高 ◆ 等 ◆ 学 ◆ 校 ◆ 教 ◆ 材

Verilog HDL 实验练习与语法手册

夏宇闻 编著

高等教育出版社

高等学校教材

Verilog HDL 实验练习与语法手册

夏宇闻　编著

高等教育出版社

内容提要

本书是《数字系统设计——Verilog 实现》(夏宇闻编著)的配套辅导用书,为想真正掌握 Verilog HDL 设计方法的读者精心设计了丰富的上机练习和范例,并附有常用语法手册,能有效地帮助读者理解主教材中讲解的知识,并将其用到实践当中去。本书可以与主教材配套使用,也可单独作为高等学校电子信息、计算机等相关专业本科生和研究生学习数字电路设计的参考用书,也可供其他工程设计人员参考使用。

图书在版编目(CIP)数据

Verilog HDL 实验练习与语法手册/夏宇闻编著.
2 版. —北京: 高等教育出版社, 2006.1
ISBN 7-04-017199-6

Ⅰ.V... Ⅱ.夏... Ⅲ.①数字系统-系统设计-高等学校-教学参考资料②硬件描述语言,VHDL-程序设计-高等学校-教学参考资料 Ⅳ.①TP271②TP312

中国版本图书馆 CIP 数据核字(2005)第 155479 号

策划编辑 董建波 **责任编辑** 董建波
封面设计 张申申 **责任印制** 孔 源

出版发行 高等教育出版社
社 址 北京市西城区德外大街 4 号
邮政编码 100011
总 机 010-58581000

经 销 蓝色畅想图书发行有限公司
印 刷 北京铭成印刷有限公司

开 本 787×1092 1/16
印 张 12.5
字 数 250 000

购书热线 010-58581118
免费咨询 800-810-0598
网 址 http://www.hep.edu.cn
http://www.hep.com.cn
网上订购 http://www.landraco.com
http://www.landraco.com.cn
畅想教育 http://www.widedu.com
版 次 2002 年 12 月第 1 版
2006 年 1 月第 2 版
印 次 2006 年 1 月第 1 次印刷
定 价 18.60 元

本书如有缺页、倒页、脱页等质量问题,请到所购图书销售部门联系调换。

版权所有 侵权必究

物料号 17199-00

前　言

数字电路设计一直都被认为是既难教，又难学的课程。学好这门课程除了要很好地掌握数字电路的基本概念和理论外，最关键的是要有能力把理论真正应用到设计实践中去，因此学生自己认真地、反复地进行上机练习和思考对于掌握 Verilog 语言和设计技术具有极其重要的意义。

本书是《数字系统设计——Verilog 实现》（夏宇闻编著）的配套辅导用书，为想真正掌握 Verilog HDL 设计方法的读者精心设计了丰富的上机练习范例和思考题，并附有常用语法手册，能有效地帮助读者理解主教材中的知识点，并将其应用到设计实践中。

全书共提供了与主教材配套的 12 个典型的设计练习示范，可以基本满足初级 Verilog 学习者的需求。为了方便初学者，按字母顺序给出了常用 Verilog 语法的参考手册，读者可以随时查询不清楚的内容。

这套书是作者十多年来在数字系统 Verilog 设计教学方面的经验总结。本书可以与主教材配套使用，也可以作为工程设计人员的参考用书。

编者

2005 年 8 月

郑重声明

高等教育出版社依法对本书享有专有出版权。任何未经许可的复制、销售行为均违反《中华人民共和国著作权法》，其行为人将承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。为了维护市场秩序，保护读者的合法权益，避免读者误用盗版书造成不良后果，我社将配合行政执法部门和司法机关对违法犯罪的单位和个人给予严厉打击。社会各界人士如发现上述侵权行为，希望及时举报，本社将奖励举报有功人员。

反盗版举报电话：(010) 58581897/58581896/58581879

传　　真：(010) 82086060

E - mail：dd@hep.com.cn

通信地址：北京市西城区德外大街 4 号
高等教育出版社打击盗版办公室

邮　　编：100011

购书请拨打电话：(010)58581118

目　　录

第一部分　设计示范与实验练习

在主教材《数字系统设计——Verilog 实现》(夏宇闻编著)学习的基础上，通过完成本书 12 个实验的练习，一定能逐步掌握 Verilog HDL 设计的要点。读者可以先理解设计示范模块中每一条语句的作用，进行功能仿真来加深理解，然后对示范模块进行综合，再分别进行综合后生成的逻辑网表和布线后生成的带布线延迟的门级器件网表的时序仿真，以加深印象和深入理解。在此基础上再独立完成每一阶段规定的练习。当 12 个实验练习做完后，便可以开始设计一些简单的逻辑电路和系统。最好有一到两个月的集中训练时间，有兴趣的读者很快就能设计相当复杂的数字逻辑系统。当然，复杂的数字系统的设计和验证，不但需要系统结构知识和丰富经验的积累，还需要了解更多的语法现象和掌握高级的 Verilog HDL 系统任务，以及与 C 语言模块接口(的方法即 PLI)，这些已超出的本书的范围。有兴趣的读者可以阅读 Verilog 语法参考资料和有关文献，自己学习。

练习一　简单的组合逻辑设计

目的：(1) 掌握基本组合逻辑电路的实现方法；

(2) 初步了解两种基本组合逻辑电路的生成方法；

(3) 学习测试模块的编写；

(4) 通过综合和布局布线了解不同层次仿真的物理意义。

下面的模块描述一个可综合的数据比较器。从语句可以很容易看出它的功能是比较数据 a 与 b，如果两个数据相同，则给出结果 1，否则给出结果 0。在 Verilog HDL 中，描述组合逻辑时常使用 assign 结构。注意，equal = (a == b)? 1:0，这是一种在组合逻辑实现分支判断时常使用的格式。

模块源代码(方法一)如下：

```
//-----------------------------文件名 compare.v ----------------------------
module compare(equal,a,b);
    input a,b;
    output equal;
        assign equal = (a==b)?1 : 0;
```

```
        //a 等于 b 时,equal 输出为 1;a 不等于 b 时,equal 输出为 0
endmodule
```

模块源代码(方法二)如下:

```
module compare(equal,a,b);
    input a,b;
    output equal;
    reg equal;
        always @(a or b)
        if(a == b)              //a 等于 b 时,equal 输出为 1
          equal = 1;
        else                    //a 不等于 b 时,equal 输出为 0
          equal = 0;            //思考:如果不写 else 部分会产生什么逻辑
endmodule
```

测试模块用于检测模块设计得是否正确。它给出模块的输入信号,观察模块的内部信号和输出信号,如果发现结果与预期的有偏差,则需要对设计模块进行修改。

测试模块源代码(方法之一)如下:

```
`timescale 1ns/1ns          //定义时间单位
`include "./compare.v"      //包含模块文件。在有的仿真调试环境中并不需要此语句,
                            //而需要从调试环境的菜单中键入有关模块文件的路径和名称
module t;
    reg a,b;
    wire equal;
    initial                 //initial 常用于仿真时信号的给出
      begin
        a = 0;
        b = 0;
        #100  a = 0; b = 1;
        #100  a = 1; b = 1;
        #100  a = 1; b = 0;
        #100  a = 0; b = 0;
        #100  $stop;        //系统任务,暂停仿真以便观察仿真波形
      end
    compare m(.equal(equal),.a(a),.b(b));      //调用被测试模块 t.m
endmodule
```

综合就是把 compare.v 文件送到 synplify 或其他综合器处理,在选定实现器件和选取生

成 Verilog 网表的前提下，启动综合器的编译。综合器会自动生成一系列文件，向操作者报告综合的结果。其中生成的 Verilog Netlist 文件（扩展名为 .vm），表示自动生成的逻辑结构网表，仍然用 Verilog 语句表示，但比输入的源文件更具体，可以用测试模块调用它做同样的仿真。运行的结果更接近实际器件。

布局布线就是把综合后生成的另一种文件（EDIF），在布线工具控制下进行处理，启动布线工具的编译。布局布线工具会自动生成一系列文件，向操作者报告布局布线的结果。其中生成的 Verilog Netlist 文件（扩展名为 .vo）表示自动生成的具体基本门级结构和连接的延迟，仍然用 Verilog 基本部件结构语句和连接线的延迟参数的重新定义表示，库中的基本部件也更进一步具体化了，比综合后的扩展名为 .vm 的文件更具体。可以用同一个测试模块调用它做同样的仿真。运行结果与实际器件运行结果几乎完全一致。

仿真波形（部分）如图 1.1 所示。

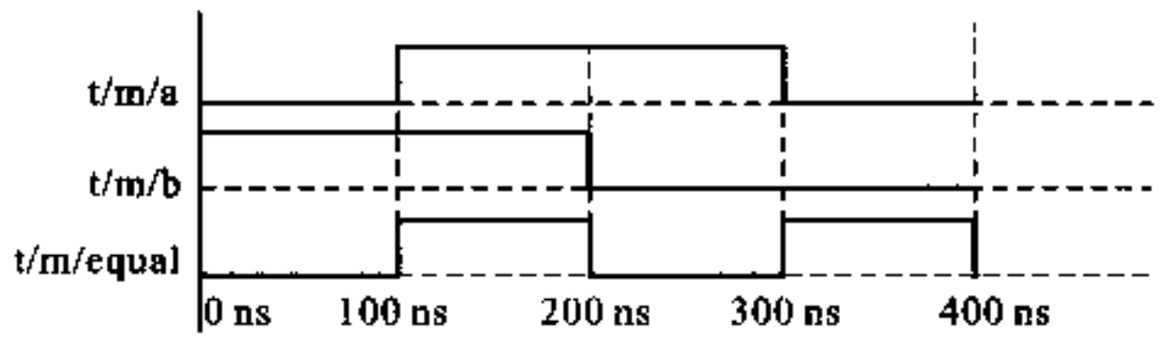


图 1.1　仿真波形

测试模块源代码（方法之二）如下：

```
`timescale 1ns/1ns          //定义时间单位
`include "./compare.v"      //包含模块文件。在有的仿真调试环境中并不需要此语句，
                            //而需要从调试环境的菜单中键入有关模块文件的路径和名称

module t;
    reg a,b;
reg clock;
wire equal;
initial                     //initial 常用于仿真时信号的给出
  begin
    a=0;
    b=0;
    clock = 0;              //定义一个时钟变量
end
always #50 clock = ~clock;          //产生周期性的时钟
always @ (posedge clock)            //在每次时钟正跳变沿时刻产生不同的 a 和 b
  begin
```

```
        a = {$random} % 2;          //每次 a 是 0 还是 1 是随机的
        b = {$random} % 2;          //每次 b 是 0 还是 1 是随机的
      end
    initial
      begin #100000 $stop; end      //系统任务,暂停仿真以便观察仿真波形
    compare m( .equal(equal),.a(a),.b(b));    //调用被测试模块 t.m
endmodule
```

[练习题]　设计一个字节(8 位)的比较器。

要求:比较两个字节的大小,如 a[7:0]大于 b[7:0]输出高电平,否则输出低电平,改写测试模型,使其能进行比较全面的测试。观测 RTL 级仿真、综合后门级仿真和布线后仿真有什么不同,并说明引起这些不同的原因。从文件系统中查阅自动生成的 compare. vm 、compare. vo 文件,和 compare. v 做比较,说出它们的不同点和相同点。

[思考题]　在测试方法二中,第二个 initial 块有什么用? 它与第一个 initial 块有什么关系? 如果在第二个 initial 块中没有写#100000 或者 $stop,仿真会如何进行? 比较两种测试方法,哪一种测试方法更全面?

练习二　简单分频时序逻辑电路的设计

目的:(1) 掌握最基本时序电路的实现方法;

(2) 学习时序电路测试模块的编写;

(3) 学习综合和不同层次的仿真。

在 Verilog HDL 中,相对于组合逻辑电路,可综合成具体电路结构的时序逻辑电路也有标准的表述方式。在可综合的 Verilog HDL 模型中,通常使用 always 块和 @(posedge clk)或 @(negedge clk)的结构来表述时序逻辑。下面是一个二分频器的可综合模型。

```
//------------------------ 文件名:half_clk.v ------------------------
 module half_clk(reset,clk_in,clk_out);
     input clk_in,reset;
     output clk_out;
     reg clk_out;
     always @(posedge clk_in)
       begin
         if(! reset) clk_out =0;
         else        clk_out = ~clk_out;
```

```
        end
    endmodule
```

在 always 块中，被赋值的信号都必须定义为 reg 型，这是由时序逻辑电路的特点所决定的。对于 reg 型数据，如果未对它进行赋值，仿真工具会认为它是不定态。为了能正确地观察到仿真结果，并确定时序电路的起始相位，在可综合风格的模块中通常定义一个复位信号 reset，当 reset 为低电平时，对电路中的寄存器进行复位。

测试模块的源代码：

```
//-----------------------文件名 top.v -------------------------
`timescale 1ns/100ps
`define clk_cycle 50
module top;
    reg clk,reset;
    wire clk_out;
    always #`clk_cycle clk = ~clk; //产生测试时钟
    initial
      begin
        clk = 0;
        reset = 1;
        #10 reset = 0;
        #110 reset = 1;
        #100000 $stop;
        end
        half_clk m0( .reset(reset),.clk_in(clk),.clk_out(clk_out));
endmodule
```

仿真波形如图 1.2 所示。

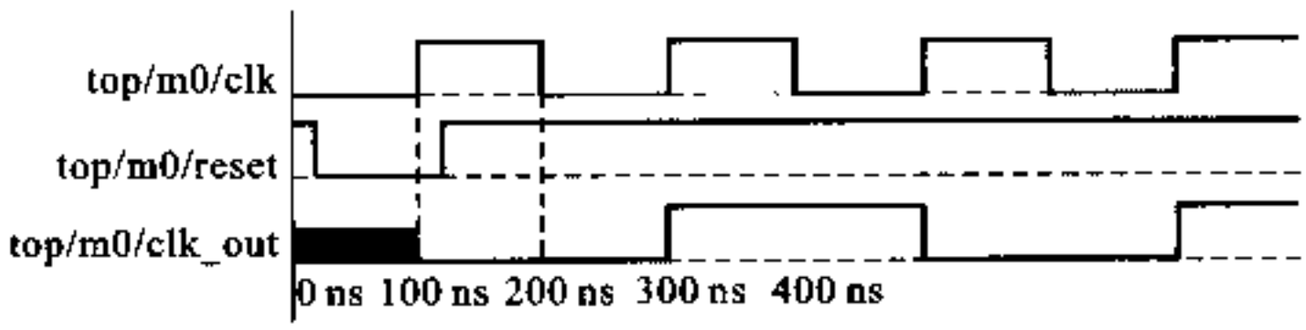


图 1.2　仿真波形

[练习题]　依然作 clk_in 的二分频 clk_out，要求输出时钟的相位与上例的输出正好反相。编写测试模块，给出仿真波形。改变输入时钟的频率，观测 RTL 级仿真、综合后门级仿

真和布线后仿真的不同,并写出报告。

[**思考题**]　如果没有 reset 信号,能否控制二分频 clk_out 信号的相位? 只用 clk 时钟沿的触发(即不用二分频产生的时钟沿)如何直接产生四分频、八分频或者十六分频的时钟? 如何只用 clk 时钟沿的触发(不用二分频产生的时钟沿)直接产生占空比不同的分频时钟?

练习三　利用条件语句实现计数分频时序电路

目的:(1) 掌握条件语句在简单时序模块设计中的使用;

(2) 学习在 Verilog 模块中应用计数器;

(3) 学习测试模块的编写、综合和不同层次的仿真。

与常用的高级程序语言一样,为了描述较为复杂的时序关系,Verilog HDL 提供了条件语句供分支判断时使用。在可综合风格的 Verilog HDL 模型中常用的条件语句有 if-else 和 case-endcase 两种结构,用法和 C 程序语言中类似。两者相比较,if-else 用于不很复杂的分支关系,实际编写可综合风格的模块特别是用状态机构成的模块时,更常用的是 case-endcase 风格的代码。下面给出的是有关 if-else 的范例,有关 case-endcase 结构的代码以后会经常用到。

下面给出的范例也是一个可综合风格的分频器,是将 10M 的时钟分频为 500K 的时钟。基本原理与二分频器是一样的,但是需要定义一个计数器,以便准确获得二十分频器。

模块源代码如下:

```
//---------------------------- fdivision.v ----------------------------
module fdivision(RESET,F10M,F500K);
    input F10M,RESET;
    output F500K;
    reg F500K;
    reg [7:0]j;
always @(posedge F10M)
if(! RESET)        //低电平复位
  begin
    F500K <= 0;
    j <= 0;
  end
else
  begin
    if(j ==19)   //对计数器进行判断,以确定 F500K 信号是否反转
      begin
```

```
            j <= 0;
            F500K <= ~F500K;
          end
        else
            j <= j+1;
      end
endmodule
```

测试模块源代码如下：

```
//------------------------ fdivision_Top.v ---------------------------
`timescale 1ns/100ps
`define clk_cycle 50
module division_Top;
    reg F10M,RESET;
    wire F500K_clk;
    always #`clk_cycle F10M_clk = ~F10M_clk;
    initial
      begin
      RESET=1;
      F10M=0;
      #100 RESET=0;
      #100 RESET=1;
      #10000 $stop;
    end
    fdivision fdivision (.RESET(RESET),.F10M(F10M),.F500K(F500K_clk));
endmodule
```

仿真波形如图 1.3 所示。

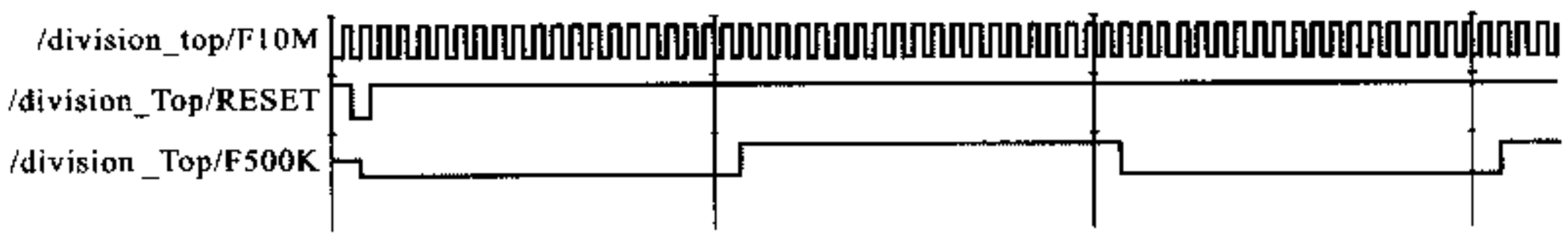


图 1.3　仿真波形

[**练习题**]　利用 10M 的时钟，设计一个单周期形状如图 1.4 所示的周期波形。

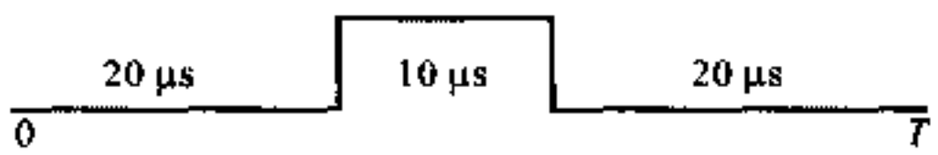


图 1.4　周期波形

练习四　阻塞赋值与非阻塞赋值的区别

目的：(1) 通过实验观察掌握阻塞赋值与非阻塞赋值的概念和区别；

(2) 了解非阻塞和阻塞赋值的不同使用场合；

(3) 学习测试模块的编写、综合和不同层次的仿真。

阻塞赋值与非阻塞赋值，在主教材中已经了解了它们之间在语法上的区别以及综合后所得到的电路结构上的区别。在 always 块中，阻塞赋值可以理解为赋值语句是顺序执行的，而非阻塞赋值可以理解为赋值语句是并发执行的。在实际的时序逻辑设计中，一般的情况下非阻塞赋值语句被更多地使用，有时为了在同一周期实现相互关联的操作，也使用了阻塞赋值语句(注意，在实现组合逻辑的 assign 结构中，必须采用阻塞赋值语句)。

下例通过分别采用阻塞赋值语句和非阻塞赋值语句的两个看上去非常相似的模块 blocking. v 和 non_blocking. v 来阐明两者之间的重要区别。

模块源代码如下：

```
//------------------------------ blocking.v ------------------------------
module blocking(clk,a,b,c);
    output [3:0] b,c;
    input  [3:0] a;
    input        clk;
    reg    [3:0] b,c;
    always @(posedge clk)
    begin
      b = a;
      c = b;
      $display("Blocking: a = %d, b = %d, c = %d ",a,b,c);
    end
endmodule
//---------------------------- non_blocking.v ----------------------------
```

```
module non_blocking(clk,a,b,c);
    output [3:0] b,c;
    input  [3:0] a;
    input        clk;
    reg    [3:0] b,c;
    always @(posedge clk)
    begin
      b <= a;
      c <= b;
      $display("Non_Blocking: a = %d, b = %d, c = %d ",a,b,c);
    end
endmodule
```

测试模块源代码如下：

```
//------------------------------ compareTop.v ------------------------------
`timescale 1ns/100ps
`include "./blocking.v"
`include "./non_blocking.v"
module compareTop;
    wire [3:0] b1,c1,b2,c2;
    reg  [3:0] a;
    reg        clk;
    initial
    begin
      clk = 0;
      forever #50 clk = ~clk; //思考:如果在本句后还有语句,能否执行? 为什么?
end
initial
begin
  a = 4'h3;
  $display("_____ _____");
  #100 a = 4'h7;
  $display("_____ _____");
  #100 a = 4'hf;
  $display("_____ _____");
  #100 a = 4'ha;
```

```
        $display("_____          ____ _");
        #100 a = 4'b2;
        $display("_____________________________");
        #100 $display("____ ____ ____ ____ _____");
        $stop;
      end
      non_blocking non_blocking(clk,a,b2,c2);
      blocking blocking(clk,a,b1,c1);
endmodule
```

仿真波形(部分)如图 1.5 所示。

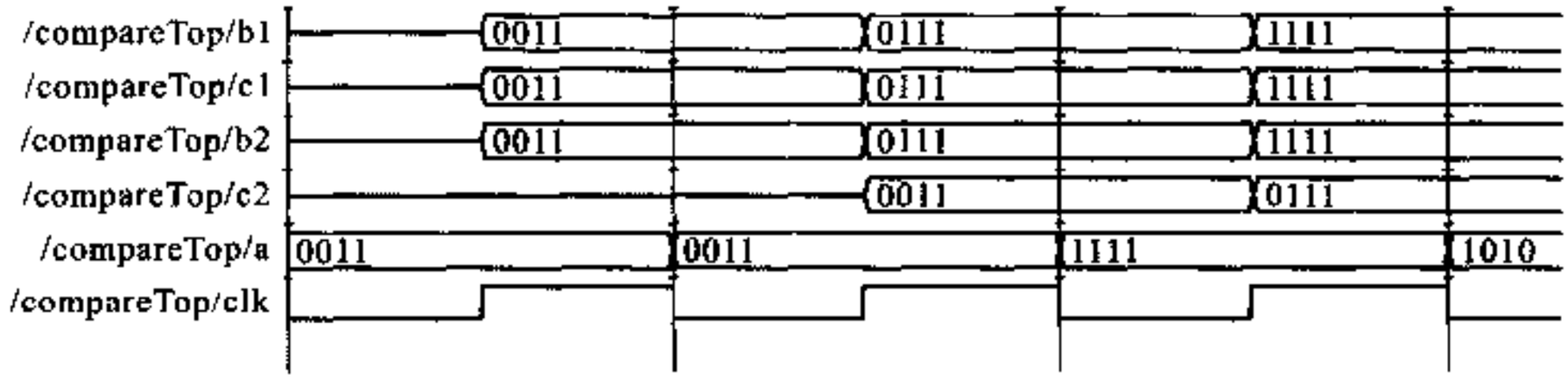


图 1.5 仿真波形

[**思考题**] 在 blocking 模块中按如下写法,仿真与综合的结果会有什么样的变化? 做出仿真波形,分析综合结果。

(1)

```
always @(posedge clk)
        begin
          c = b;
          b = a;
        end
```

(2)

```
always @(posedge clk) b = a;
always @(posedge clk) c = b;
```

练习五 用 always 块实现较复杂的组合逻辑电路

目的:(1) 掌握用 always 块实现较大组合逻辑电路的方法;

(2) 进一步了解 assign 与 always 两种组合电路实现方法的区别和注意点;

（3）学习测试模块中随机数的产生和应用；

（4）学习综合不同层次的仿真，并比较结果。

仅使用 assign 声明语句来实现组合逻辑电路，如果逻辑关系比较复杂，则不容易理解语句的功能。而适当地采用 always 块来设计组合逻辑，使源代码语句的功能容易理解。

下面是一个简单的指令译码电路的设计示例。该电路通过对指令的判断，对输入数据执行相应的操作，包括加、减、与、或和求反，并且无论是指令作用的数据还是指令本身发生变化，结果都要做出及时的反应。显然，这是一个较为复杂的组合逻辑电路，如果采用 assign 声明语句，表达起来非常复杂。示例中使用了电平敏感的 always 块，所谓电平敏感的触发条件是指在@ 后的括号内电平列表中的任何一个电平发生变化（与时序逻辑不同，它在@ 后的括号内没有沿敏感关键词，如 posedge 或 negedge），就能触发 always 块的动作，并且运用了 case 声明语句来进行分支判断，不但设计思想得到直观的体现，而且代码看起来非常整齐、便于理解。

```
//-------------------------文件名 alu.v ------------------------
`define plus 3'd0
`define minus 3'd1
`define band 3'd2
`define bor 3'd3
`define unegate 3'd4

module alu(out,opcode,a,b);
    output[7:0] out;
    reg[7:0] out;
    input[2:0] opcode;
    input[7:0] a,b;                          //操作数

    always@(opcode or a or b)                //电平敏感的 always 块
    begin
      case(opcode)
        `plus: out = a+b;                    //加操作
        `minus: out = a-b;                   //减操作
        `band: out = a&b;                    //求与
        `bor: out = a|b;                     //求或
        `unegate: out = ~a;                  //求反
        default: out =8'hx;                  //未收到指令时，输出任意态
    endcase
  end
endmodule
```

同一组合逻辑电路分别用 always 块和 assign 声明语句描述时，代码的形式大相径庭，但是在 always 块中适当运用 default（在 case 声明语句中）和 else（在 if-else 结构中），通常可以综合为纯组合逻辑，尽管被赋值的变量一定要定义为 reg 型。如果不使用 default 或 else 对缺省项进行说明，则易生成意想不到的锁存器，这一点一定要加以注意。

指令译码器的测试模块源代码如下：

```
//------------------------- alutest.v -------------------------
`timescale 1ns/1ns
`include "./alu.v"
module alutest;
    wire[7:0] out;
    reg[7:0] a,b;
    reg[2:0] opcode;
    parameter times=5;
    initial
    begin
      a=|$random|%256; //Give a radom number blongs to [0,255].
      b=|$random|%256; //Give a radom number blongs to [0,255].
      opcode=3'h0;
      repeat(times)
        begin
        #100  a=|$random|%256; //Give a radom number.
              b=|$random|%256; //Give a radom number.
              opcode=opcode+1;
        end
        #100 $stop;
    end
    alu alu1(out,opcode,a,b);
endmodule
```

仿真波形（部分）如图 1.6 所示。

[练习题] 运用 always 块设计一个八路数据选择器。要求，每路输入数据与输出数据均为 4 位二进制数，当选择开关（至少 3 位）或输入数据发生变化时，输出数据也相应地发生变化。

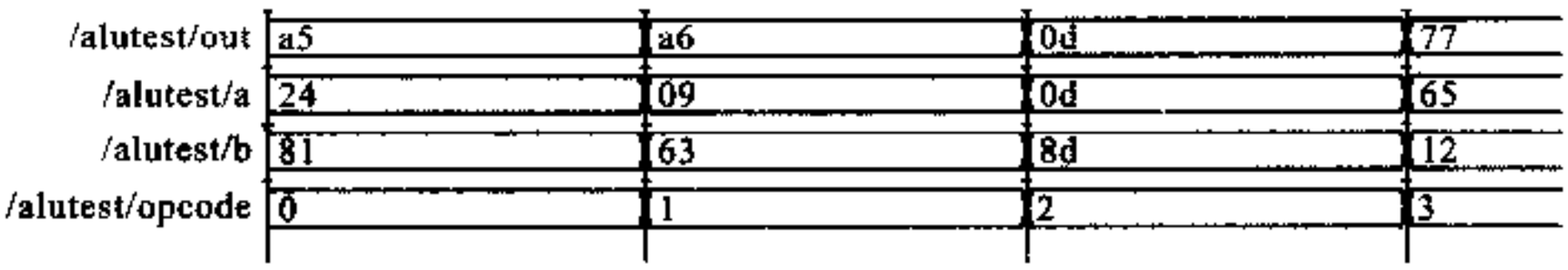


图 1.6　仿真波形

练习六　在 Verilog HDL 中使用函数

目的：(1) 了解函数的定义和在模块设计中的使用；

(2) 了解函数的可综合性问题；

(3) 了解许多综合器不能综合复杂的算术运算的情况。

与一般的程序设计语言一样，Verilog HDL 也可使用函数以适应对不同变量采取同一运算的操作。Verilog HDL 函数在综合时被理解成具有独立运算功能的电路，每调用一次函数相当于改变这部分电路的输入以得到相应的计算结果。

下面是函数调用的一个简单示范，采用同步时钟触发运算的执行，每个 clk 时钟周期都会执行一次运算。并且在测试模块中，通过调用系统任务$display 在时钟的下降沿显示每次计算的结果。

模块源代码如下：

```
//-----------------------------文件名 tryfunct.v -----------------------------
module tryfunct(clk,n,result,reset);
    output[31:0] result;
    input[3:0] n;
input reset,clk;
reg[31:0] result;
always @(posedge clk)                    //clk 的上沿触发同步运算
  begin
    if(! reset)                          //reset 为低时复位
      result <=0;
    else
      begin
        result <= n * factorial(n)/((n*2)+1);
      end                                //verilog 在整数除法运算结果中不考虑余数
```

```
end
function [31:0] factorial;          //函数定义,返回的是一个32位的数
  input [3:0] operand;              //输入只有一个4位的操作数
  reg [3:0] index;                  //函数内部计数用中间变量
  begin
    factorial = operand ? 1 : 0;    //先定义操作数为零时函数的输出为零,不为零
                                    //时为1
    for(index = 2; index <= operand; index = index + 1)
      factorial = index * factorial;   //表示阶乘的算术迭代运算
  end
endfunction
endmodule
```

测试模块源代码如下:

```
`include "./tryfunct.v"
`timescale 1ns/100ps
`define clk_cycle 50
module tryfuctTop;
    reg[3:0] n,i;
    reg reset,clk;
    wire[31:0] result;
    initial
      begin
        clk=0;
        n=0;
        reset=1;
      #100 reset=0;              //产生复位信号的负跳沿
      #100 reset=1;              //复位信号恢复高电平后才开始输入n
      for(i=0;i<=15;i=i+1)
        begin
          #200 n=i;
        end
      #100 $stop;
    end
    always #`clk_cycle clk=~clk;
```

```
    tryfunct m(.clk(clk),.n(n),.result(result),.reset(reset));
endmodule
```

上例中函数 factorial(n)实际上就是阶乘运算。必须提醒大家注意的是,许多综合器不能综合 tryfunct.v 模块。因此,我们在实际可综合电路结构的设计中,要尽量避免复杂的算术运算,把复杂的运算拆分成几个步骤,通过寄存器存储中间数据,在几个时钟周期完成。

仿真波形(部分)如图 1.7 所示。

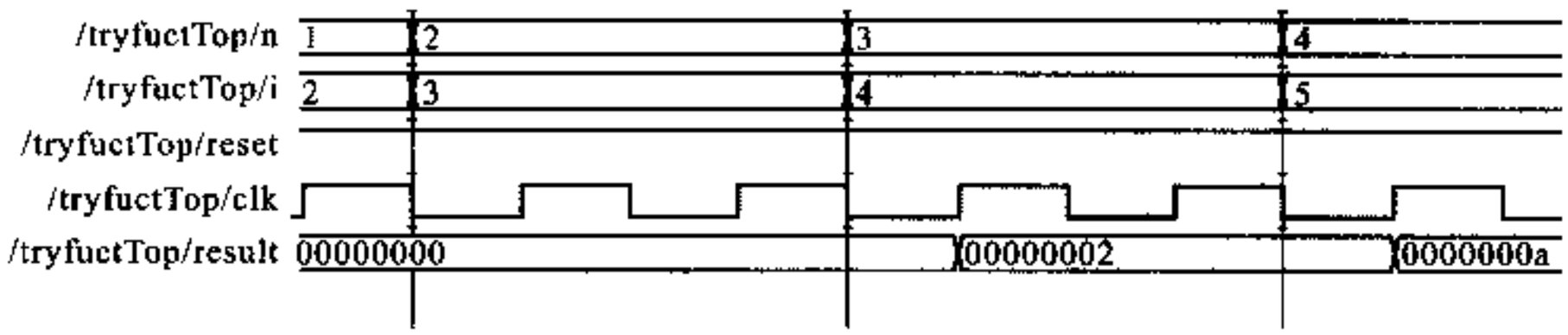


图 1.7　仿真波形

[练习题]　设计一个带控制端的逻辑运算电路,分别完成正整数的平方、立方和最大数为 5 的阶乘的运算,要求可综合。编写测试模块,并给出各种层次的仿真波形,比较它们的不同。

练习七　在 Verilog HDL 中使用任务 task 声明语句

目的:(1) 掌握任务在 Verilog 模块设计中的应用;

(2) 学会在电平敏感列表的 always 块中使用拼接操作、task 和阻塞赋值等声明语句生成复杂组合逻辑的高级方法。

仅有函数并不能完全满足 Verilog HDL 中的运算需求。当希望能够将一些信号进行运算并输出多个结果时,采用函数结构就显得非常不方便,而任务结构在这方面的优势则十分突出。任务本身并不返回计算值,但是它通过类似 C 语言中形参与实参的数据交换,非常容易地实现运算结果的调用。此外,还常常利用任务来包装模块设计中许多复杂的过程,将许多复杂的操作步骤用一个命名清晰易懂的任务隐藏起来,大大提高了程序的可读性。

下面介绍一个实例,巧妙地利用电平敏感的 always 块和一个比较两变量大小排序的任务,设计出 4 个(4 位)并行输入数的高速排序组合逻辑。可以看到,利用 task 声明语句非常方便地实现两个数据之间的交换排序,通过在电平敏感的 always 块中的多次调用,实现了四变量的高速排序。用函数无法实现相同的功能。另外,task 声明语句也避免了直接用一般

语句来描述所引起的不易理解和综合时产生冗余逻辑等问题。

模块源代码如下：

```
//----------------------------文件名 sort4.v ----------------------------
module sort4(ra,rb,rc,rd,a,b,c,d);
    output[3:0] ra,rb,rc,rd;
    input[3:0] a,b,c,d;
    reg[3:0] ra,rb,rc,rd;
    reg[3:0] va,vb,vc,vd;

    always @ (a or b or c or d)
      begin
        {va,vb,vc,vd} = {a,b,c,d};
        sort2(va,vc);              //va 与 vc 互换
        sort2(vb,vd);              //vb 与 vd 互换
        sort2(va,vb);              //va 与 vb 互换
        sort2(vc,vd);              //vc 与 vd 互换
        sort2(vb,vc);              //vb 与 vc 互换
        {ra,rb,rc,rd} = {va,vb,vc,vd};
      end

  task sort2;
    inout[3:0] x,y;
    reg[3:0] tmp;
    if(x>y)
      begin
        tmp = x;       //x 与 y 变量的内容互换,要求顺序执行,所以采用阻塞赋值方式
        x = y;
        y = tmp;
      end
  endtask

endmodule
```

值得注意的是,task 声明语句中的变量定义与模块中的变量定义不尽相同,它们并不受输入/输出类型的限制。如此例,x 与 y 对于 task sort2 来说虽然是 inout 型,但实际上它们对应的是 always 块中的变量,都是 reg 型变量。

测试模块源代码如下：

```
`timescale 1ns/100ps
`include "sort4.v"
```

```
module task_Top;
    reg[3:0] a,b,c,d;
    wire[3:0] ra,rb,rc,rd;
    initial
      begin
        a =0;b =0;c =0;d =0;
        repeat(50)
      begin
        #100 a  =|$random|% 15;
             b  =|$random|% 15;
             c  =|$random|% 15;
             d  =|$random|% 15;
      end
      #100 $stop;
      sort4 sort4 (.a(a),.b(b),.c(c),.d(d), .ra(ra),.rb(rb),.rc(rc),.rd
                  (rd));
 endmodule
```

仿真波形(部分)如图 1.8 所示。

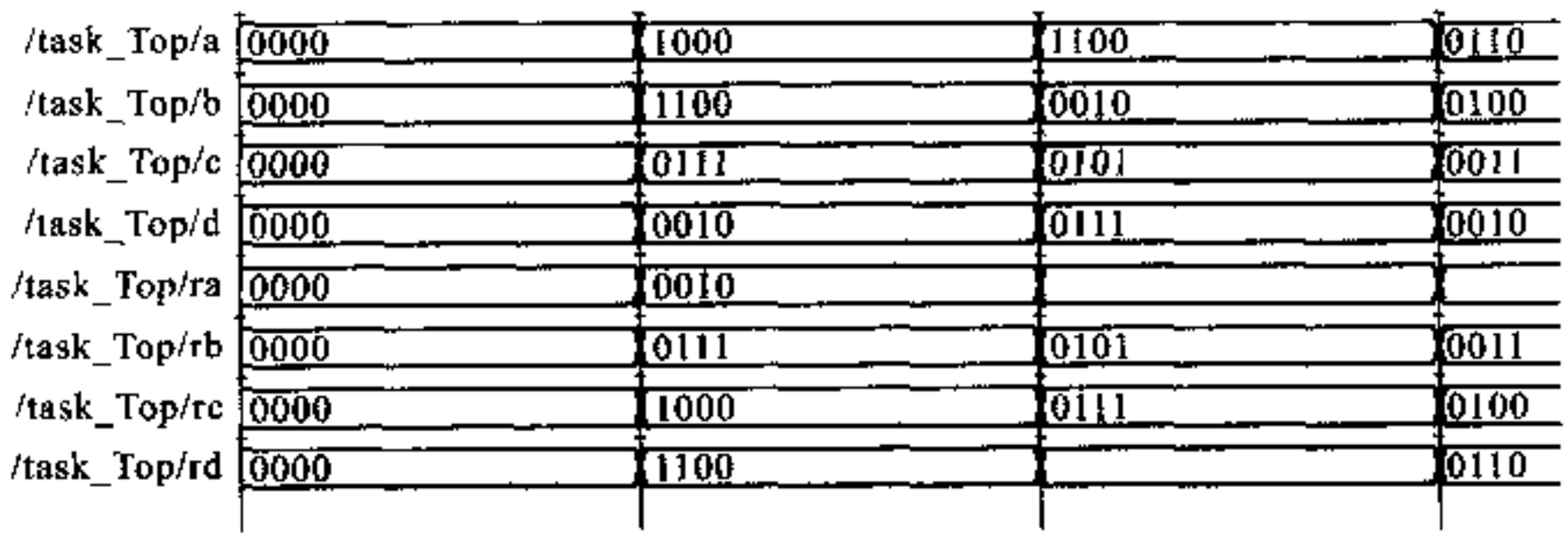


图 1.8　仿真波形

[**练习题**]　用两种不同的方法设计一个功能相同的模块,该模块能完成 4 个 8 位二进制输入数据的冒泡排序。第一种,模仿上面的例子用纯组合逻辑实现;第二种,假设 8 位数据是按照时钟节拍串行输入的,要求用时钟触发任务的执行,每个时钟周期完成一次数据交换的操作。比较两种不同方法的运行速度和消耗资源的不同。

练习八　利用有限状态机进行时序逻辑的设计

目的:(1) 掌握利用有限状态机实现一般时序逻辑分析的方法;

(2) 掌握用 Verilog 编写可综合有限状态机的标准模板;

(3) 掌握用 Verilog 编写状态机模块的测试文件的一般方法。

在数字电路中已经学习过通过建立有限状态机来进行数字逻辑的设计,而在 Verilog HDL 硬件描述语言中,这种设计方法得到进一步的发展。通过 Verilog HDL 提供的语句,可以直观地设计出适合更为复杂的时序逻辑的电路。关于有限状态机的设计方法在教材中已经作了较为详细的阐述,在此就不赘述了。

下例是一个简单的状态机设计,功能是检测一个 5 位二进制序列"10010"。考虑到序列重叠的可能,有限状态机共提供 8 个状态(包括初始状态 IDLE)。

模块源代码如下:

```
//-----------------------------文件名 seqdet.v -----------------------------
module seqdet(x,z,clk,rst,state);
    input x,clk,rst;
    output z;
    output[2:0] state;
    reg[2:0] state;
    wire z;

    parameter IDLE = 'd0, A = 'd1, B = 'd2,
                          C = 'd3, D = 'd4,
                          E = 'd5, F = 'd6,
                          G = 'd7;

    assign z = (state == E && x == 0 )? 1 : 0;
//当 x 序列 10010 最后一个 0 刚到时刻,时钟沿立刻将状态变为 E,此时 z 应该变为高 al-
 ways @(posedge clk)
if(! rst)
    begin
      state <= IDLE;
    end
else
    casex(state)
      IDLE : if(x == 1)          //第一个码位对,记状态 A
            begin
```

```
        state <= A;
      end
A: if(x==0)                    //第二个码位对,记状态 B
      begin
        state <= B;
      end
B: if(x==0)                    //第三个码位对,记状态 C
      begin
        state <= C;
      end
    else                       //第二个码位不对,前功尽弃,记状态为 F
      begin
        state <= F;
      end
C: if(x==1)                    //第四个码位对
      begin
        state <= D;
      end
    else                       //第四个码位不对,前功尽弃,记状态为 G
      begin
        state <= G;
      end
D: if(x==0)                    //第五个码位对,记状态 E
      begin
        state <= E;            //此时开始应有 z 的输出
      end
    else                       //第五个码位不对,前功尽弃,只有刚进入的 1 有用,
                               //回到第一个码位对状态,记状态 A
      begin
        state <= A;
      end
E: if(x==0)                    //连着前面已经输入的 x 序列 10010 考虑,又输入
                               //了 0 码位可以认为第三个码位已对,记状态 C
      begin
        state <= C;
      end
    else                       //前功尽弃,只有刚输入的 1 码位对,记状态为 A
```

```
            begin
              state <= A;
            end
        F: if(x ==1)                    //只有刚输入的 1 码位对,记状态为 A
            begin
              state <= A;
            end
          else                          //又有 1 码位对,记状态为 B
            begin
              state <= B;
            end
        G: if(x ==1)                    //只有刚输入的 1 码位对,记状态为 A
            begin
              state <= F;
            end
            default:state = IDLE;       //缺省状态为初始状态。
        endcase
    endmodule
```

测试模块源代码如下：

```
    //-------------------------文件名 seqdet.v -------------------------
    `timescale 1ns/1ns
    `include "./seqdet.v"
    module seqdet_Top;
        reg clk,rst;
        reg[23:0] data;
        wire[2:0] state;
        wire z,x;
        assign x = data[23];
        always #10 clk = ~clk;
        always @(posedge clk)
            data = {data[22:0],data[23]};       //形成数据向左移环行流,最高位与 x 连接
        initial
          begin
            clk =0;
            rst =1;
            #2 rst =0;
            #30 rst =1;
```

```
        data ='b1100_1001_0000_1001_0100;
        #500 $stop;
      end
      seqdet m(x,z,clk,rst,state);
endmodule
```

仿真波形如图 1.9 所示。

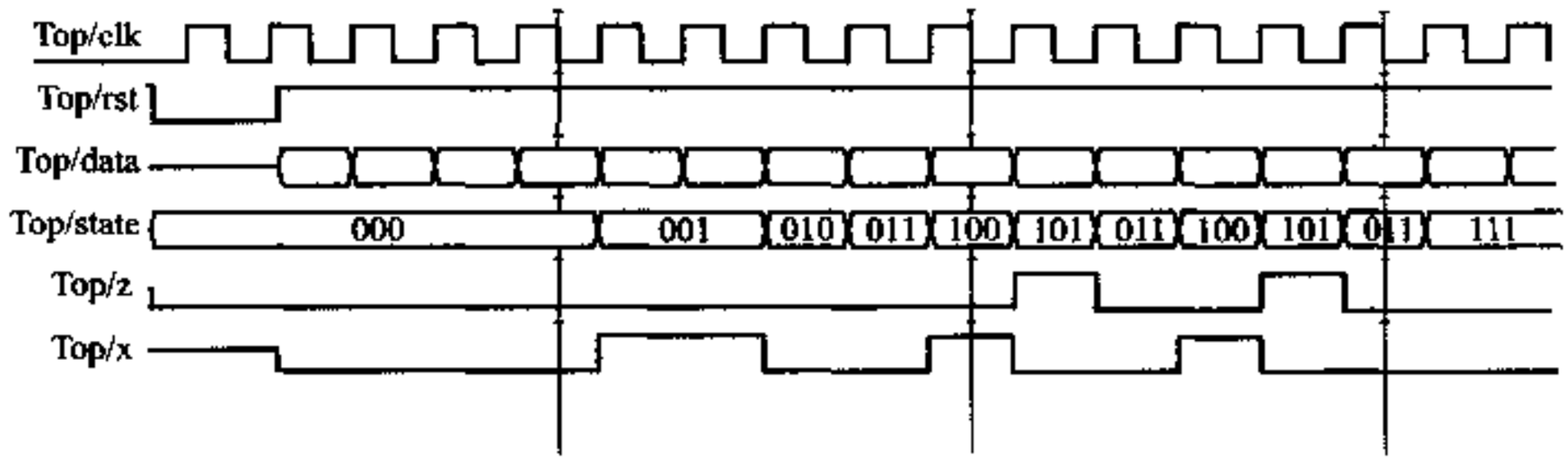


图 1.9　仿真波形

[**练习题**]　设计一个串行数据检测器。要求连续输入 4 个或 4 个以上的 1 时输出为 1，其他输入情况下为 0。编写测试模块对设计的模块进行各种层次的仿真，并观察波形，编写实验报告。

练习九　利用状态机实现比较复杂的接口设计

目的：(1) 学习运用由状态机控制的逻辑开关设计比较复杂的接口逻辑；

(2) 在复杂设计中使用 task 声明语句来提高程序的可读性；

(3) 加深对可综合风格模块的认识。

在练习八中学习了如何使用状态机的实例。实际上，单个有限状态机控制整个系统逻辑电路的运转在实际设计中是不多见。往往是状态机套用状态机，从而形成复杂的控制流。下面将提供一个这样的示例，供大家学习。

该例是一个并行数据转换为串行位流的变换器，利用双向总线输出。事实上，它是 EPROM 读写器设计中实现写功能的部分经删节得到的，为了帮助大家理解做了许多简化，去除了 EPROM 的启动、结束和 EPROM 控制字的写入等功能，只具备这样一个简单的并串转换功能。因而本设计无任何实用价值，只是为了教学目的。电路的工作步骤是：①把并行地址存入寄存器；②把并行数据存入寄存器；③连接串行单总线；④地址的串行输出；⑤数据

的串行输出;⑥挂起串行单总线;⑦给信号源应答;⑧让信号源给出下一个操作对象;⑨结束写操作。通过基本时钟,使得并行数据一位一位地输出。

模块源代码如下:

```
module writing(reset,clk,address,data,sda,ack);
    input reset,clk;
    input[7:0] data,address;
    output sda,ack;              //sda 负责串行数据输出,ack 是一个对象操作完毕后,模
                                 //块给出的应答信号
    reg link_write;              //link_write 决定何时输出
    reg[3:0] state;              //主状态机的状态字
    reg[4:0] sh8out_state;       //从状态机的状态字
    reg[7:0] sh8out_buf;         //输入数据缓冲
    reg finish_F;                //用以判断是否处理完一个操作对象
    reg ack;
    parameter
      idle=0,addr_write=1,data_write=2,stop_ack=3;
    parameter
      bit0=1,bit1=2,bit2=3,bit3=4,bit4=5,bit5=6,bit6=7,bit7=8;
    assign sda = link_write? sh8out_buf[7] : 1'bz;
    always @(posedge clk)
      begin
        if(! reset)                          //复位
        begin
          link_write<= 0;                    //挂起串行单总线
          state <= idle;
          finish_F <= 0;                     //结束标志清零
          sh8out_state<=idle;
            ack<= 0;
          sh8out_buf<=0;
        end
      else
      case(state)
        idle:
          begin
            link_write <= 0;                 //断开串行单总线
```

```
        finish_F <= 0;
        sh8out_state <= idle;
        ack <= 0;
        sh8out_buf <= address;                    //并行地址存入寄存器
        state <= addr_write;                      //进入下一个状态
      end
    addr_write:                                   //地址的输入
      begin
        if(finish_F ==0)
        begin shift8_out; end                     //地址的串行输出
        else
        begin
          sh8out_state <= idle;
          sh8out_buf <= data;                     //并行数据存入寄存器
          state <= data_write;
          finish_F <= 0;
        end
      end
    data_write:                                   //数据的写入
      begin
        if(finish_F ==0)
        begin shift8_out; end                     //数据的串行输出
        else
        begin
          link_write <= 0;
            state <= stop_ack;
            finish_F <= 0;
            ack <= 1;                             //向信号源发出应答
        end
      end
    stop_ack;                                     //向信号源发出应答结束
      begin
        ack <= 0;
        state <= idle;
      end
    endcase
```

```
    end
    task shift8_out;                          //地址和数据的串行输出
      begin
        case(sh8out_state)
        idle:
          begin
            link_write <= 1;                  //连接串行单总线,立即输出地址或
                                              //数据的最高位(MSB)
            sh8out_state <= bit7;
          end
    bit7:
      begin
        link_write <= 1;                      //连接串行单总线
        sh8out_state <= bit6;
        sh8out_buf <= sh8out_buf <<1;         //输出地址或数据的次高位(6 位)
      end
    bit6:
      begin
        sh8out_state <= bit5;
        sh8out_buf <= sh8out_buf <<1;
      end
    bit5:
      begin
        sh8out_state <= bit4;
        sh8out_buf <= sh8out_buf <<1;
      end
    bit4:
      begin
        sh8out_state <= bit3;
        sh8out_buf <= sh8out_buf <<1;
      end
    bit3:
      begin
        sh8out_state <= bit2;
        sh8out_buf <= sh8out_buf <<1;
```

```
        end
      bit2:
        begin
          sh8out_state <= bit1;
          sh8out_buf <= sh8out_buf << 1;
        end
      bit1:
        begin
          sh8out_state <= bit0;
          sh8out_buf <= sh8out_buf << 1;          //输出地址或数据的最低位(LSB)
        end
      bit0:
        begin
          link_write <= 0;                        //挂起串行单总线
          finish_F <= 1;                          //建立结束标志
        end
      endcase
    end
  endtask
endmodule
```

测试模块源代码如下：

```
`timescale 1ns/100ps
`define clk_cycle 50
module writingTop;
  reg reset,clk;
  reg[7:0] data,address;
  wire ack,sda;
always #`clk_cycle clk = ~clk;
initial
  begin
    clk=0;
    reset=1;
    data=0;
    address=0;
    #(2*`clk_cycle) reset=0;
```

```
    #(2 * `clk_cycle) reset = 1;
    #(100 * `clk_cycle) $stop;
  end
always @(posedge ack)          //接收到应答信号后,给出下一个处理对象
  begin
    data = data + 1;
    address = address + 1;
  end
writing writing( .reset(reset),.clk(clk),.data(data),
                 .address(address),.ack(ack),.sda(sda));
endmodule
```

仿真波形如图 1.10 所示。

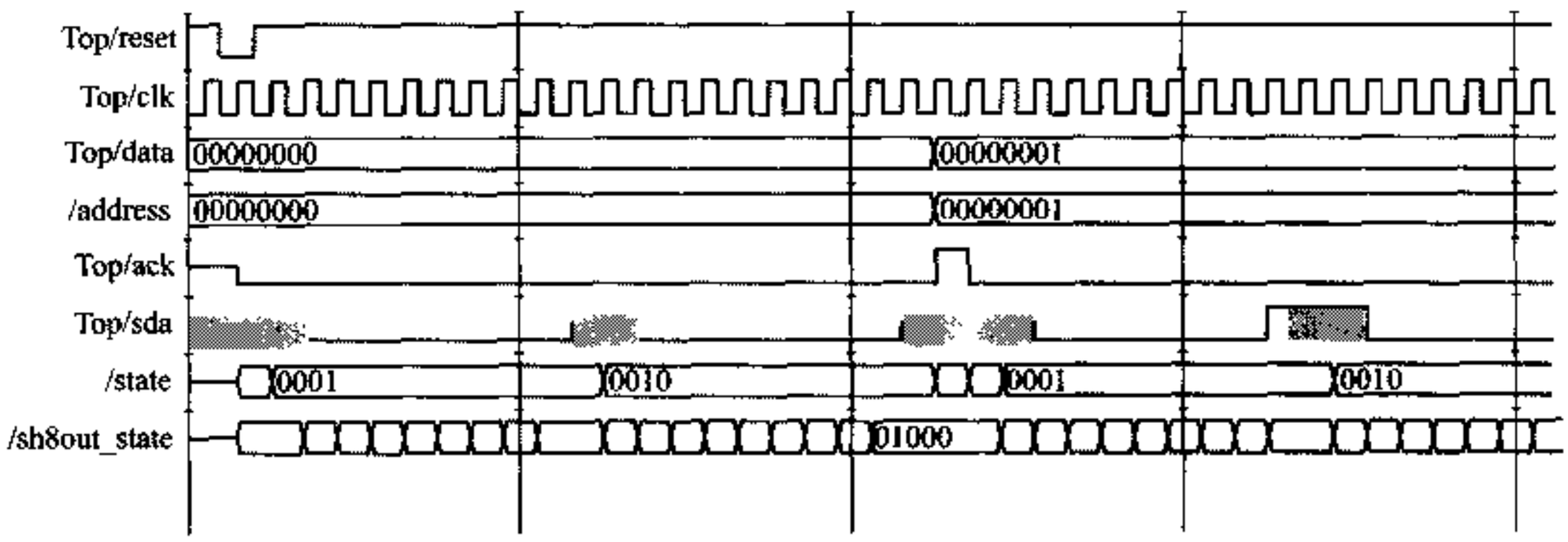


图 1.10　仿真波形

[练习题]　1. 彻底搞清楚上例,参考中级篇第八讲的实际例子,独立编写一个能实现 EEPROM 全部读写功能的并行转换为 I2C 串行总线读写信号模块。编写完整的符合工程要求的测试模块,进行各种层次的仿真,并观察波形。

2. 参考中级篇第七讲中的实际例子,编写可综合模块能把符合 I2C 串行总线要求的地址和数据信号,在只有 sda 和 scl 两个信号的前提下转换为并行的数据和地址信号。

练习十　通过模块实例调用实现大型的设计

目的:(1) 学习和掌握状态机嵌套和模块实例的连接方法;

（2）了解大型设计的层次化、结构化解决办法的技术基础；

（3）学习数据总线在模块设计中的应用和控制，掌握复杂接口模块设计的基本技术；

（4）学习和掌握用工程概念来编写较完整的测试模块，做到接近真实的完整测试。

现代硬件系统的设计过程与软件系统的开发相似，设计一个大规模的集成电路往往由模块多层次引用和组合构成。层次化、结构化的设计过程，能使复杂的系统容易控制和调试。在 Verilog 模块设计中，上层模块引用下层模块与 C 语言中程序调用有些类似，被引用的子模块在综合时作为其父模块的一部分被综合，形成相应的电路结构。在进行模块实例引用时，必须注意的是模块之间对应的端口，即子模块的端口与父模块的内部信号必须明确无误地一一对应，否则容易产生意想不到的后果。

下面的例子是根据笔者在工程设计中遇到的具体问题的一部分总结而成的。笔者对它进行了许多简化，以适用于教学的目的，在例子中突出读者在接口设计方面容易犯的错误。原来这部分模块的功能是将并行数据转化为串行数据送交外部电路编码，然后将编码后得到的串行数据转化为并行数据再交由 CPU 处理。为了简化起见，笔者省去了编码部分以及与计算机的接口。只留下 CPU 数据总线作为并行数据的出入通道。这实际上是两个独立的逻辑功能模块，一个是并行数据流转换为串行位流，另一个是将该串行位流又转换为并行数据，分别设计为两个独立的模块，然后再合并为一个模块，共享同一条并行数据总线和时钟。我们假设从并行数据总线上输入到模块的数据其到达的时间有一定的随机性，测试模块如何来表达这些问题，如何设计出能保证可靠接收和发送的接口是需要有经验的。本例简要地说明了这个问题，一定会对读者设计技术的提高有很大的帮助。

```
// ----------------------------文件名 P_S.v ----------------------------
/ ***************************************************************************************
 *** 模块功能:把在 nGet_AD_data 负跳变沿时刻后能维持约 3 个时钟周期的并行字节数据 ***
 ***              取入模块,在时钟节拍下转换为字节的位流,并产生相应字节位流的有效信号    ***
 ***************************************************************************************/
`define YES 1
`define NO 0
module P_S(Dbit_out,link_S_out,data,nGet_AD_data,clk);
    input clk;                          //主时钟节拍
    input nGet_AD_data;                 //负电平有效的取并行数据控制信号线
    input[7:0] data;                    //并行输入的数据端口
    output Dbit_out;                    //串行位流的输出
    output link_S_out;                  //允许串行位流输出的控制信号
```

```
reg [3:0] state;                    //状态变量寄存器
reg[7:0] data_buf;                  //并行数据缓存器
reg link_S_out;                     //串行位流输出的控制信号寄存器
reg d_buf;                          //位缓存器
reg finish_flag;                    //字节处理结束标志
assign Dbit_out = (link_S_out)? d_buf:0;              //给出串行数据
  always @(posedge clk or negedge nGet_AD_data)
                //nGet_AD_data 下降沿置数,寄存器清零,clk 上跳沿送出位流
  if(! nGet_AD_data)
    begin
      finish_flag <=0;
      state <= 9;
      link_S_out <=`NO;
      d_buf <= 0;
      data_buf <=0;
    end
  else
  case(state)
   9: begin
         data_buf <= data;
         state <=10;
         link_S_out <=`NO;
      end
   10: begin
         data_buf <= data;
         state <=0;
         link_S_out <=`NO;
       end
  0: begin
         link_S_out <=`YES;
         d_buf <=data_buf[7];
         state <=1;
     end
  1: begin
         d_buf <=data_buf[6];
         state <=2;
```

```
    end
2: begin
        d_buf <=data_buf[5];
        state <=3;
    end
3: begin
        d_buf <=data_buf[4];
        state <=4;
    end
4: begin
        d_buf <=data_buf[3];
        state <=5;
    end
5: begin
        d_buf <=data_buf[2];
        state <=6;
    end
6: begin
        d_buf <=data_buf[1];
        state <=7;
    end
7: begin
        d_buf <=data_buf[0];
        state <=8;
    end
8: begin
                link_S_out <=`NO;
                state <= 4'b1111;       //do nothing state
                finish_flag <=1;
            end
 default: begin
                link_S_out <=`NO;
                state <= 4'b1111;       //do nothing state
            end
    endcase
endmodule
```

```
// ----------------------------文件名 S_P.v ----------------------------
/ ***************************************************************************************
 ***                模块功能:把在位流有效信号控制下的字节位流读入模块,在时钟节拍      ***
 ***                          控制下转换为并行的字节数据,输出到并行数据口。          ***
 *************************************************************************************** /
`timescale 1ns/1ns
`define YES 1
`define NO 0
module S_P(data, Dbit_in, Dbit_ena, clk);
    output [7:0] data;              //并行数据输出口
    input Dbit_in, clk;             //字节位流输入口
    input Dbit_ena;                 //字节位流输入使能
    reg [7:0] data_buf;
    reg [3:0] state;                //状态变量寄存器
    reg p_out_link;                 //并行输出控制寄存器
    assign data = (p_out_link==`YES) ? data_buf : 8'bz;
    always@(negedge clk)
      if(Dbit_ena )
        case(state)
        0: begin
              p_out_link <=`NO;
              data_buf[7] <= Dbit_in;
              state <=1;
           end
        1: begin
              data_buf[6] <= Dbit_in;
              state <=2;
           end
        2: begin
              data_buf[5] <= Dbit_in;
           end  state <=3;
        3: begin
              data_buf[4] <= Dbit_in;
              state <=4;
           end
        4: begin
```

```
            data_buf[3] <= Dbit_in;
            state <=5;
        end
    5: begin
            data_buf[2] <= Dbit_in;
            state <=6;
        end
    6: begin
            data_buf[1] <= Dbit_in;
            state <=7;
        end
    7: begin
            data_buf[0] <= Dbit_in;
            state <=8;
        end
    8: begin
            p_out_link <= `YES;
            state <= 4'b1111;
        end
    default: state <=0;
            endcase
    else begin
            p_out_link <= `YES;
            state <=0;
        end
endmodule

// ----------------------------文件名 sys.v ----------------------------
/ ************************************************************************************
***   模块功能：把两个独立的逻辑模块(P_S 和 S_P)合并到一个可综合的模块中，          ***
***             共用一条并行总线，配合有关信号，分时进行输入/输出。                  ***
***   模块目的：学习如何把两个单向输入/输出的实例模块，连接在一起，共享一条总线。  ***
***             本模块是完全可综合模块，已经通过综合和布线后仿真。                  ***
************************************************************************************ /
`include "./P_S.v"
`include "./S_P.v"
module sys(databus,use_p_in_bus,Dbit_out,Dbit_ena,nGet_AD_data,clk);
```

```
input nGet_AD_data;                    //取并行数据的控制信号
input use_p_in_bus;                    //并行总线用于输入数据的控制信号
input clk;                             //主时钟
inout [7:0] databus;                   //双向并行数据总线
output Dbit_out;                       //字节位流输出
output Dbit_ena;                       //字节位流输出使能

wire clk;
wire nGet_AD_data;
wire Dbit_out;
wire Dbit_ena;
wire [7:0] data;

assign databus = (! use_p_in_bus)? data : 8'bzzzz_zzzz;

P_S m0(.Dbit_out(Dbit_out),.link_S_out(Dbit_ena),.data(databus),
       .nGet_AD_data(nGet_AD_data),.clk(clk));
S_P m1(.data(data), .Dbit_in(Dbit_out), .Dbit_ena(Dbit_ena),.clk
(clk));

endmodule

// ----------------------------文件名:Top.v ----------------------------
/ ***********************************************************************************************
***  模块功能:对合并在一起的可综合的模块 sys 进行测试验证。其测试信号尽可能地  ***
***           与实际情况一致,用随机数系统任务对数据的到来和时钟沿的抖动都进  ***
***           行了模拟仿真。本模块无任何工程价值,只有学习价值。              ***
*********************************************************************************************** /
`timescale 1ns/1ns
`include "./sys.v"          //改用不同级别的 Verilog 网表文件可进行不同层次的仿真
module Top;
    reg clk;
    reg[7:0] data_buf;
    reg nGet_AD_data;
    reg D_Pin_ena;          //并行数据输入 sys 模块的使能信号寄存器
    wire [7:0] data;
    wire clk2;
    wire Dbit_ena;

    assign data = (D_Pin_ena)? data_buf : 8'bz;
```

```
initial
  begin
    clk = 0;
    nGet_AD_data =1;   //置取数据控制信号初始值为高电平
    data_buf = 8'b1001_1001;   //假设的数据缓存器的初始值,可用于模拟并行数据
                               //的变化
    D_Pin_ena = 0;
end
initial
  begin
    repeat(100)
    begin
      #(100*14 + |$random| % 23) nGet_AD_data = 0;   //取并行数据开始
      #(112 + |$random| % 12) nGet_AD_data = 1;      //保持一定时间低电平
                                                     //后恢复高电平
      #(|$random| % 50) D_Pin_ena = 1; //并行数据输入 sys 模块的使能信号有效
      #(100*3 + |$random| % 5) D_Pin_ena = 0; //保持三个时钟周期后让出总线
      #333 data_buf = data_buf +1; //假设的数据变化,可为下次取的不同的数据
      #(100*11 + |$random| % 1000);
    end
end
always #(50 + $random% 2) clk = ~clk;     //主时钟的产生
sys ms( .databus(data),
        .use_p_in_bus(D_Pin_ena),
        .Dbit_out(Dbit_out),
        .Dbit_ena(Dbit_ena),
        .nGet_AD_data(nGet_AD_data),
        .clk(clk)); endmodule
```

布线后仿真波形如图 1.11 所示。

[练习题]　模仿示范题,编写一个通过申请 CPU 中断取得数据进行处理,并把处理的结果通过同一条数据总线返回 CPU 的模块。要求具体时序参数与 CPU 中断的响应时间和读写时序完全一致。要考虑尽量减少资源消耗,并提高处理速度。

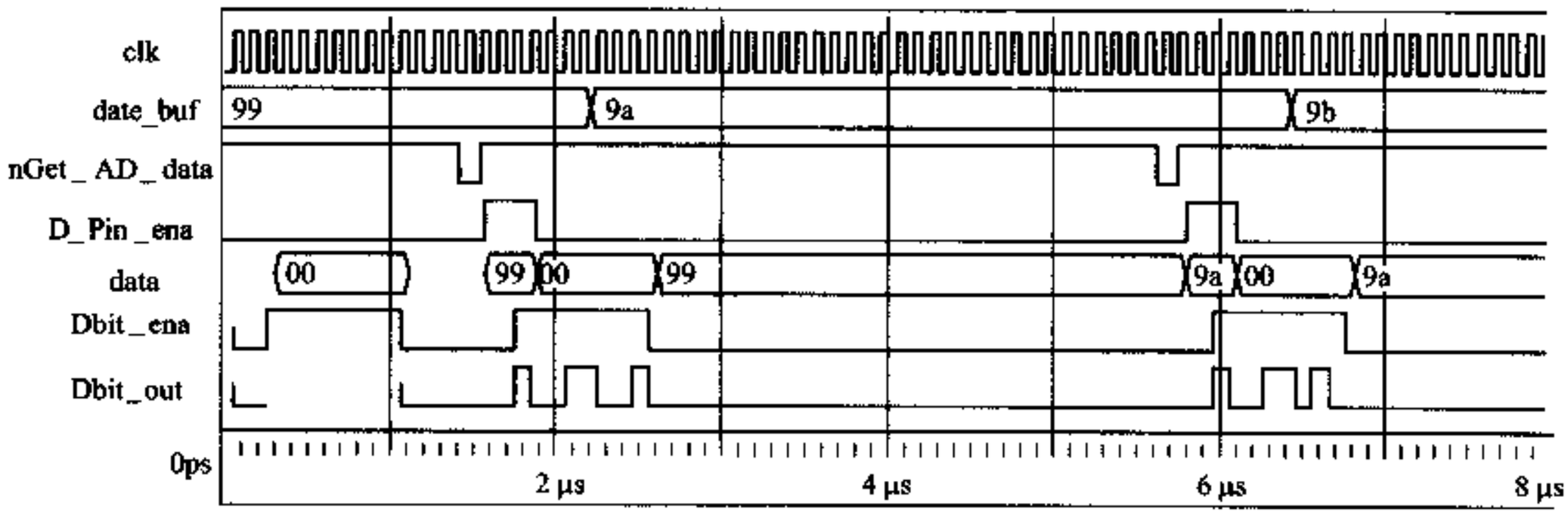


图 1.11　布线后的仿真波形

练习十一　简单卷积器的设计

目的:(1) 学习和掌握高速度计算逻辑的状态机控制基本方法;

(2) 了解计算逻辑与存储器和 A/D 模块的接口设计技术基础;

(3) 进一部学习掌握数据总线在模块设计中的应用和控制;

(4) 熟悉用工程概念来编写较完整的测试模块,做到接近真实的完整测试。

下面将共同来完成有一个实际接口器件背景的小型设计——简单卷积器,这个设计是根据真实工程设计简化而来的,专门用于教学的目的。希望通过这个设计,使读者建立起专用数字计算系统设计的基本概念。设计分成许多步骤进行,具体过程排列如下:

1. 明确设计任务

在设计之前必须明确设计的具体内容。卷积器是数字信号处理系统中常用的部件。它对模拟输入信号实时采样,得到数字信号序列。然后对数字信号进行卷积运算,再将卷积结果存入 RAM 中。对模拟信号的采样由 A/D 转换器来完成,而卷积过程由卷积器来实现。为了设计卷积器,首先要设计 RAM 和 A/D 转换器的 Verilog HDL 模型。在电子工业发达的国家,可以通过商业渠道得到非常准确的外围器件的虚拟模型。如果没有外围器件的虚拟模型,就需要仔细地阅读和分析 RAM 和 A/D 转换器的器件说明书来自行编写。因为 RAM 和 A/D 转换器不是我们设计的硬件对象,所以需要的只是它们的行为模型,精确的行为模型需要认真细致地编写,并不比可综合模块容易编写。它们与实际器件的吻合程度直接影响设计的成功。在这里把重点放在卷积器的设计上,直接给出 RAM 和 A/D 转换器的 Veril-

og HDL 模型和它们的器件参数(见附录),读者可以对照器件手册,认真阅读 RAM 和 A/D 转换器的 Verilog HDL 模型。对 RAM 和 A/D 转换器的 Verilog HDL 模型的详细了解对卷积器的设计是十分必要的。

到目前为止,对设计模块要完成的功能比较明确了。总结如下:首先它要控制 A/D 转换器进行 A/D 转换,从 A/D 转换器得到转换后的数字序列,然后对数字序列进行卷积,最后将结果存入 RAM。下面让我们一起来设计它.

2. 卷积器的设计

通过前面的练习已经知道,用高层次的设计方法来设计复杂的时序逻辑,重点是把时序逻辑抽象为有限状态机,并用可综合风格的 Verilog HDL 把这样的状态机描述出来。下面将通过注释来介绍整个程序的设计过程。我们选择 8 位输入总线,输出到 RAM 的数据总线也选择 8 位,卷积值为 16 位,分高、低字节分别写到两个 RAM 中。地址总线为 11 位。为了理解卷积器设计中的状态机,必须对 A/D 转换器和 RAM 的行为模块有深入的理解。

```
`timescale 100ps/100ps
module conl(address,indata,outdata,wr,nconvst,nbusy,enout1,enout2,CLK,
reset,start);
    input CLK,              //采用 10 MHz 的时钟
    reset,                  //复位信号
    start,                 //因为 RAM 的空间是有限的,当 RAM 存满后采样和卷积都会停止
                            //此时给一个 start 的高电平脉冲将会开始下一次的卷积
    nbusy;                  //从 A/D 转换器来的信号表示转换器的忙或闲
    output wr,              //RAM 写控制信号
    enout1,enout2,          //enout1 是存储卷积低字节结果 RAM 的片选信号
                            //enout2 是存储卷积高字节结果 RAM 的片选信号
    nconvst,               //给 A/D 转换器的控制信号,命令转换器开始工作,低电平有效
    address;                //地址输出
    input [7:0] indata;    //从 A/D 转换器来的数据总线
    output[7:0] outdata;   //写到 RAM 去的数据总线
    wire nbusy;
    reg wr;
    reg nconvst,
        enout1,
        enout2;
    reg[7:0] outdata;
```

```
reg[10:0] address;
reg[8:0] state;
reg[15:0] result;
reg[23:0] line;
reg[11:0] counter;
reg high;
reg[4:0] j;
reg EOC;

parameter h1 = 1,h2 = 2,h3 = 3;    //假设的系统系数
parameter IDLE = 9'b000000001, START = 9'b000000010, NCONVST = 9'b000000100,
          READ = 9'b000001000, CALCU = 9'b000010000, WRREADY = 9'b000100000,
          WR = 9'b001000000, WREND = 9'b010000000, WAITFOR = 9'b100000000;

parameter FMAX = 20;    //因为 A/D 转换的时间是随机的,为保证按一定的频率采样,
                        //A/D 转换控制信号应以一定频率给出。这里采样频率通过
                        //FMAX 控制为 500 kHz。

always @(posedge CLK)
  if(! reset)
    begin
        state <= IDLE;
        nconvst <= 1'b1;
        enout1 <= 1;
        enout2 <= 1;
        counter <= 12'b0;
        high <= 0;
        wr <= 1;
        line <= 24'b0;
        address <= 11'b0;
    end
  else
        case(state)
            IDLE:if(start == 1)
                begin
                    counter <= 0;//counter 是一个计数器,记录已用的 RAM 空间
                    line <= 24'b0;
                    state <= START;
                end
```

```
else
        state <= IDLE;
//START 状态控制 A/D 开始转换
START: if(EOC)
    begin
        nconvst <=0;
        high <=0;
        state <= NCONVST;
    end
else
        state <= START;
//NCONVST 状态是 A/D 转换保持阶段
        NCONVST: begin
        nconvst <=1;
        state <= READ;
    end
//READ 状态读取 A/D 转换结果,计算卷积结果
READ: begin
    if(EOC)
      begin
        line <={line[15:0],indata};
        state <= CALCU;
      end
  else
        state <= READ;
end
CALCU: begin
        result <= line[7:0] * h1 + line[15:8] * h2 + line[23:16] * h3;
        state <= WRREADY;
      end
//将卷积结果写入 RAM 时,先写入低字节,再写入高字节
//WRREADY 状态是写 RAM 准备状态,建立地址和数据信号
WRREADY:begin
                address <= counter;
                if(! high) outdata <= result[7:0];
                else outdata <= result[15:8];
```

```
                    state <= WR;
                end
        //WR 状态产生片选和写脉冲
        WR: begin
            if(! high) enout1 <=0;
                else enout2 <=0;
                wr <=0;
                state <= WREND;
            end
        //WREND 状态结束一次写操作,若还未写入高字节则转到 WRREADY 状
        //态开始高字节写入
        WREND:begin
                wr <=1;
                enout1 <=1;
                enout2 <=1;
                if(! high)
                    begin
                    high <=1;
                    state <= WRREADY;
                  end
                else state <= WAITFOR;
              end
        //WAITFOR 状态控制采样频率并判断 RAM 是否已被写满
        WAITFOR: begin
            if(j == FMAX -1)
              begin
                counter <= counter +1;
                if(! counter[11]) state <= START;
                else
                  begin
                      state <= IDLE;
                      $display($time,"The ram is used up.");
                      $stop;
                  end
            end
        else state <= WAITFOR;
            end
```

```
            default:state <= IDLE;
        endcase
//assign rd = 1; //RAM 的读信号始终保持为高
//j 记录时钟,与 FMAX 共同控制采样频率
//由于直接用 CLK 的上升沿对 nbusy 判断以
//决定某些操作是否运行时,会因为两个信号
//的跳变沿相隔太近而令状态机不能正常工作。因此
//利用 CLK 的下降沿建立 EOC 信号与 nbusy 同步,相位
//相差 180°,然后用 CLK 的上升沿判断操作是否进行
always @(negedge CLK )
  begin
       EOC <= nbusy;
       it(! reset||state == START)
       j <= 1;
       else
       j <= j + 1;
  end
endmodule
```

3. 前仿真及后仿真

程序写完后首先要做前仿真,我们可用仿真器(如 ModelSim SE/EE PLUS 5.4)来做。为检查我们写的程序,需要编写测试程序,测试程序应尽可能检测出各种极限情况。这里给出一个测试程序供参考。

```
//-----------------------------文件名 testcon1.v ---------------------------
`timescale 100ps/100ps
module testcon1;
    wire wr,
        enin,
        enout1,
        enout2;
    wire[10:0] address;
    reg rd,
        CLK,
        reset,
        start;
```

```
wire nbusy;
wire nconvst;
wire[7:0] indata;
wire[7:0] outdata;
integer i;
parameter HALF_PERIOD = 1000;
//产生 10 kHz 的时钟
initial
  begin
      rd = 1;
      i = 0;
      CLK = 1;
      forever #HALF_PERIOD CLK = ~CLK;
  end
//产生置位信号
initial
  begin
      reset = 1;
      #(HALF_PERIOD * 2 + 50) reset = 0;
      #(HALF_PERIOD * 3) reset = 1;
  end
//产生开始卷积控制信号
initial
  begin
      start = 0;
      #(HALF_PERIOD * 7 + 20) start = 1;
      #(HALF_PERIOD * 2) start = 0;
      #(HALF_PERIOD * 1000) start = 1;
      #(HALF_PERIOD * 2) start = 0;
  end
assign enin = 1;
conl con( .address(address),.indata(indata),.outdata(outdata),.wr
          (wr),
          .nconvst(nconvst),.nbusy(nbusy),.enout1(enout1),
          .enout2(enout2),.CLK(CLK),.reset(reset),.start(start));
sram ramlow( .Address(address),.Data(outdata),.SRW(wr),.SRG(rd),.SRE
```

```
(enout1));
    adc adc(.nconvst(nconvst),.nbusy(nbusy),.data(indata));
endmodule
```

因测试程序已经包括了各模块,只需编译测试程序并运行它。通过仿真器中的菜单(如ModelSim 仿真器中功能列表中 view 的下拉菜单选择 structure、signal 和 wave),可以根据需要看到各种信号的波形,由此检测程序。

图 1.12 是一个参考波形图,由它我们可以看清整个程序的时序。

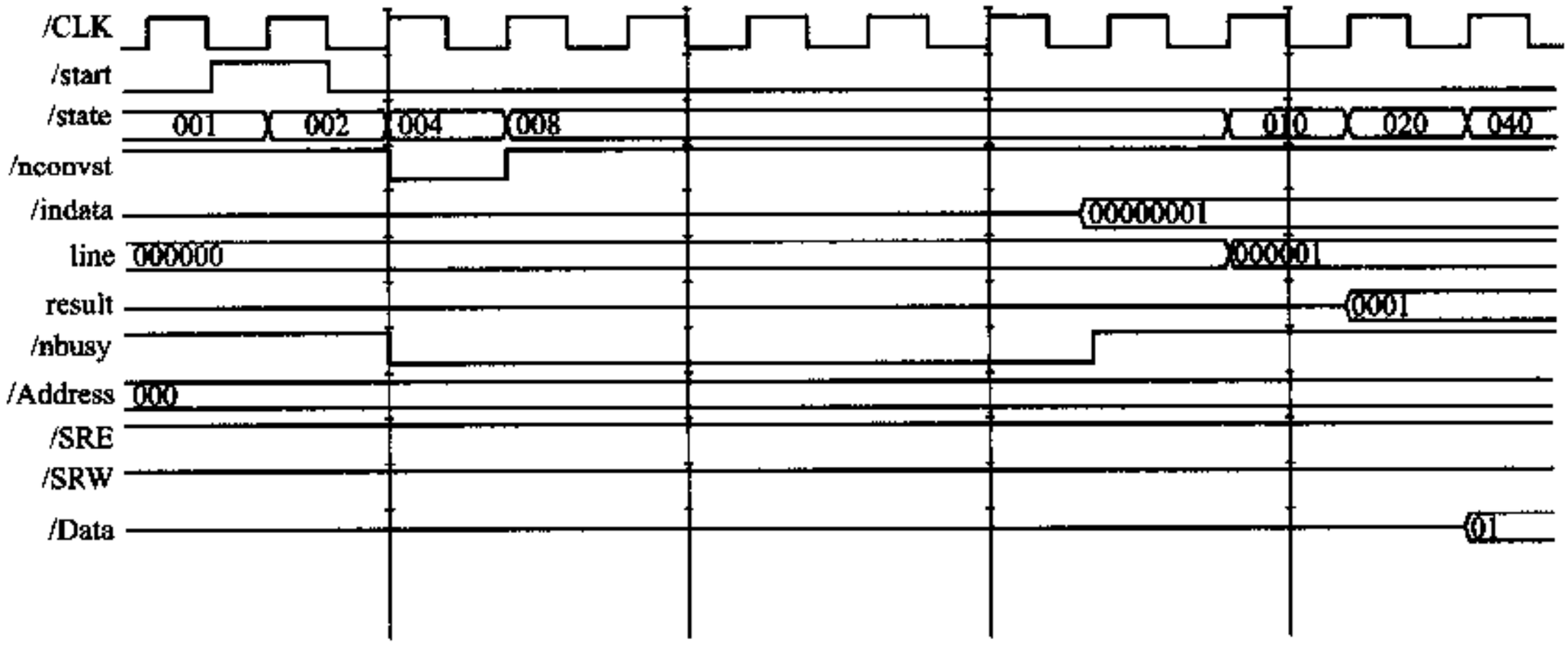


图 1.12　参考波形图

如果前仿真通过了,则可以做后仿真了。后仿真考虑了器件的延迟,更具可靠性。首先用综合器(如 Synplify)进行综合。在综合时应注意选择器件库,可选择如 Altera FLEX10K 系列 FPGA 或其他类型的 FPGA。综合完后生成了与原程序名相应的一个扩展名为 edf 的文件。然后用布线工具(如 MAX + PLUS II ver. 9.3)对刚才得到的扩展名为 edf 的文件进行编译。如果编译不出错就可得到扩展名为 vo 的两个文件,一个文件名与原文件名相同,另一文件名为 alt_max2. vo。

现在就可用仿真器(如 ModelSim)来做后仿真。步骤与前仿真一样,对于 Altera 系列的 FPGA 只需将 con1. vo 和 alt_max2. vo 两个文件重新编译,取代原先用 con1. v 编译的模型就可以了,不同的是 FPGA 具体方法有些不同,但原理都是一样的。这时将后仿真波形与前仿真波形比较就会发现后仿真把器件的延迟考虑进去了。看波形,检查结果是否正确。若不正确则改动原程序,重新进行上述步骤。

4. 卷积器的改进

我们希望设计出快速高效的卷积器。而通过对上面设计的卷积器仿真波形的分析不难发现,有很多时间被浪费在等待 A/D 转换上。同时因 A/D 转换,计算卷积和写入 RAM 是串行工作的,效率很低。为提高效率我们可以采用 3 片 A/D 转换器同时工作,并将采样过程和计算,写入 RAM 的控制改为并行工作。以下就是改进后的程序。原采样频率为 500 kHz,改进后采样频率为 2.22 MHz,为原采样频率的 4 倍多。

```
//------------------------------ con3ad.v ------------------------------
`timescale 1ns/100ps
module
 con3ad( indata, outdata, address, CLK, reset, start, nconvst1, nconvst2,
         nconvst3,
         nbusy1,nbusy2,nbusy3,wr,enout1,enout2);
input indata,
      CLK,
      reset,
      start,
      nbusy1,
      nbusy2,
      nbusy3;
output outdata,
       address,
       nconvst1,        //采用3根控制线控制3片A/D转换器
       nconvst2,
       nconvst3,
       wr,
       enout1,
         enout2;
wire[7:0] indata;
wire CLK,
     reset,
     start,
     nbusy1,
     nbusy2,
     nbusy3;
reg[7:0] outdata;
reg[10:0] address;
reg nconvst1,
    nconvst2,
    nconvst3,
```

```
        wr,
        enout1,
        enout2;
    reg[6:0] state;
    reg[5:0] i;
    reg[1:0] j;
    reg[11:0] counter;
    reg[23:0] line;
    reg[15:0] result;
    reg high;
    reg k;
    reg EOC1,EOC2,EOC3;

    parameter h1 =1,h2 =2,h3 =3;
    parameter IDLE = 7'b0000001,   READ_PRE = 7'b0000010,
              READ = 7'b0000100,      CALCU = 7'b0001000,
           WRREADY = 7'b0010000,         WR = 7'b0100000,
             WREND = 7'b1000000;

    always @(posedge CLK)
      begin
          if(! reset)
            begin
                state <= IDLE;
                counter <=12'b0;
                wr <=1;
                enout1 <=1;
                enout2 <=1;
                outdata <=8'bz;
                address <=11'bz;
                line <=24'b0;
                result <=16'b0;
                high <=0;
            end      //end of "if"
          else
            begin
                case(state)
                    IDLE:if(start)
                        begin
                            counter <=0;
```

```
                state <= READ_PRE;
            end
             else    state <= IDLE;
READ_PRE: if(EOC1||EOC2||EOC3) //由于频率相对改进前的卷积
                                //器大大提高,所以加入
                                //READ_PRE 状态对取数操作
                                //予以缓冲
                state <= READ;
            else
                state <= READ_PRE;
READ:begin
        high <=0;
        enout2 <=1;
        wr <=1;
        if(j ==1)
          begin
          if(EOC1)
            begin
                line <={line[15:0],indata};
                state <= CALCU;
            end
          else state <= READ_PRE;
        end
      else if(j ==2&&counter!=0)
        begin
            if(EOC2)
              begin
                line <={line[15:0],indata};
                state <= CALCU;
            end
         else state <= READ_PRE;
      end
    else if(j ==3&&counter!=0)
      begin
          if(EOC3)
            begin
```

```
                    line <= {line[15:0],indata};
                    state <= CALCU;
                end
            else state <= READ_PRE;
        end
    else state <= READ;
end
CALCU:begin
        result <= line[7:0] * h1 + line[15:8] * h2 + line
                  [23:16] * h;
        state <= WRREADY;
    end
WRREADY:begin
        wr <= 1;
        address <= counter;
        if(k == 1)state <= WR;
        else      state <= WRREADY;
        end
WR: begin
    if(! high)     enout1 <= 0;
    else           enout2 <= 0;
    wr <= 0;
    if(! high)     outdata <= result[7:0];
    else           outdata <= result[15:8];
    if(k == 1)     state <= WREND;
    else           state <= WR;
end
WREND:begin
        wr <= 1;
        enout1 <= 1;
        enout2 <= 1;
        if(k == 1)
          if(! high)
            begin
                high <= 1;
                state <= WRREADY;
```

```
                end
              else
                begin
                    counter <= counter + 1;
                    if(counter[11]&&counter[0])
                        state <= IDLE;
                    else state <= READ_PRE;
                end
            else state <= WREND;
        end
    default:state <= IDLE;
    endcase //end of the case
  end //end of "else"
end //end of "always"
  //计数器 i 用来记录时间
  always @(posedge CLK)
    begin
      if(! reset) i <=0;
      else
        begin
          if(i ==44) i <=0;
          else       i <= i + 1;
        end
    end
  //j 是控制信号,协调卷积器轮流从 3 片 A/D 转换器上读取数据
  always @(posedge CLK)
    begin
      if(i ==4) j <=2;
      else if(i ==10) j <=0;
      else if(i ==19) j <=3;
      else if(i ==25) j <=0;
      else if(i ==34) j <=1;
      else if(i ==40) j <=0;
    end
  //k 是计数器,用以控制写操作信号
  always @(posedge CLK)
```

```
  begin
      if(state == WRREADY||state == WR||state == WREND)
        if(k == 1) k <= 0;
        else       k <= 1;
      else k <= 0;
  end
//根据计数器 i 控制 3 片 A/D 转换信号 NCONVST1,NCONVST2,NCONVST3
always @(posedge CLK)
  begin
  if(! reset) nconvst1 <= 1;
      else if(i == 0) nconvst1 <= 0;
      else it(i == 3) nconvst1 <= 1;
  end
always @(posedge CLK)
  begin
        if(! reset) nconvst2 <= 1;
        else if(i == 15) nconvst2 <= 0;
        else if(i == 18) nconvst2 <= 1;
      end
    always @(posedge CLK)
      begin
        if(! reset) nconvst3 <= 1;
        else if(i == 30) nconvst3 <= 0;
        else if(i == 33) nconvst3 <= 1;
      end
    always @(negedge CLK)
      begin
        EOC1 <= nbusy1;
        EOC2 <= nbusy2;
        EOC3 <= nbusy3;
      end
endmodule
```

测试程序如下所示：

```
`timescale 1ns/100ps
module testcon3ad;
```

```
wire wr,
     enin,
     enout1,
     enout2;
wire[10:0] address;
reg clk,
    reset,
    start;
    rd;
wire nbusy1,
     nbusy2,
     nbusy3;
wire nconvst1,
     nconvst2,
     nconvst3;
wire[7:0] indata;
wire[7:0] outdata;
parameter HALF_PERIOD = 15;        //时钟周期为 30ns
initial
  begin
      clk = 1;
      forever #HALF_PERIOD clk = ~clk;
  end
initial
  begin
      reset = 1;
      #110 reset = 0;
      #140 reset = 1;
  end
initial
  begin
      start = 0;
      rd = 1;
      #420 start = 1;
      #120 start = 0;
      #107600 start = 1;
```

```
        #150 start =0;
    end
assign enin =1;
con3ad con3ad( .indata(indata),.outdata(outdata),.address(address),
               .CLK(clk),.reset(reset),.start(start),
               .nconvst1(nconvst1),.nconvst2(nconvst2),.nconvst3(nconvst3),
               .nbusy1(nbusy1),.nbusy2(nbusy2),.nbusy3(nbusy3),
               .wr(wr),.enout1(enout1),.enout2(enout2));
sram ramlow( .Address(address),.Data(outdata),.SRW(wr),.SRG(rd),.SRE(enout1));
adc ad_1( .nconvst(nconvst1),.nbusy(nbusy1),. data(indata));
adc ad_2( .nconvst(nconvst2),.nbusy(nbusy2),. data(indata));
adc ad_3( .nconvst(nconvst3),.nbusy(nbusy3),. data(indata));
endmodule
```

与前面一样,给出仿真波形的片段供读者参考,如图 1.13 所示。

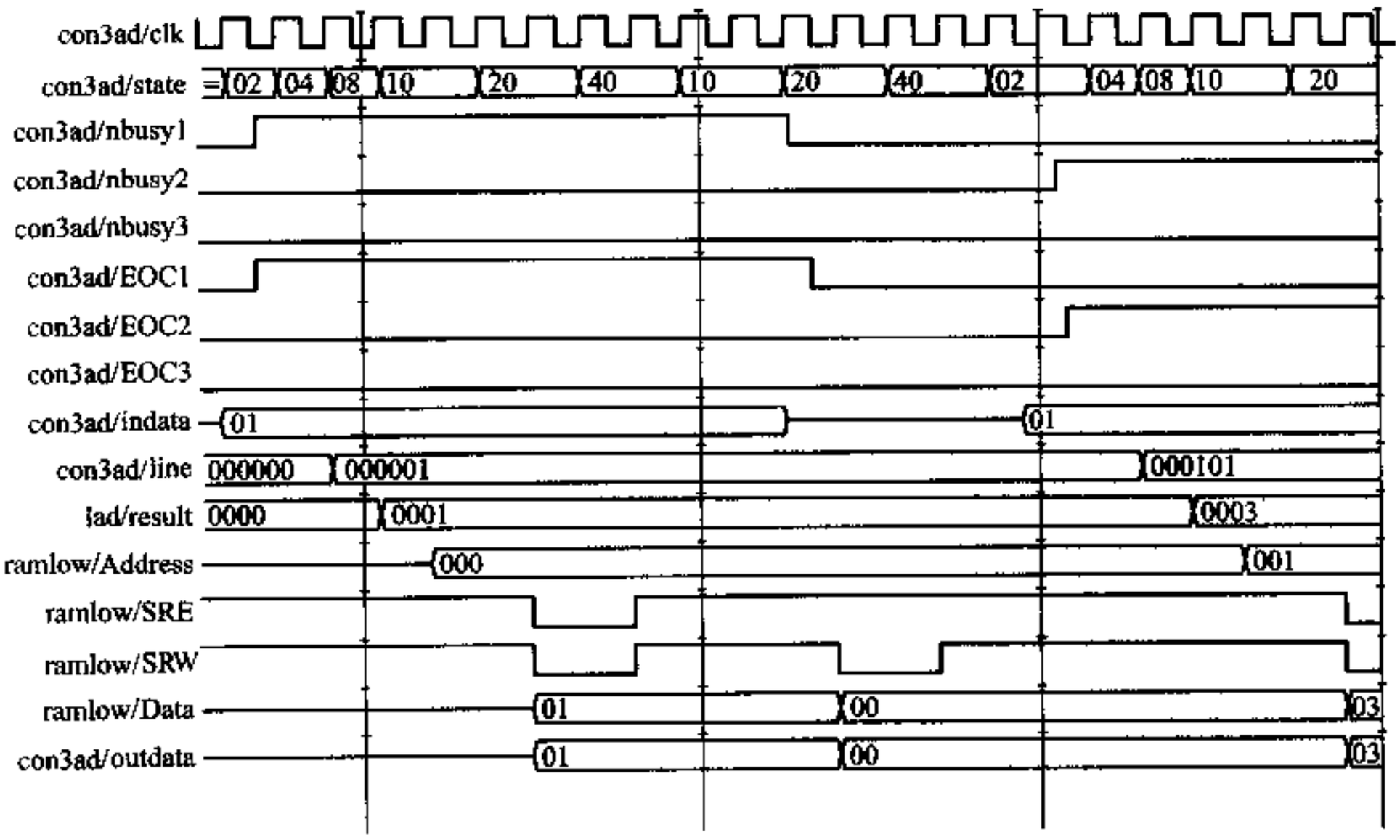


图 1.13　仿真波形片段

经过前仿真确定设计在逻辑上成立后,我们同样可以对这个改进后的卷积器使用合适的器件库进行后仿真,步骤与前面讲述的完全一样,在这里就不再赘述了。

练习十二　利用 SRAM 设计一个 FIFO

目的:(1) 学习和掌握存取队列管理的状态机设计的基本方法;

(2) 了解并掌握用存储器构成 FIFO 接口设计的基本技术;

(3) 用工程概念来编写完整的测试模块,达到完整测试覆盖。

在本练习中,要求读者利用练习十一中提供的 SRAM 模型,设计 SRAM 读写控制逻辑,使 SRAM 的行为对用户表现为一个 FIFO(先进先出存储器)。

1. 设计要求

本练习要求读者设计的 FIFO 为同步 FIFO,即对 FIFO 的读/写使用同一个时钟。该 FIFO 应当提供用户读使能(fiford)和写使能(fifowr)输入控制信号,并输出指示 FIFO 状态的非空(nempty)和非满(nfull)信号,FIFO 的输入、输出数据使用各自的数据总线:in_data 和 out_data。图 1.14 为 FIFO 接口示意。

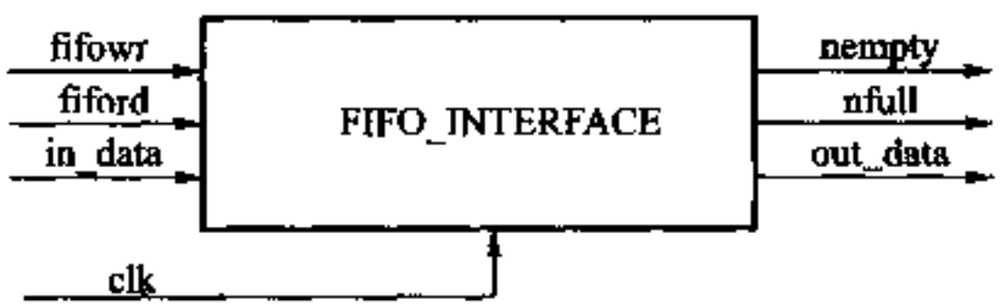


图 1.14　FIFO 接口示意

注意　这些不是设计的具体内容,而是为检验设计正确与否所提供的验证环境。此处如描述一下 SRAM 与 FIFO 的差异,并由此得到 FIFO 接口设计的关键在于 SRAM 地址产生这一结论会好一些。

2. FIFO 接口的设计思路

FIFO 的数据读写操作与 SRAM 的数据读写操作基本上相同,只是 FIFO 没有地址。所以用 SRAM 实现 FIFO 的关键点是如何产生正确的 SRAM 地址。

我们可以借用软件的方法,将 FIFO 抽象为环形数组,并用读指针(fifo_rp)和写指针(fifo_wp)两个指针,控制对该环形数组的读写。其中,读指针 fifo_rp 指向下一次读操作所要读取的单元,并且每完成一次读操作,fifo_rp 加 1;写指针 fifo_wp 则指向下一次写操作时存放数据的单元,并且每完成一次写操作,fifo_wp 加 1。由 fifo_rp 和 fifo_wp 的定义易知,当 FIFO 被读空或写满后,fifo_rp 和 fifo_wp 将指向同一单元,但在读空和写满之前 FIFO 的状态是不

同的，所以如果能区分这两种状态，再通过比较 fifo_rp 和 fifo_wp 就可以得到 nempty 和 nfull 信号了。图 1.15 为 FIFO 工作状态的示意。

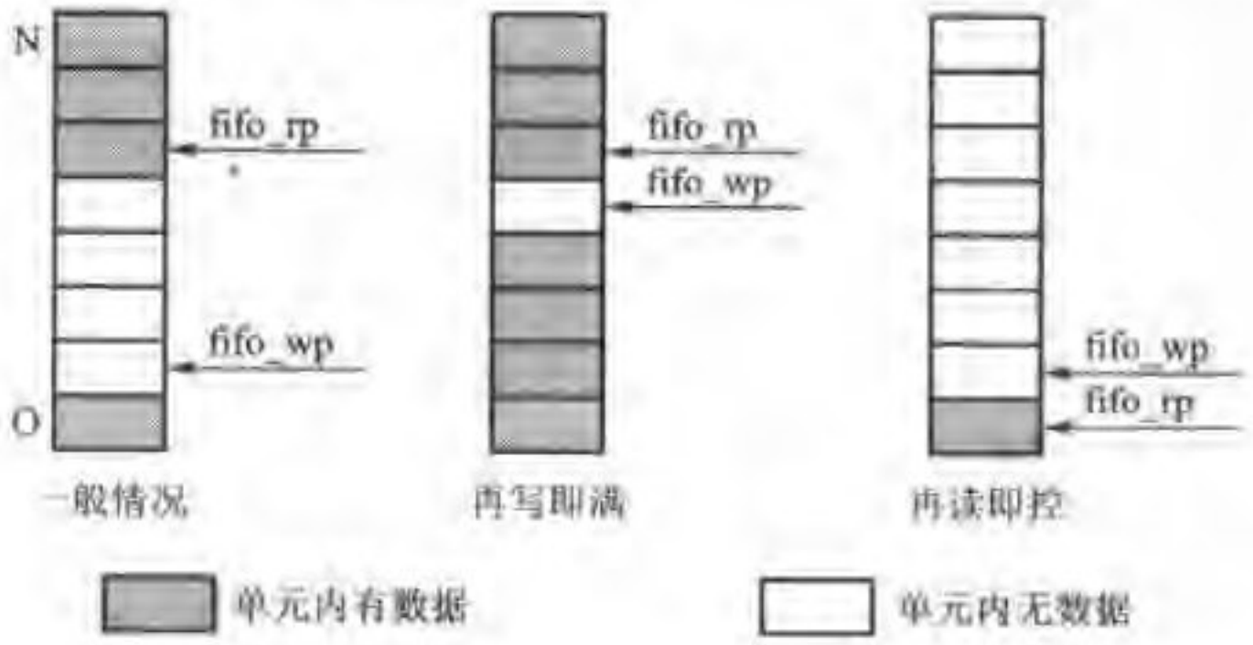


图 1.15　FIFO 工作状态示意

在得到 nfull 和 nempty 信号后，就需要考虑如何应用这两个信号来控制对 FIFO 的读写，使得 FIFO 在被写满后不能再写入，从而防止覆盖原有数据，并且在被读空后也不能再进行读操作，防止读取无效数据。

此外，在进行 SRAM 读写操作时，应该注意建立地址、数据和控制信号的先后顺序。一般情况下，希望对 SRAM 读写的波形时序如图 1.16 所示：

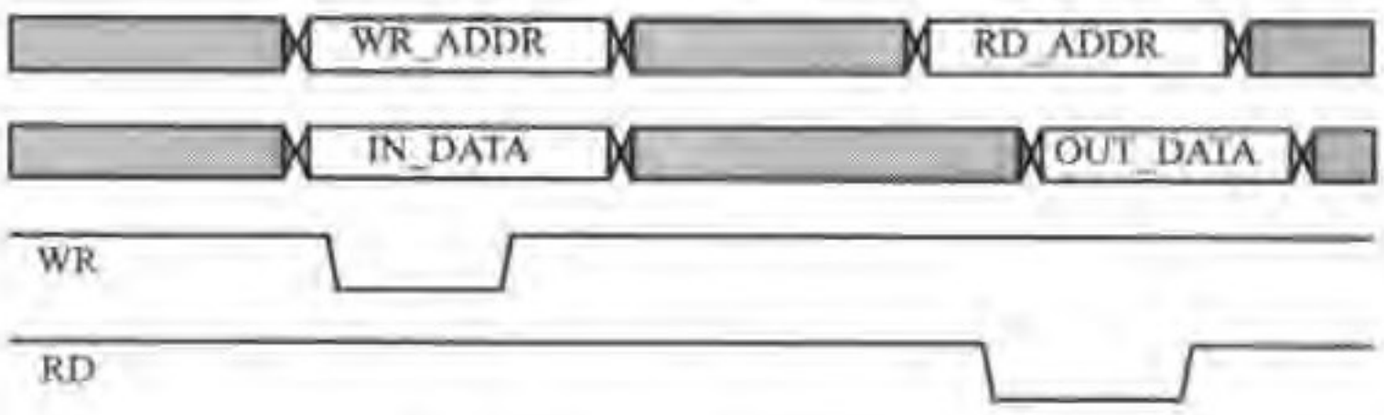


图 1.16　对 SRAM 读写的波形时序

即在写 SRAM 时，先建立地址和数据，然后置写使能信号 WR 有效，在 WR 保持有效一定时间后，先复位 WR，然后释放地址和数据总线。而在读取 SRAM 时，则先建立地址，然后置读使能 RD 有效，在 RD 维持有效一定时间后，复位 RD，同时读取数据总线上的值，然后再释放地址总线。在进行 FIFO 操作时，用户一般希望除了没有地址外，其他 3 个信号的时序关系能保持不变。请读者在设计 FIFO 控制信号与 SRAM 控制信号间逻辑关系时注意这一点。

3. FIFO 接口的测试

在完成一个设计后，需要进行测试以确认设计的正确性和完整性。而要进行测试，就需要编写测试激励和结果检查程序，即测试平台(testbench)。在某些情况下，如果设计的接口能够预先确定，测试平台的编写也可以在设计完成之前就进行，这样做的好处是在设计测试平台的同时也在更进一步深入了解设计要求，有助于理清设计思路，及时发现设计方案的错误。

编写测试激励时，除了注意对实际可能存在的各种情况的覆盖外，还要有意针对非正常情况下的操作进行测试。在本练习中，就应当进行在 FIFO 读空后继续读取、FIFO 写满后继续写入、FIFO 复位后马上读取等操作的测试。

测试激励中通常会有一些复杂操作需要反复进行，如本练习中对 FIFO 的读写操作。这时可以将这些复杂操作纳入到几个 task 声明语句中，既减小了激励编写的工作量，也使得程序的可读性更好。

下面的测试程序给读者作为参考，希望读者能先用这段程序测试所设计的 FIFO 接口，然后编写自己更全面的测试程序。

```
    //-----------------------------文件名:t.v ----------------------------
`define FIFO_SIZE 8
`include "sram.v"     //有的仿真工具不需要加这句，只要 sram.v 模块编译过就可以了
`timescale 1ns/1ns

module t;

    reg [7:0]   in_data;          //FIFO 数据总线
    reg         fiford,fifowr;    //FIFO 控制信号

    wire[7:0]   out_data;
    wire        nfull, nempty;    //FIFO 状态信号

    reg         clk,rst;

    wire[7:0] sram_data;          //SRAM 数据总线
    wire[10:0] address;           //SRAM 的地址总线
    wire rd,wr;                   //SRAM 读写控制信号

    reg [7:0] data_buf[`FIFO_SIZE:0];        //数据缓存,用于结果检查
    integer index;                           //用于读写 data_buf 的指针

    //系统时钟
    initial clk =0;
    always #25 clk = ~clk;

    //测试激励序列
```

```
initial
  begin
    fiford=1;
    fifowr=1;
    rst=1;
    #40 rst=0;
    #42 rst=1;
if (nempty) $display($time,"Error: FIFO be empty, nempty should be low.\n");
//----------------------------连续写 FIFO ----------------------------
index = 0;
repeat(`FIFO_SIZE)
begin
    data_buf[index]=$random;
    write_fifo(data_buf[index]);
    index = index + 1;
end
if (nfull) $display($time,"Error: FIFO full, nfull should be low.\n");
repeat(2) write_fifo($random);
#200
//----------------------------连续读 FIFO ----------------------------
index=0;
read_fifo_compare(data_buf[index]);
if ( ~nfull) $display($time,"Error: FIFO not full, nfull should be
                          high.\n");
repeat(`FIFO_SIZE-1)
  begin
      index = index + 1;
      read_fifo_compare(data_buf[index]);
  end
if (nempty) $display($time,"Error: FIFO be empty, nempty should be low.\n");
repeat(2) read_fifo_compare(8'bx);
reset_fifo;
//----------------------------写后读 FIFO ----------------------------
repeat(`FIFO_SIZE*2)
```

```
begin
  data_buf[0] = $random;
  write_fifo(data_buf[0]);
  read_fifo_compare(data_buf[0]);
end
//----------------------------异常操作----------------------------
reset_fifo;
read_fifo_compare(8'bx);
write_fifo(data_buf[0]);
read_fifo_compare(data_buf[0]);
  $stop;
end
  fifo_interface fifo_mk(.in_data(in_data),
                         .out_data(out_data),
                         .fiford(fiford),
                         .fifowr(fifowr),
                         .nfull(nfull),
                         .nempty(nempty),
                         .address(address),
                         .sram_data(sram_data),
                         .rd(rd),.wr(wr),
                         .clk(clk),.rst(rst) );
  sram m1 (.Address (address),
           .Data (sram_data),
           .SRG (rd),          //SRAM 读使能
           .SRE (1'b0),        //SRAM 片选,低有效
           .SRW (wr) );        //SRAM 写使能
task write_fifo;
  input [7:0] data;
  begin
      in_data = data;
      #50 fifowr = 0;         //往 SRAM 中写数
      #200 fifowr = 1;
      #50;
  end
```

```
endtask

task read_fifo_compare;
input [7:0] data;
  begin
      #50 fiford = 0;          //从 SRAM 中读数
      #200 fiford = 1;
        if (out_data != data)
          $display($time,"Error: Data retrieved (% h) not match the one
                    stored (% h). \n",out_data, data);

      #50;
        end
      endtask

      task reset_fifo;
        begin
            #40 rst = 0;
            #40 rst = 1;
        end
      endtask

endmodule
```

4. FIFO 接口的参考设计

FIFO 接口的实现有多种方案,下面给出的参考设计只是其中一种。希望读者在完成自己的设计后,和参考设计做一下比较。

```
`define SRAM_SIZE 8    //为减小对 FIFO 控制器的测试工作量,置 SRAM 空间为 8 字节
`timescale 1ns/1ns

module fifo_interface (
    in_data,        //对用户的输入数据总线
    out_data,       //对用户的输出数据总线
    fiford,         //FIFO 读控制信号,低电平有效
    fifowr,         //FIFO 写控制信号,低电平有效
    nfull,
    nempty,
    address,        //到 SRAM 的地址总线
    sram_data,      //到 SRAM 的双向数据总线
    rd,             //SRAM 读使能,低电平有效
```

```
wr,                    //SRAM 写使能,低电平有效
clk,                   //系统时钟信号
rst );                 //全局复位信号,低电平有效

//来自用户的控制输入信号
input fiford, fifowr, clk, rst;

//来自用户的数据信号
input[7:0] in_data;
output[7:0] out_data;
reg[7:0] in_data_buf,          //输入数据缓冲区
out_data_buf;                  //输出数据缓冲区

//输出到用户的状态指示信号
output nfull, nempty;
reg nfull, nempty;

//输出到 SRAM 的控制信号
output rd, wr;

//到 SRAM 的双向数据总线
inout[7:0] sram_data;

//输出到 SRAM 的地址总线
output[10:0] address;
reg[10:0] address;

//Internal Register
reg[10:0] fifo_wp,             //FIFO 写指针
       fifo_rp;                //FIFO 读指针

reg[10:0] fifo_wp_next,        //fifo_wp 的下一个值
       fifo_rp_next;           //fifo_rp 的下 一个值

reg near_full, near_empty;

reg[3:0] state;                //SRAM 操作状态机寄存器

parameter idle = 'b0000,
    read_ready = 'b0100,
    read = 'b0101,
    read_over = 'b0111,
    write_ready = 'b1000,
    write = 'b1001,
    write_over = 'b1011;
```

```
//SRAM 操作状态机
always @(posedge clk or negedge rst)
  if ( ~rst)
    state <= idle;
  else
    case(state)
      idle:                                  //等待对 FIFO 的操作控制信号
          if (fifowr ==0 && nfull)           //用户发出写 FIFO 申请,且 FIFO 未满
            state <=write_ready;
          else if(fiford ==0 && nempty)      //用户发出读 FIFO 申请,且 FIFO 未空
            state <= read_ready;
          else                               //没用对 FIFO 操作的申请
            state <= idle;
      read_ready:                            //建立 SRAM 操作所需地址和数据
            state <= read;
      read:                                  //等待用户结束当前读操作
          if (fiford == 1)
            state <= read_over;
          else
            state <= read;
      read_over:                             //继续给出 SRAM 地址以保证数据稳定
            state <= idle;
      write_ready:                           //建立 SRAM 操作所需地址和数据
            state <= write;
      write:                                 //等待用户结束当前写操作
          if (fifowr == 1)
            state <= write_over;
          else
            state <= write;
      write_over:          //继续给出 SRAM 地址和写入数据以保证数据稳定
            state <= idle;
      default: state <= idle;
endcase
//产生 SRAM 操作相关信号
assign rd = ~state[2];        //state 为 read_ready 或 read 或 read_over
```

```
assign wr = (state == write) ? fifowr : 1'b1;

always @(posedge clk)
  if ( ~fifowr)
    in_data_buf <= in_data;

assign sram_data = (state[3]) ? //state 为write_ready 或write 或write_over
    in_data_buf : 8'hzz;

always @(state or fiford or fifowr or fifo_wp or fifo_rp)
  if (state[2] || ~fiford)
    address = fifo_rp;
  else if (state[3] || ~fifowr)
    address = fifo_wp;
  else
    address = 'bz;

//产生FIFO 数据
assign out_data = (state[2]) ?
                  sram_data : 8'bz;

always @(posedge clk)
  if (state == read)
    out_data_buf <= sram_data;

//计算FIFO 读写指针
always @(posedge clk or negedge rst)
  if ( ~rst)
    fifo_rp <= 0;
  else if (state == read_over)
    fifo_rp <= fifo_rp_next;

always @(fifo_rp)
  if (fifo_rp == `SRAM_SIZE-1)
    fifo_rp_next = 0;
  else
    fifo_rp_next = fifo_rp + 1;

always @(posedge clk or negedge rst)
  if ( ~rst)
    fifo_wp <= 0;
  else if (state == write_over)
    fifo_wp <= fifo_wp_next;
```

```
always @(fifo_wp)
  if (fifo_wp == `SRAM_SIZE-1)
    fifo_wp_next = 0;
  else
    fifo_wp_next = fifo_wp + 1;

always @(posedge clk or negedge rst)
  if (~rst)
    near_empty <= 1'b0;
  else if (fifo_wp == fifo_rp_next)
    near_empty <= 1'b1;
  else
    near_empty <= 1'b0;

always @(posedge clk or negedge rst)
  if (~rst)
    nempty <= 1'b0;
  else if (near_empty && state == read)
    nempty <= 1'b0;
  else if (state == write)
    nempty <= 1'b1;

always @(posedge clk or negedge rst)
  if (~rst)
    near_full <= 1'b0;
  else if (fifo_rp == fifo_wp_next)
    near_full <= 1'b1;
  else
    near_full <= 1'b0;

always @(posedge clk or negedge rst)
  if (~rst)
    nfull <= 1'b1;
  else if (near_full && state == write)
    nfull <= 1'b0;
  else if (state == read)
    nfull <= 1'b1;

endmodule
```

[练习题]　模仿本练习的设计思路和方法，设计一个利用同类静态 RAM 的堆栈。实现最大可达 1 024 个字节的 LIFO（Last In First Out）堆栈。

第二部分　Verilog 硬件描述语言参考手册

一、关于 IEEE 1364 标准

本部分是根据 IEEE 的标准《Verilog 硬件描述语言参考手册 1364 - 1995》编写的。OVI (Open Verilog International) 根据 Cadence 公司推出的 Verilog LRM(1.6 版)编写了 Verilog 参考手册 1.0 和 2.0 版,又根据这两个版本制定了 IEEE 1364 - 1995 Verilog 标准。在推出 Verilog 标准前,由于 Cadence 公司的 Verilog-XL 仿真器广泛使用,它所提供的 Verilog LRM 成了事实上的语言标准,许多第三方厂商的仿真器都努力向这一标准靠拢。

Verilog 语言标准化的目的是将现存的通过 Verilog-XL 仿真器体现的 Verilog 语言标准化。IEEE 的 Verilog 标准与事实上的标准有一些区别。因此,仿真器可能不能完全支持以下功能。

(1) 在 UDP(用户自定义原语)和模块实例中使用数组(见 Instantiation 说明)。

(2) 含参数的宏定义(见‘define)。

(3) ‘undef.

(4) IEEE 标准不支持用数字表示的强度值(见编译预处理命令)。

(5) 有许多 Verilog-XL 支持的系统任务、系统函数和编译处理命令在 IEEE 标准中不被支持。

(6) 若在模块中其 NEt 或寄存类型变量只有一个驱动,IEEE 标准允许在一个指定块中,延迟路径的最终接点可以是一个寄存器或 NEt 类型的变量。而在此标准推出之前,对最终接点的类型有着严格得多的要求(见 Specify 说明)。

(7) 指定路径的延迟表达式最多可以达到 12 个,表达式之间需用逗号隔开。在此标准推出之前,最多只允许 6 个表达式(见 Specify 说明)。

(8) 在 NEt 类型变量的定义中,标量保留字 scalared 与矢量保留字 vectored 的位置也做了改动,原来,保留字位于矢量范围的前面,在 IEEE 标准中,它应位于 NEt 类型的后面(见 NEt 说明)。

(9) 在最小、典型、最大值常量表达式中,对于最小、典型、最大值的相对大小并无限制。而原先最小值必须小于或等于典型值,典型值必须小于或等于最大值。

(10) 在 IEEE 标准中,表示延迟的最小、典型、最大值表达式不必括在括号里,而原来必

需括在括号里。

二、Verilog 简介

在 Verilog HDL 中，我们可通过高层模块调用低层和基本元件模块，再通过线路连接（即下文中的 NET）把这些具体的模块连接在一起，描述一个极其复杂的数字逻辑电路的结构。所谓基本元件模块，就是各种逻辑门和用户定义的原语模块（UDP）。而所谓 NET 实质上就是表示电路连线或总线的网络。端口连接列表用来把外部 NET 连接到模块的端口（即引脚）上。寄存器可以作为输入信号连接到某个具体模块的输入口。NET 和寄存器的值可取逻辑值 0、1、x（不确定）和 z（高阻）。除了逻辑值外，NET 还需要一个强度（Strength）值。在开关级模型中，当 NET 的驱动器不止一个时，需要使用强度值来表示。逻辑电路的行为可以用 initial 和 always 的结构和连续赋值语句，并结合设计层次树上各种层次的模块直到最底层的模块（即 UDP 及门）来描述。

模块中每个 initial 块、always 块、连续赋值、UDP 和各逻辑门结构块都是并行执行的。而 initial 及 always 块内的语句与软件编程语言中的语句在许多方面非常类似，这些语句根据安排好的定时控制（如时延控制）和事件控制执行。在 begin-end 块内的语句按顺序执行，而在 fork-join 块中的语句则并行执行。连续赋值语句只可用于改变 NET 的值。寄存器类型变量的值只能在 initial 及 always 块中修改。initial 及 always 块可以被分解为一些特定的任务和函数。PLI（Progrmmable Language Interface，可编程语言接口）是完整的 Verilog 语言体系的一个组成部分，利用 PLI 便可像调用系统任务和函数一样来调用 C 语言编写的各种函数。

Verilog 的原代码通常键入到计算机的一个或多个文本文件中，然后把这些文本文件交给 Verilog 编译器或解释器处理，编译器或解释器就会创建用于仿真和综合所必需的数据文件。有时候，编译完了马上就能进行仿真，没有必要创建中间数据文件。

三、语法总结

典型的 Verilog 模块的结构如下：

```
module M (P1, P2, P3, P4);
  input P1, P2;
  output[7:0_P3;
  inout P4;
  reg [7:0] R1, M1[1:1024];
  wire W1, W2, W3, W4;
```

```
parameter C1 = "This is a string";
  initial
    begin : 块名
      //声明语句
    end
  always @ (触发事件)
    begin
      //声明语句
    end
//连续赋值语句
assign W1 = Expression;
wire (Strong1, Weak0) [3:0] #(2,3) W2 = Expression;
//模块实例引用
  COMP U1 (W3, W4);
  COMP U2 (.P1(W3), .P2(W4));
task T1;  //任务定义
  input A1;
  inout A2;
  output A3;
    begin
      //声明语句
    end
endtask
function [7:0] F1;  //函数定义
  input A1;
    begin
      //声明语句
      F1 = 表达式;
    end
endfunction
endmodule  //模块结束
```

声明语句如下：

```
#delay
  wait (Expression)
@(A or B or C)
```

```
@(posedge Clk)
    Reg = Expression;
    Reg <= Expression;
  VectorReg[Bit] = Expression;
  VectorReg[MSB:LSB] = Expression;
  Memory[Address] = Expression;
  assign Reg = Expression
  deassign Reg;
  TaskEnable(...);
  disable TaskOrBlock;
EventName;
  if (Condition)
    ...
  else if (Condition)
    ...
  else
    ...

case (Selection)
  Choice1:
    ...
  Choice2, Choice3:
    ...
  default:
    ...
endcase
for (I =0; I<MAX; I = I +1)
    ...
    repeat (8)
    ...

  while (Condition)
    ...

  forever
    ...
```

上面简要语法总结可供读者快速参照，请注意其语法表示方法与本书中其他内容的区别。

四、编写 Verilog HDL 源代码的标准

编写 Verilog HDL 源代码应按标准进行，其标准可分成两种。第一种是语汇代码的编写标准，标准规定了文本布局、命名和注释的约定，其目的是为了提高源代码的可读性和可维护性。第二种是综合代码的编写标准，标准规定了 Verilog 风格，其目的是为了避免碰到的不能综合和综合结果存在缺陷的问题，也为了在设计流程中及时发现综合中会发生的错误。

下面列出的代码编写标准可根据所选择的工具和个人的爱好自行作一些必要的改动。

语汇代码的编写标准

(1) 每一个 Verilog 源文件中只准编写一个模块，也不要把一个模块分成几部分写在几个源文件中。

(2) 源文件的名字应与文件内容有关，最好一致(例 ModuleName. v)。

(3) 每行只写一个声明语句或说明。

(4) 代码用一层层缩进的格式来写。

(5) 用户定义变量名的大小写应自始至终一致(例如，变量名第一个字母大写)。

(6) 用户定义变量名应该是有意义的，而且含有一定的有关信息，而局部名(例如，循环变量)可以是简单扼要的。

(7) 通过注释对 Verilog 源代码作必要的说明(当然没有必要把 Verilog 源代码已经说明的内容再注释一遍)，对接口(例如，模块参数、端口、任务、函数变量)作必要的注释很重要。

(8) 尽可能多地使用参数和宏定义，而不要在源代码的语句中直接使用字母、数字和字符串。

可综合代码的编写标准

(1) 把设计分割成较小的功能块，每一个功能块都用行为风格去设计。除了设计中对速度响应要求比较临界部分外，都应避免使用门级描述。

(2) 应建立一个定义得很好的时钟策略，并在 Verilog 源代码中清晰地体现该策略(例如采用单时钟、多相位时钟、经过门产生的时钟、多时钟域等)。保证在 Verilog 源代码中时钟和复位信号是干净的(即不是由组合逻辑或没有考虑到的门产生的)。

(3) 要建立一个定义得很好的测试(制造)策略，并认真编写其 Verilog 代码，使所有的

触发器都是可复位的，使测试能通过外部管脚进行，又没有冗余的功能等。

（4）每个 Verilog 源代码都必须遵守并符合在 always 声明语句中介绍过的某一种可综合标准模板。

（5）描述组合和锁存逻辑的 always 块，必须在 always 块开头的控制事件列表中列出所有的输入信号。

（6）描述组合逻辑的 always 块一定不能有不完全赋值，也就是说所有的输出变量必须被各输入值的组合值赋值，不能有例外。

（7）描述组合和锁存逻辑的 always 块一定不能包含反馈，也就是说在 always 块中已被定义为输出的寄存器变量绝对不能再在该 always 块中读进来作为输入信号。

（8）时钟沿触发的 always 块必须是单时钟的，并且任何异步控制输入（通常是复位或置位信号）必须在控制事件列表中列出。

（9）避免生成不想要的锁存器。在无时钟的 always 块中，由于有的输出变量被赋予了某个信号变量值，而该信号变量没在该 always 块的电平敏感控制事件中列出，这会在综合中生成不想要的锁存器。

（10）避免不想要的触发器。在时钟沿触发的 always 块中，用非阻塞的赋值语句对寄存器类型的变量赋值，综合后就会生成触发器；或者当寄存器类型的变量在时钟沿触发的 always 块中经过多次循环其值仍保持不变，综合后也会生成触发器。

（11）所有内部状态寄存器必须是可复位的，这是为了使 RTL 级和门级描述能够被复位成同一个已知的状态以便进行门级逻辑验证（这并不适用于流水线或同步寄存器）。

（12）对存在无效状态的有限状态机和其他时序电路（例如，4 位十进制计数器有 6 个无效状态），如果想要在这些无效状态下，硬件的行为也能够完全被控制，那么必须用 Verilog 明确地描述所有的 2^N 种状态下的行为，当然也包括无效状态。只有这样才能综合出安全、可靠的状态机。

（13）一般情况下，在赋值语句中不能使用延迟，除了在 Verilog 的 RTL 级描述中需要解决零延迟时钟的倾斜问题之外，使用延迟的赋值语句是不可综合的。

（14）不要使用整型和 time 型寄存器，否则将分别综合成 32 位和 64 位的总线。

（15）仔细检查 Verilog 代码中使用动态指针（例如用指针或地址变量检索的位选择或存储单元）、循环声明或算术运算部分，因为这类代码在综合后会生成大量的门，而且很难进行优化。

五、设计流程

下面是用 Verilog 和综合工具设计 ASIC 或复杂 FPGA 的基本流程。围绕着设计流程作

多次反复是必要的,可在下面的流程中加入对此的说明,而且设计流程必须根据所设计的器件和特定的应用作必要的改动。

1 系统分析和指标的确定

2 系统划分

- 2.1 顶级模块
- 2.2 模块大小估计
- 2.3 预布局

3 模块级设计,对每一模块进行如下操作

- 3.1 写 RTL 级 Verilog
- 3.2 综合代码检查
- 3.3 写 Verilog 测试文件
- 3.4 Verilog 仿真
- 3.5 写综合约束、边界条件、层次
- 3.6 预综合以分析门的数量和延迟

4 芯片综合

- 4.1 写 Verilog 测试文件
- 4.2 Verilog 仿真
- 4.3 综合
- 4.4 门级仿真

5 测试

- 5.1 修改门级网表以便进行测试
- 5.2 产生测试向量
- 5.3 对可测试网表进行仿真

6 布局布线以使设计的逻辑电路能放入芯片

7 布局布线后进行仿真、故障覆盖仿真、定时分析

六、按字母顺序查找部分

always 声明语句

always 声明语句包含一个或一个以上的声明语句(如 procedural assigmment、task enable、if、case 和 for 声明语句),在仿真运行的全过程中,在定时控制下被反复执行。

语法

always

声明语句

在程序中的位置

module- <here> -endmodule

规则

在 always 块中被赋值的只能是寄存器类型的变量，如 reg、integer、real、time、realtime。每个 always 块从仿真开始起，在仿真的过程中不断地执行，执行完块中最后一个语句，继续从 always 块的开头执行。

注意 如果 always 块中包含一个以上的语句，则这些语句必须放在 begin-end 或 fork-join 块中。如果 always 块中没有时间控制，将会无限循环。

可综合性问题

always 声明语句是在综合过程中最有用的 Verilog 声明语句之一，但它经常是不可综合的。为了得到最好的综合结果，always 块的 Verilog 程序应严格按以下的模板来编写。

```
always @ (Inputs) //所有的输入信号都必须列出，在它们之间插入逻辑关系词 or begin
... //组合逻辑关系
end

always @(Inputs) //所有的输入信号都必须列出，在它们之间插入逻辑关系词 or
if (Enable)
begin
... //锁存动作
end

always @(posedge Clock) //Clock only
begin
... //同步动作
end

always @(posedge Clock or negedge Reset)
//Clock and Reset only
begin
if (! Reset) //测试异步复位电平是否有效
... //异步动作
else
... //同步动作
```

```
end  //可产生触发器和组合逻辑
```

示例

下面是一个寄存器级 always 块的例子。

```
always @(posedge Clock or negedge Reset)
    begin
      if (! Reset)  //Asynchronous reset
          Count <= 0;
      else
      if (! Load) //Synchronous load
          Count <= Data;
      else
          Count <= Count + 1;
    end
```

下面是一个描述组合逻辑电路的 always 块的例子。

```
always @(A or B or C or D)
    begin
          R = {A, B, C, D}
          F = 0;
      begin : Loop
            integer I;
              for (I = 0; I < 4; I = I + 1)
                if (R[I])
                begin
                  F = I;
                  disable Loop;
                end
          end //Loop
    end
```

参考

更多内容请参考 Begin、Fork、Initial、Statement、Timing Control 等声明语句的说明。

assign(连续赋值)声明语句

每当表达式中 Net(即连线)或寄存器类型变量的值发生变化时,使用 assign 声明语句就可在一个或更多的电路连接中创建事件。

语法

```
|either|
assign [Strength] [Delay] NetLValue = Expression,
  NetLValue = Expression,
  ...;
  NetType [Expansion] [Strength] [Range] [Delay]
  NetName = Expression,
  NetName = Expression,
  ...; |See Net|
  NetLValue = |either|
  NetName
  NetName[ConstantExpression]
  NetName[ConstantExpression: ConstantExpression]
  | NetLValue,...|
```

在程序中的位置

```
module -< HERE >- endmodule
```

规则

两种形式的连续赋值语句效果相同。

在 assign 声明语句之前,赋值语句左边的 Net(即连线类型的变量)必须明确声明。

注意 assign 声明语句并不等同于 proccdural continuous assignment 声明语句,虽然它们很相似。要确保把 assign 声明语句放在正确的地方,它必须放在 initial 和 always 块之外。procedural continuous assignment 声明可放在被允许放置的地方执行(在 initial、always、task、function 等声明语句内部)。

可综合性问题

(1) 综合工具不能处理 assign 声明语句中的延迟和强度,在综合中被忽略。请用综合工具指定的定时约束来代替。

(2) assign 声明语句将被综合成为组合逻辑电路。

提示 用 assign 声明语句去描述那些用简洁的表达式就能够很容易表达的组合逻辑电路。函数能够用来构建表示式。在描述较复杂的组合逻辑电路方面,用 always 块比用许多句分开的 assign 声明语句更好,而且仿真的速度更快一些。当 Verilog 需要电路连线时,可用 assign 声明语句把寄存器的值传送到电路连线上(即 Net 上)。例如,把一个 initial 块中产生的测试激励信号加到一个实例模块的输入/输出端口。

示例

```
wire cout, cin;
```

```
wire [31:0] sum, a, b;
  assign {cout, sum} = a + b + cin;
wire enable;
reg [7:0] data;
wire [7:0] #(3,4) f = enable ? data : 8'bz;
```

参考

更多内容请参考 net、force、procedural continuous assignment 等声明语句的说明。

begin 声明语句

begin 声明语句用于把多个声明语句组合起来成为一个语句，而其中每个声明语句的执行是按顺序进行的。Verilog 语法经常严格要求只有一个声明语句，always 块就是这样，如果 always 块需要有多个声明语句，那么这些声明语句必须被包含在一个 begin-end 块中。

语法

```
begin [: Label
    [Declarations...]]
    Statements...
end
Declaration = |either| Register Parameter Event
```

在程序中的位置

请参照 statement 声明语句中的说明。

规则

begin-end 块必须包含至少一个声明语句。声明语句在 begin-end 块中被顺序执行。定时控制是相对于前一声明语句的。当最后的声明语句执行完毕后，begin-end 块便结束。begin-end 和 fork-join 块可以自我嵌套或互相嵌套。如果 begin-end 块包含局部声明，则它必须被命名(即必须有一个标识)。如果要禁止(disable)某个 begin-end 块，那么被禁止的 begin-end 块必须有名字。

注意 Verilog LRM 允许 begin-end 块在仿真时被交替执行。这就是说，如果 begin-end 块包含两个相邻且其间没有时间控制的声明语句时，仿真器仍有可能在同一时刻在这两个语句之间执行另一个进程的部分语句(例如另一个 always 块中的语句)。这就是为什么 Verilog 语言如果不加约束，便不能与硬件有确定的对应关系。

提示 在并不需要局部声明，也不想禁止 begin-end 块时，也可以对该 begin-end 块加标识命名，以提高其可读性。给不用在别处的寄存器作局部声明，能使声明的意图变得清楚。

示例

```
initial
  begin : GenerateInputs
      integer I;
      for (I = 0; I < 8; I = I + 1)
      #Period [A, B, C] = I;
  end
initial
  begin
      Load = 0; //Time 0
      Enable = 0;
      Reset = 0;
      #10 Reset = 1; //Time 10
      #25 Enable = 1; //Time 35
      #100 Load = 1; //Time 135
  end
```

参考

更多内容请参考 Fork、Disable、Statement 等声明语句的说明。

case 声明语句

如果 case 声明语句中控制表达式与标号分支表达式相等,则执行该分支的声明语句。

语法

```
CaseKeyword (Expression)
  Expression,... : Statement {Expression may be variable}
  Expression,... : Statement
              ... {Any number of cases}
     [default [:] Statement] {Need not be at the end}
endcase
CaseKeyword = {either} case casex casez
```

在程序中的位置

请参照 statement 声明语句中的说明。

规则

(1) 不定值(Xs)和高阻值(Zs)在 casex 声明语句中,以及高阻值在 casez 声明语句的表

达式匹配中都意味着“不必考虑”。

(2) 在 case 声明语句中最多只允许有一个 default 项。当没有一个分支标号表达式能与 case 表达式的值相等时,便执行 default 项(标号是位于冒号左边的一个表达式或用逗号隔开的几个表达式,标号也可以是保留字 default,在其后面可以跟冒号也可以不跟冒号)。

(3) 如果某标号是用逗号隔开的两个或两个以上的表达式,只要其中任何一个表达式与 case 表达式的值相等,就可执行该标号的操作。

(4) 如果没有一个标号表达式与 case 表达式的值相等,又没有 default 声明语句,该 case 声明语句就没有任何作用。

注意 (1) 如果在标号分支中有一个以上的声明语句,这些声明语句必须放在一个 begin-end 或 fork-join 块中。

(2) 只有第一个与 case 表达式的值相等的标号分支才被执行。case 声明语句的标号并不一定是互斥的,所以当错误地重复使用相同的标号时,Verilog 编译器不会提示出错。

(3) casex 或 casez 声明语句的语法是用保留字 endcase 作为结束,而不是用 endcasex 或 endcasez 来结束。

(4) 在 casex 声明语句的表达式中的 X (不定值)或 Z(高阻值)可以和任何值相等,casez 声明语句中的 Z 也是如此。这有可能会给仿真结果带来混乱。

可综合性问题

case 声明语句中的赋值语句通常被综合成多路器。如果变量(如寄存器或 Net 类型)被用作 case 声明语句的标号,它就会被综合成优先编码器(Priority Encoders)。

在一个无时钟触发的 always 块中,如有不完整的赋值(即对某些输入信号的变化其输出仍保持不变,未能及时赋值),它将被综合成透明锁存器。

在一个有时钟触发的 always 块中,如有不完整的赋值,它将被综合成循环移位寄存器。

提示 (1) 为了使仿真能顺利进行,常常用 default 作为 case 声明的最后一个分支,以控制无法与标号匹配的 case 变量。

(2) 通常情况下用 casez 比用 casex 更好一些,因为 X 的存在可能会导致仿真出现令人误解和混乱的结果。

(3) 在 casex 和 casez 声明的标号中用“?”来代替“Z”比较好,因为这样做比较清楚,是一个无关项,而不是一个高阻项。

示例

```
case (Address)
    0 : A <= 1;          //Select a single Address value
    1 : begin            //Execute more than one statement
        A <= 1;
```

```
        B <= 1;
      end
    2, 3, 4 :C <= 1;    //Pick out several Address values
    default :           //Mop up the rest
      $display("Illegal Address value % h in % m at % t", Address, $realtime);
endcase
casex (Instruction)
  8'b000xxxxx : Valid <= 1;
  8'b1xxxxxxx : Neg <= 1;
  default
      begin
        Valid <= 0;
        Neg <= 0;
      end
endcase
casez ({A, B, C, D, E[3:0]})
  8'b1??????? : Op <= 2'b00;
  8'b010????? : Op <= 2'b01;
  8'b001??? 00 : Op <= 2'b10;
      default : Op <= 2'bxx;
endcase
```

参考

更多内容请参考 if 声明语句的说明。

comment(注释声明语句)

注释应该位于 Verillog 源代码文件中。

语法

单行注释

//

多行注释

/* ... */

在程序中的位置

Comment 语句可以放在源代码的几乎任何地方,但是要注意,不能把运算符、数字、字符串、变量名和关键字分开。

规则

单行注释以两个斜杠符开始，结束于该行的末尾。

多行注释以“/ * ”符开始，中间可能有多行，结束于“ * /”符。

多行注释不能嵌套，但是，在多行注释中可以有单行注释，但在这里它没有别的特殊含义。

注意　/ * ... / * ... * / ... * / - 这样的注释会出现语法错误，要注意注释符的匹配。

提示　建议在源代码文件中自始至终用单行注释。只有在必须注释一大段的地方才用多行注释，例如在代码的开发和调试阶段，常需要详细地注释。

示例

```
//This is a comment
/*
    So is this - across three lines
*/
module ALU /* 8-bit ALU */(A, B, Opcode, F);
```

参考

更多内容请参考 Coding Standard 编码标准。

defparam(定义参数)声明语句

使用 defparam 声明语句编译时可重新定义参数值。如果是分层次命名的参数，可以在该设计层次内或外重新定义参数。

语法

```
Defparam ParameterName = Constant Expression
    ParameterName = ConstantExpression,
    ... ;
```

在程序中的位置

```
module -< HERE >- endmodule
```

可综合性问题

一般情况下是不可综合的。

提示　最好不要使用 defparam 声明语句。该声明语句过去常用于布线后的时延参数反标中，但现在时延参数反标一般用指定模块和编程语言接口(PLI)来做。在模块的实例引用时可用“#”号后跟参数的形式来重新定义参数。

示例

```
'timescale 1ns /1ps
module LayoutDelays;
    defparam Design.U1.T_f = 2.7;
    defparam Design.U2.T_f = 3.1;
    ...
endmodule
module Design (...);
    ...
    and_gate U1 (f, a, b);
    and_gate U2 (f, a, b);
    ...
endmodule
module and_gate (f, a, b);
    output f;
    nput a, b;
    parameter T_f = 2;
      and #(T_f) (f,a,b);
endmodule
```

参考

更多内容请参考 name、instantiation、parameter 声明语句的说明。

delay(延迟)声明语句

. delay 声明语句可以为 UDP 和门的实例指定延迟,也可以为连续赋值语句和线路连接指定延迟。延迟是在网表中线路连接和元件传输时的模型。

语法

```
|either|
# DelayValue
#(DelayValue[, DelayValue[, DelayValue]]) |Rise,Fall,Turn-Off|
DelayValue = |either|
UnsignedNumber
ParameterName
ConstantMinTypMaxExpression
```

在程序中的位置

更多内容请参考 assign、instantiation 和 Net 声明语句的说明。

规则

(1) 如果只给出一个延迟值,则它既表示上升延迟也表示下降延迟(即从 0 转变到 1 或从 1 转变到 0 的时延),并且还表示关闭延迟(如果电路中有这样开关)。

(2) 如果给出两个延迟值,则第一个表示上升延迟,第二个表示下降延迟,除了 tranif0、tranif1、rtranif0 和 rtranif1 外,第一个值也可表示接通延迟,第二个表示关闭延迟。

(3) 如果给出 3 个延迟值,第三个延迟值表示关闭延迟(转变到高阻),除了三态电路外,第三个延迟值表示电荷衰减时间。

(4) 延迟到 X 表示最小的指定延迟。

(5) 对于向量,从非零到零的转变被看做下降,转变到高阻被看做关闭,其余的变化被看作是上升。

注意 许多工具要求 MinTypMax 延迟表达式必须用括号括起来。例如,#(1:2:3)是合法的,而 #1:2:3 是非法的。

可综合性问题

一般综合工具不考虑延迟。综合后网表中的延迟是由综合工具的命令项强制生成的,如在综合工具中可设置本次设计综合生成的门级电路所允许的最高时钟频率。

提示 指定块的延迟(即线路路径延迟)通常是一种更加精确的延迟建模方法,可提供延迟计算机制和布线后反标信息。

参考

更多内容请参考 net、instantiation、assign、Specify、timing control 等声明语句的说明。

disable(禁止)声明语句

在运行激活的任务或命名的块时,disable 声明语句能使在所在块执行完毕以前,终止该块的执行。

语法

```
disable BlockOrTaskName;
```

在程序中的位置

请参照 statement 声明语句。

规则

(1) 禁止命名块(即定义了名称的 begin-end 或 fork-join 块)或任务便是禁止了所有由该块或该任务激活的任务,直达该块或该任务层次树的底层。继续执行禁止(块或任务)语句后的声明语句。

(2) 命名块或任务可以通过其内部的禁止声明语句实现自我禁止。

(3) 当一个任务被禁止时,以下内容未被指定:任何一个输出值或输入/输出值、尚未起作用的非阻塞赋值语句、赋值和强制声明语句所预定的事件。

(4) 函数不能被禁止。

注意 任务被自我禁止与任务返回不一样,因为任务被自我禁止的输出未定义。

可综合性问题

只有当命名块或任务自我禁止时,禁止才是可综合的,一般情况下是不可综合的。

提示 禁止可作为一种及早跳出任务的方法,用来跳出循环或继续下一步循环。

示例

```
begin : Break      //命名 Break 块
    forever
    begin : Continue     //命名 Continue 块
        ...
      disable Continue;      //Continue with next iteration
        ...
      disable Break;      //Exit the forever loop
        ...
    end     //Continue
end         //Break
```

errors(错误)声明语句

下面列出的是编写 Verilog 源代码时最常犯的错误。前面的 5 个错误大约占所有错误的 50%。

最容易犯的五大错误

(1) 进程赋值语句的左侧变量没有声明为寄存器类型;

(2) begin-end 声明语句忘了配套;

(3) 写二进制数时忘了标明数基(即'b)。这样,在编译时会把它们当作十进制数来处理;

(4) 编译引导语句用了错误的撇号,应该用向后的撇号也就是用表示重音的撇号;而表示数基的撇号,应该是普通的撇号,也就是反向的逗号;

(5) 在声明语句的末尾忘了写上分号。

其他常犯的错误

(1) 在定义任务或函数时,试图在任务或函数名后用括号来定义变量;

(2) 在调试时,忘了在测试文件中引用实例模块;

（3）使用进程连续赋值语句而没有使用连续赋值语句（即赋值语句用错了地方）；

（4）把保留字作为标识符（例如用 xor 做标识符）；

（5）always 块忘了声明定时控制（导致无休止的循环）；

（6）在事件控制列表中错误地使用了逻辑或操作符（即 ‖），而没有使用或保留字 or，例如，把 always @（a or b）写成了 always @（a ‖ b）；

（7）用缺省定义的 wire 类型变量来做矢量端口的连线；

（8）模块实例引用时端口的连接次序搞错；

（9）在嵌套的 if-else 声明语句中 begin-end 配套错误；

（10）错误地使用等号。“ = ”用于赋值，“ == ”用于作数值比较，“ === ”用于需要对 0、1、X、Z 这 4 种逻辑状态作准确比较的场合。

event（事件）声明语句

在行为模型中，event 声明语句可以用来描述通信和同步。

语法

```
event Name ,...; |Declare the event|      //事件名,…;(事件声明)
 -> EventName; |Trigger the event|        // ->事件名(触发事件)
```

在程序中的位置

请参照为 -> 所作的声明语句。

event 声明语句可以放在如下位置：

```
module -< HERE >- endmodule
begin : Label -< HERE >- end
fork : Label -< HERE >- join
task -< HERE >- endtask
function -< HERE >- endfunction
```

规则

事件没有值，也没有延迟，它们仅被事件触发声明所触发，由沿敏感定时控制启动检测。

可综合性问题

通常是不可综合的。

提示　在测试文件和系统级模块中，命名事件可用于同一个模块的不同 always 块间或不同模块（用层次名）的 always 块间传递信息。

示例

```
event StartClock, StopClock;
```

```
always
    fork
      begin: ClockGenerator
          Clock = 0;
          @ StartClock
            forever
                #HalfPeriod Clock = ! Clock;
      end
      @ StopClock disable ClockGenerator;
    join

initial
    begin : stimulus
              ...
        -> StartClock;
              ...
        -> StopClock;
              ...
        -> StartClock;
              ...
       4 -> StopClock;
    end
```

参考

更多内容请参考 timing control 声明语句的说明。

expression(表达式)声明语句

表达式可以通过一系列的操作符、变量名、数字以及次级表达式来算出一个值。其中常量表达式是一种其值可在编译过程中计算出来的表达式。标量表达式的值是一比特二进制数。时间延迟可以用最小 - 典型 - 最大(即 MinTypMax)表达式来表示。

语法

```
Expression = |either|                                  //表达式 = |以下任取其一|
Primary                                                //基本表达式
Operator Primary |unary operator|                      //运算符 基本表达式|单目运算符|
Expression Operator Expression |binary operator|      //表达式 运算符 表达式
                                                       //|双目运算符|
Expression ? Expression : Expression                   //表达式 ? 表达式 :表达式
```

```
String                                          //字符串
.....
Primary = |either|                              //基本表达式 = |以下任取其一|
Number                                          //数字
Name |of parameter, net, or register|           //变量名 |参数,网络,或者寄存器的|
Name[Expression] |bit select|                   //变量名[表达式] |位选择|
Name[Expression: Expression]|part select|       //变量名[表达式:表达式] |部分位
                                                //选择|
MemoryName[Expression]                          //存储器名[表达式]
|Expression,...| |concatenation|                //|表达式,……| |位拼接|
|Expression| Expression,...|| |replication|     //|表达式|表达式,……|| |复制|
FunctionCall                                    //函数调用
(MinTypMaxExpression)                           //(MinTypMax 表达式)
|MinTypMax expressions are used for delays|     //| MinTypMax 表达式用于延迟|
MinTypMaxExpression = |either|                  //MinTypMax 表达式 = |任取其一|
Expression                                      //表达式
Expression: Expression: Expression              //表达式:表达式:表达式
```

规则

(1) 只有矢量类型的 Net 和寄存器、整数及时间类型变量才允许选取某位及某些位;

(2) 某些位的选取必须将高位列在冒号的左侧,低位列在右侧(最高位是在 Net 或寄存器类型声明中位于冒号左边的数值);

(3) 某位或某些位的选取时,若其中包含 X 或 Z 的位,或超出位的定义范围,在编译时可能会也可能不会被认定是错误的。如果不被认定是错的,编译器会给出一个值为 X 的表达式;

(4) 没有为存储器建立某位或某些位选取的机制;

(5) 当整型常量在表达式中作为操作数时,未标明进制的有符号数(例如 -5)与标明进制的有符号数(例如 -'d5)是有所区别的。前者被视为一个有符号数,而后者被视为一个无符号数。

注意　许多工具要求在常量 MinTypMax 表达式中必须指定最小、典型和最大延迟值(例如,min <= typ <= max)。

示例

```
A + B
! A
(A && B) || C
A[7:0]
```

```
B[1]
-4'd12/3                    //是一个很大的正数
"Hello" != "Goodbye"        //此表达式为真(1)
$realtobits(R);             //系统函数调用
{A, B, C[1:6]}              //位拼接(8 位)
1:2:3                       //最小-典型-最大表达式
```

参考

更多内容请参考 delay、funetion call、name、number 和 operator 声明语句的说明。

for(循环)声明语句

for 声明语句为一般用途的循环语句,允许一条或更多的语句能被重复地执行。

语法

```
for (RegAssignment;     {initial assignment}
    Expression;         {loop condition}
    RegAssignment)      {iteration assignment}
  Statement

RegAssignment = RegisterLValue = Expression //寄存器赋值 = 寄存器值 = 表达式
RegisterLValue = {either}                   //寄存器值 = {任取其一}
RegisterName                                //寄存器名
RegisterName[Expression]                    //寄存器名[表达式]
RegisterName[ConstantExpression:ConstantExpression]  //寄存器名[常量表达
                                                     //式:常量表达式]
Memory[Expression]                          //存储器[表达式]
{RegisterLValue,...}                        //{寄存器值,……}
```

在程序中的位置

请参照 statement 声明语句的说明。

规则

当 for 循环开始执行时,循环计数变量已赋于初始值。在每一次循环执行之前(包括第一次),都必须首先检查表达式的值,如果它为假(即为 0、X 或 Z),则立刻退出循环。而在每一次循环重复执行之后,都要对迭代次数寄存器重新赋值。

注意 不要使用位宽小的 reg 类型变量作为循环变量。在测试存有负数值的寄存器变量时要格外注意。由于加减操作是可替换的,并且 reg 类型变量被看成是无符号数,所以循环表达式可能永远不会为假,从而导致循环无限地进行。

```
reg [2:0] i;                          //i 始终界于 0 与 7 之间
...
for (i = 0; i <8; i = i +1 )          //循环永远不会停止
...
for (i = -4; i <0; i = i +1 )         //循环不可能执行
...;
```

在以上这些情况中,应将循环变量 i 定义为整型。

可综合性问题

如果循环的边界是固定的,那么在综合时该循环语句被认为是重复的硬件结构。

示例

```
V = 0;
for (I = 0; I < 4; I = I + 1)
  begin
  F[I] = A[I] & B[3 - I];             //4 个独立的与门
  V = V^A[I];                         //4 个级连的异或门
  end
```

参考

更多内容请参考 forever、repeat 和 while 声明语句的说明。

force(强迫赋值)声明语句

force 声明语句类似于 procedural continuous assignment 声明语句,可对 net 和寄存器类型变量实行强制赋值。常用于调试。

语法

```
|either|                                    //|任取其一|
force NetLValue = Expression ;               //force 网络参数值 = 表达式;
force RegisterLValue = Expression ;          //force 寄存器值 = 表达式;
|either|                                    //|任取其一|
release NetLValue;                           //release 网络参数值;
release RegisterLValue;                      //release 寄存器值
NetLValue = |either|                         //网络参数值 = |任取其一|
NetName                                      //网络变量名
|NetName,...|                               //|网络变量名|
RegisterLValue = |either|                    //寄存器值 = |任取其一|
RegisterName                                 //寄存器变量名
```

```
|RegisterName,...|                                    //|寄存器变量名|
```

在程序中的位置

请参照 statement 声明语句。

规则

(1) 不能对网络变量或寄存器变量的某位或某些位实行强制赋值或释放。force 声明语句具有比 procedural continuous assignment 声明语句更高的优先级。force 声明语句将会一直发挥作用直到另一个 force 声明语句对同一 Net 变量或寄存器变量执行强迫赋值,或者直到这个 Net 变量或寄存器变量被释放。

(2) 当作用在某一寄存器上的 force 声明语句被释放,寄存器并无必要立刻改变其值。如果此时没有 procedural continuous assignment 声明语句对这个寄存器赋值,则强制赋入的值会一直保留到下一个 procedural continuous assignment 声明语句的执行。

(3) 当作用在某个 Net 变量上的 force 声明语句被释放,该 Net 变量的值将由它的驱动决定,其值有可能会立刻更新。

可综合性问题

force 声明语句是不可综合的。

提示 force 声明语句常用于测试文件的编写,调试时常需要强制对某些变量赋值。不能用于模块的行为建模(此时应使用 assign 声明语句)。

示例

```
force f = a && b;
...
release f;
```

参考

procedural continuous assignment 声明语句。

forever(无限循环)声明语句

forever 声明语句使一个或一个以上语句无限循环地执行。

语法

```
forever Statement
```

在程序中的位置

请参阅 statement 声明语句中的说明。

注意 forever 循环应包括定时控制或能够使其自身停止循环,否则循环将无限进行下去。

可综合性问题

一般情况下是不可综合的。如果 forever 循环被@(posedge Clock)形式的时间控制打断,则是可综合的。

提示　forever 声明语句在测试模块中描述时钟时很有用。常用 disable 声明语句来跳出循环。

示例

```
initial
    begin : Clocking
        Clock = 0;
        forever
        #10 Clock = ! Clock;
    end
initial
    begin : Stimulus
        ...
        disable Clocking;      //停止时钟
    end
```

参考

更多内容请参考 for、repeat、while 和 disable 声明语句的说明。

fork(集合)声明语句

可将多个语句集合在一个块中,以使它们能被并发地执行。

语法

```
fork [ : Label                          //fork [ :块名
  [Declarations...]]                    //[块内声明语句……]]
  Statements...                         //语句……
join                                    //集合
Declaration = {either}                  //块内声明语句 = {任选其一}
Register                                //寄存器变量
Parameter                               //参数
Event                                   //事件
```

在程序中的位置

请参照 statement 声明语句中的说明。

规则

fork-join 块必须至少包括一条语句。fork-join 块里的语句是并发执行的，因此 fork-join 块内语句的顺序是无所谓的。时间控制是相对于块的开始时刻的。当 fork-join 块里所有的语句执行完毕后，块也就执行完毕了。begin-end 和 fork-join 块可以自我嵌套或互相嵌套。

如果想在某 fork-join 块内包含块内局部声明语句，那么必须对该块命名（即该块必须有一个标识符号）。如果想要禁止某 fork-join 块的运行，则该块必须已被命名。

可综合性问题

fork 声明语句不可综合。

注意 fork-join 声明语句在描述并发形式的行为时很有用。

示例

```
initial
    fork : stimulus
        #20 Data = 8'hae;
        #40 Data = 8'hxx;          //本句最后执行
        Reset = 0;                 //本句最先执行
        #10 Reset = 1;
    join                           //在第 40 个时间单位时结束
```

参考

更多内容请参考 begin、disable 和 statement 声明语句中的说明。

function（函数）声明语句

function 声明语句用于把多个语句组合在一起，来定义新的数学或逻辑函数。函数是在模块内部定义的，并且通常在本模块中调用，也能根据按模块层次分级命名的函数名从其他模块中调用。

语法

```
function [RangeOrType] FunctionName; //function [返回值的类型或范围] 函数名;
  Declarations...                    //端口声明...
  Statement                          //语句
endfunction
RangeOrType = |either|               //返回值的类型或范围 = |任取其一|
integer                              //整数
real                                 //实数
time                                 //时间
```

```
realtime
Range
Range = [ConstantExpression: ConstantExpression]   //范围 = [常量表达式:常量
                                                   //表达式]
Declaration = {either}                             //端口声明 = {任取其一}
input [Range] Name,...;                            //input [范围] 变量名,……;
Register
Parameter
Event
```

在程序中的位置

```
module -<HERE>- endmodule
```

规则

(1) 函数必须至少含有一个输入变量,它不能有任何输出或输入/输出双向变量;

(2) 函数不能包含时间控制语句(如延迟#,事件控制@ 或等待 wait);

(3) 函数是通过对函数名赋值的途径返回其值的,(就好比它是一个寄存器);

(4) 函数不能启动任务;

(5) 函数不能被禁用。

注意 (1) 函数的输入变量不能象模块的端口那样列在函数名后的括弧里;在声明输入时把这些输入端口列出即可。

(2) 如果函数包含一条以上的语句,这些语句必须包含在 begin-end 或 fork-join 块中。

可综合性问题

函数的每一次调用都被综合为一个独立的组合逻辑电路块。

示例

```
function [7:0] ReverseBits;
    input [7:0] Byte;
    integer i;
    begin
        for (i = 0; i < 8; i = i + 1)
        ReverseBits[7 - i] = Byte[i];
    end
endfunction
```

参考

更多内容请参考 function call 和 task 声明语句的说明。

function call(函数调用)声明语句

function call 声明语句可返回一个供表达式使用的值。

语法

```
FunctionName (Expression,... );          //函数名(表达式,……);
```

在程序中的位置

请参照 expression 声明语句的说明。

规则

函数必须至少含有一个输入变量,所以函数调用时总是至少含有一个表达式。

可综合性问题

函数的每一次调用在综合后都会生成一个独立的组合逻辑电路块。

示例

```
Byte = ReverseBits (Byte );
```

参考

更多内容请参考 function、expression 和 task enable 声明语句的说明。

gate(门)声明语句

Verilog 已有一些建立好的逻辑门和开关的模型。在所设计的模块中能通过实例引用这些门与开关模型,从而对模块进行结构化的描述。

逻辑门

```
and (Output, Input,...)
nand (Output, Input,...)
or (Output, Input,...)
nor (Output, Input,...)
xor (Output, Input,...)
xnor (Output, Input,...)
```

缓冲器与非门

```
buf (Output,..., Input)
not (Output,..., Input)
```

三态门

```
bufif0 (Output, Input, Enable)
bufif1 (Output, Input, Enable)
```

```
notif0 (Output, Input, Enable)
notif1 (Output, Input, Enable)
```

MOS 开关

```
nmos (Output, Input, Enable)
pmos (Output, Input, Enable)
rnmos (Output, Input, Enable)
rpmos (Output, Input, Enable)
```

CMOS 开关

```
cmos (Output, Input, NEnable, PEnable)
rcmos (Output, Input, NEnable, PEnable)
```

双向开关

```
tran (Inout1, Inout2)
rtran (Inout1, Inout2)
```

双向可控开关

```
tranif0 (Inout1, Inout2, Control)
tranif1 (Inout1, Inout2, Control)
rtarnif0 (Inout1, Inout2, Control)
rtranif1 (Inout1, Inout2, Control)
```

上拉源与下拉源

```
pullup (Output)
pulldown (Output)
```

真值表

在如下真值表中,逻辑值 L 与 H 代表部分未知值。L 表示 0 或 Z,H 表示 1 或 Z。

and	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

nand	0	1	X	Z
0	1	1	1	1
1	1	0	X	X
X	1	X	X	X
Z	1	X	X	X

or	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

nor	0	1	X	Z
0	1	0	X	X
1	0	0	0	0
X	X	0	X	X
Z	X	0	X	X

xor	0	1	X	Z
0	0	1	X	X
1	1	0	X	X
X	X	X	X	X
Z	X	X	X	X

xnor	0	1	X	Z
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

buf	
Input	Output
0	0
1	1
X	X
Z	X

Not	
Input	Output
0	1
1	0
X	X
Z	X

缓冲门、非门都可以有多个输出，但这些输出值都是相同的。

Bufif0		Enable			
		0	1	X	Z
DATA	0	1	Z	H	H
	1	0	Z	L	L
	X	X	Z	X	X
	Z	X	Z	X	X

Bufif1		Enable			
		0	1	X	Z
DATA	0	Z	0	L	L
	1	Z	1	H	H
	X	Z	X	X	X
	Z	Z	X	X	X

notif0		Enable			
		0	1	X	Z
DATA	0	1	Z	H	H
	1	0	Z	L	L
	X	X	Z	X	X
	Z	X	Z	X	X

notif1		Enable			
		0	1	X	Z
DATA	0	Z	1	H	H
	1	Z	0	L	L
	X	Z	X	X	X
	Z	Z	X	X	X

pmos rpmos		Control			
		0	1	X	Z
DATA	0	0	Z	L	L
	1	1	Z	H	H
	X	X	Z	X	X
	Z	Z	Z	Z	Z

nmos rnmos		Control			
		0	1	X	Z
DATA	0	Z	0	L	L
	1	Z	1	H	H
	X	Z	X	X	X
	Z	Z	Z	Z	Z

```
cmos (W, Datain, NControl, PControl);
```

等价于：

```
nmos (W, Datain, NControl);
pmos (W, Datain, PControl);
```

规则

（1）当 nmos、pmos、cmos、tran、tranif0 和 tranif1 类型的开关开启时，信号从输入传到输出并不改变其强度。

（2）当有电阻的开关，如 rnmos、rpmos、rcmos、rtran、rtranif0 和 rtranif1 类型的开关，开启时，信号从输入传到输出会按下表减小其强度。

强度	减至
supply	pull
strong	pull
pull	weak
large	medium
weak	medium
medium	small
small	small
highz	highz

参考

更多内容请参考 UDP 和 instantiation 声明语句的说明。

if 条件声明语句

根据条件表达式的逻辑值（真/假），执行两条/块语句中的一条/块。

语法

```
if (Expression)          //if(表达式)
  Statement              //语句
{else                    //[else
  Statement]             //语句]
```

在程序中的位置

请参阅 statement 声明语句的说明。

规则

当表达式的值为非零时被认为是真,当值为0、x 或 z 时被认为是假。

注意 (1) 如果在 if 或 else 分支中有超过一条的语句需要执行,则必须用 begin-end 或 fork-join 将其包括。

(2) 在使用嵌套的 if-else 语句时,当 else 分支省略时,要特别注意。else 只与离它最近的前面的那个 if 相关联。Verilog 编译器不能判别源代码中省略的 else 分支。

可综合性问题

(1) if 声明语句中的赋值语句通常被综合为多路器。在无时钟的 always 块中,那些在输入变化时而输出仍能保持不变的赋值语句将被综合为透明锁存器,而在有时钟的 always 块中,它们则被综合为循环锁存器。

(2) 在某些情况下,嵌套的 if 语句会被综合为多层的逻辑。用 case 语句可以避免出现这种情况。

提示 如果对某些条件需要先进行测试,在这种情况下应选用嵌套的 if－else 语句。如果所有的条件优先权一致,则应选用 case 语句。

示例

```
if (C1 && C2)
    begin
      V = ! V;
      W = 0;
        if (! C3)
          X = A;
        else if (! C4)
                X = B;
        else
        X = C;
end
```

参考

更多内容请参考 case 与 operators 声明语句的说明。

initial 声明语句

initial 是在仿真一开始就执行并只执行一次的声明语句,可执行只包含一条语句或由多条语句组成的块。

语法

```
initial
    Statement
```

在程序中的位置

```
module -< HERE >- endmodule
```

可综合性问题

initial 声明语句是不可综合的。

注意 包含多个语句的 initial 块需要用 begin-end 或 fork-join 块将这些语句合成一块。

提示 在仿真测试文件中可使用 initial 声明语句来描述激励。

示例

下面的例子给出如何使用 initial 声明语句在测试文件中产生矢量。

```
reg Clock, Enable, Load, Reset;
reg [7:0] Data;
parameter HalfPeriod = 5;
    initial
      begin : ClockGenerator
        Clock = 0;
        forever
            #(HalfPeriod) Clock = ! Clock;
    end
initial
    begin
      Load = 0;
      Enable = 0;
      Reset = 0;
      #20    Reset = 1;
      #100   Enable = 1;
      #100   Data = 8'haa;
```

```
            Load = 1;
        #10 Load = 0;
        #500 disable ClockGenerator;          //停止时钟的产生
    end
```

参考

更多内容请参考 always 声明语句的说明。

instantiation(实例引用)声明语句

实例(instance)是模块、UDP 或门的唯一拷贝。通过实例的引用可以生成设计的各个层次。设计的行为也能通过引用 UDP、门和其他的模块的实例,并用电路连线(Net)将它们连接起来,从结构上加以描述。

语法

```
|either|                                          //|任取其一|
ModuleName
    [#(Expression,...)] ModuleInstance,...;      //模块名 [#(表达式,……)] 实例模
                                                 //块,……;
UDPOrGateName [Strength]
    [Delay] PrimitiveInstance,...;               //UDP 或门名[强度][延迟]原始实例,……;
ModuleInstance =                                 //实例模块 =
InstanceName [Range] ([PortConnections])         //实例名[范围]([端口连线])
PrimitiveInstance =                              //原始实例 =
[InstanceName [Range]] (Expression,...)          //[实例名[范围]](表达式,……)
Range =
    [ConstantExpression: ConstantExpression]     //范围 = [常量表达式:常量表达式]
PortConnections = |either|                       //端口连线 = |任取其一|
[Expression] ,... |ordered connection|           //[表达式],……|有顺序的连线|
 · PortName([Expression]) ,... |named connection|  // · 端口名([表达式]),……
                                                 //|指定连线|
```

在程序中的位置

```
module -< HERE >- endmodule
```

规则

(1) 命名的端口连线只能用于模块实例。

(2) 如果给出端口的连线次序列表,则在引用实例时其端口必须按次序与模块或门的端口一一对应。

(3) 如果给出命名的端口连线列表时,则在引用实例时其端口顺序是无关紧要的,但其端口的名字必须与模块的端口名字一致。

(4) 如果给出端口的连线次序列表,在引用实例时,其端口列表中若有两个邻近的逗号,则会因为没有表达式而导致相应端口未连线。如果给出命名的端口连线列表时,在引用实例时,其端口列表中若没有某端口的名字或虽有端口的名字但在括号内没有表达式,也会导致该端口未连线。

(5) 任何表达式都可用来与输入端口相连,但输出端口只能与 Net(线路)、一位或多位的连线或这些位的拼接线相连。输入表达式生成隐含的连续赋值。

(6) 如果在模块实例定义时给出了范围,其含义是定义了一个含有同种的多个子实例的实例模块。如果端口表达式的位长与定义的实例模块相应端口位长(即多个同种 UDP 或门端口位数的总和)一致时,整个表达式都将与每个子实例的端口相连。如果位长不一致,太多或太少,都会出错误。

(7) "#"符号有两种不同的用途。它既可用来强制修正实例模块中的一个或多个参量,也可用于为 UDP 或门实例指定延迟。对于实例模块,"#"符号后的第一个表达式替代模块中声明的第一个参量,第二个表达式替代模块中声明的第二个参量,以此类推。

(8) 实例引用 pullup、pulldown、tran 和 rtran 这些类型的门时不允许有延迟。

(9) 对于 nmos、pmos、cmos、rnmos、rpmos、rcmos、tran、rtran、tranif0、tranif1、rtranif0 和 rtranif1 这些类型的开关不能定义强度。

注意 (1) 在按顺序的端口的列表中很容易不小心将两个端口的次序弄混。若这些端口的位宽和方向相同,不会报告出错,只有在仿真出现错误结果后,才能找到。这类错误往往很难发现。使用命名的端口连线能避免实例模块引用中出现这类问题。

(2) 多个模块、UDP 或门的成组实例引用的语法是最近才加入 Verilog 语言的标准中,目前还没有工具支持这语法。

可综合性问题

UDP 和开关的实例引用一般是不能综合的。

提示 使用命名的端口连接方式可以提高程序的可读性并减少发生错误的可能性(见前文)。

端口表达式只使用位、部分位和位拼接的变量名。如果需要,则应尽量使用独立的连续赋值语句,对实例模块引入信号。

示例

UDP 实例引用:

```
Nand2 (weak1,pull0) #(3,4) (F, A, B);
```

模块实例引用:

```
Counter U123 (.Clock(Clk), .Reset(Rst), .Count(Q));
```

在下面的两个例子中 QB 端口没有连接：

```
DFF Ff1 (.Clk(Clk), .D(D), .Q(Q), .QB());
DFF Ff2 (Q,, Clk, D);
```

下面是在端口连线表中使用门表达式的例子：

```
nor (F, A&&B, C)      //不要这样使用
```

下面是一个多实例模块引用的例子：

```
module Tristate8 (out, in, ena);
  output [7:0] out;
  input [7:0] in;
  input ena;
  bufif1 U1[7:0] (out, in, ena);
  /* 上面的一条语句等同下面的 8 条语句
  bufif1 U1_7 (out[7], in[7], ena);
  bufif1 U1_6 (out[6], in[6], ena);
  bufif1 U1_5 (out[5], in[5], ena);
  bufif1 U1_4 (out[4], in[4], ena);
  bufif1 U1_3 (out[3], in[3], ena);
  bufif1 U1_2 (out[2], in[2], ena);
  bufif1 U1_1 (out[1], in[1], ena);
  bufif1 U1_0 (out[0], in[0], ena);
  */
endmodule
```

参考

更多内容请参考 module、user defined primitive、gate、port 声明语句的说明。

module(模块定义)声明语句

在 Verilog 语言中,模块是层次的基本单元。模块中包括声明语句、功能描述和引用一些现存的硬件部件。有些模块只用来声明可被别的模块调用的参量、任务和函数。在这类模块中没有任何 initial 块、always 块、连续赋值语句和实例引用,因而实际上不存在相应的硬件元件与之对应 。

语法

```
{either}                                  //{任取其一}
module ModuleName [(Port,...)];           //module 模块名[(端口,……)];
```

```
    ModuleItems...                                  //模块条款
endmodule                                           //endmodule
macromodule ModuleName [(Port,...)];                //macromodule 模块名[(端口,……)];
    ModuleItems...                                  //模块条款
endmodule                                           //endmodule
ModuleItem = |either|                               //模块条款 = |任取其一|
Declaration                                         //声明
Defparam                                            //参数定义
ContinuousAssignment                                //连续任务
Instance                                            //实例引用
Specify                                             //详细说明块
Initial                                             //初始化块
Always                                              //总是执行块
Declaration = |either|                              //声明 = |任取其 -|
Port                                                //端口
Net                                                 //网络
Register                                            //寄存器
Parameter                                           //参量
Event                                               //事件
Task                                                //任务
Function                                            //函数
```

在程序中的位置

在其他模块或 UDP 外。

规则

(1) 几个模块或几个 UDP(或它们的混合)可以在一个文件中进行描述(事实上,一个模块也可以分开在两个或更多的文件中描述,但不推荐这种做法)。

(2) 模块也可使用关键字 macromodule 来定义。其语法与用关键字 module 来定义模块是完全一样的。

(3) Verilog 编译器在编译宏模块时与编译一般模块时有所不同,比如不必为宏模块实例创建层次。这样,从仿真速度或存储介质的开销两个方面来说,宏模块的编译更有效率。为了达到这个目的,宏模块的编译可能受制于实现时的某些特殊条件。如果遇到这种情况,宏模块将被按一般模块编译。

注意 模块与宏模块都以关键字 endmodule 作为结束标志。

可综合性问题

(1) 每一个模块都被综合为一个独立的分层块,虽然有些工具的缺省配置规定把层次展平(为单层),但仍允许用户对综合后生成的网表层次进行控制。

(2) 不是所有的工具都支持宏模块的综合。

提示 尽量使每一个文件只包含一个模块。在大型设计中,这样做易于源代码的维护。

示例

```
macromodule nand2 (f, a, b);
    output f;
    input a, b;
     nand (f, a, b);
endmodule

module PYTHAGORAS (X, Y, Z);
    input [63:0] X, Y;
    output [63:0] Z;
    parameter Epsilon = 1.0E-6;
    real RX, RY, X2Y2, A, B;
      always @(X or Y)
        begin
          RX = $bitstoreal(X);
          RY = $bitstoreal(Y);
          X2Y2 = (RX * RX) + (RY * RY);
          B = X2Y2;
          A = 0.0;
          while ((A - B) > Epsilon || (A - B) < -Epsilon)
            begin
              A = B;
              B = (A + X2Y2 /A) /2.0;
            end
        end
      assign Z = $realtobits(A);
endmodule
```

参考

更多内容请参考 user defined primitive、instantiation、name 声明语句的说明。

name(名字)声明语句

任何用 Verilog 语言描述的"东西"都通过它的名字来识别。

语法

```
Identifier                                                    //标识符
\EscapedIdentifier |terminates with white space|  //\扩展标识符|空格表示结束|
```

规则

(1) 标识符可由字母、数字、下划线和美元符号构成。第一个字符必须是字母或下划线,而不能是数字或美元符号。

(2) 一个扩展标识符用反斜杠引出,用空格结束(空格符、制表符、回车键或换行键),并且可包含除空格以外的任何可印刷的字符。反斜杠和空格并不算作标识符的部分,例如,标识符 Fred 与扩展标识符\Fred 是相同的。

(3) 在 Verilog 中变量名是大小写敏感的。

(4) 在 Verilog 文件中,一个名字不能有多于一个以上的含义。名字的内部声明(例如在 begin-end 块中的名字)能屏蔽外部声明(例如包含该命名 begin-end 块上层模块的变量声明语句)。

hierarchical names(分级名字)声明语句

(1) Verilog HDL 中的每个标识符都有唯一的分级名字。这意味着任何 Net、寄存器、事件、参量、任务和函数都能通过使用它的分级名,在标识符的声明块外对它进行访问。

(2) 在名字层次的最上层是不需要实例引用的模块名。顶层测试模块就是一个最上层模块例子(尽管可能会有不止一个顶层模块在同一仿真中运行)。

(3) 在每个实例模块、命名块、任务和函数的定义时,便定义了名字层次树上的新层。

(4) Verilog 变量的分级名字是由从顶层模块名字开始直到包含该变量的模块实例名、命名块、任务和函数名构成,其间用小圆点隔开。

upwards name referencing(向上索引名)声明语句

包含两个标识符中间用点号隔开的分级名可能是下列情况中的一种:

(1) 当前模块所引用的实例模块中的一项(这是向下引用);

(2) 顶层模块中的一项(这是一个分级名字);

(3) 当前模块的父模块中引用的实例模块中的一项(这是向上引用);

(4) 向上引用名字的第一个标识符既可能是一个模块名也可能是一个模块实例的名字。

可综合性问题

分级名字和向上索引名在一般底档综合工具上是不可综合的。

提示 (1) 通常,应选择对读者来说有含义的名字。相对本地名而言,这一点对于全局名显得更为重要。例如,给全局复位信号起名为 G0123,这名字不好,是因为它没有含义,而把 I 作为循环变量却是易于接受的。

(2) 名字中不要使用扩展字符。如网表生成或综合这一类 EDA 工具,它们具有与 Verilog 不同的命名规则,常留给这些扩展字符某些特殊含义。

(3) 分级名字仅用于测试模块或那些无法改用别的合适名字的高层系统模型中。

(4) 避免使用向上索引名,因为它们会导致代码非常难理解,从而给调试和维护带来麻烦。

示例

以下是合法名的例子:

```
A_99_Z
Reset
_54MHz_Clock$
Module                    //与"module"是不一样的
\$%^&*()                  //扩展标识符
```

以下是因上述原因而不合法的名字:

```
123a                                  //名字不能用数字开始
$data                                 //名字不能用美元符号
module                                //名字不能用保留字
```

下面的例子说明了分级名字和向上引用名字:

```
module Separate;
    parameter P = 5;                  //Separate.P
endmodule

module Top;
    reg R;                            //Top.R
      Bottom U1 ();
Endmodule

Module Bottom;
    reg R;                            //Top.U1.R
      task T;                         //Top.U1.T
```

```
        reg R;                          //Top.U1.T.R;
        ...
      endtask
  initial
    begin : InitialBlock
      reg R;                            //Top.U1.InitialBlock.R;
        $display(Bottom.R);             //向上索引名指向 Top.U1.R
        $display(U1.R);                 //向上索引名指向 Top.U1.R
        ...
    end
  endmodule                             //end of Bottom module
```

Net(线路连接)声明语句

Net 是结构描述中为线路连接(连线和总线)建立的模型。Net 的值是由 Net 的驱动器所决定的。驱动器可以是门、UDP、实例模块或者连续赋值语句的输出。

语法

```
|either|
NetType [Expansion] [Range] [Delay] NetName,...;
trireg [Expansion][Strength] [Range] [Delay]
NetName,...;
|Net declaration with continuous assignment| 用连续声明语句对 Net 进行声明
NetType [Expansion][Strength] [Range] [Delay]
NetAssign,...;
NetAssign = NetName = Expression
NetType = |either|
wire tri |equivalent|
wor trior |equivalent|
wand triand |equivalent|
tri0
tri1
supply0
supply1
Expansion = |either|
vectored scalared
Range = [ConstantExpression: ConstantExpression]
```

在程序中的位置

```
module -< HERE >- endmodule
```

规则

(1) supply0 和 supply1 类型的 Net 分别具有逻辑值 0 和 1,并可以为它定义驱动能力(Supply Strength)。

(2) tri0 和 tri1 类型的 net,当没有驱动时,分别具有逻辑值 0 和 1,并可以为它定义驱动能力(Pull Strength)。

(3) 如果 Net 的扩展(Expansion)选项选用了关键词 vectored,则不允许对它进行某位和某些位的选择,也不允许对它定义强度,因为 PLI 会认为该 Net 是不可扩展的;如果扩展选项选用了关键词 scalared,则允许对它进行某位和某些位的选择,也允许对它定义强度,PLI 将会认为该 Net 是可扩展的,这些关键词是有参考价值的。

(4) 除了结构描述中的端口和标量连线不用声明其 Net 类型外,其他类型的 Net 变量在应用之前必须声明。

真值表

当 Net 具有两个或两个以上驱动时,同时假定其驱动器强度值均相等,这些真值表则告诉我们输出的结果。如果不相等,则驱动强度大者驱动该 Net。

wire tri	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

wand triand	0	1	X	Z
0	0	0	0	0
1	0	1	X	1
X	0	X	X	X
Z	0	1	X	Z

wor trior	0	1	X	Z
0	0	1	X	0
1	1	1	1	1
X	X	1	X	X
Z	0	1	X	Z

tri0	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	1	X	X
Z	0	1	X	0

tri1	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	1	X	X
Z	0	1	X	1

注意 (1) 当 net 未被驱动时,对 tri0 或 tri1 类型的 Net 的连续赋值不影响其值和强度,经常为强度(Strength)保持为 Pull,逻辑值保持为 0(对 tri0)或 1(对 tri1)。

(2) 在 IEEE 标准和已成事实的 Cadence 公司标准中,扩展可选项的保留字 scalared 或 vectored 的位置有所不同,在 Cadence 公司标准中,保留字位于范围(Range)选项的跟前。

可综合性问题

(1) Net 类型的变量被综合成线路连接,但是某些线路连接经优化后有可能被删去。

(2) 综合工具只支持 Net 类型中 wire 型的综合,其他的 Net 类型均不支持。

提示:(1) 在每个模块的块首明确地声明所有的 Net,即使是缺省的类型也应该明确地加以说明。通过清楚地说明设计意图,可以提高 Verilog 程序的可读性和可维护性。

(2) 只能用 supply0 和 supply1 来声明地和电源。

示例

```
wire Clock;
wire [7:0] Address;
tril [31:0] Data, Bus;
trireg (large) C1, C2;
wire f = a && b,
g = a || b;                    //连续赋值
```

参考

更多内容请参考 assign 和 register 声明语句的说明。

number(数)声明语句

数可以是整数或者实数。在 Verilog 中整数是通过若干位来表示的,其中某些位可以是不定值(X)或高阻态(Z)。

语法

```
|either|
BinaryNumber                   //二进制数
OctalNumber                    //八进制数
DecimalNumber                  //十进制数
HexNumber                      //十六进制数
RealNumber                     //实数
BinaryNumber = [Size] BinaryBase BinaryDigit...
OctalNumber = [Size] OctalBase OctalDigit...
DecimalNumber = |either|
[Sign] Digit... |signed number|
[Size] DecimalBase Digit...
HexNumber = [Size] HexBase HexDigit...
RealNumber = |either|
[Sign] Digit... .Digit...
```

```
[Sign] Digit...[.Digit...]e[Sign] Digit...
[Sign] Digit...[.Digit...]E[Sign] Digit...
BinaryBase = |either| 'b 'B
OctalBase = |either| 'o 'O
DecimalBase = |either| 'd 'D
HexBase = |either| 'h 'H
Size = Digit...
Sign = |either| +
-Digit = |either| _0 1 2 3 4 5 6 7 8 9
BinaryDigit = |either| _x X z Z ?0 1
OctalDigit = |either| _x X z Z ?0 1 2 3 4 5 6 7
HexDigit = |either| _x X z Z ?0 1 2 3 4 5 6 7 8 9 a A
b B c C d D e E f F
UnsignedNumber = Digit...
```

在程序中的位置

请参阅 expression 声明语句的说明。

规则

(1) 表示进制的字母、十六进制数、X 和 Z 在数的表示中是不区分大小写的,字符 Z 和? 在数的表示中是等价的。

(2) 数字中不能有空格,但是在表示进制的字母两侧可以出现空格;

(3) 负数表示为其二进制的补数;

(4) 数字的第一个字符不允许出现下划线,但标识符可以。为了提高数字的可读性,可用下划线把长的数字分段,在处理数字时下划线将被忽略;

(5) 位宽指明了数字的准确位数;

(6) 不指明位宽的数字,它的位宽应为 32 位或 32 位以上,这取决于主机字长;

(7) 如果位宽大于实际的二进制位数时,高位部分补 0,但除非左边最高位是 X 或 Z,在这种情况下,则补 X 或 Z;

(8) 如果位宽小于实际的二进制数位时,超过位宽的高位(左边)被舍去。

注意 定义了位宽的负数被赋值到寄存器后,它将被认为是无符号的数。

```
reg [7:0] byte;
reg [3:0] nibble;
initial
begin
nibble = -1;          //例如 4'b1111
```

```
byte = nibble;          //变为 8'b0000_1111
end
```

当寄存器类型的数或者定义了位宽的数被用在表达式中时,其值通常被当成一个无符号数。

```
integer i;
initial
i = -8'd12 /3;          //i 变成 81 (即 8'b11110100 /3)
```

可综合性问题

(1) 0 和 1 分别被综合成接地和接电源的连线,赋值为 X 的则被认为是无关项。除了使用 casex 语句,如用其他的条件语句,与 X 的比较都认为是假的(case 等式运算符 = = = 和! = = 一般情况下都是不可综合的)。

(2) 除了在 casex 和 casez 语句中 Z 被认为是无关项,在其他情况下 Z 则被用来表示三态驱动器。

提示 (1) 在 case 语句的标号中,通常用"?"要比用 Z 好。在程序的其他地方不要使用"?"号,否则会产生混淆。

(2) 用下划线来分隔较长的数字,从而提高可读性。

示例

```
-253          //有符号的十进制数
'Haf          //未定义位宽的十六进制数
6'o67         //位宽为 6 的八进制数
8'bx          //位宽为 8 的二进制数,其值为不定值
4'bz1         //位宽为 4 的二进制数,最低位为 1 其余高三位均为高阻值(4'bzzz1)
```

下面所列的数为不合法的数并解释其原因:

```
_23           //以_开头
8' HF F       //包含两个非法空格
0ae           //十进制数中出现十六进制数字
x             //是名字,不是数字(应用 1'bx)
.17           //应该是 0.17
```

参考

更多内容请参考 expression 和 string 声明语句的说明。

operator(运算符)声明语句

在表达式中,使用运算符便可根据操作数(诸如数字、参量以及其他子表示式)计算出表达式的值。Verilog 语言中的运算符和 C 语言中的很相似。

单目运算符

+、-	正负号
!	逻辑非
~	按位取反
&、~&、|、~|、^、~^、^~	缩位运算符（~^ 和 ^~ 等价）

二目运算符

+、-、*、/	算术运算符
%	取模运算符
>、>=、<、<=	关系运算符
&&、||	逻辑运算符
==、!=	逻辑等式运算符
===、!==	case 等式运算符
&、|、^、^~、~^	逐位运算符（^~ 和 ~^ 等价）
<<、>>	移位运算符

其他运算符

A ? B : C	条件运算符
{ A, B, C }	位拼接运算符
{ N{A} }	重复运算符

在程序中的位置

参考 expression 声明语句中的说明。

规则

（1）逻辑运算符把它的操作数当成布尔变量。例如，非零的操作数被认为是真（1'b1）；零被认为是假（1'b0）；不确定的值，例如 4'bXX00，因不能判断其值为真还是假，就被认为是不确定的（1'bX）。

（2）位运算符（~、&、|、^、^~、~^）和全等运算符（===、!==）把它们操作数逐位进行处理。

（3）在包含==或!=的逻辑比较式中，如果有任何一个操作数为 x 或 z，其结果便是不确定的（1'bX）。（请仔细看注意事项）

（4）在包含（<、>、<=、>=）的比较式中，如果操作数不确定，其结果为不定值。（1'bX）例如：

```
2'b10 > 1'b0x          //结果为真
2'b11 > 1'b1x          //结果不定(1'bX)
(请看注意事项)
```

(5) 缩位运算符(&、~&、|、~|、^、~^、^~)将一个矢量缩减为一个标量。

(6) 位宽确定的表达式的运算采用溢出的位不计的办法，例如,4'b1111 + 4'b0001 = 4'b0000。

(7) 整数作除法运算时,小数部分被截掉。

(8) 取模运算(%)的结果是第一个操作数被第二个操作数除的余数,符号与第一个操作数一致。

(9) 只有某些特定的运算符允许出现在实数表达式中,例如单目运算符 + 和 - 、算术运算符、关系运算符、逻辑运算符以及条件运算符。实数逻辑或关系运算符的结果是一个只有一位的值。

运算符的优先级

+、-、!、~	单目(unary)	最高优先级
*、/、%		
+、-	双目(binary)	
≪、≫		
<、<=、>、>=		
==、!=、== =、!==		
&、~&		
^、^~		
|、~|		
&&		
||		
?:		最低优先级

注意 (1) 应用 ==、!=、<、>、<=和>= 对某些位不确定的值进行比较的规则并不适用于所有的仿真器,这点请特别注意!

(2) 注意单目缩位运算符与逐位逻辑运算符之间的区别,运算符本身是相同的,可根据上下文的关系来判断是哪一种,有时必须要用括号才能表达清楚。

可综合性问题

(1) 逻辑运算符、逐位运算符、移位运算符是可综合的,都被综合成逻辑运算;

(2) 条件运算符是可综合的,被综合成多路器或带使能端的三态门;

(3) 运算符 + 、- 、* 、< 、<= 、> 、>= 、== 和 != 都是可综合的,被分别综合成加法器、减法器、乘法器和比较器;

(4) 运算符/和%一般是不可综合的,只有当能用移位寄存器来表示运算时才是可综合的。而常量的/和%运算是可综合,但结果只能用二进制数表示;

(5) 其他运算符均不能被任何工具所综合。

提示 在写表达式的时候,运用括号要比依靠运算符的优先级要好,这样可预防错误产生,并且使那些不太了解 Verilog 语言的人更容易理解你的意思。

示例

```
-16'd10                    //这是表达式,是负运算,不是有符号数
a+b
x % y
Reset && ! Enable          //与 Reset && (! Enable)相同
a && b || c && d           //与 (a && b) || (c && d)相同
~4'b1101                   //结果为 4'b0010
&8'hff                     //结果为 1'b1,即一位的逻辑值 1
```

参考

更多内容请参考 expression 声明语句的说明。

parameter(参数)声明语句

参数是为常数命名的一种手段。在 Verilog 代码模块编译时(而不是在仿真期间),可以改写参数的值。使用参数就有可能重新定义 Verilog 代码中的常数,如数组的宽度等。

语法

```
parameter Name = ConstantExpression,
Name = ConstantExpression,
...;
```

有些工具支持下列非标准的语法:

```
parameter[Range] Name = ConstantExpression,
Name = ConstantExpression,
...;
Range = [ConstantExpression: ConstantExpression]
```

在程序中的位置

```
module -< HERE >- endmodule
begin : Label -< HERE >- end
fork : Label -< HERE >- join
task -< HERE >- endtask
function -< HERE >- endfunction
```

规则

(1) 参数是常量,在仿真期间更改参数的值是非法的;

(2) 在编译期间用 defparam 或者当包含参数的模块被引用时，可以改写其参数的值。

可综合性问题

有些综合工具能把含有参数的模块当作模板，一旦读入模板，便能够用不同的参数值多次对该模板进行综合。所有的综合工具都支持不带改动参数的模块实例的综合。

提示 尽可能用参数给常数起一个有含义的名字。

示例

下面的例子是一个 *N* 位宽的(可通过参数改变位宽的)移位寄存器。实例引用该参数化移位寄存器时可重新定义不同的位宽。

```
module Shifter (Clock, In, Out, Load, Data);
  parameter NBits = 8;
  input Clock, In, Load;
  input [NBits-1:0] Data;
  output Out;
  always @(posedge Clock)
  if (Load)
  ShiftReg <= Data;
  else
  ShiftReg <= {ShiftReg[NBits-2:0], In}
  assign Out = ShiftReg[NBits-1];
endmodule

module TestShifter;
  ...
  defparam U2.NBits = 10;
  Shifter #(16) U1 (...);          //6 位移位寄存器
  Shifter U2 (...)                 //10 位移位寄存器
endmodule
```

参考

更多内容请参考`define、defparam、instantiation、specparam 声明语句的说明。

PATHPULSE $(路径脉冲参数)声明语句

(1) 在指定块中用指定参数(即用 specparam)对 PATHPULSE $参数赋值可控制脉冲的传输。这里所谓的脉冲是指在模块输出端出现的两个跳变沿和它们之间的一段持续时间，其持续时间必须小于信号从模块的输入端直到输出端的延迟。

(2) 如果使用缺省的 PATHPULSE $参数值，仿真器将不考虑脉冲，这就是指因为路径

脉冲的持续时间比模块传输延迟短，故脉冲不能传过该模块，这种效应被称为“延迟惯性”。用指定参数（即用 specparam）可给 PATHPULSE $参数赋新的值。

语法

```
|either|
PATHPULSE$ = (Limit[, Limit]); |(Reject, Error)|
PATHPULSE$Input $Output = (Limit[, Limit]);
Limit = ConstantMinTypMaxExpression
```

在程序中的位置

```
specify -< HERE >- endspecify
```

规则：

（1）如果 PATHPULSE $的第二个极限参数（即 error）没有给定，它就应该与第一个极限参数（即 reject）相同；

（2）维持时间比第一个极限参数（即 reject）短的脉冲不会输出；

（3）维持时间比第一个极限参数（即 reject）长而比第二个极限参数（即 error）短的脉冲将输出一位的不确定值（即 1'bX）；

（4）维持时间比第二个极限参数长的脉冲将正常地输送出去。

（5）用 specparam 对 PATHPULSE $input $output 参数重新赋值将改写常规值；

（6）在同一个模块中可通过使用 specparam 对 PATHPULSE $赋值来描述从输入到输出的延迟。

可综合性问题

综合工具不考虑延迟结构，包括指定块的定义。

示例

```
specify
  (clk => q) = 1.2;
  (rst => q) = 0.8;
  specparam PATHPULSE$clk$q = (0.5,1),
  PATHPULSE = (0.5);
endspecify
```

参考

更多内容请参考 specify、specparam 声明语句的说明。

port（端口）声明语句

模块的端口是硬件器件的引脚或接口的模型。

语法

```
{definition}
{either}
PortExpression {ordered list{
.PortName([PortExpression]) {named list|
PortExpression = }either|
PortReference
{ PortReference,...|
PortReference = |either|
Name
Name[ConstantExpression]
Name[ConstantExpression: ConstantExpression]
}declaration|
{either{
input [Range] Name,...; {of port reference|
output [Range] Name,...; {of port reference|
inout [Range] Name,...; {of port reference|
Range = [ConstantExpression: ConstantExpression]
{在上述部分位选择(即 Range)选项内,冒号左侧常量表达式表示最高位(即 MSB),冒号右侧
常量表达式表示最底位(即 LSB)|
```

在程序中的位置

```
module ( <HERE> ); {definition|
    <HERE> }declaration|
    ...
endmodule
```

规则

(1) 在端口列表中列出的所有端口必须按次序排列或按端口名称排列,这两种排列方式是不同的,不能混合使用。

(2) 有端口的名称但没有端口表达式,如 .A( ),则表示在本模块中定义了不与任何东西相连的端口。

(3) 每个端口除了必须在端口列表中列出外,还必须声明该端口是输出(output)、输入(input)、还是双向端口(inout)。

(4) 每个端口不但要声明是输出、输入、还是双向端口,而且还要声明是连线(wire)还是寄存器(reg)类型,如果没声明,则会隐含地认为该端口是连线类型,且其位宽与相应的端口一致。如果某端口已被声明为一矢量,则其端口的方向和类型两个声明中的位宽必须一致。

(5) 输入和双向端口不能声明为寄存器类型。

(6) 输出端口的类型不能声明为实型(real)或实时型(realtime)。

提示 (1) 在测试模块中不要定义端口。

(2) 在模块定义时不建议使用命名的端口的列表,因为很少有人这样来定义模块端口,大家都不了解这种端口的定义形式。

示例

```
module (A, B[1], C[1:2]);
input A;
input [1:1] B;
output [1:2] C;

module (.A(X), .B(Y[1]), .C(Z[1:2]);
input X;
input [1:1] Y;
output [1:2] Z;
```

参考

更多内容请参考 module、user defined primitive、instantiation 声明语句的说明。

procedural assignment(过程赋值)声明语句

procedural assignment 声明语句用来改变寄存器的值,或者安排以后的变化。

语法

```
{Blocking assignment}                          //阻塞赋值
RegisterLValue = [TimingControl] Expression ;
|Non-blocking assignment|                      //非阻塞赋值
RegisterLValue <= [TimingControl] Expression ;
RegisterLValue = {either}
RegisterName
RegisterName[Expression]
RegisterName[ConstantExpression: ConstantExpression]
Memory[Expression]
{ RegisterLValue,...}
```

在程序中的位置

请参阅 statement 声明语句中的说明。

规则

(1) 对寄存器的赋值(不包括正负号)

（2）对于实型和实时数据类型的寄存器是不允许选择某位和某几位的。

（3）当赋值语句执行时，右侧的表达式被计算出值，但是直到定时控制事件或延迟（也被称为内部指定的延迟）发生后，左侧的表达式才更新。

（4）直到左侧的表达式更新后（例如内部定义的延迟过后）阻塞赋值语句才算完成。在 begin-end 块中，只有当前一条语句执行完后，才能执行其后面的一条语句。在 fork-join 块中，只有当块中所有的阻塞赋值语句结束后，整个块才算结束。

（5）如果仿真时刻相同，要待所有的阻塞赋值语句执行后，非阻塞赋值语句才执行。

```
A <= #5 0;
A = #5 1;          //5 个时间单位后，A 将变为 0，而不是变为 1
```

注意 寄存器变量可以在一个或几个 initial 或 always 声明语句中赋值。无论何时，寄存器变量的值都是由最近的赋值所决定，与事件的来源无关。这一点与 Net 类型的变量不同。Net 可以由两个或更多的源驱动，其结果值则取决于 Net 变量的类型（wire 型、wand 型等）。

可综合性问题

（1）综合工具不考虑延迟；

（2）定时控制或延迟是不可综合的；

（3）同一个寄存器类型变量虽然可以在几个 always 声明语句中赋值，但只有在一个 always 声明语句中赋值的才有可能被综合；

（4）同一个寄存器类型变量不能既用阻塞赋值和非阻塞赋值；

（5）在描述组合逻辑的 always 块中，右侧表达式被综合成组合逻辑，左侧的表达式被综合成连线，如有不完整的赋值则综合成锁存器。在描述时序逻辑的用时钟沿触发的 always 块中，非阻塞赋值符的左侧被综合成触发器，阻塞赋值符的左侧则被综合成一个连接，除非它被用在该 always 块之外，或者在赋值之前它的值已被读取。

提示 （1）通常采用非阻塞赋值语句来生成触发器组成的时序逻辑，而阻塞赋值常用于其他方面，这样做可以防止时钟沿触发的 always 块中发生竞争冒险。这样做也可使设计意图更加清晰，又能避免生成不需要的触发器。

（2）在时钟树的模型已确定的情况下，可用一个简单的内部指定的延迟来避免 RTL 时钟沿对不齐的问题。

示例

```
always @(Inputs)
    begin : CountOnes
      integer I;
      f = 0;
```

```
    for (I =0; I <8; I = I +1)
      if (Inputs[I])
          f = f + 1;
  end
always @ Swap
  fork     //交换 a 和 b 的值
    a = #5 b;
    b = #5 a;
  join     //延迟 5 秒后完成
always @(posedge Clock)
  begin
    c <= b;     //用旧的 b 值
    b <= a;     //b 被 a 值替换
  end
```

用非阻塞赋值语句时加一个延迟来做输出与时钟沿有些偏移的仿真：

```
always @(posedge Clock)
  Count <= #1 Count + 1;
```

在时钟周期的第 5 个下降沿插入复位信号：

```
initial
  begin
    Reset = repeat(5) @(negedge Clock) 1;
    Reset = @(negedge Clock) 0;
  End
```

参考

更多内容请参考 timing control、continuous assignment 声明语句的说明。

procedural continuous assignment(过程连续赋值)声明语句

启动过程连续赋值语句，将给一个或多个寄存器赋值，并同时防止一般的过程赋值语句影响已赋值的寄存器。

语法

```
assign RegisterLValue = Expression ;
deassign RegisterLValue ;
RegisterLValue = {either}
RegisterName
```

```
RegisterName[Expression]
RegisterName[ConstantExpression: ConstantExpression]
MemoryName[Expression]
{ RegisterLValue,...}
```

在程序中的位置

请参阅 statement 声明语句的说明。

规则

(1) 过程连续赋值语句执行后,它会对指定的寄存器(组)强制地维持过程连续赋值直到解除赋值(deassign)语句的执行,或直到另一个 procedural continuous assignment 声明语句又对该寄存器(组)赋值。

(2) 用 force(强制)语句可以改写已由 procedural continuous assignment 声明语句赋值的寄存器类型变量,直到 release 语句的执行,此时强制赋值被解除而原过程连续赋值对该寄存器类型变量的作用又重新恢复。

注意 assign 声明语句与 procedural continuous assignment 声明语句尽管很相似,但并不是完全一致。在编写程序时,确认将 assign 写在正确的位置。procedural continuous assignment 声明语句可以写在声明语句允许出现的位置(在 initial、always、task、function 等块的内部),而 assign 声明语句则必须写在任何 initial 或 always 块之外。

可综合性问题

无论用什么综合工具,procedural continuous assignment 声明语句是都不能综合的。

提示 procedural continuous assignment 声明语句可以用来为异步复位和中断建立仿真模型。

示例

```
always @(posedge Clock)
    Count = Count + 1;              //受下面 always 块控制的计时钟个数的计数器
always @(Reset)                     //异步复位
    if (Reset)
        assign Count = 0;           //当 Reset 为高时,使 Count 为0,不计数
      else
        deassign Count;             //当 Reset 为低时,解除 Count 为0,
                                    //于是下一个时钟的上升沿又重新开始计数
```

参考

更多内容请参考 continuous assignment 和 force 声明语句的说明。

programming language interface(编程语言接口)声明语句

Verilog 编程语言接口 (PLI)为用户提供了在 Verilog 模块中调用用 C 语言编写的函数的手段。这些函数可以动态地访问和修改被引用的 Verilog 数据结构中的数据,用 PLI 编写的系统任务使上述功能变得容易使用。通过调用用户定义的系统任务和函数可以启动 PLI,用户编写自己的 PLI 模块的目的是扩大系统任务和函数的内容。用户自定义的系统任务和函数在调用时都用以$字符开头的任务和函数名。这与 Verilog 语言提供的系统任务和函数库名一致。如用户自定义的系统任务和函数名与原系统任务或函数名相同时,则执行用户自定义的系统任务和函数。

下面列举的是 PLI 在某些方面的应用:

(1) 延迟计数;

(2) 测试矢量读人;

(3) 波形演示;

(4) 源代码调试。

接口模型可用 C 语言或其他语言(例如 VHDL 或硬件建模工具)编写或生成。

对于 PLI 的全面讨论请参考相关书籍。

register(寄存器)声明语句

寄存器可存储在 initial、always、task 和 function 声明语句中所赋的值,广泛地应用在行为建模中。

语法

```
|either|
reg [Range] RegisterOrMemory,...;
integer RegisterOrMemory,...;
time RegisterOrMemory,...;
real RegisterName,...;
realtime RegisterName,...;
RegisterOrMemory = |either|
RegisterName
MemoryName Range
Range = [ConstantExpression: ConstantExpression]
```

在程序中的位置

```
module -< HERE >- endmodule
begin : Label -< HERE >- end
```

```
fork : Label -< HERE >- join
task -< HERE >- endtask
function -< HERE >- endfunction
```

规则

(1) 寄存器类型变量只能用 procedural assignment 声明语句赋值。

(2) 在具体实现时,整数(integer)类型的变量至小用 32 位,时间(time)类型的变量至小用 64 位寄存器。

(3) integer 或 time 类型的寄存器变量与位数相同的 reg 类型的寄存器变量行为是相同的。Integer 和 time 类型的寄存器变量也可像 reg 类型的寄存器变量一样对某位或某些位操作。而在表达式中,整数类型的值被当成有符号值,而 reg、time 类型的值被当成无符号值。

(4) 存储器类型数组中的每个元素作为整体可以进行读或写操作,如果要单独访问数组中某个元素的个别位,则必须先把这个元素的内容复制到某个位数相同的寄存器变量中才能进行。

注意 (1) 虽然 register 这个词指的是硬件寄存器(例如触发器),而寄存器(register)这个名字,在这里是指软件寄存器(即变量)。Verilog 寄存器常用于组合逻辑电路、锁存器、触发器和接口电路的描述和综合。

(2) realtime 类型寄存器变量是 Verilog 语言新增加的变量类型,目前还没有任何工具支持这种类型的变量。

(3) 对于有符号和无符号值的概念,不同版本的 Verilog 和用不同厂家的仿真器时,并不是完全一致的。因此,当使用位宽大于 32 位的有符号数或矢量时要特别注意。

可综合性问题

(1) Real、time 和 realtime 类型的寄存器变量是不可综合的。

(2) 在描述组合逻辑的 always 块中,寄存器被综合成 wire 类型;如果存在不完整赋值的情况,则被综合成锁存器。在描述时序逻辑的 always 块中,寄存器根据块内语句的内容被综合成连线(wire)或者触发器。

(3) 运用目前的综合工具,整数被综合成 32 位,其值用二进制数表示,负数则用其二进制补码表示。

(4) 根据所用语句,存储器数组会被综合成触发器或连线,而不会被综合成 RAM 或 ROM 的器件。

提示 运用 reg 类型变量来描述寄存器逻辑,integer 类型变量用于循环变量和计数,real 类型变量用于系统模块,time 和 realtime 类型变量用于测试模块中记录仿真时刻。

示例

```
reg a, b, c;
```

```
reg [7:0] mem[1:1024], byte;      //byte 不是数组只是一个 8 位的 reg 类型矢量
integer i, j, k;
time now;
real r;
realtime t;
```

下面的程序显示了 reg 类型和 integer 类型变量的一般用法。

```
integer i;
reg [15:0] V;
reg Parity;
    always @(V)
        for (i = 0; i <= 15; i = i + 1 )
          Parity = Parity^V[i];
```

参考

更多内容请参考 Net 声明语句的说明。

repeat(重复执行)声明语句

repeat 声明语句用于把一个或多个声明语句重复地执行指定的次数。

语法

```
repeat (Expression)
    Statement
```

在程序中的位置

参见 statement 声明语句的说明。

规则

重复执行的次数是由表达式的数值所决定的,如果该值为 0、X 或 Z,则不会有重复。

可综合性问题

只有部分综合工具可以综合 repeat 声明语句,而且只有当该循环中的每个循环的分支都被时钟事件,如被@(posedge Clock),所中断时才有可能被综合成电路。

示例

```
initial
    begin
      Clock = 0;
        repeat (MaxClockCycles)
          begin
```

```
            #10 Clock = 1;
            #10 Clock = 0;
        end
    end
```

参考

更多内容请参考 for、forever、while、timing control 声明语句的说明。

reserved words(关键词)声明语句

下列词汇是 Verilog 语言规定的所有的关键词,请注意,千万不要把这些标识符用作自定义的标识符,除非把它们改写为大写的字符或扩展字符。

and	for	output	strong1
always	force	parameter	supply0
assign	forever	pmos	supply1
begin	fork	posedge	table
buf	function	primitive	task
bufif0	highz0	pulldown	tran
bufif1	highz1	pullup	tranif0
case	if	pull0	tranif1
casex	ifnone	pull1	time
casez	initial	rcmos	tri
cmos	inout	real	triand
deassign	input	realtime	trior
default	integer	reg	trireg
defparam	join	release	tri0
disable	large	repeat	tri1
edge	macromokule	nmos	vectored
else	medium	rpmos	wait
end	module	rtran	wand
endcase	nand	rtranif0	weak0
endfunction	negedge	rtranif1	weak1
endprimitive	nor	scalared	while
endmodule	not	small	wire
endspecify	notif0	specify	wor
endtable	notif1	specparam	xnor
endtask	nmos	strength	xor
event	or	strong0	

Specify(指定的块延迟)声明语句

specify 声明语句(指定延迟块)用于描述从模块的输入到输出的路径延迟以及定时约束,例如信号的建立和保持时间。用指定延迟可以在设计时把模块的信号传输延迟与行为或结构分开来进行描述。

语法

```
specify
SpecifyItems...
endspecify
SpecifyItem = |either|
Specparam
PathDeclaration
TaskEnable |Timing checks only|
PathDeclaration = |either|
SimplePath = PathDelay;
EdgeSensitivePath = PathDelay;
StateDependentPath = PathDelay;
SimplePath = |either|
(Input,...[Polarity] * > Output,...)        |full|
(Input [Polarity] = > Output)               |parallel|
EdgeSensitivePath = |either|
([Edge] Input,... * > Output,...[Polarity]: Expression)
([Edge] Input = > Output [Polarity]: Expression)
StateDependentPath = |either|
if (Expression) SimplePath = PathDelay;
if (Expression) EdgeSensitivePath = PathDelay;
ifnone SimplePath = PathDelay;
Input = |either|
InputName
InputName[ConstantExpression]
InputName[ConstantExpression: ConstantExpression]
Output = |either|
OutputName
OutputName[ConstantExpression]
OutputName[ConstantExpression: ConstantExpression]
```

```
Edge = |either| posedge negedge
Polarity = |either|
+
-
PathDelay = |either|
ListOfPathDelays
(ListOfPathDelays)
ListOfPathDelays = |either|
t
t,t                              |Rise,Fall|
t,t,t                            |Rise,Fall,Turn-Off|
t,t,t,t,t,t                      |01,10,0z,z1,1z,z0|
t,t,t,t,t,t,t,t,t,t,t,t          |01,10,0z,z1,1z,z0,0x,x1,1x,x0,xz,zx|
t = MinTypMaxExpression
```

在程序中的位置

```
module -< HERE >- endmodule
```

规则

(1) 路径必须从模块的输入端开始,在该模块的输出端结束,而且在模块内部只能有一个驱动器。

(2) 在路径声明中可使用全连接符(* >)或者并行连接符(= >)来描述。全连接符指所有从输入端到输出端可能的路径,并行连接符指命名的输入端的某些位到命名的输出端的某些位的路径。

(3) 模块路径的极性可选是指着路径可以选择正极性或者负极性,分别指路径是同相的或是反相的(即路径的输入端若是正跳沿,输出端也是正跳沿,则路径是同相的,反之是反相的)。无论选哪一个都不影响仿真,但路径的相位可改变的选项便于时序分析等工具使用。

(4) 跳变沿敏感的路径数据表达式同样也不影响仿真。

(5) 与状态有关的路径延迟(SDPD)表达式只跟端口、常量、局部定义的寄存器或者 Net 类型变量有关。只有部分运算符在 SDPD 表达式中是有效的:如逐位计算符(~ 、& 、| 、^ 、^~ 、~^)、逻辑运算和逻辑等式运算符(== 、!= 、&& 、|| 、!)、缩位运算符(& 、| 、^ 、~& 、~| 、^~ 、~^)、位拼接运算符、重复拼接运算符和条件运算符({} 、{{}} 、?:)。如果条件表达式的值为真(在 SDPD 表达式中 1 、X 、Z 均被认为是真),路径延迟只影响路径。

(6) 如果没有一个 if 条件为真,则用 ifnone 来定义缺省的 SDPD。如果同一条路径既定义为 ifnone 与状态有关的路径延迟(SDPD)又定义为一般简单的路径延迟,则是非法的。

（7）无条件的路径优先于 SDPD 路径。

（8）对于同一条路径，跳变沿敏感的 SDPD 路径声明必须是唯一的，必须用不同的电平或不同的沿（或者两者）。在每条语句中，必须以同样的方式（整个端口、某一位、某些位）来引用输出端的信号。

（9）如果模块的延迟既包括指定的块延迟（specify delays）又包括分布的延迟（即由门、UDP、Net 引起的延迟），应选用最长的延迟作为每条路径的延迟。

可综合性问题

综合工具不能综合指定延迟块。指定延迟块只用于模块的延迟仿真建模。

注意 （1）目前还没有一个仿真工具支持上面语法中列出的那种可用 12 种不同跳变参数表示某条路径延迟的方法；

（2）有关路径目的地的规则比当前许多仿真工具所能支持的灵活。

提示 （1）运用指定延迟块来描述库中的单元（cell）延迟。请注意建立库模型时延迟是怎样计算的。以 PLI（编程语言接口）为基础的延迟计算，需要依据并访问设计中所有单元的指定延迟块中的信息。

（2）可运用指定延迟块来描述“黑匣子”元件的定时特性，但这时还需要借助于支持指定延迟块特性的时序验证工具或综合工具。

示例

```
module M (F, G, Q, Qb, W, A, B, D, V, Clk, Rst, X, Z);
    input A, B, D, Clk, Rst, X;
    input [7:0] V;
    output F, G, Q, Qb, Z;
    output [7:0] W;
    reg C;
        //Functional Description,功能描述
specify
    specparam TLH $Clk $Q = 3,
    THL $Clk $Q = 4,
    TLH $Clk $Qb = 4,
    THL $Clk $Qb = 5,
    Tsetup $Clk $D = 2.0,
    Thold $Clk $D = 1.0;
    //单一路径,全连接
    (A, B * > F) = (1.2:2.3:3.1, 1.4:2.0:3.2);
    //单一路径,并行连接,正极性
```

```
(V + => W) = 3,4,5;
//沿敏感路径,带极性
(posedge Clk *> Q +: D) = (TLH $Clk $Q,THL $Clk $Q);
(posedge Clk *> Qb -: D) = (TLH $Clk $Qb,THL $Clk $Qb);
//电平敏感路径
if (C) (X *> Z) = 5;
if (! C && V == 8'hff) (X *> Z) = 4;
ifnone (X *> Z) = 6; //缺省为 SDPD,从 X(不定值)到 Z(高阻值)
//时序检测
$setuphold(posedge Clk, D,Tsetup $Clk $D, Thold $Clk $D, Err);
        endspecify
    endmodule
```

参考

更多内容请参考 specparam、PATHPULSE $、$setup 等声明的语句的说明。

specparam(延迟参数)声明语句

specparam 声明语句类似于 parameter(参数)声明语句,但只能用在指定延迟块中。

语法

```
specparam Name = ConstantExpression,
Name = ConstantExpression,
...;
```

在程序中的位置

```
specify -<HERE>- endspecify
```

规则

(1) specify 块中的常量表达式可以用数字和 specparam 来定义,但不能用参数(parameter)来定义,specparam 不能用在 specify 块(即指定延迟块)外;

(2) 利用 defparam 或在块的实例引用时使用#,可以改写用 specparam 定义的延迟参数值,用编程语言接口(PLI)也可以修改其值。

提示:(1) 在 specify 块中,用 specparam 来定义命名的延迟参数比直接用数字要好;

(2) 这些延迟参数应有一个命名的规则,这样便于对它们进行修改,如果有必要的话,也可以采用 PLI 的延迟计数来进行修改。

示例

```
specify
```

```
specparam    tRise$a$f = 1.0,
             tFall$a$f = 1.0,
             tRise$b$f = 1.0,
             tFall$b$f = 1.0;
             (a *> f) = (tRise$a$f, tFall$a$f);
             (b *> f) = (tRise$b$f, tFall$b$f);
endspecify
```

参考

更多内容请参考 PATHPULSE $、specify 声明语句的说明。

statement 声明语句

运用声明语句可以描述硬件模块的行为。声明语句在定时控制的(延迟、控制程序、等待)时刻执行。若两个或两个以上的语句是一起的,必须把它们写在 begin-end 或 fork-join 块中。在 begin-end 块中每条语句是顺序执行的,在 fork-join 块中,它们是并行执行的。initial 或 always 块中的语句同其他 initial 或 always 块中的语句是同时执行的。

语法

```
{either|
;                          |Null statement|
TimingControl Statement    |Statement may be Null|
Begin
Fork
ProceduralAssignment
ProceduralContinuousAssignment
Force
If
Case
For
Forever
Repeat
While
Disable
-> EventName;  |Event trigger|
TaskEnable
```

在程序中的位置

```
initial -<HERE>
```

```
always -< HERE >
begin -< HERE >- end
fork -< HERE >- join
task -< HERE >- endtask |Null allowed|
function -< HERE >- endfunction
if() -< HERE >- else -< HERE > |Null allowed|
case - label: -< HERE >- endcase |Null allowed|
for( < HERE > ) -< HERE >
forever -< HERE >
repeat() -< HERE >
while() -< HERE >
```

参考

更多内容请参考 timing control 声明语句的说明。

strength(强度)声明语句

除了逻辑值外,Net 类型的变量还可以定义强度,因而可以更精确地建模。Net 的强度来自于动态 Net 驱动器的强度。在开关级仿真时,当 Net 由多个驱动器驱动且其值互相矛盾时,常用强度(Strength)的概念来描述这种逻辑行为。

语法

```
|either|
(Strength0, Strength1)
(Strength1, Strength0)
(Strength0)              |pulldown primitives only|
(Strength1)              |pullup primitives only|
(ChargeStrength)         |trireg nets only|
Strength0 = |either|
supply0
strong0
pull0
weak0
highz0
Strength1 = |either|
supply1
strong1
pull1
```

```
weak1
highz1
ChargeStrength = |either|
large
medium
small
```

在程序中的位置

请参照 Net、instantiation、continuous assignment 声明语句的说明。

规则

(1) 关键词 Strength0 和 Strength1 用于定义 Net 的驱动器强度。其中 Strength 表示强度,与紧跟着的 0 和 1 连起来分别表示输出逻辑值为 0 和 1 时的强度。

(2) 在强度声明中可选择不同的强度关键字来代替 strength,但(highz0, highz1) 和(highz1, highz0)这两种强度定义是不允许的,在 pullup (上拉)和 pulldown(下拉) 门的强度声明中 highz0 和 highz1 是不允许的。

(3) 默认的强度定义为 (strong0, strong1),但下述情况除外:

① 对于 pullup 和 pulldown 门,默认强度分别为(pull1) 和 (pull0)。

② 对于 trireg 的 Net,默认强度为 medium。

③ 强度定义为 supply0 和 supply1 的 Net,总是能提供强度。

(4) 在仿真期间,Net 的强度来自于 Net 上的主驱动强度(即具有最大强度值的实例或 assign 声明语句)。如果 Net 未被驱动,它会呈现高阻值,但以下情况除外:

① tri0 和 tri1 类型的 Net 分别具有逻辑值 0 和 1,并为 pull 强度。

② trireg 类型的 Net 保持它们最后的驱动值。

③ 强度为 supply0 和 supply1 的 Net 分别具有逻辑值 0 和 1,并能提供驱动能力。

(5) 强度值有强弱顺序,可从 supply(最强的)依次减弱排列到 highz(最弱的),当需要确定 Net 的确实逻辑值和强度时,或者当 Net 由多个驱动器驱动而且驱动相互间出现冲突时,出现冲突的两个强度值在如下强弱顺序表中的相对位置就会对该 Net 的真实逻辑值起作用。

supply
strong
pull
large
weak
medium
small
highz

可综合性问题

strength 声明语句不可综合。

提示 可以在$display 和$monitor 等中用特定的格式控制符 %V 显示其强度值。

示例

```
assign (weak1,weak0) f = a + b;
trireg (large ) c1,c2;
and (strong1,weak0) u1(x,y,z);
```

参考

更多内容请参考 continous assignment、instantiation、Net、$display 声明语句的说明。

string(字符串)声明语句

字符串能够用在系统任务(诸如$display 和$monitor 等)中作为变量,字符串的值可以像数字一样存储在寄存器中,也可以像对数字一样对字符串进行赋值、比较和拼接。

语法

见 string 说明。

在程序中的位置

请参见 expression 声明语句的说明。

规则

(1) 一条字符串不能占原代码的多行。

(2) 字符串可以包含下列扩展字符。

\n	换行
\t	Tab 符
\	反斜杠字符\
\"	双引号字符"
\nn	八进制的 ASCII 字符
* %%	百分号%

(3) 诸如$display 和$monitor 等的系统任务中的打印字符串可以包含特殊的格式控制符(如%b)(参见$display 的说明)。

(4) 当字符串存储于寄存器中,每个字符要占 8 位,字符以 ASCII 代码形式存储。Verilog HDL 语言的字符串的定义和 C 语言的不一样。在 C 语言中需要用,而在 Verilog HDL 语

言中不需要用 ASCII 代码的 0 字符来表示字符串的结束。

注意 在表达式中使用字符串时,请注意填加物。对字符串的处理跟对数字的处理方式一样,当字符所占的位数少于寄存器的数目时,则在字符串的左边的寄存器中填加 0。

示例

```
reg [23:0] MonthName[1:12];
initial
begin
  MonthName[1] = "Jan";
  MonthName[2] = "Feb";
  MonthName[3] = "Mar";
  MonthName[4] = "Apr";
  MonthName[5] = "May";
  MonthName[6] = "Jun";
  MonthName[7] = "Jul";
  MonthName[8] = "Aug";
  MonthName[9] = "Sep";
  MonthName[10] = "Oct";
  MonthName[11] = "Nov";
  MonthName[12] = "Dec";
end
```

参考

更多内容请参考 number, $display 声明语句的说明。

task(任务)声明语句

任务常用于把模块代码分割成由若干声明语句构成的较大的块,便于模块代码的理解和维护,也可以从模块代码的不同位置执行这样一个常见的顺序声明语句块。

语法

```
task TaskName;
[Declarations...]
Statement
endtask
Declaration = {either}
input[Range]Name,...;
output [Range] Name,...;
inout [Range] Name,...;
```

```
Register
Parameter
Event
Range =[ConstantExpression: ConstantExpression]
```

规则

(1) 若用于任务中的命名变量或参数没有在任务块中声明,则指的是在模块中声明的命名变量或参数。

(2) 任务中的 input、output 和 inout 的个数不受限制(也可以为零个)。

(3) 任务中的变量(包括输入和双向端口(inout))可以声明为寄存器型。如果没有明确地声明,则默认为寄存器型,且其位宽与相应的变量匹配。

(4) 当启动任务时,相应于任务的输入和双向端口的变量表达式的值被存入相应的变量寄存器中。当任务结束时,输入和双向端口的变量寄存器中的值又被代入启动任务的语句中相应的表达式。

注意 (1) 和模块的端口定义不一样,任务的变量不能在任务名后的括号中定义。

(2) 任务中若包括一句以上的语句,必须要用 begin-end 或 fork-join 将其包含成块。

(3) 任务的输入、双向端口、输出和局部寄存器的值都是静态存储的,也就是说即使多次启动任务,也只有一份这些寄存器的拷贝。若第一次启动的任务还未完成,便第二次启动该任务,其输入、双向端口、输出和局部寄存器的值便会被覆盖。

(4) 当被启动的任务运行结束时,输出和双向端口的值被代入任务中相应的寄存器表达式。如果任务中的输出和双向端口在赋值后有时间的控制,则相应的寄存器只能在时序控制延迟后才被更新。

(5) 同样,对输出和双向端口寄存器变量的非阻塞赋值语句也不会起作用,因为当任务返回时,赋值语句可能还未生效。

可综合性问题

包含时序控制语句的任务是不可综合的。启动的任务往往被综合成组合逻辑。

提示 (1) 复杂 RTL 模块通常需要用多个 always 块来构造。建议最好不要采用一个 always 块运行多个任务的方案。

(2) 在测试块中可用任务来产生重复的激励序列。例如,对存储器的数据读写(见例)序列。

(3) 某任务如果被多个模块引用,可以把它定义为一个独立的模块(只包括该任务),并可用层次命名来引用它。

示例

下面这个例子表示一个简单的可以综合的 RTL 任务。

```
task Counter;
    inout [3:0] Count;
    input Reset;
      if (Reset)          //同步复位
          Count = 0;      //对 RTL 必须用非阻塞方式赋值
      else
          Count = Count + 1;
endtask
```

下面这个例子说明如何在测试模块中运用任务。

```
module TestRAM;

parameter AddrWidth = 5;
parameter DataWidth = 8;
parameter MaxAddr = 1 << AddrBits;

reg [DataWidth-1:0] Addr;
reg [AddrWidth-1:0] Data;
wire [DataWidth-1:0] DataBus = Data;
reg Ce, Read, Write;

Ram32x8 Uut (.Ce(Ce), .Rd(Read), .Wr(Write),
             .Data(DataBus), .Addr(Addr));
initial
begin : stimulus
  integer NErrors;
  integer i;
  //错误开始记数
  NErrors = 0;
  //为每个地址写上地址值
for (i=0; i<=MaxAddr; i=i+1)
  WriteRam(i, i);
  //读且比较
for (i=0; i<=MaxAddr; i=i+1)
begin
  ReadRam(i, Data);
  if (Data !== i)
    RamError(i,i,Data);
end
    //小结错误个数
```

```
        $display(Completed with % 0d errors, NErrors);
    end
task WriteRam;
    input [AddrWidth-1:0] Address;
    input [DataWidth-1:0] RamData;
    begin
        Ce = 0;
        Addr = Address;
        Data = RamData;
        #10 Write = 1;
        #10 Write = 0;
        Ce = 1;
    end
endtask
task ReadRam;
    input [AddrWidth-1:0] Address;
    output [DataWidth-1:0] RamData;
    begin
        Ce = 0;
        Addr = Address;
        Data = RamData;
        Read = 1;
        #10 RamData = DataBus;
        Read = 0;
        Ce = 1;
    end
endtask
task RamError;
    input [AddrWidth-1:0] Address;
    input [DataWidth-1:0] Expected;
    input [DataWidth-1:0] Actual;
    if (Expected !== Actual )
    begin
        $display("Error reading address % h", Address);
        $display(" Actual % b, Expected % b", Actual,
                    Expected);
```

```
        NErrors = NErrors + 1;
      end
    endtask
  endmodule
```

参考

更多内容请参考 task enable、function 声明语句的说明。

task enable(任务的启动)声明语句

在模块代码中只需用任务名便可启动任务。当任务启动时,输入值通过任务的端口变量(输入和 inout 变量)传递到任务中。当任务结束时,返回值通过任务的端口寄存器变量(输出和 inout 变量)传出。

语法

```
TaskName[(Expression,...)];
```

规则

(1) 任务可以从 initial 或 always 块或其他任务中启动。任务可以多次调用。但任务不能被函数调用。

(2) 调用任务的语句中,端口表达式的顺序和任务端口变量声明的顺序必须一致。端口的个数必须与任务声明的端口变量的个数一致。

(3) 若任务的端口变量是输入时,则对应的端口变量可以是任何一种表达式;若端口变量为输出和 inout 时,对应的端口变量必须位于 procedural assignment 声明语句的左边而且必须是有效的。

(4) 当任务启动时,输入和 inout 表达式复制到相应的变量寄存器中。当任务结束时,输出和 inout 寄存器的值会复制到启动任务相应的端口寄存器中。

(5) 可以在任务内部或任务外部把任务禁止(disable)。

注意 任务中变量寄存器默认为静态的,所以当一个任务正在执行时又启动该任务时,输入和 inout 寄存器的值会被覆盖。

可综合性问题

若任务不包含时序控制,是有可能被综合的。调用的任务往往被综合成组合逻辑。

举例说明

```
task Counter;
  inout [3:0] Count;
  input Reset;
```

```
    ...
  endtask

always @(posedge Clock)
  Counter(Count, Reset);
```

参考

更多内容请参考 disable、task、function call 声明语句的说明。

timing control(时序控制)声明语句

timing control 声明语句用于延迟语句的执行或按排语句的执行顺序。时序控制可以放在语句的前面,或者在程序的 procedural assignment 声明语句表达式中的赋值操作符(即 = 或 <=)之间。前一种延迟语句的执行,后一种延迟声明的语句生效。

语法

```
|Timing controls before statements|
{either}
DelayControl
EventControl
WaitControl
|Intra-assignment Timing controls|
{either}
DelayControl
EventControl
repeat (Expression) EventControl
DelayControl = {either}
# UnsignedNumber
# ParameterName
# ConstantMinTypMaxExpression
# (MinTypMaxExpression)
EventControl = {either}
@ Name {of Register, Net or Event}
@ (EventExpression)
EventExpression = {either}
Expression
Name {of Register, Net or Event}
posedge Expression {01, 0X, 0Z, X1 or Z1}
negedge Expression {10, 1X, 1Z, Z0 or X0}
```

```
EventExpression or EventExpression
WaitControl = wait (Expression)
```

在程序中的位置

请参阅 statement、procedural assignment (for intra-assignment timing control) 声明语句的说明。

规则

(1) 在某声明语句前面插入的事件或延迟控制使原本立刻要执行的该条语句延迟执行。

(2) 当执行到 wait 时,如果其表达式为假(0 或 X),wait 控制只延迟 wait 语句后的下一条语句;当表达式为真(非 0)时,下一条语句才执行。当执行到 wait 时,如果表达式为真,下一句不延迟马上执行。

(3) 执行 procedural assignment 声明语句时,要检查赋值语句右边的表达式,如果没有内部赋值延迟,若用的是阻塞赋值,则左边的寄存器类型变量立即更新,若用的是非阻塞赋值,则在下一个仿真周期更新。如果有内部赋值延迟,左边的寄存器类型变量只有在发生内部赋值延迟后才更新。

(4) 内部赋值延迟必须是常数的,但语句前的延迟可以是常数或变量(即 Net 或 reg 类型变量)。

(5) or 列表中的任何一个信号(事件)变化(发生)时,即触发事件控制。

(6) 对于 posedge(上升沿)和 negedge(下降沿)事件触发控制,只测试表达式的最低位。要不然的话,表达式的任何变化都会触发事件。

注意 对于阻塞赋值语句而言,指定内部赋值延迟为零(#0)与不指定是不一样的,也与没有赋值延迟的非阻塞赋值语句不同。对于阻塞赋值语句而言,指定#0 意味着该语句在所有待定事件完成以后,而在非阻塞赋值完成以前进行(不指定内部赋值延迟和指定内部赋值延迟为零(#0)的赋值语句是一样的。)

可综合性问题

(1) 综合时延迟被忽略;

(2) 综合工具不支持 wait 语句和内部赋值延迟以及 repeat(重复)语句;

(3) 事件控制用于控制 always 块的执行,从而能确定综合出的逻辑是组合的还是时序的。一般情况下,always 块后紧跟着的就是事件控制,这有时也称为敏感列表。

提示 在用 RTL(寄存器传输级 HDL 语言)描述电路时,可用内部赋值延迟可来描述在给表示触发器的寄存器变量赋值时的时钟偏移现象。

示例

```
#10
```

```
#(Period/2)
#(1.2:3.5:7.1)
@Trigger
@(a or b or c)
@(posedge clock or negedge reset)
wait (! Reset)
```

非阻塞赋值时使用延迟来克服时钟的偏移：

```
always @(posedge Clock)
        Count <= #1 Count + 1;
```

在周期时钟的第 5 个下降沿复位：

```
initial
   begin
      Reset = repeat(5) @(negedge Clock) 1;
      Reset = @(negedge clock) 0;
   end
```

参考

更多内容请参考 procedural assignment、always、repeat 声明语句的说明。

user defined primitive(用户自定义原语)声明语句

用户自定义原语(UDPS)可以为小型元件建立模型,这也是模块的另一种表示方法。可以用与引用由门构建的实例同样的方式来实例引用用户自定义原语(UDPS)。

语法

```
primitive UDPName (OutputName, InputName,...);
   UDPPortDeclarations ...
   UDPBody
endprimitive
UDPPortDeclaration = {either}
   output OutputName;
   input InputName,...;
   reg OutputName; {Sequential UDP}
UDPBody = {either} CombinationalBody SequentialBody
CombinationalBody =
   table
     CombinationalEntry...
```

```
    endtable
SequentialBody =
    [initial OutputName = InitialValue;]
    table
      SequentialEntry...
    endtable
InitialValue =
    |either| 0 1 1'b0 1'b1 1'bx |not case sensitive|
CombinationalEntry = LevelInputList: OutputSymbol;
SequentialEntry =
    SequentialInputList : CurrentOutput : NextOutput;
SequentialInputList = |either|
    LevelInputList
    EdgeInputList
LevelInputList = LevelSymbol...
EdgeInputList =
    [LevelSymbol...]EdgeIndicator[LevelSymbol...]
CurrentOutput = LevelSymbol
NextOutput = |either| OutputSymbol
EdgeIndicator = |either|
    (LevelSymbol LevelSymbol)
    EdgeSymbol
OutputSymbol = |either| 0 1 x |not case sensitive|
LevelSymbol = |either| 0 1 x ? b |not case sensitive|
EdgeSymbol = |either| r f p n * |not case sensitive|
```

规则

(1) UDP 只允许有一个输出端,至少允许有一个输入端。具体实施时,对输入端的个数是有限制的,但必须至少允许 10 个输入端口。

(2) 如果某 UDP 的输出端定义为 reg 型(寄存器类型)变量,则该 UDP 是时序逻辑的 UDP,否则为组合逻辑的 UDP。

(3) 如果已对时序逻辑的 UDP 的输出进行了初始化,则只有待到在仿真开始时,初始值才开始从引用的原语实例的输出传出。

(4) 描述时序逻辑的 UDPS 可以是电平敏感的或边沿敏感的。若在真值表中有边沿敏感的指示(至少一个),则该描述时序逻辑的 UDP 为边沿敏感的。

(5) UDP 的行为在表中定义。表的行定义为不同输入条件下的输出。对于描述组合逻辑的 UDP,每一行定义为一个或多个输入的组合逻辑的输出。对于描述时序逻辑的 UDP,每

一行都要考虑 reg 类型变量的当前输出值。一行最多只能有一个边沿变化入口。行定义了在指定的边沿发生变化时，由输入值和当前输出值所产生的输出值。

（6）UDP 表中所用的特殊的电平和边沿符号含义如下：

?	0、1 或 x
b or B	0 或 1
-	输出不变
(vw)	由 v 变为 w
r or R	(01)
f of F	(10)
p or P	(01)(0x)或(x1)
n or N	(10)(1x)或(x0)
*	(??)

（7）若组合逻辑的输入值和触发边沿没有明确指定将会导致输出的不确定。

（8）不支持 Z 值。输入时 Z 看成是 X；输出值不允许设为 X。注意 ? 符号的特殊含义，它和在数字中的 ? 符号意思不一样，在数字的表示中符号? 和 Z 含义相同。

注意　在描述时序逻辑的 UDP 中，若在表中任何地方出现边沿触发的条件，则输入信号所有可能的边沿都要认真考虑并列出，因为默认的只是一种边沿的触发的条件，这将导致输出的不确定性。

可综合性问题

任何一种工具都不能综合 UDP，它只被用来建立基本的门级逻辑器件的逻辑仿真模型。

提示　（1）和行为模块相比较，用 UDP 来做仿真非常有效。为 ASIC 单元库的元件建立模型时应该使用 UDP。

（2）输出端口在一个以上的元件，应该对每个输出建立独立的 UDP。

（3）在表的第一行加上注释，指明每一列的含义。

示例

```
primitive Mux2to1 (f, a, b, sel);      //组合 UDP
    output f;
    input a, b, sel;
    table
    //a b sel : f
    0 ? 0 : 0;
    1 ? 0 : 1;
    ? 0 1 : 0;
    ? 1 1 : 1;
```

```
    0 0 ? : 0;
    1 1 ? : 1;
  endtable
endprimitive

primitive Latch (Q, D, Ena);
    output Q;
    input D, Ena;
    reg Q; //Level sensitive UDP
    table
    //D Ena : old Q : Q
    0 0 : ? : 0;
    1 0 : ? : 1;
    ? 1 : ? : -;      //保持原先值
    0 ? : 0 : 0;
    1 ? : 1 : 1;
  endtable
endprimitive

primitive DFF (Q, Clk, D);
    output Q;
    input Clk, D;
    reg Q; //Edge sensitive UDP
    initial
      Q = 1;
      table
        //Clk D : old Q : Q
          r 0 : ? : 0;          //Clock '0'
          r 1 : ? : 1;          //Clock '1'
          (0?) 0 : 0 : -;       //Possible Clock
          (0?) 1 : 1 : -;       //" "
          (? 1) 0 : 0 : -;      //" "
          (? 1) 1 : 1 : -;      //" "
          (? 0) ? : ? : -;      //Ignore falling clock
          (1?) ? : ? : -;       //" " "
          ? * : - : -;          //Ignore changes on D
      endtable
endprimitive
```

参考

更多内容请参考 module、gate、instantiation 声明语句的说明。

while(条件循环)声明语句

只要控制表达式为真(即不为零),循环语句就重复进行。

语法

```
wile ⟨Expression⟨
    Statement
```

可综合性问题

只有当循环块有事件控制(即@(posedge Clock))才可综合。

示例

```
reg [15:0] Word
while (Word)
begin
    if (Word[0])
      CountOnes = CountOnes + 1;
    Word = Word >>1;
end
```

参考

更多内容请参考 for、forever、repeat 声明语句的说明。

七、编译器指示(Compiler Directives)

编译器指示是在源代码中对 Verilog 编译器所发出的指令。在编译指示需要用反引号(`)做前导。编译器指示从它在源代码出现的地方开始生效,并一直继续生效到随后运行的所有文件,直到编译器指示结束的地方或一直运行的最后的文件。

下面有 Verilog 编译指示的摘要。摘要后面详细介绍了一些比较重要的编译指示。

注意 编译器指示的生效依赖于编译时源代码中所包含文件的执行顺序。

标准的编译器指示(Standard Compiler Directives)

在 Verilog LRM 中定义了以下编译器指示:

1. `celldefine 和`endcelldefine

`celldefine 和`endcelldefine 可用来作为分别加在模块的前面和后面的标记,以表示该模

块是一个库单元(Cell)。单元可被 PLI 子程序调用来做某种应用,比如延迟的计算。

例如:

```
`celldefine
module Nand2 {…};
...
endmodule
`endcelldefine
```

2. `default_nettype

改变 Net 类型的默认类型。如果没有该声明,默认的 Net 类型是 wire 型。

例如:

```
`default_nettype tril
```

3. `define 和`undef

`define 定义一个文本宏,`undef 取消已定义的文本宏定义。

在编译的第一阶段期间,宏(Macro)被它所定义的文本字符串取代。宏也可以用来控制条件编译(请参阅`ifdef)。关于`define 应用的更多细节见下面的说明。

4. `ifdef、`else 和`endif

`ifdef、`else 和`endif 根据是否定义了特殊的宏,来指示编译器是否要编译这一段 Verilog 源代码。详细细节见下面的说明。

5. `include

指示编译器读入包含文件的内容,并在`include 所在的地方编译该文件。

例如:

```
`include "definitions.v"
```

6. `resetall

`resetall 把现行的已启动的所有编译器指示复位到原默认值。该编译指示可以写在每个 Verilog 源文件的第一行,以防止前面别的源文件的编译指示在该源文件编译时产生不需要的结果。

例如:

```
`resetall
```

7. `timescale

`timescale 定义仿真的时间单位和精度。细节请见下面的说明。

8. `unconnected_drive 和`nounconnected_drive

`unconnected_drive 编译指示把模块没连接的输入端口设置为上拉 pull up(pull1,即逻辑 1)或为下拉 pull down(pull0, 即逻辑 0)。`nounconnected_drive 编译指示把模块没连接输入端口的设置恢复到默认值,即把没连接的输入端口值设置为高阻浮动(Z)。

例如:

```
`unconnected_drive pull0     //或 pull1 (即逻辑值为 1)
```

非标准编译器指示(Non-Standard Compiler Directives)

下面的编译指示并不属于 Verilog HDL 语言的 IEEE 标准。但在 Cadene 公司的 Verilog LRM 中提及。并不是所有的 Verilog 工具都支持以下这些编译指示。

1. `default_decay_time

若未明确给定衰减时间,则由该编译指示将其设置为默认的三态寄存器(trireg) 类型的线路连接(Net)的衰减时间。

例如:

```
`default_decay_time 50
`default_decay_time infinite     //表示无衰减时间
```

2. `default_trireg_strength

把三态寄存器(trireg) 类型的线路连接(Net)的默认强度设置为整数。用整数来表示强度并不符合 IEEE 规定的 Verilog 语言标准,但仍属于 Verilog 语言非标准扩展部分。

例如:

```
`default_trireg_strength 30
```

3. `delay_mode_distribute、`delay_mode_path、`delay_mode_unit 和`delay_mode_zero

这些编译指示都会影响延迟的仿真方式。分布式延迟是在原语实例中的延迟、赋值延迟和线路连接延迟。路径延迟是在 specify(指定)块中定义的延迟。若用单位和零延迟代替分布式延迟和路径延迟将加快仿真的过程,但会丢失真实的延迟信息。在默认情况下,仿真器会自动选择最长的延迟仿真方式,即分布式延迟和路径延迟仿真方式。

4. `define

`define 定义一个文本宏。宏在编译的第一阶段被由它定义的文本所代替。在用参数和函数表达不适合或不允许的情况下,用宏可以提高 Verilog 源代码的可读性和可维护性。

语法

```
`declaration
```

```
`define Name[(Argument,…)] Text
|usage|
`Name [(Expression,…)]
```

在程序中的位置

宏可以在模块内或模块外定义。

规则

(1) 像所有的编译指示一样,宏定义在整个文件中生效,除非被后面的`define、`undef 和`resetall 编译指示改写或清除。宏定义没有范围的限制。

(2) 若定义的宏内有参数,即在宏文本中用到参数,则当宏调用时,宏的参数被实际的参数表达式所代替。

```
`define add(a , b) a + b
        t = `add(1, 2);      //t = 1 + 2;
```

(3) 宏定义可以用反斜杠(\)跨越几行。新的一行是宏文本中的一部分。

(4) 宏文本不允许分下列语言记号:注释、数字、字符串、名称、保留名称、操作符。

(5) 不能把编译器指示名用作宏名。

注意 (1) 所有的具体电路实现工具都不支持带参数的宏。

(2) 若定义了宏,则必须把撇号(`)写在宏名的紧前面才能调用该宏。没有撇号(`)打头的名,即使名称与宏名一致,则为独立的标识符与宏定义无关。

(3) 要区别撇号(`)和表示数制的前引号(')的不同。

(4) 不要用分号来结束宏定义,除非真要在用宏代替分号,否则会引起语法错误。**提示** (1) 通常更喜欢用参数而不是用宏给无含义的字符起一个有含义的名字。

(2) 仿真时,用带参数的宏要比用同样功能的函数效率高。

示例

本例子说明在分层设计中如何用文本宏来选择不同的模块实现。这在综合时很有用,特别是当必须用 RTL 源代码模块和已综合成门级电路的模块做混合仿真时。

```
`define SUBBLOCK1 subblock1_rtl
`define SUBBLOCK2 subblock2_rtl
`define SUBBLOCK3 subblock3_gates
module TopLevel …
    `SUBBLOCK1 sub1_inst (…);
    `SUBBLOCK2 sub2_inst(…);
    `SUBBLOCK3 sub3_inst(…);
…
```

```
endmodule
```

下面的例子说明带参数的文本宏的定义和调用：

```
`define nand (delay) nand #(delay)
nand(3) (f,a,b);
nand(4) (g,f,c);
```

参考

更多内容请参考‘ifdef 的说明。

5. `ifdef

根据是否定义了特定的宏，来决定是否编译这部分 Verilog 源代码。

语法

```
`ifdef MacroName
  VerilogCode…
[`else
   VerilogCode…]
`endif
```

规则

(1) 如果宏名已经用`define 定义，只编译 Verilog 编码的第一块；

(2) 如果宏名没有定义和`else 指示出现，只编译第二块；

(3) 这些编译指示是可以嵌套的；

(4) 没被编译的代码仍然必须是有效的 Verilog 代码。

提示 这些编译指示可以用来调试模块。例如，可以在同一个模块的两种形式之间切换(如布线前仿真模块和带布线延迟的门级仿真模块之间)或有选择地开启诊断信息的打印输出。

示例

```
`define primitiveModel
module Test
…
`ifdef primitiveModel
 Mydesign_primitives UUT (…);
`else
 Mydesign_RTL UUT(…);
`endif
endmodule
```

参考

更多内容请参考`define 的说明。

6. `timescale

`timescale 用来定义时间单位和仿真精度。

语法

```
`timescale TimeUnit /PrecisionUnit
TimeUnit = Time Unit
PrecisionUnit = Time Unit
Time = {either} 1 10 100
Unit = {either} s ms us ns ps fs
```

规则

(1) 像所有的编译指示一样,`timescale 影响在该指示后的所有模块,无论位于同一个文件的还是位于独立编译的多个文件中的模块,直到碰到下一个`timescale 或`resetall 指示将其改写或复位到默认为止。

(2) 精度单位必须小于或等于时间单位。

(3) 仿真器运行的精度就是在`timescale 指示中所定义的最小精度单位。所有的延迟时间都以精度单位为准取整。

提示 在每个模块文件的第一句应写上`timescale 指示,即使在模块中没有延迟,也是如此,因为有的仿真器必需要有`timescale 指示才能正常工作。

示例

```
`timescale 10ns /1ps
```

参考

更多内容请参考$timeformat 的说明。

八、系统任务和函数(System Task and Function)

Verilog 语言包含一些很有用的系统命令和函数。用户可以像自己定义的函数和任务一样调用它们。所有符合 IEEE 标准的 Verilog 工具中一定都会有这些系统命令和函数。Cadence 公司的 Verilog 工具中还有另外一些常用的系统任务和函数,它们虽并不是标准的一部分,但在一些仿真工具中也经常见到。

请注意,各种不同的 Veriog 仿真工具可能还会加入一些厂商自己特色的系统任务和函数。用户也可以通过编程语言接口(PLI)把用户自定义的系统任务和函数加进去,以便于仿

真和调试。

所有的系统任务和系统函数的名称(包括用户自定义的系统任务),前面都要加$以区别于普通的任务和函数。下面是 Verilog 工具中常用的系统任务和函数的摘要。详细资料在后面介绍。

标准的系统任务和函数

Verilog HDL 的 IEEE 标准中包括下面的系统任务和函数。

$display、$monitor、$strobe、$write 等

用于把文本送到标准输出和或写入一个或多个文件中的系统任务。详细资料在后面介绍。

$fopen 和$fclose

```
$fopen("FileName");        |Return an integer|
$fclose(Mcd);
```

$fopen 是一个系统函数,它可以打开文件为写文件做准备。而$fclose 也是一个系统函数,它关闭由 $fopen 打开的文件。详细说明在后面介绍。

$readmemb 和$readmemh

```
$readmemb("File", MemoryName [,StartAddr[,FinishAddr]]);
$readmemh("File", MemoryName [,StartAddr[,FinishAddr]]);
```

把文本文件中的数据赋值到存储器中。详细资料在后面介绍。

```
$timeformat[(Units,Precision,Suffix,MinFieldWidth)];
```

定义用$display 等显示仿真时间的格式。详细资料在后面介绍。

$printtimescale

```
$printtimescale([ModuleInstanceName]);
```

以如下格式显示一个模块的时间单位和精度:

```
Time scale of (module_name) is unit /precision
```

如果没有参数,则显示模块的时间单位和精度。

$stop

```
$stop[(N)];          |N is 0,1,2|
```

暂停仿真。可选的参数决定诊断输出的类型。0 输出最少,1 输出多点,2 输出最多。

$finish

```
$finish[(N)];        |N is 0,1,2|
```

退出仿真,把控制权返回给操作系统。如果给出参数 *N*,则根据 *N* 值打印不同的诊断信息,见下面的解释:

0——不打印。

1——打印仿真时间和地点(默认值)。

2——打印仿真时间和地点与仿真所使用的 CPU 时间和内存的统计数据。

$time,$stime,和$realtime

```
$time;
$stime;
$realtime;
```

系统函数返回仿真的当前时间值。返回时间值的单位由调用该系统函数语句的模块的`timescale定义。

(1)$time 返回一个根据时间单位四舍五入取整的 64 位无符号整数;

(2)$stime 返回一个截去高位保留低 32 位的无符号整数;

(3)$realtime 返回一个实数。

请注意,这些系统函数没有输入,与 Verilog 的其他函数不同。

$realtobits 和$bitstoreal

```
$realtobits(RealExpression)     |return a 64 bit value|
$bitstoreal(BitValueExpression)     |return a real value|
```

实数和用位(bit)表示的数之间的互相转换。因为模块的端口不允许传输实数,故需要把实数转换为用位表示的数后才能输入/输出模块。请参阅 module 声明语句的说明。

$rtoi 和$itor

```
$rtoi(RealExpression)       |return an integer|
$itor(IntegerExpression)    |rerurn a real number|
```

实数和整数之间的互相转换。$rtoi 把实数截断后转换为整数。$itor 把整数转换为实数。

随机数产生函数

```
$random[(Seed)];
$dist_chi_square(Seed, DegreeOfFreedom);
$dist_erlang(Seed, K_ stage, Mean);
$dist_exponential(Seed, Mean);
$dist_normal(Seed, Mean, StandardDeviation);
$dist_poisson(Seed, Mean);
```

```
$dist_t(Seed, DegreeOfFreedom);
$dist_uniform(Seed, Start, End);
```

当重复调用上述函数时,根据不同概率分布的随机数产生函数条件返回其相应的随机数序列。若伪随机序列的源种相同,则伪随机序列也总是一样的。请参考关于概率与统计理论的图书,详细了解其中分布函数及其应用部分。

Specify Block Timing Checks(**指定块内的时序检查系统任务**)

```
$hold(ReferenceEvent, DataEvent, Limit [, Notifier]);
$nochange(ReferenceEvent, DataEvent, StartEdgeOffset, EndEdgeOffset [,
        Notifier]);
$period(ReferenceEvent, Limit [, Notifier]);
$recovery(ReferenceEvent, DataEvent, Limit [, Notifier]);
$setup(DataEvent, ReferenceEvent, Limit [, Notifier]);
$setuphold(ReferenceEvent, DataEvent,SetupLimit, HoldLimit [, Notifier]);
$skew(ReferenceEvent, DataEvent, Limit [, Notifier]);
$width(ReferenceEvent, Limit [, Threshold [, Notifier]]);
```

以上 8 个系统任务均为常用的时序检查系统任务。这些专用的系统任务只能在 specify 声明语句(指定块)里被调用,详细内容请参阅后面的说明。

Value Change Dump Tasks(**存储数值变化的系统任务**)

```
$dumpfile("FileName");
$dumpvars[(Levels, ModuleOrVariable,...)];
$dumpoff;
$dumpon;
$dumpall;
$dumplimit(FileSize);
$dumpflush;
```

以上 7 个系统任务用于把数值的变化存储到 VCD 文件中。VCD 文件是把仿真激励或结果传递到另一个程序(例如一个波形显示程序)的一种手段。详见后面的说明。

非标准的系统任务和函数

以下这些系统任务和函数在 Cadence 公司的 Verilog 工具中有,但它们并不属于 IEEE 标准必须包括的范围。其中有部分系统任务和函数与 Verilog 仿真工具操作时的交互方式有关。如果仿真工具支持交互方式的操作,则接受这些系统任务和函数作为其指令。

$countdrivers

```
$countdrivers (Net, [IsForced, NoOfDrivers, NoOfDriversTo0,
                                    NoOfDriversTo1, NoOfDrivrsToX ] );
```

该系统函数能返回某指定的 Net 类型标量或 Net 型矢量的某个选定位上的驱动器个数。驱动器包括原语的输出和连续赋值语句[强迫(force)启动的除外]。若 Net 含有一个以上的驱动器时,该系统函数($countdrivers)返回 0;其他情况下返回 1。在该系统任务中除第一个变量外,其余的都返回整型数。若 Net 为 force ,则 IsForced 返回 1 ,否则返回 0 。NoOfDrivers 返回驱动器个数。其他变量返回数的总和等于 NoOfDrivers 。

$list

```
$list [(ModuleInstance)];
```

在交互模式中调用此系统函数可列出在本设计中当前(或指定)范围内的源程序。

$input

```
$input("FileName");
```

从某个文本文件中读出交互命令。

$scope 和$showscopes

```
$scope(ModuleInstance);
$showscopes[(N)];
```

本系统命令用于在交互模式中设置和显示当前范围,若给定 *N* 并为非零,则还显示下面的范围。

$key、$nokey、$log 和$nolog

```
$key[("FileName")];
$nokey;
$log[("FileName")];
$nolog;
```

"key"文件记录用交互方式输入的命令,"log"文件记录在仿真期间所有写入标准设备的信息,而运行 $nokey 和 $nolog 系统任务可分别禁止这两项功能。用$key 和$log(无参数)可恢复其记录功能。如有参数,则$key 和$log 创建新的记录文件。

$reset、$reset_count 和 $reset_value

```
$reset[(StopValue[, ResetValue[, DiagnosticsValue]]);
$reset_count; {Returns an integer}
$reset_value; {Returns an integer}
```

系统任务$reset 使仿真器复位,并使仿真从头重新开始执行。StopValue 为 0 表示仿真

器复位到交互模式,允许用户自己来启动和控制仿真。而非 0 值表示仿真将会自动地从头开始仿真。ResetValue 的值可以通过$reset_value 系统函数读出。DiagnosticsValue 是指复位前仿真工具所显示信息的类型。$reset_count 返回已调用$reset 系统任务的次数。$reset_value 返回传给$reset 系统任务的值。

$save、$restart 和 $incsave

```
$save("FileName");
$incsave("FileName");
$restart("FileName");
```

$save 将完整的仿真状态保存在文件中,$restart 可以读出保存的文件。$incsave 只保存自上次调用$save 后的变化。$restart 将仿真复位并把完整的或只记录变化的文件读出。若是$restart 只记录变化的文件,原完整的仿真状态记录文件必须存在,记录变化的文件会引用完整的仿真状态文件。

$showvars

```
$showvars[(NetOrRegister,...)];
```

在标准输出设备显示 Net 和寄存器的状态。这个系统任务用于交互模式。所显示的状态信息在 Verilog LRM 工具中未作定义。状态信息可以包括当前的 Net 和寄存器值、这些 Net 和寄存器上的预定事件和 Net 的驱动器。如果未给出变量表,将显示所有当前范围的 Net 和寄存器。

$getpattern

```
$getpattern(MemoryElement);
```

$getpattern 是一个只能用于连续赋值语句的系统函数,连续赋值语句的左边必须为 Net 类型标量的位拼接。$getpattern 常与$readmemb 和 $readmemh 一起使用,可从文本文件中提取测试矢量。当有大量的标量需要输入时,$getpattern 能提供快速的处理。

$sreadmemb and $sreadmemh

```
$sreadmemb (Memory, StartAddr, FinishAddr, String, ...);
$sreadmemh (Memory, StartAddr, FinishAddr, String, ...);
```

这两个任务与$readmemb 和$readmemh 类似,只是存储器中的初始数据不是由文件输入,而是由一个或多个字符串输入。字符串格式与 $readmemb 和$readmemh 系统任务所要求的相应文件格式一致。

$scale

```
$scale(DelayName); |Returns realtime|
```

将一模块的时间值转换为调用$scale 系统任务的模块中所定义的时间单位来表示。

$scale可以引用用模块层次命名的参数(如延迟值),并将它转换为调用$scale 的模块中所定义的时间单位来表示。

常用系统任务和函数的详细使用说明

$display 和 $write

把格式化文本输出到标准输出设备及仿真器日志或其他文件。

语法

```
$display(Argument,...);
$fdisplay(Mcd, Argument,...);
$write(Argument,...);
$fwrite(Mcd, Argument,...);
Mcd = Expression |Integer value|
```

规则

$display 与 $write 的唯一区别为前者在输出结束后会自动换行而后者不会自动换行。Arguments 可以是字符串或表达式或空格。字符串内可包含以下格式控制符。若包含格式控制符(%m 除外),则每个字符串后必须有足够的表达式来为字符串中的格式控制符提供数值。

字符串中也可包含以下扩展字符:

(1) \n: 换行(Newline)

(2) \t: 制表符(Tab)

(3) \": 双引号

(4) \: 反斜杠

(5) \nnn: 用八进制表示的 ASCII 字符

不定值和高阻值这样表示(注意,八进制数的每一个数字代表 3 位,而十制数和十六进制数的每一个数字代表 4 位):对十进制数而言,若有某个数字为不定值和高阻值则写作 x、z、X、Z。若用大写的 X、Z 表示,则该数字中并非所有位(bit)为不定值和高阻值,若用小写 x、z 表示,则表示该数字中所有位(bit)为不定值和高阻值时。若参数表含二相邻逗号,则输出显示或打印--空格。

格式控制符

在字符串中允许出现下面这些格式控制符:

(1) %b %B: 二进制数(Binary)

(2) %o %O: 八进制数 (Octal)

(3) %d %D: 十进制数 (Decimal)
(4) %h %H: 十六进制数(Hexadecimal)
(5) %e %E %f %F %g %G: 实型数(Real)
(6) %c %C: 字符(Character)
(7) %s %S: 字符串(String)
(8) %v %V: 二进制数和强度(Binary and Strength)
(9) %t %T: 时间类型数(Time)
(10) %m %M: 分级实例名 (Hierarchical Instance)

格式控制符 %v 按如下的形式打印出变量的强度值:若强度值为 supply,则打印 Su; 若为 strong,则打印 St;若为 Pull,则打印 Pu;若为 Large,则打印 La;若为 Weak,则打印 We;若为 Medium,则打印 Me;若为 Small,则打印 Sm;若为 Highz,则打印 Hi。%v 也能把变量值打印为 H 和 L(这些值若用 %b 格式控制符,则只能打印为 X)。% 号后常跟有一个数用于表示打印变量值区域的宽度 (例如%10d,表示至少保留 10 位宽度给要打印的十进制数)。对十进制数,高位不足此值者以空格代替,其他进制以 0 代替,若% 号后的数为 0,则表示打印变量值区域的宽度随其值的位数自动调节。

Verilog HDL 实型数的格式符(%e, %f and %g) 其格式控制功能和 C 语言的格式符完全一样。例如,%10.3g 指至少保留 10 位宽度给要打印的十进制数,小数点后还保留 3 个数字位。若相应的变量未用格式控制符声明,则默认为是十进制数。有些系统打印任务有其自己的缺省值,如$displayb、$fwriteo、$displayh 的缺省值分别是二进制、八进制、十六进制。

示例

```
$display ("Illegal opcode %h in %m at %t", Opcode, $realtime);
$writeh ("Register values (hex .): " , reg1 , , reg2, , reg3, , reg4, "\n");
```

参考

更多内容请参考$monitor 和$strobe 的说明。

$fopen 和 $fclose

$fopen 是用于打开某个文件并准备写操作的系统任务,而 $fclose 则是关闭文件的系统任务。把文本写入文件还需要用$fdisplay、$fmonitor 等系统任务。

语法:

```
$fopen("FileName"); |Returns an integer|
$fclose(Mcd);
Mcd = Expression |Integer value|
```

在程序中的位置

请参阅 statement 声明语句的说明。

规则

一般情况下一次最多可打开 32 个文件,但若所用的操作系统不同,一次最多可打开的文件数可能不到 32。当调用$fopen 时,它返回一个 32 位(bit)(与文件有关)的无符号多通道描述符或者返回 0 值,0 值表示文件不能打开。多通道描述符可以被认为是 32 个标志每个代表 32 个文件中的一个。多通道描述符的第 0 位与标准输出设备有关,第 1 位为第 1 个文件打开的标志位,第 2 位为第 2 个文件的标志位,以此类推。当输出文件的系统任务,如$fdisplay,被调用时,其第一个参数为多通道描述符,它表示向何处写。文本被写入那些多通道描述符内标志位已设的相应文件中。

示例

```
integer MessagesFile, DiagnosticsFile, AllFiles;
    initial
      begin
        MessagesFile = $fopen("messages.txt");
        if (! MessagesFile)
          begin
            $display("Could not open \"messages.txt\"");
            $finish;
          end
        DiagnosticsFile = $fopen("diagnostics.txt");
        if (! DiagnosticsFile)
          begin
            $display("Could not open \"diagnostics.txt\"");
            $finish;
          end
        AllFiles = MessagesFile | DiagnosticsFile | 1;
        $fdisplay(AllFiles, "Starting simulation ...");
        $fdisplay(MessagesFile, "Messages from % m");
        $fdisplay(DiagnosticsFile, "Diagnostics from % m");
          ...
        $fclose(MessagesFile);
        $fclose(DiagnosticsFile);
      end
```

参考

更多内容请参考$display、$monitor、$strobe 的说明。

$monitor 等

当$monitor 系统任务所指定的参数表中中任何一个或多个 Net 或寄存器类型变量值发生变化时，便立即显示一行文本。此系统任务常用于测试模块中，以监测仿真行为的细节。

语法

```
$monitor(Argument,...);
$fmonitor(Mcd, Argument,...);
$monitoron;          |turns monitor flag on|
$monitoroff;         |turns monitor flag off|
Mcd = Expression     |Integer value|
```

规则

上面这些系统任务在变量使用的语法上与$display 系统任务完全相同。有一点与$display 系统任务不同：只能同时运行一个$monitor 系统任务。但$fmonitor 系统任务却能同时运行多个。第二次或下一次调用$monitor 系统任务，就把上一次正在执行的$monitor 系统任务取消了，用新的$monitor 系统任务代之。$monitoroff 系统任务关闭监视的功能，而$monitoron 则恢复监视的功能，它能把现存的$monitor 进程所监测到的信号不管其值是否变化立即显示出来。对$fmonitor 而言，没有与之对应的$monitoron 和 $monitoroff 系统任务。系统函数$time、$stime 和 $realtime 不会从$monitor 或$fmonitor 等系统任务触发出一行显示。

提示　在测试模块里使用$monitor 可以从任何一种 Verilog 兼容的仿真器获得仿真结果。用于生成波形图显示的任务往往与仿真器相关。

示例

```
initial
$monitor (" % t : a = % b, f = % b ",$realtime, a, f );
```

参考

更多内容请参考$display、$strobe、$fopen 的说明。

$readmemb 和 $readmemh

把文本文件中的数据读到存储器阵列中，以对存储器变量进行初始化。此文本文件的内容可以是二进制格式(用$readmemb)的，也可以是十六进制格式(用$readmemh)的。

语法

```
|System task call|
$readmemb ("File", MemoryName [, StartAddr[, FinishAddr]]);
$readmemh ("File", MemoryName [, StartAddr[, FinishAddr]]);
|Text file|
```

```
{either} WhiteSpace DataValue @ Address
WhiteSpace = {either} Space Tab Newline Formfeed
DataValue = {either}
BinaryDigit... }$readmemb}
HexDigit... {$readmemh}
Address = HexDigit...
```

规则

(1) 第一个参数是 ASCII 文件名,文件中可以包含空格、Verilog 注释语句、十六进制地址和二进制或十六进制数据。

(2) 第二个参数是存储器阵列名。

(3) 数据的位宽必须与存储器阵列的每个存储单元的位宽相同,而且每个数据之间必须用空格间隔开。数据被一个挨一个地读入连续相邻的存储器阵列中,从存储器阵列的第一个地址(若指定起始地址,则从指定的起始地址)开始,直到数据文件结束或直到存储器阵列的最后一个地址(若指定结束地址,则到指定的结束地址)为止。

(4) 地址均用十六进制数字表示且以@ 符号开头(对$readmemb 亦然)。当遇到一个地址后,下一个文本数据将被读入这个地址的存储单元。

可综合性问题

不可综合。综合工具忽略这些系统任务的存在。在可综合的设计里,从存储器阵列导出的触发器不能用这种方法初始化。如果需要上电复位对存储器阵列(RAM)初始化,则必须对其明确地编码。

提示 *存储器阵列可以存储从文本文件读出的激励源。这是把数据读进 Verilog 仿真器的唯一方式,而无须另外使用编程语言接口(PLI)或非标准语言扩展来做到这一点。*

示例

```
module Test;
    reg a,b,c,d;
    parameter NumPatterns = 100;
    integer Pattern;
    reg [3:0] Stimulus[1:NumPatterns];
    MyDesign UUT(a, b, c, d, f);
    initial
        begin
          $readmemb("Stimulus.txt", Stimulus);
          Pattern = 0;
          repeat (NumPatterns)
```

```
            begin
              Pattern = Pattern + 1;
              {a,b,c,d} = Stimulus[Pattern];
              #110;
              end
        end
    initial
        $monitor("% t a = % b b = % b c = % b d = % b : f = % b", $realtime,
                 a, b, c, d, f);
endmodule
```

$strobe

$strobe 将在所有事件都已处理完毕后的时刻打印出一行格式化的文本。

语法

```
$strobe(Argument,...);
$fstrobe(Mcd, Argument,...);
Mcd = Expression {Integer value}
```

规则

本系统任务($strobe)有关参数以及文本打印的语法与系统任务$display 完全一样。但$strobe 只打印调用此系统任务的时刻且当所有活动事件都已结束后的信息,其中可包括所有阻塞和非阻塞赋值产生的效果。

提示 在写仿真激励模块时,若想打印出仿真结果,应优先考虑使用$strobe 系统任务。因为与使用$display 或$write 比较,系统任务$strobe 可以保证显示出写入 Net 和寄存器类型变量的是一个稳定的数值。

示例

```
initial
        begin
            a = 0;
            $display(a);        //displays 0
            $strobe(a);         //displays 1 ...
            a = 1; //... because of this statement
        end
```

参考

更多内容请参考$display、$monitor, $write 的说明。

$timeformat

$timeformat 定义仿真时间的打印格式。系统任务 $timeformat 应配合格式控制符%t 使用。

语法

```
$timeformat[(Units, Precision, Suffix, MinFieldWidth)];
```

规则

(1) Units(单位)是指打印的时间单位,它是一个 0 到 -15 之间整型数,0 表示秒(s),-3 表示毫秒(ms),-6 表示微秒(μs),-9 表示纳秒(ns),-12 表示皮秒(ps),-15 表示浮秒(femtosecond),中间的整数也可用,如 -10 表示 100 皮秒(ps),依此类推;

(2) Precision 是指打印的十进制数小数点后保留的位数;

(3) Suffix 指打印时间值后跟的字符串;

(4) MinFieldWidth 指打印出的字符的最少个数,其中包括前面的空格。若需要打印的字符多,则需要取较大的整数;

(5) 缺省形式,即不指定参数,自动设置为:Units(单位)为仿真的时间精度;Precision(精度)为 0;Suffix 为无;MinFieldWidth 为 20。

提示 在使用$display、$monitor 或其他显示任务时,应使用 `timescale, $timeformat 和 $realtime (并配合 %t) 来指定和显示仿真时间。

示例

```
$timeformat (-10, 2, " x100ps", 20); //20.12 x100ps
```

参考

更多内容请参考`timescale 与$display 的说明。

Stochastic Modelling(**随机模型**)

Verilog 提供了一整套系统任务和函数,可用来启动随机序列的生成和管理以支持建立随机模型。

语法:

```
$q_initialize(q_id, q_type, max_length, status);
$q_add(q_id, job_id, inform_id, status);
$q_remove(q_id, job_id, inform_id, status);
$q_full(q_id, status); {Returns an integer}
$q_exam(q_id, q_stat_code, q_stat_value, status);
```

在程序中的位置

请参阅 statement 声明语句的说明。

概述

所有这些系统任务和函数的参数都是整型数。每个系统任务和函数都返回一整数型的状态(status)值,它为下列值之一:

(1) 0　　OK;

(2) 1　　队列已满:不能再增加工作($q_add);

(3) 2　　未定义的 q_id;

(4) 3　　队列空:不能再删除工作($q_remove);

(5) 4　　不支持的队列形式:不能创建这个队列($q_initialize);

(6) 5　　最大长度小于等于0:不能创建这个队列($q_initialize);

(7) 6　　两个相同的 q_id:不能创建这个队列($q_initialize);

(8) 7　　内存不足:不能创建这个队列($q_initialize)。

系统任务 $q_initialize

创建一个队列。q_id (输出)是唯一的队列标识符,当程序需要调用多个队列任务和函数时,可用该标识符来区别各个队列。q_type (输入)可为 1 或 2,1 表示 FIFO(先进先出)队列,2 表示 LIFO(后进先出)队列。max_length (输入)为队列所允许的最多的输入个数(即最大长度)。

系统任务 $q_add

向队列加进一个入口。q_id (输入)表示向哪个队列加输入口。job_id (输入)表示是哪个工作(job),它通常为一整型数,每次向队列加入一个新元素其值加 1,这样当队列的某一元素需要移走,可以用 job_id 来识别。inform_id (输入)用于定义与队列入口有关的信息,由用户自己来定义。

系统任务 $q_remove

从队列取一个入口。q_id (输入)表示从哪个队列取走该入口。job_id (输出)确定是哪个工作(请参阅 $q_add 的说明)。inform_id (输出)是由$q_add 存储的数值。

系统任务 $q_full

检查队列是否满。若返回值为 1 则队列满;为 0 则不满。

系统任务 $q_exam

取得队列不同类型的统计信息。下列描述中提及的时间是基于何时队列元素被加入到队列中(到达时间)以及队列元素从加入队列到被删除出去的时间差(等待时间)。时间单

位为仿真的时间精度。其中 q_stat_value (输出)参数返回取得的消息,而其中 q_stat_code (输入) 参数可以取 1~6,分别表示要求取得的信息类型。

1:当前队列长度

2:平均达到时间间隔

3:最大队列长度

4:最短等待时间

5:当前队列中队列元素的最长等待时间

6:本队列的平均等待时间

示例

```
module Queues;
    parameter Queue = 1; //Q_id
    parameter Fifo = 1, Lifo = 2;
    parameter QueueMaxLen = 8;
    integer Status, Code, Job, Value, Info;
    reg     IsFull;
    task Error; //Write error message and quit
        ...
    endtask
    initial
      begin
        //生成后进先出队列,其标号为为1,队列长度为8。
          $q_initialize (Queue, Lifo, QueueMaxLen, Status);
          if (Status )
            Error("Couldn't initialize the queue");
        //向1号队列加入从1号到8号共8个工作,每个job之间间隔10个单位时间
        //每次从1号队列加入的信息为job号加100
          for (Job = 1; Job <= QueueMaxLen; Job = Job + 1)
            begin
              #10 Info = Job + 100;
              $q_add (Queue, Job, Info, Status);
              if (Status )
                Error("Couldn't add to the queue");
              $display("Added Job % 0d, Info = % 0d", Job, Info);
              $write("Statistics:");
/* *要求取得有关当前队列长度、平均达到时间间隔、最大队列长度、最短等待时间、当前队列
```

```
中队列元素的最长等待时间和本队列的平均等待时间共6种队列信息 * */
        for (Code = 1; Code <= 6; Code = Code + 1 )
          begin
            $q_exam(Queue, Code, Value, Status);
            if (Status )
              Error("Couldn't examine the queue");
            $write("% 8d", Value); //显示6种队列信息
          end
      $display("");
    end
    //队列此时应是满的
      IsFull = $q_full(Queue, Status);
       if (Status )
         Error("Couldn't see if queue is full");
       if (! IsFull )
         Error("Queue is NOT full");
    //去除工作
         repeat (10 )
         begin
           #5 $q_remove(Queue, Job, Info, Status);
           if (Status )
             Error ("Couldn't remove from the queue");
           $display("Removed Job % 0d, Info = % 0d", Job,Info);
           $write("Statistics: ");
           for (Code = 1; Code <= 6; Code = Code + 1 )
           begin
             $q_exam(Queue, Code, Value, Status);
             if (Status )
               Error("Couldn't examine the queue");
             $write("% 8d", Value);
           end
           $display("");
         end
      end
endmodule
```

参考

更多内容请参考$random、$dist_chi_square 等系统任务的说明。

Timing Checks(时序检查)

Verilog 提供了一些系统任务,这些系统任务仅能在 specify 声明语句(指定块)里调用,以进行常见的时序检查。

语法

```
$hold(ReferenceEvent, DataEvent, Limit [, Notifier]);
$nochange(ReferenceEvent, DataEvent, StartEdgeOffset, EndEdgeOffset [,
        Notifier]);
$period(ReferenceEvent, Limit [, Notifier]);
$recovery(ReferenceEvent, DataEvent, Limit [, Notifier]);
$setup(DataEvent, ReferenceEvent, Limit [, Notifier]);
$setuphold(ReferenceEvent, DataEvent, SetupLimit, HoldLimit [, Notifier]);
$skew(ReferenceEvent, DataEvent, Limit [, Notifier]);
$width(ReferenceEvent, Limit [, Threshold [, Notifier]]);

ReferenceEvent = EventControl PortName [&&& Condition]
DataEvent = PortName
Limit = |either| ConstantExpression SpecparamName
Threshold = |either| ConstantExpression SpecparamName
EventControl = |either|
    posedge
    negedge
edge [TransitionPair,... ]
TransitionPair = |either| 01 0x 10 1x x0 x1
Condition = |either|
ScalarExpression
`ScalarExpression
    ScalarExpression == ScalarConstant
    ScalarExpression == = ScalarConstant
    ScalarExpression != ScalarConstant
    ScalarExpression !== ScalarConstant
```

规则

(1) 参考事件(ReferenceEvent)的变化提供了时序检查的时间基准,参考事件必须通过模块的输入口(input)或输入/输出口(inout)引入;

(2) 数据事件(DataEvent)的变化会启动时序检查,数据事件也必须通过模块的输入口(input)或输入/输出口(inout)引入;

(3) 如果参考事件与数据事件同时发生,这时虽不会产生建立违例报告,但会产生保持违例报告;

(4) 对于系统任务$width, 脉冲如果低于门限(Threshold)参数的设定(若设置了门限),则不会发生违例报告;

(5) 下列时序检查系统任务中的参考事件必须是沿触发的:$width、$period、$recovery、$nochange;

(6) 以上系统任务中 ReferenceEvent 参数都可以用关键字 edge,除了$recovery 和$nochange 这两个系统任务例外,它们的 ReferenceEvent 参数只能用 posedge 和 negedge;

(7) 使用 &&& 做注释的条件时序检查仅在条件为真时才执行;

(8) 在以上的系统任务里如果设置了 notifier 参数,则其必须为寄存器类型变量,当违例发生时,寄存器变量数值发生变化:若原为不定值则变为0,若原为0值则变为1,若原为1则变为0,若原为高阻则不变。

注意 这些系统任务仅能在 specify 声明语句(指定块)中调用,而不能用作程序声明语句。

参数 ReferenceEvent 和 DataEvent 在系统任务$setup 里是颠倒的。

提示 若条件比较复杂,应在 specify 声明语句(指定块)外描述条件,而把驱动条件的信号(wire 或 reg 类型)放在指定块内。

示例

```
reg Err, FastClock;        //Notifier registers
specify
    specparam Tsetup = 3.5, Thold = 1.5,
    Trecover = 2.0, Tskew = 2.0,
    Tpulse = 10.5, Tspike = 0.5;
    $hold(posedge Clk, Data, Thold);
    $nochange(posedge Clock, Data, 0, 0 );
    $period(posedge Clk, 20, FastClock)];
    $recovery(posedge Clk, Rst, Trecover);
    $setup(Data, posedge Clk, Tsetup);
    $setuphold(posedge Clk &&& ! Reset, Data, Tsetup, Thold, Err);
    $skew(posedge Clk1, posedge Clk2, Tskew);
    $width(negedge Clk, Tpulse, Tspike);
endspecify
```

参考

更多内容请参考 specify、specparam 声明语句的说明。

Value Change Dump

以下 7 个系统任务用于把数值的变化存储到 VCD 文件中。VCD 文件是把仿真激励或结果传递到另一个程序(例如一个波形显示程序)的一种手段。

语法

```
$dumpfile("FileName");
$dumpvars[(Levels, ModuleOrVariable,...)];
$dumpoff;       |suspend dumping|
$dumpon;        |resume dumping|
$dumpall;       |dump a checkpoint|
$dumplimit (FileSize);
$dumpflush; |update the dump file|
```

在程序中的位置

请参阅 statement 声明语句的说明。

规则

(1) 系统任务$dumpvars 中的参数 Levels 表示想要把指定模块的哪些层次的数值变化记录到 VCD 文件中:若设置为 1 表示仅记录指定层次模块中的变化,0 表示不但记录该层次模块还记录所有与该模块有关的下层模块中的变化;

(2) 如果没有设置任何参数,则设计中所有变量的变化均记录到 VCD 文件;

(3) FileSize 参数是用于设置 VCD 文件可记录的最多字节数;

(4) 在测试程序中可写多个系统任务$dumpvars 调用,但是每个调用必须在同一时刻(通常在仿真开始时)。

示例

```
module Test;
    ...
    initial
      begin
        $dumpfile("results.vcd");
        $dumpvars(1, Test);
      end
  //Perform periodic checkpointing of the design.
```

```
    initial
      forever
        #10000 $dumpall;
endmodule
```

Command Line Options(命令行的可选项)

虽然怎样选用启动 Verilog 仿真器工作的命令行可选项并不是的 Verilog 语法的一部分，大多数有关 Verilog 语法学习参考材料里也不提供这方面的资料，但是为了更快掌握仿真工具还是有必要介绍一些这方面的资料，因为绝大多数仿真器都支持一些常见共同的 Verilog 编译命令选项(尽管有的 Verilog 仿真工具还有自己的一些命令选项)，熟练地掌握这些共同的选项能更有效地进行仿真，提高 Verilog 仿真工具的使用效果。

UNIX Verilog 编译命令选项分为两类：一类是一个字符的，前面带有一个减号 -(如 -s)；另一类是多个字符的，前面带有一个加号(如 +word)。有些 UNIX Verilog 编译命令选项后还可跟一个值，例如，跟一个文件名(如 -f file)。

下面介绍一些最有用和最常见的 Verilog 编译命令选项(注意，并非所有仿真器都支持这些选项)：

-f CommandFile	除从命令行输读入命令选项外，还从命令文件读入更多的命令选项；
-k KeyFile	在 KeyFile 里记录仿真期间所有键入的交互命令；
-l LogFile	除了在显示器输出外还把所有仿真信息(包括$display 等的输出)记录到 LogFile 中；
-r SaveFile	从由非标准的系统任务$save 生成的文件再次开始仿真；
-s	在 0 时刻中断仿真器，以便采用交互方式来控制仿真的进行；
-u	将 Verilog 源代码中的字符都视为大写字符(字符串除外)，用此选项时要小心；
-v LibraryFile	在 LibraryFile 中寻找设计文件中缺少的 UDP 或模块。只有那些在设计文件中并未定义但已实例引用的 UDP 或模块编译器才会在 LibraryFile 中寻找。而设计中没有引用的在 LibraryFile 中的 UDP 或模块不会进行编译；
-y LibraryDirectory	在 LibraryDirectory 中的文件中寻找设计文件中缺少的 UDP 或模块，期望在该库的目录中有一个文件已定义了一个与其同名的模块。如果给出 +libext + 扩展名的命令行选项，则是指在搜寻文件时将带扩展名搜寻。例如，-y mylib + libext + .v 是指在 mylib 目录中，在扩展名为.v 的文件范围内搜寻在设计中未定义的同名的模块；
+define+ MacroName	定义一段文字作宏名(无值)。这种零值的宏可以用于‘ifdef 中来控制(条件)编译的范围；

+incdir + Directory{ + Directory...]	定义搜索路径,搜索用'include包含的文件。搜索开始于当前路径,如果没有找到,就去由+incdir+定义的搜索路径依次去找;
+libext + Extension	定义库文件扩展名。(请参阅 -y 的说明)
+notimingchecks	关闭指定块的时序检查,此举可以加速仿真,或抑制伪时序错误信息,使用此选项时需小心;
+mindelays、+typdelays、+maxdelays	以上分别是指仿真时使用最小、典型和最大延迟,缺省为使用典型延迟。仿真时不要混淆以上3种延迟;

注意 Verilog仿真器不能检查出+号后选项参数的拼写错误,这是因为Verilog命令行允许用户自己来定义+号后的参数,所以一定小心注意选项的拼写,例如"+maxdelays"要注意拼写正确。

第三部分　IEEE Verilog 1364-2001 标准简介

摘要　2001 年 3 月 IEEE 正式批准了 Verilog-2001 标准(即 IEEE 1364-2001)。Verilog-2001 标准在 Verilog-1995 的基础上有几个重要的改进。新标准有力地支持可配置的 IP 建模,大大提高了深亚微米(DSM)设计的精确性,并对设计管理作了重大改进。别的一些改进使其更加容易使用。这些改进将会影响每一个 Verilog 用户和 EDA 工具的设计人员。阅读了最近出版的几篇有关英文文献后,笔者对 Verilog-2001 新标准中的若干个改进作了简要的总结和介绍。

关键词　Verilog、HDL、硬件描述语言、EDA

一、Verilog 语言发展历史回顾

Verilog 硬件描述语言诞生于 1984 年。最初它只用在 Gateway 公司的 一个名为 Verilog-XL 的数字逻辑仿真器上。1989 年 Cadence 公司收购了 Gateway 公司。1990 年 Cadence 公布了 Verilog 硬件描述语言和它的编程语言接口(PLI)。不久 Verilog 国际组织,Open Verilog International (简称 OVI),正式成立,其宗旨是规范 Verilog 的公用部分和推广它的应用。OVI 有关 Verilog 1.0 版本的资料最初来自于 Cadence 公司的 Verilog-XL 手册。1993 年 OVI 公布了 Verilog 2.0 版本,新版本对 Verilog 做了一些改进。随后,OVI 正式向 IEEE 提出申请要求批准它为标准。不久,IEEE Verilog 标准化工作组成立,并在 1995 年正式批准其为 Verilog 语言的标准,即 IEEE 1364-1995。这是第一个得到 IEEE 官方批准的 Verilog 标准。我们应该注意到 IEEE Verilog 标准化工作组当时并没有对 Verilog 语言做任何本质上的提高。工作组的目标只是把当时应用比较普及的 Verilog 仿真工具所使用的 Verilog 语言标准化,而并没有决心彻底重新编写新标准的文档。因为最初的标准来自于手册,所以 IEEE 1364-1995 和最新推出的 IEEE 1364-2001 标准也与用户使用手册很类似。

二、IEEE 1364-2001 Verilog 标准的目标

编写 IEEE 1364-2001 Verilog 标准以下简称 Verilog-2001 标准的工作开始于 1997 年 1 月,当时确立了 3 个工作目标:

（1）改进 Verilog，使其有助于深亚微米设计和 IP（知识产权模块）建模问题；

（2）确保上述所有改进既有用，也切实可行，并使得仿真器和综合器厂商能在其产品中支持 Verilog-2001；

（3）纠正 IEEE1364-1995 Verilog 标准（以下简称 Verilog-1995 标准）参考手册中的所有错误，明确那些原来模棱两可的概念。

Verilog-2001 标准化工作组由 20 个成员组成，他们代表了各种不同的 Verilog 用户、仿真器厂商以及综合器厂商等多方的利益。

整个工作组分为 3 个执行小分队：ASIC 任务小队，提高了语言的性能以满足超深亚微米设计的时间精确性；行为级任务小队，提高了行为级和 RTL 级语言建模的性能；PLI 任务小队，增强了 Verilog 编程语言接口的功能，使其能支持前面两个小队所做的改进，同时使 PLI 又新增加了一些的功能。

三、新标准使建模性能得到很大提高

下面列出了 21 个改进之处，可以为 Verilog 设计人员提供了更强的 Verilog 建模能力。许多改进使得编写可综合 RTL 模型变得更容易也更准确。别的一些改进使得模型的维数可扩展性和可重复利用性更强。本节只是列出了新增加的功能和语法，没有涉及 Verilog-1995 的相关说明。

1. 设计管理——Verilog 配置

Verilog-1995 标准把设计管理工作交给独立的软件工具来承担，设计管理不是语言的一部分。各个仿真器厂商的设计管理工具在处理不同版本的 Verilog 模型时有各自的方法，但是这些和工具有关的方法不能适用不同版本的 Verilog 编写的模型。

而 Verilog-2001 中增加了配置块（Configuration Block），它属于新版本 Verilog 语言的一部分，可以用它对每一个 Verilog 模型指定其具体版本和其源代码的位置。出于可移植性的考虑，在配置块中，可以使用多个虚拟模型库，还需要配合独立的库映像文件（Library Map Files）指出其具体物理位置。配置块位于模块定义外。配置块的名字和模块名、原语块（primitive）名存在于同一个命名空间中。在 Verilog-2001 中新增加了关键字 config 和 endconfig，其他新增加的关键字：design、instance、cell、use 和 liblist 留给配置块使用。

本书不想介绍 Verilog 配置块的完整语法和使用规则等方面的内容。下面的例子是一个简单的设计配置。其 Verilog 源代码是典型的，其中测试（Test Bench）模块包含了设计层次树顶层的实例调用，设计顶层还包括其他模块中的实例。

```
module test;
```

```
  ...
myChip dut (...); /* instance of design */
  ...
endmodule
module myChip(...);
  ...
  adder a1 (...);
  adder a2 (...);
  ...
endmodule
```

配置模块可以指定全部或个别模块实例的源代码的位置。因为配置模块位于 Verilog 模块之外,所以需要重新配置时,Verilog 模型的源代码不需要做任何修改。在这个配置例子中,加法器实例 a1 由 RTL 库编译生成,实例 a2 则来自一个门级库。

```
config cfg4                                  /* 给配置块命名 */
design rtlLib.top                            /* 指定从哪里找到顶层模块 */
default liblist rtlLib gateLib;              /* 设置查找实例模块的缺省顺序 */
instance test.dut.a2 liblist gateLib;        /* 明确指定模块实例使用哪一个库 */
endconfig
```

配置模块使用虚拟库来指定 Verilog 模型源代码的位置。使用库映射文件将虚拟库的名字和实际的文件位置联系起来。例如:

```
library rtlLib ./* .v;                       /* RTL 库模型的位置(当前目录) */
library gateLib ./synth_out/* .v;            /* 门级库模型的位置 */
```

2. 用 Verilog 生成维数可扩展的模块

Verilog-1995 标准在设计维数可扩展和可重复使用的模块时功能有限。它虽然具有强大的实例阵列结构,但是并不具备真正维数可扩展性和设计复杂结构所需要的灵活性。

而 Verilog-2001 增加了生成循环(Generate Loop),允许生成多个模块和原型的实例,同时生成多个变量、网络、任务、函数、连续赋值、初始化过程块以及 always 过程块。可以使用 if - else 或 case 语句有条件地生成声明语句和实例引用语句。

Verilog-2001 中定义了 4 个与此相关的新关键字:generate、endgenerate、genvar 以及 localparam。其中,关键字 genvar 是一种新的数据类型,这个数据类型用于存储正的整型变量;与其他 Verilog 变量不同,它的值可以在编译和详细描述(Elaboration)时改变。生成循环中的索引变量必须定义成 genvar 类型。

关键字 localparam 表示一个常数,和 parameter 类似,但是 localparam 与 parameter 的不同

点在于它不能够使用参数重定义(Parameter Redifinition)来改变赋值。生成模块(Generate Block)同样可以通过专门的 Verilog 语句,来控制生成的将是何种对象。这些语句包括 for 循环、if-else 语句以及 case 语句等。

下面的例子说明了使用生成(generate)语句创建维数可扩展的乘法器模块实例。当乘法器的 a 和 b 的位宽参数小于 8,生成 CLA 乘法器实例。如果 a 和 b 的位宽大于或等于 8,则生成 Wallace 数乘法器。

```
module multiplier (a, b, product);
    parameter a_width = 8, b_width = 8;
    localparam product_width = a_width + b_width;
    input [a_width - 1:0] a;
    input [b_width - 1:0] b;
    output [product_width - 1:0] product;

    generate
      if((a_width < 8) || (b_width < 8))
      CLA_multiplier #(a_width, b_width)
      ul (a, b, product);
    else
    WALLACE_multiplier #(a_width, b_width)
    ul (a, b, product);
    endgenerate

endmodule
```

下面的例子说明了如何使用 generate-for 循环语句引用原语(Primitive)实例和原语实例的内部互连来创建的多位宽加法器,其位宽和引用实例数目由可重定义的参数常数指定。

```
module Nbit_adder (co, sum, a, b, ci);
parameter SIZE = 4;
output [SIZE - 1:0] sum;
output co;
input [SIZE - 1:0] a, b;
input ci;
wire [SIZE:0] c;
genvar i;
assign c[0] = ci;
assign co = c[SIZE];

generate
  for(i =0; i < SIZE; i = i + 1)
```

```
        begin:addbit
          wire n1,n2,n3; //internal nets
          xor g1 (n1, a[i], b[i]);
          xor g2 (sum[i],n1, c[i]);
          and g3 (n2, a[i], b[i]);
          and g4 (n3, n1, c[i]);
          or g5 (c[i+1],n2, n3);
        end
    endgenerate

endmodule
```

前面的例子中,每个生成的网络都有一个独立的名字,而且每个生成的原语(primitive)实例也具有独立的名字,该名字由 For 循环中程序块的名字和作为循环变量的 genvar 类型变量值组成。例如,n1 网络中的名字如下:

```
addbit[0].n1
addbit[1].n1
addbit[2].n1
addbit[3].n1
```

为第一个 xor 原语(primitive)生成的实例名字如下:

```
addbit[0].g1
addbit[1].g1
addbit[2].g1
addbit[3].g1
```

需要注意的是,这些生成的名字中包含了方括号,这在用户定义的标识符中是非法的,但是在生成的名字中是合法的。

3. 常数函数

Verilog 语法规定必须使用数值或常数表达式来定义向量的宽度和阵列的规模,例如:

```
parameter WIDTH = 8;
wire [WIDTH-1:0] data;
```

Verilog-1995 标准的局限之一是上述表达式必须基于算术操作。使用编程语句定义上述表达式的值是不允许的。

Verilog-2001 给 Verilog 函数定义了一种新用法,称为常数函数。常数函数的定义和任何 Verilog 函数的定义相同。但是常数函数只能使用这样一种结构,它的具体数值是在编译或详细描述(Elaboration)过程中被确定的。

常数函数有助于创建可改变维数和规模的可重用模型。下面的例子定义了一个称为clogb2 的函数,该函数返回一个整数(即以 2 为底的对数的极限值)。这一常数函数可用于根据 RAM 中的单元数目,以确定 RAM 地址总线必需的宽度。

```
module ram (address_bus, write, select, data);
    parameter SIZE = 1024;
    input [clogb2(SIZE) -1:0] address_bus;
    ...
      function integer clogb2 (input integer depth);
      begin
        for(clogb2 =0; depth >0; clogb2 = clogb2 +1)
          depth = depth >> 1;
        end
      endfunction
...
endmodule
```

4. 选择索引向量的一部分

Verilog-1995 标准中对向量某些位的选择是允许的,但被选择的部分必须是固定的。因此使用变量来选择长字中某个字节是不符合语法的。而 Verilog-2001 中增加了一个新的语法,称为索引的部分选择(Indexed Part Selects)。索引的部分选择由基表达式、宽度表达式和偏移方向 3 部分组成,其形式如下:

```
[base_expr +: width_expr]      //正偏移
[base_expr -: width_expr]      //负偏移
```

其中,基表达式可以随着仿真过程的运行面变化,但是宽度表达式必须是常数。偏移方向表示选择区间究竟是基表达式是加上宽度表达式,还是减去宽度表达式。例如:

```
reg [63:0] word;
reg [3:0] byte_num; //a value from 0 to 7
wire [7:0] byteN = word[byte_num *8 +: 8];
```

上例中,如果 byte_num 的值是 4,则把 word[39:32] 赋给 byteN。其中,起始位 32 由基表达式确定,终止位 39 则由基正偏移和宽度确定。

5. 多维矩阵

Verilog-1995 标准只允许一维矩阵变量。Verilog-2001 打破了这一限制,允许使用:

(1) 多维矩阵。

(2) 含有变量和网络(Net)数据类型的矩阵。

这一改进需要改变矩阵声明和矩阵索引的语法。下例说明了一维矩阵和三维矩阵的声明和索引语法。

Verilog-1995 和 Verilog-2001 标准都支持一维矩阵的描述,下面两条语句表示的是变量为 8 位寄存器组的一维矩阵:

```
reg[7:0]array1[0:255];                    //变量为 8 位寄存器组的一维矩阵
wire[7:0]out1 = array1[address];          //变量为 8 位的一维矩阵
```

只有 Verilog-2001 标准才支持三维矩阵,用语句表示如下:

```
wire [7:0] array3 [0:255][0:255][0:15];          //变量为 8 位寄存器组的三维矩阵
wire [7:0] out3 = array3[addr1][addr2][addr3];   //变量为 8 位的三维矩阵
```

6. 选择矩阵内的某位和某几位

Verilog-1995 标准不允许直接访问矩阵字的某一位或某几位。必须将整个矩阵字复制到一个暂存变量中,从暂存变量中访问。Verilog-2001 去除了这一限制,允许直接访问矩阵字的某一位或某几位。举例说明如下:

```
//选择变量为 32 位的二维矩阵中某一单元[100][7]的最高字节[31:24]
reg [31:0] array2 [0:255][0:15];
wire [7:0] out2 = array2[100][7][31:24];
```

7. 带符号的算术运算的扩展

对于整型算术操作,Verilog 根据操作数的数据类型来决定作无符号运算还是带符号运算。通常(也有例外),只要表达式中有无符号操作数,就执行无符号操作;只有当表达式所有的操作数都是带符号数时,才执行带符号操作。

在 Verilog-1995 标准中,整型数据是有符号数,寄存器和网络型数据是无符号数。Verilog-1995 的局限之一是整型数据必须具有固定的矢量位宽,大多数的 Verilog 仿真器将其定义为 32 位。因此,根据 Verilog-1995 标准,有符号的整数算术运算只限于 32 位矢量。对此,Verilog-2001 标准作了 5 点改进,大大增强了有符号数算术运算的能力:

(1) 寄存器和网络型数据可以定义为有符号。

(2) 函数返回值可以定义为有符号。

(3) 任何进制的整数都可以定义为有符号。

(4) 操作数可以由无符号变为有符号。

(5) 可以进行算术移位操作。

Verilog-1995 标准中保留了一个关键字 signed，但是没有用到。Verilog-2001 标准使用了这个关键字，使得寄存器数据类型、网络数据类型、端口以及函数都可以定义成带符号的类型。下面举几个例子说明：

```
reg signed [63:0] data;
wire signed [7:0] vector;
input signed [31:0] a;
function signed [128:0] alu;
```

在 Verilog-1995 标准中，没有指定基数（进制）的整型数被认为是有符号数，相反，指定了基数（进制）的整型数被认为是无符号数。Verilog-2001 标准增加了一个额外的标识符，字母‘s’，和基数标识符一起指定该数是带符号数。举例说明如下：

```
16'hC501          //16 位十六进制无符号数
16'shC501         //16 位十六进制有符号数
```

除了可以定义有符号数据类型和数值外，Verilog-2001 标准还增加了两个新的系统函数：$signed 和$unsigned。这两个系统函数可以将无符号数变换为带符号数，或相反。举例说明如下：

```
reg [63:0] a;                    //无符号数据类型
always @(a) begin
result1 = a /2;                  //无符号运算
result2 = $signed(a) /2;         //有符号运算
end
```

Verilog-2001 标准中另一个关于带符号数运算的改进是算术移位操作符，右移和左移分别用符号 >>> 和 <<< 表示。算术右移操作不改变数值的符号，移位时，用符号位填充空缺位。

例如，如果 8 位带符号变量 *D* 的值为 8'b10100011，3 位的逻辑右移和算术右移的结果如下：

```
D >> 3          //逻辑右移的结果是 8'b00010100
D >>> 3         //算术右移的结果是 8'b11110100
```

8. 幂运算符**

Verilog-2001 标准中增加了幂运算，用符号 ** 表示，实现与 C 语言中 pow( ) 函数相同的功能。两个操作数中只要有一个实型数，函数就返回实型数；如果两个操作数都是整型数，函数才返回整型数。幂运算常常用来计算 2^n 的值。

举例说明如下：

```
always @(posedge clock)
  result = base * * exponent;
```

9. 可重入任务和递归函数

Verilog-2001 标准中增加了一个新的关键字——automatic。这个关键字可以用于定义可重入(Re-entry)的自动任务(Task)。自动任务中的每一条声明语句,在每次当前任务的调用中,都会进行动态的分配定位。函数(Function)也可以定义为自动的,使得函数可以递归调用(函数中的每条声明语句在每次递归调用中都将动态分配定位)。自动任务或函数中的声明语句不能通过层次名调用。

Verilog-2001 标准中不用关键字 automatic 声明的任务或函数是静态的,与 Verilog-1995 标准中的任务和函数的表现完全相同。静态任务或函数中的每条声明语句是静态分配定位的,即由该任务或函数的每次调用所共享。下面的例子说明了一个递归调用自己的函数,实现 32 位无符号整型操作数的 $n!$(阶乘)的功能。

```
function automatic [63:0] factorial;
    input [31:0] n;
    if (n == 1)
    factorial = 1;
    else
    factorial = n * factorial(n-1);
endfunction
```

10. 组合逻辑的电平敏感符号@ *

为了使用 Verilog always 过程块正确地为组合逻辑建立模型,敏感列表必须包含逻辑块中用到的所有输入信号。编写大型复杂的组合逻辑模块时很容易在敏感列表中遗漏某一个输入信号,从而导致仿真和综合的结果不一致。

Verilog-2001 标准中增加了一个新的符号@ *,用于表示组合逻辑的敏感列表。@ * 表示仿真器和综合工具应该能够自动地对该符号下逻辑块中每条语句中被赋于的值敏感。下例中@ * 符号使得逻辑在 *sel*、*a* 或 *b* 变化时,*y* 值能自动地跟着变化。

```
always @*        //组合逻辑的敏感性问题
if (sel)
    y = a;
else
    y = b;
```

11. 用逗号分隔的敏感列表

Verilog-2001 标准中增加了表示敏感列表的另一种写法，即用逗号而不是 or 关键字来分隔敏感列表中的各个信号。下例表示了功能完全一致的敏感列表的两种不同的写法：

```
always @(a or b or c or d or sel)
always @(a, b, c, d, sel)
```

用逗号分隔的敏感列表并没有增加新的功能，但是更加直观，与 Verilog 中的其他信号列表更加一致。

12. 增强的文件输入输出操作

在 Verilog-1995 标准中，Verilog 语言在文件的输入/输出操作方面功能非常有限。因而，文件操作经常是借助于 Verilog PLI(编程语言接口)，通过与 C 语言中文件输入/输出库的访问来处理的。Verilog-1995 标准规定，可以同时打开的 I/O 文件数目不能超过 31 个。

Verilog-2001 标准中增加了新的系统任务和函数，为 Verilog 语言提供了强大的文件输入输出操作，而不需要使用 PLI。同时，扩展了可以同时打开的文件数目至 230。这些新的文件操作任务和函数包括：$ferror、$fgetc、$fgets、$fflush、$fread、$fscanf、$fseek、$fscanf、$ftel、$rewind 和$ungetc。Verilog-2001 标准还给出了读写字符串的系统任务，包括：$sformat、$swrite、$swriteb、$swriteh、$swriteo 和$sscanf，它们可以用来生成格式化的字符串或从字串中读取信息。

13. 超过 32 位宽的自动宽度扩展

Verilog-1995 标准中对于不指定位数的位宽超过 32 的总线赋高阻时，只会将低 32 位赋成高阻，高位将赋 0。为了将整个总线的所有位都置为高阻态，必须明确指定总线的位数。

举例说明如下：(按照 Verilog-1995 标准的规定)：

```
parameter WIDTH = 64;
reg [WIDTH-1:0] data;
data = 'bz;          //data = 'h00000000zzzzzzzz
data = 64'bz;        //data = 'hzzzzzzzzzzzzzzzz
```

Verilog-1995 标准中的填充法则使得难于编写便捷的向量位宽可变的模型。参数重定义是一种方法，但是必须修改 Verilog 源代码，改变其赋值语句中的位宽值。

Verilog-2001 标准改变了赋值扩展法则，将高阻或不定态赋给未指定位宽的信号时，可以自动地扩展到信号的整个位宽范围。

举例说明(按照 Verilog-2001 标准的规定)如下：

```
parameter WIDTH = 64;
reg [WIDTH-1:0] data;
data = 'bz;          //data = 'hzzzzzzzzzzzzzzzz
```

14. 使用参数名在线传递参数

Verilog-1995 标准有两种方法实现在模块中重新定义参数:一是使用 defparam 语句显式地重新定义;二是在模块实体调用时使用 # 符号 隐式地重新定义参数。第二种方法更为简洁,但是由于参数的重新定义与声明的位置有关,因而容易出错而且代码的含义不易于理解。

```
module ram (...);
    parameter WIDTH = 8;
    parameter SIZE = 256;
    ...
endmodule
module my_chip (...);
    ...
    //用参数名显式地重新定义参数
    RAM ram1 (...);
    defparam ram1.SIZE = 1023;
    //根据位置隐式地重新定义参数
    RAM #(8,1023) ram2 (...);
endmodule
```

Verilog-2001 标准提出了第三种方法:在线显式重新定义。这种新方法允许在线参数值按照任意的顺序排列,并且可以对重新定义的参数加注释。

```
//在线显式参数的重新定义
    RAM #( .SIZE(1023)) ram2 (...);
```

15. 端口声明和数据类型声明结合起来的声明语句

Verilog 语法要求每一个连接到模块输入或输出端口的信号必须声明两点:输入输出方向和数据类型。Verilog-1995 标准中这两个声明语句是分开写的。Verilog-2001 标准提出了更简单的写法,即将两个声明结合在一条语句中。例如:

```
module mux8 (y, a, b, en);
    output reg [7:0] y;
    input wire [7:0] a, b;
    input wire en;
```

16. ANSI 风格的输入和输出声明语句

Verilog-1995 标准使用早期的 Kernighan 和 Ritchie C 语言的语法来定义模块端口，端口的次序在小圆括号内定义，而端口的声明在小圆括号之后。

在 Verilog-1995 标准中，任务和函数定义时不用写出圆括号和其内部的端口列表，只需要用输入输出声明语句的排列次序来定义输入输出端口的次序。Verilog-2001 标准改进了模块、任务和函数的输入/输出序列定义，将端口声明语句写在包含输入/输出端口序列的圆括号内，使其更接近于 ANSI C 语言。

```
module mux8 (output reg [7:0] y,
             input wire [7:0] a,
             input wire [7:0] b,
             input wire en );
function [63:0] alu (input [63:0] a, b, input [7:0] opcode );
```

17. 寄存器声明时进行初始化赋值

Verilog-2001 标准允许在变量声明时对其进行初始化，可以不需要用专门的变量初始化进程块来对变量进行初始化。采用这种方式进行的变量初始化与在初始化进程块中进行初始化一样，在仿真的 0 时刻执行。例如，在 Verilog-1995 标准中：

```
reg clock;
  initial
    clk =0;
```

在 Verilog-2001 标准中可以写为：

```
reg clock = 0;
```

这两种表达方式的实际含义是一样的。

18. 将寄存器改变为变量

自 1984 年 Verilog 语言诞生以来，register 一词就一直用来描述 Verilog 语言中变量的一种类型。register 并不是一个关键字，只是一种数据类型(reg、integer、time、real 和 realtime)的名称。register 一词的使用通常会给 Verilog 初学者带来困扰，他们通常认为 register 和硬件中的寄存器(Flip-flops)概念是一致的。因此，Verilog-2001 标准中将 register 一词改为 variable。

19. 对条件编辑的改进

Verilog-1995 标准支持`ifdef、`else 和`endif 等条件编译语句。Verilog-2001 标准中增加

了一些条件编译控制语句,包括`ifndef 和`elsif。

20. 文件和行编译指示

Verilog 工具需要不断地跟踪 Verilog 源代码的行号和文件名。Verilog 的可编程语言接口(PLI) 可以取得并利用行号和源文件名信息,显示程序运行的错误信息时,也需要知道这些信息。但是,如果 Verilog 源码经过其他工具的处理,源码的行号和文件名信息可能会丢失。Verilog-2001 标准中增加了一个`line 编译指示,用来指定源码的行号和文件名。这样可以使源码文件中的指定位置在经过其他工具的修改(例如增加或删除一行)后,也能够保持不变。

21. 属性

创建 Verilog 语言的最初目的是建立一种用于数字仿真的硬件描述语言。随着仿真器之外的其他工具采用 Verilog 作为设计输入,这些工具需要 Verilog 语言能够加入跟指定工具有关的信息和命令。在 Verilog-1995 标准中没有这一机制,只能通过一些非标准的方式来实现,例如通过 Verilog 注释语句加入可控制综合器功能的命令。

Verilog-2001 标准中增加了一种机制,可以在 HDL 源代码中指定对象、语句或语句组的属性。这些属性称为 Attribute,可以翻译为属性。属性可以由包括仿真器在内的不同工具使用,控制工具的行为和操作。属性包含在两个 * 符号之间。属性可以应用于对象的所有实例调用,也可以只应用于对象的某一个实例调用。属性可以指定为数值(包括字符串),某对象的每个实例调用可以重新定义其属性值。

Verilog-2001 标准没有定义任何标准的属性,属性的名字和数值由工具厂商或其他标准定义。

下面是综合工具如何使用属性的例子:

```
(* parallel case *) case (1'b1)     //1 位独热码的有限状态机(FSM)
                    state[0]: ...
                    state[1]: ...
                    state[2]: ...
                  endcase
```

四、提高了 ASIC/FPGA 应用的正确性

Verilog 语言创建于 2 或 5 微来设计的时代,随着硅工艺水平的进步和设计方法学的演进,Verilog 语言也在发展。Verilog-2001 标准继续了这一发展,特别针对现在和未来的深亚

微米设计作了改进。

1. 检测脉冲的传播错误

Verilog-1995 标准可以对脚对脚(pin-to-pin)的路径延迟提供根据事件(on-event)的脉冲传播错误模型。脉冲是模型路径输入端上的扰动,其维持的时间比它在路径上的延迟小。如果使用了 on-event 的脉冲传播模型,输入脉冲的上升沿和下降沿传播到路径的输出端就变成了不定态(即逻辑值 X),其波形(即起始和维持时间内的不定值)看起来像是输入脉冲已经传播到输出。

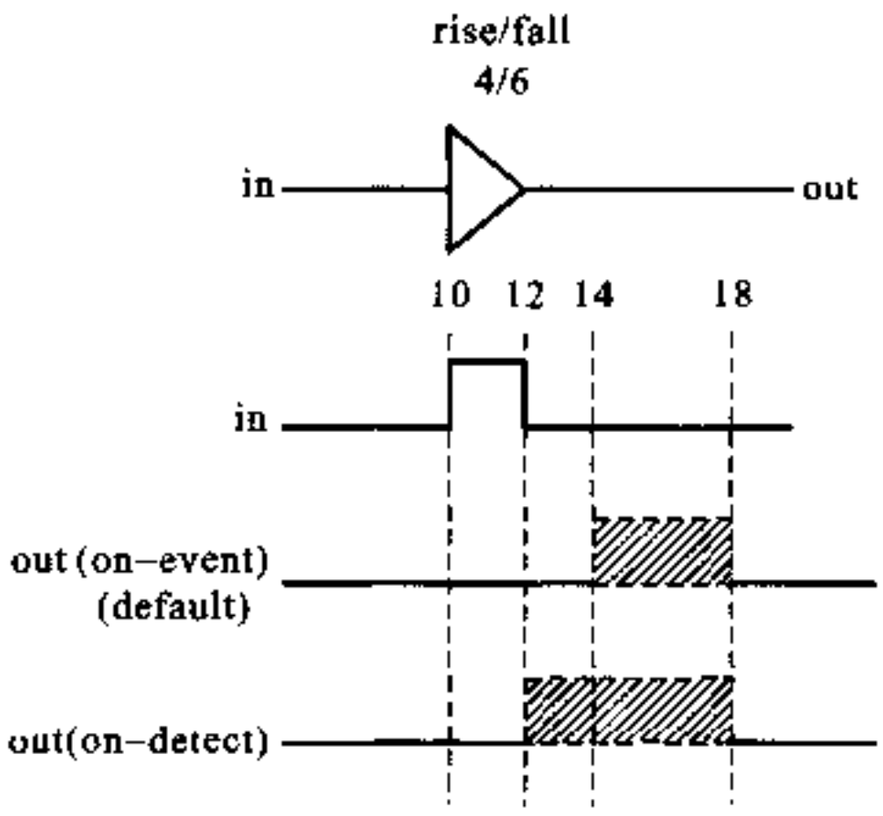


图 3.1 示例波形图

Verilog-2001 标准增加了 on-detect 脉冲传播错误模型。当输入信号为干扰脉冲时,设置 on-detect 是一种更保险的将输出设置为不定态的方法。与 on-event 模型相比,on-detect 也是将脉冲从前沿开始变成不定态,到脉冲后沿结束,不同处在于一旦检测到脉冲,前沿立刻开始变为不定态。在 Verilog 的 specify 声明语句中可以使用两个新的关键字 pulsestyle_onevent 和 pulsestyle_on-detect,来显式地指定脉冲错误传播方式是 on-event 还是 on-detect。on-event 是脉冲传播错误的默认类型。举例说明如下(波形图如图 3.1 所示):

```
specify
    pulsestyle_ondetect out;
    (in => out) = (4,6);
endspecify
```

2. 负脉冲检测

当脉冲的下降沿比上升沿先到即为负脉冲,由于路径延迟大于脉冲维持时间,输入负脉冲也可能得到不定态(逻辑值为 X)的扰动输出。在 Verilog-1995 标准中,负脉冲通常被忽略。Verilog-2001 标准提供了一种机制,可将不定态传播到输出,表明负脉冲已经产生,增加了两个关键字 showcancelled 和 noshowcancelled,在 specify 声明语句定义时,可以用它们让负脉冲传播功能启动或取消。举例说明如下:

```
specify
    showcancelled out;
    (a => out) = (2,3);
```

```
    (b => out) = (4,5);
endspecify
```

3. 新的定时约束检查

Verilog-2001 标准增加了几个新的定时约束检查,可以建立更精确的深亚微米定时模型。这些新的定时检查系统任务包括$ removal、$ recrem、$ timeskew 和$ fullskew。本书不全面介绍和探讨这些定时检查,对细节感兴趣的读者请参阅 Verilog-2001 标准。

4. 负定时约束

Verilog-2001 标准给$ setuphold 定时约束系统任务增加了 4 个参数,用这些参数可以精确地指定负电平的建立和保持时间。这两个时间参数用于定义沿冲突窗口(相对于参考信号沿)的位置,在这段时间中,数据必须是稳定的。建立和保持时间为正表示这一窗口跨过其参考的信号沿。相反,建立和保持时间为负则意味着时间冲突窗口位移到参考信号沿的前面或后面。这种情形在真实器件中是很可能发生的,因为器件内部时钟和数据信号路径延迟的不一致总是有可能存在的。$ setuphold 的新参数可加在 Verilog-1995 标准所定义的$ setuphold 参数表的后面。新的参数是可选的。如果不指定这些参数,也没有关系,$ setuphold 的语法仍旧与 Verilog-1995 标准的规定兼容。

新的定时检查系统任务 $ recrem 是 $ recovery 和$ removal 两个系统任务的结合,可以处理负值,其语法与 $ setuphold 类似。若想了解如何使用这些定时检查来指定定时约束为负值的细节,请参考 Verilog-2001 标准。

5. 提高了对 SDF(标准延迟文件)的支持

Verilog-2001 标准增加了一节,详细描述了 SDF 文件中的延迟如何与 Verilog 语言中的延迟相对应。其内容是建立在最新的 SDF 标准(即 IEEE Std 1497 – 1999 [3])基础之上的。

最新的 SDF 标准中包括了延迟标签,可以为 Verilog 程序提供延迟标注的手段。为了支持 SDF 标签,对 Verilog 语法作了一点改动。在 Verilog-1995 标准中,specparam 常数只能在 specify 声明语句(指定块)中定义。而 Verilog-2001 标准允许在模块层次声明和使用 specparam 声明语句常数。

6. 扩展了 VCD 文件

Verilog-1995 标准中定义了一个标准的 4 状态逻辑 VCD(数值变化存储)的文件格式。$ dumpvars 和其他相关的系统任务用于创建和控制 VCD 文件。Verilog-2001 标准对 VCD 文件格式作了一些扩展,在 Verilog 端口变化、线路连接强度(Net Strength)变化以及仿真结束时间等方面增加了更多的细节。为此,Verilog-2001 标准中定义了一些新的系统函数,它们

可以用来创建和控制扩展的 VCD 文件，它们是：$dumpports、$dumpportsall、$dumpportsoff、$dumpportson、$dumpportslimit 和$dumpportsflush。

五、编程语言接口(PLI)方面的改进

Verilog-2001 标准对原 Verilog 编程语言接口(PLI)部分作了大量改进。这些改进可分为 3 部分：

(1) 新 PLI 增加了许多新的功能。

(2) 新的 PLI 能支持 Verilog-2001 标准对 Verilog-1995 标准的每个改进。

(3) 对 Verilog-1995 PLI 标准作了全面的清理。

Verilog PLI 标准包括 3 个 C 功能库：TF、ACC 以及 VPI。TF 和 ACC 库是 Verilog PLI 的早期版本，新标准为了兼容旧标准(即 Verilog-1995 标准)保留了这两个库。VPI 库是最新版 PLI 标准所用的库，与旧库比较有许多优点。

Verilog-2001 标准清理和更正了旧的 TF 和 ACC 库中的许多定义，但并没有给 TF 和 ACC 库加入任何新的功能。对 Verilog PLI 的所有改进都体现在 VPI 库中。其中包括支持 Verilog 语言的许多新特性，以及提供了 6 个 VPI 新子程序：vpi_control()、vpi_get_data()、vpi_put_data()、vpi_get_userdata()、vpi_put_userdata()和 vpi_flush()。对这些 VPI 新子程序细节感兴趣的读者，请参考 Verilog-2001 标准。

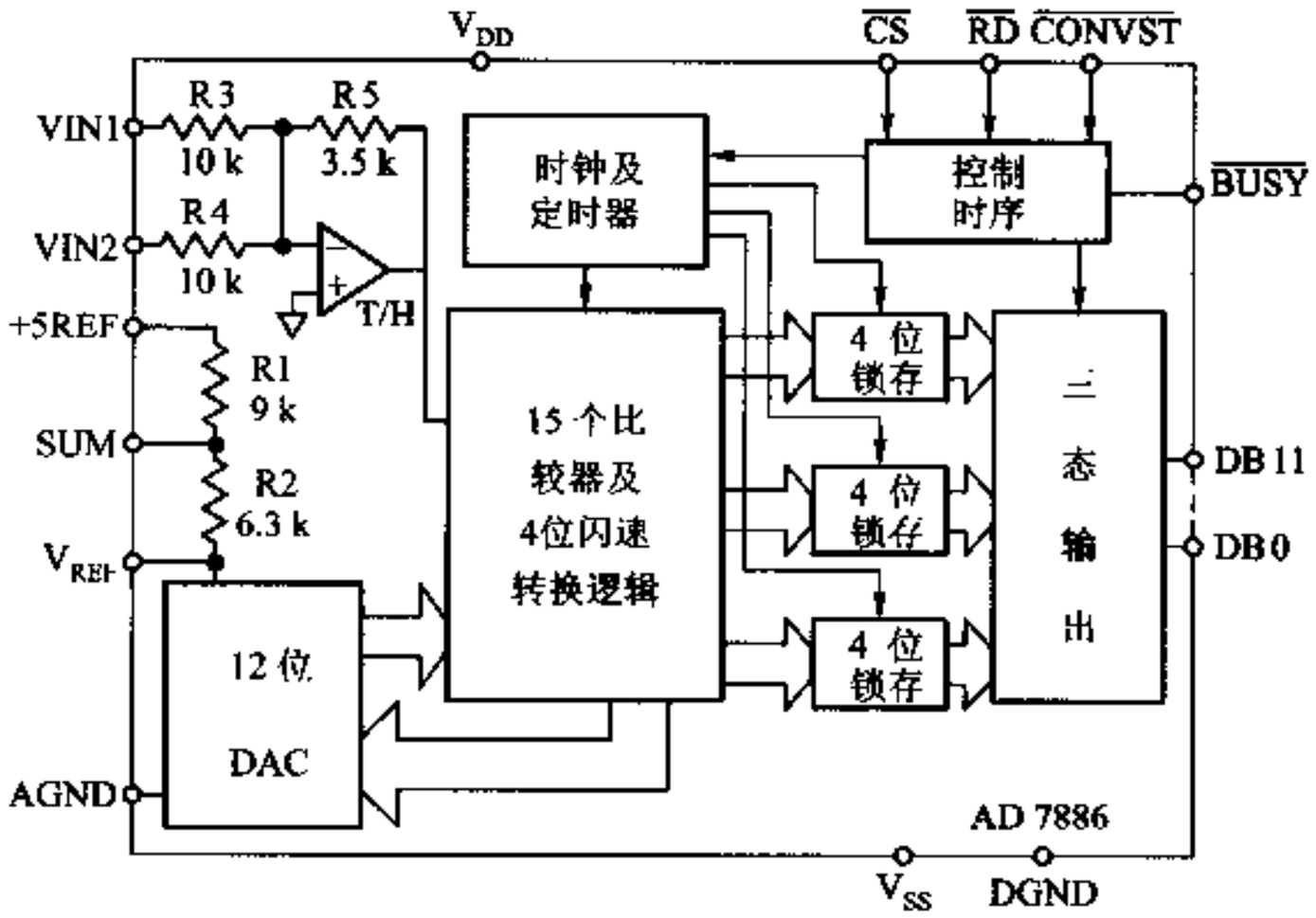
六、总　结

Verilog-2001 标准已经编写完毕，并且于 2001 年 3 月得到了 IEEE 官方的正式批准。Verilog-2001 标准对 Verilog 语言作了许多重要的改进，提供了强大的结构，可以编写可重复使用的、可升级的模型以及 IP 模型，并可给出精确的深亚微米电路的时间特性。用 Verilog 进行设计的工程师如果使用支持 Verilog-2001 标准的综合和仿真工具将从中得到极大的便利。

附录一　A/D 转换器的 Verilog HDL 模型和建立模型所需要的技术参数

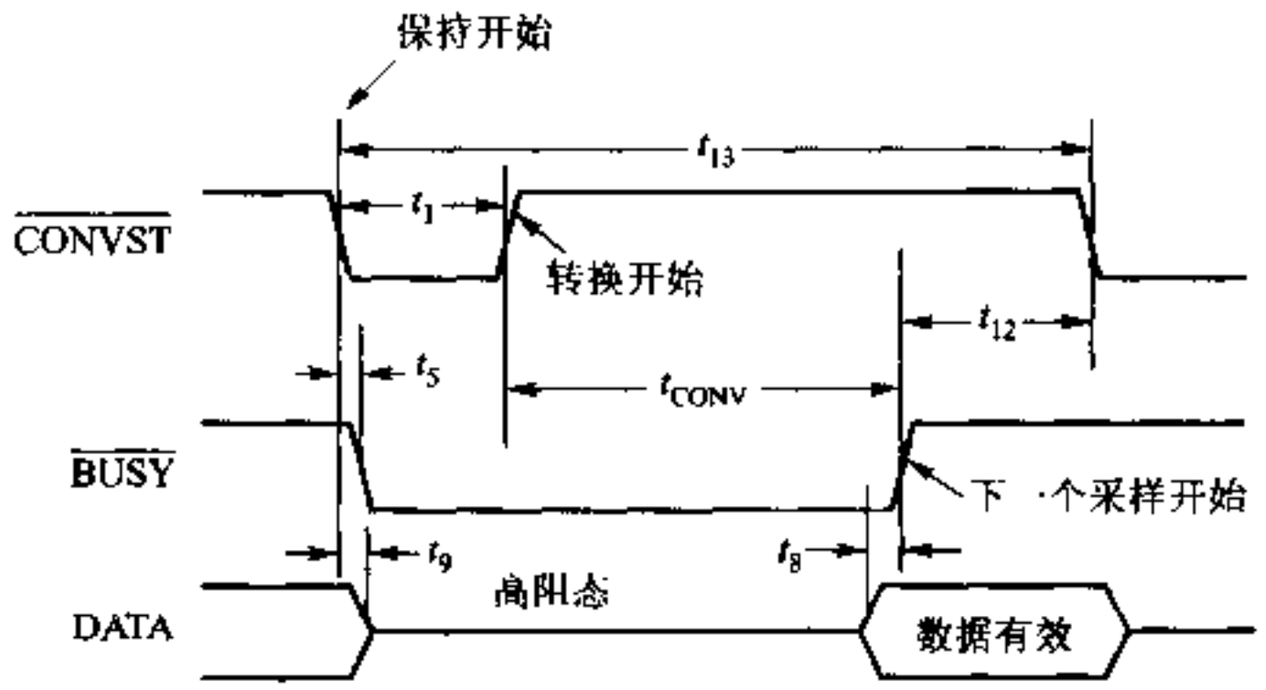
我们所用的 A/D 转换器是 AD7886。

AD7886 的时序控制有两种方法:第一种是 *CS* 和 *RD* 输入信号控制 AD7886 三态输出,如图附 1.1 所示,但 A/D 转换中三态输出封锁,这种方法适合微处理器在 AD7886 转换结束后直接把数据读出;第二种是 *CS* 和 *RD* 接到低电平,启动 A/D 转换开始后,数据线输出封锁,直到转换结束,数据输出才有效,如图附 1.2 所示,这种方法可以用 A/D 转换结束 BUSY 的上升沿触发外部锁存器锁存数据。



图附 1.1　AD7886 的功能图

在上述两种时序中,AD7886 进行转换都是由 *NCONVST* 控制的。*NCONVST* 的下降沿使采保开始跟踪信号,直到 *NCONVST* 上升沿来了,ADC 才进行转换。*NCONVST* 低脉宽度决定了跟踪 - 保持的建立时间。在 A/D 转换过程中,*BUSY* 输出位低,转换结束,*BUSY* 变为高,表示可以取走转换结果。在本设计中,我们用第二种时序控制 AD7886 工作。



图附 1.2　A/D 转换启动和数据读出时序($CS = RD = 0$)

A/D 转换器的 Verilog HDL 行为模型如下：

```
//------------------------------ adc.v ------------------------------
`timescale 100ps/100ps
module adc (nconvst, nbusy, data);
    input       nconvst;     //A/D 启动脉冲 ST,即上图中
    output      nbusy;       //A/D 工作标志,即上图中
    output      data;        //数据总线,从 AD.DATA 文件中读取数据后经端口输出
    reg[7:0]    databuf,i;   //内部寄存器
    reg         nbusy;
    wire[7:0]   data;
    reg[7:0]    data_mem[0:255];
    reg         link_bus;
    integer     tconv,
                t5,
                t8,
                t9,
                t12;
    integer     wideth1,
                wideth2,
                wideth;
//时间参数定义(依据 AD7886 手册)
always @(negedge nconvst)
```

```
        begin
            tconv =9500 + |$random| % 500;      //(type 950ns, max 1000ns)
                                                //Conversion Time
            t5 = |$random| % 1000; //(max 100ns) CONVST to BUSY Propagation Dlay
                                                //CL = 10pf
            t8 = 200;                           //(min 20ns) CL=20pf Data Setup
Time Prior to BUSY
                                                //(min 10ns) CL=100pf
            t9 = 100 + |$random| % 900;      //(min 10ns, max 100ns) Bus
                                                //Relinquish Time After CONVST
            t12 = 2500; //(type) BUSY High to CONVST Low, SHA Acquisition Time
        end
    initial
        begin
            $readmemh("adc.data",data_mem);//从数据文件 adc.data 中读取数据
            i = 0;
            nbusy = 1;
            link_bus = 0;
        end
assign data = link_bus? databuf:8'bzz;      //三态总线

/*-----------------------------------------------------------------------------
在信号 nconvst 的负跳降沿到来后,隔 t5 秒 nbusy 信号置为低,tconv 是 A/D 将模拟信号
转换为数字信号的时间,在信号 nconvst 的正跳降沿到来后经过 tconv 时间后,输出 nbusy
信号变为高。
-----------------------------------------------------------------------------*/
    always @(negedge nconvst)
        fork
        #t5 nbusy =0;
            @(posedge nconvst)
                begin
                    #tconv nbusy =1;
                end
        join
```

```
/*-----------------------------------------------------------------------------
nconvst 信号的下降沿触发,经过 t9 延迟后,把数据总线输出关闭置为高阻态,如图附 1.2 示。
nconvst 信号的上升沿到来后,经过(tconv - t8)时间,输出一个字节(8 位数据)到
databuf,该数据来自于 data_mem。而 data_mem 中的数据是初始化时从数据文件 AD.DATA
中读取的。此时应启动总线的三态输出。
-----------------------------------------------------------------------------*/
    always @(negedge nconvst)
        begin
            @(posedge nconvst)
                begin
                    #(tconv - t8) databuf = data_mem[i];
                end
            if(wideth <10000 && wideth >500)
                begin
                    if(i ==255) i =0;
                    else i = i +1;
                end
        else i = i;
end
//在模数转换期间关闭三态输出,转换结束时启动三态输出
always @(negedge nconvst)
  fork
      #t9 link_bus = 1'b0;          //关闭三态输出,不允许总线输出
      @(posedge nconvst)
          begin
              #(tconv - t8) link_bus = 1'b1;
          end
    join
/*-----------------------------------------------------------------------------
当 nconvst 输入信号的下一个转换的下降沿与 nbusy 信号上升沿之间时间延迟小于 t12 时,
将会出现警告信息,通知设计者请求转换的输入信号频率太快,A/D 器件转换速度跟不上。仿真
模型不仅能够实现硬件电路的输出功能,同时能够对输入信号进行检测,当输入信号不符合手
册要求时,显示警告信息。
-----------------------------------------------------------------------------*/
    //检查 A/D 启动信号的频率是否太快
    always @(posedge nbusy)
```

```
    begin
        #tJ2;
        if (! nconvst)
            begin
                $display("Warning! SHA Acquisition Time is too short!");
            end
        //else $display(" SHA Acquisition Time is enough! ");
    end
//检查 A/D 启动信号的负脉冲宽度是否足够和太宽
always @(negedge nconvst)
    begin
        wideth = $time;
        @(posedge nconvst) wideth = $time-wideth;
        if (wideth <= 500 || wideth > 10000)
            begin
                $display("nCONVST Pulse Width = % d",wideth);
                $display("Warning! nCONVST Pulse Width is too narrow or
                        too wide!");
                //$stop;
            end
    end
endmodule
```

附录二　2K×8 位异步 CMOS 静态 RAM HM-65162 模型

```
/ ************************************************************************************************
*       File Name                       : sram. v                                                 *
*       Function                        : 2K * 8bit Asynchronous CMOS StaticRAM                   *
************************************************************************************************ /
/ ************************************************************************************************
*       Module Name                     : sram                                                    *
*       Description                     : 2K * 8bit Asynchronous CMOS StaticRAM                   *
*       Reference                       : HM-65162reference book                                  *
************************************************************************************************ /
/ ************************************************************************************************
* sramis a Verilog HDL model for HM-65162,2K * 8bit Asynchronous CMOS Static RAM. It is used in
simulation to                                                                                     *
* substitute the real RAM to verify whether the writing or reading of the RAM isOK. This moduleis a
behavioral model                                                                                  *
* for simulation only,not synthesizable. It'swritingand reading function are verified.            *
************************************************************************************************ /

//------------------------- sram.v -------------------------
module sram(Address, Data, SRG,SRE, SRW);
    input [10:0] Address;
    input     SRG,//Output enable
              SRE,//Chipenable
              SRW;//Write enable
    inout [7:0]     Data;//Bus
    wire [10:0]     Addr = Address;
    reg [7:0]       RdData;
    reg [7:0]       SramMem [0:'h7ff];
    reg             RdSramDly, RdFlip;
    wire [7:0]      FlpData,Data;
```

```
reg WR_flag;  //To judge the signals according to the specification of
              //HM-65162
integer       i;
wire          RdSram = ~SRG & ~SRE;
wire          WrSram = ~SRW & ~SRE;
reg [10:0]    DelayAddr;
reg [7:0]     DelayData;
reg           WrSramDly;
integerfile;
assign FlpData = (RdFlip) ? ~RdData : RdData;
assign Data = (RdSramDly) ? FlpData:'hz;
********************************* parameters of read circle ********************************
                          //参数序号、最大或最小、参数含义
parameter TAVQV=90,   //2    (max)    Address access time
          TELQV=90,   //3    (max)    Chip enable access time
          TELQX=5,    //4    (min)    Chip enable output enabletime
          TGLQV=65,   //5    (max)    Output enable access time
          TGLQX=5,    //6    (min)    Output inable output enable time
          TEHQZ=50,   //7    (max)    Chip enable output disable time
          TGHQZ=40,   //8    (max)    Output enable output disable time
          TAVQX=5;    //9    (min)    Output hold from address change
********************************* parameters of write circle *******************************
parameter TAVWL=10,   //12   (min)    Address setuptime,
          TWLWH=55,   //13   (min)    Chip enable pulse setup time,
                                       //write enable pluse width,
          TWHAX=15,   //14   (min10)  Writcenable read setup time,
                                       //读上升沿后地址保留时间
          TWLQZ=50,   //16   (max)    Write enable output disable time
          TDVWH=30,   //17   (min)    Data setup time
          TWHDX=20,   //18   (min15)  Data hold time
          TWHQX=20,   //19   (min0)   Writeenable output enable time,0
          TWLEH=55,   //20   (min)    Write enable pulse setup time
          TDVEH=30,   //21   (min)    Chip enable data setup time
          TAVWH=70;   //22   (min65)  Address validto end of write
initial
  begin
```

```
        file=$fopen("ramlow.txt");
        if(! file)
    begin
        $display("Could not open the file.");
        $stop;
    end
end
initial
    begin
        for(i=0 ; i<'h7ff ; i=i+1)
        SramMem[i] = i;
    //$monitor($time,,"DelayAddr=% h,DelayData=% h",DelayAddr,DelayData);
    end
    initial RdSramDly = 0;
    initial WR_flag=1;
/************************************ READ CIRCLE ************************************/
    always @(posedge RdSram) #TGLQX RdSramDly = RdSram;
    always @(posedge SRW) #TWHQX RdSramDly = RdSram;
    always @(Addr)
        begin
        #TAVQX;
        RdFlip = 1;
        #(TGLQV - TAVQX);          //address access time
        if(RdSram) RdFlip = 0;
        end
always @(posedge RdSram)
        begin
        RdFlip = 1;
        #TAVQV;                    //Output enable accesstime
        if(RdSram) RdFlip = 0;
        end
always @(Addr)          #TAVQXRdFlip = 1;
always @(posedge SRG)   #TEHQZRdSramDly = RdSram;
always @(posedge SRE)   #TGHQZRdSramDly = RdSram;
```

```
    always @(negedge SRW)   #TWLQZRdSramDly = 0;
    always @(negedge WrSramDly or posedge RdSramDly) RdData = SramMem[Addr];
/************************************ WRITE CIRCLE ************************************/
    always @(Addr)     #TAVWL DelayAddr = Addr;     //Address setup
    always @(Data)     #TDVWH DelayData = Data;     //Data setup
    always @(WrSram)   #5 WrSramDly =WrSram;
    always @(Addr or Data or WrSram) WR_flag=1;
    always @(negedge SRW )
        begin
            #TWLWH;          //Write enablepulse width
            if (SRW)
              begin
                  WR_flag=0;
                  $display("ERROR! Can't write!
                  Write enable time (W) is too short!");
              end
        end
    always @(negedge SRW )
        begin
            #TWLEH;          //Write enablepulse setup time
            if (SRE)
              begin
                  WR_flag=0;
                  $display("ERROR! Can't write! Write enable
                            pulse setup time (E) is too short!");
              end
        end
    always @(posedge SRW )
        begin
            #TWHAX;          //Write enable read setup time
            if(DelayAddr !== Addr)
              begin
                  WR_flag=0;
                  $display("ERROR! Can't write!
                            Write enable read setuptime istoo short!");
```

```
        end
    end
always @(Data)
  if (WrSram)
    begin
        #TDVEH;          //Chip enable data setup time
        if (SRE)
          begin
              WR_flag=0;
              $display("ERROR! Can't write!
                        Chip enable Data setup time is too short!");
          end
    end
always @(Data)
  if (WrSram)
    begin
        #TDVEH;
        if (SRW)
          begin
              WR_flag=0;
              $display("ERROR! Can't write!
                        Chip enable Data setup time is too short!");
          end
    end
always @(posedge SRW )
    begin
        #TWHDX;          //Data hold time
        if(DelayData !== Data)
        $display("Warning! Data hold time is too short!");
    end
always @(DelayAddr or DelayData or WrSramDly)
  if (WrSram &&WR_flag)
    begin
        if(! Addr[5])
          begin
              #15 SramMem[Addr]=Data;
```

```
        //$display("mem[%h] = %h",Addr,Data);
        $fwrite(file,"mem[%h] = %h ",Addr,Data);
        if(Addr[0]&&Addr[1]) $fwrite(file,"\n");
    end
else
    begin
        $fclose(file);
        $display("Please check the txt.");
        $stop;
    end
        end

    endmodule
```

[练习题]　参考另外一种 A/D 变换器手册，模仿本示范题的 A/D 虚拟模块，编写该 A/D 变换器全功能的虚拟模块；参考另一种静态 RAM 手册，编写编写完整的静态 RAM 虚拟模块；根据这两种新器件的时序设计符合工程要求的卷积器。

高 ◆ 等 ◆ 学 ◆ 校 ◆ 教 ◆ 材

Verilog HDL

实验练习与语法手册

ISBN 7-04-017199-6

9 787040 171990 >

定价 18.60元