Verilog语言学习

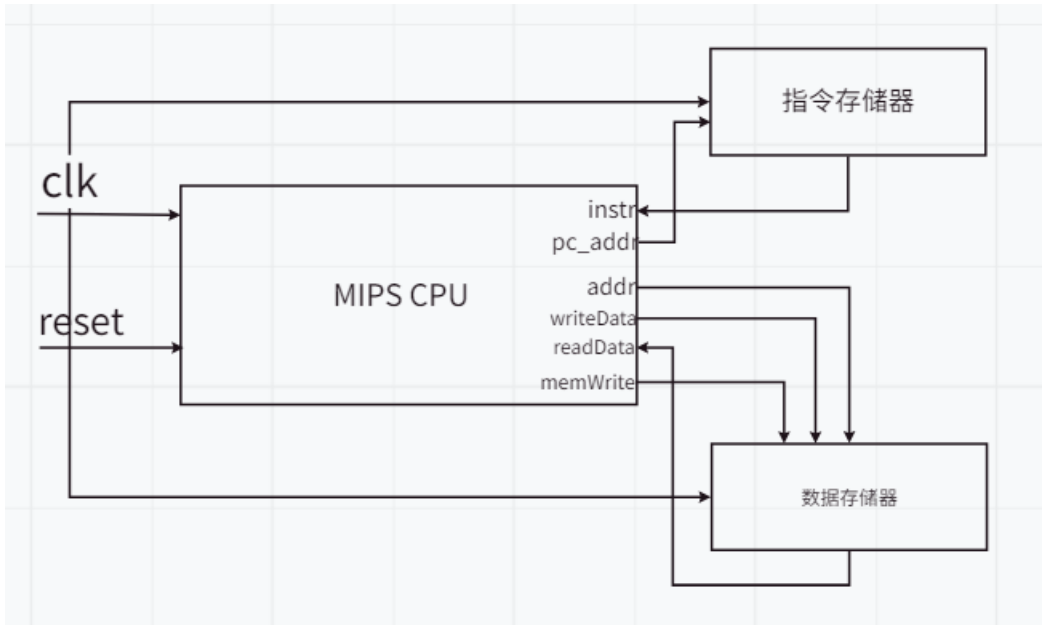
https://www.runoob.com/w3cnote/verilog-tutorial.html书签：1.1 Verilog 教程 | 菜鸟教程

Vivado设计方法——自顶向下

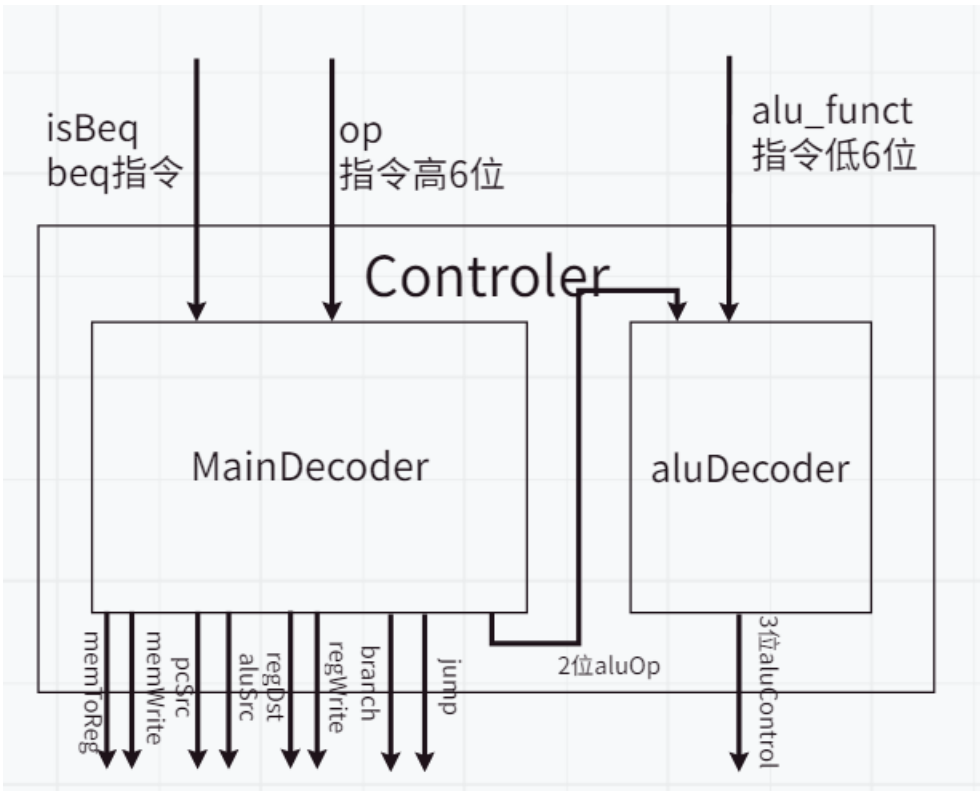
自顶向下的思想是“**先定义整个系统的顶层模块能，根据顶层模块的功能分析构成顶层模块的必要子模块；然后进一步对各个模块进行分解、设计，直到到达无法进一步分解的底层功能块**”

以之后要进行设计的MIPS CPU设计为例：

1. 顶层模块top需要实现“访指令存储器”、“访数据存储器”、“MIPS CPU处理的功能”，其结构如下图。其中的指令存储器和数据存储器已不可再分，CPU模块可再分为控制器模块controler和数据通路模块datapath



2. 控制器模块是基于**指令的高6位和低6位**数据进行译码的。其编码规则是将指令的高6位译码转化为一系列的控制信号，其中包含两位的alu控制信号，这些控制信号结合指令的低6位译码，产生控制alu运算的控制信号



3. 数据通路模块则是MIPS指令的执行过程，具体包括了：pc触发器、加法器、寄存器、符号扩展、逻辑左移两位、二路选择器、alu模块（流水线CPU还包括各种输入控制信号的触发器、阻塞模块、比较器、三路选择器）

数据通路框架图如下：

单周期无流水CPU数据通路框架图：

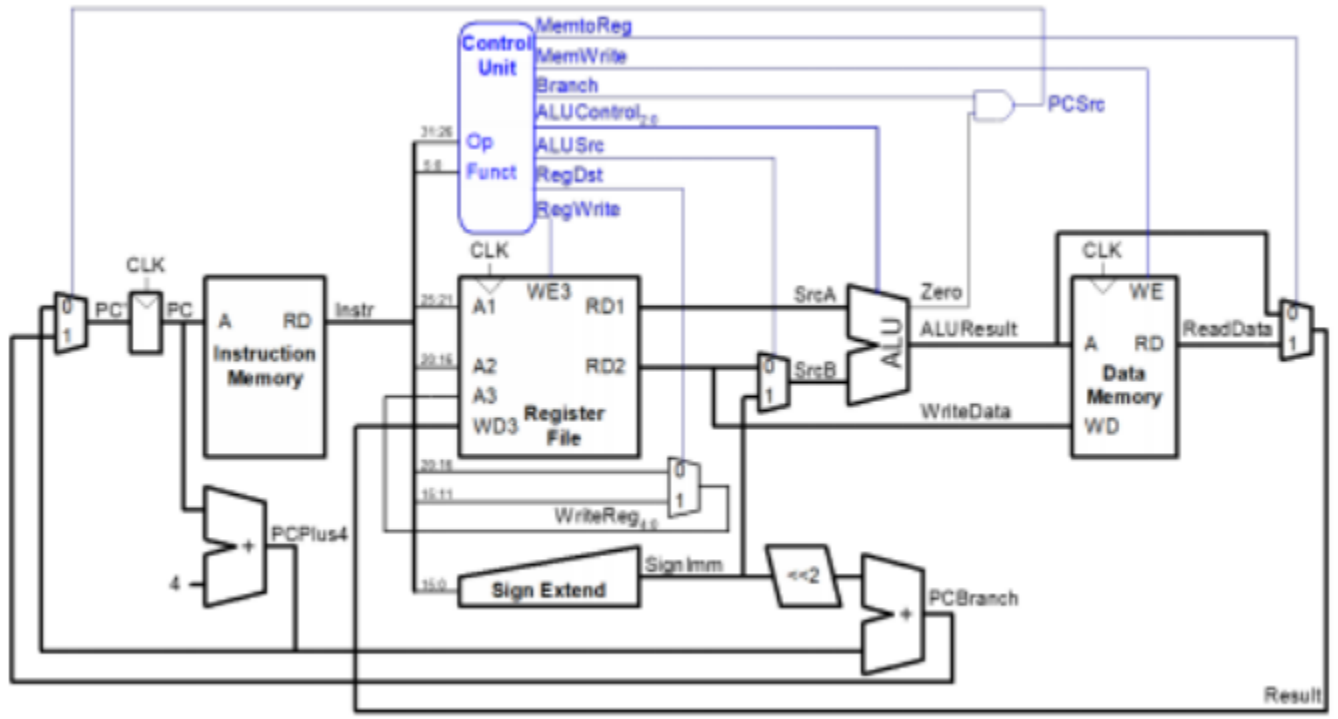
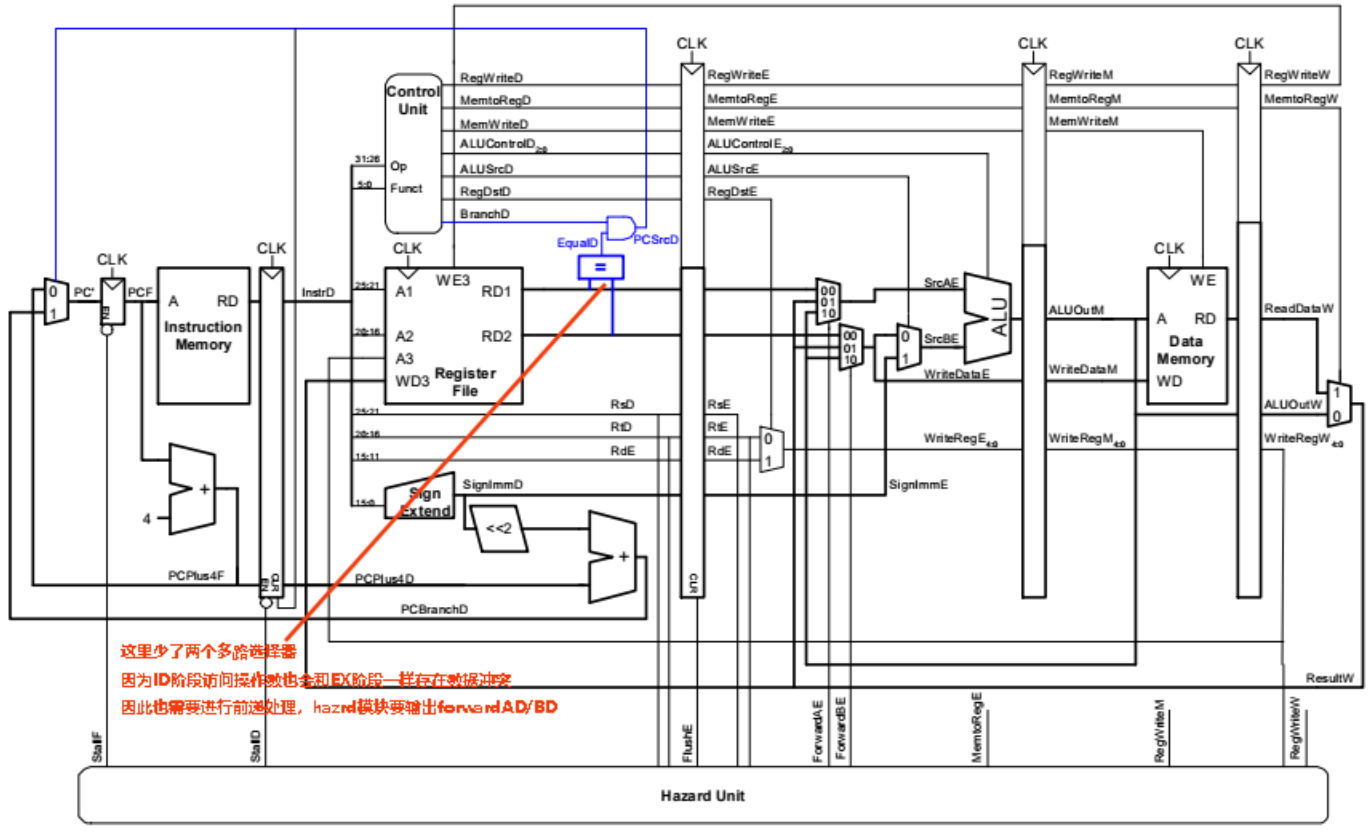


图 2: 单周期 CPU 框架图

五段流水线CPU数据通路框架图：



基础语法

✨Verilog语句是区分大小写的，每个语句必须以“;”结束

数值表示

- 电平逻辑

  0、1、x、z

- 整数数值

  数字声明时，合法的基数格式有 4 种，包括：**十进制('d 或 'D)，十六进制('h 或 'H)，二进制（'b 或 'B），八进制（'o 或 'O）**

  即一个数：*符号*±*space位宽space基数*{*'b,'o,'h,'d*}*space数值*

  不指明基数时默认十进制，不指明位宽时默认按照编译器自动分频位宽

- 实数数值：可以用十进制表示，也可以用科学计数法表示e/E
- 字符串：用""括起，字符被视为ASCII字符（一个字符8bit）

```Verilog
reg [0: 14*8-1]        str ;//需要14*8bit存储str
initial begin
    str = "www.runoob.com";//14个字符
end
```

数据类型

- wire

  wire 类型表示硬件单元之间的物理连线，由其连接的器件输出端连续驱动。**如果没有驱动元件连接到 wire 型变量，缺省值一般为 "Z"**

- reg

  寄存器（reg）用来表示存储单元，它会保持数据原有的值，直到被改写

  在 always 块中，寄存器可能被综合成边沿触发器；在组合逻辑always@(*)中可能被综合成 wire 型变量

- **向量：位宽大于1的reg、wire**

  "wire/reg [beg:end]/[end:beg] 变量名;"这里的是闭区间的beg、end

  Verillog 还支持指定 bit 位后固定位宽的向量域选择访问

  - **[bit+: width]** ：从起始 bit 位开始递增，位宽为 width

    [bit-: width] ：从起始 bit 位开始递减，位宽为 width

```
//下面 2 种赋值是等效的
A = data1[31-: 8] ;
A = data1[31:24] ;

//下面 2 种赋值是等效的
B = data1[0+ : 8] ;
B = data1[0:7] ;
```

Verilog

对信号重新组合成新的向量时，需要用到大括号{}

```
wire [31:0]   temp1, temp2 ;
assign temp1 = {byte1[0][7:0], data1[31:8]};  //数据拼接
assign temp2 = {32{1'b0}};  //赋值32位的数值0
```

Verilog

> 之前提到的字符串字符串：用""括起，字符被视为ASCII字符（一个字符8bit），就是一个向量

- 数组

变量名[beg:end]，同样是闭区间的beg、end

✍向量是一个单独的元件，位宽为 n;

数组由多个元件组成，其中每个元件的位宽为 n 或 1

它们在结构的定义上就有所区别：**向量是 [beg,end]变量名；数组是变量名[beg,end];**

> **reg数组就是mem**

- integer实质上属于reg

整数类型用关键字 integer 来声明，声明时不用指明位宽，**位宽和编译器有关，一般为32 bit**

✍integer变量实质上属于reg变量，不同的是：**reg 型变量为无符号数，而 integer 型变量为有符号数**

- real实质上属于reg

实数用关键字 real 来声明，可用十进制或科学计数法来表示

实数声明不用指定位宽（同integer），默认值为 0

- time实质上属于reg

使用特殊的**时间寄存器 time 型变量，对仿真时间进行保存。其宽度一般为 64 bit**，通过调用系统函数 **$time** 获取当前仿真时间

```
time     current_time ;
initial begin
    #100 ;
    current_time = $time ; //current_time 的大小为 100
end
```
Verilog

✍也可以声明reg去存储时间，但是需要开发者按照时间范围设置好位宽

- parameter常量

参数用来表示常量，用关键字 parameter 声明，只能赋值一次

✍本质上的类型只有寄存器型(reg)、线网型(wire等)变量类型

integer、real、time本质上都是reg，声明时均不需要指定位宽

mem是reg数组，字符串是reg/wire向量

parameter是常量（私以为本质也是reg）

表达式

✍always块内的赋值对象不能够是wire型[1]

wire型是assign语句左侧的唯一合法类型（连续赋值）

- **条件运算符"? : "是自右向左关联，其余运算符都是自左向右关联**
- **<<、>>是逻辑，<<<、>>>是算术**
- 连接运算符{}

```
A = 4'b1010 ;
B = 1'b1 ;
Y1 = {B, A[3:2], A[0], 4'h3 }; //结果为Y1='b1100_0011
```

```
Y2 = {4{B}, 3'd4};  //结果为 Y2=7'b111_1100

Y3 = {32{1'b0}};  //结果为 Y3=32h0，常用作寄存器初始化时匹配位宽的赋初值
```
Verilog

要注意Y3是{个数{位}}

- 如果操作数某一位为 X，则计算结果也会全部出现 X
- **对变量进行声明时，要根据变量的操作符对变量的位宽进行合理声明，不要让结果溢出**

> 相加的 2 个变量位宽为 4bit，那么结果寄存器变量位宽最少为 5bit。否则，高位将被截断，导致结果高位丢失。

无符号数乘法时，结果变量位宽应该为 2 个操作数位宽之和

除法结果商变量位宽是被除数位宽，余数位宽是除数位宽

- **归约操作符：对操作数的逐位进行运算，最终只产生1bit的结果**

&、~&、|、~|、^、~^

```
wire [3:0]a=4'b1011;
wire b=&a;//b=0
//~&a=1 |a=1 ~|a=0 ^a=1 ~^a=0
```
Verilog

编译指令

以**反引号 ` 开始的某些标识符**是 Verilog 系统编译指令，编译指令为 Verilog 代码的撰写、编译、调试等提供了极大的便利

使用频率较高的4个：

1. **`define,`undef,`ifdef,`elsif,`else**

   相当于C语言中的#define，定义宏，该宏在这次编译中均有作用

```
`define    DATA_DW  32
`define    S     $stop;   //用`S来代替系统函数$stop;(包括分号)
`define    WORD_DEF   reg [31:0]   //可以用`WORD_DEF来声明32bit寄存器变量
```
Verilog

2. `include

   使用 `include 可以在编译时将一个 Verilog 文件内嵌到另一个 Verilog 文件中，作用类似于 C 语言中的 #include 结构。该指令通常用于**将全局或公用的头文件包含在设计文件**里

3. `timescale

   `timescale time_unit/time_precision，时间精度大小（保留到精度的后一位）不能超过时间单位

连续赋值——对wire型赋值（构建连线）

对wire型变量赋值相当于建立逻辑电路，因此**wire型的赋值只能赋值一次**。也因为这种逻辑电路的性质，只要赋值右边的值发生变化时，就会更新该wire型变量（若干根线）

除了使用assign对wire赋值外，在wire变量的声明时，也可以赋值

```Verilog
wire [31:0]A=32'b1;
wire [31:0]B;

assign B=32'b2;
```

✍一位全加器的代码实现：

$$So = Ai \oplus Bi \oplus Ci$$

$$Co = AiBi + Ci(Ai + Bi)$$

```Verilog
module full_adder1(
  input Ai,Bi,Ci,
  output So,Co);

  assign So=Ai^Bi^Ci;
  assign Co=(Ai&Bi)|(Ci&(Ai|Bi));
endmodule
```

过程结构initial、always

一个模块中可以包含**多个 initial 和 always 语句**，但 **2 种语句不能嵌套使用**

这些语句在模块间**并行执行**，与其在模块的前后顺序没有关系

但是 **initial 语句或 always 语句内部可以理解为是顺序执行的**（非阻塞赋值除外）

Verilog代码模块间并行，模块内串行

每个 initial 语句或 always 语句都会产生一个独立的控制流，**执行时间都是从 0 时刻开始**

- initial结构

  initial 结构语句从 0 时刻开始执行，只执行一次

- always结构

  always 语句是重复执行的：always 语句块从 0 时刻开始执行其中的行为语句；当执行完最后一条语句后，便再次执行语句块中的第一条语句，如此循环反复

- 过程赋值

1. 区分连续性赋值assign和过程性赋值

   连续性赋值总是处于激活状态，任何操作数的改变都会影响表达式的结果；

   过程赋值只有在语句执行的时候，才会起作用

2. 过程性赋值包括阻塞赋值和非阻塞赋值两种，是对寄存器等类型的赋值

   阻塞赋值：属于顺序执行，即下一条语句执行前，当前语句一定会执行完毕，**用等号 = 作为赋值符**

   非阻塞赋值：属于并行执行语句，即下一条语句的执行和当前语句的执行是同时进行的，它不会阻塞位于同一个语句块中后面语句的执行，**用<=作为赋值符**（**右端仍用的旧值**）

   下列的代码可以实现两个值的同时交换（因为运算过程中仍用的旧值）

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

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

✍在设计电路时，always 时序逻辑块中多用非阻塞赋值<=，always 组合逻辑块中多用阻塞赋值=

在仿真电路时，initial 块中一般多用阻塞赋值=

条件语句与分支语句

- 条件语句

```
if (condition1)   begin    true_statement1 ; end
else if (condition2)    begin    true_statement2 ; end
else if (condition3)  begin      true_statement3 ; end
else   begin                  default_statement ; end
```

Verilog

- 多路分支语句

```
case(case_expr)
    condition1     :   begin          true_statement1 ; end
    condition2     :      begin        true_statement2 ;end
    ……
    default        :      begin        default_statement ;end
endcase
//多个condition处理相同，用，隔开
/*
condition1,condition2:begin statement end
*/
```

Verilog

循环语句

- while循环

```
while (condition) begin
    …
end
```

Verilog

- for循环

```
for(initial_assignment; condition ; step_assignment)  begin
    …
end
```
Verilog

- repeat循环

```
repeat (loop_times) begin
    …
end
```
Verilog

loop_times是个常量/数字

- forever循环

```
forever begin
    …
end
```
Verilog

forever 语句表示永久循环，相当于 while(1) ，不包含任何条件表达式，一旦执行便无限的执行下去，系统函数 $finish 可退出 forever

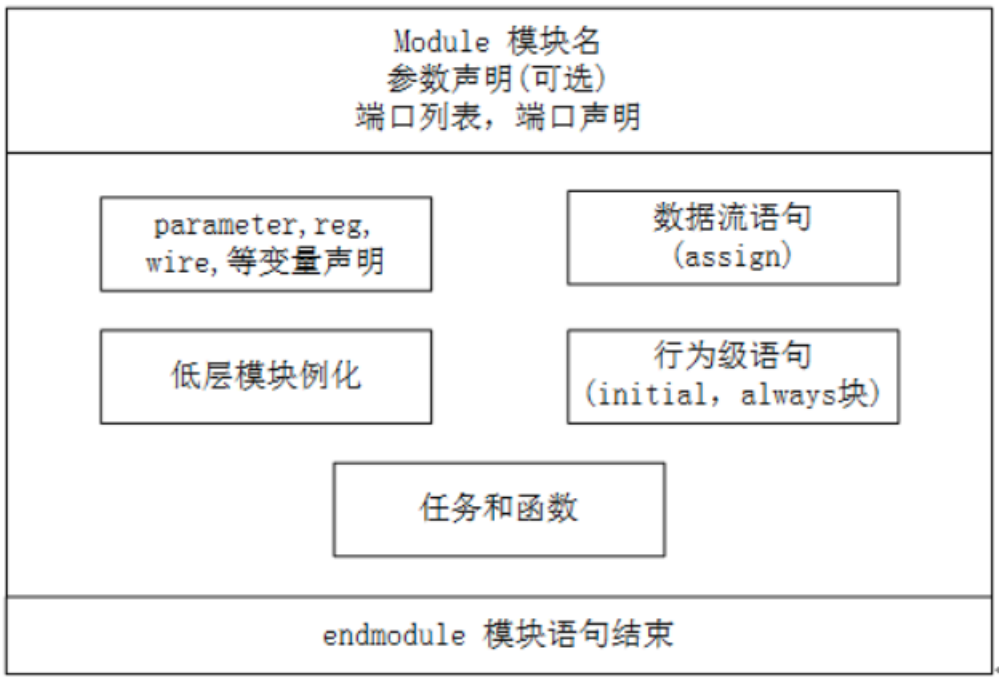
通常，forever 循环是和时序控制结构配合使用的，如下

```
reg clk;
initial begin
    clk=1;//先写后读，clk下降沿写上升沿读
end
forever begin
    #10;
    clk=0;
```

```
end
//或者循环中使用$time大于多少时调用$finish退出循环
```
Verilog

模块与端口

模块格式的定义如下：



模块内各个部分是并行的，声明的顺序只要求保证“**变量先声明再使用**”即可

模块的端口有三种类型：input、output和inout

✍定义模块时input/inout必须是wire，output是wire或reg

但是外部实例化时，input/inout是wire或reg，但output必须是wire

在 Verilog 中，**模块端口默认的声明为 wire 型变量，即当端口具有 wire 属性时，不用在实体中再次声明端口类型为 wire 型。但是，当端口有 reg 属性时，则实体中的reg 声明不可省略**

- 模块的例化

  命名例化：module名 变量名(.端口名 (测试变量名),,,,);不用模块定义在乎顺序

  顺序例化：module名 变量名(按端口定义顺序和向量大小，按序放变量);

- 使用generate进行多个相同模块的例化
  - generate的使用

    1. 声明genvar变量
    2. generate for循环 begin:名字

    end

    endgenerate

✍begin后的名字必须要有

generate for循环体内，需要用assign或always

重复例化4个1bit全加器组成4bit全加器

```
module full_adder4(
    input [3:0]   a ,   //adder1
    input [3:0]   b ,   //adder2
    input         c ,   //input carry bit

    output [3:0]  so ,  //adding result
    output        co    //output carry bit
    );

    wire [3:0]    co_temp ;
    //第一个例化模块一般格式有所差异，需要单独例化
    full_adder1  u_adder0(
        .Ai     (a[0]),
        .Bi     (b[0]),
        .Ci     (c),
        .So     (so[0]),
        .Co     (co_temp[0]));

    //generate模块内的例化：每个实例变量用之前的变量
    genvar        i ;
    generate
        for(i=1; i<=3; i=i+1) begin: adder_gen
        full_adder1  u_adder(
            .Ai     (a[i]),
            .Bi     (b[i]),
            .Ci     (co_temp[i-1]), //上一个全加器的溢位是下一个的进位
            .So     (so[i]),
            .Co     (co_temp[i]));
```

```
        end
    endgenerate

    assign co    = co_temp[3] ;

endmodule
```

Verilog

时序控制

Verilog 提供了 2 大类时序控制方法：**时延控制和事件控制**。事件控制主要分为**边沿触发事件控制与电平敏感事件控制**

时延控制

基于时延的时序控制出现在表达式中，它指定了语句从开始执行到执行完毕之间的时间间隔。

时延可以是数字、标识符或者表达式。

根据在表达式中的位置差异，时延控制又可以分为常规时延与内嵌时延。

常规时延

遇到常规延时时，该语句需要等待一定时间，然后将计算结果赋值给目标信号。

格式为：#delay procedural_statement，例如：

```
reg  value_test ;
reg  value_general ;
#10  value_general    = value_test ;
```

该时延方式的另一种写法是直接将井号 `#` 独立成一个时延执行语句，例如：

```
#10 ;
value_ single         = value_test ;
```

内嵌时延

遇到内嵌延时时，该语句先将计算结果保存，然后等待一定的时间后赋值给目标信号。

内嵌时延控制加在赋值号之后。例如：

```
reg  value_test ;
reg  value_embed ;
value_embed        = #10 value_test ;
```

需要说明的是，这 2 种时延控制方式的效果是有所不同的。

当延时语句的赋值符号右端是常量时，2 种时延控制都能达到相同的延时赋值效果。

当延时语句的赋值符号右端是变量时，2 种时延控制可能会产生不同的延时赋值效果。

事件控制

在 Verilog 中，事件是指某一个 reg 或 wire 型变量发生了值的变化

事件控制用符号 @ 表示，语句执行的条件是信号的值发生特定的变化

关键字 **posedge 指信号发生边沿正向跳变**，**negedge 指信号发生负向边沿跳变**，未指明跳变方向时，则 2 种情况的边沿变化都会触发相关事件

Verilog 中还支持使用电平作为敏感信号来控制时序，即后面语句的执行需要等待某个条件为真。Verilog 中使用关键字 wait 来表示这种电平敏感情况

```verilog
initial begin
    wait (start_enable) ;       //等待 start 信号
    forever begin
        //start信号使能后，在clk_samp上升沿，对数据进行整合
        @(posedge clk_samp)  ;
        data_buf = {data_if[0], data_if[1]} ;
    end
end
```

[1] 原因是wire型被综合为一根连接线，不具有存储器状态