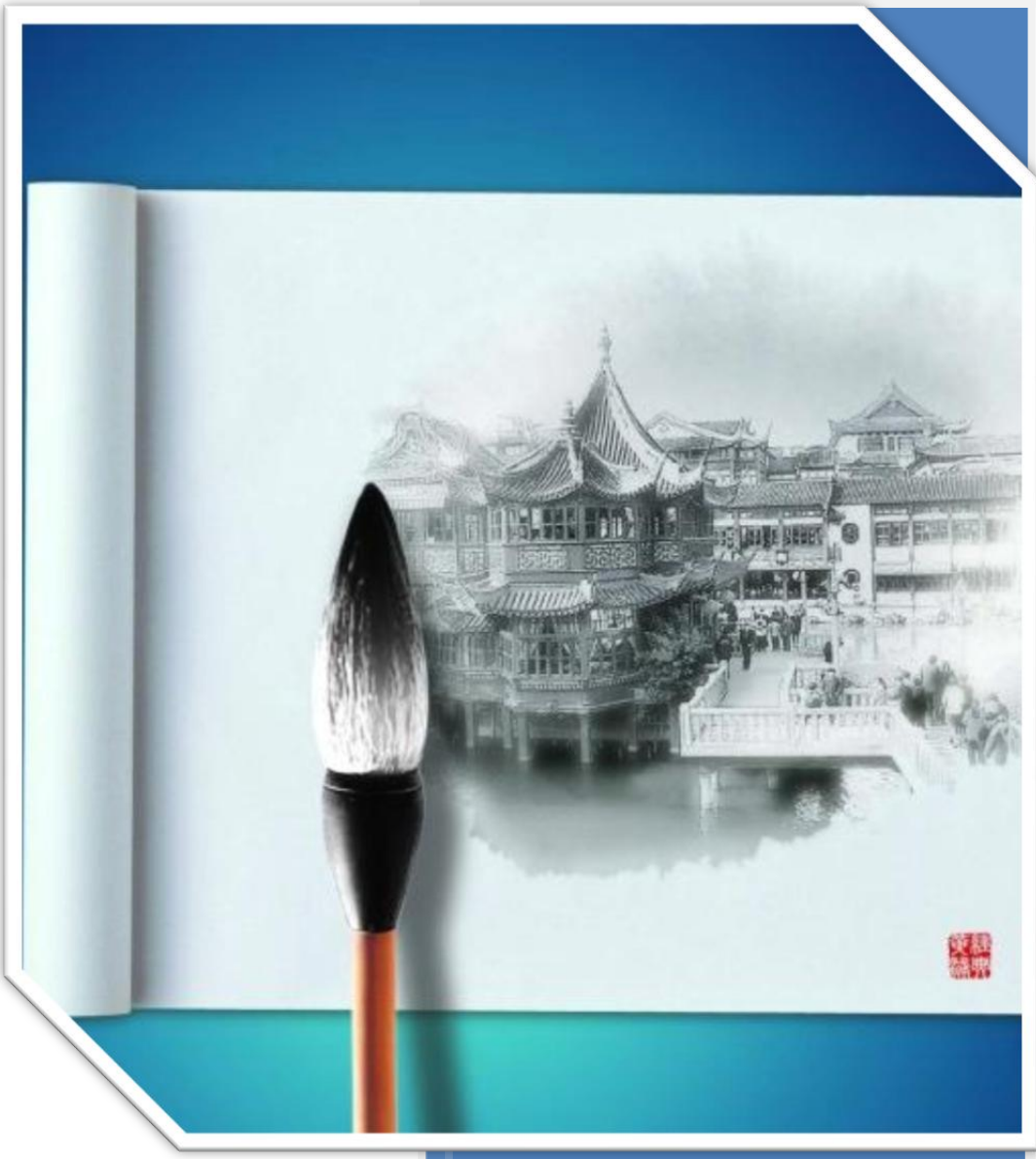
FLQ100IT02-V2.0

# HDL 基础语法篇 Verilog



zrtech

WWW.ZR-TECH.COM

# Verilog HDL硬件描述语言

## 2.1 Verilog HDL概述

### 2.1.1 Verilog HDL的特点

Verilog HDL和VHDL一样，是目前大规模集成电路设计中最具代表性、使用最广泛的硬件描述语言之一。

作为硬件描述语言，Verilog HDL具有如下特点：

1. 能够在不同的抽象层次上，如系统级、行为级、RTL（Register Transfer Level）级、门级和开关级，对设计系统进行精确而简练的描述；
2. 能够在每个抽象层次的描述上对设计进行仿真验证，及时发现可能存在的设计错误，缩短设计周期，并保证整个设计过程的正确性；
3. 由于代码描述与具体工艺实现无关，便于设计标准化，提高设计的可重用性。如果有C语言的编程经验，只需很短的时间内就能学会和掌握Verilog HDL，因此，Verilog HDL可以作为学习HDL设计方法的入门和基础。

### 2.1.2 Verilog HDL的基本结构

Verilog HDL描述是由模块（module）构成的，每个模块对应的是硬件电路中的逻辑实体。因此，每个模块都有自己独立的功能或结构，以及用于与其它模块之间相互通信的端口。例如，一个模块可以代表一个简单的门，一个计数器，一个存储器，甚至是计算机系统等。

例2-1-1 加法器的verilog描述

```
/*************************************************************/
// MODULE:          adder
// FILE NAME:       add.v
// VERSION:         v1.0
// DATE:            May 5th, 2003
// AUTHOR:          Peter
// CODE TYPE:       RTL
// DESCRIPTION:     An adder with two inputs (1 bit), one output (2 bits).
/*************************************************************/

module adder (in1, in2, sum);
input in1,in2;
output [1:0] sum;
wire in1,in2;
reg [1:0] sum;
always @ (in1 or in2)
begin
sum=in1+in2;
end
endmodule
```

从这个例子中可以看出，一段完整的代码主要由以下几部分组成：

第一部分是代码的注释部分，主要用于简要介绍设计的各种基本信息。从上面的注释中可以了解到一些基本信息，如代码中加法器的主要功能、设计工程师、完成的日期及版本。

例2-1-1的模块名是adder，有两个输入端口in1，in2和一个输出端口sum。其中，输入信号是一位的，其数据类型声明为连线型(wire)；输出是两位的寄存器类型。这些信息都可以在注释中注明。这一部分内容为可选项，建议在设计中采用，以提高代码的可维护性。

第二部分是模块定义行，这一行以module开头，然后是模块名和端口列表，标志着后面的代码是设计的描述部分。

第三部分是端口类型和数据类型的说明部分，用于端口、数据类型和参数的定义等等。

第四部分是描述的主体部分，对设计的模块进行描述，实现设计要求。模块中“always-begin”和“end”构成一个执行块，它一直监测输入信号，其中任意一个发生变化时，两个输入的值相加，并将结果赋值给输出信号。这些定义和描述可以出现在模块中的任何位置，但是变量、寄存器、线网和参数的使用必须出现在相应的描述说明部分之后。为了使模块描述清晰和具有良好的可读性，建议将所有的说明部分放在设计描述之前。

第五部分是结束行，就是用关键词endmodule表示模块定义的结束。模块中除了结束行以外，所有语句都需要以分号结束。

# 2.2 Verilog HDL语言要素

## 2.2.1 基本语法定义

Verilog HDL源代码是由大量的基本语法元素构成，其中包括空白部分（White space）、注释（Comment）、运算符（Operator）、数值（Number）、字符串（String）、标识符（Identifier）和关键字（Keyword）。

1. 注释

在代码中添加注释行可以提高代码的可读性和可维护性。Verilog HDL中注释行的定义与C语言完全一致，分为两类：

**第一类是单行注释，以“//”开始到本行行末结束，不允许续行。**

**第二类是多行注释，以“/*”开始，以“*/”结束。可以跨越多行，但是中间不允许嵌套。**

**小提示：**

Verilog同VHDL一样，对于空格不敏感。

2. 运算符

Verilog HDL中运算符可以分以下几类，如表2-2-1所列。

表2-2-1 Verilog HDL的运算符

| 运算符分类 | 所含运算符 |
|---|---|
| 算术运算符 | +，—，*，/，% |
| 关系运算符 | <，>，<=，>= |
| 相等与全等运算符 | ==，!=，===，!== |
| 逻辑运算符 | !，&&，\|\| |
| 位运算符 | ~，&，\|，^，^~或~^ |
| 缩位运算符 | &，~&，\|，~\|，^，^~/或~^ |
| 逻辑移位运算符 | <<，>> |
| 位拼接运算符 | {} |
| 条件运算符 | ?: |

其中，单目运算符包括按位取反运算符（～）、逻辑非（！）以及全部缩位运算符（&, ~&, |, ~|, ^, ^~/ 或 ~^），三目运算符只有条件运算符一个（?:），其余的均为双目运算符。

由于Verilog HDL是在C语言基础上开发的，因此两者的运算符也十分类似，本节将主要介绍与C语言不同的运算符的功能。

(1) 相等与全等运算符

这四个运算符的比较过程完全相同，不同之处在于不定态或高阻态的运算。相等运算中，如果任何一个操作数中存在不定态或高阻态，将得到一个不定态的结果；而在全等运算中，则是将不定态和高阻态看作是逻辑状态的一种，同样参与比较，当两个操作数的相应位都是X或Z时，认为全等关系成立，否则，运算结果为0。相等与全等运算符如表2-2-2所列。

表2-2-2 相等与全等运算符

| | |
|---|---|
| a===b | a 与 b 全等，包括 x,z |
| a!==b | a 与 b 不全等，包括 x,z |
| a==b | a 与 b 相等，结果可能是不定态 |
| a!=b | a 与 b 不相等，结果可能是不定态 |

例2-2-1相等与全等运算符的比较

```
// example for logical equality & case equality
module equality;
initial
begin
$display (“ ’bx ==’bx is %b”, ’bx==’bx );
$display (“ ’bx===’bx is %b”, ’bx===’bx );
$display (“ ’bz ! =’bx is %b”, ’bz !=’bx );
$display (“ ’bz !==’bx is %b”, ’bz !==’bx );
end
endmodule
```

模块equality的执行会产生如下结果：

```
’bx ==’bx is x
’bx===’bx is 1
’bz ! =’bx is x
’bz !==’bx is 1
```

**小提示：**

全等是比较是否完全匹配，只有1，0两个状态；相等则会出现不定态。

(2) 缩位运算符

缩位运算符是对单个操作数进行与、或、非等操作，与逻辑运算符的区别是最终结果和操作数的位数无关，一定是一位的逻辑值。如：

如果a为[3:0];

&a等效于（（a[0]&a[1]）&a[2]）&a[3]。

~|a等效于 ~（（（a[0] | a[1]）| a[2]）| a[3]）。

(3) 移位运算符

移位运算符是完成数据的左移、右移功能，移出的空位用0填补。

(4) 位拼接运算符

位拼接运算符是将两个或多个信号的某些位拼接起来。如

```
{a, b[3:0], w, 3'b101} = {a, b[3], b[2], b[1], b[0], w, 1'b1, 1'b0, 1'b1}
```

(5) 条件运算符

条件运算符是唯一的一个三目运算符。

条件运算符的格式为：

**<条件表达式>？<条件为真时的表达式>：<条件为假时的表达式>**

如对于2选1的MUX可以采用如下描述：

```
output = select ? input1: input2;
```

即select＝1时，输出为input1，否则为input2。

与C语言一样，Verilog HDL也规定了运算符的优先级，如表2-2-3所列。当不同优先级的运算符出现在同一个表达式时，需要遵从优先级的顺序，因此，可以通过添加括号的方式，清晰运算次序，提高代码的可读性。

表2-2-3 运算符的优先级顺序

| 运算符 | 优先级 |
|---|---|
| ! ～ | 最高 |
| * / % | |
| + — | |
| << >> | |
| < <= > >= | |
| == != === !== | |
| & !& | |
| ^ ~^ | |
| \| ~\| | |
| && | |
| \|\| | |
| ?: | 最低 |

3. 数值

Verilog HDL的数值集合由以下四个基本的值组成：

0—— 代表逻辑0或假状态；

1——代表逻辑1或真状态；

X（或x）——代表逻辑不定态；

Z（或z）——代表高阻态。

因此，前面介绍条件运算符时，对于MUX描述语句的解释是：select＝1时，输出为input1，否则为input2。而不能够认为，select＝1时，输出为input1，selece=0时输出为input2。

常数按照其数值类型可以划分为整数和实数两种。Verilog HDL的整数可以是十进制、十六进制、八进制或二进制。

整数定义的格式为：

**<位宽>'<基数><数值>**

位宽：描述常量所含二进制数的位数，用十进制整数表示，是可选项，如果没有这一项，可以从常量的值推断出。

基数：可选项，可以是b(B), o(O) ,d(D), h(H)分别表示二进制、八进制、十进制和十六进制。基数缺省默认为十进制数。

数值：是由基数所决定的表示常量真实值的一串ASCII码。如果基数定义为b或B，数值可以是0, 1, x(X), z(Z)。对于基数是d或D的情况，数值符可以是从0到9的任何十进制数，但不可以是X或Z。举例如下：

```
15          （十进制15）
'h15        （十进制21，十六进制15）
5'b10011    （十进制19，二进制10011）
12'h01F     （十进制31，十六进制01F）
'b01x       （无十进制值，二进制01x）
```

**小提示：**

（1）数值常量中的下划线“_”是为了增加可读性，可以忽略，如“8'b1100_0001”是8位二进制数。

（2）数值常量中的“？”表示高阻状态，如“2'B1?”表示2位的二进制数其中的一位是高阻状态。

Verilog HDL中实数用双精度浮点型数据来描述。实数既可以用小数（如12.79）也可以用科学计数法的方式（如23E7，表示23乘以10的7次方）来表达。带小数点的实数在小数点两侧都必须至少有一位数字。例如：

```
1.2
0.5
128.78329
1.7e8(指数符号可以是e或E)
123.3232_234_32(下划线忽略)
```

下面的几个例子是无效的格式:

```
.25
3.
3.E3
.8e-1
```

**小提示：**

实数可以根据四舍五入的原则转化为整数，将实数赋值给一个整数时，这种转化会自行发生。例如，在转化成整数时，实数25.5和25.8都变成26，而25.2则变成25。 这种赋值方式在VHDL中是不允许的。

4. 字符串

字符串指同一行内写在双引号之间的字符序列串，在表达式和赋值语句中字符串用作操作数，而且要转换成无符号整型常量，用若干个8位二进制ASCII码的形式表示，其中每8位二进制ASCII码代表一个字符。例如，字符串“ab”等价于16’h5758。

**在Verilog HDL中引入字符串的主要目的是配合仿真工具，显示一些按照指定格式输出的相关信息。**因此，需要一些特殊字符配合，输出特定格式，特殊字符如表2-2-4所列。

表2-2-4 特殊字符

| 特殊字符 | 含　义 |
|---|---|
| \n | 换行 |
| \t | Tab 键 |
| \\ | 反斜杠 \ |
| \” | 引号” |
| \ddd | 由三位八进制数表示的 ASIC 值 |
| %% | % |

例2-2-2 字符串

```
module string_test;

reg [8*14:1] stringvar;

initial
begin
stringvar = "Hello world";
$display("%s is stored as %h", stringvar,stringvar);

stringvar = {stringvar,"!!!"};
$display("%s is stored as %h", stringvar,stringvar);
end

endmodule
```

输出结果是：

```
Hello world is stored as 00000048656c6c6f20776f726c64
Hello world!!! is stored as 48656c6c6f20776f726c64212121
```

在Verilog HDL中，字符串赋值过程中，如果字符串的位数超出字符串变量的位数，截掉字符串的高位部分，低位对齐；反之，如果字符串的位数少于字符串变量的位数，高位部分用0补齐。例2-2-2中，stringvar的位宽是8*14位，即112位，而“Hello world”是8*11位，所以，输出结果的第一行出现三个空格；同理，输出数据的前六位用0代替。

5. 标识符

与C语言等一样，Verilog HDL语法中最基本的部分就是标识符。一般说来，Verilog HDL中的标识符有普通标识符、转义标识符，下面分别给以具体说明。

普通标识符的命名规则是：

**（1）必须由字母或者下划线开头，字母区分大小写；**
**（2）后续部分可以是字母、数字、下划线或$；**
**（3）总长度要小于1024个字符串的长度；**

合法的普通标识符举例如下：

```
sdfj_kiu      // 允许在标识符内部包含下划线
_sdfji        // 允许以下划线开头
```

转义标识符指以反斜杠“\”开头，以空白符结尾的任意字符串序列。空白符可以是一个空格、一个TAB键、一个制表符或者一个换行符等。转义字符本身没有意义，如

```
\sfji          // \sfji与sfji等价
\23kie         // 可以以任意可打印的字符开头
\*239d         // 理由同上
```

标识符的第一个字符不能够是“$”，因为在Verilog HDL中，“$”专门用来代表系统命令(如系统任务和系统函数)。

6. 关键字

与其它语言一样，每种语言有自己的保留字。这些字是用户所不能作为名字来使用的标识符。Veriog HDL中约有98个关键词，其中常用的有：

**always endmodule reg and**
**assign begin for case or**
**function output parameter wait**
**if else input while end**

## 2.2.2数据类型

Verilog HDL的数据类型是指在硬件数字电路中数据进行存储和传输的方式。按照物理数据类型分类，Verilog HDL中变量分为线型和寄存器型两种，两者在驱动方式、保持方式和对应的硬件实现都不相同。这两种变量在定义时要设置位宽，缺省值为一位。变量的每一位可以是0，1，x或z，其中x代表一个未被预置初始状态的变量，或是由于两个或更多个驱动装置试图将之设定为不同的值而引起的冲突型变量；z代表高阻状态或悬空状态。

本节主要介绍几种常用的数据类型：参数常量、线型变量和寄存器型变量，以及存储器的定义方式。其他数据类型可以参考IEEE1364-1995中相关定义和规定。

1. 参数（**Parameters**）

参数是常量的一种，经常用来定义延时、线宽、寄存器位数等物理量，可以增加代码的可读性和可维护性。

参数定义的格式为：

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

例2-2-3 参数定义

```
// Parameter example
module example_for_parameters (reg_a, bus_addr, …);
parameter msb=7, lsb=0, delay=10, bus_width=32;
reg [msb:lsb] reg_a;
reg [bus_width:0] bus_addr;
and #delay (out，and1, and2);
...
```

```
endmodule
```

说明：

（1）例2-2-3中用文字参数delay代替延时常数10，用bus_width代替总线宽度常数32，用msb和lsb分别代替最高有效位7和最低有效位0，增加了代码的可读性，并方便修改设计；

（2）可以在一条语句中定义多个参数，中间用逗号隔开；

（3）对于含有参数的模块通常称为参数化模块。参数化模块的设计，体现出可重用设计的思想，在仿真中也有很大的作用。

**2.** 线型变量（**Nets**）

线型变量通常表示硬件电路中元器件间的物理连接。它的值由驱动元件的值决定，并具有实时更新性。

Verilog HDL提供了多种线型变量，与电路中各种类型的连线相对应，具体如表2-2-5所列。线型变量不具备电荷保持作用（trireg型除外），因此没有存储数据的能力，其逻辑值由驱动源提供和保持。各种线型变量在没有驱动源的情况下呈现高阻态（trireg保持在不定态）。

表2-2-5 常用线型变量及说明

| 线型变量 | 功能说明 |
|---|---|
| wire, tri | 标准连线 |
| wor, trior | 多重驱动时，具有线或特性的连线 |
| wand, triand | 多重驱动时，具有线与特性的连线 |
| tri1,tri0 | 上拉电阻、下拉电阻 |
| supply1, supply0 | 电源线、地线 |
| trireg | 具有电荷保持特性的连线 |

wire型变量是常用的线型变量。

线型变量定义的格式为：

```
wire in1, in2;
wire [7:0] local_bus;
```

第一种格式中将in1，in2定义为1位的线型变量，第二种格式中将local_bus定义为8位宽的线型变量。线型变量主要通过assign语句赋值，2.3节中将进一步讲述其赋值方式。

**小提示：**

对于综合而言，wire型变量的取值可以为0，1，X,与Z。

**3.** 寄存器型变量（**Registers**）

寄存器型变量表示一个抽象的数据存储单元，它并不特指寄存器，而是所有具有存储能力的硬件电路的通称，如触发器、锁存器等。此外，寄存器型变量还包括测试文件中的激励信号。虽然这些激励信号并不是电路元件，仅是虚拟驱动源，但由于保持数值的特性，仍然属于寄存器变量。寄存器类型只能在always语句和initial语句中被赋值，并且它的值从一个赋值到另一个赋值被保存下来。寄存器型变量的缺省值是不定态X。

**寄存器型变量与线型变量的显著区别是寄存器型数据在接受下一次赋值之前，始终保持原值不变，而线型变量需要有持续的驱动。**

寄存器型变量的主要分类如表2-2-6所列，其中，integer、real和time主要用于纯数学的抽象描述，不对应任何具体的硬件电路。

表2-2-6 常用寄存器型变量及说明

| 寄存器型变量 | 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位带符号整数型变量 |
| time | 无符号时间变量 |

寄存器型变量是常用的寄存器型变量。

寄存器型变量的定义方式为：

```
reg in3, in4; //
reg [7:0] local_bus; //local_bus为8位宽的寄存器型变量
```

第一种格式中将in3,in4定义为1位的寄存器型变量，第二种格式中将local_bus定义为8位宽的寄存器型变量。寄存器型变量主要通过块语句赋值，2.3节中将进一步讲述其赋值方式。

**4.** 存储器（**Memories**）

设计中，经常有存储指令或存储数据等操作，因此，需要掌握对存储器的定义和描述方式。

存储器定义的格式为：

```
reg [wordsize-1:0] memory_name[memsize-1:0];
```

例如：

```
parameter wordsize=16, memsize=1024;
reg [wordsize-1:0] mem_ram [memsize-1:0];
```

定义了一个由1024个16位寄存器构成的存储器，即存储器的字长为16位，容量为1K。

> **小提示：**
> 存储器可以看作是寄存器组成的数组。

## 2.2.3系统任务与系统函数

前面介绍标识符时提到，标识符的第一个字符不能够是“$”，因为在Verilog HDL中，“$”专门用来代表系统命令(如系统任务和系统函数)，本节将具体介绍常用的系统命令和系统函数的功能。

Verilog HDL共提供了10类，80余种系统任务与系统功能，下面主要介绍在设计和仿真过程中常用的系统任务与系统函数。

**1.** 系统任务**$display**与**$write**

系统任务$display与$write属显示类系统任务（Display system tasks），主要用于仿真过程中，将一些基本信息或仿真的结果按照需要的格式输出。

$display 和$write调用的格式为：

**$display （“格式控制字符串”，输出变量名表项）；**
**$write （“格式控制字符串”，输出变量名表项）；**

例2-2-4 系统任务调用

```
module disp;
reg [31:0] rval;
initial
begin
rval = 101;
display("rval = %h hex %d decimal",rval,rval);
$display("rval = %o octal\nrval = %b bin",rval,rval); // line 7
$display("rval has %c ascii character value",rval);
$display("current scope is %m");
$display("%s is ascii value for 101\n ",101);
$write("rval = %h hex %d decimal",rval,rval);
$write("rval = %o octal\nrval = %b bin",rval,rval);
$write("rval has %c ascii character value\n",rval);
$write("current scope is %m");
$write("%s is ascii value for 101\n",101);
end
endmodule
```

显示结果为：

```
rval = 00000065 hex                101 decimal
rval = 00000000145 octal
rval = 00000000000000000000000001100101 bin
rval has e ascii character value
current scope is disp
e is ascii value for 101
rval = 00000065 hex                101 decimalrval = 00000000145 octal
rval = 00000000000000000000000001100101 binrval has e ascii character value
current scope is disp e is ascii value for 101
```

**小提示：**
$display与$write唯一的区别是：$display在输出结束后会自动换行，而$write则只有在加入相应的换行符“\n”时才会产生换行。

通过对例2-2-4中两组显示结果的分析，可以了解到调用系统任务$display和$write时，需要注意：

（1）格式控制字符串的内容包括两部分：一部分为与输出变量在输出时需要一并显示的普通字符，如例2-2-4第7行中的“rval =”；另一部分为对输出的格式进行格式控制的格式说明符，如第7行中的“%o”“\n”。“\n”主要用于换行，“%o”是格式说明符，格式说明符以“%”开头，后面是控制字符，将输出的数据转换成指定的格式输出，具体如表2-2-7所列。

（2）输出变量名表项指要输出的变量名。如果有多个变量需要输出，各个变量名之间可以用逗号隔开。

（3）在输出变量表项缺省时，将直接输出引号中的字符串。这些字符串不仅可以是普通的

字符串，而且可以是字符串变量。

表2-2-7 格式说明符定义

| 格式说明符 | 输出格式 |
|---|---|
| %h或%H | 以十六进制数的形式输出 |
| %d或%D | 以十进制数的形式输出 |
| %o或%O | 以八进制数的形式输出 |
| %b或%B | 以二进制数的形式输出 |
| %c或%C | 以ASCII码字符的形式输出 |
| %s或%S | 以字符串的形式输出 |
| %v或%V | 输出线型数据的驱动强度 |
| %m或%M | 输出模块的名称 |

**小提示：**
这些格式很多与C语言的printf格式相同。

**2.** 系统任务**$monitor**

系统任务$monitor也属于显示类系统任务，同样用于仿真过程中，基本信息或仿真的结果的输出。

$monitor调用的格式为：

**$monitor （“格式控制字符串”，输出变量名表项）；**

$monitor具有监控功能，当系统任务被调用后，就相当于启动了一个后台进程，随时对敏感变量进行监控，如果发现其中的任意一个变量发生变化，整个参数列表中变量或表达式的值都将输出显示。

**3.** 系统任务**$readmem**

系统任务$readmem属文本读写类系统任务（File input-output system tasks），用于从文本文件中读取数据到存储器中。$readmem可以在仿真的任何时刻被执行。

系统任务调用的格式为：

**$readmemb(“<数据文件名称>”，<存储器名称>);**
**$readmemb(“<数据文件名称>”，<存储器名称>，<起始地址>);**
**$readmemb(“<数据文件名称>”，<存储器名称>，<起始地址>，<结束地址>);**
**$readmemh(“<数据文件名称>”，<存储器名称>);**
**$readmemh(“<数据文件名称>”，<存储器名称>，<起始地址>);**
**$readmemh(“<数据文件名称>”，<存储器名称>，<起始地址>，<结束地址>);**

系统任务$readmem中，被读取的数据文件内容只能够包含空白符、注释行、二进制或十六进制的数字，同样也可以存在不定态x、高阻态z和下划线_。其中，数字不能够包含位宽和格式说明。调用$readmemb时，每个数字必须是二进制，$readmemh中必须是十六进制数字。

此外，数据文件中地址（在本节专指存储器中的地址指针）的表示格式为，“@”后面加上十六进制数字。同一个数据文件中可以出现多个地址。当系统任务遇到一个地址时，立刻将该地址后面的数据存放到存储器中相应的地址单元中。

例2-2-5 8位MCU测试代码以及数据文件的部分内容

```
// RAM的定义
Reg       [7:0]      ram[0:65535];                                    // RAM
Wire      [7:0]      data_in;                                         // 输入数据
Reg       [7:0]      data_out;                                        // 输出数据
// 输出数据的设计
always @(posedge CLK)
begin
if (RW)       data_out = #4 ram[ADDRESS];                  // 输出的数据
else              data_out = #4 8'hzz;
end
// 输入数据的设计
always @(negedge CLK)
begin
if(~RW)          #4 ram[ADDRESS] = data_in;
end
// 测试向量库的调入
initial
begin
$readmemh("c:/SimFile/00_test_all.txt",ram);
// $readmemh("c:/SimFile/01_test_lda.txt",ram);
// $readmemh("c:/SimFile/02_test_ldxldy.txt",ram);
…
end
```

数据文件“c:/SimFile/00_test_all.txt”的内容为：

```
@0000
23
23
23
45
65
67
…
@fff0
23
34
54
67
```

此例中调用的系统任务是$readmemh，因此数据文件中采用十六进制数据。此外，RAM的存放过程是按照定义中指定的顺序变化的，即从0开始到65535结束。

**4.** 系统任务**$stop**与**$finish**

$stop与$finish属仿真控制类系统任务（Simulation control system tasks），主要用于仿真过程中对仿真器的控制作用。

$stop具有暂停功能，这时，设计人员可以输入相应的命令，实现人机对话。通常执行完$stop后，会出现系统提示，

如：“Break at time.v line 13”。

$finish的作用是结束仿真进程，输出信息包括系统结束时间、模块名称等，如：

```
** Note: $finish : time.v(13)
Time: 300 ns Iteration: 0 Instance: /test
```

**5.** 系统函数**$time**

$time属于仿真时间类系统函数（Simulation time system functions），通常与显示类系统任务配合，以64位整数的形式显示仿真过程中某一时刻的时间。

例2-2-6 $time与$monitor应用示例

```
`timescale 10 ns / 1 ns
module test;
reg set;
parameter delay = 3;

initial
begin
$monitor($time,"set=",set);
#delay set = 0;
#delay set = 1;
end
endmodule
```

显示结果为：

```
0 set=x
3 set=0
6 set=1
```

**小提示：**

很多初学者会对上面介绍的以上这些系统任务不理解，其实系统任务的主要用途是在设计仿真过程中用于可视化调试，在模块逻辑设计时，是不能使用这些不可综合的语句的。

## 2.2.4编译向导

Verilog HDL中编译向导的功能和C语言中编译预处理的功能非常接近，在编译时首先对这些编译向导进行“预处理”，然后保持其结果，将其与源代码一起进行编译。

编译向导的标志是在某些标识符前添加反引号“`”，在Verilog HDL中，完整的编译向导集合如下：

| | | |
|---|---|---|
| **`define** | **`timescale** | **`include** |
| **`celldefine** | **`default_nettype** | **`else** |
| **`endcelldefine** | **`endif** | **`ifdef** |
| **`nounconnected_drive** | **`resetall** | **`unconnected_drive** |
| **`undef** | | |

这里主要介绍常用的编译向导，其他编译向导的使用可以参考IEEE1364-1995标准。

**1.** 宏定义**`define**

宏定义`define的作用是用于文本定义，和C语言的#define类似，即在编译时通知编译器，用宏定义中的文本直接替换代码中出现的宏名。

宏定义的格式为：

**`define <宏名> <宏定义的文本内容>**

宏定义语句可以用于模块的任意位置，通常写在模块的外面，有效范围是从宏定义开始到源代码描述结束。此外，建议采用大写字母表示宏名，以便于与变量名相区别。

每条宏定义语句只可以定义一个宏替换，且结束时没有分号；否则，分号也将作为宏定义内容。在调用宏定义时，也需要用撇号“`”作开头，后面跟随宏定义的宏名。

通过下面的示例可知，采用宏定义能够提高代码的可读性和可移植性。

```
`define WORDSIZE 8
reg [1:`WORDSIZE] data;

//define a nand with variable delay
`define VAR_NAND(dly) nand #dly
` VAR_NAND (2) g121 (q21, n10, n11);          // delay is 2
` VAR_NAND (5) g122 (q22, n10, n11);          // delay is 5
```

**组成宏定义的字符串不能够被以下标识符分隔开，如注释行、数字、字符串、确认符、关键词、双目和三目字符运算符，否则，该宏定义是非法的。**

例如：

```
`define first_half “start of string
$display (`first_half end of string”);
```

上面的例子就是由于被引号隔开，使得宏定义非法。

**2.** 仿真时间尺度**`timescale**

仿真时间尺度是指对仿真器的时间单位及对时间计算的精度进行定义。

格式为：

**`timescale <时间单位> /<时间精度>**

时间单位和时间精度都由整数和计时单位组成的。合法的整数有1，10，100；合法的计时单位为s、ms、us、ns、ps和fs。

在仿真时间尺度中，时间单位用来定义模块内部仿真时间和延迟时间的基准单位；时间精度用来声明该模块仿真时间的精确程度。如：`timescale 1 ns/100 ps指以1ns作为仿真的时间单位，以100ps的计算精度对仿真过程中涉及到的延时量进行计算。

时间精度和时间单位的差别最好不要太大。因为在仿真过程中，仿真时间是以时间精度累计的，两者差异越大，仿真花费的时间就越长。另外，时间精度值至少要和时间单位一样精确，时间精度值不能大于时间单位值。如果一个设计中存在多个`timescale，则采用最小的时间单位。

**小提示：**
如果不指定`timescale <时间单位> /<时间精度> ，则系统默认执行 `timescale 1ns/1ns 编译指令。

**3.** 文件包含**`include**

编译向导中，文件包含`include的作用是在文件编译过程中，将语句中指定的源代码全部包含到另外一个文件中。格式如下：

**`include "文件名"**

如：`include "global.v"

`include "../../library/mux.v"

其中，文件名中可以指定包含文件的路径，既可以是相对路径名，也可以是完整的路径名。每条文件包含语句只能够用于一个文件的包含，但是，包含文件允许嵌套包含，即包含的文件中允许再去包含另外一个文件。

**小提示：**
与C语言的include作用基本相同。

# 2.3 Verilog HDL基本语句

## 2.3.1过程语句（Structured procedures）

Verilog HDL中，所有的描述都是通过下面四种结构中的一种实现的：

(1) initial语句
(2) always语句
(3) task任务
(4) function函数

在一个模块内部可以有任意多个initial语句和always语句，两者都是从仿真的起始时刻开始执行的，**但是initial语句后面的块语句只执行一次，而always语句则循环地重复执行后面的块语句，直到仿真结束。**

task任务和function函数可以在模块内部从一处或多处被调用，具体使用方法将在后面的章节中介绍。

**1. initial语句**

initial 语句的格式为：

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

在前面的源代码里面已经多次出现 initial 语句。下面将按照 initial 块语句的形式分别介绍。

(1) 无时延控制的 initial 语句：initial 语句从 0 时刻开始执行，在下面的例子中，寄存器变量 a 在 0 时刻被赋值为 4。

```
reg a;
…
initial
a = 4;
…
```

(2) 带时延控制的initial语句：initial语句从0时刻开始执行，寄存器变量b在时刻5时被赋值为3。

```
reg b;
…
initial
#5 b = 3;
…
```

(3) 带顺序过程块(begin-end)的initial语句：initial语句从0时刻开始执行，寄存器变量start在时刻0时被赋值为0，又在时刻10时被赋值为1。

```
reg start;
…
initial
begin
start=0;
#10 start=1;
end
```

**2. always**语句

always语句在仿真过程中是不断重复执行的，描述格式为：

**always <时序表达式> <进程语句>;**

其中最常用的描述格式：

**always @（敏感信号表达式）**

**begin**

**…..**

**end**

在前面的源代码里面也同样多次出现过always语句，以下是一些基本示例：

(1) 不带时序控制的always语句：由于没有时延控制，而always语句是重复执行的，因此下面的always语句将在0时刻无限循环。

```
always clock = ~ clock;
```

(2) 带时延控制的always语句：产生一个50M的时钟。

```
always #100 clock = ~ clock;
```

(3) 带事件控制的always语句：在时钟上升沿，对数据赋值。

```
always @ (posedge clock )
data = data_in;
```

**小提示：**

对于下降沿触发,应为always@(negedge clk);时钟的上升沿与下降沿是最常用的敏感表达

（4）多个信号控制的always语句：在敏感信号发生状态变化的时候执行语句

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

```
begin
   a<=b;
   c<=d;
end
```

**小提示：**
对于多个敏感信号之间可以用or隔开，也可以使用逗号”,”隔开。

### 2.3.2 赋值语句Assignments

赋值语句是Verilog HDL中对线型和寄存器型变量赋值的主要方式，根据赋值对象的不同分为连续赋值语句和过程赋值语句，两者的主要区别是：

(1) 赋值对象不同：**连续赋值语句用于对线型变量的赋值；过程赋值语句完成对寄存器变量的赋值**，具体内容如表2-3-1所列。

表2-3-1 赋值语句的赋值对象

| 赋值语句 | 赋值对象 |
|---|---|
| 连续赋值语句 | 线型变量（标量或矢量）<br>线型变量（矢量）中的某一位<br>线型变量（矢量）中的某几位<br>上述三种的任意组合 |
| 过程赋值语句 | 寄存器型变量（标量或矢量）<br>寄存器变量（矢量）中的某位<br>寄存器变量（矢量）中的某几位存储器<br>上述四种的任意组合 |

(2) 赋值过程实现方式不同：线型变量一旦被连续赋值语句赋值后，赋值语句右端表达式中的信号有任何变化，都将实时地反映到左端的线型变量中；过程赋值语句只有在语句被执行到时，赋值过程才能够进行一次，而且赋值过程的具体执行时间还受到各种因素的影响。

(3) 语句出现的位置不同：连续赋值语句不能够出现在任何一个过程块中；过程赋值语句只能够出现在过程块中。

(4) 语句结构不同：连续赋值语句以关键词assign为先导；过程赋值语句不需要任何先导的关键词，但是，语句的赋值分为阻塞型和非阻塞型。

下面将分别介绍两种赋值语句的具体应用。

**1.** 连续赋值语句**Continuous assignments**

assign为连续赋值语句，用于对wire型变量进行赋值。其基本的描述语法为：

**assign ＃[delay] <线型变量> = <表达式>；**

例2-3-1 四位加法器的verilog的描述

```
module adder (sum_out, carry_out, carry_in, ina, inb);
output [3:0] sum_out;
output carry_out;
input [3:0] ina, inb;
input carry_in;
wire carry_out, carry_in;
```

```
wire [3:0] sum_out, ina, inb;
assign {carry_out, sum_out} = ina + inb + carry_in;
endmodule
```

例2-3-1是一个四位加法器的描述，其中，通过连续赋值语句对进位位和运算结果统一赋值，这种组合赋值过程在设计中会经常涉及到。

例2-3-2 使用带门延的assign语句

```
`timescale 1ns/1ns
module MagnitudeComparator(A,B,AgtB,AeqB,AltB);
  //parameters     first
  parameter     BUS = 8;
  parameter     EQ_DELAY = 5, LT_DELAY = 8, GT_DELAY = 8;

  input     [BUS-1:0]     A,B;
  output AgtB,AeqB,AltB ;

  wire     [BUS-1:0]     A,B;

  assign      #EQ_DELAY      AeqB = A == B;      //第13行
  assign      #GT_DELAY      AgtB = A > B;
  assign      #LT_DELAY      AltB = A < B;
endmodule
```

例2-3-3 过程赋值与连续赋值的比较

```
`timescale 1ns/1ns
module assignment_for_bit(in1,in2,out1,out2);

input [1:0] in1,in2;
output [1:0] out1,out2;

wire [1:0] out1; //定义输出信号类型
reg [1:0] out2;
wire [1:0] in1, in2; //定义输入信号类型

//     连续赋值语句部分
assign out1 =    in1 & in2;

//     过程赋值语句
always @( in1 or in2 )
out2= in1 & in2;
endmodule
```

例2-3-2和例2-3-3都含有通过连续赋值语句赋值的源代码，其中，例2-3-2的第13~15行带有延迟。

**小提示：**
大部分综合工具不支持延迟语句，因此，延迟主要用于仿真过程中。

仿真需要EDA工具的支撑才能够完成。目前，数字电路设计中前端采用的仿真工具主要有Cadence公司的Verilog-XL，NC-Verilog，Synopsys公司的VCS，Mentor公司的ModelSim，Aldec公司的Active-HDL等。图2-3-1是例2-3-2的仿真波形图（用Mentor公司的Modelsim仿真器得到），从中可以清晰的看到：

245ns时，B=8'h40,A=8'h08; 8ns后，即253ns时，AltB由0变为1。

295ns时，B=8'h00, A=8'h08; 8ns后，即303ns时，AltB由1变为0。

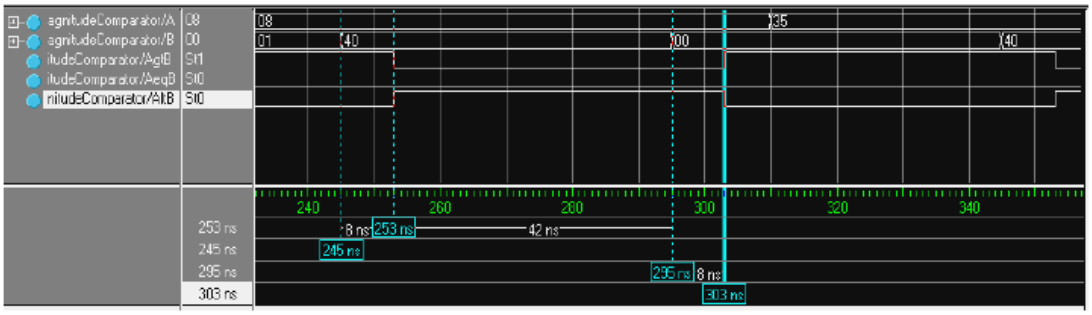



图 2-3-1 例 2-3-2 仿真波形图

例2-3-3 中第一部分是连续赋值语句部分。由于 out2 是寄存器型变量，因此第二部 采用过程赋值语句对 out2 赋值。 图 2-3-2 是例 2-3-3 综合后生成的逻辑电路图（用 Synplicity 公司的 Synplify 综合器得 ），从中可以看到，虽然 out2 是寄存器型变量，但是综合结果与线型的 out1 相同。因此，**寄存器型变量对应的硬件电路并不一定是寄存器。**

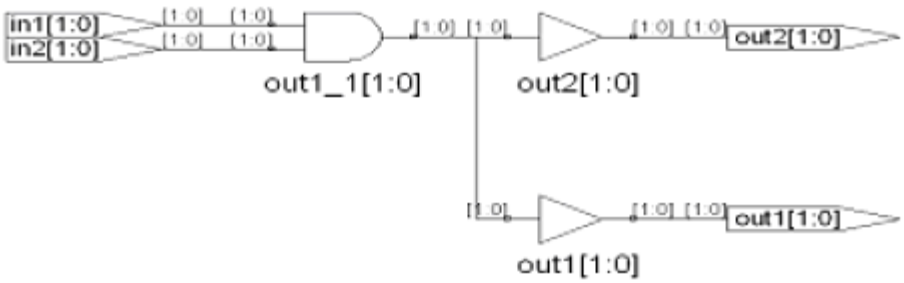



图2-3-2 例2-3-3综合的逻辑电路图

**2.** 过程赋值语句（**Procedural assignments**）

过程赋值语句用于对寄存器类变量赋值，没有任何先导的关键词，而且只能够在always语句或initial语句的过程块中赋值.其基本的描述语法为：

**<寄存器型变量> = <表达式>; <1>**

**或 <寄存器型变量> <= <表达式>; <2>**

过程赋值语句有两种赋值形式：阻塞型过程赋值（即描述方式<1>）和非阻塞性过程赋值（即描述方式<2>）。

例2-3-1中的第二部分就是采用阻塞型过程赋值方式的描述：

```
// 过程赋值语句
```

```
always @( in1 or in2 )
out2= in1 & in2;
```

**3**. 堵塞型赋值语句与非阻塞型赋值语句（**Blocking & nonblocking procedural assignment**）

按照过程赋值语句被有效执行的顺序，将过程赋值语句，读者可以根据这两种描述语句的特性，合理选择，以便得到理想的设计。

首先通过例2-3-4及其仿真波形图感性地了了解一下这两种赋值方式的定义及其区别。

例2-3-4

```
module block_nonblock (dataout_a,dataout_b,dataout_c,dataout_d,
                       data_in,clock);
input data_in,clock;
output dataout_a,dataout_b;
output dataout_c,dataout_d;
reg dataout_a,dataout_b;
reg dataout_c,dataout_d;

always @(posedge clock) //阻塞赋值
begin
dataout_a=data_in;
dataout_b= dataout _a;
end

always @(posedge clock) //非阻塞赋值
begin
dataout_c <= data_in;
dataout_d <= dataout_c;
end
endmodule
```

例 2-3-5 例2-3-4的测试程序

```
module test_blk_nonblk;

reg data_in,clock;
wire dataout_a,dataout_b;
wire dataout_c,dataout_d;

block_noublock blk_nonblk(dataout_a,dataout_b,dataout_c,dataout_d,
                          data_in,clock);

initial
begin
clock=0; data_in=0;
#40        data_in=1;
end
```

```
always    #50    clock = ~clock;

always    #100    data_in =~data_in;

initial
begin
#1000    $stop;
#20      $finish;
end

endmodule
```

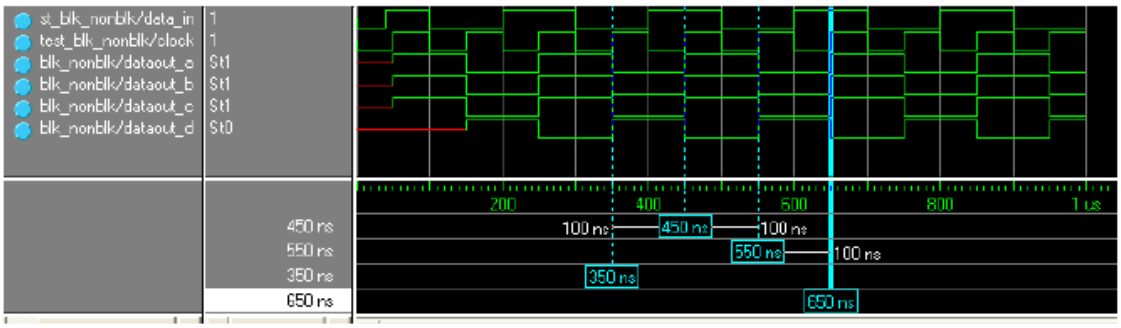



图2-3-3 例2-3-4的仿真波形图

例2-3-4中，分别通过阻塞型赋值语句对dataout_a，dataout_b赋值；非阻塞型赋值 语句对 dataout_c，dataout_d 赋值。从图 3-3-3 中可以看到，dataout_a，dataout_b 和 dataout_c 的波形图完全一致,但是dataput_d比 dataout_c 晚了一个时钟周期。

**由此可以得到阻塞与非阻塞型赋值语句的基本区别是，阻塞型赋值语句的执行受到前后顺序的影响，只有在第一条语句执行完之后才可以执行第二条语句，而在非阻塞型赋值语句中，则是在某一规定时刻同时完成，不受先后顺序的影响.从某个角度讲，非阻塞型赋值语句的执行顺序与并行块的执行十分相象，这些在后面还会讲到。**

**小提示：**
阻塞赋值按顺序执行，非阻塞赋值，块结束后并行执行。

### 2.3.3 块语句（Block statements）

语句块用来将两条或多条语句组合在一起,使其在格式上更象一条语句。块语句有两种，一种是begin-end语句，通常用来标识按照给定顺序执行的串行块（Sequential block）；一种是fork-join语句，用来标识并行执行的并行块(Parallel block)。

1.串行块**(begin-end)**

串行块具有如下特点：

**(1) 串行块中的每条语句都是依据块中的排列次序顺序执行。**

**(2) 串行块中每条语句的延时都是相对于前一条语句执行结束的相对时间。**

**(3) 串行块的起始执行时间是块中第一条语句开始执行的时间，结束时间是最后一条语句执行结束的时间。**

在例2-3-5中，就使用了串行块定义仿真的延迟时间。在并行块中还将给出其他示例。

```
// 例2-3-5中节选
initial
begin
clock=0; data_in=0;
#40       data_in=1;
end
```

2. 并行块（fork-join）

并行块具有如下特点：

**(1) 并行块中的每条语句都是同时并行执行的，与排列次序无关。**

**(2) 并行块中每条语句的延时都是相对于整个并行块开始执行的绝对时间。**

**(3) 并行块的起始执行时间是流程控制转入并行块的时间，结束时间是并行块 中按执行时间排序，最后执行的那条语句结束的时间。**

例 2-3-6中分别采用串行块和并行块描述了两个寄存器型变量，分别加以延时控制，从图2-3-4 中可以看到，两个寄存器型变量最终得到了相同的结果。

例 2-3-6 串行块与并行块

```
module    sequential_parallel;

parameter d = 50;                  // d declared as a parameter and
reg [7:0] seq, para              // seq and para are declared as 8-bit registers

initial
begin
#d seq=8'h35;
#dseq = 8'hE2;
#d seq = 8'h00;
#d seq = 8'hF7;
end

initial
fork
#50    para+8’h35;
#100 para+8’hE2;
#150 para+8’h00;
#200 para+8’hF7
Join

Initial
Begin
#400    $stop
#30      $finish;
end
```

```
endmodule
```

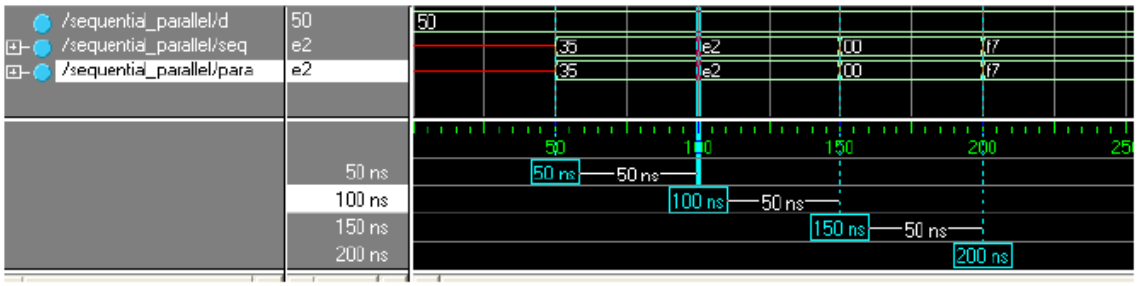



图2-3-4 例2-3-6 的仿真波形图

### 2.3.4 条件语句（Conditional statement）

**1. if-else** 语句

**if-else** 语句是用来判断所给的条件是否满足，根据判定的结果（真或假）决定执行给出的两种操作之一。Verilog HDL语言共提供了3种形式的if-else语句。

**（1）if（表达式） 块语句1;**

**（2）if（表达式） 块语句1;**
**else 块语句2;**

**（3）if （表达式1） 块语句1;**
**else if（表达式2） 块语句2;**
**else if（表达式3） 块语句3;**
**…**
**else if （表达式n）块语句n ;**
**else 块语句n+1;**

第一种情况下如果条件表达式成立（即表达式的值为1时），执行后面的块语句1；当条件表达式不成立（即表达式的值为0，x，z）时，停止执行块语句1，此时会形成锁存器，保存块语句1的执行结果。

第二种情况下，如果条件表达式不成立，执行块语句2，这样，在硬件电路上通常会形成多路选择器。

第三种情况下，依次检查表达式是否成立，根据表达式的值判断执行的块语句。由于if-else的嵌套，需要注意if与else的配对关系，以免无法实现设计要求。这一部分内容在2.7节设计实例中将继续介绍。

例2-3-7是一个考试成绩统计器的设计，其中使用了大量的if-else嵌套。例2-3-8是其仿真代码，最后是显示的输出结果。

> **小提示：**
> 如果执行语句只有一句，可以不用块语句。但是推荐全用块语句（一般是begin-end）括起来，看起来更清晰。

例 2-3-7 成绩统计器的verilog描述

```
module    score _grade(reset,Sum,Grade, Total_A,Total_B,Total_C,Total_D);

input reset;
input[3:0]Sum;
output[3:0]Grade;
```

```
output[3:0]Total_A,Total_B,Total_C,Total_D;

reg[3:0]Grade;
reg [3:0] Total_A,Total_B,Total_C,Total_D;
wire [7:0] Sum;

always@(Sum or reset)
if(~reset)
begin
   Grade=0;
   Total_D=0;
   Total_C=0;Total_B=0;Total_A=0;
   end
else
   begin
   if（Sum<8’d60）
      begin
         Grade=4’bd;
         Total_D=Total_D+1;
      end
   else if (Sum<8'd75)
      begin
         Grade=4’hc;
         Total_C=Total_C+1;
      end
   else if(Sum<8’d85)
      begin
         Grade=4’hb;
         Total_B=Total_B+1;
      end
     else
      begin
         Grade=4’ha;
         Total_A=total_A+1;
      end
end
enledmodu
```

例3-3-8 测试程序

```
 module test_score;

          reg      reset;
reg      [7:0]Sum;
wire     [3:0]Grade;
```

```
wire     [3:0]Total_A,Total-B，Total_C,Total_D;

score_grade score (reset,Sum,Grade,Total_A,Total_B,Total_C,Total_D);

initial
begin
reset=0;
Sum=8'd50;
#10 reset=1;
#100 Sum=8'd100;
#100 Sum=8'd80;
#100 Sum=8'd90;
#100 Sum=8'd60;
#100 Sum=8'd100;
#100 Sum=8'd80;
#100 Sum=8'd100;
#100 Sum=8’d90;
#100 Sum=8’d60;
#100 Sum=8’d100;
#100 Sum=8’d80;

$display(Total_A=”,Total_A);
$display("Total_B=",Total_B);
$display("Total_C=",Total_C);
$display("Total_D=",Total_D);

end
endmodule
```

最终显示结果为：

```
Total_A=6
Total_B=3
Total_C=2
Total_D=1
```

**2. case**语句

case语句构成了一个多路条件分支的结构，多用于多条件译码电路的描述中，如译码器、数据选择器、状态机及微处理器的指令译码等。Verilog HDL语言共提供了3种形式的case语句。

**（1） case（敏感表达式）**

```
    值1：   块语句1；
    值2：   块语句2；
   …
    值n:    块语句n;
    default: 块语句n+1;
```

```
    endcase
```

（2） **casez（敏感表达式）**

```
    值1：   块语句1；
    值2：   块语句2；
   …
    值n:     块语句n;
    default: 块语句n+1;
endcase
```

（3） **casex（敏感表达式）**

```
    值1：   块语句1；
    值2：   块语句2；
   …
    值n:     块语句n;
    default: 块语句n+1;
endcase
```

这三种语句的描述方式唯一的区别就是对敏感表达式的判断，**其中，第一种要求敏感表达式的值与给定的值1、值2……或值n中的一个全等时，执行后面相应的块语句；如果均不等时，执行default语句。第二种（casez）则认为，如果给定的值中有某一位（或某几位）是高阻态（z），则认为该位为“真”，敏感表达式与其比较时不予判断，只需比较其他位。第三种（casex）则扩充为，如果给定的值中有某一位（或某几位）是高阻态（z）或不定态（x），同样认为其为“真”，不予判断。**例2-3-9是采用casez和casex编写的代码，可以感性地了解cases和casex的应用。

例2-3-9 case语句的使用示例

```
// example for casez
casez (encoder)
   4'b1???: high_lvl=3;
   4'b01??: high_lvl=2;
   4'b001?: high_lvl=1;
   4'b0001: high_lvl=0;
   default: high_lvl=0;
endcase

// example for casex
casex (encoder)
   4'b1xxx: high_lvl=3;
   4'b01xx: high_lvl=2;
   4'b001x: high_lvl=1;
   4'b0001: high_lvl=0;
   default: high_lvl=0;
endcase
```

case语句与if-else语句的功能十分接近，可以全部转化为if-else语句描述。当控制条件集

中在某个敏感表达式的变化上时，同样也可以将if-else语句改写成case语句。只是，if-else语句可以实现优先权的描述，而case无法实现。具体内容及示例将在2.7节具体介绍。

> **小提示：**
> 条件语句包含if和case语句，他们都是顺序语句，应该放在always块中。

### 2.3.5 循环语句

Verilog HDL中存在4种类型的循环语句，可以控制语句的执行次数。这四种语句分别是for语句、repeat语句、while语句和forever语句。

**1. for**语句

与C语言完全相同，for语句的描述格式为：

**for ( 循环变量赋初值；循环结束条件；循环变量增值 ) 块语句；**

即在第一次循环开始前，对循环变量赋初值；循环开始后，判断初值是否符合循环结束条件，如果不符合，执行块语句，然后给循环变量增值；再次判断是否符合循环结束条件，如果符合循环结束条件，循环过程终止。

例2-3-10 采用for语句描述的七人投票器

```
module vote(pass,vote);
input [6:0]vote;
output pass;
reg pass;
reg [2:0] sum;
integer i;

always @(vote)
begin
   sum=0;
   for (i=0; i<=7; i=i+1)
   if (vote[i]) sum=sum+1;
   if (sum[2]) pass=1;
   else pass=0;
end
endmodule
```

**2. repeat**语句

repeat语句可以连续执行一条语句若干次，描述格式为：

**repeat (循环次数表达式)**
**块语句；**

在例2-3-11中，经过重复8次，实现了16位数据前8位与后8位的数据交换。

例2-3-11 repeat语句的应用

```
if(rotate==1)
repeat(8)
   begin
      temp=data[15];
      data = {data<<1,temp};
```

```
    end
```

例2-3-12 采用repeat语句实现两个8位二进制数乘法

```
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 )
      begin
         if (temp_b[1]) outcome = outcome + temp_a ;
         temp_a = temp_a <<1 ;
         temp_b = temp_b>>1 ;
      end
end
endmodule
```

**小提示：**
有些EDA工具不支持repeat语句，如MAX+PLUS II，但是QuartusII与ModelSim支持。

**3. while**语句

while语句是不停地执行某一条语句，直至循环条件不满足时退出。描述格式为：

**while（循环执行条件表达式）**

**块语句；**

while语句在执行时，首先判断循环执行表达式是否为真，如果为真，执行后面的块语句，然后再返回判断循环执行条件表达式是否为真，依据判断结果确定是否需要继续执行。

while语句与for语句十分类似，都需要判断循环执行条件是否为真，以此确定是否需要继续执行块语句。

例2-3-13 通过调用while语句实现从0到100的计数过程

```
initial
begin
   count = 0;
   while (count < 101)
      begin
      $display("Count=%d",count);
      count = count + 1;
      end
end
```

**4. forever**语句

forever语句可以无条件地连续执行语句，多用在“initial”块中，生成周期性输入波形，通常为不可综合语句。描述格式为：

**forever 块语句；**

例2-3-14 通过forever语句生成一个50M的时钟信号

```
initial
begin
    Clock=0;
    forever #100 Clock =~Clock;
end
```

### 2.3.6 任务与函数

Verilog HDL分模块对系统加以描述，但有时这种划分并不一定方便或显得勉为其难。因此，Verilog HDL还提供了任务和函数的描述方法。通常在描述设计的开始阶段，设计者更多关注总体功能的实现，之后再分阶段对各个模块的局部进行细化实现，任务和函数对这种设计思路的实现有很大的帮助。

任务和函数，是在模块内部将一些重复描述或功能比较单一的部分，作为一个相对独立地进行描述，在设计中可以多次调用。

**1. task**任务

任务可以在源代码中的不同位置执行共同的代码段，这些代码段已经用任务定义编写成任务，因此，能够从源代码的不同位置调用任务。

任务的定义与引用都在一个模块内部完成，任务内部可以包含时序控制，即时延控制，并且任务也能调用任何任务（包括其本身）和函数。

定义格式为：

```
task <任务名>;
    <端口及数据类型定义语句>
    <语句1>
    <语句2>
    …
    <语句n>
endtask
```

调用格式为：

**<任务名>（端口1，端口2, ……）；**

例2-3-15 交通灯的时序控制代码，任务的作用是用来控制等的颜色与点亮时间

```
module traffic_lights;
reg clock, red, amber, green;
parameter on = 1, off = 0, red_tics = 350, amber_tics = 30, green_tics = 200;

// initialize colors.
initial
begin
red = off;
amber = off;
```

```
    green = off;
    end

    always
    begin // sequence to control the lights.
    red = on; // turn red light on
    light(red, red_tics); // and wait.
    green = on; // turn green light on
    light(green, green_tics); // and wait.
    amber = on; // turn amber light on
    light(amber, amber_tics); // and wait.
    end


    // task to wait for “tics” positive edge clocks
    // before turning “color” light off.
    task light;
    output color;
    input [31:0] tics;
    begin
    repeat (tics) @ (posedge clock);
    color = off; // turn light off.
    end
    endtask

    always
    begin // waveform for the clock.
    #100 clock = 0;
    #100 clock = 1;
    end
endmodule // traffic_lights.
```

**小提示：**

使用任务时，要注意以下几点：

1. 定义任务与调用任务必须在同一个模块内，任务调用语句应该在always块或者task-endtask块中。
2. 定义任务时，没有端口名列表，但要进行端口与数据类型的声明。
3. 调用任务时，与调用模块一样，要列出端口名列表，但是顺序要与定义中的排序完全一致。
4. 任务中可以调用其他的任务或者函数，且调用的个数不受限制。

**2. function函数**

函数与task任务一样，也可以在模块中的不同位置执行同一段代码；不同之处是函数只能返回一个值，它不能包含任何时间控制语句。函数可以调用其它函数，但是不能调用任务。

此外，函数必须至少带有一个输入端口，在函数中允许没有输出或输入输出说明。

函数的定义格式为：

```
function <位宽说明>函数名;
    <输入端口与类型说明>
    <局部变量说明>
    begin
    <语句1>
    <语句2>
    ……
    <语句n>
    end
endfunction
```

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。其调用格式为：

**<函数名>(<表达式1>，<表达式2>，……)**

> **小提示：**
> 函数的定义蕴含声明了一个与函数名同名的，函数的内部寄存器，并作为函数的返回值传出函数。

例2-3-16 通过函数的调用实现阶乘的运算过程

```
module tryfact;
function [31:0] factorial; // define the function
input [3:0] operand;
reg [3:0] i;
begin
   factorial = 1;
   for (i = 2; i <= operand; i = i + 1)
   factorial = i * factorial;
end
endfunction

integer result, n; // test the function
initial
begin
   for (n = 0; n <= 7; n = n+1)
      begin
      result = factorial(n);
      $display("%0d factorial=%0d", n, result);
      end
end
endmodule // tryfact
```

显示结果为：

```
0 factorial=1
```

```
1 factorial=1
2 factorial=2
3 factorial=6
4 factorial=24
5 factorial=120
6 factorial=720
7 factorial=504
```

**小提示：**

任务与函数的区别

1．函数需要在一个仿真时间单位内完成；而任务定义中可以包含任意类型的定时控制部分及wait语句等。

2．函数不能调用任务，而任务可以调用任何任务和函数。

3．函数只允许有输入变量且至少有一个，不能够有输出端口和输入输出端口；任务可以没有任何端口，也可以包括各种类型的端口。

4．函数通过函数名返回一个值；任务则不需要。

## 2.4 Verilog HDL 建模概述

在HDL的建模中，主要有结构化描述方式、数据流描述方式和行为描述方式，下面分别举例说明三者之间的区别。

**小提示：**

本节内容是前几节内容的综合，读者要着重掌握模块例化与行为描述方式。

### 2.4.1 结构化描述方式

结构化的建模方式就是通过对电路结构的描述来建模，即通过对器件的调用（HDL概念称为例化），并使用线网来连接各器件的描述方式。这里的器件包括Verilog HDL的内置门如与门and，异或门xor等，也可以是用户的一个设计。结构化的描述方式反映了一个设计的层次结构。

例2-5-1：一位全加器

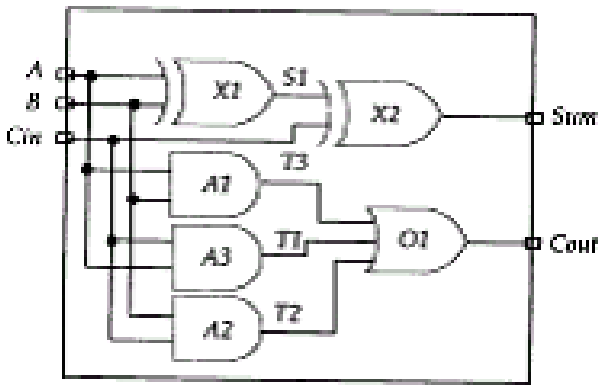



图2-4-1 一位全加器的结构图

代码：

```
module FA_struct (A, B, Cin, Sum, Count);
input A;
input B;
input Cin;
output Sum;
output Count;
wire S1, T1, T2, T3;
// -- statements -- //
xor x1 (S1, A, B);
xor x2 (Sum, S1, Cin);
and A1 (T3, A, B );
and A2 (T2, B, Cin);
and A3 (T1, A, Cin);
or O1 (Cout, T1, T2, T3 );
endmodule
```

该实例显示了一个全加器由两个异或门、三个与门、一个或门构成。S1、T1、T2、T3 则是门与门之间的连线。代码显示了用纯结构的建模方式，其中xor 、and、or 是Verilog HDL 内置的门器件。以 xor x1 (S1, A, B) 该例化语句为例：xor 表明调用一个内置的异或门，器件名称xor ，代码实例化名x1（类似原理图输入方式）。括号内的S1，A，B 表明该器件管脚的实际连接线（信号）的名称，其中 A、B是输入，S1是
输出。其他同。

例2-4-2：两位的全加器

两位的全加器可通过调用两个一位的全加器来实现。该设计的设计层次示意图和结构图如下：

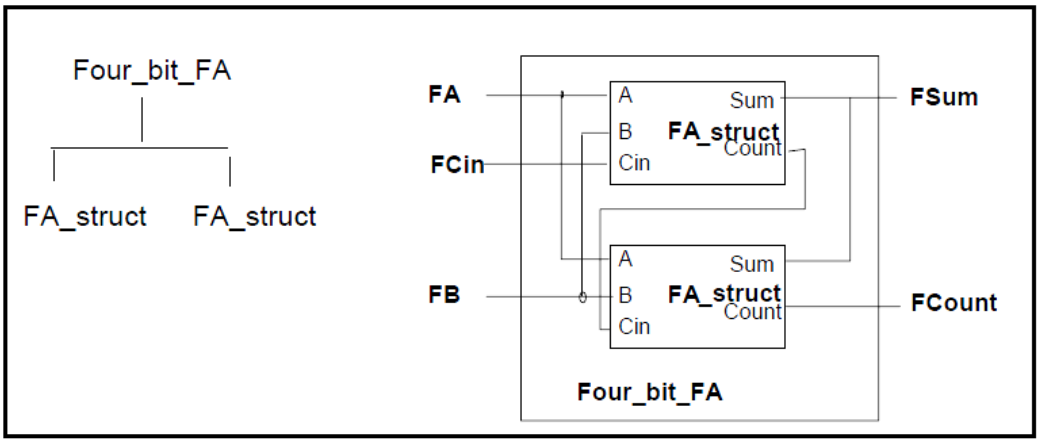



图2-4-2 两位全加器的结构示意图

代码：

```
module Four_bit_FA (FA, FB, FCin, FSum, FCout ) ;
parameter SIZE = 2;
input [SIZE:1] FA;
input [SIZE:1] FB;
input FCin;
output [SIZE:1] FSum;
```

```
output FCout;
wire FTemp;
FA_struct FA1(.A (FA[1]),.B (FB[1]),.Cin (FCin) ,.Sum (FSum[1]),.Cout
        (Ftemp));
FA_struct FA2(.A (FA[2]),.B (FB[2]),.Cin (FTemp) ,.Sum
        (FSum[2]),.Cout (FCount ));
endmodule
```

该实例用结构化建模方式进行一个两位的全加器的设计，顶层模块Four_bit_FA 调用了两个一位的全加器 FA_struct 。在这里，以前的设计模块FA_struct 对顶层而言是一个现成的器件，顶层模块只要进行例化就可以了。注意这里的例化中，端口映射（管脚的连线）采用名字关联，如 .A（FA[2]） ，其中.A 表示调用器件的管脚A，括号中的信号表示接到该管脚A的电路中的具体信号。wire 保留字表明信号Ftemp 是属线网类型（下面有具体描述）。另外，在设计中，尽量考虑参数化的问题。器件的端口映射必须采用名字关联。

**小提示：**

模块例化语句例化的格式为：

设计模块名[例化电路名](端口列表)；

其中设计模块名是用户设计的电路模块名，例化电路名是用户为了系统设计定义的标书费（为可选项），相当于电路板上为插入设计元件的插座；端口列表用于描述模块元件上引脚与插座的对应关系。

端口列表的描述方法有两种：

1. 位置关联法 **（.设计模块端口名，.设计模块端口名,…… .设计模块端口名）**但是顺序要对应。
2. 名称关联法 **（.设计模块端口名（插座引脚名），.设计模块端口名（插座引脚名），….）**顺序可以不对应，上例使用的就是名称关联法。

### 2.4.2 数据流描述方式

数据流的建模方式就是通过对数据流在设计中的具体行为的描述的来建模。最基本的机制就是用连续赋值语句。在连续赋值语句中，某个值被赋给某个线网变量（信号），语法如下：

**assign [delay] net_name = expression;**

如：

```
assign #2 A = B;
```

在数据流描述方式中，还必须借助于HDL提供的一些运算符，如按位逻辑运算符：逻辑与（&），逻辑或（|）等。

以上面的全加器为例，可用如下的建模方式：

```
`timescale 1ns/100ps
module FA_flow(A,B,Cin,Sum,Count)
input A,B,Cin;
output Sum, Count;
wire S1,T1,T2,T3;
assign # 2 S1 = A ^ B;
assign # 2 Sum = S1 ^ Cin;
```

```
assign #2 T3 = A & B;
assign #2 T1 = A & Cin;
assign #2 T2 = B & Cin ;
endmodule
```

注意在各assign 语句之间，是并行执行的，即各语句的执行与语句之间的顺序无关。如上，当A有个变化时，S1、T3、T1 将同时变化，S1的变化又会造成Sum的变化。

### 2.4.3 行为描述方式

行为方式的建模是指采用对信号行为级的描述（不是结构级的描述）的方法来建模。在表示方面，类似数据流的建模方式，但一般是把用initial 块语句或always 块语句描述的归为行为建模方式。行为建模方式通常需要借助一些行为级的运算符如加法运算符（+），减法运算符（-）等。

例2-4-3 一位全加器的行为建模

```
module FA_behav1(A, B, Cin, Sum, Cout );
input A,B,Cin;
output Sum,Cout;
reg Sum, Cout;
reg T1,T2,T3;
always@ ( A or B or Cin )
begin
Sum = (A ^ B) ^ Cin ;
T1 = A & Cin;
T2 = B & Cin ;
T3 = A & B;
Cout = (T1| T2) | T3;
end
endmodule
```

例2-4-4：一位全加器的另一种行为建模

```
module FA_behav2(A, B, Cin, Sum, Cout );
input A,B,Cin;
output Sum,Cout;
reg Sum, Cout;
always@ ( A or B or Cin )
begin
{Count ， Sum} = A + B + Cin ;
end
endmodule
```

在例2-4-4中，采用更加高级（更趋于行为级）描述方式，即直接采用“+”来描述加法。{Count，Sum}表示对位数的扩展（前面介绍过），因为两个1bit 相加，和有两位，低位放在Sum 变量中，进位放在Count 中。

**小提示：**

在实际的设计中，往往是多种设计模型的混合。一般地，对顶层设计，采用结构描述方式，对低层模块，可采用数据流、行为级或两者的结合。如上面的两bit全加器，对顶层模块（Four_bit_FA)采用结构描述方式对低层进行例化，对低层模块（FA）可采用结构描述、数据流描述或行为级描述。对于Verilog的初学者，应重点掌握高层次的描述方法。

# 相关信息

关于其他的相关信息，请访问以下网站

■ 心得交流与问题互助：

http://www.zr-tech.com/bbs

■ ZRtech之FPGA学友会 EDN小组欢迎您的加入

http://group.ednchina.com/2762/

■ ZRtech FPGA/CPLD普及风暴 EDN助学活动火热进行中

http://group.ednchina.com/2762/40844.aspx