

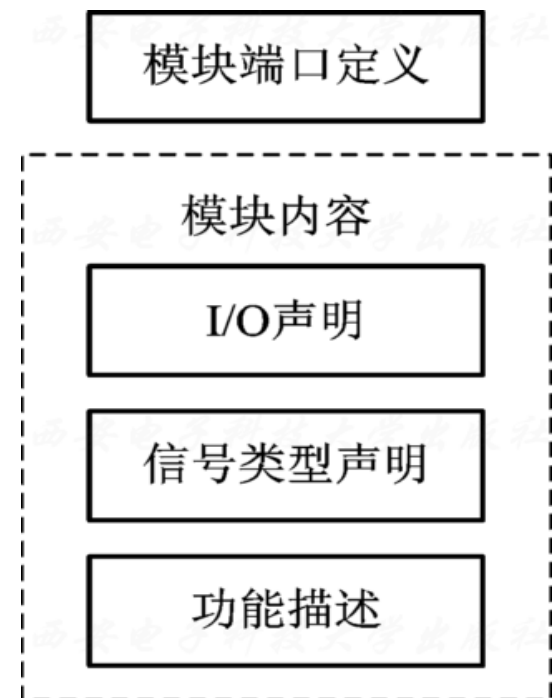
Part V Verilog HDL 及其编程

Lecture 15 硬件描述语言 Verilog HDL

一、Verilog HDL 程序的基本结构

Verilog HDL 程序由模块
(**module**)组成，模块的基本结构如
右图所示。

一个完整的模块由模块**端口定**



义和模块内容两部分组成，模块内容包括 I/O 声明、信号类型声明和功能描述。

例如，定义一个 1 位全加器 FullAdder:

```
module FullAdder(s,cout,a,b,cin);
```

```
input a,b,cin;
```

```
output s,cout;
```

```
assign {cout,s}=a+b+cin;
```

```
endmodule
```

模块的设计遵循以下规则：

①模块内容位于 **module** 和 **endmodule** 之间；
每个模块都有一个名字，即**模块名**，如 **FullAdder**，
模块名中可以包含英文字母、数字和下划线，并以
英文字母开头。

②除 **endmodule** 外，所有的**语句**后面必须有
分号 “;”。

③语句可以是**单条语句**，也可以是用 **begin** 和 **end** 两个保留字包围起来的由多条语句组成的**复合语句**。

④可以用“/*...*/”或“//...”对程序的任何部分作**注释**，增加程序的可读性和可维护性。

（一）模块端口定义

模块端口定义用来声明设计模块的**输入/输**

出端口，其格式如下：

module 模块名(端口 1, 端口 2, 端口 3, ...);

(二) 模块内容

模块内容用于对信号的 I/O 状态及信号类型进行声明，并描述模块的功能。

1、I/O 声明

模块的 I/O 声明用来声明各**端口信号**流动方

向,包括**输入(input)**、**输出(output)**和**双向(inout)**。

(1) 输入声明

如果信号**位宽**为 1, 那么声明格式为

input 端口 1, 端口 2, ...;

信号**位宽**大于 1, 那么声明格式为

input [msb:lsb] 端口 1, 端口 2, ...;

其中, **msb** 和 **lsb** 分别表示信号最高位和最低

位的编号。

(2) 输出声明

如果信号位宽为 1，那么声明格式为

output 端口 1, 端口 2, ...;

如果信号位宽大于 1，那么声明格式为

output [msb:lsb] 端口 1, 端口 2, ...;

(3) 输入、输出声明

如果信号位宽为 1，那么声明格式为

inout 端口 1，端口 2， ...;

信号位宽大于 1，那么声明格式为

inout [msb:lsb] 端口 1，端口 2， ...;

2、信号类型声明

信号类型声明用来说明电路的功能描述中所用信号的**数据类型**，常用的信号类型有**连线型**

(**wire**)、寄存器型 (**reg**)、整型 (**integer**)、实型 (**real**)、时间型 (**time**) 等。

3、功能描述

功能描述是 Verilog HDL 程序的主要部分，**用来描述模块内部结构和模块端口间的逻辑关系，在电路上相当于器件的内部结构。**

功能描述的常用方式：结构描述、数据流描述、

行为描述。

结构描述：利用**实例化语句**从基本**门级元件**开始逐级说明电路的结构。

例如，用实例化元件表示一个两输入的与门可以写为：and **u1**(q,a,b); //q=a&b

模块中每个实例化后的元件名称必须是唯一的。

数据流描述：利用**连续赋值语句 assign** 说明信号之间的运算以及信号之间的关系。

这种方式很简单，只要在 **assign** 后面加一个赋值语句即可，一般适合对组合逻辑进行描述，称为连续赋值方式。

例如，描述一个两输入的与门可写为：

```
assign a=b&c;
```

行为描述：利用过程语句 `always`、`initial` 和分支语句 `if...else...`、`case...`等说明

`always` 语句可以实现各种逻辑，常用于组合和时序逻辑的功能描述。

一个程序设计模块中可以包含一个或多个 `always` 块语句。程序运行中，在某些条件满足时，就重复执行 `always` 块中的语句。

例如, 一个带异步清除端的 D 触发器可写为:

```
always @(posedge clk or posedge clr),
```

```
begin,
```

```
    if (clr) q<=0;
```

```
    else q<=d;
```

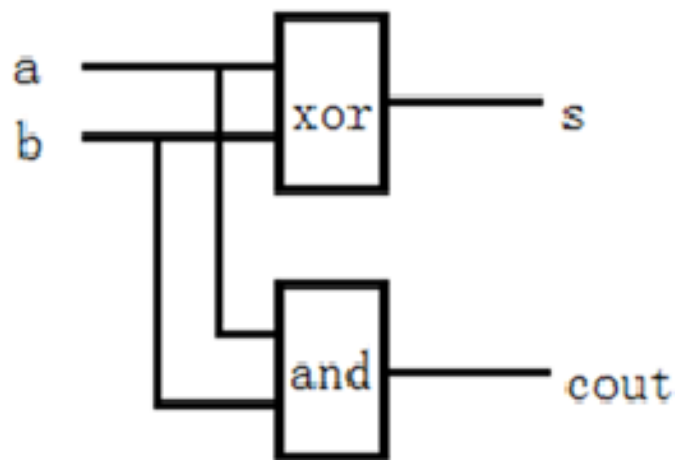
```
end
```

initial 语句与 always 块语句类似, 不过在程序中 initial 块语句只被执行一次, 常用于电路的初始化。

例如：设计一个 1 位半加器。

input		output	
a	b	s	<u>cout</u>
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$S = a \oplus b, \quad \text{cout} = a \& b$$



结构描述：

```
module HalfAdder(a,b,s,cout);  
  
  input a,b;  
  
  output s, cout;  
  
  xor myxor(s,a,b);  
  
  and myand(cout,a,b);  
  
endmodule
```

数据流描述：

```
module HalfAdder(a,b,s,cout);  
  
  input a,b;  
  
  output s, cout;  
  
  assign s=a^b;  
  
  assign cout=a&b;  
  
endmodule
```

行为描述:

```
module HalfAdder(a,b,s,cout);
```

```
    input a,b;
```

```
    output reg s, cout;
```

```
    always @(a,b) begin
```

```
        {cout,s}=a+b;
```

```
    end
```

```
endmodule
```

```
module HalfAdder(a,b,s,cout);
```

```
    input a,b;
```

```
    output reg s, cout;
```

```
    always @(a,b) begin
```

```
        case {a,b}
```

```
            2'b00:s=0;cout=0; 2'b01:s=1;cout=0;
```

```
            2'b10:s=1;cout=0; 2'b11:s=0;cout=1;
```

```
        endcase
```

```
    end
```

```
endmodule
```


4、并行执行

模块中的实例化语句、连续赋值语句、**always** 过程块语句都是并行执行的，与书写顺序无关。

当它们被综合器综合成实际电路时，会形成不同的电路块，当输入信号变化时，这些电路在满足延时的条件下同时动作，是并行执行的。

这是与高级语言最大的不同！

二、Verilog HDL 的数据类型

(一) 常量

Verilog HDL 支持三类常量。

1、逻辑常量

4 种逻辑值，实际电路只有前 3 种：

0: 逻辑 0

1: 逻辑 1

z/Z: 高阻态

x/X: 不确定

2、数值型常量

1) 整型常量

格式：<位宽>'<进制><数值>

位宽：位宽是对应的二进制宽度。

当定义的位宽比常数实际的位宽大时，在常数的左边自动填补 0，但如果常数的最左边一位是 x 或 z 时，那么就在左边自动填补 x 或 z；当定义

的位宽比常数实际的长度小时，在最左边的相应位就被截断。

进制：整型有四种进制形式：

2 进制（b 或 B）、10 进制（d 或 D）

8 进制（o 或 O）、16 进制（h 或 H）

数值：2 进制数值可以用四种基本的值来表示：

0：逻辑 0 或“假”；**1**：逻辑 1 或“真”；

x: 未知; **z**: 高阻

这四种值的解释都内置于语言中， 0 指逻辑 0， 1 指逻辑 1， z 指高阻抗， x 指逻辑不定值。x 和 z 都是不区分大小写的。

例如：6'B10X1Z0、5'O37、4'D98、7'H1A、8'h4Z

再如：-8'D76 //即-76，符号必须写在最前边

另外，整型常量还有两种表示方式：

'<进制><数值>

<数值>

在不指定位宽时，缺省位宽由机器系统决定，但至少为 32 位；如果数值中既无位宽，也无进制，则缺省为十进制数。

例如：'O35 // 位宽为 32 位的 8 进制数

'H67 // 位宽为 32 位的 16 进制数

92 //10 进制数 92

-100 //10 进制数 -100

2) 字符串型常量

字符串是用双引号括起来的字符序列，它必须写在同一行，不能分行书写。

字符串中的每个字符都是以其 ASCII 码进行存放的，一个字符串可以看作是 8 位的 ASCII 码值序

列。

另外，还存在一些特殊字符，这些特殊字符又称**转义字符**，用“\”来说明。

常用的特殊字符的表示及含义如下表：

特殊字符表示	含 义
\n	换行符
\t	制表符 Tab 键
\\	符号\
*	符号*
\ddd	3 位八进制数表示的 ASCII 码
%%	符号%

3、参数常量

可以用 **parameter** 来定义常量,即用一个标识符代表一个常量,称为**参数常量**或**符号常量**,这样可以增加程序的可读性和可维护性。

定义格式:

parameter 标识符 1=表达式 1, ...

例如: **parameter PI=3.14, A=8'B10110101;**

（二）变量

变量的数据类型很多，这里只对常用的几种变量类型进行介绍。

1、**wire** 型

wire 是网线数据类型之一，表示结构实体之间的物理连接。

网络类型的变量不仅不能储存值，而且必须

受到驱动器的驱动。如果没有驱动器连接到网络型的变量上，那么其值为高阻值。

网络型数据有很多种，但最常用的是 **wire** 型。

wire 型变量常用来表示以 **assign** 语句生成的组合逻辑信号，输入/输出信号在默认情况下自动定义为 **wire** 型。

wire 型信号可作为任何语句中的输入，也可

作为 `assign` 语句和实例化元件的输出。

定义格式：

```
wire [msb:lsb] 变量 1, 变量 2, ...;
```

其中，`wire` 是定义符；`msb` 和 `lsb` 分别表示 `wire` 型变量的最高位和最低位的编号，位宽由 `msb` 和 `lsb` 确定，如果不指定位宽，那么位宽自动默认为 1。

2、reg 型

reg 是寄存器类型，是数据存储单元的抽象，其对应的是具有状态保持功能的电路元件，如触发器、锁存器等。

reg 型变量只能在 `always` 和 `initial` 块中被赋值，通过赋值语句改变 reg 型变量的值，若 reg 型变量未被初始化，则其值为未知值 `x`。

`reg` 型变量与 `wire` 型变量的区别是：`wire` 型变量需要持续地驱动，而 `reg` 型变量保持最后一次的赋值。

定义格式：

```
reg [msb:lsb] 变量 1, 变量 2, ...;
```

其中，`reg` 是定义符；`msb` 和 `lsb` 分别表示 `reg` 型变量的最高位和最低位的编号，位宽由 `msb` 和

lsb 确定，如果不指定位宽，则自动默认为 1。

3、memory 型

memory 型是存储器型，是通过建立 reg 型数组来描述的，可以描述 RAM 存储器、ROM 存储器和 reg 文件。

定义格式：

reg [msb:lsb] 存储单元 [n:m];

其中，`[n:m]` 表示存储单元的编号范围。

例如：`reg memory1 [1023:0];`

`reg [7:0] memory2 [15:0];`

`reg [32:1] memory2 [1:512];`

对存储单元的访问可以通过数组的索引进行。

4、integer 型

integer 型是 32 位带符号整型变量，用于对循

环控制变量的说明，典型应用是高层次的行为建模，它与后面的 **time** 和 **real** 类型一样是不可综合的。也就是说，这些类型是纯数学的抽象描述，不与任何物理电路相对应。

定义格式：

integer 变量 1，变量 2，...，变量 n；

5、**time** 型

time 类型用于存储和处理时间，是 64 位无符号数，定义格式：

time 变量 1，变量 2，...，变量 n；

6、real 型

real 型是 64 位带符号实型变量，用于存储和处理实型数据，定义格式：

real 变量 1，变量 2，...，变量 n；

三、Verilog HDL 的运算符

1、算术运算符

算术运算符包括： $+$ （加法运算符或正值运算符）、 $-$ （减法运算符或负值运算符）、 $*$ （乘法运算符）、 $/$ （除法运算符）、 $\%$ （取模运算符）

2、逻辑运算符

逻辑运算符包括： $\&\&$ （逻辑与）、 $||$ （逻辑或）、 $!$

(逻辑非)

逻辑运算符操作的结果为 0 (假) 或 1 (真)。

在判断一个数是否为真时，以 0 代表“假”，以非 0 代表“真”。

3、关系运算符

关系运算符包括：< (小于)、<= (小于等于)、> (大于)、>= (大于等于)

关系运算符是用来确定两个操作数之间的关系是否成立的，如果成立，结果为 **1**（真）；如果不成立，结果为 **0**（假）。

4、等值运算符

等值运算符包括：**==**（逻辑相等）、**!=**（逻辑不等）、**===**（全等）、**!==**（非全等）

“**==**”和“**!=**”的运算结果可能为 **1**、**0** 或 **x**，

而“==”和“!=”的运算结果只有两种：1 或 0。

5、位运算符

位运算符包括： \sim （非）、 $\&$ （与）、 $\sim\&$ （与非）、 $|$ （或）、 $\sim|$ （或非）、 \wedge （异或）、 $\wedge\sim$ 或 $\sim\wedge$ （同或）

位运算符是对两个操作数按位进行逻辑运算的。当两个操作数的位数不同时，自动在位数较少的操作数的高位补 0。

6、缩减运算符

缩减运算符包括： $\&$ （与）、 $\sim\&$ （与非）、 $|$ （或）、 $\sim|$ （或非）、 \wedge （异或）、 $\wedge\sim$ 或 $\sim\wedge$ （同或）

缩减运算符与逻辑运算符的法则一样，但缩减运算符是对单个操作数按位进行逻辑递推运算的，运算结果为 1 位二进制数。

例如：reg [7:0]a; reg b; b=&a;

7、移位运算符

移位运算符包括：<<（左移）、>>（右移）

左移和右移运算符是对操作数进行逻辑移位操作的，空位用 0 进行填补。

移位运算的格式为： $a \ll n$ 或 $a \gg n$

其中， a 为操作数； n 为移位的次数。

8、条件运算符

条件运算符是： $?:$ ，是唯一的一个三目运算符，即条件运算符需要三个操作数。

9、拼接运算符


拼接运算符是： $\{ \}$ ，拼接运算符用来将两个或多个数据的某些位拼接起来。

拼接运算符格式如下：

{数据 1 的某些位，数据 2 的某些位，...}

例如：X={a [7:4], b [3], c [2:0] }

10、运算符的优先级

运 算 符	优 先 级
<div>+ (正) - (负) ! ~ * / % + (加) - (减) << >> < <= > >= == != === !== & ~& ^ ^~ ~^ ~ && ?:</div>	<div>高优先级  低优先级</div>

四、Verilog HDL 常用建模方法

（一）门级建模

门级建模即结构描述，也就是通过实例化预先定义好的各种功能模块来说明电路结构。

为此，系统提高了基本的门级元件。

例如：设计一个 3 人表决器。

```
module voter(a, b, c, result);
```

```
input a, b;
```

```
output result;
```

```
wire t1, t2, t3, t4;
```

```
and myand1(t1,a,b);
```

```
and myand2(t2,a,c);
```

```
and myand3(t3,b,c);
```

```
or myor1(t4,t1,t2);
```

```
or myor2(result,t3,t4);
```

```
endmodule
```

（二）数据流建模

数据流建模即**数据流描述**，也就是利用连续赋值语句用来驱动 **wire** 型变量，这一变量必须先定义过。使用连续赋值语句时，要输入端操作数的值发生变化，该语句就重新计算并刷新赋值结果。

连续赋值语句用来描述组合逻辑。

格式: **assign wire** 型变量名=表达式;

wire 变量名=表达式;

含义: 只要右边赋值表达式中有变量发生变化, 就重新计算表达式的值, 新结果在指定的延时时间单位以后赋值给 **wire** 型变量。如果不指定延时量, 则延时量默认为 0。

例如：设计一个 3 人表决器

```
Module voter(a,b,c,result);
```

```
Input a,b,c;
```

```
Output result;
```

```
Assign result=a&b|a&c|b&c;
```

```
Endmodule
```

```
module decoder(a,y);
```

```
input a[1:0];
```

```
output y[3:0];
```

```
assign y[0]=~a[1]&~a[0]
```

```
assign y[1]=~a[1]&a[0];
```

```
assign y[2]=a[1]&~a[0];
```

```
assign y[3]=a[1]&a[0];
```

```
endmodule
```

再如：设计一个高电平有效的 2-4 译码器。

（三）行为建模

1、过程语句

1、initial

initial 语句常用于对各变量的初始化。一个程序模块中可以有多多个 initial 语句，所有 initial 语句在程序一开始时同时执行，并且只执行一次。

格式：**initial** 语句

例如：

```
Initial Begin
```

```
reset = 1; #3 reset = 0; #5 reset = 1;
```

```
end
```

2、always

与 initial 语句一样，一个程序中可以有多个 **always** 语句，always 语句也是在程序一开始时立即被执行的，不同的是 always 语句不断地重复运

行。但 **always** 语句后跟的语句是否执行要看其敏感事件列表是否满足，若有条件满足，则运行一次语句。

格式： **always @(敏感事件列表) 语句**

always 语句后面是一个敏感事件列表，该敏感事件列表的作用是激活 **always** 语句执行的条件，敏感事件可以是电平触发，也可以是边沿触发。

电平触发的 `always` 块常用于描述组合行为，而边沿触发的 `always` 块常用于描述时序行为。

`always` 语句后面的敏感事件可以是单个事件，也可以是多个事件，多个事件之间用 `or` 连接。

如果敏感事件是电平信号，那么直接列出信号名；如果敏感事件是边沿信号，那么可分为上升沿和下降沿，上升沿触发的信号前加关键字 `posedge`，下降沿触发的信号前加关键字 `negedge`。

例如：设计一个 3 人表决器。

```
Module voter(a,b,c,result);
```

```
Input a,b,c;
```

```
Output reg result;
```

```
Always @(a,b,c) begin
```

```
Case ({a,b,c})
```

```
0:result=0; 1:result=0; 2:result=0; 3:result=1;
```

```
4:result=0; 5:result=1; 6:result=1; 7:result=1;
```

```
□
```

```
Endcase
```

```
End
```

```
endmodule
```

例如：用 clk 的上升沿使 count 加 1

```
reg [7:0] count;  
  
always @(posedge clk) begin  
    count=count+1b'1;  
  
end
```

如果要用 clk 的下降沿使 count 加 1，只需将程序中的敏感事件改为 negedge clk 即可。

例如，一个电平敏感型锁存器的程序如下：

```
module latch(enable,data,q)
    input enable;
    input [7:0] data;
    output reg [7:0] q;
    always @(enable,data) begin
        if (enable) q<=data;
    end
end
```

2、过程赋值语句

过程赋值语句是在 **initial** 或 **always** 语句块内赋值的，用于对 reg 型、memory 型、integer 型、time 型和 real 型变量赋值，这些变量在下一次过程赋值之前保持原来的值。

过程赋值语句分为两类，分别为阻塞赋值和非阻塞赋值。

(1) 阻塞赋值

阻塞赋值的赋值符为“=”，它在该语句结束时就完成赋值操作，格式：变量=赋值表达式；

(2) 非阻塞赋值

非阻塞赋值的赋值符为“<=”，它在块结束时才完成赋值操作，格式：变量<=赋值表达式；

阻塞赋值与非阻塞赋值的主要区别是：一条

阻塞赋值语句执行时，下一条语句被阻塞，即只有当一条语句执行结束，下条语句才能执行；非阻塞赋值语句中，各条语句是同时执行的。也可以理解为，阻塞赋值是串行执行的，非阻塞赋值是并行执行的。为了理解这两种赋值，下面分析两段程序的执行过程。

程序段 1: `begin r1=2; r2=r1; r3=r2; end`

程序段 2: `begin r1<=2; r2<=r1; r3<=r2; end`

3、条件语句

1) if...else 语句

Verilog HDL 提供了三种形式的 if 语句。

(1) **if** (表达式) 语句

(2) **if** (表达式) 语句 1

else 语句 2

(3) **if** (表达式 1) 语句 1

else if (表达式 2) 语句 2

⋮

else if (表达式 m) 语句 m

else 语句 n

2) case 语句

Verilog HDL 语言中的 case 语句可以直接处理多分支选择，格式：

分支表达式和各分支项表达式不必都是常量表达式。在 case 语句中，x 值和 z 值作为文字值进行比较。

Case（控制表达式）

表达式 1：语句 1

表达式 2：语句 2

⋮

表达式 m：语句 m

Default：语句 m+1

endcase

例如：设计一个 3:8 译码器。

```
module seg (SW, Q);
    input [2:0] SW;
    output [7:0] Q;
    reg [7:0] Q;
    always @(SW) begin
        case (SW)
            3'b000: Q = 8'b11111110;
            3'b001: Q = 8'b11111101;
            3'b010: Q = 8'b11111011;
            3'b011: Q = 8'b11110111;
            3'b100: Q = 8'b11101111;
            3'b101: Q = 8'b11011111;
            3'b110: Q = 8'b10111111;
            3'b111: Q = 8'b01111111;
        Endcase
    end
endmodule
```

4、循环语句

Verilog HDL 中有四类循环语句：Forever、repeat、while、for

1) forever

forever 格式：**forever** 语句

forever 常用于产生周期性的波形，例如：

Initial

Begin

clock = 0;

5 forever #10 clock = ~ clock;

end

2) repeat

repeat 用于指定次数的循环，格式：

repeat (表达式) 语句

repeat 中的表达式通常为常量表达式,表示循环的次数。如果循环计数表达式的值不确定,即为 **x** 或 **z** 时,那么循环次数按 **0** 处理。

例如: 计算 $S=1+2+3+4+\dots+100$ 。

Initial Begin

`s = 0; i = 1;`

`repeat (100) begin s = s + i; i=i+1; end`

`end`

3) while

while 循环到指定的条件为假，格式：

while（条件）语句

如果条件为 x 或 z，则按 0
（假）处理。

例如：对 256 个存储单元
初始化。

```
Reg [7:0] memory [0:255];
```

```
Initial Begin
```

```
reg [7: 0] i; i=0;
```

```
while (i<=255) begin
```

```
memory [i]=0; i=i+1;
```

```
end
```

```
end
```


4) for

按照指定的次数重复执行过程赋值语句。

格式：

for（初值表达式； 条件；

循环变量增值） 语句

例如： 用加法和移位实现

$x*y$ 。

```
Reg [15:0] x,y;
```

```
Reg [31:0] s;
```

```
Initial Begin
```

```
reg [3:0] l; s=0;
```

```
for (i=0;i<=15;i=i+1)
```

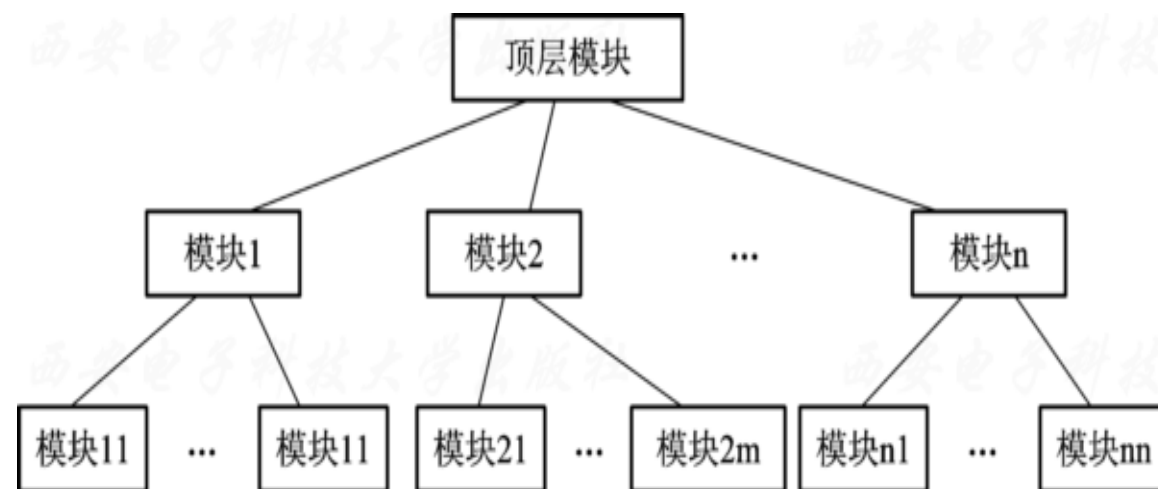
```
if (y[i]) s=s+(x<<i);
```

```
end
```

五、模块化电路设计

为了便于设计，可以将一个大的系统分层次、分模块进行设计。

一个**工程**中可以设计多个模块，其中



有一个**主模块**称为**顶层模块**，其他模块称为**子模块**，每个模块存放在一个文件中，上层模块可以调用下层模块。

在模块化设计中，上层模块对下层进行调用，每调用一次模块，被调用模块的电路就被复制一次，相当于在上层模块中生成了一个**实例**，因此，被调用的模块被称为**实例模块**或实例部件，上层模块对实例模块的调用称为模块的**实例化**。

上层模块与实例模块之间是通过端口列表进行连接的。

1、函数

类似高级语言中的函数，用 **function** 定义。

定义格式：

```
function 〈返回值的类型或范围〉 函数名  
    端口声明语句  
    类型声明语句  
    语句  
endfunction
```

调用格式： 函数名 （端口名列表）；

【例如】用 8 位高低位交换电路构造 16 位高低位交换电路。

```
module swapper16(word16,rsword16);  
  
    input [15:0] word16;  
  
    output reg [15:0] rsword16;  
  
    always @( word16)  
  
        begin  
  
            rsword16[15:8]=rsword8(word16[7:0]);  
  
            rsword16[7:0]=rsword8(word16[15:8]);  
  
        end
```

```
function[7:0] rsword8;  
  
    input[7:0] word8;  
  
    integer k;  
  
    for (k=0;k<8;k=k+1) rsword8[k]=word8[7-k];  
  
endfunction  
  
endmodule
```

函数的每一次调用都被综合成一个独立的组合逻辑电路。

2、任务

类似于高级语言中的过程，用 **task** 定义。

定义格式：

```
task 任务名;  
    端口声明语句  
    类型声明语句  
    语句  
endtask
```

调用格式：任务名（端口名列表）；

【例如】 利用 8 位全加器实现 16 位全加器。

```
module adder16(a, b, cin, sum, cout);
```

```
    input [15:0] a, b;
```

```
    input cin;
```

```
    output reg [15:0] sum;
```

```
    output reg cout;
```

```
    reg c8;
```

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

```
        begin
```

```
            adder8(a[7:0], b[7:0], cin, sum[7:0], c8);
```

```
            adder8(a[15:8], b[15:8], c8, sum[15:8], cout);
```

```
        end
```

```
task adder8;
```

```
    input [7:0] ta, tb;
```

```
    input tcin;
```

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

```
    output tcout;
```

```
    { tcout, tsum }=ta+tb+tcin;
```

```
    endtask
```

```
endmodule
```