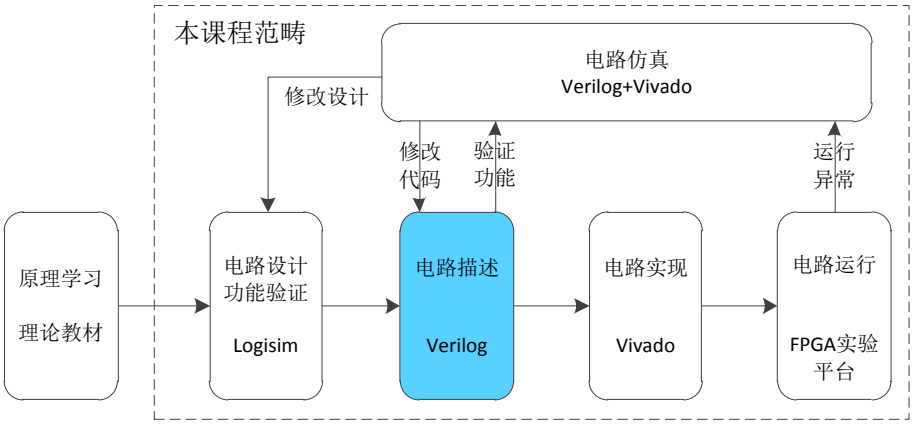
实验 04_Verilog 硬件描述语言

简介

下图以设计数字电路的一般流程为例，说明了电路设计过程中的关键步骤以及相关工具。



在简单组合逻辑和时序逻辑的实验中已经对 Verilog 语法进行了简单的介绍，用户应该已经可以阅读和编写简单的 Verilog 代码，本次实验中，我们将进一步系统的学习 Verilog 代码。

实验目的

掌握 Verilog HDL 常用语法

能够熟练阅读并理解 Verilog 代码

能够设计较复杂的数字功能电路

能够将 Verilog 代码与实际硬件相对应

实验环境

PC 一台

Windows 或 Linux 操作系统

Java 运行环境（jre）

Logisim 仿真工具

vlab.ustc.edu.cn（jre、Logisim 工具以及 Verilog 语法介绍都可在此网站获取）

实验步骤

Step1：Verilog 关键字

Verilog 语法中定义了 120 多个关键字，但常用的实际上很少，作为初学者只需要使用其中的十几个，便可以完成绝大多数的电路设计，这些关键字包括：module/endmodule、input、output、wire、reg、assign、always、initial、begin/end、posedge、negedge、if、else、case、endcase。

下面我们对这些常用关键字逐一进行介绍。

module/endmodule：这两个关键字用于表示模块的开始和结束，必须成对出现，在前面的实验中已经有过多次接触，此处不再赘述。

```
module test();
    //code
endmodule
```

input:表明端口类型为输入信号，该关键字一般用在模块的端口定义部分。

output:表明端口类型为输出信号，该关键字一般用在模块的端口定义部分。

```
module test(
input    a,b,clk,
output   o);
    //code
endmodule
```

wire：表明数据类型为线型，该关键字用在端口或内部信号的定义部分。一般来说，凡是通过 assign 关键字进行赋值的信号，都应该定

义成 wire 类型。wire 是 Verilog 的默认数据类型，对于没有显式声明的信号，一律默认为 wire 类型。

reg：表明数据类型为寄存器类型，这是与 wire 相对应的一种数据类型，该关键字用在端口或内部信号的定义部分。一般来说，凡是在 always 语句语句内部被赋值的信号，都应该被定义成 reg 类型。在 Verilog 语法内，除了 wire 和 reg，还支持多种其它的数据类型，但用的都不多，读者现在只需掌握这两种数据类型即可。

```
module test(
input wire   a,b,clk,
output reg   o);
wire   s;
   //code
endmodule
```

assign：连续赋值语句关键字，此类语句将逻辑表达式的值赋给线网类型信号。一个模块内部可以有任意多个 assign 语句,但每个 assign 语句只能包含一个连续赋值表达式。

always：过程赋值语句关键字，次类语句将逻辑表达式的值赋给寄存器类型信号。一个模块内部可以有任意多个 always 语句，其一般格式为：

always (时序控制) 过程语句

always 语句是永远在循环执行的，其中过程语句一般为一条表达式赋值语句，或者 begin/end 关键字构成的顺序过程语句块。例如“always clk = ~clk; ”always 后的语句重复执行，由于没有时序控制语句，将在 0 时刻无线循环，这种写法没有语法错误，但没有实际意义,因此 always 语句的执行必须带有时序控制,对于上述语句，可以改成“always #5 clk = ~clk; ”该语句将产生周期为 10 个时

间单位的波形（#n 表示 n 个时间单位的延时）。一般来说 Verilog 语法中的 always 语句写成如下格式：

```
always@（敏感变量列表）
    过程语句
```

只有当敏感变量列表中的事件发生时，才会执行过程语句。always 语句既可以用来实现组合逻辑赋值，也可以用来实现时序逻辑赋值，但在同一 always 语句块内两者不可同时出现，另外敏感变量列表中的事件也不一样，实现组合逻辑赋值应是电平敏感事件，时序逻辑则应该是边沿敏感事件，电平敏感事件和边沿敏感事件不能同时出现在敏感变量列表中。用 always 实现组合逻辑，可统一采用“always@(*)”的写法，而不用一一列举敏感事件。

```
module test(
input wire   a,b,
output reg   o);
wire   s;
    assign s = a & b;
    always@( * ) o = s;// *表示任意时序控制
endmodule
```

begin/end：顺序过程语句，必须成对出现，在 begin/end 中间可以有多条语句，这些语句是顺序执行的，该关键字允许嵌套，即 begin/end 内部可以包含其他的 begin/end 语句块。

posedge：该关键字后面跟信号名，表示该信号的上升沿这一事件。一般用在 always 语句的时序控制部分。

negedge：该关键字后面跟信号名，表示该信号的下降沿这一事件。一般用在 always 语句的时序控制部分。

```
module test(
input wire   a,b,clk,
output reg   o);
```

```
wire   s;
   assign s = a & b;
   always@( posedge clk ) o <= s;
endmodule
```

if：条件语句，其后跟条件判别语句，以及过程语句，有时候会与 else 语句配合使用。

else：if 语句的分支，if、else 用于实现带有优先级的条件分支，一般出现在 always 语句的过程语句部分，而不能直接在模块内部单独出现。一般用法如下：

```
  if(条件) 过程语句
  else  过程语句
module test(
input wire   a,b,clk,
output reg   o);
   always@( posedge clk )
   begin
      if(a)  o <= a;
      else   o <= b;
   end
endmodule
```

case/endcase:具有相同优先级的多路条件分支，两个关键字必须成对出现。一般出现在 always 的过程语句部分，而不能在模块内部单独出现。一般用法如下：

```
case（case 表达式）
   case 条目表达式 1：过程语句
   case 条目表达式 2：过程语句
   ...
   [default:过程语句]
endcase

module test(
input wire   a,b,clk,
output reg   o);
   always@( posedge clk )
      case({a,b})
         2'b00:  o <= 1'b0;
```

```
            2'b01:o <= 1'b0;
            2'b10:o <= 1'b0;
            2'b11:o <= 1'b1;
        endcase
endmodule
```

Step2：Verilog 代码基本结构

通过前面的学习读者已经对 Verilog 代码的结构有了初步的了解，现在我们再系统的总结一下。

```
module 模块名（
输入端口定义          //输入端口只能是 wire 类型
输出端口定义          //输入信号可根据需要定义成 wire 或 reg 类型
）；
    内部线信号定义 //内部信号可根据需要定义成 wire 或 reg 类型
    模块实例化       //实例化的输出端只能接 wire 类型信号
    assign 连续赋值语句
    always 过程语句
endmodule
```

第一：每个模块都是由关键字 module 开头，由 endmodule 结束。

第二：每个模块都应该有一个唯一的模块名，模块名不能使用 Verilog 语法的关键字。

第三：模块名后面的括号内是对输入输出信号的定义，除后面实验中要讲到的仿真文件外，任何能实际工作的模块都应该有输入和输出信号。

第四：模块主体部分只能出现四类语句（仿真文件中会用到的 initial 语句等暂不考虑）：内部信号定义、模块实例化、assign 语句、always 语句，每类语句的数量不受限制。

Step3：Verilog 数据及类型

在 Verilog 中，有四种基本的值：0,1,x,z。其中“0”表示逻辑

0 或者假，“1”表示逻辑 1 或者真，“x”表示未知状态，“z”表示高阻状态。这四种值与电路的真实状态相对应，如未被初始化的信号为“x”未知状态，输出使能关闭的三态门为“z”高阻状态。

Verilog 中的常量有三种：整数、实数、字符串，其中整数使用最为广泛。整数有两种书写方式：简单的十进制格式、基数格式。十进制格式与日常数字写法一样，如：32，-15 等；基数格式的一般写法为：位宽 ’ 进制 数值。进制有二进制（b 或 B）、八进制（o 或 O）、十进制（d 或 D）和十六进制（h 或 H）四种。如 2’b11 表示 2bit 位宽的数字，其二进制数值为 11，即十进制的 3；8’h3F 表示 8bit 位宽的数字，其十六进制数值为 3F，即十进制的 63。此外，数字中佳可以插入下划线以增加可读性，如 6’b10_1100（即 44）。

Verilog 中有两种常用的数据类型，即 wire（线网类型）和 reg（寄存器类型），关于两种数据类型的使用只需要遵循以下规则即可：凡是通过 assign 语句赋值的信号（一定是组合逻辑赋值信号），都应定义成 wire 类型，凡是在 always 语句中赋值的信号（可能是组合逻辑赋值信号、也可能是时序逻辑赋值信号），都应定义成 reg 类型。

Step4：Verilog 操作符

Verilog 语法中的操作符，按照其功能可以分为：算数操作符、关系操作符、逻辑操作符、归约操作符、条件操作符、移位操作符、拼接操作符。

算数运算符：该类操作符包含+（加）、-（减）、*（乘）、/（除）、%（取模）五种，a + 3’b101 表示两个数“a”和“3’b101”相加。

关系运算符：该类操作符包含>（大于）、<（小于）、>=（大于等于）、<=（小于等于）、==（等于）、!=（不等于）等几类，如比较结果为真，则表达式值为 1，否则为 0。如 4'h1>=a,当整数 a 为 0 或 1 时，为真，否则为假。

逻辑操作符：该类操作符包括&&（逻辑与）||（逻辑或）！（逻辑非）~（按位取反）&（按位与）|（按位或）^（按位异或）~^（按位同或），例如 a&&0 表达式值为 0，3'b101 | 3'b010 值为 3'b111。

归约操作符：该类操作符包括&（归约与）、~&（归约与非）、|（归约或）、~|（归约或非）、^（归约异或）、~^（归约同或）。这种操作符在单一操作数的所有位上操作，最终结果为 1bit 的 0 或 1。例如&2'b01 结果为 0，|4'b0010 结果为 1。

条件操作符：该类操作符只有一个，是一个三目操作符，其一般形式为“判断表达式？ 表达式 1 ：表达式 2”，如果“判断表达式”为真，则选择表达式 1，否则选择表达式 2。

移位操作符：包含<<（左移）和>>（右移）两种，两种移位均为逻辑移位，空位都补零,如 2'b11<<1 结果为 2'b10。

拼接操作符：该操作用于将多个表达式合并成一个表达式，一般形式为{表达式 1，表达式 2,....,表达式 N}。

如 a[7:4]={a[0],a[1],a[2],a[3]}，表示将 a 信号低 4 颠倒并赋值给高 4 位。b[31:0]={24{a[7]},a[7:0]}，表示将 a 信号进行符号扩展到 32 为，并赋值给 b 信号。

Step5：Verilog 表达式

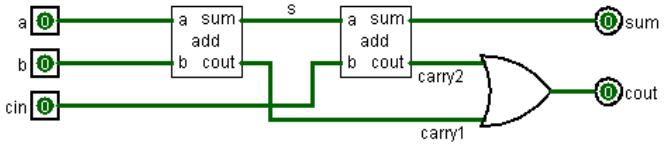
Verilog 中的表达式由操作数和操作符组成，可以在出现数值的任何地方使用。表达式可以是以下类型：常数、参数、线网、寄存器、位选择、部分选择、存储器单元、函数调用。

表达式都有一个值，因此可以将其赋给 wire 或 reg 类型的信号，也可以用在逻辑判断语句（如 if、case）中。如 assign a = 表达式 1、always@（*） if(表达式)... else ...。

Step6：模块调用

一个模块能够在另一个模块中被引用，这样就建立了描述的层次，使得设计大规模复杂电路的效率大大提高。模块实例语句的形式为：模块名　实例化名（端口关联）；

端口信号可以通过位置或名称进行关联，但两种关联方式不能混用。通过位置关联的格式为：端口表达式，通过名称关联的格式为：.端口名称（端口表达式）。下面以全加器中调用两个半加器的设计为例来对比一下两种关联方式的区别。




```
module add(
input a,b,
output sum,cout);
   //模块主体
endmodule

module full_add(
input a,b,cin,
output sum,cout);
wire s,carry1,carry2;
add add_inst1(a,b,s,carry1);//通过位置关联
add add_inst2(.a(s),.b(cin),.sum(sum),.cout(cout));//通过名称关联
```

```
endmodule
```

可以看出，两种方式各有优势，通过位置关联写法更简洁，通过名称关联可读性更强，在本课程的学习中，我们强烈推荐读者使用第二种方式，即通过名称关联。

Step7：代码实例

```
8bit 位宽 4 选 1 选择器，纯组合逻辑
module mux_4to1(          //8bit 位宽的 4 选 1 选择器
input [7:0] a,b,c,d,
input [1:0] sel,
output reg [7:0] o);      //always 语句内赋值的信号都应定义成 reg 类型
always@(*)                //always 语句内实现组合逻辑
begin
    case(sel)
        2'b00: o = a;     //组合逻辑使用“=”进行赋值
        2'b01: o = b;
        2'b10: o = c;
        2'b11: o = d;
       default: o = 8'h0;//用 case 语句实现组合逻辑时一定要有 default
    endcase
end
endmodule
================================
1~10 循环计数的计数器
module cnt_1to10(
input clk,rst_n,
output reg [3:0] cnt);
always@(posedge clk or negedge rst_n)
//时序控制条件为时钟上升沿和复位的下降沿
begin
    if(!rst_n)        //复位信号优先级最高，应是第一个判断的条件
        cnt <= 4'h1;
   else if(cnt>=10)
        cnt <= 4'h1;
    else
        cnt <= cnt + 4'h1;
end
endmodule
```

实验练习

题目 1. 阅读以下 Verilog 代码，找出其语法错误，并进行修改

```
module test(
input a,
output b);
    if(a) b = 1'b0;
    else  b = 1'b1;
endmodule
```

题目 2. 阅读以下 Verilog 代码，将空白部分补充完整

```
module test(

input [4:0] a,

_____________);

always@(*)

   b = a;

_____________
```

题目 3. 阅读以下 Verilog 代码，写出当 a = 8'b0011_0011, b = 8'b1111_0000 时各输出信号的值。

```
module test(
input  [7:0] a,b,
output [7:0] c,d,e,f,g,h,i,j,k );
  assign c = a & b;
  assign d = a | b;
  assign e = a ^ b;
  assign f = ~a;
  assign g = {a[3:0],b[3:0]};
  assign h = a >> 3;
  assign i = &b;
  assign j = (a > b) ? a : b;
  assign k = a - b;
endmodule
```

题目 4. 阅读以下 Verilog 代码，找出代码中的语法错误，并修改

```
module sub_test(
input a,b,
```

```
output reg c);
    assign c = (a<b)? a : b;
endmodule
module test(
input a,b,c,
output o);
  reg temp;
  sub_test(.a(a),.b(b),temp);
  sub_test(temp,c,.c(o));
endmodule
```

题目 5. 阅读以下 Verilog 代码，找出其中的语法错误，说明错误原因,并进行修改。

```
module sub_test(
input  a,b);
output o;
assign o = a + b;
endmodule
module test(
input a,b,
output c);
always@(*)
begin
    sub_test sub_test(a,b,c);
end
endmodule
```

总结与思考

1.请总结本次实验的收获

2.请评价本次实验的难易程度

3.请评价本次实验的任务量

4.请为本次实验提供改进建议