

## 第三章

# Verilog 硬件描述语言 基础知识

# 内 容

第一节 概述

第二节 Verilog HDL基础知识

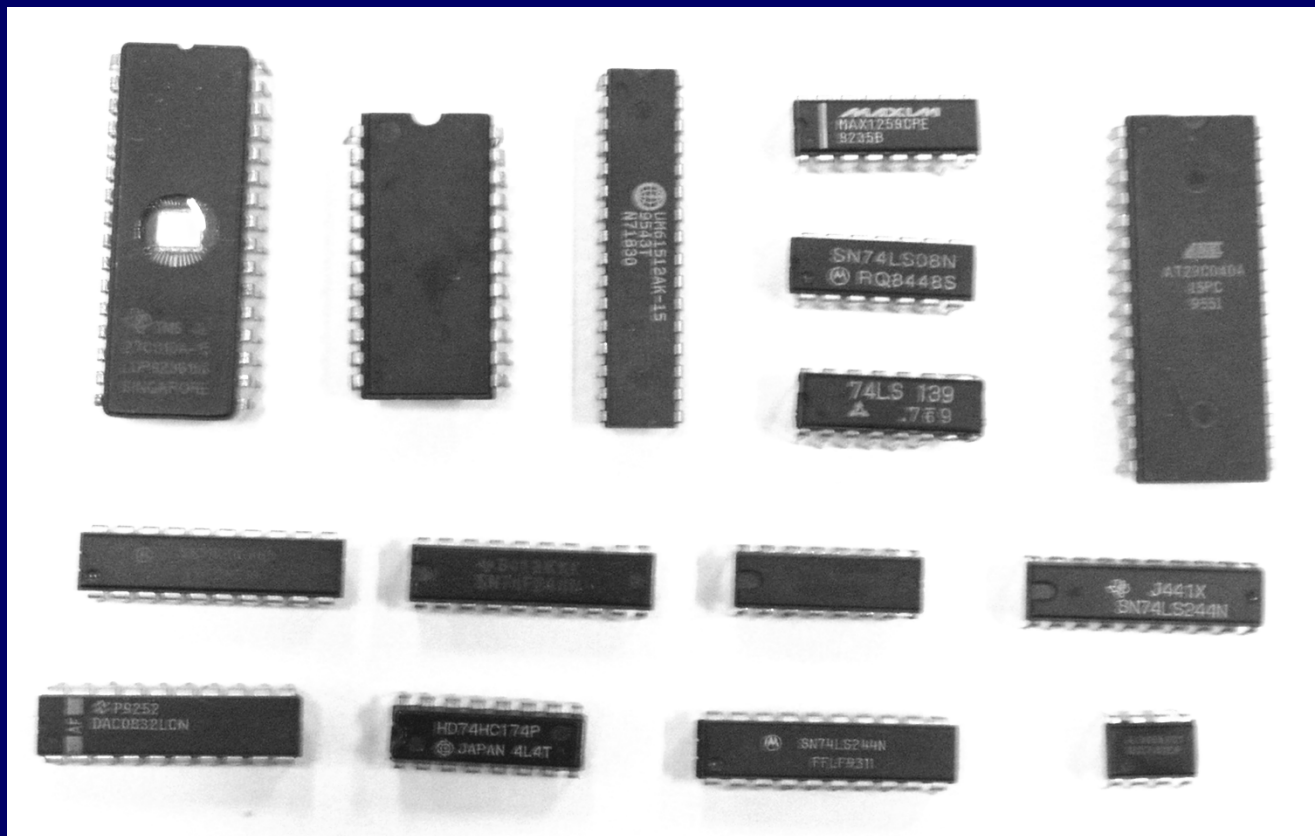
第三节 Verilog HDL模块的结构描述

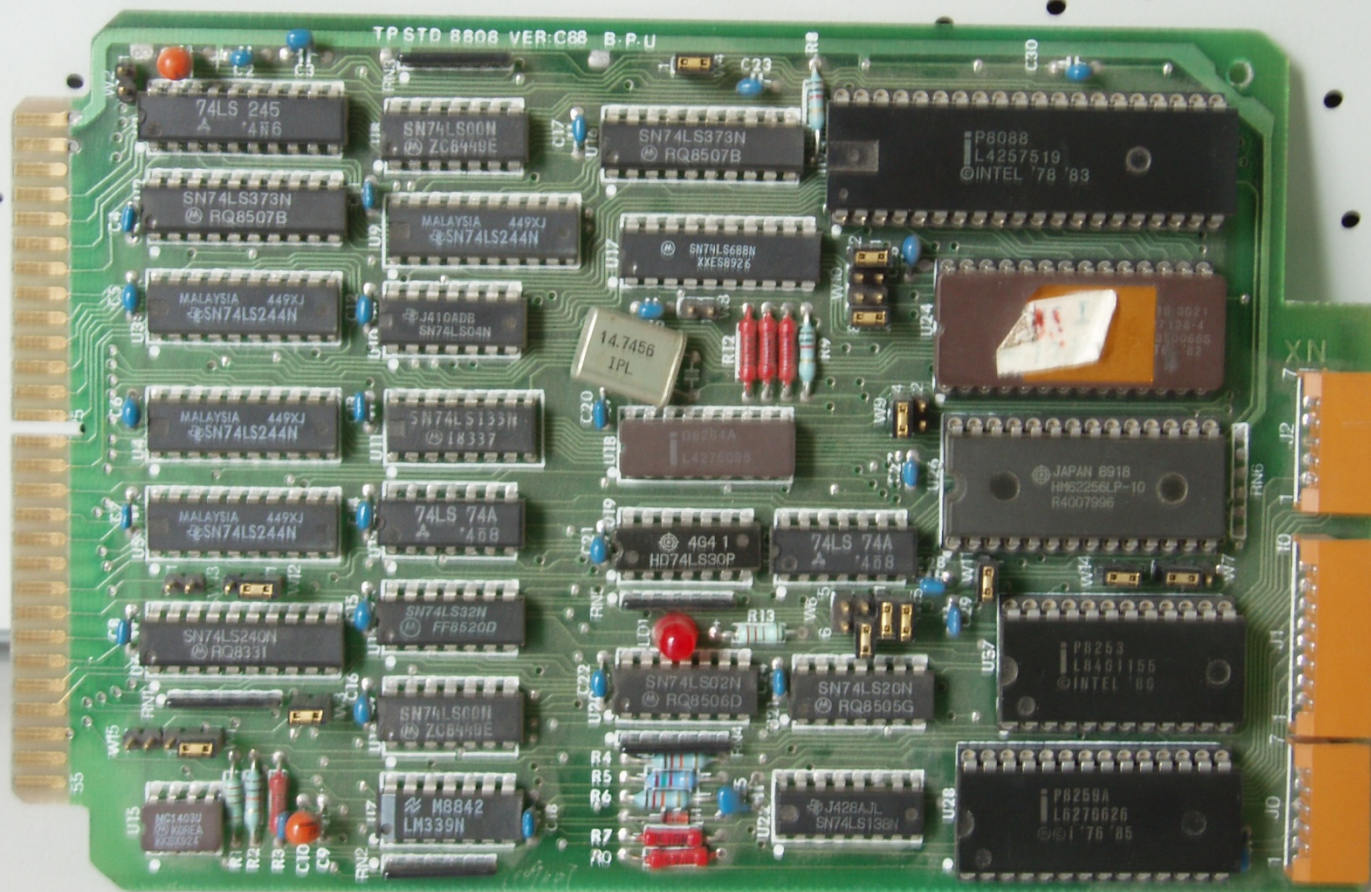
第四节 Verilog HDL模块的数据流描述

第五节 Verilog HDL模块的行为描述

## 第一节 概述

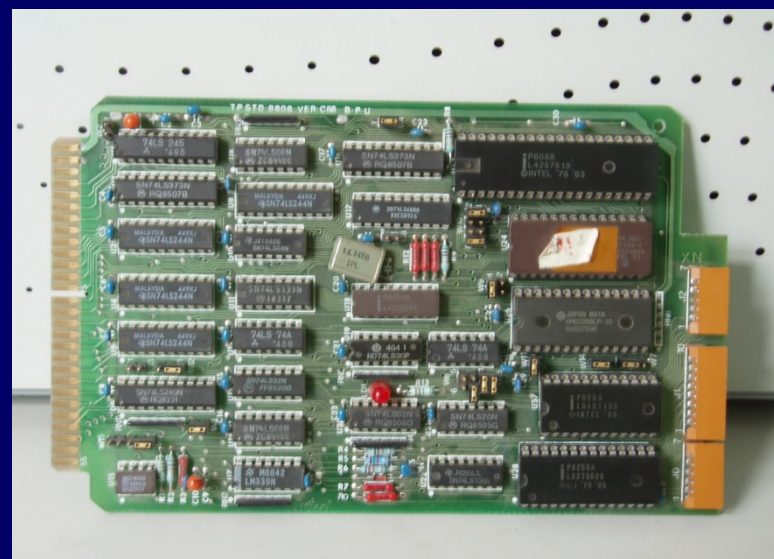
## 一.传统的数字系统设计技术





## 存在的“问题”

- (1) 逻辑器件的功能对系统设计的限制
- (2) 逻辑器件的集成度对系统规模的限制
- (3) 逻辑器件之间的布线对系统性能的限制
- (4) 不便于系统的扩展、升级
- (5) 不便于早期发现问题





## 二.现代的数字系统设计技术

基于**EDA**技术及大规模可编程逻辑器件的数字系统设计。

- 1.微电子技术以惊人的速度发展，集成电路的工艺水平达到深亚微米级，一个芯片上可集成数百万乃至上千万只晶体管，工作速度可达**Gb/s**，为制造出规模大、速度快和信息容量更高的芯片系统提供了基础条件。

特别是，大规模可编程逻辑器件（**PLD** —— **Programmable Logic Device**）的出现，引起了数字系统设计领域的革命性变革。**PLD**已成为现代数字系统的物理载体，设计师在实验室就可以设计出芯片系统，并立即投入实际应用。

**PLD**是厂家作为一种通用性器件生产的半定制电路。其特点是：

- (1) 用户通过对器件的编程实现所需要的逻辑功能；
- (2) **PLD**是一种用户可配置的逻辑器件；
- (3) 成本低、使用灵活、设计周期短、可靠性高、风险小。

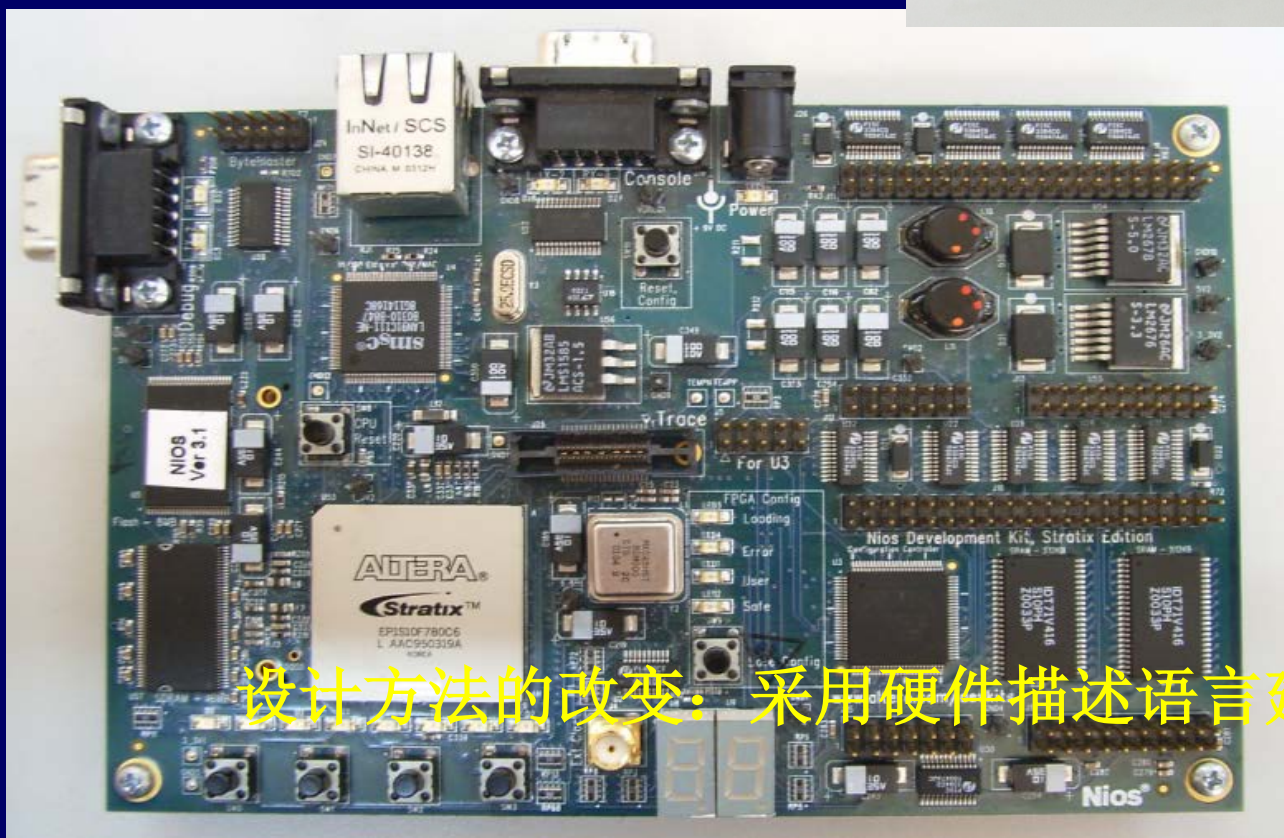
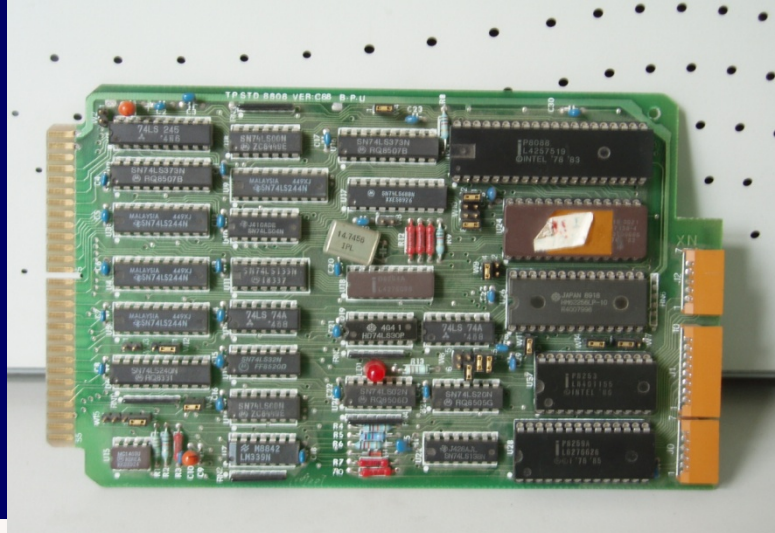
2.以计算机为工作平台，融合应用电子技术、计算机技术、智能化技术以及图形学、拓扑学、计算数学等众多学科的最新成果，辅助进行集成电路设计、电子电路设计和PCB设计的EDA（**Electronic Design Automation**,电子设计自动化）技术日趋完善。

现代EDA技术的主要特征是：

- 引入硬件描述语言；
- 支持高层次的抽象设计；
- 具有逻辑综合、行为综合、系统综合能力。



总之，PLD器件的出现和EDA技术发展，使数字系统设计的设计方法和设计技术都发生了深刻的变化，称之为**基于EDA和PLD的现代数字系统设计方法**。设计者通过设计芯片来实现各种不同的数字系统功能，即设计者可以自己定义器件的内部逻辑和管脚。增加了设计的自由度，提高了设计效率，减少芯片的种类和数量，缩小体积，降低功耗，节约成本，提高可靠性。



设计方法的改变：采用硬件描述语言建模

### 三.硬件描述语言

硬件描述语言（**HDL**）是一种采用形式化方法描述数字电路和数字系统的语言。

设计者利用这种语言可以从上层到下层（从抽象到具体），逐层描述自己的设计思想，用一系列分层次的模块来表示极其复杂的数字系统。

通过**EDA**综合工具将需要物理实现的模块组合转换成门级电路网表。

借助**EDA**仿真工具逐层进行仿真验证。

利用芯片厂家提供自动布线布局工具（适配器）将网表映射为具体器件的布线结构，最后编程下载到**PLD**器件。

从**20世纪80年代**至今，先后出现过数十种硬件描述语言。目前流行的有**VHDL**和**Verilog HDL**两种。

**1987年**，由美国国防部组织开发的**VHDL**成为**IEEE**国际标准。

**1995年**，由一个民间公司的私有财产转化而来的**Verilog HDL**也正式成为**IEEE**国际标准。

本课程主要介绍如何用**Verilog HDL**描述基本的组合逻辑电路和基本的同步时序逻辑电路，为今后进行数字系统设计打基础。

## 四. Verilog HDL的产生及发展

**Verilog HDL**是目前应用最广泛的一种硬件描述语言。据有关文献报道，在美国使用**Verilog HDL**进行设计的工程师约有十几万人，全美国有**200**多所大学讲授**Verilog HDL**的设计方法；在我国的台湾省，几乎所有著名大学的电子和计算机工程系都开设了与**Verilog HDL**有关的课程。

## Verilog HDL事件表:

1983年	由GDA（GateWay Design Automation）公司的Phil Moorby首创。
1984年	Moorby设计出第一个仿真器：Verilog-XL
1986年	Moorby提出了用于快速门级仿真的XL算法，Verilog HDL得到迅速发展。
1989年	Cadence公司收购了GDA，Verilog HDL成为其私有财产。
1990年	Verilog HDL公开发表，并成立OVI（Open Verilog International）组织，负责促进该语言的发展。
1995年	基于该语言具有简洁、高效、易学易用、功能强等优点，Verilog HDL IEEE1364-1995标准公开发表，成为标准化的硬件描述语言。
1999年	模拟和数字都适用的Verilog HDL标准公开发表。
2001年	Verilog HDL IEEE1364-2001标准公开发表。
2005年	.....



## 五. Verilog HDL的特点

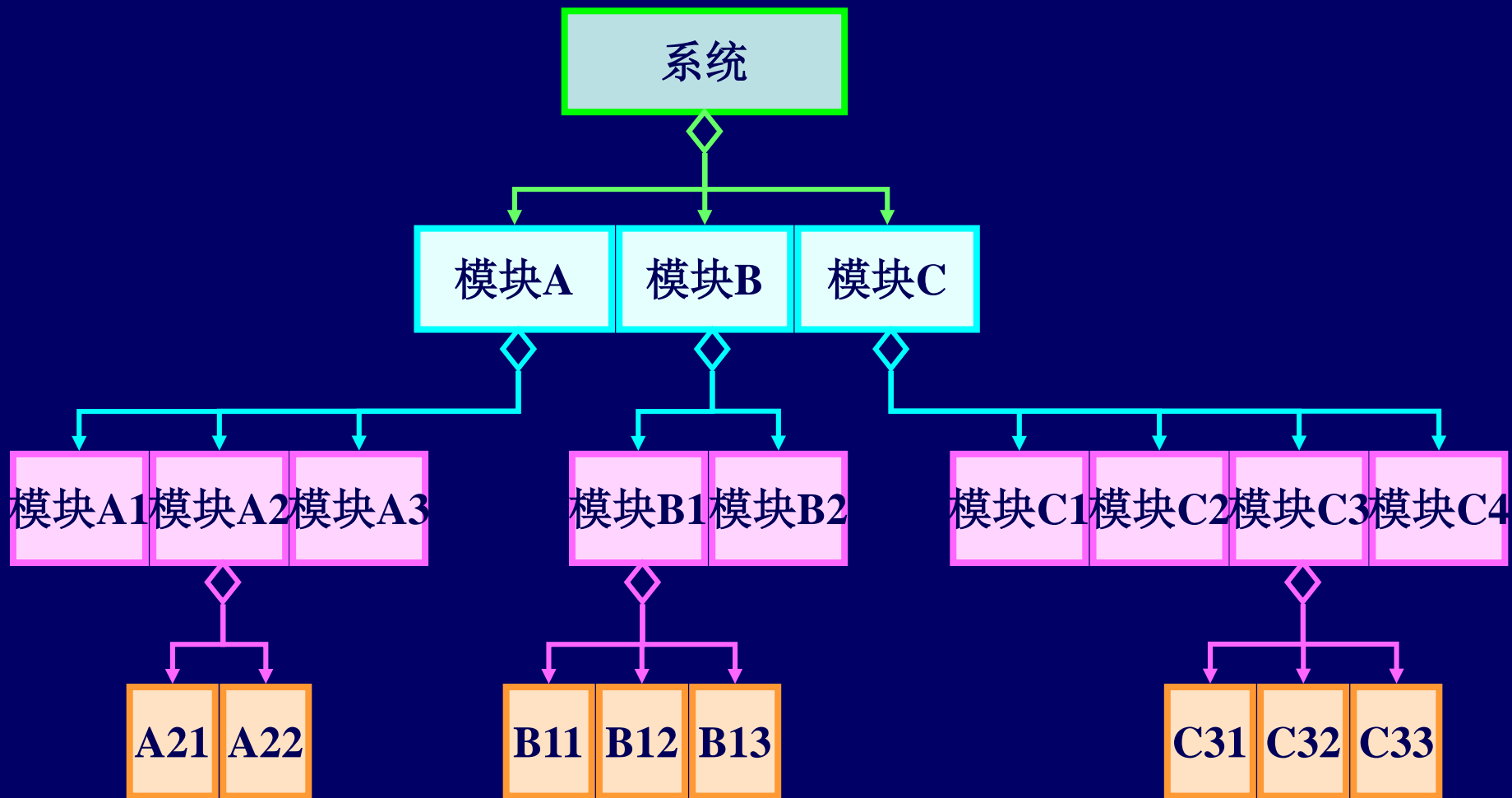
- 既能进行面向综合的电路设计，也可用于电路的模拟仿真。
- 能够在多个层次上对所设计的系统加以描述：开关级、门级、寄存器传输级、行为级。
- 不对设计的规模施加任何限制。
- 具有混合建模能力，即在一个设计中，各个模块可在不同层次上建模和描述。
- 灵活多样的电路描述风格：行为描述、结构描述、数据流描述。
- 其行为描述语句（条件、赋值、循环）类似于高级语言，易学易用。
- 内置各种基本逻辑门，适合门级建模。
- 内置各种开关级元件，适合开关级建模。
- 可灵活创建用户定义原语（**UDP**）。
- 可通过编程语言接口（**PLI**）调用C语言编写的各种函数。



## 六.层次管理的概念

现代集成电路制造工艺水平已经达到了深亚微米级（**0.15 $\mu\text{m}$** ），可以在一个芯片上集成数十万乃至数千万个典型门。设计如此大规模的电路必须采用层次化、结构化的设计方法，即总设计师将一个完整的硬件设计方案逐层分解，划分成若干个可操作的模块，由多个设计师同时设计一个硬件系统中的不同模块，上一层设计师采用行为级的逻辑模拟模块对其下一层设计者的设计进行验证。其中一些模块可采用商业化的**IP核**。

显然，这是一种**Top-Down**的设计思想，可以用设计树的形式描述。



## 第二节 Verilog HDL基础知识

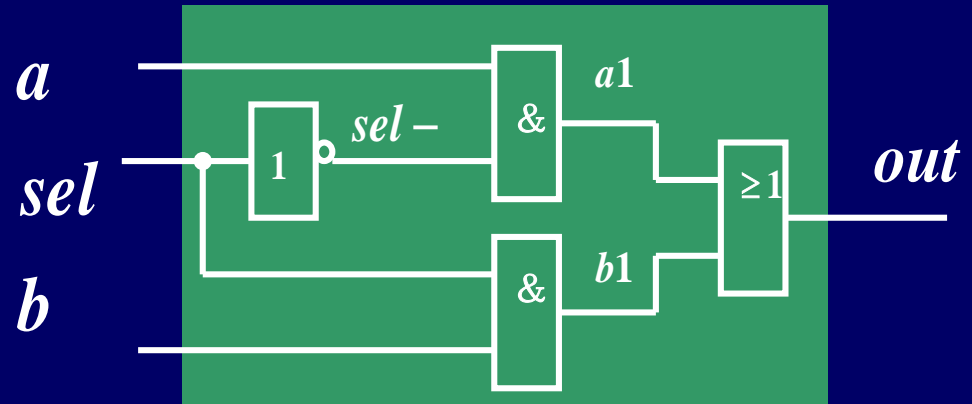
### 一.Verilog 模块的结构

**Verilog HDL**以模块集合的形式来描述数字系统。

“模块”（**module**）是**Verilog** 程序的基本设计单元，用于描述某个设计的功能、结构、与其他模块通信的外部端口。

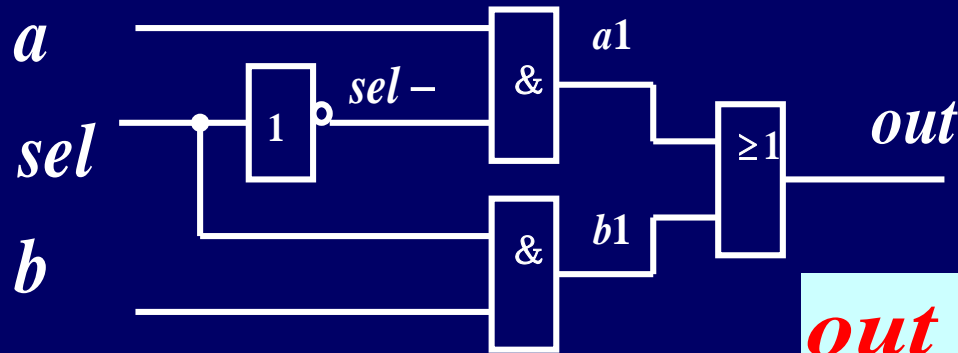
先通过三个简单的**Verilog**程序，直观地认识**Verilog**模块的结构。

例1:  
根据电路结构建模



```
module MUX2-1 (out, a, b, sel); //模块名、端口列表
    output out; //声明out为输出端口
    Input a, b, sel; //声明a,b,sel为输入端口
    wire a, b, sel, out; //定义端口信号的数据类型
    wire a1, b1, sel-; //定义内部节点信号（连线）
    not u1 (sel-, sel); //调用内置“非”门元件
    and u2 (a1, a, sel-); //调用内置“与”门元件
    and u3 (b1, b, sel); //调用内置“与”门元件
    or u4 (out, a1, b1); //调用内置“或”门元件
endmodule
```

## 例2：根据逻辑表达式建模



$$out = \overline{sel} \bullet a + sel \bullet b$$

```
module MUX2-1 (out, a, b, sel);
```

```
    output out;
```

```
    Input  a, b, sel;
```

```
    wire  a, b, sel, out;
```

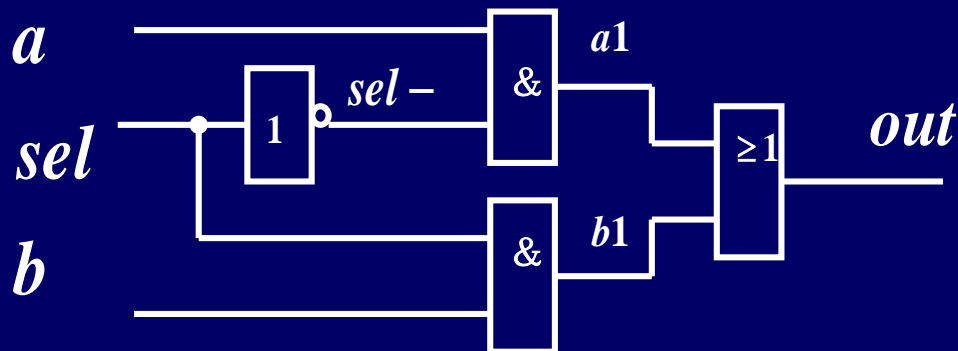
```
    assign out = ((~sel) & a) | (sel & b);
```

```
endmodule
```

定义同上

采用逻辑表达式进行逻辑功能的描述

### 例3：根据电路行为建模



```
module MUX2-1 (out, a, b, sel) ;
```

```
    output out;
```

```
    Input  a, b, sel;
```

```
    wire  a, b, sel;
```

```
    reg   out;
```

定义always中的赋值变量

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

```
        if ( ! sel )  out = a;
```

```
        else out = b;
```

```
endmodule
```

只要信号sel、a、b中有一个发生变化，就执行下面的语句。在这里已看不出电路的结构，而是一种电路行为的描述。

## Verilog 模块 基本结构

***module*** 模块名 （端口列表）；

端口定义

*input* 输入端口

*output* 输出端口

.....

数据类型说明

*wire*

*reg*

*parameter*

.....

逻辑功能定义

*assign*

*always*

语言内置门元件调用

...

***endmodule***



## 1. 模块声明

*Verilog* 模块结构完全包含在`module`和`endmodule`关键字之间。

模块声明的格式：

```
module 模块名 （端口1， 端口2、 ....） ;  
.....  
endmodule
```

## 2. 端口 (*Port*) 定义

端口是模块与外界或其他模块进行连接和通信的信号线，对模块的输入、输出端口要进行明确的定义，格式如下：

### 定义输入

```
input <[向量位宽]> 端口名1, 端口名2, ..... 端口名n;
```

### 定义输出

```
output <[向量位宽]> 端口名1, 端口名2, ..... 端口名n;
```

### 3. 数据类型和信号类型的声明

每个端口除声明为输入、输出，还要声明其数据类型是连线型（*wire*）还是寄存器型（*reg*），若没有声明，综合器默认为*wire*型。

特别提示：输入不能声明为*reg*型！

模块中用到的节点信号（连线）也必须进行数据类型的定义。

用*wire*将信号定义为连线型；  
用*reg*将信号定义为寄存器型；  
用*parameter*定义符号常量。

## 4. 逻辑功能定义

模块中最核心的部分是逻辑功能定义。下面介绍几种基本方法。

### 4.1 用“*assign*”持续赋值语句进行逻辑功能的定义

例如: *assign F*=~((*A*&*B*)/(*C*&*D*));

“*assign*”语句一般用于组合逻辑的赋值，称为持续赋值方式。是一种基于逻辑表达式的逻辑功能描述方式。

在前面的例2中，就是采用这种方式建立电路模型。

## 4.2 调用内置元件（元件例化）描述电路结构

调用 *Verilog HDL* 中提供的动态元件“画”电路图。

例如:     `and myand3 ( out, a, b, c );`



调用门元件，构造一个名为 *myand3* 的三输入与门。

`and c1 ( F, in1, in2 );`



调用门元件，构造一个名为 *c1* 的二输入与门。

前面例1中，通过调用门元件，进行信号的相互连接，构造电路的模型。

## 4.3 用 “*always*”过程块描述电路的逻辑功能

例如:

.....

```
always @ (posedge clk)
```

```
  begin
```

```
    if (reset) out=0 ;
```

```
      else out=out+1 ;
```

```
  end
```

.....

这是一个同步清零计数器的描述。每当*clk*上升沿到来时就执行一遍*begin—end*块内的语句。

前面例3中，就是采用 “*always*”描述电路的行为。

## 小结

- *Verilog HDL* 程序是由模块构成的
- 模块是可以进行层次嵌套的
- 上层模块可以通过模块调用构成更大的逻辑系统
- *Verilog* 模块分为逻辑综合模块和逻辑模拟模块
- 每个模块由模块声明、端口定义、数据类型说明、逻辑功能定义四部分构成
- *Verilog HDL* 程序的书写格式自由，一行可写多个语句，一个语句也可分写多行。
- 除 *endmodule* 外，每个语句和数据定义的最后必须有分号
- 可用 */\*...\*/* 和 *//...* 进行多行、单行注释，增强程序的可读性



## 二. *Verilog HDL*中的数字（数值）常量

*Verilog HDL* 有下列四种基本的逻辑状态:

*0*: 低电平、逻辑0或“假”

*1*: 高电平、逻辑1或“真”

*x*或*X*: 不确定或未知的逻辑状态

*z*或*Z*: 高阻状态

*Verilog HDL*中的常量是由这四类基本值组成的。

*Verilog HDL*可综合模块中常用的整数型常量的书写格式:

$\pm < size > ' < base\_format > < number >$

正、负号

当指定  
进制格  
式时，  
不能省  
略

位宽：对应二进  
制数的宽度，省  
缺为32位。

基于进制的数  
字序列

数值采用的进制格式

*b*或*B*：二进制

*d*或*D*或缺省：十进制

*h*或*H*：十六进制

*o*或*O*：八进制

$\pm < \text{size} > ' < \text{base\_format} > < \text{number} >$

例:

**659** //简单的十进制表示

**'h 837FF** /\*省缺位宽的十六进制数，位宽大于实际位数，数值高位是  
0或1，高位补0；数值高位是  $x$  或  $z$ ，高位补  $x$  或  $z$ 。\*/

**'o 7460** //省缺位宽的八进制数

**4AF** //非法的整数表示，十六进制需要 **'h**

**b001** //非法的整数表示，不能省略 **'**

**4'b0010** //四位的二进制数

**5'D3** //五位的十进制数

**8 'b0100\_1010** //使用下划线增加可读性

### 三. Verilog HDL中的标识符

由字母、数字以及符号 “\$”、 “\_”（下划线）组成。

- 标识符必须以字母或下划线开头
- 标识符是区分大小写的

合法标识符举例：

*count*

*COUNT*

*\_A1\_d2*

不正确的标识符：

*30count*

*Out\**

*\$123*

*module*

标识符常用于“模块名”或“变量名”

## 四. *Verilog HDL*中的关键字

关键字也称为保留字，是*Verilog HDL* 语言内部的专用词，用于组织语言结构，全部采用小写形式。

例如： *module*、*endmodule*、*begin*、*end*、*always*、*and*、*or*、*if*、*else*、*wire*、*reg*、*input*、*output*、.....

请注意： *ALWAYS*是标识符，与关键字*always*是不同的。

## 五. *Verilog HDL* 中的数据（变量）类型

在硬件描述语言中，数据类型用来表示数字电路中的物理连线、数据存储和传送单元。

*Verilog HDL*中共有19种数据类型，分为连线型（*Net Type*）和寄存器型（*Register Type*）。在可综合模块中，最常用的是这两类中的 *wire* 型、*reg* 型、*integer* 型和 *parameter* 型。

### 1. 连线型（*Net Type*）变量

连线型变量一般用来描述硬件电路中的各种物理连线。其特点是输出始终跟随输入的变化而变化。

两种驱动方式，

- 在结构描述中将其连接到一个逻辑门或模块的输出端；
- 用 *assign* 语句进行赋值。

## Verilog HDL提供了多种连线型变量

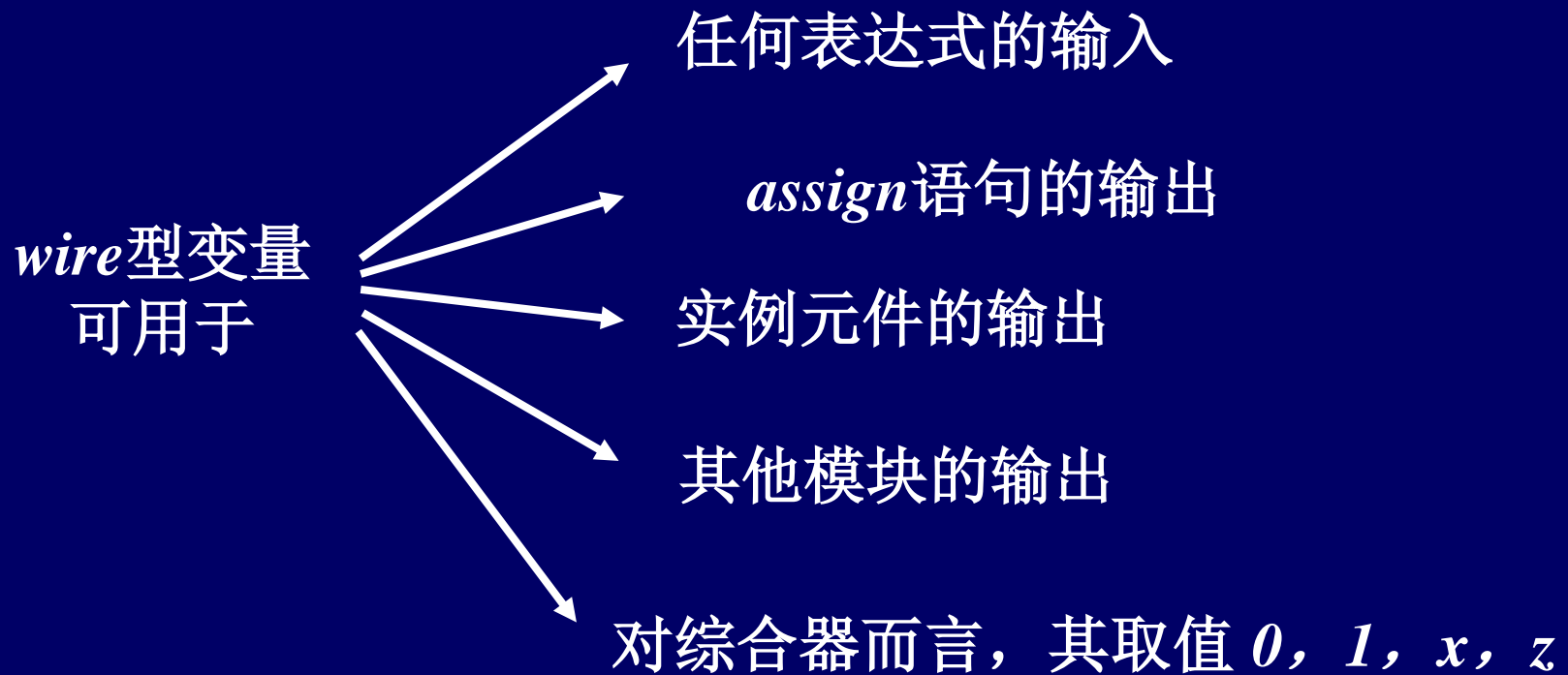
类型	功能说明
<i>wire, tri</i>	常用连线类型
<i>wor, trior</i>	具有线或特性的连线
<i>wand, triand</i>	具有线与特性的连线
<i>tri1, tri0</i>	上拉电阻，下拉电阻
<i>supply1, supply0</i>	电源（逻辑1），地（逻辑0）

重点介绍在可综合模块中最常用的是*wire*型变量



*wire*型变量常用来表示通过 *assign* 语句赋值的组合逻辑信号。

*Verilog* 模块中的**输入、输出端口**的信号类型说明省缺时，自动定义为 *wire* 型。



## *wire*型变量的定义格式

**wire** <[向量位宽]> 数据名1, 数据名2, ... .. 数据名n ;

关键字

$n-1:0$  或  $n:1$  时, 定义多位*wire*型向量;  
省略时, 定义一位的*wire*型变量。

例如，*wire*型变量或向量的定义：

```
wire a, b;           //定义两个 1 位 wire 型变量 a, b  
wire [7:0] databus; //定义一个8位宽的向量（数据总线）  
wire [20:1] addrbus ; //定义20位宽的地址总线
```

例如，*wire*型向量的全域使用

.....

```
wire [7:0] in, out; //定义两个8位wire型向量in, out  
assign out = in;    //按位对应赋值
```

.....

例如，对*wire*型向量可选域使用

.....

```
wire [7:0] out;
```

```
wire [3:0] in;
```

```
assign out [5:2] = in; // in 的 0-3 位赋值给 out 的 2-5 位
```

.....



等效于： *assign out [5] = in [3];*

*assign out [4] = in [2];*

*assign out [3] = in [1];*

*assign out [2] = in [0];*

## 2. 寄存器型 (*Register Type*) 变量

寄存器型变量对应的是具有状态保持作用的电路元件。例如：触发器，寄存器。

寄存器型变量的特点是必须进行明确的赋值，并且在被重新赋值前一直保持原值。

*register*型变量与*net*型变量的根本区别在于：

*register*型变量能够保持最后一次的赋值；*net*型变量需要有持续的驱动。

在设计中必须将寄存器型变量放在过程块语句 (*initial*, *always*) 中，通过过程赋值语句赋值。即： *initial*, *always* 过程块内被赋值的每一个信号，都必须定义为*register*型变量。

*Verilog HDL*中也提供了多种寄存器型变量，在可综合模块中常用的是*integer*型和*reg*型。

*integer*寄存器型变量是一种纯数学的抽象描述，它虽然能定义带符号的32位整型寄存器变量，但不对应任何具体的硬件电路。

*reg*型变量的定义格式：

```
reg <[向量位宽]> 数据名1, 数据名2, ... 数据名n ;
```

关键字

$n-1:0$  或  $n:1$ 时，定义多位*reg*型向量；  
省略时，定义一位的*reg*型变量。

例如，*reg*型变量、向量的定义

*reg a, b;*     //定义两个寄存器型变量*a, b*。位宽为1。

*reg [7:0] data;*   //定义位宽为8的寄存器型变量*data*。

*reg [8:1] data;*   //等效。

在表达式中，可任意使用`reg`型变量中的一位或相邻几位，分别称为位选择和域选择。

例如：

```
.....  
reg [7:0] a,b ;  
reg[3:0] c ;  
reg d ;  
  
.....  
d=a[ 7 ] & b[ 7 ] ; //位选择  
c=a[ 7:4 ] + b[ 3:0 ] ; //域选择  
.....
```



### 3. *parameter*型数据

在 *Verilog HDL* 中，用 *parameter* 定义符号常量，即定义一个标识符代表一个常量。定义格式如下：

*parameter* 参数名1=表达式1, 参数名2=表达式2, ... .. ;

例：

*parameter sel=8, code=8'ha3 ;*

//定义 *sel* 为常数8（十进制），*code* 为常数 *a3*（十六进制）

*parameter data=8, addr=data \* 2 ;*

//*data*为8，*addr*为16，用常数表达式赋值。

## 六、 Verilog HDL的运算和运算符

运算类别	单目运算 unary operator	双目运算 binary operator	三目运算 ternary operator
<u>算术运算</u>		+ - * / %	
<u>逻辑运算</u>	!	&&	
<u>位运算</u>	~	&   ^ ~^, ^~	
<u>关系运算</u>		< <= > >=	
<u>等式运算</u>		== != === !==	
<u>缩减运算</u>	& ~&   ~  ^ ~^, ^~		
<u>移位运算</u>		>> <<	
<u>条件运算</u>			? :
<u>拼接运算</u>			{ }



## 算术运算符

+ 加

- 减

\* 乘

/ 除

% 模运算符（对两个整型数据进行求余运算）

均为双目运算符。

在进行整数除法运算时，结果略去小数。

取模运算时，结果符号与第一操作数相同。

当两操作数中有一个为  $x$ ，则结果就为  $x$ 。

$10\%3$ ， 结果为1；

$12\%4$ ， 结果为0；

$-10\%3$ ， 结果为-1；

$11\%-3$ ， 结果为2。



## 逻辑运算符

**&&**      逻辑与，双目运算

**||**        逻辑或，双目运算

**!**        逻辑非，单目运算

当进行逻辑运算时：

● 若操作数是一位的，则逻辑运算真值表如下：

<b>a</b>	<b>b</b>	<b>a&amp;&amp;b</b>	<b>a  b</b>	<b>!a</b>	<b>!b</b>
<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>1</b>
<b>0</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>1</b>	<b>0</b>
<b>1</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>1</b>
<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>0</b>

● 当操作数由多位组成，则将操作数作为一个整体处理，即如操作数所有位都是“0”，则整体作为逻辑“0”；只要其中有一位是“1”，则整体作为逻辑“1”。

● 若操作数中有  $x$ ，则结果也可能是  $x$ 。

例如：设  $a = 2$        $b = 0$        $c = 4'hx$

$a \&\& b$       逻辑与， $1 \&\& 0$ ，结果为 0

$a \parallel b$       逻辑或， $1 \parallel 0$ ，结果为 1

$! a$       逻辑非， $! 1$ ，结果为 0

$a \&\& c$       逻辑与， $1 \&\& x$ ，结果为  $x$

$a \parallel c$       逻辑或， $1 \parallel x$ ，结果为 1

$! c$       逻辑非， $! X$ ，结果为  $x$



## 位运算符

~ 按位非（按位取反），单目运算。

& 按位与

| 按位或

^ 按位异或

^~, ~^ 按位异或非（按位同或）

- 将操作数按位对应，进行逻辑运算。
- 两个不同长度的数据进行位运算时，按右对齐，位数少的在高位补“0”。
- 注意 x 对运算结果的影响

## 位运算符应用举例：

若

$$A = 5'b11001,$$

$$B = 5'b101x1,$$

则

$$\sim A = 5'b00110$$

$$A \& B = 5'b10001$$

$$A | B = 5'b111x1$$

$$A \wedge B = 5'b011x0$$



## 关系运算符

$<$	小于
$<=$	小于或等于
$>$	大于
$>=$	大于或等于

比较两个操作数的大小关系，如果比较关系成立，则返回逻辑“1”；否则，返回逻辑“0”。如果两个操作数中有一个为  $x$ ，则返回值是  $x$ 。

例：  $A = 2, B = 5, D = 4'hx$

则：  $A < B$       返回逻辑“1”

$A > B$       返回逻辑“0”

$B <= D$       返回值是  $x$





## 等式运算符

==	等于	两个操作数逐位相等，返回逻辑 1；否则，返回逻辑 0。若任何一个操作数中某位是 x 或 z，则返回 x。
!=	不等于	
===	全等	对操作数中的 x 和 z 也进行比较，逐位一致，返回逻辑 1；否则，返回逻辑 0。
!==	不全等	

双目运算，运算的结果是1位的逻辑值。

例：  $a = 5'b11x01$  ，  $c = 5'b11x01$  ， 则

$a == c$  得到的结果是 x

$a === c$  得到的结果是 1



## 归约运算符

又称为“缩减运算符”，是单目运算符。

$\&$	归约与
$\sim\&$	归约与非
$ $	归约或
$\sim $	归约或非
$\wedge$	归约异或
$\sim\wedge, \wedge\sim$	归约异或非（归约同或）

对单个操作数进行归约的递推运算，运算结果是一位二进制数。

过程：先将操作数的第1位与第2位进行归约运算，运算结果再与第3位进行归约运算，依此类推，直到最后一位。

## 归约运算符举例

...

```
reg [ 3 : 0 ] a ;
```

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

...

若  $A = 5'b11001$

则

$\& A$  归约运算结果是 0，只有各位均为1，结果才为1。

$| A$  归约运算结果是 1，只有各位均为0，结果才为0

$\wedge A$  归约运算结果是 1，奇数个1，结果为1。



## 移位运算符

>> 逻辑右移

<< 逻辑左移

双目运算，将运算符左边的操作数左移或右移运算符右边的操作数指定的位数，空位补0。

即：  $A \gg n$  或  $A \ll n$ 。

例如：  $A = 6'b110010$

则  $A \gg 2$  的结果是  $6'b001100$

$A \ll 2$  的结果是  $6'b001000$



## 条件运算符

? : 三目运算符

应用方式

*signal = condition ? true-expression : false-expression;*

信号 = 条件 ? 表达式1 : 表达式2 ;

当条件成立时，信号取表达式1  
的值，反之取表达式2 的值。

## 条件运算符举例

```
module add-subtract ( a , b , op , result ) ;  
  
    input      [ 3 : 0 ] a , b ;           //定义a,b为4位输入向量  
  
    input      op ;                       //定义op为1位输入变量  
  
    output [ 3 : 0 ] result ;             //定义result为4位输出向量  
  
    parameter    ADD = 1'b0 ;           //定义字符常量  
  
    assign result = (op == ADD ) ? a + b : a - b ;  
  
endmodule
```

当 $op$ 为0, 做 $a+b$

当 $op$ 为1, 做 $a-b$



## 拼接运算符

**{     }**     将两个或多个信号的某些位拼接起来。

用法如下：

**{ 信号1的某几位, 信号2的某几位, ... .. 信号n的某几位 }**

示例：**a = 1'b1        b = 2'b00        c = 6'b101001**

则：**{ a , b }**        产生一个3位数 **3'b100**

**{ c[ 5:3 ] , a }**    产生一个4位数 **4'b1011**

嵌套使用，进行常量或变量的复制以及简化书写。

例如：**{ 3 { a , b } }** 等同于 **{ { a , b } , { a , b } , { a , b } }**

也等同于 **{ a , b , a , b , a , b }**

在程序中，可以将相关的信号拼接在一起，便于阅读理解。


例如：描述加法运算时，常将和的输出（*sum*）与进位输出（*cout*）拼接在一起使用。

```
module add ( ina , inb , cin , sum , cout ) ;  
  
    output [ 3 : 0 ] sum ;  
  
    output cout ;  
  
    input [ 3 : 0 ] ina , inb ;  
  
    input cin ;  
  
    assign { cout , sum } = ina + inb + cin ;  
  
endmodule
```





# 运算符的优先级

运算符	优先级
! ~	高 
* / %	
+ -	
<< >>	
< <= > >=	
== != === !==	
& ~&	
^ ^~	
~	
&&	
	低
?:	

先括号内，后括号外；同级运算符自左向右运算。

不同的综合开发工具在执行这些优先级时会有微小的差异，建议养成用括号控制优先级的习惯。

## 作业6:

3.1

3.2

3.3

3.4

3.5