一 UVM 部分

1 UVM 工厂机制

工厂机制的优势就是具有**重载**功能，可以方便我们**替换**验证环境中的**实例**或**已注册的类型**，使得**配置更加灵活**。（替换的过程就是覆盖的过程）

使用工厂机制的过程：

（1）将类**注册**到 factory 表中（`uvm_component/object_utils）

（2）使用 type_id::create **创建对象**（例化组件）

（3）编写相应的类对基类进行**覆盖**（写扩展类/子类）

工厂覆盖（factory 机制重载）的要求：

（1）重载的类与被重载的类在定义时需**注册**到 factory 机制中。

（2）被重载的类在实例化时，要使用 **type_id::create**（工厂机制实例化方式），而不能使用 new()。

（3）重载的类与被重载的类要有派生关系。

（4）component 与 object 不能互相重载。

2 field_automation 机制

当使用 uvm_field 系列相关宏注册 transation 成员变量之后，可以直接调用 uvm 内置方法，如 copy，compare，print，pack 等，而无需自己定义。

3 `uvm_component(object)_utils

作用是将派生自 uvm_component/object 的类注册到工厂(factory)中，这样工厂机制就可以实现：根据字符串自动创建一个类的实例，并且调用其中的函数与任务，然后这个类的 main phase 就会被自动调用（因为 main phase 就是一个任务）。

4 uvm_config_db 作用及各参数含义，与 uvm_resource_db 区别

Config_db 机制主要作用就是**传递参数**使得验证环境的配置更加灵活。Config_db 机制主要传递的有三**种**类型：

(1)**传递 virtual interface 到环境中**，使得 driver 和 monitor 能够与 DUT 连接，并驱动接口和采集接口信号。

(2) **设置单一变量值**，如 int，string，enum 等。

(3) **传递配置对象到环境**。当配置参数较多时，可以将其封装成一个 object 类，这样传递起来比较简单，不易出错。

set 参数:第一个参数是发送 component 实例的指针,第二个参数是相对于发送 component 的目标组件的指针，第三个参数表明此参数是发送给目标组件中哪一个成员变量，第四个参数是要设置的值。

get 参数：第一个参数是接收 component 实例的指针，第二个参数是相对此实例的路径，第三个参数与 set 第三个参数严格保持一致，第四个参数是要设置的变量。

传递参数未成功的原因：可能是 set 与 get 的路径不一样，或者第三个成员变量不一样。

uvm_resource_db：如果高层次和低层次都对同一变量进行写入，那么在 build 阶段，由于采取自顶向下的顺序，低层次的配置写入发生在最后，反而会将低层次作为有效数据进行

写入，因而无法实现层次化的覆盖，这就不利于集成和复用。
区别：uvm_config_db 继承于 uvm_resource_db，uvm_resource_db 是对同一配置，后写入有效，与层次关系无关；而 uvm_config_db 是最高层次的配置有效，同一层次后写入有效。

5 接口的传递（virtual interface）

在底层组件中（如 driver）通过 uvm_config_db get 到来自顶层传递的接口，传递的接口必须用 virtual 声明，即实际接口的句柄，否则传递的是一个实际的物理接口，这在软件环境中是不能实现的，会报错。

6 UVM 仿真的启动与结束

启动两种方法：
（1）在导入 uvm_pkg 文件时，会自动创建 uvm_root 所例化的对象 uvm_top，通过 uvm_top 调用 run_test()方法。在环境中输入 run_test，run_test 语句会创建一个 my_case0 的实例，并会自动调用 my_case0 的 main_phase，然后执行各个组件中的 phase 机制。
（2）在 top 模块中调用 run test()函数，然后在命令行添加：+UVM_TESTNMAE=test1(测试用例名），启动测试用例名为 test1 的测试用例。
结束：在仿真运行的 run_phase 阶段，当所有的 objection 都落下后，便会结束仿真运行，uvm 会调用$finish 将整个验证平台关掉。

7 UVM 的优势与劣势

优势：UVM 是一个框架，可以按照需要自己去搭建，流程规范，在写的过程中不用过多考虑语言实现问题，可以把精力放在场景的设计实现上。
劣势：UVM 是好几年前开发的，对于现在流行的一些十几亿或者百亿门级电路在当时是没有特别优化这样的场景的，所以对于超大型设计在仿真时会比较吃力。另一方面，虽然理论上来说在仿真时硬件的行为应该是并行的，但实际仿真器在电脑里运行时，受制于电脑，他的运行可能是串行的，只是表现出并行的一些特征，所以当门数很多的时候计算就会特别慢。

8 try 与 get 的区别

get 是阻塞调用，直到有可用的 item 为止，并返回 item 的指针
try 是非阻塞调用，如果没有可用的 item，则返回空指针

9 component 与 object 的区别，分别属于 component 与 object 的有哪些

component 继承于 object，但同时拥有 object 所不具备的一些特点：一是通过 new 的时候指定 parent 参数来形成一种**树形的结构**；二是 phase 自动执行的特点。
component: driver, monitor, sequencer, agent, scoreboard, reference model, test, env
object: item, sequence, config, map, field, phase, reg

10 UVM 验证环境组成（验证平台）

Driver：向 sequencer 索要 item，并将 item 的信息驱动到 dut 的端口上。
Monitor：收集 dut 输出信号，并将其交给后续组件，如 scoreboard。
Sequencer：负责将 sequence 发送的 item 传给 driver。
Agent：将 driver，monitor 和 sequencer 封装在一起，agent 的作用是从可重用性角度考虑。
Reference model：用于完成和 dut 相同功能。
Scoreboard：将参考模型和 dut 的输出进行比较，根据结果判断 dut 是否正确工作。
Env：相当于一个特大容器，将所有容器包含进去。

11 sequence、sequencer 和 driver 之间的通信（握手）机制，这样做的好处

如果 driver 没有 item 可用，将调用 get_next_item()来尝试从 sequencer 一侧获取 item。

sequence 在向 sequencer 发送 item 请求之前，item 应该在 sequence 中完成创建并随机化，在获取权限后，item 会经由 sequencer 最终到达 driver，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 通过 item_done()来告知 sequence 已经完成数据传送。

好处：当针对 DUT 的功能点进行不同的测试时，需要产生不同的激励代码，如果将 sequence 放在 driver 里实现，那么每启动一种测试，都需要修改 driver 里 main_phase 的代码，这不利于代码的维护，降低了验证平台的可重用性。而如果将 sequence 从验证环境独立出来，根据不同测试启动不同 sequence，大大简化了流程，同时 sequence 可以嵌套使用，从而可以最大程度的实现不同测试用例之间的重用。

12 启动 sequence 的方法（挂载到 sequencer）

（1）start 显示启动
（2）default_sequence 隐式启动
（3）`uvm_do 系列宏启动
default_sequence 一般在 testcase 里进行设置，当运行 sequencer 的时候会自动检测有没有 default_sequence。有如下两种方式设置：

```
function void testcase0::build_phase(uvm_phase phase);
    super.build_phase(phase);
    uvm_config_db#(uvm_object_wrapper)::set(this,
                                            "env.agent.main_phase",
                                            "default_sequence",
                                            testcase0_sequence::type_id::get());
endfunction

// 或者如下的方式：

function void testcase0::build_phase(uvm_phase phase);
    super.build_phase(phsae);
    my_sequence test0_sequence;

    testcase0_sequence = my_sequence::type_id::create("testcase0_sequence");

    uvm_config_db#(uvm_object_wrapper)::set(this,
                                            "env.agent.main_phase",
                                            "default_sequence",
                                            testcase0_sequence);
endfunction
```

13 sequence 发送 trans 的方式

（1）`uvm_do 系列宏发送
（2）手动调用 start，要先例化
（3）宏`uvm_create 与`uvm_send

14 层次化 sequence 构建方式

底层 sequence 用来组织 item 实例，高层 sequence（hierarchical sequence）通过层层嵌套，容纳更多的底层 sequence，通过对这些底层 sequence 进行协调和调度，最终完成一个期望的测试场景。virtual sequence 作为顶层的 sequence 面向的是多个 sequencer 的 sequence 群，本身并不需要产生 item，只需要将 sequence 挂载到对应的 sequencer 上，起到组织协调的作用，一般只会挂载到 virtual sequencer 上。

hierarchical sequence 与 virtual sequence 的区别：hierarchical sequence 面向的是同一个 sequencer，其本身也要挂载到 sequencer，而 virtual sequence 内部的 sequence 面向的是不同的 sequencer，其本身不用挂载到 sequencer。

15 sequence 与 item 的关系

一个 sequence 包含一些有序组织起来的 item 实例，item 是基于 uvm_object 类，表明了它具备 **UVM 核心基类**所必要的数据操作方法，如 copy，compare 等。item 的生命周期起始于 sequence 的 body()方法，在经历创建与随机化后，经由 sequencer 最终到达 driver，直到被 driver 消化后，生命周期才结束。

16 sequence 机制及其优势

sequence 机制包括 sequence 的启动，item 的发送，仲裁机制与锁存机制，嵌套，virtual sequence 等。

优势：UVM 的 sequence 机制最大的作用就是将 sequence 和验证平台分离开来。对一个项目而言，验证平台是相对稳定的框架，而针对各个部分要有不同的测试内容，如果将 sequence 放在 driver 里实现，那么每启动一种测试，都需要修改 driver 里 main_phase

的代码，这不利于代码的维护，降低了验证平台的可重用性。而如果将 sequence 从验证环境独立出来，根据不同测试启动不同 sequence，大大简化了流程，同时 sequence 可以嵌套使用，从而可以最大程度的实现不同测试用例之间的重用。

17 在 sequence 中使用 config_db 机制

在 sequence 中可以使用 config_db 来获取参数，主要问题是路径问题。

在 sequence 中调用 get_full_name()函数得到的是此 sequence 挂载的 squencer 以及实例化 sequence 时所传递的名字。

```
uvm_test_top.env.i_agt.sqr.case0_sequence
```

Set: 第一个参数是 this，第二个参数是此 sequence 挂载的 sequencer 路径以及通配符，因为 sequence 在实例化时名字不是固定的，第三个参数是目标 sequence 中的成员变量，第四个参数是为该成员变量设置的值。

Get: 第一个参数是 null 或者 uvm_root::get()（不能用 this，因为 sequence 不是一个 component），第二个参数是 get_full_name()，这样就可以得到此 sequence 的路径，第三个参数与 set 保持一致，第四个参数是成员变量名。

18 m_sequencer 与 p_sequencer

m_sequencer 是 sequence 的成员变量，即 sequence 一旦挂载到某一个 sequencer 上，那么该 sequencer 的句柄即被赋值于 m_sequencer(uvm_sequencer_base 类，sequencer 是 m_sequener 的子类，赋值的过程相当于向上转换)；但通过 m_sequencer 不能直接使用 sqr 里的变量，否则会出现编译错误。只能使用 cast 强制向子类转换后，才能通过 m_sequencer.xxx 来访问该 sqr 内的变量。UVM 引入 p_sequencer，通过在环境中定义了`uvm_declare_p_sequencer 宏，可以自动的实现上面所述的 cast 动作，从而可以在 sequence 中自由使用 sequencer 内的成员变量。

19 virtual_sequence 与 virtual_sequencer

virtual_sequence 承载了不同 sequencer 的 sequence 群落，它根本就不发送 transaction，只是控制其他的 sequence，起统一调度作用。

virtual_sequencer 与 sequencer 最大的区别是其本身不会传递 item，也不需要和任何 driver 相连，它只是桥接着所有底层 sequencer 句柄，是一个中心化的路由器。

20 sequencer 的仲裁特性与锁存机制

仲裁特性：

用来保证多个 sequence 同时挂载到 squencer 时，可以按照仲裁规则允许特定 sequence 中的 item 优先通过。仲裁算法有如下五种：

seq_arb_fifo:遵循 fifo 先入先出的顺序

seq_arb_strict_fifo:按照优先级，当优先级相同时按照 fifo 先入先出

seq_arb_random:完全随机

seq_arb_weighted:按照 sequence 权重随机授权

seq_arb_strict_random:按照优先级，优先级相同随机选择

锁存机制：

lock：sequence 向 sequencer 发送一个 lock 请求，该请求与其他 sequence 发送 transaction 的请求一同被放入 sequencer 的仲裁队列中，当前面的所有请求处理完毕后，sequencer 就开始响应这个 lock 请求，此后 sequencer 会一直发送此 sequence 的 transaction，直到 unlock 被调用。
grab：grab 操作优先级比 lock 高。lock 请求是插入 sequencer 仲裁队列的最后面，等到它时，它前面的仲裁请求都已经结束了。grab 请求则被放入 sequencer 仲裁队列的最前面，一发出就拥有了 sequencer 的所有权。

21 组件之间的 TLM 通信机制（方法与端口）

（1）端口类别
port：通信请求的**发起端**
export：作为 port 与 imp 的**中间层次**的端口
imp：只能作为接收请求的**响应端**
analysis port：主要用于一对多传输，它在组件中是以广播的形式向外发送数据，可以不连接或者连接一个或多个 analysis imp，没有阻塞非阻塞之分。在 analysis_imp 所在的 component，必须定义一个 write 函数。
uvm_tlm_analysis_fifo：fifo 的本质是一块缓存加上两个 imp，提供了 uvm_analysis_imp 类型的端口与 write()函数，实现了多个对应方法供用户使用。（默认深度是 1）
使用 fifo 的好处：a、无需在 scoreboard 中再写 write 函数；b、可以解决当 reference model 和 monitor 同时连接到 scoreboard 应如何处理的问题。

端口优先级：port > export > imp， 使用 connect()建立连接关系时，只有优先级高的才能调用 connect()做连接，即 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。三种端口不是 uvm_component 的子类，应该使用 new()函数在 build_phase 中创建，而不能用 create 创建。如 driver 与 sequencer 的端口连接：

```
driver.seq_item_port.connect(sequencer.seq_item_export)
```

（2）传输方向
单向传输：数据由生产端发送到接收端。
双向传输：生产端发送数据给接收端，接收端消化数据后会返回一个 response，数据的流向是双向的。
（3）操作类型
put：A 将 transaction 发送给 B。
get：A 向 B 索取一个 transaction。
transport：A 发送数据给 B，B 消化数据后会返回一个 response，数据的流向是双向的。
peek：A 向 B 索取一个 transaction，B 会将数据复制一份发送给 A，B 内部的数据不会减少。（这里的 B 指 fifo）
（4）通信步骤
a、在组件 A 和组件 B 中分别**例化端口**；
b、需要在组件 **B 中实现**两个端口对应的方法。
c、最后在环境中对组件 A，B 的**端口进行连接**，这使得组件 A 的 run_phase 中可以通过自身的端口调用组件 B 中的方法。
注：除了 TLM 通信，config_db 也是一种通信方式

22 objection 机制

通过 objection 机制可以控制验证平台的关闭。在进入某一 phase 时，uvm 会收集此 phase 提出的所有 objection，并且实时监测所有 objection 是否已经撤销，若都撤销了，那么就关闭此 phase，当所有 phase 都执行完毕，就会调用$finish 来将整个的验证平台关闭。如果 uvm 发现此 phase 没有提起任何 objection，那么就会直接跳到下一个 phase 中。

23 phase 机制

phase 机制控制着仿真按着一定的**顺序**进行，使得仿真阶段**层次化**。phase 机制主要有 9 个外加 12 个动态运行的 phase。

（1）分类：根据是否消耗时间分为 function_phase 和 task_phase，主要 phase 有 build_phase，connect_phase，run_phase，main_phase 等。

（2）执行顺序：从时间上看，9 个 phase 顺序执行，run_phase 与 12 个小 phase 并行运行。其中 build_phase 自上而下执行，其余 function_phase 都是自下而上执行。兄弟关系的 component 的相同 phase 之间是按照字典顺序来执行的。**原因：低层次的组件要在高层次组件的 build_phase 里例化，如果在 agent 的 build_phase 之前执行 driver 的 build_phase，此时 driver 还没例化，所以调用 driver.build_phase 就会报错。**

（3）main_phase:main phase 里执行一些耗费时间的语句，它主要实现了各个组件的功能，例如 driver 的 main phase 里主要是按照时序向 DUT 驱动数据，scoreboard 的 main phase 主要是负责比较数据是否一致。

（4）phase 之间的跳转可以使用 phase.jump()实现

24 run_phase 与 main_phase 的关系

run_phase 和 main phase 都 task phase，且是并行运行的，后者称为动态运行的 phase。

run_phase( ) 和 main_phase( )的区别:

如果 12 个小 phase 有一个 phase(比如 main_phase)提起了 objection ，那么 run_phase 中不需要挂起 objection 就可以执行其中的代码；但是，run_phase 的运行时间被动地受这个提起 objection 的小 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。

25 domain 概念

domain 是用来组织不同组件，实现**独立运行**的概念。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 driver 类，默认情况下，两个 driver 类中的 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 driver 就是独立运行的了。(**只针对 12 个小 phase 有效**)。

26 callback 机制

（1）callback 机制的作用

a、callback 机制最大用处就是为了提高验证平台的可重用性；

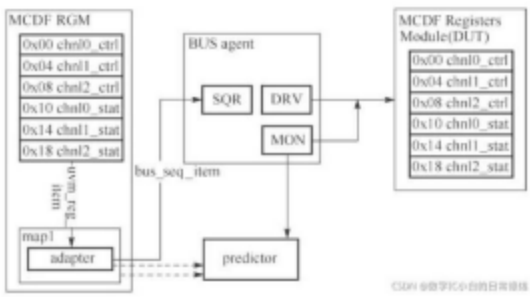
b、在不创建复杂的层次结构前提下，针对组件中的某些行为，在其之前或之后，内置一些函数，增加或修改 UVM 组件的操作，从而实现一个环境多个用例；
c、通过 callback 机制构建异常的测试用例；
（2）回调函数 callback 的使用步骤：
a、在 UVM 组件中**内嵌** callback 函数或任务；
b、声明一个 UVM callback **空壳类**；
c、对空壳类进行**扩展**；
d、在验证环境中**创建并登记** UVM callback 实例。
（3）factory 机制与回调机制区别
callback 的类还是原先的类，只是内部的 callback 函数变了，而 factory 是产生一个新的扩展类进行替换。

27 前门访问与后门访问

（1）前门访问需要通过配置寄存器总线来对 DUT 进行操作，而后门访问可以不通过总线而直接对 DUT 内部寄存器进行读写的操作。
（2）后门访问最大的优势是不消耗仿真时间，而前门访问需要消耗时间。
（3）对于只读寄存器，无法通过前门访问对其进行写操作，但是后门访问却可以。
（4）所有的前门操作都可以在波形文件中找到所有操作的记录，但后门访问却无法找到，只能依靠进行后门访问操作时输出的打印信息，这样就增加了调试的难度。

28 自动预测与显示预测

自动预测
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，这种方式被称之为自动预测。
缺点:如果出现了一些 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，都无法自动得到寄存器的镜像值和期望值。
显式预测
在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。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。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，再利用 adapter 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。



29 若寄存器地址不匹配，怎么测试出来

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，即便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。 上述的存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。

30 寄存器模型的集成

寄存器模型的集成是在 base_test 层实现。需要在 base_test 里先定义 reg model 和 adapter，在 base_test 里的 build_phase 实例化所有用到的类，然后依次调用寄存器模型 rm 的 configure()，build(例化所有寄存器)，lock_mode(调用此函数寄存器模型中就不能再加入新的寄存器了)，reset(将所有寄存器的值都设为复位值)，这样就完成了寄存器模型集成到 test 中。

31 期望值、镜像值

镜像值：对于任意一个寄存器，寄存器模型中都会有一个专门的变量用于最大可能地与 DUT 保持同步，这个变量在寄存器模型中称为 DUT 的镜像值。（镜像值由预测模型给出）
期望值：期望向寄存器中写入的值，然后利用该值更新镜像值。
将镜像值更新为期望值的方法：一是调用 write 任务，直接将期望值写入 DUT 中，然后更新镜像值与期望值;二是通过 set 函数将期望值设为某个值,然后调用 update 任务,update 任务会检查期望值和镜像值是否相同，如果不一致，那么会将期望值写入 DUT 中，并且更新镜像值。

32 寄存器模型读写操作常用函数（方法）

有 update()，mirror()，write()，read()，peek()，poke()
update()：将模型中的期望值更新到 DUT 中。update 会检查期望值和镜像值，如果两者不相等，那么将期望值更新到 DUT 中，并且更新镜像值。update 与 mirror 操作相反。
mirror()：读取 DUT 中寄存器的值，检查读取的值与镜像值是否一样，如果不一样报错。再调用 predict 函数，更新镜像值。
write()：通过前门或者后门方式向 DUT 中写入寄存器值，会产生总线 transaction。并且调用 predict 更新镜像值。
read()：通过前门或者后门方式读取 DUT 中寄存器的值，会产生总线 transaction。并且调用 predict 更新镜像值。
peek()：通过后门访问方式读取寄存器的值，不关心 DUT 的行为，即使寄存器的读写类型是不能读，也可以将值读出来。
poke()：通过后门访问方式写入寄存器的值，不关心 DUT 的行为，即使寄存器的读写类型是不能写，也可以将值写进去。

33 寄存器模型的作用（使用寄存器模型的原因）

（1）寄存器是模块交互的窗口，我们可以通过寄存器值去观察 DUT 的运行状态或改变 DUT 的功能，因此验证寄存器的读写正确非常重要，此外验证环境中的参考模型也要获取 DUT 里寄存器的值，这都需要用到寄存器模型。
（2）没有寄存器模型的话，需要通过`uvm_do 发送 sequence 的方式来访问。而实际 DUT 的寄存器可能达到了成百上千个，如果通过这种一笔一笔发 sequence 的方式，显得太繁琐，代码变得冗余，且这种方式会消耗大量仿真时间。通过引入寄存器模型可以简化寄存器的读写方式，方式有前门和后门访问，使用方法有 read、wrtie、peek、poke、update 等，而且可以选择后门访问从而不消耗仿真时间。
（3）寄存器模型本身提供了寄存器读写测试的 sequence，方便用户直接使用。

34 adapter 的集成

当进行读或写，寄存器模型会通过 sequence 产生一个 uvm_reg_bus_op 的变量，此变量中存储着操作类型（读还是写）和操作的地址，如果是写操作，还会有要写入的数据。此变量中的信息要经过一个转换器（adapter）转换后交给 bus_sequencer，随后交给 bus_driver 实现最终的前门访问读写操作。因此，必须要定义一个 adapter。
在 adapter 中需要定义两个函数：
reg2bus：将 uvm_reg_bus_op 型的变量转换成 bus_sequencer 能够接受的形式。
bus2reg：当监测到总线上有操作时，将收集来的 transaction 转换成寄存器模型能够接受的形式。

35 sequence 的分类

扁平类(flat sequence):用来组织 item 实例。
层次类(hierarchical sequence):由更高层的 sequence 用来嵌套底层的 sequence，进而让这些 sequence 按照顺序或并行方式，挂载到同一个 sequencer 上。
虚拟类(virtual sequence):由于环境中存在不同种类的 sequencer 和其对应的 sequence,需要一个虚拟的 sequence 来协调顶层的测试场景,virtual sequence 承载不同目标 sequencer 的 sequence 群落。

36 uvm_reg_map

每个寄存器在加入寄存器模型时都有其地址，map 就是储存这些地址，并将其转换成可以访问的物理地址。当寄存器模型使用前门访问来实现读或写的操作时，map 就会将地址转换成绝对地址，启动一个读或写的 sequence，并将读或写的结果返回。每个 reg_block 内至少有一个 map。
由于寄存器模型的前门访问操作最终都将由 map 完成，因此在 base_test 的 connect_phase 里，需要将 adapter 和 bus_sequencer 通过 set_sequencer 函数告知 reg_model 的 default_map，并将 default_map 设置为自动预测状态。

37 运行 sequence 的步骤

（1）创建 sequence 并例化
（2）配置或随机化序列

（3）启动 sequence，将其挂载到 sequencer 上

38 UVM 中通信同步方式，与 SV 中通信同步方式相比有什么优势

优势：SV 中用来做线程间同步的几种方法，它们分别是 semaphore、event 和 mailbox。UVM 中，同步不再只局限于同一个对象中的各个线程，还要解决各个组件之间的线程同步问题。考虑到 UVM 各个组件的封闭性原则，并不推荐通过层次索引的形式在组件中来索引公共的 event 或者 semaphore。UVM 为了解决封闭性的问题，定义了如下的类来满足组件之间的同步：

uvm_barrier：对多个线程进行同步协调，同时为了解决组件独立运作的封闭性需要，定义了新的类 uvm_barrier_pool 来全局管理这些 uvm_barrier。uvm_barrier 可以设置一定的等待域值，仅在有不少于该域值的进程在等待该对象时才触发该事件，同时激活所有正在等待的进程，使其继续进行。

uvm_event： uvm_event 是对 SV 中 event 类的一次封装。不同的组件可以通过 uvm_event_pool 这一**全局资源池**来共享**同一个** uvm_event。通过唯一的资源池对象，在环境中任何一个地方的组件都可以从资源池中获取共同的对象，避免了组件之间的互相依赖。通过 uvm_event_pool::get_global 得到一个 uvm_event 对象的句柄。两个地方调用 get_global，先调用的会创建这个 pool，后调用的只是获得这个 pool 的句柄

event 与 uvm_event 的区别：

（1）触发方式不同。event 通过->触发，@等待；uvm_event 通过 trigger()来触发，wait_trigger()等待。

（2）event 无法传递数据，而 uvm_event 可以传递数据。

（3）event 无法获取等待它的进程的数目，而 uvm_event 可以通过 get_num_waiters()来获取等待它的进程数目。

（4）event 触发时无法直接触发回调函数，而 uvm_event 可以通过 add_callback 函数来添加回调函数。

39 monitor 存在的必要性

第一，在一个大型的项目中，driver 根据某一协议发送数据，而 monitor 根据这种协议收集数据，如果 driver 和 monitor 由不同的人员实现，那么可以大大减少其中任何一方对协议理解的错误；第二，从代码的重用角度考虑，使用 monitor 是非常有必要的。

40 配置 agent 变量 is_active 必要性

在 dut 的输出端口上，往往只需要监测数据而不需要发送激励，因此只需要在 agent 中例化 monitor 即可，而不需要例化 driver 和 sequencer。

41 是否只能在 build_phase 里进行例化

component 只能在 build_phase 里例化，而 object 可以在任何 phase 里例化。

42 打印信息的分级与控制

在 UVM 中对于打印信息的控制主要通过信息的冗余度 verbosity 阈值和严重性

severity 来控制。

信息的严重性：四个等级—UVM_INFO、UVM_WARNING、UVM_ERROR、UVM_FATAL，信息的打印通过对应的宏定义实现；其中`uvm_info 宏用于打印信息，可以设置冗余度阈值，其余宏不可以。

信息的冗余度：LOW，MEDIUM，HIGH。如果设置的冗余度级别小于等于默认的阈值，则会显示打印信息,否则不会显示。UVM 默认的冗余度阈值为 UVM_MEDIUM,默认打印 medium 与 low。默认阈值也可以修改，通过命令行 +UVM_VERBOSITY=UVM_HIGH 或者函数 set_report_verbosity_level（UVM_HIGH）可修改为其他级别。

43 对 UVM 验证方法学以及 UVM 库的理解

第一，UVM以 sv 语言为基础搭建验证平台框架，将验证过程中可复用和标准化的部分都规定在类库中，这样可以减轻构建环境的负担，类似于 Python 里面的类库，UVM 的类库主要由 object、component、factory、transaction 和 sequence，report、register model 类等组成。

第二，uvm是一个框架，可以按照需要自己去搭建，流程规范，在写的过程中不用过多考虑语言实现问题，可以把精力放在场景的设计实现上。

第三，uvm是一个分层的测试平台，看起来比较复杂，但是 uvm 将大的任务分解成小的任务交给各自组件完成,各个组件相互独立又存在联系,能够快速搭建一个需要的测试平台，重用性高，会在大型验证中减轻工作负担。

44 set_drain_time/仿真结束时间

DUT 处理数据需要一定时间 t，如果 sequence 在发完最后一个 transaction 就 drop_objection，那么 t 时刻后 DUT 输出的包将无法接收到，因此 uvm 中通过设置 set_drain_time 来解决这一问题。当 UVM 在 main_phase 检测到所有的 objection 被撤销后，接下来会检查有没有设置 drain_time，如果有，则延迟 drain_time 后再进入 post_main_phase。

45 parent 机制的好处

（1）通过 parent 机制可以帮助组件找到它的父亲与孩子；
（2）对于 phase 机制可以通过 parent 进行一步一步执行，如对于 connect phase 依靠 parent 关系由底向上一步一步连接。

46 配置为什么要放在对象创建之前

确保所配置的变量都在子一级组件之前进行过配置,避免出现配置变量无法获取的现象。

47 new()与 create()的区别

new()与 create()都是为对象实例分配内存，但 create()方法可以从 factory 创建实例对象,这允许使用工厂机制进行重载时将所需对象替换为不同类型的对象,而无需编码。

48 set_timeout

如果代码在运行仿真的过程中出现锁死状态，仿真时间在消耗但仿真进度却停滞不前，这时候可以通过 set_timeout 函数设置超时时间，如果超出了设置的时间，那么给出一条 uvm_fatal 的提示信息，并退出仿真。

49 接口可以传到 sequence 中吗

可以通过 config_db 先将 virtual_interface 从 tb 传递到 sqr 中，然后 seq 调用 p_sequencer 从 sqr 中获取到接口。（至于能不能直接将接口从 tb 通过 config_db 直接 set 到 sequence 中，存疑，白皮书上说 sequence 机制对 config_db 提供了支持，但是没用过）

50 sequencer 可以和 driver 合并使用吗

不可以，从重用性角度考虑，比如 sequence 挂载，sqr 的仲裁等机制的实现等都需要交由 driver 完成，不利于验证平台的复用。

51 约束块发生了冲突了怎么办

约束冲突时：
（1）后约束覆盖前约束
（2）子类约束覆盖父类约束
（3）外部约束覆盖内部约束
（4）强约束覆盖 soft 约束

52 uvm_tlm_fifo 相较于 mailbox 的优势

（1）fifo 内置了许多端口
（2）mailbox 不支持一对多通信，而 fifo 可以

53 为什么要避免绝对路径的使用，如何避免

绝对路径的使用大大减弱了验证平台的可移植性。可使用宏和接口避免。

54 scb 的端口 analysis_imp 接收来自 monitor 与 model 的数据，就要在 scb 定义两个 write 函数，如何避免方法名冲突问题？

（1）通过宏声明端口
uvm_analysis_imp_dec（后缀，如_monitor），uvm 根据后缀定义 write 函数如 uvm_analysis_imp_monitor，这样当使用哪个端口，就会调用端口对应的 write 函数，避免了方法名冲突的问题。
（2）使用 fifo，fifo 的本质一块缓存加上两个 imp 端口，scb 中 blocking_get_port 端口可以主动从 fifo 获取数据，minitor 端口还是 analysis_port，fifo 端口是 analysis_imp，这样就不用定义 write 函数了，比 1 更加方便。

55 寄存器模型内建 sequence

复位/读写/地址映射

序列	测试级别	说明
uvm reg hw reset seq	uvm reg block uvm reg	检查寄存器模型复位值是否与硬件复位值一致
uvm reg_single bit bash seq	uvm reg	检查所有支持读写访问的域,对每一个可读写域分别写入 1 和 0 并且读出后做比较，用来检查寄存器域属性的有效性
uvm reg bit bash seq	uvm reg block	对包含的所有 uvm_reg 或者子 uvm_reg block 执行 uvm reg single bit bash seq 检查
uvm reg_single_access_seq	uvm reg	先从前门写入寄存器,而后从后门读回数值比对:接下来从后门写入寄存器，又从前门读回写入值比对.这种方式要求寄存器的 HDL 路径已经完成映射，用来检查寄存器映射的有效性
uvm reg access seq	uvm reg block	对包含的所有 uvm reg 都执行 uvm reg single access seq
uvm reg shared access seq	uvm reg	针对 uvm_reg 被包含在多个 uvm_reg_map 中时，先从一个 map 写入数值，其后从所有的 map 中读回这个寄存器数值,用来检查所有可能访问寄存器路径的有效性.

56 导入 UVM pkg 的时候，为什么要单独再 include 一下 macro.svh

uvm_macros.svh 是包含了众多宏定义的头文件，将其 include 进来可以让宏定义识别成功，否则编译时会报错。

57 介绍 field、reg、map、block

uvm_reg_field 用来针对寄存器功能域来构建对应的比特位
uvm_reg 与寄存器相匹配，其内部可以例化和配置多个 uvm_reg_field 对象
uvm_reg_map 用来储存各个寄存器的偏移地址和访问属性，并将其转换成可以访问的物理地址
uvm_reg_block 可以容纳多个寄存器（uvm_reg）和 map

58 你了解 ovm、vmm 他们是什么吗，和 uvm 什么关系？

VMM(Synopsys)中集成了寄存器解决方案 RAL。
OVM（Cadence 和 Mentor）引进了 factory 机制，但是它里面没有寄存器解决方案。
UVM 几乎完全继承了 OVM ，同时又采纳了 Synopsys 在 VMM 中的寄存器解决方案 RAL。同时，还吸收了 VMM 中的一些优秀的实现方式。

59 driver 里驱动为什么用非阻塞<=，monitor 为啥用阻塞赋值=?

防止竞争冒险

60 pre_body 和 post_body 什么情况下会被调用？

只有通过 start 启动 sequence 时,才会调用 pre_body()和 post_body(),而使用`uvm_do 等系列宏不会调用。

61 overload 和 override

override 表示方法重写或方法覆盖
对于子类从父类继承的方法中，如果子类重新声明了与其名称相同，参数相同，
返回类型也相同的方法时，就说子类重写或覆盖了父类的方法。

overload 表示方法重载
在某个类中存在多个名称相同但参数列表不同的方法时，就说这些方法相互构成重载
参数列表不同是指，参数的个数，类型，顺序中至少有一项不相同。

二 SV 部分

1 SV 的优势

SV 支持面向对象的编程，支持断言，支持更多的数据类型，支持覆盖率收集，既可以当做设计又可以当做验证语言。

2 数组与队列

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。
队列常用操作方法：insert：插入元素 pop_front：队首移出元素 push_back：队尾放入元素 delete：删除元素 shuffle：打乱顺序
定宽数组：属于静态数组，编译时便已经确定大小。其可以分为合并数组和非合并数组:合并数组是定义在类型后面，名字前面，其存储是连续的，如 bit[7:0][3:0]array;非合并数组定义在名字后面，存储是不连续的，如 bit[7:0]array[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: 针对需要超大空间但又不需要所有数据的时候使用，通过一个索引值和一个数据组成。
关联数组操作方法：
Num()和 size():返回关联数组中 item 的数目，如果数组为空，则返回 0。

Delete():如果指定了索引，则 delete()方法**将删除指定索引处的 item**。如果要删除的条目不存在，该方法不会发出警告。如果没有指定索引，则 delete()方法将删除数组中的所有元素。

Exists():**检查数组中指定索引处是否存在元素**。如果元素存在，返回 1;否则，返回 0。

First():将关联数组中第一个(最小)索引的值赋给给定的索引变量。如果数组为空，则返回 0;否则，返回 1。

Last():将关联数组中最后(最大)个索引的值赋给给定的索引变量。如果数组为空，则 0;否则，返回 1。

理解:

合并数组：bit[3:0][7:0]array：存储是连续的，32位不存放完不会开辟新空间

非合并数组：bit[7:0]array[2:0]，存储是不连续的，32位中即使没有使用也会开辟空间

3 SV数据类型

四值：integer，reg，logic，wire，time
二值：byte，int，longint，shortint，bit，real
有符号类型：byte、int、longint、shortint、integer，real
无符号类型：bit、logic、reg、net-type（如 wire、tri），time
多驱动时用 wire。
二值逻辑仿真工具开辟的存储空间更小且行为更接近真实的电路;使用四值逻辑是因为实际过程中会出现错误，从而出现 x 和 z 态来提示出错。二值逻辑的默认值是 0，四值逻辑的默认值是 x。

4 声明和例化的区别

声明是声明一个变量，其中保存类对象的句柄。
例化是创建对象，为其分配内存空间，并将声明的句柄指向这段内存空间。

5 fork join、fork join_any与fork join_none区别

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。
wait fork: 会引起调用进程阻塞，直到它的所有子进程结束。
disable fork: 用来终止所有活跃进程。

6 任务与函数区别

（1）函数能调用另一个函数，但不能调用任务，任务能调用另一个任务，也能调用另一个函数。
（2）函数总是在仿真 0 时刻就开始执行，任务可以在非零时刻执行。
（3）函数一定不能包含任何延迟、事件或者时序控制声明语句，而任务可以包含。
（4）函数至少有一个或多个输入变量，不能有输出(output)或者双向(inout)变量，任务可以没有或者多个输入(input)、输出和双向变量 。
（5）函数只能返回一个值，任务不返回任何值，任务可以通过输出或者双向变量传递多个值。

7 接口 interface、时钟块 clock blocking 与 modport

接口：用于对信号的封装，同时连接了 DUT 与验证环境。使用 interface ，不仅可以简化代码，而且提高可重用性
时钟块：基于时钟周期对信号进行驱动或者采样的方式，采样提前，驱动落后，从而消除信号竞争问题。

```
interface chnl_intf(input clk, input rstn);
  logic [31:0] ch_data;
  logic        ch_valid;
  logic        ch_ready;
  logic [ 5:0] ch_margin;
  // 定义时钟块
  clocking drv_ck @(posedge clk);
    //采样时间
    default input #1ns output #1ns;
    //声明变量方向
    output ch_data, ch_valid;
    input ch_ready, ch_margin;
  endclocking
endinterface

```

```
    clocking cb @(posedge clk);
       output enable;
       output kpe_ifmap;
       output kpe_weight;
       output ctrl_kpe_src0_enable;
       output ctrl_kpe_src1_enable;
       output ctrl_kpe_mul_enable;
       output ctrl_kpe_acc_enable;
       output ctrl_kpe_acc_rst;
       output ctrl_kpe_bypass;
       output stgr_precision_kpe_shift;
       output stgr_precision_ifmap;
       output stgr_precision_weight;
       //input  kpe_sum;
    endclocking
```

modport：用于对接口中的信号进行分组，并指定方向。因为有些信号是 dut 的输入，又是环境的输出，这就需要 modport 进行方向限制并进行分组

```
// 使用modport的接口
interface arb_if (input bit clk);
    logic [1:0] grant, request;
    logic rst;

    modport TEST (output request, rst,
                  input grant, clk);
    modport DUT (input request, rst, clk,
                 output grant);
    modport MONITOR (input request, grant, rst, clk);
endinterface
```

8 面向对象 OOP

封装、继承和多态
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：通过基类去得到一个子类，子类可以共享基类的属性和方法。
多态：得到扩展类后， 通过对基类方法进行 virtual 声明，这样当调用基类句柄指向子类时，会调用子类重写的方法，而基类和子类中方法有着同样的名字，但能够准确调用，叫做多态。
注：虚方法（多态）不会影响重载，父类无论有没有定义为虚方法（virtual），都可以对父类进行重载。

```
class basepacket;
   int A=1;
   int A=2;

   function void printA;
      $display("Basepacket::A is % d", A);
   endfunction:printA

   virtual function void printB;
      $display("Basepacket::B is % d", B);
   endfunction:printB

endclass:Basepacket
```

```
class My_packet extends Basepacket;
   int A = 3;
   int B =4;

   function void printA;
      $display("My_packet::A is % d", A);
   endfunction : printA

   virtual function void printB;
      $display("My_packet::B is % d", B);
   endfunction:printB

endclass :My_packet
 ...
 Basepacket p1 = new;
 My_packet p2 = new;
 intial begin
 p1.printA;    //打印Basepacket::A is 1
 p1.printB;    //打印Basepacket::B is 2
 p1=p2;        //p1指向My_packet 对象
 p1.printA;    //打印Basepacket::A is 1
 p1.printB;    //打印My_packet::B is 4
 p2.printA;    //打印My_packet::A is 3
 p2.printB;    //打印My_packet::B is 4
 end
```

9 事件等待与触发

等待：@与 wait()
触发：->
@与 wait 的区别：
（1）wait()是电平敏感触发，即只要()中的内容为 1 就触发，@是边沿敏感触发，只有发生 0/1 跳变才触发。
（2）wait 是只等待一次，@每时每刻都在等待
（3）@会阻塞一个进程，直到@的事件被触发(->)后，该进程才会 unblock。如果事件触发(->event)在@event 先执行，则@event 的进程会一直被阻塞住。如果->event 和@event 发生在同一个 time step，就会造成竞争冒险，因为无法确认他们哪一个先执行。
（4）event 的 triggered 属性的持续时间是一个 time slot，因此它解决了触发事件和

等待事件在同一个 time step时的竞争冒险问题。只要 wait(event.triggered)在→event 之前执行，或者在同一个 time step执行，都能正确地等到事件。

注意：若用→和@搭配，一定要先@再→；若用→和 wait搭配，则谁先谁后都可以。

```
module event_test();
  event a;    //使用关键字event来声明一个事件a

  initial begin
    #50ns;
    ->a;
  end

  initial begin
    #50ns;
    @a; //第一个进程在50ns后触发了事件a，第二个进程在1ns的时候等待a，
  end   //有可能等的到，有可能等不到，产生竞争

endmodule
```

```
module event_test();
  event a;    //使用关键字event来声明一个事件a

  initial begin
    #50;
    ->a;
    $display("Event a is being triggered!");
  end


  initial begin
    #20;
    wait(a.triggered); //使用wait来等待事件a，这种方式是一定可以等到a的
    $display("#20 a.triggered!");
  end

  initial begin
    #50;
    wait(a.triggered); //使用wait来等待事件a，这种方式是一定可以等到a的，这是和使用@来等待的区别
    $display("#50 a.triggered!");
  end

  initial begin
    #60;
    wait(a.triggered); //使用wait来等待事件a，a会被trigger一次，并且是发生在wait之前，永远等不到
    $display("#60 a.triggered!");  //不会被打印
  end

endmodule
```

如何理解 time slot 呢？

systemverilog 是为离散事件执行模型所定义的一种语言。怎么理解呢？离散指的是仿真时间上的离散性，仿真基于时间片进行，只对有效的时刻点进行仿真。而这个在仿真时间维度上的离散时间片，就称为 time slot 或者 time slot。

10 断言

断言用来**将设计功能和时序作比较的属性描述。**

断言可以用来完成：检查设计的内容；提高设计的可视度和调试能力；检查设计特性在验证中是否被覆盖。

断言分类：

（1）立即断言：非时序的；执行时如同过程语句；可以在过程块或者函数与方法中调用。

（2）并行断言：时序性的；关键词 property 用来区分立即断言和并行断言；之所以称为并行，是指他们与设计模块一同并行执行。

立即断言可以结合$fata、$error、$warning、$info 给出不同严重级别的消息提示。

并行断言只会在时钟的边沿激活，变量的值是采样到的值。

11 rand 与 randc 区别

rand：在取值范围内随机取一个值，每次随机到的值的概率是相同的（取完之后放回）。

randc：随机一次，就少一个值，当所有值都随机到后，才会重复取值（取完之后不放回）。

12 约束（constraint 要会写）

（1）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。可通过 constraint_mode (0)关闭约束。
如果想把 constraint 里的某个数据不让它随机，可以使用 rand_mode(0)

用法：

```
addr dist{2:=5, [10:12]:=8};
//addr=2,      weight5
//addr=10,     weight8
//addr = 11,   weight 8
//addr = 12,   weight 8
addr dist{2:/5, [10:12]:/8};
//addr=2,      weight5
//addr = 10,   weight 8/3
//addr = 11,   weight 8/3
//addr = 12,   weight 8/3

用法：
constraint addr range { addr dist { 2 := 3, 4 := 7}};
// addr是变量，意为addr = 2占3/10，addr = 4占7/10
```

（2）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。
（3）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，但是不可以连续使用，如 min<wxm<max 这是错误的。

13 多线程同步调度方法 event，mailbox，semaphore（重点是 event）

mailbox:主要用于两个线程之间的数据通信，通过 put()，get()，peek()函数进行数据的发送和获取（try_get 非阻塞）。如果 mailbox 满了，那么使用 put()就会阻塞住，即卡住，如果 mailbox 空了，使用 get()就会阻塞住。使用前需要用 new()进行例化。
event:事件主要是通过事件触发和事件等待进行两个线程间的同步运行。使用@(event)或者 wait(event.trigger)进行等待，→进行触发。
semaphore:通过 key 的获取和返回实现多个线程对同一资源的访问，使用 put 和 get 函数获取返回 key（try_get 非阻塞）。使用前需要用 new()进行例化。

14 ref 的类型，ref 的优点

input、output 和 inout 均属于值传递，所有的数据需要在每次方法调用的时候被复制，这种操作很占内存空间。ref 相当于一个指针，在方法调用时引用传递进来的数据，而不是复制传进来的数据，如果在方法内对数据进行修改，那么在方法外部的数据也会被修改。如果不希望方法修改传递进来的数据，那么可以给 ref 变量添加 const 修饰符，这样一旦方法内修改数据的值，那么程序就会报错。（外部修改数据，方法内可以显示；方法内修改，

外部也可以显示，方法必须是动态的，用 automatic 修饰）

15 `ifndef `define `ifdef `endif

使用 ifndef define endif 的目的：为了防止同一个文件在编译时被重复编译，引起多重定义的问题。

```
1 `ifndef APB_MASTER_DRIVER_SV
2 `define APB_MASTER_DRIVER_SV
3
4 class apb_master_driver extends uvm_dr
5     ...
6 endclass
7
8 `endif
```

ifndef 的含义：即"if not defined"，也就是说，当文件编译到这一行，如果这个文件还没有被编译过，也就是首次编译，就会执行后续的`define xxx 这句话，把后续的代码定义一次。反之，则不会再重复编译。
ifdef 的含义：即"if defined"，与 ifndef 的作用相反，如果已经编译过，那么则继续执行后面的代码。
enif 的含义：出现 ifndef 或者 ifdef 作为开头，程序块的末尾就需要有 endif 作为结束的标识。
形式有两种：
（1） 如果尚未被编译，则执行后续程序块；反之则不执行。

```
1 `ifndef xxx
2 `define xxx
3   程序块
4 `endif
```

（2） 如果已经编译过，则执行程序 1；反之则执行程序 2。

```
1 `ifdef xxx
2     程序1
3 `else
4     程序2
5 `endif
```

16public、protected、local 的区别

（1）public：该类，子类和外部均可以访问成员。
（2）protected：该类或者子类可以访问成员，而外部无法访问。
（3）local：只有该类可以访问成员，子类和外部均无法访问。
public 属于全局变量，protected 和 local 属于局部变量。

17 $random，$urandom，$urandom_range

$random()：随机产生 32bit 有符号数。
$urandom()：随机产生 32bit 无符号数。
$urandom_range()：随机产生指定范围内的无符号随机整数。

18 功能覆盖率、代码覆盖率与断言覆盖率

代码覆盖率：是针对 RTL 设计代码运行完备度的体现，包括：
行覆盖率：记录程序的各行代码被执行的情况。
条件覆盖率：记录各个条件中的逻辑操作数被覆盖的情况。
跳转覆盖率：记录单 bit 信号变量的值为 0 到 1 或 1 到 0 跳转情况。
分支覆盖率：指在 if，case，for，forever，while 等语句中各个分支的执行情况。
状态机覆盖率：用来记录状态机的各种状态被进入的次数以及状态之间的跳转情况。
功能覆盖率:功能覆盖率是用来衡量设计所实现的各种功能是否符合设计文档的功能点要求。
断言覆盖率：用于检查几个信号之间的关系，常用在检查时序上的错误，测量断言被触发的频繁程度。

19 代码覆盖率与功能覆盖率一高一低的原因

（1）代码覆盖率低 + 功能覆盖率高
A、有可能是 cover group 写得不完备，测试点分解可能也不完备。
B、DUT 中有大量冗余代码。
（2）代码覆盖率高 + 功能覆盖率低
A、功能覆盖率的采样有问题，即相关的场景都打到了，但 cover group 没有采样到。
B、cover group 中的一些 cross bin 或者 conner 点没覆盖到。
提高覆盖率的方法：
（1）新增约束
（2）添加测试用例
（3）使用不同的种子运行现有的测试用例
覆盖率的收集是一个迭代过程，需要不断反复的运行测试用例。

20 covergroup、coverpoint 和 bins 的区别，sample()以哪个为对象进行采样（要知道 covergroup 代码怎么写）

covergroup：覆盖组结构（covergroup construct）是一种用户自定义的结构类型，一旦被定义就可以像 class 一样通过 new（）创建多个实例。覆盖组可以定义在 module、program、interface 以及 class 中。
每一个覆盖组（covergroup）都必须明确以下内容：
一个时钟事件以用来同步对覆盖点的采样；
一组覆盖点（coverpoints），也就是需要采样的变量；
覆盖点之间的交叉覆盖；
可选的形式参数；
覆盖率选项；

```
covergroup cov_grp;
  cov_p1: coverpoint a;//cov_p1为覆盖点名，a为覆盖点中的变量名，也就是模块中的变量名
endgroup

cov_grp cov_inst = new();
cov_inst.sample();          //sample函数收集覆盖率
```

通过内建的 **sample()方法或者@来触发覆盖点**的采样。

coverpoint：一个覆盖组可以包含多个覆盖点，每个覆盖点有一组显式 bins 值，bins 值可由用户自己定义，每个 bins 值与采样的变量或者变量的转换有关。一个覆盖点可以是一个整型变量也可以是一个整型表达式。
ignore bins：忽略采样到的 bin，不计入统计
illgeal bins:如果采样到了该非法 bin，那么仿真会报 error
Binsof 一般与 intersect 连用，构成 binsof (x) intersect {y}，表示覆盖点 x 和给定表达式 y 的交集。
Vcs 自带收集代码覆盖率的工具，功能覆盖率的收集是通过 sample()函数采样实现的；

21 DPI 接口的使用，连接 C 语言与验证平台

DPI 能够连接 SV 与其他编程语言，如 C/C++。通过 import 导入一个 C 子程序，就可以像调用 SV 中的子程序一样来调用它。

SV 和 C 语言之间传递的最基本的数据类型是 int，双状态的 32 位的数据类型，通过 import 声明定义 C 任务和函数的原型,带有返回值的 C 函数被映射成一个 systemverilog 的函数，void 类型的 C 函数被映射为一个 SV 的任务或者 void 函数，通过“DPI-C”引入的 C 函数，可以直接在 function 中调用，但是只在该 DPI 被声明的空间内有效。

可以在 package 中封装所有的 DPI 函数,打包为 function。然后在需要的地方 import package。使用关键字 DPI-C 表示，使用压缩值(packed)的方式来保存数据类型。

SV 与 C 语言之间的数据类型映射

SV	C
byte	char
int	int
bit	svBit
logic, reg	svlogic
shortint	short int
longint	long long int

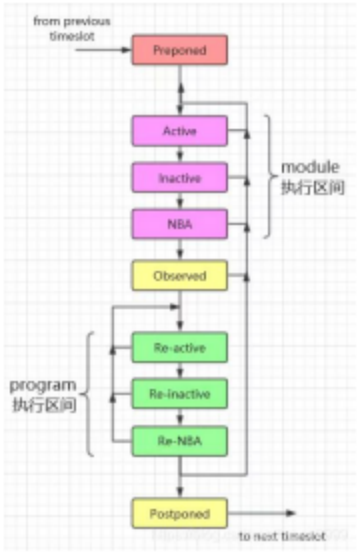
22 logic 与 wire，reg 的区别，各自有什么优劣

引入 logic 目的：方便验证人员驱动和连接硬件模块，省去考虑使用 reg 还是 wire 的精力。
引入 bit 目的：引入双状态数据类型**有利于提高仿真器的性能并减少内存的使用。**
logic 与 reg 区别：
logic 是 reg 类型的改进，它既可被过程赋值也能被连续赋值，编译器可自动推断 logic 是 reg 还是 wire。logic 只允许一个输入，不能被多重驱动，所以 inout 类型端口不能定

义为 logic。如果误接了多个驱动，使用 logic 在编译时会报错。所以单驱动时用 logic，多驱动用 wire。

23 timeslot

RTL 仿真时，可以看见在同一点发生同时变化的信号，但其实发生的顺序是有先后次序的，我们将仿真进程这一点变化的时间抽象概括并定义为 timeslot(时间片)。delta-cyle 是仿真进程中的最小时间延迟，是给组合电路的驱动添加的一个延迟。无论是 timeslot 还是 delta-cyle 都是用于解决同一时间点的信号的竞争问题。SV 仿真调度机制如下图:



域名	作用
preponed	时间片入口，断言采样时间
active/inactive/NBA	module 中代码执行时间
observed	断言检查时间
re-active/re-inactive/re-NBA	program 中代码执行时间
postponed	时间片出口

24 静态(static)变量与动态(automatic)变量

静态变量的特点：声明时就对其进行初始化；可被该类所有实例所共享，使用范围仅限该类；生命周期不会随对象的销毁而结束。

动态变量的特点：new()的时候才会初始化；生命周期随对象而存在，对象销毁，其生命周期也就结束了；必须在对象例化后才可以调用。

静态方法：内部变量都是静态的；仿真开始时就被创建，可被多个进程和方法共享 ，具有全局的静态声明周期。

动态方法：内部变量都是动态的，进入该方法动态变量会被创建，离开该方法就会被销毁。

不能在静态方法中使用动态变量。

25 隐式转换与显示转换，动态转换与静态转换

类型转换可以分为静态转换和动态转换。

静态转换即需要在转换的表达式前加上单引号即可，该方式并不会对转换值做检查。如果发生转换失败，我们也无从得知。

```
byte a;
b = int'(a);
c = unsigned'(a);
```

动态转换即需要使用系统函数$cast(tgt，src)做转换，转换失败会返回 0。

静态转换和动态转换均需要操作符号或者系统函数介入，统称为显式转换。

不需要进行转换的一些操作，我们称之为隐式转换。例如高位宽变量赋值给低位宽变量时，会截断高位。

26 堆和栈的区别

（1）管理方式不同。**栈由操作系统自动分配释放**，无需我们手动控制；**堆的申请和释放工作由程序员控制**，容易产生内存泄漏；

（2）空间大小不同。**每个进程拥有的栈大小要远远小于堆大小。**

（3）生长方向不同。**堆的生长方向向上，内存地址由低到高；栈的生长方向向下，内存地址由高到低。**

（4）分配方式不同。**堆都是动态分配的，没有静态分配的堆。栈有 2 种分配方式：静态分配和动态分配。**静态分配是由操作系统完成的；动态分配由 alloca()函数分配，但是栈的动态分配和堆是不同的，它的动态分配是由操作系统进行释放，无需我们手工实现。

（5）分配效率不同。栈由操作系统自动分配，会在硬件层级对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是由 C/C++提供的库函数或运算符来完成申请与管理，实现机制较为复杂，频繁的内存申请容易产生内存碎片。显然，**堆的效率比栈要低得多。**

（6）存放内容不同。**栈存放的内容，函数返回地址、相关参数、局部变量和寄存器内容等，而堆中具体存放内容是由程序员来填充的。**

27 类和对象的区别

1）class（类）：包含变量和方法的基本模块，是“软件”盒子。

2）object（对象）：类的实例，在 SV 中可以使用 class 来例化，是“软件”例化。

3）handle（句柄）：用来指向对象的指针，来索引对象的变量和方法。

步骤： 定义一个类→声明一个句柄→new 函数创建对象（对象就是例化的后的实例）

28 深拷贝与浅拷贝

浅拷贝：拷贝对象中的成员变量，而对于对象中的方法和实例的句柄，拷贝前后共用同一内存空间，类似于引用。

深拷贝：拷贝对象中的所有成员变量以及嵌套的实例的内容并分配新的内存空间。

```
class A ;
  int [31:0]data_b;
    B  b;       //声明句柄b
  function new();
    b = new(); //创建对象，例化句柄
  endfunction
endclass
//若浅拷贝，则只拷贝data_b和句柄b
//若深拷贝，则拷贝data_b和句柄b所例化的对象
```

29 typedef 与 struct

struct 是一组变量或常数组成的集合，可以作为整体进行操作。
结构体分类：

填充型结构体： 内部变量存储是串行连续的。

非填充型结构体：内部变量存储是并行不连续的。

typedef 常与 struct 连用，用来创建新的类型，实现对同一结构体的多次例化，提高结构体的重用性。

```
typedef struct {
            int weight;
            int height;
            logic [7:0] legs;
            logic [1:0] hands;

} animal;      //结构体名
animal duck;  //例化结构体，进行空间分配
animal dog;   //例化结构体，进行空间分配

duck.legs = 8'b10001100; //时钟结构体的名字操作整个变量
```

类与结构体的区别：
类 class 在声明之后需要例化才会构建对象实体，但是结构体 struct 在声明的时候就已经开辟了内存；
class 中可以存在方法，struct 不可以。

30 new()与 new[]的区别

new()例化 class，new[]例化数组

31 break；continue；return 的含义

（1）break 语句结束整个循环。
（2）continue 立即结束本次循环，继续执行下一次循环。
（3）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)。

32 全局变量与局部变量

全局变量：伴随程序开始执行到结束一直存在，具有静态生命周期
局部变量：生命周期同其所在域共存亡，如函数/任务中的临时变量，在方法调用结束后，其生命周期也就结束了，具有动态生命周期。

33 断言的编写

第一是激励序列 sequence 的编写，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；
第三个就是 assert 的编写，调用 property 即可，编写完断言后，可以将它用在很多地方上，比如 interface 处，基本能够检测到信号的地方都可以进行断言检测。

34 cast

（1） $cast 做枚举类型转换：
枚举类型的缺省类型为双状态 int,可以把枚举类型变量的值直接赋值给非枚举变量 如 int，但 SV 不允许在没有进行显示类型转换的情况下把 int 变量直接赋值给枚举变量。SV 要求显式的类型转换的目的在于让你意识到可能的数据越界情况。

```
typedef enum bit[1:0] {RED=0,BLUE=1,GREEN=2} COLOR_E;
COLOR_E color,c2;
int c;

initial begin
    color = BLUE;   // 赋值一个已知的合法的值
    c = color;      // 将枚举变量赋值给int,此时为 1
    c = c+1;        // int型变量递增
    if(!$cast(color,c))      // 将整型显示转换回枚举类型，如果越界会报错
        $display("cast failed for c=%0d",c);   // c的值此时为2
    $display("Color is %0d/%s",color,color.name());
    c++;                     // c的值为3，对于枚举类型已然越界
    c2 = COLOR_E'(c);        // 不做类型检查,下句c2.name()由于越界而打印不出
    $display("c2 is %0d/$s",c2,c2.name());     // 打印:c2 is 3/
end
```

（2）$cast 做句柄类型转换
这里涉及到类型转换，向上类型转换和向下类型转换：
向上类型转换：子类句柄赋给父类句柄。向上类型转换没问题，可以直接将父类句柄指向子类对象，这样父类句柄就可以索引到子类中同名变量和方法，而对于子类独有的变量和重写的方法，依然索引不到，如：

```
base_clase   bc;
sub_class    sc = new();
bc = sc;                //父类句柄指向子类对象，这是正确的
```

如果使用$cast 进行向上类型转换也不可以，因为 **cast 检查的是对象类型**，由于父类句柄与子类句柄指向的是不同的对象，故会报错。
向下类型转换：将父类句柄赋给子类句柄。直接赋值会报错，因为 **SV 编译只检查句柄类型**，如果直接使用 cast 也会报错，因为 cast 检查的是对象类型，只有当父类句柄指向子类对象时，cast 才可以转换成功，此时子类句柄和父类句柄都指向子类对象。如：

```
base_class bc;
sub_class   sc;
bc = new();
sc = bc;                // 子类句柄指向父类对象（这是错误的）；
$cast(sc,bc);           // 此时通过cast方式仍然不行；

base_class bc;
sub_class  sc1,sc2;  //子类的两个句柄
sc2 = new();
bc = sc2;
sc1 = bc;        // 不行
$cast(sc1,bc);   // 通过cast方式可以实现，可以看到bc的句柄类型虽然是父类，但其指的对象类型是子类
```

子类创建两个对象，sc1 和 sc2，将 sc2 赋给父类句柄，这是一次向上类型转换，此时父类句柄指向子类对象，此时使用 cast 将父类句柄转换为子类对象($cast(sc1,父))，这样就

成功了。

总结：当父类句柄指向子类对象时，子类句柄可以指向父类对象；

理解：为什么向上类型转换可以直接赋值，而向下就不可以呢？原因在于子类既然继承了父类，就拥有父类的一切属性，除此之外，子类还有自己独特的个性，这些是父类没有的。当进行向上类型转换时，相当于父类的句柄指向子类对象，这样的话句柄仍然能对子类对象与父类相同的属性进行访问（父类句柄在子类里进行访问到的东西一定是子类有的，不会越界，因为子类的东西已经包含了父类）。但是反过来，如果向下类型转换也那么自由，当试图把子类的句柄指向父类的对象会发生什么呢？父类本来在内存里就划好了一小块地盘，但是因为子类含有比父类更丰富的属性，它很有可能会访问父类并不包含的资源，越界了。 这就会造成严重的内存溢出，所以向下类型是需要有严格的类型检查的，阻止非法转换。

35 mailbox，队列与 fifo

（1）mailbox 与队列

A、mailbox 必须通过 new()例化，而队列只需声明；

B、mailbox 可存储不同数据类型（不建议），队列只能存储相同类型的数据；

C、mailbox 中的 put()和 get()是阻塞方法。而队列对应的存取方式 push_back()和 pop_front()为非阻塞，会立即返回，在使用 queue 取数时应先填写 wait(queue.size()>0)。

D、传参过程差异：mailbox 传递并复制 mailbox 的指针，队列形参声明的若是 ref 则是指针，默认的 input 则是数组复制。

（2）mailbox 与 fifo

fifo 内置了许多端口，更方便用户使用；fifo 支持一对多的传输，而 mailbox 不支持。

（3）fifo 与队列

可以在队列中任意一个地方插入或删除元素，而在 fifo 中则不可以，fifo 只能按照先入先出的顺序。

36 请简述 SV 和 UVM 中重载的方法

SV 中，由于其面向对象的特性，可以利用虚函数的方法实现子类对父类的重载。

UVM 中，则可以通过工厂机制来实现 override。将所有的类都注册到工厂中，并通过工厂来创建对象。

37 import 与 include 的区别

`include 将文件中所有文本原样插入另一个文件中，对其进行复制。

import 不会复制文本内容，而是引入外部模块。

我觉得 import 会更好，不会占用更多的内存。

38 Sv 中 this 和 super 的作用

this 一般指代当前类成员变量；super 一般指代父类的函数。

39 clocking 中的 input #1ns 表示什么意思

input #1ns:在 clk 上升沿的前 1ns 对其进行输入采样
output #1ns:在事件的后 1ns 进行输出驱动
Monitor 先采样，driver 后驱动

40 abc 三个线程同步启动，ab 结束后就终止三个线程，怎么实现

自己的思路是在 fork...join...里实现，a，b 两个线程结束后触发事件 event1,2，在 fork 外面等待这两个事件，当两个事件都触发了，执行 disable fork。

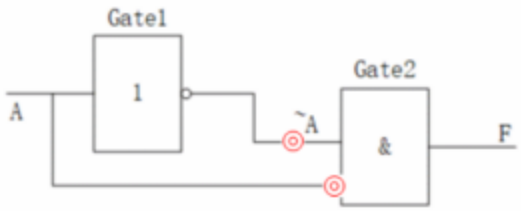
三设计部分

1 $monitor 与$display 区别

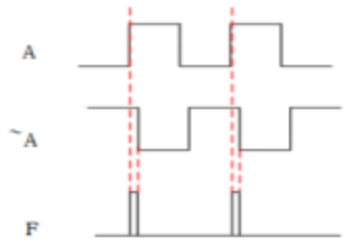
$monitor 与$display 都可以将需要观测的信号的数值显示在屏幕上。二者的主要区别是，执行$display 时，它将观测到的信号的当前数值显示在屏幕上；对于$monitor，当它观测的信号数值**发生变化时**才在屏幕上显示它的信号数值。

2 竞争与冒险，如何判断，如何消除

竞争：在组合电路中，输入变量经过不同途径传输后，到达电路中某一汇合点的时间有先有后的现象；
冒险：由于竞争而使电路输出发生瞬时错误的现象叫做冒险。（也就是由于竞争产生的毛刺叫做冒险）。
判断方法：
代数法：如果布尔式中有相反的信号则可能产生竞争和冒险现象
卡诺图：有两个相切的卡诺圈并且相切处没有被其他卡诺圈包围，就有可能出现竞争冒险；
解决方法：
1 加滤波电容；
2 引入选通信号；
3 增加冗余项；
4 信号延时法。



gate1 的输入到输出有延迟，导致~A滞后于 A 到达



~A 信号滞后于 A 到达就会导致尖峰脉冲信号产生

3 触发器与锁存器，各自优缺点

锁存器由电平触发，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。触发器由时钟触发，只有在时钟触发时才采样当前的输入，产生输出。
锁存器缺点：锁存器对输入电平敏感，受布线延迟影响较大，输出可能会产生毛刺。
锁存器优点：面积比触发器小，速度比触发器快；如果使用门电路来搭建锁存器和触发器，则锁存器消耗的门资源比触发器要少，功耗更低。

4 摩尔型与迷你状态机

Moore 状态机的输出仅与当前状态值有关，且只在时钟边沿到来时才会有状态变化。
Mealy 状态机的输出不仅与当前状态值有关，而且与当前输入值有关

5 跨时钟域，多 bit 信号能不能打两拍

单 bit

慢到快 打两拍（意思是说采集到的信号处于亚稳态，既不是 1 也不是 0，暂时处于中间态，此时需要打两拍让信号稳定下来，恢复到它本来的信号）
快到慢 在快时钟域上信号展宽，同步到慢时钟域上打两拍
两级触发器可防止亚稳态传播的原理：假设第一级触发器的输入不满足其建立保持时间，它在第一个脉冲沿到来后输出的数据就为亚稳态，那么在下一个脉冲沿到来之前，其输出的亚稳态数据在一段恢复时间后稳定下来，且稳定的数据满足第二级触发器的建立时间，如果都满足了，在下一个脉冲沿到来时，第二级触发器将不会出现亚稳态，因为其输入端的数据满足其建立保持时间。

理解：当第一级触发器采样异步输入之后，允许输出出现的亚稳态可以长达一个周期，在这个周期内，亚稳态特性减弱，最终会转化为稳态，两级触发器能大大降低亚稳态的概率。一般来说是两级触发器够了，但对于亚稳态比较强的可能不够。

多 bit

格雷码（主要避免亚稳态，要注意对于 3bit 数据必须在 0~7 范围内才能使用格雷码，如果是 0~6 就不能用，因为从 6 变到 0 格雷码变化了不止一位）。
异步 fifo
DMUX：相当于触发器的一个使能信号，判断源时钟域的单 bit 信号是否在目的时钟域成功同步，如果是，那么这个时间长度下多 bit 信号也可以同步过来且不违背建立时间。
握手处理
对于多 bit 数据跨时钟，各个 bit 之间路径延迟不一样，源时钟域给的数据是 2`b11，目的时钟域采样到的数据可能是 2`b10，因此不能采用打两拍的方法。

6 时序分析

静态时序分析是采用穷尽分析方法来提取出整个电路存在的所有时序路径，计算信号在这些路径上的传播延时，检查信号的建立和保持时间是否满足时序要求，通过对最大路径延时和最小路径延时的分析，找出违背时序约束的错误。它不需要输入向量就能穷尽所有的路径，且运行速度很快、占用内存较少，不仅可以对芯片设计进行全面的时序功能检查，而且还可利用时序分析的结果来优化设计，因此静态时序分析已经越来越多地被用到数字集成电路设计的验证中。

动态时序分析就是通常的仿真，因为不可能产生完备的测试向量，覆盖门级网表中的每

一条路径。因此在动态时序分析中，无法暴露一些路径上可能存在的时序问题；

7 低功耗设计方法

动态功耗，静态功耗，浪涌
动态功耗分为开关功耗和短路功耗，静态功耗是由漏电流引起的。
降低静态功耗的方法：电源门控；多阈值电压
降低动态功耗的方法：多电压供电；门控时钟；动态电压频率调节；并行结构与流水线技术
影响动态功耗的因素：电压，温度，工作频率
RTL 级降低功耗的方法：

(1)资源共享：如果在多处使用同一个逻辑运算，就直接复用就行了，不用每一处都重新计算一次。

(2)状态编码：使用格雷码或独热码编码。因为多 bit 信号频繁变化时会产生很多翻转功耗。

(3)门控时钟：时钟的翻转会产生很大的翻转功耗，因此在不需要工作的模块停止时钟树的翻转，使模块处于静态功耗，从而降低了整个系统的功耗。

8 阻塞与非阻塞赋值

（1）阻塞赋值（=）必须是阻塞赋值完成后，才进行下一条语句的执行；赋值一旦完成，等号左边的变量值立即变化。硬件没有对应的电路。（要点为串行，立即生效）
（2）非阻塞赋值（<=），在赋值开始时计算表达式右边的值，在本次仿真周期结束时才更新等号左边的值，即赋值不是立即生效的；非阻塞赋值允许块中其他语句同时执行。在同一个块中，非阻塞赋值表达式的书写顺序不影响赋值的结果。硬件有对应的电路。
（要点：并行，不是立即生效，同时执行）
建议在组合逻辑中使用阻塞赋值，在时序逻辑中使用非阻塞赋值。

9 亚稳态是什么，如何消除

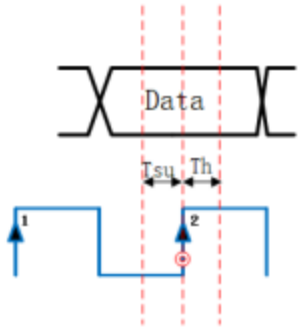
如果数据在时钟上升沿的建立时间和保持时间内发生跳变，则会产生亚稳态输出，即输出值在短时间内处于不确定态，有可能是 1，有可能是 0，也可能什么都不是，处于中间态，这个未知的状态便称为亚稳态。
产生原因：违反寄存器的建立时间和保持时间产生的。
消除方法：
(1)、跨时钟域信号传输可以使用 FIFO 进行数据缓冲。
(2)、复位电路采用异步复位、同步释放处理。
(3)、用反应更快的 FF(触发器)。
(4)、用边沿变化快速的时钟信号。
(5)、降低系统时钟频率。

10 格雷码有什么好处

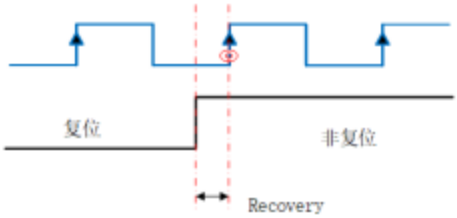
相邻之间只变 1bit，因此计数过程中最多出现一位错误，这样大大避免了亚稳态的发生。

11 建立时间与保持时间，恢复时间与移除时间，时间抖动和时间偏移

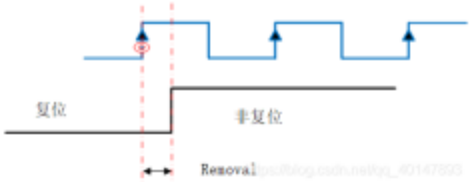
建立时间：在时钟沿到来之前，触发器输入端的数据必须保持不变的最小时间。
保持时间：在时钟沿到来之后，触发器输入端的数据必须保持不变的最小时间。



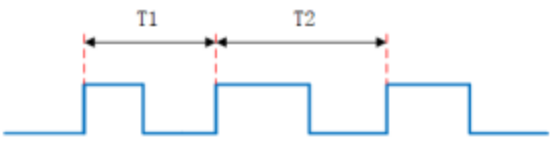
恢复时间：在撤销复位时，非复位状态必须在时钟有效沿到来之前一段时间到达。



去除时间：在撤销复位时，在时钟沿到来时之后复位信号还要保持的时间（异步复位信号撤销时离上一个有效时钟沿的最小时间长度）。

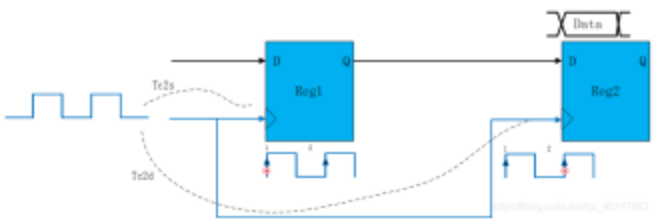


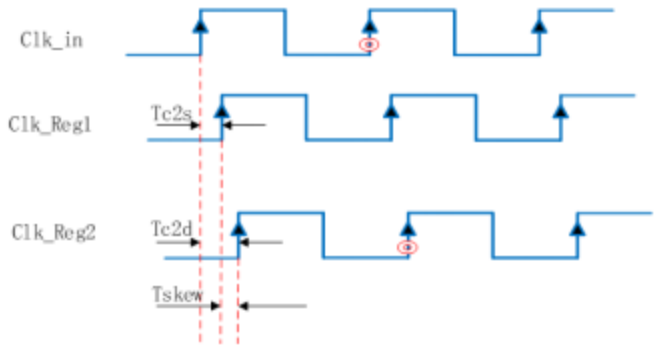
时间抖动：时钟抖动指的在某一个给定的点上时钟周期发生短暂性变化，使得时钟周期在不同的周期上可能加长或者缩短。抖动是在时钟发生器内部产生的，产生的原因和**晶格振动**以及内部电路有关。



时间抖动：T2-T1（频率上的不确定）

时钟偏斜：同一个时钟信号到达相邻两个时序单元的时间不一致，主要由布线长度以及负载不同引起的。





时钟偏斜= Tc2d - Tc2s（相位上的不确定）

12 同步电路与异步电路

同步电路：所有触发器的时钟输入端都接同一个时钟脉冲源，因而所有触发器的状态都与所加的时钟同步。
异步电路：电路中没有统一的时钟，有些触发器时钟与时钟脉冲相连，有些没有。
异步电路：
a) 电路核心逻辑是用组合电路实现；
b) 容易产生毛刺；
c) 不利于器件移植；
d) 不利于静态时序分析(STA)。
同步时序电路：
a)电路核心逻辑是用各种触发器实现；
b)电路主要信号都是时钟驱动触发器产生的；
c)可以很好的避免毛刺；
d)利于器件移植；
e)利于静态时序分析(STA)。

13 同步复位与异步复位

同步复位 ：复位信号只有在时钟上升沿到来时才能有效。

```
always@(posedge CLK)
```

优点
1） 利于仿真器仿真。
2） 可以滤除高于时钟频率的毛刺。
3） 有利于时序分析。
缺点
1） 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。
2） 由于大多数目标器件库内的 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，这样就会耗费较多的逻辑资源。

异步复位 ：无论时钟沿是否到来，只要复位信号有效，就进行复位。

```
always@(posedge CLK or negedge Rst_n)
```

优点：
1） 设计相对简单。
2） 因为大多数目标器件库的 dff 都有异步复位端口，因此采用异步复位可以节省资源。

3） 异步复位信号识别方便，而且可以很方便的使用 FPGA 的全局复位端口 GSR。
缺点：
1） 复位信号容易受到毛刺的影响。
2） 若复位释放刚好在时钟有效沿附近时，很容易使寄存器输出出现亚稳态。
推荐采用异步复位同步释放：复位信号到来的时候不与时钟同步，而复位信号释放的时候与时钟同步，这样可以避免亚稳态的产生。

14 同步 FIFO 与异步 FIFO

FIFO是一种先进先出数据缓存器，它与普通存储器的区别是没有外部读写地址线，使用起来非常简单，缺点是只能顺序读写，而不能随机读写。
（1）FIFO分类
同步 FIFO：读和写是同一个时钟。它的作用一般是做交互数据的一个缓冲。
异步 FIFO：读和写是不同的时钟，它有两个主要的作用，一个是实现数据在不同时钟域进行传递，另一个作用就是实现不同数据的位宽变换（比如输入是 16bit ，输出是 8bit）。
（2） 空满判断
同步 fifo（二进制扩展一位）：
空的条件：读写地址完全相同。
满的条件：读写地址的最高位不同其余位均相同。
异步 fifo（格雷码）：
空的条件：读写地址完全相同。
满的条件：读写地址的高 2 位不同，其余位均相同。

15 线与

线与逻辑是两个输出信号相连可以实现与的功能。在硬件上，要用 oc 门来实现（漏极或者集电极开路），为了防止因灌电流过大而烧坏 oc 门，应在 oc 门输出端接一个上拉电阻。（线或是下拉电阻）

16 门控时钟

门控时钟技术(gating clock) 是通过在时钟路径上增加逻辑门对时钟进行控制，使电路的部分模块在不需要工作时停止时钟树的翻转，而并不影响原本的逻辑状态。此时该模块的功耗相当于静态功耗，从而降低了整个系统的功耗。

17 数字电路中的逻辑电路，有什么区别

组合逻辑，时序逻辑。
组合逻辑电路：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。
时序逻辑电路: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。
常见组合逻辑电路：加法器、比较器、译码器、编码器
常见时序逻辑电路：触发器、移位寄存器、计数器、序列信号发生器

18 用 MOS 管搭出二输入与非门、或非门

与非门：上面两个 PMOS 并联，下面两个 NMOS 串联。

或非门：上面两个 PMOS 串联，下面两个 NMOS 并联。
（与非门：上并下串 或非门：上串下并）

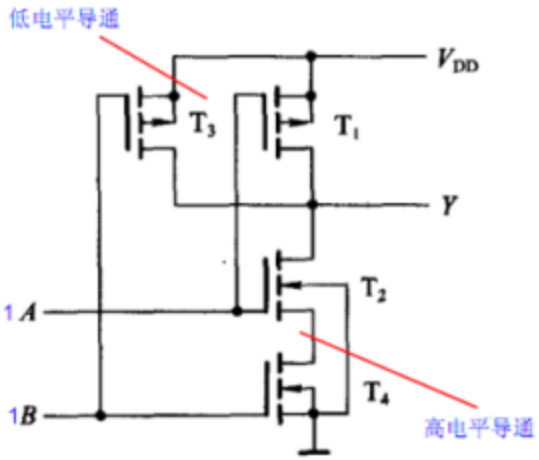


图 3.3.27 CMOS 与非门 上并下串

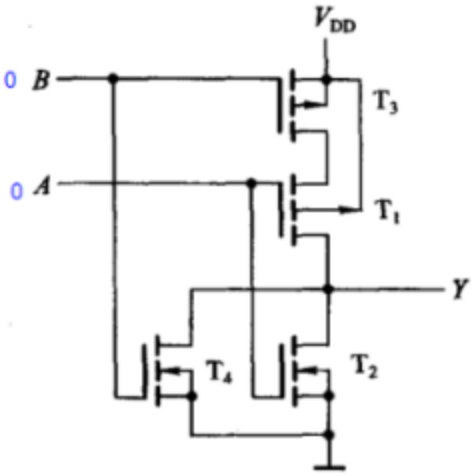


图 3.3.28 CMOS 或非门

19 状态机三段式的区别

一段式：最主要的特征是只有一个 always 块。在这种状态机的写法中，组合逻辑电路和时序逻辑电路都在一起，没有分开；因此这种写法增加了代码的复杂度且不利于代码的维护和修改，同时也不利于后期约束。
两段式：最主要的特征是有两个 always 块，将组合逻辑和时序逻辑分开。其中一个 always 块采用同步时序描述状态转移，而另一个 always 块采用组合逻辑来判断状态转移的条件，描述状态转移规律及输出。
三段式：最主要的特征是有三个 always 块。区别于两段式状态机的关键在于两段式状态机直接采用组合逻辑输出，而三段式状态机则通过在组合逻辑后再增加一级寄存器实现逻辑输出—即一个 always 块采用同步时序描述状态转移，一个 always 块采用组合逻辑判断转移条件、转移状态规律，最后一个 always 块采用同步时序描述状态的输出；
二段或三段式状态机的写法代码非常清晰，降低了编写维护代码的复杂度，清晰完整的显示出状态机的结构。其中三段式可以有效地滤除两段式状态机组合逻辑输出可能产生的毛刺信号。

20 reg 和 wire 的区别

① Wire 相当于物理连线，即输入有变化，输出马上反映；reg 相当于存储单元，一定要有触发，输出才会反映输入的状态。
② Wire 使用在连续赋值语句中，如 assign；而 reg 使用在过程语句中，如 initial、Always。

21 initial 与 always 的区别

initial 只能用于测试，且只执行一次，不可综合；always 过程块是用来描述硬件时序和组合电路，是可综合的。

22 什么是 critical path？如何处理？

关键路径通常是指同步逻辑电路中，组合逻辑时延最大的路径。关键路径是对设计性能起决定性影响的时序路径。

四 验证部分

1IC 设计的流程及使用工具

前端设计主要流程

（1）规格制定

客户向芯片设计公司（称为 Fabless，无晶圆设计公司）提出的设计要求，包括芯片需要达到的具体功能和性能方面的要求。

（2）系统级设计

用系统建模语言（如 matlab，c 等）对各个模块进行描述，为了对方案的可行性进行验证。

（3）RTL 设计

使用硬件描述语言（VHDL，Verilog HDL）将模块功能以代码来描述实现。

（4）仿真验证

仿真验证就是检验编码设计的正确性，看设计是否精确地满足了规格中的所有要求。设计和仿真验证是反复迭代的过程，直到验证结果显示完全符合规格标准。仿真验证工具 Mentor 公司的 Modelsim，Synopsys 的 VCS。该部分称为前仿真，接下来逻辑部分综合之后再一次进行的仿真可称为后仿真。

（5）逻辑综合

仿真验证通过，进行逻辑综合。逻辑综合的结果就是把设计实现的 HDL 代码翻译成门级网表 netlist。综合需要设定约束条件，就是你希望综合出来的电路在面积，时序等目标参数上达到的标准。逻辑综合需要基于特定的综合库，不同的库中，门电路基本标准单元的面积，时序参数是不一样的。所以，选用的综合库不一样，综合出来的电路在时序，面积上是有差异的。逻辑综合工具 **Synopsys 的 Design Compiler。**

（6）STA 静态时序分析

它主要是在时序上对电路进行验证，检查电路是否存在建立时间（setup time）和保持时间（hold time）的违例。STA 工具有 **Synopsys 的 Prime Time。**

（7）形式验证

这也是验证范畴，它是从功能上（STA 是时序上）对综合后的网表进行验证。常用的就是等价性检查方法，以功能验证后的 HDL 设计为参考，对比综合后的网表功能，他们是否在功能上存在等价性。这样做是为了保证在逻辑综合过程中没有改变原先 HDL 描述的电路功能。形式验证工具有 **Synopsys 的 Formality。**

注：（6）（7）属于后仿，对综合后的门级网表进行验证

后端设计流程

（1）DFT

Design ForTest，可测性设计。芯片内部往往都自带测试电路，DFT 的目的就是在设计的时候就考虑将来的测试。DFT 的常见方法就是，在设计中插入扫描链，将非扫描单元（如寄存器）变为扫描单元。DFT 工具 Synopsys 的 DFT Compiler。

（2）布局规划

布局规划就是放置芯片的宏单元模块，在总体上确定各种功能电路的摆放位置，如 IP 模块，RAM，I/O 引脚等等。布局规划能直接影响芯片最终的面积。

工具为 Synopsys 的 Astro。

（3）时钟树综合 CTS（Clock Tree Synthesis）

时钟树综合，就是时钟的布线。由于时钟信号在数字芯片的全局指挥作用，它的分布应该是对称式的连到各个寄存器单元，从而使时钟从同一个时钟源到达各个寄存器时，时钟延迟差异最小。CTS 工具为 Synopsys 的 Physical Compiler。

（4）布线(Place & Route)

这里的布线就是普通信号布线了，包括各种标准单元（基本逻辑门电路）之间的走线。比如我们平常听到的 0.13um 工艺，或者说 90nm 工艺，实际上就是这里金属布线可以达到的最小宽度，从微观上看就是 MOS 管的沟道长度。

布线工具为 Synopsys 的 Astro。

（5）寄生参数提取

由于导线本身存在的电阻,相邻导线之间的互感,耦合电容在芯片内部会产生信号噪声，串扰和反射。这些效应会产生信号完整性问题，导致信号电压波动和变化，如果严重就会导致信号失真错误。工具 Synopsys 的 Star-RCXT。

（6）版图物理验证

对完成布线的物理版图进行功能和时序上的验证。

LVS（Layout Vs Schematic）：就是版图与逻辑综合后的门级电路图的对比验证；

DRC（Design Rule Checking）：设计规则检查，检查连线间距，连线宽度等是否满足工艺要求

ERC（Electrical Rule Checking）：电气规则检查，检查短路和开路等电气规则违例；

工具为 Synopsys 的 Hercules。

（7）流片（Tape out）：在所有检查和验证都正确无误的情况下把最后的物理版图交给 Foundry，在晶圆硅片上做出实际的电路，再进行封装和测试，就得到了可以使用的芯片。

2 形式验证

形式验证分为两种，一种是等价性验证，是指从功能上对综合后的网表进行验证，以功能验证后的 HDL 设计为参考，对比综合后的网表功能，检验是否在功能在存在等价性，保证综合后没有改变原先 HDL 描述的功能，另一种是模型检查，是从数学上完备的验证电路的实现方案确实实现了电路设计所描述的功能，建立数学模型进行检查设计。

3 覆盖率驱动验证

以覆盖率的高低作为衡量验证完备性的指标。在验证计划中会指定具体的覆盖率目标，通过覆盖率验证可以确定验证是否达到要求。当然，达到目标覆盖率并不意味着验证就通过了，因为功能覆盖率是由人为定义的，有时候即便达到 100%，也未必将所有的功能场景全部覆盖了，因为人为主观定义的功能场景有时候可能存在遗漏，所以还需要对测试用例进行迭代。

4 验证的完备性

首先验证不能保证百分百完备，但我们应该朝这个方向去努力，所以要通过多种验证方法尽

可能减少潜在的风险，这主要包括 ip 级验证、子系统级验证、soc 级验证，除此之外，还有 upf 验证， fpga 原型验证等。
从前端 rtl 验证角度看，验证的完备主要包括以下四个方面：
（1）有关设计 spec 功能全部验证到
（2）所编写的 testcase 全部 pass
（3）覆盖率达到要求
（4）接口完备性确认，保证所有的接口时序都已覆盖，包括正常时序及异常时序
芯片后端也会做一些检查，如时序检查，DFT，形式验证。

5 soc 验证与 ip 验证的区别/系统级验证与模块级验证的区别

模块级：验证更关注模块内部的功能，如寄存器，数据通路，数据结构，协议信号等等细节，工作量较大。
系统级：检查模块顶层的接口，模块与模块之间的信号连接以及模块组合后的整体功能。

6 验证步骤（当你拿到一个项目（dut 待测设计）时，你如何开展验证活动）

（1）首先我要去查看 DUT 的设计文档，弄清楚 DUT 的功能与接口的时序，提取功能点
（2）画出验证框架图，开发底层组件
（3）根据细化的功能点编写测试用例，先编写一个 testcase 以及一些基础的 sequence，确保验证环境能够跑通，然后再根据其他的测试点编写新的测试用例。
（4）进行大规模随机测试，收集代码和功能覆盖率
（5）查看覆盖率报告，分析哪些还没有覆盖，针对未覆盖的地方编写定向测试以及更换不同的种子继续跑
（6）回归测试，将之前的的测试用例再同时跑一遍，在极端情况下找出设计或者验证环境的 bug。
（7）撰写验证报告，在公司里那么就要进行各种开会讨论，各种 review。
主要结合具体项目来说，比如具体项目的功能点有什么，你又是如何针对功能点编写 testcase 的，最终收集的代码富覆盖率和功能覆盖率是否都达到了 100%，收集覆盖率的语句又是在 testbench 中的哪些组件中写的等等。

7 仿真过程中遇到问题如何定位/scoreboard 数据如果出错应该怎样处理

scb 中数据比对错误地方肯定会打印出来，比如 iop 项目中比对的 ch 和 data，把不一致的 ch，data 打印出来，然后从波形中找出对应的 ch 和 data，然后检查硬件是不是应该这样处理，如果是硬件处理错误就可以将该问题交给设计人员处理

8 为什么选择验证岗位（为什么要转行）

（1）芯片国产替代化已经是大势所趋，因此我觉得 IC 的发展前景更加广阔，更加利于我今后的发展。
（2）我比较喜欢验证的工作内容。验证人员需要理解待测设计，提取功能点，搭建验证环境，大规模随机测试，debug，回归迭代，收集覆盖率，整个过程走下来我会觉得很有成就感，比较享受验证过程中遇到问题，解决问题的过程，尤其环境跑通了的时候。
（3）我觉得我的性格比较适合做验证的工作。我性格属于不是很喜欢社交，喜欢一门心思

搞技术，喜欢钻研，而验证的工作量很大，很繁琐，这就要求验证人员细心且耐心，不急躁，思考问题要全面。
（4）最后就是 IC 的薪资待遇要高于行业平均水平，我出身农村家庭，从事验证工作能更好的缓解家庭压力，也能为公司创造更大的价值。

9 测试用例怎么写

主要是写 sequence，然后在 body 里根据测试点要求写相应的激励，通过 scoreboard 判断 dut 功能是否正确实现。

10 编写 testcase 的基本思路和顺序

这个问题也是面试必问考题。首先是基本测试用例，该用例用来把 tb.sv 启动，创建 TLM 拓扑结构等，确保 tb.env 没有问题。随后是直接测试用例，验证 RESET，RAL，FSM，数据读写，算法，固定值测试等。最后是随机测试用例，考虑可以随机的测试。

11 前仿和后仿的概念，前仿可以检查亚稳态吗

前仿真
前仿真也叫 RTL 级仿真。通过仿真器如 modelsim 或 VCS 验证电路逻辑功能是否正确，即 HDL 描述是否符合设计所定义的功能期望。在前仿真过程中，通常与具体的电路版图实现无关，即没有时序延时等信息。
后仿真
后仿也叫门级仿真，是基于布局布线之后的模型进行仿真，可以考虑到真实的信号延迟，负载，电源电压变化等，后仿的目的是在考虑了这些实际因素之后，验证设计是否仍然满足要求，特别是时序要求。后仿会产生 SDF 文件。

12 验证无法检查 DUT 的哪些问题

dut 的数据处理效率问题检测不出来的，如 dut 中可以插入多个 fifo 提高效率，一个 buffer 就处理一个数据，处理完才能接收下一个数据再处理，两个 buffer 的话，第二个数据可放在第二个 buffer 里存储，等第一个 buffer 处理完第一个数据立马就可以进行切换，节省了时间。
其他的如建立保持时间，亚稳态，竞争冒险等等都检查不到

13 验证过程中遇到哪些问题，怎么解决的

刚开始的时候最大的困难就是搭建一个能跑通的验证平台，看懂和自己动手去写是两码事，搭建的过程中会遇到很多错误，通过查看 log 然后不断地去 debug，完善验证平台，使其尽可能地配置更加灵活。比如在 iop 项目里有很多个 agent，如何例化、连接，scb 中需要那么多端口，还要要避免方法名重复，这些问题刚开始思考的时候都比较困难。
其次就是对于时序的理解，时序一旦理解出错了那么在 driver 驱动数据和 monitor 收集数据的时候都会出错，scb 中比对的时候就会出错。其中 kpe 模块中没有时序，因此当时不知道 monitor 该怎么收集数据，最后通过 uvm_event 这一事件来进行同步的。
最后就是收集尽可能高的覆盖率，如何针对未被覆盖的地方写新的 case 都需要自己慢慢分析思考。

比如在一家公司实习的时候，参考模型是由同事使用 matlab 写的，matlab 如何与验证环境做适配就是一个问题，后来将 matlab 写的程序处理后的数据输出在一个文档里，scb 再从文档里读取数据。

14 对网表的理解

网表（netlist）是用于描述电路元件相互之间连接关系的，一般来说是一个遵循某种比较简单的标记语法的文本文件。

15 灰盒/黑盒/白盒验证

灰盒指的是验证对象的内部结构，只有一部分是可见的。

黑盒验证则内部完全不可见，我们只能看到设计的输入接口和输出接口，对黑盒验证，我们只能通过设计文档来了解它的功能。

白盒指的是验证对象的内部结构是完全可见的，我们可以清楚的看到设计的详细内容，白盒验证的好处是我们可以了解设计者的意图，并且验证可以达到设计上的每一点，但这需要花费更长的时间。

16 设计与验证的关系

17 为什么采样随机化的验证策略

使用随机化产生激励，可以很容易的在短时间内产生大量的随机激励，对于较大的验证空间，这些随机激励可以达到我们工程师可能会忽略的地方。相对于固定的激励（难以覆盖较大的可激励空间），随机化的激励可以简化代码编写，更能验证出 DUT 可能隐藏的错误，提高验证的完备性

18 对验证的理解

验证的职责就是确保设计和预定的期望一致，在流片之前要发现所有的问题。

第一步，验证人员首先会阅读各种设计文档以及相关协议，基于自己对芯片规格和设计方案的理解，撰写验证文档，其中包括验证框架以及分解的测试点。

第二步，基于 c/c++/sv 等语言开发组件，搭建验证环境。

第三步，根据测试点编写测试用例。

第四步，仿真和 debug，通过查看 log 结果或仿真波形来找出 rtl 以及验证环境的 bug，确保验证环境没有问题的前期下与设计人员沟通 rtl 存在的问题。

第五步，回归分析以及收集覆盖率。当所有的 case 都 pass 后，再将所有的 case 同时跑一边，在不断重跑的情况下找出极端情况下的 bug。收集覆盖率主要是代码覆盖率和功能覆盖率，基于覆盖率的高低对测试用例进行迭代。

第六步，撰写验证报告以及开各种会议，各种 review，各种讨论。

19 项目中 DUT 的 bug

iop：error 和 empty 信号，检测出来硬件没有开放这两个功能，设计上也不要求实现这两个功能，所以也没有去修改。

Uart：设计里面没有给寄存器做复位的操作，通过复位之后的读写测试发现，复位之后不

能读出 0。

20 验证人员具备的品质

(1)对任何东西都要有质疑的态度，精益求精，如对覆盖率要力求尽量达到最高
(2)不但精通验证，对于设计以及后端也要熟悉
(3)对自己不懂得问题要虚心求教，不能模棱两可

21 分解测试点的基本原则与方法

(1)测试点实际上是把设计的功能按层级分解成一个个最简单、最底层的功能点，化繁为简，方便测试用例的实现。测试点主要**从功能规格与架构规格中**提取。测试点分解需要保证的几点原则：

完备性：即不能遗漏任何功能点，特别是异常处理，边界处理，容错处理这些往往容易被忽视；

低耦合：不同测试点之间的相关性越低越好，这也直接决定了分解粒度，并影响 testcase 的开发难度；

无歧义：测试点的描述要直接而明确，不同测试点之间不存在矛盾之处。

扩展性：包含异常和边界特性。

测试点的提取可以从以下几个方面考虑：

a. 时钟功能
b. 复位功能
c. 接口时序
d. 数据通路
e. 异常处理
f. 典型电路测试点
g. DFX 测试点
h. 性能测试点
i. 压力测试点

(2) 功能测试点分解方法

A、等价类划分法

将所有可能的输入数据(有效的无效的)划分成若干个等价类；

例:参数 A 表示逻辑一轮处理的循环次数范围在[100,1000]，利用等价类划分可分为[100,200]，[200,300]……[900,1000];

B、边界值分析法

对输入的边界条件进行分析，得出边界值测试点(包含正常的异常的)；

例:配置寄存器 A 的正常配置范围是[3,8]，那么利用边界值法可分为 1,2,3,4,7,8,9,10 几个配置值；

C、正交矩阵法

对输入的各个条件正交，筛选得出可靠的测试点；

例:输入参数 A 有开启、关闭两种情况，参数 B 有模式 1 和模式 2 两种，参数 C 位宽选择有 1bit，8bit 和 64bit，利用正交矩阵分解就有 12 种情况：

A 开启、B 为模式 1、C 位宽选择 1bit；
A 开启、B 为模式 1、C 位宽选择 8bit；
A 开启、B 为模式 1、C 位宽选择 64bit；
A 开启、B 为模式 2、C 位宽选择 1bit；
A 开启、B 为模式 2、C 位宽选择 8bit；
A 开启、B 为模式 2、C 位宽选择 64bit；
A 关闭、B 为模式 1、C 位宽选择 1bit；
A 关闭、B 为模式 1、C 位宽选择 8bit；
A 关闭、B 为模式 1、C 位宽选择 64bit；
A 关闭、B 为模式 2、C 位宽选择 1bit；
A 关闭、B 为模式 2、C 位宽选择 8bit；
A 关闭、B 为模式 2、C 位宽选择 64bit；

D、错误推断法
推测法主要依赖于经验、直觉来做出简单的判断甚至是猜测，给出可能存在缺陷的条件，场景等，生产对应的测试点；
例:可能有些设计人员没考虑全反压的情况，在测试点分解时可增加接口上的反压测试，可通过 force 一定时间的 full 或 afull 信号，观察逻辑处理情况；

F、其他方法
在测试点分解后期需要加入适当的随机测试点，这时候可能覆盖的范围会比较广泛，不符合测试点的定义，但是这里没关系，后续的随机测试点是为了将多种情况混合起来测试，也是一种测试场景；
(3) 测试点与测试用例的关系：
测试用例是根据测试点写的,是用来覆盖测试点的。一个测试用例可以覆盖多个测试点，在考虑用例复杂度和仿真时间的前提下单个测试用例应覆盖尽量多的测试点；
单个测试点在一个用例中必须被测试覆盖，不可出现多个测试用例测试通过后，某个测试点才覆盖的情况。如果有，必定是测试点分解太粗，需要重新细化该测试点；
总结：一个测试用例可以包含多个测试点，一个测试点不能被分解到被多个测试用例包含(但一个测试点是可以在多个用例中被包含的)。

22 验证工程师与设计工程师什么关系

设计工程师基于 verilog 语言编写 rtl 代码实现设计文档的功能点，随后将代码交付给验证工程师,验证工程师基于 C/C++/SV 语言搭建验证环境来检测 rtl 设计代码是否正确实现了设计文档的功能点要求，验证工程师通过仿真结果会将 rtl 代码存在的 bug 反馈给设计工程师。

23 验证工程师还要与哪些别的工程师协作，具体是协作什么

(1)和架构工程师沟通，去了解更多芯片的整体架构和功能点；
(2)与 DFT 工程师交流，因为他们也需要做测试和验证，很多时候他们的验证平台就是从验证工程师的验证平台移植过去的；
(3)与 FPGA 工程师交流，他们会复用很多 IC 验证工程师的测试用例，复现一些错误现场。
(4)如果也负责后仿的话，那还经常需要和后端工程师交流，因为后仿的网表是他们提供的；

(5)如果只负责 IP level 的验证，那基本只要和前端设计工程师打交道就可以了。

24 对于随机测试定向测试认识

定向测试可以找出设计中预期的漏洞，随机测试覆盖范围往往比定向测试覆盖范围大，能够找出预期之外的漏洞。随机测试覆盖不到的地方，需要编写一些定向测试。当使用随机激励时，需要用功能覆盖率来评估验证的进展情况。

25 如果代码覆盖率 95%，功能覆盖率 80%，test pass100%，请问接下来怎么让覆盖率收敛？

针对 rtl 代码没运行的部分以及功能覆盖率未采样到的部分编写新的测试用例，可以新增约束或者使用定向测试以使覆盖率收敛。

26 回归测试的作用

将所有的测试用例不断重复地跑，从而进一步发现 dut 中隐藏的错误，提高覆盖率。

27 debug 的手段有哪些

打印显示；设置断点；波形分析

28 VCS 的一个编译过程

```
com:
    vcs -full64 -sverilog +v2k -debug_all -timescale=1ns/1ns -ntb_opts uvm-1.1 \
    -l comp.log \
    -f ./filelist.f \
    -P ${NOVAS_HOME}/share/PLI/VCS/LINUX64/novas.tab ${NOVAS_HOME}/share/PLI/VCS/LINUX64/pli.a \
    tb.sv
```

解释：
-full64：以 64 位模式编译
-sverilog：支持 sv 语法
+v2k：支持 verilog 2001 语法
-debug_all:使能所有的 debug 调试功能
-timescale=1ns/1ns：仿真时间单位和精度
-ntb_opts uvm-1.1：加载 UVM 库文件
-l comp.log：仿真信息输出到 log 文件
-f ./filelist.f：指定一个包含所有文件路径的 filelist 文件
-P：指定加载的表格文件和静态库
tb.sv：项目的顶层文件

29 如何在前期保证提取的功能点完备

要充分理解设计文档，然后根据设计文档提取出功能点，遵循测试点分解的常规方法细化出测试点，包括正常的，边界以及异常的地方。后续再根据仿真结果以及覆盖率报告进行完善。

30 随机种子的理解

产生随机数的过程中，如果不改变随机数种子，每次程序从头开始运行的过程中都会产生相同的随机数序列。对于 IC 验证来说，如果不改变 seed 的值，则每次 run 仿真时，仍旧会产生相同的激励数据。为了解决这个问题，可以在编译时加上：

```
+ntb_random_seed_automatic
```

31 如果仿真卡住，但是时间一直在往前推行，可能是什么原因？（只考虑验证环境的错误）

如果事件触发晚于等待@，就会一直阻塞住，代码没法执行；或者用了一些阻塞函数，比如 get()从一个空的 mailbox 拿 data，put()向满的 mailbox 放数据。

32 如果你的条件覆盖率(condition coverage)没有达到 100%，出了什么问题？怎么解决？

条件覆盖率主要是衡量每个子表达式结果是否取到 true 或 false。未达到 100%，即是判断语句中有条件表达式的结果取值没有取到。解决办法：通过查看覆盖率报告定位到 rtl 代码中，查看 rtl 代码，通过新增约束或者定向测试使判断语句中的子表达式都能判断到（同时适用于如何提高代码覆盖率，查看→定位→想办法覆盖）。

33 相较于科班的优势？

第一，我具备更强的自学能力和更坚定的决心，能够不断学习和探索新的知识和技能。我转 IC 的那段时间，下定决心，经常学到半夜一两点，只为了转行成功的这个目标。
第二，我具备更广泛的知识背景和多元化的学习经历，这样会使得我更能快速适应工作环境以及应对各种工作上的挑战。因此综合素质会比科班更强。

34 长这么大遇到的最大的困难？怎么解决？

我本科的时候本来有希望保研，没想到十月份得知不能保研，当时心情非常低谷，经过短暂的几天平复心情，下定决心考研，最后三个月挑灯夜战，每天学习 12 个小时，卸载所有社交娱乐软件，最终成功上岸。这段经历挺难忘的。

由于今年经济环境不好，很多公司裁员，缩招，因此秋招非常艰难，而我遇到的最大的困难就是这段时间会比较焦虑，看到别人收获了 offer 的时候，整个夜晚都会焦虑，担心毕业即失业，遇到这种情况我就会放空自己，我会去黄浦区后街那里，沿着黄浦江一侧散步，看着上海的夜景，这样能平复内心的焦虑，然后回来该学习依然学习，机会只会留给有准备的人。

35 绩点，排名，论文

一篇中文核心，一篇 SCI 二区已经被接收了

36 仿真效率比较低，通过哪些编码方式可以提高仿真效率？

1 循环条件不要带计算，要不然每次循环都要计算一次
2 能条件成立后才进行的计算，就不要着急放到前面，因为一旦条件不成立，这个计算就执行不到。
3 连接处 logic 的语义显示声明成 wire，可以加快仿真速度
4 尽量用 ref，少传递复杂数据结构
5 不要滥用动态数据结构

37 职业规划

我认为在未来的 5 年内，应该脚踏实地的不断积累技术，丰富自己的验证经验，在团队里保持谦虚的态度和前辈们学习，打好基础。在 5-10 年内，我希望自己可以成为团队的核心成员，在大型项目里担任重要职责，同时对芯片的其他流程也去学习了解，比如芯片设计和芯片后端，成为一个发展全面的芯片行业从事者。

五 总线问题

1 AHB/APB 信号有哪些?

AHB:clk,rstn,addr,pwrite,wdata,rdata,trans,burst,sel,port,size,resp,ready(13 个)

APB:clk,rstn,addr,sel,pwrite,wdata,rdata,enable,ready,slverr(10 个)

AXI:比较重要且独有的 last,valid,len

AXI:

Size: 突发传输中每个 transaction 的大小　　单位: 字节 byte

Burst: 突发类型，固定长度突发/增量突发/回环突发

Len: 突发长度，一次突发传输中 transaction 的个数

Last: 突发传输的最后一次传输（最后一个 transaction）

ID:每笔传输的 trans ID 号

Valid: 表示读/写数据和选通信号有效

2 AHB/APB/AXI 的区别

① AHB 是先进的高性能总线，AXI 是先进的可扩展接口，APB 是用于低带宽且不需要流水线操作的高性能外围设备；
② AHB 和 APB 都是单通道总线，不支持读写并行；而 AXI 是多通道总线，总共分为五个通道，能够实现读写并行；
③ AHB 和 AXI 都是多主/从设备，拥有仲裁机制；而 APB 是单主设备多从设备，不具有仲裁机制；
④ 在数据操作方面，AHB 和 AXI 支持突发传输，APB 不支持；此外，AXI 支持数据的非对齐操作，AHB 不支持。
⑤ 三者支持的最大总线宽度不同，AXI 最大支持 1024，AHB 最大支持 256，APB 最大支持 32

AHB/APB 的区别

① AHB 是先进的高性能总线，APB 用于**低带宽**且**不需要流水线总线接口**的**高性能**的外围设备。
② AHB 支持二级流水线，APB 不支持
③ AHB 拥有仲裁机制，APB 没有
④ AHB 支持多主/从设备，而 APB 是单主设备多从设备
⑤ AHB 支持突发传输，APB 不支持
⑥ 最大总线宽度不同，AHB 最大支持 256，APB 最大支持 32

3 APB2.0/3.0/4.0 区别

APB2.0 和 APB3.0 的差别： APB3.0 提供了一个低功耗的接口，并降低了接口的复杂性。且 APB3 比 APB2 增加了两个信号：
PREADY：来扩展 APB 传输，主要是增加延时；
错误信号 PSLVERR：来指示传输失败

APB3.0 和 APB4.0 的差别： 增加了 PROT 和 PRSTB 两个信号。
PPROT 一种保护信号，用于支持 APB 上的非安全交易和安全交易。
PSTRB 一个写选通信号。

4 AHB 的拆分事物是什么意思

slave 的 split 传输，当某个 master 和 slave 进行传输的时候，占用总线所有权，但是占着地方不办事儿，这个时候就会让 slave 的响应回复一下 split 信号，结束这个 master 对 slave 的访问换到别的 master 访问这个 slave，并且这个信号给仲裁器之后会将这个 master 对这个 slave 的优先级变为最低且一直是低，直到对这个 slave 的数据访问结束了，才会复原到最初的 master 优先级。

5 AXI 通道有几个，每个是做什么的

5 个通道
读和写地址通道：用于传输一次数据所需的地址和控制信息。
读数据通道：从机向主机返回读数据和读响应信息。
写数据通道：主机向从机传输写数据
写响应通道：从机通过写响应通道来返回写响应信息。

6 AHB 的 trans 类型，burst 类型

trans：

HTRANS[1:0]	传输类型	Description
00	IDLE	主设备占用总线，但没进行传输 两次burst传输中间主设备可发IDLE 此时就算slave被使能，也不会从总线上获取任何的数据信号。如果此时salve被选中，那么每一个IDLE周期slave都要通过HRESP[1:0]返回一个OKAY响应
01	BUSY	主设备占用总线，但是在burst传输过程中还没有准备好进行下一次传输 一次burst传输中间主设备可发BUSY 这时slave不会从总线上收取数据而是等待，并且通过HRESP[1:0]返回一个OKAY响应。需要注意的是，这个transfer需要给出下一拍的地址和控制信号，尽管slave不会去采样。
10	NONSEQ	表明一次单个数据的传输或者一次burst传输的第一个数据 地址和控制信号与上一次传输无关
11	SEQ	burst传输接下来的数据 地址和上一次传输的地址是相关的，这时总线上的控制信号应当与之前的保持一致，地址视情况递增或者回环。

burst:
递增传送:依上一笔的地址来递增
回环传送:如:回环长度=4;每 4 个字节要对齐在 16 字节的范围内。如果第一个地址=0x64，则传送的顺序为 0x68、0x6C、0x60。

7 如果验总线的话有哪些功能点需要验

读写通道正确；相关信号有效。

8 AXI 的 4k 边界是什么

所谓的 4K 边界是指低 12bit 为 0 的地址，例如 32'h00001000, 32'h00002000... 这些特殊的地址我们称之为 4k 边界；同理 1k 边界是指低 10bit 为 0 的地址，例如 32'h00000400,32'h00000800...
AXI 系统中，slave 地址空间一般为 4KB 的整数倍。
如: 32'ha100_1000, 32'ha100_2000, 32'ha100_3000
为什么不能跨越 4k 边界?
AXI 协议会在读/写地址通道的开头发出 addr/len/size 等信息，若一笔 burst 跨越了 A 和 B 两个 slave，则会只有 A 收到开头的 addr/len/size 等信息，B 则收不到，从而造成 burst 无法完成。

9 AXI 的乱序访问(out of order)

当总线上有多个事务时，若不同事物的 ID 一致，必须顺序完成；不同事物的 ID 不一致，则可以乱序。

10 AHB 发送一个 trans 最少几个周期

2 个

11 AXI3 与 AXI4 区别

axlen:
axi3 burst length最大为 16。
axi4 burst length最大为 256。

axi4取消 wid:
axi4取消 wid，则写通道没有 out of order 乱序和 interleave 间插功能，所有写数据都是 order 有序的，这样做的目的是乱序会带来死锁问题和严重的 buffer 资源浪费。
QoS 信号：
AxQoS 是 axi4 增加的用户服务质量信号。

12 AXI outstanding

outstanding 是指主机在没有收到 response 时可以发起多个读写 transaction 的能力

13 为什么 APB 发送数据需要两个周期

因为 APB 用在低速设备的外围，不需要像 AHB 那样两级流水，即在数据的传输周期同时可在总线上发送第二个数据的地址。

14 AHB 发送一个 single 数据的时候，如果 HREADY 信号拉低 3 个周期，一共需要几拍？

4 个

15 AHB 为什么不要跨 1k，如何避免越界访问

Burst 传输不能跨越 1kb 边界，因为 slave 的地址空间都是以 1kb 为单位的，这样做是为了让一个单独的 burst 不能访问多个 slave，避免的方式是：当在 trans 信号在 1kb 地址边界时，改成 NON_SEQ，重新发起一次 burst 传输。

16 axi 的 outsatanding 和 interleaving 的区别？

outsatanding 是对地址而言，一次 burst 还没结束，就可以发送下一 burst 地址，这样可以大大提高处理 transaction 的效率。而 out-of-order 和 interleaving 则是相对于 transaction，out-of-order 说的是发送 transaction 和接收的 cmd 之间的顺序没有关系，如先接到 A 的 cmd，再接到 B 的 cmd，则可以先发 B 的 data，再发 A 的 data；interleaving 指的是 A 的 data 和 B 的 data 可以交错，如 A1 B1 A2 B2 B3……（同一个事务之间的不同数据要按顺序，例如不能出现 A1 B2 A2 B1...）

17 为什么 AXI 没有读响应通道？

因为读响应信息 resp 可以作为读数据通道的一部分传递。

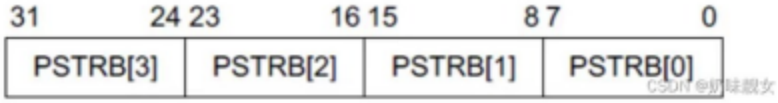
18 介绍一下 axi

AXI 是一种面向高性能、高带宽、低延迟的片内总线，它具有如下的几个特点：
（1）总线的地址/控制和数据通道是分离的；
（2）支持非对齐的数据传输；
（3）支持突发传输；

（4）具有分离的读/写数据通道；
（5）支持乱序访问；
（6）更加容易进行时序收敛。

19 apb 的 PSTRB 信号

写选通信号，指示数据总线上那几个字节数据有效。由于数据位最大为 32bit，即 4 个字节。因此 PSTRB 位宽为 4 位，每一位对应一个字节，设置为 1 时，指示该字节数据有效。需注意，在读操作中，PSTRB 所有位必须设置为低电平。
当置为高电平时，写选通脉冲指示写数据总线的相应字节通道包含有效信息。
写数据总线的每八位有一个写选通脉冲，因此 PSTRB [n] 对应于 PWDATA [(8n + 7) : (8n)]



六 项目问题

1 验证环境如何搭建的

2 transaction/item 包含哪些

3 功能点如何提取

4 xxx 项目中有哪些测试点

5 DUT 的功能

6 覆盖率怎么收集，用的什么仿真软件，哪个指令，生成的文件格式是什么，多个覆盖率文件怎么合并

vcs，指令-cm，文件格式后缀为.vdb，合并用 merge

7 Virtual sequence 中有哪些东西

8 如何配寄存器/寄存器如何和 driver 做互动？

driver 按照时序协议向寄存器写数据

9 scoreboard 怎么写的，怎么比较数据的

10 断言检查了哪些时序

11 介绍一下时序

12 仿真环境的打印如何实现的

`uvm_info

13 写一个寄存器读一个寄存器好，还是全部写完再读好

全部写完再读好

14 RAL 中，如果 set 了，然后总线上一直 get 不到 transaction，是什么导致的，怎么解决

是不是 adapter 在事务信息转换的过程出错了

15 现有两个设计人员设计的两个相同功能的 dut，你如何判断哪个更好

考虑面积与功耗

16 数据流

17 VCS 编译选项

18 寄存器的复位和读写怎么测的

19 代码和功能覆盖率为什么没达到 100%

代码：可能 RTL 某些条件分支没有跑到
功能：一些关心的值没有采样到，可能是因为随机的次数少了

20 验证环境中哪些地方会用到约束

主要是 item 用到约束，比如对于一些数据保留位需要约束为 0，注入错误激励约束 error 为 1，寄存器的保留域约束为 0 等等

21 验证结构