



硬件编程语言Verilog

齐悦

qiyuee@ustb.edu.cn



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**
 - 都是**IEEE**标准
 - 都被**ASIC**和**FPGA**综合工具支持



Verilog的历史

- 1983年，GDA(GateWay Design Automation)公司的Phil Moorby创立Verilog HDL。Phi Moorby后来成为Verilog-XL的主要设计者和Cadence公司的第一个合伙人。
- 1990年，Cadence公司收购了GDA公司
- 1991年，Cadence公司公开发表Verilog语言，成立了OVI(Open Verilog International)组织来负责Verilog HDL语言的发展。
- 1995年制定了Verilog HDL的IEEE标准，即IEEE1364。
- 2001年，Verilog标准进一步完善，形成1364-2001即目前主流版本。
- 2005年，出现修正版本1364-2005。



Verilog的特点

- 描述电路的行为
- 依靠EDA工具综合出具体的电路
- 优点
 - 工艺无关性
 - 可移植性
 - 易于维护



Verilog的抽象层次

- **Verilog**模型可以是实际电路的不同级别的抽象。这些抽象的级别和它们对应的模型类型共有以下五种
 - 系统级(system): 用高级语言结构实现设计模块的外部性能模型
 - 算法级(algorithmic): 用高级语言结构实现设计算法的模型
 - **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): 描述逻辑门以及逻辑门之间的连接的模型
 - 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



Verilog HDL与C语言比较

- **Verilog HDL**与**C**语言在很多地方非常相似。
- **C**语言的各函数之间是串行的，而**Verilog**的各个模块间是并行的

C语言	Verilog	功能	C语言	Verilog	功能
+	+	加	>=	>=	大于等于
-	-	减	<=	<=	小于等于
*	*	乘	==	==	等于
/	/	除	!=	!=	不等于
%	%	取模	~	~	取反
!	!	逻辑非	&	&	按位与
&&	&&	逻辑与			按位或
		逻辑或	^	^	按位异或
>	>	大于	<<	<<	左移
<	<	小于	>>	>>	右移

设计实例—初探Verilog

```
module Add_half ( sum, c_out, a, b );
```

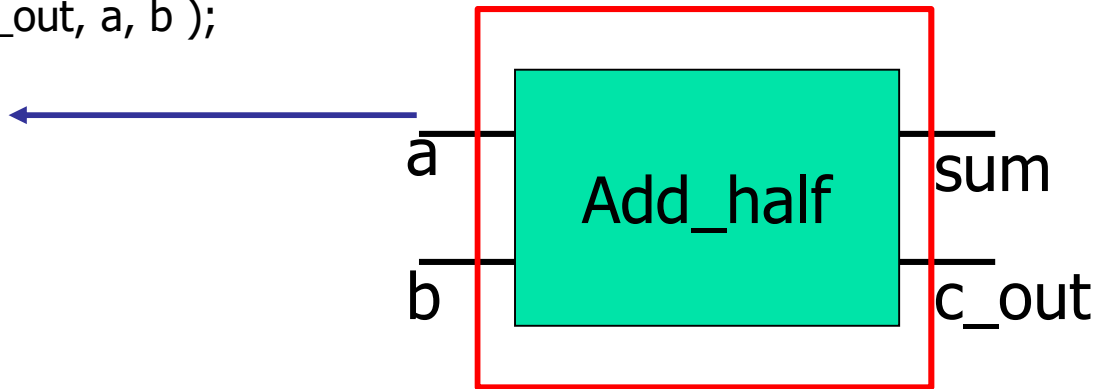
```
  input  a, b;
```

```
  output sum, c_out;
```

```
  xor (sum, a, b);
```

```
  and (c_out, a, b);
```

```
endmodule
```



- 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

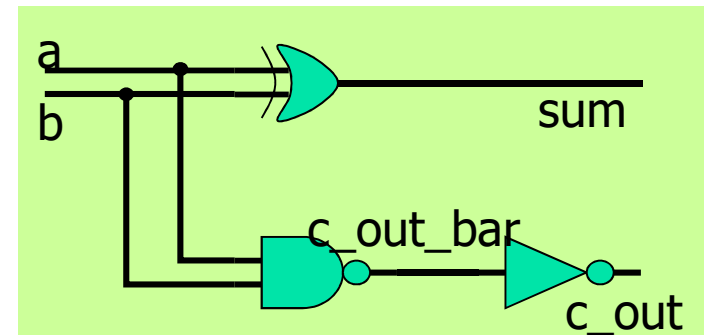
设计实例—初探Verilog

```
module Add_half ( Module name sum, c_out, a, b );  
  Module ports  
  input      a, b;  
  output    sum, c_out;  
  wire      c_out_bar;  
  
  xor (sum, a, b);  
  nand (c_out_bar, a, b);  
  not (c_out, c_out_bar);  
  
endmodule  
Verilog keywords
```

Declaration of port modes

Declaration of internal signal

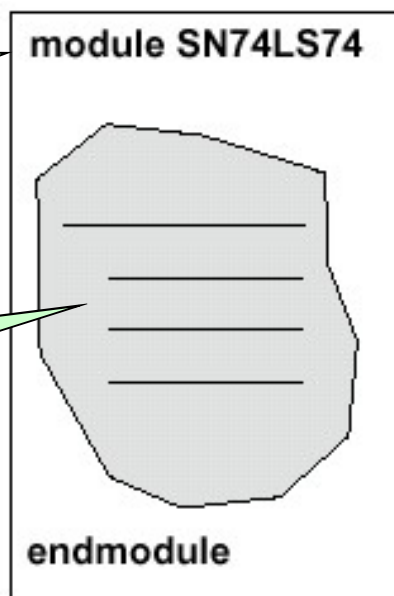
Instantiation of primitive gates



语言的主要特点—module模块

module是层次化设计的基本构件

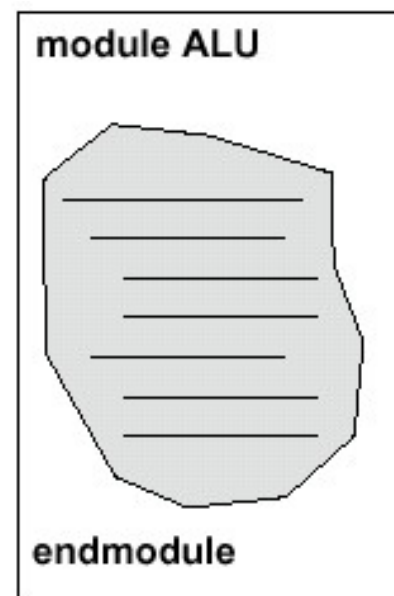
逻辑描述放在**module**内部



**Module Name
& Port List**

数据类型定义
wire/reg

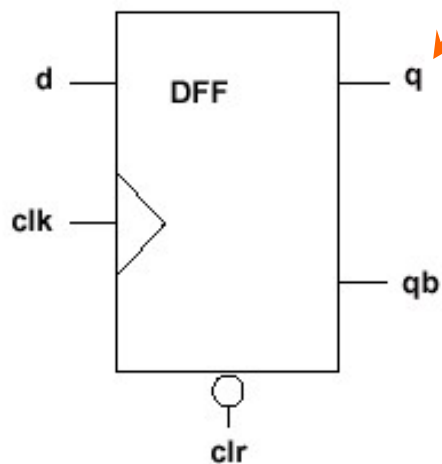
逻辑功能描述



- **module**能够表示：
 - 物理块，如IC或ASIC单元
 - 逻辑块，如一个CPU设计的ALU部分
 - 整个系统

语言的主要特点—模块端口

端口等价于硬件的引脚(pin)



```
module DFF (d, clk, clr, q, qb);  
  input d, clk, clr;  
  output q, qb;  
  
  // Internal logic represented by a grey octagon  
  
endmodule
```

端口在模块名字后的括号中列出

Verilog中有三种端口类型:

input - 输入端口

output - 输出端口

inout - 双向端口

- 注意模块的名称**DFF**，端口列表及说明
- 模块通过端口与外部通信



- 面向RTL的Verilog语法
 - 面向综合的**Verilog**语法子集
 - 面向测试的Verilog语法子集



Verilog语言要素

- 空格和注释
- 操作符
- 数
- 标识符
- 关键词
- 常用数据类型

空格和注释

- **Verilog** 格式比较自由，代码可跨行、也可多条语句写在一行
- 空格只起分隔符的作用
- 注释：

```
initial begin  clk = 0; forever  #10 clk = ~clk ;  
end
```

```
/* *****  
   code to generate clock signal  
   ***** */
```

多行注释

```
initial begin  
  clk = 0;  
  forever  
    #10 clk = ~clk ;    // invert clock  
end
```

单行注释



Verilog运算符（4.6.5节自学）

- 算术操作: **+, -, *, /, %**（取模）
- 关系操作: **>, <, >=, <=, ==**（相等）, **!=**（不等）
- 逻辑操作: **&&, ||, !,**
- 位操作: **~**（非）, **&, |, ^**（异或）, **~^**（同或）
- 归约操作: **&, ~&, |, ~|, ^, ~^**
- 移位操作: **<<, >>**
- 条件操作: **?:**
- 连接和复制: **{ , }**

数的表示方

X可以用来定义十六进制数的**4**位二进制状态，八进制数的**3**位，二进制数的**1**位。**Z**的表示方法同**X**类似。

- **<size>'<base><value>**
 - **Size:** 以**bit**为单位
 - **Base:** **b**(二进制),**o**(八进制),**d**(十进制),**h**(16进制)
 - **Value:**和进制相应的数值, **x, z, ?** (**x,z**不区分大小写)

- 例

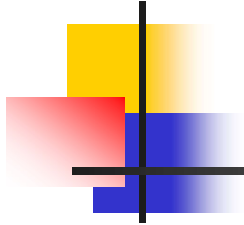
默认**32**位十进制数

- **16**
- **8'd16**
- **8'h10**
- **8'b0001_0010**
- **32'bx**
- **2'b?**

下划线“**_**”是为了增加可读性

//**32**位**x**

//**?**、**z**、**Z**都表示高阻



- 如果定义的长度比为常量指定的长度长，通常在左边填**0**补位。但是如果数最左边一位为**x**或**z**，就相应地用**x**或**z**在左边补位。

10'b10 左边添**0**补齐, **0000000010**

10'bx0x1 左边添**x**补齐, **xxxxxxxx0x1**

- 如果定义的位宽比实际的位数小，那么最左边的位相应地被截断：

3'b1001_0011 //与**3'b011**相等

5'h0FFF //与**5'h1F**相等



字符串

- 用“ ”括起来的字符，必须在一行内写完
- 在字符串中可以用 **C** 语言中的各种格式控制符，如 `\t`, `\n`, `\\` ...
- 字符串的定义

```
reg [8*12:1] stringvar;
```

```
initial begin
```

```
    stringvar = " Hello World!";
```

```
end
```

标识符

- 标识符就是用户为程序中的**Verilog** 对象所起的名字
- **a-z, A-Z, 0-9, _, \$**
- 标识符必须以字母或者下横线开头
- 标识符最长可以达到**1023**个字符
- **Verilog**语言是大小写敏感的

```
module adder(a,b,cin,s,cout);  
  input a,b,cin;  
  output s,cout;  
  ...  
endmodule
```

标识符



关键词

- **Verilog** 中大小写敏感
- 所有的**Verilog** 关键词都是小写的

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	rnmos	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			

常用数据类型

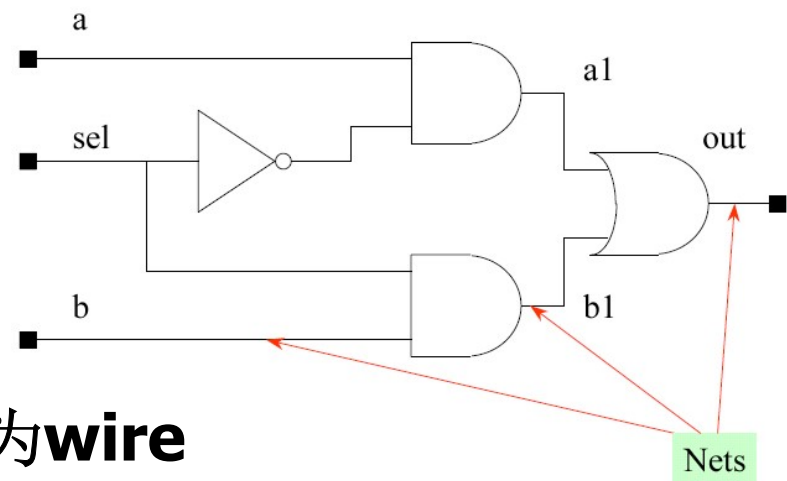
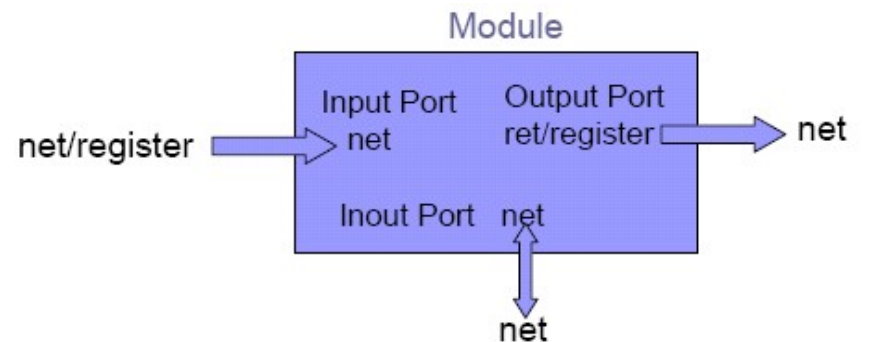
- **wire**（线网**net**）

- 表示元件之间的物理连接
- **wire a,b;**
- **wire [3:0] address;**
- **wire**的缺省值是**z**

- **reg**（寄存器**register**）

- **always/initial**过程的输出
- **reg [5:1] state, newstate;**
- **reg**的缺省值是**x**

- 除了应该定义为**reg**的都定义为**wire**





信号的声明格式

wire型变量的声明格式:

wire [n-1:0] 数据名**1**, 数据名**2**,, 数据名**m**;

reg型变量的声明格式:

reg [n-1:0] 数据名**1**, 数据名**2**,, 数据名**m**;



net和register声明举例

```
reg a;                                // 一个一位寄存器  
wire w;                             // 一个一位wire类型net  
reg [3: 0] v;                       // 从MSB到LSB的4位寄存器  
reg [7: 0] m, n;                   // 两个8位寄存器  
wire [31: 0] databus, addrbus; // 两个32位总线  
wire [35: 4] abus, bbus;
```



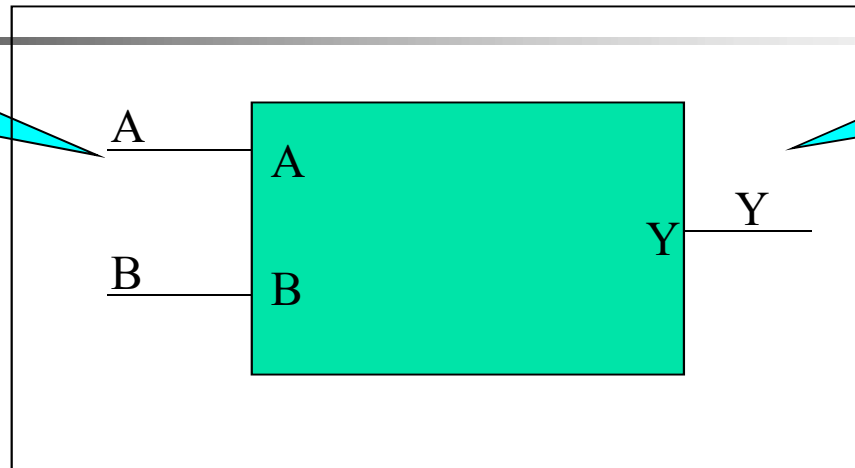
例

- 多位**wire** 型数据可按下面方法使用
wire[7:0] in, out; //定义两个8位wire型向量
assign out=in; // assign 持续赋值语句
- 使用多位数据中的几位
wire[7:0] out;
wire[3:0] in;
wire a;
assign out[5:2]=in;
assign out[7]=a;

选择正确的数据类型

输入端口可以由
net/register驱动，但
输入端口只能是**net**

双向端口输入/输出
只能是**net**类型



输出端口可以是
net/register类型，输
出端口只能驱动**net**

```
module top;
wire y;
reg a, b;
    DUT u1 (y, a, b);
    initial begin
        a = 0; b = 0;
        #5 a = 1;
    end
endmodule
```

在过程块中只能给
register类型赋值

```
module DUT (Y, A, B);
output Y;
input A, B;
wire Y, A, B;
    and (Y, A, B);
endmodule
```

若Y, A, B说明为
reg则会产生错误。



存储器(memory)

- 在**Verilog**中可以说明一个寄存器数组。

integer NUMS [7: 0]; // 包含8个整数数组变量

- **reg**类型的数组通常用于描述**存储器**

其语法为: **reg [MSB:LSB] <memory_name> [first_addr:last_addr];**

[MSB:LSB] 定义存储器字的位数

[first_addr:last_addr] 定义存储器的深度

例如:

reg [15:0] MEM [0:1023]; // 1K x 16存储器

reg [7:0] PREP ['hFFFE: 'hFFFF]; // 2 x 8存储器

- 描述存储器

reg [15: 0] MEM3 [1023: 0];

存储器寻址

- 存储器元素可以通过存储器索引 (**index**)寻址, 也就是给出元素在存储器的位置来寻址。

mem_name [addr_expr]

- **Verilog**不支持多维数组。也就是说只能对存储器字进行寻址, 而不能对存储器中一个字的位寻址。

```
module mems;
```

```
reg [8: 1] mema [0: 255]; // declare memory called mema
```

```
reg [8: 1] mem_word; // temp register called mem_word
```

```
...
```

```
initial
```

```
begin
```

```
    $displayb( mema[5]); //显示存储器中第6个字的内容
```

```
    mem_word = mema[5];
```

```
    $displayb( mem_word[8]); // 显示第6个字的最高有效位
```

```
end
```

```
endmodule
```

若要对存储器字的某些位存取, 只能通过暂寄存器传递



说明

- 寄存器赋值可以在一条赋值语句中完成，但是存储器不可以。因此在存储器被赋值时，需要定义一个索引。

```
reg [5:1] Dig;    // Dig为5位寄存器。
```

```
...
```

```
Dig = 5'b11011;  // 赋值正确
```

```
reg Bog[5:1];    // Bog为5个1位寄存器组成的的存储器组
```

```
...
```

```
Bog = 5'b11011;  // 赋值不正确
```



编译预处理

- 符号说明一个编译预处理
- 这些编译预处理使仿真编译器进行一些特殊的操作
- 编译预处理一直保持有效直到被覆盖或解除

宏定义- `define

- 提供了一种简单的文本替换的功能

`define <macro_name> <macro_text>

在编译时<macro_text>替换<macro_name>。可提高描述的可读性。

```
`define not_delay #1
`define and_delay #2
`define or_delay #1
module MUX2_1 (out, a, b, sel);
output out;
input a, b, sel;
    not `not_delay not1( sel_, sel);
    and `and_delay and1( a1, a, sel_);
    and `and_delay and2( b1, b, sel);
    or `or_delay or1( out, a1, b1);
endmodule
```

定义not_delay

使用not_delay



文本包含-`include


- 在当前内容中插入一个文件

格式: **`include "<file_name>"**

如 **`include "global.v"**

`include "parts/count.v"

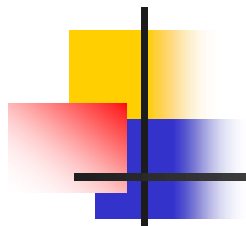
`include "../..../library/mux.v "



可以是相对路径或绝对路径

- 说明

- **include**保存文件中的全局的或经常用到的一些定义, 如文本宏
- 一个**`include**命令只能指定一个被包含的文件
- 文件包含是可以嵌套的



VERILOG电路描述



电路描述手段

■ Verilog电路描述

- 连续赋值语句 (**assign**)
- 过程 (**always**)
 - 阻塞赋值(=)
 - 非阻塞赋值(<=)
 - **if**语句
 - **case**语句
- 元件例化

连续赋值语句 (assign)

- 实现组合逻辑电路

```
module test(out2,out1,in);  
  output out2,out1;  
  input in;  
  ...  
  wire a=b&c; //declare and assign  
  
  assign out2 = ~ in ;  
  assign out1 = sel? i1:i0;  
endmodule
```

可以是复杂的布尔函数

二选一MUX

sel为' 1'时选择i1; 否则i0



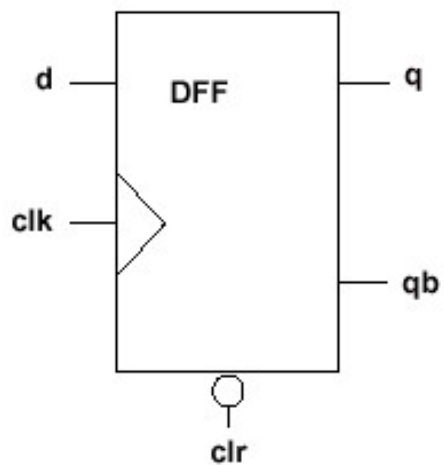
元件例化

AND u1(a,b,and_out);

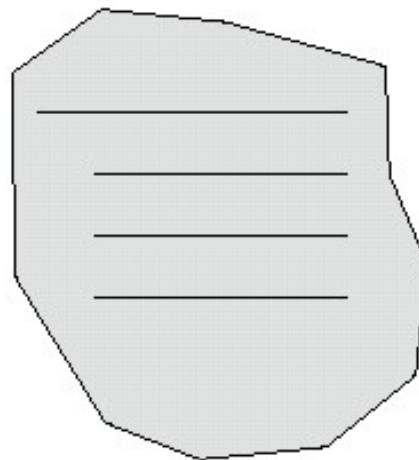
AND u1(.a(a), .b(b), .o(and_out));

- 格式:
 - *模块名 <实例名> (<端口列表>);*
 - 端口列表有两种表示方式
 - 端口名字关联:
(. 端口名 (信号值表达式), . 端口名 (信号值表达式), ……)
 - 位置关联: 隐式给出端口与信号之间的关系
(信号值表达式, 信号值表达式, ……)
- 例化的端口列表中信号的顺序要与该模块定义的端口列表中端口顺序严格一致

元件例化

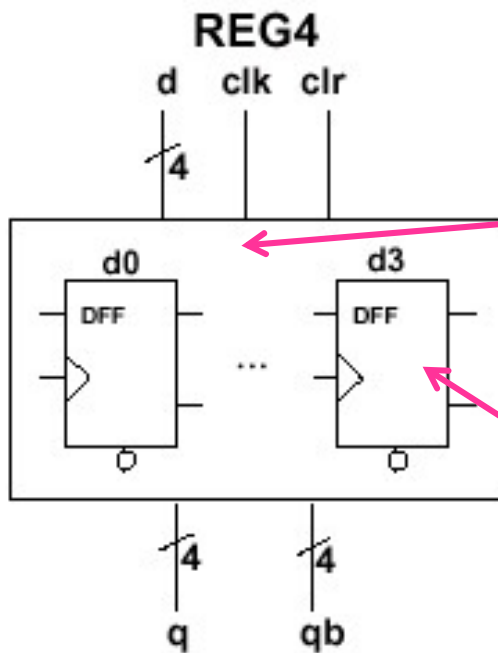


```
module DFF (d, clk, clr, q, qb);  
  input d, clk, clr;  
  output q, qb;
```



```
endmodule
```

元件例化



```
module DFF (d, clk, clr, q, qb);
```

```
....
```

```
endmodule
```

```
module REG4( d, clk, clr, q, qb);
```

```
output [3: 0] q, qb;
```

```
input [3: 0] d;
```

```
input clk, clr;
```

```
DFF d0 (d[0], clk, clr, q[0], qb[ 0]);
```

```
DFF d1 (d[1], clk, clr, q[1], qb[ 1]);
```

```
DFF d2 (d[2], clk, clr, q[2], qb[ 2]);
```

```
DFF d3 (d[3], clk, clr, q[3], qb[ 3]);
```

```
endmodule
```

过程语句

任意边沿：由输入信号中任意一个电平发生变化所引起

- 过程 (**always**)
 - 阻塞赋值(=)
 - 非阻塞赋值(<=)
 - **if**语句
 - **case**语句

```
always @(a or b or c or ...)  
begin  
    语句块(=, if, case)  
end
```

```
always @(posedge/negedge sig or...)  
begin  
    语句块(<=, if, case)  
end
```

由单个跳变沿所引起

- 两个或更多**always**模块是同时执行的
- **always** 模块描述组合逻辑电路时，用阻塞赋值语句
- **always** 模块描述时序逻辑电路时，用非阻塞赋值语句



if语句

```
if ( 表达式 )  
    语句  
else  
    语句
```

```
if ( 表达式 )  
    语句  
else if ( 表达式 )  
    语句  
else  
    语句
```

```
if ( a > b )  
    res = 1;  
else if ( a < c )  
    begin  
        res = 2;  
        out = 4;  
    end  
else  
    res = 3;
```

- 可以多层嵌套。在嵌套`if`序列中，`else`和前面最近的`if`相配对。
- 为确保正确关联，使用`begin...end`块语句指定其作用域。



case语句

```
`define OP_LOAD 2'b00
`define OP_ADD 2'b01
`define OP_SUB 2'b10

always @(op) begin
  case ( op )
    `OP_LOAD : out = din;
    `OP_SUB : out = a - b;
    default : out = a + b;
  endcase
end
```

- **default**语句可选，在没有任何条件成立时执行
- 如果未说明 **default**，**Verilog**不执行任何动作
- 多个**default**语句非法

阻塞赋值和非阻塞赋值

过程赋值有两类

阻塞过程赋值 /

阻塞过程赋值执行完成后再执行在顺序块内下一条语句。

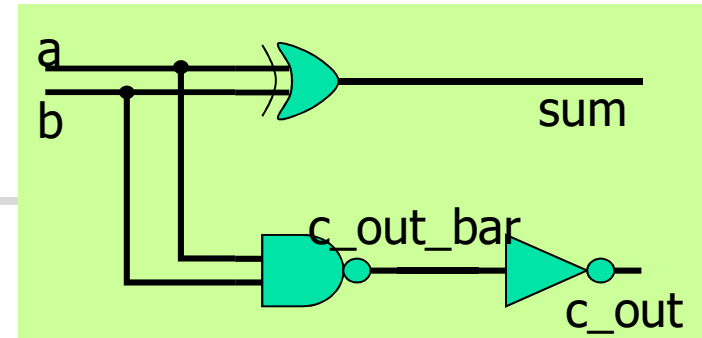
- 在同一个**always/initial**块里不要混用两种赋值语句
- 同一个变量，既进行阻塞赋值，又进行非阻塞赋值，综合时会出错
- 尽量不要在多个不同的**always**块中对同一变量赋值

```
always #5 clk = ~clk;  
always @(posedge clk)  
begin  
    a <= b; // 非阻塞过程赋值  
    b <= a; // 交换a和b值  
end  
endmodule
```

保存结果，并进行调度在时序控制指定时间的赋值。

2. 在经过相应的延迟后，仿真器通过将保存的值赋给左端表达式完成赋值。

电路功能描述例



```
module Add_half ( sum, c_out, a, b );
```

```
  input      a, b;
```

```
  output     sum, c_out;
```

```
  wire       c_out_bar;
```

```
  xor (sum, a, b);
```

```
  nand (c_out_bar, a, b);
```

```
  not (c_out, c_out_bar);
```

```
endmodule
```

行为描述

```
reg    sum, c_out;  
always @ ( a or b ) begin  
    sum = a ^ b;  
    c_out = a & b;  
end
```

结构描述

数据流描述

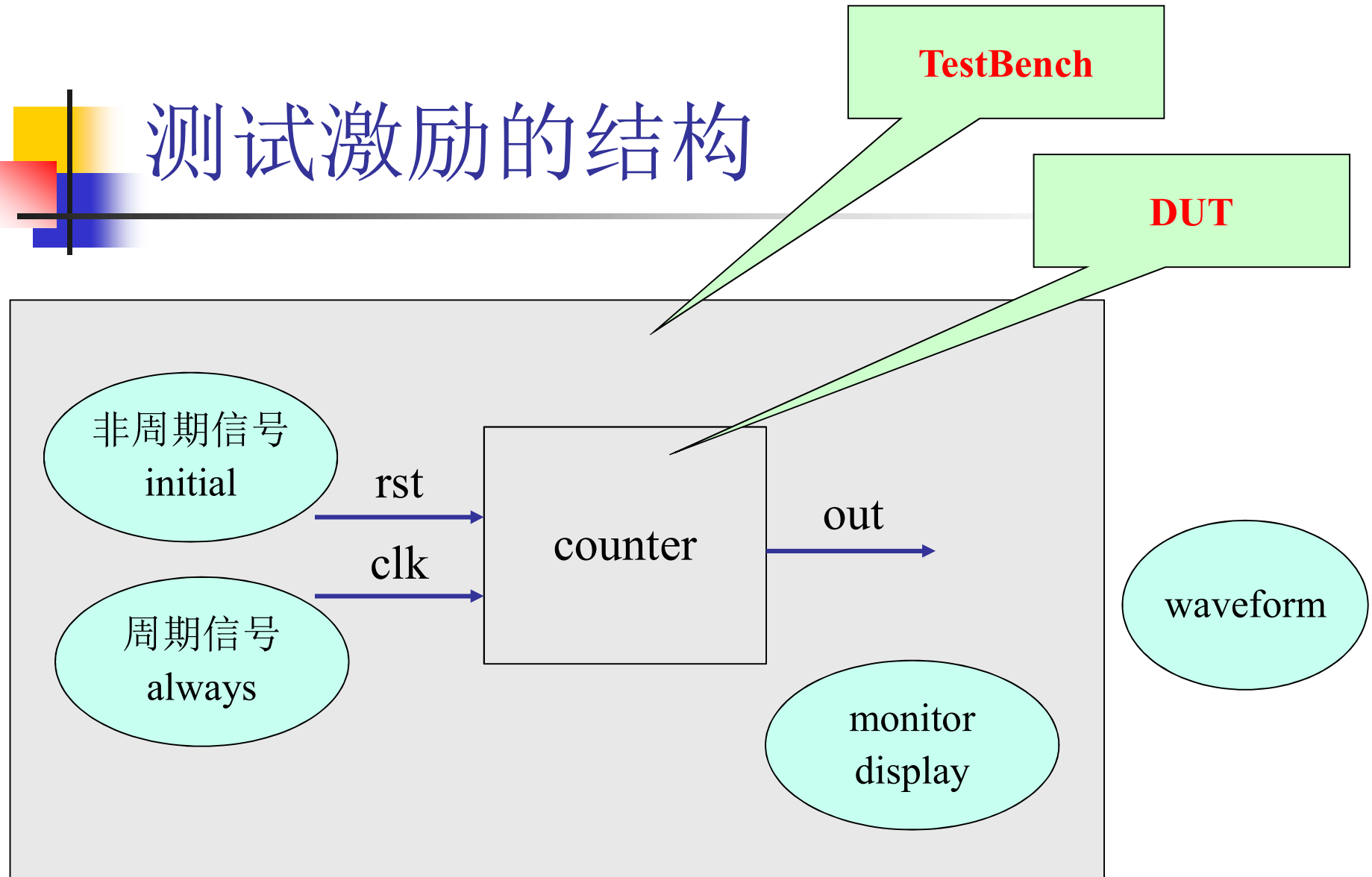
```
assign sum = a ^ b;  
assign c_out = a & b;
```

The logo consists of a black crosshair centered over a square. The square is divided into four quadrants: top-left is yellow, top-right is red, bottom-left is blue, and bottom-right is white. The word "Verilog" is written in a blue serif font to the right of the crosshair.

Verilog

- 面向RTL的Verilog语法
 - 面向综合的Verilog语法子集
 - 面向测试的**Verilog**语法子集

测试激励的结构





被测模块DUT(divece under test)

```
module counter (out, clk, rst);  
    output reg [3:0] out;  
    input clk, rst;  
    wire clk, rst;  
  
    always @ (negedge rst or posedge clk) begin  
        if(!rst)  
            out<=0;  
        else  
            out<=out+1;  
    end  
endmodule
```

- 测试模块可以通过模块名及端口说明使用被测模块。实例化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



测试模块(tb.v)

```
`timescale 10ns/1ns  
module top;  
    // Data type declaration  
  
    // Instantiate modules  
  
    // Apply stimulus  
  
    // Display results  
  
endmodule
```



为什么没
有端口?

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



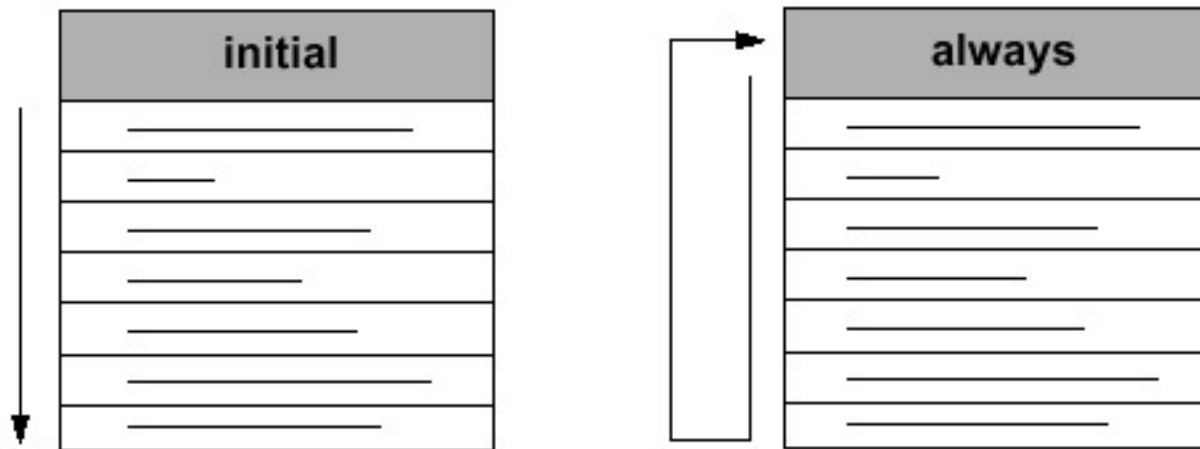
测试模块

```
module top;  
    // Data type declaration  
  
    // Instantiate modules  
    counter cnt (dout, clk, rst);  
    // Apply stimulus  
  
    // Display results  
  
endmodule
```

实例化语句

测试模块 – 过程

- 过程语句有两种：
 - *initial* : 只执行一次
 - *always* : 循环执行



通常采用过程语句进行行为级描述

激励信号在过程语句中描述

过程之间是并行执行的

测试模块 – 激励描述

```
module top;  
  // Data type declaration  
  reg clk, rst;  
  wire [3:0] dout;  
  // counter instance  
  counter cnt (dout, clk, rst);  
  // Apply stimulus  
  initial begin  
    clk = 0;  
    rst = 1;  
    #15 rst = 0;  
    #10 rst = 1;  
    #55 $finish;  
  end  
end
```

只有**always/initial**过程的输出才需要定义为**reg**

端口信号的缺省类型是**wire**，可以省略

```
always  
  #10 clk = ~clk;  
// Display results  
initial begin  
  $monitor($time, "%b %b %b",  
    rst, clk, dout);  
end  
  
endmodule
```



特殊符号 “#”

- 特殊符号 “#” 表示延迟

- 过程赋值语句里的延迟

initial begin #10 rst=1; end

- 门级实例引用的延迟

not #1 not1(nsel, sel);



系统任务和函数

- **\$<identifier>**
- **\$**符号指示这是系统任务或函数，如：
 - 返回当前仿真时间**\$time**
 - 显示/监视信号值(**\$display, \$monitor**)
 - 停止仿真**\$stop**
 - 结束仿真**\$finish**

\$monitor(\$time, "a = %b, b = %h", a, b);

当信号**a**或**b**的值发生变化时，系统任务**\$monitor**显示当前仿真时间、信号**a**值(二进制格式)、信号**b**值(16进制格式)



测试模块 – 响应输出

- **Verilog**的测试模块指中经常会用到的:
 - *\$time* 系统函数, 给出当前仿真时间
 - *\$monitor* 系统任务, 若参数列表中的参数值发生变化, 则在时间单位末显示参数值
 - *\$display* 系统任务, 在当前时间显示参数值

例如:

```
$monitor($time, o, in1, in2);
```

```
$monitor($time, , out, , a, , b, , sel);
```

```
$monitor($time, "%b %h %d %o", sig1, sig2, sig3, sig4);
```



总结initial+always

- **initial**只执行一次，**always**循环执行
- **initial**的用处：
 - 赋初值
 - 产生激励信号
 - 检查输出结果



initial赋初值

```
reg [3:0] out ;  
  
always @(negedge rst  
        or posedge clk)  
begin  
    if ( !rst )  
        out <= 0;  
    else  
        out <= out + 1;  
end
```

```
reg [3:0] out ;  
  
initial out = 0;  
  
always @(posedge clk)  
begin  
    out <= out + 1;  
end
```

- 如果不给信号赋初值？
- 为了仿真，怎么给初值？
 - 初值的各种情况、状态机的各种情况



产生激励信号

■ 时钟信号

```
always #10 clk = ~clk;
```

```
initial begin  
    clk = 0;  
    forever  
        #10 clk = ~clk;  
end
```

■ 复位信号

```
initial begin  
    rst = 1;  
    #15 rst = 0;  
    #10 rst = 1;  
    #55 $finish;  
end
```

```
initial fork  
    rst = 1;  
    #15 rst = 0;  
    #25 rst = 1;  
    #80 $finish;  
join
```

```
initial begin  
    rst <= 1;  
    #15 rst <= 0;  
    #25 rst <= 1;  
    #80 $finish;  
end
```



检查输出信号

```
`timescale 1ns/1ns  
module top;  
reg in;  
wire out;  
assign #1 out=~in;  
initial begin
```

```
    $monitor($time,,  
        "out=%b in=%b",out,in);
```

```
end  
initial begin  
    in = 0;  
    #10 in = 1;  
    #10 in = 0;  
end  
endmodule
```

```
0 out=x in=0  
1 out=1 in=0  
10 out=1 in=1  
11 out=0 in=1  
20 out=0 in=0  
21 out=1 in=0
```

```
`timescale 1ns/1ns  
module top;  
reg in;  
wire out;  
assign #1 out=~in;  
initial begin
```

```
    $display($time,,  
        "out=%b in=%b",out,in);
```

```
end  
initial begin  
    in = 0;  
    #10 in = 1;  
    #10 in = 0;  
end  
endmodule
```

```
0 out=x in=x
```




说明

- **wire** 型变量常用来表示以**assign**语句赋值的组合逻辑信号。
- 输入/输出信号缺省时自动定义为**wire** 型。
- 对综合器而言， **wire** 型信号的每一位可以取**0**， **1**， **X**或**Z**中的任意值。
- **reg** 型数据常用来表示 “**always**” 模块内的指定信号。在 “**always**” 模块内被赋值的每一个信号都必须定义成**reg**型。
- 若**reg**型数据未初始化（即缺省），则初始值为不定状态**X**。



时间尺度-timescale

- **`timescale** 说明时间单位及精度

格式: **`timescale <time_unit> / <time_precision>**

如: **`timescale 1 ns / 100 ps**

time_unit: 延时或时间的测量单位

time_precision: 延时值超出精度要先舍入后使用

- **`timescale** 必须在模块之前出现

```
`timescale 1 ns / 10 ps
```

```
// All time units are in multiples of 1 nanosecond
```

```
module MUX2_1 (out, a, b, sel);
```

```
output out;
```

```
input a, b, sel;
```

```
not #1 not1( sel_, sel);
```

```
and #2 and1( a1, a, sel_);
```

```
and #2 and2( b1, b, sel);
```

```
or #1 or1( out, a1, b1);
```

```
endmodule
```



时间尺度

- **time_precision**不能大于**time_unit**
- **time_precision**和**time_unit**的表示方法: **integer unit_string**
 - **integer**: 可以是**1, 10, 100**
 - **unit_string**: 可以是**s(second), ms(millisecond), us(microsecond), ns(nanosecond), ps(picosecond), fs(femtosecond)**
 - 以上**integer**和**unit_string**可任意组合
- 尽可能地使精度与时间单位接近, 只要满足设计的实际需要就行
 - **precision**是仿真器的仿真时间步长
 - 若**time_unit**与**precision_unit**差别很大将严重影响仿真速度。
 - 如说明一个`**timescale 1s / 1ps**, 则仿真器在**1**秒内要扫描其事件序列**10¹²**次; 而`**timescale 1s/100ms**则只需扫描**10**次。



时间尺度

- 所有**timescale**中的最小值决定仿真时的最小时间单位。因为仿真器必须对整个设计进行精确仿真

```
`timescale 10ns/ 1ns
module1 (...);
    #1.23      // 12ns
    ...
endmodule
`timescale 100ns/ 1ns
module2 (...);
    #1.23      // 123ns
    ...
endmodule
`timescale 1ps/ 100fs
module3 (...);
    #1.23      // 1.23ps
    ...
endmodule
```

连续赋值和过程赋值的比较

```
module assignment_test;
```

```
  reg [3:0] r1, r2;
```

```
  reg [4:0] sum2;
```

```
  wire [4:0] sum1;
```

```
  assign sum1 = r1 + r2;
```

continuous assignment

```
  initial begin
```

```
    r1=4'b0010; r2=4'b1001;
```

```
    sum2 = r1 + r2;
```

```
    $display(" r1  r2  sum1  sum2");
```

```
    $monitorb(r1, r2, sum1, sum2);
```

```
    #10 r1 = 4'b0011;
```

```
  end
```

```
endmodule
```

procedural assignment

Result

r1	r2	sum1	sum2
0010	1001	01011	01011
0011	1001	01100	01011