SystemVerilog总结

1. 数组

1.1 定宽数组

- Verilog要求在声明中必须给出数组的上下界。因为几乎所有数组都使用0作为索引下界，所以SystemVerilog允许置给出数组宽度的便捷声明方式。
  例如：
  int lo_hi[0:15]; //16个整数[0].....[15]
  int c_style[16]; //16个整数[0].....[15]
- 可以通过在变量名后面指定维度的方式来创建多维定宽数组。例如创建几个二维的整数数组，大小都是8行4列
  例如：
  int array2[0:7][0:3]; //完整的声明
  int array3[8][4]; //紧凑的声明
  array2[7][3] = 1; //设置最后一个元素

1.2 合并数组/非合并数组

合并数组和非合并数组的区别在于，合并数组的存放方式是连续的比特合集，中间没有任何闲置的空间。
- 合并数组声明
声明合并数组时，合并的位和数组大小作为数据类型的一部分必须在变量名前面指定。数组大小定义格式必须是[msb:lsb]，而不是[size]。
bit [3:0] [7:0] pack; //4个字节组成32比特

pack	3	2	1	0

> - 非合并数组声明
>   当索引位于数组名称后面，称为非合并数组
>   bit [7:0] un_pack[4]; //非合并数组

unpack[0]	**未使用空间**	**未使用空间**	**未使用空间**	***使用空间***
unpack[1]	**未使用空间**	**未使用空间**	**未使用空间**	*使用空间*
unpack[2]	**未使用空间**	**未使用空间**	**未使用空间**	***使用空间***
unpack[3]	**未使用空间**	**未使用空间**	**未使用空间**	***使用空间***

1.3 动态数组

> 动态数组在声明时使用空的下标[]。意味着数组的宽度不在编译时给出，而在程序运行时再指定。数组在最开始时是空的，所以你必须调用new[]操作符来分配空间，同时在方括号中传递数组宽度。
> **例如**：
> int dyn[]; //声明动态数组
> dyn = new[5]; //分配5个元素

2. 队列

- 队列的声明使用带有美元符号的下标：[$]。队列元素的标号从0到$。注意不要对队列使用构造函数new[]。
  例如:队列操作

```
int j =1;
int q[$] = {0,2,5};
int q2[$] = {3,4};
initial begin
    q.insert(1,j);          //{0,1,2,5} 在2之前插入1
```

```
6.      q.insert(3,q2);          //{0,1,2,3,4,5}在q中插入一个队列
7.      q.delete(1);             //{0,2,3,4,5}删除第1个元素
8.
9.      q.push_front(6);         //{6,0,2,3,4,5}在队列前面插入
10.     j=q.pop_back;            //{6,0,2,3,4} j=5
11.     q.push_back(8);          //{6,0,2,3,4,8}在队列末尾插入
12.     j=q.pop_front;           //{0,2,3,4,8} j=6
13.     q.delete();              //删除整个队列
14. end
```

3. 枚举类型

- 最简单的枚举类型声明包含了一个常量名称列表以及一个或多个变量
  例如：
  enum {RED,BLUE,GREEN} color;
- 创建一个署名的枚举类型有利于声明更多新变量，尤其是当这些变量被用作子程序参数或者端口模块时。需要首先创建枚举类型，然后在创建相应的变量。使用内建的name()函数，可以得到枚举变量值的字符串。一般，使用后缀“_e”来表示枚举类型。枚举值缺省为从0开始递增的整数。你可以定义自己的枚举值。
  例如：INIT代表缺省值0，DECODE代表2，IDLE代表3
  typedef enum {INIT,DECODE=2,IDLE} fsmtype_e;

4. 任务、函数以及void函数

- 在Verilog中，任务（task）和函数（function）之间有明显的区别，其中最重要的一点是，任务可以消耗时间而函数不能。函数里面不能带有诸如#100的延时语句或诸如@(posedge clk)、wait(ready)的阻塞语句，也不能调用任务。另外Veriog中函数必须有返回值，并且返回值必须被使用，例如用到赋值语句中。
- SystemVerilog对这条限制稍微有放宽，允许函数调用任务，但只能在由fork.....join_none语句生成的线程中。
- 高级参数类型
  - Verilog对参数的处理方式很简单：在子程序的开头把input和inout的值复制给本地变量，在子程序退出时则复制给output和inout值。除了标量以

外，没有任何把任何把存储器传递给Verilog程序的方法。
- 在SystemVerilog中，参数的传递方式可以指定为引用而不是复制。这种ref参数类型比input、output或inout更好用。
  向子程序传递数组时尽量使用ref以获取最佳性能。如果你不希望子程序修改数组的值，可以使用const ref类型。在这种情况下，编译器会进行检查以确保数组不被子程序修改。
- ref参数的第二个好处是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。

5. Interface

5.1 背景

当在一个接口中放入一个新的信号时，只需要在接口定义的和实际使用这个接口的模块中做修改，不需要改变其他模块。

5.2 interface中使用modport对信号分组

例如：

```
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
    );
```

```
14.
15.     modport MONITOR(
16.         input request,grant,rst,clk
17.     );
18. endinterface
```

5.3 使用时钟块控制同步信号的时序

- 一个接口可以包含多个时钟块，因为每个块中都只有一个时钟表达式，所以每一个对应一个时钟域。典型的时钟表达式如@(posedge clk)定义了单时钟沿，而@(clk)定义了DDR时钟（双时钟沿）。
- 可以在时钟块中使用default语句指定一个时钟偏斜。
- 一旦定义了时钟块，测试平台就可以用@arbif.cb表达式等待时钟，而不需要描述确切的时钟信号和边沿。这样即使改变了时钟块中的时钟或者边沿，也不需要修改测试平台的代码。
- 在时钟块中应当使用同步驱动，即“<=”操作符来驱动信号。

例如：

```
1.  interface arb_if(input bit clk);
2.      logic [1:0] grant, request;
3.      logic rst;
4.
5.      clocking cb @(posedge clk);
6.          output request;
7.          input grant;
8.      endclocking
9.
10.     modport TEST(clocking cb,//使用cb
11.         output rst,
12.     );
13.
14.     modport DUT(
15.         input request,rst,clk,
16.         output grant
17.     );
18.
```

```
endinterface

module test();

initial begin
    @arbif.cb;
    $display(.........);
end

endmodule
```

5.4 接口中的双向信号

```
interface master_if(input bit clk);
    wire [7:0] data;     //双向信号
    clocking cb @(posedge clk);
    inout data;
    endclocking

    modport TEST(clocking cb);

endinterface

program test(master_if mif);
    initial begin
    mif.cb.data <= 'z;       //三态总线
    mif.cb;
    $displayh(mif.cb.data);     //从总线读取
    mif.cb;
    mif.cb.data <= 7'h5a;    //驱动总线
    @mif.cb;
    mif.cb.data <= 'z;       //释放总线
end

endprogram
```

5.5 为什么在程序（program）中不允许使用always块

> 在System Verilog中，可以在program中使用initial块，但是不能使用always块。在program中最后一个initial块结束的时候，仿真实际上也默默结束了，就像执行了$finish一样。如果加入always块，它将永远不会结束，这样就不得不明确地调用$exit来发出程序块结束的信号。如果确实需要一个always块，可以使用“initial forever”来完成相同的事情。

6. 类class

> - 定义类
>
> 在System Verilog中，可以把类定义在program、module、package中，或者在这些地方的任何地方。类可以在程序和模块中使用。
>
> 对于每一个类来讲，通过new函数来分配和初始化对象。
> - new()和new[]的区别
>
> 两者最大的不同在于调用new()仅创建了一个对象，而new[]操作则建立一个含有多个元素的数组。new()可以使用参数设置对象的数值，而new[]只需要使用一个数值类设置数组的大小。
> - 使用对象

```
Transaciton t;              //声明一个Transaction句柄
t=new();                    //创建一个Transaction对象
t.addr=32'h42;              //设置变量的值
t.display();                //调用一个子程序
```

7. 静态变量和全局变量

> 在SystemVerilog中，可以在类中创建一个静态变量。该变量将被这个类的所有实例所共享，并且它的使用范围仅限于这个类。通过static关键词来创建静态变量。

8. this是什么

> 当使用一个变量名的时候，SystemVerilog将先在当前作用域内寻找，接着在上一级作

> 用域内寻找，直到找到该变量为止。这也是Verilog所采用的算法。但是如果你在类的很深的底层作用域，却想明确地引用类一级的对象呢？这种风格的代码在够早函数里最常见，因为这个时候程序员使用相同的变量名和参数名。例如，关键词“this”明确地告诉SystemVerilog正在将局部变量oname复制给类一级变量oname。
> **例如**：

```
class Scoping;
    string oname;

    function new(string oname);
        this.oname = oname;      //类变量oname=局部变量oname
    endfunction

endclass
```

9. 编译顺序的问题

> 有时候需要编译一个类，而这个类包含一个尚未定义的类。声明这个被包含的类的句柄将会引起错误，因为编译器还不认识这个新的数据类型。这个时候需要声明使用typedef语句声明一个类名
> **例如**：使用typedef 定义class 语句

```
typedef class Statistics;        //定义低级别类

class Transaction;
    Statistics stats;        //使用Statistics类

    ........
endclass

class Statistics;    //定义Statistics类

    ........
endclass
```

10. SystemVerilog随机化

10.1 带有随机变量的简单类

例：简单的随机类

```
class Packet;
    //随机变量
    rand bit[31:0] src, dst, data[8];
    randc bit[7:0] kind;

constraint c {  src >10;
    src<15;
}

endclass

Packet P;
initial begin
    p=new();          //产生一个包
    assert(p.randomize());
    else $fatal(0,”Packet::randomize failed”);
    transmit(p);
end
//randomize()函数为类里所有的rand和randc类型的随机变量赋值一个随机值，并保证不违背所有的有效约束。
```

10.2 约束表达式

在一个表达式中最多只能使用一个关系操作符（ <、<=、==、>=、> ）
例如：不正确的排序约束

```
class order;
    rand bit[7:0] lo,med,hi;
    constraint bad{lo<med<hi;}       //错误！
endclass
//正确的约束：
class order;
    rand bit[7:0];
```

```
    constraint good {    lo<med;
        med<hi;
    }
endclass
```

10.3 权重分布

dist操作符允许产生权重分布，这样某些值的选取机会要比其他值更大一些。dist操作符带有一个值的列表以及相应的权重，中间用:=或:/分开。值和权重可以是常数或变量。值可以是一个值或值的范围，例如[lo:hi]。权重不用百分比表示，权重的和也不必是100。:=操作符表示值范围内的每一个值的范围是相同的，:/操作符表示权重要均分到值范围内的没一个值。

```
rand int src,dst;
constranint c_dist{
    src dist{0:=40,[1:3]:=60};
    //src=0,weight=40/220
    //src=1,weight=60/220
    //src=2,weight=60/220
    //src=3,weight=60/220

    dst dist{0:/=40,[1:3]:/60};
    //src=0,weight=40/100
    //src=1,weight=20/100
    //src=2,weight=20/100
    //src=3,weight=20/100

}
```

上述例子中，src的值可能是0,1,2,3。其中0的权重是40，1,2和3的权重是60，权重的和是220。src取0的概率是40/220，取1,2,3的概率分别都是60/220。
dst的值也可能是0,1,2,3。其中0的权重是40,1,2和3的总权重是60，权重的和是100。dst取0的概率是40/100,取1,2和3的概率分别是20/100。
需要强调的是，值和权重可以是常数或者变量。可以使用权重变量来随时改变置值的概率分布，甚至可以吧权重设为0，从而删除一个值。

10.4 inside 运算符

用inside运算符产生一个值的集合。在集合里也可以使用变量。
例：随机值的集合

```
rand int c; //随机变量
int lo,hi;
    constraint c_range {
        c inside {[lo:hi]}; //lo<=c 并且c<=hi
}
```

可以使用$来代表取值范围里的最小值和最大值。
如果想选择一个集合之外的值，只需要用取反操作符!对约束取反，例如：!(c inside {[lo:hi]});

10.5 条件约束

Systemverilog支持两种关系操作：->和if-else

10.6 双向约束

约束快不像自上而下的程序代码，它们是声明性的代码，是并行的，所有的约束表达式同时有效。如果用inside操作符约束变量的取值范围是[10:15],然后用另外一个表达式约束变量必须大于20，SystemVerilog对两个约束同时求解，最终限定变量的范围是21~50。

10.7 控制多个约束块

在运行期间，可以使用内建的constraint_mode()函数打开或者管边约束。
例如：
p.c_short.constraint_mode(0); //禁止c_short
p.constraint_mode(0); //禁止所有的约束
p.c_long.constraint_mode(1); //打开c_long约束

10.8 内嵌约束

> randomize() with语句进行随机化
> **例如**：
> asser(t.randomize() with {addr==2000; data > 10;}); //强制addr去固定值，data>10

10.9 SystemVerilog提供的新的一些随机函数：

> 1)$random() -------平均分布，返回32位有符号随机数；
> 2)$urandom() -------平均分布，返回32位无符号随机数；
> 3)$urandom_range() --------在指定范围内的平均数
> 4)$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
> $urandom_range()函数有两个参数，一个是上限参数和一个可选的下限参数。

10.10 随机序列randomsequence

> **例如**：

```
initial bein
    for(int i=0; i<15; i++) begin
        strem: cfg_read :=1 | io_read :=2 | mem_read :=5;
        cfg_read: {cfg_read_task;} | {cfg_read_task;} cfg_read;
        mem_read: {mem_read_task;} | {mem_read_task;} mem_read;
        io_read: {io_read_task;} | {io_read_task;}io_read;
    end
end
```

> 上述例子产生了一个stream序列，它可以是cfg_read,io_read或mem_read，随机

> 序列的引擎会随机地从三种操作中选取一种。cfg_read的权重是1，io_read的权重是2，所以被选中的概率是前者的一倍。mem_read的权重是5，被选中的概率最高。
> cfg_read可以是对cfg_read_task任务的一次调用，也可以是在该任务调用后尾随一个cfg_read。所以，cfg_read_task至少会被调用一次，也可能被调用很多次。

10.11使用randcase

> **例如**：

```
initial begin
    int len;
    randcase
        1:  len=\$urandom_range(0,2);  //10%:1,2,or 2
        8:  len=\$urandom_range(3,5);  //80%:3,4or 5
        1:  len=\$urandom_range(6,7);  //10%:6 or 7
    endcase
end
```

11. 线程

- fork.....join fork.....join_any fork.....join_none
- 等待所有线程
  当程序中所有的initial块全部执行完毕，仿真器就退出了。但在多个线程中，有些线程运行时间比较长，可以使用wait fork语句来等待所有子线程结束。
  例如：用wait fork等待所有子线程结束

```
task run_threads;
    ......          //创建一些事务
    fork
        check_trans(tr1);       //产生第一个线程
        check_trans(tr2);       //产生第二个线程
        check_trans(tr3);       //产生第二个线程
```

```
    join_none

    .......          //完成其他的工作
    //在这里等待上述线程结束
    wait fork;
endtask
```

- 停止线程
  Verilog中的disable语句可以用于停止System Veriolog中的线程。

12. 线程间的通信

测试平台中的所有线程都需要同步并交换数据。所有这些数据交换和控制同步被称为线程间的通信。

12.1 事件

- Verilog中通过@操作符来等待事件。该操作符是边沿敏感的，所以它总是阻塞的，等待事件的变化。其他线程可以通过->操作符来触发事件，解除对第一个线程的阻塞。
- SystemVerilog引入triggered()函数，可用于查询某个事件是否已被触发，包括在当前时刻。线程可以等待这个函数的结果，而不用在@操作符上阻塞。
- 可以使用电平敏感的wait(e1.triggered())来替代边沿敏感的阻塞语句@e1。

12.2 信箱

- 对信箱最简单的理解是把它看成一个具有有源端和收端的FIFO。源端把数据放进信箱，收端则从信箱中取出数据。信箱可以有容量上的上限，也可以没有。当源端线程试图向一个容量固定并且已经饱和的信箱里放入数值时，会发生阻塞直到信箱里的数据被移走。同样地，如果收端线程试图从一个>空信箱里移走数据，它也会被阻塞直到有数据放入信箱里。
- 信箱是一种对象，必须调用new函数来进行实例化。例化时有一个可选的参数

> size,用以限制信箱中的条目。如果size是0或者没有指定，则信箱是无限大的，可以容纳任意多的条目。
> - put -------- 把数据放入信箱，如果信箱为满，则该操作壶阻塞
> - get -------- 从信箱中移除数据，如果信箱为空，则该操作会阻塞
> - peek-------- 获取信箱里数据的拷贝而不是移除
> - 注：默认情况下，信箱没有类型，所以允许在其中放入任何混合类型的数据。但是不要这么做！务必一个信箱里只放一种类型的数据。

13. 面向对象

> 在基类中将函数标记位virtual，这样就可以在扩展类中在需要的时候重新定义。应该将类中的子程序定义为虚拟的，这样它们就可以在扩展类中重定义。这一点适应于所有的任务和函数,除了new函数。因为new函数在对象创建时调用，所以无法扩展。
> SystemVerilog始终基于句柄类型来调用new函数。
> 在扩展类中可以通过使用supper前缀调用基类中的函数，可以调用上一层类的成员，但是SystemVerilog不允许用类似super.super.new的方式进行多层调用。
>
> - 类型向下转换$cast
>
>> 类型向下转换或者类型转换是指将一个指向基类的指针转换成一个指向派生类的指针。
>> **例如**：基类和派生类

```
class transaction;
    .........
endclass

class badtr extends transaction;
    .........
endclass
```

> 将一个派生类语句赋值给一个基类句柄，并且不需要任何特殊代码：

```
transaction tr;
badtr bad, bad2;
bad = new();          //构建badtr扩展对象
tr=bad;               //基类句柄指向扩展对象
$display(tr.src);     //显示基类对象的变量成员
tr.dispaly;           //调用badtr::display
```

> 当基类对象拷贝到一个扩展类的句柄时，会发生什么？这种操作会失败。
> $cast会检查句柄所指向的对象类型，而不仅仅检查句柄本身。一旦对象跟目的对象是同一类型，或者是目的类的扩展类，就可以从基类句柄tr中拷贝扩展对象的地址给扩展对象的句柄了。
> $cast(bad2,tr);

- 虚方法

  > 当试图使用句柄来调用一个子程序时，会发生什么？
  > **例如**：Transaction类和BadTr类

```
    class Transaction;
      rand bit[31:0] src,dst,data[8];
      bit[31:0] crc;

      virtual function void calc_crc();      //异或所有域
        crc=src^dst^data.xor;
      enfunction

    endcalss: Transaction
```

```
    class BadTr extends Transaction;
      rand bit bad_crc;

```

```
    virtual function void calc_crc();
      super.cal_crc();          //计算正确的crc
      if(bad_crc) crc=~crc;     //产生错误的crc位
    endfunction

endclass:BadTr
```

> 下面使用不同的类型的句柄的一个代码块：

```
    Transacion tr;
    BadTr bad;

    initial begin
        tr=new();
        tr.calc_crc;            //调用Transacion::calc_crc

        bad=new();
        bad.calc_crc();         //调用BadTr::calc_crc

        tr=bad;                 //基类句柄指向扩展对象
        tr.calc_crc();          //调用BadTr::calc_crc
    end
```

> 当需要决定调用哪个虚类方法的时候，SystemVerilog根据对象的类型，而非句柄的类型来决定调用什么方法。在上述例子中，tr指向一个扩展类对象（Transaction），所以调用的的方法是BadTr::calc_crc。
> 如果没有对calc_crc使用virtual修饰符,SystemVerilog会根据句柄的类型tr,而不是对象的类型，就会导致最后那个语句调用Transaction::calc_crc，这可能不是想要的结果。

14. 功能覆盖率

> 功能覆盖率是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。从设计规范着手，创

建一个验证计划，详细列出要测试什么以及如何进行测试。

14.1 覆盖率类型

- 代码覆盖率
  - 行覆盖率：多少行代码已经被执行过。
  - 路径覆盖率：在穿过代码和表达式的路径中有哪些已经被执行过。
  - 反转覆盖率：哪些单比特量的值为0或1。
  - 状态机覆盖率：状态机中有哪些状态和状态转换已经被访问过。

  不用添加任何额外的代码，工具会通过分析源代码自动完成代码覆盖率的统计。当运行完成所有的测试，代码覆盖率工具便会创建相应的数据库。代码覆盖率衡量的是测试对于设计规范的“实现”究竟测试得多彻底，而非针对验证计划。原因很简单，测试达到100%的覆盖率，并不意味着工作已经完成。
- 功能覆盖率
  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
  断言是用于一次性地或在一段时间内核对两个设计信号之间关系的声明性代码。它可以跟随设计和测试平台一起仿真，也可以被形式检车工具所证实。

14.2 覆盖组

- 概述

  - 覆盖组与类相似，一次定义后便可以进行多次实例化。它含有覆盖点、选项、形式参数和可选触发。
  - 一个覆盖组(covergroup)可以包含一个或多个数据点(coverpoint)
  - covergroup可以定义在类中，也可以定义在interface或者module中
  - covergroup可以采样任何可见的变量，比如程序或模块变量、接口信号或者设计中的任何信号
  - 当拥有多个独立的covergroup时，每个covergroup可以根据需要自行使能或禁止
  - 每组covergroup可以定义单独的触发采样事件，允许从多个源头收集数据
  - 每个covergroup必须被实例化才可以用来采集数据
    例如：在类里定义覆盖组

```
class Transactor;
    Transaction tr;
    mailbox mbx_in;

    covergroup Covport;
        coverpoint tr.port;
    endcovergroup

    function new(maibbox mbx_in);
        Covport=new();
        this.mbx_in=mbx_in;
    endfunction

    task main;
        forever begin
            tr=mbx_in.get;                  //获取下一个任务
            ifc.cb.port<=tr.port;           //发送到待测设计中
            ifc.cb.data<=tr.data;
            Covport.sample();               //收集覆盖率
        end
    endtask
```

```
endclass
```

- 覆盖组的触发
  - 功能覆盖率的两个主要部分是采样的数据和数据被采样的时刻
  - 当上述两个条件都满足之后，测试平台便会触发covergroup
  - 这个过程可以通过直接使用sample()函数来完成，或者在coverproup的定义中采用阻塞表达式
  - 阻塞表达式可以使用wait或@来实现在信号或事件上的阻塞
  - 如果希望在程序代码中显示地触发覆盖组，或者不存在可以标识采样时刻的信号或者事件，又或者在一个覆盖组里有很多个实例需要独立触发，可以采用sample()函数
  - 如果想借助已有的事件或信号来触发覆盖组，可以在covergroup声明中使用阻塞语句

使用事件触发覆盖组:

```
event trans_ready;
covergroup CovPort @(trans_ready);
    coverproup ifc.cb.port;      //测量覆盖率
endcoverproup
//覆盖组covPort在测试平台触发trans_ready事件时进行采样
```

使用SystemVerilog断言进行触发

```
module mem(simple_bus sb);
    bit[7:0] data,addr;
    event write_event;

    cover property
    (@(posedge sb.clock)sb.write_ena==1);
    ->write_event;
```

```
8.
9.  endmodule
10.
11. program automatic test(simple_bus sb);
12.
13.     covergroup Write_cg @(\$root.top.m1.write_event);
14.         coverpoint \$root.top.m1.data;
15.         coverpoint \$root.top.m1.addr;
16.     endgroup
17.
18.     Write_cg wcg;
19.
20.     initial begin
21.         wcg=new();
22.         //在此处添加激励
23.         sb.write_ena<=1;
24.         ......
25.         #10000 $finish;
26.
27.     end
28. endprogram
29. //与直接调用sample方法相比，使用事件触发的好处在于能够借助已有的事件。
```

14.3 数据采样

- 概述
  - 当在覆盖点（coverpoint）上指定一个变量或表达式时，SV便会创建很多的”仓（bin）“来记录每个数值被捕捉到的次数。这些仓是衡量功能覆盖率的基本单位
  - 每个covergroup中可以定义多个coverpoint,coverpoint中可以自定义多个bin
  - 每次covertgroup被触发，SV都会在一个或多个仓里留下标记，在仿真末尾，所有带标记的仓会被汇聚到一个新创建的数据库中，生成覆盖率

- 个体仓和整体覆盖率

  - 为了计算出一个点上的覆盖率，首先必须确定所有可能数值的个数，这也被称之为域。一个仓中可能有一个或多个值。
  - 覆盖率就是采样值的数目除以域中仓的数目。比如一个3比特变量覆盖点的域是0:7，正常情况下总共有8个仓。如果在仿真的过程中有七个bin的值被采样到，那么报告就会给出这个点的覆盖率是87.5%(7/8)。
  - 所有的coverpoint组合在一起构成了所在coverproup的覆盖率
  - 所有的coverproup组组合在一起构成了整个仿真的覆盖率

- bin的创建

  - SV会自动为覆盖点创建bin
  - covergroup选项auto_bin_max指明了自动创建bin的最大数目，默认是64
  - 如果coverpoint或者表达式的值域超过指定的最大值，SV会把值域范围平均分配给auto_bin_max个仓。例如，一个16比特变量有65536个可能值，所以64个bin中每一个都是覆盖了1024个值
  - 实际操作中，最好自定义bin,或者降低自动建仓的最大数量

```
1.  //在所有覆盖点中使用auto_bin_max
2.  coverproup CovPort;
3.      options.auto_bin_max=2;      //分成两个bin,影响port和data
4.      coverpoint tr.port;
5.      coverpoint tr.data;
6.  endgroup
```

- 命名覆盖点的仓

  - 命名bin的最简单的方式是使用[]
  - coverpoint是使用大括号{}围起来的。因为对bin的命名是声明语句而

非程序性语句。大括号的末尾并没有带分号
    - 一般情况下，指定bin的时候，建议使用default语句来捕捉那些可能被忘记的数值

```
covergroup CovKind;
    coverpoint tr.kind{
        bins zero={0};              //1个bin代表kind=0
        bins lo={[1:3],5};          //1个bin代表1:3和5
        bins hi[]={[8:$]};          //8个独立的bin:8....15
        bins misc=default;          //1个bin代表剩余所有的值
    }                               //没有分号
endgroup
```

- 条件覆盖率
    - 可以使用关键字iff给coverpoint添加条件
    - 该做法常用于在复位期间关闭关闭覆盖以忽略一些杂散的触发
    - 可以使用start和stop函数来控制覆盖组里各个独立的实例

```
covergroup CoverPort;
    //当reset==1时不要收集覆盖率数据
    coverpoint port iff(!bus_if.reset);
endgroup
```

```
initial begin
    CovPort ck=new();       //实例化覆盖组
    //复位期间停止收集覆盖率数据
    #1ns ck.stop();
    bus_if.reset=1;
    #100ns bus_if.reset=0;  //复位结束
```

```
    ck.start();
    ...
end
```

- 为枚举类型创建bin
  - 对于枚举类型，SV会为每个可能值创建一个bin
  - auto_bin_max在收集枚举类型的覆盖率时不起作用

```
typedef enum{INIT,DECODE,IDLE} fsmstate_e;
fsmstate_e pstate,nstat;          //声明自由类型变量
covergroup cg_fsm;
    coverpoint pstate;
endgroup
```

- 反转覆盖率
  - 可以确定值之间的转换过程
  - 可以确定值之间转换的次数

```
covergroup CoverPort;
    coverpoint port{
        bins t1=(0=>1),(0=>2),(0=>3);          //0变为1、2或3
        bins t2=(0=>1[*3]=>2);                 //(0=>1=>1=>1=>2)
        bins t3=(0=>1[*3:5]=>2)                //需要对数值1进行3次，4次或5次重复
    }
endgroup
```

- wildcard通配符
  - 使用关键字wildcard来创建多个状态或反转
  - 在表达式中，任何X、Z或？都会被当成0或1的通配符

```
bit[2:0] port;
coverproup CoverPort;
    coverpoint port{
        wildcard bins even={3'b??0};        //代表偶数值
        wildcard bins odd={3'b??1};         //代表奇数值
    }
endgroup
```

- 忽略数值
  - 在某些coverpoint上，可能始终得不到全部的可能值，对于这些值不打算计算在覆盖率收集的范围内，有两种解决办法：
  - 可以明确定义bin来涵盖所有的期望值
  - 让SV自动建仓，然后使用ignore_bins排除掉那些不用来计算功能覆盖率的值

```
bit[2:0] low_ports_0_5;          //只使用数值0-5
covergroup CoverPort;
    coverpoint low_ports_0_5{
        ignore_bins hi={[6,7]}; //忽略最后两个仓
```

```
    }
endgroup
```

- 不合法的仓
  - 有些值不仅应该被忽略，而且如果出现还应该报错。这种情况最好在测试平台中使用代码进行监测，但也可以使用illegal_bins对bin进行标识。
  - 如果在覆盖组中发现了不合法的数值，那就是测试程序或者bin定义出来问题。

```
bit[2:0] low_ports_0_5;          //只使用数值0-5
covergroup CoverPort;
    coverpoint low_ports_0_5{
        illegal_bins hi={[6,7]}; //如果出现便报错
    }
endgroup
```

14.4 交叉覆盖率

- 概述
  - coverpoint记录的是单个变量或表达式的观测值。
  - SV中cross结构可以用来记录一个组里两个或两个以上coverpoint的组合值。
  - cross语句只允许带coverpoint或者简单的变量名。

```
class Transaction;
    rand bit[3:0]kind;
    rand bit[2:0]port;
enclass

Transaction tr;

corvergroup CovPort;
    kind: coverpoint tr.kind;        //创建覆盖点kind
    port: coverpoint tr.port;        //创建覆盖点port

    cross kind,port;                 //把kind和port交叉
endgroup
```

- 排除掉部分交叉覆盖仓
  - 使用ignore_bins可以减少仓的数目。在交叉覆盖中，可以使用binsof和intersect分别指定覆盖点和数值集。
  - binsof使用的是小括号(),而intersect指定的是一个范围，所以使用的是大括号{}。

```
covergroup Coverport;

    port:coverpoint tr.port{
        bins port[]={[0:$]};
    }

    coverpoint tr.kind{
        bins zero={0};               //1个bin代表kind=0
        bins lo={[1:3]};             //1个bin代表1:3
        bins hi[]={[8:$]};           //8个独立的bin:8....15
        bins misc=default;           //1个bin代表剩余所有的值
    }

```

```
    cross port,kind{
        ignore_bins hi=binsof(port)intersect{7};          //排除掉所有代表port=7和任意kind值的组合
        ignore_bins md=binsof(port)intersect{0}&&
                       binsof(kind)intersect{[9:11]};     //排除掉port=0和kind为9, 10, 11的组合, 总共3个仓
        ignore_bins lo=binsof(kind.lo);                   //使用仓名排除掉kind.lo
    }
endgroup
```

- 交叉覆盖率的权重
  - 一个组的总体覆盖率是基于所有简单的覆盖点和交叉覆盖率的。
  - 如果希望对一个coverpoint上的变量或表达式进行采样，而这个coverpoint将会被用到cross语句中，应该将它的权重设置为0。

```
covergroup CovPort;
    kind:coverpoint tr.kind{
        bins zero={0};
        bins lo={[1:3]};
        bins hi[]={[8:$]};
        bins misc=default;
        option.weight=5;          //在总体中所占的分量
    }
    port:coverport tr.port{
        bins port[]={[0:$]};
        option.weight=0;          //在整体中不占任何分量
    }

    cross kind,port{
        option.weight=10;         //给予交叉更高的权重
    }

endgroup
```

14.5 覆盖选项

- 单个实例的覆盖率
  - 如果测试平台对一个覆盖组进行多次实例化，默认情况下SV会把所有的实例的覆盖率数据汇聚到一起。
  - 如果有几个发生器，每个发生器需要所产生的事务数据流都不相同，需要查看单独的报告，可以使用per_instance。
  - 选项per_instance只能放在覆盖组里，不能用于覆盖点或交叉点。

```
coverproup CoverLength
    coverpoint tr.length;
    option.per_instance=1;
endcovergroup
```

- 覆盖组的注释
  - 可以在覆盖组报告中增加注释以使报告更易于分析。
  - 注释可以尽量简单，例如使用验证计划中的小节号或标签。
  - 如果只实例化一次的覆盖组，可以使用type选项。
  - 如果有多个实例，可以为每个实例加入单独的注释，前提是同时使用了per_instance选项。

```
covergroup CoverPort;
```

```
    type_option.comment="Section 3.2.14 Port numbers";
    coverpoint port;
endgroup
```

```
covergroup CoverPort(int lo,hi,string comment);
    option.comment=comment;
    option.per_instance=1;
    coverpoint port{
        bins rande={[li:hi]};
    }
endgroup

.....

CoverPort cp_lo=new(0,3,"Low port numbers");
CoverPort cp_hi=new(4,7,"High port numbers");
```

- 覆盖阈值
  - 通过option.at_least设置仓至少命中的次数。
  - option.at_least如果定义在覆盖组里，那么它会作用于所有的覆盖点。如果定义在一个点上，那它只对该点有效。
- 打印空仓
  - 默认情况下，覆盖率报告只会给出带有采样值得bin。
  - 使用cross_num_print_missing选项可以让仿真和报告工具给出所有bin，尤其是那些没有被命中的bin。

```
covergroup CovPort;
    kind:coverpoint tr.kind;
    port:coverpoint tr.port;
    cross kind,port;
    option.cross_num_print_missing=1000;
endgroup
```

- 覆盖率目标
  - 一个覆盖组或者覆盖点的目标是达到改组或者该点被认为已经被完全覆盖的水平，默认情况下是100%。
  - 通过option.goal可以设置覆盖率目标低于100%。
- 覆盖率的查询
  - get_coverage()可以得到所有覆盖组的总覆盖率，该函数返回一个介于0~100的实数,其用法如CoverGroup::get_coverage()或cgInst::get_coverage()。
  - get_inst_coverage()可以得到一个特定覆盖组实例的覆盖率，该函数返回一个介于0~100的实数,其用法如cgInst.get_inst_coverage()。