

*Spring, 2020*

1090140071

# FPGA设计及应用

## Verilog HDL基础1：语法

大连理工大学 电信学部  
夏书峰

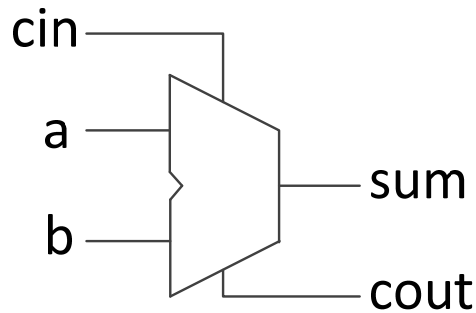
# Verilog HDL基本语法

- 一 Verilog HDL基本知识
- 二 设计流程和通用设计方法论
- 三 Verilog语法基本概念
- 四 Verilog常用语法一模块、数据类型、表达式
- 五 Verilog常用语法一运算符
- 六 Verilog常用语法一语句：条件、循环、结构
- 七 Verilog常用语法一系统任务等
- 八 Verilog编译预处理
- 九 Verilog语法总结

# 一. Verilog HDL—基本知识

- **HDL**—Hardware Description Language—硬件描述语言：是一种用形式化方法来描述数字电路和系统的语言。
- 可以从上层到下层（从抽象到具体）逐层描述设计思想，并用一系列分层次的模块来表示极其复杂的数字系统。
- 利用**EDA**工具仿真和验证，并把要变为物理电路的模块经自动综合工具转换到门级电路网表
- 再用**FPGA**或**ASIC**自动布局布线工具把网表转换为具体电路布线结构的实现

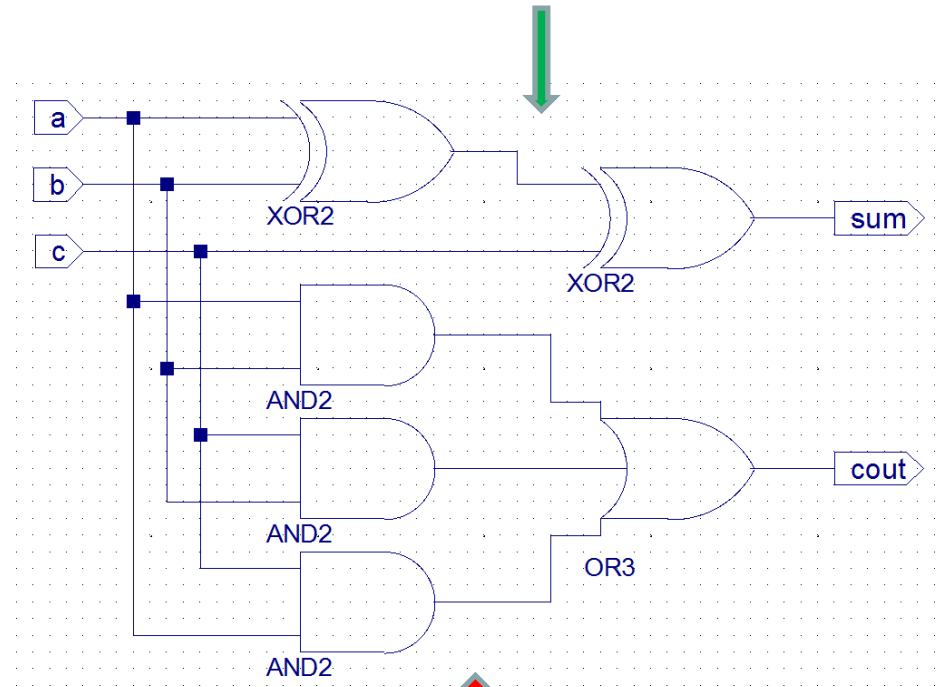
# 1-bit full adder



cin	a	b	cout	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Verilog HDL:

```
assign {cout, sum} = a + b + cin;
```



$$\text{sum} = a \oplus b \oplus \text{cin}$$

$$\text{cout} = a \times b + a \times \text{cin} + b \times \text{cin}$$

# Verilog HDL—历史

- 1983年Verilog HDL由Gateway Design Automation公司Phil Moorby首创，Verilog由C语言发展而来
- 1985年Moorby设计了第一个Verilog-XL仿真器并于1986年提出门级仿真的XL快速算法
- 1989年Cadence公司收购GDA公司，将其私有化
- 1990年Cadence公司成立Open Verilog International组织并将Verilog HDL语言公开
- 1995年由于Verilog HDL语言的优越性，IEEE将其收为标准，即IEEE1364-1995
- 2001年IEEE发布标准IEEE1364-2001，对Verilog HDL语言扩充，使综合和仿真性能大幅提高
- 2002年IEEE发布IEEE1364.1-2002标准规定Verilog语言RTL级综合的语法和语义
- 2005年IEEE发布IEEE1364-2005，是对2001的改进，解决了一些定义不清的问题并纠正了一些错误
- 上述标准可从<https://standards.ieee.org>下载全文

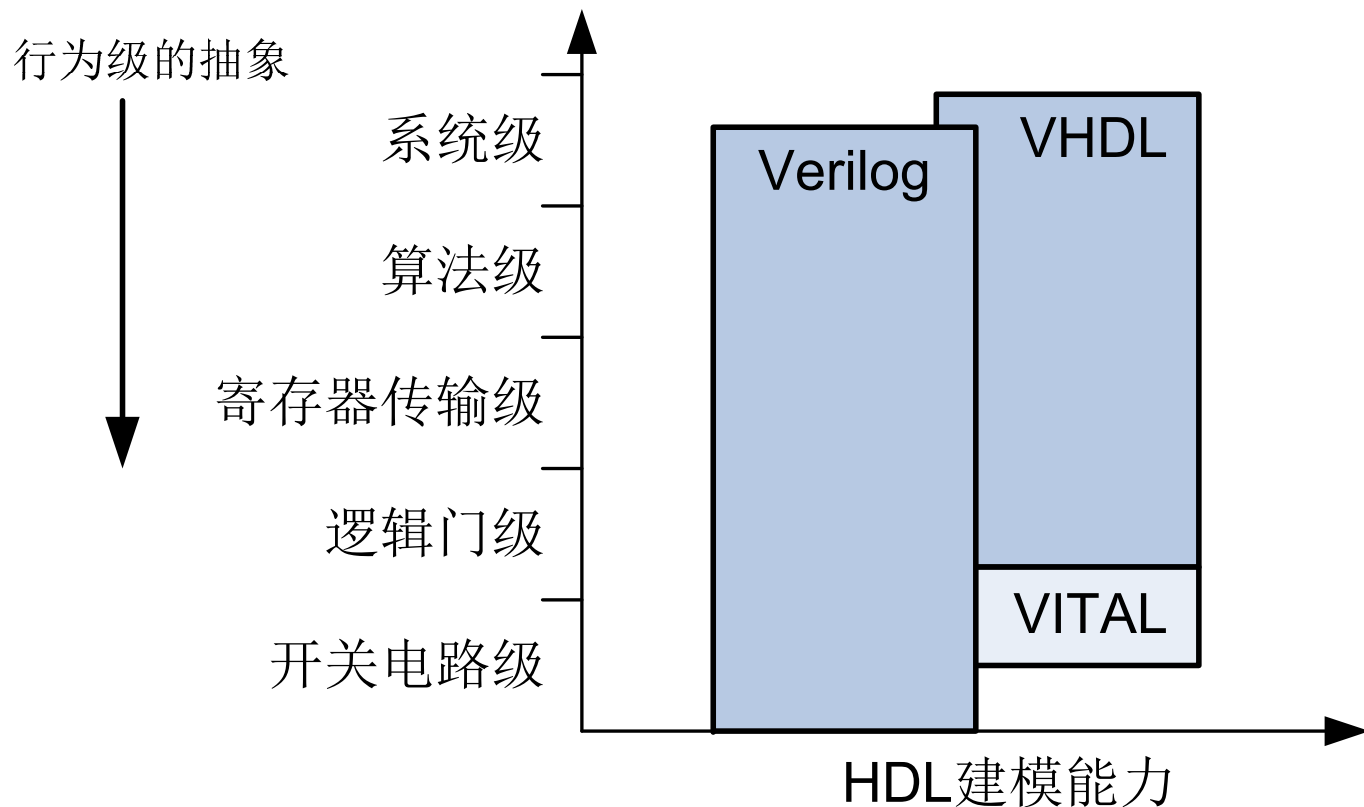
# Verilog HDL与VHDL共同之处

- 都成为国际标准：HDL最早于1962年由Iverson公司提出，到1980's已经出现百余种HDL，而最终成为IEEE标准的是VHDL(1987)和Verilog HDL(1995)。目前市场上所有EDA/EDSA工具都支持这两种语言。
- 能形式化的抽象表示电路的行为和结构；
- 支持逻辑设计中层次与范围的描述；
- 可借助高级语言的精巧结构来简化电路行为的描述；
- 具有电路仿真和验证机制以保证设计的正确性；
- 支持电路描述由高层到低层的综合转换；
- 硬件描述与实现工艺无关（工艺参数可通过语言提供的接口包含进去）；
- 便于文档管理，易于理解和设计重用。

# Verilog HDL与VHDL区别

- 从推出过程看，VHDL偏重标准化考虑，而Verilog与EDA工具结合更紧密：VHDL是应美国国防部VHSIC计划电子部件供应商统一数据交换格式要求推出的，而Verilog是在最大的EDA/ESDA工具供应商Cadence扶持下成长的；
- Verilog HDL代码风格更简洁明了、高效便捷，易于掌握。单从结构描述上看，二者代码长度之比3:1；
- Verilog早在1983年就推出，拥有更广泛的设计群体，美国、日本、台湾8:2，国内可能VHDL居多  
Verilog 20hour-3month, VHDL 6month专业培训
- 一般认为Verilog HDL在系统级抽象方面比VHDL略差，但在门级和开关电路级描述能力比VHDL强得多。但二者都在不断完善中。

# Verilog HDL与VHDL建模能力比较



- 仅供参考



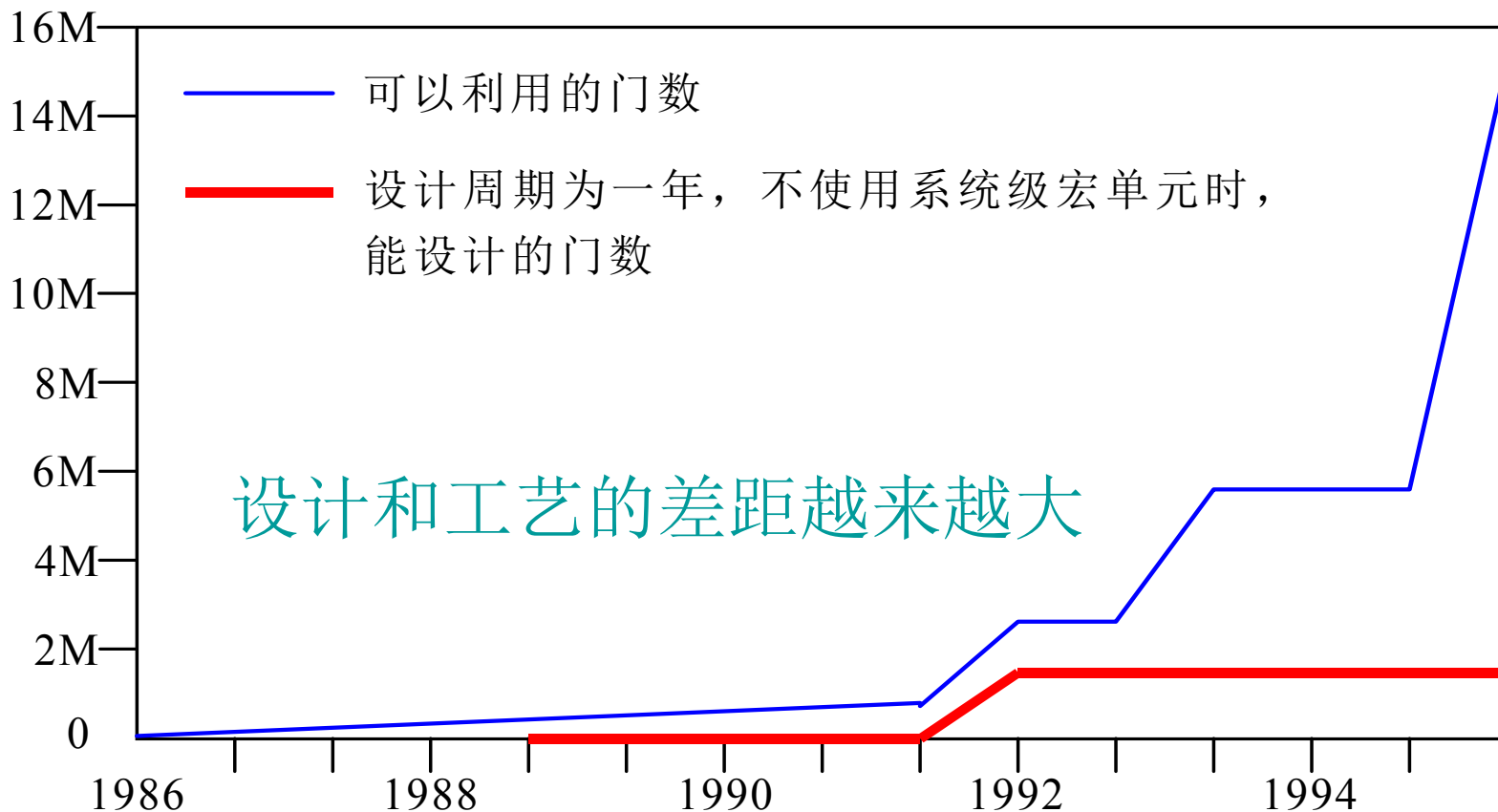
# Verilog HDL设计复杂数字电路的优点

- 传统原理图输入法的FPGA和ASIC设计工作往往只能采用厂家提供的专用电路图输入工具完成，且需要艰苦的手工布线。设计规模发展，要求的设计时间却越来越短，促使HDL在设计中应用；
- Verilog易学易用，语法与C语言相似；
- Verilog HDL允许在同一个电路模型内进行不同抽象层次的描述，如开关、门、RTL或行为级；仿真验证时其测试矢量也可用同一种语言描述；
- 设计的模块的信号位宽易于改变，修改后可适应不同规模应用；
- Verilog是国际标准，所有的制造厂商都提供综合后用于Verilog的逻辑仿真元件库，便于已有设计的移植和集成；

(续)

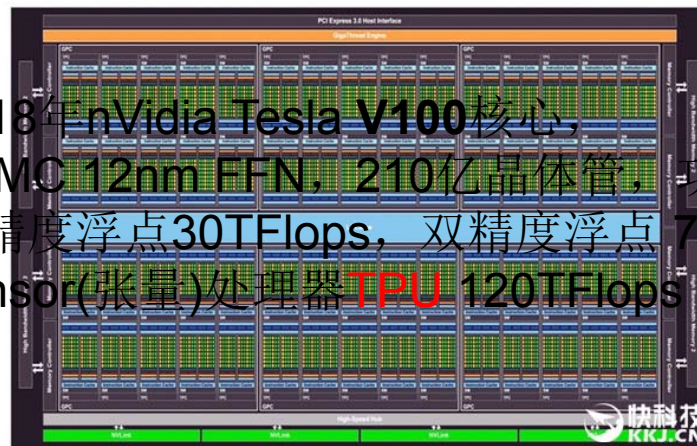
- Verilog综合器生成标准电子设计互换格式文件EDIF，独立于实现工艺；工艺无关性使工程师在功能设计、逻辑验证阶段不必过多考虑门级及工艺实现具体细节，实际是充分利用计算机的计算能力在EDA工具帮助下减轻人的繁琐劳动；
- 标准化的Verilog提高了设计好的软核的可重用性。
- 编程接口（PLI）是Verilog语言最重要特征之一，使得设计者可以通过自己编写的C代码访问Verilog内部的数据结构。设计者可以使用PLI按照自己的需要来配置Verilog HDL仿真器。

\* 经功能验证、可综合的、电路结构5000门以上的Verilog HDL模型称为“软核”(Soft Core)，由软核构成的器件称为虚拟器件。它们易于与外部逻辑结合在一起。



2016年nVidia PASCAL **GP100**核心，  
TSMC 16nm FinFET，153亿晶体管，  
610mm<sup>2</sup>  
双精度浮点 5.2TFlops，功耗300W  
此时Intel CPU约25亿晶体管

2018年nVidia Tesla **V100**核心，  
TSMC 12nm FFN，210亿晶体管，功耗300W  
半精度浮点30TFlops，双精度浮点7.5TFlops  
Tensor(张量)处理器**TPU** 120TFlops



# 硬件描述语言HDL的发展趋势

- 数字电路速度和复杂性正飞速增长，这要求设计者从更高的抽象层次描述电路，也即设计者只需从功能角度设计，**EDA**工具完成从设计到实现的复杂转换，并达到近似的优化效果；
- 目前主流的设计方法是**RTL**级设计，即从**RTL**描述生成网表；行为级综合工具允许直接对电路的算法和行为进行描述，**EDA**工具负责在各阶段进行转换和优化，这类方法已经被业界广泛应用；
- **SystemVerilog**在2005年成为国际标准，**IEEE1800-2005**
- **SystemC**在2006年成为国际标准，**IEEE1666-2005**。

## 二. 设计流程和通用设计方法论

### 2.1 完整工程通用设计方法

### 2.2 Verilog HDL设计方法

# 2.1 完整工程通用设计方法

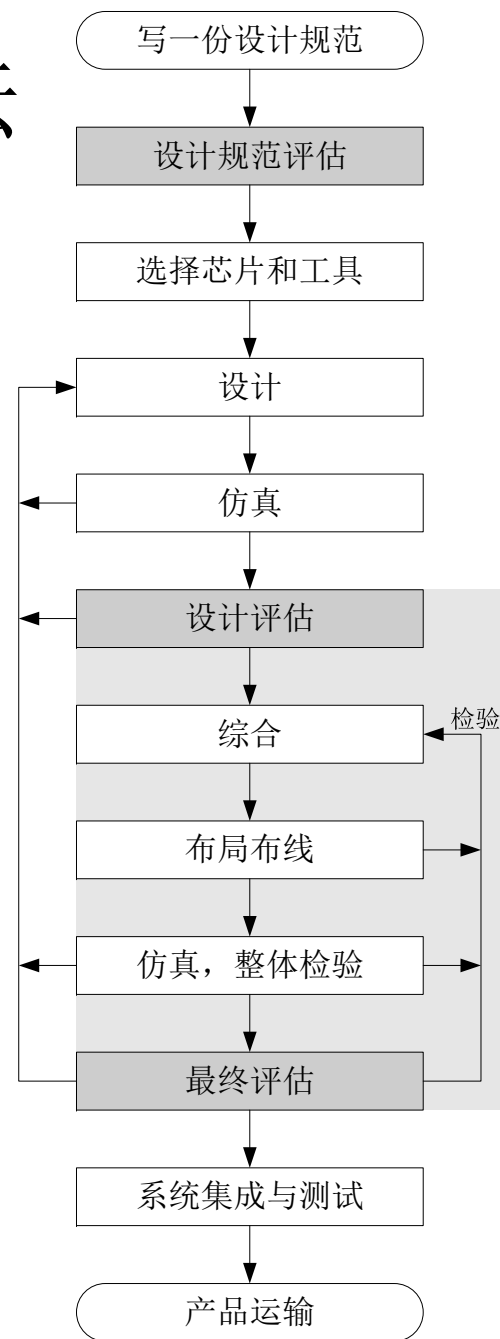
1. **设计规范**：可使工程师了解整个设计项目的情况及他自己在项目中承担的部分任务，有利于节省时间和设计成本，有助于避免对整个项目的错误理解。一份设计规范通常包含如下信息：

- 展示芯片与系统关系的外部框图
- 展示每一个功能部分的内部框图
- IO引脚的描述，包括：驱动能力，电平标准
- 定时估计：

对于输入引脚建立和保持时间、对于输出引脚的传递时间、时钟周期时间等

- 器件门数估计
- 封装形式
- 功耗目标
- 价格目标
- 测试程序（测试规划，测试软件）

\* 设计规范不是制定好就一成不变，在工程实施中可能根据实际情况变化被不断修改。



2. 设计规范评估：所有与项目设计相关的人员都应该参与设计规范评估，包括市场、销售、软件、应用人员，以期尽早发现错误或遗漏；
3. 选择器件和工具：选择最合适的器件，选择和了解综合工具，避免因软件问题而贻误工程；
4. 设计：使用自上而下的方法、按器件结构来工作，使用同步设计、防止亚稳定性的出现、避免悬浮节点、避免总线冲突；
5. 检验：图中灰色阴影的几个过程组成；注意发现未曾仿真的设计死角（代码覆盖），综合之后将门电路级仿真结果与RTL级仿真结果对照要相一致；布局布线完毕要进行定时分析和仿真；
6. 最终评估：形式上的确认本设计最终完成；
7. 系统集成与测试：芯片投产后应该进行长时间的连续强化试验，以确定器件和系统的可靠性。

# 通用设计方法期望达到的目标：

- 设计一个器件：
  - 没有具体制造方面的缺陷；
  - 在整个器件寿命过程中工作可靠；
  - 在系统中所起作用是正确的；
- 有效地设计这种器件：
  - 用最短的时间；
  - 使用最少的资源，包括人力资源；
- 有效的规划设计：
  - 在设计过程中尽早的制定出一个合理的时间表；
  - 预先了解所需的资源状况，并在设计过程中尽早的分配他们；



## 2.2 Verilog HDL设计方法

Verilog HDL支持如下设计方法:

- 自下而上 Bottom-Up
- 自上而下 Top-Down
- 两种方法综合

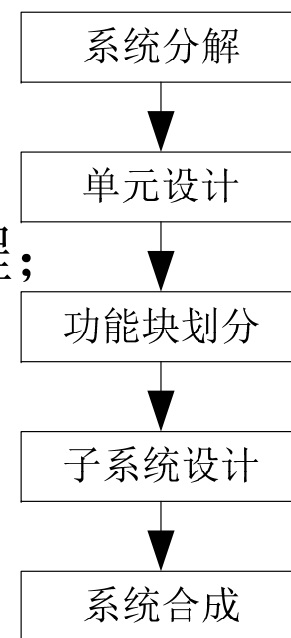
# 自下而上 Bottom-Up设计方法

设计过程：

- 根据系统要求编制技术规格书，画出系统控制流程图
- 对系统功能进行细化，划分功能模块，画出系统框图
- 功能模块细化和电路设计
- 功能模块整合和全系统调试

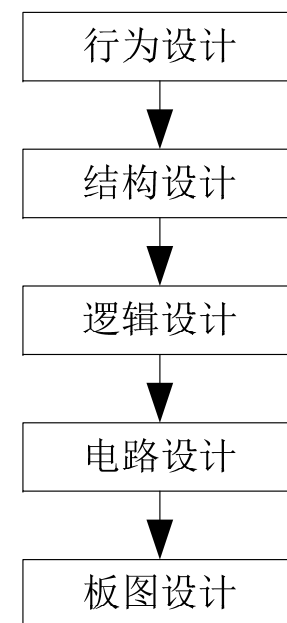
优缺点：

- 优点：是以往采用的方法，设计人员熟悉该过程；实现各子块所需时间短；
- 缺点：容易发生对系统整体功能把握不足情况；先完成小模块，整个系统实现需要较长时间；对设计人员互相协作有较高要求；若结构设计的错误容易造成较大损失。

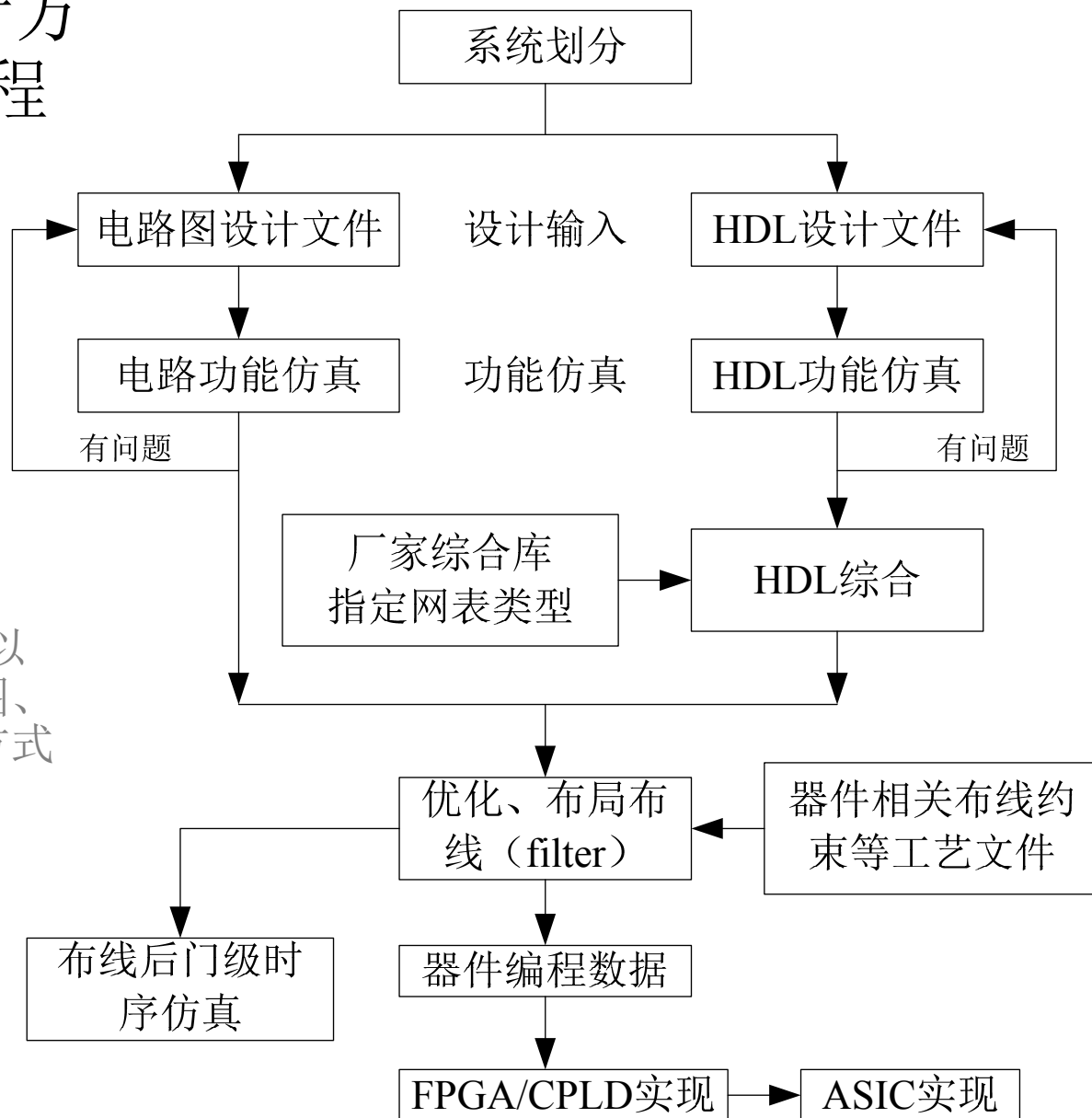


# 自上而下Top-Down设计方法

- 设计步骤：
  1. 对整个系统进行方案设计和高层次的功能划分；
  2. 用低层次的具体实现去充实高层次的功能；
  3. 一个设计可视为分等级的树形结构，最末枝的单元是已存在的、已经制造的、已经开发好的、可外购得到的IP等；
- 优点：
  1. 在设计周期开始就做好了系统分析；
  2. 主要仿真和调试过程在高层次完成，此时与工艺无关，能在早期发现结构设计上的错误，避免设计工作浪费，也减少逻辑仿真的工作量；
  3. 方便了从系统级划分和管理整个项目，使得百万门以上复杂数字电路设计成为可能，并可减少设计人员，避免不必要的重复设计，提高设计的一次成功率；
- 缺点：
  1. 采用的综合工具不同，得到的最小单元不标准；
  2. 制造成本可能很高（利用外购IP等）；



# 高层次设计方法典型流程

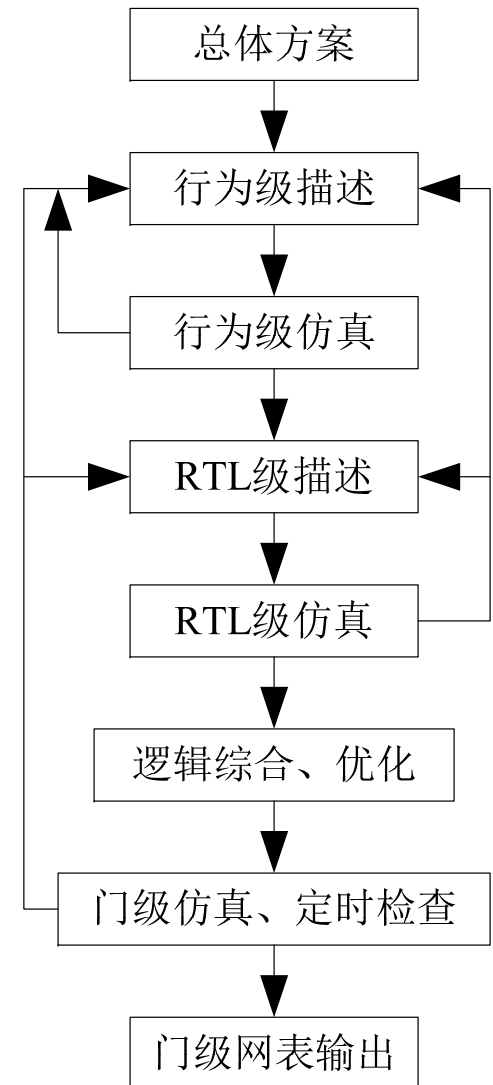


\*输入方法还可以有框图、状态图、波形图等图形方式

# Verilog HDL设计流程

通常分为3个层次进行

- 行为描述：设计系统的初始阶段，通过对系统行为仿真发现功能设计中存在的问题。完成行为描述之后，还要将其转换为RTL级描述，因为现有EDA工具大多只能支持RTL级描述的HDL文件的自动逻辑综合。
- RTL：Register Transfer Level描述，行为描述方式过于抽象，RTL级的行为描述才开始与硬件有关
- 逻辑综合：用逻辑综合工具，将RTL级的程序转换成用基本逻辑元件表示的文件（门级网表），综合结果也可用原理图方式输出。网表需要进行门级仿真和定时检查，无问题才能结束前端设计。



# 行为级HDL代码示例

```
// Multiplier for two unsigned numbers  
always @ (posedge multiply_en)  
begin  
    product <= a*b;  
end
```

# RTL级HDL代码示例

// Multiplier for two unsigned 4-bit numbers:

input [4:1]a, b; output [4:1] count, product; reg [4:1] count, product;

always @ (posedge clk)

begin

if (multiply\_en == 1) begin

count <= 4;

product <= 0;

end

if (count) begin

if (b[count]) begin //b[count]==1

product <= (product << 1) + a;

end

else begin //b[count]==0

product <= product << 1;

end

count <= count - 1;

end

end

# 门电路级HDL代码示例

// Multiplier for two unsigned 4-bit numbers

```
module UnsignedMultiply(clk, a, b, multiply_en, product);
```

```
    input clk, multiply_en;
```

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

```
    output [7:0] product;
```

```
    wire clk;
```

```
    wire un1_count5_axb_1;
```

.....//此处省略40余行

```
LUT2_6 un1_count_5_axb_1_Z(
```

```
    .I0(count[1]),
```

```
    .I1(count16),
```

```
    .O(un1_count5_axb_1));
```

.....//此处省略类似模块实例50余个，近300行

```
endmodule
```



### 三. Verilog语法的基本概念

- Verilog HDL是一种用于数字系统设计的语言，用Verilog HDL描述的电路设计就是该电路的Verilog HDL模型，也称为模块（Module ['mɒdju:l / 'mɒdʒul]）。
- 一个复杂电路的完整Verilog HDL模型由若干个Verilog HDL模块构成，每个模块又可由若干个子模块构成。有些需综合成具体电路，有些只是用于提供测试向量等用途。
- Verilog模型可以是实际电路不同级别的抽象：
  - (1)系统级(System-level)——用语言提供的高级结构能够实现所设计模块的外部性能模型；
  - (2)算法级(algorithm-level)——用语言提供的高级结构能够实现算法运行的模型；
  - (3)RTL级(register transfer level)——描述数据在寄存器之间的流动和如何处理、控制这些数据流动；
  - (4)门级(gate-level)——描述逻辑门以及逻辑门之间连接的模型
  - (5)开关级(switch-level)——描述器件中三极管和存储节点及它们之间连接的模型。

(1)(2)(3)属于行为描述，从(3)开始才与逻辑电路有明确对应关系。前端设计人员需要掌握(1)~(4)，后端设计人员还需要掌握(5)。

# 行为描述语言的功能

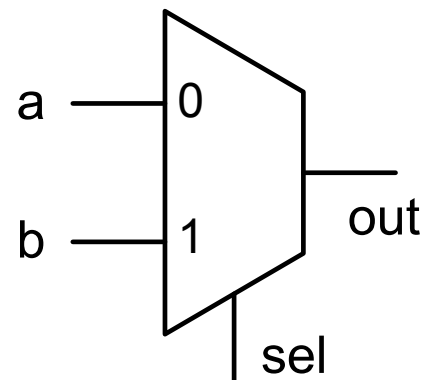
**Verilog HDL**行为描述语言作为一种结构化和过程性的语言，语法结构非常适合于算法级和**RTL**级模型设计。这种行为描述语言具有以下功能：

- 可描述顺序执行或并行执行的程序结构；
- 用延迟表达式或事件表达式来明确控制过程的启动时间；
- 通过命名的事件触发其他过程里的激活行为或停止行为；
- 提供了条件如**if-else**、**case**等循环程序结构；
- 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结构(**function**)；
- 提供了用于建立表达式的算术运算符、逻辑运算符、位运算符
- 结构化语言也适合于门级和开关级模型设计；

# 模块的基本概念

- 例3.1

```
module mux2(out, a, b, sel);  
    input a, b, sel;  
    output out;  
    reg out;  
    always @(a or b or sel)  
        if (!sel) out = a;  
        else out = b;  
endmodule
```

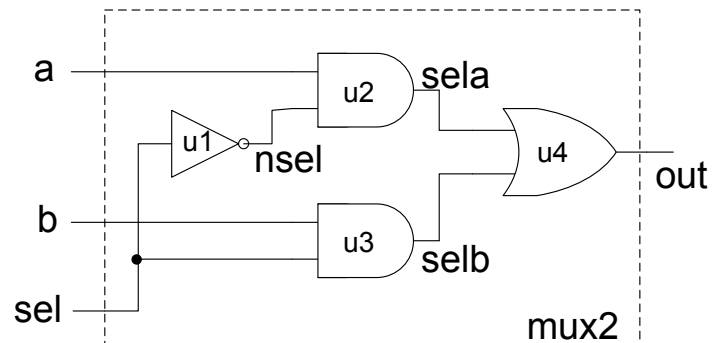


行为级描述

若编写Verilog模块时遵从一些基本规则（可综合的规则），就可以把例3.1行为描述转换为例3.2的门级描述模块，这个过程称为综合/合成（Synthesis）；

## 例3.2

```
module mux2(out, a, b, sel);  
    input a, b, sel;  
    output out;  
    wire nsel, sela, selb;  
    not u1(nsel, sel);  
    and u2(sela, a, nsel);  
    and u3(selb, b, sel);  
    or u4(out, sela, selb);  
endmodule
```



门级描述

not、and、or是组合逻辑基本元件原语（Primitive），是Verilog HDL保留字；  
例子中 #1 表示门输入到输出延迟为1个仿真时间单位；

### 例3.3

```
module adder4(cout, sum, a, b, cin);  
    input [3:0] a, b;  
    input cin;  
    output cout;  
    output [3:0] sum;  
    assign {cout, sum} = a+b+cin;  
endmodule
```

4位的全加器，根据两个4bit的加数a、b和进位输入cin计算和sum和进位输出cout。

assign是连续赋值语句，属于行为描述方式，后面会讲到；可以看到Verilog HDL的全部程序是位于module和endmodule声明语句之间的。

## 例3.4

```
module compare2(equal, a, b);  
    input [1:0] a, b;    //声明输入信号a、b  
    output equal;        //声明输出信号equal  
    assign equal = (a==b) ? 1 : 0; //条件运算符  
    /*若a、b相等equal输出高电平，否则低电平*/  
endmodule
```

- 2位的比较器，对两个2bit的数a、b进行比较，若a、b相等，输出高电平，否则输出低电平。
- //和/\*...\*/都是注释符号，分别用来注释一行和一段，被注释掉部分不被编译（但编译器可能会检查）。
- 可见Verilog HDL的很多语法和C语言非常相似。

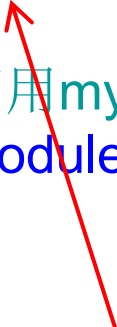
## 例3.5

```
module trist1(out, in, en);  
    input in, en;  
    output out;  
    bufif1 mybuf(out, in, en);  
endmodule
```

- 例3.5描述了一个名为trist1的三态驱动器，通过调用Verilog语言提供的原语库中三态驱动器元件bufif1实现。库元件的实现是（被例化为）mybuf。
- 例3.6中存在着两个module，首先用行为描述方式描述了一个三态驱动器模型mytri，然后在另一个模块trist2中调用了mytri的实例部件tri\_inst，带“.”的表示被引用模块的端口，名称必须一致，“()”中表示本模块中与之相连的端口。

## 例3.6

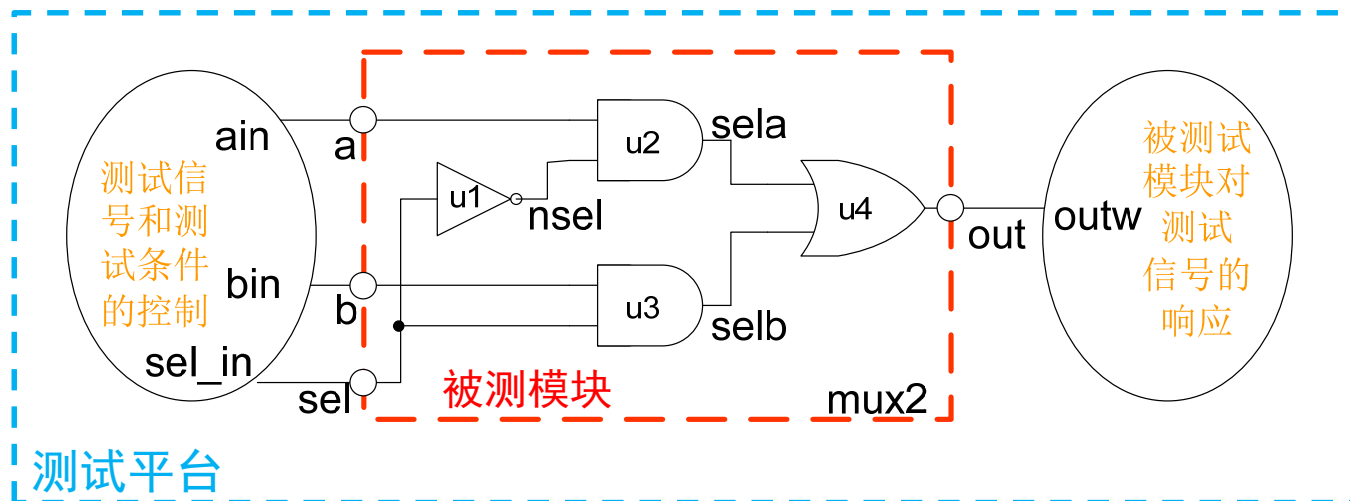
```
module trist2(sout, sin, ena);  
    input  sin, ena;  
    output sout;  
    mytri tri_inst(.out(sout),  
                   .in(sin), .en(ena));  
    //调用mytri模块定义的实例tri_inst  
endmodule
```



```
module mytri(out, in, en);  
    input  in, en;  
    output out;  
    assign out = en ? in : 1'bz;  
    //1'bz表示1bit的数值(高阻z)  
endmodule
```

\* HDL中对模块的调用/实例化表示硬件电路的建立(一个拷贝)

# Verilog用于模块的测试



- Verilog语言可以用来描述变化的测试信号，描述测试信号的变化和测试过程的模块叫做**测试平台**（**Test bench**或**Test fixture**），可对电路模块（无论行为的或结构的）进行动态的全面测试。
- 通过观测被测试模块的输出信号是否符合要求，可以调试和验证逻辑系统的设计和结构正确与否，可以发现问题并及时修改。如上图所示。



## 例3.7 对例3.1和3.2的mux2测试

```
`include "mux2.v" //Verilog HDL默认扩展名.v

module tstmux;
reg ain, bin, sel_in;
reg clock;
wire outw;

initial //过程描述语句
begin //初始化寄存器变量
    ain = 0;
    bin = 0;
    sel_in = 0;
    clock = 0;
end

always #50 clock = ~clock;
//产生一个不断重复的、
//周期为100个单位时间的
//时钟信号clock

always @(posedge clock)
begin
    ain = {$random} %2; //$系统函数
    #3 bin = $random % 2; //延时3个单位
end

always #10000 sel_in = ! sel_in;
//产生周期10000单位时间的选通信号变化
mux2
m(.out(outw), .a(ain), .b(bin), .sel(sel_in));
//实例引用mux2模块(m是mux2的实例),
//加入测试信号流, 以观察模块输出out.
endmodule

//在功能(即行为)级上进行的测试称前仿真
/*在门级上进行的测试称门级仿真*/
/*若再加入工艺技术及布局布线延迟模型
进行的仿真就称为布线后仿真*/
```

# Verilog HDL语法基本概念小结

- (1) Verilog HDL程序由模块构成，每个模块实现特定功能，其内容位于 `module` 和 `endmodule` 之间；
- (2) Verilog HDL模块基本可以分成两种类型：一种是为了最终生成实际的电路结构，另一种只是为了测试某部分电路的逻辑功能是否正确；
- (3) 模块可进行层次嵌套(调用)，所以大型数字电路设计才可以分割成不同的小模块来进行；
- (4) 如果每个模块都可综合，则用综合工具可以把它们的功能描述都转换为逻辑单元描述，最后可以用一个上层模块通过实例引用把这些模块连接起来，把它们整合成一个很大的逻辑系统；
- (5) 每个模块都要进行端口定义，并说明是输入口还是输出口，然后再对模块的功能进行描述；
- (6) Verilog HDL程序的书写格式自由，一行可以写多个语句，一个语句也可以分写多行（和C语言类似）；
- (7) 除了 `endmodule`、`(begin-) end` 和 `(fork-) join` 等成对的结构定义语句外，每个语句和数据定义的最后必须有分号 “;”
- (8) 可以用 `//...` 和 `/*...*/` 对Verilog HDL程序的任何部分作注释，一个友好的、有使用价值的源程序都应当加上必要的注释，以增强程序的可读性和可维护性。

# 思考题

1. Verilog语言有何作用？构成模块的关键词是什么？
2. 为什么说用Verilog HDL可以构成非常复杂的电路结构？
3. 为什么可以用比较抽象的描述来设计具体的电路结构？
4. 是否所有抽象的模块都可以通过综合工具转为电路结构？
5. 什么是综合？综合由什么工具来完成的？
6. 通过综合产生的是什么？产生的结果有什么用处？
7. 仿真是什么？为何进行仿真？
8. 仿真可以在几个层次上进行？每个层次的仿真有何意义？
9. 模块的端口是如何描述的？
10. 引用实例模块的时候，如何在主模块中连接信号线？
11. 如何产生连续的周期的测试时钟？
12. 如果不用initial块，是否可以产生测试时钟？
13. 从例程是否可明白initial与always两种过程块有何不同？

# 四. Verilog常用语法

## 一模块、数据类型、表达式

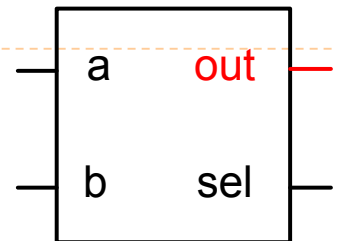
- 模块的结构
- 数据类型及常量和变量
- 小结
- 思考题

## 4.1 模块的结构

```
module blk(a,b,sel, out);  
    input a, b, sel;  
    output out;  
    //parameter addr=0;  
    reg out;  
    always @(a or b or sel)  
        if (!sel) out = a;  
        else out = b;  
endmodule
```

```
module 模块名(端口列表);  
    端口定义input, output,  
        inout;  
    [参数定义(可选);]  
    [数据类型定义;]  
    行为描述;  
endmodule
```

上面Verilog HDL设计中，模块第2、3行说明接口信号流向，第4、5行说明模块逻辑功能；全部Verilog结构位于module和endmodule声明语句之间。



可见Verilog程序包括4个主要部分：端口定义、IO说明、内部信号声明和功能定义。

# 模块的端口定义

模块的端口列表声明了模块的输入输出口，格式如下：

**module** 模块名 ( 端口1, 端口2, 端口3, 端口4, ...端口n );

例如: **module mytri (out, in, en);**

- 端口列表里是模块所有外部端口，端口方向由后续的IO口说明语句决定；
- 模块被引用时，其端口可以用两种方法连接：

(1) 引用时用”.” 标明原模块定义时规定的端口名：

如前面的: **mytri tri\_inst(.out(sout), .in(sin), .en(ena));**

信号名与被引用模块端口名一一对应，顺序可交换，提高了程序的可读性和可移植性。

(2) 引用时严格按模块端口定义顺序连接，不需标端口名：

模块名(连接端口1信号名, 连接端口2信号名, ...);

如: **mytri tri\_inst (sout, sin, ena);**

语法简洁，但要求端口连接顺序必需与被引用的模块定义时的端口列表顺序严格一致。

# 模块内容—IO说明格式

- 输入口：  
input [信号位宽-1:0] 端口名1 ;  
input [信号位宽-1:0] 端口名2 ;  
...  
input [信号位宽-1:0] 端口名i ;
- 输出口：  
output [信号位宽-1:0] 端口名1 ;  
output [信号位宽-1:0] 端口名2 ;  
...  
output [信号位宽-1:0] 端口名j ;
- 输入/输出口：  
inout [信号位宽-1:0] 端口名1 ;  
inout [信号位宽-1:0] 端口名2 ;  
...  
inout [信号位宽-1:0] 端口名k ;

例：

- 输入口：  
input [23:0] add\_bus24 ;  
input [1:15] add\_bus16 ;  
input a, b, c; // 1 bit
- 输出口：  
output [8:1] ctl\_bus8 ;  
output [15:0] ctl\_bus16 ;  
...
- 输入/输出口：  
inout [15:0] data\_bus1,dbus2 ;  
//一次定义两个16bit总线

\* I/O说明也可以写在端口列表里（?将被限制为wire型变量），如下：

```
module name (input iport1, input iport2, ..., output oport1, output oport2, ... );
```

# 模块内容一内部信号说明

- 模块内部用到的与端口有关的**reg**和**wire**类型变量的声明，如：

**reg** [位宽-1:0] R1, R2, ...;

**wire** [位宽: 1] w1, w2, ...;



# 模块内容—功能定义

- 模块的核心部分，有3种方法产生组合逻辑：

## 1) 用实例元件

如： **and** u1 (q, a, b);

与电路图输入方式下调用元件库一样，实例名(u1)必须唯一。

## 2) 用连续赋值语句assign

如： **assign** q = a & b;

语法简单，assign后面接表达式即可，是描述组合逻辑最常用的方法之一。

## 3) 用行为描述语句always

如： **always @ (a or b)**      **//always @(\*)**

**begin**

**q = a & b;**

**end**

**always**语句既可描述组合逻辑也可描述时序逻辑。

# 模块内容一理解要点

- Verilog中所有过程块（如**initial**、**always**）、连续赋值语句、实例引用都是并行执行的；而在**begin...end**块内的语句是顺序执行的，**if...else**也必须顺序执行，否则无意义；
- 它们表示的是一种通过变量名互相连接的关系；
- 在同一模块中三者出现的先后顺序没有关系（它们是并行执行的）；
- 只有连续赋值语句**assign**和实例引用语句可以独立于过程块而存在于模块的功能定义部分。

以上4点与C语言有很大不同，许多类似C语言的语句只能出现于过程块（即**always**、**initial**）中，而不能随意出现在模块功能定义范围内。

## 4.2 数据类型及常量和变量

- Verilog HDL中，数据类型用来表示数字电路硬件中的数据储存和传送元素。
- 4个基本类型：**reg**、**wire(=tri)**、**integer**、**parameter**
- 其它的：**large**、**medium**、**small**、**scalared**、**time**、**real**、**realtime**、**tri0**、**tr1**、**triand**、**trior**、**triereg**、**vectored**、**wand**、**wor**、**supply0**、**supply1**
- 除**time**、**realtime**，都与基本逻辑单元建库有关，但不是所有的类型都可以被自动综合软件综合成实际电路。
- 可以分成两大类：常量、变量

## 4.2.1. 常量—数值1

### 1. 整数

1) 二进制整数(b或B)

2) 十进制整数(d或D)

3) 十六进制整数(h或H)

4) 八进制整数(o或O)

表达方式有以下3种:

1) <位宽><进制><数字> //是完整的描述方式

8'b10101111

8位宽的二进制表示

8'ha2

8位宽的十六进制表示

5'd3

5位宽的十进制

2) <进制><数字>

//位宽缺省，至少32位

'hcc700

缺省位宽的十六进制数

'o5270

缺省位宽的八进制数

3) <数字>

//缺省的是十进制

567

缺省位宽的十进制数

4ff

非法的整数表示（应为'h4ff）

\* 位宽指明数字精确位数，指二进制位

## 4.2.1. 常量—数值2

### 2. x和z

逻辑值取值可为下面4种之一：

- |                  |               |
|------------------|---------------|
| 1) 0: 逻辑0或假状态    | 2) 1: 逻辑1或真状态 |
| 3) x: 未知状态或不关心状态 | 4) z/?: 高阻态   |

x和z可以用来定义十六进制数的4位二进制数状态、八进制的3位二进制数状态、二进制数的1位状态；**x不能表示十进制数的数字**。例如：

4'b10x0	//位宽为4的二进制数，LSB第2位为不定值
4'b101z	//位宽为4的二进制数，LSB为高阻值
12'dz	//位宽为12的十进制数，值为高阻值(方法1)
12'd?	//位宽为12的十进制数，值为高阻值(方法2)
8'h4x	//位宽为8的十六进制数，LSB4位为不定值
'bx	//缺省位宽的二进制数，不定值
8'dx是正确的，但12'd5x错误，x不能表示十进制数码	

## 4.2.1. 常量—数值3

### 3. 负数

被定义为负数的数值只需在位宽表达式最前面加负号“-”，这个负号不可以放在别的地方。例如：

-8'd50                   //8位二进制数补码表示的-50

8'd-50                   //非法的负数表示法

### 4. 实数

Verilog HDL中的实数可以用十进制和科学计数法两种格式表示，若采用十进制表示，小数点两边必须都有数字，否则为非法表示。实数不能被自动综合。如：

15.2                    //十进制表示的实数

1.01e-5                //科学计数法表示的 $1.01 \times 10^{-5}$

90.或.101e6           //非法的实数表示

### 5. 下划线 “\_” (underscore)

用于分割数的表达以提高可读性，只能用于具体数字之间，但不能放在首位或用于位宽、进制表达处。例如：

16'b1010\_1011\_1101\_0001                   //合法格式 分为4个nibble

8'b\_0011\_1100                   //非法格式

## 4.2.2 常量—参数([pə'ræmitə]parameter)型

- Verilog HDL中用parameter定义一个标识符来代表常量，可提高程序的可读性及可维护性。parameter型数据是常数型数据，有别于宏定义“`#define ABC 15`”，宏定义是直接字符替换。格式如下：

**parameter 参数名1=表达式, 参数名2=表达式, ..., 参数名n=表达式;**

赋值语句中的常数表达式只能包含数字或此前定义过的参数：

```
parameter msb = 7;  
parameter r = 5.7, f = 3.0;  
parameter byte_size = 8, byte_msb = byte_size - 1;  
parameter average_delay = (r + f) / 2;
```

参数型常量经常用于定义延迟时间和变量宽度等。在模块或实例被引用时，可通过参数传递方法，改变在被引用模块或实例中已经定义的参数值。见下面的例子：

# 参数型示例

```
module decoder(a, f);  
    parameter WIDTH = 1, POLARITY = 1;  
    ...  
endmodule
```

```
module top;  
    wire [3:0] A4;    wire [4:0] A5;  
    wire [15:0] F16;  wire [31:0] F32;  
    decorder #(4, 0) d1(A4, F16);  
    decorder #(5)    d2(A5, F32);  
endmodule
```

上例中通过#(4,0)传递参数，实际调用的是参数“WIDTH=4, POLARITY=0”的模块d1；  
通过#(5,1)传递参数，实际调用的是参数“WIDTH=5, POLARITY=1”的模块d2



## 4.2.3 常量一字符串

- 字符串是用半角引号” ”扩起来的字符序列，不能分成多行书写，所有字符必须包含到同一行中（与C语言要求相同）。
- 若字符串用作Verilog HDL表达式或赋值语句中的操作数，会被作为8位的ASCII值序列处理，每个字符对应其8位ASCII值。
- 例如：“this is a string”、“print out a message\n”、“bell\007”等

转义字符：

字符串变量声明

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

```
initial
```

```
begin
```

```
    stringvar = "Hello world!"
```

```
end
```

字符串操作

字符串可以进行连接等操作

```
stringvar = {stringvar, "!!!"};
```

转义字符	含义
\n	换行符
\t	Tab符
\\	字符 \
\*	字符 *
\ooo	3位八进制
%%	字符 %

## 4.3 变量

- 变量即在程序运行过程中其值可以改变的量，Verilog 中有很多种变量，这里只介绍常用的几种。

网络型： `wire`， `tri`

寄存器型： `reg`， `integer`

存储器型： `memory`（是 `reg` 型数组）

## 4.3.1 变量—wire型

- Verilog中的连线型数据及其功能描述

连线型数据	功能描述
wire, tri	常见的连线(网线)类型
wor, tiror	多重驱动时具有线或特性
wand, triand	多重驱动时具有线与特性
triereg	具有电荷保持特性的连线型
tri1	上拉电阻
tri0	下拉电阻
supply1	电源线, 逻辑1
supply0	电源线, 逻辑0

# wire / tri型的真值表—驱动强度一致时

wire/tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

- **wire**型数据表示结构实体(例如门)之间的物理连接, 不能存储数值, 且必须受到驱动器(门或**assign**连续赋值语句)的持续驱动才能保持数值, 若没有驱动器连接到**wire**型变量, 该变量为高阻**z**。
- **wire**与**tri**都用于连接器件单元, 具有相同的语法格式和功能, 两种名称表达相同的概念是为了与模型中使用变量的情形相一致, 也即:  
    **wire**型通常用于表示单个门或**assign**驱动的连线型数据  
    **tri**则用来表示多驱动器驱动的连线型数据。
- 若**wire**与**tri**型变量没有定义逻辑强度(**logic strength**), 多驱动源情况下会发生冲突, 从而产生不确定值**x**。假设两个驱动源强度一致, 则多驱动的真值表如上表所示。

# wire型变量的语法

- Verilog HDL程序模块输入输出信号类型默认为wire型。wire型信号可以用作任何表达式的输入，也可以作为assign或实例元件的输出。定义wire型变量语法如下：

**wire** [n-1:0] 数据名1,数据名2,数据名3, ..., 数据名i;

**wire** [n:1] 数据名1,数据名2,数据名3, ..., 数据名i;

//都可以定义i条总线，每条总线内有n条线(wire)

- 举例：

**wire** a; //定义一个1bit的wire型数据a

**wire** [7:0] b; //定义一个8bit的wire型数据b

**wire** [4:1] c, d; //同时定义两个4bit的wire型数据c、d

## 4.3.2 变量—寄存器reg型

- 寄存器**register**是数据存储单元的抽象，定义关键字**reg**。通过赋值语句可以改变寄存器存储的值，其作用与改变触发器存储的值相当。**reg类型数据默认初始值为不定值x**
- **reg**型数据可以赋正值也可以赋负值，但当**reg**用于一个表达式的操作数时，将被当作无符号数，即正值；
- **reg**类型的数据并不一定被综合成寄存器
- 在**always**块内被赋值的每一个信号都必须定义成**reg**型
- 定义格式如下，和**wire**类似：

**reg** [n-1:0] 数据名1,数据名2,数据名3, ..., 数据名i;

**reg** [n:1] 数据名1,数据名2,数据名3, ..., 数据名i;

//都可以定义i个**reg**型数据，每个**reg**型数据有n个bit，举例：

**reg** rega; //定义一个1bit的名为rega的**reg**型数据

**reg** [3:0] regb; //定义一个4bit的**reg**型数据regb

**reg** [4:1] regc, regd; //同时定义两个4bit的**reg**型数据

## 4.3.3 变量—存储器memory型

- 通过reg型变量建立数组来对memory建模，可以描述RAM、ROM和reg文件
- 数组中一个单元通过一个数组索引进行寻址。Verilog中没有多维数组，memory型数据是通过扩展reg型数据的地址范围来生成的。定义格式如下：

`reg [n-1:0] mem_name [m-1:0];`

或者 `reg [n-1:0] mem_name [m:1];`

其中`reg [n-1:0]`定义了每个存储单元的大小，即位数n；存储器名后的`[m-1:0]`则定义了该存储器中有多少这样的寄存器。

例如：`reg [7:0] mema [255:0];`

定义了一个名为mema的存储器，该存储器有256个8位的存储器单元，其地址范围是0到255。

**\* 注意：**对存储器进行地址索引的表达式必须是常数表达式

# memory.2

- 在同一个数据类型声明语句中可以同时定义存储器和reg型数据，例如下面：

```
parameter WORDSIZE = 16, MEMSIZE = 256;  
reg [WORDSIZE-1:0] mem [MEMSIZE-1:0], writereg, readreg;
```

- 尽管memory与reg定义格式类似，但需注意一个由n个1位reg构成的存储器组不同于一个n位的寄存器！例如：

```
reg [n-1:0] rega;    //一个n位的寄存器  
reg mema [n-1:0];    //n个1位的reg构成的memory组
```

如下赋值：  
    rega = 0;      //合法赋值  
    mema = 0;      //非法赋值

若需要对memory中的1bit存储单元读写操作，必须指定该单元在memory中的地址，如下方法合法：

```
mema[3] = 0; //给memory中第3个存储单元赋值
```



# 关于字节编址Little-Endian和Big-Endian

32bit数据0x87654321

在内存中存储格式:

```
p = 0x0;
for (i=0; i<256; i++) {
    *fifo = *p++;
}
```

字节地址	存储内容
0004H	.....
0003H	0x87
0002H	0x65
0001H	0x43
0000H	0x21

Little-Endian

字节地址	存储内容
0004H	.....
0003H	0x21
0002H	0x43
0001H	0x65
0000H	0x87

Big-Endian

Little-Endian:

n	n+1	n+2	n+3	n+4	.....
.....	0x21	0x43	0x65	0x87	.....

Big-Endian:

n	n+1	n+2	n+3	n+4	.....
.....	0x87	0x65	0x43	0x21	.....

## 4.4 小结

1. Verilog HDL模块中所有的过程块(如initial、always块), 连续赋值语句、实例引用都是并行的
2. 它们表示的是一种通过变量名互相连接的关系
3. 在同一模块中各个过程块、各条连续赋值语句和各条实例引用语句这三者出现的先后顺序没有关系
4. 只有连续赋值语句(assign)和实例引用语句(已定义模块名实例化实体instance的语句)可以独立于过程块而存在于模块的功能定义部分
5. 被实例引用的模块, 其端口可以通过不同名的连线或寄存器类型变量连接到别的模块相应的输出输入信号端
6. always块内被赋值的每个信号都必须被定义成reg型, 但并不一定被综合成寄存器

以上与C语言有很大不同, 许多C语言类似语句只能出现在过程块中, 而不能随意出现在模块功能定义范围内。

# 五 Verilog HDL运算符

Verilog HDL运算符汇总表

运算符类型	运算符	说明
算术运算符	+, -, *, /, %	加, 减, 乘, 除, 取模
赋值运算符	=, <=	阻塞, 非阻塞
关系运算符	>, <, >=, <=	大于, 小于, 大于等于, 小于等于
相等运算符	==, !=, ===, !==	相等, 不等, 全等, 非全等
逻辑运算符	&&,   , !	逻辑与, 逻辑或, 逻辑非
条件运算符	? :	条件(结果二选一)
位运算符	&,  , ~, ^, ^~或~^	位操作: 与, 或, 非, 异或, 同或
移位运算符	<<, >>	左移, 右移
拼接运算符	{ }	连接
归约运算符	&, ~&,  , ~ , ^, ~^或^~	归约: 与, 与非, 或, 或非, 异或, 同或

注: 表中红色运算符是Verilog HDL比C语言新增的

## 5.1 算术运算符 (+, -, \*, /, %)

- 操作数为wire或reg则是无符号数
- 对于整型和实型变量，可以是有符号数
- 若两个操作数有一个含有x，则结果未知

```
module ArithTest;
```

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

```
initial begin
```

```
    a=4'b1100; /*a=12*/    b=4'b0011; /*b=3*/    c=4'b1011; /*b=11*/
```

```
    $display(a*b);          //结果4 (10 0100, 按操作数最长位数截短)
```

```
    $display(a/b);          //4
```

```
    $display(a+b);          //15
```

```
    $display(a+c);          //7 (1_0111截短到4位)
```

```
    $display(a-b);          //9
```

```
    $display((a+1'b1)%b);    //1
```

```
    $display(-10%3);         //-1
```

```
    $display(11%-3);         //2
```

```
end
```

```
endmodule
```

## 5.2 位运算符 (&, |, ~, ^, ^~或~^)

- 取非运算符~是单目运算符
- 异或运算符^的规则：相同为0，不同为1，含有x时，结果也为x
- 同或运算符~^与异或运算符^的结果相反
- 长度不同操作数做位运算，系统会右端对齐，并给位数少的数高位补0
- 不要将位运算符和逻辑运算符混淆：比如对于取非操作，令a=4'b1100，其逻辑值为1；则逻辑非!a为0，而按位非~a为4'b0011，仍为逻辑1。

例如：module BitTest;

```
reg [3:0] a, b, c, d, e;
```

```
initial begin
```

```
    a=4'b1100; b=4'b0011; c=4'b0101; d = 4'b1xx0; e = 4'b0;
```

```
    $displayb(~a);           //结果0011
```

```
    $displayb(a&c);          //0100
```

```
    $displayb(a|b);          //1111
```

```
    $displayb(b^c);          //0110
```

```
    $displayb(a~^c);         //0110
```

```
    $displayb(d&e);          //0000
```

```
end
```

```
endmodule
```

## 5.3 逻辑运算符 (&&, ||, !)

a	b	!a	!b	a&& b	a  b
0	0	1	1	0	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	1	1

例如: `module LogicalTest;`

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

`initial begin`

`a=2; b=0; c=4'hx;`

`$display(a&&b);`

`//结果0`

`$display(a||b);`

`//1`

`$display(!a);`

`//0`

`$display(a||c);`

`//1`

`$display(a&&c);`

`//x`

`$display(!c);`

`//x`

`$display(b&&c);`

`//0`

`end`

`endmodule`

## 5.4 关系运算符 (>, <, >=, <=)

- 所有关系运算符有相同优先级别，而关系运算符优先级别低于算术运算符，如：

$a < \text{size}-1$  同  $a < (\text{size}-1)$ ，而  $\text{size}-(1 < a)$  不同于  $\text{size}-1 < a$

例如：

```
module RelationTest;
```

```
reg [3:0] a, b, c, d, e;
```

```
initial begin
```

```
    a=2; b=5; c=2; d=4'hx; e= -2;
```

```
    $display(a<b);           // 1
```

```
    $display(a>b);           // 0
```

```
    $display(a>=c);          // 1
```

```
    $display(d<=a);          // x
```

```
    $display(4'b0 <= 4'hx);   // x
```

```
end
```

```
endmodule
```

```
$display(-2<b);             // 0
```

```
$display(-2<5);             // 1
```

```
$display(2-5);              // -3
```

```
$display(a-b);              // 13
```

```
$display(e);                // 14
```

## 5.5 相等运算符 (==, !=, ===, !==)

- ==与!=又称逻辑等式运算符，若操作数含x或z，结果可能为不定值
- ===与!==则将x与z看作一种逻辑状态，===的两个被比较操作数必须完全一致结果才能为1，否则为0。（注意：===和!==不可综合）
- 常用于case表达式判别，故又称“case等式运算符”
- ==与===的真值表如下：

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

例如：

```
module LogicalTest;
reg [3:0] a, b, c, d, e, f;
initial begin
    a=4; b=7; c=4'b010; d=4'bx10;
    e=4'bx101; f=4'bxx01;
    $displayb(c); //结果0010
```

```
$displayb(d); //xx10
$display(a==b); //0
$display(c!=d); //x
$display(c!=f); //1
$display(c!==d); //1
end
endmodule
```



## 5.6 移位运算符 (<<, >>)

- $a \ll n$  或  $a \gg n$ ,  $a$  是操作数,  $n$  表示移几位, 用0填充移出的空位
- 注意移位前后变量位数, 赋值给位数更多的变量会看到这样的结果:  
 $4'b1001 \ll 1 = 5'b10010$ ;  $4'b1001 \ll 2 = 6'b100100$ ;  $1 \ll 6 = 32'b1000000$ ;  
 $4'b1001 \gg 1 = 4'b0100$ ;  $4'b1001 \gg 4 = 4'b0000$ ;

- 例子:

```
module ShiftTest;
reg [3:0] a, b, c, d;
initial begin
    a=4'b1010;
    $displayb(a<<1);           //结果4'b0100
    $displayb(a>>2);           // 4'b0010
    $displayb(4'bx<<2);         // 4'bx00
    $displayb(4'b1101<<2);     // 4'b0100
end
endmodule
```

## 5.7 连接与复制操作 { } concatenation

- 把多个信号的某些位拼接起来进行运算操作，或将多个小的表达式合并形成一个大的表达式，各式间用逗号分隔  
 $\{a,b[3:0],w,3'b101\}$ 即 $\{a,b[3],b[2],b[1],b[0],w,1'b1,1'b0,1'b1\}$
- 表达式中不允许存在未指明位数的信号，因为计算拼接信号位宽时系统必须知道每个信号的位宽
- 可以用重复法（复制运算符）简化表达式，例如：  
 $\{4\{w\}\} = \{w,w,w,w\}$ ;      $\{b,\{3\{a,b\}\}\} = \{b,a,b,a,b,a,b\}$   
用于表示重复次数的表达式必须是常数（上面的4和3）

```
module RelationTest;
    reg a; reg [1:0]b; reg [5:0] c;
    initial begin
        a=1'b1; b=2'b00; c=6'b101001;
        $displayb({a, b});           //结果3'b100
        $displayb({c[5:3], a});       // 4'b1011
        $displayb({4{a}});           // 4'b1111
    end
endmodule
```

## 5.8 归约/缩减reduction运算符

- 单目运算符包括归约与&、归约或|、归约异或^及它们的非运算~&、~|、~^
- 普通的位操作，结果位数与操作数位数相同；而归约运算是对单个操作数递推运算：先将操作数第1位与第2位进行运算，再将结果与第3位运算，依此类推，直至最后1位。归约操作的结果是1位二进制数。例如：

reg [3:0] b; reg c; c = &b; 相当于  
c = ( (b[0] & b[1]) & b[2] ) & b[3];

```
module ReductionTest;
  reg [3:0] a, b, c;
  initial begin
    a=4'b1111; b=4'b0101;
    c = 4'b0x1z;
    $displayb(&a); //结果1
    $displayb(|b); // 1
    $displayb(^b); // 0
```

```
    $displayb(&c); //结果0
    $displayb(|c); // 1
    $displayb(^c); // x
  end
endmodule
```

## 5.9 条件运算符 (? :)


- 第一个操作数为1，算子返回第二个操作数；第一个操作数为0，则返回第三个操作数；若第一个操作数为x或z，则按下表逻辑将第二、三操作数按位比较得最终结果：

?:	0	1	x	z
0	0	x	x	x
1	x	1	x	x
x	x	x	x	x
z	x	x	x	x

```
module add_or_sub(a, b, op, result);  
    parameter ADD=1'b0;  
    input [7:0] a, b;    input op;  
    output [7:0] result;  
    assign result = (op==ADD)? a+b : a-b;  
endmodule
```

## 5.10 其他问题

1. 运算符优先级如右图：
2. 关键词：Verilog中关键词是事先定义好的确认符，用来组织语言结构。关键词用**小写字母**定义，书写时避免出错。详见Verilog HDL标准。
3. 考虑到程序易读性和可移植性的要求，编写Verilog程序时注意不要与关键词冲突。

<code>+ - ! ~ (unary)</code> 一元的	Highest precedence
<code>* / %</code>	
<code>+ - (binary)</code>	
<code>&lt;&lt; &gt;&gt;</code>	
<code>&lt; &lt;= &gt; &gt;=</code>	
<code>== != === !==</code>	
<code>&amp; ~&amp;</code>	
<code>^ ^~ ~^</code>	
<code>  ~ </code>	
<code>&amp;&amp;</code>	
<code>  </code>	
<code>?: (conditional operator)</code>	Lowest precedence

## 5.11 小结

1. 无论逻辑运算、逻辑比较还是逻辑等式等逻辑操作一般发生在条件判断语句中，其输出只有1或0，可理解为成立(真)或不成立(假)
2. 借助位连接运算符，可用一个信号名来表示由多位信号组成的复杂信号，其中每个功能信号可以有自己独立的名称和位宽。例如控制信号可以用如下的位连接表示：  
`assign control = {read, write, sel[2:0], halt, load_instr,...};`
3. 合理使用归约(缩减)运算符可使程序简洁、明了。

# 六. Verilog HDL语句

- 块语句
- 赋值语句
- 条件、分支语句
- 循环语句
- 结构说明语句
- 小结

# 6.1 块语句

块语句通常用来将两条或者多条语句组合在一起，使其格式上看来更像一条语句。块语句有两种：

- 一种是**begin-end**语句，通常用来标识顺序执行的语句，这种块称为顺序块；
- 另一种是**fork-join**语句，通常用来表示并行执行的语句，这种块称为并行块。

**块名：** Verilog HDL中可以给每个块取一个名字，只需将名字加到关键词**begin**或**fork**之后，有以下原因：

- 有名块可以在块内定义局部变量，即只在块内使用的变量；
- 有名块可允许该块被其他语句调用，如**disable**语句。
- Verilog HDL里所有的变量是静态的，即有一个唯一的存储地址，跳入跳出块不影响存储在变量内的值，块名可提供在任何仿真时刻确认变量值的方法。



## 6.1.1 顺序块begin-end

- 块内语句顺序执行，即上面一条语句执行完后下面的语句才能执行；
- 每条语句的延迟时间相对于前一条语句的仿真时间而言；
- 直到最后一条语句执行完，程序流程控制才跳出该语句块

格式:

```
begin
```

```
    语句1;
```

```
    语句2;
```

```
    ...
```

```
    语句n;
```

```
end
```

```
begin : 块名
```

```
    块内声明语句;
```

```
    语句1;
```

```
    ...
```

```
    语句n;
```

```
end
```

- 块名即该块的名字，一个标识；
- 块内声明可以是参数声明、**reg**变量声明、**integer**型变量声明和**real**型变量声明语句；

# 顺序块(续)

```
begin
    areg = breg;
    creg = areg; //creg值为areg值
end
//第一条语句先执行, areg值更新
//为breg值, 两语句之间没有延时,
//creg值实际更新为areg值。
```

```
begin
    areg = breg;
    #10 creg = areg;
    //在两条语句之间延迟10个时间单位
end
```

```
parameter d=50;           //声明延迟参数d
reg [7:0] r;               //声明r是8bit寄存器变量
begin                      //由一系列延迟控制组合产生一个时序波形
    #d r = 'h35;
    #d r = 'he2;
    #d r = 'h00;
    #d r = 'hf7;
    #d -> end_wave;       //触发事件end_wave
end
```

## 6.1.2 并行块fork-join

- 块内语句同时执行，流程一进到并行块，块内语句一起开始并行执行
- 块内每条语句延迟时间是相对于程序流程控制进入到块内的仿真时间的
- 延迟时间用来给赋值语句提供执行时序；
- 当按时间顺序排在最后的语句执行完，或者一个**disable**语句执行时，程序流程控制跳出该并行块。

格式:

**fork**

语句1;

语句2;

...

语句n;

**join**

**fork : 块名**

块内声明语句;

语句1;

...

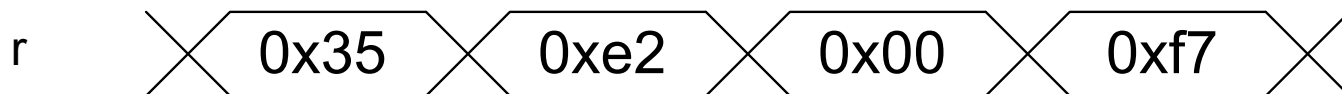
语句n;

**join**

- 块名即该块的名字，相当于一个标识符；
- 块内声明可为参数声明、**reg**变量声明、**integer**型变量声明和**real**型变量声明语、**time**型变量声明、**event**说明语句

## 并行块(续)

```
reg [7:0] r; //声明r是8bit寄存器变量
fork          //由一系列延迟控制组合产生一个时序波形
    #50  r = 'h35;
    #100 r = 'he2;
    #150 r = 'h00;
    #200 r = 'hf7;
    #250 -> end_wave; //触发事件end_wave
join
```

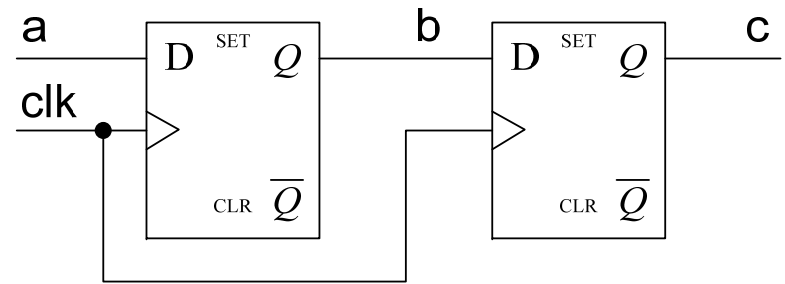


该并行块可以产生与前面顺序块相同的数据波形，注意延迟时间的计算方法。

## 6.2.1 赋值语句—非阻塞(nonblocking)赋值

- 形式如**b<=a**，<=是赋值运算不是比较运算，形式一样，但含义和使用位置不同，系统不会搞错；
- 块结束后才完成赋值操作；
- b的值并不是立即改变的；
- 这是时序逻辑常用的赋值方法(特别是编写可综合模块时)

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



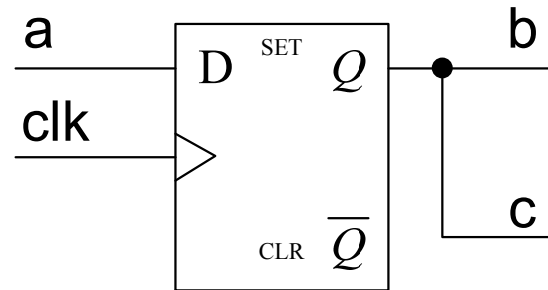
上面模块中用非阻塞赋值方式定义了两个reg型变量b和c，时钟上升沿到来时，b等于a，（同时）c等于b，这里形成了两个触发器。

注意：赋值是在always块结束后执行的，b为原来a值，c为原来b值。

## 6.2.2 赋值语句—阻塞(blocking)赋值

- 形式如 `b=a`
- 语句执行完后块才结束;
- `b`的值在赋值语句执行完后立刻就改变
- 边沿触发的`always`块中使用阻塞赋值可能产生意想不到的结果

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



上面模块中用阻塞赋值方式定义了两个`reg`型变量`b`和`c`，时钟上升沿到来时，发生如下变化：`b`立刻取`a`值，（之后）`c`取`b`值(等于`a`)，**同一时刻但有先后**。生成上面的电路，只有一个触发器，这可能并不是设计的初衷。

## 6.2.3 起始时间和结束时间

- 并行块和顺序块都有起始时间和结束时间的概念。对于顺序块，起始时间就是第一条语句开始被执行的时间，结束时间是最后一条语句执行完的时间。而对于并行块，起始时间对于块内所有语句来说都是相同的，即程序流程进入该块的时间，而结束时间是按时间排序在最后的语句执行结束的时间。
- 当一个块嵌入另一个块时，块的起始和结束时间很重要，跟在后面的语句只有在该块结束时间到了才能开始执行。
- **fork-join**块内各条语句可以不按顺序给出，就是说并行块里各语句的前后关系无关紧要，如下例：

```
fork
    #250 -> end_wave;
    #100 r = 'he2;
    #200 r = 'hf7;
    #150 r = 'h00
    #50  r = 'h35;
join
```

## 6.3 条件语句

- if-else语句
- case语句(case/casex/casez)



## 6.3.1 条件语句if-else

- if 语句用来根据给定条件的判定结果给出两种操作之一。  
Verilog HDL提供3种形式:

- 形式(1)

```
if ( a > b )  
    out1 = in1;
```

- 形式(2)

```
if ( a > b )  
    out1 = in1;  
else  
    out1 = in2;
```

- 形式(3)

```
if      ( a > b )  
    out1 = in1;  
else if ( a == b )  
    out1 = in2;  
else if .....  
    .....  
else  
    out1 = in3;
```

# 关于if-else说明

- (1) if后面都有表达式，()里部分，一般为逻辑表达式或关系表达式，系统对表达式为1按“真”处理，若为0、x、z，按“假”处理。
- (2) 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 )
- (3) if和else后面可以包含一个内嵌的操作语句，有多个语句时需要用begin-end关键词包含起来组成复合块语句，如右边程序：
- (4) else语句不能单独使用，必须和if语句成对使用

```
if ( a > b )  
begin  
    out1 <= in1;  
    out2 <= in2;  
end  
else  
begin  
    out1 <= in2;  
    out2 <= in1;  
end
```

# 关于if-else说明(续)

(5) if语句可以嵌套使用，**else**总是和它上面最近的if配对。若if与**else**的数量不同，为实现设计者意图和便于程序阅读，可以用**begin-end**块语句来确定配对关系。

```
if ( index > 0 )
  for (i=0; i<index; i=i+1)
    if (mem[i] >0 )
      begin
        $display("...");
        mem[i] = 0;
      end
else /*WRONG*/
  $display("error-index is zero");
```

```
if ( index > 0 )
  begin
    for (i=0; i<index; i=i+1)
      if (mem[i] >0 )
        begin
          $display("...");
          mem[i] = 0;
        end
    end
else /*RIGHT*/
  $display("error-index is zero");
```

上面程序原意应是**else**与第一个if配对，而实际上**else**是与第2个if成对，因为它离第2个if最近，正确写法如右侧程序。

## 6.3.2 条件语句case

- **case**语句是一种多分支语句，可直接处理if语句难以处理的多分支问题，常用于微处理器的指令译码等
- 一般形式如下3种：
  - (1) **case** (表达式) <case分支项> **endcase**
  - (2) **casex** (表达式) <case分支项> **endcase**
  - (3) **casez** (表达式) <case分支项> **endcase**
- 其中，**case**分支项的一般格式如下：
  - 分支表达式1 : 语句1;
  - 分支表达式2 : 语句2;
  - ...
  - 分支表达式n : 语句n;
  - default** : 语句; /\*默认项\*/

# case语句说明1

- (1) **case**括号内的表达式称为控制表达式，通常表示控制信号的某些位；**case**分支项内的表达式称为分支表达式，常用控制信号的具体状态值来表示，又称常量表达式。
- (2) 控制表达式与分支表达式值相等时就执行分支表达式后面的语句，若所有分支表达式值都不匹配，则执行**default**后面的语句。
- (3) 可无**default**项，一个**case**语句里只可有一个**default**项
- (4) 每个**case**分支项的分支表达式的值必须互不相同，否则就会矛盾(同一个表达式值多种执行方案)
- (5) 执行完**case**分支项后的语句，就跳出**case**语句结构，终止**case**语句执行。

```
case ( rega )  
    16'd0: result=7'b0111111;  
    16'd1: result=7'b1011111;  
    16'd2: result=7'b1101111;  
    16'd3: result=7'b1110111;  
    16'd4: result=7'b1111011;  
    16'd5: result=7'b1111101;  
    16'd6: result=7'b1111110;  
    default: result=7'bx;  
endcase
```

# case语句说明2

- (6) 用**case**表达式进行比较时，只有当信号对应位的值能明确进行比较时，比较才能成功，因此需要详细说明**case**分支项的分支表达式的值(位宽表达式)
- (7) **case**语句所有表达式值的位宽必须相等，只有这样控制表达式与分支表达式才能进行逐位对应比较。用'**bx**'、'**b0**'来替代'**n'bx**'、'**n'b0**'是不正确的，因为'**bx**'默认宽度是机器字节宽度，常为32位，并非**case**控制表达式位宽**n**。(Modelsim将位宽较短的分支表达式补0后比较)
- (8) **case**语句提供**casex**和**casez**语句用于处理控制表达式和分支表达式中含有**x**和**z**位的情况，以此可以灵活设置信号比较方式：  
**casez**用于处理不考虑高阻**z**的比较过程；  
**casex**用于处理**x**和**z**都不必考虑的情况。

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

```
reg [7:0] r, mask;
mask = 8'bx0x0x0x0;
casex ( r ^ mask )
    8'b001100xx : stat1;
    8'b1100xx00 : stat2;
    8'b00xx0011 : stat3;
    8'bx001100 : stat4;
endcase
```

## 6.4 循环语句

- **forever** : 连续执行的语句
- **repeat** : 连续执行一条语句n次
- **while** : 执行一条语句到某个条件不满足,  
若一开始就不满足, 则一次也不执行
- **for** : 根据给定条件循环执行语句

## 6.4.1 forever语句

- 常用于产生周期波形，作为仿真测试信号。与 **always** 语句不同之处在于它不能独立写在程序中，而**必须存在于initial块里（只执行一次）**。
- 语法格式如下：
  - (1) **forever** 语句;
  - (2) **forever**  
    **begin**  
        多条语句;  
    **end**



## 6.4.2 repeat语句

- 格式:
  - (1) **repeat** (表达式)  
语句;
  - (2) **repeat** (表达式)  
**begin**  
多条语句;  
**end**
- 表达式通常为常量表达式
- 右面例子用**repeat**循环及加法和移位操作实现乘法器

```
parameter size=8, longsize=16;
reg [size : 1] opa, opb;
reg [longsize : 1] result;
begin : multi
    reg [longsize : 1] shf_opa;
    reg [longsize : 1] shf_opb;
    shf_opa = opa; shf_opb = opb;
    result = 0;
    repeat (size)
        begin
            if (shf_opb[1])
                result=result+shf_opa;
            shf_opa= shf_opa<<1;
            shf_opb = shf_opb>>1;
        end
    end
end
```

## 6.4.3 while语句

- 格式:
  - (1) **while** (表达式) 语句;
  - (2) **while** (表达式)  
    **begin**  
        多条语句;  
    **end**
- 右面例子用**while**循环语句对8位二进制数**rega**中“1”的个数进行计算

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

## 6.4.4 for循环语句

- 格式：  
**for**(表达式1;表达式2;表达式3)  
    语句;
- 执行过程：
  - (1) 先求解表达式1
  - (2) 再求解表达式2,若为真(非0),则执行**for**内嵌语句,然后执行第(3)步;若为假,则结束循环,到(5)步
  - (3) 若表达式(2)为真, 在执行指定语句后求解(3)
  - (4) 转回第(2)步继续执行
  - (5) 执行**for**语句后面的语句
- 最简单的应用形式如下, 易于理解:  
**for** (循环变量初值; 循环结束条件; 循环变量增值)  
也可以用**while**循环语句实现

# for语句续1

- 用for语句实现前面用repeat语句实现的乘法器

```
parameter size = 8, longsize = 16;
```

```
reg [size : 1] opa, opb;
```

```
reg [longsize : 1] result;
```

```
begin : multi
```

```
    integer bindex;
```

```
    result = 0;
```

```
    for ( bindex =1; bindex <= size; bindex = bindex+1 )
```

```
        if ( opb[bindex] )
```

```
            result = result + ( opa << ( bindex-1) );
```

```
end
```

# for语句续2

- for语句中，循环变量增值表达式可以不必是一般常规加法或减法表达式，下例是对rega这个8位二进制数中1个数计数的另一种方法：

```
begin : count1s
```

```
    reg [7:0] tmpreg;
```

```
    count =0;
```

```
    for (tmpreg = rega; tmpreg; tmpreg=tmpreg>>1)
```

```
        if ( tmpreg[0])
```

```
            count = count +1;
```

```
end
```

## 6.5 结构说明语句

- **initial**
  - **always**
  - **task** : 任务
  - **function** : 函数
- 
- **task**和**function**分别用来定义任务和函数，利用任务和函数可以把一个很大的程序模块分解成许多小的任务和函数，便于理解和调试。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。

## 6.5.1 initial语句

- initial语句格式如下：

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

- 例，初始化寄存器areg：

```
initial
begin
    areg = 0;
    for (i=0;i<size;i=i+1)
        mem[i] = 0;
end
```

//注意，顺序块里未指定延时，所有程序同时执行，即初始化过程不需要时间；

- 例：initial  
begin

```
        inputs = 'b0000000; //生成激励波形作为测试仿真信号
#10 inputs = 'b011001;
#10 inputs = 'b011011;
#10 inputs = 'b011000;
#10 inputs = 'b001000;
```

```
end // initial语句常用于测试文件和虚拟模块编写
```

## 6.5.2 always语句

- 声明格式: **always** <时序控制> <语句>
- **always**语句具有不断重复执行特性, 只有和一定时序控制结合才有用, 否则会产生**仿真死锁**, 如:  
**always areg = ~areg;**  
将生成0延迟无限循环跳变过程, 发生仿真死锁。
- 增加时序控制后: **always #PERIOD areg = ~areg;**  
产生周期为 $2 \times \text{PERIOD}$ 的无限延续波形, 可作时钟

例:

```
reg [7:0] counter;  reg tick;
always @ (posedge areg)
begin
    tick = ~ tick;
    counter = counter + 1;
end
//每当areg信号上升沿出现,
//把tick信号反相, counter增加1
```

```
always @ (posedge clk or negedge reset)
begin
    ... ;
end //边沿触发, 时序逻辑
always @ (a or b or c)
begin
    ... ;
end //电平触发, 组合逻辑
```



# Initial和always比较

- 一个程序模块可以有多个**initial**和**always**过程块，每个**initial**和**always**块在仿真的一开始就同时立即开始运行。
- **initial**语句只执行一次，而**always**语句则不断重复活动，直到仿真过程结束。
- **always**语句后面跟随的程序块是否运行由**always**后面的触发条件决定，若满足就运行一次，再满足条件再运行，直到仿真过程结束(**always**语句一直在检测触发条件是否满足，若满足就执行包含的语句块)

## 6.5.3 任务task语句

- 任务定义语法

```
task <任务名>;  
    <端口及数据类型说明语句>  
    <语句1>  
    ...  
    <语句n>  
endtask
```

- 任务调用和变量传递

调用： <任务名> (端口1, 端口2, ..., 端口n);

```
task my_task; //任务定义  
    input a, b;  
    inout c;  
    output d, e;  
    ... 语句 ... //执行任务工作语句  
    c = foo1; //对任务输出变量赋值  
    d = foo2;  
    e = foo3;  
endtask
```

- 任务完成后，控制就传回启动过程，若任务内部有**定时控制**，则启动的时间可以与任务返回的时间不同；
- 任务可以启动其他任务，任务数量没有限制，只有当所有任务启动完成后，控制才能返回。

调用任务： my\_task(v, w, x, y, z);  
/\*任务调用变量(v, w, x, y, z)与任务定义I/O变量(v, w, x, y, z)之间一一对应，任务启动时，由v, w, x传入的变量赋给a, b, c，任务完成后的输出又通过c, d, e赋给了x, y, z。\*/

# task例子：简单的交通灯控制器

```
module traffice_lights;
    reg clock, red, amber, green;
    parameter red_ticks = 350, on = 1,
               amber_ticks = 30, off = 0,
               green_ticks = 200 ;
    initial red = off; //交通灯初始化
    initial amber = off;
    initial green = off;
    //交通灯控制时序
    always begin
        red=on;
        light(red, red_ticks);
        green=on;
        light(green, green_ticks);
        amber=on;
        light(amber, amber_ticks);
    end
```

```
task light; //交通灯定时任务
    output color;
    input [31:0] ticks;
    begin
        repeat(ticks)
            @(posedge clock);
            color = off;
    end
endtask

always begin //时钟发生模块
    #100 clock=0;
    #100 clock=1;
end

endmodule
```

## 6.5.4 函数function语句

- 函数定义语法

**function** <返回值类型或范围> (函数名);

<端口说明语句>

<变量类型说明语句>

**begin**

<语句> ...

**end**

**endfunction**

//函数要返回一个用于表达式的值

- 函数的定义蕴涵了由<返回值类型或范围>定义的、与函数同名的、函数内部的寄存器。即函数的定义把函数返回值寄存器的名称初始化为与函数同名的内部变量。

- 函数调用:

<函数名> (<表达式> <, <表达式>\*>);

//“返回值类型或范围”若缺省,  
// 返回值为1位寄存器类型数据

函数定义例子:

**function** [7:0] getbyte;

**input** [15:0] address;

**begin**

<说明语句>

//从地址提取低字节程序

getbyte = result\_expression;

//把结果赋值给返回值字节

**end**

**endfunction**

函数调用例子, 对两次调用getbyte的结果进行位拼接运算生成一个word:

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

## • function使用规则

- (1) 函数定义不能包含任何时间控制语句，即用#、@或wait标识的语句；
- (2) 函数不能启动任务；
- (3) 定义函数时至少需要有一个输入变量；
- (4) 函数定义中必须有一条赋值语句给与函数同名的一个内部变量赋以函数的结果值作为返回值；

## • function与task说明语句不同点

- (1) 函数只能与主模块共用一个仿真时间单位，而任务可以定义自己的仿真时间单位；
- (2) 函数不能启动任务，而任务可以启动其他任务和调用函数；
- (3) 函数至少要有一个输入变量，而任务可没有或有多个任何类型变量；
- (4) 函数返回一个值，而任务不返回值：函数通过返回一个值来响应输入信号值，Verilog把函数返回值作为表达式操作符；任务可支持多种目的，可有多个结果，结果只能通过被调用任务的输出或者总线端口送出。

```
switch_bytes ( old_word, new_word );      (任务)
new_word = switch_bytes ( old_word );     (函数)
```

# function举例

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

左面是一个可以进行阶乘运算的名为**factorial**的函数，该函数返回一个**32位**的寄存器类型的值。右面调用该函数计算验证**2~9**的阶乘，并打印出部分结果。右下角是用**ModelSim**软件仿真输出的结果。

```
//（续左侧）函数的测试—————
reg [3:0] n;
initial begin
    for (n=2;n<=9;n=n+1)
        begin
            $display("n=%d result =%d",
                n, factorial(n) );
        end
    end
endmodule //模块结束
```

```
# n= 2 result =      2
# n= 3 result =      6
# n= 4 result =     24
# n= 5 result =    120
# n= 6 result =   720
# n= 7 result =  5040
# n= 8 result = 40320
# n= 9 result = 362880
```

## 6.6 小结

1. 阻塞赋值语句，如果没写延迟，看起来是在同一时刻运行，实际上有先后，前面的先运行，然后再运行后面的，因此阻塞赋值语句的顺序与逻辑行为有很大关系。而非阻塞赋值时，**begin**到**end**间所有非阻塞语句都在同一时刻被赋值，其逻辑行为与顺序没有关系，硬件实现时二者有很大不同
2. **begin~end**块语句与C语言里**{}**类似。测试模块中，描述测试信号常在**initial**和**always**过程中使用并行块，这种描述方法时间关系只与起点比较，有时这样表达较容易和清楚
3. 一个模块可以有多个**initial**和**always**过程块，在仿真一开始就同时立即开始运行
4. **initial**语句在模块中只执行一次，而**always**则不断重复执行直到仿真结束
5. **always**块后的语句是否运行取决于其触发条件是否满足，满足则运行过程块一次，循环往复直到仿真过程结束
6. **always**可以边沿触发也可以电平触发，可以是单个或多个信号，中间需用关键词**or**连接。
7. 沿触发的**always**块常常描述时序行为，如有限状态机等
8. 电平触发的**always**块常用来描述组合逻辑行为

## 七. Verilog常用语法—系统任务等

Verilog HDL约有20个左右的系统函数和任务，  
重点讲几个常用于调试和查错的任务

- \$display和\$write                      标准输出任务
- \$monitor                                仿真监控任务
- \$finish和\$stop                        仿真结束任务
- \$time和\$realtime等                  时间函数
- \$fopen...\$readmemh等              文件、存储器处理
- \$random                                随机数函数



# 7.1 \$display和\$write标准输出任务

- 格式：\$display(p1, p2, ..., pn); //输出后自动换行  
\$write(p1, p2, ..., pn); //不自动换行
- 两个任务都用来将“输出表列” p2~pn按照“格式控制”参数p1给定的格式输出。p1是用双引号扩起来的字符串，可以包括格式说明符和普通字符两种信息。
- 格式说明符有：%h、%d、%o、%b、%c、%v、%m、%s、%t、%e、%f、%g等  
普通字符可以包含\n、\t、\\、\"、\o、%%等转义符
- \$display输出列表数据显示宽度自动按输出格式调整，总是用表达式值最大可能占的位数显示当前值；十进制输出时结果前面的0用空格代替，其他进制结果的0仍显示出来。在%和表示进制字符间插0(如%0h)可使输出数据宽度自动调整，总是用最少位数显示表达式当前值。
- 若表达式所有位均为不定值或高阻，则输出结果为小写x或z  
若表达式部分位为不定值或高阻，则输出结果为大写X或Z

格式符	含义
%h或%H	以十六进制的形式输出
%d或%D	以十进制的形式输出
%o或%O	以八进制的形式输出
%b或%B	以二进制的形式输出
%c或%C	以ASCII码字符的形式输出
%s或%S	以字符串的形式输出
%v或%V	输出网络型数据信号强度
%m或%M	输出等级层次的名字
%t或%T	以当前的时间格式的形式输出
%e或%E	以指数的形式输出实型数
%f或%F	以十进制的形式输出实型数
%g或%G	以指数或者十进制的形式输出实型数，以较短的结果输出
%x	十六进制
%i	读入十进制，八进制，十六进制，在编译时通过数据前置区分
%u	无符号十进制数

```
module disp;
    reg [31:0] rval=101;    pulldown (pd);    wire w;    reg r=0;
    initial begin    //rval = 101;
        $display("rval=%h hex %d decimal", rval, rval);
        $display("rval=%o otal %b binary", rval, rval);    //
        $display("rval=%0o otal %0b binary", rval, rval);    //
        $display("rval has %c ascii character value", rval);    //
        $display("pd strength value is %v", pd);    //
        $display("w strength value is %v", w);
        $display("r strength value is %v", r);
        $display("current scope is %m");    //
        $display("%s is ascii value for 101", 101);    //
        $write("simulation time is %t ", $time);
        $write("%h %o", 12'b001x_xx10_1x01, 12'b001_xxx_
    end
endmodule
```

```
//%m module hierarchy
//%s string
```

```
# rval=00000065 hex      101 decimal
# rval=00000000145 otal 00000000000000000000000000001100101 binary
# rval=145 otal 1100101 binary
# rval has e ascii character value
# pd strength value is StX //means state X, 需要持续驱动才会有非X值
# w strength value is StX
# rval strength value is St0 //means state 0
# current scope is disp
# e is ascii value for 101
# simulation time is      0 XXX 1x5X //注意这里由于write没换行
```

## 7.2 \$monitor仿真监控任务

- 格式: `$monitor(p1, p2, ..., pn);`  
    `$monitor;` //输出一空白行  
    `$monitoron;`和`$monitoroff;`
- 格式1的语法与`$display`语法一样, 自动换行, 用于连续监控指定信号, 若发现其中任何一个信号发生变化, 就按p1指定格式在时间步结束时输出整个信号表列。
- 同一时刻只允许一个**`$monitor`**输出, 在多模块调试时, 若许多模块都调用`$monitor`, 需要配合`$monitoron`和`$monitoroff`, 把需要监视的模块的监视打开, 监视完毕再关闭, 以允许其他模块使用`$monitor`输出
- 与`$display`不同之处在于`$monitor`通常在`initial`块中调用, 只要不调用`$monitoroff`, 便不间断的对设定的信号监视

# \$monitor例子

```
module TestMonitor;
  integer a, b;
  initial begin
    a = 2;
    b = 4;
    forever begin
      #5 a = a + b;
      #5 b = a - 1;
    end
  end
end

initial #40 $finish(2);

initial begin
  $monitor ($time,
    " a=%d, b=%d", a, b);
end
endmodule
```

ModelSim仿真结果输出:

```
#    0  a=      2, b=      4
#    5  a=      6, b=      4
#   10  a=      6, b=      5
#   15  a=     11, b=      5
#   20  a=     11, b=     10
#   25  a=     21, b=     10
#   30  a=     21, b=     20
#   35  a=     41, b=     20
# ** Note: Data structure takes
1835040 bytes of memory
#    Process time 1822.58 seconds
#    $finish   : TestMonitor.v(12)
#    Time: 40 ns Iteration: 0 Instance:
/TestMonitor
```

## 7.3 \$stop和\$finish仿真结束任务

- **\$finish**格式:

`$finish;`           //退出仿真器,回主操作系统

`$finish(n);`       //

n=0 不输出任何信息

n=1 输出当前仿真时刻和位置

n=2 输出当前仿真时刻,耗费内存和CPU时间

- **\$stop**格式:

`$stop;`           //暂停仿真,进入用户交互状态

`$stop(n);`       //

可通过输入其它命令(如run xxx)继续仿真过程

## 7.4.1 时间函数—\$timeformat和\$sprnttimescale

- `$timeformat(<unit>, <precision>, <suffix>, <min_field_width>);`

用于控制%t格式显示时间信息的方式，其中：

**unit** 用于指定时间单位0 ~ -15，单位是秒

**precision** 显示时间信息精度，小数点后位数

**suffix** 后缀如ms、us、ns、ps、fs等，给用户看的

**min\_field\_width** 时间信息最小字符数

- 例如：

```
$timeformat(-10, 3, "x0.1nS", 10);
```

```
$display("Current simulation time is: %t", $time);
```

在仿真时间为10nS时的输出结果为：

```
# Current simulation time is: 100.000x0.1nS
```

- `$sprnttimescale(module_hierachical_name);`

显示指定模块时间单位与精度，若无参数则显示包含该任务调用的所有模块的时间单位与精度。

如ModelSim默认：

```
# Time scale of (TestMonitor) is 1ns / 1ns
```

## 7.4.2 时间函数— \$time、\$stime和\$realtime

- \$time 返回一个**64位整数**表示当前仿真时刻
- \$stime 返回一个**32位整数**表示当前仿真时刻
- \$realtime 返回一个**实数**表示当前仿真时刻

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

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

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

# 0 set=x
# 1.6 set=0
# 3.2 set=1
```



# 关于上面例子

- 左面例子**TestTime**在**16ns**时将寄存器**set**设为**0**，在**32ns**时将**set**设为**1**，但由**\$time**记录的**set**变化时刻却并非如此，是由于以下两个原因：
  - (1) **\$time**显示时刻受时间尺度(**timescale**)比例影响，上页时间尺度是**10ns**，而**time**输出总是时间尺度的倍数，因此**16ns/32ns**输出为**2/3**
  - (2) **\$time**总是输出整数，经尺度变换的数字输出时先进行取整，**1.6/3.2**四舍五入取整后为**2/3**。这里时间精度并不影响数字的取整。
- 右面例子中**\$realtime**返回时刻是实数，因此仿真时刻经尺度变换后直接输出，并不取整。

## 7.5.1 文件管理

- **<文件句柄> = \$fopen("<文件名>");**

打开文件: 若路径和文件名正确,返回一个32位的句柄,其中只有一位是1,否则返回出错信息。标准输出占据0x01句柄,第一次调用**\$fopen**返回句柄是(0x1<<1),此后每次调用返回的句柄的1位置依次左移。例如:

```
integer handleA, handleB;
```

```
initial begin
```

```
    handleA = $fopen("myfile1.out");    //handleA = 0x02
```

```
    handleB = $fopen("myfile2.out");    //handleB = 0x04
```

```
end
```

- **\$fclose(<文件句柄>);**

文件被关闭,句柄失效,不能再使用句柄写入文件。

- **\$fdisplay(<文件句柄s>, <输出格式>);**

**\$fwrite(<文件句柄s>, <输出格式>);**

**\$fmonitor(<文件句柄s>, <输出格式>);**

输出到文件: 与**display**、**write**、**monitor**类似,不过输出定向到文件

```
handleC = handleA | 1;    //用位操作"|"对句柄进行多位设置
```

```
$fdisplay(handleC, "Hello!"); //同时输出到文件和标准设备
```

## 7.5.2 存储器管理

- `$readmemb/$readmemh`("数据文件名", 存储器名, <起始地址>, <结束地址>);  
前者要求输入文件用二进制数字存数据,后者要求输入文件按十六进制存数据。
  - (1) 数据文件只能包含: 空白、注释行、二进制或十六进制数字, 下划线、x、z使用与Verilog HDL程序要求一致, 数字不能包含位宽说明和格式说明。
  - (2) 数据文件中可以指定某段数据数据存放地址,用十六进制格式@h...h表示
  - (3) 若地址信息在系统任务和数据文件里都进行了说明,那么文件里的地址必须在任务地址参数声明范围内,否则报错

例如: `reg [7:0] mem [1:256];`  
`initial $readmemh("mem.data", mem);`  
`initial $readmemh("mem.data", mem, 128, 1);`

## 7.6 随机函数\$random

- \$random %b; (b>0)

给出一个范围(-b+1):(b-1)有符号32位随机整数

- 利用随机数可以产生随机脉冲序列或随机宽度的脉冲序列用于电路测试。如下例：

```
`timescale 1ns/1ns
module random_pulse (dout);
    output [9:0] dout;
    reg [9:0] dout;
    integer delay1, delay2, k;
    initial begin
        #10 dout = 0;
        for (k=0; k<100; k=k+1)
            begin
                delay1 = 20 * ({$random} % 6);
                //delay1为0/20/40/60/80/100nS
```

```
                delay2 = 20 * (1 + {$random} % 3 );
                //delay2为20/40/60ns
                #delay1 dout =1 << ({$random}%10);
                //延时0~100ns,dout的某位随机出现1
                #delay2 dout = 0;
                //脉冲宽度在20~60ns随机变化
            end
        end
    endmodule
```

dout仿真部分波形如下图：



## 7.7 系统任务小结

- (1) 若不作设置,系统默认\$monitor输出是打开的;  
多模块调试时需配合\$monitoron和\$monitoroff使用;
- (2) \$monitor与\$display不同之处在于\$monitor在initial中调用一次就可以连续监视感兴趣的信号变化,不需要也不能在always过程块中重复调用\$monitor;
- (3) \$time常用在\$monitor中,用于做时间标记;
- (4) \$stop和\$finish常用在测试模块initial块中, 配合时间延迟来控制仿真持续时间;
- (5) \$random可用来产生边沿位置随机、脉冲宽度随机的波形,正确应用它可以有效的发现实际设计中存在的问题;
- (6) \$readmemb/h在编写测试程序时也非常有用,可用来生成给定的复杂数据流,复杂数据可以用C语言产生,存在文件中,用\$readmem导入存储器,再按节拍输出,这在验证算法逻辑电路时特别有用。

# 八. 编译预处理

Verilog HDL和提供了编译预处理功能，允许使用20多个以"```"开头的特殊命令(不是一般的语句)对程序进行预处理，然后再进行通常的编译。

- ``define` 宏定义
- ``include` 文件包含
- ``ifdef...`else...`endif` 条件编译
- ``timescale` 时间定标
- ``uselib` 工作库定义

# 8.1 宏定义`define

- 语法: **`define** 标识符(宏名) 字符串(宏内容)

用一个指定的标识符(名字)代替一个字符串,编译预处理时进行"宏展开"将宏名简单的文本替换为字符串。一些说明如下:

- (1) **`define**可以出现在模块内外均可,有效范围是宏定义之后到源文件结束,或者可以用**`undef**取消宏定义;
- (2) 引用已定义的宏名时必须在宏名前面加上符号"**``**"表示经过宏定义;
- (3) 预处理时宏定义只是做简单文本替换,不作语法检查,编译时才检查语法;
- (4) 宏定义(包括其他预编译指令)不是Verilog HDL语句,不必在行末加分号,若加分号则会被作为宏内容一起替换;但后面可以加**//**或**/\*\*/**注释;
- (5) 宏定义时可以引用已定义的宏名,可以嵌套置换;
- (6) 宏名和宏内容必须在同一行声明,若宏内容中包含注释部分,预处理时会作为被置换部分;
- (7) 宏内容可以是空格,此时宏内容被定义为空,引用时不会有内容被置换;
- (8) 组成宏内容的字符串不能被以下记号分开: 注释行、数字、字符串、确认符、关键词、双目和三目运算符;
- (9) 宏名不应使用与预处理命令相同或与程序内部变量名相同的宏名,否则可能引起兼容性问题或者不便于程序阅读;

# `define例子

```
`define cycle 20 //clock period
module testDefine;
    reg clk;
    always #(`cycle/4) clk = ~ clk;
    //always #(20/4) clk = ~ clk;
endmodule
```

```
`define NAND(dval) nand #(dval)
module testDefine;
    wire y; reg a, b;
    `NAND(3) a1(y, a, b );
    //nand #(3) a1(y, a, b);
endmodule
```

```
`define exp a+b+c+d;
//行尾加分号导致错误
assign out = `exp + e;
    //a+b+c+d;+e;
```

```
`define aa a + b
`define cc c + `aa //嵌套
assign out = `cc;
    //out = c + a + b;
```

```
`define half "start of
$display(`half end string");
//非法的宏定义
```



## 8.2 文件包含`include

- 语法: ``include "文件名"`

一个源文件可以将另一个源文件的全部内容包含进来,可将常用的宏定义命令、任务、函数或类型声明等包含进来,减轻重复劳动,在模块调用时非常有用。

- (1) 包含多个文件要用多个`include
- (2) `include可以出现在Verilog HDL源程序的任何地方,文件名可以使用绝对路径也可以使用相对路径,例如:  
``include "parts/count.v"`
- (3) 可将多个`include命令写在一行,行内只可以出现空格和注释信息  
``include "a.v" `include "b.v" //合法`
- (4) 若文件1包含文件2,而文件2用到文件3,则可在文件1用两个include命令分别包含文件2和3,且文件3应出现在文件2前;
- (5) 一个被包含的文件可以包含另一个文件,即文件包含可以嵌套;
- (6) 同一个工程里含有的文件可不包含。

//文件1: aaa.v

```
module aaa(a, b, out);  
    input a, b; output out;  
    assign out = a^b;  
endmodule
```

//文件2: bbb.v

```
`include "aaa.v"  
module bbb(c, d, e, out);  
    input c, d, e; output out;  
    wire out_a;  
    aaa v(.a(a),.b(d),.out(out_a));  
    assign out = e & out_a;  
endmodule
```

## 8.3 条件编译命令`ifdef...`else...`endif

- 语法两种(宏名是用`define`定义的):

(1) **`ifdef** 宏名(标识符)      (2) **`ifdef** 宏名(标识符)  
      程序段1                                程序段1

**`else**

**`endif**

程序段2

**`endif**

//当宏名被定义过,则对  
//程序段1编译,2被忽略;  
//反之编译2,忽略1

//当宏名被定义过,  
//则对程序段1编译  
//否则忽略该段程序

- 被忽略掉不被编译的程序部分也要符合Verilog HDL语法规则
- 通常在程序中用到条件编译命令的情况有以下几种:
  - (1) 选择一个模块的不同代表部分
  - (2) 选择不同的时序或结构信息
  - (3) 对不同的EDA工具, 选择不同的激励

## 8.4 时间定标命令`timescale

- 格式: **`timescale** <仿真时间单位>/<时间精度>  
用来说明其后模块的时间单位和精度,使得同一个设计里可以包含采用不同时间单位的模块,例如一个设计的两个模块的时间延迟单位分别为ns和ps, EDA工具可以正确进行仿真和测试。
- **时间单位**用于定义模块仿真时间和延迟时间基准单位,有效数字只能取1、10、100, 时间单位有s、ms、us、ns、ps、fs ( $10^0 \sim 10^{-15}s$ );  
**时间精度**参数用于对延迟时间值进行取整操作(仿真前),该参数又被称为“取整精度”。
- 时间精度数值应当小于或等于时间单位值, 如:  
``timescale 1ns/1ps` //合法定标  
``timescale 1ns/10ns` //非法定标
- 若在同一设计里多个模块用到的时间单位不同, 可能用到以下操作:  
(1) 用`timescale声明本模块中用到的时间单位和精度  
(2) 用系统任务\$prnttimescale输出一个模块时间单位和时间精度  
(3) 用系统函数\$time/\$stime/\$realtime及%t格式显示时间信息
- 多个带不同`timescale定义的模块包含在一起时只有最后一个起作用, 因此属于同一个项目但`timescale定义不同的多个模块最好分开编译, 以免把时间搞混。

## 8.5 工作库定义命令`uselib

- 语法： **`uselib** <指定库文件>  
`uselib后面可以是一个带路径的库文件名或一个带路径的库文件目录；若该项缺省表示取消上一次对工作库的定义
- 例如：

```
`uselib file=/models/rtl_lib
```

```
ALU i1(y1, a, b, op); //RTL module
```

```
`uselib dir=/models/gate_lib libext=.v
```

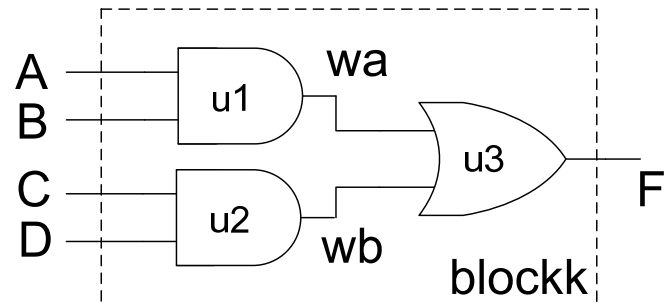
```
ALU i1(y1, a, b, op); //Gate module
```

```
`uselib //close lib
```

## 九. 语法总结

- Ex1

用Verilog HDL描述右面的电路模块



- 参考答案

```
module blockk(F, A, B, C, D);
    input A, B, C, D;
    output F;
```

```
    assign F = ( (A & B) | (C & D) );
```

```
endmodule
```

//3. always 方式

```
reg F;
```

```
always @(A, B, C, D)
```

```
    F = ( (A & B) | (C & D) );
```

## Ex2

- 指出下面几个信号的最高位和最低位

(1) reg [1:0] SEL;

(2) input [0:2] add;

(3) wire [16:23] ab;

- 答案

MSb:最高位, LSb:最低位

MSb SEL[1], LSb SEL[0];

MSb add[0], LSb add[2];

MSb ab[16], LSb ab[23];

## Ex3 下面哪些表示4位输入矢量方法正确

(1) input P[3:0], Q, R;

(2) input P, Q, R [3:0];

(3) input P[3:0], Q[3:0], R[3:0];

(4) input [3:0] P, [3:0]Q, [0:3]R;

(5) input [3:0] P, Q, R;

- 答案

×

×

×

×

✓

## Ex4 指出变量类型(wire/reg)

(1) assign A=B;

(2) always #1  
    Count = C +1;

(3) Cin/Cout/C3/C5

module F(A, B,Cin,Sum,Cout);

    input A, B, Cin;

    output Sum, Cout; ....

endmodule

module Test; ...

    F u1(C1, C2, C3, C4, C5);...

endmodule

答案

A(wire), B(wire/reg)

Count(reg),

C(wire/reg)

A、 B、 Cin(wire)

Sum、 Cout(wire/reg)

C1~C3(wire/reg)

C4、 C5(wire)

//直接调用F模块对应

//信号相连, C5是输出

# Ex5

在下面的程序段中，当ADDRESS取值为5'b0X000时，**casex**执行完后A和B的值各为多少？

A = 0;

B = 0;

casex (ADDRESS)

5'b00??? : A = 1;

5'b01??? : B = 1;

5'b10?00, 5'b11?00 :

begin

A = 1; B = 1;

end

endcase

- 答案

A = 1; B = 0;



# Ex6

- 下列程序中，当V的值发生变化且为-1时，执行完**always**块后**Count**的值应为多少？

```
reg [7:0] V;  
reg [2:0] Count;  
always @(V) begin  
    Count = 0;  
    while (~V[Count])  
        Count = Count + 1;  
end
```

- 答案 **Count = 0;**  
**V = 8'b1111\_1111**  
(reg把-1补码作无符号数)  
  
**Count = 0**  
**V[0] = 1, ~V[0] = 0**  
**while 不满足条件, 不执行**  
**块内操作, 故Count保持**  
**不变, 仍为0**

# Ex7

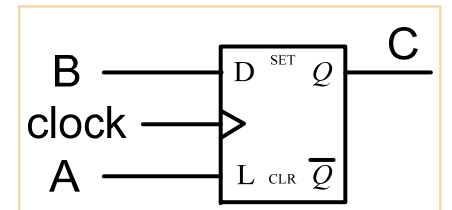
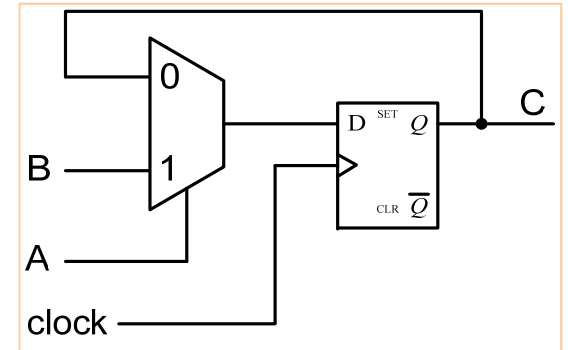
- 下面的模块被综合后可能是哪种硬件实现？

always @(posedge clock)

if (A)

C = B;

- (1) 不能被综合;
- (2) 一个上升沿触发器和一个多路器
- (3) 一个输入是A/B/clock的三输入与门
- (4) 一个透明锁存器
- (5) 一个有clock和使能引脚的上升沿触发器



- 答案 (2)和(5)

# Ex8

- 在下面表达式中选出正确的

(1)  $4'b1010 \& 4'b1101 = 1'b1$

(2)  $\sim 4'b1100 = 1'b1$

(3)  $!4'b1010 \parallel !4'b0000 = 1'b1$

(4)  $\&4'b1101 = 1'b1$

(5)  $1'b0 \parallel 1'b1 = 1'b1$

(6)  $4'b1011 \&\& 4'b0100 = 4'b1111$

(7)  $4'b0101 << 1 = 5'b01011$

(8)  $! 4'b0010 = 1'b0$

(9)  $4'b00001 \parallel 4'b00000 = 1'b1$

- 答案

×

×

(3) ✓

×

(5) ✓

×

×

(8) ✓

(9) ✓

# Ex9

- {1,0}与下面哪个值相等

(1) 2'b01

(2) 2'b10     //{1'b1, 1'b0}

(3) 2'b00

(4) 64'H0000\_000000002

(5) 64'H000000001\_00000000

- 答案 (5)

位拼接运算必须指明位数，若不指明则隐含着为32位二进制整数

ModelSim编译时报错：Illegal concatenation of an unsized constant.

# Ex10

- 下面用initial块给reg [7:0] V赋值，指明每种情况下V的8位数字都是什么值

```
reg [7:0] V;
```

```
initial begin
```

```
    V = 8'b0;
```

```
    V = 8'b1;
```

```
    V = 8'bX;
```

```
    V = 8'bZX;
```

```
    V = 8'bXXZZ;
```

```
    V = 8'b1x;
```

```
end
```

- 答案

```
8'b00000000
```

```
8'b00000001
```

```
8'bXXXXXXXX
```

```
8'bZZZZZZZX
```

```
8'bXXXXXXZZ
```

```
8'b0000001x
```