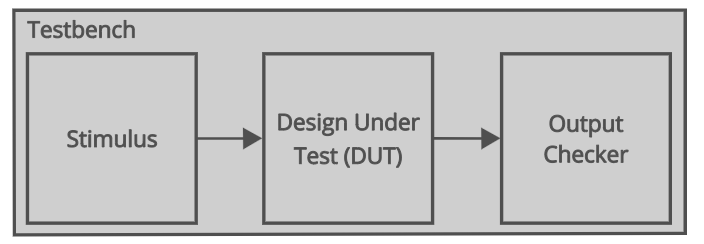
实验3. 简单TestBench编写

1. **实验目的**
2. 学习使用VCS进行systemverilog的编译与波形输出
3. 学习使用verdi进行波形观察
4. 学习Makefile的编写
5. **TestBench编写介绍**
6. 本testbench仅仅包括建模时间、initial块（initial block）和任务（task）。

在使用verilog设计数字电路时，设计人员通常还会创建一个testbench来仿真代码以确保其按预期设计运行。设计人员可以使用多种语言构建testbench，其中最流行的是VHDL、Verilog和System verilog。

testbench由不可综合的verilog 代码组成，这些代码生成被测设计的输入并验证被测设计的输出是否正确（输出是否符合预期）。

 testbench基本模块包括：**激励**（stimulus block）是为FPGA设计生成的输入。**输出校验**（output checker）检查被测模块的输出是否符合预期。**被测模块**（design under test，DUT）即是编写的verilog模块，testbench的主要设计目的就是对其进行验证，以确保在特定输入下，其输出均与预期一致。对于较大规模的设计，激励和输出校验可以位于单独的文件中，也可以将所有这些不同的模块都包含在同一个文件中。



|  |
| --- |
|  |

module Module\_tb ();

// 定义并产生激励信号

// ......

Test\_module #(

// 参数接口

) test (

// 待测模块端口

);

endmodule

1. 例化被测模块

编写testbench的第一步是创建一个verilog模块作为测试的顶层。在这种情况下，设计人员要创建的是一个没有输入和输出的模块。这是因为设计人员希望testbench模块是完全独立的（self contained）。

下面的代码片段展示了一个空模块的语法，这可以被用作testbench。

module ();

//在这里写testbench

endmodule :

创建了一个testbench之后，必须例化被测设计，这可以将信号连接到被测设计以激励代码运行。

下面的代码片段展示了如何例化一个16位加法器（被测模块）。

module module\_tb();

input [15:0] a\_in, b\_in; //输入声明

output [16:0] sum\_out; //输出声明

// 例化一个位宽为16的adder模块

adder #(

.WIDTH(16)

) u\_adder16 (

.a(a\_in), //

.b(b\_in),

.sum(sum\_out)

);

endmodule

完成此操作后，就可以开始将激励写入testbench。激励包括**时钟信号**和**复位信号**，以及创建发送到testbench的测试数据。

1. Verilog中的建模时间（Modelling Time）

testbench代码和设计代码之间的主要区别是testbench并不需要被综合成实际电路，为此可以使用时间控制语句这种特殊结构。事实上，这对于创建测试激励至关重要。

在Verilog中使用 # 字符后跟多个时间单位来模拟延时。例如，下面的verilog代码展示使用延时运算符等待10个时间单位。

#10

这里要注意的是代码末尾并没有分号。

将延时语句写在与赋值相同的代码行中也很常见，这可以有效地行使调度功能，将信号的变化安排在延迟时间之后。下面的代码片段是此种情况的一个示例。

#10 a = 1'b1; // 在10个时间单位后，a将被赋值为1

时间单位（Timescale ）编译指令timescale预编译指令来确定。这只需要在testbench中运行该指令一次，而且应在模块外完成。

下面的代码片段展示用来在 verilog 中指定时间单位和精度的编译指令。

`timescale 1ns/1ps

指定时间的单位，则指定时间精度。

1. Verilog initial block（初始块）

在initial 块中编写的任何代码都会在仿真开始时执行一次且仅执行一次。

下面的 verilog 代码展示了initial 块的一般语法。

initial begin

//这里写代码

end

与always块不同，**在initial块中编写的verilog代码几乎是不可综合的**，因此其几乎只被用于仿真。但在verilog RTL中也可以使用initial块来初始化信号（几乎很少用）。

为了更好地理解如何使用initial块在verilog中编写激励，现在假设想要测试一个基本的两输入与门，为此**需要编写代码来生成所有可能的四种输入**。此外还需要使用**延时运算符以在生成不同的输入之间延迟一段时间**，以允许信号有时间来传播。

下面的 verilog 代码展示了在initial块中编写此测试的方法。

initial begin

// 每隔10个时间单位就生成一个输入

and\_in = 2b'00;

#10 and\_in = 2b'01;

#10 and\_in = 2b'10;

#10 and\_in = 2b'11;

end

1. Verilog Foever 循环（Loop）

在verilog testbench中可以使用一种重要的循环类型——foever循环。

使用这个构造时，实际上是创建了一个无限的循环，这意味着创建了一段在仿真过程中将永远运行的代码。

下面的verilog代码展示了用来编写foever循环的一般语法。

forever begin

// your code goes here

end

当用其他编程语言编写代码时，**无限循环一般被视为应极力避免的严重错误。**但是，verilog与其他编程语言不同，**编写verilog代码是在描述硬件而不是在编写软件。**

因此，至少有一种情况是可以使用无限循环的----时钟信号。为此需要一种定期连续反转信号的方法，foever循环与此相当契合。

下面的 verilog 代码展示了如何使用foever循环在testbench中生成一个时钟。需要注意的是，所编写的任何循环都必须包含在过程块（procedural block）中或生成块（generate块）中。

initial begin

clk = 1'b0;

forever begin

#1 clk = ~clk;

end

end

1. Verilog系统任务（System Tasks）

在 verilog中编写testbench时，有一些**内置的任务和函数**可以提供帮助。这些**被统称为系统任务或系统函数**，这些函数总是以美元符号（$）开头。$display、$monitor和$time。

1. $display

$display 是 verilog 中最常用的系统任务之一。$display的使用方式与C语言中的printf函数非常类似，这意味着设计人员可以轻松地在testbench中创建文本语句，并使用它们来显示有关仿真状态的信息。

1. $monitor

$monitor函数与 $display函数非常相似，但它一般被用来监视testbench的信号值，这些信号中的任何一个改变状态，都会在终端打印一条消息。

1. $time

在testbench中常用的最后一个系统任务是 $time 。这个系统任务可以用来获取当前的仿真时间。

1. $strobe

 $strobe 在一个仿真时间步（time-step）中，仅在当前time-step中**所有的处理**都结束后执行一次。(**观察第一个#100延时两个strobe的输出位置**)

//例子：

`timescale 1ns / 1ps

module sim\_top( );

wire [7:0] y;

reg [7:0] a = 1, b = 2;

assign y = a + b;

initial begin

$monitor("@%0t: $monitor(): a = %d; b = %d; y = %d", $time, a, b, y);

$display("@%0t: $display(): Hello, welcome to SystemVerilog!", $time);

a = 5;

$strobe("@%0t: $strobe(): Hello, welcome to SystemVerilog!", $time);

#100;

$display("@%0t: $display(): a = %d; b = %d; y = %d", $time, a, b, y);

$strobe("@%0t: $strobe(): a = %d; b = %d; y = %d", $time, a, b, y);

a = 9;

$display("@%0t: $display(): a = %d; b = %d; y = %d", $time, a, b, y);

$strobe("@%0t: $strobe(): a = %d; b = %d; y = %d", $time, a, b, y);

#100;

b = 6;

$display("@%0t: $display(): a = %d; b = %d; y = %d", $time, a, b, y);

#100;

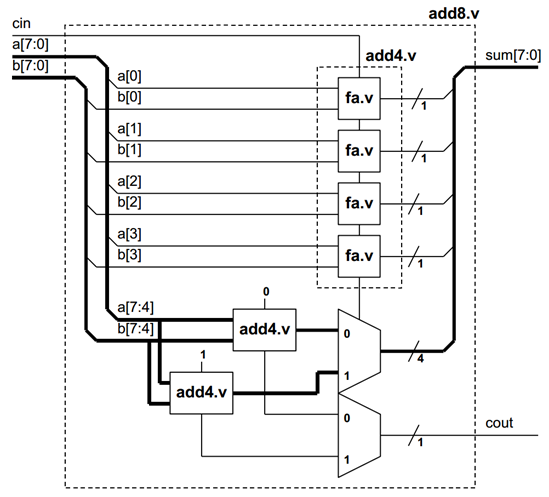
$finish;

end

endmodule

1. Verilog testbench示例

下图所示电路为一位加法器 fa.v （full\_adder）组成4位加法器 add4.v，再组成一个8位加法器，**使用资源换性能的思路，减小了行波进位加法器的进位延迟。**顶层文件为add8.v，testbench为addertb.v。



1. 创建一个testbench模块

在testbench中做的第一件事就是声明一个空模块来写入代码。下面的代码片段展示了此testbench的模块声明。请注意，最好让被测试设计的名称与testbench的名称保持相似。一般可以简单地将 \_tb 或 \_test 附加到被测设计名称的末尾。

module example\_tb ();

//在这里写测试代码

endmodule : example\_tb

1. 例化被测模块

现在只有一个空白的testbench模块，接下来需要例化要测试的设计模块。上图给出的8位全加器。此加法器只需例化一次。

add8 u1(a\_test, b\_test, cin\_test, sum\_test, cout\_test);

其中，

reg [7:0] a\_test, b\_test; //input

reg cin\_test; //input carrier

wire [7:0] sum\_test; //output

wire cout\_test; //output carrier

为了在不同时刻显示出加法的输出，需要在testbench中生成一个时钟和复位信号。可以在initial块中为时钟和复位信号编写代码。

对于时钟信号，可以使用forever关键字在仿真期间持续运行时钟信号。使用此语法将每1ns进行一次反转，从而实现500MHz的时钟频率，选择此频率纯粹是为了实现快速仿真，实际上FPGA 中的 500MHz 时钟速率很难实现，所以testbench的时钟频率应尽量与硬件时钟频率匹配。

下面的 verilog 代码展示了如何在testbench中生成时钟和复位信号。

//生成时钟信号

initial begin

clk = 1'b0;

forever #1 clk = ~clk;

end

//生成复位信号

reset = 1'b1;

#10

reset = 1'b0;

end

1. 编写测试激励信号

最后一部分是编写测试激励。为了测试被测电路，需要依次为加法器所有输入生成所有可能的数据。因此，a\_test取值范围是[0:ff] b\_test取值范围是[0:ff]。一种好的办法是将a\_test，b\_test看成是18bit数据的高低8位。输入的进位位为18bit数据的最高两位。这样可以保证加法器对所有可能的数据都可以进行有效的相加。

for (test = 0; test <= 18'h1ffff; test = test +1) begin

//验证代码

end

1. 编写验证过程

验证过程只需要对比使用例化的加法器的输出与三个输入的数据（包括进位）累加和对比即可。若全部相等则完成验证，若不等则退出并通过任务函数显示出错的位置。

if ({cout\_test, sum\_test} !== (a\_test + b\_test + cin\_test)) begin

$display("\*\*\*ERROR at time = %0d \*\*\*", $time);

$display("a = %h, b = %h, sum = %h; cin = %h, cout = %h",

a\_test, b\_test, sum\_test, cin\_test, cout\_test);

$finish;

end

1. **实验要求**

1）参考以上过程及图片，设计8位全加器及其仿真验证环境。

2）vcs+verdi编译并Dump输出波形。（需要在test bench中dump波形出来）

3）编写Makefile实现仿真验证的自动化。