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 – Lab03_Guide

实验简介：

将顶层 Testbench 中的部分功能分离出来，并且它们分别封装在不同类中，在 Testbench 中例化出这些类的对象，调用这些对象的任务来实现相应的功能，这样的 Testbench 结构成为“分层结构”。这样的 Testbench 更容易搭建，并且使用和维护起来也更加的方便。
在分层的 Testbench 中，各个层次之间需要进行事务传递，Mailbox 是实现这一过程的重要角色，它不仅能够传递事务，而且在一定程度上同步进程。

实验目的：

★ 将 Execute.tb.sv 中的部分功能分离出来，并将其封装到类中来实现，构建一个具有分层结构的 Testbench；
★ 熟练掌握在 Testbench 中使用 Mailbox 来实现不同组件之间事务传递。

实验准备：

➢ 进入本次实验的实验目录

```
cd verification/svv/lab03
```

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。

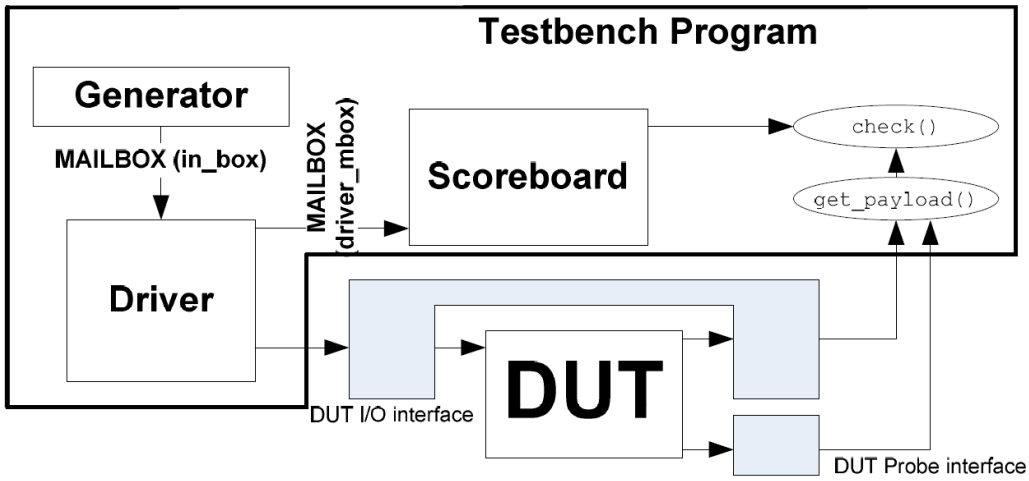
实验步骤：

1. 分层的 Testbench

在先前的两个实验中，Execute.tb.sv 文件当中包含了 Testbench 的所有内容：激励的产生，激励的发送，结果的收集以及最后结果的检查。这些功能分别是通过几个任务来实现的：

- 激励的产生 – gen()
- 激励的发送 – send()
- 结果的收集 – recv()
- 结果的检查 – check()

现在，我要将这些其中的部分功能从 Execute.tb.sv 中分离出来，将它们封装到不同的类中，以形成一个具有分层结构的 Testbench，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；
- Scoreboard：计分板，寄存所有发送给 DUT 的事务；
- mailbox：实现 Generator 与 Driver，Driver 与 Scoreboard 之间的事务传递。

除此之外，我们这次在 Testbench 中加入内部信号的监测功能，可以通过 DUT Probe interface 将预处理器的输出信号采集下来，

2. 事务发生器 – Generator

在 lab02 中，我们使用 gen 任务做为产生激励的事务发生器。在本次实验中，将 gen 任务从 Testbench 的顶层抽取出来，将它的功能封装到 Generator 类中来实现。

```
class Generator;
    string  name;
    Packet  pkt2send;

    typedef mailbox #(Packet) in_box_type;
    in_box_type in_box;

    int     packet_number;
    int     number_packets;
    extern function new(string name = "Generator", int number_packets);
    extern virtual task gen();
    extern virtual task start();
endclass

function Generator::new(string name = "Generator", int number_packets);
    this.name = name;
    this.pkt2send = new();
    this.in_box = new;
    this.packet_number = 0;
    this.number_packets = number_packets;
endfunction
```

定义一个 mailbox，传递的数据类型是 Packet。in_box_type 是一个 mailbox 的类型，in_box 是一个 mailbox 的指针。使用该 mailbox 实现 Generator 与 Driver 的事务传递。

注意，mail_box 在使用之前需要调用 new 创建一个实例。

```
task Generator::gen();

    pkt2send.name = $psprintf("Packet[%0d]", packet_number++);
    if (!pkt2send.randomize())
    begin
        $display(" \n%m\n[ERROR]%0d gen(): Randomization Failed!", $time);
        $finish;
    end
    pkt2send.enable = $urandom_range(0,1);
endtask

task Generator::start();
     $display ($time, "ns:  [GENERATOR] Generator Started");
     fork
         for (int i=0; i<number_packets || number_packets <= 0; i++)
         begin
             gen();
             begin
                 Packet pkt = new pkt2send;
                 in_box.put(pkt); // FUNNY ..
             end
         end
         $display($time, "ns:   [GENERATOR] Generation Finished Creating %d
Packets  ", number_packets);
     join_none
endtask
```

Generator 的所有功能都包含在 start 任务当中，这里启动两个并行进程：一个用于随机化产生激励对象，另一个用于将激励放入 mailbox 中。

Generator 将使用 Mailbox 来完成与其他组件的通信工作。Mailbox 就像是一个队列，可以将数据放入其中，也可以从其中取出数据。但 Mailbox 的功能不仅仅于此，在 Mailbox 满的情况下，如果试图向其中添加数据，此时该进程会阻塞在此处，直到 Mailbox 中有空间存放数据时才会得到释放而继续进程；同样的，如果试图从一个空的 Mailbox 中取数据，则进程同样会被阻塞，直到 Mailbox 中有可用数据时才会得到释放而继续进程。正是由于 Mailbox 的这个特性，才使得 Mailbox 可以将两个进程同步起来。

在声明 new 函数时，我们为其传入了一个参数 number_packets，用该变量来控制产生多少个 packet。

Mailbox 是一种对象，在使用之前需要调用 new 函数。

start()任务是用来启动 Generator 功能的入口，也就是说如果想要使用 Generator 的功能，只要调用 start()任务就可以了。该任务内部主体使用了 fork join_none，表示该任务将是一个并行的线程。

3. 激励驱动器 – Driver

Driver 用于实现 send()任务的功能。

在构造 Driver 类时，先构造了一个基类 DriverBase。

DriverBase 是一个基础类，里面实现了一个 Driver 所要实现的最基本的功能；而 Driver

则是针对某一种可能情况而扩展出来的类，Driver 具有 DriverBase 所具有的所有功能。

```
class DriverBase;
    virtual   Execute_io.TB Execute;
    string    name;
    Packet    pkt2send;

    reg [6:0]                    payload_control_in;
    reg [`REGISTER_WIDTH-1:0]    payload_src1, payload_src2;
    reg [`REGISTER_WIDTH-1:0]    payload_imm, payload_mem_data;
    reg                          payload_enable;

    extern function new(string name = "DriverBase", virtual
Execute_io.TB Execute);
    extern virtual task send();
    extern virtual task send_payload();

endclass

function DriverBase::new(string name = "DriverBase", virtual
Execute_io.TB Execute);
    this.name    = name;
    this.Execute = Execute;
endfunction

task DriverBase::send();
    send_payload();
endtask

task DriverBase::send_payload();
    $display($time, "ns:  [DRIVER] Sending Payload Begin");

    Execute.cb.src1              <= payload_src1;
    Execute.cb.src2              <= payload_src2;
    Execute.cb.imm               <= payload_imm;
    Execute.cb.mem_data_read_in  <= payload_mem_data;
    Execute.cb.control_in        <= payload_control_in;
    Execute.cb.enable_ex         <= payload_enable;

endtask
```

虚接口，它是 DUT 和 Driver 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 Driver 的 new 函数传递进来。

将通过 new 函数传递进来的实际的 interface 与本地的 virtual interface 建立关联。这样引用 virtual interface 中的信号，就等同于引用实际的 interface 中的信号。

将激励通过 interface 驱动给 DUT 的输入端口。

DriverBase 中包含了 lab02 中的 send_payload()和 send()两个任务，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。

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

```
`include "DriverBase.sv"
class Driver extends DriverBase;

  typedef mailbox #(Packet) in_box_type;
  in_box_type in_box = new;
  typedef mailbox #(Packet) out_box_type;
  out_box_type out_box = new;

  extern function new(string name = "Driver", in_box_type in_box,
                      out_box_type out_box, virtual Execute_io.TB Execute);
  extern virtual task start();
endclass

function Driver::new(string name= "Driver", in_box_type in_box,
                     out_box_type out_box, virtual Execute_io.TB Execute);
  super.new(name, Execute);
  this.in_box = in_box;
  this.out_box = out_box;
endfunction

task Driver::start();
    reg [6:0]   control_in_temp;
    int get_flag = 10;
    int packets_sent = 0;
    $display ($time, "ns:  [DRIVER] Driver Started");
   fork
       forever
       begin
            in_box.get(pkt2send); // grab the packet in the q
            packets_sent++;
            control_in_temp = {pkt2send.operation_gen,
                               pkt2send.immp_regn_op_gen, pkt2send.opselect_gen};
            $display ($time, "[DRIVER] Sending in new packet BEGIN");
            this.payload_control_in = control_in_temp;
            this.payload_src1 = pkt2send.src1;
            this.payload_src2 = pkt2send.src2;
            this.payload_imm =  pkt2send.imm;
            this.payload_mem_data = pkt2send.mem_data;
            this.payload_enable = pkt2send.enable;

            send();
            out_box.put(pkt2send);
```

扩展自 DriverBase

in_box 用于 Generator 与 Driver 的通信

out_box 用于 Scoreboard 与 Driver 的通信

Driver 的 new 函数重载了基类 DriverBase 的 new 函数。参数包含了两个 mailbox 和 virtual interface。

增加 start()任务，将 Driver 中的所有功能包含进来。

将刚刚发送的 packet 存入 out_box 中，最后将会传入 Scoreboard 中去。

```
            if(in_box.num() == 0)
            begin
                break;
            end
            @(Execute.cb);
        end
    join_none
endtask
```


在 Driver 的 new 函数中，调用了 super.new()，意思是调用父类的 new()函数，也就是 DriverBase 的 new()函数。

Driver 中使用了 in_box 和 out_box 两个 mailbox，in_box 为了实现从 Generator 中获取 packet，out_box 为了实现将 packet 发送给 Scoreboard。

与 Generator，Driver 中提供了 start()任务作为调用 Driver 全部功能的入口，并且 start()任务主体包含在 fork join_none 中，将会启动为一个并行的线程。

4. 计分板 – Scoreboard

```
class Scoreboard;
    string   name;              // unique identifier

    typedef mailbox #(Packet) out_box_type;
    out_box_type driver_mbox;     // mailbox for Packet objects
from Drivers

    extern function new(string name = "Scoreboard", out_box_type
driver_mbox = null);
endclass

function    Scoreboard::new(string    name    =    "Scoreboard",
out_box_type driver_mbox = null);
    this.name = name;
    if (driver_mbox == null)
        driver_mbox = new();
    this.driver_mbox = driver_mbox;
endfunction
```

该计分板的功能比较的简单，它仅仅包含了一个用于和 Driver 通信的 mailbox。但需要注意的一点是，虽然这里的 Scoreboard 没有实际的功能，但在以后的学习中会逐渐的完善它。

5. 接口 – interface

该实验需要采集 DUT 内部的预处理器的输出信号，这里为 interface 添加新的内容：

```
interface DUT_probe_if(
    input bit clock,
    input logic [`REGISTER_WIDTH-1:0]  aluin1,
```

```
    input logic [`REGISTER_WIDTH-1:0]  aluin2,
    input logic [2:0]        opselect,
    input logic [2:0]        operation,
    input logic [4:0]        shift_number,
    input logic              enable_shift,
    input logic              enable_arith
    );

    clocking cb @(posedge clock);
    default input #1 output #1;

        input  aluin1;
        input  aluin2;
        input  opselect;
        input  operation;
        input  shift_number;
        input  enable_shift;
        input  enable_arith;
    endclocking
endinterface
```

在 Execute.test_top.sv 文件中，将预处理器的输出信号从 DUT 当中拉出来连接到 interface 上：

```
DUT_probe_if DUT_probe(
    .clock(SysClock),
    .aluin1(dut.aluin1),
    .aluin2(dut.aluin2),
    .opselect(dut.opselect),
    .operation(dut.operation),
    .shift_number(dut.shift_number),
    .enable_shift(dut.enable_shift),
    .enable_arith(dut.enable_arith)
    );
```

同样的，要将这个 interface 与 Testbench 连接起来：

```
// 在 Execute.test_top.sv 文件中
Execute_test test(top_io, DUT_probe);
// 在 Execute.tb.sv 文件中
program Execute_test(Execute_io.TB Execute, DUT_probe_if Prober);
```

原来的 Testbench 顶层（Execute_test）变成了两个 interface，其中 Execute 是 DUT 外部的信号接口，而 Prober 是 DUT 内部预处理输出信号的接口。

6. 实例化组件并启动进程

将 Generator、Driver 以及 Scoreboard 在 Testbench 的顶层中个创建一个对象，这些对象就是 Testbench 的组件，在 Execute.tb.sv 文件中：

```
Generator generator; // generator object
```

```
Driver drvr;          // driver objects
Scoreboard sb;        // scoreboard object
initial begin
    number_packets = 21;
    generator = new("Generator", number_packets);
    sb = new(); // NOTE THAT THERE ARE DEFAULT
                // VALUES FOR THE NEW FUNCTION
    // FOR THE SCOREBOARD
    drvr = new("drvr[0]", generator.in_box,
                sb.driver_mbox, Execute);
    reset();
    generator.start();
    drvr.start();
    fork
        recv();
    join
    repeat(number_packets+1) @(Execute.cb);
end
```

在使用之前要先实例化对象

启动 Generator 和 Driver 的功能，只需要调用它们的 start()任务即可。

值得注意的是，尽管 generator.start(); 和 drvr.start(); 语句写在 recv() 语句之前，但它们三者之间是并行执行的，原因在于 generator.start(); 和 drvr.start();的函数主体是包含在 fork join_none 中。

7. 结果收集与检查 – recv()

本实验中结果的收集与检查是通过 recv()实现的，其过程与 lab02 大致相同，除了包含了之前所以的功能之外，还添加了对预处理器输出结果的检查功能。请认真阅读这部分代码，弄清其代码的含义。

8. 修改脚本文件 Makefile

按照下图所示，修改 Makefile 文件中的第 10 行：

```
1
2
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 Driver.sv Scoreboard.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 Execute_test_top
14
15 clean:
16   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. 请描述一下 fork join/join_any/join_none 三者之间的区别

答：

2. Execute.tb.sv 文件中，generator.start ，drvr.start 以及 recv 三个任务之间的执行关系是什么，并且说明原因。

答：

3. 在 Testbench 中，driver、Generator 以及 Scoreboard 是如何进行事务传递的。

答：

4. 请简要回答 Driver 的 Start() 任务都做了哪些事情。

答：

5. 修改 check() 方法，使它能够对所有的类型的操作进行结果检查。

答：

6. 请多次运行仿真，试图找出 DUT 可能存在的错误。

答：