数字验证重难点

SystemVerilog的重难点

数据类型

二值/四值逻辑

编码时一定要注意操作符左右两侧的符号类型要一致

二值逻辑：包括 bit, byte, shortint, int, longint等，均为有符号类型（bit除外）；这些类型的值只包含{0，1}，目的是模拟计算机验证环境，提高仿真性能，节约空间。若有四值逻辑数给其赋值，x,z会默认被赋值为0，因此二值逻辑数要远离DUT

四值逻辑：包括logic, integer, reg, wire等，均为无符号变量（integer除外）；目的是模拟外部物理世界

关于logic需要注意，不能被多驱动（如三态时），其他都可以当成wire用

静态数组、动态数组、关联数组、队列

静态数组是指其数组的大小在定义时被显性地指定。动态数组有一维或多维，其中非压缩部分的一维的大小在定义的时候未被指定，在使用前才被分配。关联数组允许通过任何类型的索引来访问数组的成员。队列被用于定义一个顺序的数据集合。

```
bit  s_array[5]//长度为5的静态数组，也可以写为[4:0]

int  d_array[];            //动态数组，compile时长度不确定，simulation时长度确定
d_array = new[20];     //new分配空间

int a_array[string key]; //类似于python的字典，稀疏存储

byte data_queue[$];//$符号代表的是无限
```

关于静态数组，SV相比verilog额外引入了压缩数组和非压缩数组的概念，主要是数据实际存储消耗空间的区别。

```
logic [3:0][7:0] data; // 2-D packed array包含4个8位的压缩数组

logic [7:0] data [3:0];//包含4个8位数据类型的非压缩数组
```

使用原理和其他语言一样，主要弄清这些之间的差异和使用场景。

结构体、联合体

结构体默认是变量类型，也可以申明为线网（wire）类型，对于线网结构体的所有成员必须都是四值逻辑的

结构体同样也有压缩和非压缩的概念。**默认情况下，结构体是非压缩的**。结构体成员是独立的变量或常量。使用packed显示的声明一个压缩结构体。压缩结构体按照指定的顺序以相邻的位来存储结构体成员。**压缩结构体被当做一个向量存储**，结构体的第一个成员在向量的最左边。向量的最低位是结构体最后一个成员最低位，其位编号为bit 0。如下所示（类似**小端模式**）：

```
struct packed {
  logic valid;
  logic [ 7:0] tag;
  logic [31:0] data;
} data_word;
```

packed structure的成员可以通过成员名引用（<struct_name>.<mem_name>）也可以使用结构体向量的相应位来引用（<struct_name>[M:N]）。

联合体（union）

- 联合体只存储一个元素，但该元素可以有**多种表示方法**，每种表示方法可以是不同数据类型
- Union声明及成员引用方法同struct，关键词为**union**。
- Union内的成员公用同一存储空间。所以对其中一个成员赋值，其他成员也会相应变化，只是数据类型不同而已

压缩联合体（可综合）中每个成员位数相同，**只存储整数值**。（一般用于存signed,unsigned两种类型，比如，当需要存signed类型时，给ele0赋值，需要存unsigned类型时，给ele1赋值）

```
typedef struct packed {
  logic [15:0] source_address;
  logic [15:0] destination_address;
  logic [23:0] data;
  logic [ 7:0] opcode;
} data_packet_t;
union packed {
  data_packet_t packet; //压缩结构体
  logic [7:0][7:0] bytes; // 压缩数组
} dreg;
```

该例中，值可以使用byte格式的数组写入，然后以data_packet格式读出相同的值(共享空间)。

结构体联合体常用于数据打包传输方便，不管在设计还是验证中都很常用。

typedef的用处

自定义任意数据类型

typedef reg [15:0] opreg_t; //创建新的类型opreg_t等价于reg [15:0]

typedef enum {WRITE=0,READ=1} WR //枚举类型

上面等价于：int map[string][$]; //map是一个关联数组，下标元素是string，存储元素是int[$]

自定义结构体可以在模块，接口或者包中定义

typedef struct { //结构体定义，不分配存储器

 logic [31:0] a, b;

 logic [7:0] opcode;

 logic [23:0] address;

} instruction_word_t;

instruction_word_t IW; // 结构体实例化时分配存储区

类型预定义

typedef class B;//typedef B

class A;

  B b;

  int xx;

endclass

class B;

  A a;

  logic tmp;

endclass

如上在定义class A时，使用了class B，但B还没有定义（工具按代码顺序编译），可以在前面先typedef class B.也可以直接用typedef B，表示后面会有B的定义。

编程结构

不管是设计还是验证，都是基于层次化的结构，verilog只有module的例化，而SV的数据“对象”更加丰富一些，会有不同的数据对象的实例化，来构成层次化结构。

Module和program

Module指的是硬件世界，program指的是软件世界，通常都作为比较顶层的容器。Program一般用的少，只需要注意一些点：

可在module直接被例化（硬件世界里可以包括软件世界），但是里面不能出现和硬件相关的过程语句和实例（如always，module等，软件世界里不能包括硬件世界），也不能由其他program语句，可以

有initial，task等块。program**内部定义的变量赋值的方式应该采用阻塞赋值**（软件方式），在驱动外部的硬件信号时应该使用非阻塞赋值（硬件方式）。$exit()强制结束该系统函数所在的program。

Interface

Interface是沟通软硬件世界的桥梁，即它既可以当成module这样的实体（内部可以有过程语句和连续赋值语句等），也可以当成class这样的数据类型。

从基本功能上看，他就是多个信号线的集合，或者说一捆信号，方便这些信号的同步（clocking块）和连线（modport块）。

过程块

不管什么样的结构容器内，都是由过程（执行）语句和申明语句组成的。

logic signed [7:0] a;//申明语句

a = -11;//执行语句

执行语句除了连续赋值语句(assign)之外，其它主要由过程块组成。其中包括initial块、always块的变型(always_comb, always_ff, always_latch)。SystemVerilog还增加了finial块。initial块是在程序开始的时候运行，final块是在程序结束前，所有执行进程结束后才执行。always块和其变型在其敏感条件满足的时候开始运行，在敏感条件不满足的时候被挂起。过程块可以是串行执行(begin...end)，也可以并行执行(fork...join、fork...join_any、fork...none)

fork join内部各子线程并行，直到所有子线程运行完毕才会进入下一个阶段

fork join_any内部各子线程并行，任意一个子线程运行结束就可以进入下一个阶段

fork join_none内部各子线程并行，无需等待直接进入下一个阶段

函数和任务

函数和任务的本质是执行语句（内部不能存在always等过程快），只是做了封装复用和参数的传递。Verilog中的函数和任务特点如下

	函数function	任务task
执行与消耗	不会消耗仿真时间，不能含有控制仿真时间的语句	可能会消耗仿真时间，可以包含仿真时间控制语句，如#100，时钟周期@（），wait（）以及事件event等语句
形参变量	函数中至少包含一个输入变量进行传参	任务中可以没有输入、输出变量
调用方式	function无法调用task，只能调用function	而task可以调用task,也可以调用function
返回值	一个可以返回数据的function只能返回一个单一数值	而任务或者void function不会返回数值，一般通过output信号直接输出。
返回方式	函数的返回可以通过调用return语句实现，当不需要返回值时可以将函数定义为void类型	任务没有返回值，但是也可以用return来结束task

在systemveilog中，允许函数调用任务，但是只能在由fork...join_none语句生成的线程中调用。同时也支持没有输入参数。方法的定义可以指定参数，input、output、inout及**ref**皆可。inout在开始的时候复制，在结束的时候输出；ref传递引用（句柄或者指针）；input： 在开始的时候复制值；output：在开始的时候复制值；如果为了避免外部传入的ref参数会被方法修改，则可以添加**const**修饰符，来表示变量是**只读变量**

数据生命周期

automactic：自动存储（相当于**局部变量**）　　**static**：静态存储（相当于**全局变量**）

类class中定义的任务和函数总是自动automatic的。 自动任务中声明的所有项都为每次调用动态分配。 所有形式参数和局部变量都存储在堆栈上。

在module、interface、program、package中定义的任务和函数默认为静态static的，所有声明的变量都是静态分配的。 如果一个程序是静态的，那么所有的子程序只能共享一个内存空间。子程序的每次执行都会覆盖之前子程序运行产生的结果。

面向对象

面向对象基本特点

封装：class作为一个容器将许多成员封装在一起，限制了作用域

继承：子类继承自父类，拥有父类所有的特点且在此基础上还可以扩展更多

多态：当子类和父类存在相同的方法，且“相同的“方法（名字，类型，参数都一样）表现出不同的态势，即重载虚函数

类的基本概念

类可以在SystemVerilog中的program/module/package内部或者外部定义，可以在program或者module内使用。关于类的几个名词

- 属性(property)：也就是类中定义的数据(变量)成员；
- 方法(method)：也就是类中定义的函数或者任务；
- 句柄(handle)：也就是指向对象的指针，即该对象的地址入口

构造函数：

在类中定义的new函数，可以加上一些初始操作。在别的地方new出某个对象时，会自动调用其构造函数，并分配一个空间。在SV中当不再有句柄指向某个对象时（或者把对象赋值为null），会自动施放掉该对象的内存空间（动态释放）。

成员类型：

所有成员默认为public，protected 关键字表示变量和方法只能在当前类或者其子类被访问到，而local就更加苛刻了，只允许在当前类的访问

This和super：

当父类和子类有相同名字的成员，子类里面调用该成员，会使用子类内的成员；若在子类里面想要使用父类的成员，此时就需要使用super来调用。super和this来指示作用域，若没有指引，则由近到远来引用。**函数内部局部变量>当前类成员变量>父类或更底层的变量**，但不会去追溯其子类。

虚（virtual）方法：

由于多态(父类和子类中有同名的方法)的存在，父类的句柄可以指向父类，也可以指向子类，但子类句柄不能指向父类对象。因为**父类句柄可以在子类对象中找到父类所有的属性和方法**，但无法通过父类句柄调用子类方法。这种在编译阶段就确定调用方法所处作用域的方式称为**静态绑定**，为了实现**动态绑定**需要定义虚方法，就可以**实现父类句柄调用子类方法**

对象复制

句柄直接赋值只是不同的句柄指向了同一个对象，并不是对象的复制。只有new才会创建新的对象。

Packet p1;
Packet p2;
p1=new；
p2=new p1；
最后一条语句是构造函数new的第二次执行，因此创建了一个新的对象，它内部的属性全部复制自p1的内容，这种方法的复制称作**浅复制**(shallow copy, 或称作浅拷贝)。所有的变量都被复制：整数、字符串、对象的句柄等。然而，其中包含的对象没有被复制，复制的只是它的句柄；与前面例子一样，存在相同对象的两个名字；即时在类的声明中包含了构造new的例化。

类自带的方法还包括.copy（）和.clone，两个都是深复制，区别是copy需要先new一个对象，而clone直接一步到位。

类型转换

从父类向子类的转换称为**向下类型转换** （即 child_handle = father_handle，直接这样复制是不行的）。反之则称为**向上类型转换**（即 father_handle = handle_handle）。$cast （动态转换）则是用

于向下类型转换。在向下类型转换时，**只有当父类指针指向的对象和待转换的类型一致**时，cast才会成功。

父类句柄哪怕已经指向了子类对象（句柄复制是前提），但只要没通过cast转换句柄的类型，访问时还是按句柄类型访问父类对象的非虚方法，而虚方法则是按照对象的类型访问；cast转换后则直接都按照一般习惯的句柄类型来访问。

此外，cast系统函数还可以用于任何类型的转换，$cast(目标，源)。相对应的还有静态转换，也是其他语言经典的，如int`(a)。

随机约束

产生随机的方式：

1. 采用SystemVerilog内置的系统函数来产生随机数，其中有$urandom和$urandom_range等，除此之外，还有一些列标准概率分布的系统函数：$random、$dist_uniform、$dist_normal、$dist_exponetial、$dist_possion、$dist_chi_square、$dist_i和dist_erlang
2. 基于对象的随机生成，随机地初始化对象的数据成员的值
3. 标准随机函数std::randomize()可以随时对任意变量进行随机化并添加约束

随机对应种子，通过改变每次仿真的随机种子来实现随机出不同的值。

With关键词用于跟上后续的约束，主要有==，inside，大于小于，range等类型

用得最多的还是基于2，因为可以提前写好约束constraint。

```
class Stim;
const bit [31:0] CONGEST_ADDR = 42; //定义常量
typedef enum {READ,WRITE,CONTROL} stim_e;
randc stim_e kind;   //枚举类型也可以随机
rand bit [31:0] len,src,dst;  //随机变量
bit congestion_test; //非随机变量
/***************约束*******************/
constraint c_stim{
len < 1000;
len > 0;
if (congestion_test){
dst inside {[CONGEST_ADDR + 100:CONGEST_ADDR - 100]};
src == CONGEST_ADDR;
}
```

else

src inside {0,[2:10],[100:107]};

}

Endclass

需要注意constraint用的是大括号，不能用begin end这些。If还可以替代为：条件->约束

没有在类中用rand声明的成员变量也可以在外部被随机化。对象调用randomize（）函数之前，对象只完成初始化. post_randomize是类自带的一个函数，当某个类的实例的randomize函数被调用后，post_randomize会紧随其后无条件地被调用。

随机约束权重：

dist操作符带有一个值得列表以及相应得权重，中间用：=或：/分开。值和权重可以是常数或者变量。

rand int scr,dst;

constraint c_dist{

src dist {0:=40,[1:3]:=60};// :=的意思是产生40个0，1-3每个数产生60个，共产生220个数

//src = 0,weight(权重) = 40/220

//src = 1,weight(权重) = 60/220

//src = 2,weight(权重) = 60/220

//src = 3,weight(权重) = 60/220

dst dist (0:/40,[1:3]:/60);// :/的意思是0的概率是40%，1-3的概率一共是60%

//dst = 0,weight(权重) = 40/100

//dst = 1,weight(权重) = 20/100

//dst = 2,weight(权重) = 20/100

//dst = 3,weight(权重) = 20/100

}

覆盖率

覆盖率的意义在于对随机激励的验收，去检查到底哪些关心的点被激励打到了

代码覆盖率

行覆盖率（line coverage）：

记录程序的各行代码被执行的情况。

条件覆盖率（condition coverage）：

记录各个条件中的逻辑操作数被覆盖的情况。

翻转覆盖率（toggle coverage）：

记录单bit信号变量的值为0/1跳转情况，如从0到1，或者从1到0的跳转。

分支覆盖率（branch coverage）：

又称路径覆盖率（path coverage），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（FSM coverage）：

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。

代码覆盖率是由工具客观统计的，对于VCS工具则是加上-cm系列的编译选项

功能覆盖率

在进行覆盖率测试之前，先总结归纳覆盖点，建立功能覆盖率模型（验证人员用一定的语法自己写的）。然后通过覆盖率采样、分析得出功能覆盖的信息。覆盖率模型是明确验证计划中的哪些功能点需要进行测试来确保最终设计满足规范。一般主要分为配置相关、数据随机性相关、时序功能相关。

```
covergroup CovKind;
    coverpoint tr.kind{
        bins zero={0};              // 1 个仓代表 kind==0
        bins lo={[1:3],5};          // 1 个仓代表 1:3 和 5 的值
        bins hi[]={[8:$]};          // 8 个独立的仓:8...15
        bins misc=default;          // 1 个仓代表剩余的所有值
    }                               // 没有分号
endgroup     // CoverKind
```

断言

SVA是一个比较大的分支，常用于检查一系列信号是否按照预定的行为变化（更侧重时序）。

即时断言：即时断言一般出现在 always 顺序执行块中，于其他的 RTL 语句混在一起，只有当该顺序语句所在的分支被执行的时候才会被触发，其触发条件和顺序语句的触发条件完全一致。即时断言在形式化验证中并不常用。

并发断言：并发断言是形式化验证中常用的一类的断言。并发断言可以出现在 always 顺序块中或者在 always 块之外单独出现。并发断言的触发条件一定是某一个时钟的跳变沿。因此，在并发断言中通常会用@(clock event)来定义断言的触发条件，而即时断言不能在断言中定义触发条件

UVM的重难点

基本框架

了解基本类uvm_object,了解整个uvm基类的继承结构和常用的基类。

uvm_component有两大特性是uvm_object所没有的，一是通过在new的时候指定parent参数来形成一种树形的组织结构，二是有phase的自动执行特点

整个uvm树形结构就是验证环境的层次结构，需要注意uvm特殊的点就在于这个树形结构是独立于顶层tb的。在tb中通过run_test()方法启动uvm世界。

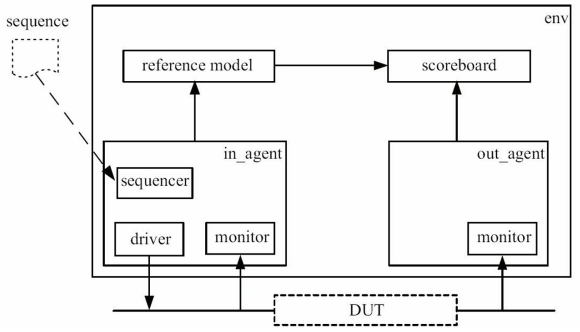


图2-10 带sequence的UVM验证平台

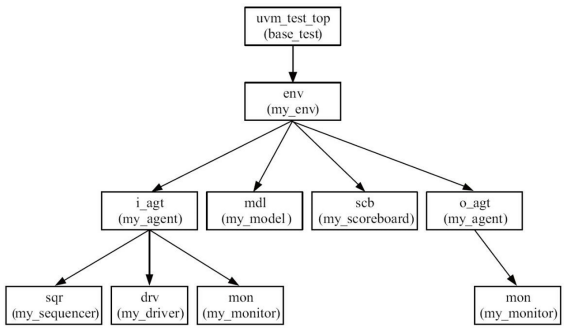


图2-11 UVM树的生长：加入base_test

基本机制

工厂机制

原理：将所有注册到工厂的类存放在两个关联数组中（一个是类型一个是名字），记录是否被修改过。

使用过程：父类和子类均注册到工厂，调用uvm的override函数进行替换（有按名字和按类型两种替换方式，还可以指定特定的实例替换而不是全部替换）

Phase机制

对象的例化可以通过调用new()函数但是无法保证层次化环境的例化先后关系和组件在例化后的连接，SV无法在底层组件例化之前完成对底层的配置。因此phase机制的意义在于让整个验证环境井然有序更加稳定。

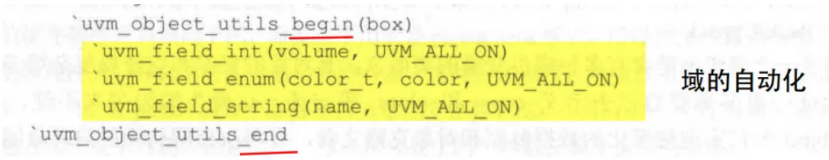
phase	函数/任务	执行顺序	功能	典型应用
build	函数	自顶向下	创建和配置测试平台的结构	创建组件和寄存器模型，设置或获取配置
connect	函数	自底向上	建立组件之间的连接	连接TLM/TLM2的端口，连接寄存器模型和adapter
end_of_elaboration	函数	自底向上	测试环境的微调	显示环境结构，打开文件，为组件添加额外配置
start_of_simulation	函数	自底向上	准备测试环境的仿真	显示环境结构，设置断点，设置初始运行的配置值
run	任务	自底向上	激励设计	提供激励、采集数据和数据比较，与OVM兼容
extract	函数	自底向上	从测试环境中收集数据	从测试平台提取剩余数据，从设计观察最终状态
check	函数	自底向上	检查任何不期望的行为	检查不期望的数据
report	函数	自底向上	报告测试结果	报告测试结果，将结果写入到文件中
final	函数	自顶向下	完成测试活动结束仿真	关闭文件，结束联合仿真引擎

只有uvm_component及其继承于uvm_component的子类才会按照phase机制的顺序先后执行。

与run_phase并行的还有12个run_time phase（小phase），任务phase决定了仿真时长，通过objection机制控制仿真结束。

Field机制

Filed机制在于统一管理所在意的变量，并能够批量调用某些方法（如复制，打印，比较等）在注册类时可以加上需要包括起来的变量，并指定不同变量可以进行操作的开关。



Config_db机制

因为run_test()方法自动实例化类，实例化的是脱离top_tb结构层次的实力。不能像tb.my_driver.xxx那样去访问成员，所以需要用到config机制去操作想改变的。run_test也只会实例化一个类，名字固定为uvm_test_top。利用config机制相比通过参数或预编译指令的重新编译，修改环境更加灵活，也更省时间

宏机制

uvm封装好了很多宏，调用起来会提高效率，但同时封装也隐藏了很多内部细节

TLM

事务就是一系列操作的抽象打包，或者说多根信号线变化的集合。transaction级别不关心具体的时序，uvm世界都是基于事务基本去进行传输比较等操作的，而driver和monitor就是事务级别和pin级别的转换者。

Sequence机制

Sequence机制的意义在于将多样化的激励与相对稳定的验证环境分开，改变时只需要改变需要改动的而不是每次都大规模改动整个验证环境。相当于同样的枪可以打出不同类型的子弹。

Sequence相当于一个弹夹，seq_item相当于弹夹里面的一颗子弹，代表一个抽象事物的执行。

需要了解sequencer和driver之间的握手交互流程

需要了解sequence的三种启动方式

Reg_model

寄存器模型的本质就是重新定义了验证平台与DUT的寄存器接口，使验证人员更好地组织及配置寄存器，简化流程、减少工作量。

前门访问，指的是通过模拟cpu在总线上发出读指令，进行读写操作。在这个过程中，仿真时间（$time函数得到的时间）是一直往前走的

后门访问是与前门访问相对的概念。它并不通过总线进行读写操作，而是直接通过层次化的引用来改变寄存器的值