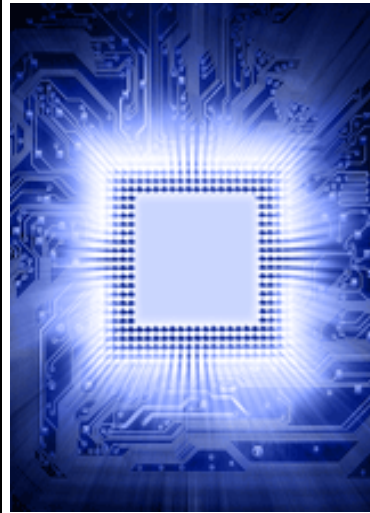


## 5.2

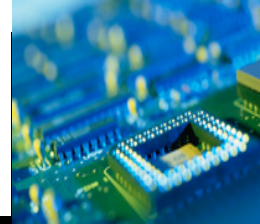
# Verilog 硬件描述语言

---





# 主要内容



1. 引言
2. Verilog HDL程序结构
3. 逻辑系统、网格、变量和常量
4. 向量和操作符
5. 数组
6. 逻辑操作符和表达式
7. 编译器指令
8. 结构形式的设计元素
9. 数据流形式的设计元素
10. 行为形式的设计元素
11. 函数和任务
12. 时间尺度
13. 模拟
14. 测试平台
15. 时序逻辑设计的Verilog特性
16. 综合
17. 不同抽象级别的Verilog HDL模型
18. 设计技巧



# 1 引言



- 硬件描述语言Hardware Description Language
  - 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。
- 与原理图设计方法相比：
  - 更适于描述大规模的系统
  - 在抽象的层次上描述系统的结构与功能
- 采用HDL的优点：
  - 语言的公开可利用性
  - 设计与工艺的无关性
  - 宽范围的描述能力——系统级、算法级、RTL级、门级、开关级
  - 便于组织大规模系统的设计
  - 便于设计的复用、交流、保存与修改

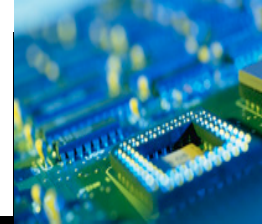


# 1 引言



- 两种使用最广泛的HDL：Verilog和VHDL
  - VHDL 的英文全名是VHSIC (Very High Speed Integrated Circuits) Hardware Description Language ，于1983年由美国国防部提出，由IEEE进一步发展，并在1987年12月作为IEEE-STD-1076标准发布，1993年被更新为IEEE-STD-1164 标准，目前已被广泛应用。
  - 全方位HDL，包括从系统到电路的所有设计层次。
  - 支持结构、数据流和行为3种描述形式的混合描述。

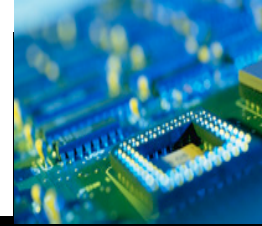
# 1 引言



- 什么是Verilog HDL
- Verilog HDL是一种用于数字逻辑电路设计的硬件描述语言（Hardware Description Language），可以用来进行数字电路的仿真验证、时序分析、逻辑综合。
  - 用Verilog HDL描述的电路设计就是该电路的Verilog HDL模型。
  - Verilog HDL 既是一种行为描述语言也是一种结构描述语言。
- 既可以用电路的功能描述，也可以用元器件及其之间的连接来建立Verilog HDL模型。



# 1 引言



## Verilog HDL的发展历史

- 1983年，由GDA（GateWay Design Automation）公司的Phil Moorby首创；
- 1989年，Cadence公司收购了GDA公司；
- 1990年，Cadence公司公开发表Verilog HDL；
- 1995年，IEEE制定并公开发表Verilog HDL1364-1995标准；
- 2001年，标准Verilog-2001（IEEE1364-2001）公开发表。
- 2005年，System Verilog 成为IEEE标准（IEEE STD 1800），并且被视为最有前途的高级验证语言之一。



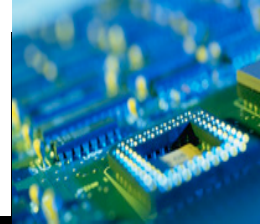
# 1 引言



- **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



# 1 引言



不同层次的Verilog HDL抽象

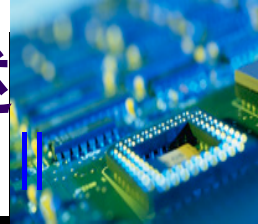
- Verilog HDL模型可以是实际电路的不同级别的抽象。抽象级别可分为五级：
  - 系统级(system level): 用高级语言结构（如case语句）实现的设计模块外部性能模型；
  - 算法级(algorithmic level): 用高级语言结构实现的设计算法模型（写出逻辑表达式）；
  - RTL级(register transfer level): 描述数据在寄存器之间流动和如何处理这些数据的模型；
  - 门级(gate level): 描述逻辑门（如与门、非门、或门、与非门、三态门等）以及逻辑门之间连接的模型；
  - 电路级(switch level): 描述器件中三极管和储存节点及其之间连接的模型。





# Verilog 在2个抽象级上建模

## 混合描述



### 行为级

- 用功能块之间的数据流对系统进行描述
- 在需要时在函数块内进行逻辑建模

### RTL级/功能级

- 用功能块内部或功能块之间的数据流和控制信号

### 结构级/门级

- 用基本单元(**primitive**)或低层元件(**component**)的连接来描述系统以得到更高的精确性，特别是时序方面。
- 在综合时用特定工艺和低层元件将**RTL**描述映射到门级网表

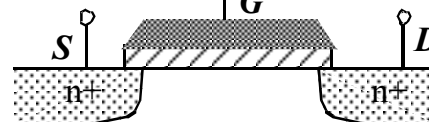
系统级和算法级

RTL级

逻辑门级

电路级

版图级



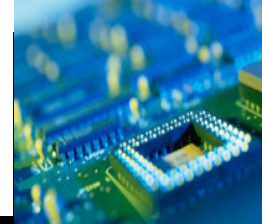
行为描述  
(Behavior)

+

数据流描述  
(Data Flow)

结构描述  
(Structure)

# 1 引言



- Verilog HDL的特性

在某些语法结构上  
与C语言非常相似!

- 设计过程可以按层次进行分解;
- 每个设计元素都有定义好的接口（用于元素之间的联结）和简明的功能规格说明（用于模拟）;
- 功能规格既可利用行为算法亦可利用定义元素操作的实际硬件结构;
- 并发性、定时和时钟都可以模型化
- 可以模拟一个设计的逻辑操作过程和定时情况;
- Verilog既是一种行为描述的语言也是一种结构描述语言。

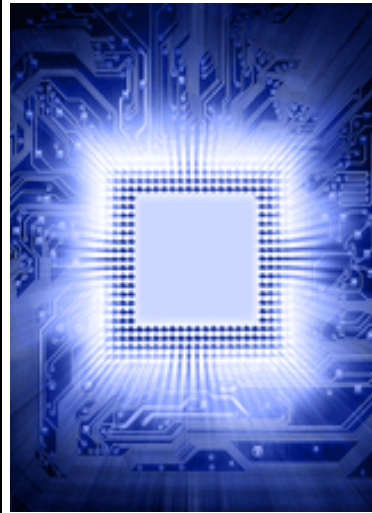


# Verilog、VHDL和C



- Verilog与VHDL的主要区别：
  - VHDL侧重于系统级描述，从而更多的为系统级设计人员所采用
  - Verilog侧重于电路级描述，从而更多的为电路级设计人员所采用
- Verilog HDL与 C语言存在本质上的区别：
  - Verilog是一种硬件语言，最终是为了产生实际的硬件电路或对硬件电路进行仿真
  - C语言是一种软件语言，是控制硬件来实现某些功能
- SystemVerilog and SystemC：面向SOC

## 2 程序结构





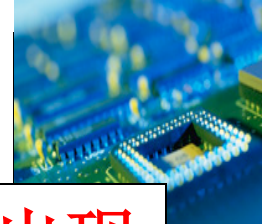
# 1 Verilog程序结构



- Verilog进行设计和编程的基本单元是模块`module`：  
包含说明和语句的一个文本文件。
- `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
- 一个verilog源文件中可以有多个模块，且对排列顺序不做要求。
- 模块通过输入和输出端口被高层的模块调用，但隐藏了内部的实现细节。

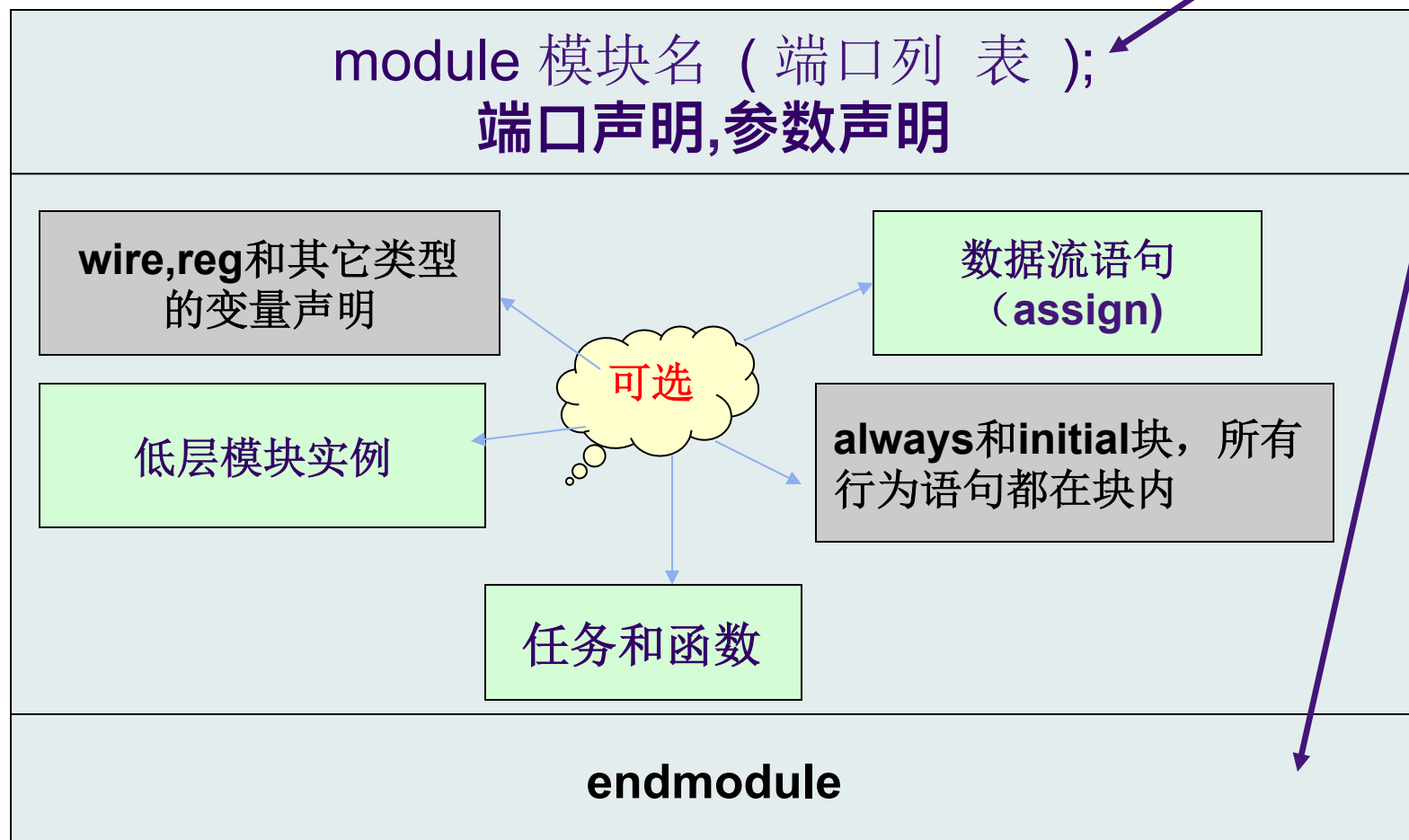


# 1 Verilog程序结构



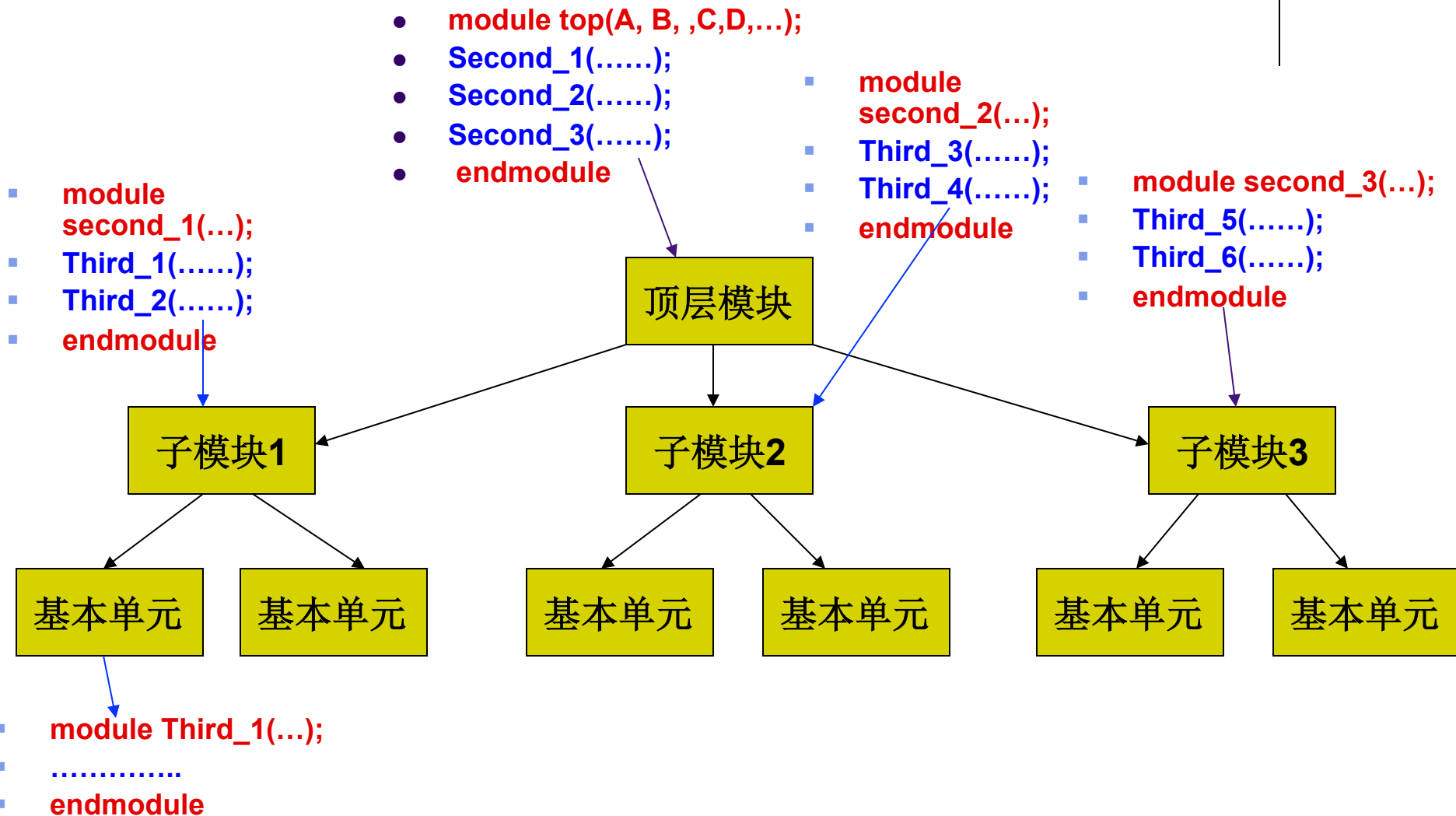
## 模块内部构成

必须出现





# 模块层次图





# Verilog 模块说明的语法



1 端口定义

2 I/O说明

3 信号说明

4 函数、  
任务说明

5 并发语句  
功能描述

```
module module-name (port-name, ...);
```

```
    input declarations
```

```
    output declarations
```

```
    inout declarations
```

```
    net declarations
```

```
    variable declarations
```

```
    parameter declarations
```

```
    function declarations
```

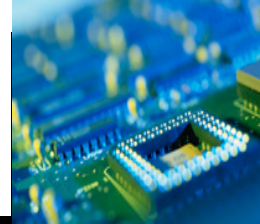
```
    task declarations
```

```
    concurrent statements
```

```
endmodule
```

实例语句  
连续赋值语句  
Always程序段





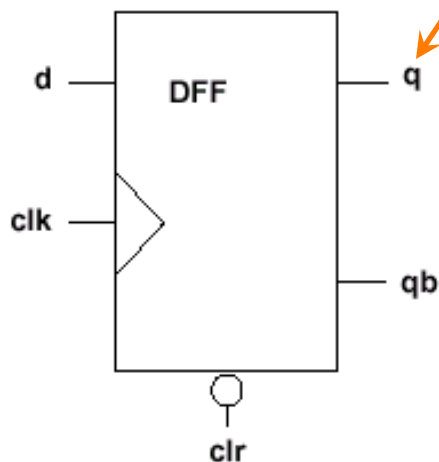
- 端口(Terminal): 模块通过端口与外部通信
  - 端口是模块与外界环境交互的接口。例如IC芯片的输入、输出引脚就是它的端口。
  - 对于外部环境来讲，模块内部是不可见的，对模块的调用(实例引用)只能通过其端口进行。
  - 这种特点为设计者提供了很大的灵活性：只要接口保持不变，模块内部的修改并不会影响到外部环境。
- 模块端口等价于芯片的管脚 (pin)
- 指定信号/端口的方向
  - input: 输入到模块的信号。
  - output: 由模块输出的信号。
  - inout: 该信号既可作为输入又可作为输出。用于三态输入输出引脚



# 端口列表和端口声明



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*

也可以采用类似ANSI C格式来声明端口

- **module D\_FF (input d, clk, clr, output reg q,qb);**
- .....  
.....
- **endmodule**



# 保留字

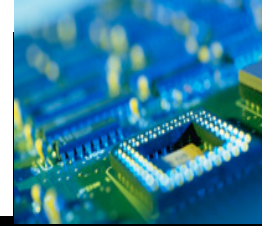


## 保留字reserved words/关键字keywords

- Verilog定义的特殊字符，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）。
- 用小写字母定义
  - always and assign begin buf buf if0 bufif1 case casex casez cmos deassign default defparam disable edge else end endcase endmodule endfunction endprimitive endspecify endtable endtask event for force forever fork function highz0 highz1 if ifnone initial inout input integer join large macrmodule medium module
  - nand negedge nmos nor not notif0 notif1 or output parameter pmos posedge primitive pull0 pull1 pullup pulldown rcmos real realtime reg release repeat rmos rpmos rtran rtranif0 rtranif1 scalared small specify specparam strong0 strong1 supply0 supply1
  - table task time trantranif0 tranif1 tri tri0 tri1 triand trior trireg vectored wait wand weak0 weak1 while wire wor xnor xor

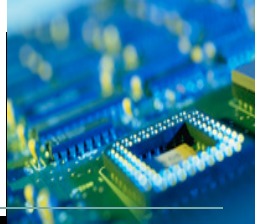


# 标识符



标识符identifier : 标识符不能与关键字同名!

- 任何用Verilog HDL语言描述的“东西”都通过其名字来识别，这个名字被称为标识符。如源文件名、模块名、端口名、变量名、常量名、实例名等。
- 标识符以字母或下划线开头，其中可以包含由字母、数字、下划线和美元\$符号。
- 在Verilog HDL中变量名是区分大小写的!
- 通过有序的界(range)指定[msb:lsb]来说明多位或向量信号。
  - [7:0], [0:7], 13:20]
- 合法的名字：
  - A\_99\_Z, Reset, \_54MHz\_Clock\$, Module
- 不合法的名字: 123a, \$data, module, 7seg.v



# 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** bits)          unsized decimal (zero-extended to 32

'H83a	unsized hexadecimal (zero- extended to 32 bits)
-------	---

**8'b1100 0001**      8-bit binary

<b>64'hff01</b> bits)	64-bit hexadecimal (zero- extended to 64 bits)
--------------------------	--

9 ' 017 9-bit octal

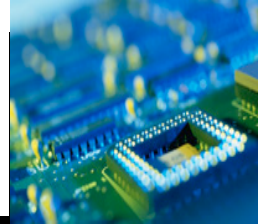
**32'bz01x**                      Z-extended to 32 bits

**3'b1010 1101**      3-bit number, truncated to 3'b101

### 6.3 decimal notation



# 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 $\times 10^{\text{指数}}$



# 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"**



# 延时说明 #



- “#”用于说明过程(**procedural**)语句和门的实例的延时，但不能用于模块的实例化。

```
module MUX2_1 (out, a, b, sel) ;  
  output out ;  
  input a, b, sel ;  
    not #1 not1( sel_, sel);  
    and #2 and1( a1, a, sel_);  
    and #2 and2( b1, b, sel);  
    or #1 or1( out, a1, b1);  
endmodule
```

- 门延时有很多类名字：门延时(**gate delay**)，传输延时(**propagation delay**)，固有延时(**intrinsic delay**)，对象内在延时(**intra-object delay**)





# 空白符和注释



```
module MUX2_1 (out, a, b, sel);
```

格式自由

```
// Port declaration
```

← 单行注释  
到行末结束

```
output out;
```

```
input sel, // control input
```

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。

```
    b, /* data inputs */
```

```
a;
```

```
/*
```

```
    The netlist logic selects input a' when
```

← 多行注释，在/\* \*/内

```
    sel = 0 and it selects "b" when sel =
```

```
    1.
```

```
*/
```

```
    not (sel_, sel);
```

```
    and (a1, a, sel_), (b1, b, sel); //
```

What does this

```
// line do?
```



# 基本结构



## 简单的Verilog HDL例子

[例] 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;  
    input cin;  
    assign {cout,sum}=a+b+cin;  
endmodule
```

端口定义

I/O说明

// 输入端口声明

功能描述

- 整个Verilog HDL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!



# Verilog HDL基本结构



- 小结:

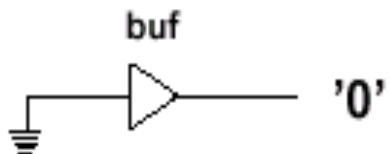
- Verilog HDL程序是由模块构成的。每个模块嵌套在module和endmodule声明语句中。模块是可以进行层次嵌套的。
- 每个Verilog HDL源文件中只准有一个顶层模块，其它为子模块。
- 每个模块要进行端口定义，并说明输入输出端口，然后对模块的功能进行行为逻辑描述。
- 程序书写格式自由，一行可以写几个语句，一个语句也可以分多行写。
- 除了endmodule语句、begin\_end语句和fork\_join语句外，每个语句和数据定义的最后必须有分号。
- 可用/\*.....\*/和//...对程序的任何部分作注释。加上必要的注释，以增强程序的可读性和可维护性。



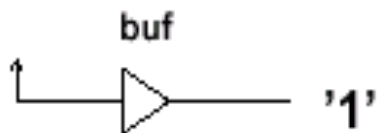
## 2 逻辑系统、网格、变量和常量



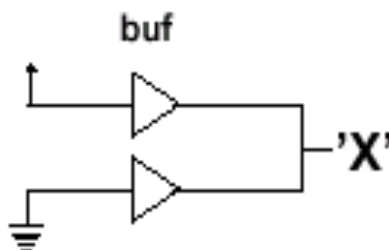
- 1码位宽的信号只能取4种可能值之一：



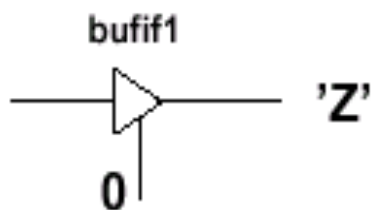
0: 逻辑0或假, Low, False, Logic Low, Ground, VSS, Negative Assertion



1: 逻辑1或真, High, True, Logic High, Power, VDD, VCC, Positive Assertion



x: 未知逻辑值, Occurs at Logical Which Cannot be Resolved Conflict



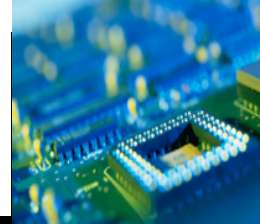
z: 高阻（三态逻辑中的高阻态）, HiZ, High Impedance, Tri- Stated



# 数据类型



- 数据类型是用来表示数字电路中的数据存储和传送单元。
- 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)



- Verilog有两类信号：网格和变量。
- 网格/线网(net)对应于物理电路中的连线。表示结构实体（如门）之间的物理连接。
- 常见的网格类型：
  - wire, tri: 连线类型
  - wor, trior: 具有线或特性的连线
  - wand, triand: 具有线与特性的连线)
  - tri1, tri0: 上拉电阻和下拉电阻
  - supply1, supply0: 电源（逻辑1）和地（逻辑0）
- 网格说明有两个用途：
  - 指定输入输出端口的网格类型；
  - 说明将要在模块内的结构描述中建立连通性的信号。

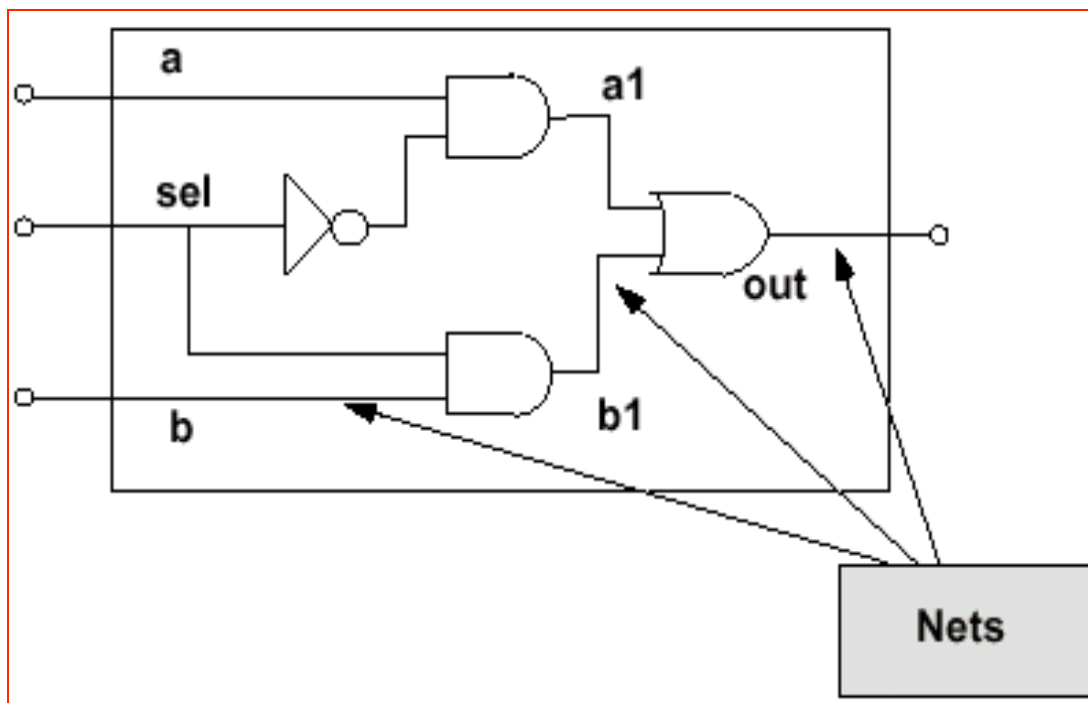


# 网格net（线网）



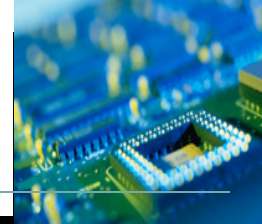
**net**需要被持续的驱动，驱动它的可以是门和模块。

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**门的输入信号变化时将传输到线网**net**上。





## 网格类的类型



- 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, <b>trior</b>	多驱动源线或
wand, <b>triand</b>	多驱动源线与
<b>triereg</b>	能保存电荷的net
<b>tri1, tri0</b>	无驱动时上拉/下拉



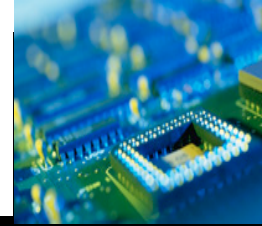
- 没有声明的net的缺省类型为 1 位(标量)wire类型。但这个缺省类型可由下面的编译指导改变:

```
`default_nettype <nettype>
```





# 网格类的类型



- **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。例如，可以用**tri**类型表示一个**net**有多个驱动源。或者将一个**net**声明为**tri**以指示这个**net**可以是高阻态**Z**(**high-impedance**)。可推广至**wand**和**triand**、**wor**和**trior**
- **wand**、**wor**有线逻辑功能；与**wire**的区别见下页的表。
- **triereg**类型很象**wire**类型，但**triereg**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>
```

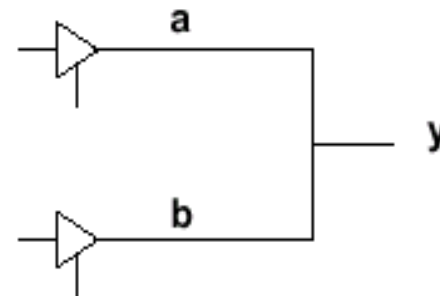
**nettype**不能是**supply1**和**supply0**。



# 网格类在发生逻辑冲突时的决断



- **Verilog**有预定义的决断函数
- 支持与工艺无关的逻辑冲突决断
  - **wire-and**用于集电极开路电路
  - **wire-or**用于射极耦合电路



Wire/Tri

$\begin{smallmatrix} b \\ a \end{smallmatrix}$	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

y

Wand/Triand

$\begin{smallmatrix} b \\ a \end{smallmatrix}$	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

y

Wor/Trior

$\begin{smallmatrix} b \\ a \end{smallmatrix}$	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

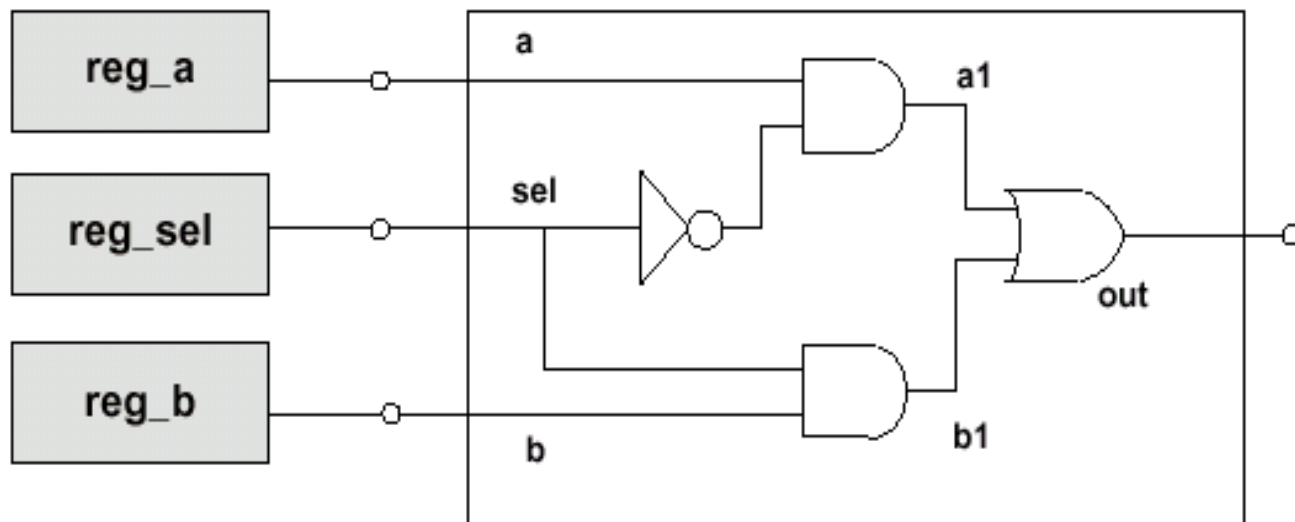
y



## (2)寄存器类 (register)

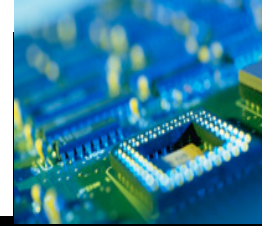


- 寄存器类型在赋新值以前保持原值
- 寄存器类型大量应用于行为模型描述及激励描述。在下面的例子中，**reg\_a**、**reg\_b**、**reg\_sel**用于施加激励给**2:1**多路器。
- 用行为描述结构给寄存器类型赋值。给**reg**类型赋值是在过程块中。





# 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	可定义的可符号整数变量，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

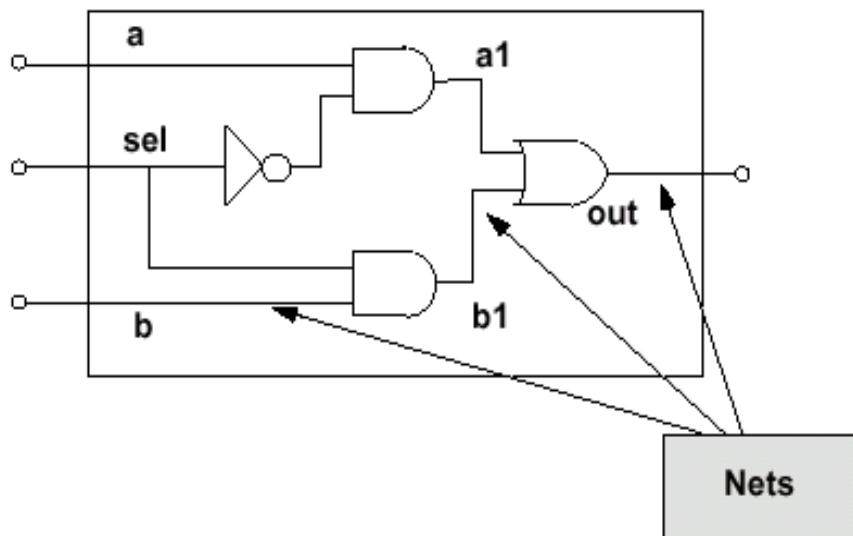
- 不要混淆寄存器数据类型与结构级存储元件，如udp\_dff



### (3) Verilog中net和register区别



- 网格类型用于对结构化器件之间的物理连线的建模。主要有**wire** 和**tri** 两种。
- 由于网格类型代表的是物理连接线，因此它不存贮逻辑值，必须由器件所驱动。
- 当一个**wire** 类型的信号没有被驱动时，缺省值为**Z**（高阻）。
- 信号没有定义数据类型时，缺省为**wire** 类型。





## (3) Verilog中net和register区别



### 寄存器（Register）

- 寄存器类型通常用于对存储单元的描述，如D型触发器、ROM等。存储器类型的信号当在某种触发机制下分配了一个值，在分配下一个值之时保留原值。 **reg** 是最常用的寄存器类型，表示无符号整数变量。
- 但必须注意的是，**reg** 类型的变量，**不一定是存储单元**，如在**always** 语句中进行描述的必须用**reg** 类型的变量。
- 寄存器类型的值可取负数，但若该变量用于表达式的运算中，则按无符号类型处理



## (4) 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

- **举例:**

```
wire a;
```

```
wand w; // 一个标量wand类型net
```

```
tri [15: 0] busa; // 16位三态总线
```

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



# Verilog中net和register声明语法



## 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

## ● 举例:

`reg a;` //一个标量寄存器

`reg [3: 0] v;` // 从MSB到LSB的4位寄存器向量

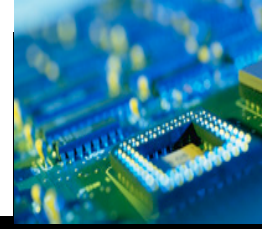
`reg [7: 0] m, n;` // 两个8位寄存器

`integer A, B, C;` //3个整数型寄存器





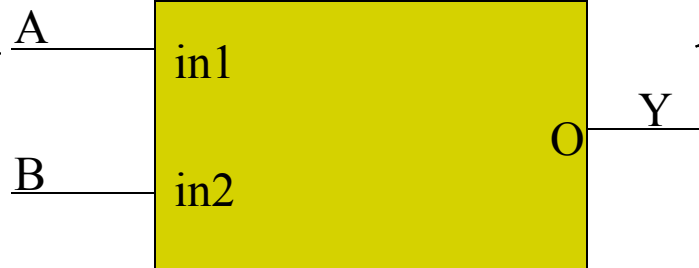
## (5)端口的数据类型选择



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
```

在过程块中只能给**register**类型赋值

```
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。



## (5)端口的数据类型选择



- a. 输入端口
  - 从模块内部来讲，输入端口必须为线网 (net) 数据类型；
  - 从模块外部来看，输入端口可以连接到线网 (net) 或 reg 数据类型的变量。
- b. 输出端口
  - 从模块内部来讲，输出端口可以为线网 (net) 或 reg 数据类型；
  - 从模块外部来看，输出端口必须连接到线网 (net) 数据类型的变量。
- c. 输入/输出端口
  - 从模块内部来讲，输入/输出端口必须为线网 (net) 数据类型；
  - 从模块外部来看，输入/输出端口必须连接线网 (net) 数据类型的变量。
- d. 位宽匹配
  - 在进行调试或模块调用时，verilog 允许模块内、外位宽不同，一般情况下编译器会给出警告。
- e. 未连接端口
  - Verilog 允许模块实例端口保持未连接状态。



## 例：端口的互连



- //D 触发器
- ~~module D\_FF (d, clk,clr,q,qb);~~
- ~~output s1,qb;~~
- ~~input d,clk,clr;~~
- ~~//调用D\_FF模块，这里 qb 值保存~~
- ~~命名为dff0~~
- ~~D\_FF dff0 (a,~~
- ~~b,c,s1,s2);~~
- ~~endmodule~~
- endmodule

- //D 触发器
- module Top
- reg a,b;
- wire c,s1,s2;
- //调用D\_FF模块，这里命名
- 名为dff0
- D\_FF dff0 (a, b,c,s1,s2);
- .....
- endmodule



## 4.参数 (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,  
              file = "/ usr1/  
jdough/ design/ mem_ file. dat";  
    . . .  
    wire [p1: 0] w1; // A wire  
declaration using parameter  
    . . .  
endmodule
```

注意：参数file不是string，而是一个整数，其值是所有字母的扩展ASCII值。若file="AB"，则file值为8'h4142。用法：

**\$fopen(file);**

**\$display("%s", file);**



## (5)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

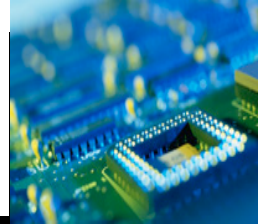
例如:

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

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



# 存储器寻址(Memory addressing)



- 存储器元素可以通过存储器索引 (**index**)寻址, 也就是给出元素在存储器的位置来寻址。

**mem\_name** [**addr\_expr**]

- Verilog**不支持多维数组。也就是说只能对存储器字进行寻址, 而不能对存储器中一个字的位寻址。

```
module mems;  
  reg [8: 1] mema [0: 255]; // declare memory called mema  
  reg [8: 1] mem_word;      // temp register called mem_  
  word  
  . . .  
  initial  
    begin  
      $displayb( mema[5]);           //显示存储器中第6个字的  
      mem_word = mema[5];  
      $displayb( mem_word[8]);       // //显示第6个字的有效位  
    end
```

若要对存储器字的某些位存取, 只能通过暂存器传递



## 3 向量和操作符



- 向量vector
  - Verilog允许各个1比特信号分组结合在一起。
  - 网格、常量、变量都可以定义为向量。
  - 向量使用有序的界[msb,lsb]来定义。
  - 如： `reg [15:0] word1,word2;`
  - 位选择语法：方括号和常数或子界范围来选择。
  - 如 `word1[5],word1[6:2]`
  - 串接concatenation: `{}, {2`b00,2`b11}`
  - 复制replication: `n{}, {2{word1},word2}`
  - 算术和移位操作



# 运算符及表达式



一、算术运算符

二、逻辑运算符

三、位运算符

四、关系运算符

五、等式运算符

六、缩减运算符

七、移位运算符

八、条件运算符

九、位拼接运算符

十、运算符的优先级





# 运算符及表达式



## ● 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
- 等式运算符
- 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、等式运算符，逻辑、位运算符的大部分
  - 三目运算符——带三个操作数  
条件运算符



# 运算符及表达式



## 一、算术运算符

双目运算符

算术运算符	说明
+	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！
- %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！

除法和取模操作一般都不能被综合，除非除数是2的幂

[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**。



# 运算符及表达式



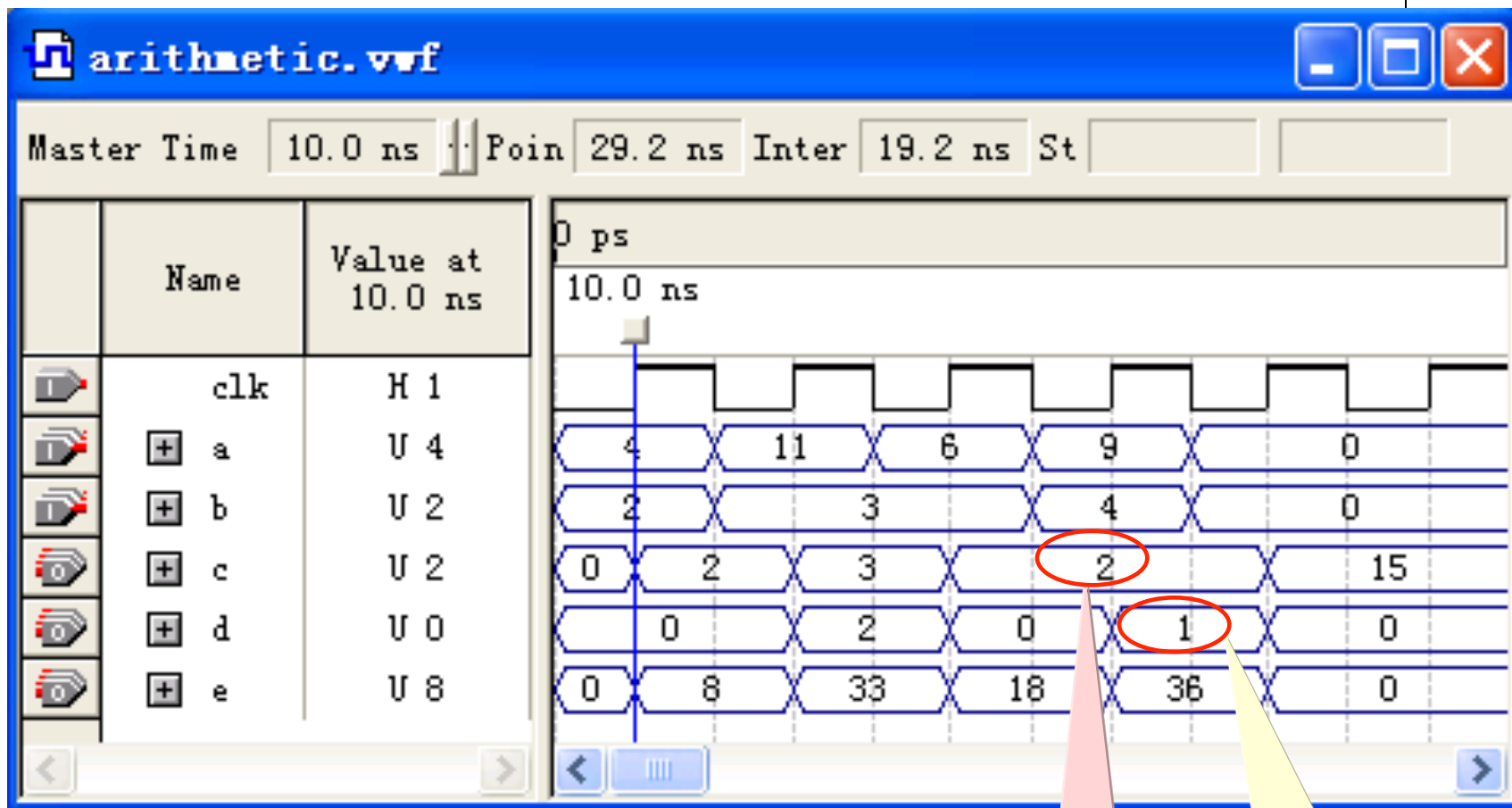
## ● [例] 除法和求模运算的区别

注意/和%的区别!

```
abc arithmetic.v
1 // 除法和求模运算的区别
2 module arithmetic(clk,a,b,c,d,e) ;
3     output [3:0]  c,d;
4     output [7:0]  e;
5     input  [3:0]   a,b;
6     input          clk;
7     reg [3:0]      c,d;
8     reg[7:0]       e;
9     always @ (posedge clk)
10         begin
11             c=a/b; //整数除法运算时，结果值略去小数部分，只取整数部分！
12             d=a%b; //求余数
13             e=a*b; //乘法
14         end
15     endmodule
```



# 运算符及表达式



arithmetric.v

$$9/4 = 2$$

$$9\%4 = 1$$



# 运算符及表达式



## 二、逻辑运算符

- 逻辑运算符把它的操作数当作**布尔变量**:
  - **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)。

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

❖ 进行逻辑运算后的结果为**布尔值** (为1或0或x) !



# 运算符及表达式



- “&&”和 “||”的优先级除高于条件运算符外，低于关系运算符、等式运算符等几乎所有运算符；
- 逻辑非 “!” 优先级最高。
- [例]  $(a > b) \&\& (b > c)$  可简写为:  $a > b \&\& b > c$   
 $(a == b) || (x == y)$  可简写为:  $a == b || x == y$

y

$(a == b) || (x == y)$

可简写为:

$a == b || x == y$

为提高程序的可读性，明确表达各运算符之间的优先关系，建议使用**括号**！



# 运算符及表达式



## 三、位运算符

单目运算符

双目运算符

位运算符	说明
$\sim$	按位取反
$\&$	按位与
$ $	按位或
$\wedge$	按位异或
$\wedge \sim, \sim \wedge$	按位同或

- 位运算其结果与操作数位数相同。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运算。
- 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  $A \& B = (5'b11001) \& (5'b00101) = 5'b00001$



# 运算符及表达式



## ● [例] &&运算符和&（按位与）的区别

```
logic_demo.v - Text Editor
// Logic Operators and bit Operators.

module logic_demo(outc,outd,a,b) ;
    output        outc;
    output[3:0]    outd;
    input  [3:0]    a,b;
    assign outc = a&&b;
    assign outd = a&b;
endmodule
```

&&运算的结果为1位的逻辑值

/\* Logic and operation \*/

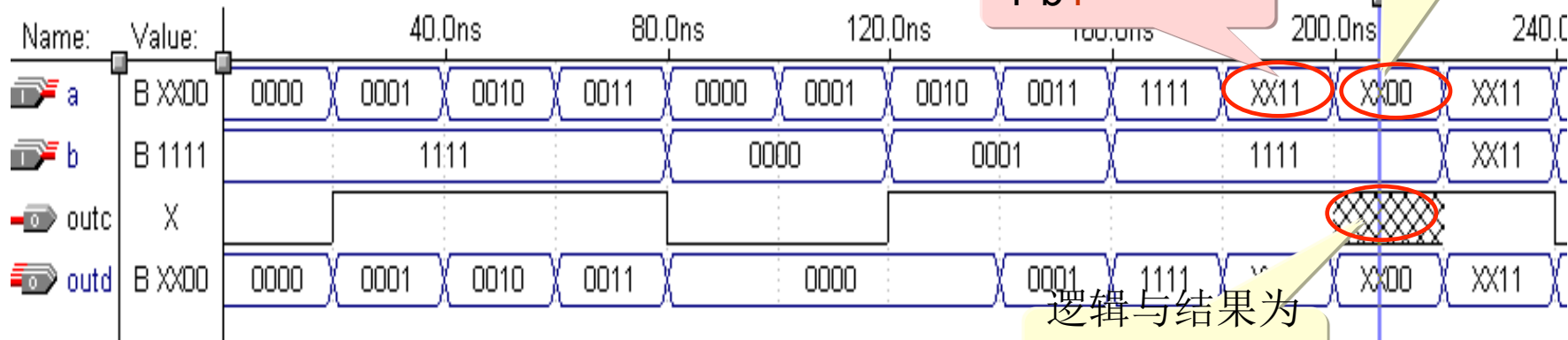
/\* bit and operation

被认为是

1'bX

被认为是

1'b1



逻辑与结果为

1'bX





# 运算符及表达式



## 四、关系运算符

双目运算符

关系运算符	说明
<	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，若关系为真，则返回值为1；若声明的关系为假，则返回值为0；若某操作数为不定值x，则返回值为x。
- 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!



# 运算符及表达式



## 五、等式运算符

双目运算符

MAX + PLUS II和  
Quartus II都不支持!

等式运算符	说明
==	等于
!=	不等于
===	全等
!==	不全等

- 运算结果为1位的逻辑值1或0或x。
- 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。
- 所有的等式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”。



# 运算符及表达式



“==”的真值表

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

等于运算的结果  
可能为1或0或x

“===”的真值表

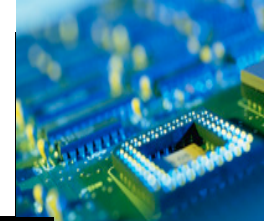
===	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

全等于运算的  
结果只有1或0

- [例] `if(A == 1'bx) $display("AisX");` //当A为不定值时，式 `(A == 1'bx)` 的运算结果为x，则该语句不执行  
`if(A === 1'bx) $display("AisX");` //当A为不定值时，式 `(A === 1'bx)` 的运算结果为1，该语句执行



# 运算符及表达式



## 六、缩减运算符

单目运算符

注意缩减运算符和  
位运算符的区别!

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

- 运算法则与位运算符类似，但运算过程不同！
- 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、非运算，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]`



# 运算符及表达式



## 七、移位运算符

单目运算符

移位运算符	说明
>>	右移
<<	左移

只有当右操作数为常数时MAX + PLUS II支持!

- 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。

- [例]  $4'b1001 \gg 3 = 4'b0001$ ;  $4'b1001 \gg 4 = 4'b0000$   
 $4'b1001 \ll 1 = 5'b1001$ ;  $4'b1001 \ll 2 = 6'b100100$ ;  
 $1 \ll 6 = 32'b1000000$

右移位不变,  
但右移的数据  
会丢失!

左移会扩充位数!

❖ 将操作数右移或左移 $n$ 位, 相当于将操作数除以或乘以 $2^n$ 。



# 运算符及表达式

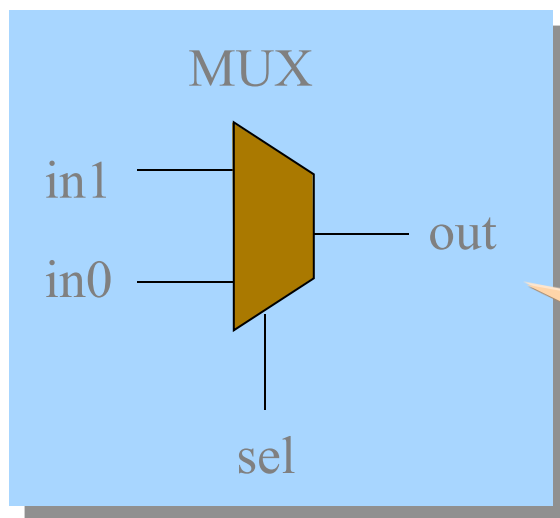


## 八、条件运算符

三目运算符

当条件为真，信号取表达式1的值；为假，则取表达式2的值。

- 条件运算符为 `?` :
- 用法: 信号 = 条件? 表达式1: 表达式2
- [例] 数据选择器 `assign out = sel? in1:in0;`



`sel=1时out=in1;`  
`sel=0时out=in0`



# 运算符及表达式



## 九、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}

- 例如在进行加法运算时，可将进位输出与和拼接在一起使用。

- [例1] 

```
output [3:0] sum;           //和
output cout;               //进位输出
input[3:0] ina,inb;
input cin;
assign {cout,sum} = ina + inb +cin; //进位与和拼接在一起
```

- [例2] 

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



# 运算符及表达式



用于表示重复的表达式  
必须为**常数**表达式！

- 可用**重复法**简化表达式，如： $\{4\{w\}\}$  // 等同于  $\{w, w, w, w\}$
- 还可**嵌套方式**简化书写，如：  
 $\{b, \{3\{a, b\}\}\}$  // 等同于  $\{b, \{a, b\}, \{a, b\}, \{a, b\}\}$ ，也等同于  $\{b, a, b, a, b, a, b\}$

❖ 在位拼接表达式中，不允许存在没有指明位数的信号，必须指明信号的位数；若未指明，则**默认为32位的二进制数**！

❖ 如  $\{1, 0\} = 64'h000000001\_000000000$ ，  
注意  $\{1, 0\}$  不等于  $2'b10$





# 运算符及表达式



## 十、运算符的优先级

### 运算符的优先级

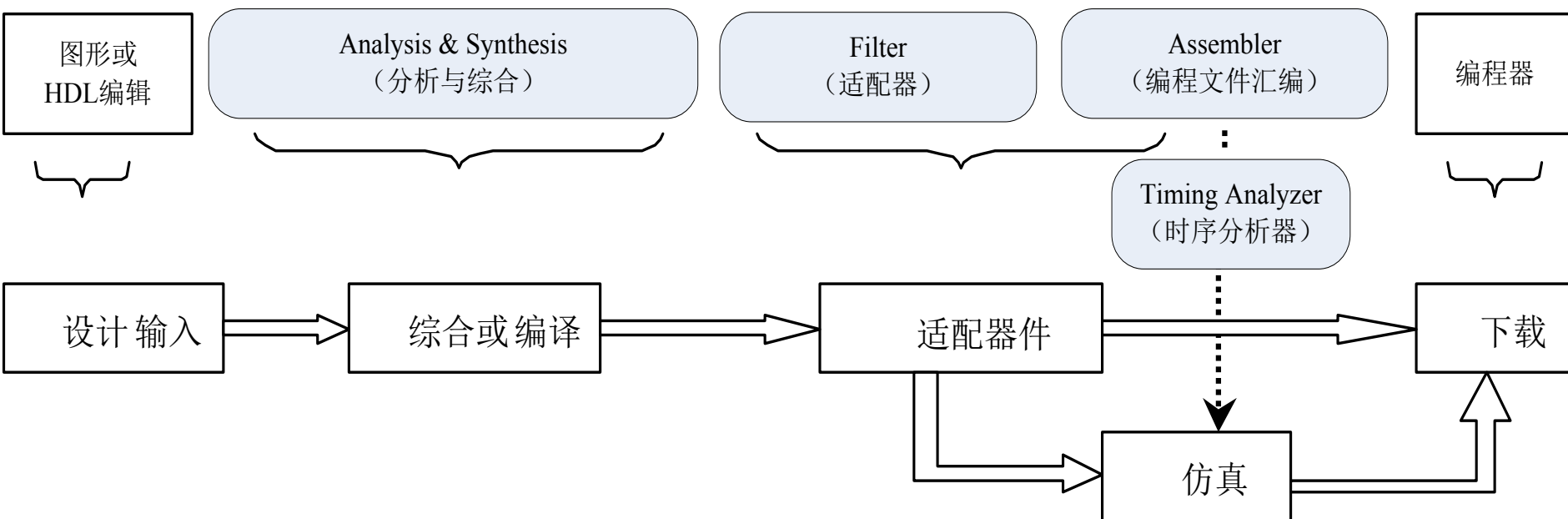
类 别	运 算 符	优先级
逻辑、位运算符	! ~	高 ↓ 低
算术运算符	* / %	
	+ -	
移位运算符	<< >>	
关系运算符	< <= > >=	
等式运算符	= = ! = == = != =	
缩减、位运算符	& ~&	
	^ ^~	
	~	
逻辑运算符	&&	
条件运算符	? :	

➤ 为提高程序的可读性，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)



# Quartus II 简介



## Quartus II设计流程



# PLD设计EDA工具软件



## 1. Quartus II

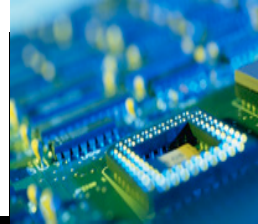
- 美国Altera公司自行设计的第四代PLD开发软件
- 目前版本：9.0
- 可以完成PLD的设计输入、逻辑综合、布局与布线、仿真、时序分析、器件编程的全过程
- 同时还支持SOPC（可编程片上系统）设计开发

## 2. ModelSim

- 美国Mentor Graphics公司的子公司Model Technology开发的仿真工具
- 目前版本：6.0
- 业界使用最广泛的HDL语言仿真器之一
- 支持VHDL、Verilog HDL或混合HDL语言设计
- 仿真功能强大，仿真速度快！

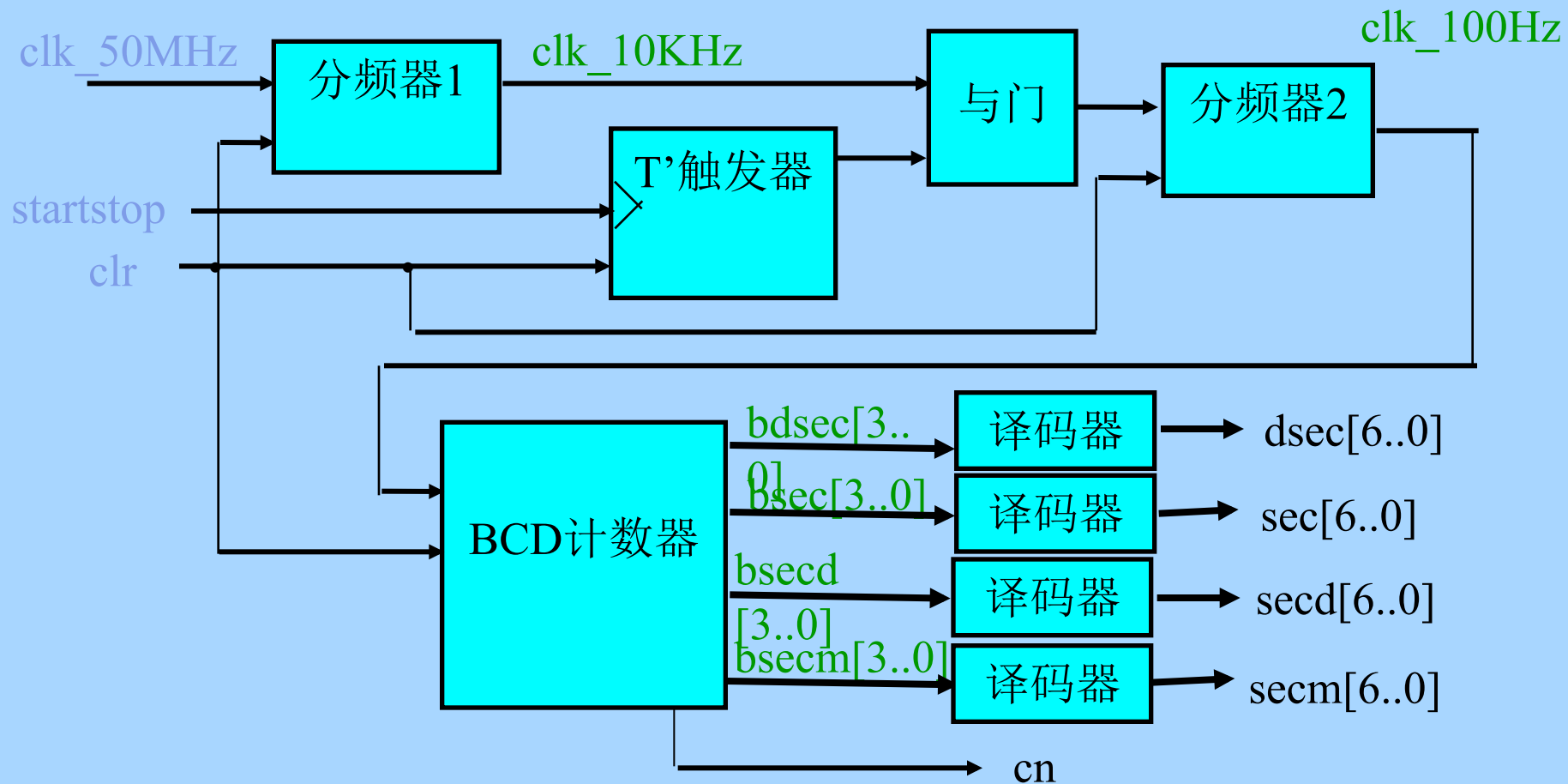
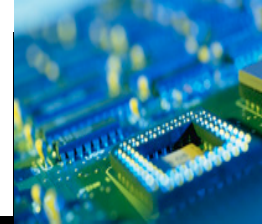


# 实例演示：电子秒表电路的设计



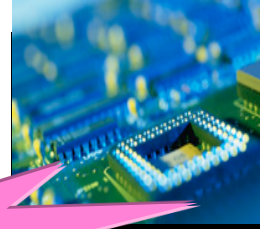
- 假设系统时钟为50MHz，PLD器件为EP1S10F780C6。
- 设计思路——采用自顶向下的设计方法：
  - 需要两个分频器，将50MHz分频为10KHz，将10KHz分频为100Hz；
  - 需要一个BCD码计数器，可分别对秒和百分秒位循环计数；
  - 需要一个译码器，将BCD计数器的输出译码为7段显示器的7段输入。

# 系统功能框图





# 采用Quatus II 的PLD设计方法



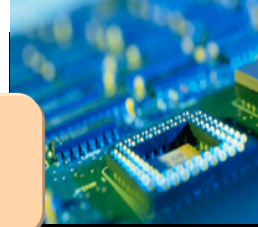
千万记住  
哦！

1. 首先在资源管理器下创建一个**工作目录**
2. 在Quatus II中创建一个**工程**。
3. 子模块设计：每个**模块**可以用HDL语言描述，对每个模块进行编译、**仿真**，通过后然后生成模块符号。
4. 顶层设计：创建一个**顶层图形文件**，将各模块符号放到图中，添加输入、输出引脚，连线；编译，仿真。
5. 给输入、输出引脚**分配引脚**号码，编程下载。

工作目录和工程名  
不能有空格和汉字！



最好每个工程都有自己的工作目录!



- 第1步：在资源管理器下创建一个工作目录second。

- 第2步：启动Quatus II，执行菜单命令“File > New Project Wizard”，创建一个工程，工程名为second。

若要打开一个已有的工程，则执行“File > Open Project ...”命令。

- 第3步：设计子模块

- (1) 执行菜单命令“File > New”，新建一个文本文件clkdiv100.v，采用Verilog HDL语言描述；
- (2) 存盘；
- (3) 指定该子模块为顶层实体，执行“Processing > Start Compilation”命令，对其进行全编译。

[模块1] 10KHz到100Hz的分频电路（采用Verilog HDL语言描述）

模块名(同文件名)

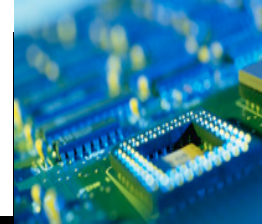
```
module clkdiv100 (clr,clkout,count);  
    input clr,clkout;           // 输入端口声明  
    output clkout,count;        // 输出端口声明  
    reg[6:0] count;  
    reg clkout;  
    always @(posedge clkout or negedge clr)  
    begin  
        if (!clr) count<=0;    // 异步清零！低有效
```

(1) 端口定义

(2) I/O说明

(3) 功能描述

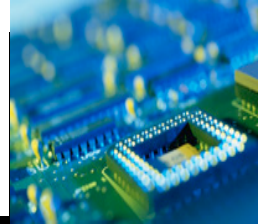




## 子模块设计——100分频器（续）

续前页

```
else if (count[6:0]==99)
    begin
        clkout<=1;    // clkout只在计数值为99时为"1"
        count[6:0]<=0;
    end
else
    begin
        clkout<=0;    // clkout在其他时候都为"0"
        count[6:0]<=count[6:0]+1;
    end
end
endmodule
```

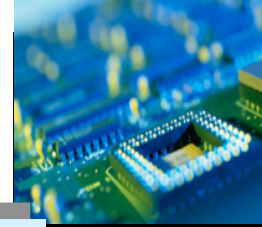


## [模块2] BCD计数器

```
module bcdcnt(dsec,sec,secd,secm,cn,clk,clr);  
    input clk,clr;           //Tclk = 0.01s  
    output[3:0] dsec,sec,secd,secm;  
    output cn;               //秒高位向分钟的进位  
    reg[3:0] dsec,sec,secd,secm;  
    reg cn;  
    always @(posedge clk or negedge clr)  
    begin  
        if (!clr)           // (1) 异步清零!  
            begin  
                cn<=0;       //进位信号也必须清零!  
                dsec[3:0]<=0; sec[3:0]<=0;  
                secd[3:0]<=0; secm[3:0]<=0;  
            end  
    end
```



# 子模块设计——BCD计数器（续1）

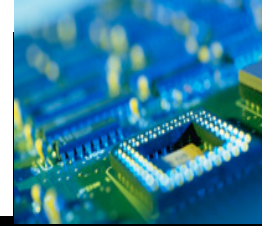


续  
前  
页

```
else // (2) 计数，采用4个if语句的嵌套
begin
    if(secm[3:0]==9) //百分秒低位是否为9?
        begin
            secm[3:0]<=0;
            if(secd[3:0]==9) //百分秒高位是否为9?
                begin
                    secd[3:0]<=0;
                    if(sec[3:0]==9) //秒低位是否为9?
                        begin
                            sec[3:0]<=0;
                            if(dsec[3:0]==5) //秒高位是否为5?
                                dsec[3:0]<=0;
                            else dsec[3:0]<=dsec[3:0]+1;
                        end
                    else sec[3:0]<=sec[3:0]+1;
                end
            else secd[3:0]<=secd[3:0]+1;
        end
    end
    else secm[3:0]<=secm[3:0]+1;
```



## 子模块设计——BCD计数器（续2）

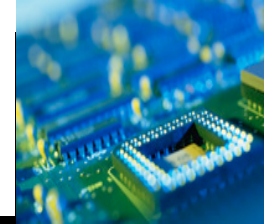


续

//（3）产生向分钟的进位信号

```
                                i      f
((dsec[3:0]==5)&&(sec[3:0]==9)&&(secd[3:0]==9)&&(secm[3:0]==9))
    cn<=1;
    else cn<=0;
end
end
endmodule
```

# 仿真子模块



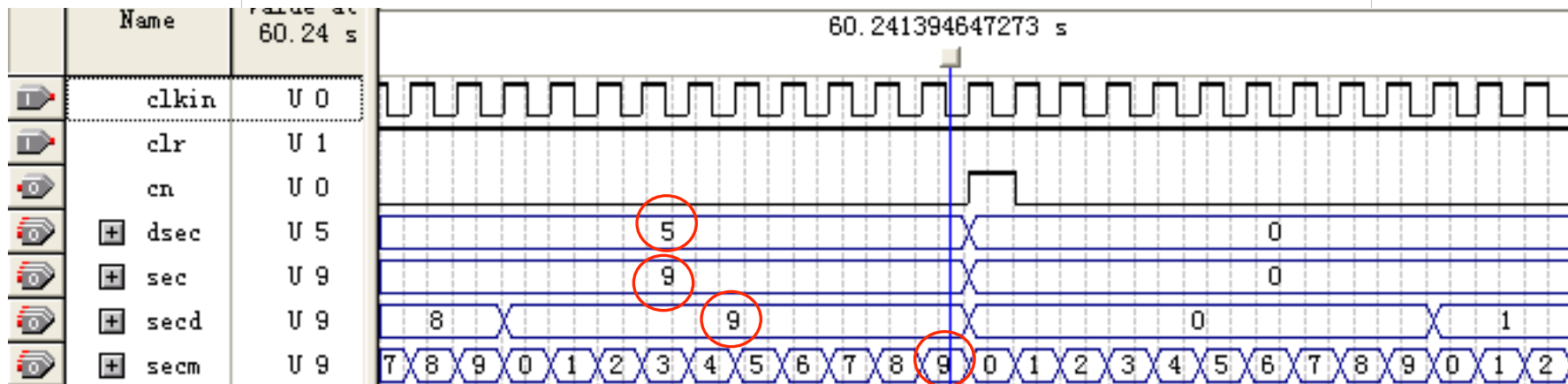
## 第3步:

### ❌ (4) 进行仿真

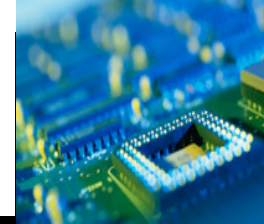
新建一个仿真波形文件.vwf，编辑输入波形；

执行“Tools> Simulation Tool”命令，打开仿真器工具窗口；

单击Start按钮，开始仿真。

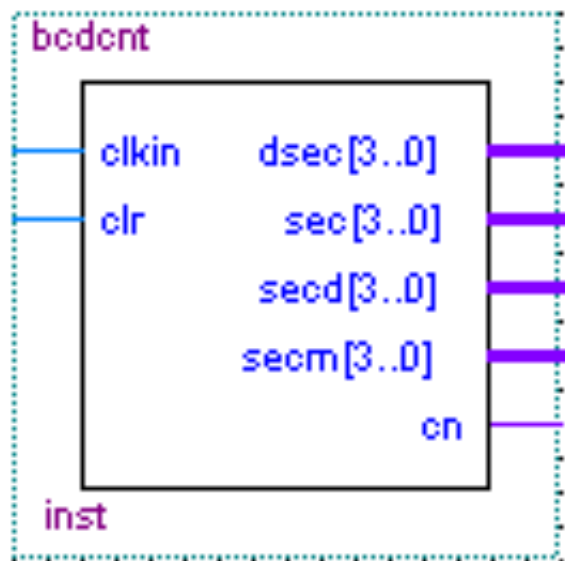


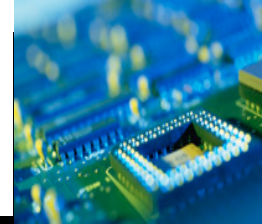
bcdcnt.vwf



## ➤ (5) × 创建模块符号

仿真通过后，执行“**File > Create/Update > Create Symbol Files for Current File**”菜单命令，创建模块符号（文件后缀为**.bsf**）。





# 子模块设计——7段码译码器

[模块3] 7段LED显示器(共阳极) 译码器。

```
module p7seg(out,data);  
    input [3:0]data ; //7段显示器输入  
    output [6:0] out; //7段显示器字段输出
```

out[6:0]相当于a,b,c,d,e,f,g

```
reg [6:0] out;  
always @(data )
```

```
    case (data)
```

```
        4'd0: out <= 7'b00000001 ;
```

```
        4'd1: out <= 7'b10011111 ;
```

```
        4'd2: out <= 7'b00100010 ;
```

```
        4'd3: out <= 7'b00000110 ;
```

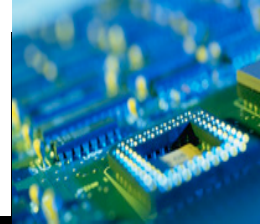
```
        4'd4: out <= 7'b10011100 ;
```

```
        4'd5: out <= 7'b01001100 ;
```

case语句适于  
对同一个控制  
信号取不同的  
值时，输出取  
不同的值！



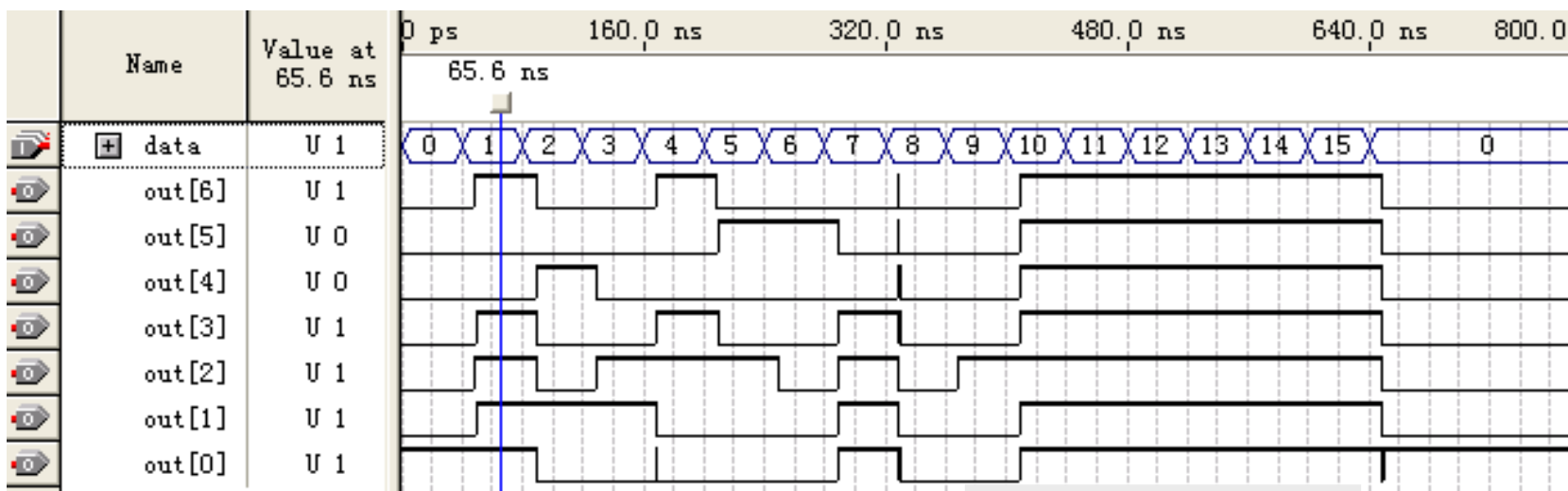
# 子模块设计——7段码译码器（续）



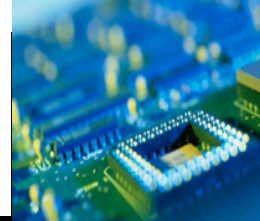
续  
前  
页

```
4'd6: out <= 7'b0100000 ;  
4'd7: out <= 7'b0001111 ;  
4'd8: out <= 7'b0000000 ;  
4'd9: out <= 7'b0000100 ;  
default:out <= 7'b1111111;  
//当data为4'hA~4'hF时，七段显示器不亮  
endcase
```

endmodule







## 第4步：设计顶层图形文件

- (1) 创建一个顶层图形文件`second.bdf`，将各模块符号放到图中，添加输入、输出引脚，连线；

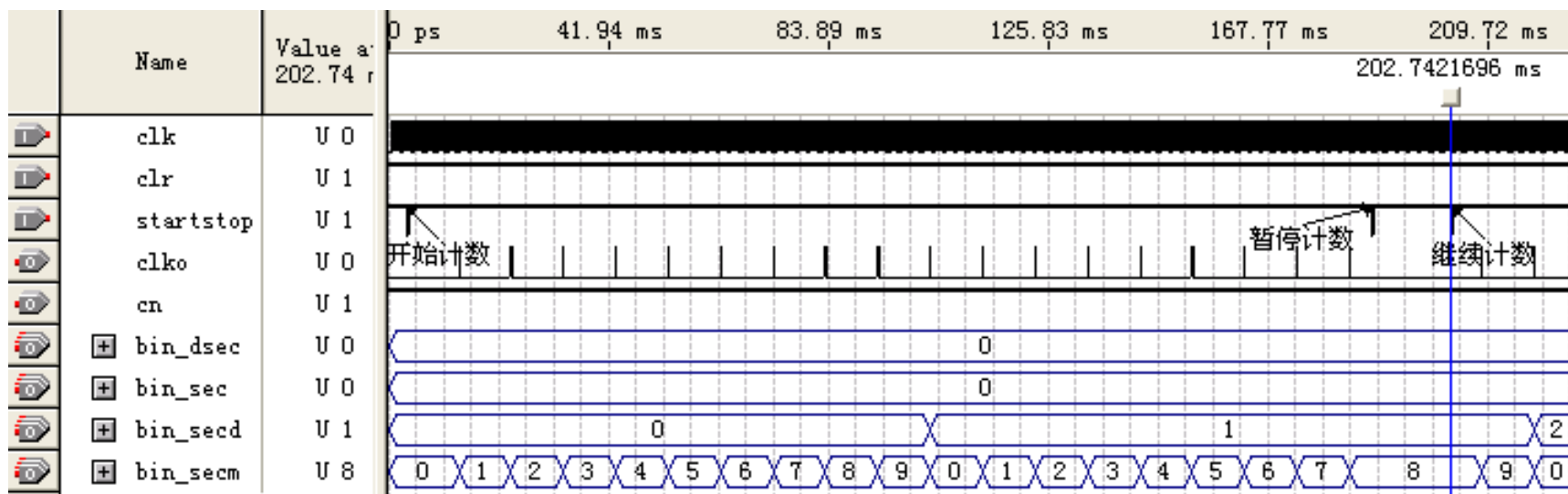


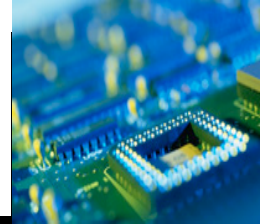


# 编译和时序仿真














- (2) 将该图形文件设置为顶层实体；
- (3) 进行编译器选项设置；
- (4) 全编译；
- (5) 对顶层图形文件仿真（如果必要的话）。





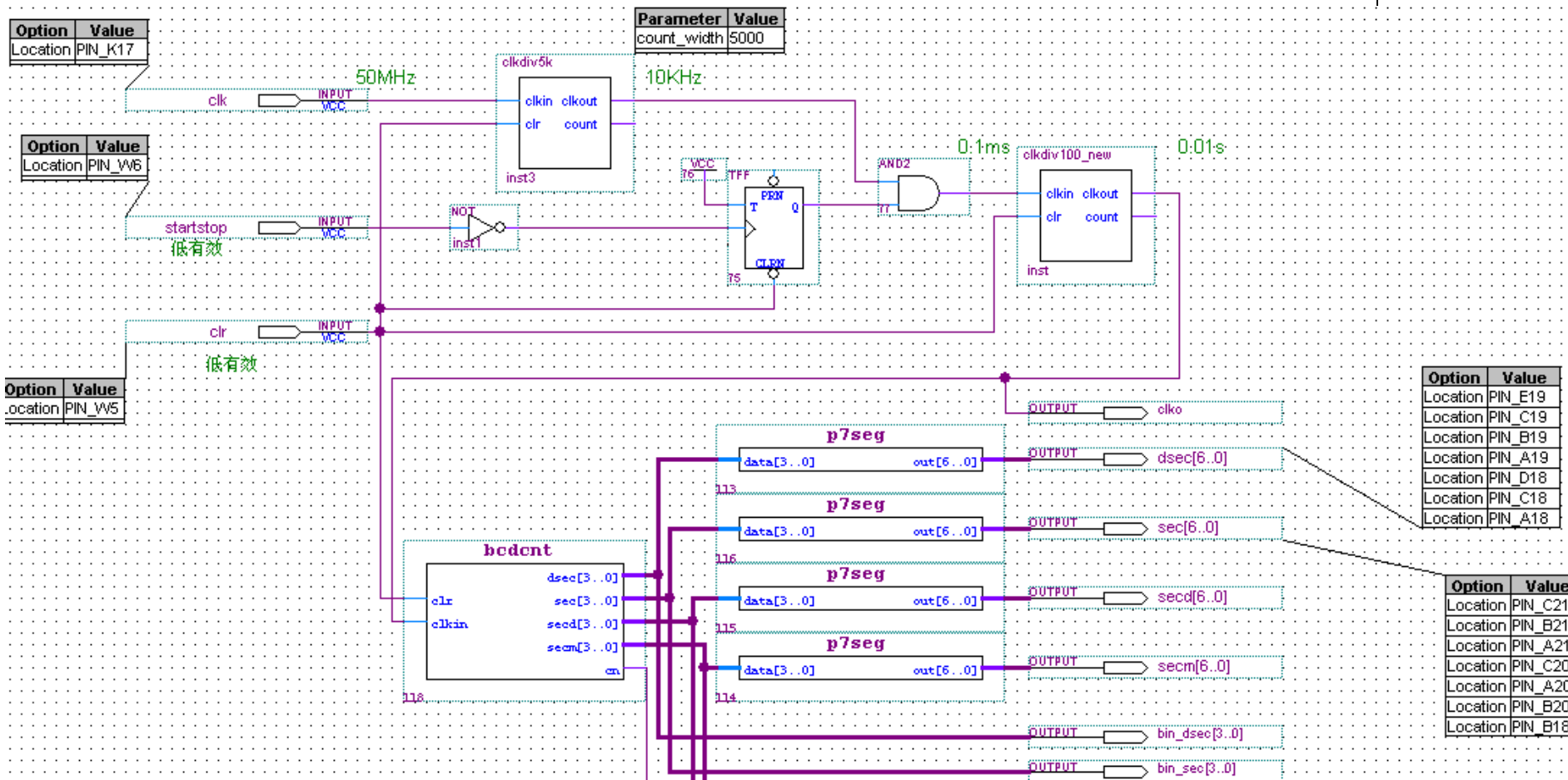
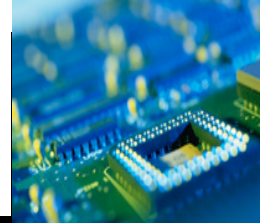
- 第5步：对下载用顶层文件（如second\_download.bdf）指定目标器件，给输入、输出引脚分配引脚号码，编程下载

- (1) 在Assignment Editor 中进行引脚锁定

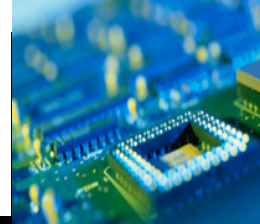
	To	Location	I/O Bank	I/O Standard	General Function
1	 clr	PIN_W5	6	LVTTL	Row I/O
2	 dsec[0]	PIN_E19	3	LVTTL	Column I/O
3	 dsec[1]	PIN_C19	3	LVTTL	Column I/O
4	 dsec[2]	PIN_B19	3	LVTTL	Column I/O
5	 dsec[3]	PIN_A19	3	LVTTL	Column I/O
6	 dsec[4]	PIN_D18	3	LVTTL	Column I/O
7	 dsec[5]	PIN_C18	3	LVTTL	Column I/O
8	 dsec[6]	PIN_A18	3	LVTTL	Column I/O
9	 sec[0]	PIN_C21	3	LVTTL	Column I/O
10	 sec[1]	PIN_B21	3	LVTTL	Column I/O
11	 sec[2]	PIN_A21	3	LVTTL	Column I/O

Assignment  
Editor

# 引脚锁定 (续)



second\_download.bdf



- (2) 将该图形文件设置为顶层实体，编译，则生成编程目标文件second.sof文件（编程目标文件自动与其工程同名）；
- (3) 编程下载；  
执行“Tools > Programmer”命令，在编程器窗口中选中“Program/Configure”复选框；单击Start按钮，开始编程下载。若完成编程，则在Message窗口中显示“Configuration succeeded”。
- (4) 在线校验。  
利用实验板上的按钮，模拟启动计数、暂停计数和继续计数，以及异步清零功能，然后观察数码管和LED的显示，看是否与预定的功能相符。

PLD器件和EDA技术的出现改变了传统的数字系统设计思想，使硬件设计变得简单、高效！

