     phase.drop_objection(this);
15    endtask : run_phase

```

图 195. 用 sequencer 启动 virtual sequence

这个例子里我们在顶层 sequence 里传入 sequencer, 它会在启动 fibonacci_sequence 和 short_random_sequence 的时候使用 sequencer.

这里是 parallel_sequence:

```

class parallel_sequence extends uvm_sequence #(uvm_sequence_item);
  `uvm_object_utils(parallel_sequence);

  protected reset_sequence reset;
  protected short_random_sequence short_random;
  protected fibonacci_sequence fibonacci;

  function new(string name = "parallel_sequence");
    super.new(name);
    reset = reset_sequence::type_id::create("reset");
    fibonacci = fibonacci_sequence::type_id::create("fibonacci");
    short_random = short_random_sequence::type_id::create("short_random");
  endfunction : new

  task body();
    reset.start(m_sequencer);
    fork
      fibonacci.start(m_sequencer);
      short_random.start(m_sequencer);
    join
  endtask : body
endclass : parallel_sequence

```

图 196. 使用 m_sequencer 启动并行的 sequence

`start()`方法存储了 `m_sequencer` 数据参数, 然后调用 `body()` task. `m_sequencer` 用来调用其他 sequence 的 `start()`.

我们首先调用 `reset_sequence` 的 `start()`, 然后用 `fork/join` 结构来并行启动 `fibonacci_sequence` 和 `short_random_sequence`. `sequencer` 会对两个 sequence 仲裁, 保证它们能用到 DUT.

`uvm_sequencer` 支持很多仲裁机制. 默认的是 FIFO, 就是我们在运行仿真时所见的穿插 sequence 项目的方式:

```

UVM_INFO @ 0: reporter [RNTST] Running test parallel_test...
UVM_INFO <snip> [FIBONACCI] Fib(01) = 00
UVM_INFO <snip> [FIBONACCI] Fib(02) = 01
UVM_INFO <snip> [SHORT RANDOM] random command: A: 22 B: 21 op: and_op = 0000
UVM_INFO <snip> [FIBONACCI] Fib(03) = 01
UVM_INFO <snip> [SHORT RANDOM] random command: A: e4 B: 22 op: rst_op = 0000
UVM_INFO <snip> [FIBONACCI] Fib(04) = 02
UVM_INFO <snip> [SHORT RANDOM] random command: A: 93 B: ed op: add_op = 0000
UVM_INFO <snip> [FIBONACCI] Fib(05) = 03
UVM_INFO <snip> [SHORT RANDOM] random command: A: 83 B: 7f op: add_op = 0000
UVM_INFO <snip> [FIBONACCI] Fib(06) = 05
UVM_INFO <snip> [SHORT RANDOM] random command: A: b2 B: 71 op: xor_op = 0000
UVM_INFO <snip> [FIBONACCI] Fib(07) = 08

```

图 197. 并行 sequence 发出的交错操作

我们可以看到斐波那契指令和随机 sequence 中的随机指令穿插到一起了. 我们检查过了, 这些指令没有互相干扰, 斐波那契数列还是跟往常一样.

总结

在本章, 我们学习了 UVM 技术的最后一篇: UVM sequence. Sequence 让我们得以拆分测试激励和验证平台结构, 混合不同的激励创建多个 test.

我们学习了怎样将验证平台转换成 sequence 驱动的验证平台, 怎样派生正确的类来使用 sequence, 怎样把数据从 DUT 传回验证平台. 然后我们见识了怎样在 sequence 里启动其他 sequence, 怎样并行启动 sequence.

我们也见识里创建多个激励行为并在 test 和 sequence 里混合的强大.

第二十四章

UVM 伴我同行

据说在赢得合气道黑带之后才算真正开始合气道学习之路。同样的说法也适用于在你学会 UVM Primer 的所有内容的时候。

这本入门教程首先教授了 UVM 的机制。你已经学会了怎么通过定义类，例化并连接对象来创建基于对象的验证平台。你也学习了怎样把验证平台拆分成更小更易重用的小块，还有怎么用 UVM sequence 的激励来驱动验证平台。

那接下来的是什么呢？

接下来你要开始使用 OOP 技术来创建验证平台了。幸运的是，你不会太孤单。UVM 是丰富强大的开发工具。开始了使用 OOP 验证平台之后，你就会解锁充满新技术和方法的世界。

享受 UVM 吧！