**Q1.定宽数组、动态数组、关联数组、队列各自的特点和使用方式。**

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[$]这样的符号进行申明: int q[$]，使用广泛，多应用与scoreboard;

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压缩数组（存储空间连续）; bit[7:0] name [3:0]非压缩数组（存储空间非连续）;

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，该部分的重点要学习关联数组的遍历方法。

**Q2.fork...join/fork...join\_any/fork...join\_none之间的异同**

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。

**Q3.mailbox、event、semaphore之间的异同**

mailbox: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。

**Q4.@(event\_handle)和wait(event\_handle.triggered)区别**

[@(event\_handle):](javascript:void(0);)边沿触发，事件会一直等待，直到触发为止，阻塞型；

wait(event\_handle.triggered)：电平触发，如果事件在当前已经被触发，则不会引起阻塞，非阻塞型，当event被多次触发时，避免使用wait.

**Q5.task和function异同区别**

function:至少要有一个输入变量,一个返回值,只能运用于纯粹的数字或者逻辑运算 ；

task:可以内置常用的耗时语句,可能会被运用于需要耗时的信号采样或者驱动场景。

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，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.

**Q6.使用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，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。

**Q7.同步FIFO和异步FIFO的作用和区别**

FIFO在硬件上是一种地址依次自增的Simple Dual Port RAM，按读数据和写数据工作的时钟域是否相同分为同步FIFO和异步FIFO，其中同步FIFO是指读时钟和写时钟为同步时钟，常用于数据缓存和数据位宽转换；异步FIFO通常情况下是指读时钟和写时钟频率有差异，即由两个异步时钟驱动的FIFO，由于读写操作是独立的，故常用于多比特数据跨时钟域处理。

**Q8.SystemVerilog中OOP的三大特性**

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。

**Q9.详述对于ref类型的理解**

ref参数类型是引用

向子程序传递数组时应尽量使用ref获取最佳性能，如果不希望子程序改变数组的值，可以使用const ref类型

在任务里可以修改变量而且修改结果对调用它的函数随时可见。

**Q10.外部约束如何使用，有哪几种方式**

使用方法例示：

class Packet;

rand bit [7:0] length;

rand bit [7:0] payload[];

constraint c\_valid {length > 0;payload.size() == length;}

constraint c\_external;

endclass

constraint Packet::c\_external {length == 1;} //外部约束

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n/num。

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

**Q11.代码覆盖率、功能覆盖率、SVA覆盖率都是衡量什么的**

代码覆盖率：是针对RTL设计代码的运行完备度的体现，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，只要仿真就可以收集，可以看DUT的哪部分代码没有动，如果有一部分代码一直没动看一下是不是case没有写到。

功能覆盖率：与spec比较来发现，design是否行为正确，需要按verification plan来比较进度。用来衡量哪些设计特征已经被测试程序测试过的一个指标，首要的选择是使用更多的种子来运行现有的测试程序；其次是建立新的约束，只有在确实需要的时候才会求助于定向测试，改进功能覆盖率最简单的方法是仅仅增加仿真时间或者尝试新的随机种子。验证的目的就是确保设计在实际环境中的行为正确。设计规范里详细说明了设备应该如何运行，而验证计划里则列出了相应的功能应该如何激励、验证和测量

断言覆盖率：用于检查几个信号之间的关系，常用在查找错误，主要是检查时序上的错误，测量断言被触发的频繁程度。

**Q12.为什么选择验证工作**

因为喜欢，因为热爱，因为行业需要；

**Q13.立即断言和并发断言的特点**

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：

第一是布尔表达式，布尔表达式是组成断言的最小单元，断言可以由多个逻辑事件组成，这些逻辑事件可以是简单的布尔表达式.在SVA中，信号或事件可以使用常用的操作符，如：&&, ||, !, ^，&等；

第二个序列sequence编写，sequence是布尔表达式更高一层的单元，一个sequence中可以包含若干个布尔表达式，同时在sequence中可以使用一些新的操作符，如 ## 、重复操作符、序列操作符;

第三个是属性property的编写，property是比sequence更高一层的单元，也是构成断言最常用的模块，其中最重要的性质是可以在property中使用蕴含操作符(|-> |=>);

**Q14.SystemVerilog中面向对象编程的优势**

1、易维护

采用面向对象思想设计的结构，可读性高，由于继承的存在，即使改变需求，那么维护也只是在局部模块，所以维护起来是非常方便和较低成本的。

2、质量高

在设计时，可重用现有的，在以前的项目的领域中已被测试过的类使系统满足业务需求并具有较高的质量。

3、效率高

在软件开发时，根据设计的需要对现实世界的事物进行抽象，产生类。使用这样的方法解决问题，接近于日常生活和自然的思考方式，势必提高软件开发的效率和质量。

4、易扩展

由于继承、封装、多态的特性，自然设计出高内聚、低耦合的系统结构，使得系统更灵活、更容易扩展，而且成本较低。

**Q15.如何保证验证的完备性**

首先不可能百分百完全完备，即遍历所有信号的组合，这既不经济也不现实。所以只能通过多种验证方法一起验证尽可能减少潜在风险，一般有这些验证流程：ip级验证、子系统级验证、soc级验证，除这些以外，还有upf验证、fpga原型验证等多种手段。前端每走完一个阶段都需要跟设计以及系统一起review验证功能点，测试用例，以及特殊情况下的波形等。

芯片后端也会做一些检查，像sta、formality、DFM、DRC检查等，也会插入一些DFT逻辑供流片回来测试用。流片归来进行测试，有些bug可以软件规避，有些不能规避，只能重新投片。

**Q16.权重约束中”:=”和”:/”的区别**

: =操作符表示值范围内的每一个值的权重是相同的,比如[1:3]:=40,表示1，2，3取到的概率为40/120；

:／操作符表示权重要平均分到值范围内的每一个值,比如[1:3]:/60,表示1，2，3取到的概率为20/60。

**Q17.rand和randc区别**

rand: 表示随机，比如有56个数，每次的随机概率都是1/56；

randc: 表示周期性随机，比如56个数，第一次随机的概率为1/56，第二次随机的概率为1/55，以此类推。

**Q18.break、continue和returen的含义用法**

break 语句结束整个循环。

continue 立即结束本次循环，继续执行下一次循环。

return 语句会终止函数的执行并返回函数的值(如果有返回值的话)。

**Q19.function中return语句执行之后，function里剩下的代码语句还会执行吗**

return之后，function里剩下的语句不能执行，其是终止函数的执行，并返回函数的值。

**Q20.触发器和锁存器的区别**

触发器: 时钟触发，受时钟控制，只有在时钟触发时才采样当前的输入，产生输出。

锁存器: 由电平触发，非同步控制。在使能信号有效时锁存器相当于通路，在使能信号无效时锁存器保持输出状态。触发器由时钟沿触发，同步控制。

锁存器对输入电平敏感，受布线延迟影响较大，很难保证输出没有毛刺产生；触发器则不易产生毛刺。

**Q21.为什么要使用两级触发器进行同步**

使用二级触发器进行同步可以极大的避免亚稳态的出现：

至于为什么“打两拍”可以减少亚稳态，我觉得根本原因是在于寄存器是一个双稳态器件，只有0和1两种稳定的状态。在绝大多数时候，第一级Flop的Q端，即使出现亚稳态，也会在一个时钟周期内稳定下来，这样第二级Flop所采到的就是一个稳定的值。“打两拍”就是利用这一点，切断了亚稳态的传播。

**Q22.setup和hold违例分别是什么**

setup time是指在时钟有效沿之前，数据输入端信号必须保持稳定的最短时间。

hold time是指在时钟有效沿之后，数据输入端信号必须保持稳定的最短时间。

hold time时序检查确保新数据不会在触发器稳定输出初始数据之前过早到达D端而覆盖其初始数据。

总结为一句话：当前待传输的数据，相对于Capture edge来说，必须早来（setup time）晚走（hold time）。

**Q23.SVA中and、intersect、or、throughout、$past如何使用**

and：可以按照逻辑与的方式组合两个序列，当两个序列都匹配时，整个属性成功。

**示例：**

property p；

sequence\_a and sequence\_b；

endproperty

【注意】sequence\_a和sequence\_b的检查起始点必须一样，但是两者结束的结束点可以不一样，整个属性的成功点以两个sequence中最后一个成功的sequence的匹配点为属性的成功点。

or：可以按照逻辑或的方式组合两个序列，当两个序列有一个匹配时，整个属性成功。

**示例：**

property p；

sequence\_a or sequence\_b；

endproperty

【注意】sequence\_a和sequence\_b的检查起始点必须一样，但是两者结束的结束点可以不一样，整个属性的成功点取决于最早匹配成功的sequence，只要两个序列有一个匹配成功，整个属性就认为匹配。

intersect两侧的表达式都是sequence，不能是property，使用intersect时需要确保其两侧的sequence必须同时开始，且当两个sequence最终同时匹配时才认为两个序列的intersect匹配，即两个sequence匹配的长度必须相同，intersect操作符和and操作符很相似，但是and没有要求两个序列最终匹配时同时匹配。

在SVA中有时需要某些信号在一个序列的检查过程中一直保持一个状态，此时就可以使用throughout操作符，当这些信号的状态发生不期望的变化时，序列的检查即认为失败。

[$past](javascript:void(0);)主要用于检查当前表达式前一个时钟周期(如果不指定number\_of\_ticks，默认指向前一个时钟周期)值是否为真，如果位真则$past返回为真。其格式如下：

$past(expression[,number\_of\_ticks]);

其中number\_of\_ticks可以指定检查当前时刻之前number\_of\_ticks个周期的采样值.

**Q24.SVA中”->”和”=>”区别**

@ (posedge clk) a |-> b 断定clk上升沿后，a事件"开始发生"，同时，b事件发生

@ (posedge clk) a |=> b 断定clk上升沿后，a事件开始发生，下一个时钟沿后，b事件开始发生。

**Q25.如何关闭约束**

根据不同需要，来选择使能哪些约束块，禁止哪些约束块的要求，可以使用内建的constrai nt\_mode()函数打开或者关闭多个约束块；rand\_mode()函数打开或者关闭随机变量。

**Q26.deep copy和shallow copy区别**

如果拷贝的对象里的元素只有值，没有引用，那浅拷贝和深拷贝没有差别，都会将原有对象复制一份，产生一个新对象，对新对象里的值进行修改不会影响原有对象，新对象和原对象完全分离开。

如果拷贝的对象里的元素包含引用（像一个列表里储存着另一个列表，存的就是另一个列表的引用），那浅拷贝和深拷贝是不同的，浅拷贝虽然将原有对象复制一份，但是依然保存的是引用，所以对新对象里的引用里的值进行修改，依然会改变原对象里的列表的值，新对象和原对象完全分离开并没有完全分离开。而深拷贝则不同，它会将原对象里的引用也新创建一个，即新建一个列表，然后放的是新列表的引用，这样就可以将新对象和原对象完全分离开。

**Q27.队列常用的方法有哪些**

队列常用的使用方法：insert，delete, push\_back和pop\_front

Push插入，pop取出

Front前边，back后边

**Q28.local和protected区别**

类中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。

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。

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。

**Q29.常用的debug方法有哪些**

1.打印显示

2.设置断点

3.波形分析

**Q30.亚稳态的危害**

亚稳态是指触发器无法在某个规定时间段内达到一个可确认的状态。

亚稳态违背了时序，无法在规定的时间内达到稳定，且亚稳态具有传输功能，会传递给后一级的触发器，从而导致电路出错。亚稳态最后会稳定下来，但是所需的时间较长。

**Q31.二进制码、格雷码、独热码的特点**

二进制码：基本的机器语言，每一位只能是0或1，逢二进一；

格雷码：在一组数的编码中，若任意两个相邻的代码只有一位二进制数不同，则称这种编码为格雷码（Gray Code），另外由于最大数与最小数之间也仅一位数不同，即“首尾相连”。

独热码：又称一位有效编码，其方法是使用N位状态寄存器来对N个状态进行编码，每个状态都由他独立的寄存器位，并且在任意时候，其中只有一位有效。

**Q32.packed array和unpacked array的区别**

packed array的声明方式：bit [7:0][3:0] packed\_array:表示packed\_array有4个8比特数组成，且存储空间连续；

unpacked array的声明方式：bit [7:0] unpacked\_array[3:0]:表示unpacked\_array有4个8比特数组成，存储空间不连续；

**Q33.阻塞赋值和非阻塞赋值的区别**

阻塞赋值语句（“=”）和非阻塞赋值语句（“<=”）

阻塞：一般对应电路中的组合逻辑赋值，等号右端的结果会立刻赋值给左端。

非阻塞：一般对应电路中的时序逻辑赋值，等号右端的结果不会立刻赋值给左端。

在always语句中，阻塞赋值等号左端的参数如果参与该模块的其他运算，则按照赋值后的结果参与运算，而非阻塞赋值等号左端的参数依旧按照未赋值前的结果参与运算。

Assign语句中用阻塞赋值，

组合电路用阻塞赋值，时序电路用非阻塞赋值

**Q34.过程性语句和连续赋值语句的区别**

连续赋值语句由assign来标示，而过程赋值语句不能包含这个关键词；

连续赋值语句中左侧的数据类型必须是线网数据类型，而过程赋值语句中的被赋值数据类型则必须是寄存器类型的变量；

连续赋值语句不能出现在过程块（initial或always）中，而过程赋值语句可以；

连续赋值语句主要用来对组合逻辑电路进行建模以及对线网数据间的连接进行描述，而过程赋值语句主要用来对时序逻辑电路进行行为描述；

连续赋值语句对被赋值线网型数据的赋值是“连续”的（即赋值表达式的任何变化都会在立刻反应在线网数据的取值上），而过程性赋值语句，只有在过程赋值语句被执行时才执行赋值操作，语句执行完后被赋值变量的取值不再受到赋值表达式的影响（注意这里的一次是指：在initial块中，过程性赋值只顺序执行一次，而在always块中，每一次满足always的条件时，都要顺序执行一次该always块中的语句。）。

**Q35.initial和always的异同**

always：

always是为了描述硬件的行为，而在使用时需要注意哪种使用方式是时序电路描述，哪种使用方式是组合电路描述。

always中的@(event..)敏感列表是为了模拟硬件信号的触发行为，需要正确对标硬件行为和always过程块描述。

所以说，always过程块是用来描述硬件时序电路和组合电路的正确打开方式，因此只可以在module或者interface中使用。

initial：

initial从名字也可以看得出来，与always在执行路径上有明显区别，即initial非常符合软件的执行方式，即只执行一次。

initial和always一样，无法被延迟执行，即在仿真一开始它们都会同时执行，而不同initial和always之间在执行顺序上是没有顺序可言的。

initial从其执行路径的属性来看，它不应该存在于硬件设计代码中，它本身不可综合，对于描述电路没有任何帮助。

initial就是为了测试而生的，由于测试需要按照时间顺序的习惯即软件方式来完成，所以initial便可以实现这一要求。

在Verilog时代，所有的测试语句都可以被放置在initial中，为了便于统一管理测试顺序，建议将有关测试语句都放置在同一个initial过程块中。

initial过程块可以在module、interface和program中使用。对于过程块的书写方式，请记住用begin..end将其作用域包住。

**Q36.FSM有哪几种？区别是什么？**

状态机描述时关键是要描述清楚几个状态机的要素，即如何进行状态转移，每个状态的输出是什么，状态转移的条件等。具体描述时方法各种各样，最常见的有三种描述方式：

（1）一段式：整个状态机写到一个always模块里面，在该模块中既描述状态转移，又描述状态的输入和输出；

（2）二段式：用两个always模块来描述状态机，其中一个always模块采用同步时序描述状态转移；另一个模块采用组合逻辑判断状态转移条件，描述状态转移规律以及输出；

（3）三段式：在两个always模块描述方法基础上，使用三个always模块，一个always模块采用同步时序描述状态转移，一个always采用组合逻辑判断状态转移条件，描述状态转移规律，另一个always模块描述状态输出(可以用组合电路输出，也可以时序电路输出)。

**Q37.数字电路中为什么要使用触发器**

用触发器是因为触发器能保存数据，保存电路状态；触发器是在时钟边沿触发，用时钟同步是让整个电路能同步整齐划一的工作；另外，如果只用组合逻辑，无法保存数据，这样得到的数据无法供以后使用。

Q38.异步复位和同步复位各自特点和区别

同步复位：当时钟上升沿检测到复位信号，执行复位操作，时钟沿有效是前提 。always@(posedge clk);

优点：

a、有利于仿真器的仿真；

b、可以使所设计的系统成为 100% 的同步时序电路，有利于时序分析，而且可综合出较高的 Fmax；

c、由于只在时钟有效电平到来时才有效，所以可以滤除高于时钟频率的复位毛刺。

缺点：

a、复位信号的有效时长必须大于时钟周期，才能真正被系统识别并完成复位任务。同时还要考虑诸如 clk skew 、组合逻辑路径延时 、复位延时等因素（所以复位信号有时需要脉冲展宽，用以保证时钟有效期间有足够的复位宽度）；

b、由于大多数的逻辑器件的目标库内的 DFF 都只有异步复位端口，所以，倘若采用同步复位的话，综合器就会在寄存器的数据输入端口插入组合逻辑，这样就会一方面额外增加FPGA内部的逻辑资源，另一方面也增加了相应的组合逻辑门时延。

异步复位：无论时钟沿是否到来，只要复位信号有效，就对系统进行复位。always @ ( posedge clk or negedge rst\_n );

优点：

a、大多数目标器件库的 DFF 都有异步复位端口，那么该触发器的复位端口就不需要额外的组合逻辑，这样就可以节省资源；

b、设计相对简单；

c、异步复位信号识别方便（电路在任何情况下都能复位而不管是否有时钟出现）。

缺点：

a、最大的问题在于它属于异步逻辑，问题出现在复位释放时，而不是有效时，如果复位释放接近时钟有效沿，则触发器的输出可能进入亚稳态（此时 clk 检测到的 rst\_n 的状态就会是一个亚稳态，即是0是1是不确定的），从而导致复位失败。

b、可能因为噪声或者毛刺造成虚假复位信号（比如以前的游戏机玩到一半突然复位）（注意：时钟端口、清零和置位端口对毛刺信号十分敏感，任何一点毛刺都可能会使系统出错，因此判断逻辑电路中是否存在冒险以及如何避免冒险是设计人员必须要考虑的问题）；

c、静态定时分析比较困难。

d、对于 DFT (Design For Test可测性设计)设计，如果复位信号不是直接来自于 I/O 引脚，在 DFT 扫描和测试时，复位信号必须被禁止，因此需要额外的同步电路。

推荐使用异步复位、同步释放的方式，并且复位信号为低电平有效

**Q39.异步复位同步释放代码实现**

Q40.数字电路通常分为哪两种电路

分为组合逻辑电路和时序逻辑电路。

组合逻辑电路在逻辑功能上的特点是任意时刻的输出仅仅取决于该时刻的输入，与电路原来的状态无关。

而时序逻辑电路在逻辑功能上的特点是任意时刻的输出不仅取决于当时的输入信号，而且还取决于电路原来的状态，或者说，还与以前的输入有关

**Q41.illegal\_bins和ignore\_bins命中分别会怎么样？命中是否会计入覆盖率统计**

illegal\_bins 表示非法的bin，如果采样命中会报错；

ignore\_bins 表示忽略的bin，如果采样命中直接忽略，并不计入覆盖率统计。

**Q42.负数采用二进制如何表示**

负数采用二进制的补码形式表示，最高位的0表示正，1表示负；

例如-1，1的原码为0001，反码为1110，补码为1111，,1111表示-1.

**Q43.4值逻辑变量赋值给二值逻辑变量时，x和z对应什么值**

对应0

**Q44.类中this是什么**

this表示指向本身的指针，通过this可以绑定方法和对象，可以区分方法体的形参和对象成员，this只能作为实参传递，this的生命周期从调用方法开始存在，当方法结束生命周期也随之结束，this只能应用于动态生命周期，不能应用于static中。

**Q45.子类中super是什么？**

super主要用于在子类中访问父类的属性和方法。

**Q46.在IC验证中，我们一般对哪些内容进行随机化**

器件配置：通过寄存器和系统信号；

环境配置：随机化验证环境，例如：合理的时钟和外部反馈信号；

原始输入数据：例如数据包的长度，宽度，数据发送的顺序；

延时：握手信号之间的时序关系，例如：vaild和ready，req和ack之间的时序关系；

异常协议

**Q47.通过函数返回数组有哪些方法**

可以通过for/foreach遍历数组的方法打印返回

**Q48.什么是clocking block的skew**

在clocking block中，实际上指定了默认的input偏差和output偏差，只不过这个偏差在宏观的时间刻度上观察不到，对输入性质的信号指定的默认采样时间是1step，对于输出性质的信号指定的默认驱动时间为0，其中1step是一个较为特殊的时间单位，一般指输入信号在采样信号事件发生之前的输入信号的稳定值，可以认为1step实际采样时刻发生在上一时间槽的最后时刻，1step不像ns、ps等之类可以用来设置设计的时间精度和时间单位，它指定了当前采样值来自于当前时间槽的pre-pone区，0则表示在采样事件发生后立刻将对应的值驱动到输出端口，与1step不同的是，该时刻的采样发生在采样事件发生的当前时间槽的Observed Region。

另外，也可以自定义，例如：default input 1ns output 1ns

**Q49.并发断言的主要组成有哪些**

并发断言的用法的话，主要是有三个层次：

第一是布尔表达式，布尔表达式是组成断言的最小单元，断言可以由多个逻辑事件组成，这些逻辑事件可以是简单的布尔表达式.在SVA中，信号或事件可以使用常用的操作符，如：&&, ||, !, ^，&等；

第二个序列sequence编写，sequence是布尔表达式更高一层的单元，一个sequence中可以包含若干个布尔表达式，同时在sequence中可以使用一些新的操作符，如 ## 、重复操作符、序列操作符;

第三个是属性property的编写，property是比sequence更高一层的单元，也是构成断言最常用的模块，其中最重要的性质是可以在property中使用蕴含操作符(|-> |=>);

**Q50.如何检查随机化是否成功**

可以通过立即断言检查随机化是否成功

例如 assert（p.randomize()) 来检查p句柄指向的实例是否实例化成功；

**Q51.什么时候randomize()失败**

1.变量对象没有声明rand或randc；

2.constraint 有冲突，无法随机出值；

**Q52.黑盒验证、灰盒验证、白盒验证**

黑盒测试也称功能测试，它是通过测试来检测每个功能是否都能正常使用。在测试中，把程序看作一个不能打开的黑盒子，在完全不考虑程序内部结构和内部特性的情况下，在程序接口进行测试，它只检查程序功能是否按照需求规格说明书的规定正常使用，程序是否能适当地接收输入数据而产生正确的输出信息。黑盒测试着眼于程序外部结构，不考虑内部逻辑结构，主要针对软件界面和软件功能进行测试。黑盒测试是以用户的角度，从输入数据与输出数据的对应关系出发进行测试的。很明显，如果外部特性本身设计有问题或规格说明的规定有误，用黑盒测试方法是发现不了的。

白盒测试又称结构测试、透明盒测试、逻辑驱动测试或基于代码的测试。白盒测试是一种测试用例设计方法，盒子指的是被测试的软件，白盒指的是盒子是可视的，你清楚盒子内部的东西以及里面是如何运作的。"白盒"法全面了解程序内部逻辑结构、对所有逻辑路径进行测试。"白盒"法是穷举路径测试。在使用这一方案时，测试者必须检查程序的内部结构，从检查程序的逻辑着手，得出测试数据。贯穿程序的独立路径数是天文数字。

灰盒测试，是介于白盒测试与黑盒测试之间的一种测试，灰盒测试多用于集成测试阶段，不仅关注输出、输入的正确性，同时也关注程序内部的情况。灰盒测试不像白盒那样详细、完整，但又比黑盒测试更关注程序的内部逻辑，常常是通过一些表征性的现象、事件、标志来判断内部的运行状态。

**Q53.竞争与冒险是什么**

竞争： 信号由于不同路径传输到达某一汇合点的时间有先有后的现象。（当一个门的输入有两个或两个以上的变量发生改变时，由于这些变量是经过不同路径产生的，使得他们的状态改变的时刻有先有后，这种时差引起的现象称为竞争。）

冒险： 由于竞争现象所引起的电路输出发生瞬间错误（产生干扰脉冲）的现象。

**Q54.虚接口有什么好处**

interface是连接硬件的，其是硬件语言；但对于验证来说，其描述语言往往是软件语言，interface无法在基于OOP的测试平台中实例化，因此我们无法通过interface把激励传送到DUT中；为了解决这个问题，引入了virtual interface，使得基于OOP的验证环境可以通过虚接口把激励传送给DUT。

**Q55.接口的使用有什么优势**

将有关信号封装在同一个接口中， 对于设计和验证环境都便于维护和使用。 如果你需要新添加信号， 只需要在接口中定义这个信号， 而在使用这个接口的模块或者验证环境中做出相应修改。

由于接口既可以在硬件世界(module)中使用，又可以在软件世界(class)中使用，interface作为SV中唯—的硬件和软件环境的媒介交互，它的地位不可取代。

接口由于可以例化的特性， 使得对于多组相同的总线， 在例化和使用时变得更加灵活，不仅使得代码变得简洁， 也更易于验证环境的管理和维护。

**Q56.$cast在句柄转换时如何使用**

在SV中，子类的句柄可以直接赋值于父类的句柄，父类的句柄指向子类的对象，这是允许的。但是如果反过来，将子类的句柄指向父类的对象，是非法的，编译器在编译过程中会报错，如果使用$cast进行转换，$cast(子类句柄，父类句柄）编译器在编译过程中会编译通过，实现父类句柄赋值于子类句柄的强制转换。

**Q57.为什么要进行后仿真**

后仿真也称为时序仿真或者布局布线后仿真，是指电路已经映射到特定的工艺环境以后，综合考虑电路的路径延迟与门延迟的影响，验证电路能否在一定时序条件下满足设计构想的过程，是否存在时序违规。

**Q58.如何进行后仿真**

先做zero\_delay仿真，确保加载的网标文件是正确的，然后，再将SDF文件反标到网标文件上。

**Q59.什么是后仿真**

后仿真也称为时序仿真或者布局布线后仿真，是指电路已经映射到特定的工艺环境以后，综合考虑电路的路径延迟与门延迟的影响，验证电路能否在一定时序条件下满足设计构想的过程，是否存在时序违规。其输入文件为从布局布线结果抽象出来的门级网表、testbench和扩展为sdo或sdf的标准时延文件。sdo、sdf的标准时延文件不仅包含门延迟，还包括实际布线延迟，能较好地反映芯片的实际工作情况。一般来说后仿真是必选的，检查设计时序与实际的FPGA运行情况是否一致，确保设计的可靠性和稳定性。

**Q60.当task的通过ref传递数据时，如果task内部对数据进行了修改，task外部是否立即可以看到数据被修改了还是要等到task执行完才能看到**

向子程序传递数组时应尽量使用ref获取最佳性能，如果不希望子程序改变数组的值，可以使用const ref类型

在任务里可以修改变量而且修改结果对调用它的函数随时可见。

**Q61.使用packed struct定义下面数据包：**

31:24 23:16 15:4 3:0

p1 p2 p3 p4

**Q62.随机化的优势是什么？是不是意味着不再需要定向case了**

随机测试相比于定向测试可以减少相当多的代码量，而产生的激励较定向测试也更加多样化；

但是有一些随机覆盖不到的点在验证后期还需要验证人员便编写定向case去完成。

**Q63.randomize with{....}中的约束与class中的约束是什么关系**

SV 中允许使用randomize with{...}的内嵌约束方法在来增加额外的约束，这和在类中声明的约束是等效的，但是此时要注意内部约束和外部约束之间应该是协调的，如果出现相互违背的情况，那么随机值求解就会失败，一般在有外部约束的情况下，习惯性的将内部约束声明为软约束类型。

**Q64.如何基于随机化的验证环境写定向测试**

通过对随机测试的分析，找出随机测试没有覆盖到的地方，有针对性的编写定向测试case，进行验证。

**Q65.为什么数字电路系统中要使用二进制**

数字电路是由模拟电路转换过来的，在模拟电路中通常会使用到的高低电平在数字电路中表现为1和0，在逻辑电路的使用中，只有断开和闭合两种状态，所以使用二进制即可。（个人理解）

**Q66.数字电路中可能存在的风险问题有哪些？**

常见的风险就是竞争冒险，当一个信号在经过不同的路径时，到达输出端的时间不一致，会产生延迟，从而引起竞争，使得电路出现瞬时的错误。另外，由于时钟信号的skew hold要求，可能在异步电路中出现亚稳态的情况，由于亚稳态具有传递性，所以会影响电路的稳定性，造成错误。

**Q67.什么是虚方法**

将父类的方法前面加virtual修饰，定义为虚方法，在子类中定义与父类相同变量和方法名的方法时，当父类的句柄指向子类的对象时，仿真器会动态的查找子类中与父类具有相同名称的方法。

简而言之，虚方法允许不同类型的句柄去访问属于该类的方法，比如父类例化的句柄，当把它的类型指定为子类类型，那么它就可以访问子类的方法。也就说以类型去调用方法，这就是虚方法。

**Q68.低功耗方法你了解哪些**

正在了解，过后更新

**Q69.使用FSM设置序列检测器（序列：110110）**

wait update

**Q70.描述你对数字集成电路验证流程的认识**

1、搞清楚要验证的东西

对设计spec进行阅读和理解，把DUT的结构、功能，时序弄清楚。

2、编写验证计划，指导性的文件

（1）提取验证功能点

（2）明确DUT接口信号（所有信号的名字，位宽，功能，时序关系等

（3）TB的架构（能够描述每一个组件的功能）

（4）检查点（check point）

（5）功能覆盖率（覆盖点）

（6）测试用例的规划（testcase尽可能的规划完整）

（7）结束标准

3、搭建TB&Debug&调通第一个最基本的testcase，然后再编写测试用例debug,进行主要验证。

4、Regression（回归测试，一天一次，有随机测试）

5、分析代码/功能覆盖率，增加新的测试用例

6、测试报告，测试结果：测试用例（pass/fail），覆盖率报告

**Q71.虚接口是什么**

virtual interface

即interface是连接硬件的，其是硬件语言；但对于验证来说，其描述语言往往是软件语言，interface无法在基于OOP的测试平台中实例化，因此我们无法通过interface把激励传送到DUT中；为了解决这个问题，引入了virtual interface，使得基于OOP的验证环境可以通过虚接口把激励传送给DUT。

**Q72.预定义的随机方法有哪些**

1.$random()----平均分布，返回32位有符号随机值；

2.$urandom()----平均分布，返回32位无符号随机值；

3.$urandom\_range()----在指定范围内的随机分布；

4.$dist\_exponential()----指数衰落；

5.$dist\_normal()----钟型分布；

6.$dist\_poisson()----钟型分布；

7.$dist\_uniform()----平均分布；

**Q73.预定义的随机方法是否可以重载**

**Q74.预定义的随机方法执行顺序和执行情况**

**Q75.package用途是什么**

package类似于库，可以对类名做一个隔离，将一簇相关的类组织在单一的命名空间里，使得分属于不同模块验证环境的类来自于不同的package，可以用package来解决类的归属问题。

**Q76.package如何使用**

package用于将软件进行包装，以此来与全局的命名空间进行隔离。可以通过import对package进行引用，使package在该空间中可见，切记只是引用，而非定义，如果该空间也被定义为package，在下一个空间中引用，则第一次的package将不在可见，但是可以通过export进行重新引用，可见。

**Q77.如何在子类中调用父类中的方法**

子类可以通过使用super来调用父类的方法。

**Q78.bit[7:0]和byte有什么区别**

bit[7:0]表示8位二进制数；

byte表示一个字节，1byte=8bit；

**Q79.类中的方法和类外的方法有什么区别**

为了保证类的良好可读性，经常搭配extern使用，将类中的一些方法定在在类的外部;从功能实现的角度来看，类内和类外的方法没有明显区别。

**Q80.如何将类中的方法定义在类外**

例如：

**Q81.modport的用途是什么**

modport应用于interface中，用于将接口信号分类，其相当于一个小型的接口模块，以方便相同类型的信号接口在进行连接时使用，可指定信号的输入输出类型。

**Q82.struct和union的异同**

union （ 共用体）：构造数据类型,也叫联合体

用途：使几个不同类型的变量共占一段内存(相互覆盖)

struct ( 结构体 )：是一种构造类型

用途： 把不同的数据组合成一个整体——自定义数据类型

**主要区别：**

1. struct和union都是由多个不同的数据类型成员组成, 但在任何同一时刻, union中只存放了一个被选中的成员; 而struct的所有成员都存在。在struct中，各成员都占有自己的内存空间，它们是同时存在的,一个struct变量的总长度等于所有成员长度之和，遵从字节对其原则; 在Union中，所有成员不能同时占用它的内存空间，它们不能同时存在 , Union变量的长度等于最长的成员的长度。
2. 对于union的不同成员赋值, 将会对其它成员重写, 原来成员的值就不存在了,所以，共同体变量中起作用的成员是最后一次存放的成员; 而对于struct的不同成员赋值是互不影响的。

**Q83.$rose和posedge区别**

$rose和posedge都是用来捕捉边沿，但定义有所不同：

**posedge**

Define ： 数值由0->1的时刻

**$rose**

Define ：在仿真中$rose并不是单纯的判断信号的跳边沿，而是判断时钟采样信号前后是否存 在0->1/x->1/z->1的变化。

**Q84.如何在fork...join结构中kill进程**

disable fork会kill disable fork 所在的当前线程以及所有子线程；

**Q85.什么是覆盖率驱动的验证**

覆盖率驱动（CDV）验证是指通过覆盖率的收集去衡量验证进度，换句话说，就是我们验证了半天，究竟验证到什么地步，通过验证率来衡量。首先要保证验证的完备性，要求我们能识别的点都要验证到，需要说明，并不是验证率达到要求，就可以说验证ok了，只能说我们的风险变小了，不能说一点风险没有。

**Q86.如何检查句柄是否指向有效对象**

句柄相当于时指针，也就是地址指向，判断其是否指向有效对象可以理解为判断该地址索引下的空间是否有值，有值即认为是有效的。

**Q87.semaphore用处是什么**

当多个线程访问同一资源的时候，而这个资源只允许一个线程访问的时候，就可以用旗语来进行控制；如果把资源比作为一个仓库的话，旗语就相当于一把钥匙，每个线程需要拿到钥匙后才能对资源进行访问。当然也可以有多把钥匙，对应同一资源可以同时被访问的最大数量。

旗语有如下几个操作函数：

new(); //对“钥匙”进行实例化

get(); //线程获取钥匙，阻塞

put(); //线程将钥匙放回，阻塞

try\_get(); //非阻塞

**Q88.为什么要使用断言**

1.使用断言可以缩短研制周期；

2.使用断言可以使设计中存在的各种问题更容易被动态监测观察；

3.使用断言内嵌的覆盖率统计功能（cover）可以更加容易的获得对于功能的覆盖性；

4.断言的可读性较一般描述语言更容易理解；

5.通过全局控制实现设计中断言的开关；

6.断言可以加速形式验证，提高形式验证的效率；

**Q89.如何在clocking block中声明异步信号**

**Q90.代码覆盖率和功能覆盖率的关系**

代码覆盖率和功能覆盖率，二者无必然的联系。而为了保证验证的完备性，在收集覆盖率时，要求代码覆盖率和功能覆盖率同时达到要求。

**如果你收集的代码覆盖率很高但是功能覆盖率却很低，这个可能是：**

design没有完全按照功能spec文档规格实现所有功能，设计不完善；

验证平台monitor监视器存在漏洞，设计代码实现了功能但没有覆盖到他们；

功能正确，但是发送的激励不对，对应功能的覆盖率无法收集。

**如果你收集的功能覆盖率很高但是代码覆盖率却很低，这可能是：**

设计代码没有按照spec实现功能，在design中是无效代码，

用户定义的功能覆盖率收集存在错误，没有完全定比覆盖率收集的边界，测试计划未捕获所有设计功能/场景/边界，或者缺少功能覆盖率监视器。

代码覆盖率中未覆盖的设计代码可能会映射到这些功能上。

在实现功能覆盖率监视器时可能存在潜在的错误，导致它们收集了错误的覆盖率。因此，在验证项目中，对用户定义的功能覆盖率指标及其实现进行适当的检查很重要。

**Q91.什么是验证计划，应该包含哪些部分**

**一般的验证计划会从技术和项目角度去制定：**

从技术角度而言，我们需要考虑的有验证的功能点，验证的层次，测试用例，验证方法和覆盖率要求；

从项目部分来看，我们也需要考虑使用的工具，人力安排，进度安排和风险评估。

**Q92.类中的静态方法使用注意事项有哪些**

* 静态方法只能访问类的静态属性，而访问非静态属性是非法的，会导致编译错误。
* 静态方法不能是虚拟的（virtual）

**Q93.initial和final的区别**

initial过程语句在仿真开始的时候启用，一个initial过程语句只执行一次，当语句执行完后，它的活动就停止了。

final过程在仿真结束时启用，并且只执行一次，final过程类似于initial过程，它定义了一个语句过程块，不同的是它发生在仿真时间的末尾，并且执行时没有延迟。finial通常用于显示有关仿真的统计信息。

**Q94.建模存储器，使用什么类型的数组**

存储器属于寄存器数组类型，一般利用二位数组进行建模。

**Q95.如何避免测试平台和dut之间的竞争冒险**

* Module中的initial语句块调度在active region， Program中的initial语句块调度在reactive region。
* 在program中使用非阻塞赋值来驱动设计信号，在re-NBA区域对其进行更新。
* 通过带有 #0 input skews的clocking blocks。

**Q96.logic、bit、wire区别**

Wire：用于连接不同的元件，不存储值，被连续赋值（assign）或端口（port）驱动。

Reg：并不意味实际的存储器，代表verilog/sv 中的数据存储元素，保存值，直到被下一次赋值（阻塞或非阻塞），可被综合为触发器，锁存器或组合电路。过程赋值

Logic：不能被多驱动，改进reg数据类型，可以被连续赋值, 过程赋值

**Q97.什么是抽象类**

抽象类又称为虚类，如virtual class bus;

抽象类可以被继承，拥有类的封装功能但是不可以被直接例化使用。

**Q98.always@\*与always\_comb区别**

always是循环执行语句，与原始的always块不一样的是，三个新的always块是专门针对可综合性RTL逻辑建模而定义的，而原始always块则是万金油。事实上，always\_ff, always\_comb, always\_latch能实现的，always都能够实现。

[always\_comb]

always\_comb用于可综合组合逻辑的建模，不需要明确地定义敏感列表。缺省地，所有出现在always\_comb中所有input信号（从always\_comb的外部获得赋值，出现在块内赋值语句的右边，或者条件判断语句中出现的信号，等等）都被推断属于它的敏感列表。但是内部定义的用阻塞方式进行赋值的临时变量不属于敏感列表。

always\_comb与always @(\*)在表达组合逻辑时几乎是等价的，但是仍然存在细微的区别。

[always\_ff]

always\_ff用于可综合时序逻辑的建模。

必须带由posedge或者negedge所定义的敏感列表。通常就是”@(posedge clock, negedge resetN”.

在always\_ff块中只能使用非阻塞赋值。

[always\_latch]

由于在一般的同步设计中是不允许（或至少不推荐）使用latch式设计，所以这里不作说明。在一般的设计中不用它就是了。

**Q99.简述验证结构**

1.测试平台(testbench)是整个验证系统的总称，它包括验证结构中的各个组件、组件之间的连接关系、测试平台的配置和控制。从更系统的意义来讲，它还包括编译仿真的流程、结果分析报告和覆盖率检查等。

2.从狭义上讲，我们主要关注验证平台的结构和组件部分，他们可以产生设计所需要的各种输入，也会在此基础上进行设计功能的检查。

**Q100.parameter、define和typedef之间区别**

define：作用 -> 常用于定义常量可以跨模块、跨文件;

范围 -> 整个工程;

parameter： 作用 -> 常用于模块间参数传递;

范围 -> 本module内有效的定义;

typedef：作用-> 用来创建新的数据类型;

范围 -> 本module内有效的定义;

**Q101.new()和new[]的区别**

**new( ):**

1， 在使用new( )函数创建对象过程中，除了会为类中各个对象分配内存之外，还会将变量设置为默认数值;

2， new函数由于是构造函数，其本身不能有返回值，而是返回一个指向类对象的句柄，其类型就是类本身。

**new[ ]:**

一般new [ ]也是用来开辟内存并初始化，但是它主要用于设置动态数组的大小

**与new( )不同点：**

1， 调用new函数是用于将对象实例化，而new[ ]则是创建了一个含有多个元素的数组;

2， new( )函数可以利用参数来改变对象中变量的值，而new[ ]的参数仅仅被用作设置动态数组的大小。

**Q102.solve...before如何使用**

1.例子

rand bit a；

rand bit b；

constraint c1 {

a != b ;

solve a before b;

}

2.结果

假如 a随机成了0，那么b就只能是1了。

假如 a随机成了1，那么b就只能是0了。

3.使用solve和before的条件

不能使用randc。

随机值只能是整数。

**Q103.mailbox和队列的异同**

mailbox需要例化，而队列不用。

mailbox中可以同时储存不同数据类型的数据（不建议），而队列不行。

队列中push\_back和pop\_front()为非阻塞，再取数时要保证队列大小大于0 （queue.size() > 0），而mailbox既有阻塞又又非阻塞。

mailbox传递形式参数时，传递拷贝的是它的指针，而对于queue若不声明为ref(默认为input)，那么传递数组时会对数组进行拷贝而不是影响原来的数组。

**Q104.什么是静态变量**

静态变量就是被static修饰的变量 （如 static int a）

**Q105.什么是生命周期**

在SV中，我们将数据的生命周期分为**动态（automatic）和静态（static）**

局部变量是动态生命周期，其生命周期同所在域共存亡，例如function/task中的临时变量，调用结束后，临时变量的生命也将终结。

全局变量是静态生命周期，从程序执行开始到结束一直存在，例如module中的变量默认都是全局变量。

如果数据变量被声明为automatic，那么在进入该进程/方法后，automatic变量会被创建，离开后会被销毁。

而static变量从仿真开始时就被创建，在进程/方法执行过程中，自身不会别销毁，还会被多个进程/方法共享

**Q106.交叉覆盖率的优点**

交叉覆盖率：可以同时测量两个或两个以上的覆盖点的值，如果一种由N种取值，另一个有M种取值，需要N X M交叉仓来存储所有的组合。

使用交叉覆盖率在搭配使用ignore bins/illegal bins等声明可以将覆盖率定义到更需要的点上。

**Q107.pass\_by\_value和pass\_by\_ref区别**

pass\_by\_value 是将参数传递给function或task的默认方法，每个子例程保留该参数的本地副本，如果在子例程中更改了参数，不会影响原来的值.

pass\_by\_ref function和task直接访问作为参数传递的指针变量，传递的是指针，指向同一对象，如果不想更改数组值，可以使用const ref。

**Q108.$display和$write区别**

二者的区别在于，前者会自动换行，而后者不会;

**Q109.同一个作用范围内使用枚举类型需要注意什么**

枚举类型支持first，last，next，prev操作

枚举类型默认的值为int，若未在声明中指定数值，则第一个值为0，第二个值为1，依次递增；若在声明中需要指定数值，则必须满足所有的数值必须唯一这一条件，对于没有指定的元素，其数值是按照前一个元素的数值加1

枚举类型取值范围不能超多基础类型的有效范围！！！给四值逻辑变量赋值X/Z是合法的，但是必须给其之后的变量显式赋值，一般在设计中很少赋值为x/z

**Q110.敏感信号列表信号缺失会如何**

不完整的信号列表会造成不同的仿真和综合结果，因此需要保证敏感信号的完备性;

当敏感信号不完备时，会使得仿真结果不一样，这是因为仿真器在工作时不会自动补充敏感信号表。

如果缺少信号，则无法触发和该信号相关的仿真进程，也就得不到正确的仿真结果。

**Q111.covergroup在类中使用和类外分别如何使用**

**Q112.简述回调机制**

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，只内部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

**Q113.三段式状态机是哪三段（状态转移、组合逻辑描述状态转移规律、电路输出）**

第一个always语句实现同步状态跳转；

第二个always语句采用组合逻辑判断状态转移条件；

第三个always语句描述状态输出(可以用组合电路输出，也可以时序电路输出)。

**Q114.什么是虚接口，为什么要使用虚接口**

virtual interface的本质是指向interface的指针，因此其并不是一个真实存在的实体，而interface是一个真实存在的实体；基于OOP的测试平台会通过调用virtual interface来间接操作实际的信号。

通过引入interface可以简化模块儿之间的连接，即interface是连接硬件的，其是硬件语言；但对于验证来说，其描述语言往往是软件语言，interface无法在基于OOP的测试平台中实例化，因此我们无法通过interface把激励传送到DUT中；为了解决这个问题，引入了virtual interface，使得基于OOP的验证环境可以通过虚接口把激励传送给DUT。

**Q115.Verilog中for能不能综合**

在硬件电路Verilog中for语句一般不在可综合代码中使用，因为for循环会被综合器展开为所有变量情况的执行语句，每个变量独立占用寄存器资源，每条执行语句并不能有效地复用硬件逻辑资源，造成巨大的资源浪费。简单的说就是：for语句循环几次，就是将相同的电路复制几次，因此循环次数越多，占用面积越大，综合就越慢。

**Q118.SystemVerilog中##n表示什么**

##表示**周期延迟**符号，如##n表示在n个时钟周期后，**##0表示在当前周期**，即交叠周期。

##[min:max]表示在一个范围内的时钟周期延迟。min和max必须是**非负数**，序列会在从min到max时间窗口中最早的时间来匹配。

**Q119.UVM指的是什么？具有哪些特点，为什么要使用UVM？**

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。但是UVM又不仅仅是封装，其还有一些内建的基类，方法和函数供验证人员去使用，极大的方便了验证人员搭建验证平台.

**常见的UVM机制：**

* field\_automation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。

* config\_db机制

config\_db机制在UVM验证平台中主要用于参数传递。它们通常都是成对出现的，set函数是寄信，而get函数是收信。函数格式如下：

uvm\_config\_db# (Type)::set(this,"inst\_path","field",value); //将value寄送给 “field(字段)”

uvm\_config\_db# (Type)::get(this,"","field",field) //field将接收参数的传递

结果得到：field = value；

所谓“字段”，是变量名包含的内容。在set和get函数中为第三个参数，字段在set和get中必须保持一致，才能保证参数的正确传递。例如要将参数值10，传递给int类型的变量data,则：

uvm\_config\_db# (int)::set(this,"env.i\_agt.drv","dat",10); //将10寄送给env组件中的成员变量“data”

uvm\_config\_db# (int)::get(this,"","dat",data) //字段“dat”保持一致，成员变量data将接收参数的传递

结果得到：data = 10;

* objection机制

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](javascript:void(0);)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。

* factory机制

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。

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。

* callback机制

在UVM验证平台中，callback机制最大的用处就是提高验证平台的可重用性。很多情况下，验证人员期望在一个项目中开发的验证平台能够用于另外一个项目。但是，通常来说，完全的重用是比较难实现的，两个不同的项目之间或多或少会有一些差异。如果把两个项目不同的地方使用callback函数来做，而把相同的地方写成一个完整的env，这样重用时，只要改变相关的callback函数，env可完全的重用。

除了提高可重用性外，callback机制还用于构建异常的测试用例，VMM用户会非常熟悉这一点。只是在UVM中，构建异常的测试用例有很多种方式，如factory机制重载，而callback机制只是其中的一种。

* phase机制

在不同的时间做不同的事情，这就是UVM中phase的设计哲学。但是仅仅划分成phase是不够的，phase的自动执行功能才极大方便了用户。当new语句执行完成后，后边的connect语句肯定就会自动执行。现引入phase概念，将前面的new的部分包裹进build\_phase里面，把后边的connect语句包裹进connect\_phase里边，很自然的，当build\_phase执行结束后就应该自动执行connect\_phase。

phase的引入在很大程度上解决了因代码顺序杂乱可能会引起的问题。遵循UVM的代码顺序划分原则（如build\_phase做例化工作，connect\_phase做连接工作等），可以在很大程度上减少验证平台开发者的工作量，使其从一段杂乱的工作中解脱出来。

UVM中的phase按照其是否消耗仿真时间（$time打印出的时间）的特性，可以分成两大类：一类是function\_phase，如产生build\_phase、connect\_phase等，这些phase都不消耗仿真时间，通过函数来实现；另一类是task\_phase，如run\_phase等，他们消耗仿真时间，通过任务来实现。给DUT施加激励、监测DUT的输出都是在这些phase中完成的。

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

* sequence机制

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。

* 通信机制TLM

UVM中，通常使用TLM（Transaction Level Modeling）实现component之间transaction级别的通信。TLM现在已经发展到TLM2.0，是对TLM1.0的扩展。但是TLM1.0足以满足大多数完整的验证平台的通信要求，所以主要学习TLM1.0。

**Q120.简介工厂机制（factory）**

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。

**Q121.简介事务级建模**

TLM通信步骤：

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

2.在target中实现TLM通信方法 ,例如，在comp2中由于有一个uvm\_nonblocking\_put\_imp #(request, comp2)nbp\_imp，因此需要实现两个方法try\_put()和can\_put();而comp4中有一个uvm blocking get\_imp #(request, comp bg\_imp，则需要实现对应的方法get()。

3.在两个对象中创建TLM端口。

4.在更高层次中通过connect()函数将两个对象的端口进行连接。

从数据流向来看，传输方向可以分为单向(unidirection）和双向( bidirection)

1.单向传输:由initiator发起request transaction。

2.双向传输:由initiator发起request transaction，传送至target;而target在消化了request transaction后，会发起response transaction，继而返回给initiator。

端口的按照类型可以划分为三种:

1.port:经常作为initiator的发起端，initiator凭借port才可以访问target的TLM通信方法。

2.export:作为initiator和target中间层次的端口。

3.imp:只能作为target接收request的末端，它无法作为中间层次的端口，所以imp的连接无法再次延伸。

端口的使用：

1.就单向端口而言，声明port和export作为request发起方，需要指定transaction类型参数，而声明imp作为request接收方，不但需要指定transaction类型，也需要指定它所在的component类型。

2.就声明双向端口而言，指定参数需要考虑双向传输的因素，将传输类型transaction拆分为request transaction类型和response transaction类型。

3.从对应连接关系得出TLM端口连接的一般做法:

* 在initiator端例化port，在中间层次例化export，在target端例化imp。
* 多个port可以连接到同一个export或者imp;但是单个port或者export无法连接多个imp。这可以抽象为多个initiator可以对同一个target发起request，但是同一个initiator无法连接多个target。
* port应为request起点，imp应为request终点，而中间可以穿越多个层次。基于单元组件的自闭性考虑，建议在穿越的中间层次声明export，继而通过层层连接实现数据通路。
* port可以连接port、export或者imp; export可以连接export或者imp; imp只能作为数据传送的终点,无法扩展连接。

Peek和get的流向一样

1.阻塞传输方式将blocking前缀作为函数名的一部分，而非阻塞方式则名为nonblocking。阻塞端口的方法类型为task，这保证了可以实现事件等待和延时;非阻塞端口的方式类型为function，这确保了方法调用可以立即返回。

2.我们从方法名也可以发现，例如uvm\_blocking\_put\_PORT提供的方法task put()会在数据传送完后返回，uvm\_nonblocking\_put\_PORT对应的两个函数try\_put()和can\_put()是立刻返回的。

uvm\_put\_PORT则分别提供了blocking和nonblocking的方法，这为通讯方式提供了更多选择。blocking阻塞传输的方法包含:

Put（）：initiator先生成数据Tt，同时将该数据传送至target。

Get（）:initiator从target获取数据Tt，而target中的该数据Tt则应消耗。

Peek(): initiator从target获取数据Tt，而target中的该数据Tt还应保留。

这六个非阻塞函数与对应阻塞任务的区别在于，它们必须立即返回，如果try\_xxx函数可以发送或者获取数据，那么函数应该返回1，如果执行失败则应返回0。或者通过can\_xxx函数先试探target是否可以接收数据，如果可以，再通过try\_xxx函数发送，提高数据发送的成功率。

**Q122.uvm\_component和uvm\_object的区别**

* UVM验证平台所有的组件都要派生自UVM的类。（ uvm\_component和uvm\_object以及他俩的“孩子”）。uvm\_component继承与uvm\_object。
* SV中“固定资产”（test、environment、agent等）派生自uvm\_component。SV中的“非固定资产”（比如数据包transaction等的产生）派生自uvm\_object。
* 只有uvm\_component派生的类，才有节点（每个结点是一个class实例）。
* uvm\_component和uvm\_object的主要区别：uvm\_object是有生命周期的；uvm\_component没有生命周期。也就是说仿真过程中，uvm\_object在启动后可以finish；uvm\_component是伴随仿真过程一直要存在的。
* uvm\_component及派生的类是uvm树的骨架，需要标明parent表明继承关系，uvm\_object不用说明parent。每一个派生自uvm\_component或其派生类的类在其new函数中要指明两个参数：name和parent，这是uvm\_component类的一大特征（如果parent设定为null，那它将作为uvm\_top的“孩子”；parent设定为this，那它的parent为当前的类）

**Q123.UVM中run\_phase和main\_phase的区别**

* main\_phase( )是12个分支任务 phase 中的一个，run\_phase 的执行贯穿这12个分支 phase。
* run\_phase( )和 main\_phase( )是并行执行的，只有等12个分支phase全部执行完，才开始执行 extract\_phase 等后面的phase。

run\_phase( ) 和 main\_phase( )的区别：

* 首先，Objection机制是UVM中唯一可以控制仿真开始和结束的方式。在任务 phase 中，至少有一个任务 phase 要在消耗第一条消耗仿真时间的语句执行之前要挂起 Objection。
* 如果12个分支中有一个任务phase（比如main\_phase）挂起了 Objection ，那么 run\_phase 中不需要挂起 Objection 就可以执行其中的代码；但是这时，run\_phase 的运行时间被动地受这个挂起 Objection 的分支任务phase的控制。
* 而如果在 run\_phase 中挂起了 Objection ，没有在main\_phase中挂起，main\_phase中的操作则不会执行。

**Q124.为什么要使用phase机制**

phase机制主要目的就是确保各个组件之间可以有序运行

那么UVM 的各个phase又是如何运行的呢？

1.各个phase按照一定的顺序，从上往下顺序运行，

2.每个组件内部的class 按照phase名称顺序运行。比如driver内有两个class 都有build phase 和main phase 那么，是两个class内的build phase都运行完了以后再运行main phase

3.UVM内部的各个组件中的phase也是按照顺序运行。同理driver 和sequencer都有build phase main phase 那么是当driver 和sequencer内的build phase 都运行完以后再会运行driver 和sequencer的main phase以上即为顺序运行

4.UVM内部的phase会自动运行，但是用户自定义的phase只有调用的时候才会运行。如main phase , main phase11, main phase22。假如没有调用main phase11那么这个phase是不会运行的

**Q125.m\_sequencer和p\_sequencer区别**

equence/item一旦“挂载”到某一个sequencer上，该sequencer的句柄即被赋值于m\_sequencer(uvm\_sequencer\_base类)；而p\_sequencer通常需要通过在定义sequence类使，通过宏`uvm\_declare\_p\_sequencer(SEQUENCER)声明，间接定义p\_sequencer成员变量。m\_sequencer与p\_sequencer均指向同一个挂载的sequencer句柄，只是前者使父类句柄，后者往往是所挂载sequencer句柄（子类句柄）

**Q126.top-down phase、bottom-up phase有哪些**

**Q127.为什么build\_phase是top-down phase，connect\_phase是bottom-up phase**

build\_phase需要验证平台根据高层次组件的配置来决定建立低层次的组件，所以其是top-down phase。connect\_phase需要在build\_phase之后完成验证组件之间TLM连接

**Q128.$size用于packed array和unpacked array分别得到的什么**

得到的是高维度的array size。

**Q129.class和struct的异同**

二者都可以定义数据成员

类变量在声明之后，需要构造函数才构建对象实体，class在声明变量时已经开辟内存

类不仅可以声明数据变量成员，而且可以声明方法，这是struct做不到的

从根本上说，struct是一种数据结构，而class包含数据成员以及针对这些成员的操作方法

**Q130.class和module的异同**

数据和方法定义方面：两者都使用相同的模板来创建实例，都可以作为封闭的容器来存储数据和方法。

例化方面：module例化是静态的，在编译链接时就完成；而SV中class的例化是动态的，可以在仿真的任何阶段声明并动态创建新的对象，这也使类的例化方式更加灵活而节省空间。

封装性方面：模块内的变量和方法对外部都是开放的，而类可以根据需要来确定外部访问的权限：pubilc，protected，local

继承性方面：模块没有继承性而言

**Q131.对象创建的初始化顺序**

（1）给对象分配内存。

（2）将对象的实例变量自动初始化为其变量类型的默认值。

（3）初始化对象，给实例变量赋予正确的初始值。

**Q132.子类和父类中是否可以定义相同名称的成员变量和方法（非虚方法）**

可以；

当父类的句柄执行子类的对象时，调用父类的方法时，默认只输出父类的内容，因为父类未声明虚方法，所以不会进一步调用子类的方法输出重写的部分。

**Q133.为什么需要随机**

验证的目的是通过大量的测试激励去验证DUT功能的完备性，如果只是依靠定向测试，这样的工作量的很大的，而且定向测试可能会遗漏掉某些验证场景，所以我们在验证中一般都会采用随机的方法，随机出感兴趣的验证激励，通过不断更换随机种子，对某一验证场景进行充分验证，以保证验证的完备性。

**Q134.线程间通信控制共享资源的原因是什么**

在uvm中，对线程通信的资源都会放到一个资源池中，该资源池作为一个全局变量可以供验证需要，完成线程间的通信。

**Q135.uvm\_transaction和uvm\_seq\_item的关系**

uvm\_transaction是从uvm\_object派生的用于对事务进行建模的基类。

sequence item是在uvm\_transaction的基础上还添加了一些其他信息的类，例如：sequence id。建议使用uvm\_sequence\_item实现基于sequence的激励。

**Q136.p\_sequencer是什么？**

p\_sequencer 通常需要通过在定义sequence时声明`uvm\_declare\_p\_sequencer(SQR）,间接定义p\_sequencer成员变量，m\_sequencer和p\_sequencer均指向同一挂载的sequencer句柄，只是前者是父类句柄，后者是子类句柄。

**Q137.m\_sequencer是什么？**

m\_sequencer 是sequence或sequence\_item 的成员变量，即sequence/item一旦挂载到某个sequencer上，那么该sequencer的句柄就被赋值为m\_sequencer

（uvm\_sequencer\_base类）。

**Q138.new()和create有什么区别**

UVM推荐使用内置方法type\_name::type\_id::create（），而不是直接调用构造函数new()创建组件或事务对象。create方法在内部调用factory机制以查找所请求创建的类型，然后调用构造函数new（）以实际创建一个对象而无需更改任何代码。

**Q139.如何启动sequence**

启动sequence需要执行三个步骤，如下所示：

创建一个序列。使用工厂创建方法创建一个序列：

my\_sequence\_c seq;

seq = my\_sequence\_c ::type\_id :: create（“ my\_seq”）

配置或随机化序列。

seq.randomize（）

开始一个sequence。使用sequence.start（）方法启动序列。start方法需要输入一个指向sequencer的参数。关于sequence的启动，UVM进一步做了封装。

还可以通过default\_sequence自动启动，uvm\_config\_db#(uvm\_object\_wrapper):: set (this,sequencer.main\_phase,default\_sequence,sequence::type\_id::get())

**Q140.copy和clone的区别**

create（）方法用于构造一个对象。copy（）方法用于将一个对象复制到另一个对象。clone（）方法同时完成对象的创建和复制。

**Q141.Agent中的Active mode和Passive mode区别**

ACTIVE agent是可以在其操作的接口上生成激励，其包含driver和sequencer。PASSIVE agent不生成激励，只能监视接口，这意味着在PASSIVE agent中将不会创建driver和sequencer。

**Q142.在UVM的工厂机制中，为什么要使用注册机制**

只有先将类进行注册，在利用工厂方法创建之后才可以应用工厂机制提供的方法对已经实例化的类以及已经注册的类型进行覆盖。

**Q143.简述UVM的工厂机制**

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。

**Q144.UVM中的RAL什么，可以用来干什么？**

UVM RAL（寄存器抽象层）\*\*是UVM支持的功能，有助于使用抽象寄存器模型来验证设计中的寄存器以及DUT的配置。UVM寄存器模型寄存器抽象模型反映了寄存器设计的结构规范，提供了一种跟踪DUT寄存器内容和位置的方式。这是硬件和软件工程师的共同参考。RAL的其他一些功能包括支持寄存器的front door和backdoor初始化以及内置的功能覆盖率支持。

**Q145.简述系统级、子系统级和模块级验证**

模块验证：侧重点在模块本身功能的验证，验证计划的重点是feature和验证架构，然后列出testcase，模块能够覆盖的绝不到下一级验证去覆盖。主要内容有：检查参数设置、寄存器读写、协议检查、中断和复位、状态机跳转、工作模式覆盖、RAM的读写功能边界等等。

子系统验证：侧重点在系统的互联性，更加关注系统的工作模式和复杂场景应用。主要内容有：中断的产生、DMA功能、IP的模式功能、Memory读写等等。

系统验证：侧重点在软硬件协同仿真，关键系统路径的覆盖，芯片工作模式和测试模式以及数据通路和性能等。主要内容有：基本IP功能、CLK/RESET、IO MUX 、多个IP同时工作、程序的启动、工作模式和应用场景测试。

**Q146.IP和VIP分别指的是什么**

数字 IC 验证领域的 IP 指的是 **Intellectual Property(IP) core**，VIP 指的是**verificati on Intellectual Property(IP) core**

**Q147.set\_config\_\*和uvm\_config\_db区别**

set\_config\_\*可以映射到相应的uvm\_ config\_db：

set\_config\_int(…)=>uvm\_config\_db#(uvm\_bit\_stream\_t) :: set(cntxt, …)

set\_config\_string(…)>uvm\_config\_db#(string) :: set(cntxt,…)

set\_config\_object (…)>uvm\_config\_db#(uvm\_object) :: set(cntxt,…)

**Q148.$stop、$finish和final如何使用**

$stop(), 函数触发，仿真会停止，但是还可以继续restart；

$finish(),函数触发，仿真会停止，不可以继续restart；

final 类似于initial，只执行一次，不同的是final在仿真结束时执行。

**Q149.简述virtual sequence和virtual sequencer作用**

virtual sequence是一个包含和执行多个子sequence的容器，virtual sequencer是包含其他sequencer的容器以使得virtual sequence中的每个子sequence都能在相应的sequencer上获得执行。

**Q150.简述code review的重要性**

* 提高代码质量
* 提前发现bug
* 统一代码规范
* 提高团队成员代码技能