Verilog OJ 题库

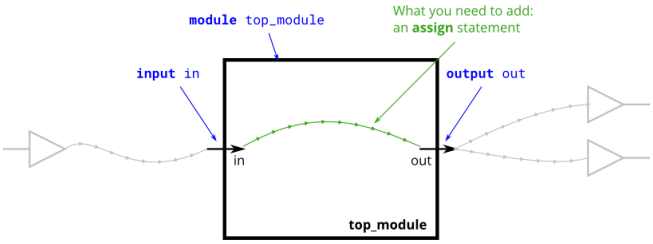
教程简介
本教程通过一系列由浅入深的 Verilog 编程练习，帮助用户快速掌握 Verilog 语法。

【题目 1】：输出 1
问题描述：
编写 Verilog 代码，使电路输出信号 1
代码模板：
```
module top_module( output one );
// 请用户在下方编辑代码
      //assign one = … ;
//用户编辑到此为止
endmodule
```
总结说明：
Verilog 模块代码有固定的格式要求：
-以关键字 module 开始，以关键字 endmodule 结束
-关键字 module 后跟的是模块名，模块名可以由用户自定义，模块名与关键字 module 之间以空格隔开
-Verilog 语法中，用分号（;）表示一条语句的结束
-Verilog 语法中，双斜杠后面跟的是单行注释，仅仅是为了增加代码可读性
-模块名后面的括号内是对于模块端口信号的定义，端口信号类型一般包括输入、输出、输入输出三类，分别对应关键字 input、output、inout，端口信号名称可由用户自定义。
-本教程中后台测试程序会对模块进行例化和仿真，因此请勿擅自更改模块名称和端口信号
-assign 是 Verilog 语法中非常重要的一个关键字，用于对组合逻辑信号赋值。
-1'b1 表示的是 1bit 位宽的信号 1，该信号用二进制表示（b）。同理 1'b0，表示的是 1bit 位宽的信号 0

【题目 2】：输出 0
问题描述：
编写 Verilog 代码，使电路输出信号 0
代码模板：
```
module top_module( output zero );
// 请用户在下方编辑代码
      assign zero = ___ ;
//用户编辑到此为止
endmodule
```
总结说明：
-在 Verilog 中，信号命名一般采用望文生义的方式，这样可以增加代码的可读性
-在 Verilog 中，严格区分大小写，如 Zero 和 zero 表示的是不同的信号
-在 Verilog 中，关键字都是小写的，信号命名时不得与关键字相同

【题目 3】：wire
问题描述：

wire 是 Verilog 的关键字，用于表征信号类型的，其含义是线网，wire 可理解为物理连线，但又有所不同，因为 verilog 中的 wire 是有方向的，例如设计一模块，模块名命名为 top_module，输入信号名为 in，输出信号名为 out，使 in 与 out 直连，如下图所示：

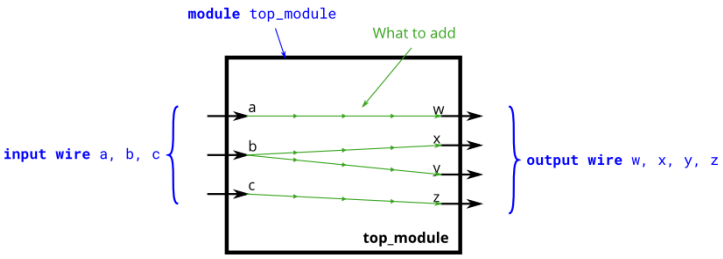


请使用 assign 语句将代码补充完整，使其实现上述电路图的功能
代码模板：

```
module top_module( input in, output out );
// 请用户在下方编辑代码

//用户编辑到此为止
endmodule
```

总结说明：
-上述代码表示使用一根线将 in 与 out 信号连起来，信号的传递是单向的，从 in 到 out

【题目 4】：多个端口的模块
问题描述：
创建一个名为 top_module 的 Verilog 模块，包含 3 个输入信号，4 个输出信号使其信号连接关系如下图所示



因为有 4 个输出信号，因此需要使用 4 个 assign 语句分别对信号赋值
代码模板：

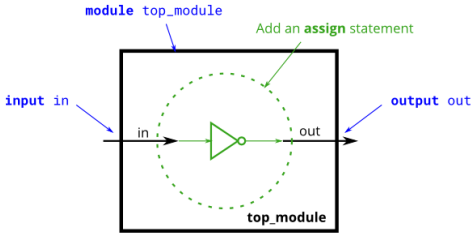
```
module top_module(
    input a,b,c,
    output w,x,y,z );
// 请用户在下方编辑代码

//用户编辑到此为止
endmodule
```

总结说明：
-对于同类型的信号，可以同一个关键字后进行定义，信号之间用逗号（,）隔开
-一般来说，一个 assign 关键字只能实现一条赋值语句，对于不同信号的赋值，需要用到多个 assign 语句

【题目 5】：非门

问题描述：

创建一个名为 top_module 的 Verilog 模块，实现非门的功能，如下图所示



代码模板：

```
module top_module( input in, output out );
// 请用户在下方编辑代码

//用户编辑到此为止
endmodule
```

总结说明：

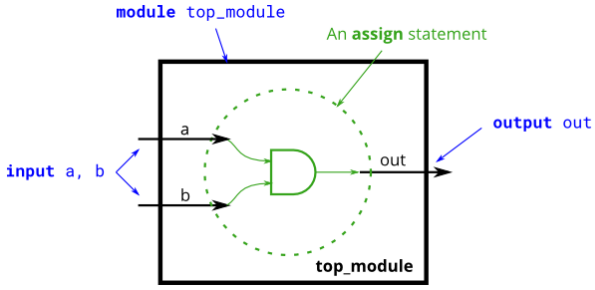
-在 Verilog 中“~”符号是一个单目运算符，表示对信号按位取反，这里 in 信号只有一位，因此就是对 in 信号取反。 “！”运算符也可实现该功能（但对于多 bit 信号，两者存在区别）

-在 Verilog 中有几种常用的逻辑操作运算符，与（&）、或（|）、异或（^）（这些运算符对单 bit 信号和多 bit 信号有些区别，目前我们只考虑单 bit 信号）

【题目 6】：与门

问题描述：

创建一个 Verilog 模块，实现与门的逻辑功能，如下图所示：



代码模板：

```
module top_module(
  input a,
  input b,
  output out );
// 请用户在下方编辑代码

//用户编辑到此为止
endmodule
```

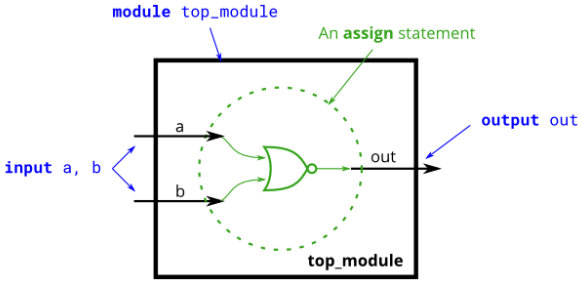
总结说明：

-多个相同类型的输入信号（数据类型、位宽都一样），既可以使用同一关键字，也可以分别用 input 关键字定义，对于输出信号也是一样

【题目 7】：或非门

问题描述：

创建一个 Verilog 模块，实现或非门的逻辑功能，如下图所示



代码模板：

```
module top_module(
    input a,
    input b,
    output out );
// 请用户在下方编辑代码

//用户编辑到此为止
endmodule
```

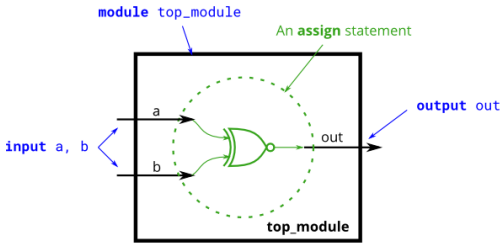
总结说明：

-在逻辑表达式中，通过使用括号可以实现优先级的区分，括号内的运算优先级高于括号外，括号可以嵌套

【题目 8】：同或门

问题描述：

创建一个 Verilog 模块，实现同或门的逻辑功能，对于两输入同或门来说，输入相同时输出 1，输入不同时输出 0，正好与异或门相反，如下图所示



代码模板：

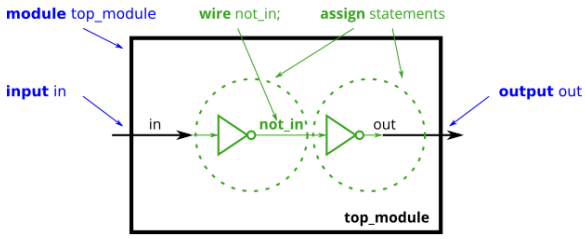
```
module top_module(
    input a,
    input b,
    output out );
// 请用户在下方编辑代码

//用户编辑到此为止
endmodule
```

总结说明：

【题目 9】：线网型中间信号

问题描述：

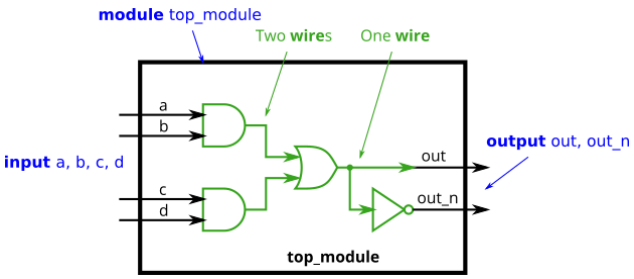
之前的 verilog 模块结构都比较简单，输出信号可直接用输入信号的逻辑表达式表示出来，模块功能稍微复杂时，一般都会用到中间信号，以下图为例，输入信号 in 经过两个非门后输出到 out 端口，为了在 verilog 模块中表示两个非门中间的这跟信号，需要将其定义为线网型（wire）信号，此处我们命名为 not_in。



上述模块的 verilog 代码为：

```
module top_module (
    input in,               // Declare an input wire named "in"
    output out             // Declare an output wire named "out"
);

    wire not_in;           // Declare a wire named "not_in"

    assign out = ~not_in;  // Assign a value to out (create a NOT gate).
    assign not_in = ~in;   // Assign a value to not_in (create another NOT gate).

endmodule   // End of module "top_module"
```

请根据上述示例，完成下图中电路所对应的 Verilog 模块



代码模板：

```
module top_module(
    input a,
    input b,
    input c,
    input d,
    output out,
    output out_n    );
// 请用户在下方编辑代码

//用户编辑到此为止
endmodule
```

总结说明：

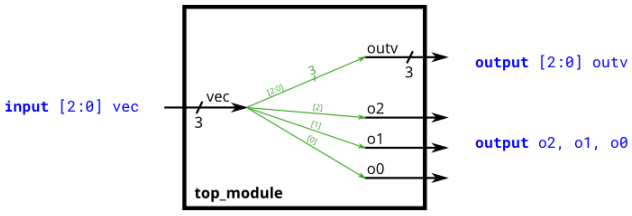
-输入输出信号也有数据类型，不声明数据类型时默认为 wire 类型，例如题目中的 a、b、c、

d、out、out_not 信号都是 wire 类型
-题目中的或门输出可以定义成一个中间信号，也可以直接用输出信号 out 表示，因为 out 端口与或门输出之间没有逻辑门，是直接连在一起的。

【题目 10】：向量
问题描述：
向量是为了编写、阅读代码方便，将一组相关的信号用一个向量名称统一命名的方式。例如：wire [7:0] w,声明了一个 8bit 位宽的向量信号 w，实际上代表的是 8 个 1bit 的 wire 型信号。注意向量信号的声明是将位宽信息放在信号名之前，这与 C 语言不太一样。我们可以将向量信号中的一位或多位单独拿来使用。如下所示：

```
wire out;
wire [3:0] out_4;
wire [99:0] my_vector;                          // Declare a 100-element vector
assign out = my_vector[11] & my_vector[10];  // Part-select one bit out of the vector
assign out_4 = my_vector[23:20];                //  选择其中 4bit 信号
```

创建一 verilog 模块，具有一个 3bit 位宽的输入向量信号，然后将其输出到 3bit 位宽的输出向量信号，同时再分别输出到 3 个 1bit 位宽的输出信号，如下图所示



代码模板：

```
module top_module (
    input wire [2:0] vec,
    output wire [2:0] outv,
    output wire o2,
    output wire o1,
    output wire o0    ); // Module body starts after module declaration
//  请用户在下方编辑代码

//用户编辑到此为止
endmodule
```

总结说明：
-向量信号在本质上还是多个单 bit 信号的集合

【题目 11】：向量_续 1
向量在使用时必须被声明，其格式为：
type [upper:lower] vector_name;
type 表示的是数据类型，一般常用的有 wire 和 reg 两类，对于端口信号，还应额外加上端口类型，如 input、output 等，以下是一些常见的向量声明示例：

```
wire [7:0] w;          // 8-bit wire
reg  [4:1] x;          // 4-bit reg
```

```
output reg [0:0] y;     // 1-bit reg that is also an output port (this is still a vector)
input wire [3:-2] z;    // 6-bit wire input (negative ranges are allowed)
output [3:0] a;             // 4-bit output wire. Type is 'wire' unless specified otherwise.
wire [0:7] b;               // 8-bit wire where b[0] is the most-significant bit.
```

在声明向量时，“大小端”方式都支持，如：wire [0:7] b 是大端，wire [7:0] w 是小端。一旦使用某种方式声明后，后续使用是也必须采用相同的方式，如声明时为小端方式 wire [3:0] vec，使用是就不能采用大端方式 vec[0:3]。

一般来说，我们推荐使用小端模式，位宽为 width-1：0，例如 output [3:0] a。

1bit 位宽的信号也可以采用向量方式声明，如：output reg [0:0] y，等同于 output reg y。

位宽不匹配：

在 verilog 中，支持在不同位宽的信号之间进行赋值，这一特性导致了很多难以被发现的电路功能错误，如下代码所示：

```
wire [2:0] a, c;          // Two vectors
assign a = 3'b101;   // a = 101
assign b = a;             // b =     1    implicitly-created wire
assign c = b;             // c = 001    <-- bug
my_module i1 (d,e); // d and e are implicitly one-bit wide if not declared.
                                  // This could be a bug if the port was intended to be a vector.
```

b、e 信号没有进行声明，所以默认为 1bit 的 wire 信号，但很多情况是程序员忘记了对两个信号进行声明，最终导致了 bug。在代码顶端增加宏定义：`default_nettype none 可以对没有声明的信号报错，能够有效避免这类 bug 的产生。

定义数组

在信号后面添加向量，可以定义信号组，如下代码所示

```
reg [7:0] mem [255:0]; // 256 个信号，每个信号都是 8bit 位宽的 reg 类型.
reg mem2 [28:0];         // 29 个信号，每个信号都是 1bit 位宽的 reg 类型
```

访问向量信号

可以直接通过向量信号名称使用信号，如下所示：

```
assign w = a;
```

如 a 位宽小于 w，则会将 a 信号高位补零，如 a 位宽大于 w，则舍弃 a 的高位，最终使得位宽相等。

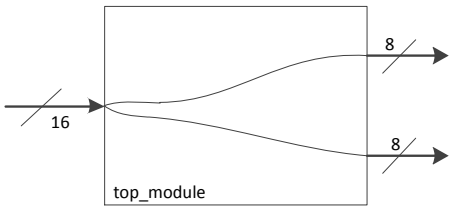
Part-select

对于向量信号，也可以选取向量中的部分位宽进行单独操作（也称为 part-select），如下所示：

```
w[3:0]          // Only the lower 4 bits of w
x[1]              // The lowest bit of x，x 被定义为 reg [4:1] x
x[1:1]           // ...also the lowest bit of x
z[-1:-2]       // Two lowest bits of z，z 被定义为 input wire [3:-2] z
b[3:0]          // Illegal. Vector part-select must match the direction of the declaration.
b[0:3]          // The *upper* 4 bits of b.
assign w[3:0] = b[0:3]; // Assign upper 4 bits of b to lower 4 bits of w.w[3]=b[0],w[2]=b[1]...
```

问题描述：

创建一 Verilog 模块，将 16bit 输入信号分成两个 8bit 的信号，然后输出，如下图所示



代码模板：

```
`default_nettype none       // Disable implicit nets. Reduces some types of bugs.
module top_module(
    input wire [15:0] in,
    output wire [7:0] out_hi,
    output wire [7:0] out_lo );
//  请用户在下方编辑代码

//用户编辑到此为止
endmodule
```

总结说明：

【题目 12】：向量_续 2

问题描述：

一个 32bit 的向量信号包含有 4 个字节（bit[31:24]、bit[23:16]等），创建一个电路，用以调整 4 个字节的顺序，该电路经常用于在不同大小端系统之间进行数据交互：

AaaaaaaaBbbbbbbbCccccccDddddddd => DddddddCccccccBbbbbbbbAaaaaaaa

提示：part-select 操作即可以用于赋值语句的左侧也可用于右侧。

代码模板：

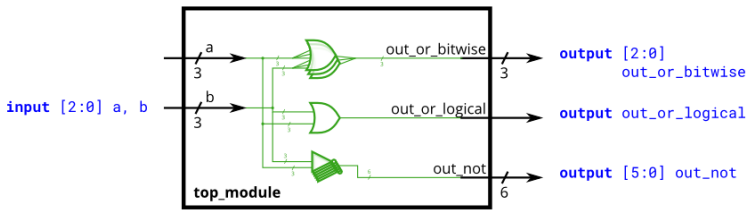
```
module top_module(
    input [31:0] in,
    output [31:0] out );//
    // assign out[31:24] = ...;
endmodule
```

总结说明：

【题目 13】：位操作_1

题目描述：

创建一个电路，包含两个 3bit 的输入信号 a 和 b，分别对 ab 进行按位或、逻辑或操作，以及将 ab 拼接成 6bit 信号后进行按位取反，如下图所示：



代码模板：

```
module top_module(
    input [2:0] a,
```

```
    input [2:0] b,
    output [2:0] out_or_bitwise,
    output out_or_logical,
    output [5:0] out_not
);

endmodule
```

总结说明：

-对于同一个 wire 信号，只能在一个 assign 语句中进行赋值

-对于向量信号，可以使用 part-select 将其分成若干部分，在使用 assign 语句分别赋值。

【题目 14】：位操作_2

题目描述：

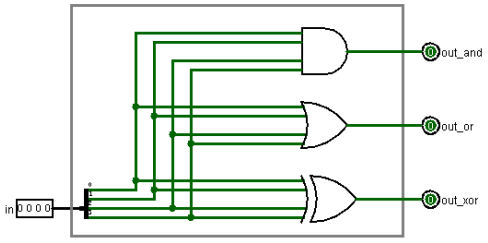
创建一个组合逻辑电路，包含 4bit 输入（in[3:0]），和 3 个输出，分别为：

-out_and：四输入与门的输出信号

-out_or：四输入或门的输出信号

-out_xor：四输出异或门的输出信号

电路结构如下图所示



代码模板：

```
module top_module(
    input [3:0] in,
    output out_and,
    output out_or,
    output out_xor
);

endmodule
```

总结说明：

-对于多 bit 位宽的向量信号，可以看成是多个信号的组合，可以分开来使用

-逻辑门可以有多个输入信号

【题目 15】：向量拼接

题目描述：

part_selection 用于选择向量信号中的一部分，而向量拼接算子{a,b,c}用于将多个信号组合成一个位宽更大的向量信号，如：

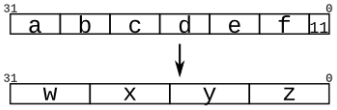
{3'b111, 3'b000}　等同于 6'b111000

{1'b1, 1'b0, 3'b101} 等同于 5'b10101

{4'ha, 4'd10} 等同于 8'b10101010 // 4'ha and 4'd10 are both 4'b1010 in binary
向量拼接时，每个信号都需要有明确的位宽，这样拼接后的信号才会有明确的位宽。例如，{1,2,3}就是非法的，因为无法确定各信号的位宽，语法检查时会报错。
向量拼接算子既可以用于赋值语句的左侧，也可用于右侧，如下所示：

```
input [15:0] in;
output [23:0] out;
assign {out[7:0], out[15:8]} = in;
assign out[15:0] = {in[7:0], in[15:8]};
assign out = {in[7:0], in[15:8]};
```

创建 Verilog 电路，将 6 个 5bit 位宽的输入信号，以及 2bit 的常量信号 2'b11 拼接成 32bit 的向量信号，并将其拆成 4 个 8bit 的信号，分别赋值给 4 个输出信号，如下图所示：

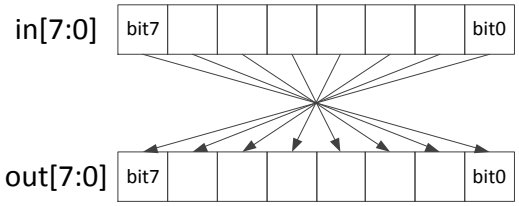


代码模板：

```
module top_module (
    input [4:0] a, b, c, d, e, f,
    output [7:0] w, x, y, z );//
    // assign { ... } = { ... };
endmodule
```

总结说明：
-通过合理的使用 part-selection 和拼接算子，能够实现精确到每一根信号线的控制。

【题目 16】：向量翻转
题目描述：
创建 verilog 电路，将 8bit 的输入信号按 bit 翻转，并输出到输出端口，如下图所示



代码模板：

```
module top_module(
    input [7:0] in,
    output [7:0] out
);

endmodule
```

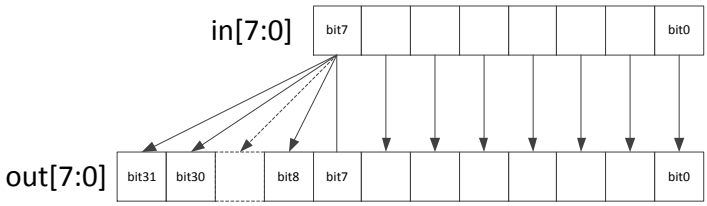
总结说明：
-不能够使用 out[7:0] = in[0:7]来实现上述功能，这种写法会报语法错误。

【题目 17】：复制算子_1
题目描述：
复制算子是拼接算子的一种特殊情况，如 a={b,b,b,b,b,b}便可以写成 a={6{b}}的形式。复制

算子的格式为：{num{vector}}，其中 num 必须为常量。如下所示：

```
{5{1'b1}}                 // 5'b11111 (or 5'd31 or 5'h1f)
{2{a,b,c}}                // The same as {a,b,c,a,b,c}
{3'd5, {2{3'd6}}}     // 9'b101_110_110
```

创建一 verilog 电路，将一个 8bit 位宽的输入信号进行符号位扩展，并通过 32bit 的输出端口输出，如下图所示



代码模板：

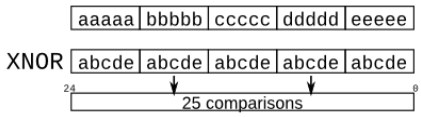
```
module top_module (
    input [7:0] in,
    output [31:0] out );//
    // assign out = { replicate-sign-bit , the-input };
endmodule
```

总结说明：

【题目 18】：复制算子_2

题目描述：

创建一 verilog 电路，包含 5 个 1bit 输入，使所有输入两两进行同或（两 bit 相同时输出 1，不同时输出 0），并将结果通过 25bit 的向量信号输出，如下图所示：



使用复制算子实现该电路，可以大大减少代码量，提高编码效率。

代码模板：

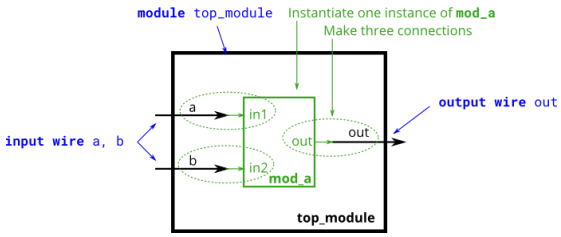
```
module top_module (
    input a, b, c, d, e;
    output [24:0] out );//
    // The output is XNOR of two vectors created by
    // concatenating and replicating the five inputs.
    // assign out = ~{ ... } ^ { ... };
endmodule
```

总结说明：

【题目 19】：模块例化

题目描述：

通过前面一系列的练习，用户应当已经熟悉单个模块电路的设计了。对于功能上更复杂的电路模块，一般都是由若干子模块以及附加的功能电路构成的。



在模块实例化过程中，被例化模块的端口信号是最重要的，用户甚至可以不知道模块的内部结构。上图展示了一个非常简单的包含有子模块电路的电路结构，在此电路中，创建模块 mod_a 的一个实例化，并将该实例化模块的三个端口（in1,in2,out）与顶层电路的三个端口(a,b,out)直接连接，其中 mod_a 模块的代码如下：

```
module mod_a ( input in1, input in2, output out );
    // Module body
    assign out = in1 & in2; //这只是一个简单的示例
endmodule
```

模块实例化一般有两种语法格式，分别称为基于端口名称的实例化和基于端口位置的实例化。

基于位置的实例化和 C 语言中的函数调用类似（只是语法上类似，实际上该例化会产生实际的硬件电路），以上述 mod_a 模块的实例化为例，可以在上层模块中使用以下语句：

```
module top_module(input wa,input wb,output wc);
    mod_a inst_name1(wa,wb,wc);
endmodule
```

其中 inst_name1 是 mod_a 模块的实例化名称，可以由用户自定义，通过这种例化方式，便实现了端口对应：wa←→in1, wb←→in2, wc←→out 。

基于端口名称的实例化如下所示

```
module top_module(input wa,input wb,output wc);
    mod_a   inst_name2(
    .out    (wc),
    .in1    (wa),
    .in2    (wb));
endmodule
```

本教程推荐用户使用基于端口名称的例化方式，因为这种方式编写的代码可读性更强。

试创建一 verilog 电路，并按照上图中所示实例化 mod_a 模块（建议使用基于端口名称的方式实例化）。

代码模板：

```
module top_module(input a,input b,output out);

endmodule
```

总结说明：

-推荐使用基于端口名称的实例化方式

-模块调用就像是一个树形的层次结构，不允许循环调用，如 a 调用 b，b 又调用 a，也不允许模块调用自身，即模块 c 中又实例化模块 c。

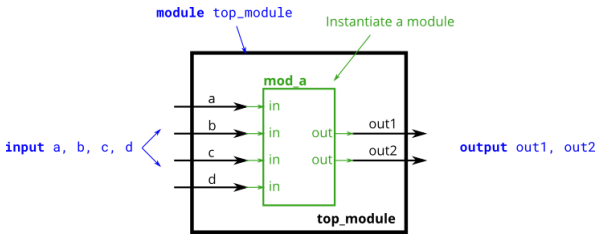
-不允许在进程块（如 always、initial 等）或赋值语句（如 assign 语句）内进行模块实例化

-模块的实例化名称可以自定义，如在同一模块中要对一个模块多次实例化，需要有不同的实例化名称。

【题目 20】：基于端口位置的实例化

题目描述：

创建一 verilog 电路，实现对模块 mod_a 基于端口位置的实例化，如下图所示：



其中 mod_a 模块的代码为：

```
module mod_a ( output out1, out2,
                    input in1,in2,in3,in4 );
    assign out1 = in1 & in2 & in3 & in4;  //这只是一个简单的示例
    assign out2 = in1 | in2 | in3 | in4;      //这只是一个简单的示例
endmodule
```

代码模板：

```
module top_module (
    input a,
    input b,
    input c,
    input d,
    output out1,
    output out2
);

endmodule
```

总结说明：

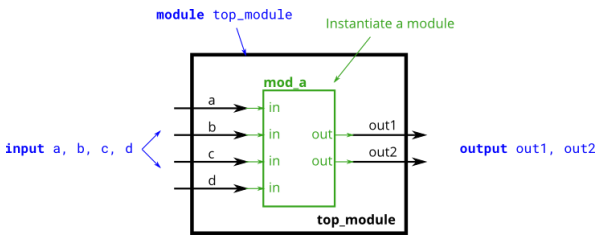
-实例化名称可以与模块名称相同

-实例化模块时，需要注意端口信号的位宽相匹配，本例中都是 1bit，所以不存在问题

【题目 21】：基于端口名称的实例化

题目描述：

创建一 verilog 电路，实现对模块 mod_a 基于端口名称的实例化，如下图所示：



Port in mod_a	Port in top_module
output out1	out1
output out2	out2
input in1	a
input in2	b
input in3	c
input in4	d

其中 mod_a 模块的代码为：

```
module mod_a ( output out1, out2,
                    input in1,in2,in3,in4 );
    assign out1 = in1 & in2 & in3 & in4;  //这只是一个简单的示例
```

```
    assign out2 = in1 | in2 | in3 | in4;       //这只是一个简单的示例
endmodule
```

代码模板：

```
module top_module (
    input a,
    input b,
    input c,
    input d,
    output out1,
    output out2
);

endmodule
```

总结说明：

-无

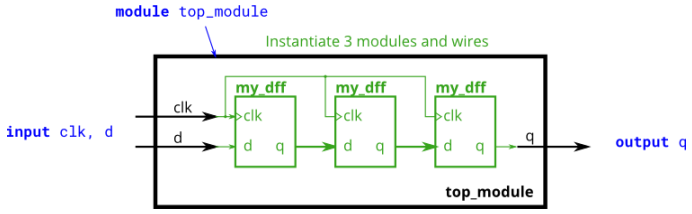
【题目 22】： 多个模块的例化

题目描述：

对于给定模块 my_dff，包含两个输入信号和一个输出信号（D 触发器模块），其代码如下：

```
module my_dff(input clk,input d,output reg q);
    always@(posedge clk)
        q <= d;
endmodule
```

请创建一 verilog 模块，在该模块中将 my_dff 模块例化 3 次，并串行连接，使其构成一个长度为 3 的移位寄存器，其中 3 个模块公用一个 clk 信号，如下图所示：



为实现电路功能，用户需要在顶层模块定义一些内部信号，从而能够将 3 个例化的模块进行连接。

代码模板：

```
module top_module ( input clk, input d, output q );

endmodule
```

总结说明：

-mod_a 模块为触发器，是时序逻辑电路中最基本最核心的功能电路，此处是本教程中第一次设涉及，用户稍作了解即可，后面会专门学习

-mod_a 模块中的 q 信号被定义成了 reg 型，这是不同于 wire 的另一种常用数据类型

-mod_a 模块中使用了关键字 always，这是 verilog 中常用的进程语句，用于实现对 reg 类型数据的赋值

-mod_a 模块中使用了关键字 posedge，表示信号的上升沿，同步时序逻辑电路都是边沿

敏感的，该关键字在时序逻辑电路中经常遇到。
-mod_a 模块中对 q 赋值语句为 q<=d，称为非阻塞赋值，是时序逻辑中常用的赋值方式
-在同一模块中，每个信号和子模块都需要有唯一的名称。

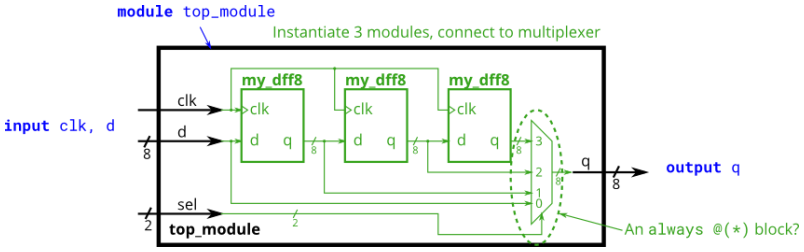
【题目 23】： 模块与向量信号

题目描述：

对于给定模块 my_dff8，其代码如下所示：

```
module my_dff(input clk,input [7:0] d,output reg [7:0] q);
    always@(posedge clk)
        q <= d;
endmodule
```

试创建一 verilog 模块，对 my_dff8 模块例化 3 次，并串行连接，构成一个 8bit 位宽长度为 3 的移位寄存器，同时可以通过选择信号选择输出结果，如下图所示：



代码模板：

```
module top_module (
    input clk,
    input [7:0] d,
    input [1:0] sel,
    output [7:0] q );

endmodule
```

总结说明：

-对于较为复杂的组合逻辑，可以使用 always 进程块实现，其功能与 assign 相同
-always 语句既可以描述组合逻辑（电平敏感，always@(*)），也可以描述时序逻辑(边沿敏感，always@(posedge clk))，除了后面的敏感变量格式不同外，进程块内的赋值方式也不一样（组合逻辑采用阻塞赋值方式“=”，时序逻辑采用非阻塞赋值方式“<=”）。
-case 是 verilog 中常用的一个关键字，其对应的电路是一个多路复用器，对于用 case 语句实现的组合逻辑，一般最后应加上 default 语句，以防电路综合时生成锁存器。

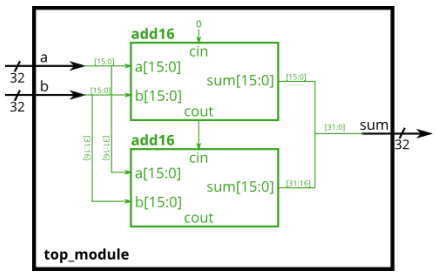
【题目 24】： 加法器

题目描述：

对于给定的 16bit 加法器电路，其代码如下：

```
module add16 ( input[15:0] a, input[15:0] b, input cin, output[15:0] sum, output cout );
    assign {cout,sum} = a + b + cin;
endmodule
```

试创建一 verilog 模块，在该模块中实例化两个 16bit 的加法器，并进行适当的连接，最终构成一个 32bit 的加法器，该加法器输入进位位为 0，如下图所示：



代码模板：

```
module top_module(
    input [31:0] a,
    input [31:0] b,
    output [31:0] sum
);

endmodule
```

总结说明：

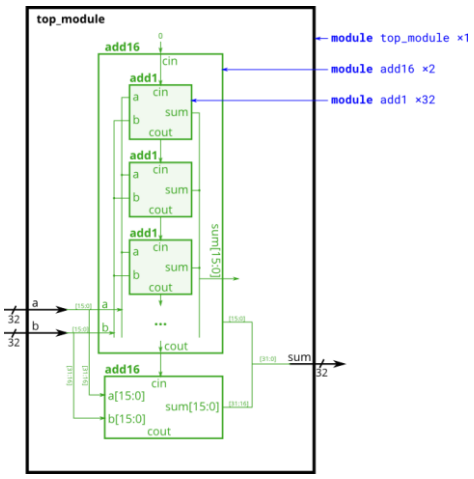
-模块例化时，输入信号端口必须提供数据，输出信号如用不到可以悬空

-上层模块与被例化模块的端口信号名称可以相同也可以不同，这些信号有不重合的作用域

【题目 25】： 多层次例化加法器

题目描述：

在此练习中，用户需要创建一个包含两层调用的电路，在顶层模块中，实例化两个 16bit 位宽的加法器 add16,而 add16 模块又是通过例化 16 个 1bit 全加器实现的，如下图所示：



在本设计中，一共涉及到 3 个模块，分别是：顶层模块、add16 模块、add1 模块，其中 add16 模块源代码如下：

```
module add16 ( input[15:0] a, input[15:0] b, input cin, output[15:0] sum, output cout )
wire c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15;
add1 inst_0(.a(a[0]),.b(b[0]),.cin(cin),.sum(sum[0]),.cout(c1));
add1 inst_1(.a(a[1]),.b(b[1]),.cin(c1),.sum(sum[1]),.cout(c2));
add1 inst_2(.a(a[2]),.b(b[2]),.cin(c2),.sum(sum[2]),.cout(c3));
add1 inst_3(.a(a[3]),.b(b[3]),.cin(c3),.sum(sum[3]),.cout(c4));
add1 inst_4(.a(a[4]),.b(b[4]),.cin(c4),.sum(sum[4]),.cout(c5));
```

```
add1 inst_5(.a(a[5]),.b(b[5]),.cin(c5),.sum(sum[5]),.cout(c6));
add1 inst_6(.a(a[6]),.b(b[6]),.cin(c6),.sum(sum[6]),.cout(c7));
add1 inst_7(.a(a[7]),.b(b[7]),.cin(c7),.sum(sum[7]),.cout(c8));
add1 inst_8(.a(a[8]),.b(b[8]),.cin(c8),.sum(sum[8]),.cout(c9));
add1 inst_9(.a(a[9]),.b(b[9]),.cin(c9),.sum(sum[9]),.cout(c10));
add1 inst_10(.a(a[10]),.b(b[10]),.cin(c10),.sum(sum[10]),.cout(c11));
add1 inst_11(.a(a[11]),.b(b[11]),.cin(c11),.sum(sum[11]),.cout(c12));
add1 inst_12(.a(a[12]),.b(b[12]),.cin(c12),.sum(sum[12]),.cout(c13));
add1 inst_13(.a(a[13]),.b(b[13]),.cin(c13),.sum(sum[13]),.cout(c14));
add1 inst_14(.a(a[14]),.b(b[14]),.cin(c14),.sum(sum[14]),.cout(c15));
add1 inst_15(.a(a[15]),.b(b[15]),.cin(c15),.sum(sum[15]),.cout(cout);
endmodule
```

用户需要自行完成顶层模块和 add1 模块的 verilog 代码。

代码模板：

```
module top_module (
    input [31:0] a,
    input [31:0] b,
    output [31:0] sum);//

endmodule

module add1 ( input a, input b, input cin,   output sum, output cout );
    // Full adder module here
endmodule
```

总结说明：

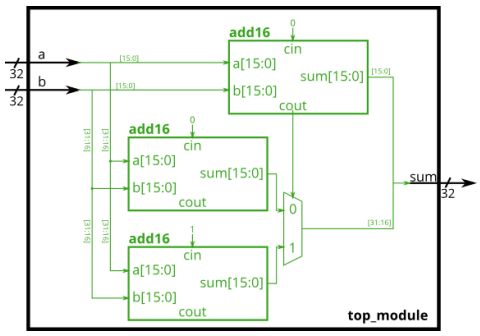
-如在设计中需要大量重复例化一个模块多次（如 add16 模块中多次例化 add1），除使用上述 add16 模块中的实现方式外，还可以考虑使用 generate 关键字，具体用法可自行调研，此处不做要求。

-使用层次化的设计方法，可以像堆积木一样，通过简单的电路构成复杂的功能电路，如 CPU 等

【题目 26】： 进位选择加法器

题目描述：前例中的加法器成为串行进位加法器，只有等前一级的加法器运算结束产生进位位之后，下一级加法器才能利用进位位进行计算，因此电路延时会随加法器串联级数的增加而线性增加，这使得电路计算速度大大降低。设每一级全加器的延时为 t，则 32bit 加法器的延时则为：32t。

为降低电路整体延时，我们可以按下图进行设计：



我们将电路分为两段，每段实现 16bit 的加法，为了使高 16 位与低 16 位同时进行运算，我们采用两个 add16 对高位进行计算，区别在于进位位分别为 0 和 1，最终通过低 16 位加法器的输出进位作为选择控制信号，选择高 16 位的运算结果。这样，32bit 加法器的延时就变为：$16t+t_{mux2} \approx 16t$,延时降低了接近一倍，这种以空间（增加电路）换时间（提高速度）的做法，在数字电路设计中经常使用。

请创建 Verilog 模块，实现上图中的电路结构，其中 add16 不需要用户编写，其声明如下：

```
module add16 ( input[15:0] a, input[15:0] b, input cin, output[15:0] sum, output cout );
    assign {cout,sum} = a + b + cin;
endmodule
```

代码模板：

```
module top_module(
    input [31:0] a,
    input [31:0] b,
    output [31:0] sum
);

endmodule
```

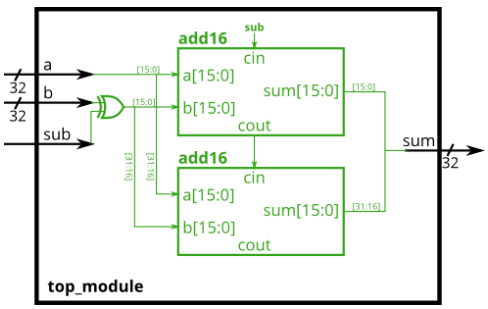
总结说明：

-增加电路结构，以提高电路整体性能，是电路设计中的常用方法

【题目 27】： 加法减法器

题目描述：

通过对加法器进行改造，可以支持加、减两种运算。我们知道，电路中有符号数通常使用补码表示，如-b 其补码为：~b + 1（按位取反然后加 1）。因此，对于减法算式 a-b,可以理解为 a+(-b) = a+(~b+1)= a + (~b) +1,因此对于减法运算，可以将加法器进行如下改造实现：



实现减法运算时，首先通过 32bit 的异或门，将信号 b 按位取反，同时将输入进位位置 1，

实现加法运算时，b 保持不变，输入进位位置 0。
其中 add16 模块代码如下，用户可直接调用：

```
module add16 ( input[15:0] a, input[15:0] b, input cin, output[15:0] sum, output cout );
    assign {cout,sum} = a + b + cin;
endmodule
```

请创建 Verilog 模块，实现上述电路功能。
代码模板：

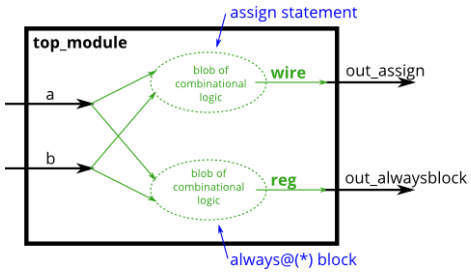
```
module top_module(
    input [31:0] a,
    input [31:0] b,
    input sub,
    output [31:0] sum
);

endmodule
```

总结说明：
-无

【题目 28】： always 过程块_组合逻辑
题目描述：
所有的数字电路都是由逻辑门和连线构成的，因此理论上来说都可以通过模块的连接和 assign 语句进行描述，然而在很多情况下这并不是最方便的一种方式，过程块提供了一种更加方便的描述方式，always 过程块便是其中最常用的一种。
对于可综合电路（即能转化成实际电路的 verilog 描述方式，与之相对的是不可综合电路，多用于电路仿真，不能转换成实际电路），有两种 always 块的语法形式：
-组合逻辑电路：always@(*)
-时序逻辑电路：always@(posedge clk)
组合逻辑电路的 always 块与 assign 语句等效，用户描述组合逻辑电路时，可根据便利性选择其中一种方式使用。两者生成的硬件电路一般是等效的，但在语法规则上稍有不同：
-assign 语句只能对一个信号进行赋值，always 块内可对多个信号进行赋值
-assign 语句中被赋值信号为 wire 类型，always 块内被赋值信号需定义为 reg 类型
-always 块内支持更加丰富的语法，如使用 if…else..、case 等适合实现交复杂的组合逻辑
例如下述两条语句是等效的（out1 需定义为 wire 类型，out2 需定义为 reg 类型，但这仅仅是语法上的要求，生成的电路并没有区别）：

```
assign out1 = a & b | c ^ d;
always @(*) out2 = a & b | c ^ d;
```

其对应的电路图如下所示：

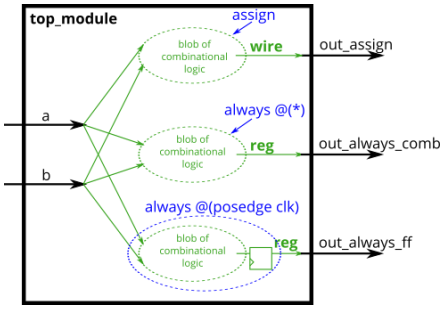


always 语句后的括号内放的是敏感变量列表，对于上例来说，可以写成 always @(a,b,c,d) out2 = a & b | c ^ d，但为了简单起见，我们一般都用符号*代替。
试创建一 verilog 模块，实现一与门，分别用 assign 语句和 always 块实现。
代码模板：

```
module top_module(
    input a,
    input b,
    output wire out_assign,
    output reg out_alwaysblock
);

endmodule
```

总结说明：
-无

【题目 29】： always 过程块_时序逻辑
题目描述：
通过前例已经了解到，对于可综合电路，有两种 always 块的语法形式：
-组合逻辑电路：always@(*)
-时序逻辑电路：always@(posedge clk)
用 always 描述的时序逻辑电路，除了像组合逻辑 always 块那样生成组合逻辑电路外，还会生成一组触发器（或称寄存器），用于寄存组合逻辑的输出。寄存器的输出只有在时钟的上升沿时（posedge clk）才会更新，其余时刻均保持不变。
阻塞赋值和非阻塞赋值：
在 Verilog 中，有三种赋值方式，分别为：
-连续赋值（如 assign x = y;），该赋值方式只能用于过程块(如 always 块)之外
-阻塞赋值（如 x = y;），该赋值方式只能用在过程块（如 always@（*））内
-非阻塞赋值（如 x <= y;），该赋值方式只能用在过程块内（如 always@（posedge clk））
在设计 Verilog 模块时，请遵循以下原则：
-在组合逻辑的 always 块内采用阻塞赋值
-时序逻辑的 always 块内采用非阻塞赋值
违背这一原则将可能导致难以发现的电路错误，且可能导致仿真与综合的不一致，请用户切记。至于为何这样，初学者可以不必理会，简单理解为 verilog 语法规范性要求即可。
创建一 verilog 电路，分别采用上述三种赋值方式实现异或门电路，如下图所示：



代码模板：

```
module top_module(
```

```
    input clk,
    input a,
    input b,
    output wire out_assign,
    output reg out_always_comb,
    output reg out_always_ff     );

endmodule
```

总结说明：

-always 块内被赋值的信号都应定义成 reg 类型

-always 块内，组合逻辑采用阻塞赋值（a=b），时序逻辑采用非阻塞赋值（a<=b）

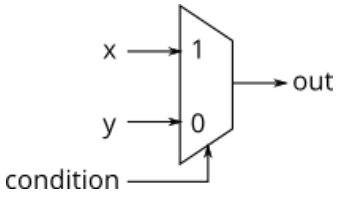
-always 语句括号内是敏感变量列表，时序逻辑是边沿敏感的，posedge clk 表示的是 clk 信号的上升沿，此外，还可以是 negedge clk，表示 clk 信号的下降沿。

【题目 30】： if…else…语句

题目描述：

if 语句用于过程块内部，其对应的电路是二选一的选择器，以下述代码为例：

```
always@(*)
begin
    if(condition)   out = x;
    else            out = y;
end
```



上述代码与下面的 assing 语句完全等效：

```
assign out = (condition) ? x : y;
```

试创建一 Verilog 模块，分别采用 assing 语句和过程块内的 if 语句实现下述选择器电路：

sel_b1	sel_b2	out_assign out_always
0	0	a
0	1	a
0	0	a
1	1	b

代码模板：

```
module top_module(
    input a,
    input b,
    input sel_b1,
    input sel_b2,
    output wire out_assign,
    output reg out_always     );

endmodule
```

总结说明：

-if…else…语句对应硬件上的二选一选择器

-if…else…可以嵌套使用，如 if… else if… else if… else…

-使用 if 语句描述组合逻辑时，务必加上 else 语句，以免产生锁存器（数字电路设计中应

尽力避免产生锁存器）

【题目 31】： if 语句_锁存器

题目描述：

使用 Verilog 设计电路时，应按照如下流程：

-确定你需要的电路或逻辑门

-确定输入输出信号，以及产生输出信号的组合逻辑块

-确定组合逻辑块后面是否加上一组触发器。

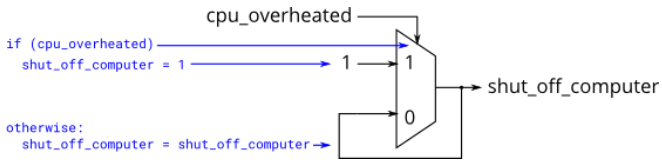
因此，应极力避免这样的心态：试着写下一段代码，然后期待其生成正确的电路，如：

- if (cpu_overheated) then shut_off_computer = 1;

- if (~arrived) then keep_driving = ~gas_tank_empty;

语法正确的代码并不一定能产生功能正常的电路，一般来说都是因为不小心引入了锁存器造成的。如上述例子所示，除了指定的情况外（cpu_overheated），还有一些其它情况，这时会发生什么？在 verilog 中，其结果就是保持不变，这意味着要记住当前状态，从而产生了锁存。

```
always @(*) begin
    if (cpu_overheated)
        shut_off_computer = 1;
end
always @(*) begin
    if (~arrived)
        keep_driving = ~gas_tank_empty;
end
```



为消除锁存器，我们应当使组合逻辑过程块中的条件完备，即 if 语句后应加上 else 语句。试修改上述两段代码，以消除锁存器。

代码模板：

```
module top_module (
    input       cpu_overheated,
    output reg shut_off_computer,
    input       arrived,
    input       gas_tank_empty,
    output reg keep_driving   ); //
    always @(*) begin
        if (cpu_overheated)
            shut_off_computer = 1;
    end
    always @(*) begin
        if (~arrived)
            keep_driving = ~gas_tank_empty;
    end
endmodule
```

总结说明：

-使用 if 语句描述组合逻辑时，务必加上 else 语句，以免产生锁存器（数字电路设计中应尽力避免产生锁存器）

【题目 32】： case 语句

题目描述：

Verilog 中的 case 语句几乎等同于 if…else if…else…序列,其语法与 C 语言中的 switch 语句类似，如下例所示：

```
always @(*) begin         // This is a combinational circuit
      case (in)
        1'b1: out = 1'b1;   //  对应 if(in>=1)  语句
        1'b0: out = 1'b0;   //  对应 else 语句
        default: out = 1'bx; //使用 case 实现组合逻辑时，必须有 default，以防出现锁存器
      endcase                    //endcase 语句表示 case 语句结束，两者成对使用
end
```

-case 语句以关键字 case 开始，以 endcase 结束，两者成对出现

-case 语句中每个条目只执行一条语句，如要在一个条目下进行多个赋值，需要将多条预计放在 begin/end 关键字之间

-case 条目允许重复或部分重叠，第一个匹配到的条目有效

当有多个条目进行选择时，使用 case 语句比 if…else…语句方便很多，本例中，使用 Verilog 设计六选一选择器，当 sel 信号在 0~5 时，选择对应的数据输出，否则输出 0，输入输出数据位宽均为 4bit。

代码模板：

```
module top_module (
      input [2:0] sel,
      input [3:0] data0,
      input [3:0] data1,
      input [3:0] data2,
      input [3:0] data3,
      input [3:0] data4,
      input [3:0] data5,
      output reg [3:0] out     );//
      always@(*) begin    // This is a combinational circuit
            case(...)
      end

endmodule
```

总结说明：

【题目 33】： 优先编码器

题目描述：

优先编码器是一种组合电路，当给定一个输入位矢量时，输出矢量中第一个 1 位的位置。例如，给定输入 8'b10010000 的 8 位优先级编码器将输出 3'd4，因为位[4]是高的第一位。设计一个 4 位优先编码器电路。如果没有输入位为高电平（即输入为零），则输出为零。请注意，4 位数字有 16 种可能的组合。

代码模板：

```
module top_module (
```

```
    input [3:0] in,
    output reg [1:0] pos    );

endmodule
```

总结说明：

-无

【题目 34】： casez 语句

题目描述：

对于一个 8bit 输入信号的优先级编码器。例如，输入 8'b10010000 应该输出 3'd4，因为位[4]是第一个高位。如果使用上例中的 case 语句，则需要包含 256 个条目。但如果 case 语句中的 case 项支持 don-care 位，我们可以将其减少到 9 个条目。这就是 casez 的用途：它将值为 z 的位在比较中视为无关紧要。

例如，这将实现上一练习中的 4 输入优先级编码器：

```
always @(*) begin
    casez (in[3:0])
        4'bzzz1: out = 0;     // in[3:1] can be anything
        4'bzz1z: out = 1;
        4'bz1zz: out = 2;
        4'b1zzz: out = 3;
        default: out = 0;
    endcase
end
```

case 语句的行为就好像每个项都是按顺序检查的。有某些输入（例如 4'b1111）匹配多个条目，则选择第一个匹配项（因此 4'b1111 匹配第一个项，out=0，而不匹配后面的任何项）。

还有一个类似的 casex，它将 x 和 z 都视为不在乎，此处不再介绍。

创建一 Verilog 模块，实现 8bit 信号输入的优先级编码器

代码模板：

```
module top_module (
    input [7:0] in,
    output reg [2:0] pos    );

endmodule
```

总结说明：

-无

【题目 35】： 避免锁存器

题目描述：

假设您正在构建一个电路来处理游戏中 PS/2 键盘上的扫描码。给定接收到的扫描码的最后两个字节，您需要指示是否按下了键盘上的一个箭头键。这涉及到一个相当简单的映射，它可以实现为一个 case 语句（或者 if…else if…），包含四个 case。

Scancode [15:0]	Arrow key
16'he06b	left arrow
16'he072	down arrow
16'he074	right arrow
16'he075	up arrow
Anything else	none

您的电路有一个 16 位输入和四个输出，该电路识别这四个扫描码并确认正确的输出。为避免产生锁存，必须在所有四种情况和默认（default）情况下，为所有四个输出指定一个值。这可能涉及许多不必要的输入。解决这个问题的一个简单方法是在 case 语句之前为输出分配一个“默认值”：

```
always @(*) begin
    up = 1'b0; down = 1'b0; left = 1'b0; right = 1'b0;
    case (scancode)
        ... // Set to 1 as necessary.
    endcase
end
```

这种类型的代码确保在所有可能的情况下为输出赋值（0），除非 case 语句重写赋值。这也意味着 default 项变得不必要。

请试着将上述代码补充完整。

代码模板：

```
module top_module (
    input [15:0] scancode,
    output reg left,
    output reg down,
    output reg right,
    output reg up    );

endmodule
```

总结说明：

-本例中，综合器会生成一个行为与代码描述功能相同的电路，但硬件并不是按顺序“执行”verilog 代码的。

【题目 36】： 条件运算符

题目描述：

Verilog 中有一个跟 C 语言中类似的三目条件运算符（ ? : ）,其语法格式为：

(condition ? if_true : if_false)

该表达式可以用于为其它信号赋值，例如：signal = condition ? if_true : if_false;

该语句等同于：

```
if(condition)   signal = if_true;
else            signal = if_false;
```

因涉及到 3 个操作数，并且能实现条件运算的功能，因此称为三目条件运算符。

下面是几个使用该运算符的例子：

```
(0 ? 3 : 5)       // 条件为假，因此表达式的值为 5
(sel ? b : a)     // 二选一选择器
```

```
always @(posedge clk)              // 触发器
  q <= toggle ? ~q : q;
always @(*)                        // 有线状态机(FSM)
  case (state)
    A: next = w ? B : A;
    B: next = w ? A : B;
  endcase
assign out = ena ? q : 1'bz;   // 三态门
((sel[1:0] == 2'h0) ? a : (sel[1:0] == 2'h1) ? b : c ) //嵌套使用
```

试设计一计算最小值功能的 Verilog 模块，给定四个无符号数，求最小值。提示：

1. 可以综合使用比较运算符（< or >）和条件运算符（? :）。
2. 有必要的话，可以定义中心变量

代码模板：

```
module top_module (
    input [7:0] a, b, c, d,
    output [7:0] min);//

endmodule
```

总结说明：

-有限状态机（FSM）是一种非常常用且典型的电路结构，概括来说，就是使电路在有限的几个状态之间转换，并以此控制电路的运行。后面会通过专门的题目进行学习。

【题目 37】：归约运算符

题目描述：

您已经熟悉了两个值之间的位运算，例如 a&b 或 a^b。有时，您需要创建一个对一个向量的所有位进行操作的逻辑门，如（a[0]&a[1]&a[2]&a[3]…），如果向量很长的话，也会很麻烦。归约运算符可以对向量的位进行 AND、OR 和 XOR 运算，产生一位输出：

```
& a[3:0]        // AND: a[3]&a[2]&a[1]&a[0]. Equivalent to (a[3:0] == 4'hf)
| b[3:0]        // OR:   b[3]|b[2]|b[1]|b[0]. Equivalent to (b[3:0] != 4'h0)
^ c[2:0]        // XOR: c[2]^c[1]^c[0]
```

这些是只有一个操作数的单目运算符（类似于 NOT 运算符！和~）。您还可以反转这些门的输出来创建 NAND、NOR 和 XNOR 门，例如（~&d[7:0]）。

当传输数据使用的是一个不完美的渠道时，经常使用奇偶校验作为一种简单的方法来检测错误。创建一个将为 8 位字节计算奇偶校验位的电路（这将在字节中添加第 9 位）。我们将使用“偶数”奇偶校验，其中奇偶校验位只是所有 8 个数据位的异或。

代码模板：

```
module top_module (
    input [7:0] in,
    output parity);

endmodule
```

总结说明：

【题目 38】D 触发器

题目描述：
D 触发器是一种在时钟信号的边缘（通常是上升沿）存储位并定期更新的电路。在 Verilog 中，时序逻辑电路描述（一般语法为：always@(posedge clk)）都会生成组合逻辑块加 D 触发器的电路结构。
D 触发器是最简单的“组合逻辑后跟一个触发器”形式，其中组合逻辑部分只是一根导线。

创建 verilog 模块，实现一个 D 触发器。
代码模板：

```
module top_module (
    input clk,      // Clocks are used in sequential circuits
    input d,
    output reg q );//
    // Use a clocked always block
    //     copy d to q at every positive edge of clk
    //     Clocked always blocks should use non-blocking assignments
endmodule
```

总结说明：
-时序逻辑电路在 Verilog 中都要用 always 进程块实现
-always 进程块的敏感变量列表中需要列出信号边沿（一般是时钟信号的上升沿）
-时序逻辑电路是由组合逻辑电路及触发器构成的
-时序逻辑电路输出信号只在敏感变量的边沿时刻（一般是时钟信号的上升沿）更新
-在 always 进程块内被赋值的信号需定义成 reg 类型
-时序逻辑电路采用非阻塞赋值方式（<=）

【题目 39】寄存器
题目描述：
寄存器在本质上来说就是触发器，我们一般将由多个触发器构成的向量信号成为寄存器。试设计一个在时钟上升沿出发的 8bit 位宽的寄存器。
代码模板：

```
module top_module (
    input clk,
    input [7:0] d,
    output reg [7:0] q
);

endmodule
```

总结说明：
-无
【题目 40】有复位功能的寄存器
题目描述：
创建一个带有复位信号（reset）的 8bit 位宽的寄存器，在触发信号（clk）的上升沿，当复位信号为高电平时，寄存器输出 0，否则与输入信号 d 同步。

代码模板：

```
module top_module (
    input clk,
    input reset,                // Synchronous reset
    input [7:0] d,
    output reg [7:0] q
);

endmodule
```

总结说明：

-在该电路中，复位信号只能在时钟信号的上升沿起作用，这称为同步复位（与之对应的有异步复位电路）

-信号的复位值也可以为非零的数据。

-带有复位功能的电路，应保证复位信号的优先级最高，即放在第一个 if 语句中

【题目 41】下降沿触发的寄存器

题目描述：

在时序逻辑电路中，敏感变量不但可以是触发信号的上升沿（posedge），也可以是下降沿（negedge）,试创建 8 bit 位宽的寄存器，所有 DFF 都应该由 clk 的下降沿（负边缘）触发。同时采用高电平有效的同步复位方式，复位值为 0x34 而不是零。

代码模板：

```
module top_module (
    input clk,
    input reset,
    input [7:0] d,
    output [7:0] q
);

endmodule
```

总结说明：

-在同一个 always 进程块中，同一触发信号只能使用一种边沿，即上升沿和下降沿不可同时使用。(例如：always@(posedge clk or negedge clk)，这种写法是错误的)

【题目 42】异步复位的寄存器

题目描述：

在前面的例子中，复位信号只能在触发信号的触发边沿才能起作用，也就是说如果没有触发边沿就无法对电路复位，这大大限制了复位的功能，因此还有一种经常使用的复位方式，称为异步复位。

为了使复位信号不依赖于时钟边沿，则复位信号也应该放在 always 进程块的敏感变量列表中。复位信号高电平有效和低电平有效在编码时稍有不同，对于高电平有效的复位信号来说，可以写成如下形式：

```
always@(posedge clk or posedge reset)
begin
    if(reset) ···
```

```
    else …
end
```
如果是低电平有效的复位信号，则应写成如下形式：
```
always@(posedge clk or negedge reset)
begin
    if(~reset) …
    else …
end
```
试创建 Verilog 模块，实现一个时钟上升沿触发的，支持高电平有效的异步复位功能的 8bit 寄存器，寄存器复位值为 0。
代码模板：
```
module top_module (
    input clk,
    input areset,    // active high asynchronous reset
    input [7:0] d,
    output [7:0] q
);

endmodule
```
总结说明：
-时序逻辑的 always 进程块敏感变量列表中，一般包含的是时钟和复位信号的触发边沿
-在同一 always 块中，不能将同一信号的上升沿和下降沿同时作为触发信号
-always 敏感变量列表中，边沿触发和电平触发不能混用（如：always@(posedge clk or areset)是错误的，要么全是边沿敏感，要么全是电平敏感，不能混用）
-复位信号高电平有效是，敏感变量列表中应使用复位信号的上升沿作为触发信号，否则使用复位信号的下降沿

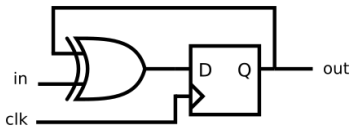
【题目 43】带使能的寄存器
题目描述：
在前面的电路中，寄存器输出端 q 在每个时钟的上升沿都会更新一次，但有时候我们可能需要使输出端保持不变，这时就需要加入使能信号，创建一 16bit 位宽（2byte）的寄存器，其中每字节都由一个使能信号控制，使能为 0 时，输出保持不变，使能为 1 时更新 q。时钟上升沿触发，同步复位，复位低电平有效，复位值为 0。
代码模板：
```
module top_module (
    input clk,
    input resetn,
    input [1:0] byteena,
    input [15:0] d,
    output [15:0] q
);

endmodule
```
总结说明：

-在时序逻辑中，if 语句后不跟 else 并不会产生锁存器，case 语句的条件也可以不完备
-在 verilog 中，一个信号不能在多个 always 块中赋值，否则会报错或导致电路功能错误
-一个 always 块中可以对对个信号赋值，但建议只处理一个或一类信号，不同的信号分成多个 always 块处理。

【题目 44】触发器+逻辑门

题目描述：

编写 verilog 代码，实现下图所示的电路功能



代码模板：

```
module top_module (
    input clk,
    input in,
    output out);

endmodule
```

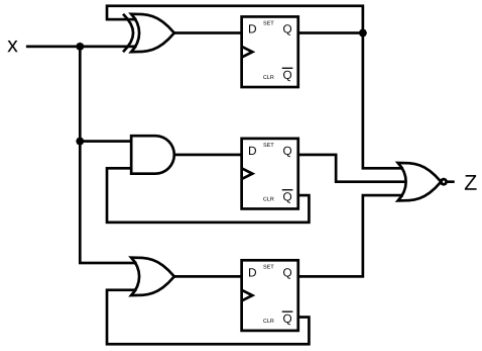
总结说明：

-无

【题目 45】寄存器+逻辑门

题目描述：

编写 Verilog 代码，实现下图所示的电路功能，假设所有 D 触发器的初始复位值为 0



代码模板：

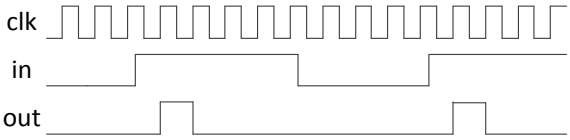
```
module top_module (
    input clk,
    input x,
    output z
);

endmodule
```

总结说明：

-无

【题目 46】上升沿检测

题目描述：
在实际应用中，我们经常需要对某个信号的边沿进行检测，并以此作为后续动作的触发信号（例如电脑键盘的某个按键被按下或者被松开，在电路中则对应的是电平的变化）。
设计一个电路，包含 clk 信号、1bit 输入信号 in 和 1bit 输出信号 out，当 in 信号从 0 变为 1 时（相对于 clk，该信号变化频率很慢），out 信号在 in 信号上升沿附近输出 1 个时钟周期的高电平脉冲，其余时刻都为 0，如下图所示



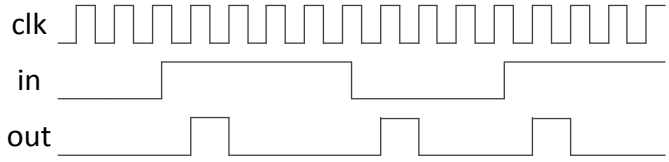
代码模板：

```
module top_module (
    input clk,
    input in,
    output out
);

endmodule
```

总结说明：

【题目 47】双边沿检测

题目描述：根据上题的思路，设计一双边沿检测电路，即在输入信号的上升沿和下降沿附近时刻，各输出一个高电平脉冲，如下图所示



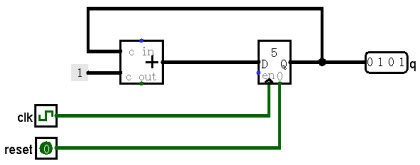
代码模板：

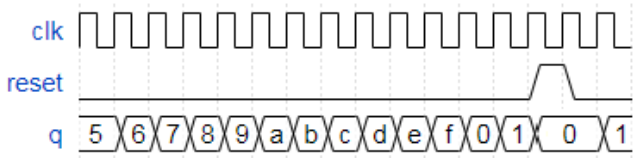
```
module top_module (
    input clk,
    input in,
    output out
);

endmodule
```

总结说明：
-无

【题目 48】计数器

题目描述：
计数器是一种比较简单且常用的时序逻辑电路，下面电路图是一个从 0 到 15 循环计数的累加计数器，每个时钟的上升沿计数值加一，采用异步复位方式，高电平有效，复位值为 0，该电路是由一个 4bit 加法器和一个 4bit 寄存器构成，波形如下所示。





试编写 Verilog 代码，完成上述电路功能。

代码模板：

```
module top_module (
    input clk,
    input reset,        // 异步复位，高电平有效，复位值为 0
    output reg [3:0] q);

endmodule
```

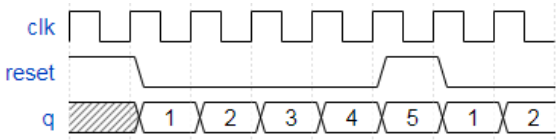
总结说明：

-4'b1111 + 4'b1=5'b10000，取低 4 位的话结果为 0，因此计数到 F 后回从 0 重新计数

【题目 49】十进制计数器

题目描述：

设计一个十进制计数器电路，从 1 到 10 循环计数，采用同步复位方式，高电平有效，复位值为 1，如下面波形图所示



代码模板：

```
module top_module (
    input clk,
    input reset,            // Synchronous active-high reset
    output reg [3:0] q);

endmodule
```

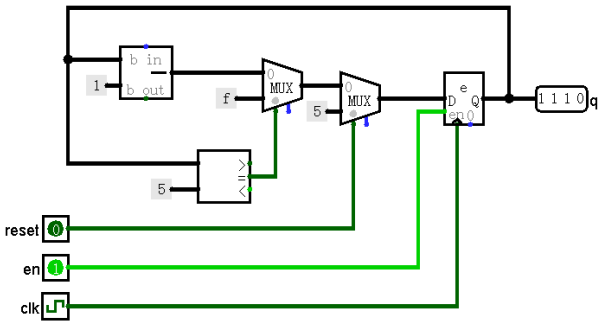
总结说明：

-无

【题目 50】带使能的计数器

题目描述：

创建一带有使能信号的递减计数器，当使能信号有效（高电平）时，从 15 到 5 循环递减计数，每个周期减 1，使能信号无效时，计数值保持不变，电路采用同步复位方式，高电平有效，复位值为 5，电路结构如下图所示



代码模板：

```
module top_module (
    input clk,
    input reset,
    input en,
    output reg [3:0] q);

endmodule
```

总结说明：

-无

【题目 51】秒表

题目描述：

在前面的时序逻辑电路中，我们没有对时钟频率做限定，但实际上在某些电路中时钟频率对设计有显著影响，例如秒表等计时电路。假设时钟频率为 4Hz，要求设计一个秒表电路，每 1 秒钟计数加一，计数结果用 BCD 码表示，共需 8bit，其中高 4 位为十位数（0~5，每 10 秒钟加 1），低 4 位为个位数（0~9，每 1 秒钟加 1），电路采用高电平有效的同步复位方式，复位值为 0。

代码模板：

```
module top_module(
    input clk,  //4Hz
    input reset,
    output [7:0] ss);

endmodule
```

总结说明：

-当时钟频率过高，需要降频时，可通过计数器生成一个特定的脉冲信号（如本例中的 one_ss_pulse 信号），以该脉冲信号作为计数器的使能信号，以达到降频计数的目的

-将多个计数器组合使用，可以实现更加灵活的计数功能

【题目 52】移位寄存器

题目描述：

构建一个 4 位移位寄存器（右移），具有异步复位、同步加载和启用功能。

-areset：将移位寄存器重置为零。

-load：用数据[3:0]加载移位寄存器，而不是移位。

-ena：右移（q[3]变为零，q[0]移出并消失）。

-q：移位寄存器的内容。

-如果 load 和 ena 输入同时有效，则 load 输入具有更高的优先级。

代码模板：

```
module top_module(
    input clk,
    input areset,   //异步、高有效、复位值为 0
    input load,
    input ena,
    input [3:0] data,
    output reg [3:0] q);

endmodule
```

总结说明：

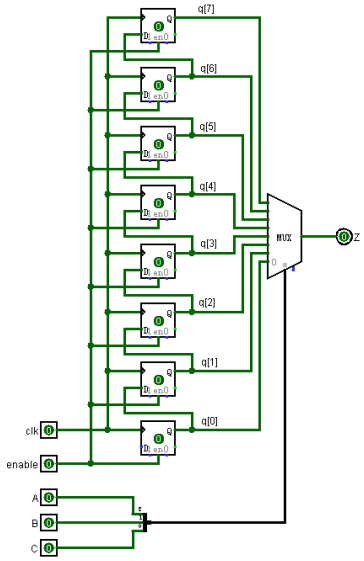
-无

【题目 53】查找表

题目描述：

在本题中，请设计一个 8x1 存储器的电路，其中写入内存是通过移位来完成的，而读取是“随机访问”，就像在典型的 RAM 中一样，以三个输入（A、B、C）作为地址，实现对存储器单元的随机访问。

首先，使用 8 个 D 型触发器实现一个 8 位移位寄存器。将触发器输出标记为 Q[0]···Q[7]。移位寄存器输入应被调用 S，它将输入 Q[0]（高位通过移位操作输入）。通过使能信号控制是否移动。电路的行为应如下：当 ABC 为 000 时，Z=Q[0]，ABC 为 001 时，Z=Q[1]，等等。您的电路应仅包含 8 位移位寄存器和多路复用器(该电路称为 3 输入查找表（LUT））。



代码模板：

```
module top_module (
    input clk,
    input enable,
    input S,
    input A, B, C,
    output reg Z );

endmodule
```

总结说明：

【题目 54】ROM

题目描述：

在数字电路中，不可避免的需要对数据进行存储和访问，虽然寄存器能够实现数据的存储功能，但使用寄存器进行大量数据的存储并不现实，这是因为：第一，寄存器实现成本过高；第二，寄存器结构复杂，难以提高集成度。因此，一般使用存储器进行大段数据的存储，常见的存储器有 ROM（只读存储器）、RAM（随机访问存储器）和 FIFO（先进先出队列）。ROM 顾名思义，该存储器里面的数据是预选设置好的，在使用时只能读取，不能写入，在使用 FPGA 作为硬件平台时，一般是通过 EDA 工具中的 IP 生成工具来生成的。此外，我们也可以通过使用 Verilog 定义一个数组来实现，并由 EDA 工具自动例化成相应的 IP 核，这样做对于开发者来说简单了很多，但性能多少会受些影响。

对于存储器，其内部数据的初始化有两种方式：(1）在 Verilog 代码中指定初始内容（2）在外部文件中指定初始内容。在实际中，一般采用第二种方式。

对于存储器来说，有两个非常重要的参数：地址位宽和数据位宽，地址位宽与存储器的存储深度（即存储单元数量）相关，如地址宽度为 N，则存储器深度为 2^N，数据位宽则表示每个存储单元所包含的 bit 数。

下例是一个 4*8bit 的 ROM 实例，该 ROM 通过在 Verilog 代码中指定内容进行初始化

```
module rom_4x8bit(
input      [1:0] addr
output    [7:0] q);
reg  [7:0] mem [3:0];
initial
begin
      mem[0] = 8'h00;
      mem[1] = 8'h00;
      mem[2] = 8'h00;
      mem[3] = 8'h00;
end
assign q = mem[addr];
endmodule
```

试根据上述示例，设计一个 8*4bit 的 ROM，并对其进行初始化，使其初始化数据为“0,1,2,3,…”

代码模板：

```
module top_module（
input      [2:0] addr,
output    [3:0] q);

endmodule
```

总结说明：

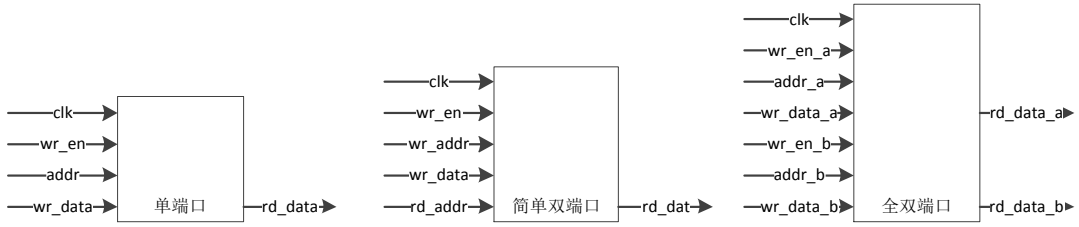
-无

【题目 55】RAM

题目描述：

RAM 全称为随机访问存储器，可以对其内部的单元进行读写访问，根据读写端口数量和类

型，可以分为单端口（读写端口共用地址信号）、简单双端口（读写端口独立，可同时进行读写操作）、全双端口（有两套读写端口）等不同类型。
下图是三种不同 RAM 的接口信号对比，图中列出的都是基本接口信号，根据具体要求，还可以增加复位、读使能、字节使能等端口信号。



下例是一个 4*8bit 的单端口 RAM 的实例，采用外部文件进行初始化

```
module ram_one_port(
input      clk,
input      [1:0] addr,
input      wr_en,
input      [7:0] wr_data,
output     [7:0] rd_data);
reg        [7:0] mem[3:0];
initial
begin
      $readmemh("memfile.dat",mem);
end
assign rd_data = mem[addr];
always@(posedge clk)
begin
      if(wr_en)
            mem[addr] <= wr_data;
end
endmodule
```

上例中使用系统函数$readmemh，该函数可以将文件中的数据初始化到存储器数组中，memfile.dat 为 RAM 的初始化文件，内容如下（16 进制格式，可以使用文本工具编辑）：

```
00
01
02
03
```

试根据上述示例，设计一个 8*16bit 的简单双端口 RAM，并使用 memfile.dat 文件对其初始化，在本例中，该文件已在后台提供，用户可直接使用。
代码模板：

```
module ram_one_port(
input              clk,
input              wr_en,
input      [2:0]   wr_addr,
input      [15:0] wr_data,
input      [2:0]   rd_addr,
```

output [15:0] rd_data);

endmodule

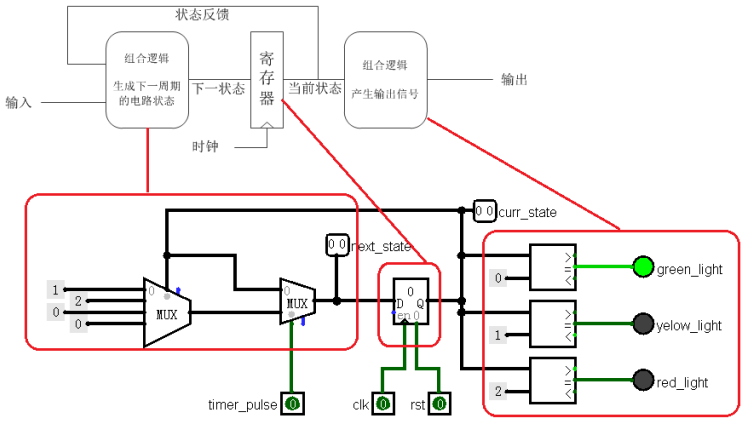
总结说明：

-采用外部文件对存储器进行初始化比 verilog 代码内部初始化话更加方便

【题目 56】有限状态机

题目描述：

下图是摩尔型有限状态机的结构图，我们可以发现其包含三个部分，第一部分为纯组合逻辑，通过现态和输入信号生成次态信号，第二部分为时序逻辑，该时序逻辑非常简单，只包含一个带有复位功能的寄存器单元，复位时现态信号变为初始值，否则在每个时钟的上升沿将次态信号赋值给现态信号。第三部分为组合逻辑，该部分通过现态信号生成各输出信号。

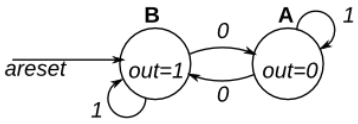


对于上述电路，其 Verilog 代码实现为：

```
module traffic_ctrl(
input     clk,
input     rst,
input     timer_pulse,
output    green_light
);
parameter     C_PASS    = 2'b00;
parameter     C_TRANS = 2'b01;
parameter     C_STOP    = 2'b10;
reg [1:0]     curr_state;
reg [1:0]     next_state;
//有限状态机第一部分
always@(*)
begin
      if(timer_pulse)
      begin
            case(curr_state)
                  C_PASS: next_state = C_TRANS;
                  C_TRANS:next_state = C_STOP;
                  C_STOP: next_state = C_PASS;
```

```
                default:next_state = C_PASS;
            endcase
        end
        else
            next_state = curr_state;
end
//有限状态机第二部分
always@(posedge clk or posedge rst)
begin
        if(rst)
            curr_state    <= C_PASS;
        else
            curr_state    <= next_state;
end
//有限状态机第三部分,各输出信号的赋值都应放在此部分
assign green_light = (curr_state==C_PASS)? 1'b1 : 1'b0;
//...
endmodule
```

试用 Verilog 代码实现下图中的摩尔型状态机，有两个状态，一个输入(in)，一个输出(out)。实现这个状态机。请注意，电路采用异步复位，复位状态是 B。



代码模板：

```
module top_module(
        input clk,
        input areset,       // Asynchronous reset to state B
        input in,
        output out);//
        parameter A=0, B=1;
        reg state, next_state;
        always @(*) begin       //有限状态机第一段
            // State transition logic
        end
        always @(posedge clk, posedge areset) begin       //有限状态机第二段
            // State flip-flops with asynchronous reset
        end
        //有限状态机第三段，信号输出逻辑
        // assign out = (state == ...);
endmodule
```

总结说明：

- parameter 是 Verilog 的一个关键字，通过定义参数，可以提高代码可读性和可维护性
-有限状态机是一种常用的电路设计方式，在 verilog 中一般采用三段式的写法。

【题目 57】读代码找错误

题目描述：

如果设计的 Verilog 模块不能正常工作，说明存在语法或者功能上的错误，这时需要对错误进行定位，对于语法错误，可通过 EDA 工具中的语法检查功能进行定位，对于已经掌握了 Verilog 语法的用户来说，很容易便能解决。对于电路功能上的错误，可以通过以下两种途径定位：阅读代码、电路仿真。

以下是一个 8bit 位宽的二选一选择器的 Verilog 代码，但存在错误，请仔细阅读代码，修改代码使其能正常工作。

代码模板：

```
module top_module (
    input sel,
    input [7:0] a,
    input [7:0] b,
    output out    );

    assign out = (~sel & a) | (sel & b);

endmodule
```

总结说明：

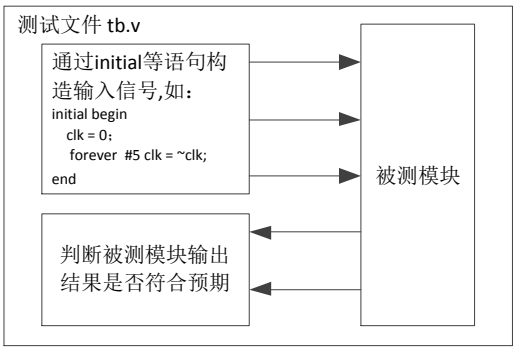
-当电路模块功能有误时，可首先仔细阅读一遍 Verilog 源代码，会有很大概率发现问题

【题目 58】编写仿真文件

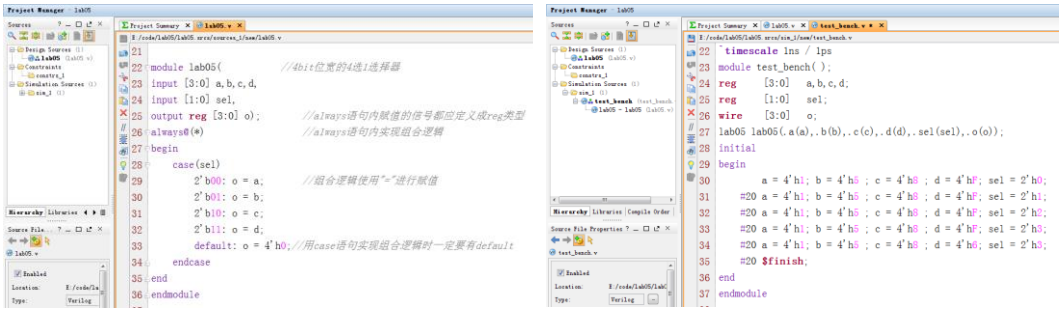
题目描述：

在 Verilog 设计中，仿真是非常重要的一环，绝大部分的电路功能错误都可以通过仿真进行定位并解决。

仿真的过程为：用户使用 Verilog 编写一个没有任何输入信号的测试模块（一般命名为 tb.v,即 testbench 的缩写），在该模块中调用被仿真的模块（及我们使用 Verilog 编写的功能模块），并使用 initial 等关键字构造被测模块的输入信号，通过专门的仿真工具进行仿真，并查看各个信号的波形变化



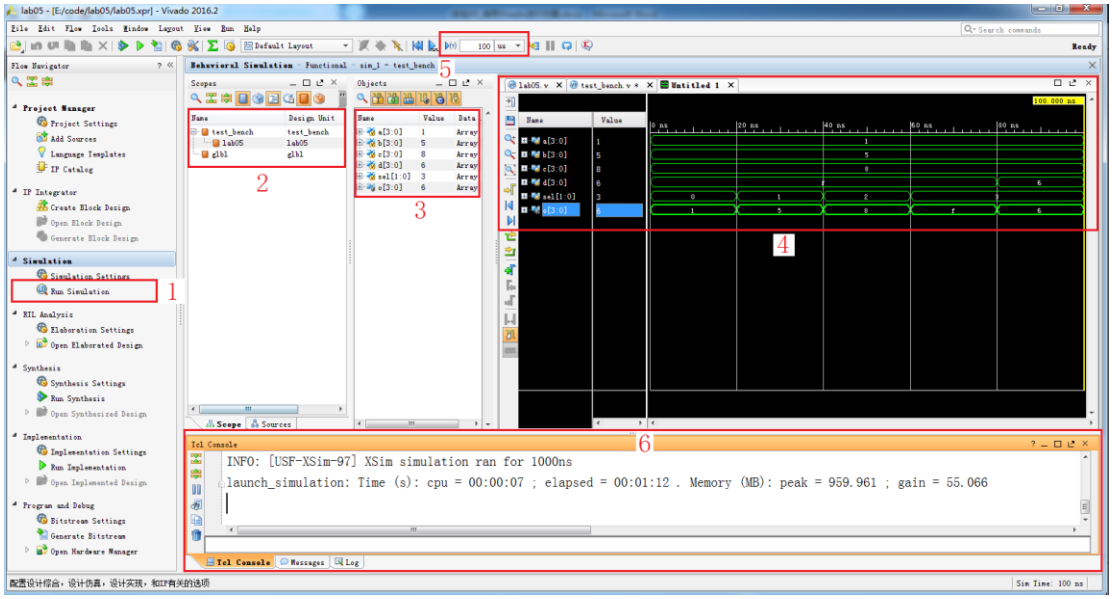
以四选一选择器的仿真为例，被测模块和测试文件的 Verilog 代码如下所示：



```
module lab05(              //4bit位宽的4选1选择器
input [3:0] a, b, c, d,
input [1:0] sel,
output reg [3:0] 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 = 4'h0;//用case语句实现组合逻辑时一定要有default
    endcase
end
endmodule
```

```
`timescale 1ns / 1ps
module test_bench( );
reg     [3:0]   a, b, c, d;
reg     [1:0]   sel;
wire    [3:0]   o;
lab05 lab05(.a(a),.b(b),.c(c),.d(d),.sel(sel),.o(o));
initial
begin
        a = 4'h1; b = 4'h5 ; c = 4'h8 ; d = 4'hF; sel = 2'h0;
    #20 a = 4'h1; b = 4'h5 ; c = 4'h8 ; d = 4'hF; sel = 2'h1;
    #20 a = 4'h1; b = 4'h5 ; c = 4'h8 ; d = 4'hF; sel = 2'h2;
    #20 a = 4'h1; b = 4'h5 ; c = 4'h8 ; d = 4'hF; sel = 2'h3;
    #20 $finish;
end
endmodule
```

仿真结果如下图所示：

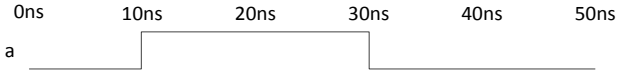


通过观察波形我们可以发现，该电路的仿真波形符合四选一选择器的行为特性，Verilog 代码设计正确。

在 initial 进程块内，语句都是顺序执行的，通过延时符号"#"实现时序控制。例如：

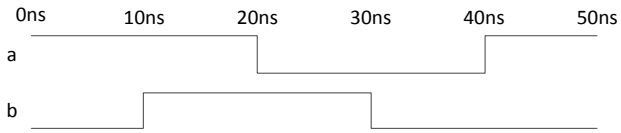
```
`timescale 1ns/1ps
module tb();
reg a;
initial begin
        a = 1'b0;
        #10 a = 1'b1;
        #20 a = 1'b0;
        #20 $finish;
end
```

对应的波形为：



其中`timescale 1ns/1ps 表示时间精度，分辨率为 1ps，时间单位为 1ns，#10 表示延时 10 个时间单位（即 10ns）,$finish 为仿真专用的系统函数，表示仿真结束

试编写测试文件，实现如下波形：



代码模板：

```
module tb();
reg a,b;

endmodule
```

总结说明：

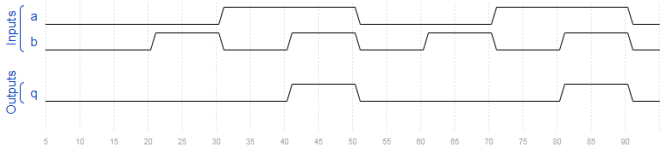
【题目 59】组合逻辑模块仿真

题目描述：

以下是给定模块 mymodule 的 Verilog 代码：

```
module mymodule(
input a,b,
output q);
assign q = a & b;
endmodule
```

试编写仿真文件，使其符合如下波形（时间单位为 1ns）



代码模板：

```
module tb();
reg a,b;
wire q;
//对 ab 信号进行初始化
//例化 mymodule 模块
endmodule
```

总结说明：

-无

【题目 60】生成时钟信号

题目描述：

给定被测试模块 dut，其 verilog 代码如下：

```
module dut(input clk);
    //空模块，仅用于测试
endmodule
```

请编写仿真文件，对该模块进行仿真(dut 模块可直接调用，不需要用户编写)，clk 信号应符合以下波形：

代码模板：

```
module tb();
    //信号定义
    //信号生成
    //模块例化
endmodule
```

总结说明：

-无

【题目 61】单端口 RAM 仿真

题目描述：

以下是一个单端口 RAM 的 verilog 描述

```
module ram_one_port(
input     clk,
input     [1:0] addr,
input     wr_en,
input     [7:0] wr_data,
output    [7:0] rd_data);
reg       [7:0] mem[3:0];
initial
begin
    $readmemh("memfile.dat",mem);
end
assign rd_data = mem[addr];
always@(posedge clk)
begin
    if(wr_en)
        mem[addr] <= wr_data;
end
endmodule
```

如对该模块进行仿真，需要在 tb 文件中生成 clk、addr、wr_en、wr_data 等 RAM 所需的输入信号（在 tb 中都需定义为 reg 类型），并查看 rd_data 端口的输出数据。

clk 信号可以使用以下语句生成：

```
initial begin
    clk = 0;
    forever #5 clk = ~clk; //生成周期为 10 的一个时钟信号，forever 为 verilog 的关键字
end
```

addr 信号应该在时钟的上升沿变化，我们可以借助前面生成的 clk 信号来生成 addr，如：

```
initial begin
    addr = 2'b0;
    repeat(4) begin              //repeat 为 verilog 关键字，表示重复操作
        @(posedge clk);          //等待 clk 信号的上升沿到来
        #1 addr = addr + 1;      //clk 上升沿 1 个时间单位后，addr 加一
    end
end
```

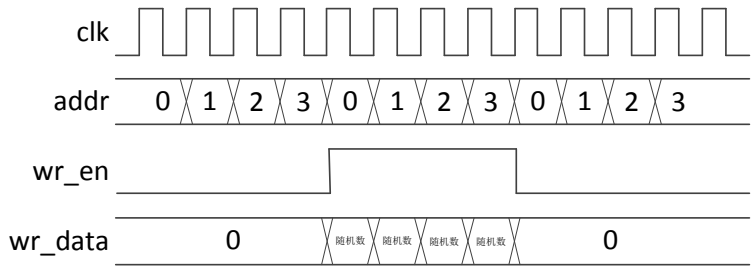
对于 wr_en 信号，我们可以使其持续 4 个周期的高电平，如下所示：

```
initial begin
    wr_en = 0;
    #501;                  //延时一段时间，
    @(posedge clk);
    #1 wr_en = 1;
    @(posedge clk);
    @(posedge clk);
    @(posedge clk);
    @(posedge clk);   //等待 4 个 clk 上升沿
    #1 wr_en = 0;
end
```

此外，我们还可以使用系统函数生成随机数，来作为测试数据，以 wr_data 为例：

```
initial begin
    wr_data = 8'h0;
    repeat(4) begin
        wait(wr_en==1'b1);
        #1 wr_data = $random%256;
        @(posedge clk);
    end
end
```

请根据上述提示，按照以下波形的要求，编写单端口 ram 的仿真文件



代码模板：

```
module tb();
    //信号定义
    //信号生成
    //例化被测模块
endmodule
```

总结说明：

-无