文档版本： Ver1.0
最后修改日期： 2015-05-30
修改人： William

E 课网 – UVM 实战培训

www.eecourse.com

klin@eecourse.com

SVV 实验 – Lab05_Guide

实验简介：

在之前的实验中，我们已经建立了一个简单的分层的 Testbench。在本次实验中，我们将为 Testbench 添加统计元素 – 覆盖率。

实验目的：

★ 熟练掌握 coverage 的写法，用法
★ 使用 QuestaSim 查看覆盖率

实验准备：

➢ 进入本次实验的实验目录

```
cd verification/svv/lab05
```

该目录中包含了本次实验所需要的相关代码和文档。

➢ 请确认你已经获得本次实验的 DUT 参考手《E 课网 UVM 实战培训 SVV 实验 DUT 手册.pdf》

➢ 如果在上述准备工作中遇到任何困难，请及时与讲师联系解决。

实验步骤：

1. 了解 coverage

在芯片验证中，为了衡量验证的进度，常常需要借助于 SystemVerilog 所提供的 Functional Coverage。Coverage 需要验证工程师根据 DUT 的 Spec 自行的制订需要覆盖的功能点，只有一个全面的 Coverage 才被认为是有意义的，对于一个不全面的 Coverage 来说，即使验证达到了 100%的覆盖率，并不代表验证工作的结束。因此，Coverage 功能点制订是至关重要的，一般需要多人进行反复的 review 才能最终确定下来。

Coverage 作为一种独立的结构可以定义在任何地方，常用的做法是将其嵌套在 Class 中。该实验将 Coverage 嵌入 Scoreboard 中。

以 Coverage 为参考，及时分析 Coverage 的覆盖情况，并针对未覆盖的功能点制订相应的 Testcase 来进行覆盖，这种验证策略被称为是“Coverage-driven Verification”（覆盖率驱动的验证）。

2. 定义 Coverage 结构 – Covergroup

在 Scoreboard 中嵌入 Coverage 结构（Scoreboard_coverage.sv 文件）。

```
// COVERAGE ADDITION
covergroup Arith_Cov_Ver1;
  coverpoint    pkt_sent.imm;
  coverpoint    pkt_sent.src1;
  coverpoint    pkt_sent.src2;
  coverpoint    pkt_sent.opselect_gen;
  coverpoint    pkt_sent.operation_gen;
endgroup
……
```

①. covergroup 的定义：在代码中可以看到定义了一个名为 Arith_Cov_Ver1 的 covergroup。该 covergroup 目的是对输入 DUT 信号所有可能性的统计。coverpoint 表示所要覆盖的点，默认情况下对该点的所有值进行统计。使用 pkt_sent 对象做为统计对象，对该对象中的成员进行覆盖率的统计。我们可以看到，Arith_Cov_Ver1 所要覆盖的点十分的庞大，光是 src1 就有 2^{32} 可能，是不可能全部都覆盖（事实上，在这种情况下，SystemVerilog 会默认将它们划分在最多 64 个范围内）。

除了 Arith_Cov_Ver1 之外，我们还定义了 Arith_Cov_Ver2 和 Arith_Cov_Ver3。在这两个 covergroup 中，我们适当的对 coverpoint 所关注的点做了一些设定，这样更符合实际的情况。

②. 实例化 covergroup： covergroup 和 class 一样，需要实例化之后才能被使用，其实就是申请内存空间。covergroup 的实例化放在 Scoreboard 的 new()函数中，也就是说，在 Scoreboard 实例化的时候，covergroup 也随之被实例化了。这里我们定义并实例化了很多种 covergroup。

```
/* Arith_Cov_Ver1 = new();*/
//Arith_Cov_Ver2 = new();
/*Arith_Cov_Ver3 = new();*/
All_Ip_Operands_Cov =new();
All_Inter_Operands_Cov =new();
Arith_Cov=new();
Shift_Cov=new();
Mem_Rd_Cov=new();
Mem_Write=new();
……
```

③. Coverage 数据采样：只有在执行了采样之后，数据才会被统计入 Coverage 当中。Coverage 的采样有手动和自动两种方式，在本例中我们使用手动的方式进行采样。在 Scoreboard 的 check 任务中可以看到如下代码：

```
……
/* // COVERAGE ADDITION */
//Arith_Cov_Ver1.sample();
//Arith_Cov_Ver2.sample();
/*Arith_Cov_Ver3.sample();*/
All_Ip_Operands_Cov.sample();
All_Inter_Operands_Cov.sample();
Arith_Cov.sample();
Shift_Cov.sample();
Mem_Rd_Cov.sample();
Mem_Write.sample();
……
```

调用 covergroup 的 sample 函数，即可对该 covergroup 中所包含的所有的 coverpoint 进行一次采样统计，注意这里只采样统计一次。由于 check()任务每过一个时钟周期都会被调用一次，所以事实上每一个周期都会调用 sample 函数。

④. coverpoint 值的更新：coverpoint 的值需要被更新，这样每次 sample 才不会一直统计相同的值。比如 Arith_Cov_Ver1 中，pkt_sent 中的值就需要更新，

但是更新要有依据，这个更新发生在 Scoreboard 的 start()任务当中：

```
……
task Scoreboard::start();
  ……
  fork
    forever begin
      ……
        while (receiver_mbox.num()) begin
          receiver_mbox.get(pkt_cmp);
          driver_mbox.get(pkt_sent);
          check();
        end
    end
  join_none
  ……
endtask
```

pkt_sent 的值来自于 driver，每 get 成功之后 pkt_sent 的值就会发生变化。而 check()发生在该动作之后。因此 sample 采样才会得到不同的值。

⑤. 获取 covergroup 的覆盖率：可以调用 get_coverage()来获取目前该 covergroup 所达到的覆盖率数值，该值是一个表示覆盖率百分比的实数，以此来了解当前的覆盖率情况。

```
……
/* coverage_value1 = Arith_Cov_Ver1.get_coverage();*/
//coverage_value2 = Arith_Cov_Ver2.get_coverage();
/*coverage_value3 = Arith_Cov_Ver3.get_coverage();*/
coverage_value1 =   All_Ip_Operands_Cov.get_coverage();
coverage_value2 =   All_Inter_Operands_Cov.get_coverage();
coverage_value3 =   Arith_Cov.get_coverage();
coverage_value4 =   Shift_Cov.get_coverage();
coverage_value5 =   Mem_Rd_Cov.get_coverage();
coverage_value6 =   Mem_Write.get_coverage();
……
```

在 Scoreboard 的 check()任务中，可以看到相关代码。

⑥. Arith_Cov_Ver2 中的 Coverpoint：

```
……
opselect_cov1: coverpoint pkt_sent.opselect_gen;
opselect_cov2: coverpoint pkt_sent.opselect_gen {
        bins shift = {0};
        bins arith = {1};
        bins mem = {[4:5]};
}
……
```

上面两个 coverpoint 其实是对同一个变量进行覆盖率统计，但是我们知道，opselect_gen 只有在等于 0、1、4、5 才是合法的，而其它的值是没有意义的，那么就不需要对这些值进行覆盖率统计。opselect_cov1 会统计其他不合法的值，

而 opselect_cov2 则不会，一旦使用了自定义的 bins，那些不在自定义 bins 的涵盖范围内的数值就会被忽略掉。自定义 bins 的另一个好处就是仓的名字即为定义的仓，在观察覆盖率的时候这些仓名会显示出来，增强阅读性。比如如果“1”被覆盖了，就表示名为 arith 的仓被覆盖了；而只要“4”或者“5”有一个被覆盖了，就表示名为 mem 的仓被覆盖了。通过定义 bins，可以使得 coverpoint 只统计我们所感兴趣的值，而忽略那些不需要进行覆盖率统计的值。

⑦. Arith_Cov_Ver3 中的 Coverpoint：

```
……
src1_cov: coverpoint pkt_sent.src1 {
        bins zero = {0};
        bins allfs = {32'hffffffff};
        bins special1 = {32'h55555555};
        bins special2 = {32'haaaaaaaa};
        bins positive = {[0:'1]} iff(pkt_sent.src1[31] == 1'b0);
        bins negative = {[0:'1]} iff(pkt_sent.src1[31] == 1'b1);
}
……
```

前面讲到过，src1 是一个 32 位的变量，它可能的值有 2^{32} 种，我们不可能也没有必要全部进行覆盖，只需要覆盖一些关键的值即可。在 Arith_Cov_Ver3 中，对于 src1 coverpoint 我们自定义了一些 bins：zero 仓表示值为 0，allfs 仓表示全 1，其他特殊的值还有“01”循环（32'h55555555）和“10”循环（32'haaaaaaaa），正数和负数。一般情况下，只要上述都覆盖到了，我们就认为是全部覆盖到了。关键字 **iff** 表示为仓添加条件，满足了后面条件的值才被认为是 hit 到该仓，比如

```
bins positive = {[0:'1]} iff(pkt_sent.src1[31] == 1'b0);
```

表示 src1 的最高位为 0 时，positive 仓被 hit 到。事实上，不仅仅是仓，覆盖点也可以使用 **iff**。

⑧. 其他形式的 bins 的定义请认真阅读代码，弄清楚它们所代表的含义。

3. 修改脚本文件 Makefile

按照下图所示，修改 Makefile 文件中的第 2、10、13、16 行：

```
1
2  SEED=$(shell date +"%s")
3  all: create_lib compile simulate
4
5
6  create_lib:
7    vlib work
8
9  compile:
10   vlog -l comp.log -mfcu -sv data_defs.v Packet.sv OutputPacket.sv Driver.sv Receiver.sv
   Scoreboard_coverage.sv Generator.sv Ex_Preproc.vp Arith_ALU.vp Shift_ALU.vp ALU.vp Top.v
   Execute.if.sv Execute.tb.sv Execute.test_top.sv
11
12 simulate:
13   vsim -l sim.log -novopt -coverage -sv_seed ${SEED} Execute_test_top -do "run -all"
14
15 clean:
16   rm -rf work mti_lib transcript modelsim.ini *.log *.wlf wlf* *.ucdb compile*
```

在之前的实验中，细心的学员会发现，虽然激励的产生使用了随机化的方式，但是每一

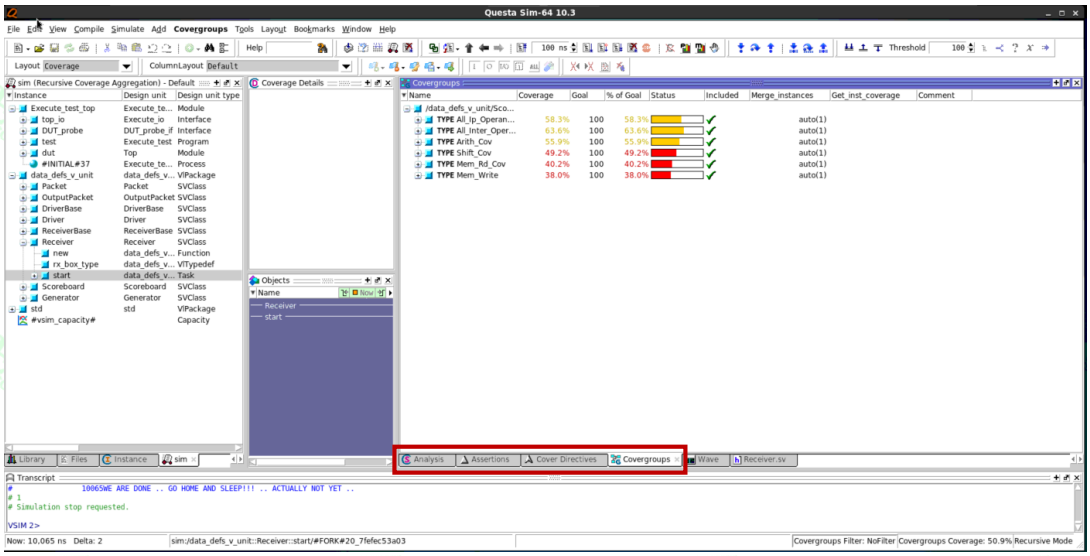
次仿真之后得到的结果却是一样的，原因在于这是一种伪随机，每次运行仿真时，随机化的种子都是默认的，如果随机化的种子相同，那么随机化出来的激励也是相同的。因此，为了能使我们的激励真正的得到随机，也就是每次仿真都能得到不同的随机值，我们就需要在每次仿真时给予不同的种子。

上述 Makefile 的第二行，我们添加了一个"SEED"变量，该变量的值取决于当前系统的时间（date +"%s" 是 shell 命令，可以直接在 Terminal 中运行，注意+之前有空格，之后没有），因为每次运行仿真的系统时间肯定是不一样的，所以每次调用 Makefile 时，得到的"SEED"的值也是不一样的。然后我们将这个"SEED"的值做为 QuestaSim 仿真时的随机化种子，每次"SEED"的值不同，那么激励随机化的值也是不一样的。那么如何将"SEED"做为 QuestaSim 仿真时的种子呢？看第 13 行，在调用"vsim"命令时，加上"**-sv_seed**"参数，该参数表示告诉 QuestaSim 在仿真的时候，将该参数后面的值做为随机化的种子，也就是"${SEED}"做为随机化种子。"${ }"表示对变量值的引用。

在第 13 行中，加入"**-coverage**"参数，表示在仿真时，启动 coverage 统计界面，默认情况下，不启动。

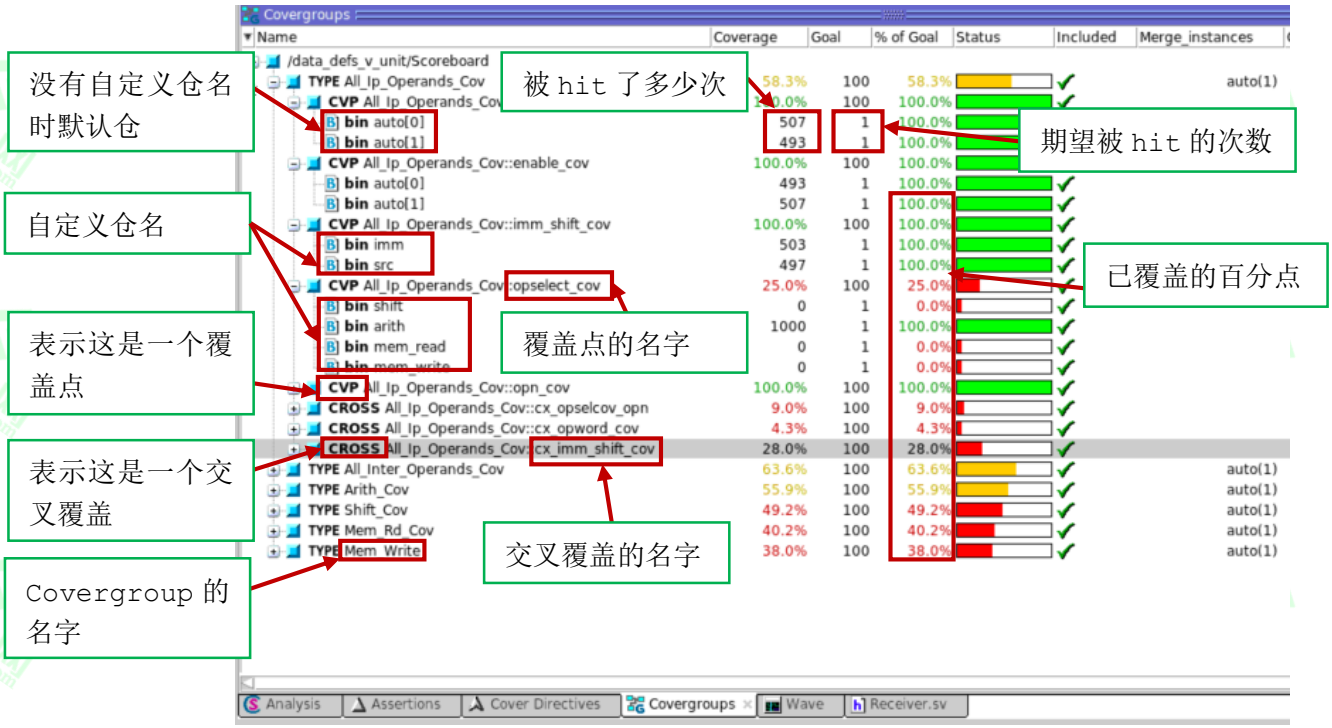
4. Coverage 统计 GUI 界面

执行 make 命令，等到仿真结束之后可以看到如下界面：



相对应之前的 GUI，这里多了几个标签。

点击"Covergroups"标签可以看到 Functional Coverage 的统计信息，这些 covergroup 是在 Scoreboard_Coverage 中定义的。

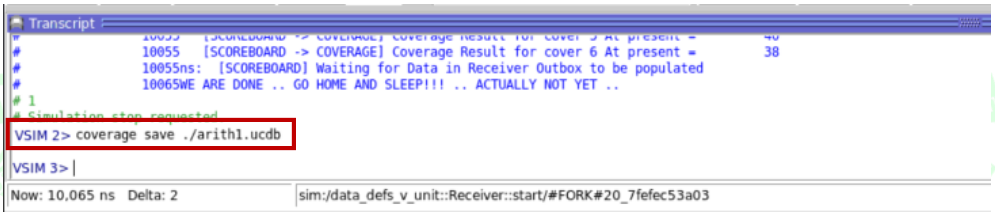


点击前面的“+”，可以将 coverpoint 展开，能够详细的看到被覆盖的情况。

请对照“Scoreboard_Coverage.sv”文件中各个 covergroup 的定义和该 coverage 的统计表，理解它们的对应关系。

5. Coverage 的相关操作

①. 保存 coverage 文件：



如上图所示，在 Transcript 中输入

“coverage save ./arith1.ucdb”

命令，将当前的 coverage 数据保存下来，后缀名为 ucdb。执行上述命令之后，会在当前目录下产生一个“arith1.ucdb”的文件。

②. 多个 coverage 文件的合并：

在实际的项目中，testcase 不止有一个，每次跑完一个 testcase 之后，都会将覆盖率保存在一个以 ucdb(Universal Coverage DataBase)为后缀名的文件当中。可以将这些文件合并成一个文件，合并后的 coverage 文件包含了所有 testcase 的覆盖率情况。在本次实验中，我们通过使用两个不同的 packet 来跑两次仿真，上面的仿真用的是“Packet.sv”文件，下面，我们使用“Packet2.sv”文件来代替“Packet.sv”文件再跑一次仿真，来产生第二个 coverage 文件。

修改 Makefile，如下：

```
1
2  SEED=$(shell date +"%s")
3  all: create_lib compile simulate
4
5
6  create_lib:
7    vlib work
8
9  compile:
10   vlog -l comp.log -mfcu -sv data_defs.v Packet2.sv OutputPacket.sv Driver.sv Receiver.sv
   Scoreboard_coverage.sv Generator.sv Ex_Preproc.vp Arith_ALU.vp Shift_ALU.vp ALU.vp Top.v
   Execute.if.sv Execute.tb.sv Execute.test_top.sv
11
12 simulate:
13   vsim -l sim.log -novopt -coverage -sv_seed ${SEED} Execute_test_top -do "run -all"
14
15 clean:
16   rm -rf work mti_lib transcript modelsim.ini *.log *.wlf wlf* *.ucdb compile*
```

这里主要是将原来的“Packet.sv”修改为“Packet2.sv”。

修改完毕之后，执行 make 命令（注意，因为 clean 中会将 ucdb 文件清理掉，所以不要执行 make clean）。等待仿真结束之后，

在 Transcript 中输入

“coverage save ./arith2.ucdb”

命令，将当前的 coverage 数据保存下来，后缀名为 ucdb。执行上述命令之后，会在当前目录下产生一个“arith2.ucdb”的文件。

下面，我们要将“arith1.ucdb” 和“arith2.ucdb”合并成一个文件，

在 Transcript 中输入

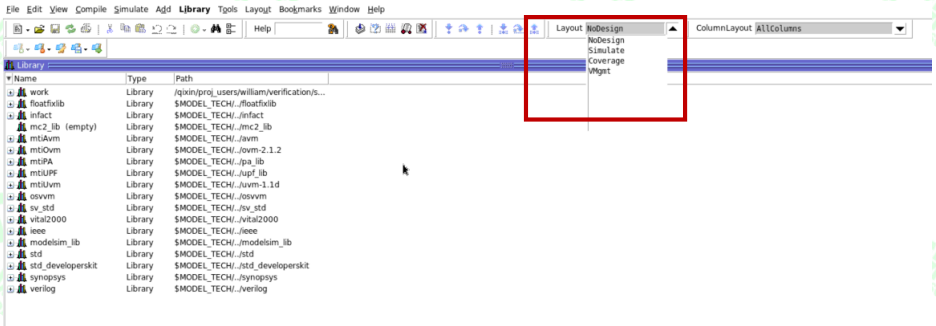
“vcover merge arith_final.ucdb ./arith2.ucdb ./arith2.ucdb”

上述命令是 coverage 的合并命令，其中“arith_final.ucdb”为合并之后产生的 UCDB 文件名，最后的“./arith1.ucdb ./arith2.ucdb”为要合并的目标文件。如果命令执行成功，则会在当前目录下生成一个“arith_final.ucdb”文件。

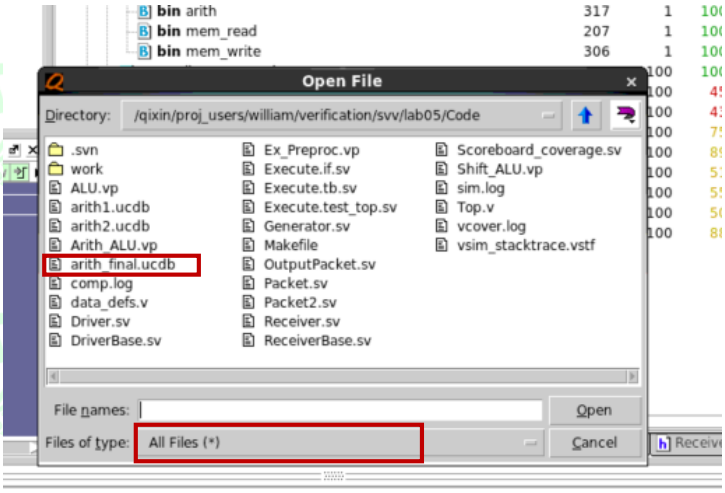
打开 UCDB 文件：

重新启动一个新的 QuestaSim（在 Terminal 中执行命令 vsim），

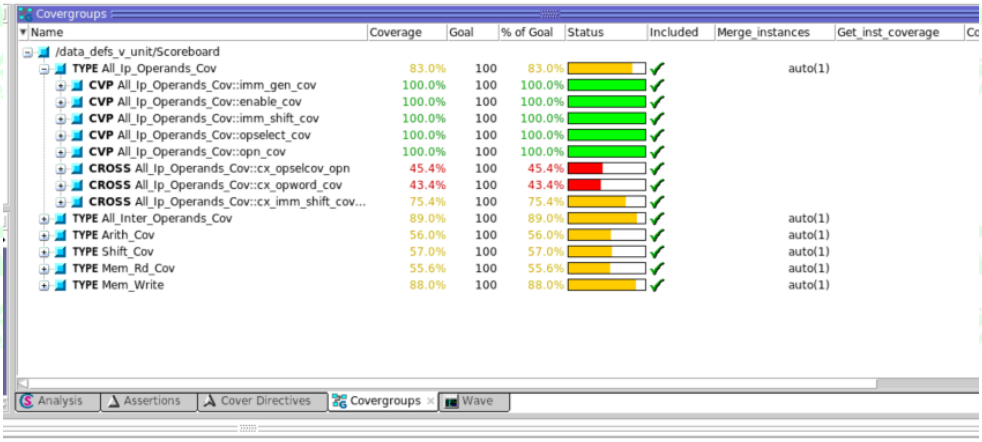
出现 GUI 之后，在“Layout”中选择“coverage”，如下图：



然后在菜单栏执行“File” -> “Open”。在“Files of type”选择“All Files(*)”，在文件栏中找到“arith_final.ucdb”文件，将其打开。



点击“Covergroups”标签即可看到覆盖率统计。



多运行几次仿真，将每次的 Coverage 文件都保存下来，然后将它们合并起来，看看是否可以达到 100%的覆盖率。

实验问题：

请将以下问题以回帖的方式在论坛上作答。

1. 打开“Scoreboard_coverage.sv”文件，将第 668 行以及 718 行的注释打开，（即：实例化一个 Arith_Cov_Ver1 并进行覆盖率统计）保存文件后运行仿真，仿真结束之后，对名为 Arith_Cov_Ver1 的 Covergroup 进行观察，然后回答以下问题：
   ①. 该 Covergroup 中一共包含了多少个 Coverpoint？它们的名字各自是什么？
   ②. 将这些 coverpoint 和代码中的定义对应起来。
   ③. 每一个 coverpoint 中各自都包含了几个 bins，并说明原因。

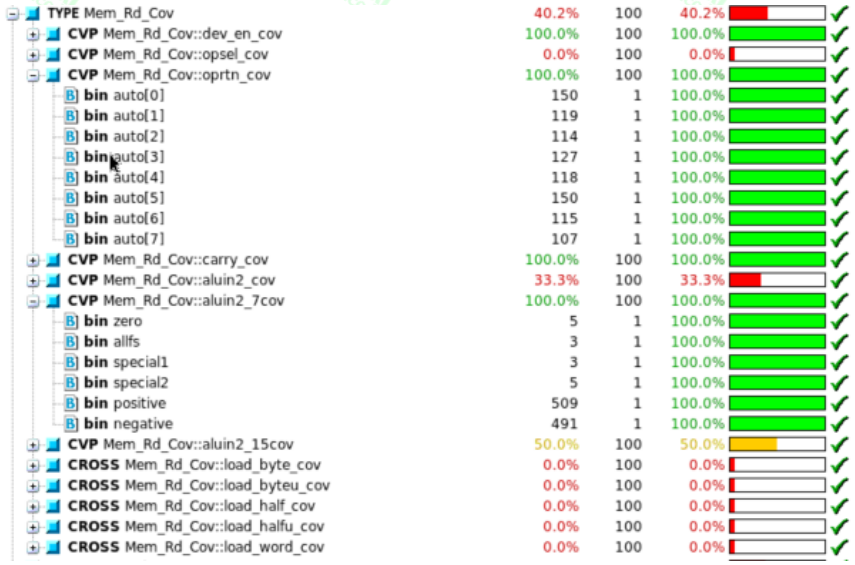
   答：

2. 请解释下列代码中，阴影部分代码的含义（Scoreboard_coverage.sv 文件中的第 131~146 行）。

```
131         opselect_cov: coverpoint pkt_sent.opselect_gen {
132                     bins shift = {0};
133                     bins arith = {1};
134                     bins mem_read = {5};
135                     bins mem_write = {4};
136         }
137
138         opn_cov: coverpoint pkt_sent.operation_gen;
139
140         cx_opselcov_opn: cross opselect_cov, opn_cov
141                         {
142                     bins shift = binsof(opselect_cov.shift) && binsof(opn_cov) intersect {0,1,2,3};
143                     bins arith =  binsof(opselect_cov.arith) && binsof(opn_cov);
144                     bins mem_read =  binsof(opselect_cov.mem_read) && binsof(opn_cov) intersect {0,1,2,4,5};
145                     bins mem_write =  binsof(opselect_cov.mem_write) && binsof(opn_cov);
146                 }
```

答：

3. 看覆盖率图，回答下列问题



①. 图中有多少个 Covergroup，它们的名字是什么。

②. 图中有多少个 coverpoint，它们的名字是什么。

③. 图中有多少个 cross，它们的名字是什么。

④. 名为 aluin2_7cov 的 coverpoint/cross 中包含了几个仓，写出它们的名字以及各自被 hit 了多少次。

答：