

# 基于FPGA的 组合逻辑电路设计和实现 EDA实验一

赵晓燕  
电工电子实验中心

# 主要内容

- 一、设计文件输入方法
- 二、HDL硬件描述语言
- 三、VerilogHDL基本语法
- 四、基于FPGA的组合电路设计
- 五、EDA实验一内容布置

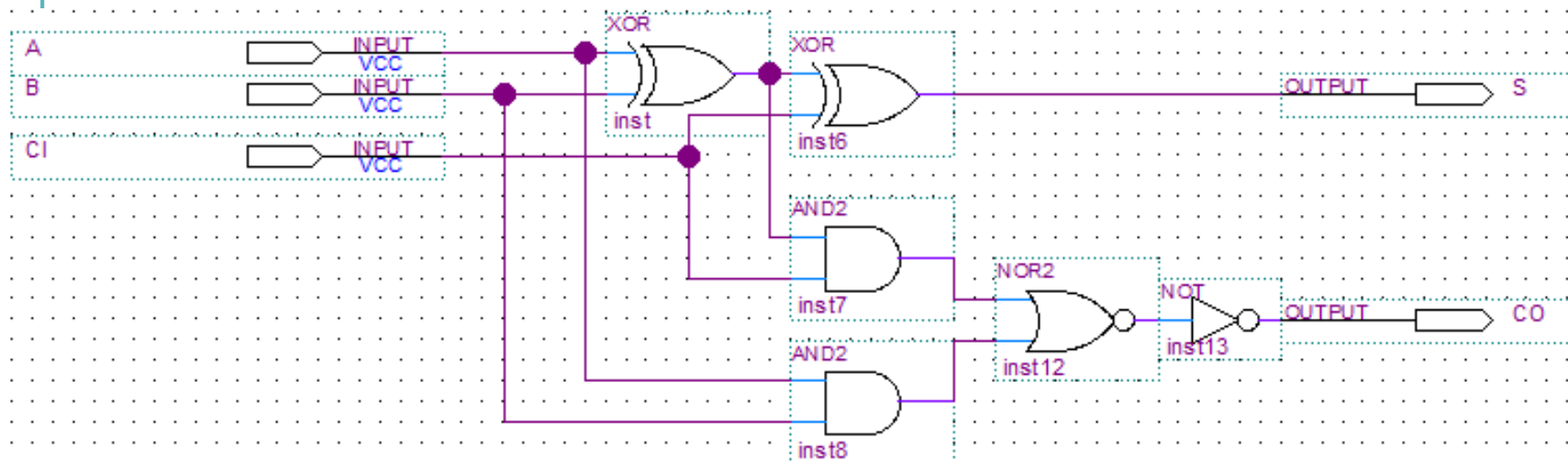
# 一、设计文件的输入方法

➤ 原理图输入

➤ 代码（HDL）输入


➤ 混合输入

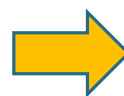
# 原理图输入 (以一位全加器为例)



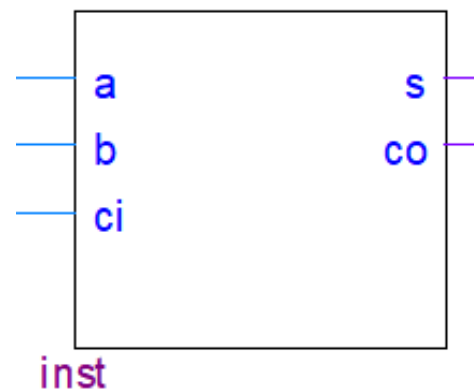
封装



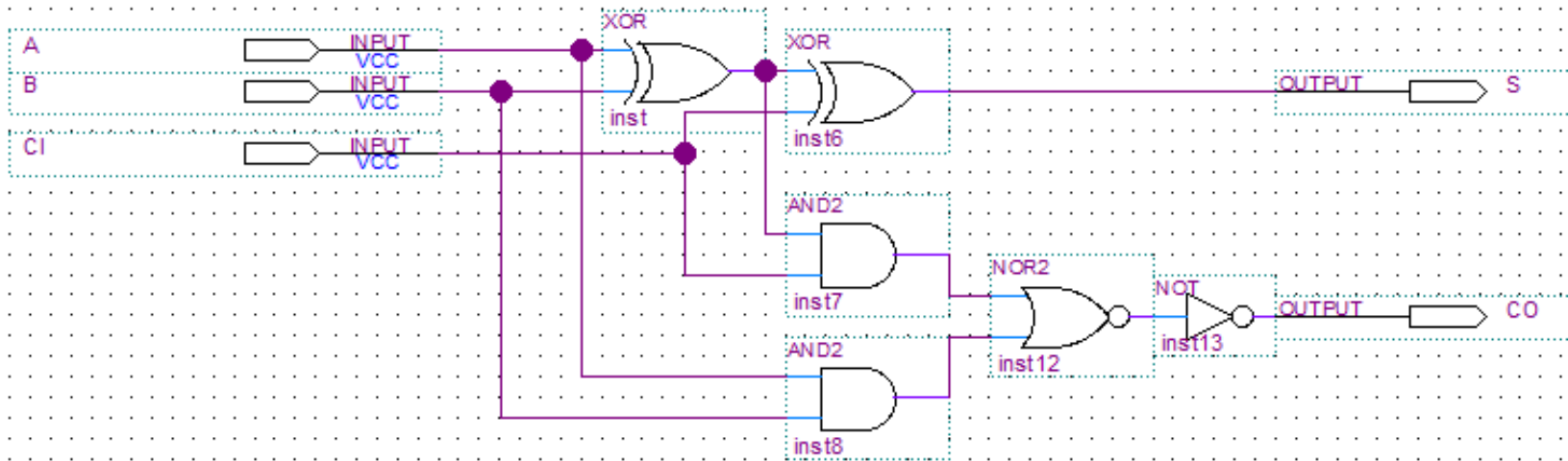
Open	
Remove File from Project	
 Set as Top-Level Entity	Ctrl+Shift+J
Create AHDL Include Files for Current File	
Create Symbol Files for Current File	
Properties...	



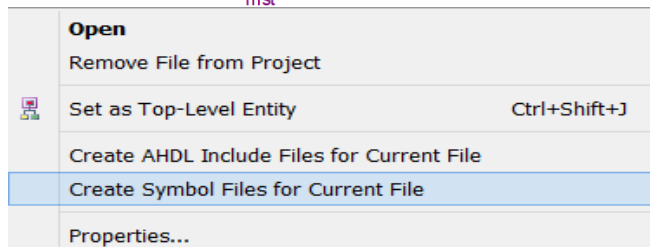
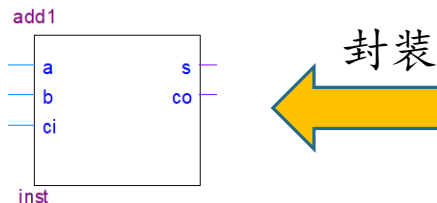
add1



# 代码 (HDL) 输入



## 或者：HDL输入



```
module add1(a,b,ci,s,co);
```

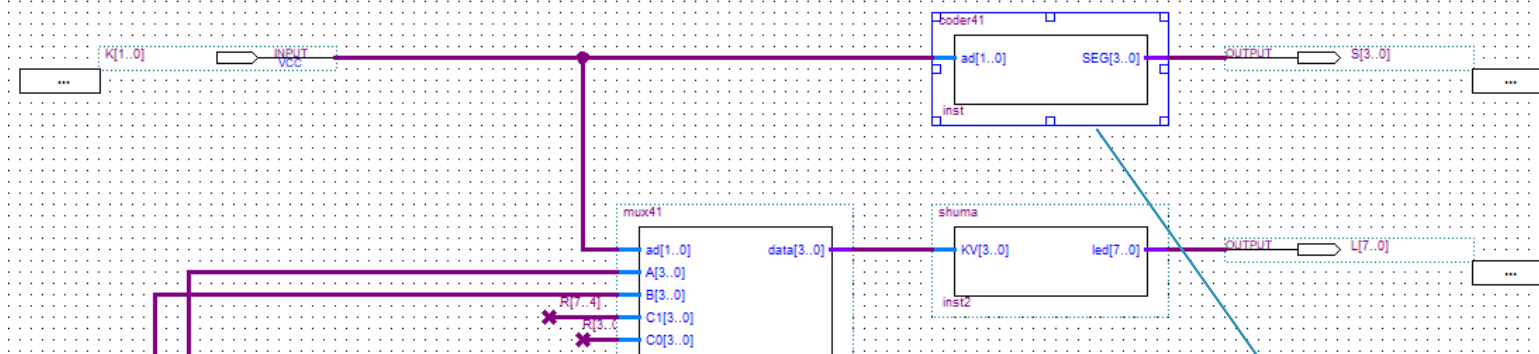
```
input a,b,ci;
```

```
output s,co;
```

```
assign {co, s} = a + b + ci;
```

```
endmodule
```

# 混合输入



- 电路信号关系直观明确
- 顶层模块原理图输入
- 底层模块代码输入/原理图输入

```
1 module coder41(ad,SEG);
2   input[1:0] ad;
3   output[3:0] SEG;
4   reg [3:0] SEG;
5
6   always@(*)
7   case(ad)
8     2'b00: SEG=4'b0111;
9     2'b01: SEG=4'b1011;
10    2'b10: SEG=4'b1101;
11    2'b11: SEG=4'b1110;
12
13  endcase
14 endmodule
```

## 二、HDL硬件描述语言

➤ 硬件描述语言描述电路功能与结构

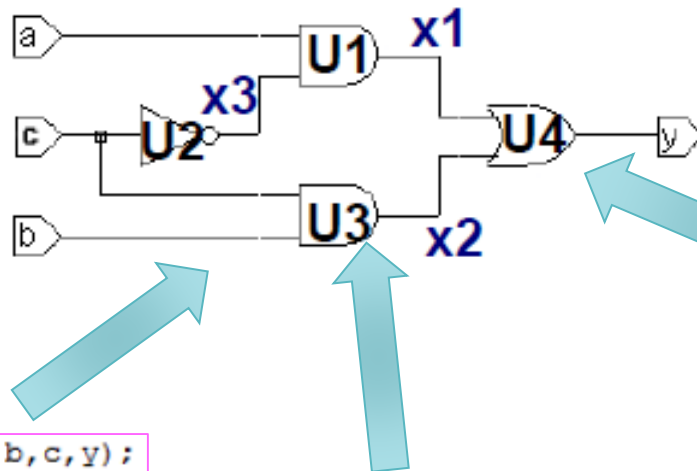
➤ 两种流行语言 Verilog HDL和VHDL

- Verilog HDL语言结构灵活，语法上类似C语言，易于掌握
  - VHDL (Very High Speed Integrated Circuit Hardware Description Language) 语言高层抽象能力要稍优一些。语言规范十分严谨，甚至于繁琐，但可读性好。
- 大学、研究机构更多使用VHDL，而工业界更多使用Verilog HDL

Verilog HDL与VHDL描述加法器电路

library ieee;	
Use ieee.std_logic_1164.all;	
Use ieee.std_logic_arith.all;	
entity vadd is	module kadd (a,b,c,s);
port (a,b: in std_logic_vector(7 downto 0);	input[7:0] a ,b;
c: in std_logic_vector(0 to 0);	input c;
s : out std_logic_vector( 8 downto 0 ));	output[8:0] s;
End vadd;	
architecture rtl of vadd is	
begin	
s <= unsigned(a)+unsigned(b)+unsigned(c);	assign s = a+b+c;
End rtl;	endmodule

# VerilogHDL语言描述电路



```
module mux21 (a,b,c,y);  
  input a,b,c;  
  output y;  
  
  wire x1,x2,x3;  
  
  and U1 (x1,a,x3);  
  not U2 (x3,c);  
  and U3 (x2,c,b);  
  or U4 (y,x1,x2);  
  
endmodule
```

结构描述

```
module mux21 (a,b,c,y);  
  input a,b,c;  
  output y;  
  
  assign y=(a&~c) | (b&c);  
  
endmodule
```

函数描述

```
module mux21 (a,b,c,y);  
  input a,b,c;  
  output y;  
  
  reg y;  
  
  always@(a or b or c)  
  begin  
    case (c)  
      1'b0:y<=a;  
      1'b1:y<=b;  
      //default:y<=a;  
    endcase  
  end  
  
endmodule
```

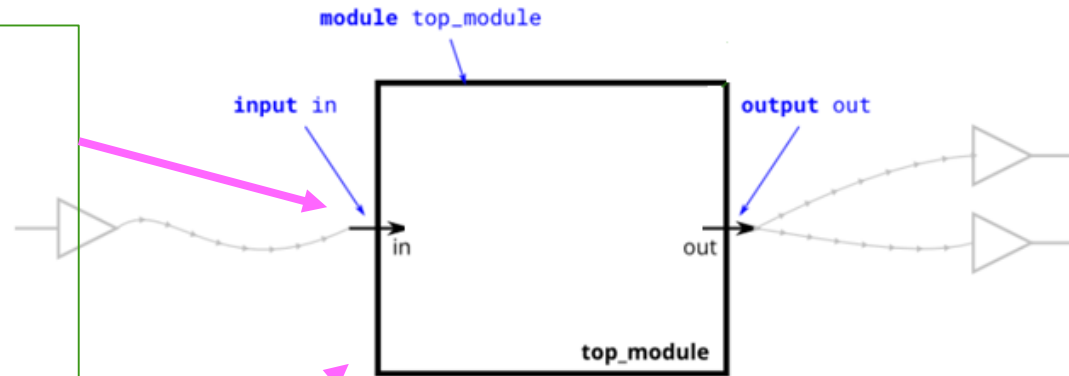
行为描述



# Verilog基本语法-定义模块module

```
module top_module ( in,out );  
  input in; // 声明输入线in  
  output out; //声明输出线名称out  
  
  //代码主体  
  
endmodule
```

Verilog-1995

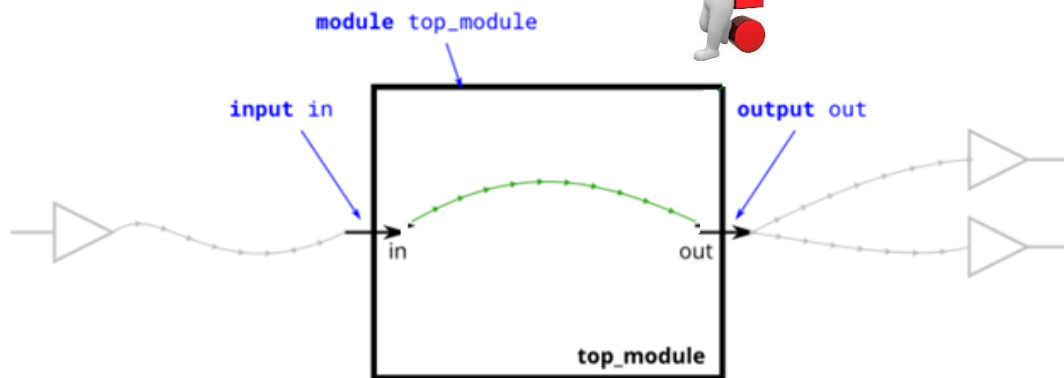


```
module top_module (input in, output out);  
  
  //代码主体  
  
endmodule
```

Verilog-2021

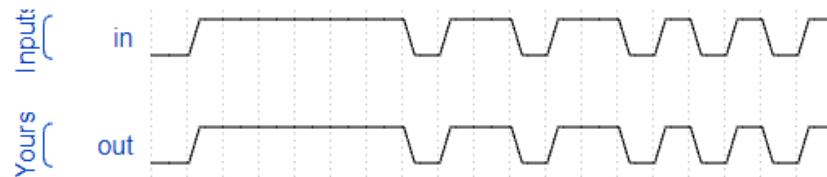
- 关键字`module`定义模块名称
- 关键字`input`、`output`定义输入、输出端口
- 关键字`endmodule`结束模块定义
- 每一条语句结尾要有`;`

# Verilog基本语法--创建连线assign

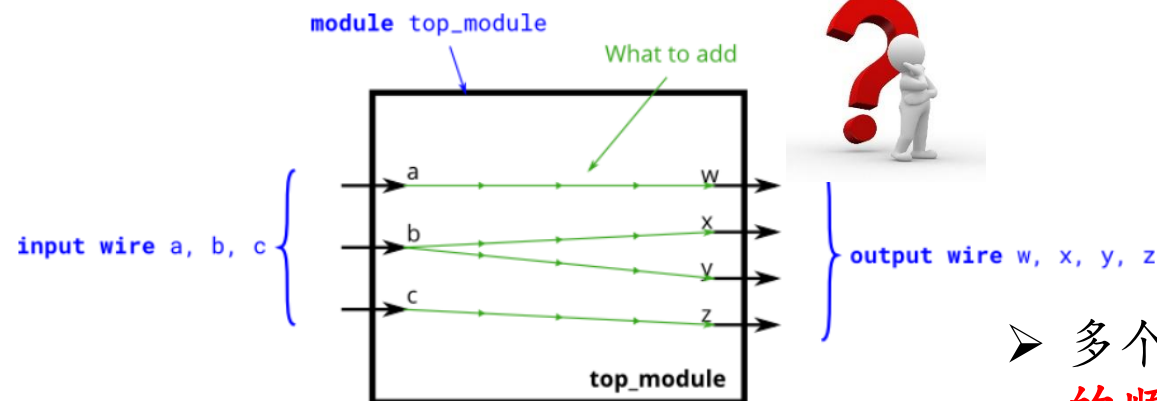


- **assign** left\_side = right\_side
- **assign** 语句（“连续赋值”）描述事物之间的连接，将右侧变量连续分配给（驱动）左侧变量
- RHS 变化，LHS 立即变化

```
module top_module( input in, output out );  
  
    assign out=in;  
  
endmodule
```



# Verilog基本语法--创建连线assign

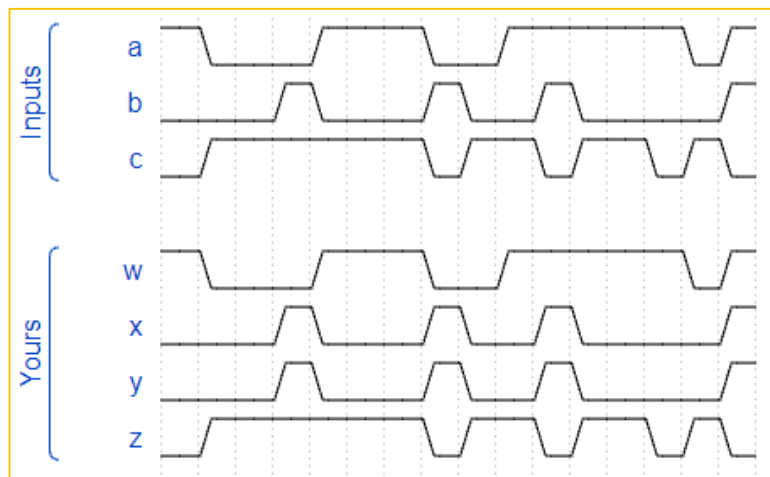


- 多个赋值语句在代码中**出现**的顺序，不影响电路最终功能实现。与编程语言不同。

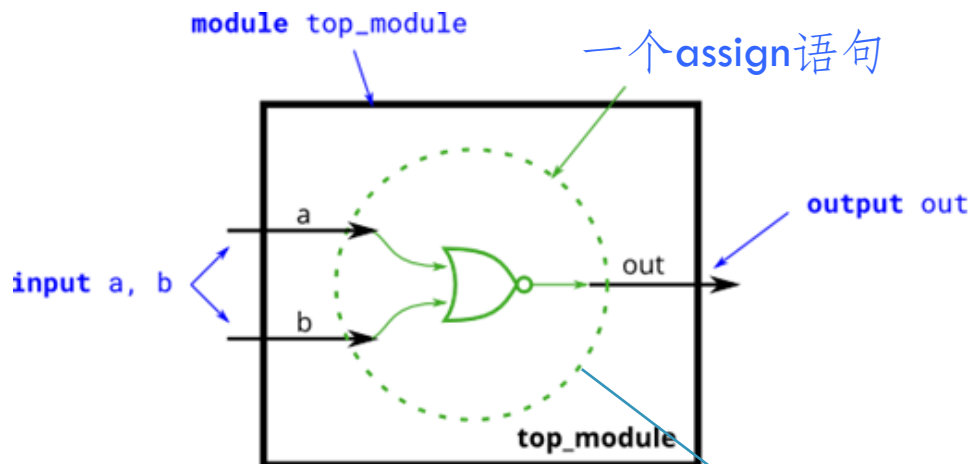
```
module top_module(  
    input a,b,c,  
    output w,x,y,z );  
    .  
endmodule
```

```
assign w=a;  
assign x=b;  
assign y=b;  
assign z=c;
```

```
endmodule
```



# Verilog基本语法-----添加门电路



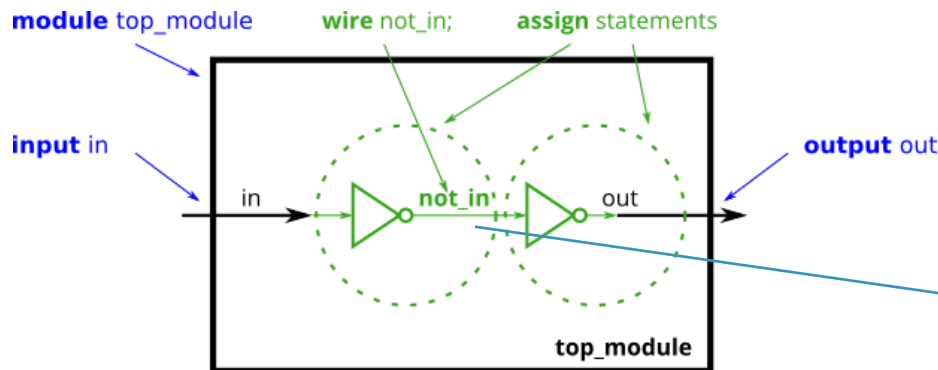
逻辑	按位	逻辑
非	~	!
与	&	&&
或		
异或	^	无

```
module top_module(  
    input a,  
    input b,  
    output out );
```

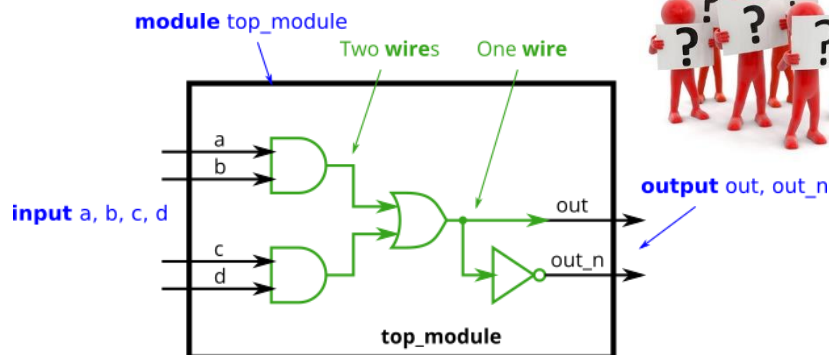
```
    assign out=!(a || b);
```

```
endmodule
```

# Verilog基本语法----- 创建内部连线wire



```
module top_module (  
    input in,           // 声明输入端口 in  
    output out          // 声明输出端口 out  
);  
  
    // 声明内部连接线  
    wire not_in;  
  
    assign out = ~not_in; // 产生非门  
    assign not_in = ~in;  // 产生另一个非门  
endmodule // 名为“top_module”模块结束
```

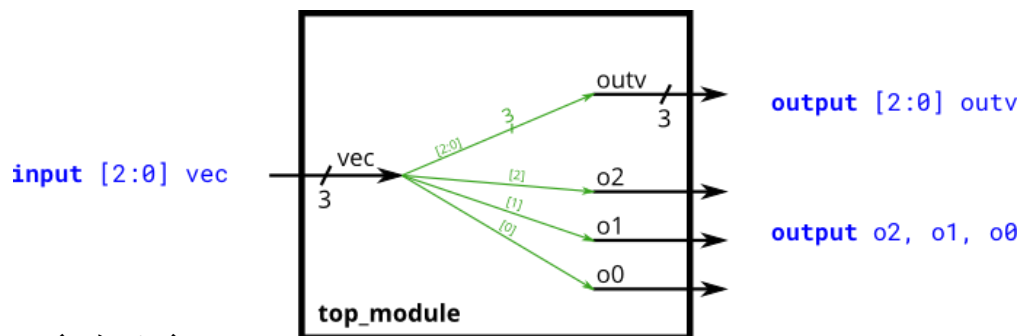


4个赋值语句

非门输入的连线实际上是模块的输出连线，不必声明三条内部连线变量

- 关键字 `wire` 定义线型变量
- 内部连线变量在首次使用前要声明为 `wire` 类型
- 模块输入、输出端口变量默认为 `wire` 类型
- 在模块内部声明的内部连线变量 `not_in` 从模块外部看不到
- 创建非门的先后顺序不影响最终电路的实现

# Verilog基本语法-----向量操作



## ➤ 向量声明

- 在功能上等同于具有多条单独的线（总线）
- 向量声明将维度放在**向量名称之前**



```
wire [99:0] my_vector;
```

```
// 声明包含100个元素的向量
```

## ➤ 向量部分选择



```
assign out = my_vector[10];
```

```
// 从向量里选一位输出
```

```
module top_module (  
    input wire [2:0] vec,  
    output wire [2:0] outv,  
    output o2,  
    output o1,  
    output o0 );
```

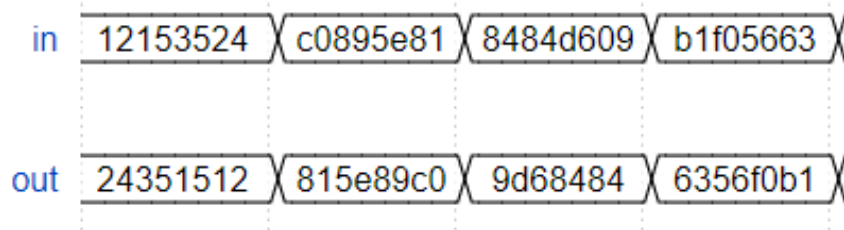
```
    assign outv=vec;  
    assign o0=vec[0];  
    assign o1=vec[1];  
    assign o2=vec[2];
```

```
endmodule
```

# Verilog基本语法-----向量操作



## 向量中字节互换



通过向量部分选择  
进行赋值，不需给  
整个向量赋值。

```
module top_module(  
    input [31:0] in,  
    output [31:0] out );
```

```
    assign out[31:24]=in[7:0];  
    assign out[23:16]=in[15:8];  
    assign out[15:8]=in[23:16];  
    assign out[7:0]=in[31:24];
```

```
endmodule
```

4'b10\_11



下划线（可以忽略）

进制(d,b,o,h)

位宽（十进制数表示）



## 向量中位互换

assign out[7:0] = in[0:7];



assign out={in[0],in[1],in[2],in[3],in[4],in[5],in[6],in[7]};



# 向量操作-----拼接运算符{ }

- 拼接运算符 { } 将部分向量连接在一起创建更大的向量。
- 要定义每个组成部分的宽度，拼接中不允许使用未确定位宽的常量。



```
{3'b111, 3'b000} => 6'b111000  
{1'b1, 1'b0, 3'b101} => 5'b10101  
{4'ha, 4'd10} => 8'b10101010 // 4'ha 和 4'd10 都是 4'b1010
```

- 拼接运算符可用于赋值语句的左侧和右侧。



```
input [15:0] in;  
output [23:0] out;  
assign {out[7:0], out[15:8]} = in; // 交换两字节  
assign out[15:0] = {in[7:0], in[15:8]}; // 同上。  
assign out = {in[7:0], in[15:8]}; // 右侧16位向量扩展为左侧24位向量，  
out[23:16]是0。前两个例子中 out[23:16]  
不被赋值。
```



# 向量操作----- 向量复制

➤ `{num{vector}}` 表示 `vector` 重复 `num` 次。注意需要2个花括号

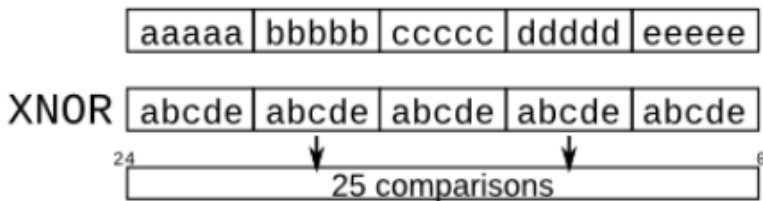


```
{5{1'b1}} // 5'b11111 (or 5'd31 or 5'h1f)
```

$$\{2\{a,b,c\}\} \quad // \text{ 同 } \{a,b,c,a,b,c\}$$

`{3'd5, {2{3'd6}}}` // 9'b101\_110\_110.是一个向量101和两个向量110的级联

```
{5{1'b1}}           // 5'b11111 (or 5'd31 or 5'h1f)
```



```
out[24] = ~a ^ a;
```

```
out[23] = ~a ^ b;
```

```
out[22] = ~a ^ c;
```

...

$$\text{out}[1] = \tilde{e}^d;$$
$$\text{out}[0] = \sim e \wedge e;$$

$$\text{assign out} = \sim\{\{5\{a\}\}, \{5\{b\}\}, \{5\{c\}\}, \{5\{d\}\}, \{5\{e\}\}\} \wedge \{5\{a,b,c,d,e\}\};$$

# Verilog基本语法-----数据类型

常量

- 数字

`8'b0000_0100`

`6'h1f`

`128`

- 参数

```
parameter WIDTH = 8;
```

```
wire [WIDTH-1:0] data;
```

`4'b10_11`



下划线 (可以忽略)

进制 (d,b,o,h)

位宽 (十进制数表示)

变量

- 线型 (wire)      assign赋值 =
- 寄存器型 (reg)      always赋值 <=

# Verilog基本语法-----运算符

## ➤ 位运算符 $\sim$ 、 $\&$ 、 $|$ 、 $\wedge$ 、 $\sim\wedge$ (同或)

- 运算数是矢量，运算逐位进行



```
4'b0100 | 4'b1001 = 4'b1101  
~8'b0110_1100 = 8'b1001_0011
```

$\sim a$	NOT
$a \& b$	AND
$a   b$	OR
$a \wedge b$	XOR
$a \sim\wedge b$	XNOR

## ➤ 逻辑运算符 $!$ 、 $\&\&$ 、 $||$

- 产生一个逻辑值



```
4'b0000 || 4'b0111 = 1  
4'b0000 && 4'b0111 = 0  
!4'b0000 = 1
```

$!a$	NOT
$a \&\& b$	AND
$a    b$	OR

# Verilog基本语法-----运算符

## ➤ 关系运算符 >、<、>=、<=、==、!=



```
(4'b1011 < 4'b0111) = 0
```

```
(4'b1011 != 4'b0111) = 1
```

## ➤ 条件运算符 ?:

```
<conditional_expression> ? <expression1> : <expression2>
```



```
assign out=(sel)?in1:in0;
```

## ➤ 拼接运算符 {}

- 级联多个运算数



```
{5'b10110, 2'b10, 1'b0}=8'b1011_0100
```

- 同一个运算数级联多次



```
{3{3'b101}}=9'b101_101_101
```

# Verilog基本语法-----运算符

## ➤ 算数运算符 $+$ 、 $-$ 、 $*$ 、 $/$ 、 $\%$

- $/$  除法运算
- $\%$  求模运算

## ➤ 移位运算符 $\gg$ 、 $\ll$



```
8'b0011_1100 >> 2 = 8'b0000_1111  
8'b0011_1100 << 2 = 8'b1111_0000
```

## ➤ 位缩减运算符

- 只有一个运算数  $\sim$ 、 $\&$ 、 $|$
- 对每个向量完成逐位运算
- 产生1比特运算结果



```
&4'b0101 = 0 & 1 & 0 & 1 = 0  
|4'b0101 = 0 | 1 | 0 | 1 = 1
```

# Verilog基本语法---always过程块

- always过程块提供行为描述
- always过程块可描述组合电路和时序电路
- always过程块表达形式

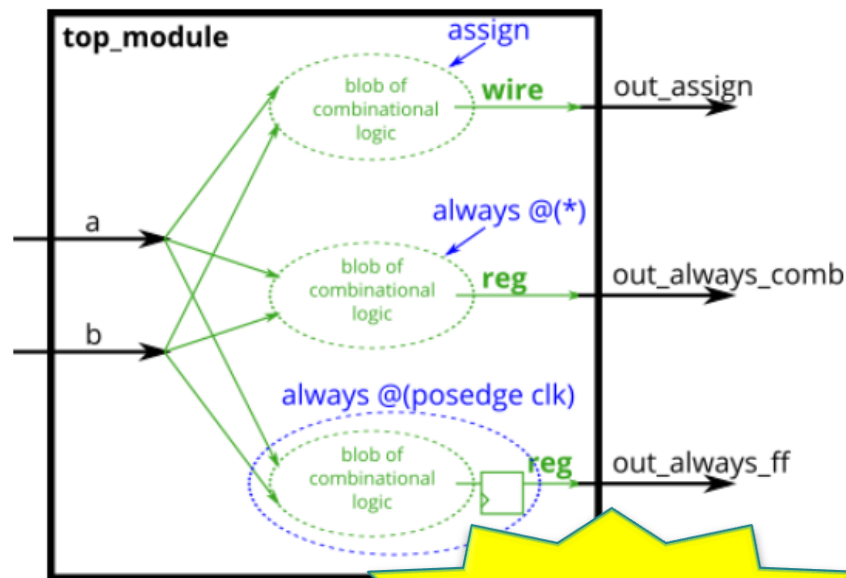
**always@ (敏感列表)**

**begin**

**赋值语句**

**end**

- always@ (a, b, c) 描述组合电路
- always@ (\*) 描述组合电路
- always@ (posedge clk ) 描述时序电路
- always@ (negedge clk, posedge rst) 描述时序电路



**并行**

- 赋值语句包括if else语句、case语句、for循环语句，不包括连续赋值语句 **assign**，组合电路用“=”赋值，时序电路用“<=”赋值。
- begin end中的语句顺序执行
- assign赋值语句声明的变量是 **wire**类型，而在 **always过程块**中的变量必须声明为 **reg**类型

# always过程块--组合电路



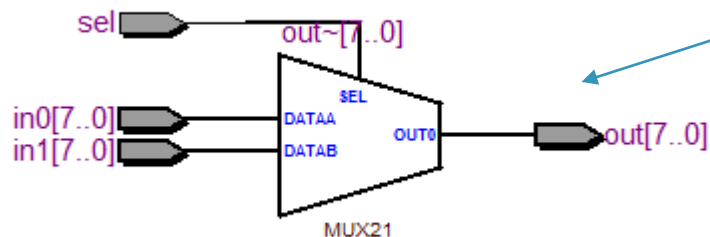
## 8位二选一数据选择器

Tools/Netlist Viewers/RTL Viewer  
查看QuartusII综合的电路

```
module Mux_2to1_8bit(  
    input  wire[7:0] in0,  
    input  wire[7:0] in1,  
    input  wire      sel,  
    output wire [7:0] out  
);
```

```
    assign out=(sel)?in1:in0;
```

```
endmodule
```



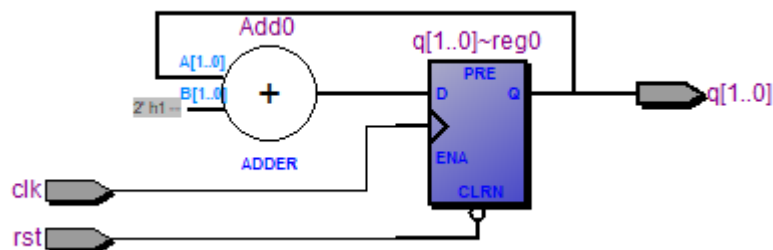
```
output reg [7:0] out  
always@(*)  
    case (sel)  
        1'b0: out=in0;  
        1'b1: out=in1;  
    endcase
```

```
output reg [7:0] out  
always@(*)  
    if (sel==0)  
        out=in0;  
    else  
        out=in1;
```

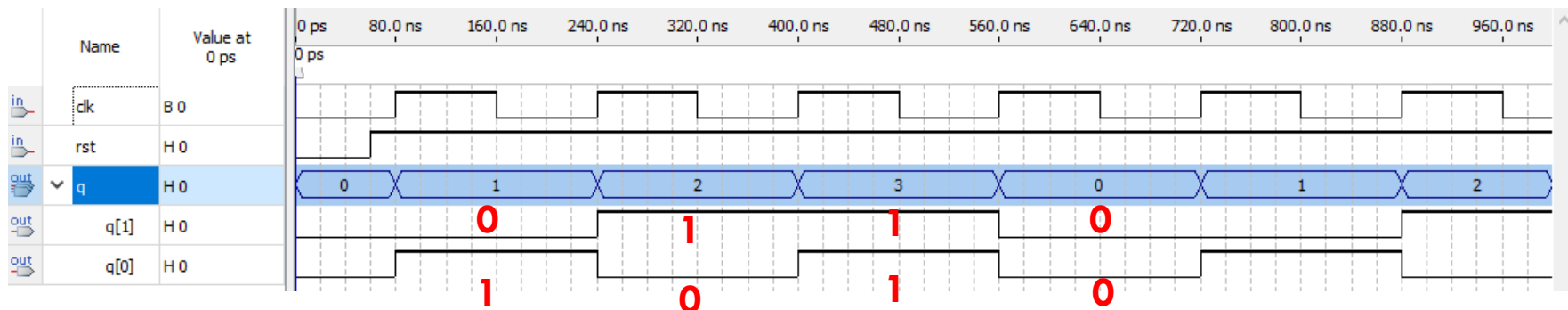
# always过程块---时序电路



## 4进制计数器



```
module count_4(clk,rst,q);  
input clk,rst;  
output [1:0]q;  
reg [1:0]q;  
  
always@(posedge clk or negedge rst)begin  
    if(!rst)  
        q<=0;  
    else  
        q<=q+1; // 累加1  
    end  
endmodule
```





# always过程块---时序电路

```
module femp(clki,rst,clk1,clk2);  
input clki,rst;  
output clk1,clk2;
```



分频器

```
reg [30:0]q;
```

```
always@(posedge clki, negedge rst)
```

```
    if(!rst)  q<=0;  
    else  q=q+1;
```

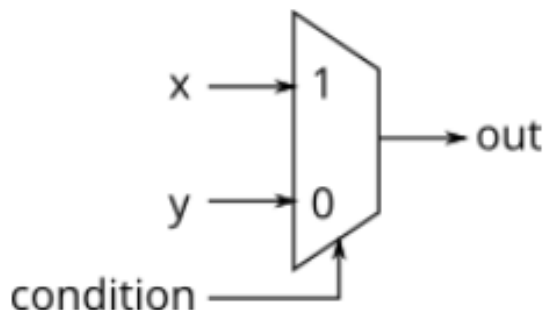
```
assign clk1=q[16]; //输出250hz  
assign clk2=q[24]; //输出1HZ
```

```
endmodule
```

计数器输出  
相邻位满足  
2倍频关系

# Verilog基本语法---if语句

- if 语句通常创建一个 2选1多路复用器
- 如果if后面的条件为真则选择一个输入，如果条件为假则选择另一个输入。
- 一般每个if 语句要与else匹配



只有当 **out** 总是被分配一个值时，电路才是组合的。

```
always @(*) begin
    if (condition) begin
        out = x;
    end
    else begin
        out = y;
    end
end
```

**assign** out = (condition) ? x : y;

# Verilog基本语法---case语句

```
always @(*) begin    // 组合电路
```

```
    case (in)
```

```
        1'b1: begin
```

```
            out = 1'b1;  
            out1 = 1'b1;  
        end
```

```
        1'b0: out = 1'b0;
```

```
        default: out = 1'bx;
```

```
    endcase
```

```
end
```

- case 语句以 case 开头，每个 case 项只能执行一个语句，不需要关键字“switch”“break”。
- 一个case需要执行多个语句，则必须使用 begin ... end。
- case语句分支要完整，否则要包括 default
- 允许使用重复的case项。选择第一个匹配项执行。C 不允许重复的case 项。见casez例子

# case语句



## 4位优先编码器

例如：输入4'b1001，编码器输出 2'd0

```
module top_module (  
    input [3:0] in,  
    output reg [1:0] pos );  
    always@(*)
```

```
if((in[0]==1))    pos=2'd0;  
else if(in[1]==1) pos=2'd1;  
else if(in[2]==1) pos=2'd2;  
else if(in[3]==1) pos=2'd3;  
else pos=2'd0;
```

```
endmodule
```

OR



```
module top_module (input [3:0] in,  
                    output reg [1:0] pos );  
    always @(*) begin  
        case(in)
```

```
4'h1: pos = 0;  
4'h2: pos = 1;  
4'h3: pos = 0;  
4'h4: pos = 2;  
4'h5: pos = 0;  
4'h6: pos = 1;  
4'h7: pos = 0;  
4'h8: pos = 3;  
4'h9: pos = 0;  
4'ha: pos = 1;  
4'hb: pos = 0;  
4'hc: pos = 2;  
4'hd: pos = 0;  
4'he: pos = 1;  
4'hf: pos = 0;  
default: pos = 0;
```

```
endcase  
end  
endmodule
```

# casez语句

➤ casez语句值为z的位视为无关位

```
always @(*) begin
```

```
    casez (in[3:0])
```

```
        4'bzzz1: out = 0;
```

```
        // in[3:1] 可以为任何值
```

```
        4'bzz1z: out = 1;
```

```
        4'bz1zz: out = 2;
```

```
        4'b1zzz: out = 3;
```

```
        default: out = 0;
```

```
    endcase
```

```
end
```



## 4位优先编码器

例如：输入4'b1001，编码器输出 2'd0

➤ 请注意某些输入（例如 4'b1111）如何匹配多个 case 项。选择第一个匹配项（因此 4'b1111 匹配第一个项目，out = 0，但不匹配任何后面的项目）。

➤ ?与 z 相同。2'bz0 即 2'b?0

# 分支不完全的case语句

```
module mux4(input a, b, c,  
            input [1:0] sel,  
            output reg out );
```

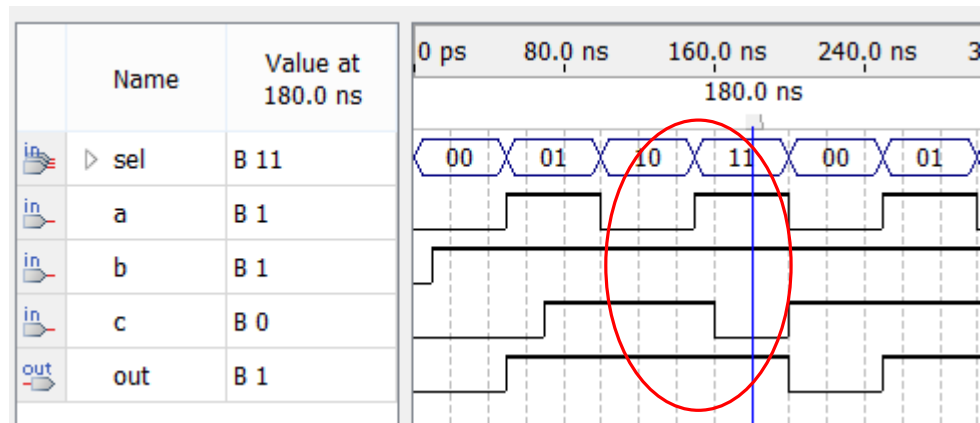
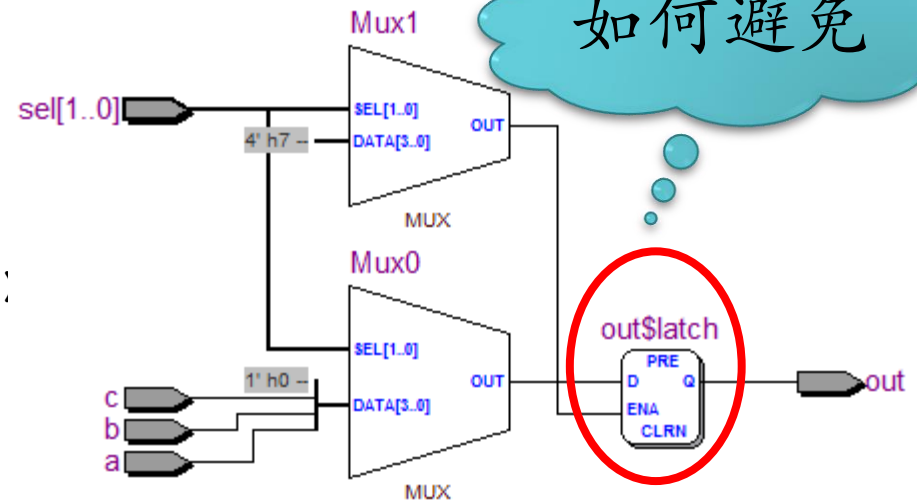
```
always @( a or b or c or sel )  
begin
```

```
    case ( sel )  
        2'b00 : out = a;  
        2'b01 : out = b;  
        2'b10 : out = c;
```

```
    endcase
```

```
end
```

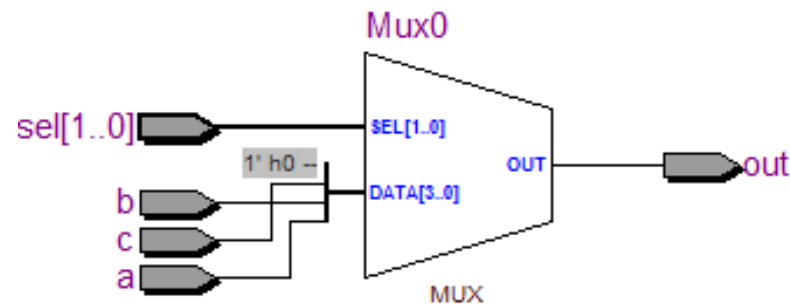
```
endmodule
```



# 分支不完全的case语句

```
always @( a or b or c or sel )  
begin  
    case ( sel )  
        2'b00 : out = a;  
        2'b01 : out = b;  
        2'b10 : out = c;  
        default: out=1'b0;  
    endcase  
end
```

```
always @( a or b or c or  
sel )  
begin  
    out=1'b0;  
    case ( sel )  
        2'b00 : out = a;  
        2'b01 : out = b;  
        2'b10 : out = c;  
    endcase  
end
```



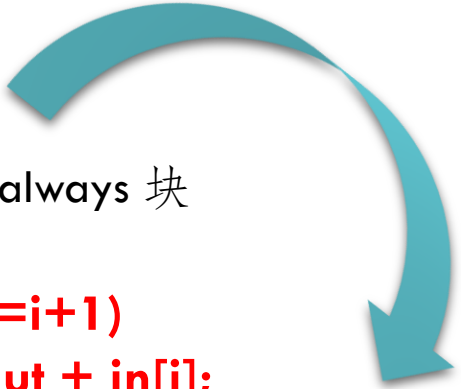
# Verilog基本语法--for循环



统计255 位输入向量中“1”的个数

```
module top_module (  
    input [254:0] in,  
    output reg [7:0] out  
);  
  
    integer i;  
    always @(*) begin // 组合always 块  
        out = 0;  
        for (i=0;i<255;i=i+1)  
            out = out + in[i];  
    end  
  
endmodule
```

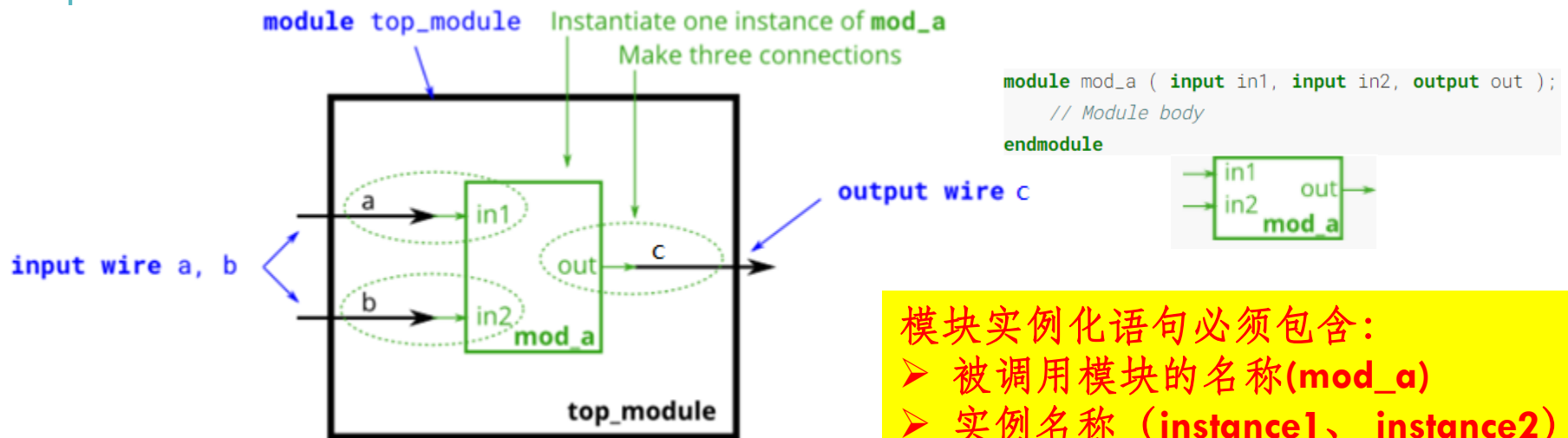
verilog中的for循环是将相同的电路复制多次，占用资源、综合慢、甚至综合出错，**建议尽量少用。**



in	0	1	3	7	aaaa	f00000
out	0	1	2	3	8	4



# 模块实例化（调用功能电路）



模块实例化语句必须包含：

- 被调用模块的名称(mod\_a)
- 实例名称 (instance1、instance2)
- 连接的端口名

通过端口位置关联：

```
mod_a instance1 ( a, b, c );
```

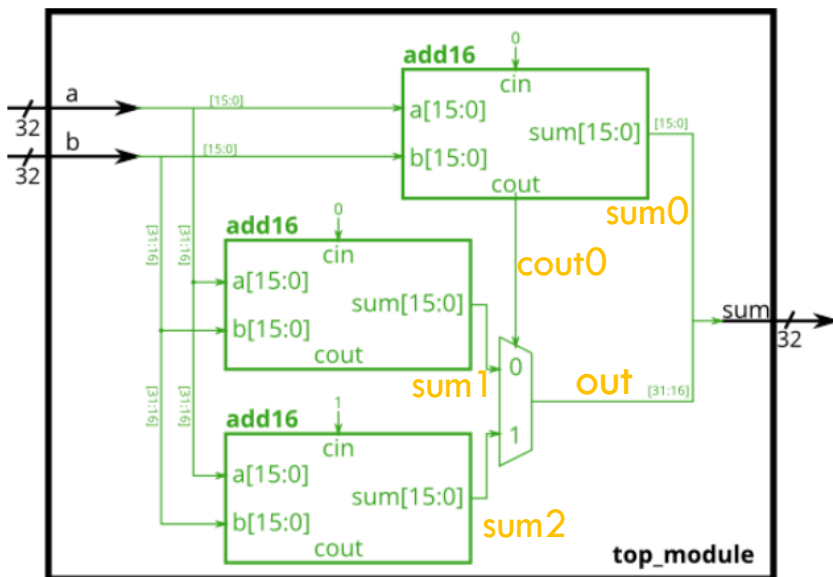
- 类似 C 的用法。实例化一个模块时，端口根据被调用模块声明中端口位置，从左到右依次连接
- 缺点：如果被调用模块的端口列表顺序发生变化，需修改所有实例端口顺序。

通过端口名称关联：

```
mod_a instance2 ( .out(c), .in1(a), .in2(b) );
```

- 按端口名称连接到被调用模块的端口，即使被调用模块端口列表发生变化，连线也能保持正确连接。
- 调用时端口名称之前加句点
- 缺点：语法表达更加冗长。

# 模块实例化



- 复杂电路由功能电路、连线、逻辑门组成，形成电路的层次结构
- 一个模块中实例化（调用）另一个模块时
  - 只要所有模块都属于同一个项目
  - 不需要嵌套

```
module top_module(
    input [31:0] a,
    input [31:0] b,
    output [31:0] sum
);
    wire [15:0] sum0, sum1, sum2;
    reg [15:0] out;
    wire cout0, cout1, cout2;
```

```
    assign sum={out,sum0} ;
```

```
    always@(sum1 or sum2 or cout0)
        case(cout0)
            1'b0: out=sum1;
            1'b1: out=sum2;
        endcase
```

```
    add16 inst0(a[15:0],b[15:0],0,sum0,cout0);
    add16 inst1(a[31:16],b[31:16],0,sum1,cout1);
    add16 inst2(a[31:16],b[31:16],1,sum2,cout2);
```

```
endmodule
```

# 四、基于FPGA的电路设计

## Verilog代码基本结构

```
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;
```

```
    always  
    begin  
        // Statements  
    end
```

```
    // Continuous assignments...  
    assign W1 = Expression;
```

```
    // Module instances...  
    COMP U1 (W3, W4);  
    COMP U2 (.P1(W3), .P2(W4));
```

```
endmodule
```

### ➤ 先设计电路后写代码

- 自顶向下的设计
- 自底向上的实现

### ➤ 语法正确的代码不一定会产生合理的电路

- **注意警告 (10240) :**  
推断出电路存在锁存器
- 查看功能仿真波形、

### RTL Viewer

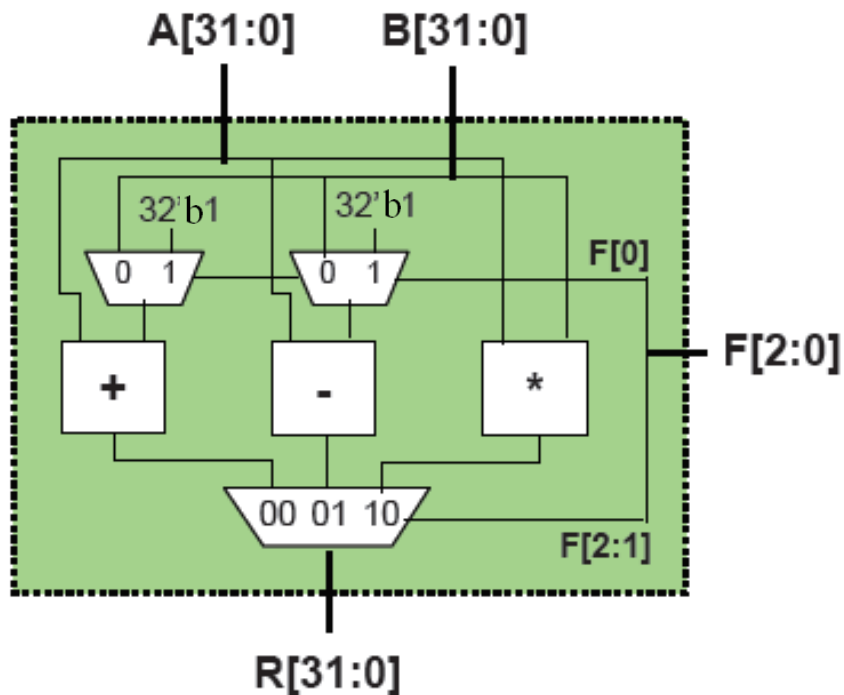
### ➤ 并行执行语句

- 过程语句
- 连续赋值语句
- 模块实例化语句

# 自顶向下的设计

- Modularity is essential to the success of large designs
- A Verilog module may contain submodules that are “wired together”
- High-level primitives enable direct synthesis of behavioral descriptions (functions such as additions, subtractions, shifts (<< and >>), etc.

## Example: A 32-bit ALU



## Function Table

F2	F1	F0	Function
0	0	0	$A + B$
0	0	1	$A + 1$
0	1	0	$A - B$
0	1	1	$A - 1$
1	0	X	$A * B$

# 自底向上的实现

## 2-to-1 MUX

```
module mux32two(i0,i1,sel,out);
input [31:0] i0,i1;
input sel;
output [31:0] out;

assign out = sel ? i1 : i0;

endmodule
```

## 3-to-1 MUX

```
module mux32three(i0,i1,i2,sel,out);
input [31:0] i0,i1,i2;
input [1:0] sel;
output [31:0] out;
reg [31:0] out;

always @ (i0 or i1 or i2 or sel)
begin
    case (sel)
        2'b00: out = i0;
        2'b01: out = i1;
        2'b10: out = i2;
        default: out = 32'bx;
    endcase
end
endmodule
```

## 32-bit Adder

```
module add32(i0,i1,sum);
input [31:0] i0,i1;
output [31:0] sum;

assign sum = i0 + i1;

endmodule
```

## 32-bit Subtractor

```
module sub32(i0,i1,diff);
input [31:0] i0,i1;
output [31:0] diff;

assign diff = i0 - i1;

endmodule
```

## 16-bit Multiplier

```
module mul16(i0,i1,prod);
input [15:0] i0,i1;
output [31:0] prod;

// this is a magnitude multiplier
// signed arithmetic later
assign prod = i0 * i1;

endmodule
```

# 自底向上的实现

## ■ Given submodules:

```
module mux32two(i0,i1,sel,out);
module mux32three(i0,i1,i2,sel,out);
module add32(i0,i1,sum);
module sub32(i0,i1,diff);
module mul16(i0,i1,prod);
```

## ■ Declaration of the ALU Module:

```
module alu(a, b, f, r);
  input [31:0] a, b;
  input [2:0] f;
  output [31:0] r;
```

```
  wire [31:0] addmux_out, submux_out;
  wire [31:0] add_out, sub_out, mul_out;
```

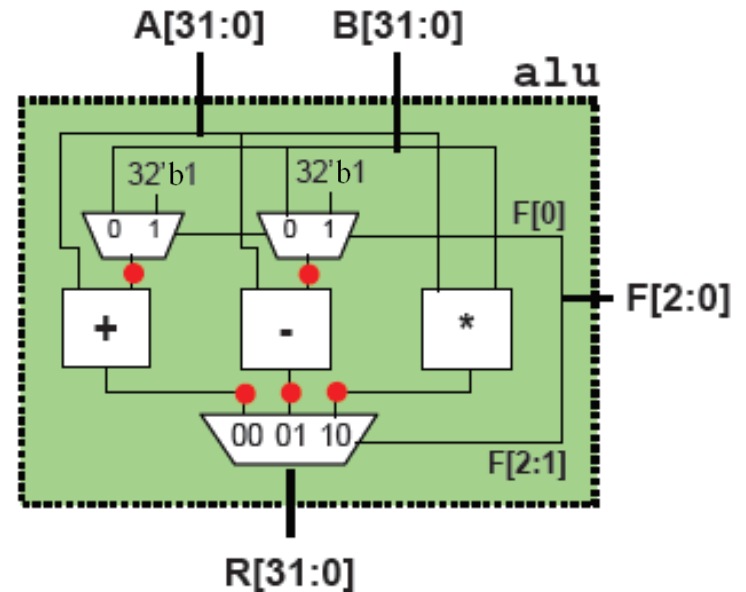
```
  mux32two    adder_mux(b, 32'b1, f[0], addmux_out);
  mux32two    sub_mux(b, 32'b1, f[0], submux_out);
  add32       our_adder(a, addmux_out, add_out);
  sub32       our_subtractor(a, submux_out, sub_out);
  mul16       our_multiplier(a[15:0], b[15:0], mul_out);
  mux32three  output_mux(add_out, sub_out, mul_out, f[2:1], r);
```

endmodule

module  
names

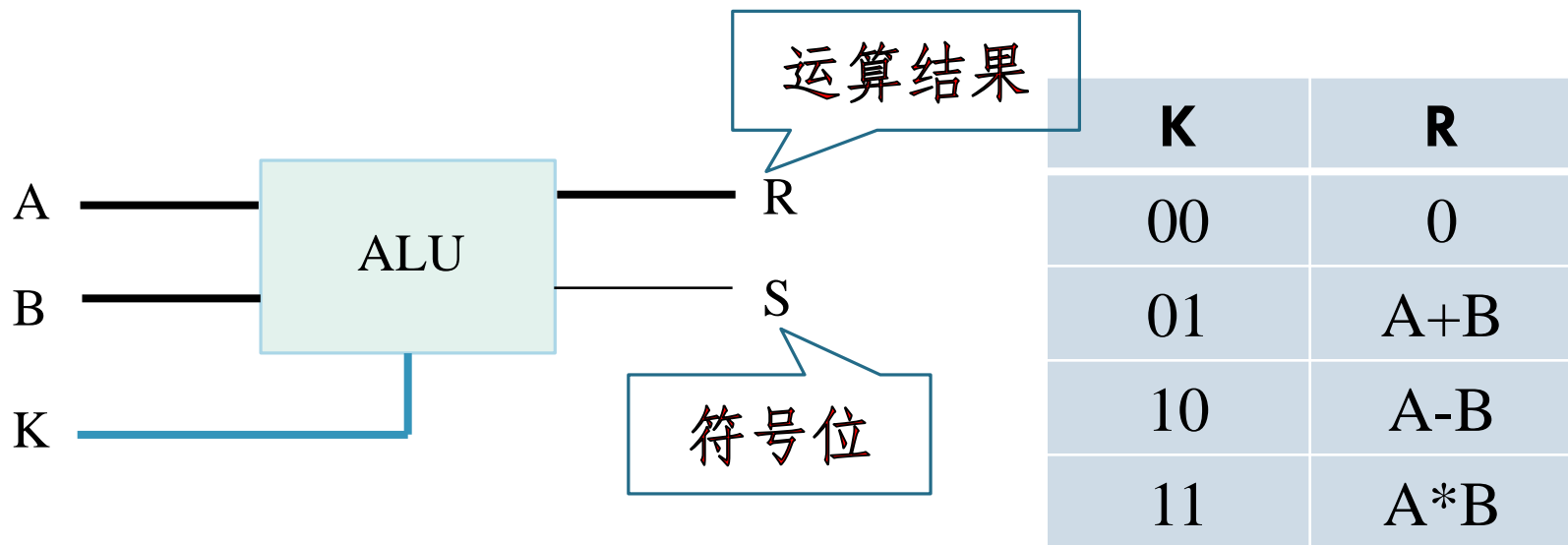
(unique)  
instance  
names

corresponding  
wires/regs in  
module alu

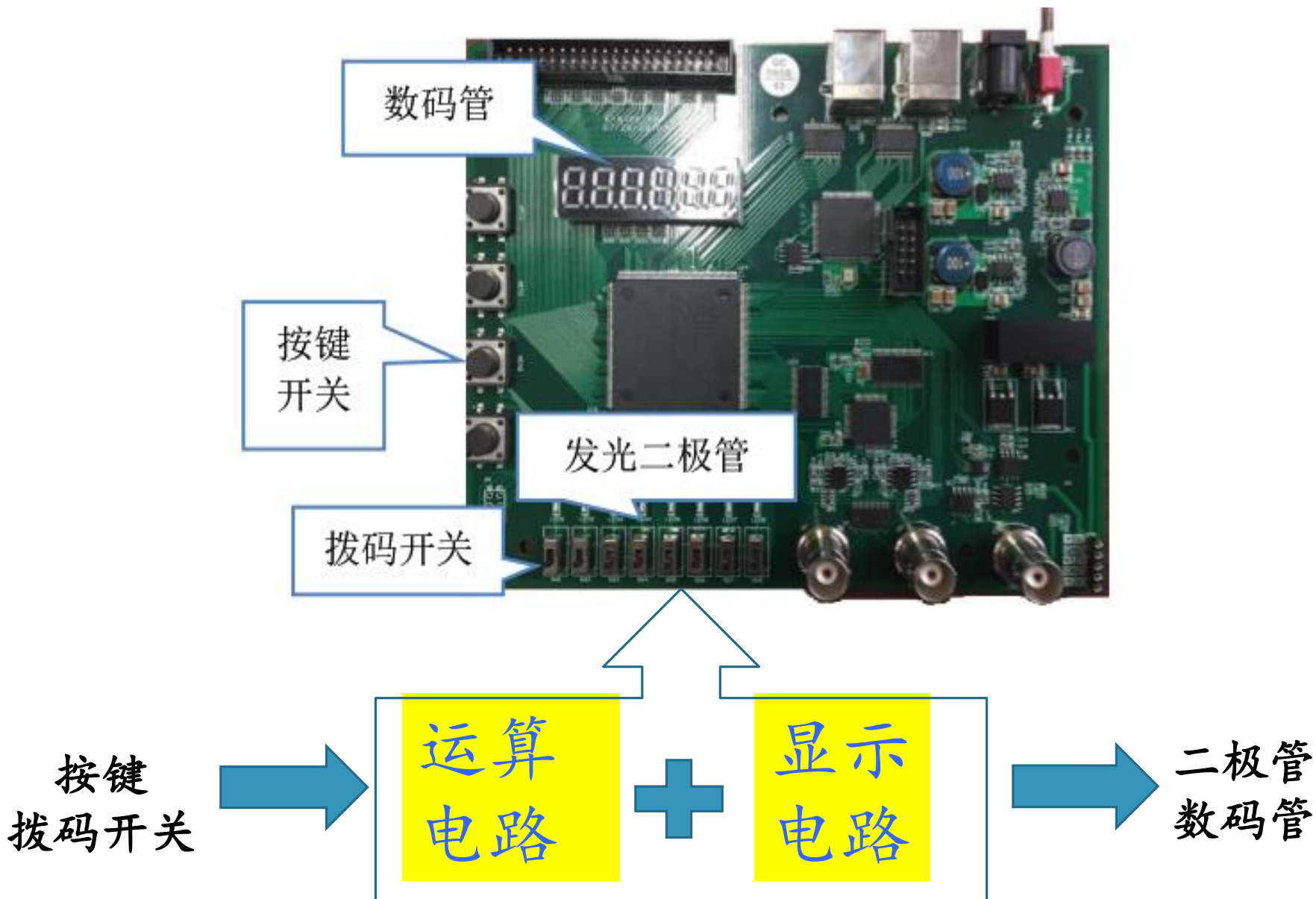


# 五、EDA实验一内容布置

基于FPGA实现一个简易计算器：



1. 其中A和B的取值范围为0~15；用实验板上的拨码开关和按键开关模拟输入；在数码管上以十六进制形式显示运算数和运算结果，负号用发光二极管显示。
2. 修改设计，要求数码管只显示运算数A和运算结果R，A用十六进制显示，R用十进制显示；



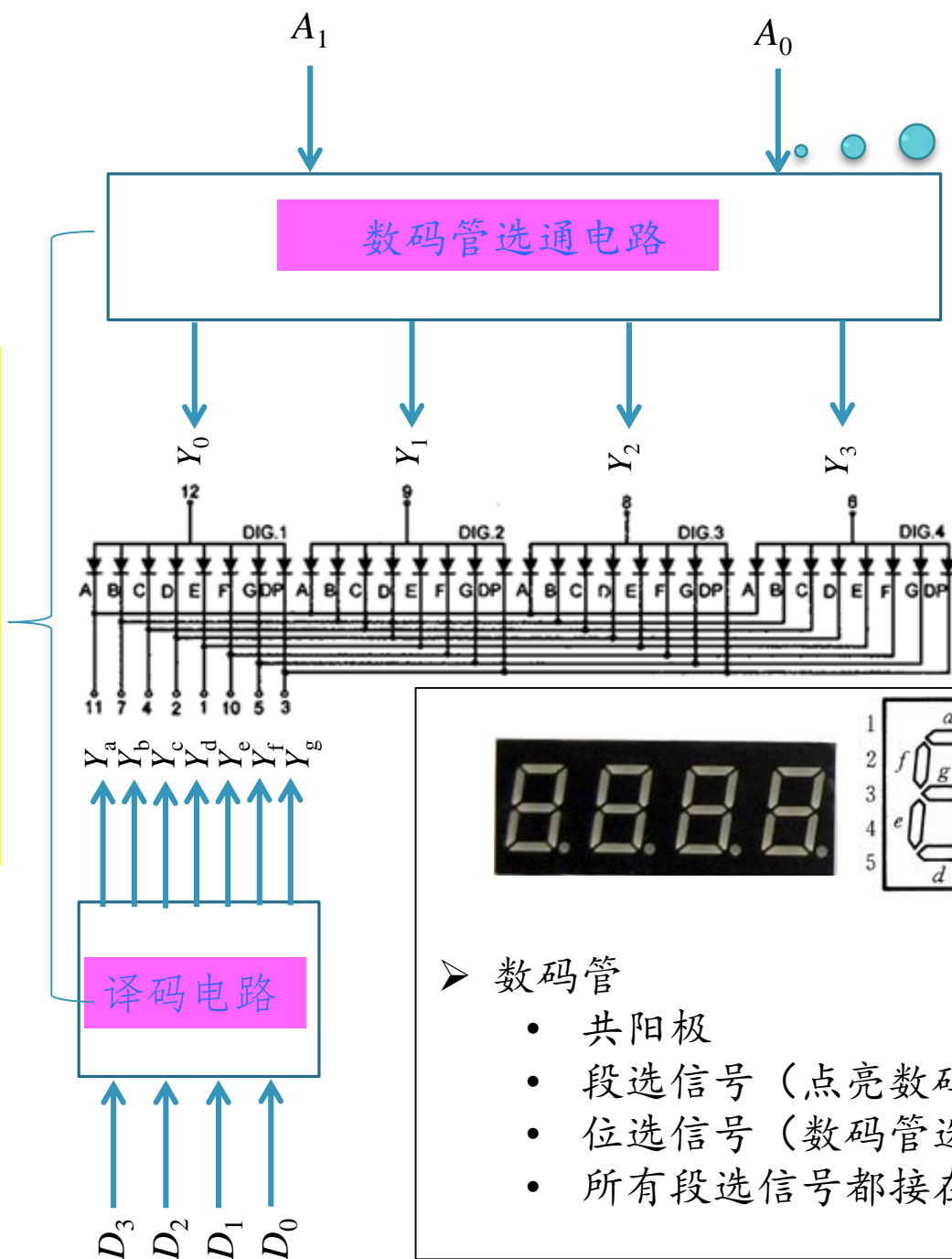


# 按键控制 数码管选 通

数码管自动  
轮流选通  
扫描频率  
**250Hz**



## 数码管显示电路



### ➤ 数码管

- 共阳极
- 段选信号（点亮数码管每一段）是低电平有效
- 位选信号（数码管选通信号）是低电平有效
- 所有段选信号都接在一起