Quartus 11 中调用 ModelSim-Altera 6.5 实例指导

编者：Ourfpga　www.Ourfpga.com
备注：参考网络上文档及代码编写此文档，在此对他们表示感谢！

此处默认您已经安装好 quartus11.0 软件。。。

一.Modelsim_Altera_ase 软件安装

睿智 FPGA 开发板配套光盘内提供了 Modelsim_Altera_ase 的安装包，ase 版本是 altera start edition，即入门版，免费使用的；ae 是 altera edition，，需要破解，支持更多功能吧。

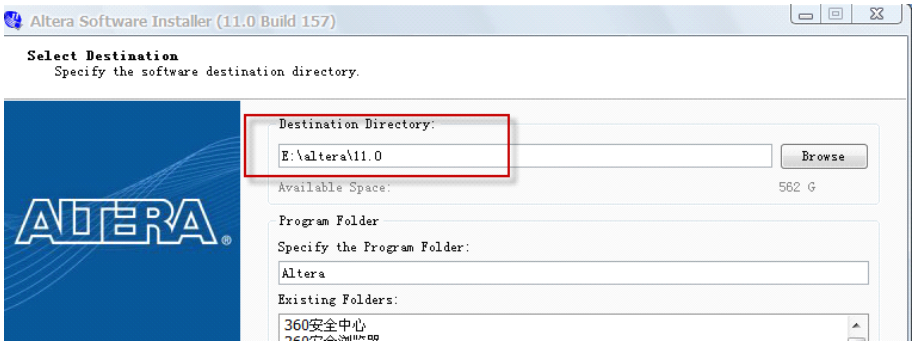
但对于我们，ase 版本已经足够了。装上就能使用。就不费破解的事了。

如想安装 ae 版本，请参考 Bingo 写过的教程，网页地址如下：

http://www.cnblogs.com/crazybingo/archive/2011/02/21/1959893.html

此处安装11.0_Altera_Modelsim_ase_windows.exe，。具体步骤如下：

（1）打开安装目录下的 setup，一路 next，直到选择路径的时候，选择与 quartus ii 安装目录相同的路径。如下图所示，我的电脑上装在 E 盘上了，您要根据您的设置来改。



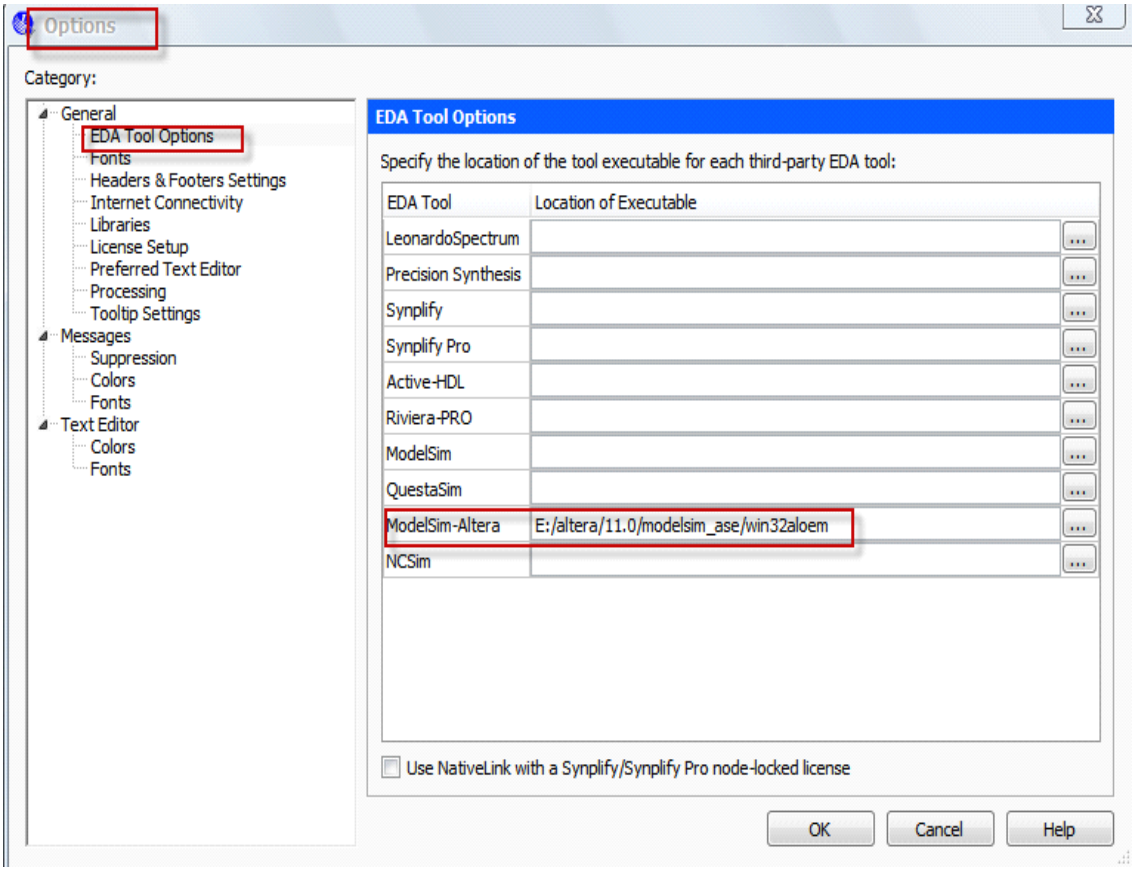
（2）继续 next，静默，等待安装完毕……

（3）安装完毕，出现以下界面：

（4）OK，Finish！

（5）打开 Quartus II，打开菜单 Tool-Options，在 EDA Tool Options 中的 Modelsim-Altera，选择 Moldelsim-Altera 应用程序的根目录，配置 Modelsim-Altera

应用程序第三方软件路径。如下图所示：在该选项卡中下面的 ModelSim-Altera 一项指定安装路径为 E:/Altera/11.0/modelsim_ae/win32aloem（其中 E:/Altera/11.0/modelsim_ae/为我电脑中 ModelSim-Altera 6.5e 的安装路径）

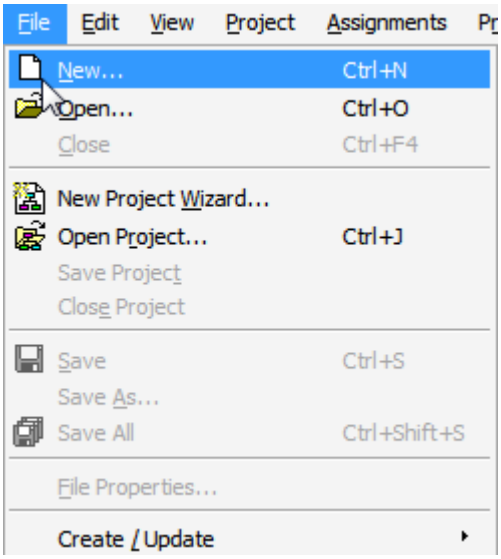


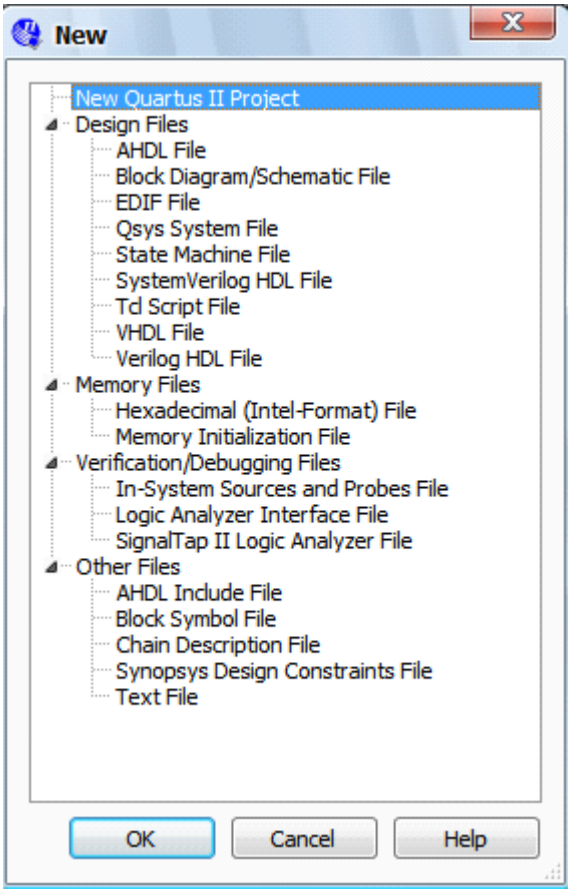
至此，Altera-Modelsim ase 版本安装完毕

二．如何在 Quartus II 11.0 中调用 ModelSim-Altera

我以一个简单的实例来描述整个过程：

先弄一个工程，打开 QuartusII，菜单 file---new，　新建一个工程





新建一个 verilog HDL File，代码：

```
module modelsim_test(clk,rst_n,div);
input clk;
input rst_n;

output div;
reg div;
```

```
always@(posedge clk or negedge rst_n)
    if(!rst_n)div<=1'b0;
    else div<=~div;

Endmodule
```

很简单的一个代码，是二分频电路。

我们要做什么呢，就是对这个电路进行 Modelsim 仿真，
下面，我们要设置一下：

在 Quartus II 11.0 界面菜单栏中选择 Assignments->Settings。

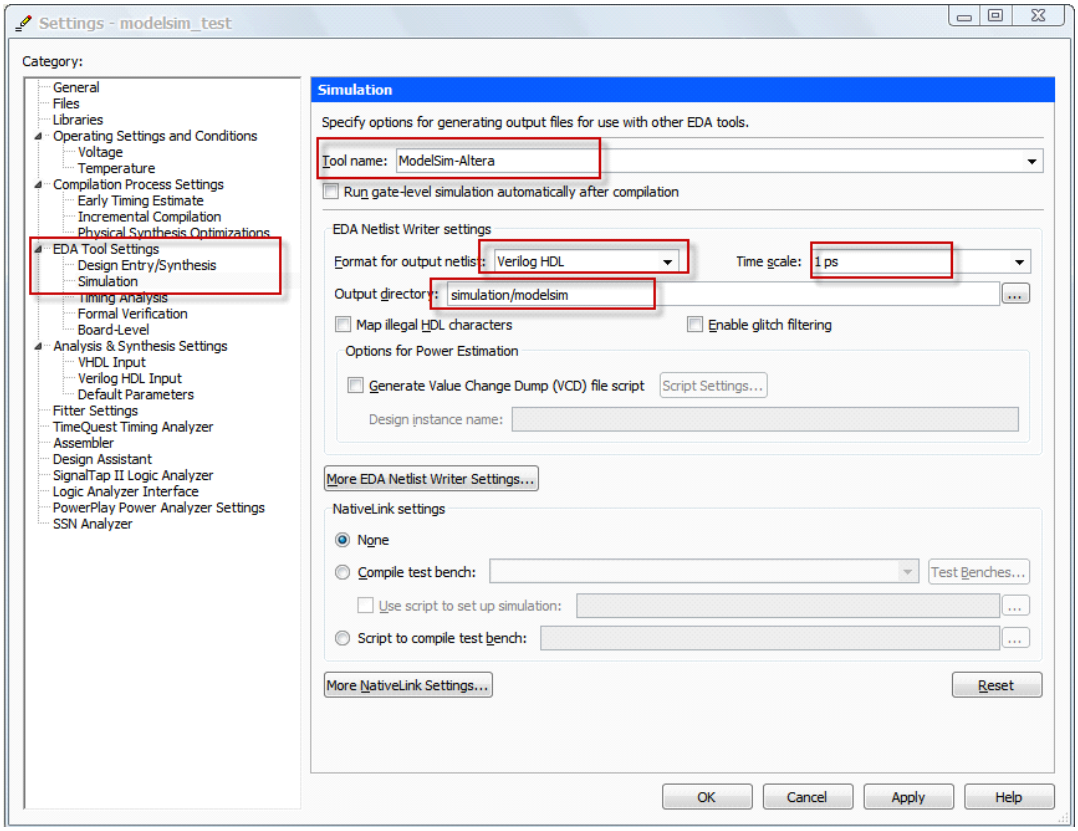
1.选中该界面下 EDA Tool settings 中的 Simulation 一项；

2.Tool name 中选择 ModelSim-Altera；

3.Format for output netlist 中选择开发语言的类型 Verilog 或者 VHDL 等，

4.Time scale 指定时间单位级别

5.Output directory 指定测试文件模板的输出路径（该路径是工程文件的相对路径）。



进入到一个关键步骤：

生成仿真测试文件

选择 Quartus II 11.0 开发界面菜单栏下 Processing->Start->Start Test Bench Template Writer，提示生成成功。这个生成的仿真测试文件（ modelsim_test 工程文件下 modelsim 下目录下找到后缀名为".vt"的文件）并根据自己需要进行编辑。下面是生成的文件原样，还没改：

```
`timescale 1 ps/ 1 ps
module modelsim_test_vlg_tst();
// constants
// general purpose registers
reg eachvec;
// test vector input registers
reg clk;
reg rst_n;
// wires
wire div;

// assign statements (if any)
modelsim_test i1 (
// port map - connection between master ports and signals/registers
    .clk(clk),
    .div(div),
    .rst_n(rst_n)
);
initial
begin
// code that executes only once
// insert code here --> begin

// --> end
$display("Running testbench");
end
always
// optional sensitivity list
// @(event1 or event2 or .... eventn)
begin
// code executes for every event on sensitivity list
// insert code here --> begin

@eachvec;
// --> end
end
endmodule
```

注意：QuartusII 中 testbench 文件的后缀是.vt，产生的模板文件只是包含了端口映射，端口声明等，具体的功能还是需要设计者自己编写，下面我们在模板上修改，编写 testbench，代码如下：

```
timescale 1 ps/ 1 ps
module modelsim_test_vlg_tst();
reg clk;
reg rst_n;
wire div;
```

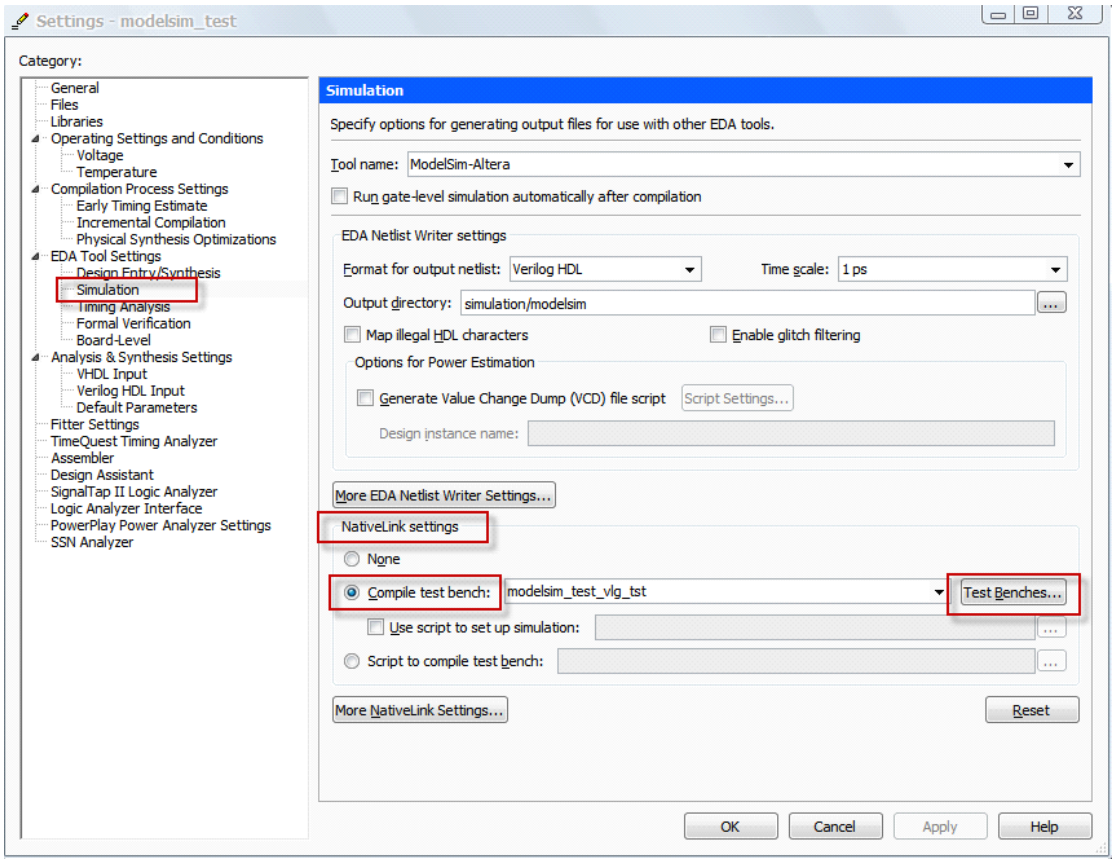
```
modelsim_test i1(

    .clk(clk),
    .div(div),
    .rst_n(rst_n)
);

initial
begin
      clk=0;
       forever
       #10 clk=~clk;

end

initial
begin
  rst_n=0;
  #1000 rst_n=1;
  #1000;
  $stop;

end
endmodule
```

代码为什么这么写，就涉及到 testbench 的编写方法学习，这个不在本文档的讨论范围，不过文档最后，附上了网上找到的一个 testbench 编写教学，大家可以参考。

下面是很关键的步骤，请一定看好，你如果设错了，就不能成功。

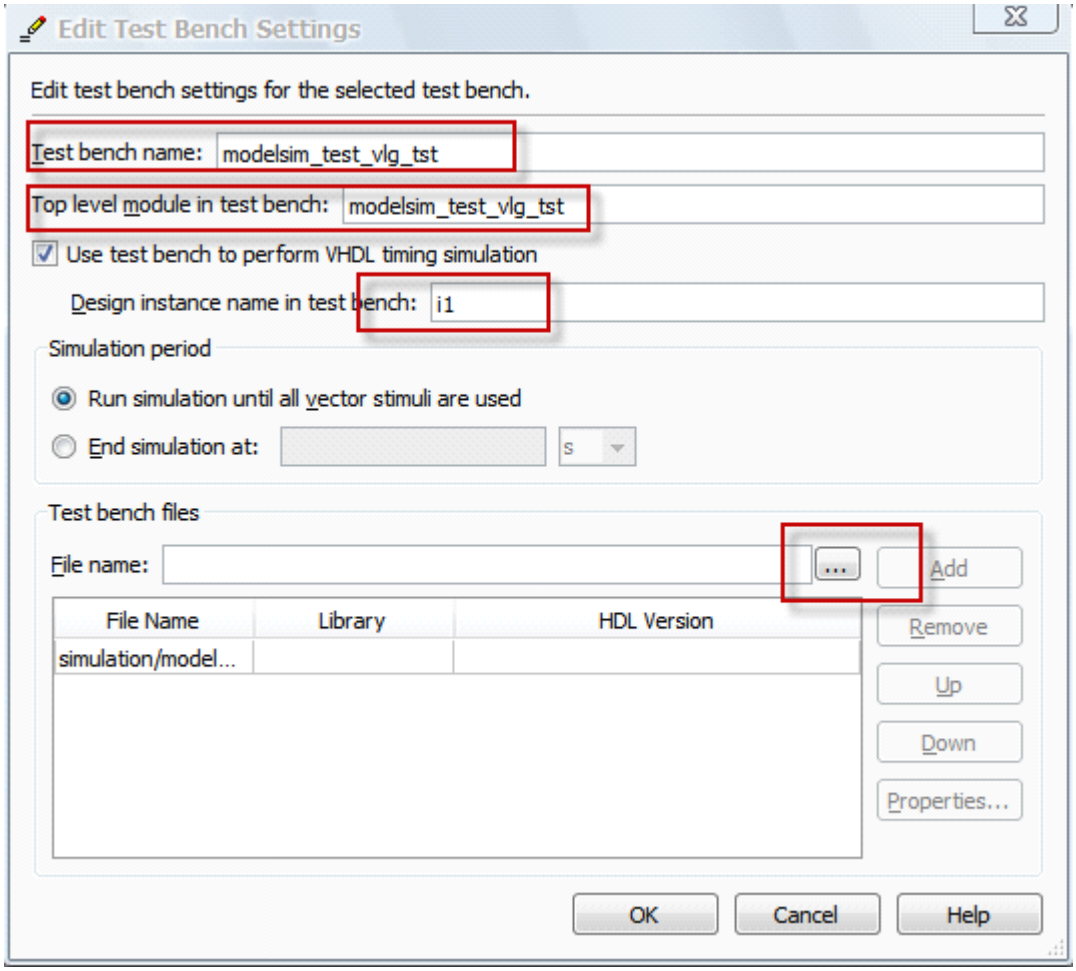
在 Quartus II 11.0界面菜单栏中选择 Assignments->Settings。

1.选中该界面下 EDA Tool settings 中的 Simulation 一项；在 NativeLink settings 中选择 Compile test bench 并点击后面的 Test Benches

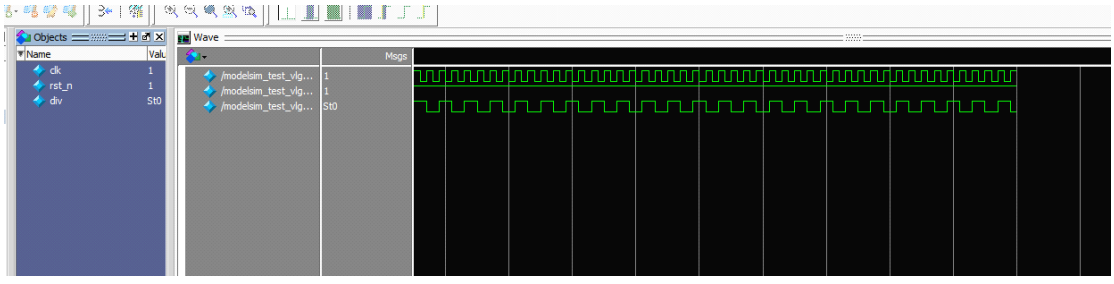


在 Test Benches 中点击 New
见下图：
在 Test bench name 栏要填写刚刚我们创建的 testbench 文件的实体名
即 modelsim_test_vlg_tst;
在 Top level modele in test bench 中也填写 modelsim_test_vlg_tst;
在 Design instance name in test bench 中 i1
这里你可以直接从 testbench 的文件里直接复制过来，避免手误写错。

然后在 Test bench files 中浏览添加 testbench 文件，然后记住点击 add,一步一步 OK。



一切准备就绪，下面在 Quartus II 11.0界面菜单栏中选择菜单栏 Tools 中的 Run EDA Simulation Tool->EDA RTL Simulation 进行行为级仿真，接下来就可以看到 ModelSim-Altera 6.5e 的运行界面，观察仿真波形。



可以看到 div 信号是 clk 的二分频了。

如何编写 testbench 的总结（非常实用的总结）

1．激励的设置

相应于被测试模块的输入激励设置为 reg 型，输出相应设置为 wire 类型，双向端口 inout 在测试中需要进行处理。

方法 1：为双向端口设置中间变量 inout_reg 作为该 inout 的输出寄存，inout 口在 testbench 中要定义为 wire 型变量，然后用输出使能控制传输方向。

eg：

inout [0:0] bi_dir_port;

wire [0:0] bi_dir_port;

reg [0:0] bi_dir_port_reg;

reg bi_dir_port_oe;

assign bi_dir_port=bi_dir_port_oe?bi_dir_port_reg:1'bz;
用 bi_dir_port_oe 控制端口数据方向，并利用中间变量寄存器改变其值。等于两个模块之间用 inout 双向口互连。往端口写（就是往模块里面输入)

方法 2：使用 force 和 release 语句，这种方法不能准确反映双向端口的信号变化，但这种方法可以反映块内信号的变化。具体如示：

```
module test();
wire data_inout;
reg data_reg;
reg link;
#xx; //延时
force data_inout=1'bx; //强制作为输入端口
.....
#xx;
release data_inout; //释放输入端口
endmodule
```

从文本文件中读取和写入向量
1）读取文本文件：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：

```
reg [7:0] mem[1:256] // a 8-bit, 256-word 定义存储器 mem
initial $readmemh ( "mem.data", mem ) // 将.dat 文件读入寄存器 mem 中
initial $readmemh ( "mem.data", mem, 128, 1 ) // 参数为寄存器加载数据的地址始终
```

2）输出文本文件：打开输出文件用?$fopen 例如：

```
integer out_file; // out_file 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
```

设计中的信号值可以通过$fmonitor, $fdisplay,

2. Verilog 和 Ncverilog 命令使用库文件或库目录

ex). ncverilog -f run.f -v lib/lib.v -y lib2 +libext+.v //一般编译文件在 run.f 中，库文件在 lib.v 中,lib2 目录中的.v 文件系统自动搜索
使用库文件或库目录,只编译需要的模块而不必全部编译

3．Verilog Testbench 信号记录的系统任务:

1). SHM 数据库可以记录在设计仿真过程中信号的变化. 它只在 probes 有效的时间内记录你 set probe on 的信号的变化.

ex). $shm_open("waves.shm"); //打开波形数据库

$shm_probe(top, "AS"); // set probe on "top",

第二个参数: A -- signals of the specific scrope

S -- Ports of the specified scope and below, excluding library cells

C -- Ports of the specified scope and below, including library cells

AS -- Signals of the specified scope and below, excluding library cells

AC -- Signals of the specified scope and below, including library cells

还有一个 M ,表示当前 scope 的 memories, 可以跟上面的结合使用, "AM" "AMS" "AMC"

什么都不加表示当前 scope 的 ports;

$shm_close //关闭数据库

2). VCD 数据库也可以记录在设计仿真过程中信号的变化. 它只记录你选择的信号的变化.

ex). $dumpfile("filename"); //打开数据库

$dumpvars(1, top.u1); //scope = top.u1, depth = 1

第一个参数表示深度, 为 0 时记录所有深度; 第二个参数表示 scope,省略时表当前的 scope.

$dumpvars; //depth = all scope = all

$dumpvars(0); //depth = all scope = current

$dumpvars(1, top.u1); //depth = 1 scope = top.u1

$dumpoff //暂停记录数据改变,信号变化不写入库文件中

$dumpon //重新恢复记录

3). Debussy fsdb 数据库也可以记录信号的变化,它的优势是可以跟 debussy 结合,方便调试.

如果要在 ncverilog 仿真时,记录信号, 首先要设置 debussy:

a. setenv LD_LIBRARY_PATH LD_LIBRARY_PATH

(path for debpli.so file (/share/PLI/nc_xl//nc_loadpli1))

b. while invoking ncverilog use the +ncloadpli1 option.

ncverilog -f run.f +debug +ncloadpli1=debpli:deb_PLIPtr

fsdb 数据库文件的记录方法,是使用$fsdbDumpfile 和$fsdbDumpvars 系统函数,使用方法参见 VCD

注意: 在用 ncverilog 的时候,为了正确地记录波形,要使用参数: "+access+rw", 否则没有读写权限

在记录信号或者波形时需要指出被记录信号的路径，如：tb.module.u1.clk.

……

关于信号记录的系统任务的说明：

在 testbench 中使用信号记录的系统任务，就可以将自己需要的部分的结果以及波形文件记录下来（可采用 sigalscan 工具查看），适用于对较大的系统进行仿真，速度快，优于全局仿真。使用简单，在 testbench 中添加：

initial begin

$shm_open("waves.shm");

$shm_probe("要记录信号的路径“，”AS“）；

＃10000

$shm_close; 即可。

4. ncverilog 编译的顺序: ncverilog file1 file2 ....

有时候这些文件存在依存关系,如在 file2 中要用到在 file1 中定义的变量,这时候就要注意其编译的顺序是

从后到前,就先编译 file2 然后才是 file2.

5. 信号的强制赋值 force

首先, force 语句只能在过程语句中出现,即要在 initial 或者 always 中间. 去除 force 用 release 语句.

initial begin force sig1 = 1'b1; ... ; release sig1; end

force 可以对 wire 赋值,这时整个 net 都被赋值; 也可以对 reg 赋值.

6．加载测试向量时，避免在时钟的上下沿变化

为了模拟真实器件的行为，加载测试向量时，避免在时钟的上下沿变化，而是在时钟的上升沿延时一个时间单位后，加载的测试向量发生变化。如：

assign #5 c=a^b

……

@(posedge clk) #(0.1*`cycle) A=1;

****

//testbench 的波形输出

module top;

...

initial

begin

$dumpfile("./top.vcd"); //存储波形的文件名和路径,一般是.vcd 格式.

$dumpvars(1,top); //存储 top 这一层的所有信号数据

$dumpvars(2,top.u1); //存储 top.u1 之下两层的所有数据信号(包含 top.u1 这一层)

$dumpvars(3,top.u2); //存储 top.u2 之下三层的所有数据信号(包含 top.u2 这一层)

```
$dumpvars(0,top.u3); //存储 top.u3 之下所有层的所有数据信号
end
endmodule

//产生随机数,seed 是种子
$random(seed);
ex: din <= $random(20);

//仿真时间,为 unsigned 型的 64 位数据
$time
ex:
...
time condition_happen_time;
...
condition_happen_time = $time;
...
$monitor($time,"data output = %d", dout);
...
//参数
parameter para1 = 10,
para2 = 20,
para3 = 30;

//显示任务
$display();
//监视任务
$monitor();

//延迟模型
specify
...
//describ pin-to-pin delay
endspecify

ex:
module nand_or(Y,A,B,C);
```

input A,B,C;

output Y;

AND2 #0.2 (N,A,B);

OR2 #0.1 (Y,C,N);

specify

(A*->Y) = 0.2;

(B*->Y) = 0.3;

(C*->Y) = 0.1;

endspecify

endmodule

//时间刻度

`timescale 单位时间/时间精确度

//文件 I/O

1.打开文件

integer file_id;

file_id = fopen("file_path/file_name");

2.写入文件

//$fmonitor 只要有变化就一直记录

$fmonitor(file_id, "%format_char", parameter);

eg fmonitor(file_id, "%m: %t in1=%d o1=%h", $time, in1, o1);

//$fwrite 需要触发条件才记录

$fwrite(file_id, "%format_char", parameter);

//$fdisplay 需要触发条件才记录

$fdisplay(file_id, "%format_char", parameter);

$fstrobe();

3.读取文件

integer file_id;

file_id = $fread("file_path/file_name", "r");

4.关闭文件

$fclose(fjile_id);

5.由文件设定存储器初值

$readmemh("file_name", memory_name"); //初始化数据为十六进制

$readmemb("file_name", memory_name"); //初始化数据为二进制

//仿真控制

$finish(parameter); //parameter = 0,1,2

$stop(parameter);

//读入 SDF 文件

$sdf_annotate("sdf_file_name", module_instance, "scale_factors");

//module_instance: sdf 文件所对应的 instance 名.

//scale_factors:针对 timming delay 中的最小延时 min,典型延迟 typ,最大延时 max 调整延迟参数

//generate 语句,在 Verilog-2001 中定义.用于表达重复性动作

//必须事先声明 genvar 类型变量作为 generate 循环的指标

eg:

genvar i;

generate for(i = 0; i < 4; i = i + 1)

begin

assign = din *= i % 2;*

end

endgenerate

//资源共享

always @(A or B or C or D)

sum = sel ? (A+B) C+D);

//上面例子使用两个加法器和一个 MUX,面积大

//下面例子使用一个加法器和两个 MUX,面积小

always @(A or B or C or D)

begin

tmp1 = sel ? A:C;

tmp2 = sel ? B ;

end

always @(tmp1 or tmp2)

sum = tmp1 + tmp2;

****

模板：

module testbench; //定义一个没有输入输出的 module

reg …… //将 DUT 的输入定义为 reg 类型

……

wire…… //将 DUT 的输出定义为 wire 类型

……

//在这里例化 DUT

initial

begin

…… //在这里添加激励(可以有多个这样的结构)

end

always…… //通常在这里定义时钟信号

initial

//在这里添加比较语句(可选)

end

initial

//在这里添加输出语句(在屏幕上显示仿真结果)

end

endmodule

一下介绍一些书写 Testbench 的技巧：

1.如果激励中有一些重复的项目，可以考虑将这些语句编写成一个 task，这样会给书写和仿真带来很大方便。例如，一个存储器的 testbench 的激励可以包含 write，read 等 task。

2.如果 DUT 中包含双向信号(inout)，在编写 testbench 时要注意。需要一个 reg 变量来表示其输入，还需要一个 wire 变量表示其输出。

3.如果 initial 块语句过于复杂，可以考虑将其分为互补相干的几个部分，用数个 initial 块来描述。在仿真时，这些 initial 块会并发运行。这样方便阅读和修改。

4.每个 testbench 都最好包含$stop 语句，用以指明仿真何时结束。

最后提供一个简单的示例(转自 Xilinx 文档)：

DUT：

```
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
```

Testbench：

```
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
```