



# 数字系统课程设计

东南大学信息科学与工程学院

2023年

# 硬件描述语言语法与使用

- ◆ 硬件描述语言
- ◆ Verilog 简单介绍
- ◆ Verilog 设计FPGA的方法
- ◆ Verilog 语法学习



# 硬件描述语言（HDL）

## ■ HDL（Hardware Description Language）

### ■ Verilog

- Verilog是由Gateway设计自动化公司的工程师于1983年末创立的。
- Verilog成为了IEEE 1364-1995标准，即通常所说的Verilog-95。
- Verilog-2001是对Verilog-95的一个重大改进版本，目前是Verilog的最主流版本，被大多数商业电子设计自动化软件包支持。
- Verilog与C语言在语法上有相似之处，因此具有C语言基础的设计人员更容易掌握它



# 硬件描述语言（HDL）

## Verilog与VHDL比较

- Verilog与VHDL是目前最常用的硬件描述语言，都是IEEE标准，他们有以下几点不同：
  - 从推出过程来看，VHDL偏重于标准化的考虑，而Verilog与EDA设计软件结合更加紧密。Verilog语言标准化晚于VHDL，VHDL1987年进入IEEE标准，Verilog则在1995年才进入IEEE；
  - 与VHDL相比，Verilog语言的设计风格更加简明了、高效便捷。如果单纯从程序长短上考察，Verilog代码相对较短。VHDL风格像PASCAL语言，语法呆板，容易学习；Verilog则像C语言，语法灵活，但是初学者容易出错；
  - 在PLD设计领域，Verilog与VHDL语言在市场占有率上几乎平分秋色，而在ASIC领域，Verilog占有绝大部分市场。



# Verilog简介

- ◆ 1983年由Gateway Design Automation(GDA)公司 Philip Moorby为其模拟器产品开发的硬件建模语言
- ◆ 1987年Synonsys公司开始使用Verilog HDL行为语言作为综合工具的输入。
- ◆ 1990年，Cadence公司成立OVI(Open Verilog International)组织来负责推广Verilog
- ◆ 1995年，IEEE制定了Verilog HDL标准，即IEEE Std 1364 - 1995
- ◆ 2000年，IEEE公布的Verilog 2001标准，其大幅度地提高了系统级和可综合性能。



# Verilog设计FPGA的基本方法

- ◆1.文本编辑:用任何文本编辑器都可以进行,也可以用专用的EDA编辑环境Verilog文件保存为.v文件.
- ◆2.逻辑综合与布局布线:使用综合器和布局布线工具对文件进行处理.
- ◆3.仿真:利用仿真软件验证电路的功能和时序.
- ◆4.编程下载: 确认仿真无误后,将文件下载到芯片中.



# Verilog的基本结构-模块 (module)

- ◆ 模块是Verilog的基本描述单位，用于描述某个设计的功能或结构及与其它模块通信的外部接口。单个模块就可以实现VHDL中实体的定义和结构体的描述。

- ◆ 模块声明

**module** <模块名> (模块端口列表); 例:

<模块内容>

**endmodule**

- ◆ 端口列表

- 列出所有端口名称

- ◆ 端口声明

<端口类型> <端口名>;

- ◆ 端口类型

- input → 输入端口
- output → 输出端口
- inout → 双向端口

**module** HalfAdder (A, B, Sum, Carry);

**input** A, B;

**output** Sum, Carry;

**wire** A, B;

**wire** Sum, Carry;

**assign** Sum=A^B;

**assign** Carry=A&B;

**endmodule**



# Verilog模块的端口信号名和端口模式

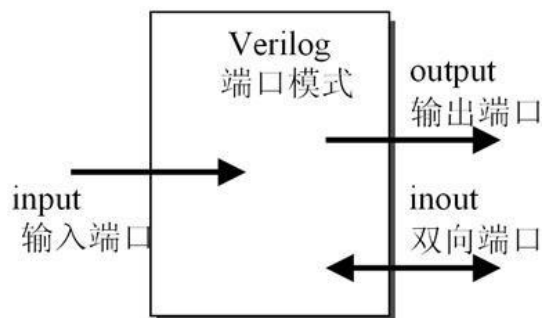
## 端口定义语句的一般格式

input 端口名 1, 端口名 2, ... ;

output 端口名 1, 端口名 2, ... ;

inout 端口名 1, 端口名 2, ... ;

input [msb : lsb] 端口名 1, 端口名 2, ... ;





# Verilog的基本结构-模块 (module)

- ◆ Verilog模块结构完全嵌在module和endmodule声明语句之间
- ◆ 每个Verilog程序包括四个主要部分：端口定义、I/O说明、内部信号声明、功能定义。
- ◆ 模块中，可以采用下述方式描述设计：
  - 结构方式
  - 数据流方式
  - 行为方式
  - 上述方式的混合



# Verilog结构化描述方式

- ◆通过调用逻辑元件、描述它们之间的连接来建立逻辑电路的Verilog模型；
  - 逻辑元件：Verilog内置的逻辑门、自主开发的已有模块，商业IP模块；
- ◆可使用
  - 内置门：not, and, or...
  - 开关级：nmos, cmos, tran...
  - 用户自定义的结构
  - 模块实例：其它module单元



# Verilog结构化描述方式

## ◆门级建模

- 利用Verilog内置的基本门级元件以及它们之间的连接来构筑逻辑电路的模型；

- `module` 模块名（端口列表）；

端口定义：

`input`

`output`

数据类型说明：

`wire`

门级建模描述：

`and` （输出，输入.....）；

`not` （输出，输入.....）；

.....

`endmodule`

例子：

```
module ex_and(a,b,c);  
input a,b;  
output c;  
and (c,a,b);  
endmodule
```



# Verilog数据流描述方式

- ◆ 根据信号(变量)之间的逻辑关系，采用连续赋值语句，描述逻辑电路的方式；

- 使用连续赋值语句

```
assign [delay] LHS_net=RHS_expression;
```

**assign 线网信号名(wire类型)=运算表达式(操作数+操作符);**

- module 模块名（端口列表）；

端口定义：

input

output

数据类型说明：

wire

逻辑功能定义：

assign（逻辑功能表达式1）；

.....

assign（逻辑功能表达式n）；

endmodule

assign语句并发执行，与位置无关

例子：

```
module HalfAdder(A, B, Sum, Carry);
```

```
input A, B;
```

```
output Sum, Carry;
```

```
assign Sum=A^B;(异或)
```

```
assign Carry=A&B;
```

```
endmodule
```



# Verilog数据流描述方式

- ◆ 操作数(19种数据类型)，常用的以下四种：
  - ◆ parameter型：用于参数的描述；
  - ◆ wire型：用于描述线网；
  - ◆ reg型：用于描述寄存器；
  - ◆ integer型：用于描述整数类型；
- ◆ 数字：基本格式： <位宽>'<进制><数值>  
举例：2'b01;4'd11;6'o37;8'haf等  
-4'd6 存为-6的补码



# Verilog数据流描述方式

- ◆ parameter型：一般定义在模块内，位置和端口声明所处级别相同；
- ◆ 格式：  
parameter 参数名1=表达式1， 参数名2=表达式2；  
parameter size=8；  
parameter width=size-1；
- ◆ 注意：
  - (1) 定义在模块内，位置和端口声明所处级别相同；
  - (2) 作用范围仅在此模块内以及实例化之后的本模块；
  - (3) 在模块实例化的过程中可以用defparam对参数进行更改；



# Verilog数据流描述方式

◆ wire型：用于描述模块中可能是连线的情况；输入输出端口默认为wire型；相当于硬件电路里的物理连接，特点是输出值紧跟输入值变化

◆ 格式：

wire [宽度声明] 线网名1, 线网名2;

wire x;

wire [3:0] y,m;

◆ 注意：

宽度声明部分，建议使用[n-1:0]的范围来定义宽度



# Verilog数据流描述方式

◆ reg型：用于描述触发器存储值的性质；主要用在verilog的行为级建模中；具有保持作用；

◆ 格式：

(1)表示一位或多位的寄存器：

```
reg x;
```

```
reg [3:0] x;
```

(2)定义存储器：

```
reg [n-1:0] 存储器名称 [0:m-1];
```

```
reg [7:0] mema [0:255];
```

◆ 注意：

如果定义的是存储器，需要对每个存储单元进行初始化  
整体赋值；



不能



# Verilog数据流描述方式

◆integer型：一种特殊的数据类型；

◆格式：

```
integer i;
```

```
i=1;
```

注意区别于reg型：reg中存储的数据为无符号数，integer中存储的数据是有符号数；其宽度一般默认为32位；



# 标识符规则

## 合法

Decoder\_1, FFT, Sig\_N, Not\_Ack, State0, \_Decoder\_, REG

## 不合法

2FFT	// 起始为数字
Sig_#N	// 符号“#”不能成为标识符的构成
Not-Ack	// 符号“-”不能成为标识符的构成
data__BUS	// 标识符中不能有双下划线
reg	// 关键词
ADDER*	// 标识符中不允许包含字符*



# Verilog的基本操作符

## Bitwise operators

~ NOT  
& AND  
| OR  
^ EXOR

## More Operators:

>> Shift right  
<< Shift left  
+ Add  
- Subtract  
\* Multiply  
/ Divide  
% Modulus

## Relational Operators:

== Equal to  
!= Not equal  
< Less than  
> Greater than  
<= Less than or equal  
>= Greater than or equal  
&& AND  
|| OR

拼接: { }

```
reg [1:0] a,b;  
a=2'b00,b=2"11;  
x1={a,b}; //0011
```

条件: ?:

```
assign outa=s?in1:in2;
```



# 运算符 (Operators)

## ◆ 位拼接和复制运算 { }

```
wire a1;  
wire [3:0] a4;  
wire [7:0] b8, c8, d8;  
...  
assign b8 = {a4, a4};  
assign c8 = {a1, a1, a4, 2'b00};  
assign d8 = {b8[3:0], c8[3:0]};
```

位拼接运算 { }，通过组合元素和小数组构成大数组。

位拼接运算的实施涉及输入和输出信号的重新连接并且只需要“连线”。

位拼接运算 N{ }，复制封闭字符串，其中复制常数 N 指定复制的次数，例如，{4{2'b01}} 返回 8'b010101，如算术移位运算可以简化为：

```
assign sha = {3{a[8]}, a[8:3]};
```



# 运算符 (Operators)

## ◆ 条件运算

?:

```
[signal] = [Boolean-exp]  
?[true-exp]:[false-exp]
```

```
assign max = (a>b) ? a :  
b;
```

[boolean\_exp]是一个布尔表达式，结果返回真（1'b1）或假（1'b0），当为真时，[true\_exp]赋给[signal],为假则将[false\_exp]赋给[signal]。

条件运算可以看成if-else语句的简化：

```
i f [boolean-exp] then  
    [signal] = [true-exp] ;  
e l s e  
    [signal] = [false-exp];
```

条件运算可以进行级联和嵌套来指定所需选择，例求较大值电路返回a,b和c的最大值：

```
assign max = (a>b) ? ((a>c) ? a : c) : ((b>c) ? b : c);
```



# Verilog行为描述方式

- ◆ 结构描述侧重表示一个电路由哪些基本元件组成，以及这些基本元件的相互连接关系；
- ◆ 数据流描述侧重于逻辑表达式以及verilog中运算符的灵活运用；
- ◆ 行为描述关注逻辑电路的输入、输出的因果关系(行为特性)，即在何种输入条件下，产生何种输出(操作)，并不关心电路的内部结构，**EDA**综合工具能自动将行为描述转换成电路结构，形成网表文件。



# Verilog行为描述方式

## ◆ 行为级建模中的两种结构：

- initial语句：只执行一次（一般用于仿真测试）

```
initial    begin
```

```
.....
```

```
end
```

- always语句：循环重复执行

```
always @(事件控制列表)
```

```
begin
```

```
.....
```

```
end
```

- 二者都可在**module**中出现多次，执行顺序是同时执行；
- 二者不支持嵌套使用



# Verilog行为描述方式

## ◆设计模型:

- module 模块名(端口列表);

端口定义

input 输入端口;

output 输出端口;

数据类型说明

reg

parameter

逻辑功能定义

always @ (敏感事件列表)

begin

阻塞、非阻塞、if-else、case、for等行为语句

end

endmodule





# always语句

- ◆ always语句：循环重复执行，每次的循环由敏感信号列表触发；

```
always @(时序控制方式)
```

```
begin
```

```
.....
```

```
end
```

时序控制方式分为3种：

(1)基于延迟的控制(一般用于仿真测试)

```
initial clock=0;
```

```
always #5 clock=~clock;
```




# always语句

## (2)基于电平敏感的控制(一般用于组合电路)

### ◆ 例子

```
module HalfAdder(A, B, Sum, Carry);  
    input  A, B;  
    output Sum, Carry;  
    always@(A or B) //当信号A或信号B的值发生改变时  
        begin  
            Sum=A^B;  
            Carry=A&B;  
        end  
endmodule
```

(\*)



# always语句

(3)基于边沿敏感的控制(一般用于时序电路)

posedge clock: 时钟上升沿到来时

negedge clock: 时钟下降沿到来时

◆ 例子:

```
always@(posedge clock)
begin
    if(! reset)
        q<=0;
    else
        q<=d;
end
```

**注意:**(1)一个verilog模块中可以由多个always进程, 它们是并行执行;

(2) 在每一个always语句中, 最好只使用一种类型的敏感信号列表;



# 块语句

- ◆ 顺序块和并行块
- ◆ 块语句作用：将两条或更多条过程语句组合在一起
- ◆ 分类：
  - `begin...end`：语句顺序执行；
  - `fork...join`：语句并行执行；  
(不被综合)



# if-else语句----语法

## □ If-else语句的简化语法如下：

```
if [表达式]
  begin
    [顺序执行语句] ;
    [顺序执行语句] ;
  end
else
  begin
    [顺序执行语句] ;
    [顺序执行语句] ;
  end
```

- [表达式]项一般为逻辑表达式或者关系表达式，也可以是一位变量。语句先对表达式判断，如果表达式为真，则执行下面分支的语句，否则执行else分支的语句。
- else分支具有选择性，可以省略。如果分支里只有一条语句，则定界符begin和end可以省略。



# if-else语句

## ◆if-else条件分支语句

- ◆ 1、if所接的条件判断中必须返回一个逻辑值；
- ◆ 2、如果待执行的语句有多条时，可以使用begin...end来进行封装；
- ◆ 3、if语句可以嵌套使用；
- ◆ 4、if语句是有优先级的；
- ◆ 5、语句结尾是“;”；
- ◆ 6、对于每一个出现的if语句，都应有一个else对应条件为假的情况；如添加空语句“else ;”



# if-else语句

## ◆ if-else条件分支语句应用举例

- module self\_from\_three(q,sela,selb,a,b,c);

input sela,selb,a,b,c;

output q;

reg q;

always@(\*)

begin

if(sela) q=a;

else if(selb) q=b;

else q=c;

end

endmodule

sela	selb	语句
0	0	q=c
0	1	q=b
1	0	q=a
1	1	q=a



排在前面的分支项操作具有最高优先级



# case语句

- case语句的简化语法如右所示：
- case语句是一条多路决策语句，其将**case[表达式]**和多个表达式**[分支项]**进行比较，程序跳入与当前**[表达式]**相等的**[分支项]**对应的分支执行。
- 最后一条分支为可选的，关键词是**default**，包含**[表达式]**未指定的所有值。如果一条分支中只有一条语句，则定界符**begin**和**end**可以省略。

```
case [表达式]
  [分支项1]:
    begin
      [顺序执行语句] ;
      . . .
    end
  [分支项2]:
    begin
      [顺序执行语句] ;
      . . .
    end
  [分支项3]:
    begin
      [顺序执行语句] ;
      ...
    end
  . . .
  default:
    begin
      [顺序执行语句];
    end
endcase
```





# case语句

◆ case全等比较分支控制语句（多分支）

■ case(表达式)

分支1: 语句1;

分支2: 语句2;

.....

default: 默认项;

endcase



# case语句

## ◆ case全等比较分支控制语句（多分支）

### ■ case语句实例

.....

```
case(op_code)
  2'b00:out=a|b;
  2'b01:out=a&b;
  2'b10:out=~(a&b);
  2'b11:out=a^b
```

default: out=0;

endcase

.....

控制表达式和分支表达式之间进行按位全等比较，必须每一位都相等。这种比较包含了信号的0、1、x、z四种状态



# case语句

◆ case分支控制语句有三种形式：（多分支）

- case: 全等比较分支控制
  - casex
  - casez
- } 局部比较分支控制

局部分支控制：

- casez: 认为表达式中的z值和?是无关值（即对应为无需匹配）；
- casex: 认为表达式中的z, x值和?为无关值；
- 由于z和x一般出现在仿真中，我们更倾向于采用？



# casex 和casez 语句

## □ 使用casez语句的优先编码器

```
module prio_encoder_casez( input  [3:0]  r,output  reg  [2:0]  y);  
    always @*  
        casez (r)  
            4'b1???: y = 3'b100;  
            4'b01??: y = 3'b011;  
            4'b001?: y = 3'b010;  
            4'b0001: y = 3'b001;  
            4'b0000: y = 3'b000;    // 这里可以使用default  
        endcase  
    endmodule
```



# case语句

## ◆case语句

- ◆ 1、case语句中每个分支条件必须不同；
- ◆ 2、如果待执行的语句有多条时，可以使用begin...end来进行封装；
- ◆ 3、case语句不需要break；
- ◆ 4、case语句在执行时被视为并行结构；
- ◆ 5、语句结尾是“;”；
- ◆ 6、case语句只能有一个default语句，必须使用，防止综合后生成锁存器；

## ◆循环语句(while、for、repeat.....)



# 过程赋值语句

◆过程赋值语句：在initial和always语句中对变量进行赋值的语句。

◆类型：

■ 阻塞赋值语句：

- ◆ "=";
- ◆ 按顺序执行，一条阻塞赋值语句执行结束后，才能继续执行下一条阻塞赋值语句；
- ◆ 语句执行结束后，左侧值会立刻改变；

■ 非阻塞赋值语句：

- ◆ "<=";
- ◆ 同一时间点，前面语句的赋值不能立刻被后面语句使用；
- ◆ 所有的赋值是在一个时间点结束的时候统一完成的；

原则：组合逻辑电路使用阻塞语句建模；时序逻辑电路使用非阻塞语句建模；



# 阻塞语句vs非阻塞语句

## ◆ 例1：非阻塞赋值

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

结果：**b**更新为**a**的值，**c**为上  
个时钟周期**b**的值。

## ◆ 例2：阻塞赋值

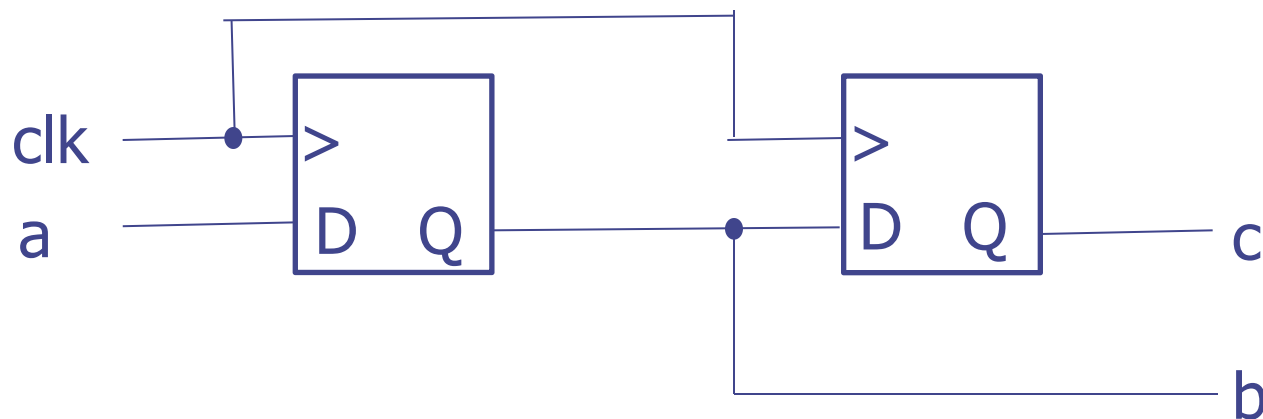
```
module block(clk,a,b,c);  
    input clk,a;  
    output b,c;  
    reg b,c;  
    always @(posedge clk)  
    begin  
        b=a;  
        c=b;  
    end;  
endmodule
```

结果：**b**、**c**更新为**a**的值。

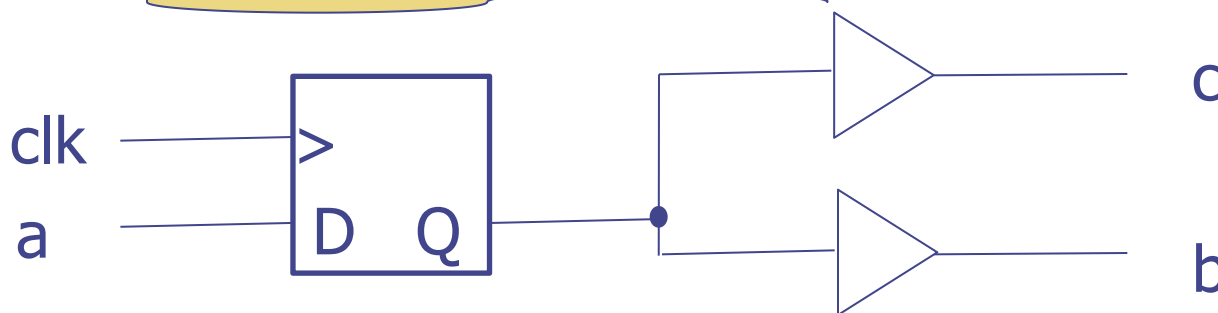


# 阻塞语句vs非阻塞语句

◆两个程序进行逻辑综合后的结果如下：



## 例1 非阻塞赋值的综合

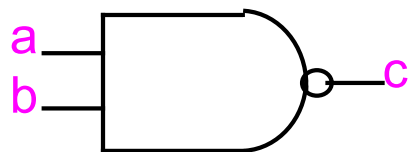


## 例2 阻塞赋值的综合





# Verilog组合逻辑-举例



```
module example1(c,a,b);  
  input a, b;  
  output c;
```

```
module example2 (a,b,c);  
  // Port modes  
  input a,b;  
  output c;
```

```
  // Functionality  
  assign c = ~(a & b);
```

```
endmodule
```

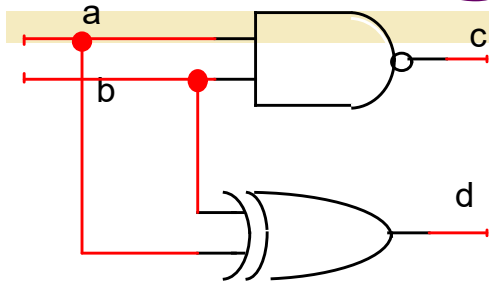
```
  // Registered identifiers  
  reg c;
```

```
  // Functionality  
  always @ (a or b)  
    c = ~(a & b);  
endmodule
```

Sensitivity list



# Verilog组合逻辑-举例



```
module example3(a,b,c,d);
```

```
// Port modes
```

```
input a, b;
```

```
output c;
```

```
output d;
```

```
// Registered identifiers
```

```
reg c,d;
```

```
// Functionality
```

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

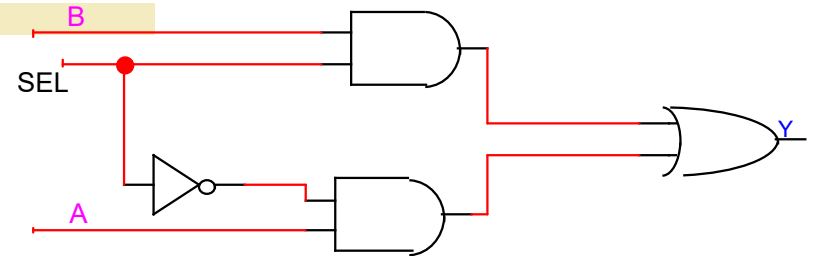
```
begin
```

```
c <= ~(a & b);
```

```
d <= a ^ b;
```

```
end
```

```
endmodule
```



```
module Mux2 (A, B, Sel, Y);
```

```
input A, B, Sel;
```

```
output Y;
```

```
reg Y;
```

```
// Functionality
```

```
always @ (A or B or Sel)
```

```
if (Sel==0)
```

```
assign c = ~(a & b);
```

```
assign d = a ^ b;
```

```
endmodule
```



# Verilog时序逻辑-举例

```
module D_FF (D, Clock, Q);
```

```
input D, Clock;
```

```
output Q;
```

```
// Registered identifiers
```

```
reg Q;
```

```
// Functionality
```

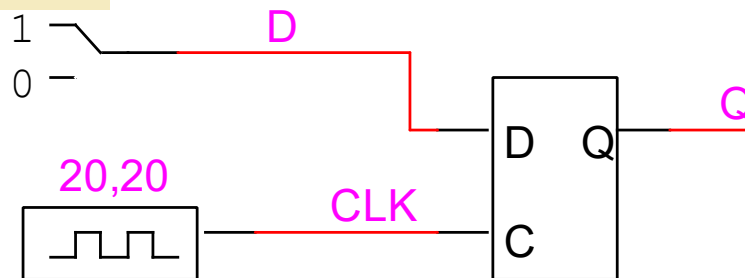
```
always @ (posedge Clock)
```

```
Q <= D;
```

```
endmodule
```



Sensitivity list



```
module D_FF
(D,Clock,Q_bar,Q,Reset);
input D,Clock,Reset;
output Q, Q_bar;
reg Q;
always @ (posedge Clock or
posedge Reset)
    if (Reset == 1)
        Q <= 0;
    else
        Q <= D;
    assign Q_bar = ~Q;
endmodule
```



# 状态机定义

有限状态机（Finite State Machine, FSM）简称状态机，是用来表示系统中的有限个状态及这些状态之间的转移和动作的模型。

这些转移和动作依赖于当前状态和外部输入，它下一步的状态逻辑通常是重新建立的，也称之为随机逻辑。



# 状态机设计

◆ 状态机是大型电子设计的基础,通常用来实现数字系统的控制器.

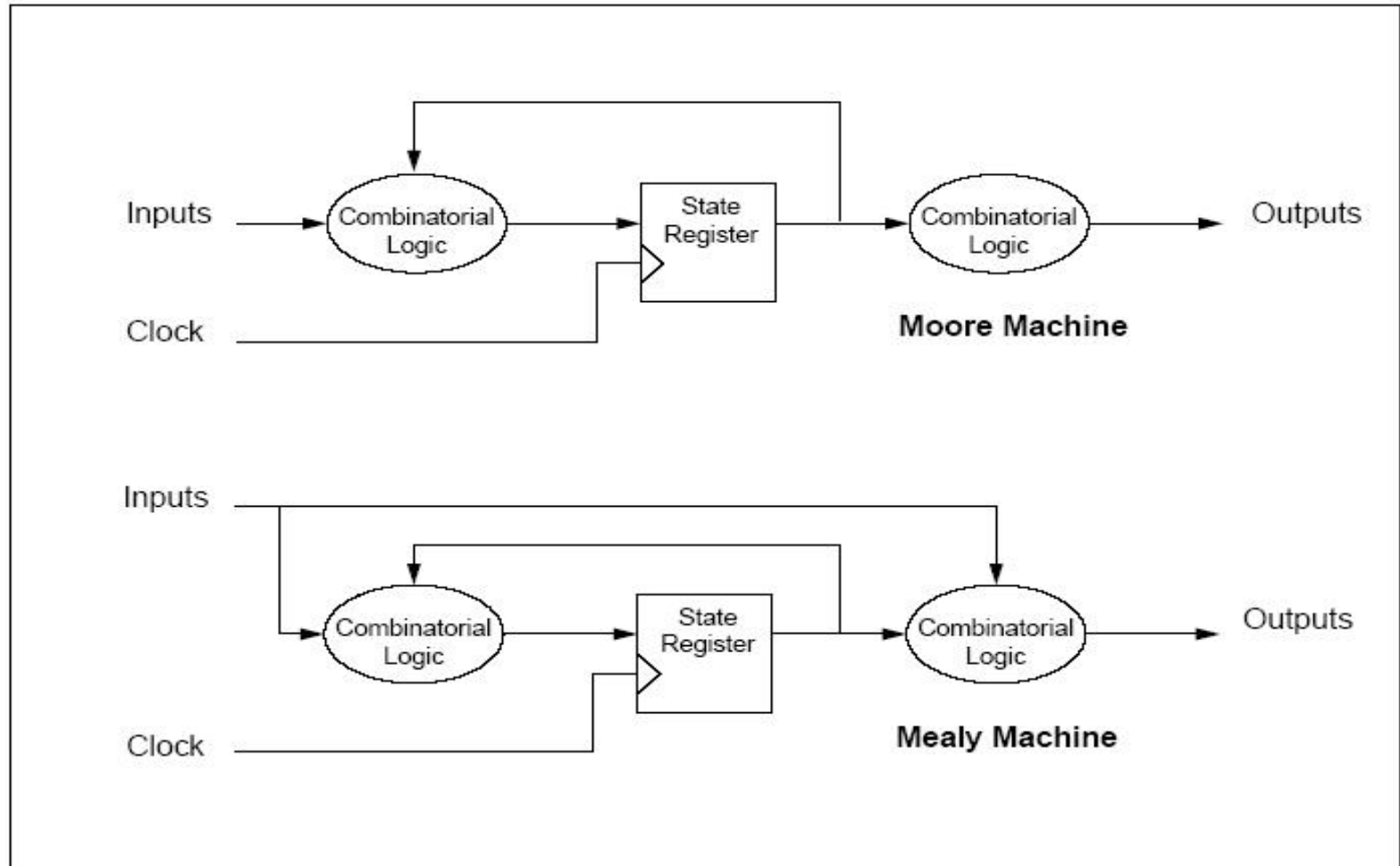
最基本的两种状态机方式:

**Moore型** 较简单的一种状态机,输出仅是当前状态的函数.

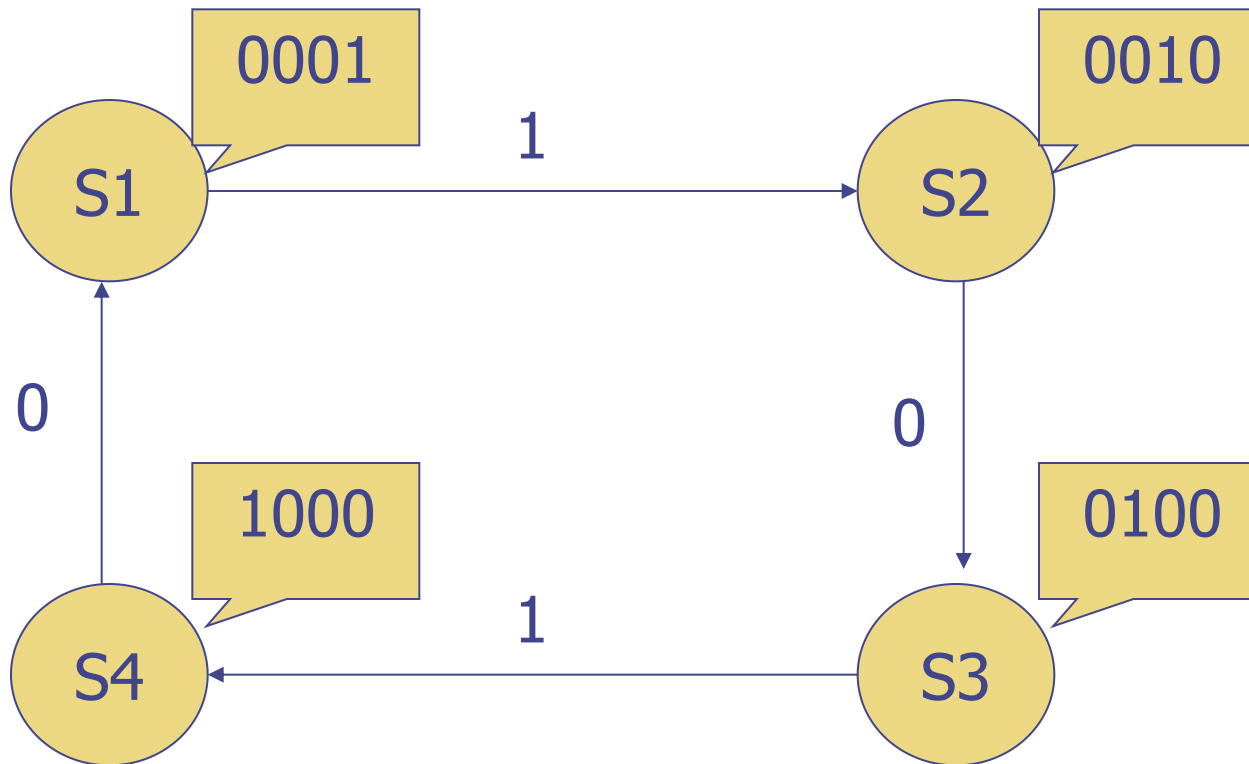
**Mealy型** 输出是当前状态和输入的函数.



# Moore VS Mealy



# Moore型状态机



# Verilog时序逻辑-Moore状态机举例

```
module fsm_moore(  
    input clk,  
    input datain,  
    input reset,  
    output [3:0] data_out  
);  
//参数声明  
parameter[1:0] s1=2'b00,  
                s2=2'b01,  
                s3=2'b10,  
                s4=2'b11;  
  
//内部信号声明  
reg[1:0] current_state;  
reg[1:0] next_state;  
reg[3:0] dataout;
```





# Verilog时序逻辑-Moore状态机举例

//状态寄存器

```
always @ (posedge clk or negedge reset) begin
    if(!reset)
        current_state <= s1;
    else
        current_state <= next_state;
end
```

//次态的组合逻辑

```
always @ (*) begin
    case(current_state)
        s1:begin
            if(datain==1'b1) next_state = s2;
            else next_state = s1;
        end
    endcase
end
```



# Verilog时序逻辑-Moore状态机举例

```
s2: begin
    if(datain==1'b0) next_state = s3;
    else    next_state = s2;
end
s3: begin
    if(datain==1'b1) next_state = s4;
    else    next_state = s3;
end
s4: begin
    if(datain==1'b0) next_state = s1;
    else    next_state = s4;
end
default:next_state =s1;
end case
end
```



# Verilog时序逻辑-Moore状态机举例

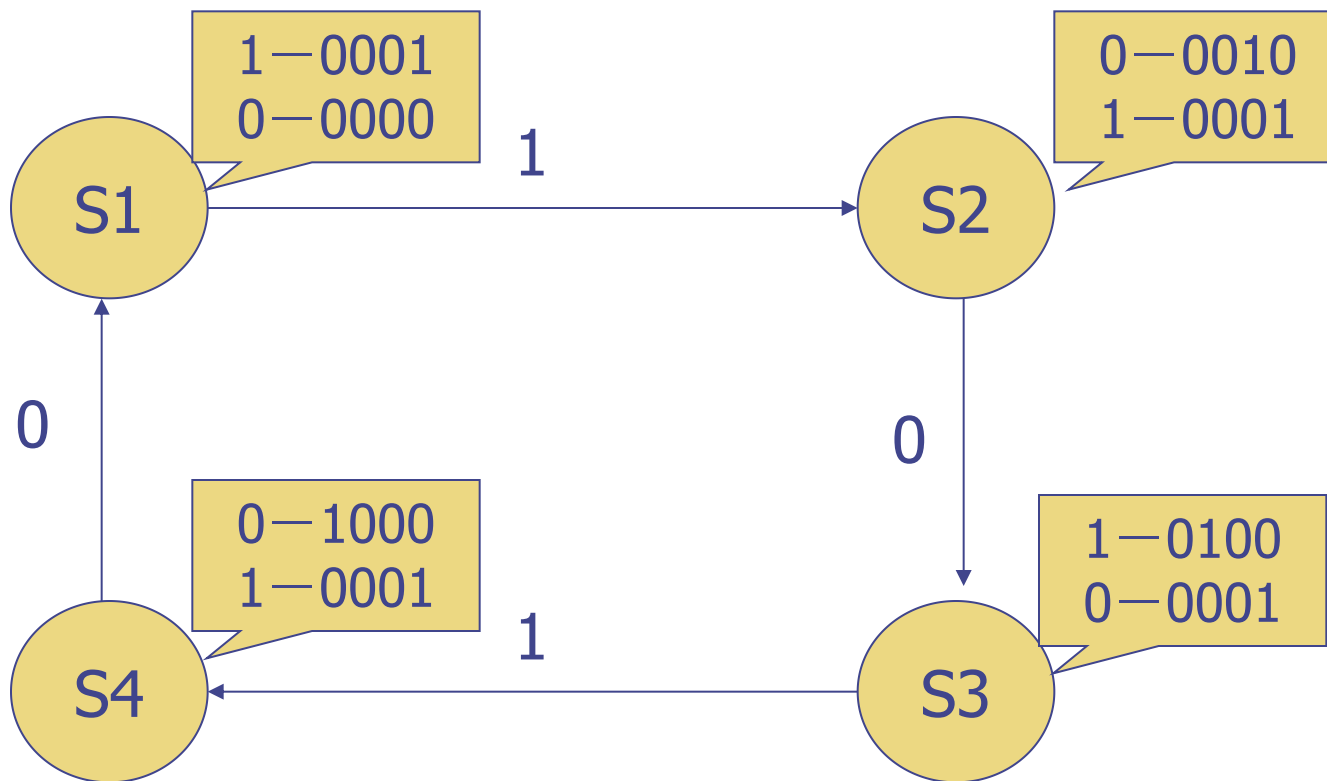
//输出逻辑

```
always @ (posedge clk) begin
    case(next_state)
        s1: dataout<= 4'b0001;
        s2: dataout<= 4'b0010;
        s3: dataout<= 4'b0100;
        s4: dataout<=4'b1000;
        default: dataout<= 4'b0000;
    endcase
end

assign data_out=dataout;
endmodule
```



# Mealy型状态机



# Verilog时序逻辑-Mealy状态机举例

```
module fsm_mealy(  
    input clk,  
    input datain,  
    input reset,  
    output [3:0] data_out  
);  
//参数声明  
parameter[1:0] s1=2'b00,  
                s2=2'b01,  
                s3=2'b10,  
                s4=2'b11;  
  
//内部信号声明  
reg[1:0] current_state;  
reg[1:0] next_state;  
reg[3:0] dataout;
```



# Verilog时序逻辑-Mealy状态机举例

//状态寄存器

```
always @ (posedge clk or negedge reset) begin
    if(!reset)
        current_state <= s1;
    else
        current_state <= next_state;
end
```

//次态的组合逻辑

```
always @ (*) begin
    case(current_state)
        s1:begin
            if(datain==1'b1) next_state = s2;
            else next_state = s1;
        end
    endcase
end
```



# Verilog时序逻辑-Mealy状态机举例

```
s2: begin
    if(datain==1'b0) next_state = s3;
    else    next_state = s2;
end
s3: begin
    if(datain==1'b1) next_state = s4;
    else    next_state = s3;
end
s4: begin
    if(datain==1'b0) next_state = s1;
    else    next_state = s4;
end
default:next_state =s1;
end case
end
```



# Verilog时序逻辑-Mealy状态机举例

//输出逻辑

```
always @ (posedge clk) begin
    case(next_state)
        s1: if(datain==1'b1)
                dataout<= 4'b0001;
            else dataout<=4'b0000;
        s2: if(datain==1'b0)
                dataout<= 4'b0010;
            else dataout<=4'b0001;
        s3: if(datain==1'b1)
                dataout<= 4'b0100;
            else dataout<=4'b0001;
        s4: if(datain==1'b0)
                dataout<= 4'b1000;
            else dataout<=4'b0001;
        default: dataout<= 4'b0000;
    endcase
endmodule
```





# Verilog时序逻辑-状态机设计模板

## Tips

```
module mod_name ( ... );
    input ... ;
    output ... ;

    parameter size = ... ;
    reg [size-1: 0] current_state;
    wire [size-1: 0] next_state;

    // State definitions
    parameter state_0 2'b00;
    parameter state_1 2'b01;

    always @ (current_state or the_inputs) begin
        // Decode for next_state with case or if statement
        // Use blocked assignments for all register transfers to ensure
        // no race conditions with synchronous assignments
    end

    always @ (negedge reset or posedge clk) begin
        if (!reset == 1'b0) current_state <= state_0;
        else
            current_state <= next_state;
    end

    //Output assignments
endmodule
```

Break FSMs into four blocks:

State definitions-Next state calculations (decoder)-Registers or flip-flops calculation-Output calculations (logic)

//state flip-flops  
reg [2:0] state, next\_state;

//1、state definitions

parameter S0=2'b00 S1=2'b 01, S2=2'b 10, S3=2'b11,...

// 2、State machine descriptions, next state calculations

always @(state or....)

begin case (state)

.....  
End

//3、register or flip-flop calculation

always@(posedge clk)

state<=next\_state

//4、Output calculations

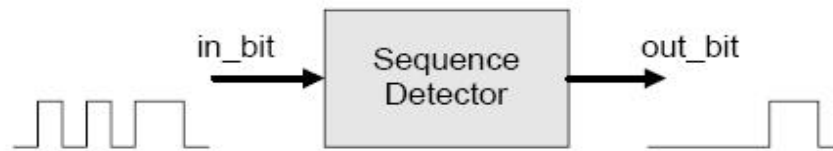
Output=f(state, inputs)

← Next State  
Logic

← State  
Register

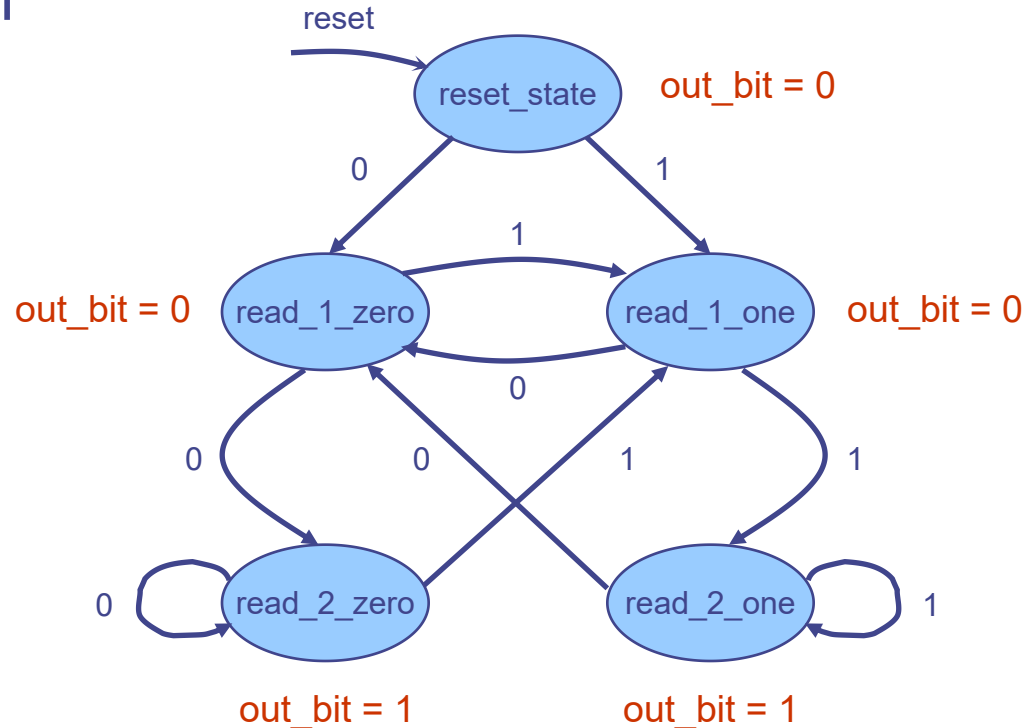


# Verilog时序逻辑-Moore状态机举例



Functionality: Detect two successive 0s or 1s in the serial input bit stream

**FSM  
Flow-Chart**



# Verilog时序逻辑-Moore状态机举例

```
module seq_detect (clock, reset, in_bit,
                  out_bit);
    input clock, reset, in_bit;
    output out_bit;

    reg [2:0] state_reg, next_state;

    // State declaration
    parameter reset_state = 3'b000;
    parameter read_1_zero = 3'b001;
    parameter read_1_one = 3'b010;
    parameter read_2_zero = 3'b011;
    parameter read_2_one = 3'b100;

    // state register
    always @ (posedge clock or posedge reset)
        if (reset == 1)
            state_reg <= reset_state;
        else
            state_reg <= next_state;

    // next-state logic
```

```
always @ (state_reg or in_bit)
    case (state_reg)
        reset_state:
            if (in_bit == 0)
                next_state =
read_1_zero;
            else if (in_bit == 1)
                next_state =
read_1_one;
            else next_state =
reset_state;
        read_1_zero:
            if (in_bit == 0)
                next_state =
read_2_zero;
            else if (in_bit == 1)
                next_state =
read_1_one;
            else next_state =
reset_state;
        else next_state =
reset_state;
```



# Verilog时序逻辑-Moore状态机举例

read\_2\_zero:

```
if (in_bit == 0)
    next_state = read_2_zero;
else if (in_bit == 1)
    next_state = read_1_one;
```

read\_1\_one:

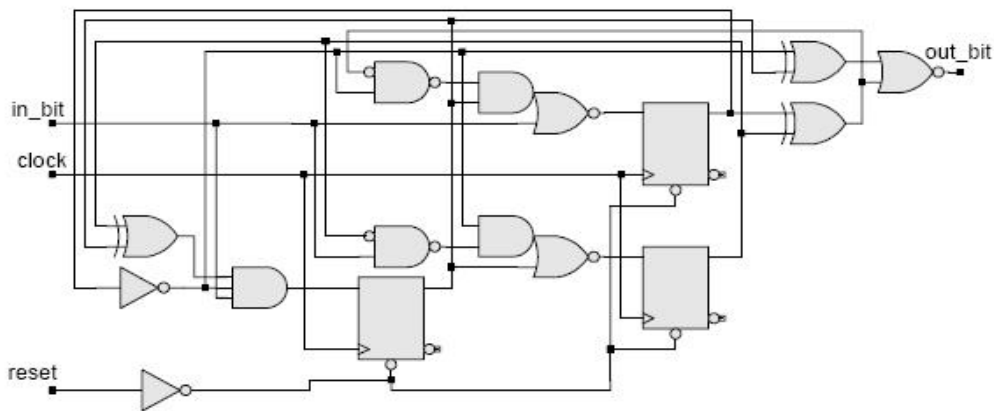
```
if (in_bit == 0)
    next_state = read_1_zero;
else if (in_bit == 1)
    next_state = read_2_one;
else next_state = reset_state;
```

read\_2\_one:

```
if (in_bit == 0)
    next_state = read_1_zero;
else if (in_bit == 1)
    next_state = read_2_one;
else next_state = reset_state;
default: next_state = reset_state;
```

```
endcase assign out_bit = ((state_reg == read_2_zero) || (state_reg ==
    read_2_one)) ? 1 : 0;
```

endmodule



# 课程内容

---

一、数字系统设计的一些概念和技巧

二、VHDL与Verilog语言

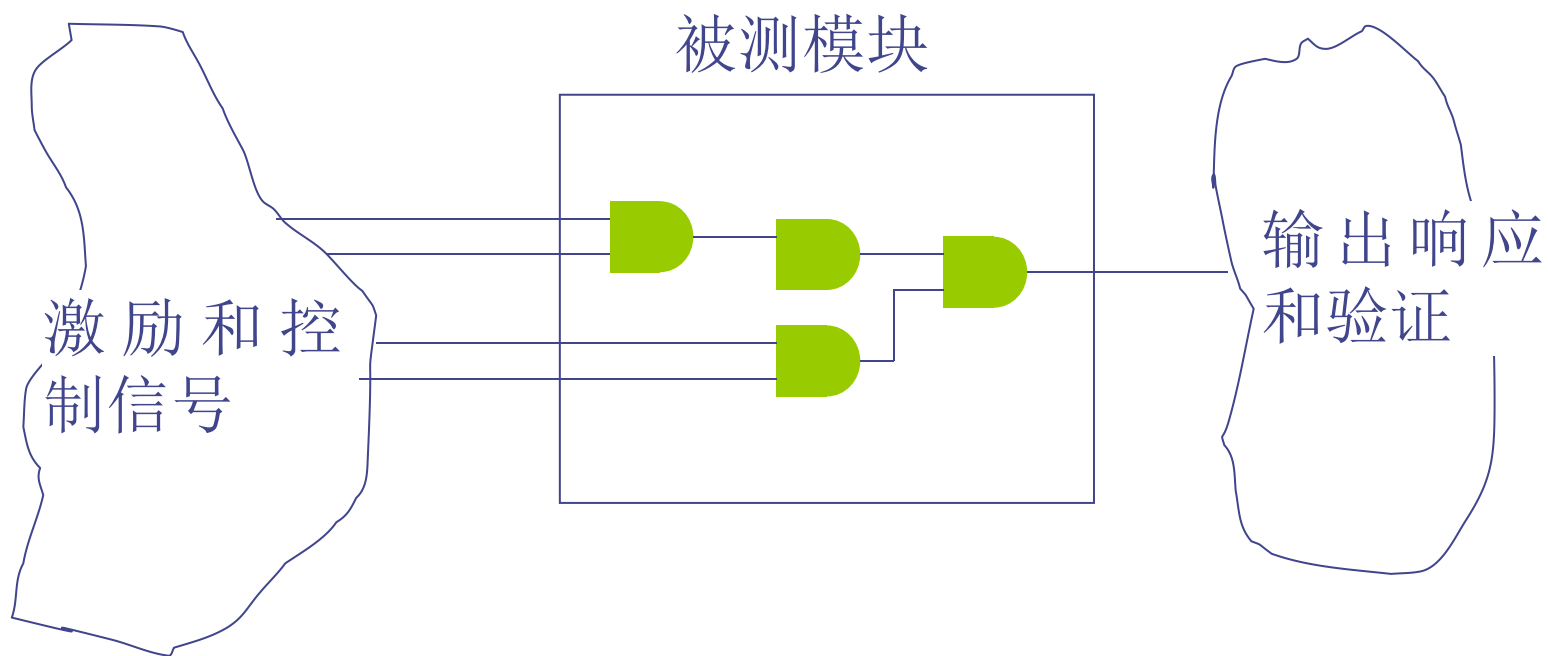
**三、硬件设计模块的仿真与测试**

四、软件开发环境VIVADO

五、硬件平台简介



# Verilog模块的测试



# Verilog模块的测试 - 测试文件结构

## ◆测试文件通常包含以下内容：

```
module Test_bench(); //通常无输入输出  
    信号或变量声明定义  
    使用 initial 或 always语句产生激励  
    例化待测试模块  
    监控和比较输出响应  
Endmodule
```



# Verilog模块的测试 – 结构举例

## 测试模块常见的形式:

```
module testfirstmodule;
    reg ...;          //被测模块输入
    wire...;          //被测模块输出

    initial
        begin
            ...;
            end ... //产生测试激励或控制信号

    always #delay
        begin
            ...;
            end ... //产生测试激励或控制信号

    ForTestingModule m(.in1(ina), .in2(inb), .out1(outa));
    //被测模块的实例引用

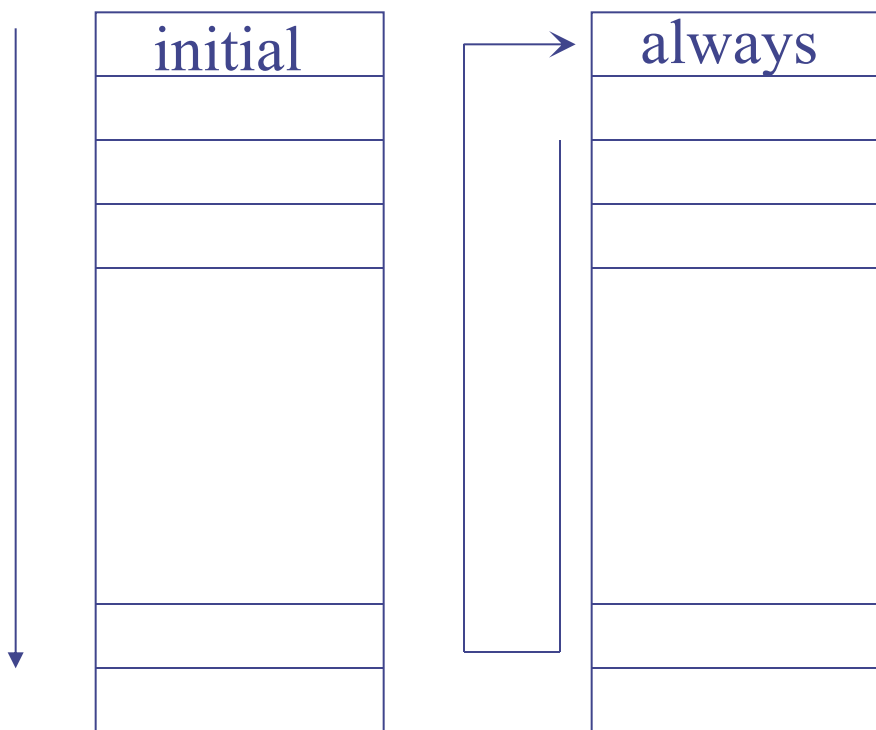
endmodule
```





# Verilog模块的测试

测试模块中常用的过程块：



所有的过程块都在0时刻同时启动；它们是并行的，在模块中不分前后。

- **initial**块只执行一次
- **always**块 只要符合触发条件可以循环执行



# Verilog模块的测试 –信号的初始化

- ◆ 代码中的变量的初始化可以用initial进行初始化，也可以在定义的时候进行初始化
- ◆ 在一定的触发条件下，例如复位中进行初始化
- ◆ 利用Verilog语言的读文件功能，从文本文件中读取数据（该数据可以通过C/C++、MATLAB等软件语言生成）



# Verilog模块的测试 - 激励源产生

## ■ 绝对时间:

```
initial begin
```

```
    Reset = 1; //仿真时间零点激励
```

```
    Load = 0; //仿真时间零点激励
```

```
    Count = 0; //仿真时间零点激励
```

```
    #100 Reset = 0;  
    //绝对时间100激励
```

```
    #20 Load = 1;  
    //绝对时间120激励  
    //相对上一个时间点20  
    #20 Count = 1;  
    //绝对时间140激励  
    //相对上一个时间点20
```

```
end
```

## ■ 相对时间:

```
always @ (posedge clock)
```

```
    Count <= Count + 1; //绝对时间的递增
```

```
initial begin
```

```
    if (Count <= 5) begin  
        Reset = 1; Load = 0;
```

```
    end
```

```
    else begin //触发事件，产生下列激励
```

```
        Reset = 0; Load = 1;  
    end
```

```
End
```

```
initial begin
```

```
    if (Count == 0110) begin  
        Load <= 0;  
        $display("Terminal.");  
    end
```

```
end
```



# Verilog模块的测试 – 测试时钟产生

## ◆ 基于initial 语句的方法:

```
parameter clk_period = 10;  
reg clk;  
initial begin  
    clk = 0;  
    forever  
        # (clk_period/2) clk = ~clk;  
End
```

## ◆ 基于always 语句的方法:

```
parameter clk_period = 10;  
reg clk;  
initial  
    clk = 0;  
always # (clk_period/2) clk = ~clk;
```



# Verilog模块的测试 - 复位产生

## ■ 异步复位:

```
parameter rst_repid =  
    100;  
reg rst_n;
```

```
initial  
begin  
    rst_n = 0;  
    # rst_repid;  
    rst_n = 1;
```

End

## ■ 同步复位:

```
parameter rst_repid =  
    100;  
reg rst_n;
```

```
initial  
begin  
    rst_n = 1;  
    @( posedge clk);  
    rst_n = 0;  
    # rst_repid;  
    @( posedge clk);  
    rst_n = 1;
```

end



# Verilog模块的测试 - 结果输出

## ◆ 关键词\$display和\$monitor实现结果的输出

在终端中打印信号的ASCII值

initial begin

```
$timeformat(-9,1,"ns",12); //设置输出时钟格式
```

```
$display(" Time Clk Rst Ld SftRg Data Sel"); //显示输入的字符串
```

```
$monitor("%t %b %b %b %b %b %b" , //设置输出信号格式  
$realtime, clock, reset, load, shiftreg, data, sel); //指定输出的信号
```

End

## ◆ 关键词\$stop停止仿真



# Verilog模块的测试 - 仿真时间与精度

- ◆ 关键字timescale定义测试文件的单位时间，和仿真的精度

`'timescale reference_time/precision`

其中，reference\_time是单位时间的度量，precision决定了仿真的推进延迟精度，同时也设置了仿真的推进步进单位。

例如

```
'timescale 1 ns / 1 ps           //度量参考为1ns，精度为1ps
#5 reset = 1; // 5个仿真时间延迟，相当于 $5 \times 1\text{ns} = 5\text{ns}$  的仿真时间
```

- ◆ 布局布线时将仿真的时延与和物理器件的时延相关联



# Verilog仿真文件设计示例

## ◆ 实例--带有异步复位、置数功能的十进制计数器

```
■ module counter_10(clk,reset,load,en,D,Q,C);  
    input clk,reset,load,en;  
    input [3:0] D;  
    output C;  
    output [3:0] Q;  
    reg [3:0] Q;
```

```
    always @(posedge clk or posedge reset)  
    begin  
        if(reset==1) Q<=4'd0;  
        else if(load==1) Q<=D;  
        else if(en) Q<=Q;  
        else if(Q==9) Q<=0;  
        else Q<=Q+1;  
    end
```

```
    assign C=(Q==9);  
endmodule
```





# Verilog仿真文件设计示例

## ◆ 实例--带有复位、置数功能的十进制计数器

- module counter\_10\_t();

- reg clk,reset,load,en;

- reg [3:0] D;

- wire C;

- wire [3:0] Q;

- parameter clk\_period=20;

- counter\_10 ins\_counter\_10(  
.clk(clk),  
.reset(reset),  
.load(load),  
.en(en),  
.D(D),  
.C(C),  
.Q(Q)  
);

initial

begin

reset=1;load=0;en=0;D=4'd9;clk=0;

#20 reset=0; //复位功能

#120 en=1; //暂停计数功能

#50 load=1; //置数功能

#20 en=0;

#20 load=0; //验证load和en的优先级

end

always #(clk\_period/2) clk=~clk;

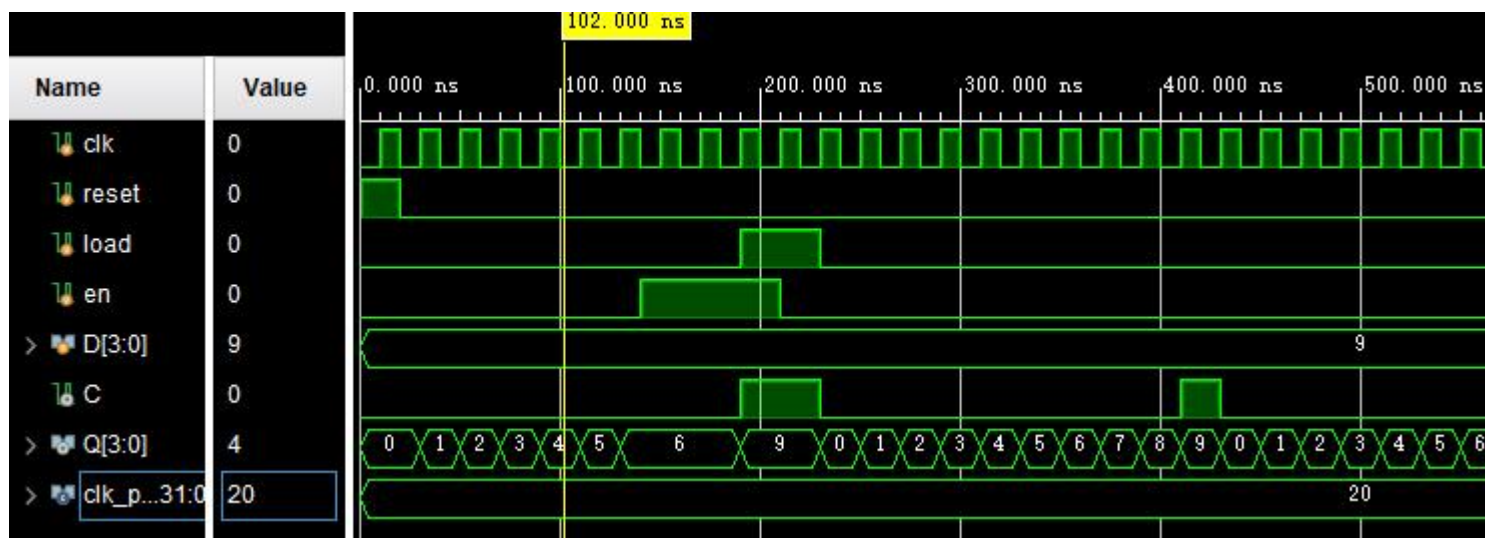
endmodule



# Verilog仿真文件设计示例

## ◆ 实例--帶有复位、置数功能的十进制计数器仿真波形

- reset复位功能;
- en暂停计数功能;
- load置数功能;
- load、en优先级;



# 可综合模型设计

- ◆ 要保证Verilog HDL赋值语句的可综合性，在建模时应注意以下要点：
- ◆ (1) 尽量使用同步方式设计电路。
- ◆ (2) 建议采用行为语句来完成设计。
- ◆ (3) 用**always**过程块描述组合逻辑，应在敏感信号列表中列出所有的输入信号。
- ◆ (4) 所有的内部寄存器都应该能够被复位，在使用FPGA实现设计时，应尽量使用器件的全局复位端作为系统总的复位。
- ◆ (5) 对时序逻辑描述和建模，应尽量使用非阻塞赋值方式。
- ◆ (6) 对组合逻辑描述和建模，应尽量使用阻塞赋值方式。
- ◆ (7) 但在同一个过程块中，最好不要同时用阻塞赋值和非阻塞赋值。
- ◆ (8) 不能在一个以上的**always**过程块中对同一个变量赋值。而对同一个赋值对象不能既使用阻塞式赋值，又使用非阻塞式赋值。
- ◆ (9) 如果不打算把变量推导成锁存器，那么必须在if语句或**case**语句的所有条件分支中都对变量明确地赋值。
- ◆ (10) 避免混合使用上升沿和下降沿触发的触发器。
- ◆ (11) 避免在**case**语句的分支项中使用x值或z值。

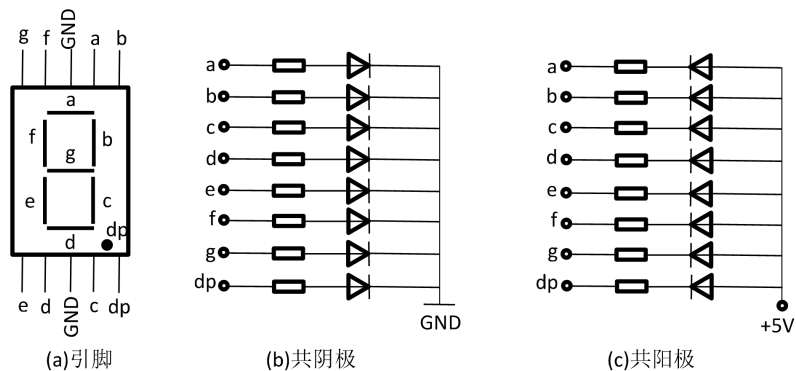


# 十六进制数七段LED显示译码器

□ 七段LED显示器也称为七段数码管，其示意图如下图

- 包括七个LED管和一个圆形LED小数点。
- 按LED单元连接方式可以分为**共阳数码管**和**共阴数码管**，共阳数码管是指将所有发光二极管的阳极接到一起形成公共阳极(COM)的数码管，COM接到逻辑高电平，当某一字段发光二极管的阴极为低电平时，相应字段就点亮。
- 共阴数码管是指将所有发光二极管的阴极接到一起形成公共阴极(COM)的数码管，COM接到逻辑低电平，当某一字段发光二极管的阳极为高电平时，相应字段就点亮。

七段码LED示意图



# 十六进制数七段LED显示译码器

十六进制数共阳七段LED显示译码器功能表

hex[3:0]	sseg[6:0]
0000	0000001
0001	1001111
0010	0010010
0011	0000110
0100	1001100
0101	0100100
0110	0100000
0111	0001111
1000	0000000
1001	0000100
1010	0001000
1011	1100000
1100	0110001
1101	1000010
1110	0110000
1111	0111000



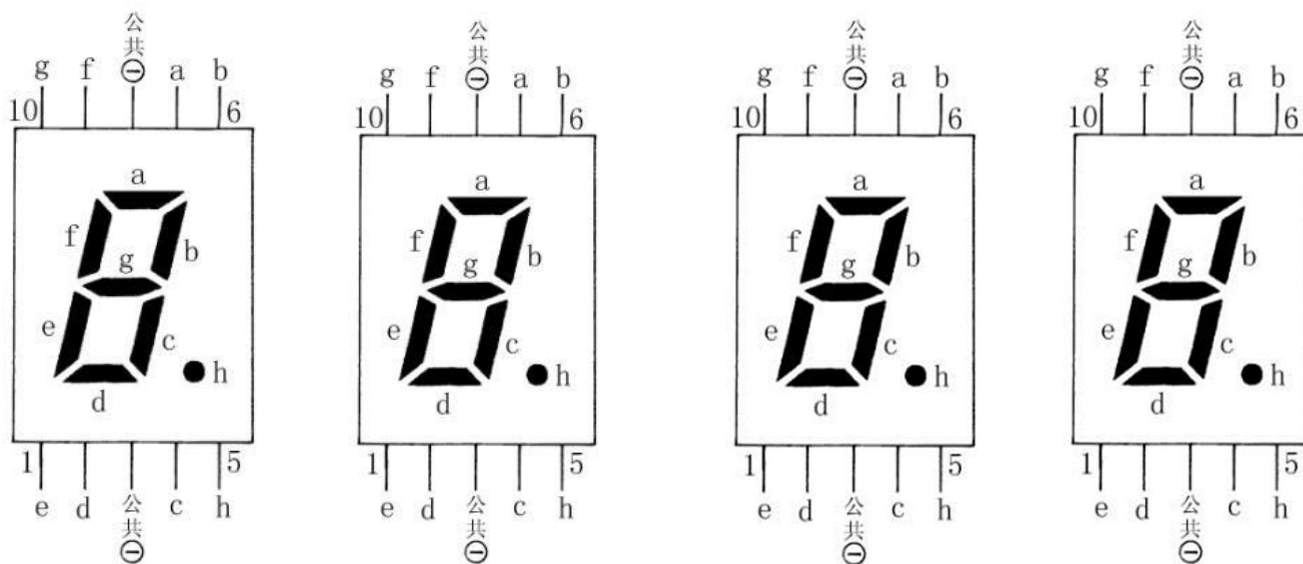
# LED显示输出

```
◆ reg dp, cg, cf, ce, cd, cc, cb, ca;
◆ always @(data) begin
◆     case(data)
◆         4'h0: {ca, cb, cc, cd, ce, cf, cg, dp} = 8'b0000_0011;
◆         4'h1: {ca, cb, cc, cd, ce, cf, cg, dp} = 8'b1001_1111;
◆         4'h2: {ca, cb, cc, cd, ce, cf, cg, dp} = 8'b0010_0101;
◆         4'h3: {ca, cb, cc, cd, ce, cf, cg, dp} = 8'b0000_1101;
◆         4'h4: {ca, cb, cc, cd, ce, cf, cg, dp} = 8'b1001_1001;
◆         4'h5: {ca, cb, cc, cd, ce, cf, cg, dp} = 8'b0100_1001;
◆         4'h6: {ca, cb, cc, cd, ce, cf, cg, dp} = 8'b0100_0001;
◆         4'h7: {ca, cb, cc, cd, ce, cf, cg, dp} = 8'b0001_1111;
◆         4'h8: {ca, cb, cc, cd, ce, cf, cg, dp} = 8'b0000_0001;
◆         4'h9: {ca, cb, cc, cd, ce, cf, cg, dp} = 8'b0000_1001;
◆         4'ha: {ca, cb, cc, cd, ce, cf, cg, dp} = 8'b0001_0001;
◆         4'hb: {ca, cb, cc, cd, ce, cf, cg, dp} = 8'b1100_0001;
◆         4'hc: {ca, cb, cc, cd, ce, cf, cg, dp} = 8'b1110_0101;
◆         4'hd: {ca, cb, cc, cd, ce, cf, cg, dp} = 8'b1000_0101;
◆         4'he: {ca, cb, cc, cd, ce, cf, cg, dp} = 8'b0110_0001;
◆         4'hf: {ca, cb, cc, cd, ce, cf, cg, dp} = 8'b0111_0001;
◆         default: {ca, cb, cc, cd, ce, cf, cg, dp} = 8'b1111_1111;
◆     endcase
◆ end
◆ assign segments = {dp, cg, cf, ce, cd, cc, cb, ca};
```



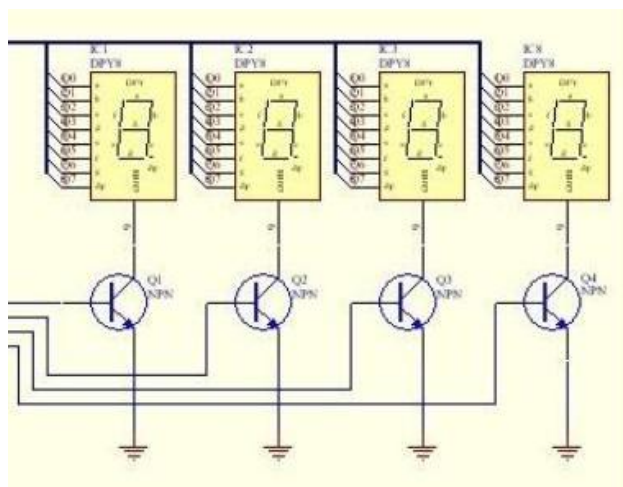
# 数码管扫描显示电路

- 在前面我们介绍了单个数码管显示电路的设计。每个数码管包括7个笔画和1个小圆点，需要8个IO口来进行控制。采用这种控制方式，当使用多个数码管进行显示时，每个数码管都需要8个IO口。



# 数码管扫描显示电路

- 在实际应用中，为了减少FPGA芯片IO口的使用数量，一般会采用分时复用的扫描显示方案进行数码管驱动。
- 以四个数码管显示为例，采用扫描显示方案时，四个数码管的8个段码并接在一起，再用4个IO口分别控制每个数码管的公共端，动态点亮数码管。这样只用12个IO口就可以实现4个数码管的显示控制，比静态显示方式的32个IO口数量大大减少。

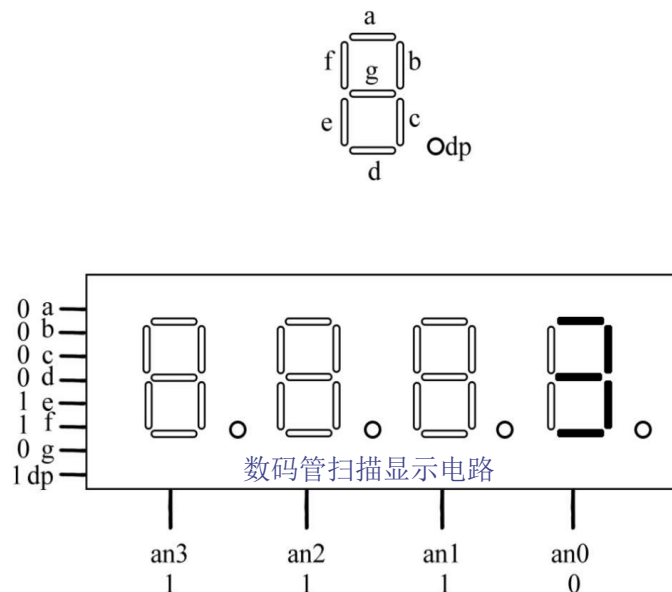




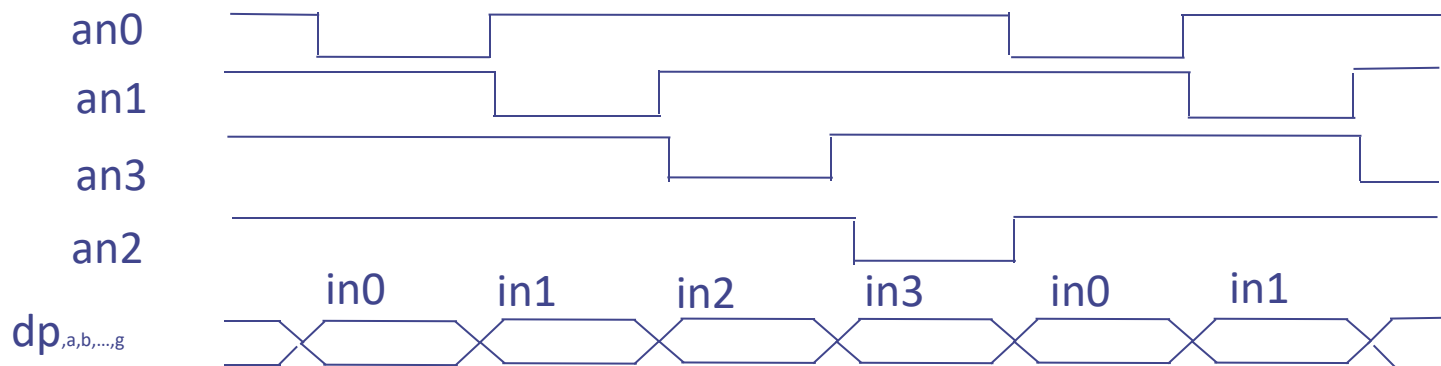
# 数码管扫描显示电路

如图所示，在最右端的数码管上显示“3”时，并接的段码信号为“00001101”，4个公共端的控制信号为“1110”。这种控制方式在同一时间只会点亮一个数码管，采用分时复用的模式轮流点亮数码管。

分时复用的扫描显示利用了人眼的视觉暂留特性，如果公共端控制信号的刷新速度足够快，人眼就不会区分出LED的闪烁，认为4个数码管是同时点亮。



# 数码管扫描显示电路



数码管扫描显示电路时序图

- 分时复用的数码管显示电路模块含有四个控制信号 $an3$ 、 $an2$ 、 $an1$ 和 $an0$ ，以及与控制信号一致的输出段码信号 $sseg$ 。
- 控制信号的刷新频率必须足够快才能避免闪烁感，但也不能太快，以免影响数码管的开关切换，最佳工作频率为1000Hz左右。



# 数码管扫描显示电路

```
module scan_led_disp
(
    input clk,reset,
    input [7:0] in3,in2,in1,in0,
    output reg [3:0] an,
    output reg [7:0] sseg
);
    localparam N = 16; //对输入50MHz时钟进行分
    频(50 MHz/2^16)
    reg [N-1:0] regN;

    always @(posedge clk,posedge reset)
        if (reset)
            regN <= 0;
        else
            regN <= regN + 1;
```

```
    always @*
        case (regN[N-1:N-2])
            2'b00: begin
                an = 4'b1110;
                sseg = in0;

            end
            2'b01: begin
                an = 4'b1101;
                sseg = in1;

            end
            2'b10: begin
                an = 4'b1011;
                sseg = in2;

            end
            default: begin
                an = 4'b0111;
                sseg = in3;

            end
        endcase
    endmodule
```

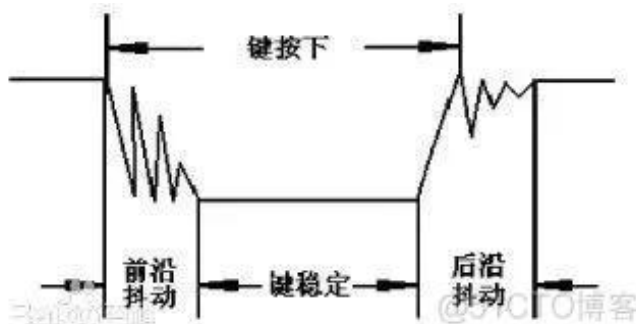
4位数码管动态显示电路的描述示例



# 按键消抖电路

## 简介：

- ◆ 我们在进行按键的时候往往会发生抖动的现象。
- ◆ 通常的按键所用开关为机械弹性开关，当机械触点断开、闭合时，由于机械触点的弹性作用，一个按键开关在闭合时不会马上稳定地接通，在断开时也不会一下子断开因而在闭合及断开的瞬间均伴随有一连串的抖动。这样的抖动会对我们的按键操作产生一些干扰，比如：有时候按下了一次按键，但是会发生很多次的功能的变化，这就是因为抖动的存在



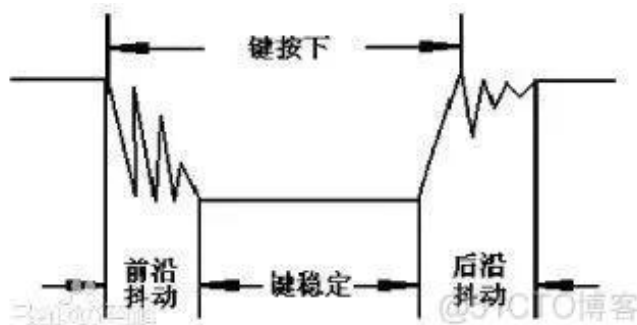
- ◆ 在机械按键的触点闭合和断开时，都会产生抖动，为了保证系统能正确识别按键的开关，就必须对按键的抖动进行处理。
- ◆ 按键的抖动对于人类来说是感觉不到的，但对硬件来说，则是完全可以感应到的，而且还是一个很“漫长”的过程，因为硬件的速度在“微秒”级，而按键抖动的时间至少在“毫秒”级。



# 按键消抖电路

## ◆ 按键消抖的方法:

- ◆ 按键稳定闭合时间长短是由操作人员决定的，通常都会在 **100ms** 以上，刻意快速按的话能达到 **40-50ms** 左右，很难再低了。抖动时间是由按键的机械特性决定的，一般都会在 **10ms** 以内，为了确保程序对按键的一次闭合或者一次断开只响应一次，必须进行按键的消抖处理。当检测到按键状态变化时，不是立即去响应动作，而是先等待闭合或断开稳定后再进行处理。按键消抖可分为硬件消抖和软件消抖。
- ◆ 当按键较多时，硬件方法将导致系统硬件电路设计复杂化，硬件消抖将无法胜任，这时常采用软件方法进行消抖。常用软件方法去抖，即检测出键闭合后执行一个延时程序，**5ms~10ms** 的延时，让前沿抖动消失后再一次检测键的状态，如果仍保持闭合状态电平，则确认为真正有键按下。当检测到按键释放后，也要给 **5ms~10ms** 的延时，待后沿抖动消失后才能转入该键的处理程序。我们使用 **Verilog HDL** 来实现的，因此就是属于软件消抖了。



# 按键消抖电路

## ◆ 软件消抖的基本原理：

- ◆ 在检测到有按键按下时，不是立即认定此键已被按下，而是执行一个**10ms**左右(具体时间应视所使用的按键进行调整)的延时程序后，再确认该键电平是否仍然保持闭合状态电平，若仍然保持，则确认该键真正被按下。
- ◆ 一般来说，软件消抖的方法是不断检测按键值，直到按键值稳定。实现方法：假设未按键时输入**1**，按键后输入为**0**，抖动时不定。可以做以下检测：检测到按键输入为**0**之后，延时**5ms~10ms**，再次检测，如果按键还为**0**，那么就认为有按键输入。延时的**5ms~10ms**恰好避开了抖动期，从而消除了前沿抖动的影响。同理，在检测到按键释放后，再延时**5~10ms**，消除后沿抖动，然后再对键值进行处理。不过一般情况下，我们通常不对按键释放的后沿进行处理，实践证明，这样也能满足一定的要求。



# 一个实例

- ◆ 以一个十字路口交通管理信号灯为例。设该交通灯系统用于主干道与乡间公路之间的交叉路口，要求是优先保证主干道的畅通，因此平时处于“主干道绿灯，乡间道红灯”状态，只有在乡间公路有车辆要穿行主干道时才将交通灯切向“主干道红灯，乡间道绿灯”，一旦乡间公路无车辆通过路口，交通灯又回到“主绿、乡红”的状态。此外，主干道每次通行的时间不得短于1分钟，乡间公路每次通行时间不得长于20秒。而在两个状态交换过程中出现的“主黄，乡红”和“主红，乡黄”状态，持续时间都为4秒。

