


# 目 录

第一章	绪论
第二章	系统级设计
第三章	Verilog HDL硬件描述语言
第四章	逻辑综合
第五章	可编程逻辑器件
第六章	物理版图设计基础
第七章	仿真验证
第八章	集成电路设计发展趋势



# 内 容

- 第一节 硬件描述语言
- 第二节 Verilog HDL设计入门
- 第三节 Verilog基础知识
- 第四节 行为描述
- 第五节 数据流描述
- 第六节 结构描述
- 第七节 用户自定义元件
- 第八节 其他论题
- 第九节 仿真
- 第十节 实例分析



## 第八节 其他论题

8.1 任务与函数

8.2 wait语句与命名事件

8.3 命名的块与disable

8.4 作用域规则和层次名

8.5 延时描述



## 第八节 其他论题

### 8.1 任务与函数

8.2 wait语句与命名事件

8.3 命名的块与disable

8.4 作用域规则和层次名

8.5 延时描述

# Verilog的任务及函数

结构化设计是将任务分解为较小的，更易管理的单元，并将可重用代码进行封装。这通过将设计分成模块，或任务和函数实现。

- 任务 (task)
  - ✓ 通常用于调试，或对硬件进行行为描述
  - ✓ 可以包含时序控制 (#延迟, @, wait)
  - ✓ 可以有 input, output, 和 inout 参数
  - ✓ 可以调用其他任务或函数
- 函数(function)
  - ✓ 通常用于计算，或描述组合逻辑
  - ✓ 不能包含任何延迟；函数仿真时间为0
  - ✓ 只含有input参数并由函数名返回一个结果
  - ✓ 可以调用其他函数，但不能调用任务

# Verilog的任务及函数

- 任务和函数必须在**module**内调用
- 在任务和函数中不能声明**wire**
- 所有输入/输出都是**局部寄存器**
- 任务/函数执行完成后才返回结果。

例如，若任务/函数中有**forever**语句，则永远不会返回结果

# 任务

## 主要特点：

- 任务可以有input,output 和 inout参数。
- 传送到任务的参数和与任务I/O说明顺序相同。尽管传送到任务的参数名称与任务内部I/O说明的名字可以相同，但在实际中这通常不是一个好的方法。参数名的唯一性可以使任务具有好的模块性。
- 可以在任务内使用时序控制。
- 在Verilog中任务定义一个新范围（scope）
- 要禁止任务，使用关键字disable。

从代码中多处调用任务时要小心。因为任务的局部变量只有一个拷贝，并行调用任务可能导致错误的结果。在任务中使用时序控制时这种情况时常发生。

在任务或函数中引用调用模块的变量时要小心。如果想使任务或函数能从另一个模块调用，则所有在任务或函数内部用到的变量都必须列在端口列表中。

# 任务

下面的任务中有输入，输出，时序控制，并且引用了一个module变量。但没有双向端口，也没有显示。

任务调用时的参数按任务定义的顺序列出。

```
module mult (clk, a, b, out, en_mult);  
    input clk, en_mult;  
    input [3: 0] a, b;  
    output [7: 0] out;  
    reg [7: 0] out;  
    always @(posedge clk)  
        multme (a, b, out); // 任务调用  
    task multme; // 任务定义  
        input [3: 0] xme, tome;  
        output [7: 0] result;  
        wait (en_mult)  
            result = xme * tome;  
    endtask  
endmodule
```



# 任务的定义与调用

```
module Has_Task;  
parameter MAXBITS = 8;  
  
task Reverse_Bits;                                //任务定义  
input [MAXBITS - 1:0] Din;  
output [MAXBITS - 1:0] Dout;  
integer K;  
begin  
  for (K = 0; K < MAXBITS; K = K + 1)  
    Dout [MAXBITS - K - 1] = Din [K];  
  end  
endtask  
  
reg [MAXBITS - 1:0] Reg_X, New_Reg; //任务调用  
Reverse_Bits(Reg_X, New_Reg);  
  
...  
endmodule
```

# 任务的例子

```
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; //采用阻塞赋值方式。
    x=y;
    y=tmp;
end
endtask
endmodule
```

# 函数 (function)

```
module orand (a, b, c, d, e, out);  
    input [7: 0] a, b, c, d, e;  
    output [7: 0] out;  
    reg [7: 0] out;  
    always @( a or b or c or d or e)  
        out = f_or_and (a, b, c, d, e); // 函数调用  
function [7:0] f_or_and;  
    input [7:0] a, b, c, d, e;  
    if (e == 1)  
        f_or_and = (a | b) & (c | d);  
    else  
        f_or_and = 0;  
    endfunction  
endmodule
```

函数中不能有时序控制，但调用它的过程可以有时序控制。

函数名 **f\_or\_and** 在函数中作为 register 使用

# 函数

## 主要特性:

- 函数定义中不能包含任何时序控制语句。
- 函数至少有一个输入，不能包含任何输出或双向端口。
- 函数只返回一个数据，其缺省为reg类型。
- 传送到函数的参数顺序和函数输入参数的说明顺序相同。
- 函数在模块（**module**）内部定义。
- 函数不能调用任务，但任务可以调用函数。
- 函数在Verilog中定义了一个新的范围（**scope**）。
- 虽然函数只返回单个值，但返回的值可以直接给信号连接赋值。这在需要多个输出时非常有效。

```
{o1, o2, o3, o4} = f_or_and (a, b, c, d, e);
```

# 函数

要返回一个向量值（多于一位），在函数定义时在函数名前说明范围。函数中需要多条语句时用**begin**和**end**。

不管在函数内对函数名进行多少次赋值，值只返回一次。下例中，函数还在内部声明了一个整数。

```
module foo;
    input [7: 0] loo;
    output [7: 0] goo;
    // 可以持续赋值中调用函数
    wire [7: 0] goo = zero_count ( loo );
    function [3: 0] zero_count;
        input [7: 0] in_bus;
        integer i;
        begin
            zero_count = 0;
            for (i = 0; i < 8; i = i + 1)
                if (! in_bus[ i ])
                    zero_count = zero_count + 1;
        end
    endfunction
endmodule
```

# 函数

函数返回值可以声明为其它register类型: integer, real, 或time。

在任何表达式中都可调用函数

```
module checksub (neg, a, b);  
    output neg;  
    reg neg;  
    input a, b;  
    function integer subtr;  
        input [7: 0] in_a, in_b;  
        subtr = in_a - in_b; // 结果可能为负  
    endfunction  
    always @ (a or b)  
        if (subtr( a, b) < 0)  
            neg = 1;  
        else  
            neg = 0;  
endmodule
```

# 函数

函数中可以对返回值的个别位进行赋值。

函数值的位数、函数端口甚至函数功能都可以参数化。

```
...  
parameter MAX_BITS = 8;  
reg [MAX_BITS: 1] D;  
function [MAX_BITS: 1] reverse_bits;  
    input [MAX_BITS-1: 0] data;  
    integer K;  
    for (K = 0; K < MAX_BITS; K = K + 1)  
        reverse_bits [MAX_BITS - (K+ 1)] = data [K]  
endfunction  
always @ (posedge clk)  
    D = reverse_bits (D) ;  
...
```



## 第八节 其他论题

8.1 任务与函数

**8.2 *wait*语句与命名事件**

8.3 命名的块与disable

8.4 作用域规则和层次名

8.5 延时描述



# wait语句

- wait语句相当于定时控制中的事件控制部分。不同的是@开头的事件控制都是边沿触发，而wait语句用的是电平触发方式。wait是使两个进程并发的手段，可实现自定时系统（例如通过完全互锁握手信号），不需要像时钟这样的外部同步信号。

- wait语句以如下形式给出：

**wait** (*Condition*)  
*procedural\_statement*

- 过程语句只有在条件为真时才执行，否则过程语句一直等待到条件为真。如果执行到该语句时条件已经为真，那么过程语句立即执行。在上面的表示形式中，过程语句是可选的。例如：

```
wait (Sum > 22)  
Sum = 0;  
wait (DataReady)  
Data = Bus;  
wait (Preset);
```



# 命名事件

- 命名事件必须在使用前声明。声明形式如下：

**`event Ready, Done;`**

事件声明说明**`Ready`**和**`Done`**为两个命名事件。

- 声明命名事件后，可以使用事件触发语句创建事件。形式如下：

**`-> Ready;`**

**`-> Done;`**



# 命名事件

- 命名事件上的事件能够同变量上的事件一样被监控，即使用@机制，例如：

`@ ( Done ) <动作语句>`

所以只要**D o n e** 上的事件触发语句被执行，一个事件就在**D o n e** 上发生，这使<动作语句>执行。

# 命名事件

- 命名事件是除线网类型和寄存器类型以外的一种特殊的数据类型，他没有值的概念，只用于表明一个抽象的事件在某一特定时刻或特定条件下被触发。
- 每触发一次，相当于产生一个数字信号课程中所定义的单位冲击函数，可以通过@ (事件名)的方式对它进行检测。
- 命名事件在定义它的模块中，不必将它列入端口名表项中作为输出，而其它对该命名事件进行检测的模块也不必将它列为输入端口。
- 命名事件只是一个纯数学意义上的抽象事件的表达，它的主要用途也是作为模块间通信的握手信号，但通常只出现在对模块功能的前期描述中，之后将被其他具体的实现方式所替代，完成综合化设计。



# 命名事件的示例

```
event Ready, Done
```

```
initial
```

```
->Done;
```

```
always
```

```
@(Done) begin
```

```
...
```

```
//触发下一个always 语句。
```

```
//在Ready 上创建一个事件。
```

```
->Ready;
```

```
end
```

```
always
```

```
@(Ready) begin
```

```
...
```

```
//创建事件来触发前面的always 语句。
```

```
->Done;
```

```
end
```



## 第八节 其他论题

8.1 任务与函数

8.2 wait语句与命名事件

**8.3 命名的块与*disable***

8.4 作用域规则和层次名

8.5 延时描述



## 命名的块（Named Blocks）

- 可以在**begin**或**fork**之后加上块名实现对一个块进行命名
- 在命名的块中可以定义局部变量
- 可以用关键字**disable**使一个命名的块失去作用
- 命名块的使用缩短了仿真时间



# 命名的块举例

```
module named_blk;  
    . . .  
    begin : seq_blk  
    . . .  
    end  
    . . .  
    fork : par_blk  
    . . .  
    join  
    . . .  
endmodule
```





# disable的使用

- **disable**可以终止命名的块或**task**的活动，语法格式是：

**disable** <name\_of\_block>

**disable** <task\_name>

- **disable**语句在尚未执行该命名块或任务任何一条语句前，就从该命名块/任务执行中返回。
- 禁止执行命名块或任务后，所有在事件队列中由该命名块/任务安排的事件都将被删除。
- **disable**通常不可综合



```
module do_arith(out, a, b, c, d, e, clk, en_mult);  
    input clk, en_mult;  
    input [7: 0] a, b, c, d, e;  
    output [15: 0] out;  
    reg[15: 0] out;  
  
    always @( posedge clk)  
        begin : arith_block           // 命名的块  
            reg[3: 0] tmp1, tmp2; // 局部变量定义  
                {tmp1, tmp2} = f_or_and(a, b, c, d, e);  
                    // function调用  
                if (en_mult) multme(tmp1, tmp2, out);  
                    // task调用  
        end
```



## disable的举例（续）

```
always @( negedge en_mult)
    begin // Abort the arithmetic
        disable multme; // *** Disable task ***
        disable arith_block; // *** Disable named block ***
    end
    // Task and function definitions go here
endmodule
```



## 第八节 其他论题

8.1 任务与函数

8.2 wait语句与命名事件

8.3 命名的块与disable

**8.4 作用域规则和层次名**

8.5 延时描述



# 作用域规则

- 一个标识符的作用域是Verilog描述中该标识符可以被识别的范围。作用域规则定义了这个范围。
- 作用域规则：
  - 模块名称是全局可见的。
  - 可以定义标识符的实体包括：模块、任务、函数和命名的块。每个实体定义了标识符的局部作用域。局部作用域分别是：
    - 模块：module -endmodule
    - 任务：task -endtask
    - 函数：function -endfunction
    - 命名的块：begin: name -end



## 作用域规则(2)

- 标识符在局部作用域之外也是可见的。规则：
  - 超前引用：在标识符定义之前就引用。模块、任务、函数和命名的块的标识符可以超前引用。可以在这些实体定义之前进行一个**module**的实体化，调用一个任务或函数，或终止一个命名的块。
  - 非超前引用：寄存器和线网不能超前引用。需要先定义后使用。通常是在使用它们的局部作用域开始处定义它们。
  - 可以超前引用的实体（模块、任务、函数和命名的块）中，由模块实体化构成了一个向上作用域。从底层可以识别每一个比它层次高的局部作用域中的超前引用标识符。



# 层次名

- 层次名可以唯一标识整个设计中的任何一个任务、函数、命名的块、寄存器和线网。
- 层次名是由“.”隔开的一系列标识符组成的路径名称。



# 作用域和层次名称举例

```
module top;  
    reg r; // hierarchical name is top.r  
    wire w; // hierarchical name is top.w  
    b instance1 ();  
  
    always  
    begin : y  
        reg q; // hierarchical name is top.y.q  
    end  
  
    task t;  
    begin:c // hierarchical name is top.t.c  
        reg q; // hierarchical name is top.t.c.q  
        disable y; // OK  
    end  
    endtask  
endmodule
```

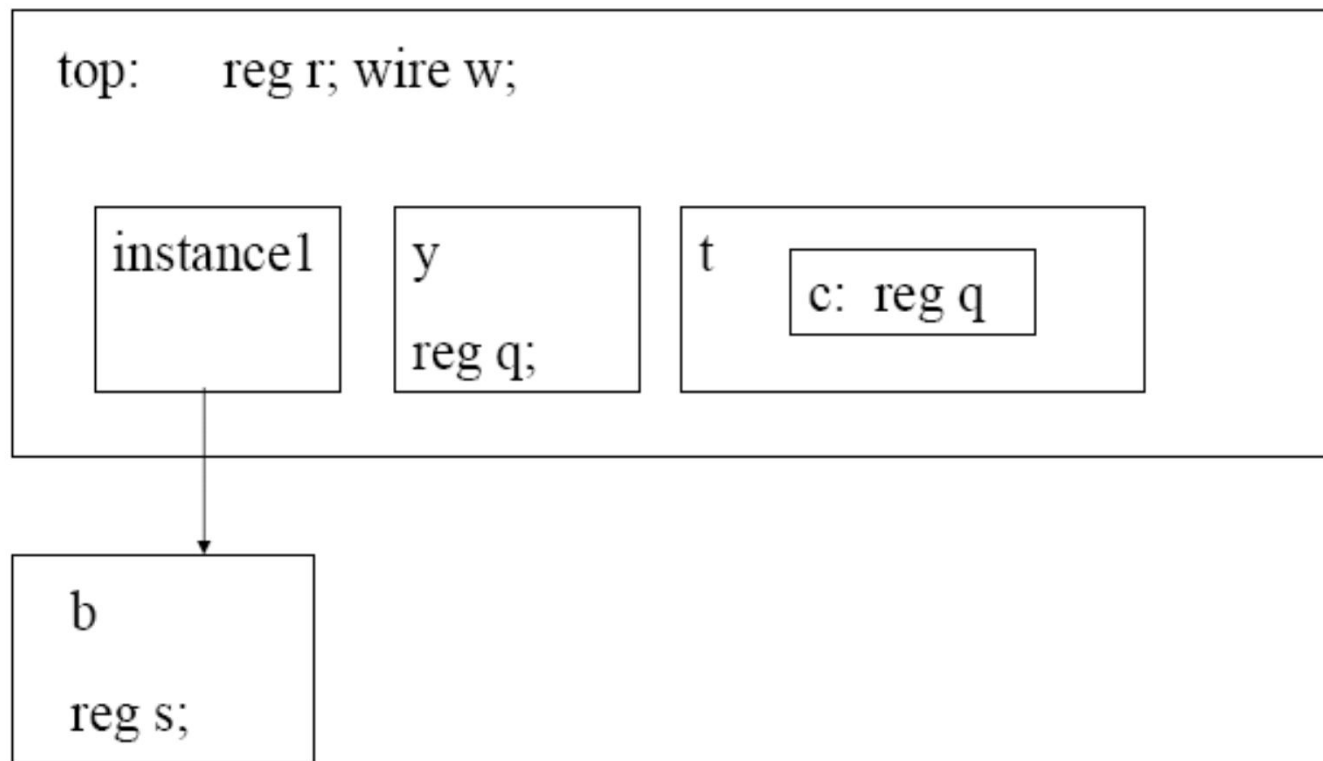




## 作用域和层次名称举例（续）

```
module b;  
    reg s;                // hierarchical name is top.instance1.s  
    always  
    begin  
        t;                // OK  
        disable y;        // OK  
        disable c;        // Nope, c is not known  
        disable t.c;      // OK  
        s = 1;            // OK  
        r = 1;            // Nope, r is not known  
        top.r = 1;        // OK  
        t.c.q = 1;        // OK  
        y.q = 1;          // OK, a different q than t.c.q  
    end  
endmodule
```

## 作用域和层次名称举例（续）



# 作用域和层次名称举例（续）

本例中的层次名为：

*Top.C1.Art*

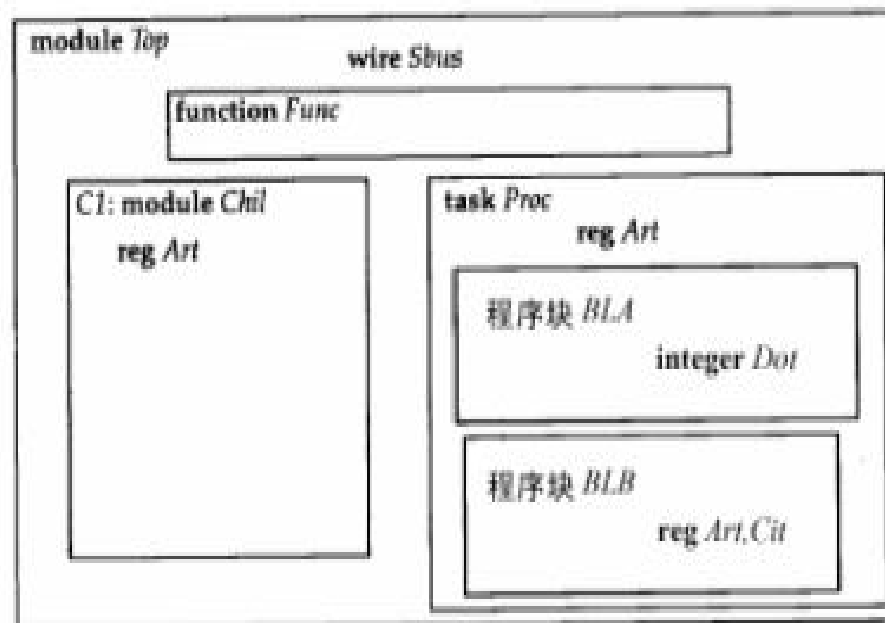
*Top.Proc.Art*

*Top.Proc.BLB.Art*

*Top.Proc.BLA.Dot*

*Top.Proc.BLB.Cit*

*Top.Sbus*





## 第八节 其他论题

8.1 任务与函数

8.2 wait语句与命名事件

8.3 命名的块与disable

8.4 作用域规则和层次名

**8.5 延时描述**

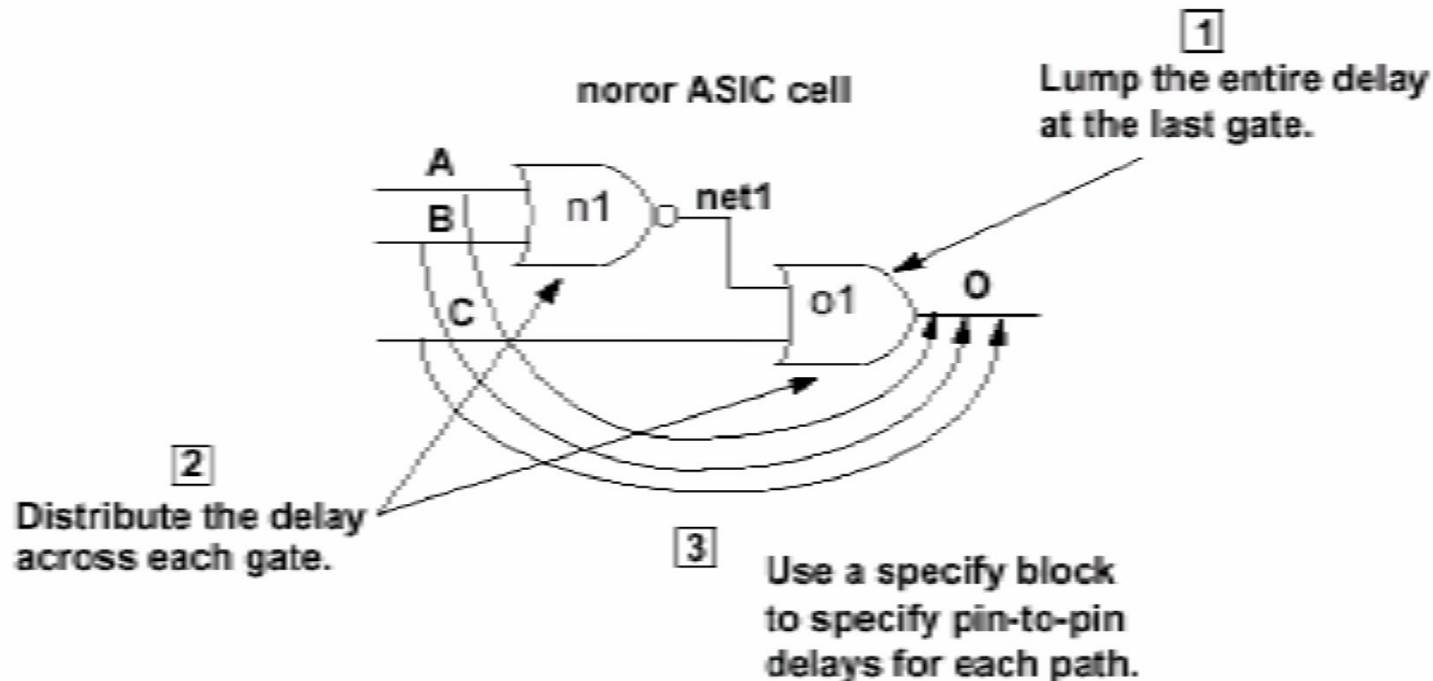


# 基本概念

- 模块路径（**Module Path**）：经过模块连接一个输入端口（包括input和inout）与一个输出端口（包括output和inout）的路径
- 路径延迟（**Path Delay**）：一条特定路径的延迟
- 上升延迟（**Rise Delay**）：与变化到1相关的延迟
- 下降延迟（**Fall Delay**）：与变化到0相关的延迟
- 关断延迟（**Turn-off Delay**）：与变化到高阻Z相关的延迟

# 延迟模型的种类

- 集中延迟（**Lumped Delay**）：将所有的延迟放在最后的门上，所有路径延迟相同
- 分布延迟（**Distributed Delays**）：将延迟分配在不同的门上，不同的路径可以有不同的延迟
- 模块路径延迟（**Module Path Delays**）：说明一个模块从输入到输出的延迟。所有路径可以精确描述，时间关系和功能可以划分开





# 不同延迟模型的特点

- 集中延迟：延迟可集中在驱动输出的最后一个门上
  - 集中延迟易于描述
  - 但只有在简单的情况才比较精确
- 分布延迟：将延迟分布在各个门上
  - 比集中延迟精确，但也不是总能精确描述
- 模块路径延迟：在一个**specify**块中说明模块的路径延迟
  - 易于描述
  - 可以精确匹配延迟说明
  - 对0、1和Z之间的转化可以指定不同的延迟
  - 可以将时序与功能分开
  - 是最常用的延迟类型

# 集中延迟举例

```
`timescale 1ns/ 1ns  
module noror( Out, A, B, C);  
    output Out;  
    input A, B, C;  
    nor n1 (net1, A, B);  
    or #3 o1 (Out, C, net1);  
endmodule
```

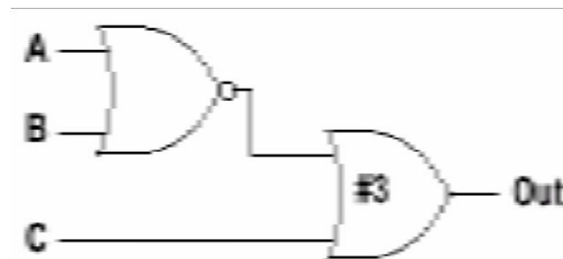
描述的路径延迟:

**A-> Out is 3**

**B-> Out is 3**

**C-> Out is 3**

简单，无法描述不同路径的延迟





# 分布式延迟举例

```
module noror( Out, A, B, C);  
  output Out;  
  input A, B, C;  
  nor #2 n1 (net1, A, B);  
  or #1 o1 (Out, C, net1);  
endmodule
```

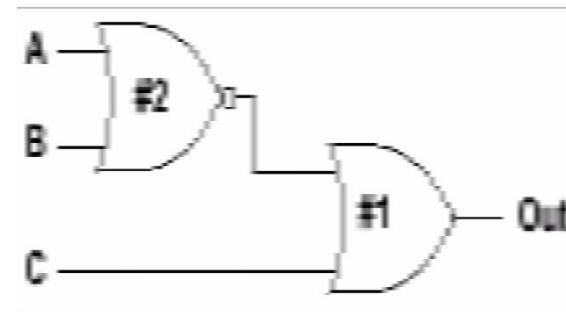
描述的路径延迟:

**A-> Out is 3**

**B-> Out is 3**

**C-> Out is 1**

对实际模块可能变得很复杂；对同一个原语的输入延迟相同



# 路径延迟举例

```
module noror( O, A, B, C);  
    output O; input A, B, C;  
    nor n1 (net1, A, B);  
    or o1 (O, C, net1);  
    specify  
        (A ==> O) = 2;  
        (B ==> O) = 3;  
        (C ==> O) = 1  
    endspecify  
endmodule
```

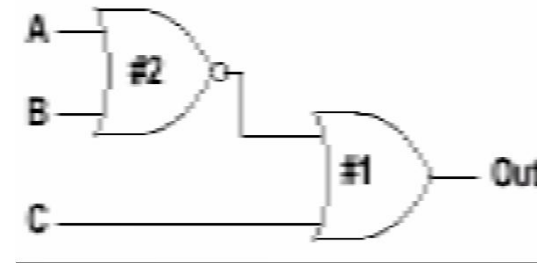
描述的路径延迟

**A-> Out is 2**

**B-> Out is 3**

**C-> Out is 1**

用**specify**指定路径的延迟，可以达到最大精度





# 延时说明（**specify**）块

- **specify** 块指定了模块的时间信息，这个时间信息和功能是独立的。
- **specify** 块以**specify** 开始，以**endspecify**结束。
- **specify** 块出现在**module**范围中。
- 在**specify** 块中可以使用**specparam**定义参数。
- 通常，延时说明块有如下用途：
  - 1) 指定源和目的之间的路径；
  - 2) 为这些路径分配时延；
  - 3) 对模块进行时序校验。



# 延时说明块格式

**specify**

*spec\_param\_declarations* // 参数说明

*path\_declarations* // 路径说明

*system\_timing\_checks* // 系统时序校验说明

**endspecify**



# 模块路径

- 并行和完全连接两种模块路径：

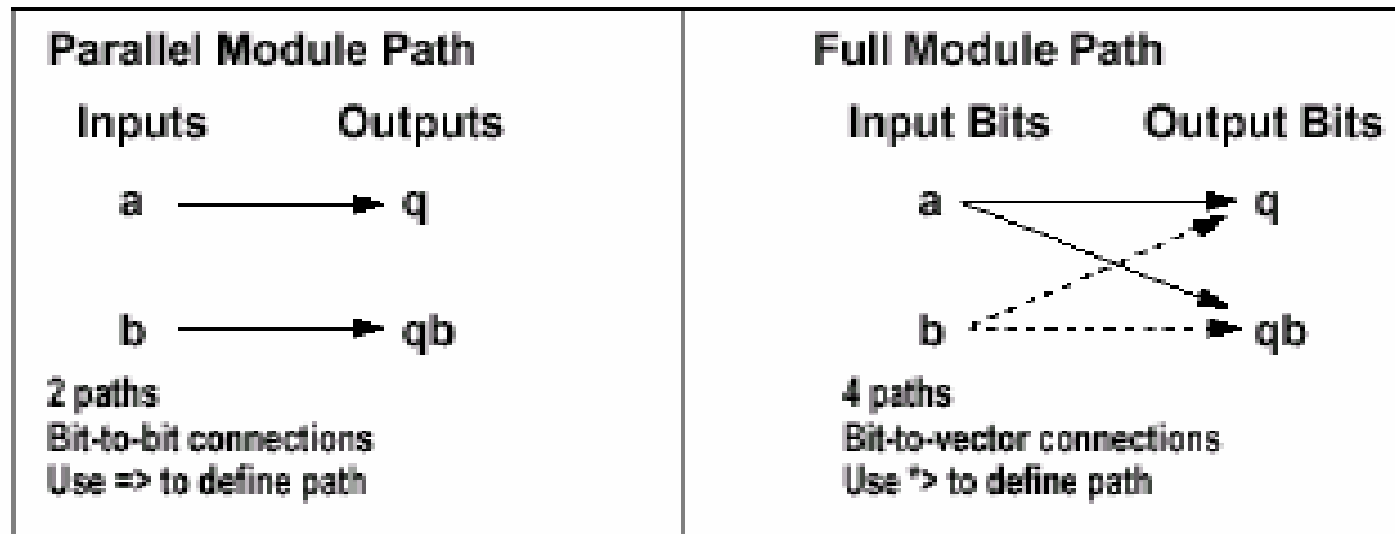
*source \*> destination*

// 指定一个完全连接：源参数上的每一位都与目的参数的所有位相连接。

*source => destination*

// 指定一个并行连接：源参数上的每一位分别与目的参数的位一一连接。

# 并行和完全连接模块路径举例（1）



`(a, b => q, qb) = 15;`

is equivalent to

`(a => q) = 15;`  
`(b => qb) = 15;`

`(a, b *> q, qb) = 15;`

is equivalent to:

`(a => q) = 15;`  
`(b => q) = 15;`  
`(a => qb) = 15;`  
`(b => qb) = 15;`



## 并行和完全连接模块路径举例（2）

- **// Path delay specified from a to out and from b to out.**  
**(a, b => out, out) = 2.2;**
- **// Rise and Fall delay specified from r to o1 and o2.**  
**(r \*> o1, o2) = (1, 2);**
- **/\* Path delays specified from a[1] to b[ 1] and from a[ 0] to b[ 0]. \*/**  
**(a[1: 0] => b[1: 0]) = 3; // parallel connection**
- **// Path delays specified for all paths from a to o.**  
**(a[7: 0] \*> o[7: 0]) = 6.3; // full connection**



# 依赖于状态的路径延迟

- 当指定条件成立时赋予指定模块路径的延迟。其中的if不能使用else项。

- 举例

```
module XOR2 (x, a, b);  
  input a, b;  
  output x;  
  xor(x, a, b);  
  specify  
    if (a) (b=> x) = (5: 6: 7);  
    if (!a) (b=> x) = (5: 7: 8);  
    if (b) (a=> x) = (4: 5: 7);  
    if (!b) (a=> x) = (5: 7: 9);  
  endspecify  
endmodule
```





# 模块的时序检测

- Verilog提供了多种时序验证系统任务，下面是可在延时说明块内使用的时序验证系统任务。
  - **\$setup**: 确定数据信号是否在使能信号变化之前保持足够长时间的稳定
  - **\$hold**: 检查数据信号是否在使能控制信号变化之后保持足够长时间的稳定
  - **\$setuphold**: 将\$setup和\$hold的功能结合在一起
  - **\$period**: 时钟周期检测，检查时钟的相同边之间的时间间隔是否小于指定值
  - **\$width**: 信号宽度检测，检查从一个边沿到相反边沿的延迟是否违反要求
  - **\$skew**: 指定两个信号之间的最大允许延迟



# 延时说明块举例

**specify**

//指定参数:

**specparam** tCLK\_Q = (4:5:6);

**specparam** tSETUP = 2.8, tHOLD = 4.4;

//指定通路的路径时延:

(Clock \*> Q) = tCLK\_Q;

(Data \*> Q) = 12;

(Clear, Preset \*> Q ) = (4,5);

//时序验证:

**\$setuphold** (negedge Clock, Data, tSETUP, tHOLD );

**endspecify**



# 内 容

- 第一节 硬件描述语言
- 第二节 Verilog HDL设计入门
- 第三节 Verilog基础知识
- 第四节 行为描述
- 第五节 数据流描述
- 第六节 结构描述
- 第七节 用户自定义元件
- 第八节 其他论题
- 第九节 仿真
- 第十节 实例分析



## 第九节 仿真

9.1 目的

9.2 波形产生

9.3 从文本文件中读取向量

9.4 向文本文件中写入向量

9.5 实例



## 第九节 仿真

### 9.1 目的

### 9.2 波形产生

### 9.3 从文本文件中读取向量

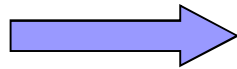
### 9.4 向文本文件中写入向量

### 9.5 实例

# 仿真的目的

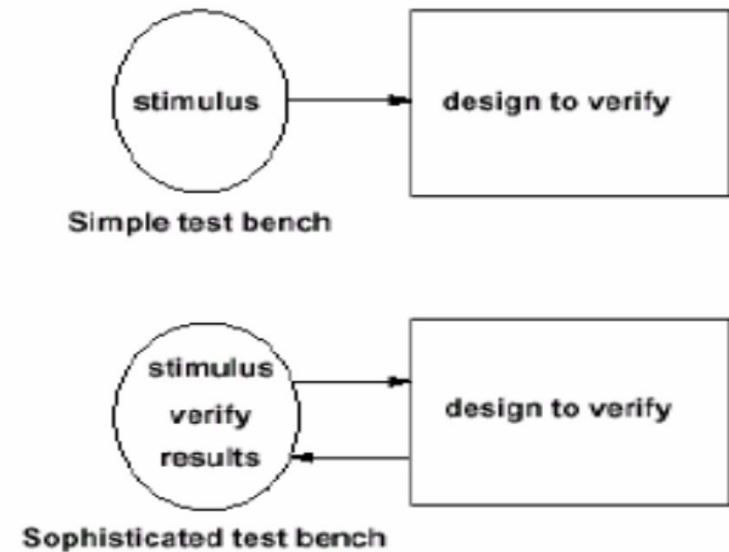
## ■ 通过仿真发现设计的逻辑与时序错误

- 简单的测试环境
- 复杂的测试环境



## ■ 仿真的过程:

- 产生仿真激励;
- 将输入激励加入到测试模块并收集其输出响应;
- 将响应输出与期望值进行比较。





# 典型的仿真程序

```
module Test_Bench;
```

```
// 通常测试验证程序没有输入和输出端口。
```

```
Local_reg_and_net_declarations
```

```
Generate_waveforms_using_initial_&_always_  
statements
```

```
Instantiate_module_under_test
```

```
Monitor_output_and_compare_with_expected_  
values
```

```
endmodule
```

测试中，通过在测试验证程序中进行实例化，激励自动加载于测试模块。



## 第九节 仿真

9.1 目的

**9.2 波形产生**

9.3 从文本文件中读取向量

9.4 向文本文件中写入向量

9.5 实例



# 值序列

**initial**

**begin**

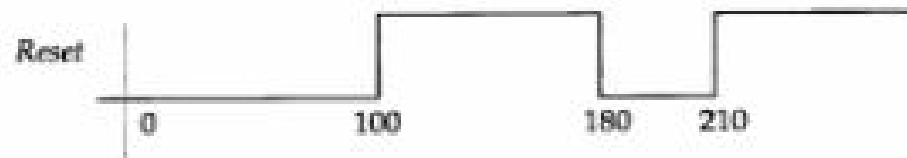
*Reset* = 0;

#100 *Reset* = 1;

#80 *Reset* = 0;

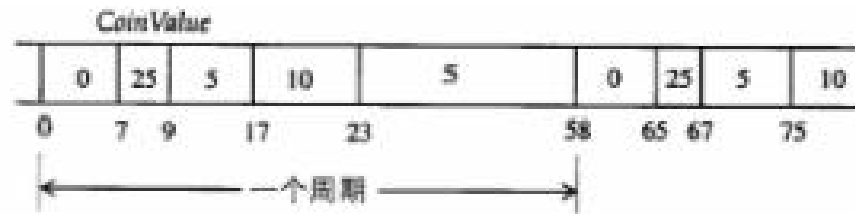
#30 *Reset* = 1;

**end**



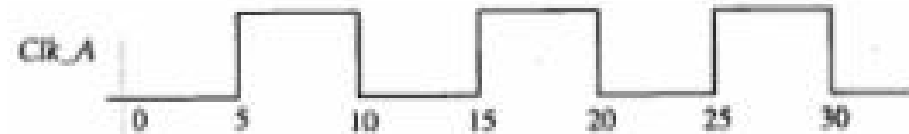
# 重复值序列

```
parameter REPEAT_DELAY = 35;  
integer CoinValue;  
always  
begin  
  CoinValue = 0;  
  #7 CoinValue = 25;  
  #2 CoinValue = 5;  
  #8 CoinValue = 10;  
  #6 CoinValue = 5;  
  #REPEAT_DELAY;  
end
```



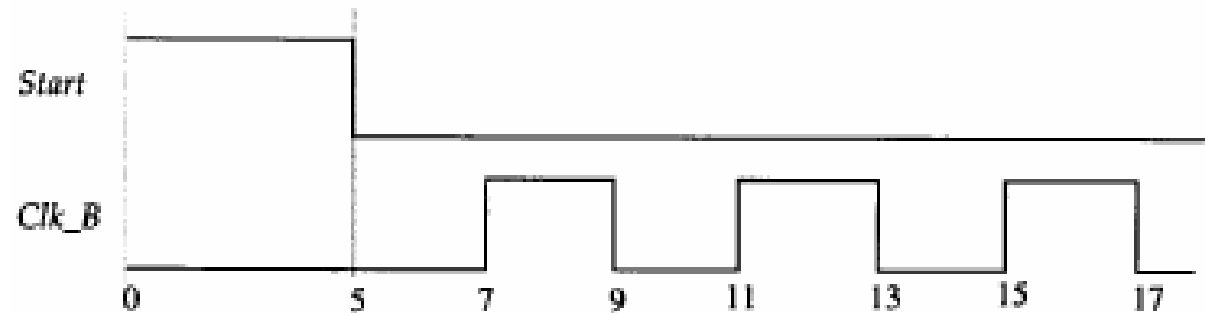
# 重复波形

```
module Gen_Clk_A (C / k _ A);  
output C / k _ A;  
reg C / k _ A;  
parameter tPERIOD = 10;  
initial  
  C / k _ A = 0;  
always  
  # (t P E R I O D / 2) C / k _ A = ~ C / k _ A;  
endmodule
```



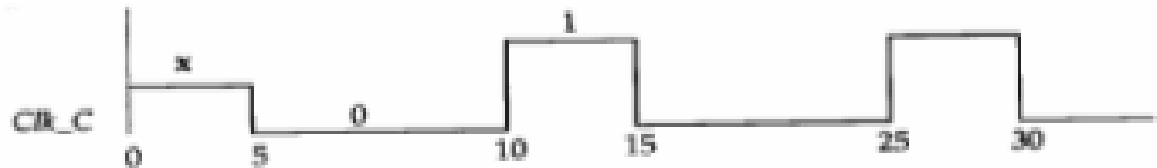
# 重复波形


```
module Gen_Clk_B (Clk_B);  
output Clk_B;  
reg Start;  
initial  
begin  
    Start = 1;  
    #5 Start = 0;  
end  
nor #2 (Clk_B, Start, Clk_B);  
endmodule
```



# 非等占空比的时钟

```
module Gen_Clk_C (Clk_C);  
parameter tON = 5, tOFF = 10;  
output Clk_C;  
reg Clk_C;  
always  
begin  
#tON Clk_C = 0;  
#tOFF Clk_C = 1;  
end  
endmodule
```



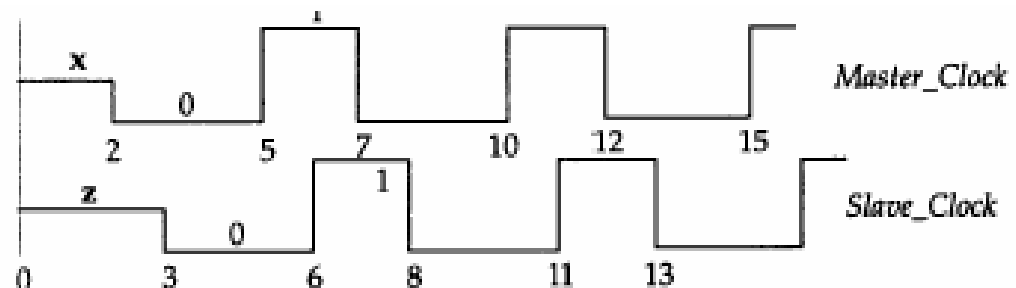


# 确定数目的时钟

```
module Gen_Clk_E (Clk_E);  
output Clk_E;  
reg Clk_E;  
parameter Tburst = 10, Ton = 2, Toff = 5;  
initial  
begin  
  Clk_E = 1'b0;  
  repeat(Tburst)  
  begin  
    #Toff Clk_E = 1'b1;  
    #Ton Clk_E = 1'b0;  
  end  
end  
endmodule
```

# 相移时钟

```
module Phase (Master_Clk, Slave_Clk );  
output Master_Clk, Slave_Clk;  
reg Master_Clk;  
wire Slave_Clk;  
parameter tON = 2, tOFF = 3, tPHASE_DELAY = 1;  
always  
begin  
# tON Master_Clk = 0;  
# tOFF Master_Clk = 1;  
end  
assign #tPHASE_DELAY Slave_Clk = Master_Clk;  
endmodule
```





## 第九节 仿真

9.1 目的

9.2 波形产生

**9.3 从文本文件中读取向量**

9.4 向文本文件中写入向量

9.5 实例



# 从文本文件中读取向量

- 可用 **\$ readmemb** 系统任务从文本文件中读取向量(通常可能包含输入激励和输出期望值)。下面为测试3位全加器电路的例子。假定文件“test.vec”包含如下两个向量。

A	B		期望的	Sum
010	010	0	1000	
010	011	1	1100	
		Cin		
			期望的	Count




# 从文本文件中读取向量

## 3位全加器的例子

```
module Adder1Bit (A, B, Cin, Sum, Cout) ;  
input A, B, Cin;  
output Sum, Cout;  
assign Sum = (A ^ B) ^ Cin;  
assign Cout = (A ^ B) | (A & Cin) | (B & Cin) ;  
endmodule
```

```
module Adder3Bit (First, Second, Carry_In, Sum_Out, Carry_Out) ;  
input [0:2] First, Second;  
input Carry_In;  
output [0:2] Sum_Out;  
output Carry_Out;  
wire [0:1] Car ;  
Adder1Bit  
A1 (First[2], Second[2], Carry_In, Sum_Out [2], Car[ 1 ] ),  
A2 (First[1], Second[1], Car[1], Sum_Out [1], Car[ 0 ] ) ,  
A3 (First[0], Second[0], Car[0], Sum_Out [0], Carry_Out) ;  
endmodule
```



```
module Test Bench;
parameter BITS = 11, WORDS = 2;
reg [1:BITS] Vmem[1:WORDS];
reg [0:2] A, B, Sum_Ex;
reg Cin, Cout_Ex;
integer J;
wire [0:2] Sum;
wire Cout;
//被测试验证的模块实例。
Adder3BitF1(A, B, Cin, Sum, Cout);
```

```
initial
begin
$readmemb("test.vec", Vmem);
for(J = 1; J <= WORDS; J = J + 1)
begin
{A, B, Cin, Sum_Ex, Cout_Ex} = Vmem[J];
#5; //延迟5 个时间单位等待电路稳定。
if( (Sum != Sum_Ex) || (Cout != Cout_Ex) )
$display("****Mismatch on vector %b ****", Vmem[J]);
else
$display("No mismatch on vector %b", Vmem[J]);
end
end
endmodule
```



## 第九节 仿真


9.1 目的

9.2 波形产生

9.3 从文本文件中读取向量

**9.4 向文本文件中写入向量**

9.5 实例



## 向文本文件中写入向量

- 设计中的信号值能通过如**\$fdisplay**、**\$fmonitor**和**\$fstrobe**等具有写文件功能的系统任务输出到文件中。
- 下面是与前一节中相同的测试验证模块实例，本例中的验证模块将所有输入向量和观察到的输出结果输出到文件“**mon.Out**”中。

```

module F_Test_Bench;
  parameter BITS = 11, WORDS = 2;
  reg [1:BITS] Vmem [1:WORDS];
  reg [0:2] A, B, Sum_Ex;
  reg Cin, Cout_Ex;
  integer J;
  wire [0:2] Sum;
  wire Cout;
  //待测试验证模块的实例。
  Adder3BitF1 (A, B, Cin, Sum, Cout);


```

```

initial
begin: INIT_LABEL
  integer Mon_Out_File;
  Mon_Out_File = $fopen ("mon.out");
  $readmemb ("test.vec", Vmem);
  for (J = 1; J <= WORDS; J = J + 1)
    begin
      {A, B, Cin, Sum_Ex, Cout_Ex} = Vmem [J];
      #5; //延迟5 个时间单位, 等待电路稳定。
      if ( (Sum != Sum_Ex) || (Cout != Cout_Ex) )
        $display ("****Mismatch on vector %b ****", Vmem [J]);
      else
        $display ("No mismatch on vector %b", Vmem [J]);
      //将输入向量和输出结果输入到文件:
      $fdisplay (Mon_Out_File, "Input = %b%b%b, Output = %b%b",
        A, B, Cin, Sum, Cout);
    end
  $fclose (Mon_Out_File);
end
endmodule

```

**\$fopen**是用来打开某个文件并  
准备写操作的系统任务;  
**\$fclose**是关闭文件的系统任务



# 向文本文件中写入向量

下面是仿真执行后文件“Mon.out”包含的内容。

Input = 0100100, Output = 1000

Input = 0100111, Output = 1100



## 第九节 仿真

9.1 目的

9.2 波形产生

9.3 从文本文件中读取向量

9.4 向文本文件中写入向量


**9.5 实例**





# 触发器

```
module MSDFF (D, C, Q, Qbar) ;  
input D, C;  
output Q, Qbar;  
not  
  NT1 (NotD, D),  
  NT2 (NotC, C),  
  NT3 (NotY, Y);  
nand  
  ND1 (D1, D, C) ,  
  ND2 (D2, C, NotD) ,  
  ND3 (Y, D1, Ybar) ,  
  ND4 (Ybar, Y, D2) ,  
  ND5 (Y1, Y, NotC) ,  
  ND6 (Y2, NotY, NotC) ,  
  ND7 (Q, Qbar, Y1) ,  
  ND8 (Qbar, Y2, Q) ;  
endmodule
```



```
module Test;  
reg D, C;  
wire Q, Qb;
```

```
MSDFF M1(D, C, Q, Qb);
```

```
always  
#5 C = ~ C;
```

```
initial  
begin  
D = 0;  
C = 0;  
#40 D = 1;  
#40 D = 0;  
#40 D = 1;  
#40 D = 0;  
$stop;  
end
```

```
initial  
$monitor("Time = %t ::", $time, "C=%b, D=%b, Q=%b,  
Qb=%b", C, D, Q, Qb );  
endmodule
```

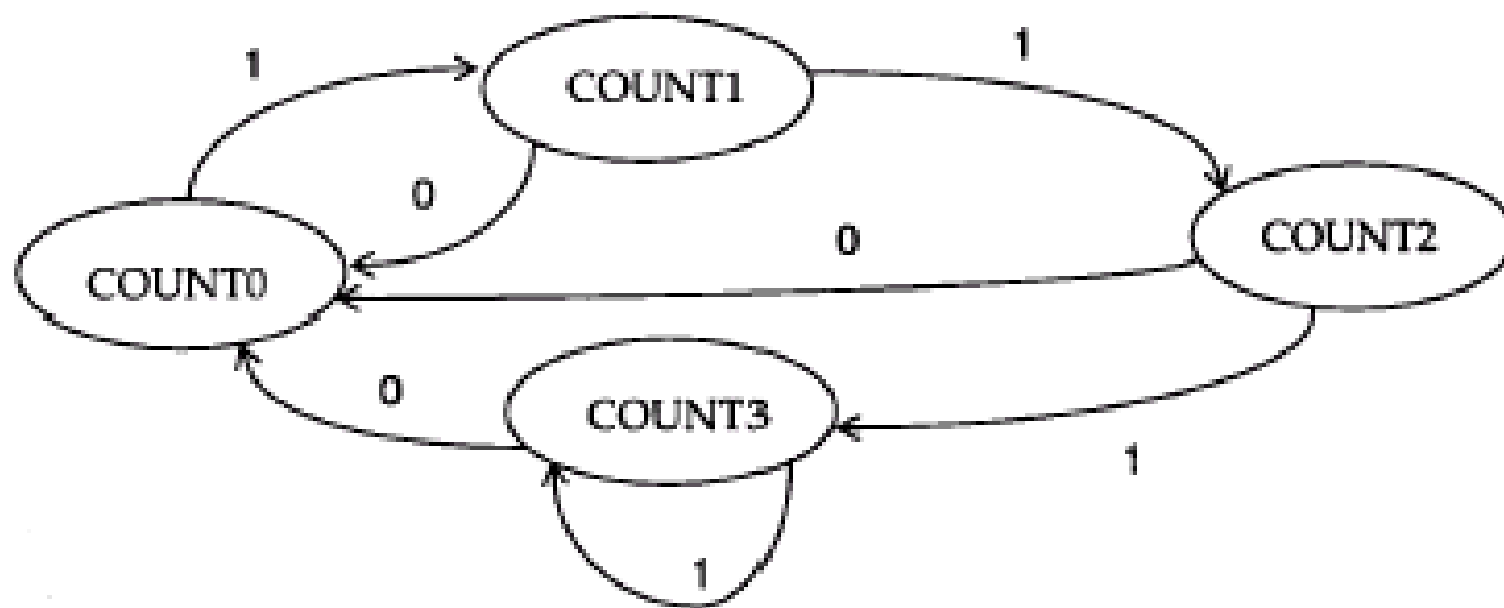



# 仿真结果

Time= 0:: C=0, D=0, Q=x, Qb=x  
Time= 5:: C=1, D=0, Q=x, Qb=x  
Time= 10:: C=0, D=0, Q=0, Qb=1  
Time= 15:: C=1, D=0, Q=0, Qb=1  
Time= 20:: C=0, D=0, Q=0, Qb=1  
Time= 25:: C=1, D=0, Q=0, Qb=1  
Time= 30:: C=0, D=0, Q=0, Qb=1  
Time= 35:: C=1, D=0, Q=0, Qb=1  
Time= 40:: C=0, D=1, Q=0, Qb=1  
Time= 45:: C=1, D=1, Q=0, Qb=1  
Time= 50:: C=0, D=1, Q=1, Qb=0  
Time= 55:: C=1, D=1, Q=1, Qb=0  
Time= 60:: C=0, D=1, Q=1, Qb=0  
Time= 65:: C=1, D=1, Q=1, Qb=0  
Time= 70:: C=0, D=1, Q=1, Qb=0  
Time= 75:: C=1, D=1, Q=1, Qb=0


Time= 80:: C=0, D=0, Q=1, Qb=0  
Time= 85:: C=1, D=0, Q=1, Qb=0  
Time= 90:: C=0, D=0, Q=0, Qb=1  
Time= 95:: C=1, D=0, Q=0, Qb=1  
Time= 100:: C=0, D=0, Q=0, Qb=1  
Time= 105:: C=1, D=0, Q=0, Qb=1  
Time= 110:: C=0, D=0, Q=0, Qb=1  
Time= 115:: C=1, D=0, Q=0, Qb=1  
Time= 120:: C=0, D=1, Q=0, Qb=1  
Time= 125:: C=1, D=1, Q=0, Qb=1  
Time= 130:: C=0, D=1, Q=1, Qb=0  
Time= 135:: C=1, D=1, Q=1, Qb=0  
Time= 140:: C=0, D=1, Q=1, Qb=0  
Time= 145:: C=1, D=1, Q=1, Qb=0  
Time= 150:: C=0, D=1, Q=1, Qb=0  
Time= 155:: C=1, D=1, Q=1, Qb=0

# 时序检测器





```
module Count3_Is (Data, Clock, Detect3_Is);  
input Data, Clock;  
output Detect3_Is;  
integer Count;  
reg Detect3_Is;  
initial  
begin  
    Count = 0;  
    Detect3_Is = 0;  
end  
always  
@ (negedge Clock) begin  
    if (Data == 1)  
        Count = Count + 1;  
    else  
        Count = 0;  
    if (Count >= 3)  
        Detect3_Is = 1;  
    else  
        Detect3_Is = 0;  
    end  
end  
endmodule
```



```
module Test;
reg Data, Clock;
integer Out_File;
Count3_Is F1 (Data, Clock, Detect );
initial
begin
Clock = 0;
forever
#5 Clock = ~ Clock ;
end
initial
begin
Data = 0;
#5 Data = 1;
#40 Data = 0;
#10 Data = 1;
#40 Data = 0;
#20 $stop; // 仿真暂停。
end
initial
begin
//在文件中保存监控信息。
Out_File = $fopen("results.vectors");
$monitor(Out_File,"Clock = %b, Data = %b, Detect = %b",
Clock, Data, Detect);
end
endmodule
```

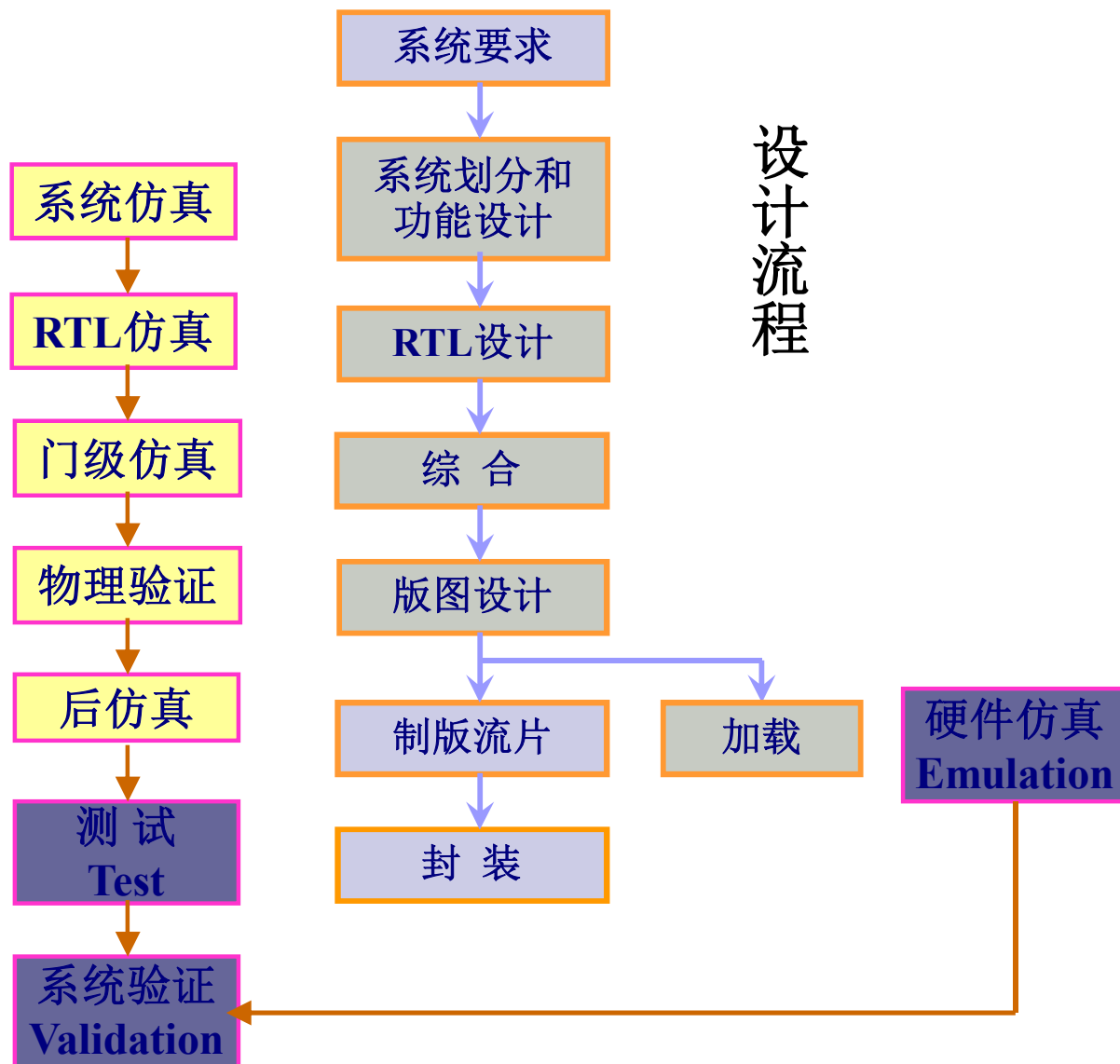


# 小结

- 目前设计验证的工作已经占总的设计工作量的40%~70%，其中很大部分是仿真。
- 仿真要求能尽可能全面地覆盖实际工作环境中的各种情况，它与测试不同，但有类似的地方。
- 仿真主要针对设计的功能，需要考虑功能覆盖是否全面。对设计来说还要考虑仿真过程对代码的覆盖率：
  - 语句覆盖率：对代码中每条语句的覆盖情况
  - 路径覆盖率：执行一系列指令时对各种分支的组合情况的覆盖情况
  - 表达式覆盖率：对判断条件的表达式的覆盖情况

# 设计的每个环节都需要验证

验证流程




设计流程

集成电路设计 II

设计创意

+ 验证





# 内 容

- 第一节 硬件描述语言
- 第二节 Verilog HDL设计入门
- 第三节 Verilog基础知识
- 第四节 行为描述
- 第五节 数据流描述
- 第六节 结构描述
- 第七节 用户自定义元件
- 第八节 其他论题
- 第九节 仿真
- 第十节 实例分析



# 实例分析

## ■ 组合电路设计示例

- 加法器
- 乘法器
- 数据分配器
- 译码和编码器
- 表决器

## ■ 时序电路设计示例

- D 触发器
- JK 触发器
- 同步计数器
- 移位寄存器
- 有限状态机

# 加法器\_半加器

## ■ 数据流描述的1位半加器

```
module halfadder (sum,cout,a,b);  
input a,b;  
output sum,cout;  
assign sum=a^b;  
assign cout=a&b;// carry out;  
endmodule
```

## ■ 结构描述的1位半加器

```
module half_add1(a,b,sum,cout);  
input a,b;  
output sum,cout;  
and (cout,a,b);  
xor (sum,a,b);  
endmodule
```

## ■ 行为描述的1位半加器

```
module half_add4(a,b,sum,cout);  
input a,b;  
output sum,cout;  
reg sum,cout;  
always @(a or b)  
begin  
sum= a^b;  
cout=a&b;  
end  
endmodule
```



# 加法器\_全加器

## ■ 1位全加器\_行为描述

```
module full_add4(a,b,cin,sum,cout);
    input a,b,cin;
    output sum,cout;
    reg sum,cout;
    reg m1,m2,m3;
    always @(a or b or cin)
    begin
        sum = (a ^ b) ^ cin;
        m1 = a & b;
        m2 = b & cin;
        m3 = a & cin;
        cout = (m1|m2)|m3;
    end
endmodule
```

## ■ 1位全加器\_数据流描述

```
module full_add(a,b,cin,sum,cout);
    input a,b,cin;
    output sum,cout;
    assign {cout,sum}=a+b+cin;
endmodule
```

## ■ 1位全加器\_结构描述

```
module full_add1(a,b,cin,sum,cout);
    input a,b,cin;
    output sum,cout;
    wire s1,m1,m2,m3;
    and (m1,a,b),
        (m2,b,cin),
        (m3,a,cin);
    xor (s1,a,b),
        (sum,s1,cin);
    or (cout,m1,m2,m3);
endmodule
```



# 全减器

```
module full_sub(diff,sub_out,x,y,sub_in);  
output diff,sub_out;  
input x,y,sub_in;  
reg diff,sub_out;  
//行为描述  
always@(x or y or sub_in)  
case ({x,y,sub_in})  
    3'b000 : begin diff= 0; sub_out = 0;end  
    3'b001 : begin diff= 1; sub_out = 1;end  
    3'b010 : begin diff= 1; sub_out = 1;end  
    3'b011 : begin diff= 0; sub_out = 1;end  
    3'b100 : begin diff= 1; sub_out = 0;end  
    3'b101 : begin diff= 0; sub_out = 0;end  
    3'b110 : begin diff= 0; sub_out = 0;end  
    3'b111 : begin diff= 1; sub_out = 1;end  
    default: begin diff= x; sub_out =x;end  
endcase  
endmodule
```



# 乘法器

- 四位乘法器

```
module mult_4 (X, Y, Product);
```

```
input [3:0] X, Y;
```

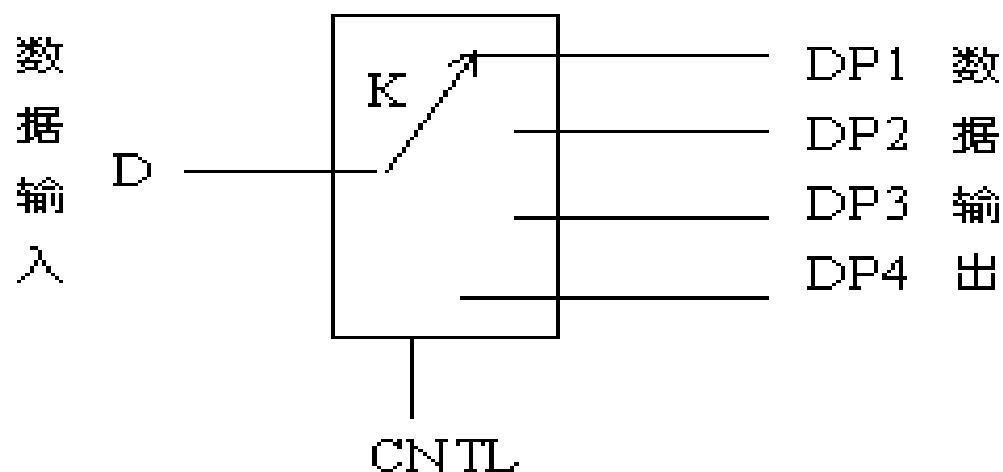
```
output [7:0] Product;
```

```
assign Product=X * Y;
```

```
endmodule
```

# 数据分配器

数据传输中，有时需要将数据分配到不同的数据通道上，能够完成这种功能的电路称为数据分配器，其电路结构为单输入多输出形式，功能如同多位开关，将输入**D**送到指定的数据通道上。



**module** demux (reset, cntl, d,dp1,dp2,dp3,dp4);

**input** reset; //复位信号

**input** [1:0] cntl; //控制信号，决定输入数据的流向

**input** [3:0] d; //输入数据

**output** [3:0] dp1; //数据通道1

**output** [3:0] dp2; //数据通道2

**output** [3:0] dp3; //数据通道3

**output** [3:0] dp4; //数据通道4

**wire** reset;

**wire** [1:0] cntl;

**wire** [3:0] d;

**reg** [3:0] dp1, dp2, dp3, dp4;

**always @** (reset or cntl or d)

**if** (reset)

**begin** //复位

dp1=4'b0;

dp2=4'b0;

dp3=4'b0;

dp4=4'b0;

**end**

**else**

**case** (cntl) //通道选通

2'b00: dp1 = d;

2'b01: dp2 = d;

2'b10: dp3 = d;

2'b11: dp4 = d;

**default: begin** //缺省

dp1=4'bzzzz;

dp2=4'bzzzz;

dp3=4'bzzzz;

dp4=4'bzzzz;

**end**

**endcase**

**endmodule**



**module** demux (reset, cntl, d, dp1, dp2, dp3, dp4);

**input** reset; //复位信号

**input** [1:0] cntl; //控制信号，决定输入数据的流向

**input** [3:0] d; //输入数据

**output** [3:0] dp1; //数据通道1

**output** [3:0] dp2; //数据通道2

**output** [3:0] dp3; //数据通道3

**output** [3:0] dp4; //数据通道4

**wire** reset;

**wire** [1:0] cntl;

**wire** [3:0] d;

**reg** [3:0] dp1, dp2, dp3, dp4;

**always @** (posedge clk)

**if** (reset)

**begin** //复位

dp1=4'b0;

dp2=4'b0;

dp3=4'b0;

dp4=4'b0;

**end**

**else**

**begin**

//通道选通

**if** ( cntl[1] )

**if** ( cntl[0] )

dp4=d;

**else**

dp3=d;

**else**

**if** ( cntl[0] )

dp2=d;

**else**

dp1=d;

**else**

**begin** //缺省

dp1 = 4'bzzzz;

dp2 = 4'bzzzz;

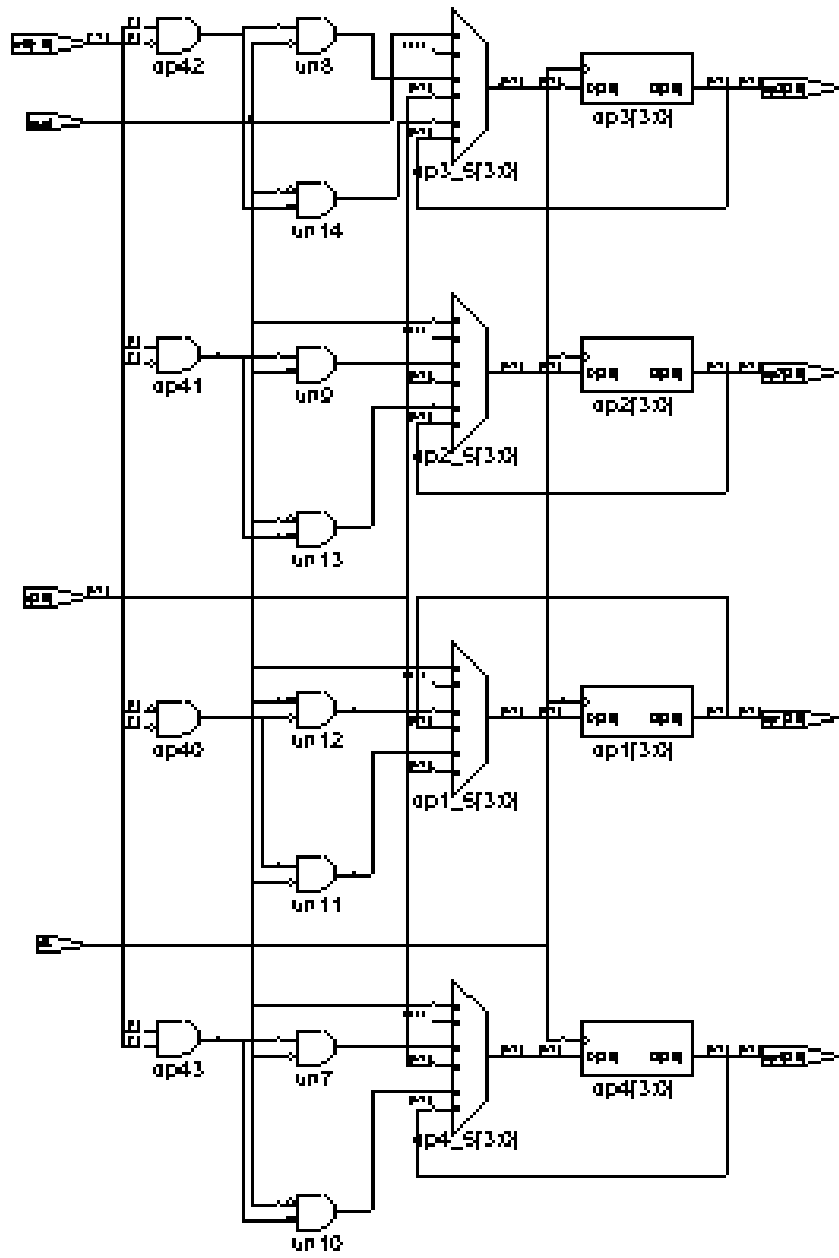
dp3 = 4'bzzzz;

dp4 = 4'bzzzz;

**end**

**end**


# 综合结果



# 译码器

3-8译码器有三个二进制输入端a、b、c和8个译码输出端Y0~Y7。根据输入a、b、c的值确定输出端Y0~Y7中一位有效，从而达到译码的目的。此外，3-8译码器还有一个三位的控制端cntl[2:0]，只有cntl=3'b100时才进行正常译码，否则Y0~Y7输出为高电平。





```
module decoder(a,b,c,cntl,y);
    input a,b,c;
    input [2:0] cntl;
    output [7:0] y;
    wire a,b,c;
    wire [2:0] cntl;
    reg [7:0] y;
    wire [2:0] data_in;
    assign data_in={c,b,a};           //输入码
    always @ (data_in or cntl)
        if ( cntl == 3'b100 )
            case (data_in)           //译码
                3'b000: y=8'b1111_1110;
                3'b001: y=8'b1111_1101;
                3'b010: y=8'b1111_1011;
                3'b011: y=8'b1111_0111;
                3'b100: y=8'b1110_1111;
                3'b101: y=8'b1101_1111;
                3'b110: y=8'b1011_1111;
                3'b111: y=8'b0111_1111;
            endcase
        else y=8'b1111_1111;         //失效
    endmodule
```



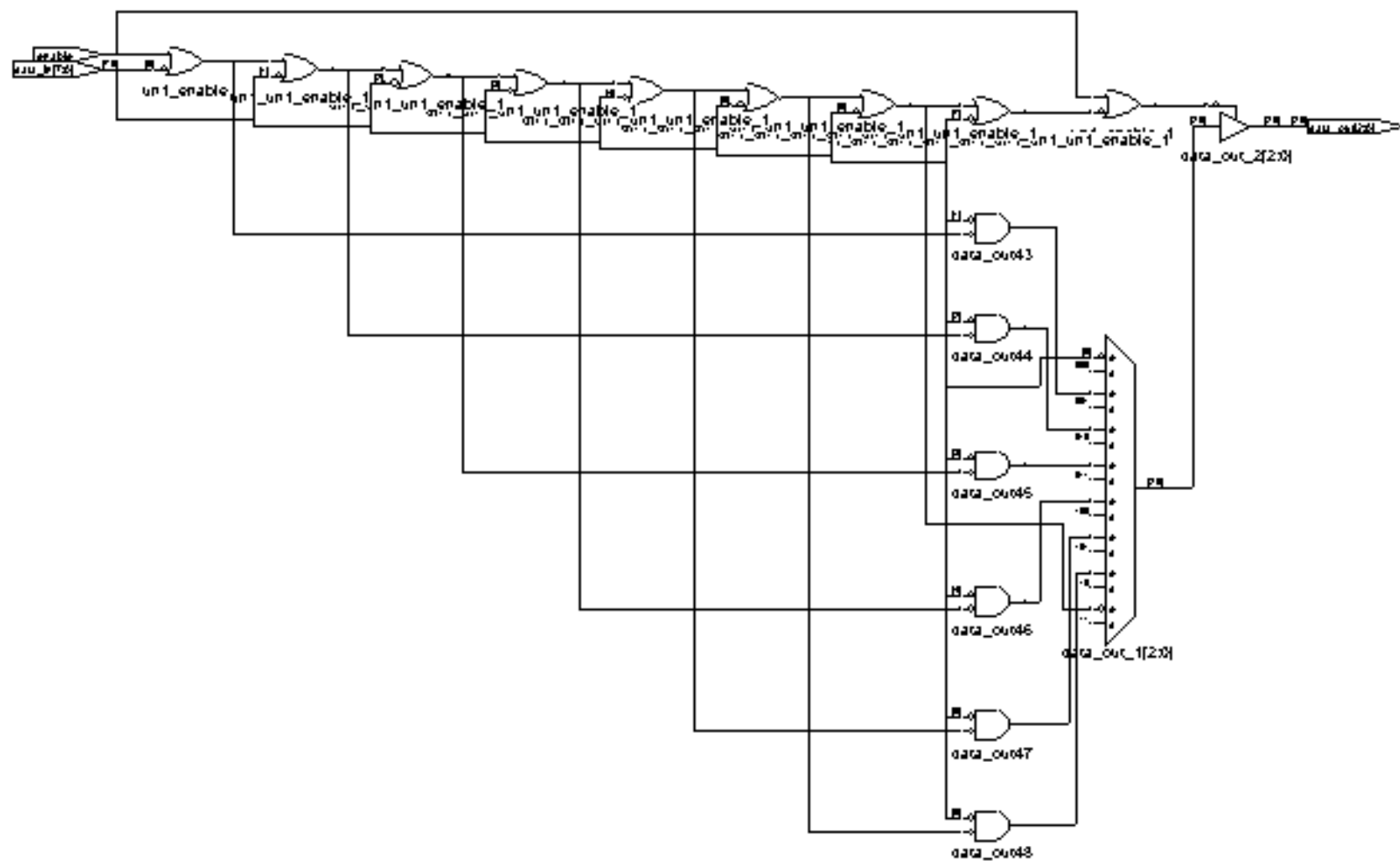
# 编码器

介绍一具有优先级的8-3编码器。可以使用if-else-if语句描述优先级电路，其基本语言结构如下：

<b>if</b>	(条件表达式1)	块语句1
<b>else if</b>	(条件表达式2)	块语句2
.....		
<b>else if</b>	(条件表达式n)	块语句n
<b>else</b>		块语句n+1



# 综合结果





# 七人表决器

## ■ for 语句描述的七人投票表决器

```
module voter7(pass,vote);  
output pass; // 通过为高电平，否则为低电平  
input[6:0] vote; // 7个投票输入，通过为高，否定为低  
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
```





# 典型功能模块

## ■ 组合电路设计示例

- 加法器
- 乘法器
- 数据分配器
- 译码和编码器

## ■ 时序电路设计示例

- D 触发器
- JK 触发器
- 同步计数器
- 移位寄存器
- 同步有限状态机



# D 触发器

```
module DFF(q,qn,d,clk,set,reset);  
    input d,clk,set,reset;  
    output q,qn;  
    reg q,qn;  
    always @(posedge clk)  
    begin  
        if (reset)  
        begin  
            q <= 0;  
            qn <= 1; //同步清0, 高电平有效  
        end  
        else if (set)  
        begin  
            q <= 1;  
            qn <= 0; //同步置1, 高电平有效  
        end  
    end
```

```
    else  
    begin  
        q <= d;  
        qn <= ~d;  
    end  
end  
endmodule
```



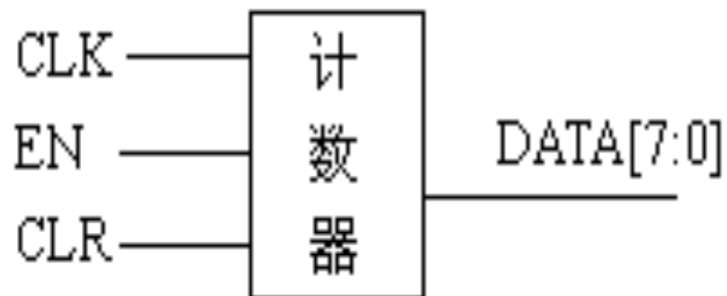
# JK触发器

## ■ 带异步清0、异步置1的JK触发器

```
module JK_FF(CLK,J,K,Q,RS,SET);  
    input CLK,J,K,SET,RS;  
    output Q;  
    reg Q;  
    always @(posedge CLK or negedge RS or negedge SET)  
    begin  
        if(!RS) Q <= 1'b0;  
        else if(!SET) Q <= 1'b1;  
        else case({J,K})  
            2'b00 : Q <= Q;  
            2'b01 : Q <= 1'b0;  
            2'b10 : Q <= 1'b1;  
            2'b11 : Q <= ~Q;  
            default: Q<= 1'bx;  
        endcase  
    end  
endmodule
```

# 同步计数器

同步计数器按时钟的节拍计数，可以设置异步或同步使能端和清零端。具有同步使能/清零端的8位二进制同步计数器示意图如下：





# 源 码

```
module counter(clk,en,clr,data);  
  input clk, en , clr;  
  output [7:0] data;  
  reg [7:0] data;  
  always @( posedge clk )  
    begin  
      if ( en )  
        if (clr || data == 8'b1111_1111)  
          data <= 8'b0000_0000;  
        else   data <= data+1;  
    end  
endmodule
```



```
`timescale 1ns/1ns
```

```
//定义时间精度
```

```
module test;
```

```
reg clk, en, clr;
```

```
wire [7: 0] data;
```

```
counter counter(clk,en,clr,data);
```

```
//源码实例
```

```
initial
```

```
//产生时钟信号，周期为100个时间单位
```

```
begin
```

```
    #10 clk=1;
```

```
    forever #50 clk=~clk;
```

```
end
```

```
initial
```

```
//测试使能功能
```

```
begin
```

```
    #10 en=0;
```

```
    #190 en=1;
```

```
    #150 en=0;
```

```
    #240 en=1;
```

```
    #1970980 en=0;
```

```
    #140 en=1;
```

```
end
```

```
initial
```

```
//测试清零功能
```

```
begin
```

```
    #10 clr=0;
```

```
    #130 clr=1;
```

```
    #150 clr=0;
```

```
end
```

```
initial
```

```
begin
```

```
    #2000000 $stop;
```

```
//经过2000000个时间单位，停止运行
```

```
    #2000 $finish;
```

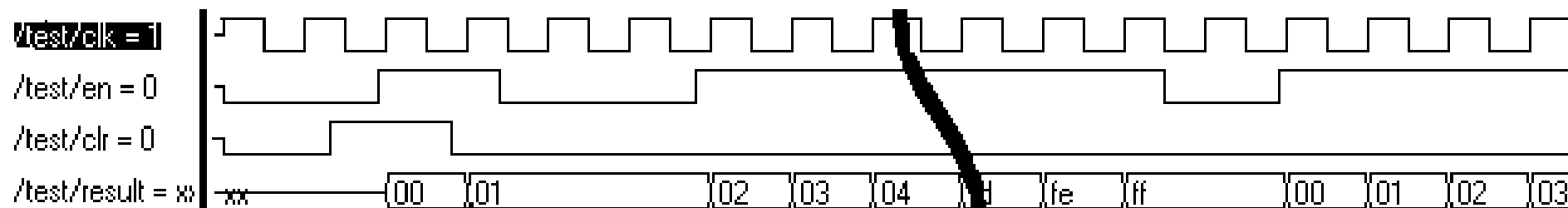
```
//再经过2000个时间单位，结束
```

```
end
```

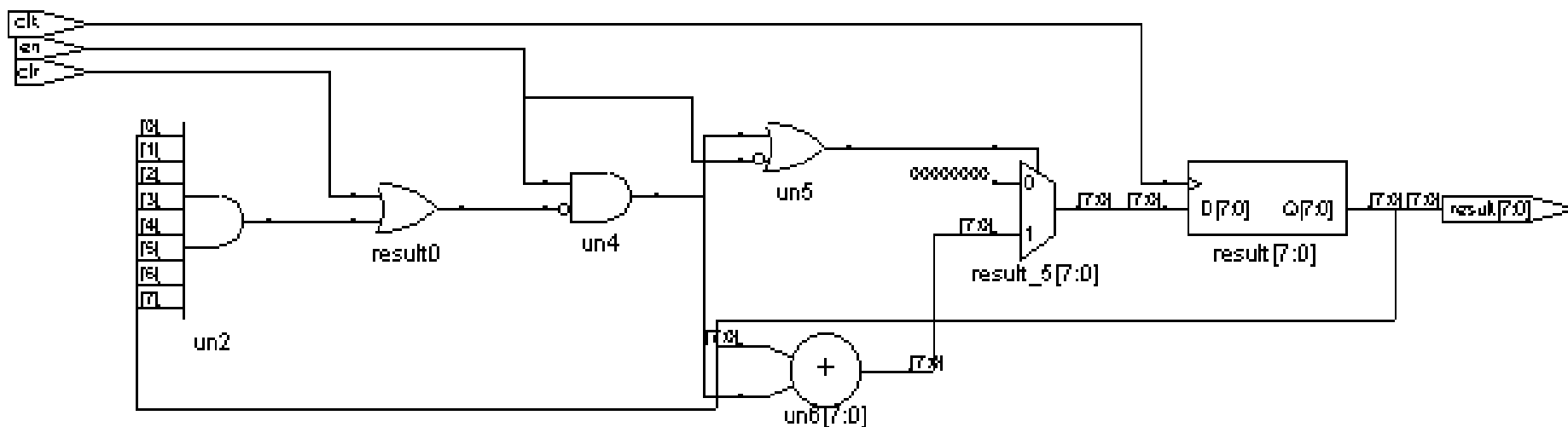
```
endmodule
```

# 测试文件

# 仿真时序图



## 综合结果





# 移位寄存器

## 左移寄存器的源码:

```
module      shift_left( clk, en, clr, data_in, data_out );
  input      clk,en,clr;
  input      [7:0] data_in;
  output     [7:0] data_out;
  wire       [7:0] data_in;
  reg        [7:0] data_out;
  always @ ( posedge clk )
    if ( en )
      if ( clr )
        data_out[7:0] = 8'b0;
      else
        data_out[7:0] = data_in << 1 ;
endmodule
```



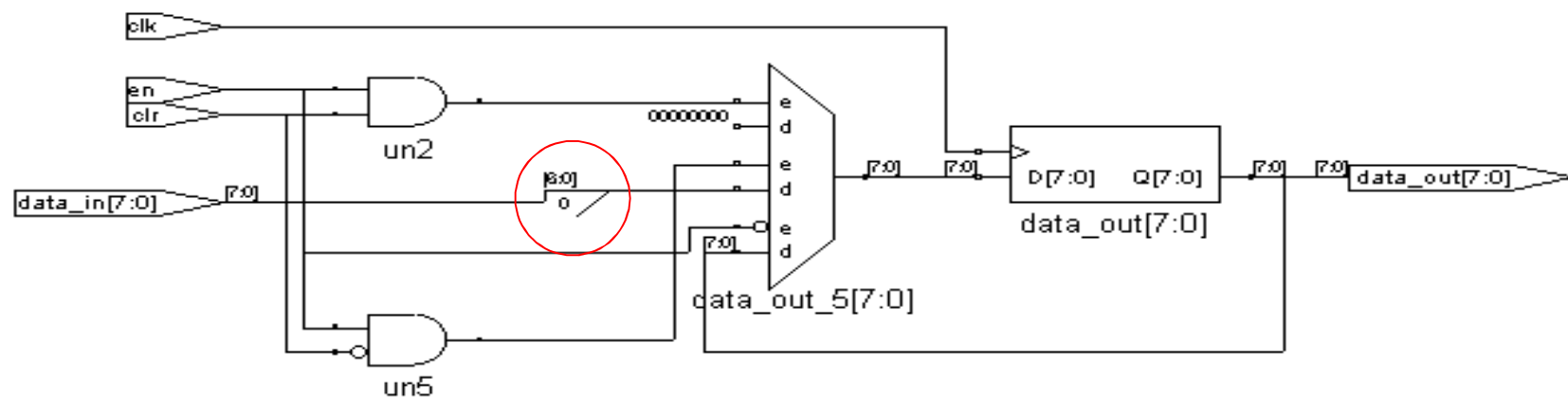


# 移位寄存器（Con't）

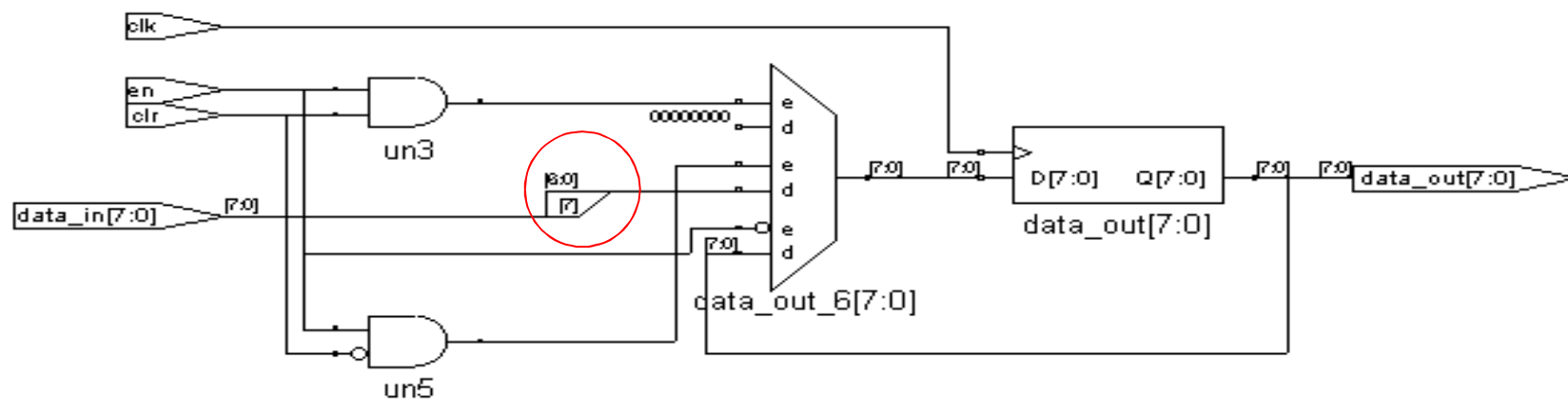
循环左移寄存器的源码：

```
module      shift_left( clk, en, clr, data_in, data_out );
  input     clk,en,clr;
  input     [7:0] data_in;
  output    [7:0] data_out;
  wire      [7:0] data_in;
  reg       [7:0] data_out;
  always @ ( posedge clk )
    if ( en )
      if ( clr )
        data_out[7:0] = 8'b0;
      else
        begin
          data_out[7:1] = data_in [6:0];
          data_out[0] = data_in [7];
        end
    end
endmodule
```

# 综合结果



左移寄存器

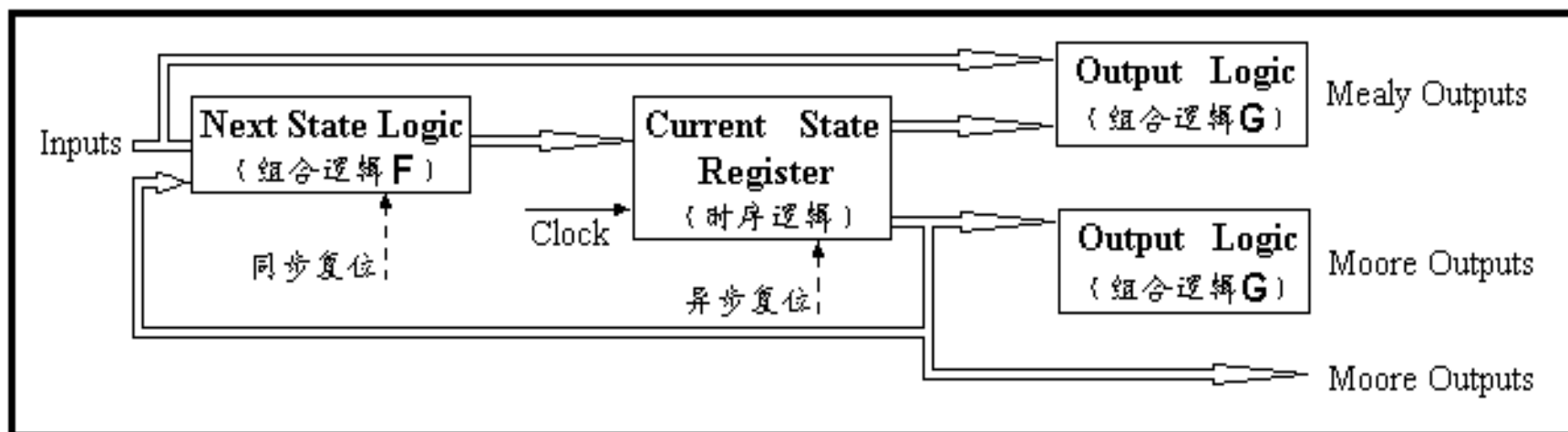


循环寄存器

# 同步有限状态机

有限状态机 (Finite State Machine ) 通常由时序电路和组合逻辑等硬件电路组成，按照预定的时序协调相关控制信号的动作，是完成特定操作序列的控制中心。

同步有限状态机由三部分组成，即当前状态(CS, Current State)、下一状态 (NS, Next State ) 和输出逻辑( OL, Output Logic)。当前状态由一个N位寄存器组成，状态是否改变、怎样改变取决于产生下一状态的组合逻辑F的输出，F是当前状态和输入信号的函数；输出逻辑G产生状态机的输出信号，由状态机的原始输入和当前状态决定。



# Mealy状态机和Moore状态机

根据状态机输出的确定方式可以将状态机分为Mealy型和Moore型。Mealy型状态机的输出不但取决于当前状态，还取决于各个外部输入值，而Moore型只取决于当前状态


$$NS = F(CS, input)$$

Mealy状态机

$$output = G(CS, input)$$

$$output = G(CS)$$

Moore状态机



## 举 例

动售饮料机的控制状态机，根据两种币值的投币信号指示售货机是否发货，以及是否找零。

此种售货机要求每次投币一枚，分为五角和一元两种，所以，输入信号five\_jiao和one\_yuan不会同时为1，只有三种组合会引起状态发生转移。饮料价格为2.5元，当已投入2.5元时，仍继续投币，则售一瓶饮料后转至FIVE或TEN状态；若已投入3元，则将找零的五角作为基数，状态转移至TEN或FIFTEEN，开始新的转移。

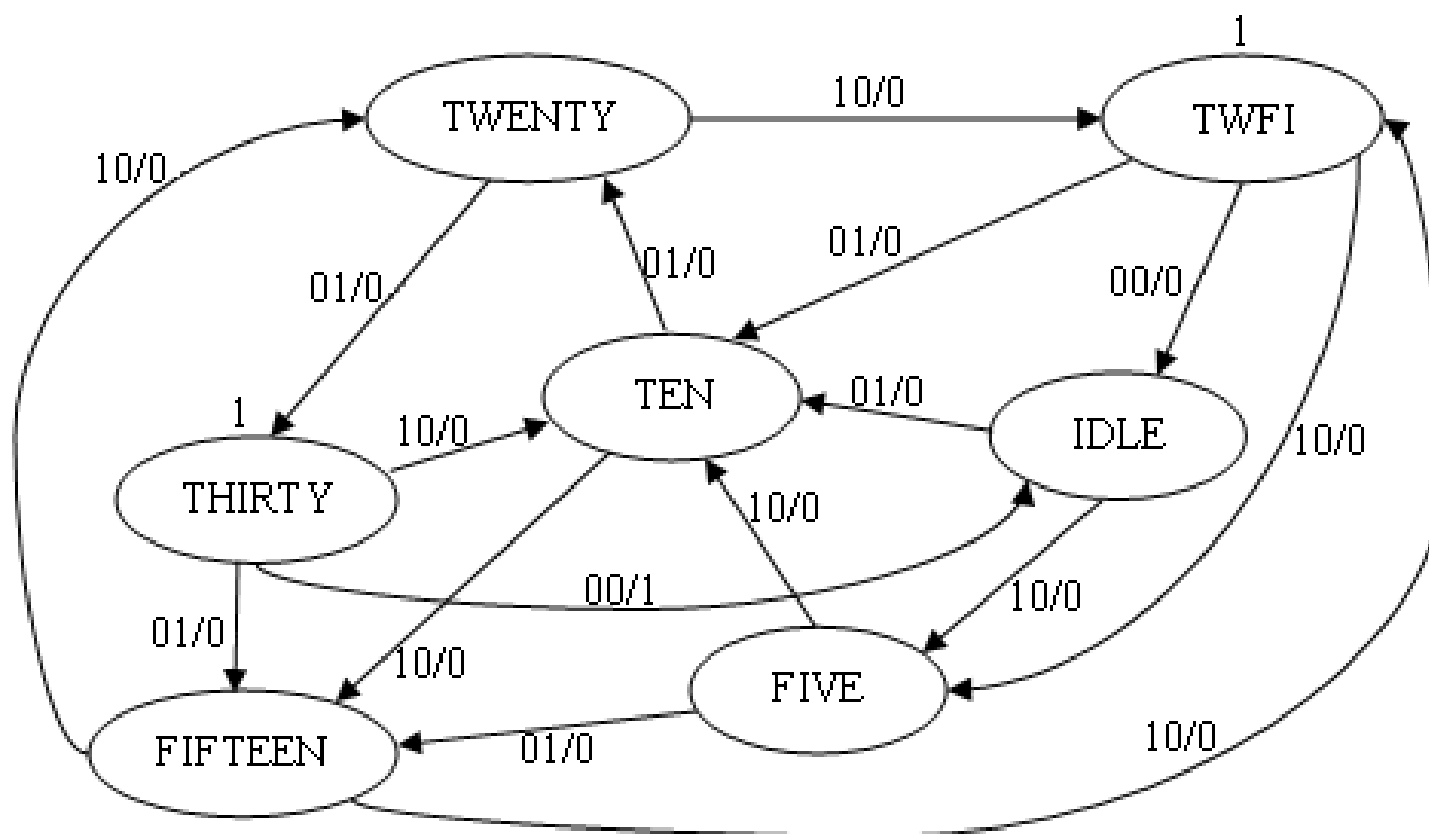
通常采用状态图和状态表直观地描述状态机的时序关系。

# 状态表

input		current state	Next State	output	
five_jiao	one_yuan			five_jiao_out	sell
1	0	IDLE	FIVE	0	0
0	1		TEN	0	0
0	0		IDLE	0	0
1	0	FIVE	TEN	0	0
0	1		FIFTEEN	0	0
0	0		FIVE	0	0
1	0	TEN	FIFTEEN	0	0
0	1		TWENTY	0	0
0	0		TEN	0	0
1	0	FIFTEEN	TWENTY	0	0
0	1		TWFI	0	0
0	0		FIFTEEN	0	0
1	0	TWENTY	TWFI	0	0
0	1		THIRTY	0	0
0	0		TWENTY	0	0
1	0	TWFI	FIVE	0	1
0	1		TEN	0	1
0	0		IDLE	0	1
1	0	THIRTY	TEN	0	1
0	1		FIFTEEN	0	1
0	0		IDLE	1	1

# 状态图

状态图中，小圆圈表示电路的各个状态，箭头表示每个时钟周期后状态转移的方向。此外，箭头旁注明了状态转换前的输入变量取值和输出值。通常将输入变量的取值写在斜线以上，**Mealy**型输出值写在斜线以下，**Moore**型输出标注在圆圈上方。





# 源 码

```
module auto_sell ( five_jiao,one_yuan, clk, reset, sell, five_jiao_out);  
    input five_jiao,one_yuan;  
    input clk, reset;  
    output sell, five_jiao_out;  
    reg sell, five_jiao_out;  
    reg [6:0] current_state;  
    reg [6:0] next_state;  
    `define IDLE      7'b00000001  
    `define FIVE      7'b00000010  
    `define TEN       7'b00000100  
    `define FIFTEEN   7'b00001000  
    `define TWENTY    7'b00010000  
    `define TWFI      7'b00100000    //TWENTY-FIVE  
    `define THIRTY    7'b01000000
```





```
always @ (posedge clk)
```

```
begin
```

```
    current_state = next_state;
```

```
end
```

```
always @ (current_state or reset or five_jiao or one_yuan)
```

```
begin
```

```
    if (!reset) begin
```

```
        next_state = `IDLE;
```

```
        five_jiao_out = 0; sell=0;
```

```
    end
```

```
    else
```

```
        case (current_state)
```

```
        `IDLE: begin
```

```
            five_jiao_out = 0;
```

```
            sell=0;
```

```
            if (five_jiao)
```

```
                next_state = `FIVE;
```

```
            else if (one_yuan)
```

```
                next_state = `TEN;
```

```
            else
```

```
                next_state = `IDLE;
```

```
        end
```

```
        `FIVE: begin
```

```
            five_jiao_out = 0;
```

```
            sell=0;
```

```
            if (five_jiao)
```

```
                next_state = `TEN;
```

```
            else if (one_yuan)
```

```
                next_state = `FIFTEEN;
```

```
            else
```

```
                next_state = `FIVE;
```

```
        end
```




```
`TEN: begin
five_jiao_out = 0;
if (five_jiao)
else if (one_yuan)
else
end
`FIFTEEN: begin
five_jiao_out = 0;
if (five_jiao)
else if (one_yuan)
else
end
`TWENTY: begin
five_jiao_out = 0;
if (five_jiao)
else if (one_yuan)
else
end
`TWFI: begin
sell=1;
if (five_jiao)
else if ( one_yuan )
else
end
sell=0;
next_state = `FIFTEEN;
next_state = `TWENTY;
next_state = `TEN;

sell=0;
next_state = `TWENTY;
next_state = `TWFI;
next_state = `FIFTEEN;

sell=0;
next_state = `TWFI;
next_state = `THIRTY;
next_state = `TWENTY;

five_jiao_out=0;
next_state=`FIVE;
next_state = `TEN;
next_state=`IDLE;
```



```
`THIRTY: begin
    sell=1;
    if (five_jiao) begin
        next_state=`TEN;
        five_jiao_out=0;
    end
    else if ( one_yuan ) begin
        next_state = `FIFTEEN;
        five_jiao_out = 0;
    end
    else begin
        next_state=`IDLE;
        five_jiao_out=1;
    end
end
default: begin
    next_state=`IDLE;
    sell=0;
    five_jiao_out=0;
end
endcase
end
endmodule
```



# 状态编码

状态编码的方式主要有：

## ■ 二进制编码（Binary）

- 采用普通的二进制数代表每个状态。
- 缺点：从一个状态转换到相邻状态时，有可能有多个bit发生变化，瞬变次数多，容易产生毛刺，引起逻辑错误。

## ■ 格雷编码（Gray Code）

- 格雷码是一种在相邻计数值之间只有一位发生变化的编码方式。
- 优点：格雷编码节省逻辑单元，而且在状态的顺序转换中，相邻状态每次只有一个bit产生变化，减少了瞬变的次数，也减少了产生毛刺和一些暂态的可能。

## ■ 一位热编码（One-hot）

- one-hot编码即采用n位来编码具有n个状态的状态机。
- 采用one-hot编码，虽然多用了触发器，但可以有效节省和简化组合电路。对于寄存器数量多，而门逻辑相对缺乏的FPGA器件来说，采用这种编码方式可以有效提高电路的速度和可靠性，也有利于提高器件资源的利用率。



## 3种编码方式对比

状态	二进制编码	格雷编码	一位热码
state0	000	000	00000001
state1	001	001	00000010
state2	010	011	00000100
state3	011	010	00001000
state4	100	110	00010000
state5	101	111	00100000
state6	110	101	01000000
state7	111	100	10000000



# 状态编码的定义

有两种方式可用于定义状态编码：parameter和`define

parameter state0=2'b00,	`define state0 2'b00
state1=2'b01,	`define state1 2'b01
state2=2'b11,	`define state2 2'b11
state3=2'b10;	`define state3 2'b10
.....	.....
case (state)	case (state)
state0: ...;	`state0: ...;
state1: ...;	`state1: ...;
.....	.....



# 状态机的描述方法

在Verilog HDL中可以用许多种方法来描述有限状态机，最常用的方法是用**always**块语句和**case**语句。

有限状态机由当前状态、下一状态和输出逻辑三部分组成，可以依据状态机的不同结构采用不同的Verilog描述方法，如：

1. 将 CS、NS与OL 分别描述；
2. 将 CS、NS与OL 混合描述；
3. 将 NS 与 OL 混合，CS 单独描述；
4. 将 CS 与 NS 混合，OL单独描述；
5. 将 CS 与 OL 混合， NS单独描述。



# 有限状态机设计步骤

有限状态机设计的一般步骤：

- (1) 逻辑抽象，得出状态转换图（表）
- (2) 状态化简
- (3) 状态分配
- (4) 选定触发器类型并求出状态方程、驱动方程和输出方程
- (5) 按照方程得出逻辑图


用Verilog HDL来描述有限状态机，可以充分发挥硬件描述语言的抽象建模能力，使用**always**块语句和**case (if)**等语句及赋值语句即可方便实现，具体的逻辑化简、逻辑电路和触发器映射均可由计算机自动完成，上述设计步骤中的第**2、4、5**步不再需要很多的人为干预，使电路设计工作得到简化，效率也有很大提高。






# 总 结

- Verilog语法仅仅是在设计中需要遵循的规则，掌握这些规则并不意味着能够设计出好的电路来，因此，需要根据“好”电路的标准来设计代码。
- 评价指标：
  - 面积，时序，功耗等；
  - 可读性，可靠性，可移植性，可扩展性等；
  - 与EDA相关部分：可综合性，是否为同步电路等。




# 编写代码的一些建议（1）

- 为了设计维护的方便，设计文档要齐全，设计中应当有充分的注释。
- 一行不要太长。
- 标识符要取有意义的名字，命名风格要一致
- 复位、时钟等全局信号确定统一的名称、同步/异步、上升/下降边有效、高/低电平有效等。
- 其他低有效信号使用统一的后缀（定义一套命名规则）。
- 在代码中应当使用缩入格式，将同类的语句对齐，代码中相应的部分对齐。
- 设计中可以利用参数定义提高可读性和可维护性。
- 尽量不使用**disable**等非结构化语句



## 编写代码的一些建议（2）

- 所有的设计中最好确定一个固定的**timescale**。
- 尽量使用同步时序逻辑，对无法避免的异步逻辑须特别注意与同步逻辑之间的隔离和信号同步问题。
- 设计中不应出现任何形式的组合逻辑反馈。
- 设计中应当简化逻辑，同时不要引入无用的信号。
- 在设计中不使用**initial**，如需要初值用复位逻辑。



## 编写代码的一些建议（3）

- 组合块的敏感表中不应当用边沿敏感信号，敏感表应当完整。时序块的敏感表中只应当是时钟和异步置、复位。敏感表中不应当出现过多的边沿敏感信号。分清置、复位异步控制信号的作用。
- 设计中一个信号不应当在多个**always**中赋值。
- 不相关的信号不要在一个过程块中赋值
- 条件判断不要重复，条件分支应当完备。
- 对于**case**语句，尽量列出所有的组合情况，对于不会出现的组合情况，用**default**进行说明，其中的逻辑值可以取 x，也可以取已有的分支值。一般取 x 较好，取已有的分支值可能带来冗余逻辑。



## 编写代码的一些建议（4）

- 尽量不用顶层的逻辑，这些逻辑应当划分在调用的模块中实现。
- 一个过程块中最好不要同时使用阻塞赋值和非阻塞赋值。
- 组合逻辑中没有必要用非阻塞赋值。组合逻辑中用阻塞赋值就可以，时序逻辑中用非阻塞赋值比较好。
- 注意向量的宽度，在对向量赋值时也应当指明数值的宽度。