第5章 Verilog HDL 基础内容

CHAPTER 5

本章介绍 IEEE Std 1364—2005 版本的 Verilog HDL 基础内容，其中包括 Verilog HDL 的程序结构、Verilog HDL 要素、Verilog HDL 数据类型、Verilog HDL 表达式、Verilog HDL 分配、Verilog HDL 门级和开关级描述、Verilog HDL 行为描述语句、Verilog HDL 任务和函数、Verilog HDL 层次化结构、Verilog HDL 系统任务和函数，以及 Verilog HDL 编译指示语句。

5.1 Verilog HDL 程序结构

描述复杂的硬件电路，设计人员总是将复杂的功能划分为简单的功能，模块是提供每个简单功能的基本结构。设计人员可以采取“自顶向下”的思路，将复杂的功能模块划分为低层次的模块。这一步通常由系统级的总设计师完成，而低层次的模块则由下一级的设计人员完成。自顶向下的设计方式有利于系统级的层次划分和管理，提高了效率，降低了成本。

使用 Verilog HDL 描述硬件的基本设计单元是模块(module)。复杂的电子电路的构建，主要是通过模块间的相互连接调用来实现的。Verilog HDL 中的模块类似 C 语言中的函数，它能够提供输入、输出端口，可以通过例化调用其他模块，也可以被其他模块例化调用。模块中可以包括组合逻辑和时序逻辑。

模块是并行运行的，通常需要一个高层模块通过调用其他模块的实例来定义一个封闭的系统，包括测试数据和硬件描述。一个模块通过它的端口(输入/输出端口)为更高层的设计模块提供必要的连通性，但是又隐藏了其内部的具体实现。这样，在修改其模块的内部结构时不会对整个设计的其他部分造成影响。

图 5.1 给出了 Verilog 模块的程序结构。Verilog 模块由 module 和 endmodule 之间的语句定义，每个 Verilog 模块包括端口定义、数据类型说明和逻辑功能定义。其中，模块名是模块唯一的标识符，端口列表由模块各个输入、输出和双向端口组成。通过这些端口该模块可以与其他模块连接；数据类型说明用来指定模块内用到的数据对象是网络还是变量；逻辑功能定义是通过使用逻辑功能语句实现具体的逻辑功能。

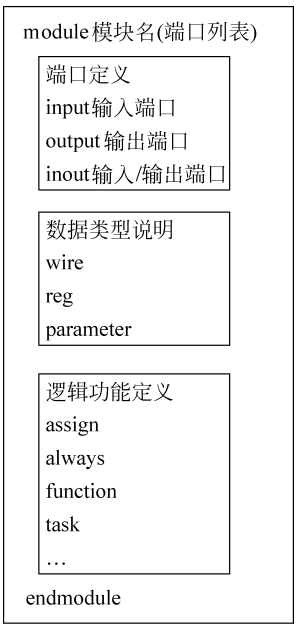

图 5.1 Verilog 模块的程序结构

Verilog HDL 具有下面的特征。

(1) 每个 Verilog HDL 源文件都以.v 作为文件扩展名。

(2) Verilog HDL 区分大小写,也就是说,大小写不同的标识符是不同的。

(3) Verilog HDL 程序的书写与 C 语言类似,一行可以写多条语句,也可以一条语句分成多行书写。

(4) 每条语句以分号结束,endmodule 语句后不加分号。

(5) 空白(新行、制表符和空格)没有特殊意义。


视频讲解

5.1.1 模块声明

模块声明包括模块名字、模块的输入和输出端口列表。模块定义的语法格式如下:

```
module<module_name>(port_name1,…,port_namen);
…
…
…
endmodule
```

其中,module_name 为模块名,是该模块的唯一标识; port_name * 为端口名,这些端口名之间使用“,”分隔。

5.1.2 模块端口定义

端口是模块与外部其他模块进行信号传递的通道(信号线),模块端口分为输入、输出或双向端口。

(1) 输入端口定义的语法格式如下:

```
input<input_port_name>, …,<other_inputs>…;
```

其中,input 为关键字,用于声明后面的端口为输入端口; input_port_name 为输入端口的名字; other_inputs 为用逗号分隔的其他输入端口的名字。

(2) 输出端口定义的语法格式如下:

```
output<output_port_name>,…,<other_outputs>…;
```

其中,output 为关键字,用于声明后面的端口为输出端口; output_port_name 为输出端口的名字; other_outputs 为用逗号分隔的其他输出端口的名字。

(3) 输入输出端口(双向端口)的定义格式如下:

```
inout<inout_port_name>,…,<other_inouts>…;
```

其中,inout 为关键字,用于声明后面的端口为输入输出类型的端口; inout_port_name 为输入输出端口的名字; other_inouts 为用逗号分隔的其他输入/输出端口的名字。

在声明端口的时候,除了声明其输入/输出类型外,还需要注意以下几点。

(1) 在声明输入端口、输出端口或者输入/输出端口时,还要声明其数据类型。对于端口来说,可用的数据类型是网络类型或者 reg 类型。当没有明确指定端口类型时,端口默认为网络类型。

(2) 可以将输出端口重新声明为 reg 类型。无论是在网络类型说明还是 reg 类型说明中,网络类型或 reg 类型必须与端口说明中指定的宽度相同。

(3) 不能将输入端口和双向端口指定为 reg 类型。

【例 5.1】 端口声明实例如代码清单 5-1 所示。

代码清单 5-1 端口声明的 Verilog HDL 描述

```
module test(a,b,c,d,e,f,g,h);
input[7:0] a;                    //没有明确的说明——网络是无符号的
```

```
input[7:0] b;
input signed[7:0] c;          //明确的网络说明——网络是有符号的
input signed[7:0] d;
output[7:0] e;                //没有明确的网络说明——网络是无符号的
output[7:0] f;
output signed[7:0] g;         //明确的网络说明——网络是有符号的
output signed[7:0] h;
wire signed[7:0] b;           //从网络声明中,端口 b 继承了有符号的属性
wire[7:0] c;                  //网络 c 继承了来自端口的有符号的属性
reg signed[7:0] f;            //从 reg 声明中,端口 f 继承了有符号的属性
reg[7:0] g;                   //reg 类型端口 g 继承了有符号的属性
endmodule
```

注意,在 Verilog HDL 中,也可以使用 ANSI C 风格进行端口声明。这种风格声明的优点是避免了在端口列表和端口声明语句中重复端口名。如果声明中未指明端口的数据类型,那么默认端口为 wire 数据类型。下面给出上述模块的另一种声明方法。

【例 5.2】 ANSI C 风格的端口说明实例如代码清单 5-2 所示。

代码清单 5-2 ANSI C 风格端口的 Verilog HDL 描述

```
module test (
input[7:0] a,
input signed[7:0] b, c, d,    //多个共享相同属性的端口,可以一起声明
output[7:0] e,                //必须在每个端口声明中,单独声明每个端口的属性
output reg signed[7:0] f, g,
output signed[7:0] h) ;       //在模块的其他地方重新声明端口都是非法的
endmodule
```

5.1.3 逻辑功能定义


视频讲解

逻辑功能定义是 Verilog HDL 程序结构中最重要的部分,逻辑功能定义用于实现模块中的具体功能。在逻辑功能定义部分,可以使用多种方法实现逻辑功能,主要包含以下 4 种。

1. 分配语句实现逻辑定义


视频讲解

分配语句是最简单的逻辑功能描述,由 assign 语句定义逻辑功能。

【例 5.3】 分配语句用于逻辑功能定义的例子。

```
assign F = ~((A&B)|(~(C&D)));
```

2. 模块调用

所谓模块调用,是指从模块模板生成实际的电路结构对设计中其他对象的操作,这样的电路结构对象称为模块实例,模块调用也称为实例化(例化)。每个实例都有它自己的名字、变量、参数和 I/O 接口。一个 Verilog 模块可以由任意多个其他模块调用。

在 Verilog HDL 中,不能嵌套定义模块,即在一个模块的定义内不能包含另一模块的定义;但是可以包含对其他模块的复制,即调用其他模块的例化。模块的定义和模块的例化是两个不同的概念。在一个设计中,只有通过模块调用(例化)才能使用一个模块。

【例 5.4】 顶层模块调用底层模块的例子如代码清单 5-3 所示。

代码清单 5-3 顶层模块调用底层模块的 Verilog HDL 描述

```
module top;                   // module 关键字定义模块 top
reg clk;                      // reg 关键字定义 reg 类型变量 clk
reg [0:4] in1;                // reg 关键字定义 reg 类型变量 in1,宽度为 5 位
reg [0:4] in2;                // reg 关键字定义 reg 类型变量 in2,宽度为 5 位
wire [0:4] o1;                // wire 关键字定义网路 o1,宽度为 5 位
wire [0:4] o2;                // wire 关键字定义网络 o2,宽度为 5 位
vdff m1(o1, in1, clk);        // 调用/例化模块 vdff,将其例化为 m1
```

```
vdff m2(o2, in2, clk);          // 调用/例化模块 vdff,将其例化为 m2
endmodule                       // endmodule 标识模块 top 的结束
```

3. 在 always 过程赋值

always 块经常用于描述时序逻辑电路,也可描述组合逻辑电路。下面先给出一个使用 always 过程实现计数器的例子,在后续章节会详细地说明 always 过程。

【例 5.5】 always 过程实现计数器的例子如代码清单 5-4 所示。

代码清单 5-4 always 过程实现计数器的 Verilog HDL 描述

```
always @(posedge clk)           // always 关键字定义过程语句,括号内为敏感信号 clk
begin                           // begin 关键字标识过程语句的开始,类似 C 语言的"{"
  if(reset)                     // if 关键字标识条件语句,条件为 reset 信号为逻辑"1"
     out <= 0;                  // out 复位为 0
  else                          // else 关键字标识条件语句 if…else,条件为 clk 上升沿
     out <= out + 1;            // out 加 1,结果分配给 out
end                             // end 关键字标识过程语句的结束
```

4. 模块和函数调用

模块调用和函数调用非常相似,但是在本质上又有很大差别。

(1) 一个模块代表拥有特定功能的一个电路块,每当在其他模块内调用一次该模块时,就会在其他模块内复制一次该模块所表示的电路结构(即生成被调用模块的一个实例)。

(2) 模块调用不能像函数调用一样具有"退出调用"的操作,因为硬件电路结构不会随着时间而发生变化,被复制的电路块将一直存在。

后续章节会详细介绍函数和任务的声明和调用方法。下面给出一个函数调用的例子,用于帮助读者了解函数调用方法。

【例 5.6】 函数调用的例子如代码清单 5-5 所示。

代码清单 5-5 函数调用的 Verilog HDL 描述

```
module tryfact;
//定义函数
function automatic integer factorial;
input[31:0] operand;
integer i;
if(operand >= 2)
    factorial = factorial (operand - 1) * operand;
else
    factorial = 1;
endfunction
//测试函数
integer result;
integer n;
initial
 begin
    for(n = 0; n <= 7; n = n + 1)
    begin
        result = factorial(n);
         $display(" % 0d factorial = % 0d", n, result);
    end
  end
endmodule
```

注:在 Verilog HDL 中,begin 和 end 关键字的功能类似于 C 语言中的{ },在 begin 和 end 的中间是一系列逻辑行为描述语句。


视频讲解

5.1.4 设计实例一:Verilog HDL 结构框架的设计与实现

本节将通过一个简单的设计实例说明 Verilog 模块的完整框架结构,以帮助读者理解一

个完整 Verilog 模块的构成形式。在该例子中，包含设计文件 top. v 和仿真文件 test. v，如代码清单 5-6 和代码清单 5-7 所示。

代码清单 5-6 top. v 文件

```
module top(                          // module 关键字定义模块 top
input clk,                           // input 关键字定义输入端口 clk,类型默认为 wire
input rst,                           // input 关键字定义输入端口 rst,类型默认为 rst
input [3:0] a,                       // input 关键字定义输入端口 a,宽度为 4 位,类型默认为 wire
input [3:0] b,                       // input 关键字定义输入端口 b,宽度为 4 位,类型默认为 wire
output z                             // output 关键字定义输出端口 z,宽度为 1 位,类型默认为 wire
    );
assign z = compare(a_buf,b_buf); // assign 关键字定义分配语句,将函数比较结果分配给输出 z

reg [3:0] a_buf,b_buf;               // reg 关键字定义 reg 类型变量 a_buf, b_buf

function [0:0] compare;              // function 关键字定义函数 compare
input [3:0] m,n;                     // 函数的输入为 m 和 n,默认类型为 wire 网络,宽度为 4 位
begin                                // begin 关键字表示函数体的开始,类似 C 语言的"{"
  if(m >= n)                         // if 关键字定义条件语句,如果 m 大于或等于 n
     compare = 1'b1;                 // 将比较结果 1'b1 赋值给函数的返回值 compare,同函数名
  else                               // else 关键字构成 if…else,如果 m 小于 n
     compare = 1'b0;                 // 将比较结果 1'b0 赋值给函数的返回值 compare
end                                  // end 关键字表示函数体的结束,类似 C 语言的"}"
endfunction                          // endfunction 关键字定义函数的结束

always @(posedge clk or posedge rst) // always 关键字定义过程语句,()内为敏感信号 clk 和 rst 变化
begin                                // begin 标识过程语句的开始,类似 C 语言的"{"
if(rst)                              // if 关键字定义条件语句,如果 rst 输入为逻辑"1"(高电平)
begin                                // begin 标识过程语句的开始,类似 C 语言的"{"
    a_buf <= 4'b0000;                // reg 类型变量 a_buf 初始化为"0000",符号"<= "表示非阻塞赋值
    b_buf <= 4'b0000;                // reg 类型变量 b_buf 初始化为"0000",符号"<= "表示非阻塞赋值
end                                  // end 关键字表示 if(rst)条件的结束,类似 C 语言的"}"
else                                 // else 关键字定义条件语句
begin                                // begin 标识 else 条件的开始,类似 C 语言的"{"
    a_buf <= a;                      // 非阻塞赋值/分配语句,将 a 保存到 a_buf, 实际就是触发器
    b_buf <= b;                      // 非阻塞赋值/分配语句,将 b 保存到 b_buf,实际就是触发器
end                                  // end 关键字表示 else 条件的结束,类似 C 语言的"}"
end                                  // end 关键字表示 always 语句的结束
endmodule                            // endmodule 关键字表示模块 top 的结束
```

注：在配套资源\eda_verilog\example_5_1 目录下，用高云云源软件打开 example_5_1. gprj。

思考与练习 5-1 对该设计执行详细描述的过程，并打开 RTL 级原理图，查看该设计实现的电路结构。

思考与练习 5-2 对该设计执行综合的过程，并打开综合后的原理图，查看该设计实现的电路结构。

代码清单 5-7 test. v 文件

```
`timescale 1ns/1ps                   // "`"表示预编译指令,timescale 声明时间标度为 1ns/1ps
module test;                         // module 关键字声明模块 test
reg clk;                             // 声明 reg 类型变量 clk
reg rst;                             // 声明 reg 类型变量 rst
reg [3:0] d0;                        // 声明 reg 类型变量 d0,宽度为 4 位,范围为[3:0]
reg [3:0] d1;                        // 声明 reg 类型变量 d1,宽度为 4 位,范围为[3:0]
wire res;                            // 声明 wire 型数据 res,宽度为 1 位
top Inst_top(                        // Verilog 元件例化语句,调用 top 模块并将其例化为 Inst_top
  .clk(clk),                         // 将 top 模块的引脚 clk 连接到 test 模块的变量 clk
  .rst(rst),                         // 将 top 模块的引脚 rst 连接到 test 模块的变量 rst
  .a(d0),                            // 将 top 模块的引脚 a 连接到 test 模块的变量 d0
  .b(d1),                            // 将 top 模块的引脚 b 连接到 test 模块的变量 d1
```

```
    .z(res)                          // 将 top 模块的引脚 z 连接到 test 模块的网络 res
);
initial                              // initial 关键字声明初始化部分
begin                                // begin 关键字表示初始化部分的开始,类似 C 语言的"{"
 rst = 1'b1;                         // 变量 rst 初值设置为逻辑"1"(高电平),复位 top 模块
 #20;                                // 符号#表示延迟,逻辑"1"状态持续 20ns,ns 由 timescale 定义
 rst = 1'b0;                         // 变量 rst 初值设置为逻辑"0"(低电平),释放复位,复位无效
end                                  // end 关键字表示 initial 的结束,类似 C 语言的"}"

initial                              // initial 关键字声明初始化部分
begin                                // begin 关键字表示初始化部分的开始,类似 C 语言的"{"
 d0 = 6;                             // 给 d0 赋值/分配十进制数 6
 d1 = 4;                             // 给 d1 赋值/分配十进制数 4
 #105;                               // 符号"#"表示延迟,该赋值状态保持 105ns,ns 由 timescale 定义
 $display("%d compare with %d, result is %b",d0,d1,res);    //调用 Verilog 系统任务 display,
                                                            //打印信息
 d0 = 10;                            // 给 d0 赋值/分配十进制数 10
 d1 = 15;                            // 给 d1 赋值/分配十进制数 15
 #105;                               // 符号"#"表示延迟,该赋值状态保持 105ns,ns 由 timescale 定义
 $display("%d compare with %d, result is %b",d0,d1,res);   //调用 Verilog 系统任务 display,
                                                            //打印信息
 d0 = 13;                            // 给 d0 赋值/分配十进制数 13
 d1 = 13;                            // 给 d1 赋值/分配十进制数 13
 #105;                               // 符号"#"表示延迟,该赋值状态保持 105ns,ns 由 timescale 定义
$display("%d compare with %d, result is %b",d0,d1,res);   //调用 Verilog 系统任务 display,
                                                           //打印信息
end                                  // end 关键字表示 initial 的结束,类似 C 语言的"}"

always                               // always 关键字定义过程语句
begin                                // begin 关键字表示过程语句的开始,类似 C 语言的"{"
  clk = 1'b0;                        // 给 clk 赋值逻辑"0"(低电平)
  #10;                               // 符号"#"表示延迟,该赋值状态保持 10ns,ns 由 timescale 定义
  clk = 1'b1;                        // 给 clk 赋值逻辑"1"(高电平)
  #10;                               // 符号"#"表示延迟,该赋值状态保持 10ns,ns 由 timescale 定义
end                                  // end 关键字表示 always 过程语句的结束

endmodule                            // endmodule 关键字表示模块 test 的结束
```

注：在配套资源\eda_verilog\example_5_2 目录下,用 ModelSim 软件打开 postsynth_sim.mpf。

思考与练习 5-3 在 ModelSim 软件中对该设计执行综合后仿真,并观察波形窗口的波形,以及 Transcript 窗口中打印的信息。

读者通过上面的分析和观察过程,应该进一步地理解下面的一些本质问题。

(1) 为什么区分设计文件和仿真文件？显然,设计文件中的模块最终是要转换为数字逻辑电路中的组合逻辑和时序逻辑,并用高云 FPGA 内的逻辑资源来实现对应的逻辑功能,而仿真文件中的模块,显然是要生成测试向量对设计文件中实现的电路功能进行测试。

在仿真文件中,通过 Verilog 元件例化调用了设计文件中设计的模块,并在仿真文件中生成了测试向量对设计的模块进行测试。这些测试向量包括复位信号 rst、时钟信号 clk,以及输入的数据 d0 和 d1,并通过 Wave 窗口和 Transcript 窗口以两种不同的方式显示所设计模块的输出。

(2) Verilog HDL 的作用是什么？Verilog HDL 既可以用于设计文件中描述电路的模型,又可以在仿真文件中生成用于测试电路功能的测试向量。为了使读者更好地理解 Verilog HDL,将设计文件中用于描述电路模型的 Verilog HDL 称为数据流/RTL 级描述,简单理解就是所描述的电路结构模型将通过高云云源软件的综合工具转换为具体的组合逻辑和时序逻辑单元中的电路实现,并通过高云云源软件中的实现工具转换为高云 FPGA 中的布局和布线；将仿真文件中用于生成测试向量的 Verilog HDL 称为行为级描述。Verilog HDL 行为级描述很像我们

使用C语言编程一样。我们知道,C语言程序是运行在中央处理单元(central processing unit,CPU)上的,从另一个角度,就是用C语言编写测试程序测试CPU的功能正确与否。显然,用于生成测试向量的Verilog HDL描述不能生成硬件电路结构。

(3) 在整个设计中,仿真的作用又是什么呢?在整个设计过程中,需要仿真吗?这应该从两方面来说明。

① 既然在高云云源软件中已经通过设计文件对电路模型进行了描述,又通过设计综合和设计实现将其转换为电路结构,但是没有任何人有100%的把握一次就能把电路模型描述得完美无缺。那么,怎么办呢?就是在把设计下载到FPGA芯片之前,通过软件提供的仿真/模拟工具进行模拟,看看所有输入的组合和实际的输出结果是不是满足设计要求,这样就可以在把设计下载到FPGA芯片之前尽可能早地发现设计错误,并修正设计中出现的错误。

② 当设计下载到FPGA之后,如果发现设计功能不能满足设计要求,该如何处理呢?很多读者在没有接触在线逻辑分析仪工具之前,自然会想到,能不能通过软件提供的仿真/模拟功能把FPGA出现不正常输出所对应的输入组合在软件上复现出来,从而修正原始设计中出现的设计错误。

(4) 为什么将Verilog称为硬件描述语言(hardware description language,HDL),很多读者都说Verilog HDL和C语言很像,这点从表面上看似乎是对的,但是两者有着本质的区别。前面提到,C语言和Verilog HDL的行为级描述非常相似,但是C语言和Verilog HDL的RTL/数据流描述就不是一回事了。因为C语言本质上是软件,运行在CPU上,而Verilog HDL本质上是用来对组合逻辑和时序逻辑进行建模,最终是要转换为逻辑电路的基本元件,包括逻辑门、进位逻辑、多路复用器、触发器和锁存器等。

准确理解上面这些问题,将为读者后续高效率地学习Verilog HDL提供很大的帮助。

5.2 Verilog HDL要素

Verilog HDL要素主要包括注释、间隔符、标识符、关键字、系统任务和函数、编译器命令、运算符、数字、字符串和属性。

5.2.1 注释

Verilog HDL有两种形式的注释,该语法规定和C语言一致。

1) 单行注释

起始于双斜线"//",表示该行结束以及新的一行开始。单行注释符号"//"在块注释语句内并无特定含义。

2) 多行注释(块注释)

以符号单斜线星号"/*"作为开始标志,以星号单斜线"*/"作为结束标志。块注释不能嵌套。

5.2.2 间隔符

间隔符包括空格字符(\b)、制表符(\t)、换行符(\n)及换页符,这些字符起到与其他词法标识符相分隔的作用。

间隔符除起到分隔的作用外,在必要的地方插入相应的空格或换行符,可以使程序文本易于用户阅读与修改。

在字符串中,将空格和制表符认为是有意义的字符。

5.2.3 标识符

Verilog HDL 中的标识符可以是任意一组字母、数字、$ 符号和_(下画线)符号的组合,是一个对象唯一的名字。对于标识符来说:

(1) 标识符的第一个字符必须是字母或者下画线;

(2) 标识符区分大小写。

在 Verilog HDL 中,标识符分为简单标识符和转义标识符。

1. 简单标识符

简单标识符是由字母、数字、货币符号($)和下画线构成的任意序列。简单标识符的第一个符号不能使用数字或 $ 符号,且简单标识符对大小写敏感。

【例 5.7】 简单标识符定义的例子。

```
shiftreg_a
busa_index
error_condition
merge_ab
_bus3
n$657
```

2. 转义标识符

转义标识符可以在一条标识符中包含任何可打印字符。转义标识符以\(反斜线)符号开头,以空白结尾。

空白可以是一个空格、一个制表符或换行符。

【例 5.8】 转义标识符的例子。

```
\busa+index
\-clock
\***error-condition***
\net1/\net2
\{a,b}
\a*(b+c)
```

5.2.4 关键字

Verilog HDL 内部所使用的词称为关键字或保留字,不能随便使用这些保留字。所有的关键字都使用小写字母。

Verilog HDL(IEEE 1364—2005)关键字列表如下。

always	for	output	supply0
and	force	parameter	supply1
assign	forever	pmos	table
begin	fork	posedge	task
buf	function	primitive	time
bufif0	highz0	pull0	tran
bufif1	highz1	pull1	tranif0
case	if	pullup	tranif1
casex	ifnone	pulldown	tri
casez	initial	rcmos	tri0
cmos	inout	real	tri1
deassign	input	realtime	triand

default	integer	reg	trior
defparam	join	release	trireg
disable	large	repeat	vectored
edge	macromodule	rnmos	wait
else	medium	rpmos	wand
end	module	rtran	weak0
endcase	nand	rtranif0	weak1
endmodule	negedge	rtranif1	while
endfunction	nmos	scalared	wire
endprimitive	nor	small	wor
endspecify	not	specify	xnor
endtable	notif0	specparam	xor
endtask	notif1	strong0	
event	or	strong1	

注：如果关键字前面带有转义符，则不再作为关键字使用。

5.2.5 系统任务和函数

为了便于设计者对仿真过程进行控制，以及对仿真结果进行分析，Verilog HDL 提供了大量的系统功能调用，大致可以分为两类。

(1) 任务型功能调用，称为系统任务。

(2) 函数型功能调用，称为系统函数。

Verilog HDL 中以 $ 字符开始的标识符表示系统任务或系统函数，它们的区别主要包含以下几方面。

(1) 系统任务可以返回 0 个或多个值。

(2) 系统函数只有一个返回值。

(3) 系统函数在 0 时刻执行，即不允许延迟；而系统任务可以包含延迟。

【例 5.9】 系统任务的 Verilog HDL 描述例子。

```
$display("display a message");
$finish;
```

5.2.6 编译器命令

同 C 语言中的编译预处理命令一样，Verilog HDL 也提供了大量编译命令。通过这些编译命令，使得 EDA 工具厂商用它们的工具解释 Verilog HDL 模型变得相当容易。以`(重音符号)开始的某些标识符是编译器命令。在编译 Verilog HDL 程序时，特定的编译器命令均有效，即编译过程可跨越多个文件，直到遇到其他的不同编译命令为止。

【例 5.10】 编译器命令的 Verilog HDL 描述例子。

```
`define wordsize 8
```

5.2.7 运算符

Verilog HDL 提供了丰富的运算符，关于运算符的内容将在后面详细介绍。

5.2.8 数字

本节主要介绍整数型常量和实数型常量。


视频讲解

1. 整数型常量

整数型常量可以按如下方式表示。

1）简单的十进制格式

这种形式的整数定义为带有一个可选的“+”(一元)或“-”(一元)操作符的数字序列。

2）基数表示法

这种形式的整数格式为

```
<size><'base_format><number>
```

其中：

(1) <size>,定义将数字 number 转换为二进制数后,计算得到的位宽。该参数是一个非零的无符号十进制常量。

(2) <'base_format>,撇号是指定位宽格式表示法的固有字符,不能省略。base_format 是用于指定数的基数格式(进制)的一个字母,对大小写不敏感。在撇号后可以添加下面的基数标识：①字母 s/S,表示该数为有符号数；②字母 o/O,表示八进制；③字母 b/B,表示二进制；④字母 d/D,表示十进制；⑤字母 h/H,表示十六进制。

注：撇号和 base_format 之间不能有空格。

(3) <number>是基于基数值的无符号数字序列,由基数格式所对应的数字串组成。数值 x 和 z 以及十六进制中的 a 到 f 不区分大小写。每个数字之间,可以通过‘_’符号连接。

对于没有位宽和基数的十进制数,将其作为有符号数。然而,如果带有基数的十进制数包含了 s/S,则认为是有符号数；如果只带有基数,则认为是无符号数。s/S 指示符,不影响指定的位符号,只是对它的理解问题。

在位宽常数前的+或者-号,是一个一元的加或者减操作符。

注：负数应该用二进制的补码表示。

对于非对齐宽度整数的处理,遵循下面的规则。

(1) 当位宽小于无符号数的实际位数时,截断相应的高位部分。

(2) 当位宽大于无符号数的实际位数,且数值的最高位是“0”或“1”时,相应的高位部分补“0”或“1”。

(3) 当位宽大于无符号数的实际位数,且数值的最高位是“x”或“z”时,相应的高位部分补“x”或“z”。

(4) 如果未指定无符号数的位宽,那么默认的位宽至少为 32 位。

【例 5.11】 未指定位宽常数的例子。

```
659                 //十进制数
'h 837FF            //十六进制数
'o7460              //八进制数
```

【例 5.12】 指定位宽常数的例子。

```
4'b1001             //4 位二进制数
5'D3                //5 位十进制数
3'b01x              //3 位数,其最低有效位未知("x"表示不确定)
12'hx               //12 位数,其值不确定
16'hz               //16 位数,其值为高阻("z"表示高阻状态)
```

【例 5.13】 带符号常数的例子。

```
8'd-6               //非法声明
-8'd6               //定义了 6 的二进制补码,共 8 位,等效于-(8'd6)
4'shf               //定义了 4 位数,将其理解为-1 的二进制补码,等效于-4'h1
```

```
-4'sd15              //等效于-(-4'd1),或者 '0001'
16'sd?               //和 16'sbz 相同
```

注：符号“?”用于替换“z”，对于十六进制，设置为 4 位；对于八进制，设置为 3 位；对于二进制，设置为 1 位。

【例 5.14】 自动左对齐常数的例子。

```
reg[11:0] a, b, c, d;
initial begin
    a = 'h x;        //生成 xxx
    b = 'h 3x;       //生成 03x
    c = 'h z3;       //生成 zz3
    d = 'h 0z3;      //生成 0z3
end
reg[84:0] e, f, g;
    e = 'h5;         //生成{82{1'b0},3'b101}
    f = 'hx;         //生成{85{1'hx}}
    g = 'hz;         //生成{85{1'hz}}
```

【例 5.15】 带下画线常数的例子。

```
27_195_000
16'b0011_0101_0001_1111
32 'h 12ab_f001
```

注：当分配 reg 数据类型时，带宽度限制的负常数和带宽度限制的有符号数都是符号扩展，而不考虑 reg 本身是否是有符号的。

2. 实数型常量

在 IEEE Std 754—1985 中，对实数的表示进行了说明，该标准用于双精度浮点数。实数可以用十进制记数法或者科学记数法表示。

注：十进制小数点两边，至少要有一个数字。

【例 5.16】 有效实数常量表示的例子。

```
1.2
0.1
2394.26331
1.2E12               //指数符号为 e 或者 E
1.30e-2
0.1e-0
23E10
29E-2
236.123_763_e-12     //忽略下画线
```

【例 5.17】 无效实数常量表示的例子。

```
.12
9.
4.E3
.2e-7
```

3. 实数到整数的转换

Verilog HDL 规定，通过四舍五入的方法将实数转换为最近的整数，而不是截断它。

【例 5.18】 对实数转换为整数的表示。

```
42.446 和 42.45 转换为整数 42
92.5 和 92.699 转换为整数 93
-15.62 转换为整数 -16
-26.22 转换为整数 -26
```


视频讲解

5.2.9 字符串

字符串是双引号内的字符序列，用一串 8 位二进制 ASCII 码的形式表示，每个 8 位二进制 ASCII 码代表一个字符。例如，字符串“ab”等价于 16'h5758。如果字符串用作 Verilog HDL 表达式或赋值语句的操作数，则将字符串看作无符号整数序列。

1. 字符串变量声明

字符串变量是寄存器 reg 类型变量，它的位宽等于字符串的字符个数乘以 8。

【例 5.19】 字符串变量的声明如代码清单 5-8 所示。

存储 12 个字符的字符串“Hello China!”需要 8×12(即 96)位宽的寄存器。

代码清单 5-8 字符串变量的 Verilog HDL 描述

```
reg[8*12:1] str;
  initial
    begin
      str = "Hello China!";
   end
```

2. 字符串操作

可以使用 Verilog HDL 的操作符对字符串进行处理，由操作符处理的数据是 8 位 ASCII 码的序列。对于宽度非对齐的情况，采用下面的方式进行处理。

(1) 在操作过程中，如果声明的字符串变量位数大于字符串实际长度，则在赋值操作后，字符串变量左端(即高位)的所有位补“0”。这一点与非字符串值的赋值操作是一致的。

(2) 如果声明的字符串变量位数小于字符串实际长度，那么截断字符串的左端，这样就丢失了高位字符。

【例 5.20】 字符串操作的例子如代码清单 5-9 所示。

代码清单 5-9 字符串操作的 Verilog HDL 描述

```
module string_test;
     reg[8*14:1] stringvar;
initial
begin
    stringvar = "Hello China";
    $display("%s is stored as %h",stringvar,stringvar);
    stringvar = {stringvar."!!!"};
    $display("%s is stored as %h",stringvar,stringvar);
end
endmodule
```

输出结果为

```
Hello China is stored as 00000048656c6c6f20776f726c64
Hello China!!! is stored as 48656c6c6f20776f726c64212121
```

3. 特殊字符

在某些字符之前可以加上一个引导性的字符(转义字符)，这些字符只能用于字符串中。表 5.1 列出了这些特殊字符的表示和意义。

表 5.1 特殊字符的表示和意义

特殊字符的表示	意　义
\n	换行符
\t	Tab 键
\\	符号\
\"	符号"
\ddd	3 位八进制数表示的 ASCII 码值(0≤d≤7)

5.2.10 属性

随着工具的扩展，除了仿真器使用 Verilog HDL 作为其输入源外，还包含另外一个机制，即在 Verilog HDL 源文件中指定对象、描述和描述组的属性。这些属性可以用于各种工具，包括仿真器和控制工具的操作行为。

指定属性的格式为

```
( * attribute_name = constant_expression * )
```

或者

```
( * attribute_name * )
```

【例 5.21】 将属性添加到 case 描述的 Verilog HDL 例子。

```
( * full_case, parallel_case * )
case(foo)
          < rest_of_case_statement >
( * full_case = 1 * )
( * parallel_case = 1 * )                    //多个属性
case(foo)
  < rest_of_case_statement >
```

或者

```
( * full_case,                               //没有分配值
    parallel_case = 1 * )
case(foo)
      < rest_of_case_statement >
```

【例 5.22】 添加 full_case 属性，但是没有 parallel_case 属性的 Verilog HDL 例子。

```
( * full_case * )                            //没有指定 parallel_case
case(foo)
    < rest_of_case_statement >
```

或者

```
( * full_case = 1, parallel_case = 0 * )
case(foo)
    < rest_of_case_statement >
```

【例 5.23】 将属性添加到模块定义的 Verilog HDL 例子。

```
( * optimize_power * )
module mod1 (< port_list >);
```

或者

```
( * optimize_power = 1 * )
module mod1 (< port_list >);
```

【例 5.24】 将属性添加到模块例化的 Verilog HDL 例子。

```
( * optimize_power = 0 * )
mod1 synth1 (< port_list >);
```

【例 5.25】 将属性添加到 reg 声明的 Verilog HDL 例子。

```
( * fsm_state * )reg [7:0] state1;
( * fsm_state = 1 * )reg [3:0] state2, state3;
reg[3:0] reg1;                               //这个 reg 没有设置 fsm_state
( * fsm_state = 0 * )reg [3:0] reg2;         //这个也没有
```

【例 5.26】 将属性添加到操作符的 Verilog HDL 例子。

```
a = b + ( * mode = "cla" * ) c;              //将属性模式的值设置为字符串 cla
```

【例 5.27】 将属性添加到一个 Verilog 函数调用 Verilog HDL 例子。

```
a = add ( * mode = "cla" * ) (b, c);
```

【例 5.28】 将属性添加到一个有条件操作符的 Verilog HDL 例子。

```
a = b ? ( * no_glitch * ) c : d;
```

注：高云综合工具所支持的属性的格式为

```
/ * synthesis attribute_name = constant_expression * /
```

这与 Verilog HDL 所支持的属性的声明格式有所不同。

5.2.11 设计实例二：有符号加法器的设计与验证

本节将设计一个有符号的加法器，并对设计的加法器进行验证。通过该设计实例，对 5.2 节的内容进行系统地理解和掌握。

读者可以在高云云源软件和 ModelSim 软件中使用代码清单 5-10 和代码清单 5-11 给出的设计代码和仿真代码，执行详细描述、设计综合、行为仿真、设计约束、设计实现、时序仿真和设计下载等。

代码清单 5-10 adder.v 文件

```
  / * 属性 syn_dspstyle 的值为"dsp", 告诉高云综合工具使用 FPGA 内的 DSP 模块实现加法器 * /
  / * 而不使用分布式的逻辑资源实现加法器 * /
module adder(                          // module 关键字定义模块 adder
    input signed [7:0] a,              // input 和 signed 关键字声明端口 a 是输入且有符号
    input signed [7:0] b,              // input 和 signed 关键字声明端口 b 是输入且有符号
    output signed [7:0] y              // input 和 signed 关键字声明端口 c 是输入且有符号
    ); / * synthesis syn_dspstyle = "dsp" * /;  // 高云的属性设置,属性 syn_dspstyle 取值为 dsp
assign y = a + b;                      // 端口 a 和端口 b 的输入,执行有符号加法,结果给 y
endmodule                              // endmodule 关键字表示模块 adder 的结束
```

在高云云源软件中，显示的 RTL 级网表如图 5.2 所示。

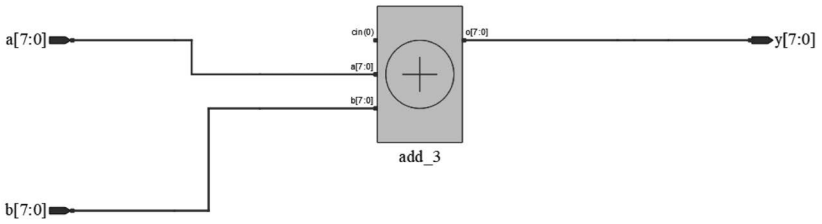

图 5.2 有符号加法器的 RTL 结构

注：在配套资源\eda_verilog\example_5_3 目录下，用高云云源软件打开 example_5_3.gprj。

代码清单 5-11 test.v 文件

```
module test;                           // 关键字 module 定义模块
reg signed [7:0] a,b;                  // 关键字 reg 和 signed 定义 reg 类型变量 a 和 b 是有符号的
wire signed [7:0] y;                   // 关键字 wire 和 signed 定义网络 y 是有符号的

adder Inst_adder(                      // Verilog 元件例化语句,将模块 adder 例化为 Inst_adder
    .a(a),                             // 被调用模块 adder 的端口 a 连接到调用模块的变量 a
    .b(b),                             // 被调用模块 adder 的端口 b 连接到调用模块的变量 b
    .y(y)                              // 被调用模块 adder 的端口 y 连接到调用模块的变量 y
    );

initial                                // 关键字 initial 定义初始化部分
begin                                  // 关键字 begin 表示初始化的开始,类似 C 语言的"{"
```

```
    a = 8'sd210;                        // 210 对应有符号数 - 46,即 a = - 46
    b = 8'd67;                          // b = 67
    #100;                               // 符号"#"定义延迟,#100 表示持续 100ns
    $display("%d + %d = %d",a,b,y);     // 符号"$"表示 Verilog 的系统任务,display 打印信息
    a = - 8'sd200;                      // 200 对应有符号数 - 56, - 8'sd200 = 56,a = 56
    b = - 8'd27;                        // - 8'd27 = - 27,b = - 27
    #100;                               // 符号"#"定义延迟,#100 表示持续 100ns
    $display("%d + %d = %d",a,b,y);     // 符号"$"表示 Verilog 的系统任务,display 打印信息
    a = - 8'd56;                        // - 8'd56 = - 56, a = - 56
    b = - 8'd45;                        // - 8'd45 = - 45, b = - 45
    #100;                               // 符号"#"定义延迟,#100 表示持续 100ns
    $display("%d + %d = %d",a,b,y);     // 符号"$"表示 Verilog 的系统任务,display 打印信息
    a = 8'b10100101;                    // 二进制数"10100101",对应有符号数 - 91
    b = 8'b01010111;                    // 二进制数"01010111",对应有符号数 87
    #100;                               // 符号"#"定义延迟,#100 表示持续 100ns
    $display("%d + %d = %d",a,b,y);     // 符号"$"表示 Verilog 的系统任务,display 打印信息
    a = 8'h9d;                          // 十六进制数 9d,对应有符号数 - 99,a = - 99
    b = 8'h1f;                          // 十六进制数 1f,对应有符号数 31,b = 31
    #100;                               // 符号"#"定义延迟,#100 表示持续 100ns
    $display("%d + %d = %d",a,b,y);     // 符号"$"表示 Verilog 的系统任务,display 打印信息
end
endmodule
```

注：在配套资源\eda_verilog\example_5_4 目录下，用 ModelSim SE-64 10.4c 打开 postsynth_sim.mpf。

当使用 ModelSim 执行综合后仿真时，在 Transcript 窗口中打印的信息如图 5.3 所示。

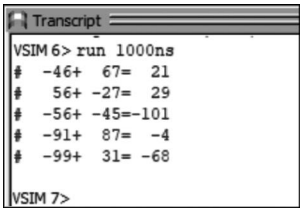

图 5.3　在 Transcript 窗口打印的信息

思考与练习 5-4　将代码清单 5-10 中的属性设置代码去掉，重新综合，查看综合后的原理图，并对该结果进行分析，说明属性的作用。

思考与练习 5-5　修改代码清单 5-10 中代码，将端口的有符号标识关键字 signed 去掉，即把有符号加法改成无符号加法，并修改代码清单 5-11 中的测试向量，将测试向量均改为无符号数，对设计重新执行综合后仿真，以验证设计的正确性。

思考与练习 5-6　修改代码清单 5-10 中的代码，将加法改成减法操作，并修改代码清单 5-11 中的测试向量，对设计重新执行综合后仿真，以验证设计的正确性。


视频讲解

5.3　Verilog HDL 数据类型

Verilog HDL 数据类型包括值的集合、网络和变量、向量、强度、隐含声明、网络类型、reg 类型、整数/实数/时间、数组、参数和 Verilog 命名空间。

5.3.1　值的集合

Verilog HDL 有 4 种基本的值："0"，逻辑"0"或"假"状态；"1"，逻辑"1"或"真"状态；"x"(X)，未知状态，对大小写不敏感；"z"(Z)，高阻状态，对大小写不敏感。

注：

(1) 对这 4 种值的解释都内置于 Verilog HDL 中，如一个为"z"的值总是意味着高阻抗，一个为"0"的值通常是指逻辑"0"。

(2) 通常地，将门的输入或一个表达式中的"z"值解释成"x"，在 MOS 原语中例外。

5.3.2 网络和变量

在 Verilog HDL 中，根据赋值和保持值方式的不同，可将数据类型分为两大类：网络类型和变量类型，它们代表了不同的硬件结构。

1. 网络声明

网络表示结构化实体之间的物理连接，例如门。网络类型不保存值（除 trireg 以外），其输出始终随着输入的变化而变化。如果无驱动器连接到网络，网络的值为“z”（高阻），除非网络是 trireg。

对于没有声明的网络，默认类型为一位（标量）wire 类型；Verilog HDL 禁止再次声明已经声明过的网络、变量或参数。下面给出声明网络类型的语法格式：

```
<net_type> [range] [delay] <net_name>[,net_name];
```

其中：

（1）net_type 表示网络类型数据。

（2）range 用于指定数据为标量或向量。若没有声明范围，则表示数据类型为 1 位的标量。否则，由该项指定数据的向量形式。

（3）delay 指定仿真延迟时间。

（4）net_name 为网络名字。可以一次定义多个相同属性的网络，多个网络之间用逗号分隔。

2. 变量声明

变量是对数据存储元件的抽象。从当前赋值到下一次赋值之前，变量应当保持当前的值不变。过程中的赋值语句可看作触发器，它将引起数据存储元件中值的改变。

（1）对于 reg、time 和 integer 这些变量类型数据，它们的初始值应当是未知（x）。

（2）对于 real 和 realtime 变量类型数据，默认的初始值是 0.0。

（3）如果使用变量声明赋值语句，那么变量将声明赋值语句所赋的值作为初值，这与 initial 结构中对变量的阻塞赋值等效。

5.3.3 向量

在一个网络或 reg 类型声明中，如果没有指定其范围，默认将其当作 1 比特位宽，也就是通常所说的标量。通过指定范围，声明多位的网络类型或 reg 类型数据，则称为向量（也叫作矢量）。

1. 向量声明

向量范围由常量表达式来说明。msb_constant_expression（最高有效位常量表达式）代表范围的左侧值，lsb_constant_expression（最低有效位常量表达式）代表范围的右侧值，右侧表达式的值可以大于、等于或小于左侧表达式的值。

网络类型和 reg 类型向量遵循以 2 为模（2^n）的乘幂算术运算法则，此处的 n 值是向量的位宽。如果没有将网络类型和 reg 类型向量声明为有符号量或者将其连接到一个已声明为有符号的数据端口，则该向量当作无符号的向量。

【例 5.29】 向量声明的 Verilog HDL 例子。

```
wand w;                          //wand 类型的标量
tri[15:0] busa;                  //一个三态 16 位总线
trireg(small) storeit;           //低强度的一个充电保存点
reg a;                           //reg 类型的标量
reg[3:0] v;                      //4 位的 reg 类型的向量，由 v[3]、v[2]、v[1]和 v[0]构成
```

```
reg signed[3:0] signed_reg;          //一个 4 位的向量,其范围为 - 8 到 7
reg[ - 1:4] b;                       //一个 6 位 reg 类型的向量
wire w1, w2;                         //声明两个线网络
reg[4:0] x, y, z;                    //声明三个 5 位的 reg 类型变量
```

2. 向量网络类型数据的可访问性

vectored 和 scalared 是向量网络类型或向量寄存器类型数据声明中的可选择关键字。如果使用这些关键字,那么向量的某些操作就会受约束。

(1) 如果使用关键字 vectored,则禁止向量的位选择或部分位选择以及指定强度,而 PLI 就会认为未展开数据对象。

(2) 如果使用关键字 scalared,则允许向量的位选择或部分位选择,PLI 认为展开数据对象。

【例 5.30】 关键字 vectored 和 scalared 的 Verilog HDL 例子。

```
tri scalared[63:0] bus64;            //一个将被展开的总线
tri vectored[31:0] data;             //一个未被展开的总线
```

5.3.4 强度

在一个网络类型数据类型声明中,可以指定两类强度。

1) 电荷量强度

只有在 trireg 网络类型的声明中,才可以使用该强度。一个 trireg 网络类型数据用于模拟一个电荷存储节点,该节点的电荷量将随时间而逐渐衰减。在仿真时,对于一个 trireg 网络类型数据,其电荷衰减时间应当指定为延迟时间。电荷量强度可由下面的关键字来指定电容量的相对大小:①small;②medium;③large。默认的电荷强度为 medium。

2) 驱动强度

在一个网络类型数据的声明语句中,如果对数据对象进行了连续赋值,就可以为声明的数据对象指定驱动强度。门级元件的声明只能指定驱动强度。根据驱动源的强度,其驱动强度可以是 supply、strong、pull 或 weak。

注:*在高云云源软件综合工具中,将忽略网络中的强度定义。*

【例 5.31】 强度 Verilog HDL 描述的例子。

```
trireg a;                            //trireg 网络,其电荷量强度为 medium
trireg(large) #(0,0,50) cap1;        //trireg 网络,其电荷量强度为 large,电荷衰减时间为 50 个
                                     //时间单位
trireg(small)signed [3:0] cap2;      //有符号的 4 位 trireg 向量,其电荷强度为 small
```

5.3.5 隐含声明

如果没有显式声明网络或者变量,则在下面的情况中,默认将其指定为网络类型:

(1) 在一个端口表达式的声明中,如果没有对端口的数据类型进行显式说明,则默认的端口数据类型就为网络类型;并且,默认的网络类型向量的位宽与向量型端口声明的位宽相同。

(2) 在原语或模块例化的端口列表中,如果事先没有对端口的数据类型进行显式说明,那么默认的端口数据类型为网络类型标量。

(3) 如果一个标识符出现在连续赋值语句的左侧,而事先未声明该标识符,那么该标识符的数据类型隐式声明为网络类型标量。

5.3.6 网络类型

在 Verilog HDL 中提供了下面的网络类型,如表 5.2 所示。

表 5.2 网络类型

序号	网络类型	序号	网络类型	序号	网络类型
1	wire	5	triand	9	trireg
2	wand	6	trior	10	supply0
3	wor	7	tri0	11	supply1
4	tri	8	tri1	12	uwire

1. wire 和 tri 网络类型

wire 和 tri 网络连接元件。网络类型 wire 和 tri 的语法和功能应该相同，提供了两个名字，以便网络的名字可以只是网络在该模型中的用途。wire 网络可以用于由单个门或连续分配驱动的网络，tri 网络可以用于多个驱动器驱动一个网络的情况。

来自一个 wire 或 tri 网络上的相同强度的多个源的逻辑冲突将导致“x”(不确定)值。表 5.3 是多个驱动器驱动 wire 和 tri 网络的真值表。

表 5.3 多个驱动器驱动 wire 和 tri 网络的真值表

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

2. 连线网络

连线网络类型有 wor、wand、trior 和 triand，用于建模连线逻辑配置。连线网络使用不同的真值表来解决多个驱动器驱动相同网络时产生的冲突。wor 和 trior 网络将创建连线或配置，以便当任何驱动器为“1”时，网络最终的值为“1”。wand 和 triand 网络将创建连线或配置，如果任何驱动器为“0”，则网络值为“0”。

网络类型 wor 和 trior 的语法和功能应相同。网络类型 wand 和 triand 的语法和功能应相同。表 5.4 和表 5.5 给出了连线网络的真值表，假设两个驱动器的强度相等。

表 5.4 wand 和 triand 网络的真值表

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

表 5.5 wor 和 trior 网络的真值表

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

【例 5.32】 连线网络 Verilog HDL 描述的例子如代码清单 5-12 所示。

代码清单 5-12 连线网络的 Verilog HDL 描述

```
module wired_net(                    // module 关键字定义模块 wired_net
input a,                             // input 关键字定义 wire 类型端口 a
input b,                             // input 关键字定义 wire 类型端口 b
input c,                             // input 关键字定义 wire 类型端口 c
output wand d,                       // output 关键字定义 wand 类型端口 d
```

```
output wor e                          // output 关键字定义 wor 类型端口 e
);
assign d = a;                         // assign 语句将 a 连接到 d
assign d = b;                         // assign 语句将 b 连接到 d
assign d = c;                         // assign 语句将 c 连接到 d

assign e = a;                         // assign 语句将 a 连接到 e
assign e = b;                         // assign 语句将 b 连接到 e
assign e = c;                         // assign 语句将 c 连接到 e

endmodule                             // endmodule 标识模块 wired_net 的结束
```

对该设计执行综合后的电路结构如图 5.4 所示。

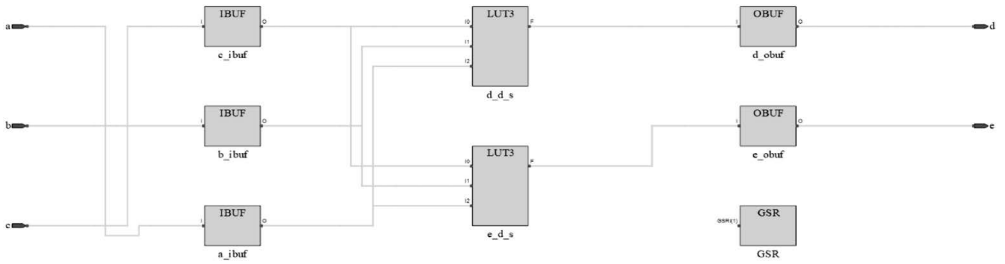

图 5.4　执行综合后的电路结构

注：在配套资源\eda_verilog\example_5_5 目录下，用高云云源软件打开 example_5_5.gprj。

使用 Verilog HDL 编写仿真文件，如代码清单 5-13 所示。

代码清单 5-13　测试向量的 Verilog HDL 描述

```
`timescale 1ns / 1ps                  // 预编译指令,timescale 定义时间标度 1ns/1ps
module test;                          // module 关键字定义模块 test
reg a,b,c;                            // reg 关键字定义 reg 类型变量 a、b 和 c
wand d;                               // wand 关键字定义"线与"网络 d
wor e;                                // wor 关键字定义"线或"网络 e
wired_net Inst_wire_net(              // 元件调用/例化语句,将 wired_net 例化为 Inst_wire_net
  .a(a),                              // 模块 wired_net 端口 a 连接到模块 test 的 reg 类型变量 a
  .b(b),                              // 模块 wired_net 端口 b 连接到模块 test 的 reg 类型变量 b
  .c(c),                              // 模块 wired_net 端口 c 连接到模块 test 的 reg 类型变量 c
  .d(d),                              // 模块 wired_net 端口 d 连接到模块 test 的 wand 类型网络 d
  .e(e)                               // 模块 wired_net 端口 e 连接到模块 test 的 wor 类型网络 e
);
initial                               // initial 关键字定义初始化部分
begin                                 // begin 关键字标识初始化部分的开始,类似 C 语言的"{"
a = 1'b0;                             // 变量 a 分配/赋值"0"
b = 1'b0;                             // 变量 b 分配/赋值"0"
c = 1'b0;                             // 变量 c 分配/赋值"0"
#100;                                 // 符号"#"标识延迟,#100 表示持续 100ns
$display("%b wand %b wand %b = %b", a,b,c,d);  // 调用系统任务 display,打印信息
$display("%b wor %b wor %b = %b", a,b,c,e);    // 调用系统任务 display,打印信息

a = 1'bx;                             // 变量 a 分配/赋值"x"
b = 1'b1;                             // 变量 b 分配/赋值"1"
c = 1'b1;                             // 变量 c 分配/赋值"1"
#100;                                 // 符号"#"标识延迟,#100 表示持续 100ns
$display("%b wand %b wand %b = %b", a,b,c,d);  // 调用系统任务 display,打印信息
$display("%b wor %b wor %b = %b", a,b,c,e);    // 调用系统任务 display,打印信息

a = 1'bz;                             // 变量 a 分配/赋值"z"
b = 1'b1;                             // 变量 b 分配/赋值"1"
c = 1'b1;                             // 变量 c 分配/赋值"1"
```

```
#100;                                    // 符号"#"标识延迟,#100 表示持续 100ns
$display("%b wand %b wand %b = %b", a,b,c,d);  // 调用系统任务 display,打印信息
$display("%b wor %b wor %b = %b", a,b,c,e);    // 调用系统任务 display,打印信息
end                                      // end 标识 initial 部分的结束,类似 C 语言的"}"
endmodule                                // endmodule 标识模块 test 的结束
```

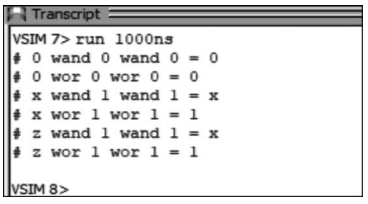

图 5.5 行为级仿真后在 Transcript 窗口显示的信息

对该设计执行综合后仿真后,在 Transcript 窗口中显示的信息如图 5.5 所示。

注:在配套资源\eda_verilog\example_5_6 目录下,用 ModelSim SE-64 10.4c 打开 example_5_6.mpf。

3. Supply 网络

supply0 和 supply1 网络可用于对电路中的电源进行建模,这些网络有供电强度。

4. uwire 网络

uwire 网络是一种未解析或单驱动的网络,用于建模时仅允许单个驱动的网络。uwire 类型可用于强化这种限制。将 uwire 网络的任何一位连接到多个驱动源是错误的。将 uwire 网络连接到双向传输开关的双向端子是错误的。后面介绍的端口连接规则将确保在网络层次结构中强制执行该限制,否则将提示警告信息。

注:高云云源软件综合工具不支持 tri0、tri1 和 trireg 网络类型,因此本节不对这些网络类型进行任何说明。

5.3.7 reg 类型

通过过程分配/赋值语句给 reg 类型变量赋值。由于在分配的过程中间,reg 类型变量保持值不变,所以它用于对硬件寄存器进行建模,可以对边沿敏感(如触发器)和电平敏感(如置位/复位和锁存器)的保存元素进行建模。

注:一个 reg 类型变量不一定代表一个硬件存储元素,这是因为它也能用于表示一个组合逻辑。

reg 类型变量与网络类型的区别主要在于:

(1) reg 类型变量保持最后一次的赋值。只能在 initial 或 always 内部对 reg 类型变量进行赋值操作。

(2) 网络类型数据需要有连续的驱动源。

reg 类型变量声明的格式如下:

```
<reg_type> [range] <reg_name>[, reg_name];
```

其中:

(1) range 为向量范围,[MSB: LSB]格式只对 reg 类型有效;

(2) reg_name 为 reg 类型变量的名字,一次可定义多个相同属性的 reg 类型变量,使用逗号分隔。


视频讲解

5.3.8 整数、实数、时间和实时时间

在 Verilog HDL 中,整数用关键字 integer 声明,实数用关键字 real 声明,时间用关键字 time 声明,实时时间用关键字 realtime 声明。

除了 reg 类型对硬件建模外,HDL 模型中的变量还有其他用途。尽管 reg 类型变量可用于一般的用途,例如计算一个特殊网络变化值的次数,但是提供整数和时间变量数据类型是为

了方便,使得能够对自己的描述进行标记。

整数 integer 是一个通用变量,用于处理不被看作硬件寄存器的数量。

时间变量用于在仿真中保存和操作仿真时间长度,这里需要时序检查以及用于诊断和调试的目的。这种数据类型通常与 $time 系统函数一起使用。

integer 和 time 变量的赋值方式与 reg 相同。过程赋值用于触发它们值的变化。

时间变量的行为至少与 64 位的 reg 相同,最低有效位为第 0 位。它们应为无符号量,并对其进行无符号运算。相反,整数变量应该看作有符号的 reg,最低有效位为第 0 位。对整数变量执行算术运算应产生二进制补码结果。

允许 reg、integer 和 time 变量的位选择和部分选择。实现可能会限制整数变量的最大长度,但至少应该为 32 位。

Verilog HDL 支持实数常量和实数变量数据类型,以及整数和时间变量数据类型。除了下面的限制外,声明为 real 的变量可以在使用整数和时间变量的相同位置使用。

(1) 并非所有的 Verilog HDL 运算符都可以与实数值一起使用,后面将给出实数和实数变量的有效和无效运算符列表。

(2) 实数变量在声明中不能使用范围。

(3) 实数变量应默认初始值为零。

realtime 的声明应与 real 声明同义,并且可以互换使用。

1. 运算符和实数

对实数和实数变量使用逻辑或关系运算符的结果是一位标量值。并非所有 Verilog HDL 运算符都可以与涉及实数和实数变量的表达式一起使用。在下面的情况下,禁止使用实数常量和实数变量。

(1) 应用于实数变量的边沿描述符(posedge、negedge)。

(2) 声明为 real 的变量的位选择或部分选择引用。

(3) 向量的位选择或部分选择引用的实数索引表达式。

2. 转换

当表达式分配/赋值给实数时,发生隐式转换。在转换时,网络或变量中的"x"或"z"应看作 0。

【例 5-33】 实数与整数转换和运算的 Verilog HDL 描述的例子如代码清单 5-14 所示。

代码清单 5-14 实数与整数转换和运算的 Verilog HDL 描述

```
module top(                    // 关键字 module 定义模块 top
  input [7:0] a,               // 关键字 input 定义输入端口 a,宽度为 8 位,索引为 a[7]~a[0]
  output [7:0] x               // 关键字 output 定义输出端口 b,宽度为 8 位,索引为 x[7]~x[0]
   );
assign x = m[7:0] + a;         // 关键字 assign,整数 m[7:0]与输入端口 a 的结果赋值到输出端口 x
integer m;                     // 关键字 integer 定义整数 m
real n,i;                      // 关键字 real 定义实数 n 和 i
initial                        // 关键字 initial 表示初始化部分
begin                          // begin 标识初始化部分的开始,类似 C 语言的"{"
  n = 45.0;                    // 实数 n 赋值为 45.0
  i = 46.1;                    // 实数 i 赋值为 46.1
  m = $rtoi(n + i);            // 符号"$"标识 Verilog HDL 系统任务,rtoi 将 n + i 的结果转换为整数
end                            // end 标识初始化部分的结束,类似 C 语言的"}"
endmodule                      // endmodule 标识模块的结束
```

注:在配套资源\eda_verilog\example_5_7 目录下,用高云云源软件打开 example_5_7.xpr。

5.3.9 数组


视频讲解

用于网络或变量的数组声明,声明了标量或向量的元素类型,如表 5.6 所示。

表 5.6 声明与元素类型之间的关系

声　明	元素类型
reg x[11:0]	标量 reg
wire [0:7] y[5:0]	8 位宽度的网络线,索引为从 0 到 7
reg [31:0] x[127:0]	32 位宽 reg

注：数组宽度不影响元素宽度。

数组可用于将声明的元素类型分组为多维对象。数组应通过在声明的标识符后面指定元素的地址范围来声明。每个维度都应该以地址范围表示。指定数组索引的表达式应该是常量整数表达式。常量整数表达式的值可以是正整数、负整数或零。

一个声明语句可用于声明所声明数据类型的数组和元素。这种能力使得在同一声明语句中声明与元素向量宽度匹配的数组和元素变得更加方便。

在一次赋值中,可以给一个元素分配一个值,但不能使用完整或部分数组维度为表达式分配值,也不能使用完整或部分数组维度为表达式提供值。要为数组元素分配值,应只是每个维度的索引。索引可以是表达式。该选项提供了一种机制,根据电路中其他变量和网络的值来引用不同数组元素。例如,程序计数器 reg 可用于索引到 RAM。

实现限制了数组的最大容量,但至少允许它们有 16 777 216(2^{24})个元素。

1. 网络数组

网络数组的元素可以用与一个标量或向量网络相同的方式使用,它们对于连接到循环生成结构内的模块实例的端口非常有用。

2. reg 和变量数组

所有变量类型(reg、integer、time、real、realtime)的数组都是可能的。

3. 存储器

具有 reg 类型元素的一维数组也称为一个存储器,这些存储器能用于为 ROM、RAM 和寄存器文件建模。数组中的每个 reg 称为一个元素或字,它们通过单个数组索引进行寻址。

在一次分配/赋值中,可以给一个 n 维 reg 分配一个值,但是不能给完整的存储器分配值。为了给存储器字分配一个值,必须要指定一个索引。索引可以是表达式。这个选项提供一个机制,用于引用不同的存储器字,这取决于电路中其他变量或网络的值。例如,程序计数器 reg 就能用于索引 RAM。

【例 5.34】 声明数组的 Verilog HDL 描述的例子如代码清单 5-15 所示。

代码清单 5-15 声明数组的 Verilog HDL 描述

```
reg[7:0] mema[0:255];          //声明一个数组 mema 为 256×8 比特 reg 类型变量,
                               //其索引为 0～255,宽度为 8 位
reg arrayb[7:0][0:255];        //声明一个二维数组,其数据为 1 位 reg 类型变量
wire w_array[7:0][5:0];        //声明线类型网络数组
integer inta[1:64];            //64 个整数值的数组
time chng_hist[1:1000]         //有 1000 个时间值的数组
integer t_index;
```

【例 5.35】 分配数组元素的 Verilog HDL 描述的例子如代码清单 5-16 所示。

代码清单 5-16 分配数组元素的 Verilog HDL 描述

```
mema = 0;                      //非法的描述,尝试给整个数组写 0
arrayb[1] = 0;                 //非法的描述,尝试写元素[1][0]…[1][255]
arrayb[1][12:31] = 0;          //非法的描述,尝试写元素[1][12]…[1][31]
mema[1] = 0;                   //给 mema 的第二个元素分配 0
arrayb[1][0] = 0;              //给索引[1][0]指向的元素分配 0
```

```
inta[4] = 33559;                    //给数组的某个元素分配整数值 33559
chng_hist[t_index] = $time;         //给当前索引指向的元素分配仿真时间
```

【例 5.36】 不同存储器 Verilog HDL 描述的例子。

```
reg [1:n] rega;                     //一个 n 位的深度为 1 的 reg 类型变量(存储器)
reg mema [1:n];                     //一个 1 位的深度为 n 的 reg 类型变量(存储器)
```

5.3.10 参数

Verilog HDL 中的参数既不属于变量类型也不属于网络类型范畴。参数不是变量,而是常量。Verilog HDL 提供了两种类型的参数:

(1) 模块参数;

(2) 指定参数。

所有这些参数都可以指定范围。默认地,parameter 和 specparams 保持必要的宽度,用于保存常数的值。当指定范围时,按照指定的范围确定。

1. 模块参数

模块参数定义的格式为

```
parameter par_name1 = expression1, … , par_namen = expressionn;
```

其中:

(1) par_name1,…,par_namen 为参数的名字;

(2) expression1,…,expressionn 为表达式;

可一次定义多个参数,用逗号隔开。参数的定义是局部的,只在当前模块中有效。

使用 parameter 定义的参数表示常数,因此,不能在运行时修改它们。但是,可以在编译时修改模块的参数,使其值与声明分配中指定的值不同。这允许自定义模块实例,可以使用 defparam 语句或在模块例化语句中修改参数。参数典型的用途是指定变量的延迟和宽度。

一个模块参数可以指定类型和范围,规则如下。

(1) 没有指定类型和范围的参数,将根据分配给参数最终的值来确定类型和范围。

(2) 一个指定范围,但没有指定类型的参数,将是参数声明的范围,并且是无符号的。符号和范围将不受到后面所分配值的影响。

(3) 一个指定类型,但没有指定范围的参数,将是参数指定的类型。一个有符号的参数,默认为分配给参数最后值的范围。

(4) 一个指定有符号类型和范围的参数,将是有符号的,并且是参数指定的范围,其符号和范围将不受到后面分配值的影响。

(5) 一个没有指定范围,但是有指定符号类型或者没有指定类型的参数,有一个隐含的范围,其 lsb 为 0,msb 等于或者小于分配给参数最后的值。

(6) 一个没有指定范围,但是有指定符号类型或者没有指定类型的参数,并且为其分配的最终值未指定宽度,其隐含范围的 lsb 为 0,msb 应等于依赖于实现的值,至少为 31。

允许不是 real 类型的所有其他类型参数的位选择和部分选择。

【例 5.37】 参数的 Verilog HDL 描述的例子如代码清单 5-17 所示。

代码清单 5-17 参数的 Verilog HDL 描述

```
parameter msb = 7;                  //定义 msb 为常数值 7
parameter e = 25, f = 9;            //定义两个常数
parameter r = 5.7;                  //定义 r 为实数参数
parameter byte_size = 8,byte_mask = byte_size - 1;
parameter average_delay = (r + f) / 2;
parameter signed [3:0] mux_selector = 0;
```

```
parameter real r1 = 3.5e17;
parameter p1 = 13'h7e;
parameter [31:0] dec_const = 1'b1;  //值转换到 32 位
parameter newconst = 3'h4;          //暗示其范围为[2:0]
parameter newconst = 4;             //暗示其范围为[31:0]
```

2. 本地参数

除了不能直接被 defparam 语句修改，或者被模块例化参数分配以外，本地参数和参数是一致的。本地参数可以分配包含参数的常数表达式，这些参数可以通过 defparam 语句或者模块例化参数值分配进行修改。

3. 指定参数

关键字 specparam 声明了一个特殊类型的参数，这个参数专用于提供时序和延迟值，但是可以出现在任何没有分配参数的表达式内，它不是一个声明范围描述的一部分。在 specify 块内或主模块内，允许指定参数。高云云源软件综合工具忽略指定参数。

当声明了一个指定的参数在一个指定块的外部时，在引用之前必须声明。分配给指定参数的值可以是任何常数表达式。不像模块参数那样，不能在语言内修改一个指定的参数。但是，可以通过 SDF 注解修改。

指定参数和模块参数是不能交换的。此外，模块参数不能分配一个包含指定参数的常数表达式。表 5.7 给出了 specparam 和 parameter 的不同之处。

表 5.7 specparam 和 parameter 的不同之处

specparam(指定参数)	parameter(模块参数)
使用关键字 specparam	使用关键字 parameter
在一个模块内或者指定块内声明	在指定的块外声明
只能在一个模块内或者指定块内使用	不能在指定块内使用
可以被分配指定参数和参数	不能分配指定参数
使用 SDF 注解覆盖值	使用 defparam 或者例化声明参数值传递来覆盖值

【例 5.38】 声明和使用指定参数的 Verilog HDL 描述的例子如代码清单 5-18 所示。

代码清单 5-18 声明和使用指定参数的 Verilog HDL 描述

```
module test;                        // module 关键字定义模块 test
specify                             // specify 关键字用于声明指定块
specparam delay = 10.0;             // specparam 关键字用于指定参数
endspecify                          // endspecify 关键字标识指定块的结束
reg a,b;                            // reg 关键字定义 reg 类型变量 a 和 b
wire c;                             // wire 关键字定义 wire 类型网络 c
and #delay Inst_and(c,a,b);         // 调用 Verilog HDL 内建的逻辑与门 and

initial                             // initial 关键字定义初始化部分
begin                               // begin 关键字表示初始化部分的开始,类似 C 语言的"{"
  a = 1'b0;                         // 变量 a 分配/赋值"0"
  b = 1'b0;                         // 变量 b 分配/赋值"0"
  #delay;                           // 持续指定参数 delay 指定的时间
  a = 1'b1;                         // 变量 a 分配/赋值"1"
  b = 1'b1;                         // 变量 b 分配/赋值"1"
  #delay;                           // 持续指定参数 delay 指定的时间
  a = 1'b0;                         // 变量 a 分配/赋值"0"
  b = 1'b1;                         // 变量 b 分配/赋值"1"
  #delay;                           // 持续指定参数 delay 指定的时间
end                                 // end 关键字表示初始化部分的结束
endmodule                           // endmodule 关键字表示模块 test 的结束
```

思考与练习 5-7 在 ModelSim SE-64 10.4c 中建立新的设计工程，创建名为 test.v 的仿真文件，并执行 RTL/综合后仿真，在波形窗口中观察波形，说明指定参数的用法。

5.3.11 Verilog HDL 名字空间

在 Verilog HDL 中有几类名字空间：其中两类为全局名字空间，其余为局部名字空间。全局名字空间是定义和文本宏。定义名字空间统一了所有 module(模块)和 primitive(原语)的定义。一旦某个名字用于定义一个模块和原语，那么它将不能再用于声明其他模块或原语。

文本宏名字空间是全局的。由于文本宏字由重音符“`”引导，所以它与其他名字空间区分明确。文本宏名的定义逐行出现在设计单元源程序中，可以重复定义它，即同一宏名后面的定义将覆盖其前面的定义。

本地名字空间包括块、模块、生成块、端口、指定块和属性。一旦在块、模块、端口、生成块或指定块名字空间中定义了名字，则不能在该空间内再次定义该名字(使用相同或不同类型)。在属性名字空间内重新定义名字是合法的。

块名字空间由命名块、函数和任务构造引入。它统一了命名块、函数、任务、参数、命名事件和变量类型声明的定义。变量类型声明包括 reg、integer、time、real 和 realtime 声明。

模块名字空间由模块和原语构造引入。它统一了函数、任务、命名块、模块实例、生成块、参数、命名事件、生成变量(genvars)、网络类型声明和变量类型声明的定义。声明的网络类型包括 wire、wor、wand、tri、trior、triand、tri0、tri1、trireg、uwire、supply0 和 supply1。

生成块名字空间由生成构造引入。它统一了函数、任务、命名块、模块实例、生成块、本地参数、命名事件、生成变量(genvars)、网络类型声明和变量类型声明的定义。

端口名字空间由模块、原语、函数和任务构造引入。它提供了一种在结构定义上位于两个不同名字空间中的两个对象之间的连接的方法。连接可以是单向的(input 或 output)或者双向的(inout)。端口名字空间与模块和块名字空间重叠。本质上，端口名字空间指定不同名字空间中名字之间的连接类型。声明的端口类型包括 input、output 和 inout。端口名字空间引入的端口名字可以通过声明与端口名相同名字的变量或连线重新引入模块名字空间。

指定块名字空间由指定构造引入。

属性名字空间由符号“(*”和“*)”构造包围，只能在属性名字空间中定义和使用属性名字，不能在该名字空间内定义任何其他类型的名字。

5.3.12 设计实例三：可变宽度乘法器的设计和实现

本节将在仿真文件中通过元件例化和 defparam 语句两种方式修改设计文件中乘法器的位宽，以满足不同位宽乘法运算的要求，并通过仿真文件中的测试向量对不同位宽乘法器的功能进行验证。

读者可以在高云云源软件和 ModelSim 软件中通过创建工程来添加代码清单 5-19 给出的设计文件和代码清单 5-20 给出的仿真文件，并对该设计进行设计综合和综合后仿真，以验证设计结果的正确性。

代码清单 5-19 top.v 文件

```
module top                                  // module 关键字定义模块 top
#(parameter width = 4)                      // parameter 关键字定义参数 width
(
   input [width - 1:0] a,                   // input 关键字定义端口 a,参数 width 决定端口宽度
   input [width - 1:0] b,                   // input 关键字定义端口 b,参数 width 决定端口宽度
   output [2 * width - 1:0] x               // output 关键字定义端口 x,参数 width 决定端口宽度
 )/ * synthesis syn_dspstyle = "dsp" */;    // 高云属性 syn_dspstyle 取值 dsp,使用 DSP 块实现
assign x = a * b;                           // assign 关键字将端口 a 和端口 b 输入相乘结果给端口 x
endmodule                                   // 关键字 endmodule 指示模块 top 的结束
```

注：在配套资源\eda_verilog\example_5_8\目录下，用高云云源软件打开 example_5_8.gprj。

代码清单 5-20　test.v 文件

```
`timescale 1ns / 1ps            // `timescale 预编译命令,定义时间精度/分辨率对应于 1ns/1ps
module test;                    // module 关键字定义模块 test
reg [7:0] in1,in2;              // reg 关键字定义 8 位 reg 类型变量 in1, in2
reg [15:0] in3,in4;             // reg 关键字定义 16 位 reg 类型变量 in3, in4
wire [15:0] out1;               // wire 关键字定义 16 位网络类型数据 out1
wire [31:0] out2;               // wire 关键字定义 32 位网络类型数据 out2

/* Verilog 元件例化,将模块 top 例化为 Inst_top_1 */
top #(8) Inst_top_1(            // #(8)修改参数,8 对应 top 模块中的参数 width,
                                // 其值修改为 8
   .a(in1),                     // top 模块的端口 a 映射到 test 模块的 reg 类型变量 in1
   .b(in2),                     // top 模块的端口 b 映射到 test 模块的 reg 类型变量 in2
   .x(out1)                     // top 模块的端口 x 映射到 test 模块的线网络类型数据 out1
);
defparam Inst_top_2.width = 16;// defparam 语句将例化模块 Inst_top_2 中参数 width 值改为 16

/* Verilog 元件例化,将模块 top 例化为 Inst_top_2 */
top Inst_top_2(
   .a(in3),                     // top 模块的端口 a 映射到 test 模块的 reg 类型变量 in3
   .b(in4),                     // top 模块的端口 b 映射到 test 模块的 reg 类型变量 in4
   .x(out2)                     // top 模块的端口 x 映射到 test 模块的线网络类型数据 out2
);

initial                         // initial 关键字声明初始化的区域
begin                           // begin 关键字标识初始化区域的开始,类似 C 语言的"{"
  in1 = 58;                     // 通过 reg 类型变量 in1 给例化元件 Inst_top_1 的端口 a 赋值 58
  in2 = 79;                     // 通过 reg 类型变量 in2 给例化元件 Inst_top_2 的端口 b 赋值 79
  #100;                         // "#"标记时间长度 100,持续 100ns,单位由 timescale 确定
  $display("%d * %d = %d",in1,in2,out1);  // 调用 Verilog 系统任务 display 打印 in1 + in2 的
                                          // 结果 out1
end                             // 关键字 end 标识初始化区域的结束,类似 C 语言的"}"

initial                         // initial 关键字声明初始化的区域
begin                           // begin 关键字标识初始化区域的开始,类似 C 语言的"{"
  in3 <= 1343;                  // 通过 reg 类型变量 in3 给例化元件 Inst_top_2 端口 a 赋值 1343
  in4 <= 2561;                  // 通过 reg 类型变量 in4 给例化元件 Inst_top_2 端口 b 赋值 2561
  #100;                         // "#"标记时间长度 100,持续 100ns,单位由 timescale 确定
  $display("%d * %d = %d",in3,in4,out2);  // 调用 Verilog 系统任务 display 打印 in3 + in4 的
                                          // 结果 out2
end                             // end 关键字标识初始化区域的结束,类似 C 语言的"}"
endmodule                       // endmodule 关键字标识模块 test 的结束
```

注：在配套资源\eda_verilog\example_5_9\目录下，用 ModelSim SE-64 10.4c 打开 postsynth-sim.mpf。

对该设计执行综合后仿真，在 Transcript 窗口打印的信息如图 5.6 所示。在 Wave 窗口显示的结果如图 5.7 所示。

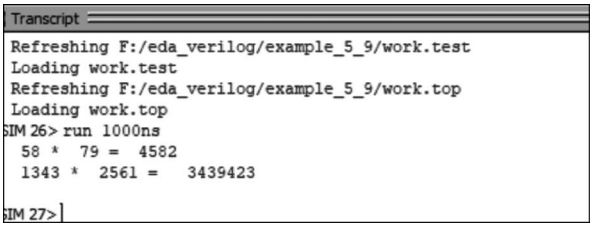

图 5.6　在 Transcript 窗口打印的信息

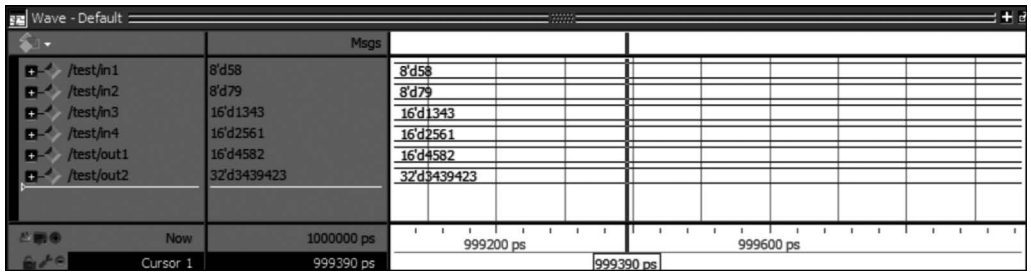

图 5.7　在 Wave 窗口显示的结果

5.4　Verilog HDL 表达式

表达式是将操作数和操作符组合在一起产生结果的一种结构，它可以出现在有任何数值运算的地方。

5.4.1　操作符


视频讲解

Verilog HDL 中的操作符按功能可以分为以下类型，包括算术操作符、关系操作符、相等操作符、逻辑操作符、按位操作符、归约操作符、移位操作符、条件操作符，以及连接和复制操作符。

按运算符所包含操作数的个数可分为三类，包括单个操作符、双操作符和三操作符。

1. Verilog HDL 支持的操作符


视频讲解

表 5.8 给出了 Verilog HDL 支持的操作符。

表 5.8　Verilog HDL 支持的操作符

操作符类型	功　能	操作符类型	功　能
{} {{}}	并置和复制操作符	|	按位或操作符
一元＋　一元－	一元操作符	^	按位异或操作符
＋　－　＊　/　＊＊	算术运算符	^～或～^	按位异或非操作符
％	取模运算符	&	归约与操作符
＞　＞＝　＜　＜＝	关系操作符	～&	归约与非操作符
！	逻辑非操作符	|	归约或操作符
&&	逻辑与操作符	～|	归约或非操作符
||	逻辑或操作符	^	归约异或操作符
＝＝	逻辑相等操作符	^～或～^	归约异或非操作符
！＝	逻辑不相等操作符	＜＜	逻辑左移操作符
＝＝＝	条件(case)相等操作符	＞＞	逻辑右移操作符
！＝＝	条件(case)不相等操作符	＜＜＜	算术左移操作符
～	按位取反操作符	＞＞＞	算术右移操作符
&	按位与操作符	?：	条件操作符

2. 实数支持的操作符

表 5.9 给出了用于实数表达式有效操作符列表。

表 5.9　Verilog HDL 实数支持的操作符

操作符类型	功　能	操作符类型	功　能
一元＋　一元－	一元操作符	！　&&　||	逻辑操作符
＋　－　＊　/　＊＊	算术运算符	＝＝　！＝	逻辑相等操作符
＞　＞＝　＜　＜＝	关系操作符	？＝	条件操作符

3. 操作符的优先级

表 5.10 给出了所有操作符的优先级。同一行内的操作符具有相同的优先级。表中优先级从高到低进行排列。

(1) 除条件操作符从右向左关联外,其余所有操作符自左向右关联。

(2) 当表达式中有不同优先级的操作符时,先执行高优先级的操作符。

(3) 圆括号可用于改变优先级的顺序。

表 5.10 操作符的优先级

操作符类型	优先级顺序
+ - ! ~ & ~& | ~| ^ ~^ ^~ (一元)	最高优先级
**	↓
* / %	
+ - (二元)	
<< >> <<< >>>	
< <= > >=	
== != === !==	
&(二元)	
^ ^~ ~^(二元)	
|(二元)	
&&	
||	
?:(条件操作符)	
{} {{}}	最低优先级

4. 表达式中使用整数

在表达式中,可以使用整数作为操作数,一个整数可以表示为如下几种。

(1) 没有宽度、没有基数的整数,如 12。

(2) 没有宽度、有基数的整数,如'd12、'sd12。

(3) 有宽度、也有基数的整数,如 16'd12、16'sd12。

一个没有基数标识整数的负数值不同于一个带有基数标识的整数。对于没有基数标识的整数,将其理解为以二进制补码存在的负数。带有无符号基数标识的一个整数,将其理解为一个无符号数。

【例 5.39】 整数表达式中使用整数 Verilog HDL 描述的例子。

```
integer IntA;
IntA = -12 / 3;        //结果是 -4
IntA = -'d 12 / 3;     //结果是 1431655761, -12 的 32 位补码是 FFFFFFF4,FFFFFFF4/3 = 1431655761
IntA = -'sd 12 / 3;    //结果是 -4
IntA = -4'sd 12 / 3;   //-4'sd12 是 4 位的负数 1100,等效于 -4。-(-4) = 4。4/3 = 1
```

5. 算术操作符

表 5.11 给出了二元算术操作符的定义。

表 5.11 二元算术操作符的定义

操作类型	功能	操作类型	功能
a+b	a 加 b	a/b	a 除 b
a-b	a 减 b	a%b	a 模 b
a * b	a 乘 b	a ** b	a 的 b 次幂乘

(1) 整数除法截断任何小数部分,如 7/4 的结果为 1。

(2) 对于除法和取模运算,如果第二个操作数为 0,则整个结果的值为 x。

(3) 当第一个操作数除以第二个操作数时,模运算符(例如 y%z)给出余数。模运算的结果应取第一个操作数的符号。

(4) 对于幂乘运算,当其中的任何一个数是实数时,结果的类型也为实数。如果幂乘的第一个操作数为 0,并且第二个操作数不是正数;或者第一个操作数是负数,第二个操作数不是整数,则没有指定其结果。

表 5.12 给出了幂乘操作符规则。

表 5.12 幂乘操作符规则

op2	op1				
	负数<−1	−1	0	1	正数>1
正数	op1 * op2	op2 是奇数→−1 op2 是偶数→1	0	1	op1 * op2
0	1	1	1	1	1
负数	0	op2 是奇数→−1 op2 是偶数→1	x	1	0

(5) 对于一元操作,其优先级大于二元操作。表 5.13 给出了一元操作符。

表 5.13 一元操作符

一元操作符	功 能
+m	一元加 m(和 m 一样)
−m	一元减 m

(6) 在算术操作符中,如果任意操作数的位值是“x”或“z”,那么整个结果为 x。

【例 5.40】 算术操作 Verilog HDL 描述的例子。

```
10 % 3 = 1              // 10 % 3 产生余数为 1
11 % 3 = 2              // 11 % 3 产生余数为 2
12 % 3 = 0              // 12 % 3 不产生余数
-10 % 3 = -1            // 结果的符号与第一个操作数的符号相同
11 % -3 = 2             // 结果的符号与第一个操作数的符号相同
-4'd12 % 3 = 1          // -4'd12 看成一个大的正数,当除以 3 时,余数为 1
3 ** 2 = 9              // 3 乘以 3
2 ** 3 = 8              // 2 乘以 2 乘以 2
2 ** 0 = 1              // 任何数的零指数运算,结果为 1
0 ** 0 = 1              // 零的零指数,结果也为 1
2.0 ** -3'sb1 = 0.5     // 2.0 是实数,给出了实数的倒数
2 ** -3'sb1 = 0         // 2 ** -1 = 1/2,整数除法截断到 0
0 ** -1 = 'bx           // 0 ** -1 = 1/0,整数除零是'bx
9 ** 0.5 = 3.0          // 实数开平方
9.0 ** (1/2) = 1.0      // 整数除法截断指数为 0
-3.0 ** 2.0 = 9.0       // 有定义,因为实数 2.0 仍然是整数值
```

6. 包含 reg 和 integer 的算术表达式

分配给 reg 类型变量或网络的值看作无符号值,除非已经将 reg 类型变量或网络声明为有符号值。分配给 integer、real 或 realtime 变量的值应该看作有符号值。分配给 time 变量的值应该看作无符号值。有符号的值(除了分配给 real 和 realtime 变量之外)应使用二进制补码表示。分配给 real 和 realtime 变量的值应该使用浮点表示。在有符号和无符号之间的转换应保持相同的位,只是对位的解释不同而已。

表 5.14 给出了算术操作数对数据类型的理解。

表 5.14 算术操作数对数据类型的理解

数据类型	理　解	数据类型	理　解
无符号网络	无符号	整数	有符号,二进制补码
有符号网络	有符号,二进制补码	时间	无符号
无符号 reg	无符号	实数、实时时间	有符号,浮点
有符号 reg	有符号,二进制补码		

【例 5.41】 在表达式中使用 integer 和 reg 数据类型 Verilog HDL 描述的例子。

```
integer intA;
reg [15:0] regA;
reg signed [15:0] regS;
intA = -4'd12;
regA = intA / 3;        //表达式是 -4, intA 是 integer 数据类型,regA 的值是 65532
regA = -4'd12;          //regA 的值是 65524
intA = regA / 3;        //表达式的值为 21841,regA 是 reg 类型的数据
intA = -4'd12 / 3;      //表达式的结果为 1431655761,是一个 32 位的寄存器数据
regA = -12 / 3;         //表达式结果为 -4,一个整数类型,regA 的值是 65532
regS = -12 / 3;         //表达式结果为 -4。regS 是有符号 reg 类型变量
regS = -4'sd12 / 3;     //表达式结果为 1。-4'sd12 为 4
```

7. 关系操作符

表 5.15 给出了关系操作符列表。

表 5.15 关系操作符列表

关系操作符类型	功　能	关系操作符类型	功　能
a<b	a 小于 b	a<=b	a 小于或等于 b
a>b	a 大于 b	a>=b	a 大于或等于 b

关系操作符有如下特点。

(1) 关系操作符的结果为真("1")或假("0")。

(2) 如果操作数中有一位为"X"或"Z",那么结果为一位的"X"。

(3) 如果关系运算存在无符号数,则将表达式看作无符号数。当操作数位宽不同时,则位宽较小的操作数将零扩展到位宽较大操作数宽度范围。

(4) 如果关系运算都是有符号数,则将表达式看作有符号的。当操作数位宽不同时,则位宽较短的操作数将符号扩展到位宽较大操作数的宽度范围。

(5) 所有关系运算符的优先级相同,但是比算术运算符的优先级要低。

(6) 如果操作数中有实数,则将所有操作数都转换为实数,然后再进行关系运算。

【例 5.42】 关系操作符 Verilog HDL 描述的例子。

```
a < foo - 1             //等价于 a < (foo - 1)
foo - (1 < a)           //不等价于 foo - 1 < a
```

8. 相等操作符

表 5.16 列出了相等操作符。

表 5.16 相等操作符列表

相等操作符类型	功　能
a===b	a 等于 b,包含 x 和 z
a!==b	a 不等于 b,包含 x 和 z
a==b	a 等于 b,结果可能未知(比较不包含 x 和 z)
a!=b	a 不等于 b,结果可能未知(比较不包含 x 和 z)

相等操作符有如下特点。

(1) 相等操作符有相同的优先级。

(2) 如果相等操作中存在无符号数,则将表达式看作无符号数。当操作数位宽不同时,则位宽较小的操作数将0扩展到位宽较大操作数的宽度范围。

(3) 如果相等操作都是有符号数,则将表达式看作有符号数。当操作数位宽不同时,则位宽较小的操作数将符号扩展到位宽较大操作数的宽度范围。

(4) 如果操作数中有实数,则将所有操作数都转换为实数,然后再比较两个实数。

(5) 对于逻辑相等和逻辑不相等的操作符("==="和"!="),如果由于操作数中的不确定("x")和高阻("z")位,关系是不明确的,则结果应该是一位未知的值("x")。

(6) 对于case中的相等和不相等操作符("===="和"!=="),应按照过程case语句中的方式进行比较。比较中应包含"x"或"z"位,结果的匹配应看作相等。这些运算符的结果始终为确定的值,或者是"0"或者是"1"。

9. 逻辑操作符

符号&&(逻辑与)和符号||(逻辑或)用于逻辑的连接。逻辑比较的结果为"1"(真)或者"0"(假)。当结果模糊的时候,为"x"。&&(逻辑与)的优先级大于||(逻辑或)。逻辑操作的优先级低于关系和相等操作。

符号!(逻辑非)是一元操作符。该操作符将非零或真操作数转换为0,将零或假操作数转换为1,模糊的真值保持为x。

【例5.43】 逻辑操作Verilog HDL描述的例子1。

假设alpha=127,beta=0。

```
regA = alpha && beta;  //regA 设置为 0
regB = alpha || beta;  //regB 设置为 1
```

【例5.44】 逻辑操作Verilog HDL描述的例子2。

```
a < size - 1 && b != c && index != lastone
```

为了便于理解和查看设计,推荐使用下面的方法描述上面的逻辑操作:

```
(a < size - 1) && (b != c) && (index != lastone)
```

【例5.45】 逻辑操作Verilog HDL描述的例子3。

```
if (!inword)
```

也可以表示为

```
if (inword == 0)
```

10. 按位操作符

表5.17给出了对于不同操作符按位操作的结果。

表5.17 不同操作符按位操作的结果

&(二元按位与)	0	1	x	z	|(二元按位或)	0	1	x	z
0	0	0	0	0	0	0	1	x	x
1	0	1	x	x	1	1	1	1	1
x	0	x	x	x	x	x	1	x	x
z	0	x	x	x	z	x	1	x	x
^(二元按位异或)	0	1	x	z	^~(二元按位异或非)	0	1	x	z
0	0	1	x	x	0	1	0	x	x
1	1	0	x	x	1	0	1	x	x
x	x	x	x	x	x	x	x	x	x
z	x	x	x	x	z	x	x	x	x
~(一元非)	1	0	x	x					

如果操作数位宽不相等，位宽较小的操作数在最左侧添 0 补位。

11. 归约操作符

归约操作符在单个操作数的所有位上操作，并产生一位结果。归约操作符如下。

1）&.（归约与）

（1）如果存在位值为“0”，那么结果为“0”。

（2）如果存在位值为“x”或“z”，结果为“x”。

（3）其他情况结果为“1”。

2）～&.（归约与非）

与归约操作符 &. 相反。

3）|（归约或）

（1）如果存在位值为“1”，那么结果为“1”。

（2）如果存在位“x”或“z”，结果为“x”。

（3）其他情况结果为“0”。

4）～|（归约或非）

与归约操作符|相反。

5）^（归约异或）

（1）如果存在位值为“x”或“z”，那么结果为“x”。

（2）如果操作数中有偶数个“1”，结果为“0”。

（3）其他情况结果为“1”。

6）～^（归约异或非）

与归约操作符^相反。

归约异或操作符用于决定向量中是否有位为“x”。

表 5.18 给出了一元归约操作结果。

表 5.18　一元归约操作结果的列表

操作数	&.	～&.	|	～|	^	～^
4'b0000	0	1	0	1	0	1
4'b1111	1	0	1	0	0	1
4'b0110	0	1	1	0	0	1
4'b1000	0	1	1	0	1	0

12. 移位操作符

移位操作符有两种类型：逻辑移位操作符（“<<”和“>>”）及算术移位操作符（“<<<”和“>>>”）。

左移操作符（“<<”和“<<<”）应将其左侧操作数向左移动由右侧操作数所指定的位数。对于左移操作符，当向左移动时，空出的位数由“0”填充。

右移操作符（“>>”和“>>>”）应将其左侧操作数向右移动由右侧操作数所指定的位数。对于逻辑右移，应使用“0”填充空出位的位置。对于算术右移，分为两种情况。

（1）如果结果类型是无符号的，算术右移将用“0”填充空出位。

（2）如果结果类型是有符号的，则使用左操作数的最高有效位（即符号位）的值来填充空出位。

如果右操作数具有“x”或“z”值，则结果不确定。右操作数总是看作无符号数，对结果的有符号性没有影响。结果有符号性由左侧操作数和表达式的剩余部分决定。

【例 5.46】 移位操作符 Verilog HDL 描述的例子 1 如代码清单 5-21 所示。

代码清单 5-21 移位操作的 Verilog HDL 描述(1)

```
module shift;
reg[3:0] start, result;
initial begin
  start = 1;
  result = (start << 2);        //假设 start 的值为"0001",移位结果是"0100"
end
endmodule
```

【例 5.47】 移位操作符 Verilog HDL 描述的例子 2 如代码清单 5-22 所示。

代码清单 5-22 移位操作的 Verilog HDL 描述(2)

```
module ashift;
reg signed [3:0] start, result;
initial begin
  start = 4'b1000;
  result = (start >>> 2);       //假设 start 的值为"1000",移位结果是"1110"
end
endmodule
```

13. 条件操作符

条件操作符,也称为三目操作符,应为右关联运算符。该操作符根据条件表达式的值选择表达式,形式如下:

```
cond_expr ? expr1:expr2
```

(1) 如果 cond_expr 为真(值为“1”),选择 expr1。

(2) 如果 cond_expr 为假(值为“0”),选择 expr2。

(3) 如果 cond_expr 为“x”或“z”,结果是按以下逻辑将 expr1 和 expr2 按位操作的值:“0”与“0”得“0”;“1”与“1”得“1”;其余情况为“x”。

【例 5.48】 条件操作符 Verilog HDL 描述的例子。

```
wire [15:0] busa = drive_busa ? data : 16'bz;
```

当 drive_busa 为”1”时,data 驱动总线 busa;如果 drive_busa 为“x”,则一个不确定值“x”驱动总线 busa;否则,未驱动 busa。

14. 连接和复制操作符

连接(也称为并置)操作是将位宽较小的表达式合并形成位宽较大的表达式的一种操作,其描述格式如下:

```
{expr1,expr2,...,exprN}
```

由于非定长常数的长度未知,因此不允许连接非定长常数。

一个只能用于连接的操作符是复制,复制就是将一个表达式复制多次的操作,其描述格式如下:

```
{ replication_constant {expr}}
```

其中,replication_constant 为非负数,它是非“z”和非“x”的常数,用于表示复制的次数;expr 为需要复制的表达式。

注:包含有复制的连接表达式,不能出现在分配的左侧操作数,也不能连接到 output 或者 input 端口上。

【例 5.49】 连接操作 Verilog HDL 描述的例子。

```
{a, b[3:0], w, 3'b101}
```

等效于

```
{a, b[3], b[2], b[1], b[0], w, 1'b1, 1'b0, 1'b1}
```

【例 5.50】 复制操作 Verilog HDL 描述的例子。

```
{4{w}}
```

等效于

```
{w, w, w, w}
```

【例 5.51】 复制和连接操作 Verilog HDL 描述的例子。

```
{b, {3{a, b}}}
```

等效于

```
{b, a, b, a, b, a, b}
```

复制操作可以复制值为 0 的常数,在参数化代码时非常有用。带有 0 复制常数的复制,被认为是大小为 0,并且被忽略。这样一个复制,只能出现在至少有一个连接操作数是正数的连接中。

【例 5.52】 复制和连接操作分配限制 Verilog HDL 描述的例子。

```
parameter P = 32;
//下面对于 1 到 32 是有效的
assign b[31:0] = { {32 - P{1'b1}}, a[P - 1:0] } ;
//对于 P = 32 来说,下面是非法的,因为 0 复制单独出现在一个连接中
assign c[31:0] = { {{32 - P{1'b1}}}, a[P - 1:0] }
//对 P = 32 来说,下面是非法的
initial
    $displayb({32 - P{1'b1}}, a[P - 1:0]);
```

【例 5.53】 复制操作 Verilog HDL 描述的例子。

```
result = {4{func(w)}} ;
```

等效于

```
y = func(w) ;
result = {y, y, y, y} ;
```


视频讲解

5.4.2 操作数

可以在表达式中指定几种类型的操作数。最简单的类型是对完整形式的网络、变量或参数的引用。也就是说,只给出网络、变量或参数的名字,在这种情况下,组成网络、变量或参数值的所有位都用作操作数。

如果需要求一个向量网络、向量 reg、integer、time 变量或参数的单个位,则要使用位选择操作数。如果需要求一个向量网络、向量 reg、integer、time 变量或参数的某些相邻的位,则要使用部分选择操作数。

数组元素或数组元素的位选择或部分选择可以被引用为操作数。可以将其他操作数的并置/连接(包括嵌套并置/连接)指定为操作数。函数调用是操作数。

1. 向量位选择和部分选择寻址

位选择从向量网络、向量 reg、integer、time 变量或参数中提取特定的位,该位可以使用表达式进行寻址。如果位选择/部分选择超出地址范围,或者位选择为"x"或"z",则返回的结果为"x"。向量网络、向量 reg、integer、time 变量或参数中的几个连续位可以寻址,称为部分选择。

对于部分选择,有以下两种类型。

(1) 向量寄存器或者网络的常数部分选择,表示为

```
vect[msb_expr:lsb_expr]
```

其中,msb_expr 和 lsb_expr 为常数的整数表达式。

(2) 向量网络、向量寄存器、时间变量或者参数的索引部分选择,表示为

```
reg[15:0] big_vect;
reg[0:15] little_vect;
big_vect[lsb_base_expr + : width_expr]
little_vect[msb_base_expr + : width_expr]
big_vect[msb_base_expr - : width_expr]
little_vect[lsb_base_expr - : width_expr]
```

其中,msb_base_expr 和 lsb_base_expr 为常数的整数表达式,可以在运行时改变值; width_expr 为正的宽度表达式。

对于下面这两个方式选择从 base_expr 开始并按位升序排列的位,所选择的位数等于宽度表达式。

```
big_vect[lsb_base_expr + : width_expr]
little_vect[msb_base_expr + : width_expr]
```

对于下面这两个方式选择从 base_expr 开始并按位降序排列的位,所选择的位数等于宽度表达式。

```
big_vect[msb_base_expr - : width_expr]
little_vect[lsb_base_expr - : width_expr]
```

【例 5.54】 数组部分选择 Verilog HDL 描述的例子。

```
reg[31: 0] big_vect;
reg [0 :31] little_vect;
reg[63:0] dword;
integer sel;
big_vect[ 0 + : 8]      // == big_vect[ 7 : 0]
big_vect[15 - : 8]      // == big_vect[15 : 8]
little_vect[ 0 + : 8]   // == little_vect[0 : 7]
little_vect[15 - : 8]   // == little_vect[8 :15]
dword[8 * sel + : 8]    //带有固定宽度的变量部分选择
```

【例 5.55】 数组初始化、位选择和部分选择 Verilog HDL 描述的例子。

```
reg[7:0] vect;
vect = 4;               //用"00000100"填充,msb 是 7,lsb 是 0
```

(1) 如果 adder 值为 2,则 vect[addr]返回“1”;

(2) 如果 addr 超过范围,则 vect[addr]返回“x”;

(3) 如果 addr 为 0、1、3～7,则 vect[addr]返回“0”;

(4) vect[3:0]返回“0100”;

(5) vect[5:1]返回“00010”;

(6) vect[返回 x 的表达式]返回“x”;

(7) vect[返回 z 的表达式]返回“x”;

(8) 如果 addr 的任何一位是“x”或者“z”,则 addr 的值为“x”。

2. 数组和存储器寻址

对于下面的存储器声明,表示为 8 位宽度、1024 个深度:

```
reg[7:0] mem_name[0:1023];
```

存储器地址的语法应包含存储器名字 mem_name 和地址表达式 addr_expr:

```
mem_name[addr_expr]
```

其中，addr_expr 为任意整数表达式。例如：

```
mem_name[mem_name[3]]
```

表示存储器的间接寻址。

【例 5.56】 存储器寻址 Verilog HDL 描述的例子。

```
reg[7:0] twod_array[0:255][0:255];          //声明 256×256 的 8 位的数组
wire threed_array[0:255][0:255][0:7];       //声明 256×256×8 的 1 位的数组
twod_array[14][1][3:0];                     //访问字的低 4 位
twod_array[1][3][6];                        //访问字的第 6 位
twod_array[1][3][sel];                      //使用可变的位选择
threed_array[14][1][3:0];                   //非法
```

3. 字符串

字符串是双引号内的字符序列，用一串 8 位二进制 ASCII 码的形式表示，每个 8 位二进制 ASCII 码代表一个字符。任何 Verilog HDL 的操作符均可操作字符串。当给字符串所分配的值小于所声明字符串的宽度时，用 0 补齐左侧。

Verilog HDL 操作符支持的字符串操作包括复制、连接和比较：①通过分配实现复制；②通过连接操作符实现连接；③通过相等操作符实现比较。

当操作向量 reg 内的字符串的值时，reg 应该至少为 8×n 位(n 是 ASCII 字符的个数)，用于保存 n 个 8 位的 ASCII 码。

【例 5.57】 字符串连接的 Verilog HDL 描述。

```
reg[8*10:1]s1,s2;
initial begin
s1 = "Hello";
s2 = " world!";
if ({s1,s2} == "Hello world!")
 $display("strings are equal");
end
```

该例子中的比较失败，这是因为在给字符串变量分配值的时候按下面进行填充：

```
        s1 = 000000000048656c6c6f
        s2 = 00000020776f726c6421
{s1,s2} = 000000000048656c6c6f00000020776f726c6421
```

注：对于空字符串""来说，将其看作 ASCII 中的 NUL("\0")，其值为 0，而不是字符串"0"。


视频讲解

5.4.3 延迟表达式

Verilog HDL 中，延迟表达式的格式为用圆括号括起来的三个表达式，这三个表达式之间用冒号分隔开。三个表达式依次代表最小、典型和最大延迟时间值。

【例 5.58】 延迟表达式 Verilog HDL 描述的例子。

```
(a:b:c) + (d:e:f)
```

表示：

(1) 最小延迟值为 a+d 的和；

(2) 典型延迟值为 b+e 的和；

(3) 最大延迟值为 c+f 的和。

【例 5.59】 分配 min:typ:max 格式值 Verilog HDL 描述的例子。

```
val - (32'd 50: 32'd 75: 32'd 100)
```

5.4.4 表达式的位宽

为了使表达式求值得到一致性的结果,控制表达式的位宽非常重要。某些情况下可采取最简单的解决方法,例如,如果指定了两个16位的reg类型向量的按位与操作,那么结果就是一个16位的值。然而,在某些情况下,究竟有多少位参与表达式求值或者结果有多少位,并不容易看出来。例如,两个16位操作数之间的算术加法,是应该使用16位求值还是该使用17位(允许进位位溢出)求值?答案取决于建模器件的类型以及该器件是否处理进位位溢出。Verilog HDL利用操作数的位宽决定参与表达式求值的位数。

控制表达式位宽的规则已经制定好,因此大多数实际情况下都有一个简单的解决方法。

(1) 表达式位宽由包含在表达式内的操作数和表达式的上下文决定。

(2) 自主表达式的位宽由它自身单独决定,如延迟表达式。

(3) 上下文决定型表达式的位宽由该表达式自己的位宽以及它是另一个表达式的一部分这一事实来确定。例如,一个分配操作中右侧表达式的位宽由它自己的位宽和分配符左侧的位宽来决定。

表5.19说明了表达式的形式如何决定表达式结果的位宽,表中i、j和k表示操作数的表达式,L(i)表示表达式i的位宽,op表示操作符。

表5.19 表达式位宽规则

表 达 式	结果值的位宽	说 明
不定长常数	与整数相同	
定长常数	与给定的位宽相同	
i op j,操作符op为:+、-、*、/、%、&、|、^、^~或~^	max(L(i),L(j))	
op i,操作符op为:+、-、~	L(i)	
i op j,操作符op为:===、!==、==、!=、&&、||、>、>=、<或<=	1位	在求表达式的值时,每个操作数的位宽都先变为max(L(i),L(j))
i op i,操作符op为:&、~&、|、~|、^、^~或~^	1位	所有操作数都是自主表达式
i op j,操作符op为:>>、<<、**、>>>或<<<	L(i)	j是自主表达式
i? j:k	max(L(j),L(k))	i是自主表达式
{i,…,j}	L(i)+…+L(j))	所有操作数都是自主表达式
{i{j,…,k}}	I*(L(i)+…+L(j))	所有操作数都是自主表达式

在表达式求值过程中,中间结果应当采用最大操作数的位宽(如果是在赋值语句中,也包括赋值符左侧),在表达式求值过程中要注意避免数据的丢失。

【例5.60】 位长度问题Verilog HDL描述的例子。

```
reg [15:0] a, b, answer;                    // 16位reg类型
answer = (a + b)>> 1;                       // 不能正常操作
```

其中,a和b相加可能导致溢出,然后右移1位以保留16位答案中的进位。然而出现了一个问题,因为表达式中的所有操作都是16位宽度,所以,表达式(a+b)产生16位宽的中间结果,从而在评估执行1位右移操作之前丢失了进位。

解决方法是强制表达式(a+b)使用至少17位进行求值。例如,将整数值0添加到表达式将使用整数的位执行计算。下面的例子将产生预期的结果:

```
answer = (a + b + 0)>> 1;                   // 将正常操作
```

【例 5.61】 自主表达式 Verilog HDL 描述的例子如代码清单 5-23 所示。

代码清单 5-23 自主表达式的 Verilog HDL 描述

```
reg[3:0] a;
reg[5:0] b;
reg[15:0] c;
initial begin
    a = 4'hF;
    b = 6'hA;
    $display("a * b = %h", a * b);              //表达式的宽度由自己确定
    c = {a ** b};                               //由于使用连接符,表达式的宽度由 a ** b 确定
    $display("a ** b = %h", c);
    c = a ** b;                                 //表达式的宽度由 c 确定
    $display("c = %h", c);
end
```

仿真器的输出结果如下：

```
a * b = 16                                      //由于位宽为 6 位,所以'h96 被截断到'h16
a ** b = 1                                      //表达式的位宽为 4 位(a 的位宽)
c = ac61                                        //表达式的位宽为 16 位(c 的位宽)
```

5.4.5 有符号表达式

为了得到一致性的结果,控制表达式的符号非常重要。可以使用两个系统函数来处理类型的表示。

(1) $signed()返回相同位宽的有符号值。

(2) $unsigned()返回相同位宽的无符号值。

【例 5.62】 调用系统函数进行符号转换 Verilog HDL 描述的例子。

```
reg[7:0] regA, regB;
reg signed [7:0] regS;
regA = $unsigned( - 4);                         //regA = 8'b11111100
regB = $unsigned( - 4'sd4);                     //regB = 8'b00001100
regS = $signed (4'b1100);                       //regS = - 4
```

下面是表达式符号类型规则。

(1) 表达式的符号类型仅仅取决于操作数,与左侧值无关。

(2) 十进制数是有符号数。

(3) 基数格式数值是无符号数,除非符号(s)用于基数说明。

(4) 无论操作数是何类型,其位选择结果为无符号型。

(5) 无论操作数是何类型,其部分位选择结果为无符号型,即使部分位选择指定了一个完整的向量。

(6) 无论操作数是何类型,连接(或复制)操作的结果为无符号型。

(7) 无论操作数是何类型,比较操作的结果(“1”或“0”)为无符号型。

(8) 通过强制类型转换为整型的实数为有符号型。

(9) 任何自主操作数的符号和位宽由操作数自己决定,不取决于表达式的其余部分。

(10) 对于非自主操作数遵循下面的规则：①如果任何操作数为实数,则结果为实数;②如果任何操作数为无符号,则结果为无符号,且与操作符无关;③如果所有操作数都有符号,则结果都有符号,除非另有指定。

5.4.6 分配和截断

如果右操作数的位宽大于左操作数的位宽,则右操作数的最高有效位会丢失,以进行位宽匹配。当出现位宽不匹配时,并不要求实现过程警告或者报告与分配位宽不匹配的任何错误。

截断符号表达式的符号位,可能会改变结果的符号。

【例 5.63】 位宽不匹配分配 Verilog HDL 描述的例子 1。

```
reg[5:0] a;
reg signed[4:0] b;
initial begin
    a = 8'hff;                          //分配完后,a = 6'h3f
    b = 8'hff;                          //分配完后,b = 5'h1f
end
```

【例 5.64】 位宽不匹配分配 Verilog HDL 描述的例子 2。

```
reg[0:5] a;
reg signed [0:4] b, c;
initial begin
    a = 8'sh8f;                         //分配完后,a = 6'h0f
    b = 8'sh8f;                         //分配完后,b = 5'h0f
    c =  -113;                          //分配完后,c = 15
end
```

【例 5.65】 位宽不匹配分配 Verilog HDL 描述的例子 3。

```
reg[7:0] a;
reg signed [7:0] b;
reg signed [5:0] c, d;
initial begin
    a = 8'hff;
    c = a;                              //分配完后,c = 6'h3f
    b =  -113;
    d = b;                              //分配完后,d = 6'h0f
end
```

5.5 Verilog HDL 分配

分配(也称为赋值)是最简单的机制,用于给网络和变量设置相应的值。Verilog HDL 提供了两种基本形式的分配。

(1) 连续分配,用于给网络分配值。

(2) 过程分配,用于给变量分配值。

Verilog HDL 还额外提供了两种分配形式: assign/deassign 和 force/release,称为过程连续分配。

一个分配由两部分构成,包括左侧和右侧,它们通过“=”分隔,或者在非阻塞过程赋值中,使用“<=”分隔。右侧可以是任意表达式。左侧表示要分配右侧值的变量。左侧可以采用表 5.20 给出的形式之一,这取决于分配是连续分配还是过程分配。

表 5.20 分配描述中的有效的左侧格式

描 述 类 型	左 侧
连续分配	1. 网络(标量或向量) 2. 向量网络的常数位选择 3. 向量网络的常数部分选择 4. 向量网络的常数索引的部分选择 5. 以上任何左侧的连接或者嵌套的连接

续表

描 述 类 型	左　　侧
过程分配	1. 变量(标量或向量) 2. 向量 reg、整数或者时间变量的比特选择 3. 向量 reg、整数或者时间变量的部分选择 4. 向量 reg、整数或时间变量索引的部分选择 5. 存储器字 6. 以上任何左侧的连接或者嵌套的连接


视频讲解

5.5.1 连续分配

连续分配/赋值将值驱动到网络上,包括向量和标量。每当右侧的值发生变化时,应进行分配。连续分配提供了一种在不指定门互连的情况下建模组合逻辑的方法。相反,该模型指定驱动网络的逻辑表达式。

1. 网络声明分配

前面讨论了声明网络的两种方法,这里给出第三种方法,即网络声明分配。在声明网络的相同描述中,允许在网络上使用连续分配。

【例 5.66】 连续分配的网络声明格式 Verilog HDL 描述的例子。

```
wire mynet = enable ;
```

注:由于一个网络只能声明一次,所以对于一个特定的网络来说,只能有一个网络声明分配。这与连续分配描述是不一样的。在连续分配描述中,一个网络可以接受连续分配形式的多个分配。

2. 连续分配语句

连续分配将为一个网络数据类型设置一个值。网络可能明确的声明,或者根据隐含声明规则继承一个隐含声明。

给一个网络进行分配是连续的和自动的。换句话说,任何时候,只要右侧的一个操作表达式的操作数发生变化,则将改变整个右侧表达式。如果右侧表达式新的值和以前的值不同,则将给左侧分配新的值。

连续分配描述格式为

```
assign  variable = expression;
```

其中,variable 为网络类型信号; expression 为赋值表达式。

【例 5.67】 使用连续分配实现带进位的 4 位加法器 Verilog HDL 描述的例子如代码清单 5-24 所示。

代码清单 5-24　包含进位的 4 位加法器的 Verilog HDL 描述

```
module adder (sum_out, carry_out, carry_in, ina, inb);
output [3:0] sum_out;
output carry_out;
input [3:0] ina, inb;
input carry_in;
wire carry_out, carry_in;
wire [3:0] sum_out, ina, inb;
assign {carry_out, sum_out} = ina + inb + carry_in;
endmodule
```

【例 5.68】 使用连续分配实现 4∶1 的 16 位总线多路复用器的 Verilog HDL 描述的例子,如代码清单 5-25 所示。

代码清单 5-25 4∶1 的 16 位总线多路复用器的 Verilog HDL 描述

```
module select_bus(busout, bus0, bus1, bus2, bus3, enable, s);
parameter n = 16;
parameter Zee = 16'bz;
output [1:n] busout;
input [1:n] bus0, bus1, bus2, bus3;
input enable;
input [1:2] s;
tri [1:n] data;                                    //声明网络
tri [1:n] busout = enable ? data : Zee;            //包含连续分配的网络声明
//包含四个连续分配的分配描述
assign
data = (s == 0) ? bus0 : Zee,
data = (s == 1) ? bus1 : Zee,
data = (s == 2) ? bus2 : Zee,
data = (s == 3) ? bus3 : Zee;
```

3. 延迟

在连续分配中,延迟确定将右侧操作数的变化分配到左侧的时间间隔。如果左侧是一个标量网络,这种分配的效果和门延迟是一样的,即可以为输出上升、下降和改变为高阻指定不同的延迟。

如果左边是一个向量网络,则可以应用最多三个延迟。下面的规则用于确定哪个延迟用于控制分配。

(1) 如果右边从非零变化到零,则应该使用下降延迟。

(2) 如果右边变化到高阻,则应该使用关闭延迟。

(3) 对于其他情况,应该使用上升延迟。

注:在连续分配中指定延迟,是网络声明的一部分,它用于指定一个网络延迟。这与指定一个延迟再为网络进行连续分配是不同的。在一个网络声明中,可以将一个延迟值应用于一个网络中。

【例 5.69】 一个延迟值应用到一个网络 Verilog HDL 描述的例子。

```
wire #10 wireA;
```

这描述了任何改变的值,需要延迟 10 个时间单位后,才能应用到 wireA 网络。

注:对于一个向量网络的分配,当在声明中包含分配时,不能将上升延迟和下降延迟应用到单个的位。

4. 强度

用户可以在一个连续分配中指定驱动强度。高云云源软件综合工具将忽略强度的定义。这只应用于为下面类型的标量网络进行分配的情况:

wire	tri	trireg
wand	triand	tri0
wor	trior	tri1

在网络声明或通过使用 assign 关键字在一个单独的分配中指定连续分配驱动强度。

如果提供了强度说明,应该紧跟关键字(用户网络类型的关键字或者 assign),并且在任何指定的延迟前面。当连续分配驱动网络时,应该按照指定的值进行仿真。

一个驱动强度说明应该包含一个强度值,当给网络分配的值是“1”时使用第一个强度值;当分配的值是“0”时使用第二个强度值。下面的关键字,用于为分配“1”指定强度值:

supply1 strong1 pull1 weak1 highz1

下面的关键字,用于为分配“0”指定强度值:

supply0　strong0　pull0　weak0　highz0

两个强度说明的顺序是任意的。下面两个规则将约束强度说明。

(1) 强度说明(highz1,highz0)和(highz0,highz1)认为是非法的结构。

(2) 如果没有指定驱动强度,它将默认为(strong1,strong0)。

5.5.2 过程分配

连续分配驱动网络的行为类似于逻辑门驱动网络,右侧的分配表达式可以认为是连续驱动网络的组合逻辑电路。与之不同的是,过程分配为变量赋值。过程分配没有持续性,变量一直保存着上一次分配的值,直到下一次为变量进行了新的过程分配为止。

过程分配发生在下面的过程中,如 always、initial、task 和 function 中,可以认为是触发式的分配。当仿真中的执行流到达过程中的赋值时,发生触发。事件控制、延迟控制、if 语句、case 语句和循环语句都可以控制是否评估赋值。

变量声明分配是过程分配的一个特殊情况,用于给变量分配一个值。它允许在用于声明变量的相同语句中,给变量分配一个初值。过程分配应该是一个常数表达式,该分配没有连续型,取而代之的是,变量一直保持该分配值,直到下一个新的分配到来。不能对一个数组使用变量声明分配,变量声明分配只能用于模块级,如果在 initial 模块和变量声明分配中,为相同的变量分配了不同的值,没有定义评估的顺序。

【例 5.70】 定义一个 4 位变量并且分配初值 Verilog HDL 描述的例子。

```
reg[3:0] a = 4'h4;
```

等价于

```
reg[3:0] a;
initial a = 4'h4;
```

【例 5.71】 为数组分配初值非法 Verilog HDL 描述的例子。

```
reg[3:0] array[3:0] = 0;
```

【例 5.72】 声明两个整数,第一个分配值为 0 的 Verilog HDL 描述的例子。

```
integer i = 0, j;
```

【例 5.73】 声明两个实数变量,为其分配值为 2.5 和 300000 的 Verilog HDL 描述的例子。

```
real r1 = 2.5, n300k = 3E6;
```

【例 5.74】 声明一个时间变量和一个实时时间变量,并分配初值 Verilog HDL 描述的例子。

```
time t1 = 25;
realtime rt1 = 2.5;
```

5.6 Verilog HDL 门级描述

本节介绍 Verilog HDL 提供的内置门级建模原语以及在一个设计中使用这些原语的方法。高云云源软件综合工具不支持开关级电路和 pull 门。

Verilog HDL 预定义了 14 个逻辑门原语,用于提供门级电路建模工具。使用门级建模的优势包括:

(1) 在真实门电路和模型之间,门提供了接近于一对一的映射;

(2) 不存在等效于双向传输门的连续分配。

5.6.1 门声明


视频讲解

对一个门的例化声明应该包含下面的说明。

(1) 用于命名门原语类型的关键字。

(2) 驱动强度(可选)。

(3) 传输延迟(可选)。

(4) 命名门实例的标识符(可选)。

(5) 实例数组的范围(可选)。

(6) 终端连接列表。

1. 门类型说明

表 5.21 列出了 Verilog HDL 提供的内建门列表。

表 5.21 Verilog HDL 提供的内建门列表

n 输入门	n 输出门	三 态 门	pull 门
and	buf	bufif0	pulldown
nand	not	bufif1	pullup
nor		notif0	
or		notif1	
xnor			
xor			

2. 驱动强度说明

驱动强度指定了门例化输出终端逻辑值的强度,高云云源软件综合工具忽略驱动强度说明。表 5.22 给出了可以使用驱动强度说明的门类型。

表 5.22 可以使用驱动强度说明的门类型

and	nand	buf	not	pulldown
or	nor	bufif0	notif0	pullup
xor	xnor	bufif1	notif1	

用于一个门例化的驱动强度说明,除了 pullup 和 pulldown 以外,应该有 strength1 说明和 strength0 说明。strength1 说明指定了逻辑 1 的信号强度; strength0 说明指定了逻辑 0 的信号强度。驱动强度在门类型关键字后,在延迟说明的前面。strength0 说明可以在 strength1 之后,也可以在其之前。在圆括号内,通过逗号,将 strength0 说明和 strength1 说明分隔。

(1) pullup 门只有 strength1 说明,strength0 说明是可选的。

(2) pulldown 门只有 strength0 说明,strength1 说明是可选的。

注:

(1) strength1 包含下面的关键字:

supply1 strong1 pull1 weak1

(2) strength0 包含下面的关键字:

supply0 strong0 pull0 weak0

(3) 将 strength1 指定为 highz1,将引起门或者开关输出一个逻辑值 z,而不是 1; 将 strength0 指定为 highz0,将引起门或者开关输出一个逻辑值 z,而不是 0。强度说明(highz0, highz1)和(highz1, highz0)是无效的。

【例 5.75】 下面给出了一个集电极开路 nor 门 Verilog HDL 描述的例子。

```
nor (highz1,strong0) n1(out1,in1,in2);
```

在这个例子中，nor 逻辑门输出 z，而不是 1。

3. 延迟说明

在一个声明中，可选的延迟说明指定了贯穿门和开关的传播延迟。高云云源软件工具将忽略延迟说明。可用于行为级仿真，如果在声明中没有指定门和开关延迟说明，则没有传播延迟。根据门的类型，一个延迟说明最多包含三个延迟值。pullup 和 pulldown 例化声明将不包含延迟描述。

4. 原语例化标识符

可以为门实例指定一个可选的名字。如果以数组的形式声明了多个例化，则需要使用一个标识符来命名例化。

5. 范围说明

当要求重复例化的时候，这些例化之间是不同的。通过向量索引的连接来区分它们。

为了指定一个例化数组，例化的名字后面应该跟着范围，使用两个常数表达式指定范围、左侧索引（lhi）和右侧索引（rhi）；它们通过“[ ]”中的“:”分隔；范围[lhi : rhi]，表示 abs(lhi-rhi)+1 例化数组。

注：一个例化数组的范围应该是连续的。一个例化标识符只关联一个范围，用于声明例化数组。

6. 原语例化连接列表

终端列表描述了门或开关连接模型剩余部分的方法。门和开关的类型限定了表达式。连接列表通过“()”括起来，“()”号内的端口通过“,”分隔。输出或者双向终端总是出现在连接列表的开始，后面跟着输入。

对于

```
nand #2 nand_array[1:4]( ... ) ;
```

声明了四个实例，作为 nand_array[1]、nand_array[2]、nand_array[3]和 nand_array[4]标识符进行引用。

【例 5.76】 两个等效门例化 Verilog HDL 描述的例子 1 如代码清单 5-26 所示。

代码清单 5-26 两个等效门例化的 Verilog HDL 描述(1)

```
module driver (in, out, en);
input [3:0] in;
output [3:0] out;
input en;
bufif0 ar[3:0] (out, in, en);              //三态缓冲区数组
endmodule

module driver_equiv (in, out, en);
input [3:0] in;
output [3:0] out;
input en;
bufif0 ar3 (out[3], in[3], en);            //每个单独的声明
bufif0 ar2 (out[2], in[2], en);
bufif0 ar1 (out[1], in[1], en);
bufif0 ar0 (out[0], in[0], en);
endmodule
```

【例 5.77】 两个等效门例化 Verilog HDL 描述的例子 2 如代码清单 5-27 所示。

代码清单 5-27 两个等效门例化的 Verilog HDL 描述(2)

```
module busdriver (busin, bushigh, buslow, enh, enl);
input [15:0] busin;
```

```
output [7:0] bushigh, buslow;
input enh, enl;
driver busar3 (busin[15:12], bushigh[7:4], enh);
driver busar2 (busin[11:8], bushigh[3:0], enh);
driver busar1 (busin[7:4], buslow[7:4], enl);
driver busar0 (busin[3:0], buslow[3:0], enl);
endmodule

module busdriver_equiv (busin, bushigh, buslow, enh, enl);
input [15:0] busin;
output [7:0] bushigh, buslow;
input enh, enl;
driver busar[3:0] (.out({bushigh, buslow}), .in(busin),.en({enh, enh, enl, enl}));
endmodule
```

5.6.2 逻辑门

逻辑门的关键字包括如下几个：

and nand nor or xor xnor

表 5.23 给出了多个逻辑输入门的真值表。

表 5.23 多个逻辑输入门的真值表

and	0	1	x	z	or	0	1	x	z
0	0	0	0	0	0	0	1	x	x
1	0	1	x	x	1	1	1	1	1
x	0	x	x	x	x	x	1	x	x
z	0	x	x	x	z	x	1	x	x
nand	0	1	x	z	nor	0	1	x	z
0	1	1	1	1	0	1	0	x	x
1	1	0	x	x	1	0	0	0	0
x	1	x	x	x	x	x	0	x	x
z	1	x	x	x	z	x	0	x	x
xor	0	1	x	z	xnor	0	1	x	z
0	0	1	x	x	0	1	0	x	x
1	1	0	x	x	1	0	1	x	x
x	x	x	x	x	x	x	x	x	x
z	x	x	x	x	z	x	x	x	x

门级逻辑设计描述中可使用具体的门例化语句。简单的门实例语句的格式如下：

```
gate_type [instance_name](term1, term2, …,termN);
```

其中，gate_type 为前面所列出门的关键字；instance_name 为例化标识符；term1，term2，…，termN 用于表示与门的输入/输出端口相连的网络是可选的。

延迟说明应该是 0 个、1 个或者 2 个延迟，如果描述中包含两个延迟，则第一个延迟确定输出上升延迟；第二个延迟确定输出下降延迟。两个延迟中较小的一个应用于输出跳变到 x 的延迟。如果只有一个延迟，将应用于上升和下降延迟。如果没有指定则没有传播延迟。

注：这六种类型的逻辑门，有多个输入，但只有一个输出。终端列表的第一个终端将连接到门的输出；其他终端将连接到输入。

【例 5.78】 带有延迟和强度声明逻辑门的 Verilog HDL 描述的例子。

```
specify                                  // specify 关键字定义指定块
  specparam r_delay = 6;                 // specparam 关键字指定参数 r_delay 为 6
  specparam f_delay = 7;                 // specparam 关键字指定参数 f_delay 为 7
```

```
endspecify                                          // endspecify 关键字标识指定块的结束
reg a,b;                                            // reg 关键字定义 reg 类型变量 a 和 b
wire c;                                             // wire 关键字定义 wire 型网络 c
and (strong0,strong1) #(r_delay,f_delay) Inst_and(c,a,b);   // 调用内建 and 门,包含强度和
                                                            // 延迟声明
```

注：逻辑门中的强度声明和延迟声明仅用于行为级仿真中，不能用于 RTL/数据流描述。

思考与练习 5-8 说明两个延迟 r_delay 和 f_delay 对逻辑门上升和下降沿的影响。

5.6.3 输出门

多输出逻辑门的例化，使用下面的关键字：

buf not

其延迟特性同逻辑门的延迟说明。表 5.24 给出了输出门的真值表。

表 5.24 输出门的真值表

buf		not	
输入	输出	输入	输出
0	0	0	1
1	1	1	0
x	x	x	x
z	x	z	x

这些门都只有一个输入、一个/多个输出。输出门实例语句的基本语法如下：

```
multiple_output_gate_type [instance_name] (out1,out2,…,outn,inputA);
```

其中，multiple_output_gate_type 为输出门的关键字；instance_name 为可选的例化标识符；out1，out2，…，outN，inputA 为输出/输入端口。只有最后的端口 inputA 是输入端口，其余的所有端口为输出端口。延迟说明同逻辑门的延迟说明。

【例 5.79】 多输出门 Verilog HDL 描述的例子。

```
buf  b1(out1,out2,in);
```

在该门实例语句中，in 是缓冲门的输入，out1 和 out2 是输出；b1 是例化标识符。

5.6.4 三态门

三态逻辑门的例化，使用下面的关键字：

bufif0 bufif1 notif1 notif0

这些门用于对三态驱动器建模，这些门有一个输出、一个数据输入和一个控制输入。表 5.25 给出了三态逻辑门的真值表。

表 5.25 三态逻辑门的真值表

bufif**0**	CONTROL				bufif**1**	CONTROL			
	0	1	x	z		0	1	x	z
数据	0	z	L	L	数据	z	0	L	L
	1	z	H	H		z	1	H	H
	x	z	x	x		z	x	x	x
	z	z	x	x		z	x	x	x

续表

notif0	CONTROL					notif1	CONTROL				
		0	1	x	z			0	1	x	z
数据	0	1	z	H	H	数据	0	z	1	H	H
	1	0	z	L	L		1	z	0	L	L
	x	x	z	x	x		x	z	x	x	x
	z	x	z	x	x		z	z	x	x	x

注：

(1) 符号 L,表示结果为“0”或“z”；符号 H,表示结果为“1”或“z”。

(2) 跳变到 H 或者 L 的延迟和跳变到 x 的延迟是一样的。

三态门例化语句的基本语法如下：

```
tristate_gate [instance_name](outputA,inputB,control);
```

其中,tristate_gate 为三态门的关键字；instance_name 为可选的例化标识符；outputA 是输出端口；inputB 是数据输入端口；control 是控制输入端口。根据控制输入的值,可以将输出驱动到高阻状态,即值 z。

三态门的延迟说明应该是 0 个、1 个、2 个或者 3 个延迟。

(1) 如果说明中包含 3 个延迟,则第一个延迟确定输出上升延迟,第二个延迟确定输出下降延迟,第三个延迟确定跳变到 x 的延迟。

(2) 如果说明中包含 2 个延迟,则第一个延迟确定输出上升延迟,第二个延迟确定输出下降延迟,两个延迟中较小的一个延迟用于确定跳变到 x 和 z 的延迟。

(3) 如果只有 1 个延迟,将应用到所有的输出跳变延迟。

(4) 如果没有指定延迟,则门没有传播延迟。

【例 5.80】 带有延迟和强度声明三态门的 Verilog HDL 描述的例子。

```
specify                                          // specify 关键字定义指定块
  specparam r_delay = 6;                         // specparam 关键字指定参数 r_delay 为 6
  specparam f_delay = 7;                         // specparam 关键字指定参数 f_delay 为 7
  specparam x_delay = 8;                         // specparam 关键字指定参数 f_delay 为 7
 endspecify                                      // endspecify 关键字标识指定块的结束
reg in,control;                                  // reg 关键字定义 reg 类型变量 in 和 control
wire out;                                        // wire 关键字定义 wire 类型网络 out
bufif0 #(r_delay,f_delay,x_delay) Inst_bufif0(out,in,control); // 调用/例化内建三态逻辑门 buifo
```

注：逻辑门中的延迟声明仅用于行为级仿真中,不能用于 RTL/数据流描述。

思考与练习 5-9 说明三个延迟 r_delay、f_delay 和 x_delay 对三态逻辑门上升和下降沿的影响。

5.6.5 上拉和下拉源

例化上拉和下拉源使用下面关键字：

pullup　　pulldown

上拉源 pullup 将终端列表中的网络置为 1,下拉源 pulldown 将终端列表中的网络置为 0。在没有强度说明的情况下,放置在网络上的这些信号源将为 pull 强度。如果在 pullup 上有 strength1 说明或者在 pulldown 上有 strength0 说明,信号应该有强度说明,将忽略在 pullup 上的 strength0 说明或者在 pulldown 上的 strength1 说明。

这些源没有延迟说明。

pull门没有输入只有输出,门实例的端口表只包含1个输出。

【例5.81】 pullup门Verilog HDL描述的例子。

```
pullup (strong1) p1 (neta), p2 (netb);
```

在该例化语句中,p1例化驱动neta,p2例化驱动netb,并且为strong1强度。

5.7 Verilog HDL行为建模语句

截止到本书目前所介绍的内容,引入的语言结构允许在相对详细的层次上描述硬件。用逻辑门和连续分配/赋值对电路建模,可以很好地反映所建模电路的逻辑结构;然而,这些结构没有提供描述系统的复杂高级方面所必需的抽象能力。本节介绍的程序结构更适合解决类似微处理器或实现复杂的时序检查等问题。


视频讲解

5.7.1 行为模型概述

Verilog行为模型包含控制仿真和操作之前描述数据类型变量的过程语句。这些语句包含在过程中。每个过程都有与其相关的活动流程。

活动开始于控制结构initial和always。每个initial结构和每个always结构启动一个单独的活动流。所有活动流是并发的,以建模硬件固有的并发性。

一个完整的Verilog行为模型如代码清单5-28所示。

代码清单5-28 一个完整行为模型的Verilog HDL描述

```
module behave;              // module关键字定义模块behave
reg [1:0] a, b;             // reg关键字定义reg类型变量a和b,宽度为2位,范围为[1:0]
initial                     // initial关键字定义初始化部分
begin                       // begin关键字标识初始化部分的开始,类似C语言的"{"
  a = 'b1;                  // 给变量a赋值/分配"1"
  b = 'b0;                  // 给变量b赋值/分配"0"
end                         // end关键字标识初始化部分的结束,类似C语言的"}"
always                      // always关键字标识过程语句
begin                       // begin关键字标识过程语句的开始,类似C语言的"{"
  #50 a = ~a;               // 延迟50ns后,对变量a取反后,赋值/分配值给a
end                         // end关键字标识过程语句的结束,类似C语言的"}"
always                      // always关键字标识过程语句
begin                       // begin关键字标识过程语句的开始,类似C语言的"{"
  #100 b = ~b;              // 延迟100ns后,对变量b取反后,赋值/分配值给b
end                         // end关键字标识过程语句的结束,类似C语言的"}"
endmodule                   // endmodule关键字标识模块behave的结束
```

在对该行为级模型仿真时,由initial和always结构所定义的所有流的仿真时间为零时刻一起开始。Initial结构执行一次,而always结构重复执行。

在该模型中,reg变量a和b在仿真时间为零时刻分别初始化为“1”和“0”。然后,initial结构完成,在仿真运行期间不会再次执行。该initial结构包含了begin-end块(也称为顺序块)。在这个begin-end块中,首先初始化a,紧跟着初始化b。

always结构也是从零时刻开始,但变量的值不发生变化,直到到达由延迟控制(由“#”标识)指定的时间为止。因此,reg类型变量a的值在50个时间单位后翻转,reg类型变量b的值在100个时间单位后翻转。因为always结构重复,该模型将产生两个方波信号。reg类型变量a以100个时间单位的周期切换,reg类型变量b以200个时间单位的周期切换。在整个仿真运行过程中,这两个always结构始终在同时进行。

5.7.2 过程语句

过程分配用于更新 reg、integer、time、real、realtime 和存储器数据类型。对于过程分配和连续分配来说,有以下不同之处。

1) 连续分配

连续分配驱动网络。只要一个输入操作数的值发生变化,则更新和求取所驱动网络的值。

2) 过程分配

在过程流结构的控制下,过程分配更新流结构内变量的值。

过程分配的右边可以是任何求取值的表达式,左侧应该是一个变量,它用于接收右侧表达式所引用分配的值。过程分配的左侧可以是下面的一种格式:

(1) reg、integer、real、realtime 或者 time 数据类型,表示分配给这些数据类型的名字。

(2) reg、integer、real、realtime 或者 time 数据类型的位选择,表示分配到单个的比特位。

(3) reg、integer、real、realtime 或者 time 数据类型的部分选择,表示一个或者多个连续比特位的部分选择。

(4) 存储器字,表示一个存储器的单个字。

(5) 任何上面四种形式的并置/连接或者嵌套的并置/连接,这些语句对右侧的表达式进行有效的分隔,将分隔的部分按顺序分配到并置或者嵌套并置的不同部分中。

Verilog HDL 包含两种类型的过程分配/赋值语句。

(1) 阻塞过程分配语句。

(2) 非阻塞过程分配语句。

1. 阻塞过程分配

阻塞过程分配语句应在执行顺序块中紧跟随其后的语句之前执行。阻塞过程分配语句不能阻止并行块中跟随其后的语句的执行。

阻塞过程分配/赋值所使用的等号“=”赋值运算符也用于过程连续分配和连续分配中。

【例 5.82】 阻塞过程分配 Verilog HDL 描述的例子。

```
rega = 0;
rega[3] = 1;                          //位选择
rega[3:5] = 7;                        //部分选择
mema[address] = 8'hff;                //分配到一个存储器元素
{carry, acc} = rega + regb;           //并置
```

2. 非阻塞过程分配

非阻塞过程分配允许分配调度,但不会阻塞过程内的流程。在相同的时间段内,当有多个变量分配时,使用非阻塞过程分配。该分配不需要考虑顺序或者互相之间的依赖性。

非阻塞赋值操作符与小于或等于关系操作符相同,都使用符号“<=”。读者应根据该符号出现的上下文来理解该符号的具体含义。在表达式中使用“<=”时,应将其解释为关系操作符;在非阻塞过程分配中使用“<=”时,应将其解释为分配/赋值运算符。

【例 5.83】 阻塞和非阻塞过程分配 Verilog HDL 描述的例子 1,如代码清单 5-29 所示。

代码清单 5-29 阻塞和非阻塞分配的 Verilog HDL 描述(1)

```
module evaluates2 (out);
output out;
reg a, b, c;
initial
begin
```

```
  a = 0;                                    在 0 时刻,初始化后 a = 0,b = 1
  b = 1;
  c = 0;
end
always c = #5 ~c;
always @(posedge c)
begin
  a <= b;                                   在 5 个时间单位后,a = 1,b = 0
  b <= a;
end
endmodule
```

在该例子中,通过两步评估非阻塞过程分配。第一步,在 c 的上升沿(posedge c),仿真器评估非阻塞分配/赋值的右侧,并在非阻塞赋值更新事件结束时安排新的赋值;第二步,当仿真工具激活非阻塞分配更新事件时,仿真器更新每个非阻塞分配语句的左侧。

时间步骤的结束意味着非阻塞分配是时间步骤中执行的最后一个赋值,只有一个例外。非阻塞分配事件可以创建阻塞分配事件。这些阻塞分配事件应该在安排的非阻塞事件后面处理。

【例 5.84】 阻塞和非阻塞过程分配 Verilog HDL 描述的例子 2,如代码清单 5-30 所示。

代码清单 5-30 阻塞和非阻塞分配的 Verilog HDL 描述(2)

```
module non_block1;
reg a, b, c, d, e, f;
//阻塞分配
initial begin
    a = #10 1;                              //在第 10 个时间单位时,给 a 分配 1
    b = #2 0;                               //在第 2 个时间单位时,给 b 分配 0
    c = #4 1;                               //在第 4 个时间单位时,给 c 分配 1
end
//非阻塞分配
initial begin
    d <= #10 1;                             //在第 10 个时间单位时,给 d 分配 1
    e <= #2 0;                              //在第 2 个时间单位时,给 e 分配 0
    f <= #4 1;                              //在第 4 个时间单位时,给 f 分配 1
end
endmodule
```

在第 2 个时间单位,调度变化 e=0;在第 4 个时间单位,调度变化 f=1;在第 10 个时间单位,调度变化 d=10。

不像用于阻塞分配/赋值的一个事件或延迟控制,非阻塞分配不会阻塞过程流。非阻塞赋值评估并调度赋值,但是它不会阻止在一个 begin-end 块中后续语句的执行。

【例 5.85】 阻塞和非阻塞过程分配 Verilog HDL 描述的例子 3,如代码清单 5-31 所示。

代码清单 5-31 阻塞和非阻塞分配的 Verilog HDL 描述(3)

```
module non_block1;
reg a, b;
initial begin
    a = 0;                                  a = 1,b = 0
    b = 1;
    a <= b;
    b <= a;
end
initial begin
     $monitor ( $time, ,"a = %b b = %b", a, b);
     #100 $finish;
     end
endmodule
```

在该例子中,第一步,仿真器评估非阻塞分配的右侧,并且在当前时间步骤结束时安排分配;第二步,在当前时间步骤结束时,仿真器更新每个非阻塞分配语句的左侧。

正如例子所示,仿真器评估和调度当前时间步骤结束时的分配,并可以使用非阻塞过程分配执行交换操作。

【例 5.86】 阻塞和非阻塞过程分配 Verilog HDL 描述的例子 4,如代码清单 5-32 所示。

代码清单 5-32 阻塞和非阻塞分配的 Verilog HDL 描述(4)

```
module multiple;
reg a;
initial a = 1;
initial begin
    a <= #4 0;                          //在第 4 个时间单位,调度 a = 0
    a <= #4 1;                          //在第 4 个时间单位,调度 a = 1,结果 a = 1
end
endmodule
```

应保留对给定变量执行不同非阻塞赋值的顺序。换句话说,如果一组非阻塞分配的执行顺序明确,则非阻塞分配目标的结果更新顺序应该与执行顺序相同。

【例 5.87】 阻塞和非阻塞过程分配 Verilog HDL 描述的例子 5,如代码清单 5-33 所示。

代码清单 5-33 阻塞和非阻塞分配的 Verilog HDL 描述(5)

```
module multiple;
reg a;
initial a = 1;
initial begin
    a <= #4 0;                          //在第 4 个时间单位,调度 a = 0
    a <= #5 1;                          //在第 5 个时间单位,调度 a = 1
end
endmodule
```

【例 5.88】 阻塞和非阻塞过程分配 Verilog HDL 描述的例子 6,如代码清单 5-34 所示。

代码清单 5-34 阻塞和非阻塞分配的 Verilog HDL 描述(6)

```
module multiple2;
reg a;
initial a = 1;
initial a <= #4 0;                      //在第 4 个时间单位,调度 a = 0
initial a <= #4 1;                      //在第 4 个时间单位,调度 a = 1
//在第 4 个时间单位, a = ?
//寄存器类型数据 a 分配的值是不确定的
endmodule
```

如果仿真器同时执行两个过程块,并且过程块包含对相同变量的非阻塞分配操作符,则变量最终的值是不确定的。

【例 5.89】 阻塞和非阻塞过程分配 Verilog HDL 描述的例子 7,如代码清单 5-35 所示。

代码清单 5-35 阻塞和非阻塞分配的 Verilog HDL 描述(7)

```
module multiple3;
reg a;
initial #8 a <= #8 1;                  //在第 8 个时间单位执行,在第 16 个时刻更新为 1
initial #12 a <= #4 0;                 //在第 12 个时间单位执行,在第 16 个时刻更新为 0
endmodule
```

从该例子可知,针对同一变量的两个非阻塞赋值位于不同块中这一事实本身并不足以使对变量的赋值顺序不确定。因为确定将 a 更新为“1”是在更新 a 为“0”之前安排的,所以在时隙 16 时 a 的值为“0”。

【例 5.90】 阻塞和非阻塞过程分配 Verilog HDL 描述的例子 8,如代码清单 5-36 所示。

代码清单 5-36　阻塞和非阻塞分配的 Verilog HDL 描述(8)

```
module multiple4;
reg r1;
reg [2:0] i;
initial begin
for (i = 0; i <= 5; i = i + 1)
    r1 <= # (i * 10) i[0];
end
endmodule
```

该例子说明如何将 i[0]的值分配给 r1,以及如何在每个时间延迟后调度分配。图 5.8 给出了仿真结果的波形图。

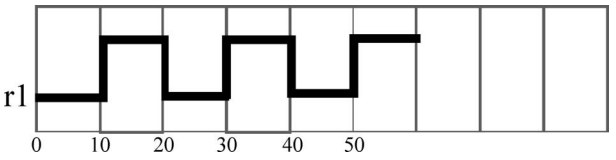

图 5.8　仿真波形图

5.7.3　过程连续分配

使用关键字 assign 和 force 的过程连续分配是过程语句。在变量或者网络上,允许连续驱动表达式。高云云源软件综合工具不支持过程连续分配。

assign 语句中赋值/分配的左侧应该是变量引用或变量并置/连接,它不能是存储器字(数组引用)或变量的比特位选择或者部分选择。

相反,force 语句中赋值/分配的左侧应该是变量引用或网络引用,它也可以是变量或网络的并置,它不允许向量变量的比特位选择或者部分选择。

1. assign 和 deassign 过程语句

assign 过程连续分配语句将覆盖对变量的所有过程分配。deassign 过程分配将终止对一个变量的过程连续分配。变量将保持相同的值,直到通过一个过程分配或者一个过程连续分配语句,给变量分配一个新的值为止。

如果关键字 assign 应用于已存在过程连接赋值的变量,则在新的过程连续分配/赋值之前,新的过程连续分配应该取消分配(deassign)变量。

2. force 和 release 过程语句

force 和 release 过程语句提供了另一种形式的过程连续分配,这些语句的功能和 assign/deassign 类似,但是 force 可以用于网络和变量。左侧的分配可以是变量、网络、向量网络的常数位选择和部分选择、并置,它不能是存储器字(数组引用)或者向量变量的位选择和部分选择。

对一个变量的 force 操作将覆盖对变量的一个过程分配或者一个分配过程的连续分配,直到对该变量使用了 release 过程语句为止。当 release 时,如果当前变量没有一个活动的分配过程连续分配,不会立即改变变量的值,变量将保留当前的值,直到对该变量的下一个过程分配或者过程连续分配为止。release 一个当前有活动分配过程的连续分配的变量时,将立即重新建立那个分配。

对一个网络的 force 过程语句,将覆盖网络所有的驱动器,包括门输出、模块输出和连续分配,直到在该网络上执行一个 release 语句为止。当 release 时,网络将立即分配由网络驱动器所分配的值。

【例 5.91】 过程连续分配 Verilog HDL 描述的例子，如代码清单 5-37 所示。

代码清单 5-37 过程连续分配的 Verilog HDL 描述

```
module test;
reg a, b, c, d;
wire e;
and and1 (e, a, b, c);
initial begin
     $monitor("%d d=%b,e=%b", $stime, d, e);
    assign d = a & b & c;
    a = 1;
    b = 0;
    c = 1;
    #10;
    force d = (a | b | c);
    force e = (a | b | c);
    #10;
    release d;
    release e;
    #10 $finish;
end
endmodule
```

最终的结果如下：

```
0   d=0,e=0
10  d=1,e=1
20  d=0,e=0
```

5.7.4 条件语句

条件语句(或 if-else 语句)用于确定是否执行一条语句。

如果表达式评估为真(即具有非零的已知值)，则应执行第一条语句。如果表达式的评估为假(即具有零值或值为“x”或“z”)，则不应该执行第一条语句。如果存在 else 语句且表达式为假，则应该执行 else 语句。

因为 if 表达式的数值被测试为零，所以某些快捷方式是可能的。例如，以下两条语句表达了相同的逻辑：

```
if(expression)
if(expression!=0)
```

因为 if-else 的 else 部分是可选的，所以当嵌套的 if 序列中省略 else 时，可能会产生混淆。如果缺少 else，则总是通过最近的关联 else 来解决该问题。在下面的例子中，else 与内部 if 一起使用，如缩进所示。

```
if (index > 0)
      if (rega > regb)
         result = rega;
      else                              // else适用于前面的 if
         result = regb;
```

如果不需要这种关联，则应使用 begin-end 块语句强制进行正确的关联，如下所示。

```
if (index > 0) begin
        if (rega > regb)
            result = rega;
end
else result = regb;
```

下面将详细讨论 if-else-if 结构，其完整的语法格式如下：

```
if(expression)
    statement_or_null;
else if(expression)
    statement_or_null;
else
    statement_or_null;
```

其中，expression 为不同的条件表达式，statement_or_null 为该条件表达式下的具体语句，当该条件表达式下有多条语句时，需要将这些语句包含在 begin-end 关键字中间。

这个 if 语句序列(称为 if-else-if 结构)是编写多路决策的最通用方法。表达式应该按顺序进行评估。如果任何表达式为真，则应执行与其关联的语句，这将终止整个链。每条语句要么是单个语句，要么是一组语句。

if-else-if 结构的最后一个 else 部分，处理上述任何一个条件都不满足的情况或默认情况。有时对默认情况没有明确的行为，在这种情况下，可以省略后面的 else 语句，也可以将其用于错误检查以捕获不可能的条件。

【例 5.92】 if-else-if 语句的 Verilog HDL 描述的例子，如代码清单 5-38 所示。

代码清单 5-38　if-else-if 语句的 Verilog HDL 描述

```
//声明寄存器和参数
reg [31:0] instruction, segment_area[255:0];
reg [7:0] index;
reg [5:0] modify_seg1,
          modify_seg2,
          modify_seg3;
parameter
          segment1 = 0, inc_seg1 = 1,
          segment2 = 20, inc_seg2 = 2,
          segment3 = 64, inc_seg3 = 4,
          data = 128;
//测试索引变量
if (index < segment2) begin
    instruction = segment_area [index + modify_seg1];
    index = index + inc_seg1;
end
else if (index < segment3) begin
    instruction = segment_area [index + modify_seg2];
    index = index + inc_seg2;
end
else if (index < data) begin
    instruction = segment_area [index + modify_seg3];
    index = index + inc_seg3;
end
else
    instruction = segment_area [index];
```

在该代码片段中，使用 if-else 语句测试变量索引，以确定是否必须将 reg 类型变量 modify_seg1、modify_seg2 和 modify_seg3 中的一个添加到存储器地址，以及将哪个增量添加到 reg 类型变量 index。在该代码片段中的前 10 行代码声明 reg 类型变量和参数。


视频讲解

5.7.5　case 语句

case 语句是一个多条件分支语句，用于测试一个表达式是否匹配相应的其他表达式和分支。语法如下：

```
case/casez/casex(case - expr)
  case_item_expr_1:  procedural_statement_1;
```

```
    case_item_expr_2:  procedural_statement_2;
          …
    case_item_expr_n:  procedural_statement_n;
    default:           procedural_statement_n + 1;
endcase
```

其中,case_expr 为条件表达式; case_item_expr_1,…,case_item_expr_n 为条件值; procedual_statement_1,…,procedual_statement_n+1 为描述语句。

【例 5.93】 case 语句 Verilog HDL 描述的例子 1,如代码清单 5-39 所示。

代码清单 5-39 case 语句的 Verilog HDL 描述(1)

```
reg[15:0] rega;
reg [9:0] result;
case (rega)
    16'd0: result = 10'b0111111111;
    16'd1: result = 10'b1011111111;
    16'd2: result = 10'b1101111111;
    16'd3: result = 10'b1110111111;
    16'd4: result = 10'b1111011111;
    16'd5: result = 10'b1111101111;
    16'd6: result = 10'b1111110111;
    16'd7: result = 10'b1111111011;
    16'd8: result = 10'b1111111101;
    16'd9: result = 10'b1111111110;
    default: result = 'bx;
endcase
```

括号中给出的 case 表达式应该在任何 case 条目表达式 case_item_expr_n 之前精确计算一次。case 条目表达式应该按照给出的确切顺序进行比较和评估。如果有 default(默认)的 case 条目,则在线性搜索期间忽略它。在线性搜索期间,如果 case 条目表达式中的一个条目匹配括号中给出的 case 表达式,则应执行与该 case 条目相关的语句,并且终止线性搜索。如果所有比较失败,并且给出了 default 条目,则应执行 default 条目语句,如果未给出 default 条目语句,并且所有的比较都失败了,则不应该执行任何 case 条目语句。

除了语法之外,case 语句在两个重要方面与多路 if-else-if 结构不同。

(1) if-else-if 结构中的条件表达式比比较一个表达式更通用。

(2) 当表达式中有“x”和“z”值时,case 语句提供了确定的结果。

在 case 表达式比较中,只有当每位与值“0”、“1”、“x”和“z”完全匹配时,比较才会成功。因此,在 case 语句中指定表达式要谨慎。所有表达式的位长度应该相等,以便执行精确的位匹配。所有 case 条目表达式以及括号中 case 表达式的长度应该等于最长 case 表达式和 case 条目表达式的长度。如果这些表达式中的任何一个是无符号的,则所有这些表达式都应看作无符号的。如果所有这些表达式都是有符号的,则应将它们都看作有符号的。

提供处理“x”和“z”值的 case 表达式比较的原因:它提供了一种检测此类值并减少其存在可能产生的悲观情绪的机制。

【例 5.94】 case 语句 Verilog HDL 描述的例子 2,如代码清单 5-40 所示。

代码清单 5-40 case 语句的 Verilog HDL 描述(2)

```
case(select[1:2])
    2'b00: result = 0;
    2'b01: result = flaga;
    2'b0x,
    2'b0z: result = flaga ? 'bx : 0;
    2'b10: result = flagb;
    2'bx0,
```

```
    2'bz0: result = flagb ? 'bx : 0;
    default: result = 'bx;
endcase
```

该例子中，当 select[1]＝“0”并且 flaga＝“0”时，即使 select[2]＝“x”/“z”，结果也是“0”。

【例 5.95】 case 语句中“x”和“z”条件 Verilog HDL 描述的例子，如代码清单 5-41 所示。

代码清单 5-41　case 语句中处理“x”和“z”条件的 Verilog HDL 描述(3)

```
case(sig)
  1'bz: $display("signal is floating");
  1'bx: $display("signal is unknown");
  default: $display("signal is %b", sig);
endcase
```

1. 包含无关的 case 语句

提供其他两种类型的 case 语句，以允许在 case 比较中处理无关条件。其中一个将高阻(“z”)看作无关，另一个则将高阻和不确定(“x”)看作无关。

这些 case 语句可以用与传统 case 语句相同的方式使用，但它们分别以关键字 casez 和 casex 开头。

在比较期间，case 表达式或 case 条目的任何位中的无关值(casez 的“z”值，casex 的“z”和“x”值)应看作无关值，且不考虑该位的位置。case 表达式中的无关条件可用于动态控制应在任何时候比较哪些位。

字面数字的语法允许在这些 case 语句中使用问号“?”代替“z”。这为 case 语句中的无关位的规范提供了一种方便的格式。

【例 5.96】 casez 语句 Verilog HDL 描述的例子，如代码清单 5-42 所示。

代码清单 5-42　casez 语句的 Verilog HDL 描述

```
reg[7:0] ir;
casez (ir)
8'b1???????: instruction1(ir);
8'b01??????: instruction2(ir);
8'b00010???: instruction3(ir);
8'b000001??: instruction4(ir);
endcase
```

该例子给出了 casez 语句的用法。它展示了一个指令译码，其中最高有效位的值选择应该调用哪个任务。如果 ir 的最高位为“1”，则调用任务 instruction1，而不考虑 ir 其他位的值。

【例 5.97】 casex 语句 Verilog HDL 描述的例子，如代码清单 5-43 所示。

代码清单 5-43　casex 语句的 Verilog HDL 描述

```
reg[7:0] r, mask;
mask = 8'bx0x0x0x0;
casex (r ^ mask)
    8'b001100xx: stat1;
    8'b1100xx00: stat2;
    8'b00xx0011: stat3;
    8'bxx010100: stat4;
endcase
```

在该例子中，给出了 casex 语句的用法。它给出了在仿真过程中如何动态控制无关条件的极端情况。在这种情况下，如果 r＝8'b01100110，则调用任务 stat2。

2. case 语句中的常数表达式

常数表达式也可以作为 case 表达式。常数表达式的值将要和 case 条目中的表达式进行比较，以寻找匹配的条件。

【例 5.98】 case 中常数表达式 Verilog HDL 描述的例子，如代码清单 5-44 所示。

代码清单 5-44 case 中常数表达式的 Verilog HDL 描述

```
reg[2:0] encode ;
case (1)
    encode[2] : $display("Select Line 2") ;
    encode[1] : $display("Select Line 1") ;
    encode[0] : $display("Select Line 0") ;
    default:   $display("Error: One of the bits expected ON");
endcase
```

在该例子中，case 表达式是常数表达式(1)。case 条目是表达式(位选择)，并与常数表达式进行比较以进行匹配。

5.7.6 循环语句

有 4 种类型的循环语句，这些语句提供了一种控制语句执行 0 次、一次或多次的方法。

(1) forever。连续执行语句。高云云源软件综合工具不支持该循环。

(2) repeat。将语句执行固定的次数。如果表达式评估为未知或高阻，则应将其看作 0，不应执行任何语句。

(3) while。执行语句，直到表达式变为 false(假)。如果表达式以 false(假)开始，则语句根本不会执行。

(4) for。通过三步流程控制相关语句的执行。

① 执行分配，通常用于初始化控制执行循环次数的变量。

② 评估表达式。如果结果为 0，for 循环将退出。如果不为 0，for 循环应执行其相关的语句，然后执行步骤③。如果表达式评估为不确定或高阻值，则应将其看作 0；

③ 执行分配，通常用于修改循环控制变量的值，然后重复步骤②。

【例 5.99】 repeat 循环 Verilog HDL 描述的例子，如代码清单 5-45 所示。

代码清单 5-45 repeat 循环的 Verilog HDL 描述

```
parameter size = 8, longsize = 16;
reg [size:1] opa, opb;
reg [longsize:1] result;
begin : mult
    reg [longsize:1] shift_opa, shift_opb;
    shift_opa = opa;
    shift_opb = opb;
    result = 0;
    repeat (size) begin
        if (shift_opb[1])
            result = result + shift_opa;
        shift_opa = shift_opa << 1;
        shift_opb = shift_opb >> 1;
    end
end
```

在该例子中，repeat 循环通过加法和移位操作符实现乘法器。

【例 5.100】 while 循环 Verilog HDL 描述的例子，如代码清单 5-46 所示。

代码清单 5-46 while 循环的 Verilog HDL 描述

```
begin : count1s
  reg [7:0] tempreg;
  count = 0;
  tempreg = rega;
while (tempreg) begin
```

```
  if (tempreg[0])
    count = count + 1;
  tempreg = tempreg >> 1;
  end
end
```

在该例子中,计算 rega 中逻辑“1”的个数。

【例 5.101】 while 循环和 for 循环等效的 Verilog HDL 描述的例子。

```
begin
    initial_assignment;          // 初始化分配/赋值语句
while (condition) begin
      statement                  // 语句
      step_assignment;           // 步长赋值和分配
    end
end
```

for 循环只使用两行代码实现上面的逻辑,伪代码如下所示。

```
for(initial_assignment; condition; step_assignment)
                statement
```

5.7.7 过程时序控制

Verilog HDL 有两种类型的显式时序控制,用于控制过程语句何时发生。第一种类型是延迟控制,其中表达式指定从最初遇到语句到实际执行语句之间的时间间隔。延迟表达式可以是电路状态的动态函数,但它可以是一个简单的数字,可以在时间上分隔语句的执行。延迟控制是指定激励波形描述时的一个重要特征。

第二种类型的时序控制是事件表达式,它允许延迟语句的执行,直到在与此过程同时执行的过程中发生某些仿真事件。仿真事件可以是网络或变量值的变化(隐含事件),也可以是由其他过程触发的显式命名事件的发生(显式事件)。最常见的情况,事件控制是时钟信号的上升沿(posedge)或下降沿(negedge)。

到目前为止遇到的过程语句,都是在未使仿真时间前进的情况下执行。仿真时间可以通过以下 3 种方法前进。

(1) 延迟控制,由符号“#”引入。

(2) 事件控制,由符号“@”引入。

(3) wait 语句,其操作方式类似事件控制和 while 循环的组合。

1. 延迟控制

跟随延迟控制的过程语句应该在执行延迟,延迟时间为相对于过程语句之前的延迟控制指定的延迟。如果延迟表达式评估为不确定或高阻值,则应该理解为零延迟。如果延迟表达式评估为负值,则应将其理解为与时间变量位宽相同的二进制补码的无符号整数。延迟表达式允许指定参数,它们可以被 SDF 注解覆盖,在这种情况下,重新评估表达式。

【例 5.102】 延迟控制 Verilog HDL 描述例子 1。

```
#10 rega = regb;
```

该例子中,延迟分配/赋值 10 个时间单位。

【例 5.103】 延迟控制 Verilog HDL 描述例子 2。

```
#d rega = regb;                          //d 定义为一个参数
#((d+e)/2) rega = regb;                  //延迟是 d 和 e 的平均值
#regr regr = regr + 1;                   //延迟在 regr 寄存器中
```

在该例子中的3个分配提供了符号“#”后面的表达式。分配/赋值的执行延迟了表达式指定的仿真时间量。

2. 事件控制

通过一个网络、变量或者一个声明事件的发生,来同步一个过程语句的执行。网络和变量值的变化可以作为一个事件,用于触发语句的执行,这就称为检测一个隐含事件。事件基于变化的方向,即朝着值“1”(posedge)或者朝着值“0”(negedge):

(1) negedge:①检测到从“1”跳变到“x”、“z”或“0”;②检测到从“x”或“z”跳变到“0”。

(2) posedge:①检测到从“0”跳变到“x”、“z”或“1”;②检测到从“x”或“z”跳变到“1”。

【例5.104】 边沿控制语句Verilog HDL描述的例子。

```
@r rega = regb;                                  //由寄存器r内值的变化控制
@(posedge clock) rega = regb;                    //时钟上升沿控制
forever @(negedge clock) rega = regb;            //时钟下降沿控制
```

3. 命名事件

除了网络和变量外,可以声明一种新的数据类型——事件。一个用于声明事件数据类型的标识符称为一个命名的事件。它可以用在事件表达式内,用于控制过程语句的执行。命名的事件可以来自一个过程。这样,运行控制其他过程中的多个行为。

事件不应包含任何数据,命名事件的特征:它可以在任何特定的时间发生;它没有持续时间;它的出现可以通过使用前面描述的事件控制语法来识别。

通过激活具有规则给出的语法的事件触发语句,使声明的事件发生。更改事件控制表达式中事件数组的索引不会使事件发生。事件控制语句(例如,@trig rega=regb;)应使其包含过程的仿真等待,直到其他过程执行适当的事件触发语句(例如,->trig)。命名事件和事件控制提供了一种强大而高效的方法来描述两个或多个并发活动进程之间的通信和同步。如一个小波形时钟发生器,它通过在电路等待事件发生时周期性地发信号通知显式事件的发生来同步同步电路的控制。

4. 事件或者操作符

可以表示任何数目事件的逻辑“或”,这样任何一个事件的发生将触发跟随在该事件后的过程语句。关键字or或字符“,”,用于事件逻辑“或”操作符,它们的组合可以用于事件表达式中。逗号分隔的敏感列表和or分隔的敏感列表是同步的。

【例5.105】 两个或三个事件逻辑或关系Verilog HDL描述的例子。

```
@(trig or enable) rega = regb;                   //由trig或者enable控制
@(posedge clk_a or posedge clk_b or trig) rega = regb;
```

【例5.106】 使用逗号作为事件逻辑“或”操作符Verilog HDL描述的例子。

```
always @(a, b, c, d, e)
always @(posedge clk, negedge rstn)
always @(a or b, c, d or e)
```

5. 隐含的表达式列表

在RTL级仿真中,一个事件控制的事件表达式列表是一个公共的漏洞。在时序控制描述中,设计者经常忘记添加需要读取的一些网络或者变量。当比较RTL和门级版本的设计时,经常可以发现这个问题。隐含的表达式@*是一个简单的方法,用于解决忘记添加网络或者变量的问题。

语句中出现的所有网络和变量标识符将自动添加到事件表达式中,但下面例外。

(1) 仅出现在wait或事件表达式中的标识符。

(2) 仅在分配/赋值左侧的变量中显示为层次化变量标识符。

在赋值/分配的右侧、函数和任务调用中、在 case 和条件表达式中、在赋值/分配左侧的索引变量或在 case 条目表达式中,作为变量出现的网络和变量均应包含在这些规则中。

【例 5.107】 隐含事件 Verilog HDL 描述的例子 1。

```
always @( * )                                    //等效于@(a or b or c or d or f)
      y = (a & b) | (c & d) | myfunction(f);
```

【例 5.108】 隐含事件 Verilog HDL 描述的例子 2。

```
always @ * begin                                 //等效于@(a or b or c or d or tmp1 or tmp2)
    tmp1 = a & b;
    tmp2 = c & d;
    y = tmp1 | tmp2;
end
```

【例 5.109】 隐含事件 Verilog HDL 描述的例子 3。

```
always @ * begin                                 //等效于@(b)
    @(i) kid = b;                                //i 没有添加到@ *
end
```

【例 5.110】 隐含事件 Verilog HDL 描述的例子 4。

```
always @ * begin                                 //等效于@(a or b or c or d)
      x = a ^ b;
      @ *                                        //等效于@(c or d)
          x = c ^ d;
end
```

【例 5.111】 隐含事件 Verilog HDL 描述的例子 5。

```
always @ * begin                                 //和 @(a or en)一样
    y = 8'hff;
    y[a] = !en;
end
```

【例 5.112】 隐含事件 Verilog HDL 描述的例子 6。

```
always @ * begin                                 //等效于 @(state or go or ws)
    next = 4'b0;
case (1'b1)
    state[IDLE]: if (go) next[READ] = 1'b1;
                 else next[IDLE] = 1'b1;
    state[READ]: next[DLY ] = 1'b1;
    state[DLY ]: if (!ws) next[DONE] = 1'b1;
                else next[READ] = 1'b1;
    state[DONE]: next[IDLE] = 1'b1;
endcase
end
```

6. 电平敏感的事件控制

可以延迟一个过程语句的执行,直到一个条件变成真。使用 wait 语句可以实现这个延迟控制,它是一种特殊形式的事件控制语句。wait 语句的本质对电平敏感,这与基本的事件控制对边沿敏感不同。

wait 语句对条件进行评估。当条件为假时,在等待语句后面的过程语句将保持阻塞状态,直到条件变为真为止。

【例 5.113】 等待事件 Verilog HDL 描述的例子。

```
begin
wait (!enable) #10 a = b;
```

```
    #10 c = d;
end
```

7. 分配内的时序控制

前面介绍的延迟和事件控制结构,是在语句和延迟执行的前面。相比较而言,在一个分配语句中包含分配内延迟和事件控制,以不同的方式修改活动流。

分配内延迟或事件控制应延迟将新值分配到左侧,但应在延迟之前而不是延迟之后评估右侧表达式。

分配内的时序控制可以用于阻塞分配和非阻塞分配。repeat 事件控制说明在一个事件发生了指定数目后的内部分配延迟。如果有符号的 reg 所保存的重复次数小于或等于 0,则将产生分配,这就好像不存在重复结构。

表 5.26 给出了分配内时序控制的等效性比较。

表 5.26 分配内时序控制的等效性比较

带有分配内时序控制结构	没有分配内时序控制结构

a = #5 b;	begin
	temp = b;
	#5 a = temp;
	end

a = @(posedge clk) b;	begin
	temp = b;
	@(posedge clk) a = temp;
	end

a =repeat(3)
@(posedge clk) b;
	begin
	temp = b;
	@(posedge clk);
	@(posedge clk);
	@(posedge clk) a = temp;
	end

下面的三个例子使用 fork-join 行为结构。在关键字 fork 和 join 之间的所有语句同时执行。

下面的例子显示了可以通过使用分配内时序控制来防止竞争条件。

【例 5.114】 fork-join 结构 Verilog HDL 描述的例子 1。

```
fork
      #5 a = b;
      #5 b = a;
join
```

上面的例子在同一仿真时间采样并设置 a 和 b 的值,从而创建了竞争条件。

下面的例子中,使用时序控制内部分配形式防止竞争条件。

```
fork                                          //数据交换
    a = #5 b;
    b = #5 a;
join
```

分配内时序控制起作用,因为分配/赋值内延迟导致在延迟之前评估 a 和 b 的值,并导致在延迟后进行分配/赋值,所以,一些实现分配内时序控制的现有工具在评估右侧的每个表达式时使用临时存储。

分配内等待事件也是有效的。在下面的例子中,每当遇到分配/赋值语句时,会计算右侧

的表达式,但赋值会延迟到时钟信号的上升沿。

【例 5.115】 fork-join 结构 Verilog HDL 描述的例子 2。

```
fork                                                    //数据移位
    a = @(posedge clk) b;
    b = @(posedge clk) c;
join
```

下面的例子给出非阻塞分配/赋值的分配内延迟的重复事件控制。

【例 5.116】 分配内时序控制 Verilog HDL 描述的例子 1。

```
a <= repeat(5) @(posedge clk) data;
```

图 5.9 给出了在 5 个时钟上升沿(posedge clk)后给 a 分配/赋值。

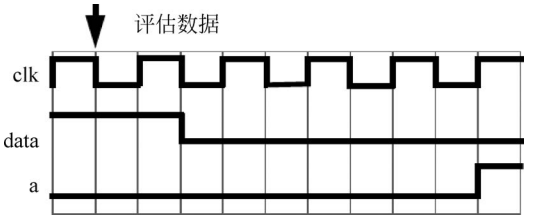

图 5.9 仿真波形图

【例 5.117】 分配内时序控制 Verilog HDL 描述的例子 2。

```
a <= repeat(a + b) @(posedge phi1 or negedge phi2) data;
```

视频讲解

5.7.8 块语句

块语句提供一个方法,将多条语句组合在一起。这样,它们看上去好像一条语句。Verilog HDL 中有以下两类块语句。

(1) 顺序块(begin…end)。顺序块中的过程语句按给定次序顺序执行。

(2) 并行块(fork…join)。并行块中的过程语句并行执行。

1. 顺序块

顺序块中的语句按顺序方式执行。每条语句中的延迟值与其前面语句执行的仿真时间相关。一旦顺序块执行结束,继续执行跟随顺序块过程的下一条语句。

【例 5.118】 顺序块内分配/赋值 Verilog HDL 描述的例子 1。

```
begin
    areg = breg;
    creg = areg;                                        // creg 保存 breg 的值
end
```

在该例子中的顺序块使得两个分配/赋值有确定的结果。首先执行第一个分配/赋值,并且在控制传递到第二个分配之前更新 areg。

【例 5.119】 顺序块内分配/赋值 Verilog HDL 描述的例子 2。

```
begin
    areg = breg;
    @(posedge clock) creg = areg;                       // 延迟分配/赋值,直到时钟的上升沿
end
```

可以在顺序块中使用延迟控制,以在时间上分离两个分配/赋值。

【例 5.120】 顺序块内分配/赋值 Verilog HDL 描述的例子 3。

```
parameter d = 50;                                       // parameter 关键字定义参数 d
reg [7:0] r;                                            // reg 关键字定义 reg 类型变量 r
```

```
begin                                          // 由顺序延迟控制波形
  #d r = 'h35;
  #d r = 'hE2;
  #d r = 'h00;
  #d r = 'hF7;
  #d -> end_wave;                              // 触发称为 end_wave 的事件
end
```

该例子说明使用顺序块和延迟控制的组合来指定时序波形。

2. 并行块

并行块应该具有以下特征。

(1) 语句应该同时执行。

(2) 每条语句的延迟值应相对于进入块的仿真时间进行考虑。

(3) 延迟控制可用于为分配/赋值提供时间排序。

(4) 当执行最后一条按时间排序的语句时,控制将从块中传递出去。

fork-join 块中的时序控制不必按时间顺序排序。

【例 5.121】 并行块内分配的 Verilog HDL 描述的例子 1。

```
fork
    #50 r = 'h35;
    #100 r = 'hE2;
    #150 r = 'h00;
    #200 r = 'hF7;
    #250 -> end_wave;
join
```

该例子中产生的波形与例 5.120 产生的波形完全相同。

3. 块名字

顺序块和并行块都可以通过在关键字 begin 或 fork 后面添加块名字来命名。块的命名有下面几个目的。

(1) 它允许为块声明本地变量、参数和命名事件。

(2) 它允许在诸如 disable 这样的语句中引用块。

所有的变量应该是静态的,也就是说,所有变量都有一个唯一的位置,离开或进入块不会影响保存在它们中的值。

块名字提供了在任何仿真时间唯一标识所有变量的方法。

4. 启动和结束时间

顺序块和并行块都有启动和结束时间的概念。对于顺序块,启动时间是执行第一条语句时,结束时间是执行最后一条语句时。对于并行块,所有语句的启动时间是相同的,结束时间是最后一次执行按时间顺序语句的时间。

顺序块和并行块可以互相嵌入,允许更容易表达复杂的控制结构,并具有更高度的结构。当块彼此嵌入时,块启动和结束的时间很重要。在达到块的结束时间之前,即在块完全完成之前,不得继续执行块后面的语句。

【例 5.122】 并行块内分配的 Verilog HDL 描述的例子 2。

```
fork
    #250 -> end_wave;
    #200 r = 'hF7;
    #150 r = 'h00;
    #100 r = 'hE2;
    #50 r = 'h35;
join
```

与例 5.115 相比较，该例子以相反的顺序书写代码，但仍然产生相同的波形。

【例 5.123】 并行块内分配的 Verilog HDL 描述的例子 3。

```
begin
    fork
        @Aevent;
        @Bevent;
    join
    areg = breg;
end
```

当在两个单独的事件发生后进行赋值时，称为事件的连接，fork-join 块就会有用。两个事件可以以任何顺序发生(甚至可以在同一仿真时间发生)，fork-join 块将完成，并进行分配/赋值。相反，如果 fork-join 块是一个 begin-end 块，并且 Bevent 发生在 Aevent 之前，那么该块将等待下一个 Bevent。

【例 5.124】 并行块和顺序块嵌套的 Verilog HDL 描述的例子。

```
fork
    @enable_a
    begin
        #ta wa = 0;
        #ta wa = 1;
        #ta wa = 0;
    end
    @enable_b
    begin
        #tb wb = 1;
        #tb wb = 0;
        #tb wb = 1;
    end
join
```

该例子显示了两个顺序块，每个块将在其控制事件发生时执行。事件控制在 fork-join 块中，因为它们并行执行，所以顺序块也可以并行执行。

5.7.9 结构化的过程

在 Verilog HDL 中，所有的过程由下面 4 种语句指定。

(1) initial 结构。

(2) always 结构。

(3) 任务。

(4) 函数。

initial 和 always 结构在仿真开始的时候使能。initial 结构只能执行一次，其活动在语句完成时结束。相反，always 结构应重复执行，只有在仿真结束时，其活动才会停止。initial 和 always 结构之间不应该有隐含的执行顺序。initial 结构不需要在 always 结构之前进行调度和执行。模块中可定义的 initial 和 always 结构的数量不应该有限制。

任务(task)和函数(function)是从其他过程中的一个或多个位置使能的过程。

1. initial 语句

【例 5.125】 initial 语句 Verilog HDL 描述的例子。

```
initial begin
    areg = 0;                                   //初始化一个 reg
for (index = 0; index < size; index = index + 1)
    memory[index] = 0;                          //初始化存储器字
end
```

在该例子中,使用 initial 结构在仿真开始时初始化变量。

【例 5.126】 包含延迟控制的 initial 语句 Verilog HDL 描述的例子。

```
initial begin
  inputs = 'b000000;                    //在 0 时刻初始化
  #10 inputs = 'b011001;                //第 1 个模式
  #10 inputs = 'b011011;                //第 2 个模式
  #10 inputs = 'b011000;                //第 3 个模式
  #10 inputs = 'b001000;                //第 4 个模式
end
```

initial 结构的另一个典型用法是波形描述语句,执行一次波形描述用于向被测试的电路提供激励源。

2. always 语句

在仿真期间,always 结构连续地重复。always 结构由于其循环本质,当和一些形式的时序控制一起使用时,它是非常有用的。如果一个 always 结构无法控制仿真时间的提前,将引起死锁。

【例 5.127】 带有零延迟控制 always 语句 Verilog HDL 描述的例子。

```
always areg = ~areg;
```

【例 5.128】 带有延迟控制 always 语句 Verilog HDL 描述的例子。

```
always #half_period areg = ~areg;
```

5.7.10 设计实例四:同步和异步复位 D 触发器的设计与实现

在复杂数字系统设计中,同步和异步复位是两个重要的概念,其描述方法也截然不同。简单来说,同步复位就是在时钟边沿采样复位信号的电平。当复位信号电平有效时,复位触发器的输出;异步复位就是只要复位信号有效,就复位触发器的输出。本节将通过一个具体的设计实例来说明两者的区别。在该设计实例中,使用了使能端输入,当使能端有效时,D 触发器才能正常工作,如代码清单 5-47 所示。对该设计进行综合后仿真的文件,如代码清单 5-48 所示。

代码清单 5-47 top.v 文件

```
module top(                          // module 关键字定义模块 top
input rst,                           // input 关键字定义输入端口 rst
input ce,                            // input 关键字定义输入端口 ce
input clk,                           // input 关键字定义输入端口 clk
input d,                             // input 关键字定义输入端口 d
output reg q1,                       // output 和 reg 关键字定义 reg 类型输出端口/变量 q1
output reg q2                        // output 和 reg 关键字定义 reg 类型输出端口/变量 q2
    );
initial                              // initial 关键字定义初始化部分
begin                                // begin 关键字标识初始化部分的开始,类似 C 语言的"{"
    q1 = 1'b0;                       // 阻塞过程分配/赋值,端口 q1 分配初值"0"
    q2 = 1'b0;                       // 阻塞过程分配/赋值,端口 q2 分配初值"0"
end                                  // end 关键字标识初始化部分的结束,类似 C 语言的"}"

/************************ 异步复位电路 ***************************/
always @(posedge rst,posedge clk)    // always 关键字声明过程语句,()里为敏感信号 rst 和 clk
begin                                // begin 关键字标识过程语句的开始,类似 C 语言的"{"
if(rst)                              // if 关键字定义条件语句,如果 rst 为逻辑"1"(高电平)
  q1 <= 1'b0;                        // 非阻塞过程分配/赋值,端口 q1 复位为"0"
else                                 // else 关键字定义,当 posedge clk(clk 上升沿)出现
  if(ce == 1'b1)                     // if 关键字定义条件语句,如果 ce 等于逻辑"1"(高电平)
     q1 <= d;                        // 非阻塞过程分配/赋值,端口 q1 赋值/分配 d
```

```
end                                    // end 标识 always 过程语句的结束,类似 C 语言的"}"

/************************ 同步复位电路 ***************************/
always @(posedge clk)                  // always 关键字声明过程语句,()内为敏感信号 clk
begin                                  // begin 关键字标识过程语句的开始,类似 C 语言的"{"
if(rst)                                // 在时钟上升沿到来时,如果 rst 为逻辑"1"(高电平)
  q2 <= 1'b0;                          // 非阻塞过程分配/赋值,端口 q2 复位为"0"
else                                   // 在时钟上升沿到来时,如果 rst 为逻辑"0"(低电平)
  if(ce == 1'b1)                       // if 关键字定义条件语句,如果 ce 等于逻辑"1"(高电平)
    q2 <= d;                           // 非阻塞过程分配/赋值,端口 q2 赋值/分配 d
end                                    // end 标识 always 过程语句的结束,类似 C 语言的"}"
endmodule                              // endmodule 标识模块 top 的结束
```

在高云云源软件中,对代码清单 5-47 给出的代码进行综合,得到综合后的电路结构如图 5.10 所示。

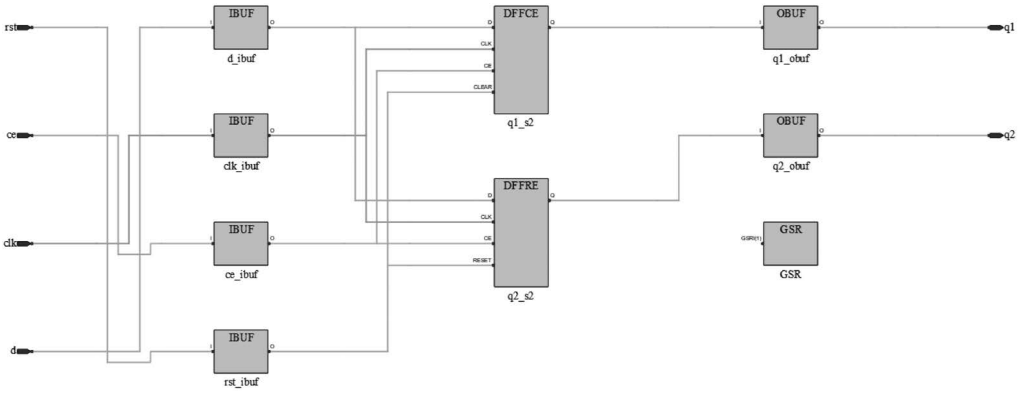

图 5.10　综合后的电路结构

注：在配套资源\eda_verilog\example_5_10\目录下,用高云云源软件打开 example_5_10.gprj。

代码清单 5-48　test.v 文件

```
`timescale 1ns / 1ps                   // 预编译指令 timescale 定义,精度和分辨率分别为 1ns 和 1ps
module test;                           // 关键字 module 定义模块 test
reg rst,ce,clk,d;                      // reg 关键字定义 reg 类型变量 rst、ce、clk 和 d
wire q1,q2;                            // wire 关键字定义网络类型 q1 和 q2
integer i;                             // integer 关键字定义整型变量 i
top Inst_top(                          // Verilog 元件例化,将调用 top 模块,将其例化为 Inst_top
  .rst(rst),                           // top 模块的 rst 端口连接到 test 模块的 reg 类型变量 rst
  .ce(ce),                             // top 模块的 ce 端口连接到 test 模块的 reg 类型变量 ce
  .clk(clk),                           // top 模块的 clk 端口连接到 test 模块的 reg 类型变量 clk
  .d(d),                               // top 模块的 d 端口连接到 test 模块的 reg 类型变量 d
  .q1(q1),                             // top 模块的 q1 端口连接到 test 模块的网络类型 q1
  .q2(q2)                              // top 模块的 q2 端口连接到 test 模块的网络类型 q2
    );

initial                                // initial 关键字声明初始化部分
begin                                  // begin 关键字标识初始化部分的开始,类似 C 语言的"{"
  rst = 1'b1;                          // 变量 rst 分配/赋值"1"
  #50;                                 // "#"标识延迟,#50 表示持续 50ns,由 timescale 定义
  rst = 1'b0;                          // 变量 rst 分配/赋值"0"
  #20;                                 // "#"标识延迟,#20 表示持续 20ns,由 timescale 定义
  rst = 1'b1;                          // 变量 rst 分配/赋值"1"
  #50;                                 // "#"标识延迟,#50 标识持续 50ns,由 timescale 定义
  rst = 1'b0;                          // 变量 rst 分配/赋值"0"
end                                    // end 标识初始化部分的结束,类似 C 语言的"}"

initial                                // initial 关键字声明初始化部分
```

```
begin                                  // begin 关键字标识初始化部分的开始,类似 C 语言的"{"
  clk = 1'b0;                          // 变量 clk 分配/赋值"1"
end                                    // end 标识初始化部分的结束,类似 C 语言的"}"

initial                                // initial 关键字声明初始化部分
begin                                  // begin 关键字标识初始化部分的开始,类似 C 语言的"{"
  i = 0;                               // 整型变量 i 赋值/分配值 0
  while(1)                             // while 关键字定义无限循环
  begin                                // begin 关键字标识无限循环的开始,类似 C 语言的"{"
    d = #23 i[0];                      // 延迟 23ns 后,调度分配/赋值,将 i[0]分配/赋值给变量 d
    i = i + 1;                         // 整型变量 i 的值递增后分配/赋值给自己
  end                                  // end 标识无限循环 while 的结束,类似 C 语言的"}"
end                                    // end 标识初始化部分的结束,类似 C 语言的"}"

initial                                // initial 关键字声明初始化部分
begin                                  // begin 关键字标识无限循环的开始,类似 C 语言的"{"
  ce = 1'b1;                           // 变量 ce 分配/赋值"1"
end                                    // end 标识初始化部分的结束,类似 C 语言的"}"

always                                 // always 关键字声明过程语句
begin                                  // begin 关键字标识无限循环的开始,类似 C 语言的"{"
  #20 clk = ~clk;                      // "#"标识延迟,#20 标识持续 20ns,clk 取反后赋值给 clk
end                                    // end 标识初始化部分的结束,类似 C 语言的"}"
endmodule                              // endmodule 标识 test 模块的结束
```

在 ModelSim 软件中,对代码清单 5-48 给出的代码进行综合后仿真,综合后仿真结果如图 5.11 所示。

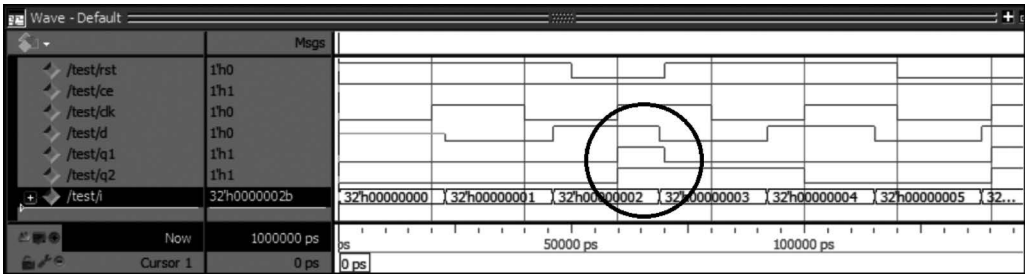

图 5.11　综合后仿真的波形

注:在配套资源\eda_verilog\example_5_11\目录下,用 ModelSim SE-64 10.4c 打开 postsynth_sim.mpf。

从图 5.11 中用大圆圈标记的地方可知,对于异步复位电路,当 rst 信号由逻辑"0"跳变到逻辑"1"后(即由无效变为有效时),q1 的输出立即跳变到逻辑"0"(复位状态);对于同步复位电路,当 rst 信号由逻辑"0"跳变到逻辑"1"后(即由无效变为有效时),q2 的输出并没有立即跳变到逻辑"0"(复位状态),而是在下一个时钟上升沿到来时才跳变到逻辑"0"。

思考与练习 5-10　将代码清单 5-47 中给出的 rst 信号的复位极性由逻辑"1"有效改为逻辑"0"有效,重新执行综合,并查看综合后的电路,说明与图 5.10 的不同(提示:参考高云 FPGA 内触发器的原理和复位的极性),并相应地修改代码清单 5-48 中 rst 的复位向量代码,执行综合后仿真,并查看综合后仿真的波形。

5.7.11　设计实例五:软件算法的硬件实现与验证

本节将下面的软件算法:

$$y(n)=\frac{x(n)+x(n-1)+x(n-2)+x(n-3)}{4}$$

使用硬件实现。通过该算法的硬件实现，说明使用 FPGA 实现算法的本质特点，以及使用硬件实现算法比使用软件实现算法的优势。在该设计中，输入数据 x 的宽度为 8 位，求和后的结果 y 的宽度也为 8 位。

通过观察上面的式子，可知该算法包含：

(1) 延迟运算，即 x(n－1)、x(n－2)和 x(n－3)。在硬件上，延迟运算可使用 FPGA 内的 D 触发器实现。

(2) 加法运算，x(n)＋x(n－1)＋x(n－2)＋x(n－3)。在硬件上，延迟运算可使用 FPGA 内分立的逻辑资源(LUT、进位逻辑等)或片内专用的 DSP 模块实现。

(3) 除法运算，即加法的和除以 4。为了简化硬件的实现结构，将除以 4 的操作转换为向右移 2 位的操作。

该算法的 Verilog HDL 描述，如代码清单 5-49 所示。对该算法执行综合后仿真的 Verilog HDL 描述，如代码清单 5-50 所示。

代码清单 5-49　top.v 文件

```
module top(                              // module 关键字定义模块 top
input rst,                               // input 关键字定义输入端口 rst
input clk,                               // input 关键字定义输入端口 clk
input signed [7:0] d,                    // input 和 signed 关键字定义有符号输入端口 d,8 位
output signed [7:0] res                  // output 和 signed 关键字定义有符号输出端口 res,8 位
);
reg signed [7:0] d1,d2,d3;               // reg 和 signed 关键字定义有符号 reg 类型变量 d1, d2, d3
assign res = (d + d1 + d2 + d3 + 0)>> 2; // d + d1 + d2 + d3 的结果右移两位的结果分配/赋值给 res
always @(negedge rst or posedge clk) // always 过程语句,()内为敏感信号 rst 高电平和 clk 上升沿
begin                                    // begin 关键字标识过程语句的开始,类似 C 语言的"{"
if(!rst)                                 // if 定义条件语句,(!rst)表示 rst 为逻辑"0", !为逻辑取反
begin                                    // begin 关键字标识 if 语句的开始,类似 C 语言的"{"
  d1 <= 8'b0;                            // 非阻塞赋值,d1 分配/赋值为 0
  d2 <= 8'b0;                            // 非阻塞赋值,d2 分配/赋值为 0
  d3 <= 8'b0;                            // 非阻塞赋值,d3 分配/赋值为 0
end                                      // end 关键字标识 if 语句的结束,类似 C 语言的"}"
else                                     // else 条件为 posedge clk(clk 的上升沿)
begin                                    // begin 关键字表示 else 语句的开始,类似 C 语言的"{"
  d1 <= d;                               // d 非阻塞分配/赋值给 d1
  d2 <= d1;                              // d1 非阻塞分配/赋值给 d2
  d3 <= d2;                              // d2 非阻塞分配/赋值给 d3
end                                      // end 关键字标识 else 语句的结束,类似 C 语言的"}"
end                                      // end 关键字标识 always 语句的结束,类似 C 语言的"}"
endmodule                                // endmodule 标识 top 模块的结束
```

在高云云源软件中，打开代码清单 5-49 所对应的 RTL 级电路结构，如图 5.12 所示。

(1) always 过程语句中的三条非阻塞过程分配/赋值语句“d1＜＝d；d2＜＝d1；d3＜＝d2”，生成了三个宽度为 8 位的 D 触发器结构，这三个 D 触发器串联在一起。从前面数字电路的设计实例可知，每个触发器结构充当延迟一个时钟周期的作用，这就是硬件与软件算法的直接映射。

(2) 从每个 8 位宽度触发器的输出连接到 FPGA 内部的加法器，然后右移两位，这与代码中的分配/赋值语句对应。

注：在配套资源\eda_verilog\example_5_12\目录下，用高云云源软件打开 example_5_12.gprj。

代码清单 5-50　test.v 文件

```
`timescale 1ns / 1ps                     // 预编译指令 timescale 定义时间标度精度/分辨率 = 1ns/1ps
module test;                             // module 关键字定义模块 test
reg rst,clk;                             // reg 关键字定义 reg 类型变量 rst 和 clk
reg signed [7:0] d;                      // reg 和 signed 关键字定义有符号 reg 类型变量 d,宽度为 8 位
wire signed [7:0] res;                   // wire 和 signed 关键字定义有符号网络类型变量 res,宽度为 8 位
```

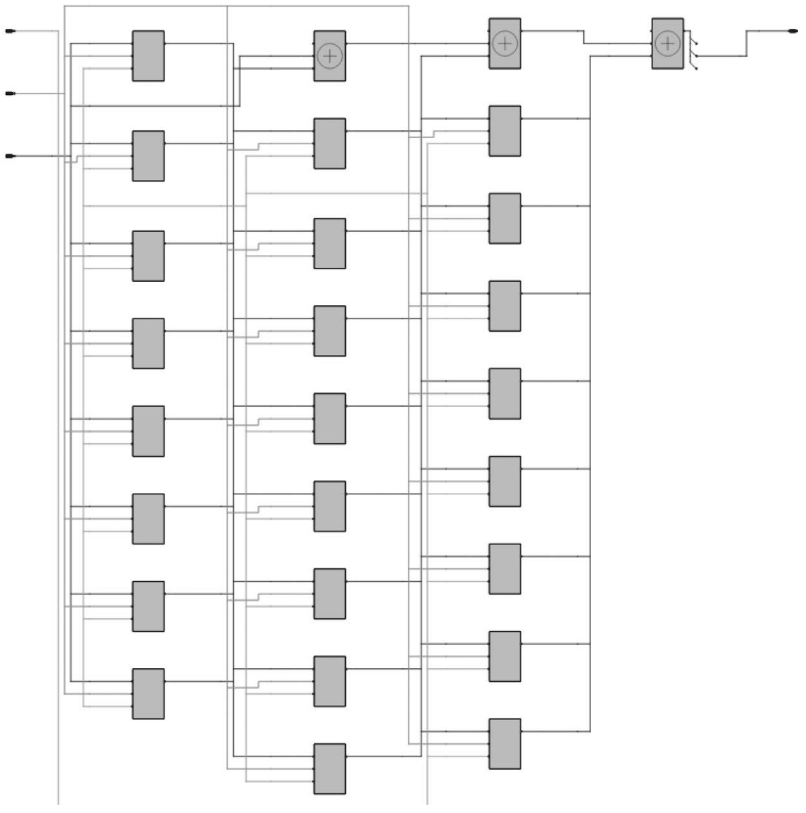

图 5.12　RTL 级的电路结构

```
integer i = 0;                          // integer 关键字定义整型变量 i,初始值为 0
parameter N = 50;                       // parameter 关键字定义参数 N 并分配/赋值 50
top Inst_top(                           // Verilog 元件例化,调用/例化模块 top,将其例化为 Inst_top
.rst(rst),                              // top 模块端口 rst 连接到 test 模块 reg 类型变量 rst
.clk(clk),                              // top 模块端口 clk 连接到 test 模块 reg 类型变量 clk
.d(d),                                  // top 模块端口 d 连接到 test 模块 reg 类型变量 d
.res(res)                               // top 模块端口 res 连接到 test 模块网络 res
);
initial                                 // initial 关键字声明初始化部分
begin                                   // begin 关键字标识初始化部分的开始,类似 C 语言的"{"
  rst = 1'b1;                           // 给 rst 变量分配/赋值"1"
  #20;                                  // "#"标识延迟,#20 表示持续 20ns,单位由 timescale 定义
  rst = 1'b0;                           // 给 rst 变量分配/赋值"0"
end                                     // end 标识初始化部分的结束

initial                                 // initial 关键字声明初始化部分
begin                                   // begin 关键字标识初始化部分的开始,类似 C 语言的"{"
  clk = 1'b0;                           // 给 clk 变量分配/赋值"0"
end                                     // end 标识初始化部分的结束

always                                  // always 关键字声明过程语句
begin                                   // begin 关键字标识过程语句的开始,类似 C 语言的"{"
  #10 clk = ~clk;                       // 每隔 10ns clk 信号取反,产生方波信号,用作 clk 激励
end                                     // end 标识过程语句的结束

initial                                 // initial 关键字声明初始化部分
begin                                   // begin 关键字标识过程语句的开始,类似 C 语言的"{"
 #7;                                    // "#"标识延迟,#7 表示延迟 7ns,ns 由 timescale 定义
 while(1)                               // while 关键字声明无限循环
 begin                                  // begin 关键字标识无限循环的开始,类似 C 语言的"{"
```

```
    /* 调用系统任务 $sin()产生正弦波,乘 128 转换为 8 位数,调用系统任务 $rtoi 转换为整数 */
    d = $rtoi( $sin(2 * 3.1415926 * (i % N)/N) * 128);
    i = i + 1;                          // 整型变量 i 递增后分配/赋值给自己
    #20;                                // "#"标识延迟,#20 表示持续 20ns,ns 由 timescale 定义
  end                                   // end 关键字标识无限循环的结束,类似 C 语言的"}"
end                                     // end 关键字标识初始化部分的结束,类似 C 语言的"}"
endmodule                               // endmodule 关键字标识 test 模块的结束
```

对该电路执行综合后仿真,为了直观地看到以波形显示的 d[7:0]和 res[7:0],分别右击 d[7:0]和 res[7:0],出现浮动菜单,选择 Format→Analog(Automatic)。然后,再分别右击 d[7:0]和 res[7:0],出现浮动菜单,选择 Radix→ Decimal。综合后仿真的结果如图 5.13 所示。

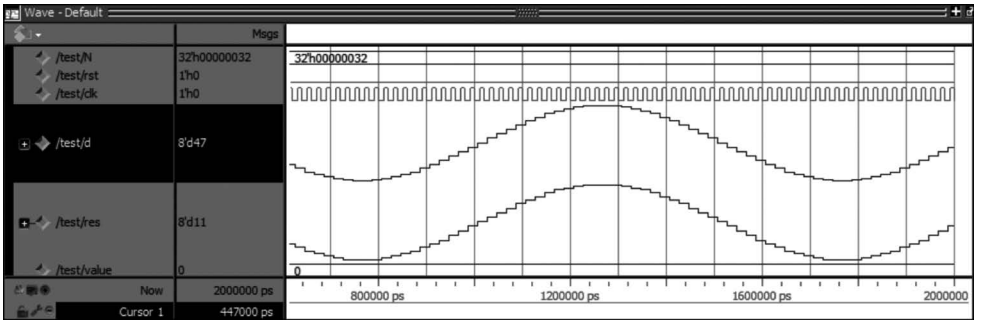

图 5.13　对设计执行综合仿真的结果

注:在配套资源\eda_verilog\example_5_13\目录下,用 ModelSim SE-64 10.4c 打开 postsynth_sim.mpf。

思考与练习 5-11　将代码清单 5-49 中的非阻塞赋值全部改为阻塞赋值,然后重新查看 RTL 级的电路结构有什么不同。通过所生成电路结构的不同,说明非阻塞赋值和阻塞赋值的区别。

5.8　Verilog HDL 任务和函数

任务和函数提供了在一个描述中从不同位置执行公共程序的能力,它们也提供了将一个大的程序分解成较小程序的能力,这样更容易阅读和调试源文件描述。


视频讲解

5.8.1　任务和函数的区别

下面给出了任务和函数的区别规则。

(1) 函数应该在一个仿真时间单位内执行;任务可以包含时间控制语句。

(2) 函数不能使能任务。但是,任务可以使能其他任务和函数。

(3) 函数至少有一个 input 类型的参数,没有 output 或者 inout 类型的参数;一个任务可以有零个或者更多任意类型的参数。

(4) 函数返回一个值,而任务不返回值。

函数的目的是通过返回一个值来响应输入的值。一个任务可以支持多个目标,可以计算多个结果的值。通过一个任务调用,只能返回传递的 output 和 inout 类型的参数结果。使用函数作为表达式内的一个操作数,由函数返回操作数的值。

5.8.2　任务和任务使能

通过定义要传递给任务的参数值和接收结果的变量的语句使能任务。当任务完成后,控制应返回到使能过程。因此,如果任务中有时序控制,则使能任务的时间可能与返回控制的时

间不同。一个任务可以使能其他任务,而这些任务又可以使能其他的任务,而不限制使能任务的数量。无论使能了多少任务,在所有使能的任务完成之前,控制都不会返回。

1. 任务声明

在 Verilog HDL 中,有两种可选的任务声明语法。

第一种语法应该以关键字 task 开头,后面跟可选的关键字 automatic,后面再跟着任务的名字和分号,并以关键字 endtask 结束。关键字 automatic 声明了一个可重入的自动任务,为每个并发任务入口动态分配所有任务声明。任务声明可以指定下面的内容,包括 input 参数、output 参数、inout 参数,以及可以在过程块中声明的所有数据类型。

第二种语法应该以关键字 task 开头,后面跟任务名字和括号括起来的任务端口列表。任务端口列表应该由 0 个或多个逗号分隔的任务端口项组成。右括号后应该有分号。任务体应紧跟其后,然后是关键字 endtask。

没有可选关键字 automatic 的任务是静态任务,所有声明的条目都是静态分配的。这些条目应该在同时执行的任务的所有用途中共享。带有可选关键字 automatic 的任务是自动任务。自动任务中声明的所有条目都在每次调用时动态分配。层次化引用不能访问自动任务条目。自动任务可以通过使用其层次结构名字来调用。

2. 任务使能和参数传递

任务使能语句应以逗号分隔的表达式列表形式传递参数,表达式列表用括号括起来。

如果任务定义没有参数,则任务使能语句中不应提供参数列表;否则,将有一个有序的表达式列表,该列表与任务定义中的参数列表的长度和顺序相匹配。空表达式不能用作任务使能语句中的参数。

如果任务中的一个参数声明为 input,则对应的表达式可以是任何表达式。参数列表中表达式的求值顺序未定义。如果参数声明为 output 或 inout,则表达式应限制为在过程分配/赋值左侧有效的表达式。下面的条目满足这个要求。

(1) reg、integer、real、realtime 和 time 变量。

(2) 存储器引用。

(3) reg、integer 和 time 变量的并置。

(4) 存储器引用的并置。

(5) reg、integer 和 time 变量的位选择和部分选择。

任务使能语句的执行应将使能语句中列出的表达式的输入值传递给任务中指定的参数。执行任务返回时,将该任务 output 和 input 类型的参数的值传递给任务使能语句中的相应变量。任务的所有参数应通过值而不是引用(即指向值的指针)传递。

【例 5.129】 包含 5 个参数的 task 基本结构 Verilog HDL 描述的例子。

```
task my_task;
input a, b;
inout c;
output d, e;
begin
…                                    //执行任务的语句
…
c = foo1;                            //初始化结果寄存器的分配
d = foo2;
e = foo3;
end
endtask
```

或者采用下面的描述方式：

```
task my_task (input a, b, inout c, output d, e);
begin
    …                                  //执行任务的语句
    …
    c = foo1;                          //分配用于初始化结果寄存器
    d = foo2;
    e = foo3;
end
endtask
```

用下面的语句使能任务：

```
my_task (v, w, x, y, z);
```

其中，任务使能参数(v,w,x,y,z)对应于任务所定义的参数(a,b,c,d,e)。

在使能任务期间，input 和 inout 类型的参数(a,b,c)接受传递的值(v,w,x)。这样，执行任务使能调用产生下面的分配：

```
a = v;
b = w;
c = x;
```

作为任务处理的一部分，定义的任务 my_task 将计算结果分配到 c,d,e。当完成任务时，下面的分配将计算得到的值，并返回到被执行的调用过程。

```
x = c;
y = d;
z = d;
```

【例 5.130】 描述交通灯时序 task 语句 Verilog HDL 描述的例子，如代码清单 5-51 所示。

代码清单 5-51　交通灯任务的 Verilog HDL 描述

```
module traffic_lights;
reg clock, red, amber, green;
parameter on = 1, off = 0, red_tics = 350,
          amber_tics = 30, green_tics = 200;
//初始化颜色
initial red = off;
initial amber = off;
initial green = off;
always begin                           //控制灯的时序
    red = on;                          //打开红灯
    light(red, red_tics);              //等待
    green = on;                        //打开绿灯
    light(green, green_tics);          //等待
    amber = on;                        //打开琥珀色灯
    light(amber, amber_tics);          //等待
end
//等待'tics'的任务，上升沿时钟
task light;
output color;
input [31:0] tics;
begin
repeat (tics) @ (posedge clock);
color = off;                           //关闭灯
end
endtask
always begin                           //时钟波形
#100 clock = 0;
#100 clock = 1;
end
endmodule                              //traffic_lights 模块结尾
```

3. 任务存储器使用和并行运行

一个任务可以同时使能多次,将并行调用的每个自动任务的所有变量进行复制,用于保存该调用的状态。静态任务的所有变量是静态的,即在一个模块例化中,每个变量对应于每个声明的本地变量,而不管并行运行的任务个数。然而,在模块不同实例中的静态任务,应有彼此独立的存储。

在静态任务里声明的变量(包含 input、output 和 inout 类型的参数),应在调用之间保留它们的值。

在自动任务里声明的变量(包含 output 类型的变量)应在执行进入其范围时初始化为默认的初始化值。根据任务使能语句中所列出的参数,将 input 和 inout 类型参数初始化为表达式传递的值。

因为在自动任务中声明的变量在任务调用结束时将进行分配解除,因此它们不能用在该点之后引用它们的某些结构中。

(1) 不能使用非阻塞分配或者过程连续分配对其分配值。

(2) 不能通过过程连续分配或者过程 force 语句引用它们。

(3) 不能在非阻塞分配的分配内事件控制中引用它们。

(4) 不能使用系统任务 $monitor 和 $dunpvars 跟踪它们。

5.8.3 禁止命名的块和任务

disable 语句提供了一种能力,用于终止与并行活动过程相关的活动,而保持 Verilog HDL 过程描述的本质。disable 语句提供了用于在执行所有任务语句前,终止一个任务的一个机制。如退出一个循环语句,或跳出语句,用于继续一个循环语句的其他循环。在处理异常条件时,disable 非常有用,如硬件中断和全局复位。

任何形式的 disable 语句应该终止一个任务或命名块的活动,应该继续执行在跟随块或跟随任务使能语句后的语句。命名块或任务内使能的所有活动也应终止。如果任务使能语句是嵌套的(即一个任务使能其他任务,并且一个任务又使能另一个),则禁止链中的任务将禁用链上向下的所有任务。如果多次使能一个任务,则禁止该任务的所有激活。

如果禁止任务,则不会指定由任务启动的以下活动的结果。

(1) output 和 input 参数的结果。

(2) 调度,但不执行,非阻塞赋值。

(3) 过程连续分配(assign 和 force 语句)。

在块和任务中可以使用 disable 语句来禁止包含 disable 语句的特定块或任务。disable 语句可用于禁止函数中命名的块,但不能用于禁止函数。如果函数中的 disable 语句禁止了调用该函数的块或任务,则行为未定义。对于任务的所有并发执行,禁止自动任务或自动任务内的块与常规任务相同。

【例 5.131】 禁止块 Verilog HDL 描述的例子 1。

```
begin : block_name
    rega = regb;
    disable block_name;
    regc = rega;                    //不执行该分配
end
```

【例 5.132】 禁止块 Verilog HDL 描述的例子 2。

```
begin : block_name
…
```

```
if (a == 0)
disable block_name;
…
end                                    //结束命名的块
//继续执行命名块后面的代码
  …
```

【例 5.133】 禁止任务 Verilog HDL 描述的例子。

```
task proc_a;
begin
    …
    …
if (a == 0)
    disable proc_a;                    //如果真,则返回
    …
    …
end
endtask
```

【例 5.134】 禁止块 Verilog HDL 描述的例子 3。

```
begin : break
  for (i = 0; i < n; i = i + 1) begin : continue
    @clk
      if (a == 0)                      //"继续"循环
          disable continue;
        statements
        statements
    @clk
      if (a == b)                      //从循环中断
          disable break;
        statements
        statements
    end
end
```

在该例中,disable 的功能相当于 C 语言中的 continue 和 break 语句。

【例 5.135】 在 fork-join 块中禁止语句 Verilog HDL 描述的例子。

```
fork
  begin : event_expr
    @ev1;
    repeat (3) @trig;
    #d action (areg, breg);
  end
    @reset disable event_expr;
join
```

【例 5.136】 在 always 块中禁止语句 Verilog HDL 描述的例子。

```
always begin : monostable
    #250 q = 0;
end
always @retrig begin
    disable monostable;
    q = 1;
end
```

5.8.4 函数和函数调用

函数的目的是返回要在表达式中使用的值。本节介绍了定义和使用函数的方法。

1. 函数声明

函数定义应该以关键字 function 开头，后面跟可选的关键字 automatic，再跟函数返回值的可选范围或类型，后面跟着函数的名字，再是分号或者括在括号中的函数端口列表，然后是分号，最后以关键字 endfunction 结束。

函数的范围或类型的使用是可选的。如果没有指定函数的范围或类型，则函数默认的返回值是标量。如果使用，则指定函数的返回值是 real、integer、time、realtime 或具有[n:m]范围的向量(可选有符号的)。

函数应该至少声明一个输入。

关键字 automatic 声明一个可重入的自动函数，所有函数声明都为每个并发函数调用动态分配。不能通过层次引用来访问自动函数项。自动函数可以通过使用其层次结构名字来调用。

使用两种方法中的其中一种来声明函数的输入。第一种方法是函数名字后面应加上分号，分号后面应该跟着一个或多个输入声明(可选的混合块条目声明)。在函数条目声明之后，应该有行为描述语句，然后是 endfunction 关键字。

第二种方法是有函数名字，后面跟着一个左括号以及一个或多个输入声明，用逗号分隔。在所有输入声明之后，应该有一个右括号和一个分号。在分号之后，应该有 0 个或多个块条目声明，后面跟着行为语句，然后是 endfunction 关键字。

【例 5.137】 函数声明 Verilog HDL 描述的例子。

```
function[7:0] getbyte;
input [15:0] address;
begin
  …
  getbyte = result_expression;
end
endfunction
```

也可以用下面的函数格式：

```
function[7:0] getbyte (input [15:0] address);
begin
    …
    getbyte = result_expression;
end
endfunction
```

2. 从函数返回值

函数定义应隐式声明与函数同名的函数内部变量。这个变量默认为一位 reg 或与函数声明中指定的类型相同。函数定义通过将函数结果分配给与函数同名的内部变量来初始化函数的返回值。

在声明函数的作用域中声明与函数同名的另一个对象是非法的。在函数内部，有一个带有函数名字的隐含变量，可以在函数内的表达式中使用。因此，在函数范围内声明与函数同名的另一个对象也是非法的。

3. 函数调用

函数调用是表达式中的操作数。

函数调用参数的求值顺序未定义。

【例 5.138】 函数调用 Verilog HDL 描述的例子。

```
word = control ? {getbyte(msbyte), getbyte(lsbyte)}:0;
```

通过并置/连接，该例子两次调用 getbyte 函数的结果来创建一个字。

4. 函数规则

函数比任务有更多的限制。以下规则管理函数的用法。

(1) 函数定义不应该包含更多时间控制语句，即任何包含 #、@或 wait 的语句。

(2) 函数不应使能任务。

(3) 函数定义应至少包含一个输入参数。

(4) 函数定义不得将任何参数声明为 output 或 inout。

(5) 函数不应该有任何非阻塞赋值或过程连续赋值。

(6) 函数不应该有任何事件触发器。

【例 5.139】 可重入函数调用 Verilog HDL 描述的例子，如代码清单 5-52 所示。

代码清单 5-52 可重入函数的 Verilog HDL 描述

```
module tryfact;
//定义函数
function automatic integer factorial;
input [31:0] operand;
integer i;
if (operand >= 2)
    factorial = factorial (operand - 1) * operand;
else
    factorial = 1;
endfunction

//测试函数
integer result;
integer n;
initial begin
for (n = 0; n <= 7; n = n + 1) begin
    result = factorial(n);
    $display("%0d factorial = %0d", n, result);
    end
end
endmodule                                    //tryfact 结尾
```

该例子定义了一个名为 factorial 的函数，该函数返回一个整数值。递归调用阶乘函数并打印结果。

仿真结果如下：

```
0 factorial = 1
1 factorial = 1
2 factorial = 2
3 factorial = 6
4 factorial = 24
5 factorial = 120
6 factorial = 720
7 factorial = 5040
```

5. 常数函数

常数函数的调用支持在对设计进行详细的描述时，建立复杂计算的值。对于一个常数函数的调用，模块调用函数的参数是一个常数表达式。常数函数是 Verilog HDL 普通函数的子集，应该满足以下的约束。

(1) 它们不应包含层次引用。

(2) 在一个常数函数内的任意函数调用，应该是当前模块的本地函数。

(3) 它可以调用任何常数表达式中所允许的系统函数,对其他系统函数的调用是非法的。

(4) 忽略在一个常数函数内的所有系统任务。

(5) 在使用一个常数函数调用前,应该定义函数内所有的参数值。

(6) 所有不是参数和函数的标识符,应该在当前函数内本地声明。

(7) 如果使用 defparam 语句直接或者间接影响的任何参数值,则结果是未定义的,这将导致一个错误,或使函数返回一个未确定的值。

(8) 它们不应该在一个生成块内声明。

(9) 在任何要求一个常数表达式的上下文中,它们本身不能使用常数函数。

常数函数调用在详细描述(elaboration)时求值。它们的执行不会影响仿真时使用的变量的初始值,也不会影响详细描述时函数的多次调用。在每种情况下,变量都会按照正常仿真的方式进行初始化。

下面的例子定义了一个常数函数 clogb2,根据 ram 来确定一个 ram 地址线的宽度。

【例 5.140】 常数函数调用 Verilog HDL 描述的例子,如代码清单 5-53 所示。

代码清单 5-53 常数函数的 Verilog HDL 描述

```
module ram_model (address, write, chip_select, data);
    parameter data_width = 8;
    parameter ram_depth = 256;
    localparam addr_width = clogb2(ram_depth);
    input [addr_width - 1:0] address;
    input write, chip_select;
    inout [data_width - 1:0] data;
//定义 clogb2 函数
    function integer clogb2;
        input [31:0] value;
      begin
        value = value - 1;
        for (clogb2 = 0; value > 0; clogb2 = clogb2 + 1)
            value = value >> 1;
      end
    endfunction
reg [data_width - 1:0] data_store[0:ram_depth - 1];
    //ram_model 剩余部分
```

例化这个 ram_model,包含参数分配:

```
ram_model #(32,421) ram_a0(a_addr,a_wr,a_cs,a_data);
```

5.9 Verilog HDL 层次化结构

Verilog HDL 支持将一个模块嵌入其他模块的层次化描述结构。高层次模块创建低层次模块的例化,并且通过 input、output 和 inout 端口进行通信。这些模块的端口为标量或者向量。

顶层模块是源文本中包含的模块,但不出现在任何模块例化语句中。即使模块例化出现在自身未例化的生成块中,这也适用。模型应包含至少一个顶层模块。

5.9.1 模块例化

例化(Instantiation)允许一个模块将另一个模块的副本合并到自身中。模块定义不嵌套。换句话说,一个模块定义不应在其 module-endmodule 关键字对中包含另一个模块的文本。

模块定义通过例化另一个模块来嵌套它。模块例化语句创建已定义模块的一个或多个命名实例。

更通俗地说,在一个模块中例化另一个模块,也就是在一个模块中调用另一个模块,或者说在一个模块中嵌入另一个模块。模块例化的结果就是产生被调用模块的副本,称为被调用模块的实例。可以在单个模块例化语句中指定一个或多个模块实例(模块的相同副本)。例如,计数器模块可以例化 D 触发器模块以创建触发器的多个实例。

模块例化可以包含范围规范,这允许创建实例数组。为门和原语定义的实例数组的语法和语义也适用于模块。

端口连接列表仅用于定义有端口的模块。然而,括号始终是必需的。当使用有序端口连接方法给出端口连接列表时,列表中的第一个元素应该连接到模块中声明的第一个端口,第二个连接到第二个端口,以此类推。

连接可以是对变量或网络标识符、表达式或空白的简单引用。表达式可以用于向模块输入端口提供值。空白端口连接应表示不连接端口的情况。当通过名字连接端口时,可以通过在端口列表中省略它或在括号中不提供表达式来指示未连接的端口(例如,.port_name())。


视频讲解

5.9.2 覆盖模块参数值

Verilog HDL 提供了两种定义参数的方法,可以在模块参数端口列表中定义,也可以作为模块的条目定义。一个模块声明中可以包含一种或两种类型的参数定义,也可以不包含参数定义。

模块参数可以有类型说明和范围说明。根据下面的规则,确定参数对一个参数类型和范围覆盖的结果。

(1) 没有类型和范围说明的参数声明,默认由最终参数值的类型和范围确定该参数的属性。

(2) 包含范围说明但没有类型说明的参数,其范围是参数声明的范围且类型是无符号的。覆盖值将转换到参数的类型和范围。

(3) 包含类型说明但没有范围说明的参数,其类型是参数声明的类型。覆盖值将转换到参数的类型。有符号的参数将默认到最终覆盖参数值的范围。

(4) 包含有符号类型说明和范围说明的参数,其类型是有符号的,范围是参数声明的范围。覆盖值将最终转换到参数的类型和范围。

【例 5.141】 参数 Verilog HDL 描述的例子,如代码清单 5-54 所示。

代码清单 5-54 参数的 Verilog HDL 描述

```
module generic_fifo
#(parameter MSB = 3, LSB = 0, DEPTH = 4)
//可以覆盖这些参数
(input [MSB:LSB] in,
input clk, read, write, reset,
output [MSB:LSB] out,
output full, empty );
localparam FIFO_MSB = DEPTH * MSB;
localparam FIFO_LSB = LSB;
//这些参数是本地的,不能被覆盖
//通过修改参数影响它们,模块将正常工作
reg [FIFO_MSB:FIFO_LSB] fifo;
    reg [LOG2(DEPTH):0] depth;
always @(posedge clk or reset) begin
  casex ({read,write,reset})
    //实现 fifo
```

```
  endcase
end
endmodule
```

Verilog HDL 提供了两种方法,用于修改非本地参数的值。

(1) defparam 语句,允许使用层次化的名字,给参数分配值。

(2) 模块例化参数值分配,允许在模块例化行内给参数分配值。通过列表的顺序或者名字,分配模块例化参数的值。

如果 defparam 分配和模块例化参数冲突时,模块内的参数将使用 defparam 指定的值。

【例 5.142】 分配参数值 Verilog HDL 描述的例子,如代码清单 5-55 所示。

代码清单 5-55 分配参数值的 Verilog HDL 描述

```
module foo(a,b);
  real r1,r2;
  parameter [2:0] A = 3'h2;
  parameter B = 3'h2;
  initial begin
    r1 = A;
    r2 = B;
     $display("r1 is %f r2 is %f",r1,r2);
  end
endmodule                              //foo

module bar;
    wire a,b;
    defparam f1.A = 3.1415;
    defparam f1.B = 3.1415;
    foo f1(a,b);
endmodule                              //bar
```

该例中 A 指定了范围,而 B 没有,所以将 f1.A=3.1415 的浮点数转换为定点数 3。3 的低三位赋值给 A,而 B 由于没有说明范围和类型,所以没有进行转换。

使用 defparam 语句,通过在设计中使用参数的层次化名字,在任何模块例化中,均可修改参数的值。defparam 语句对于一个模块内一组参数值同时覆盖是非常有用的。

1. defparam 语句

使用 defparam 语句,在整个设计过程中,通过参数的层次结构名字修改任意模块实例中的参数值。但是,在生成块实例或实例数组中或其下的层次结构中的 defparam 语句不得修改该层次结构之外的参数值。

生成块的每个实例都被看作一个单独的层次结构范围。因此,该规则意味着,即使其他例化是由同一循环生成结构创建的,生成块中的 defparam 语句也不能以同一生成块的另一例化的参数为目标。

【例 5.143】 Verilog HDL 描述将不会修改参数的值的例子。

```
genvar i;
generate
for (i = 0; i < 8; i = i + 1) begin : somename
    flop my_flop(in[i], in1[i], out1[i]);
    defparam somename[i + 1].my_flop.xyz = i;
end
endgenerate
```

类似地,实例数组的一个实例中的 defparam 语句可能不以数组的另一个实例的参数为目标。

defparam 分配/赋值右侧的表达式应该为常数表达式,仅涉及数字和参数的引用。引用

的参数(在 defparam 的右侧)应在与 defparam 语句相同的模块中声明。

defparam 语句对于覆盖分组在一个模块中所有参数的分配/赋值非常有用。在单个参数有多个 defparam 的情况下,该参数取源文本中遇到的最后一个 defparam 语句的值。在多个源文件中遇到 defparam 时,例如通过库搜索找到 defparam,参数从中获取其值的 defparam 是未定义的。

【例 5.144】 Verilog HDL 将无法定义参数使用的值的例子,如代码清单 5-56 所示。

代码清单 5-56 defparam 修改参数值的 Verilog HDL 描述

```
module top;
reg clk;
reg [0:4] in1;
reg [0:9] in2;
wire [0:4] o1;
wire [0:9] o2;
vdff m1 (o1, in1, clk);
vdff m2 (o2, in2, clk);
endmodule

module vdff (out, in, clk);
parameter size = 1, delay = 1;
input [0:size - 1] in;
input clk;
output [0:size - 1] out;
reg [0:size - 1] out;
always @(posedge clk)
    # delay out = in;
endmodule

module annotate;
defparam
    top.m1.size = 5,
    top.m1.delay = 10,
    top.m2.size = 10,
    top.m2.delay = 20;
endmodule
```

在该例子中,模块 annotate 有 defparam 语句,其覆盖 top 模块中例化 m1 和 m2 模块的 size 和 delay 参数值。显然,在该例子中,模块 top 和 annotate 都将看作顶层模块。

2. 模块实例参数值分配

为模块实例内的参数赋值的另一种方法是使用按顺序列表或按名字为模块实例参数分配值。这两种类型的模块实例参数分配不能混用。特别是模块实例的参数分配应完全按顺序或完全按名字。

表面上,按顺序列表分配的模块实例参数值类似给门实例分配延迟值,按名字分配类似按名字连接模块端口。它向模块定义中指定的任何参数提供模块特定实例的值。

在命名块、任务或函数中声明的参数只能使用 defparam 语句直接重新定义。但是,如果参数值取决于第二个参数,则重新定义第二个参数也将更新第一个参数的值。

(1) 通过列表顺序分配参数值。

采用这种方式分配参数,其分配的顺序应该和模块内声明参数的顺序一致。当使用该方法时,没有必要为模块内的所有参数分配值。然而,不能跳过参数。因此,给模块内声明参数的子集分配值的时候,组成这个子集的参数声明应先于其余参数的声明。另一种为所有参数分配值的可选方法是,对那些不需要新值的参数使用默认的值(即与模块定义内的参数声明中所分配的值相同)。

【例 5.145】 按顺序分配参数值 Verilog HDL 描述的例子，如代码清单 5-57 所示。

代码清单 5-57 按顺序分配参数值的 Verilog HDL 描述

```
module tb1;
wire [9:0] out_a, out_d;
wire [4:0] out_b, out_c;
reg [9:0] in_a, in_d;
reg [4:0] in_b, in_c;
reg clk;
//测试平台时钟和激励生成代码...
//通过列表顺序对带有参数值分配的四个 vdff 例化
//mod_a 有新的参数值,size = 10 且 delay = 15
//mod_b 为默认的参数值(size = 5, delay = 1)
//mod_c 有默认的参数值 size = 5 和新的参数值 delay = 12
//为了改变参数的值,也需要说明默认宽度值
//mod_d 有新的参数值 size = 10,延迟为默认值
  vdff #(10,15) mod_a (.out(out_a), .in(in_a), .clk(clk));
  vdff mod_b (.out(out_b), .in(in_b), .clk(clk));
  vdff #( 5,12) mod_c (.out(out_c), .in(in_c), .clk(clk));
  vdff #(10) mod_d (.out(out_d), .in(in_d), .clk(clk));
endmodule

module vdff (out, in, clk);
parameter size = 5, delay = 1;
output [size - 1:0] out;
input [size - 1:0] in;
input clk;
reg [size - 1:0] out;
always @(posedge clk)
  #delay out = in;
endmodule
```

不能覆盖本地参数值。因此，不能将其作为覆盖参数值列表的一部分。

【例 5.146】 对于本地参数处理 Verilog HDL 描述的例子，如代码清单 5-58 所示。

代码清单 5-58 本地参数处理的 Verilog HDL 描述

```
module my_mem (addr, data);
parameter addr_width = 16;
localparam mem_size = 1 << addr_width;
parameter data_width = 8;
...
endmodule
module top;
...
my_mem #(12, 16) m(addr,data);
endmodule
```

在本例中，addr_width 的值分配了 12，data_width 的值分配了 16。由于列表的顺序，不能显式为 mem_size 分配值，由于声明表达式，mem_size 的值为 4096。

(2) 通过名字分配参数值。

通过名字分配参数是将参数的名字和它新的值显式地进行连接。参数的名为被例化模块内所指定参数的名字。当使用这种方法时，不需要给所有参数分配值，只需指定需要分配新值的参数。

【例 5.147】 通过名字分配部分参数值 Verilog HDL 描述的例子，如代码清单 5-59 所示。

代码清单 5-59 通过名字分配部分参数值的 Verilog HDL 描述

```
module tb2;
wire [9:0] out_a, out_d;
```

```
wire [4:0] out_b, out_c;
reg [9:0] in_a, in_d;
reg [4:0] in_b, in_c;
reg clk;
//测试平台时钟和激励生成代码...
//通过名字分配包含参数值的四个例化 vdff
//mod_a 有新的参数 size = 10 和 delay = 15
//mod_b 有默认的参数值 (size = 5, delay = 1)
//mod_c 有默认的参数值 size = 5 和新的参数值 delay = 12
//mod_d 有一个新的参数值 size = 10,延迟保持它的默认参数值
  vdff #(.size(10),.delay(15)) mod_a (.out(out_a),.in(in_a),.clk(clk));
  vdff mod_b (.out(out_b),.in(in_b),.clk(clk));
  vdff #(.delay(12)) mod_c (.out(out_c),.in(in_c),.clk(clk));
  vdff #(.delay( ),.size(10) ) mod_d (.out(out_d),.in(in_d),.clk(clk));
endmodule

module vdff (out, in, clk);
parameter size = 5, delay = 1;
output [size - 1:0] out;
input [size - 1:0] in;
input clk;
reg [size - 1:0] out;
always @(posedge clk)
  #delay out = in;
endmodule
```

在相同的顶层模块中,当例化模块时,使用不同的重新定义的参数类型是合法的。

【例 5.148】 混合使用不同参数定义类型 Verilog HDL 描述的例子。

```
module tb3;
//声明和代码
//使用位置参数例化和名字参数例化的混合声明是合法的
  vdff #(10, 15) mod_a (.out(out_a), .in(in_a), .clk(clk));
  vdff mod_b (.out(out_b), .in(in_b), .clk(clk));
  vdff #(.delay(12)) mod_c (.out(out_c), .in(in_c), .clk(clk));
endmodule
```

不允许在一个例化模块中,同时使用两种混合参数分配的方法,如

```
vdff #(10, .delay(15)) mod_a (.out(out_a), .in(in_a), .clk(clk));
```

上述描述是非法的。


视频讲解

5.9.3 端口

端口提供了由模块和原语组成的硬件描述的方法,如模块 A 可以例化模块 B,通过适当的端口连接到模块 A。这些端口的名字可不同于在模块 B 内所指定的内部网络和变量的名字。

1. 端口列表

在每个模块声明顶部端口列表中,每个端口的端口引用可以是:

(1) 一个简单的标识符或者转义标识符;

(2) 在模块内声明向量的位选择;

(3) 在模块内声明向量的部分选择;

(4) 上面形式的并置/连接。

端口表达式是可选择的。由于可以定义不连接到模块内部任何内容的端口,所以一旦定义了端口,不能使用相同的名字定义其他端口。只有端口表达式的第一种类型端口模块是隐含端口。第二种类型的端口是显式端口。这明确指定了用于按端口名字连接模块实例端口的

端口标识符。

2. 端口声明

如果端口声明中包含一个网络或者变量类型,则将端口看作完全的声明。如果在一个变量或者网络数据类型声明中再次声明端口,则会出现错误。由于这个原因,端口的其他内容也应该在这样一个端口声明中进行声明,包括有符号和范围定义(如果需要的话)。

如果端口声明中不包含一个网络或者变量类型,则可以在变量或者网络数据类型声明中再次声明端口。如果将网络或者变量声明为一个向量,在一个端口中的两个声明中应该保持一致。一旦在端口定义中使用了该名字,则不允许在其他端口声明或者在数据类型声明中再次进行声明。

实现可能限制一个模块定义中端口的最大数目,但至少为 256。

有符号属性可以附加到端口声明或者对应的网络或 reg 声明,也可以添加到两者。如果端口或网络/reg 已经被声明为有符号的,则另一个也应该当作有符号的。

隐含网络应该看作无符号的。连接到没有明确网络声明的端口的网络应看作无符号的,除非端口已声明为有符号的。

3. 通过列表顺序连接模块例化

通过列表顺序连接模块例化是一种连接端口的方法,即在例化模块的时候端口连接的顺序和定义模块内端口的顺序相一致。

【例 5.149】 通过列表顺序连接端口 Verilog HDL 描述的例子,如代码清单 5-60 所示。

代码清单 5-60 通过列表顺序连接端口的 Verilog HDL 描述

```
module topmod;
    wire [4:0] v;
    wire a,b,c,w;
    modB b1 (v[0], v[3], w, v[4]);
endmodule

module modB (wa, wb, c, d);
inout wa, wb;
input c, d;

tranif1 g1 (wa, wb, cinvert);
not #(2, 6) n1 (cinvert, int);
and #(6, 5) g2 (int, c, d);
endmodule
```

该例中实现了下面的端口连接。

(1) 模块 modB 内定义的端口 wa 连接到 topmod 模块的位选择 v[0]。

(2) 端口 wb 连接到 v[3]。

(3) 端口 c 连接到 w。

(4) 端口 d 连接到 v[4]。

在仿真时,modB 的实例 b1,首先激活 and 门 g2,在 int 上产生一个值;这个值触发 not 门 n1,在 cinvert 上产生输出,然后激活 tranif1 门 g1。

4. 通过名字连接模块例化

另一种将模块端口和例化模块端口连接的方法是通过名字。下面将给出通过名字连接模块例化的几种方式。

【例 5.150】 通过名字连接端口的 Verilog HDL 描述的例子 1。

```
ALPHA instance1 (.Out(topB),.In1(topA),.In2());
```

在该例子中，例化模块将其信号 topA 和 topB 连接到模块 ALPHA 所定义的 In1 和 Out。ALPHA 提供的端口中，至少没有使用其中一个端口，该端口的名为 In2。该例子中，没有提到例化中未使用的其他端口。

【例 5.151】 通过名字连接端口 Verilog HDL 描述的例子 2，如代码清单 5-61 所示。

代码清单 5-61 通过名字连接端口的 Verilog HDL 描述

```
module topmod;
    wire [4:0] v;
    wire a,b,c,w;
    modB b1 (.wb(v[3]),.wa(v[0]),.d(v[4]),.c(w));
endmodule
module modB(wa, wb, c, d);
    inout wa, wb;
    input c, d;
    tranif1 g1(wa, wb, cinvert);
    not#(6, 2) n1(cinvert, int);
    and #(5, 6) g2(int, c, d);
endmodule
```

因为这些连接是按名字建立的，所以它们出现的顺序并不重要。

【例 5.152】 多个模块实例端口非法连接 Verilog HDL 描述的例子。

```
module test;
a ia (.i (a), .i (b),                    //非法连接输入端口两次
.o (c), .o (d),                          //非法连接输出端口两次
.e (e), .e (f));                         //非法连接输入输出端口两次
endmodule
```

5. 端口连接中的实数

real 数据类型不能直接连接到端口上，而应该采用间接的方式进行连接。系统函数 $realtobits 和 $bitstoreal 用于在模块端口之间传递位模式。

【例 5.153】 通过系统函数传递实数 Verilog HDL 描述的例子。

```
module driver (net_r);
  output net_r;
  real r;
  wire [64:1] net_r =  $realtobits(r);
endmodule

module receiver (net_r);
  input net_r;
  wire [64:1] net_r;
  real r;
  initial assign r =  $bitstoreal(net_r);
endmodule
```

6. 连接不同的端口

一个模块的端口可看作两个条项（例如网络、reg 和表达式）之间的链路或者连接，即内部到模块实例以及外部到模块实例。

下面的端口连接规则给出了通过端口接收值的条目（内部条目用于输入，外部条目用于输出）应该为结构网络表达式。

一个声明为 input（output），但是用于 output（input）或 inout，应该强制为 inout。如果没有强制到 inout，将产生警告信息。

7. 端口连接规则

1）规则一

input 或者 inout 端口是网络类型。

2）规则二

每个端口连接应为源到接收的连续分配，其中一个连接条目应为信号源，另一个应为信号接收。分配应为输入或输出端口从源到接收的连续分配。该分配是用于 input 端口的非强度降低的晶体管连接。在一个分配中，只有接收是网络或者结构化的网络表达式。

结构化的网络表达式是一个端口表达式，其操作数可以是：

（1）标量网络；

（2）向量网络；

（3）向量网络的常数位选择；

（4）向量网络的部分选择；

（5）结构化网络表达式的并置/连接。

下面的外部条目不能连接到模块的 output 或者 inout 端口。

（1）变量。

（2）不同于下面的其他表达式：①标量网络；②向量网络；③向量网络的常数位选择；④向量网络的部分选择；⑤上述表达式的并置/连接。

3）规则三

如果一个端口两侧的任何一个网络类型是 uwire，如果没有将网络合并为一个网络，则会出现警告信息。

8. 不同端口连接导致的网络类型

当不同的网络类型通过模块端口连接时，端口两侧的网络可以采用同一种类型。所得的类型如表 5.27 所示。

表 5.27 不同网络类型连接后最后类型的确定

内部网络	外部网络								
	wire,tri	wand,triand	wor,trior	trireg	tri0	tri1	uwire	supply0	supply1
wire,tri	ext	ext	ext	ext	ext	ext	ext	ext	ext
wand,triand	int	ext	ext warn	ext warn	ext warn	ext warn	ext warn	ext	ext
wor,trior	int	ext warn	ext	ext warn	ext warn	ext warn	ext warn	ext	ext
trireg	int	ext warn	ext warn	ext	ext	ext	ext warn	ext	ext
tri0	int	ext warn	ext warn	int	ext	ext warn	ext warn	ext	ext
tri1	int	ext warn	ext warn	int	ext warn	ext	ext warn	ext	ext
uwire	int	int warm	int warm	int warn	int warn	int warn	ext	ext	ext
supply0	int	int	int	int	int	int	int	ext	ext warn
supply1	int	int	int	int	int	int	int	ext warn	ext

表 5.27 中，外部网络表示模块例化中指定的网络，内部网络表示模块定义中指定的网络。ext 为应使用的外部网络类型，int 为应使用的内部网络类型，warn 为应发出警告。

所使用类型的网络称为主导网络。类型被改变的网络称为被支配的网络。允许将主导网络和被支配的网络合并为单个网络，其类型为主导网络的类型。最终的网络称为“仿真”网络，

而被支配的网络称为“倒塌”的网络。

“仿真”网络应该采用主导网络规定的延迟。如果主导网络类型为 trireg，则为 trireg 网络指定的任何强度值应适用于“仿真”网络。

1）网络类型解析规则

当一个端口连接的两个网络是不同的网络类型时，生成的单个网络可以被指定为以下之一。如果两个网络中的一个处于主导网络，则为主导网络类型，或模块外部的网络类型；当不存在主导网络类型时，应使用外部网络类型。

2）网络类型表

表 5.27 显示了由网络类型解析规则决定的网络类型。仿真网络应采用表中规定的网络类型和该网络规定的延迟。如果选择的仿真网络是 trireg，则为 trireg 网络指定的任何强度值都适用于仿真网络。

9. 通过端口连接有符号值

符号属性不应跨越层次。为了使有符号类型跨越层次，必须在不同层次结构级别的对象声明中使用 signed 关键字。端口上的任何表达式应看作分配/赋值中的任何其他表达式。它应该有类型、宽度和评估，并使用与分配/赋值相同的规则将结果值分配给端口另一侧的对象。


视频讲解

5.9.4 生成结构

在一个模型中，Verilog HDL 用于有条件或者多次例化生成块。生成块是一个或者多个模块条目的集合。生成块不包含端口声明、参数声明、指定块或者 specparam 声明。在生成块中，允许包含其他生成结构。生成结构提供了通过参数值影响模型结构的能力。它允许更简单的描述包含重复结构的模块，并使递归模块例化成为可能。

Verilog HDL 提供了两种生成结构的类型。

(1) 循环生成结构，允许单个生成块多次例化到一个模型。

(2) 条件生成结构，包含 if-generate 或者 case-generate 结构，从一堆可以选择的生成块中例化出最多一个生成块。

术语生成方案是指确定例化所生成的模块或者生成多个模块的方法，它包含出现在一个生成结构中的条件表达式、case 替换和循环控制语句。

在对模型进行详细说明(elaboration，计算机综合过程的一部分)的过程中，评估生成方案。当分析完 HDL 后，仿真之前进行详细的说明。详细说明涉及展开模块例化，计算参数值，解析层次的名字，建立网络连接和准备用于仿真的模型。尽管生成方案使用和行为语句类似的语法，但是在仿真的时候，并不执行它们，因此，在生成方案中的所有表达式必须是常数表达式。

对生成结构的详细描述产生零个或者多个生成块。在某些方面，对生成块的例化类似于模块的例化。它创建了一个新的层次结构，并使块内的对象、行为结构和模块实例得以存在。

关键字 generate 和 endgenerate 可以在模块中用于定义生成区域。生成区域使模块描述中可能出现生成结构的文本跨度。可以选择使用生成区域。使用生成区域时，模块中没有语义差异。解析器可以选择识别生成区域，以针对错误使用生成构造关键字生成不同的错误消息。生成区域不能嵌套，它们只能直接出现在模块中。如果使用 generate 关键字，则应该使用 endgenerate 关键字进行匹配。

1. 循环生成结构

一个循环生成结构允许使用类似 for 循环语句的语法多次实例化生成块。

在循环生成方案中，使用循环索引变量之前，应在 genvar 声明中声明循环索引变量。

genvar 在详细说明过程中用作整数，用于评估循环的次数并创建生成块的实例，但在仿真时不存在。不可以在循环生成方案外的其他地方使用 genvar。

循环生成方案中的初始化和迭代分配都应分配给同一个 genvar。初始化分配/赋值不应引用右侧的循环索引变量。

在循环生成结构的内部，有一个隐含的 localparam 声明，这是一个整数参数，它和循环索引变量有相同的名字和类型。在生成模块内，该参数的值是当前详细描述中索引变量的值。该参数可以用于生成块内的任何地方，其可以使用带有一个整数值的普通参数，它可以用层次结构名字引用。

由于这个隐含的 localparam 和 genvar 有相同的名字，任何对循环生成块内名字的引用都是对 localparam 的引用，而不是对 genvar 的引用，结果是不可能在两个嵌套的循环生成结构中使用相同的 genvar。

可以命名或者不命名一个生成结构，它们只有一个条目，而不需要 begin/end 关键字。即使没有 begin/end 关键字，它仍然为一个生成块，与所有生成块一样，在例化时，它包括一个单独的范围和一个新的层次结构级别。

如果已命名了生成块，则它是生成块实例数组的声明。此数组中的索引值是 genvar 在详细描述过程中假定的值。这可以是一个稀疏数组，因为 genvar 值不必形成连续的整数范围。即使循环生成方案没有生成块的实例，也会认为已声明数组。如果未命名生成块，则不能使用层次结构名字以外的层次结构名字引用其中的声明。

如果生成块实例数组的名字与任何其他声明（包括任何其他生成块实例数组）冲突，则应为错误。如果在循环生成方案的评估过程中重复 genvar 值，则应为错误。如果在循环生成方案的评估过程中 genvar 的任何位被设置为“x”或“z”，将是一个错误。

【例 5.154】 合法和非法循环生成结构 Verilog HDL 描述的例子，如代码清单 5-62 所示。

代码清单 5-62 生成循环结构的不同 Verilog HDL 描述

```
module mod_a;
genvar i;
//不要求"generate", "endgenerate"
for (i = 0; i < 5; i = i + 1) begin:a
    for (i = 0; i < 5; i = i + 1) begin:b
    …                                  //错误——使用"i"作为两个嵌套生成循环的索引
    end
end
endmodule

module mod_b;
genvar i;
reg a;
for (i = 1; i < 0; i = i + 1) begin: a
  …                                    //错误——"a"和 reg 类型"a"冲突
end
endmodule

module mod_c;
genvar i;
for (i = 1; i < 5; i = i + 1) begin: a
…
end
for (i = 10; i < 15; i = i + 1) begin: a
  …                                    //错误——"a"和前面的名字冲突
end
endmodule
```

【例 5.155】 实现格雷码到二进制码转换 Verilog HDL 描述的例子，如代码清单 5-63 所示。

代码清单 5-63 格雷码到二进制码转换的 Verilog HDL 描述

```
module gray2bin1 (bin, gray);
  parameter SIZE = 8;                    //该模块参数化
  output [SIZE-1:0] bin;
  input [SIZE-1:0] gray;
  genvar i;
  generate
    for (i=0; i<SIZE; i=i+1) begin:bit
      assign bin[i] = ^gray[SIZE-1:i];
    end
endgenerate
endmodule
```

下面两个例子中的模型是使用循环生成 Verilog 门原语的纹波加法器的参数化模块。

【例 5.156】 循环生成逐位进位加法器 Verilog HDL 描述的例子 1，如代码清单 5-64 所示。

代码清单 5-64 循环生成逐位进位加法器的 Verilog HDL 描述

```
module addergen1 (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;
  wire [SIZE-1:0] t [1:3];
  genvar i;
  assign c[0] = ci;
        //层次化的门例化名字:
        //异或门: bit[0].g1 bit[1].g1 bit[2].g1 bit[3].g1
        //bit[0].g2 bit[1].g2 bit[2].g2 bit[3].g2
        //与门: bit[0].g3 bit[1].g3 bit[2].g3 bit[3].g3
        //bit[0].g4 bit[1].g4 bit[2].g4 bit[3].g4
        //或门: bit[0].g5 bit[1].g5 bit[2].g5 bit[3].g5
        //使用多维网络进行连接 t[1][3:0] t[2][3:0] t[3][3:0](共 12 个网络)
      for(i=0; i<SIZE; i=i+1) begin:bit
        xor g1 ( t[1][i], a[i], b[i]);
        xor g2 ( sum[i], t[1][i], c[i]);
        and g3 ( t[2][i], a[i], b[i]);
        and g4 ( t[3][i], t[1][i], c[i]);
        or g5 ( c[i+1], t[2][i], t[3][i]);
      end
    assign co = c[SIZE];
endmodule
```

该例子使用生成循环外部的二维网络声明来建立门原语之间的连接。

【例 5.157】 循环生成加法器 Verilog HDL 描述的例子 2，如代码清单 5-65 所示。

代码清单 5-65 循环生成加法器的 Verilog HDL 描述

```
module addergen1 (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;
  //层次化的门例化名字:
  //异或门: bit[0].g1 bit[1].g1 bit[2].g1 bit[3].g1
```

```
  //bit[0].g2 bit[1].g2 bit[2].g2 bit[3].g2
  //与门: bit[0].g3 bit[1].g3 bit[2].g3 bit[3].g3
  //bit[0].g4 bit[1].g4 bit[2].g4 bit[3].g4
  //或门: bit[0].g5 bit[1].g5 bit[2].g5 bit[3].g5
  //使用下面的网络名字连接
  //bit[0].t1 bit[1].t1 bit[2].t1 bit[3].t1
  //bit[0].t2 bit[1].t2 bit[2].t2 bit[3].t2
  //bit[0].t3 bit[1].t3 bit[2].t3 bit[3].t3
    for(i = 0; i < SIZE; i = i + 1) begin:bit
        wire t1, t2, t3;
        xor g1 ( t1, a[i], b[i]);
        xor g2 ( sum[i], t1, c[i]);
        and g3 ( t2, a[i], b[i]);
        and g4 ( t3, t1, c[i]);
        or g5 ( c[i + 1], t2, t3);
    end
  assign co = c[SIZE];
endmodule
```

该例子使用生成循环内的网络声明来为循环的每个迭代生成连接门原语所需的连线。

【例 5.158】 多层生成模块 Verilog HDL 描述的例子,如代码清单 5-66 所示。

代码清单 5-66 多层生成模块的 Verilog HDL 描述

```
parameter SIZE = 2;
  genvar i, j, k, m;
  generate
    for (i = 0; i < SIZE; i = i + 1) begin:B1                    //范围 B1[i]
      M1 N1();                                                   //例化 B1[i].N1
        for (j = 0; j < SIZE; j = j + 1) begin:B2                //范围 B1[i].B2[j]
          M2 N2();                                               //例化 B1[i].B2[j].N2
              for (k = 0; k < SIZE; k = k + 1) begin:B3          //范围 B1[i].B2[j].B3[k]
                  M3 N3();                                       //例化 B1[i].B2[j].B3[k].N3
              end
        end
        if (i > 0) begin:B4                                      //范围 B1[i].B4
          for (m = 0; m < SIZE; m = m + 1) begin:B5              //范围 B1[i].B4.B5[m]
              M4 N4();                                           //例化 B1[i].B4.B5[m].N4
          end
        end
  end
endgenerate
```

在该例子中显示了多级生成循环中的分层生成块实例名字。对于生成循环创建的每个块实例,通过在生成块标识符的末尾添加[genvar value]来索引循环的生成块标识符。这些名字可用于分层路径名字。

2. 条件生成结构

条件生成结构包含 if-generate 和 case-generate。在详细描述的过程中,基于给出的常数表达式,从一组备选的生成块中最多选择一个生成块。如果存在需要生成的块,则将其例化到模型中。

条件生成结构中的生成块可以是命名的或未命名的,并且它们可以仅由一个条目组成,而不需要使用 begin-end 关键字包围。即使没有 begin-end 关键字,它仍然是一个生成块,与所有生成块一样,在例化时,它包含一个单独的范围和一个新的层次结构级。

因为最多例化了一个备选生成块,所以在单个条件生成结构中允许存在多个同名块,不允许任何命名的生成块与任何其他条件或循环中的生成块具有相同的名字。

如果命名了为例化选择的生成块,那么这个名字声明了一个生成块实例,并且是它创建的作用域的名字。如果没有命名选择用于例化的生成块,它仍然会创建一个作用域,但不能使用层次结构名字以外的名字来引用层次结构其中内部的声明。

如果条件生成结构中的生成块仅包含一个自身为条件生成结构的条目，并且该条目未被begin-end关键字包围，则该生成块不会被看作单独的作用域。在该块中的生成结构称为直接嵌套。直接嵌套结构的生成块好像被看作属于外部结构。因此，它们可以与外部结构的生成块具有相同的名字，并且它们不能与包含外部结构的作用域中的任何声明（包含该作用域中其他生成结构中的其他生成块）具有相同的名字。允许在没有创建不必要的生成块层次结构的情况下表达复杂的条件生成方案。

最常见的方法是创建一个if-else-if生成方案，其中包含任意个数的else-if子句，所有这些子句都可以生成同名的块，因为只选择一个用于例化。允许在同一复杂生成方案中组合if-生成和case-生成结构。直接嵌套仅适用于嵌套在条件生成结构中的条件生成结构，它不以任何方式应用于循环生成结构。

【例5.159】 if-else生成结构Verilog HDL描述的例子，如代码清单5-67所示。

代码清单5-67　if-else生成结构的Verilog HDL描述

```
module test;
parameter p = 0, q = 0;
wire a, b, c;
//-------------------------------------------------------------
//代码或者生成u1.g1例化或者没有生成例化
//u1.g1例化下面的一个门{and, or, xor, xnor),根据条件
//{p,q} == {1,0}, {1,2}, {2,0}, {2,1}, {2,2}, {2, default}
//-------------------------------------------------------------
if (p == 1)
    if (q == 0)
        begin : u1                  //如果p==1和q==0, 则例化
            and g1(a, b, c);        //AND的层次名为test.u1.g1
        end
    else if (q == 2)
        begin : u1                  //如果p==1和q==2, 则例化
            or g1(a, b, c);         //OR的层次名为test.u1.g1
          end
    //添加"else"结束"(q == 2)"的描述
    else ;                          //如果p==1和q!=0或2, 则没有例化
else if (p == 2)
  case (q)
  0, 1, 2:
    begin : u1                      //如果p==2和q==0,1, 或者2,则例化
      xor g1(a, b, c);              //XOR层次名为test.u1.g1
    end
  default:
    begin : u1                      //如果p==2和q!=0,1, 或者2,则例化
      xnor g1(a, b, c);             //XNOR的层次名为test.u1.g1
    end
endcase
endmodule
```

在该例子中，生成结构将最多选择一个名为u1的生成块。该块中的门实例的分层名字将是test.u1.g1。当嵌套if-生成结构时，else总是属于最近的if结构。

注：与上面的例子一样，可以插入带有空生成块的else，以便后续else属于外部if结构。begin/end关键字也可以用来消除歧义，然而，这将违反直接嵌套的标准，并且将创建额外级别的生成层次结构。

条件生成结构使得模块可以包含自身的例化，循环生成结构也是如此，但使用条件生成更容易实现。通过正确使用参数可以终止生成的递归，从而生成合法的模型层次结构。确定了顶层模块的规则，包含自身实例的模块将不再是顶层模块。

【例 5.160】 一个参数化乘法器 Verilog HDL 描述的例子，如代码清单 5-68 所示。

代码清单 5-68 参数化乘法器的 Verilog HDL 描述

```
module multiplier(a,b,product);
parameter a_width = 8, b_width = 8;
localparam product_width = a_width + b_width;
    //不能通过 defparam 语句或者例化语句直接修改 #
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)) begin: mult
        CLA_multiplier #(a_width,b_width) u1(a, b, product);
        //例化一个 CLA 乘法器
    end
    else begin: mult
        WALLACE_multiplier #(a_width,b_width) u1(a, b, product);
        //例化一个 Wallace 树乘法器
    end
endgenerate
    //层次化的例化名为 mult.u1
endmodule
```

【例 5.161】 case 生成结构 Verilog HDL 描述的例子，如代码清单 5-69 所示。

代码清单 5-69 case 生成结构的 Verilog HDL 描述

```
generate
  case (WIDTH)
    1: begin: adder                //实现 1 比特加法器
        adder_1bit x1(co, sum, a, b, ci);
    end
    2: begin: adder                //实现 2 比特加法器
        adder_2bit x1(co, sum, a, b, ci);
    end
    default:
      begin: adder                 //其他超前进位加法器
        adder_cla #(WIDTH) x1(co, sum, a, b, ci);
      end
    endcase
//这个层次例化的名字是 adder.x1
endgenerate
```

【例 5.162】 for 循环生成结构 Verilog HDL 描述的例子，如代码清单 5-70 所示。

代码清单 5-70 for 循环生成结构的 Verilog HDL 描述

```
module dimm(addr, ba, rasx, casx, csx, wex, cke, clk, dqm, data, dev_id);
parameter [31:0] MEM_WIDTH = 16, MEM_SIZE = 8;     //in mbytes
input [10:0] addr;
input ba, rasx, casx, csx, wex, cke, clk;
input [7:0] dqm;
inout [63:0] data;
input [4:0] dev_id;
genvar i;
    case ({MEM_SIZE, MEM_WIDTH})
      {32'd8, 32'd16}:                              //8M×6 位宽
      begin: memory
        for (i = 0; i < 4; i = i + 1) begin:word
          sms_08b216t0 p(.clk(clk), .csb(csx), .cke(cke),.ba(ba),
                         .addr(addr), .rasb(rasx), .casb(casx),
                         .web(wex), .udqm(dqm[2 * i + 1]), .ldqm(dqm[2 * i]),
                         .dqi(data[15 + 16 * i:16 * i]), .dev_id(dev_id));
          //层次化例化名字是 memory.word[3].p,
```

```
            //memory.word[2].p, memory.word[1].p, memory.word[0].p,
            //和任务 memory.read_mem
        end
        task read_mem;
          input [31:0] address;
          output [63:0] data;
          begin                                          //在 sms 模块内调用 read_mem
              word[3].p.read_mem(address, data[63:48]);
              word[2].p.read_mem(address, data[47:32]);
              word[1].p.read_mem(address, data[31:16]);
              word[0].p.read_mem(address, data[15: 0]);
          end
        endtask
      end
      {32'd16, 32'd8}:                                   //16M x 8 位宽度
      begin: memory
        for (i = 0; i < 8; i = i + 1) begin:byte
            sms_16b208t0 p(.clk(clk), .csb(csx), .cke(cke),.ba(ba),
                           .addr(addr), .rasb(rasx), .casb(casx),
                           .web(wex), .dqm(dqm[i]),
                           .dqi(data[7 + 8 * i:8 * i]), .dev_id(dev_id));
            //层次化的例化名字 memory.byte[7].p,memory.byte[6].p, ... , memory.byte[1].p,
            //memory.byte[0].p
            //和任务 memory.read_mem
        end
        task read_mem;
          input [31:0] address;
          output [63:0] data;
          begin                                          //在 sms module 模块调用 read_mem
              byte[7].p.read_mem(address, data[63:56]);
              byte[6].p.read_mem(address, data[55:48]);
              byte[5].p.read_mem(address, data[47:40]);
              byte[4].p.read_mem(address, data[39:32]);
              byte[3].p.read_mem(address, data[31:24]);
              byte[2].p.read_mem(address, data[23:16]);
              byte[1].p.read_mem(address, data[15: 8]);
              byte[0].p.read_mem(address, data[7: 0]);
          end
        endtask
      end
        //其存储器情况
      endcase
endmodule
```

3. 用于未命名生成块的外部名字

虽然未命名的生成块没有可用于分层名字的名字，但它需要一个名字以使外部接口可以引用它。为此，将为每个未命名的块分配一个名字。

对于一个给定范围内的每个生成结构，都分配了一个数字。在该范围内，首先是以文字形式的结构，其数字是 1；对于该范围的每个随后生成结构其值递增 1；对于所有未命名的生成块，将其命名为 genblk<n>，n 为分配给该结构的数字，如果这个名字和明确声明的名字冲突，则在数字前一直加 0，直到没有冲突为止。

【例 5.163】 未命名生成块 Verilog HDL 描述的例子，如代码清单 5-71 所示。

代码清单 5-71 未命名生成块的 Verilog HDL 描述

```
module top;
    parameter genblk2 = 0;
    genvar i;
    //下面的生成块有隐含的名字 genblk1
    if (genblk2) reg a;                                  //top.genblk1.a
```

```
  else reg b;                                          //top.genblk1.b
  //下面的生成块有隐含的名字 genblk02,因为已经声明 genblk2 为标识符
  if (genblk2) reg a;                                  //top.genblk02.a
  else reg b;                                          //top.genblk02.b
  //下面的生成块有隐含的名字 genblk3,但是有明确的名字 g1
  for (i = 0; i < 1; i = i + 1) begin : g1             //块的名字
      //下面的生成块有隐含名字 genblk1
      //作为 g1 内第 1 个嵌套的范围
    if (1) reg a;                                      //top.g1[0].genblk1.a
  end
    //下面的生成块有隐含的名字 genblk4,由于它属于 top 内的第四个生成块
    //如果没有明确命名为 g1,前面的生成块命名为 genblk3
  for (i = 0; i < 1; i = i + 1)
    //下面的生成块有隐含名字 genblk1
    //作为 genblk4 内第一个嵌套的生成块
    if (1) reg a;                                      //top.genblk4[0].genblk1.a
    //下面的生成块有隐含的名字 genblk5
    if (1) reg a;                                      //top.genblk5.a
endmodule
```

5.9.5 层次化的名字


视频讲解

在 Verilog HDL 描述中,每个标识符应该有一个唯一的层次路径名字。模块层次和模块内所定义的条目,如任务和命名块定义了这些名字。名字的层次可看作一个树状结构。其中每个模块实例、生成块实例、任务、函数或命名的 begin-end 或 fork-join 块在树的特殊分支上定义了一个新的层次或范围。

设计描述包含一个或多个顶层模块。每个这样的模块构成了名字结构的顶层。此根模块或这些并行根模块构成设计描述或描述中的一个或多个层次结构。在任何模块内,每个模块实例(包含实例数组)、生成块实例、任务定义、函数定义,以及命名的 begin-end 或 fork-join 块都应定义层次结构的新分支。命名块以及任务和函数内的命名块应创建新分支。未命名的生成块是例外,它们创建仅在块内以及块实例化的任何层次结构内可见的分支。

在分层名字树中的每个节点应是与标识符相关的单独范围。特殊标识符在任何范围内最多只能声明一次。

任何命名的 Verilog 对象或分层名字引用,都可以通过并置/连接模块名字、模块实例名字、生成块、任务、函数或包含它的命名块,以其完整形式唯一引用。字符“.”应用于分隔分层中的每个名字,但嵌入分层名字引用中的转义标识符除外,其后是由空格和字符“.”组成的分隔符。任何对象的完整路径名字应从顶层(根)模块开始,此路径名可以在层次结构中的任何级或并行层次结构中使用。路径名中的第一个节点名也可以是层次结构的顶层,该层次结构从使用路径的级开始(这允许并使能向下引用条目)。自动任务和函数中声明的对象也是例外,不能通过层次结构名字引用访问。在未命名的生成块中声明的对象也是例外,它们只能由块内以及块例化的任何层次结构中的层次结构名字引用。

层次路径名字引用实例数组或循环生成块的名字后面可以紧跟方括号中的常量表达式。该表达式选择数组的特定实例,因此称为实例选择。表达式的计算结果应为数组的合法索引值之一。如果数组名字不是层次结构名字中的最后一个路径元素,则需要实例选择表达式。

【例 5.164】 模块实例和命名模块层次结构的 Verilog HDL 描述的例子,如代码清单 5-72 所示。

代码清单 5-72 模块实例和命名模块层次结构的 Verilog HDL 描述

```
module mod (in);
input in;
```

```
always @(posedge in) begin : keep
reg hold;
       hold = in;
end
endmodule

module cct (stim1, stim2);
input stim1, stim2;
//例化 mod
  mod amod(stim1), bmod(stim2);
endmodule

module wave;
reg stim1, stim2;
cct a(stim1, stim2);                                    //例化 cct
initial begin :wave1
     #100 fork :innerwave
             reg hold;
         join
     #150 begin
             stim1 = 0;
          end
  end
  endmodule
```

【例 5.165】 在层次化结构中修改值 Verilog HDL 描述的例子。

```
begin
    fork:mod_1
        reg x;
        mod_2.x = 1;
    join
    fork :mod_2
        reg x;
        mod_1.x = 0;
    join
end
```

5.9.6 向上名字引用

模块或者模块实例的名字对于识别模块以及它在层次中的位置已经足够。一个更低层次的模块，能引用层次中该模块上层模块内的条目。如果知道高层模块或者它的例化名字，则可以引用它的名字。对于任务、函数、命名的块和生成块，Verilog HDL 将检查模块内的名字，直到找到名字或者到达了层次的根部。它只能在更高的封闭模块中搜索名字，而不是实例。

向上名字引用的格式如下：

```
scope_name.item_name
```

其中，scope_name 为模块实例名字或生成块的名字。

【例 5.166】 向上名字引用 Verilog HDL 描述的例子。

```
module a;
integer i;
  b a_b1();
endmodule

module b;
integer i;
  c b_c1(), b_c2();
initial                                                 //向下的路径引用,i 的两个副本
```

```
    #10 b_c1.i = 2;                          //a.a_b1.b_c1.i, d.d_b1.b_c1.i
endmodule

module c;
integer i;
initial begin                                //i的本地名字应用的四个副本
    i = 1;                                   //a.a_b1.b_c1.i, a.a_b1.b_c2.i,
                                             //d.d_b1.b_c1.i, d.d_b1.b_c2.i
    b.i = 1;                                 //向上的路径引用,i的两个副本
                                             //a.a_b1.i, d.d_b1.i
end
endmodule

module d;
integer i;
    b d_b1();
initial begin                                //i的每个副本的全路径名字引用
a.i = 1;
d.i = 5;
a.a_b1.i = 2;
d.d_b1.i = 6;
a.a_b1.b_c1.i = 3;
d.d_b1.b_c1.i = 7;
a.a_b1.b_c2.i = 4;
d.d_b1.b_c2.i = 8;
end
endmodule
```

在该例子中,有四个模块a、b、c和d,每个模块都包含一个整数i。在模型层次结构这一段中的最高层模块是a和d。此处有两个模块b的副本,因为模块a和d例化了b。此处有c.i的四个副本,因为b的两个副本分别实现例化了c两次。

5.9.7 范围规则

在Verilog HDL中,下面的元素定义了一个新的范围,包括模块、任务、函数、命名块和生成块。

标识符只能用于声明范围内的一个条目。这个规则意味着声明两个或多个同名变量,或将任务命名为同一模块内的变量,或为门实例指定与其输出连接的网络名字相同的名字,都是非法的。对于生成块,无论生成块是否例化,此规则都适用。对于条件生成结构中的生成块,有一个例外。

如果在任务、函数、命名块或生成块内直接引用标识符(无层次路径),则应在任务、函数、命名块或本地生成块内,或在包含任务、函数和命名块的名字树的同一分支中较高的模块、任务、函数或命名块内,或生成块中声明。如果在本地声明,则应该使用本地条目;如果没有,则继续向上搜索,直到找到该名字的条目或遇到模块边界。如果条目是变量,则应在模块边界处停止;如果条目是任务、函数、命名块或生成块,它将继续搜索更高级别的模块,直到找到为止。这一事实意味着任务和函数可以按名字使用和修改包含模块中的变量,而无须通过其端口。

如果标识符用层次结构名字引用,则路径可以以模块名字、实例名字、任务、函数、命名块或命名生成块开头。应首先在当前级别搜索名字,然后在更高级别模块中搜索名字,直到找到为止。因为可以同时使用模块名字和实例名字,所以如果有与实例名字相同的模块,则优先使用实例名字。

【例 5.167】 在图 5.14 中，每个矩形表示一个局部范围。可用于向上搜索的范围向外扩展到以模块 A 的边界为外部边界的包含的所有矩形。因此，块 G 可以直接引用 F、E 和 A 中的标识符，不能直接引用 H、B、C 和 D 中的标识符。

【例 5.168】 通过名字访问变量 Verilog HDL 描述的例子。

```
task t;
reg s;
begin : b
    reg r;
    t.b.r = 0;                                    //这三行访问相同的变量 r
    b.r = 0;
    r = 0;
    t.s = 0;                                      //这两行访问相同的变量 s
    s = 0;
end
endtask
```

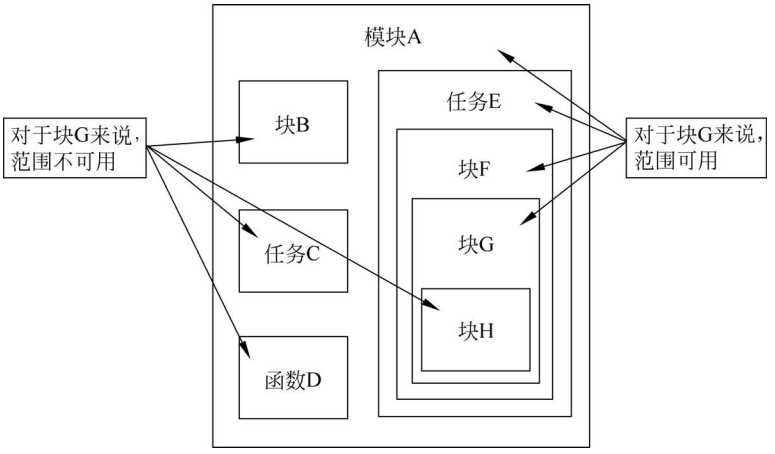

图 5.14 标识符的范围


视频讲解

5.9.8 设计实例六：*N* 位串行进位加法器的设计与实现

本节将使用循环生成语句生成 *N* 位串行进位加法器结构。设计文件如代码清单 5-73 所示，仿真文件如代码清单 5-74 所示。

代码清单 5-73 top. v 文件

```
/ ********************* full_adder 模块描述了一位全加器的功能 ***************** /
module full_adder(                          // module 关键字定义模块 full_adder
input a,                                    // input 关键字定义输入端口 a
input b,                                    // input 关键字定义输入端口 b
input ci,                                   // input 关键字定义输入端口 ci
output reg sum,                             // output 和 reg 关键字定义输出端口 sum(和)
output reg co                               // output 和 reg 关键字定义输出端口 co(进位)
);
always @( * )                               // always 关键字定义过程语句，* 为隐含敏感信号
begin                                       // begin 关键字标识过程语句的开始，类似 C 语言的"{"
case ({a,b,ci})                             // case 关键字定义分支语句，条件 a,b,ci 并置
 3'b000 : begin sum = 1'b0; co = 1'b0; end  // 条件取值为"000",sum(和) = 0,co(进位) = 0
 3'b001 : begin sum = 1'b1; co = 1'b0; end  // 条件取值为"001",sum(和) = 1,co(进位) = 0
 3'b010 : begin sum = 1'b1; co = 1'b0; end  // 条件取值为"010",sum(和) = 1,co(进位) = 0
 3'b011 : begin sum = 1'b0; co = 1'b1; end  // 条件取值为"011",sum(和) = 0,co(进位) = 1
 3'b100 : begin sum = 1'b1; co = 1'b0; end  // 条件取值为"100",sum(和) = 1,co(进位) = 0
 3'b101 : begin sum = 1'b0; co = 1'b1; end  // 条件取值为"101",sum(和) = 0,co(进位) = 1
 3'b110 : begin sum = 1'b0; co = 1'b1; end  // 条件取值为"110",sum(和) = 0,co(进位) = 1
```

```
 3'b111 : begin sum = 1'b1; co = 1'b1; end   // 条件取值为"111",sum(和) = 1,co(进位) = 1
 default : ;                                 // 其他情况
endcase                                      // endcase 标识分支语句的结束
end                                          // end 标识过程语句的结束
endmodule                                    // endmodule 标识模块 full_adder 的结束

/ ************** top 模块描述了一个宽度为 N 位的串行进位的加法器的功能 ************ /
module top #(parameter N = 8)(               // module 关键字定义模块 top,parameter 定义参数 N
input [N - 1:0] a,                           // input 关键字定义输入端口 a,宽度为 N
input [N - 1:0] b,                           // input 关键字定义输入端口 b,宽度为 N
input ci,                                    // input 关键字定义输入端口 ci
output [N - 1:0] sum,                        // output 关键字定义输出端口 sum,宽度为 N
output co                                    // output 关键字定义输出端口 co
);
assign co = c[N];                            // 将网络 C[N]的值作为加法器的进位输出
assign c[0] = ci;                            // 将最低位的进位输入 ci 分配/赋值给网络 c[0]
wire [N:0] c;                                // wire 关键字定义网络 c,宽度为 N
genvar i;                                    // genvar 关键字定义生成变量 i

generate                                     // generate 关键字定义生成结构
for(i = 0;i < N;i = i + 1)                   // for 关键字定义了循环生成结构
begin: adder                                 // begin 关键字标识循环结构的开始,adder 为块名字
full_adder Inst_full_adder                   // 调用元件 full_adder,将其例化为 Inst_full_adder
(a[i],b[i],c[i],sum[i],c[i + 1]);            // 端口的位置关联方法
end                                          // end 关键字标识循环结构的结束
endgenerate                                  // endgenerate 关键字标识生成结构的结束
endmodule                                    // endmodule 关键字标识模块 top 的结束
```

在高云云源软件中打开 RTL 级电路结构,如图 5.15 所示。

注：*在配套资源\eda_verilog\example_5_14 目录下,用高云云源软件打开 exaple_5_14.gprj。*

代码清单 5-74　test.v 文件

```
`timescale 1ns / 1ps                         // 预编译指令 timescale 定义时间标度、精度/分辨率
module test;                                 // module 关键字定义模块 test
parameter N = 8;                             // parameter 关键字定义参数 N,值为 8
reg [N - 1:0] x;                             // reg 关键字定义 reg 类型变量 x,宽度为 N
reg [N - 1:0] y;                             // reg 关键字定义 reg 类型变量 y,宽度为 N
reg ci;                                      // reg 关键字定义 reg 类型变量 ci
wire [N - 1: 0] sum;                         // wire 关键字定义网络 sum,宽度为 N
wire co;                                     // wire 关键字定义网络 co
top Inst_top(                                // 元件例化/调用语句,将模块 top 例化为 Inst_top
 .a(x),                                      // 模块 top 的端口 a 连接到 test 模块的变量 x
 .b(y),                                      // 模块 top 的端口 b 连接到 test 模块的变量 y
 .ci(ci),                                    // 模块 top 的端口 ci 连接到 test 模块的变量 ci
 .sum(sum),                                  // 模块 top 的端口 sum 连接到 test 模块的网络 sum
 .co(co)                                     // 模块 top 的端口 co 连接到 test 模块的网络 co
);
initial                                      // initial 关键字定义初始化部分
begin                                        // begin 关键字标识初始化部分的开始,类似 C 语言的"{"
ci = 1'b0;                                   // 变量 ci 的初始值为 0
end                                          // end 关键字表标识初始化部分的结束,类似 C 语言的"}"
initial                                      // initial 关键字定义初始化部分
begin                                        // begin 关键字标识初始化部分的开始,类似 C 语言的"{"
x = 56;                                      // 变量 x 分配/赋值 56
y = 67;                                      // 变量 y 分配/赋值 67
#100;                                        // 维持 100ns
$display("%d + %d = %d, co = %d", x,y,sum,co); // 打印 x + y 的结果 sum,以及进位 co 的值
x = 100;                                     // 变量 x 分配/赋值 100
y = 90;                                      // 变量 y 分配/赋值 90
#100;                                        // 维持 100ns
$display("%d + %d = %d, co = %d", x,y,sum,co); // 打印 x + y 的结果 sum,以及进位 co 的值
```

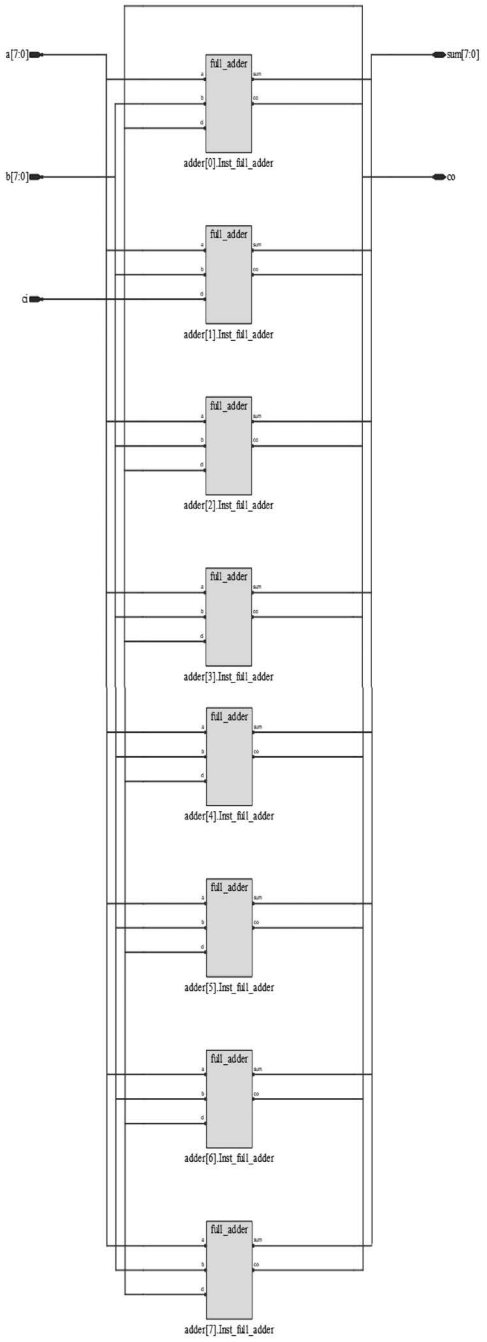

图 5.15　RTL 级电路结构

```
x = 120;                              // 变量 x 分配/赋值 120
y = 200;                              // 变量 y 分配/赋值 200
#100;                                 // 维持 100ns
$display("%d + %d = %d, co = %d", x,y,sum,co); // 打印 x + y 的结果 sum,以及进位 co 的值
x = 91;                               // 变量 x 分配/赋值 91
y = 138;                              // 变量 y 分配/赋值 138
#100;                                 // 维持 100ns
$display("%d + %d = %d, co = %d", x,y,sum,co); // 打印 x + y 的结果 sum,以及进位 co 的值
end                                   // end 关键字表标识初始化部分的结束,类似 C 语言的"}"
endmodule                             // endmodule 关键字标识 test 模块的结束
```

在 ModelSim 软件中对该设计执行综合后仿真，在 Transcript 窗口中打印的信息如图 5.16 所示。

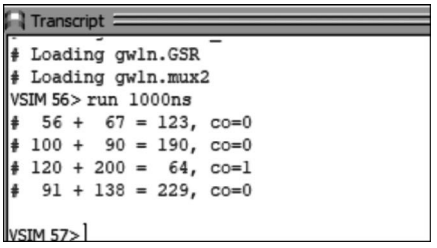

图 5.16　在 Transcript 窗口中打印的信息

注：在配套资源\eda_verilog\example_5_15 目录下，用 ModelSim SE-64 10.4c 打开 postsynth_sim.mpf。

5.10　系统任务和函数

本节介绍 Verilog HDL 的系统任务和函数，分为以下几类。

显示任务

$display	$write	
$displayb	$writeb	
$displayh	$writeh	
$displayo	$writeo	
$strobe	$monitor	
$strobeb	$monitorb	
$strobeh	$monitorh	
$strobeo	$monitoro	
	$monitoroff	
	$monitoron	

文件 I/O 任务

$fclose	$fopen	
$fdisplay	$fwrite	
$fdisplayb	$fwriteb	
$fdisplayh	$fwriteh	
$fdisplayo	$fwriteo	
$fstrobe	$fmonitor	
$fstrobeb	$fmonitorb	
$fstrobeh	$fmonitorh	
$fstrobeo	$fmonitoro	
$swrite	$sformat	
$swriteb	$fgetc	
$swriteh	$ungetc	
$swriteo	$fgets	
$fscanf	$sscanf	
$fread	$rewind	
$fseek	$ftell	
$fflush	$ferror	
$feof	$readmemb	
$sdf_annotate	$readmemh	

时间标度任务

$printtimescale	$timeformat

仿真控制任务

$finish	$stop

PLA 建模任务

$async $and $array	$async $and $plane
$async $nand $array	$async $nand $plane
$async $or $array	$async $or $plane
$async $nor $array	$async $nor $plane
$sync $and $array	$sync $and $plane
$sync $nand $array	$sync $nand $plane
$sync $or $array	$sync $or $plane

随机分析任务

$q_initialize	$q_add
$q_remove	$q_full
$q_exam	

仿真时间函数

$realtime	$stime
$time	

转换函数

$bitstoreal	$realtobits
$itor	$rtoi
$signed	$unsigned

概率分布函数

$random	$dist_chi_square
$dist_erlang	$dist_exponential
$dist_normal	$dist_poisson
$dist_t	$dist_uniform

命令行输入

$test $plusargs	$value $plusargs

数学函数

$clog2	$asin
$ln	$acos
$log10	$atan
$exp	$atan2
$sqrt	$hypot
$pow	$sinh
$floor	$cosh
$ceil	$tanh
$sin	$asinh
$cos	$acosh
$tan	$atanh


视频讲解

5.10.1 显示系统任务

显示系统任务用于信息显示和输出。这些系统任务进一步分为显示和写入任务、探测监控任务和连续监控任务。

1. 显示和写入任务

这些是显示信息的主要系统任务例程。这两组任务完全相同,只是 $display 会自动在输出末尾添加换行符,而 $write 任务则不会。

$display 和 $write 任务显示其参数的顺序与它们在参数列表中显示的顺序相同。每个参数可以是带引号的字符串、返回值的表达式或空参数。

除非插入某些转义序列以显示特殊字符或指定后续表达式的显示格式,否则字符串参数的内容将按字面形式输出。

转义序列以 3 种形式插入字符串。

(1) 特殊字符"\"表示后面的字符是文字字符或不可打印字符。

(2) 特殊字符"%"表示下一个字符应解释为格式规范,该规范为后续表达式参数建立显示格式。对于字符串中出现的每个"%"字符(%m 和%%除外),应在字符串后面提供相应的表达式参数。

(3) 特殊字符串"%%"表示百分号字符"%"的显示。

任何空参数都会在显示中产生一个空格字符(空参数由参数列表中两个相邻逗号表示)。

在没有参数的情况下调用 $display 任务时,只需打印一个换行符。没有参数的 $write 任务根本不会打印任何内容。

1) 用于特殊字符的转义序列

如表 5.28 所示,转义序列用于打印特殊的字符。

表 5.28 转义序列用于打印特殊的字符

参 数	描 述
\n	换行
\t	制表符
\\	字符\
\"	字符"
\ddd	3 位八进制数表示的 ASCII 码值
%%	字符%

【例 5.169】 转义序列用于打印特殊字符 Verilog HDL 描述的例子。

```
module disp;
initial begin
    $display("\\\t\\\n\"\123");
end
endmodule
```

仿真该例子将输出:

```
\   \
"S
```

2) 格式规范

表 5.29 给出转义序列格式规范。当包含在字符串参数中时,每个转义序列指定后续表达式的显示格式。对于字符串出现的每个"%"字符(%m 和%%除外),参数列表中的字

符串后面都应该有一个相应的表达式。显示字符串时,表达式的值将替换格式规范。

任何没有相应格式规范的表达式参数都使用 $display 和 $write 中的默认十进制格式、$displayb 和 $writeb 中的二进制格式、$displayo 和 $writeo 中的八进制格式以及 $displayh 和 $writeh 中的十六进制格式显示。

表 5.29 转义序列格式规范

输出格式符	格 式 说 明
%h 或%H	以十六进制显示
%d 或%D	以十进制显示
%o 或%O	以八进制显示
%b 或%B	以二进制显示
%c 或%C	以 ASCII 字符形式显示
%v 或%V	显示网络信号强度
%l 或%L	显示库绑定信息
%m 或%M	显示模块分层名
%s 或%S	以字符串显示
%t 或%T	显示当前时间格式
%u 或%U	未格式化的二值数据
%z 或%Z	未格式化的四值数据

格式化规范%l(或%L)是为了显示特定模块的库信息而定义的。该信息应显示为 library.cell,与提取当前模块实例的库名字和当前模块实例单元名字相对应。

格式化规范%u(或%U)是为写入不带格式(二进制值)的数据而定义的。应用程序应将指定数据二值二进制表述传输到输出流。这个转义序列可以用于任何现有的显示系统任务,尽管 $fwrite 应该是首选的。源中的任何不确定“x”或高阻“z”应看为零。此格式说明符用于支持没有“x”和“z”概念的外部程序之间的数据传输。建议需要保留“x”和“z”的应用程序使用%z I/O 格式规范。

数据应该以底层系统的原本的端格式写入文件(即,按照与使用 PLI 和使用 C 语言 write(2)系统调用相同的字节顺序)。数据应该以 32 位为单位写入,首先写入包含 LSB 的字。

注:对于 POSIX 应用程序,可能需要使用 wb、wb+或 w+b 说明符打开未格式化 I/O 的文件,以避免系统实现与特殊字符匹配的未格式化流中的 I/O 更改模式。

格式化规范%z(或%Z)是为写入不带格式(二进制值)的数据而定义的。应用程序应将指定数据的四值二进制表示传输到输出流。这个转义序列可以用于任何现有的显示系统任务,尽管 $fwrite 应该是首选的。此格式说明符用于支持与识别并支持“x”和“z”概念的外部程序之间的数据传输。不需要保留“x”和“z”的应用程序建议使用%u I/O 格式规范。

数据应该以底层系统的原本的端格式写入文件(即,以与使用 PLI 相同的原本端格式,数据采用 s_vpi_vecval 结构,并使用 C 语言 write(2)系统调用将结构写入磁盘)。数据应该以 32 位为单位写入,结构包含先写入的 LSB。

表 5.30 给出了用于显示实数的指定格式。

表 5.30 用于显示实数的指定格式

参 数	描 述
%e 或%E	指数格式显示实数
%f 或%F	浮点格式显示实数
%g 或%G	以上两种格式中较短的格式显示实数

%t 格式规范与 $timeformat 系统任务配合使用，以指定统一的时间单位、时间精度和格式，用于报告来自使用不同时间单位和精度的各个模块的时序信息。

【例 5.170】 display 系统任务 Verilog HDL 描述的例子，如代码清单 5-75 所示。

代码清单 5-75　display 系统任务的 Verilog HDL 描述

```
module disp;
reg [31:0] rval;
pulldown (pd);
initial begin
    rval = 101;
    $display("rval = %h hex %d decimal",rval,rval);
    $display("rval = %o octal\nrval = %b bin",rval,rval);
    $display("rval has %c ascii character value",rval);
    $display("pd strength value is %v",pd);
    $display("current scope is %m");
    $display("%s is ascii value for 101",101);
    $display("simulation time is %t", $time);
end
endmodule
```

仿真该例，将显示下面的结果：

```
rval = 00000065 hex 101 decimal
rval = 00000000145 octal
rval = 00000000000000000000000001100101 bin
rval has e ascii character value
pd strength value is StX
current scope is disp
e is ascii value for 101
simulation time is          0
```

3）所显示数据的位宽

对于表达式参数，写到输出文件(或者终端)时，将自动调整值的位宽。例如，当以十六进制显示 12 位表达式的结果时，分配 3 个字符；当以八进制显示时，分配 4 个字符。这是因为表达式最大的可能值为 FFF(十六进制)和 4095(十进制)。

当以十进制显示时，将前面的零去掉，用空格代替。对于其他基数/进制，总是显示前面的零。

如下所示，通过在%字符和表示基数的字符之间插入一个 0，覆盖所显示的数据自动调整位宽。

```
$display("d = %0h a = %0h", data, addr);
```

【例 5.171】 显示不同数据位宽 Verilog HDL 描述的例子。

```
reg[11:0] r1;
initial begin
    r1 = 10;
    $display( "Printing with maximum size - :%d::%h:", r1,r1 );
    $display( "Printing with minimum size - :%0d::%0h:", r1,r1 );
end
endmodule
```

仿真后的结果如下：

```
Printing with maximum size -  :10: :00a:
Printing with minimum size -  :10::a:
```

第一个 $display 是标准的显示格式；第二个 $display 使用%0 的格式。

4）未知和高阻值

当一个表达式的结果包含一个未知或者高阻值的时候，下面的规则用于显示值。

(1) 对于%d的格式,规则如下:①如果所有位是未知值,显示单个小写x字符;②如果所有位为高阻值,显示单个小写z字符;③如果是某些而不是全部的位为未知值,显示大写的X字符;④如果是某些而不是全部的位为高阻值,显示大写的Z字符。除非有一些位为未知值,在这种情况下,显示大写字符X;⑤十进制数总是在一个固定宽度的区域向右对齐。

(2) 对于%h和%o的格式,规则如下:①每4位为一组表示一个十六进制数字,每3位为一组表示一个八进制数字;②如果一个组内的所有位都是未知值,为该进制的某个数字显示小写字母x;③如果一个组内的所有位都是高阻值,为该进制的某个数字显示小写字母z;④如果一个组内的某些位为未知值,为该进制的某个数字显示大写字母X;⑤如果一个组内的某些位为高阻值,为该进制的某个数字显示大写字母Z,除非有一些位为未知值,在这种情况下,为该进制的某个数字显示大写字母X。

(3) 在二进制格式(%b)中,使用字符"0"、"1"、"x"和"z"分别打印每一位。

【例5.172】 显示未知值和高阻值Verilog HDL描述的例子。

```
$display("%d", 1'bx);x
$display("%h", 14'bx01010); xxXa
$display("%h %o", 12'b001xxx101x01,12'b001xxx101x01);
```

结果如下:

```
x
xxXa
XXX 1x5X
```

5) 强度格式

%v格式用于显示标量网络的强度。对于字符串中出现的每个%v规范,参数列表中的字符串后面应该有相应的标量引用。用三个字符格式报告一个标量网络的强度:前两个字符表示强度,第三个字符指示标量当前的值,其值如表5.31所示。

表5.31 强度格式的逻辑值

参 数	描 述	参 数	描 述
0	用于逻辑"0"值	Z	用于一个高阻值
1	用于逻辑"1"值	L	用于一个逻辑"0"或者高阻值
X	用于一个未知值	H	用于一个逻辑"1"或者高阻值

前两个字符即强度字符,可以是两个字母助记符或者一对十进制数字。通常,使用两个字符助记符表示强度信息。然而,少数情况下使用一对十进制数字表示各种信号强度。表5.32给出了用于表示不同强度级的助记符。

表5.32 用于表示信号强度的助记符

助 记 符	强 度 名 称	强 度 级
Su	Supply驱动	7
St	强驱动	6
Pu	Pull驱动	5
La	大电容	4
We	弱驱动	3
Me	中电容	2
Sm	小电容	1
Hi	高阻	0

强度格式提供了4种驱动强度和3种电荷存储强度。驱动强度与门输出和连续分配输出关联;电荷存储强度和trireg类型网络相关。

对于逻辑"0"和"1",信号没有强度范围时,使用助记符;否则,逻辑值前面有两个十进制

数字，表示最大和最小强度级。

对于未知值，当 0 和 1 强度分量在相同的强度级时，使用一个助记符；否则，未知值 X 前面有两个十进制数字，分别用于表示 0 和 1 强度级。

高阻强度没有已知的逻辑值，用于这个强度的逻辑值是 Z。

对于值 L 和 H，助记符始终表示强度级。

【例 5.173】 强度级显示 Verilog HDL 描述的例子。

```
always
#15 $display( $time,,"group = %b signals = %v %v %v",{s1,s2,s3},s1,s2,s3);
```

下面给出了这样一个调用可能的输出：

```
0   group = 111 signals = St1 Pu1 St1
15  group = 011 signals = Pu0 Pu1 St1
30  group = 0xz signals = 520 PuH HiZ
45  group = 0xx signals = Pu0 65X StX
60  group = 000 signals = Me0 St0 St0
```

表 5.33 解释了输出中不同的强度格式。

表 5.33 强度格式的解释

强 度 格 式	解　　释
St1	强驱动“1”值
Pu0	pull 驱动“0”值
HiZ	高阻状态
Me0	中等电容强度的 0 电荷存储
StX	强驱动未知值
PuH	pull 驱动强度为“1”或高阻值
65X	有强驱动 0 分量和 pull 驱动 1 分量的未知值
520	0 值，可能的强度范围从 pull 驱动到中电容

6）层次化名字格式

%m 格式标识符不接受参数。相反，它使显示任务打印的调用包含格式标识符的系统任务模块、任务、函数或者命名块层次名字。当有很多模块实例调用系统任务时，这非常有用。一个明确的应用是一个触发器或者锁存器时序检查消息，%m 格式标识符精确地找到负责时序检查消息的模块实例。

7）字符串格式

%s 格式标识符用于将 ASCII 码打印为字符。对于字符串中出现的每个%s，参数列表中的字符串后面都应该有相应的参数。相关参数理解为一个 8 位十六进制 ASCII 码，每 8 位表示一个字符。如果参数是一个变量，它的值右对齐，最右的值是字符串最后字符的最低有效位。在字符串的末尾不要求终止符，不打印前导零。

2. 探测监控任务

探测监控任务包含 $strobe、$strobeb、$strobeh 和 $strobeo。

系统任务 $strobe 提供在选定时间内显示仿真数据的能力。该时间表示当前仿真时间结束，此时刻仿真时间内的所有仿真时间都已经发生。该任务参数的指定方式与 $display 系统任务完全相同。

【例 5.174】 $strobe 任务 Verilog HDL 描述的例子。

```
forever@(negedge clock)
    $strobe ("At time %d, data is %h", $time,data);
```

在该例子中，在时钟的每个下降沿，$strobe 写时间和数据信息到标准的输出和日志文件。该行为应在仿真时间之前发生，并且在该时间发生的所有其他事件之后发生，以确保写入的数据是仿真时间的正确数据。

3. 连续监控任务

连续监控任务包括 $monitor、$monitorb、$monitorh 和 $monitoro。

$monitor 任务提供了监视和显示指定为任务参数的任何变量或表达式的值的能力。此任务参数指定的方式与 $display 系统任务完全相同，包括特殊字符和格式规范的转义序列的使用。

当一个或多个参数调用 $monitor 任务时，仿真器会设置一种机制，即每次参数列表中的变量或表达式改变值（$time、$stime 或 $realtime 系统函数除外）时，这个参数列表就会在时间步骤结束时显示，就像由 $display 任务报告一样。如果两个或多个参数同时更改值，则只生成一个展示新值的显示。

只能一次激活一个 $monitor 显示列表；然而，在仿真过程中，可以多次发出带有新显示列表的新 $monitor 任务。

$monitoron 和 $monitoroff 任务控制使能和禁止监视的监视器标志。使用 $monitoroff 关闭标志并禁用监视。$monitoron 系统任务可用于打开标志，以便使能监视，并且最近对 $monitor 的调用可以恢复显示。对 $monitoron 的调用应该在它调用之后立即显示，无论值是否发生变化，这用于监视会话开始时建立初始值。默认情况下，在仿真开始时打开监视器标志。

5.10.2 文件输入/输出系统任务和函数


视频讲解

用于文件操作的系统任务和函数分为下面的类型。

（1）打开和关闭文件的函数和任务。

（2）输出值到文件的任务。

（3）输出值到变量的任务。

（4）从文件中读取值，然后加载到变量或者存储器的任务和函数。

1. 打开和关闭文件

函数 $fopen 打开由 file_name 参数指定的文件，并返回 32 位多通道描述符 multi_channel_descriptor 或 32 位文件描述符 fd，这取决于是否存在参数类型 type。其语法格式如下：

```
multi_channel_descriptor = $fopen("file_name");
```

或

```
fd = $fopen("file_name",type);
```

其中，file_name 为一个字符串，或者是包含字符串的 reg，该字符串用于命名要打开的文件；type 是一个字符串，或者是包含表 5.34 中一个表单字符串的 reg，该表单指示如何打开文件，如果省略类型，则打开文件进行写入，并返回多通道描述符 mcd。如果提供了 type，则按照指定的 type 打开文件，并返回文件描述符 fd。

表 5.34 文件描述符的类型

参 数	描 述
“r”或者“rb”	打开文件，用于读
“w”或者“wb”	截断到长度零，或者创建文件用于写
“a”或者“ab”	添加，打开用于在文件的末尾(EOF)写入或创建用于写入
“r+”“r+b”或者“rb+”	打开用于更新（读和写）
“w+”“w+b”或者“wb+”	截断或者创建用于更新
“a+”“a+b”或者“ab+”	添加、打开或者创建用于在 EOF 更新

多通道描述 mcd 是一个 32 位 reg，其中设置了一个用于指示打开了哪个文件的位。mcd 的最低有效位（第 0 位）总是标准输出。输出被定向到两个或多个用多通道描述符打开的文件，方法是将它们的 mcd 按位“或”并写入结果值。

多通道描述符的最高有效位（第 31 位）是保留的，应始终清除，将实现最多 31 个文件，通过多通道描述符进行输出。

文件描述符 fd 是 32 位。fd 的最高有效位（第 31 位）是保留的，并且应始终设置，这允许文件输入和输出函数的实现来确定如何打开文件。剩余的位保存一个小数字，指示打开了什么文件。预先打开三个文件描述符：STDIN、STDOUT 和 STDERR，分别具有值 32'h8000_0000、32'h8000_0001 和 32'h8000_0002。STDIN 预打开用于读取，STDOUT 和 STDERR 预打开用于添加。

与多通道描述符不同，文件描述符不能通过按位“或”组合，以便将输出指向多个文件。相反，根据表 5.34，基于类型值文件通过文件描述符打开，用于输入、输出、输入和输出，以及添加操作。

如果无法打开文件（文件不存在，并且指定类型为“r”“rb”“r+”“r+b”或“rb+”，或权限不允许在该路径打开文件），则为 mcd 或 fd 返回零。应用程序可以调用 $ferror 来确定最近错误的原因。

上述类型中的“b”用于区分二进制文件和文本文件。许多系统（如 Unix）不区分二进制文件和文本文件，在这些系统中，忽略“b”。然而，一些系统（例如运行 Windows NT 的计算机）对某些二进制值执行数据映射，这些二进制值写入文件或从文件中读取，这些文件是为文本访问而打开的。

$fclose 系统任务关闭 fd 指定的文件或关闭多通道描述符 mcd 指定的文件。不允许对 $fclose 关闭的任何文件描述符进行输出或输入。$fclose 操作会隐式取消对文件描述符或多通道描述符活动的 $fmonitor 和/或 $fstrobe 操作。$fopen 功能应重新使用已关闭的通道。

注：任何时候可以同时打开的输入和输出通道的数量取决于操作系统，某些操作系统不支持打开文件进行更新。

2. 文件输出系统任务

四个格式化的显示任务（$display、$write、$monitor 和 $strobe）中的每个都有一个对应的任务，与标准输出相反，它们写入特定的文件。这些对应任务 $fdisplay、$fwrite、$fmonitor 和 $fstrobe 接受与它们所基于的任务相同类型的参数，但有一个例外：第一个参数应为多通道描述符或文件描述符，指示文件输出的方向。多通道描述符要么是变量，要么是采用 32 位无符号整数值形式的表达式的结果。

$fstrobe 和 $fmonitor 系统任务的工作方式与 $strobe 和 $monitor 系统任务相同，只是它们使用多通道描述符写入文件进行控制。与 $monitor 不同，可以将任意数量的 $fmonitor 任务设置为同时处于活动状态。然而，$monitoron 和 $monitoroff 任务没有对应项。任务 $fclose 用于取消活动的 $fstrobe 或 $fmonitor 任务。

【例 5.175】 设置多通道描述符 Verilog HDL 描述的例子。

```
integer
    messages, broadcast,
    cpu_chann, alu_chann, mem_chann;
initial begin
    cpu_chann =  $fopen("cpu.dat");
    if (cpu_chann == 0) $finish;
        alu_chann =  $fopen("alu.dat");
```

```
    if(alu_chann == 0) $finish;
        mem_chann = $fopen("mem.dat");
    if (mem_chann == 0) $finish;
        messages = cpu_chann | alu_chann | mem_chann;
    //broadcast 包含标准的输出
    broadcast = 1 | messages;
end
endmodule
```

在该例子中,使用 $fopen 函数打开三个不同的通道。然后,函数返回的三个通道描述符在按位“或”运算中组合,并分配给整数变量消息。然后,可以将 messages 变量用作文件输出任务中的第一个参数,以将输出同时指向所有三个通道。为了创建一个将输出指向标准输出的描述符,messages 变量是一个带常数 1 的按位“或”的值,这有效地使能了通道 0。

以下的文件输出任务显示了如何使用前面打开的通道。

```
$fdisplay( broadcast, "system reset at time %d", $time );

$fdisplay( messages, "Error occurred on address bus",
              " at time %d, address = %h", $time, address );

forever @(posedge clock)
  $fdisplay( alu_chann, "acc = %h f = %h a = %h b = %h", acc, f, a, b );
```

3. 将数据格式化为字符串

$swrite 的命令格式如下:

```
string_output_task_name ( output_reg , list_of_arguments ) ;
```

其中,string_output_task_name 为输出任务名字,包括 $swrite、$swriteb、$swriteh、$swriteo。output_reg 为输出 reg 变量名字;list_of_arguments 为参数列表。$swrite 的第一个参数是一个 reg 类型的变量,用于保存写入的字符串。

$swrite 任务系列基于 $fwrite 任务系列,并接受与其所基于的任务相同类型的参数,但有一个例外:$swrite 的第一个参数应该是一个 reg 变量,最终字符串应写入该变量,而不是一个指定要写入最终字符串的文件的变量。

$sformat 的命令格式如下:

```
$sformat ( output_reg , format_string , list_of_arguments ) ;
```

系统任务 $sformat 与系统任务 $swrite 类似,但有一个主要区别:与输出系统任务的显示和写入系列不同,$sformat 始终将其第二个参数(仅第二个)解释为格式化字符串。此格式参数可以是静态字符串,例如“data is %d”,也可以是内容被解释为格式字符串的 reg 变量。没有其他参数被解释为格式化字符串。$sformat 支持 $display 支持的所有格式说明符。

$sformat 的其余参数将使用在 format_string 中的任何格式说明符进行处理,直到所有此类格式说明符都用完为止。如果没有为格式说明符提供足够的参数或提供的参数太多,则应用程序应发出警告并继续执行。如果可能,应用程序可以静态地确定格式说明符和参数数量不匹配,并发出编译时错误消息。

注:如果 format_string 是 reg,则可能无法在编译时确定其值。

【例 5.176】 $sformat 系统任务 Verilog HDL 描述的例子,如代码清单 5-76 所示。

代码清单 5-76 $sformat 系统任务的 Verilog HDL 描述

```
module test;                          // module 关键字定义模块 test
reg [5*8-1:0] str1 = "hello";         // reg 关键字定义 reg 类型向量变量 str1 并初始化
reg [5*8-1:0] str2 = "world";         // reg 关键字定义 reg 类型向量变量 str2 并初始化
```

```
integer i = 10;                          // integer 关键字定义整数 i 并初始化
reg [14 * 8 - 1:0] string;               // reg 关键字定义 reg 类型向量变量 string 并初始化
initial                                  // initial 关键字定义初始化部分
begin                                    // begin 关键字标识初始化部分的开始,类似 C 语言的"{"
 $sformat(string, " % s  % s  % 2d",str1,str2,i);  // 调用系统任务 sformat,不同数据合并后
                                                   // 转换为字符串 string
 $display(" % s",string);                // 调用系统任务 display,显示字符串 string
end                                      // end 关键字标识初始化段的结束
endmodule                                // endmodule 关键字标识 test 模块的结束
```

当在 ModelSim 软件中执行行为仿真时,在 Transcript 窗口中显示如下的信息:

```
hello world 10
```

4. 从文件中读取数据

使用文件描述符打开的文件只有在使用 r 或 r+类型值打开时才能读取。

1) 一次读一个字符

【例 5.177】 一次读取一个字符 Veriog HDL 描述的例子 1。

```
c =  $fgetc ( fd );
```

从 fd 指定的文件中读取一字节,如果发生读取错误,则将 c 设置为 EOF(-1)。调用 $ferror,可以确定读取错误的原因。

【例 5.178】 一次读取一个字符 Veriog HDL 描述的例子 2。

```
code =  $ungetc ( c, fd );
```

将 c 指定的字符插入到文件描述符 fd 指定的缓冲区,字符 c 将在对该文件描述符的下一个 $fgetc 调用时返回。文件本身并不变化。如果将字符推到文件描述符时发生错误,将代码设置为 EOF; 否则,将代码设置为 0。

2) 一次读一行

【例 5.179】 一次读一行 Verilog HDL 描述的例子。

```
integer code ;
code =  $fgets ( str, fd );
```

从 fd 指定的文件中将字符读到 reg 类型的 str,直到将 str 填满,或者读到新的一行,或者遇到 EOF 条件。如果 str 的长度不是整数字节,则不使用最高有效部分字节来确定大小。

如果发生错误,则将代码设置为 0; 否则,将在代码中返回读取的字符数。

3) 读格式化数据

【例 5.180】 $fscanf 系统任务 Verilog HDL 描述的例子。

```
integer code ;
code =  $fscanf ( fd, format, args );
code =  $sscanf ( str, format, args );
```

$fscanf 从文件描述符 fd 指定的文件中读取; $sscanf 从 reg 类型 str 中读取。

这两个函数都读取字符,根据格式解释它们并保存结果。这两个函数都期望将控制字符串、格式和一组指定结果放置位置的参数作为参数。如果格式的参数不足,则行为未定义。如果在保留参数时耗尽了格式,则忽略多余的参数。

如果参数太小以至于无法容纳转换后的输入,则通常会传输最低有效位。可以使用 Verilog 支持的任何长度的参数。但是,如果目标是 real 或 realtime,则传输值+Inf(或-Inf)。格式可以是字符串常量或包含字符串常量的 reg。该字符串包含转换规范,用于将输入转换为参数。控制字符串可以包含以下内容。

(1) 空白字符(空格、制表符、新的一行或换行符),对于除字符 c 的所有描述符,忽略输入字段前面的空白。对于 $scanf,空字符也应该看作空白。

(2) 一个普通字符(不是%)必须匹配输入流中的下一个字符。

(3) 转换规范由字符%、可选的赋值抑制字符“*”、指定可选的最大数字字段宽度的十进制数字字符串和转换代码组成。

转换规范指导下一个输入字段的转换,结果被放置在相应参数指定的变量中,除非用字符“*”标识禁止分配/赋值。在这种情况下,不应提供任何参数。

禁止分配/赋值提供了一种描述要跳过的输入字段的方法。输入字段定义为非空格字符的字符串;它扩展到下一个不合适的字符,或者直到耗尽最大字段宽度(如果指定了一个)。

(1) %。此时输入中需要一个%,未完成任何分配/赋值。

(2) b。匹配一个二进制数,由集合 0、1、X、x、Z、z、? 和_中的序列组成。

(3) o。匹配一个八进制数,由集合 0、1、2、3、4、5、6、7、X、x、Z、z、? 和_中的字符序列组成。

(4) d。匹配一个可选的有符号十进制数,由来自集合“+”或“-”的可选符号组成,后面跟着来自集合 0、1、2、3、4、5、6、7、8、9 和_的一系列字符,或集合 x、X、z、Z、? 中的单个值。

(5) h/x。匹配一个十六进制数,由集合 0、1、2、3、4、5、6、7、8、9、a、A、b、B、c、C、d、D、e、E、f、F、X、x、Z、z、? 和_中的序列组成。

(6) f/e/g。匹配一个浮点数。浮点数的格式是一个可选符号(“+”或“-”),后面跟着一个来自 0、1、2、3、4、5、6、7、8、9 集合的数字串,可选包含小数点字符“.”,后面跟着包括 e/E 的可选指数部分,接着是可选符号,再是来自 0、1、2、3、4、5、6、7、8、9 集合的数字串。

(7) v。匹配一个网络信号的强度,由三个字符序列组成。这种转换不是特别有用,因为强度值实际上只能有效地分配给网络,而 $fscanf 只能将值分配给 reg 类型(如果分配给 reg 类型,则将值转换为等效的四值)。

(8) t。匹配浮点数。浮点数的格式是一个可选的符号(“+”或“-”),后跟一个来自 0、1、2、3、4、5、6、7、8、9 集合的数字串,可选包含小数点字符“.”,接着是包括 e 或 E 的可选指数部分、可选符号和一组来自 0、1、2、3、4、5、6、7、8、9 集合的数字串。然后根据由 $timeformat 设置的当前时间标度对匹配的值进行标定和四舍五入。例如,如果时间标度是`timescale 1ns/1ps,且时间格式为“$timeformat(-3,2,“ms”,10);”那么用 $scanf(“10.345”,“%t”,t)读取的值将返回 10350000.0。

(9) c。匹配单个字符,返回该字符的 8 位 ASCII 值。

(10) s。匹配一个字符串,该字符串是非空白字符序列。

(11) u。匹配未格式化(二进制)数据。应用程序应从输入传输足够的数据以填充目标 reg 类型变量。数据是从匹配的 $fwrite(“%u”,data)或从用其他编程语言(如 C、Perl 或 FORTRAN)编写的外部应用程序中获得的。

应用程序应将二值二进制数据从输入流传输到目标 reg 类型变量,将数据扩展为四值格式。这个转义序列可以用于任何现有的输入系统任务,尽管 $fscanf 应该是首选的。由于输入数据不能表示 x 或 z,所以无法理解在最终 reg 变量中获得的 z 或 z。此格式说明符用于支持与没有 x 和 z 概念的外部程序之间的数据传输。

鼓励需要保留 x 和 z 的应用程序使用%z I/O 格式规范。

数据应以底层系统的原本端格式从文件中读取(即,按照与使用 PLI 和使用 C 语言 read(2)系统调用相同的端顺序)。

对于 POSIX 应用程序,可能需要使用“rb”“rb”或“r+b”说明符打开未格式化 I/O 的文

件，以避免 I/O 更改模式的系统实现在匹配特殊字符的未格式化流中。

(12) z。格式化规范%z(或%Z)是为读取不带格式(二进制值)的数据而定义的。应用程序应将指定数据的四值二进制表示从输入流传输到目标 reg 类型变量。该转义序列可用于任何现有的输入系统任务，但 $fscanf 应是首选的。

该格式说明符专门用于支持与识别和支持 x 和 z 概念的外部程序之间的数据传输。不需要保留 x 和 z 的应用程序建议使用%u I/O 格式规范。

数据应以底层系统的原本端格式从文件中读取，即，按照与使用 PLI 相同的端顺序，数据采用 s_vpi_vecval 结构)，C 语言 read(2)系统调用用于从磁盘读取数据。

对于 POSIX 应用程序，可能需要使用“rb”“rb+”或“r+b”说明符打开未格式化 I/O 的文件，以避免 I/O 更改模式的系统实现在匹配特殊字符的未格式化流中。

(13) m。以字符串形式返回当前层次结构路径，不从输入文件或 str 参数读取数据。

如果%后有无效的转换字符，则操作结果取决于实现。

如果 $sscanf 的格式字符串或 str 参数包含未知位(z 或 z)，则系统任务应返回 EOF。

如果在输入过程中遇到 EOF，转换将停止。如果 EOF 发生在读取与当前命令匹配的任何字符之前(在允许的情况下，前导空格除外)，则当前命令的执行将因输入失败而终止，否则以下命令(如果有)的执行将因输入失败而终止(除非当前命令的执行因匹配失败而终止)。

如果转换在冲突的输入字符上终止，则在输入流中不读取有问题的输入字符。尾部空白(包括换行符)保留为未读，除非与命令匹配。文字匹配和抑制分配/赋值的成功与否并不能直接确定。

成功匹配和分配输入项的数量以代码形式返回；在输入字符和控制字符串之间的早期匹配失败的情况下，该数字可以是 0。如果输入在第一次匹配失败或转换之前结束，则返回 EOF。应用程序可以调用 $ferror 来确定最近错误的原因。

4) 读二进制数据

【例 5.181】 ＃fread 系统任务 Verilog HDL 描述的例子。

```
integer code;
code = $fread(myreg, fd);
code = $fread(mem, fd);
code = $fread(mem, fd, start);
code = $fread(mem, fd, start, count);
code = $fread(mem, fd, , count);
```

该例子将 fd 指定的文件中的二进制数据读取到 reg 类型变量 myreg 或存储器 mem 中。start 是可选参数。如果存在，start 用于要加载的存储器中第一个元素的地址；如果不存在，应使用存储器中编号最低的位置。count 是可选参数。如果存在，count 应为 mem 中应加载的最大位置数；如果未提供，则应将可用数据填入存储器。如果 $fread 正在加载 reg，则忽略 start 和 count。

如果在系统任务中没有指定寻址信息，并且数据文件中没有出现地址规范，则默认起始地址是存储器声明中给出的最低地址。连续的字被加载到最高地址，直到存储器满或完全读取数据文件为止。如果在没有完成地址的任务中指定了起始地址，则加载从指定的开始地址开始，并继续向存储器声明中给定的最高地址加载。

start 是存储器中的地址。对于 start＝12 和存储器 up[10:20]，第一个数据将在 up[12]加载。对于存储器 down[20:10]，加载的第一个位置是 down[12]，然后是 down[13]。

文件中的数据应该逐字节读取，以填充请求。8 位宽的存储器使用每个存储器 1 字节加

载,而 9 位宽的存储器使用每个存储器 2 字节加载。数据以大端形式从文件中读取,第一字节读取用于填充存储器元素中的最高有效位置。如果存储器宽度不能被 8(8、16、24、32)整除,则由于截断,文件中的所有数据都不会加载到存储器中。

从文件中加载的数据被看作"二值"数据。数据中设置的位被解释为 1,未设置的位被解释为 0。使用 $fread 无法读取 x 或 z 的值。

如果读取文件时发生错误,则 code 设置为零;否则,将在 code 中返回读取的字符数。应用程序可以调用 $ferror 来确定最近错误的原因。

注:没有二进制模式和 ASCII 模式,可以自由地混合来自同一文件的二进制和格式化的读取命令。

5. 文件定位

【例 5.182】 $ftell 系统任务 Verilog HDL 描述的例子。

```
integer pos ;
pos = $ftell ( fd );
```

在 pos 中返回 fd 所指向文件从开始到当前位置的偏移量,该偏移量将由对该文件描述符的后续操作读取或写入。该值可用于后续的 $fseek 调用,以将文件重新定位到该位置。任何重定位将取消任何 $ungetc 操作。如果发生错误,则返回 EOF。

【例 5.183】 $fseek 和 $rewind 系统任务 Verilog HDL 描述的例子。

```
code = $fseek ( fd, offset, operation );
code = $rewind ( fd );
```

在 fd 指定文件上设置下一个输入/输出的位置。根据 0、1 和 2 的操作值,下一个位置是从开始、从当前位置,或者从文件结束的有符号距离偏置字节处:

(1) 0,将位置设置等于偏置字节。

(2) 1,将位置设置等于当前位置加偏置。

(3) 2,将位置设置等于文件结束位置加偏置。

$rewind 等价于 $fseek(fd,0,0)。

使用 $fseek 或 $rewind 重新定位当前文件位置将取消任何 $ungetc 操作。

$fseek()允许将文件位置指示符设置为超过文件中现有数据的末尾。如果稍后在该处写入数据,则间隙中数据的后续操作读取应返回 0,直到数据实际写入间隙。$fseek 本身不会扩展文件的大小。

当打开文件用于添加(即,当类型为"a"或"a+")时,不可能覆盖文件中已有的信息。$fseek 用于将文件指针重新定位到文件中的任何位置,但是当输出写入文件时,将忽略当前文件指针。所有输出都在文件末尾写入,并使文件指针在输出末尾重新定位。

如果重新定位文件时发生错误,则代码设置为-1;否则,代码设置为 0。

6. 刷新输出

【例 5.184】 $fflush 系统任务 Verilog HDL 描述的例子。

```
$fflush ( mcd );
$fflush ( fd );
$fflush ( );
```

将任何缓冲的输出写入到 mcd 或 fd 指向的文件。如果调用 $fflush 没有参数,则写到所有打开的文件。

7. I/O 错误状态

如果某个文件 I/O 例程检测到任何错误,则返回错误代码。通常,这对于正常操作来说是足

够的(即,如果打开可选配置文件失败,应用程序通常会继续使用默认值)。然而,有时获取有关错误的更多信息对于正确的应用程序操作是有用的。在这种情况下,可以使用 $ferror 函数:

```
integer errno;
errno =  $ferror ( fd, str );
```

最近一次文件 I/O 操作遇到的错误类型的字符串描述符将写入 str 中,str 的宽度至少为 640 位。错误代码的整数值在 errno 中返回。如果最近的操作未导致错误,则返回的值应为零,并清除 reg 类型变量 str。

8. 检测文件结束

【例 5.185】 $feof 系统任务 Verilog HDL 描述的例子。

```
integer code;
code =  $feof ( fd );
```

当检测到文件结束时,返回非 0 的值;否则,返回 0。

9. 从文件中加载存储器数据

$readmemb 和 $readmemh 这两个系统任务读取指定文本文件中的数据并将其加载到指定存储器中。在仿真过程中,任何一项任务都可以随时执行。要读取的文本文件应仅包含以下内容:空白(空格、换行符、制表符和换页);注释(允许两种类型的注释);二进制或十六进制数。

数字既没有指定长度也没有指定基数/进制格式。对于 $readmemb,每个数字应为二进制。对于 $readmemh,数字应为十六进制。不确定值(x 或 X)、高阻值(z 或 Z)和下画线(_)可用于指定 Verilog HDL 源描述中的数字,应使用空格和/或注释分隔数字。

在下面的讨论中,术语地址(address)是指对存储器进行建模的数组的索引。

当读取文件时,遇到的每个数字都分配给存储器的一个连续的字元素。通过在系统任务调用中指定开始和/或结束地址以及通过在数据文件指定地址来控制寻址。

当数据文件中出现地址时,格式为 at 字符(@)后跟着十六进制数,如下所示:

```
@hh…h
```

数字中允许使用大小写数字。@和数字之间不允许空格。可以使用数据文件中所需的任意多个地址规范。当系统任务遇到地址时,它加载从该存储器地址开始的后续数据。

如果系统任务中未指定寻址信息,且数据文件中未出现地址规范,则默认起始地址应为存储器中的最低地址。应加载连续的字,直到达到存储器中的最高地址或完全读取数据文件。如果在没有结束/完成地址的任务中指定了开始地址,则加载应从指定的开始地址开始,并应向存储器中的最高地址继续。在这两种情况下,即使在数据文件中指定了地址后,加载也应该继续向上。

如果开始地址和结束地址都被指定为任务的参数,则加载应从开始地址开始,并向结束地址继续。如果开始地址大于结束地址,则地址将在连续加载之间递减,而不是递增。即使在数据文件中指定了地址后,加载也应继续遵循此方向。

当在系统任务和数据文件中都指定了寻址信息时,数据文件中的地址应在系统任务参数指定的地址范围内;否则,发出错误消息,并终止加载操作。

如果文件中的数据字数与起始地址到结束地址所暗示的范围内的字数不同,并且数据文件中没有出现地址范围,则应发出警告消息。

【例 5.186】 $readmemh 系统任务 Verilog HDL 描述的例子。

```
reg[7:0] mem[1:256];
initial $readmemh("mem.data", mem);
```

```
initial $readmemh("mem.data", mem, 16);
initial $readmemh("mem.data", mem, 128, 1);
```

(1) 上例中的第二句代码没有显式声明地址。在仿真时间 0 时刻,在存储器地址 1 开始加载数据。

(2) 上例中的第三句代码声明起始地址,但没有声明结束地址。在地址 16 开始加载数据,连续向上到地址 256。

(3) 上例中的第四句代码声明开始地址和结束地址。如果开始地址大于结束地址,则地址递减。在地址 128 开始加载数据,连续向下递减到地址 1。

10. 从 SDF 文件中加载时序数据

$sdf_annotation 系统任务将时序数据从 SDF 文件读取到设计的指定区域。

$sdf_annotate 系统任务的语法格式如下:

```
$sdf_annotate ("sdf_file" [ , [ module_instance ] [ , [ "config_file" ]
              [ , [ "log_file" ] [ , [ "mtm_spec" ]
              [ , [ "scale_factors" ] [ , [ "scale_type" ] ] ] ] ] ] ] );
```

其中,sdf_file 为要打开的 sdf 文件名,由字符串表示,或者保存在包含文件名字字符串的 reg 类型中; module_instance 为可选参数,说明在 SDF 文件中注解的范围。SDF 注解器使用指定例化的层次级运行注解,允许数组索引。如果没有指定该参数,SDF 注解器使用包含该系统任务调用的模块作为 module_instance; config_file 为可选字符串参数,提供了配置文件的名字,该文件提供与注解相关的许多方面的详细控制; log_file 为可选字符串参数,提供了在 SDF 注解期间日志文件的名字。来自 SDF 文件时序数据的每个注解都会在日志文件中生成一个条目; mtm_spec 为可选字符串参数,指定应注释 min/typ/max 三元组中的哪个成员,这将覆盖配置文件中的任何 MTM-SPEC 关键字,表 5.35 给出了 mtm_spec 的参数。scale_factors 为可选的字符串参数,当注解时序值时使用 scale_factors,如"1.6:1.4:1.2",将使得最小值乘以 1.6,典型值乘以 1.4,最大值乘以 1.2,默认值为"1.0:1.0:1.0"。scale_factors 覆盖配置文件中的任何 SCALE_FACTORS 关键字,scale_type 为可选的字符串参数,指定将 scale_factor 用于 min/typ/max 的方式,表 5.36 给出了 scale_type 的参数。这将覆盖配置文件中的任何 SCALE_TYPE 关键字。

表 5.35 mtm_spec 的参数

关 键 字	描 述
MAXIMUM	注解最大值
MINIMUM	注解最小值
TOOL_CONTROL (默认)	注解由仿真器选择的值
TYPICAL	注解典型的值

表 5.36 scale_type 的参数

关 键 字	描 述
FROM_MAXIMUM	将 scale_factor 应用到最大值
FROM_MINIMUM	将 scale_factor 应用到最小值
FROM_MTM(默认)	将 scale_factor 应用到 min/typ/max 值
FROM_TYPICAL	将 scale_factor 应用到典型值

5.10.3 时间标度系统任务

Verilog HDL 提供了两种时间标度任务函数。

(1) $printtimescale。

(2) $timeformat。

1. $printtimescale

$printtimescale 系统任务显示了用于特殊模块的时间单位和精度。其语法格式如下：

```
$printtimescale(module_hierarchical_name);
```

其中，module_hierarchical_name 为模块层次化名字。如果没有指定参数，则输出包含该任务调用的所有模块的时间单位与精度。

以下面的格式显示时间标度信息：

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

【例 5.187】 $printtimescale 系统任务 Verilog HDL 描述的例子。

```
`timescale 1ms / 1us
module a_dat;
initial
     $printtimescale(b_dat.c1);
endmodule

`timescale 10fs / 1fs
module b_dat;
    c_dat c1 ();
endmodule

`timescale 1ns / 1ns
module c_dat;
⋮
endmodule
```

运行后的显示结果如下：

```
Time scale of (b_dat.c1) is 1ns / 1ns
```

2. $timeformat

$timeformat 系统任务执行以下两个功能。

(1) 它指定%t 格式规范如何报告 $write、$display、$strobe、$monitor、$fwrite、$fdisplay、$fstrobe 和 $fmonitor 系统任务组的时间信息。

(2) 它为以交互方式输入的延迟设置时间单位。

该任务语法格式如下：

```
$timeformat(< units >,< precision >,< suffix >,< numeric_field_width >);
```

其中，< units >用于指定时间单位，其取值范围为 0～－15，各值所代表的时间单位如表 5.37 所示；< precision >指定所要显示时间信息的精度；< suffix >表示诸如“ms”、“ns”之类的字符；< numeric_field_width >说明时间信息的最小字符数。

表 5.37 units 所代表的时间单位

units	时间单位	units	时间单位
0	1s	－8	10ns
－1	100ms	－9	1ns
－2	10ms	－10	100ps
－3	1ms	－11	10ps
－4	100μs	－12	1ps
－5	10μs	－13	100fs
－6	1μs	－14	10fs
－7	100ns	－15	1fs

注：虽然 s、ms、ns、ps 和 fs 是秒、毫秒、纳秒、皮秒和飞秒的常用 SI 单位符号，但由于编码字符集中缺少希腊字母 m(mu)，所以“us”代表微秒的 SI 单位符号。

$timeformat 系统任务执行以下两个操作。

(1) 它为以后交互输入的所有延迟设置时间单位。

(2) 在调用另一个 $timeformat 系统任务之前，它为源描述中所有模块中指定的所有%t格式设置时间单位、精度号、后缀字符串和最小字段宽度。

$timeformat 系统任务参数的默认值如表 5.38 所示。

表 5.38 $timeformat 系统任务参数的默认值

参 数	默 认
units	源描述中所有`timescale 编译器命令的最小时间精度参数
precision	0
suffix	空字符串
numeric_field_width	20

下面的例子显示了在 $timeformat 系统任务中使用%t 来指定统一的时间单位、时间精度和时序信息格式。

【例 5.188】 $timeformat 系统任务 Verilog HDL 描述的例子，如代码清单 5-77 所示。

代码清单 5-77 $timeformat 系统任务的 Verilog HDL 描述

```
`timescale 1ms / 1ns
module cntrl;
initial
    $timeformat( - 9, 5, " ns", 10);
endmodule

`timescale 1fs / 1fs
module a1_dat;
reg in1;
integer file;
buf #10000000 (o1,in1);
initial begin
    file = $fopen("a1.dat");
    #00000000 $fmonitor(file," %m: %t in1 = %d o1 = %h", $realtime,in1,o1);
    #10000000 in1 = 0;
    #10000000 in1 = 1;
end
endmodule

`timescale 1ps / 1ps
module a2_dat;
reg in2;
integer file2;
buf #10000 (o2,in2);
initial begin
    file2 = $fopen("a2.dat");
    #00000 $fmonitor(file2," %m: %t in2 = %d o2 = %h", $realtime,in2,o2);
    #10000 in2 = 0;
    #10000 in2 = 1;
end
endmodule
```

执行完后，结果如下：

(1) 文件 a1.dat 的内容：

```
a1_dat: 0.00000 nsin1 = x o1 = x
```

```
a1_dat: 10.00000 ns in1 =  0 o1 = x
a1_dat: 20.00000 ns in1 =  1 o1 = 0
a1_dat: 30.00000 ns in1 =  1 o1 = 1
```

(2) 文件 a2.dat 的内容：

```
a2_dat: 0.00000 ns in2 = x o2 = x
a2_dat: 10.00000 ns in2 = 0 o2 = x
a2_dat: 20.00000 ns in2 = 1 o2 = 0
a2_dat: 30.00000 ns in2 = 1 o2 = 1
```

5.10.4 仿真控制任务

Verilog HDL 提供了两个仿真控制系统任务：

(1) $finish；

(2) $stop。

1. $finish

系统任务 $finish 使仿真器退出，将控制返回到操作系统。语法格式为

```
$finish [ ( n ) ] ;
```

其中，n=0，不打印任何信息；n=1，打印仿真时间和位置；n=2，打印仿真时间、位置，以及在仿真时，CPU 和存储器的利用率。

2. $stop

系统任务 $stop 挂起仿真。在这一阶段，可能将交互命令发送到仿真器。语法格式为

```
$stop [ ( n ) ] ;
```

5.10.5 随机分析任务

Verilog HDL 提供了一个系统任务和函数集合，用于管理队列，这些任务便于随机分析队列模型的实现。

1. $q_initialize

该系统任务创建一个新的队列。其语法格式如下：

```
$q_initialize (q_id, q_type, max_length, status);
```

其中，q_id 是整数，用于标识一个新的队列；q_type 是整数输入。其值标识队列的类型，表 5.39 给出了 $q_type 值的类型；status 表示该操作成功或者错误的状态。max_length 是整数输入，标识队列中允许入口的最大的个数。

表 5.39 $q_type 值的类型

q_type 值	队 列 类 型
1	先进先出
2	后进先出

2. $q_add

该任务在队列添加入口。其语法格式如下：

```
$q_add (q_id, job_id, inform_id, status);
```

其中，q_id 是整数，用于标识添加入口的一个队列；job_id 是整数输入，标识工作；inform_id 是整数输入，与队列入口相关，它的含义由用户定义，如代表在一个 CPU 模型中一个入口的执行时间；status 表示该操作成功或者错误的状态。

3. $q_remove

该任务从一个队列接收一个入口。其语法格式如下：

```
$q_remove (q_id, job_id, inform_id, status);
```

其中，q_id 是整数，用于标识将移除哪个队列；job_id 是整数输入，标识正在移除的入口；inform_id 是整数输出，在 $q_add 时由队列管理器保存它，它的含义由用户定义；status 表示该操作成功或者错误状态。

4. $q_full

该系统任务用于检查一个队列是否有空间用于其他入口。其语法格式如下：

```
$q_full (q_id, status)
```

其中，status 表示该操作成功或者错误的状态。当队列满时，返回 1；否则，返回 0。

5. $q_exam

该系统任务提供队列 q_id 活动性的统计信息。其语法格式如下：

```
$q_exam (q_id, q_stat_code, q_stat_value, status);
```

根据 q_stat_code 所要求的信息，返回 q_stat_value。表 5.40 给出了 $q_exam 系统任务的参数。

表 5.40 $q_exam 系统任务的参数

q_stat_code 内所要求的值	从 q_stat_value 返回值的信息
1	当前队列长度
2	平均到达时间
3	最大队列长度
4	最短等待时间
5	用于队列内工作的最长等待时间
6	队列中的平均等待时间

6. 状态编码

所有的队列管理任务和函数返回一个输出状态码，表 5.41 给出了状态码的值及其含义。

表 5.41 状态码的值及含义

状 态 码 值	含 义
0	OK
1	队列满，不能添加
2	未定义的 q_id
3	队列空，不能移除
4	不支持的队列类型，不能创建队列
5	0=>指定的长度，不能创建队列
6	重复 q_id，不能创建队列
7	没有足够的存储器，不能创建队列

【例 5.189】 随机分析任务 Verilog HDL 描述的例子。

```
always @(posedge clk)
begin
  //检查队列是不是满
 $q_full(queue1, status);
  //如果满，则显示信息和移除一个条目
if (status) begin
   $display("Queue is full");
```

```
    $q_remove(queue1, 1, info, status);
  end
    //添加一个新的条目到队列 queue1
  $q_add(queue1, 1, info, status);
    //如果有错误,显示消息
  if (status)
    $display("Error %d",status);
  end
end
```

5.10.6 仿真时间函数

Verilog HDL 提供系统函数用于返回当前的仿真时间。

1. $time

该系统函数用于返回 64 位的整型仿真时间,与调用该函数模块的时间尺度相关。

【例 5.190】 $time 系统函数 Verilog HDL 描述的例子,如代码清单 5-78 所示。

代码清单 5-78 $time 系统函数的 Verilog HDL 描述

```
`timescale 10ns / 1ns
module test;
reg set;
parameter p = 1.55;
initial begin
    $monitor( $time,,"set = ",set);
    #p set = 0;
    #p set = 1;
end
endmodule
```

该例子的输出如下:

```
0 set = x
2 set = 0
3 set = 1
```

在该例子中,在仿真时间 16ns 时,给 reg 类型变量分配一个值 0;在仿真时间 32ns,分配值 1。下面的步骤决定了 $time 系统函数返回的时间值。

(1) 仿真时间 16ns 和 32ns,被标定到 1.6 和 3.2,因为用于模块的时间单位是 10ns,因此这个模块报告的时间值是 10ns 的倍数。

(2) 值 1.6 四舍五入到 2,3.2 四舍五入为 3,这是因为 $time 系统函数返回一个整数。时间精度不会引起这些值的四舍五入。

2. $stime

$stime 系统函数返回一个无符号整数,它是一个 32 位的时间值,按照调用它的模块的时间标度(timescale)缩放。如果实际仿真时间不适合 32 位,则返回当前仿真时间的低 32 位。

3. $realtime

该系统函数向调用它的模块返回实时仿真时间,它与调用该函数模块的时间尺度相关。

【例 5.191】 $realtime 系统函数 Verilog HDL 描述的例子。

```
`timescale 10ns / 1ns
module test;
reg set;
parameter p = 1.55;
initial begin
    $monitor( $realtime,,"set = ",set);
    #p set = 0;
```

```
    #p set = 1;
end
endmodule
```

输出结果如下：

```
0 set = x
1.6 set = 0
3.2 set = 1
```

5.10.7 转换函数

转换函数可用于常数表达式。以下函数处理实数。

(1) $rtoi(real_value)，通过截断小数值将实数转换为整数。

(2) $itor(integer_value)，将整数转换为实数。

(3) $realtobits(real_value)，在模块端口之间传递位模式；将实数转换为该实数的64位表示(向量)。

(4) $bitstoreal(bit_value)，将位模式转换为实数(与$realtobits相反)。

这些函数接受或生成的实数应符合IEEE 754的实数表示。转换应将结果四舍五入为最接近的有效表示。

【例5.192】 $realtobits和$bitstoreal系统函数Verilog HDL描述的例子，如代码清单5-79所示。

代码清单5-79 $realtobits和$bitstoreal系统函数的Verilog HDL描述

```
module driver (net_r);
output net_r;
real r;
wire [64:1] net_r =  $realtobits(r);
endmodule

module receiver (net_r);
input net_r;
wire [64:1] net_r;
real r;
initial assign r =  $bitstoreal(net_r);
endmodule
```

5.10.8 概率分布函数

Verilog HDL提供了系统函数，根据标准的概率函数，返回整数值。

1. $random函数

系统函数$random提供了生成随机函数的机制。每次调用该函数时都会返回一个新的32位随机数。随机函数是有符号整数，它可以是正的或负的。

参数seed控制$random返回的数字，以便不同的种子生成不同的随机流。参数seed为reg类型、integer类型或time类型的变量。在调用$random之前，应将种子值分配给该变量。

【例5.193】 $random系统任务Verilog HDL描述的例子1。

```
$random % b
```

其中，b>0。该例子产生数的范围为[(−b+1)：(b−1)]。

```
reg [23:0] rand;
rand = $random % 60;
```

上面的代码片段产生[−59，59]的随机数。

【例 5.194】 $random 系统任务 Verilog HDL 描述的例子 2。

```
reg [23:0] rand;
rand = { $random} % 60;
```

在该例子中,将并置/连接操作符添加到系统任务 $random,产生[0,59]的随机数。

2. $dist_函数

根据在函数名中指定的概率函数,下列系统函数产生伪随机数。

(1) $dist_uniform (seed,start,end);

(2) $dist_normal(seed,mean,standard_deviation,upper);

(3) $dist_exponential(seed,mean);

(4) $dist_poisson(seed,mean);

(5) $dist_chi_square(seed,degree_of_freedom);

(6) $dist_t(seed,degree_of_freedom);

(7) $dist_erlang(seed,k_stage, mean)。

系统函数的所有参数都是整数值。对于 exponential、poisson、chi-square、t 和 erlang 函数,参数 mean、degree_of_freedom 和 k_stage 应该大于 0。

这些函数中的每个都返回伪随机数,其特征由函数名字描述。换句话说,$dist_uniform 返回在其参数指定的间隔内均匀分布的随机数。

对于每个系统函数,seed 参数都是 inout 参数;也就是说,将一个值传递给函数,并返回一个不同的值。给定相同的种子,系统函数应始终返回相同的值。通过设置使系统的操作可重复执行以方便调试。参数 seed 应该是一个整数变量,由用户初始化,仅由系统函数更新,以确保实现所需要的分布。

在 $dist_uniform 函数中,start 和 end 参数是绑定返回值的整数输入,起始值应小于结束值。

$dist_normal、$dist_exponential、$dist_poisson 和 $dist_erlang 使用的 mean 参数是一个整数输入,它使函数返回的平均值接近指定值。

与 $dist_normal 函数一起使用的 standard_deviation 参数是一个整数输入,有助于确定密度函数的形状。standard_deviation 数字越大,返回值的范围就越大。

与 $dist_chi_square 函数和 $dist_t 函数一起使用的 degree_of_freedom 参数是一个整数输入,有助于确定密度函数的形状。较大的数字将返回值分布在更大的范围内。

5.10.9 命令行输入

在仿真中,获得使用信息的另一种方法是用带有命令的指定信息来调用仿真器。该信息以一个可选参数的格式提供给仿真器,以一个“+”字符开始,这使得这些参数明显区别于其他仿真器参数。

1. $test$plusargs (string)

$test$plusarg 系统函数在 plusargs 列表中搜索用户指定的 plusarg_string。字符串在系统函数的参数中指定为字符串或解释为字符串的非实数变量。按提供的顺序搜索在命令行上出现的 plusargs。如果提供的一个 plusargs 的前缀与提供的字符串中的所有字符匹配,则函数返回一个非零整数。如果命令行中没有与提供的字符串匹配的 plusarg,则函数返回整数零值。

【例 5.195】 $test$plusargs 系统函数 Verilog HDL 描述的例子,如代码清单 5-80 所示。

代码清单 5-80 $test$plusargs 系统函数的 Verilog HDL 描述

```
module test;
 initial begin
   if ( $test$plusargs("HELLO")) $display("Hello argument found.");
```

```
    if ( $test$plusargs("HE")) $display("The HE subset string is detected.");
    if ( $test$plusargs("H")) $display("Argument starting with H found.");
    if ( $test$plusargs("HELLO_HERE")) $display("Long argument.");
    if ( $test$plusargs("HI")) $display("Simple greeting.");
    if ( $test$plusargs("LO")) $display("Does not match.");
  end
endmodule
```

在 ModelSim 主界面中，选择 Simulate→Start Simulation，弹出 Start Simulation 对话框，如图 5.17 所示，单击 Verilog 选项卡，在 User Defined Arguments(＋＜plusarg＞)文本框中输入＋HELLO。

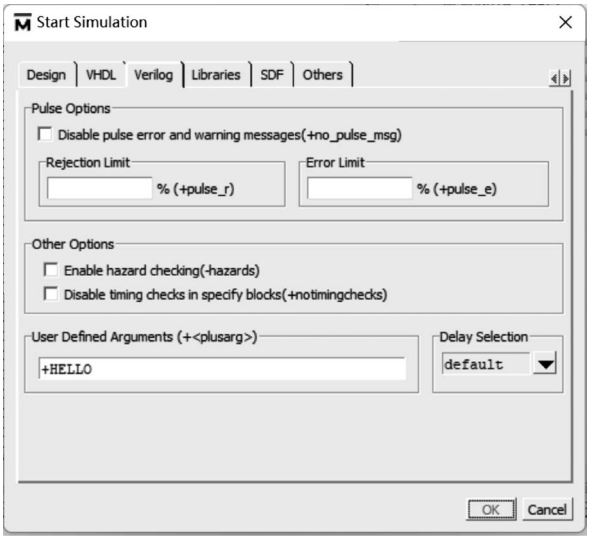

图 5.17　在 Simulation 标签界面下添加选项

使用 ModelSim 软件对该设计执行行为级仿真，在 Transcript 窗口中输出的结果如图 5.18 所示。

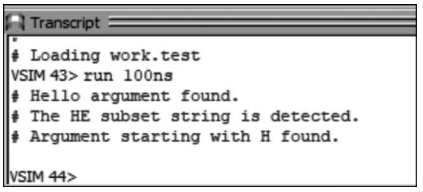

图 5.18　执行行为级仿真的输出结果

2. $value$plusargs (user_string, variable)

$value$plusargs 系统函数在 plusargs 列表(如 $test$plusargs 系统函数)中查找用户定义的 plusarg_string。系统函数内的第一个参数指定的字符串作为一个字符串或者一个非实数变量。该字符串不包含命令行参数前面的“＋”号。在命令行所提供的 plusargs，按照所提供的顺序进行查找。如果提供的 plusargs 中一个前缀匹配所提供字符串的所有字符，则函数返回一个非零的整数，字符串的剩余部分转换为 use_string 内指定的类型，结果值保存在所提供的变量中。如果没有找到匹配的字符串，函数返回一个整数 0，不修改所提供的变量。当函数返回 0 的时候，不产生警告信息。

user_string 是下面的格式“plusarg_stringformat_string”。格式化字符串和 $display 系统任务一样。下面是合法的格式。

(1) %d，十进制转换。

(2) %o，八进制转换。

(3) %h，十六进制转换。

(4) %b，二进制转换。

(5) %e,实数指数转换。

(6) %f,实数十进制转换。

(7) %g,实数十进制或指数转换。

(8) %s,字符串(没有转换)。

来自 plusargs 列表的第一个字符串提供给仿真器,匹配 user_string 指定的 plusarg_string 部分,将是用于转换的可用 plusarg 字符串。匹配 plusarg 的剩余字符串将从一个字符串转换为格式字符串指定的格式,并保存在所提供的变量中。如果没有剩余的字符串,则保存到变量的值为 0 或者为空的字符串值。

如果变量的位宽大于转换后的值,则在保存的值前面补零。如果变量不能保留转换后的值,则把值截断。如果值是负数,则认为值大于所提供的变量。如果在字符串中用于转换的字符是非法的,则将变量的值设置为'bx。

【例 5.196】 $value$plusargs 系统函数 Verilog HDL 描述的例子,如代码清单 5-81 所示。

代码清单 5-81 $value$plusargs 系统函数的 Verilog HDL 描述

```
`define STRING reg [1024 * 8:1]
module goodtasks;
  `STRING str;
  integer int;
  reg [31:0] vect;
  real realvar;
  initial
    begin
      if ( $value$plusargs("TEST = %d",int))
        $display("value was %d",int);
      else
        $display("+TEST = not found");
      #100 $finish;
    end
  endmodule
```

在 ModelSim 软件的 Start Simulation 对话框中,单击 Verilog 选项卡,在 User Defined Arguments (+<plusarg>)文本框中输入

```
+TEST=123
```

对该设计执行行为级仿真的输出结果如下:

```
value was          123
```

【例 5.197】 $value$plusargs 命令行 Verilog HDL 描述的例子,如代码清单 5-82 所示。

代码清单 5-82 $value$plusargs 命令行 Verilog HDL 描述

```
module ieee1364_example;
  real frequency;
  reg [8 * 32:1] testname;
  reg [64 * 8:1] pstring;
  reg clk;
  initial
    begin
      if ( $value$plusargs("TESTNAME = %s",testname))
        begin
          $display(" TESTNAME = %s.",testname);
          $finish;
        end
      if (!( $value$plusargs("FREQ + %0F",frequency)))
          frequency = 8.33333;      //166 MHz
          $display("frequency = %f",frequency);
```

```
        pstring = "TEST%d";
    if ( $value$plusargs(pstring, testname))
        $display("Running test number %0d.",testname);
    end
endmodule
```

(1) 在 ModelSim 软件的 Start Simulation 对话框中,单击 Verilog 选项卡,在 User Defined Arguments(+<plusarg>)文本框中输入

```
+TESTNAME=bar
```

对该设计执行行为级仿真的输出结果如下:

```
TESTNAME=        bar.
```

(2) 在 ModelSim 软件的 Start Simulation 对话框中,单击 Verilog 选项卡,在 User Defined Arguments(+<plusarg>)文本框中输入

```
+FREQ+9.234
```

对该设计执行行为级仿真的输出结果如下:

```
frequency = 9.234000
```

(3) 在 ModelSim 软件的 Start Simulation 对话框中,单击 Verilog 选项卡,在 User Defined Arguments(+<plusarg>)文本框中输入

```
+TEST23
```

对该设计执行行为级仿真的输出结果如下:

```
frequency = 8.333330
Running test number 23.
```

5.10.10 数学函数

Verilog HDL 提供了整数和实数数学函数,数学系统函数可以用在常数表达式中。

1. 整数数学函数

【例 5.198】 整数数学函数 Verilog HDL 描述的例子。

```
integer result;
result = $clog2(n);
```

系统函数 $clog2 将返回基 2 对数的计算结果。参数可以是一个整数或任意宽度的向量值。将参数看作无符号的数。如果参数值为 0,则产生的结果也为 0。

该系统函数可用于寻址给定大小存储器所需的最小地址宽度或表示给定数量的状态所需要的最大向量宽度。

2. 实数数学函数

表 5.42 给出了 Verilog 到 C 实数数学函数的交叉列表,这些函数接受实数参数,返回实数结果。这些行为匹配等效的 C 语言标准数学库函数。

表 5.42 实数数学函数

Verilog 函数	等效的 C 函数	描 述
$ln(x)	log(x)	自然对数
$log10(x)	log10(x)	基 10 对数
$exp(x)	exp(x)	指数函数
$sqrt(x)	sqrt(x)	平方根

续表

Verilog 函数	等效的 C 函数	描　　述
$pow(x,y)	pow(x,y)	x 的 y 次幂
$floor(x)	floor(x)	向下舍入
$ceil(x)	ceil(x)	向上舍入
$sin(x)	sin(x)	正弦
$cos(x)	cos(x)	余弦
$tan(x)	tan(x)	正切
$asin(x)	asin(x)	反正弦
$acos(x)	acos(x)	反余弦
$atan(x)	atan(x)	反正切
$atan2(x,y)	atan2(x,y)	(x/y)反正切
$hypot(x,y)	hypot(x,y)	(x*x+y*y)的平方根
$sinh(x)	sinh(x)	双曲正弦
$cosh(x)	cosh(x)	双曲余弦
$tanh(x)	tanh(x)	双曲正切
$asinh(x)	asinh(x)	反双曲正弦
$acosh(x)	acosh(x)	反双曲余弦
$atanh(x)	atanh(x)	反双曲正切


视频讲解

5.10.11　设计实例七：只读存储器初始化和读操作的实现

在计算机系统中，通常需要事先将以二进制表示的“机器指令”保存到程序存储器中。当给计算机系统上电后，在处理器时钟和处理器内程序计数器(program counter，PC)的共同驱动下，从程序存储器中取出“机器指令”。在该设计中模拟了这个过程。在只读存储器(read only memory，ROM)中事先保存了以十六进制表示的数据，如代码清单 5-83 所示。

代码清单 5-83　text.txt 文件

```
0200A
00300
08101
04000
08601
0233A
00300
08602
02310
0203B
08300
04002
08201
00500
04001
02500
00340
00241
04002
08300
08201
00500
08101
00602
04003
```

```
0241E
00301
00102
02122
02021
00301
00102
02222
04001
00342
0232B
00900
00302
00102
04002
00900
08201
02023
00303
02433
00301
04004
00301
00102
02137
02036
00301
00102
02237
04004
00304
04040
02500
02500
02500
0030D
02341
08201
0400D
```

在该文件中，每个十六进制数占用一行，每个十六进制数等效的二进制数为 20 位。一共 64 个数据，因此，需要的 ROM 的最小容量是 64×20 位。由于深度为 64，所以需要访问这 64 行数据所需要的最小的地址线的位数为 6 位。

实现访问该 ROM 的 Verilog HDL 代码，如代码清单 5-84 所示。

代码清单 5-84　rom.v 文件

```
module rom(                          //module 关键字定义模块 rom
    input en,                        //input 关键字定义输入端口 en
    input [5:0] addr,                //input 关键字定义输入端口 addr,宽度为 6
    input clk,                       //input 关键字定义输入端口 clk
    output reg[19:0] data            //output 和 reg 关键字定义输出端口 data,宽度为 20
    );
parameter ADDRESS_WIDTH = 6;         //parameter 关键字定义参数 ADDRESS_WIDTH
localparam rom_depth = 2 ** ADDRESS_WIDTH - 1; //localparam 关键字定义本地参数 rom_depth
/ ************** 属性 rom_style 声明使用 FPGA 内的 BRAM 实现 memory 结构 ************ /
  ( * rom_style = "block" * ) reg [19:0] memory[0:rom_depth];  //声明 memory 宽度和深度
 initial                             //initial 关键字定义初始化部分
 begin                               //begin 关键字标识初始化部分的开始,类似 C 语言的"{"
       $readmemh("text.txt", memory); //调用 readmemh 读取 text.txt 文件以初始化 memory
 end                                 //end 关键字标识初始化部分的结束,类似 C 语言的"}"
```

```
always @(posedge clk)                    //always 关键字定义过程语句,敏感信号 clk 上升沿
begin                                    //begin 关键字标识过程语句的开始,类似 C 语言的"{"
    if(en)                               //if 关键字定义条件语句,如果 en 为逻辑"1",成立
        data <= memory[addr];            //从地址 addr 指向的 memory 单元中读取数据 data
end                                      //end 关键字标识过程语句的结束,类似 C 语言的"}"
endmodule                                //endmodule 关键字标识模块 rom 的结束
```

在高云云源软件中,对代码清单 5-84 给出的代码进行设计综合,生成综合后的电路结构如图 5.19 所示。

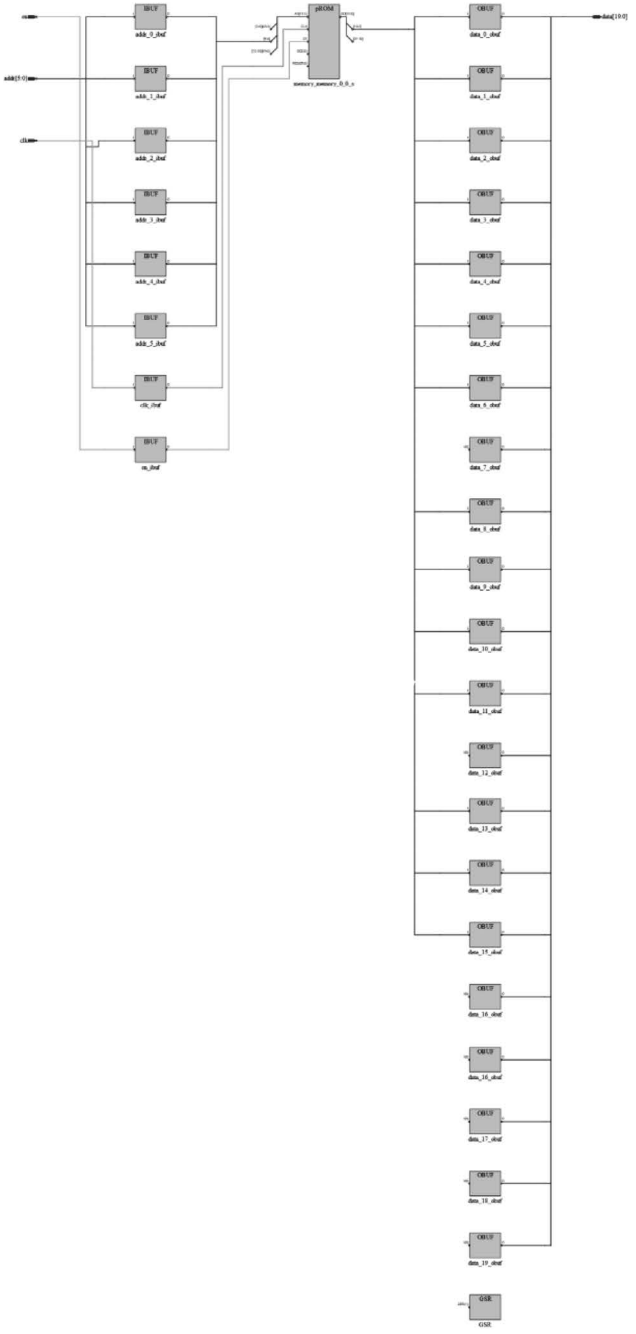

图 5.19　综合后生成的电路结构

注：在配套资源\eda_verilog\example_5_16 目录下，用高云云源软件打开 example_5_16.gprj。

使用 ModelSim 对该设计进行综合后仿真的代码，如代码清单 5-85 所示。

代码清单 5-85　test.v 文件

```
`timescale 1ns / 1ps                    // `timescale 预编译指令定义时间精度/分辨率
module test;                            // module 关键字定义模块 test
reg [5:0] addr;                         // reg 关键字定义 reg 类型变量 addr,宽度为 6 位
reg en;                                 // reg 关键字定义 reg 类型变量 en
reg clk;                                // reg 关键字定义 reg 类型变量 clk
wire [19:0] data;                       // wire 关键字定义网络 data,宽度为 20 位
rom Inst_rom(en,addr,clk,data);         // 调用/例化元件 rom,并将其例化为 Inst_rom,采用端口位置关联

initial                                 // initial 关键字定义初始化部分
begin                                   // begin 关键字标识初始化部分的开始,类似 C 语言的"{"
  clk = 1'b0;                           // clk 分配/赋值逻辑"0"
  en = 1'b1;                            // en 分配/赋值逻辑"1"
  addr = 0;                             // addr 分配/赋值"0"
end                                     // end 关键字标识初始化部分的结束,类似 C 语言的"}"

always                                  // always 定义过程语句
begin                                   // begin 关键字标识过程语句的开始,类似 C 语言的"{"
  #10 clk = ~clk;                       // 每隔 10ns,clk 取反,为 clk 生成方波信号
end                                     // end 关键字标识过程语句的结束,类似 C 语言的"}"

initial                                 // initial 关键字定义初始化部分
begin                                   // begin 关键字标识初始化部分的开始,类似 C 语言的"{"
  while(1)                              // while 关键字定义无限循环
    begin                               // begin 关键字标识无限循环的开始,类似 C 语言的"{"
      #20;                              // 持续 20ns,ns 由`timescale 定义
      $display("the content of addr( %d) = %h",addr,data); // 调用系统任务 $display
      addr = addr + 1;                  // 变量 addr 递增
    end                                 // end 关键字标识无限循环的结束,类似 C 语言的"}"
end                                     // end 关键字标识初始化部分的结束,类似 C 语言的"}"
endmodule                               // endmodule 标识模块 test 的结束
```

在 ModelSim 软件中执行综合后仿真，在 Transcript 窗口中打印的一部分信息如图 5.20 所示。在 Wave 窗口中显示的波形如图 5.21 所示。

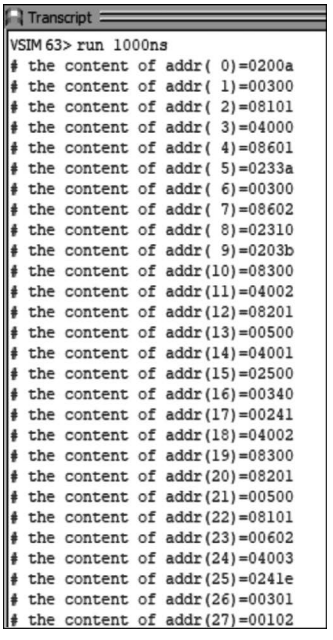

图 5.20　在 Transcript 窗口打印的一部分信息

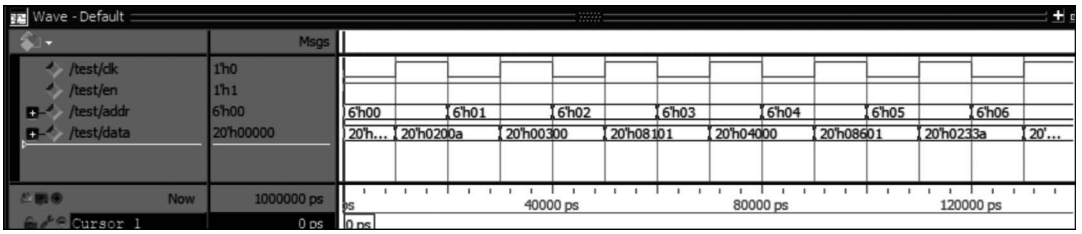

图 5.21　在 Wave 窗口中显示的波形

注：配套资源\eda_verilog\example_5_17 目录下，用 ModelSim SE-64 10.4c 打开 postsynth_sim.mpf。

5.11　Verilog HDL 编译器命令

所有的 Verilog 编译器命令前面都有字符“`”，该符号称为重音符(ASCII 码值为 0x60)。它与撇号字符“'”(ASCII 码值为 0x27)不同。编译器命令的作用域从处理它的点开始，跨越所有处理的文件，一直延伸到另一个编译器命令取代它或处理完成的点。

5.11.1　`celldefine 和 `endcelldefine

这两个命令用于将模块标记为单元模块，它们表示包含模块定义。某些 PLI 使用单元模块用于这些应用，如计算延迟。推荐这两个命令配对使用，但不是必需的。源代码中任一命令的出现控制模块是否标记为单元模块。

该命令可以出现在源代码描述中的任何地方。但是，推荐将其放在模块定义的外部。ModelSim 综合工具忽略这两个命令。

5.11.2　`default_nettype

该命令用于为隐含网络指定网络类型，也就是为那些没有被说明的连线定义网络类型。它只可以出现在模块声明的外部，允许多个`default_netype 命令。

如果没有出现`default_netype 命令，或者如果指定了`resetall 命令，则隐含的网络类型是 wire。当`default_netype 设置为 none 时，需要明确地声明所有网络；如果没有明确地声明网络，则产生错误。`default_netype 命令格式如下：

```
`default_nettype default_nettype_value
```

其中，default_nettype_value 的值可以是 wire、tri、tri0、tri1、wand、triand、wor、trior、trireg、uwire 和 none。

5.11.3　`define 和 `undef

提供了一个文本宏替换工具，以便可以使用有意义的名字来表示常用的文本片段。例如，在整个描述中重复使用常数的情况下，如果需要修改常数的值，则文本宏非常有用，因为源描述中只有一个地方需要修改。文本宏功能不受编译器命令`resetall 的影响。

1. `define 命令

命令`define 为文本替换创建宏。语法格式如下：

```
`define text_macro_name macro_text
```

该命令可以在模块定义的内部和外部使用。定义文本宏之后,可以在源描述中使用字符“`”,后跟宏名字。编译器应将宏的文本替换为字符串 text_macro_name 以及后面的任何实际参数。所有编译器的命令应看作预定义的宏名字,将编译器命令重新定义为宏名字是非法的。

可以使用参数定义文本宏,这允许为每次使用单独定制宏。

宏文本可以是与文本宏名字在同一行上指定的任意文本。如果需要多行指定文本,则新的一行前面应加上反斜线符号“\”。第一个没有反斜线的新的一行应结束宏文本。由反斜线符号“\”开头的新的一行也应该以带有换行符的扩展宏形式替换(但是没有前面的反斜线符号“\”)。

当使用形参定义文本宏时,形参的范围应扩展到宏文本的末尾。在宏文本中可以以与标识符相同的方式使用形参。

如果使用了形参,则形参名字列表应包含在宏名字后面的括号中。正式参数的名字应该是简单标识符,用字符逗号和可选的空格分隔。左侧括号应紧跟在文本宏名字之后,中间没有空格。

如果文本中包含单行注释(即用字符“//”指定的注释),则该注释不应成为替换文本的一部分。宏文本可以为空,在这种情况下,将文本宏定义为空,并且在使用宏时不替换任何文本。

使用文本宏的语法格式如下:

```
`text_macro_identifier [ ( list_of_actual_arguments ) ]
```

对于没有参数的宏,每次出现`text_marco_name 时,都应按原样替换文本。但是,具有一个或多个参数的文本,应用每个形参替换为宏使用中用作实际参数的表达式来扩展。

要使用由参数定义的宏,文本宏的名字后面应加上括号中的实际参数列表,并用逗号分隔。文本宏名字和左括号之间允许空白。实际参数的个数应该匹配形参的个数。

一旦定义了文本宏名字,就可以在源描述中的任何位置使用它,也就是说,没有范围限制,可以交互定义和使用文本宏。

为宏文本指定的文本不能拆分为以下词汇标记,包括注释、数字、字符串、标识符、关键字和操作符。

【例 5.199】 `define 命令 Verilog HDL 描述的例子 1。

```
`define wordsize 8
reg [1:`wordsize] data;
//define a nand with variable delay
`define var_nand(dly) nand #dly
`var_nand(2) g121 (q21, n10, n11);
`var_nand(5) g122 (q22, n10, n11);
```

【例 5.200】 `define 命令 Verilog HDL 非法描述的例子 2。

```
`define first_half "start of string
$display(`first_half end of string");
```

该例子是非法的,因为它是跨字符串拆分的。

【例 5.201】 `define 命令 Verilog HDL 非法描述的例子 3。

```
`define max(a,b)((a) > (b) ? (a) : (b))
n = `max(p+q, r+s);
```

将要扩展为

```
n = ((p+q) > (r+s)) ? (p+q) : (r+s);
```

每个实际参数都被字面上对应的形参取代。因此,当表达式用作实参时,表达式将被全部替换。如果在宏文本中使用了形参,这会导致表达式多次求值。

2. `undef 命令

`undef 命令用于取消前面定义的宏。如果先前并没有使用命令`define 进行宏定义，那么使用`undef 命令将会导致一个警告。`undef 命令的语法格式如下：

```
`undef text_macro_identifier
```

一个未定义的文本宏没有值，就如同它从未被定义。

5.11.4 `ifdef、`else、`elsif、`endif 和`ifndef

1. `ifdef 编译器命令

这些条件编译器命令用于在编译期间可选地包括 Verilog HDL 源描述的行。`ifdef 编译器命令检查文本宏的定义。如果定义了文本宏的名字，则包含`ifdef 命令后面的行。如果未定义文本宏的名字，并且存在`else 命令，则编译此源代码。`ifndef 编译器命令检查文本宏的定义。如果未定义文本宏的名字，则包含`ifdef 命令后面的行。如果定义文本宏的名字，并且存在`else 命令，则编译此源代码。

如果存在`elsif 命令(而不是`else)，编译器将检查文本宏名字的定义。如果存在名字，则包含`elsif 命令后面的行。`elsif 命令等效于编译器命令系列`else `ifdef…`endif。该命令不需要相应的`endif 命令。该命令前面应加上`ifdef 或`ifndef 命令。

这些命令可能出现在源描述中的任何位置。`ifdef、`else、`elsif、`endif 和`ifndef 编译器命令可能有用的情况包括：选择模块的不同表示形式，如行为级、结构级或开关级；选择不同的时序或结构信息；为一个给定运行选择不同的激励源。

一个完整的`ifdef、`elsif、`else 和`endif 编译器命令的语法如下：

```
`ifdef text_macro_identifier
    ifdef_group_of_lines
`elsif text_macro_identifier
    elsif_group_of_lines
`else
    else_group_of_lines
`endif
```

一个完整的`ifndef、`elsif、`else 和`endif 编译器命令的语法如下：

```
`ifndef text_macro_identifier
    ifndef_group_of_lines
`elsif text_macro_identifier
    elsif_group_of_lines
`else
    else_group_of_lines
`endif
```

其中，text_macro_identifier 是 Verilog HDL 标识符，ifdef_group_of_lines、ifndef_group_of_lines、elsif_group_of_lines 和 else_group_of_lines 是 Verilog HDL 源代码描述的一部分，`else 和`elsif 编译器命令和所有的 group_of_lines 都是可选的。

`ifdef、`elsif、`else 和`endif 编译器命令以以下方式协同工作。

(1) 当遇到`ifdef 时，将测试`ifdef 对应的 text_macro_identifier，以查看它是否在 Verilog HDL 源描述中使用了`define 定义为文本宏名字。

(2) 如果定义了`ifdef 对应的 text_macro_identifier，则将 ifdef_group_of_lines 编译为描述的一部分；如果存在`else 或`elsif 编译器命令，则忽略这些编译器命令和相应的 group_of_lines。

(3) 如果未定义`ifdef 对应的 text_macro_identifier，则忽略 ifdef_group_of_lines。

(4) 如果有`elsif 编译器命令，则测试`elsif 对应的 text_macro_identifier，以查看它是否

在 Verilog HDL 源描述中使用了`define 定义为文本宏名字。

(5) 如果定义了`elsif 对应的 text_macro_identifier,则将 elsif_group_of_lines 编译为描述的一部分;如果存在其他`elsif 或`else 编译器命令,则忽略这些编译器命令和相应的 group_of_lines。

(6) 如果尚未定义第一个`elsif 对应的 text_macro_identifier,则忽略第一个 elsif_group_of_lines。

(7) 如果有多个`elsif 编译器命令,则按照 Verilog HDL 源描述中的顺序,像第一个`elsif 编译器命令一样对它们进行评估。

(8) 如果有`else 编译器命令,将 else_group_of_lines 作为描述的一部分进行编译。

`ifndef、`elsif、`else 和`endif 编译器命令以以下方式协同工作。

(1) 当遇到`ifndef 时,将测试`ifndef 对应的 text_macro_identifier,以查看它是否在 Verilog HDL 源描述中使用了`define 定义为文本宏名字。

(2) 如果未定义`ifndef 对应的 text_macro_identifier,则将 ifndef_group_of_lines 编译为描述的一部分;如果存在`else 或`elsif 编译器命令,则忽略这些编译器命令和相应的 group_of_lines。

(3) 如果定义了`ifndef 对应的 text_macro_identifier,则忽略 ifndef_group_of_lines。

(4) 如果有`elsif 编译器命令,则测试`elsif 对应的 text_macro_identifier,以查看它是否在 Verilog HDL 源描述中使用了`define 定义为文本宏名字。

(5) 如果定义了`elsif 对应的 text_macro_identifier,则将 elsif_group_of_lines 编译为描述的一部分;如果存在其他`elsif 或`else 编译器命令,则忽略这些编译器命令和相应的 group_of_lines。

(6) 如果尚未定义第一个`elsif 对应的 text_macro_identifier,则忽略第一个 elsif_group_of_lines。

(7) 如果有多个`elsif 编译器命令,则按照 Verilog HDL 源描述中的顺序,像第一个`elsif 编译器命令一样对它们进行评估。

(8) 如果有`else 编译器命令,将 else_group_of_lines 作为描述的一部分进行编译。

尽管编译器命令的名字包含在与文本宏名字相同的名字空间中,但编译器命令的名字被看作不是由`ifdef、`ifndef 和`elsif 定义的。

应允许嵌套`ifdef、`ifdef、`else、`elsif 和`endif 编译器命令。

编译器忽略的任何 group_of_lines 仍应遵循 Verilog HDL 的空白、注释、数字、字符串、标识符、关键字和操作符约定。

【例 5.202】 `ifdef 命令 Verilog HDL 描述的例子 1,如代码清单 5-86 所示。

代码清单 5-86 `ifdef 命令的 Verilog HDL 描述

```
module and_op (a, b, c);
output a;
input b, c;
`ifdef behavioral
    wire a = b & c;
`else
    and a1 (a,b,c);
`endif
endmodule
```

该例子中给出了用于条件编译的`ifdef 命令的简单用法。如果定义了标识符 behavioral,则将编译一个连续的网络分配/赋值;否则,将 and 门例化为 a1。

【例 5.203】 嵌套`ifdef 命令 Verilog HDL 描述的例子 2，如代码清单 5-87 所示。

代码清单 5-87 嵌套`ifdef 命令的 Verilog HDL 描述

```
module test(out);
output out;
`define wow
`define nest_one
`define second_nest
`define nest_two
  `ifdef wow
    initial $display("wow is defined");
    `ifdef nest_one
      initial $display("nest_one is defined");
      `ifdef nest_two
        initial $display("nest_two is defined");
      `else
        initial $display("nest_two is not defined");
      `endif
    `else
        initial $display("nest_one is not defined");
    `endif
  `else
      initial $display("wow is not defined");
      `ifdef second_nest
          initial $display("second_nest is defined");
      `else
          initial $display("second_nest is not defined");
      `endif
  `endif
endmodule
```

【例 5.204】 条件编译命令嵌套 Verilog HDL 描述的例子，如代码清单 5-88 所示。

代码清单 5-88 条件编译命令嵌套的 Verilog HDL 描述

```
module test;
  `ifdef first_block
    `ifndef second_nest
        initial $display("first_block is defined");
    `else
        initial $display("first_block and second_nest defined");
    `endif
  `elsif second_block
      initial $display("second_block defined, first_block is not");
  `else
      `ifndef last_result
          initial $display("first_block, second_block,"
            " last_result not defined.");
      `elsif real_last
          initial $display("first_block, second_block not defined,"
            " last_result and real_last defined.");
      `else
          initial $display("Only last_result defined!");
      `endif
  `endif
endmodule
```

5.11.5 `include

文件包含(`include)编译器命令用于在编译期间将源文件的内容插入到另一个文件中，结果就好像包含的源文件内容出现在`include 编译器命令的地方。`include 编译器命令可用于包含全局或通用的定义和任务，而没有在模块边界内封装重复代码。

使用`include编译器命令的优势在于：提供配置管理的组成部分；改进Verilog HDL源文件描述的组织；便于维护Verilog HDL源文件描述。

`include编译器命令的语法如下：

```
`include "filename"
```

编译器命令`include可以在Verilog HDL描述中的任何位置指定。filename是要包含在源文件中的文件名字。filename可以是完整或相对路径名字。

只有空格或注释可以与`include编译器命令出现在同一行。

使用`include编译器命令包含在源文件中的文件可能包含其他`include编译器命令。包含文件的嵌套层数是有限的。

【例5.205】 `include命令Verilog HDL描述的例子。

```
`include "parts/count.v"
`include "fileB"
`include "fileB"                    //包含 fileB
```

5.11.6 `resetall

该编译器遇到`resetall命令时，会将所有的编译命令重新设置为默认值。推荐在源文件的开始放置`resetall。将`resetall命令放置在模块内或者UDP声明中是非法的。

5.11.7 `line

对于Verilog工具来说，跟踪Verilog源文件的名字和文件中的行号是非常重要的，这些信息可以用于调试错误消息或者源代码，Verilog PL1可以访问它。

然而，在很多情况下，Verilog源文件由其他工具进行了预处理。由于预处理工具可能在Verilog HDL源文件中添加了额外的行，或者将多个源代码行合并为一个行，或者并置多个源文件，等等，可能会丢失原始的源文件和行信息。

`line编译器命令可以用于指定的原始源代码的行号和文件名。如果其他过程修改了源文件，这允许定位原始的源文件。当指定了新行的行号和文件名时，编译器就可以正确地定位原始的源文件位置。然而，这要求相应的工具不产生`line命令。

编译器应保持正在编译的文件的当前行号和文件名。`line命令应该设置为跟随在命令中所指定的行号和文件名。该命令可以在Verilog HDL源文件描述中的任何位置指定。然而，只有空白与`line命令出现在同一行，注释不允许与`line命令位于同一行。`line命令中的所有参数都是必需的。该命令的结果不受`resetall命令的影响。

其语法格式如下：

```
`line number "filename" level
```

其中，number是一个正整数，用于指定跟随文本行的新行行号，filename是一个字符串常数，将其看作文件的新名字，filename可以是完整路径名或者相对路径名；level参数的值可以是0、1或2：①当为1时，输入一个include行后的下面一行是第一行；②当为2时，退出一个include行后的下面一行是第一行；③当为0时，指示任何其他行。

【例5.206】 `line命令Verilog HDL描述的例子。

```
`line3 "orig.v" 2
//该行是 orig.v 存在 include 文件后的第 3 行
```

当编译器处理文件的剩余部分和新文件时，读取每一行时，行号应递增，并且名字应更新

为当前正在处理的新文件。每个文件开头的行号应重置为1。开始读取包含文件时,应保存当前行和文件名,以便在包含文件结束时恢复。更新的行号和文件名信息应可用于PLI访问。库搜索的机制不受`line编译器命令的影响。

5.11.8 `timescale

该命令指定了时间单位和跟随它的模块的时间精度。时间单位是时间值(如仿真时间和延迟值)的测量单位。

要在同一设计中使用具有不同时间单位的模块,下面的时间标度结构非常有用。

(1) `timescale编译器命令,用于指定设计中模块中的时间度量单位和时间精度。

(2) $printtimescale系统任务,用于显示模块的时间单位和精度。

(3) $time和$realtime系统函数,$timeformat系统任务和%t格式规范用于指定如何报告时间信息。

`timescale编译器命令指定时间和延迟值的测量单位,以及在读取另一个`timescale编译器命令之前,遵循该命令的所有模块中延迟的精度。如果没有指定`timescale或已通过`resetall命令复位,时间单位和精度由仿真器指定。如果某些模块指定了`timescale,而其他模块没有指定,则会出现错误。

`timescale编译器命令格式如下:

```
`timescale time_unit/time_precision
```

其中,time_unit参数指定时间和延迟的度量单位。time_precision参数指定在仿真中使用延迟值之前如何四舍五入。即使设计中其他地方有更小的time_precision参数,使用的值也精确到此处指定的时间单位内,设计中所有`timescale编译器命令中最小的time_precision参数决定了仿真时间单位的精度。

time_precision参数至少应与time_unit参数一样准确,它不能指定比time_unit更长的单位时间。

这些参数中的整数值指定值大小的数量级,有效的整数为1、10和100。字符串表示度量单位,有效的字符串为s、ms、us、ns、ps和fs。

【例5.207】 `timescale命令Verilog HDL描述的例子1。

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

在该例子中,由于time_unit参数为1ns,所以命令后面的模块中的所有时间值都是1ns的倍数。由于time_precision参数为1ps或千分之一纳秒,所以延迟四舍五入为带三位小数的实数,或精确到千分之一纳秒以内。

【例5.208】 `timesscale命令Verilog HDL描述的例子2。

```
`timescale 10us / 100ns
```

在该例子中,由于time_unit参数为10μs,所以命令后面的模块中的所有时间值都是10μs的倍数。由于time_precision参数为100ns或十分之一微秒,所以延迟四舍五入到十分之一微秒以内。

【例5.209】 `timescale命令Verilog HDL描述的例子3。

```
`timescale 10ns / 1ns
module test;
reg set;
parameter d = 1.55;
```

```
initial begin
    #d set = 0;
    #d set = 1;
end
endmodule
```

根据时间精度,参数 d 的值从 1.55 四舍五入到 1.6。模块的时间单位是 10ns,精度是 1ns。因此,参数 d 的延迟从 1.6 标定到 16。

5.11.9 `unconnected_drive 和 `nounconnected_drive

当一个模块所有未连接的端口出现在`unconnected_drive 和`nounconnected_drive 命令之间时,将这些未连接的端口上拉或者下拉,而不是按通常的默认值处理。

命令`unconnected_drive 使用 pull1/pull0 参数中的一个:当指定 pull1 时,所有未连接的端口自动上拉;当指定 pull0 时,所有未连接的端口自动下拉。

建议成对使用`unconnected_drive 和`nounconnected_drive 命令,但不是强制要求。这些命令在模块外部成对指定。

`resetall 命令包括`nounconnected_drive 命令的效果。

5.11.10 `pragma

`pragma 命令是一个结构化的说明,它用于改变对 Verilog HDL 源文件的理解。由这个命令所引入的说明称为编译指示。编译指示不同于 Verilog HDL 标准所指定的结果,它为指定实现的结果。其语法格式如下:

```
`pragma pragma_name [ pragma_expression { , pragma_expression } ]
```

pragma 规范由跟在`pragma 命令后面的 pragma_name 名字标识。pragma_name 后面是 pragma_expression 的可选列表,用于限定更改的 pragma_name 指示的解释。除非另有规定,否则未被实现识别的 pragma_name 的 pragma 命令不会对 Verilog HDL 源文本的解释产生影响。

reset 和 resetall 编译指示应恢复受编译指示影响的相关 pragma_keywords 的默认值和状态,这些默认值应为工具在处理任何 Verilog 文本之前定义的值。reset 编译指示应复位所有 pragma_keyword 出现的所有 pragma_name 的状态。resetall 编译指示应复位所有由实现识别的 pragma_name 的状态。

5.11.11 `begin_keywords 和 `end_keyword

`begin_keywords 和`end_keyword 命令用于指定在一个源代码块中,基于不同版本的 IEEE Std1364 标准,确定用于关键字的保留字。该对命令只指定那些作为保留关键字的标识符。只能在设计元素(模块、原语和配置)外指定该关键字,并且需要成对使用。其语法格式如下:

```
`begin_keywords "version_specifier"
    …
`end_keyword
```

其中,version_specifier 为可选的参数,包括 1364-1995、1364-2001、1364-2001-noconfig 和 1364-2005。

【例 5.210】 `begin_keywords 和`end_keyword 命令 Verilog HDL 描述的例子。

```
`begin_keywords "1364 - 2001"        //使用 IEEE Std 1364 - 2001Verilog 关键字
module m2 (...);
wire [63:0] uwire;                   //uwire 不是 1364 - 2001 的关键字
…
endmodule
`end_keywords
```