数字逻辑基础

数制

数码概念：数制中为表示基本数值大小所使用的不同数字符号。如十进制有10个数码0~9，二进制两个数码0、1

基数概念：数制中所使用数码的个数。如十进制使用10个数码，故基数为10；二进制基数为2

位权概念：数制中某位置上的数字1嗦表示数值的大小。如十进制数435中，4所在位置的位权为100

1. 十进制数（Decimal）
2. 二进制数（Binary）
3. 八进制数（Octal）
4. 十六进制数（Hexadecimal）（数码由0~9和A~F组成）
5. 数制之间的转换

（1）非十进制数转换成十进制数

方法：按位权展开后求和即可。如二进制数(10.1) = 1 * 2^1 + 0 * 2^0 + 1 * 2^-1 = 十进制数(2.5)

（2）十进制数转换成非十进制数

①整数部分：不断除以基数取余数，直到商为0，从下到上读取余数

②小数部分：不断除以基数取整数，从上到下读取整数，直到满足精度要求为止

（3）二进制数与八进制数或十六进制数之间的转换

八进制数中的数码0~7与3位二进制数000~111一一对应，十六进制中0~F与二进制0000~1111对应。按位转换即可

码制

对于符号数，一般将数的最高位设为符号位，0表示+，1表示-。这种符号数称为机器数，机器数有三种表示方法:

1、原码

0正1负表示法。如果两个异号的数相减，对机器来说会比较麻烦。可以将减法变成加法，于是引入反码和补码。

2、反码

正数的反码与原码相同，负数的反码为其绝对值的原码按位取反。

3、补码

正数的补码与原码相同，负数的补码为其绝对值的原码按位取反然后+1，即反码+1。

常用编码

1、顺序二进制编码

就是二进制码，特点是相邻两个数的差值为1

2、格雷码

二进制码0111加1变成1000时有三个位发生了变化，由于各个位变化在时间上的差异可能会短暂的出现1111、1011等其他代码，导致电路状态错误。使用格雷码可以避免这一错误。格雷码又称**循环码**，特点是相邻两个数之间只有一个位不相同。设二进制码B = B(n-1)...B(i+1)B(i)...B(0)，对应格雷码G = G(n-1)...G(i+1)G(i)...G(0)，则有：**G(n-1) = B(n-1)，G(i) = B(i)⊕B(i+1)；B(i) = B(i+1)⊕G(i)**　　⊕ 异或符号

3、独热码

只有一个二进制位为1，其余都为0的编码叫做独热码。独热码常用于时序逻辑电路中状态机的设计。

4、二—十进制编码（BCD码）

十进制数	8421码	2421码	5211码	余3码	余3格雷码
0	0000	0000	0000	0011	0010
1	0001	0001	0001	0100	0110
2	0010	0010	0100	0101	0111
3	0011	0011	0101	0110	0101
4	0100	0100	0111	0111	0100
5	0101	1011	1000	1000	1100
6	0110	1100	1001	1001	1101
7	0111	1101	1100	1010	1111
8	1000	1110	1101	1011	1110
9	1001	1111	1111	1100	1010

5、ASCLL码

卡诺图化简逻辑函数

2^n个最小项合并成一项时，可消去n个变量

数据类型、常量、变量

标识符可以由字母、数字、符号 _ 和 **$** 组成， 但必须以下划线或字母开头。

Verilog HDL 三种数据类型：parameter型、reg型、wire型

parameter常量（符号常量）

```
parameter a = 6, b = 9;  // 合法格式
parameter c = a * 2;  // 非法格式。右边必须为常数表达式，即数字或者先前定义过的符号常量
// 常用参数来定义延迟时间和变量宽度
// 可用字符串表示的任何地方，都可以用定义的参数来代替
// 参数是本地的，其定义只在本模块内有效
// 在模块或实例引用时，课通过参数传递改变在被引用模块或实例中定义的参数
```

整常数的3种表达方式

表达方式	说明	举例
<位宽>'<进制><数字>	完整的表达方式	8'b11000101 或 8'hc5
<进制><数字>	缺省位宽，则位宽由机器系统决定，至少32位	hc5
<数字>	缺省进制为十进制，位宽默认为32位	197

二进制 b/B，十进制 d/D，十六进制 h/H，八进制o/O

x表示不定值，**z**表示高阻值

case语句中 **？** 表示高阻值

模块实例引用时参数的传递

方法一：

```
// defparam语句在编译时可重新定义参数值
module mod(out, ina, inb);  // 被引用模块
    ...
    parameter cycle = 8, real_constant = 2.039, file = "/design/mem_file.dat";
    ...
endmodule
module test;
    ...
    mod mk(out, ina, inb);  // 对模块mod的实例引用
    defparam mk.cycle = 6, mk.file = "../my_mem.dat";  // 参数的传递
    ...
endmodule
// 若不能综合时可用 "#" 号后跟参数的语法来重新定义参数（方法二）
```

方法二：

```
module test;
    ...
    mod # (5, 3.20, "../my_mem.dat") mk(out, ina, inb); // 对模块mod的实例引用，参数必须一一对应
    ...
endmodule
```

变量

三种：网络型（nets）、寄存器型（register）、数组（memory）

nets型变量：输出始终随输入的变化而变化的变量(不保存值)，比如硬连线

常用nets型变量：

- **wire**，tri：连线类型（两者功能一致）

- wor，trior：具有线或特性的连线（两者功能一致）
- wand，triand：具有线与特性的连线（两者功能一致）
- tri1，tri0：上拉电阻和下拉电阻
- supply1，supply0：电源（逻辑1）和地（逻辑0）

> **wire型变量**：最常用的nets型变量，常用来表示以assign语句赋值的组合逻辑信号。**模块中的输入/输出信号类型缺省为wire型。**

```
wire a, b, c;
wire[7:0] d, e, f; // 宽度为8的总线d, e, f
```

register型变量：保存值，（寄存器）（触发器）

常用register型变量：

- **reg**：常代表触发器
- interger：32位带符号整数型变量
- real：64位带符号实数型变量
- time：无符号时间变量

register型变量与nets型变量的根本区别：register型变量需要被明确地赋值，并且在被重新赋值前一直保持原值。

register型变量必须通过过程赋值语句赋值！不能通过assign语句赋值！

在过程块内被赋值的每个信号必须定义成register型！

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

```
reg[4:0] regc, regd; // regc，regd为5位宽的reg型向量
// 用reg型变量生成组合逻辑举例：
module rw1(a, b, out1, out2);
    input a, b;
    output out1, out2;
    reg out1;
    wire out2;
    assign out2 = a;  // 连续赋值语句
    always @(b)  // 电平触发
        out1 <= ~b;  // 过程赋值语句
endmodule
// 用reg型变量生成触发器举例：
module rw2(clk, d, out1, out2);
    input clk, d;
    output out1, out2;
    reg out1;
    wire out2;
    assign out2 = d & ~out1;  // 连续赋值语句
    always @(posedge clk)  // 上升沿触发
        begin
```

```
            out1 <= d;  // 过程赋值语句
        end
endmodule
```

memory型变量（数组）

定义：由若干个相同宽度的reg型向量构成的数组。

- Verilog HDL中通过reg型变量建议数组来对**存储器**建模。
- memory型变量可描述RAM、ROM和reg文件。
- memory型变量通过扩展reg型变量的地址范围来生成。

```
reg[7:0] rega;  // 一个8位的寄存器
reg[7:0] mema [3:0];  // 由4个8位寄存器组成的存储器
rega = 0;  // 合法赋值语句
mema = 0;  // 非法赋值语句
mema[8] = 1;  // 合法
mema[1023:0] = 0;  // 合法，对存储器大范围赋值
```

运算符（基本和C语言一样）

逻辑运算符（&&，||，！）

非0为真（1'b1），0为假（1'b0），不确定的操作数如 4'bxx00 记为（1'bx)。4'bxx11是真，因为他非0

缩减运算符

```
// 符号和位运算一样，但是缩减是单目运算
// &，~&(与非)，|，~|(或非)，^(异或)，~^或^~(同或、异或非)
// 对单个操作数进行递推运算
reg[3:0] a;
b = |a;  // 等效于 b = ((a[0] | a[1]) | a(2)) | a[3])
```

位拼接运算符

```
// 例1
output[3:0] sum;  // 和
output cout;  // 进位输出
input[3:0] ina, inb;
input cin;
assign {cout, sum} = ina + inb + cin;  // sum与cout拼接在一起
// 例2
{a, b[3:0], w, 3'b101} = {a, b[3], b[2], b[1], b[0], w, 1'b1, 1'b0, 1'b1};  // 复杂的赋值
{4{w}}  // 等同于{w, w, w, w}
{b, {3{a, b}}}  // 等同于{b, {a, b}, {a, b}, {a, b}}，也等同于{b, a, b, a, b, a, b}
```

赋值语句和块语句

连续赋值语句

assign语句：用于对wire型变量赋值。例：

```
wire a, b, c;
assign c = a & b;
```

过程赋值语句

非阻塞（non - blocking）赋值方法：赋值符号为 <= ，如 b <= a;

阻塞（blocking）赋值方法：赋值符号为 = ，如 b = a;

```
wire a;
reg b, c;
always @(posedge clk)
    begin  // begin...end就是块语句
        b <= a;  //非阻塞赋值语句
        c <= b;  //这两句是并行执行的，所以本次上升沿时钟信号到来时，a的值赋给了b，b原来的值赋给了c
        // 这两句改成b = a; c = b; 的话b和c的值就相等
    end
```

条件语句

if-else语句C语言中的 { } 换成begin...end

always块语句和assign语句是并行执行的

case语句：

- **case**语句中，分支表达式每一位的值都是确定的0或1
- **casez**语句中，若分支表达式某些位的值为高阻值z，则不考虑对这些位的比较
- **case**语句中，若分支表达式某些位的值为高阻值z或不定值x，则不考虑对这些位的比较
- 在分支表达式中，可用？来表识x或z

```
// 例：(啥也不是，就看case格式)
output out;
input a, b, c, d;
input[3:0] select;
reg out;
always @(select[3:0] or a or b or c or d)
    begin
        module mux_z(out, a, b, c, d, select);
            casez(select)
                4'b???1: out = a;
                4'b??1?: out = b;
                4'b?1??: out = c;
                4'b1???: out = d;
            endcase
        endmodule
    end
```

for语句和**while**语句跟C语言一样，改一下begin...end就行

repeat语句

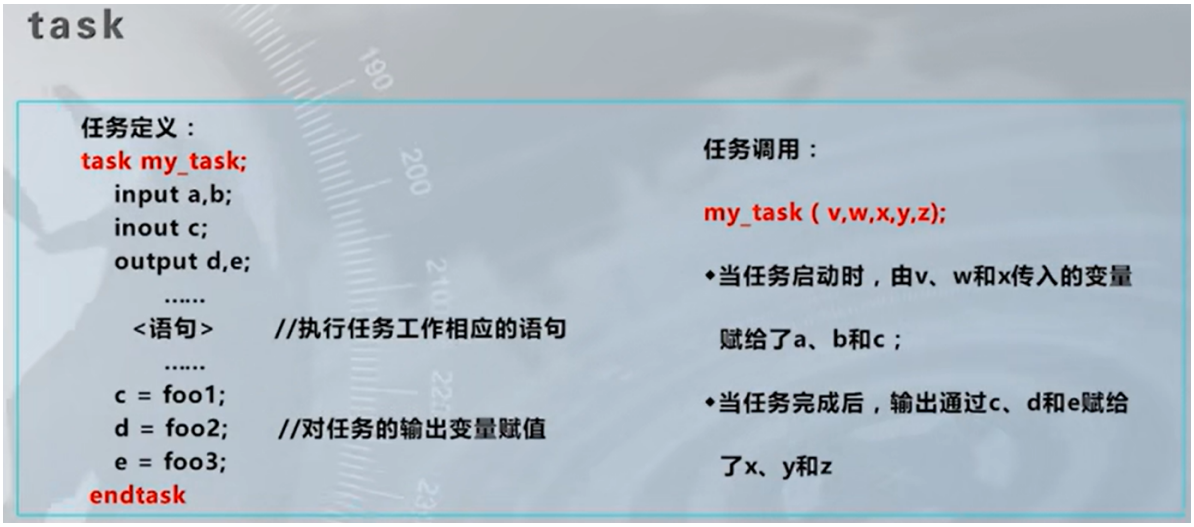
```
repeat(count)  // 括号里是执行次数
    begin
        ...  // 块内语句重复执行count次
    end
```

结构说明语句

initial说明语句：只执行一次（用来初始化）

always说明语句：当条件满足时重复执行

task说明语句：可在程序模块中的一处或多处调用（只能在同一模块内定义与调用）（无返回值）

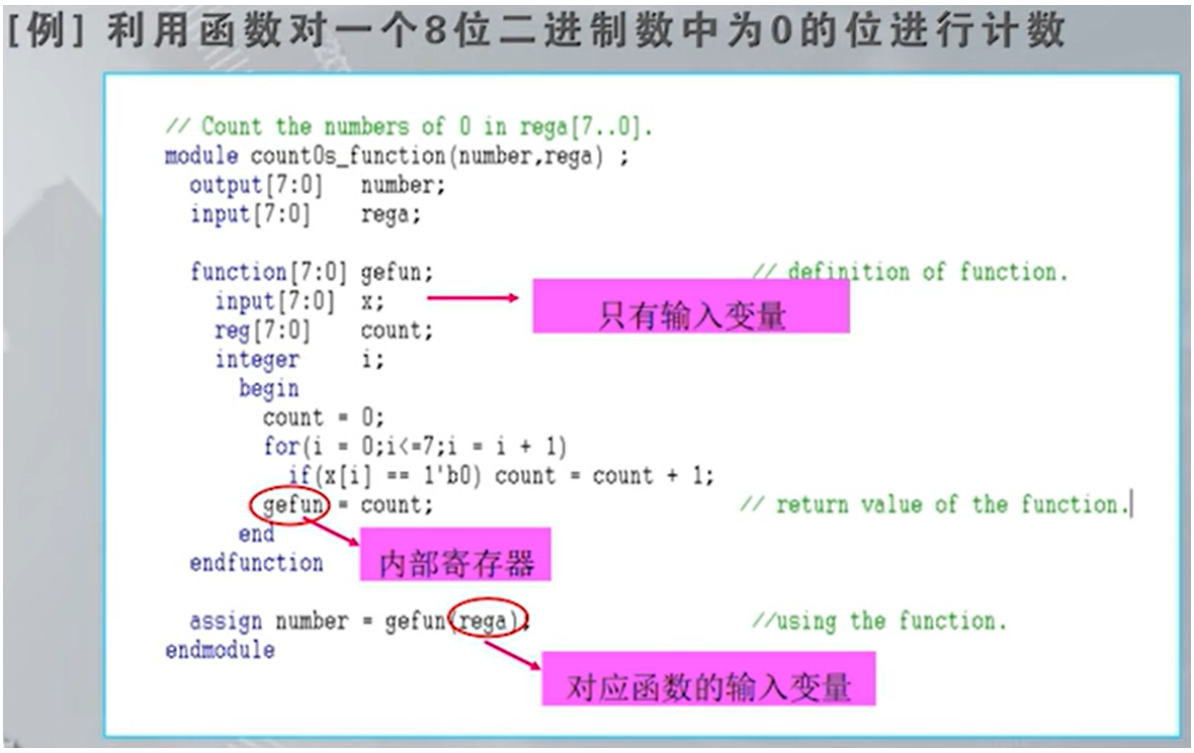


function说明语句：可在程序模块中的一处或多处调用（有返回值）

（在模块内部定义，通常在本模块内调用，也能根据按模块层次分级命名的函数名从其他模块调用）

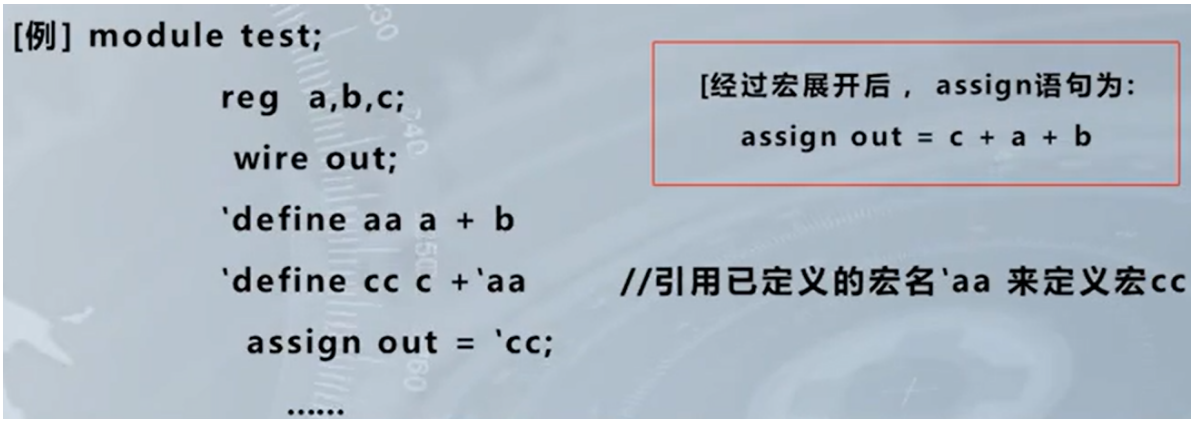
函数使用规则：

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



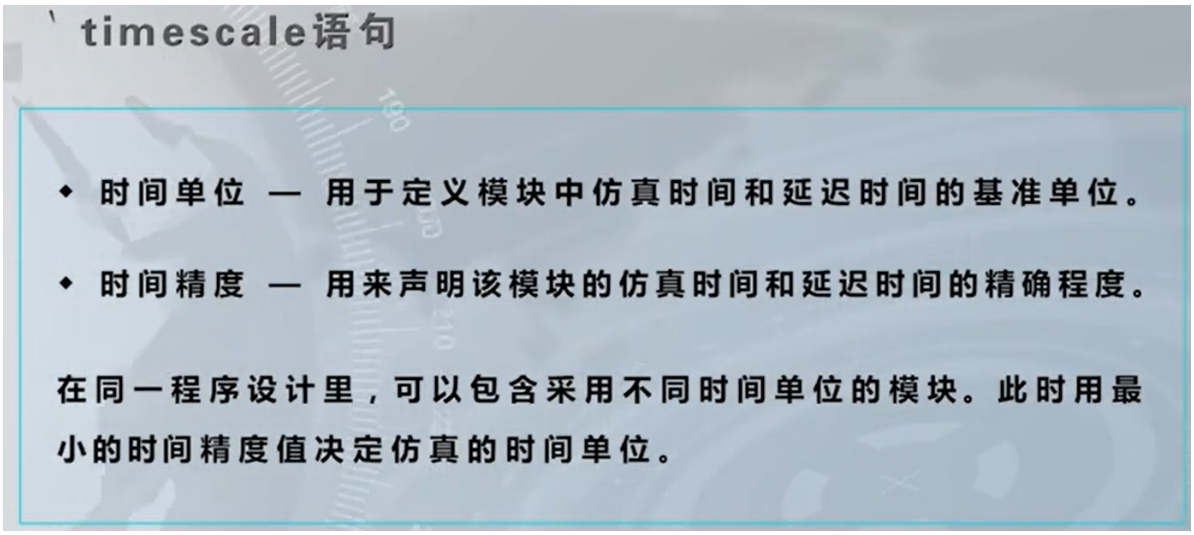
编译预处理语句

`define语句



`include语句（双引号包含被引用文件名）

`timescale语句




```
`timescale 1ps/1ns  // 非法！时间精度值不能大于时间单位值
`timescale 1ns/1ps  // 合法
```