硬件描述语言简介

Verilog HDL的发展历史

- 1984: Gateway Design Automation 推出 Verilog 初版
- 1989: Gateway 被Cadence Design Systems 公司收购
- 1990: Cadence 向业界公开 Verilog HDL 标准
- 1993: OVI 提升 the Verilog 标准，但没有被普遍接受
- 1995: IEEE 推出 Verilog HDL (IEEE 1364-1995)标准
- 2001: IEEE 推出 Verilog IEEE Std1364-2001 标准
- 2002: IEEE 推出 Verilog IEEE Std1364.1-2002 标准
- 2002: Accellera 对 SystemVerilog 3.0 进行标准化
  - Accellera 是OVI & VHDL International (VI)合并后的国际标准化组织
- 2003: Accellera 标准化后的SystemVerilog 3.1
- 2006: IEEE 推出带SystemVerilog 扩展的Verilog新标准

为什么称 SystemVerilog 3.x?

- SystemVerilog 是 对Verilog 革命性的扩展
- Verilog 1.0
  - IEEE 1364-1995 “Verilog-1995” 标准
  - 第一代 IEEE Verilog 标准
- Verilog 2.0
  - IEEE 1364-2001 “Verilog-2001” 标准
  - 第二代 IEEE Verilog 标准
  - 显著提升了 Verilog-1995 标准的性能
- SystemVerilog 3.x
  - 国际标准化组织对Verilog-2001的扩展
  - 第三代 Verilog 标准
  - DAC-2002 - SystemVerilog 3.0
  - DAC-2003 - SystemVerilog 3.1

VHDL: VHSIC Hardware Description Language

VHSIC—Very High Speed Integrated Circuit（1982年）由美国国防部(DOD)制定，以作为各合同商 之间提交复杂电路设计文档的一种标准方案

1985年完成了该标准方案的第一版，1987年成为IEEE标准，即IEEE-1076标准（VHDL'87）。

1988年，美国国防部规定所有官方的ASIC设计必须以VHDL为设计描述语言（美国国防部标准MIL-STD-454L）。

1993年， IEEE对VHDL标准作了若干修改、更新，从更高的抽象层次和系统描述能力上扩展VHDL的内容，升级为新的VHDL标准IEEE-1164（VHDL'93）。

1996年，IEEE将电路综合的标准程序与规格引入VHDL，成为VHDL综合标准IEEE-1076.3。

2000年升级为VHDL标准(IEEE STD 1076-2000)

2008年升级为VHDL标准(IEEE STD 1076-2008)

Verilog HDL与数字系统设计

Verilog HDL简介

第一节 引言

Verilog HDL是一种应用广泛的硬件描述语言，可用于从算法级、门级到开关级的多种抽象层次的数字系统设计。

这种语言具有简捷、高效、易学易用、功能强等优点．因此逐渐为众多设计者所接受和喜爱。

Verilog HDL语言于1995年成为IEEE标准，称为IEEE Standard 1364-1995。

从语法结构上看，Verilog HDL与C语言有许多相似之处，并继承和借鉴了C语言的多种操作符和语法结构。

Verilog HDL的一些主要特点：

能形式化地表示电路的结构和行为。

借用高级语言的结构和语法，例如条件语句、赋值语句和循环语句等。在Verilog HDL中都可以使用，既简化了电路的描述，又方便了设计人员的学习和使用。

能够在多个层次上对所设计的系统加以描述，从开关级、门级、寄存器级(RTL)到功能级和系统级，都可以描述。设计的规模可以是任意的，不对设计的规模施加任何限制。

Verilog HDL具有混合建模能力，即在一个设计中各个模块可以在不同设计层次上建模和描述。

基本逻辑门，例如and、or和nand等都内置在语言中；开关级结构模型，例如pmos和nmos等也被内置在语言中，用户可以直接调用。

用户定义原语(UDP)创建的灵活性。用户定义的原语既可以是组合逻辑原语，也可以是时序逻辑原语。Verilog HDL还具有内置逻辑函数。

第二节 Verilog HDL基本结构

一、Verilog HDL例子

1．一个8位全加器的Verilog HDL的源代码

```
module adder8(cout,sum,ina,inb,cin);
output[7:0] sum;
output cout;
input[7:0] ina,inb;
input cin;
assign {cout,sum} = ina + inb + cin;
endmodule
```

2．一个8位计数器的Verilog HDL的源代码

```
module counter8(out,cout,data,load,cin,clk);
output[7:0] out;
output cout;
input[7:0] data;
input load,cin,clk;
reg[7:0] out;
always @(posedge clk)
  begin
    if (load)
        out = data;
    else
        out = out + cin;
  end
assign cout = &out&cin;
endmodule
```

可以看出：

① Verilog HDL程序是由模块构成的。每个模块的内容都是嵌在module和endmodule两个语句之间，每个模块实现特定的功能，模块是可以进行层次嵌套的。

② 每个模块首先要进行端口定义，并说明输入(input)和输出(output)，然后对模块的功能进行逻辑描述。

③ Verilog HDL程序的书写格式自由，一行可以写几个语句，一个语句也可以分多行写。

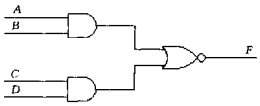
④ 除了endmodule语句外，每个语句的最后必须有分号。

⑤ 可以用 /*……*/ 和 //……对Verilog HDL程序的任何部分作注释。

二、Verilog HDL模块的结构

Verilog HDL的基本设计单元是“模块(b1ock)”。一个模块由两部分组成，一部分描述接口；另一部分描述逻辑功能。

例：下图是一个“与—或—非”门电路。



该电路表示的逻辑函数可以写为：

$$F = \overline{AB + CD}$$

用Verilog HDL可对该电路描述如下：

```
module AOI (A,B,C,D,F);          //模块名为 AOI(端口列表 A,B,C,D,F)
input A,B,C,D;                   //定义模块的输入端口 A,B,C,D
output F;                        //定义模块的输出端口 F
  assign F= ~((A&B)|(C&D));      //模块内的逻辑描述
endmodule
```

Verilog HDL结构完全嵌在module和endmodule声明语句之间，每个 Verilog HDL程序包括4个主要部分：

端口定义

I / O说明

信号类型声明

功能描述

1. 模块的端口定义

模块的端口声明给出了模块的输入和输出口。其格式如下：

module模块名（口1，口2，口3，口4，……）

2. 模块内容

包括I / O说明，信号类型声明和功能定义。

(1) I / O说明的格式如下：

输入口：input端口名1，端口名2，……端口名N；

输出口：output端口名1，端口名2，……端口名N；

I / O说明也可以写在端口声明话句里。其格式如下：

module module-name(input port1，input port2，…output port1,output port2,…);

(2) 信号类型声明：

用来说明逻辑描述中所用信号的数据类型及函数声明。

如在上一节的计数器模块中：

reg[7:0] out;　　// 定义out的数据类型为reg(寄存器)型

对于端口信号的缺省定义类型为wire (连线) 型。

三、 逻辑功能定义

模块中最重要的部分是逻辑功能定义。在模块中进行逻辑描述有3种常用的方法。

用“assign”语句

用元件例化 (instaniate)

用“always” 块语句

1．用“assign”语句

如：assign F＝~((A ＆ D) | (C ＆ D));

这种方法的句法很简单，只须写一个“assign”，后面再加一个方程式即可。

“assign”语句一般适合于对组合逻辑进行赋值，称为连续赋值方式。

2．用元件例化 (instantiate)

如：and myand3(f，a，b，c);

这个语句利用Verilog HDL提供的与门库，定义了一个三输入的与门。采用实例元件的方法同在电路图输入方式下调入库元件一样，键入元件的名字和引脚的名字即可。要求每个实例元件的名字必须是惟一的。

3．用“always” 块语句

在上一节的计数器模块中：

```
a1ways @ (posedge  cIk)   // 每当时钟上升沿到来时执行一
begin                     //遍块内语句
 if ( load )
   out = data;
 else
   out = out + cin;
end
```

“always” 块可用于产生各种逻辑，常用于描述时序逻辑。这个例子用“always”块生成了一个带有同步置数的计数器。“a1ways” 块可用很多种描述手段来表达逻辑，如此例中就用了 if else语句来表达逻辑关系。

综上所述，可给出verilog HDL模块的模板如下:

```
module  ＜顶层模块名＞（＜输入输出端口列表＞）；
outpot   输出端口列表；   // 输出端口声明
input    输入端口列表；   // 输入端口声明

/ * 定义数据，信号的类型，函数声明，用关键字wire，reg，task，function等定义 */

// 使用assign语句定义逻辑功能
wire 结果信号名；
assign ＜结果信号名＞＝＜表达式＞；
```

```
// 使用always块描述逻辑功能
always @ (＜敏感信号表达式＞)，
 begin
// 过程赋值
// if语句
// case语句
// while，repeat，for循环语句
// task，function 调用
end
```

```
// 模块元件例化
＜module-name模块名＞ ＜instance-name例化元件名＞ （＜port-list 端口列表＞）；

// 门元件例化
gate-type-keyword＜instance-name例化元件名＞(＜port-list＞);
endmodule
```

第三节 数据类型及常量、变量

verilog HDL中共有19种数据类型。

4个最基本的数据类型是

integer型、 parameter型、 reg型 和wire型。

其他的类型有

large型、 medium型、 scalared型、 time型、

small型、 tri型、 trio型、 tril型、

triand型、 trior型、 trireg型、 vectored型、

wand型 和wor型等。

verilog HDL中也有常量和变量之分，它们分别属于以上这些类型。

一、常量

在程序运行过程中，其值不能被改变的量称为常量。下面首先对在可综合的verilog HDL语言中使用的数字及其表示方式进行介绍。

1．数字

(1) 整数

在verilog HDL中，整数型常量(即整常数)有以下4种进制表示形式:

二进制整数(b或B)；

十进制整数(d或D)；

十六进制整数(h或H)；

八进制整数(o或O)。

完整的数字表达式为：

＜位宽＞’＜进制＞＜数字＞

位宽为对应二进制数的宽度，如：

8’b11000101 // 位宽为8位的二进制数11000101

8’hc5 // 位宽为8位的十六进制数c5；

十进制的数可以缺省位宽和进制说明，如：

197 // 代表十进制数197

(2) x和z值

x表示不定值，z表示高阻值。

每个字符代表的宽度取决于所用的进制，例如：

8’b1001xxxx； 等价于8’h9x；

8’b1010zzzz； 等价于8’haz；

在case语句中使用x进行匹配，可增强程序的可读性。在较长的数之间可用下划线分开，如16’b1010_1101_0010_1001。当常量不说明位数时，默认值为32位。此外，“?”是高阻态z的另一种表示符号。

2. parameter

在verilog HDL中，用parameter来定义常量，即用Parameter来定义一个标志符，代表一个常量，称为符号常量。其定义格式如下：

parameter 参数名1 = 表达式，参数名2 = 表达式，参数名3 = 表达式……；

例如：

parameter se1＝8，code＝8’ha3；

// 分别定义参数sel为常数8(十进制)，参数code为常数a3(十六进制)

又如：

parameter datawidth = 8，addrwidth = datawidth*2；

在上句中，是用常数表达式进行赋值的。

二、变量

变量是在程序运行过程中其值可以改变的量。变量分为两种：一种为网络型(nets type)，另一种为寄存器型(register type)。下面分别对这两种变量进行说明，并着重介绍其中最常用的wire、reg两种变量类型以及由它们构成的向量(vectors)和数组(array)。

1．nets变量

nets型变量指输出始终根据输入的变化而更新其值的变量，它一般指的是硬件电路中的各种物理连接。Verilog HDL中提供了多种nets型变量，具体如下表所示。

常用的 nets 型变量及说明

类　　型	功 能 说 明
wire，tri	连线类型（wire 和 tri 功能完全相同）
wor，trior	具有线或特性的连线（两者功能一致）
wand，triand	具有线与特性的连线（两者功能一致）
tri1，tri0	分别为上拉电阻和下拉电阻
supply1，supply0	分别为电源（逻辑 1）和地（逻辑 0）

这里着重介绍wire型变量。

wire是一最种常用的nets型变量，wire型数据常用来表示以assign语句赋值的组合逻辑信号。Verilog HDL模块中的输入／输出信号类型缺省时自动定义为wire型。wire型信号可以用做任何方程式的输入，也可以用做“assign”语句和例化元件的输出。对于综合而言，其取值为0，1，X，Z。

wire型变量的定义格式如下：

wire 数据名1，数据名2，……数据名n；

例如：

wire a, b；　　// 定义了两个wire型变量a，b

上面两个变量a，b的宽度都是1位，若定义一个向量(vectors)，可按以下方式：

wire [n−1：0] 数据名1, 数据名2，……数据名m；

wire [n：1]　数据名1，数据名2，……数据名m；

它们定义了数据的宽度为n位。

如下面定义了8位宽的数据总线，20位宽的地址总线：

```
Wire[7：0] databus；
Wire[19：0] addrbus;
或
Wire[8：1] databus；
Wire[20：1] addrbus;
```

wire型向量可按以下方式使用：

```
wire [7：0] in, out ;    // 定义了两个8位wire型向量in, out
assign out=in；
```

若只使用其中某几位，可直接指明，但应注意宽度要一致。如：

```
Wire[7：0] out;
wire [ 3：0] in;
assign out [5：2]=in；    //out向量的第2到第5位与in向量相等
即等效于：  assign out[5] = in[3];
            assign out[4] = in[2];
            assign out[3] = in[1];
            assign out[2] = in[0];
```

2. register型变量

register型变量对应的是具有状态保持作用的电路元件，如触发器、寄存器等。register型变量与nets型变量的根本区别在于：reegister型变量需要被明确地赋值，并且在被重新赋值前一直保持原值。在设计中必须将寄存器型变量放在过程块语句(如initial, always)中，通过过程赋值语句赋值。另外，在always，initial等过程块内被赋值的每一个信号都必须定义成寄存器型。

verilog HDL中，有4种寄存器型变量，具体如下表所示。

常用的rigister型变量及说明

类　　型	功　能　说　明
reg	常用的寄存器型变量
integer	32 位带符号整数型变量
real	64 位带符号实数型变量
time	无符号时间变量

integer、real和time等3种寄存器型变量都是纯数学的抽象描述。reg型变量是最常用的一种寄存器型变量，下面着重对其进行介绍。

reg型变量的定义格式类似于wire型，具体格式为：

reg 数据名1，数据名2，……数据名n；

例如：

reg a, b;　　// 定义了两个reg型变量a, b

上面两个变量a，b的宽度都是1位，若定义一个向量，可按以下方式：

reg[n－1：0] 数据名1，数据名2，……数据名m；

reg[n：1] 数据名1，数据名2，……数据名m；

它们定义了数据的宽度为n位。如下面的语句定义了8位宽的数据：

reg[7：0] data；　　// 定义data为8位宽的reg型向量

或 reg [8: 1] data；

3. 数组

若干个相同宽度的向量构成数组，reg型数组变量即为memory型变量。即可定义存储器型数据。如：

reg [7：0] mymem[1023：0]；

上面的语句定义了一个1024个字节、每个字节宽度为8位的存储器。通常，存储器采用如下方式定义：

parameter wordwidth＝8，memsize＝1024；

reg [wordwidth-1：0] mymem [memsize-1：0]；

上面的语句定义了一个宽度为8位、1024个存储单元的存储器mymem，若对该存储器中的某一单元赋值的话，采用如下方式：

mymem [8]＝1；　// mymem存储器中的第8个单元赋值为1

注意：verilog HDL中的变量名、参数名等标记符是对大小写字母敏感的。

第四节 运算符及表达式

verilog HDL的运算符范围很广，按功能分包括以下几类：

算术运算符、 逻辑运算符、 关系运算符、
等式运算符、 缩减运算符、 条件运算符、
位运算符、 移位运算符 拼接运算符 共9类。

如果按运算符所带操作数的个数来区分，运算符可分为3类，分别为：

单目运算符(unary operator) 运算符可带一个操作数；
双目运算符(binary operator) 运算符可带两个操作数；
三目运算符(ternary operator) 运算符可带三个操作数。
下面对这些运算符分别做说明。

一、算术运算符（Arithmetic operators）

常用的算术运算符包括：

十 加
— 减
* 乘(常数或乘数是2的整数次幂数)
/ 除(常数或除数是2的整数次幂数)
% 求模(常数或右操作数是2的整数次幂数)

以上的算术运算符都属于双目运算符。前面4种用于常用的加、减、乘、除四则运算，%是求模运算符，或称为求余运算符，比如9%4的值为1，6%3的值为0。

二、逻辑运算符（Logical operators）

&& 逻辑与
|| 逻辑或
！ 逻辑非
如A的非表示为：！A；
A和B的与表示为A&&B；
A和B的或表示为A || B；

三、位运算符（Bitwise operators）

位运算是将两个操作数按对应位进行逻辑运算。位运算包括：

~ 按位取反
& 按位与
| 按位或
^ 按位异或
^~，~^ 按位同或（符号^~与~^是等价的）
如：A=5'b11001；B=5'b10101；则
~A＝5'b00110；
A＆B＝5'b10001；
A | B＝5'b11101；
A^B＝5'b01100。
需要注意的一点是两个不同长度的数据进行位运算时，会自动地将两个操作数按右端对齐，位数少的操作数会在高位用0补齐。

四、关系运算符（Relational operators）

关系运算符包括：

＜ 小于
＜＝ 小于或等于
＞ 大于
＞＝ 大于或等于
注：其中“＜＝”操作符也用于表示信号的一种赋值操作。

在进行关系运算时，如果声明的关系是假，则返回值是0；如果声明的关系是真，则返回值是1；如果某个操作数的值不定，则其结果是模糊的，返回值是不定值。

五、等式运算符（Equalityoperators）

等式运算符有4种，分别为：

＝＝ 等于
!= 不等于
＝＝＝ 全等
!＝＝ 不全等

这4种运算符都是双目运算符，得到的结果是1位的逻辑值。如果得到，说明运算结果为真；如果得到0，说明运算结果为假。

相等运算符(＝＝)和全等运算符(＝＝＝)的区别是参与比较的两个操作数必须逐位相等，其相等比较的结果才为1，如果某些位是不定态或高阻值，其相等比较得到的结果就会是不定值。而全等比较(＝＝＝)是对这些不定态或高阻值的位也进行比较，两个操作数必须完全一致，其结果才为1，否则结果是0。

比如：设寄存器变量a＝5'b11x01，b＝5'b11x01，则“a＝＝b”得到的结果为不定值x，而“a＝＝＝b”得到的结果为1。

六、缩减运算符（Reduction operators）

缩减运算符是单目运算符，它包括下面几种：

& 与

~& 与非

| 或

~| 或非

^ 异或

^~，~^ 同或

缩减运算符与位运算符的逻辑运算法则一样，但缩减运算是对单个操作数进行与、或、非递推运算的。如：

reg [3：0] a；

b＝& a； 等效于b＝((a [0] & a [1]) & a [2]) & a [3]；

例：若A＝5'b11001，则：

&A＝0； // 只有A的各位都为1时，其与缩减运算的值才为1。

| A＝1； // 只有A的各位都为0时，其或缩减运算的值才为0。

七、移位运算符（Shist operators）

移位运算符包括：

＞＞ 右移

＜＜ 左移

verilog HDL的移位运算符只有左移和右移两个。其用法为：

A＞＞n或A＜＜n

表示把操作数A右移或左移n位，同时用0填补移出后的空位。

例：假如A＝5'b11001，则：

A＞＞2的值为5'b00110； // 将A右移2位，用0填补移出后的空位。

八、条件运算符（Conditional operators）

条件运算符为：

？：

它是一个三目运算符，对3个操作数进行运算，其定义同C语言中的定义一样，方式如下：

signal＝condition ？ true_expression ： false_expression；

即：信号＝条件？表达式 1 ：表达式2；

当条件成立时，信号取表达式1的值，反之取表达式2的值。

例如对2选1的MUX可描述如下：

out＝sel ？ in1 ： in0 ；

即sel＝1时 out＝in1 ； sel＝0时out＝in0。

九、位拼接运算符（Concatenation operators）

位拼接运算符：

{ }

它将两个或多个信号的某些位拼接起来。用法如下：

{ 信号1的某几位，信号2的某几位，……，信号n的某几位}

例如，在进行加法运算时，可将进位输出与和拼接在一起使用：

output [3 ： 0] sum； // sum代表和

output cout； // cout为进位输出

input [3 ： 0]，ina，inb；

input cin；

assign {cout，sum}＝ina＋inb＋cin； // 进位与和拼接在一起

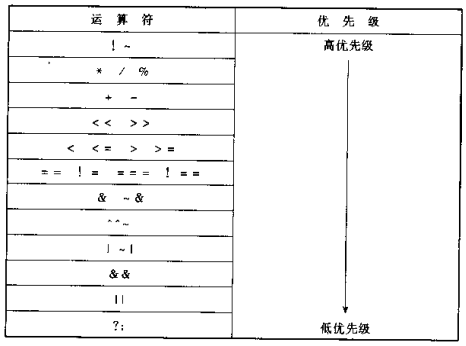
位拼接可以嵌套使用，还可以用重复法来简化书写，如：

{3{a，b}}等同于{{a，b}，{a，b}，{a，b}}，也等同于{a，b，a，b，a，b}。

十、运算符的优先级

以上运算符的优先级如下表所示。为避免出错，同时为增加程序的可读性，在书写程序时可用括号（）来控制运算的优先级。

运算符的优先级



第五节 语 句

Verilog HDL支持许多语句，从而成为结构化和过程性的语言。 Verilog HDL的语句包括：

赋值语句、

条件语句、

循环语句、

结构说明语句、

编译预处理语句。

每一类又包括几种不同的语句，具体如下表所示。

Verilog HDL的语句

赋值语句	连续赋值语句
	过程赋值语句
条件语句	if-else 语句
	case 语句
循环语句	forever 语句
	repeat 语句
	while 语句
	for 语句
结构说明语句	initial 语句
	always 语句
	task 语句
	function 语句
编译预处理语句	'define 语句
	'include 语句
	'timescale 语句

第六节 赋 值 语 句

一、常用的赋值语句

在verilog HDL中，信号有以下赋值方式和赋值语句。

1．连续赋值语句（Continuous Assignments）

assign为连续赋值语句，它用于对wire型变量进行赋值。例如：

```
assign c＝a&b;
```

在上面的赋值中，a、b、c三个变量皆为wire型变量，a和b信号的任何变化，都将随时反映到c上来，因此称为连续赋值方式。

2．过程赋值语句（Procedural Assignments）

过程赋值语句用于对寄存器类型（reg）的变量进行赋值。过程赋值有以下两种方式。

(1)非阻塞(non-blocking)赋值方式

赋值符号为 <＝，如 b<＝a;

非阻塞赋值在块结束时才完成赋值操作，即b的值不是立刻改变的。

(2)阻塞(b1ocking)赋值方式

赋值符号为 ＝，如b = a;

阻塞赋值在该语句结束时就完成赋值操作，即b的值在该赋值语句结束后立刻改变。如果在—个块语句中，有多条阻塞赋值语句．那么在前面的赋值语句没有完成之前，后面的语句就不能被执行，就像被阻塞(blocking)了一样，因此称为阻塞赋值方式。

二、阻塞赋值和非阻塞赋值的区别

阻塞赋值方式和非阻塞赋值方式的区别常给设计人员带来问题。问题主要是对“a1ways”模块内的reg型变量的赋值不易把握。为区分非阻塞赋值与阻塞赋值的不同，可看下面两例：

非阻塞赋值

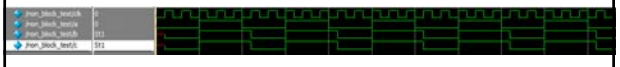
```
module non_block(c,b,a,clk);
input clk,a;
output c,b;
reg c,b;
always @(posedge clk)
  begin
    b<=a;
    c<=b;
  end
endmodule
```

阻塞赋值

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

将上面两段代码用 Modelsim 进行仿真，可分别得到如下图所示的波形。

非阻塞赋值仿真波形图

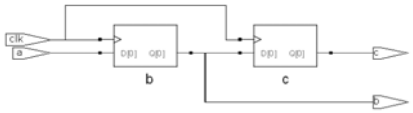


阻塞赋值仿真波形图

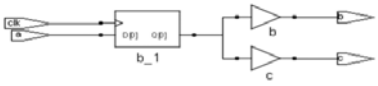
对于非阻塞赋值，c的值落后b的值一个时钟周期，这因为该“always”块每一个时钟周期执行一次，因此信号的值每一个时钟周期更新一次，c的值是上一时钟周期的b值。

对于阻塞赋值，c的值和b的值一样，这是因为b的值是立即更新的。

用Synplify Premier 综合可得出这两种描述所对应的电路分别如下图所示。



非阻塞赋值描述的电路



阻塞赋值描述的电路

第七节 条 件 语 句

条件语句有if-else语句和case语句两种。它们都是顺序语句，故应在“always”块内。下面对这两种语句分别进行介绍。

一、if-else语句

其格式与c语言中if-else语句类似，使用方法有以下3种。

① if (表达式) 语句1;

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3) 语句3;
……
else if (表达式n) 语句n;
else 语句 n + 1;

三种方式中，“表达式”一般为逻辑表达式或关系表达式，也可能是一位的变量。系统对表达式的值进行判断，若为0，x，z，按“假”处理；若为1，按“真”处理，执行指定语句。语句可是单句，也可是多句，多句时用“begin-end”语句括起来。对于if语句的嵌套，若不清楚if和else的匹配，最好用“begin-end”语句括起来。

下例是用if-else语句实现的模为60的BCD码加法计数器。

```
module count60(qout,cout,data,load,cin,reset,clk);
output[7:0] qout;
output cout;
input[7:0] data;
input load,cin,clk,reset;
reg[7:0] qout;
always @(posedge clk)                          //上升沿时刻计数
 begin
     if (reset)          qout = 0;             //同步复位
     else  if (load)     qout = data;          //同步置数
     else  if (cin)
         begin
             if(qout[3:0] = =9)                //低位是否为 9,是则往下执行
             begin
               qout[3:0] = 0;                  //回 0,并判断高位是否为 5
               if(qout[7:4] = =5)
                   qout[7:4] = 0;
               else
                 qout[7:4] = qout[7:4] + 1;    //高位不为 5,则加 1
             end
             else                              //低位不为 9,则加 1

              qout[3:0] = qout[3:0] + 1;
         end
 end
assign cout = ((qout = =8'h59) &cin)? 1:0;     //产生进位输出信号
endmodule
```

二、case语句

相对if语句只有两个分支而言，case语句是一种多分支语句，故case语句多用于多条件译码电路，如描述译码器、数据选择器、状态机及微处理器的指令译码等。case语句有case、casez、casex共3种表示方式。下面分别予以说明。

1. Case语句

case语句的使用格式如下：

```
case ( 敏感表达式 )
   值1：语句1;              // case分支项
   值2：语句2;
   ……
   值n：语句n;
   default：语句n + 1;      // default语句可以省略
endcase
```

当敏感表达式的值为值1时，执行语句1；为值2时，执行语句2；依此类推，如果敏感表达式的值与上面列出的值都不相同的话，则执行default后面的语句。default语句如果不需要，可以去掉。

例：用case语句编写的BCD码—七段数码管译码电路。

```
module decode4_7(decodeout,indec);
output[6:0] decodeout;   //decodeout 为输出到七段数码管的信号
input[3:0] indec;        //indec 为输入的 4 位 BCD 码
reg[6:0] decodeout;
always @(indec)
 begin
  case(indec)            //使用 case 语句进行译码
    4'd0:decodeout = 7'b1111110;
    4'd1:decodeout = 7'b0110000;
    4'd2:decodeout = 7'b1101101;
    4'd3:decodeout = 7'b1111001;
    4'd4:decodeout = 7'b0110011;
```

```
        4'd5:decodeout = 7'b1011011;
        4'd6:decodeout = 7'b1011111;
        4'd7:decodeout = 7'b1110000;
        4'd8:decodeout = 7'b1111111;
        4'd9:decodeout = 7'b1111011;
        default: decodeout = 7'bx;
    endcase
  end
endmodule
```

2. casez与casex语句

case语句中，敏感表达式与值1～值n间的比较是一种全等比较，必须保证两者的对应值全等。casez与casex语句是case语句的两种变体，三者的表示形式中惟一的区别是三个关键词case、casez、casex的不同。在casez语句中，如果分支表达式某些位的值为高阻z，那么对这些位的比较就不予考虑，因此只需关注其他位的比较结果。而在casex语句中，则把这种处理方式进一步扩展到对x的处理，即如果比较的双方有一方的某些位的值是x或z，那么这些位的比较就都不予考虑。

此外，还有另外一种标识x或z的方式，即用表示无关值的“?”来表示。

下面是一个采用casez语句描述并使用了符号“?”的数据选择器的例子。

```
module mux_z(out, a, b, c, d, select);
output out;
input a, b, c, d;
input[3:0] select;
reg out;
always @(select or a or b or c or d)
  begin
      casez (select)
          4'b??? 1: out = a;
          4'b?? 1?: out = b;
          4'b? 1??: out = c;
          4'b1???: out = d;          //后面不需再加 default 语句
      endcase
  end
endmodule
```

3. 使用条件语句注意事项

在使用条件语句时，应注意列出所有条件分支，否则，编译器认为条件不满足时，会引进一个触发器保持原值。这一点可用于设计时序电路，例如在计数器的设计中，条件满足则加1，否则保持不变；而在组合电路设计中，应避免这种隐含触发器的存在。当然，一般不可能列出所有分支，因为每一变量至少有4种取值0，1，z，x。为包含所有分支，可在if语句最后加上else，在case语句的最后加上default语句。为进一步说明这一点，请看下面的例子。

隐含触发器举例：

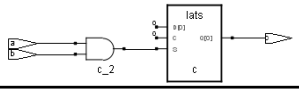
```
module  burie_dff (a,b,c);
output c;
input a,b;
reg c;

always @(a or b)

  if ((b==1)&&(a==1))  c=a&b;

endmodule
```

设计者原意是设计一个二输入与门，但因if语句中无else语句，在对此语句逻辑综合时会认为else语句中为“c=c；”，即保持不变。因此可能形成的电路如下图所示。



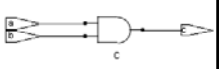
对上例进行仿真时，只要出现“a=1”及“b=1”，则“c=1”后c的值一直都维持为1。

为改正此错误，仅需加上下面的语句即可：

```
else c=0;
```

即“always”块成为：

```
always @(a or b)
begin
  if ((b==1)&&(a==1))  c=a&b;
  else  c=0;
end
```



在利用“if-else”语句设计组合电路时要防止这类错误的发生。

第八节 循环语句

在verilog HDL中存在4种类型的循环语句，用来控制语句的执行次数。这4种语句分别为：

① forever 连续地执行语句，多用在“initial”块中，以生成周期性输入波形。

② repeat 连续执行一条语句n次。

③ while 执行一条语句，直到某个条件不满足。

④ for语句。

一、for语句

for语句的使用格式如下(同C语言)：

for (表达式1；表达式2；表达式3) 语句

即：for (循环变量赋初值；循环结束条件；循环变量增值)

执行语句

下面通过“7人表决器”的例子说明for语句的使用：

通过一个for循环语句统计赞成的人数，若超过4人则通过。用vote[6：0] 表示7人的投票情况，“1”代表赞成，即vote[i] 为“1”代表第i个人赞成，pass＝“1”表示表决通过。

例：用for语句描述的7人投票表决器

```
module voter7(pass,vote);
output pass;
input[6:0] vote;
reg[2:0] sum;
integer i;
reg pass;
always @(vote)
  begin
    sum = 0;
    for(i = 0;i < = 6;i = i + 1)                    //for 语句
        if(vote[i])          sum = sum + 1;
        if(sum[2])           pass = 1;          //若超过 4 人赞成,则 pass = 1
        else                 pass = 0;
  end
endmodule
```

例：使用for循环语句实现两个8位二进制数的乘法操作

```
module mult_for(outcome, a, b);
parameter size = 8;
input[size:1] a,b;                          //两个操作数
output[2 * size:1] outcome;                 //结果
reg[2 * size:1] outcome;
 integer i;

always @(a or b)
  begin
    outcome = 0;
    for(i = 1;i < = size;i = i + 1)          //使用 for 语句
    if(b[i]) outcome = outcome + (a < < (i - 1));
  end
endmodule
```

二、repeat语句

repeat语句的使用格式为：

repeat (循环次数表达式) 语句；

或 repeat (循环次数表达式) begin

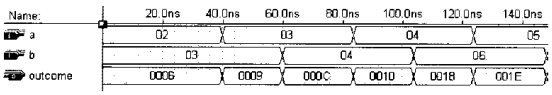
……

end

例：利用repeat 循环语句和移位操作实现两个8位二进制数的乘法:

```
module mult_repeat(outcome, a, b);
parameter size = 8;
input[size:1] a,b;
output[2 * size:1] outcome;
reg[2 * size:1] temp_a,outcome;
reg[size:1] temp_b;
always @(a or b)
  begin
      outcome = 0;
      temp_a = a;
      temp_b = b;
      repeat(size)              //repeat 语句
          begin
      if(temp_b[1])     //如果 temp_b 的最低位为 1,就执行下面的加法
          outcome = outcome + temp_a;
      temp_a = temp_a < < 1;      //操作数 a 左移一位
      temp_b = temp_b > > 1;      //操作数 b 右移一位
      end
  end
endmodule
```

下图所示是该乘法器的功能仿真波形图:



三、while 和 forever 语句

1．while 语句

while 语句的使用格式如下：

while (循环执行条件表达式) 语句；

或 while (循环执行条件表达式) begin

……

end

while语句在执行时，首先判断循环执行条件表达式是否为真。若为真，执行后面的语句或语句块，然后再回头判断循环执行条件表达式是否为真；若为真，再执行一次后面的语句，如此不断，直到条件表达式不为真。因此在执行的语句中，必须有一条改变循环执行条件表达式的值的语句。

2．forever 语句

forever 语句的使用格式如下：

forever 语句；

或 forever begin

……

end

forever循环语句连续不断地执行后面的语句或语句块，常用来产生周期性的波形，作为仿真激励信号。它与always语句的不同之处在于它一般用在initial语句块中。若要用它进行模块描述，可用disable语句进行中断。

第九节 结构说明语句

Verilog HDL中的任何过程模块都从属于以下4种结构说明语句：

initial

a1ways

task

function

在一个模块 (module)中，使用initial和always语句的次数是不受限制的。initial说明语句一般用于仿真中的初始化，仅执行一次；a1ways块内的语句则是不断重复执行的；task和function语句可以在程序模块中的一处或多处调用。

一、always块语句

always @ (＜敏感信号表达式event-expression ＞)

begin

// 过程赋值

// if语句

// case语句

// while，repeat，for循环

// task，function 调用

end

“always”块内的内容前面已经讲过，此处只讨论敏感信号表达式以及如何写敏感信号表达式。

1．敏感信号表达式“event-expression”

敏感信号表达式又称事件表达式或敏感表，当该表达式的值改变时，就会执行一遍块内语句。因此在敏感信号表达式中应列出影响块内取值的所有信号(一般为所有输入信号)。若有两个或两个以上信号时，它们之间用“or”连接。

例如，下面用case语句描述的4选1数据选择器，只要输入信号in0，in1，in2，in3或选择信号sel [1：0] 改变则输出改变，所以敏感信号表达式写为：

in0 or in1 or in2 or in3 or sel

例：用case语句描述的4选1数据选择器

```
module mux4_1(out,in0,in1,in2,in3,sel);
output out;
input in0,in1,in2,in3;
input[1:0] sel;
reg out;
always @(in0 or in1 or in2 or in3 or sel)
    case(sel)
     2'b00:out = in0;
     2'b01:out = in1;
     2'b10:out = in2;
     2'b11:out = in3;
     default:out = 2'bx;
    endcase
endmodule
```

2．posedge与negedge关键字

对于时序电路，事件是由时钟边沿触发的。为表达边沿这个概念，verilog HDL提供了Posedge与negedge两个关键字来描述。比如在计数器逻辑描述中用到：

always @ (posedge clk)　　/ / 上升沿时刻计数

posedge clk表示时钟信号clk的上升沿，negedge clk表示时钟信号clk的下降沿。

在这里，敏感信号表达式中没有列出输入信号data、load、cin、reset，这是因为它是同步置数、同步清零。

这些信号要起作用，必须有时钟的上升沿来到，因此敏感信号表达式中只须列出时钟的上升沿：posedge clk。对于异步的清零 / 置数，应按以下格式书写敏感信号表达式。

如时钟信号为clk，clear为异步清零信号，则敏感信号表达式可写为：

always @ (posedge clk or posedge clear)　// 高电平清零有效

always @ (posedge clk or negedge clear)　// 低电平清零有效

若有其他异步控制信号，均可按此方式加入。注意，在块内的逻辑描述要与敏感信号表达式中信号的有效电平一致。

如，下列描述是错误的：

```
always @(posedge clk or negedge clear)          //低电平清零有效
begin
  if(clear)     //与敏感信号表达式中低电平清零有效矛盾,应改为 if(! clear)
    qout = 0;
  else
    qout = in;
end
```

二、initial 语句

initial 语句使用的格式如下：

```
initial
 begin
 语句1;
 语句2;
 ……
end
```

例：

```
initial
  begin
    reg1 = 0;
    for(addr = 0;addr < size;addr = addr + 1)
    memory[addr] = 0;          //对 memory 存储器进行初始化
  end
```

在上面的例子中．使用initial语句首先将一个变量reg1初始化为0，然后又用for循环语句将memory存储器进行初始化，将其所有的存储单元都置为“0”。

三、task 和 function 语句

task和function语句分别用来定义任务和函数。利用任务和函数可以把一个大的程序模块分解成许多小的任务和函数，以方便调试，并且能使写出的程序清晰易懂。

1. 任务（task）

任务(task)定义与调用的格式分别如下：

定义：task＜任务名＞；

端口及数据类型声明语句；

其他语句；

endtask

任务调用的格式为：

＜任务名＞（端口1，端口2，……）；

下面是一个定义任务的例子：

```
task test;
input in1,in2;
output out1,out2;

    #1 out1 = in1 & in2;
    #1 out2 = in1 | in2;
endtask
```

当调用该任务时，使用如下语句：

test (data1，data2，code1，code2)；

任务调用变量和定义时说明的I / O变量是一一对应的。调用任务test时，变量data1和data2的值赋给in1和in2，而任务完成后输出通过out1和out2赋给了code1和code2。

使用任务时，需要注意的几点是：

① 任务的定义与调用须在一个module模块内。

② 定义任务时，没有端口名列表，但需要在后面进行端口和数据类型的说明。

③ 当任务被调用时，任务被激活。任务的调用与模块调用一样通过任务名调用实现。调用时，需列出端口名列表，端口名的排序和类型必须与任务定义中的排序和类型一致。

④ 一个任务可以调用别的任务和函数，可以调用的任务和函数个数不限。

2．函数（function）

函数的目的是返回一个用于表达式的值。函数的定义格式为:

function ＜返回值位宽或类型说明＞ 函数名；

端口声明；

局部变量定义；

其他语句；

endfunction

例：函数的使用

```
function[7:0] gefun;
input[7:0] x;
reg[7:0] count;
integer i;
  begin
    count = 0;
    for (i = 0;i < = 7;i = i + 1)
        if (x[i] = 1'b0) count = count + 1;
    gefun = count;
  end
endfunction
```

上面的gefun函数循环核对输入的每一位，计算出0的个数，并返回一个适当的值。

＜返回值位宽或类型说明＞是一个可选项，如果缺省，则返回值为一位关于寄存器类型的数据。

函数的定义中蕴含了一个与函数同名的、函数内部的寄存器。在函数定义时，将函数返回值所使用的寄存器设为与函数同名的内部变量，因此函数名被赋予的值就是函数的返回值。例如在上面的例子中，gefun最终赋予的值即为函数的返回值。

函数的调用是通过将函数作为表达式中的操作数来实现的。调用格式如下：

＜函数名＞ (＜表达式＞ ＜表达式＞)；

比如使用连续赋值语句调用函数gefun时，可以采用如下语句：

assign out＝is_legal ? gefun(in)：1’b0；

函数的使用与任务相比有更多的限制和约束。

例如，函数不能启动任务，在函数中不能包含有任何的时间控制语句，同时定义函数时至少要有一个输入参量等。在使用时，这些都需要注意。

下例中定义了一个阶乘运算的函数factorial，该函数返回一个32位的寄存器类型的值。

例：阶乘运算函数

```
module funct(clk,n,result,reset);
output[31:0] result;
input[3:0] n;
input reset,clk;
reg[31:0] result;
always @(posedge clk)
begin
   if(!reset)  result<=0;
      else  begin
               result <= 2 * factorial(n);     //函数的调用
            end
end

function[31:0] factorial;                  //函数的定义
input[3:0] opa;
reg[3:0] i;
  begin
    factorial = opa ? 1 : 0;
      for(i= 2; i <= opa; i=i+1)
     factorial = i* factorial;
  end
endfunction
endmodule
```

3．任务与函数的区别

任务与函数的区别

	任　务(task)	函　数(function)
输入与输出	可有任意个各种类型的参数	至少有一个输入，不能将 inout 类型作为输出
调用	任务只可在过程语句中调用，不能在连续赋值语句 assign 中调用	函数可作为表达式中的一个操作数来调用，在过程赋值和连续赋值语句中均可以调用
调用其他任务和函数	任务可调用其他任务和函数	函数可调用其他函数，但不可以调用其他任务
返回值	任务不向表达式返回值	函数向调用它的表达式返回一个值

在使用Verilog HDL的结构说明语句时。还要注意的一点是有的编译器对某些结构说明语句是不支持的。

如MAX+PLUS II，该软件支持a1ways、function语句，但不支持task和initial语句。因此，设计者在使用时应尽量使用所用编译器支持的语句来描述设计。

第十节 编译预处理语句

Verilog HDL和C语言一样也提供了编译预处理功能。Verilog HDL允许在程序中使用特殊的编译预处理语句。在编译时，通常先对这些特殊语句进行“预处理”，然后再将预处理的结果和源程序一起进行编译。

预处理命令以符号” ´”开头，以区别于其他语句。Verilog HDL提供了十几条编译预处理语句，比较常用的有‘define，‘include和‘timescale等语句，下面分别介绍这些常用的预处理语句。

一、‘define语句

‘define语句用来将一个简单的名字或标志符(也称为宏名）代表一个复杂的名字或字符串，其一般形式为：

‘define 标志符 (宏名) 字符串

如：‘define IN ina+inb+inc+ind

在以上语句中，用简单的宏名IN来代替了一个复杂的表达式ina+inb+inc+ind。采用了这样的形式定义后，在后面的程序中就可以直接用IN来代表表达式ina+inb+inc+ind了。

如程序中若出现：

assign out＝‘IN + ine；

它实际上与assign out＝ina+inb+inc+ind + ine；是完全等价的。

又如：

‘define WORDSIZE 8

reg [‘WORDSIZE ： 1] data； // 相当于定义reg [8：1] data；

从上面的例子可以看出：

① ‘define用于将一个简单的宏名来代替一个字符串或一个复杂的表达式。

② 宏定义语句行末不加分号，这一点尤其要注意。

③ 在引用已定义的宏名时，不要忘了在宏名的前面加上符号“ ‘”，以表示该名字是一个宏定义的名字。

④ 采用宏定义，可以简化程序的书写，而且便于修改。若需要改变某个变量，只须改变‘define定义行，一改全改。如在上面的例子中，定义data是一个8位的寄存器变量，若要将其改为16位，只须将定义行改为“ ‘difine WORDSIZE l6”即可。

二、’include 语句

’include是文件包含语句，它可将一个文件全部包含到另一个文件中，其一般形式为：

’include “文件名”

下面是一个16位加法器的例子。

在这个例子中，adderl6模块使用‘include语句调用了一个通用的加法器adder模块，或者说adderl6模块整个包含了adder模块。

例：用’include语句设计的16位加法器

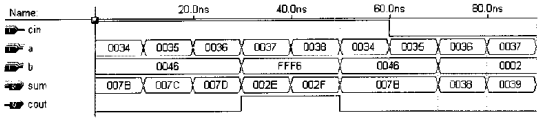
```
`include "adder.v"

module adder16(cout,sum,a,b,cin);
output cout;
parameter my_size=16;
output[my_size-1:0] sum;
input[my_size-1:0] a,b;
input cin;

adder  my_adder (cout,sum,a,b,cin);      //调用adder 模块
endmodule
//下面是adder 模块代码
module adder(cout,sum,a,b,cin);
parameter size=16;
output cout;
output[size-1:0] sum;
input cin;
input[size-1:0] a,b;
assign {cout,sum}=a+b+cin;
endmodule
```

有的软件并不支持‘include语句，如MAX+PLUS II。

这时可先使用Synplify软件将上例进行编译。在编译前将目标器件(Target)选择为Altera 的器件（如FLEX10K），编译后生成adderl6.edf文件，再打开MAX+PLUS II软件，将adderl6.edf文件设为顶层文件，编译仿真，即可得到下图所示的仿真波形。



使用＇include语句时需注意以下几点：

① 一个＇include语句只能指定一个被包含的文件。

② ＇include语句可以出现在源程序的任何地方。被包含的文件若与包含文件不在同一个子目录下，必须指明其路径。

③ 文件包含允许多重包含，比如文件1包含文件2，文件2又包含文件3等。

三、＇timescale语句

＇timescale语句用于定义模块的时间单位和时间精度，其使用格式如下：

＇timescale ＜时间单位＞ / ＜时间精度＞

其中用来表示时间度量的符号有：s、ms、us、ns、ps和fs，分别表示秒、10^{-3}秒、10^{-6}秒、10^{-9}秒、10^{-12}秒和 10^{-15}秒。

下面举例说明＇timescale语句的用法。

例：＇timescale 1ns / 1ps

上面的语句表示本模块的时间单位是1ns，时间精度为1ps，模块中所有的时间都必须是1ns的整数倍，同时模块中的延迟时间可用带3位小数的实数型参量来表示，因为时间精度为1ps。

例：＇timescale语句的应用

```
'timescale   10ns/1ns
……
reg sel;
initial
    begin
     # 10     sel = 0;        //在 10ns × 10 时刻,sel 变量被赋值为 0
     # 10     sel = 1;        //在 10ns × 20 时刻,sel 变量被赋值为 1
    end
    ……
```

在上例中，＇timescale语句定义了本模块的时间单位为10ns，时间精度为1ns。以10ns 为计量单位，在不同的时刻，寄存器型变量sel被赋予了不同的值。

第十一节 语句的顺序执行与并行执行

用verilog HDL模块来设计电路，首先应该清楚哪些操作是同时发生的，哪些是顺序发生的。

在“a1ways”模块内，逻辑是按照指定的顺序执行的，“aLways”块内的语句称为顺序语句，因为这些语句完全按照书写的顺序来执行。

“always”模块之间，是同时执行的，或者说是并行执行的。两个或更多个“a1ways”模块、“assign”语句、例化元件等都是同时执行的。

通过下面的例子，可以看到“a1ways”模块内的语句是顺序执行的，而“a1ways”模块之间是并行执行的。

顺序执行模块1

```
module serial1(q, a, clk);
input clk;
output q,a;
reg q,a;
always @ (posedge clk)
  begin
    q=~q;
    a=~q;
  end
endmodule
```

顺序执行模块2

```
module serial1(q, a, clk);
input clk;
output q,a;
reg q,a;
always @ (posedge clk)
  begin
    a=~q;
    q=~q;
  end
endmodule
```

上面的两个例子，其区别只是在“always”模块内，把两个赋值语句的顺序相互颠倒。

模块1的仿真波形

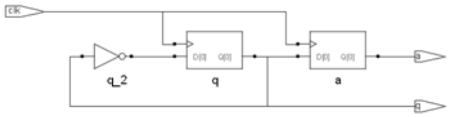
模块2的仿真波形

从波形我们可以看到：

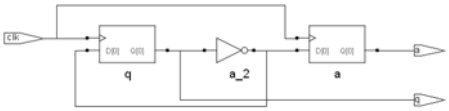
在模块1中，q先取反，然后再取反给a， a和q的波形相位相反；

在模块2中，q取反后赋值给a和q，a和q的波形是完全一样的。

模块1、模块2所对应的电路图如下所示：



模块1对应的电路图



模块2对应的电路图

观察下面两个模块，它们的结果是否相同？

模块1

```
module paral1(q,a,clk);
output q,a;
input clk;
reg q,a;
always @(posedge clk)
  begin
    q = ~q;
  end
always @(posedge clk)
  begin
    a = ~q;
  end
endmodule
```

模块2

```
module paral2(q,a,clk);
output q,a;
input clk;
reg q,a;
always @(posedge clk)
  begin
    a = ~q;
  end
always @(posedge clk)
  begin
    q = ~q;
  end
endmodule
```

第十二节 不同抽象级别的Verilog HDL 模型

Verilog HDL是一种能够在多个级别对数字电路和数字系统进行描述的高级语言，Verilog HDL模型可以是对实际电路的不同级别的抽象。这些抽象级别一般可分为5级：

① 系统级 (System Level)

② 算法级 (Algorithm Level)

③ 寄存器传输级 (RTL，Register Transfer Level)

④ 门级(Gate Level)

⑤ 开关级(Switch Level)

系统级、 算法级、 寄存器传输级 、 门级、 开关级

其中，前3种属于高级别的描述方法，又称为行为级的描述。门级模型是描述逻辑门以及逻辑门之间连接关系的模型。而开关级的模型则是描述器件中三极管和存储节点以及它们之间连接关系的模型。Verilog HDL在开关级提供了一套完整的组合型原语(primitive)，可以精确地建立MOS器件的底层模型。

一、Verilog HDL 门级描述

Verilog HDL中提供了丰富的门类型关键字，用于电路的门级描述。Verilog HDL有关门类型的关键字共有26个，比较常用的有下面几个：

not： 非门
and： 与门
nand： 与非门
or： 或门
nor 或非门
xor： 异或门
xnor： 异或非门
buf： 缓冲器
bufif1，bufif0，notif1，notif0： 各种三态门

调用门原语的句法如下：

门类型关键字 <例化的门名称>（<端口列表>）

端口列表按下列顺序列出：

(输出，输入1，输入2，输入3……)；

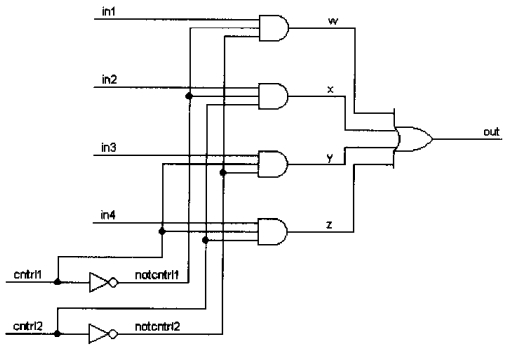
对于三态门，则按如下顺序列出端口：

(输出，输入，使能控制端)；

如： and myand(out，inl，in2，in3)； // 三输入与门

and(out，inl，in2)； // 二输入与门

bufif1 mytri (out，in，enable)； // 高电平使能的三态门



对上图所示的用基本门实现的4选1 MUX的电路，可采用门级Verilog HDL描述如下：

```
module mux4_1(out,in1,in2,in3,in4,cntrl1,cntrl2);
output out;
input in1,in2,in3,in4,cntrl1,cntrl2;
wire notcntrl1,notcntrl2,w,x,y,z;
not (notcntrl1,cntrl1);
not (notcntrl2,cntrl2);
and (w,in1,notcntrl1,notcntrl2);
and (x,in2,notcntrl1,cntrl2);
and (y,in3,cntrl1,notcntrl2);
and (z,in4,cntrl1,cntrl2);
or (out,w,x,y,z);
endmodule
```

二、Verilog HDL的行为级描述

下面分别用逻辑功能、case语句、条件运算符等行为描述方式来实现上例中的4选1 MUX。

1. 逻辑功能描述

例：采用功能描述的4选1 MUX

```
module mux4_1(out,in1,in2,in3,in4,cntrl1,cntrl2);
output out;
input in1,in2,in3,in4,cntrl1,cntrl2;
assign out = (in1 & ~cntrl1 & ~cntrl2)|
             (in2 & ~cntrl1 & cntrl2)|
             (in3 & cntrl1 & ~cntrl2)|
             (in4 & cntrl1 & cntrl2);
endmodule
```

2．case语句描述

例：用case语句描述的4选1 MUX

```
module mux4_1(out,in1,in2,in3,in4,cntrl1,cntrl2);
output out;
input in1,in2,in3,in4,cntrl1,cntrl2;
reg out;
always @(in1 or in2 or in3 or in4 or cntrl1 or cntrl2)
   case({cntrl1,cntrl2})
    2'b00:out = in1;
    2'b01:out = in2;
    2'b10:out = in3;
    2'b11:out = in4;
    default:out = 2'bx;
   endcase
endmodule
```

3．条件运算符

例：用条件运算符描述的4选1 MUX

```
module mux4_1(out,in1,in2,in3,in4,cntrl1,cntrl2);
output out;
input in1,in2,in3,in4,cntrl1,cntrl2;
assign out = cntrl1? (cntrl2? in4:in3): (cntrl2? in2:in1);
endmodule
```

对于设计者而言，采用的描述级别越高，设计越容易。但对于特定的综合器而言，有可能无法将某些抽象级别高的描述转化成电路。所以，设计者应该了解所用综合器的性能，以选择适当的描述级别，或者选择更好更高级的综合软件。

这里介绍了Verilog HDL的语法结构，包括变量、语句、模块和不同抽象级别的电路的设计和描述。在学习这些语法结构时，尤其要注意一些容易出错的地方。如阻塞赋值和非阻塞赋值，顺序执行和并行执行等，必须弄清它们之间的区别。

另外需要注意的是所有的Verilog HDL编译软件都只是支持该语言的某一个子集，有的软件支持的子集大一些，有的支持的子集小一些。在使用Verilog HDL进行编程时，首先必须弄清楚所用编译软件的功能，哪些Verilog HDL语句是被支持的．哪些语句是不被支持的。然后，尽量用编译软件支持的语句来描述所设计的系统。还有一种方法，就是多个软件配合使用。