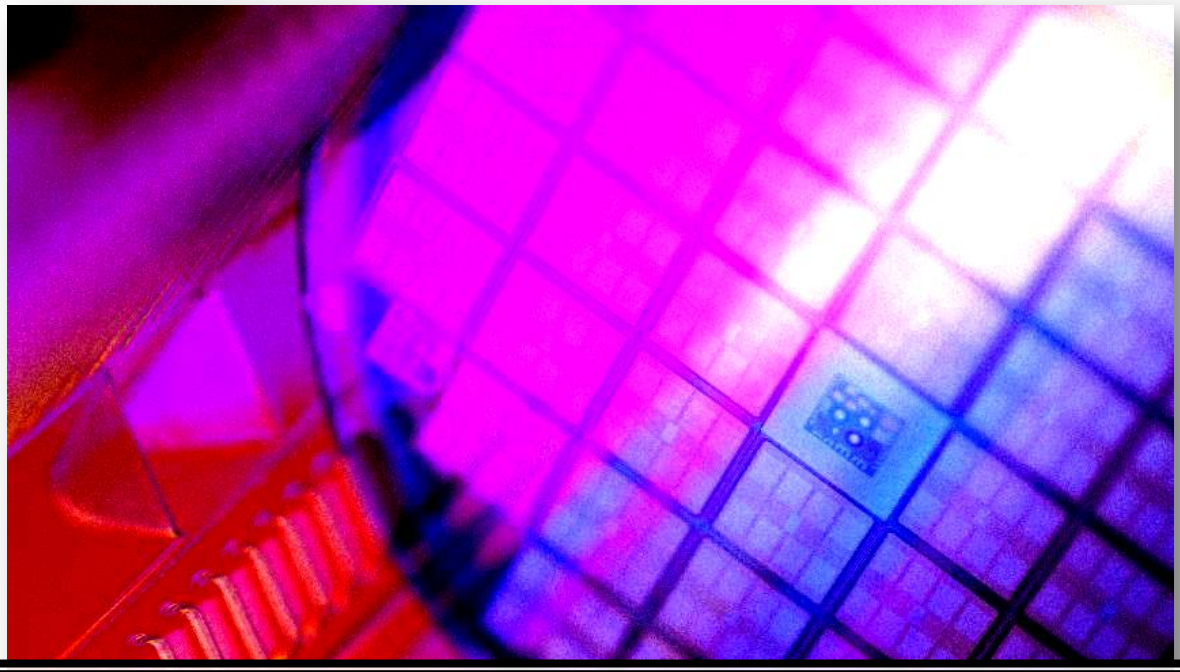
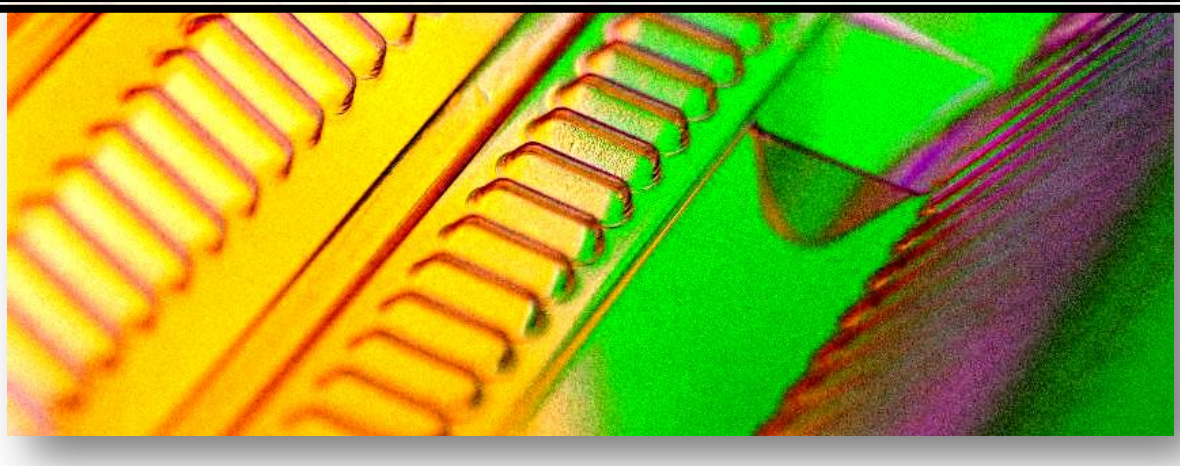
FEC100IT03-V1.0



zrtech

FPGA/CPLD 开发套件实验教程

--仿真，调试，设计篇



WWW.ZR-TECH.COM

实验二、modelsim 仿真指南（二）

实验目的：

本节给大家介绍testbench的编写方法，使用户掌握采用modelsim验证电路的功能与性能是否与预期的设计相符。

实验原理：

1． Testbench简介

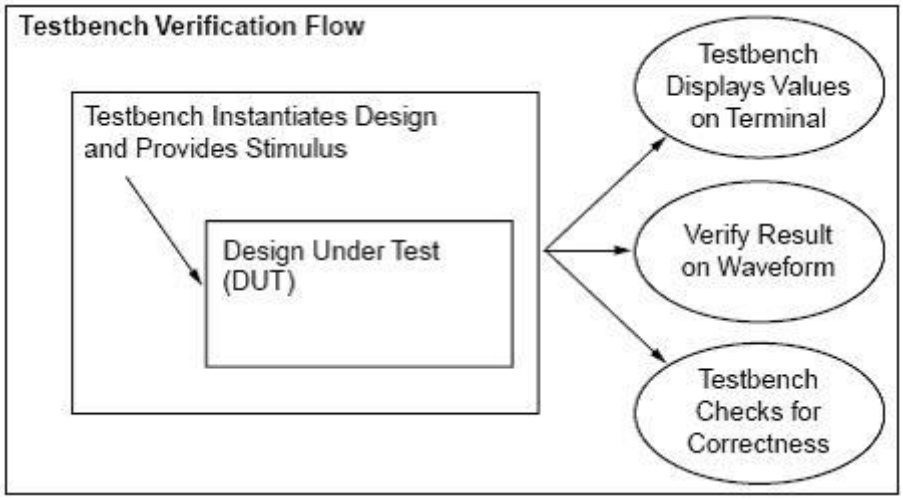
由于设计的规模越来越大也越来越复杂，数字设计的验证已经成为一个日益困难和繁琐的任务。验证工程师们依靠一些验证工具和方法来应付这个挑战。对于几百万门的大型设计，工程师们一般使用一套形式验证（formal verification）工具。然而对于一些小型的设计，设计工程师常常发现用带有 testbench 的 HDL 仿真器就可以很好地进行验证。

Testbench 已经成为一个验证高级语言(HLL --High-Level Language) 设计的标准方法。通常 testbench 完成如下的任务：

- 实例化需要测试的设计（DUT）；
- 通过对 DUT 模型加载测试向量来仿真设计；
- 将输出结果到终端或波形窗口中加以视觉检视；
- 另外，将实际结果和预期结果进行比较。

通常 testbench 用工业标准的 VHDL 或 Verilog 硬件描述语言来编写。Testbench 调用功能设计，然后进行仿真。复杂的 testbench 完成一些附加的功能——例如它们包含一些逻辑来选择产生合适的设计激励或比较实际结果和预期结果。

下图给出了一个如上所描述步骤的标准 HDL 验证流程。由于 testbench 使用 VHDL 或 Verilog 来描述，testbench 的验证过程可以根据不同的平台或不同的软件工具实现。由于 VHDL 或 Verilog 是公开的通用标准，使用 VHDL 或 Verilog 编写的 testbench 以后也可以毫无困难地重用（reuse）。由于 verilog 在 testbench 的描述能力上要好于 VHDL，我们这里着重介绍 verilog 的 testbench 设计，VHDL 的 testbench 只做简要介绍。



2．为什么要写testbench

经常看到论坛里有人问我们为什么要写 testbench，总是觉得不好回答。与写 testbench 相对应的功能手段还有画波形图，两者相比，画波形图的方法更加直观和易于入门，那为什么我们还要写 Testbench 呢？原因有以下五点：

第一，画波形图只能提供极低的功能覆盖。

画波形无法产生复杂的激励，因此它只产生极其有限的输入，从而只能对电路的极少数功能进行测试；而 testbench 以语言的方式描述激励源，容易进行高层次的抽象，可以产生各种激励源，轻松地实现远高于画波形图所能提供的功能覆盖率。以 PCI 转以太网电路设计为例，该设计并不复杂，但却已经需要考虑多种情况：PCI 的配置读写、存储器读写等操作；以太网的短包、长包等。如果这些激励都用画波形图完成，其工作量是难以想象的；用 testbench 则可以轻松完成这些工作。

第二，画波形无法实现验证自动化。

对于规模设计来说，仿真时间很长，如果一个需要仿真一天设计在检错时仅通过画波形图来观测，将几乎不能检查出任何错误；而 testbench 是以语言的方式进行描述的，能够很方便地实现对仿真结果的自动比较，并以文字的方式报告仿真结果。我们甚至可以在下班时开始仿真，然后第二天早上上班时再查看验证结果。

第三，画波形图难于定位错误。

用画波形图进行仿真是一种原始的墨盒验证法，无法使用新的验证技术；而 testbench 可以通过在内部设置观测点，或者使用断言等技术，快速地定位问题。

第四，画波形的可重用性和平台移植性极差。

如果将一个 PCI 转 100Mb 以太网的设计升级到 PCI 转 1000Mb 以太网，这时原来画的波形图将不得不重新设计，耗费大量的人力物力及时间；但若使用 testbench，只需要进行一些小的修改就可以完成一个新的测试平台，极大地提高了验证效率。

第五，通过画波形的验证速度极慢。

Testbench 的仿真速度比画波形图的方式快几个数量级，在 Quartus 下画波形需半个小时才能跑出来的仿真结果，在 ModelSim 下使用 testbench 可能只需几秒钟就可以完成。

所以，在设计中除了那些极简单的设计（如调用厂商提供的 MegaCore），推荐通过写 testbench 的方法来做功能验证。

2．如何写testbench

由于 testbench 只用来进行仿真，它们没有那些适用于综合的 RTL 语言子集的语法约束限制，而是所有的行为结构都可以使用。因而 testbench 可以编写的更为通用，使得它们可以更容易维护。Testbench 模块没有输入输出，在 Testbench 模块内例化待测设计的顶层模块，并把测试行为的代码封装在内，直接对测试系统提供测试激励。所有 testbench 包含了如下表的基本程序段。正如上面所提到的，testbench 通常包含附加功能，如在终端上可视的结果和内建的错误检测。

VHDL	Verilog
Entity and Architecture Declaration	Module Declaration
Signal Declaration	Signal Declaration
Instantiation of Top-level Design	Instantiation of Top-level Design
Provide Stimulus	Provide Stimulus

下面是一个基本的 Verilog 语言的 Testbench 结构模块：

```
module testbench;

    // 数据类型声明

    // 对被测试模块实例化

    // 产生测试激励

    // 对输出响应进行收集

endmodule
```

一般来讲，在数据类型声明时，和被测模块的输入端口相连的信号定义为reg类型，这样便于在initial语句和always语句块中对其进行赋值；和被测模块输出端口相连的信号定义为wire类型，便于进行检测。Testbench模块最重要的的任务就是利用各种合法的语句，产生适当的时序和数据，以完成测试，并达到覆盖率要求。此外建议对测试模块实例化使用名称关联的方式。

3．产生时钟信号

使用系统时钟的时序逻辑设计必须产生时钟。时钟信号在 VHDL 或 Verilog 中可以很容易地实现。以下是 VHDL 和 Verilog 的时钟发生示例。

VHDL:

```
-- Declare a clock period constant.
Constant ClockPeriod : TIME := 10 ns;

-- Clock Generation method 1:
Clock <= not Clock after ClockPeriod / 2;

-- Clock Generation method 2:
GENERATE CLOCK: process
begin
wait for (ClockPeriod / 2)
Clock <= ’1’;
wait for (ClockPeriod / 2)
Clock <= ’0’;
end process;
```

Verilog:

```
// Declare a clock period constant.
Parameter ClockPeriod = 10;
// Clock Generation method 1:
initial begin
Clock = 0;
forever Clock = #(ClockPeriod / 2) ~ Clock;
end
// Clock Generation method 2:
always #(ClockPeriod / 2) Clock = ~Clock;
```

注意在 VHDL 中，延时使用 after 或者 wait for，在 verilog 中，采用#号表示延迟，下面的 testbench 着重将介绍 verilog 的书写方式，使用 VHDL 的朋友可以参阅其他相关教材。

3．激励的产生

对于 testbench 而言 ,端口应当和被测试的 module 对应好。input 对应的端口应当申明为 reg, output 对应的端口申明为 wire，Verilog 的 initial 块与文件中的其他 initial 块是同时执行。然而，在每一个 initial 块中，事件是按照书写的顺序执行的。这说明在每一个并行块中的激励序列从序仿真时间零点开始。为了代码有更好的可读性和更方便的可维护性，应采用多个块来分割复杂的测试激励。

1）直接赋值。
一般用 initial 块给信号赋初值，initial 块执行一次，always 或者 forever 表示由事件激发反复执行。

举例，一个 module

```
module exam();
reg rst_n,clk,data;
initial
begin
        clk=1'b0;
        rst=1'b1;
        #10
        rst=1'b0;
        #500
        rst=1'b1;
end
always
begin
        #10
             clk=~clk;
end
```

同前所述， #号的意思是指延迟相应的时间单位。该时间单位由 timscale 决定.一般在 testbench 的开头定义时间单位和仿真精

度，比如`timescale 1ns/1ps，前面一个是代表时间单位，后面一个代表仿真时间精度。以上面的例子而言，一个时钟周期是 20 个单位，也就是 20ns。而仿真时间精度的概念就是，你能看到 1.001ns 时对应的信号值，而假如 timescale 1ns/1ns，1.001ns 时候的值就无法看到。对于一个设计而言，时间刻度应该统一，如果设计文件和 testbench 里面的时间刻度不一致，仿真器默认以 testbench 为准。一个较好的办法是写一个 global.v 文件，然后用 include 的办法，可以防止这个问题。

2）构成任务

对于反复执行的操作，可写成task,然后调用，比如

```
task load_count;
        input [3:0] load_value;
        begin
            @(negedge clk_50);
                    $display($time, " << Loading the counter with %h >>", load_value);
            load_l = 1'b0;
            count_in = load_value;
            @(negedge clk_50);
            load_l = 1'b1;
        end
endtask //of load_count

initial
begin
   load_count(4'hA);    // 调用 task
end
```

其他像forever,for,function等等语句用法类似，虽然不一定都能综合，但是用在testbench里面很方便，大家可以自行查阅参考文档。

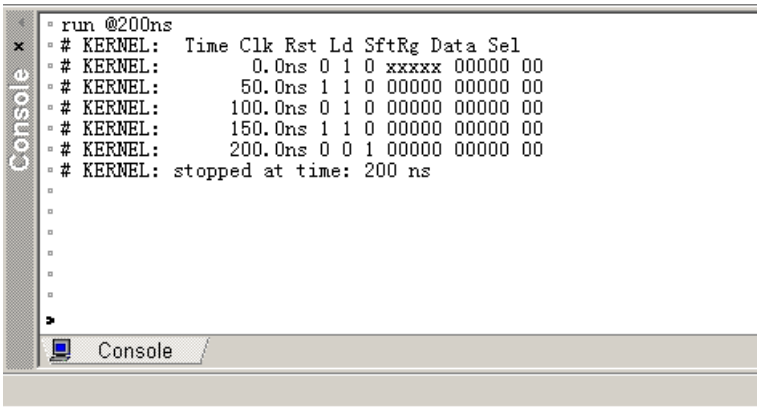
4.显示仿真结果

在Verilog中可以非常方便地使用系统函数$display( )和$monitor( )显示结果。VHDL没有等效的显示指令，它提供了std_textio标准文本输入输出程序包。下面是verilog示例,它将在终端屏幕上显示一些值。

```
// pipes the ASCII results to the terminal or text editor
initial begin
$timeformat(-9,1,"ns",12);
$display(" Time Clk Rst Ld SftRg Data Sel");
$monitor("%t %b %b %b %b %b %b", $realtime, clock, reset, load, shiftreg, data, sel);
end
```

系统函数$display在终端屏幕上输出引用的附加说明文字（“。。。”）。$write 与$display类似，其区别是$display显示完后自动换行，$write不换行。系统函数$monitor操作不同。因为它的输出是事件驱动的。例中的变量$realtime（由用户赋值到当前的仿真时间）用于触发信号列表中值的显示。信号表由变量$realtime开始，跟随其他将要显示的信号名（clock, reset, load 等）。以%开始的关键字包含一个格式描述的表，用来控制如何格式化显示信号列表中的每个信号的值。格式列表是位置确定的。每个格式说明有序地与信号列表中的信号顺序相关。比如%t说明规定了$realtime的值是时间格式。并且第一个%b说明

符格式化clock的值是二进制形式。Verilog提供附加的格式说明，比如%h用于说明十六进制，%d说明十进制，%c说明显示为八进制。下图说明格式显示结果



4.文件输入与输出

有时候，需要大量的数据输入，直接赋值的话比较繁琐，可以先生成数据，再将数据读入到寄存器中，需要时取出即可。用 $readmemb 系统任务从文本文件中读取二进制向量(可以包含输入激励和输出期望值)。$readmemh 用于读取十六进制文件。例如：

```
treg [7:0]    mem[1:256]    //    a 8-bit, 256-word 定义存储器 mem

initial    $readmemh ( "E:/readhex/mem.dat", mem ) //  将.dat 文件读入寄存器 mem 中

initial    $readmemh ( "E:/readhex/mem.dat", mem, 128, 1 ) //  参数为寄存器加载数据的地址始终
```

亦可以将仿真的结果采用文件输出，
设计中的信号值可以通过$fmonitor，$fdisplay,$fwrite。其中$fmonitor 只要有变化就一直记录，$fdisplay 和$fwrite 需要触发条件才记录
例子：

```
integer file_id;    // file_id 是一个文件描述，需要定义为 integer 类型
out_file = $fopen ( " cpu.data " ); // cpu.data 是需要打开的文件，也就是最终的输出文本
initial begin
    $fmonitor(file_id, "%m: %t in1=%d o1=%h", $time, in1, o1);
end

always@(a or b)
begin
    $fwrite(file_id,"At time%t a=%b b=%b",$realtime,a,b);
end
```

实验结果：

利用 modelsim 对计数器的 Verilog 测试程序功能仿真，并将仿真结果打印在屏幕上，并存储到文件。

具体步骤：

1、先在 Quartus II 里生成一个例子程序

具体步骤略，可参考 Quartus 的具体步骤，例程代码见 counter..v。

```
module counter (clk, reset, enable, count);

input clk, reset, enable;
output [3:0] count;
reg [3:0] count;

always @ (posedge clk)
if (reset == 1'b1)
    count <= 0;
else if ( enable == 1'b1)
    count <= count + 1;
end
endmodule
```

2．编写对应的testbench文件

```
module counter_test;
reg clk, reset, enable;
wire [3:0] count;
integer file_id;
out_file = $fopen ( " counter.data " );

counter counter _t(
.clk (clk),
.reset     (reset),
.enable (enable),
.count     (count)
);

initial begin
    clk = 0;
    reset = 0;
    enable = 0;
end

always
    #5     clk =     ! clk;

initial     begin
    $display("\t\ttime,\tclk,\treset,\tenable,\tcount");
    $fdisplay(file_id, "\t\ttime,\tclk,\treset,\tenable,\tcount");
    $monitor("%0d,\t%0b,\t%0b,\t%0b,\t%0d",$time, clk,reset,enable,count);
    $fmonitor(file_id, "%0d,\t%0b,\t%0b,\t%0b,\t%0d",$time, clk,reset,enable,count);
end

initial
      #100     $finish;
endmodule
```

从以上测试代码可以看出，我们不但对计数器进行了测试，而且将测试的结果打印在显示器，且存在了文件" counter.data " 中

3．打开modelsim进行功能仿真

ModelSim 流程，我们再复习一下：

- 直接选择工程目录下的源文件与测试文件进行编译
- 启动仿真器，指定顶层设计单元
- 添加待观察信号

- 查看和调试结果

这些步骤和上一节完全相同，我们就不再赘述了。

实验总结：

编写 Testbench 的目的是对硬件描述语言写的电路进行仿真，从而验证电路的功能与性能是否与预期的设计相符。对于复杂时序电路的仿真，modelsim 加测试激励要比采用 vwf 波形仿真高效很多。testbench 刚开始写起来估计会有些不太习惯，但是多写写就会熟练很多了。如果有的朋友实在不想编写测试激励，却想体验一下 modelsim 的强大功能，也可以在 quartus 中将 vwf 转换成对应的测试激励文件。

课后作业：

将本节实验继续完成下去，实现综合后仿真与时序仿真。观查实验结果。

文档内部编号：FEC1001T03

编号说明：
首一字母：F-FPGA系列
首二字母：L-理论类 E-实验类 T-专题类
首三字母：C-普及类 Q-逻辑类 S-软核类
数字前两位：代表年度
数字后两位：同类文档顺序编号
尾字母/数字：C目录，T正文，数字表示章节号

修订记录

版本号	日期	描述	修改人
1.0	11.1.2	初稿完成	左超