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 – Lab04_Guide

实验简介：

在 lab03 中，我们将激励抽象为一个类并将 Testbench 中一些功能使用 OOP 的方式分离出来 – 激励的产生和激励的驱动。在该实验中，我们会继续将其他的功能使用 OOP 的方式分离出来，这些功能包括：

- 接收器 – Receiver　负责接收 DUT 的输出信号
- 计分板 – Scoreboard 将 checker 的功能合并进来

将 DUT 的输出信号抽象为一个类。

通过该实验，使得整个 Testbench 的各个功能完全的独立开来，形成一个简单的可重用性的 Testbench。

实验目的：

★ 继续将 `Execute.tb.sv` 中剩余部分的功能分离出来，并将其封装到类中来实现，完善 Testbench 的分层结构；

★ 熟练掌握在 Testbench 中使用 `Mailbox` 来实现不同组件之间事务传递。

★ 初步建立起一个具有可重用性的 Testbench 的概念。

实验准备：

- 进入本次实验的实验目录

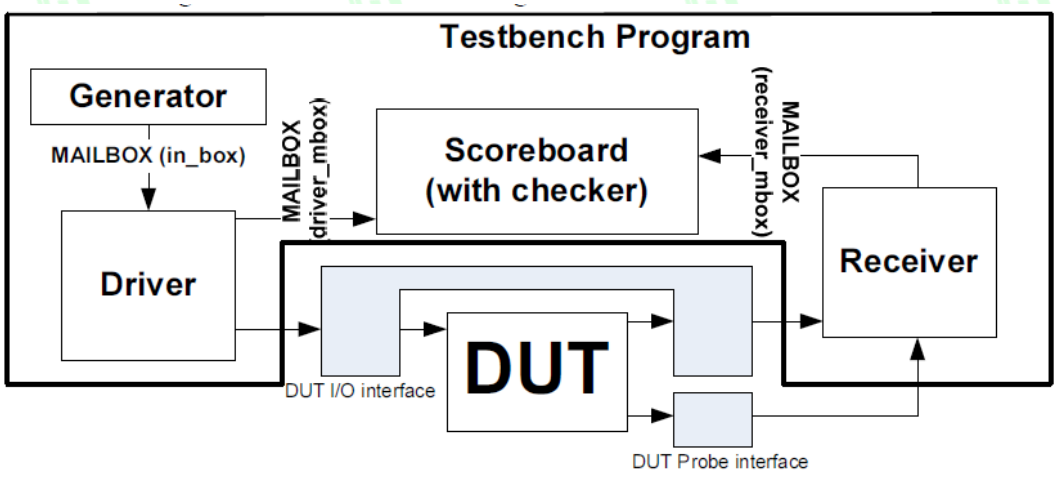
  ```
  cd verification/svv/lab04
  ```

  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. 完善 Testbench 的分层结构

下面，在 lab03 的基础上，继续将 `Execute.tb.sv` 文件当中所包含的剩余功能使用 OOP 的方式将它们分离出来，进一步完善 Testbench 的分层结构，其结构如下图所示：



其中：

- Generator：事务（激励）发生器，将 gen()任务封装入该类中，实现激励的产生功能；
- Driver：激励驱动器，将 send()任务封装入该类中，实现激励的驱动功能，在驱动完激励之后，将该激励通过 mailbox 发送给 Scoreboard；
- Receiver：DUT 输出信号接收器，将 recv()任务封装入该类中，负责捕获 DUT 的输出信号，包括预处理器的输出信号和 ALU 的输出信号，并将捕获的信号通过 mailbox 发送给 Scoreboard；
- Scoreboard：计分板，完善 lab03 中的 Scoreboard，将 check()任务封装入该类中，负责模拟 DUT 的行为以及结果比对功能；
- mailbox：实现 Generator 与 Driver，Driver 与 Scoreboard，Receiver 与 Scoreboard 之间的事务传递。

2. 输出事务抽象类 – OutputPacket

在该实验中，我们将 DUT 的输出信号也抽象为一个事务类。

```
class OutputPacket;

    string  name;
    reg [`REGISTER_WIDTH-1:0]   aluout;
    reg                         carry_out;
    reg                         mem_write_en;
    reg [`REGISTER_WIDTH-1:0]   mem_data_write_out;

    reg [`REGISTER_WIDTH-1:0]   aluin1;
    reg [`REGISTER_WIDTH-1:0]   aluin2;
    reg [2:0]                   opselect;
    reg [2:0]                   operation;
    reg [4:0]                   shift_number;
    reg                         enable_shift;
    reg                         enable_arith;
    reg                         enable;

    extern function new(string name = "OutputPacket");
endclass

function OutputPacket::new(string name = "OutputPacket");
    this.name = name;
endfunction
```


在 OutputPacket 类中，包含了 DUT 顶层的输出信号和预处理器的输出信号，每当 Receiver 捕获到一组 DUT 的输出，这些信号都会被存储在该类的一个对象中，做为某一个激励的结果输出。

3. DUT 输出信号接收器 – Receiver

用于实现 recv()任务的功能。

与 Driver 类似，在构造 Receiver 类时，先构造了一个基类 ReceiverBase。ReceiverBase 是一个基础类，里面实现了一个 Receiver 所要实现的最基本的功能；

而 Receiver 则是针对某一种可能情况而扩展出来的类，Receiver 具有 ReceiverBase 所具有的所有功能。该类稍微有点复杂，这里只截取需要说明的部分：

```
class ReceiverBase;

    virtual   Execute_io.TB Execute;
    virtual   DUT_probe_if  Prober;
    ......

    extern function new(string   name = "DriverBase",
                        virtual Execute_io.TB Execute,
                        virtual DUT_probe_if  Prober);

    extern virtual task recv();
    extern virtual task get_payload();
    extern virtual task get_memdout();

endclass
```

虚接口，它是 DUT 和 Receiver 实例进行信息交互的接口。注意，这里必须是 virtual 类型的。

在 Program 中将实际的接口通过 Receiver 的 new 函数传递进来。

这些任务用于实现对 DUT 输出信号的捕获功能。

ReceiverBase 中包含了 recv()、get_payload() 以及 get_memdout() 三个任务，并且可以在实例化对象时，通过 new 函数将实际的 interface 传递进来。需要注意的是，在类中不能直接使用 interface，而是要通过 virtual interface 与实际的 interface 建立对应关系，间接的引用 interface 中的相关信号。

Receiver 是从 ReceiverBase 基类扩展而来的，它包含了 ReceiverBase 的所有功能，实现代码如下：

```
`include "ReceiverBase.sv"

class Receiver extends ReceiverBase;

    typedef mailbox #(OutputPacket) rx_box_type;
    rx_box_type  rx_out_box;

    extern function new(string name = "Receiver",
                        rx_box_type rx_out_box,
                        virtual Execute_io.TB  Execute,
                        virtual DUT_probe_if   Prober);
    extern virtual task start();
endclass

function Receiver::new(string name = "Receiver",
                       rx_box_type rx_out_box,
                       virtual Execute_io.TB Execute,
                       virtual DUT_probe_if Prober);
```

扩展自 ReceiverBase

该 mailbox 用于 Receiver 与 Scoreboard 通信

mailbox 做为 new 函数的一个参数

```
  super.new(name, Execute, Prober);
  this.rx_out_box = rx_out_box;
endfunction

task Receiver::start();
  $display($time, "ns:  [RECEIVER]  RECEIVER STARTED");
  @(Execute.cb);
  fork
    forever begin
      get_memdout();
      @(Execute.cb);
      recv();
      rx_out_box.put(pkt_cmp);
      $display($time, "ns:  [RECEIVER -> GETPAYLOAD]  Payload Obtained");
    end
  join_none

endtask
```

在对 DUT 的输出进行采样之前，要等待一个时钟周期。（即在激励驱动到接口的下一个周期开始采样输出）

使用 fork join_none 来启动多个 recv()进程，每过一个时钟周期启动

Receiver 中使用了一个 rx_out_box，为了将捕获到的 outputpacket 发送给 Scoreboard。recv()任务调用 get_payload 任务来从 DUT 的输出端口采集输出信号。Start()任务是调用 Receiver 功能的入口，在 Testbench 的顶层会调用该任务来启动 Receiver。

4. 计分板 – Scoreboard

下面将对 Scoreboard 的功能进行完善，Scoreboard 将会实现以下功能：

- 从 Driver 获取驱动给 DUT 的输入激励
- 从 Receiver 获取从 DUT 捕获到的输出
- 根据 Driver 送来的激励，模拟 DUT 的功能行为，将这些激励计算出对应的结果，该结果称为“期望值”
- 将从 Receiver 送来的 DUT 的输出与自己计算的结果做对比，判断 DUT 的功能是否正确

```
class Scoreboard;
    ……

    typedef mailbox #(Packet) out_box_type;
    out_box_type driver_mbox;

    typedef mailbox #(OutputPacket) rx_box_type;
    rx_box_type   receiver_mbox;
```

该 mailbox 用于 Driver 与 Scoreboard 通信

该 mailbox 用于 Receiver 与 Scoreboard 通信

```
    extern function new(string name = "Scoreboard",
                        out_box_type driver_mbox = null,
                        rx_box_type receiver_mbox = null);
    extern virtual task start();
    extern virtual task check();
    extern virtual task check_memwrite();
    extern virtual task check_arith();
    extern virtual task check_shift();
    extern virtual task check_memread();
    extern virtual task check_preproc();

endclass

.....

task Scoreboard::start();

  .....

  fork
    forever begin

      while(receiver_mbox.num() == 0) begin
        $display ($time, "ns:  [SCOREBOARD] Waiting for
                  Data in Receiver Outbox to be populated");
        #`CLK_PERIOD;
      end

      while (receiver_mbox.num()) begin
        $display ($time, "ns:  [SCOREBOARD] Grabbing Data
                          From both Driver and Receiver");
        receiver_mbox.get(pkt_cmp);
        driver_mbox.get(pkt_sent);
        check();
      end

    end

  join_none

  $display ($time, "[SCOREBOARD] Forking of Process Finished");
endtask

.....
```

不断的查询 receiver_mbox 中是否有事务，如果没有继续查询，如果有则跳出循环。

如果 mbox 中有事务，则取出来进行检查。check()执行检查过程，该任务将会调用 check_arith()、check_preproc()、check memread()以及 check preproc()

在 Scoreboard 中，依然使用 start()任务做为启动 Scoreboard 的入口，并且其

中的代码均包含在 fork join_none 中。**.num** 是 mailbox 内建的一个函数，用于返回 mailbox 中所存储的事务的个数，通过这个函数可以很容易的知道一个 mailbox 中是否有可用的数据。

认真阅读其他代码，弄清它们的功能。

5. 实例化组件并启动进程

将 Receiver 在 Testbench 的顶层中个创建一个对象，在 Execute.tb.sv 文件中：


```
program Execute_test (Execute_io.TB Execute,
                      DUT_probe_if Prober);
  parameter   reg_wd     =   `REGISTER_WIDTH;

  Generator   generator;// generator object
  Driver      drvr;     // driver objects
  Scoreboard  sb;       // scoreboard object
  Receiver    rcvr;     // Receiver Object
  ……
  initial begin
    number_packets =50;
    generator = new("Generator", number_packets);
    sb = new();
    drvr = new("drvr[0]", generator.in_box,
                sb.driver_mbox, Execute);
    rcvr = new("rcvr[0]", sb.receiver_mbox,
                Execute, Prober);
    reset();
    generator.start();
    drvr.start();
    sb.start();
    rcvr.start();
    repeat(number_packets+1) @(Execute.cb);
  end
    ……
endprogram
```

在顶层实例化 Receiver

分别调用 Generator、Driver、Scoreboard 以及 Receiver 的 start 方法来启动它们，由于都是包含在 fork join_none 中，所以它们所启动的进程是并行执行的。

到此为止，我们将 Testbench 的功能都使用 OOP 的方式分离出来，构成了分层的 Testbench 结构。该结构具有一个重要的特点：可重用性。只要调用这些功能的接口不变(start 任务)，想要改变 Testbench 的行为，只需要改变相应层次的行为即可：比如想要产生特定的激励，只需要改变 Generator 即可。

6. 修改脚本文件 Makefile

按照下图所示，修改 Makefile 文件中的第 10 行：

```
all: create_lib compile simulate


create_lib:
  vlib work

compile:
  vlog -l comp.log -mfcu -sv data_defs.v Packet.sv OutputPacket.sv Driver.sv Scoreboard.sv Generator.sv Receiver.sv
Ex_Preproc.vp Arith_ALU.vp Shift_ALU.vp ALU.vp Top.v Execute.if.sv Execute.tb.sv Execute.test_top.sv

simulate:
  vsim -l sim.log -novopt Execute_test_top -do "run -all"

clean:
  rm -rf work mti_lib transcript modelsim.ini *.log *.wlf
```


修改完毕之后，运行仿真，观察实验结果。
（注意，执行之前请先执行“make clean”清理工作环境）

实验问题：

请将以下问题以回帖的方式在论坛上作答。

1. 简述 Receiver 的 start()任务所实现的功能。

   答：

2. 画出该 Testbench 的结构示意图，并指出各个组件所对应的 Layer。

   答：

3. 简单的描述一下一个事务从产生到最后结果检查的数据流（数据在 DUT 以及平台组件之间的传递和处理）。

   答：

4. 将 Execute.tb.sv 文件中的第 25 行代码注释掉
   （***repeat(number_packets+1) @(Execute.cb);***）
   执行仿真之后观察仿真结果，并解释出现该结果的原因。

   答：