1. SV 语法考察

1) 定宽数组、动态数组、关联数组、队列各自特点和使用 0516

队列：队列结合了链表和数组的有点，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]];索引值必须是惟一的。

2) 结构体的使用 0516

结构体 struct，其相比于其他数据结构，主要的特点就是可以由不同类型的数据类型变量组成，然后可以对这个结构体进行操作。Struct {bit [2:0] a,int c}name;

联合体 union 缺是对同一个内存变量可以有不同类型，比如一个寄存器有时候为整数，有时为小数。

3) 多线程 fork join、fork join_any、fork join_none 的用法差异 0514

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。

4) 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。

5) @signal 触发和 wait(signal)的区别

@(signal)是对 signal 的边缘进行触发，因此如果此时此刻可，进程还未运行到@signal 就已经触发，那么会被遗漏。

Wait(signal)是等待 signal 的高电平触发的，因此，出现上述的情况，也不会遗漏。

6) 简述在 TB 中使用 interface 和 clocking block 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，iinterface 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。

7) 对 C++基础的理解（类的封装、继承、多态） 0516

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

申明，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。

8) SV 中什么需求下必须使用 virtual methods

当我们要通过对象进行识别使用何种方法时，我们需要将方法定义成 virtual，这样才可以准确的根据句柄指向的对象进行调用，而不是直接调用基类方法。一般情况下，我们会除了 new 函数以外，其他方法都设置为 virtual，以防出现问题。

9) $cast()强制向下转换和直接转换

类被申明创建对象后，就会在内存空间中有一块物理意义上的存储空间，new()函数后会返回该对象的句柄。句柄可以指向基类或者扩展类的对象，当将基类对象句柄赋给扩展类对象句柄时（将扩展类句柄指向基类对象时会出错），必须使用$cast()进行强制转换；当将扩展类句柄赋值给基类句柄时（将基类句柄指向扩展类对象时），可以直接赋值转换。

10) 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，我们可以自动参数随机的想要的数据。

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。

11) 以下代码中，x 取值的概率 x dist {0:=1,[1:3]:=1};

X 取 0/1/2/3 的值分别是 1/4。

12) 验证环境中 C 如何 access 和 dut 中寄存器，是如何联系的，以及 DPI 接口使用方法

Sv 中可以使用 DPI 接口来与 C 进行一个连接调用，通过 DPI 接口，我们可以实现 C 与 dut 的一个交互。

DPI 使用方法：首先，我们的 C 程序最好是不耗时的。通过调用 DPI 接口，

使用 import “DPI-C”function int factorial(input int i)；通过 inport 关键字就可以申明 function 使用了其他语言实现，C。注意的是 input 指的是数据从 SV 流向 C，且参数不能定义为 ref 类型。除此之外，保证调用的参数是 SV 和 C 都兼容的。DPI 只能调用 C 中静态的方法，因此不可以调用对象中的方法，解决办法是创建和 C 中对象方法进行通信的静态方法。通过创建一个对象并返回其句柄 ，然后在一个静态方法里面使用对象句柄来调用 C++对象中的方法。

2. UVM 的理解考察

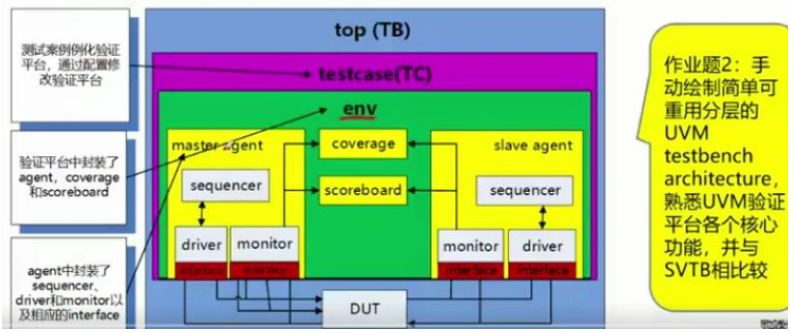
1) 请谈一下你对 UVM 验证方法学的理解

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。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。

2) 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。

3) 举例说明 UVM 组件中常用的方法，各种 phase 关系，phase 机制作用。 0516

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 functtion，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，不通组件中的同一个 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。

4) 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）。

5) 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。

6) 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）。

7) 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。

8) 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。

9) Virtual sequence 和 virtual sequencer 中 virtual 含义

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。

10) 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。

11) 请简述一下一个 slave vip 的 dataflow

首先是先要明白 TB 的一个运行机制，TB 从 top 层 run_test()进行调用 test case 开始运行，通过 UVM 的 phase 机制进行层次化的仿真。数据流首先是 driver 提出请求获取 item，sequennce 的 body 函数产生随机化的 item 后，通过 sequencer 发送给了 driver，driver 转换为 DUT 接口信号进行驱动，发送完后会继续进行下一次请求，知道 sequence 中 body 函数结束，则平台结束；然后 monitor 进行采集并转化为 item，将其发给 model 和 scoreboard，最后在 scoreboard 中进行比较。

12) 为什么要避免绝对路径的使用，如何避免？

绝对路径我们可以叫做硬代码，其可重用性不高，很多时候需要更改多个地方才可以，容易出错。避免的话，通过一层一层嵌套传递，而不是直接顶层跨越传递，还有就是使用通配符进行传递。

13) 如何在 driver 中使用 interface,为什么 0514

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。

14) 你了解 uvm 的 factory 和 callbback 机制吗

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 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

15) OVM 和 UVM 有什么区别

UVM 基本完全继承了 OVM 的特性，不过在此基础上进行了一些改进。其中 sequence 和 sequencer 的注册方式变更是改变之一，以前的 ovm 注册 sequence 时不仅实现注册到 factory 中，而且还会与 sequencer 绑定，将所有的 sequence 都加入到 sequencer 的一个静态数组中，有点类似于现在的 sequence lib，但是功能简单。

16) UVM 中各 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。

17) UVM 中如何启动一个 sequence 0514

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。

18) 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 去使得 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);

19) 如果将一个模块的寄存器接口由 APB 变为 AXILite，在寄存器模型的使用中有什么需要改变？

首先要注意 APB 和 AXILite 总线和接口的区别，APB 接口比较简单，其主要是有地址、数据、读写、使能端口，只要满足读写和使能的时序就可以进行读写数据，基本没有其他信号了。AXI 总线接口比较复杂，其将信号分为 5 个通道，分别为读地址、读数据、写地址、写数据、应答通道，每个通道都可以进行握手，可以多个通道互不干扰的传输，且其控制信号比如突发传输等，而 APB 不能突发传输。

因此，首先，寄存器模型的使用：总线传输要改变，加入其余通道，寄存器的 item 类要更改。其次，还需要加入更多的控制功能，比如 AXI 的突发和应答机制。寄存器模型中的 field 级别和 reg 级别都需要修改符合 AXI 控制的要求。

20) 你所搭建的验证平台为什么要用 RAL

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。

21) 如何确保你的电路完全正确，做了什么验证？一些 corner case 如何解决的 0514

一般来说，第一步我会在看完文档后，根据 spec 文档的功能点去列举我需要测试编写的 test_case 和 sequence，然后去运行，确保 TB 运行没有问题，波形没问题后，检查运行后的代码覆盖率和功能覆盖率还有断言覆盖率，分析覆盖率没

有达到的原因，比如代码覆盖率低功能覆盖率高（case 写少了，功能覆盖 point 少了）、或者代码覆盖率高功能覆盖率低（设计是否完整，测试是否多样性，臃肿的 coverpoint）等，然后去添加 case 或者修改 DUT，继续测试，更换 seed 测试，当覆盖率不在增加时，我们需要去分析是那些信号没有覆盖到，根据这些去分析出是需要什么激励，然后可以编写一些定向测试去覆盖，这样去提高覆盖率，将一些边界情况 corner case 解决，最后还需要做回归测试，随机化不同的随机种子进行验证。

3. 验证思想

1) 请描述你所验证的模块功能

我验证的是一个基于 AHB 总线的 sram 模块，该模块主要功能是有两方面：1 是作为一个 AHB 从机，去与 AHB 总线进行通信，二是 sram 的低功耗片选和 memory bist 的功能。作为 AHB 从机主要实现单周期读写、8/16/32bit 读写位宽、连续与非连续读写，非递增和递增 4 读写；作为 sram 要能够根据 ADDR 进行低功耗片选的功能和能够切换到 memory bist 测试。

2) 请谈谈验证的思想，验证人员和设计人员思考的差异

我觉得验证是目标驱动行动的，验证的主要目标是追求按时、保质、低耗的结果，最快的速度，最小的成本，找出尽可能多的 bug，来保证设计的正确。

因此，基于这个目标，我们需要能够评判我们达到这个目标的指标，现在一般是采用基于覆盖率驱动的随机验证，通过产生随机化激励，去测试我们的 DUT，然后收集相应的覆盖率，尽可能使得覆盖率 100%。

差异的话，设计人员主要考虑的是如何去通过 RTL 语言实现模块的功能，怎么去更好的设计实现以达到设计的要求。验证人员主要考虑的是该设计是否有 bug，bug 在哪里，怎么去产生测试激励去找出 bug 或者碰出 bug 来。

3) 你写过 assertion 吗？assertion 分几种？简述一下 assertion 的用法

写过。

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。

4) 断言 and 和 intersect 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

5) 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。

6) 你们项目中都会考虑哪些 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。

7) 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。

8) Function coverage 和 code coverage 的区别，以及听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。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。

9) 对一个加密模块和一个解密模块进行验证，如果数据先进行加密紧接着进行解密，如果发现输入数据和输出数据相同，能不能说明这两个模块功能没问题，为什么？

不能。因为如果我们不对中间进行插入 monitor 监测中间结果的话，可能加密解密都没工作，这样也是输入数据和输出数据相同，但其实两个模块都没有正常工作，除此之外，还有可能虽然工作了但是是错误的，但碰巧输出和输入一致。一般我们会分开验证，最后在集成验证，也可以在模块中间插入 monitor 进行监测。

10) 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他 bug；第七步就是总结。

11) 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。

12) 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，导致我认为没错。

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。

13) 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。

4. 实例题

1) 分析代码覆盖率时，verilog 语句 if(a || (b && c))有哪几种条件需要覆盖？请用表格列出每种条件下 abc 的值，是 0 是 1 无所谓请用*表示

a = 1 b /c 任意、a = 0 b = 1 c =1

5. 项目 debug 心得总结

..\..\word\设计与验证\soc 项目总结\ahb_sram\ahb_apb_sram.docx

1) 参数传递不成功：在调试过程中，由于编写了 ahb_config 类，在最后又例化了他的扩展类，可是却忘记传递参数，导致运行的是基类（因此编写了传递不成功就使用基类的函数），从而修改一直无效。Config 传递不成功还是要报错误最好。
2) 关联数组超出范围报错：原因是因为我对于 32bit 的位宽数据，有时为了简单定义成了 int 类型，导致问题出错。Int 是有符号数，bit[31:0]是无符号数。
3) 数据接口信号设置不对，导致一直处于某个触发或者等待状态，从而可能导致 TB 一直运行，没有 sequence 发送等情况。根据波形图和 driver 等组件时序设置，一对一进行排查。
4) 一定要在结束时设置 set_drain_time()函数，设定一个结束时间，这样才能保证最后一个包能够正常发送采集比较，不然发送后会直接结束，导致结果出错。

5) Scoreboard 中不能仅仅只是统计 match 和 mismatch 的个数，还需要比较得到的 match 个数与本应该发送的个数比较，因为有时候虽然没有 mismatch，但是可能压根就没有发送出来或者没搜集到。
6) 时序分析不对导致的 bug，解决办法就是根据时序图去和 driver，还有 monitor 进行一步一步的比较分析。
7) Run_phase 和 main_phase 混乱使用导致 TB 出现 bug，现在还是没有解决。提出的解决办法是：当使用 run_phase 时，不使用其他 12 个并行的小 phase; 使用其他 12 个 phase 如 main_phase 时不使用 run_phase。

6. 脚本语言题型复习

1) Perl 正则表达式的相关问题

2) Perl 标量和数组还有 hash 表示

3) Perl 循环的控制语句有哪三个，有哪些循环

4) Perl 条件控制语句有哪些

5) Perl 子函数如何调用

6) Perl 文件读写和文件夹读写操作方法