Verilog HDL

第一节 Verilog HDL入门

- 学习内容
  - HDL概念
  - Verilog的历史
  - Verilog的用途
  - Verilog的特点
  - Verilog的抽象层次
  - Verilog语言的描述风格
  - Verilog和C语言的区别
  - 示例
    - 语言特点

什么是硬件描述语言HDL

- 具有特殊结构能够对硬件逻辑电路的功能进行描述的一种高级编程语言
- 这种特殊结构能够：
  - 描述电路的连接
  - 描述电路的功能
  - 在不同抽象级上描述电路
  - 描述电路的时序
  - 表达具有并行性
- HDL主要有两种：Verilog和VHDL
  - Verilog起源于C语言，因此非常类似于C语言，容易掌握
  - VHDL起源于ADA语言，格式严谨，不易学习。
  - VHDL出现较晚，但标准化早。IEEE 1706-1985标准。

Verilog的历史

- Verilog HDL是在1983年由GDA(GateWay Design Automation)公司的Phil Moorby所创。Phil Moorby后来成为Verilog-XL的主要设计者和Cadence公司的第一个合伙人。
- 在1984~1985年间，Moorby设计出了第一个Verilog-XL的仿真器。
- 1986年，Moorby提出了用于快速门级仿真的XL算法。
- 1990年，Cadence公司收购了GDA公司
- 1991年，Cadence公司公开发表Verilog语言，成立了OVI(Open Verilog International)组织来负责Verilog HDL语言的发展。
- 1995年制定了Verilog HDL的IEEE标准，即IEEE1364。

Verilog的用途

- **Verilog的主要应用包括：**
  - **ASIC和FPGA工程师编写可综合的RTL代码**
  - 高抽象级系统仿真进行系统结构开发
  - 测试工程师用于编写各种层次的测试程序
  - 用于ASIC和FPGA单元或更高层次的模块的模型开发

Verilog的特点

- 支持不同抽象层次的精确描述以及混合模拟，如行为级、 RTL级、结构级等
- 设计、测试、模拟所用的语法都相同
- 较高层次的描述与具体工艺无关
- 提供了类似C语言的高级程序语句，如if-else， for, while, break, case，loop以及int等数据类型
- 提供了算术、逻辑、位操作等运算符
- 包含完整的组合逻辑元件，如and、or、xor等，无需自行定义
- 支持元件门级延时和元件门级驱动强度(nmos, pmos)

Verilog的抽象层次

- 系统级：C等高级语言描述
- 行为级：模块的功能描述
- RTL级：寄存器与组合电路的合成
- 逻辑门级：基本逻辑门的组合（and, or, nand）
- 开关级：晶体管开关的组合(nmos, pmos)

Verilog 语言的描述形式

- 结构型描述
  - 从电路结构的角度来描述电路模块
  - 通过实例描述，将Verilog已定义的基本实例嵌入到语言中
- 数据流型描述
  - 通过assign连续赋值实现组合逻辑功能的描述
  - 连续赋值语句右边所有的变量受持续监控，只要这些变量有一个发生变化，整个表达式将被重新赋值给左端
- 行为描述
  - 只对系统行为与功能进行描述，不涉及时序电路实现，是一种高级语言描述的方法，有很强的通用性
  - 主要包括过程结构、语句块、时序控制、流控制4个方面

Verilog和C语言区别

- 与C语言的联系与区别

项目	C	Verilog
执行顺序	顺序执行	并行执行
时序概念	无延迟	存在延迟
语法限制	灵活完善	限制严格，需要有数字电路的知识

示例

边沿触发型D触发器

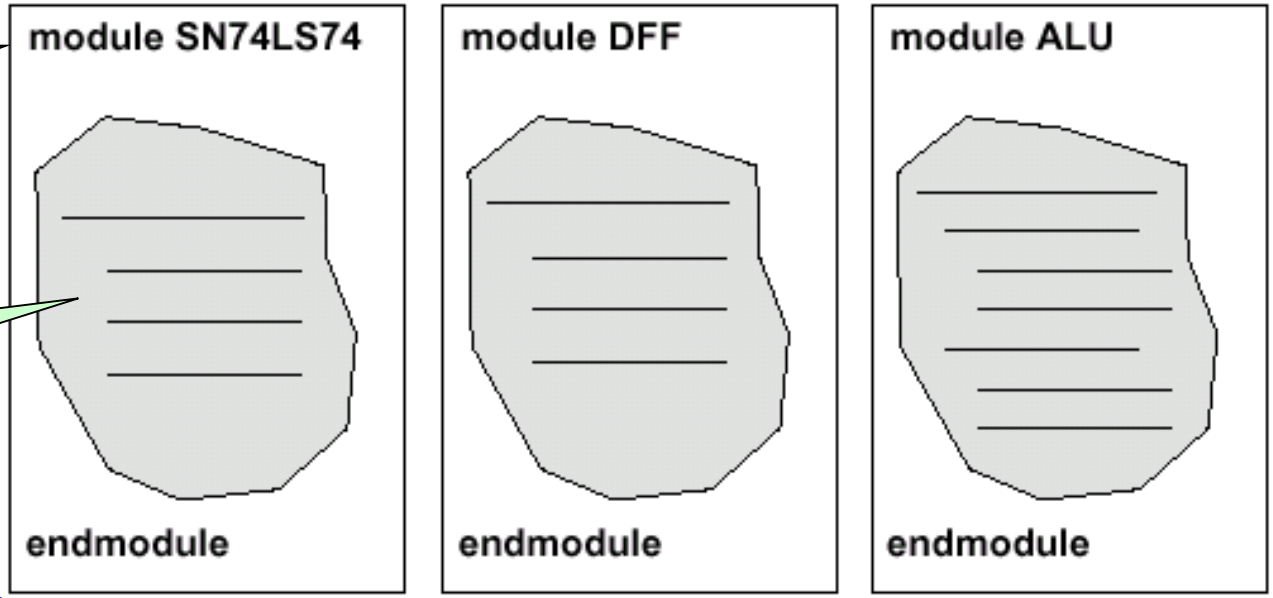
```
module DFF1 (d, clk, q);        //D触发器基本模块
    output q;
    input clk,d;
    reg q;
    always @ (posedge clk)      //clk上升沿启动
    q<=d;                       //当clk有上升沿时d被锁入q
endmodule
```

语言的主要特点

module(模块)

module是层次化设计的基本构件

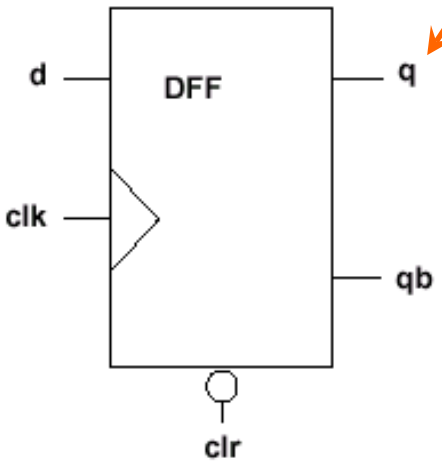
逻辑描述放在module内部



- module能够表示：
  - 物理块，如IC或ASIC单元
  - 逻辑块，如一个CPU设计的ALU部分
  - 整个系统
- 每一个模块的描述从关键词*module*开始，有一个名称（如SN74LS74，DFF，ALU等等），由关键词*endmodule*结束。

语言的主要特点—模块端口(module ports)

端口等价于硬件的引脚(pin)




```
module DFF (d, clk, clr, q, qb);
  input d, clk, clr;
  output q, qb;

endmodule
```

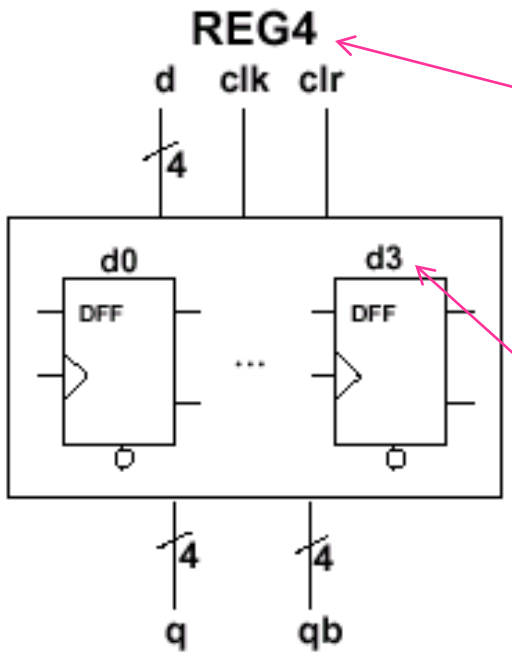
端口在模块名字后的括号中列出

端口可以说明为 ***input, output及inout***

- 注意模块的名称DFF，端口列表及说明
- 模块通过端口与外部通信

语言的主要特点

模块实例化(module instances)




```
module DFF (d, clk, clr, q, qb);
    ....
endmodule

module REG4( d, clk, clr, q, qb);
    output [3: 0] q, qb;
    input [3: 0] d;
    input clk, clr;
    DFF d0 (d[ 0], clk, clr, q[ 0], qb[ 0]);
    DFF d1 (d[ 1], clk, clr, q[ 1], qb[ 1]);
    DFF d2 (d[ 2], clk, clr, q[ 2], qb[ 2]);
    DFF d3 (d[ 3], clk, clr, q[ 3], qb[ 3]);
endmodule
```

语言的主要特点

模块实例化(module instances)

- 可以将模块的实例通过端口连接起来构成一个大的系统或元件。
- 在上面的例子中，**REG4**有模块**DFF**的四个实例。注意，每个实例都有自己的名字(d0, d1, d2, d3)。实例名是每个对象唯一的标记，通过这个标记可以查看每个实例的内部。
- 实例中端口的次序与模块定义的次序相同。
- 模块实例化与调用程序不同。每个实例都是模块的一个完全的拷贝，相互独立、并行。

第二节 Verilog的语言规则

学习内容：

1. 文字规则
2. 操作符

文字规则—空白符和注释

```
module MUX2_1 (out, a, b, sel);
  // Port declarations
    output out;
    input sel, // control input
                b, /* data inputs */ a;
/*
  The netlist logic selects input ”a” when
  sel = 0 and it selects ”b” when sel = 1.
*/
    not (sel_, sel);
    and (a1, a, sel_), (b1, b, sel); // What does this line do?
    or (out, a1, b1);
endmodule
```

单行注释
到行末结束

多行注释，在/*　*/内

格式自由

使用空白符提高可读性及代码组织。Verilog忽略空白符除非用于分开其它的语言标记。

文字规则—整数常量和实数常量

Verilog中，常量(literals)可是整数也可以是实数

- 整数的大小可以定义也可以不定义。整数表示为：

  <size>’<base><value>

  其中 **size**：大小，由十进制数表示的位数(bit)表示。缺省为32位

  base：数基，可为2(b)、8(o)、10(d)、16(h)进制。缺省为10进制

  value：是所选数基内任意有效数字，包括X、Z。

- 实数常量可以用十进制或科学表示法表示。

`12`	unsized decimal (zero-extended to 32 bits)
`'H83a`	unsized hexadecimal (zero- extended to 32 bits)
`8'b1100_0001`	8-bit binary
`64'hff01`	64-bit hexadecimal (zero- extended to 64 bits)
`9'O17`	9-bit octal
`32'bz01x`	Z-extended to 32 bits
`3’b1010_1101`	3-bit number, truncated to 3’b101，高位截断
`6.3`	decimal notation
`32e-4`	scientific notation for 0.0032
`4.1E3`	scientific notation for 4100

文字规则—整数常量和实数常量

- 整数的大小可以定义也可以不定义。整数表示为：
  - 数字中（_）忽略，便于查看
  - 没有定义大小(size)整数缺省为32位
  - 缺省数基为十进制
  - 数基(base)和数字(16进制)中的字母无大小写之分
  - 当数值value大于指定的大小时，截去高位。如 2’b1101表示的是2’b01
- 实数常量
  - 实数可用科学表示法或十进制表示
  - 科学表示法表示方式：

    <尾数><e或E><指数>， 表示： 尾数×$10^{指数}$

文字规则—字符串（string)

Verilog中，字符串大多用于显示信息的命令中。Verilog没有字符串数据类型

- 字符串要在一行中用双引号括起来，也就是不能跨行。
- 字符串中可以使用一些C语言转义(escape)符，如 `\t` `\n`
- 可以使用一些C语言格式符(如%b)在仿真时产生格式化输出：

”This is a normal string”

”This string has a \t tab and ends with a new line\n”

”This string formats a value: val = %b”

文字规则—字符串（string)

转义符及格式符将在验证支持部分讨论

格式符

%h	%o	%d	%b	%c	%s	%v	%m	%t
hex	oct	dec	bin	ACSII	string	strength	module	time

转义符

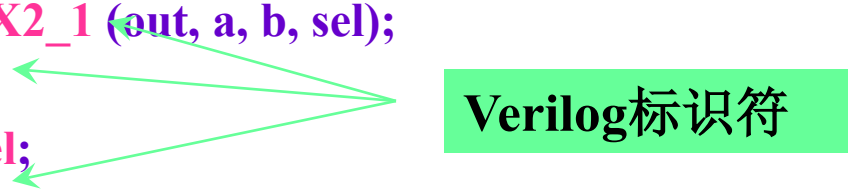
\t	\n	\\	\”	\<1-3 digit octal number>
tab	换行	反斜杠	双引号	ASCII representation of above

格式符%0d表示没有前导0的十进制数

文字规则—标识符(identifiers)

- 标识符是用户在描述时给Verilog对象起的名字
- 标识符必须以字母(a-z, A-Z)或(_)开头，后面可以是字母、数字、($)或(_)。
- 最长可以是1023个字符
- 标识符区分大小写，sel和SEL是不同的标识符
- 模块、端口和实例的名字都是标识符

```
module MUX2_1 (out, a, b, sel);
output out;
input a, b, sel;
   not  not1 (sel_, sel);
   and and1 (a1, a, sel_);
   and and2 (b1, b, sel);
   or   or1   (out, a1, b1);
endmodule
```



文字规则—标识符(identifiers)

- 有效标识符举例：

  shift_reg_a

  busa_index

  _bus3

- 无效标识符举例：

  34net　　// 开头不是字母或“_”

  a*b_net　// 包含了非字母或数字，　“$”“_”

  n@238　　//包含了非字母或数字，　“$”“_”

- Verilog区分大小写，所有Verilog关键词使用小写字母。

操作符

下表以优先级顺序列出了Verilog操作符。

操作符类型	符号
连接及复制操作符	{}　{{}}
算术操作符	* /　% + -
逻辑移位操作符	<<　>>
关系操作符	>　<　>=　<=
相等操作符	==　===　!=　!==
按位操作符	~　&　|　^　~^或^~
逻辑操作符	!　&&　||
条件操作符	？：

Verilog中的大小(size)与符号

- Verilog根据表达式中变量的长度对表达式的值自动地进行调整。
- Verilog自动截断或扩展赋值语句中右边的值以适应左边变量的长度。
- 当一个负数赋值给无符号变量如reg时，Verilog自动完成二进制补码计算

```
module sign_size;
    reg [3:0] a, b;
    reg [15:0] c;
  initial begin
        a = -1;          // a是无符号数，因此其值为1111
        b = 8; c= 8;  // b = c = 1000
    #10 b = b + a;    // 结果10111截断, b = 0111
    #10 c = c + a;    // c = 10111
  end
endmodule
```

算术操作符

+	加
-	减
*	乘
/	除
%	模

- 将负数赋值给reg或其它无符号变量使用2的补码算术。
- 如果操作数的某一位是x或z，则结果为x
- 在整数除法中，余数舍弃
- %运算中使用第一个操作数的符号

```
module arithops ();
    parameter five = 5;
    integer ans, int;
    reg [3: 0] rega, regb;
    reg [3: 0] num;
  initial begin
    rega = 3;
    regb = 4'b1010;
    int = -3;     //int = 1111……1111_1101
  end
  initial fork
    #10 ans = five * int;        // ans = -15
    #20 ans = (int + 5)/ 2;      // ans = 1
    #30 ans = five/ int;         // ans = -1
    #40 num = rega + regb;  // num = 1101
    #50 num = rega + 1;          // num = 0100
    #60 num = int;               // num = 1101
    #70 num = regb % rega; // num = 1
    #80 $finish;
join
endmodule
```

integer和reg类型在算术运算中，integer是有符号数，而reg是无符号数。

按位操作符

~	not
&	and
|	or
^	xor
~ ^	xnor
^ ~	xnor

- 按位操作符对矢量中相对应位运算。

regb = 4'b1 0 1 0

regc = 4'b1 x 1 0

num = regb & regc = 1 0 1 0 ;

- 位值为x时不一定产生x结果。

当两个操作数位数不同时，位数少的操作数零扩展到相同位数。

```
a = 4'b1011;
b = 8'b01010011;
c = a | b; // a零扩展为 8'b00001011
```

```
module bitwise ();
    reg [3: 0] rega, regb, regc;
    reg [3: 0] num;
  initial begin
    rega = 4'b1001;
    regb = 4'b1010;
    regc = 4'b11x0;
  end
  initial fork
    #10 num = rega & 0;       // num = 0000
    #20 num = rega & regb; // num = 1000
    #30 num = rega | regb;    // num = 1011
    #40 num = regb & regc; // num = 10x0
    #50 num = regb | regc;    // num = 1110
    #60 $finish;
  join
endmodule
```

逻辑操作符

!	not
&&	and
||	or

- 逻辑操作符的结果为一位1，0或x。
- 逻辑操作符只对逻辑值运算。
- 如操作数为全0，则其逻辑值为false
- 如操作数有一位为1，则其逻辑值为true
- 若操作数只包含0、x、z，则逻辑值为x

逻辑反操作符将操作数的逻辑值取反。例如，若操作数为全0，则其逻辑值为0，逻辑反操作值为1。

```
module logical ();
    parameter five = 5;
    reg ans;
    reg [3: 0] rega, regb, regc;
  initial
    begin
      rega = 4'b0011;     //逻辑值为“1”
      regb = 4'b10xz;     //逻辑值为“1”
      regc = 4'b0z0x;     //逻辑值为“x”
  end
  initial fork
    #10 ans = rega && 0;      // ans = 0
    #20 ans = rega || 0;      // ans = 1
    #30 ans = rega && five;   // ans = 1
    #40 ans = regb && rega;   // ans = 1
    #50 ans = regc || 0;      // ans = x
    #60 $finish;
  join
endmodule
```

逻辑反与位反的对比

! logical not 逻辑反

~ bit-wise not 位反

- 逻辑反的结果为一位1，0或x。
- 位反的结果与操作数的位数相同

逻辑反操作符将操作数的逻辑值取反。例如，若操作数为全0，则其逻辑值为0，逻辑反操作值为1。

```
module negation();
    reg [3: 0] rega, regb;
    reg [3: 0] bit;
    reg log;
  initial begin
    rega = 4'b1011;
    regb = 4'b0000;
  end
  initial fork
    #10 bit = ~rega; // num = 0100
    #20 bit = ~regb; // num = 1111
    #30 log = !rega; // num = 0
    #40 log = !regb; // num = 1
    #50 $finish;
  join
endmodule
```

移位操作符

>>	逻辑右移
<<	逻辑左移

- 移位操作符对其左边的操作数进行向左或向右的位移位操作。
- 第二个操作数（移位位数）是无符号数
- 若第二个操作数是x或z则结果为x

<< 将左边的操作数左移右边操作数指定的位数

>> 将左边的操作数右移右边操作数指定的位数

在赋值语句中，如果右边(RHS)的结果:

位宽大于左边，则把最高位截去

位宽小于左边，则零扩展

```
module shift ();
    reg [9: 0] num, num1;
    reg [7: 0] rega, regb;
  initial     rega = 8'b00001100;
  initial fork
    #10 num <= rega << 5 ;  // num = 01_1000_0000
    #10 regb  <= rega << 5 ;  // regb =      1000_0000
    #20 num <= rega >> 3;   // num = 00_0000_0001
    #20 regb  <= rega >> 3 ;  // regb =      0000_0001
    #30 num <= 10'b11_1111_0000;
    #40 rega <= num << 2;   //rega =       1100_0000
    #40  num1 <= num << 2;//num1=11_1100_0000
    #50 rega <= num >> 2;   //rega =       1111_1100
    #50  num1 <= num >> 2;//num1=00_1111_1100
    #60 $finish;
  join
endmodule
```

左移先补后移

右移先移后补

建议：表达式左右位数一致

关系操作符

>	大于
<	小于
>=	大于等于
<=	小于等于

- 其结果是1’b1、1’b0或1’bx。

```
module relationals ();
    reg [3: 0] rega, regb, regc;
    reg val;
  initial begin
    rega = 4'b0011;
    regb = 4'b1010;
    regc = 4'b0x10;
  end
  initial fork
    #10 val = regc > rega ;   // val = x
    #20 val = regb < rega ;   // val = 0
    #30 val = regb >= rega ; // val = 1
    #40 val = regb > regc ;   // val = 1
    #50 $finish;
  join
endmodule
```

rega和regc的关系取决于x

无论x为何值，regb>regc

相等操作符

= 赋值操作符，将等式右边表达式的值拷贝到左边。

注意逻辑等与case等的差别

== 逻辑等

==	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

```
a = 2'b1x;
b = 2'b1x;
if (a == b)
   $display(" a is equal to b");
else
   $display(" a is not equal to b");
```

2’b1x==2’b0x

值为0，因为不相等

2’b1x==2’b1x

值为x，因为可能不相等，也可能相等

=== case等

==	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

```
a = 2'b1x;
b = 2'b1x;
if (a === b)
   $display(" a is identical to b");
else
   $display(" a is not identical to b");
```

2’b1x===2’b0x

值为0，因为不相同

2’b1x==2’b1x

值为1，因为相同

Case等只能用于行为描述，不能用于RTL描述。

相等操作符

==	逻辑等
!=	逻辑不等

- 其结果是1'b1、1'b0或1'bx。
- 如果左边及右边为确定值并且相等，则结果为1。
- 如果左边及右边为确定值并且不相等，则结果为0。
- 如果左边及右边有值不能确定的位，但值确定的位相等，则结果为x。
- !=的结果与= =相反

值确定是指所有的位为0或1。不确定值是有值为x或z的位。

```
module equalities1();
    reg [3: 0] rega, regb, regc;
    reg val;
  initial begin
    rega = 4'b0011;
    regb = 4'b1010;
    regc = 4'b1x10;
  end
  initial fork
    #10 val = rega == regb ;  // val = 0
    #20 val = rega != regc;    // val = 1
    #30 val = regb != regc;    // val = x
    #40 val = regc == regc;   // val = x
    #50 $finish;
  join
endmodule
```

相等操作符

===	相同(case等)
!==	不相同(case不等)

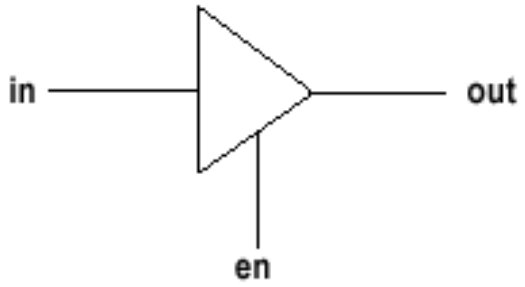
- 其结果是1'b1、1'b0或1'bx。
- 如果左边及右边的值相同（包括x、z），则结果为1。
- 如果左边及右边的值不相同，则结果为0。
- !==的结果与 === 相反

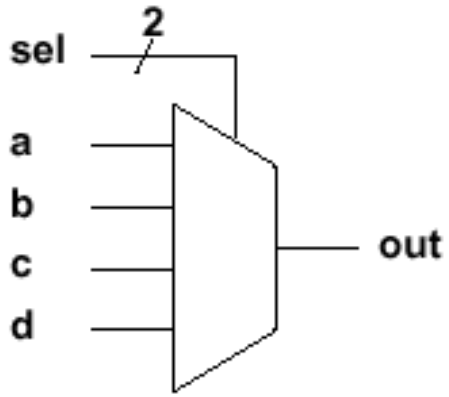
综合工具不支持

```
module equalities2();
      reg [3: 0] rega, regb,
regc;
      reg val;
   initial begin
      rega = 4'b0011;
      regb = 4'b1010;
      regc = 4'b1x10;
   end
   initial fork
      #10 val = rega === regb ;
// val = 0
      #20 val = rega !== regc;
// val = 1
      #30 val = regb === regc;
// val = 0
      #40 val = regc === regc;
// val = 1
      #50 $finish;
   join
endmodule
```

条件操作符

?: 条件






```
module likebufif( in, en, out);
      input in;
      input en;
      output out;
   assign out = (en == 1) ? in : 'bz;
endmodule

module like4to1( a, b, c, d, sel, out);
      input a, b, c, d;
      input [1: 0] sel;
      output out;
   assign out = sel == 2'b00 ? a :
                sel == 2'b01 ? b :
                sel == 2'b10 ? c : d;
endmodule
```

如果条件值为x或z，则结果可能为x或z

条件操作符

条件操作符的语法为：

<LHS> = <condition> ? <true_expression>:<false_expression>

其意思是：if condition is TRUE, then LHS=true_expression, else LHS = false_expression

每个条件操作符必须有三个参数，缺少任何一个都会产生错误。

最后一个操作数作为缺省值。

registger = condition ? true_value:false_value;

上式中，若condition为真则register等于true_value；若condition为假则register等于false_value。一个很有意思的地方是，如果条件值不确定，且true_value和false_value不相等，则输出不确定值。

例如：assign out = (sel == 0) ? a : b;

若sel为0则out =a；若sel为1则out = b。如果sel为x或z，若a = b =0，则out = 0；若a≠b，则out值不确定。

级联操作符

{ } 级联

可以从不同的矢量中选择位并用它们组成一个新的矢量。

用于位的重组和矢量构造

在级联和复制时，必须指定位数，否则将产生错误。

下面是类似错误的例子：

```
a[7:0] = {4{ ´b10}};
b[7:0] = {2{ 5}};
c[3:0] = {3´b011, ´b0};
```

级联时不限定操作数的数目。在操作符符号{ }中，用逗号将操作数分开。例如：

```
{A, B, C, D}
```

```
module concatenation;
    reg [7: 0] rega, regb, regc, regd;
    reg [7: 0] new;
  initial begin
    rega = 8'b0000_0011;
    regb = 8'b0000_0100;
    regc = 8'b0001_1000;
    regd = 8'b1110_0000;
  end
  initial fork
    #10 new = {regc[ 4: 3], regd[ 7: 5],
                regb[ 2], rega[ 1: 0]};
    // new = 8'b11111111
    #20 $finish;
  join
endmodule
```

复制

{ {} } 复制

复制一个变量或在{ }中的值

前两个{ 符号之间的正整数指定复制次数。

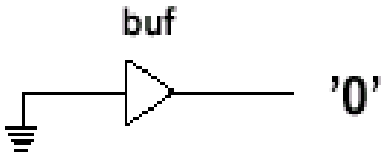
```
module replicate ();
    reg [3: 0] rega;
    reg [1: 0] regb, regc;
    reg [7: 0] bus;
  initial begin
    rega = 4'b1001;
    regb = 2'b11;
    regc = 2'b00;
  end
  initial fork
      #10 bus <= {4{ regb}}; // bus = 11111111
   // regb is replicated 4 times.
      #20 bus <= { {2{ regb}}, {2{ regc}} };
   // bus = 11110000. regc and regb are each
   // replicated, and the resulting vectors
   // are concatenated together
      #30 bus <= { {4{ rega[1]}}, rega };
   // bus = 00001001. rega is sign-extended
      #40 $finish;
  join
endmodule
```

第三节　Verilog的逻辑系统及数据类型

学习内容：

- 学习Verilog逻辑值系统
- 学习Verilog中不同类的数据类型
- 理解每种数据类型的用途及用法
- 数据类型说明的语法

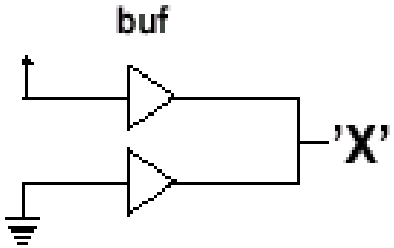
Verilog采用的四值逻辑系统



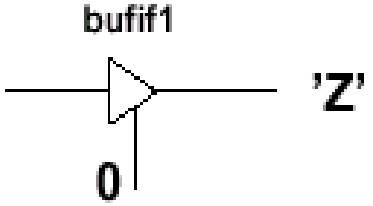
’0’, Low, False, Logic Low, Ground,VSS, Negative Assertion

‘1’, High, True, Logic High, Power,

VDD, VCC, Positive Assertion



’X’ Unknown: Occurs at Logical Which Cannot be Resolved Conflict



HiZ, High Impedance, Tri- Stated,

Disabled Driver (Unknown)

主要数据类型

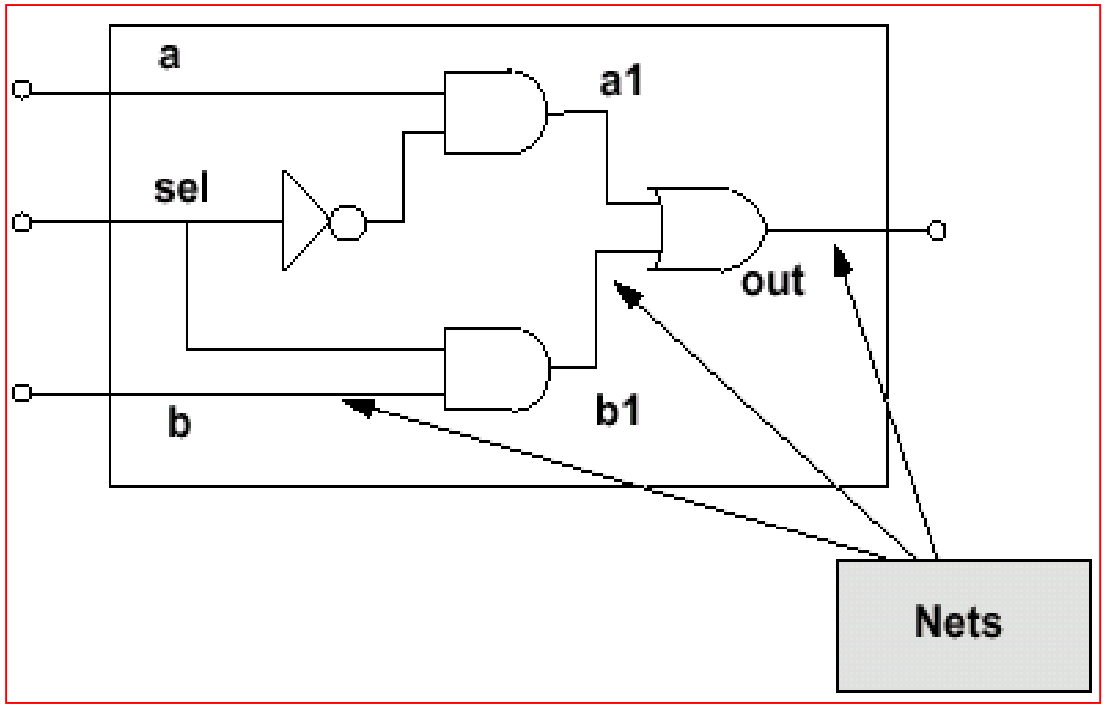
Verilog主要有三类(class)数据类型：

- **net （线网）：表示器件之间的物理连接**
- **register （寄存器）： 表示抽象存储元件**
- **parameters(参数)：运行时的常数(run-time constants)**

net（线网）

net需要被持续的驱动，驱动它的可以是门和模块。

当net驱动器的值发生变化时， Verilog自动的将新值传送到net上。在例子中，线网out由or门驱动。当or门的输入信号置位时将传输到线网net上。



net类的类型（线网）

- 有多种net类型用于设计(design-specific)建模和工艺(technology-specific)建模

net类型	功 能
wire, tri	标准内部连接线(缺省)
supply1, supply0	电源和地
wor, trior	多驱动源线或
wand, triand	多驱动源线与
trireg	能保存电荷的net
tri1, tri0	无驱动时上拉/下拉

- 没有声明的net的缺省类型为 1 位(标量)wire类型。

net类的类型（线网）

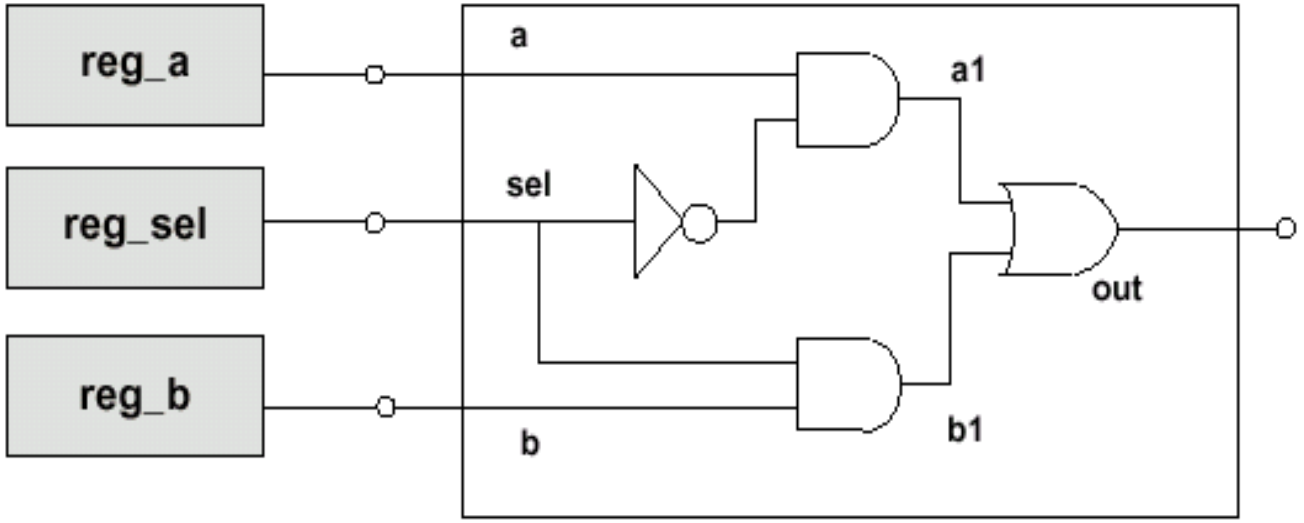
- 常用于组合电路描述。
- wire类型是最常用的类型，只有连接功能。
- wire和tri类型有相同的功能。用户可根据需要将线网定义为wire或tri以提高可读性。tri型的信号综合后具有三态的功能。
- wand、wor有线逻辑功能。
- trireg类型很象wire类型，但trireg类型在没有驱动时保持以前的值。这个值的强度随时间减弱。
- 修改net缺省类型的编译指导：

  `default_nettype <nettype>

  nettype不能是supply1和supply0。

寄存器类 （register)

- 寄存器类型在赋新值以前保持原值
- 寄存器类型大量应用于行为模型描述及激励描述。在下面的例子中，reg_a、reg_b、reg_sel用于施加激励给2:1多路器。
- 用行为描述结构给寄存器类型赋值。给reg类型赋值是在过程块中。



寄存器类的类型

- **寄存器类有四种数据类型**

寄存器类型	功能
reg	可定义的无符号整数变量，可以是标量(1位)或矢量，是最常用的寄存器类型
integer	32位有符号整数变量，**算术操作产生二进制补码形式的结果。**通常用作不会由硬件实现的的数据处理。
real	双精度的带符号浮点变量，用法与integer相同。
time	64位无符号整数变量，用于仿真时间的保存与处理
realtime	与real内容一致，但可以用作实数仿真时间的保存与处理

Verilog中net和register声明语法

- **net声明**

  <net_type> [range] [delay] <net_name>[, net_name];

  net_type： net类型

  range: 矢量范围，以[MSB：LSB]格式

  delay： 定义与net相关的延时

  net_name: net名称，一次可定义多个net, 用逗号分开。

- **寄存器声明**

  <reg_type> [range] <reg_name>[, reg_name];

  reg_type：寄存器类型

  range： 矢量范围，以[MSB：LSB]格式。只对reg类型有效

  reg_name ： 寄存器名称，一次可定义多个寄存器，用逗号分开

Verilog中net和register声明语法

- **举例：**

  reg a;　　//一个标量寄存器

  wand w; // 一个标量wand类型net

  reg [3: 0] v; // 从MSB到LSB的4位寄存器向量

  reg [7: 0] m, n; // 两个8位寄存器

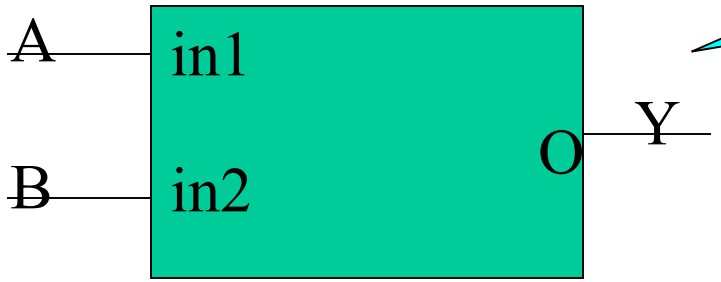
  tri [15: 0] busa; // 16位三态总线

  wire [0: 31] w1, w2; // 两个32位wire，MSB为bit0

选择正确的数据类型

输入端口可以由net/register驱动，但输入端口只能是net

输出端口可以是net/register类型，输出端口只能驱动net



双向端口输入/输出只能是net类型

```
module top;
wire y;
reg a, b;
    DUT u1 (y, a, b) ;
    initial   begin
            a = 0; b = 0;
        #5 a = 1;
     end
endmodule
```

```
module DUT (Y, A, B);
output Y;
input A, B;
wire Y, A, B;
    and (Y, A, B) ;
endmodule
```

若Y，A，B说明为reg则会产生错误。

在过程块中只能给register类型赋值

选择数据类型时常犯的错误

信号类型确定方法总结如下：

- 信号可以分为端口信号和内部信号。出现在端口列表中的信号是端口信号，其它的信号为内部信号。
- 对于端口信号，输入端口只能是net类型。输出端口可以是net类型，也可以是register类型。若输出端口在过程块中赋值则为register类型；若在过程块外赋值(包括实例化语句），则为net类型。
- 内部信号类型与输出端口相同，可以是net或register类型。判断方法也与输出端口相同。若在过程块中赋值，则为register类型；若在过程块外赋值，则为net类型。

下面所列是常出的错误及相应的错误信息(error message)

- 用过程语句给一个net类型的或忘记声明类型的信号赋值。
  信息：illegal …… assignment.
- 将实例的输出连接到声明为register类型的信号上。
  信息：<name> has illegal output port specification.
- 将模块的输入信号声明为register类型。
  信息：incompatible declaration, <signal name> ……

选择数据类型时常犯的错误举例

example.v

修改前：

```
module example(o1, o2, a, b, c, d);
    input a, b, c, d;
    output o1, o2;
    reg c, d;
    reg o2
    and u1(o2, c, d);
    always @(a or b)
        if (a) o1 = b; else o1 = 0;
endmodule
```

修改后：

```
module example(o1, o2, a, b, c, d);
    input a, b, c, d;
    output o1, o2;
//    reg c, d;
//    reg o2
    reg o1;
    and u1(o2, c, d);
    always @(a or b)
        if (a) o1 = b; else o1 = 0;
endmodule
```

选择数据类型时常犯的错误举例

```
Compiling source file "example.v"
Error!  Incompatible declaration, (c) defined as input
        at line 2                                  [Verilog-IDDIL]
        "example.v", 5:
Error!  Incompatible declaration, (d) defined as input
        at line 2                                  [Verilog-IDDIL]
        "example.v", 5:
Error!  Gate (u1) has illegal output specification      [Verilog-GHIOS]
        "example.v", 8:
3 errors
```

第一次编译信息

verilog –c example.v

```
Compiling source file "example.v"
Error!  Illegal left-hand-side assignment               [Verilog-ILHSA]
        "example.v", 11: o1 = b;
Error!  Illegal left-hand-side assignment               [Verilog-ILHSA]
        "example.v", 12: o1 = 0;
2 errors
```

第二次编译信息

参数（parameters)

- 用参数声明一个可变常量，常用于定义延时及宽度变量。
- 参数定义的语法：parameter <list_of_assignment>;
- 可一次定义多个参数，用逗号隔开。
- 在使用文字(literal)的地方都可以使用参数。
- 参数的定义是局部的，只在当前模块中有效。
- 参数定义可使用以前定义的整数和实数参数。

```
module mod1( out, in1, in2);
  . . .
parameter cycle = 20, prop_ del = 3,
          setup = cycle/2 - prop_del,
          p1 = 8,
          x_ word = 16'bx,
           . . .
   wire [p1: 0] w1; // A wire declaration using parameter
    . . .
endmodule
```

参数重载（overriding)

模块实例化时参数重载

```
module mod1( out, in1, in2);
  . . .
  parameter  p1 = 8,
             real_constant = 2.039,
             x_word = 16'bx,
             file = "/usr1/jdough/design/mem_file.dat";
. . .
endmodule
module top;
  . . .
   mod1 #( 5, 3.0, 16'bx, "../ my_mem. dat")  I1( out, in1, in2);
  . . .
endmodule
```

次序与原说明相同

使用#

因为#说明延时的时候只能用于gate或过程语句，不能用于模块实例。

gate（primitives)在实例化时只能有延时，不能有模块参数。

为什么编译器认为这是参数而不是延时呢？

寄存器数组(Register Arrays)

- 在Verilog中可以说明一个寄存器数组。

  integer NUMS [7: 0]; // 包含8个整数数组变量

  time t_vals [3: 0];   // 4个时间数组变量

- reg类型的数组通常用于描述存储器

  其语法为： reg [MSB:LSB] <memory_name> [first_addr:last_addr];

  [MSB:LSB]定义存储器字的位数

  [first_addr:last_addr]定义存储器的深度

  例如：

  reg [15: 0] MEM [0:1023]; // 1K x 16存储器

  reg [7: 0] PREP [‘hFFFE: ’hFFFF]; // 2 x 8存储器

- 描述存储器时可以使用参数或任何合法表达式

  parameter wordsize = 16;

  parameter memsize = 1024;

  reg [wordsize-1: 0] MEM3 [memsize-1: 0];

第四节 结构描述(structural modeling)

学习内容：

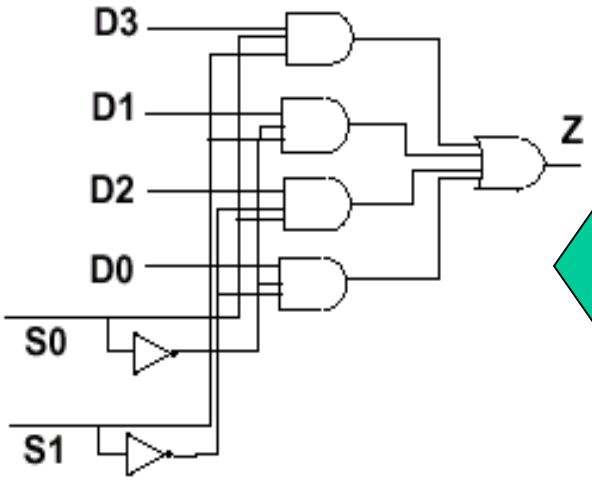
- 结构描述？
- 如何使用Verilog的基本单元(primitives)
- 如何构造层次化设计
- 了解Verilog的逻辑强度系统

结构描述

- 用门来描述器件的功能
- 基于基本元件和底层模块实例化语句
- 最接近实际的硬件结构
- 主要使用元件的定义、使用声明以及元件实例化来构建系统
- **primitives(基本单元) : Verilog语言已定义的具有简单逻辑功能的功能模型(models)**

结构描述

- Verilog结构描述表示一个逻辑图
- 结构描述用已有的元件构造。

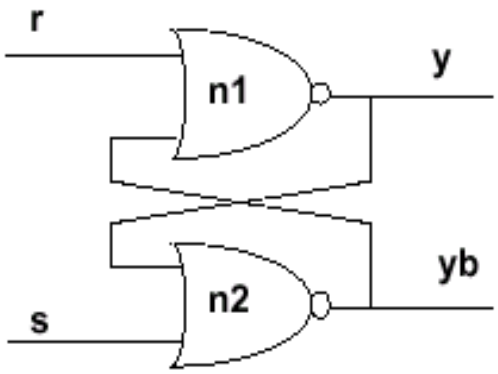


结构描述等价于逻辑图，都是连接简单元件构成更复杂元件

```
module MUX4x1( Z, D0, D1, D2, D3, S0, S1);
   output Z;
   input D0, D1, D2, D3, S0, S1;
     and (T0, D0, S0_, S1_),
         (T1, D1, S0_, S1),
         (T2, D2, S0, S1_),
         (T3, D3, S0, S1);
     not (S0_, S0), (S1_, S1);
     or (Z, T0, T1, T2, T3);
endmodule
```

同一种门可以通过一个语句实例化

忽略了门的实例名。



Latch

```
module rs_latch (y, yb, r, s);
      output y, yb;
      input r, s;
      nor n1( y, r, yb);
      nor n2( yb, s, y);
endmodule
```

通过门的实例使用门

结构描述（续）

- 结构描述等价于逻辑图。它们都是连接简单元件来构成更为复杂的元件。Verilog使用其连接特性完成简单元件的连接。
- 在描述中使用元件时，通过建立这些元件的实例来完成。
- 上面的例子中MUX是没有反馈的组合电路，使用中间或内部信号将门连接起来。描述中忽略了门的实例名，并且同一种门的所有实例可以在一个语句中实例化。
- 上面的锁存器(latch)是一个时序元件，其输出反馈到输入上。它没有使用任何内部信号。它使用了实例名并且对两个*nor*门使用了分开的实例化语句。

Verilog基本单元（primitives)

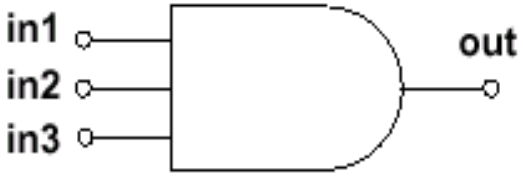
- Verilog基本单元提供基本的逻辑功能，也就是说这些逻辑功能是预定义的，用户不需要再定义这些基本功能。
- 基本单元是Verilog开发库的一部分。大多数ASIC和FPGA元件库是用这些基本单元开发的。基本单元库是自下而上的设计方法的一部分。

基本单元名称	功能
and	**Logical And**
or	**Logical Or**
not	**Inverter**
buf	**Buffer**
xor	**Logical Exclusive Or**
nand	**Logical And Inverted**
nor	**Logical Or Inverted**
xnor	**Logical Exclusive Or Inverted**

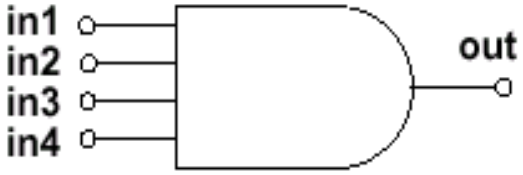
基本单元的引脚 (pin)的可扩展性

- 基本单元引脚的数目由连接到门上的net的数量决定。因此当基本单元输入或输出的数量变化时用户不需要重定义一个新的逻辑功能。
- 所有门（除了not和buf）可以有多个输入，但只能有一个输出。
- not和buf门可以有多个输出，但只能有一个输入。


```
and (out, in1, in2);
```




```
and (out, in1, in2, in3);
```




```
and (out, in1, in2, in3, in4);
```

带条件的基本单元

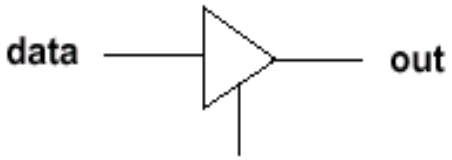
- Verilog有四种不同类型的条件基本单元
- 这四种基本单元只能有三个引脚：output, input, enable
- 这些单元由enable引脚使能。
  - 当条件基本单元使能信号无效时，输出高阻态。

基本单元名称	功能
bufif1	条件缓冲器，逻辑 1 使能
bufif0	条件缓冲器，逻辑 0 使能
notif1	条件反相器，逻辑 1 使能
notif0	条件反相器，逻辑 0 使能

带条件的基本单元（续）

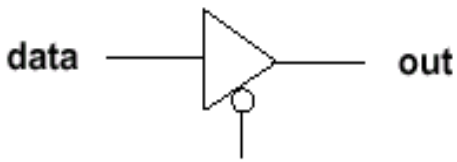
- 条件基本单元有三个端口：输出、数据输入、使能输入

bufif1



bufif1 (out, data, enable)

bufif0



bufif0 (out, data, enable)

	bufif1	enable 0	1	x	z
data	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

	bufif0	enable 0	1	x	z
data	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x

基本单元实例化

- 在端口列表中，先说明输出端口，然后是输入端口
- 实例化时实例的名字是可选项

```
and      (out, in1, in2, in3, in4); // unnamed instance
buf  b1 (out1, out2, in);           // named instance
```

- 延时说明是可选项。所说明的延时是固有延时。输出信号经过所说明的延时才变化。没有说明时延时为0。

```
notif0  #3.1  n1  (out, in, cntrl); // delay specified
```

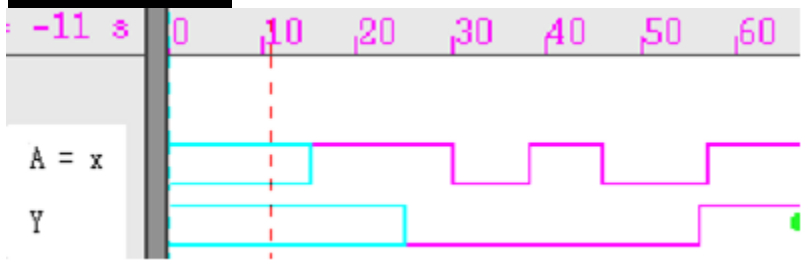
- 信号强度说明是可选项

```
not (strong1, weak0) n1 (inv, bit); // strength specified
```

固有延时

```
module intr_sample;
   reg A;    wire Y;
   not  #10  intrinsic (Y, A);
initial  begin
   #15  A = 1;    #15  A = 0;    #8   A = 1;
   #8   A = 0;    #11  A = 1;    #10 $finish;
end
endmodule
```

仿真波形



模块实例化(module instantiation)

- 模块实例化时实例必须有一个名字。
- 使用位置映射时，端口次序与模块的说明相同。
- 使用名称映射时，端口次序与位置无关
- 没有连接的输入端口初始化值为x。

```
module comp (o1, o2, i1, i2);
    output   o1, o2;
    input   i1, i2;
    . . .
endmodule

module test;
    comp c1 (Q, R, J, K); // Positional mapping
    comp c2 (.i2(K),  .o1(Q),  .o2(R),  .i1(J)); // Named mapping
    comp c3 (Q,   ,  J,  K);       // One port left unconnected
    comp c4 (.i1(J),  .o1(Q));   // Named, two unconnected ports
endmodule
```

没有连接时通常会产生警告

名称映射的语法：

.内部信号（外部信号）

实例数组(Array of Instances)

- 实例名字后有范围说明时会创建一个实例数组。在说明实例数组时，实例必须有一个名字 (包括基本单元实例)。其说明语法为：

  <模块名字> <实例名字> <范围> (<端口>);

```
module driver (in, out, en);
    input [2: 0] in;
    output [2: 0] out;
    input en;
    bufif0  u[2:0]  (out, in, en);   // array of buffers
endmodule
```

范围说明语法：
[MSB : LSB]

```
module driver_equiv (in, out, en);
    input [2: 0] in;
    output [2: 0] out;
    input en;
  // Each primitive instantiation is done separately
    bufif0 u2 (out[2], in[2], en);
    bufif0 u1 (out[1], in[1], en);
    bufif0 u0 (out[0], in[0], en);
endmodule
```

两个模块功能完全等价

实例数组(Array of Instances)(续)

- 如果范围中MSB与LSB相同，则只产生一个实例。
- 一个实例名字只能有一个范围。
- 下面以模块comp为例说明这些情况


```
module oops;
    wire y1, a1, b1;
    wire [3: 0] a2, b2, y2, a3, b3, y3;

    comp u1  [5: 5] (y1, a1, b1); // 只产生一个comp实例
    comp m1 [0: 3] (y2, a2, b2);
    comp m1 [4: 7] (y3, a3, b3); //  非法
endmodule
```

m1作为实例阵列名字使用了两次

逻辑强度(strength)模型

- Verilog提供多级逻辑强度。
- 逻辑强度模型决定信号组合值是可知还是未知的，以更精确的描述硬件的行为。
- 下面这些情况是常见的需要信号强度才能精确建模的例子。
  - 开极输出(Open collector output)(需要上拉）
  - 多个三态驱动器驱动一个信号
  - MOS充电存储
  - ECL门（emitter dotting)
- 逻辑强度是Verilog模型的一个重要部分。通常用于元件建模，如ASIC和FPGA库开发工程师才使用这么详细的强度级。但电路设计工程师使用这些精细的模型仿真也应该对此了解。

逻辑强度(strength)模型（续）

- 用户可以给基本单元实例或net定义强度。
- 基本单元强度说明语法：

  <基本单元名> <强度> <延时> <实例名> （<端口>）；

  例：nand (strong1, pull0) #(2: 3: 4) n1 (o, a, b); // strength and delay

  or (supply0, highz1) (out, in1, in2, in3); // no instance name

- 用户可以用%v格式符显示net的强度值

  $monitor ($ time,," output = %v", f);

- 电容强度(large, medium, small)只能用于net类型trireg和基本单元tran

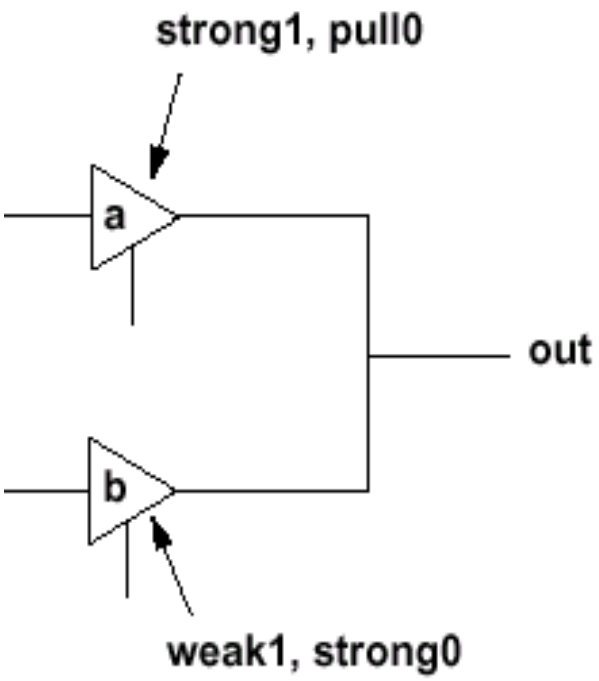
  例如：trireg (small) tl;

信号强度值系统

	Level	Type	%v formats		Specification
Supply	7	Drive	Su0	Su1	supply0, supply1
Strong	6	Drive(default)	St0	St1	strong0, strong1
Pull	5	Drive	Pu0	Pu1	pull0, pull1
Large	4	Capacitive	La0	La1	large
Weak	3	Drive	We0	We1	weak0, weak1
Medium	2	Capacitive	Me0	Me1	medium
Small	1	Capacitive	Sm0	Sm1	small
High Z	0	Impedance	Hi0	Hi1	highz0, highz1

Verilog多种强度决断

- 在Verilog中，级别高的强度覆盖级别低的强度



a's output	b's output	out
strong1	strong0	strongX
pull0	weak1	pull0
pull0	strong0	strong0
strong1	weak1	strong1
pull0	HiZ	pull0
HiZ	weak1	weak1
HiZ	HiZ	HiZ

第五节　数据流级描述

学习内容：

- 如何使用连续赋值进行数据流级描述

注：

数据流级描述是抽象层次描述的一种。它从数据流动的角度来描述整个电路，即数据的传输和变化情况。体现在描述语句中，重点是在整个电路从输入到输出的过程中，输入信号经过哪些处理或者运算，最终才能得到最后的输出信号。

持续赋值(continuous assignment)

- 可以用持续赋值语句描述组合逻辑，代替用门及其连接描述方式。
- 持续赋值在过程块外部使用。
- 持续赋值用于net驱动。
- 持续赋值只能在等式左边有一个简单延时说明。
  - 只限于在表达式左边用#delay形式
- 持续赋值可以是显式或隐含的。

语法：

<assign> [#delay] [strength] <net_name> = <expressions>;

```
wire out;
assign out = a & b; // 显式
wire inv = ~in; // 隐含
```

持续赋值(continuous assignment)(续)

持续赋值的例子

```
module assigns (o1, o2, eq, AND, OR, even, odd, one, SUM, COUT,
                a, b, in, sel, A, B, CIN);
    output [7:0] o1, o2;
    output [31:0] SUM;
    output eq, AND, OR, even, odd, one, COUT;
    input a, b, CIN;
    input [1:0] sel;
    input [7:0] in;
    input [31:0] A, B;
    wire [7:0] #3 o2;              // 没有赋值，但设置了延时
    tri  AND = a& b, OR = a| b; // 两个隐含赋值
    wire #10 eq = (a == b);        // 隐含赋值，并说明了延时
    assign o1[7:4] = in[3:0], o1[3:0] = in[7:4]; // 部分选择
    tri #5 even = ^in, odd = ~^ in;    // 延时，两个赋值
    wire one = 1'b1;    // 常数赋值
    assign {COUT, SUM} = A + B + CIN ; // 给级联赋值
endmodule
```

持续赋值(continuous assignment)(续)

in的值赋给o1，但其每位赋值的强度及延迟可能不同。如果o1是一个标量（scalar）信号，则其延迟和前面的条件缓冲器上的门延迟相同。对向量线网（net）的赋值上的延迟情况不同。0赋值使用下降延迟，Z赋值使用关断延迟，所有其他赋值使用上升延迟。

上面的例子显示出持续赋值的灵活性和简单性。持续赋值可以：

- 隐含或显式赋值
- 给任何net类型赋值
- 给矢量net的位或部分赋值
- 设置延时
- 设置强度
- 用级联同时给几个net类变量赋值
- 使用条件操作符
- 使用用户定义的函数的返回值
- 可以是任意表达式，包括常数表达式

持续赋值(continuous assignment)(续)

使用条件操作符的例子：

```
module cond_assigns (MUX1, MUX2, a, b, c, d);
    output MUX1, MUX2;
    input a, b, c, d;
    assign MUX1 = sel == 2'b00 ? a :
                  sel == 2'b01 ? b :
                  sel == 2'b10 ? c : d;
    tri1 MUX2 = sel == 0 ? a : 'bz,  MUX2 = sel == 1 ? b : 'bz,
        MUX2 = sel == 2 ? c : 'bz,   MUX2 = sel == 3 ? d : 'bz;
endmodule
```

- 从上面的例子可以看出，持续赋值的功能很强。可以使用条件操作符，也可以对一个net多重赋值(驱动)。
- 在任何时间里只有一个赋值驱动MUX2到一个非三态值。如果所有驱动都为三态，则mux2缺省为一个上拉强度的1值。

第六节　行为描述

学习内容：

- 行为建模的基本概念
- Verilog中高级编程语言结构

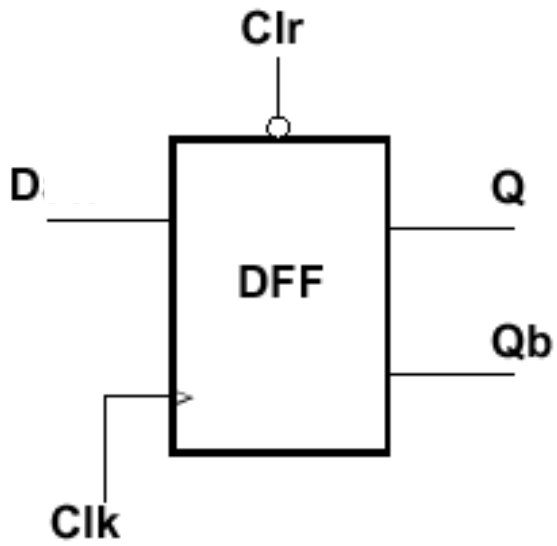
注：

RTL描述方式是行为描述方式的子集，通常是指可综合的行为描述。在本节中的综合部分将详细介绍哪些行为级结构同样可以用于RTL描述。

行为描述

- 行为级描述是对系统的高抽象级描述。在这个级别，表达的是输入和输出之间转换的行为，不包含任何结构信息。
- Verilog有高级编程语言结构用于行为描述，包括：

  wait, while, if then, case和forever
- Verilog的行为建模是用一系列以高级编程语言编写的并行的、动态的过程块来描述系统的工作。



DFF

在每一个时钟上升沿，

若Clr不是低电平，

置Q为D值，

置Qb为D值的反

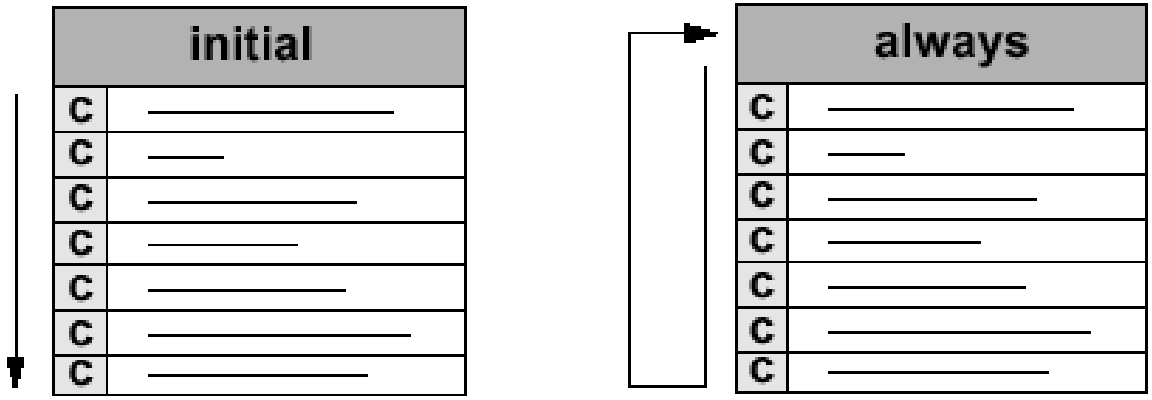
无论何时Clr变低

置Q为0，

置Qb为1

过程(procedural)块

- 过程块是行为模型的基础。
- 过程块有两种：
  - initial块，只能执行一次
  - always块，循环执行
- 过程块中有下列部件
  - 过程赋值语句：描述过程块中的数据流
  - 高级结构（循环，条件语句）：描述块的功能
  - 时序控制：控制块的执行及块中的语句。



过程赋值(procedural assignment)

- 在过程块中的赋值称为过程赋值。
- 在过程赋值语句中表达式左边的信号必须是寄存器类型（如reg类型）
- 在过程赋值语句等式右边可以是任何有效的表达式，数据类型也没有限制。
- 如果一个信号没有声明则缺省为wire类型。使用过程赋值语句给wire赋值会产生错误。

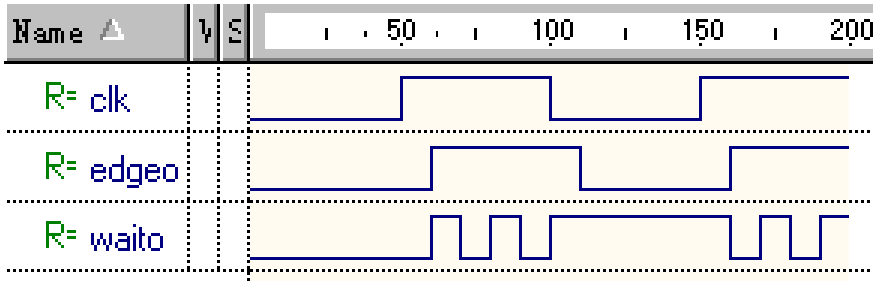
```
module adder (out, a, b, cin);
    input a, b, cin;
    output [1:0] out;
    wire a, b, cin;
    reg half_sum;
    reg [1: 0] out;
  always @( a or b or cin)
  begin
    half_sum = a ^ b ^ cin ; // OK
    half_carry = a & b | a & !b & cin | !a & b & cin ; // ERROR!
    out = {half_carry, half_ sum} ;
  end
endmodule
```

half_carry没有声明

过程时序控制

在过程块中可以说明过程时序。过程时序控制有三类：

- 简单延时(#delay)：延迟指定时间步后执行
- 边沿敏感的时序控制：@(<signal>)
  - 在信号发生翻转后执行。
  - 可以说明信号有效沿是上升沿(posedge)还是下降沿(negedge)。
  - 可以用关键字or指定多个参数。
- 电平敏感的时序控制：wait(<expr>)
  - 直至expr值为真时（非零）才执行。
  - 若expr已经为真则立即执行。




```
module wait_test;
reg clk, waito, edgeo;
initial begin clk = 0;edgeo=0;waito=0;end
always #10 clk = ~clk;
always @(clk) #2 edgeo = ~edgeo;
always wait(clk) #2 waito = ~waito;
endmodule
```

简单延时

在test bench中使用简单延时（#延时）施加激励，或在行为模型中模拟实际延时。

```
module muxtwo (out, a, b, sl);
    input a, b, sl;
    output out;
    reg out;
  always @( sl or a or b)
    if (! sl)
        #10 out = a;
// 从a到out延时10个时间单位
    else
        #12 out = b;
//从b到out延时12个时间单位
endmodule
```

在简单延时中可以使用模块参数parameter:

```
module clock_gen (clk);
    output clk;
    reg clk;
    parameter cycle = 20;
    initial clk = 0;
    always
        #(cycle/2) clk = ~clk;
endmodule
```

边沿敏感时序

时序控制@可以用在RTL级或行为级组合逻辑或时序逻辑描述中。可以用关键字*posedge*和*negedge*限定信号敏感边沿。敏感表中可以有多个信号，用关键字*or*连接。

```
module reg_ adder (out, a, b, clk);
    input clk;
    input [2: 0] a, b;
    output [3: 0] out;
    reg [3: 0] out;
    reg [3: 0] sum;
  always @( a or b) // 若a或b发生任何变化，执行
    #5 sum = a + b;
  always @( negedge clk) // 在clk下降沿执行
        out = sum;
endmodule
```

注：事件控制符or和位或操作符|及逻辑或操作符||没有任何关系。

wait语句

wait用于行为级代码中电平敏感的时序控制。

下面 的输出锁存的加法器的行为描述中，使用了用关键字or的边沿敏感时序以及用wait语句描述的电平敏感时序。

```
module latch_adder (out, a, b, enable);
    input enable;
    input [2: 0] a, b;
    output [3: 0] out;
    reg [3: 0] out;
  always @( a or b)
   begin
        wait (!enable) // 当enable为低电平时执行加法
        out = a + b;
   end
endmodule
```

注：综合工具还不支持wait语句。

命名事件(named event)

在行为代码中定义一个命名事件可以触发一个活动。命名事件不可综合。

```
module add_mult (out, a, b);
  input [2: 0]  a, b;
  output [3: 0] out;
  reg [3: 0] out;
 //***define events***
  event add, mult;
  always@ (a or b)
    if (a> b)
      -> add; // *** trigger event ***
    else
      -> mult; // *** trigger event ***
// *** respond to an   event trigger ***
  always @( add)
    out = a + b;
// *** respond to an event trigger ***
  always @( mult)
     out = a * b;
endmodule
```

在例子中，事件add和mult不是端口，但定义为事件，它们没有对应的硬件实现。

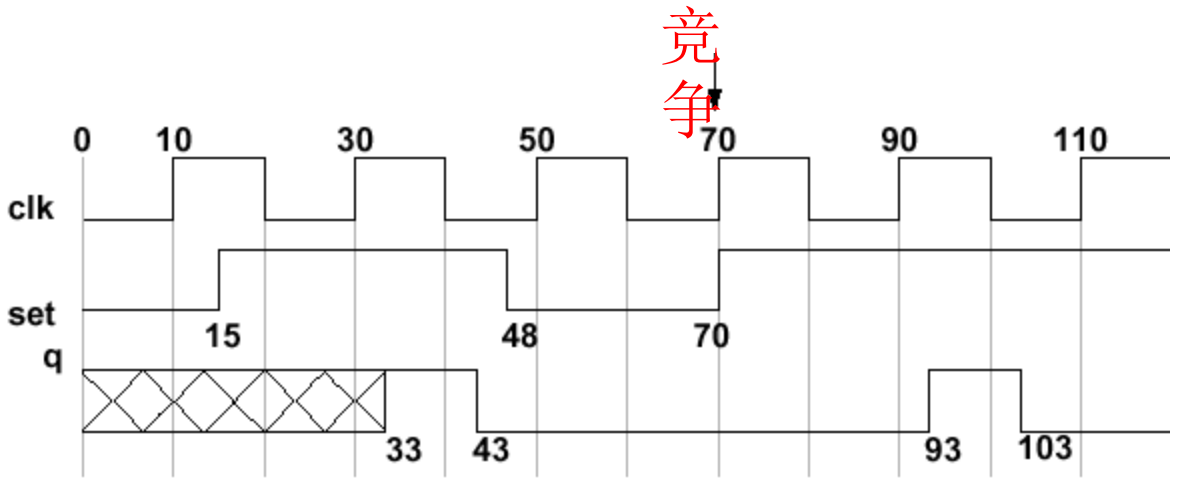
- 是一种数据类型，能在过程块中触发一个使能。
- 在引用前必须声明
- 没有持续时间，也不具有任何值
- 只能在过程块中触发一个事件。
- ->操作符用来触发命名事件。

a大于b，事件add被触发，控制传递到等待add的always块。

如果a小于或等于b，事件mult被触发，控制被传送到等待mult的always块。

行为描述举例

```
always wait (set)
begin
   @( posedge clk) #3  q = 1;
                       #10 q = 0;
   wait (! set);
end
```



在上面的例子中发生下面顺序的事件：

1. 等待set=1，忽略时刻10的clk的posedge。
2. 等待下一个clk的posedge，它将在时刻30发生。
3. 等待3个时间单位，在时刻33（30+3）置q=1。
4. 等待10个时间单位，在时刻43（33+10）置q=0。
5. 等待在时刻48发生的set=0。
6. 等待在时刻70发生且与clk的上升沿同时发生的set=1。
7. 等待下一个上升沿。时刻70的边沿被忽略，因为到达该语句时时间已经过去了，如例子所示，clk=1。

重要内容：在实际硬件设计中，事件6应该被视为一个竞争（race condition）。在仿真过程中，值的确定倚赖于顺序，所以是不可预测的。这是不推荐的建模类型。

RTL描述举例

下面的RTL例子中只使用单个边沿敏感时序控制。

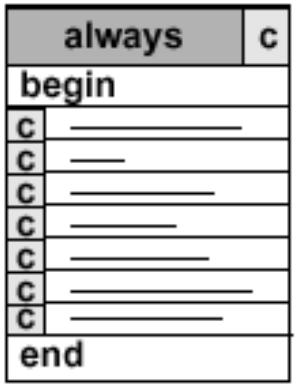
```
module dff (q, qb, d, clk);
    output q, qb;
    input d, clk;
    reg q, qb;
  always @( posedge clk)
  begin
     q = d;
    qb = ~d;
  end
endmodule
```

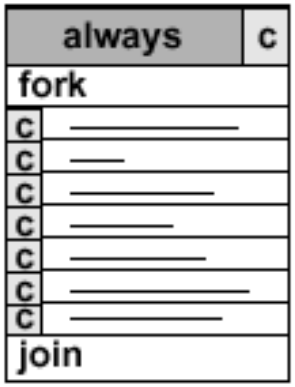
块语句

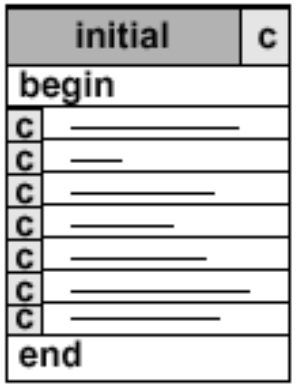
块语句用来将多个语句组织在一起，使得他们在语法上如同一个语句。

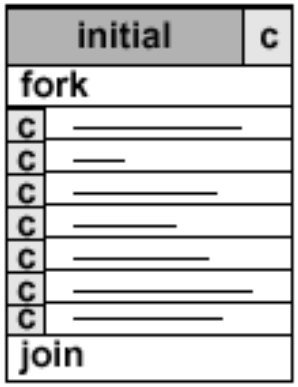
块语句分为两类：

- 顺序块：语句置于关键字begin和end之间，块中的语句以顺序方式执行。
- 并行块：关键字fork和join之间的是并行块语句，块中的语句并行执行。









- Fork和join语句常用于test bench描述。这是因为可以一起给出矢量及其绝对时间，而不必描述所有先前事件的时间。

块语句（续）

- 在顺序块中，语句一条接一条地计算执行。
- 在并行块中，所有语句在各自的延迟之后立即计算执行。

```
begin
    #5 a = 3;
    #5 a = 5;
    #5 a = 4;
end
```

```
fork
    #5 a = 3;
    #15 a = 4;
    #10 a = 5;
join
```

上面的两个例子在功能上是等价的。Fork-join例子里的赋值故意打乱顺序是为了强调顺序是没有关系的。

注意fork-join块是典型的不可综合语句，并且在一些仿真器时效率较差。

延迟赋值语句

语法： **LHS = <timing_ control> RHS;**

时序控制延迟的是赋值而不是右边表达式的计算。

在延迟赋值语句中**RHS**表达式的值都有一个隐含的临时存储。

可以用来简单精确地模拟寄存器交换和移位。

```
begin
  temp= b;
  @(posedge clk) a = temp;
end
```

LHS: Left-hand-side

RHS: Right-hand-side

延迟赋值语句

并行语句在同一时间步发生，但由仿真器在另外一个时间执行。

在下面的每个例子中，a和b的值什么时候被采样？

在下面的每个例子中，什么时候给a和b赋值？

b值拷贝到a然后回传

```
begin
    a = #5 b;
    b = #5 a;
    #10 $diplay(a, b);
end
```

a和b值安全交换

```
fork
    a = #5 b;
    b = #5 a;
    #10 $diplay(a, b);
join
```

在左边的例子中，b的值被立即采样（时刻0），这个值在时刻5赋给a。a的值在时刻5被采样，这个值在时刻10赋给b。

注意，另一个过程块可能在时刻0到时刻5之间影响b的值，或在时刻5到时刻10之间影响a的值。

在右边的例子中，b和a的值被立即采样（时刻0），保存的值在时刻5被赋值给他们各自的目标。这是一个安全传输。

注意，另一个过程块可以在时刻0到时刻5之间影响a和b的值。

非阻塞过程赋值

```
module swap_vals;
    reg a, b, clk;
  initial begin
    a = 0; b = 1; clk = 0;
  end
  always #5 clk = ~clk;
  always @( posedge clk)
    begin
      a <= b; // 非阻塞过程赋值
      b <= a; // 交换a和b值
  end
endmodule
```

过程赋值有两类

阻塞过程赋值

非阻塞过程赋值

阻塞过程赋值执行完成后再执行在顺序块内下一条语句。

非阻塞赋值不阻塞过程流，仿真器读入一条赋值语句并对它进行调度之后，就可以处理下一条赋值语句。

若过程块中的所有赋值都是非阻塞的，赋值按两步进行：

1. 仿真器计算所有RHS表达式的值，保存结果，并进行调度，在时序控制指定时间的赋值。
2. 在经过相应的延迟后，仿真器通过将保存的值赋给LHS表达式完成赋值。

非阻塞过程赋值（续）

阻塞与非阻塞赋值语句行为差别举例1

```
module non_block1;
    reg a, b, c, d, e, f;
  initial begin // blocking assignments
    a = #10 1; // time 10
    b = #2   0; // time 12
    c = #4   1; // time 16
  end
  initial begin // non- blocking assignments
    d <= #10 1; // time 10
    e <= #2   0; // time 2
    f <= #4   1; // time 4
  end
  initial begin
    $monitor($ time,," a= %b b= %b c= %b d= %b e= %b f= %b", a, b, c, d, e, f);
   #100 $finish;
  end
endmodule
```

输出结果：

0 a= x b= x c= x d= x e= x f = x

2 a= x b= x c= x d= x e= 0 f = x

4 a= x b= x c= x d= x e= 0 f = 1

10 a= 1 b= x c= x d= 1 e= 0 f = 1

12 a= 1 b= 0 c= x d= 1 e= 0 f = 1

16 a= 1 b= 0 c= 1 d= 1 e= 0 f = 1

非阻塞过程赋值（续）

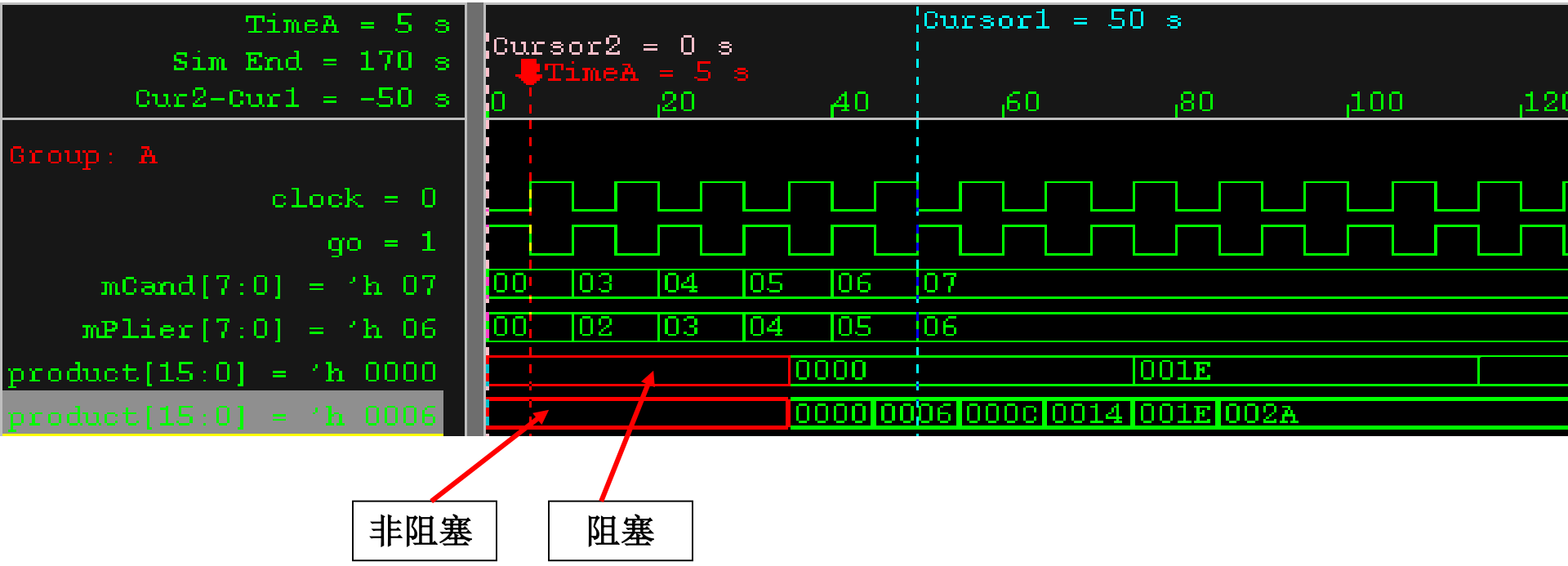
阻塞与非阻塞赋值语句行为差别举例2

```
module pipeMult(product, mPlier, mCand, go, clock);
input          go, clock;
input [7:0]    mPlier, mCand;
output [15:0]  product;
reg [15:0] product;
always @(posedge go)
  product = repeat (4) @(posedge clock) mPlier * mCand;
endmodule
```

```
module pipeMult(product, mPlier, mCand, go, clock);
input          go, clock;
input [7:0]    mPlier, mCand;
output [15:0]  product;
reg [15:0] product;
always @(posedge go)
  product <= repeat (4) @(posedge clock) mPlier * mCand;
endmodule
```

非阻塞过程赋值（续）

阻塞与非阻塞赋值语句行为差别举例2波形

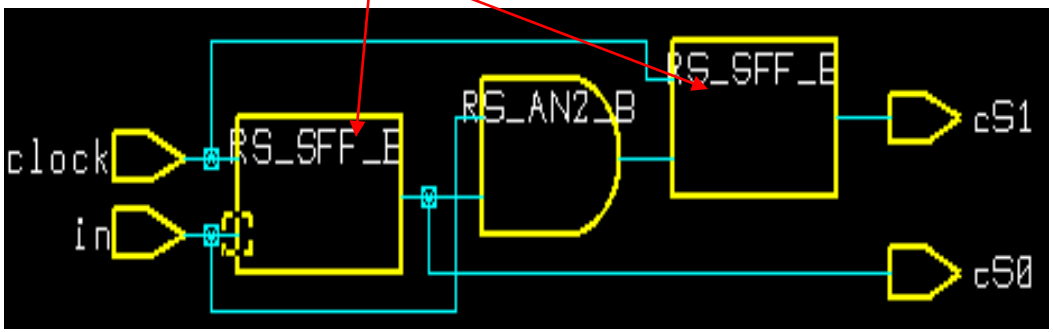


非阻塞　　阻塞

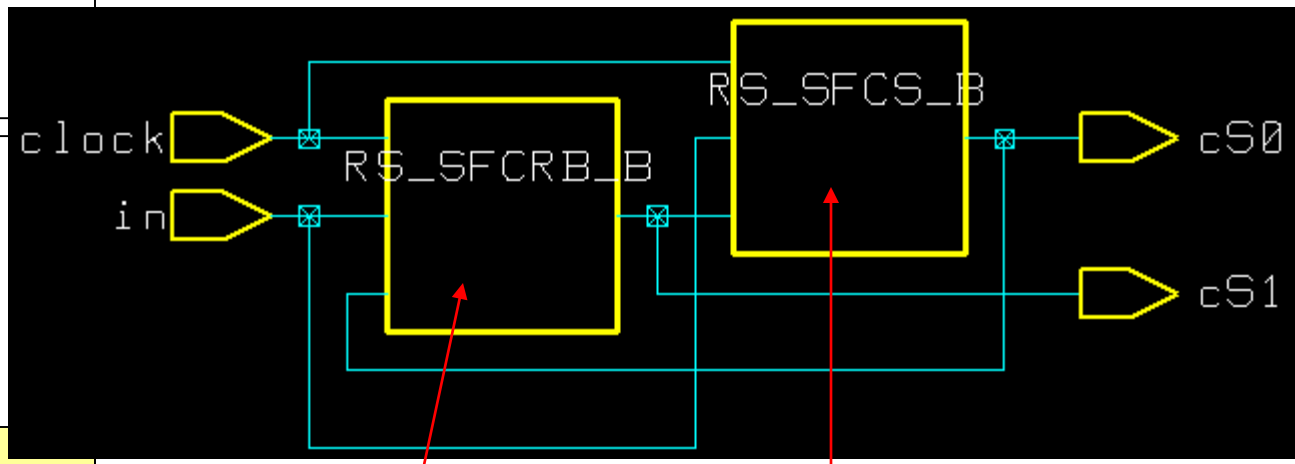
非阻塞过程赋值（续）

阻塞与非阻塞赋值语句行为差别举例3

```
module fsm(cS1, cS0, in, clock);
input in , clock;
output cS1, cS0;
reg cS1, cS0;
always @(posedge clock) begin
   cS1 = in & cS0;   //同步复位
   cS0 = in | cS1;   //cS0 = in
end
endmodule
```




```
module fsm(cS1, cS0, in, clock);
input in , clock;
output cS1, cS0;
reg cS1, cS0;
always @(posedge clock) begin
   cS1 <= in & cS0;   //同步复位
   cS0 <= in | cS1;    //同步置位
end
endmodule
```



非阻塞过程赋值（续）

举例4：非阻塞赋值语句中延时在左边和右边的差别

```
module exchange;
    reg[3:0] a, b;
    initial begin
          a=1; b=4;
       #2 a=3; b=2;
       #20 $finish;
    end
    initial
$monitor($time, "\t%h\t%h", a, b);
    initial begin
          #5 a <=  b;
          #5 b <=  a;
    end
endmodule
```

time	a	b
0	1	4
2	3	2
5	2	2

```
module exchange;
    reg[3:0] a, b;
    initial begin
          a=1; b=4;
       #2 a=3; b=2;
       #20 $finish;
    end
    initial
$monitor($time, "\t%h\t%h", a, b);
    initial begin
          a <=  #5 b;
          b <=  #5 a;
    end
endmodule
```

time	a	b
0	1	4
2	3	2
5	4	1

条件语句（if分支语句）

***if* 和 *if-else* 语句：**

```
always #20
    if (index > 0) // 开始外层 if
        if (rega > regb) // 开始内层第一层 if
            result = rega;
        else
            result = 0; // 结束内层第一层 if
    else
      if (index == 0)
          begin
            $display(" Note : Index is zero");
            result = regb;
          end
        else
          $display(" Note : Index is negative");
```

描述方式：

```
if (表达式)
    begin
        ……
    end
else
    begin
        ……
    end
```

- 可以多层嵌套。在嵌套*if*序列中，*else*和前面最近的if相关。
- 为提高可读性及确保正确关联，使用begin…end块语句指定其作用域。

条件语句（case分支语句）

***case*语句：**

```
module compute (result, rega, regb, opcode);
input [7: 0] rega, regb;
input [2: 0] opcode;
output [7: 0] result;
reg [7: 0] result;
always @( rega or regb or opcode)
    case (opcode)
        3'b000 : result = rega + regb;
        3'b001 : result = rega - regb;
        3‘b010 : // specify multiple cases with the same result
        3'b100 : result = rega / regb;
        default : begin
            result = 'bx;
            $display (" no match");
         end
     endcase
endmodule
```

在Verilog中重复说明case项是合法的，因为Verilog的case语句只执行第一个符合项。

条件语句-case语句

case语句是测试表达式与另外一系列表达式分支是否匹配的一个多路条件语句。

- Case语句进行逐位比较以求完全匹配（包括x和z）。
- Default语句可选，在没有任何条件成立时执行。此时如果未说明default，Verilog不执行任何动作。
- 多个default语句是非法的。

重要内容：

使用default语句是一个很好的编程习惯，特别是用于检测x和z。

Casez和casex为case语句的变体，允许比较无关(don't-care）值。

- case表达式或case项中的任何位为无关值时，在比较过程中该位不予考虑。
- 在casez语句中，？和 z 被当作无关值。
- 在casex语句中，?，z 和 x 被当作无关值。

case语法：

```
case <表达式>
        <表达式>：赋值语句或空语句；
        <表达式>：赋值语句或空语句；
            default：赋值语句或空语句；
```

循环(looping)语句

有四种循环语句：

repeat：将一块语句循环执行确定次数。

repeat (次数表达式） <语句>

while：在条件表达式为真时一直循环执行

while (条件表达式） <语句>

forever：重复执行直到仿真结束

forever <语句>

不可综合

for：在执行过程中对变量进行计算和判断，在条件满足时执行

for(赋初值；条件表达式；计算） <语句>

循环(looping)语句-repeat

***repeat*：将一块语句循环执行确定次数。**

***repeat* (次数表达式） 语句**

```
// Parameterizable shift and add multiplier
module multiplier( result, op_a, op_b);
    parameter size = 8;
    input [size:1] op_a, op_b;
    output [2* size:1] result;
    reg [2* size:1] shift_opa, result;
    reg [size:1] shift_opb;
    always @( op_a or op_b) begin
        result = 0;
        shift_opa = op_a; // 零扩展至16位
        shift_opb = op_b;
        repeat (size) begin
            #10 if (shift_opb[1]) result = result + shift_opa;
            shift_opa = shift_opa << 1; // Shift left
            shift_opb = shift_opb >> 1; // Shift right
        end
    end
endmodule
```

循环(looping)语句

while：只要表达式为真(不为0)，则重复执行一条语句(或语句块)

```
. . .
reg [7: 0] tempreg;
reg [3: 0] count;
. . .
    count = 0;
    while (tempreg) // 统计tempreg中 1 的个数
    begin
        if (tempreg[ 0]) count = count + 1;
        tempreg = tempreg >> 1; // Shift right
    end
end
. . .
```

循环(looping)语句

forever：一直执行到仿真结束

forever应该是过程块中最后一条语句。其后的语句将永远不会执行。

forever语句不可综合，通常用于test bench描述。

```
...
reg clk;
initial
    begin
    clk = 0;
    forever
        begin
            #10 clk = 1;
            #10 clk = 0;
    end
    …
end
...
```

这种行为描述方式可以非常灵活的描述时钟，可以控制时钟的开始时间及周期占空比。仿真效率也高。

循环(looping)语句

for：只要条件为真就一直执行

条件表达式若是简单的与0比较通常处理得更快一些。但综合工具可能不支持与0的比较。

```
// X检测
for (index = 0; index < size; index = index + 1)
    if (val[ index] === 1'bx)
        $display (" found an X");

// 存储器初始化; “!= 0”仿真效率高
for (i = size; i != 0; i = i - 1)
    memory[ i- 1] = 0;

// 阶乘序列
factorial = 1;
for (j = num; j != 0; j = j - 1)
    factorial = factorial * j;
```

行为级零延时循环

当事件队列中所有事件结束后仿真器向前推进。但在零延时循环中，事件在同一时间片不断加入，使仿真器停滞后那个时间片。

在下面的例子中，对事件进行了仿真但仿真时间不会推进。当always块和forever块中没有时序控制时就会发生这种事情。

```
module comparator( out, in1, in2);
output [1: 0] out;
input [7: 0] in1, in2;
reg [1: 0] out;
    always
        if (in1 == in2)
            out = 2'b00;
          else if (in1 > in2)
            out = 2'b01;
          else
            out = 2'b10;
    initial
        #10 $finish;
endmodule
```

第七节 Verilog中的高级结构

学习内容：

- 任务和函数的定义和调用
- 怎样使用命名块
- 怎样禁止命名块和任务

Verilog的任务及函数

结构化设计是将任务分解为较小的，更易管理的单元，并将可重用代码进行封装。这通过将设计分成模块，或任务和函数实现。

- 任务（task)
  - ✓ 通常用于调试，或对硬件进行行为描述
  - ✓ 可以包含时序控制（#延迟，@, wait）
  - ✓ 可以有 input，output，和inout参数
  - ✓ 可以调用其他任务或函数
- 函数(function)
  - ✓ 通常用于计算，或描述组合逻辑
  - ✓ 不能包含任何延迟；函数仿真时间为0
  - ✓ 只含有input参数并由函数名返回一个结果
  - ✓ 可以调用其他函数，但不能调用任务

Verilog的任务及函数

- 任务和函数必须在**module**内调用
- 在任务和函数中不能声明**wire**
- 所有输入/输出都是局部寄存器
- 任务/函数执行完成后才返回结果。

  例如，若任务/函数中有**forever**语句，则永远不会返回结果

任务

下面的任务中含有时序控制和一个输入，并引用了一个module变量，但没有output、inout、内部变量，也不显示任何结果。

时序控制中使用的信号（例如clk）一定不能作为任务的输入，因为输入值只向该任务传送一次。

```
module top;
   reg clk, a, b;
   DUT u1 (out, a, b, clk);
   always #5 clk = !clk;
   task neg_clocks;
      input [31:0] number_of_edges;
      repeat( number_of_edges) @( negedge clk);
   endtask
   initial begin
      clk = 0; a = 1; b = 1;
      neg_clocks(3); // 任务调用
      a = 0;  neg_clocks (5);
      b = 0;
   end
endmodule
```

任务

主要特点：

- 任务可以有input,output 和 inout参数。
- 传送到任务的参数和与任务I/O说明顺序相同。尽管传送到任务的参数名称与任务内部I/O说明的名字可以相同，但在实际中这通常不是一个好的方法。参数名的唯一性可以使任务具有好的模块性。
- 可以在任务内使用时序控制。
- 在Verilog中任务定义一个新范围（scope)
- 要禁止任务，使用关键字disable 。

从代码中多处调用任务时要小心。因为任务的局部变量只有一个拷贝，并行调用任务可能导致错误的结果。在任务中使用时序控制时这种情况时常发生。

在任务或函数中引用调用模块的变量时要小心。如果想使任务或函数能从另一个模块调用，则所有在任务或函数内部用到的变量都必须列在端口列表中。

任务

下面的任务中有输入，输出，时序控制和一个内部变量，并且引用了一个module变量。但没有双向端口，也没有显示。

任务调用时的参数按任务定义的顺序列出。

```
module mult (clk, a, b, out, en_mult);
    input clk, en_mult;
    input [3: 0] a, b;
    output [7: 0] out;
    reg [7: 0] out;
    always @( posedge clk)
        multme (a, b, out); // 任务调用
    task multme; // 任务定义
        input [3: 0] xme, tome;
        output [7: 0] result;
        wait (en_mult)
            result = xme * tome;
    endtask
endmodule
```

函数（function）

```
module orand (a, b, c, d, e, out);
        input [7: 0] a, b, c, d, e;
        output [7: 0] out;
        reg [7: 0] out;
        always @( a or b or c or d or e)
        out = f_or_and (a, b, c, d, e); // 函数调用
        function [7:0] f_or_and;
        input [7:0] a, b, c, d, e;
        if  (e = = 1)
           f_or_and = (a | b) & (c | d);
        else
           f_or_and = 0;
        endfunction
endmodule
```

函数中不能有时序控制，但调用它的过程可以有时序控制。

函数名f_or_and在函数中作为register使用

函数

主要特性：

- 函数定义中不能包含任何时序控制语句。
- 函数至少有一个输入，不能包含任何输出或双向端口。
- 函数只返回一个数据，其缺省为reg类型。
- 传送到函数的参数顺序和函数输入参数的说明顺序相同。
- 函数在模块（module)内部定义。
- 函数不能调用任务，但任务可以调用函数。
- 函数在Verilog中定义了一个新的范围（scope)。
- 虽然函数只返回单个值，但返回的值可以直接给信号连接赋值。这在需要有多个输出时非常有效。

```
{o1, o2, o3, o4} = f_ or_ and (a, b, c, d, e);
```

函数

要返回一个向量值（多于一位），在函数定义时在函数名前说明范围。函数中需要多条语句时用begin和end。

不管在函数内对函数名进行多少次赋值，值只返回一次。下例中，函数还在内部声明了一个整数。

```
module foo;
input [7: 0] loo;
output [7: 0] goo;
// 可以持续赋值中调用函数
wire [7: 0] goo = zero_count ( loo );
  function [3: 0] zero_count;
  input [7: 0] in_ bus;
  integer I;
  begin
    zero_count = 0;
    for (I = 0; I < 8; I = I + 1)
    if (! in_bus[ I ])
    zero_count = zero_count + 1;
  end
  endfunction
endmodule
```

函数

函数返回值可以声明为其它register类型：integer, real, 或time。

在任何表达式中都可调用函数

```
module checksub (neg, a, b);
        output neg;
        reg neg;
        input a, b;
        function  integer  subtr;
                input [7: 0] in_a, in_b;
                subtr = in_a - in_b; // 结果可能为负
        endfunction
        always @ (a or b)
                if (subtr( a, b) < 0)
                    neg = 1;
                else
                    neg = 0;
endmodule
```

函数

函数中可以对返回值的个别位进行赋值。

函数值的位数、函数端口甚至函数功能都可以参数化。

```
. . .
parameter MAX_BITS = 8;
reg [MAX_BITS: 1]  D;
function  [MAX_BITS: 1]  reverse_bits;
        input [MAX_BITS-1: 0] data;
        integer K;
        for (K = 0; K < MAX_BITS; K = K + 1)
           reverse_ bits [MAX_BITS - (K+ 1)] = data [K];
endfunction
always @ (posedge clk)
        D = reverse_bits (D) ;
. . .
```

Verilog 系统函数

- 统一以“$”开头
  - 输出控制：$display,$write,$monitor
  - 模拟时标：$time,$realtime
  - 进程控制：$finish,$stop
  - 文件读写：$readmem
  - 其它：　　$random $signed $unsigned $fopen $fclose $fdisplay $fwrite $fmonitor……

Verilog 系统函数

- **$display与 $write**

$write**和**$display列出所指定信号的值，它们的功能都相同，唯一不同点在**$display**输出结束后会自动换行，而**$write**不会换行。

例：

```
$write ("%b \t %h \t %d \t %o\n", a, b, c, d)；

$display ("%b \t %h \t %d \t %o", a, b, c, d)；
```

Verilog 系统函数

输出格式说明符以及转义字符

Format Specification	Escaped character
%h or %H display in hexadecimal format %d or %D display in decimal format %o or %O display in octal format %b or %B display in binary format %c or %C display in ASCII format %v or %V display net signal strength %m or %M display hierarchical name %s or %S display as a string %t or %T display in current time format	\n is the new line character \t is the tab character \\ is the backslash character \” is the ” character \o 1-3 digits octal number %% is the percent character

Verilog 系统函数

- **$monitor：输出变量的任何变化，都会输出一次结果；而$write和$display每调用一次执行一次**

例：

```
module monitor_test；
  reg in; wire out；
  not #1 U0(out, in)；
  initial
    $monitor($time, "out = %b in = %b", out, in)；
  initial begin
    in = 0；
    # 10 in = 1；
    # 10 in = 0；
   end
endmodule
```

输出结果为：(注意延迟！)

0 out = x in = 0

1 out = 1 in = 0

10 out = 1 in = 1

11 out = 0 in = 1

20 out = 0 in = 0

21 out = 1 in = 0

例:

```
module monitor_test；
    reg in；wire out；
    not #1 U0(out, in)；
    initial
      $display($time, "out = %b in = %b", out, in)；
    initial begin
        in = 0；
        #10 in = 1；
        #10 in = 0；
    end
endmodule
```

输出结果为：

```
out = x  in = x
```

Verilog系统函数

- **模拟时标：返回从执行到调用时刻的时间**

  $time : 返回一个 64-bit 的整数。

  $realtime : 返回一个实数。

例：

$monitor($time, "out = %b in = %b", out, in)；

Verilog 系统函数

- **$finish与$stop**
  - $finish终止仿真器的运行
  - $stop暂停模拟程序的执行，不退出仿真进程

- **$readmem:把文件内容读入指定存储器**

  readmemb(“文件名”，存储器名，起始地址，结束地址);

  readmemh(“文件名”，存储器名，起始地址，结束地址);

例：

```
reg [7：0] mem[1：256];
initial $readmemh（”mem.data”, mem）；
initial $readmemh（”mem.data”, mem, 128, 156）；
```

命名块(named block)

- 在关键词begin或fork后加上：<块名称> 对块进行命名

```
module named_ blk;
 . . .
   begin : seq_blk
. . .
   end
. . .
   fork : par_blk
. . .
   join
. . .
endmodule
```

- 在命名块中可以声明局部变量
- 可以使用关键词disable禁止一个命名块
- 命名块定义了一个新的范围
- 命名块会降低仿真速度

禁止命名块和任务

```
module do_arith (out, a, b, c, d, e, clk, en_mult);
      input clk, en_mult;
      input [7: 0] a, b, c, d, e;
      output [15: 0] out;
      reg [15: 0] out;
      always @( posedge clk)
            begin : arith_block // *** 命名块 ***
                  reg [3: 0] tmp1, tmp2; // *** 局部变量 ***
                  {tmp1, tmp2} = f_or_and (a, b, c, d, e); // 函数调用
                  if (en_mult)  multme (tmp1, tmp2, out); // 任务调用
            end
      always @( negedge en_mult) begin // 中止运算
            disable  multme ; // *** 禁止任务 ***
            disable  arith_block; // *** 禁止命名块 ***
      end
// 下面[定义任务和函数
      ……
endmodule
```

禁止命名块和任务

- **disable**语句终结一个命名块或任务的所有活动。也就是说，在一个命名块或任务中的所有语句执行完之前就返回。

  语法：

  disable <块名称>

  或

  disable <任务名称>

- 当命名块或任务被禁止时，所有因他们调度的事件将从事件队列中清除
- **disable**是典型的不可综合语句。
- 在前面的例子中，只禁止命名块也可以达到同样的目的：所有由命名块、任务及其中的函数调度的事件都被取消。