杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

《计算机组成原理课程设计》

第3章 Verilog HDL基础

主讲教师：

第3章 Verilog HDL基础

3.1 Verilog HDL概述

- 1、数字电路的设计方法
- 2、Verilog HDL程序结构

1、数字电路的设计方法

❖（1） 自下而上的设计方法

- 从基本单元出发，对设计进行逐层划分的过程

❖（2）自上而下的设计方法

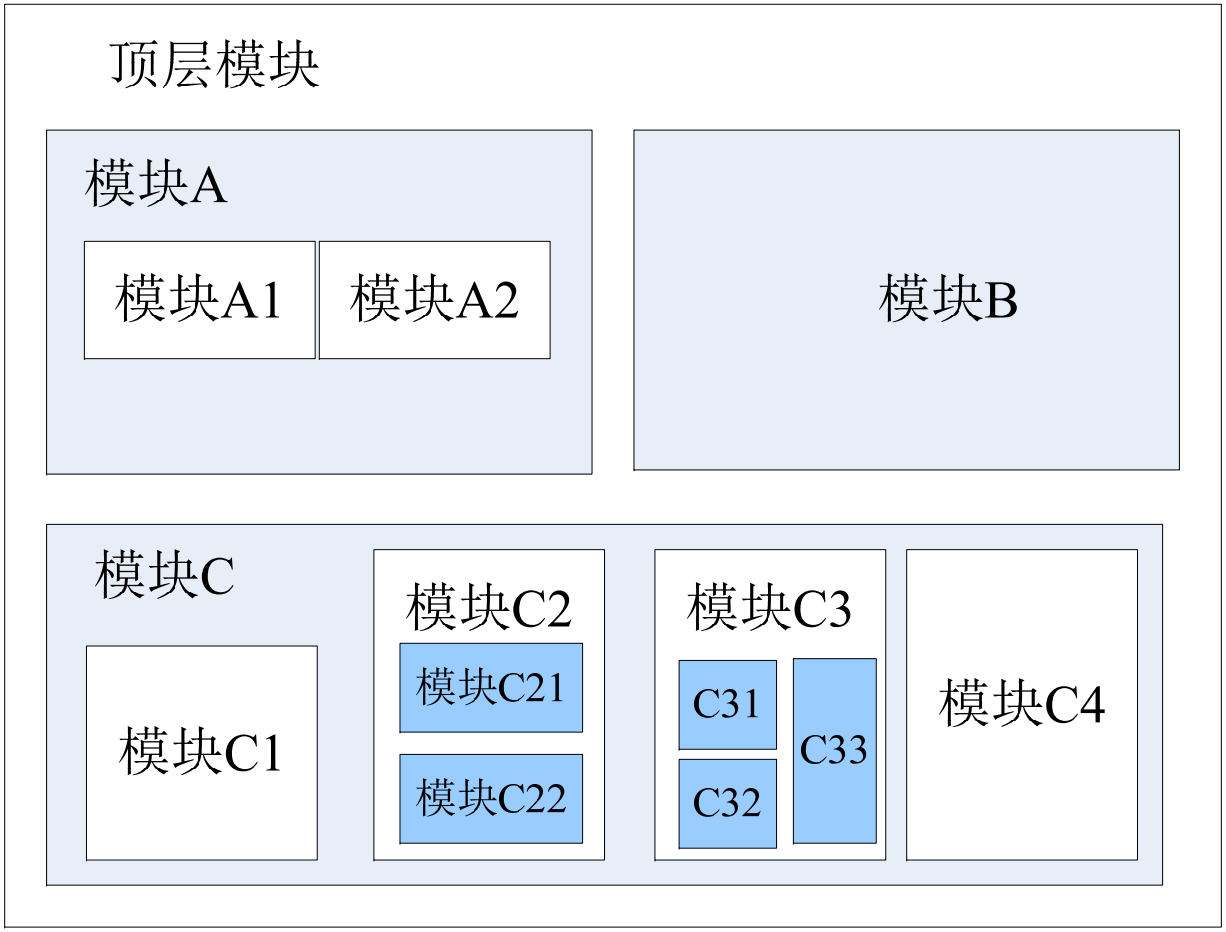
- 从系统级开始，把系统划分为基本单元，然后再把基本单元划分为下一层次的基本单元，直到可用**EDA**元件实现为止。

❖（3）混合的设计方法

- 复杂数字逻辑电路和系统设计过程，通常是以上两种设计方法的结合。
- 在高层系统用自上而下的设计方法实现，而使用自下而上的方法从库元件或设计库中调用已有的设计单元。

2、Verilog HDL程序结构

❖ Verilog HDL程序就是模块的集合：



2、Verilog HDL程序结构

- 模块：代表一个基本的功能块，一个模块可以是一个元件，也可以是低层次模块的组合。
  - 模块通过接口（输入和输出端口）被高层的模块调用，但隐藏了内部的实现细节。
  - 使得设计者可以方便地对某个模块内部进行修改，而不影响设计的其他部分。
- 使用Verilog HDL完成某个数字电路设计的过程，其实就是一个个模块的程序设计过程。

3.2 Verilog HDL的模块

1、模块的结构

2、模块的声明与内容

3、模块实例与调用

4、时间单位与时延

1、模块的结构

- 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其与其他模块通信的外部端口。
- 一个模块可以在另一个模块中调用。
- 模块是并行运行的，通常需要一个高层模块通过调用其他模块的实例来定义一个封闭的系统，包括测试数据和硬件描述。

1、模块的结构

❖ **模块的结构：**

❖ **关键字：**

- **module**
- **endmodule**

module 模块名(端口列表)

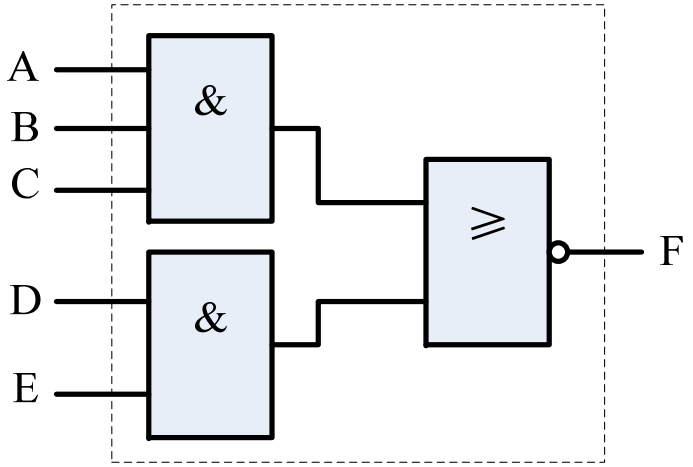
端口定义	input 输入端口 output 输出端口 inout 双向端口
数据类型声明 变量定义	wire reg parameter task function
逻辑功能定义	低层模块实例 数据流语句(assign) 行为描述语句(always) 激励语句(initial) 开关级/门级原语 UDP原语

endmodule

1、模块的结构

❖举例：

$$F = \overline{ABC + DE}$$




```
module  First_M (A, B, C, D, E, F ); //端口名和端口列表
  input          A, B, C, D, E;    //声明输入端口；
  output         F;                //声明输出端口；
  wire           A, B, C, D, E, F; //声明端口的数据类型
       assign  F = ~ (( A & B &C) | ( D &E ));
               //逻辑功能描述
endmodule
```

2、模块的声明与内容

❖完整的Verilog模块由4个部分组成：

- （1）模块声明
- （2）端口类型定义
- （3）数据类型声明和变量定义
- （4）逻辑功能描述

2、模块的声明与内容

❖ （1）模块声明

```
module 模块名(端口名1,端口名2,……端口名n);
   ……                    //其他语句
endmodule                 //模块结束关键字
```

- 每个模块必须有一个模块名，惟一地标识这个模块。
- 端口列表用于描述这个模块的输入和输出端口，它是可选的——可以没有端口列表。
- 对模块进行调用时，按端口列表定义分别连接到模块实例的端口列表信号。

❖ 在Verilog中，不允许在模块声明中嵌套模块。

2、模块的声明与内容

❖ （2）端口类型定义

- 端口是指模块与外界或其他模块进行连接、通信的信号线，它是模块与外界环境交互的接口，就好像IC芯片的输入、输出引脚。
- 对于外界来说，模块内部是不可见的，对模块的调用只能通过模块实例的端口进行。
- 端口定义用于指明端口列表中每一个端口的类型：
  - 输入端口：输入引脚
  - 输出端口：输出引脚
  - 双向端口，双向引脚

2、模块的声明与内容

❖ （2）端口类型定义

```
input   [位宽]  输入端口名1, 输入端口名2,……输入端口名n;
output  [位宽]  输出端口名1, 输出端口名2,……输出端口名n;
inout   [位宽]  双向端口名1, 双向端口名2,……双向端口名n;
```

- ❖ 位宽的格式是[high:low]，high指明最高位的序号，low指明最低位的序号，它们都是常量，low一般是1或者0。
- ❖ 位宽省略时，表明是默认位宽为1；
- ❖ 不同位宽的端口要分别定义。

2、模块的声明与内容

❖（2）端口类型定义

- 举例：

```
input[3:0]        a, b;
    //a、b是位宽为4位的输入端口,a[3]是高位,a[0]为低位
input             cin;      // cin是输入端口，默认位宽为1位
output[4:1]       sum;
    // sum是4位的输出端口,sum[4]是高位,sum[1]为低位
inout [31:0]      bus;      //bus是31位的双向端口
```

2、模块的声明与内容

❖（3）数据类型声明和变量定义

- 模块描述中使用的所有信号（包括端口、寄存器和中间变量等）都必须要事先定义，声明其数据类型后方可使用。
- Verilog HDL有19种数据类型，最常用的是：
  - 线网型（wire）
  - 寄存器型（reg）
  - 参数型（parameter）
- 变量、寄存器、信号和参数等的声明部分必须在使用前出现 。
- 端口的数据类型声明缺省时，EDA综合器将其默认为wire型。

2、模块的声明与内容

❖ （3）数据类型声明和变量定义

- 举例：

```
reg [3:0]  a,b;    //定义a和b的数据类型为4位的寄存器型reg
wire       cin, cout;          //cin、cout的数据类型为1位线网型wire
parameter x=8, y=x*2;  //定义常量x为8，常量y为16
```

2、模块的声明与内容

❖ （4）逻辑功能描述

- 它是一个模块中最重要的部分，用于描述模块的行为和功能、子模块的调用和连接、逻辑门的调用、用户自定义部件的调用、初始态赋值initial、always块、连续赋值语句assign等等。
- 在Verilog模块中，可用4种方式描述其逻辑功能：
  - 结构描述方式：可使用开关级原语、门级原语和用户定义的原语方式描述；
  - 数据流描述方式：使用连续赋值语句assign进行描述；
  - 行为描述方式：使用过程结构描述时序行为。
  - 3种描述方式的混合

3、模块实例与调用

- 模块的声明类似于一个模板，使用这个模板就可以创建实际的对象。
- 创建对象方法：调用模块
- 从模板创建对象的过程，称为实例化；创建的对象称为模块实例。
- 模块间的相互调用就是通过引用实例来完成的。

```
//顺序端口连接
模块名称 实例名称(实例端口1, 实例端口2,……实例端口n);
//命名端口连接
模块名称 实例名称( .模块端口1(实例端口1), .模块端口2(实例端口2),
……,.模块端口n(实例端口n));
```

3、模块实例与调用

❖ 【例3.2】：使用【例3.1】的模块First_M创建一个实例MyFirst_UUT。

```
module      First_M _Inst;              //创建实例的模块
    wire    a, b, c, d, e, f            //声明6个线网信号
    First_M  MyFirst_UUT (a, b, c, d, e, f);
endmodule
```

顺序端口连接，默认a连接到A，b连接到B，依次类推，f连接到F

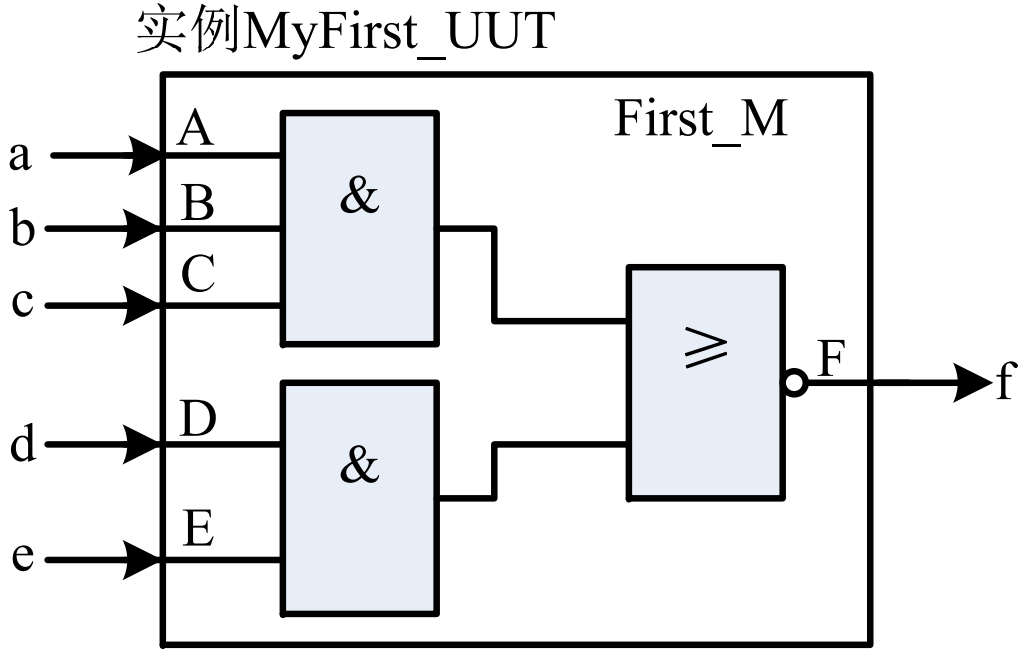
```
module  First_M _Inst;      //创建实例的模块
  reg   a, b, c, d, e;      //声明5个reg信号
  wire  f;                  //声明1个wire信号
  First_M          MyFirst_UUT (
          .F(f) ,
          .A(a),
          .B(b),.C(c), .D(d),.E(e),
          );
endmodule
```

命名端口连接，可以不按声明的顺序

连接实例的输入端口A到reg变量a

3、模块实例与调用

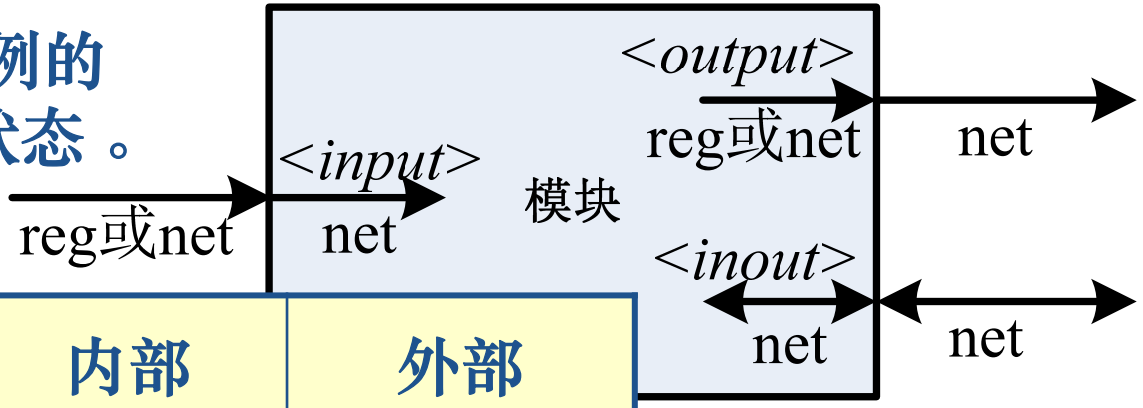
- 模块声明相当于定义了一个电路，而实例引用（即调用模块）就相当于又复制了一个和模块相同的电路，然后将其端口信号与外部信号连接（实例端口）。
- 模块实例MyFirst_UUT和模块First_M的关系



3、模块实例与调用

❖当一个模块中调用（实例引用）另一个模块时，端口之间的连接必须遵守一些规则，否则会报错。

Verilog允许模块实例的端口保持未连接的状态。



	内部	外部
输入端口input	net	net、reg
输出端口output	net、reg	net
输入输出端口inout	net	net

4、时间单位与时延

- Verilog HDL模型中，所有时延都根据单位时间来定义。
  - 编译器指令`timescale：用于定义时延的单位和时延精度，将时间单位与实际时间相关联。
  - 指令格式为：

`timescale 时延单位 / 时间精度

  - 时延单位和时间精度：由值1、10、和100以及单位s、ms、us、ns、ps和fs组成。
  - 举例：
    - `timescale 1ns/100ps /*时延单位为1ns，时延精度为100ps，即所有的时延必须被限定在0.1ns内*/
    - `timescale 10ns/1ns //时延单位为10ns，时延精度为1ns

4、时间单位与时延

- `timescale编译器指令需在模块描述前定义，并且影响后面所有的时延值。

```
`timescale 1ns/100ps
assign #2 Sum = A ^ B;
// #2指2个时间单位，即2ns
```

A、B发生变化后延时2时间单位执行赋值

```
`timescale 10ns/1ns
assign #2 Sum = A ^ B;
// #2指2个时间单位，即20ns
```

3.3 词法约定

- 与C语言类似，语言要素包括以下内容:
  - （1）标识符
  - （2）关键字
  - （3）注释
  - （4）格式

3.3 词法约定

❖ （1）标识符

- 以字母或者下划线“_”开头的，由字母、数字、$符号和下划线“_”组成的字符串。
- 标识符是区分大小写的。
- 以“$”为开头的标识符是为系统函数保留的，不能做普通标识符的起始字符。

合法标识符：
Sum
SUM　//与Sum不同，区分大小写
_Row_1
C79_F$

非法标识符：
$C79_F　//不能以$开头
356_Day　//不能以数字开头
and　//and是关键字
S-T*　//不能含有-和*等其他字符

3.3 词法约定

❖ （2）关键字

- 即保留字，用于特定的上下文，代表一定的特殊含义
- 只有小写的关键字才是保留字。
  - always: 关键字
  - ALWAYS：非关键字，用户定义的标识符
- 用户在定义模块名、端口名、变量名、参数名时，不能与关键字相同。

3.3 词法约定

❖ （2）Verilog的常见关键字

分类	常见关键字
定义	module endmodule input output inout function endfunction task endtask
数据类型	wire reg parameter integer signed
语句	assign always if else begin end case casex casez endcase default for posedge negedge
内置原语	and or not nand nor xor xnor notif0 notif1 bufif0 bufif1

3.3 词法约定

❖ （3）注释

- 单行注释：以“//”开始，忽略从“//”到行尾的内容。
- 多行注释：以“/*”开始，结束于“*/”，允许注释有多行。
- 多行注释不允许嵌套，但是允许单行注释可以嵌套在多行注释中。

■ 举例：

合法的注释：

```
a = b &&c;      //这里是单行注释
/*  这里可以
写多行注释 */

/*  这是
a=b|c           //合法的嵌套注释
*/
```

非法的注释：

```
/*  这是 /*不合法*/  的注释 */
```

因为有/**/嵌套

```
92  /* wire X,Y,Z;
93  /* reg G,H;*/
94  */
```

ERROR : /* in comment
ERROR : */ outside comment

```
ERROR:HDLCompiler:114 - "D:\FPGA\FPGAJX\Demo\M_Inst\M_nand.v" Line 93: /* in comment
ERROR:HDLCompiler:112 - "D:\FPGA\FPGAJX\Demo\M_Inst\M_nand.v" Line 94: */ outside comment
```

3.3 词法约定

❖ （4）格式

- 区分大小写，即大小写不同的标识符是不同的；但是关键字都是小写的。
- Verilog HDL是自由格式的，即结构可以跨越多行编写，也可以在一行内编写。
- 空白符：用于分隔标识符，包括空格（\b）、制表符（\t）和换行符（\n）
  - 空白符没有特殊意义（除了在字符串中），在编译阶段被忽略

3.3 词法约定

❖ （4）格式

```
initial begin Top = 3'b001; #2 Top = 3' b011; end   //一行书写完毕
```

等价于

```
initial
  begin
     Top = 3'b001;
     #2 Top = 3' b011;
  end
```

3.4 数据类型

1、常量

2、变量

3、标量与向量

4、参数

1、常量

- Verilog HDL的信号值集合：
  - 1）0：逻辑0或“假”，低电平；
  - 2）1：逻辑1或“真”，高电平；
  - 3）x 或者X：未知，不确定；
  - 4）z 或者Z：高阻抗。
- 在门的输入或一个表达式中的为“z”的值通常解释成“x”。
- x值和z值都是不分大小写的，也就是说，值0x1z与值0X1Z相同。
  - 0x1z=0X1Z
- Verilog HDL中的常量是由以上这四类基本值组成的。

1、常量

- Verilog HDL中有三类常量：
  - （1）整型
  - （2）实数型
  - （3）字符串型

1、常量

❖ （1）整型常量

<位数> ‘<进制> <数字序列>

- 位数：指明数字的二进制位数，只能用十进制数表示；
- 进制：表明数字序列的进制，有4种进制：
  - b或B：二进制
  - o或O：八进制
  - d或D：十进制
  - h或H：十六进制
- 数字序列：用连续的0~9、a~f或者A~F来表示，不区分大小写，不同进制只能使用规定的部分数字。

1、常量

❖ （1）整型常量

- 举例：

```
4'b1x01        //4位（二进制）的二进制数1x01
5'O37          //5位（二进制）的八进制数37
9'D256         //9位（二进制）的十进制数256
12'habc        //12位（二进制）的十六进制数abc
```

1、常量

❖ （1）整型常量

说明：

- **默认位宽：与仿真器和使用的计算机有关（最小为32位）；**
- **默认进制：默认为十进制数；**
- **负数：在<位数>前加负号“-”；负数数值用二进制补码表示**
- **符号说明“s”：用于表示参加算术运算是带符号数；**
- **下划线符号“_”：可出现在数字序列的任何位置，用来提高易读性，在编译阶段将被忽略；**
-

1、常量

■ 举例：

```
 'Ha5          //位数省略，默认为32位的十六进制数
6789           //位数和进制都省略，这是一个32位的十进制数6789
-8'd9          //这是一个8位的以二进制补码表示的十进制数-9
-8'so11        //这是一个用于算术运算的带符号的八进制数-11
12'B1011_1010_1111
               //用下划线提高可读性，等价于12'B101110101111
```

1. 常量

- “？”：用来代替高阻“z”，以增强casez和casex语句的可读性；
- x（或z）：在十六进制值中代表4位x（或z），在八进制中代表3位x（或z），在二进制中代表1位x（或z）；
- 高位自动扩展：<位数>比<数字序列>的长度长，Verilog约定：
  - 最高位是0、x或z：则用0、x或z自动扩展，填补最高位；
  - 最高位是1：则用0自动扩展，填补最高位；
- 截断： <位数>比<数字序列>的长度更短，则最左边的位被截断。

1. 常量

■ 举例：

```
4'B10??          //等价于4'B10zz
12'H56x          //12位的十六进制数，即二进制的01010110xxxx
9'o5Z6           //9位的八进制数，即二进制的101zzz110
32'bz            //32位的高阻值
10'b10           //左边添0占位，等价于0000000010
10'bx0x1         //左边添x占位，等价于xxxxxxx0x1
10'bz01101       //左边添z占位，等价于zzzzz01101
3'b1001_0011     //左边高位截断，等价于3'b011
5'H0FFF          //左边高位截断，等价于5'H1F；
```

1. 常量

■ 举例：非法的整型常量定义

下面是一些非法常量定义：

```
8'd-9          //非法表示，符号不能在<进制>和<数字序列>之间
3' b001        //非法：撇“'”和<进制>b之间不允许出现空格
56f            //非法：十六进制数必须要有'<进制>，'h或'H
B10111         //非法：<进制>前必须要有撇“'”
(2+3)'b10      //非法：位长不能够为表达式
```

1、常量

❖ （2）实数常量

- 实数常量定义：
  - 十进制计数法
  - 科学计数法
- 说明：
  - 小数点两边都要有数字
  - 下划线符号“_”被忽略
  - 用e或E（不区分大小写）隔开系数和指数；
  - 实数通过四舍五入被转换为最相近的整数

1、常量

❖（2）实数常量

```
2.0              //小数点两边都必须要有数字，不能为2或者2.
1125.678
0.1
23_5.1e2         //e后面的数字为指数；忽略下划线；真值为
                 //23510.0
3.6E2            // E与e相同；其值为360.0
5E-4             //其值为0.0005
```

1、常量

❖ （3）字符串常量

- 字符串是用双引号括起来的一个字符序列。
- 对于字符串的限制是：必须在一行中书写完，不能分成多行书写，也即不能包含回车符。
- 用8位的ASCII码表示字符串中的字符。
- 字符串常量定义：

```
"Hi,Verilog HDL!"          //是字符串
 “CLICK->HERE”             //是字符串
reg [8*15:1] String_1;     //  变量
 initial
     String_1 = “Hi,Verilog HDL!”   //变量赋字符串初值
```

1、常量

❖ （3）字符串常量

- **特殊字符**：在显示字符串时具有特定的意义，例如换行符、制表符和显示参数的值等。
- 如果要在字符串中显示这些特殊字符，则必须在前面添加前缀**转义字符**，主要为反斜线“\”和“%”

转义字符	显示的特殊字符
\n	换行符
\t	tab（制表空格）
\\	\
\"	"
%%	%
\ooo	八进制数ooo对应的字符

2、变量

- 端口和变量声明时，需要指明其数据类型，以便分配存储空间或明确信号属性。
- Verilog HDL有19种数据类型，依据其信号的物理特性，分为两大类：
  - （1）线网类型（net type）：11种
  - （2）寄存器类型（register type）：5种

2、变量

❖（1）线网类型

- 表示Verilog结构化元件间的物理连线，它由其连接器件的输出来连续驱动，例如连续赋值或门的输出。
- 如果没有驱动元件连接到线网，线网的缺省值为z。

❖线网数据类型包含下述11种线网子类型：

- wire：普通线网
- tri：三态线网（多驱动源线网）
- wand：线与线网
- wor：线或线网
- triand：线与三态线网
- trior：线或三态线网

- trireg：三态寄存器
- tri0：三态线网0（无驱动源时为0）
- tri1：三态线网1（无驱动源时为1）
- supply0：电源地
- supply1：电源

2、变量

❖（1）线网类型

❖线网类型说明语法为：

net_type [msb:lsb] net1, net2, . . . , netN;

- net_type：是上述线网类型的一种；
- msb和lsb：用于定义线网位宽的常量表达式；
- 位宽定义是可选的；如果没有定义位宽，缺省的线网位宽为1位。

2、变量

❖（1）线网类型

❖声明举例：

```
wire Ready, Ack;          //定义了2个1位的连线
wire [31:0] data;         //data是32位的线网变量
wand [2:0] Addr;          //Addr是3位的线与变量
```

- 如下的定义是非法的：

```
wire [4:0] x, [3:1]y;     //不同的位宽要分开定义
wire z[7:0];              //定义格式不对
```

2、变量

❖（1）线网类型

❖在所有网线型数据类型中，最常用的是wire型：

❖wire型端口信号/变量

- 可作为任何语句中的输入
- 可作为实例化原件（模块）的输出
- 只能用assign语句驱动，且不可放在always语句块中
- 默认情况下，模块的输入/输出端口自动定义位wire型
- 可以被综合器综合，综合为连接线

2、变量

（2）寄存器类型

- 表示一个抽象的数据存储单元，它保持原有的数值，直到被改写。
- 寄存器类型变量只能在always和initial语句中被赋值，并且它的值从一个赋值到另一个赋值之间被保存下来。
- 寄存器类型的变量具有x 的缺省值。

有5种不同的寄存器类型：

- reg：寄存器类型
- integer：整型
- time：时间寄存器
- real：实数类型
- realtime：实数时间类型（与real类型完全相同）

2、变量

❖（2）寄存器类型

- 1）reg型
  - 最常用的寄存器类型，对应触发器、锁存器等具有状态保持功能的电路元件 。
  - 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。
  - reg型变量的定义格式如下：

    reg[msb：lsb] 变量l, 变量2, …, 变量n;

    - reg是类型关键字；
    - [msb：lsb]用于定义reg型变量的位宽，msb是最高位序号，lsb是最低位序号；如果不指定位宽，则自动默认为l。

2、变量

❖（2）寄存器类型

- 1）reg型
  - 定义举例：

```
reg [3:0] Sum;          //Sum为4 位寄存器
reg Cnt;                //1位寄存器
reg [1:32] MAR, MDR;    //定义了2个32位寄存器
```

2、变量

- 2）存储器类型
  - 存储器类型通常用来为RAM和ROM建模，它实际上就是寄存器的一维数组。
  - 数组的每个元素称为一个元素或者字（word），由索引来指定；字的位宽（字长）可以为1位或者多位。
  - 存储器属于寄存器数组类型
  - 数组的维数有限制，最大是3维数组

注意线网数据类型没有相应的存储器类型。

2、变量

- 定义格式

reg [msb: lsb] memory1[lower1:upper1], memory2[lower2: upper2],. . . ;

  - reg： 类型关键字
  - [msb: lsb]定义存储器字的位宽
  - [lower1: upper1]定义存储器的字数（即数组的元素个数）。

2、变量

- 举例

```
reg [0:3] MyMem [0:63];   //MyMem为64×4bits的存储器
reg membit [0:1023];      // membit为1024×1bit的存储器
reg [0:1023] Data;        // Data是一个又1024位的寄存器
```

2、变量

❖2）存储器类型

- 定义举例：

```
parameter ADDR_SIZE = 1024, WORD_SIZE = 8;
reg [0: WORD_SIZE-1] Ram1 [ ADDR_SIZE-1 : 0], DataReg;
```

- Ram1是1K×8位寄存器数组（存储器），
- DataReg是一个8位寄存器；
- Ram1[345]：访问存储器Ram1的地址为345的那个单元的字节数据。

2、变量

- 3）integer寄存器类型
  - 整数寄存器包含整数值，整数寄存器可以作为普通寄存器使用，典型应用为高层次行为建模。
  - 整数型变量定义：

integer integer1, integer2,. . . intergerN;

  - 整数声明中容许无位界限，默认是宿主机的位宽，最少容纳32位。
  - 整数变量是有符号数，而reg型变量是无符号数。
  - 整数不能作为位向量访问。

2、变量

❖（2）寄存器类型

■ 3）integer寄存器类型

■ 定义举例：

```
integer A, B, C;          //三个整数型寄存器。
integer List1 [3:6];      //一组四个寄存器。
```

2、变量

❖4）time类型

- time类型的寄存器用于存储和处理时间
- Time类型变量定义格式:

time time_id1, time_id2, … , time_idn;

  - time: 时间类型关键字
  - 每个标识符存储一个至少64位的时间值，其宽度与具体实现有关
  - 时间类型的寄存器只存储无符号数

2、变量

❖4）time类型

- 定义举例：
  - time Events [0:31];　//32个时间值数组
  - time CurrTime；　　//CurrTime 存储一个时间值

2、变量

❖5）real和realtime类型

- real和realtime类型完全一样
- 它们都不能被综合
- real类型变量定义格式

real real_reg1, real_reg2, … , real_regN;

- real： 实数类型关键字
- real说明的实数变量的缺省值为0
- 默认32位，不允许对real变量声明值域、位界限或字节界限。

2、变量

❖5）real和realtime类型

- 定义举例

```
real Swing, Top;
        //将值x和z赋予real类型寄存器时，这些值作0处理
real RamCnt;
…
ramCnt = ‘b01x1Z;
```

3、标量与向量

- 向量：位宽大于1
- 标量：位宽为1
- 线网和寄存器类型的数据均可声明为向量，如果在声明中未指定位宽，则默认为标量。

```
wire a;                    //标量线网变量，默认位宽=1
wire [15:0]   ab,db;       //定义了2个16位的向量线网
reg [1:16]    AR,DR;       //定义了2个16位的寄存器
reg clk;                   //标量寄存器，默认
```

3、标量与向量

- 访问wire线网向量或者reg寄存器向量时，可以对向量整体访问（全选）或者部分访问（位选或域选）。
- 整体访问：全选
  - 全选：用“变量名”直接访问，相当于访问变量的所有位。
- 部分访问：位选或域选
  - 位选：用“变量名[序号]”方式访问，相当于访问变量的某一位（由序号指定）。
  - 域选：用“变量名[high：low]”方式访问，相当于访问变量的部分位（由序号high和low指定）

3、标量与向量

```
wire [7:0] in,out;              //in和out为8位wire线网向量
wire [4:1] temp1;               // temp1为4位wire线网向量
reg [3:0] temp2;                // temp2为4位reg寄存器向量
assign out=in;                  //8位全选，整体访问
assign out[7:4]=in[3:0];    //域选，将in的低4位赋值给out的高4位
assign out[6:3]=temp1;    //out域选，temp1全选，将temp1赋值给out
                                的3~6位
assign in[7:4]=temp2;   //reg变量temp2对wire变量in的高4位持续驱动
```

4、参数

- 参数是一个常量，常用于定义时延和变量的宽度。
- 使用parameter说明的参数只被赋值一次。
- 参数说明形式如下：

```
parameter param1=const_expr1, param2 = const_expr2 , . . . ,
    paramN = const_exprN;
```

- 定义举例：

```
parameter LINELENGTH = 132, ALL_X_S = 16'bx;    //整型常量参数
parameter BIT = 1, BYTE = 8, PI = 3.14;      //PI是实数型常量参数
parameter STROBE_DELAY = ( BYTE + BIT) / 2;
                                  //由表达式计算结果赋值参数
parameter TQ_FILE = " /home/bhasker/TEST/add.tq";
                                  //字符串型常量参数
```

3.5 表达式与操作符

- **表达式由操作数和操作符构成**，其作用是根据操作符的意义对操作数计算出一个结果。
- **操作数**：常数、参数、线网、寄存器、线网和寄存器的位选与域选、时间、存储器、函数调用。
- **常量表达式**：在编译时就计算出常数值的表达式，常量表达式由**常量和参数**构成。
- **标量表达式**：计算结果为1位的表达式。

3.5 表达式与操作符

- 注意区分无符号数和有符号数：
- 下列情况被当做无符号数：
  - 线网型
  - 一般寄存器型（reg）
  - 基数格式表示形式的整数，譬如5’d12、-5’d12
- 下列情况被当做有符号数：
  - 整数寄存器（integer）
  - 十进制形式的整数，譬如12、-12

3.5 表达式与操作符

举例：

```
-12 / 4          //有符号数，结果是-3
-'d12 / 4        /*无符号数，结果是-12的32位二进制补码除以4，即
                 （2^32-12）÷4=1073741821  */
```

3.5 表达式与操作符

```
reg[3:0] A,B;
interger C,D,E;
   A=4'b0110; B=4'b0100;
   C=7;  D=4;  E=2;
```

- 操作符：9类
  - （1）算术操作符

操作符符号	执行操作	操作数个数	举例
+	算术加	1或2	A+B //结果为4'b 1010
-	算术减	1或2	A-B //结果为4'b 0010
*	乘法	2	A*B //结果为5'b 11000
/	除法	2	C/D //结果为1，整除
%	模除	2	C%D //结果为3，取余数
**	求幂	2	C**E //结果为7^2=49

3.5 表达式与操作符

❖ （1）算术操作符

- 功能：执行算术运算。
- 整数除法截断任何小数部分。
- 模除操作符求出与第一个操作符符号相同的余数
- 如果操作数的任意一位是X或Z，则结果为X
- “+”和“-”操作符：可以当做单目操作符使用，用于表示操作数正负号，而且比双目操作符具有更高的优先级
- 算术表达式结果的长度由最长的操作数决定
- 在赋值语句下，算术操作结果的长度由操作符左端目标操作数长度决定。

3.5 表达式与操作符

a = 0 ; b = 1;
C =3 ; D =0 ;
E=4'b0x10;

- 操作符：9类
  - （2）逻辑操作符

操作符符号	执行操作	操作数个数	举例
&&	逻辑与	2	a && b //结果为0 C&&D //结果为1&&0=0
||	逻辑或	2	a || b //结果为1 C || D //结果为1 || 0=1
!	逻辑非	1	!a 结果为1，!b结果为0 !C结果为0，!D结果为1

3.5 表达式与操作符

■（2）逻辑操作符

- 功能：逻辑操作符在逻辑值0或1上进行逻辑运算，运算结果也为1个1位的值：0表示假，1表示真，x表示不确定。
- 对于向量操作，非0向量作为1处理。
- 如果任意一个操作数包含x，结果也为x。
- 逻辑操作符一般对变量和表达式进行操作。

```
a = 0 ;  b = 1;
C =3 ;  D =0 ;
E=4'b0x10;
```

```
（C==3）&&（D==0）
  // 即当C等于3且D等于0时，表达式值为1；否则，值为0。
```

3.5 表达式与操作符

- 操作符：9类
  - （3）关系操作符

A= 6 ； B = 8;
C=4’b0x10;

操作符符号	执行操作	操作数个数	举例
>	大于	2	A > B //结果为0
<	小于	2	A < B //结果为1
>=	大于等于（不小于）	2	A >= B //结果为0
<=	小于等于（不大于）	2	A <= C //结果为x

3.5 表达式与操作符

- （3）关系操作符
  - 功能：比较两个操作数的大小关系，运算结果是1位的逻辑状态。
  - 如果操作数中有一位为X或Z，那么结果为X。
  - 如果操作数长度不同，长度较短的操作数在左边添0补齐。

3.5 表达式与操作符

- 操作符：9类
  - （4）等价操作符

```
A= 6 ;          B = 8;
C=4'b1001;      D=4'b1011;
E=4'b1xxz;      F=4'b1xxx;
G=4'b1xxz;
```

符号	执行操作	操作数个数	说明	举例

==	等于	2	若操作数中有x或z，则结果为x	A == B //结果为0
				E == G //结果为x

!=	不等于	2	若操作数中有x或z，则结果为x	A!= B //结果为1 D!= E //结果为x

===	全等	2	比较包括x和z	E === G //结果为1
				E === F //结果为0

!==	不全等	2	比较包括x和z	D !== E //结果为1 G !== E //结果为0

3.5 表达式与操作符

- （4）等价操作符
  - 功能：比较两个操作数是否等价，运算结果是1位的逻辑状态。
  - 全等比较===和!==：数值中x和z严格按位比较。即：不进行解释，并且结果一定可知（确定的0或1）。
  - 逻辑比较==和!=：数值中x和z具有通常的意义，且结果可以不为x。也就是说，在逻辑比较中，如果两个操作数之一包含x或z，结果为未知的值（x）。
  - 如果操作数的长度不相等，长度较小的操作数在左侧添0补位。

3.5 表达式与操作符

- 操作符：9类
  - （5）位运算操作符

A=4’b1011;　　B=4’b1101;
C=4’b0000;　　D=4’b1xxx;

符号	执行操作	操作数个数	举例
&	按位与	2	A & B //结果为4’b1001 A & D //结果为4’ b10xx
|	按位或	2	A | B //结果为4’b1111 A | D //结果为4’b1x11
^	按位异或	2	A^B //结果为4’b0110 A^D //结果为4’b0xxx
~	按位取反	1	~A //结果为4’b0100 ~D //结果为4’ b0xxx
~^或^~	按位同或	2	A~^B //结果为4’b1001 B~^C //结果为4’b0010

3.5 表达式与操作符

- （5）位运算操作符
  - 功能：两个操作数按位进行逻辑运算操作，并产生向量结果。
  - 位运算操作符：结果是一个向量，每位都是按位运算得出；
  - 逻辑操作符：结果是一位逻辑值0、1或x。
  - 关于x的位运算规则：
    - x θ x=x
    - 0&x=0，1&x=x
    - 0|x=x，1|x=1
    - 0^x=x，1^x=x，0^~x=x，1^~x=x
  - 如果操作数的长度不相等，长度较小的操作数在左侧添0补位。

3.5 表达式与操作符

- 操作符：9类
  - （6）归约操作符

```
A=4'b1011;
B=4'b1101;
C=4'b0000;
D=4'b1x0x;
```

符号	执行操作	操作数个数	举例
&	归约与	1	&A //结果为0 &D //结果为0
~&	归约与非	1	~&B //结果为1 ~&D //结果为x
|	归约或	1	| A //结果为1 | D //结果为1
~|	归约或非	1	~| C //结果为1 ~| D //结果为0
^	归约异或	1	^C //结果为0 ^D //结果为x
~^或^~	归约同或	1	~^B //结果为0 ~^D //结果为x

3.5 表达式与操作符

- （6）归约操作符
  - 功能：缩减操作符，对一个操作数的各位逐位进行逻辑运算，产生一个一位的逻辑值。
  - 运算过程：对操作数的所有位，逐位地从左到右进行逻辑运算。
  - 归约操作符：单目操作符，结果是一位逻辑值（0、1或者x）；
  - 位运算操作符：双目操作符，结果是一个向量。

3.5 表达式与操作符

- （6）归约操作符
  - 归约运算规则：
    - 归约与：操作数向量中有1个0，结果一定为0；
    - 归约或：操作数向量中有1个1，结果一定为1；
    - 归约异或：不含x或z的操作数向量中有奇数个1，结果一定为1；
    - 归约同或：不含x或z的操作数向量中有偶数个1，结果一定为1；
    - 归约异或/同或：操作数向量中有1个x，结果一定为x。

3.5 表达式与操作符

❖操作符：9类

- （7）移位操作符

```
A=4'b1011;   B=4'b1101;
integer C=-10;
//二进制为1111_1111_1111_1111
_1111_1111_1111_0110
```

符号	执行操作	操作数个数	举例
>>	逻辑右移	2	A>>1 //结果为4'b0101 B>>2 //结果为4'b0011
<<	逻辑左移	2	A<<1 //结果为4'b0110 B<<2 //结果为4'b0100
>>>	算术右移	2	C>>>2 /*结果为32'b1111_ 1111_1111_ 1111_1111_1111_1111_1101，即-3*/
<<<	算术左移	2	C<<<3 /*结果为32'b1111_ 1111_1111_ 1111_1111_1111_1011_0000，即-80*/

3.5 表达式与操作符

- （7）移位操作符
  - 功能：将操作符左边的操作数左移或者右移若干位，位数由操作符右边的操作数指定。。
  - 逻辑移位对无符号操作，算术移位对有符号数操作；
  - 逻辑右移和逻辑左移操作时，空闲位填0补位；
  - 算术左移时低位添0补位，算术右移时高位填符号位补位（正数补0，负数补1）；
  - 右移1位相当于除以2，左移1位在不溢出的情况下，相当于乘以2；

3.5 表达式与操作符

❖操作符：9类

- （8）条件操作符
  - 格式如下：

条件表达式？真表达式：假表达式

  - 如果条件表达式为真（即值为1），则运算返回真表达式；
  - 如果条件表达式为假（即值为0），则运算返回假表达式。
  - 如果条件表达式为x或z，将真表达式和假表达式按位操作后返回
    - 运算逻辑如下： 0与0得0，1与1得1，其余情况为x。
  - 举例：变量c要取a和b中值大的那个数，则可以：

```
c = (a>b) ? a : b          //即若a>b，则c=a，否则c=b
```

3.5 表达式与操作符

❖操作符：9类

- （9）拼接和复制操作符
  - 拼接：将小表达式合并形成大表达式，格式如下：

    {表达式1, 表达式2, ……, 表达式n}

举例：

```
wire [7:0] Dbus;
wire [11:0] Abus;
assign Dbus = {Abus [3:0], Abus [7:4]}; //正确
{Abus, 5}  //错误：常数5的长度不确定
```

3.5 表达式与操作符

❖操作符：9类

- （9）拼接和复制操作符
  - 复制操作符：用于指定拼接时的重复次数，格式如下：

    {重复次数 {表达式1, 表达式2, ……, 表达式n}}
  - 举例：

```
Abus = {3{4'b1011}};              //位向量12'b1011_1011_1011
Abus = {{4{Dbus[7]}}, Dbus};      //可用于符号扩展
{3{1‘b1}}                         //结果为111
{3{Ack}}                          //结果与{Ack, Ack, Ack}相同。
```

3.5 表达式与操作符

❖操作符优先级：

- 除条件操作符从右向左关联外，其余所有操作符自左向右关联。
- 从最高优先级（顶行）到最低优先级（底行）排列，同一行中的操作符优先级相同。

操作	符号	优先级
单目操作	+ - ! ~	最高
乘、除、模除	* / %	
加、减	+ -	
移位	>> <<	
关系	< <= > >=	
等价	== != === !==	
缩减 逻辑 位运算	& ~&	
	^ ~^	
	| ~|	
	&&	
	||	
条件	?:	最低

3.5 表达式与操作符

❖ 举例:

- A+B-C　　//等价于(A+B)-C
- A？B：C？D：F　　//等价于A?B:(C?D:F)
- (A？B：C)？D：F　　//圆括号可以改变优先级

3.7 Verilog HDL建模方式

1、建模方式概述

2、结构建模方式

3、数据流建模方式

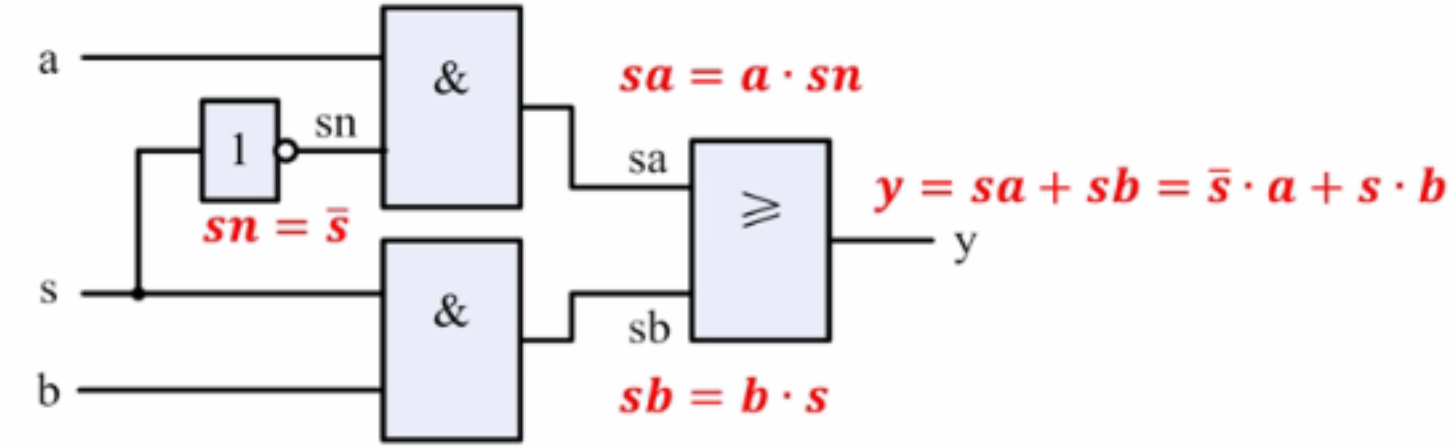
4、行为建模方式

1、建模方式概述

- 在Verilog模块中，可用下述方式描述一个设计：
  - （1）结构描述方式：使用门级和开关级内置器件来设计与描述逻辑电路；
  - （2）数据流描述方式：通过说明数据的流程对模块的逻辑功能进行描述；
  - （3）行为描述方式：只注重电路实现的算法，对电路行为进行描述；
  - （4）混合描述方式：上述描述方式的混合

1、建模方式概述

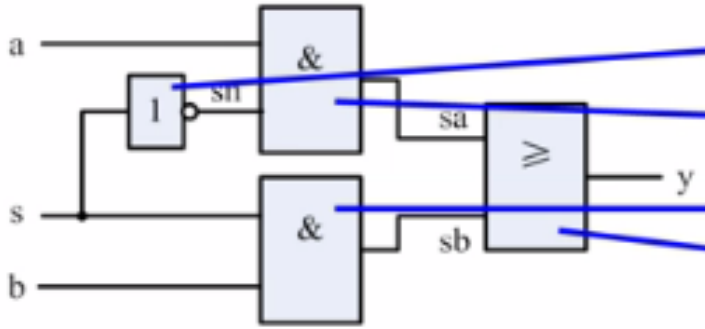
❖ 【例3.3】使用三种基本建模方式，描述模块mux2to1，实现图3.6所示的1位二选一电路。



1位二选一电路图

1、建模方式概述

- 特点：根据电路结构建模
- 建模方法：使用基本的内置逻辑门，将电路图翻译成Verilog的模块语句。
- 适用场合：有详细电路图的设计。




```
module mux2to1_1(y,a,b,s);
    output      y;
    input       a,b,s;
    wire        a,b,s,y;
    wire        sn,sa,sb

    not         u1(sn,s);
    and         u2(sa,a,sn);
    and         u3(sb,b,s);
    or          u4(y,sa,sb);
endmodule
```

1、建模方式概述

- 特点：根据逻辑表达式建模，只关心逻辑表达式，而不关心具体的门电路。
- 建模方法：根据最简逻辑表达式，采用连续赋值语句（assign）对输出变量进行赋值。
- 适用场合：有明确逻辑表达式的设计。

$$y = \bar{s} \cdot a + s \cdot b$$

```
module      mux2to1_2(y,a,b,s);
    output  y;
    input   a,b,s;
    wire    a,b,s,y;
    assign  y = ((~s) & a) | (s & b);
    //等价于 assign y =s ? b : a;
endmodule
```

1、建模方式概述

- 特点：根据电路行为建模，看不到电路的内部结构，只看到电路表现的行为。
- 建模方法：基于电路行为的因果关系来描述模块的逻辑功能。
- 适用场合：有明确的因果关系与行为逻辑的设计 。

```
module mux2to1_3(y,a,b,s);
        output y;
        input  a,b,s;
        wire   a,b,s;
        reg    y;
        always @(s or a or b)
               if (!s) y = a;
               else y = b;
endmodule
```

1、建模方式概述

总结：

- 设计者可以根据需要，在每个模块内部，对逻辑功能进行不同方式的描述，这与建模的抽象层次有关。
- 模块对外显示的功能都是一样的，仅与外部环境有关，与内部的抽象层次无关；而模块的内部结构对外部环境来说是透明的。

1、建模方式概述

- Verilog有4个抽象层次：从低到高
  - 开关级（几乎不用）
  - 门级
  - 数据流级
  - 行为级（或称算法级）
- 寄存器传输级RTL：能够被逻辑综合工具综合的行为级建模和数据流级建模的结合。
- 在经过综合工具综合之后，综合的结果一般都是门级结构的描述

1、建模方式概述

- 一般来说，抽象的层次越高，设计的灵活性越高，和工艺的无关性越高；随着抽象层次的降低，灵活性逐渐变差，工艺的相关性变大，微小的调整可能导致多处的修改。

2、结构建模方式

- 结构建模方式：通过调用逻辑元件，描述信号之间的连接，建立逻辑电路的模型。
- 可使用的逻辑元件：
  - 1）内置开关（晶体管）；
  - 2）内置逻辑门；
  - 3）用户定义的门级原语；
  - 4）模块实例；
- 逻辑元件之间，通过使用线网信号来相互连接

2、结构建模方式

❖ 内置门级元件：

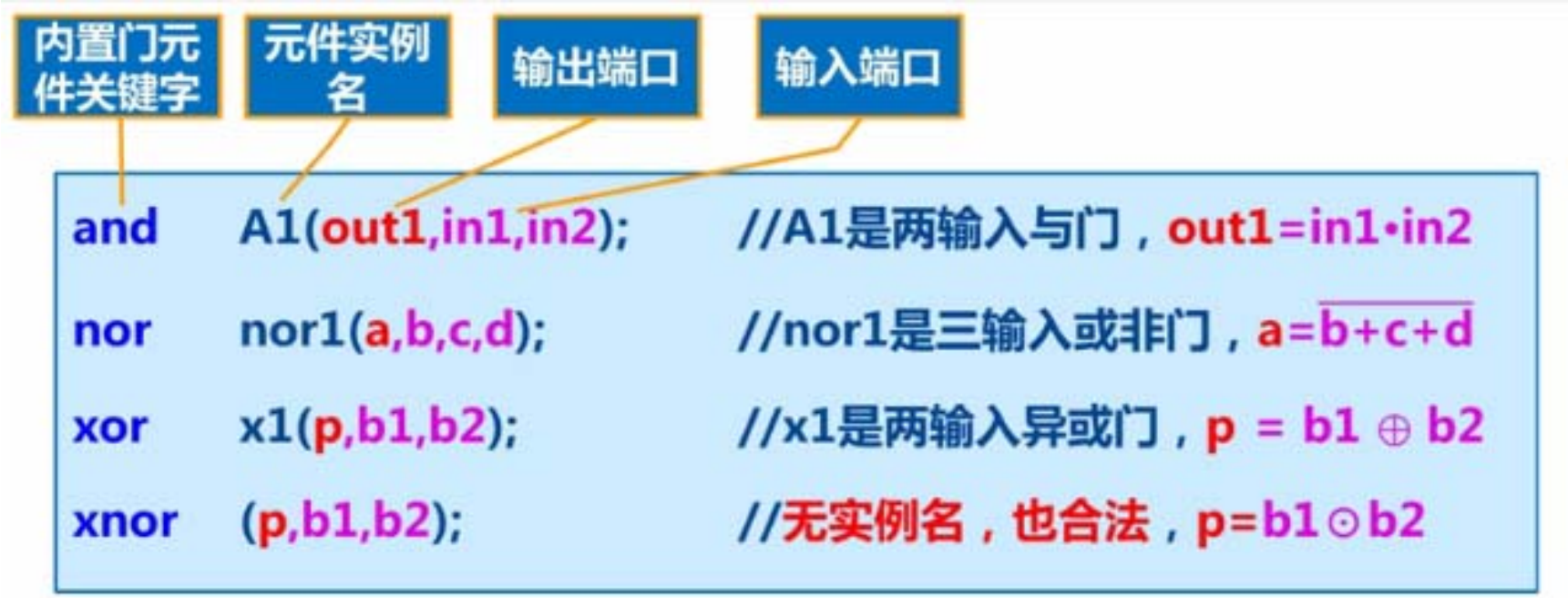
类型	关键字	元件功能	元件调用
多输入门	and	与门	<元件名> <实例名>(<输出>,<输入1>,…,<输入n>);
	nand	与非门	
	or	或门	
	nor	或非门	
	xor	异或门	
	xnor	异或非（同或）门	
多输出门	buf	缓冲器	<元件名> <实例名> (<输出1>,……,<输出n>, <输入>);
	not	非门	
三态门	bufif0	低使能三态门	<元件名> <实例名>(<数据输出>, <数据输入>, <控制输入);
	bufif1	高使能三态门	
	notif0	低使能三态非门	
	notif1	高使能三态非门	

2、结构建模方式

❖内置逻辑门的元件调用（实例引用）方式

- **端口列表排列**：输出在前，输入在后，最后是控制输入
- **多输入门**：可以有多个输入，但输出只有一个
- **多输出门**：可以有多个输出，但输入只有一个
- **三态门**：一个数据输出，一个数据输入，一个控制输入
- Verilog中的内置门级元件是一种**动态模型**，可以根据用户**调用时的端口列表**，**动态生成**相应的门电路。

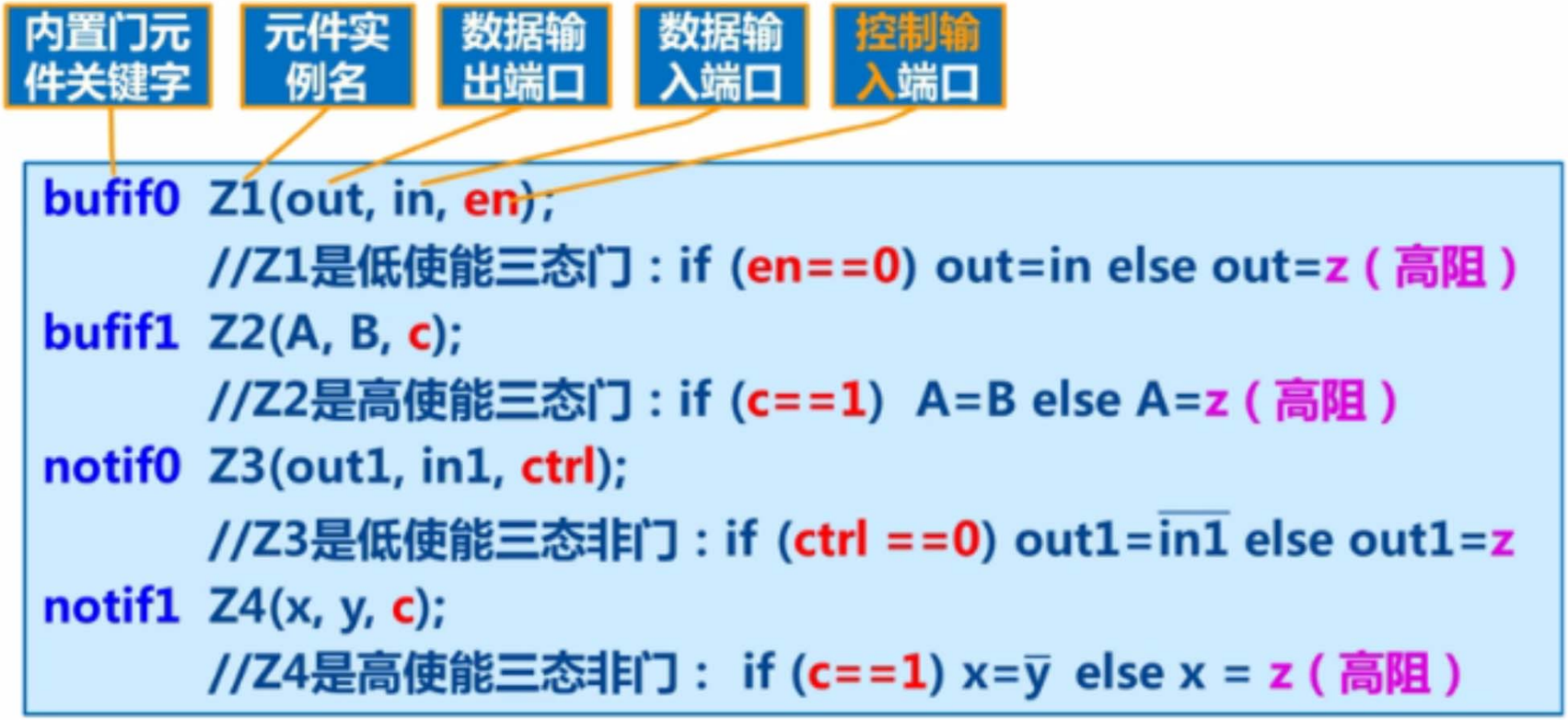
2、结构建模方式


内置门元件关键字
元件实例名
输出端口
输入端口
and A1(out1,in1,in2); //A1是两输入与门，out1=in1•in2
nor nor1(a,b,c,d); //nor1是三输入或非门，a=b+c+d
xor x1(p,b1,b2); //x1是两输入异或门，p = b1 ⊕ b2
xnor (p,b1,b2); //无实例名，也合法，p=b1⊙b2

2、结构建模方式


内置门元件关键字
元件实例名
输出端口
输入端口
not not1(out1,out2,in);
//not1是两输出非门， out1=out2=in
buf buf1(Aout1,Aout2,Aout3,Bin);
//buf1是三输出缓冲门， Aout1=Aout2=Aout3=Bin

2、结构建模方式


内置门元件关键字
元件实例名
数据输出端口
数据输入端口
控制输入端口
bufif0 Z1(out, in, en);
//Z1是低使能三态门：if (en==0) out=in else out=z（高阻）
bufif1 Z2(A, B, c);
//Z2是高使能三态门：if (c==1) A=B else A=z（高阻）
notif0 Z3(out1, in1, ctrl);
//Z3是低使能三态非门：if (ctrl ==0) out1=in1 else out1=z
notif1 Z4(x, y, c);
//Z4是高使能三态非门： if (c==1) x=y else x = z（高阻）

2、结构建模方式

- 当同一种内置逻辑门元件，被多次引用时，可以连续调用，并用“，”隔开各个实例名和端口列表。

■ 举例：用一个“and”定义三个与门元件实例

```
and     AU1 (T3, A, B),          //用“,”隔开
        AU2 (T2, B, Cin),        //省略and
        AU3 (T1, A, Cin);        //省略and
```

2、结构建模方式

❖ 门级描述与结构建模的**模型**：

module 模块名(端口列表)

端口定义
input　　输入端口
output　输出端口

数据类型声明
wire

门级建模与描述
and　　U1(输出,输入1,…输入n);
not　　U2(输出1,…输出n,输入);
bufif0　U3(输出,输入,使能控制);

endmodule

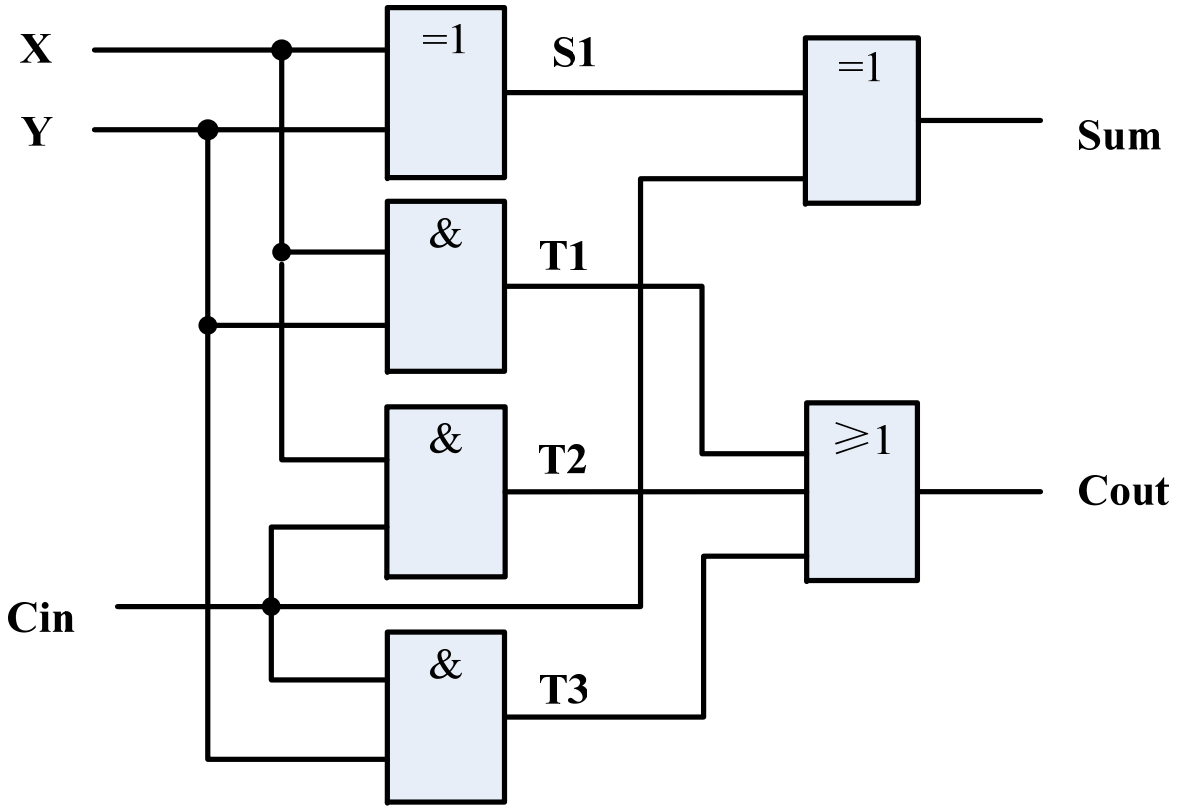
2、结构建模方式

❖门级描述与结构建模

- 门级建模描述的是电路结构，它用于将组合逻辑电路转换为Verilog HDL描述。
- 门级建模的程序不像逻辑电路那样直观。
- 需要依赖传统的逻辑电路分析方法加以分析、画出电路图，然后使用Verilog HDL的门级元件描述电路结构
- 门级描述和结构建模并不是Verilog建模的主流方法。
- 门级描述可以用以实现自底向上设计的底层模块库，供自顶向下的设计来调用。

2、结构建模方式

❖ 【例4】使用结构建模方式，描述1位全加器模块Full_Adder，电路如图3.7所示。



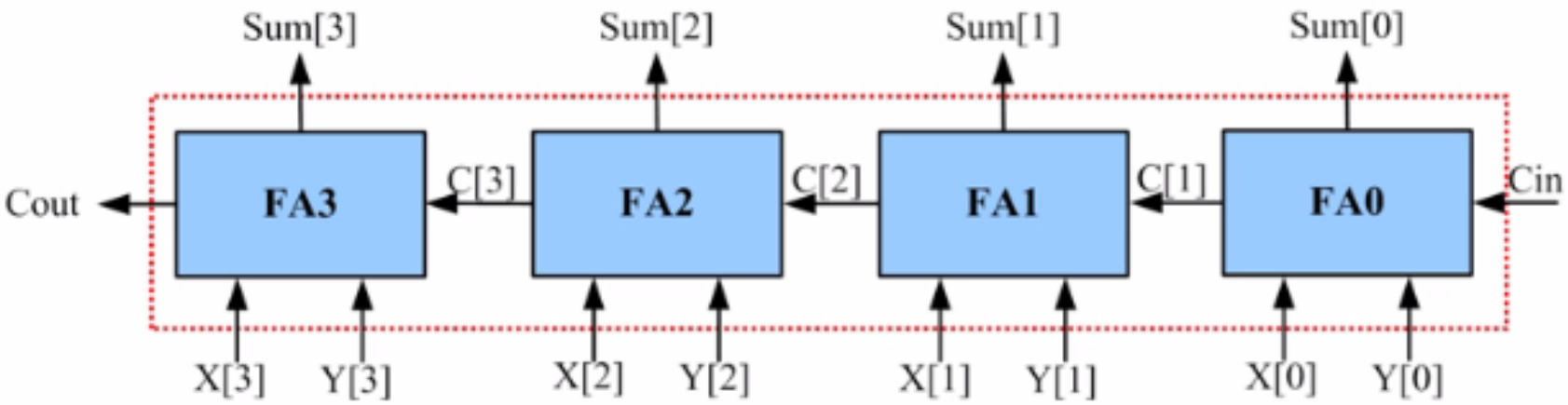
2、结构建模方式

❖ 全加器模块Full_Adder：

```
module Full_Adder (Sum, Cout, X, Y, Cin);
    output      Sum, Cout;
    input       X, Y, Cin;
    wire        S1, T1, T2, T3;
    xor  XU1 (S1, X, Y),                 //S1 = X ⊕ Y
         XU2 (Sum, S1, Cin);             //Sum = S1 ⊕ Cin
    and  AU1 (T1, X, Y),                 //T1 = X·Y
         AU2 (T2, X, Cin),               //T2 = X·Cin
         AU3 (T3, Y, Cin);               //T3 = Y·Cin
    or   OU1 (Cout, T1, T2, T3);         //Cout = T1+T2+T3
endmodule
```

2、结构建模方式

- 基于例4的1位全加器，对4位加法器进行结构建模，电路如图所示



2、结构建模方式

```
module FourBitFA (X, Y, Cin, Sum, Cout );
      parameter SIZE = 4;
      input [SIZE-1:0] X, Y;
      output [SIZE-1:0] Sum;
      input Cin;
      output Cout;
      wire [SIZE-1: 1] C;
       Full_Adder   FA0( Sum[0], C[1], X[0], Y[0], Cin ),
                    FA1( Sum[1], C[2], X[1], Y[1], C[1] ),
                    FA2(.A(X[2]),.B(Y[2]),.Cin(C[2]),.Sum(Sum[2]),.Cout(C[3])),
                    FA3(.Sum(Sum[3]),.Cout(Cout),.A(X[3]),.B(Y[3]),.Cin(C[3]));
endmodule
```

3、数据流建模方式

- **数据流建模**：是通过信号变量之间的**逻辑关系**，采用**连续赋值语句（assign）**来描述逻辑电路的功能。
  - 实际上，数据流描述就是将传统意义上的**逻辑表达式**，用Verilog HDL的assign语句来表达。
- **逻辑综合** ：借助于**计算机辅助设计工具**，自动将电路的**数据流设计直接转换为门级结构** 。
- 数据流建模方式已经成为主流的设计方法 。

3、数据流建模方式

❖连续赋值语句assign：用来驱动wire型变量

- wire型变量不具备数据保持能力，只有被连续驱动后，才能取得确定的值。
- 若wire型变量没有得到任何驱动，它的取值为不确定的“x”

❖连续赋值语句格式如下：

assign #延时量 wire型变量名 = 赋值表达式;

- 只要右边赋值表达式中使用的操作数发生变化，就重新计算表达式的值，新结果在指定的时延后赋值给左边的wire型变量。
- 时延：定义了右边表达式操作数变化与赋值给左边表达式之间的延迟时间。如果没有定义时延值，缺省时延为0，即立即赋值。

3、数据流建模方式

❖连续赋值语句assign的特点：

- 赋值目标：必须为线网型变量，不能是寄存器型变量；
- 连续赋值语句是并发执行的，也就是说各语句的执行顺序与其在描述中出现的顺序无关。
- 体现了组合逻辑电路的特征：任何输入发生变化，输出立即随之而变。
- 多用来描述组合理辑电路。

3、数据流建模方式

❖连续赋值语句assign

- 对线网变量的连续赋值，一般的方法是：
  - 对线网变量先声明，再用assign连续赋值语句赋值
- Verilog还提供隐式连续赋值：
  - 声明线网变量的同时，对其进行赋值；
  - 效果等同于一般赋值方法。
- 举例

一般赋值方法：

```
wire    out;
assign  out = in1 & in2;
```

=

隐式赋值方法：

```
wire    out = in1 & in2;
```

3、数据流建模方式

- 数据流描述与建模的模型
  - 逻辑功能描述部分：用 assign 语 句 对 wire型变量进行连续驱动；赋值表达式基于信号的逻辑函数表达式

module 模块名(端口列表)

端口定义
input　　输入端口
output　　输出端口

数据类型声明
wire

数据流级描述
assign　线网变量名1 = 表达式1;
……
assign　线网变量名n = 表达式n;

endmodule

3、数据流建模方式

- 使用数据流建模方式，描述1位全加器模块Full_Adder。

$$Sum = X \oplus Y \oplus Cin$$

$$Cout = X \bullet Y + (X \oplus Y) \bullet Cin$$

```
module Full_Adder (Sum, Cout, X, Y, Cin);
    output      Sum, Cout;
    input       X, Y, Cin;
    assign      Sum = X ^ Y ^ Cin;
    assign      Cout = ( X & Y ) | ((X ^ Y) & Cin);
endmodule
```

3、数据流建模方式

❖使用数据流建模方式，描述4位全加器模块

```
module FA_4bit (
        output  [3:0]     Sum,
        output            Cout,
        input   [3:0]     X,
        input   [3:0]     Y,
        input             Cin
        );
        //逻辑功能定义
        assign  { Cout ,Sum} = X + Y + Cin;
endmodule
```

4、行为建模方式

- 行为描述：关注逻辑电路的外部行为、输入输出变量的因果关系，而不关心电路的内部结构。
- 行为级建模是从算法的角度，即从电路外部行为的角度对其进行描述。
- 行为描述关心电路在何种输入下，产生何种输出，而不关心电路具体的实现。
- 行为级建模是最高的抽象层次，在这个层次上设计数字电路更类似于使用C语言编程。
- 行为级建模语法结构：always、initial、if-else、case 语句等

4、行为建模方式

可综合的行为描述模型

```
module 模块名(端口列表)
    端口定义
    input     输入端口
    output    输出端口
    数据类型声明
    reg
    parameter
    行为或算法级描述
    always @(敏感事件列表)
    begin
        阻塞/非阻塞过程赋值语句
        if-else、case、for语句
    end
endmodule
```

不可综合的行为描述模型

```
module 模块名(端口列表)
    端口定义
    input     输入端口
    output    输出端口
    数据类型声明
    reg
    parameter
    行为或算法级描述
    initial
    begin
        阻塞/非阻塞过程赋值语句
        if-else、case、for语句
    end
endmodule
```

4、行为建模方式

- Verilog中的行为描述有**两种结构化的过程语句**：
  - **always语句**：该语句总是循环执行，可以被逻辑综合工具接受，综合为门级描述；
  - **initial语句**：该语句只执行一次，不能被逻辑综合工具接受，用于初始化变量的值。
- 只有**寄存器类型数据**能够在这两种语句中被赋值，且寄存器类型数据在被赋新值前保持原有值不变；
- **所有的initial语句和always语句在0时刻并发执行。**

4、行为建模方式

- （1）initial语句
- （2）always语句
- （3）过程赋值语句
- （4）if-else语句
- （5）case语句
- （6）循环语句
  - forever循环
  - repeat循环
  - while循环
  - for循环

4、行为建模方式

- （1）initial语句：只执行一次，不能被逻辑综合工具接受，用于初始化变量的值，常用于仿真。

```
module          ini_demo;
    reg a,b,m,n,k
    Initial     k = 1’b0;
    initial
        begin
            #5    a = 1’b1;
            #20   b = 1’b0;
        end;
    initial
    begin
            #10   m = 1’b1;
            #25   n = 1’b0;
    end;
    Initial   #50$finish;
endmodule
```

各条语句的执行顺序：

时间	所执行的语句
0	k = 1’b0;
5	a = 1’b1;
10	m = 1’b1;
25	b = 1’b0;
35	n = 1’b0;
50	$finish;

4、行为建模方式

（2）always结构语句：

- 一个程序中可以有多个always语句，每个always语句内的所有行为语句构成了always语句块；
- 所有的always块在仿真0时刻开始顺序执行其中的行为语句；
- always语句不断地重复运行，即最后一条always块语句执行完后，又开始执行always块的第一条指令。
- 敏感事件列表：敏感事件列表中的事件一旦发生变化，就执行一次always语句块。

4、行为建模方式

（2）always结构语句：

- always语句格式如下：

```
always    @(敏感事件列表)              语句;
    //always语句块只有单条语句
always    @(敏感事件列表)    //有多条语句
begin
    语句;
    ……
end
```

4、行为建模方式

❖（2）always结构语句：

- 敏感事件列表：是激活always语句执行的条件
  - 单个事件或多个事件，多个事件之间用“or”关键字或者“,”连接；
  - 电平触发或者边沿触发，电平触发的always块常用于描述组合逻辑的行为，而边沿触发的always块常用于描述时序行为；
  - 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；
  - @*或者@(*)：表示对always语句块中的所有输入变量的变化是敏感的。

4、行为建模方式

❖（2）always结构语句：

- 用always语句描述组合逻辑和时序逻辑电路区别：

	组合逻辑电路	时序逻辑电路
敏感事件列表	不应包含posedge和negedge关键字	用posedge和negedge关键字描述同步信号的有效跳变沿
	应包含所有输入信号	不一定要包含所有输入信号
所有被赋值的信号	声明为reg型	声明为reg型
所有的赋值语句	一律采用阻塞赋值语句	一律采用非阻塞赋值语句

4、行为建模方式

❖ （2）always结构语句：

- 举例：组合逻辑电路

```
moduleFirst_M (A, B, C, D, E, F );
    input       A, B, C, D, E;
    output      reg   F; //有赋值的输出变量必须是reg型
    always @( A or B or C or D or E)
        //等价于always @( A, B, C, D, E)，也等价于always @(*)
        F = ~ (( A & B &C) | ( D &E ));   //逻辑功能描述
endmodule
```

4、行为建模方式

❖ （2）always结构语句：

- 举例：时序逻辑电路

```
module          UP_DFF (
    Input       CLK,
    input       CLR,
    input       D,
    output      reg   Q);
    always @( negedge CLR or posedge CLK)
    begin
        if (!CLR)          Q <= 1’b0;     //异步清零
        else               Q <= D ;       //触发
    end
endmodule
```

4、行为建模方式

❖（3）过程赋值语句：

- 用于对寄存器型变量赋值，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：
  - 阻塞赋值：串行执行

    变量 = 赋值表达式;
  - 非阻塞赋值：并行执行

    变量 <= 赋值表达式;
  - 主要区别：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；而非阻塞赋值的各条语句是同时执行的。

4、行为建模方式

❖（3）过程赋值语句：

程序1：阻塞赋值

```
always @(posedge clk)
begin
        reg1 = in1;
        reg2 = in2 ^ in3;
        reg3 = reg1;  //reg1的新值
end
```

程序2：非阻塞赋值

```
always @(posedge clk)
begin
        reg1 <= in1;
        reg2 <= in2 ^ in3;
        reg3 <= reg1;  //reg1的旧值
end
```

4、行为建模方式

- （4）其他语句：
  - if-else语句和case语句：自学
    - 注意对于没有完全枚举的情况，需要用else和default给出默认值。
  - 循环语句：尽量少用，比较耗费资源。

The End!