

Verilog HDL基础

计算机组成原理课程组

(刘旭东、肖利民、牛建伟、栾钟治)

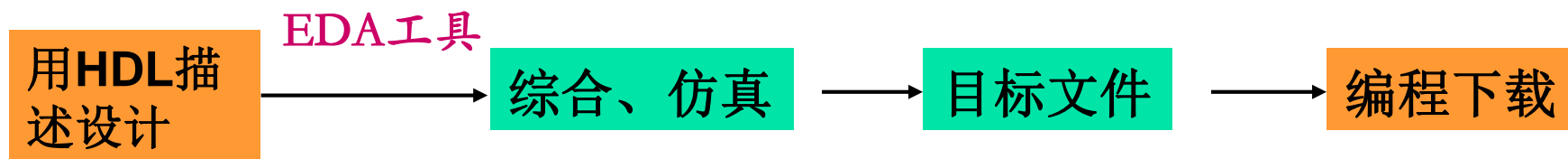
Tel : 82316285

Mail: liuxd@buaa.edu.cn

liuxd@act.buaa.edu.cn

Verilog HDL概述

- ❖ **硬件描述语言**（**Hradware Description Language**）是一种用形式化方法（即文本形式）来描述和设计数字电路和数字系统的高级模块化语言。它是设计人员和**EDA**工具之间的一个桥梁，主要用于编写**设计文件**，在**EDA**工具中建立电路模型；也用来编写**测试文件**进行仿真。



- ❖ **HDL**发展至今已有近三十年的历史，到**20世纪80年代**，已出现了数十种硬件描述语言。**80年代后期**，**HDL**向着标准化、集成化的方向发展，最终**VHDL**、**Verilog HDL**先后成为**IEEE**标准。

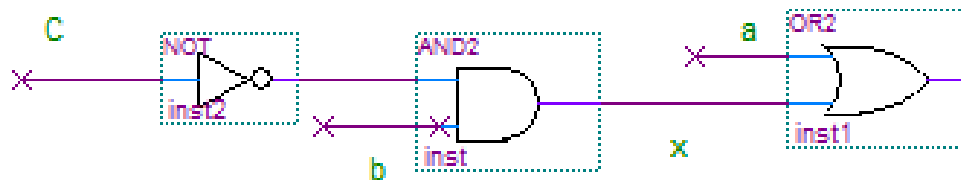
➤ **VHDL**：VHSIC Hardware Description Language（**VHSIC**——**Very High Speed Integrated Circuits**），甚高速集成电路的硬件描述语言，来源于美国军方，**1987年**成为**IEEE**标准。目前标准化程度最高的一种**HDL**。

Verilog HDL概述

- ❖ Verilog HDL可以用来进行数字电路的建模、仿真验证、时序分析、逻辑综合。
- ❖ Verilog HDL抽象级别：系统级，算法级，RTL级，门级，开关级
- ❖ Verilog HDL具有**行为描述**和**结构描述**功能。行为描述包括系统级、算法级和RTL级3种抽象级别；结构描述包括包括门级和开关级2种抽象级别。
- ❖ 语法结构上的主要**特点**
 - 形式化地表示电路的**行为**和**结构**；
 - 借用**C语言**的结构和语句；
 - 可在多个层次上对所设计的系统加以描述，语言对设计规模不加任何限制；
 - 具有混合建模能力：一个设计中的各子模块可用不同级别的抽象模型来描述；
 - 基本逻辑门、开关级结构模型均内置于Verilog HDL语言库中，可直接调用；

Verilog HDL模块的结构

- ❖ Verilog的基本设计单元是“**模块（module）**”，实现一个特定功能
- ❖ 一个“与门”、“加法器”、**ALU**都可以是一个模块
- ❖ Verilog模块的结构由在**module**和**endmodule**关键词之间的**4**个主要部分组成：



1

端口定义

2

I/O说明

3

信号类型声明

4

功能描述

```
module block1(a,b,c,d );  
    input a, b, c; /* I/O变量缺省为wire变量*/  
    output d;  
    wire x;  
    assign d = a | x; //组合逻辑功能描述  
    assign x = ( b & ~c );  
endmodule
```

结构描述

Verilog HDL实例

一、简单的Verilog HDL例子

【例1】 8位全加器

模块名(文件名)

```
module adder8 ( cout,sum,a,b,cin );  
    output cout;           // 输出端口声明  
    output [7:0] sum;  
    input [7:0] a,b;       // 输入端口声明  
    input cin;  
    assign {cout,sum}=a+b+cin;  
endmodule
```

端口定义

I/O说明

功能描述

- ❖ 整个程序嵌套在module和endmodule声明语句中。
- ❖ 每条语句相对module和endmodule最好缩进2格或4格！
- ❖ // 表示注释部分，一般只占据一行。对编译不起作用！
- ❖ []: 数组/总线定义
- ❖ {}: 位拼接运算符

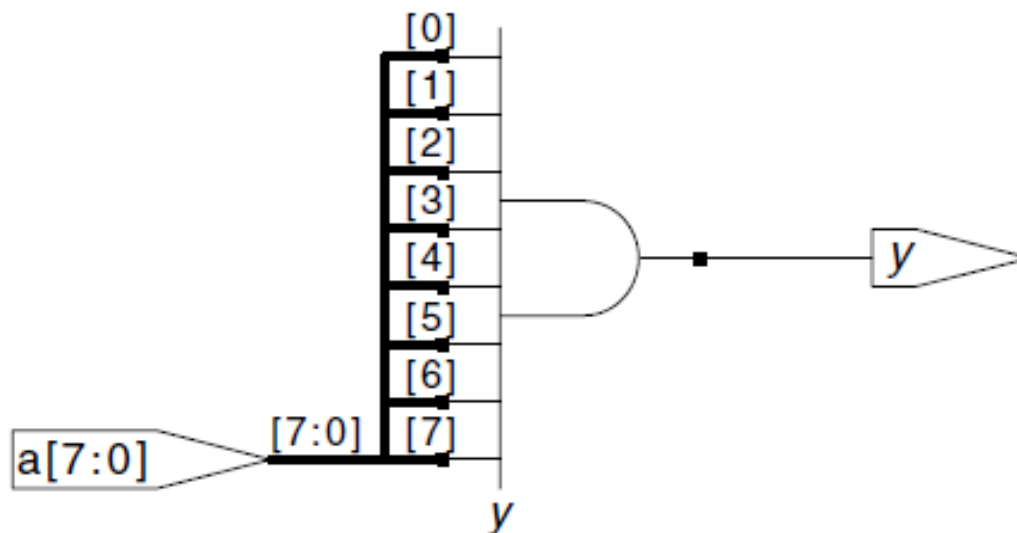
单行注释符

Verilog HDL实例

【例2】 8位与门

```
module and8 (input [7:0] a, output y);  
    assign y & a; // &a is much easier to write than  
    // assign y= a[7] & a[6] & a[5] & a[4] & a[3] & a[2] & a[1] & a[0];  
endmodule
```

❖ &: 缩位操作符

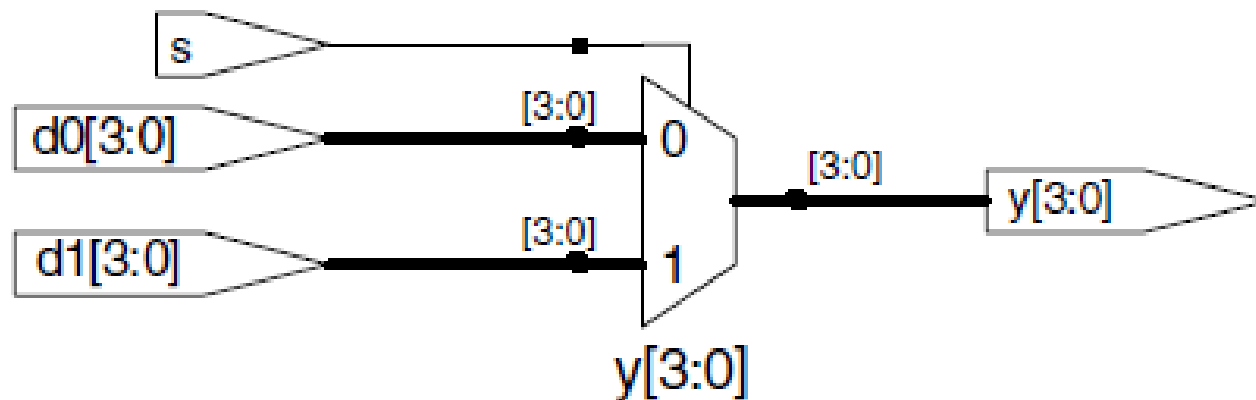


Verilog HDL实例

【例3】 多路选择器

```
module mux2 (input [3:0] d0, d1,  
  input s, output [3:0] y);  
  assign y = s ? d1 : d0;  
endmodule
```

❖ ?： 条件运算符

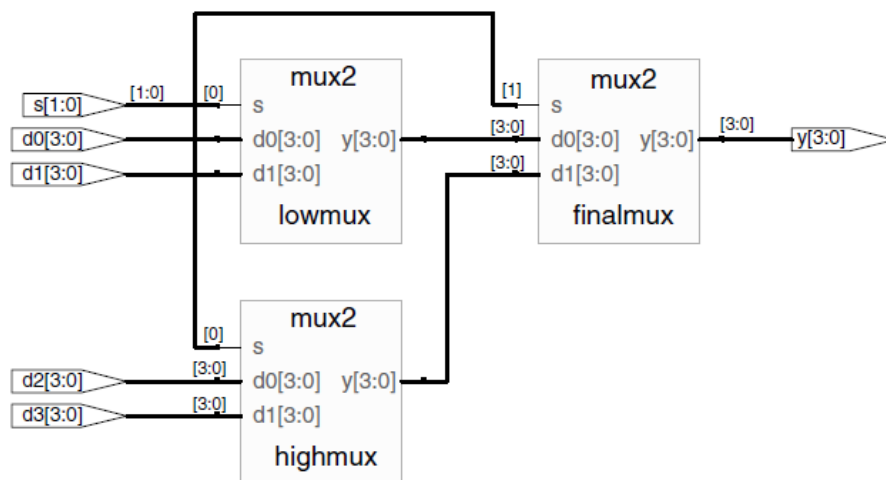


Verilog HDL实例

【例4】多路选择器

```
module mux4 (input [3:0] d0, d1, d2, d3, input [1:0] s, output [3:0] y);  
  wire [3:0] low, high;  
  mux2 lowmux (d0, d1, s[0], low);  
  mux2 highmux (d2, d3, s[0], high);  
  mux2 finalmux (low, high, s[1], y);  
Endmodule
```

```
module mux2 (input [3:0] d0, d1, input s, output [3:0] y);  
  tristate t0 (d0, ~s, y);  
  tristate t1 (d1, s, y);  
endmodule
```



Verilog HDL模块的结构

1、模块端口定义

- 模块端口定义用来声明设计电路模块的输入输出端口，端口定义格式：
module 模块名(端口1, 端口2, 端口3, ...);
- 在端口定义的圆括弧中，是设计电路模块与外界联系的全部输入输出端口信号或引脚，它是设计实体对外的一个通信界面，是外界可以看到的部分（不包含电源和接地端），多个端口名之间用“,”分隔。

2、I/O说明

- 模块的I/O说明用来声明模块端口定义中各端口数据流动方向，包括输入（**input**）、输出（**output**）和双向（**inout**）。
input 端口1, 端口2, 端口3, ...;
output 端口1, 端口2, 端口3, ...;
inout 端口1, 端口2, 端口3, ...;

- 端口定义、I/O说明和程序语句中的标点符号及圆括弧均要求用**半角**符号书写！

Verilog HDL模块的结构

3、信号类型声明

- 信号类型声明用来说明设计电路的功能描述中，所用的信号的数据类型以及函数声明。
- 信号的数据类型主要有**连线**（**wire**）、**寄存器**（**reg**）、**整型**（**integer**）、**实型**（**real**）和**时间**（**time**）等类型。

4、功能描述

- 功能描述是**Verilog HDL**程序设计中最主要的部分，用来描述设计模块的**内部结构**和模块端口间的**逻辑关系**，在电路上相当于器件的内部电路结构。
- 功能描述可以用**assign**语句、元件例化（**instantiate**）、**always**块语句等方法来实现，通常把确定这些设计模块描述的方法称为**建模**。

逻辑功能定义

❖ 在Verilog 模块中有3种方法可以描述电路的逻辑功能:

① 用assign 语句

数据流描述

常用于描述
组合逻辑

```
assign x = ( b & ~c );
```

② 用元件例化 (instantiate)

```
and myand3( f,a,b,c);
```

结构描述

门元件关键字

例化元件名

- ❖ 注1: 元件例化即是调用Verilog HDL提供的元件或由某子模块定义的实例元件;
- ❖ 注2: 元件例化包括门元件例化和模块元件例化;
- ❖ 注3: 例化元件名也可以省略! 若不省略, 则每个实例元件的名字必须唯一! 以避免与其它调用该元件的实例相混淆。

逻辑功能定义（续）

③ 用“always”块语句

行为描述

```
always @(posedge clk) // 每当时钟上升沿到来时执行一遍块内语句
begin
    if(load)      out = data;           // 同步预置数据
    else          out = data + 1 + cin; // 加1计数
end
```

- ❖ 注1：“always”块语句常用于描述时序逻辑，也可描述组合逻辑。
- ❖ 注2：“always”块可用多种手段来表达逻辑关系，如用if-else语句或case语句。
- ❖ 注3：“always”块语句与assign语句是并发执行的，assign语句一定要放在“always”块语句之外！

Verilog HDL语句

❖ 语句及函数的比较

语句及函数	C语言	Verilog HDL
函数	无参函数，有参函数	function 块语句
赋值语句	赋值变量 = 表达式;	阻塞赋值，非阻塞赋值
条件语句	if-else	if-else
条件语句	switch	case
循环语句	for	for
循环语句	while	while
中止语句	break	break
宏定义语句	define （以符号#开头）	define （以符号'开头）
格式输出函数	printf	printf

运算符的比较

C语言	Verilog HDL	功能	C语言	Verilog HDL	功能
+	+	加	<=	<=	小于等于
-	-	减	==	==	等于
*	*	乘	!=	!=	不等于
/	/	除	~	~	按位取反
%	%	取模	&	&	按位与
!	!	逻辑非			按位或
&&	&&	逻辑与	^	^	按位异或
		逻辑或	<<	<<	左移
>	>	大于	>>	>>	右移
<	<	小于	?:	?:	等同于if-else
>=	>=	大于等于			

Verilog HDL与C语言的运算符几乎完全相同!

3.2 Verilog HDL模块的结构

❖ Verilog HDL程序的三种描述方式

结构 (Structural) 描述

- 对设计电路的**结构**进行描述，即描述设计电路使用的元件及这些元件之间的连接关系
- 属于**低层次**的描述方法，包括门级和开关级2种抽象级别

行为 (Behavioral) 描述

- 对设计电路的**逻辑功能**的描述，并不关心设计电路使用哪些元件以及这些元件之间的连接关系
- 属于**高层次**的描述方法，包括系统级、算法级和寄存器传输级等3种抽象级别

数据流 (Data Flow) 描述

- 采用持续赋值语句

3.3 Verilog HDL模块

❖ Verilog HDL模块的模板（仅考虑用于逻辑综合的程序）

```
module <顶层模块名> (<输入输出端口列表>);  
    output 输出端口列表;  
    input 输入端口列表;  
    //（1）使用assign语句定义逻辑功能  
    wire <结果信号名>;  
    assign <结果信号名> = 表达式;  
    //（2）使用always块定义逻辑功能  
    always @(<敏感信号表达式>)  
        begin  
            //过程赋值语句  
            //if语句  
            //case语句  
            //while,repeat,for循环语句  
            //task,function调用  
        end
```


3.3 Verilog HDL模块的模板

❖ Verilog HDL模块的模板（续）

// （3）元件例化

```
< module_name > < instance_name > (<port_list>); // 模块元件例化  
<gate_type_keyword> < instance_name > (<port_list>); // 门元件例化  
endmodule
```

例化元件名也
可以省略！

2.5.2 Verilog HDL的词法

- ❖ Verilog HDL源程序由空白符分隔的词法符号流组成。
- ❖ 词法符号包括空白符、注释、操作符（运算符）、常数、字符串、标识符及关键字。

一、空白符和注释

- ❖ Verilog HDL的空白符包括空格、Tab、换行和换页符号。空白符如果不是出现在字符串中，编译源程序时将被忽略。
- ❖ 注释用来帮助读者理解程序，编译源程序时将被忽略。注释分为行注释和块注释两种方式。
 - 行注释用符号//（两个斜杠）开始，注释到本行结束。
 - 块注释用/*开始，用*/结束。块注释可以跨越多行，但它们不能嵌套。

二、常数

- ❖ 常数包括整数、**x**（未知）和**z**（高阻）值、负数、实数
- ❖ 整数的**4**种进制表示形式：

- 二进制整数（**b**或**B**）；
- 十进制整数（**d**或**D**）；
- 十六进制整数（**h**或**H**）；
- 八进制整数（**o**或**O**）。

建议最好写明
位宽和进制—
—清楚，不易
出错！

整数的 3 种表达方式	说 明	举 例
<位宽> ' <进制符号> <数字>	完整的表达方式	8'b11000101或 8'hc5
<进制符号> <数字>	缺省位宽，则位宽由机器系统决定，至少 32 位	hc5
<数字>	缺省进制为十进制，位宽默认为 32 位	197

- ❖ 这里位宽指对应二进制数的宽度。
- ❖ 整数型常量是可以综合的，而实数型和字符串型常量都是不可综合的

x和z值

❖ x和z值

- x表示不定值，z表示高阻值；
- x和z代表的二进制位数取决于所用的进制

“?”是z的另一种表示符号，建议在case语句中使用?表示高阻态z

【例2.17】casez (select)

```
4'b???1: out = a;  
4'b??1?: out = b;  
4'b?1??: out = c;  
4'b1???: out = d;  
endcase
```

❖ 负数

- 在位宽前加一个负号，即表示负数，负数通常表示为该负数的二进制补码
- 如：-8'd5 // -5的补码，= 8'b11111011

实数 (Real)

❖ 实数的两种表示法

➤ 十进制表示法

- 2.0, 5.678, 0.1 //合法
- 2. //非法, 小数点两侧都必须有数字

➤ 科学计数法

- 43_5.1e2 // 等于 $435.1 \times 10^2 = 43510$
- 5E-4 //等于 $5 \times 10^{-4} = 0.0005$, e与E相同

❖ 实数通过四舍五入被转换为最相近的整数

【例】42.446, 42.45 //若转换为整数都是42

92.5, 92.699 //若转换为整数都是93

-15.62, -25.26 //若转换为整数分别为-16, -25

- 下划线 “_”可随意用在整数或实数的数字中间, 以提高可读性; 但数字的第1个字符不能是下划线, 也不能用在位宽和进制处。

三、字符串

❖ **字符串**是用双引号括起来的可打印字符序列，不能多行书写。

❖ **作用**：在仿真时显示一些相关信息，或者指定显示的格式

▶ 例：“INTERNAL ERROR”， “this is an example for Verilog HDL”

❖ 字符串能够用在系统任务（如\$display、\$monitor）中作为变量，字符串的值可像数值一样存储在寄存器中，也可以像对数字一样对字符串进行赋值、比较和拼接操作

【例】\$display(\$time,,,”a=%h b=%h c=%h”,a,b,c);

// 显示当前仿真时间，空3格后显示a=xx b=xx c=xx

❖ 字符串属于reg型变量，宽度为字符串中字符的个数乘以8。

【例】reg[8*12:1] stringvar;

initial

begin

stringvar = “Hello world!”;

end

四、标识符

- ❖ 任何用**Verilog HDL**语言描述的对象都通过其名字来识别，这个名字被称为**标识符**。标识符可由字母、数字、下划线和\$符号构成。
- ❖ 如源文件名、模块名、端口名、变量名、常量名、实例名等。
- ❖ 定义标识符时应遵循如下规则
 - ① **首字符必须是字母或下划线，不能是数字或\$符号！**
 - ② 字符数不能多于**1024**个。
 - ③ 大小写字母是不同的。
 - ④ **不要与关键字同名！**

❖ 合法的名字：

- **A_99_Z**
- **Reset**
- **_54MHz_Clock\$**
- **Module**

❖ 不合法的名字：

- **123a**
- **\$data**
- **module**
- **7seg.v**
- **out*** **//不允许包含字符***

五、关键字

- ❖ **关键字（保留字）**——Verilog HDL事先定义好的确认符，用来组织语言结构；或者用于定义Verilog HDL提供的门元件（如and, not, or, buf）。
- ❖ 每个关键字全部用**小写**字母定义！
——如always, assign, begin, case, casex, else, end, endmodule, for, function, if, input, module, output, repeat, table, time, while, wire
- ❖ Verilog -1995的关键字有**97**个（见教材表2.14），Verilog -2001增加了5个共**102**个。

六、运算符及表达式

❖ 运算符也称为操作符，是Verilog HDL预定义的函数符号，这些函数对被操作的对象（即操作数）进行规定的运算，得到一个结果。

❖ 运算符按功能分为9类：

- 算术运算符
- 逻辑运算符
- 关系运算符
- 等值运算符
- 缩减运算符
- 条件运算符
- 位运算符
- 移位运算符
- 位拼接运算符

❖ 运算符按操作数的个数分为3类：

- 单目运算符——带一个操作数
- 逻辑非！，按位取反~，缩减运算符，移位运算符
- 双目运算符——带两个操作数
- 算术、关系、等值运算符，逻辑运算符（除逻辑非外）、位运算符（除按位取反外）
- 三目运算符——带三个操作数
- 条件运算符

1、算术运算符

❖ 双目运算符

算术运算符	功能
+	加
-	减
*	乘
/	除
%	求模

- ❖ 进行整数除法运算时，结果值略去小数部分，只取整数部分！
- ❖ **求模**即是求一个数被另一个数相除后所得的余数。**%**称为**求模**（或**求余**）运算符，要求**%**两侧均为**整型**数据；
- ❖ 求模运算结果值的符号位取第一个操作数的符号位！
【例】 $-11\%3$ 结果为-2
- ❖ 进行算术运算时，若某操作数为不定值**x**，则整个结果也为**x**。

2、逻辑运算符

❖ 逻辑运算符把它的操作数当作**布尔变量**（逻辑1、逻辑0或不定值）：

➤ **非零**的操作数被认为是**真**(1'b1)；

➤ **零**被认为是**假**(1'b0)；

➤ **不确定**的操作数如4'bxx00，被认为是不确定的（可能为零，也可能为非零）（记为1'bx）； 但4'bxx11被认为是真（记为1'b1，因为它肯定是非零的）。

【例】 若A=4'b0000，B=4'b0101

则 !A=1'b1，A&&B=1'b0，A||B=1'b1。

逻辑运算符	功能
&&(双目)	逻辑与
(双目)	逻辑或
!(单目)	逻辑非

❖ 如果操作数不止一位，应将操作数作为一个整体来对待！

❖ 进行逻辑运算后的结果为布尔值（为1或0或x）！

3、位运算符

位运算符	功能
~	按位取反
&	按位与
	按位或
^	按位异或
^~, ~^	按位同或

单目运算符

双目运算符

- ❖ 位运算符中的双目运算符要求对两个操作数的相应位**逐位**进行逻辑运算。位运算其结果与操作数位数相同。
- ❖ 两个不同长度的操作数进行位运算时，将自动按**右端对齐**，位数少的操作数会在高位用**0**补齐。

【例】 若 $A = 5'b11001$, $B = 3'b101$,
则 $A \& B = (5'b11001) \& (5'b00101) = 5'b00001$

4、关系运算符

❖ 双目运算符

❖ 用来对两个操作数进行比较。

也是非阻塞赋值
运算的赋值符号

关系运算符	功能
<	小于
<=	小于或等于
>	大于
>=	大于或等于

❖ 运算结果为1位的逻辑值1或0或x。关系运算时，若声明的关系为真，则返回值为1；若关系为假，则返回值为0；若某操作数为不定值x，则返回值为x，表示结果是模糊的。

❖ 所有的关系运算符优先级别相同。

❖ 关系运算符的优先级低于算术运算符。

【例】 $a < \text{size} - 1$ 等同于： $a < (\text{size} - 1)$

$\text{size} - (1 < a)$ 不等同于： $\text{size} - 1 < a$

括号内先运算

算术运算先运算

5、等值运算符

❖ 双目运算符

等值运算符	功能
==	等于
!=	不等于
===	全等
!==	不全等

Quartus II不支持!

- ❖ 运算结果为1位的逻辑值1或0或x。
- ❖ 所有的等值运算符优先级别相同。
- ❖ ==和!==运算符常用于case表达式的判别，又称为“case等式运算符”。

❖ 等于运算符(==)和全等运算符(===)的区别:

- 使用等于运算符时，两个操作数必须逐位相等,结果才为1；若某些位为x或z，则结果为x。
- 使用全等运算符时，若两个操作数的相应位形式上完全一致（如同是1，或同是0，或同是x，或同是z），则结果为1；否则为0。

6、缩减（缩位）运算符

- ❖ 单目运算符
- ❖ 运算法则与位运算符类似，但运算对象只有一个，运算过程不同！

注意缩减运算符和位运算符的区别！

缩减运算符	功能
&	与
~&	与非
	或
~	或非
^	异或
^~, ~^	同或

位运算符没有

- ❖ 对单个操作数进行递推运算,即先将操作数的最低位与第二位进行与、或、与非、或非等运算，再将运算结果与第三位进行相同的运算，依次类推，直至最高位。
- ❖ 运算结果缩减为1位二进制数。

【例】 `reg[3:0] a;`

`b=a;` //等效于 `b=((a[0]|a[1])|a[2])|a[3]`

【例2.18】 设 $A = 8'b11010001$ ，则 $\&A = 0$ （在与缩减运算中，只有A中的数字全为1时，结果才为1）； $|A = 1$ （在或缩减运算中，只有A中的数字全为0时，结果才为0）。

7、移位运算符

❖ 单目运算符

❖ 常用于移位寄存器的设计

移位运算符	功能
>>	右移
<<	左移

❖ 用法: $A \gg n$ 或 $A \ll n$

将操作数右移或左移 n 位, 同时用 n 个0填补移出的空位。注意操作数的位数不变!

【例】 $4'b1001 \gg 3$ 的结果 = $4'b0001$; $4'b1001 \gg 4$ 的结果 = $4'b0000$

$4'b1001 \ll 1$ 的结果 = $4'b0010$; $4'b1001 \ll 2$ 的结果 = $4'b0100$;

$1 \ll 6 = 32'b00...01000000$

左移的数据
会丢失!

右移的数据
会丢失!

将操作数右移或左移 n 位, 相当于将操作数除以或乘以 2^n 。

8、条件运算符

❖ 三目运算符

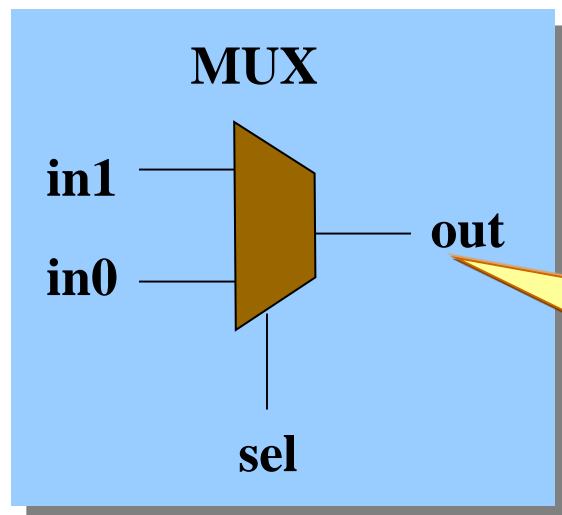
❖ 常用于数据选择器的设计

❖ 条件运算符为？：

❖ 用法：信号 = 条件？表达式1：表达式2；

当条件为真，信号取表达式1的值；为假，则取表达式2的值。

【例】数据选择器 `assign out = sel? in1:in0;`



`sel=1`时 `out=in1`;
`sel=0`时 `out=in0`

9、位拼接运算符

❖ 位拼接运算符为 { }

❖ 用于将两个或多个信号的某些位拼接起来，表示一个整体信号。

❖ 用法： {信号1的某几位, 信号2的某几位,, 信号n的某几位}


➤ 例如在进行全加运算时，可将进位输出与算术和拼接在一起使用。

```
【例2.19】 output [3:0] sum;           //算术和
              output cout;             //进位输出
              input[3:0] ina,inb;
              input cin;
              assign {cout,sum} = ina + inb +cin; //进位与算术和拼接在一起
```

➤ 位拼接可以嵌套使用，或用重复法简化书写

```
【例2.20】 {3{a,b[3:0]}}
            ={ {a,b[3],b[2],b[1],b[0]}, {a,b[3],b[2],b[1],b[0]}, {a,b[3],b[2],b[1],b[0]}}
```

运算符的优先级

类 别	运 算 符	优先级
逻辑非、按位取反	! ~	高  低
算术运算符	* / %	
	+ -	
移位运算符	<< >>	
关系运算符	< <= > >=	
等式运算符	= = ! = == = ! ==	
缩减运算符	& ~&	
	^ ^~	
	~	
逻辑运算符	&&	
条件运算符	? :	

❖ 为避免错误，提高程序的可读性，建议使用括号来控制运算的优先级！

【例】 (a>b)&&(b>c)
(a==b)|| (x==y)
(!a)|| (a>b)

七、Verilog HDL数据对象

❖ Verilog HDL数据对象是指用来存放各种类型数据的容器，包括常量和变量。

(1) 常量

- ▶ 常量用来存放一个恒定不变的数据，一般在程序前部定义。用 **parameter** 来定义一个标识符，代表一个常量，称为**符号常量**或 **parameter** 常量。

parameter 常量名1 = 表达式, 常量名2 = 表达式, ..., 常量名n = 表达式;

- ▶ **parameter** 是常量定义关键字，常量名是用户定义的标识符，表达式是为常量赋的值。
- ▶ 每个赋值语句的右边必须为**常数**表达式，即只能包含数字或先前定义过的符号常量！
- ▶ **parameter** 常量常用来定义**延迟时间**和**变量宽度**。当程序中有多处地方用到相同的常量时，建议用 **parameter** 常量来定义—便于修改，有意义
- ▶ **parameter** 常量是**本地**的，其定义只在本模块内有效。

【例】 **parameter** addrwidth = 16;

变量

(2) 变量

- ❖ 在程序运行过程中，其值可以改变的量，称为**变量**。
- ❖ 其数据类型有**19**种，常用的有**3**种：
 - 网络型（**nets type**）
 - 寄存器型（**register type**）
 - 数组（**memory type**）

❖ 其它数据类型：**large**型、**medium**型、**scalared**型、**small**型、**time**型、**tri**型、**tri0**型、**tri1**型、**triand**型、**trior**型、**trireg**型、**vectored**型、**wand**型、**wor**型等

1. nets型变量

- ❖ **网络型变量**（nets型变量）是输出值始终随输入的变化而变化的变量。
- ❖ 一般用来定义电路中的各种物理连线。
- ❖ 有两种驱动方式：在结构描述中将其连接到一个门元件或模块的输出端；或用assign语句对其赋值
- ❖ 常用的nets型变量
 - wire, tri: 连线类型（两者功能一致），可综合
 - wor, trior: 具有线或特性的连线（两者功能一致）
 - wand, triand: 具有线与特性的连线（两者功能一致）
 - tri1, tri0: 上拉电阻和下拉电阻
 - supply1, supply0: 电源（逻辑1）和地（逻辑0），可综合

wire型变量

❖ wire型变量

- 最常用的**nets**型变量，常用来表示以**assign**语句赋值的**组合**逻辑信号。
- 模块中的输入/输出信号类型**缺省**为**wire**型——当对输入/输出信号不加以信号类型声明时，则输入/输出信号为**wire**型。
- 可用做任何方程式的输入，或“**assign**”语句和实例元件的输出。

格式

wire 变量名1,变量名2, ...,变量名n;

【例】 将输入**a**赋值给**wire**型变量**b**

```
input a;
```

```
wire b;      /* 中间节点。若为output信号，则  
默认为wire型变量，不必单独声明 */
```

```
assign b=a; //当a变化时，b立即随之变化
```

wire型向量（总线）

位宽为1位的变量称为**标量**，

位宽超过1位的变量称为**向量**。向量的宽度定义：

[MSB : LSB] /* MSB(Most Significant Bit, 最高有效位) ,
LSB (Least Significant Bit, 最低有效位) */

格式

wire[n-1:0] 变量名1,变量名2, ...,变量名m;
或 **wire[n:1]** 变量名1,变量名2, ...,变量名m;

每条总线
位宽为**n**

共有**m**
条总线

【例】 wire型向量

wire[7:0] in,out;

assign out=in; //将等号右边的值赋给等号左边的变量。

register型变量

2. register型变量

❖ **寄存器型变量**（**register型变量**）对应**具有状态保持作用**的电路元件（如触发器、寄存器等），常用来表示**过程块**语句（如**initial**, **always**, **task**, **function**）内的指定信号。

❖ 常用的**register型变量**

➤ **reg**: 常代表触发器、寄存器，**可综合**

➤ **integer**: 32位带符号整数型变量，**可综合**

➤ **real**: 64位带符号实数型变量，表示实数寄存器，用于仿真

➤ **time**: 无符号时间变量，用于对仿真时间的存储与处理

纯数学的
抽象描述

register型变量与nets型变量的区别

- ❖ **register**型变量需要被明确地赋值，并且在被重新赋值前一直保持原值。
- ❖ **register**型变量必须通过**过程**赋值语句赋值！不能通过**assign**语句赋值！
- ❖ **nets**型变量必须通过**assign**语句赋值！不能通过过程赋值语句赋值！
- ❖ 在**always**、**initial**、**task**、**function**等过程块内被赋值的每个信号必须定义成**register**型！

reg型变量

❖ reg型变量

- **reg型变量**是数字系统中存储设备的抽象，常用于具体的硬件描述，是最常用的寄存器型变量。
- 它是在过程块中被赋值的信号，**往往**代表触发器（**沿**触发时），但**不一定**就是触发器（也可以是组合逻辑信号，**电平**触发时）！

格式

reg 变量名1,变量名2,,变量名n;

❖ reg型向量（总线）

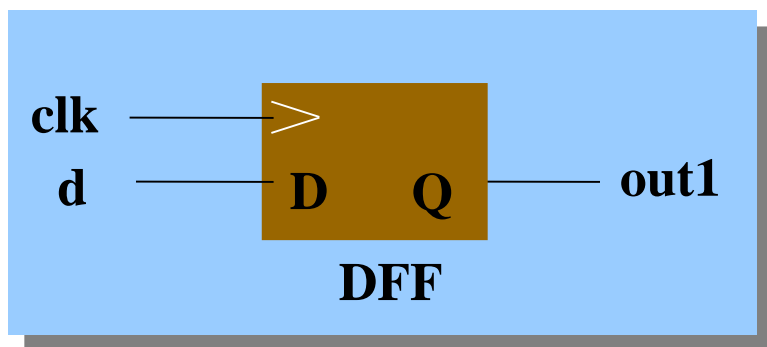
reg[n-1:0] 变量名1,变量名2, ...,变量名m;
或 **reg[n:1]** 变量名1,变量名2, ...,变量名m;

- 【例】** `reg[4:1] regc,regd; //regc,regd为4位宽的reg型向量`
`reg[0:7] data; //8位寄存器型变量，最高有效位是0，最低有效位是7`
- 向量定义后可以采用多种使用形式（即赋值）
`data=8'b00000000; data[5:3]=3'B111; data[7]=1;`

reg型变量生成触发器和组合逻辑举例

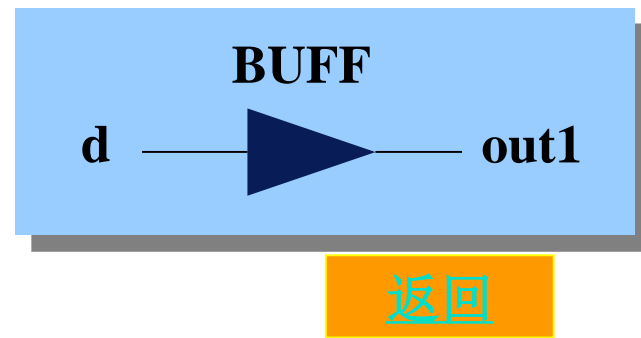
【例】在时钟沿触发的always块中，用reg型变量生成触发器

```
module rw1( clk, d, out1) ;  
    input clk, d ;  
    output out1 ;  
    reg out1 ;  
    always @(posedge clk) //沿触发  
        out1 <= d;  
endmodule
```



【例】使用电平触发，用reg型变量生成组合逻辑

```
module rw2( clk, d, out1);  
    input clk, d;  
    output out1;  
    reg out1;  
    always @(d) //电平触发  
        out1 <= d;  
endmodule
```



3.3 Verilog HDL常用语句

- 一、结构声明语句
- 二、赋值语句
- 三、条件语句
- 四、循环语句
- 五、语句的顺序执行与并行执行

- ❖ 语句是构成Verilog HDL程序不可缺少的部分。
- ❖ Verilog HDL的语句包括赋值语句、条件语句、循环语句、结构声明语句和编译预处理语句（本课程不学习）等类型，每一类语句又包括几种不同的语句。
- ❖ 有些语句属于顺序执行语句，有些语句属于并行执行语句。

Verilog HDL常用语句

赋值语句	门基元赋值语句	
	连续赋值语句	
	过程赋值语句	
块语句	begin_end 语句	
	fork_join 语句	Quartus II不支持
条件语句	if_else 语句	
	case 语句	
循环语句	forever 语句	
	repeat 语句	
	while 语句	
	for 语句	
结构声明语句	initial 语句	Quartus II不支持
	always 语句	
	task 语句	
	function 语句	
编译预处理语句	'define 语句	
	'include 语句	Quartus II不支持
	'timescale 语句	Quartus II不支持

一、结构声明语句

❖ 具有某种独立功能的电路，均在过程块中描述，**Verilog HDL**的任何过程模块都是放在结构声明语句中，结构声明语句分为**4**种：

- **initial**语句——沿时间轴只执行一次
- **always**语句——不断重复执行，直到仿真结束
- **task**语句——可在程序模块中的一处或多处调用
- **function**语句——可在程序模块中的一处或多处调用

1、**always**块语句

❖ 包含一个或一个以上的声明语句(如:过程赋值语句、任务调用、条件语句和循环语句等)，在仿真运行的全过程中，在定时控制下被**反复**执行。

规则

- ❖ **always**块通常带触发条件；
- ❖ 在**always**块中被赋值的只能是**register**型变量。
- ❖ 每个在仿真一开始便开始执行，当执行完块中最后一个语句，继续从**always**块的开头执行。

always块语句

格式

```
always @(敏感信号表达式)
begin
    // 过程赋值语句;
    // if语句, case语句;
    // for语句, while语句, repeat语句;
    // tast语句、function语句;
end
```

一个变量不
能在多个
always块中
被赋值!

错误的写法:

```
always @ (posedge clk)
    q=q+1;
always @ (negedge reset)
    q=0;
```

正确的写法:

```
always @ (posedge clk or
negedge reset)
begin
    if(!reset) q=0; //异步清零
    else q=q+1;
end
```


2、initial语句

- ❖ **initial**语句是面向模拟仿真的过程语句，通常不能被逻辑综合工具支持。**initial**块内的语句仅执行一次。

【例2.26】对各变量进行初始化。

格式

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

```
parameter size=16;
reg[3:0] addr;
reg reg1;
reg[7:0] memory[0:15];
initial
begin
    reg1 = 0;
    for(addr=0;addr<size;addr=addr+1);
        memory[addr]=0;
end
```

用途

- ❖ 在仿真的初始状态对各变量进行**初始化**；
- ❖ 在测试文件中**生成激励波形**（如时钟信号）作为电路的仿真信号。

3、task语句

- ❖ **task**语句用来由用户定义任务，任务类似高级语言中的子程序，用来单独完成某项具体任务，并可以被模块或其他任务调用。
- ❖ 当希望能够对多个信号进行一些运算并输出**多个**结果（即有多个输出变量）时，宜采用任务结构。
- ❖ 常常利用任务来帮助实现结构化的模块设计，将批量的操作以任务的形式独立出来，使设计简单明了，而且便于调试。

任务定义

```
task <任务名>;  
    端口声明语句;  
    数据类型声明语句;  
    实现逻辑功能的语句;  
endtask
```

注意无端口列表！

任务调用

```
<任务名> (端口1,端口2,.....);
```

端口名列表与任务定义中的I/O变量一一对应！

task语句使用注意事项

- 注1: 任务的定义与调用必须在一个**module**模块内!
- 注2: 任务被调用时, 需列出端口名列表, 端口名的**排序**与**类型**必须与任务定义中的**I/O变量**相一致!
- 注3: 一个任务可以调用其他任务和函数。

【例】任务的定义与调用。

任务定义

```
task my_task;  
  input a,b;  
  inout c;  
  output d,e;  
  
  .....  
  <语句>    //执行任务工作相应的语句  
  .....  
  c = foo1;  
  d = foo2;    //对任务的输出变量赋值  
  e = foo3;  
  
endtask
```

任务调用

```
my_task (v,w,x,y,z);
```

- 当任务启动时, 由v、w和x传入的变量赋给了a、b和c;
- 当任务完成后, 输出通过c、d和e赋给了x、y和z。

4、function语句

- ❖ **function**语句用来定义函数，函数的目的是通过返回一个用于某表达式的值，来响应输入信号。适于对不同变量采取同一运算的操作。
- ❖ 函数在模块内部定义，通常在本模块中调用，也能根据按模块层次分级命名的函数名从其他模块调用。而任务只能在同一模块内定义与调用！

函数定义

只能是input

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

缺省则返回1位
reg型数据

函数调用

<函数名> (<表达式1>, <表达式2>,)

与函数定义中的
输入变量对应！

function语句举例

函数在**综合**时被理解成具有独立运算功能的电路，每调用一次函数，相当于改变此电路的输入，以得到相应的计算结果。

【例2.29】 利用函数对一个8位二进制数中为0的位进行计数。

```
function[3:0] get0; //函数的定义，计算x中0的个数
input [7:0] x;      → 只有输入变量
reg[3:0] count; integer i;
begin count=0;
  for(i=0;i<=7;i=i+1) //循环核对x中的每一位
    if(x[i]==1'b0) count=count+1;
  get0 = count; //将运算结果赋给与函数同名的内部寄存器
endfunction
```

内部寄存器

→ 对应函数的输入变量

assign number = get0(rega); //对函数的调用

函数的调用是通过将函数作为调用函数的表达式中的**操作数**来实现的！

二、赋值语句

❖ 赋值语句分为3类:

1、门基元赋值语句（门元件例化）

基本逻辑门关键字 (门输出, 门输入1, 门输入2, ..., 门输入n);

- 基本逻辑门关键字是Verilog HDL预定义的逻辑门，包括 **and**、**or**、**not**、**xor**、**nand**、**nor**等；圆括弧中内容是被描述门的输出和输入信号。
- 例如，具有**a**、**b**、**c**、**d** 这4个输入和**y**为输出的与非门的门基元赋值语句为**nand (y,a,b,c,d);**；
该语句与**assign y = ! (a && b && c && d);**等效

2、连续赋值语句（assign语句）

用于对**wire**型变量赋值，是描述**组合逻辑**最常用的方法之一。

assign 赋值变量 = 表达式;

➤ 【例】 4输入与非门

assign y = ! (a && b && c && d);

- 连续赋值语句的“=”号两边的变量都应该是**wire**型变量。
- 在执行中，输出**y**的变化跟随输入**a**、**b**、**c**、**d**的变化而变化，反映了信息传送的连续性。

【例】 2选1多路选择器

```
module mux2_1(out,a,b,sel);
```

```
    input a,b,sel; output out;    //输入、输出信号默认为wire型变量
```

```
    assign out =( sel==0) ? a:b; //若sel为0，则out=a；否则out=b
```

```
endmodule
```

3、过程赋值语句

3、过程赋值语句

用于对**reg**型变量赋值，过程赋值语句出现在**initial**和**always**块语句中，有两种赋值方式：

➤ 阻塞（**blocking**）赋值方式：

赋值符号为**=**，如 **b = a** ；

赋值变量 = 表达式；

➤ 非阻塞（**non-blocking**）赋值方式：赋值符号为**<=**，如 **b <= a** ；

赋值变量 <= 表达式；

非阻塞赋值与阻塞赋值的区别

(1) 非阻塞赋值方式

always @(posedge clk)

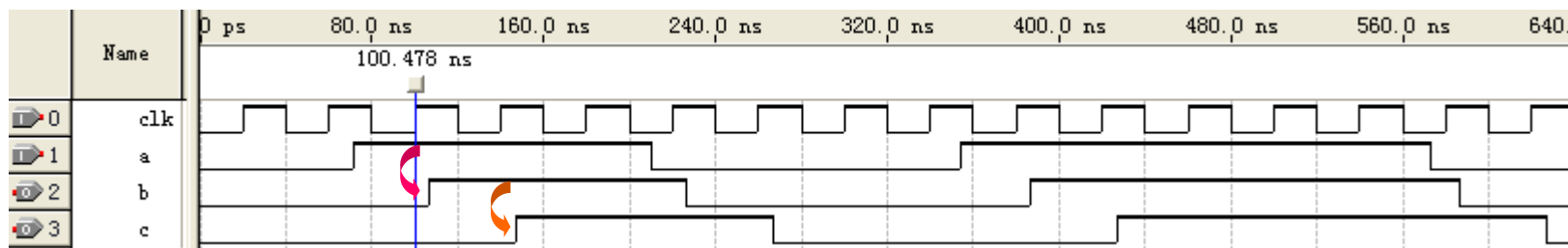
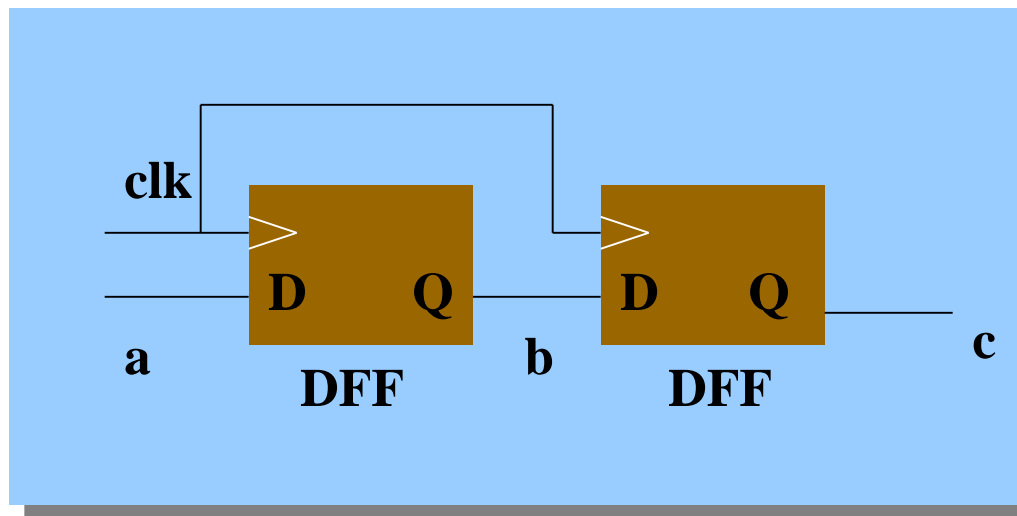
begin

b <= a;

c <= b;

end

非阻塞赋值在
块结束时才完
成赋值操作！



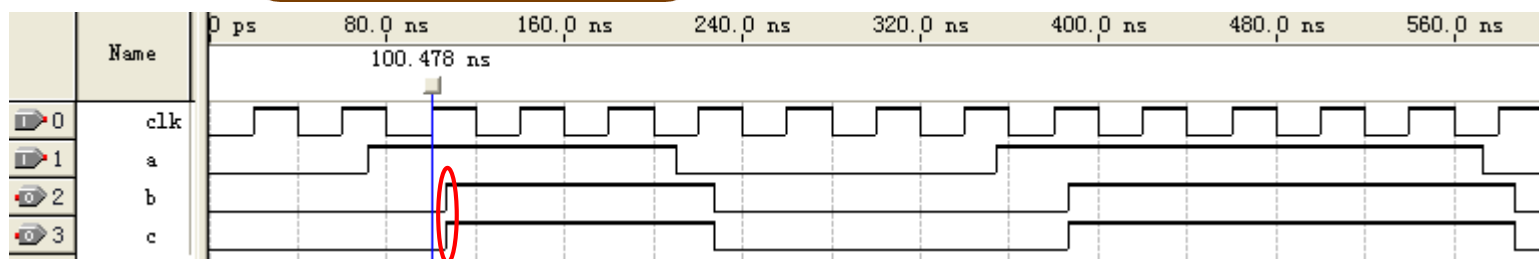
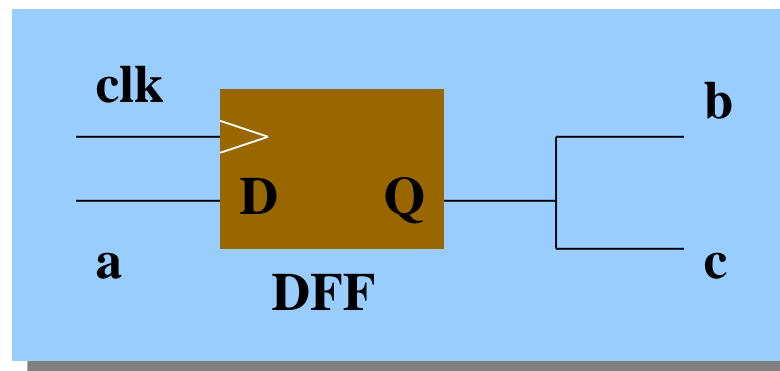
- **c**的值比**b**的值落后一个时钟周期！
- 若块内有多条赋值语句，则在块结束时同时赋值。
- 多条非阻塞赋值语句**并行**执行！

非阻塞赋值与阻塞赋值的区别（续）

(2) 阻塞赋值方式

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

阻塞赋值在**该语句**结束时就完成赋值操作！



- ◆ **c**的值与**b**的值一样！
- ◆ 在一个块语句中，如果有多条阻塞赋值语句，在前面的赋值语句没有完成之前，后面的语句就不能被执行，就像被阻塞了一样，因此称为**阻塞赋值方式**。
- ◆ 多条阻塞赋值语句**顺序**执行！

三、条件语句

条件语句分为两种：**if-else**语句和**case**语句；
它们都是顺序语句，应放在“**always**”块内！

对于每个判定
只有**两个**分支

1、**if-else**语句

- ❖ 判定所给条件是否满足，根据判定的结果（真或假）决定执行给出的两种操作之一。
- ❖ **if-else**语句有**3**种形式，格式与C语言中的**if-else**语句类似
 - 其中“表达式”为逻辑表达式或关系表达式，或一位的变量。
 - 若表达式的值为**0**、或**x**、或**z**，则判定的结果为“假”；若为**1**，则结果为“真”。
 - 执行的语句可为单句，也可为多句；多句时一定要用“**begin_end**”语句括起来，形成一个复合块语句。

if-else语句的表示方式

方式1（非完整性IF语句）：

```
if (表达式) 语句1;
```

方式3（多重选择的IF语句）：

适于对不同的条件，执行不同的语句

方式2（二重选择的IF语句）：

```
if (表达式1) 语句1;  
else        语句2;
```

```
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 语句n;
```

❖ else子句不能作为语句单独使用，它是if语句的一部分，必须与if配对使用！

❖ 允许一定形式的表达式简写方式，如：

➤ if(expression) 等同于 if(expression == 1)

➤ if(! expression) 等同于 if(expression != 1)

2、case语句

多分支选择语句

适于对**同一组**控制信号取不同的值时，输出取不同的值！

- ❖ 当敏感表达式取不同的值时，执行不同的语句。
- ❖ **功能**：当某个或某组（控制）信号取不同的值时，给另一个（输出）信号赋不同的值。常用于**多条件**译码电路（如译码器、数据选择器、状态机、微处理器的指令译码）！

case语句
的**3**种形式

case

casez

casex

case语句与if-else语句有什么区别呢？

case语句的语法格式

```
case (敏感表达式)
  值1: 语句1;
  值2: 语句2;
  ...
  值n: 语句n;
  default: 语句n+1;
endcase
```

❖说明:

- 其中“敏感表达式”又称为“控制表达式”，通常表示为控制信号的某些位。当有多个信号时，可用位拼接符将它们连接起来：
- 【例】 `case({D3,D2,D1,D0})`
- 值1~值n称为分支表达式，用控制信号的具体状态值表示，因此又称为常量表达式。
- default项可有可无，一个case语句里只能有一个default项！
- 值1~值n必须互不相同，否则矛盾。
- 值1~值n的位宽必须相等，且与控制表达式的位宽相同。

四、循环语句

❖ 循环语句用来控制语句的执行次数。分为4种：

- **for**语句——有条件的循环语句。通过3个步骤来决定语句的循环执行：
 - (1) 给控制循环次数的变量赋初值。
 - (2) 判定循环执行条件，若为假则跳出循环；若为真，则执行指定的语句后，转到第(3)步。
 - (3) 修改循环变量的值，返回第(2)步。
- **repeat**语句——连续执行一条语句n次
- **while**语句——执行一条语句直到某个条件不满足。首先判断循环执行条件表达式是否为真，若为真，则执行后面的语句或语句块，直到条件表达式不为真；若不为真，则其后的语句一次也不被执行！
- **forever**语句——无限连续地执行语句，可用**disable**语句中断！多用在**initial**块中，以生成时钟等周期性波形

1、for语句

功能：一般用途的循环语句，允许一条或更多的语句被重复地执行。

一般形式

for (表达式1;表达式2;表达式3) 语句

简单应用形式

for (循环指针 = 初值; 循环指针 < 终值; 循环指针 = 循环指针 + 步长值)

begin 执行语句; **end**

for语句比**while**语句简洁！

两条语句

规则：当**for**循环开始执行时，循环变量已赋予初值。在每一次循环执行之前（包括第一次），都必须检查表达式2（循环执行条件），若它为假（0、x或z），则立刻退出循环。而在每一次循环执行之后，都要使循环变量增值。

for语句举例

【例2.34】用for语句描述的7人投票表决器：若超过4人（含4人）投赞成票，则表决通过。

```
module vote7 ( pass,vote );  
    output pass;  
    input [6:0] vote;  
    reg[2:0] sum;    //sum为reg型变量，用于统计赞成的人数  
    integer i;      // 循环变量  
    reg pass;  
    always @(vote)  
    begin  
        sum = 3'b000;    //sum初值为0  
        for(i = 0;i<=6;i = i+1)    //for语句  
            if(vote[i]) sum = sum+1; //只要有人投赞成票，则 sum加1  
        if(sum >=3'd4) pass =1'b 1; //若超过4人赞成，则表决通过  
        else  
            pass =1'b 0;  
    end  
endmodule
```

将循环变量定义为整型

2、repeat语句

只有部分综合工具可以综合此语句!

- ❖ **功能**: 把一条或多条语句**连续**执行指定的次数。
- ❖ **规则**: 重复执行的次数由循环次数表达式的值决定, 若该值为**0**、**x**或**z**, 则不会重复执行。

格式

repeat (循环次数表达式) 语句

或

```
repeat (循环次数表达式)
begin
.....
end
```

执行语句为**多**条语句

3、while语句

- ❖ **功能**：有条件地执行一条或多条语句。只要循环执行条件表达式为真，则循环语句就重复执行！
- ❖ **规则**：首先判断循环执行条件表达式是否为真。若为真，则执行后面的语句或语句块；然后再回头判断循环执行条件表达式是否为真，若为真，再执行一次后面的语句；如此不断，直到条件表达式不为真。

格式

```
while (循环执行条件表达式)
begin
.....
end
```

while语句
通常用在测试文件中！

1. 首先判断循环执行条件表达式是否为真，若不为真，则其后的语句一次也不被执行！
2. 在执行语句中，必须有一条改变循环执行条件表达式的值的语句！
3. while语句只有当循环块有事件控制（即@（posedge clock））时才可综合！

while语句举例

【例2.37】用while语句对一个8位二进制数中值为1的位进行计数

```
module count1s_while ( count,rega,clk );
    output[3:0] count;
    input [7:0]  rega;
    input clk;
    reg[3:0]    count;
    always @(posedge clk)
        begin:count1
            reg[7:0] tempreg;           // 用作循环执行条件表达式
            count = 0;                  // count初值为0
            tempreg = rega;              // tempreg初值为rega
            while(tempreg)               // 若tempreg非0，则执行以下语句
                begin
                    if(tempreg[0]) count = count+1;
                                        // 只要tempreg最低位为1，则count加1
                    tempreg = tempreg >> 1; // 右移1位
                end
            end
        end
endmodule
```

如何用for语句
改写此程序呢?

改变循环执行条件表达式的值

4、forever语句

❖ **功能：**无条件连续执行**forever**后面的语句或语句块。

格式

```
forever  
begin  
.....  
end
```

forever语句
通常用在测试文件中！

- **forever**循环应包括定时控制或能够使其自身停止循环，否则循环将无限进行下去！
- 常用在测试文件中产生周期性的波形，作为**仿真激励**信号。
- 可综合性：尽管**Quartus II**支持该语句，但**一般情况下是不可综合的**！如果**forever**循环被**@(posedge clock)**形式的时间控制打断，则是可综合的。

五、语句的顺序执行与并行执行

❖ Verilog HDL中有顺序执行语句和并行执行语句之分。

1、语句的顺序执行

- 在 “**always**”模块内，对于阻塞赋值语句，逻辑按书写的顺序执行，若随意颠倒赋值语句的书写顺序，可能导致不同的结果！（见下页例子）。
- 而对于非阻塞赋值语句，是并发执行的。
- 注意阻塞赋值语句当本语句结束时即完成赋值操作！

语句的顺序执行举例

【例2.40】顺序执行模块1。

```
module serial1(q,a,clk);  
    output q,a;  
    input clk;  
    reg q,a;  
    always @(p  
        begin  
            q=~q; //阻塞赋值语句  
            a=~q;  
        end  
    endmodule
```

对前一时刻的q值取反

对当前时刻的q值反

a和q的波形反相！

【例2.41】顺序执行模块2。

```
module serial2(q,a,clk);  
    output q,a;  
    input clk;  
    reg q,a;  
    always @  
        begin  
            a=~q;  
            q=~a;  
        end  
    endmodule
```

对前一时刻的q值取反

对前一时刻的q值取反

a和q的波形完全相同！

2、语句的并行执行

- 多个“always”模块、“assign”语句、实例元件调用、“always”模块内的非阻塞赋值语句都是并行执行的！
- 它们在程序中的先后顺序对结果并没有影响。
- 【例2.42】将两条赋值语句分别放在两个“always”模块中，若颠倒两个“always”模块顺序，对仿真结果没有影响，同【例2.41】——q和a的波形完全一样。

【例2.42】 并行执行模块。

```
module parall1(q,a,clk);  
    output q,a;  
    input clk;  
    reg q,a;  
    always @(posedge clk)  
        begin  
            q=~q;  
        end  
    always @(posedge clk)  
        begin  
            a=~q;  
        end  
endmodule
```


2.5.4 不同抽象级别的Verilog HDL模型

- ❖ 用Verilog HDL描述的电路称为该设计电路的Verilog HDL模型。
- ❖ 一个复杂电路的完整Verilog HDL模型由若干个Verilog HDL模块构成，每个模块由若干的子模块构成——可分别用不同抽象级别的Verilog HDL描述。
- ❖ 在同一个Verilog HDL模块中可有多种级别的描述。
 - **系统级(system level)**: 用高级语言结构（如case语句）实现的设计模块外部性能模型；
 - **算法级(algorithmic level)**: 用高级语言结构实现的设计算法模型（写出逻辑表达式）；
 - **RTL级(register transfer level)**: 描述数据在寄存器之间流动和如何处理这些数据的模型；
 - **门级(gate level)**: 描述逻辑门（如与门、非门、或门、与非门、三态门等）以及逻辑门之间连接的模型；
 - **开关级(switch level)**: 描述器件中三极管和储存节点及其之间连接的模型。