

第6章 结构语句、系统任务、函数 语句和显示系统任务

6.1 结构说明语句

Verilog语言中的任何过程模块都从属于以下四种结构的说明语句。

(1) initial说明语句

(2) always说明语句

(3) task说明语句

(4) function说明语句

`initial`和`always`说明语句在仿真的一开始即开始执行。

`initial`语句只执行一次。相反，`always`语句则是不断地重复执行，直到仿真过程结束。

`always`语句后面跟着的过程块是否运行，则要看它的触发条件是否满足，如满足则运行过程块一次，再次满足再运行一次，直至仿真过程结束。

在一个模块中，使用`initial`和`always`语句的次数是不受限制的。

`task`和`function`语句可以在程序模块中的一处或多处调用。

6.1.1 initial语句

initial语句的格式如下:

```
initial  
begin  
    语句1;  
    语句2;  
    .....  
    语句n;  
end
```

例6.1 用initial 块对存储器变量赋初始值

```
initial
begin
    areg=0; //初始化寄存器areg
    for (index=0; index<size; index=index+1)
        memory[index]=0; //初始化一个memory
end
```


例6.2 用initial语句来生成激励波形

```
initial
begin
    inputs = 'b000000; //初始时刻为0
    #10 inputs = 'b011001;
    #10 inputs = 'b011011;
    #10 inputs = 'b011000;
    #10 inputs = 'b001000;
end
```

always语句由于其不断活动的特性，只有和一定的时序控制结合在一起才有用。

如果一个always语句没有时序控制，则这个always语句将会使仿真器产生死锁。

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

但如果加上时序控制，则这个always语句将变为一条非常有用的描述语句。

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

例6.5

```
reg[7:0] counter;  
    reg tick;  
    always @(posedge areg)  
        begin  
            tick = ~tick;  
            counter = counter + 1;  
        end
```


always 的时间控制可以是沿触发也可以是电平触发的，可以单个信号也可以多个信号，中间需要用关键字 or 连接。

```
always @(posedge clock or posedge reset)
```

```
begin
```

```
...
```

```
end
```

```
always @(a or b or c)
```

```
begin
```

```
...
```

```
end
```

边沿触发的always块常常描述时序行为，如有限状态机。如果符合可综合风格要求，则可通过综合工具自动地将其转换为表示寄存器组和门级组合逻辑的结构，而该结构应具有时序所要求的行为

电平触发的always块常常用来描述组合逻辑的行为。如果符合可综合风格要求，可通过综合工具自动将其转换为表示组合逻辑的门级逻辑结构或带锁存器的组合逻辑结构，而该结构应具有所要求的行为。

一个模块中可以有多多个always块，它们都是并行运行的。

always块的or事件控制

有时，多个信号或者事件中任意一个发生的变化都能触发语句或语句块的执行。在Verilog语言中，可以使用“或”表达式来表示这种情况。由关键词“or”连接的多个事件名或信号名组成的列表称为敏感列表。关键词“or”被用来表示这种关系。

Verilog 1364-2001版本的语法中，对于原来的规定作了补充：关键词“or”也可以使用“,”来代替。使用“,”来代替关键词“or”也适用于跳变沿敏感的触发器。

//有异步复位的电平敏感锁存器

```
always @ ( reset or clock or d )
```

//等待复位信号reset 或 时钟信号clock 或 输入信号d 的改变

```
begin
```

```
    if ( reset )                //若 reset 信号为高，把q置零
```

```
        q = 1'b0 ;
```

```
    else if ( clock ) //若clock 信号为高，锁存输入信号d
```

```
        q = d ;
```

```
end
```


//有异步复位的电平敏感锁存器

```
always @ ( reset , clock , d )
```

//等待复位信号reset 或 时钟信号clock 或 输入信号d的改变

```
begin
```

```
    if ( reset )    // 若 reset 信号为高，把q置零
```

```
        q = 1'b0 ;
```

```
    else if ( clock )// 若clock 信号为高，锁存输入信号d
```

```
        q = d ;
```

```
end
```



```
//用reset异步下降沿复位，clk正跳变沿触发的D寄存器
always @ ( posedge clk , negedge reset )
//注意：使用逗号来代替关键字or
if ( ! reset )
    q <= 0 ;
else
    q <= d ;
```

如果组合逻辑块语句的输入变量很多，那么编写敏感列表会很繁琐并且容易出错。针对这种情况，**Verilog**提供另外两个特殊的符号：**@*** 和**@(*)**，它们都表示对其后面语句块中所有输入变量的变化时敏感的。

/用or 操作符的组合逻辑块

//编写敏感列表很繁琐并且容易漏掉一个输入

```
always @(a or b or c or d or e or f or g or h or p or m)
begin
    out1 = a ? b + c : d + e ;
    out2 = f ? g + h : p + m ;
end
```

//不用上述方法，用符号 @(*) 来代替，
//可以把所有输入变量都自动包括进敏感列表。

```
always @ ( * )
```

```
begin
```

```
    out1 = a ? b + c : d + e ;
```

```
    out2 = f ? g + h : p + m ;
```

```
end
```

电平敏感时序控制 wait

always

```
wait (count_enable) #20 count=count+1;
```

仿真器连续监视count_enable的值，若其值为0，则不执行后面的语句，仿真会停顿下来；如果其值为1，则在20个时间单位之后执行这条语句。如果count_enable始终为1，那么count将每过20个时间单位加1。

6.2 task和function说明语句

在设计中，设计者经常需要在程序的多个不同地方实现同样的功能。这表明有必要把这些公共的代码提取出来，将其组成子程序，然后在需要的地方调用这些子程序，以避免重复编码。**Verilog**语言提供的任务和函数可以将较大的设计划分为较小的代码段，允许设计者将在多个地方使用的相同代码提取出来，编写成任务和函数，以使代码简洁、易懂。

task和function说明语句的不同点

- (1) 函数只能与主模块共用同一个仿真时间单位，而任务可以定义自己的仿真时间单位。
- (2) 函数不能启动任务，而任务能启动其它任务和函数。
- (3) 函数至少要有一个输入变量，而任务可以没有或有多多个任何类型的变量。
- (4) 函数返回一个值，而任务则不返回值。

task说明语句

如果传给任务的变量值和任务完成后接收结果的变量已定义，就可以用一条语句启动任务。任务完成以后控制就传回启动过程。如任务内部有定时控制，则启动的时间可以与控制返回的时间不同。任务可以启动其它的任务，其它任务又可以启动别的任务，可以启动的任务数是没有限制的。不管有多少任务启动，只有当所有的启动任务完成以后，控制才能返回。

任务的定义

定义任务的语法如下：

```
task <任务名>;  
    <端口及数据类型声明语句>  
begin  
    <语句1>  
    <语句2>  
    .....  
    <语句n>  
end  
endtask
```



任务的调用及变量的传递

启动任务并传递输入输出变量的声明语句的语法如下：

<任务名>(端口1， 端口2， ...， 端口n);

```
module call_task;
    reg[15:0] old_word;
    reg[15:0] new_word;

    task switch_bytes;
        input[15:0] old_w;
        output[15:0] new_w;
        reg[15:0] temp;
        //此变量可以不要，此处是为了说明task内变量的定义

    begin
        temp=old_w;
        new_w[15:8]=temp[7:0];
        new_w[7:0]=temp[15:8];
    end
endtask
```



```
initial
begin
    old_word=16'h3fa2;
    switch_bytes(old_word,new_word);
    #100;
    old_word=16'habcd;
    switch_bytes(old_word,new_word);
end
endmodule
```

```
module traffic_lights;
    reg  clock, red, amber, green;
    parameter  on=1, off=0, red_tics=350,
        amber_tics=30, green_tics=200;
    //交通灯初始化
    initial    red=off;
    initial    amber=off;
    initial    green=off;
    //交通灯控制时序
    always
    begin
        red=on;           //开红灯
        light(red, red_tics); //调用等待任务
        green=on;          //开绿灯
        light(green, green_tics); //等待
        amber=on;          //开黄灯
        light(amber, amber_tics); //等待
    end
```

```
//定义交通灯开启时间的任务
task light;
output color;
input[31:0] tics;
begin
    repeat(tics)
        @(posedge clock); //等待tics个时钟的上升沿
        color=off; //关灯
    end
endtask
//产生时钟脉冲的always块
always
begin
    #100 clock=0;
    #100 clock=1;
end
endmodule
```

function说明语句

函数的目的是返回一个用于表达式的值。

定义函数的语法：

```
function <返回值的类型或范围> (函数名);
```

```
<端口说明语句>
```

```
<变量类型说明语句>
```

```
begin
```

```
<语句>
```

```
.....
```

```
end
```

```
endfunction
```

从函数返回的值

函数的定义蕴含声明了与函数同名的、函数内部的寄存器。如在函数的声明语句中<返回值的类型或范围>为缺省,则这个寄存器是一位的, 否则是与函数定义中<返回值的类型或范围>一致的寄存器。函数的定义把函数返回值所赋值寄存器的名称初始化为与函数同名的内部变量。

函数的调用

函数的调用是通过将函数作为表达式中的操作数来实现的。

<函数名> (<表达式>, ..., <表达式>)

函数的使用规则

与任务相比较函数的使用有较多的约束，下面给出的是函数的使用规则：

函数的定义不能包含有任何的时间控制语句，即任何用#、@、或wait来标识的语句。

函数不能启动任务。

定义函数时至少要有一个输入参量。

在函数的定义中必须有一条赋值语句给函数中的一个内部变量赋以函数的结果值，该内部变量具有和函数名相同的名字。

```
module call_function;
    reg[15:0] old_word;
    reg[15:0] new_word;

    function[15:0] switch_bytes;
        input[15:0] old_w;
        reg[15:0] temp;//此变量可以不要，此处是为了说明
        function内变量的定义
    begin
        temp=old_w;
        switch_bytes[15:8]=temp[7:0];
        switch_bytes[7:0]=temp[15:8];
    end
endfunction
```

```
initial
begin
    old_word=16'h3fa2;
    new_word=switch_bytes(old_word);
    #100;
    old_word=16'habcd;
    new_word=switch_bytes(old_word);
end
endmodule
```

6.2.4函数使用举例

奇偶校验位的计算

```
//定义一个模块，其中包含能计算偶校验位的函数（calc_parity）  
module parity;  
reg [31:0] addr;  
reg parity;  
  
initial  
begin  
    addr = 32'h3456_789a;  
    #10 addr = 32'hc4c6_78ff;  
    #10 addr = 32'hff56_ff9a;  
    #10 addr = 32'h3faa_aaaa;  
end
```

```
//每当地址值发生变化，计算新的偶校验位
always @(addr)
begin
    parity = calc_parity(addr);
    //第一次启动校验位计算函数 calc_parity
    $display("Parity calculated = %b", calc_parity(addr) );
    // 第二次启动校验位计算函数 calc_parity
end
//定义偶校验计算函数
function calc_parity;
input [31:0] address;
begin
    //适当地设置输出值，使用隐含的内部寄存器calc_parity
    calc_parity = ^address; //返回所有地址位的异或值
end
endfunction
endmodule
```


使用C风格进行变量声明的函数定义

```
//定义偶校验位计算函数，该函数采用ANSI C 风格的变量声明
function calc_parity (input [31:0] address) ;
begin
    //适当地设置输出值，使用隐含的内部寄存器calc_parity
    calc_parity = ^address; //返回所有地址位的异或值
end
endfunction
```

// 定义一个包含移位函数的模块

```
module shifter;
```

// 左/右 移位寄存器

```
`define LEFT_SHIFT      1'b0
```

```
`define RIGHT_SHIFT     1'b1
```

```
reg [31:0] addr, left_addr, right_addr;
```

```
//reg control;
```

//每当新地址出现时就计算右移位和左移位的值

```
always @(addr)
```

```
begin
```

//调用下面定义的具有左右移位功能的函数

```
    left_addr = shift(addr, `LEFT_SHIFT);
```

```
    right_addr = shift(addr, `RIGHT_SHIFT);
```

```
end
```

```
//定义移位函数，其输出是一个32位的值
function [31:0] shift;
input [31:0] address;
input control;
begin
    //根据控制信号适当地设置输出值
    shift = (control == `LEFT_SHIFT) ? (address << 1) :
(address >> 1);
end
endfunction
initial
begin
    addr=32'h0001_0000;
    #100;
    addr=32'h0200_0000;
end
endmodule
```

6.2.5 自动（递归）函数

Verilog中的函数是不能够进行递归调用的。

若在函数声明时使用关键字**automatic**，那么改函数可以递归调用。

//用函数的递归调用定义阶乘计算

module ex6_14 ;

//定义自动（递归）函数

function automatic integer factorial ;

input [31:0] oper ;

integer i ;

begin

if (oper >= 2)//此处修改

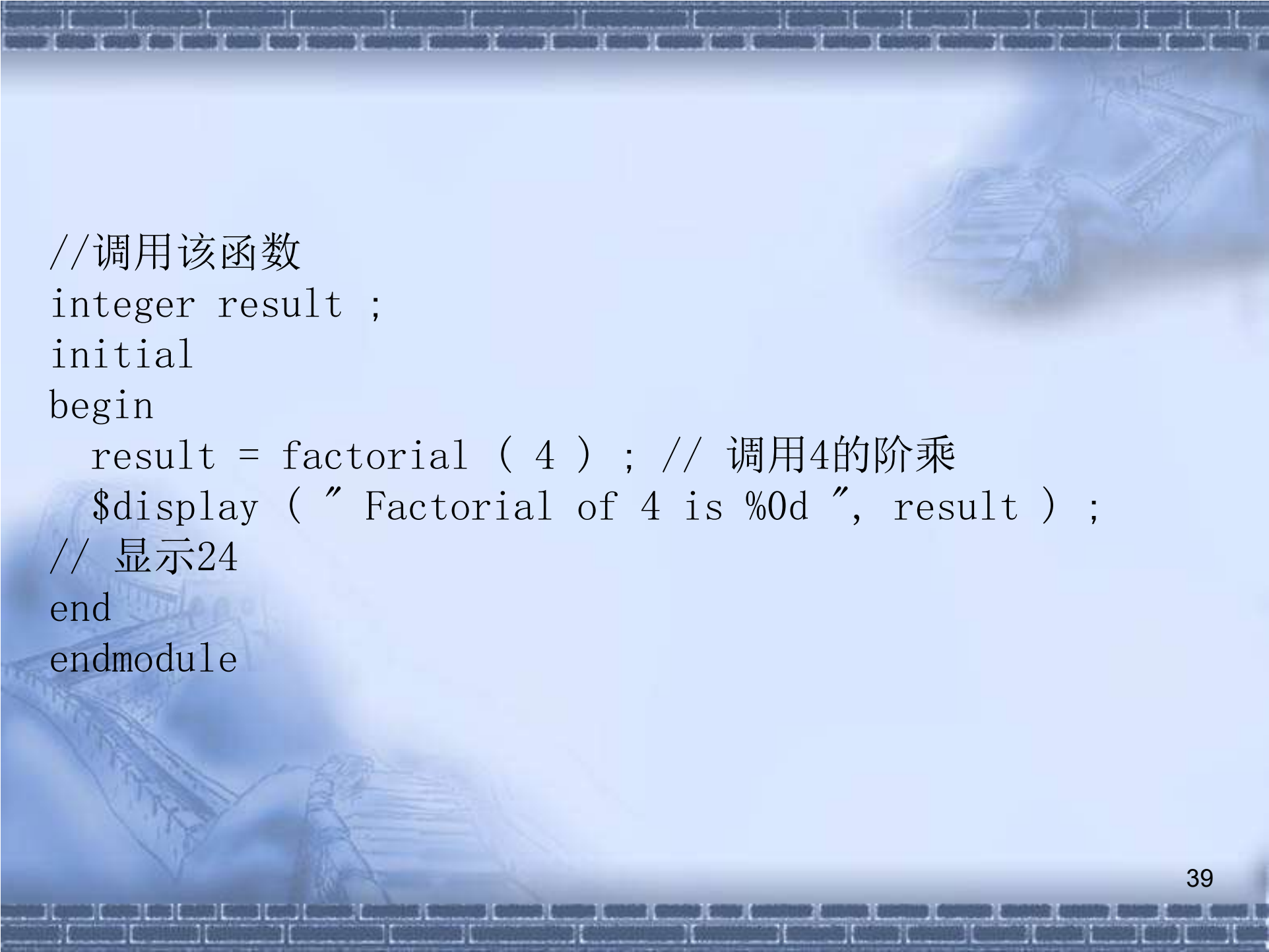
factorial = factorial(oper-1)*oper; //递归调用

else

factorial = 1 ;

end

endfunction



```
//调用该函数
integer result ;
initial
begin
    result = factorial ( 4 ) ; // 调用4的阶乘
    $display ( " Factorial of 4 is %0d ", result ) ;
// 显示24
end
endmodule
```

```
module test_task;
```

```
    task automatic factorial; /*如果把automatic去掉, 则  
    仿真结果不正确*/
```

```
        output integer result;
```

```
        input  integer n;
```

```
        integer temp;
```

```
begin
```

```
    if(n>1)
```

```
    begin
```

```
        factorial(temp, n-1);
```

```
        result=temp*n;
```

```
    end
```

```
    else
```

```
        result=1;
```

```
end
```

```
endtask
```

```
integer num, res_out;

always @(num)
begin
    factorial(res_out, num);
    $display("num=%d,  result is %d", num, res_out);
end

initial
begin
    num=5;
    #100;
    num=9;
    #50;
    num =3;
end

endmodule    /* run 500 ns    */
```

6.2.6 常量函数

常量函数实际上是一个带有某些限制的常规Verilog函数。这种函数能够用来引用复杂的值，因而可以用来代替常量。

```
module ex6_15(addr_bus, out);  
parameter  RAM_DEPTH = 256 ;  
input[clogb2(RAM_DEPTH)-2:0]  addr_bus ;    //  
output[clogb2(RAM_DEPTH)-2:0] out;  
  
function integer clogb2(input integer depth) ;  
begin  
    for (clogb2=0;depth>0;clogb2=clogb2+1)  
        depth = depth >> 1 ;  
end  
endfunction
```

```
    assign out=addr_bus;  
endmodule  
//force addr_bus 8'h0 , 8'h23 200
```

6.2.7带符号函数

带符号函数的返回值可以作为带符号数进行运算。


```
module ex6_16 ;  
reg[3:0] vector;  
function signed[3:0] compute_signed(input[3:0] vector);  
  
    compute_signed=vector;  
endfunction  
  
initial  
begin  
    vector=4'b1001;  
    if(compute_signed(vector)<-3)  
        $display("less...");  
    else  
        $display("more...");  
    end  
endmodule
```

6.4 常用的系统任务

6.4.1 \$display和\$write任务

格式

```
$display(p1,p2,...,pn);
```

```
$write(p1,p2,...,pn);
```

这两个函数和系统任务的作用是用来输出信息，即将参数p2到pn按参数p1给定的格式输出。参数p1通常称为“格式控制”，参数p2至pn通常称为“输出表列”。

这两个任务的作用基本相同。**\$display**自动地在输出后进行换行，**\$write**则不是这样。如果想在一行里输出多个信息，可以使用**\$write**。

因为**\$write**在输出时不换行，要注意它的使用。可以在**\$write**中加入换行符\n，以确保明确的输出显示格式。

```
module disp;
reg[31:0] rval;

    initial
    begin
        rval=101;
        $display("rval=%h hex %d decimal", rval, rval);
        $display("rval=%o octal %b binary", rval, rval);
        $display("rval has %c ascii character
value",rval);

        $display("current scope is %m");
        $display("%s is ascii value for 101",101);
        $display("simulation time is %t",$time);
    end
endmodule
```


6.4.2 文件输出

打开文件 使用系统任务\$ fopen

用法: \$fopen(“<文件名>”);

用法: <文件句柄>=\$fopen(“<文件名>”);

任务\$ fopen返回一个被称为多通道描述符的32位值。多通道描述符中只有一位被设置成1. 标准输出有一个多通道描述符, 其最低位(第0位)被设置成1. 标准的输出也称通道为0。标准输出一直是开放的。以后对\$ fopen的每一次调用打开一个新通道, 并且返回一个设置第1位、第2位等, 直到32位描述的第30位。第31位是保留位。通道号与多通道描述符中被设置成1的位相对应。

多通道的优点在于可以有选择的同时写多个文件。

写文件

`$fdisplay` `$fmonitor` `$fwrite` `$fstrobe`都用于写文件

这些任务在语法上与常规系统任务`$display`、`$monitor`等类似

关闭文件 使用系统任务`$fclose`

`$fclose(<文件描述符>);`

```
module ex6_20;
```

```
//多通道描述符
```

```
integer handle1, handle2, handle3; //整型数为 32 位  
//标准输出是打开的; descriptor = 32'h0000_0001 (第0位  
置1)
```

```
integer desc1, desc2, desc3 ; // 三个文件的描述符
```

```
initial
```

```
begin
```

```
    handle1 = $fopen("file1.out");
```

```
    //handle1 = 32'h0000_0002 (bit 1 set 1)
```

```
    handle2 = $fopen("file2.out");
```

```
    //handle2 = 32'h0000_0004 (bit 2 set 1)
```

```
    handle3 = $fopen("file3.out");
```

```
    //handle3 = 32'h0000_0008 (bit 3 set 1)
```

```
desc1 = handle1 | 1; //按位或; desc1 = 32'h0000_0003
$fdisplay(desc1, "Display 1");
//写到文件file1.out和标准输出stdout
desc2 = handle2 | handle1; //desc2 = 32'h0000_0006
$fdisplay(desc2, "Display 2");
//写到文件file1.out和file2.out
desc3 = handle3 ; //desc3 = 32'h0000_0008
$fdisplay(desc3, "Display 3 desc1=%d desc2=%d
    desc3=%d", desc1, desc2, desc3); //只写到文件file3.out
$fclose(handle1|handle2|handle3); //本句不写也可以
end

endmodule
```

6.4.3 显示层次

通过任何显示任务，例如`$display`、`$write`、`$monitor`或者`$strobe`任务中的`%m`选项的方式可以显示任何级别的层次。

//显示层次信息

```
module M;  
  initial  
    $display("Displaying in %m");  
endmodule
```

//调用模块M

```
module top;  
  
  M m1( );  
  M m2( );  
  M m3( );  
  
endmodule
```


仿真输出如下所示：

Displaying in top.m1

Displaying in top.m2

Displaying in top.m3

这一特征可以显示全层次路径名，包括模块实例、任务、函数和命名块。

6.4.4选通显示

\$strobe 和**\$display**任务除了一点小的差别外，其它非常相似。

如果许多其它语句和**\$display**任务在同一个时间单位执行，那么这些语句与**\$display**任务的执行顺序是不确定的。如果使用**\$strobe**，该语句总是在同时刻的其他赋值语句执行完成后才执行。

```
module ex6_22;
reg[3:0] a, b, c, d;
reg clock;

always @ (posedge clock)
    $strobe("%t Displaying a = %b,    c = %b", $time, a , c);
//显示正跳变沿时刻的值

//选通显示
always @ (posedge clock)
begin
    a = b ;
    c = d ;
end
```

```
initial
    clock=0;
always #10 clock=~clock;
initial
begin
    b=4'h0;
    d=4'h2;
    #25;
    b=4'h9;
    d=4'h5;
end
endmodule
```

//在例6.22中，时钟上升沿的值在语句a = b和c = d执行完之后才显示。

//如果使用\$display，\$display可能在语句a = b和c = d之前执行，结果显示不同的值。

//run 45

值变转储文件

值变转储文件(VCD)是一个ASCII文件，它包含仿真时间、范围与信号的定义以及仿真运行过程中信号值的变化等信息。设计中的所有信号或者选定的信号集合在仿真过程中都可以被写入VCD文件。后处理工具可以把VCD文件作为输入并把层次信息、信号值和信号波形显示出来。

\$dumpvars 用来选择要转储的模块实例或者模块实例信号

\$dumpfile选择VCD文件

\$dumpon和**\$dumpoff** 选择转储过程的起点和终点

\$dumpall选择生成检测点。

//指定VCD文件名。若不指定VCD文件，则由仿真器指定一缺省文件名

```
initial
```

```
$dumpfile(“myfile.dmp”); //仿真信息转储到  
myfile.dmp文件
```

```
// 转储模块中的信号
```

```
initial
```

```
$dumpvars; //没有指定变量范围，把设计中全部信号都转储
```

```
initial
```

```
$dumpvars(1, top); //转储模块实例 top中的信号
```

```
//数1 表示层次的等级， 只转储top下第一层信号  
//即转储top模块中的变量，而不转储在top中调用  
//模块中的变量
```

```
initial
    $dumpvars (2, top.m1);    //转储top.m1模块下两层的信号
initial
    $dumpvars (0, top.m1); ///数0 表示转储top.m1模块下面各
个层的所有信号
//启动和停止转储过程
initial
begin
    $dumpon ;    //启动转储过程
    #100000 $dumpoff ; //过了100000个仿真时间单位后，停
止转储过程
end
```

//生成一个检查点, 转储所有VCD变量的现行值。

```
initial
```

```
    $dumpall ;
```

VCD数据库不记录仿真结束时的数据。因此如果希望看到最后一次数据变化后的波形, 必须在仿真结束前使用\$dumpall

ModelSim中使用vcd2wlf命令来把vcd文件转换成wlf文件

```
vcd2wlf count.vcd count.wlf
```

```
module counter_10c (Q, clock, clear, ov);  
output [3:0] Q;  
output ov;  
input clock, clear;  
reg [3:0] Q=4'b0000;  
reg ov;
```

```
always @ (posedge clear or negedge clock)  
begin  
    if (clear)  
        Q<=4'b0;  
    else if (Q==8)  
        begin  
            Q<=Q+1;  
            ov<=1'b1;  
        end  
end
```

```
else if (Q==9)
  begin
    Q<=4'b0000;
    ov<=1'b0;
  end
else
  begin
    Q<=Q+1;
    ov<=1'b0;
  end
end
endmodule
```



```
module test_counter10c;  
    reg clock, clear;  
    wire ov;  
    wire[3:0] Q;  
  
    counter_10c U1(Q, clock, clear, ov);  
  
    always  
    begin  
        clock=0;  
        #10;  
        clock=1;  
        #10;  
    end
```

```
initial
```

```
begin
```

```
    clear=0;
```

```
    #123 clear=1;
```

```
    #50  clear=0;
```

```
end
```

```
initial
```

```
begin
```

```
    $dumpfile("count.dmp");//$dumpfile("count.vcd");
```

```
    $dumpvars(1,U1);
```

```
    #1005 $dumpall;
```

```
end
```

```
endmodule
```

```
//run 1008
```