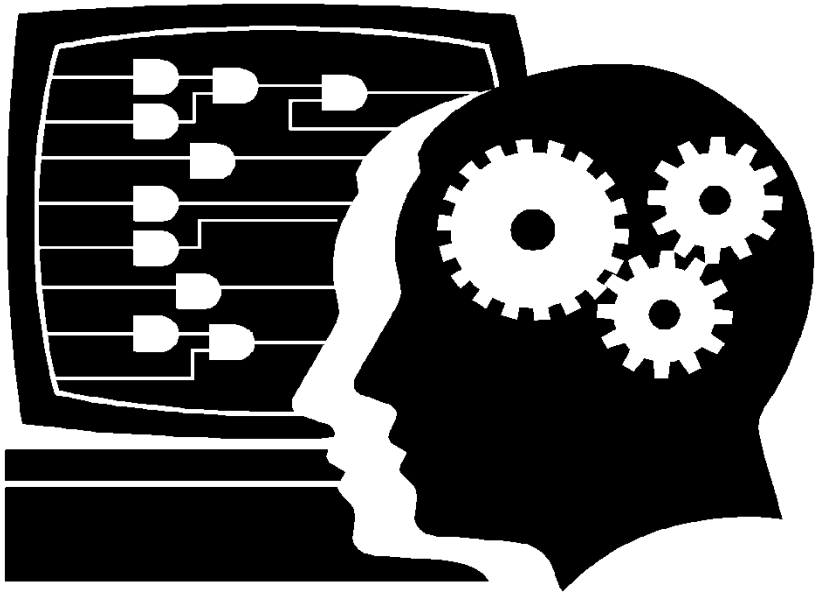
Verilog设计进阶



主要内容

- 过程语句（initial、always）
- 块语句（begin-end、fork-join）
- 赋值语句（assign、=、<=）
- 条件语句（if-else、case、casez、casex）
- 循环语句（for、forever、repeat、while）
- 编译指示语句（`define、`include、`ifdef、`else、`endif）
- 任务（task）与 函数（function）
- 顺序执行与并发执行

1 过程语句

- **initial**
- **always**
- 在一个模块（**module**）中，使用**initial**和**always**语句的次数是不受限制的。**initial**语句常用于仿真中的初始化，**initial**过程块中的语句仅执行一次；**always**块内的语句则是不断重复执行的。

always过程语句使用模板

- always @(<敏感信号表达式event-expression>)

```
begin
//过程赋值
//if-else，case，casex，casez选择语句
//while，repeat，for循环
//task，function调用
end
```

- “always”过程语句通常是带有触发条件的，触发条件写在敏感信号表达式中，只有当触发条件满足时，其后的“begin-end”块语句才能被执行。

敏感信号表达式“event-expression”

- 敏感信号表达式又称事件表达式或敏感信号列表，即当该表达式中变量的值改变时，就会引发块内语句的执行。因此敏感信号表达式中应列出影响块内取值的所有信号。若有两个或两个以上信号时，它们之间用“or”连接。
- 例如：

```
@(a)                                    //当信号a的值发生改变
@(a or b)                               //当信号a或信号b的值发生改变
@(posedge clock)                        //当clock 的上升沿到来时
@(negedge clock)                        //当clock 的下降沿到来时
@(posedge clk or negedge reset)
                    //当clk的上升沿到来或reset信号的下降沿到来
```

敏感信号列表举例（4选1数据选择器）

```
module  mux4_1(out,in0,in1,in2,in3,sel);
output  out;
input  in0,in1,in2,in3;
input[1:0]  sel;
reg  out;
always @(in0 or in1 or in2 or in3 or sel)
        //敏感信号列表
case(sel)
  2'b00:    out=in0;
  2'b01:    out=in1;
  2'b10:    out=in2;
  2'b11:    out=in3;
  default: out=2'bx;
endcase
endmodule
```

posedge和negedge关键字

- 对于时序电路，事件通常是由时钟边沿触发的，为表达边沿这个概念，Verilog提供了posedge和negedge关键字来描述。比如：
- 【例】同步置数、同步清零的计数器

```
module count(out,data,load,reset,clk);
output[7:0] out;
input[7:0] data;
input load,clk,reset;
reg[7:0] out;
always @(posedge clk)                    //clk上升沿触发
   begin
   if(!reset)       out=8'h00;       //同步清0，低电平有效
   else  if(load)   out=data;        //同步预置
   else             out=out+1;       //计数
   end
endmodule
```

2 块语句

- 块语句是由块标志符**begin-end**或**fork-join**界定的一组语句，当块语句只包含一条语句时，块标志符可以缺省。
- **begin-end**串行块中的语句按串行方式顺序执行。
- 比如：

```
begin
regb=rega;
regc=regb;
end
```

- 由于**begin-end**块内的语句顺序执行，在最后，将**regb**、**regc** 的值都更新为**rega**的值，该**begin-end**块执行完后，**regb**、**regc** 的值是相同的。

3 赋值语句

■**1、持续赋值语句（Continuous Assignments）**

assign为持续赋值语句，主要用于对wire型变量的赋值。

比如：assign c=a&b;

在上面的赋值中，a、b、c三个变量皆为wire型变量，a和b信号的任何变化，都将随时反映到c上来。

2、过程赋值语句（Procedural Assignments）

过程赋值语句多用于对reg型变量进行赋值。过程赋值有阻塞（blocking）赋值和非阻塞（non_blocking）赋值两种方式。

（1）非阻塞（non_blocking）赋值方式

赋值符号为“<=”， 如：b<= a;

非阻塞赋值在整个过程块结束时才完成赋值操作，即b的值并不是立刻就改变的。

（2）阻塞（blocking）赋值方式

赋值符号为“=”， 如：b= a;

阻塞赋值在该语句结束时就立即完成赋值操作，即b的值在该条语句结束后立刻改变。如果在一个块语句中，有多条阻塞赋值语句，那么在前面的赋值语句没有完成之前，后面的语句就不能被执行，仿佛被阻塞了（blocking）一样，因此称为阻塞赋值方式。

阻塞赋值与非阻塞赋值

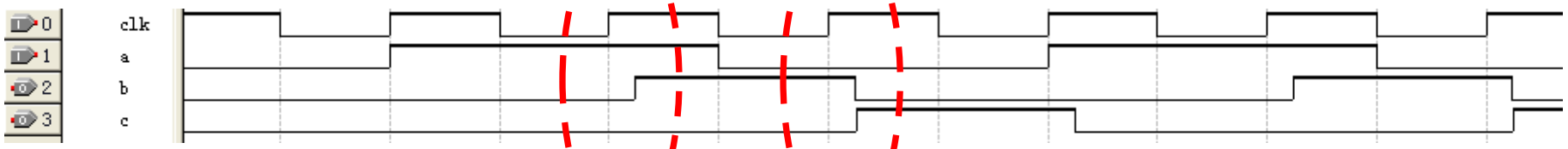
非阻塞赋值

```
module  non_block(c,b,a,clk);
output  c,b;
input  clk,a;
reg  c,b;
always @(posedge clk)
     begin
     b<=a;
     c<=b;
     end
endmodule
```

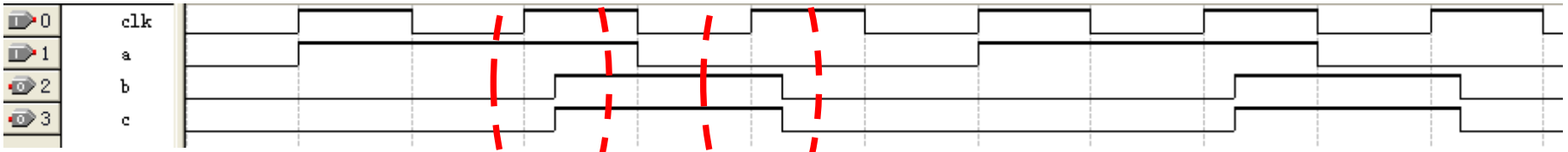
阻塞赋值

```
module  block(c,b,a,clk);
output  c,b;
input  clk,a;
reg  c,b;
always @(posedge clk)
     begin
     b=a;
     c=b;
     end
endmodule
```

阻塞赋值与非阻塞赋值

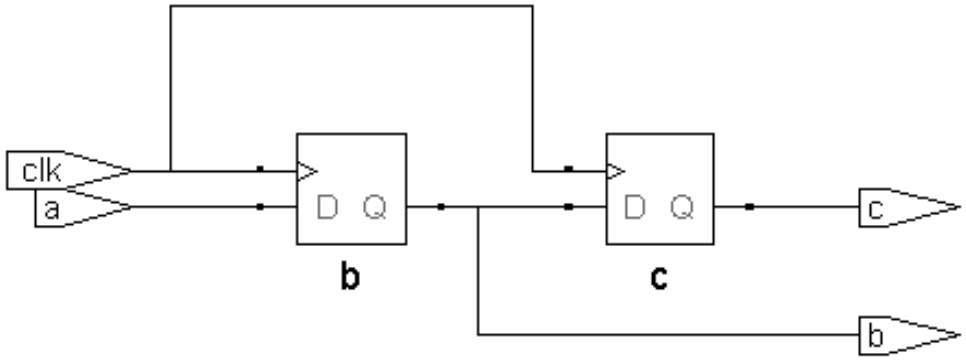


非阻塞赋值仿真波形图

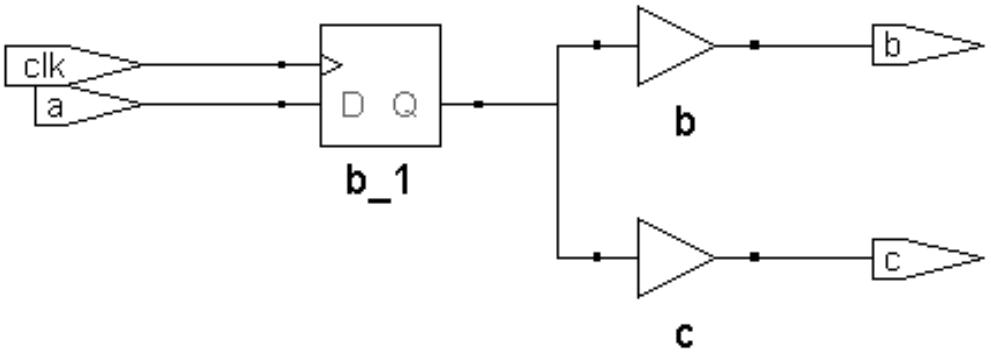


阻塞赋值仿真波形图

阻塞赋值与非阻塞赋值



非阻塞赋值综合结果



阻塞赋值综合结果

4 条件语句

- *（if-else语句）*
- **if-else语句使用方法有以下3种：**

（1）if（表达式） 语句1；

（2）if（表达式） 语句1；

else 语句2；

（3）if（表达式1） 语句1；

else if（表达式2） 语句2；

else if（表达式3） 语句3；

……

else if（表达式n） 语句n；

else 语句n+1;

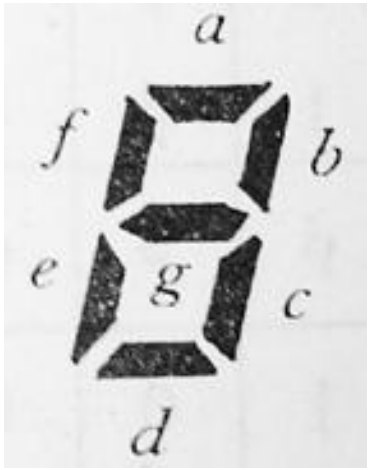
case语句

- case语句的使用格式如下。

```
case （敏感表达式）
        值1：  语句1；                    //case分支项
        值2：  语句2；
        ……
        值n：  语句n;
        default：语句n+1；
endcase
```

BCD码—七段数码管显示译码器

```
module  decode4_7(decodeout,indec);
output[6:0]  decodeout;
input[3:0]  indec;
reg[6:0]  decodeout;
always @(indec)
  begin
   case(indec)                                    //用case语句进行译码
     4'd0:decodeout=7'b1111110;
     4'd1:decodeout=7'b0110000;
     4'd2:decodeout=7'b1101101;
     4'd3:decodeout=7'b1111001;
     4'd4:decodeout=7'b0110011;
     4'd5:decodeout=7'b1011011;
     4'd6:decodeout=7'b1011111;
     4'd7:decodeout=7'b1110000;
     4'd8:decodeout=7'b1111111;
     4'd9:decodeout=7'b1111011;
     default: decodeout=7'bx;
   endcase
  end
endmodule
```



5 循环语句

在Verilog中存在四种类型的循环语句，用来控制语句的执行次数。这四种语句分别为:

（1）forever：连续地执行语句；多用在“initial”块中，以生成时钟等周期性波形。

（2）repeat：连续执行一条语句n次。

（3）while：执行一条语句直到某个条件不满足。

（4）for：有条件的循环语句。

- for loop

```
initial
  begin
     for(i=0;i<4;i=i+1)
        out = out +1;
 end
```

- repeat loop

```
initial
  begin
     repeat(5)
        out = out +1;
  end
```

- while loop

```
initial
  begin
     i=0;
     while(i<0)
        i=i+1;
  end
```

for语句

- for语句的使用格式如下（同C语言）：

for（表达式1；表达式2；表达式3）语句；

即：for（循环变量赋初值；循环结束条件；循环变量增值）执行语句；

用for语句描述的七人投票表决器

```
module  voter7(pass,vote);
output  pass;
input[6:0]  vote;
reg[2:0]  sum;
integer  i;
reg  pass;
always @(vote)
  begin
sum=0;
for(i=0;i<=6;i=i+1)          //for语句
if(vote[i]) sum=sum+1;
if(sum[2])  pass=1;          //若超过4人赞成，则pass=1
else          pass=0;
   end
endmodule
```

repeat语句

- repeat语句的使用格式为：

  repeat（循环次数表达式） 语句；

或 repeat（循环次数表达式） begin

……

end

用repeat实现8位二进制数的乘法

```
module mult_repeat(outcome, a, b);
parameter  size=8;
input[size:1]  a,b;
output[2*size:1]  outcome;
reg[2*size:1]  temp_a,outcome;
reg[size:1]  temp_b;
always @(a or b)
begin
outcome=0;
temp_a=a;  temp_b=b;
repeat(size)                    //repeat语句，size为循环次数
begin
if(temp_b[1])                   //如果temp_b的最低位为1，就执行下面的加法
outcome=outcome +temp_a;
temp_a=temp_a<<1;       //操作数a左移一位
temp_b=temp_b>>1;       //操作数b右移一位
end
end
endmodule
```

6 编译指示语句

- Verilog允许在程序中使用特殊的编译向导（Compiler Directives）语句，在编译时，通常先对这些向导语句进行“预处理”，然后再将预处理的结果和源程序一起进行编译。
- 向导语句以符号“`”开头，以区别于其它语句。Verilog提供了十几条编译向导语句，如：`define、`ifdef、`else、`endif、`restall等。比较常用的有`define，`include和`ifdef、`else、`endif等。

宏替换`define

- `define语句用于将一个简单的名字或标志符（或称为宏名）来代替一个复杂的名字或字符串，其使用格式为：

  `define 宏名（标志符） 字符串

  如：`define sum ina+inb+inc+ind

  在上面的语句中，用简单的宏名sum来代替了一个复杂的表达式ina+inb+inc+ind，采用了这样的定义形式后，在后面的程序中，就可以直接用sum来代表表达式ina+inb+inc+ind了。

文件包含`include

- `include是文件包含语句，它可将一个文件全部包含到另一个文件中。其格式为：

`include “文件名”

- 使用`include语句时应注意以下几点：

（1）一个`include语句只能指定一个被包含的文件。

（2）`include语句可以出现在源程序的任何地方。被包含的文件若与包含文件不在同一个子目录下，必须指明其路径名。

（3）文件包含允许多重包含，比如文件1包含文件2，文件2又包含文件3等。

7 任务与函数

- 任务（task）
- 任务定义格式：

  task <任务名>；　　//注意无端口列表

  端口及数据类型声明语句；

  其它语句；

  endtask
- 任务调用的格式为：

  <任务名>（端口1，端口2，……）；
- 需要注意的是：任务调用时和定义时的端口变量应是一一对应的。

使用任务时，需要注意以下几点：

- 任务的定义与调用须在一个**module**模块内。
- 定义任务时，没有端口名列表，但需要紧接着进行输入输出端口和数据类型的说明。
- 当任务被调用时，任务被激活。任务的调用与模块调用一样通过任务名调用实现，调用时，需列出端口名列表，端口名的排序和类型必须与任务定义中的相一致。
- 一个任务可以调用别的任务和函数，可以调用的任务和函数个数不限。

- 函数（function）

  函数的目的是返回一个值，以用于表达式的计算。

- 函数的定义格式：

  function <返回值位宽或类型说明> 函数名；

  端口声明；

  局部变量定义；

  其它语句；

  endfunction

- <返回值位宽或类型说明>是一个可选项，如果缺省，则返回值为1位寄存器类型的数据。

函数举例

```
function[7:0] get0;
input[7:0] x;
reg[7:0] count;
integer i;
    begin
    count=0;
    for (i=0;i<=7;i=i+1)
    if(x[i]=1'b0)  count=count+1;
    get0=count;
    end
endfunction
```

上面的get0函数循环核对输入数据x的每一位，计算出x中0的个数，并返回一个适当的值。

在使用函数时，需注意

- 函数的定义与调用须在一个**module**模块内。
- 函数只允许有输入变量且必须至少有一个输入变量，输出变量由函数名本身担任，在定义函数时，需对函数名说明其类型和位宽。
- 定义函数时，没有端口名列表，但调用函数时，需列出端口名列表，端口名的排序和类型必须与定义时的相一致。这一点与任务相同
- 函数可以出现在持续赋值**assign**的右端表达式中。
- 函数不能调用任务，而任务可以调用别的任务和函数，且调用任务和函数个数不受限制。

任务与函数的比较

比较项目	任务（task）	函数（function）
输入与输出	可有任意个各种类型的参数	至少有一个输入， 不能将 inout 类型作为输出
调用	任务只可在过程语句中调用， 不能在连续赋值语句 assign 中调用	函数可作为表达式中的一个操作数来调用，在过程赋值和连续赋值语句中均可以调用
定时事件控制（#，@和wait）	任务可以包含定时和事件控制语句	函数不能包含这些语句
调用其它任务和函数	任务可调用其它任务和函数	函数可调用其它函数，但不可以调用其它任务
返回值	任务不向表达式返回值	函数向调用它的表达式返回一个值

8 顺序执行与并发执行

- 两个或更多个“always”过程块、“assign”持续赋值语句、实例元件调用等操作都是同时执行的。
- 在“always”模块内部，其语句如果是非阻塞赋值，也是并发执行的；而如果是阻塞赋值，则语句是按照指定的顺序执行的，语句的书写顺序对程序的执行结果有着直接的影响。

顺序执行的例子

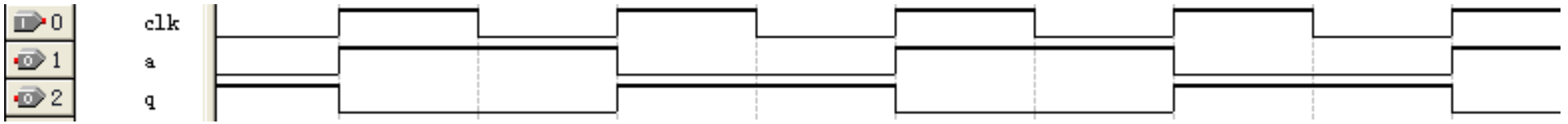
顺序执行模块1

```
module  serial1(q,a,clk);
output  q,a;
input  clk;
reg  q,a;
always @(posedge clk)
     begin
     q=~q;
     a=~q;
     end
endmodule
```

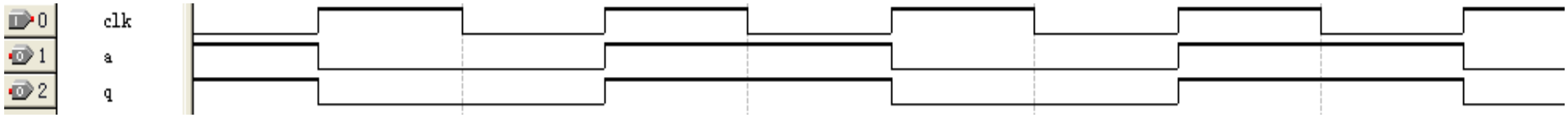
顺序执行模块2

```
module  serial2(q,a,clk);
output  q,a;
input  clk;
reg  q,a;
always@(posedge clk)
     begin
     a=~q;
     q=~q;
     end
endmodule
```

顺序执行



顺序执行模块1仿真波形图



顺序执行模块2仿真波形图