目录

1. 定宽数组、动态数组、关联数组、队列各自特点和使用

队列:队列结合了链表和数组的优点，可以在一个队列的任何位置进行增加或者删除元素。其通过[$]这样的符号进行申明: int q[$];

定宽数组:属于静态数组，编译时便已经确定大小。其可以分为压缩定宽数组和非压缩定宽数组:压缩数组是定义在类型后面，名字前面;非压缩数组定义在名字后面。Bit [7:0][3:0] name; bit[7:0] name [3:0];

动态数组:其内存空间在运行时才能够确定，使用前需要用 new[]进行空间分配。

关联数组:其主要针对需要超大空间但又不是全部需要所有数据的时候使用，类似于 hash，通过一个索引值和一个数据组成: bit [63:0] name[bit[63:0]];索引值必须是唯一的。

2. 多线程 fork join、fork join_any、fork join_none 的用法差异

Fork join:内部 begin end 块并行运行，直到所有线程运行完毕才会进入下一个阶段。

Fork join_any:内部 begin end 块并行运行，任意一个 begin end 块运行结束就可以进入下一个阶段。

Fork join_none:内部 begin end 块并行运行，无需等待可以直接进入下一个阶段。

3. 多线程的同步调度方法

多线程之间同步主要由 mailbox、event、semaphore 三种进行一个通信交互。

mailbox 邮箱:主要用于两个线程之间的数据通信，通过 put 函数和 get 函数还有 peek 函数进行数据的发送和获取。

Event:事件主要用于两个线程之间的一个同步运行，通过事件触发和事件等待进行两个线程间的运行同步。使用@(event)或者 wait(event.trigger)进行等待，->进行触发。

Semaphore:旗语主要是用于对资源访问的一个交互，通过 key 的获取和返回实现一个线程对资源的一个访问。使用 put 和 get 函数获取返回 key。一次可以多个。

4. Task 和 function 的比较

function 至少要有一个输入变量;

function 至少一个返回值；

function 可以包含时序控制；

task 可以自定义仿真时间

1) 对于初学者，傻瓜式用法即全部采用 task 来定义方法，因为它可以内置常用的耗时语句。

2) 对于有经验的使用者，请今后对这两种方法类型加以区别，在非耗时方法定义时使用 function，在内置耗时语句时使用 task。这么做的好处是在遇到了这两种方法定义时，就可以知道 function 只能运用于纯粹的数字或者逻辑运算，而 task 则可能会被运用于需要耗时的信号采样或者驱动场景。

3) 如果要调用 function，则使用 function 和 task 均可对其调用;而如果要调用 task，我们建议使用 task 来调用，这是因为如果被调用的 task 内置有耗时语句，则外部调用它的方法类型必须为 task。function 和 task 的比较，面试经常会用

5. 简述在 TB 中使用 interface 和 clocking blocking 的好处

Interface 是一组接口，用于对信号进行一个封装，捆扎起来。如果像 verilog 中对各个信号进行连接，每一层我们都需要对接口信号进行定义，若信号过多，很容易出现人为错误，而且后期的可重用性不高。因此使用 interface 接口进行连接，不仅可以简化代码，而且提高可重用性，除此之外，interface 内部提供了其他一些功能，用于测试平台与 DUT 之间的同步和避免竞争。

Clocking block:在 interface 内部我们可以定义 clocking 块，可以使得信号保持同步,对于接口的采样 vrbg 和驱动有详细的设置操作，从而避免 TB 与 DUT 的接口竞争，减少我们由于信号竞争导致的错误。采样提前，驱动落后，保证信号不会出现竞争。

6. 对 C++基础的理解（类的封装、继承、多态）

类主要有三个特性:封装、继承、多态。

封装:通过将一些数据和使用这些数据的方法封装在一个集合里，成为一个类。

继承:允许通过现有类去得到一个新的类，且其可以共享现有类的属性和方法。现有类叫做基类，新类叫做派生类或扩展类。

多态:得到扩展类后，有时我们会使用基类句柄去调用扩展类对象，这时候调用的方法如何准确去判断是想要调用的方法呢?通过对类中方法进行 virtual 声明，这样当调用基类句柄指向扩展类时，方法会根据对象去识别，调用扩展类的方法，而不是基类中的。而基类和扩展类中方法有着同样的名字，但能够准确调用，叫做多态。

7. 类的 public、protected 和 local 的区别

1) 如果没有指明访问类型，那么成员的默认类型是 public，子类和外部均可以访问成员。

2) 如果指明了访问类型是 protected，那么只有该类或者子类可以访问成员，而外部无法访问。

3) 如果指明了访问类型是 local，那么只有该类可以访问成员，子类和外部均无法访问。

8. 带约束的随机类的语法和使用（权重约束和条件约束、范围约束）

随机化是 SV 中极其重要的一个知识点，通过设定随机化和相关约束，我们可以自动随机的想要的数据。

权重约束 dist:有两种操作符: :=n :/n 第一种表示每一个取值权重都是 n,第二种表示每一个取值权重为 num/n。

条件约束 if else 和-> (case): if else 就是和正常使用一样;->通过前面条件满足后可以触发后面事件的发生。

范围约束 inside: inside{[min:max]};范围操作符，也可以直接使用大于小于符号进行，不可以连续使用，如 min<wxm<max 这是错误的。

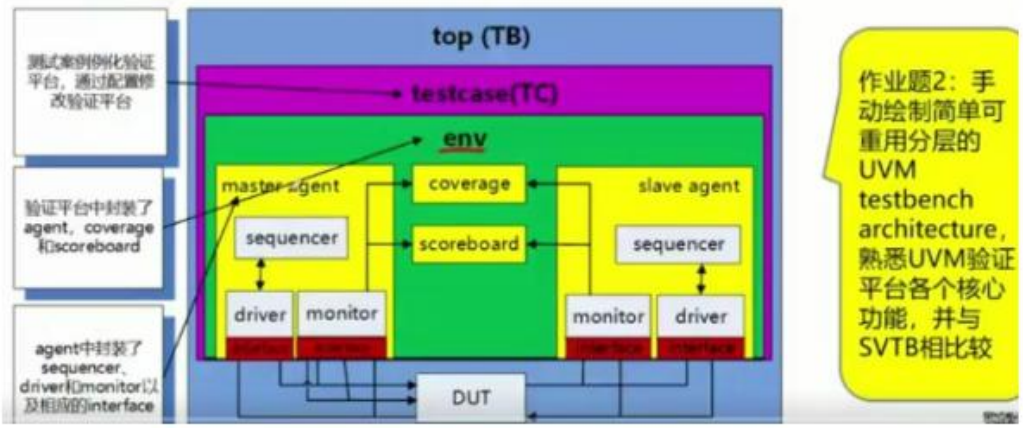
9. 对 UVM 验证方法学的理解

刚开始接触的时候，我认为 UVM 其实就是 SV 的一个封装，将我们在搭建测试平台过程中的一些重复性和重要的工作进行封装,从而使我们能够快速的搭建一个需要的测试平台，并且可重用性还高。因此我当时觉得它就是一个库。

不过，随着学习的不断深入，当我深入理解 UVM 中各种机制和模型的构造和相互之间关系之后,我觉得其实 UVM 方法学对于使用何种语言其实并不重要，重要的是他的思想，比如:在 UVM 中有 sequence 机制，以往如果我们使用 SV 进行 TB 搭建时，我们一般会采用 driver 一个类进行数据的产生，转换，发送，或者使用 generator 和 driver 两个进行，这种方式可重用性很低，而且代码臃肿;但是在 UVM 中我们通过将 sequence、sequencer、driver、sequence_item 拆开，相互独立而又有联系，因此我们只需关注每一个类需要做的工作就可以，可重用性高。我在学习 sequence 时，我经常把 sequence 比作蓄水池，sequence_item 就是水，sequencer 就是一个调度站，driver 就是总工厂，通过这种方式进行处理，我们的总工厂不需要管其他，只需处理运送过来的水资源就可以，而 sequencer 只需要调度水资源，sequence 只需要产生不同的水资源。而这种处理方式和现实世界中的生产模式又是基本吻合的。除此之外，还有好多好多，其实 UVM 方法学中很多思想就是来源于经验，来源于现实生活，而不在乎是何种语言。

10. 请谈一下 UVM 的验证环境结构，各个组件间的关系

画出 UVM 的验证环境结构，如图所示



首先，UVM 测试平台基本是由 object 和 component 组成的，其中 component 搭建了 TB 的一个树形结构，其基本包含了 driver、monitor、sequencer、agent、scoreboard、model、env、test、top;然后 object 一般包含 sequence_item、config 和一些其他需要的类。各个组件相互独立，又通过 TLM 事务级传输进行通信，除此之外，DUT 与 driver 和 monitor 又通过 interface 进行连接，实现驱动和采集，最后在 top 层进行例化调用 test 进行测试。

11. 举例说明 UVM 组件中常用的方法，各种 phase 关系，phase 机制作用

UVM 中有很多非常有趣的机制，例如 factory 机制，field_automation 机制，phase 机制，打印机制，sequence 机制，config_db 机制等，这些机制使得我们搭建的 UVM 能够有很好的可重用性和使得我们平台运行有秩序稳定。

例如 phase 机制，phase 机制主要是使得 UVM 的运行仿真层次化，使得各种例化先后次序正确。UVM 的 phase 机制主要有 9 个，外加 12 个小 phase。主要的 phase 有 build phase、connect phase、run phase、report phase、final phase 等，其中除了 run phase 是 task，其余都是 function，然后 build phase 和 final phase 都是自顶向下运行,其余都是自底向上运行。Run phase 和 12 个小 phase(reset phase、configure phase、main phase、shutdown phase)是并行运行的，有这 12 个小 phase 主要是进一步将 run phase 中的事务划分到不同的 phase 进行，简化代码。注意，run phase 和 12 个小 phase 最好不要同时使用。从运行上来看，9 个 phase 顺序执行，不同组件中的同一个 phase 执行有顺序，build phase 为自顶向下，只有同一个 phase 全部执行完毕才会执行下一个 phase。

所有的 phase 按照以下顺序自上而下自动执行：(九大 phase,其中 run phase 又分为 12 个小 phase)

build_pase

connect_phase

end_of_elaboration_phase

start_of_simulation_phase

run_pase

extract_phase

check_phase

report_phase

final_phase

其中，*run_phase*按照以下顺序自上而下执行:

pre_reset_phase

reset_phase

post_reset_phase

pre_configure_phase

configure_phase

post_configure_phase

pre_main_phase

main_phase

post_main_phase

pre_shutdown_phase

shutdown_phase

post_shutdown_phase

12. phase 中的 domain 概念

Domain 是用来组织不同组件，实现独立运行的概率。默认情况下，UVM 的 9 个 phase 属于 common_domain，12 个小 phase 属于 uvm_domain。例如，如果我们有两个 dirver 类，默认情况下，两个 driver 类中的复位 phase 和 main phase 必须同时执行,但是我们可以设置两个 driver 属于不同的 domain,这样两个 dirver 就是独立运行的了，相当于处于不同的时钟域(只针对 12 个小 phase 有效)。

13. run_phase 和 main_phase 之间的关系;

run_phase 和 main phase（动态运行）都是 task phase，且是并行运行的，后者称为动态运行(run-time)的 phase。如果想执行一些耗费时间的代码，那么要在此 phase 下任意

一个 component 中至少提起一次 objection，这个结论只适用于 12 个 run-time 的 phase。对于 run_phase 则不适用，由于 run_phase 与动态运行的 phase 是并行运行的，如果 12 个动态运行的 phase 有 objection 被提起，那么 run_phase 根本不需要 raise_objection 就可以自动执行。

14. main_phase 要如何跳转到 reset_phase;

在 main_phase 执行过程中，突然遇到 reset 信号被置起，可以用 jump()实现从 mian_phase 到 reset_phase 的跳转：

15. UVM component 和 UVM object 的关系和差异

UVM 中 component 也是由 object 派生出来的，不过相比于 object, component 有很多其没有的属性，例如 phase 机制和树形结构等。在 UVM 中，不仅仅需要 component 这种较为复杂的类，进行 TB 的层次化搭建，也需要 object 这种基础类进行 TB 的事务搭建和一些环境配置等。

16. UVM 组件的通信方式 TLM 的接口分类和用法，peek 和 get 的差异

UVM 中采用事务级传输机制进行组件间的通信，可以大大提高仿真的速度和使得我们简化组件间的数据传输，简化工作，TLM 独立于组件之外，降低组件间的依赖关系。UVM 接口主要由 port、export、imp;驱动这些接口方式有 put、get、peek、transport、analysis 等。其中 peek 是查看端口内部的数据事务但是不删除，get 是获取后立即删除。我们一般会先使用 peek 进行获取数据，但不删除(保证 put 端不会立马又发送一个数据)，处理完毕后再用 get 删除。

Imp 只能作为终点接口，transport 表示双向通信，analysis 可以连接多个 imp(类似于广播)。

17. 单向通信、双向通信、多向通信

单向通信指的是从 initiator 到 target 之间的数据流向是单一方向的

双向通信（bidirectional communication）的两端也分为 initiator 和 target，但是数据流向在端对端之间是双向的

多向通信：initiator 与 target 之间的相同 TLM 端口数目超过一个时的处理解决办法。

18.通信管道：适用于一端到多端的传输（TLM FIFO/analysis port/analysis TLM FIFO/request & response 通信管道）

1) TLM FIFO：可以进行数据缓存，uvm_tlm_fifo 类是一个新组件，它继承于 uvm_component,而且已经预先内置了多个端口以及实现了多个对应方法。

- uvm_tlm_fifo 的功能类似于 mailbox,不同的地方在于 uvm_tlm_fifo 提供了各种端口供用户使用。我们推荐在 initiator 端例化 put_port 或者 get_peek_port，来匹配 uvm_tlm_fifo 的端口类型。
- 当然，如果用户例化了其它类型的端口，uvm_tlm_fifo 还提供 put、get 以及 peek 对应的端口:

2) analysis port：一端对多端

- 除了端对端的传输，在一些情况下还有多个组件会对同一个数据进行运算处理。
- 如果这个数据是从同一个源的 TLM 端口发出到达不同组件，这就要求该种端口可以满足从一端到多端的需求。
- 如果数据源端发生变化需要通知跟它关联的多个组件时，我们可以利用软件的设计模式之一观察者模式(observer pattern）来实现。

广播 observer pattern 的核心在于:

第一，这是从一个 initiator 端到多个 target 端的方式。

第二，analysis port 采取的是"push"模式，即从 initiator 端调用多个 target 端的 write()函数来实现数据传输。

3) Analysis TLM FIFO

- 由于 analysis 端口提出实现了一端到多端的 TLM 数据传输，而一个新的数据缓存组件类 uvm_tlm_analysis_fifo 为用户们提供了可以搭配 uvm_analysis_port 端口 uvm_analysis_imp 端口和 write()函数。
- uvm_tlm_analysis_fifo 类继承于 uvm_tlm_fifo，这表明它本身具有面向单一 TLM 端口的数据缓存特性，而同时该类又有一个 uvm_analysis_imp 端口 analysis_export 并且实现了 write()函数:

uvm_analysis_imp #(T,uvm_tlm_analysis_fifo #(T)) analysis_export;

- 基于 initiator 到多个 target 的连接方式，用户如果想轻松实现一端到多端的数据传输，可以插入多个 uvm_tlm_analysis_fifo，我们这里给出连接方式:

- 将 initiator 的 analysis port 连接到 tlm_analysis_fifo 的 get_export 端口，这样数据可以从 initiator 发起,写入到各个 tlm_analysis_fifo 的缓存中。
- 将多个 target 的 get_port 连接到 tlm_analysis_fifo 的 get_export，注意保持端口类型的匹配，这样从 target 一侧只需要调用 get()方法就可以得到先前存储在 tlm_analysis_fifo 中的数据。

19. Analysis port 是否可以不连或者连多个 impport

都可以。Analysis port 类似于广播，其可以同时对多个 imp 进行事务通信，只需要在每一个对应的 imp 端口申明 write()函数即可。

对比 put.get,peek port，他们都只能进行一对一传输，且也必须申明对应的函数如 put(、get()、peek()、can_put()/do_put()等。

Fifo 是可以不用申明操作函数的，其内部封装了很多的通信端口，如 analysis_export 等，我们只需要将端口与其连接即可实现通信。

20. Sequence 和 item（uvm_sequece，uvm_sequence_item）以及 sequence 的分类

1) item 是基于 uvm_object 类，这表明了它具备 UVM 核心基类所必要的数据操作方法，例如 copy、 clone、compare、record 等。
2) item 对象的生命应该开始于 sequence 的 body（）方法，而后经历了随机化并穿越 sequencer 最终到达 driver，直到被 driver 消化之后，它的生命一般来讲才会结束。
3) item 与 sequence 的关系

   一个 sequence 可以包含一些有序组织起来的 item 实例，考虑到 item 在创建后需要被随机化，sequence 在声明时也需要预留一些可供外部随机化的变量，这些随机变量一部分是用来通过层级传递约束来最终控制 item 对象的随机变量，一部分是用来对 item 对象之间加以组织和时序控制的。
4) Sequence 的分类:

   扁平类（flat sequence）：这一类往往只用来组织更细小的粒度，即 item 实例构成的组织。

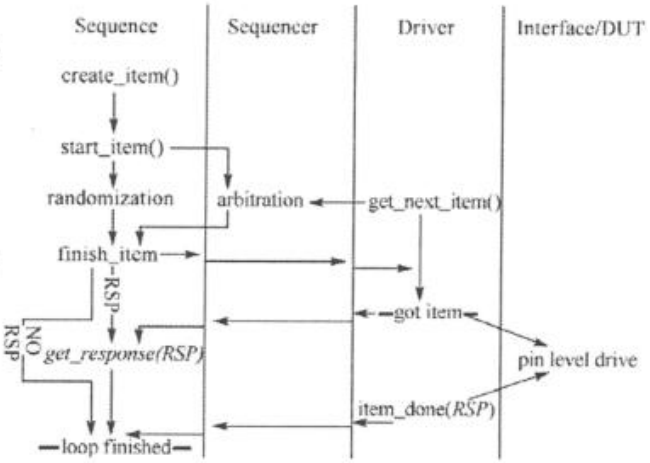
   层次类(hierarchical sequence)：这一类是由更高层的 sequence 用来组织底层的 sequence,进而让这些 sequence 或者按照顺序方式，或者按照并行方式，挂载到同一个 sequencer 上。

   虚拟类（virtual sequence)：这一类则是最终控制整个测试场景的方式，鉴于整个

环境中往往存在不同种类的 sequencer 和其对应的 sequence，我们需要一个虚拟的 sequence 来协调顶层的测试场景。之所以称这个方式为 virtual sequence，是因为该序列本身并不会固定挂载于某一种 sequencer 类型上，而是将其内部不同类型 sequence 最终挂载到不同的目标 sequencer 上面。这也是 virtual sequence 不同于 hierarchical sequence 的最大一点。

21. Sequence、sequencer 和 driver 之间的通信时序

- 无论是sequence还是driver，它们通话的对象都是sequencer。当多个sequence试图要挂载到同一个sequencer上时，涉及sequencer的仲裁功能。
- 重点分析sequencer作为sequence与driver之间握手的桥梁，是如何扮演好这一角色的。
- 我们将抽取去这三个类的主要方法，利用时间箭头演示出完整的TLM通信过程。



- 对于sequence而言，无论是flat sequence还是hierarchical sequence，进一步切分的话，流向sequencer的都是sequence item，所以就每个item的"成长周期"来看，它起始于create_item()，继而通过start_item()尝试从sequencer获取可以通过的权限。
- 对于sequencer的仲裁机制和使用方法我们暂且略过，而driver一侧将一直处于"吃不饱"的状态，如果它没有了item可以使用，将调用get_next_item()来尝试从sequencer一侧获取item。
- 在sequencer将通过权限交给某一个底层的sequence前，目标sequence中的item应该完成随机化，继而在获取sequencer的通过权限后，执行finish_item()。

- 接下来sequence中的item将穿过sequencer到达driver一侧，这个重要节点标志着sequencer第一次充当通信桥梁的角色已经完成。
- driver在得到新的item之后，会提取有效的数据信息，将其驱动到与DUT连接的接口上面。
- 在完成驱动后，driver应当通过item_done()来告知sequence已经完成数据传送，而sequence在获取该消息后，则表示driver与sequence双方完成了这一次item的握手传输。
- 在这次传递中，driver可以选择将RSP作为状态返回值传递给sequence，而sequence也可以选择调用get_response(RSP)等待从driver一侧获取返回的数据对象。

在多个 sequence 同时向 sequencer 发送 item 时，需要有 ID 信息表明该 item 从哪个 sequence 来，ID 信息在 sequence 创建 item 时就赋值了。

22. 请谈一下 virtual sequencer 和 sequencer 的区别以及为什么要用 virtual sequencer

Virtual sequencer 主要用于对不同的 agent 进行协调时，需要有一定顶层的 sequencer 对内部各个 agent 中的 sequencer 进行协调，因此 virtual sequencer 是面向多个 sequencer 的多个 sequence 群，而 sequencer 是面向一个 sequencer 的 sequence 群。Virtual sequencer 桥接着所有底层的 sequencer 的句柄，其本身也不需要传递 item，不需要和 driver 连接。只需要将其内部的底层 sequencer 句柄和 sequencer 实体对象连接。

23. Sequence 和 sequencer 的关系

sequence 机制用于产生激励，它是 UVM 中最重要的机制之一。sequence 机制有两大组成部分：sequence 和 sequencer。在整个验证平台中 sequence 处于一个比较特殊的位置。sequence 不属于验证平台的任何一部分，但是它与 sequencer 之间有着密切的关系。只有在 sequencer 的帮助下，sequence 产生的 transaction 才能最终送给 driver；同样，sequencer 只有在 sequence 出现的情况下才能体现出其价值，如果没有 sequence，sequencer 几乎没有任何作用。除此之外，sequence 与 sequencer 还有显著的区别。从本质上说，sequencer 是一个 uvm_component,而 sequence 是一个 uvm_object。与 my_transaction 一样，sequence 也有其生命周期。它的生命周期比 my_transaction 要更长一点，其内部的 transaction 全部发送完毕后，它的生命周期也就结束了。

24. 将 item 挂载到 sequencer 上的方法：(start_item 和 finish_item) 以及发送 sequence/item 方法（uvm_do/uvm_do_with）

uvm_sequence.;start_item (uvm_sequence_item item, int set_priority = -1,

uvm_sequencer_base sequencer=null);

uvm_sequence:finish_item (uvm_sequence_item item, int set_priority = -1);

发送 sequence/item 方法解析

1) 在使用 start_item/finish_item 这对方法时需要创建 item 并进行随机化处理
2) 对于一个 item 的完整发送，sequence 要在 sequencer 一侧获得通过权限，才可以顺利将 item 发送至 driver

- 创建item。
- 通过start_item()方法等待获得sequencer的授权许可，其后执行parent sequence的方法pre_do()。
- 对item进行随机化处理。
- 通过finish_item()方法在对item进行了随机化处理之后，执行parent sequence的mid_do()，以及调用uvm_sequencer::send_request()和uvm_sequencer::wait_for_item_done()来将item发送至sequencer再完成与driver之间的握手。最后，执行了parent_sequence的post_do()。

25. Sequencer 的仲裁特性（set_arbitration）及锁定机制（lock 和 grab）

1) 仲裁特性

- 在实际使用中，我们可以通过uvm_sequencer::set_arbitration(UVM_SEQ_ARB_TYPE val)函数来设置仲裁模式，这里的仲裁模式UVM_SEQ_ARB_TYPE有下面几种值可以选择：
  - UVM_SEQ_ARB_FIFO：默认模式。来自于sequences的发送请求，按照FIFO先进先出的方式被依次授权，和优先级没有关系。
  - UVM_SEQ_ARB_WEIGHTED：不同sequence的发送请求，将按照它们的优先级权重随机授权。
  - UVM_SEQ_ARB_RANDOM：不同的请求会被随机授权，而无视它们的抵达顺序和优先级。
  - UVM_SEQ_ARB_STRICT_FIFO：不同的请求，会按照它们的优先级以及抵达顺序来依次授权，因此与优先级和抵达时间都有关。
  - UVM_SEQ_ARB_STRICT_RANDOM：不同的请求，会按照它们的最高优先级随机授权，与抵达时间无关。
  - UVM_SEQ_ARB_USER：用户可以自定义仲裁方法user_priority_arbitration()来裁定哪个sequence的请求被优先授权。

2) 锁定机制

- uvm_sequencer提供了两种锁定机制，分别通过lock()和grab()方法实现，这两种方法的区别在于：
  - lock()与unlock()这一对方法可以为sequence提供排外的访问权限，但前提条件是，该sequence首先需要按照sequencer的仲裁机制获得授权。而一旦sequence获得授权，则无需担心权限被收回，只有该sequence主动解锁（unlock）它的sequencer，才可以释放这一锁定的权限。lock()是一种阻塞任务，只有获得了权限，它才会返回。
  - grab()与ungrab()也可以为sequence提供排外的访问权限，而且它只需要在sequencer下一次授权周期时就可以无条件地获得授权。与lock方法相比，grab方法无视同一时刻内发起传送请求的其它sequence，而唯一可以阻止它的只有已经预先获得授权的其它lock或者grab的sequence。
  - 这里需要注意的是，由于"解铃还须系铃人"，如果sequence使用了lock()或者grab()方法，必须在sequence结束前调用unlock()或者ungrab()方法来释放权限，否则sequencer会进入死锁状态而无法继续为其余sequence授权。

26. Virtual sequence 和 virtual sequencer 的区别以及 virtual 含义

- virtual sequence可以承载不同目标sequencer的sequence群落，而组织协调这些sequence的方式则类似于高层次的hierarchical sequence。virtual sequence一般只会挂载到virtual sequencer上面。
- virtual sequencer与普通的sequencer相比有着很大的不同，它们起到了桥接其它sequencer的作用，即virtual sequencer是一个链接所有底层sequencer句柄的地方，它是一个中心化的路由器。
- 同时virtual sequencer本身并不会传送item数据对象，因此virtual sequencer不需要与任何的driver进行TLM连接。所以UVM用户需要在顶层的connect阶段，做好virtual sequencer中各个sequencer句柄与底层sequencer实体对象的一一对接，避免句柄悬空。

Virtual 含义就是其 sequencer 并不需要传递 item，也不会与 driver 连接，其只是一个去协调各个 sequencer 的中央路由器。通过 virtual sequencer 我们可以实现多个 agent 的多个 sequencer 他们的 sequence 的调度和可重用。Virtual sequence 可以组织不同 sequencer 的 sequence 群落。

27. 为什么会有 sequence、sequencer 以及 driver，为什么要分开实现，这样做的好处是什么？

在 UVM 中有 sequence 机制，以往如果我们使用 SV 进行 TB 搭建时，我们一般会采用 driver 一个类进行数据的参数,转换,发送,或者使用 genetor 和 driver 两个进行，这种方式可重用性很低，而且代码臃肿;但是在 UVM 中我们通过将 sequence、sequencer、driver、sequence_item 拆开，相互独立而又有联系，因此我们只需关注每一个类需要做的工作就可以，可重用性高。我在学习 sequence 时，我经常把 sequence 比作蓄水池，sequence_item 就是水，sequencer 就是一个调度站，driver 就是总工厂，通过这种方式进行处理，我们的

总工厂不需要管其他，只需处理运送过来的水资源就可以，而 sequencer 只需要调度水资源，sequence 只需要产生不同的水资源。

28. 如何在 driver 中使用 interface，为什么

Interface 如果不进行 virtual 申明的话是不能直接使用在 dirver 中的,会报错，因为 interface 申明的是一个实际的物理接口。一般在 dirver 中使用 virtual interface 进行申明接口，然后通过 config_db 进行接口参数传递，这样我们可以从上层组件获得虚拟的 interface 接口进行处理。Config_db 传递时只能传递 virtual 接口，即 interface 的句柄，否则传递的是一个实际的物理接口，这在 driver 中是不能实现的，且这样的话不同组件中的接口一一对应一个物理接口，那么操作就没有意义了。

29. 你了解 uvm 的 factory 和 callback 机制嘛

Factory 机制也叫工厂机制，其存在的意义就是为了能够方便的替换 TB 中的实例或者已注册的类型。一般而言，在搭建完 TB 后，我们如果需要对 TB 进行更改配置或者相关的类信息，我们可以通过使用 factory 机制进行覆盖，达到替换的效果，从而大大提高 TB 的可重用性和灵活性。要使用 factory 机制先要进行:

1.将类注册到 factory 表中

2.创建对象，使用对应的语句 （type_id::create）

3.编写相应的类对基类进行覆盖。

Callback 机制其作用是提高 TB 的可重用性,其还可进行特殊激励的产生等，与 factory 类似，两者可以有机结合使用。与 factory 不同之处在于 callback 的类还是原先的类，只是内部的 callback 函数变了，而 factory 这是产生一个新的扩展类进行替换。

1) UVM 组件中内嵌 callback 函数或者任务

2)定义一个常见的 uvm_callbacks class

3) 从 UVM callback 空壳类扩展 uvm_callback 类

4)在验证环境中创建并登记 uvm_callback

30. field_automation 机制和 objection 机制

对 field_automation 最直观的感受是，他可以自动实现 copy、compare、print 等三个函数。当使用 uvm_field 系列相关宏注册之后，可以直接调用以上三个函数，而无需自己定义。这极大的简化了验证平台的搭建，尤其是简化了 driver 和 monitor，提高了效率。

UVM 中通过 objection 机制来控制验证平台的关闭，需要在 drop_objection 之前先 raise_objection。验证在进入到某一 phase 时，UVM 会收集此 phase 提出的所有 objection，

并且实时监测所有 objection 是否已经被撤销了,当发现所有都已经撤销后，那么就会关闭此 phase，开始进入下一个 phase。当所有的 phase 都执行完毕后，就会调用$finish 来将整个验证平台关掉。如果 UVM 发现此 phase 没有提起任何 objection，那么将会直接跳转到 下一个 phase 中。

UVM 的设计哲学就是全部由 sequence 来控制激励生成，因此一般情况下只在 sequence 中控制 objection。另外还需注意的是，raise_objection 语句必须在 main_phase 中第一个消耗仿真时间的语句之前。

31. Config_db 的作用，以及传递其使用时的参数含义

Config_db 机制主要作用就是传递参数使得 TB 的可配置性高，更加灵活。Config _db 机制主要传递的有三种类型:

一种是 interface 虚拟接口，通过传递 virtual interface 使得 dirver 和 monitor 能够与 DUT 连接，并驱动接口和采集接口信号。

第二种是单一变量参数，如 int,string,enum 等，这些主要就是为了配置某些循环次数，id 号是多少等等。

第三种是 object 类，这种主要是当配置参数较多时，我们可以将其封装成一个 object 类，去包含这些属性和相关的处理方法,这样传递起来就比较简单明朗，不易出错。

Config_db 的参数主要由四个参数组成，如下所示，第一个参数为父的根 parent，第二个参数为接下来的路径，对应的组件，第三个是传递时的名字（必须保持一致），第四个是变量名。

```
uvm_config_db #(virtual interface) :: set(uvm_root::get(),"uvm_test_top.c1",'vif",vif);
uvm_config_db #(virtual interface) :: get(this,"","vif",vif);
```

32. UVM 中各个 component 之间是如何组织运行的，串行还是并行，通过什么机制进行调度的

Component 之间通过在 new 函数创建时指定 parent 参数指定子关系，通过这种方法来将 TB 形成一个树形结构。

UVM 中运行是通过 Phase 机制进行层次化仿真的。从组件来看各个组件并行运行，从 phase 上看是串行运行，有层次化的。Phase 机制的 9 个 phase 是串行运行的，不同组件中的同一个 phase 都运行完毕后才能进入下一个 phase 运行，同一个 phase 在不同组件中的运行也是由一定顺序的, build 和 final 是自顶向下。

33. UVM 如何启动一个 sequence

启动 sequence 有很多的方法:常用的方法有使用 default sequence 进行调用，其会将对应的 sequence 与 sequencer 绑定，当 dirver 请求获得 req 时，sequencer 就会调用对应的 sequence 去运行 body 函数，从而产生 req。

除此之外，还可以使用 start 函数进行，其参数主要就是对应的需要绑定的 sequencer 和该类的上层 sequence。如此，就可以实现启动 sequence 的功能。

注意:一般仿真开始结束会在 sequence 中 raise objection 和 drop objection

34. 你所搭建的验证平台为什么要用 RAL（寄存器）

首先，我们要了解寄存器对于设计的重要性，其是模块间交互的窗口，我们可以通过读寄存器值去观察模块的运行状态,通过写寄存器去控制模块的配置和功能改变。然后，为什么我们需要 RAL 呢?由于前面寄存器的重要性，我们可以知道，如果我们不能先保证我们寄存器的读写正确，那么就不用谈后续 DUT 是否正确了，因此，寄存器的验证是排在首要位置的。那么我们应该用什么方法去读写和验证寄存器呢?采用 RAL 寄存器模型去测试验证，是目前最成功的方法吧，寄存器模型独立于 TB 之外，我们可以搭建一个测试寄存器的 agent，去通过前门或者后门访问去控制 DUT 的寄存器,使得 DUT 按照我们的要求去运行。除此之外，UVM 中内建了很多 RAL 的 sequence，用于帮助我们去检测寄存器，除此之外，还有一些其他的类和变量去帮助我们搭建，以提高 RAL 的可重用性和便捷性还有更全的覆盖率。

35. 前门访问和后门访问的区别，后门访问的路径怎么配置：

[1] 前门访问和后门访问的比较

1) 前门访问，顾名思义指的是在寄存器模型上做的读写操作，最终会通过总线 UVC 来实现总线上的物理时序访问，因此是真实的物理操作。
2) 后门访问，指的是利用 UVM DPI (uvm_hdl_read()、uvm_hdl_deposit()) ,将寄存器的操作直接作用到 DUT 内的寄存器变量，而不通过物理总线访问。
3) 前门访问在使用时需要将 path 设置为 UVM_FRONTDOOR
4) 在进行后门访问时，用户首先需要确保寄存器模型在建立时，是否将各个寄存器映射到了 DUT 一侧的 HDL 路径：使用 add_hdl_path

•总结了前门访问和后门访问的主要差别如下：

前门访问	后门访问
通过总线协议访问需要耗时，且在总线访问结束时才能结束前门访问	通过 UVM DPI 关联硬件寄存器信号路径，直接读取或修改硬件，不需要访问时间，零时刻响应
一般读写只能按字（word）读写，无法直接读写寄存器域	可以对寄存器或寄存器域直接做读写
依靠监测总线来对寄存器模型内容做预测	依靠 auto prediction 方式自动对寄存器内容做预测
正确反映了时序关系	不受硬件时序控制，对硬件做的后门访问可能发生时序冲突
通过总线协议，可以有效捕捉总线错误，继而验证总线访问路径	不受总线时序功能影响

5) 从上面的差别可以看出，后门访问较前门访问更便捷更快一些，但如果单纯依赖后门访问也不能称之为“正道”。

6) 实际上，利用寄存器模型的前门访问和后门访问混合方式，对寄存器验证的完备性更有帮助。

[2] 后门访问路径设置

后门访问

•示例中通过uvm_reg_block::add_hdl_path()，将寄存器模型关联到了DUT一端，而通过uvm_reg::add_hdl_path_slice完成了将寄存器模型各个寄存器成员与HDL一侧的地址映射。

•另外，寄存器模型build()函数最后以lock_model()结尾，该函数的功能是结束地址映射关系，并且保证模型不会被其它用户修改。

•在寄存器模型完成了HDL路径映射后，我们才可以利用uvm_reg或者uvm_reg_sequence自带的方法进行后门访问。后门访问也有几类方法提供：

- uvm_reg::read()/write()，在调用该方法时需要注明UVM_BACKDOOR的访问方式。
- uvm_reg_sequence::read_reg()/write_reg()，在使用时也需要注明UVM_BACKDOOR的访问方式。
- 另外，uvm_reg::peek()/poke()两个方法，也分别对应了读取寄存器（peek）和修改寄存器（poke）两种操作，而用户无需指定访问方式尾UVM_BACKDOOR，因为这两个方法本来就只针对于后门访问。

36. 如果寄存器的地址不匹配的错误怎么测试出来

在通过前门配置寄存器 A 之后，再通过后门访问来判断 HDL 地址映射的寄存器 A 变量值是否改变，最后通过前门访问来读取寄存器 A 的值。

- 有的时候，即便通过先写再读的方式来测试一个寄存器，也可能存在地址不匹配的情况。譬如寄存器A地址本应该0x10，寄存器B地址本应该为0x20；而在硬件实现中，寄存器A对应的地址为0x20，寄存器B对应的地址为0x10。像这种错误，即使通过先写再读的方式也无法有效测试出来，那么不妨在通过前门配置寄存器A之后，再通过后门访问来判断HDL地址映射的寄存器A变量值是否改变，最后通过前门访问来读取寄存器A的值。上述的方式是在前门测试的基础之上又加入了中途的后门访问和数值比较，可以发现地址映射到错误寄存器的问题。

37. 寄存器模型的常规方法（期望值、镜像值、真实值）

mirror、desired、actual value（）

1) 我们在应用寄存器模型的时候，除了利用它的寄存器信息，也会利用它来跟踪寄存器的值。寄存器有很多域，每一个域都有两个值。
2) 寄存器模型中的每一个寄存器，都应该有两个值，一个是镜像值(mirrored value)，一个是期望值(desired value) 。
3) 期望值是先利用寄存器模型修改软件对象值，而后利用该值更新硬件值;镜像值是表示当前硬件的已知状态值。
4) 镜像值往往由模型预测给出，即在前门访问时通过观察总线或者在后门访问时通过自动预测等方式来给出镜像值
5) 镜像值有可能与硬件实际值不一致

38. Prediction 的分类（自动预测和显式预测）

UVM 提供了两种用来跟踪寄存器值的方式，我们将其分为自动预测（auto prediction）和显式预测(explicit)。

如果用户想使用自动预测的方式，还需要调用函数 uvm_reg_map::set_auto predict()。

两种预测方式的显著差别在于，显式预测对寄存器数值预测更为准确，我们可以通过下面对两种模式的分析得出具体原因。

自动预测

1) 如果用户没有在环境中集成独立的 predictor，而是利用寄存器的操作来自动记录每一次寄存器的读写数值，并在后台自动调用 predict()方法的话，这种方式被称之为自动预测。
2) 这种方式简单有效，然而需要注意,如果出现了其它一些 sequence 直接在总线层面上对寄存器进行操作（跳过寄存器级别的 write/read 操作,或者通过其它总线来访问寄存器等这些额外的情况，都无法自动得到寄存器的镜像值和预期值。

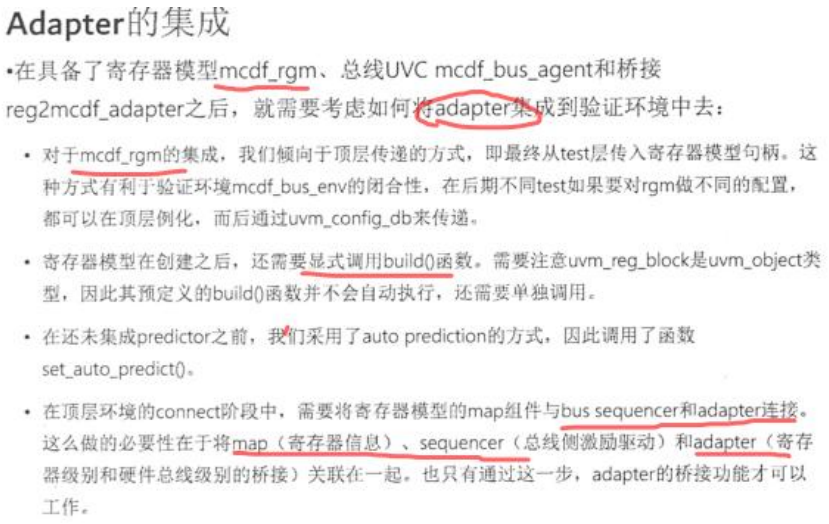
显式预测

1) 更为可靠的一种方式是在物理总线上通过监视器来捕捉总线事务，并将捕捉到的事务传递给外部例化的 predictor，该 predictor 由 UVM 参数化类 uvm_reg_predictor 例化并集成在顶层环境中。
2) 在集成的过程中需要将 adapter 与 map 的句柄也一并传递给 predictor,同时将 monitor 采集的事务通过 analysis port 接入到 predictor 一侧。

3) 这种集成关系可以使得，monitor 一旦捕捉到有效事务，会发送给 predictor，再由其利用 adapter 的桥接方法，实现事务信息转换，并将转化后的寄存器模型有关信息更新到 map 中。

4) 默认情况下，系统将采用显式预测的方式，这就要求集成到环境中的总线 UVC monitor 需要具备捕捉事务的功能和对应的 analysis port，以便于同 predictor 连接。

39. 寄存器怎么配置，adapter 怎么集成


Adapter的集成

•在具备了寄存器模型mcdf_rgm、总线UVC mcdf_bus_agent和桥接reg2mcdf_adapter之后，就需要考虑如何将adapter集成到验证环境中去：

- 对于mcdf_rgm的集成，我们倾向于顶层传递的方式，即最终从test层传入寄存器模型句柄。这种方式有利于验证环境mcdf_bus_env的闭合性，在后期不同test如果要对rgm做不同的配置，都可以在顶层例化，而后通过uvm_config_db来传递。
- 寄存器模型在创建之后，还需要显式调用build()函数。需要注意uvm_reg_block是uvm_object类型，因此其预定义的build()函数并不会自动执行，还需要单独调用。
- 在还未集成predictor之前，我们采用了auto prediction的方式，因此调用了函数set_auto_predict()。
- 在顶层环境的connect阶段中，需要将寄存器模型的map组件与bus sequencer和adapter连接。这么做的必要性在于将map（寄存器信息）、sequencer（总线侧激励驱动）和adapter（寄存器级别和硬件总线级别的桥接）关联在一起。也只有通过这一步，adapter的桥接功能才可以工作。

40. AMBA 总线中 AHB/APB/AXI 协议的区别

AHB(Advanced High-performance Bus)高级高性能总线。

APB(Advanced Peripheral Bus)高级外围总线

AXI (Advanced eXtensible Interface)高级可拓展接口

AHB 主要是针对高效率、高频宽及快速系统模块所设计的总线，它可以连接如微处理器、芯片上或芯片外的内存模块和 DMA 等高效率模块。

APB 主要用在低速且低功率的外围，可针对外围设备作功率消耗及复杂接口的最佳化。APB 在 AHB 和低带宽的外围设备之间提供了通信的桥梁，所以 APB 是 AHB 的二级拓展总线。

AXI 高速度、高带宽，管道化互联，单向通道，只需要首地址，读写并行，支持乱序，支持非对齐操作，有效支持初始延迟较高的外设，连线非常多。

41. AHB 协议

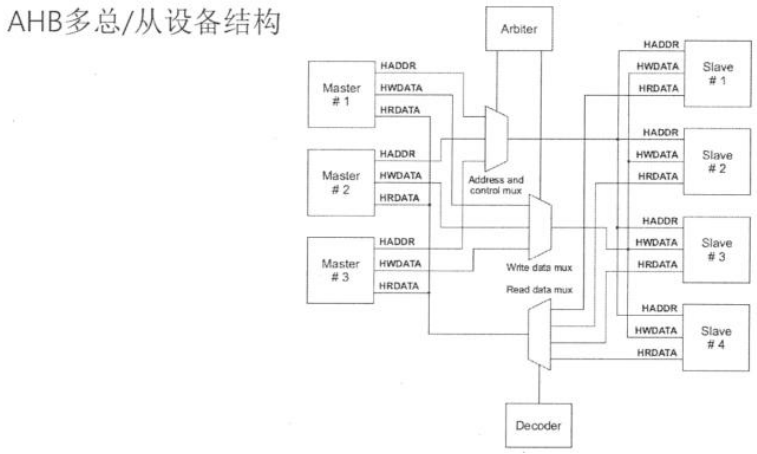
AHB 的组成

Master:能够发起读写操作，提供地址和控制信号，同一时间只有 1 个 Master 会被激活。

Slave:在给定的地址范围内对读写操作作响应，并对 Master 返回成功、失败或者等待状态。

Arbiter:负责保证总线上一次只有 1 个 Master 在工作。仲裁协议是规定的，但是仲裁算法可以根据应用决定。

Decoder:负责对地址进行解码，并提供片选信号到各 Slave。
每个 AHB 都需要 1 个仲裁器和 1 个中央解码器。



AHB 基本信号
HADDR:32 位系统地址总线。
HTRANS:M 指示传输状态，NONSEQ、SEQ、IDLE、BUSY。
HWRITE:传输方向 1-写，0-读。
HSIZE:传输单位。
HBURST:传输的 burst 类型，SINGLE、INCR、WRAP4、INCR4 等。
HWDATA:写数据总线，从 M 写到 S。
HREADY:S 应答 M 是否读写操作传输完成，1-传输完成，0-需延长传输周期。
HRESP:S 应答当前传输状态，OKAY、ERROR、RETRY、SPLIT。
HRDATA:读数据总线，从 S 读到 M。

42. APB 协议及读写操作

- 系统初始化为IDLE状态，此时没有传输操作，也没有选中任何从模块。
- 当有传输要进行时，PSELx=1,，PENABLE=0，系统进入SETUP状态，并只会在SETUP状态停留一个周期。当PCLK的下一个上升沿到来时，系统进入ENABLE状态。
- 系统进入ENABLE状态时，维持之前在SETUP状态的PADDR、PSEL、PWRITE不变，并将PENABLE置为1。传输也只会在ENABLE状态维持一个周期，在经过SETUP与ENABLE状态之后就已完成。之后如果没有传输要进行，就进入IDLE状态等待；如果有连续的传输，则进入SETUP状态。

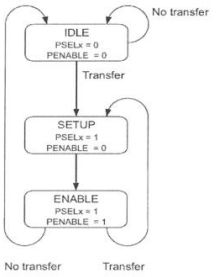


Figure 5-2 State diagram

APB接口

写操作

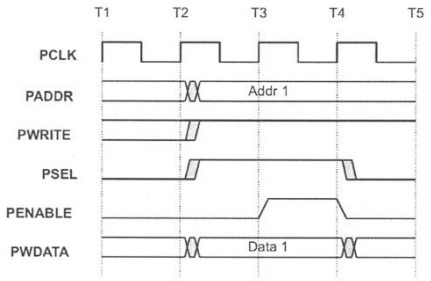


Figure 5-3 Write transfer

APB接口

- 写操作发生时，伴随着地址线、写数据线、写信号线以及选择线一同变化。
- 写操作的第一个周期称之为SETUP周期。
- 下一个周期，PENABLE信号线置起，这表示ENABLE周期。
- 在ENABLE周期，地址线、数据线和控制线都应该保持有效。
- 在ENABLE周期结束后，本次写操作结束。
- PENABLE在写操作周期结束后，会同PSEL一同拉低，除非又需要立即跟随下一次传输。
- 为了省电，地址信号和写信号在一次传输过后不会改变，直到下一次传输发生。

读操作

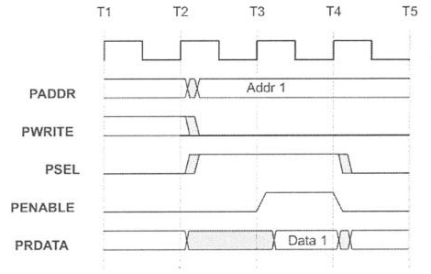


Figure 5-4 Read transfer

APB接口

- 地址线、写信号线、选择线将同写操作时一样保持不变。
- 从端需要在ENABLE周期内，返回PRDATA。
- PRDATA将在ENABLE周期的下一个周期被采样。

43. 你写过 assertion 嘛，assertion 分几种？简述一下 assertion 的用法

Assertion 可以分为立即断言和并发断言。

立即断言的话就是和时序无关，比如我们在对激励随机化时，我们会使用立即断言，如果随机化出错我们就会触发断言报错。

并发断言的话主要是用来检测时序关系的，由于在很多模块或者总线中，单纯使用覆盖率或者事务 check 并不能完全检测多个时序信号之间的关系，但是并发断言却可以使用简洁的语言去监测,除此之外,还可以进行覆盖率检测。

并发断言的用法的话，主要是有三个层次，第一是序列 sequence 编写，将多个信号的关系用断言中特定的操作符进行表示;第二是属性 property 的编写，它可以将多个 sequence 和多个 property 进行嵌套,外加上触发事件;第三就是 assert 的编写，调用 property 就可以。编写完断言后我们可以将它用在很多地方，比如 DUT 内部，或者在 top 层嵌入 DUT 中，还可以在 interface 处进行编写，基本能够检测到信号的地方都可以进行断言检测。

44. 断言 and、intersect、or 和 through 的区别

And 指的是两个序列具有相同的起始点，终点可以不同。

Intersect 指的是两个序列具有相同的起始点和终点。

Or 指的是两个序列只要满足一个就可以

Throughout 指的是满足前面要求才能执行后面的序列

45. a[*3]、a[->3]和 a[=3]区别

a[*3]指的是:重复 3 次 a，且其与前后其他序列不能有间隔，a 中间也不可有间隔。

a[->3]指的是:重复 3 次，其 a 中间可以有间隔，但是其后面的序列与 a 之间不可以有间隔。

A[=3]指的是:只要重复 3 次，中间可随意间隔。

46. 项目中会考虑哪些 coverage

主要会考虑三个方面吧，代码覆盖率，功能覆盖率，断言覆盖率。比如说代码覆盖率，主要由行覆盖率、条件覆盖率、fsm 覆盖率、跳转覆盖率、分支覆盖率，他们是否都是运行到的，比如 fsm，是否各个状态都运行到了，然后不同状态之间的跳转是否也都运行到了。功能覆盖率的话主要是自己编写 covergroup 和 coverpoint 去覆盖我们想要覆盖的数据和地址或者其他控制信号。断言覆盖率主要检测我们的时序关系是否都运行到了,比如总线的地址数据读写时序关系是否都有实现。

47. Coverage 一般不会直接达到 100%，当你发现 condition 未 cover 到的时候，你该怎么做?

Condition 又称为条件覆盖率，当条件覆盖率未被覆盖时，我们需要通过查看覆盖率报告去定位哪些条件没有被覆盖到,是因为没有满足该条件的前提条件还是因为根本就遗漏了这些情况，根据这个我们去编写相应的 case，进而将其覆盖到。

48. Function coverage 和 code coverage 的区别，以及他们分别对项目的含义

功能覆盖率主要是针对 spec 文档中功能点的覆盖检测，code 覆盖率主要是针对 RTL 设计代码的运行完备度的体现，其包括行覆盖率、条件覆盖率、FSM 覆盖率、跳转覆盖率、分支覆盖率（只要仿真就可以，看看 DUT 的哪些代码没有动，如果有一部分代码一直没动，看一下是不是 case 没写到）。功能覆盖率和代码覆盖率两者缺一不可，功能覆盖率表示着代设计是否具备这些功能,代码覆盖率表示我们的测试是否完备，代码是否冗余。当功能覆盖率高而代码覆盖率低时，表示 covergroup 是不是写少了，case 写少了;或者代码冗余。当功能覆盖率很低而代码覆盖率高时，表示代码设计是不是全面，功能点遗漏;covergroup 写的是不是冗余了。只有当两者覆盖率都高的时候才表明我们验证的大部分是可靠的。代码覆盖

率很难达到 100%，一般情况下达到 90%多已经非常不错了，如果有一部分代码没有被触动到，需要有经验的验证工程师去分析，如果确实没啥问题，就可以签字通过了

49. 你在做验证时的流程是怎么样的，你是怎么做的。

对于流程的话，首先第一步我会先去查看 spec 文档，将模块的功能和接口总线时序搞明白，尤其是工作的时序，这对于后续写 TB 非常重要;第二步我会根据功能点去划分我的 TB 应该怎么搭建，我的 case 大致会有哪些，这些功能点我应该如何去覆盖，时序应该如何去检查，总结列出这样的一个清单;第三步开始去搭建我们的 TB，包括各种组件，和一些基础的 sequence 还有 test，暂时先就写一两个基础的 sequence,然后还有一些环境配置参数的确定等,最后能够将 TB 正常运行, 保证无误;第四步就是根据清单去编写 sequence 和 case，然后去仿真，保证仿真正确性，收集覆盖率;第五步就是分析收集的覆盖率，然后查看覆盖率报告去分析还有哪些没有被覆盖，去写一些定向 case，和更换不同的 seed 去仿真;第六步就是回归测试 regression，通过不同的 seed 去跑，收集覆盖率和检测是否有其它 bug;第七步就是总结

50. 你在进行验证的过程中，碰到过什么难点，重点是什么呢?

刚开始的难点还是 TB 的搭建，想要搭建出一个可重用性很高的 TB，配置灵活的 TB 还是有一定困难，对于哪些参数应该放在配置类，哪些参数应该放在事务类的抉择，哪些单独配置。除此之外, 还有就是时序的理解, 这对于 driver 和 monitor 还有 sequence 和 assertion 的编写至关重要，只有正确理解时序才能编写出正确的 TB。最后就是实现覆盖率的尽可能高，这也是比较困难的，刚开始的 case 好写，也比较快就可以达到较高的覆盖率，但是那些边边角角的 case 需要自己去琢磨，去分析还需要写什么 case。这些难点就是重点，还要能够自动化监测判断是否正确。

51. 你发现过哪些验证过程中的 bug，如何发现的?

发现过的 bug 还是很多的，我这里就把不讲什么编译错误的 bug 了，最难的是编译没有错但是最后结果错的 bug。比如:

1) 我有一次在编写完 TB 后，TB 里面有一个 config 类，用于配置环境的相关变量的，然后我后来又写了一个他的派生类，但是我忘记了在顶层传递到 env 去，因此 env 中 config_get 就没有收到这个参数, 而我又在内部写了如果没有收到则使用原先的基类参数，这导致我当时调了半天，无论如何更改配置都无效，最后才发现这个问题。后面反思主要问题其实还是在于我在使用 config_db 的方法不对，我因为没有传递到就是用默认，但是却没用报 warning 或者 fatal，导致我认为没有错误

2) 还有就是时序的分析不对，导致仿真一直在某个阶段等待信号或者事件触发，没法运行。不过这个通过打印的消息可以比较快的检查出来。

3) 一定要设置好 set_drain_time，我刚开始有时候会忘记设置，或者设置时间不对，导致检查结果出错，这个检查办法就是去查看波形会发现最后一个激励是没有发出来的，这个也是可以检查出来的。

4) run phase 和 main phase 混用，导致平台运行出错，有一次我将 reset_phase 和 run_phase 混用，想要能够进行 phase 之间跳跃，出现 bug，并没有实现跳跃，后来查资料说 run_phase 和 12 个小 phase 最好不要混用。

52. 你的验证环境是什么？目录结构是什么样的

我是使用 UVM 验证方法学搭建的 TB，然后在 VCS 平台进行仿真的。目录结构的话:主要由 RTL 文件、doc 文件、tb 文件、sim 文件、script 文件这几部分。

53. UVM 有什么优点：

UVM 的优点：UVM 有各个机制、促进验证平台的标准化，UVM 中 test sequence 和验证平台是隔离独立的，可以更好的控制激励而不需要重新设计 agent. 改变测试 sequence 可以简单高效提高代码覆盖率。**UVM 支持工业标准，这会促进验证平台标准化。此外，UVM 通过 OOP（面向对象编程）的特点（例如继承）以及使用覆盖组件提高了重复使用率。因此 UVM 环境方便移植，架构清晰，组件连接方便，有利于进行大规模的验证。**

缺点：代码冗余，工作量大，运行速度有缺失

54. 工厂机制的好处是什么？（具有重载功能）

factory 机制的优势在于其具有重载功能。重载并不是 factory 机制的发明，只是 factory 机制的重载与这些重载都不一样。要想使用 factory 机制的重载功能，必须满足以下要求：

1) 无论是重载的类（parrot）还是被重载的类（bird），都要在定义时注册到 factory 机制中。

2) 被重载的类（bird）在实例化时，要使用 factory 机制的方式进行实例化，而不能使用传统的 new 的方式。

3) 最重要的是，重载的类（parrot）要与被重载的类（bird）有派生关系。重载的类必须派生自被重载的类，被重载的类必须是重载类的父类。

4) component 与 object 之间互相不能重载。虽然 uvm_component 是派生自 uvm_object，但是这两者根本不能重载。因为，从两者的 new 参数的函数就可以看出来，二者互相重载时，多出来的一个 parent 参数会使 factory 机制无所适从。

55. 通过工厂进行覆盖有什么要求？

第一，无论是重载的类（parrot）还是被重载的类（bird），都要在定义时注册到 factory 机制中。

第二，被重载的类（bird）在实例化时，要使用 factory 机制式的实例化方式，而不能使用传统的 new 方式。

第三，最重要的是，重载的类（parrot）要与被重载的类（bird）有派生关系。重载的类必须派生自被重载的类，被重载的类必须是重载类的父类。

56. vritual function/task 的作用是什么？这属于 oop 的什么特性？

多态

允许在派生类中重新定义与基类同名的函数，并且可以通过基类句柄调用基类和派生类中的同名函数。

57. 域的自动化有什么好处？

可以自动实现 copy、compare、print 等三个函数。当使用 uvm_field 系列相关宏注册之后，可以直接调用以上三个函数，而无需自己定义。这极大的简化了验证平台的搭建，尤其是简化了 driver 和 monitor，提高了效率。

58. 如何启动 test？

总结：1）在导入uvm_pkg文件时，会自动创建UVM_root所例化的对象UVM_top，UVM顶层的类会提供run_test()方法充当UVM世界的核心角色，通过UVM_top调用run_test()方法。 2）在环境中输入run_test来启动UVM验证平台，run_test语句会创建一个my_case0的实例，得到正确的test_name 2）依次执行uvm_test容器中的各个component组件中的phase机制，按照顺序， 1.build-phase（子顶向下构建UVM 树） 2.connet_phase(子低向上连接各个组件) 3.end_of_elaboration_phase 4.start_of_simulation_phase 5.run_phase() objection机制仿真挂起，通过start启动sequence（每个sequence都有一个body任务。当一个sequence启动后，会自动执行sequence的body任务）,等到sequence发送完毕则关闭objection，结束run_phase()（UVM_objection提供component和sequence共享的计数器，当所有参与到objection机制中的组件都落下objection时，计数器counter才会清零，才满足run_phase()退出的条件（UVM入门P45）） 5.执行后面的phase

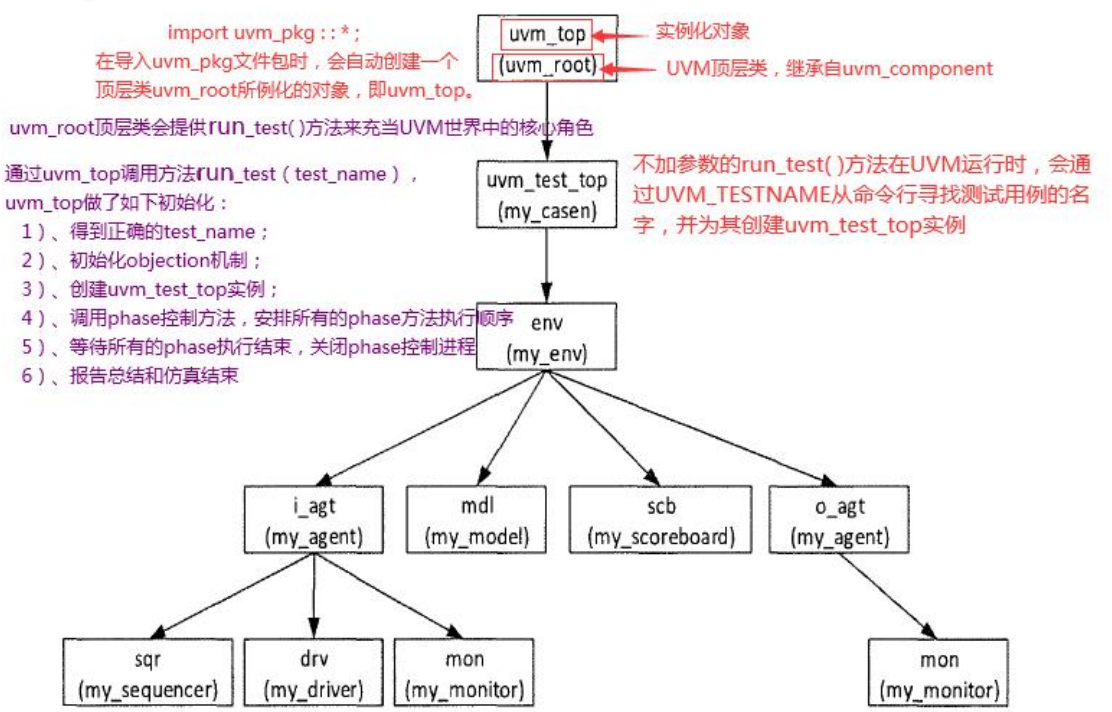


图 2　完整的 UVM 树

59. Config_db 有什么参数?

有五个参数

```
uvm_config_db#(T)::set(uvm_component cntxt, string inst_name, string field_name,     T value);

uvm_config_db#(T)::get(uvm_component cntxt, string inst_name, string field_name, ref T value);
```

例子：

```
uvm_config_db#(virtual my_if)::set(null, "uvm_test_top", "vif", input_if)

uvm_config_db#(virtual my_if)::get(this,                 "", "vif",       vif)
```

在以上两个函数set和get是使用它时要调用的函数，set表示把要处理的资源放进全局可见的数据库，get表示从全局可见的数据库输出需要的资源，使用set和get函数时有五个参数需要制定，**第一个是uvm_config_db类的参数#（T），T表示要set或get的资源的类型，它可以是虚拟接口，sequencer等等，第二个cntxt和第三个参数inst_name一起定义了uvm_config_db中set或get函数的作用范围。第四个参数决定了是对作用范围中的哪个对象或变量进行操作，第五个参数value会存储当前操作对象的句柄或者操作变量的值。**需要注意的是，当制定第二个参数是一个uvm组件时，uvm会用它的全局名字取替换它，而全局名字会通过uvm的get_full_name来获取。

60. 如果环境中有两个 config_db set，哪个有效?

UVM 更高的层次更接近用户，为了让用户少和底层组件打交道，所以层次越高优先级越高，高层次的 set 会覆盖底层次的 set,如果是层次相同再看时间先后顺序，谁发生的晚谁有效，时间靠后的会覆盖之前的。

61. rand bit data[100]，如何约束它随意一位是 1，剩下的都是 0？

```
另外的思路 rand int x;
constraint t  {x>=0 && x <=99;}
after randomize..execute below contest...
foreach (data[i])
if(i==CLASSA.x) data[i]=1;
else data[i]=0;
```

62. VIP 怎么写？

阶段 1（定义)。

[1] 功能特性提取

[2] 特性覆盖率创建及映射

[3] VIP 的架构

阶段 2(VIP 基本搭建)

[1] driver，sequencer，monitor (少量特性实现)。

[2] 实现基本的端到端的 sequence

阶段 3(完成 monitor 与 scoreboard)

[1] 完成 monitor -100%实现（checkers，assertions）

[2] 完成 scoreboard -100%实现（数据完整性检查)

[3] 在 monitor 中，完成监测到的 transaction 与 function coverage 实现映射。

[4] 为映射更多的基本功能覆盖率，创建其它 sequences。

阶段 4(扩充 test 和 sequence 阶段)

[1] 实现更多 sequences，从而获得 80%的功能覆盖率

阶段 5(完成标准)

[1] Sequence 最终可以实现 100%的功能覆盖率。

[2] 回归测试结果和最终的总结报告。

63. 验证流程，验证环境怎么搭

验证流程:

[1] 看 spec 文档和协议，将 DUT 的功能和接口总线时序搞明白

[2] 制定验证计划和测试点分解

[3] 写 VIP 或者是用别人给的 VIP，搭建验证环境和 TB，包括各种组件，各个模块的 pkg，基础的 sequence 还有 test，暂时先就写一两个基础的 sequence,然后还有一些环境配

置参数的确定等,最后能够将 TB 正常运行，保证无误;

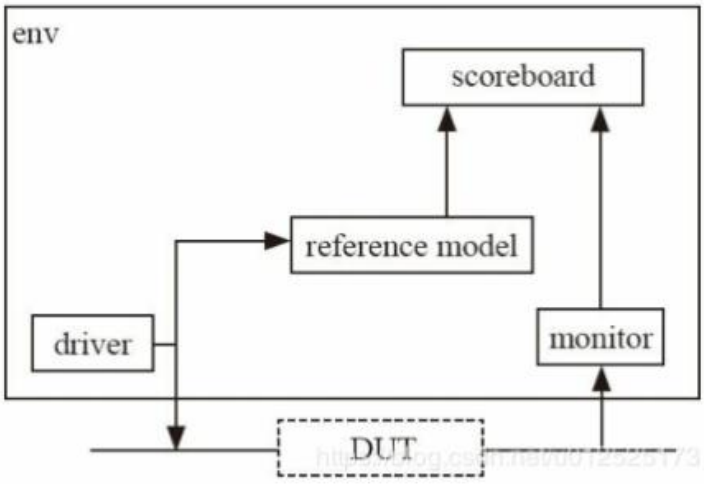
[4] 根据测试点编写 sequence 和 case，然后去仿真，保证仿真正确性，收集覆盖率;

[5] 分析收集的覆盖率，然后查看覆盖率报告去分析还有哪些没有被覆盖，去写一些定向 case，和更换不同的 seed 去仿真;

[6] 回归测试 regression，通过不同的 seed 去跑，收集覆盖率和检测是否有其它 bug;

[7] 总结

验证环境的搭建:



driver 给 DUT 发送激励，montior 监测 DUT 输出的数据，参考模型（reference model）能实现与 DUT 相同的功能，scoreboard 把 monitor 接受到的数据和 reference model 的输出数据进行比对，如果比对成功就表示 DUT 能完成设计的功能，

driver,monitor 和 reference model 合起来叫做环境

64. Uvm_component_utils 有什么作用

当然，factory 机制的实现被集成在了一个宏中：uvm_component_utils。这个宏最主要的任务是，将字符串登记在 UVM 内部的一张表中，这张表是 factory 功能实现的基础。只要在定义一个新的类时使用这个宏，就相当于把这个类注册到了这张表中。这样，factory 机制可以实现：根据一个字符串自动创建一个类的实例，并且调用其中的函数（function）和任务（task），这个类的 main_phase 就会被自动调用。

65. Run_phase 和 main_phase 的区别

run_phase 和 main phase 都是 task phase，且是并行运行的，后者称为动态运行(run-time)的 phase。如果想执行一些耗费时间的代码，那么要在此 phase 下任意一个 component 中至少提起一次 objection，这个结论只适用于 12 个 run-time 的 phase。对于

run_phase 则不适用，由于 run_phase 与动态运行的 phase 是并行运行的，如果 12 个动态运行的 phase 有 objection 被提起，那么 run_phase 根本不需要 raise_objection 就可以自动执行。

uvm 的 run_phase 会与 12 个 task phase 并行执行，12 个 task phase 包括 pre_reset_phase, reset_phase,...main_phase 等等，一般的大部分人并不太用除了 main_phase 之外的别的 task phase。

因此一般使用可以认为 main_phase 跟 run_phase 几乎相同的作用，但是 run_phase 跟 12 个 task phase 所属的 domain 并不一样，run_phase 属于 common_domain，其余的 12 个 task phase 属于 uvm_domain.

66. 寄存器模型的方法（期望值、镜像值、真实值）

1、mirror、desired、actual value（）

我们在应用寄存器模型的时候，除了利用它的寄存器信息，也会利用它来跟踪寄存器的值。寄存器有很多域，每一个域都有两个值。

寄存器模型中的每一个寄存器，都应该有两个值，一个是镜像值(mirrored value)，一个是期望值(desired value) 。

期望值是先利用寄存器模型修改软件对象值，而后利用该值更新硬件值;镜像值是表示当前硬件的已知状态值。

镜像值往往由模型预测给出，即在前门访问时通过观察总线或者在后门访问时通过自动预测等方式来给出镜像值

镜像值有可能与硬件实际值不一致

67. TLM 怎么用

tlm 通信的步骤:

1.分辨出 initiator 和 target，producer 和 consumer。

2.在 target 中实现 tlm 通信方法。

3.在俩个对象中创建 tlm 端口。

4.在更高层次中将俩个对象进行连接。

端口类型有三种:

1.port，一般是 initiator 的发起端。

2.export，作为 initiator 和 target 的中间端口。

3.imp，只能作为 target 接受 request 的末端。

4.多个 port 可以连接同一个 export 或 imp，但是单个 port 或 export 不能连接多个 imp。

端口的连接：通过 connect 函数进行连接，例如 A(initiator)Y 与 B 进行连接，可以使用 A.port.connect(B.export)

uvm_*_imp#(T,IMP);IMP 定义中第一个参数 T 是这个 IMP 传输的数据类型，第二个参数 IMP 是实现这个接口所在的 component。

68. 覆盖率的分类

代码覆盖率，功能覆盖率、断言覆盖率

69. CDC 跨时钟域

单 bit（慢时钟域到快时钟域）：用快时钟打两拍，直接采一拍大概率也是没问题的，两拍的主要目的是消除亚稳态；

单 bit（快时钟域到慢时钟域）：握手、异步 FIFO、异步双口 RAM；快时钟域的信号脉宽较窄，慢时钟域不一定能采到，可以通过握手机制让窄脉冲展宽，慢时钟域采集到信号后再“告诉”快时钟域已经采集到信号，确保能采集到；

多 bit：异步 FIFO、异步双口 RAM、握手、格雷码；

多 bit 中，强烈推荐异步 FIFO，我在实际工程中使用多次，简单方便。

70. Sequence 是怎么启动的？

71. Seq,seqr,driver 通信的细节

72. Case 是怎么写的，case 代码结构介绍一下

73. 覆盖率定义了多少个 bins

74. 寄存器配置的原理，你配置的数据怎么进入 DUT 的