8　硬件描述语言Verilog HDL

1 引言
2 Verilog HDL基本结构
3 数据类型及常量、变量
4 运算符及表达式
5 语句
6 赋值语句和块语句
7　条件语句
8 循环语句
9 结构说明语句
10 编译预处理语句
11 语句的顺序执行与并行执行

什么是硬件描述语言HDL

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

为什么使用HDL

- 使用HDL描述设计具有下列优点：
  - 设计在高层次进行，与具体实现无关
  - 设计开发更加容易
  - 早在设计期间就能发现问题
  - 能够自动的将高级描述映射到具体工艺实现
  - 在具体实现时才做出某些决定
- HDL具有更大的灵活性
  - 可重用
  - 可以选择工具及生产厂
- HDL能够利用先进的软件
  - 更快的输入
  - 易于管理

1 引言

内容概要

一、什么是Verilog HDL

二、Verilog HDL的发展历史

三、不同层次的Verilog HDL抽象

四、Verilog HDL的特点

1 引言

一、什么是Verilog HDL

- **Verilog HDL是一种用于数字逻辑电路设计的硬件描述语言（Hradware Description Language），可以用来进行数字电路的仿真验证、时序分析、逻辑综合。**
  - 用Verilog HDL描述的电路设计就是该电路的Verilog HDL模型。
  - Verilog HDL既是一种行为描述语言也是一种结构描述语言。

- 既可以用电路的功能描述，也可以用元器件及其之间的连接来建立Verilog HDL模型。

1 引言

二、Verilog HDL的发展历史

- **1985年，由GDA（GateWay Design Automation）公司的Phil Moorby首创；**
- **1989年，Cadence公司收购了GDA公司；**
- **1990年， Cadence公司公开发表Verilog HDL；**
- **1995年， IEEE制定并公开发表Verilog HDL1564-1995标准；**
- **1999年，模拟和数字电路都适用的Verilog标准公开发表**
- 2001年3月IEEE正式批准了Verilog-2001标准(IEEE1364-2001)，与Verilog-1995相比有提高。

三、不同层次的Verilog HDL抽象

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

四、Verilog HDL的特点

与C语言非常相似！

- **语法结构上的主要特点：**
  - 形式化地表示电路的行为和结构；
  - 借用C语言的结构和语句；
  - 可在多个层次上对所设计的系统加以描述，语言对设计规模不加任何限制；
  - 具有混合建模能力：一个设计中的各子模块可用不同级别的抽象模型来描述；
  - 基本逻辑门、开关级结构模型均内置于语言中，可直接调用；
  - 易创建用户定义原语（UDP，User Designed Primitive）。
- **易学易用，功能强**

2 Verilog HDL基本结构

内容概要

一、简单的Verilog HDL例子
二、Verilog HDL模块的结构
三、逻辑功能定义
四、关键字
五、标识符
六、编写Verilog HDL源代码的标准

2 Verilog HDL基本结构

一、简单的Verilog HDL例子

［例1］8位全加器

模块名(文件名)

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

端口定义

I/O说明

功能描述

- 整个Verilog HDL程序嵌套在module和endmodule声明语句中。
- 每条语句相对module和endmodule最好缩进2格或4格！
- // …… 表示注释部分，一般只占据一行。对编译不起作用！

单行注释符

2 Verilog HDL基本结构

［例2］ 8位计数器

```
module  counter8 ( out,cout,data,load, cin,clk );
    output [7:0] out;
    output  cout;
    input [7:0] data;
    input load, cin,clk ;
    reg[7:0] out;
    always @(posedge clk)
      begin
        if(load)
          out = data;                  // 同步预置数据
        else
           out = out + 1 +  cin;       // 加1计数
      end
  assign cout = &out & cin;   //若out为8'hFF，cin为1，则cout为1
endmodule
```

端口定义

I/O说明

信号类型声明

功能描述

缩减运算符

位运算符

2 Verilog HDL基本结构

[例3] 2位比较器

```
module  compare2 ( equal,a,b);
    output equal;
    input [1:0] a,b;
    assign equal = ( a = = b ) ? 1:0;
   / * 如果a等于b，则equal 为1，否则为0 * /
endmodule
```

条件运算符

连续赋值语句

- /* ..... */内表示注释部分，一般可占据多行。对编译不起作用！

多行注释符

2 Verilog HDL基本结构

门元件例化

[例4] 三态驱动器

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

门元件关键字

例化元件名

bufif1的真值表

Inputs		Output
IN	ENABLE	OUT
X	0	Z
1	1	1
0	1	0

门元件例化——程序通过调用一个在Verilog语言库中现存的实例门元件来实现某逻辑门功能。

2 Verilog HDL基本结构

模块元件例化

[例5] 三态驱动器

顶层模块

子模块名

例化元件名

```
module  trist1(out,in,enable);
     output  out;
     input   in, enable;
     mytri  tri_inst(out,in,enable);
endmodule
```

子模块

```
module mytri(out,in,enable);
     output  out;
     input   in, enable;
     assign out = enable? in:'bz;
      /* 如果enable为1，则out = in，否则为高阻态 */
endmodule
```

模块元件例化——顶层模块（trist1）调用由某子模块（mytri）定义的实例元件（tri_inst）来实现某功能。

2 Verilog HDL基本结构

总 结

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

2 Verilog HDL基本结构

二、Verilog HDL模块的结构

- Verilog的基本设计单元是“模块(block)”。
- Verilog 模块的结构由在module和endmodule关键词之间的4个主要部分组成：

1. 模块声明
2. 端口定义
3. 信号类型声明
4. 功能描述

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

Verilog模块结构

Verilog模块结构完全嵌在module和endmodule之间关键字之间，每个Verilog程序包括4个主要部分：
模块声明、
端口定义、
信号类型说明、
功能描述。

1)模块声明

模块声明包括模块名字和模块输入、输出端口列表。其格式如下：

module 模块名 (端口1，端口2，端口3……);

…….

endmodule。

2)端口(Port)定义

端口是模块与外界或其他模块连接和通信的信号线，

有三种类型的端口:输入端口(input)、输出端口(output)和输入/输出端口(inout).

对模块的输入、输出端口要明确说明，其格式为：

Input 端口1，端口2，….端口N; //输入端口

output 端口1，端口2，….端口N; //输出端口

Inout 端口1，端口2，….端口N; //双向端口

3)信号类型声明

对端口的输入输出要明确说明外，还要进行信号数据类型的定义。Verilog语言提供了各种信号类型，分别模拟实际电路中的各种物理连接和物理实体

常用的数据类型包括连线型(wire)、寄存器型(reg)两种。如果信号的数据类型没有定义，则综合器将其默认为wire型。

注：输入和双向端口不能声明为寄存器型

定义信号数据类型举例：

```
reg    cout;          //定义信号cout数据类型为reg型
wire   a,b,c;         //定义信号a,b,c数据类型为wire(连线）型
```

2 Verilog HDL基本结构

4)逻辑功能定义

- 在Verilog模块中有多种方法可以描述电路的逻辑功能：

(1) 用assign 语句 连续赋值语句

常用于描述组合逻辑

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

(2) 用元件例化（instantiate）

```
and myand5( f,a,b,c);
```

门元件例化

门元件关键字

例化元件名

- 注1：元件例化即是调用Verilog HDL提供的元件；
- 注2：元件例化包括门元件例化和模块元件例化；
- 注3：每个实例元件的名字必须唯一！以避免与其它调用元件的实例相混淆。
- 注4：例化元件名也可以省略！

模块元件例化

门元件例化

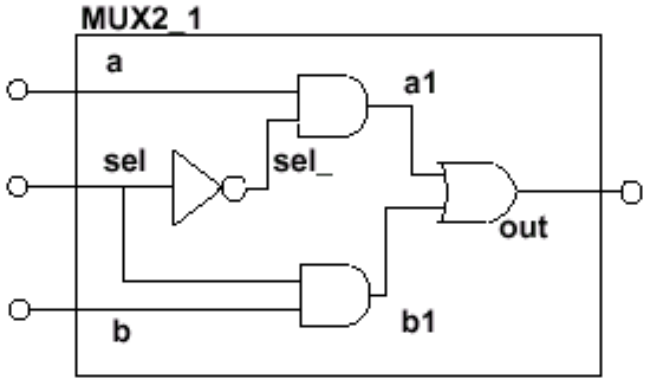
调用元件方法类似于在电路图输入方式下调入图形符号来完成设计，这种方法侧重于电路的结构描述。在Verilog语言中，可通过调用如下元件的方式来描述电路的结构：

1.调用Verilog内置门元件(门级结构描述)

2.调用开关级元件(开关级结构描述)

3.用户定义元件UDP(也在门级)

门元件例化　门级描述2选1




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

已定义的Verilog基本单元的实例

- **a, b, sel**是输入端口，**out**是输出端口。所有信号通过这些端口从模块输入/输出。

一个模块可以通过模块名及端口说明使用另一个模块。实例化模块时不需要知道其实现细节。这正是自上而下设计方法的一个重要特点。模块的实现可以是行为级也可以是门级，但并不影响高层次模块对它的使用。

模块实例化(module instantiation)

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

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

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

没有连接时通常会产生警告

名称映射的语法：

.内部信号（外部信号）

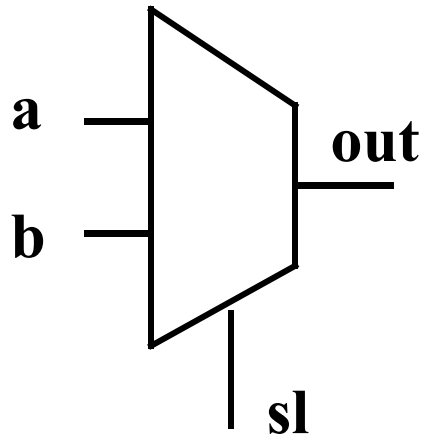
2 Verilog HDL基本结构

(3) 用 “always” 块语句

结构说明语句

例 多路选择器---- 2选1

```
module muxtwo(out,a,b,sl);
  input  a,b,sl;
  output  out;
  reg  out;
  always @ (sl or a or b)
        if(!sl)  out=a;
           else  out=b;
endmodule
```



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

2 Verilog HDL基本结构

❖ Verilog HDL模块的模板（仅考虑用于逻辑综合的部分）

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

2 Verilog HDL基本结构

```
// （3）元件例化
   < module_name > < instance_name > (<port_list>);       // 模块元件例化
   <gate_type_keyword> < instance_name > (<port_list>); // 门元件例化
endmodule
```

例化元件名
也可以省略！

Verilog HDL描述方式

一、Verilog HDL的门级描述

二、Verilog HDL的行为级描述

Verilog模型可以是实际电路不同级别的抽象。所谓不同的抽象级别，实际上是指同一个物理电路，可以在不同的层次上用Verilog语言来描述它，如果只从行为和功能的角度来描述某一电路模块，就称为行为模块；如果从电路结构的角度来描述该电路模块，就称为结构模块。抽象的级别和它们对应的模块类型常可以分为以下5种

(1) 系统级(system)
(2) 算法级(algorithmic)
(3) RTL级(RegisterTransferLevel):
(4) 门级(gate-level):
(5) 开关级(switch-level)

系统级、算法级和RTL级是属于行为级的，门级是属于结构级的。

抽象级(Levels of Abstraction)

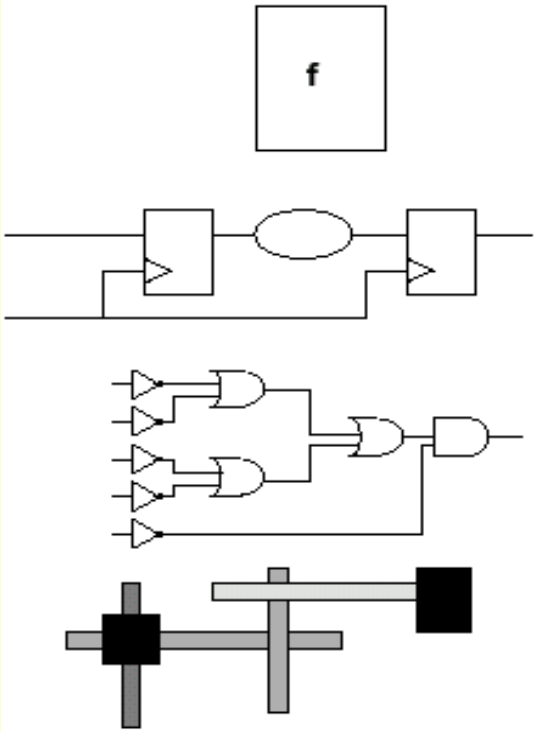
- Verilog既是一种行为描述的语言也是一种结构描述语言。Verilog模型可以是实际电路的不同级别的抽象。这些抽象的级别包括：

系统说明
-设计文档/算法描述

RTL/功能级
-Verilog

门级/结构级
-Verilog

版图/物理级
-几何图形



行为综合
↓
综合前仿真
↓
逻辑综合
↓
综合后仿真
↓
版图

抽象级(Levels of Abstraction)

Verilog可以在三种抽象级上进行描述

- 行为级
  - 用功能块之间的数据流对系统进行描述
  - 在需要时在函数块之间进行调度赋值。
- **RTL级/功能级**
  - 用功能块内部或功能块之间的数据流和控制信号描述系统
  - 基于一个已定义的时钟的周期来定义系统模型
- 结构级/门级
  - 用基本单元(primitive)或低层元件(component)的连接来描述系统以得到更高的精确性，特别是时序方面。
  - 在综合时用特定工艺和低层元件将RTL描述映射到门级网表

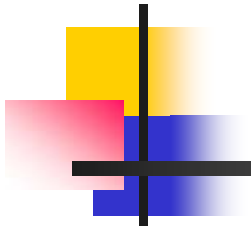
- 设计工程师在不同的设计阶段采用不同的抽象级
  - 首先在行为级描述各功能块，以降低描述难度，提高仿真速度。
  - 在综合前将各功能模块进行**RTL**级描述。
  - 用于综合的库中的大多数单元采用结构级描述。在结构级描述部分将对结构级(门级)描述进行更详细的说明。
- **Verilog**还有一定的晶体管级描述能力及算法级描述能力

硬件描述语言的可综合性问题

所谓逻辑综合就其实质而言是设计流程中的一个阶段，在这一阶段中将较高级抽象层次的描述自动地转换成较低层次描述。就现在达到的水平而言，所谓逻辑综合就是通过综合器把HDL程序转换成标准的门级结构网表，而并非真实具体的电路。而真实具体的电路还需要利用ASIC和FPGA制造厂商的布局布线工具根据综合后生成的标准的门级结构网表来产生。为了能转换成标准的门级结构网表，HDL程序的编写必须符合特定综合器所要求的风格。由于门级结构、RTL级的HDL程序的综合是很成熟的技术，所有的综合器都支持这两个级别HDL程序的综合。

对于数字系统的逻辑设计工程师而言，熟练地掌握门级、RTL级、算法级、系统级是非常重要的。而对于电路基本部件（如门、缓冲器、驱动器等）库的设计者而言，则需要掌握用户自定义源语元件（UDP）和开关级的描述。

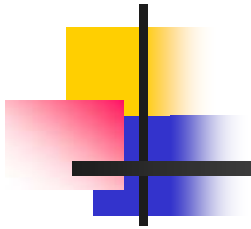
一个复杂电路的完整Verilog HDL模型是由若干个Verilog HDL模块构成的，每一个模块又可以由若干个子模块构成。这些模块可以分别用不同抽象级别的Verilog HDL描述，在一个模块中也可以有多种级别的描述。利用Verilog HDL语言结构所提供的这种功能就可以构造一个模块间的清晰层次结构来描述极其复杂的大型设计。



Verilog HDL描述方式

模块内具体逻辑行为的描述方式又称为建模方式。根据设计的不同要求，每个模块内部具体的逻辑行为描述方式可以分为四个不同的抽象级别。

- 对于外部来说，看不到逻辑行为的具体实现方式。因此，模块的内部具体逻辑行为描述相对于外部其它模块来说是不可见的。
- 改变一个模块内部逻辑行为的描述方式，并不会影响该模块与其它模块的连接关系。



Verilog HDL描述方式

Verilog HDL提供了下面四种方式描述具体的逻辑行为：

- 行为级描述方式
- 数据流描述方式
- 结构级描述方式
- 开关级描述方式

行为级描述

Verilog HDL的行为级描述是最能体现电子设计自动化风格的硬件描述方式

- 它既可以描述简单的逻辑门，也可以描述复杂的数字系统乃至微处理器。
- 既可以描述组合逻辑电路，也可以描述时序逻辑电路。因此，它是Verilog HDL最高抽象级别的描述方式。
- 可以按照要求的设计算法来实现一个模块，而不用关心该模块具体硬件实现的细节。
- 这种抽象级别描述方式非常类似c编程。
- 一般行为级描述用于对设计进行仿真研究。

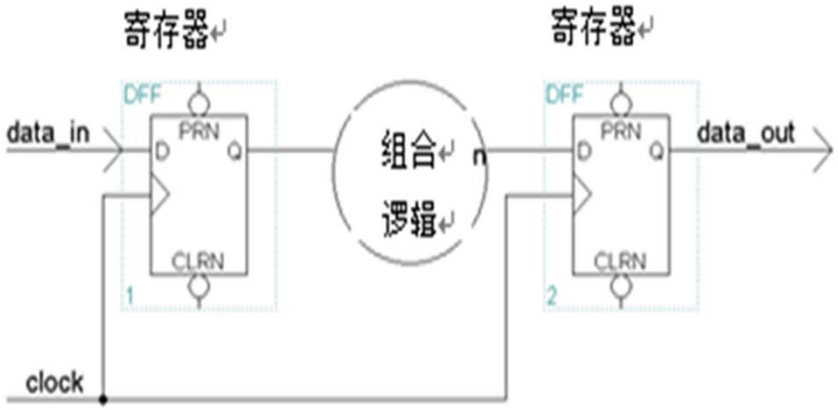
数据流描述方式

数据流描述方式，也称寄存器传输级（Register Transfer，RTL）描述方式。

所谓的数据流描述可以这样理解，即：在一个复杂的数字系统中，应该包含有数据流和控制流。

控制流用于控制数据的“流向”，即：数据将要到达的地方。

从寄存器传输级的角度，可以这样理解，即：在寄存器之间插入组合逻辑电路。



在一个复杂的数字系统中，任何数据从输入到输出，都需要经过寄存器，寄存器用于重定序和记忆。

数据流描述方式

行为级描述方式到底和数据流描述方式的本质区别在什么地方？

下面对其进行分析：

- 行为级描述中，包含一些设计元素，在FPGA内无法找到相应的逻辑单元来实现这些设计元素。
- 而数据流描述中，只包含可以在FPGA内实现的设计元素。
- 行为级描述，一般只用于对设计进行仿真，也就是生成对设计的测试向量，通过特定的仿真软件来测试设计有无设计缺陷。但是，不能转换成FPGA的具体物理实现。
- 而数据流描述，用于对设计进行综合，最后下载到FPGA器件进行具体的物理实现。

结构级描述方式

结构描述就是在设计中，通过调用库中的元件或者是已经设计好的模块来完成设计实体功能的描述。

通常情况下，在使用层次化设计时，一个高层次模块会调用一个或者多个低层次模块。这种模块的调用是通过模块例化语句实现的。

开关级描述方式

从本质上来说，开关级属于结构化描述方式，但是其描述更接近于底层的门级和开关级电路。

突出说明开关级描述方式，是为了说明Verilog HDL对底层强大的描述功能。

例：Verilog HDL开关级描述例子

```
module driver (in, out, en);
input [3:0] in;
output [3:0] out;
input en;
    bufif0 ar[3:0] (out, in, en); // 三态缓冲器阵列
endmodule
```

不同抽象级别的Verilog HDL模型

结构描述，最直观！

一、 Verilog HDL的门级描述

- 门级描述即直接调用门原语进行逻辑的结构描述。
  - 以门级为基础的结构描述所建立的硬件模型不仅是可仿真的，也是可综合的；
  - 一个逻辑网络由许多逻辑门和开关组成，用逻辑门的模型来描述逻辑网络最直观！
- 门类型的关键字有26个，常用的有9个：
  not，and，nand，or，nor，xor，xnor，buf，
  bufif1，bufif0，notif1，notif0（各种三态门）
- 调用门原语的句法：

可省略！

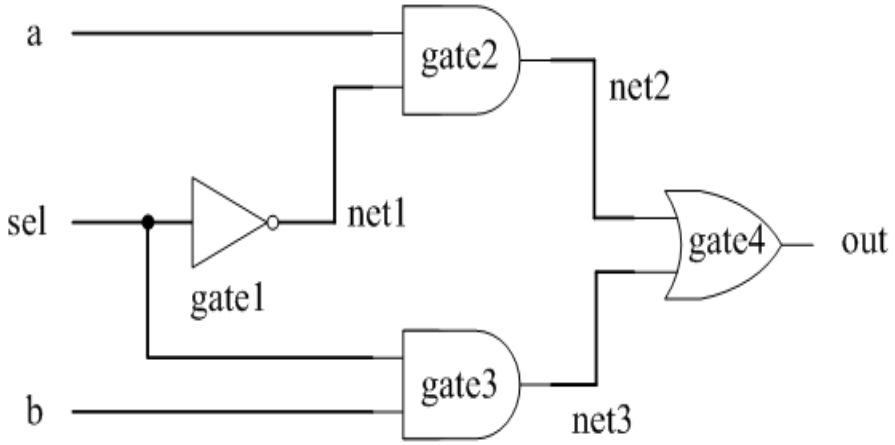
门类型关键字 〈例化的门名称〉（〈端口列表〉）；

注1：在端口列表中输出信号列在最前面；
注2：门级描述不适于描述复杂的系统！

门级（结构）风格的描述

- 在**Verilog HDL**中可使用以下方式描述电路结构：
  - 内置门原语（门级）
  - 开关级原语（晶体管级）
  - 用户定义的原语（门级）
  - 模块实例 （创建层次结构描述电路）

例：2选1数据选择器（MUX）电路，使用内置门原语对2选1 MUX的结构进行门级风格描述：




```
module mux_ str (out, a, b, sel);
        input a, b, sel;
        output out;
        not gate1 (net1, sel);
        and gate2 (net2, a, net1);
        and gate3 (net3, b, sel);
        or gate4 (out, net2, net3);
endmodule
```

模块例化语句的基本格式如下：

<module_name> <list_of_variable> <module_example_name>（<list_of_port>）；

其中：

- module_name是指被调用模块指定的模 块名。
- list_of_variable是可选项，它是由一些参数值组成的一个有序列表，将这些参数值传递给被调用模块实例内的各个参数。
- module_example_name是所生成的模块实例所命名的一个名字，它是被调用模块实例的唯一标识。
- list_of_port是由外部信号信号组成的一个有序列表， 这些外部信号端口表示与模块实例各个端口的连接。所以，<端口连接表>指明了模块实例端口与外部电路的连接情况。

- 注：在Verilog HDL中提供了两种方法用于端口信号的连接。可以按照端口列表的顺序进行端口的映射，也可以通过端口的名字进行映射。

三人表决器模块：

```
module voter(a,b,c,y);
  input a,b,c;
   output y;
   assign y=a&b | b&c | a&c;
endmodule
```

模块调用（实例化）

```
voter  inst(.b(b),.y(y),.c(c),.a(a));        //通过端口与信号的名字关联

voter  inst(a,b,c,y);                        //通过位置关联
```

模块例化规则

- 在某一模块内，可以多次调用同一模块。但是，每次调用生成的模块实例名不能重复。
- 实例名和模块名的区别是：
  - 模块名表示不同的模块，即用来区分电路单元的不同种类.
  - 而实例名则表示不同的模块实例，用来区分电路系统中的不同硬件电路单元。

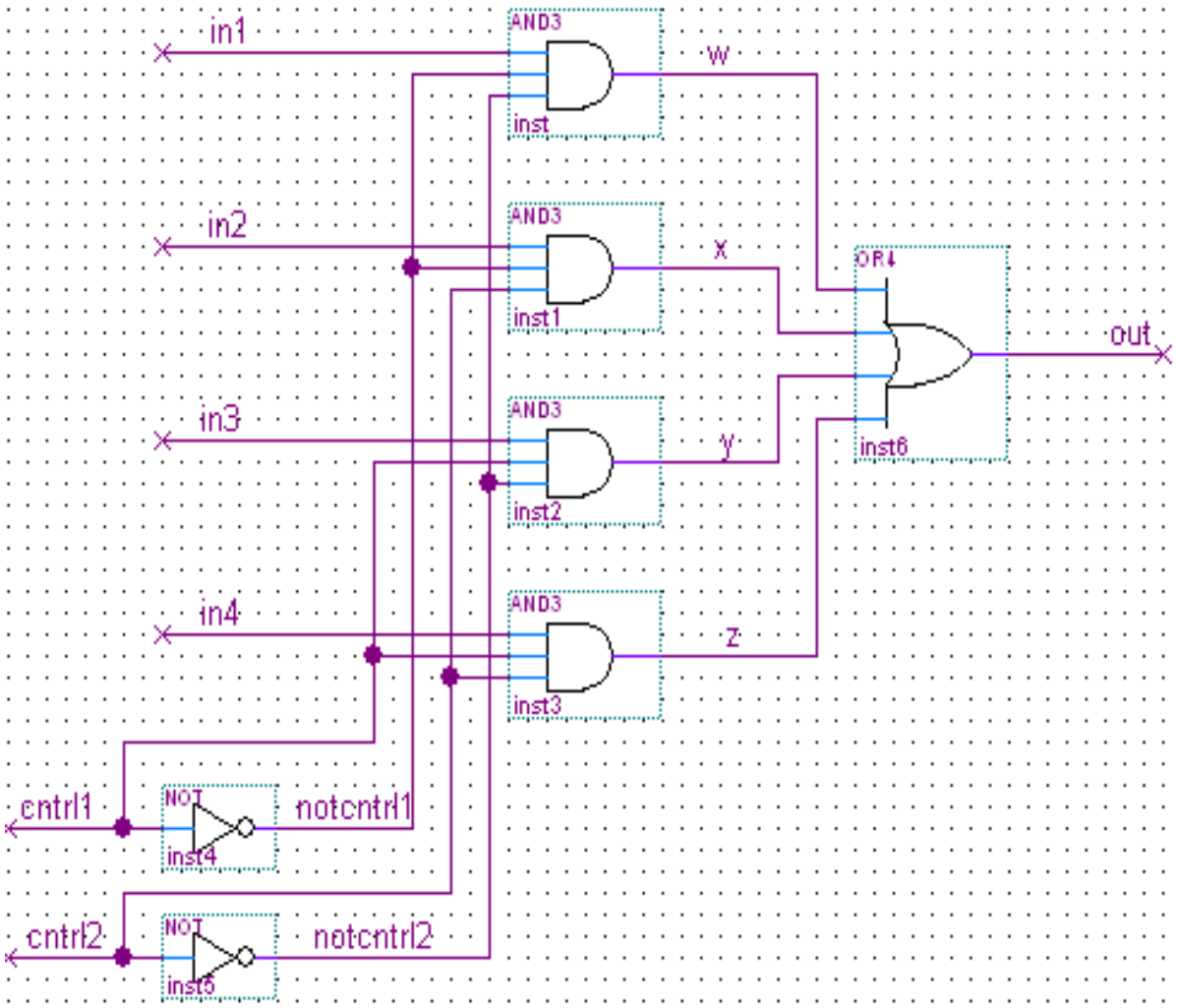
不同抽象级别的Verilog HDL模型

[例1] 调用门原语实现4选1数据选择器

❖ 注：首先必须根据逻辑功能画出逻辑电路图！

真值表

输 入		输出
cntrl1	**cntrl2**	**out**
0	0	in1
0	1	in2
1	0	in5
1	1	in4



```
//4_1 multiplexer using gate primitive.
module mymux (out, in1, in2, in3, in4, cntrl1, cntrl2);
        output  out;
        input   in1, in2, in3, in4, cntrl1, cntrl2;
        wire    notctrl1,notcntrl2,w,x,y,z;
        not(notcntrl1,cntrl1);
        not(notcntrl2,cntrl2);
        and(w,in1,notcntrl1,notcntrl2);
        and(x,in2,notcntrl1,cntrl2);
        and(y,in3,cntrl1,notcntrl2);
        and(z,in4,cntrl1,cntrl2);
        or(out,w,x,y,z);
endmodule
```

注：这里省略了所有的例化门元件名称！

不同抽象级别的Verilog HDL模型

包括系统级，算法级，RTL级

二、Verilog HDL的行为级描述

1. 逻辑功能描述——算法级

[例2] 用逻辑表达式实现4选1数据选择器

```
module mux4_1(out,in1,in2,in5,in4,cntrl1,cntrl2);
    output out;
    input in1,in2,in5,in4,cntrl1,cntrl2;
    assign out=(in1 & ~cntrl1 & ~cntrl2) |
                (in2 & ~cntrl1 & cntrl2) |
                (in5 & cntrl1 & ~cntrl2) |
                (in4 & cntrl1 & cntrl2) ;
endmodule
```

注：首先必须根据逻辑功能写出逻辑表达式！

不同抽象级别的Verilog HDL模型

2. case语句描述——系统级

——只需知道输入与输出间的真值表！比调用门原语和采用逻辑功能描述都简洁！

✓[例5] 用case语句描述4选1数据选择器

```
module mux4_1(out,in1,in2,in5,in4,cntrl1,cntrl2);
    output out;
    input in1,in2,in5,in4,cntrl1,cntrl2;
    reg out;
    always @(in1 or in2 or in5 or in4 or cntrl1 or cntrl2)
      case ({cntrl1,cntrl2})
        2'b00:out=in1;
        2'b01:out=in2;
        2'b10:out=in5;
        2'b11:out=in4;
        default :out=1'bx;
      endcase
endmodule
```

case语句应放在always块内！

不同抽象级别的Verilog HDL模型

3.条件运算符描述——算法级

——只需知道输入与输出间的真值表！

✓[例4] 用条件运算符描述4选1数据选择器

```
module mux4_1(out,in1,in2,in5,in4,cntrl1,cntrl2);
    output out;
    input in1,in2,in5,in4,cntrl1,cntrl2;
    assign out= cntrl1? (cntrl2? in4 :in5) : (cntrl2? in2 :in1) ;
endmodule
```

当cntrl1=1时执行

当cntrl1=0时执行

注：比调用门原语，采用逻辑表达式或case语句描述代码更简单！但也更抽象！且耗用器件资源更多！

行为描述

- 抽象层次更高的设计风格，是为了综合和仿真的目的而进行的，常用于复杂数字系统的顶层逻辑设计。
- 2选1数据选择器（MUX）电路，使用行为描述方式建模：

```
module mux_beha (out, a, b, sel);
    input a, b, sel;
    output out;
    reg out;
    always @(a,b,sel)
        out=(sel)?a:b;
endmodule
```

数据流风格的描述（寄存器传输级）

- 最基本的机制：使用连续赋值语句
- 在连续赋值语句中，线网类型变量被赋予某个值，右边表达式的操作数无论何时发生变化，表达式都重新计算，计算结果被赋予左边表达式的线网类型变量。
- 各assign语句是并行执行的，即各语句的执行与语句的编写顺序无关。

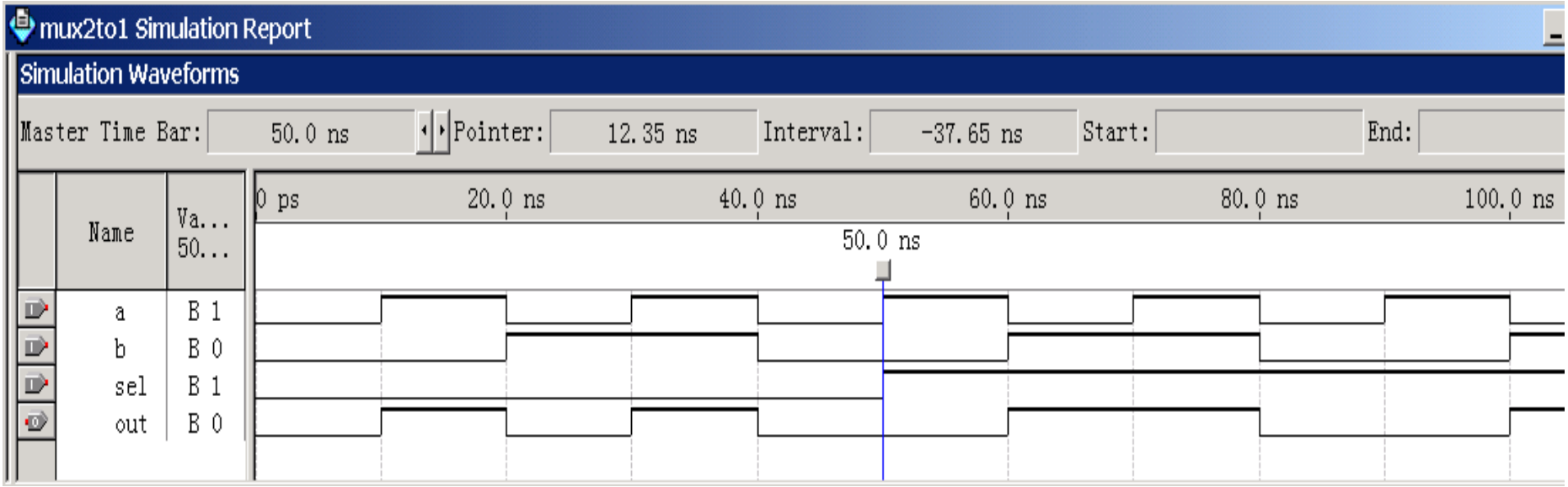
例：2选1数据选择器（MUX）电路，使用数据流描述方式描述：

```
module mux_flow (out, a, b, sel);
        input a, b, sel;
        output out;
        assign out=(sel)?a:b;
endmodule
```

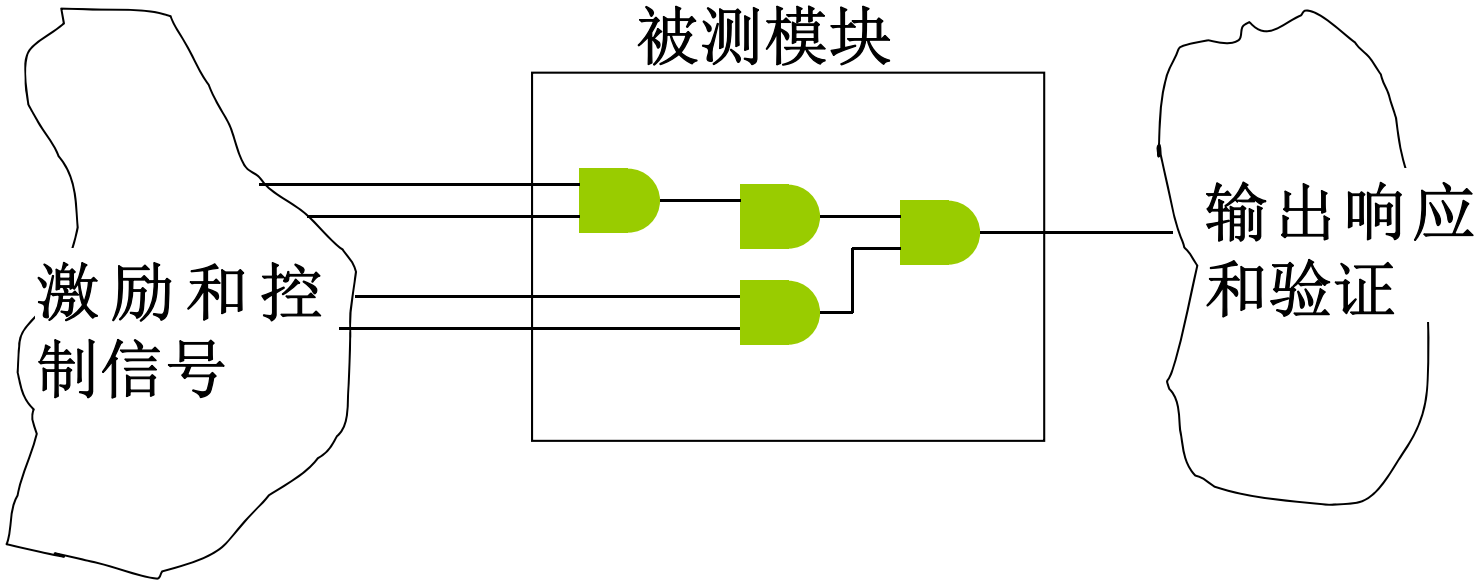
当 a、b、sel 有变化时，out 将同时变化

4 逻辑功能的仿真与测试

逻辑电路的设计块完成后，就要测试这个设计块描述的逻辑功能是否正确。为此必须在输入端口加入测试信号，而从其输出端口检测其结果是否正确，这一过程常称为搭建测试台 ***test bench*** 。根据仿真软件的不同，搭建测试平台的方法也不同。



被测器件(DUT,device under test)

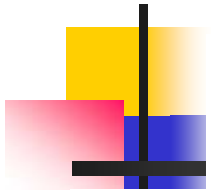


- 测试台 (*test bench*) 提供测试激励及验证机制。
- *Test bench* 使用行为级描述。

Test Bench 模板

```
module testbench;
    // Data type declaration

    // Instantiate modules

    // Apply stimulus

    // Display results
endmodule
```

由于*testbench*是最顶层模块，不会被其它模块实例化。因此不需要有端口。



测试模块常见的形式

```
module test;
reg …;              //被测模块输入变量类型定义
wire…;              //被测模块输出变量类型定义
Testedmd     m(.in1(ina),in2(inb),.out1(outa),.out2(outb)   );
   //被测模块的实例引用
initial begin …;    …;   …;   end … … //产生测试信号
always #delay   begin …;   end   … …    //产生测试信号

initial begin ….; ….; ….      end     //记录输出和响应
endmodule
```

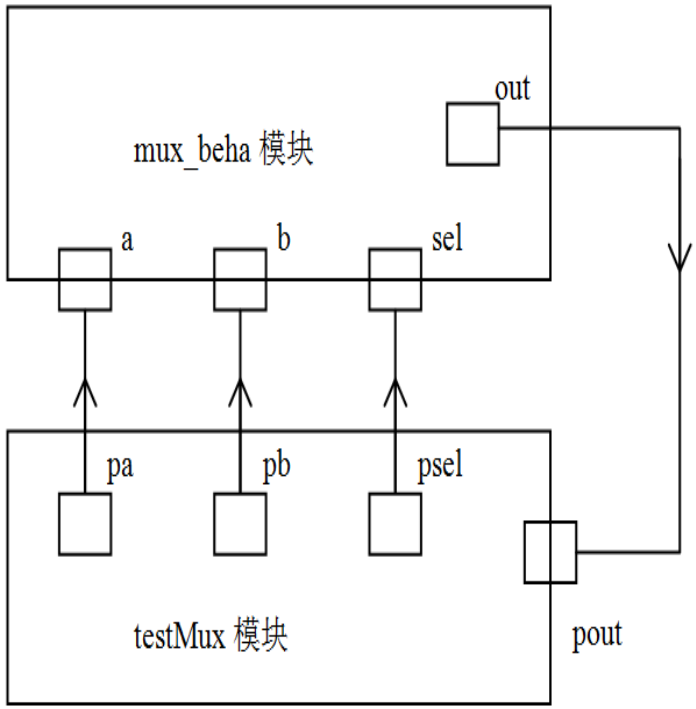
Test Bench —过程(procedural block)

- ***Test Bench*** 通常采用过程语句进行行为级描述。***Test Bench***的激励信号在一个过程语句中描述。
- 过程语句有两种：
  - ***initial*** ：只执行一次
  - ***always*** ：循环执行
- 过程语句的活动与执行是有差别的：
  - 所有过程在时间0处于活动状态，并根据用户定义的条件等待执行；
  - 所有过程并行执行，以描述硬件内在的并行性；

测试平台的编写

- 测试平台（**testbench**）：为了模拟数据的输入和输出、测试和验证功能模块正确性而编写的另一个模块。

```
module testMux;
   reg pa,pb,psel;
   wire pout;
   mux_beha tmux(pout,pa,pb,psel);   // 调用 mux_beha 模块，
                                     //按端口顺序对应方式连接
   initial
          begin
              pa=0;pb=0;psel=0;   // 赋予初值
              #5 pa=1;            // 5 个单位时间延迟后进行赋值
              #5 pb=1;
              #5 pa=0;
              #5 psel=1;
              #5 pa=1;
              #5 pb=0;
              #5 pa=0;
          end
   initial
     $monitor("time=%t,a=%b,b=%b,sel=%b,out=%b",$time,pa,pb,psel,pout);
endmodule
```



如何观察被测模块的响应：

- 在initial 块中，用系统任务$time 和 $monitor
- $time　返回当前的仿真时刻
- $monitor 只要在其变量列表中有某一个或某几个变量值发生变化，便在仿真单位时间结束时显示其变量列表中所有变量的值。

例：

```
initial
begin
$monitor ($time, , “out=%b a=%b sel=%b”,out,a,b,sel);
end
```

如何把被测模块的输出变化记录到数据库文件中？

（文件格式为VCD,大多数的波形显示工具都能读取该格式）

可用以下七个系统任务：

1) $dumpfile(“file.dump”); //打开记录数据变化的数据文件

2) $dumpvars(); //选择需要记录的变量

3) $dumpflush; //把记录在数据文件中的资料转送到硬盘保存

4) $dumpoff; //停止记录数据变化

5) $dumpon; //重新开始记录数据变化

6) $dumplimit(<file_size>); //规定数据文件的大小（字节）

7) $dumpall; //记录所有指定信号的变化值到数据文件中

如何把被测模块的响应变化记录到数据库文件中？

举例说明：

```
$dumpvars;                    //记录各层次模块中所有信号的变化
$dumpvars(1,top);    //只记录模块top中所有信号的变化
$dumpvars(2,top.u1); //记录top模块中实例u1和它以下一层子模块所有
    信号的变化
$dumpvars(0,top.u2,top.u1.u13.q); //记录top模块中实例u2和它本
    层所有信号的变化,还有top.u1.u13.q信号的变化。
$dumpvars(3,top.u2,top.u1);
//记录top模块中u2和u1所有信号的变化（包括其两层以下子模块的信号变化）。
```

不同抽象级别的Verilog HDL模型

小结

- 采用的描述级别越高，设计越容易，程序代码越简单；但耗用器件资源更多。对特定综合器，可能无法将某些抽象级别高的描述转化为电路！
- 基于门级描述的硬件模型不仅可以仿真，而且可综合，且系统速度快。
- 所有Verilog HDL编译软件只是支持该语言的一个子集。
- 尽量采用编译软件支持的语句来描述设计；或多个软件配合使用。
- 一般用算法级（写出逻辑表达式）或RTL级来描述逻辑功能，尽量避免用门级描述，除非对系统速度要求比较高的场合才采用门级描述。

不同抽象级别的Verilog HDL模型

思考

（1）采用什么描述级别更合适？

- 系统级描述太抽象，有时无法综合成具体的物理电路；门级描述要求根据逻辑功能画出逻辑电路图，对于复杂的数字系统很难做到；
- 而算法级和RTL级描述级别适中，代码不是很复杂，且一般容易综合成具体的物理电路，故建议尽量采用算法级和RTL级来描述。

（2）怎样减少器件逻辑资源的耗用？

- 当器件容量有限时，为减少器件逻辑资源的耗用，建议少用if-else语句和case语句，尽量直接使用逻辑表达式来描述系统的逻辑功能；
- 或者用case语句取代if-else语句。

2 Verilog HDL基本结构

用户程序中的变量、节点等名称不能与关键字同名！

四、关键字

- 关键字——事先定义好的确认符，用来组织语言结构；或者用于定义Verilog HDL提供的门元件（如and，not，or，buf）。
- 用小写字母定义！

  ——如always，assign，begin，case，casex，else，end，for，function，if，input，output，repeat，table，time，while，wire

2 Verilog HDL基本结构

Verilog HDL关键字

and	edge	function	negedge
always	else	highz0	nor not
assign	end	highz1	notif0
begin	endcase	if	notif1
buf	endfunction	ifnone	nmos
bufif0	endprimitive	initial	or
bufif1	endmodule	inout	output
case	endspecify	input	parameter
casex	endtable	integer	pmos
casez	endtask	join	posedge
cmos	event	large	primitive
deassign	for	macromodule	pulldown
default	force	medium	pullup
defparam	forever	module	pull0
disable	fork	nand	pull1

Verilog HDL关键字（续）

rcmos
real
realtime
reg
release
repeat
rnmos
rpmos
rtran
rtranif0
rtranif1
scalared
small
specify
specparam
strength
strong0
strong1
supply0
supply1
table
task
tran
tranif0
tranif1
time
tri
triand
trior
trireg
tri0
tri1
vectored
wait
wand
weak0
weak1
while
wire
wor
xnor
xor

2 Verilog HDL基本结构

五、标识符

标识符不能与关键字同名！

- 任何用Verilog HDL语言描述的“东西”都通过其名字来识别，这个名字被称为标识符。
- 如源文件名、模块名、端口名、变量名、常量名、实例名等。
- 标识符可由字母、数字、下划线和$符号构成；但第一个字符必须是字母或下划线，不能是数字或$符号！
- 在Verilog HDL中变量名是区分大小写的！
- 合法的名字：
  - A_99_Z
  - Reset
  - _54MHz_Clock$
  - Module
- 不合法的名字：
  - 125a
  - $data
  - module
  - 7seg.v

六、编写Verilog HDL源代码的标准

- 编写Verilog HDL源代码的标准分为两类：

（1）语汇代码的编写标准

规定了文本布局、命名和注释的约定，以提高源代码的可读性和可维护性。

（2）综合代码的编写标准

规定了Verilog风格，尽量保证能够综合，以避免常见的不能综合及综合结果存在缺陷的问题，并在设计流程中及时发现综合中存在的错误。

❖综合：将用HDL语言或图形方式描述的电路设计转换为实际门级电路（如触发器、逻辑门等），得到一个网表文件，用于进行适配（在实际器件中进行布局和布线）。

2 Verilog HDL基本结构

1 语汇代码的编写标准

（1）每个Verilog HDL源文件中只准编写一个顶层模块，也不能把一个顶层模块分成几部分写在几个源文件中。

（2）源文件名字应与文件内容有关，最好与顶层模块同名！源文件名字的第一个字符必须是字母或下划线，不能是数字或$符号！

（5）每行只写一个声明语句或说明。

（4）源代码用层层缩进的格式来写。

2 Verilog HDL基本结构

1 语汇代码的编写标准（续）

（5）定义变量名的大小写应自始至终保持一致（如变量名第一个字母均大写）。

（6）变量名应该有意义，而且含有一定的有关信息。局部变量名（如循环变量）应简单扼要。

（7）通过注释对源代码做必要的说明，尤其对接口（如模块参数、端口、任务、函数变量）做必要的注释很重要。

（8）常量尽可能多地使用参数定义和宏定义，而不要在语句中直接使用字母、数字和字符串。

- 参数定义（用一个标识符来代表一个常量）的格式：
  parameter 参数名1=表达式，参数名2=表达式，……;
- 宏定义（用一个简单的宏名来代替一个复杂的表达式）的格式：
  ’define 标志符（即宏名）字符串（即宏内容）

2 Verilog HDL基本结构

2 综合代码的编写标准

（1）把设计分割成较小的功能块，每块用行为风格设计。除设计中对速度响应要求比较临界的部分外，都应避免门级描述。

（2）建立一个好的时钟策略（如单时钟、多相位时钟，经过门产生的时钟、多时钟域等）。保证源代码中时钟和复位信号是干净的（即不是由组合逻辑或没有考虑到的门产生的）。

（5）建立一个好的测试策略，使所有触发器都是可复位的，使测试能通过外部管脚进行，又没有冗余的功能。

（4）所有源代码都必须遵守并符合在always块语句的4种可综合标准模板之一。

（5）描述组合和锁存逻辑的always块，必须在always块开头的控制事件列表中列出所有的输入信号。

2 综合代码的编写标准（续1）

（6）描述组合逻辑的always块，一定不能有不完全赋值，即所有输出变量必须被各输入值的组合值赋值，不能有例外。

（7）描述组合和锁存逻辑的**always**块一定不能包含反馈，即在**always**块中已被定义为输出的寄存器变量绝对不能再在该**always**块中读进来作为输入信号。

（8）时钟沿触发的**always**块必须是单时钟的，且任何异步控制输入（通常是复位或置位信号）必须在控制事件列表中列出。

例：**always @(posedge clk or negedge set or negedge reset)**

（9）避免生成不想要的锁存器。在无时钟的**always**块中，若有的输出变量被赋了某个信号变量值，而该信号变量并未在该**always**块的电平敏感控制事件中列出，则会在综合中生成不想要的锁存器。

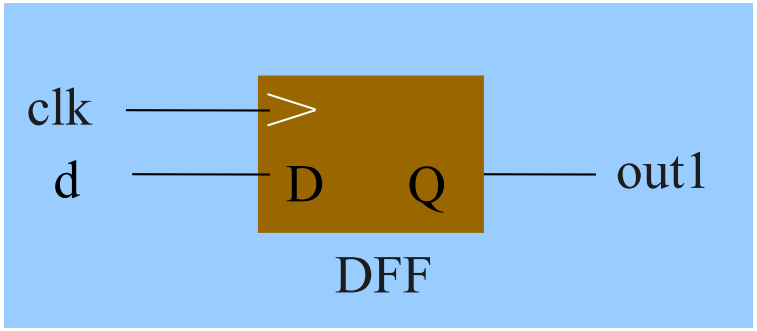
2 综合代码的编写标准（续2）

（10）避免生成不想要的触发器。

- 在时钟沿触发的always块中，如果用非阻塞赋值语句对reg型变量赋值；或者当reg型变量经过多次循环其值仍保持不变，则会在综合中生成触发器。
- 用reg型变量生成触发器举例：

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

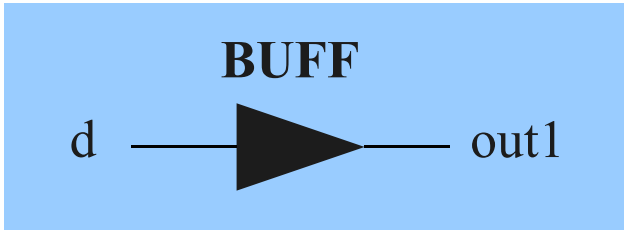
非阻塞赋值语句



2 综合代码的编写标准（续5）

- 若不想生成触发器，而是希望用reg型变量生成组合逻辑，则应使用电平触发：

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



2 Verilog HDL基本结构

2 综合代码的编写标准（续4）

（11）所有内部状态寄存器必须是可复位的，这是为了使RTL级和门级描述能够被复位成同一个已知的状态，以便进行门级逻辑验证。

（12）对存在无效状态的有限状态机和其他时序电路（如4位十进制计数器有6个无效状态），必须明确描述所有的2的N次幂种状态下的行为（包括无效状态），才能综合出安全可靠的状态机。

（15）一般地，在赋值语句中不能使用延迟，否则是不可综合的。

（14）不要使用integer型和time型寄存器，否则将分别综合成52位和64位的总线。

（15）仔细检查代码中使用动态指针（如用指针或地址变量检索的位选择或存储单元）、循环声明或算术运算部分，因为这类代码在综合后会生成大量的门，且难以优化。

3 数据类型及常量、变量

内容概要

一、数据类型

二、常量

三、变量

3 数据类型及常量、变量

一、数据类型

- 数据类型是用来表示数字电路中的数据存储和传送单元。
- Verilog HDL中共有19种数据类型；
- 其中4个最基本的数据类型为：
  - integer型
  - parameter型
  - reg型
  - wire型

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

二、常量

- 在程序运行过程中，其值不能被改变的量，称为常量。
  - 数字（包括整数，x和z值，负数）
  - parameter常量（或称符号常量）

数据类型及常量、变量

(1) 整数型常量（即整常数）的4种进制表示形式：

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

- 整常数的5种表达方式：

表达方式	说明	举例
<位宽>’<进制><数字>	完整的表达方式	8’b11000101或 8 ’hc5
<进制><数字>	缺省位宽，则位宽由机器系统决定，至少52位	hc5
<数字>	缺省进制为十进制，位宽默认为52位	197

❖注：这里位宽指对应二进制数的宽度。

3 数据类型及常量、变量

8’b1001xxxx
或8 ’h9x

8’b1010zzzz
或8 ’haz

(2) x和z值

- x表示不定值，z表示高阻值；

- 每个字符代表的二进制数的宽度取决于所用的进制；
- 当用二进制表示时，已标明位宽的数若用x或z表示某些位，则只有在最左边的x或z具有扩展性！为清晰可见，最好直接写出每一位的值！
  - [例]8’bzx = 8’bzzzz_zzzx
  - [例]8’b1x = 8’b0000_001x
- “？”是z的另一种表示符号，建议在case语句中使用？表示高阻态z
  - [例] casez (select)

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

3 数据类型及常量、变量

(5) 负数

- 在位宽前加一个减号，即表示负数
- 如：-8'd5　//5的补数，=8'b11111011
- 减号不能放在位宽与进制之间，也不能放在进制与数字之间！
- 8 ' d-5　//非法格式

❖ 为提高可读性，在较长的数字之间可用下划线_隔开！但不可以用在<进制>和<数字>之间。

如：16'b1010_1011_1100_1111 //合法

8'b_0011_1010 //非法

❖ 当常量未指明位宽时，默认为52位。

- 10 = 52'd10 = 52'b1010
- -1 = -52'd1 = 52'b1111.....1111 = 52'hFFFFFFFF

3 数据类型及常量、变量

(4) parameter常量（符号常量）

- 用parameter来定义一个标识符，代表一个常量——称为符号常量。

格式

```
parameter 参数名1 = 表达式,参数名2 = 表达式, ……;
```

参数型数据的确认符

赋值语句表

- 每个赋值语句的右边必须为常数表达式，即只能包含数字或先前定义过的符号常量!

```
parameter addrwidth = 16;          //合法格式
parameter addrwidth = datawidth*2; //非法格式
```

- 常用参数来定义延迟时间和变量宽度。
- 可用字符串表示的任何地方,都可以用定义的参数来代替。
- 参数是本地的，其定义只在本模块内有效。
- 在模块或实例引用时，可通过参数传递改变在被引用模块或实例中已定义的参数!

- 模块实例引用时参数的传递——方法之一：

利用defparam定义参数声明语句！

格式

defparam 例化模块名.参数名1 = 常数表达式,
例化模块名.参数名2 = 常数表达式, ……;

- defparam语句在编译时可重新定义参数值。
- 可综合性问题：一般情况下是不可综合的。
- 提示：不要使用defparam语句！在模块的实例引用时可用“#”号后跟参数的语法来重新定义参数。

5.5 数据类型及常量、变量

[例]

```
module mod ( out, ina, inb);
      …
    parameter cycle = 8, real_constant = 2.059,
                  file = “/user1/jmdong/design/mem_file.dat”;
      …
endmodule

module test;
      …
    mod mk(out,ina,inb);        // 对模块mod的实例引用
    defparam  mk.cycle = 6, mk.file = “../my_mem.dat”;   // 参数的传递
      …
endmodule
```

被引用模块

例化模块名

参数名

5.5 数据类型及常量、变量

建议用此方法！

- 模块实例引用时参数的传递 [illegible]，利用特殊符号“#”

格式

被引用模块名 # （参数1,参数2,...）例化模块名（端口列表）；

[例]

```
module mod ( out, ina, inb);
      …
    parameter cycle = 8, real_constant = 2.059,
                  file = “/user1/jmdong/design/mem_file.dat”;
      …
endmodule

module test;
      …
    mod # (5, 5.20, “../my_mem.dat”)  mk(out,ina,inb); // 对模块mod的实例引用
      …
endmodule
```

被引用模块

参数的传递

必须与被引用模块中的参数一一对应！

5.5 数据类型及常量、变量

三、变量

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

1. nets型变量

- **定义——输出始终随输入的变化而变化的变量。**
  表示结构实体（如门）之间的物理连接。
- 常用nets型变量：
  - wire，tri：连线类型（两者功能一致）
  - wor，trior：具有线或特性的连线（两者功能一致）
  - wand，triand：具有线与特性的连线（两者功能一致）
  - tri1，tri0：上拉电阻和下拉电阻
  - supply1，supply0：电源（逻辑1）和地（逻辑0）

3 数据类型及常量、变量

- **wire**型变量
  - 最常用的nets型变量，常用来表示以assign语句赋值的组合逻辑信号。
  - 模块中的输入/输出信号类型缺省为wire型。
  - 可用做任何方程式的输入，或“assign”语句和实例元件的输出

格式

```
wire 数据名1,数据名2, ……,数据名n;
```

wire型向量（总线）

```
wire[n-1:0] 数据名1,数据名2, ……,数据名m;
或 wire[n:1] 数据名1,数据名2, ……,数据名m;
```

每条总线位宽为**n**

共有**m**条总线

2. register型变量

- 定义——对应具有状态保持作用的电路元件（如触发器、寄存器等），常用来表示过程块语句（如initial，always，task，function）内的指定信号。
- 常用register型变量：
  - reg：常代表触发器
  - integer：52位带符号整数型变量
  - real：64位带符号实数型变量
  - time：无符号时间变量

纯数学的抽象描述

5.5 数据类型及常量、变量

- register型变量与nets型变量的根本区别是：register型变量需要被明确地赋值，并且在被重新赋值前一直保持原值。
- register型变量必须通过过程赋值语句赋值！不能通过assign语句赋值！
- 在过程块内被赋值的每个信号必须定义成register型！

3 数据类型及常量、变量

- **reg**型变量
  - **定义——在过程块中被赋值的信号，**往往代表触发器，但不一定就是触发器（也可以是组合逻辑信号）！

格式

```
reg 数据名1,数据名2, ……,数据名n;
```

reg型向量（总线）

```
reg[n-1:0] 数据名1,数据名2, ……,数据名m;
或 reg[n:1] 数据名1,数据名2, ……,数据名m;
```

每个向量位宽为**n**

共有**m**个reg型向量

  - [例] reg[4:1] regc,regd; //regc,regd为4位宽的reg型向量

3 数据类型及常量、变量

Verilog中reg与wire的区别

reg型变量既可生成触发器，也可生成组合逻辑； wire型变量只能生成组合逻辑。

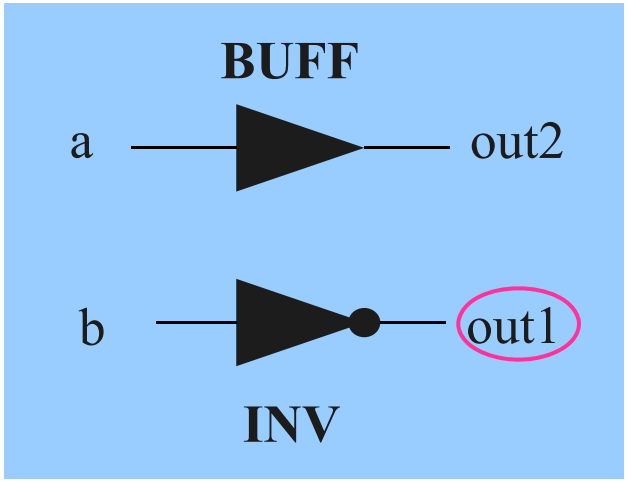
- 用reg型变量生成组合逻辑举例：

```
module  rw1( a, b, out1, out2 ) ;
    input a, b;
    output out1, out2;
    reg out1;
    wire out2;
    assign out2 = a ;
    always @(b)
      out1 <= ~b;
endmodule
```

连续赋值语句

电平触发

过程赋值语句



3 数据类型及常量、变量

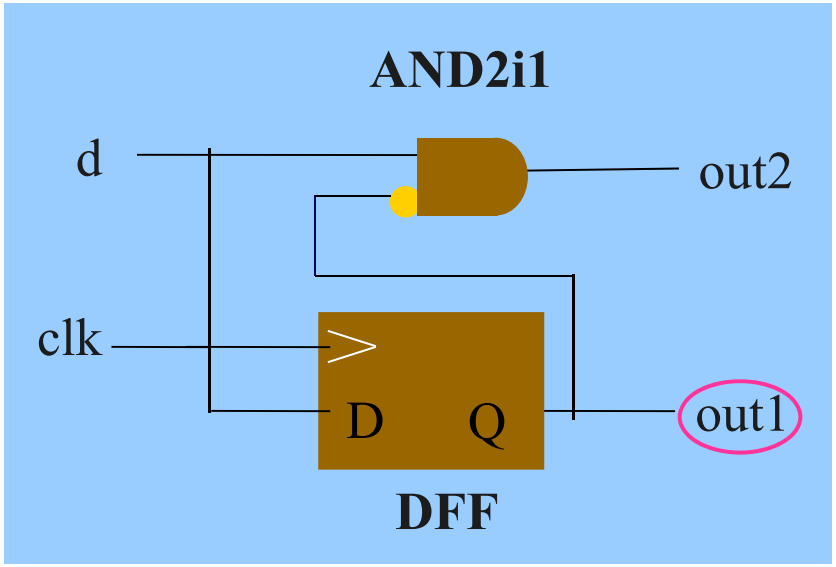
- 用reg型变量生成触发器举例：

```
module  rw2( clk, d, out1, out2 );
    input clk, d;
    output out1, out2;
    reg out1;
    wire out2;
    assign out2 =  d  & ~out1 ;
    always @(posedge clk)
      begin
        out1 <=  d ;
      end
endmodule
```

连续赋值语句

沿触发

过程赋值语句



3 数据类型及常量、变量

3. memory型变量——数组

Quartus II 不支持！

- 定义——由若干个相同宽度的reg型向量构成的数组。
- Verilog HDL通过reg型变量建立数组来对存储器建模。
- memory型变量可描述RAM、ROM和reg文件。
- memory型变量通过扩展reg型变量的地址范围来生成：

```
reg[n-1:0] 存储器名[m-1:0];
或 reg[n-1:0]存储器名[m:1];
```

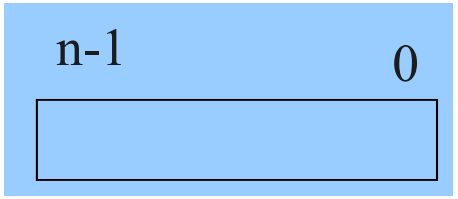
每个存储单元位宽为**n**

共有**m**个存储单元

❖Verilog HDL中的变量名、参数名等标记符是对大小写字母敏感的！

3 数据类型及常量、变量

memory型变量与reg型变量的区别



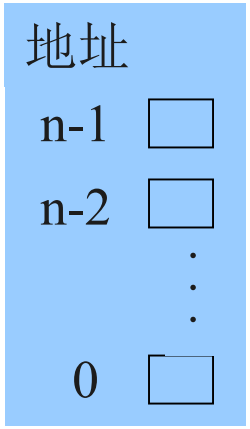
- **含义不同**

[例] reg[n-1:0] rega; //一个n位的寄存器

reg mema [n-1:0]; //由n个1位寄存器组成的存储器

- **赋值方式不同**

一个n位的寄存器可用一条赋值语句赋值；一个完整的存储器则不行！若要对某存储器中的存储单元进行读写操作，必须指明该单元在存储器中的地址！



[例] rega = 0; //合法赋值语句

mema = 0; //非法赋值语句

mema[8] = 1; //合法赋值语句

mema[1025:0] = 0; //合法赋值语句

必须指明存储单元的地址！

4 运算符及表达式

内容概要

一、算术运算符
二、逻辑运算符
三、位运算符
四、关系运算符
五、等式运算符
六、缩减运算符
七、移位运算符
八、条件运算符
九、位拼接运算符
十、运算符的优先级

- 运算符按功能分为9类：
  - 算术运算符
  - 逻辑运算符
  - 关系运算符
  - 等式运算符
  - 缩减运算符
  - 条件运算符
  - 位运算符
  - 移位运算符
  - 位拼接运算符

- 运算符按操作数的个数分为5类：
  - 单目运算符——带一个操作数
    逻辑非！，按位取反~，缩减运算符，移位运算符
  - 双目运算符——带两个操作数
    算术、关系、等式运算符，逻辑、位运算符的大部分
  - 三目运算符——带三个操作数
    条件运算符

双目运算符

一、算术运算符

MAX + PLUS II不支持
“/”和“%”运算！
Quartus II都支持！

算术运算符	说明
+	加
—	减
*	乘
/	除
%	求模

- 进行整数除法运算时，结果值略去小数部分，只取整数部分！
- %称为求模（或求余）运算符，要求%两侧均为整型数据；
- 求模运算结果值的符号位取第一个操作数的符号位！

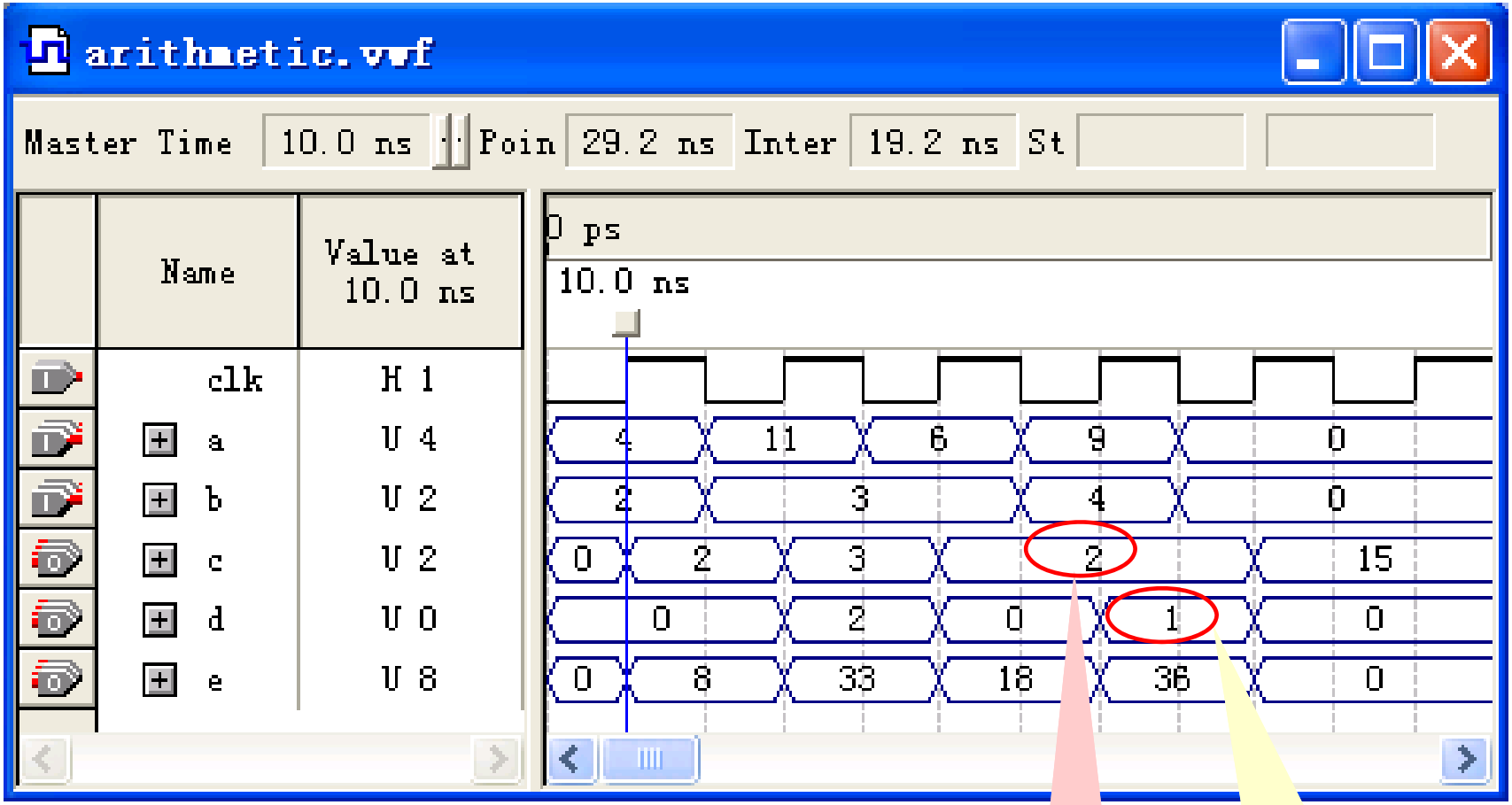
  ［例］-11%5　　结果为-2
- 进行算术运算时，若某操作数为不定值x，则整个结果也为x。

- ［例］除法和求模运算的区别

注意/和%的区别！

arithmetic.v

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



arithmetic.vwf

二、逻辑运算符

- 逻辑运算符把它的操作数当作布尔变量：
  - 非零的操作数被认为是真(1'b1)；
  - 零被认为是假(1'b0)；
  - 不确定的操作数如4'bxx00，被认为是不确定的(可能为零，也可能为非零)(记为1'bx)；但4'bxx11被认为是真(记为1'b1，因为它肯定是非零的)。

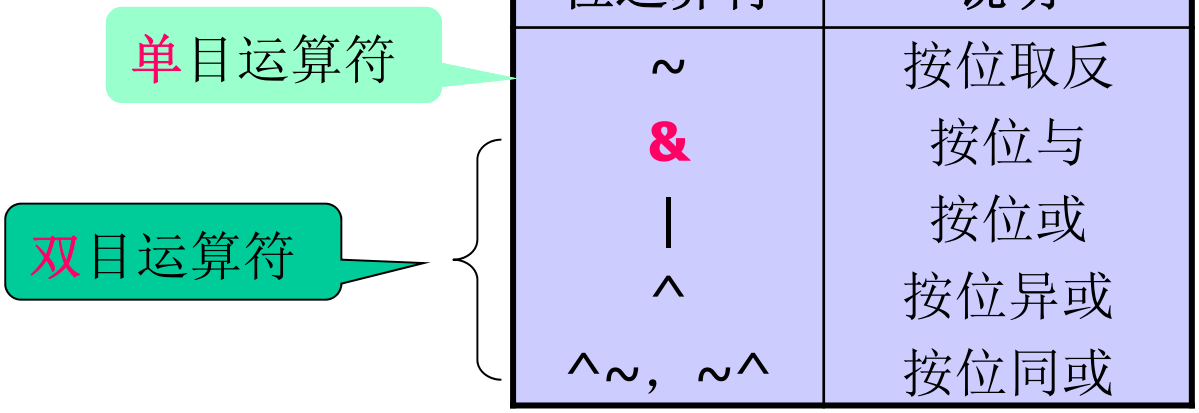
逻辑运算符	说明
&&(双目)	逻辑与
||(双目)	逻辑或
!(单目)	逻辑非

❖ 进行逻辑运算后的结果为布尔值（为1或0或x）！

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

为提高程序的可读性，明确表达各运算符之间的优先关系，建议使用括号！

三、位运算符



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

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

[例] 若A = 5'b11001，B = 5'b101，

则A & B = （5'b11001）&（5'b00101）= 5'b00001

101

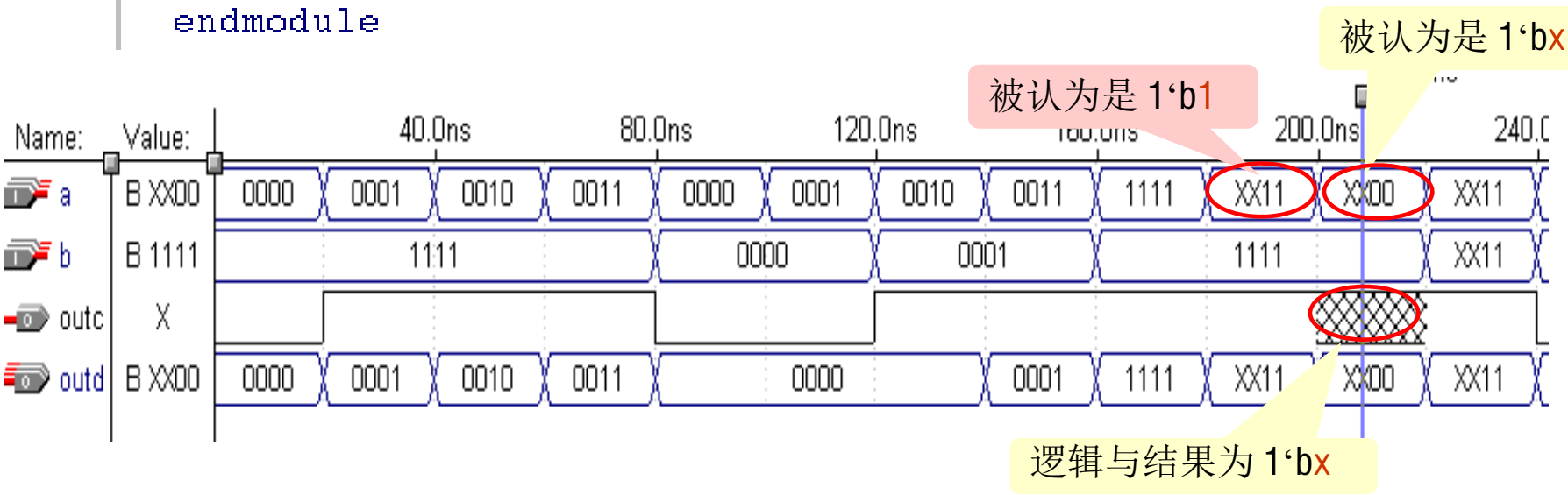
注意&&和&的区别！

- ［例］&&运算符和&（按位与）的区别

logic_demo.v – Text Editor

```
// Logic Operators and bit Operators.

module logic_demo(outc,outd,a,b) ;
  output          outc;
  output[3:0]     outd;
  input [3:0]     a,b;
  assign outc = a&&b;        /* Logic and operation */
  assign outd = a&b;         /* bit and operation   */
endmodule
```

&&运算的结果为1位的逻辑值



四、关系运算符

双目运算符

关系运算符	说明
<	小于
<=	小于或等于
>	大于
>=	大于或等于

- 运算结果为1位的逻辑值1或0或x。关系运算时，若关系为真，则返回值为1；若声明的关系为假，则返回值为0；若某操作数为不定值x，则返回值为x。
- 所有的关系运算符优先级别相同。
- 关系运算符的优先级低于算术运算符。
- [例] a<size – 1　　等同于：a<(size – 1)

  size – (1<a)　　不等同于：size–1<a

括号内先运算！

算术运算先运算！

五、等式运算符

双目运算符

等式运算符	说明
==	等于
！=	不等于
===	全等
！==	不全等

MAX + PLUS II和 Quartus II都不支持!

- 运算结果为1位的逻辑值1或0或x。
- **等于运算符(==)和全等运算符(===)的区别:**
  - 使用等于运算符时，两个操作数必须逐位相等,结果才为1；若某些位为x或z，则结果为x。
  - 使用全等运算符时，若两个操作数的相应位完全一致（如同是1，或同是0，或同是x，或同是z）,则结果为1；否则为0。
- 所有的等式运算符优先级别相同。
- ==和!==运算符常用于case表达式的判别，又称为“case等式运算符”。

表5-1 “= =”的真值表

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

等于运算的结果可能为1或0或x

表5-2 “= = =”的真值表

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

全等于运算的结果只有1或0

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

  if(A = = = 1'bx) $display(“AisX”); //当A为不定值时，式（A===1'bx）的运算结果为1，该语句执行

六、缩减运算符

单目运算符

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

注意缩减运算符和位运算符的区别！

- 运算法则与位运算符类似，但运算过程不同！
- 对单个操作数进行递推运算，即先将操作数的最低位与第二位进行与、或、非运算，再将运算结果与第三位进行相同的运算，依次类推，直至最高位 。
- 运算结果缩减为1位二进制数。
- [例]reg[5:0] a;

  b=|a;　　//等效于 b =((a[0] | a[1]) | a(2)) | a[5]

七、移位运算符

单目运算符

移位运算符	说明
>>	右移
<<	左移

只有当右操作数为常数时MAX + PLUS II支持！

- 用法：A>>n　　或 A<<n

  将操作数右移或左移n位，同时用n个0填补移出的空位。

- [例] 4'b1001>>5 = 4'b0001;　4'b1001>>4 = 4'b0000

  4'b1001<<1 = 5'b10010;　4'b1001<<2 = 6'b100100;

  1<<6 = 52'b1000000

右移位数不变，但右移的数据会丢失！

左移会扩充位数！

❖将操作数右移或左移n位，相当于将操作数除以或乘以2^n。

八、条件运算符

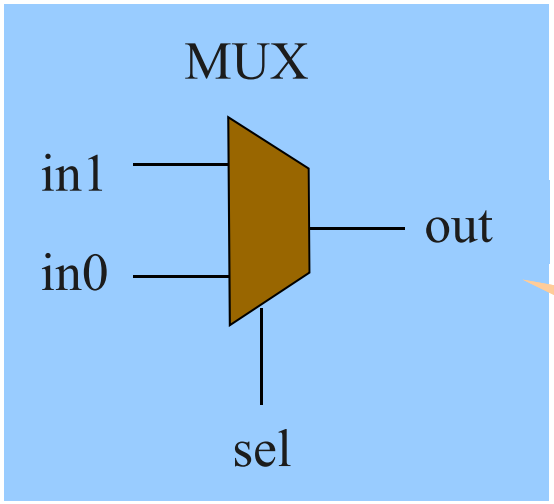
三目运算符

- 条件运算符为？：
- 用法：

信号 = 条件？表达式1：表达式2

当条件为真，信号取表达式1的值；为假，则取表达式2的值。

- ［例］数据选择器**assign out = sel? in1:in0;**



sel=1时out=in1;
sel=0时out=in0

九、位拼接运算符

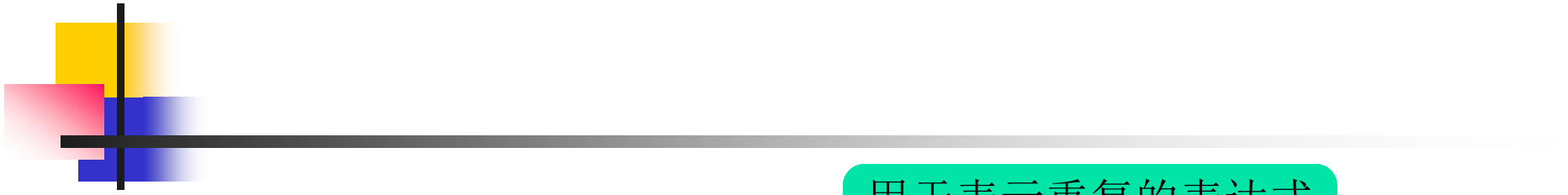
- 位拼接运算符为{ }
- 用于将两个或多个信号的某些位拼接起来，表示一个整体信号。
- 用法：{信号1的某几位，信号2的某几位，……，信号n的某几位}

- 例如在进行加法运算时，可将进位输出与和拼接在一起使用。
- [例1] output [5:0] sum; //和

```
output cout;                          //进位输出
input[5:0] ina,inb;
input cin;
assign {cout,sum} = ina + inb +cin;  //进位与和拼接在一起
```

- [例2] {a,b[5:0],w,5'b101}

```
= {a,b[5],b[2],b[1],b[0],w,1'b1,1'b0,1'b1}
```



用于表示重复的表达式必须为常数表达式！

- 可用重复法简化表达式，如：{4{w}} //等同于{w,w,w,w}
- 还可用嵌套方式简化书写，如：

{b,{5{a,b}}} //等同于{b,{a,b},{a,b},{a,b}}，也等同于{b,a,b,a,b,a,b}

- 在位拼接表达式中，不允许存在没有指明位数的信号，必须指明信号的位数；若未指明，则默认为52位的二进制数！
- 如{1,0} = 64'h00000001_00000000，
  注意{1,0}不等于2'b10

十、运算符的优先级

表5-5 运算符的优先级

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

- 为提高程序的可读性，建议使用括号来控制运算的优先级！
- [例] (a>b)&&(b>c)

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

  (!a)||(a>b)

5 语句

内容概要

- 赋值语句
- 块语句
- 条件语句
- 循环语句
- 结构说明语句
- 编译预处理语句

5 语句

表 Verilog HDL的语句

赋值语句	连续赋值语句	
	过程赋值语句	
块语句	begin_end语句	
	fork_join语句	Quartus II不支持
条件语句	if_else语句	
	case语句	
循环语句	forever语句	MAX+PLUS II不支持
	repeat语句	MAX+PLUS II不支持
	while语句	MAX+PLUS II不支持
	for语句	
结构说明语句	initial语句	Quartus II不支持
	always语句	
	task语句	MAX+PLUS II不支持
	function语句	
编译预处理语句	‘define语句	
	‘include语句	Quartus II不支持
	‘timescale语句	Quartus II不支持

5 语句

- 注：上表中，凡Quartus II不支持的语句是不可综合的，通常用在测试文件中；未注明“Quartus II不支持”的语句均是可综合的。
  - repeat语句和task语句MAX+PLUS II不支持，但Quartus II支持；
  - forever语句、 while语句MAX+PLUS II不支持，Quartus II支持，但通常用在测试模块中；
  - 表中只有4种语句（fork_join， initial，‘include， ‘timescale）是Quartus II不支持的，它们通常用在测试模块中（ModelSim软件支持）。

6 赋值语句和块语句

内容概要

一、赋值语句

二、非阻塞赋值与阻塞赋值的区别

三、块语句

一、赋值语句

- 分为两类：

(1) **连续赋值语句**——assign语句，用于对wire型变量赋值，是描述组合逻辑最常用的方法之一。

[例] assign c=a&b;　//a、b、c均为wire型变量

(2) **过程赋值语句**——用于对reg型变量赋值，有两种方式：

- 非阻塞(non-blocking)赋值方式：

  赋值符号为<=，如 b <= a ;

- 阻塞(blocking)赋值方式：

  赋值符号为=，如 b = a ;

非阻塞赋值与阻塞赋值方式的主要区别

- 非阻塞（non-blocking)赋值方式（ b<= a)：
  - b的值被赋成新值a的操作，并不是立刻完成的，而是在块结束时才完成；
  - 块内的多条赋值语句在块结束时同时赋值；
  - 硬件有对应的电路。
- 阻塞（blocking)赋值方式（ b = a)：
  - b的值立刻被赋成新值a；
  - 完成该赋值语句后才能执行下一句的操作；
  - 硬件没有对应的电路，因而综合结果未知。

❖ 建议在初学时只使用一种方式，不要混用！

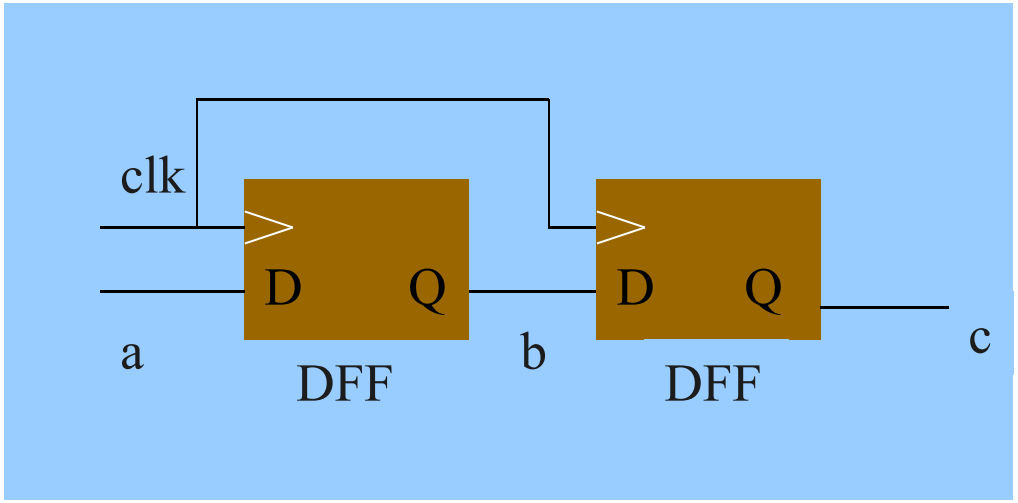
❖ 建议在可综合风格的模块中使用非阻塞赋值！

二、非阻塞赋值与阻塞赋值的区别

1. 非阻塞赋值方式

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

非阻塞赋值在块结束时才完成赋值操作！

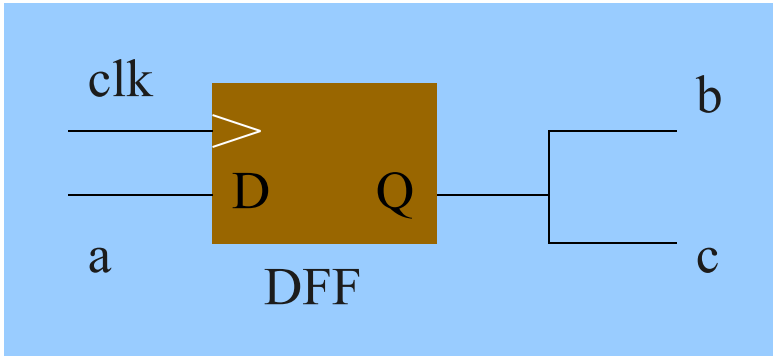


注：c的值比b的值落后一个时钟周期！

2. 阻塞赋值方式

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

阻塞赋值在该语句结束时就完成赋值操作！



注：在一个块语句中，如果有多条阻塞赋值语句，在前面的赋值语句没有完成之前，后面的语句就不能被执行，就像被阻塞了一样，因此称为阻塞赋值方式。

这里c的值与b的值一样 ！

三、块语句

- 用来将两条或多条语句组合在一起，使其在格式上更像一条语句，以增加程序的可读性。
- 块语句有两种：
  - begin_end语句——标识顺序执行的语句
  - fork_join语句——标识并行执行的语句

Quartus II不支持，通常用在测试文件中

1.顺序块

用begin_end标识的块

特点

- 块内的语句是顺序执行的；
- 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顺序块。

顺序块的格式

```
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
```

注：块内声明语句可以是参数声明、reg型变量声明、integer型变量声明、real型变量声明语句。

[例]begin

b = a ;

c = b; //c的值为a的值

end

[例] begin

b = a ;

#10 c = b; //在两条赋值语句间延迟10个时间单位

end

注：这里标识符“#”表示延迟；

在模块调用中“#”表示参数的传递

[例]用顺序块和延迟控制组合产生一个时序波形。

```
parameter d = 50;
reg[7:0] r;
 begin                          //由一系列延迟产生的波形
    # d  r = ' h55 ;
    # d  r = ' hE2 ;
    # d  r = ' h00 ;
    # d  r = ' hF7 ;
    # d  –> end_wave;    //触发事件end_wave
end
```

注：每条语句的延迟时间d是相对于前一条语句的仿真时间而言的！

2. 并行块

用fork_join标识的块

fork语句是不可综合的！

特点

- 块内的语句是同时执行的；
- 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时的仿真时间而言的；
- 延迟时间用于给赋值语句提供时序；
- 当按时间排序在最后的语句执行完或一个disable语句执行时，程序流程控制跳出该并行块。

并行块的格式

```
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
```

注：块内声明语句可以是参数声明、reg型变量声明、integer型变量声明、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]用并行块和延迟控制组合产生一个时序波形。

波形同例5

```
reg[7:0] r;
fork                                    //由一系列延迟产生的波形
    # 50   r = ' h55 ;
    # 100  r = ' hE2 ;
    # 150  r = ' h00 ;
    # 200  r = ' hF7 ;
    # 250  –> end_wave;        //触发事件end_wave
join
```

注：在fork_join块内，各条语句不必按顺序给出！但为增加可读性，最好按被执行的顺序书写！

7 条件语句

内容概要

一、if-else语句

二、case语句

三、使用条件语句注意事项

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

对于每个判定
只有两个分支

一、if-else语句

- 判定所给条件是否满足，根据判定的结果（真或假）决定执行给出的两种操作之一。
- if-else语句有5种形式
  - 其中“表达式”为逻辑表达式或关系表达式，或一位的变量。
  - 若表达式的值为0、或z，则判定的结果为“假”；若为1，则结果为“真”。
  - 语句可为单句，也可为多句；多句时一定要用“begin_end”语句括起来，形成一个复合块语句。

方式1：

```
if（表达式） 语句1;
```

方式2：

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

方式3：

适于对不同的条件，执行不同的语句

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

- 允许一定形式的表达式简写方式，如：
  - if(expression) 等同于if(expression = = 1)
  - if(! expression) 等同于if(expression ! = 1)

- if语句可以嵌套；
- 若if与else的数目不一样，注意用“begin_end”语句来确定if与else的配对关系！

当if与else的数目不一样时，最好用“begin_end”语句将单独的if语句括起来：

if语句的嵌套：

```
if（表达式1）
   if（表达式2）语句1；
   else          语句2；
else
   if（表达式5）语句5；
   else          语句4；
```

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

- ［例］模为60的BCD码加法计数器counter60.v

count60.v

```
1  /* 模为60的BCD码加法计数器 */
2  module count60 (qout, cout, data, load, cin, reset, clk);
3    output[7:0]  qout;
4    output  cout;
5    input[7:0]   data;
6    input load, cin, reset, clk;
7    reg[7:0]    qout;
8    always @ (posedge clk)
9      begin
10       if (reset)          qout = 0;        // 同步复位
11       else if (load)      qout = data;     // 同步置数
12       else if (cin)                        // 若cin = 1，执行加1计数；否则qout保持不变
13         begin
14           if (qout[3:0] == 9)              // 低位是否为9?
15             begin
16               qout[3:0] = 0;                // 是则回0
17               if (qout[7:4] == 5)          // 高位是否为5?
18                 qout[7:4] = 0;              // 是则回0
19               else
20                 qout[7:4] = qout[7:4] +1;   // 高位不为5，则高位加1
21             end
22           else
23             qout[3:0] = qout[3:0]+1;       // 低位不为9，则低位加1
24         end
25     end
26   assign cout = ((qout == 8'h59)&cin)? 1:0; //产生进位输出
27 endmodule
```

在always块内的语句是顺序执行的！

always块语句和assign语句是并行执行的！

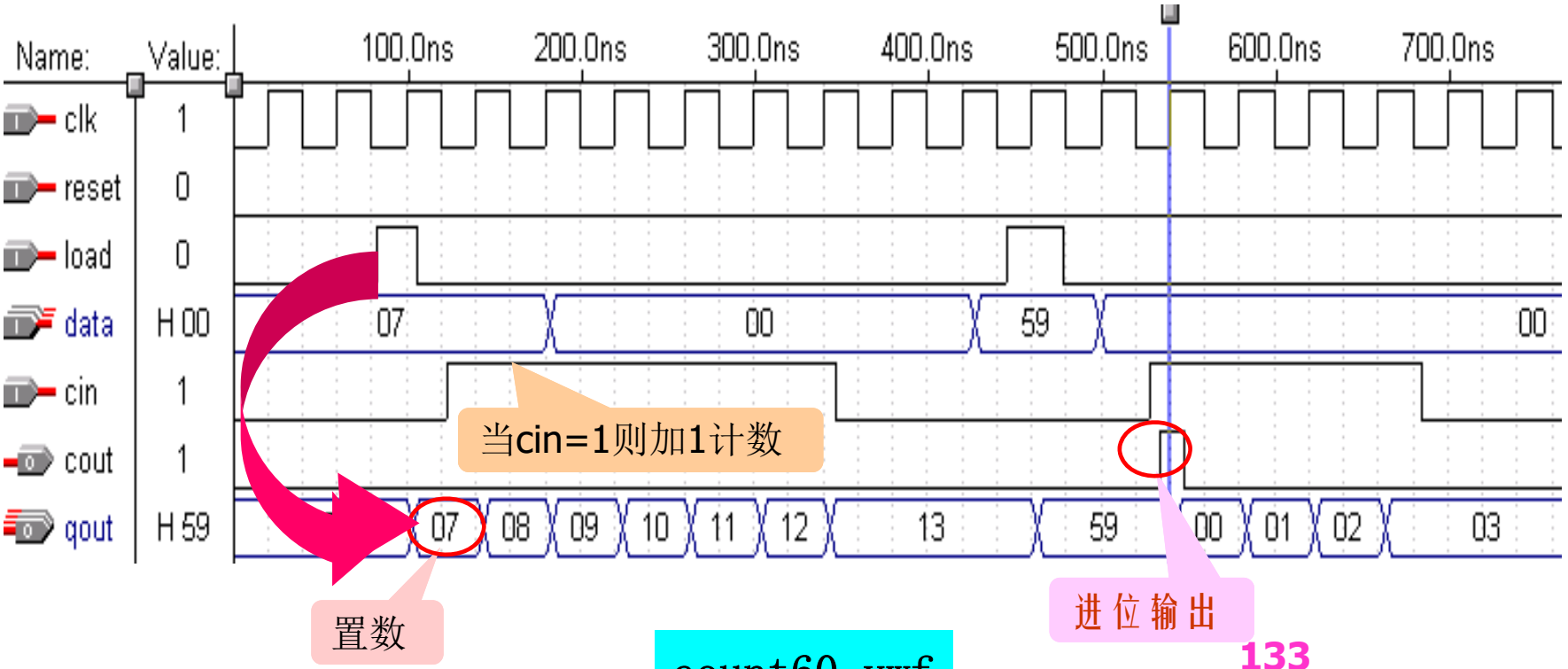
- 注意：if (reset)
  else if (load)
  else if (cin)

  不要写成5个并列的if语句：

  if (reset)
  if (load)
  if (cin)
- 因为这样写则是同时对5个信号reset、load和cin进行判断，现实中很可能出现三者同时为“1”的情况，即5个条件同时满足，则应该同时执行它们对应的执行语句，但5条执行语句是对同一个信号qout赋不同的值，显然相互矛盾。故编译时会报错！

5.7 条件语句

- cin为来自下一级计数器的进位
- always与assign语句是并行执行的！



二、case语句

多分支语句

适于对同一个控制信号取不同的值时，输出取不同的值！

- 当敏感表达式取不同的值时，执行不同的语句。
- **功能**：当某个（控制）信号取不同的值时，给另一个（输出）信号赋不同的值。常用于多条件译码电路（如译码器、数据选择器、状态机、微处理器的指令译码）！
- case语句有5种形式：case，casez，casex

1. case语句

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

case语句与if-else语句有什么区别呢？

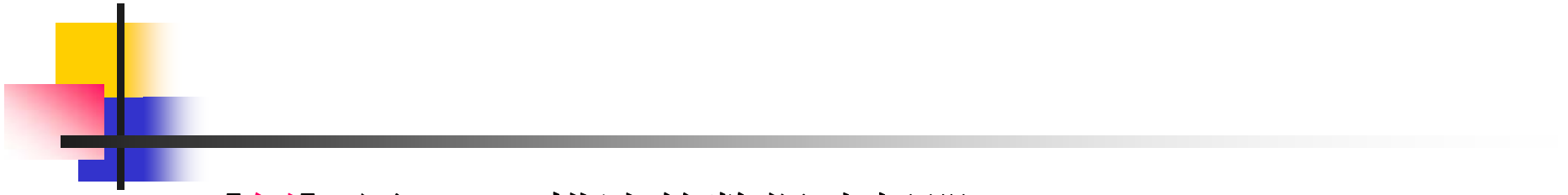
- 说明：
  - 其中“敏感表达式”又称为“控制表达式”，通常表示为控制信号的某些位。
  - 值1~值n称为分支表达式，用控制信号的具体状态值表示，因此又称为常量表达式。
  - default项可有可无，一个case语句里只能有一个default项!
  - 值1~值n必须互不相同，否则矛盾。
  - 值1~值n的位宽必须相等，且与控制表达式的位宽相同。

2. casez与casex语句

是case语句的两种变体

- 在case语句中，分支表达式每一位的值都是确定的（或者为0，或者为1）；
- 在casez语句中，若分支表达式某些位的值为高阻值z，则不考虑对这些位的比较；
- 在casex语句中，若分支表达式某些位的值为z或不定值x，则不考虑对这些位的比较。

❖ 在分支表达式中，可用“？”来标识x或z。



[例] 用casez描述的数据选择器

```
module mux_z(out,a,b,c,d,select);
     output out;
     input a,b,c,d;
     input[5:0] select;
     reg out; //必须声明
    always@ (select[5:0] or a or b or c or d)
    begin
       casez (select)
          4'b???1: out = a;
          4'b??1? : out = b;
          4'b? 1?? : out = c;
          4'b 1??? : out = d;
       endcase
    end
endmodule
```

这里“？”
表示高阻态

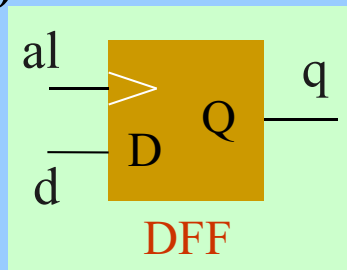
三、使用条件语句注意事项

- **应注意列出所有条件分支，否则当条件不满足时，编译器会生成一个锁存器保持原值！**
- 这一点可用于设计时序电路，如计数器：条件满足时加1，否则保持原值不变。
- 而在组合电路设计中，应避免生成隐含锁存器！有效的方法是在**if**语句最后写上**else**项；在**case**语句最后写上**default**项。

如何正确使用if语句？

生成了不想要的锁存器：

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

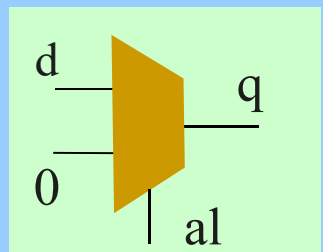


当al为0时，q保持原值！

不会生成锁存器：

[例] 设计一个数据选择器

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



当al为0时，q等于0！

如何正确使用case语句？

生成了不想要的锁存器：

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

当sel为00或11以外的值时，q保持原值！

不会生成锁存器：

[例] 设计一个数据选择器

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

- 避免生成锁存器的原则：
  - 如果用到if语句，最好写上else项；
  - 如果用到case语句，最好写上default项。

8 循环语句

内容概要

一、for语句

二、repeat语句

三、while和forever语句

8 循环语句

- 循环语句分为4种：
  - **for**语句——通过5个步骤来决定语句的循环执行：
    - （1）给控制循环次数的变量赋初值。
    - （2）判定循环执行条件，若为假则跳出循环；若为真，则执行指定的语句后，转到第（5）步。
    - （5）修改循环变量的值，返回第（2）步。
  - **repeat**语句——连续执行一条语句n次
  - **while**语句——执行一条语句，直到循环执行条件不满足；*若一开始条件即不满足，则该语句一次也不能被执行！*
  - **forever**语句——无限连续地执行语句，可用**disable**语句中断！

一、for语句

一般形式

for （表达式1；表达式2；表达式5）语句

简单应用形式

for（循环变量赋初值；循环执行条件；循环变量增值）
　　执行语句

两条语句

- **相当于采用while语句建立的循环结构：**

```
begin
   循环变量赋初值;
   while(循环执行条件)
      begin
         <执行语句>
         循环变量增值;
      end
end
```

8条语句

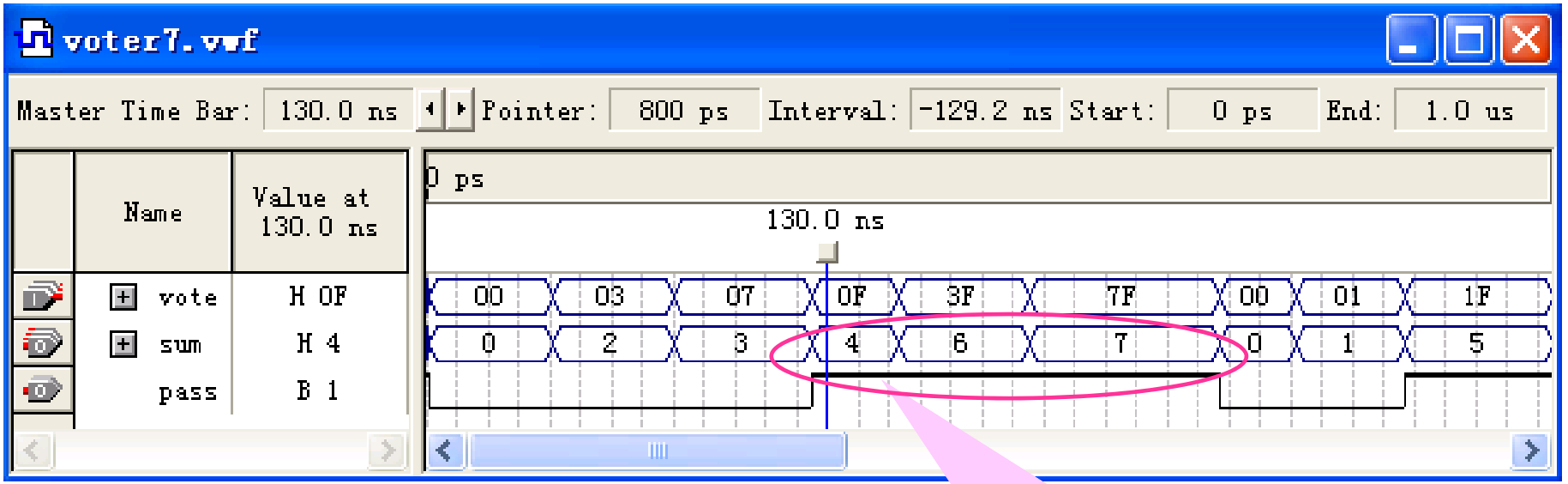
for语句比while语句简洁！

[例]用for语句描述的7人投票表决器：若超过4人（含4人）投赞成票，

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

或写为if(sum[2:0]>=5'd4)

voter7.vwf



超过4人赞成,则pass=1

作业

用for语句描述11人投票表决器：若超过6人（含6人）投赞成票，则表决通过。

[例] 用for语句初始化memory。

```
begin:init_mem
   reg[7:0] tempi;     //存储器的地址变量
   for(tempi = 0;tempi<memsize;tempi = tempi+1)
      memory[tempi] = 0;
end
```

[例] 用for语句实现两个8位二进制数乘法

注：当执行语句有多条时，可用begin_end语句将其括起来！

mult_for.v

mult_for.v - Text Editor

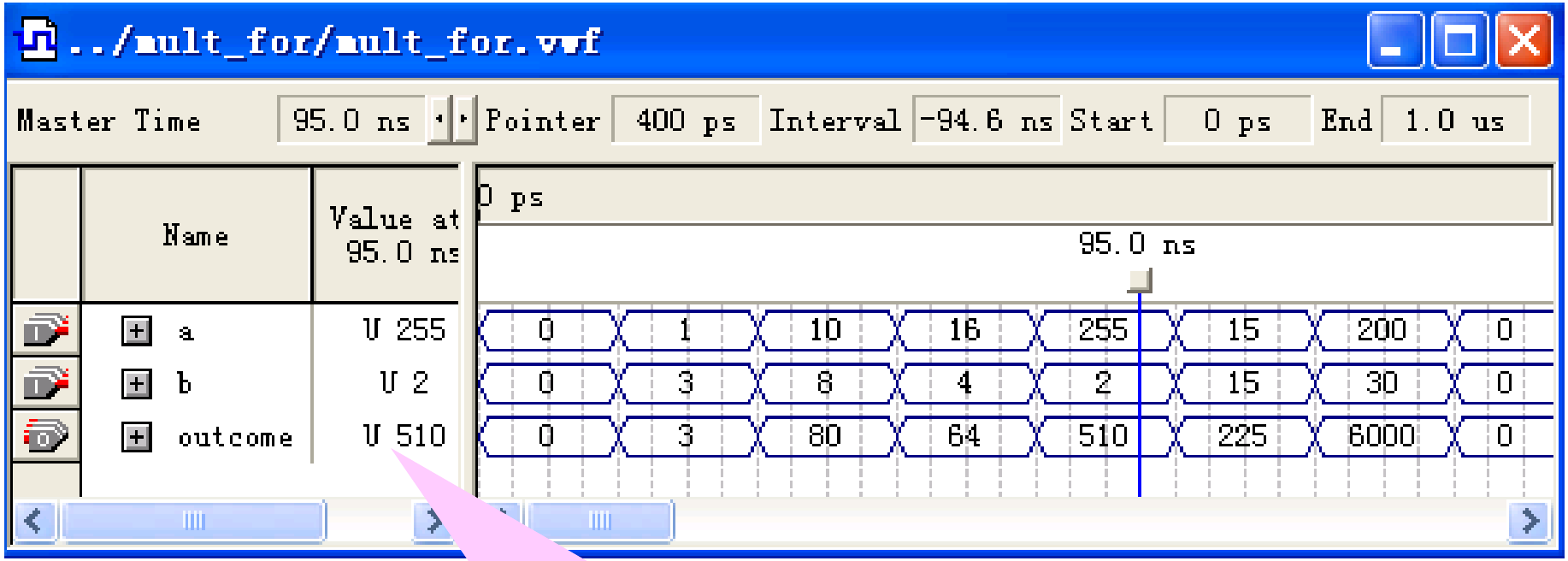
```
//8 bits multiplication.

module mult_for (outcome, a,b);
   parameter size=8;
   output[2*size:1]   outcome;
   input[size:1]      a,b;
   reg[2*size:1]      outcome;
   integer            i;

   always@(a or b)
      begin
         outcome = 0;
         for(i=1;i<=size;i=i+1)
            if(b[i]) outcome = outcome+(a<<(i-1));
      end
endmodule
```

a为被乘数,b为乘数

等同于if(b[i]= =1)

a左移(i-1)位，同时用(i-1)个0填补移出的位

mult_for.vwf（功能仿真）



建议用无符号十进制表示，直观！

MAX + PLUS II不支持，
但Quartus II支持！

二、repeat语句

- 连续执行一条或多条语句n次。

只有部分综合工具可以综合此语句！

格式

```
repeat （循环次数表达式）语句
```

或

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

执行语句为多条语句

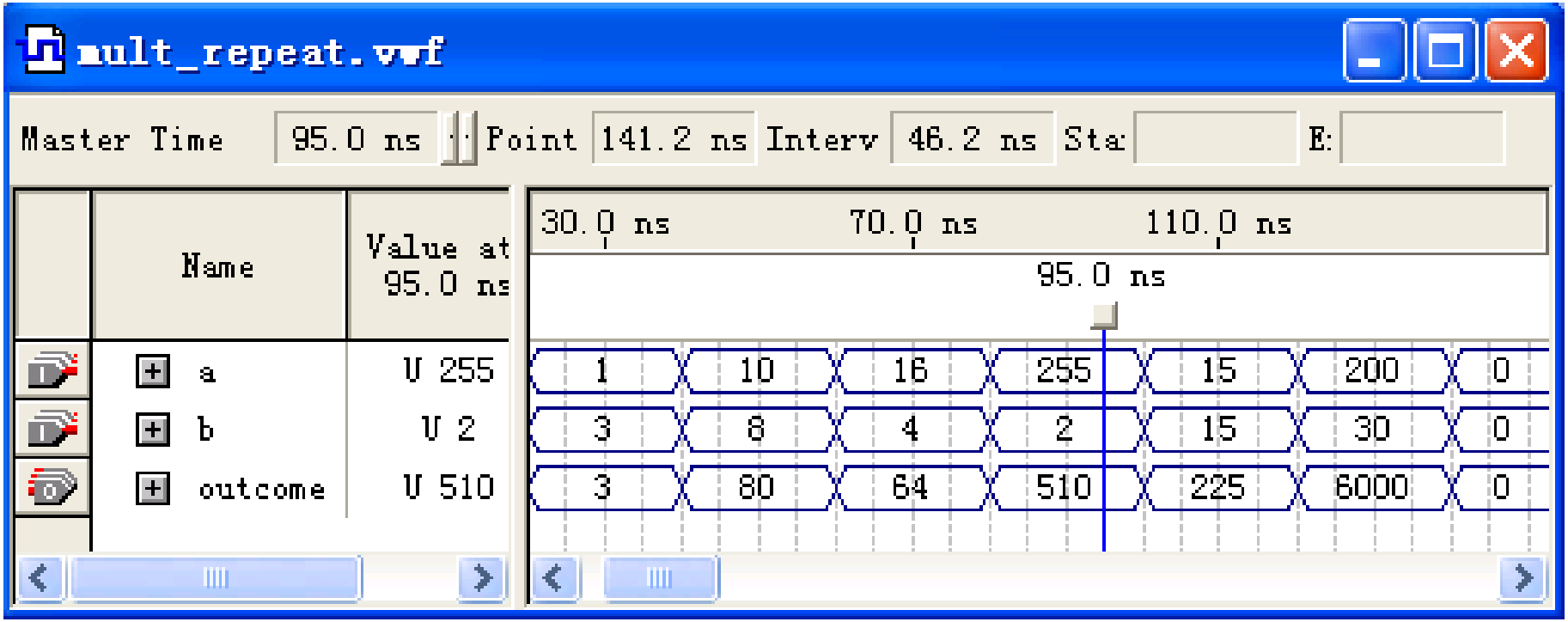
[例]用repeat语句和移位操作实现两个8位二进制数乘法

mult_repeat.v

```
1  //8 bits multiplication using Repeat Statement.
2  module mult_repeat (outcome,a,b);
3    parameter size=8;
4    output[2*size:1]  outcome;
5    input[size:1]     a,b;      //a为被乘数,b为乘数
6    reg[2*size:1]     outcome;
7    reg[2*size:1]     temp_a;   //中间变量，存放操作数a左移一位后的结果
8    reg[size:1]       temp_b;   //中间变量，存放操作数b右移一位后的结果
9    always@(a or b)
10     begin
11       outcome = 0;
12       temp_a = a;
13       temp_b = b;
14       repeat(size)
15         begin
16           if(temp_b[1])        //如果temp_b的最低位为1，则执行下面的加法
17             outcome = outcome + temp_a;
18           temp_a = temp_a<<1; //操作数a左移一位，以便代入上式，求部分积
19           temp_b = temp_b>>1; //操作数b右移一位，以便取temp_b[1]
20         end
21     end
22 endmodule
```


注：不如采用for语句简单！

mult_repeat.vwf（功能仿真）



注：仿真结果同mult_ for.vwf！

MAX + PLUS II均不支持
Quartus Ⅱ均支持！

三、 while和forever语句

1.while语句

- 有条件地执行一条或多条语句。
- 首先判断循环执行条件表达式是否为真。若为真，则执行后面的语句或语句块；然后再回头判断循环执行条件表达式是否为真，若为真，再执行一次后面的语句；如此不断，直到条件表达式不为真。

格式

```
while （循环执行条件表达式）语句
```

或

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

注1：首先判断循环执行条件表达式是否为真，若不为真，则其后的语句一次也不被执行！

注2：在执行语句中，必须有一条改变循环执行条件表达式的值的语句！

注5：**while**语句只有当循环块有事件控制（即**@**（**posedge clock**））时才可综合！

[例]用while语句对一个8位二进制数中值为1的位进行计数。

```
module  count1s_while ( count,rega,clk );
    output[5:0] count;
    input [7:0]   rega;
    input clk;
    reg[5:0]      count;
    always @(posedge clk)
      begin:count1
         reg[7:0] tempreg;          //用作循环执行条件表达式
         count = 0;                 // count初值为0
         tempreg = rega;            // tempreg 初值为rega
         while(tempreg)             // 若tempreg非0，则执行以下语句
            begin
               if(tempreg[0])      count = count+1;
                                   //只要tempreg最低位为1，则 count加1
               tempreg = tempreg >>1;  //右移1位
            end
      end
endmodule
```

如何用for语句改写此程序呢？

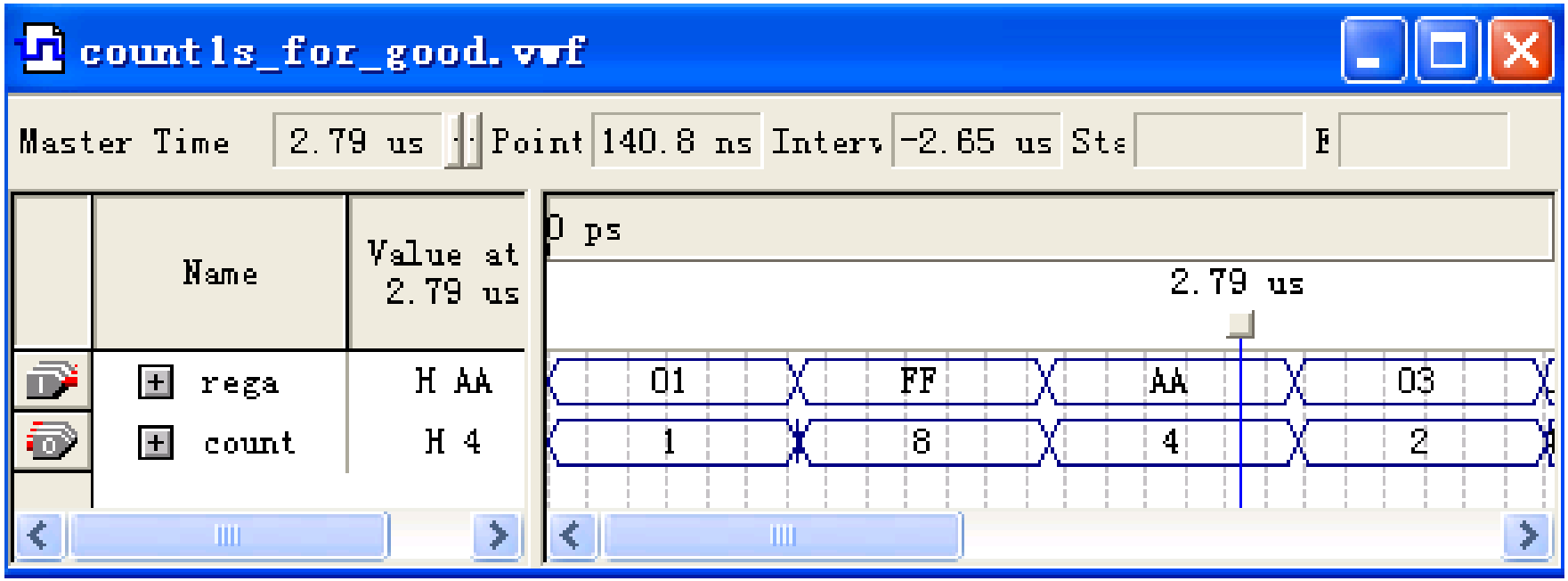
改变循环执行条件表达式的值

［例］用for语句对一个8位二进制数中值为1的位进行计数。

count1s_for_good.v

```
1  /* Count the numbers of 1 in rega[7..0]. It's the simplest.*/
2  module count1s_for_good (count, rega);
3    output [3:0]       count;
4    input  [7:0]       rega;
5    reg[3:0]           count;
6    always @ (rega)
7    begin: block
8      integer i;
9      count = 0;
10     for (i = 0; i <= 7; i = i + 1)
11       if (rega[i] == 1)
12         count = count + 1;
13   end
14 endmodule
```

count1s_for_good.vwf



2. forever语句

一般情况下是不可综合的！常用在测试文件中

- 无条件连续执行forever后面的语句或语句块。

格式

```
forever 语句
```

或

```
forever
  begin
    ……
  end
```

```
initial
  begin : Clocking
    clk = 0;
    #10 forever #10 clk = !clk;
  end
initial
  begin : Stimulus
    ……
    disable Clocking; // 停止时钟
  end
```

- 常用在测试模块中产生周期性的波形，作为仿真激励信号。
- 常用disable语句跳出循环！

注：不同于always语句，不能独立写在程序中，一般用在initial语句块中！

9 结构说明语句

内容概要

一、always块语句

二、initial语句

三、task和function语句

9 结构说明语句

结构说明语句分为4种

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

一、always块语句

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

规则

- 在always块中被赋值的只能是register型变量（如reg，integer，real，time）。
- 每个always块在仿真一开始便开始执行，当执行完块中最后一个语句，继续从always块的开头执行。

格式

always <时序控制> <语句>

注1：如果always块中包含一个以上的语句，则这些语句必须放在begin_end或fork_join块中！

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

注2：always语句必须与一定的时序控制结合在一起才有用！
如果没有时序控制，则易形成仿真死锁！

- [例]生成一个0延迟的无限循环跳变过程——形成仿真死锁！

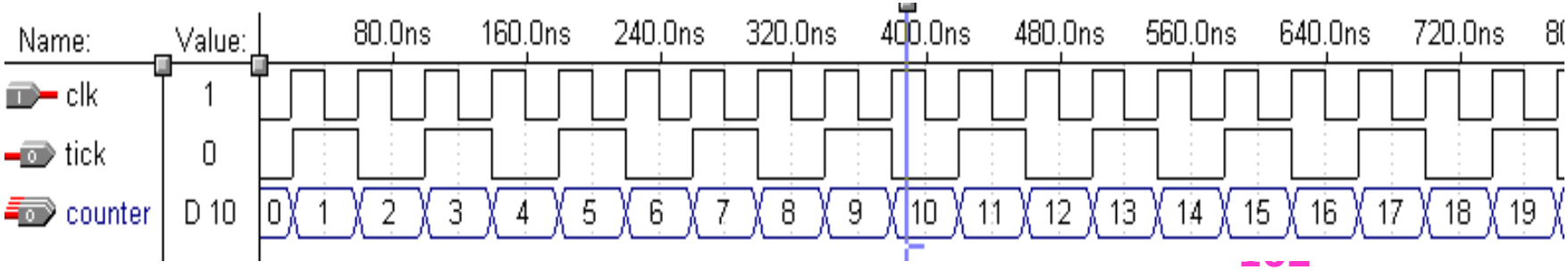
```
always  areg = ~areg;
```

- [例]在测试文件中，用于生成一个无限延续的信号波形——时钟信号

```
‘define half_period 50
 module half_clk_top;
      reg reset, clk;  // 输入信号
      wire clk_out;   // 输出信号
       always  #half_period clk = ~clk;
           ……
 endmodule
```

- [例] 用always块语句产生T'FF和8位二进制计数器。

```
// Use always statement to generate T'FF and binary counter.
module always_demo (counter, tick,clk);
  output [7:0]      counter;
  output            tick;
  input             clk;
  reg[7:0]          counter;
  reg               tick;
  always @ (posedge clk)
    begin
      tick = ~tick;                 // T'FF
      counter = counter + 1;        //binary counter
    end
endmodule
```



always块语句模板

```
always @(<敏感信号表达式>)
  begin
    // 过程赋值语句
    // if语句
    // case语句
    // while，repeat，for循环
    // task，function调用
  end
```

一个变量不能在多个always块中被赋值！

- 敏感信号表达式又称事件表达式或敏感表，当其值改变时，则执行一遍块内语句；
- 在敏感信号表达式中应列出影响块内取值的所有信号！（一般为输入）
- 敏感信号可以为单个信号，也可为多个信号，中间需用关键字or连接！
- 敏感信号不要为x或z，否则会阻挡进程！

常用于描述
时序逻辑

常用于描述
组合逻辑

- always的时间控制可以为沿触发，也可为电平触发。
- 关键字posedge表示上升沿；negedge表示下降沿。

由两个沿触发的always 块

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

由多个电平触发的always 块

```
always@ (a or b or c)
  begin
      ……
  end
```

可综合性问题

- always块语句是用于综合过程的最有用的语句之一，但又常常是不可综合的。为得到最好的综合结果， always块程序应严格按以下模板来编写：

模板1

```
always @ (Inputs) //所有输入信号必须列出，用or隔开
  begin
     ……              //组合逻辑关系
  end
```

模板2

```
always @ (Inputs) //所有输入信号必须列出，用or隔开
  if (Enable)
    begin
      ……              //锁存动作
    end
```

模板5

```
always @ (posedge Clock)  // Clock only
  begin
    ……                          // 同步动作
  end
```

模板4

```
always @ (posedge Clock or negedge Reset)
// Clock and Reset only
  begin
    if (! Reset)          // 测试异步复位电平是否有效
      ……                  // 异步动作
    else
      ……                  // 同步动作
  end                     // 可产生触发器和组合逻辑
```

返回“综合代码的编写标准”

注意

(1) 当always块有多个敏感信号时，一定要采用if - else if语句，而不能采用并列的if语句！否则易造成一个寄存器有多个时钟驱动，将出现编译错误。

```
always @ posedge min_clk or negedge reset)
  begin
    if (reset)
      min<=0;
    else if (min=8'h59) //当reset无效且min=8'h59时
      begin
          min<=0;h_clk<=1;
      end
  end
```

千万别写成if哦！

(2) 通常采用异步清零！只有在时钟周期很小或清零信号为电平信号时（容易捕捉到清零信号）采用同步清零。

MAX+PLUS II 和
Quartus II均不支持！

不可综合！
常用在测试文件中

二、initial语句

格式

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

[例] 利用initial语句生成激励波形。

```
initial
  begin
     inputs = 'b000000;
     #10 inputs = 'b011001;
     #10 inputs = 'b011011;
     #10 inputs = 'b011000;
     #10 inputs = 'b001000;
   end
```

用途

- 在仿真的初始状态对各变量进行初始化；
- 在测试文件中生成激励波形作为电路的仿真信号。

- [例] 对各变量进行初始化。

```
……
parameter size=16;
reg[5:0] addr;
reg reg1;
reg[7:0] memory[0:15];
initial
   begin
        reg1 = 0;
        for(addr=0;addr<size;addr=addr+1);
            memory[addr]=0;
    end
……
```

三、task和function语句

- task和function语句分别用来由用户定义任务和函数。
- 任务和函数往往是大的程序模块中在不同地点多次用到的相同的程序段。
- 利用任务和函数可将一个很大的程序模块分解为许多较小的任务和函数，便于理解和调试。
- 输入、输出和总线信号的值可以传入、传出任务和函数。

1.任务（task）

MAX + PLUS II不支持
但Quartus II支持！

- 当希望能够对一些信号进行一些运算并输出多个结果（即有多个输出变量）时，宜采用任务结构。
- 常常利用任务来帮助实现结构化的模块设计，将批量的操作以任务的形式独立出来，使设计简单明了。

包含定时控制语句的任务是不可综合的！

任务定义

```
task <任务名>;
   端口及数据类型声明语句;
   其他语句;
endtask
```

任务调用

```
<任务名>（端口1, 端口2, ……）;
```

注1：任务的定义与调用必须在一个module模块内！
注2：任务被调用时，需列出端口名列表，且必须与任务定义中的I/O变量一一对应！
注5：一个任务可以调用其他任务和函数。

[例] 任务的定义与调用。

任务定义

```
task my_task;
   input a,b;
   inout c;
   output d,e;
       ……
    <语句>      //执行任务工作相应的语句
       ……
   c = foo1;
   d = foo2;     //对任务的输出变量赋值
   e = foo5;
endtask
```

任务调用

```
my_task（v,w,x,y,z);
```

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

[例] 通过任务调用完成4个4位二进制输入数据的冒泡排序。

```
module sort4(ra,rb,rc,rd,a,b,c,d);
  output[3:0] ra,rb,rc,rd;
  input[3:0] a,b,c,d;
  reg[3:0] ra,rb,rc,rd;
  reg[3:0] va,vb,vc,vd;//中间变量，用于存放两个数据比较交换的结果
  always @ (a or b or c or d)
    begin
      {va,vb,vc,vd} = {a,b,c,d};
    /* 任务的调用 */
      sort2(va,vc); //比较va与vc，较小的数据存入va
      sort2(vb,vd); //比较vb与vd，较小的数据存入vb
      sort2(va,vb); //再比较va与vb，较小的数据存入va（最小值）
      sort2(vc,vd); //再比较vc与vd，较小的数据存入vc（则vd为最大值）
      sort2(vb,vc); //再比较vb与vc谁更小，较小的数据存入vb
      {ra,rb,rc,rd} = {va,vb,vc,vd};
    end
  task sort2; // 任务：比较两个数，按从小到大的顺序排序
    inout[3:0]x,y; //双向类型
    reg[3:0] tmp;
    if(x>y)
      begin
        tmp=x;//x与y变量的内容互换，要求顺序执行，所以采用阻塞赋值方式
        x=y;
        y=tmp;
      end
  endtask
endmodule
```

任务的调用

任务的定义

sort4.v的测试文件

task_Top.v

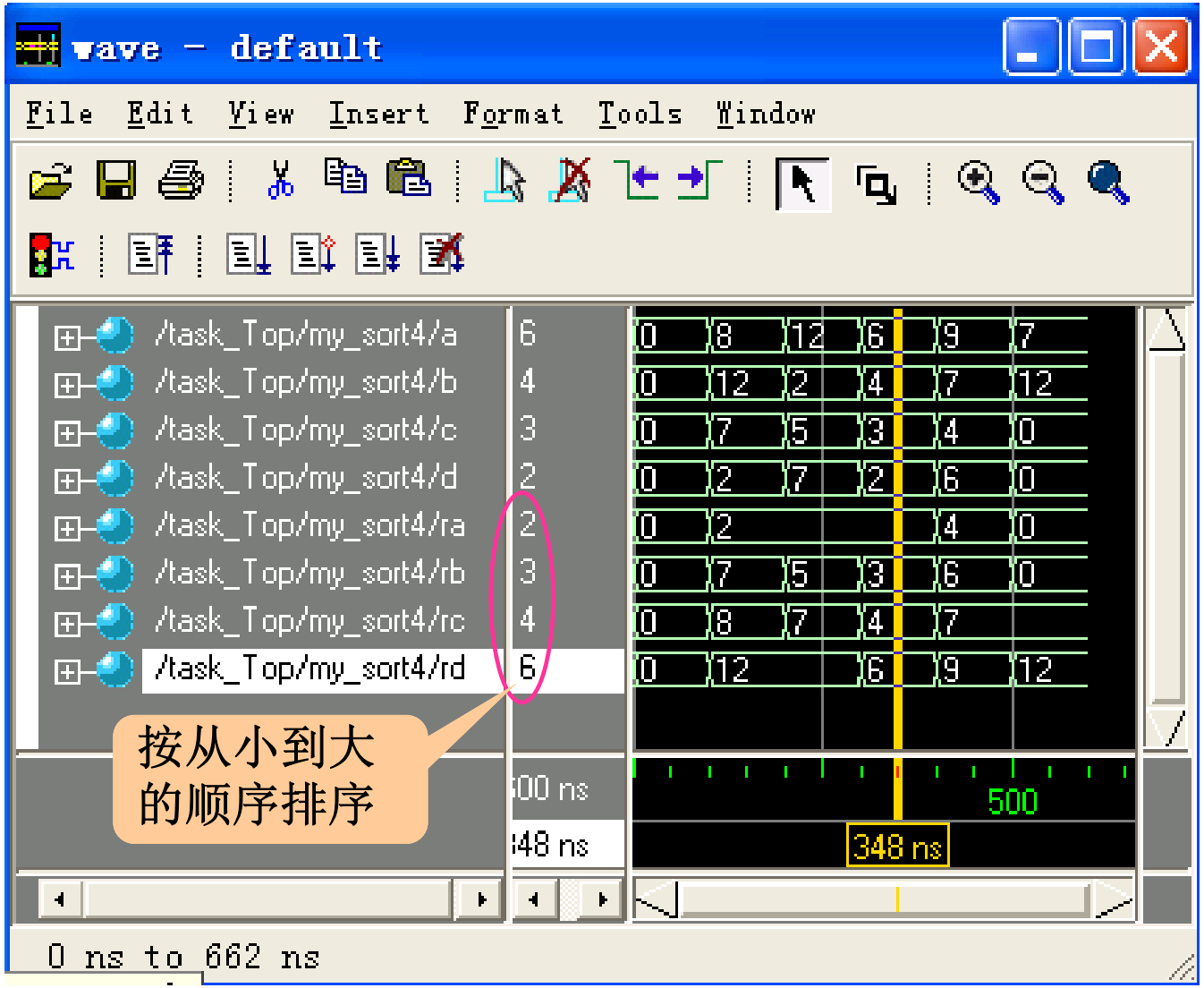
```
1  /* sort4.v的测试文件*/
2  'timescale 1ns/100ps
3  'include "./sort4.v"
4  module task_Top;
5    reg[3:0] a,b,c,d;
6    wire[3:0] ra,rb,rc,rd;
7    initial          //赋予激励
8      begin
9        a=0;b=0;c=0;d=0;
10       repeat(5)//循环5次
11       begin
12         # 100 a={$random}%15;
13               b={$random}%15;
14               c={$random}%15;
15               d={$random}%15;
16       end
17       # 100 $stop;//暂停仿真
18     end
19   sort4 my_sort4(.ra(ra),.rb(rb),.rc(rc),.rd(rd),.a(a),.b(b),.c(c),.d(d));
20 endmodule
```

- $random为系统任务，返回一个52位的带符号的随机数；
- 一般用法为： $random % b

其中b>0，它给出了一个范围在-b+1~b-1之间的随机数。

- {$random}%15通过位拼接操作，产生一个0~14之间的随机数。

sort4.v的仿真波形task_Top.wlf



可以综合!

2.函数（function）

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

函数定义

```
function <返回值位宽或类型说明> 函数名;
    端口声明;
    局部变量定义;
    其他语句;
endfunction
```

缺省则返回1位reg型数据

函数调用

〈函数名〉（〈表达式〉〈表达式〉）

与函数定义中的输入变量对应！

注1：函数的调用是通过将函数作为调用函数的表达式中的操作数来实现的！

```
function[7:0] gefun;                    //函数的定义
   input [7:0] x;
        ……
    <语句>                               //进行运算
    gefun = count;                      //赋值语句
endfunction

assign number = gefun(rega); //对函数的调用
```

内部寄存器

注2：函数在综合时被理解成具有独立运算功能的电路，每调用一次函数，相当于改变此电路的输入，以得到相应的计算结果。

函数的使用规则

- 函数的定义不能包含任何时间控制语句——用延迟#、事件控制@或等待wait标识的语句。
- 函数不能启动（即调用）任务!
- 定义函数时至少要有一个输入参量! 且不能有任何输出或输入/输出双向变量。
- 在函数的定义中必须有一条赋值语句，给函数中的一个内部寄存器赋以函数的结果值，该内部寄存器与函数同名。

［例］利用函数对一个8位二进制数中为0的位进行计数。

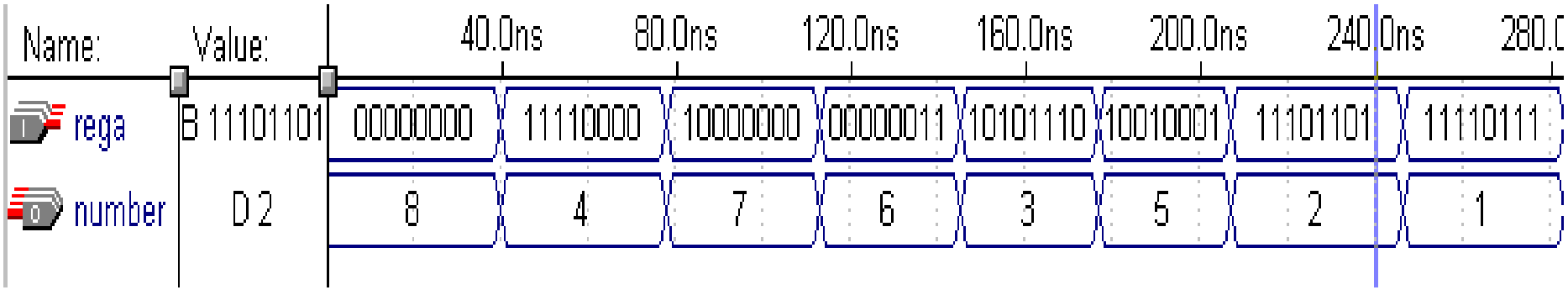
```
// Count the numbers of 0 in rega[7..0].
module count0s_function(number,rega) ;
  output[7:0]   number;
  input[7:0]    rega;

  function[7:0] gefun;                              // definition of function.
    input[7:0]  x;
    reg[7:0]    count;
    integer     i;
      begin
        count = 0;
        for(i = 0;i<=7;i = i + 1)
          if(x[i] == 1'b0) count = count + 1;
        gefun = count;                              // return value of the function.
      end
  endfunction

  assign number = gefun(rega);                      //using the function.
endmodule
```

只有输入变量

内部寄存器

对应函数的输入变量

count0s_function.vwf



- [例]阶乘运算函数

函数定义

```
//factorial function.
module tryfunct(result,clk,reset,n);
function[31:0] factorial;
  input[3:0] op;
  reg[3:0] ina;
  begin
     factorial=op?1:0;
     for(ina=2;ina<=op;ina=ina+1)
      factorial=ina*factorial;
   end
endfunction

  output [31:0] result;
  input[3:0] n;
  input reset,clk;
  reg[31:0] result;

  always @(posedge clk)
    begin
      if(!reset) result<=0;
      else result<=factorial(n);
    end
endmodule
```

内部寄存器

函数名被赋予的值就是函数的返回值！

clk的上升沿触发同步运算

函数的调用

```
// The test module of tryfunct.v
`include "./tryfunct.v"
`timescale 1ns/100ps
`define clk_cycle 50

module tryfuncttop;
  reg[3:0] n,i;
  reg reset,clk;
  wire[31:0] result;
  initial
    begin
     n=0;
     reset=1;
     clk=0;
     #100 reset=0;
     #100 reset=1;
     for(i=0;i<=15;i=i+1)
      begin
        #200 n=i;
      end
   #100 $stop;
   end

  always #`clk_cycle clk=~clk;

  tryfunct tryfunct(.clk(clk),.n(n),.result(result),.reset(reset));

endmodule
```

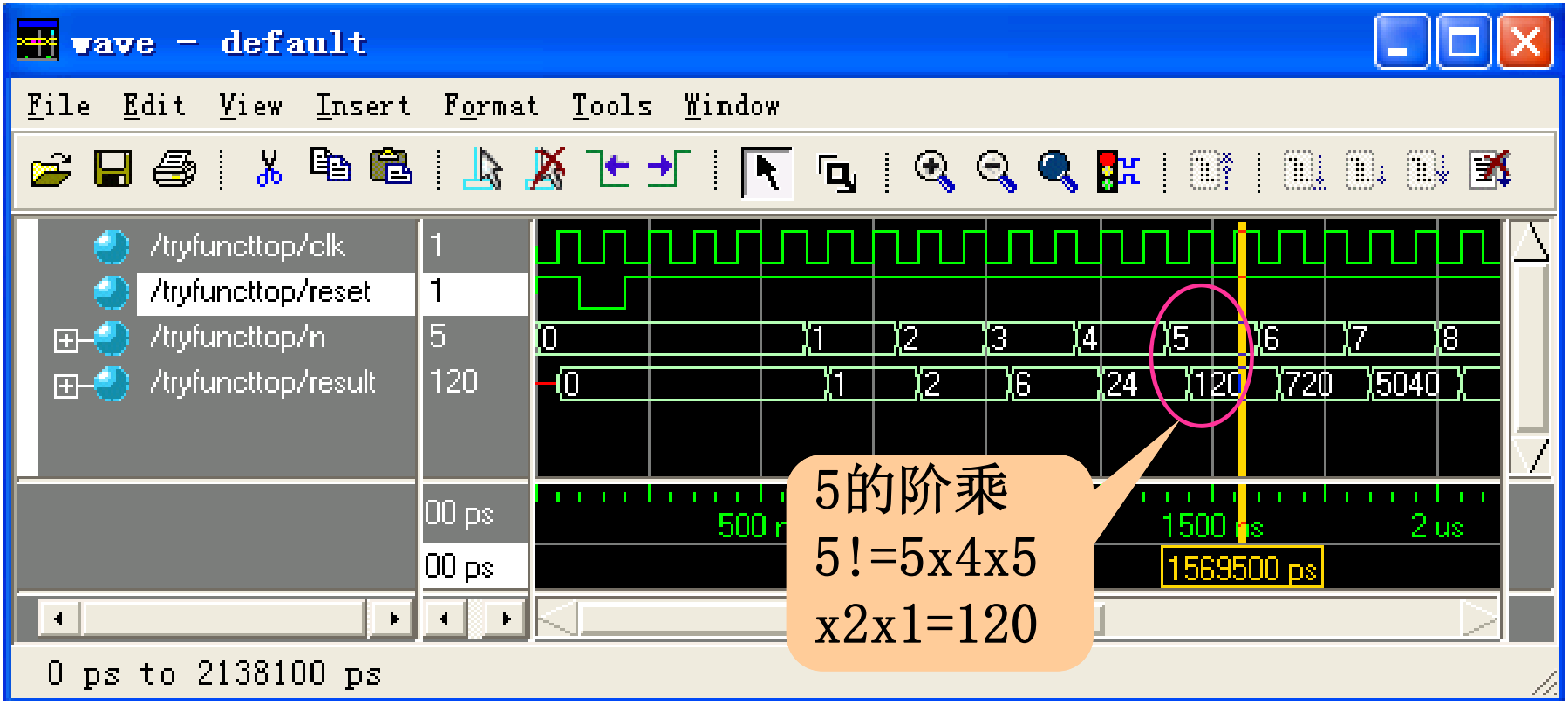
tryfunct.v的测试模块

对各变量进行初始化，并生成激励波形

产生时钟波形

模块元件例化

tryfuncttop.v的仿真波形(用Modelsim编译、仿真)



n的阶乘n!= n·(n-1)·(n-2) ……2·1

5.9 结构说明语句

表 任务与函数的区别

	任务（task）	函数（function)
目的或用途	可计算多个结果值	通过返回一个值，来响应输入信号
输入与输出	可为各种类型（包括inout型）	至少有一个输入变量，但不能有任何output或inout型变量
被调用	只可在过程赋值语句中调用，不能在连续赋值语句中调用	可作为表达式中的一个操作数来调用，在过程赋值和连续赋值语句中均可调用
调用其他任务和函数	任务可调用其他任务和函数	函数可调用其他函数，但不可调用其他任务
返回值	不向表达式返回值	向调用它的表达式返回一个值

10 编译预处理语句

内容概要

一、 `define语句

二、 `include语句

三、 `timescale语句

10 编译预处理语句

- “编译预处理”是Verilog HDL编译系统的一个组成部分。编译预处理语句以西文符号“`”开头——注意，不是单引号“’”！
- 在编译时，编译系统先对编译预处理语句进行预处理，然后将处理结果和源程序一起进行编译。

一、`define语句

- 宏定义语句——用一个指定的标志符（即宏名）来代表一个字符串（即宏内容）。

格式　`define 标志符（即宏名）字符串（即宏内容）

- [例] `define IN ina+inb+inc+ind
- 宏展开——在编译预处理时将宏名替换为字符串的过程。

❖宏定义的作用：

- 以一个简单的名字代替一个长的字符串或复杂表达式；
- 以一个有含义的名字代替没有含义的数字和符号。

关于宏定义的

- 宏名可以用大写字母，也可用小写字母表示；但建议用大写字母，以与变量名相区别。
- 丶define语句可以写在模块定义的外面或里面。宏名的有效范围为定义命令之后到源文件结束。
- 在引用已定义的宏名时，必须在其前面加上符号“丶”！
- 使用宏名代替一个字符串，可简化书写，便于记忆，易于修改。
- 预处理时只是将程序中的宏名替换为字符串，不管含义是否正确。只有在编译宏展开后的源程序时才报错。
- 宏名和宏内容必须在同一行中进行声明！

- 宏定义不是Verilog HDL语句，不必在行末加分号！
- 如果加了分号，会连分号一起置换！

```
[例] module test;
        reg  a,b,c,d,e,out;
        `define expression a + b + c + d;
        assign out = `expression + e;
                ……
```

错误！

经过宏展开后，assign语句为：

```
assign out = a + b + c + d; + e;     //出现语法错误！
```

➢ 在进行宏定义时，可引用已定义的宏名，实现层层置换。

```
[例] module test;
        reg  a,b,c;
         wire out;
        `define aa a + b
        `define cc c +`aa      //引用已定义的宏名`aa 来定义宏cc
          assign out = `cc;
            ……
```

经过宏展开后， assign语句为:

assign out = c + a + b;

二、`include语句

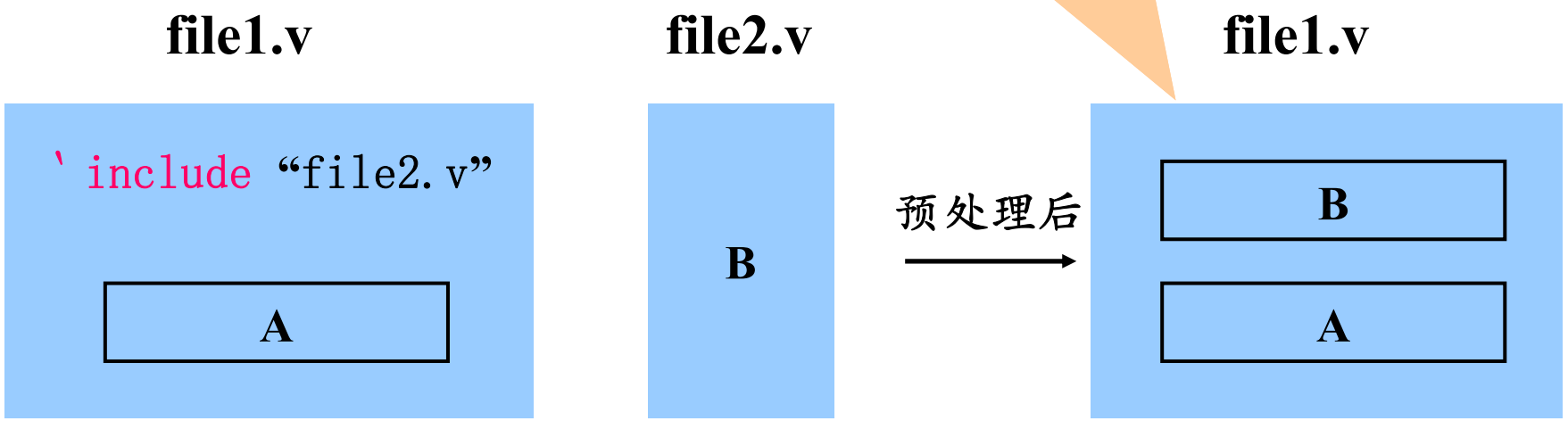
MAX + PLUS II和Quartus II都不支持！通常用在测试文件中。

- **文件包含语句**——一个源文件可将另一个源文件的全部内容包含进来。

格式

```
`include "文件名"
```

将file2.v中全部内容复制插入到`include "file2.v"命令出现的地方



使用`include语句的好处

- 避免程序设计人员的重复劳动！不必将源代码复制到自己的另一源文件中，使源文件显得简洁。

(1) 可以将一些常用的宏定义命令或任务（task）组成一个文件，然后用`include语句将该文件包含到自己的另一源文件中，相当于将工业上的标准元件拿来使用。

(2) 当某几个源文件经常需要被其他源文件调用时，则在其他源文件中用`include语句将所需源文件包含进来。

- [例] 用`include语句设计16位加法器

```
1  // 2.Using include statement mode.
2  `include "adder.v"
3
4  module adder16(cout,sum,a,b,cin) ;                    // 16_bit adder.
5    output              cout;
6    parameter my_size = 16;
7    output[my_size-1:0]    sum;
8    input[my_size-1:0]     a,b;
9    input                  cin;
10   adder #(my_size) my_adder(cout,sum,a,b,cin); //Using adder module.
11 endmodule
```

改变被引用模块adder中的参数size为my_size

adder模块

```
// 1_bit adder.
module adder(cout,sum,a,b,cin) ;
  output                  cout;
  parameter size = 1;
  output[size-1:0]        sum;
  input[size-1:0]         a,b;
  input                   cin;
  assign{cout,sum} = a + b + cin;
endmodule
```

位拼接

关于文件包含的说明

- 一个`include语句只能指定一个被包含的文件；若要包含n个文件，需用n个`include语句。

```
`include “aaa.v” “bbb.v” //非法！
```

```
`include “aaa.v”
`include “bbb.v” //合法！
```

- ` include语句可出现在源程序的任何地方。被包含的文件若与包含文件不在同一子目录下，必须指明其路径！

```
`include “parts/count.v” //合法！
```

- 可将多个`include语句写在一行；在该行中，只可出现空格和注释行。

`include “aaa.v” `include “bbb.v” //合法！

- 文件包含允许嵌套。

file1.v

```
`include “file2.v”
……
……
……
```

file2.v

```
`include “file5.v”
……
……
……
```

file5.v

```
(不包含`include 命令)
……
……
……
```

10 编译预处理语句

三、`timescale语句

MAX + PLUS II和Quartus II都不支持！通常用在测试文件中。

- 时间尺度语句——用于定义跟在该命令后模块的时间单位和时间精度。

格式

`timescale <时间单位> / <时间精度>

- 时间单位——用于定义模块中仿真时间和延迟时间的基准单位；
- 时间精度——用来声明该模块的仿真时间和延迟时间的精确程度。
- 在同一程序设计里，可以包含采用不同时间单位的模块。此时用最小的时间精度值决定仿真的时间单位。

- 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！

```
`timescale 1ps / 1ns        // 非法！
`timescale 1ns / 1ps        // 合法！
```

- 在`timescale语句中，用来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；
- 单位为秒（s）、毫秒（ms）、微秒（us）、纳秒（ns）、皮秒（ps）、毫皮秒（fs）。

[例] `timescale语句应用举例。

```
`timescale 10ns / 1ns //时间单位为10ns，时间精度为1ns
……
reg  sel;
initial
   begin
      #10 sel = 0;  // 在10ns×10时刻，sel变量被赋值为0
      #10 sel = 1;  // 在10ns×20时刻，sel变量被赋值为1
   end
……
```

11 语句的顺序执行与并行执行

内容概要

一、语句的顺序执行

二、语句的并行执行

11 语句的顺序执行与并行执行

一、语句的顺序执行

- 在 “always”模块内，逻辑按书写的顺序执行。
- 顺序语句——“always”模块内的语句。
- 在 “always”模块内，若随意颠倒赋值语句的书写顺序，可能导致不同的结果！
- 注意阻塞赋值语句当本语句结束时即完成赋值操作！

语句的顺序执行与并行执行

[例]顺序执行模块1。

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

对前一时刻的q值取反

对当前时刻的q值取反

a和q的波形反相！

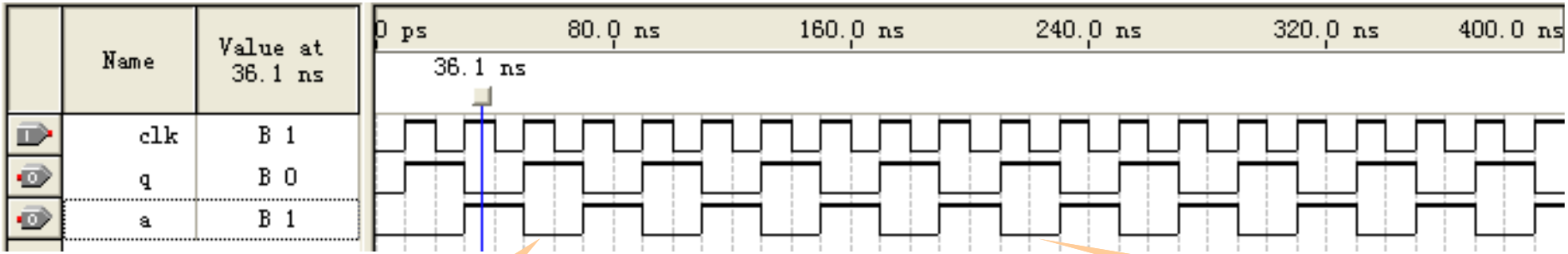
[]顺序执行模块2。

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

对前一时刻的q值取反

对前一时刻的q值取反

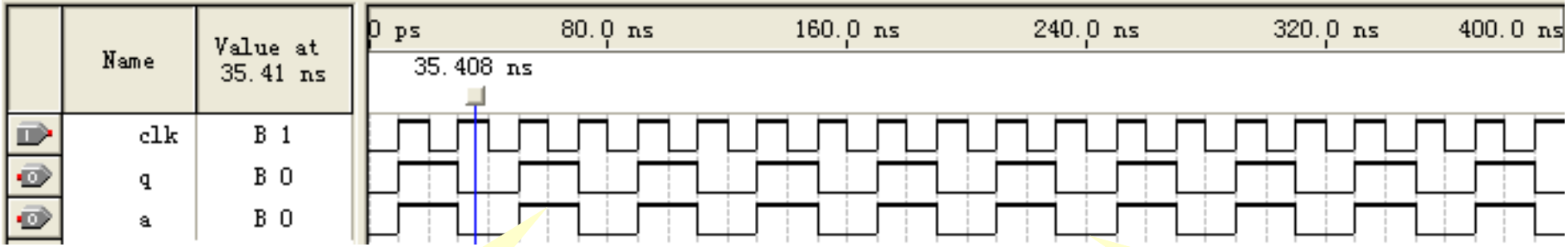
a和q的波形完全相同！



q=~q;
a=~q;

serial1.vwf

a和q的波形反相！



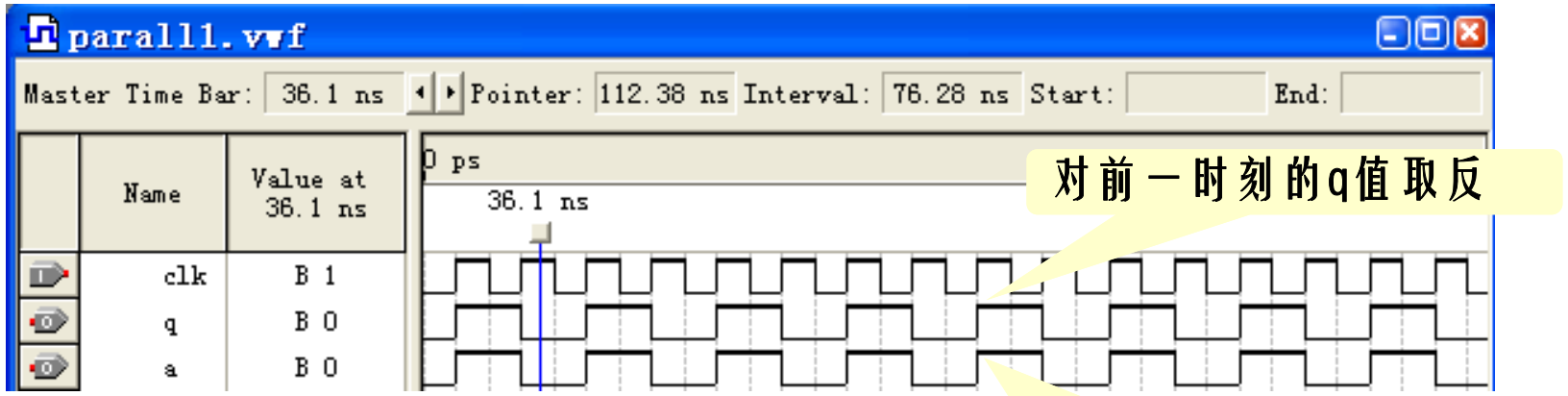
a=~q;
q=~q;

serial2.vwf

a和q的波形完全一样！

二、语句的并行执行

- “always”模块、“assign”语句、实例元件都是同时（即并行）执行的！
- 它们在程序中的先后顺序对结果并没有影响。
- 下面将两条赋值语句分别放在两个“always”模块中，尽管两个“always”模块顺序相反，但仿真波形完全相同，q和a的波形完全一样。



parall1.vwf

[例]并行执行模块1。

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

[例]并行执行模块2。

```
module parall2(q,a,clk);
    output q,a;
    input clk;
    reg q,a;
    always @(posedge clk)
      begin
        a=~q;
      end
    always @(posedge clk)
      begin
        q=~q;
      end
endmodule
```

12 设计技巧

- 建议：

(1) 在进行设计前，一定要仔细分析并熟悉所需设计电路或系统的整个工作过程；合理划分功能模块；并弄清每个模块输入和输出间的逻辑关系！

(2) 在调试过程中，仔细阅读并理解错误信息，随时查阅教材和课件上有关语法，纠正语法错误。

1. 一个变量不能在多个always块中被赋值！

- 这个问题一定要注意！否则编译不能通过。

[例1] 带异步清零、异步置位的D触发器

```
module DFF1(q,qn,d,clk,set,reset);
  output q,qn;
  input  d,clk,set,reset;
  reg    q,qn;
  always @(posedge clk or negedge set or negedge reset)
    begin
      if(!reset)     begin
                     q=0;qn=1;
                     end
      else if(!set)  begin
                     q=1;qn=0;
                     end
      else           begin
                     q=d;qn=~d;
                     end
    end
endmodule
```

正确的写法

注：当某个变量有多个触发条件时，最好将它们放在一个always块中，并用if-else语句描述在不同触发条件下应执行的操作！

错误的写法

```
1  module DFF1_error(q,qn,d,clk,set,reset);
2    output q,qn;
3    input  d,clk,set,reset;
4    reg    q,qn;
5    always @(posedge clk or negedge reset )
6      begin
7        if(!reset)     begin
8                       q=0;qn=1;
9                       end
10       else           begin
11                      q=d;qn=~d;
12                      end
13     end
14   always @(posedge clk or negedge set)
15     begin
16       if(!set)       begin
17                      q=1;qn=0;
18                      end
19       else           begin
20                      q=d;qn=~d;
21                      end
22     end
23 endmodule
```

注：这里q和qn在两个always块中都被赋值！因为always块之间是并行操作，造成某些语句可能是互相矛盾的，所以编译器无所适从，只能报错！

Error: Can't resolve multiple constant drivers for net "q" at DFF1_error.v(5)

Error: Constant driver at DFF1_error.v(14)

Error: Can't elaborate top-level user hierarchy

2、在always块语句中，当敏感信号为两个以上的时钟边沿触发信号时，应注意不要使用多个if语句！以免因逻辑关系描述不清晰而导致编译错误。

- [例2] 在数码管扫描显示电路中，设计一个中间变量，将脉冲信号start转变为电平信号enable。

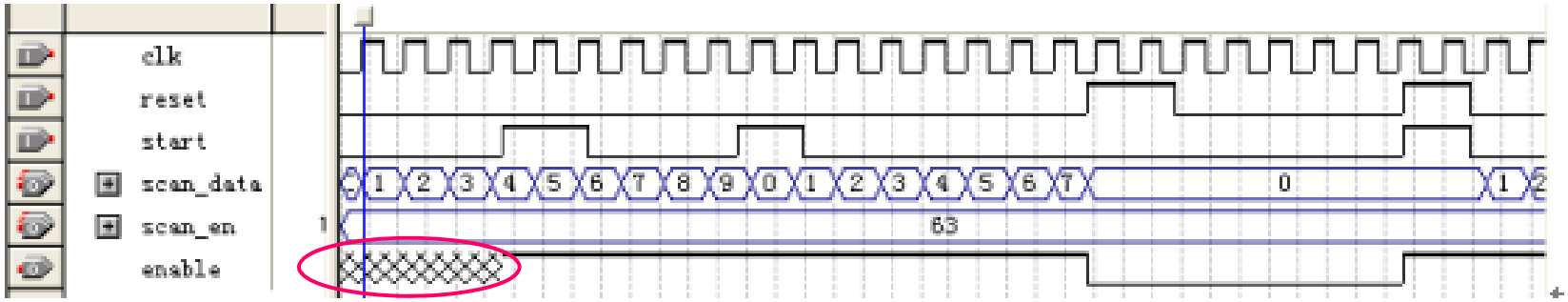
```
always@(posedge start or posedge reset)
   if (reset) enable <=0;
   if (start) enable<=1;
```

错误的写法

- 编译后出现了多条警告信息，指明在语句always @(posedge start or posedge reset)中，变量enable不能被分配新的值！

```
Info: Running Quartus II Analysis & Synthesis
Info: Command: quartus_map --import_settings_files=on --export_settings_files=off clkscan1 -c clkscan1
Info: Found 1 design units, including 1 entities, in source file always_example2.v
Warning: Verilog HDL Always Construct warning at always_example2.v(12): variable enable may not be assigned a new va
Warning: Verilog HDL warning at always_example2.v(22): can't infer register for Procedural Assignment in Always Cons
Warning: Verilog HDL assignment warning at always_example2.v(27): truncated value with size 5 to match size of targe
```

- 其仿真波形如下：

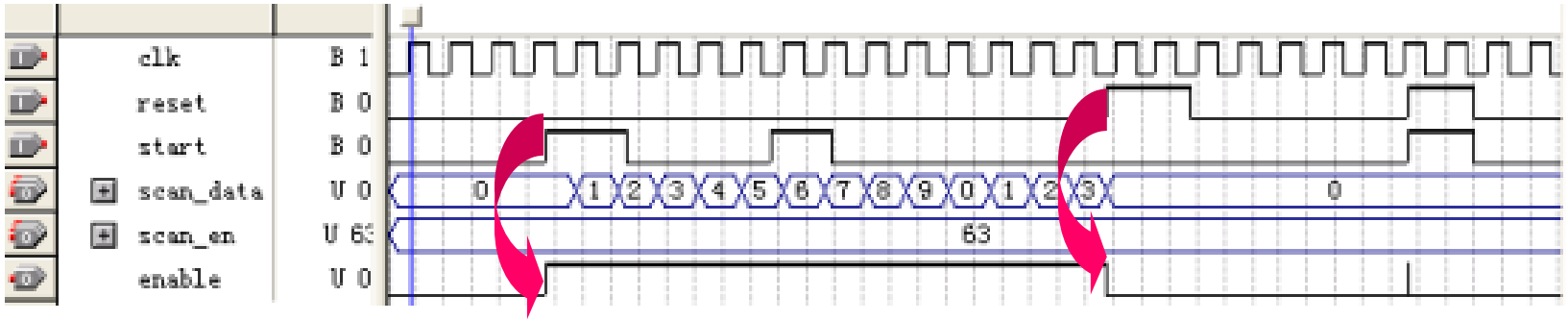


注：由于在最初一段，start和reset均为0，导致enable为不定态，则scan_data开始加1计数（正确情况应是在按下start时scan_data才开始加1计数）。当start和reset同时为1时，enable=1，则scan_data开始加1计数。

正确的写法

```
always@(posedge start or posedge reset)
    if (reset) enable <=0;
    else enable<=1;
```

- 语句“else enable<=1;”隐含了reset无效、且start有效的意思，因此与else if(start) enable<=1;效果一样！
- 正确的仿真波形如下：



注：可见在最初一段，当start和reset均为0时，enable被认为初值为0，则scan_data不计数，保持初值为0；一旦start有效时，则scan_data才开始加1计数。当start和reset同时为1时，先执行的是“if (reset) enable <=0; ”，故enable仍为0，则scan_data保持原值0。

3、当输出信号为总线信号时，一定要在I/O说明中指明其位宽！否则在生成逻辑符号时，输出信号被误认为是单个信号，而没有标明位宽，就不会当成总线信号。

- [例5] 声明一个位宽为5的输出信号**run_cnt**，其类型为reg型变量。

错误的写法

```
output run_cnt;
reg[4:0]run_cnt;
```

正确的写法

```
output[4:0] run_cnt; //这里一定要指明位宽！
reg[4:0]run_cnt;
```

4、当要用到计数器时，一定要根据计数最大值事先计算好所需的位宽！若位宽不够，则计数器不能计到你设定的最大值，当该计数器用作分频时，则输出时钟始终为0，所设计电路将不能按预定功能正常工作！

- [例4]如某同学在做乐曲演奏电路实验时，对乐曲演奏子模块的仿真完全正确，high[5:0]、mid[5:0]、low[5:0]都有输出，但下载时音名显示数码管始终为000。
- 这主要是因为他在分频子模块中clk_4Hz的分频用计数器count_4位宽设置不够，则clk_4Hz输出为0，故音名显示计数器high[5:0]、mid[5:0]、low[5:0]输出始终为0，电路不能正常工作。

错误的写法

```
module f20MHz_to_6MHz_4Hz(clkin,clr,clk_6M,clk_4);
    input clkin,clr;
    output          clk_6M,clk_4;
    reg             clk_6M,clk_4;
    reg[2:0]        count_6M;
    reg[15:0]       count_4;
    parameter       count_6M_width=5;
    parameter       count_4_width=5000000;
    always@(posedge clkin or posedge clr)
      begin
        if(clr) begin
              count_4=0;     clk_4=0;
          end
        else
         begin
            if(count_4==count_4_width-1) //此条件不可能满足!
              begin
                 count_4=0; clk_4=1;
              end
            else
              begin
                 count_4=count_4+1; clk_4=0;
              end
         end
      end
endmodule
```

2^25=8588608，故计数器位宽应为25，应写为[22:0] 。若写成[15:0]，则clk_4一直为0，则下载后数码管显示一直为0，扬声器一直是一个音调

5. 注意程序书写规范：语句应注意缩进，if-else语句注意对齐，应添加必要的注释！

6. 注意区分阻塞赋值和非阻塞赋值的区别。

- 在一个源程序中，要么都采用阻塞赋值语句，要么都采用非阻塞赋值语句，最好不要混合使用，否则可能逻辑关系出错！
- 为易于综合，建议均采用非阻塞赋值语句！