Verilog HDL 语言讲义

北京大学信息科学技术学院

2010.10

目录

一 VERILOG 语言简介 4

二 VERILOG 模块的基本概念 4

2.1 模块的结构 6

2.1.1 模块声明和端口定义 6

2.1.2 模块内容 6

三 VERILOG 语言要素 7

3.1 数据类型及常量、变量 7

3.1.1 常量 8

3.1.2 变量 10

3.2 运算符及表达式 11

3.2.1 逻辑运算符 11

3.2.2 位运算符 12

3.2.3 移位运算符 12

3.3 赋值语句和块语句 13

3.3.1 赋值语句 13

3.3.2 块语句 13

3.4 条件语句和循环语句 15

3.4.1 条件语句 15

3.4.2 循环语句 16

3.5 结构说明语句 17

3.5.1 initial 语句 18

3.5.2 always 语句 18

3.5.3 function 语句 19

四 组合逻辑模块 20

4.1 全加器 20

4.2 比较器 20

4.3 多路选择器 21

4.4 总线和总线操作 22

五 时序逻辑模块 23

5.1 状态机 23

5.2 状态机的置位和复位 25

5.2.1 异步置位和复位 25

5.2.2 同步置位与复位 26

六 VERILOG HDL 的描述方式 28

6.1 行为描述方式 28

6.2 RTL 描述方式 29

6.3 结构描述方式 30

七 VERILOG HDL 测试模块(TEST FIXTURE) 30

八 QUARTUS II 和 MAXPLUS II、MODELSIM 的使用 32

8.1 QUARTUS II 的使用 ..... **错误!未定义书签。**
8.2 MAXPLUS II 的使用 ..... **错误!未定义书签。**
8.3 MODELSIM 的使用 ..... **错误!未定义书签。**
九 参考文献 32
其他： ..... 错误!未定义书签。

一 Verilog 语言简介

Verilog HDL 语言作为硬件描述语言的一种，它在现代电子设计中发挥了巨大作用。用该语言可以对波形和电路进行描述。用该语言可以进行仿真验证和面向硬件实现的可综合设计。Verilog HDL 语言可以用于模拟和数字电路硬件描述，本讲义只涉及数字设计部分。1983 年 GDA 公司的 Phil Moorby 首创 Verilog HDL。1989 年 Cadence 公司收购了 GDA。2001 年 IEEE 发布了 Verilog 语言的 Verilog HDL IEEE1364-2001 国际标准。

Verilog HDL 是一种通用的硬件描述语言。它的语法和 C 语言有很多相似之处，所以有 C 编程经验的人容易学习掌握。Verilog 语言适用于不同层次的描述，它可用于行为级、寄存器传输级、和门级以及电路开关级等设计。Verilog HDL 语言得到绝大多数流行 EDA 开发软件的支持。绝大多数的制造厂商都支持 Verilog HDL 设计，所以用 Verilog HDL 进行 IC 设计容易选择制造厂商。Verilog 具有编程语言接口（PLI）,它可以支持 C 语言对 Verilog 内部的访问。

Verilog HDL 的内容比较多。本讲义对最基本的 Verilog 语言语法进行简要讲解，使得读者对最 Verilog 语言有个初步了解。在掌握这些基本概念后，通过练习读者可用简单的写法完成组合逻辑和时序逻辑设计以及测试程序（test fixture ）设计。通过这些学习为读者进一步深入学习硬件描述语言打下基础。

二 Verilog 模块的基本概念

Verilog HDL 程序的基本设计单元是“模块”。一个模块由几个部分组成，下面是两个简单的 Verilog HDL 程序，从中了解 Verilog 模块的基本概念：

[例 1.1]

```
module AOI(a,b,c,d,f);
   input a,b,c,d;
   output f;
   wire a,b,c,d,f;
      assign f=~((a&b) | (c&d));
endmodule
```

这个例子是一个“与-或-非”门电路的模块。

[例 1.2]

```
module muxtwo(ain,bin,select,sout);
   output sout;
   input ain,bin,select;
   mymux2 m(.out(sout),.a(ain),.b(bin),.sl(select));
   //调用由 mymux2 模块定义的实例元件 m，即把已定义过的模块 mymux2 在
   //本模块中具体化为 m
endmodule

module mymux2(out,a,b,sl);
   input a,b,sl;
   output out;
   reg out;
      always @ (sl or a or b)
         if(!sl) out=a;
         else out=b;
endmodule
```

这个例子描述了一个二选一多路器，在这个例子中存在着两个模块。模块 muxtwo 调用了由模块 mymux2 定义的实例部件 m，模块 muxtwo 是上层模块，模块 mymux2 称为子模块。在实例部件 m 中，带“.”的表示被引用模块的端口，名称必须与被引用模块的端口一致，小括号中表示在本模块中与之连接的线路。

通过以上两个例子可以看到：

- Verilog 程序是由模块构成的。每个模块的内容都是位于 module 和 endmodule 两个语句之间。
- 模块是可以层次嵌套的。
- 每个模块要进行端口定义，并说明是输入口还是输出口，然后对模块的功能进行描述。
- Verilog HDL 程序的书写格式自由，一行可以写几个语句，一个语句也可以分写多行。
- 除了 endmodule 语句外，每个语句和数据定义的最后必须有分号。
- 可以用/*……*/或//……对 Verilog HDL 程序的任何部分作注释。

2.1 模块的结构

2.1.1 模块声明和端口定义

模块声明包括模块名字和模块的端口声明，其格式如下：

module 模块名(口 1，口 2，口 3，口 4，……);

在引用模块时其端口可以用两种方法连接：

严格按照模块定义的端口顺序来连接，不用标明原模块定义时规定的端口名，例如：

模块名（连接端口 1 信号名，连接端口 2 信号名，连接端口 3 信号名，……);

在引用时用“.”符号，标明原模块是定义时规定的端口名，例如：

模块名（.端口 1 名（连接信号 1 名），.端口 2 名（连接信号 2 名），……);

推荐使用第二种表示方法，这样表示的好处是可以用端口名与被引用模块的端口对应，而不必严格按端口顺序对应，提高了程序的可读性和移植性。

模块结束的标志为关键字：endmodule。

2.1.2 模块内容

模块的内容包括 I/O 说明，内部信号声明和功能定义。

（1）I/O 说明的格式：

输入口： input [width-1：0] 端口名 1;

输出口： output [width-1：0] 端口名 1;

输入/输出口： inout [width-1：0] 端口名 1;

注：width 为信号位宽，若 width 为 1，则[width-1：0]可省略；若几个端口为同样类型（同为输入、输出或输入/输出，而且 width 一样），则可在同一个语句里声明，例如：

input [width-1：0] 端口名 1，端口名 2，端口名 3，……;

（2）内部信号说明：

在模块内用到的和与端口有关的 wire 和 reg 类型变量的声明。

如：reg [width-1：0] R 变量 1，R 变量 2，……;

wire [width-1：0] W 变量 1，W 变量 2，……;

（3）逻辑功能定义：

有 3 种方法可以在模块中产生逻辑。

用“assign”持续赋值语句定义，如：

assign F=~((A&B)|(C&D));

调用元件（元件例化），如：

and #2 myand(out，a，b);

这个语句调用了一个跟与门（and）一样的名为 myand 的与门，其输入端为 a，b，输出为 out。输出延迟为两个单位时间。

用“always”过程块赋值，如：

```
always @(posedge clk)
begin
   if(reset) out=0;
   else out=out+1;
end
```

注：在 Verilog 模块中所有过程块（如：initial 块、always 块）、连续赋值语句、实例引用都是并行的；在同一个模块中这三者出现的先后顺序没有关系；只有连续赋值语句 assign 和实例引用语句可以独立于过程块而存在于模块的功能定义部分。别的很多和 C 语言类似的语句只能出现在过程块，如 initial 和 always 中。

三 Verilog 语言要素

3.1 数据类型及常量、变量

Verilog HDL 中总共有 19 种数据类型。其中 4 个最基本的数据类型是 reg 型、wire 型、integer 型和 parameter 型。其他类型有 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。

Verilog HDL 语言中也有常量和变量之分。它们分别属于以上这些类型。

3.1.1 常量

整数(integer)型

在 Verilog HDL 中，整型常量的格式如下：

+/-<位宽><进制><数字>

整数常量有以下 4 种进制：

二进制（b 或 B）

十进制（d 或 D 或缺省）

十六进制（h 或 H）

八进制（o 或 O）

在表达式中，位宽指明了数字的精确位数。例如：

```
8'b10101100   //位宽为 8 的数的二进制表示，'b 表示二进制
8'ha2         //位宽为 8 的数的十六进制表示，'h 表示十六进制
```

在数字电路中，x 代表不定值，z 代表高阻值。z 还有一种表达方式是可以写作“？”，例如：

```
4'b10x0       //位宽为 4 的二进制数，从低位数起第 2 位为不定值
4'b101z       //位宽为 4 的二进制数，从低位数起第 1 位为高阻值
12'd?         //位宽为 12 的十进制数，其值为高阻值
```

在书写和使用整数时应注意以下问题：

1. 较长的数字将可用下划线分开，但不可以用在位宽和进制处，只能用在具体的数字之间，而且下划线不能用作首字符。例如：

   ```
   16'b1010_1011_1111_1010    //合法格式
   8'b_0011_1010              //非法格式
   ```

2. 缺省位宽为 32 位，缺省进制为十进制；但若说明了进制而无说明位宽时，其宽度为相应值中定义的位数。如：

   ```
   'o721          //9 位八进制数
   'hAF           //8 位十六进制数
   ```

3. 如果定义的位宽比实际位数长，通常在左边填 0 补位，但如果数最左边一位为 x 或 z 值，就相应地用 x 或 z 在左边补位；如果定义的位宽比实际位数小，那么最

左边的位相应地被截断。

4. 整数可以带符号，并且+/-号应写在最左边，而不可以放在位宽、进制和具体的数之间。例如：

   -8’d5　　//这个表达式表示 5 的补数（用八位二进制数表示）

   8’d-5　　//非法格式

5. 数字中不能有空格，但在表示进制的字母两侧可以有空格。

参数（parameter）型

在 Verilog HDL 中用 parameter 来定义常量，即用 parameter 来定义一个标识符代表一个常量，称为符号常量，即标识符形式的常量。parameter 型数据是一种常数型的数据，其格式如下：

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

parameter 是参数型数据的确认符。确认符后跟着一个用逗号分隔开的赋值语句表。在每一个赋值语句的右边必须是一个常数表达式，即该表达式只能包含数字或先前已经定义过的参数。例如：parameter　byte_size=8，byte_msb=byte_size-1；

在模块或实例引用时，可通过参数传递改变在被引用模块或实例中已定义的参数。

[例 2.1]

```
module Decode(A,F);
  parameter   Width=1，Polarity=1;
  .
  .
  .
endmodule
module Top
  wire[3:0] A4;
  wire[4:0] A5;
  wire[15:0] F16;
  wire[31:0] F32;
  Decode #（4,0） D1（A4，F16）;
  Decode #（5）D2（A5，F32）;
endmodule
```

这个例子在引用模块 Decode 实例时，用了两个参数类型变量：Width 和 Polarity 且都为1。在 Top 模块中引用 Decode 实例时，可通过参数的传递来改变定义时已规定的参数值，即通过#（4，0）实际调用的是参数 Width 和 Polarity 分别为 4 与 0 的 Decode 模块；通过#（5）实际调用的是参数 Width 为 5，而 Polarity 仍为 1 的 Decode 模块。

3.1.2 变量

wire 型

wire 型数据用来表示以 assign 关键字指定的组合逻辑信号。Verilog 程序模块中输入、输出信号类型默认时自动定义为 wire 型。wire 型信号可以用做任何方程式的输入，也可以用做“assign”语句或实例元件的输出。wire 型信号的格式如下：

wire [n-1：0] 数据名 1，数据名 2，……，数据名 i；

wire [n：1] 数据名 1，数据名 2，……，数据名 i；//共有 i 条总线，每条总线内有 n 条线路

reg 型

reg 是寄存器数据类型的关键字，通过赋值语句可以改变寄存器储存的值，其作用与改变触发器储存的值相当。设计者可以通过结构语句来控制是否执行这些赋值语句，这些控制结构描述了硬件触发条件，例如时钟的上升沿和多路器的选通信号。

reg 型数据常用来表示“always”模块内的指定信号，常代表触发器。在设计中要由“always”模块通过使用行为描述语句来表达逻辑关系。在“always”块内被赋值的每一个信号都必须定义成 reg 型。

reg 型数据的格式如下：

reg [n-1：0] 数据名 1，数据名 2，……，数据名 i；

reg [n：1] 数据名 1，数据名 2，……，数据名 i；

Verilog HDL 可以通过对 reg 型变量建立数组来对存储器建模，可以描述 RAM 型存储器、ROM 存储器和 reg 文件。在 Verilog HDL 语言中没有多维数组存在。但是可以通过扩展 reg 型数据的地址范围来生成 memory 型数据。其格式如下：

reg [n-1：0] 存储器名[m-1：0]；

reg [n-1：0] 存储器名[m：1];

在这里，reg[n-1：0]定义了存储器中每一个存储单元的大小，即该存储单元是一个 n 位的寄存器，存储器名后的[m-1：0]或[m：1]则定义了该存储器中有多少个这样的寄存器。

在同一个数据类型声明语句里，可以同时定义存储器型数据和 reg 型数据，例如：

reg [15：0] mema[255:0]，writereg，readreg；

如果想对 memory 中的存储单元进行读写操作，必须指定该单元在存储器中的地址，例如：

```
mema[3]=0；    //合法赋值语句
mema=0；       //非法赋值语句
```

3.2 运算符及表达式

Verilog HDL 语言底运算符按功能可以分为以下几类：

算术运算符（+，－，×，／，%）；
赋值运算符（=，<=）；
关系运算符（>，<，>=，<=，==，!=，===，!==）；
逻辑运算符（&&，||，!）；
条件运算符（？：）；
位运算符（~，|，^，&，^~）；
移位运算符（<<，>>）；
拼接运算符（{}）；

以下只简单介绍逻辑运算符、位运算符、移位运算符和拼接运算符。

3.2.1 关系运算符

```
>     //大于
<     //小于
>=    //大于等于
<=    //小于等于
==    //等于
!=    //不等于
===   //等于
!==   //不等于
```

进行逻辑运算时，如果声明的关系是假的，则返回值是 0，如果声明的关系是真的，则返回值为 1。

“==”和“===”的不同之处在于：用“==”运算时，如果操作数中某些位是不定值 x 或高阻值 z，结果为不定值 x，而用“===”运算时，在对操作数进行比较时对不定值 x 和高阻值 z 也进行比较，只有两个操作数完全相同时，其结果才是 1，否则就是 0。

3.2.2 位运算符

~ //按位取反

| //按位或

^ 按位异或

& //按位与

^~ //按位同或（异或非）

3.2.3 移位运算符

“<<”（左移）和“>>”（右移）运算符的使用方法如下：

a<<n 或 a>>n

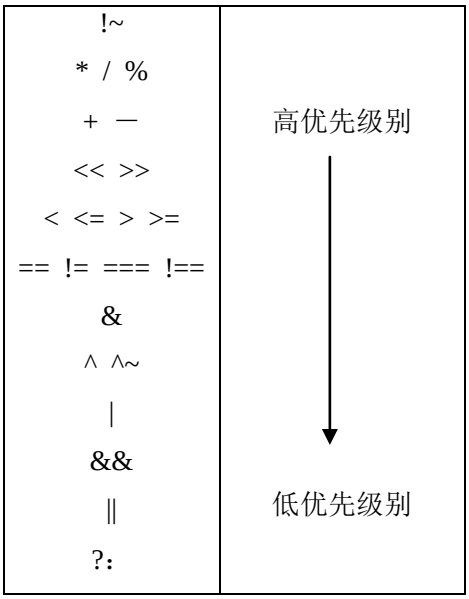
a 代表要进行移位的操作数，n 代表要移几位。这两种移位运算都用 0 来填补移出的空位。

拼接运算符的用法如下：

{a，b[3：0]} //等价于{a，b[3]，b[2]，b[1]，b[0]}；

各种运算符的优先级别如下所示：

优先级别



3. 3 赋值语句和块语句

3.3.1 赋值语句

Verilog HDL 语言中，信号有两种赋值方式：非阻塞赋值方式（如：b<=a）和阻塞赋值方式（如：b=a）；

1) 非阻塞（Non-Blocking）赋值方式：

（1）赋值所在块结束之后才能真正完成赋值操作；

（2）b 的值并不是立刻改变；

2) 阻塞（Blocking）赋值方式：

（1）赋值完成之后才能结束该语句所在块，或者执行下一条语句；

（2）b 的值在赋值语句执行执行完毕之后立刻改变；

3.3.2 块语句

顺序块

顺序块内的语句是按顺序执行的，即只有上面一条语句执行完后下面的语句才能执行；每条语句的延迟时间是相对于前一条语句的仿真时刻而言的；直到最后一条语句执行完，程序流程控制才跳出该语句块。

以下是一个顺序块的例子：

[例 2.2]

```
parameter   d=50;
reg [7：0]   r;
begin                    //由一系列延迟产生的波形
   #d    r=’h35;
   #d    r=’hE2;
   #d    r=’h00;
   #d    r=’hE7;
   #d    ->end_wave;      //触发事件 end_wave
end
```

并行块

并行块内的语句是同时执行的，即程序流程控制一进入到该并行块，块内语句就开始同时地执行；块内每条语句的延迟时间是相对于程序流程控制进入到块内的仿真时刻而言的；延迟时间是用来给赋值语句提供执行时序的；当按时间时序排序在最后的语句执行完后，或一个 disable 语句执行时，程序流程控制跳出该程序块。

以下是一个并行块的例子：

[例 2.3]

```
fork
   #50    r=’h35;
   #100   r=’hE2;
   #150   r=’h00;
   #200   r=’hE7;
   #250   ->end_wave;
join
```

这个例子用并行块替代了前面的顺序块产生波形，用这两种方法生成的波形是一样的。

fork 和 join 是并行块的标识符，相当于顺序块的 begin 和 end。在并行块中，各条语句在前还是在后是无关紧要的。

3.4 条件语句和循环语句

3.4.1 条件语句

if_else 语句

Verilog HDL 语言提供了三种形式的 if 语句：

（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 m)　语句 m；

　　else　　语句 n；

case 语句

case 语句是一种多分支选择语句，一般格式如下：

```
case(表达式)
  分支表达式 1：　　语句 1；
  ……
  分支表达式 m：　　语句 m；
  默认项（default 项）　　语句 n；
endcase
```

以下为一个简单的使用 case 语句的例子：

[例 2.4]

```
reg[15：0]　rega;
reg[9：0]　result;
  case(rega)
```

```
        16'd0:    result=10'b0111111111;
        16'd1:    result=10'b1011111111;
        16'd2:    result=10'b1101111111;
        16'd3:    result=10'b1110111111;
        16'd4:    result=10'b1111011111;
        16'd5:    result=10'b1111101111;
        16'd6:    result=10'b1111110111;
        16'd7:    result=10'b1111111011;
        16'd8:    result=10'b1111111101;
        16'd9:    result=10'b1111111110;
        default:  result=10'bx;
    endcase
```

注：使用条件语句应注意条件的完备性，否则会有意想不到的结果。例如下面这个例子就生成了并不想要的锁存器。

```
always@(al  or  d)
  begin
    if(al)   q<=d;        //产生锁存器
  end
```

正确的写法应该如下：

```
always@(al  or d)
  begin
    if(al)   q<=d;
    else  q<=0;           //无锁存器
  end
```

3.4.2 循环语句

Verilog HDL 语言中存在着 4 种类型的循环语句，用来控制执行语句的执行次数。

forever：连续的执行语句。

repeat：连续执行一条语句 n 次。

while：执行一条语句直到某个条件不满足，如果一开始就不满足则语句一次也不能被执行。

for：执行语句直到某个条件不满足，实际上相当于 while 语句。

下面只介绍 while 语句和 for 语句：

while 语句

while 语句的格式如下：

```
while(表达式) 语句；
while(表达式)
begin
   语句 1；
   语句 2；
   ……
end
```

for 语句

for 语句的格式如下：

```
for(循环变量赋初值；循环结束条件；循环变量增值) 语句；
for(循环变量赋初值；循环结束条件；循环变量增值)
  begin
     语句 1；
     语句 2；
     ……
  end
```

3.5 结构说明语句

Verilog HDL 语言中的任何过程模块都从属于以下 4 种结构的说明语句：

Initial 说明语句；

Always 说明语句；

Task 说明语句；

Function 说明语句。

下面只对 initial、always 和 function 语句加以深入介绍。

一个程序模块可以有多个 initial 和 always 过程块。每个 initial 盒 always 说明语句在仿真的一开始同时立刻开始执行。Initial 语句只执行一次，而 always 语句则是不多的重复直到仿真过程结束。但 always 语句后跟着的过程块是否运行，则要看它的触发条件是否满足。

Function 即为函数，函数的目的是返回一个值，以用于表达式的计算。

3.5.1 initial 语句

initial 语句的格式如下：

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

initial 语句通常用来在仿真开始时对各变量进行初始化，或者用 initial 语句来生成激励波形。

3.5.2 always 语句

always 语句由于其不断重复执行特性，只有和一定的时序控制结合在一起才有用。其一般格式如下：

```
always<时序控制>     语句；或
always<时序控制>
  begin
    语句 1；
    语句 2；
    ……
  end
```

以下是两个使用 always 语句的例子：

[例 2.5]

```
always  #half_period  areg=~areg;
```

这个例子里，生成了一个周期为 2*half_period 的无限延续的信号波形。

[例 2.6]

```
reg[7：0] counter；
reg tick；
  always@(posedge areg)
```

```
begin
   tick=~tick;
   counter=counter+1;
end;
```

这个例子中，每当 areg 信号的上升沿出现时则把 tick 信号反相，并且 counter 增加 1。这种时间控制是 always 语句最常用的。posedge 表示上升沿，negedge 表示下降沿。

3.5.3 function 语句

function 语句的一般格式为：

```
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
```

<返回值位宽或类型说明>是一个可选项，如果缺省，则返回一位寄存器类型的数据。

以下是一个用函数定义的七段译码器。这是一种真值表表述法，直接将问题说清楚，剩下的门级设计由编译软件完成。

[例 2.7]

```
module segment7(din,dout);
 input [3:0] din;
 output [6:0] dout;

 function [6:0] decode;          //函数定义
 input [3:0] din;                //函数只有输入，输出为函数名本身
  case (din)
  4'b0000: decode=7'b1111110;
  4'b0001: decode=7'b0110000;
  4'b0010: decode=7'b1101101;
  4'b0011: decode=7'b1111001;
  4'b0100: decode=7'b0110011;
  4'b0101: decode=7'b1011011;
  4'b0110: decode=7'b1011111;
  4'b0111: decode=7'b1110000;
```

```
    4'b1000: decode=7'b1111111;
    4'b1001: decode=7'b1111011;
    default: decode=7'b0000000;
    endcase
  endfunction
  assign dout=decode(din);
endmodule
```

这个例子中，七段译码器输出电平“1”则为亮，输出电平“0”则为暗。

四 组合逻辑模块

4.1 全加器

在数字电路课程里已学过加法电路，即全加器。用 Verilog HDL 描述全加器是相当容易的，只需要把运算表达式写出就可以了。

以下是一个位数可以由用户定义的全加器模块：

[例 3.1]

```
module Adder(Cin,A,B,Sum,Cout);
  parameter WIDTH=4;
  input Cin;
  input [WIDTH-1:0] A,B;
  output Cout;
  output [WIDTH-1:0] Sum;

  assign {Cout,Sum}=A+B+Cin;//完成加法操作，输出最高位为 Cout

endmodule
```

在这个例子里，A 和 B 表示两个加数，Cin 表示来自低位的进位，Cout 表示向高位的进位。WIDTH 是加数的位宽。

4.2 比较器

数值大小比较逻辑在计算逻辑中是常用的一种逻辑电路，以下是一个位数可以由用户定义的比较电路模块：

[例 3.2]

```
module compare(A,B,AGB,ASB,AEB);
    parameter WIDTH=8;
    input[WIDTH-1:0] A,B;
    output AGB,ASB,AEB;
    reg AGB,ASB,AEB;

    always @ (A or B)
    begin
        AGB=(A>B)?1:0;        //若 A>B，则 A 大于 B 的信号为 1，否则为 0
        ASB=(A<B)?1:0;        //若 A<B，则 A 小于 B 的信号为 1，否则为 0
        AEB=(A==B)?1:0;       //若 A=B，则 A 等于 B 的信号为 1，否则为 0
    end
endmodule
```

在这个例子里，A 和 B 是要比较的两个输入，AGB、ASB 和 AEB 分别是 A 大于 B、A 小于 B 和 A 等于 B 的信号。WIDTH 是 A 和 B 的位宽。

4.3 多路选择器

多路选择器是一个多输入，单输出的组合逻辑电路，它根据地址码（选择码）的不同，从多个输入数据流中选取一个，让其输出到公共的输出端。

以下是一个带使能控制信号（nCS），输入和输出的数据位宽可以由用户定义的八路数据通道选择器模块：

[例 3.3]

```
module Mux_8(addr,S_in0,S_in1,S_in2,S_in3,S_in4,S_in5,S_in6,S_in7,S_out,nCS);
  parameter WIDTH=4;
  input[2:0] addr;
  input[WIDTH-1:0] S_in0,S_in1,S_in2,S_in3,S_in4,S_in5,S_in6,S_in7;
  input nCS;
  output[WIDTH-1:0] S_out;
  reg[WIDTH-1:0] S_out;

  always @(addr or S_in0 or S_in1 or S_in2 or S_in3 or S_in4 or S_in5 or S_in6 or S_in7
  or nCS)
      begin
```

```
    if(!nCS)                        //nCS 低电平使多路选择器工作
        case(addr)
        3'b000: S_out=S_in0;
        3'b001: S_out=S_in1;
        3'b010: S_out=S_in2;
        3'b011: S_out=S_in3;
        3'b100: S_out=S_in4;
        3'b101: S_out=S_in5;
        3'b110: S_out=S_in6;
        3'b111: S_out=S_in7;
        endcase
    else                            //nCS 高电平关闭多路选择器
        S_out=0;
    end
endmodule
```

4.4 总线和总线操作

总线是运算部件之间数据流通的公共通道。各运算部件和数据寄存器可以通过带控制端的三态门与总线连接。通过对控制端电平的控制来确定在某一时间片断内，总线归哪些部件使用（任何时间只能有一个部件发送，但可以有多个接收）。

下面是一个简单的与总线有接口的模块如何对总线进行操作的例子（总线端口以及数据寄存器位宽可以由用户定义）：

[例 3.4]

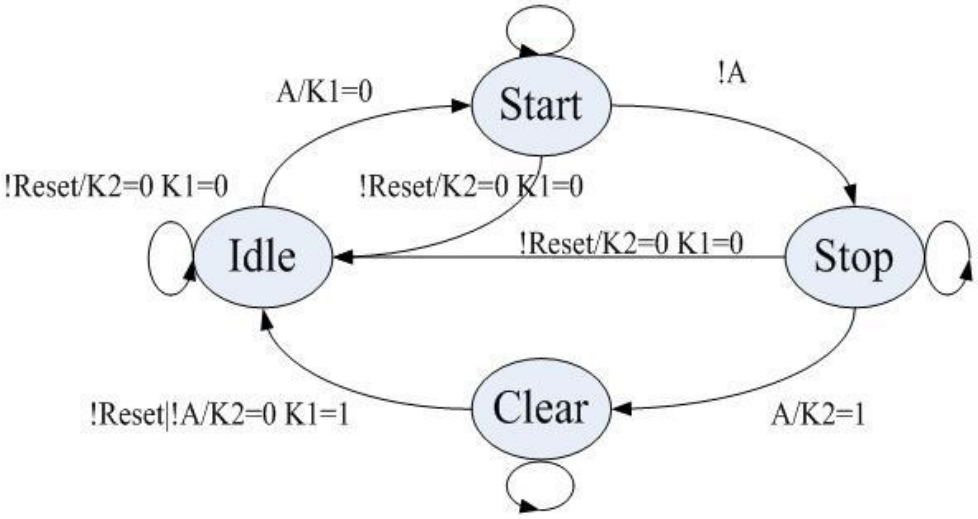
```
module SampleOfBus(DataBus,outen,write);
    parameter WIDTH=12;
    inout[WIDTH-1:0] DataBus;           //总线双向端口
    input outen;                        //向总线输出数据的控制电平
    input write;                        //将总线数据写入寄存器的控制信号
    reg[WIDTH-1:0] outregs;             //模块内数据寄存器

    assign DataBus=(outen)?outregs:12'hzzz;  //当 outen 为高电平时
    //通过总线把储存在 outregs 的计算结果输出，否则为高阻态

    always @ (posedge write)            //每当 write 信号上跳沿时
```

```
    begin
        outregs<=DataBus*5;          //接收总线上数据并乘 5
                                     //把计算结果存入 outregs
    end
endmodule
```

五 时序逻辑模块

5.1 状态机

在 Verilog HDL 中可以用很多种方法来描述有限状态机，最常用的办法就是用 always 语句和 case 语句。下图所示的状态转移图表示了一个简单的有限状态机：



上图所示的状态转移图表示了一个 4 状态的有限状态机，它的同步时钟是 Clock，输入信号是 A 和 Reset，输出信号是 K2 和 K1。状态的转移只能在同步时钟（Clock）的上升沿时发生，往哪个状态的转移则取决于目前所在的状态和输入的信号（Reset 和 A）。

下面就是用 Verilog HDL 语言实现的上述状态机的例子：

[例 4.1]

```
module fsm(Clock,Reset,A,K2,K1);
  parameter IDLE=2'b00,START=2'b01,STOP=2'b10,CLEAR=2'b11;//声明四个状态
  input Clock,Reset,A;
  output K2,K1;
  reg K2,K1;
```

```
reg[1:0] state;

always @(posedge Clock)
  if(!Reset)
    begin
      state<=IDLE;
      K2<=0;K1<=0;
    end
  else
    case(state)
      IDLE: begin
              if(A)
                begin
                  state<=START;
                  K1<=0;
                end
              else
                state<=IDLE;
            end
      START: begin
               if(!A)
                 state<=STOP;
               else
                 state<=START;
             end
      STOP: begin
              if(A)
                begin
                  state<=CLEAR;
                  K2<=1;
                end
              else
                state<=STOP;
            end
      CLEAR: begin
               if(!A)
```

```
                begin
                  state<=IDLE;
                  K2<=0;
                  K1<=1;
                end
              else
                state<=CLEAR;
            end
        endcase
    endmodule
```

5.2 状态机的置位和复位

5.2.1 异步置位和复位

异步置位与复位是与时钟无关的。当异步置位与复位到来时它们立即分别置触发器的输出为 1 和 0，不需要等待时钟沿到来才置位或复位。把它们列入 always 块的事件控制括号内就能触发 always 块的执行，因此，当它们到来时就能立即执行一次指定的操作。所以当触发条件（如时钟的跳变沿）反复出现时，就可以反复执行指定的操作。

状态机的异步置位和复位是用 always 和事件控制实现的。先让我们来看一下事件控制的语法：

事件控制语法：

@(<沿关键词 时钟信号
or 沿关键词 复位信号
or 沿关键词 置位信号>)

沿关键词包括 posedge （用于高电平有效的 set，reset 或上升沿触发的时钟）和 negedge（用于低电平有效的 set，reset 或下降沿触发的时钟），信号可以按任意顺序列出。

事件控制实例：

异步高电平有效的置位（时钟的上升沿）：@（posedge clk or posedge set）

异步低电平有效的复位（时钟的上升沿）：@（posedge clk or negedge reset）

异步低电平有效的置位和高电平有效的复位（时钟上升沿）：@（posedge clk or negedge set or posedge reset）

下面是一个异步的具有高电平有效的置位/复位端的 D 触发器的例子：

[例 4.2]

```
module dff_asy(q,qb,d,clk,set,reset);
  input d,clk,set,reset;
  output q,qb;
  reg q,qb;    //声明 q 和 qb 为 reg 类型，因为它需要在 always 块内赋值

  always @ (posedge clk or posedge set or posedge reset)
    begin
      if(reset)
        begin
          q<=0;       //复位
          qb<=1;
        end
      else
        if(set)
          begin
            q<=1;    //置位
            qb<=0;
          end
        else
          begin
            q<=d;    //与时钟同步的逻辑
            qb<=~d;
          end
      end
endmodule
```

5.2.2 同步置位与复位

同步置位与复位是指只有在时钟的有效跳变沿时刻置位或复位，信号才能使触发器置位或复位（即，使触发器的输出分别转变为逻辑 1 或 0）。因此不要把 set 和 reset 信号名列入 always 块的事件控制表达式，因为当它们有变化时不应触发 always 块的执行。相反，always 块的执行应只由时钟有效跳变沿触发，是否置位或复位应在 always 块中首先检查 set 和 reset 信号的电平。所以 set 和 reset 信号的电平必须至少维持大于时钟沿的间隔时间，否则 set 和 reset 不能每次都能有效地完成置位和复位的工作。为此在编写测试模块和设计与其配合的电路时要注意这个问题。

事件控制语法：

@(<沿关键词 时钟信号>)

其中，沿关键词是指在 posedge（正沿触发的时钟）或 negedge（负沿触发的时钟）事件控制实例：

正沿触发

@(posedge clk)

负沿触发

@(negedge clk)

以下是一个同步的具有高电平有效的置位/复位端的 D 触发器的例子：

[例 4.3]

```
module dff_sync(q,qb,d,clk,set,reset);
  input d,clk,set,reset;
  output q,qb;
  reg q,qb;

  always @ (posedge clk)
    begin
      if(reset)
        begin
          q<=0;        //复位
          qb<=1;
        end
      else
        if(set)
          begin
            q<=1;    //置位
            qb<=0;
          end
        else
          begin
            q<=d;   //与时钟同步的逻辑
            qb<=~d;
          end
    end
endmodule
```

六 Verilog HDL 的描述方式

Verilog 模型可以是实际电路中不同级别的抽象。抽象的级别和他们对应的模块类型常常可以分为以下 5 种，即 verilog 语法支持 5 中不同的描述方法：

1. 系统级（system）
2. 算法级（algorithmic）
3. RTL 级（Register Transfer-Level）
4. 门级（gate-level）
5. 开关级（switch-level）

通常把系统级、算法级称为行为级，门级和开关级称为结构级。即 Verilog HDL 语言有三种描述方式：行为描述方式、RTL 级描述方式和结构描述方式。

6.1 行为描述方式

行为描述方式是对系统数学模型的描述，其抽象程度比 RTL 描述方式和结构描述方式更高。在行为描述方式的程序中大量采用算术运算、关系运算、惯性延时、传输延时等难于进行逻辑综合和不能进行逻辑综合的语句。一般说来，采用行为描述方式的 VHDL 语言程序主要用于系统数学模型的仿真或者系统工作原理的仿真。

下面给出一个乘法器的行为级描述的例子：

[例 5.1]

```
module multi(a,b,c);
input [3:0] a,b;
output [7:0] c;

reg [7:0] p;

always @ (a or b)
begin
  p=8'b00000000;
  if (a[0])
    p[4:0]=p[4:0]+{1'b0,b};

  if (a[1])
    p[5:1]=p[5:1]+{1'b0,b};
```

```
  if (a[2])
    p[6:2]=p[6:2]+{1'b0,b};

  if (a[3])
    p[7:3]=p[7:3]+{1'b0,b};
end
assign c=p;
endmodule
```

在上面这个例子中，输入是两个 4 位的无符号数，输出是它们相乘得到的 8 位无符号数。这里的行为描述可以被综合成组合逻辑。

6.2 RTL 描述方式

采用行为描述方式的程序，在一般情况下只能用于行为层次的仿真，而不能进行逻辑综合，只有将其改写为 RTL 描述方式才能进行逻辑综合，也就是说，RTL 描述方式才是真正可以进行逻辑综合的描述方式。

RTL 描述方式，是一种明确规定寄存器描述的方法。由于受逻辑综合的限制，在采用 RTL 描述方式时，要么采用寄存器硬件的一一对应的直接描述，要么采用寄存器之间的功能描述。

下面给出一个带清零端 4 位寄存器的 RTL 级描述的例子：

[例 5.2]

```
module reg_4(d,clk,clr,q);
input clk,clr;
input [3:0] d;
output [3:0] q;

reg [3:0] q;

always @ (posedge clk or posedge clr)
begin
  if (clr)
    q<=0;
  else
    q<=d;
```

```
end

endmodule
```

上面这个例子是一个异步清零的 4 位寄存器，clr 为 1 时寄存器清零，否则在 clk 的上跳沿将输入写入寄存器。

6.3 结构描述方式

结构描述方式，就是在多层次的设计中，高层次的设计模块调用低层次的设计模块，或者直接用门电路设计单元来构成一个复杂的逻辑电路的描述方法。结构描述方式最能提高设计效率，它可以将已有的设计成果，方便地用到新的设计中去。

下面给出一个全加器的结构描述方式的例子：

[例 5.3]

```
module full_adder(Cin,A,B,Sum,Cout);
input Cin,A,B;
output Cout,Sum;
wire s_tmp,t1,t2,t3;

and  (t1,A,B),
     (t2,A,Cin),
     (t3,B,Cin);
xor  (s_tmp,A,B),
     (Sum,S_tmp,Cin);
or   (Cout,t1,t2,t3);
endmodule
```

七 Verilog HDL 测试模块(Test Fixture)

Verilog 还可以用来描述变化的测试信号。描述测试信号的变化和测试过程的模块叫做测试平台，(Testbench 或 Testfixture)，它可以对 Verilog 描述的电路模块进行动态的全面测试。通过观测被测试模块的输出信号是否符合要求，可以调试和验证逻辑系统的设计和结构正确与否，并发现问题及时修改。

测试程序与一般的 Verilog 模块没有根本的区别，其特点主要表现为：

- 测试模块只有模块名字，没有端口列表；

- 输入信号<激励信号>一般定义为 reg 型，以保持信号值；输出信号（显示信号）一般定义为 wire 型；
- 在测试模块中调用被测试模块；
- 一般用 initial，always 过程块来定义激励信号波形。

测试模块中要写很多激励波形。典型的在 initial 中对信号赋初值。然后再 always 等句子中可以给出变化波形。

比如一个周期为 100ns 的时钟的写法如下：

```
`timescale  1ns
Reg clk;
Initial  clk=0;
always@  # 50 clk<=~clk;
```

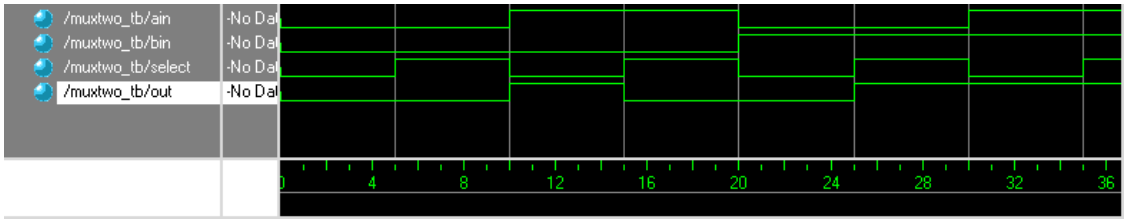
以下是用 Verilog 语言对前面二选一多路器进行前仿真的一个例子：

[例 6.1]

```
`include"muxtwo.v"
module muxtwo_tb;
  reg ain,bin,select;
  wire out;

initial
begin
        ain=0;
        bin=0;
        select=0;

        #5 select=1;
        #5 ain=1;select=0;
        #5 select=1;
        #5 ain=0;bin=1;select=0;
        #5 select=1;
        #5 ain=1;bin=1;select=0;
        #5 select=1;
end
```

```
muxtwo m(.ain(ain),.bin(bin),.select(select),.sout(out));

endmodule
```

以上是 2 选 1 多路器的一个测试平台，用 ModelSim 编译仿真，仿真波形如下所示：



八 Verilog 编译和下载软件的使用

（暂空）

九 参考文献

[1] 《Verilog 数字系统设计教程》，夏宇闻编著，北京航空航天大学出版社，2003 年 7 月第 1 版

[2] 《Verilog HDL 程序设计教程》，王金明编著，徐志军主审，人民邮电出版社，2004 年 1 月第 1 版

[3] *MODELING, SYNTHESIS, AND RAPID PROTOTYPING WITH THE VERILOG HDL*, Michael D.Ciletti, University of Colorado, Colorado Springs，1999

[4] *IEEE Standard Verilog Hardware Description Language*, IEEE Computer Society, Sponsored by the Design Automation Standards Committee, 28 September 2001