

课程号: B3I493310

开课学期:2020-2021学年秋季



# 数字系统设计

上课期间请正确佩戴口罩

北京航空航天大学

微电子学院

贾小涛





## 第二章 硬件描述语言——Verilog

- 硬件描述语言概述
- 程序结构
- 语言要素
- 数据类型
- 表达式和运算符
- 建模方式



# 硬件描述语言概述

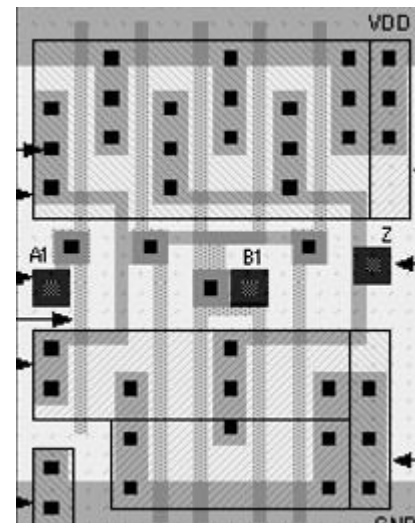
## ■ 硬件描述语言HDL(Hardware Description Language)

- 是具有特殊结构能对硬件逻辑电路的功能进行描述的一种高级编程语言
- 是硬件设计人员和电子设计自动化(EDA)工具之间的接口，其主要目的是用来编写设计文件，建立电子系统行为级的仿真模型。

```
module twomux (out, a, b, sl)
  input a, b, sl;
  output out;
  not u1 (nsl, sl);
  and #1 u2 (sela, a,
nsl);
  and #1 u3 (selb, b, sl)
  or   #2 u4 (out, sela,
selb);
endmodule
```



**EDA**

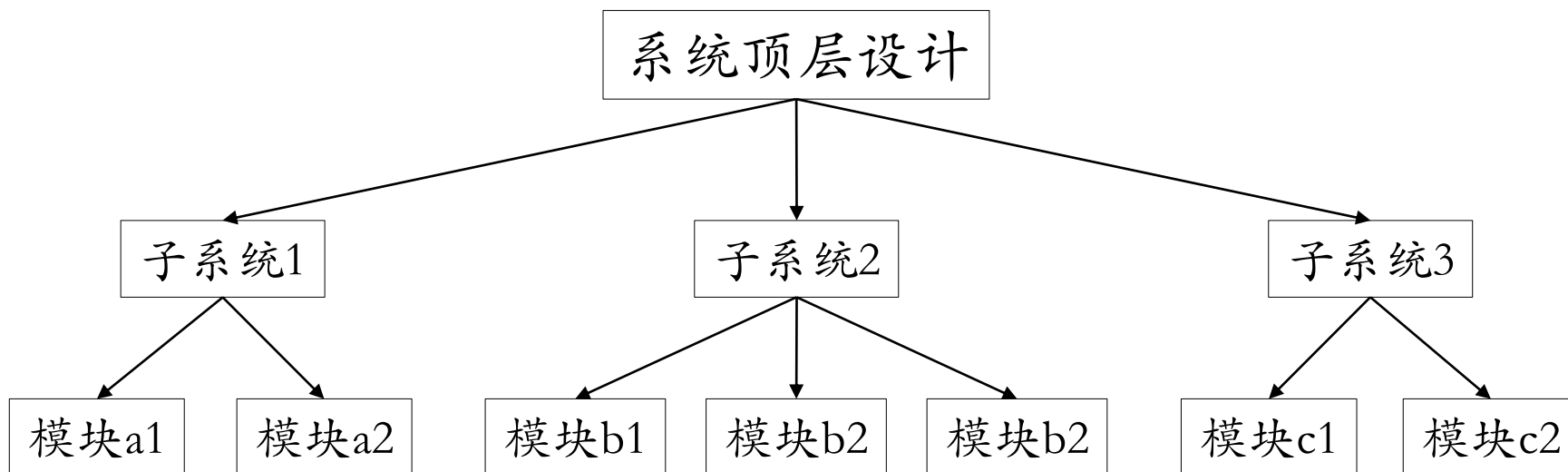




# 硬件描述语言概述

## ■ 硬件描述语言

- 易于存储、阅读，便于计算机模拟和处理
- 成功应用于设计的各个阶段：建模、仿真、验证、综合等
- 自顶向下的设计模式





# 硬件描述语言概述

## ■ 主要的硬件描述语言

**Verilog HDL**: Gateway Design Automation

**VHDL**: 美国国防部

**OO VHDL**: 美国国防部（面向对象）

**DE VHDL**: 美国杜克大学

**VITAL**: 美国电气与电子工程师协会（IEEE）



# 硬件描述语言概述

## ■ 主要的硬件描述语言

- 目前主要使用的两种HDL

**Verilog HDL: Gateway Design Automation**

**1995年成为IEEE标准**

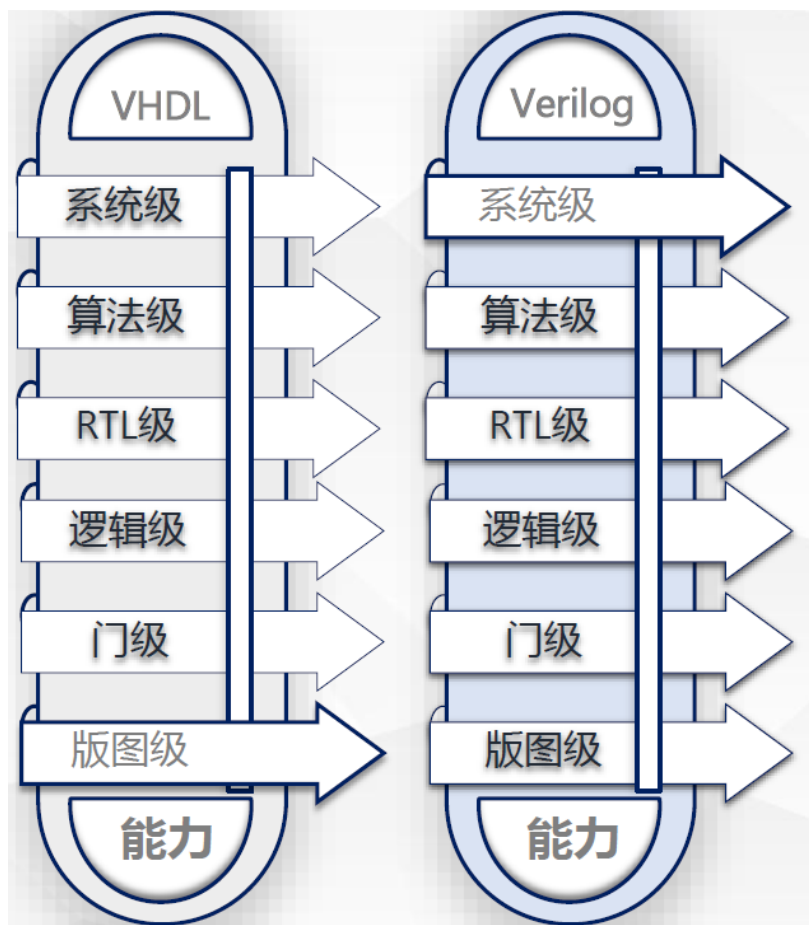
**VHDL: 美国国防部**

**1987年成为IEEE标准**



# 硬件描述语言概述

## ■ 两种硬件描述语言建模能力比较



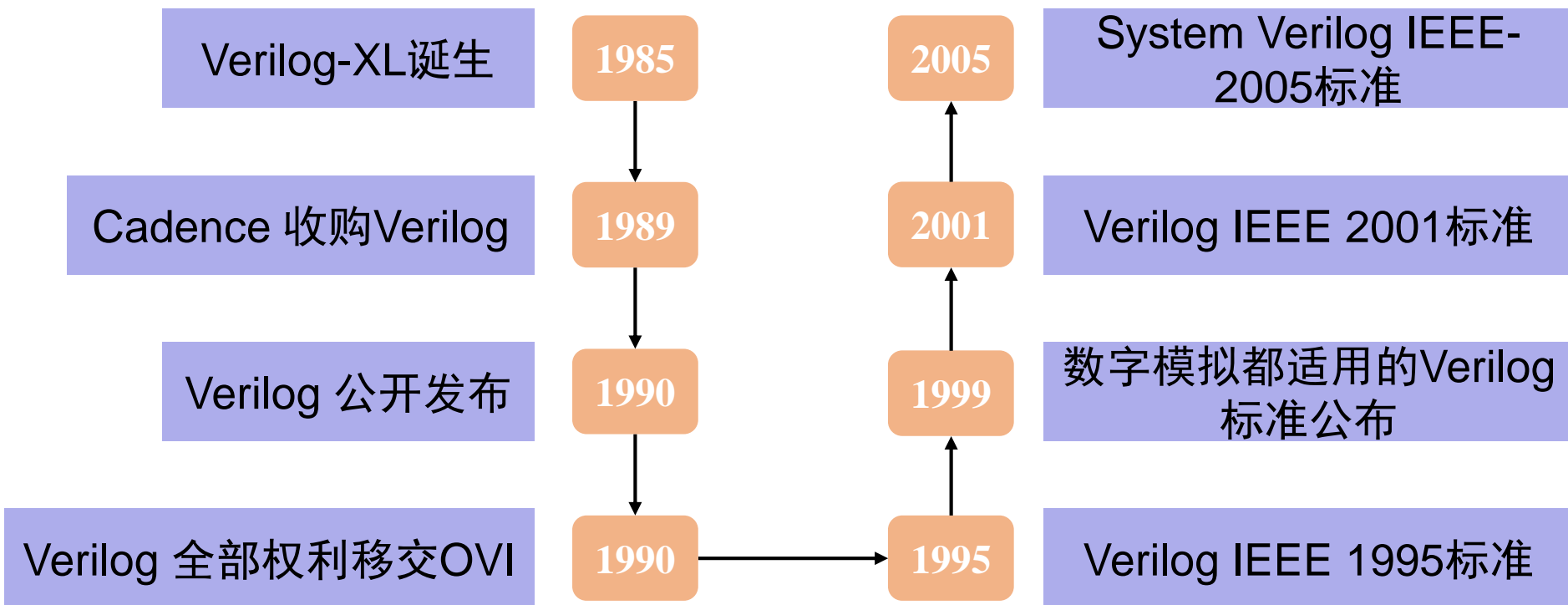
### System Verilog

结合了来自 Verilog、VHDL、C++的概念，提高系统级设计能力



# 硬件描述语言概述

## ■ Verilog 的发展史







# 硬件描述语言概述

## ■ 知识产权核（IP核）

- 是基于ASIC或FPGA预先设计好的电路功能模块
- 复杂芯片设计时常调用已有IP核
- 时序、面积、功率可配置
- IP调用可以提高系统的设计效率，缩短系统的设计周期
- 常见的IP核
  - FIFO
  - PLL
  - RAM



# 硬件描述语言概述

## ■ 知识产权核（IP核）

### 软核

使用RTL或更高级别的描述定义的功能块，不涉及电路的具体实现；验证通过并且可综合  
**灵活、可移植、可重用；电路性能无法得到保证**

### 固核

提供参数化描述的电路功能块，方便设计者根据特定的设计需求对核心进行优化。  
**灵活的参数增加了电路性能的可预测性**

### 硬核

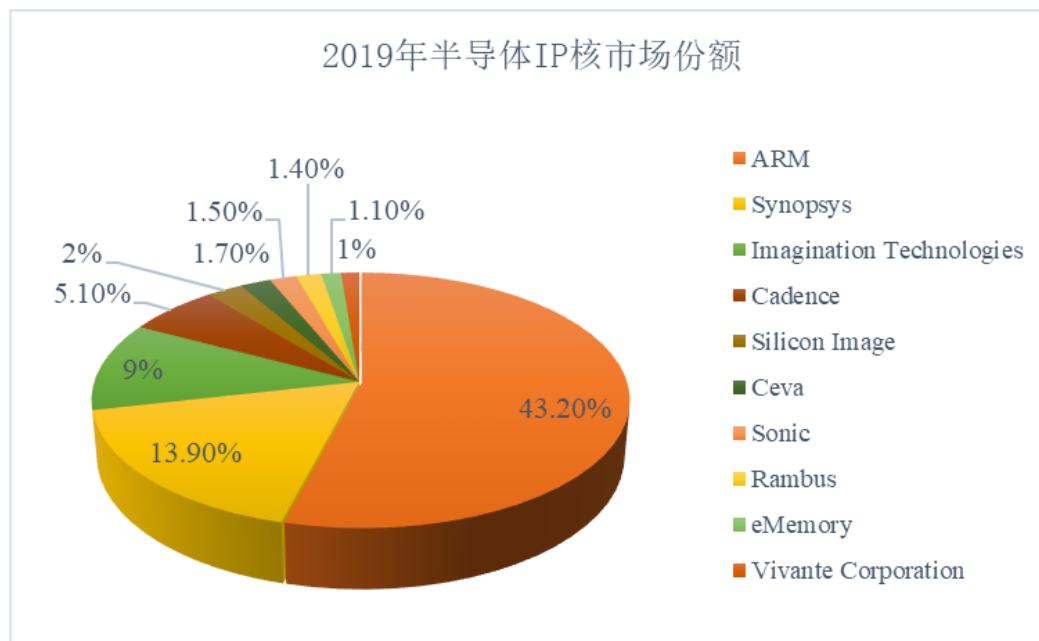
具有固定版图，并且针对特定应用程序和工艺进行了高度优化的功能块  
**具有稳定的功能，以及可预测的性能，且更易于实现IP保护；缺乏灵活性及可移植性**



# 硬件描述语言概述

## ■ 知识产权核（IP核）

### — 2019年半导体IP核市场份额分布

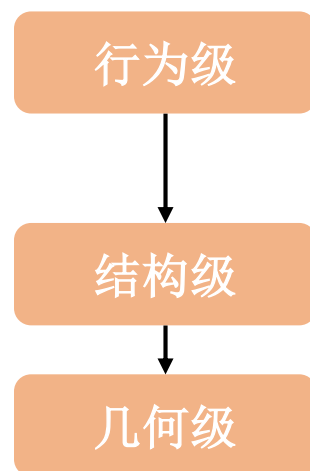




# 硬件描述语言概述

## ■ Verilog的应用

- 用电路的功能/行为描述电路，也可以用元器件及它们之间的连接关系描述电路
- 支持数字电路系统中5种不同的描述方法
  - 系统级 (System)
  - 算法级 (Algorithm)
  - 寄存器传输级 (RTL)
  - 门级 (Gate)
  - 开关级 (Switch-level)





# 硬件描述语言概述

## ■ Verilog的特点与功能

- 结构化、过程性语言
- 工艺无关性、可移植性、易于维护
- 提供算术运算符、逻辑运算符、位运算符
- 提供条件、循环程序结构
- 可描述顺序执行或并行执行的程序结构
- 提供了可带参数且非零延续时间的任务(task)程序结构
- 提供了可定义新的操作符的函数（function）程序结构



# 硬件描述语言概述

## ■ 如何设计一个32位的与门？

### — 电路原理图/版图方法

- 设计一个2位与门
- 由2位与非门形成32位与门

### — Verilog 描述

```
module and32(a, b, out);  
    input [31:0] a, b;  
    output [31:0] out;  
    wire [31:0] a, b;  
    reg [31:0] out;  
    always @ (a, b)  
    begin  
        c=a&b;  
    end  
endmodule
```



## 第二章 硬件描述语言——Verilog

- 硬件描述语言概述
- 程序结构
- 语言要素
- 数据类型
- 表达式和运算符
- 建模方式

# Verilog 程序结构



## ■ 模块(Module)是Verilog的基本设计单元

- 复杂电路的构建，主要是通过模块的相互连接调用来实现
- 模块包含在关键字module endmodule之内
- 模块提供输入、输出端口，可以调用其他模块
- 模块可以被其他模块例化
- 包括组合逻辑电路和时序逻辑电路





# Verilog 程序结构

## ■ 模块(Module)是Verilog的基本设计单元

**module** 模块名 (端口1, 端口2, 端口3,);

模块端口  
说明

端口类型声明; // **input/output/inout**  
参数定义; //可选  
数据类型定义; // **wire/reg**等

实例化底层模块或基本门级元件;

逻辑功能  
描述

连续复制语句; // **assign**

过程结构块; // **initial/always**  
行为描述语句

**endmodule**

module 名字 (端口列表);

端口定义;

**input** 输入端口

**output** 输出端口

**inout** 双向端口

数据类型说明

**wire**

**reg**

**parameter**

逻辑功能描述

**always**

**assign**

**task**

**function**

元件例化

调用模块

.....

**endmodule**



# Verilog 程序结构

## ■ 端口定义

- 端口是模块与外部其他模块进行信号传递的通道
- 端口列表中的所有端口必须在模块中进行声明

— input

— output

— inout

```
module and2( a, b, out);  
    input a, b;  
    output out;  
    wire a, b;  
    reg out;  
    always @ (a, b)  
    begin  
        c=a&b;  
    end  
endmodule
```



# Verilog 程序结构

## ■ 端口定义

— 内部端口与外部端口

— 从内部看:

➤ 输入端口为wire型

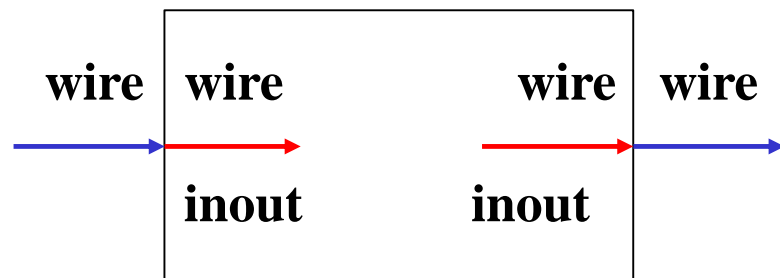
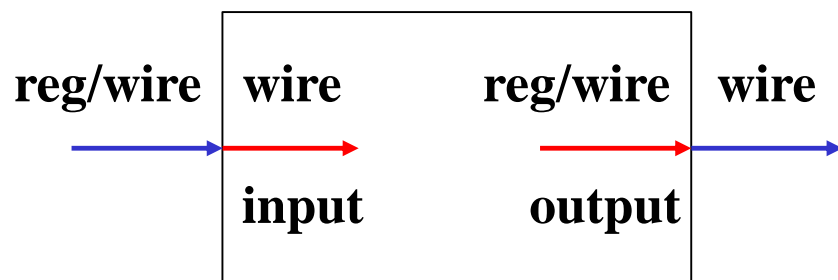
➤ 输出可以为wire也可以为reg

— 从外部看

➤ 输入端口为reg/wire

➤ 输出端口为wire

— 没有明确指定时，端口默认数据类型为wire



# Verilog 程序结构



## ■ 端口定义

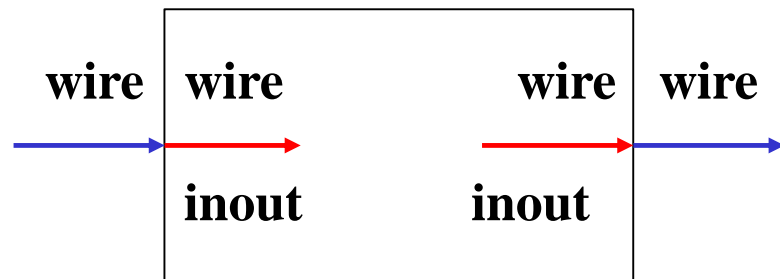
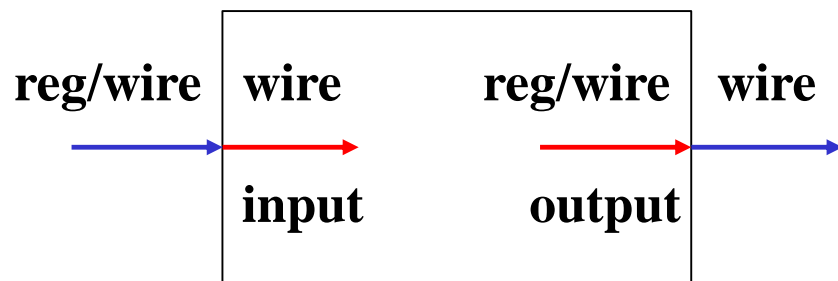
- 允许位宽不匹配，但不建议
- 允许端口悬空
- 端口与外部信号的连接

### ➤ 顺序连接

and2 and\_ins1(a, b, c)

### ➤ 命名端口连接

and2 and\_ins2(.a(a), .b(b), .out(c))





# Verilog 程序结构

## ■ 示例

```
module name(a, b, c, d);  
    input [7:0] a;  
    input [7:0] b;  
    output [3:0] c;  
    output [3:0] d;  
    reg [3:0] c;  
endmodule
```

```
module name(a, b, c, d);  
    input [7:0] a, b;  
    output [3:0] c;  
    output reg [3:0] d;  
endmodule
```



## 第二章 硬件描述语言——Verilog

- 硬件描述语言概述
- 程序结构
- 语言要素
- 数据类型
- 表达式和运算符
- 建模方式



# Verilog 语言要素

- 注释
- 间隔符
- 标识符
- 关键字
- 数字

# Verilog 语言要素



## ■ 注释

- 单行注释 //
- 多行注释 /\* \*/
- 多行注释不能嵌套
- 非法多行注释

```
module name(a, b, c, d);  
    /*input [7:0] a;  
    /*input [7:0] b;  
    output [3:0] c;*/  
    output [3:0] d;  
    reg [3:0] c;*/  
endmodule
```

注释不要使用中文  
移植过程中易出现乱码

中文注释，实验扣分

## 合法多行注释

```
module name(a, b, c, d);  
    /*input [7:0] a;  
    input [7:0] b;  
    //output [3:0] c;  
    output [3:0] d;  
    reg [3:0] c;*/  
endmodule
```





# Verilog 语言要素

## ■ 间隔符，又称空白符

- 空格符 \b
- 制表符 \t
- 换行符 \n
- 换页符
- 编译和综合是，间隔符不具有任何意义
- 增加程序的可读性
- 字符串中的间隔符具有意义



# Verilog 语言要素

## ■ 标识符

- 用于命名
  - 模块名、端口名、变量名
- 由字母、数字、\$、\_组成
- 开头必须是字母或者\_
- 区分大小写
- 普通标识符
- 转义标识符

命名要通俗易懂  
不要使用保留字

### 合法标识符:

and2  
merge\_ab  
fullAdder  
MUX  
a\$c  
\_test

### 不合法标识符:

0test  
test&test  
%test  
a+b-c



# Verilog 语言要素

## ■ 关键字

— 又称保留字，约上百种

— 用户不要使用关键字

— 小写

— Always?

always	and	assign	begin	buf
bufif0	bufif1	case	casex	casez
cmos	deassign	default	defparam	disable
edge	else	end	endcase	endfunction
endmodule	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	notif1
pull0	pull1	pulldown	pullup	rcmos
reg	release	repeat	rmos	rpmos
rtran	rtranif0	rtranif1	scalared	small
specify	specparam	strong0	strong1	supply0
supply1	table	task	time	tran
tranif0	tranif1	tri	tri0	tri1
triand	trior	vectored	wait	wand
weak0	weak1	while	wire	wor
xnor	xor			

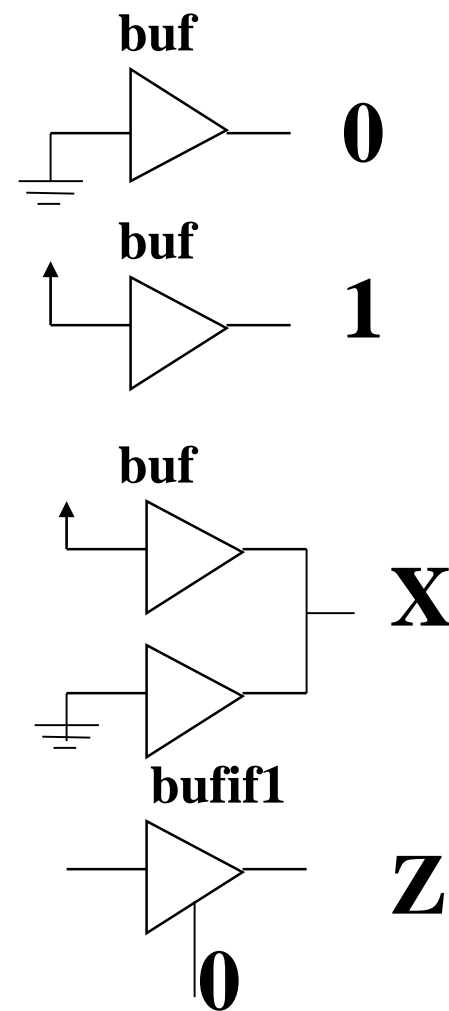


# Verilog 语言要素

## ■ 数值

— Verilog有四种基本的逻辑数值状态

状态	含义
0	低电平、逻辑0、Low、False、Ground、VSS
1	高电平、逻辑1、True、Power、VDD、VCC
x/X	不确定状态、Unknown
z/Z	高阻值、HiZ、Disabled Driver





# Verilog 语言要素

## ■ 数值

- 数值表示
- `<+/-><size>'<base_format><number>`

符号

位宽

'进制

数字

补码的方式存储

表示number  
转换为二进制  
后的位宽

表示数字所  
采用的进制

除了正常各种进制的数字之外，  
还可使用x和z



## ■ 数值

### — 数值表示

进制	符号	数值范围
二进制	b/B	0/1
八进制	o/O	0-7
十进制	d/D	0-9
十六进制	h/H	0-9,a-f

通常不使用八进制和十进制



# Verilog 语言要素

## ■ 数值

符号

位宽

'进制

数字

- 符号默认是+；位宽默认为32位；进制默认为十进制
- 进制省略原则
  - 只有10进制能省略
  - 不能单独省略，只能与位宽同时省略
- 位宽省略会造成端口不匹配，导致硬件资源增加，**建议使用完整的表达方式**
- 数字中间可通过下划线分割，增加可读性



# Verilog 语言要素

## ■ 数值

### — 位宽对齐

#### ➤ 位宽大于实际位数

✓ 最高位为0/1：左端补0扩展

» 1010    0000\_1010

» 0110    0000\_0110

✓ 最高位为x/z：左端补x/z

» xx01    xxxx\_xx01

» z101    zzzz\_z101

✓ 未指定位宽：默认补齐32位

位宽小于实际位数？  
高位截断



# Verilog 语言要素



## ■ 数值

Number representation	Binary equivalent	Explanation
4'd5	0101	Decimal 5 is interpreted as a 4-bit number
8'b101	00000101	Binary 101 is turned into an 8-bit number
8'hb_3	10110011	Binary equivalent of hex; underscore is ignored
10'o752	0111101010	Octal 752 with a 0 padded to its left to make it a 10-bit number
8'hf	00001111	Hexadecimal f is expanded to 8 bits by padding zeros to its left
'hxa	32'hxa	The number of bits is omitted
99	32'd99	Only decimal number can omit database



# Verilog 语言要素

## ■ 数值

### — 未指明位宽的常数

- 659 // decimal number
- 'h837FF //hexadecimal number
- 'o7460 //octal number
- 4af //illegal X

### — 指明位宽的常数

- 4'b1001 //4-bit binary number
- 5'd3 // 5-bit decimal number equivalent to 5'b00011



# Verilog 语言要素

## ■ 数值

### — 有符号常量

➤ `8'd-6` // illegal syntax

➤ `-8'd6` // 8-bit -6 1111\_1010

### — 自动左对齐

➤ `reg [11:0] a, b, c, d;`

➤ `a = 3'h0d3` // 0d3

➤ `b = 3'h3x` //03x

➤ `c = 3'hza` //zza

为什么a可以用  
3位16进制表示？



## 第二章 硬件描述语言——Verilog

- 硬件描述语言概述
- 程序结构
- 语言要素
- 数据类型
- 表达式和运算符
- 建模方式

# Verilog 数据类型



## ■ Verilog共有19中数据类型

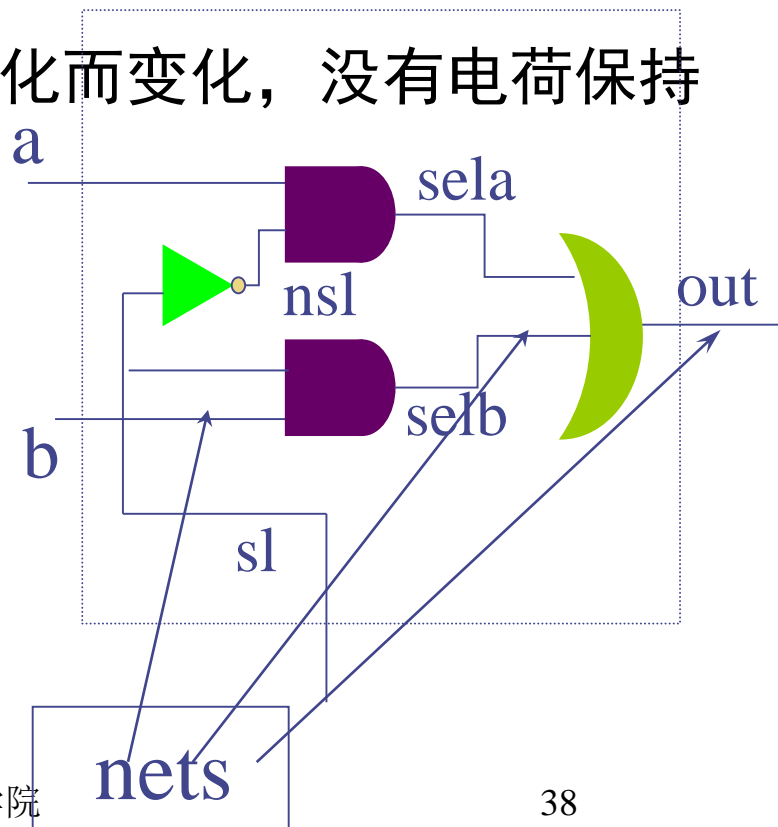
- 线网型
- 寄存器型
- 参数型
- 其他



# Verilog 数据类型

## ■ 线网型

- 相当于硬件电路中的各种物理连接（导线和节点），用来连接各个模块以及输入输出
- 特点：输出的值紧跟输入值的变化而变化，没有电荷保持作用（triereg除外）
- 必须有驱动源驱动
  - 门元件
  - 通过assign 赋值语句对其进行赋值





# Verilog 数据类型

## ■ 常用的net型变量及可综合性说明

类 型	功 能	可综合性
wire, tri	两种常见的net型变量	√
wor, trior	具有线或特性的多重驱动连线	
wand, triand	具有线与特性的多重驱动连线	
tri1, tri0	分别为上拉电阻和下拉电阻	
supply1, supply0	分别为电源（逻辑1）和地（逻辑0）	√
triereg	具有电荷保持作用的连线，可用于电容的建模	

Verilog HDL提供了多种net型数据，如表所示，表中符号“√”表示可综合。



## ■ 线网类型数据的声明语法

`<net_type> [range] [delay] <list_of_net_name>;`

其中：

**net\_type:** 表示网络型数据数据的类型。

**range:** 用来指定数据为标量或矢量。若该项默认，表示数据类型为1位的标量；反之，由该项指定数据的矢量形式。

**delay:** 指定仿真延迟时间。

**list\_of\_net\_name:** net名称，一次可定义多个net，用逗号分开。





# Verilog 数据类型

## ■ 线网类型数据的声明示例

```
wand w;           // 一个标量wand类型net
tri [15: 0] bus;   // 16位三态总线
wire [31: 0] w1, w2; // 两个32位wire, MSB为bit31
wire [0: 31] w1, w2; // 两个32位wire, MSB为bit0
```

MSB: Most Significant Bit

LSB: Least Significant Bit

wire是最常用的net类型，Verilog HDL模块中的输入、输出信号在没有明确指定数据类型时都被默认为wire型。

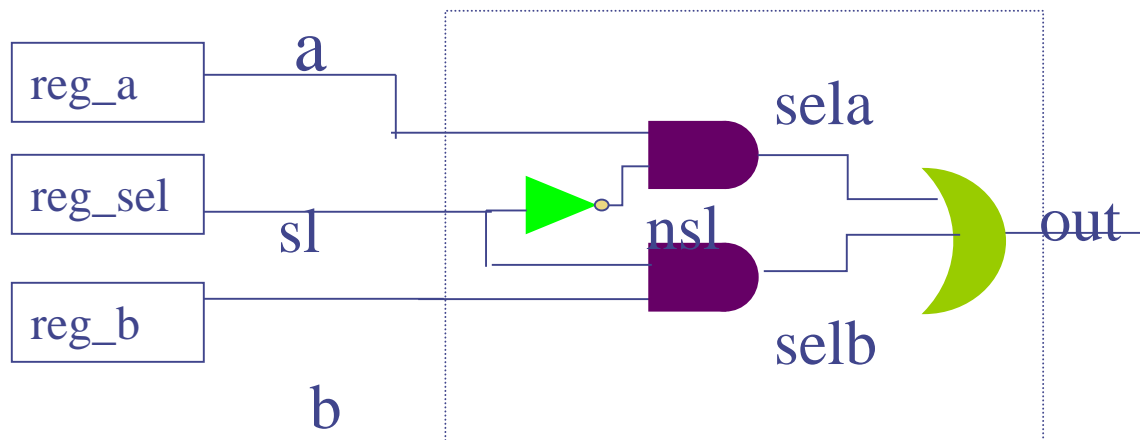


# Verilog 数据类型

## ■ 寄存器类型

类 型	功 能	可综合性
reg	可变位宽变量	√
integer	32位带符号整型变量	√
real	64位带符号实型变量	
time	64位无符号时间变量	

reg型变量是最常用的variable型变量





# Verilog 数据类型

## ■ reg

`<reg_type> [range] <list_of_reg_name>;`

其中：

reg\_type为寄存器类型；

range为矢量范围，[MSB: LSB]格式，只对reg类型有效；

list\_of\_reg\_name为reg型变量的名字，一次可定义多个reg型变量，使用逗号分开。

# Verilog 数据类型



## ■ reg 数据类型示例

```
reg out; //定义一个1位的reg型数据out  
reg[7:0] address; //定义一个8位的reg型数据address  
reg[16:1] data; //reg型数据data的宽度是16位
```

**reg**类型数据的默认值是未知的。

**reg**的赋值过程为过程赋值（阻塞赋值和非阻塞赋值），**reg**可以保持每次的赋值，所以可以综合为硬件寄存器，同时也可以综合为逻辑线网



# Verilog 数据类型

## ■ Reg

reg型数据的值通常被认为是无符号数，如果给reg型数据中存入一个负数，通常会被视为整数。

```
module reg_syn1(a,b,c,e,f);  
    input a,b,c;  
  
    reg[4:1] out; //定义一个1个4的位寄存器out  
  
    ...  
    out=5;        //这条赋值语句给out赋值5（0101）  
    out=-2;       /*这条赋值语句给out赋值-2，out的值为14（1110），1110是2的补码形式*/  
    */
```



# Verilog 数据类型

## ■ Integer

integer是整数寄存器，也是Verilog HDL中最常用的变量类型。

integer型变量的定义格式：

```
integer integer1, integer2, integer3, .....,  
integerN[msb:lsb];
```

例如：

```
integer A, B;           //定义了3个整数型寄存器  
integer data[1:3];      //定义了一组寄存器，其名称分别为data[3], data[2], data[1]
```



# Verilog 数据类型

## ■ real

real是实数型寄存器，采用双精度浮点数一般用于测试模板中存储仿真时间。

real型变量的定义格式：

real real\_reg1, real\_reg 2, real\_reg 3, ..., real\_reg N;

例如：

real Swing, Top;

例如：

real Rc;

d

两种表达方式：十进制法和科学计数法

小数点前后必须有数字！

1.2 1.5e2 （合法）

.88 .9e-3 （非法）



# Verilog 数据类型

## ■ time

time是时间寄存器，用于存储和处理时间，通常用在系统函数\$time中。

time型变量的定义格式：

```
time time_reg1, time_reg2, time_reg3, ..., time_regN[msb:lsb];
```

例如：

```
time events[0:31];           //时间值数组，可以存储32个时间值
```

```
time currtime;               //currtime可以存储一个时间值
```





## ■ 参数 (Parameter)

1. 参数型数据是被命名的常量
2. 参数型常量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在被引用模块或实例中以定义参数。
3. 数据的具体类型（整数、实数、字符串）是由所赋的值来决定的。
4. 参数是本地的，其定义只在本模块内有效。



# Verilog 数据类型

## ■ 参数 (Parameter)

### — 示例

```
parameter msb=15;
```

```
//定义最高位为第15位
```

```
parameter a=2, b=3;
```

```
//定义两个常量a, b
```

```
parameter delay=10;
```

```
//定义延时为10个时间单位
```

```
parameter msb=4, size=msb-1;
```

```
//定义表达式
```



# Verilog 数据类型

## ■ 参数重定义

```
module M1(....);  
    parameter para1 = 5 ;  
    parameter para2 = “../temp” ;  
    input...;  
    output...;  
    .....  
endmodule
```

方法一：

使用关键字： **defparam**

```
module top ( .....)  
    input...;  
    output...;  
    defparam U1.para1 = 10 ;  
    defparam U1.para2 = “../temp”  
    M1 U1 (.....);  
endmodule
```



# Verilog 数据类型

## ■ 参数重定义

方法二:

参数传递(列表式)

```
module top ( .....)  
    input....;  
    output....;  
    M1 # ( 10, “../temp”)  
        U1 (.....);  
endmodule
```

方法二:

参数传递 (端口列表式)

```
module top ( .....)  
    input....;  
    output....;  
    M1 # ( .param1(10), .param2(“../temp”)  
        U1 (.....);  
endmodule
```

defparam可综合性问题:一般情况下是不可综合的.  
提示:通常不要使用defparam语句!



# Verilog 数据类型

## ■ 标量与向量

- 位宽为1位的变量称为标量（Scalar），如果在变量声明中没有指定位宽，则默认为标量（1位）
- 位宽大于1位的变量（包括net型和variable型）称为向量（vector）。向量通过位宽定义语法[msb:lsb]指定地址范围

wire reset;      //reset为wire型标量

reg S;            //S为reg型标量

wire[2:0] sel ; //3位的向量sel，其中sel[2]是最高有效位，sel[0]是最低有效位

reg[0:3] A;        //A[0]是最高有效位，A[3]是最低有效位

reg[3:0] data;     //4位的总线data




# Verilog 数据类型

## ■ 位选择与域选择

- 在向量中，可以指定其中的某一位或若干相邻位进行操作，这些指定的一位或相邻位分别称为位选择或域选择（部分选择）

`S=data[3]`                      //位选择：将data的第3位赋值给变量S  
`D=data[3:0];`                //域选择：将data的第3、2、1、0位的  
   值赋给变量D

<code>wire[15:0] out; wire[3:0] in;</code>		<code>assign out[8]=in[3];</code>
<code>assign out[8:5]=in;</code>		<code>assign out[7]=in[2];</code>
		<code>assign out[6]=in[1];</code>
		<code>assign out[5]=in[0];</code>



# Verilog 数据类型

## ■ 数组 (array)

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

```
reg data[7:0];           //8个1位寄存器组成的存储器data
reg[8:1] data[7:0];      //8个8位寄存器组成的存储器data
reg[0:3] A;              //A[0]是最高有效位, A[3]是最低有效位
reg[3:0] data;           //4位的总线data
```

也可以用parameter参数定义存储器的尺寸, 可便于修改,

```
parameter wordsize=16, memorysize=256;
reg[wordsize-1:0] Amem[memorysize-1:0], writereg, readreg;
```



# Verilog 数据类型

## ■ 数组（array）

### — vector 与 array的不同

reg[7:0] rega;	//1个8位的寄存器
reg Amem[7:0];	//8个1位寄存器组成的存储器Amem

### — 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。

### — 对存储器赋值时，只能对存储器的某一个单元进行赋值

rega=0;	//合法的赋值语句
Amem=0 ;	//非法的赋值语句





# Verilog 数据类型

## ■ 数组 (array)

```
reg[7:0] Bmem[31:0];           //存储器定义  
  
Bmem[4]=8'b00110011;          //Bmem存储器的第4个单元被赋值  
                                为8'b00110011  
  
Bmem[7]=21;                    //Bmem存储器的第7个单元被赋值  
                                为十进制数21
```



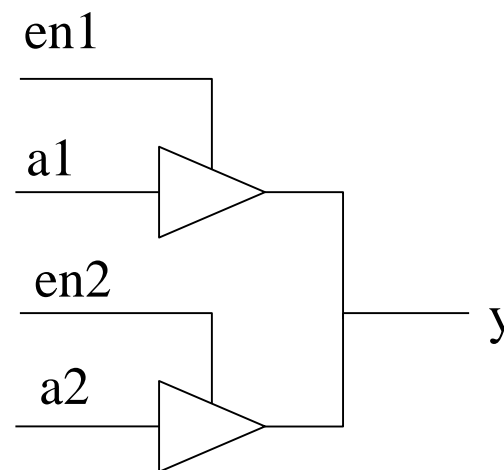
# Verilog 数据类型

## ■ wire和reg是Verilog最常用的两种数据类型

- wire必须有驱动，多个驱动源可以同时驱动一个线网

```
module deivers1(y, a1, en1, a2, en2);  
    output y;  
    input a1, en1, a2, en2;  
    assign y = en1? a1; 1'bz;  
    assign y = en2? a2; 1'bz;  
endmodule
```

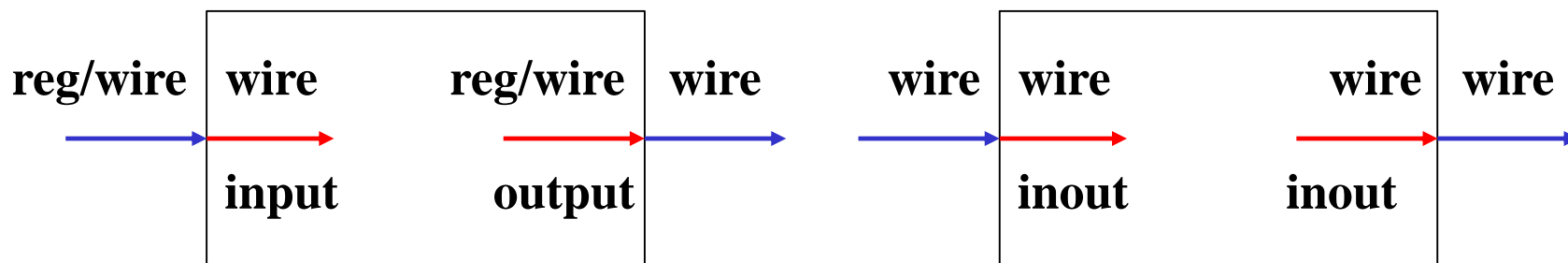
- 等效电路图是什么呢？





# Verilog 数据类型

- wire和reg是Verilog最常用的两种数据类型
  - 何时使用wire? 何时使用reg?
  - 内部信号: 如果信号变量是在过程块 (**initial块 或 always块**)中被赋值的, 必须把它声明为寄存器类型变量





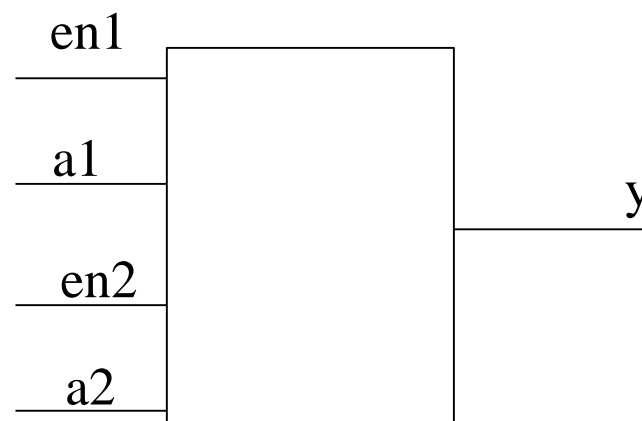
# Verilog 数据类型

## ■ wire和reg是Verilog最常用的两种数据类型

- reg型可以保存变量的值，直至输入信号发生变化

```
always @(a1 or en1)
    if(en1) y = a1;
    else y = 1'bz;
always @(a2 or en2)
    if (en2) y = a2;
    else y = 1'bz
```

- 等效电路图是什么呢？





## 第二章 硬件描述语言——Verilog

- 硬件描述语言概述
- 程序结构
- 语言要素
- 数据类型
- **表达式**
- 建模方式



# 表达式

## ■ 表达式=操作符+操作数

## ■ 操作符

### — 按功能分为九类

➤ 算术运算符、逻辑运算符、位运算符、关系运算符、等式运算符、缩位运算符、移位运算符、条件运算符和位拼接运算符

### — 按操作数数目分为3类

➤ 单目运算符、双目运算符、三目运算符



# 表达式

## ■ 算术运算符

+	加	-	减
*	乘	/	除
%	求余（求模）	**	乘方（求幂）

11/2      //结果为3，11除以3，商为3.333...，截去小数部分，取整数部分3

11/3      //结果为3，11除以3，商为3.333...，截去小数部分，取整数部分3

8%4      //结果为0，8除以4，余数为0



# 表达式

## ■ 算术表达式

例如：

reg[0:3] A, B, C;

reg[0:5] F;

...

A=B+C;      //长度由B, C和A中最长的长度决定, 长度为4位  
F=B+C;      /\*长度由F的长度决定 (F, B, C中最长的长度为  
                    F的长度), 长度为6位\*/





# 表达式

## ■ 算术表达式

```
wire[4:1] A, B;
```

```
wire[1:5] C;
```

```
wire[1:6] D;
```

```
wire[1:8] Y;
```

```
assign Y = (A+C)+(B+D);
```

/\*赋值语句，(A+C)和(B+D)两个算术操作的结果长度取每个操作的最大操作数的长度（5位和6位），最终Y的结果取决于Y的长度（8位）\*/



# 表达式

## ■ 算术表达式

```
module arithmetic_test;
reg[3:0] a,b,c;
initial
begin
a=4'b1100;           //12
b=4'b0011;           //3
c=4'b0010;           //2
$display(a*b) :      //结果为4（4'b0100），等于36的低4位
$display(a/b);       //结果为4
$display(a+b) :      //结果为15
$display(a-b);       //结果为9
$display(c**b) :     //结果为8
```



# 表达式

## ■ 位运算

$\sim$       按位取反       $\&$       按位与  
 $|$       按位或       $\wedge$       按位异或  
 $\sim^A$ 或 $\sim^B$       按位同或（符号 $\sim^A$ 和 $\sim^B$ 是等价的）

按位取反  
的真值表

$\sim$	0	1	x	z
结果	1	0	x	x

按位与、按位  
或、按位异或  
的真值表

&	0	1	x	z		0	1	x	z	^	0	1	x	z
0	0	0	0	0	0	0	1	x	x	0	0	1	x	x
1	0	1	x	x	1	1	1	1	1	1	1	0	x	x
x	0	x	x	x	x	x	1	x	x	x	x	x	x	x
z	0	x	x	x	z	x	1	x	x	z	x	x	x	x

## ■ 位运算

```
module bit_test;  
  reg[3:0] a,b,c;  
  reg[5:0] d;  
  reg[4:0] e;  
  initial  
  begin  
    a=4'b1100;  
    b=4'b0011;  
    c=4'b0101;  
    d=6'b001010;  
    e=5'bx1100;  
    $display(~a);
```

```
    $display(a|b);           //结果为4'b1111  
    $display(b^c);           //结果为4'b0110  
    $display(a~^c);          //结果为4'b0110  
    $display(a&d);           //结果为6'b001000  
    $display(d|e);           //结果为x  
  
  end  
endmodule  
  
//结果为4'b0011
```



# 表达式

## ■ 逻辑运算

&&    逻辑与            ||    逻辑或  
!      逻辑非

module logic_test;	\$display(a  c);	//结果为1
reg[3:0] a,b,c;	\$display(a&&c) ;	//结果为x
initial	\$display(b  c);	//结果为x
begin	\$display(!c);	//结果为x
a=6;	end	
b=0;	endmodule	
c=4'hx;		
\$display(a&b);	//结果为0	
\$display(a  b);	//结果为1	
\$display(!a);	//结果为0	



# 表达式

## ■ 逻辑运算与位运算比较

- 逻辑运算的输出长度为1（0/1表示 True or False）
- 位运算的输出长度由输入决定

a	b	a&b	a b	a&&b	a  b
0	1	0	1	0	1
000	000	000	000	0	0
000	001	000	001	0	1
011	001	001	011	1	1



# 表达式

## ■ 缩位运算

&	与	~&	与非
	或	~	或非
^	异或	^~或~^	同或

- 与位运算的操作符基本一致（没有取反运算）
- 单目运算（位运算为双目运算）

reg[3:0] a;

y=&a;

//等效于b=((a[0]&a[1])&a[2])&a[3];



# 表达式

## ■ 缩位运算

```
module reduction_test;                                $displayb(~^b);           //结果为1
  reg[3:0] a,b,c;                                       $displayb(&c);             //结果为0
  initial                                              $displayb(|c);            //结果为1
  begin                                                $displayb(^c);            //结果为x
    a=4'b1101;                                         end
    b=4'b1111;                                         endmodule
    c=4'b1x01;
    $displayb(&a);                                     //结果为0
    $displayb(|a);                                     //结果为1
    $displayb(~|a);                                    //结果为0
    $displayb(^b);                                     //结果为0
```





# 表达式

## ■ 关系运算符

<	小于	>	大于
<=	小于等于	>=	大于等于

```
module relate_test;
reg[3:0] a,b,c,d;
initial
begin
    a=3;
    b=6;
    c=3;
    d=4'hx;

    $display(a<b);           //结果为1
    $display(a>b);           //结果为0
    $display(a>=c);          //结果为1
    $display(d<=c);          //结果为x

end
endmodule
```



# 表达式

## ■ 等式运算符

$==$       等于       $!=$       不等于  
 $===$     全等       $!==$     不全等

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

全等、不全等无法进行综合  
等于、不等于综合成比较器



# 表达式

## ■ 等式运算符

```
module equality_test;
    reg[3:0] a,b,c,d,e,f,g,i;
    initial
        begin
            a=4;
            b=7;
            c=4'b010;
            d=4'bx10;
            e=4'bx101;
            f=4'bxx01;
            g=4'bx01;
            i=4'b111;

            $displayb(c);           //结果为0010
            $displayb(d);           //结果为xx10
            $display(a==b);         //结果为0
            $display(b==i);         //结果为1
            $display(c!=d);         //结果为x
            $display(c!=f);         //结果为1
            $display(d===e);        //结果为0
            $display(c!==(d));      //结果为1
            $display(f===g);        //结果为1
        end
endmodule
```



# 表达式

## ■ 移位运算符

<<      逻辑左移      >>      逻辑右移  
<<      算术左移      >>      算术右移

- 逻辑左移、逻辑右移、算术左移，移入0
- 算术右移，移入标志位

a	a>>2	a>>>2	a<<2	a<<<2
0100_1111	0001_0011	0001_0011	0011_1100	0011_1100
1100_1111	0011_0011	1111_0011	0011_1100	0011_1100



# 表达式

## ■ 拼接运算符

位拼接运算符（“{}”）

使用格式如下：

{操作数1，操作数2，操作数3，.....，操作数n}

{a, b, c}

{a[1], b[2], c[2:3]} //等同于{a[1], b[2], c[2], c[3]}



## ■ 拼接运算符

```
module concatenate_test;
  reg[1:0] a;
  reg[2:0] b;
  reg[3:0] c;
  initial
    begin
      a=2'b01;
      b=3'b101;
      c=4'b1110;
      $displayb({a,b});           //结果为01101
      $displayb({a,c[2:1]});      //结果为0111
    end
endmodule
```



# 表达式

## ■ 拼接运算符

位拼接运算符还可以嵌套使用，如果多次拼接同一个操作数，重复的次数可以用常数指定，这时位拼接运算符又称为复制运算符。其使用格式如下：

{重复次数{操作数}}

{2{a, b}}	//等同于{{a, b}, {a, b}}和{a, b, a, b}
{5{2'b01}}	//等同于0101010101
{4{k}}	//等同于{k, k, k, k}



# 表达式

## ■ 拼接运算符

```
module replicate_test;
  reg[1:0] a;
  reg b;
  reg[5:0] c;
  initial
    begin
      a=2'b01;
      b=1'b1;
      $displayb({2{a}});           //结果为0101
      $displayb({3{b}});          //结果为111
      c={2{a}};
      $displayb(c);                //结果为000101
    end
endmodule
```





# 表达式

## ■ 条件运算符

条件运算符（“?:”）

其使用格式如下：

结果=条件表达式?表达式1:表达式2;

y=ctrl?a:b;                      //如果ctrl=1，则y=a； 如果  
                                    如果ctrl=0，则y=b;

或y=(ctrl==0)?b:a;    //功能与上句相同

条件运算符还可以嵌套使用。

y=ctrl1?(ctrl0?a:b):(ctrl0?c:d);



# 表达式

## ■ 运算优先级

运算符	优先级
+, -, !, ~ (单目运算符)	最高优先级
*, /, %	
+, - (双目运算符)	
<<, >>	
<, <=, >, >=	
==, !=, ===, !==	
&, ~&	
^, ~^	
, ~	
&&	
	最低优先级
?:	



# 表达式

## ■ 操作数

- 直接以完成的方式使用变量：引用名字
- 只使用变量的某位数据：位选择（bit-select）
- 只使用变量的某些数据：域选择（part-select）
- 数组元素的位选择和域选择可以当做操作数
- 连接操作生成的数据可以当做操作数
- 函数调用可以当做操作数



# 表达式

## ■ 操作数

### — 位选择的使用规则

- vector net; vector reg; integer; time; parameter
- 索引值可以是一个表达式
- 索引值超出范围，返回x
- 索引位为x/z，返回x

```
reg [3:0] regA;  
integer intB;  
regA[2] = intB[31]
```



# 表达式

## ■ 操作数

### — 域选择的使用规则

- vector net; vector reg; integer; time; parameter
- 对标量、real、realtime不可使用域选择
- 完全超出范围：read->x, write->无意义
- 部分超出范围：read->正常值+x; write->截断
- 两种使用方法

vect[msb : lsb]

msb, lsb, width必须为常数

vect[msb -: width]

vect[lsb +: width]



# 表达式

## ■ 操作数

### — 示例

reg [31 : 0] vect;

integer index;

vect[0 +: 8];

== vect[7 : 0]

vect[15 -: 8];

== vect[15 : 8]

vect[index + :4];

variable index + fixed width

reg[0 : 31] vect\_b;

vect\_b[0 +: 8];

== vect\_b[0 : 7]



# 表达式

## ■ 操作数位长

- 思考：16bit 的数据A 与 16bit的数据B进行加法运算应该如何选择位宽？

```
reg [15 : 0] a, b;
```

```
reg [15 : 0] sumA;
```

```
reg [16 : 0] sumB;
```

```
sumA = a + b;           //expression evaluates using 16 bits
```

```
sumB = a + b;           //expression evaluates using 17 bits
```



# 表达式

## ■ 操作数位长

### — 位长规则

- 自决定 (self-determined)
  - ✓ 缩位运算、拼接运算等
- 上下文决定 (context-determined)
  - ✓ 加法运算，位运算
- 混合型
  - ✓ 移位运算，条件运算



# 表达式



Expression	Bit length	Comments
Unsize constant number <sup>a</sup>	Same as integer	
Sized constant number	As given	
i op j, where op is: + - * / % &   ^ ~ ^~	max(L(i),L(j))	
op i, where op is: + - ~	L(i)	
i op j, where op is: == != == != > >= < <=	1 bit	Operands are sized to max(L(i),L(j))
i op j, where op is: &&	1 bit	All operands are self-determined
op i, where op is: & ~&   ~  ^ ~^ ~^!	1 bit	All operands are self-determined
i op j, where op is: >> << ** >>> <<<	L(i)	j is self-determined
i ? j : k	max(L(j),L(k))	i is self-determined
{i,...j}	L(i)+..+L(j)	All operands are self-determined
{i{j,...k}}	i * (L(j)+..+L(k))	All operands are self-determined



# 表达式

## ■ 表达式位长

### — 示例（随堂测试）

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

```
reg [2:0] c;
```

```
reg[4:0] d;
```

```
initial begin
```

```
  a = 4'b0000;
```

```
  b = 4'b1111;
```

```
  c = 3'b100
```

```
  d = b + c;
```

```
$displayb(b + c);
```

```
$displayb(d);
```

```
$displayb(d? c : a);
```

```
$displayb(d? a : c);
```

```
end
```

b + c : 0011

d : 10011

d? c : a : 0100

d? a : c : 0000

给出上面4个输出结果



## ■ 符号控制

- 符号只依赖于操作数，不依赖于左端项
- 不指明进制的十进制数是符号数
- 指明进制，但没有sign标志的是无符号数
- 位选择/域选择/连接操作/比较操作的结果是无符号数，不管操作数是否有符号，即使域选择的内容是整个向量
- real强制转换integer时的结果是有符号数
- 任何自决定操作数的符号和位长由操作数自身决定
- 符号数与无符号数进行运算，按无符号数进行



## 第二章 硬件描述语言——Verilog

- 硬件描述语言概述
- 程序结构
- 语言要素
- 数据类型
- 表达式和运算符
- 建模方式



# 建模方式

## ■ 结构级建模

- 根据逻辑电路图中的结构（电路原理图），实例引用Verilog中内置的基本门级元件、用户定义的原件或其他模块，来描述电路结构图中的元件以及元件之间的连接关系

## ■ 内置门级元件

- 内置12个门级元件（primitive）模型，引用门级元件对电路图进行描述，称为门级建模

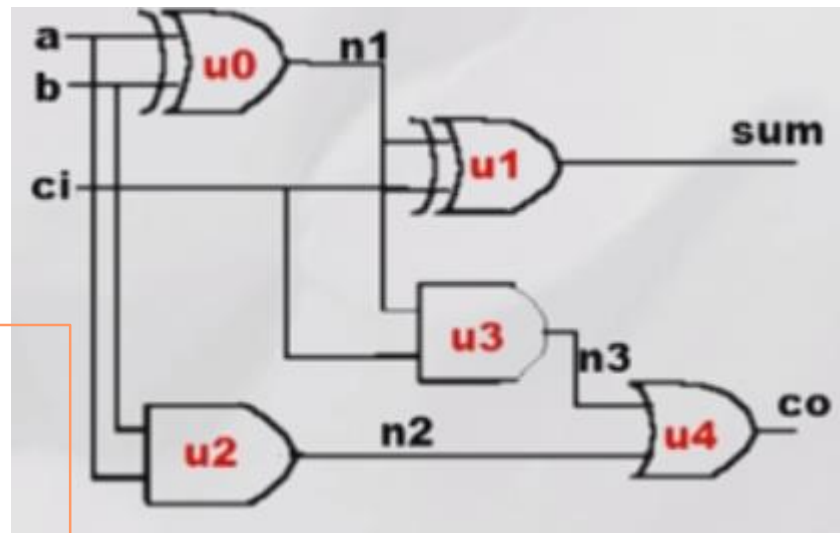
## ■ 用户自定义模块

# 建模方式

## ■ 结构级建模示例

### — 1位全加器

```
module add(a, b, ci, sum, co);  
    input a, b, ci;  
    output sum, co;  
  
    xor u0(n1, a, b),  
        u1(sum, n1, ci);  
    and u2(n2, a, b),  
        u3(n3, n1, ci);  
    or u4(co, n2, n3);  
  
endmodule
```





# 建模方式

## ■ 结构级建模的局限性

- 基于电路原理图，该方式要求具有逻辑电路原理图相关知识，对设计者要求过高
- 电路复杂，逻辑门较多时，效率低



# 建模方式

## ■ 数据流建模

### — 运算符

运算符分类	所含运算符
算术运算符	$+$ , $-$ , $*$ , $/$ , $\%$ , $**$
关系运算符	$<$ , $>$ , $<=$ , $>=$
相等运算符	$==$ , $!=$ , $===$ , $!==$
逻辑运算符	$!$ , $\&\&$ , $  $
按位运算符	$\sim$ , $\&$ , $ $ , $\wedge$ , $\wedge\sim$ or $\sim\wedge$
缩位运算符	$\&$ , $\sim\&$ , $ $ , $\sim $ , $\wedge$ , $\wedge\sim$ or $\sim\wedge$
移位运算符	$<<$ , $>>$ , $<<<$ , $>>>$
条件运算符	$?:$
拼接和复制运算符	$\{ \}$ , $\{ \{, \} \}$



# 建模方式



## ■ 数据流建模

- `assign` 变量名=表达式;



- 只能对`wire`性变量进行赋值



# 建模方式

## ■ 数据流建模示例

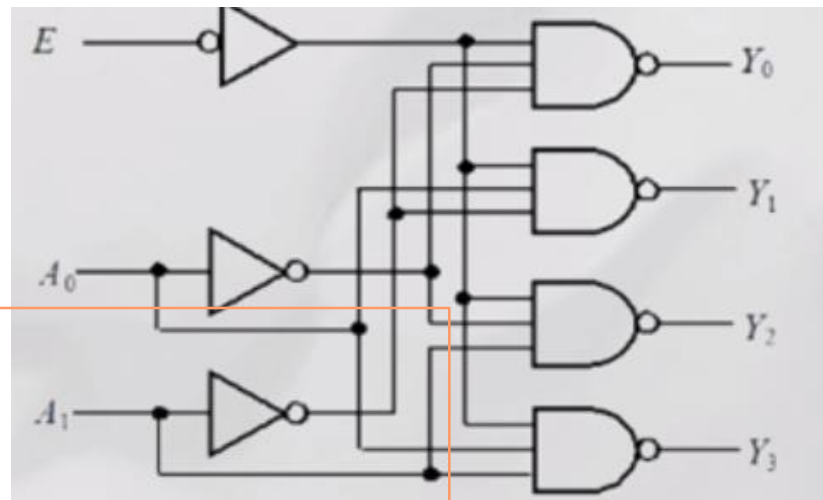
### — 2-4译码器

```
module decoder (A0, A1, E, Y);
```

```
    input A0, A1, E;  
    output [3:0] Y;
```

```
    assign Y[0] = ~(~A1 & ~A0 & ~E)  
    assign Y[1] = ~(~A1 & A0 & ~E)  
    assign Y[2] = ~(A1 & ~A0 & ~E)  
    assign Y[3] = ~(A1 & A0 & ~E)
```

```
endmodule
```



$$\begin{aligned} Y_0 &= \overline{A_1} \cdot \overline{A_0} \cdot \overline{E} \\ Y_1 &= \overline{A_1} \cdot A_0 \cdot \overline{E} \\ Y_2 &= A_1 \cdot \overline{A_0} \cdot \overline{E} \\ Y_3 &= A_1 \cdot A_0 \cdot \overline{E} \end{aligned}$$



# 建模方式

## ■ 数据流建模小结

### — 建模方式

- 电路图原理
- 逻辑代数表达式
- 电路实际行为

### — 局限性

- assign 等号左边必须为线网类型变量
- 仅适合与组合逻辑电路的功能描述



# 建模方式

## ■ 行为级建模

- 描述数字逻辑电路的功能和算法
- **always** : 过程性赋值
  - 逻辑表达式
  - 条件语句 (**if**)
  - 多路分支语句 (**case-endcase**)
  - 循环语句 (**for, while**)
- **initial** : 面向仿真的过程语句



# 建模方式

## ■ always结构型语句

`always @ (A or B)`

`always @ (*)`

`always @ ()`

事件控制运算符

敏感事件表

`always @ (事件控制表达式)`

顺序语句块

`begin:` 块名  
块内局部变量的定义;  
过程赋值语句1;  
.  
.  
.  
过程赋值语句n;  
`end`

等号左边变量  
必须为`reg`型,  
而不能是`wire`  
型



## 第二章 硬件描述语言——Verilog

- 硬件描述语言概述
- 程序结构
- 语言要素
- 数据类型
- 表达式和运算符
- 建模方式
- 仿真测试



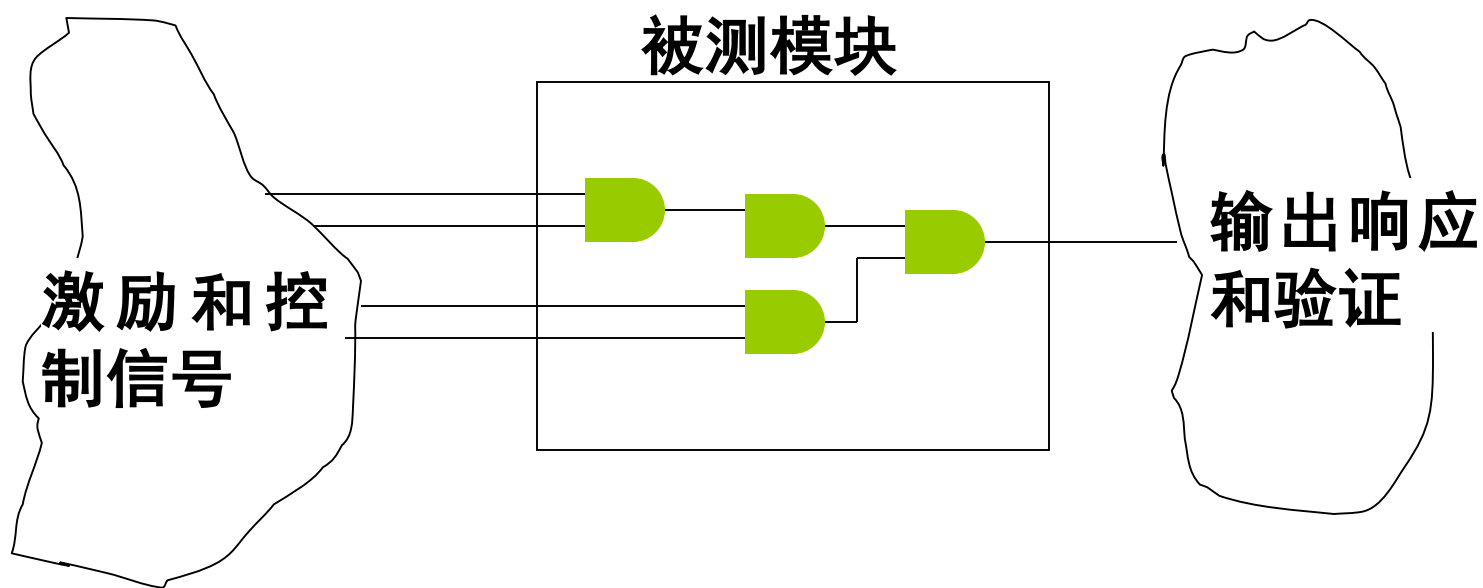
# 仿真测试

- 目的：测试通过Verilog编写的电路逻辑功能是否正确
- 方法：使用Verilog编写Testbench,也称为测试模块，并通过仿真软件ModelSim进行仿真
- Testbench功能
  - 产生模拟激励（波形）——加入vpulse
  - 将产生的激励加入到被测试模块
  - 将输出相应与期望进行比较



# 仿真测试

## ■ 原理示意图





# 仿真测试



## ■ Testbench结构

```
module circuit_tb;
```

参数定义; //可选

数据类型定义; // `wire/reg`等

调用待测试模块

```
initial always
```

产生激励信号

实例化待测试模块

监控和比较输出响应

```
endmodule
```



# 仿真测试

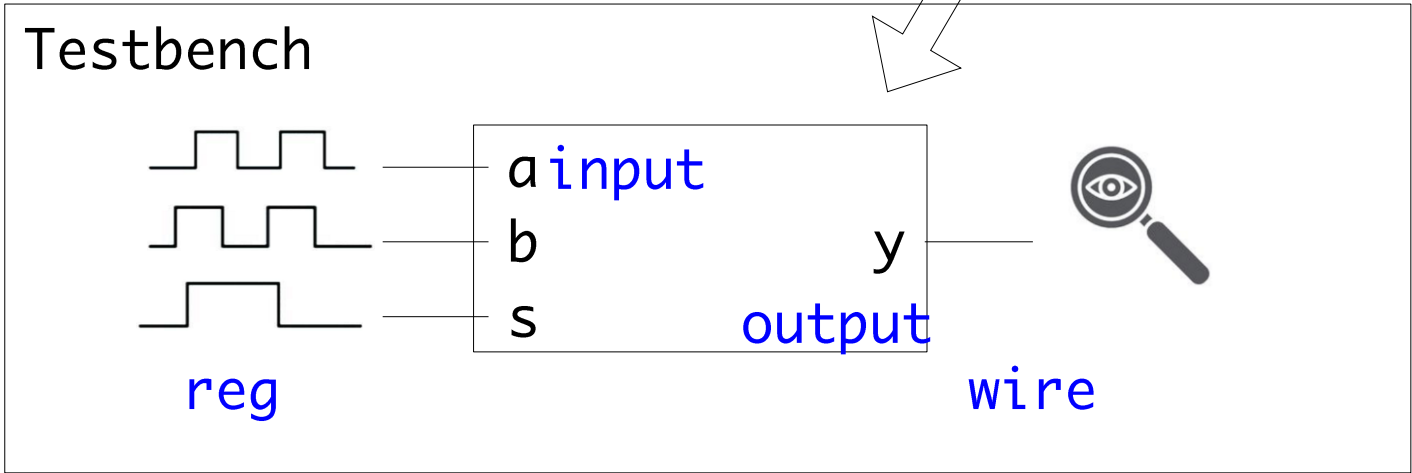
## ■ 电路仿真实例

— 以2路选择器为例1

```
module mux21 (a, b, s, y)
    input a, b, s;
    output y;
    assign y = s? a : b;
endmodule
```

待测  
模块

实例化





# 仿真测试

## ■ 电路仿真实例

时间基准单位/时间精度

```
`timescale 1ns / 10ps
module mux21_tb;
    reg a, b, s;
    wire y;
    mux21 instance1(.a(a), .b(b), .s(s), .y(y));
    initial
    begin
        a = 0; b = 0; s = 0;
        #5 s = 1;
        #5 a = 1; s = 0;
        #5 s = 1;
        #5 a = 0; b = 1; s = 0;
        #5 s = 1;
        #5 a = 1; s = 0;
    end
endmodule
```

实例化待测模块

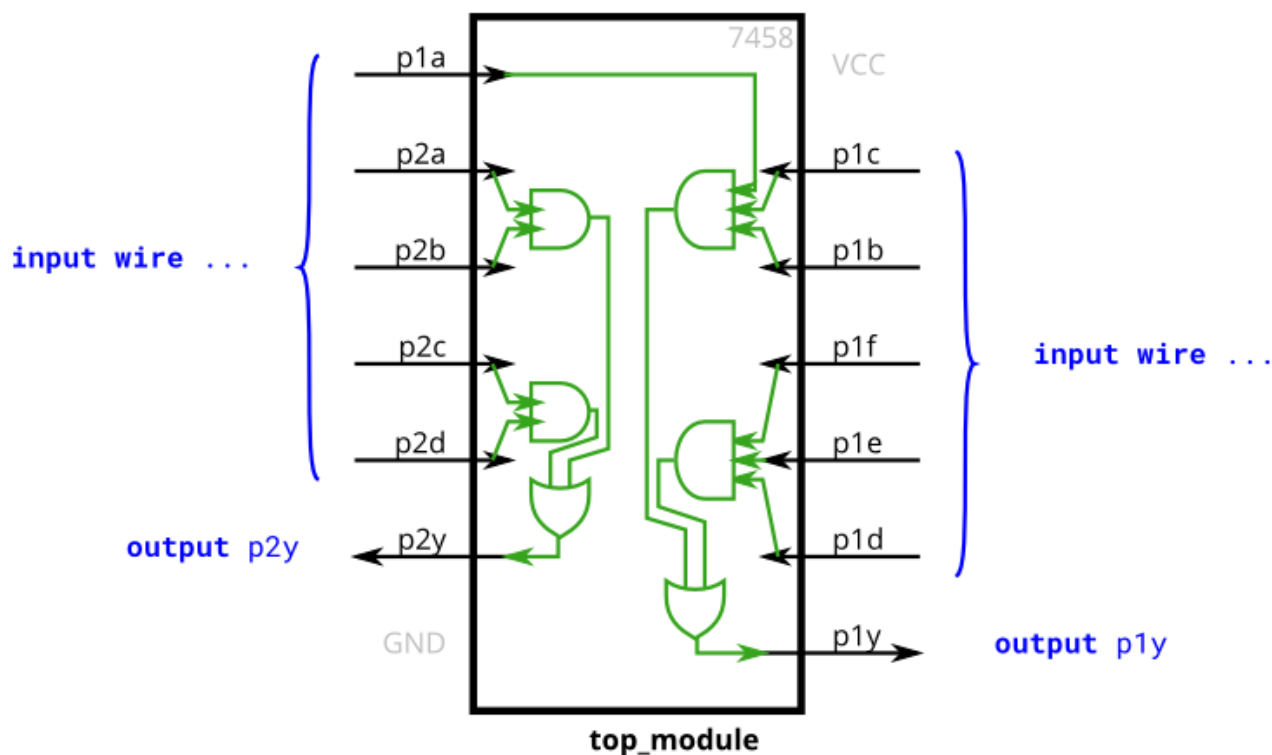


# 作业

- 按照要求利用Verilog HDL描述给定电路，并设计testbench文件
- 服务器地址：10.111.3.128
- 软件：modelSim
- 启动命令：vsim

# 作业1

- 分别用结构级和数据流方式对7458芯片电路进行建模，并设计Testbench文件。





# 作业2

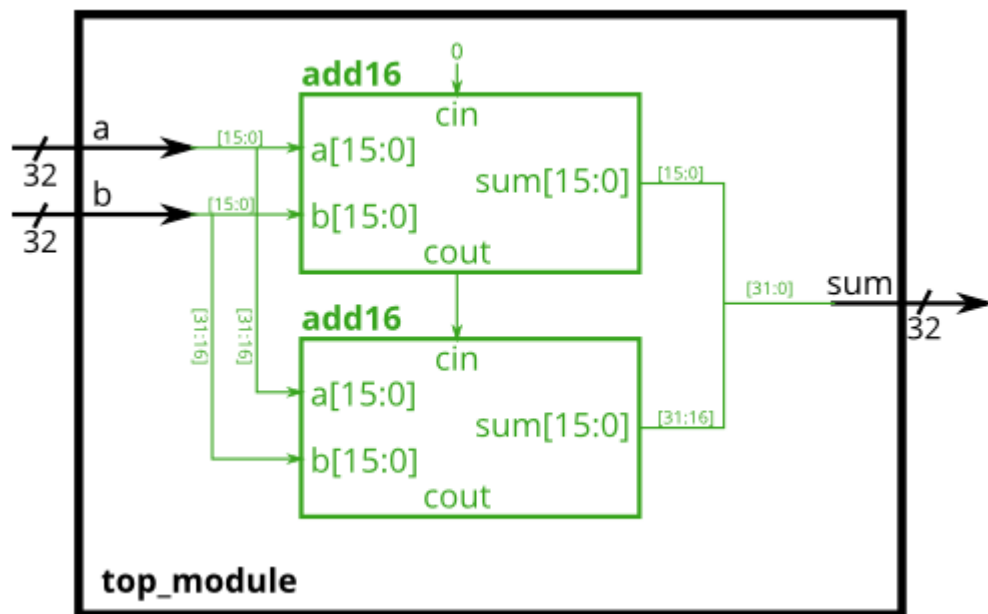
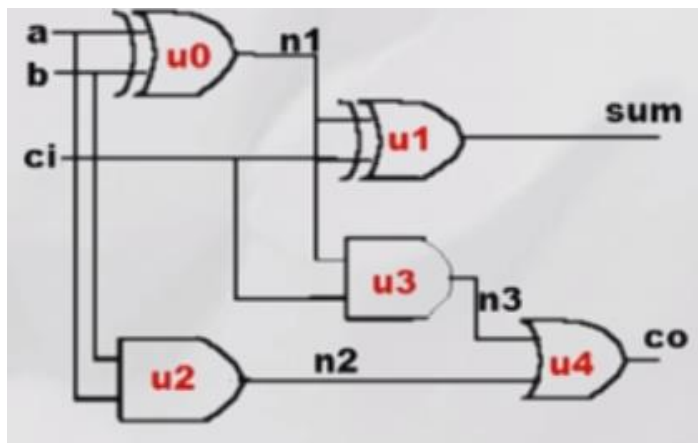
- 32位的向量可以看做由4个字节（byte）组成，每个字节包含8位（bit）。设计Verilog HDL代码实现向量按字节倒序输出。
  - 示例：AaaaaaaaaBbbbbbbbCcccccccDddddddd => DdddddddCcccccccBbbbbbbbAaaaaaaaa



# 作业3

- 1. 任意采用一种建模方式设计一个16位加法器
  - `add16(a, b, cin, sum, cout)`
- 2. 调用16位加法器，设计一个32位加法器
  - `add32(a, b, cin, sum, cout)`
  - 分别采用顺序连接和命名端口连接两种方式
- 3. 编写Testbench 文件
  
- 电路图见下一页

# 作业3







# Verilog 基本要素图

## Verilog HDL基本要素图

