

## MCDF 课程实验 0（部分）

Hi，欢迎进入 SV 芯片验证的实验部分。实验 0 会帮助大家认识什么是设计功能描述文档，让大家在理解设计描述的同时，可以结合设计文件(Verilog 实现)，在展开实验之前可以一方面回顾 Verilog 的知识，另外一方面认识 MCDF 的结构。

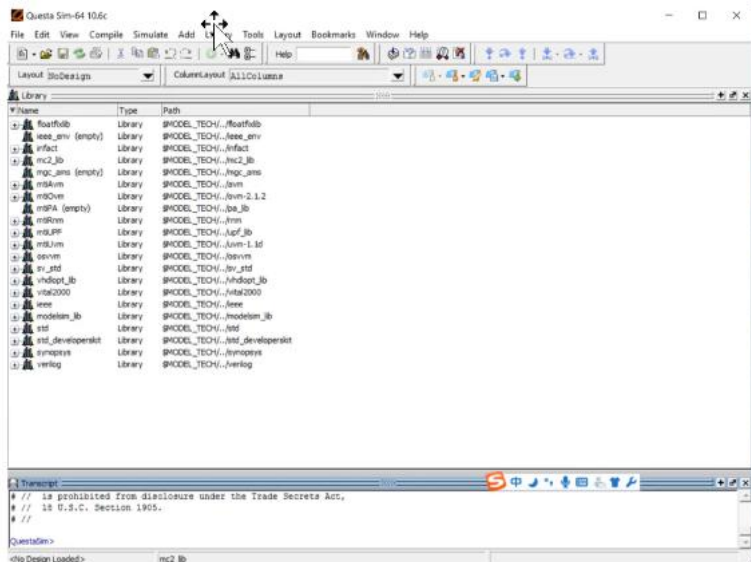
本次实验重点在于帮助大家实现这些实验前的要求：

- 安装仿真工具 Questasim。不建议安装 Modesim 或者 Quatus/ISE 等仿真工具。一方面 Questasim 对 SV 语言的支持很好，另外一方面实验需要同样的工具，保持大家在实验环境的一致性。
- 为了简易期间，路桑不对大家的代码编辑器提出要求，同学们可以在 Questasim 自带的编辑窗口中编辑代码，或者使用其他代码编辑器。
- 在实验综合练习环节，路桑会带领大家一同阅读可能是你人生中的第一份“硬件设计功能描述” ( hardware design function specification)文档。这一份 MCDF 功能文档将会贯穿我们整个课程和实验环节，所以同学们务必反复阅读该文档。
- 了解 MCDF 的 Verilog 设计和各个模块之间的功能、连接和交互。我们接下来的实验也将围绕这些模块或者 MCDF 子系统运用 SV 所学知识逐步升级验证环境。

### Questasim 仿真器安装.

工具安装资源和具体步骤在云协作，在安装时同学们需要注意以下几点：

- 分清楚你的机器是 64 位机器还是 32 机器，就最近几年的笔记本而言，64 位机器已经是普遍情况，而安装时也需要选择 64 位安装包还是 32 位安装包。。
- 注册机制需要按照云协作中的步骤，重点主要在于 WINDOWS 环境变量设置环节。正确安装之后，运行 Questasim 之后不会再出现“注册或者试用”的软件声明，那么恭喜你，实验所需要的唯一软件就已经安装成功喽！



## ■ MCDF 功能描述

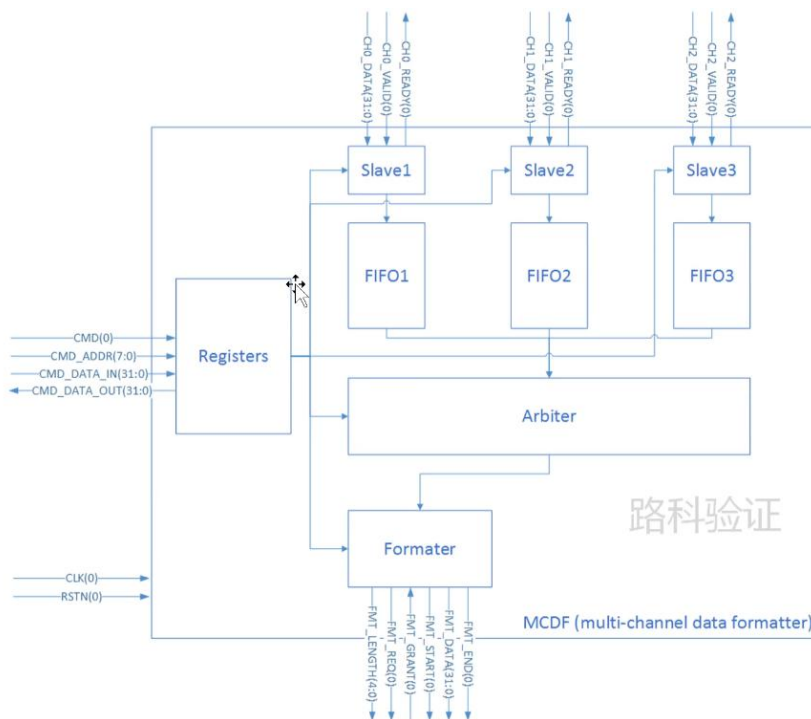
为了模拟实际情景,我们给出贯穿于 **SystemVerilog** 课程及实验的硬件设计 **MCDF**,并且遵循硬件设计描述的方式,介绍它的结构、功能、寄存器和时序。在以后的 **SV** 部分中,我们也将围绕这个硬件设计来考虑测试平台的构成。日后对测试平台的构建论述,需要经常引用 **MCDF** 的功能描述,请同学们对此注意。同时,熟悉硬件描述的方式,也是进入验证领域的一项基本技能。那么,就从这个规模适中的设计开始了解吧。

## 功能描述

该设计我们称之为多通道数据整形器(MCDF, multi-channel data formatter), 它可以将上行(uplink)多个通道数据经过内部的 FIFO,最终以数据包(data packet)的形式送出。

由于上行数据和下行数据的接口协议不同,我们也将后面的接口描述和时序部分进一步讲解。此外,多通道数据整形器也有寄存器的读写接口,可以支持更多的控制功能。

## ■ 设计结构



从上图的 **MCDF** 结构来看主要可以分为如下几个部分:

- 1.上行数据的通道从端(Channel Slave), 负责接收上行数据, 并且存储到其 **FIFO** 中。
- 2.仲裁器(Arbitrator) 可以选择从不同的 **FIFO** 中读取数据, 进而将数据进一步传送至整形器(formatter)。
- 3.整形器(Formatter)将数据按照一定的接口时序送出至下行接收端。
- 4.控制寄存器(Control Registers)有专用的寄存器读写接口, 负责接收命令并且对 **MCDF** 的功能做出修改。

■接口描述

系统信号接口

- CLK(0): 时钟信号。
- RSTN(0):复位信号，低位有效。

通道从端接口

- CHx\_DATA(31:0):通道数据输入。
- CHx\_VALID(0):通道数据有效标志信号，高位有效。
- CHx\_READY(0):通道数据接收信号，高位表示接收成功。

整形器接口

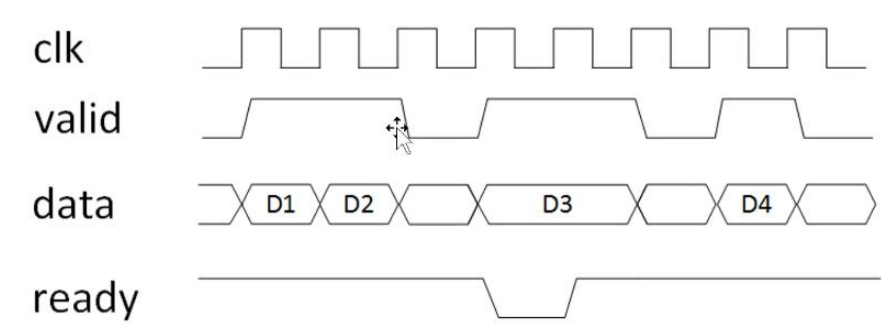
- FMT\_CHID(1:0):整形数据包的通道 ID 号。
- FMT\_LENGTH(4:0):整形数据包长度信号。
- FMT\_REQ(0):整形数据包发送请求。
- FMT\_GRANT(0): 整形数据包被允许发送的接受标示。
- FMT\_DATA(31:0): 数据输出端口。
- FMT\_START(0): 数据包起始标示。
- FMT\_END(0):数据包结束标示。

控制寄存器接口

- CMD(1:0):寄存器读写命令。
- CMD\_ADDR(7:0):寄存器地址。
- CMD\_DATA\_IN(31:0):寄存器写入数据。
- CMD\_DATA\_OUT(31:0):寄存器读出数据。

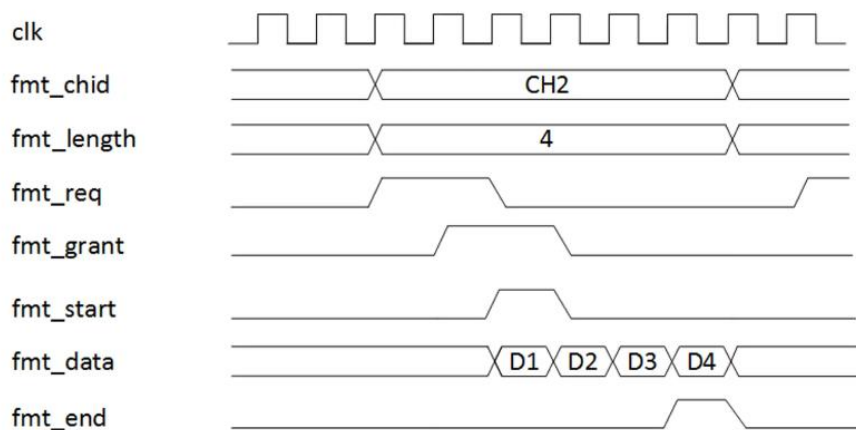
■接口时序

通道从端接口的时序



当 valid 为高时，表示要写入数据。如果该时钟周期 ready 为高，则表示已经将数据写入;如果该时钟周期 ready 为低，则需要等到 ready 为高的时钟周期才可以将数据写入。

整形器接口时序

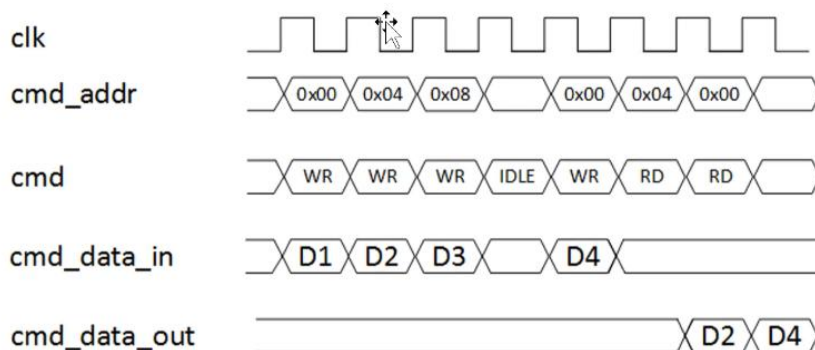


整形器发送数据是按照数据包的形式发送的，可以选择数据包的长度有 4、8、16 和 32。整形器必须完整发送某个通道的数据包后，才可以转而准备发送下一个数据包，在发送数据包期间，fmt\_chid 和 fmt\_length 应该保持不变，直到数据包发送完毕。

在整形器准备发送数据包时，首先应该将 fmt\_req 置为高，同时等待接收端的 fmt\_grant。当 fmt\_grant 变为高时，应该在下一个周期将 fmt\_req 置为低。Fmt\_start 也必须在接收到 fmt\_grant 高有效的下一个时钟被置为高，且需要维持一个时钟周期。在 fmt\_start 被置为高有效的同一个周期，数据也开始传送，数据之间不允许有空闲周期，即应该连续发送数据，直到发送完最后一个数据时，fmt\_end 也应当被置为高并保持一个时钟周期。

相邻的数据包之间应该至少有一个时钟周期的空闲，即 fmt\_end 从高位被拉低以后，至少需要经过一个时钟周期，fmt\_req 才可以被再次置为高。

#### 控制寄存器接口时序



在控制寄存器接口上，需要在每一个时钟解析 cmd。cmd 为写指令时，需要把数据 cmd\_data\_in 写入到 cmd\_addr 对应的寄存器中；当 cmd 为读指令时，即需要从 cmd\_addr 对应的寄存器中读取数据，并在下一个周期，将数据驱动至 cmd\_data\_out 接口。

#### ■ 寄存器描述

##### 地址 0x00 通道 1 控制寄存器 32bits 读写寄存器

bit(0): 通道使能信号。1 为打开, 0 位关闭。复位值为 1。

bit(2:1): 优先级。0 为最高, 3 为最低。复位值为 3。

bit(5:3): 数据包长度, 解码对应表为, 0 对应长度 4, 1 对应长度 8, 2 对应长度 16, 3 对应长度 32, 其它数值(4-7) 均暂时对应长度 32。复位值为 0。

bit(31:6): 保留位, 无法写入。复位值为 0。

**地址 0x04 通道 2 控制寄存器 32bits 读写寄存器**  
同通道 1 控制寄存器描述。

**地址 0x08 通道 3 控制寄存器 32bits 读写寄存器**  
同通道 1 控制寄存器描述。

**地址 0x10 通道 1 状态寄存器 32bits 只读寄存器**  
bit(7:0): 上行数据从端 FIFO 的可写余量，同 FIFO 的数据余量保持同步变化。复位值为 FIFO 的深度数。  
bit(31:8): 保留位，复位值为 0。

**地址 0x14 通道 2 状态寄存器 32bits 只读寄存器**  
同通道 1 状态寄存器描述。

**地址 0x18 通道 3 状态寄存器 32bits 只读寄存器**  
同通道 1 状态寄存器描述。

# MCDF 实验 1

## TB1. 从 Verilog 到 SV 的进场

同学们可以先下载 `tb1.v` 文件，该文件同上一次实验 0 的测试文件 `tb1.v` 的内容是一致的。接下来，同学们需要将 `tb1.v` 的文件名修改为 `tb1.sv`，即将文件后缀修改为 `.sv`。接下来就要开始我们的实验要求啦！

### 要求 1.1

在修改为 `tb1.sv` 之后，同学们可以按照之前的步骤，编译仿真，查看仿真行为是否同 `tb1.v` 的仿真行为一致？这说明了什么呢？

### 要求 1.2

同学们可以将 `tb1.sv` 中的信号变量类型由 `reg` 或者 `wire` 修改为 `logic` 类型，再编译仿真，查看行为是否同修改之前的一致呢？这是为什么？

### 要求 1.3

在步骤 2 的基础上，如果同学们将 `rstn` 的类型由 `logic` 修改为 `bit` 类型，再编译仿真，行为是否同步骤 2 的一致呢？这是为什么？

不一致

## TB2. 方法 task 和函数 function

同学们在 `tb2.sv` 文件中，可以看到不同于 `tb1.sv` 文件的是，之前产生时钟和发起复位的两个 `initial` 过程块语句都被两个 `task` 即 `clk_gen()` 和 `rstn_gen()` 取代了。接下来同学们请完成实验部分：

### 要求 2.1

不做修改的情况下，对 `tb2.sv` 进行编译仿真，时钟信号和复位信号还正常吗？为什么？

### 要求 2.2

同学们在路桑标记的两个 `initial` 块中分别调用产生时钟和复位的 `task`，再编译仿真查看时钟信号和复位信号，是否恢复正常呢？

### 要求 2.3

为什么要将两个 `task` 在两个 `initial` 块中调用？这是为什么呢？是否可以在一个 `initial` 块中调用呢？如果可以，调用它们的顺序是什么？

### 要求 2.4

同学们是否可以读出目前时钟的周期和频率呢？该如何测量呢？如果我们想进化 `clk_gen()` 方法，使其变为可以设置时钟周期的任务 `ckgen(intperiod)`，那么该怎么修改目前的任务 `clk_gen()` 呢？修改成功后，请在 `initial` 块中调用任务 `cdk_gen(20)`，看看波形中的时钟周期是否变为 20ns 呢？

## 要求 2.5

如果将 `timescale 1ns/1ps` 修改为 `timescale 1ps/1ps`，那么仿真中的时钟周期是否发生变化?这是为什么呢?

## TB3. 数组的使用

同学们在实验 0 环境中有没有对“`data_test`”部分比较好奇呢?譬如为什么要产生这么多的数据?而如果要发起激励的时候，每个 slave 的数据之间有什么相同和不同的地方呢?接下来我们会使用 `tb3.sv` 实验文件，对数组类型多加练习吧!

### 要求 3.1

如果路桑现在要求同学们对每一个 slave 的数据发出 100 个数，那么大家该怎么实现呢?请按照你的方式实现吧!

### 要求 3.2

如果我们现在要先生成 100 个数，并对它们按照日前的数值规则进行赋值，那么请同学们创建 3 个动态数组，分别放置要发送给 3 个 slave 的数据。

### 要求 3.3

接下来我们利用之前生成的数组数据，将它们读取并发送给三个 channel。

## TB3. 验证结构

为了实现清晰的验证结构，我们希望将 DUT 和激励发生器(stimulator) 之间划分。因此，我们可以将激励方法 `chnl_write()`封装在新的模块 `lhn1_initiator` 中，请同学们下载 `tb4.sv`，在接下来开展实验步骤前，同学们可以将“数组的使用”环节中添加的代码部分移柏到 `tb4.sv` 对应的位置上。

从 `tb4.sv` 中同学们可以发现之前的 `initil` 语句块“`channel_write_task`”已经不见了，在其位置上的变为了三个例化的 `shnl_initiator` 实例 `chnl0_init`、`chnl1_init` 和 `chnl2_init`。它们的作用扮演每个 `channel_slave` 通道对应的 `stimulator`，发送激励，因此我们在其模块 `chnl_initiator` 中定义了它的三个方法，即 `set_name()`、`chpl_write()`和 `chnl_idle()`。

### 要求 4.1

`chnl_idle()` 要实现的一个时钟周期的空闲，在该周期中，`ch.valid` 应为低，`ch.data` 应为 0。

### 要求 4.2

`Chnl_write()`要实现一次有效的写数据，并随后调用 `chnl_idle()`，实现一个空闲周期，在实现有效写数据时，请同学们考虑如何使用 `ch_ready` 信号，结合功能描述的 `channel_slave` 接口时序来看，只有当 `valid` 为高且 `ready` 为高时，数据写入才算成功，如果此时 `ready` 为低，那么则应该保持数据和 `valid` 信号，直到 `ready` 拉高时，数据写入才算成功。

### 要求 4.3

`set_name()`即设置实例的名称，在 `initial` 过程块 “`data_test`”中，在发送各个 `channel` 数据前，请设置各个 `channel_initiator` 的实例名称，这样方便在仿真时各个实例的打印信息可以显示它们各自的名称、数据发送时间和数据内容，便于阅读和调试。

### 要求 4.4

最后，同学们进入了本次实验的最后一个步骤了，路桑之所以提出发送更多的数据，并发送更紧凑高速的数据，是为了同学们可以观察到，是否你的二个 `channel.slave` 各自的 `chX_ready` 信号可以拉低呢？如果拉低了，这代表着什么？那么请你试试看，考虑如何发送更多更快的数据，让 `MCDT` 的三个 `chx_ready` 信号可以拉低吧。



# MCDF 实验 2

实验 2 的部分我们将主要回顾之前接口、仿真结束、类和包的使用。在这一个试验中，同学们将逐渐从使用硬件盒子验证过渡到使用接口和软件盒子(class)来验证设计。而这一个实验之所以重要也是因为它是硬件验证方式与软件验证方式之间的过渡，同时作为验证环境的启蒙，同学们在本实验的最后一个小实验中也能够初步体会到类的继承和层次包含关系，而这些都是作为日后学习高阶 UVM 知识的重要基础。好了，接下来让我们进入这次的 4 个小实验部分。

请下载 lab2 中的四个实验文件 tb1.sv、tb2.sv、tb3.sv 和 tb4.sv，同时不要忘记检查最新的 DUT MCDT 的版本是否是最新的，这一点也很重要。

## tb1 接口的使用

tb1.sv 的代码部分承接的是上一次实验的最后代码部分，在使用接口之前我们需要定义接口 chnl\_intf 和内部的端口，同时我们也声明了一个时钟块(clocking\_block)，它的功能是为了消除可能存在的竞争(race)问题，确保时钟驱动数据之间有一定的延迟，以便于 DUT 顺利采样。

因此更新后的代码，同学们可以发现例化的实例包括了只产生数据的 channel\_generator，只负责发送数据的 channel\_initiator 以及作为验证组件和 DUT 之间的接口 chnl\_intf。而例化之后，在 initial 块中需要通过调用组件的方法完成初始化、名字设置和空闲周期的设置。这里初始化是为了设置 ID，名字设置时为了稍后打印时容易区分各个组件，而空闲周期的设置则关系到我们接下来的实验要求：

### ■要求 1.1

首先观察波形，同学们可以发现 channel\_initiator 发送的数据例如 valid 和 data 与时钟 clk 均在同一个变化沿，没有任何延迟。同我们课程中所讲到的，这种 0 延迟的数据发送不利于波形查看和阅读，因此请同学们在已有代码的基础上使用 intf.mk 的方式来做数据驱动，并且再观察波形，查看驱动的数据与时钟上升沿的延迟是多少。

### ■要求 1.2

为了更好地控制相邻数据之间的空闲间隔，我们又引入了一个变量 idle\_cycles，它表示相邻有效数据之间的间隔。已有代码会使得有效数据之间保持固定的一个空闲周期，我们需要大家使用 idle\_cycles 变量，来灵活控制有效数据之间的空闲周期。通过这个方法，在 tb 的 initial 块中我们通过方法 set\_idle\_cycles()使得三个 channel initiator 的空闲周期变为 0，即可以实现有效数据的连续发送。

## tb2 仿真的结束

tb2.sv 中同学们需要课外学习 fork-join 的基本功能和使用方法，了解它的并行运行特性，以此来实现三个 chnl\_initiator 同时发送数据的要求。同时我们又将不同的 test 也组装到 task 中，以此来区分不同的测试内容，这是由于每一个测试任务的测试目的和要求都不

相同，具体要求可以在代码中查找。

tb2.sv 需要首先移植 tb1.sv 的要求内容，接下来再完成新的实验要求。

## ■要求 2.1

可以参考 task basic test(), 来完成 burst test()。它的要求是使得每个 chnl\_initiator 的 idle\_cycles 设置为 0，同时发送 500 个数据，最后结束测试。

## ■要求 2.2

参考 task basic test()来完成 task fifo. full test()。它的要求是无论采取什么数值的 idle\_cycles，也无论发送多少个数据，只要各个 chnl\_initiator 的不停发送使得对应的 channel 缓存变为满标志(ready 拉低)，那么可以在三个 channel 都拉低过 ready 时(不必要同时拉低，先后拉低即可)，便可以立即结束测试。

# tb3 类的例化和类的成员

在这一部分，我们便将之前用来封装验证功能的硬件盒子(module) 中的数据和内容移植到软件盒子(class) 中来，同学们可以通过前后代码的相同点和不同点来比较实用类的时候，需要注意什么地方，同时也可以基本掌握类的例化，类的成员变量访问权限以及类的成员方法如何定义和使用。

## ■要求 3.1.

在将 module chnl\_initiator 和 module chnl\_generator 分别改造为 class chnl\_initiator 和 class chnl\_generator 后，同学们也可以发现我们同时定义了一个用来封装发送数据的类 chnl\_trans。要求 3.1 需要在 initial 块中分别例化 3 个已经声明过的 chnl\_initiator 和 3 个 chnl\_generator。

## ■要求 3.2.

由于每一个 chnl\_initiator 都需要使用接口 chnl\_intf 来发送数据，在发送数据之前我们需要确保 chnl\_initiator 中的接口不是悬空的，即需要由外部被传递。所以接下来的实验要求需要通过调用 chnl\_initiator 中的方法来完成接口的传递。

## ■要求 3.3.

接下来就可以调用已经定义过的三个 test 任务来展开测试了。

## ■要求 3.4.

最后是关于类的例化问题，请同学们观察 chnl\_generator 在例化 chnl\_trans t 时，有没有不恰当的地方，如果有请指出来现有的代码会造成什么样的潜在问题呢？

## tb4 包的定义和类的继承.

到了 tb4.sv, 我们又进一步引入了新的类 chnl\_agent、chnl\_root\_test、chnl\_basic\_test、chnl\_burst\_test 和 chnl\_fifo\_full\_test, 同时将所有的类(都是与 channel 相关的验证组件类), 封装到专门包裹软件类的容器 package chnl\_pkg 中且完成编译。因此编译后的 chnl .pkg. 会被默认编译到 work 库中, 与其它的 module 是一同并列放置的。

关于 chnl\_agent, 我们将它作为一个标准组件单元, 它应该包括 generator、driver(initiator)和 monitor。在 tb4.sv 中, 我们暂时只有 chnl\_generator 和 chnl\_initiator, 因此将它们放在 agent 中例化。同时, 我们也将之前用 task 来实现的测试任务也由类来实现。可以发现, 父类是 chnl\_root\_test, 而我们已经先移植了 chnl\_basic\_test, 接下来需要同学们实现另外两个类。

### ■要求 4.1

由于我们将各个类首先封装在了 package chnl\_pkg 中, 因此在 module tb4 中要声明类的句柄, 首先应该从 chnl\_pkg 中引入其中定义的类。

### ■要求 4.2

可以参考之前已经实现的 burst\_test() 和 fifo\_full\_test()任务, 以及已经实现的类 chnl\_basic\_test, 按照同样的要求来实现两个新的类 chnl\_burst\_test 和 chnl\_fifo\_full\_test。

### ■要求 4.3

例化三个测试环境。在 4.2 中继承了三个 chnl\_root\_test 的子类, 并在 import package 之后, 创建了对应的句柄。现在要例化三个 test, 即创建对应的对象, 并将句柄指向这些对象。

### ■要求 4.4

将每个通道的接口, 赋值给对应的 chnl\_initiator。

### ■要求 4.5

开始运行。

## MCDF 实验 3

实验 3 的部分我们将**主要就随机约束和环境结构做实践**。在这一个试验中，同学们将升级实验 2 部分中对 generator 和 initiator 之间的数据生成和数据传输的处理，同时我们也将**完善何时结束测试**，将其主动权交于 generator 而不再是 test 组件。在组件结构实践部分中，同学们将在原有的 initiator、generator、agent 和 test 组件的基础上再认识 monitor 和 checker，并且使其构成一个有机的整体，最终**可以通过在线比较数据**的方式完成对 MCDT 的测试。

### TB1.随机约束

为了使同学们更早习惯各个验证文件独立放置的原则，我们已经先将 chnl\_pkg1.sv 文件和 tb1.sv 文件独立开来，所以 tb1 需要编译两个文件即 chnl\_pkg1.sv 和 tb1.sv。在这个试验中**我们会进一步了解随机约束在类中定义方式、如何随机化对象、随机种子的使用方法、对象的产生等等**。接下来,请同学们按照实验要求开始练习吧。

#### 要求 1.1.

我们继承了大部分实验 2 的代码，包括 chnl\_basic\_test 类,而对于这个类所需要生成的数据包我们提出了新的约束要求。需要注意的是，与实验 2 不同的是，这次数据类 chnl\_trans 的定义发生了很大的变化，它不再只局限于包含一个数据，而是多个数据，同时跟数据之间时间间隔的控制变量也在该类中声明为随机成员变量，那么**请按照代码中具体的约束来定义 chnl\_basic\_test 类，注意该代码的修改需要在 chnl\_trans 类中实现，因为目前的代码结构只有 chnl\_trans 类的更新是较为合适的办法。**

### 要求 1.2.

我们需要将原本在 `chnl_root_test` 类中用来结束仿真的 `$finish()` 变迁到 `generator` 中，那么请将它放置到合适的地方，然后由 `generator` 来结束仿真吧。

### 要求 1.3.

同学们尝试着多次重新启动仿真，可以使用 `"restart"` 命令来重启，再对比连续两次生成的随机数据，看看它们之间是否相同呢？然后再在仿真器命令行处使用命令

```
"vsim -novopt -solvefaildebug -sv_seed 0 work.tb1"
```

来加载仿真。这里我们多传递了两个必须的仿真参数，`-solvefaildebug` 是为了调试随机变量的，而 `-SV seed NUM` 则是为了传递随机的种子。那么使用这个命令再看，是否与之前没有使用 `-sV_seed 0` 的命令产生了相同的数据呢？最后，请改为使用

```
"vsim -novopt -solvefaildebug -sv seed random work.tb1"
```

命令再来比较前后两次的结果，是否相同呢？那么，你对 `-sV seed random` 的仿真选项的认识是什么？

### 要求 1.4

在仿真的最后，同学们可以发现最后一个打印出来的 `chnl_trans` 对象的 `obj. id` 值是 1200，那么这代表什么含义？为什么会有 1200 个 `chnl_obj` 对象产生呢？整个仿真过程中，在同一时刻，最多的时候一同有几个 `chnl_trans` 对象在仿真器内存中存在呢？这么做对内存的利用是否合理？你是否还有更好的办法使得在同一时间 `chnl_trans` 对象的数量比代码中用到的更少呢？

## TB2.更加灵活的测试控制

如果我们要实现不同的 test 类，例如 chnl\_basic\_test、chnl\_burst\_test 和 chnl\_fifo\_full\_test，那么我们对于不同的 test 需要对 cinl\_generator 的随机变量做出不同的控制，继而进一步控制其内部随机的 chnl\_trans 对象。也就是说，随机化也是可以分层次的，例如在 test 层可以随机化 generator 层，而依靠 generator 被随机化的成员变量，再来利用它们进一步随机化 generator 中的 chnl\_trans 对象，由此来达到顶层到底层的随机化灵活控制。那么从这个角度出发，我们就需要将 generator 从 agent 单元中搬迁出来，并且搁置在 test 层中来方便 test 层的随机控制，因此我们在 chnl\_pkg2.sv 和 tb2.sv 中主要带领大家认识如何更好的组织验证结构，从而实现更加方便的测试控制。

### 要求 2.1

由于我们将 generator 搬迁到 test 层次中，所以在要求 2.1 中需要将 gen 和 agent 中组件的 mailbox 连接起来，方便 gen 与 agent 中 init 的数据通信。

### 要求 2.2

在领略了如何在 test 中的 do\_config 对 gen[0]进行随机化控制后，你需要对 gen[1]也按照代码中的具体要求进行随机控制。

### 要求 2.3

请按照代码中的具体要求对 gen[2]进行随机控制。

### 要求 2.4.

请按照代码中的具体要求，在 chnl\_burst\_test::do\_config()任务中对三个 generator 进行随机控制。

### 要求 2.5

请按照代码中的具体要求，在 chnl\_fifo\_full\_test::do\_config()任务中对三个

generator 进行随机控制。

## 要求 2.6

在 tb2.sv 中，我们对于测试的选择将由仿真时的参数传递来完成。这意味着，以后的递归测试，即创建脚本命令，由仿真器读入，分别传递不同的随机种子和测试名称即可完成对应的随机测试，而这种方式即是回归测试的雏形。所以请同学们按照之前的仿真命令，在命令窗口中添加额外的命令"+TESTNAME=testname"，这里的+ TESTNAME=表示的仿真命令项，在由内部解析之后，testname 会被捕捉并且识别，例如你可以传递命令"+TESTNAME=chnl\_burst\_test" 来在仿真时运行测试 chnl\_burst\_test。请同学充分理解要求 2.6 的代码部分，懂得如何捕捉命令，如何解析命令，最后如何选择正确的测试来运行。

## TB3 测试平台的结构

最后一个实验部分即指导同学们认识验证环境的其它组件，monitor 和 checker。并且通过合理的方式来构成最终用来测试 MCDT 的验证环境，在这个环境中同学需要再回顾 generator、initiator、monitor 和 checker 各自的作用。在顶层环境中，我们将 checker 置于 test 层中，而不是 agent 中，需要同学们思考这么做的好处在什么地方。同时需要在认识 generator 和 Initiator 有数据通信的同时，可以掌握 monitor 与 checker 之间的数据通信，还有 checker 如何针对 MCDT 利用内部的数据缓存进行数据比较。

## 要求 3.1

在 chnl\_monitor 类和 mcdt\_monitor 类各自的 mon\_trans()方法中需要采集正确的数据，将它们写入 mailbox 缓存，同时将捕捉的数据也打印出来,便于我们的调试。

### 要求 3.2

在 `chnl_agent` 中，参考如何例化的 `initiator` 对象，也对 `chnl_monitor` 对象开始例化、传递虚接口和使其运行。

### 要求 3.3

在 `chnl_checker` 的任务 `do_compare()` 中，同学需要从 `checker` 自己的数据缓存 `mailbox` 中分别取得一个输出端的采集数据和一个输入端的采集数据，继而将它们的内容进行比较，需要注意的是，输出端的缓存只有一个，而输入端的缓存有三个，同学需要考虑好从哪个输入端获取数据与输出端缓存的数据进行比对。

### 要求 3.4

在顶层环境 `chnl_root_test` 中。同学们先要对 `mcdt_monitor` 和 `chnl_checker` 进行例化。传递虚接口，并且将 `chnl_monitor.mcdt_monitor` 的邮箱句柄分别指向 `chnl_checker` 中的邮箱实例。

实验 3 的整体要求较之前的两个实验都要困难一些，因为它首次引入和随机约束和完整的验证结构。在充分理解实验 3 的要求基础上，我们接下来的实验 4 将会迎来更大的 MCDF 子系统验证环境，到时候我们可以更好地梳理子系统的验证环境，结合课程所讲的验证结构逐步完成 MCDF 的验证计划。



## MCDF 实验 4

首先路桑要恭喜你，你已经迈过了 SV 小白的阶段。在实验 3 中，你可以通过一个初具规模的 MCDT 的验证环境理解验证的一些要素。在接下来进入实验 4 之前，你头脑中需要再复习这些概念。**第一，验证环境按照隔离的观念，应该分为硬件 DUT，软件验证环境，和处于信号媒介的接口 interface。第二，对于软件验证环境，它需要经历建立阶段 (build)，连接阶段(connect)，产生激励阶段(generate) 和发送激励阶段(transfer)。**只有当所有的激励发送完毕并且比较完全之后，才可以结束该测试。在课堂上，你跟着路桑回顾了实验 3 的代码结构之后，你就可以进入实验 4 了。

实验 4 的环境是完善的，这对你而言，是个好消息。不妨再像个刘姥姥进大观园一样，看看实验 4 的代码与实验 3 相比，是不是又更上一层楼了呢？这其中的原因主要有两方面，一方面是因为我们从实验 4 开始在验证**更大的子系统**，即 MCDF。与 MCDT 相比，MCDF 主要添加了寄存器控制和状态显示功能，同时也添加了一个重要的数据整形功能，因此，你会发现实验 4 的验证文件多了。另外一方面，代码之所以增多是为了让整个验证环境的各个组件之间相互独立，功能清晰，**你可以发现不同的 package 之间的功能是独立的**，同一个 package 中的各个验证组件的功能也是独立的。那么不同组件之间的同步和通信依靠什么呢？没错，那就是我们已经学习到的 event 和 mailbox。

怎么开始试验 4 呢？大胆地编译所有的文件，然后给出仿真命令就可以了：

```
Vsim -novopt -classdebug -solvefaildebug -sv_seed 0  
  
+TESTNAME=mcdf_data_consistence_basic_test -  
  
|mcdf_data_consistence_basic_tast.log work.tb
```

来看看上面主要的仿真命令项：

-classdebug，这是为了提供更多的 SV 类调试功能

-solvefaildebug，这是为了在 SV 随机化失败之后有更多的信息提供出来

-sv\_seed 0，暂时给固定的随机种子 0

+TESTNAME=mcdf\_data\_consistence\_basic\_test，这是指定仿真选择的调试

-| mcdf\_data\_consistence\_basic\_test\_sim.log，这是让仿真的记录保存在特定的测试文件名称中

那么在开始测试并且最终结束之后，在你的项目目录下会有两个文件产生

mcdf\_data\_consistence\_basic\_test\_sim.log 会保存所有的仿真信息，而

mcdf\_data\_consistence\_basic\_test\_check.log 则只会保存 mcdf\_checker 做数据比较

和最终比较报告的信息。需要注意的是，在你测试的过程中，如果测试发生了错误则会立

即停止下来，你可以根据当前时刻的错误报告定位出数据比较错误的时间点，再去核对波

形，如果经过确认确实是硬件的行为有问题，那么路桑需要恭喜你，你发现了一个 BUG，

赶快举手来跟路桑领红包吧！

那么验证环境这么完善，是不是你不需要做实验 4 了呢？NONON...我们实验 4 的重点是要求同学们在本次试验中领会，**怎么样才是一个完整的验证计划和实施**。还记不记得我们课程中谈到的，**什么是验证计划的核心要素点？那就是围绕着设计的功能点罗列出需要展开测试的功能点，以及如何展开测试，并且指明测试的验收标准是什么。我们实验 4 中主要以测试是否通过为原则，而在实验 5，同学们将进一步掌握代码覆盖率和功能覆盖率的验收标准。**

## 要求 1.1 ( 必做 )

从路桑带领大家勾画出实验 3 的验证环境框图，想必你已经对验证框图的重要性有体会了吧？没错，从验证框图可以更好地在理解验证环境的组件和组件之间的通信连接情况。无论对于你接下来着手构建环境，还是将它作为你验证代码的一个形象说明，它都比代码更直接地形容整个验证环境。那么接下来，请你仍然按照我们之前课堂中所讲的，准备好章 A4 纸、铅笔和橡皮，来**理解实验 4 中的验证环境**。不用着急，从我们这次实验一开始所讲的，在大脑中回顾一下，验证环境的核心要素是什么，然后开始画一下你理解的验证结构吧。

## 要求 1.2 ( 必做 )

接下来，路桑就会指定你需要在 `mcd_f_pkg.sv` 中参考以后测试类 `mcd_f_data_consistence_basic_test` 而去创建其它的测试类名，**它们对应的测试功能点，以及测试标准是什么**。在你按照要求，实现了所有的测试之后，记得将每个测试都至少跑一遍，如果你跑的过程中发现 `mcd_f_checker` 报错，那么看看是不是发现了什么设计问题，如果你有信心它是一个漏洞，那么就赶快联系助教吧，助教会帮你把它记录到漏洞跟踪表格中，而后带设计修改确认漏洞修复之后，他会发布下一版的设计，到时候也会及时在班级群众通知大家，大家记得要及时下载更新后的设计版本哦。理论上，如果你对同一个测试，每次使用不同的随机种子即 `"-sv_seed RANDOM"`，那么你跑得次数越多，就越有机会发现新的“惊喜”，而至于你需要跑多少遍，或者每个测试中需要发送多少数据包（发送的越多越好，那么究竟要发送多少次才可以停下来呢？标准是什么？）？我们将在实验 5 中为同学们揭晓覆盖率的验证量化方式。从目前的测试来看，那就尽管让仿真跑下去吧，你完全可以去吃一顿饭，在吃饭的功夫中让测试不断发包，而目前测试的标准是一旦数据包比较错误就会停止，到时候就可以在“案发”的第一时间去探索真相了。

测试功能点	测试内容	测试通过标准	测试类名
寄存器读写测试	所有控制寄存器的读写测试所有状态寄存器的读写测试。	读写值是否正确	mcd_f_reg_write_read_test
寄存器稳定性测试	非法地址读写，对控制寄存器的保望域进行读写，对状态寄存器进行写操作。	通过写入和读出，确定寄存器的值是预期值，而不是紊乱值，同时非法寄存器操作也不能影响 MCDF 的整体功能	Mcdf_reg_illegal_access_test
数据通道开关测试	对每一个数据通道对应的控制寄存器域 en 配置为 0，在关闭状态下测试数据写入是否通过	在数据通道关闭情况下，数据无法写入，同时 ready 信号应该保持为低，表示不接收数据，但又能使得数据不。被丢失，因此数据只会停留在数据通道端口	Mcdf_channel_disable_test
优先级测试	将不同数据通道配置为相同或者不同的优先级，在数据通道使能的情况下进行测试	如果优先级相同，那么 arbiter 应该采取轮询机制从各个通道接收数据，如果优先级不同，那么 arbiter 应该先接收高优先级通道的数据，同时，最终所有的数据都应该从 MCDF 发送出来	Mcdf_arbiter_priorit_test
下行从端低带宽测试	将 MCDF 下行数据接收端设置为小存储量，低带宽的类型，由此使得在由 formatter 发送出数据之后，下行从。端有更多的机会延迟 grant 信号的置位，用来模拟真实场景	在 req 拉高之后，grant 应该在至少两个时钟周期以后拉高，以此来模拟下行从端数据余量不足的情况。当这种激励时序发生 10 次之后，可以停止测试。	Mcdf_formatter_grant_test

### 要求 1.3 ( 选做 )

也许你在阅读要求 1.2 的测试标准的时候，心生疑惑，是否路桑目前提供的 mcdf checker 有能力可以完成所有的这些通过标准检查呢？实际上，我们还有一些地方需要填补，你能找到 mcdf checker 还需要单独实现哪些用来检查的功能吗？这里，路桑给你一点提示吧：

- 寄存器的读写，无论是否非法，都可以参考 mcdf\_data\_consistence\_basic\_test 中比较数据的函数来做比对。

- 数据通道的开关，在数据通道关闭的情况下，mcdf checker 应该不会收到输入端或者输出端的监测数据(想想为什么？)，因此不应该出现数据比较的信息。那么在这种情况下，怎么来判断这个测试通过呢？最简单的是，这个测试在最后的报告阶段总结中，error 信息的统计数量为 0，但这仍然不够。因为我们还要检查一个简单的时序，即当 slave channel 的被关闭时，当 valid 拉高时，ready 信号不应该出现拉高的情况，否则设计是有问题的，因为数据可能没有被真正写入到 FIFO，或者 slave channel 此时并没有被真正关闭。那么如何实现这一检查呢？路桑给的建议是，在 mcdf\_intf 中，将 en 信号监测到，将其传入 mcdf\_checker，同时也将 3 个 chnl\_intf 传入 mcdf\_checker。这样，mcdf checker 可以观测到所需的 valid、ready 和 en 信号，来完成这个检查。不妨给个 task 名称来保持代码统一吧，mcdf\_checker.do\_channel\_disable\_check()。

- 对于优先级的检查，mcdf\_checker 依然无法从 mcdf\_refmod 的预测数据中获取数据包前后的信息，也就是说，mcdf\_checker 依然无法判断最终从 mcdf 送出的数据是否符合了优先级的数据包发送顺序？那么，不妨依然将需要观察的 arbiter 信号在 mcdf\_intf 中声明并且观测，再新建一个 task 来保证优先级的检查吧，

mcdf\_checker.do\_arbiter\_priority\_check0.

●那么发包的长度，mcdcf checker 是否已经可以保证检查到了呢？是的，已经可以保证这点了，那么你是否能够从代码中看出，路桑是如何保证 formatter 最终发送的各个通道从端的数据包长度的检查的呢？谈谈你的看法。

●至于最后下行从端低带宽的测试，需要同学使用好如何对下行从端的配置约束，使其可以实现在低带宽数据消耗的情况下，自身的缓存量逐渐减少，而频繁在 formatter request 信号拉高时而延迟 grant 信号的拉高，以此来模拟真实情况。那么你还需要考虑的是，如何来观测 request 与 grant 之间超过两个时钟周期的时序延迟呢？你可以考虑由 test 直接使用 fmt intf 中的信号来做时序观察和计数，当满足 10 次之后，即可以停止数据的发送。如何停止数据的发送呢？可以考虑使用 fork-join/join. any/join. none 和 disablefork 等用法。

不过即使你最终没有完成要求 1.3，也不必为此难过和叹气，你现在就跟登山的探路者一样，回过头去再看实验 1、实验 2 和实验 3，是不是恍然间产生一种感觉，那真是——山又比一山高，不识庐山真面目，只缘身在此山中。所以，只管尽力去实现你的代码，要求 1.3 即便在验收的时候遗憾没有做完，你也已经做完实验 4 的必选部分了呢，距离最后一个实验 5 只是一步之遥，而路桑会在实验 5 的时候给出实验 4 的参考代码，到时候你再看看，你跟路桑的思路，是不是你的代码更加飘逸更有创造性呢？一定是你的！因为无知者无畏嘛！

## MCDF 实验 5

嗨！首先要恭喜你，终于坚持到我们最后这一个实验了！

我们本次实验将带领大家认识如何定义覆盖率，如何从验证计划到测试用例的实现，最后再到覆盖率的量化。从本次实验，大家可以掌握验证量化的两种基本数据，即代码覆盖率和功能覆盖率。从这两种覆盖率,我们就可以掌握何时结束验证，确认验证的完备性。那么，接下来就让我们开始吧。

首先请下载实验 5 的代码。实验 5 的代码是基于实验 4 的代码，主要修改的文件即 `mcd_f_pkg.sv` 在 `mcd_f_pkg.sv` 中，路桑已经为大家添加了一个关于 MCDF 的覆盖率模型 `mcd_f_coverage`,并且将它例化在顶层环境中。你可以阅读代码，了解 `mcd_f_coverage` 的例化、虚接口的传递、覆盖率的定义和采样。

同时，路桑为大家提供了一个新的 test,即 `mcd_f_full_random_test`. 这个 test 尽可能的将一些测试相关参数在仿真时进行了随机化。在接下来的仿真中，你依然可以复用你之前实验 4 按照要求创建的几个 test,将它们搬迁到 lab5 中。不过,我们实验 5 的要求是，需要同学们最终达到“尽可能高的代码覆盖率和功能覆盖率”。

**通过等级:**代码覆盖率大于 90%，功能覆盖率大于 90%。

**优秀等级:**代码覆盖率大于 95%，功能覆盖率等于 100%。

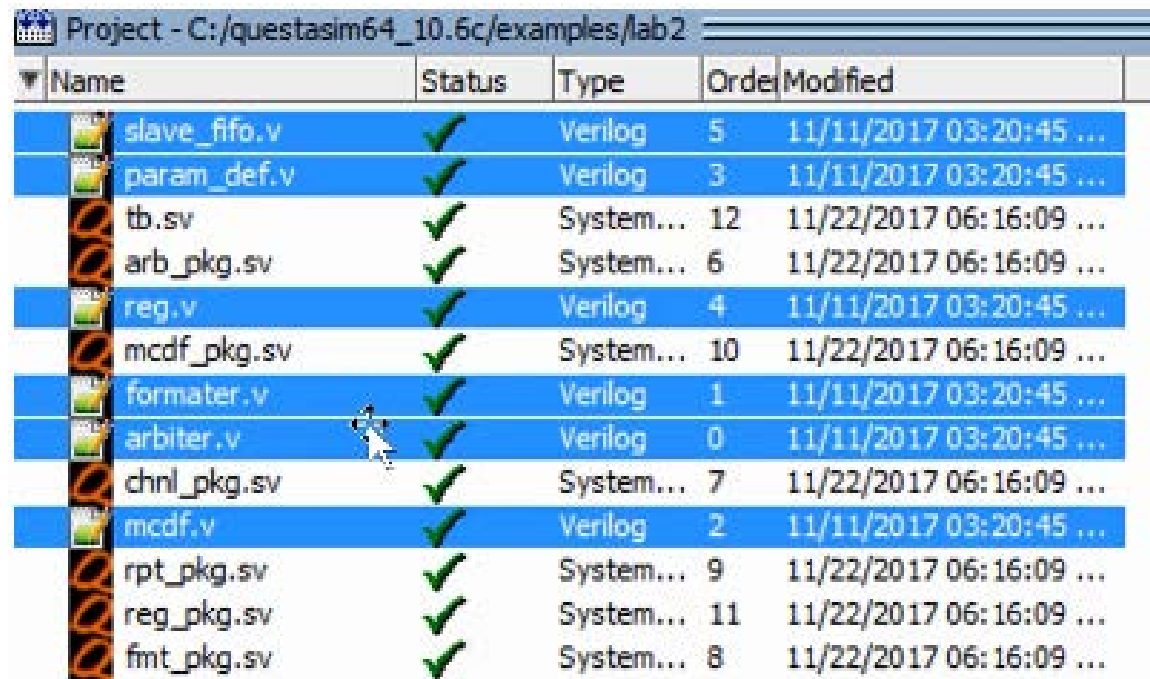
所以结合我们本次实验最终的验收目标，你可以复用你之前的测试用例，也可以使用路桑的测试用例。当然，在你学习了覆盖率相关的课程之后，你就懂得了，当最终覆盖率无法提升时，需要修改你的约束，或者创建新的 test。

### 编译

在编译过程中，我们需要对于设计相关的文件设置额外的覆盖率编译选项。如下面的

截图:

1、只选择与设计有关的文件

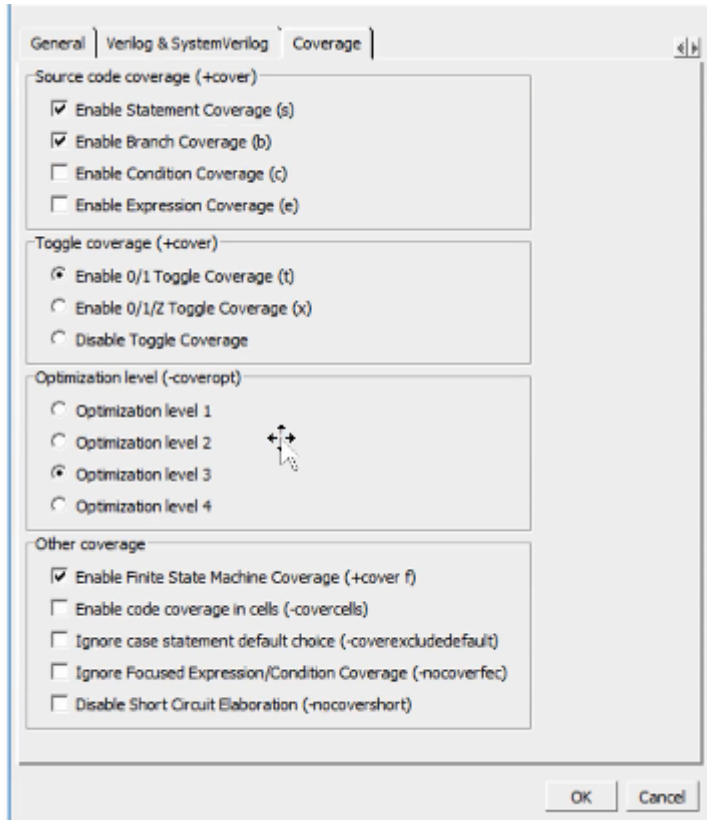


Name	Status	Type	Order	Modified
slave_fifo.v	✓	Verilog	5	11/11/2017 03:20:45 ...
param_def.v	✓	Verilog	3	11/11/2017 03:20:45 ...
tb.sv	✓	System...	12	11/22/2017 06:16:09 ...
arb_pkg.sv	✓	System...	6	11/22/2017 06:16:09 ...
reg.v	✓	Verilog	4	11/11/2017 03:20:45 ...
mcd_f_pkg.sv	✓	System...	10	11/22/2017 06:16:09 ...
formater.v	✓	Verilog	1	11/11/2017 03:20:45 ...
arbiter.v	✓	Verilog	0	11/11/2017 03:20:45 ...
chnl_pkg.sv	✓	System...	7	11/22/2017 06:16:09 ...
mcd_f.v	✓	Verilog	2	11/11/2017 03:20:45 ...
rpt_pkg.sv	✓	System...	9	11/22/2017 06:16:09 ...
reg_pkg.sv	✓	System...	11	11/22/2017 06:16:09 ...
fnt_pkg.sv	✓	System...	8	11/22/2017 06:16:09 ...

2.点击右键，选择 compile -> compile properties,在弹出设置栏的 coverage 一栏中，

如图选择以下选项，然后点击 OK。





3.完成所有文件的编译"Compile All" 。这一步将在编译 DUT 文件时生成代码覆盖率的模型，而我们之所以没有给 TB 相关文件添加代码覆盖率选项，是由于测试平台的覆盖率不是我们需要关注的对象。

**注意:**与测试相关的文件不要设置覆盖率编译选项。

## 仿真

接下来，在仿真窗口(transcript) 中，可以参考路桑的仿真命令:

```
vsim -i -classdebug -solvefaildebug -coverage -coverstore
```

```
C:/questasim64 10.6c/examples -testname mcdf_full_random_test -sv_seed
```

```
random +TESTNAME=mcdf_full_random_test -l mcdf_full_random_test.log
```

```
work.tb.
```

**注：**此处-l 中的 l 为小写的 L，不是符号 "I" 。

这里需要注意的是标注黄色的仿真命令,这些新增的命令说明如下：

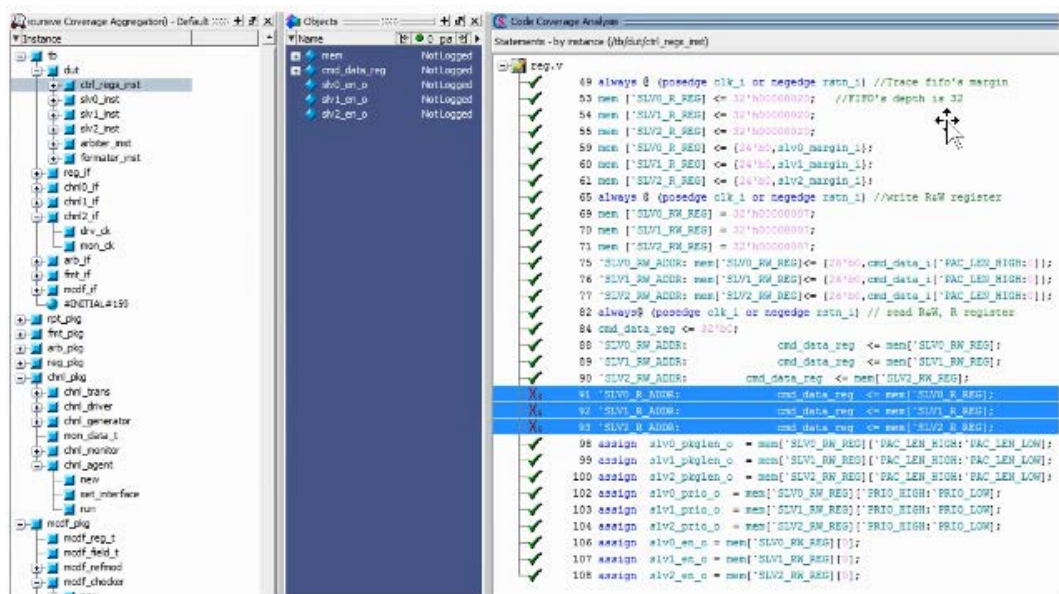
- coverage:会在仿真时产生代码覆盖率数据，功能覆盖率数据则默认会生成，与此选项无关。

- coverstore COVERAGE\_STORAGE\_PATH:这个命令是用来在仿真在最后结束时，生成覆盖率数据并且存储到 COVERAGE\_STORAGE\_PATH。你可以自己制定 COVERAGE\_STORAGE\_PATH，但需要注意路径名中不要包含中文字符。

- testname. TESTNAME:这个选项是你需要添加本次仿真的 test 名称，你可以使用同+TESTNAME 选项一样的 test 名称。这样在仿真结束后，将在 COVERAGE\_STORAGE\_PATH 下产生一个覆盖率数据文件 "{TESTNAME}\_{SV\_SEED}.data"。由于仿真时我们传入的种子是随机值，因此我们每次提交测试，在测试结束后都将产生一个独一无二的覆盖率数据。例如 mcdf\_full\_random\_test\_1988811153.data。

接下来在仿真窗口敲入命令“ run -all” ，在仿真最后自动结束时(大约 10ms)会弹出仿真要求结束的对话框"Are you sure you want to finish?"，你可以点击 NO，但一定要选择菜单栏“ Simulate -> end simulation"来结束本次仿真。只有你结束了本次仿真,你才会得到上面提到的覆盖率数据，例如 mcdf full random\_test 1988811153.data。

在本次仿真过程中或者结束时，你也可以利用仿真器直接查看代码覆盖率或者功能覆盖率。首先你需要选中 View -> Coverage -> "Code Coverage Analysys"和"Covergroups"。如果你需要查看代码覆盖率，那么选择新添加的 Analysis 窗口，然后逐个点击 Sim 窗口中 DUT 层次中的个别模块，例如下图中点击 “ctrl regs. Inst” 你可以在 Analysis 窗口中看到哪些代码被执行了而哪些代码在本次仿真中没有执行过。



如果你需要查看功能覆盖率，那么你可以在新添加的“Covergroups”窗口中查看本次仿真所收集到的功能覆盖率。

Name	Class Type	Coverage	Goal	% of Goal	Status	Included
/mcd/mcd/mcdf_coverage		53.9%				
TYPE cg_mcdf_reg_write_read	mcdf_cover...	66.6%	100	66.6%		
CVP cg_mcdf_reg_write_read::addr	mcdf_cover...	50.0%	100	50.0%		
CVP cg_mcdf_reg_write_read::cmd	mcdf_cover...	100.0%	100	100.0%		
CROSS cg_mcdf_reg_write_read::cmdXaddr	mcdf_cover...	66.6%	100	66.6%		
TYPE cg_mcdf_reg_legal_access	mcdf_cover...	41.6%	100	41.6%		
CVP cg_mcdf_reg_legal_access::addr	mcdf_cover...	33.3%	100	33.3%		
CVP cg_mcdf_reg_legal_access::cmd	mcdf_cover...	100.0%	100	100.0%		
CVP cg_mcdf_reg_legal_access::wdata	mcdf_cover...	50.0%	100	50.0%		
CVP cg_mcdf_reg_legal_access::rdata	mcdf_cover...	100.0%	100	100.0%		
CROSS cg_mcdf_reg_legal_access::cmdXaddrXdata	mcdf_cover...	41.6%	100	41.6%		
TYPE cg_channel_disable	mcdf_cover...	61.1%	100	61.1%		
CVP cg_channel_disable::ch0_en	mcdf_cover...	50.0%	100	50.0%		
CVP cg_channel_disable::ch1_en	mcdf_cover...	50.0%	100	50.0%		
CVP cg_channel_disable::ch2_en	mcdf_cover...	50.0%	100	50.0%		
CVP cg_channel_disable::ch0_vld	mcdf_cover...	50.0%	100	50.0%		
CVP cg_channel_disable::ch1_vld	mcdf_cover...	100.0%	100	100.0%		
CVP cg_channel_disable::ch2_vld	mcdf_cover...	100.0%	100	100.0%		
CROSS cg_channel_disable::ch0Xch1Xch2	mcdf_cover...	61.1%	100	61.1%		
TYPE cg_arbiter_priority	mcdf_cover...	25.0%	100	25.0%		
CVP cg_arbiter_priority::ch0_prio	mcdf_cover...	25.0%	100	25.0%		
CVP cg_arbiter_priority::ch1_prio	mcdf_cover...	25.0%	100	25.0%		
CVP cg_arbiter_priority::ch2_prio	mcdf_cover...	25.0%	100	25.0%		
TYPE cg_formatter_length	mcdf_cover...	29.1%	100	29.1%		
CVP cg_formatter_length::id	mcdf_cover...	33.3%	100	33.3%		
CVP cg_formatter_length::length	mcdf_cover...	25.0%	100	25.0%		
TYPE cg_formatter_grant	mcdf_cover...	100.0%	100	100.0%		
CVP cg_formatter_grant::delay_req_to_grant	mcdf_cover...	100.0%	100	100.0%		

## 合并覆盖率

你可以参考上面的仿真步骤，运行不同的仿真，或者运行同一个 test，它们都会生成独一无二的数据库。接下来，你就可以将之前统一在 COVERAGE\_STORAGE\_PATH 下面生成的 xxx.data 覆盖率数据做合并了。你可以在 Questasim 的仿真窗口中敲入命令。

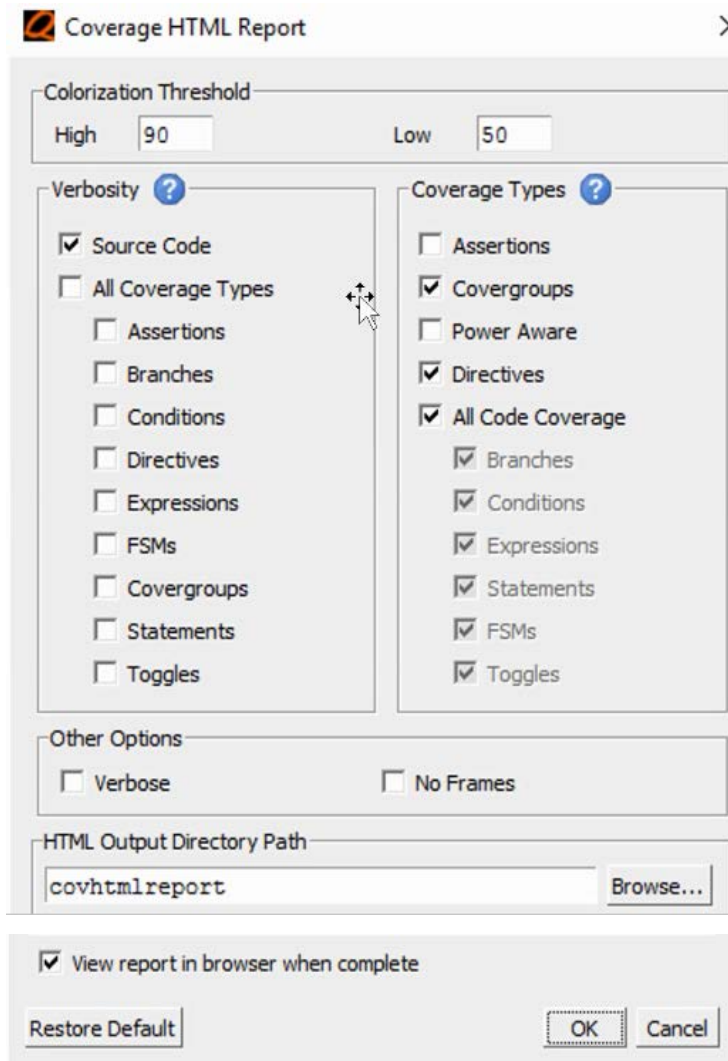
●vcover merge -out merged\_coverage.ucdb C:/questasim64 10.6c/examples

这里标注黄色的部分依然代表 COVERAGE\_STORAGE\_PATH。这个命令即是将你之前产生的若干个 xxx.data 的覆盖率合并在一起，生成一个合并在一起的覆盖率文件。所以，在测试前期，你提交的测试越多，那么理论上覆盖率的增长也就越明显。

接下来，你可以点击 File -> Open 来打开这个合并后的 UCDB 覆盖率数据库(注意选择文件类型 UCDB 就可以看到这个文件了)。当你打开这个数据库之后，你可以发现合并后的数据库要比之前单独提交的任何一个测试在仿真结束时的该次覆盖率都要高。例如你可以在 covergroups 窗口栏中查看功能覆盖率,也可以在 Analysis 窗口中查看代码覆盖率。

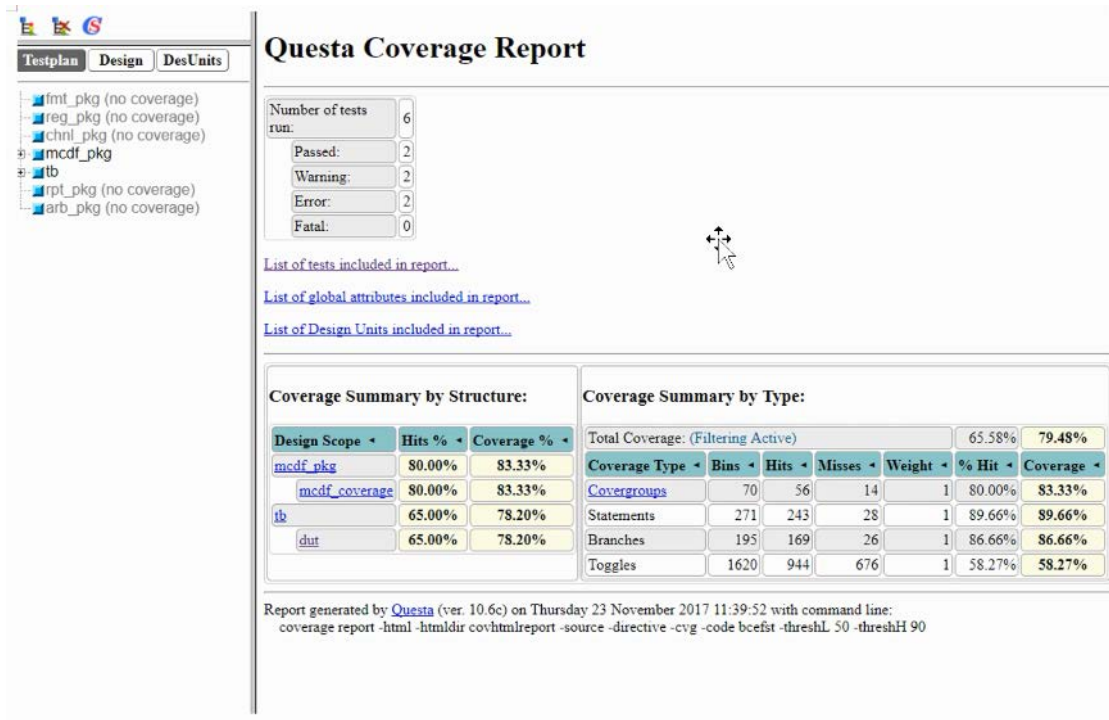
## 分析覆盖率

你可以依旧使用 Questasim 来打开 UCDB 利用工具来查看覆盖率，或者更直观的方式是在打开当前覆盖率数据库的同时，生成 HTML 报告。选择 Tools -> Coverage Report-> HTML，按照下图所示进行勾选：



单击 OK 后，Questasim 就会帮助你生成一份详尽的 HTML 覆盖率文档。由于我们勾选了这些内容：

- Covergroups
- Statements。
- Branches。
- Toggles。



所以接下来你需要让 DUT 的代码覆盖率和 mcdf coverage 的功能覆盖率达到我们最终的验收要求。你可以选用路桑的 mcdf\_full\_random\_test 或者复用你之前在实验 4 中实现的测试。当你发现测试无法再提高覆盖率时，你需要修改约束或者创建新的测试最终来达到我们验证完备性的要求。

祝你好运！

## UVM 入门和进阶实验 0

嗨，大咖们，在我们结束了 SV 的验证平台构建的学习之后，我们终于来到了 UVM 的世界。有没有一点兴奋呢？希望你们短暂的兴奋过后不要紧接着就是“无从下手”。就像我们在课堂上学习到的，UVM 整个验证结构的包如果要仔细钻研起来，真的要花不少的功夫。如果我们真得先从“一”开始学习，那么要学习到 UVM 验证平台搭建的精髓恐怕在有限的时间内还是难免差强人意。那么怎么办呢？这也是路桑在书中谈到，我们不会同其它 UVM 培训一样，将 UVM 解构得七零八碎，仔细教同学们知识点，因为这样仍然很难让大家在培训结束时，对 UVM 建立起完整的理念，深刻理解其优点。

所以，路桑在这里再次提醒大家，我们一开始就走的是高举高打的路子。至少在大家学习了 SV 的验证平台打发，掌握了框架建立的套路之后，已经能够对接下来 UVM 世界的理解输送基本弹药了。所以，我们不会围绕着各个语法点、特性来逐个解释，而是仍然按照学习 SV 时候的核心流程，即：

- 如何搭建验证框架
- 验证组件之间的连接和通信
- 如何编写测试用例，继而完成复用和覆盖率收敛 熟悉吗？没错！思路是一样的。所以

以同学们需要跟随这些核心点，在已经从 SV 的森林中

走出来之后，我们即将又要踏进 UVM 这篇茂密的雨林，那里有很多未知的、神秘的、令人惊奇的、各式各样的特性和组件在等待着大家呢。但是不管走到哪里，无论是课上听讲，还是课下实验，当你有点困惑、有点手足无措的时候，再将我们接下来的知识地图即上面的“核心流程”默念一遍，想一想我们在 UVM 的世界中，究竟是走到哪里了？哪怕是迷失，那也只是短暂性的。



我们的 UVM 入门和进阶实验 0 还是同之前 SV 验证实验 0 思想一样让大家通过简单的实验要求，在轻松愉悦的气氛下，踏出 UVM 世界的第一步，从而掌握下面的基本概念和仿真操作：

- 懂得如何编译 UVM 代码
- 理解 SV 和 UVM 之间的关系
- 了解 UVM 验证顶层盒子与 SV 验证顶层盒子之间的联系
- 掌握启动 UVM 验证的必要步骤。

## 编译

编译文件 `uvm_compile.sv`，待正常编译正常结束。在 work 库中仿真模块 `uvm_compile`，在命令窗口中敲入 `"run -all"`，可以观察到仿真输出语句：

```
VSIMS> run -all
#
#-----
# UVM-1.1d
# (C) 2007-2013 Mentor Graphics Corporation
# (C) 2007-2013 Cadence Design Systems, Inc.
# (C) 2006-2013 Synopsys, Inc.
# (C) 2011-2013 Cypress Semiconductor Corp.
#-----
#
# ***** IMPORTANT RELEASE NOTES *****
#
# You are using a version of the UVM library that has been compiled
# with `UVM_NO_DEPRECATED undefined.
# See http://www.eda.org/svdb/view.php?id=3313 for more details.
#
# You are using a version of the UVM library that has been compiled
# with `UVM_OBJECT_MUST_HAVE_CONSTRUCTOR undefined.
# See http://www.eda.org/svdb/view.php?id=3770 for more details.
#
# (Specify +UVM_NO_RELEASENOTES to turn off this notice)
#
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(215) @ 0: reporter [Questa UVM] QUESTA_UVM-1.2.2
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(217) @ 0: reporter [Questa UVM] questa_uvm::init(+struct)
# UVM_INFO /vobs/soc_db-sys-prj/MCDF/uvm_basic_labs/lab0/uvm_compile.sv(8) @ 0: reporter [UVM] Hello, welcome to REV UVM training!
# UVM_INFO /vobs/soc_db-sys-prj/MCDF/uvm_basic_labs/lab0/uvm_compile.sv(10) @ 1000: reporter [UVM] Bye, and more gifts waiting for you!
```

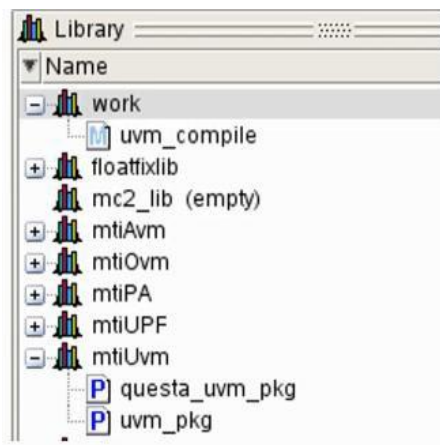
恭喜你，已经将 UVM 包导入了进来。在文件中的这两句接下来请你记住，无论在什么地方，我们的验证顶层都将需要这两句话：

```
import uvm_pkg::*;

`include "uvm_macros.svh"
```



这两句即代表着从预编译的 UVM 库(UVM 开源库+ Questa 的 UVM 定制部分库), 这一点可以在 Library view 中观察到:



在仿真开始后, 你将可以观察到一些欢迎语, 即 “UVM-1.1d” 的版本以及有哪些 EDA 行业大佬参与背书到其中了。如果你在使用更新的 Questasim 版本, 那么默认情况下, 你的 UVM 版本可能会显示 “UVM-1.2”。理论上呢, 越新的版本使用起来在后期需要更新知识的成本最小, 从 UVM-1.1d 到 UVM-1.2 还是有一些使用方法的变化, 不过暂时不会影响到同学们的正常使用。因为在我们接下来的实验部分的代码对于两个版本都是通用的, 不会存在版本兼容的问题。

接下来, 你可以看到我们使用了 UVM 的消息宏打印出来的消息, “Hell..”, “Bye...”, 那么这是不是意味着你已经 “进入” UVM 的世界了呢? 其实还不... 接下来的部分路桑会给你解释, UVM 世界的入口究竟在哪里。

```
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(215) @ 0: reporter [Questa UVM] QUESTA_UVM-1.2.2
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(217) @ 0: reporter [Questa UVM] questa_uvm::init(+struct)
# UVM_INFO /vobs/soc_db-sys-prj/HCDF/uvm_basic_labs/lab0/sv_class_inst.sv(15) @ 0: reporter [SV_TOP] test started
# UVM_INFO /vobs/soc_db-sys-prj/HCDF/uvm_basic_labs/lab0/sv_class_inst.sv(9) @ 0: reporter [SV_TOP] SV TOP creating
# UVM_INFO /vobs/soc_db-sys-prj/HCDF/uvm_basic_labs/lab0/sv_class_inst.sv(17) @ 0: reporter [SV_TOP] test finished
```

## SV 和 UVM 之间的关系

接下来, 添加 sv\_class\_inst.sv 文件, 编译, 仿真, 看看发生了什么? 实际上这个实

验是 SV 模块实验环节的抽象，它只是在顶层 module 容器要例化软件验证环境的顶层，即 SV class top. 在接下来的阶段，从打印出的信息可以看得出来，相当于从测试的开始，到

验证环境

的搭建，激励的发送，检查的执行等，最后又到了测试的结束。因此这是 SV 模块实验的

“一”，即一生二，二生三，三生万物的那个“顶层”。

我们接下来再添加 uvm\_class\_inst.sv, 编译, 仿真, 看看发生了什么? 从打印的信息来看，也是在模拟验证结构的创立，只不过这一次我们利用了 UVM 的类 uvm\_component 来定义了 top 类，继而在创建了这个“顶层”验证结构。那么这个顶层，算不算做是 UVM 环境呢？

```
UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(215) @ 0: reporter [Questa UVM] QUESTA_UVM-1.2.2
UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(217) @ 0: reporter [Questa UVM] questa_uvm::init(+struct)
UVM_INFO /vobs/soc_db-sys-prj/HCDF/uvm_basic_labs/lab0/uvm_class_inst.sv(18) @ 0: reporter [UVM_TOP] test started
UVM_INFO /vobs/soc_db-sys-prj/HCDF/uvm_basic_labs/lab0/uvm_class_inst.sv(11) @ 0: t [UVM_TOP] SV TOP creating
UVM_INFO /vobs/soc_db-sys-prj/HCDF/uvm_basic_labs/lab0/uvm_class_inst.sv(20) @ 0: reporter [UVM_TOP] test finished
```

先不着急下结论，同学们做到这里，请注意，我们所谓的 UVM 验证环境指的首先是提供一个 UVM 的“容器”，即接下来所有的 UVM 对象都会放置在其中，这样也可以成为 UVM 的顶层, 这就类似于之前 SV 模块实验中的顶层容器 test 一样。

## UVM 验证顶层与 SV 验证顶层的对比

我们再来看另外一个文件 uvm\_test\_inst.sv, 编译, 仿真, 再看看发生了什么? 似乎打印的信息比较多了, 如果我们将它划分的, 可以分为两个部分:

- 测试运行时的消息，这一些消息与之前的几个例子没有太多差别
- 仿真结束后的总结消息，这一些消息是之前例子没有的，至于其中的具体含义，我们可以在后续学习了更多的 UVM 知识后，带领大家解读。

```

# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(215) @ 0: reporter [Questa UVM] QUESTA_UVM-1.2.2
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(217) @ 0: reporter [Questa UVM] questa_uvm::init(+struct)
# UVM_INFO /vobs/soc_db-sys-prj/HCDF/uvw_basic_labs/lab0/uvw_test_inst.sv(22) @ 0: reporter [UVM_TOP] test started
# UVM_INFO /vobs/soc_db-sys-prj/HCDF/uvw_basic_labs/lab0/uvw_test_inst.sv(11) @ 0: uvw_test_top [UVM_TOP] SV TOP creating
# UVM_INFO @ 0: reporter [RIFTEST] Running test top...
# UVM_INFO /vobs/soc_db-sys-prj/HCDF/uvw_basic_labs/lab0/uvw_test_inst.sv(15) @ 0: uvw_test_top [UVM_TOP] test is running
# UVM_INFO verilog_src/uvw-1.1d/src/base/uvw_objection.svh(1268) @ 0: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO : 7
# UVM_WARNING : 0
# UVM_ERROR : 0
# UVM_FATAL : 0
#
# ** Report counts by id
# [Questa UVM] 2
# [RIFTEST] 1
# [TEST_DONE] 1
# [UVM_TOP] 3
#
# ** Note: $finish : /nfs/xa/proj/inway/eda/mentor/questasim/10.2e_3/questasim/linux_x86_64/./verilog_src/uvw-1.1d/src/base/uvw_root.svh(430)
# Time: 0 ns Iteration: 216 Instance: /uvw_test_inst

```

## 启动 UVM 验证的必要步骤

这个例子即 `uvw_test_inst` 显然与 `uvw_class_inst` 带给人的感官不同，如果路桑问你 哪个可以作为 UVM 验证顶层容器的话，你估计会选择 `uvw test inst` 吧。没错,我们之所以这样选择，是基于下面几处代码的不同以及它们所带来的影响所造成的:

- 只有继承于 `uvw test` 的类，才有“资格”可以作为 UVM 验证环境的顶层。（暂时不用问路桑为什么，但是请结合之前 SV 模块的实验常识，是不是觉得这个规定有它的考量在呢？）
- 创建顶层验证环境，有且只能依赖于 `run_test( "UVM_TEST_NAME" )`来传递，或者稍后你可以学习到可以通过仿真参数传递，而不是通过构造函数 `new()` 来实验的。尽管 `new()` 可以创建一个对象，但是不能做与顶层验证环境相关的其它工作。（这一点，我们也暂时了解到这里，因为我们还只是刚刚买票“入园参观”，对于其 UVM 世界的秘密我们还有很多很多需要学习。）

所以，同学们，你已经学习到进入 UVM 世界的“随意门”。利用路桑给你的套路，在接下来，无论我们要讲 UVM 关于工厂和它的特性，还是 UVM 验证组件的创建和连接，又或者其它的示例，我们都将利用这一“随意门”来进入 UVM 世界。至于这样做有什么神奇的功效，路桑会在 UVM 组件的介绍课堂中，跟大家进一步探讨 `uvw_test` 类的作用。

# UVM 入门和进阶实验 1

当带领大家进入一个大的景观园区，为了防止迷路，“小朋友”们最好还是不要乱跑，容易迷路的。我们接下来实验 1 的环节，将带领大家了解下面主要几个部分：

- 工厂的注册、创建和覆盖机制
- 域自动化以及 uvm object 的常用方法
- phase 机制
- config 机制
- 消息管理 是不是觉得已经很多了呢?坚持，坚持一下！因为只要趟过这一关，接下来的实验 2 就

要进入我们本次 UVM 模块学习的主线了，现在的铺垫极其重要，否则在进入主线学习的时候，可能会“消化不良”呢！路桑尽量让每一个特性在以下实验中独立开来，让大家在做每一个实验要求的时候，可以不被其它知识点所影响。所以，实验 1 的代码还是相对简单的，这有一点像 SV 模块的实验 0-3 的部分。不过接下来的实验 2 由于将要进入主线学习 UVM 的结构，这里会提醒大家，代码量可能会“暴增”，希望能在实验 2 的时候多花点时间预习一下呢。另外，推荐大家在不懂的地方，可以查看 UVM 的类手册 (class\_reference)或者红宝书。我们所有培训模块的主线，都是从红宝书移植过来的哦。

在接下来进行各项实验之前，我们需要掌握除了使用实验 0 中的 `run test( "UVM_TESTNAME" )` 之外，还有什么方法可以灵活地选择创建哪一个 UVM 测试用例，即我们可以在仿真时通过传递仿真选项+`UVM_TESTNAME= my test` 来实现。这一点使得既避免了多次修改代码和编译，也使得我们可以通过仿真时参数来控制 UVM 的架构创建。

## 工厂的注册、创建和覆盖机制

我们需要理解为什么需要注册，而在 UVM 世界中的注册一共有哪几种方式呢？只有两种用来注册的宏，请你记住它们：

- ``uvm_object_utils(T)`
- ``uvm_component_utils(T)`

也就是说，**凡是继承的 `uvm object` 类或者 `uvm component` 类**(不出意外的话，所有 UVM 世界中的类都是它们其中的一种)，**你都需要在它们的代码中选择一种宏并且声明，其差别只是需要你辨别该类是 `uvm_object` 的子类还是 `uvm_component` 的子类。**

打开实验文件 `factory_mechanism.sv`，编译，在仿真时敲入命令：

```
vsim -novopt -classdebug +UVM_TESTNAME=object_create
```

```
work.factory_mechanism
```

上述命令指的是我们将运行测试 `object_create`，注意观察代码中，是否所有的 UVM 类都已经用上述注册宏的一种来声明了呢？没错，所以**这是我们在本节实验中学习到的第一个“套路”**。接下来，我们先不解释为什么需要“注册宏”，而是进入创建和覆盖这两部分的实验要求，待你完成实验要求之后，我们可以再来回顾——为什么需要“注册宏”。

- 请按照实验 1.1 和 1.2 的具体要求，实现代码，并且运行 UVM 测试

`object_create` 和 `component_create`，观察打印信息，检查 `t2/t3/t4` 和 `u2/u3/u4` 是否被创建。

- 请按照实验 1.3 和 1.4 的具体要求，实现代码，并且运行 UVM 测试

`object_override` 和 `component_override`，观察打印信息，检查 `t2/t3/t4` 和 `u2/u3/u4` 的类型是否被替换，再考虑为什么 `t1` 和 `u1` 没有被类型覆盖呢？

在完成了关于“创建” UVM 对象和“覆盖” UVM 类之后，我们还可以再来做以下尝

试，帮助同学进一步认识“注册宏”的作用：

- 请移除 trans 类型的`uvm\_object\_utils()`注册宏说明，再进行编译仿真，观察是否有编译或者运行错误？请思考大致为什么？
- 同样地，请移除 unit 类型的`uvm\_component\_utils()`注册宏说明，再进行编译仿真，观察是否有编译或者运行错误？请思考大致为什么？

## 域自动化和 uvm\_object 的常用方法

到了域自动化的环节，我们需要了解常见的域声明相关的宏，以及在声明了之后，uvm\_object 类所具备的常见方法。在这个实验环节中，我们主要就一些常见标量的域声明宏做以练习，继而再对 uvm\_object 提供的几种常见方法和其回调函数(callback) 加以掌握。接下来，请在文件 uvm\_object\_methos.sv 中完成以下实验要求：

- 实验 2.1，请学习使用域自动化的宏方法。可参考红宝书表 10.2 和表 10.3。
- 实验 2.2，请学习 uvm object:compare() 方法。
- 实验 2.3，请尝试掌握 uvm pkg. 中常见的一些全局控制对象，例如 uvm\_default\_comparer，该对象可参考 uvm\_comparer 类提供的方法。具体还包括哪些全局对象，可以参考红宝书图 10.7 uvm .pkg 的全局对象。
- 实验 2.4，请学习自定义 uvm object 的一些回调函数，例如 do. compare()，并且理解预定义函数 compare() 与 do. compare() 的调用顺序和关系。
- 实验 2.5、2.6，请学习 uvm\_object::print() 及 uvm\_object::copy() 函数，再结合之前的 compare() 函数，理解域自动化的意义以及它带来的便捷性。

注意，此时应在仿真器中输入指令：

```
vsim -novopt -classdebug +UVM_TESTNAME=object_methods_test  
work.object_methods
```

## Phase 机制

这一部分实验可以参照红宝书 10.5 节 phase 机制。phase 机制使得验证环境从组建、到连接、再到执行得以分阶段执行，按照层次结构和 phase 顺序严格执行，继而避免一些依赖关系，也使得 UVM 用户可以正确地将不同的代码放置到不同的 phase 块中。请在文件 phase\_order.sv 中完成以下实验要求：

- 实验 3.1，请参考 comp1 类定义的几个主要 phase 方法，也分别对类 comp2 和 comp3 定义相应的 phase 方法。在完成后，可运行 phase\_order\_test，观察按照层次结构，phase\_order.test、comp1、comp2 和 comp3 在不同 phase 阶段所执行 phase 的先后顺序。

- 实验 3.2，请完成 phase\_order\_test 的另外 2 个细分的 phase，即 reset\_phase 和 main\_phase，可以参考其 run\_phase 的实习方法，并且也分别耗时 1us。然后再运行 phase\_order\_test，再次观察各个组件在执行不同 phase 时的顺序，对于 phase\_order\_test 而言，由于引入了 run\_phase 并行开始执行的 reset\_phase 和 main\_phase，这会使得仿真最终在什么时间结束？为什么在此时结束呢？

在仿真器中输入指令：

```
vsim -novopt -classdebug +UVM_TESTNAME=phase_order_test  
work.phase_order
```

## config 机制

这一部分实验可以参照红宝书 10.6 节 config 机制。我们在本次实验中，也将分别完成以下几种传递方式：

- 接口传递
- 单一变量传递
- 对象传递

请在文件 `uvm_config.sv` 中完成以下实验要求：

- 实验 4.1，请完成接口从 `uvm_config` 模块到验证环境中的传递，使得 `c1` 和 `c2` 可以得到接口，并且检查接口是否最终得到。

- 实验 4.2，请完成配置对象 `config.obj` 从 `uvm_config.test` 到 `c1` 和 `c2` 的传递。
- 实验 4.3，请完成在顶层 `uvm_config_test` 对 `c1.var1` 和 `c2.var2` 的变量设置。
- 实验 4.4，请分别进一步思考如下问题：

。接口传递与 `run_test()` 之间是否存在顺序？

。在 `uvm_config_test::build_phase()` 中，如果将 `c1` 的例化提前到刚进入 `build_phase()` 中时，而将后续 `config_db` 传递的操作放置于其后，是否可行？为什么？

。通过上述两个问题，你认为 `config.db` 在使用时，需要注意什么地方？

在仿真器中输入指令：

```
vsim -novopt -classdebug +UVM_TESTNAME=uvm_config_test
```

```
work.uvm_config
```



## 消息管理

请同时参考红宝书 10.7 节消息管理，在已经初步掌握消息宏`uvm\_info()`、`uvm\_warning()`、`uvm\_error()`以及`uvm\_fatal()`的同时，也需要学习，如何在后期验证环境较为稳定的时候，减少不必要消息的打印，以此来协助提高仿真速度。请在文件`uvm\_message.sv`中完成以下要求：

- 实验 5.1，请使用消息过滤方法`set\_report\_verbosity\_level\_hier()`在`uvm\_message\_test::build\_phase()`中屏蔽所有层次的消息，也就是不允许有任何`uvm\_message\_test`及其以下组件的消息在仿真时打印出来。
- 实验 5.2，请先注释实验 5.1 的代码，继而转用`set\_report\_id\_verbosity\_level\_hier()`来过滤以下 ID 的消息，即“BUILD”、“CREATE”和“RUN”。待方法使用之后，请与实验 5.1 打印消息的结果进行对比，检查两种方式打印结果一致。
- 实验 5.3，也许你已经发现，`config\_obj`的“CREATE”消息以及`uvm\_message`的“TOPTB”消息依然被打印了出来，请思考这是为什么？是否之前 5.1 和 5.2 提供的方法也可以屏蔽这些消息？如果不可以，那么是否有其它办法可以屏蔽这些消息？请尝试使用`uvm\_root::get()`来获取最顶层的（即`uvm\_message\_test`的顶层）来控制过滤“CREATE”和“TOPTB”的消息。

在仿真器中输入指令：

```
vsim -novopt -classdebug +UVM_TESTNAME=uvm_message_test  
work.uvm_message
```

## UVM 入门和进阶实验 2

同学们进入了实验 2，也依然会遇到“代码量暴增”的情况。不过还好，我们已经经过了 SV 模块实验 3 到实验 4 代码暴增的同样情况，这点心理承受能力还是有的。而且，相比较于 SV 而言，UVM 在实验 2 的代码，你一定非常熟悉！为什么这样讲呢？因为它们几乎是全部从 SV 最后的实验 5 移植过来的，它们包括了：

- SV 验证环境结构
- SV 组件之间的通信管道
- SV 的激励产生和发送模式
- SV 的数据检查和报告
- SV 的测试开始与结束方式
- SV 的配置方式

如果你对上面的一些要素还不熟悉，请再通读 SV 模块的实验 5 代码，分别对照上述的元素。之所以要求大家做这样的回顾，是因为我们从实验 2 开始一直贯穿到 UVM 入门模块的实验 5，会将之前 SV 的实验代码逐渐利用 UVM 所学的知识点，完全按照 UVM 的特性，最终移植为纯粹的 UVM 测试。那么这么设计实验的原因在哪儿呢？对学员的帮助有什么呢？用处太大了！

- 在 SV 的实验部分，我们做的是“加法”，即代码量不断增加，逐一完善为 SV 的“纯软件”类型的灵活验证环境。
- 在 UVM 的实验部分，我们做的是“减法”，即将搬迁来的 SV 验证环境中上述的验证环境元素，逐一替换为 UVM 的特性，继而让同学们在亲自动手实验的过程中，能够充分比较和理解 UVM 的特性，以及与 SV 对应特性相比，如此替换

的优势在什么地方。

- 由上述两点可以看出， 我们的实验代码部分在接下来的试验中并不会“暴增”，  
而只是做 UVM 对应特性的替换。在替换过程中， 我们除了需要按照实验要求完成代码， 还应该思考， SV 与 UVM 在验证环境上不同的实验方式， 究竟孰优孰劣？

那么， 我们接下来就进入到我们 UVM 入门的实验“正餐”， 关于 SV 完整验证环境的移植。我们本次实验中， 需要将之前学习到的以下知识充分应用和理解。它们包括：

- 各个验证组件的使用
- 验证组件之间的层次关系
- 工厂的“注册”和“创建”
- “域自动化”和 uvm object 的预定义方法
- “phase” 的自动执行和顺序关系
- 消息宏的简单使用
- 通过 config\_db 对接口的传递
- 测试的选择和开始， 以及对仿真结束的控制 看起来需要掌握的有点多， 不过呢，  
路桑为了让大家能够抓住重点， 顾大局而不拘小

节， 已经实现完成了初步的结构改造， 鼓励大家对路桑的问题加以思考， 稍后我们也会在实验代码回顾环节， 帮助大家做详细的解答。那么， UVM 世界的“构建之门” 就为大家敞开了， 注意喽， 这次你不再仅仅是“观光客” 去感受 UVM 的世界观（实验 1）， 而是要真正在已经初步完成 SV 环境到 UVM 环境的迁移过程中， 去认真审视， UVM 世界与 SV 世界的不同。

## 验证组件和层次构建

我们首先将各个 package 中的 SV 组件替换为 UVM 组件，在替换过程中，我们需要遵循以下基本规则：

- 实现组件对应原则：
  - 。 SV 的 Transaction 类对应 uvm sequence item。
  - 。 SV 的 driver 类对应 uvm driver 。
  - 。 SV 的 generator 类我们稍后会替换为 uvm sequence + uvm sequencer。
  - 。 SV 的 monitor 对应 uvm monitor。
  - 。 SV 的 agent 对应 uvm. agent.
  - 。 SV 的 env 对应 uvm env.
  - 。 SV 的 checker 对应 uvm scoreboard.
  - 。 SV 的 reference model 和 coverage model 均对应 uvm component.
  - 。 SV 的 test 对应 uvmtest
- 在遵循以上对应原则的过程中，我们在进行类的转换时，需要注意：
  - 。 SV 的上述类均需**继承于其对应的 UVM 类**
  - 。 在类定义过程中，一定需要使用 ``uvm_object_utils()` 或者 ``uvm_component_utils()` **完成类的注册**。至于注册时，该使用哪一个类，需要清楚各个类分别是 uvm\_object 类还是 uvm\_component 类。因此，类的地图可以在红宝书的图 10.1 中找到，请大家将扫描放大，放在你学习工作环境最显眼的地方，路桑当年学习日语音阶的时候，也是遵照这个方法。UVM 的类确实错综复杂，有了路桑这张类库地图，闯荡验证江湖就不那么胆怯喽。 **所以这一点，即指的是 UVM 类的注册需要遵循的规则。**

。在使用上述工厂注册宏的时候，会伴随着“域声明自动化”，一般而言，我们建议将 sequence item 定义时，应当伴随其域声明，即利用 ``uvm_object_utils_begin` 和 ``uvm_object_utils_end` 完成。这是由于对于 sequence item 对象的拷贝、比较和打印等操作比较多，因此建议在定义 sequence item 类时，也完成域的自动化声明。

。UVM 初学者，一定要注意构造函数 `new()` 的声明方式。请不要试图在构造函数上“玩花样”。路桑指的是，uvm\_object 的相关类，它的 new 函数参数只有一个即 string name，而 uvm\_component 的相关类它的 new 函数参数只能有两个，即 string name 和 uvm\_component parent，除此以外，增加或者减少参数都是非法的哦。所以这一点，即 UVM 类需要遵循的构造函数定义的规则。

。在组件之间的层次关系构建中，我们依然按照之前 SV 组件的层次关系。因此，通过保留这些层次关系，只需要稍后在不同的 phase (阶段) 完成组件的例化和连接即可。这一点，我们将在稍后的 phase 阶段实验思考环节进一步说明。

因此，请同学们按照上述的转换方式，参考 SV 实验 5 的代码和路桑给出的代码，进行对应和参照，思考上述 SV 到 UVM 的代码迁移过程。

## 测试的开始和结束

从这一部分开始，我们可以再次加深对测试的开始、环境构建的过程、连接以及结束的控制。请同学们就以下几点，完成对 SV 迁移到 UVM 环境的思考：

- 在 tb.sv 文件中，路桑已经注释了之前关于 SV 各个 test 声明、例化、外部参数传递、执行选择过程以及开始测试的过程。同时，路桑添加了与 UVM 相关的语句：

### 。通过 uvm\_config\_db 完成了各个接口从 TB (硬件-侧)到验证环境

**mcd\_f\_env(软件-侧)的传递。**这也很好地实现了以往 SV 函数的剥离，即 UVM 用户不再需要深入到目标组件一侧，调用其 set\_interface()即可完成传递。这种传递方式有赖于 config\_db 的数据存储和层次传递特性。而在 mcd\_f\_env 中，路桑暂时保留了 mcd\_f\_env 的 set\_interface()以及其各个子组件的 set\_interface()函数。所以，可以看得出来，路桑改造的仅仅是在 TB 与 mcd\_f\_env 之间的接口传递，然而理论上，我们可以移除所有的 set\_interface()函数，完全使用 uvm\_config\_db\_set 和 get 方法，从而使得 mcd\_f\_env 与其各个子组件之间也实现“层次剥离”，这样也就进一步促进了组件之间的独立性。

- 。调用 run\_test()函数即完成了 test 的选择、例化和开始测试。这也是红宝书 10.5.2 节如何开始 UVM 仿真中所阐述的部分，因此通过对照 SV 代码，大家可以进一步理解为何 run\_test()如此重要，它提供的便利包括：

- 用户可以在代码中指定 UVM test,或者为了避免代码频繁修改，可以通过 + UVM\_TESTNAME= mytest 在仿真选项中灵活传递 test 名。
- 在 run\_test()执行中，它会初始化 objection 机制，即查看 objection 有

没有挂起的地方，因此，在 test 或者 generator 中必须至少有一处地方使用 phase.raise. objection(来挂起仿真，避免仿真退出，而在仿真需要结束时，使用 phase.drop. objection()来允许仿真可以退出。请在 MCDF test 中查找这两个函数的调用，理解对应 test 将在何时结束，并且试着注释掉这两句话，看看将会有有什么惊喜等着你，试着去解释一下原因。

- **创建 uvm test 组件，及其以下的各层组件群。**

- **调用 phase 控制方法，按照所有 phase 按照顺序执行。**这一点，我们在之前的实验 1 已经做了代码练习，请大家再认真思考，uvm\_component 类中的 build\_phase 与 new函数相比，它们之间的关系是什么？有先后顺序吗？有包含顺序吗？另外，new 函数中可以执行哪些语句，而 build\_phase 函数又适合执行哪些语句？

- 在 build\_phase 中需要注意，凡是 UVM 类,均应该使用

"T:type\_id:create()"的方式完成创建，这也为后来工厂的类型覆盖(override)提供了方便。

- 此外，关于 connect\_phase 函数的练习，我们在本试验中做的不多，更

多的代码实现将在实验 3 即 TLM通信中实现。同学们可以先在

Mcdf\_checker 以及 mcdf\_env 中观察其 connect\_phase 函数，**理解**

**Connect\_phase 中需要做的动作是什么？**那么与之前 SV 的 new 函数相比，其既做了对象的例化，又做了对象的连接,然而在 UVM 中，将对象的例化放置在 build\_phase 中，而将对象的连接放置在 connect\_phase 中，这么做的好处在哪里呢？

- 除了我们将 SV 中 new()函数的对象例化和连接分别移植到 UVM 的

build\_phase 和 connect\_phase 函数以外，我们也对 SV 组件的 run 任务做了小的改动，可以观察到的细微变化即是将其改为 run\_phase 任务。除了名字上的细小差别，在学习了，上一节 phase 控制以后，同学们需要认识到，UVM 的顶层(uvm\_root)会控制所有 phase 的执行顺序，而且各个 phase 不再需要手动执行了。这意味着，之前在 SV 中，我们需要从 test 到 env 再到各个 agent，一层层地调用 run 任务，继而“一盏盏地点亮”组件，与这种稍显笨拙的方式相比，UVM 做了什么？在整个 phase 调度方面，它就是一个贴心的管家有木有？完全不再需要你操心了呢！



## UVM 入门和进阶实验 3

同学们进入了实验 3，在跟路桑一起学习了 TLM 通信、同步通信原件以后，我们又将会之前实验 2 已经完成的验证环境结构的基础上，继续改造。在实验 3 中，我们将 SV 环境移植到 UVM 的重点将主要在以下几个方面：

- TLM 单向通信端口和多向通信端口的使用。
- TLM 的通信管道。
- UVM 的回调类型 `uvm_callback`。
- UVM 的一些仿真控制函数。

### TLIM 单向通信和多向通信

在之前的 monitor 到 checker 的通信，以及 checker 与 reference model 之间的通信，都是**通过 mailbox 以及在上层进行其句柄的传递实现的**。我们在接下来的实验要求中，需要大家使用 TLM 端口进行通信，做逐步的通信元素和方法的替换。同时，路桑在代码中也做了实验要求的注释，方便大家再指定的地方按照要求实现代码。

如图 1 所示，大家可以发现，我们将会 monitor、checker 和 reference model 上面添加若干 TLM 的通信端口，而其端口类型我们在实验代码中也有具体要求。接下来请逐步按照下列要求完成：

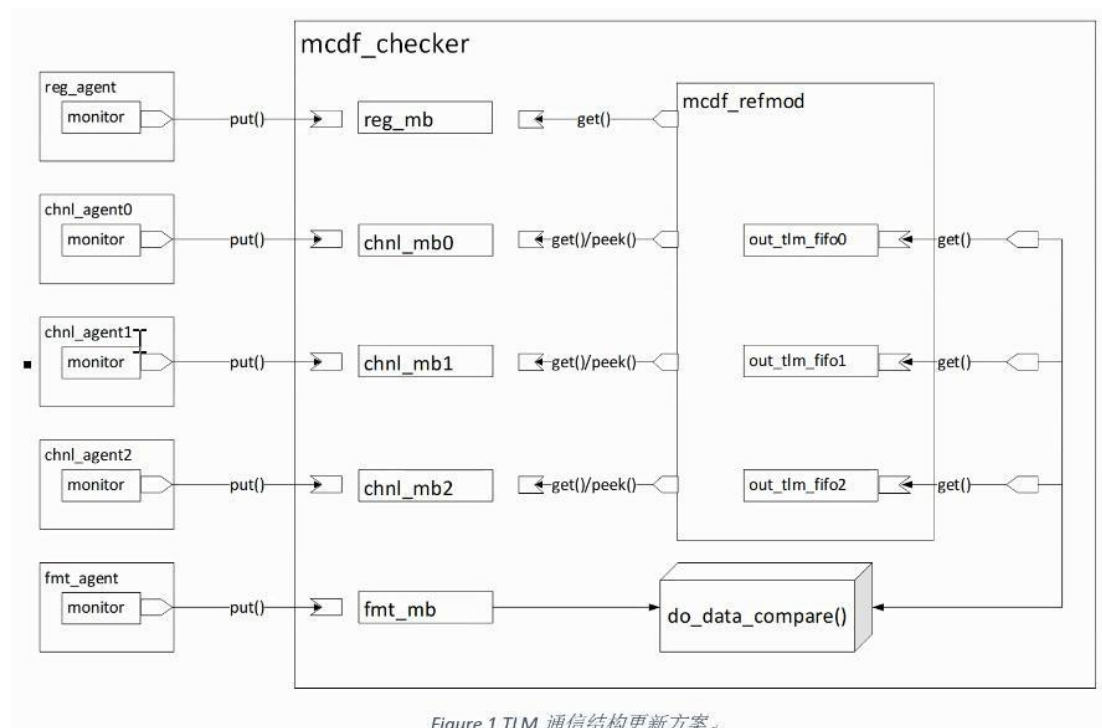
1. 请将在 **monitor 中的用来与 checker 中的 mailbox 通信的 `mon_mb` 句柄**替换为对应的 `uvm_blocking_put_port` 类型。
2. 在 checker 中声明与 monitor 通信的 **import 端口类型**，以及与 reference model 通信的 **import 端口类型**。具体类型可以参考代码中的注释，需要注意的是，由于 checker 与多个 monitor 以及 reference\_model 通信，是典型的多方

向通信类型，因此，我们需要使用多端口通信的宏声明方法，请参考红宝书 12.2.3 的实例。在使用了宏声明端口类型之后，再在 checker 中声明其句柄，并且完成例化。

如图 1 所示，请继续在 mcdf\_refmod 中声明用来与 mcdf\_checker 中的 import 连接的端口，并且完成例化，同时注意其端口类型的区别。关于端口类型，可以参考红宝书表 12.1。在完成声明和例化之后，请继续将原来的 mailbox 句柄调用 方法的方式，改为用 TLM 端口呼叫方法的方式。

请在 mcdf\_env 的 connect\_phase() 阶段，完成 monitor 的 TLM port 与 mcdf\_checker TLM import 的连接。

请在 mcdf\_checker 的 connect\_phase() 阶段，完成 mcdf\_refmode 的 TLM port 与 mcdf\_checker 的 TLM import 的连接。



## TLM 通信管道

在完成了上述实验之后，你可能会抱怨，看起来工作量增加了不少呢！怎么会说，

TLM 通信有它的好处呢？那路桑再阐述几个 TLM 通信的优点：

- 通信函数可以定制化，例如你可以定制 `put()/get()/peek()` 的内容和参数，这其实比 mailbox 的通信更加灵活。
- 将组件实现了完全的隔离，可以参考红宝书图 12.4，因为只有通过层次化的 TLM 端口连接，我们就可以很好地避免直接将不同层次的数据缓存对象的句柄进行“空中传递”。而 TLM 端口按照层次的连接，虽然看起来有点繁复，但也正因为这一点，可以使得组件之间保持很好的独立性呢。

那么有没有既可以使用 TLM 端口，又不用像上述实验需要自己实现具体的 `get()/peek()/put()` 方法呢？当然有啦！依然可以参考红宝书 12.3.1 节，关于 `uvm_tlm_fifo` 类的使用。请按照以下要求，完成本实验：

1. 将原本在 `mcdf_refmod` 中的 `out_mb` 替换为 `uvm_tlm_fifo` 类型，并且完成例化，以及对应的变量名替换。
2. 将原本在 `mcdf_checker` 中的 `exp_mbs[3]` 的邮箱句柄数组，替换为 `uvm_blocking_get_port` 类型句柄数组，并且做相应的例化以及变量名替换。
3. 在 `mcdf_checker` 中，完成在 `mcdf_checker` 中的 TLM port 端口到 `mcdf_refmod` 中的 `uvm_tlm_fifo` 自带的 `blocking_get_export` 端口的连接。

在完成这个实验环节之后，请开始编译原有的仿真测试，进行仿真，检查仿真结果是否与实验 2 的结果保持一致。另外，请在思考，上述两个实验环节中，针对一般的数据存储和 TLM 端口连接，那一种方式更为简便？

## UVM 回调类

接下来，我们将练习 `uvm_callback` 的定义、链接和使用方式。由此，我们可以将原有的 `mcdf_data_consistence_basic_test` 和 `mcdf_full_random_test` 的类实现方式（即类继承方式）修改为回调函数的实现方式，帮助同学们认识，完成类的复用除了可以使用继承，还可以使用回调函数。请按照以下要求实现代码：

1. 请在路桑给的 `uvm_callback` 类中，预先定义需要的几个虚方法。
2. 请使用 `callback` 对应的宏，完成目标 `uvm_test` 类型与目标 `uvm_callback` 类型的关联。
3. 请继续在目标 `uvm_test` 类型指定的方法中，完成 `uvm_callback` 的方法回调指定。
4. 请分别完成 `uvm_callback` 和对应 `test` 类的定义：
  - a. `cb_mcdf_data_consistence_basic` 和 `cb_mcdf_data_consistence_basic_test`
  - b. `cb_mcdf_full_random` 和 `cb_mcdf_full_random_test`

在完成上述代码后，可以指定经过修改的 `test` 类

`cb_mcdf_data_consistence_basic_test` 和 `cb_mcdf_full_random_test`，并且与其之前对应的两个类 `mcdf_data_consistence_basic_test` 和 `mcdf_full_random_test` 做比较，由此加深理解——`uvm_callback` 的方式是如何协助完成类的复用的。同时也可以对比之前 SV 模块的 `callback` 方法实现，体会 `uvm_callback` 的优势在什么地方？

## UVM 仿真控制方法

我们也可以回顾实验 1 对 `uvm_root` 类的应用，学习更多关于 `uvm_root` 类的方法。

请按照以下实验要求实现代码：

1. 请在 `mcdf_base_test` 类中添加新的 phase 函数 `end_of_elaboration_phase()`。

同时利用 `uvm_root` 类来将信息的冗余度设置为 `UVM_HIGH`，以此来允许更多低级别的信息打印出来。另外，请 `uvm_root::set_report_max_quit_count()` 函数来设置仿真退出的最大数值，即如果 `uvm_error` 数量超过其指定值，那么仿真会退出。该变量默认数值为 -1，表示仿真不会伴随 `uvm_error` 退出。

2. 请利用 `uvm_root::set_timeout()` 设置仿真的最大时间长度，同时由于此功能的生效，可以清除原有方法 `do_watchdog()` 的定义和调用。

哇哦，到这里路桑要恭喜你呢，我们可是胜利在望啊！下一次实验，我们将会结合对于 `sequence item`, `sequence`, `sequencer` 和 `driver` 的理解，来将目前验证结构中的 `generator` 和 `driver` 之间的结构关系，修改为 UVM 的结构和序列，一起期待它吧！

## UVM 入门和进阶实验 4

我们在完成了实验 3 之后，大家已经可以看到，monitor、reference model 与 checker 之间的通信是通过 TLM 端口或者 TLM FIFO 来完成，相较于之前的 mailbox 句柄连接，更加容易定制，也使得组件的独立性提高。接下来，我们要在实验 4 中完成以下内容：

- 将产生 transaction 并且发送至 driver 的 generator 组件，拆分为 sequence 与 sequencer
- 在拆分的基础上，实现底层的 sequence。完成 sequencer 与 driver 的连接和通信工作。
- 构建顶层的 virtual sequencer。
- 将原有的 mcdf\_base\_test 拆分为 mcdf\_base\_virtual\_sequence 与 mcdf\_base\_test，前者发挥产生序列的工作，后者只完成挂载序列的工作。
- 将原有的 mcdf\_data\_consistence\_basic\_test 和 mcdf\_full\_random\_test 继续拆分为对应的 virtual\_sequence 和轻量化的顶层 test。

由于上述要求均是整体服务于我们的最终目标，即将 generator、driver 与 test 的关系最终移植为 sequence、sequencer、driver 和 test 的关系，可谓是一个完整的移植过程，因此我们本实验的目标聚焦为 sequence 的使用。

### driver 与 sequencer 的改建

(实验 1.1) 移除原有在各个 driver 中的 mailbox 句柄，以及在 do\_driver() 任务中使用 mailbox 句柄通信的方式，转而用 uvm\_driver::seq\_item\_port 进行通信。同时，请定义对应的 uvm\_sequencer。

## 底层 sequence 的提取

(实验 1.2)请将原来在各个 generator 中发送 transaction 的任务，提取为各个对应的底层 sequence 这里需要特别注意，将 mcd\_f\_base\_test 中的 idle\_reg()、write\_reg()和 read\_reg()也可以提取并在 reg\_pkg 中定义为对应的 sequence，即 idle\_reg\_sequence、write\_reg\_sequence 和 read\_reg\_sequence。

## sequencer 的创建和连接

(实验 1.3)在各个 agent 中声明、创建对应的 sequencer，并且将其与 driver 通过 TLM port 连接起来。

## 移除 generator 的踪迹

(实验 1.4) generator 这个 SV 实验阶段的历史产物，就要从我们的验证结构中清除了。请在 mcd\_f\_base\_test 中移除它们的声明、创建以及和 driver 之间的连接。

## 移除 uvm\_base\_test 的 transaction 发送方法

(实验 1.5)请将已经被从 uvm\_base\_test 移植到 reg\_pkg 中的方法 idle\_reg()、write\_reg()和 read\_reg()从 uvm\_base\_test 中移除。由此可以看到，uvm\_base\_test 只变成了容器的性质，在它内部主要由 mcd\_f\_env 将来添加的 mcd\_f\_config 配置对象以及被用来挂载的顶层 sequence 构成。

## 添加顶层的 virtual sequencer

(实验 1.6)由于 MCDF 验证环境中存在多个底层的 sequencer 和 sequence，因此这

就需要有顶层的 virtual sequencer 与 virtual sequence 统一调度，可以参考红宝书

13.5.2 节。请实现 MCDF 的顶层 virtual sequencer.

(实验 1.7)在定义了 mcdf\_virtual\_sequencer 之后，请将其在 mcdf\_env 中声明、例化，并且完成其与底层 sequencer 的句柄连接。

### 重构 mcdf\_base\_test

(实验 1.8)原有的 mcdf\_base\_test 除了承担其容器的功能，还在其 run\_phase 阶段中实现了 sequence 的分阶段发送功能。我们在添加了顶层的 virtual\_sequencer 之后，**需要将所有发送序列的顺序和组织等内容均移植到 mcdf\_base\_virtual\_sequence，因此需要将 mcdf\_base\_test::run\_phase() 发送序列的功能移植到新定义的 mcdf\_base\_virtual\_sequence 一侧**，而在移植后，mcdf\_base\_test::run\_phase()只需要挂载对应的顶层 virtual sequence 即可。

### 重构 mcdf\_data\_consistence\_basic\_test

(实验 1.9)在理解了 mcdf\_base\_test 与 mcdf\_base\_virtual\_sequence 的关系之后，我们可以继续重构 mcdf\_data\_consistence\_basic\_test，将其产生和发送 transaction 的任务，均移植到 mcdf\_data\_consistence\_basic\_virtual\_sequence，而进一步减轻 mcdf\_data\_consistence\_basic\_test 的体重。由此大家需要加深认识，即测试的动态场景往往是由 virtual sequence 统一组织的，而 test 层往往之后做 run\_phase 前的一些验证环境的配置。



## 重构 mcdf\_full\_random\_test

(实验 1.10)可以继续参照实验 1.9 的要求，也将 mcdf\_full\_random\_test 的内容重构为 mcdf\_full\_random\_virtual\_sequence 和 mcdf\_full\_random\_test。

就是在这样一个完整的实验中，同学们可以掌握底层 sequence、sequencer 和 driver 之间的关系，以及复杂环境中的 virtual sequence、virtual sequencer 和 test 之间的关系。在完成了本次实验之后，我们在下一节将迎来 UVM 入门模块的最后一个实验，即将 UVM 寄存器模块移植到 MCDF 验证环境中，学习寄存器模块的常规使用方法，并且完成初步的测试用例移植。让我们一起期待吧！

## UVM 入门和进阶实验 5

嗨，我们终于携手走到了 UVM 入门的最后一个实验，关于 UVM 寄存器模块的应用。

通过对 UVM 寄存器模块的学习，我们将会在本次实验中掌握：

- 对 `uvm_reg` 的定义，以及 `uvm_reg_block` 的组织。
- 对 `uvm_reg_adapter` 的定义，以及它与 `uvm_reg_block` 之间的关系。
- 对 `uvm_reg_predictor` 的使用，以及它与 `uvm_reg_adapter` 和 `uvm_reg_block` 之间的关系。
- 改造之前的寄存器发送序列，并以 `uvm_reg` 的操作方式去取代。
- 应用内建的寄存器序列，做全面的寄存器测试。

### 寄存器模型的完善和嵌入

在接下来的实验中，同学们可以看到，`uvm_reg` 和 `uvm_reg_block` 的定义已经完成。

请结合红宝书 14.1 节，来理解 `mcdf_rgm` 寄存器模块的定义。同时，请继续完成以下的实验步骤：

- 实验 1.1，请实现 `reg2mcdf_adapter` 类的方法 `reg2bus` 以及 `bus2reg`，可参考红宝书 14.2.3 节。
- 实验 1.2，请在 `mcdf_env` 中分别声明 `register_block`，`adapter` 和 `predictor`，并完成例化。同时，在 `connect` 阶段中，请组织它们的关系，做必要的句柄连接。

## 寄存器模型的使用

寄存器模型很大的一个优势在于它的抽象性和复用性，接下来，我们将会改造之前的 激励序列。请按照以下要求完成实验：

实验 2.2，请将 `mcd_f_data_consistence_basic_virtual_sequence` 原有的由总线 `sequence` 实现的寄存器读写，改为由寄存器模型操作的寄存器读写方式，可参考红宝书 14.2.5 节。

实验 2.3，请将 `mcd_f_full_random_virtual_sequence` 原有的由总线 `sequence` 实现的寄存器读写，改为由寄存器模型预先设置寄存器值，再统一做总线寄存器 更新的方式，并且稍后由后门读取的方式取得寄存器值，加以比较。

## 寄存器内建序列的应用

参考红宝书 14.3.5 节，UVM 已经内建了一些寄存器序列，在接下来的实验中，我们将选择一些序列对寄存器展开全面测试：

- 实验 3.1，在新建的 `mcd_f_reg_builtin_virtual_sequence` 类中，请使用 `uvm_reg_hw_reset_seq`，`uvm_reg_bit_bash_seq` 和 `uvm_reg_access_seq` 对 MCDF 寄存器模块展开全面测试。在仿真过程中，请注意理解寄存器测试序列中打印的消息，对照红宝书表 14.6，理解这些内建寄存器测试序列的作用。

经过六次 UVM 入门和进阶实验的“艰苦作战”，UVM 新兵训练终于可以结束了！恭喜你，已经成为了一名新鲜出道的 UVM verifier。在短暂的兴奋过后，路桑还会将你送上“真枪实弹”的 UVM 实战现场，去零距离地感受一什么是真实的硬件设计 (MCDF 在设计上的升级版本)，以及什么是更加完善的验证环境(现有 MCDF 验证环境的升级)。在接下来的 UVM 实战模块中，我们将进一步降低授课时间而提高编程时间，目的仍然在于培养大家在“眼到”之后，可以“手到”和“心到”，知行合一。那么，就让我们在接下来的 UVM 实战模块中见吧！

# UVM 实战 1

在 UVM 实战部分中, apb\_pkg\_origin 文件夹下的文件是需要大家完善的。大家不妨先打开我们的 VIP 模板文件, 即 template\_pkg 文件夹下的文件先看一下, 你会发现 apb\_pkg\_origin 几乎没有添加其他太多的东西。当然这正是符合我们课程设计的, 在 UVM 入门部分的实验中, 我们将每周的知识点进行分解, 保证各知识点的独立性, 以此来巩固本周所学内容。而在实战部分, 则是更多的模拟实际工作场景, 因此我们要从零开发自己的 APB 总线 VIP, 模板只能给大家提供一个框架, 而具体的内容实现, 则留给读者自己去完成。

请大家按照以下实验步骤, 完成本次实验内容:

1、自己尝试在 apb\_pkg\_origin 文件夹下的 apb\_if.sv 以及 apb\_transfer.sv 文件中, 根据我们课上所讲的 APB 接口协议, 来定义 trans 中的内容, 并实现 apb 接口中变量和 master、slave、monitor 的时钟块定义。

2、在 apb\_master\_driver.svh 头文件总已经定了方法: get\_and\_drive()、drive\_transfer()、send\_idle()、reset\_signals(), 在 apb\_master\_driver.sv 中实现实现它们, 要求

- ①get\_and\_drive()不仅能够发送 trans, 还能够接收有返回值rsp。

- ②send\_idle()在每次读写操作结束时, 产生随机周期的空闲时间。

- ③根据 APB 协议, 复位时, reset\_signals()不必对 addr 和 pwrite 信号进行处理。

3、在 `apb_master_seq_lib.sv` 文件中，仿照 `example_apb_master_seq`，实现父类序列 `apb_master_base_seq`，与子类 `apb_master_single_write_sequence`、`apb_master_single_read_sequence`、`apb_master_write_read_sequence` 并完善其 `body` 内容，要求这三个子类 `sequence` 分别完成以下功能：

- ①完成一次写操作
- ②完成一次读操作
- ③完成一次先写后读的操作

4、继续创建两个子类 `sequence`，`apb_master_burst_write_sequence` 和 `apb_master_burst_read_sequence`，完成连续的写操作，和连续读操作。

5、实现 `apb_master_monitor` 中的 `collect_transfer` 方法。

6、为了模拟 `slave` 端连接的从设备，首先需要在 `apb_slave_driver.svh` 头文件中定义一个关联数组 `mem`，用它来模拟存储器，我们将对其进行写数据操作，和读数据操作。与 `master driver` 中的情况类似，请大家在 `apb_slave_driver.sv` 中实现，由 `svh` 头文件声明的方法和任务：`get_and_drive`、`drive_response`、`send_idle`、`reset_signals`。需要注意的是，在这个实验中，`slave` 的 `sequence` 暂时不起到控制作用，我们到下一个实验，会进一步对它进行升级。

7、新建文件 `apb_tests.svh`，在其中实现以下内容：

- ① `env` 和思考在此实验中，`env` 组件下包含哪些内容，需要创建哪些内容。

② 实现父类 test: apb\_base\_test。

③ apb\_base\_test\_sequence。首先声明一个关联数组 mem，后面会用它来储存 write 操作的数据，然后在其中实现以下方法：

- a. check\_mem\_data( bit[31:0] addr, bit[31:0] data ): 检查 mem 中对应地址的数据是否与 data 相同，如果比较成功，返回 1，否则返回 0。
- b. wait\_reset\_release: 等待复位完成。
- c. wait\_cycles(int n): 等待 n 个周期。
- d. get\_rand\_addr(): 产生一个随机的地址。并且地址的高 20 位，和低 2 位都为 0。

④ apb\_single\_transaction\_sequence。在这个 sequence 中需要完成以下测试：

- a. 仿真开始阶段，先等待复位完成，然后等待 10 个周期。
- b. 运行 single write，为方便调试，让发送的 data 等于 addr。为了方便对比，将发送的数据全部暂存在关联数组 mem 中。
- c. 运行 single read。并将读到的数据，与 mem 中暂存的数据进行对比。
- d. 先运行 single write，然后运行 single read。并比较写入数据是否正确。
- e. 运行 write read，并检查写入数据是否正确。

注意以上四个测试 sequence 分别运行次数 100 次，或者指定测试次数 test\_num 次。

⑤ apb\_single\_transaction\_test。完成 apb\_single\_transaction\_sequence 的创建与挂载，开始运行。

⑥ apb\_burst\_transaction\_sequence。Burst 模式下的收发数据测试序列，请注意与连续操作 single transaction 的不同之处。

⑦apb\_burst\_transaction\_test。与⑤相同，不再赘述。

完成以上测试内容之后，将全部文件添加到仿真软件中，只需要编译 apb\_pkg.sv 和 apb\_tb.sv 即可，思考为什么？然后在 library 中，找到 apb\_tb，并右键选择开始仿真。观察测试序列的仿真波形，是否与预期的结果一致。

本次实验需要大家完成的内容较多，大家可以在完成以上内容之后，与我们给出的参考答案进行对比，看看你与路桑的思路有什么区别呢？有精力的朋友，可继续完成覆盖率检测部分的代码。



## UVM 实战 2

本周的实战任务比较轻松，回顾上周跟 APB 总线协议相关内容，复习协议时序，完成我们课上要求的 assertion 检查与覆盖率收集。

### APB 总线协议的断言检查

①在 PSEL 为高时，PADDR 总线不可为 X 值。此断言检查作为实例已经在 apb\_if.sv 文件中编辑好了。需要同学们仿照这个 property，完成后续任务。

②在 PSEL 拉高的下一个周期，PENABLE 以应该拉高。

③在 PENABLE 拉高的下一个周期，PENABLE 拉低。

④在 PSEL 和 PWRITE 同时保持为高的阶段，PWDATA 需要保持不变。

⑤在下一次传输开始前，上一次的 PADDR 和 PWRITE 信号应该保持不变。此处应该留意，在 burst 模式和 normal 模式下，PSEL 和 PENABLE 信号分别应该是什么样的行为，并且应该如何去实现这个 property，使它在两种不同的模式下都可以正确检测。另，PADDR 和 PWRITE 两个信号的检测可以分成两个 property。

⑥在 PENABLE 拉高的同一个周期，PRDATA 也应该变化（在 READ 模式下）。

### APB 总线协议的断言覆盖率

①单独非连续写操作。这部分作为实例，请同学们仿真实例完成后续内容。

②连续写操作。

③对同一个地址，先做写操作再不间隔做读操作。

④对同一个地址，不间隔做连续两次写操作，在从中读数据。

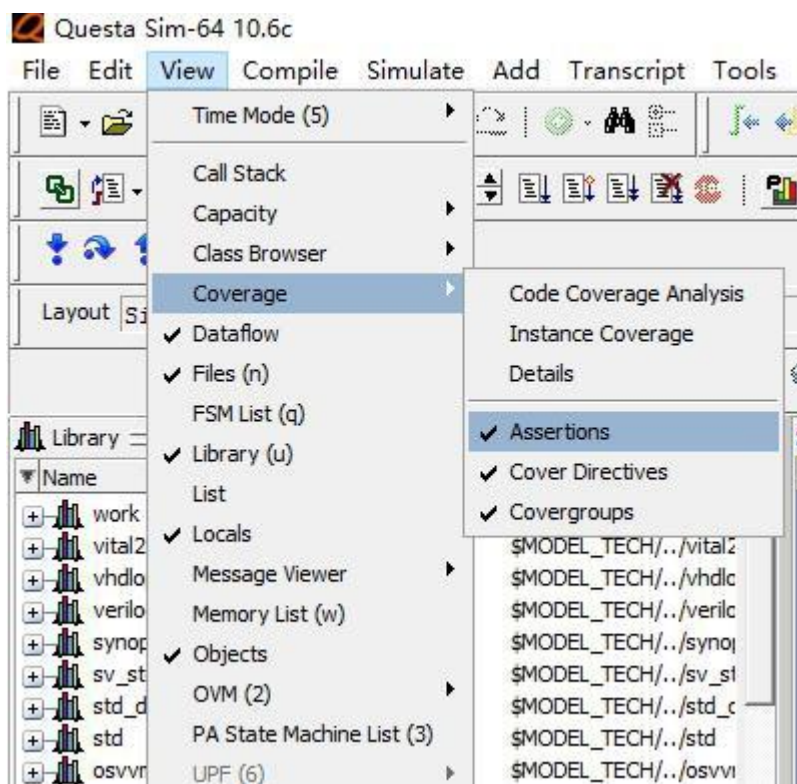
- ⑤单独非连续读操作。
- ⑥连续读操作。
- ⑦对同一个地址，连续做读写读操作。

与 UVM 实战 1 一样，只需要编译两个文件即可，在命令栏中输入

```
vsim -novopt -sv_seed random -assertdebug -assertcover -classdebug
```

+UVM\_TESTNAME=apb\_single\_transaction\_test work.apb\_tb，其他命令我们在之前的实验中都已经使用过，添加的命令-assertdebug 和-assertcover，分别用来调试 assertion 和收集其覆盖率。紧接着输入 log -r /\*，保存所有记录。

如下图所示，勾选 view->coverage->assertions/cover directives/covergroups。



输入命令 run -all开始仿真。

## UVM 实战 4

在之前的 SV 和 UVM 入门进阶实验中，大家可能因为工作原因，没时间仔细去完成每一个实验。但在 UVM 实战部分，这 4 个实战实验的内容，是大家在实际的验证工作中必然会使用到的，因此希望大家认真对待。

### UVM 的 C 测试环境

在子系统级验证环境中，主要由 UVM 完成测试序列，但从子系统集成到系统级测试时，更多的会用到 C 的测试用例。请大家按照以下步骤完成本次实战内容：

- 在 UVM package 中通过声明的 `mcdp_dpi_c_adapter` 句柄，实现对应的 `writew/readw/wait_cycles/report_info` 等方法。这些方法将在 C 来调用，也可以直接使用 SV 来使用。
- 练习在 UVM 的 package 中实现 UVM 一侧方法的 `export`，以及 C 侧方法的 `import`。
- 完成编译之后，注意从输出的文件中找到 UVM `export` 出的函数定义，并将其作为稍后 C 文件一侧的头文件(header file)。
- 创建 C 文件，编写函数 `dpi_c_thread`，将 SV 一侧的 `run_top_virtual_sequence` 中寄存器的 UVM 操作方式修改为 C 代码。即将 `mcdp_pkg.sv` 文件下，`run_top_virtual_sequence` 中注释的部分代码，填写到 `dpi.c` 文件中，主要 sv 与 c 语言语法上的区别。
- Linux 环境：在稍后的编译时，按照提示给入完整的编译指令，同时编译 SV 文件和 C 文件: `gcc -g -fPIC -shared dpi.c -o dpi.so`

Windows 环境：下载解压 questasim-gcc-4.5.0-w64vc12 按照云盘实验

文档要求 “SV 与 C 联步骤” 去编译运行仿真

- 启动 UVM 测试，在 UVM 和 C 两侧设置断点，理解 UVM 和 C 互相的调用过程：  
`vsim -novopt work.tb -sv_lib dpi +UVM_TESTNAME=mcdf_dpi_C_test`
- 在 UVM 波形一侧检查寄存器访问时序是否与 C 函数一侧的内容一致。

## 寄存器模型的深度应用

- 在我们之前自动生成的寄存器模型 `mcdf_rgm_pkg.sv` 文件下，`mcdf_rgm` 类中，补全其后门访问路径。
- （选做）随机选择测试某些寄存器的后门读写访问。这个测试可以手动新建一个测试 sequence，也可以选择使用 `mcdf_reg_builtin_virtual_sequence` 中的 `uvm` 寄存器模型内建测试序列进行测试。
- 在 `mcdf_full_random_virtual_sequence` 序列的 `task do_reg` 中，对寄存器模型做随机化，对指定域做配置，最后对其做 `update`。注意修改前后两种寄存器模型访问方法的区别，并且可以发现，后者具有更好的可读性。
- （选做）通过后门访问，直接去的某些硬件寄存器域的数值，与之前的配置做对比，检查寄存器配置是否成功。
- 理解寄存器模型中寄存器域(register field)是如何构成寄存器的，乃至整个寄存器模型。
- 思考如何通过索引方式来获取默写寄存器域所在的寄存器的地址、比特位等信息。
- （选做）继而在路桑给出部分代码的基础上，添加少量代码实现寄存器域值的前门读写，并且通过基本的单元测试(unit test)。

## 寄存器覆盖率

- 通过执行 `mcdf_reg_builtin_test`，来检查最终寄存器模型的覆盖率。
- 在 `adapter`和寄存器模型中设置断点，理解寄存器覆盖率收集的逻辑。
- 执行 `mcdf_reg_builtin_test`，分析寄存器覆盖率收集，理解哪些 `coverpoint` 被覆盖，哪些没有被覆盖。
- 通过分析寄存器覆盖率模型中，哪些数值无法被覆盖，继而更新寄存器文件。

## 总线解析——UVM RECORD 使用

在通常硬件时序调试中，我们看到的数据总线即将总线中的各个成员收集在波形窗口中，这给我们的调试带来了不便。

在 UVM 中我们可以通过在 `monitor` 中实现 `transaction record` 的功能，结合 UVM `record` 方法和 EDA 仿真器的支持，使得总线数据传输在时序和波形上的调试更为方便。

- 根据路桑给的随堂代码，跑仿真添加 `transaction` 到波形中，从波形窗口中理解 `transaction` 抽象级别的信息内容和调试的便捷性。
- 在 `tb.sv` 中设置  

```
uvm_config_db#(int)::set(uvm_rot::get(),"*", "recording_detail", 1);
```
- 在仿真时添加仿真选项 `vsim -novopt -classdebug -msgmode both -uvmcontrol=all work.tb`
- `log -r /*`
- `View -> UVM Details`，在 `SIM` 窗口中选中相应的 UVM 组件，继而在 UVM `Details` 窗口中选中对应的 `transactionstream`，右键点击添加到波形窗口，观察波形窗口中的 `transaction`。

## 总线解析——在线解析

结合 monitor 采样 bus transaction 和寄存器模型，在仿真时通过采样的总线信息，从寄存器模型中查找对应的寄存器，继而将读写信息、寄存器信息乃至域的信息都一并打印出来。

这种方式便于摆脱波形调试，在前期利用仿真信息(simulation log)即能够完成调试，这有利于多数不了解 UVM 验证环境的人可以展开有处理器在内的系统调试。

这种总线在线解析的方式与 transaction 波形展示的方式可以互相配合。

- 根据路桑给的代码，跑仿真注意仿真 log 文件，并且与对应的寄存器访问序列对应，检查解析出的信息是否一致，查找关键字“REGANA”。
- 阅读在线解析代码 mcdf\_bus\_analyzer::do\_reg\_analysis()，理解其解析的逻辑。
-