

分突出。任务本身并不返回计算值,但它通过类似 C 语言中的形参与实参的数据交换,非常容易地实现运算结果的调用。此外,还常常利用任务来包装模块设计中的许多复杂的过程,将许多复杂的操作步骤用一个命名清晰易懂的任务隐藏起来,大大提高了程序的可读性。

下面介绍一个实例,它巧妙地利用电平敏感的 always 块和一个比较两变量大小排序的任务,设计出 4 个(4 位)并行输入数的高速排序组合逻辑。可以看到,利用 task 非常方便地实现了两个数据之间的交换排序,通过在电平敏感的 always 块中的多次调用,实现了 4 变量的高速排序。是用函数无法实现相同的功能;另外,task 也避免了直接用一般语句来描述所引起的不易理解和综合时产生冗余逻辑等问题。

模块源代码:

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

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

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

值得注意的是,task 中的变量定义与模块中的变量定义不尽相同,它们并不受输入输出

类型的限制。如上例,x与y对于task sort2来说虽然是inout型,但实际上它们对应的是always块中变量,属于reg型变量。

测试模块源代码:

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

module task_Top;
    reg[3:0] a,b,c,d;
    wire[3:0] ra,rb,rc,rd;

    initial
        begin
            a=0;b=0;c=0;d=0;
            repeat(50)
                begin
                    # 100      a = { $random } %15;
                    b = { $random } %15;
                    c = { $random } %15;
                    d = { $random } %15;
                end
            # 100 $stop;
        end

    sort4 sort4 (.a(a),.b(b),.c(c),.d(d),.ra(ra),.rb(rb),.rc(rc),.rd(rd));
endmodule
```

使用任务后的仿真波形(部分)如实验图7所示。

/task_Top/a	0000	1000	1100	0110
/task_Top/b	0000	1100	0010	0100
/task_Top/c	0000	0111	0101	0011
/task_Top/d	0000	0010	0111	0010
/task_Top/ra	0000	0010		
/task_Top/rb	0000	0111	0101	0011
/task_Top/rc	0000	1000	0111	0100
/task_Top/rd	0000	1100		0110

实验图7 task的仿真波形

【练习题6】用两种不同的方法设计一个功能相同的模块,该模块能完成四个8位2进制输入数据的冒泡排序。第一种,模仿上面的例子用纯组合逻辑实现;第二种,假设8位数据

是按照时钟节拍串行输入的,要求用时钟触发任务的执行法,每个时钟周期完成一次数据交换的操作。比较两种不同方法的运行速度和消耗资源的不同。

练习八 利用有限状态机进行时序逻辑的设计

目的:

- (1) 掌握利用有限状态机实现一般时序逻辑分析的方法;
- (2) 掌握用 Verilog 编写可综合的有限状态机的标准模板;
- (3) 掌握用 Verilog 编写状态机模块的测试文件的一般方法。

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

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

模块源代码:

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

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

assign z = (state==E && x==0)? 1 : 0;
//当 x 序列 10010 最后一个 0 刚到时刻,时钟沿立刻将状态变为 E,此时 z 应该变为高
always @(posedge clk)
    if(!rst)
        begin
            state<= IDLE;
        end
    else
        casex(state)
            IDLE : if(x==1)           //第一个码位对,记状态 A
                begin
                    state<= A;
                end
            A : if(x==0)           //第二个码位对,记状态 B
                begin
                    state<= B;
                end
            B : if(x==1)           //第三个码位对,记状态 C
                begin
                    state<= C;
                end
            C : if(x==0)           //第四个码位对,记状态 D
                begin
                    state<= D;
                end
            D : if(x==1)           //第五个码位对,记状态 E
                begin
                    state<= E;
                end
            E : if(x==0)           //第六个码位对,记状态 F
                begin
                    state<= F;
                end
            F : if(x==1)           //第七个码位对,记状态 G
                begin
                    state<= G;
                end
            G : if(x==0)           //第八个码位对,记状态 IDLE
                begin
                    state<= IDLE;
                end
        endcase
end
```

```

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

```

```

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

```

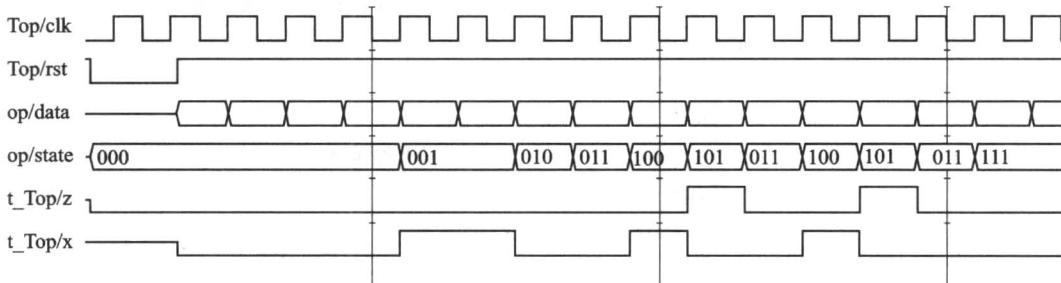
测试模块源代码:

```

//-----文件名 seqdet.v -----
'timescale 1ns/1ns
`include "./seqdet.v"
module seqdet_Top;
reg clk,rst;
reg[23:0] data;
wire[2:0] state;
wire z,x;
assign x=data[23];
always #10 clk = ~clk;
always @(posedge clk)
    data={data[22:0],data[23]}; //形成数据向左移环行流,最高位与 x 连接
initial
begin
    clk=0;
    rst=1;
#2 rst=0;
#30 rst=1;
    data='b1100_1001_0000_1001_0100;
#500 $stop;
end
seqdet m(x,z,clk,rst,state);
endmodule

```

状态机设计的仿真波形如实验图 8 所示。



实验图 8 状态机设计的仿真波形

【练习题 7】 设计一个串行数据检测器。要求是:连续 4 个或 4 个以上为 1 时输出为 1, 其他输入情况下为 0。编写测试模块对设计的模块进行各种层次的仿真, 并观察波形, 编写实验报告。

练习九 利用状态机实现比较复杂的接口设计

目的:

- (1) 学习运用由状态机控制的逻辑开关, 设计出一个比较复杂的接口逻辑;
- (2) 在复杂设计中使用任务(task)结构, 以提高程序的可读性;
- (3) 加深对可综合风格模块的认识。

在练习八的示范题中, 学习了如何使用状态机的实例。实际上, 单个有限状态机控制整个系统逻辑电路的运转, 这在实际设计中是不多见的。一般情况下, 往往是状态机套用状态机, 从而形成复杂的控制流。下面将提供这样的一个示例, 供大家学习。

该例是一个并行数据转换为串行位流的变换器, 利用双向总线输出。事实上, 它是 EPROM 读写器设计中实现写功能的部分程序经删节得到的。为了帮助读者的理解做了许多简化, 去除了 EPROM 的启动、结束和 EPROM 控制字的写入等功能, 只具备这样一个简单的并串转换功能。因而本设计无任何实用价值, 只是为了教学目的。电路工作的步骤是:

- (1) 把并行地址存入寄存器;
- (2) 把并行数据存入寄存器;
- (3) 连接串行单总线;
- (4) 地址的串行输出;
- (5) 数据的串行输出;
- (6) 挂起串行单总线;
- (7) 给信号源应答;
- (8) 让信号源给出下一个操作对象;
- (9) 结束写操作。

通过基本时钟的运行, 使得并行数据一位一位地输出。

模块源代码:

```
module writing(reset,clk,address,data,sda,ack);
    input reset,clk;

```

```

input[7:0] data,address;

inout sda;                                //串行数据的输出或者输入接口
output ack;                                 //模块给出的应答信号
reg link_write;                            //link_write 决定何时输出
reg[3:0] state;                             //主状态机的状态字
reg[4:0] sh8out_state;                     //从状态机的状态字
reg[7:0] sh8out_buf;                        //输入数据缓冲
reg finish_F;                              //用以判断是否处理完一个操作对象
reg ack;

parameter
idle=0,addr_write=1,data_write=2,stop_ack=3;
parameter
bit0=1,bit1=2,bit2=3,bit3=4,bit4=5,bit5=6,bit6=7,bit7=8;
assign  sda = link_write? sh8out_buf[7] : 1'bz;

always @(posedge clk)
begin
if(! reset)                                //复位
begin
link_write    <= 0;                      //挂起串行单总线
state         <= idle;
finish_F     <= 0;                      //结束标志清零
sh8out_state <=idle;
ack          <= 0;
sh8out_buf   <=0;
end
else
case(state)

idle:
begin
link_write    <= 0;                      //断开串行单总线
finish_F     <= 0;
sh8out_state<=idle;
ack          <= 0;
sh8out_buf   <=address;                 //并行地址存入寄存器
state         <= addr_write;            //进入下一个状态
end

addr_write:                                //地址的输入

```

```

begin
    if(finish_F==0)
        begin shift8_out; end //地址的串行输出
    else
        begin
            sh8out_state <= idle;
            sh8out_buf <= data; //并行数据存入寄存器
            state<= data_write;
            finish_F<= 0;
        end
    end

data_write:
begin
    if(finish_F==0)
        begin shift8_out; end //数据的写入
    else
        begin
            link_write <= 0;
            state <= stop_ack;
            finish_F <= 0;
            ack <= 1; //向信号源发出应答
        end
    end
endcase
end

task shift8_out; // 地址和数据的串行输出
begin
    case(sh8out_state)
        idle:
            begin
                link_write <= 1; //连接串行总线,立即输出地址或数据
                //的最高位(MSB)
                sh8out_state <= bit7;
            end
    end
end

```

```
bit7:  
begin  
    link_write <= 1;           //连接串行单总线  
    sh8out_state <= bit6;  
    sh8out_buf <= sh8out_buf<<1; //输出地址或数据的次高位(bit6)  
end  
  
bit6:  
begin  
    sh8out_state <= bit5;  
    sh8out_buf <= sh8out_buf<<1;  
end  
  
bit5:  
begin  
    sh8out_state <= bit4;  
    sh8out_buf <= sh8out_buf<<1;  
end  
  
bit4:  
begin  
    sh8out_state <= bit3;  
    sh8out_buf <= sh8out_buf<<1;  
end  
  
bit3:  
begin  
    sh8out_state <= bit2;  
    sh8out_buf <= sh8out_buf<<1;  
end  
  
bit2:  
begin  
    sh8out_state <= bit1;  
    sh8out_buf <= sh8out_buf<<1;  
end  
  
bit1:  
begin  
    sh8out_state <= bit0;  
    sh8out_buf <= sh8out_buf<<1; //输出地址或数据的最低位(LSB)  
end  
  
bit0:
```

```

begin
    link_write <= 0;           //挂起串行单总线
    finish_F <= 1;           //建立结束标志
end
endcase
end
endtask
endmodule

```

测试模块源代码：

```

'timescale 1ns/100 ps
`define clk_cycle 50
module writingTop;
    reg reset,clk;
    reg[7:0] data,address;
    wire ack,sda;

    always # `clk_cycle clk = ~clk;

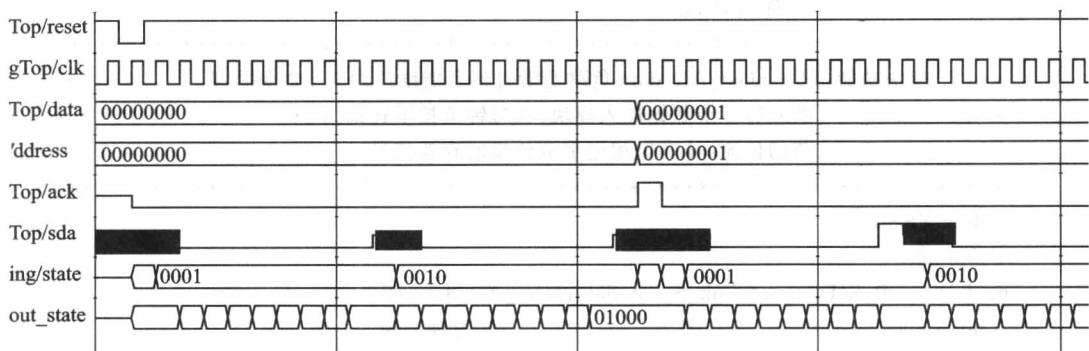
    initial
        begin
            clk=0;
            reset=1;
            data=0;
            address=0;
            #(2 * `clk_cycle) reset=0;
            #(2 * `clk_cycle) reset=1;
            #(100 * `clk_cycle) $stop;
        end

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

```

复杂接口设计的状态仿真波形如实验图 9 所示。在这里波形图内的长黑块为双向单总线 sda 的高阻状态。

【练习题 8】 彻底搞清楚上例,参考第二部分的第 16 章的实际例子,独立编写一个能实现 EEPROM 全部读写功能的并行转换为 I²C 串行总线读写信号的模块。编写完整的符合工程要求的测试模块,进行各种层次的仿真,并观察波形。



实验图 9 复杂接口设计的状态机仿真波形

【练习题 9】 参考第二部分的第 15 章中的[例 15.2]，编写可综合模块能把符合 I²C 串行总线要求的地址和数据信号，在只有 sda 和 scl 两个信号的前提下转换为并行的数据和地址信号。

练习十 通过模块实例调用实现大型系统的设计

目的：

- (1) 学习和掌握状态机的嵌套和模块实例的连接方法；
- (2) 了解大型系统设计的层次化、结构化解决办法的技术基础；
- (3) 学习数据总线在模块设计中的应用和控制，掌握复杂接口模块设计的基本技术；
- (4) 学习和掌握用工程概念来编写较完整的测试模块，做到接近真实的完整测试。

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

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

```

//----- 文件名 P_S.v -----
// **** 模块功能:把在 nGet_AD_data 负跳变沿时刻后能维持约三个时钟周期 ****
//       的并行字节数据取入模块,在时钟节拍下转换为字 ****
//       节的位流,并产生相应字节位流的有效信号 ****
// ****

`define YES 1
`define NO 0

module P_S(Dbit_out,link_S_out,data,nGet_AD_data,clk);
    input clk;                                //主时钟节拍
    input nGet_AD_data;                      //负电平有效时取并行数据控制信号线
    input [7:0] data;                         //并行输入的数据端口
    output Dbit_out;                         //串行位流的输出
    output link_S_out;                       //允许串行位流输出的控制信号

    reg [3:0] state;                         //状态变量寄存器
    reg[7:0] data_buf;                       //并行数据缓存器
    reg link_S_out;                          //串行位流输出的控制信号寄存器
    reg d_buf;                               //位缓存器
    reg finish_flag;                         //字节处理结束标志

    assign Dbit_out = (link_S_out)? d_buf:0;//给出串行数据

    always @(posedge clk or negedge nGet_AD_data)
        // nGet_AD_data 下降沿置数,寄存器清零,clk 上跳沿送出位流
        if(! nGet_AD_data)
            begin
                finish_flag <= 0;
                state <= 9;
                link_S_out <= 'NO;
                d_buf <= 0;
                data_buf <= 0;
            end
        else
            case(state)
                9:   begin
                        data_buf <= data;
                        state <= 10;
                        link_S_out <= 'NO;
                    end
                10:  begin
                        data_buf <= data;
                        state <= 0;
                        link_S_out <= 'NO;
                    end
            end
    end

```

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

```

//----- 文件名 S_P.v -----
// **** 模块功能:把在位流有效信号控制下的字节位流读入模块,在时钟节拍控制 ****
// *** 下转换为并行的字节数据,输出到并行数据口 ****
// ****

`timescale 1ns/1ns

`define YES 1
`define NO 0

module S_P(data, Dbit_in, Dbit_ena, clk);
    output [7:0] data;           //并行数据输出口
    input   Dbit_in, clk;       //字节位流输入口
    input   Dbit_ena;          //字节位流使能输入口

    reg [7:0] data_buf;
    reg [3:0] state;          //状态变量寄存器
    reg p_out_link;           //并行输出控制寄存器

    assign data = (p_out_link=='YES) ? data_buf : 8'bz;

    always@(negedge clk)
        if(Dbit_ena )
            case(state)
                0: begin
                    p_out_link <='NO;
                    data_buf[7] <= Dbit_in;
                    state <=1;
                end
                1: begin
                    data_buf[6] <= Dbit_in;
                    state <=2;
                end
                2: begin
                    data_buf[5] <= Dbit_in;
                    state <=3;
                end
                3: begin
                    data_buf[4] <= Dbit_in;
                    state <=4;
                end
                4: begin
                    data_buf[3] <= Dbit_in;
                    state <=5;
                end
                5: begin
                    data_buf[2] <= Dbit_in;
                end
            endcase
        end
endmodule

```

```

        state <= 6;
    end
6: begin
    data_buf[1] <= Dbit_in;
    state <= 7;
end
7: begin
    data_buf[0] <= Dbit_in;
    state <= 8;
end
8: begin
    p_out_link <= 'YES;
    state <= 4'b1111;
end
default: state <= 0;
endcase
else begin
    p_out_link <= 'YES;
    state <= 0;
end
endmodule
//----- 文件名 sys.v -----
/*
*** 模块的功能:把两个独立的逻辑模块(P_S 和 S_P)合并到一个可综合的模块中, ***
*** 共用一条并行总线,配合有关信号,分时进行输入/输出 ***
*** 模块的目的:学习如何把两个单向输入/输出的实例模块,连接在一起,共享一条 ***
*** 总线 ***
*** 本模块是完全可综合模块,已经通过综合和布线后仿真 ***
*/
`include "./P_S.v"
`include "./S_P.v"
module sys(databus,use_p_in_bus,Dbit_out,Dbit_ena,nGet_AD_data,clk);
    input nGet_AD_data; //取并行数据的控制信号
    input use_p_in_bus; // 并行总线用于输入数据的控制信号
    input clk; //主时钟
    inout [7:0] databus; //双向并行数据总线
    output Dbit_out; //字节位流输出
    output Dbit_ena; //字节位流输出使能
    wire clk;
    wire nGet_AD_data;
    wire Dbit_out;
    wire Dbit_ena;
    wire [7:0] data;

```

```

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

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

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

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

initial
begin
    clk = 0;
    nGet_AD_data = 1;        //置取数据控制信号初始值为高电平
    data_buf = 8'b1001_1001;   //假设数据缓存器的初始值,可用于模拟并行数据的变化
    D_Pin_ena = 0;
end

initial
begin
    repeat(100)
    begin
        #(100 * 14 + { $random } % 23) nGet_AD_data = 0; //取并行数据开始
        #(112 + { $random } % 12) nGet_AD_data = 1;      //保持一定时间低电平后恢复高电平
        #({ $random } % 50) D_Pin_ena = 1; //并行数据输入 sys 模块的使能信号有效
        #(100 * 3 + { $random } % 5) D_Pin_ena = 0; //保持三个时钟周期后让出总线
        #333 data_buf = data_buf + 1; //假设数据变化,可为下次取得不同的数据
        #(100 * 11 + { $random } % 1000);
    end
end

```

```

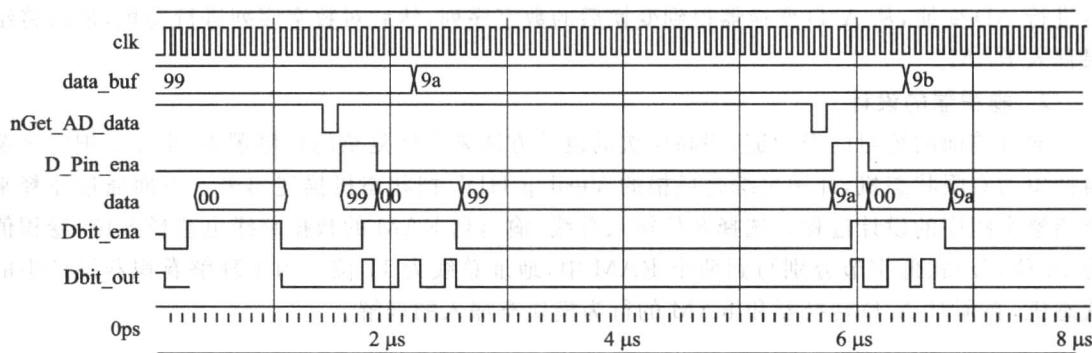
    end
endmodule

always #(50+$random%2) clk = ~clk; //主时钟的产生

sys ms(.databus(data),
      .use_p_in_bus(D_Pin_ena),
      .Dbit_out(Dbit_out),
      .Dbit_ena(Dbit_ena),
      .nGet_AD_data(nGet_AD_data),
      .clk(clk));
endmodule

```

布线后仿真波形如实验图 10 所示。



实验图 10 布线后仿真波形

【练习题 10】 模仿示范题, 编写一个通过申请 CPU 中断后取得数据进行处理, 并把处理的结果通过同一条数据总线返回 CPU 的模块。要求具体时序参数与 CPU 中断的响应时间和读写时序完全一致。并考虑尽量减少资源消耗, 并提高处理速度。

练习十一 简单卷积器的设计

目的:

- (1) 学习和掌握高速计算逻辑状态机的控制基本方法;
- (2) 了解计算逻辑与存储器和 AD 模块的接口设计技术基础;
- (3) 进一步掌握数据总线在模块设计中的应用和控制;
- (4) 熟悉用工程概念来编写较完整的测试模块, 做到接近真实的完整测试。

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

1. 明确设计任务

在设计之前必须明确设计的具体内容。卷积器是数字信号处理系统中常用的部件, 它首先对模拟输入信号实时采样, 得到数字信号序列。然后对数字信号进行卷积运算, 再将卷积结

果存入 RAM 中。对模拟信号的采样由 A/D 转换器来完成,而卷积过程由卷积器来实现。为了设计卷积器,首先要设计 RAM 和 A/D 转换器的 Verilog HDL 模型。在电子工业发达的国家,可以通过商业渠道得到非常准确的外围器件的虚拟模型。如果没有外围器件的虚拟模型,就需要仔细地阅读和分析 RAM 和 A/D 转换器的器件说明书自行编写。因为 RAM 和 A/D 转换器不是我们设计的硬件对象,所以需要的只是它们的行为模型,精确的行为模型需要认真细致地编写,并不比可综合模块容易编写。它们与实际器件的吻合程度直接影响设计的成功。在这里我们把重点放在卷积器的设计上,直接给出 RAM 和 A/D 转换器的 Verilog HDL 模型和它们的器件参数(见附录),读者可以对照器件手册,认真阅读 RAM 和 A/D 转换器的 Verilog HDL 模型。对 RAM 和 A/D 转换器的 Verilog HDL 模型的详细了解对卷积器的设计是十分必要的。

到目前为止,对设计模块要完成的功能比较明确了。总结如下:首先它要控制 A/D 变换器进行 AD 变换,从 A/D 变换器得到变换后的数字序列,然后对数字序列进行卷积,最后将结果存入 RAM。

2. 卷积器的设计

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

```

`timescale 100 ps/100 ps
module con1(address,indata,outdata,wr,nconvst,nbusy,enout1,enout2,CLK,reset,start);
    input clk          //采用 10 MHz 的时钟
    input reset        //复位信号
    input start        //因为 RAM 的空间是有限的,当 RAM 存满后采样和卷积都会停止
                      //此时给一个 start 的高电平脉冲将会开始下一次的卷积
    output nbusy;      //从 A/D 转换器来的信号表示转换器的忙或闲
    output wr;         //RAM 写控制信号
    output enout1, enout2; //enout1 是存储卷积低字节结果 RAM 的片选信号
                           //enout2 是存储卷积高字节结果 RAM 的片选信号
    output nconvst;    //给出 A/D 转换器的控制信号,命令转换器开始工作,低电平有效
    output address;    //地址输出

    input [7:0] indata; //从 A/D 转换器来的数据总线
    output[7:0] outdata; //写到 RAM 去的数据总线

    wire nbusy;
    reg wr;
    reg nconvst;
    reg enout1;

```

```

    enout2;
    reg[7:0] outdata;
    reg[10:0] address;
    reg[8:0] state;
    reg[15:0] result;
    reg[23:0] line;
    reg[11:0] counter;
    reg      high;
    reg[4:0] j;
    reg      EOC;

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

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

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

```

```

        state <= IDLE;
//START 状态控制 A/D 开始转换
START: if(EOC)
begin
    nconvst <=0;
    high <=0;
    state <= NCONVST;
end
else
    state <= START;
//NCONVST 状态是 A/D 转换保持阶段
NCONVST: begin
    nconvst <=1;
    state <= READ;
end

//READ 状态读取 A/D 转换结果,计算卷积结果
READ: begin
    if(EOC)
begin
    line <={line[15:0],indata};
    state <= CALCU;
end
else
    state <= READ;
end

CALCU: begin
    result <=line[7:0] * h1+line[15:8] * h2+line[23:16] * h3;
    state <= WRREADY;
end

//将卷积结果写入 RAM 时,先写入低字节,再写入高字节
//WRREADY 状态是写 RAM 准备状态,建立地址和数据信号
WRREADY: begin
    address <=counter;
    if(! high)  outdata <=result[7:0];
    else        outdata <=result[15:8];
    state <= WR;
end

//WR 状态产生片选和写脉冲
WR: begin
    if(! high)  enout1 <=0;
    else        enout2 <=0;

```

```

        wr <= 0;
        state <= WREND;
    end
    //WREND 状态结束一次写操作,若还未写入高字节则转到 WRREADY 状态
    //态开始高字节写入
    WREND:begin
        wr <= 1;
        enout1 <= 1;
        enout2 <= 1;
        if(! high)
            begin
                high <= 1;
                state <= WRREADY;
            end
        else    state <= WAITFOR;
    end
    //WAITFOR 状态控制采样频率并判断 RAM 是否已被写满
    WAITFOR: begin
        if(j == FMAX - 1)
            begin
                counter <= counter + 1;
                if(! counter[11])   state <= START;
                else
                    begin
                        state <= IDLE;
                        $display( $time, "The ram is used up.");
                        $stop;
                    end
            end
        else    state <= WAITFOR;
    end
    default:state <= IDLE;
endcase
// assign rd=1; //RAM 的读信号始终保持高电平
//记录时钟,与 FMAX 共同控制采样频率,由于直接用 CLK 的上升沿对 nbusy 判断,以决定某些操作是否运行时,会因为两个信号的跳变沿相隔太近而令状态机不能正常工作,
//因此利用 CLK 的下降沿建立 EOC 信号与 nbusy 同步,相位相差 180°,然后用 CLK 的上升沿判断操作是否进行
always @(negedge CLK )
begin
    EOC <= nbusy;
    if(! reset | state==START)

```

```

j <=1;
else
j <=j+1;
end
endmodule

```

3. 前仿真及后仿真

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

```

//----- testcon1.v -----
`timescale 100 ps/100 ps
module testcon1;
    wire wr,
          enin,
          enout1,
          enout2;
    wire[10:0] address;
    reg rd,
             CLK,
             reset,
             start;
    wire nbusy;
    wire nconvst;
    wire[7:0] indata;
    wire[7:0] outdata;
    integer i;

parameter HALF_PERIOD=1000;

//产生 10 kHz 的时钟
initial
begin
    rd=1;
    i=0;
    CLK=1;
    forever # HALF_PERIOD CLK=~CLK;
end
//产生置位信号
initial
begin
    reset=1;
    #(HALF_PERIOD * 2 + 50) reset=0;
end

```

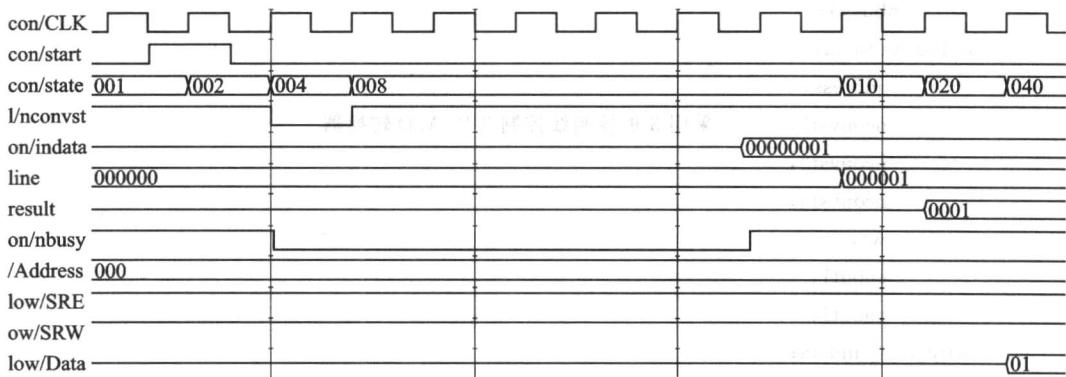
```

$> ./convtest # (HALF_PERIOD * 3) reset=1;
end
//产生开始卷积控制信号
initial
begin
    start=0;
# (HALF_PERIOD * 7 + 20) start=1;
# (HALF_PERIOD * 2) start=0;
# (HALF_PERIOD * 1000) start=1;
# (HALF_PERIOD * 2) start=0;
end
assign enin = 1;
con1 con(. address(address),. indata(indata),. outdata(outdata),. wr(wr),
          . nconvst(nconvst),. nbusy(nbusy),. enout1(enout1),
          . enout2(enout2),. CLK(CLK),. reset(reset),. start(start));
sram ramlow(. Address(address),. Data(outdata),. SRW(wr),. SRG(rd),. SRE(enout1));
adc adc(. nconvst(nconvst),. nbusy(nbusy),. data(indata));
endmodule

```

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

实验图 11 是一个参考波形图,由此可以看清整个程序的时序。



实验图 11 参考波形图

如果前仿真通过了,则可以做后仿真了。后仿真考虑了器件的延时,更具可靠性。首先用综合器(如:Synplify)进行综合。在综合时应注意选择器件库,如 Altera FLEX10K 系列 FPGA 或其他类型的 FPGA。综合完后生成了与原程序名相应的一个扩展名为 edf 的文件。然

后用布线工具(如:MAX+PLUS II ver. 9.3)对刚才得到的扩展名为 edf 的文件进行编译。如果编译不出错就可得到扩展名为 vo 的两个文件;一个文件名与原文件名相同,另一个文件名为 alf_max2.vo。

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

4. 卷积器的改进

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

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

```

reg[7:0] outdata;
reg[10:0] address;
reg nconvst1,
          nconvst2,
          nconvst3,
          wr,
          enout1,
          enout2;
reg[6:0] state;
reg[5:0] i;
reg[1:0] j;
reg[11:0] counter;
reg[23:0] line;
reg[15:0] result;
reg high;
reg k;
reg EOC1, EOC2, EOC3;

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

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

```

```

begin
    counter <=0;
    state <=READ_PRE;
end
else      state <=IDLE;

READ_PRE: if(EOC1||EOC2||EOC3) //由于频率相对改进前的卷积器
//大大提高,所以加入 READ_PRE
//状态对取数操作
//加以缓冲
    state <=READ;
else
    state <=READ_PRE;

READ:begin
    high <=0;
    enout2 <=1;
    wr <=1;
    if(j==1)
        begin
            if(EOC1)
                begin
                    line <={line[15:0],indata};
                    state <=CALCU;
                end
            else      state <=READ_PRE;
        end
    else if(j==2&&.counter!=0)
        begin
            if(EOC2)
                begin
                    line <={line[15:0],indata};
                    state <=CALCU;
                end
            else      state <=READ_PRE;
        end
    else if(j==3&&.counter!=0)
        begin
            if(EOC3)
                begin
                    line <={line[15:0],indata};
                    state <=CALCU;
                end
            else      state <=READ_PRE;
        end
    end
end

```

```

        else    state <=READ_PRE;
      end
      else    state <=READ;      //读取数据阶段
    end
    else    state <=WRREADY;   //写入准备阶段
  end
end

CALCU:begin
  result <=line[7:0] * h1+line[15:8] * h2+line[23:16] * h;
  state <=WRREADY;
end

WRREADY:begin
  wr <=1;
  address <=counter;
  if(k==1)state <=WR;
  else      state <=WRREADY;
end

WR: begin
  if(! high) enout1 <=0;
  else      enout2 <=0;
  wr <=0;
  if(! high) outdata <=result[7:0];
  else      outdata <=result[15:8];
  if(k==1)   state <=WREND;
  else      state <=WR;
end

WREND:begin
  wr <=1;
  enout1 <=1;
  enout2 <=1;
  if(k==1)
    if(! high)
      begin
        high <=1;
        state <=WRREADY;
      end
    else
      begin
        counter <=counter+1;
        if(counter[11]&&counter[0])
          state <=IDLE;
        else    state <=READ_PRE;
      end
  else    state <=WREND;
end

```

```

        end
    default:state <= IDLE;
endcase //end of the case
end // end of "else"
end // end of "always"

//计数器 i 用来记录时间
always @(posedge CLK)
begin
if(! reset) i <=0;
else
begin
if(i==44) i <=0;
else      i <=i+1;
end
end

//j 是控制信号,协调卷积器轮流从 3 片 A/D 上读取数据
always @(posedge CLK)
begin
if(i==4) j <=2;
else if(i==10) j <=0;
else if(i==19) j <=3;
else if(i==25) j <=0;
else if(i==34) j <=1;
else if(i==40) j <=0;
end

//k 是计数器,用以控制写操作信号
always @(posedge CLK)
begin
if(state==WRREADY||state==WR||state==WREND)
if(k==1) k <=0;
else      k <=1;
else   k <=0;
end

//根据计数器 i 控制 3 片 A/D 转换信号 NCONVST1,NCONVST2,NCONVST3
always @(posedge CLK)
begin
if(! reset) nconvst1 <=1;
else if(i==0) nconvst1 <=0;
else if(i==3) nconvst1 <=1;
end

```

```

always @(posedge CLK)
begin
    if( ! reset) nconvst2 <=1;
    else if(i==15) nconvst2 <=0;
    else if(i==18) nconvst2 <=1;
end

always @(posedge CLK)
begin
    if( ! reset) nconvst3 <=1;
    else if(i==30) nconvst3 <=0;
    else if(i==33) nconvst3 <=1;
end

always @ (negedge CLK)
begin
    EOC1 <= nbusy1;
    EOC2 <= nbusy2;
    EOC3 <= nbusy3;
end
endmodule

```

测试程序如下：

```

'timescale 1 ns/100 ps

module testcon3ad;
    wire wr,
    wire enin,
    wire enout1,
    wire enout2;
    wire[10:0] address;
    reg clk,
    reg reset,
    reg start,
    reg rd;
    wire nbusy1,
    wire nbusy2,
    wire nbusy3;
    wire nconvst1,
    wire nconvst2,
    wire nconvst3;
    wire[7:0] indata;

```

```

wire[7:0] outdata;
parameter HALF_PERIOD=15;//时钟周期为 30 ns

initial
begin
    clk=1;
    forever # HALF_PERIOD clk=~clk;
end

initial
begin
    reset=1;
    # 110 reset=0;
    # 140 reset=1;
end

initial
begin
    start=0;
    rd=1;
    # 420 start=1;
    # 120 start=0;
    # 107600 start=1;
    # 150     start=0;
end

assign enin=1;

con3ad con3ad(.indata(indata),.outdata(outdata),.address(address),
    .CLK(clk),.reset(reset),.start(start),
    .nconvst1(nconvst1),.nconvst2(nconvst2),.nconvst3(nconvst3),
    .nbusy1(nbusy1),.nbusy2(nbusy2),.nbusy3(nbusy3),
    .wr(wr),.enout1(enout1),.enout2(enout2));

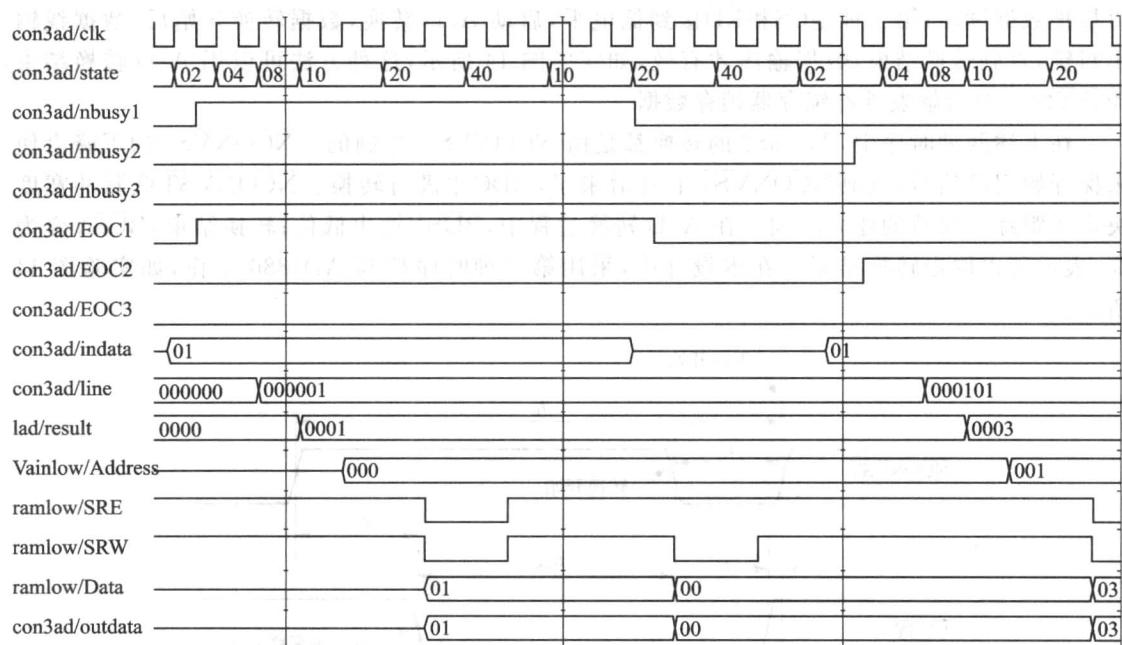
sram ramlow(.Address(address),.Data(outdata),.SRW(wr),.SRG(rd),.SRE(enout1));

adc ad_1(.nconvst(nconvst1),.nbusy(nbusy1),.data(indata));
adc ad_2(.nconvst(nconvst2),.nbusy(nbusy2),.data(indata));
adc ad_3(.nconvst(nconvst3),.nbusy(nbusy3),.data(indata));
endmodule

```

与前面一样,给出部分仿真波形供大家参考,如实验图 12 所示。

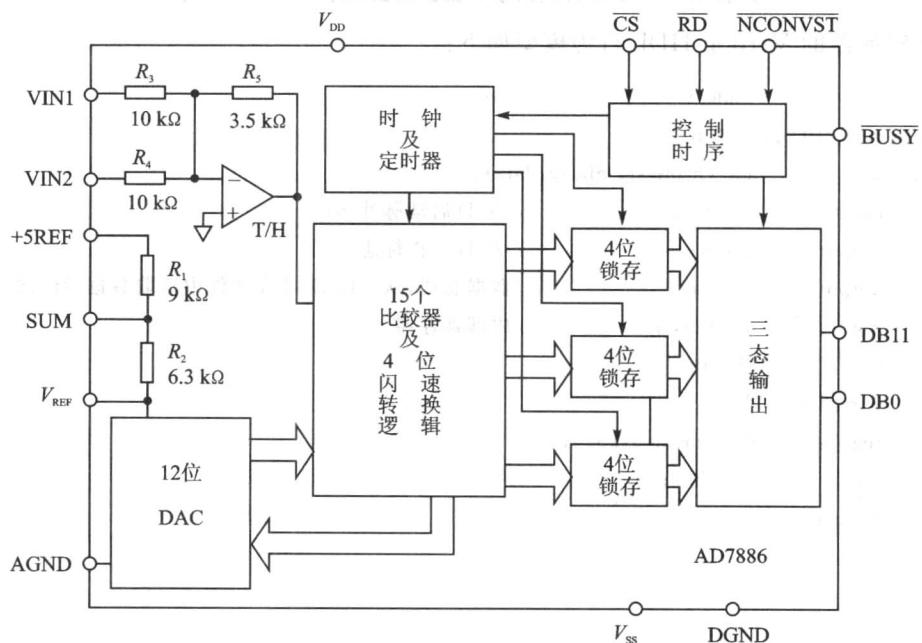
经过前仿真确定的设计在逻辑上成立后,同样可以对这个改进后的卷积器使用合适的器件库进行后仿真,步骤与前面讲述的完全一样,在这里就不再赘述了。



实验图 12 部分仿真波形

附录一 A/D 转换器的 Verilog HDL 模型机所需要的技术参数

这里所采用 A/D 转换器 AD7886，其功能图如实验图 13 所示。

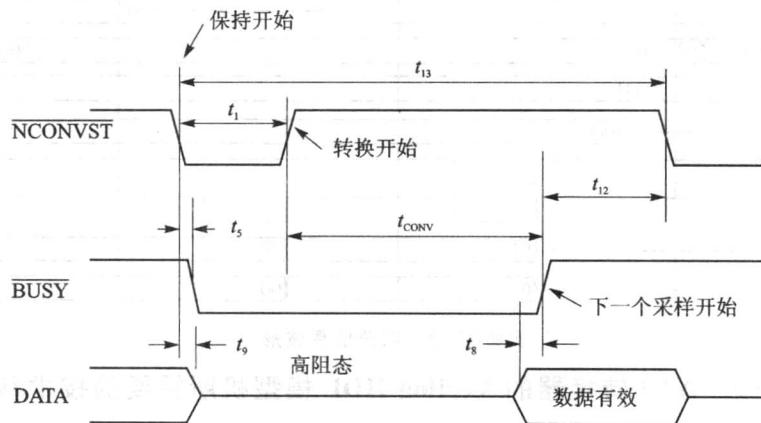


实验图 13 AD7886 的功能图

AD7886 的时序控制有两种方法：第一种是 \overline{CS} 和 \overline{RD} 输入信号控制 AD7886 的三态输出（如图中所示），但 A/D 转换中三态输出封锁，这种方法适合微处理器在 AD7886 转换结束后

直接把数据读出;第二种是 \overline{CS} 和 \overline{RD} 接到低电平,启动A/D转换,数据转换开始后,数据线输出封锁,直到转换结束,数据输出才有效,如实验图14所示,这种方法可以用A/D转换结束 \overline{BUSY} 的上升沿触发外部锁存器锁存数据。

在上述两种时序中,AD7886的转换都是由NCONVST控制的。NCONVST的下降沿使采保开始跟踪信号,直到NCONVST上升沿来了,ADC才进行转换。 $\overline{NCONVST}$ 低脉冲宽度决定了跟踪一保持的建立时间。在A/D转换过程中, \overline{BUSY} 输出低位;转换结束, \overline{BUSY} 变为高,表示可以取走转换结果。在本设计中,采用第二种时序控制AD7886工作,如实验图14所示。



实验图14 A/D转换启动和数据读出时序 ($CS=RD=0$)

A/D转换器的Verilog HDL行为模型如下：

```
//----- adc.v -----
`timescale 100 ps/100 ps

module      adc (nconvst, nbusy, data);
    input      nconvst;           // A/D启动脉冲 ST
    output     nbusy;            // A/D工作标志
    output     data;             // 数据总线,从 AD.DATA 文件中读取数据后经端口输出
    reg[7:0]   databuf,i;       // 内部寄存器
    reg        nbusy;
    wire[7:0]  data;
    reg[7:0]   data_mem[0:255];
    reg        link_bus;
    integer    tconv,
                t5,
                t8,
                t9,
                t12;
    integer    widtht1,
                widtht2,
                widtht;
```

```

//时间参数定义(依据 AD7886 手册):
always @(negedge nconvst)
begin
    tconv = 9500 + { $random } % 500; // (type 950, max 1000ns) Conversion Time
    t5 = { $random } % 1000; // (max 100 ns) CONVST to BUSY Propagation Delay
                           // CL = 10 pF
    t8 = 200;           // (min 20 ns) CL=20pF Data Setup Time Prior to BUSY
                           // (min 10 ns) CL=100pF
    t9 = 100 + { $random } % 900;
                           // (min 10ns, max 100ns) Bus Relinquish Time After CONVST

    t12 = 2500;        // (type) BUSY High to CONVST Low, SHA Acquisition Time
end

initial
begin
    $readmemh("adc.data",data_mem); // 从数据文件 adc.data 中读取数据
    i = 0;
    nbusy = 1;
    link_bus = 0;
end

assign data = link_bus? databuf:8'bzz; // 三态总线
/* -----
在信号 NCONVST 的负跳降沿到来后,隔  $t_5$  秒后,使 NBUSY 信号置为低, $t_{conv}$  是 AD 将模拟
信号转换为数字信号的时间,在信号 NCONVST 的正跳降沿到来后经过  $t_{conv}$  时间后,输出
NBUSY 信号由低变为高。
----- */
always @(negedge nconvst)
fork
    #t5 nbusy = 0;
    @(posedge nconvst)
begin
    #tconv nbusy = 1;
end
join
/* -----
NCONVST 信号的下降沿触发,经过  $t_9$  延时后,把数据总线输出关闭并置为高阻态,如图中
所示。NCONVST 信号的上升沿到来后,经过( $t_{conv} - t_8$ )时间,输出一个字节(8 位数据)到数
据缓冲器 databuf,该数据来自于 data_mem。而 data_mem 中的数据是初始化时从数据文件
AD.DAT 中读取的,此时应启动总线的三态输出。
----- */

```

NCONVST 信号的下降沿触发,经过 t_9 延时后,把数据总线输出关闭并置为高阻态,如图中所示。NCONVST 信号的上升沿到来后,经过($t_{conv} - t_8$)时间,输出一个字节(8 位数据)到数据缓冲器 databuf,该数据来自于 data_mem。而 data_mem 中的数据是初始化时从数据文件 AD.DAT 中读取的,此时应启动总线的三态输出。

```
----- */
always @ (negedge nconvst)
begin
    @(posedge nconvst)
    begin
        # (tconv-t8) databuf = data_mem[i];
    end
    if(wideth <10000 && wideth>500)
    begin
        if(i==255) i=0;
        else i=i+1;
    end
    else i = i;
end
```

```
//在模数转换期间关闭三态输出,转换结束时启动三态输出
always @ (negedge nconvst)
fork
    # t9 link_bus = 1'b0; //关闭三态输出,不允许总线输出
    @(posedge nconvst)
    begin
        # (tconv-t8) link_bus=1'b1;
    end
end
```

当 NCONVST 输入信号的下一个转换的下降沿与 NBUSY 信号上升沿之间时间延迟小于 t_{12} 时,出现警告信息,通知设计者请求转换的输入信号频率太快,A/D 器件转换速度跟不上。仿真模型不仅能够实现硬件电路的输出功能,同时能够对输入信号进行检测,当输入信号不符合手册要求时,显示警告信息。

```
----- */
//检查 A/D 启动信号的频率是否太快
always @ (posedge nbusy)
begin
    # t12;
    if (! nconvst)
        begin
            $display("Warning! SHA Acquisition Time is too short!");
        end
    else
        begin
            $display(" SHA Acquisition Time is enough!");
        end
end
```

```

end

// 检查 A/D 启动信号的负脉冲宽度是否足够和太宽
always @(negedge nconvst)
begin
    wideth= $ time;
    @(posedge nconvst) wideth= $ time-wideth;
    if (wideth<=500 || wideth > 10 000)
        begin
            $ display("nCONVST Pulse Width = %d",wideth);
            $ display("Warning! nCONVST Pulse Width is too narrow or too wide!");
            // $ stop;
        end
end
endmodule

```

附录二 2K * 8 位 异步 CMOS 静态 RAM HM - 65162 模型

```

/*
 * File Name : sram.v
 * Function  : 2K * 8bit Asynchronous CMOS Static RAM
 */
/*
 * Module Name : sram
 * Description : 2K * 8bit Asynchronous CMOS Static RAM
 * Reference : HM - 65162 reference book
 */
/*
 * sram is a Verilog HDL model for HM - 65162, 2K * 8bit Asynchronous CMOS Static RAM. It is used in simulation to substitute the real RAM to verify whether the writing or reading of the RAM is OK. This module is a behavioral model for simulation only, not synthesizable. Its writing and reading function are verified.
 */
//----- sram.v -----
module sram(Address, Data, SRG, SRE, SRW);
    input [10:0] Address;
    input          Data;
    input          SRG; // Output enable
    input          SRE; // Chip enable
    input          SRW; // Write enable

```

```

inout [7:0] Data; // Bus

wire [10:0] Addr = Address;
reg [7:0] RdData;
reg [7:0] SramMem [0:'h7ff];
reg [7:0] RdSramDly, RdFlip;
wire [7:0] FlpData, Data;
reg WR_flag; //To judge the signals according to the specification of HM - 65162
integer i;

wire RdSram = ~SRG & ~SRE;
wire WrSram = ~SRW & ~SRE;
reg [10:0] DelayAddr;
reg [7:0] DelayData;
reg WrSramDly;

integer file;

assign FlpData = (RdFlip) ? ~RdData : RdData;
assign Data = (RdSramDly) ? FlpData : 'hz;
// ***** parameters of read circle *****
//参数序号、最大或最小、参数含义
parameter TAVQV=90, //2 (max) Address access time
         TELQV=90, //3 (max) Chip enable access time
         TELQX=5, //4 (min) Chip enable output enable time
         TGLQV=65, //5 (max) Output enable access time
         TGLQX=5, //6 (min) Output inable output enable time
         TEHQZ=50, //7 (max) Chip enable output disable time
         TGHQZ=40, //8 (max) Output enable output disable time
         TAVQX=5; //9 (min) Output hold from address change

// ***** parameters of write circle *****
parameter TAVWL=10, //12 (min) Address setup time,
          TWLWH=55, //13 (min) Chip enable pulse setup time,
                      //write enable pluse width,
          TWHAX=15, //14 (min10) Write enable read setup time,
                      //读上升沿后地址保留时间
          TWLQZ=50, //16 (max) Write enable output disable time
          TDVWH=30, //17 (min) Data setup time
          TWHDX=20, //18 (min15) Data hold time
          TWHQX=20, //19 (min0) Write enable output enable time,0
          TWLEH=55, //20 (min) Write enable pulse setup time
          TDVEH=30, //21 (min) Chip enable data setup time

```

```

TAVWH=70; //22 (min65) Address valid to end of write

initial
begin
    file= $ fopen("ramlow.txt");
    if(!file)
        begin
            $ display("Could not open the file.");
            $ stop;
        end
    end
end

initial
begin
    for(i=0 ; i<'h7ff ; i=i+1)
        SramMem[i] = i;
// $ monitor( $ time, "DelayAddr=%h,DelayData=%h",DelayAddr,DelayData);
end

initial RdSramDly = 0;
initial WR_flag=1;

/* **** READ CIRCLE ****/
always @(posedge RdSram) # TGLQX RdSramDly = RdSram;
always @(posedge SRW) # TWHQX RdSramDly = RdSram;
always @(Addr)
begin
    # TAVQX;
    RdFlip = 1;
    #(TGLQV - TAVQX); //address access time
    if (RdSram) RdFlip = 0;
end

always @(posedge RdSram)
begin
    RdFlip = 1;
    # TAVQV; // Output enable access time
    if (RdSram) RdFlip = 0;
end

always @(Addr) # TAVQX RdFlip = 1;

always @(posedge SRG) # TEHQZ RdSramDly = RdSram;

```

```

always @(posedge SRE)      # TGHQZ     RdSramDly = RdSram;
always @(negedge SRW)      # TWLQZ     RdSramDly = 0;

always @(negedge WrSramDly or posedge RdSramDly)   RdData = SramMem[Addr];

// **** WRITE CIRCLE ****
always @(Addr)      # TAVWL DelayAddr = Addr; //Address setup
always @(Data)       # TDVWH DelayData = Data; //Data setup
always @(WrSram)    # 5 WrSramDly = WrSram;
always @(Addr or Data or WrSram) WR_flag=1;

always @(negedge SRW)
begin
    # TWLWH;           //Write enable pulse width
    if (SRW)
        begin
            WR_flag=0;
            $display("ERROR! Can't write!");
            Write enable time (W) is too short!";
        end
    end

always @(negedge SRW)
begin
    # TWLEH;           //Write enable pulse setup time
    if (SRE)
        begin
            WR_flag=0;
            $display("ERROR! Can't write! Write enable
pulse setup time (E) is too short!");
        end
    end

always @(posedge SRW)
begin
    # TWHAX;           //Write enable read setup time
    if(DelayAddr != Addr)
        begin
            WR_flag=0;
            $display("ERROR! Can't write!
Write enable read setup time is too short!");
        end
    end

```

```

always @(Data)
if (WrSram)
begin
# TDVEH;           //Chip enable data setup time
if (SRE)
begin
WR_flag=0;
$display("ERROR! Can't write!");
Chip enable Data setup time is too short!");
end
end

```

```

always @(Data)
if (WrSram)
begin
# TDVEH;
if (SRW)
begin
WR_flag=0;
$display("ERROR! Can't write!");
Chip enable Data setup time is too short!");
end
end

```

```

always @ (posedge SRW )
begin
# TWHDX;           //Data hold time
if(DelayData != Data)
$display("Warning! Data hold time is too short!");
end

```

```

always @(DelayAddr or DelayData or WrSramDly)
if (WrSram && WR_flag)
begin
if(! Addr[5])
begin
# 15 SramMem[Addr]=Data;
// $ display("mem[%h]=%h", Addr, Data);
$ fwrite(file, "mem[%h]=%h", Addr, Data);
if(Addr[0]&&Addr[1]) $ fwrite(file, "\n");
end
else

```

```

begin
    $fclose(file);
    $display("Please check the txt.");
    $stop;
end
end

endmodule

```

【练习题 11】 参考另外一种 AD 变换器手册,模仿本示范题的 AD 虚拟模块,编写该 AD 变换器全功能的虚拟模块;参考另一种静态 RAM 手册,编写完整的静态 RAM 虚拟模块;根据这两种新器件的时序设计符合工程要求的卷积器。

练习十二 利用 SRAM 设计一个 FIFO

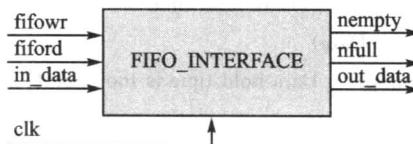
目的:

- (1) 学习和掌握存取队列管理的状态机设计的基本方法;
- (2) 了解并掌握用存储器构成 FIFO 的接口设计的基本技术;
- (3) 用工程概念来编写完整的测试模块,达到完整测试覆盖。

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

1. 设计要求

本练习要求同学设计的 FIFO 为同步 FIFO,即对 FIFO 的读/写使用同一个时钟。该 FIFO 应当提供用户读使能(fiford)和写使能(fifowr)输入控制信号,并输出指示 FIFO 状态的非空(nempty)和非满(nfull)信号,FIFO 的输入、输出数据使用各自的数据总线:in_data 和 out_data。实验图 15 为 FIFO 接口示意图。



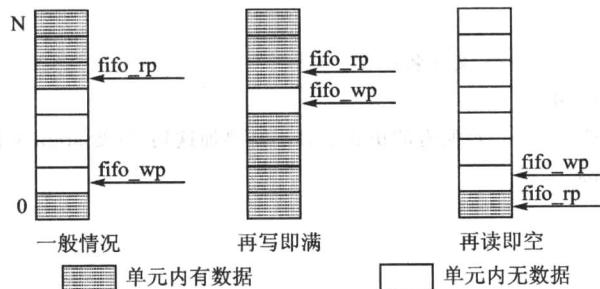
实验图 15 FIFO 接口示意

2. FIFO 接口的设计思路

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

我们可以借用软件中的方法,将 FIFO 抽象为环形数组,并用两个指针,即读指针(fifo_rp)和写指针(fifo_wp)控制对该环形数组的读写。其中,读指针 fifo_rp 指向下次读操作所要读取的单元,并且每完成一次读操作,fifo_rp 加一;写指针 fifo_wp 则指向下次写操作时存放数据的单元,并且每完成一次写操作,fifo_wp 加一。由 fifo_rp 和 fifo_wp 的定义可知,当 FIFO 被读空或写满后,fifo_rp 和 fifo_wp 将指向同一单元,但在读空和写满之前 FIFO 的状态是不同的,所以如果能区分这两种状态,再通过比较 fifo_rp 和 fifo_wp 就可以得到 nempty

和 nfull 信号了。实验图 16 为 FIFO 工作状态的示意。

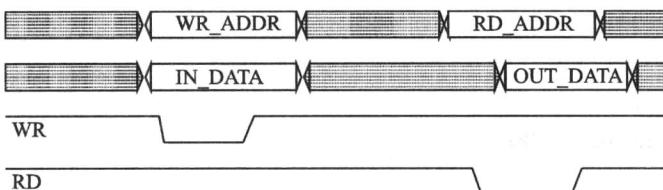


实验图 16 FIFO 的工作状态

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

此外,在进行 SRAM 的读写操作时,应该注意建立地址、数据和控制信号的先后顺序。一般情况下,实验图 17 为 SRAM 读写时序波形图。

在写 SRAM 时,先建立地址和数据,然后置写使能信号 WR,使其有效。在 WR 保持一定时间的有效后,先复位 WR,再释放地址总线和数据总线。而读取 SRAM 时,则先建立地址,然后置读使能信号 RD 有效,在 RD 维持一定时间有效后,复位 RD;同时读取数据总线上的值,然后再释放地址总线。在进行 FIFO 操作时,用户一般希望除了没有地址外,其他三个信号的时序关系能保持不变。在设计 FIFO 控制信号与 SRAM 控制信号间的逻辑关系时应注意这一点。



实验图 17 SRAM 读写时序

3. FIFO 接口的测试

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

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

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

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

```

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

module t;
    reg [7:0]      in_data;          //FIFO 数据总线
    reg           fiford,fifowr;     //FIFO 控制信号
    wire[7:0]       out_data;        //FIFO 数据总线
    wire           nfull, nempty;    //FIFO 状态信号
    reg             clk,rst;         //时钟和复位信号
    wire[7:0]       sram_data;       //SRAM 数据总线
    wire[10:0]      address;        //SRAM 地址总线
    wire            rd,wr;          //SRAM 读写控制信号
    reg [7:0]       data_buf['FIFO_SIZE:0]; //数据缓存,用于结果检查
    integer index;               //用于读写 data_buf 的指针

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

    //测试激励序列
    initial
        begin
            fiford=1;
            fifowr=1;
            rst=1;
            # 40 rst=0;
            # 42 rst=1;
            if (nempty) $display($time,"Error: FIFO be empty, nempty should be low.\n");
        end
    //----- 连续写 FIFO -----
    index = 0;
    repeat('FIFO_SIZE)

```

```

begin
    data_buf[index] = $ random;
    write_fifo(data_buf[index]);
    index = index + 1;
end

if (nfull) $ display($ time, "Error: FIFO full, nfull should be low. \n");
repeat(2) write_fifo($ random);
# 200

//-----连续读 FIFO-----
index=0;
read_fifo_compare(data_buf[index]);
if (~nfull) $ display($ time, "Error: FIFO not full, nfull should be high. \n");

repeat('FIFO_SIZE-1)
begin
    index = index + 1;
    read_fifo_compare(data_buf[index]);
end

if (nempty) $ display($ time, "Error: FIFO be empty, nempty should be low. \n");
repeat(2) read_fifo_compare(8'bx);

reset_fifo;

//----写后读 FIFO-----
repeat('FIFO_SIZE * 2)
begin
    data_buf[0] = $ random;
    write_fifo(data_buf[0]);
    read_fifo_compare(data_buf[0]);
end

//---- 异常操作-----
reset_fifo;
read_fifo_compare(8'bx);
write_fifo(data_buf[0]);
read_fifo_compare(data_buf[0]);

$ stop;
end

```

```

fifo_interface fifo_mk (.in_data(in_data),
                      .out_data(out_data),
                      .fifo_rd(fiford),
                      .fifo_wr(fifowr),
                      .nfull(nfull),
                      .nempty(nempty),
                      .address(address),
                      .sram_data(sram_data),
                      .rd(rd), .wr(wr),
                      .clk(clk), .rst(rst));
endinterface

sram m1 (.Address(address),
          .Data(sram_data),
          .SRG(rd), //SRAM 读使能
          .SRE(1'b0), //SRAM 片选,低有效
          .SRW(wr)); //SRAM 写使能

task write_fifo;
  input [7:0] data;
  begin
    in_data=data;
    #50    fifowr=0; //往 SRAM 中写数
    #200   fifowr=1;
    #50;
  end
endtask

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

task reset_fifo;
  begin

```

```

#40 rst=0;
#40 rst=1;
end
endtask

endmodule

```

4. FIFO 接口的参考设计

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

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

```

module fifo_interface(
    in_data,                                //用户的输入数据总线
    out_data,                                //用户的输出数据总线
    fiford,                                   //FIFO 读控制信号,低电平有效
    fifowr,                                   //FIFO 写控制信号,低电平有效
    nfull,                                   
    nempty,                                 
    address,                                  //到 SRAM 的地址总线
    sram_data,                               //到 SRAM 的双向数据总线
    rd,                                       //SRAM 读使能,低电平有效
    wr,                                       //SRAM 写使能,低电平有效
    clk,                                      //系统时钟信号
    rst );                                     //全局复位信号,低电平有效

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

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

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

//输出到 SRAM 的控制信号
output      rd, wr;

```

```

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

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

//Internal Register
reg[10:0]        fifo_wp,           //FIFO 写指针
                  fifo_rp;          //FIFO 读指针
reg[10:0]        fifo_wp_next,     //fifo_wp 的下一个值
                  fifo_rp_next;    //fifo_rp 的下一个值

reg              near_full, near_empty;

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

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

//SRAM 操作状态机
always @(posedge clk or negedge rst)
if (~rst)
  state <= idle;
else
  case(state)
    idle:                      //等待 FIFO 的操作控制信号
      if (fifowr==0 && nfull)   //用户发出写 FIFO 申请,且 FIFO 未满
        state<=write_ready;
      else if(fiford==0 && nempty) //用户发出读 FIFO 申请,且 FIFO 未空
        state<=read_ready;
      else                      //没有对 FIFO 操作的申请
        state<=idle;
    read_ready:                 //建立 SRAM 操作所需地址和数据
      state <= read;
  endcase
end

```

```

read;                                //等待用户结束当前读操作
    if (fifo_rd == 1)
        state <= read_over;           //继续给出 SRAM 地址以保证数据稳定
    else
        state <= read;

read_over;                            //继续给出 SRAM 地址以保证数据稳定
    state <= idle;

write_ready;                          //建立 SRAM 操作所需地址和数据
    state <= write;               //等待用户结束当前写操作
if (fifo_wr == 1)
    state <= write_over;          //继续给出 SRAM 地址和写入数据以保证数据稳定
else
    state <= write;

write;                                //等待用户结束当前写操作
    if (fifo_wr == 1)
        state <= write_over;      //继续给出 SRAM 地址和写入数据以保证数据稳定
    else
        state <= idle;

default: state<=idle;
endcase

//产生 SRAM 操作相关信号
assign rd = ~state[2];                //state 为 read_ready 或 read 或 read_over
assign wr = (state == write) ? fifo_wr : 1'b1;

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

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

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

```

```

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

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

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

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

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

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

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

always @(posedge clk or negedge rst)

```

```

        if ( $\sim$ rst)
    nempty  $\leq$  1'b0;
else if (near_empty && state == read)
    nempty  $\leq$  1'b0;
else if (state == write)
    nempty  $\leq$  1'b1;

always @ (posedge clk or negedge rst)
if ( $\sim$ rst)
    near_full  $\leq$  1'b0;
else if (fifo_rp == fifo_wp_next)
    near_full  $\leq$  1'b1;
else
    near_full  $\leq$  1'b0;

always @ (posedge clk or negedge rst)
if ( $\sim$ rst)
    nfull  $\leq$  1'b1;
else if (near_full && state == write)
    nfull  $\leq$  1'b0;
else if (state == read)
    nfull  $\leq$  1'b1;
endmodule

```

【练习题 12】 模仿上题的设计思路和方法,设计一个利用同类静态 RAM 的堆栈,即实现最大可达 1 024 个字节的 LIFO (Last In First Out) 堆栈。

第四部分 语法篇

语法篇 1 关于 Verilog HDL 的说明

一、关于 IEEE 1364 标准

本 Verilog 硬件描述语言参考手册是根据 IEEE 的标准“Verilog 硬件描述语言参考手册 1364—1995”编写的。OVI (Open Verilog International)根据 Cadence 公司推出的 Verilog LRM(1.6 版)编写了 Verilog 参考手册 1.0 和 2.0 版;OVI 又根据以上两个版本制定了 IEEE1364—1995 Verilog 标准。在推出 Verilog 标准前,由于 Cadence 公司的 Verilog - XL 仿真器被广泛使用,它所提供的 Verilog LRM 成了事实上的语言标准。许多第三方厂商的仿真器都努力向这一已成事实的标准靠拢。

Verilog 语言标准化的目的是将现存的通过 Verilog - XL 仿真器体现的 Verilog 语言标准化。IEEE 的 Verilog 标准与事实上的标准有一些区别。因此,仿真器有可能不完全支持以下的一些功能。

- (1) 在 UDP(用户自定义原语)和模块实例中使用数组(见 Instantiation 说明)。
- (2) 含参数的宏定义(见 `define)。
- (3) `undef 的使用。
- (4) IEEE 标准不支持用数字表示的强度值(见编译预处理命令)。
- (5) 有许多 Verilog - XL 支持的系统任务、系统函数和编译处理命令在 IEEE 标准中不支持。
- (6) 若在模块中,Net 类型或寄存类型变量只有一个驱动,IEEE 标准允许在一个指定块中,延迟路径的最终节点可以是一个寄存器或 Net 类型的变量。而在此标准推出之前,对最终节点的类型有着严格的要求(见 Specify 说明)。
- (7) 指定路径的延迟表达式最多可以达到 12 个,表达式之间需用逗号隔开。而在此标准推出之前,最多只允许 6 个表达式(见 Specify 说明)。
- (8) 在 Net 类型变量的定义中,标量保留字 scalared 与矢量保留字 vectored 的位置也做了改动。原先,保留字位于矢量范围的前面,在 IEEE 标准中,它应位于 Net 类型的后面(见 Net 说明)。
- (9) 在最小—典型—最大常量表达式中,对于最小、典型与最大值的相对大小并无限制;而原先最小值必须小于或等于典型值,典型值必须小于或等于最大值。

(10) 在 IEEE 标准中,表示延迟的最小—典型—最大表达式不必括在括号里,而原先必须括在括号里。

二、Verilog 简介

在 Verilog HDL 中,可通过高层模块调用低层和基本元件模块,再通过线路连接(即下文中的 Net),把这些具体的模块连接在一起,来描述一个极其复杂的数字逻辑电路的结构。所谓基本元件模块就是各种逻辑门和用户定义的原语模块(即下文中的 UDP),而所谓 Net 实质上就是表示电路连线或总线的网络。端口连接列表用来把外部 Net 连接到模块的端口(即引脚)上。寄存器可以作为输入信号连接到某个具体模块的入口。Net 和寄存器的值可取逻辑值 0,1,x(不确定)和 z(高阻)。除了逻辑值外,Net 还需要有一个强度(Strength)值。在开关级模型中,当 Net 的驱动器不止一个时,还需要使用强度值来表示。逻辑电路的行为可以用 initial 和 always 的结构和连续赋值语句,并结合设计层次树上各种层次的模块直到最底层的模块(即 UDP 及门)来描述。

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

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

三、语法总结

1. 典型的 Verilog 模块结构

```
module M (P1, P2, P3, P4);
    input P1, P2;
    output [7:0] P3;
    inout P4;
    reg [7:0] R1, M1[1:1024];
    wire W1, W2, W3, W4;
    parameter C1 = "This is a string";
    initial
        begin : 块名
            // 声明语句
        end
end
```

```

    always@ (触发事件)
        begin
            // 声明语句

            end
        // 连续赋值语句
        assign W1 = Expression;
        wire (Strong1, Weak0) [3:0] #(2,3) W2 = Expression;
        // 模块实例引用
        COMP U1 (W3, W4);
        COMP U2 (.P1(W3), .P2(W4));

    task T1;           // 任务定义
        input A1;
        inout A2;
        output A3;
        begin
            // 声明语句
            end
        endtask

    function [7:0] F1; // 函数定义
        input A1;
        begin
            // 声明语句
            F1 = 表达式;
        end
    endfunction

endmodule           // 模块结束

```

2. 声明语句

```

# delay
wait (Expression)
@(A or B or C)
@(posedge Clk)

Reg= Expression;
Reg <= Expression;

VectorReg[Bit] = Expression;
VectorReg[MSB:LSB] = Expression;
Memory[Address] = Expression;
assign Reg = Expression

```

```

deassign Reg;
TaskEnable(...);
disable TaskOrBlock;
EventName;

if (Condition)
  :
else if (Condition)
  :
else
  :
case (Selection)
  Choice1:
  :
  Choice2, Choice3:
  :
default:
  :
endcase

for (I=0; I<MAX; I=I+1)
  :
repeat (8)
  :
while (Condition)
  :
forever
  :

```

上面的简要语法总结可供读者快速查找,应注意其语法表示方法与本手册中其他地方的不同。

四、编写 Verilog HDL 源代码的标准

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

下面列出的代码编写标准可根据所选择的工具和个人的爱好自行做一些必要的改动。

1. 语汇代码的编写标准

- (1) 每一个 Verilog 源文件中只准编写一个模块,也不要在一个模块分成几部分或写在几个源文件中。
- (2) 源文件的名字应与文件内容有关,最好一致(例 ModuleName.v)。
- (3) 每行只写一个声明语句或说明。
- (4) 如上面的许多例子所示,用一层层缩进的格式来写。
- (5) 用户定义变量名的大小写应自始至终一致(例如,变量名第一个字母大写)。
- (6) 用户定义变量名应该是有意义的,而且含有一定的有关信息。而局部变量名(例如循环变量)可以是简单扼要的。
- (7) 通过注释对 Verilog 源代码作必要的说明,当然没有必要把 Verilog 源代码已能说明的再注释一遍,而对接口(例如模块参数、端口、任务、函数变量)做必要的注释很重要。
- (8) 尽可能多地使用参数和宏定义,而不要在源代码的语句中直接使用字母、数字和字符串。

2. 可综合代码的编写标准

- (1) 把设计分割成较小的功能块,每一块用行为风格去设计这些块。除了设计中对速度响应要求比较临界的部分外,都应避免使用门级描述。
- (2) 应建立一个定义得很好的时钟策略,并在 Verilog 源代码中清晰地体现该策略(例如采用单时钟、多相位时钟、经过门产生的时钟和多时钟域等)。保证在 Verilog 源代码中的时钟和复位信号是干净的(即不是由组合逻辑或没有考虑到的门产生的)。
- (3) 要建立一个定义得很好的测试(制造)策略,并认真编写其 Verilog 代码,使所有的触发器都是可复位的,使测试能通过外部引脚进行,又没有冗余的功能等。
- (4) 每个 Verilog 源代码都必须遵守并符合在 Always 声明语句中介绍过的某一种可综合标准模板。
- (5) 描述组合和锁存逻辑的 always 块,必须在 always 块开头的控制事件列表中列出所有的输入信号。
- (6) 描述组合逻辑的 always 块一定不能有不完全赋值,也就是说所有的输出变量必须被各输入值的组合值赋值,不能有例外。
- (7) 描述组合和锁存逻辑的 always 块一定不能包含反馈,也就是说在 always 块中已被定义为输出的寄存器变量绝对不能再在该 always 块中读进来作为输入信号。
- (8) 时钟沿触发的 always 块必须是单时钟的,并且任何异步控制输入(通常是复位或置位信号)必须在控制事件列表中列出。
- (9) 避免生成不想要的锁存器。在无时钟的 always 块中,由于有的输出变量被赋予某个信号的变量值,而该信号变量没在该 always 块的电平敏感控制事件中列出,这会在综合中生成不想要的锁存器。
- (10) 避免不想要的触发器。在时钟沿触发的 always 块中,用非阻塞的赋值语句对寄存器类型的变量赋值,综合后会生成触发器;或者当寄存器类型的变量在时钟沿触发的 always 块中经过多次循环其值仍保持不变,综合后也会生成触发器。
- (11) 所有内部状态寄存器必须是可复位的,这是为了使 RTL 级和门级描述能够被复位成同一个已知的状态以便进行门级逻辑验证,但这并不适用于流水线或同步寄存器。

(12) 对存在无效状态的有限状态机和其他时序电路(例如,四位十进制计数器有六个无效状态),如果想要在这些无效状态下,硬件的行为也能够完全被控制,那么必须用 Verilog 明确地描述所有的 2^n 次幂种状态下的行为,当然也包括无效状态。只有这样才能综合出安全可靠的状态机。

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

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

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

五、设计流程

用 Verilog 和综合工具设计 ASIC 或复杂 FPGA 的基本流程是:围绕着设计流程作多次反复是必要的,但下面的流程没有对此加以说明。而且设计流程必须根据所设计的器件和特定的应用做必要的改动。其基本流程如下:

(1) 系统分析和指标的确定。

(2) 系统划分:

- 顶级模块;

- 模块大小估计;

- 预布局。

(3) 模块级设计,即对每一模块:

- 写 RTL 级 Verilog;

- 综合代码检查;

- 写 Verilog 测试文件;

- Verilog 仿真;

- 写综合约束、边界条件和层次;

- 预综合以分析门的数量和延时。

(4) 芯片综合:

- 写 Verilog 测试文件;

- Verilog 仿真;

- 综合;

- 门级仿真。

(5) 测试:

- 修改门级网表以便进行测试;

- 产生测试矢量;

- 对可测试网表进行仿真。

(6) 布局布线以使设计的逻辑电路能放入芯片。

(7) 布局布线后仿真、故障覆盖仿真和定时分析。

语法篇 2 Verilog 硬件描述语言参考手册

一、Verilog HDL 语句与常用 标志符(按字母顺序排列)

1. always 声明语句

包含一个或一个以上的声明语句(如:进程赋值语句、任务启动、条件语句、case 语句和循环),在仿真运行的全过程中,在定时控制下被反复执行。

语 法:

```
always  
    声明语句
```

在程序中所处位置:

```
module- <here> -endmodule
```

规 则:

在 always 块中被赋值的只能是寄存器类型的变量,如 reg, integer, real, time, realtime。每个 always 在仿真一开始便开始执行,在仿真的过程中不断地执行,当执行完 always 块中的最后一个语句后,继续从 always 的开头执行。

注 意:

如果 always 块中包含有一个以上的语句,则这些语句必须放在 begin_end 或 fork_join 块中。如果 always 中没有时间控制,将会无限循环。

可综合性问题:

always 声明语句是用于综合过程的最有用的 Verilog 声明语句之一,然而 always 语句经常是不可综合的。为了得到最好的综合结果,always 块的 Verilog 程序应严格按以下的模板来编写。

模板 1:

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

模板 2:

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

模板 3：

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

模板 4：

```
always @(posedge Clock or negedge Reset)
// Clock and Reset only
begin
  if (! Reset)                // 测试异步复位电平是否有效
    :
    // 异步动作
  else
    :
    // 同步动作
end
// 可产生触发器和组合逻辑
```

举例说明：

(1) 寄存器级 always 实例：

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

(2) 描述组合逻辑电路的 always 块实例：

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

end

参阅 Begin, Fork, Initial, Statement 和 Timing Control 声明语句的说明。

2. assign 连续赋值声明语句

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

语 法 :

```
{either}
assign [ Strength] [ Delay] NetLValue = Expression,
NetLValue = Expression,
...
NetType [ Expansion] [ Strength] [ Range] [ Delay]
NetName = Expression,
NetName = Expression,
...
...; {See Net}
```

```
NetLValue = {either}
NetName
NetName[ ConstantExpression]
NetName[ ConstantExpression: ConstantExpression]
{ NetLValue,...}
```

在程序中所处位置 :

```
module-<here>- endmodule
```

规 则 :

两种形式的连续赋值语句效果相同。

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

注 意 :

连续赋值并不等同于进程连续赋值语句, 虽然它们相似。确保把 assign 放在正确的地方。连续赋值语句必须放在任何 initial 和 always 块外; 进程连续赋值语句可放在该语句被允许放的位置执行(在 initial, always, task, function 等块内部)。

可综合性问题 :

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

(2) 连续赋值语句将被综合成为组合逻辑电路。

提 示 :

用连续赋值语句去描述那些用简洁的表达式就能够很容易表达的组合逻辑电路。函数能够用来构建表示式。在描述较复杂的组合逻辑电路方面, 用 always 块比用许多句分开的连续赋值语句更好, 而且仿真的速度更快一些。当 Verilog 需要电路连线上(即 Net 上)。例如, 把一个 initial 块中产生的测试激励信号加

到一个实例模块的输入输出端口。

举例说明：

```
wire cout, cin;
wire [31:0] sum, a, b;
assign {cout, sum} = a + b + cin;

wire enable;
reg [7:0] data;
wire [7:0] #(3,4) f = enable ? data : 8'bz;
```

参阅 Net,Force 进程连续赋值语句的说明。

3. begin 声明语句

用于把多个声明语句组合起来成为一个语句,而其中每个声明语句是按顺序执行。Verilog 语法经常严格要求只有一个声明语句,例如 always 就是这样。如果 always 需要多个声明语句,那么这些声明语句必须被包含在一个 begin-end 块中。

语 法：

```
begin [: Label]
  [ Declarations... ]
  Statements...
end
```

Declaration = {either} Register Parameter Event

在程序中所处位置：

参阅 Statement 声明语句的说明。

规 则：

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

注 意：

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

提 示：尽管从技术上讲,将 begin-end 块用作局部声明是完全可行的,但建议不要这样做。甚至在并不需要局部声明,也不想禁止 begin-end 块时,也可以对该 begin-end 块加标识命名,以提高其可读性。给不用在别处的寄存器作局部声明,能使声明的意图变得更清楚。

举例说明：在本例中将通过一个简单的例子来说明如何使用 begin-end 块。

```

initial
begin : GenerateInputs
    integer I;
    for (I = 0; I < 8; I = I + 1)
        # Period {A, B, C} = I;
end

initial
begin
    Load = 0;           // Time 0
    Enable = 0;
    Reset = 0;
    #10 Reset = 1;      // Time 10
    #25 Enable = 1;     // Time 35
    #100 Load = 1;      // Time 135
end

```

参阅 Fork, Disable, Statement 声明语句的说明。

4. case 声明语句

如果 case 控制表达式与标号分支表达式相等, 则执行该分支的声明语句。

语 法:

```

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

```

CaseKeyword = {either} case casex casez

在程序中所处位置:

参阅 Statement 声明语句的说明。

规 则:

(1) 不确定值(Xs)和高阻值(Zs)在 caseX 声明语句中, 以及(Zs)在 casez 声明语句的表达式匹配中都意味着“不必考虑”。

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

(3) 如果某标号用逗号隔开两个或两个以上表达式时, 只要其中任何一个表达式与 case 表达式的值相等, 就可执行该标号的操作。

(4) 如果没有一个标号表达式与 case 表达式的值相等, 又没有 default 声明语句, 该 case

声明语句没有任何作用。

注意：

- (1) 如果在标号分支中有一个以上的声明语句,这些声明语句必须放在一个 begin-end 或 fork-join 块中。
- (2) 只有第一个与 case 表达式的值相等的标号分支才被执行。case 语句的标号并不一定是互斥的,所以当错误地重复使用相同的标号时,Verilog 编译器不会提示出错。
- (3) casex 或 casez 声明语句的语法是用保留字 endcase 作为结束,而不是用 endcasex 或 endcasez 来结束。
- (4) 在 Casex 声明语句的表达式中,x(不定值)或 z(高阻值)可以和任何值相等,casez 中的 z 也是如此。这有可能会给仿真结果带来混乱。

可综合性问题：

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

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

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

提示：

- (1) 为了使仿真能顺利进行,常常用 default 作为 case 声明的最后一个分支,以控制无法与标号匹配的 Case 变量。
- (2) 通常情况下用 casez 比用 casex 更好一些,因为 x 的存在可能会导致仿真并出现令人误解和混乱的结果。
- (3) 在 casex 和 casez 声明的标号中用“?”来代替“z”比较好,因为这样做比较清楚,是一个无关项,而不是一个高阻项。

举例说明：

```
case (Address)
  0 : A <= 1;           // Select a single Address value
  1 : begin               // Execute more than one statement
    A <= 1;
    B <= 1;
  end
  2, 3, 4 : C <= 1;     // Pick out several Address values
  default :              // Mop up the rest
    $display("Illegal Address value %h in %m at %t", Address, $realtime);
endcase

caseX (Instruction)
  8'b000xxxxx : Valid <= 1;
  8'b1xxxxxxx : Neg <= 1;
  default
    begin
```

```
    Valid <= 0;
    Neg <= 0;
```

```
end
endcase
```

```
casez ({A, B, C, D, E[3:0]})
  8'b1?????? : Op <= 2'b00;
  8'b010????? : Op <= 2'b01;
  8'b001??? 00 : Op <= 2'b10;
  default : Op <= 2'bxx;
endcase
```

参阅 IF 条件声明语句的说明。

5. comment 注释语句

注释语句应该位于 Verilog 源代码文件中。

语 法：

单行注释

//

多行注释

/ * ... * /

在程序中所处位置：

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

规 则：

单行注释以两个斜杠符开始,结束于该行的末尾;多行注释以“/*”符开始,中间可能有多行,结束于“*/”符;多行注释不能嵌套,但是,在多行注释中可以有单行注释,在这里它没有别的特殊含义。

注 意：

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

提 示：

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

举例说明：

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

参阅 Coding Standard 编码标准的说明。

6. defparam 定义参数声明语句

编译时可重新定义参数值。如果是分层次命名的参数，可以在该设计层次内或外的任何地方重新定义参数。

语 法：

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

在程序中所处位置：

module-<HERE>-endmodule

可综合性问题：

一般情况下是不可综合的。

提 示：

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

举例说明：

```
'timescale 1ns / 1ps  
module LayoutDelays;  
    defparam Design.U1.T_f = 2.7;  
    defparam Design.U2.T_f = 3.1;  
    ...  
endmodule
```

```
module Design(...);  
    ...  
    and_gate U1(f, a, b);  
    and_gate U2(f, a, b);  
    ...  
endmodule
```

```
module and_gate(f, a, b);  
    output f;  
    input a, b;  
    parameter T_f = 2;  
    and #(T_f)(f,a,b);  
endmodule
```

参阅 Name, Instantiation, Parameter 声明语句的说明。

7. delay 时延

可以为 UDP 和门的实例指定时延,也可以为连续赋值语句和线路连接指定时延。时延是在网表中线路连接和元件传输时延的模型。

语 法:

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

在程序中所处位置:

参照连续赋值语句、实例引用、线路连接声明语句的说明。

规 则:

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

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

(3) 如果给出三个延迟,第三个延迟表示关闭延迟(转变到高阻),除了三态电路外,第三个延迟表示电荷衰减时间。

(4) 延迟到 X 表示最小的指定延迟。

(5) 对于矢量,从非零到零的转变被看作下降,转变到高阻被看作关闭,其余的变化被看作是上升。

注 意:

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

可综合性问题:

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

提 示:

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

参阅线路连接、实例引用、连续赋值, Specify、定时控制等声明语句的说明。

8. disable 禁止

在运行激活的任务或命名的块时, Disable 能使在所在块执行完以前, 终止该块的执行。

语 法:

```
disable BlockOrTaskName;
```

在程序中所处位置：

参照 Statement 语句的说明。

规 则：

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

(2) 命名块或任务可以通过其内部的禁止声明语句实现自我禁止。

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

(4) 函数不能被禁止。

注 意：

如果一个任务被自我禁止,这跟任务返回不一样,因为输出未定义。

可综合性问题：

只有当命名块或任务自我禁止时,禁止才是可综合的,一般情况下是不可综合的。

提 示：

用禁止作为一种及早跳出任务的方法,用来跳出循环或继续下一步循环。

举例说明：

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

9. errors 错误

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

最容易出现的 5 个错误：

- (1) 进程赋值语句的左侧变量没有声明时为寄存器类型。
- (2) begin-end 声明语句忘了匹配。
- (3) 写二进制数时忘了标明数基(即 'b)。这样,在编译时会把它们当作十进制数来处理。
- (4) 编译引导语句用了错误的撇号,应该用向后的撇号也就是用表示重音的撇号;而表示数基的撇号,应该是普通的撇号,也就是反向的逗号。
- (5) 在声明语句的末尾忘了写上分号。

其他常见的错误：

- (1) 在定义任务或函数时,试图在任务或函数名后用括号来定义变量。
- (2) 在调试时,忘了在测试文件中引用实例模块。
- (3) 使用进程连续赋值语句而没有使用连续赋值语句(即赋值语句用错了地方)。
- (4) 把保留字作为标识符(例如用 xor 做标识符)。
- (5) Always 块忘了声明定时控制(导致无休止的循环)。
- (6) 在事件控制列表中错误地使用了逻辑或操作符(即 ||),而没有使用或保留字 or,例如把 always @ (a or b) 写成了 always @ (a || b)。
- (7) 用默认定义的 wire 类型变量来做矢量端口的连线。
- (8) 模块实例引用时端口的连接顺序搞错。
- (9) 在嵌套的 if-else 语句中,begin-end 配套错误。
- (10) 错误地使用等号:“=”用于赋值,“==”用于作数值比较,“====”用于需要对 0,1,X,Z 这 4 种逻辑状态作准确比较的场合。

10. event 事件

在行为模型中 Events 可以用来描述通信和同步。

语 法：

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

在程序中所处位置：

请参阅为 -> 所做的声明语句的说明。

事件声明语句可以放在下面的地方：

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

规 则：

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

可综合性问题:通常是不可综合的。

提 示:

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

举例说明：

```
event StartClock, StopClock;
always
```

```

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

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

```

参阅 Timing Control 定时控制语句的说明。

11. Expression 表达式

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

语 法：

Expression = {either}	表达式 = {以下任取其一}
Primary	基本表达式
Operator Primary {unary operator}	运算符 基本表达式 {单目运算符}
Expression Operator Expression {binary operator}	表达式 运算符 表达式 {双目运算符}
Expression ? Expression : Expression	表达式? 表达式: 表达式
String	字符串
Primary = {either}	基本表达式 = {以下任取其一}
Number	数字
Name {of parameter, net, or register}	变量名{参数, 网络, 或者寄存器的}
Name[Expression] {bit select}	变量名[表达式] {位选择}
Name[Expression: Expression] {part select}	变量名[表达式: 表达式] {部分位选择}

MemoryName[Expression]	存储器名[表达式]
{ Expression,... } { concatenation}	{表达式,...} {位拼接}
{ Expression{ Expression,... } } { replication}	{表达式{表达式,...}} {复制}
FunctionCall	函数调用
(MinTypMaxExpression)	(MinTypMax 表达式)
{MinTypMax expressions are used for delays}	{MinTypMax 表达式用于延迟}
MinTypMaxExpression = {either}	MinTypMax 表达式 = {任取其一}
Expression	表达式
Expression: Expression: Expression	表达式:表达式:表达式

规则：

- (1) 只有矢量类型的 Net 变量和寄存器、整数及时间类型变量才允许选取某位及某些位。
- (2) 某些位的选取必须将高位列在冒号的左侧,低位列在右侧。其中最高位是在 Net 变量或寄存器类型声明中位于冒号左边的数值。
- (3) 某位或某些位的选取若其中包含 X 或 Z 的位,或超出位的定义范围,在编译时可能会被认定为是错误的。如果不被认定是错的,编译器会给出一个值为 X 的表达式。
- (4) 没有为存储器建立某位或某些位选取的机制。

(5) 当整型常量在表达式中作为操作数时,未标明进制的有符号数(例如 -5)与标明进制的有符号数(例如 -'d5)是有所区别的。前者被视作一个有符号数,而后者被视作一个无符号数。

注意:

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

举例说明:

```
A + B
! A
(A && B) || C
A[7:0]
B[1]
-4'd12/3           // 是一个很大的正数
"Hello" != "Goodbye" // 此表达式为真(1)
$realtobits@();
{A, B, C[1:6]}    // 位拼接(8 位)
1:2:3              // 最小—典型—最大表达式
```

参阅延迟、函数调用、变量名、数字和操作数语句的说明。

12. for 循环声明语句

一般用途的循环语句。允许一条或更多的语句能被重复地执行。

语法:

```
for (RegAssignment; ~ ~ ~ {initial assignment}
```

Expression;	{loop condition}
RegAssignment	{iteration assignment}
Statement	
RegAssignment = RegisterLValue = Expression	寄存器赋值 = 寄存器值 = 表达式
RegisterLValue = {either}	寄存器值 = {任取其一}
RegisterName	寄存器名
RegisterName[Expression]	寄存器名[表达式]
RegisterName[ConstantExpression; ConstantExpression]	寄存器名[常量表达式; 常量表达式]
Memory[Expression]	存储器[表达式]
{ RegisterLValue, ... }	{寄存器值,……}

在程序中所处位置：

参照 Statement 语句的说明。

规 则：

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

注 意：

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

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

在以上这些情况中,应将循环变量 i 定义为整型。

可综合性问题：

如果循环的边界是固定的,那么在综合时该循环语句被认为是重复的硬件结构。

举例说明：

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

参阅 Forever, Repeat, While 语句的说明。

13. force 强迫赋值

类似于进程连续赋值语句,可对 Net 变量和寄存器类型变量实行强制赋值,常用于调试。

语 法:

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

在程序中所处位置:

参阅声明语句的说明。

规 则:

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

(2) 当作用在某一寄存器上的 force 被释放时,寄存器并无必要立刻改变其值。如果此时没有进程连续赋值对这个寄存器赋值,则强制赋入的值会一直保留到下一个进程赋值语句的执行。

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

可综合性问题:

强迫赋值语句是不可综合的。

提 示:

强迫赋值常用于测试文件的编写,调试时常需要强制对某些变量赋值,不能用于模块的行为建模(此时应使用连续赋值语句)。

举例说明:

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

参照进程连续赋值语句的说明。

14. forever 声明语句

使一个或一个以上语句无限循环地执行。

语 法：

forever Statement

在程序中所处位置：

参阅 Statement 语句的说明。

注意：

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

可综合性问题：

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

提 示：

forever 在测试模块中描述时钟时很有用,常用 disable 来跳出循环。

举例说明：

```
initial
begin : Clocking
    Clock = 0;
    forever
        # 10 Clock = ! Clock;
    end
```

```
initial
begin : Stimulus
    :
    disable Clocking; // 停止时钟
end
```

参阅 For, Repeat, While 和 Disable 语句的说明。

15. fork 声明语句

可将多个语句集合在一个块中,以使它们能被并发地执行。

语 法：

fork [: Label

[Declarations...]]

Statements...

join

Declaration = {either}

Register

Parameter

Event

fork[,块名

[块内声明语句.....]]

语句.....

join

块内声明语句 = {任选其一}

寄存器变量

参 数

事 件

在程序中所处位置：

参照 Statement 语句的说明。

规 则：

fork-join 块必须至少包括一条语句。Fork-join 块里的语句是并发执行的，因此 Fork-join 块内语句的顺序是任意的。时间控制是相对于块的开始时刻的。当 fork-join 块里所有的语句执行完毕后，块也就执行完毕了。Begin-end 和 fork-join 块可以自身嵌套或互相嵌套。如果想在某 fork-join 块内包含块内局部声明语句如果想要禁止某 fork-join 块的运行，则该块必须已命名。那么必须对该块命名(即该块必须有一个标识符号)。

可综合性问题：

fork 语句不可综合。

注 意：

Fork-join 语句在描述并发形式的行为时很有用。

举例说明：

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

参阅 Begin, Disable, Statement 语句的说明。

16. function 函数

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

语 法：

function [RangeOrType] FunctionName;	function [返回值的类型或范围] 函数名；
Declarations...	端口声明.....
Statement	语 句
endfunction	endfunction
RangeOrType = {either}	返回值的类型或范围 = {任取其一}
integer	整 数
real	实 数
time	时 间
realtime	
Range	
Range = [ConstantExpression: ConstantExpression]	范围 = [常量表达式:常量表达式]
Declaration = {either}	端口声明 = {任取其一}
input [Range] Name, ...;	input [范围] 变量名,;
Register	
Parameter	
Event	

在程序中所处位置：

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

规 则：

- (1) 函数必须至少含有一个输入变量,不能有任何输出或输入/输出双向变量。
- (2) 函数不能包含时间控制语句(如延迟#、事件控制@或等待 wait)。
- (3) 函数是通过对函数名赋值的途径返回其值的,就好比是一个寄存器。
- (4) 函数不能启动任务。
- (5) 函数不能被禁用。

注 意：

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

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

可综合性问题：

函数的每一次调用都被综合为一个独立的组合逻辑电路块。

举例说明：

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

参阅 Function Call 和 Task 语句的说明。

17. function Call 函数调用

函数的调用可返回一个供表达式使用的值。

语 法：

Function Name (Expression, …); 函数名(表达式,……);

在程序中所处位置：

参阅 Expression 语句的说明。

规 则：

函数必须至少含有一个输入变量,所以函数调用时总是至少含有一个表达式。

可综合性问题：

函数的每一次调用在综合后都会生成一个独立的组合逻辑电路块。

举例说明：

Byte = ReverseBits (Byte);

参阅 Function, Expression 和 Task Enable 语句的说明。

18. gate 门

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

逻辑门:

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

缓冲器与非门:

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

三态门:

```
bufif0 (Output, Input, Enable)  
bufif1 (Output, Input, Enable)  
notif0 (Output, Input, Enable)  
notif1 (Output, Input, Enable)
```

MOS 开关:

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

CMOS 开关:

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

双向开关:

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

双向可控开关:

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

上拉源与下拉源:

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

真值表：

附表 1 至附表 10 为各种逻辑函数真值表，逻辑值 L 与 H 代表部分未知值。L 表示 0 或 Z，H 表示 1 或 Z。

附表 1 与门真值表

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

附表 2 与非门真值表

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

附表 3 或门真值表

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

附表 4 或非门真值表

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

附表 5 xor 门真值表

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

附表 6 xnor 真值表

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

附表 7 缓冲器及与非门真值表

buf		Not	
Input	Output	Input	Output
0	0	0	1
1	1	1	0
X	X	X	X
Z	Z	Z	X

缓冲门、非门都可以有多个输出,但这些输出值都是相同的。

附表 8 缓冲器使能端真值表

Bufif0		Enable				Bufif1		Enable			
		0	1	X	Z			0	1	X	Z
D A T A	0	0	Z	L	L	D A T A	0	Z	0	L	L
	1	1	Z	H	H		1	Z	1	H	H
	X	X	Z	X	X		X	Z	X	X	X
	Z	X	Z	X	X		Z	Z	X	X	X

附表 9 缓冲器非门使能端真值表

notif0		Enable				notif1		Enable			
		0	1	X	Z			0	1	X	Z
D A T A	0	1	Z	H	H	D A T A	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

附表 10 MOS型控制端真值表

PMOS RPMOS		Control				NMOS RN莫斯		Control			
		0	1	X	Z			0	1	X	Z
D A T A	0	0	Z	L	L	D A T A	0	Z	0	L	L
	1	1	Z	H	H		1	Z	1	H	H
	X	X	Z	X	X		X	Z	X	X	X
	Z	Z	Z	Z	Z		Z	Z	Z	Z	Z

cmos (W, Datain, NControl, PControl);

等价于：

nmos (W, Datain, NControl);

pmos (W, Datain, PControl);

规则：

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

(2) 当有电阻的开关,如 rnmos, rpmos, rcmos, rtran, rtranif0 和 rtranif1 类型的开关,开启时,信号从输入传到输出会按附表 11 减小其强度。

参阅 UDP(用户自定义原语)和 Instantiation(实例引用)语句的说明。

19. if 条件声明语句

根据条件表达式的逻辑值(真/假),执行两条/块语句中的一条/块。

语法：

if (Expression)	if(表达式)
Statement	语句
[else	[else
Statement]	语句]

附表 11 强度减弱

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

在程序中所处位置：

参阅 Statement 语句的说明。

规 则：

当表达式的值为非零时被认为是真,当值为零、X 或 Z 时被认为是假。

注 意：

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

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

可综合性问题：

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

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

提 示：

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

举例说明：

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

参阅 Case 和 Operators 语句的说明。

20. initial 声明语句

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

语 法：

initial

Statement

在程序中所处位置：

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

可综合性问题：

initial 语句是不可综合的。

注意：

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

提示：

在仿真测试文件中可使用 initial 语句来描述激励。

举例说明：

下面的例子给出如何使用 initial 在测试文件中产生矢量：

```
reg Clock, Enable, Load, Reset;
reg [7:0] Data;
parameter HalfPeriod = 5;
initial
begin : ClockGenerator
    Clock = 0;
    forever
        #(HalfPeriod) Clock = ! Clock;
end
initial
begin
    Load = 0;
    Enable = 0;
    Reset = 0;
    #20      Reset = 1;
    #100     Enable = 1;
    #100     Data = 8'haa;
    Load = 1;
    #10 Load = 0;
    #500     disable ClockGenerator; //停止时钟的产生
end
```

参阅 Always 语句的说明。

21. instantiation 实例引用

实例(instance)是模块,是 UDP 或门的唯一复制。通过实例的引用可以生成设计的各个层次。设计的行为也能通过引用 UDP、门和其他的模块的实例,并用电路连线(Net)将它们连接起来,从结构上加以描述。

语法：

{either}	{任取其一}
ModuleName	
[#(Expression, …)] ModuleInstance, … ;	模块名 [#(表达式,……)]实例模块,……;
UDPOrGateName [Strength]	

[Delay] PrimitiveInstance, …;	UDP 或门名 [强度] [延迟] 原始实例, ……;
ModuleInstance =	实例模块 =
InstanceName [Range] ([PortConnections])	实例名 [范围] ([端口连线])
PrimitiveInstance =	原始实例 =
[InstanceName [Range]] (Expression, …)	[实例名[范围]](表达式, ……)
Range =	范围 = [常量表达式:常量表达式]
[ConstantExpression: ConstantExpression]	端口连线 = {任取其一}
PortConnections = {either}	[表达式], …… {有顺序的连线}
[Expression], … {ordered connection}	• PortName([Expression]), … {named connection} • 端口名([表达式]), …… {指定连线}
• PortName([Expression]), … {named connection}	

在程序中所处位置：

module -<HERE>- endmodule

规 则：

(1) 命名的端口连线只能用于模块实例。

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

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

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

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

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

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

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

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

注 意：

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

(2) 多个模块、UDP 或门的成组实例引用的语法是最近才加入 Verilog 语言的标准中，目

前还没有工具支持这语法。

可综合性问题：

UDP 和开关的实例引用一般是不能综合的。

提示：

使用命名的端口连接方式以提高程序的可读性并减少发生错误的可能性(见前文)。

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

举例说明：

UDP 实例引用 1:Nand2 (weak1,pull0) #(3,4) (F, A, B);

模块实例引用 2:Counter U123 (.Clock(Clk), .Reset(Rst), .Count(Q));

在下面的两个例子中 QB 端口没有连接：

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

DFF Ff2 (Q, , Clk, D);

下面是端口连线表中使用门表达式的例子：

nor (F, A&~B, C) // 不要这样使用

下面是一个多实例模块引用的例子：

```
module Tristate8 (out, in, ena);
    output [7:0] out;
    input [7:0] in;
    input ena;

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

参阅 Module, User Defined Primitive, Gate 和 Port 语句的说明。

22. module 模块定义

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

语 法：

{either}	{任取其一}
module ModuleName [(Port,...)];	module 模块名[(端口,……)];
ModuleItems...	模块条款
endmodule	endmodule
macromodule ModuleName [(Port,...)];	macromodule 模块名[(端口,……)];
ModuleItems...	模块条款
endmodule	endmodule
ModuleItem = {either}	模块条款={任取其一}
Declaration	声 明
Defparam	参数定义
ContinuousAssignment	连续任务
Instance	实例引用
Specify	详细说明块
Initial	初始化块
Always	总是执行块
Declaration = {either}	声明={任取其一}
Port	端 口
Net	网 络
Register	寄存器
Parameter	参 量
Event	事 件
Task	任 务
Function	函 数

在程序中所处位置：

在其他模块或 UDP 除外。

规 则：

- (1) 几个模块或几个 UDP(或它们的混合)可以在一个文件中进行描述。事实上,一个模块也可以分开,并在两个或更多的文件中描述,但不推荐这种做法。
- (2) 模块也可使用关键字 `macromodule` 来定义。其语法与用关键字 `module` 来定义模块是完全一样的。
- (3) Verilog 编译器在编译宏模块时与编译一般模块时有所不同,比如不必为宏模块实例创建层次。这样,从仿真速度或存储介质的开销两方面来说,宏模块的编译更有效率。为了达到这个目的,宏模块的编译可能受实现时的某些特殊条件的限制。如果遇到这种情况,宏模块将被按一般模块编译。

注 意：

模块与宏模块都以关键字 `endmodule` 作为结束标志。

可综合性问题：

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

(2) 不是所有的工具都支持宏模块的综合。

提示：

尽量使每一个文件只包含一个模块。在大型设计中，这样做易于源代码的维护。

举例说明：

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

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

参阅 User Defined Primitive, Instantiation 和 Name 语句的说明。

23. name 名字

任何用 Verilog 语言描述的“东西”都通过它的名字来识别。

语法：

Identifier

\EscapedIdentifier {terminates with white space} \ 扩展标识符{空格表示结束}

规 则：

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

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

(3) 在 Verilog 中变量名区别大小写，对大小写敏感。

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

24. Hierarchical Names 分级名字

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

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

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

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

25. Upwards Name Referencing 向上索引名

包含两个标识符中间用点号隔开的分级名可能是下列情况中的一种：

(1) 当前模块所引用的实例模块中的一项(这是向下引用)。

(2) 顶层模块中的一项(这是一个分级名字)。

(3) 当前模块的父模块中的引用的实例模块中的一项(这是向上引用)。

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

可综合性问题：

分级名字和向上索引名在一般低档综合工具上是不可综合的。

提 示：

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

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

(3) 分级名字仅用于测试模块或那些无法改用别的合适的名字的高层系统模型中。

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

举例说明：

以下是合法名的例子：

```
A_99_Z
Reset
_54_MHz_Clock $
Module           // 与 "module" 是不一样的
\$%^&*()        // 扩展标识符
```

以下是因上述原因而不合法的名字：

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

下面的例子说明了分级名字和向上引用名字：

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

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

Module Bottom;
    reg R;                  // Top. U1. R

    task T;                // Top. U1. T
        reg R;              // Top. U1. T. R;
        :
    endtask

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

26. Net 线路连接

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

语 法：

```

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

```

在程序中所处位置：

```
module—<HERE>—endmodule
```

规 则：

- (1) supply0 和 supply1 类型的 net 变量分别具有逻辑值 0 和 1，并可以为它定义驱动能力(Supply strength)。
- (2) tri0 和 tri1 类型的 nets，当没有驱动时，分别具有逻辑值 0 和 1，并可以为它定义驱动能力(Pull strength)。
- (3) 如果 net 变量的扩展(Expansion)选项选用了关键词 vectored，则不允许对它进行某位和某些位的选择，也不允许对它定义强度，PLI 会认为该 net 变量是不可扩展的；如果扩展选项选用了关键词 scalared，则允许对它进行某位和某些位的选择，也允许对它定义强度，PLI 将会认为该 net 变量是可扩展的，这些关键词是有参考价值的。
- (4) 除了结构描述中的端口和标量连线不用声明其 net 类型外，其他类型的 net 变量在应用之前必须声明。

Truth Table 真值表

当 Net 具有两个或两个以上驱动时，同时假定其驱动器强度值均相等，附表 12 为 Wire tri, Wand triand, Wor trior, tri0 和 tri1 的真值表则显示输出的结果。如果驱动器强度不相等，则驱动强度大者，驱动该 Net 变量。

注 意：

- (1) 当 net 变量未被驱动时，对 tri0 或 tri1 类型的 net 变量的连续赋值不影响其值和强