


北京航空航天大学  
Beijing University of Aero. and Astro.  
( BEIHANG University )

2020

# 数字电路与系统 便携式实验

## Verilog硬件描述语言简介 (上)



1



digic

## 教学方法

- 讲座——集中突击最常用的语法和用法
- 实验——“零存整取”，实践Verilog HDL
- 讲义——Verilog HDL硬件描述语言简介
- 参考书——李洪革, 李峭, 何锋. Verilog硬件描述语言与设计. 北京: 北京航空航天大学出版社, 2016

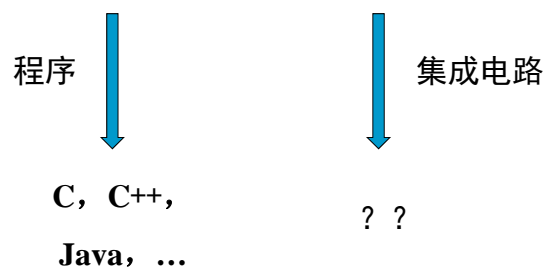
2



## 数学计算与数字系统设计

### ■ 计算 (Computing)

- 我们处于一个数学计算的世界：运算、逻辑、控制.....
- “软件实现” && “硬件实现”



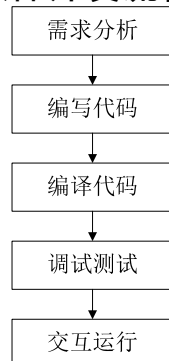
3



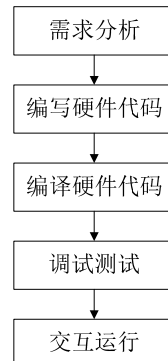
## 数学计算与数字系统设计

### ■ 复杂数字系统设计

#### ➤ 软件开发流程



#### ➤ 硬件开发流程



HDL: Hardware Description Language

4



## 可编程器件发展

- **第一阶段**：上世纪 70年代初到 70年代中，可编程只读存储器(PROM)、紫外线可擦除只读存储器(EPROM)、电可擦只读存储器(EEPROM)3 种，只能完成简单的数字逻辑功能。
- **第二阶段**：上世纪 70年代中到 80年代中，可编程阵列逻辑(PAL)、通用阵列逻辑(GAL)，完成各种逻辑运算功能。
- **第三阶段**：上世纪 80年代到 90年代末，现场可编程门阵列(FPGA)、复杂可编程逻辑设备(CPLD)，能够实现超大规模的电路，编程方式也很灵活。
- **第四阶段**：上世纪 90年代末到目前，可编程片上系统（SoC、PSoC、SoPC）、片上网络（NoC），不仅实现了软件需求和硬件设计的完美结合，还实现了高速与灵活性的完美结合。

5



## 硬件描述语言

- HDL：形式化的方法来描述数字电路和系统的语言。
- 设计者利用HDL可以从上层到下层（从抽象到具体），逐层描述自己的设计思想，用一系列分层次的模块来表示极为复杂的数字系统。然后利用EDA工具逐层进行**功能仿真**验证，然后然后自动**综合**成门级电路网表。接下来用专用集成电路（ASIC）或现场可编程门阵列（FPGA）自动**布局布线**工具把网表转换成具体的电路布线结构，并进行**时序仿真**。

### ❖ HDL

- ✓ Verilog
- ✓ VHDL

6



## 硬件描述语言

- HDL：形式化的方法来描述数字电路和系统的语言。
- **综合**：本课程指将设计输入编译成由与门、或门、非门、RAM、触发器等基本逻辑单元组成的逻辑连接网表，而并非真实的门级电路。广义上讲，综合是指将较高级抽象层次的描述转化成较低层次的描述。
- **功能仿真**：前仿真，对设计的电路进行逻辑功能验证，此时的仿真没有延迟信息。

7



## 硬件描述语言

- HDL：形式化的方法来描述数字电路和系统的语言。
- **布局布线**：把逻辑映射到目标器件的固有硬件结构资源中，并决定逻辑的最佳布局。真实具体的门级电路需要利用制造商的布局布线功能，根据综合后生成的标准门级结构网表来产生。
- **时序仿真**：后仿真，将布局布线的延时信息反标注到设计网表中检测有无时序违规现象。此时的仿真包含延迟信息，**真实**地反映芯片的实际工作情况。

8



## 硬件描述语言

### ■ 使用HDL的好处：设计的可移植性

- 电路的逻辑功能容易理解；
- 便于计算机对逻辑进行分析处理；
- 把逻辑设计与具体电路的实现分成两个独立的阶段来操作；
- 逻辑设计与实现的工艺无关；
- 逻辑设计的资源积累可以重复利用；
- 可以由多人共同更好更快地设计非常复杂的逻辑电路（几十万门以上的逻辑系统）。

9



## 硬件描述语言

### ■ 现代数字系统设计方法

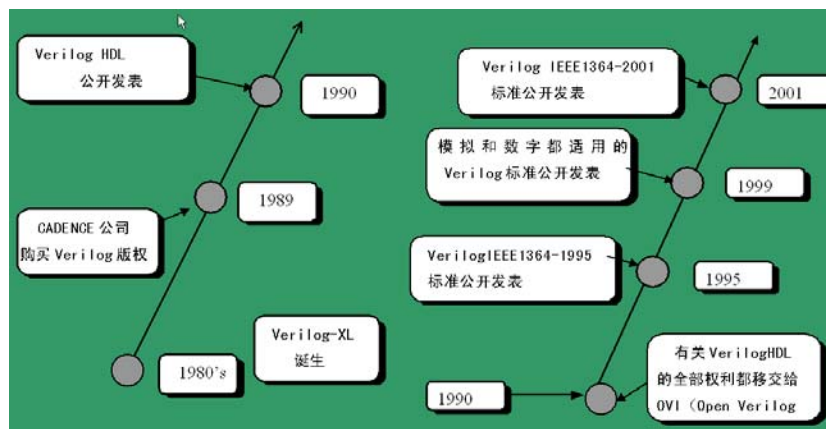
- |                      |                    |
|----------------------|--------------------|
| - 选用合适的 EDA仿真工具；     | - 选用合适的基本逻辑元件库和宏库  |
| - 选用合适电路图输入和HDL编辑工具； | - 租用或购买必要的IP核；     |
| - 逐个编写可综合HDL模块；      | - 选用合适的综合器；        |
| - 逐个编写HDL测试模块；       | - 综合得到门级电路结构；      |
| - 逐个做HDL 电路逻辑仿真；     | - 布局布线，得到时延文件；     |
| - 编写HDL总测试模块；        | - 后仿真；             |
| - 做系统电路逻辑总仿真；        | - 定型，FPGA编码或ASIC投片 |

10



## 硬件描述语言

### ■ Verilog HDL:



11



## Verilog

### ■ Verilog与C语言

C	Verilog
sub-function	module, function, task
if-then-else	if-then-else
Case	Case
{,}	begin, end
For	For
While	While
Break	Disable
Define	Define
Int	Int
Printf	monitor, display, strobe

12



## Verilog

### ■ Verilog与C语言

C	Verilog	功能
*	*	乘
/	/	除
+	+	加
-	-	减
%	%	取模
!	!	反逻辑
&&	&&	逻辑且
		逻辑或
>	>	大于
<	<	小于
>=	>=	大于等于
<=	<=	小于等于
==	==	等于
!=	!=	不等于
~	~	位反相
&	&	按位逻辑与
		按位逻辑或
^	^	按位逻辑异或
~^	~^	按位逻辑同或
>>	>>	右移
<<	<<	左移
?:	?:	同等於if-else敘述

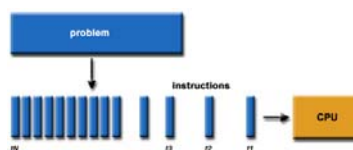
13



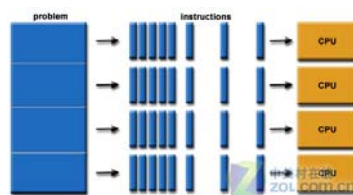
## Verilog

### ■ Verilog与C语言

#### ➤ 单核程序执行



#### ➤ 多核程序执行



14

# Verilog

- Verilog与C语言
  - 硬件并行处理结构

15

# Verilog HDL模块结构

一个模块代表一个基本的功能单元

- Verilog HDL的基本设计单元是**模块**，一个模块是由两部分组成的，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。

```

module block(a,b,c,d);
  input a,b;
  output c,d;
  // ...
endmodule

```

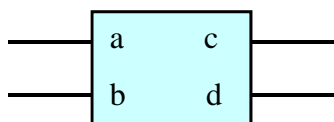
- 程序代码——功能实现
- 电路图框图

16





## Verilog HDL模块结构



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

- 电路图符号的引脚也是程序模块的端口，程序模块描述了电路图符号所实现的逻辑功能。例如：第2, 3行代码说明接口的信号流向，第4, 5行代码说明了模块的逻辑功能。
- Verilog结构位于module和endmodule声明语句之间，每个Verilog程序包括4个主要部分：端口声明，I/O声明，“数据类型”（内部信号）声明，功能定义。

17



## Verilog HDL模块结构

此为VerilogHDL-1995标准风格，VerilogHDL-2001标准将端口、I/O和参数定义组合起来书写。

### ■ Verilog模块的 端口声明 和 I/O声明

#### ➢ 模块的端口声明

```
module <模块名> (口1, 口2, 口3, ...);
```

端口声明列表

端口声明列表 中的 端口名——标识模块内部信号与外部的接口

#### ➢ I/O声明

- 输入口： input[信号位宽-1: 0]端口名i;
- 输出口： output[信号位宽-1: 0]端口名j;
- 输入/输出口： inout[信号位宽-1: 0]端口名k;

18



## Verilog 模块结构

### ■ Verilog模块基本结构

module <模块名> 端口声明

I/O声明 (input, output, inout);

参数定义 (可选, 常用于定义常量)

“数据类型” 声明 (内部信号说明)

连续赋值语句 (**assign**)

过程块——行为描述语句 (initial, **always**)

任务和函数 (task, function)

低层次模块的实例

延时说明块

endmodule



A Real Example

19



## Syntax Summary

### Module Structure

```
module M (P1, P2, P3, P4);
  input P1, P2;
  output [7:0] P3;
  inout P4;

  reg [7:0] R1, M1[1:1024];
  wire W1, W2, W3, W4;
  parameter C1 = "This is a string";

  initial
  begin : BlockName
    // Statements
  end

  always
  begin
    // Statements
  end

  // Continuous assignments...
  assign W1 = Expression;
  wire (Strong1, Weak0) [3:0] #(2,3) W2 = Expression;

  // Module instances...
  COMP U1 (W3, W4);
  COMP U2 (.P1(W3), .P2(W4));
```

20



## Verilog 模块结构——当前的例子

### ■ 接口声明的风格有-1995和-2001两种

➤ 如：

```
module and_gate (
    input wire a, b,
    output wire c
);
    and(c,a,b); // and g1(c, a, b);
endmodule
```

21



## Verilog 模块结构——当前的例子

### ■ 逻辑门的描述方法（本课程采用第一种）

```
/* 第一种方法是逻辑门单元 */
module and_2( in1, in2, out );
    input in1, in2; // 输入信号
    output out; // 输出信号
    and m1( out, in1, in2 );
    //这是逻辑门实例化描述法,and是“逻辑与”的关键字,m1是设计者自定义的实例化名,括号内是
    //逻辑门的输出和输入端口
endmodule

/* 第二种方法是连续赋值法 */
module and_2( in1, in2, out );
    input in1, in2; // 输入信号
    output out; // 输出信号
    assign out = in1 & in2;
    //assign是连续赋值语句的关键字,在后续章节中讲述
endmodule
```

22



## Verilog 模块结构——当前的例子

### ■ 软件仿真测试

- 注意 “`” 与 “`” 的不同（前者在键盘的左上角）
- 再次注意模块的实例化方法

```
`timescale 1ns/100ps
module and2_tb;
  reg in_a, in_b ;
  wire out_c ;
  initial begin
    in_a = 0 ; in_b = 0;
    #50
    // ...
  end
  and2 and2_u1 (.a(in_a), .b(in_b), .c(out_c) );
endmodule
```

23



## 模块结构



### ■ Verilog HDL模块的实例化

例如：

思考：如果设计修改添加了若干端口，“端口命名法”的优势...

#### ➢ 方法1:

```
Block1 dut1(in_a, in_b, in_c, ou_d, ou_e);
```

#### ➢ 方法2:

```
Block1 dut2(.a(in_a), .d(ou_d), .b(in_b));
```

- 比较：方法2不必严格按照端口的顺序对应，提高了程序的**可读性**和**可移植性**；
- 而且，方法2（端口命名法）**可以悬空**某个端口而不用特别说明。

24

## 模块结构——功能定义

### ■ Verilog模块内容---功能定义

- 用assign 声明语句  
assign a=b & c;
- 用实例元件  
and #2 ul (q, a, b);
- 用 always块  
always @(posedge clk or clr)  
begin  
if (clr) q <=0;  
else if (en)q <=d;  
end

过程 & 行为

✓组合逻辑

✓时序逻辑  
✓组合逻辑

25

## 模块结构——功能定义

### ■ 注意：

- Verilog模块中的所有的
  - 过程块（包含always和initial块）
  - 连续赋值语句（assign）
  - 实例化元件

都是**并行**的！

- 它们通过变量名相互连接
- 在同一模块中，功能实现 与 不同功能的代码小节的出现次序**无关**

26



## 变量

### ■ 变量

- 线网（连线）——wire
- reg变量
  - 与寄存器有关，但不能等价于寄存器
  - 过程块中被赋值的变量必须是reg类型
- 位宽，如： `reg [3:0] B;`

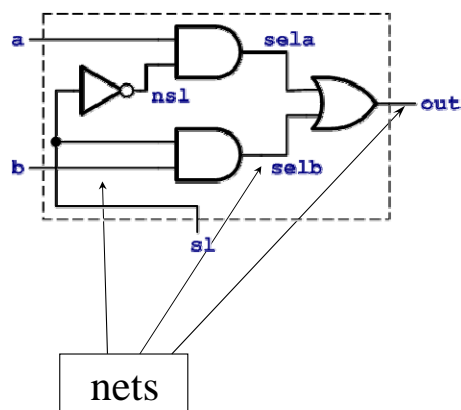
### ■ 数组

27



### ■ Net (网连接):

- 由模块或门驱动的连线。
- 驱动端信号的改变会立刻传递到输出的连线上。
- 例如：右图上，selb信号的改变，会自动地立刻影响或门的输出。

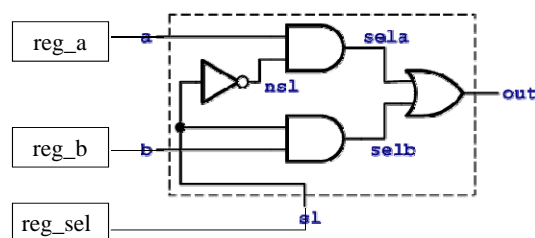


28



## 寄存器（register）类型变量

- register 型变量能保持其值，直到它被赋于新的值；
- 实现 —— reg型变量是 锁存器/触发器 的抽象
- 测试（test bench）——用于行为建模，产生测试的激励信号。
- 常用行为语句结构来给寄存器类型的变量赋值。



29



## 常量——整数、实数和逻辑值

### ■ Verilog HDL语言中常数可以是整数或实数

表 达 方 式	说 明	举 例
<位宽> ' <进制> <数字>	完整的表达方式	8'b11001010 8'hca 8'HCa
'<进制> <数字>	缺省位宽，则位宽由机器系统决定，至少32位	'hca
<数字>	缺省进制为十进制，位宽默认为32位	202

#### 说明：

- <位宽> 指对应二进制数的宽度
- <数字> 可以是所选基数的任何合法的值，包括 **不定值 x** 和 **高阻值 z** 。

30

digitC

(续)

注意区别:

```

define FSM_STA_IDLE 2b00
include "counter.v"

```

assign test\_var = 'b1100\_1010;

又如: 打字机  
(Courier) 字体

'b 'B 'd 'o

例如:

- 8'b11001010, 8'b1100\_1010, 8'hCA //单引号
- 4'b10x0, 12'dx // ✓ (仅在modelsim)
- 12'd2x // "err, X and Z values not allowed in decimal constants."
- "z"的另一种表达方式为"? " // 本人不太喜欢这种表示

31

digitC

## ■ 常量 (续)

■ 用十进制表示的实常数, 也可以用浮点数科学计数法表示

如: 32e-4 (表示0.0032)  
4.1E3 (表示 4100)

Be Careful! 与浮点数标准和存储方法有关...

❖ 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32位。

➢ 10 = 32'd10 = 32'b1010  
➢ -1 = -32'd1 = 32'b1111.....1111 = 32'hFFFFFFFF

32





## 运算符

- 算术运算符: +, -, ×, /, %
- 赋值运算符: =, <=
- 关系运算符: >, <, >=, <=
- 等式运算符: ==, !=, ===, !==
- 逻辑运算符: &&, ||, !
- 条件运算符: ?:
- 位运算符: ~, |, ^, &, ^
- 移位运算符: <<, >>
- 拼接运算符: {}
- 缩减运算符
- 其它

■ && 与 &, || 与 | 区别?

4'b1101 && 4'b0101=?

4'b1101 & 4'b0101=?

4'b1101 && 4'b0101= 1'b1

4'b1101 & 4'b0101= 4b'0101

33



## 运算符

- 算术运算符: +, -, ×, /, %
- 赋值运算符: =, <=
- 关系运算符: >, <, >=, <=
- 等式运算符: ==, !=, ===, !==
- 逻辑运算符: &&, ||, !
- 条件运算符: ?:
- 位运算符: ~, |, ^, &, ^~
- 移位运算符: <<, >>
- 拼接运算符: {}
- 缩减运算符
- 其它

■ reg[3:0] A,B;

■ reg[5:0] C;

■ C={ A[3:1], B[3:1] }

34



## ■ 位拼接运算符可以使用“重复拼接”

➤ 例如：

`{ 4 {w} } // 等同于 { w, w, w, w }`

`{ b, { 3 {a, b} } // 等同于 { b, a, b, a, b, a, b }`

35



## 赋值语句 之 连续赋值语句

### ■ 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。

`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 ;`

36



## 连续赋值

```
assign [drive_strength] [#(delay)]
    net_lvalue = express;
```

- 赋值式的左边必须是一个线网标量或矢量，或是标量或矢量线网变量利用拼接运算符“{ }”的组合，**不能**是寄存器变量
- 模块的端口和端口表达式存在一种隐含连续赋值的关系

37



## 过程赋值——always过程块

- 不断重复执行的结构化语句，符合标准化结构编写的代码是可综合的

```
always @( 时序控制 ) <语句>
```

如果没有时序控制，则always语句将会生成一个死锁。如：

**always areg=~areg;**

但如果加上时序控制，就会变成很有用的信号发生器：

**always #half\_period areg=~areg;**

可以用边沿触发或电平敏感触发等多种时序控制方式。

38



## 过程赋值——always语句（续）

- 边沿触发例子：一般用于时序逻辑

```
reg[7:0] counter;
reg tick;
always @(posedge areg)
begin
    tick=~tick;
    counter=counter+1;
end
```

多个上升沿触发：

**always@(posedge clock or posedge reset)**

下降沿触发：

**always@(negedge clock)**



## 过程赋值——always语句（续）

- 电平触发例子

```
always @( a or b or c)
begin
    if()
    else if()
    else
end
```

电平触发的**always**一般用于描述较为复杂的组合逻辑电路。  
(利用行为描述语句)



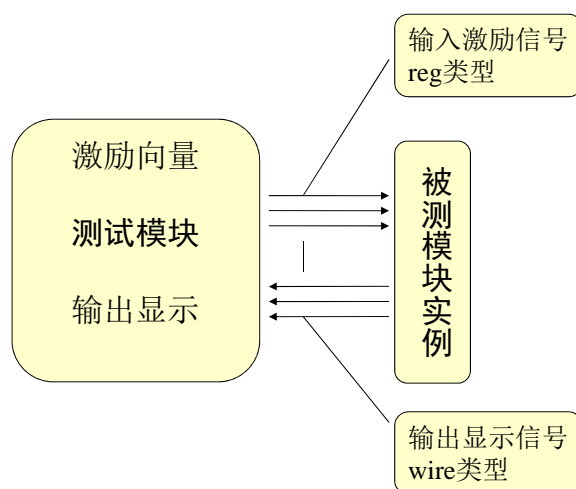
## 测试平台

- 测试平台（Test Bench或Test Fixture），或者称为测试基准，它为测试或仿真一个Verilog HDL程序搭建了一个平台，我们给被测试的模块施加激励信号，通过观察被测试模块的输出响应，从而判断其逻辑功能和时序关系正确与否。

41



## 测试



“测试平台”（testbench）

42



## 测试平台程序的一般结构

```

module 仿真测试模块名; //顶层模块, 无端口列表
    各种输入、输出变量定义
    数据类型说明
    //其中激励信号定义为reg型, 显示信号定义为wire型
    integer
    parameter
    待测试模块实例 // unit-under-test
    激励向量定义
        (使用 always、initial过程块; function, task 结构等;
         if-else, for, case, while, repeat, disable等控制语句)
    显示格式定义
        ($monitor, $time, $display 等)
  
```

43



功能仿真测试时用, **不可综合!** 单位由`timescale定义

## 特殊符号 “#” ——延迟

- 特殊符号 “#” 常用来表示延迟
- ✓ 在过程赋值语句时表示延迟, 例:
 

```
initial begin #10 rst=1; #50 rst=0; end
```
- ✓ 在门级实例引用时表示延迟
 

```
例: not #1 not1(nsel, sel);
      and #2 and2(a1, a, nsel);
```
- ✓ 在模块实例化时, 延迟参数可以作为参数传递

44



## 特殊符号 “->”

### ■ 命名事件控制（named event control）

- 用关键字event定义命名事件
  - event在同步时序逻辑的测试中很有用，但不可综合
- 在测试中，通过 “->”触发该事件
- 触发的事件由 “@”后事件列表识别

### ■ 例如：

// 在最后一个数据包到达后，数据存储器进行存储

```
event received_data;
```

```
always @( posedge clock )
```

```
    if( last_data_packet ) -> received_data ;
```

```
always @( received_data ) // 等待触发
```

```
    data_buf = { data_pkt[0], data_pkt[1], data_pkt[2],  
                data_pkt[3] } ;
```

45



## 编译指令

- 编译引导语句用主键盘左上角小写键 “ ` ” 起头，用于指导仿真编译器在编译时采取一些特殊处理
- 编译引导语句一直保持有效，直到被取消或重写
- `resetall` 编译引导语句把所有设置的编译引导恢复到缺省状态

### ■ 常用的编译引导有：

```
`define  
`include  
`timescale  
`uselib  
`resetall
```

```
.....
```

46



- 使用`define 编译引导，能提供简单的文本替代功能

``define <宏名> <宏文本>`

在编译时会用宏文本来替代源代码中的宏名，合理地使用`define可以提高程序的可读性

- 举例说明：

``define on 1'b1`

``define off 1'b0`

``define and_delay #3`

``define`与参数设定有共同点，  
只不过`define是全局的

在程序中可以用有含义的文字来表示没有意思的数码，提高了程序的可读性，在程序中可以用`on，`off，`and\_delay分别表示 1，0和 #3 。

47



**btw:** 在ModelSim的开发环境中，在同一个工作文件夹，可以不用相互`include;  
UNIX系统下用make文件脚本编译。

- 使用`include 编译引导，在编译时能把其指定的整个文件包包括进来一起处理

- 举例说明：

``include "global.v"`

``include "parts/counter.v"`

``include "../..../library/mux.v"`

- 合理地使用`include 可以使程序简洁、清晰、条理清楚、易于查错

48





- ``timescale` 用于说明程序中的时间单位和仿真精度  
举例说明: ``timescale 1ns/100ps // "#delayValue"`

- ``timescale` 语句必须放在模块边界**前面**

举例说明: ``timescale 1ns/100ps`  
`module MUX2_1(out,a,b,sel);`  
`// ...`  
`not #1 not1(nsel, sel);`  
`and #2 and1(a1, a, nsel);`  
`// ...`  
`endmodule`

- 尽可能使精度与时间单位接近, 只要满足设计实际需要就行

举例说明: 在上例中所有的时间单位都是1ns的整数倍

49



#### ■ 时间单位 :

fs	(飞秒)	femtoseconds:	$1.0e^{-15}$	秒
ps	(皮秒)	picoseconds:	$1.0e^{-12}$	秒
ns	(纳秒)	nanoseconds:	$1.0e^{-9}$	秒
us	(微秒)	microseconds:	$1.0e^{-6}$	秒
ms	(毫秒)	milliseconds:	$1.0e^{-3}$	秒
s	(秒)	seconds:	1.0	秒

50



- 仿真步长即仿真单位（STU）是所有参加仿真模块中由`timescale`指定的精度中最高（即时间最短）的那个决定的：（STU=100fs）

■ 举例：

```

`timescale 1ns/10ps
module M1(...);
not #1.23 not1(nsel, sel); //1.23 ns中共有12300个STU
endmodule

`timescale 100ns/1ns
module M2(...);
not #1.23 not1(nsel, sel); //123 ns中共有1230000个STU
endmodule

`timescale 1ps/100fs
module M3(...);
not #1.23 not1(nsel, sel); //1.23 ps中共有12个STU
                                // 此时STU=100fs，以最小为准
endmodule
  
```

51



注释：库（lib），已有的资源。

`uselib 编译引导语句：

- 用于定义仿真器到哪里去找库元件
- ✓ 如果该引导语句启动的话，它就一直有效，直到遇到另外一个`uselib的定义或`resetall语句
- ✓ 比其他配置库搜索路径的命令选项作用大
- ✓ 如果仿真器在`uselib定义的地点找不到器件库，它不会转向由编译命令行-v 和-y选项指定的器件库去找。

52



### ■ 使用 `uselib 的语法:

`uselib 器件库1的地点 器件库2的地点 .....

“器件库地点” 可用以下两种方法表示:

- 1) file = 库文件名的路径
- 2) dir = 库目录名的路径 libext = .文件扩展

例如:

```
`uselib dir = /lib/FAST_lib/
`uselib dir = /lib/TTL_lib/    libext=.v
    file = /libs/TTL_U/udp.lib
```

53



## \$<标识符>——系统任务

- ‘\$’ 符号表示 Verilog 的**系统任务和函数**,常用的系统任务和函数有下面几种:

符号	意义
\$time	找到当前的仿真时间
\$display, \$monitor	显示和监视信号值的变化
\$stop	暂停仿真
\$finish	结束仿真

- 举例: initial \$monitor(\$time,"a=%b, b=%h", a, b);  
// 每当a 或b值变化时该系统任务都显示当前的仿真时刻并分别用二进制和十六进制显示信号a和 b的值

54



## 课程信息



### ■ 教师：何锋

- 新主楼 F-712
- e-Mail: [fenghe@buaa.edu.cn](mailto:fenghe@buaa.edu.cn)
- 电话: 8233.8712

### ■ 助教

- 梁咏元（新主F-713）、臧光界（新主F-713）
- 确定课代表，课代表统一收发作业

### ■ **周二**下课之后，交上一周的作业