

第一部分 Verilog数字设计基础

第一章 Verilog的基本知识

第二章 Verilog语法的基本概念

第三章 模块的结构、数据类型、变量和基本运算符

第四章 运算符、赋值语句和结构说明语句


第五章 条件语句、循环语句、块语句与生成语句

第六章 结构语句、系统任务、函数语句和显示系统任务

第七章 调试用系统任务和常用编译预处理语句

第八章 语法概念总复习练习

第1章 Verilog的基本知识

- 1.1 硬件描述语言HDL
 - 1.2 Verilog HDL的历史
 - 1.3 Verilog HDL和VHDL的比较
 - 1.4 Verilog的应用情况和适用的设计
 - 1.5 采用Verilog HDL设计复杂数字电路的优点
 - 1.6 采用Verilog HDL的设计流程简介
 - 小结
 - 思考题
- 

1.1 硬件描述语言HDL

概念：硬件描述语言**HDL**（Hardware Description Language）是一种形式化方法来描述数字电路和系统的语言。

HDL功能：

HDL功能

数字系统仿真、验证（全部语法支持）

数字系统设计、综合（部分语法支持）

HDL两种国际标准：

IEEE标准

Verilog HDL

VHDL

1.2 Verilog HDL的历史

1.2.1 什么是Verilog HDL

1.2.1 什么是Verilog HDL

Verilog HDL是硬件描述语言的一种，用于数字系统设计。设计者可用它进行各种级别的逻辑设计，可用它进行数字逻辑系统的仿真验证、时序分析、逻辑综合。

注：它是目前应用最广泛的一种硬件描述语言。

目前在美国使用Verilog HDL进行设计的工程师大约有10万多人，全美国有200多所大学讲授Verilog语言的设计方法。在台湾地区几乎所有著名大学的电子和计算机工程系都讲授与Verilog有关的课程。

1.2.2 Verilog HDL的产生及发展

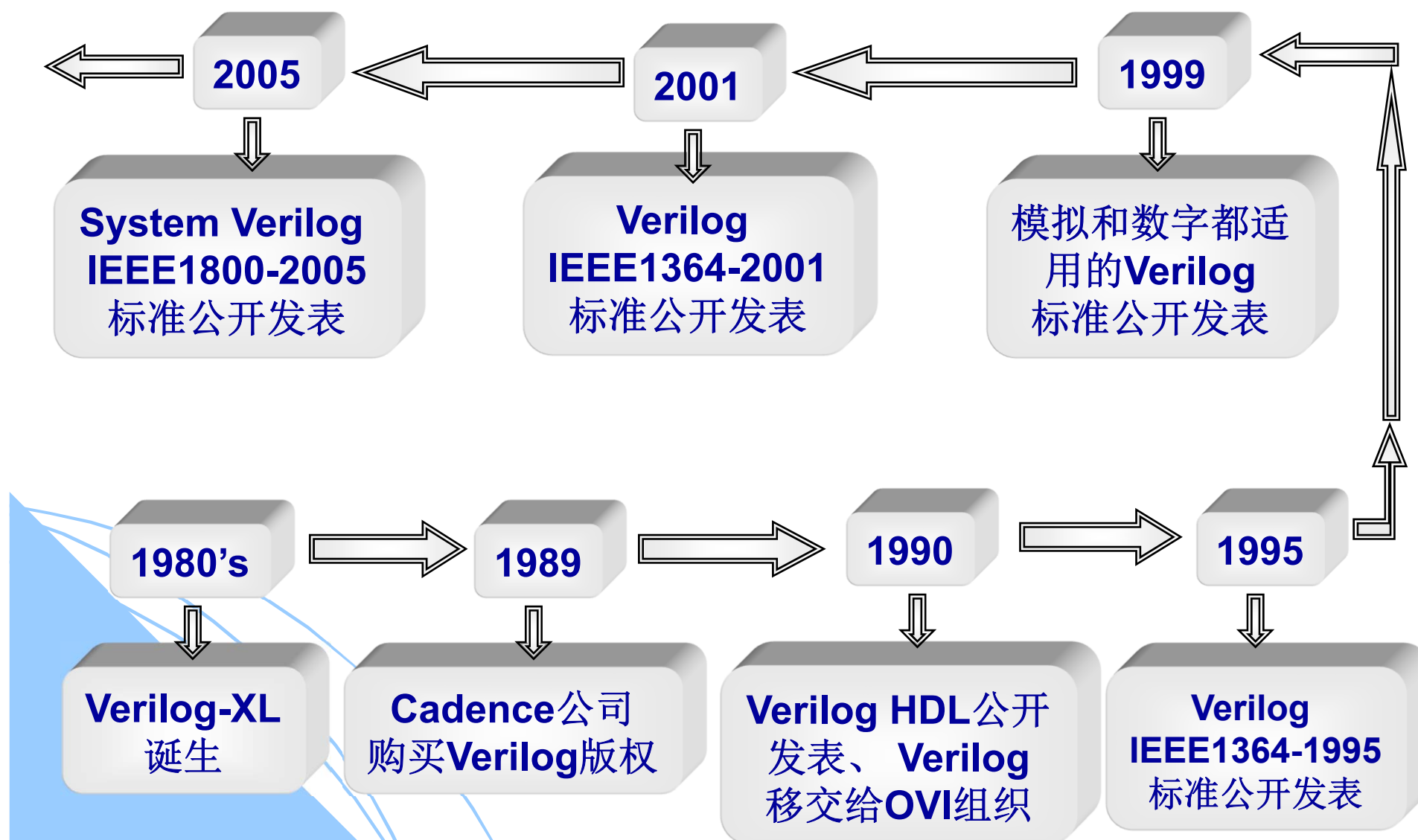


图1.1 Verilog的发展历史

1.3 Verilog HDL和VHDL的比较

共同点

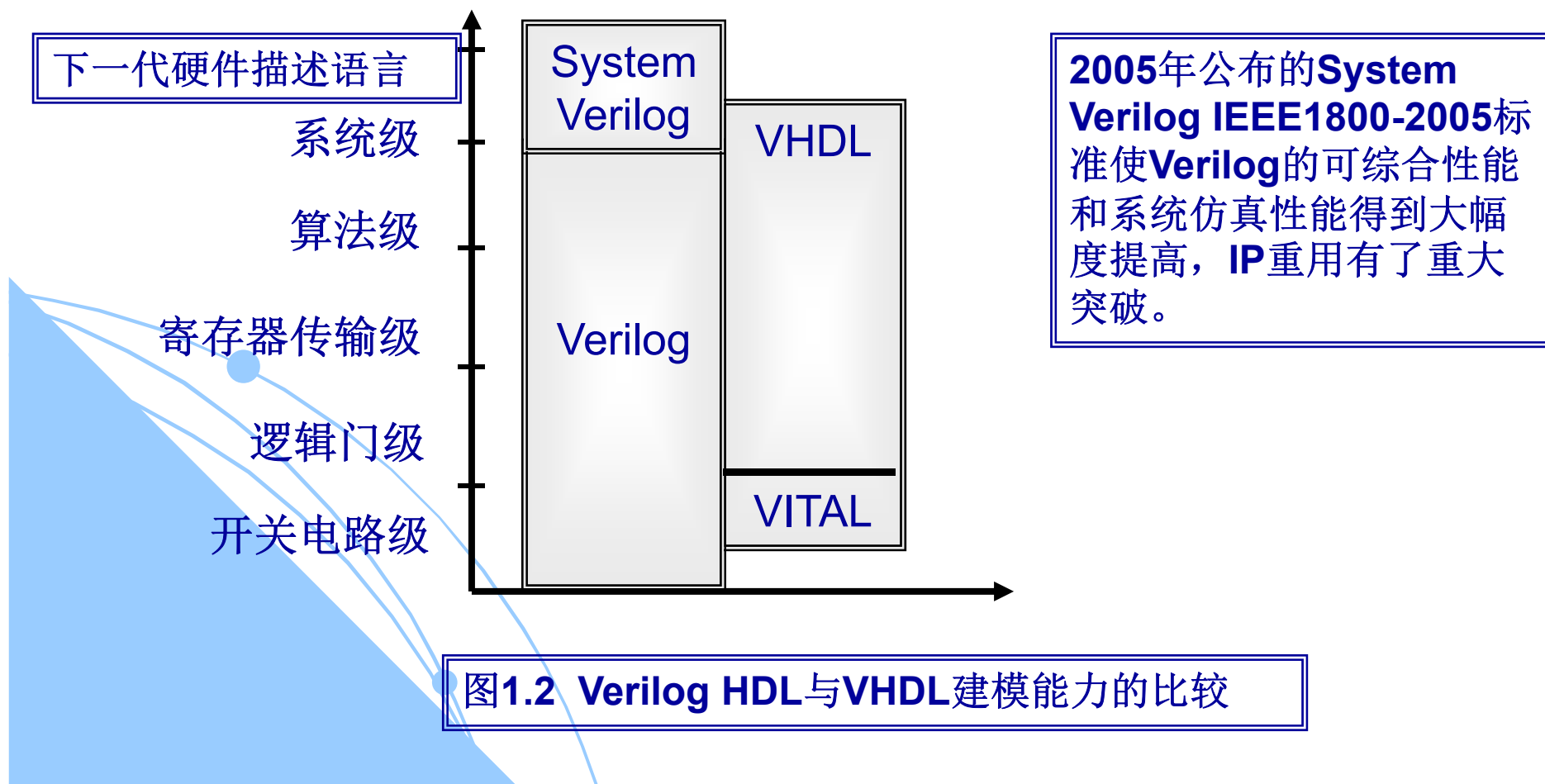
- 能形式化的抽象表示电路的行为和结构；
- 支持逻辑设计中层次与范围的描述；
- 可借用高级语言的精巧结构来简化电路行为的描述；
- 具有电路仿真与验证机制以保证设计的正确性；
- 支持电路描述由高层到低层的综合转换；
- 硬件描述与实现工艺无关；
- 便于文档管理；易于理解 and 设计重用。

区别

Verilog拥有更广泛的设计群体，成熟的资源也比VHDL丰富；

与VHDL相比，Verilog HDL容易掌握，与C语言类似。

2005年以前，Verilog在系统级抽象方面比VHDL略差一些，而在门级开关电路描述方面比VHDL强的多。2005年后，系统抽象能力得到彻底改变。



1.4 Verilog的应用情况和适用的设计

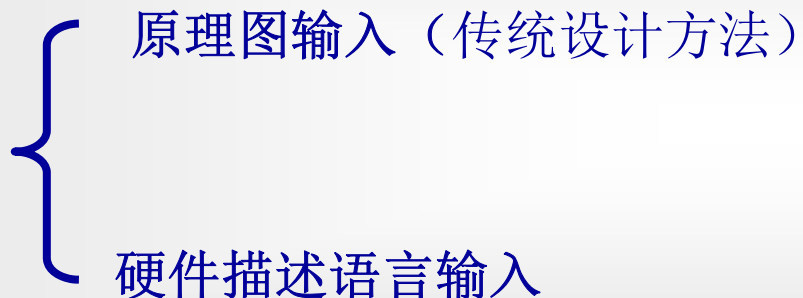
在美国，应用Verilog和VHDL的比例是80%和20%；日本和台湾与美国相同；而在欧洲VHDL发展的比较好。

Verilog HDL适用的描述层次：

- 1、系统级（**System**） 部分可物理实现，主要用于仿真
- 2、算法级（**Allogrthem**） 部分可物理实现，主要用于仿真
- 3、寄存器传输级（**RTL**） 可完全物理实现，用于电路设计
- 4、逻辑级（**Logic**） 可完全物理实现，用于电路设计
- 5、门级（**Gate**） 可完全物理实现，用于电路设计
- 6、电路开关级（**Switch**） 软件中不涉及，用于芯片设计

1.5 采用Verilog HDL设计复杂数字电路的优点

电路的两种基本计算机输入方式：



语言输入与原理图输入方式相比的优点：

- 1、容易把设计移植到不同厂家的不同芯片中去；
- 2、信号位数容易修改，可以很方便的适应不同规模的应用；
- 3、与实现工艺无关；Verilog HDL综合器生成标准的电子设计互换格式（EDIF）文件，方便文档交换与保存；

由于VerilogHDL不断更新改进，最新标准为IEEE1800-2005，使得该语言具有更广阔的发展前景，目前最新的EDA工具都进行了更新以支持最新的Verilog标准。

采用Verilog语言，设计的可重用性好。

概念：

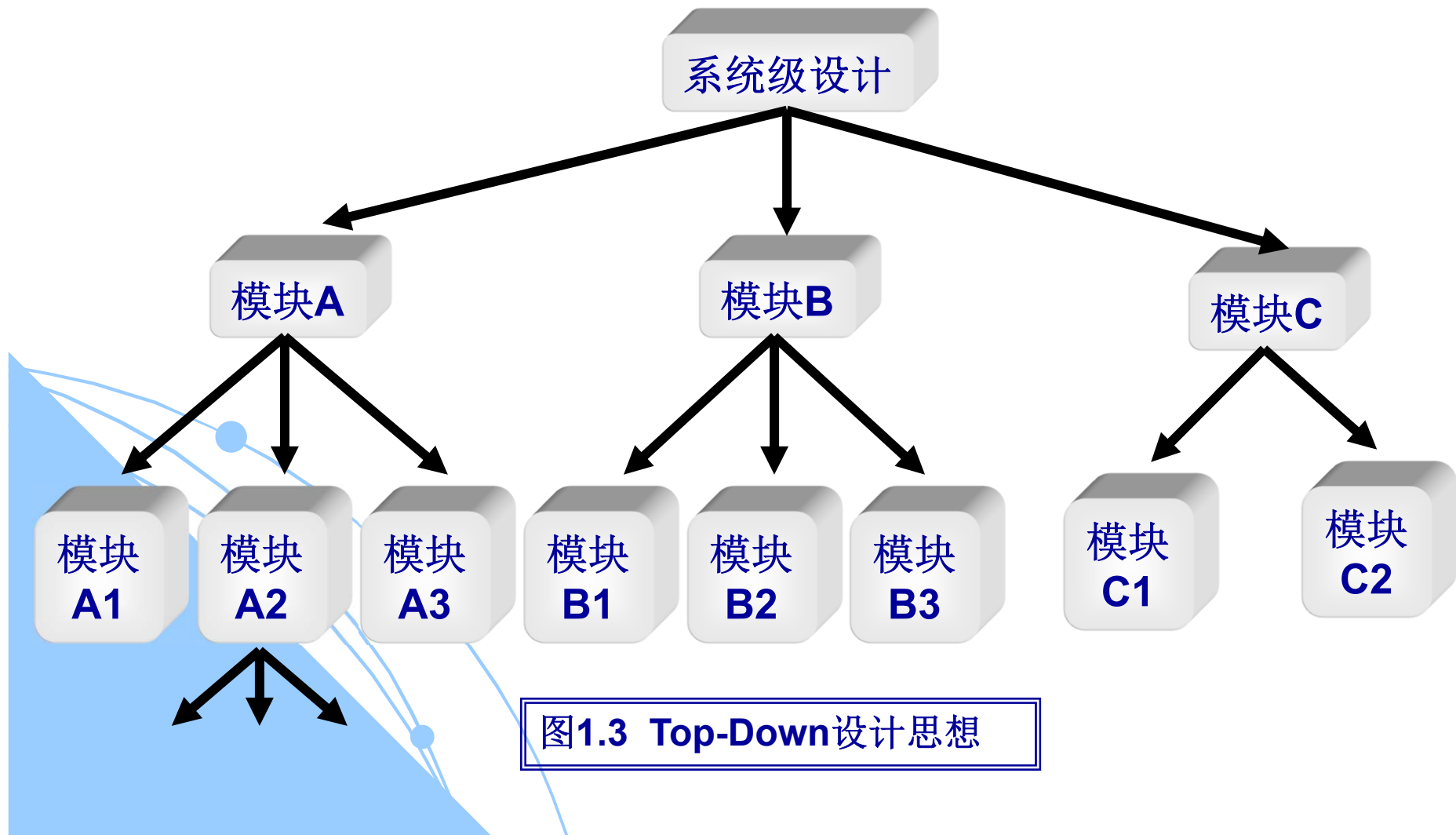
软核 (Soft Core)： 功能经过验证的、可综合的、实现后电路结构总门数在5000门以上的Verilog HDL模型。由软核构成的器件称为虚拟器件。三者中灵活性最高。

固核 (Firm Core)： 指在某一种现场可编程门阵列 (FPGA) 器件上实现的、经验证是正确的、总门数在5000门以上电路结构编码文件。

硬核 (Hard Core)： 指在某一种专用集成电路 (ASIC) 工艺的器件上实现的、经验证是正确的、总门数在5000门以上的电路结构版图掩膜。

1.6 Verilog HDL的设计流程

方法：Top-Down（自顶向下）设计思想。



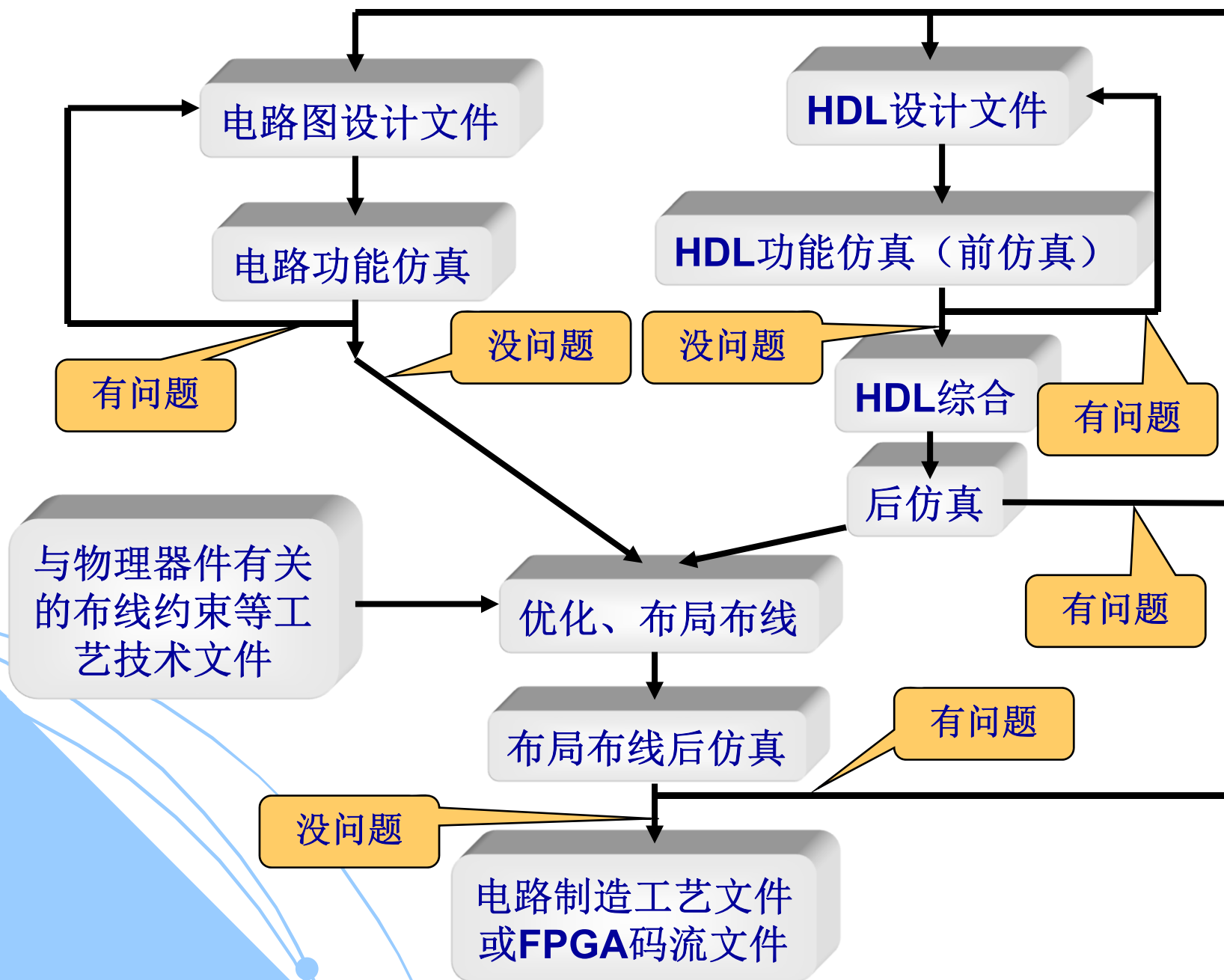


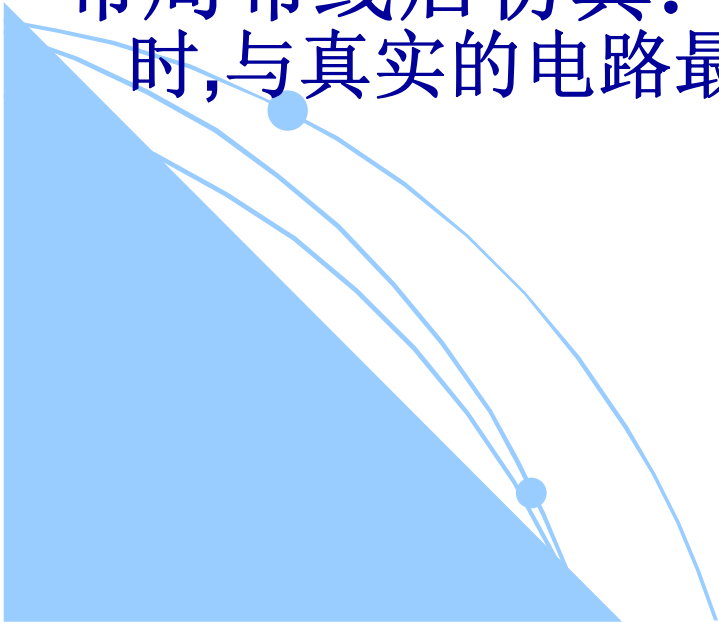
图 1.4 HDL设计流程图

行为仿真：行为的验证和验证模块分割的合理性；

前仿真：即 **RTL**级仿真，检查有关模块逻辑执行步骤是否正确。

后仿真：用门级模型做验证，检查由门的互连构成的逻辑其功能是否正确。

布局布线后仿真：在门级模型的基础上加上了布线延时,与真实的电路最接近的验证。



小结

小结:

- 1) 采用Verilog HDL设计方法比采用电路图输入的方法更有优越性;
- 2) 在两种符合IEEE标准的硬件描述语言中, Verilog HDL与VHDL相比更加基础、更易掌握;
- 3) Verilog HDL可用于复杂数字逻辑电路和系统的总体仿真、子系统仿真和具体电路综合等各个设计阶段;
- 4) Top-Down的设计方法方便了从系统级划分和管理整个项目, 使得超大规模的复杂数字电路的设计成为可能, 并可减少设计人员, 避免不必要的重复设计, 提高了设计的一次成功率。

思考题


- 1、什么是硬件描述语言？它的主要作用是什么？
- 2、目前世界上符合IEEE标准的硬件描述语言有哪两种？它们各有什么特点？
- 3、什么情况下需要采用硬件描述语言的设计方法？
- 4、采用硬件描述语言设计方法的优点是什么？有什么缺点？
- 5、简单叙述一下利用EDA工具并采用硬件描述语言（HDL）的设计方法和流程。
- 6、硬件描述语言可以用哪两种方式参与复杂数字电路的设计？
- 7、用硬件描述语言设计的数字系统需要经过哪些步骤才能与具体的电路相对应？
- 8、为什么说用硬件描述语言设计的数字逻辑系统具有最大的灵活性并可以映射到任何工艺的电路路上？
- 9、软核是什么？虚拟器件是什么？它们的作用是什么？
- 10、集成电路行业中IP的含义是什么？固核是什么？硬核是什么？与软核相比它们各有什么特点？各适用于什么场合？
- 11、简述Top-Down设计方法和硬件描述语言的关系。
- 12、System Verilog与Verilog有什么关系？适合于何种设计？

第2章 Verilog语法的基本概念

概念：用Verilog HDL描述的电路设计就是该电路的Verilog HDL模型，也称为模块——**module**。

概念：Verilog HDL既是一种**行为描述**的语言也是一种**结构描述**的语言。

也就是说，无论描述电路功能行为的模块或描述元件或较大部件互连的模块都可以用**Verilog**语言来建立电路模型。



概念： Verilog模型可以是实际电路的不同级别的抽象。这些抽象的级别和它们所对应的模型类型共有以下5种：

(1)系统级(system—level)： 用语言提供的高级结构能够实现所设计模块的外部性能模型。

(2)算法级(algorithm—level)： 用语言提供的高级结构能够实现算法运行的模型。

(3)RTL级(Register Transfer Level)： 描述数据在寄存器之间的流动和如何处理、控制这些数据流动的模型。

以上三种都属于行为描述，只有RTL级才与逻辑电路有明确的对应关系。

(4)门级(gate—level)： 描述逻辑门以及逻辑门之间连接的模型。
与逻辑电路有确定的连接关系，以上四种数字系统设计师必须掌握。

(5)开关级(switch—level)： 描述器件中三极管和储存节点以及它们之间连接的模型。

2.1 Verilog模块的基本概念

先介绍几个简单的Verilog HDL程序，了解Verilog模块的特性

【例2.1】

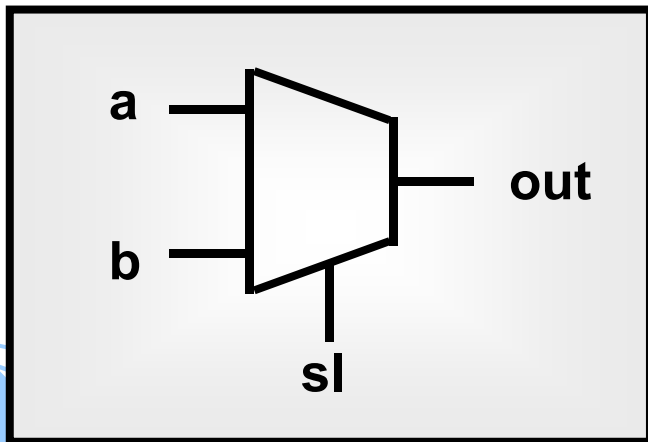


图 2.1 二选一多路器（一）

行为描述

```
module muxtwo(out, a, b, s1);  
  
    input a, b, s1; //信号属性  
    output out;    //信号属性  
    reg out;       //信号属性  
    //实现功能描述  
    always @(s1 or a or b)  
        if(!s1) out=a;  
        else out=b;  
  
endmodule
```

【例2.2】

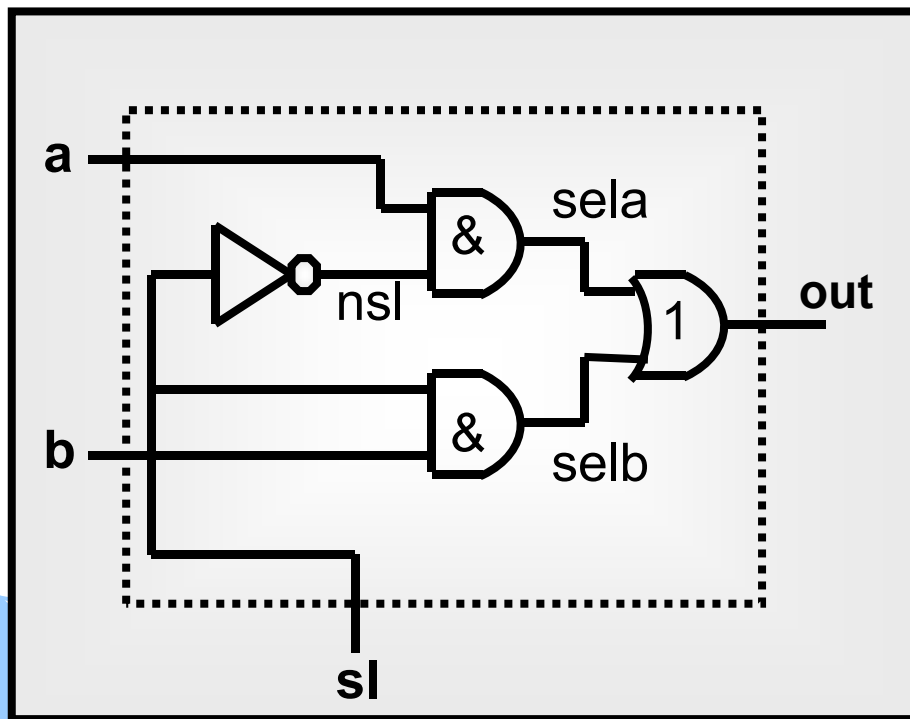


图 2.2 二选一多路器逻辑图

```
module muxtwo(out, a, b, s1);
    input a, b, s1;
    output out;
    wire ns1, sela, selb;

    assign ns1=~s1;
    assign sela=a&ns1;
    assign selb=b&s1;
    assign out=sela|selb;
endmodule
```

基于逻辑表达式
的描述

【例2.3】

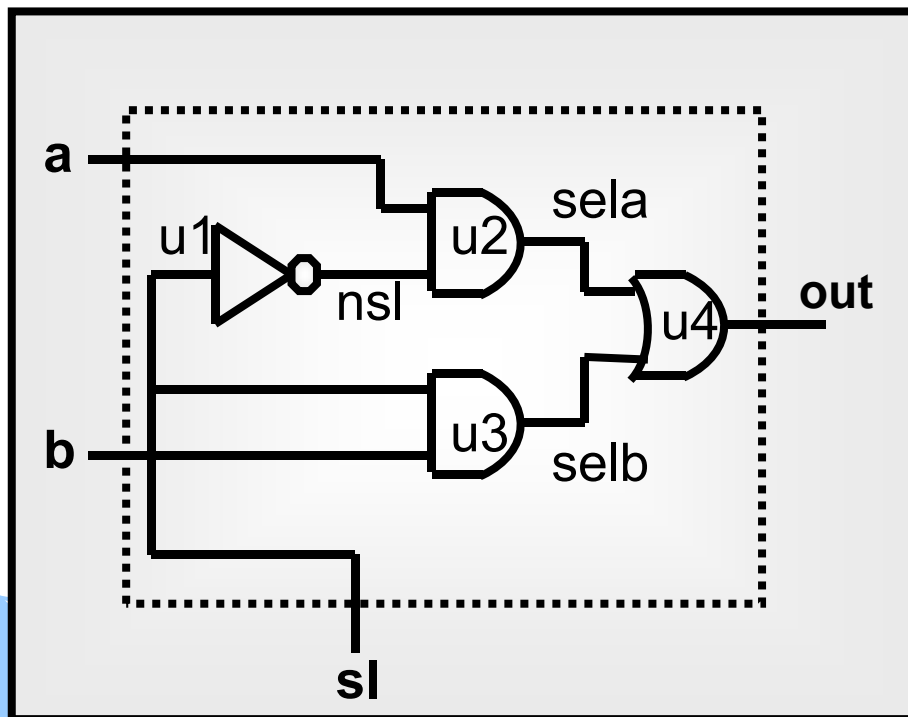


图 2.3 多路选择器（二）

```
module muxtwo(out, a, b, s1);  
    input a, b, s1;  
    output out;  
  
    not    u1(ns1, s1);  
    and    u2(sela, a, ns1);  
    and    u3(selb, b, s1);  
    or     u4(out, sela, selb);  
  
endmodule
```

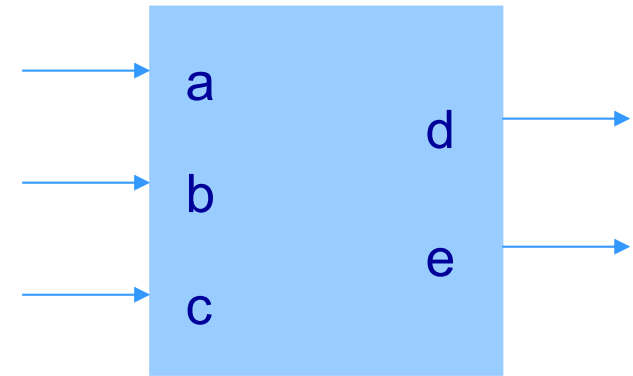
结构描述

参考返回

编写Verilog HDL模块的练习

请在下面的空格中填入适当的符号
使其成为右图的**Verilog** 模块：

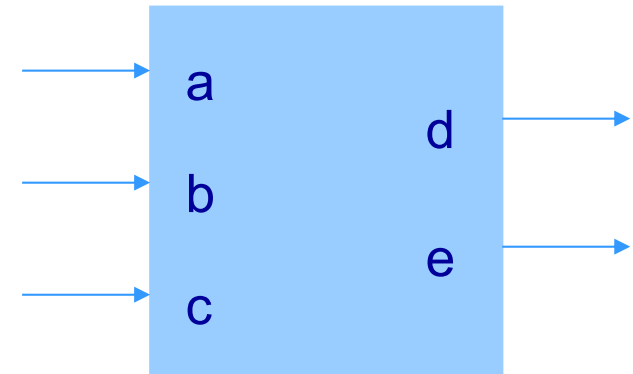
```
module block1(a, b, —, —, — );  
input  —, —, —;  
—— d, —;  
      assign d = a | ( b & ~c );  
      assign e = ( b & ~c );
```



编写Verilog HDL模块的练习

请在下面的空格中填入适当的符号
使其成为右图的**Verilog** 模块：

```
module block1(a, b, c, d, e );  
input  a, b, c;  
output  d, e ;  
        assign d = a | ( b & ~c ) ;  
        assign e = ( b & ~c ) ;  
endmodule
```



概念：综合——Synthesis

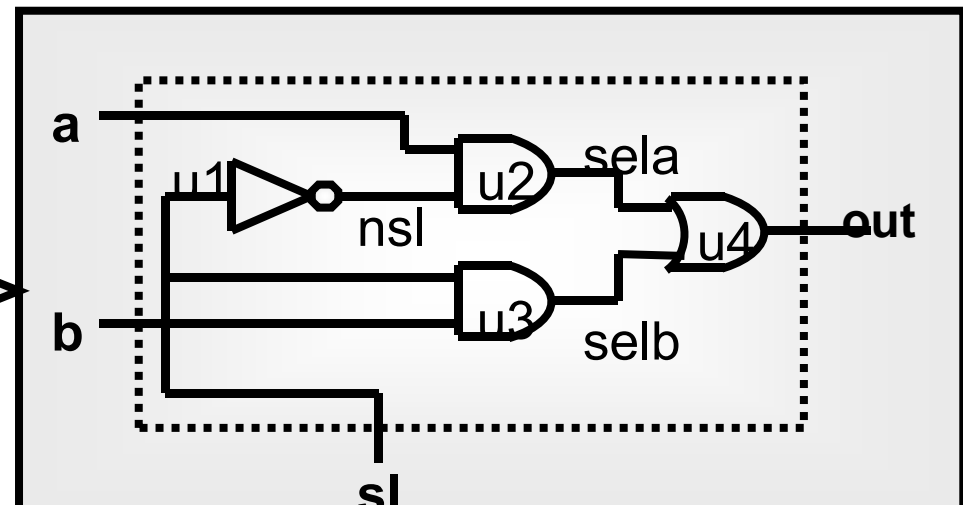
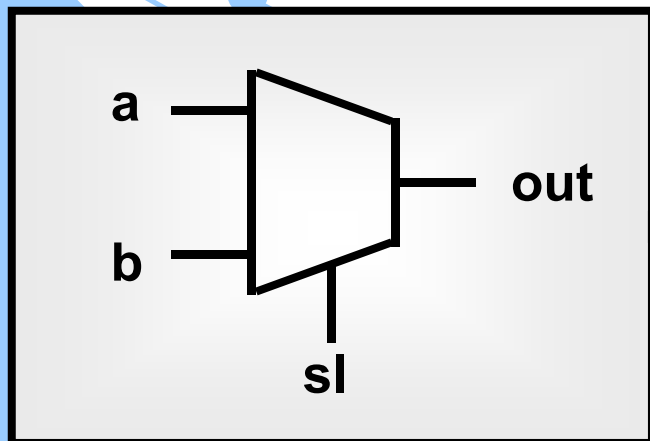
Verilog模块（程序）通过计算机上运行的综合软件工具（属EDA软件）把【例2.1】的行为描述通过【例2.2】逻辑表达式的中间形式自动转换为【例2.3】结构描述的模块，这个过程叫做综合(Synthesis)。

电路的
行为

电路的
逻辑关系

电路的
结构

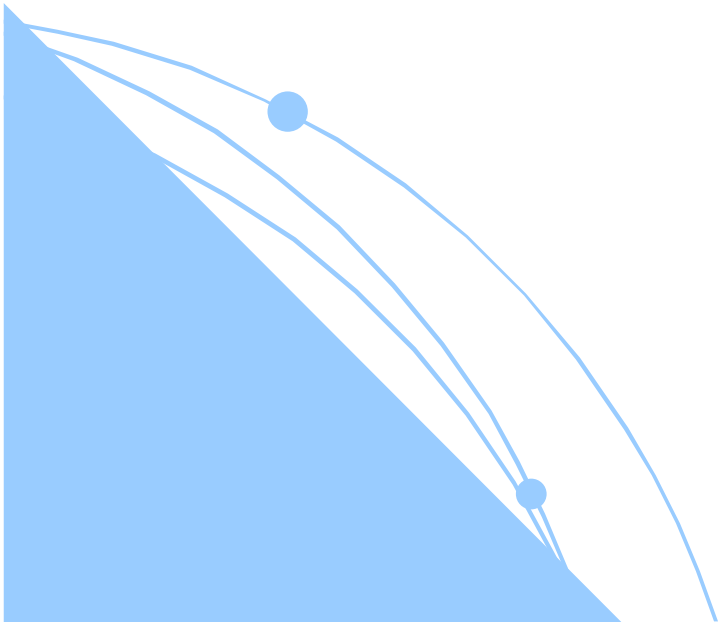
Synthesis 综合



下面再看几个简单的模块，目的是初步了解Verilog语法最重要的几个基本概念：

概念：

并行性；
层次结构性；
可综合性；
测试平台 (Testbench)



【例2.4】

```
module adder(cout,sum,a,b,cin);  
    input [2:0] a,b;  
    input cin;  
    output cout;  
    output [2:0] sum;  
    assign {cout,sum} = a+b+cin;  
endmodule
```

信号定义

连续赋值
语句

外部端
口信号
定义

模块开始
与结束

说明:

此程序通过连续赋值语句描述了一个名为adder的3位加法器。它可以根据两个3位数a、b和进位(cin)计算出和(sum)及向上进位(cout)。整个Verilog HDL程序是位于module和endmodule声明语句之间的。

【例2.5】

```
module compare(equal,a,b);  
    output equal; //声明输出信号equal  
    input [1:0] a,b; //声明输入信号a, b  
    assign equal= (a==b) ? 1:0;  
    /*a,b相等，输出为1；否则为0*/  
endmodule
```

程序
注释

说明：

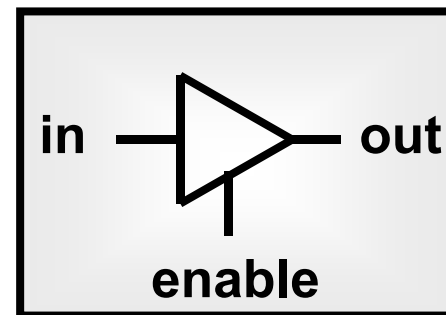
这个程序通过连续赋值语句描述了一个名为compare的比较器。对2比特数a、b进行比较，如a与b相等，则输出equal为高电平，否则为低电平。

/*.....*/ 和 //表示注释部分。

【例2.6】

元件例化
instance

```
module trist2(out,in,enable);  
    output out;  
    input in,enable;  
    bufif1 mybuf(out,in,enable);  
endmodule
```



说明:

这个程序描述了一个名为**trist2**的三态驱动器。通过调用Verilog语言提供的原语库中的三态驱动器元件**bufif1**来实现其逻辑功能。在**trist2**模块中所用到的三态驱动器**bufif1**的具体名字叫做**mybuf**，这种引用现成元件或模块的方法叫做实例化或实例引用，是表示电路构造的一种常用语法现象。

【例2.7】

顶层

```
module trist1(sout,sin,ena);  
    output sout;  
    input sin,ena;  
    mytri tri_inst(.out(sout), .in(sin), .enable(ena));  
endmodule
```

底层
子模块

```
-----  
module mytri(out,in,enable);  
    output out;  
    input in,enable;  
    assign out=enable ? in: 'bz;  
endmodule
```

参考返回

说明：在实例部件tri_inst中，带“.”表示被引用模块的端口，名称必须与被引用模块的端口一致；
小括号中名称为本模块信号，表示与被调用模块的连接关系。

- 在**Verilog** 模块中有三种方法可以生成逻辑电路:

- 用 **assign** 语句:

```
assign cs = ( a0 & ~a1 & ~a2 );
```

- 用 元件的实例调用:

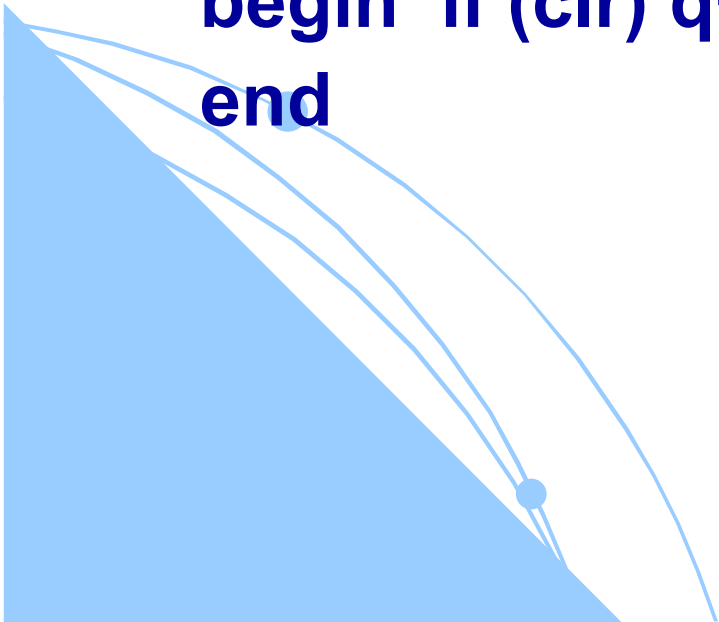
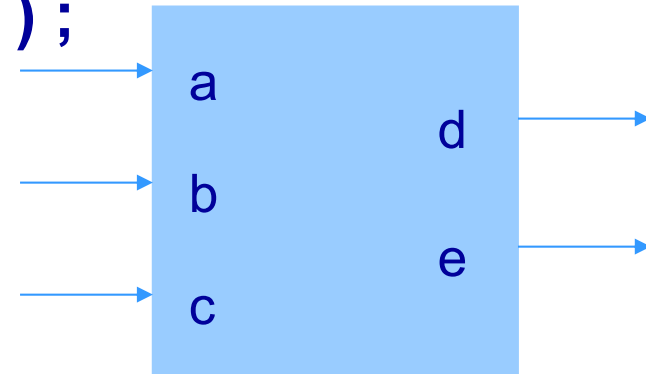
```
and2 and_inst ( q, a, b);
```

- 用 **always** 块:

```
always @ (posedge clk or posedge clr)
```

```
begin if (clr) q<= 0; else if (en) q<= d;
```

```
end
```



- 如在模块中逻辑功能由下面三个语句块组成：

```
assign cs = ( a0 & ~a1 & ~a2 );    // -----1
```

```
and2 and_inst ( qout, a, b);        // -----2
```

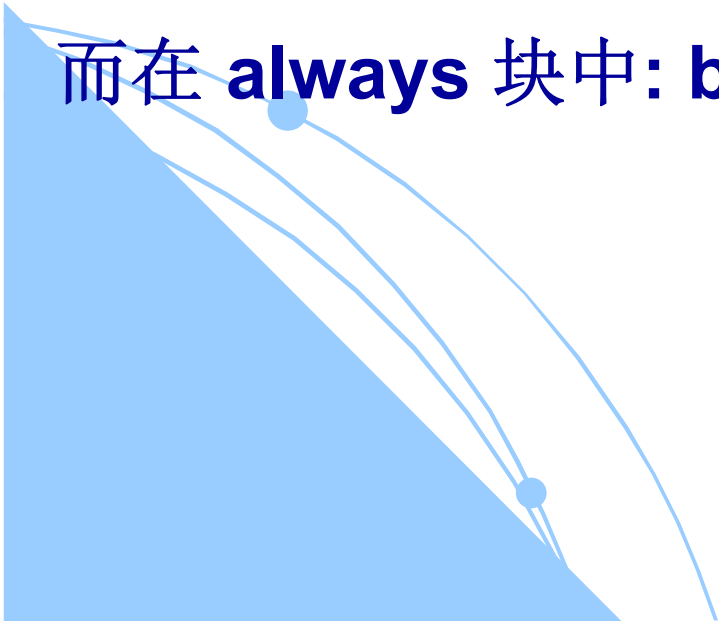
```
always @ (posedge clk or posedge clr) //-----3
```

```
begin if (clr) q<= 0; else if (en) q<= d;
```

```
end
```

三条语句是并行的，它们产生独立的逻辑电路；

而在 **always** 块中: **begin** 与 **end** 之间是顺序执行的。



说明:

上面这些例子都是可以综合的，通过综合工具可以自动转换为由与门、或门和非门组成的加法器、比较器和三态门等组合逻辑。

模块中最重要的部分是逻辑功能定义部分。有3种方法可在模块中产生逻辑。

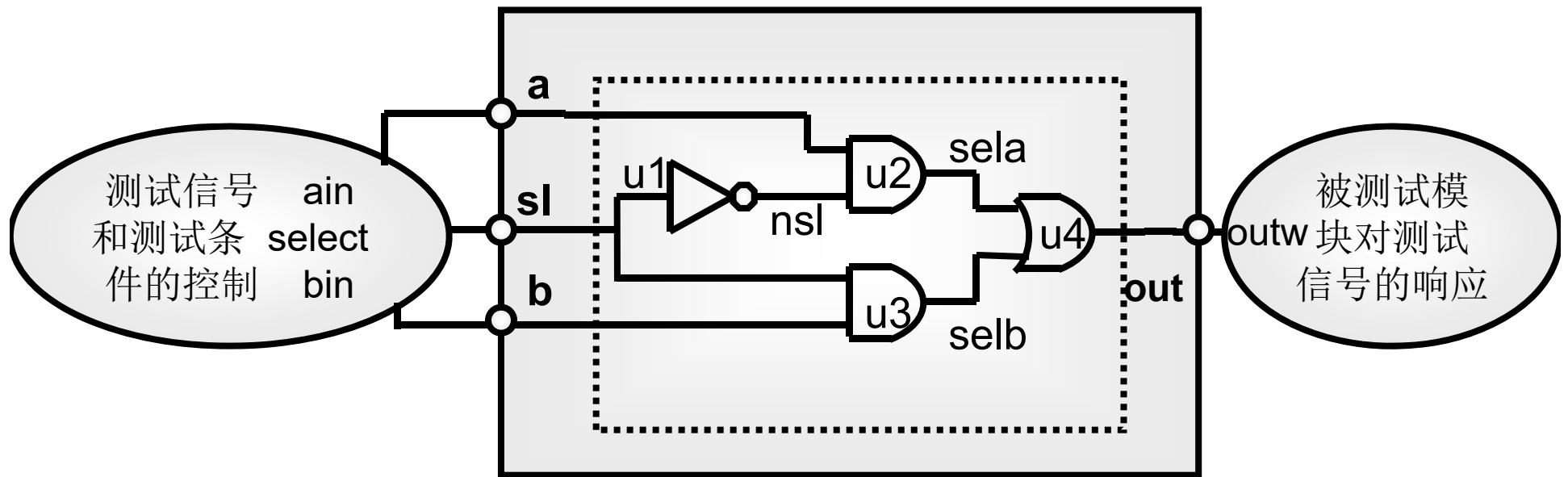
- 用 “**assign**” 连续赋值语句
- 用元件例化方法（即元件调用）
- 用 “**always**” 块

2.2 Verilog用于模块的测试

概念:

描述测试信号的变化和测试过程的模块叫做测试平台
(**Testbench**或**TestFixture**)。

Verilog可以用来描述变化的测试信号。测试平台可以对上面介绍的电路模块(无论是行为的或结构的)进行动态的全面测试。通过观测被测试模块的输出信号是否符合要求,可以调试和验证逻辑系统的设计系统和结构正确与否,并发现问题及时修改。



返回元
件例化

图 2.5 Verilog用于模块测试

【例2.8】

Verilog测试模块，对例2.1~2.3的多路器进行全面的测试。

```
`include "muxtwo.v"
module t;
    reg ain,bin,select;
    reg clock;
    wire outw;

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

    always #50 clock=~clock;
    always @(posedge clock)
        begin
            #1 ain={$random}%2;
            #3 bin={$random}%2;
        end

    always #10000 select=!select;
    muxtwo m(.out(outw),.a(ain),.b(bin),.sl(select));
endmodule
```

程序说明:

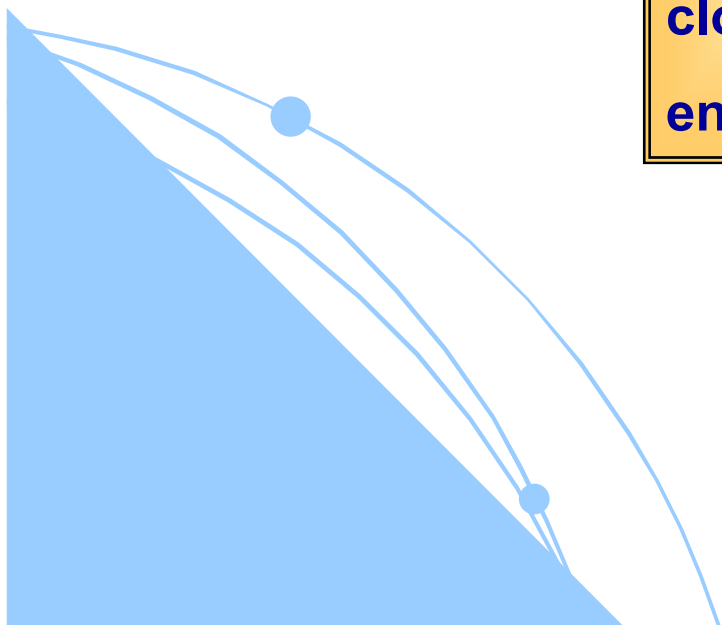
1

信号的初始化

```
initial  
begin  
  ain=0;  
  bin=0;  
  select=0;  
  clock=0;  
end
```

initial块只执行一次

把寄存器型变量
初始化为确定值，
即0时刻值。

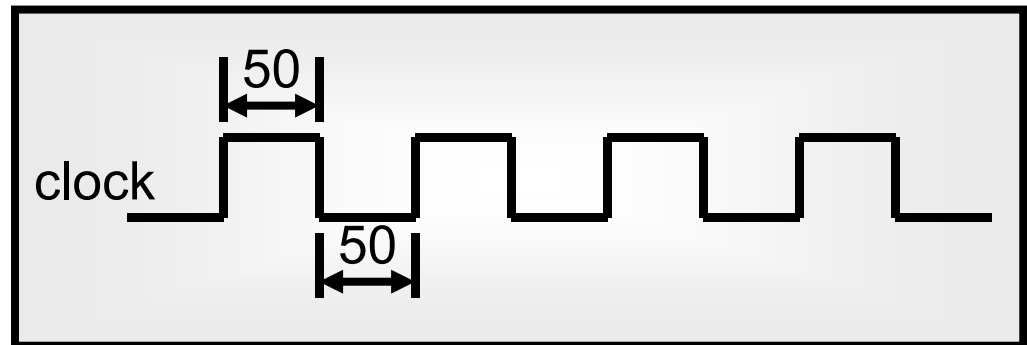


时钟信号的产生方法

```
always #50 clock=~clock;
```

always块循
环执行

产生一个不断重复
的、周期为100个
单位时间的时钟信
号clock

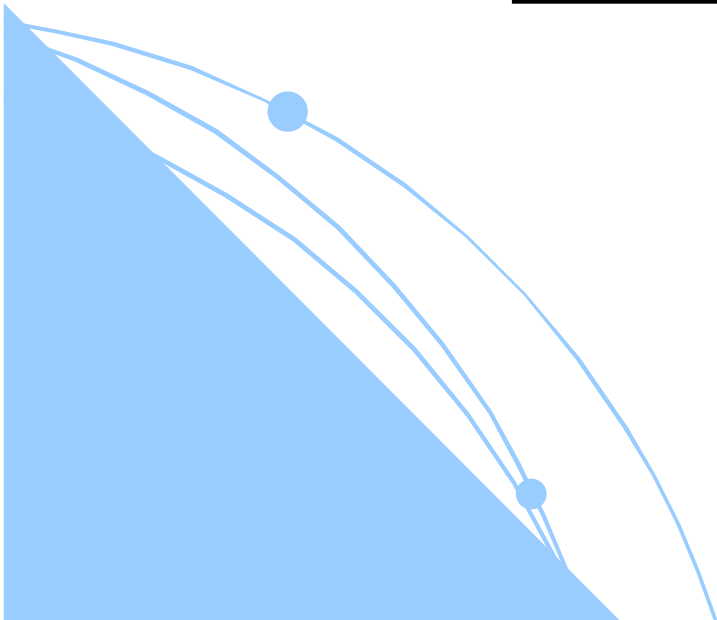
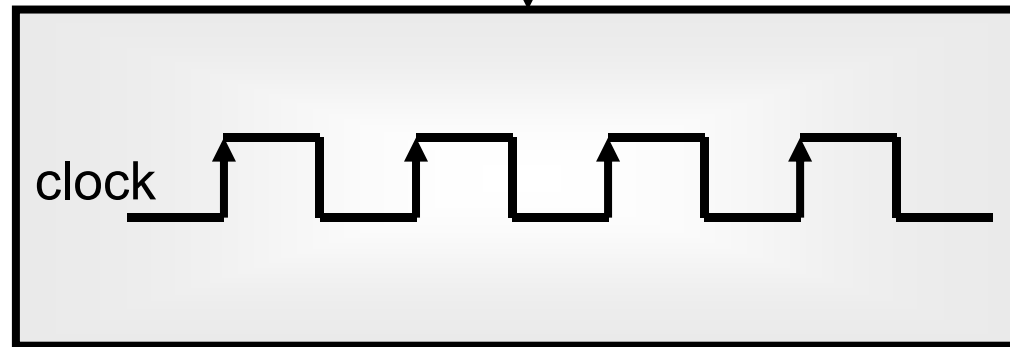


3

时钟信号的上升沿触发

posedge 上升沿
negedge 下降沿

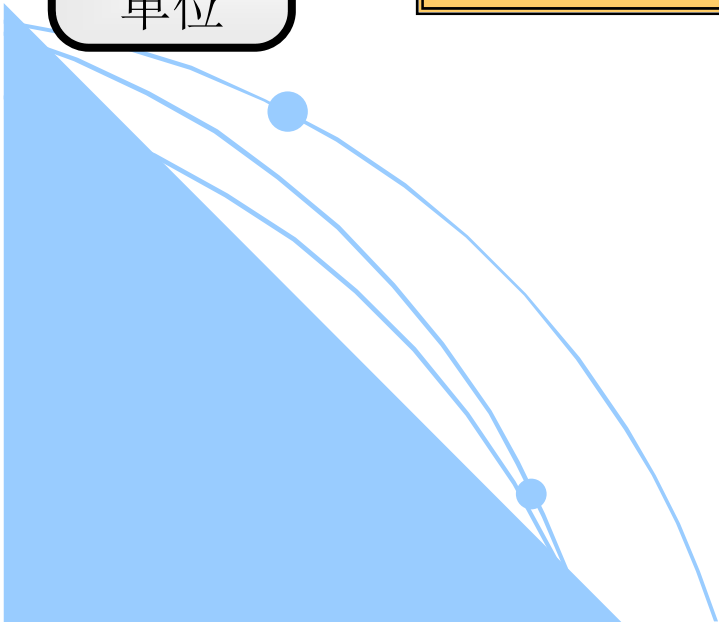
always @(posedge clock)



```
always @(posedge clock)
begin
    ain={$random}%2;
    #3 bin={$random}%2;
end
```

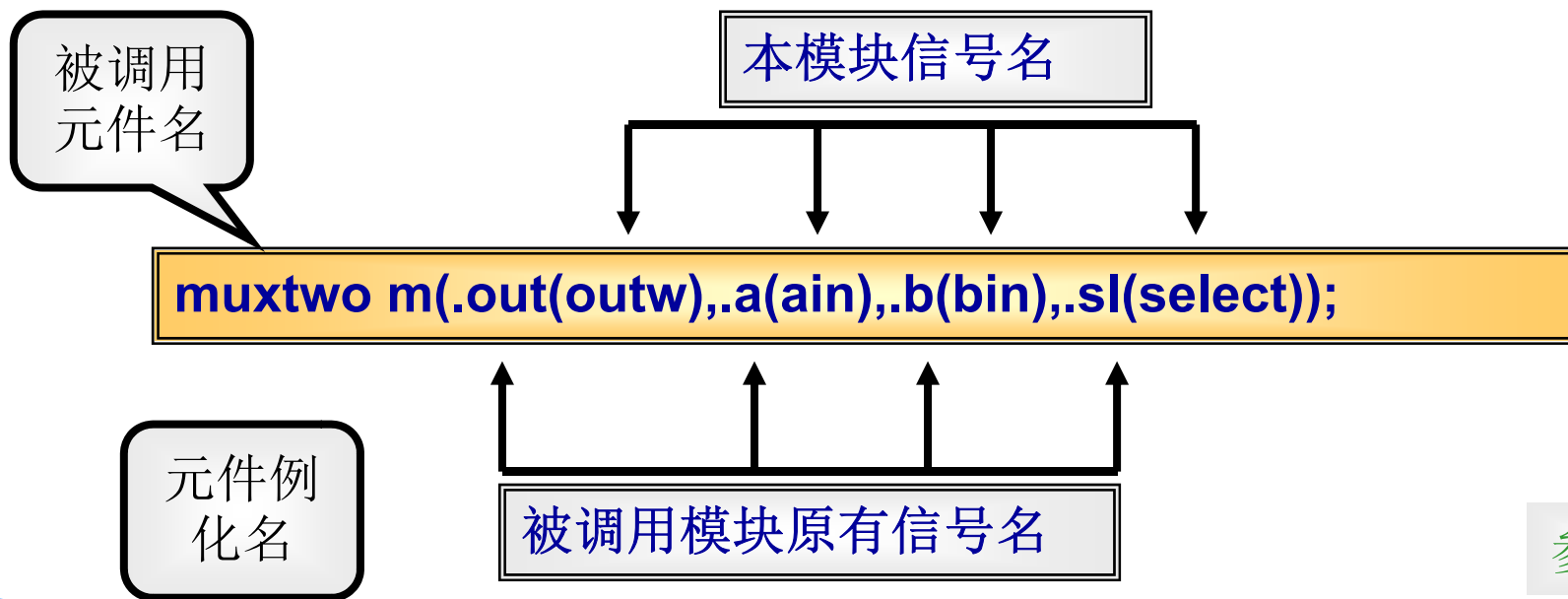
\$random-产生随机数
{ \$random }%2为产生
0-1的随机数

延时3
个时间
单位



5

元件的调用



参考

说明：m表示在本测试模块中有一个名为m的muxtwo的模块；

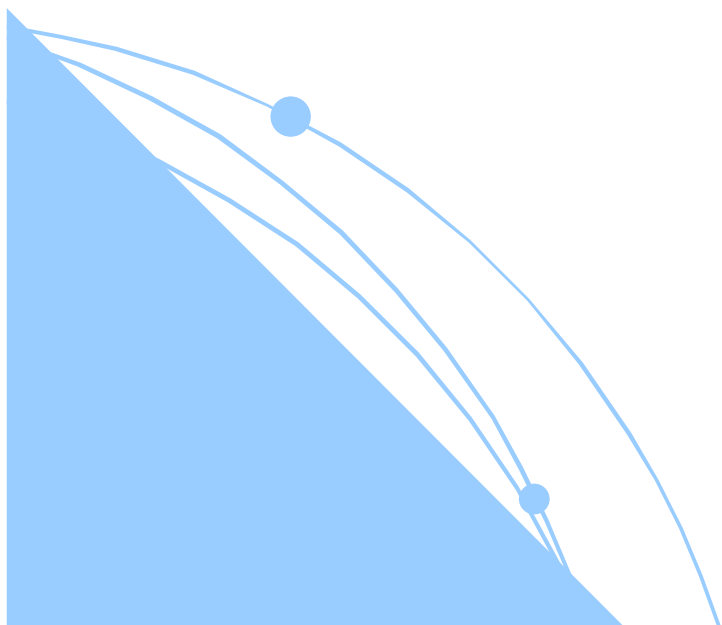
其四个端口分别为：.out()，.a()，.b()，.sl()；

“.”表示端口，后面为端口名，名称必须与muxtwo模块定义的端口名一致；小括号内的信号名为与该端口连接的信号线名，必须在本模块中定义，说明其类型。

Verilog模块的编写和验证举例

--- 8位加法器的设计和验证 ---

新设计方法的总体印象



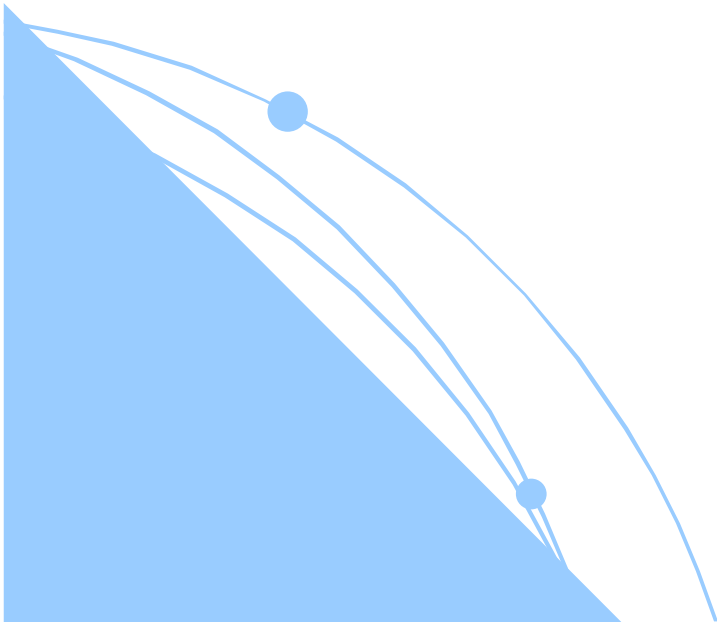
Verilog HDL入门

```
module myadder(clock, reset, a, b, sum);
parameter width = 8;
input clock, reset;
input [width-1:0] a, b;
output [width :0] sum;
reg [width-1:0] a_reg, b_reg;
reg [ width : 0 ] sum;

always @(posedge clock or negedge reset)
    if (!reset) begin
        a_reg <= ' b0; b_reg <= ' b0; sum<= ' b0;
    end
end
```

```
else begin
    a_reg <= a;
    b_reg <= b;
    sum <= a_reg + b_reg ;
end
```

```
endmodule
```



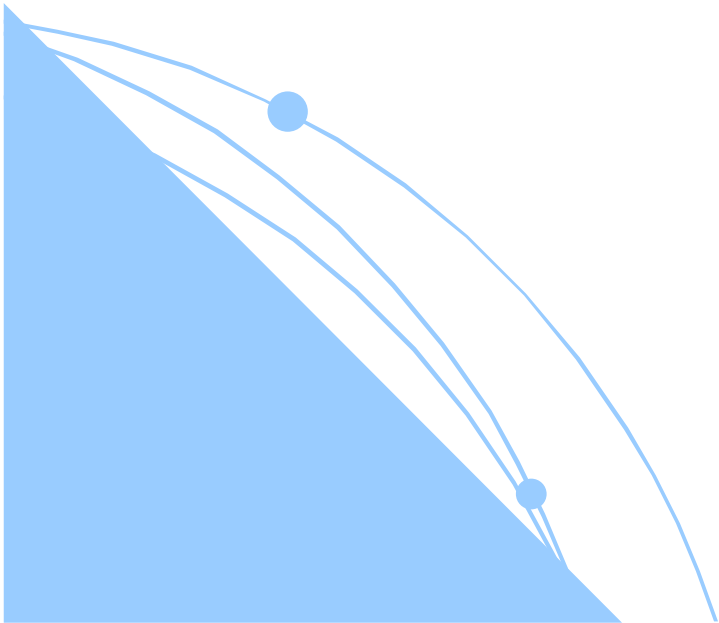
Verilog HDL模块的测试

```
`include myadder.v
module t;
  wire [8:0] sumout;
  reg [7:0] ain, bin;
  reg rst, clk;
  myadder(.clock(clk), .reset(rst), .a(ain), .b(bin), .sum(sumou
t));
  initial begin rst = 1; clk = 0; ain = 0; bin=3; #70 rst=0;
# 70 rst = 1; end
  always #50 clk = ~clk;
  always @(posedge clk)
    begin #2 ain = ain + 2; #3 bin = bin +5; end
endmodule
```

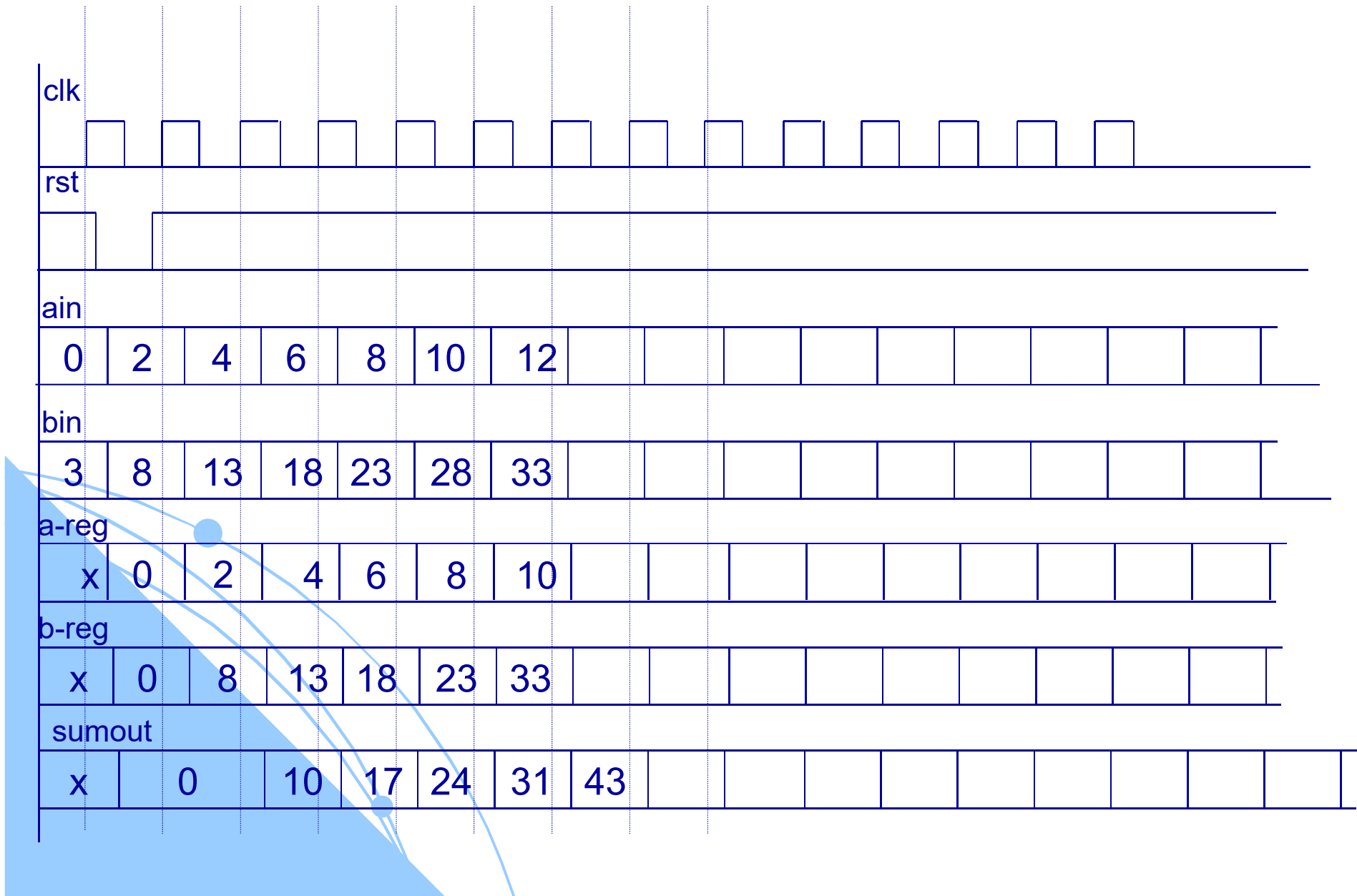
t 模块中Verilog HDL语句的功能 可以对**myadder** 模块进行测试，**myadder** 模块输入了必须的信号：

rst, clk, ain, bin

观测该模块的输出：**sumout** 看一看它是否符合设计要求。



有延迟的门级结构加法器的波形图



2.3 小结

(1) Verilog HDL程序由模块构成。每个模块的内容都位于`module`和`endmodule`两个语句之间。每个模块实现特定的功能。

(2) 模块是可以进行层次嵌套的。因此，可以将大型的数字电路设计分割成不同的小模块来实现特定的功能。

(3) 如果每个模块都是可以综合的，则通过综合工具可以把它们的功能描述全都转换为最基本的逻辑单元描述，最后可以用一个顶层模块通过实例引用把这些模块连接起来，整合成一个很大的复杂逻辑系统。

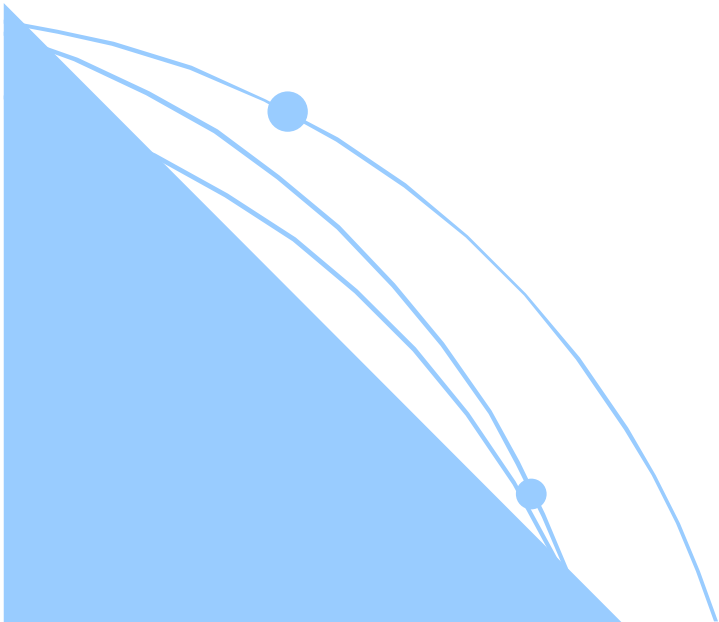
(4) Verilog模块可以分为两种类型：一种是为了让模块最终能生成电路的类型，另一种只是为了测试所设计电路的逻辑功能是否正确。

(5) 每个模块要进行端口定义，并说明它是输入口还是输出口，然后对模块的功能进行描述。

(6)Verilog HDL程序的书写格式自由，一行可以写几个语句，一个语句也可以分写多行。

(7)除了**endmodule**语句外，每个语句和数据定义的最后必须有分号。

(8)可以用 `/*.....*/` 和 `//`对Verilog HDL程序的任何部分作注释。一个好的、有使用价值的源程序都应当加上必要的注释，以增强程序的可读性和可维护性。



思考题

- 1、Verilog语言有什么作用？
- 2、构成模块的关键词是什么？
- 3、为什么说可以用Verilog构成非常复杂的电路结构？
- 4、为什么可以用比较抽象的描述来设计具体的电路结构？
- 5、是否任意抽象的符合语法的Verilog模块都可以通过综合工具转变为电路结构？
- 6、什么叫综合？
- 7、综合是用什么工具来完成的？
- 8、通过综合产生的是什么？产生的结果有什么用处？
- 9、仿真是什么？为什么要进行仿真？
- 10、仿真可以在几个层面上进行？每个层面的仿真有什么意义？
- 11、模块的端口是如何描述的？
- 12、在引用实例模块的时候，如何在主模块中连接信号线？
- 13、如何产生连续的周期性测试时钟？
- 14、如果不用initial块，能否产生测试时钟？
- 15、从本讲的简单例子，是否能明白always块与initial块有什么不同？
- 16、为什么说Verilog可以用来设计数字逻辑电路和系统？

第3章 模块的结构、数据类型、变量和基本运算符

- 3.1 模块的结构
- 3.2 数据类型及其常量和变量
- 3.3 运算符及表达式

小结

思考题



3.1 模块的结构

Verilog的基本设计单元是“模块”。其组成有两部分。一部分描述接口，另一部分描述逻辑功能，即定义输入是如何影响输出的。

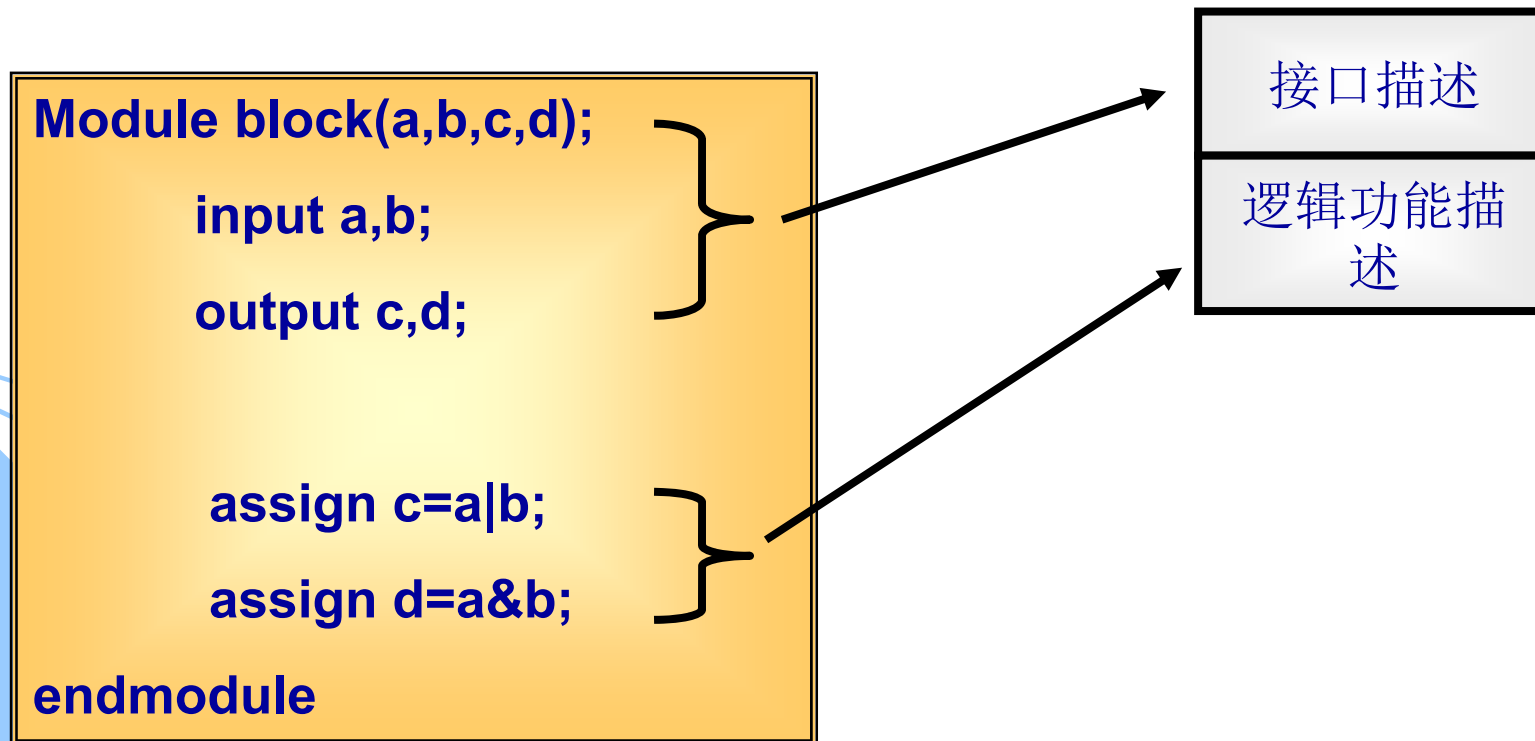
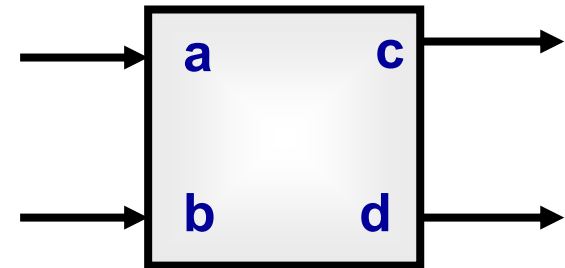


图 3.1 模块结构的组成

程序中的接口描述就是用来说明电路图符号的引脚名称及其信号流向；逻辑功能描述说明了该电路符号的具体逻辑功能。

```
Module block(a,b,c,d);  
    input a,b;  
    output c,d;  
    assign c=a|b;  
    assign d=a&b;  
endmodule
```



每个Verilog程序包括4个主要部分：

程序组成

端口定义

I/O说明

内部信号声明

功能定义

3.1.1 模块的端口定义

语法:

模块的端口声明了模块的输入输出口。其格式如下:

module 模块名(口1, 口2, 口3, 口4,);

说明:

模块的端口表示的是模块的输入和输出口名,也就是与其它模块联系端口的标识。在模块被引用时,在引用的模块中,有些信号要输入到被引用的模块中,有些信号需要从被引用的模块中取出来。

语法:

在引用模块时其端口的两种连接方法:

(1)按顺序引用

在引用时, 严格按照模块定义的端口顺序来连接, 不用标明原模块定义时规定的端口名, 例如:

模块名(连接端口1信号名, 连接端口2信号名, 连接端口3信号名,,);

参考程序

(2)按名称引用

在引用时用“.”符号, 标明原模块定义时规定的端口名, 例如:

模块名(. 端口1名(连接信号1名), 端口2名(连接信号2名),,);

参考程序

3.1.2 模块内容

模块的内容包括I / O说明、内部信号声明和功能定义。

1. I / O说明的格式

语法:

输入口: **input** 端口名1; //一位宽信号定义
input [信号位宽-1: 0] 端口名1;
input [信号位宽-1: 0] 端口名2;
.....
input [信号位宽-1: 0] 端口名i; //(共有i个输入口)

语法:

输出口: **output** 端口名1; //一位宽信号定义
output [信号位宽-1: 0] 端口名1;
output [信号位宽-1: 0] 端口名2;
.....
output [信号位宽-1: 0] 端口名j; //(共有j个输出口)

语法:

输入 / 输出口:

inout 端口名1; //一位宽信号定义

inout [信号位宽-1: 0] 端口名1;

inout [信号位宽-1: 0] 端口名2;

.....

inout [信号位宽-1: 0] 端口名k; //(共有k个双向总线端口)

语法:

I / O说明也可以写在端口声明语句里。其格式如下:

module module_name(input port1, input port2, ...
output port1, output port2...);

2. 内部信号说明

在模块内用到的和与端口有关的两种变量类型：**wire**、**reg**

语法：

wire 变量1，变量2....:

wire [width-1: 0] 变量1，变量2....:

reg 变量1，变量2....;

reg [width-1: 0] 变量1，变量2....;

说明：

用input定义的端口信号没有类型说明，只有output定义的信号和内部信号具有wire、reg类型声明要求。信号定义为output类型时默认隐含为wire型。

3. 功能定义

模块中最重要的部分是逻辑功能定义部分。有3种方法可在模块中产生逻辑。

- 用 “**assign**” 连续赋值语句
- 用元件例化方法（即元件调用）
- 用 “**always**” 块

注意：

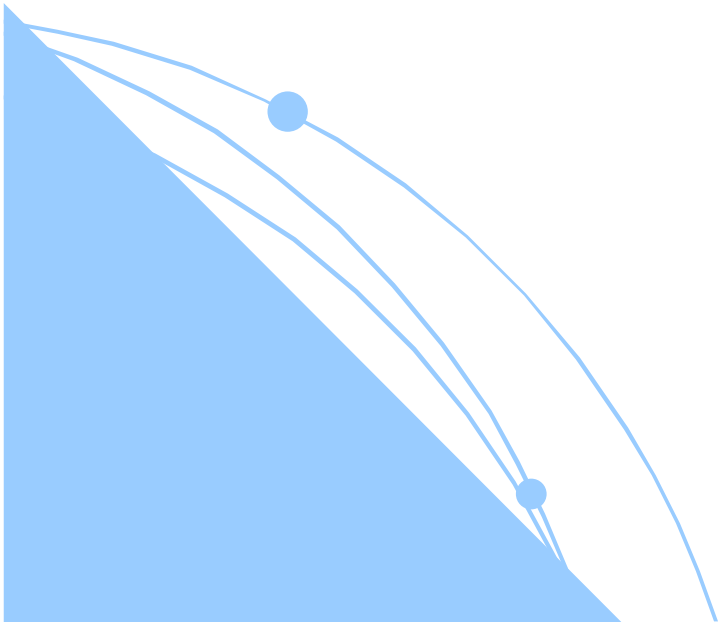
前面程序中出现的**initial**块，只能用于测试平台程序。

1) 用"assign"声明语句

这种方法的句法很简单，只需写一个“assign”，后面再加一个方程式即可。如：

assign a=b & c;

例中的方程式描述了一个有两个输入的与门。

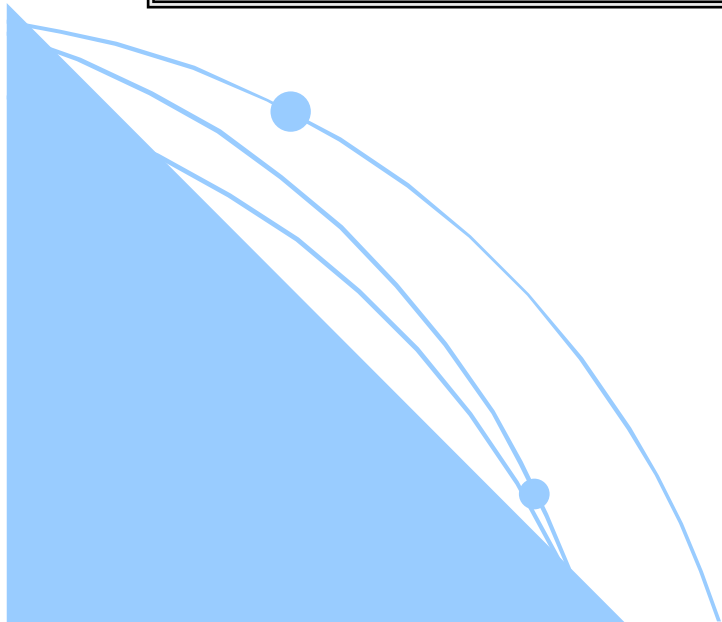


2) 用元件例化方法（即元件调用）

采用实例元件的方法像在电路图输入方式下调入库元件一样，键入元件的名字和相连的引脚即可，如：

and #2 u1(q, a, b);

表示在设计中用到一个跟与门(and)一样的名为u1的与门，其输入端为a，b，输出为q。输出延迟为2个单位时间。要求每个实例元件的名字必须是唯一的，以避免与其他调用与门(and)的实例混淆。



3) 用 “always”块

采用 “assign”语句是描述组合逻辑最常用的方法之一。而
“always”块既可用于描述组合逻辑也可描述时序逻辑。

“always”块可用很多种描述手段来表达逻辑，例如下面的程序中
就用了if...else语句来表达逻辑关系。

“带异步清零端的D触发器” 的描述

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

赋值运算符

“与门” 的描述:

```
always @(a or b)
begin
if ((a==1)&&(b==1))
c='b1;
else
c='b0;
end
```

3.1.3理解要点

要点:

上面的例子分别采用了“assign”语句、实例元件和“always”块。这3个例子描述的逻辑功能是同时执行的。也就是说，如果把这3项写到一个Verilog模块文件中去，它们的位置顺序不会影响实现的功能。这3项是同时执行的，也就是并发的。

要点:

在“always”模块内，逻辑是按照指定的顺序执行的。“always”块中的语句称为“顺序语句”。但是，两个或更多的“always”模块之间是同时执行的，但是模块内部的语句是顺序执行的。

3.1.4 要点总结

牢记：

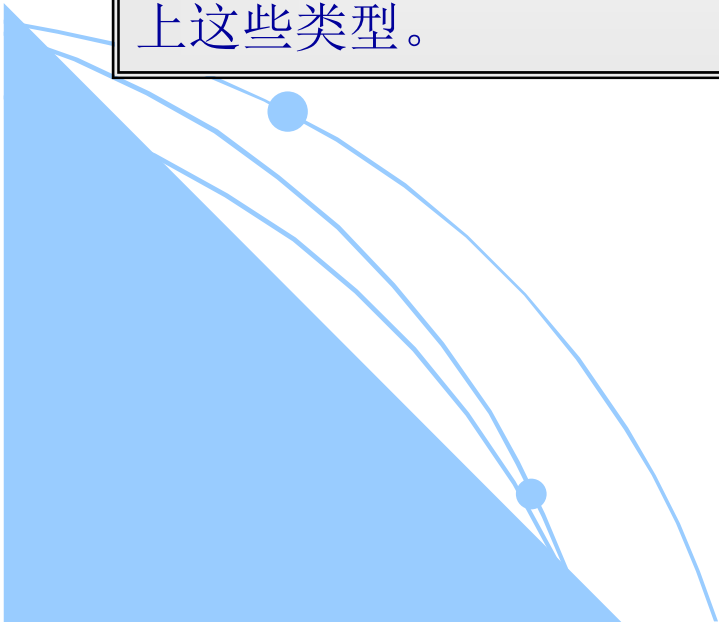
- (1)在**Verilog**模块中所有过程块(**initial**块、**always**块)、连续赋值语句、实例引用都是并行的；
- (2)实例引用表示的是一种通过变量名实现互相连接的关系；
- (3)在同一模块中这三者出现的先后顺序没有关系；
- (4)只有连续赋值语句**assign**和实例引用语句可以独立于过程块而存在于模块的功能定义部分。

3.2 数据类型及其常量和变量

Verilog HDL中总共有19种数据类型。数据类型是用来表示数字电路硬件中的数据储存和传送元素的。需要掌握的常用类型有4种：

reg型
wire型
integer型
parameter型

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

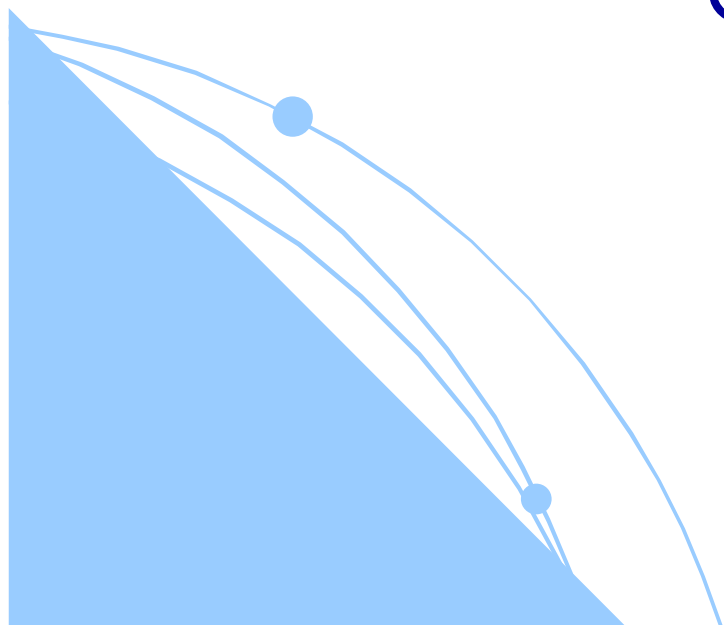


3.2.1 常 量

概念:

在程序运行过程中，其值不能被改变的量称为常量。

常量 { 数字常量
符号常量

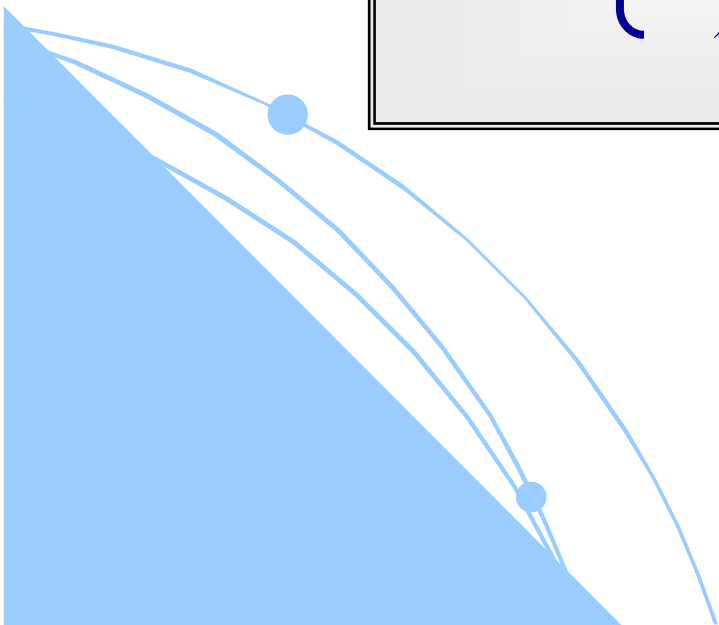


1. 数字

(1) 整数

整型常量有4种进制表示形式：

进制 { 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)。



数字表达方式有以下3种:

1) <位宽><进制><数字>

这是一种全面的描述方式。

2) <进制><数字>

数字的位宽采用缺省位宽(这由具体的机器系统决定, 但至少32位)。

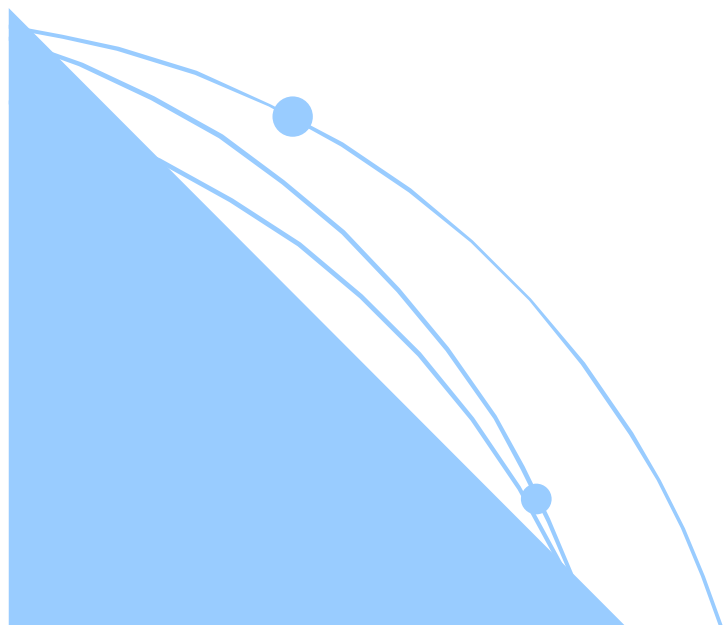
3) <数字>

采用缺省进制(十进制), 位宽采用缺省位宽。

在表达式中，位宽指明了数字的精确位数。见下例：

`8'b10101100` //位宽为8的数的二进制表示，'b表示二进制

`8'ha2` //位宽为8的数的十六进制表示，'h表示十六进制



(2)x和z值

x代表不定值，**z**代表高阻值。一个**x**可以用来定义十六进制数的4位二进制数的状态，八进制数的3位，二进制数的1位。**z**的表示方式同**x**类似。

z的另一种表达方式是“?”。在使用**case**表达式时建议使用这种写法，以提高程序的可读性，见下例：

4'b10x0 //位宽为4的二进制数，从低位数起第2位为不定值

4'b101z //位宽为4的二进制数，从低位数起第1位为高阻值

12'dz //位宽为12的十进制数，其值为高阻值(第1种表达方式)

12'd? //位宽为12的十进制数，其值为高阻值(第2种表达方式)

8'h4x //位宽为8的十六进制数，其低4位值为不定值

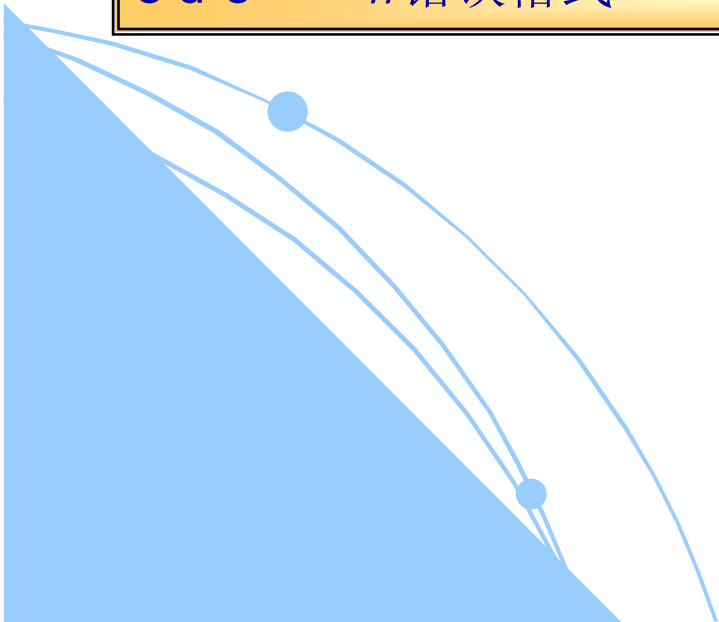
(3)负数

在位宽表达式前加负号来表示负数。负号必须写在数字定义表达式的最前面。

注意，负号不可以放在位宽和进制之间，也不可以放在进制和具体的数之间。见下例：

`-8'd5` //这个表达式代表5的补数(用八位二进制数表示)

`8'd-5` //错误格式

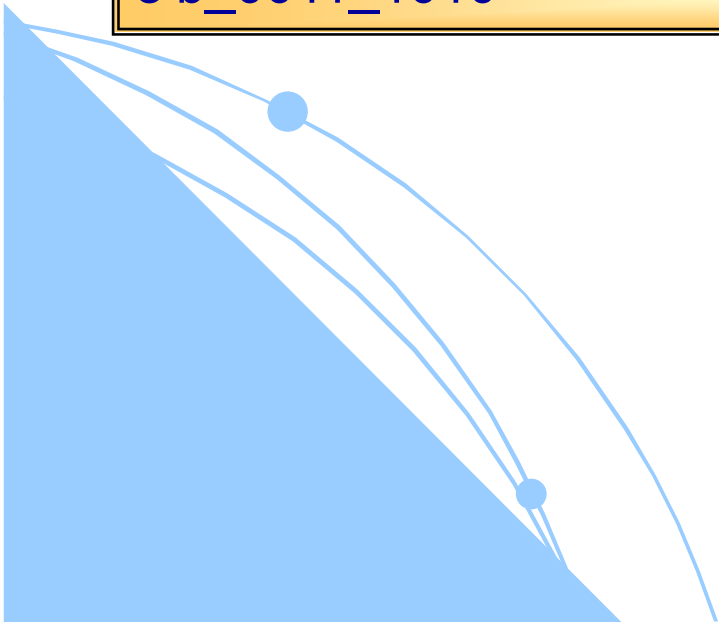


(4)下画线(underscore)

下画线可以用来分隔数的表达以提高程序可读性，但不可以用在位宽和进制处，只能用在具体的数字之间。见下例：

16'b1010_1011_1111_1010 //正确格式

8'b_0011_1010 //错误格式



当常量不说明位数时，默认值是32位，每个字母用8位的ASCII值表示。
例：

`10=32'd10=32'b1010`

`1=32'd1=32'b1`

`-1=-32'd1=32'hFFFFFFFF`

`'BX=32'BX=32'BXXXXXXXX...X`

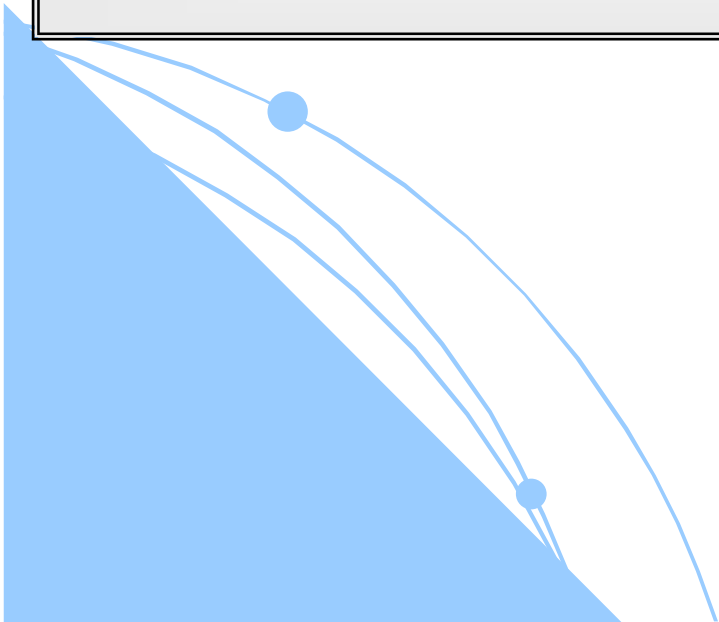
`"AB"=16'B01000001_01000010 //字符串AB为十六进制数16'h4142`

2. 参数(parameter)型

概念：在Verilog HDL中用parameter来定义常量，即用parameter来定义一个标识符代表一个常量，称为**符号常量**，即标识符形式的常量。

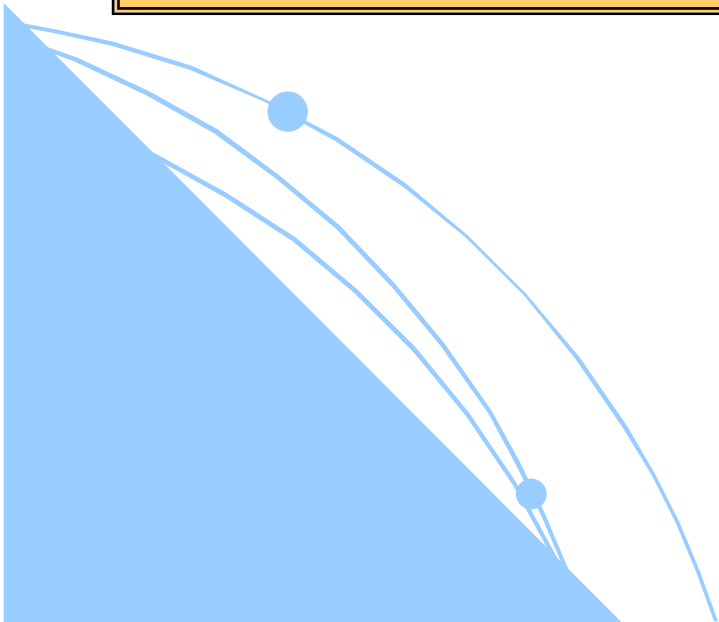
parameter型数据说明格式如下：

parameter 参数名1=表达式,, 参数名n=表达式;



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

```
parameter msb=7;           //定义参数msb为常量7
parameter e=25, f=29;       //定义两个常数参数
parameter r=5.7;           //声明r为一个实型参数
parameter byte_size=8, byte_msb=byte_size-1; //用常数表达式
赋值
parameter average_delay=(r+f) / 2; //用常数表达式赋值
```



参数型常数的作用：

参数型常数经常用于定义延迟时间和变量宽度。在模块或实例引用时，可通过参数传递改变被引用模块或实例中已定义的参数。下面将通过两个例子说明在层次调用的电路中改变参数的用法。

【例3.1】

```
module Decode(A,F);  
    parameter Width=1,Polarity=1;  
    ...  
endmodule
```

```
module Top(...);  
    wire [3:0] A4;  
    wire [4:0] A5;  
    wire [15:0] F16;  
    wire [31:0] F32;  
    Decode #(4,0) D1(A4,F16);  
    Decode #(5)   D2(A5,F32);  
endmodule
```

//模块Decode定义时用了两个参数型变量：width和polarity，且都为1.在Top模块中引用decode实例时，可通过参数的传递来改变定义时已经规定的参数值，即通过#（4,0），实例D1实际引用的是参数width和polarity分别为4和0时的decode模块；#(5)实际引用的是参数width为5，polarity仍为1时的decode模块

【例3.2】

下面是一个由多层次模块构成的电路。在一个模块中改变另一个模块的参数时，使用**defparam**命令。

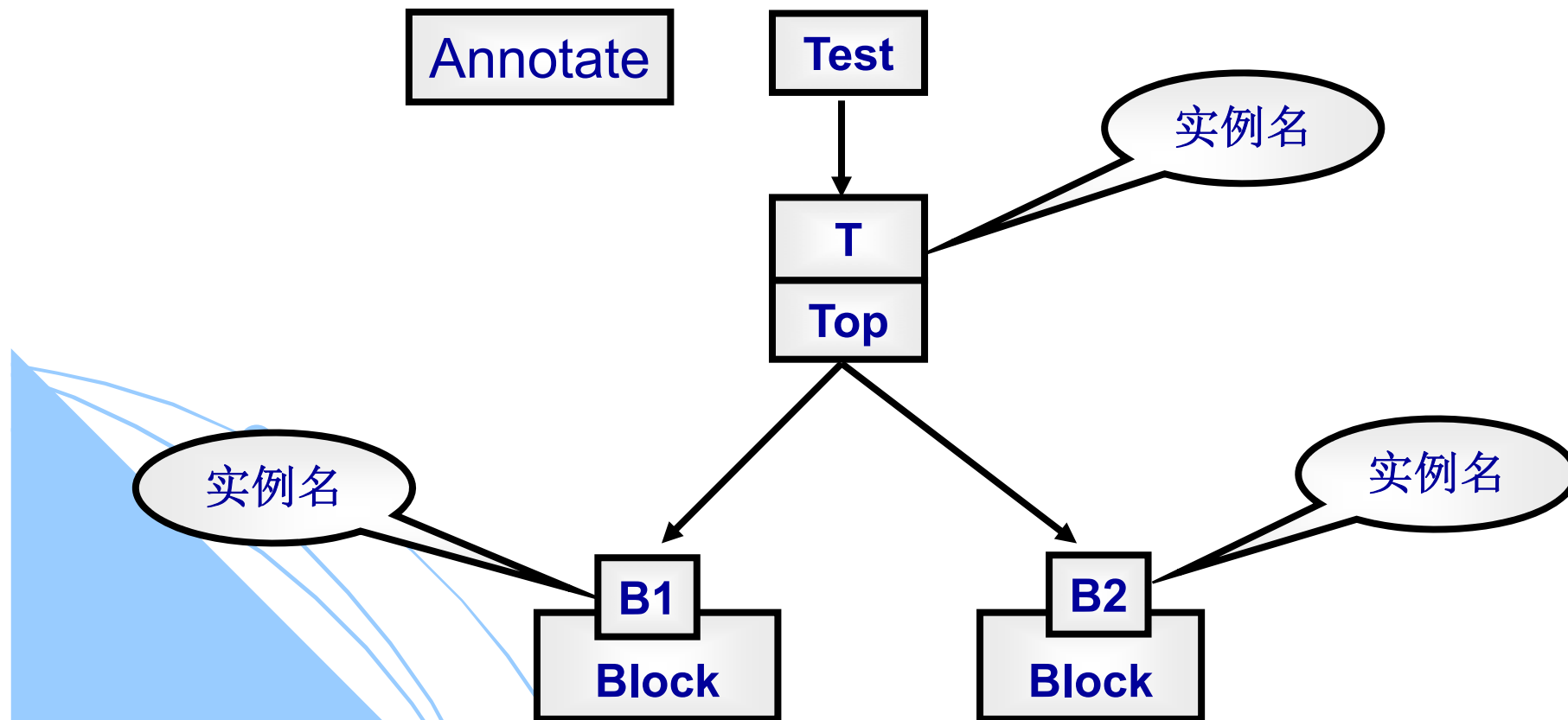


图 3.2 多层次模块构成的电路

```
`include "Top.v"
`include "Block.v"
`include "Annotate.v"
module Test;
    wire W;
    Top T();
endmodule
```

```
module Top;
    wire w;
    Block B1();
    Block B2();
endmodule
```

```
module Block;
    parameter P=0;
endmodule
```

```
module Annotate;
```

```
    defparam;
```

Test.T.B1.P=2; //多层次模块中，参数变量的命名规则，
用点号来表示不同的模块层次

```
    Test.T.B2.P=3;
endmodule
```

3.2.2 变 量

变量即在程序运行过程中其值可以改变的量，在Verilog HDL中变量的数据类型有很多种，这里只对常用的几种进行介绍。多驱动源情况略。

1. Wire型

说明：

- (1) **wire**网络数据类型表示结构实体(例如门)之间的物理连接。
- (2) 网络类型的变量不能储存值，而且它必须受到驱动器(例如门或连续赋值语句**assign**)的驱动。如果没有驱动器连接到网络类型的变量上，则该变量就是高阻的，即其值为z。
- (3) **wire**型数据常用来表示以**assign**关键字指定的组合逻辑信号。
- (4) **Verilog**程序模块中输出信号类型默认时自动定义为**wire**型。**wire**型信号可以用做任何方程式的输入，也可以用做“**assign**”语句或实例元件的输出。

wire型信号的格式如下：

wire [n-1: 0] 数据名1, 数据名2,数据名i;

或 **wire [n: 1] 数据名1, 数据名2,数据名i;**

说明：

wire是wire型数据的确认符；[n-1: 0]和[n: 1]代表该数据的位宽，即该数据有几位；最后是数据的名字。如果一次定义多个数据，数据名之间用逗号隔开。声明语句的最后要用分号表示语句结束。

看下面的几个例子：

wire a; //定义了一个1位的wire型数据，

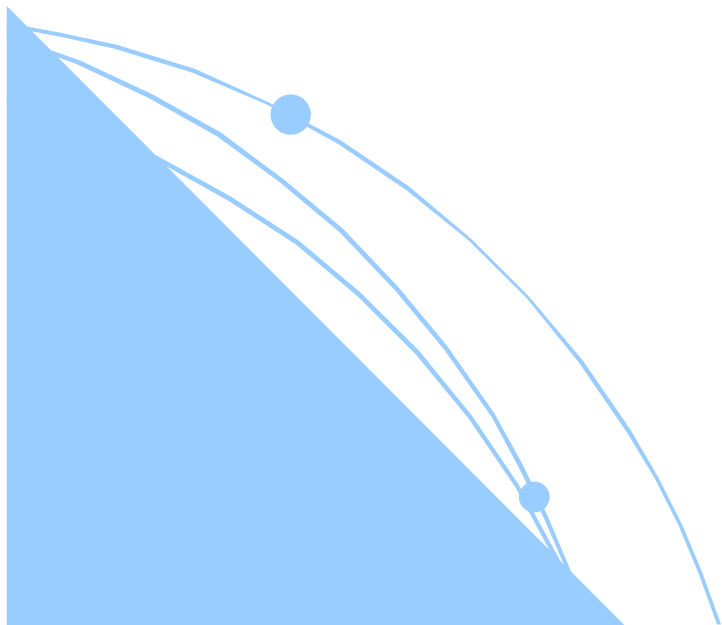
wire [7: 0] b; //定义了一个8位的wire型数据b

wire [4: 1] c, d; //定义了二个4位的wire型数据c和d

2. reg型

说明:

- (1) reg型是寄存器数据类型。
- (2) reg类型数据的默认初始值为不定值x。
- (3) reg型数据常用来表示“always”模块内的指定信号，常代表触发器。
- (4) 在“always”块内被赋值的每一个信号都必须定义成reg型。



reg型数据的格式如下:

或

```
reg [n-1: 0] 数据名1, 数据名2, .....数据名i;  
reg [n: 1]   数据名1, 数据名2, .....数据名i;
```

说明:

reg是reg型数据的确认标识符; [n-1: 0]和[n: 1]代表该数据的位宽, 即该数据有几位(bit); 最后是数据的名称。如果一次定义多个数据, 数据名之间用逗号隔开。声明语句的最后要用分号表示语句结束。

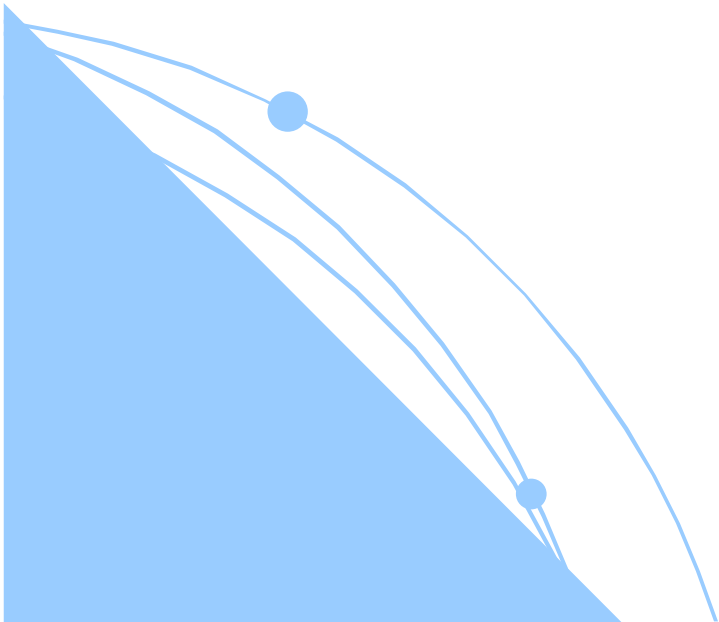
看下面的几个例子:

```
reg rega;    //定义了一个1位的名为rega的reg型数据  
reg [3: 0] regb;    //定义了一个4位的名为regb的reg型数据  
reg [4: 1] regc, regd    //定义了二个4位的名为regc和regd的reg型数据
```


重要说明:

reg型信号并不一定是寄存器或触发器的输出，它只表示被定义的信号将用在“**always**”块内。

而当**reg**型信号确实是寄存器或触发器的输出时，只有**always**块或**initial**块可以操作该类型信号。



3. memory型

说明:

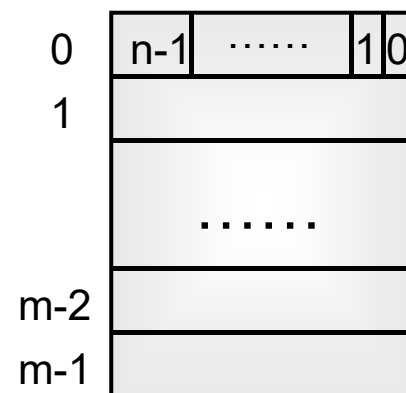
- (1) Memory型即存储器类型;
- (2) Verilog HDL通过对reg型变量建立数组来对存储器建模, 可以描述RAM型存储器, ROM存储器和reg文件。
- (3) 数组中的每一个单元通过一个数组索引进行寻址。
- (4) 在Verilog语言中没有多维数组存在。

memory型数据的格式如下:

reg [n-1: 0] 存储器名 [m-1: 0]

或

reg [n-1: 0] 存储器名 [m: 1];



说明:

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

下面举例说明：

```
reg [7: 0] mema [255: 0];
```

说明：

这个例子定义了一个名为mema的存储器，该存储器有256个8位的存储器。该存储器的地址范围是0到255。

注意：对存储器进行地址索引的表达式必须是常数表达式。

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

```
parameter wordsize=16, //定义两个参数  
        memsize=256;  
reg [wordsize-1: 0] mem[memsize-1: 0], writereg, readreg;
```

说明:

尽管memory型数据和reg型数据的定义格式很相似,但要注意其不同之处。如一个由n个1位寄存器构成的存储器组是不同于一个n位的寄存器的。见下例:

reg [n-1: 0] rega; //一个n位的寄存器

reg mema [n-1: 0]; //一个由n个1位寄存器构成的存储器组

一个n位的寄存器可以在一条赋值语句里进行赋值,而一个完整的存储器则不行。见下例:

rega=0; //正确赋值语句

mema=0; //错误的赋值语句

如果想对memory中的存储单元进行读写操作,必须指定该单元在存储器中的地址。下面的写法是正确的:

mema[3]=0; //给memory中的第3个存储单元赋值为0

3.3 运算符及表达式

按功能分类可分为

算术运算符(+, -, X, / , %)

赋值运算符(=, <=)

关系运算符(>, <, >=, <=)

逻辑运算符(&&, ||, !)

条件运算符(?:)

位运算符(~, |, ^, &, ^~)

移位运算符(<<, >>)

拼接运算符({ })

按所带操作数
个数可分为

{ 单目运算符
双目运算符
三目运算符

例：

`clock = ~clock;` // ~ 是一个单目取反运算符，`clock` 是操作数

`c = a | b;` // 是一个二目按位或运算符，`a` 和 `b` 是操作数

`r = s ? t : u;` // ?: 是一个三目条件运算符，`s`，`t`，`u` 是操作数

3.3.1 基本的算术运算符

在Verilog HDL语言中，算术运算符又称为二进制运算符，共有下面几种：

- (1) + (加法运算符，或正值运算符，如`rega+regb`，`+3`);
- (2) - (减法运算符，或负值运算符，如`rega-3`，`-3`);
- (3) * (乘法运算符，如`rega*3`);
- (4) / (除法运算符，如`5 / 3`);
- (5) % (模运算符，或称为求余运算符，要求%两侧均为整型数据。如`7 % 3`的值为1)。

在进行整数除法运算时，结果值要略去小数部分，只取整数部分；而进行取模运算时，结果值的符号位采用模运算式里第一个操作数的符号位。见下例：

模运算表达式	结果	说明
$10\%3$	1	余数为1
$11\%3$	2	余数为2
$12\%3$	0	余数为0，即无余数
$-10\%3$	-1	结果取第一个操作数的符号位，所以余数为-1
$11\%-3$	2	结果取第一个操作数的符号位，所以余数为2

注意：在进行算术运算操作时，如果某一个操作数有不确定的值**x**，则整个结果也为不定值**x**。

3.3.2 位运算符

Verilog HDL作为一种硬件描述语言，是针对硬件电路而言的。在硬件电路中信号有4种状态值，即 **1**，**0**，**x**，**z**。

Verilog HDL提供了以下5种位运算符：

- 1) \sim //取反
- 2) $\&$ //按位与
- 3) $|$ //按位或
- 4) \wedge //按位异或
- 5) $\wedge\sim$ //按位同或(异或非)

- 写出下列真值表

\sim	结果
1	
0	
x	

	0	1	x
0			
1			
x			

&	0	1	x
0			
1			
x			

\wedge	0	1	x
0			
1			
x			

$\wedge \sim$	0	1	x
0			
1			
x			

(1)“取反”运算符 \sim ： \sim 是一个单目运算符，用来对一个操作数进行按位取反运算。

表3.2 (a) 取反运算符的运算规则

\sim	结果
1	0
0	1
X	X

例如：

`rega='b1010; //rega的初值为' b1010`

`rega=~rega; //rega的值进行取反运算后变为' b0101`

(2)“按位与”运算符&：按位与运算就是将两个操作数的相应位进行与运算。

表3.2 (b) “按位与”运算规则

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

(3)“按位或”运算符|：按位或运算就是将两个操作数的相应位进行或运算。

表3.2 (c) “按位或”运算规则

	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

(4)“按位异或”运算符^(也称之为XOR运算符): 按位异或运算就是将两个操作数的相应位进行异或运算。

表3.2 (d) “按位异或”运算规则

\wedge	0	1	X
0	0	1	X
1	1	0	X
X	X	X	X

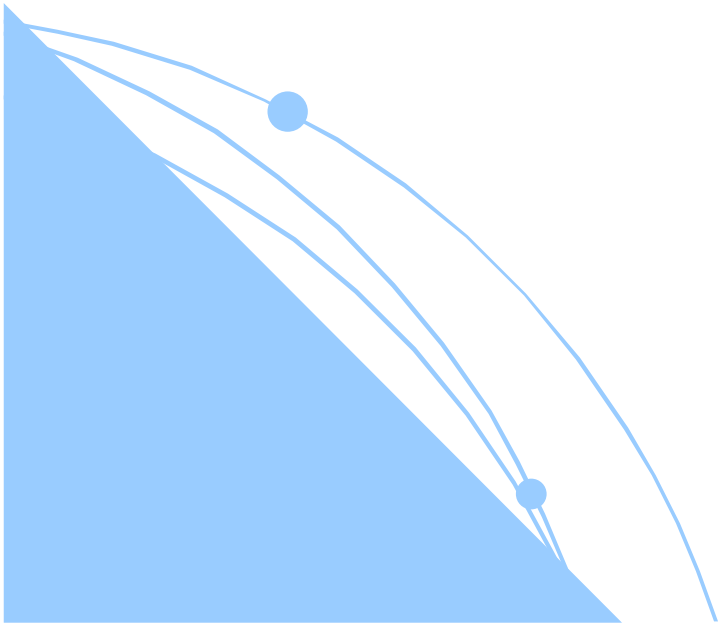
(5)“按位同或” 运算符 $\wedge\sim$ ：按位同或运算就是将两个操作数的相应位先进行异或运算再进行非运算。

表3.2 (e) “按位同或” 运算规则

$\wedge\sim$	0	1	X
0	1	0	X
1	0	1	X
X	X	X	X

注意：

不同长度的数据进行位运算：两个长度不同的数据进行位运算，系统会自动的将两者按**右端对齐**。位数少的操作数会在相应的高位用**0**填满，以使两个操作数按位进行操作。



3.4 小结

(1)在Verilog模块中所有过程块(如：**initial**块、**always**块)、连续赋值语句、实例引用都是并行的；

(2)实例引用表示的是一种通过变量名互相连接的关系；

(3)在同一模块中各个过程块、各条连续赋值语句和各条实例引用语句这三者出现的先后顺序没有关系；

(4)只有连续赋值语句(即用关键词**assign**引出的语句)和实例引用语句(即用已定义的模块名引出的语句)，可以独立于过程块而存在于模块的功能定义部分。

(5)被实例引用的模块，其端口可以通过不同名的连线或寄存器类型变量连接到别的模块相应的输出、输入信号端；

(6)在“**always**”块内被赋值的每一个信号都必须定义成**reg**型。

思考题

- 1、模块由几个部分组成？
- 2、端口分为几种？
- 3、为什么端口要说明信号的位宽？
- 4、能否说模块相当于电路图中的功能模块，端口相当于功能模块的引脚？
- 5、模块中的功能描述可以由哪几类语句或语句块组成？它们出现的顺序会不会影响功能的描述？
- 6、这几类描述中哪一种直接与电路结构有关？
- 7、最基本的Verilog变量有几种类型？
- 8、reg型和wire型变量的差别是什么？
- 9、由连续赋值语句（assign）赋值的变量能否是reg类型的？
- 10、在always块中被赋值的变量能否是wire类型的？如果不能是wire类型，那么必须是什么类型的？它们表示的一定是实际的寄存器吗？
- 11、参数类型的变量有什么用处？
- 12、Verilog语法规定的参数传递和重新定义功能有什么直接的应用价值？
- 13、逻辑比较运算符小于等于“<=”和非阻塞赋值“<=”的表示是完全一样的，为什么Verilog在语句解释和编译时不会搞错？
- 14、是否可以说实例引用的描述实际上就是严格意义上的电路结构描述？

第四章 运算符、赋值语句和结构说明语句

- 4.1 逻辑运算符
- 4.2 关系运算符
- 4.3 等式运算符
- 4.4 移位运算符
- 4.5 位拼接运算符
- 4.6 缩减运算符
- 4.7 优先级别
- 4.8 关键词
- 4.9 赋值语句和块语句
- 小结
- 思考题 ●

4.1 逻辑运算符

在Verilog HDL语言中存在3种逻辑运算符：

(1) &&逻辑与；

(2) ||逻辑或；

(3) !逻辑非。

表4.1 逻辑运算真值表

a	b	!a	!b	a&&b	a b
真	真	假	假	真	真
真	假	假	真	假	真
假	真	真	假	假	真
假	假	真	真	假	假

逻辑运算符中“&&”和“||”的优先级别低于关系运算符，“!”高于算术运算符。见下例：

(a>b)&&(x>y) 可写成: **a>b&& x>y**

(a==b)|| (x==y) 可写成: **a==b|| x==y**

(!a)|| (a>b) 可写成: **!a || a>b**

为了提高程序的可读性，明确表达各运算符间的优先关系，建议使用括号。

4.2 关系运算符

关系运算符共有以下4种：

(1) $a < b$ a 小于 b

(2) $a > b$ a 大于 b

(3) $a \leq b$ a 小于或等于 b

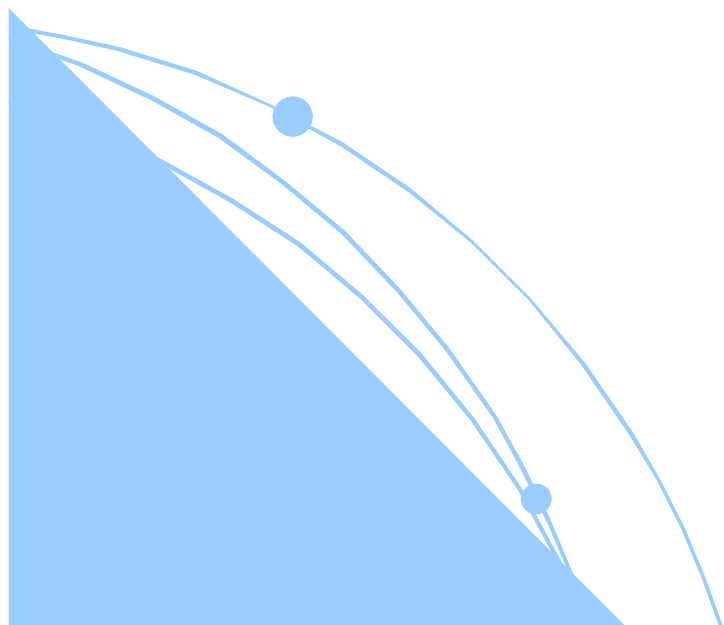
(4) $a \geq b$ a 大于或等于 b

说明：关系为假(false)，则返回值是0；关系为真(true)，则返回值是1；如果某个操作数的值不定，则关系是模糊的，返回值是不定值x。

所有的关系运算符有着相同的优先级别。关系运算符的优先级别低于算术运算符的优先级别。见下例：

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

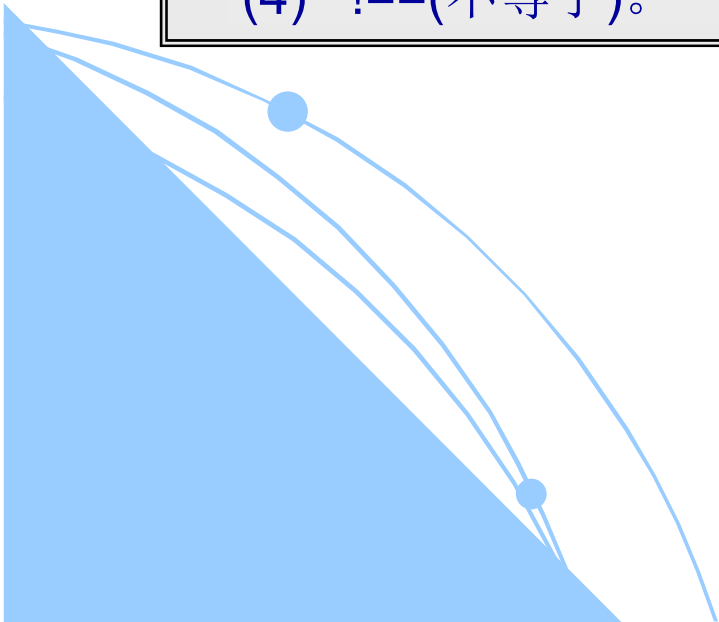
$\text{size} - (1 < a)$ 不同于 **$\text{size} - 1 < a$**



4.3 等式运算符

在Verilog HDL语言中存在4种等式运算符；

- (1) ==(等于)；
- (2) !=(不等于)；
- (3) ===(等于)；
- (4) !==(不等于)。



“==”和 “!=”又称为逻辑等式运算符，其结果由两个操作数的值决定。由于操作数中某些位可能是不定值x和高阻值z，结果可能为不定值x。

“===”和 “!==”运算符在对操作数进行比较时对不定值x和高阻值z也进行比较，两个操作数必须完全一致，其结果才是1，否则为0。“===”和 “!==”运算符常用于case表达式的判别，所以又称为“case等式运算符”。

这4个等式运算符的优先级别相同。
例如：

if(A == 1'bx) \$display("A is X"); (当A等于x时，这个语句不执行)
if(A === 1'bx) \$display("A is X"); (当A等于X时，这个语句执行)

表4.2 等式运算符的真值表

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

4.4 移位运算符

在Verilog HDL中有两种移位运算符：“<<”(左移位运算符)和“>>”(右移位运算符)。其使用方法如下：

a>>n 或 a<<n

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

```
module shift;  
  reg [3: 0] start, result;  
  initial  
  begin  
    start = 1;    //start在初始时刻设为值0001  
    result=(start<<2);  //移位后，start的值0100，然后赋给result  
  end  
endmodule
```

从上面的例子可以看出，**start**在移过两位以后，用0来填补空出的位。

进行移位运算时应注意移位前后变量的位数，下面将给出一例：

$4'b1001 \ll 1 = 5'b10010;$

$4'b1001 \ll 2 = 6'b100100;$

$1 \ll 6 = 32'b1000000;$

$4'b1001 \gg 1 = 4'b0100;$

$4'b1001 \gg 4 = 4'b0000;$

4.5 位拼接运算符

在Verilog HDL语言中有一个特殊的运算符：{ }

即：位拼接运算符(**concatenation**)

用这个运算符可以把两个或多个信号的某些位拼接起来进行运算操作。其使用方法如下：

{信号1的某几位， 信号2的某几位，信号n的某几位}

即把某些信号的某些位详细地列出来，中间用逗号分开，最后用大括号括起来表示一个整体信号。见下例：

{a, b[3: 0], w, 3'b101} 也可以写成为：

{a, b[3], b[2], b[1], b[0], w, 1'b1, 1'b0, 1'b1}

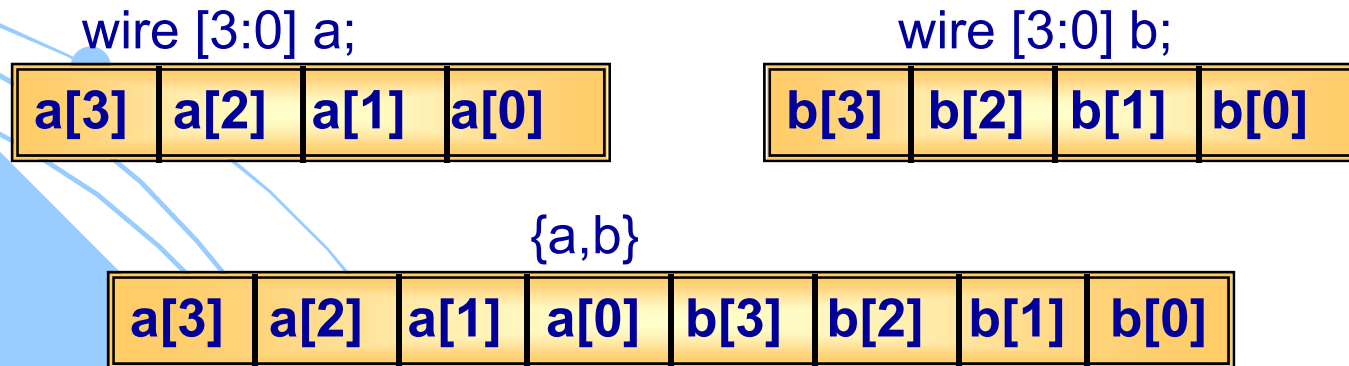
位拼接还可以用重复法来简化表达式。见下例：

{ 4 { w } } 等同于 **{w, w, w, w}**

位拼接还可以用嵌套的方式来表达。见下例：

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

注意：在位拼接表达式中不允许存在没有指明位数的信号。因为在计算拼接信号位宽的大小时必须知道其中每个信号的位宽。



a和b可表达的十进制数范围均为0~15，而{a,b}可表示0~255范围的十进制数

4.6 缩减运算符

缩减运算符(reduction operator)是单目运算符，也有与、或、异或、同或等运算。运算规则类似于位运算符的运算规则，但其运算过程不同。位运算是操作数的相应位进行与、或、非运算，操作数是几位数则运算结果也是几位数。而缩减运算是操作数进行与、或、异或、同或等的递推运算，最后的运算结果是1位的二进制数。

缩减运算的具体运算过程是这样的：

第一步先将操作数的第1位与第2位进行与、或等运算；

第二步将运算结果与第3位进行与、或等运算，依次类推，直至最后1位。

例如：reg [3:0] B;


reg C;

C=&B;

相当于：

C=((B[0] & B[1]) & B[2]) & B[3];

4.7 优先级别

优先级别	
<p>! ~</p> <p>* / %</p> <p>+ -</p> <p><< >></p> <p>< <= > >=</p> <p>== != === !==</p> <p>&</p> <p>^ ^~</p> <p> </p> <p>&&</p> <p> </p> <p>?:</p>	<p>高优先级</p>  <p>低优先级</p>

4.8 关键词

在Verilog HDL中，所有的关键词是事先定义好的确认符，用来组织语言结构。关键词是用小写字母定义的，因此在编写源程序时要注意关键词的书写，以避免出错。

always, and, assign, begin, buf, bufif0, 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, Initial, inout, input, integer, join, large, macromodule, medium, module, nand, negedge, nmos, nor, not, notif0, nodf1, or, output, parameter, pmos, posedge, primitive, pull0, pull1, pullup, pulldown, rcmos, reg, releases, repeat, mmos, rpmos, rtran, rtranif0, rtranif1, scalared, small, specify, specparam, strength, strong0, strong1, supply0, supply1, table, task, time, tran, tranif0, tranif1, tri, tri0, trl1, triand, trior, trireg, vectored, wait, wand, weak0, weak1, while, wire, wor, xnor, xor

4.9 赋值语句和块语句

4.9.1 赋值语句

在Verilog HDL语言中，信号有两种赋值方式：

(1)非阻塞(**non_blocking**)赋值方式(如 $b \leq a$;):

- 在语句块中，上面语句所赋的变量值不能立即就为下面的语句所用；
- 块结束后才完成赋值操作,块结束前被赋值的变量保持上一次所赋的值；
- 在编写可综合的时序逻辑模块时，这是最常用的赋值方法。意即，在`always`块中经常使用。

(2)阻塞(**blocking**)赋值方式(如 $b = a$;):

- 赋值语句执行完后，块才结束；
- b 的值在赋值语句执行完后立刻就改变；
- 在时序逻辑中使用时（在沿触发的`always`块中使用时），综合后可能会产生意想不到的结果。

前面所举的例子中的“always”模块内的reg型信号都是采用下面的非阻塞赋值方式：

$b \leq a;$

这种方式的赋值不是立刻执行的，也就是说“always”块内的下一条语句执行后，b并不等于a，而是保持原来的值，“always”块结束后，才进行赋值。

而阻塞赋值方式，如：

$b = a;$

这种赋值方式是立刻执行的，也就是说执行下一条语句时，b已等于a。

对这两种赋值方式，使用不正确则会得到错误的电路结构。下面举例说明：

【例4.1】

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

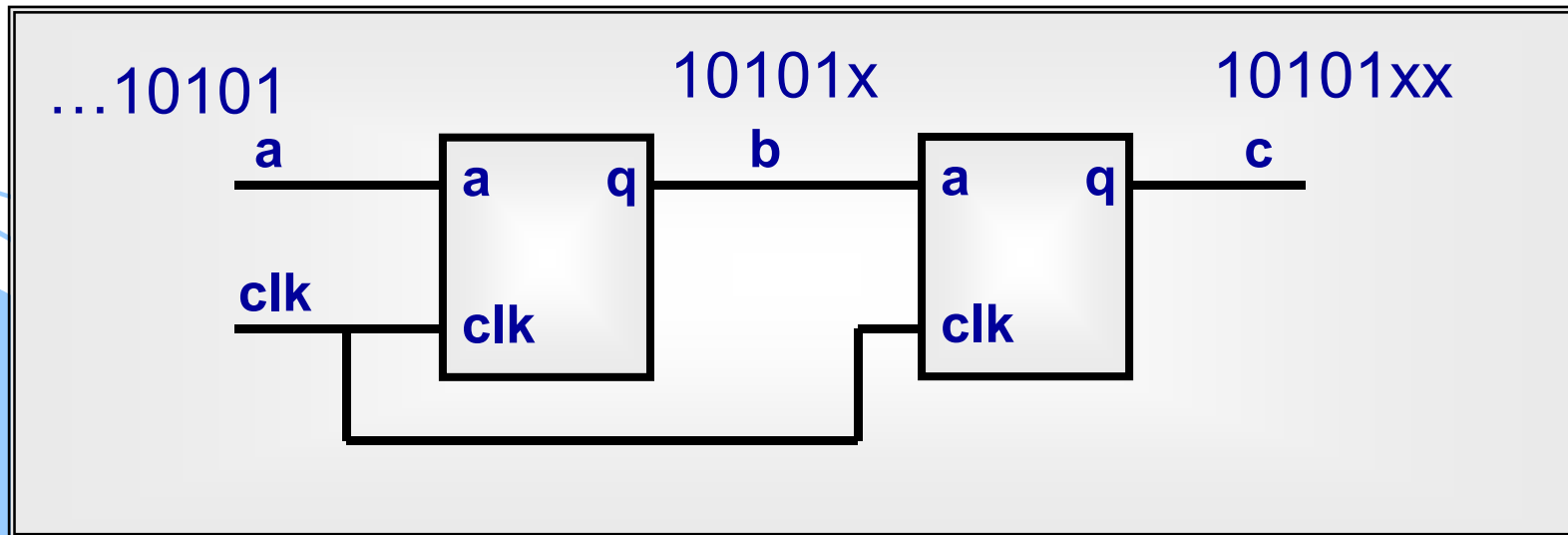
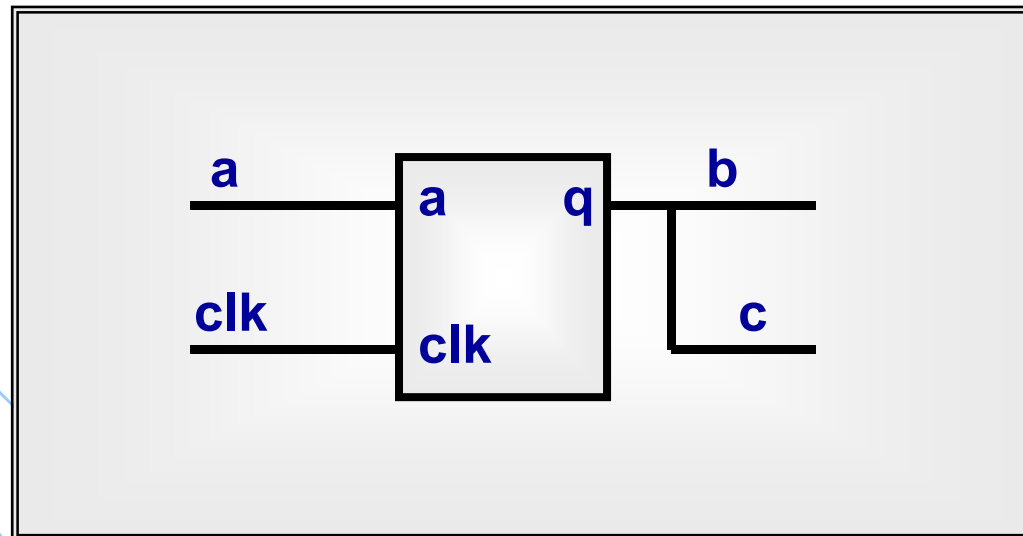


图 4.1 非阻塞赋值方式的“always”电路图

【例4.2】

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



错误设计！

图 4.2 阻塞赋值方式的“always”电路图

4.9.2 块语句

块语句通常用来将两条或多条语句组合在一起，使其在格式上看更像一条语句。块语句有两种：

- 块语句 {
- begin_end** 用来标识顺序执行的语句，用它来标识的块称为**顺序块**；
 - fork_join** 用来标识并行执行的语句，用它来标识的块称为**并行块**。

1. 顺序块

顺序块的特点：

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

顺序块的格式如下：

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

或

```
begin: 块名  
  块内声明语句  
  语句1:  
  语句2;  
  .....  
  语句n;  
end
```

说明：

- 块名即该块的名字，一个标识名，其作用后面再详细介绍。
- 块内声明语句可以是参数声明语句、**reg**型变量声明语句、**integer**型变量声明语句和**real**型变量声明语句。

【例4.3】

```
begin
    areg=breg;
    creg=areg;    //creg的值为breg的值
end
```

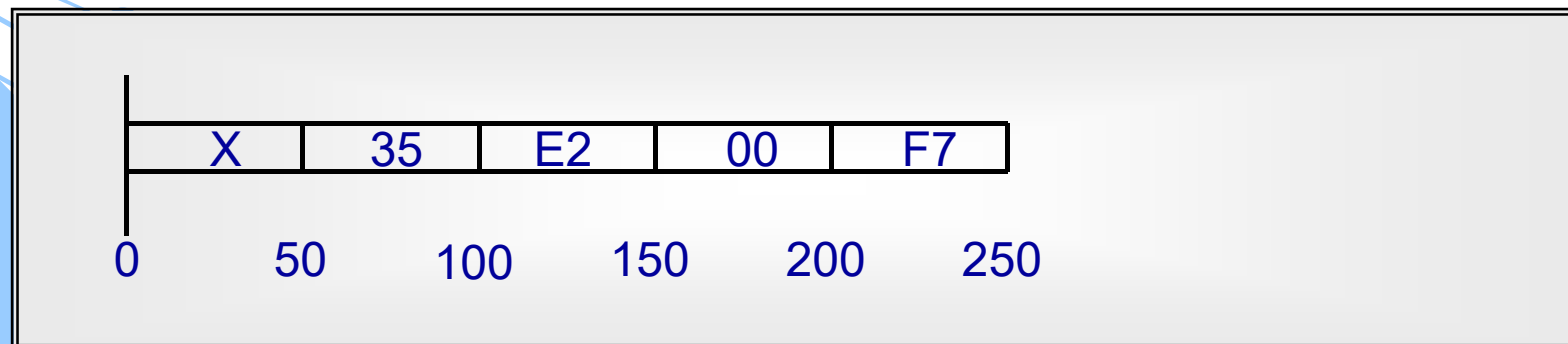
【例4.3】

```
begin
    areg=breg;
    #10 creg=areg;
    //在两条赋值语句间延迟10个时间单位
end
```



【例4.5】

```
parameter d=50;    //声明d是一个参数
reg [ 7: 0] r;      //声明r是一个8位的寄存器变量
begin              //由一系列延迟产生的波形
    #d r=' h35;
    #d r=' hE2;
    #d r=' h00;
    #d r=' hF7;
    #d ->end_wave;  //触发事件end__wave
end
```



2. 并行块

并行块的特点：

- (1)块内语句是同时执行的，即程序流程控制一进入到该并行块，块内语句则开始同时并行地执行。
- (2)块内每条语句的延迟时间是相对于程序流程控制进入到块内的仿真时间。
- (3)延迟时间是用来给赋值语句提供执行时序的。
- (4)当按时间时序排序在最后的语句执行完后或一个**disable**语句执行时，程序流程控制跳出该程序块。

并行块的格式如下：

fork

语句1;

语句2;

.....

语句n,

join

或

fork: 块名

块内声明语句

语句1;

语句2;

.....

语句n;

join

说明：

- 块名即标识该块的一个名字，相当于一个标识符。
- 块内说明语句可以是参数说明语句、**reg**型变量声明语句、**integer**型变量声明语句、**real**型变量声明语句、**time**型变量声明语句和事件(**event**)说明语句。

【例4.6】

fork

#50 r=' h35;

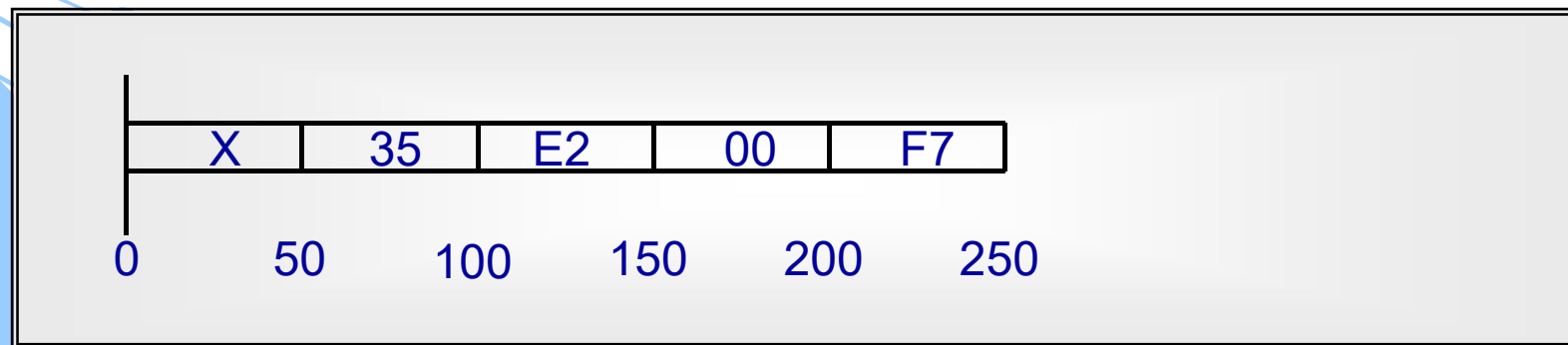
#100 r=' hE2;

#150 r=' h00;

#200 r=' hF7;

#250 —> end_wave; //触发事件end_wave

join



3. 块 名

在Verilog HDL语言中，可以给每个块取个名字，只需将名字加在关键词begin或fork后面即可。这样做的原因有以下几点：

1)可以在块内定义局部变量，即只在块内使用的变量；

2)可以允许块被其他语句调用，如**disable**语句；

3) 在**Verilog**语言里，所有的变量都是静态的，即所有的变量都只有一个唯一的存储地址，因此进入或跳出块并不影响存储在变量内的值。因此，块名就提供了一个在任何仿真时刻确认变量值的方法。

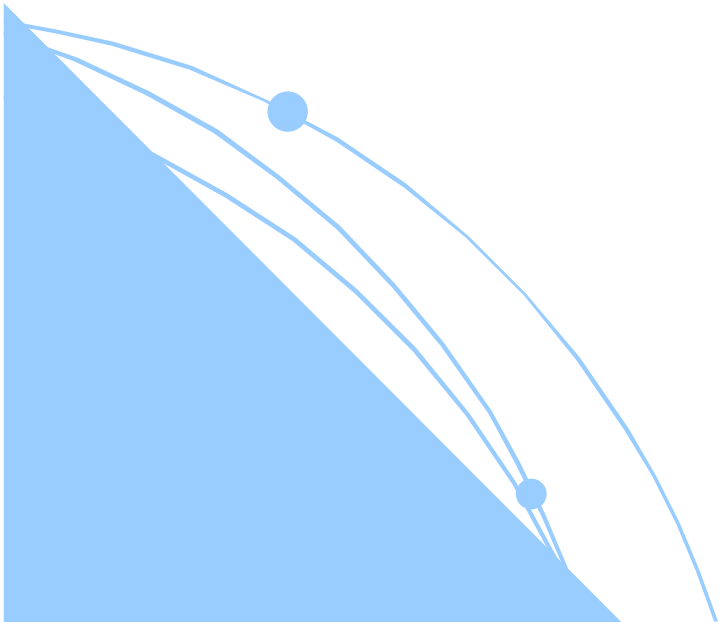
4. 起始时间和结束时间

在并行块和顺序块中都有一个起始时间和结束时间的概念。

对于顺序块，起始时间就是第一条语句开始被执行的时间，结束时间就是最后一条语句执行完的时间；

对于并行块，起始时间对于块内所有的语句是相同的，即程序流程控制进入该块的时间，其结束时间是按时间排序在最后的语句执行结束的时间。

当一个块嵌入另一个块时，块的起始时间和结束时间是很重要的。跟在块后面的语句在该块的结束时间到了才能开始执行，也就是说，只有该块完全执行完后，后面的语句才可以执行。



在fork_join块内，各条语句不必按顺序给出，因此在并行块里，各条语句在前还是在后是无关紧要的。

【例4.7】

```
fork
    #250  —>end_wave;
    #200  r=' hF7;
    #150  r=' h00;
    #100  r=' hE2;
    #50   r=' h35;
join
```

在这个例子中，各条语句并不是按被执行的先后顺序给出的，但同样可以生成前面例子中的波形。

4.10 小结

(1)无论是逻辑运算、逻辑比较还是逻辑等式等逻辑操作一般发生在条件判断语句中，其输出只有1或0，也可以理解为成立（真）或不成立（假）。

(2)位拼接运算符{ }在C语言中没有定义，但在Verilog中是一种很有用的语法。它可用一个信号名来表示由多位信号组成的复杂信号，其中每个功能信号可以有自己独立的名字和位宽。例如控制信号，可以用如下的位拼接来表示：

```
assign control={read, write, sel[2: 0], halt, load_instr, ...};
```

这样可以大大提高程序的可读性和可维护性。

(3)缩减运算符(reduction operator)也是C语言所没有的，合理地使用缩减运算符可以使程序简洁、明了。

(4)阻塞和非阻塞赋值也是C语言所没有的。我们应当理解这是非常重要的概念，特别在编写可综合风格的模块中要加以注意。在硬件实现时这两者有很大的不同。

(5)begin end块语句与C语言中的大括号对(即{ })类似，而fork join语句在C语言中没有定义，但其语义并不难理解。在测试模块中，描述测试信号常在initial和always过程块中使用并行块。这种描述方法，由于时间关系只与起点比较，有时这样表达比较容易和清楚。

思考题

- 1、逻辑运算符与按位逻辑运算符有什么不同，它们各在什么场合使用？
- 2、指出两种逻辑等式运算符的不同点，解释书上的真值表。
- 3、拼接符的作用是什么？为什么说合理地使用拼接符可以提高程序的可读性和可维护性？拼接符表示的操作其物理意义是什么？
- 4、如果都不带时间延迟，阻塞和非阻塞赋值有什么不同？举例说明它们的不同点。
- 5、举例说明顺序块和并行块的不同？
- 6、如果在顺序块中，前面有一条语句是无限循环，下面的语句能否进行？
- 7、如果在并行块中，发生上述情况，会如何呢？

第五章 条件语句、循环语句、块语句与生成语句

5.1 条件语句（if_else语句）

5.2 case语句

5.3 条件语句的语法

5.4 多路分支语句

5.5 循环语句

5.6 顺序块和并行块

5.7 生成块

5.8 举例

小结

思考题

5.1 条件语句（if_else语句）

if语句是用来判定所给定的条件是否满足，根据判定的结果（真或假）决定执行给出的两种操作之一。Verilog HDL提供了3种形式得if语句。

（1）if（表达式）语句

例如：

```
if(a>b)
    out1=int1;
```

注意：if语句在生成电路时具有优先级！因此，类似优先级编码器的电路可用if语句描述。

(2) **if** (表达式)

语句1

else

语句2

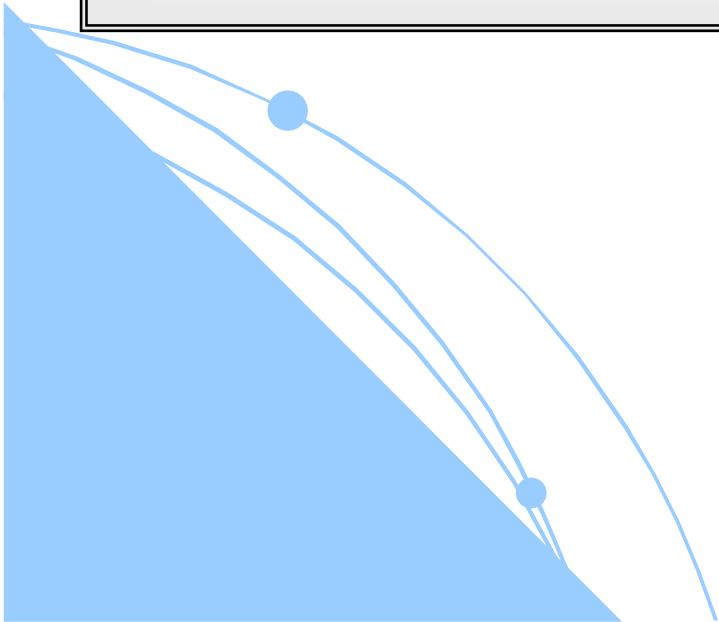
例如:

```
if(a>b)
```

```
    out1=int1;
```

```
else
```

```
    out1=int2;
```



```
(3) if (表达式)      语句1;  
    else if (表达式2)  语句2;  
    else if (表达式3)  语句3;  
    .....  
    else if (表达式m)  语句m;  
    else               语句n;
```

例如:

```
if(a>b)  out1=int1;  
else if(a==b) out1=int2;  
else      out1=int3;
```

注意：条件语句必须在过程块语句中使用。所谓过程块语句是指由**initial**和**always**语句引导的执行语句集合。除了这两种块语句引导的**begin end**块中可以编写条件语句外，模块中的其他地方都不能编写。

例如： **always @(some_event)**
 begin
 if(a>b) **out1=int1;**
 else if(a==b) **out1=int2;**
 else **out1=int3;**
 end

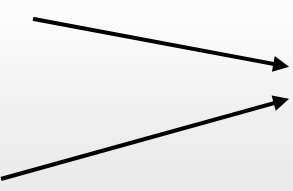
说明:

(1) 3种形式的if语句在if后面都有“表达式”，一般为逻辑表达式或关系表达式。系统对表达式的值进行判断，若为**0**，**x**，**z**，按“假”处理；若为**1**，按“真”处理，执行指定的语句。

(2) 第2)、第3)种形式的if语句，在每个else前面有一个分号，整个语句结束处有一个分号。

例如:

```
if(a>b)
    out1=int1;
else
    out1=int2;
```



各有一个分号

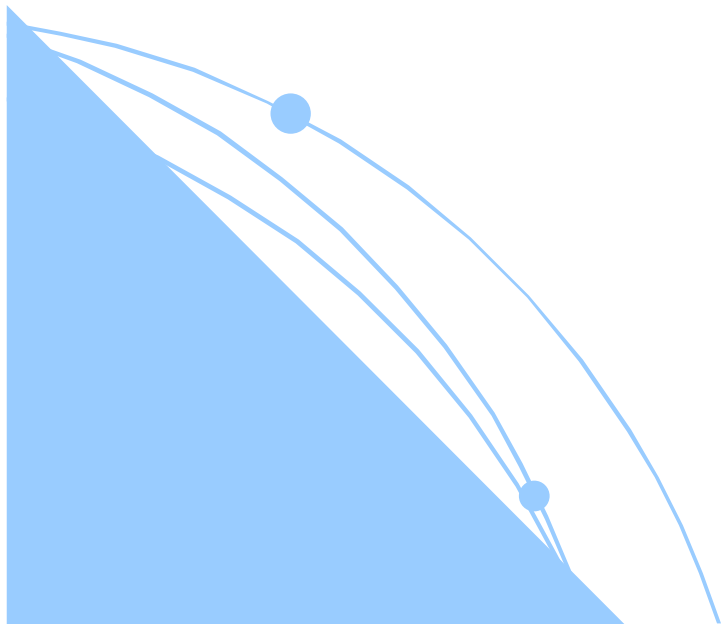
(3) 在if和else后面可以包含一个内嵌的操作语句，也可以有多个操作语句，此时用begin和end这两个关键词将几个语句包含起来成为一个复合块语句。如：

```
if(a>b)
    begin
        out1<=int1;
        out2<=int2;
    end
else
    begin
        out1<=int2;
        out2<=int1;
    end
```

(4) 允许一定形式的表达式简写方式。如下面的例子：

if(expression) 等同与 if(expression==1)

if(!expression) 等同与 if(expression!=1)



(5) if语句的嵌套。在if语句中又包含一个或多个if语句称为if语句的嵌套。一般形式如下：

```
if(expression1)
    if(expression2)  语句1; (内嵌if)
    else              语句2;
else
    if(expression3)  语句3; (内嵌if)
    else              语句4;
```

应当注意if与else的配对关系，**else**总是与它上面最近的**if**配对。如果if与else的数目不一样，为了实现程序设计者的目的，可以用begin_end块语句来确定配对关系。例如：

```
if( )
    begin
        if( ) 语句1 (内嵌if)
        end
    else
        语句2
```

5.2 case语句

一般形式如下：

(1) case (表达式)	<case分支项>	endcase
(2) casez (表达式)	<case分支项>	endcase
(3) casex (表达式)	<case分支项>	endcase

case分支项的一般格式如下：

分支表达式:	语句;
默认项(default 项):	语句;

说明:

(1) **case**括弧内的表达式称为控制表达式，**case**分支项中的表达式称为分支表达式。

(2) 当控制表达式的值与分支表达式的值相等时，执行分支表达式后面的语句。如果所有的分支表达式的值都没有与控制表达式的值相匹配，就执行**default**后面的语句。

(3) 一个**case**语句里只准有一个**default**项。对于**default**项：
若分支表达式将所有情况列出，可不写**default**项；
若有未列出的可能值，应写**default**项。

case (表达式) <case分支项> endcase

下面是一个简单的使用case语句的例子。该例子中对寄存器rega译码，以确定result的值。

```
reg [15: 0] rega;  
reg [9: 0] result;  
case(rega)  
16'd0:    result=10'b0111111111;  
16'd1:    result=10'b1011111111;  
16'd2:    result=10'b1101111111;  
16'd3:    result=10'b1110111111;  
16'd4:    result=10'b1111011111;  
16'd5:    result=10'b1111101111;  
16'd6:    result=10'b1111110111;  
16'd7:    result=10'b1111111011;  
16'd8:    result=10'b1111111101;  
16'd9:    result=10'b1111111110;  
default:  result=10'bx;  
endcase
```

(4) 每一个**case**分项的分支表达式的值必须互不相同，否则就会出现矛盾(对表达式的同一个值，有多种执行方案)。

(5) 执行完**case**分项后的语句，则跳出该**case**语句结构，终止**case**语句的执行。（注：与C语言不同，Verilog中的**case**语句不需要写**break**）

(6) 在用**case**语句表达式进行比较的过程中，只有当信号的对应位的值能明确进行比较时，比较才能成功。因此，要注意详细说明**case**分项的分支表达式的值。

(7) **case**语句的所有表达式值的位宽必须相等，只有这样，控制表达式和分支表达式才能进行对应位的比较。一个经常犯的错误是用'**bx**，'**bz**来替代**n'bx**，**n'bz**，这是不对的，因为信号**x**，**z**的默认宽度是机器的字节宽度，通常是32位(此处**n**是**case**控制表达式的位宽)。

下面将给出case,casez,casex的真值表，如表1.5.1所列。

表5.1 case，casez和casex的真值表

case	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

casez	0	1	x	z
0	1	0	0	1
1	0	1	0	1
x	0	0	1	1
z	1	1	1	1

不考虑Z值

casex	0	1	x	z
0	1	0	1	1
1	0	1	1	1
x	1	1	1	1
z	1	1	1	1

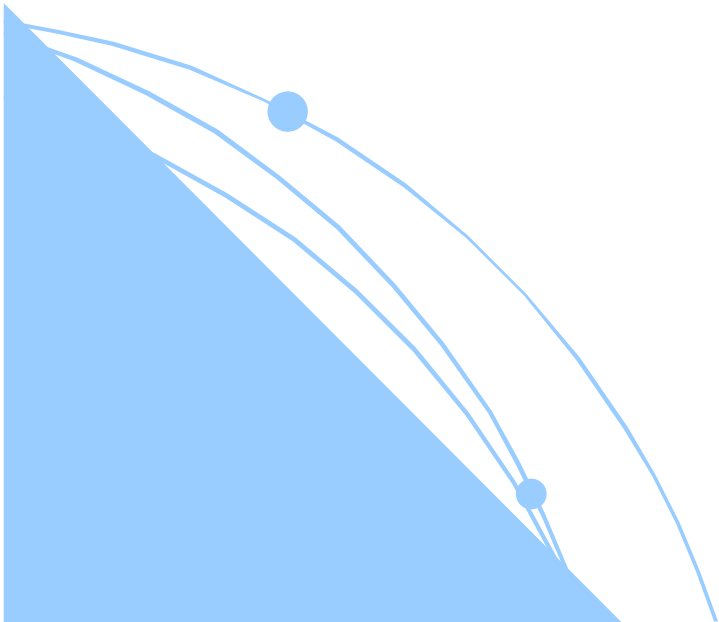
不考虑X和Z值

case语句与**if_else_if**语句的区别主要有两点：

(1)与**case**语句中的控制表达式和多分支表达式这种比较结构相比，**if_else_if**结构中的条件表达式更为直观一些。

(2)对于那些分支表达式中存在不定值**x**和高阻值**z**的位时，**case**语句提供了处理这种情况的手段。

(3)**if**语句具有优先级，可用于描述具有优先级的电路；**case**语句不具有优先级，只能用于描述无优先级的电路。



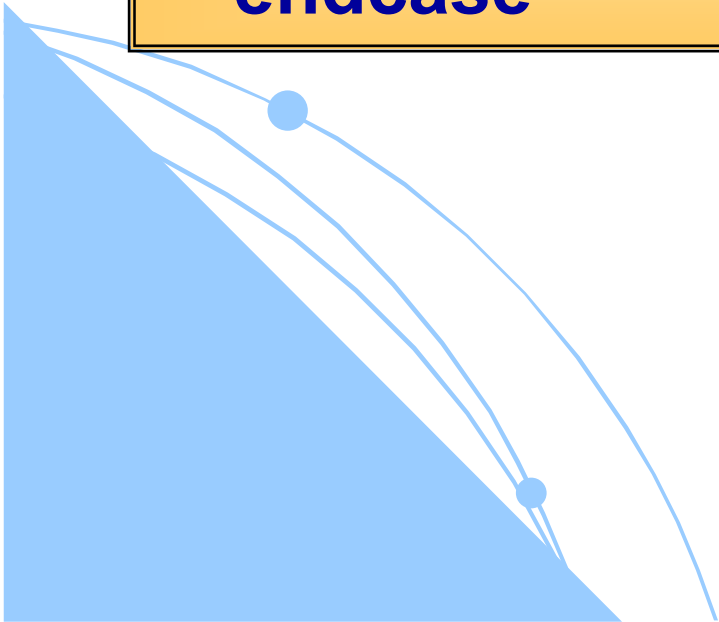
下面的两个例子介绍了处理值为x，z位的case语句。

【例5.1】

```
case(select[1: 2])  
  2'b00:  result=0;  
  2'b01:  result=flaga;  
  2'b0x,  
  2'b0z:  result=flaga?'bx: 0;  
  2'b10:  result=flagb;  
  2'bx0,  
  2'bz0:  result=flagb?'bx: 0;  
  default: result=' bx;  
endcase
```


【例5.2】

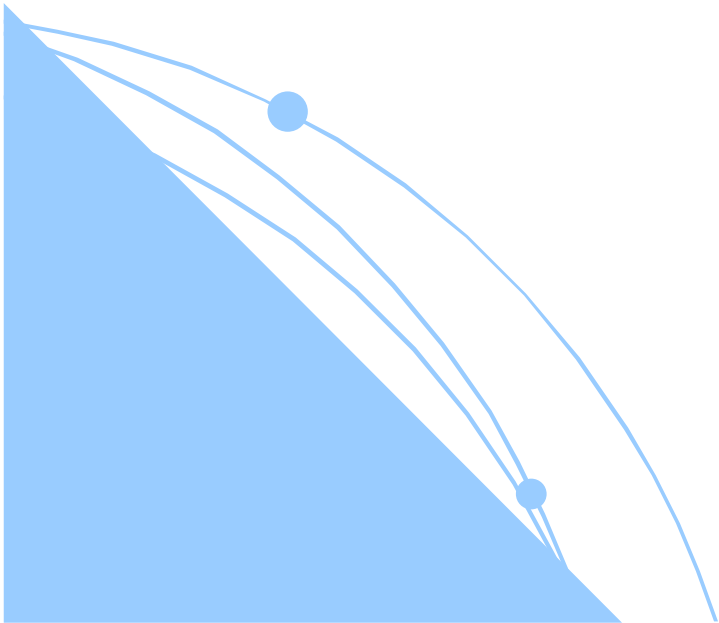
```
case(sig)  
  1'bz:  $display("signal is floating");  
  1'bx:  $display("signal is unknown");  
  default: $display("signal is %b", sig);  
endcase
```



Verilog HDL针对电路的特性提供了**case**语句的其他两种形式，即**casez**和**casex**，用来处理比较过程中的不必考虑的情况(don't care condition)。

其中**casez**语句用来处理不考虑高阻值**z**的比较过程，**casex**语句则将高阻值**z**和不定值**x**都视为不必关心的情况。

所谓不必关心的情况，即在表达式进行比较时，不将该位的状态考虑在内。这样，在**case**语句表达式进行比较时，就可以灵活地设置对信号的某些位进行比较。见下面的两个例子：



【例5.3】

```
reg[7: 0] ir;  
casez(ir)  
8'b1??????: instruction1(ir);  
8'b01?????: instruction2(ir);  
8'b00010???: instruction3(ir);  
8'b000001??: instruction4(ir);  
endcase
```

【例5.4】

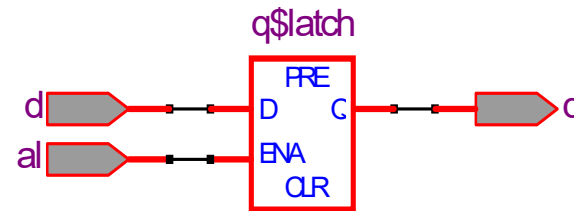
```
reg[7: 0] r, mask;  
mask=8'bx0x0x0x0;  
case(r^mask) //r^mask=x-x-x-x-  
8'b001100xx: stat1;  
8'b1100xx00: stat2;  
8'h00xx0011: stat3;  
8'bx001100: stat4;  
endcase
```

使用条件语句不当在设计中生成了不必要的锁存器

Verilog HDL设计中容易犯的一个通病是由于对语言理解不全面，使用不准确，从而生成了并不想要的锁存器。下面给出了一个在“**always**”块中不正确使用**if**语句，造成这种错误的例子。

```
always @(al or d)  
begin  
if(al) q=d;  
end
```

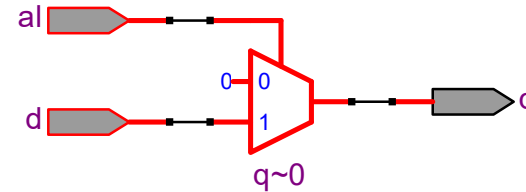
有锁存器



使用综合工具进行
综合后的电路结果

```
always @(a1 or d)
begin
  if(a1) q=d;
  else   q=0;
end
```

无锁寄存器



使用综合工具进行
综合后的电路结果

要点：在"**always**"块内，如果在给定的条件下变量没有赋值，这个变量将保持原值，也就是说会生成一个锁存器。

Verilog HDL程序的另一种偶然生成锁存器是在使用case语句时缺少default项的情况下发生的。

```
always @(sel[1:0] or a or b)
case (sel[1:0])
2'b00: q<=a;
2'b11: q<=b;
endcase
```

有锁存器

```
always @(sel[1:0] or a or b)
case (sel[1:0])
2'b00: q<=a;
2'b11: q<=b;
default: q<='b0
endcase
```

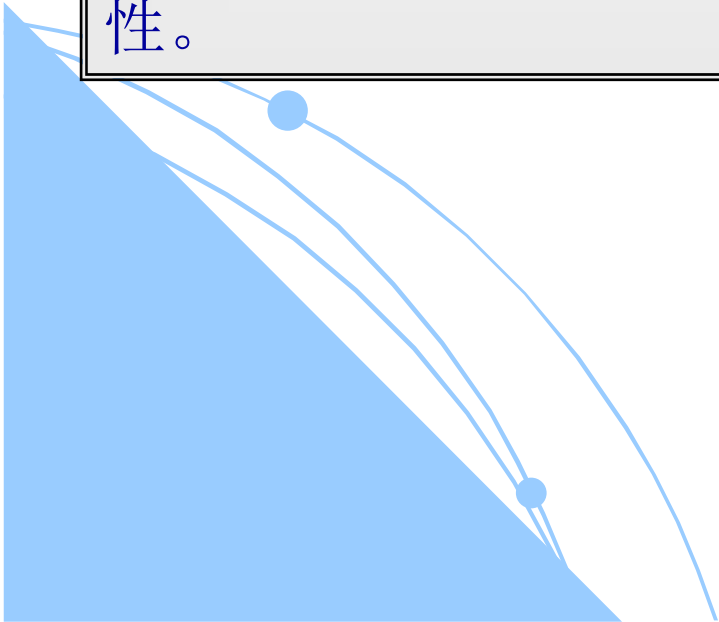
无锁存器

总结：

为避免产生不必要的锁存器，应遵循下面两个原则：

- 1、如果用到**if**语句，最好写上**else**项；
- 2、如果用**case**语句，最好写上**default**项。

遵循上面两条原则，就可以避免发生这种设计错误，使设计者更加明确设计目标，同时也增强了Verilog程序的可读性。



5.3 条件语句的语法

//第一类条件语句

```
if (!lock) buffer=data;  
if (enable) out=in;
```

//第二类条件语句

```
if(number_queued<MAX_Q_DEPTH)  
begin  
    data_queue=data;  
    number_queued=number_queued+1;  
end  
else  
    $display("Queue Full.Try again");
```

//第三类条件语句

if (alu_control==0)

 y=x+z;

else if(alu_control==1)

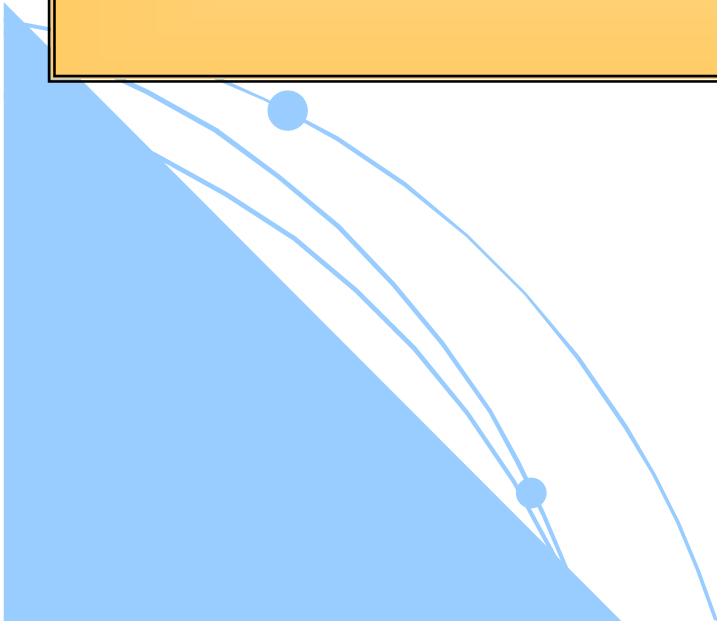
 y=x-z;

else if(alu_control==2)

 y=x*z;

else

 \$display("Invalid ALU control signal");



5.4 多路分支语句

【例5.6】 使用**case**语句实现四选一多路选择器

```
module
mux4_to_1(out,i0,i1,i2,i3,s1,s0);
//根据输入/输出图的端口声明
output out;
input i0,i1,i2,i3;
input s1,s0;
//输出声明为寄存器类型
reg out;
/*任何信号改变，都会引起输出信号的重新计算,使输出out重新计算的所有输入信号必须写入always@(...)的变量列表中*/
```

```
always @(s1 or s0 or i0
or i1 or i2 or i3)
begin
    case({s1,s0})
        2'b00:out=i0;
        2'b01:out=i1;
        2'b10:out=i2;
        2'b11:out=i3;
        default:out=1'bx;
    endcase
end
endmodule
```

5.5 循环语句

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



5.5.1 forever语句

forever语句的格式如下：

forever 语句；

或

forever

begin

多条语句

end

forever循环语句常用于产生周期性的波形，作为仿真测试信号。它与**always**语句不同之处在于它不能独立写在程序中，而必须写在**initial**块中。

5.5.2 repeat语句

repeat语句的格式如下：

repeat(表达式) 语句;

或

repeat(表达式)
begin
多条语句
end

在repeat语句中，其表达式通常为常量表达式。

移位相加乘法器原理

$$\begin{array}{r} 1001 \quad (9) \\ \times 0110 \quad (6) \\ \hline 0000 \\ 1001 \\ 1001 \\ + 0000 \\ \hline 00110110 \quad (54) \end{array}$$

$$\begin{array}{r}
 1001 \quad (9) \\
 \times 0110 \quad (6) \\
 \hline
 0000 \\
 1001 \\
 1001 \\
 + 0000 \\
 \hline
 00110110 \quad (54)
 \end{array}$$

```
parameter size=8, longsize=16;
reg [size: 1] opa, opb;
reg [longsize: 1] result;
.....
begin: mult
reg [longsize: 1] shift_opa, shift_opb;
shift_opa=opa;
shift_opb=opb;
result=0;
repeat(size)
begin
    if(shift_opb[1])
        result=result+shift_opa;
        shift_opa=shift_opa<<1;
        shift_opb=shift_opb>>1;
    end
end
end
```


5. 2. 3 while语句

while语句的格式如下：

while(表达式) 语句

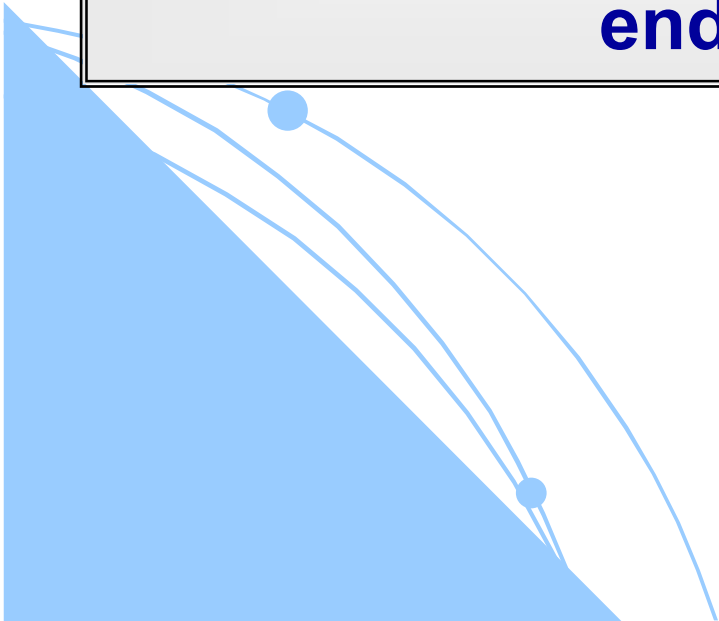
或

while(表达式)

begin

多条语句

end



下例用while循环语句对rega这个8位二进制数中值为1的位进行计数。

```
begin: count1s  
  reg[7: 0] tempreg;  
  count=0;  
  tempreg=rega;  
  while(tempreg)  
  begin  
    if(tempreg[0]) count=count+1;  
    tempreg=tempree>>1;  
  end  
end
```

5.5.4 for语句

for语句的一般形式为：

for(表达式1;表达式2;表达式3) 语句;

它的执行过程为：

- (1)先求解表达式1。
- (2)再求解表达式2，若其值为真(非0)，则执行for语句中指定的内嵌语句，然后执行下面的第3步；若为假(0)，则结束循环，转到第(5)步。
- (3)若表达式2为真，在执行指定的语句后，求解表达式3。
- (4)转回第(2)步继续执行。
- (5)执行for语句下面的语句。

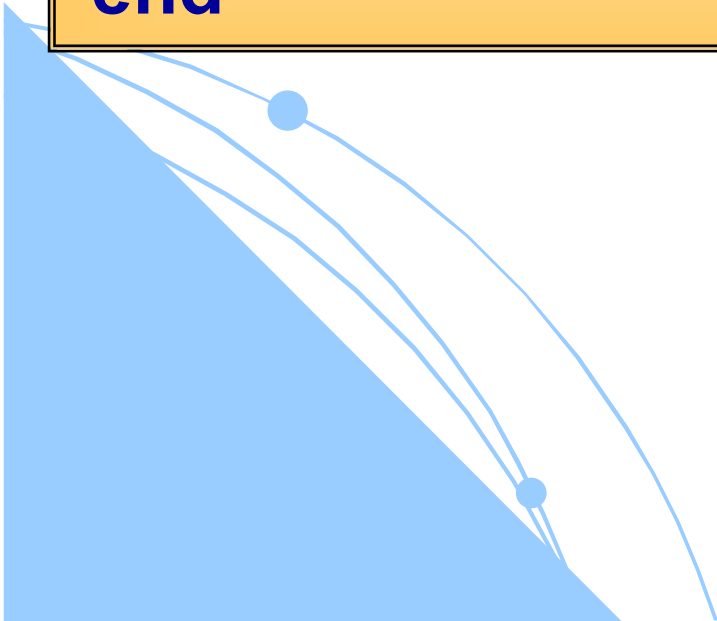
for语句最简单的应用形式是很容易理解的，其形式如下：

**for(循环变量赋初值；循环结束条件；循环变量增值)
 执行语句;**

下面分别举两个使用for循环语句的例子。例5. 5用for语句来初始化memory。

【例5.7】

```
begen: init_mem  
reg[7: 0] temp_i;  
for(temp_i=0; temp_i<memsize; temp_i=temp_i+1)  
    memory[temp_i]=0;  
end
```



例5.8则用**for**循环语句来实现前面用**repeat**语句实现的乘法器。

【例5.8】

$$\begin{array}{r}
 1001 \quad (9) \\
 \times 0110 \quad (6) \\
 \hline
 0000 \\
 1001 \\
 1001 \\
 + 0000 \\
 \hline
 00110110 \quad (54)
 \end{array}$$

```
parameter size=8, longsize=16;
reg [size: 1] opa, opb;
reg [longsize: 1] result;
.....
begin: mult
reg [longsize: 1] shift_opa, shift_opb;
shift_opa=opa;
shift_opb=opb;
result=0;
repeat(size)
begin
    if(shift_opb[1])
        result=result+shift_opa;
        shift_opa=shift_opa<<1;
        shift_opb=shift_opb>>1;
    end
end
```

例5.8则用for循环语句来实现前面用repeat语句实现的乘法器。

【例5.8】

```
parameter size=8, longsize=16;  
reg [size: 1] opa, opb;  
reg [longsize: 1] result;  
.....  
begin: mult  
    integer bindex;  
    result=0;  
    for(bindex=1; bindex<=size; bindex=bindex+1)  
        if(opb[bindex])  
            result=result+(opa<<(bindex-1));  
end
```

在**for**循环中，循环变量增值表达式可以不必是一般的常规加法或减法表达式。请用**for** 循环重新实现下面的语句：即用**for** 循环实现**rega**八位二进制数中值为**1**的位进行计数

```
begin: count1s
    reg[7: 0] tempreg;
    count=0;
    tempreg=rega;
    while(tempreg)
    begin
        if(tempreg[0]) count=count+1;
        tempreg=tempreg>>1;
    end
end
```

在for循环中，循环变量增值表达式可以不必是一般的常规加法或减法表达式。下面是对rega八位二进制数中值为1的位进行计数的另一种方法。

```
begen: count1s  
    reg[7: 0] tempreg;  
    count=0;  
  
for(tempreg=rega;tempreg;tempreg=tempreg>>1)  
    if(tempreg[0])  
        count=count+1;  
end
```


5.6 顺序块和并行块

5.6.1 块语句的类型

块语句包括两种类型：

顺序块 并行块

1、顺序块——过程块

关键字 **begin...end**

用于将多条语句组成顺序块。其特点如下：

- (1) 顺序块中的语句是一条接一条顺序执行的；
- (2) 如果语句包括延迟或事件控制，那么延迟总是相对于前面一条语句执行完成的仿真时间；

【例5.9】

//说明1

```
reg x,y;  
reg [1:0] z,w;  
initial  
begin  
    x=1'b0;  
    y=1'b1;  
    z={x,y};  
    w={y,x};  
end
```

仿真0时刻x、y、z、w的最终值分别为0、1、1、2。执行这4个赋值语句有顺序，但不需要执行时间。

//说明2:带延迟的顺序块

```
reg x,y;  
reg [1:0] z,w;  
initial  
begin  
    x=1'b0; //仿真时刻0完成  
    #5 y=1'b1; //仿真时刻5完成  
    #10 z={x,y}; //仿真时刻15完成  
    #20 w={y,x}; //仿真时刻35完成  
end
```

x、y、z、w的最终值仍为0、1、1、2，但块语句完成时的仿真时刻为35，除第一句外，以后每执行一条语句都需要等待。

2、并行块

关键字 `fork...join`

其特点如下：

- (1) 并行块内的语句并发执行；
- (2) 语句执行的顺序是由语句内延迟或事件控制决定的；
- (3) 语句中的延迟或事件控制是相对于块语句开始执行的时刻而言的。

注意：

顺序块与并行块的根本区别在于：

当控制转移到块语句的时刻，并行块中所有的语句同时开始执行，语句之间的先后顺序是无关紧要的。

将上例带有延迟控制的顺序块程序改写为并行块来分析，如下：

【例5.10】**并行块**

```
reg x,y;  
reg [1:0] z,w;  
initial  
fork  
    x=1'b0; //仿真时刻0完成  
    #5 y=1'b1; //仿真时刻5完成  
    #10 z={x,y}; //仿真时刻10完成  
    #20 w={y,x}; //仿真时刻20完成  
join
```

所有语句在仿真0时刻开始执行，仿真结果与前例相同。这个并行块执行结束的时间为第20个仿真时间单位，而不再是35个。

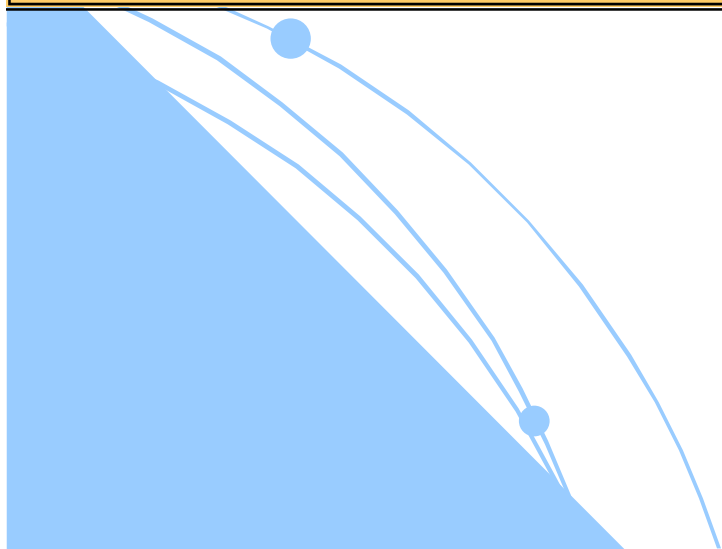
【例5.10】

请用并行块语句实现如下顺序块的仿真时序及功能

//说明2:带延迟的顺序块

```
reg x,y;  
reg [1:0] z,w;  
initial  
begin  
    x=1'b0;  
    #5 y=1'b1;  
    #10 z={x,y};  
    #20 w={y,x};  
end
```

```
reg x,y;  
reg [1:0] z,w;  
initial  
fork  
    x=1'b0;  
    #5 y=1'b1;  
    #15 z={x,y};  
    #35 w={y,x};  
join
```



```
reg x,y;  
reg [1:0] z,w;  
initial  
fork  
    x=1'b0;  
    y=1'b1;  
    z={x,y};  
    w={y,x};  
join
```

该程序如何执行？

使用并行块时需要注意，如果两条语句在同一时刻对同一变量产生影响，将会引起隐含的竞争，这种情况需要避免。

上例中的z、w可能为0、1，也可能为2'bxx，这与所用仿真器软件工具有关，而和语言本身无关。

5.6.2 块语句的特点——嵌套块、命名块和命名块的禁用

1、嵌套块

块可以嵌套使用，顺序块和并行块能够混合在一起使用。如【例5.11】所示。

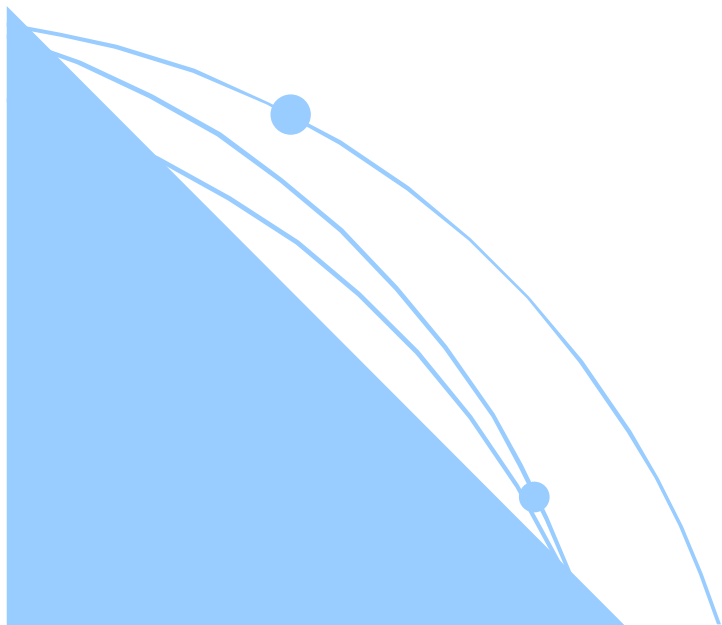
【例5.11】

```
//嵌套块
initial
begin
    x=1'b0;
    fork
        #5 y=1'b1;
        #10 z={x,y};
    join
        #20 w={y,x};
end
endmodule
```

2、命名块

块可以具有自己的名字，称为命名块。特点如下：

- (1) 命名块中可以声明局部变量；
- (2) 命名块是设计层次的一部分，命名块中声明的变量可以通过层次名引用进行访问；
- (3) 命名块可以被禁用，停止其执行。



【例5.11】

```
//命名块
module top;
initial
begin: block1 //名字为block1的顺序命名块
    integer i; //整型变量i是block1命名块的静态本地变量
                //可以用层次名top.block1.i被其它模块访问
    ...
end
initial
fork: block2 //名字为block2的并行命名块
    reg i; //寄存器变量i是block2命名块的静态本地变量
            //可以用层次名top.block2.i被其它模块访问
    ...
join
```

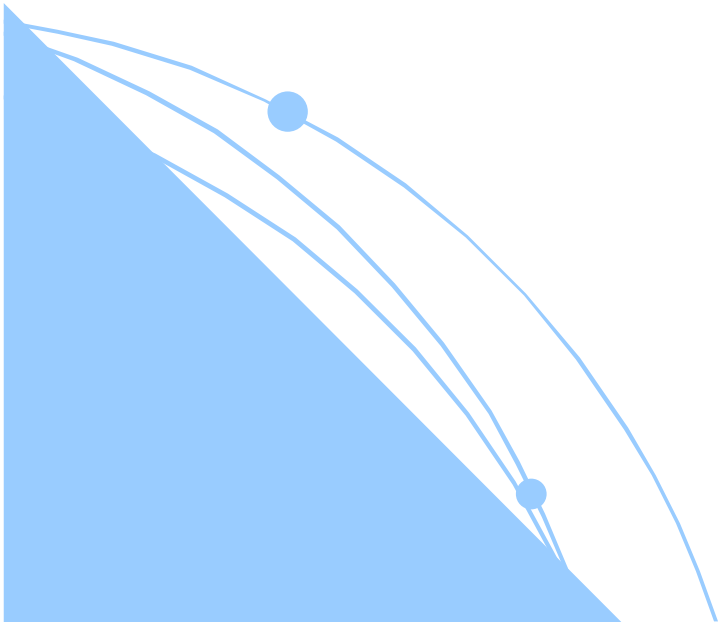
3、命名块的禁用

关键字: **disable**

Verilog通过**disable**提供了一种中止命名块执行的方法。**disable**可以用来从循环中退出、处理错误条件以及根据控制信号来控制某些代码段是否被执行。

块语句禁用后，紧接在块语句后的语句被执行。

disable类似C语言中的**break**，但**break**只能退出当前所在的循环，而**disable**则可以禁用设计中任意一个命名块。



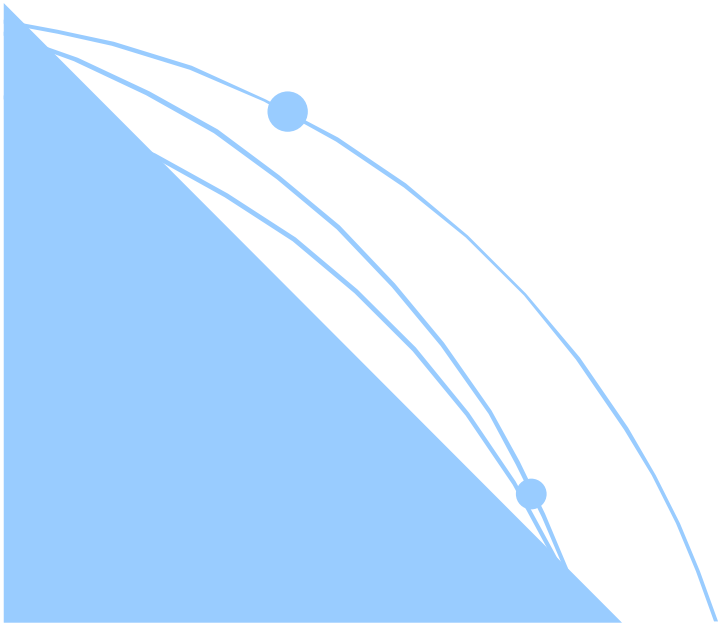
【例5.13】

命名块的禁用,从寄存器的低有效位开始查找第一个值为1的位

```
reg [15:0] flag;  
integer i;  
initial  
begin  
    flag=16'b0010_0000_0000_0000;  
    i=0;  
    begin: block1  
        while(i<16)  
            begin  
                if(flag[i])  
                    begin  
                        $display("Encountered a True bit at element number %d",i);  
                        disable block1; //找到值为1的位，禁用block1  
                    end  
                i=i+1;  
            end  
        end  
    end  
end
```

5.7 生成块

略。



5.8 举例

【例5.18】行为级描述的四选一多路选择器

```
module mux4_to_1(out,i0,i1,i2,i3,s1,s0);  
output out;  
input i0,i1,i2,i3;  
input s1,s0;  
reg out;  
always @(s1 or s0 or i0 or i1 or i2 or i3)  
begin  
    case({s1,s0})  
        2'b00:out=i0;  
        2'b01:out=i1;  
        2'b10:out=i2;  
        2'b11:out=i3;  
        default:out=1'bx;  
    endcase  
end  
endmodule
```

【例5.19】四位计数器的行为级描述

```
module counter(Q, clock, clear);  
output [3:0] Q;  
input clock, clear;  
reg [3:0] Q;  
  
always @(posedge clear or negedge clock)  
begin  
    if(clear)  
        Q<=4'd0;  
    else  
        Q<=Q+1;  
end  
endmodule
```

思考题

- 1、为什么建议在编写Verilog模块程序时，如果用到if语句，建议大家把配套的else情况也考虑在内？
- 2、用if（条件1）语句；else if（条件2）语句；elseif（条件3）语句；…else 语句和用case endcase表示不同条件下的多个分支是完全相同的，还是有什么不同？
- 3、如果case语句的分支条件没有覆盖所有可能的组合条件，定义了default项和没有定义default项有什么不同？
- 4、仔细阐述case、casex和casez之间的不同。
- 5、forever语句如果运行了，在它下面的语句能否运行？它位于begin—end块和位于fork—join块有什么不同？
- 6、forever语句、repeat语句能否独立于过程块而存在，即能否不在initial或always块中使用？
- 7、用for循环为存储器许多单元赋值时是否需要时间？为什么如果不定义时间延迟，它可以不需要时间就把不管多大的存储器赋值完毕？
- 8、for循环是否可以表示可以综合的组合逻辑？请举例说明。

9、在编写测试模块时用什么方法可以使for循环按照时钟的节拍运行？请比较图5.3所示程序段。

```
always @(posedge clk)
begin
    for(i=0;i<=1024;i=i+1)
        mem[i]=i;
end
```

这样写能不能按照时钟节拍来对mem[i]赋值？右边框内的程序呢？

```
initial
begin
    for(i=0;i<=1024;i=i+1)
        begin
            mem[i]=i;
            @(posedge clk)
        end
end
```

图5.3 两种程序段

- 10、声明一个名为oscillate的寄存器变量并将它初始化为0，使其每30个时间单位进行一次取反操作。不要使用always语句（提示：使用forever循环）。
- 11、设计一个周期为40个时间单位的时钟信号，其占空比为25%。使用always和initial块进行设计，将其在仿真0时刻的值初始化为0。
- 12、给定下面含有阻塞过程赋值语句的initial块，每条语句在什么仿真时刻开始执行？a、b、c和d在仿真过程中的中间值和仿真结束时的值是什么？

```
initial
begin
a=1' b0;
b=#10 1' b0;
c=#5 1' b0;
d=#20 {a, b, c};
end
```

- 13、在第12题中，如果initial块中包括的是非阻塞过程赋值语句，那么各个问题的答案是什么？

14、下面例子中d的最终值是什么？

```
initial
begin
b=1' b1;c=1' b0;
#10 b=1' b0;
end
```

```
initial
begin
    d=#25(b|c);
end
```

15、使用带有同步清零端的D触发器（清零端高电平有效，在时钟下降沿执行清零操作）设计一个下降沿触发器的D触发器，只能使用行为语句（提示：D触发器的输出q应当声明为寄存器变量）。使用设计出的D触发器输出一个周期为10个时间单位的时钟信号。

16、使用带有异步清零端的D触发器设计第15题中要求的D触发器（在清零端变为高电平后立即执行清零操作，无需等待下一个时钟下降沿），并对这个D触发器进行测试。

17、使用wait语句设计一个电平敏感锁存器，该锁存器的输入信号为d和clock，输出为q，其功能是当clock=1时q=d。

18、使用条件语句设计【例5.17】中的四选一多路选择器，外部端口必须保持不变。

19、使用case语句设计八功能的算术运算单元（ALU），其输入信号a和b均为4位，还有功能选择信号select为3位，输出信号为out（5位）。算术运算单元ALU所执行的操作与select信号有关，具体关系如表5.1所列（忽略输出结果中的上溢和下溢的位）。

表 5.1 select信号的功能

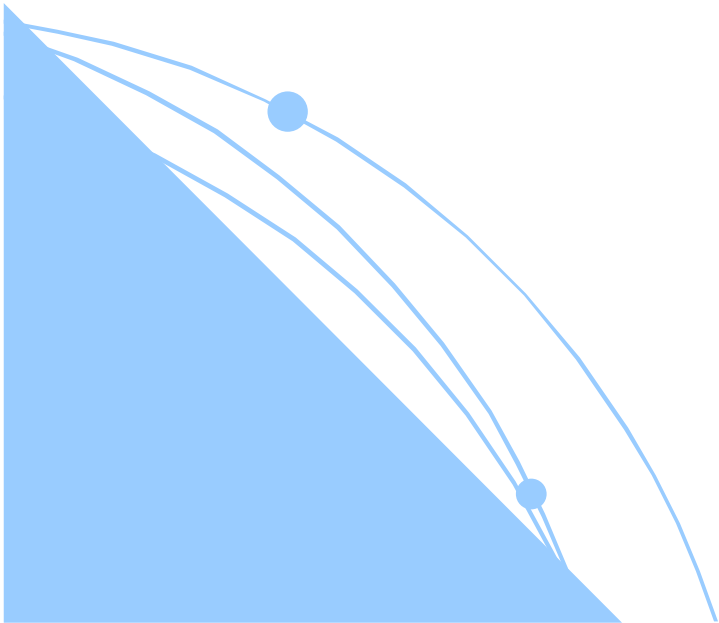
select信号	功能
3'b000	out=a
3'b001	out=a+b
3'b010	out=a-b
3'b011	out=a/b
3'b100	out=a%b
3'b101	out=a<<1
3'b110	out=a>>1
3'b111	out=a>b(大小比较)

- 20、使用while循环设计一个时钟信号发生器。其时钟信号的初值为0，周期为10个时间单位。
- 21、使用for循环对一个长度为1024（地址从0~1023）、位宽为4的寄存器类型数组cache_var进行初始化，把所有单元都设置为0。
- 22、使用forever循环设计一个时钟信号，周期为10，占空比为40%，初值为0。
- 23、使用repeat将语句a=a+1延迟20个时钟上升沿之后再执行。
- 24、下面是一个内嵌顺序块和并行块的块语句。该块的执行结束时间是多少？事件的顺序是怎样的？每条语句的仿真结束时间是多少？

```
initial  
begin  
  x=1'b0;  
  #5 y=1'b1;  
  fork  
    #20 a=x;  
    #15 b=y;  
  join  
  #40 x=1'b1;
```

```
fork  
  #10 p=x;  
  begin  
    #10 a=y;  
    #30 b=x;  
  end  
  #5 m=y;  
join  
end
```

25、用forever循环语句、命名块（named block）和禁用（disabling of）命名块来设计一个八位计数器。这个计数器从count=5开始计数，到count=67结束计数。每个时钟正跳变沿计数器加一，时钟的周期为10. 计数器的计数只用了一次循环，然后就被禁用了（提示：使用disable语句）。



第六章 结构语句、系统任务、函数语句和显示系统任务

6.1 结构说明语句

6.2 **task**和**function**说明语句

6.3 关于使用任务和函数的小结

6.4 常用的系统任务

6.5 其它系统函数和任务

小结

思考题

6.1 结构说明语句

Verilog语言中的任何过程模块都从属于以下4种结构的说明语句:

{
 initial说明语句
 always说明语句
 task说明语句
 function说明语句

一个程序模块可以有多个initial和always过程块。每个initial和always语句在仿真的一开始同时立即开始执行。initial语句只执行一次，而always语句则是不断地重复活动着，直到仿真过程结束。

always语句后跟着的过程块是否运行，要看它的触发条件是否满足，如满足则运行过程块一次，再次满足则再运行一次，直至仿真过程结束。
(注：无条件的always语句是不允许的，将造成仿真死锁！如：
always clock=~clock;)

在一个模块中，使用initial和always语句的次数是不受限制的。

task和function语句可以在程序模块中的一处或多处调用。



6.1.1 initial语句

initial语句的格式如下：

initial

begin

语句1;

语句2;

.....

语句n;

end

【例6.1】用initial 块对存储器变量赋初始值

```
initial  
  begin  
    areg=0;    //初始化寄存器areg  
    for(index=0; index<size; index=index+1)  
      memory[index]=0;    //初始化一个memory  
  end
```

在这个例子中则是用initial语句在仿真开始时对各变量进行初始化，注意这个初始化过程不需要任何时间，即在0ns时间内，便可以完成存储器的初始化工作。

【例6.2】用initial语句来生成激励波形

```
initial
begin
    inputs = ' b000000;    //初始时刻为0
    #10 inputs = ' b011001;
    #10 inputs = ' b011011;
    #10 inputs = ' b011000;
    #10 inputs = ' b001000;
end
```

从这个例子中，我们可以看到initial语句的另一用途，即用initial语句来生成激励波形作为电路的测试仿真信号。

注意：一个模块中可以有多多个initial块，它们都是并行运行的。initial块常用于测试文件的编写，用来产生仿真测试信号和设置信号记录等仿真环境。

6.1.2 always语句

声明格式如下：

always <时序控制> <语句>

always语句在仿真过程中是不断活动着的，但**always**语句后跟着的过程块是否执行，则要看它的触发条件是否满足，如满足则运行过程块一次；如不断满足，则不断地循环执行。

always语句由于其不断重复执行特性，只有和一定的时序控制结合在一起才有用。如果一个**always**语句没有时序控制(条件或延时)，则这个**always**语句将会生成一个仿真死锁。例如：

always areg = ~areg;

这个**always**语句将会生成一个0延迟的无限循环跳变过程，这时会发生仿真死锁。

但如果加上时序控制，则将变为一条非常有用的描述语句，如：

always #half_period areg = ~areg;

上面的语句生成了一个周期为： $\text{period} = 2 * \text{half_period}$ 的无限延续的信号波形。常用这种方法描述时钟信号，并作为激励信号来测试所设计的电路。

【例6.5】

```
reg [7: 0] counter;  
reg tick;  
always @(posedge areg)  
begin  
    tick = ~tick; //产生何种电路?  
    counter = counter + 1; //产生何种电路?  
end
```

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

要点:

always的时间控制可以是**边沿触发**也可以是**电平触发**;
可以是单个信号也可以是多个信号, 当有多个信号时中间需要用关键字**or**连接。

```
always @(posedge clock or posedge reset)  
begin  
.....  
end
```

由两个边沿触发的
always只要其中一个
沿出现, 就立即执行
一次过程块

```
always @(a Or b or c)  
begin  
.....  
end
```

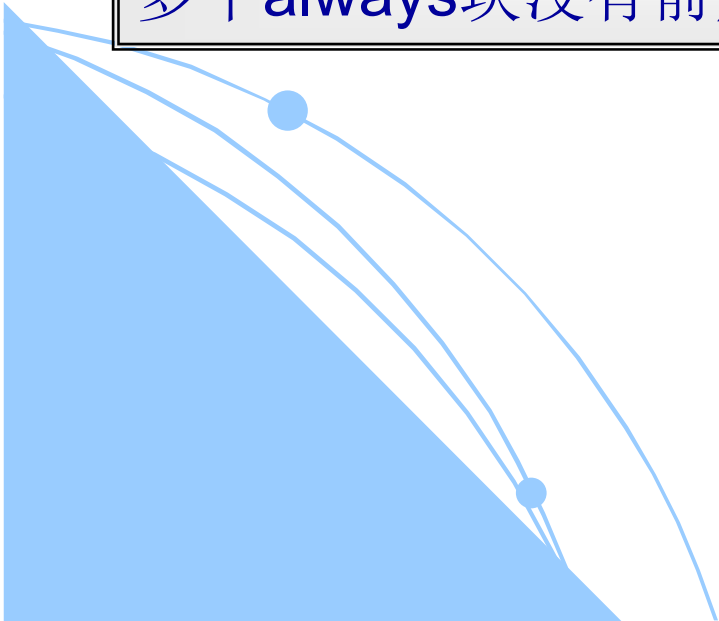
由多个电平触发的
always块, 只要a、b、
c中任何一个发生变化,
从高到低或从低到高
都会执行一次过程块

要点:

沿触发的**always**块用于描述时序逻辑，如有限状态机；
电平触发的**always**块用来描述组合逻辑。

要点:

一个模块中可以有多多个**always**块，它们都是并行运行的，
多个**always**块没有前后之分。



1、always块的or事件控制

当多个信号或者事件中任意一个发生的变化都能够触发语句或语句块的执行。这时，可以使用“or”表达式来表示这种情况。

由关键词“or”连接的多个事件名或信号名组成的列表称为**敏感列表**。

关键词“or”也可以用“，”代替。（MAX+plus II不支持该语法，其它工具如Quartus II可以支持）

【例6.6】

```
//带异步复位的电平敏感锁存器
always @(reset or clock or d)
begin
    if(reset)
        q=1'b0;
    else if(clock)
        q=d;
end
```


【例6.7】

//带异步复位的电平敏感锁存器

```
always @(reset , clock , d)
```

```
begin
```

```
    if(reset)
```

```
        q=1'b0;
```

```
    else if(clock)
```

```
        q=d;
```

```
end
```

//用reset异步下降沿复位，clock正跳变沿触发的D寄存器

```
always @(posedge clk,negedge reset)
```

```
if(!reset)
```

```
    q<=0;
```

```
else
```

```
    q<=d;
```

当需要将所有输入信号列入敏感列表时，可以使用两种方法简化书写：

@* 和 @(*)

【例6.8】@*操作符的使用

//一般写法

```
always @(a or b or c or d or e or f or g or h or p or m)
```

```
begin
```

```
...
```

```
end
```

//简化写法

```
always @(*)
```

```
begin
```

```
...
```

```
end
```

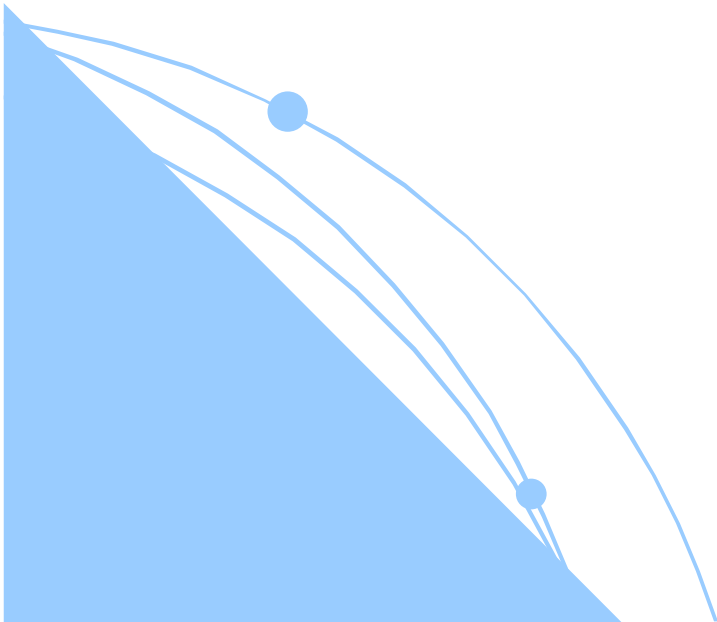
2.电平敏感时序控制

前面的事件控制使用符号@和敏感列表来表示，Verilog中也可以采用另外一种表示电平敏感时序控制的方法：

关键字wait表示等待电平敏感的条件为真。

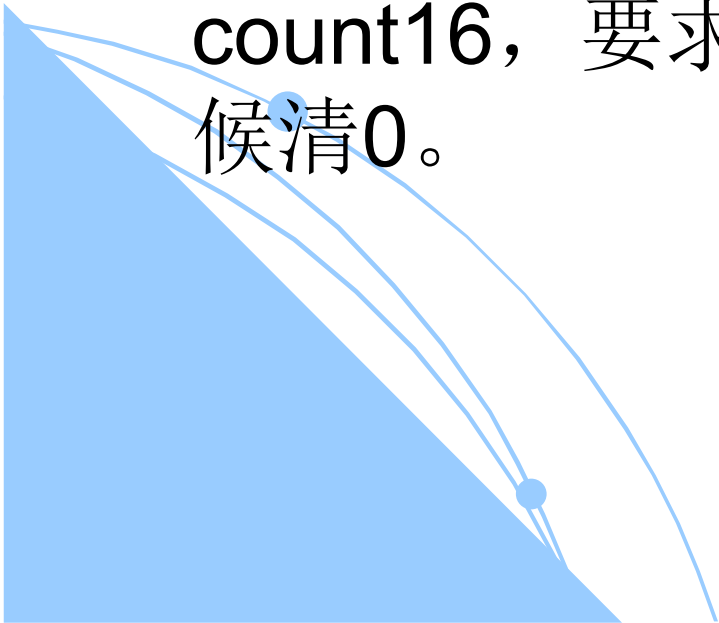
always

```
wait (count_enable) #20 count=count+1;
```



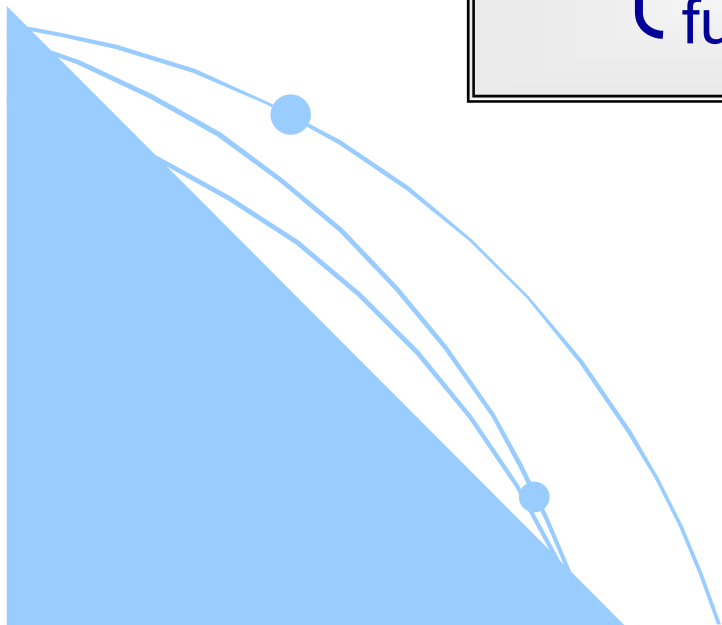
课堂quiz

- 1、实现一个分频器，输入的是系统提供100MHz，输出频率1Hz，占空比是1:2。
- 2、用以上分频器做一个16位的减1计数器count16，要求每1秒减一次，rstn有效的时候清0。



6.2 task和function说明语句

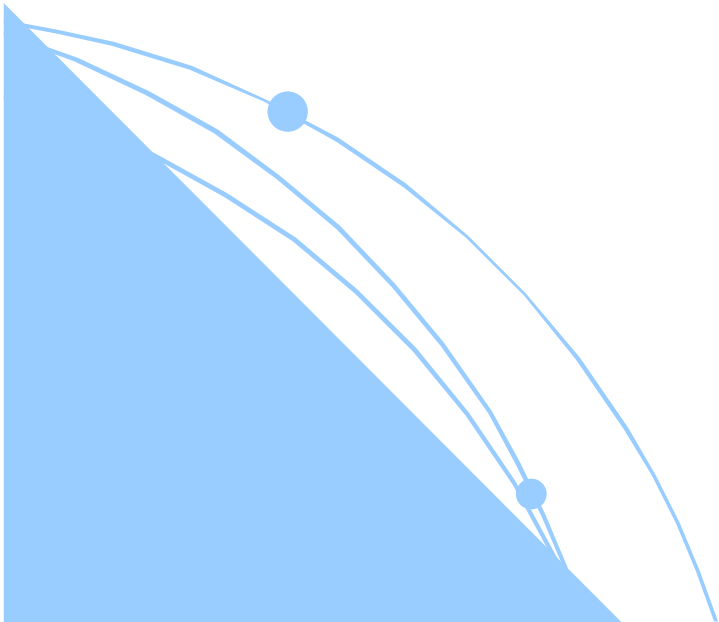
{ task —— 任务
function —— 函数



6.2.1 task和function说明语句的不同点

task和function说明语句的不同点：

- (1)函数只能与主模块共用同一个仿真时间单位，而任务可以定义自己的仿真时间单位。
- (2)函数不能启动任务，而任务能启动其他任务和函数。
- (3)函数至少要有一个输入变量，而任务可以没有或有多多个任何类型的输入变量。
- (4)函数返回一个值，而任务则不返回值。



task和**function**的调用方法不同，例如：

定义一任务或函数对一个**16**位的字进行操作，让高字节与低字节互换，把它变为另一个字(假定这个任务或函数名为：**switch_bytes**)。

任务返回的结果是通过输出端口的变量，因此，**16**位字字节互换任务的调用为：

```
switch_bytes(old_word, new_word);
```

任务**switch_bytes**把输入**old_word**的字的**高、低**字节互换放入**new_word**端口输出，而函数返回的新字是通过函数本身的返回值，因此，**16**位字字节互换函数的调用为：

```
new_word=switch_bytes(old_word);
```

6.2.2 task说明语句

任务可以启动其他的任务，其他任务又可以启动别的任务，可以启动的任务数是没有限制的。不管有多少任务启动，只有当所有的任务启动完成以后，控制才能返回。

(1)任务的定义。定义任务的语法如下：

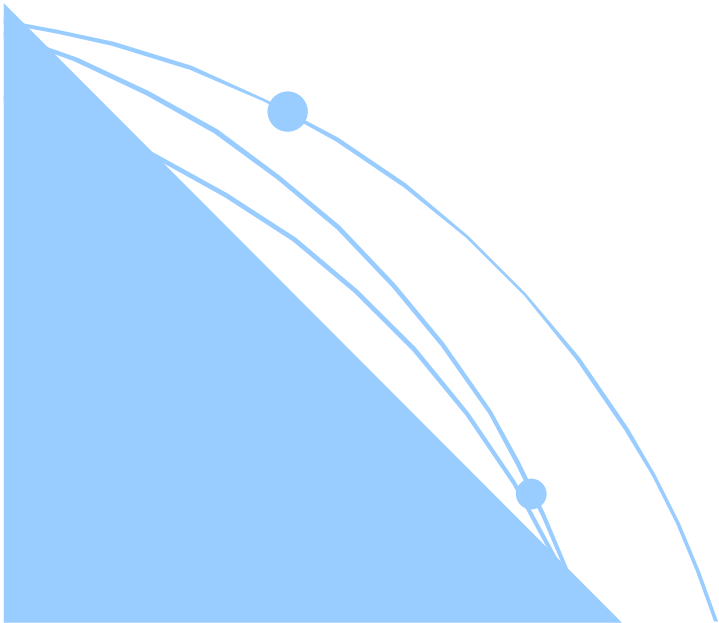
```
task <任务名>;  
    <端口及数据类型声明语句>  
    <语句1>  
    <语句2>  
    .....  
    <语句n>  
endtask
```


(2)任务的调用及变量的传递

启动任务并传递输入，输出变量的声明语句的语法如下：

任务的调用：

<任务名> (端口1, 端口2, ..., 端口n);



下面的例子说明怎样定义任务和调用任务。

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; //赋初始值
```

```
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);
```

说明:

任务调用变量(v, w, x, y, z)和任务定义的I/O变量(a, b, c, d, e)之间是一一对应的。当任务启动时,由v, w和x传入的变量赋给a, b和c,而当任务完成后的输出通过c, d和e赋给x, y和z。

下面是一个具体的例子来说明怎样在模块设计中使用任务，使程序容易读懂。

```
module traffic_lights;
reg clock, red, amber, green; //amber 琥珀色，黄色
parameter on=1, off=0, red_tics=350,
amber_tics=30, green_tics=200;
//交通灯初始化
initial red=off;
initial amber=off;
initial green=off;
//交通灯控制时序
always
begin
    red=on;    //开红灯
    light(red, red_tics) //调用等待任务
    green=on;   //开绿灯
    light(green, green_tics); //等待
    amber=on;   //开黄灯
    light(amber, amber_tics); //等待
end
```

//定义交通灯开启时间的任务

task light;

output color;

input [31: 0] tics;

begin

repeat(tics)

@(posedge clock); //等待tics个时钟的上升沿

color=off; //关灯

end

endtask

//产生时钟脉冲的always块

always

begin

#100 clock=0;

#100 clock=1;

end

endmodule

这个例子描述了一个简单的交通灯的时序控制，并且该交通灯有它自己的时钟产生器。

6.2.3 function说明语句

函数的目的是返回一个用于表达式的值。

(1)定义函数的语法:

function <返回值的类型或范围> (函数名);

 <端口说明语句>

 <变量类型说明语句>

begin

 <语句>

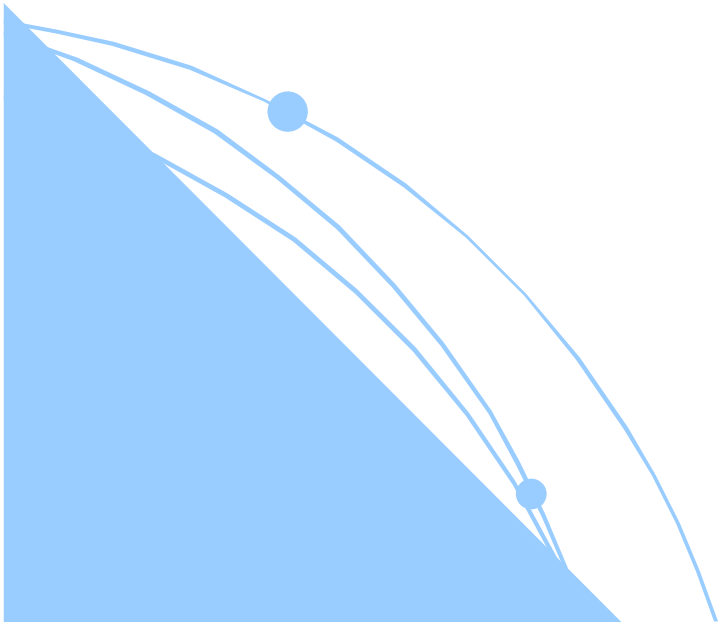
end

endfunction

 注意: <返回值的类型或范围>这一项是可选项, 如缺省则返回值为
一位寄存器类型数据。

下面举例说明：

```
function [7: 0] getbyte;  
  input [15: 0] address;  
  begin  
    <说明语句> //从地址字中提取低字节的程序  
    getbyte=result_expression; //把结果赋予函数的返回字节  
  end  
endfunction
```

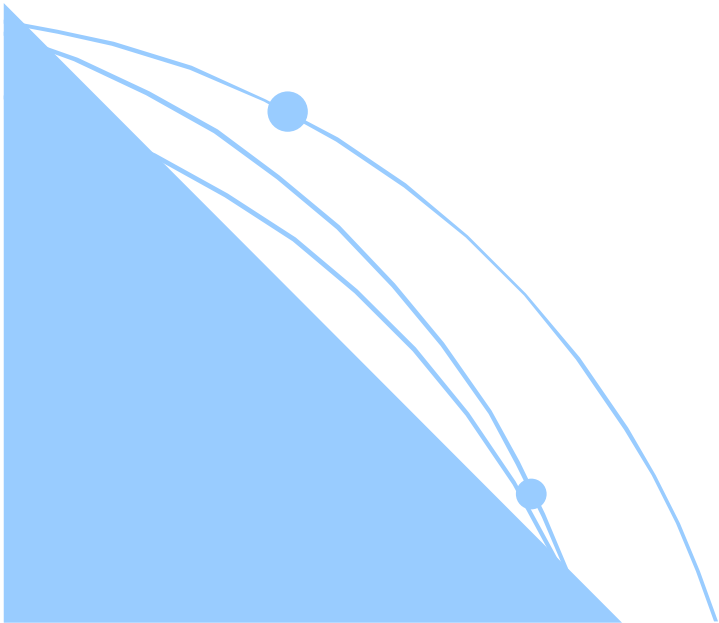


(2)从函数返回的值:

函数的定义隐含声明了与函数同名的内部寄存器。

如在函数的声明语句中<返回值的类型或范围>为缺省，则这个寄存器是一位的，否则是与函数定义中<返回值的类型或范围>一致的寄存器。

函数的定义把函数返回值所赋值寄存器的名称初始化为与函数同名的内部变量。上面的例子说明了这个概念：**getbyte**被赋予的值就是函数的返回值。



(3)函数的调用:

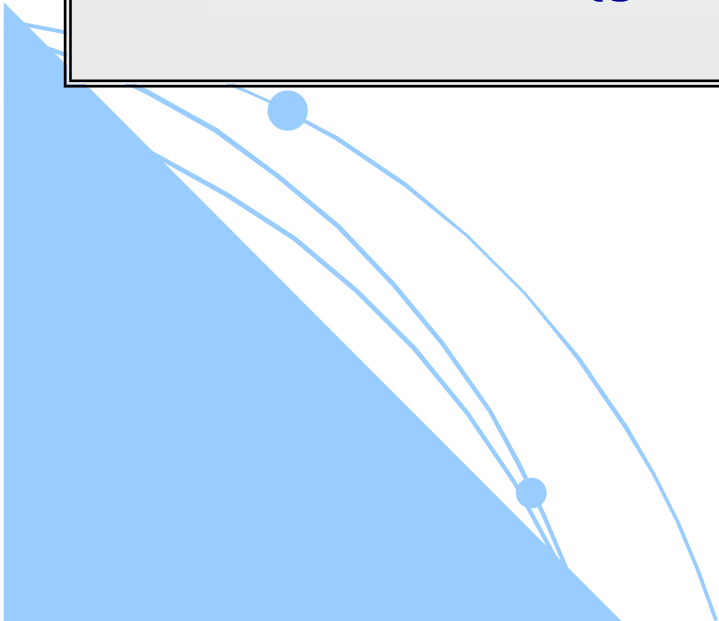
函数的调用是通过将函数作为表达式中的操作数来实现的。其调用格式如下:

变量=<函数名> (<表达式>, <表达式>,);

其中, 函数名作为确认符。

下面的例子中通过对两次调用函数getbyte的结果进行位拼接运算来生成一个字:

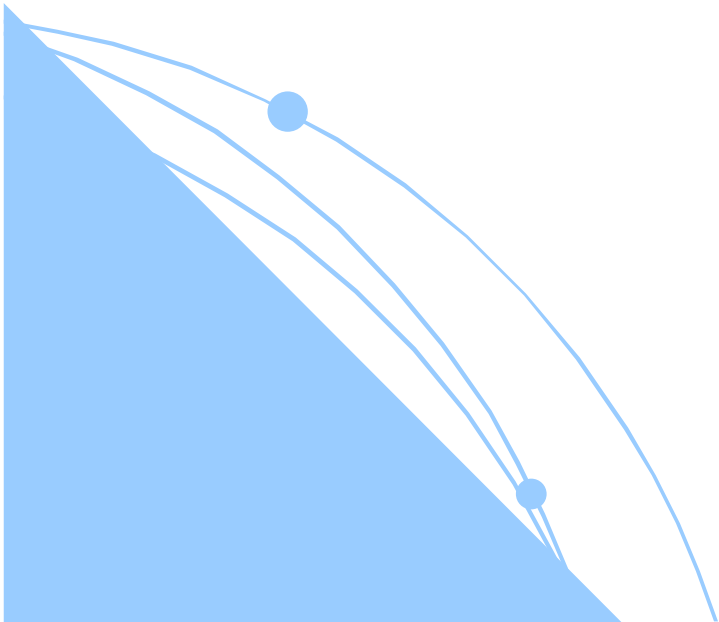
word=control?{getbyte(msbyte), getbyte(lsbyte)}: 0;



(4)函数的使用规则:

与任务相比较, 函数的使用有较多的约束。下面给出的是函数的使用规则:

- 函数的定义不能包含有任何的时间控制语句, 即任何用#、@或wait来标识的语句;
- 函数不能启动任务;
- 定义函数时至少要有一个输入参数;
- 在函数的定义中必须有一条赋值语句给函数的一个内部变量赋结果值, 该内部变量和函数同名。



【例6.10】 定义一个可进行阶乘运算的名为**factorial**的函数，该函数返回一个32位的寄存器型的值，可以后向调用自身并打印结果。

```
module tryfact;  
    //函数的定义  
    function [31:0] factorial;  
        input [3:0] operand;  
        reg [3:0] index;  
        begin  
            factorial=1;  
            for(index=2;index<=operand;index=index+1)  
                factorial=index*factorial;  
        end  
    endfunction
```

//函数的测试

reg [31:0] result;

reg [3:0] n;

initial

begin

result=1;

for(n=2;n<=9;n=n+1)

begin

\$display("Partial result n=%d result=%d",n,result);

result=n*factorial(n)/((n*2)+1);

end

\$display("Finalresult=%d",result);

end

endmodule //模块结束

6.2.4 函数的使用举例

【例6.11】偶校验位的计算

偶校验——数据中“1”的个数为偶数的时候，这个校验位就是“0”，否则这个校验位就是“1”。即加上校验位，1的个数始终为偶数。

```
module parity;  
reg [31:0] addr;  
reg parity;  
initial  
begin  
    addr=32'h3456_789a;  
    #10 addr=32'hc4c6_78ff;  
    #10 addr=32'hff56_ff9a;  
    #10 addr=32'h3faa_aaaa  
end
```

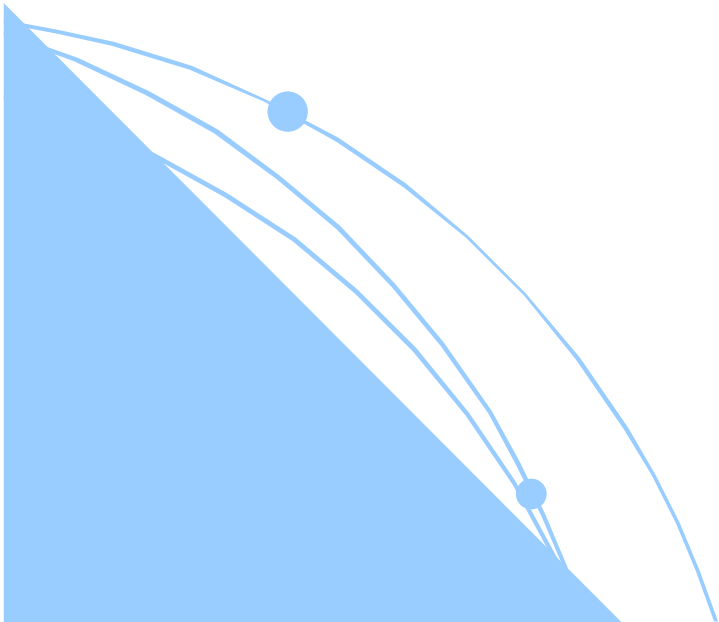
```
//每当地址值发生变化，计算新的偶校验位
always @(addr)
begin
    parity=calc_parity(addr);
    $display("Parity calculated=%b",cal_parity(addr));
end
//定义偶校验计算函数
function calc_parity;
input [31:0] address;
begin
    calc_parity= ^address;//返回所有地址位的异或值
end
endfunction

endmodule
```

可以使用C风格进行函数定义

【例6.12】

```
function calc_parity(input [31:0] address);  
begin  
    calc_parity= ^address;//返回所有地址位的异或值  
end  
endfunction
```



【例6.13】左/右移位寄存器

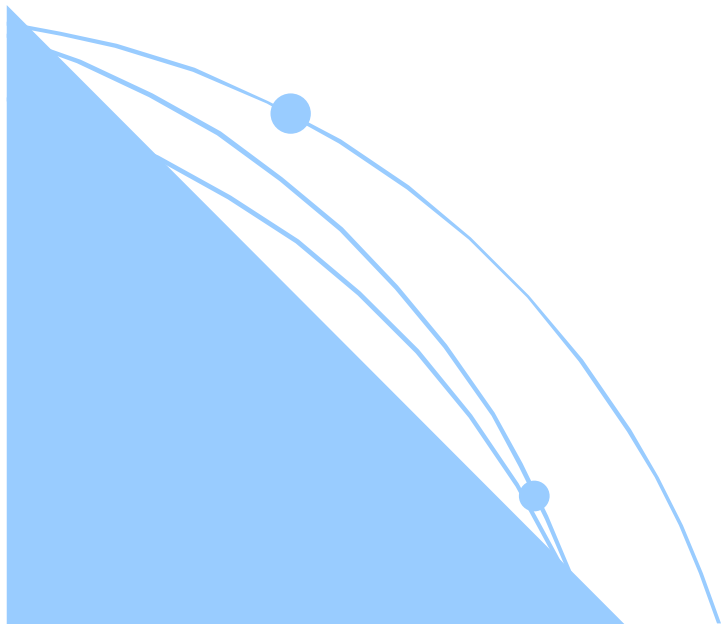
```
module shifter;
`define LEFT_SHIFT 1'b0
`define RIGHT_SHIFT 1'b1
reg [31:0] addr,left_addr,right_addr;
reg control;
always @(addr)
begin
    left_addr=shift(addr,`LEFT_SHIFT);
    right_addr=shift(addr,`RIGHT_SHIFT);
end
function [31:0] shift;
input [31:0] address;
input control;
begin
    shift=(control==`LEFT_SHIFT)?(address<<1):(address>>1);
end
endfunction
endmodule
```

6.2.5 自动（递归）函数

6.2.6 常量函数

6.2.7 带符号函数

略



6.3 关于使用任务和函数的小结

(1) 任务和函数都是用来对设计中多处使用的公共代码进行定义；使用任务和函数可以将模块分割成许多个可独立管理的子单元，增强了模块的可读性和可维护性；它们和C语言中的子程序起相同的作用。

(2) 任务可以具有任意多个输入、输入/输出 (**inout**) 和输出变量；在任务中可以使用延迟、事件和时序控制结构，在任务中可以调用其他的任务和函数。

(3) 可重入任务使用关键字**automatic**进行定义，它的每一次调用都对不同的地址空间进行操作。因此在被多次并发调用时，它仍然可以获得正确的结果。

(4) 函数只能有一个返回值，并且至少要有一个输入变量；在函数中不能使用延迟、事件和时序控制结构，但可以调用其他函数，不能调用任务。

(5) 当声明函数时，**Verilog**仿真器都会隐含地声明一个同名的寄存器变量，函数的返回值通过这个寄存器传递回调用处。

(6) 递归函数使用关键字**automatic**进行定义递归函数的每一次调用都拥有不同的地址空间。因此对这种函数的递归调用和并发调用可以得到正确的结果。

(7) 任务和函数都包含在设计层次之中，可以通过层次名对它们进行调用。

6.4 常用的系统任务

在Verilog HDL语言中，每个系统函数和任务前面都用一个标识符\$来确认。这些系统函数和任务提供了非常强大的功能。下面对一些常用的系统函数和任务逐一加以介绍。

6.4.1 \$display和\$write任务

格式：

```
$display(p1, p2, ..., pn);
```

```
$write(p1, p2, ..., pn);
```

这两个函数和系统任务是用来输出信息，即将参数p2~pn按参数p1给定的格式输出。参数p1通常称为“格式控制”，参数p2~pn通常称为“输出表列”。这两个任务的作用基本相同。

\$display自动地在输出后进行换行，**\$write**则不进行自动换行。如果想在`一行`里输出多个信息，可以使用**\$write**。在**\$display**和**\$write**中，其输出格式控制是用双引号括起来的字符串。

(1)格式说明——将输出的数据转换成指定的格式输出。

输出格式	说明
%h 或 %H	以十六进制数的形式输出
%d 或 %D	以十进制数的形式输出
%o 或 %O	以八进制数的形式输出
%b 或 %B	以二进制数的形式输出
%c 或 %C	以 ASCII 码字符的形式输出
%v 或 %V	输出网络型数据信号强度
%m 或 %M	输出等级层次的名字
%s 或 %S	以字符串的形式输出
%t 或 %T	以当前的时间格式输出
%e 或 %E	以指数的形式输出实型数
%f 或 %F	以十进制数的形式输出实型数
%g 或 %G	以较短的结果按指数或十进制输出实型数

(2)普通字符，即需要原样输出的字符。其中一些特殊的字符可以通过表6.2给出的转换序列来输出。表中的字符形式用于格式字符串参数中，用来显示特殊的字符。

换码序列	功能
\n	换行
\t	横向跳格
\\	反斜杠字符\
\"	双引号字符"
\o	1~3位八进制数代表的字符
%%	百分号%

在\$display和\$write的参数列表中，“输出列表”是需要输出的一些数据，可以是表达式。下面举几个例子加以说明。

【例6.17】

```
module disp;  
  initial  
  begin  
    $display("\\t% %\n\"123");  
  end  
endmodule;
```

输出结果为：

\ %
“S

从上面的这个例子中可以看到一些特殊字符的输出形式(八进制数123就是字符S)。

【例6.18】

```
module disp;
  reg [31: 0] rval;
  pulldown(pd);
  initial
    begin
      rval=101;
      $display("rval=%h hex %d decimal", rval, rval);
      $display("rval=%o octal %b binary", rval, rval);
      $display("rval has%c ASCII character value", rval);
      $display("current scope is %m");
      $display("%s is ascii value for 101", 101);
      $display("simulation time is %t", $time);
    end
endmodule
```

[illegible]

rval=00000065 hex 101 decimal

```
rval=00000000145 octal 000000000000000000000000000000001100101 binary
```

rval has e ascii character Value

current scope is disp

e is ascii value for 101

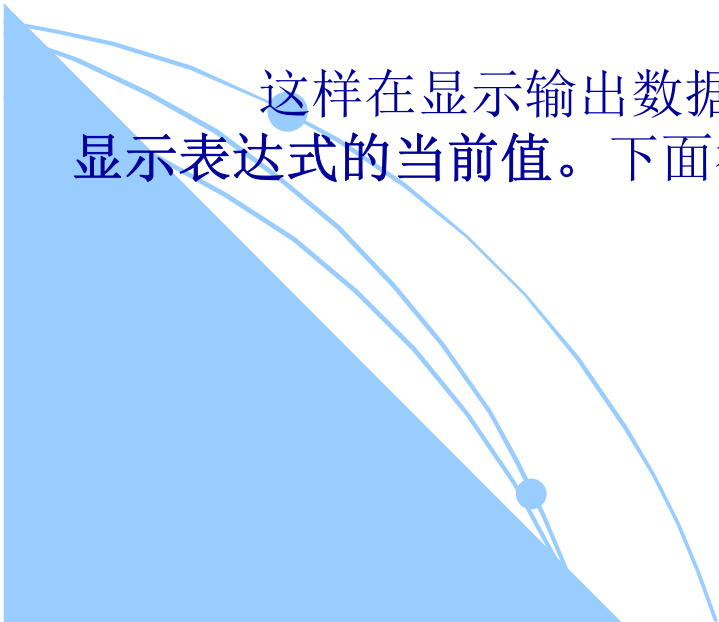
simulation time is 0

输出数据的显示宽度：

在`$display`中，输出列表中数据的显示宽度是自动按照输出格式进行调整的。这样，在显示输出数据时，在经过格式转换以后，总是用表达式的最大可能值所占的位数来显示表达式的当前值。在用十进制数格式输出时，输出结果前面的0值用空格来代替。对于其他进制，输出结果前面的0仍然显示出来。例如对于一个值的位宽为12位的表达式，如按照十六进制数输出，则输出结果占3个字符的位置；如按照十进制数输出，则输出结果占4个字符的位置。这是因为这个表达式的最大可能值为FFF(十六进制)、4095(十进制)。可以通过在%和表示进制的字符中间插入一个0，便自动调整显示输出数据宽度的方式。见下例：

```
$display( "d=%0h  a=%0h" , data, addr);
```

这样在显示输出数据时，在经过格式转换以后，总是用最少的位数来显示表达式的当前值。下面举例说明。



【例6.8】

```
module printval;  
  reg [11: 0] r1;  
  Initial  
  begin  
    r1=10;  
    $display("Printing with maximum size=%d=%h", r1, r1);  
    $display("Printing with minimum size=%0d=%0h", r1, r1);  
  end  
endmodule
```

输出结果为:

Printing with maximum size=10=00a:

Printing with minimum size =10=a;

如果输出列表中表达式的值包含有不确定的值或高阻值，其结果输出遵循以下规则。

(1) 在输出格式为十进制的情况下：

- 如果表达式值的所有位均为不定值，则输出结果为小写的x，
- 如果表达式值的所有位均为高阻值，则输出结果为小写的z；
- 如果表达式值的部分位为不定值，则输出结果为大写的X；
- 如果表达式值的部分位为高阻值，则输出结果为大写的Z。

(2) 在输出格式为十六进制和八进制的情况下：

● 每4位二进制数为一组代表1位十六进制数，每3位二进制数为一组代表1位八进制数。

● 如果表达式值相对应的某进制数的所有位均为不定值，则该位进制数的输出的结果为小写的x。

● 如果表达式值相对应的某进制数的所有位均为高阻值，则该位进制数的输出结果为小写的z。

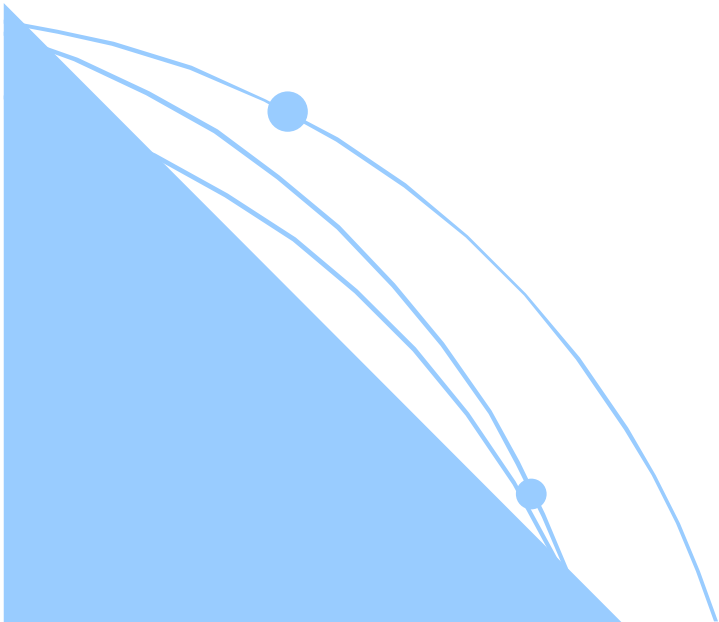
● 如果表达式值相对应的某进制数的部分位为不定值，则该位进制数输出结果为大写的X。

● 如果表达式值相对应的某进制数的部分位为高阻值，则该位进制数输出结果为大写的Z。

对于二进制输出格式，表达式值的每一位的输出结果为0、1、x、z。下面举例说明：

```
$display("%d", 1'bx);           x
$display("%h", 14'bx0_1010);    xxXa
$display("%h%o", 12'b001x_xx10_1x01, 12'b001_xxx_101_x01);
                                XXX 1x5X
```

因为\$write在输出时不换行，要注意它的使用。可以在\$write中加入换行符\n，以确保明确的输出显示格式。



6.4.4 选通显示

\$strobe系统任务

\$strobe系统任务与**\$display**使用方法相同。

它们之间的区别在于：

如果许多其他语句与**\$display**任务在同一个时间单位执行，那么这些语句与**\$display**任务的执行顺序是不确定的。

而**\$strobe**总是在同时刻的其他赋值语句执行完成后才执行。

【例6.22】

```
always @(posedge clock)
begin
a=b;
c=d;
end
always @(posedge clock)
    $strobe("Displaying a=%b,c=%b",a,c); //在a、b被赋值后才执行
```

小结

在本章中学习了Verilog语法中两种最重要的结构语句：**initial**和**always**；还学习了如何定义和使用任务与函数，及几个常用系统任务：**\$display**、**\$write**、**\$strobe**、**\$fopen**、**\$fclose**、**\$fdisplay**、**\$fmonitor**的用法。

需要掌握的是：

- (1)一个程序模块可以有多个**initial**和**always**过程块。
- (2)每个**initial**和**always**说明语句在仿真的一开始便同时立即开始运行。
- (3)**initial**语句在模块中只执行一次。
- (4)**always**语句则是不断地重复执行，直到仿真过程结束。
- (5)**always**语句后跟着的过程块是否运行，则要看它的触发条件是否满足，如满足则运行过程块一次，再次满足则再运行一次，循环往复直至仿真过程结束。
- (6)**always**的时间控制可以是沿触发也可以是电平触发，可以是单个信号也可以是多个信号，中间需要用关键字**or**连接。
- (7)沿触发的**always**块常常描述时序行为，如有限状态机。
- (8)电平触发的**always**块常常用来描述组合逻辑的行为。
- (9)**\$display**和**\$write**与C语言的对应语句的格式控制很类似，但有些不同，需要注意。**\$strobe**显示变量的时刻比**\$display**确定。
- (10)任务和函数在以后还要详细讲解，目前不必花费太多时间。
- (11)文件的读写与C语言很类似，可以参考本章的例子学习使用。编写复杂系统的测试模块，掌握文件的读写是非常有必要的。

思考题

- 1、怎样理解initial语句只执行一次的概念？
- 2、在initial语句引导的过程块中是否可以有循环语句？如果可以，是否与思考题1相矛盾？
- 3、怎样理解由always语句引导的过程块是不断活动的？
- 4、不断活动与不断执行有什么不同？
- 5、怎样理解沿触发和电平触发的不同？
- 6、是不是可以说沿触发是有间隔的，在一定的时间区间里只需要注意有限的点，而电平触发却需要注意无穷多个点？
- 7、沿触发的always块和电平触发的always块各表示什么类型的逻辑电路的行为？为什么？
- 8、简单叙述任务与函数的不同点。
- 9、简单叙述\$display、\$write和\$strobe的不同点。
- 10、简单叙述Verilog1364—2001版语法规定的电平敏感列表的简化写法。
- 11、如何在Verilog测试模块中，利用文件的读写产生预定格式的信号，并记录有测试价值的信号？

第七章 调用系统任务和常用编译预处理语句

7.1 系统任务\$monitor

7.2 时间度量系统函数\$time

7.3 系统任务\$finish

7.4 系统任务\$stop

7.5 系统任务\$readmemb和\$readmemh

7.6 系统任务\$random

7.7 编译预处理

小结

思考题

7.1 系统任务\$monitor

格式:

```
$monitor(p1, p2, ..., pn);  
$monitor;  
$monltoron;  
$monitoroff;
```

作用: 监控和输出参数列表中的表达式或变量值。其参数列表的规则和\$display一样。

当启动一个带有一个或多个参数的\$monitor任务时, 每当参数列表中变量或表达式的值发生变化时, 整个参数列表中变量或表达式的值都将输出显示。如果同一时刻, 两个或多个参数的值发生变化, 则在该时刻只输出显示一次。但在\$monitor中, 参数可以是\$time系统函数。这样, 参数列表中变量或表达式的值同时发生变化的时刻可以通过标明同一时刻的多行输出来显示。如:

```
$monitor($time, , "rxd=%b txd=%b" , rxd, txd);
```

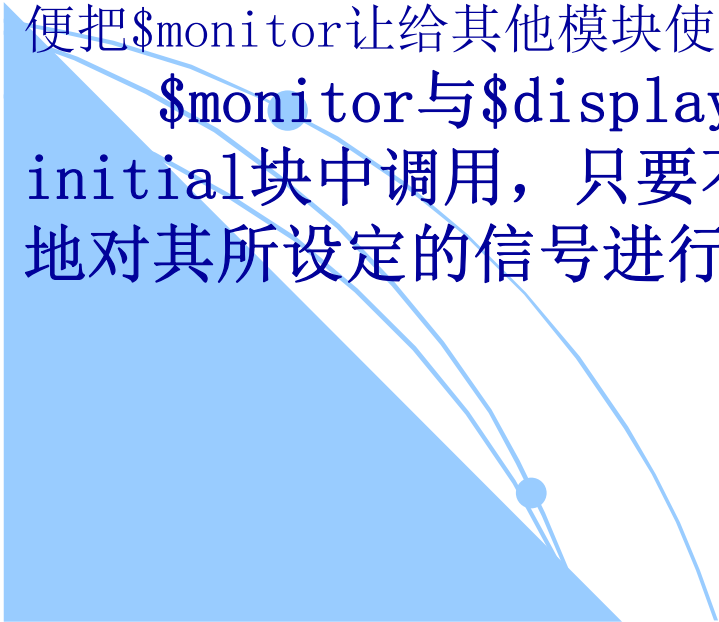
注: , , 表示空参数, 显示为空格。

\$monitoron和\$monitoroff这两个任务的作用是通过打开和关闭监控标志来控制、监控任务\$monitor的启动和停止。其中：

\$monitoroff 任务用于关闭监控标志，停止监控任务\$monitor;
\$monitoron则用于打开监控标志，启动监控任务\$monitor。

通常在调用\$monitoron来启动\$monitor时，不管\$monitor参数列表中的值是否发生变化，总是立刻输出显示当前时刻参数列表中的值，这样可在监控的初始时刻设定初始比较值。在默认情况下，控制标志在仿真的起始时刻就已经打开了。在多模块调试的情况下，许多模块中都调用了\$monitor，因为任何时刻只能有一个\$monitor起作用，因此需配合\$monitoron与\$monitoroff使用，把需要监视的模块用\$monitoron打开，在监视完毕后及时用\$monitoroff关闭，以便把\$monitor让给其他模块使用。

\$monitor与\$display的不同之处还在于\$monitor往往在initial块中调用，只要不调用\$monitoroff，\$monitor便不间断地对其所设定的信号进行监视。



7.2 时间度量系统函数\$time

在Verilog HDL中有两种类型的时间系统函数：`$time`和`$realtime`。用这两个时间系统函数可以得到当前的仿真时刻。

1. 系统函数\$time

`$time`可以返回一个以64位的整数来表示当前的仿真时刻值。该时刻是以模块的仿真时间尺度为基准的。下面举例说明。

[例7. 1]

```
`timescale 10 ns / 1ns
module test;
    reg set;
    parameter p=1.6;
    initial
    begin
        $monitor($time, , "set=" , set);
        #p set=0;
        #p set=1;
    end
endmodule
```

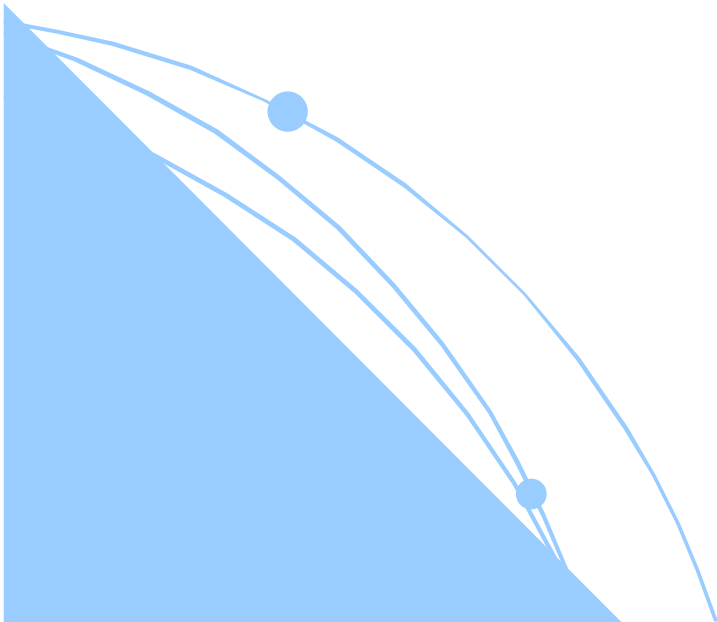
输出结果为:

```
0 set=x
2 set=0
3 set=1
```

在这个例子中，模块test想在时刻为16 ns时将寄存器set设置为0，在时刻为32ns时将寄存器设置set为1。但是由\$time记录的set变化时刻却和预想的不一样。这是由下面两个原因引起的：

(1)\$time显示时刻受时间尺度比例的影响。在例7. 1中，时间尺度是10 ns，因为\$time输出的时刻总是时间尺度的倍数，这样将16 ns和32ns输出为1. 6和3. 2。

(2)因为\$time总是输出整数，所以在输出经过尺度比例变换的数字输出时，要先进行取整。在例7. 1中，1. 6和3. 2经取整后为2和3输出。在这里，时间的精确度并不影响数字的取整。



2. \$realtime系统函数

\$realtime和\$time的作用是一样的，只是\$realtime返回的时间数字是一个实型数，该数字也是以时间尺度为基准的。下面举例说明：

[例7. 2]

```
`timescale 10ns / 1ns
module test;
    reg set;
    parameter p=1.55;
    initial
    begin
        $monitor($realtime, , “set=”, set);
        #p set=0;
        #p set=1;
    end
endmodule
```

```
# Compile of sim1.v was successful.
ModelSim> vsim work.test
# vsim work.test
# Loading work.test
VSIM 9> run
# 0 set=x
# 1.6 set=0
# 3.2 set=1
VSIM 10> |
```

从这个例子可以看出，\$realtime将仿真时刻经过尺度变换以后即输出，不需进行取整操作。所以，\$realtime返回的时刻是实型数。

7.3 系统任务\$finish

格式:

```
$finish;
```

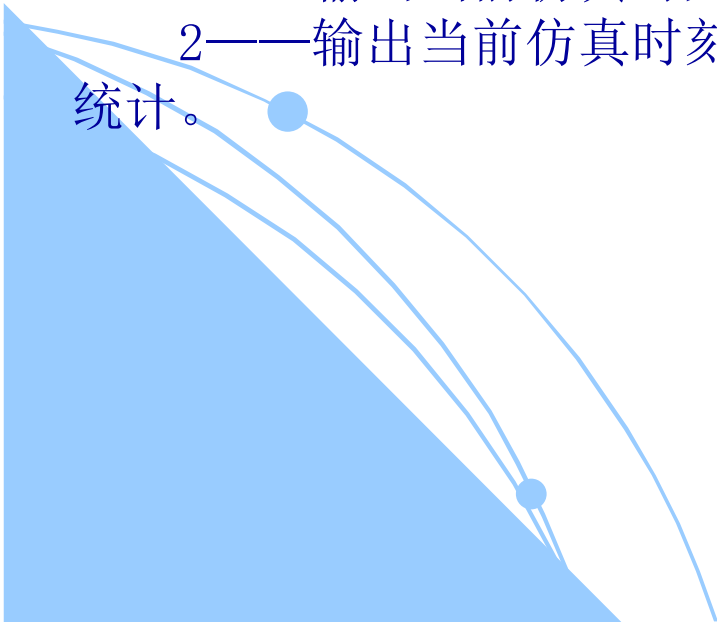
```
$finish(n);
```

系统任务\$finish的作用是退出仿真器，返回主操作系统，也就是结束仿真过程。任务\$finish可以带参数，根据参数的值输出不同的特征信息。如果不带参数，默认\$finish的参数值为1。下面给出了对于不同的参数值，系统输出的特征信息：

0——不输出任何信息；

1——输出当前仿真时刻和位置；

2——输出当前仿真时刻，位置和在仿真过程中所用memory及CPU时间的统计。



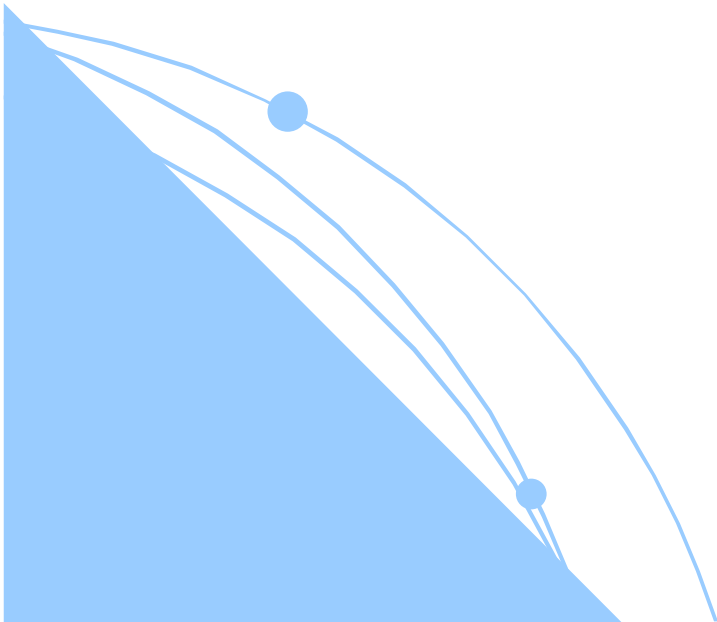
7.4 系统任务\$stop

格式:

`$stop;`

`$stop(n);`

\$stop任务的作用是把**EDA**工具(例如仿真器)置成暂停模式,在仿真环境下给出一个交互式的命令提示符,将控制权交给用户。这个任务可以带有参数表达式。根据参数值(0, 1或2)的不同,输出不同的信息。参数值越大,输出的信息越多。



7.5 系统任务\$readmemb和\$readmemh

在Verilog HDL程序中有两个系统任务\$readmemb和\$readmemh用来从文件中读取数据到存储器中。这两个系统任务可以在仿真的任何时刻被执行，其使用格式共有以下6种；

- (1) \$readmemb(“<数据文件名>”，<存储器名>);
- (2) \$readmemb(“<数据文件名>”，<存储器名>，<起始地址>);
- (3) \$readmemb(“<数据文件名>”，<存储器名>，<起始地址>，<结束地址>);
- (4) \$readmemh(“<数据文件名>”，<存储器名>);
- (5) \$readmemh(“<数据文件名>”，<存储器名>，<起始地址>);
- (6) \$readmemh(“<数据文件名>”，<存储器名>，<起始地址>，<结束地址>))。

在这两个系统任务中，被读取的数据文件的内容只能包含：空白位置（空格，换行，制表格(tab)和form-feeds进纸）、注释行（//形式的和/*...*/形式的都允许）、二进制或十六进制的数字。

数字中不能包含位宽说明和格式说明，对于\$readmemb系统任务，每个数字必须是二进制数字，对于\$readmemh系统任务，每个数字必须是十六进制数字。数字中不定值x或X，高阻值z或Z，和下画线(—)的使用方法及代表的意义与一般Verilog HDL程序中的用法及意义是一样的。另外，数字必须用空白位置或注释行来分隔开。

在下面的讨论中，地址一词指对存储器(memory)建模的数组的寻址指针。当数据文件被读取时，每一个被读取的数字都被存放到地址连续的存储器单元中去。存储器单元的存放地址范围由系统任务声明语句中的起始地址和结束地址来说明，每个数据的存放地址在数据文件中进行说明。

当地址出现在数据文件中，其格式为字符“@”后跟上十六进制数。如：

@hh...h

对于这个十六进制的地址数中，允许大写和小写的数字。在字符“@”和数字之间不允许存在空白位置。可以在数据文件里出现多个地址。当系统任务遇到一个地址说明时，系统任务将该地址后的数据存放到存储器中相应的地址单元中去。

例如：文件路径 “c:\memdata.dat”, 则有：

reg [7:0] mema [0:255];

.....

\$readmemb("c:\memdata.dat", mema, 0, 255);

对于上面6种系统任务格式，需补充说明以下5点：

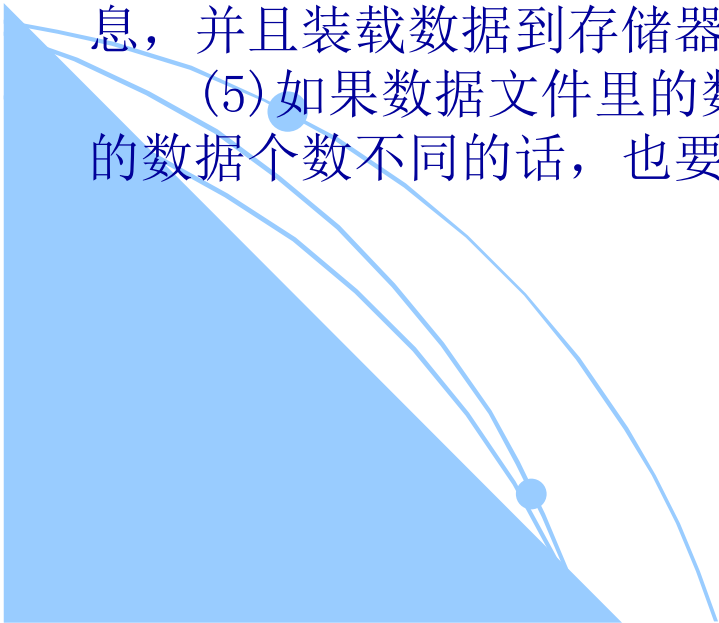
(1) 如果系统任务声明语句中和数据文件里都没有进行地址说明，则缺省的存放起始地址为该存储器定义语句中的起始地址。数据文件里的数据被连续存放到该存储器中，直到该存储器单元存满为止或数据文件里的数据全部读完。

(2) 如果系统任务中说明了存放的起始地址，没有说明存放的结束地址，则数据从起始地址开始存放，存放到该存储器定义语句中的结束地址为止。

(3) 如果在系统任务声明语句中，起始地址和结束地址都进行了说明，则数据文件里的数据按该起始地址开始存放到存储器单元中，直到该结束地址，而不考虑该存储器的定义语句中的起始地址和结束地址。

(4) 如果地址信息在系统任务和数据文件里都进行了说明，那么数据文件里的地址必须在系统任务中地址参数声明的范围之内。否则将提示错误信息，并且装载数据到存储器中的操作被中断。

(5) 如果数据文件里的数据个数和系统任务中起始地址及结束地址暗示的数据个数不同的话，也要提示错误信息。



下面举例说明。

先定义一个有256个地址的字节存储器mem:

```
reg[7: 0] mem[1: 256];
```

下面给出的系统任务以各自不同的方式装载数据到存储器mem中。

```
initial $readmemh("mem.data", mem);
```

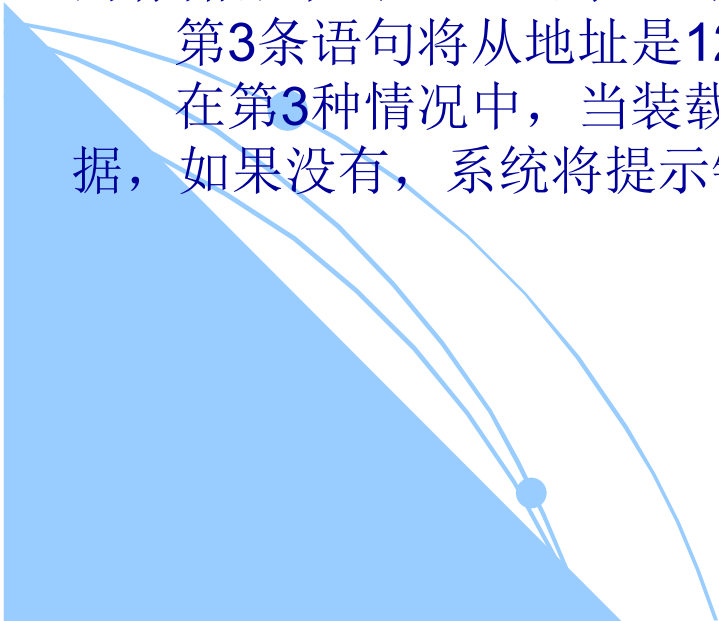
```
initial $readmemh("mem.data", mem, 16);
```

```
initial $readmemh("mem.data", mem, 128, 1);
```

第1条语句在仿真时刻为0时，将数据装载到以地址为1的存储器单元为起始存放单元的存储器中去；

第2条语句将装载数据到以单元地址是16的存储器单元为起始存放单元的存储器中去，一直到地址是256的单元为止；

第3条语句将从地址是128的单元开始装载数据，一直到地址为1的单元。在第3种情况中，当装载完毕，系统要检查在数据文件里是否有128个数据，如果没有，系统将提示错误信息。



7.6 系统任务random

这个系统函数提供了一个产生随机数的手段。当函数被调用时返回一个32位的随机数。它是一个带符号的整型数。

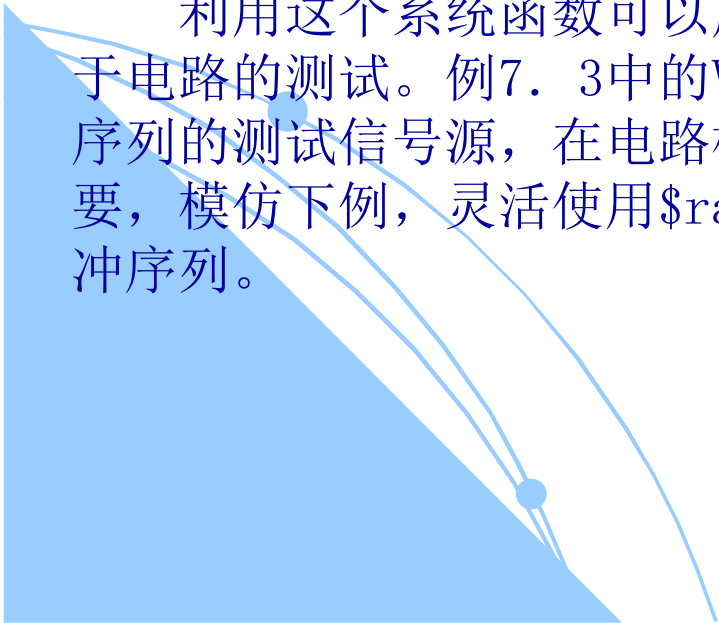
\$random一般的用法是： $\$random\%b$ ，其中 $b>0$ 。它给出了一个范围在 $(-b+1): (b-1)$ 中的随机数。下面给出一个产生随机数的例子：

```
reg [23: 0]rand;  
rand=$random%60;
```

上面的例子给出了一个范围在-59到59之间的随机数，下面的例子通过位拼接操作产生一个值在0~59之间的数。

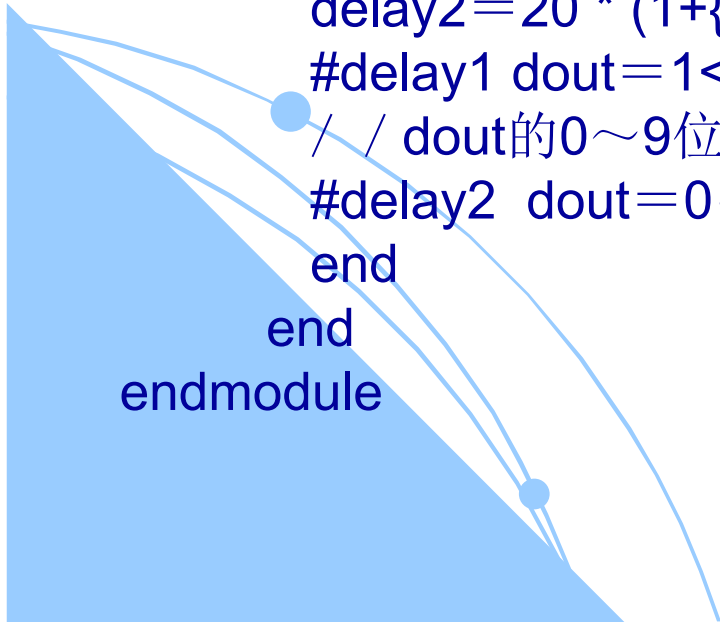
```
reg[23: 0]rand;  
rand={ $random } %60;
```

利用这个系统函数可以产生随机脉冲序列或宽度随机的脉冲序列，以用于电路的测试。例7. 3中的Verilog HDL模块可以产生宽度随机的随机脉冲序列的测试信号源，在电路模块的设计仿真时非常有用。可以根据测试的需要，模仿下例，灵活使用\$random系统函数编制出与实际情况类似的随机脉冲序列。



[例7. 3]

```
`timescale 1ns / 1ns
module random_pulse(dout);
output [9: 0] dout;
reg [9: 0] dout;
integer delay1, delay2, k;
initial
begin
#10 dout=0;
for(k=0; k<100; k=k+1)
begin
delay1=20 * ({$random}%6);          // delay1在0到100 ns间变化
delay2=20 * (1+{$random}%3);        // delay2在20到60 ns间变化
#delay1 dout=1<<({$random}%10);
// dout的0~9位中随机出现1，并出现的时间在0~100 ns间变化
#delay2 dout=0;                      // 脉冲的宽度在在20到60 ns间变化
end
end
endmodule
```



7.7 编译预处理

7.7.1 宏定义'define

用一个指定的标识符(即名字)来代表一个字符串，它的一般形式为：

`' define标识符(宏名) 字符串(宏内容)`

如：

`' define signal string`

它的作用是指定用标识符**signal**来代替**string**这个字符串，在编译预处理时，把程序中在该命令以后所有的**signal**都替换成**string**。这种方法使用户能以一个简单的名字代替一个长的字符串，也可以用一个有含义的名字来代替没有含义的数字和符号，因此把这个标识符(名字)称为“宏名”。在编译预处理时将宏名替换成字符串的过程称为“宏展开”。`'define`是宏定义命令。

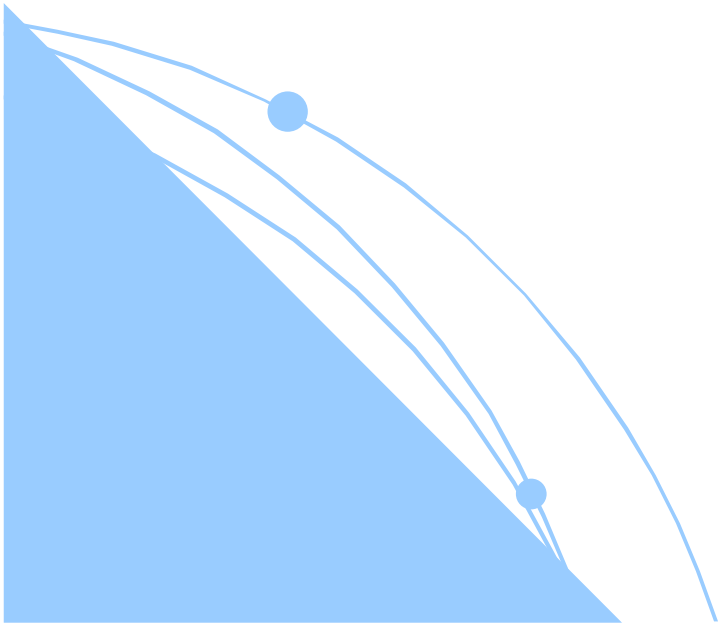
[例7.4]

```
'define WORDSIZE 8
module
reg[1: `WORDSIZE] data;    // 相当于定义reg [1: 8] data;
```

7. 7. 2 “文件包含” 处理'include

所谓“文件包含”处理是一个源文件可以将另外一个源文件的全部内容包含进来，即将另外的文件包含到本文件之中。Verilog HDL语言提供了'include命令来实现“文件包含”的操作。其一般形式为：

'include“文件名”



7. 7. 3 时间尺度timescale

``timescale`命令用来说明跟在该命令后的模块的时间单位和时间精度。使用``timescale`命令可以在同一个设计里包含采用了不同的时间单位的模块。例如，一个设计中包含了两个模块，其中一个模块的时间延迟单位为ns，另一个模块的时间延迟单位为ps。EDA工具仍然可以对这个设计进行仿真测试。

``timescale`命令的格式如下；

``timescale<时间单位> / <时间精度>`

在这条命令中，时间单位参量是用来定义模块中仿真时间和延迟时间的基准单位的。时间精度参量是用来声明该模块的仿真时间的精确程度的，该参量被用来对延迟时间值进行取整操作(仿真前)，因此该参量又可以被称为取整精度。如果在同一个程序设计里，存在多个``timescale`命令，则用最小的时间精度值来决定仿真的时间单位。另外时间精度至少要和时间单位一样精确，时间精度值不能大于时间单位值。

在``timescale`命令中，用于说明时间单位和时间精度参量值的数字必须是整数，其有效数字为1, 10, 100，单位为s, ms, us, ns, ps, fs。这几种单位的意义说明如表1. 7. 1所列。

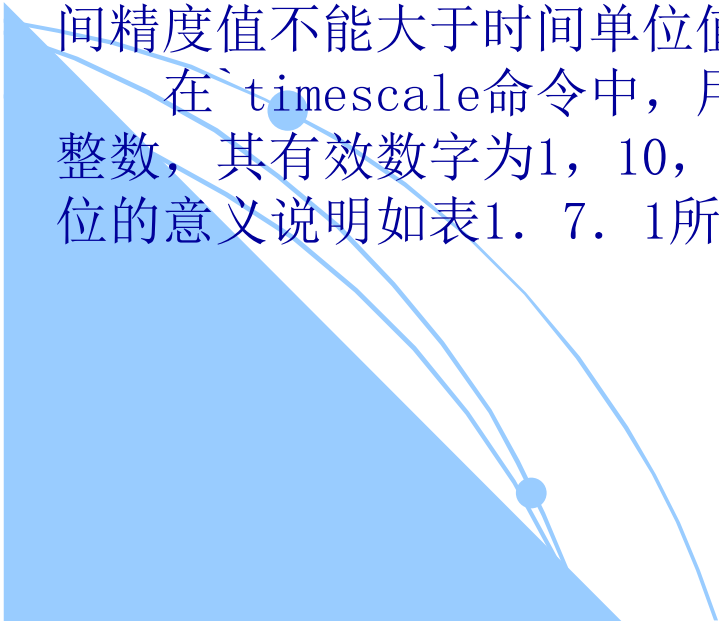


表7.1 常用时间单位与定义

时间单位	定义
s	秒 (1s)
ms	千分之一秒 (10^{-3}s)
μs	百万分之一秒 (10^{-6}s)
ns	十亿分之一秒 (10^{-9}s)
ps	万亿分之一秒 (10^{-12}s)
fs	千万亿分之一秒 (10^{-15}s)

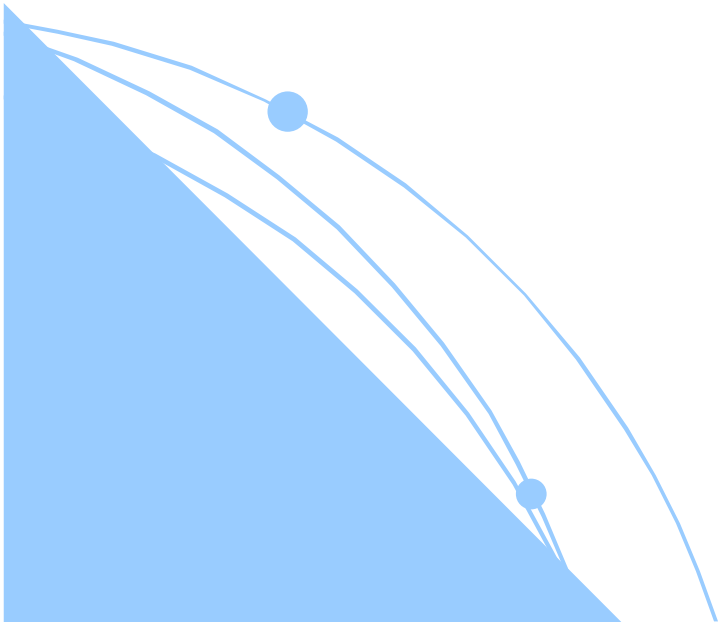
下面举例说明`timescale命令的用法。

[例7. 9]

`timescale 1ns / 1ps: 在这个命令之后，模块中所有的时间值都表示是1 ns的整数倍。这是因为在‘timescale命令中，定义了时间单位为1 ns。模块中的延迟时间可表达为带3位小数的实型数，因为’ timescale命令定义时间精度为1 Ps。

[例7. 10]

`timescale 10us / 100ns: 在这个例子中，`timescale命令定义后，模块中的时间值均为10us的整数倍。因为`timesacle命令定义的时间单位是10us 。延迟时间的最小分辨率为十分之一us(100 ns)，即延迟时间可表达为带一位小数的实型数。



[例7. 11]

```
`timescale 10ns / 1ns
module test;
  reg set;
  parameter d=1.55;
  initial
  begin
    #d set=0;
    #d set=1;
  end
endmodule
```

在这个例子中，`timescale命令定义了模块test的时间单位为10 ns、时间精度为1 ns。因此在模块test中，所有的时间值应为10 ns的整数倍，且以1ns为时间精度。这样经过取整操作，存在参数d中的延迟时间实际是16 ns(即 $1.6 \times 10 \text{ ns}$)，这意味着在仿真时刻为16 ns时寄存器set被赋值为0，在仿真时刻为32ns时寄存器set被赋值为1。仿真时刻值是按照以下的步骤来计算的。

(1) 根据时间精度，参数d值被从1.55取整为1.6。

(2) 因为时间单位是10 ns，时间精度是1 ns，所以延迟时间#d为时间单位的整数倍，即为16 ns。

(3) EDA工具预定在仿真时刻为16 ns的时候给寄存器set赋值为0(即语句#d set=0; 执行时刻)，在仿真时刻为32 ns的时候给寄存器set赋值为1(即语句#d set=1; 执行时刻)。

7.7.4 条件编译命令`ifdef、`else、`endif

一般情况下，Verilog HDL源程序中所有的行都将参加编译。但是有时希望对其中的一部分内容只有在满足条件时才进行编译，也就是对一部分内容指定编译的条件，这就是“条件编译”。有时，希望当满足条件时对一组语句进行编译，而当条件不满足是则编译另一部分。

条件编译命令有以下几种形式。

(1)`ifdef 宏名(标识符)：

程序段1

`else

程序段2

`endif

它的作用是当宏名已经被定义过(用`define命令定义)，则对程序段1进行编译，程序段2将被忽略；否则编译程序段2，程序段1被忽略。

(2)`ifndef 宏名(标识符)：

程序段1

`endif

这里的“宏名”是一个Verilog HDL的标识符，“程序段”可以是Verilog语句组，也可以是命令行。这些命令可以出现在源程序的任何地方。注意：被忽略掉不进行编译的程序段部分也要符合Verilog程序的语法规则。

通常在Verilog HDL程序中用到`ifdef、`else、`endif编译命令的情况有以下几种：

- 选择一个模块的不同代表部分；
- 选择不同的时序或结构信息；
- 对不同的EDA工具，选择不同的激励。

7.8 小结

在本讲中我们学习了Verilog语法中几种最常用的调试和查错系统任务以及编写实用模块时常用的编译预处理语句。这些语句是非常有用的，可以说是编写调试模块所必须掌握的。只有掌握它们才能够编写出验证复杂数字系统所必须的严格、完整的测试模块，从而设计出有实用价值的复杂逻辑电路系统。需要注意的有以下几点：

(1) 在多模块调试的情况下，\$monitor需配合\$monitoron与\$monitoroff使用。

(2) \$monitor与\$display的不同之处在于\$monitor是连续监视数据的变化，因而往往只要在测试模块的initial块中调用一次就可以监视被测试模块的所有感兴趣的信号，不需要也不能在always过程块中调用\$monitor。(3) \$time常用在\$monitor中，用来做时间标记。

(4) \$stop和\$finish常用在测试模块的initial模块中，配合时间延迟用来控制仿真的持续时间。

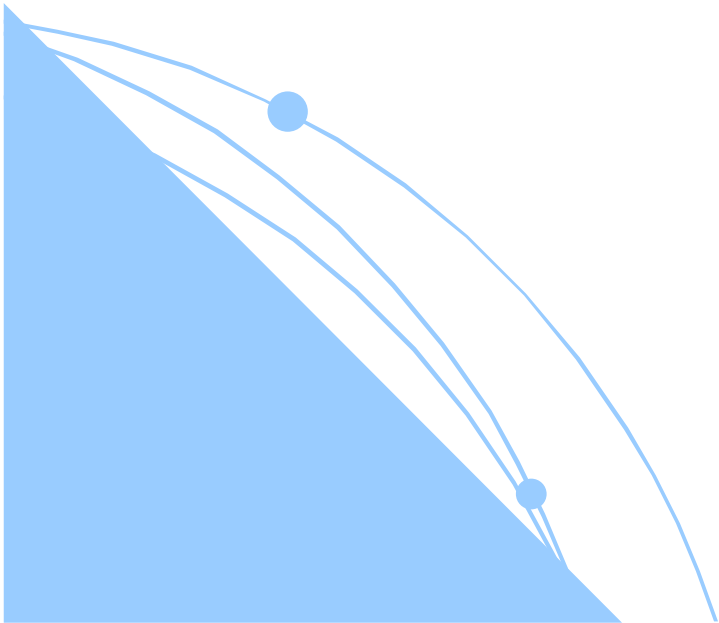
(5) \$random在编写测试程序是非常有用的，可以用来产生边沿不稳定的波形，和随机出现的脉冲。正确地使用它能有效地发现实际设计中存在的问题。

(6) \$readmem在编写测试程序也是非常有用的，可以用来生成给定的复杂数据流。复杂数据可以用C语言产生，存在文件中。用\$readmem取出存入存储器，再按节拍输出，这在验证算法逻辑电路时特别有用。

(7)在用`timescale时需要注意的是，当多个带不同timescale定义的模块包含在一起时只有最后一个才起作用。所以属于一个项目，但`timescale定义不同的多个模块最好分开编译，不要包含在一起编译，以免把时间单位搞混。

(8)宏定义字符串引用时，不要忘记，要用“`”引导。这与C语言不同，C语言直接引用就行，但Verilog必须用“`”引导。

(9)include等编译预处理也必须用“`”引导，而不是与C语言一样用“#”引导或不需要引导符。

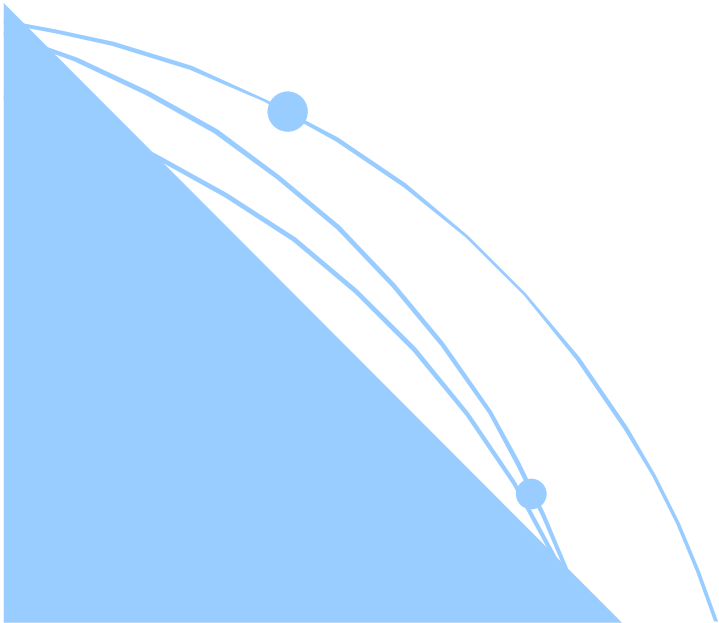


思考题

- 1、为什么在多模块调试的情况下\$monitor需要配合\$monitoron和\$monitoroff来工作？
- 2、请用\$random配合求模运算编写：
 - (1) 用于测试的跳变沿抖动为周期1/10的时钟波形。
 - (2) 随机出现的脉宽随机的窄脉冲。
- 3、Verilog的编译预处理与C语言的编译预处理有什么不同？
- 4、请仔细阐述`timescale编译预处理的作用？
- 5、不同`timescale定义的多模块仿真测试时需要注意什么？
- 6、为什么说系统任务\$readmem可以用来产生用于算法验证的及其复杂的测试用数据流？
- 7、为什么说熟练的使用条件编译命令可以使源代码有更大的灵活性，可以适用于不同的实现对象，如不同工艺的ASIC或速度规模不同的FPGA或CPLD，从而为软核的商品化创造条件？

第八章 语法概念总复习练习

略



结束！

