TestBench 编程指南

如今数字设计的规模变得越来越庞大,设计的复杂程度也越来越高,这就使得设计的验证变得越来越困难,而且费时费力。为了应对这种挑战，验证工程师依靠各种验证工具和方法。对于大型设计，如几百万门的设计，通常采用一整套正式的验证工具。然而，对于小一些的设计,设计工程师发现往往采用带 TestBench 的 HDL 仿真工具是最好的途径。

TestBench 已经变成验证高级语言设计的一种标准的方法。通常，TestBench 执行以下任务:

- 例化设计，使其可测试（DUT－design under test）;
- 通过将测试向量应用到模型来仿真例化后的可测试的设计;
- 将结果输出到终端，或者输出波形窗口;
- 将真实的结果和期望的结果进行比较;

一般，TestBench 采用工业标准的 VHDL 或者 Verilog 硬件描述语言来编写。TestBench 调用功能设计，然后仿真。复杂的测试文件执行附加功能——例如,他们包含逻辑以决定合适的设计激励或者比较真实的结果和期望的结果。

以下章节将讨论一个组织良好的测试文件的组成，以及例举了一个带有自检的测试文件(自动将真实的结果和预期的结果进行比较)。下图是一个标准的 HDL 验证的流程。自从测试文件可以用 VHDL 或者 Verilog 编写以来，测试验证流程就可以在平台和供应商的工具交叉

进行。同时，由于 VHDL 和 Verilog 都是标准的公用的语言，所以用 VHDL 或者是 Verilog 描述的验证可以很简单的被再使用。

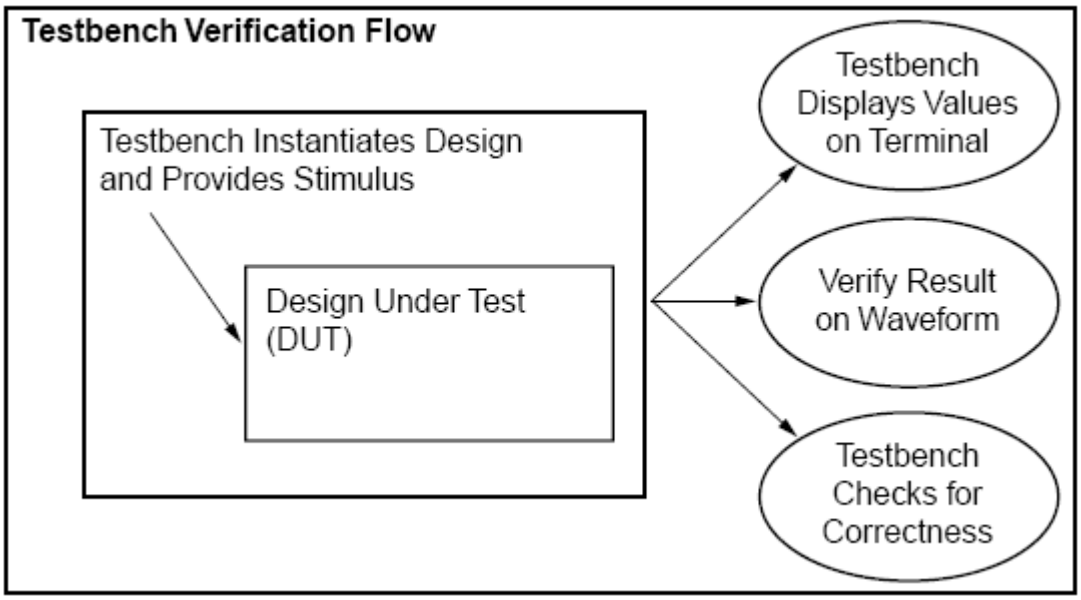


图 1. HDL 验证流程

测试文件构成:

测试文件可以采用 VHDL 或者 Verilog 语言编写。由于测试文件只是用来仿真的，他们就不被用于综合的 RTL 语言子集的语法所约束。相反，所有行为结构都可以被使用。这样，测试文件可以被写的更通用，更易于维护。

所有的测试文件都包含以下基本内容，如表 1。如上所属，测试文件经常同时包含附加功能，如结果的可视化显示和内建错误检测。

表 1

VHDL	Verilog
实体和结构体申明	模块申明
信号申明	信号申明

顶层设计实例化	顶层设计实例化
提供激励	提供激励

以下例子展示了一些在测试文件中被频繁使用的结构。

时钟信号发生:

采用系统时钟时序化逻辑的设计需要产生一个时钟。重复时钟在 VHDL 或者 Verilog 源码中可以很简单的发生。以下就是在 VHDL 和 Verilog 中的时钟发生例子:

VHDL:

```
--申明时钟周期常量
Constant clockperiod : time :=10 ns;
--时钟发生方法之一:
Clock<=not clock after clockperiod/2;
--时钟发生方法之二:
Generate clock : process
Begin
   Wait for (clockperiod/2);
   Clock<='1';
   Wait for (clockperiod/2);
   Clock<='0';
End process;
```

Verilog:

```
//时钟周期定义
Parameter clockperiod =10;
//方法 1
Initial begin
```

```
   Forever clock=#(clockperiod/2) ~clock;
End
//方法 2
Initial begin
   Always #(clockperiod/2) clock=~clock;
End
```

激励产生:

要得到测试验证结果，必须给 DUT 提供激励信号。在测试文件中并发激励模块经常被用来提供必须的激励。有两种方式: 绝对时间激励和相对时间激励。在第一种方式中仿真值的描述是相对于仿真时间为零的。相比，相对时间激励提供初始值，然后等待再次触发激励前的事件的发生。两种方法根据设计者的需求可以在同一个测试文件中使用。表 2，表 3 是上述两种方法的 VHDL 以及 Verilog 的例子。

表 2 : 绝对时间激励

VHDL-绝对时间激励方式	Verilog－绝对时间激励方式
Mainstimulus : process Begin Reset<='1'; Load<='0'; Count_updn<='0'; Wait for 100 ns; Reset<='0'; Wait for 20 ns; Load<='1'; Wait for 20 ns; Count_updn<='1'; End process;	Initial begin Reset=1; Load=0; Count_updn=0; #100 reset=0; #20 load=1; #20 count_updn=1; end

表 3 : 相对时间激励

VHDL	Verilog

First : Process(clock)
Begin
If rising edge(clock) then
Tb_count<=tb_count+1;
End if;
End process;
Second : Process
Begin
If tb_count<=5 then
Reset<='1';
Load<='0';
Count_updn<='0';
Else
Reset<='0';
Load<='1';
Count_updn<='1';
End process;
Final :Process
Begin
If count="1100" then
Count_updn<='0';
Report"terminal count reached,now counting down."
End if;
End process;
	always@(posedge clock)
	tb_count<=tb_count+1;
	initial begin
	if(tb_count<=5)
	begin
	reset=1;
	load<=0;
	count_updn = 0;
	end
	else
	begin
	reset=0;
	load=1;
	count_updn=1;
	end
	end
	initial begin
	if(count==1100) begin
	count_updn<=0;
	$display("terminal count reached,now counting down.");
	End
	End

VHDL 进程以及 Verilog 初始化模块与测试文件中的其他进程或者初始化模块是同时被执行的。然而，在每个（进程或者初始的）模块中，事件都是按照书写顺序依次执行的。将一个复杂的激励序列用多个模块来简化，以增加易读性和易维护性。

结果显示:

在 Verilog 中可以很便利的使用$display 和$monotor 关键词来显示结果。虽然 VHDL 没有类似的显示描述命令,但他提供了 std_textio 包，允许文件 I/O 重定向到显示终端（这一技术的使用例子详见下面

的自测试)。

以下是一个结果在终端屏幕显示的 Verilog 例子:

```
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
```

$display 关键词将括号内双引号中的文本输出到显示终端。$monitor 则不一样,他的输出是事件驱动。在例子中,$realtime 变量(用户定义的)被用来触发信号列表中的显示值。信号列表由 $realtime 变量开始,紧接着是其他要显示的信号(如 clock,reset,load 等)。关键词“%”包含一系列的格式描述,用于控制各个信号值以何种格式显示在终端。格式列表是有位置规定的,每个格式描述要和信号列表中信号顺序相对应。例如,%t 描述了 $realtime 的值显示为时间格式,第一个%b 格式符指定了 clock 值显示为二进制格式。Verilog 提供了额外的格式描述符,例如,%h 用于十六进制,%d 用于十进制,%o 用于八进制(完整的关键词和格式描述符请参考 Verilog 书籍)。

上述例子的显示结果如图二:

```
VSIM 11> run 200 ns
#    Time Clk Rst Ld SftRg Data Sel
#     0.0ns 0 1 0 xxxxx 00000 00
#    50.0ns 1 1 0 00000 00000 00
#   100.0ns 0 1 0 00000 00000 00
#   150.0ns 1 1 0 00000 00000 00

VSIM 12>
```

对于一般的 FPGA 设计来说,简单的测试文件足以满足设计验证,

下面我们谈谈简单的测试文件验证设计。

简单的测试文件

简单的测试文件例化设计，然后提供激励。测试输出通过图形的方式显示在仿真工具的波形窗口，或者以文本的方式送到用户终端，或存为文本文件。下面是一个简单的 Verilog 移位寄存器的设计:

```
  module shift_reg (clock, reset, load, sel, data,
shiftreg);
  input clock;
  input reset;
  input load;
  input [1:0] sel;
  input [4:0] data;
  output [4:0] shiftreg;
  reg [4:0] shiftreg;
   always @ (posedge clock)
   begin
    if (reset)
     shiftreg = 0;
     else if (load)
     shiftreg = data;
    else
     case (sel)
      2'b00 : shiftreg = shiftreg;
      2'b01 : shiftreg = shiftreg << 1;
      2'b10 : shiftreg = shiftreg >> 1;
      default : shiftreg = shiftreg;
     endcase
   end
  endmodule
```

以下简单的测试文件例子例化上述的移位寄存器的设计。

Verilog:

```
  module testbench; // declare testbench name
  reg clock;
  reg load;
  reg reset; // declaration of signals
  wire [4:0] shiftreg;
  reg [4:0] data;
```

```
reg [1:0] sel;
   // instantiation of the shift_reg design below
   shift_reg dut(.clock (clock),
           .load (load),
           .reset (reset),
           .shiftreg (shiftreg),
           .data (data),
           .sel (sel));
  //this process block sets up the free running
clock
  initial begin
   clock = 0;
   forever #50 clock = ~clock;
  end
  initial begin// this process block specifies the
  stimulus.
   reset = 1;
   data = 5'b00000;
   load = 0;
   sel = 2'b00;
   #200
   reset = 0;
   load = 1;
   #200
   data = 5'b00001;
   #100
   sel = 2'b01;
   load = 0;
   #200
   sel = 2'b10;
   #1000 $stop;
  end
  initial begin // this process block pipes the
  ASCII results to the
  //terminal or text editor
   $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
endmodule
```

这个测试文件例化了设计，建立了时钟，然后提供了激励。所有的进程都是从仿真时间为零时开始的并且是并发的。符号“#”描述

了在下一激励到来时的延时，$stop 命令使仿真工具停止测试文件的仿真（所有的测试文件都应包含停止命令）。最后，$monitor 命令将结果以 ASCII 形式显示在终端或者使文本编辑器中。以下是 VHDL 的例子:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity testbench is
end entity testbench;
architecture test_reg of testbench is
 component shift_reg is
  port(clock : in std_logic;
       reset : in std_logic;
       load : in std_logic;
       sel : in std_logic_vector(1 downto 0);
       data : in std_logic_vector(4 downto 0);
       shiftreg : out std_logic_vector(4 downto
       0));
end component;

signal clock, reset, load: std_logic;
signal shiftreg, data: std_logic_vector(4 downto
0);
signal sel: std_logic_vector(1 downto 0);
constant ClockPeriod : TIME := 50 ns;
begin
 UUT : shift_reg port map (clock => clock,
                           reset => reset,
                           load => load,
                           data => data,
                           shiftreg => shiftreg
                           );
 process
 begin
  clock <= not clock after (ClockPeriod / 2);
 end process;
 process
 begin
  reset <= '1';
  data <= "00000";
```

```
    load <= '0';
    set <= "00";
    wait for 200 ns;
    reset <= '0';
    load <= '1';
    wait for 200 ns;
    data <= "00001";
    wait for 100 ns;
    sel <= "01";
    load <= '0';
    wait for 200 ns;
    sel <= "10";
    wait for 1000 ns;
  end process;
end test_reg;
```

这个 VHDL 测试文件在功能上和前面的 Verilog 测试文件的功能类似，都将结果显示在终端。在 VHDL 中，std_textio 包被用来将信息显示在终端，这个将在后面叙述。下面将讨论设计的自动验证。

自动验证

测试文件的的自动验证是值得推荐的，特别是对于大型设计。自动验证可以缩短检查设计正确性所需的时间，以及最小化人为错误。

常用的自动测试验证的方法有多种:

1. 数据对比。首先，建立一个包含期望输出的数据文件(“golden vector”文件)；其次，捕捉仿真输出并且将其和预先建立的包含期望值的向量文件进行比较。然而，由于没有提供从输出到输入文件的指针，这一方法的一个缺点就是一旦有错误发生就很难跟踪。
2. 波形比较。波形比较可以是自动的也可以人工执行。这个方法

使用了测试文件比较器，将测试输出波形和参考波形进行比较。

3. 自检测试。自检测试文件在运行时自动将输出结果和期望的结果进行比较，而不是在仿真结束后。由于错误跟踪信息可以建立在测试文件中显示何处存在设计错误，所以调试的时间将明显减少。以下将进一步阐述。

自检测试

自检测试通过在测试文件中置入一系列期望的向量。这些向量将在定义的时间内和真实的方针结果进行比较。如果真实的结果和期望的结果匹配，则仿真成功。否则，测试文件报告差异。

自检测试的执行对于同步设计来说稍微简单，因为期望的结果和真实的结果的比较可以在一个时钟的边沿或者在每 N 个时钟周期后进行。比较的方式取决于设计。例如，一个存储器 IO 的测试文件应该在每次新的数据写入或者从存储器的某个位置读出时检查结果。同样，如果一个设计大量使用了组合模块，那么组合延时必须在描述期望结果时被考虑进去。

在自检测试文件中，期望输出在运行时经常被用来和真实输出比较，以提供自动错误检查。这一技术在中小设计中极为有效。然而，由于可能的输出组合随着设计的复杂程度成指数增加，编写一个大型设计的自检测试文件就变得极为困难并且耗时。下面分别就 Verilog 和 VHDL 进行举例:

Verilog 例子: 例化设计后，描述了期望的结果，后面的代码中

将真实结果和期望的结果进行了比较，并且将结果送到终端。如果有不匹配的地方，就显示“end of good simulation”。如果发生不匹配在错误报告的同时显示不匹配的期望值和真实值。

```
‘timescale 1 ns / 1 ps
module test_sc;
reg tbreset, tbstrtstop;
reg tbclk;
wire [6:0] onesout, tensout;
wire [9:0] tbtenthsout;
parameter cycles = 25;
reg [9:0] Data_in_t [0:cycles];
// ////////////////////////////
// Instantiation of the Design
// ////////////////////////////
stopwatch UUT (.CLK (tbclk), .RESET
(tbreset), .STRTSTOP (tbstrtstop),
.ONESOUT (onesout), .TENSOUT
(tensout), .TENTHSOUT (tbtenthsout));
wire [4:0] tbonesout, tbtensout;
 assign tbtensout = led2hex(tensout);
 assign tbonesout = led2hex(onesout);
//////////////////////////////////////////////
/////////////////
//EXPECTED RESULTS
//////////////////////////////////////////////
/////////////////
initial begin
 Data_in_t[1] =10’b1111111110;
 Data_in_t[2] =10’b1111111101;
 Data_in_t[3] =10’b1111111011;
 Data_in_t[4] =10’b1111110111;
 Data_in_t[5] =10’b1111101111;
 Data_in_t[6] =10’b1111011111;
 Data_in_t[7] =10’b1110111111;
 Data_in_t[8] =10’b1101111111;
 Data_in_t[9] =10’b1011111111;
 Data_in_t[10]=10’b0111111111;
 Data_in_t[11]=10’b1111111110;
 Data_in_t[12]=10’b1111111110;
 Data_in_t[13]=10’b1111111101;
 Data_in_t[14]=10’b1111111011;
```

```
 Data_in_t[15]=10'b1111110111;
 Data_in_t[16]=10'b1111101111;
 Data_in_t[17]=10'b1111011111;
 Data_in_t[18]=10'b1110111111;
 Data_in_t[19]=10'b1101111111;
 Data_in_t[20]=10'b1011111111;
 Data_in_t[21]=10'b0111111111;
 Data_in_t[22]=10'b1111111110;
 Data_in_t[23]=10'b1111111110;
 Data_in_t[24]=10'b1111111101;
 Data_in_t[25]=10'b1111111011;
end
reg GSR;
   assign glbl.GSR = GSR;
initial begin
   GSR = 1;
 // ////////////////////////////////
 // Wait till Global Reset Finished
 // ////////////////////////////////
 #100 GSR = 0;
end
// /////////////////
// Create the clock
// /////////////////
initial begin
 tbclk = 0;
 // Wait till Global Reset Finished, then cycle
 clock
 #100 forever #60 tbclk = ~tbclk;
end
initial begin
 // ////////////////////////////
 // Initialize All Input Ports
 // ////////////////////////////
 tbreset = 1;
 tbstrtstop = 1;
 // ///////////////////////
 // Apply Design Stimulus
 // ///////////////////////
 #240 tbreset = 0;
 tbstrtstop = 0;
 #5000 tbstrtstop = 1;
 #8125 tbstrtstop = 0;
 #500 tbstrtstop = 1;
```

```
 #875 tbreset = 1;
 #375 tbreset = 0;
 #700 tbstrtstop = 0;
 #550 tbstrtstop = 1;
 //
 ////////////////////////////////////////////
 ////////
 // simulation must be halted inside an initial
 statement
 //
 ////////////////////////////////////////////
 ////////
 //
  #100000 $stop;
end
integer i,errors;
////////////////////////////////////////////
// Block below compares the expected vs. actual
results
// at every negative clock edge.
////////////////////////////////////////////
always @ (posedge tbclk)
begin
 if (tbstrtstop)
 begin
  i = 0;
  errors = 0;
 end
else
begin
 for (i = 1; i <= cycles; i = i + 1)
 begin
  @(negedge tbclk)
  // check result at negedge
  $display("Time%d ns; TBSTRTSTOP=%b; Reset=%h;
  Expected
  TenthsOut=%b; Actual TenthsOut=%b", $stime,
  tbstrtstop, tbreset,
  Data_in_t[i], tbtenthsout);

  if ( tbtenthsout !== Data_in_t[i] )
  begin
     $display(" ------ERROR. A mismatch has
     occurred-----");
```

```
        errors = errors + 1;
      end
    end
    if (errors == 0)
     $display("Simulation finished
     Successfully.");
    else if (errors > 1)
     $display("%0d ERROR! See log above for
     details." errors);
    else
     $display("ERROR! See log above for details.");
     #100 $stop;
    end
  end
endmodule
```

这个例子中自检文件的设计可以引入到其他任何一个测试中，当然，期望输出和信号名需要修改以重新利用。如果检测不需要在每个时钟边沿，那么 for 循环可以根据需要修改。如果仿真成功，要求打印的信息将显示在终端。如下是个存储器仿真的结果:

```
VSIM 14> run 23000 ns
#
# Beginning Simulation...
# Pattern#       0 time    1950: WE=0; Address=0; Data=00; Expected Q=xx; Actual Q=xx
# Pattern#       1 time    2950: WE=1; Address=0; Data=01; Expected Q=01; Actual Q=01
# Pattern#       2 time    3950: WE=1; Address=1; Data=02; Expected Q=02; Actual Q=02
# Pattern#       3 time    4950: WE=1; Address=2; Data=04; Expected Q=04; Actual Q=04
# Pattern#       4 time    5950: WE=1; Address=3; Data=08; Expected Q=08; Actual Q=08
# Pattern#       5 time    6950: WE=1; Address=4; Data=10; Expected Q=10; Actual Q=10
# Pattern#       6 time    7950: WE=1; Address=5; Data=20; Expected Q=20; Actual Q=20
# Pattern#       7 time    8950: WE=1; Address=6; Data=40; Expected Q=40; Actual Q=40
# Pattern#       8 time    9950: WE=1; Address=7; Data=80; Expected Q=80; Actual Q=80
# Pattern#       9 time   10950: WE=0; Address=0; Data=07; Expected Q=01; Actual Q=01
# Pattern#      10 time   11950: WE=0; Address=1; Data=08; Expected Q=02; Actual Q=02
# Pattern#      11 time   12950: WE=0; Address=2; Data=09; Expected Q=04; Actual Q=04
# Pattern#      12 time   13950: WE=0; Address=3; Data=10; Expected Q=08; Actual Q=08
# Pattern#      13 time   14950: WE=0; Address=4; Data=11; Expected Q=10; Actual Q=10
# Pattern#      14 time   15950: WE=0; Address=5; Data=12; Expected Q=20; Actual Q=20
# Pattern#      15 time   16950: WE=0; Address=6; Data=13; Expected Q=40; Actual Q=40
# Pattern#      16 time   17950: WE=0; Address=7; Data=14; Expected Q=80; Actual Q=80
# Pattern#      17 time   18950: WE=1; Address=0; Data=aa; Expected Q=aa; Actual Q=aa
# Pattern#      18 time   19950: WE=0; Address=0; Data=55; Expected Q=aa; Actual Q=aa
# Pattern#      19 time   20950: WE=1; Address=0; Data=55; Expected Q=55; Actual Q=55
# Pattern#      20 time   21950: WE=0; Address=0; Data=aa; Expected Q=55; Actual Q=55
# Good!  End of Good Simulation.
# ** Note: $finish    : F:/projects/appnotes/testbenches/ver_example/test.v(112)
#    Time: 22950 ns  Iteration: 0  Instance: /test
# 1
# Break at F:/projects/appnotes/testbenches/ver_example/test.v line 112

VSIM 15>
```

VHDL 例子: 在 VHDL 中向量文件包含期望结果。VHDL 的 textio 包被用来从向量文件中读取数据，并且显示错误信息:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
LIBRARY ieee;
USE IEEE.STD_LOGIC_TEXTIO.ALL;
USE STD.TEXTIO.ALL;
ENTITY testbench IS
END testbench;
ARCHITECTURE testbench_arch OF testbench IS
COMPONENT stopwatch
 PORT (
      CLK : in STD_LOGIC;
      RESET : in STD_LOGIC;
      STRTSTOP : in STD_LOGIC;
      TENTHSOUT : out STD_LOGIC_VECTOR (9 DOWNTO
      0);
       ONESOUT : out STD_LOGIC_VECTOR (6 DOWNTO
    0);
      TENSOUT : out STD_LOGIC_VECTOR (6 DOWNTO 0)
 );
END COMPONENT;
SIGNAL CLK : STD_LOGIC;
SIGNAL RESET : STD_LOGIC;
SIGNAL STRTSTOP : STD_LOGIC;
SIGNAL TENTHSOUT : STD_LOGIC_VECTOR (9 DOWNTO 0);
SIGNAL ONESOUT : STD_LOGIC_VECTOR (6 DOWNTO 0);
SIGNAL TENSOUT : STD_LOGIC_VECTOR (6 DOWNTO 0);
constant ClockPeriod : Time := 60 ns;
FILE RESULTS: TEXT IS OUT "results.txt";
signal i: std_logic;
BEGIN
 UUT : stopwatch PORT MAP (
                          CLK => CLK,
                          RESET => RESET,
                          STRTSTOP => STRTSTOP,
                          TENTHSOUT =>
                          TENTHSOUT,
                          ONESOUT => ONESOUT,
                          TENSOUT => TENSOUT
                          );
 stimulus: PROCESS
```

```
begin
 reset <= '1';
 strtstop <= '1';
 wait for 240 ns;
 reset <= '0';
 strtstop <= '0';
 wait for 5000 ns;
 strtstop <= '1';
 wait for 8125 ns;
 strtstop <= '0';
 wait for 500 ns;
 strtstop <= '1';
 wait for 875 ns;
 reset <= '1';
 wait for 375 ns;
 reset <= '0';
 wait for 700 ns;
 strtstop <= '0';
 wait for 550 ns;
 strtstop <= '1';
end process stimulus;
clock: process
begin
 clk <= '1';
 wait for 100 ns;
 loop
 wait for (ClockPeriod / 2);
 CLK <= not CLK;
 end loop;
end process clock;
check_results : process
variable tmptenthsout: std_logic_vector(9
downto 0);
variable l: line;
variable good_val, good_number, errordet:
boolean;
variable r : real;
variable vector_time: time;
variable space: character;
file vector_file: text is in "values.txt";
begin
 while not endfile(vector_file) loop
 readline(vector_file, l);
 read(l, r, good => good_number);
```

```
    next when not good_number;
    vector_time := r * 1 ns;
    if (now < vector_time) then
       wait for vector_time - now;
    end if;
    read(l, space);
    read(l, tmptenthsout, good_val);
    assert good_val REPORT "bad tenthsoutvalue";
    wait for 10 ns;
    if (tmptenthsout /= tenthsout) then
       assert errordet REPORT "vector mismatch";
    end if;
    end loop;
    wait;
  end process check_results;
end testbench_arch;
  library XilinxCoreLib;
  CONFIGURATION stopwatch_cfg OF testbench IS
  FOR testbench_arch
    FOR ALL : stopwatch use configuration
    work.cfg_tenths;
    END FOR;
  END FOR;
END stopwatch_cfg;
```

测试向量文件如下，包含了期望的仿真值。

```
-- Vector file containing expected results
0 1111111110
340 1111111110
400 1111111101
460 1111111011
520 1111110111
580 1111101111
640 1111011111
700 1110111111
760 1101111111
820 1011111111
880 0111111111
940 1111111110
1000 1111111110
1060 1111111101
1120 1111111011
1180 1111110111
1240 1111101111
```

```
1300 1111011111
1360 1110111111
1420 1101111111
1480 1011111111
1540 0111111111
1600 1111111110
1660 1111111110
1720 1111111101
1780 1111111011
```

如果检测到错误发生，以下信息将显示在仿真工具窗口：

```
VSIM 166> restart
VSIM 167> run 1500 ns
# ** Error: vector mismatch
#    Time: 710 ns  Iteration: 0  Instance: /testbench
# ** Error: vector mismatch
#    Time: 1250 ns  Iteration: 0  Instance: /testbench

VSIM 168>
```

测试文件编写指导

和规划电路设计可以得到性能更好的电路一般，规划好测试文件可以提高仿真的验证性能。

- 首先在设计测试文件前要了解仿真工具。

  虽然现在通用的仿真工具符合 HDL 的工业标准，但是这些标准并不是都包含了很多重要的仿真描述语句。不同的仿真工具有不同的特性，能力以及特有的性能，以致于产生不同的结果。

  A. 基于事件的仿真和基于周期的仿真：

  仿真工具采用基于事件或者基于周期的仿真方法。基于事件的仿真工具当一个输入，信号或者门的值变化时得到一个仿真结果。在基于事件的仿真工具中，为了得到优化的时序仿真一个延时值可以带有门级和网络延时。基于周期的仿真工具针对于同步设计。他优化了组合逻辑以及在每个时钟周期分析结

果。这一特性使得基于周期的仿真工具比基于事件的仿真工具更快和更节省内存使用。然而，由于基于周期的仿真工具不允许详细的时序描述，所以是不够精确的。

B. 事件安排

基于事件的仿真工具的供应商使用了不同的算法来安排仿真事件。然而，在同一仿真时间内发生的不同事件就会取决于仿真工具的安排算法而被安排在不同的顺序（计算机执行指令是顺序的，而硬件描述的硬件操作是并发的，这是一个矛盾体）。为了避免算法的依赖性以及为了得到正确的结果，事件驱动的测试文件就应该清楚准确的描述激励顺序。

- 避免使用死循环

当一个事件被增加到基于事件触发的仿真工具中的时候，CPU和内存的使用就会增加，仿真的进程也会变慢。除非对测试文件很重要，否则不要用无限循环来提供激励。通常时钟产生是在一个无限循环中，但是并不适合其他信号。

- 将激励分解到逻辑模块中

在测试文件中，所有的“INITIAL”(Verilog)和process(VHDL)模块都是并发的。将不相关的激励分解到独立的模块中将使得测试激励的顺序变得更容易执行和观察。由于每个并发的模块都是相对于时间为零的，所以用独立的模块激励的传递就变得更简单。使用独立的激励模块可以使测试文件易于产生，维护

以及升级。

- 避免显示不重要的数据

  大型设计的测试文件可能包含超过 10K 的事件和无数的信号。大量信号的结果显示极大的影响仿真时间。为了得到适当的仿真速度最好的办法就是每 n 个时钟周期只采样相关的信号。

如何将激励分割成独立的 TASKS 或者是 PROCEDURES?

上面我们有提到，在一个大型，复杂的设计中，测试文件中的激励也是比较复杂的，如何编写有效的测试代码成为了能否提高设计验证效率的关键因素。在一个大的测试文件中，激励的分割应该以增加代码的透明度和维护的便利性为原则。VERILOG 中的 TASK 以及 VHDL 中的 Procedures 被用来分割信号。在下面的例子中，测试文件用来仿真一个 SDRAM 控制器的设计。这个设计包含了很多相关的激励的模块，所以测试文件通过申明在测试文件中被调用的独立的 TASK 来分割激励，以检验独立的设计功能。

VERILOG:

```
task addr_wr;
 input [31 : 0] address;
 begin
  data_addr_n = 0;
  we_rn = 1;
  ad = address;
 end
endtask
task data_wr;
 input [31 : 0] data_in;
 begin
  data_addr_n = 1;
  we_rn = 1;
  ad = data_in;
```

```
    end
  endtask
  task addr_rd;
    input [31 : 0] address;
    begin
      data_addr_n = 0;
      we_rn = 0;
      ad = address;
    end
  endtask
  task data_rd;
    input [31 : 0] data_in;
    begin
      data_addr_n = 1;
      we_rn = 0;
      ad = data_in;
    end
  endtask
  task nop;
  begin
    data_addr_n = 1;
    we_rn = 0;
    ad = hi_z;
  end
  endtask
```

这些任务描述了设计功能的独立的元素：地址读写、数据读写、无操作。一旦申明，这些任务就可以被激励的进程调用，如下：

```
Initial begin
nop ; // Nop
#( 86* `CYCLE +1); addr_wr (32'h20340400); //
Precharge, load
Controller MR
#(`CYCLE); data_wr (32'h0704a076); // value for
Controller MR
#(`CYCLE); nop ; // Nop
#(5 * `CYCLE); addr_wr (32'h38000000); // Auto
Refresh
#(`CYCLE); data_wr (32'h00000000); //
#(`CYCLE); nop ; // Nop
…
…
end
```

VHDL:

以下是上述同一个设计的 VHDL 测试文件，通过子程序来分割:

```
Stimulus : process
 procedure addr_wr (address: in
 std_logic_vector(31 downto 0)) is
 begin
  data_addr_n <= '0';
  we_rn <= '1';
  ad <= address;
 end addr_wr;
 procedure data_wr (data_in: in
 std_logic_vector(31 downto 0 )) is
 begin
  data_addr_n <= '1';
  we_rn <= '1';
  ad <= data_in;
 end data_wr;
 procedure addr_rd (address: in
 std_logic_vector(31 downto 0)) is
 begin
  data_addr_n <= '0';
  we_rn <= '0';
  ad <= address;
 end addr_rd;
 procedure data_rd (data_in: in
 std_logic_vector(31 downto 0)) is
 begin
  data_addr_n <= '1';
  we_rn <= '0';
  ad <= data_in;
 end data_rd;
 procedure nop is
 begin
  data_addr_n <= '1';
  we_rn = '0';
  ad = 'Z';
 end nop;
begin
 nop ; -- Nop
 wait for 200 ns;
 addr_wr (16#20340400#); -- Precharge, load
 Controller MR
```

```
 wait for 8 ns;
 data_wr (16#0704a076#); -- value for Controller
 MR
 wait for 8 ns;
 nop ; -- Nop
 wait for 40 ns;
 addr_wr (16#38000000#); -- Auto Refresh
 wait for 8 ns;
 data_wr (16#00000000#);
 wait for 8 ns;
 nop ; -- Nop
 ..
 ..
End process;
```

将激励分割成独立的任务或者是子程序使得激励更容易被执行以及代码具有更好的可读性。

在很多设计中都会涉及到一个问题，就是双向口，这在测试文件的设计中需要和普通的 I/O 口区别开，下面举一个例子说明:

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity bidir_IO is
port (DATA : inout STD_LOGIC_VECTOR(1 downto 0);
      READ_WRITE : in STD_LOGIC);
end bidir_infer;

architecture Lattice of bidir_IO is
signal LATCH_OUT : STD_LOGIC_VECTOR(1 downto 0);
begin
 process(READ_WRITE, DATA)
 begin
  if (READ_WRITE = '1') then
     LATCH_OUT <= DATA;
  end if;
 end process;
 process(READ_WRITE, LATCH_OUT)
 begin
  if (READ_WRITE = '0') then
```

```
      DATA(0) <= LATCH_OUT(0) and LATCH_OUT(1);
      DATA(1) <= LATCH_OUT(0) or LATCH_OUT(1);
    else
      DATA(0) <= 'Z';
      DATA(1) <= 'Z';
    End if;
  End process;
End Lattice;
```

在这个例子中，如何对双向口进行仿真的测试文件需要如下方式建立:

```
library ieee;
use ieee.std_logic_1164.all;

Entity testbench is
End testbench;

Architecture test_bidir of testbench is
  Component bidir_IO
  port (DATA : inout STD_LOGIC_VECTOR(1 downto 0);
        READ_WRITE : in STD_LOGIC);
  end component;
signal read_writet: std_logic;
signal datat, data_top : std_logic_vector(1
downto 0);
begin
  datat <= data_top when (Read_writet = '1') else
  (others => 'Z');
  data_top <= datat when (Read_writet = '0') else
  (others => 'Z');
  uut : bidir_IO port map (datat, read_writet);
  process begin
    read_writet <= '1';
    data_top <= "10";
    wait for 50 ns;
    read_writet <= '0';
    wait;
  end process;
end test_bidir;
```

在上述代码中可见，双向数据总线由 Read_writet 信号控制。下面就针对使用 Verilog HDL 的设计者进行举例说明。设计代码如下:

```
module bidir_infer (DATA, READ_WRITE);
input READ_WRITE ;
inout [1:0] DATA ;
reg [1:0] LATCH_OUT ;
 always @ (READ_WRITE or DATA)
 begin
 if (READ_WRITE == 1)
  LATCH_OUT <= DATA;
 end
 assign DATA = (READ_WRITE == 1) ? 2'bZ :
 LATCH_OUT;
endmodule
```

测试文件:

```
module test_bidir_ver;
reg read_writet;
reg [1:0] data_in;
wire [1:0] datat, data_out;
 bidir_infer uut (datat, read_writet);
 assign datat= (read_writet == 1) ? data_in : 2'bZ;
 assign data_out=(read_writet ==0) ? datat : 2'bZ;
 initial begin
  read_writet = 1;
  data_in = 11;
  #50 read_writet = 0;
 end
endmodule
```

在上面的例子中已经讨论了如“$monitor,$display,$time”等非常有用的 VERILOG 语言结构,下面额外讨论几个可在测试文件中使用的 VERILOG 结构: force/release, force 和 release 可以用来重新更改子程序中的寄存器或者信号网络的赋值。下面举例说明其使用:

```
module testbench;
..
..
initial begin
reset = 1;
force DataOut = 101;
#25 reset = 0;
#25 release DataOut;
..
```

```
..
end
endmodule
```

assign/deassign，类似于 force/release，但是在设计中 assign/deassign 只对寄存器使用。通常用来设置输入值。如 force 一样，通过子程序可以使用 assign 重新改变原来的值。其实用如下例子所示:

```
module testbench;
..
..
initial begin
reset = 1;
DataOut = 101;
#25 reset = 0;
release DataOut;
..
..
end
initial begin
#20 assign reset = 1;// this assign statement
        overrides the earlier statement
#25 reset = 0;
#50 release reset;
endmodule
```

timescales，直接被用来描述测试的单位时间，同时会影响仿真工具的精度，语法结构如下:

·timescales reference_time/precision

Reference_time 就是参考时间，也就是作为衡量的单位时间。Precision 就是精度，决定了延时和单位步进时间。下面就是一个·timescales 的使用实例:

```
‘timescale 1 ns / 1 ps
// this sets the reference time to 1 ns and
precision to 1 ps.
module testbench;
```

```
..
..
initial begin
#5 reset = 1; // 5 unit time steps correspond to
5 * 1ns = 5ns in simulation time
#10 reset = 0;
..
end
initial begin
$display ("%d , Reset = %b", $time, reset); // this
display
// statement will get executed
// on every simulator step, ie, 1 ps.
end
endmodule
```

如果仿真使用时序延时值，为了包含延时仿真运行时间精度必须大于最小的延时。例如，如果在仿真库中使用了 9ps 的延时，那么仿真的精度必须是 1ps 以提供 9ps 的延时。

还有，在对存储器进行仿真时，往往需要对存储器进行初始化，在LATTICE PLD开发工具中，存储器初始化文件为mem文件，在ALTERA和XILINX开发工具中存储器初始化文件为mif文件。Mem文件的编写可以直接用文本文件编辑工具进行编写，文件内容可以采用二进制，十六进制或者是地址 + 十六进制的格式来编写。然后存为mem文件。至于在测试文件中读取存储器内容则采用$readmemb和$readmemh命令，语法结构如下:

```
$readmemb ("<design.mif/mem>", design_instance);
```

VHDL 中还提供了存储器初始化文件记录，可以用来输入存储模块的内容，语法如下: FILE meminifile: TEXT IS IN "<design.mif>";

无论是设计源代码还是测试文件代码的编写风格都是很重要的，良好的代码风格可以提高设计的可读性和可维护性。所以建议在编写

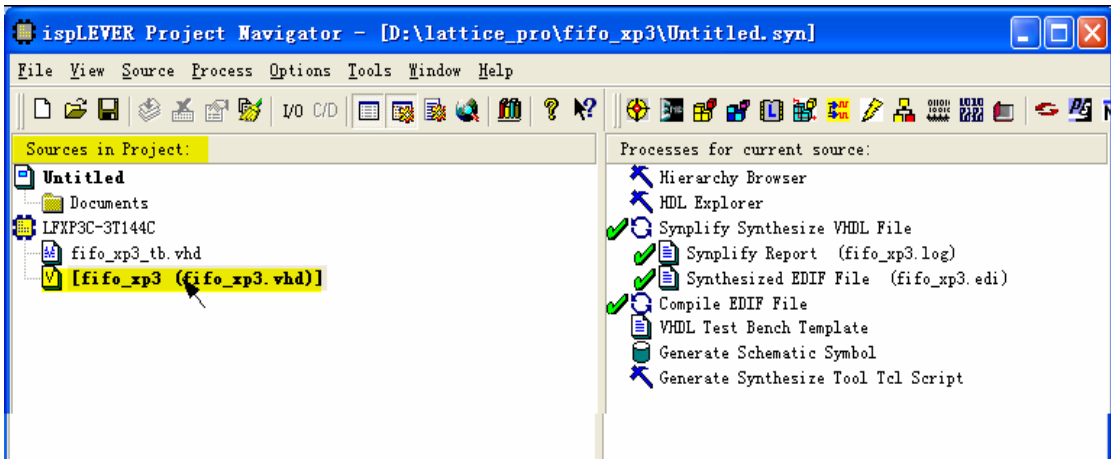
测试代码的时候应保持清晰的结构和可读性，以便以重用和维护。

LATTICE 仿真流程

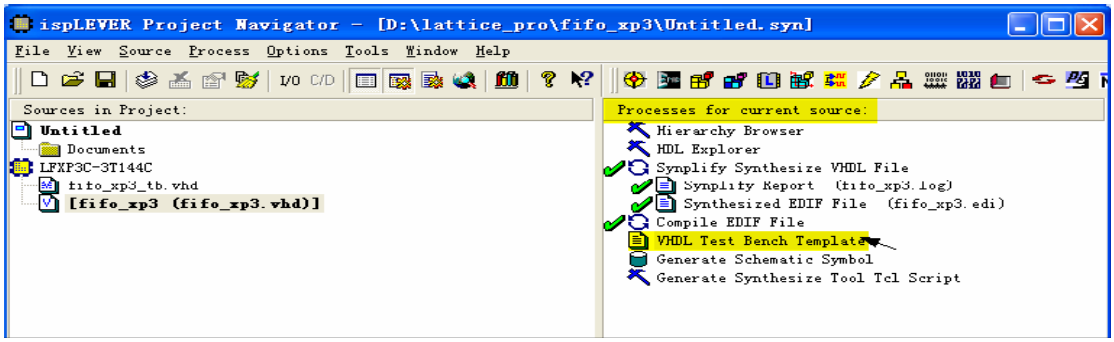
首先编写测试文档，你可以选择自己编写，也可以通过 LATTICE 自带的模版生成功能，先生成模版，然后通过修改模版内容，初始化以及提供激励来完成测试文件的编写。详细步骤如下:

VHDL Testbench:

1. 在“source in Project”中点击所要仿真的设计文件，如下图所示：点击顶层文件（fifo_xp3）

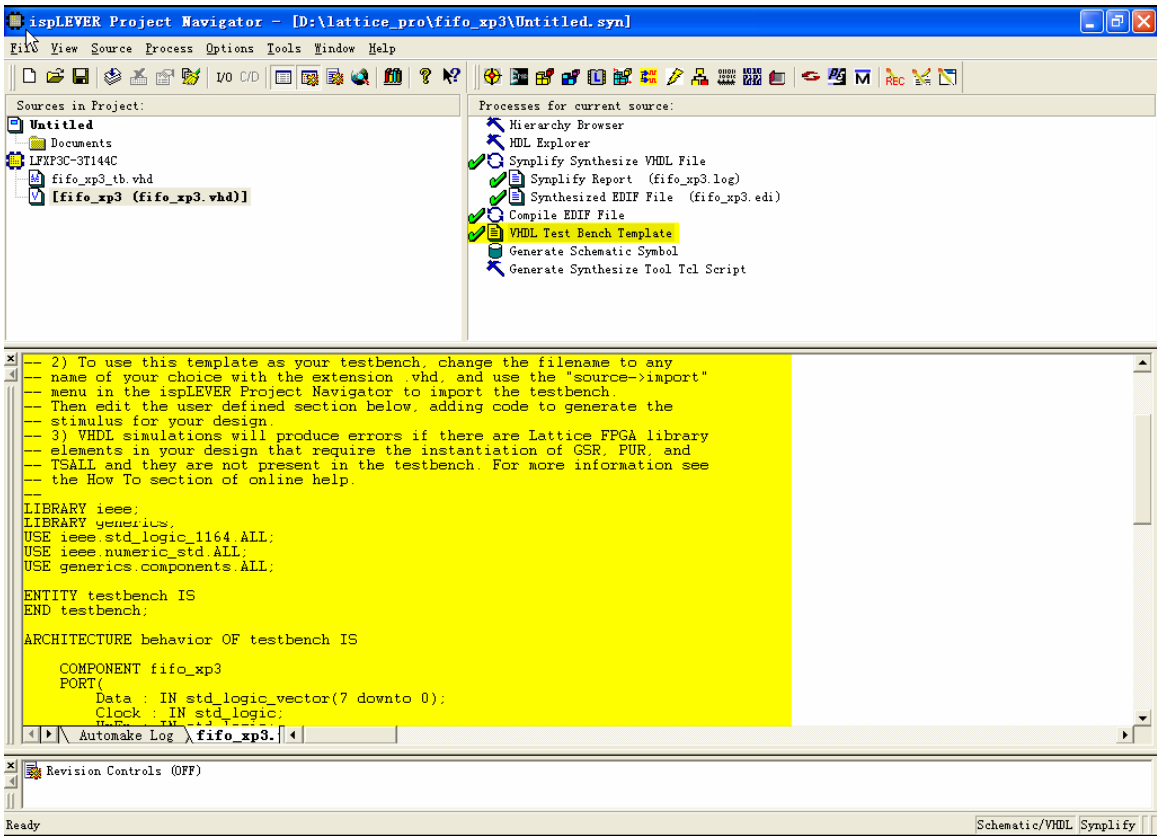


2. 在“Processes for current source”中双击“VHDL Test Bench Template”：



3. 生成测试文件模版，“VHDL Test Bench Template”选项前有

绿色的勾显示，下面的信息栏出现的是模版文件的内容。



本例中生成的模版文件如下:

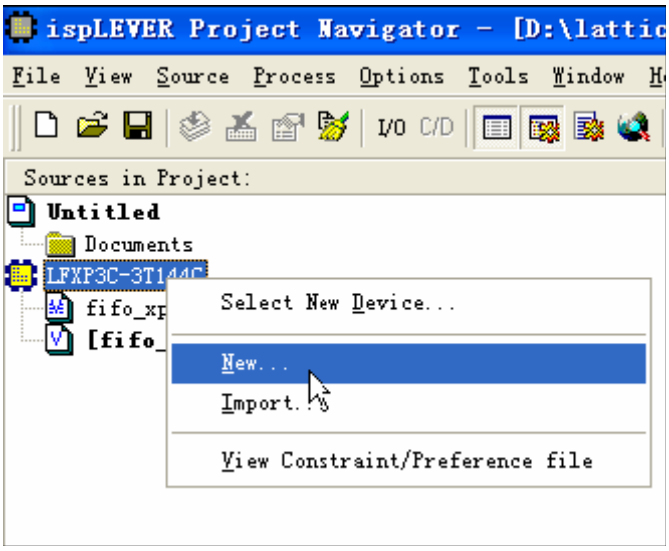
```
-- VHDL Test Bench Created from source file
fifo_xp3.vhd -- 01/25/07 20:53:09
--
-- Notes:
-- 1) This testbench template has been
automatically generated using types
-- std_logic and std_logic_vector for the ports
of the unit under test.
-- Lattice recommends that these types always be
used for the top-level
-- I/O of a design in order to guarantee that the
testbench will bind
-- correctly to the timing (post-route) simulation
model.
-- 2) To use this template as your testbench, change
the filename to any
```

```
-- name of your choice with the extension .vhd,
and use the "source->import"
-- menu in the ispLEVER Project Navigator to import
the testbench.
-- Then edit the user defined section below, adding
code to generate the
-- stimulus for your design.
-- 3) VHDL simulations will produce errors if there
are Lattice FPGA library
-- elements in your design that require the
instantiation of GSR, PUR, and
-- TSALL and they are not present in the testbench.
For more information see
-- the How To section of online help.
--
LIBRARY ieee;
LIBRARY generics;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE generics.components.ALL;
ENTITY testbench IS
END testbench;
ARCHITECTURE behavior OF testbench IS
COMPONENT fifo_xp3
PORT(
Data : IN std_logic_vector(7 downto 0);
Clock : IN std_logic;
WrEn : IN std_logic;
RdEn : IN std_logic;
Reset : IN std_logic;
Q : OUT std_logic_vector(7 downto 0);
Empty : OUT std_logic;
Full : OUT std_logic
);
END COMPONENT;
SIGNAL Data : std_logic_vector(7 downto 0);
SIGNAL Clock : std_logic;
SIGNAL WrEn : std_logic;
SIGNAL RdEn : std_logic;
SIGNAL Reset : std_logic;
SIGNAL Q : std_logic_vector(7 downto 0);
SIGNAL Empty : std_logic;
SIGNAL Full : std_logic;
BEGIN
```

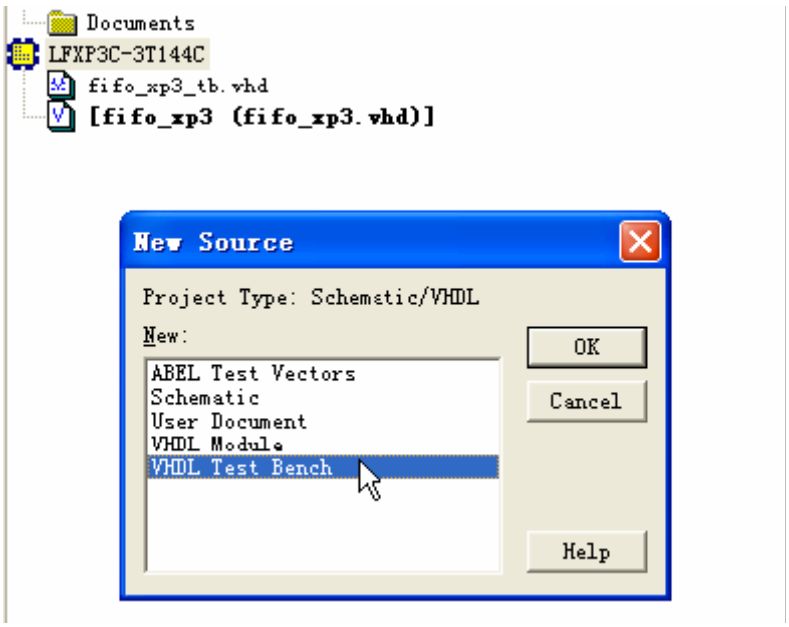
```
uut: fifo_xp3 PORT MAP(
Data => Data,
Clock => Clock,
WrEn => WrEn,
RdEn => RdEn,
Reset => Reset,
Q => Q,
Empty => Empty,
Full => Full
);

-- *** Test Bench - User Defined Section ***
tb : PROCESS
BEGIN
wait; -- will wait forever
END PROCESS;
-- *** End Test Bench - User Defined Section ***
END;
```

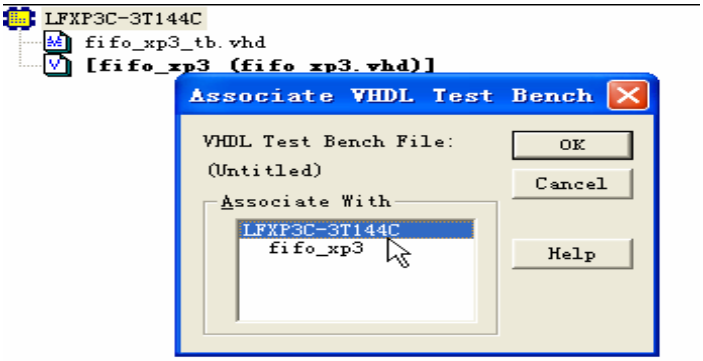
4. 在工程中新建一个.VHD文件



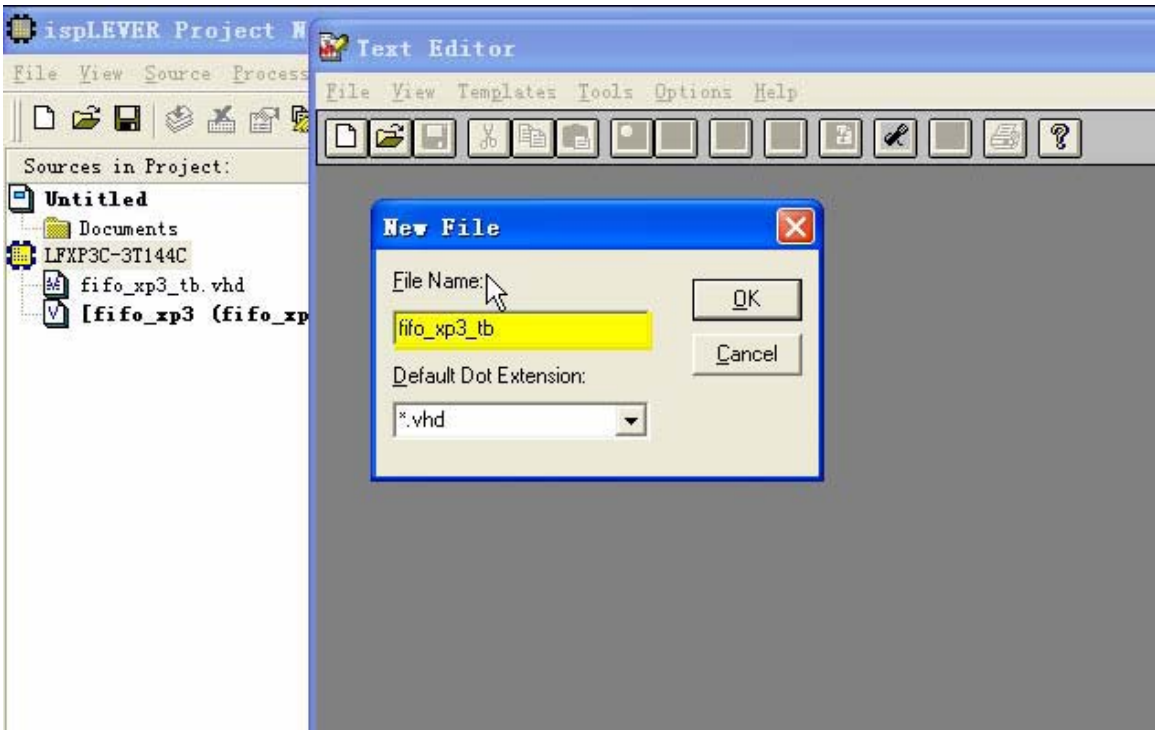
出现如下对话框，



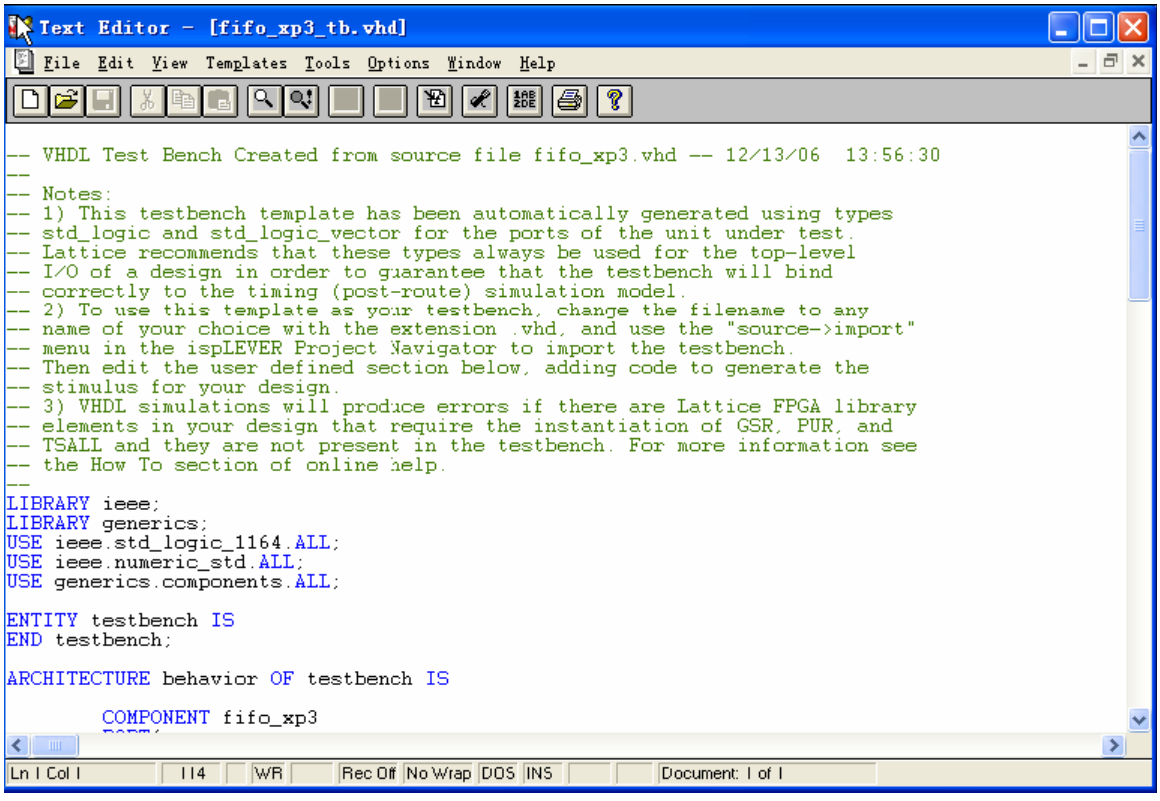
选择“VHDL Test Bench”，点击OK，然后选择要针对的仿真文件或器件



点击OK，填入文件名点击OK



然后把刚刚生成的测试模版文件内容复制粘贴到新建的文件中

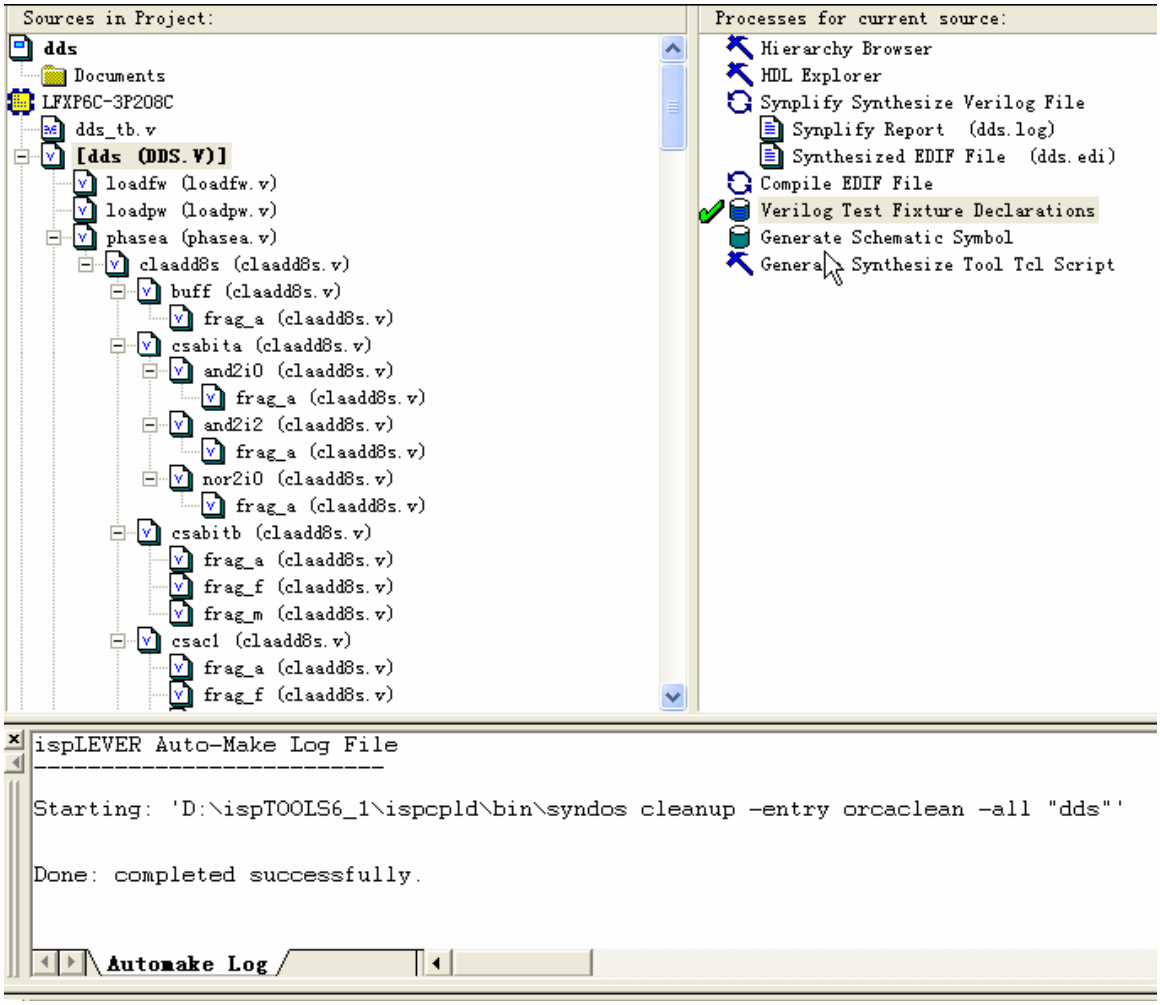


最后在这个文件中添加信号激励信息，测试文件就编写完成，可

以利用测试文件进行仿真。

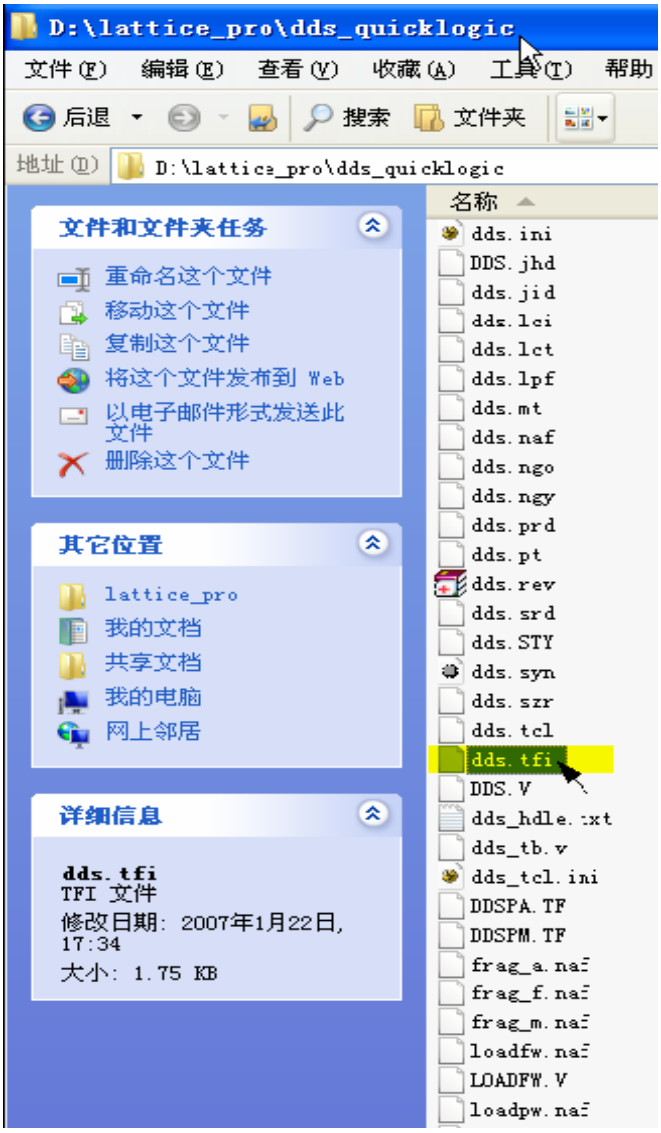
VERILOG 测试模版生成:

- 首先选择工程中的源文件，然后双击工具右侧“Processes for current source”中的 Verilog Test Fixture Declarations 选项



信息框出现成功提示，且 Verilog Test Fixture Declarations 前出现绿色的勾就表示已经生成“*.TFI”文件，但这个文件还不能直接用来进行仿真使用，需要进行部分修改;

- 然后在工程目录下找到*.tfi文件，将其后缀名改成*.TF文件;



- 直接导入工程，然后编辑*.TF文件： 源文件:

```
// TOOL: vlog2tf
// DATE: 01/22/07 17:34:37
// TITLE: Lattice Semiconductor Corporation
// MODULE: dds
// DESIGN: dds
// FILENAME: dds.tfi
// PROJECT: dds
// VERSION: 1.0
// NOTE: DO NOT EDIT THIS FILE
//
This file is generated by the Verilog Test
Fixture Declarations process and
// contains an I/O and instance declarations of
```

```
the Verilog source file
// you selected from the Sources in Project list.
// Notes:
// 1) This include file (.tfi) should be
referenced by your text fixture using
// the `include compile directive using the
syntax: `include "<file_name>.tfi"
// 2) If your design I/O changes, rerun the
process to obtain new I/O and
// instance declarations.
// 3) Verilog simulations will produce errors
if there are Lattice FPGA library
// elements in your design that require the
instantiation of GSR, PUR, and TSALL
// and they are not present in the test fixture.
For more information see the
// How To section of online help.
// Inputs
reg RESETN;
reg PNCLK;
reg SYSCLK;
reg [31:0] FREQWORD;
reg FWWRN;
reg [7:0] PHASEWORD;
reg PWWRN;
// Outputs
wire IDATA;
wire QDATA;
wire COS;
wire SIN;
wire MCOS;
wire MSIN;
wire DACCLK;
wire [7:0] DACOUT;
// Bidirs

// Instantiate the UUT
dds UUT (
.RESETN(RESETN),
.PNCLK(PNCLK),
.SYSCLK(SYSCLK),
.FREQWORD(FREQWORD),
.FWWRN(FWWRN),
.PHASEWORD(PHASEWORD),
```

```
.PWWRN(PWWRN),
.IDATA(IDATA),
.QDATA(QDATA),
.COS(COS),
.SIN(SIN),
.MCOS(MCOS),
.MSIN(MSIN),
.DACCLK(DACCLK),
.DACOUT(DACOUT)
);
// Initialize Inputs
`ifdef auto_init
initial begin
RESETN = 0;
PNCLK = 0;
SYSCLK = 0;
FREQWORD = 0;
FWWRN = 0;
PHASEWORD = 0;
PWWRN = 0;
end
`endif
```

————————————————————————————————

—

加头:

```
`timescale 1ns/1ps
`define auto_init
module tst_xx;
```

加尾:

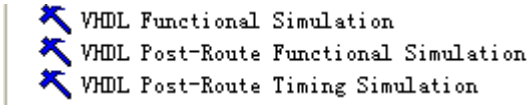
```
endmodule
```

- 修改激励: 改initial里的内容。这样整个VERILOG测试文件生成完成，可以用于仿真。

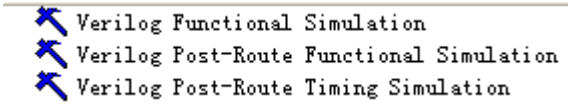
测试文件编写好以后，在开发工具的左侧的“source in projects”中选中这个文件，右侧的“process for current source”窗口中出

现如下3个选项：

VHDL



VERILOG



根据需要，双击这3个选项中的一个。第一个选项是布局布线前的功能仿真，也称为前仿真；第二个是布局布线后的功能仿真，也就是后功能仿真；第三个是布局布线后的时序仿真，通常称为后仿真。后仿真是带有布局布线后的器件相关延时信息的仿真，是设计在真实器件中的表现的一个可视化体现。双击后软件自动启动modelsim仿真工具，运行仿真。运行时间根据用户需要可以进行设置（两种方式：命令行输入RUN xx (TIME)；在波形显示窗口工具栏输入需要仿真的时间，然后点击RUN即可）。

可以自行比较分析波形，也可以通过前面叙述的方法编写测试代码使得测试可以自动比较波形或者结果。

*ENJOY YOURSELF!

*ONLY FOR REFERENCE（refer to xilinx application note：test benches）

陈家钧 Lattice 产品线 应用工程师(FAE)
威健国际贸易（上海）有限公司深圳分公司 福州办事处
手机：13960772601 电话：0591-87589289
邮箱：jason.chen@weikeng.com.cn
MSN：chenjj79@163.com
网址：http://www.latticesemi.com
http://www.weikeng.com.cn