数电实验sv班 Final Review

经12-计18　张诗颖　2021011056

配置Vivado使用VSCode编辑器：`C:\Users\Catherine\AppData\Local\Programs\Microsoft VS Code\Code.exe [file name] +[line number]`
注意：为了防止VSCode卡顿，需要在点开Vivado之前先打开VSCode

1 实验装置

1. 分立实验芯片机器模块

实验中用的分立器件封装都是DIP（Dual In-line Package，双列直插），有**14脚**和**16脚**两种

以**74LS00**插入14脚DIP为例，插入芯片后的供电需要：① 红色连接线连接芯片电源引脚（P14）到**+VCC**接线孔，②黑色连接线连接芯片接地引脚（P7）到-GND接线孔

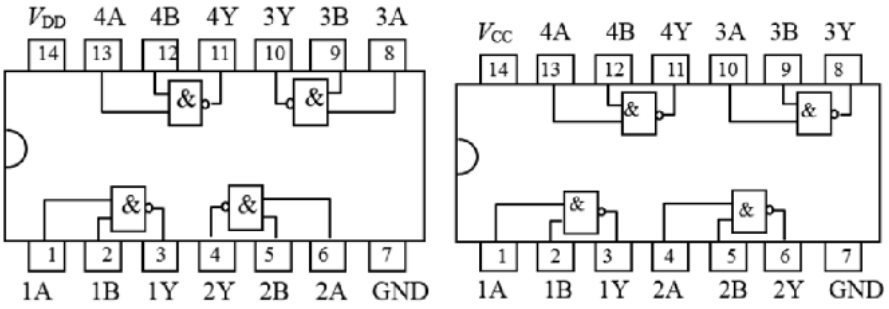
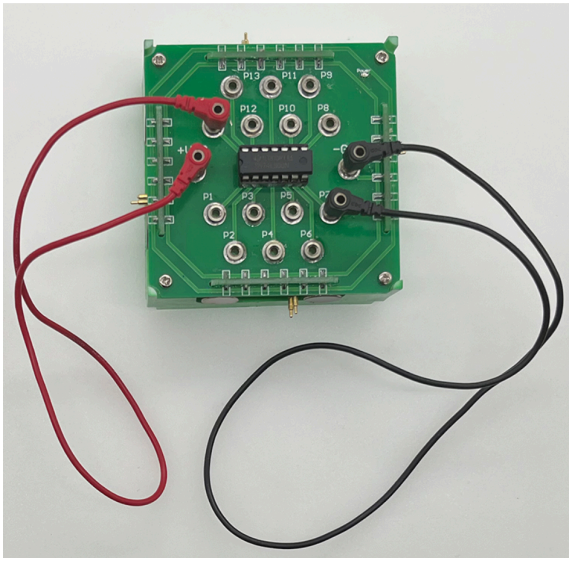


图 1 CD4011（左）与 74LS00（右）器件引脚图



*其他情况同理，如**74LS161**插入16脚DIP：P8接GND，P16接VCC*

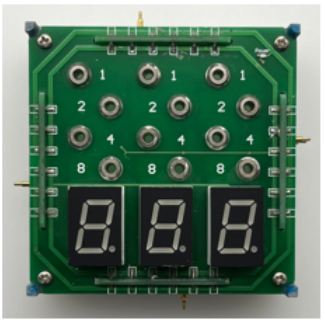
2. 时钟模块

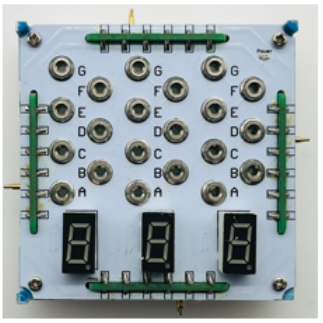
最上部分的5个插线孔：可以对外输出5种时钟频率（1M, 2M, 4M, 8M, 16M）
`CLK`和`RST`：平时保持高电平逻辑`1`，按下后为低电平逻辑`0`，带有防抖功能

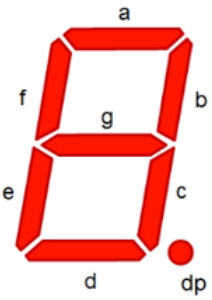
3. 数码管模块

带译码器的七段数码管模块：按照**8421码**输入，数码管即可显示对应数字（0-9）
无带译码器的七段数码管模块：相当于七个发光二极管的组合，输入为七个控制端

插线孔上输入高电平，相应的数码段就会变亮


带译码的七段数码管模块


无译码的七段数码管模块



注意：用本文档中的 `decoder.sv` 实现无带译码器的数码管时，连线遵循“**高位**”连 `A` 的原则

```
module decoder (
    input wire [3:0] sw,
    output reg [6:0] seg
    );

    always_comb begin
        case (sw)
            4'd0: seg = 7'b1111110;
            4'h1: seg = 7'b0110000;
            4'h2: seg = 7'b1101101;
            4'h3: seg = 7'b1111001;
            4'h4: seg = 7'b0110011;
            4'h5: seg = 7'b1011011;
            4'h6: seg = 7'b1011111;
            4'h7: seg = 7'b1110000;
            4'h8: seg = 7'b1111111;
            4'h9: seg = 7'b1110011;
            4'ha: seg = 7'b1110111;
            4'hb: seg = 7'b0011111;
            4'hc: seg = 7'b1001110;
            4'hd: seg = 7'b0111101;
            4'he: seg = 7'b1001111;
            4'hf: seg = 7'b1000111;
            default: seg = 7'b0;
        endcase
    end
endmodule
```

4. 可编程模块（XC7A35）

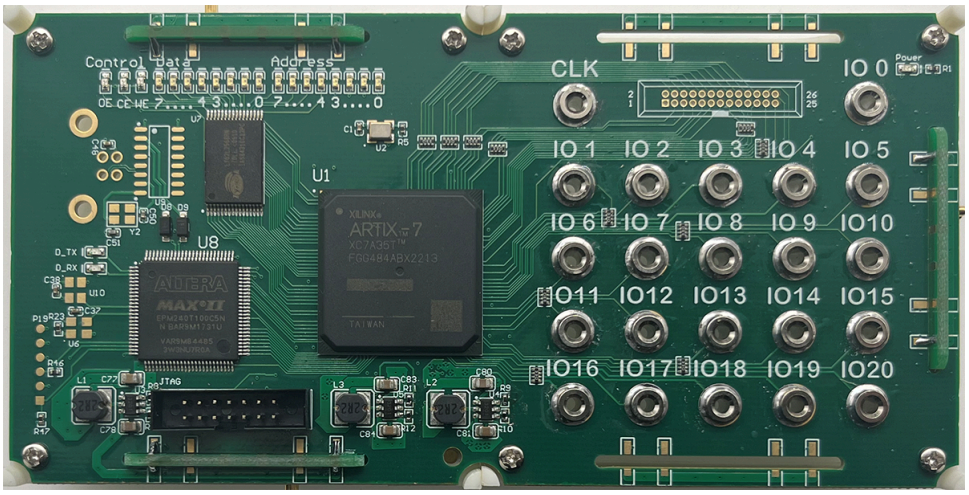
主芯片：可编程逻辑器件FPGA，型号为**XC7A35TFGG484**

下载插座：JTAG的2×7下载插座，可以连接对应下载器，将综合后的Bitstream文件从PC端下载到FPGA中

SRAM芯片：在FPGA的左上方，容量为256K × 8bit；上方的一排发光二极管分别连接到了内存的控制线Control（`nCE`，`nOE`，`nWE`），数据线（`DQ0` ~ `DQ7`）和地址线（`A0` ~ `A17`）

`CLK`：用于连接外部输入的时钟

`IO0`：用于连接外部输入的复位信号



2 示波器的使用

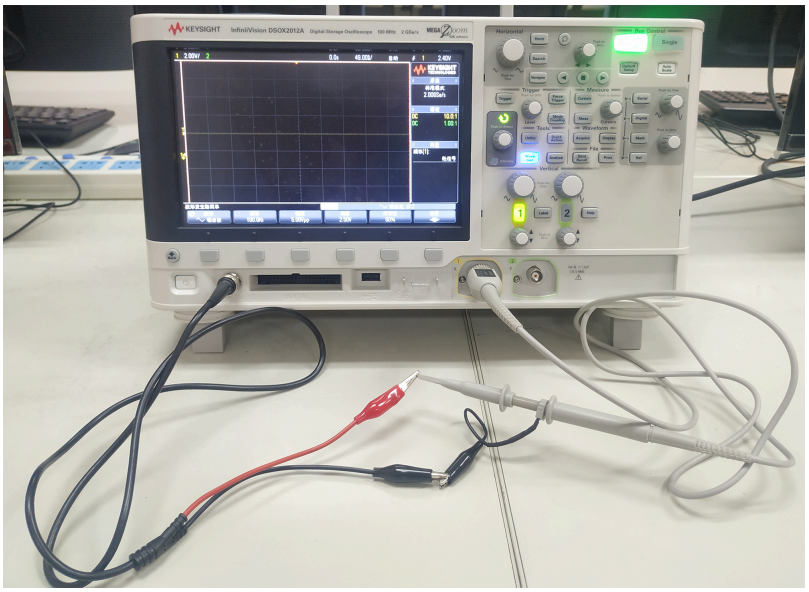
1. 探头校准

① 示波器探头连接到`Demo 2`端子上，探头的黑色架子与示波器中间的**接地端子**链接
② 按下`Auto Scale`查看波形，查看右侧`通道`栏中两个通道的倍率是否与实际探头的衰竭倍率一致

其中，`Demo 2`端子输出的是探头补偿信号，接地端子专门是为了`Demo`所准备的接地端子

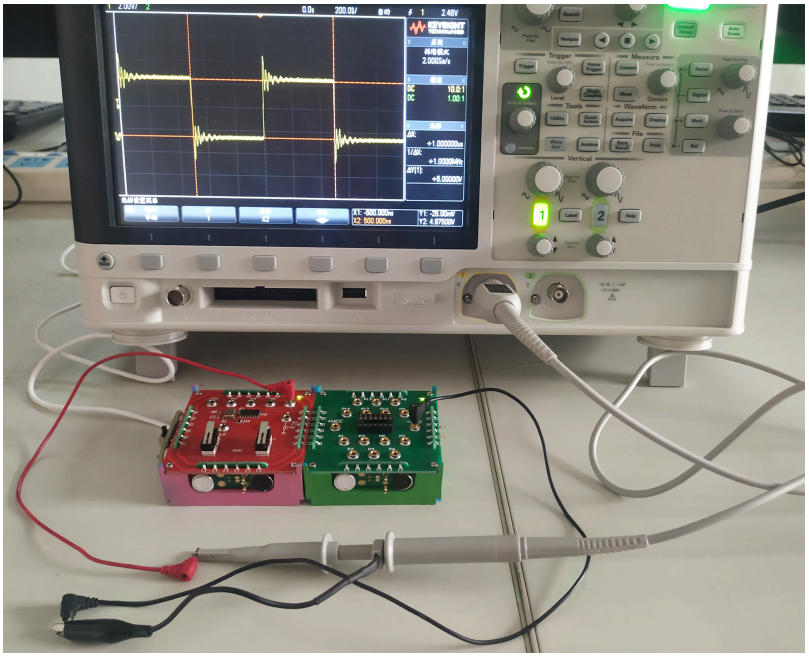
2. 示波器内置信号发生器

按下示波器的`Wave Gen`按键，根据屏幕下方软件设置生成需要的波形（可选的包括：波形、频率、幅度）；调整好之后使用`Auto Scale`按键，即可看到稳定的输出信号波形



3. 时钟模块

需要注意的是：地线连接GND！（这么做的目的是让模块和示波器共地）



4. 测量

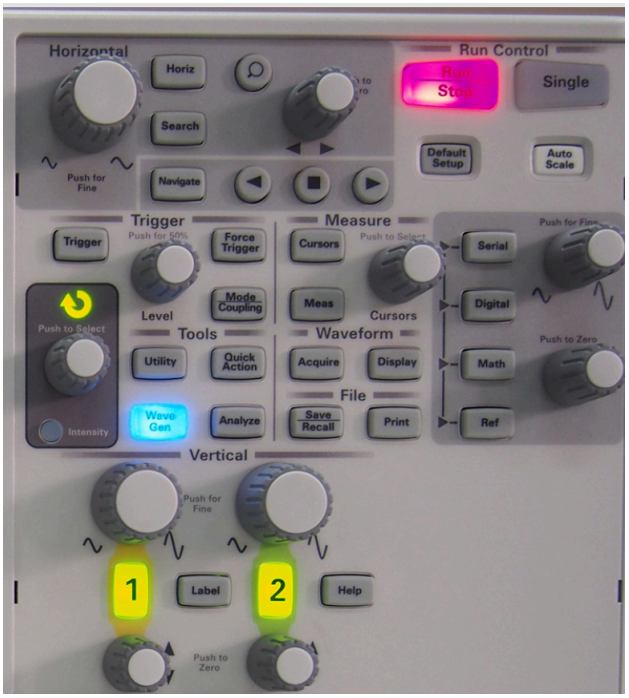
按下示波器上的`Cursors`按钮，使用`X1`、`X2`、`Y1`、`Y2`四条光标进行测量，在屏幕右侧的`光标`栏中就可以看到数据结果

也可以通过`Meas`键自动测量，但可能数据不准确

`Trigger`：触发设置。通过设置触发电压，可以选择画波形的时间节点。如果屏幕上显示的波形抖动不定，有可能是因为出发电压高于波形的最大电压值。触发信号在屏幕上用`T`表示。

一般使用`Auto Gen`即可。

- **与非门平均延迟测量**：即输入源上升沿到输出源下降沿的延迟
- **电压传输特性测量**：输出需要连接时钟模块上的10k电阻，需要通过`Horiz`按钮进入`水平设置菜单`，选择`时基模式`为`XY`，波形会显示成**传输特性曲线**（即通道2（输出电压）相对于通道1（输入电压）的函数关系）



3 加法器

3.1 组合逻辑电路实现

补码表示法（Two's complement）：正数的补码就是其原码；负数的补码是其绝对值原码按位取反+1（可以将符号位理解为负权值）；符号位`0`代表正数，`1`代表负数

补码表示法的好处：① 消除了零的重复表示，② 将加法和减法运算统一到相同的硬件电路进行，消除了符号位的特殊处理

原码表示法：即绝对值的二进制+符号位

以下为两位二进制数的"补码-原码"真值表：

C_1	Y_1	Y_0	D	C_1'	Y_1'	Y_0'
0	0	0	0	0	0	0
0	0	1	1	0	0	1
0	1	0	2	0	1	0
0	1	1	3	0	1	1
1	0	1	-3	1	1	1
1	1	0	-2	1	1	0
1	1	1	-1	1	0	1

1. 两位全加器

> 输入两个两位数A_1A_0（进位C_{-1}）以及B_1B_0，输出加法结果Y_1Y_0以及进位结果C_1

① 首先实现一位全加器（半加器，即两个单比特的二进制数相加，产生一个和值和一个进位制）
② 讲两个一位全加器组合称为两位全加器

2. 补码结果减法器

> 输入两个两位数A_1A_0（进位C_{-1}）以及B_1B_0，输出减法结果F_1F_0以及符号位结果C_1

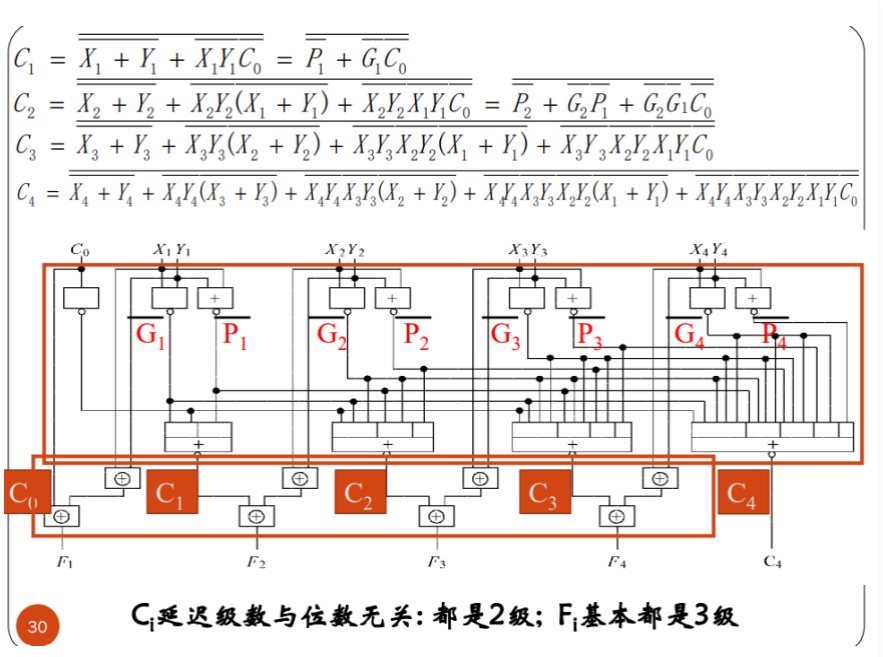
将B逐位取反、全加器进位C_{-1}逐位取反接入两位全加器，结果处的进位结果C_1取反即可

3. 原码结果减法器

基于补码结果减法器，将结果的不嘛转换成原码即可

3.2 FPGA实现

1. **半加器**（Half Adder）：不考虑来自低位的进位信号，其输入为1bit的被加数和加数，输出为两位（本位的和以及向高一位的进位）
2. **全加器**（Full Adder）：输入为被加数、加数以及低一位来的进位，输出为本位的和及向高一位的进位
    - **逐次进位加法器**（Ripple Adder）：利用全加器级联可以构成多位二进制加法器
    - **超前进位加法器**（Carry Look-Ahead Adder）




```
`include "Decoder.sv"
module Adder_sv(
        input wire [3:0] A,
        input wire [3:0] B,
        input wire Cin,
        output reg [3:0] seg,
        output reg [3:0] seg2
    );
    wire [4:0] F;
    wire Cout;
    assign {Cout, F} = A + B + Cin; // Cout useless here

    Decoder decoder (
        .sw(F),
        .seg(seg),
        .seg2(seg2)
    );
```

```
endmodule

module Decoder (
        input wire [4:0] sw, //拨动开关输入
        output reg [3:0] seg, //四段数码管输出（自带译码）
        output reg [3:0] seg2
    );
    always_comb begin
        if (sw < 10) begin
            seg = sw;
            seg2 = 0;
        end
        else if (sw < 20) begin
            seg = sw - 10;
            seg2 = 1;
        end
        else if (sw < 30) begin
            seg = sw - 20;
            seg2 = 2;
        end
        else begin
            seg = sw - 30;
            seg2 = 3;
        end
    end
endmodule
```

4 Vivado开发流程

> 1. Vivado不支持中文目录，务必不要将项目放到中文目录下使用！！否则会综合不过！！
> 2. 芯片型号：`xc7a35tfgg484-2`

综合（$synthesis$）→ 实现（$implementation$）→ 生成比特流（$bitstream$）

LUT和FF门电路构建 → 元器件放置即连线优化 → 可以被加载到$FPGA$上的比特流

其中，implementation的部分需要完成管脚绑定constraints

有时可能需要仿真（simulation）

- **仿真**

  延时不可综合，但是在仿真中非常有用，如：`assign #5 z1 = ~a`（延迟5ns）

  ```
  'timescale 1ns/1ps
  ```

  使用`Test Bench`进行仿真：可以用Verilog HDL的嵌入内部命令，如`$display`，`$monitor`和`$readmemh`等用于调试

5 System Verilog 语法

PLD（可编程逻辑器件）: Programmable Logic Device
CPLD（复杂课程实话逻辑装置）: Complex Programmable Logic Device
FPGA（现场可编程门阵列）: Field-Programmable Gate Array

1. 基于**查找表**（Look-Up-Table / LUT）的原理与结构
   本质上就是一个RAM，软件自动计算逻辑电路所有的可能结果事先写入RAM
   输入信号进行逻辑运算就等于输入地址进行查表，找出地址对应的内容，然后输出即可
2. **触发器**（Flip-Flop / FF）结构（寄存器）

5.1 基本语法

1. 四值电平逻辑

`0`（逻辑0或"假"），`1`（逻辑1或"真"），`x`或`X`（未知），`z`或`Z`（高阻态）

2. 整数数值表示方法

用`'d`，`'h`，`'b`，`'o`表示十、十六、二、八进制
如：`4'b1011`是一个4bit数值，`32'h3022_c0de`是一个32bit数值

3. 数据类型

`wire`（`net`型）：硬件单元之间的物理连线，驱动方式一般为：① `assign`进行持续赋值，② 连到一个module的输入输出端

`reg`（`variable`型）：表示存储单元；保持数据原有的值直到被改写；`always`，`initial`等过程块内被赋值的信号必须定义为此类variable型

`logic`：通用地代替`wire`和`reg`，是System Verilog中的特有关键字，类似C++中的`auto`
`parameter`：可以理解为C++中的`define`宏定义，并不真正开辟存储空间

4. 模块声明

以下为一位全加器的示例：

```
module full_adder(A, B, Cin, F, Cout);
    input wire A, B, Cin;
    output wire F, Cout;

    assign F = A ^ B ^ Cin;
    assign Cout = (A & B) | (Cin & (A ^ B));
endmodule
```

5. 运算符及其优先级

见第五讲 system verilog入门（二）.pdf，page8-12

5.2 语法进阶

1. 组合逻辑与连续赋值 assign

```
assign LHS_target = RHS_expression;
```

`LHS_target` 必须是一个标量或者线型向量，而**不能是 `reg` 类型**
`RHS_expression` 的类型没有要求，可以是标量或线型或存器向量，也可以是函数调用

只要 `RHS_expression` 表达式的值变化，`RHS_expression` 就会立刻重新计算，同时赋值给 `LHS_target`

2. 过程语句 always，initial

`always` 语句会不断循环重复执行，`initial` 语句仅执行一次，用于仿真中的初始化
`always`，`initial` 等过程块内**被赋值的信号**必须定义为 `variable` 型（**不能使用 `wire` 类型**）

- **initial**：写仿真的时候通过 `initial` 赋初值！
- **always** 过程块：当敏感列表中的事件发生时，语句被执行

  ```
  always @ (sensitivity list)
      statement;
  ```

  - 基于触发器的时序逻辑：`always_ff @ (posedge clk)`
    - 基于时序逻辑的 `always` 语句块内**不使用 `assign` 语句**
    - `always` 语句块内采用**非阻塞赋值 `<=`**（non-blocking）：整个过程块结束后才完成赋值操作
      对比**阻塞赋值 `=`**（blocking）：语句结束时立即完成赋值操作

      ——**时序逻辑使用非阻塞赋值，组合逻辑使用阻塞赋值**

      阻塞赋值和非阻塞赋值的仿真波形区别见第6讲 system verilog入门（三）.pdf, page12, 16
      阻塞赋值只在"同一个 `always` 块中出现多个阻塞赋值"时有影响，此时赋值代码的顺序很重要
  - 组合逻辑：`always_comb`
    组合逻辑上可以使用连续赋值语句 `assign`

    不带有记忆效应的触发器：`always @ (inv, data)`（在inv和data电平变化时触发），相当于 `always_comb`，也建议写成 `always_comb`
  - **不允许在不同的always过程块中给相同的信号赋值**
  - 可以有多个 `always` 过程块

```
module example (input            clk,
                input      [3:0] d,
                output reg [3:0] q);

  wire [3:0] normal;          // standard wire
  reg  [3:0] special;         // assigned in always

  always_ff @ (posedge clk)
    special <= d;             // first FF array

  assign normal = ~special; // simple assignment

  always_ff @ (posedge clk)
    q <= normal;              // second FF array
endmodule
```

> 如果special本来是0（normal是1，q是1），d本来是0，然后某一个时刻d变成了1，那么在紧接着的时钟上沿：special变成1，q维持1，normal变成0，然后在下一个posedge clk，q才会变成0

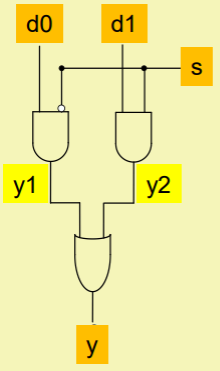
这一切的顺序是：clk上升沿到来 => 寄存器值切换 => 组合逻辑计算完毕 => 寄存器下一输入准备完成 => clk上升沿到来，其中假设d通过时钟触发组合逻辑重新计算

在这个例子里，就是：clk上升沿到来 => speical更新为原d值，q更新为原normal值 => d更新 => special完成更新为d的准备（等到clk一来就更新）=> 问题中的clk上升沿到来 => special更新为新d值，normal更新为新special值，q仍然未原normal值，但完成了新normal的准备

3. 三种描述风格

- **结构描述**（不常用）

```
module mux2(input wire d0, d1,
            input wire s,
            output wire y);

   wire ns, y1, y2;

   not  g1 (ns, s);
   and  g2 (y1, d0, ns);
   and  g3 (y2, d1, s);
   or   g4 (y, y1, y2);

endmodule
```



内置门元件见第五讲 system verilog入门（二）.pdf，page25-27

- **行为描述**：最常用

```
always_comb begin
    if (!s) y = d0;
    else y = d1;
end
```

- **数据流描述**：`assign y = d0 & !s | s1 & s`

5.3 时序逻辑电路描述

1. 同步复位与异步复位

- **异步复位**

  复位信号的采样独立于时钟，具有最高的优先权：

  ```
  always_ff @ (posedge clk, negedge rst) begin
      if (rst == 0) q <= 0; // when reset, reg q
      else q <= d; // when clk
  end
  ```

  异步只能置**常数**!

- **同步复位**

  复位信号是相对于时钟进行采样的；复位应该有足够长的激活时间，以便在时钟边沿采样成功
  如下例中，复位只在时钟上升时发生：

  ```
  always_ff @ (posedge clk) begin
      if (rst == 0) q <= 0; // when reset, reg q
      else q <= d; // when clk
  end
  ```

2. 锁存器问题 latch

一句话理解锁存器：组合逻辑的“触发”（这种情况下极有可能没有讨论`else`而**被综合成锁存器**（边沿触发器）），即组合逻辑因为缺少`else`而被迫拥有了时序逻辑`hold`的特性——这种电路没法自动分析时序是否有问题

```
always @ (clk, d)
    if (clk) q <= d;
// 被综合成锁存器：
always_latch
    if (clk) q <= d;
```

以下情况会被综合成锁存器（使用`always`）：

- 组合逻辑（`always_comb`）中`if`语句没有`else`，或者`case`的条件不能够完全列举且没有`default`，或者`if`和`case`中的输出变量赋值给自己——会报Warning
  解决方法：块的开始处先对输出变量给出缺省赋值

  与组合逻辑不同，寄存器带有存储功能，因此默认维持输出原值的逻辑没有问题，**不需要**像组合逻辑那样覆盖全部可能的分支。

- 组合逻辑用了`always @ ()`语句，但其中的判等`if`变量不在sensitivity list里面
  `always @ (data) begin if (enable)... end`
  解决方法：写成`always_comb`，不需要写敏感信号表（相比于只写`always`，在写成锁存器的时候会报Warning）

System Verilog将`always`细化成了`always_comb`，`always_latch`和`always_ff`，使得前后二者如果出现问题被综合成锁存器的时候会报Warning，提示修改，减少了写成latch的可能性

5.4 管脚绑定

示例

用模拟考试（摩尔斯电码）的管脚绑定文件作为示例（部分改动，便于考场上复制黏贴）：

```
#CLK input
set_property -dict {PACKAGE_PIN J19 IOSTANDARD LVCMOS33} [get_ports CLK];
#CLK接插孔

#RST input
set_property -dict {PACKAGE_PIN K18 IOSTANDARD LVCMOS33} [get_ports RST];
#IO0接插孔

# Code input
set_property -dict {PACKAGE_PIN M21 IOSTANDARD LVCMOS33} [get_ports Code[4]];
#IO1接插孔
set_property -dict {PACKAGE_PIN N20 IOSTANDARD LVCMOS33} [get_ports Code[3]];
#IO2接插孔
set_property -dict {PACKAGE_PIN N22 IOSTANDARD LVCMOS33} [get_ports Code[2]];
#IO3接插孔
set_property -dict {PACKAGE_PIN P21 IOSTANDARD LVCMOS33} [get_ports Code[1]];
#IO4接插孔
set_property -dict {PACKAGE_PIN P22 IOSTANDARD LVCMOS33} [get_ports Code[0]];
#IO5接插孔

set_property -dict {PACKAGE_PIN T21 IOSTANDARD LVCMOS33} [get_ports gear[2]];
#IO6接插孔
set_property -dict {PACKAGE_PIN U21 IOSTANDARD LVCMOS33} [get_ports gear[1]];
#IO7接插孔
set_property -dict {PACKAGE_PIN R21 IOSTANDARD LVCMOS33} [get_ports gear[0]];
#IO8接插孔

# Mode input
set_property -dict {PACKAGE_PIN W21 IOSTANDARD LVCMOS33} [get_ports Mode[2]];
#IO11接插孔
set_property -dict {PACKAGE_PIN W22 IOSTANDARD LVCMOS33} [get_ports Mode[1]];
#IO12接插孔
set_property -dict {PACKAGE_PIN Y22 IOSTANDARD LVCMOS33} [get_ports Mode[0]];
#IO13接插孔

# dmorse output
set_property -dict {PACKAGE_PIN Y21 IOSTANDARD LVCMOS33} [get_ports dmorse[6]];
  #IO14接插孔
set_property -dict {PACKAGE_PIN AB22 IOSTANDARD LVCMOS33} [get_ports dmorse[5]];
    #IO15接插孔
set_property -dict {PACKAGE_PIN AA18 IOSTANDARD LVCMOS33} [get_ports dmorse[4]];
    #IO16接插孔
set_property -dict {PACKAGE_PIN AB18 IOSTANDARD LVCMOS33} [get_ports dmorse[3]];
    #IO17接插孔
set_property -dict {PACKAGE_PIN AA20 IOSTANDARD LVCMOS33} [get_ports dmorse[2]];
    #IO18接插孔
set_property -dict {PACKAGE_PIN AB21 IOSTANDARD LVCMOS33} [get_ports dmorse[1]];
    #IO19接插孔
set_property -dict {PACKAGE_PIN AA21 IOSTANDARD LVCMOS33} [get_ports dmorse[0]];
    #IO20接插孔
```

```
# required if touch button used as manual clock source
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets CLK_IBUF]

set_property CFGBVS VCCO [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]
```

其中，查阅`可编程模块`的`接插孔`信息，可以得到FPGA具体的管脚链接表

6 有限状态机

1. 四种描述方式

> 描述**现态（CS）**逻辑：
>
> ```
> always_ff @(posedge CLK, posedge RST) begin
>     if (RST)    state <= S0;
>     else        state <= next_state;
> end
> ```
>
> 描述**次态（NS）**逻辑：
>
> ```
> always_comb begin
>     case(state)
>         S0:
>             if (x)  next_state <= S1;
>             else    next_state <= S0;
>         // ...
>         defaut:     next_state <= S0;
>     endcase
> end
> ```
>
> 描述**输出（OL）**逻辑：
>
> ```
> always_comb begin
>     case(state)
>         S3:
>             out = 1'b1;
>         default: out = 1'b0;
>     endcase
> end
> // or simply: assign out = (state == S3);
> ```

- 三过程描述：现态（CS）+ 次态（NS）+ 输出逻辑（OL）各用一个always过程描述
- 双过程描述：（CS+NS）+ OL，一个过程描述现态和次态的时序逻辑，另一个描述输出逻辑
- 双过程描述：CS +（NS+OL），一个过程用来描述现态，另一个描述次态和输出逻辑
- 单过程描述：放在一个`always`过程中进行描述

2. 状态编码

顺序编码（binary code），一位热码（one-hot code），格雷码（gray code）...

	State Variables		
State	**One-Hot Code**	**Binary Code**	**Gray Code**
S0	00001	000	000
S1	00010	001	001
S2	00100	010	011
S3	01000	011	010
S4	10000	100	110

Table 1:An example of state Encoding for a 4 state Machine

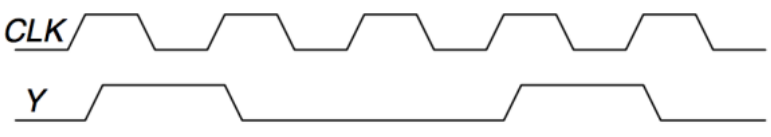
7 计数器实验

1. 时钟分频

> 由时钟模块产生`1M`的时钟，输出其十分频到示波器上显示

```
module FenPin( // 使用单过程状态机实现
    input wire CLK,
    input wire RST,
    output reg result
    );

    reg [4:0] count; // better: use integer! at least NOT REG, but REG [4:0]

    always_ff @(posedge CLK or posedge RST) begin // 异步复位，不要同时响应时钟的上边沿和下边沿
        if (RST) begin
            result <= 0;
            count <= 0; // IMPORTANT!
        end
        else begin
            if (count == 4) begin // 4 = 10 / 2 - 1
                count <= 0;
                result <= ~result;
            end else count <= count + 1;
        end
    end

endmodule
```

> 三分频：在每三个时钟周期中，输出Y为高电平



```
module divideby3FSM (input clk, input reset, output q); // 使用三过程状态机实现
    reg [1:0] state, nextstate;
    parameter S0 = 2'b00; parameter S1 = 2'b01; parameter S2 = 2'b10;

    always_ff @ (posedge clk, posedge reset) // state register, CS
        if (reset)  state <= S0;
        else        state <= nextstate;

    always_comb // next state logic, NS
        case (state)
            S0:         nextstate = S1;
            S1:         nextstate = S2;
            S2:         nextstate = S0;
            default:    nextstate = S0;
        endcase

    assign q = (state == S0); // output logic, OL
endmodule
```

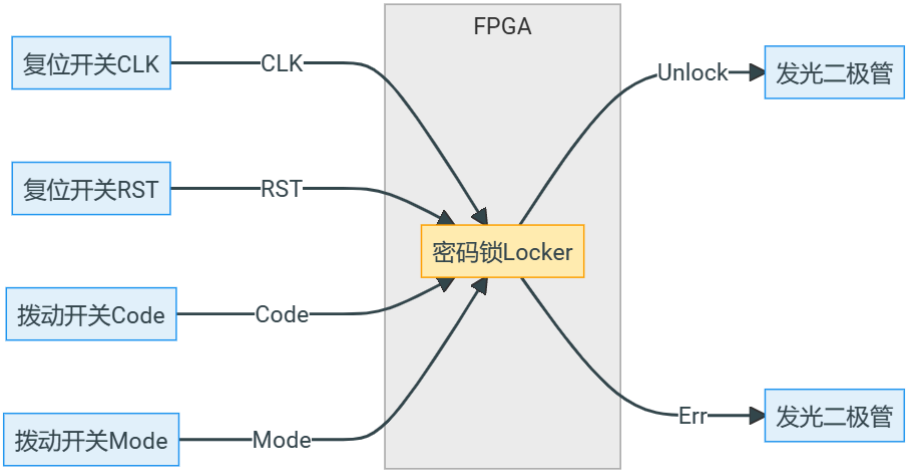
2. 可以控制启动和暂停的计数器

基础功能 + 时钟模块分频 + 开关控制启动 / 暂停

```
always_ff @(posedge CLK) begin // 注意这里！！只能同步复位，不能异步复位，这里相当于赋值变量：cnt <= STOP == 0 ? 0 : cnt（RST上升沿导致的赋值只能把变量设置成预先设定的常数，不能有if之类的——异步只能置常数），如果在复位逻辑中实现了复杂的逻辑，可能会导致latch的生成，综合器也会有警告。
    if (STOP == 0) begin
        if (RST) begin
            cnt_H <= 0;
            cnt_L <= 0;
            cnt <= 0;
        end else begin
            if (cnt == 1000000) begin
                cnt <= 0;
                // counter++
                if (cnt_L == 9) begin
                    if(cnt_H == 5) begin
                        cnt_H <= 0;
                        cnt_L <= 0;
                    end else begin
                        cnt_H <= cnt_H + 1;
                        cnt_L <= 0;
                    end
                end else begin
                    cnt_L <= cnt_L + 1;
                end
            end else cnt <= cnt + 1;
        end
    end
end
```

8 密码锁实验（状态机）

> 设计一个四位 16 进制串行电子密码锁，其具体功能如下：
> （1）设置密码：用户可串行设置四位 16 进制密码；
> （2）验证密码：用户串行输入密码，如密码符合则点亮开锁灯，若不符合则点亮错误灯；
>
> 提高要求：
> （1）密码预置：为管理员创建万用密码以备管理。
> （2）系统报警：开锁三次失败后点亮报警灯，并锁定密码锁，只有输入管理员密码才可开锁，并解除报警。




```
`timescale 1ns / 1ps

module Locker(
    input wire[3:0] Code,
    input wire Mode,
    input wire CLK,
    input wire RST,
    output reg Unlock, // always中被赋值一定要是reg类型不能是wire
    output reg Err,
    output reg Warning
    );

    reg [15:0] token;
    reg [15:0] user_password; // set by user
    parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3, S4 = 4; // 4 stages
    parameter admin_password = 16'hffff;  // administrator's password

    reg [2:0] state, next_state;
    reg [1:0] cnt_failed = 0;

    always_ff @(posedge CLK or posedge RST)
    begin
        if (RST) begin
            state <= S0;
        end else begin
            case(state)
                S0: token[3:0] <= Code;
```

```
                S1: token[7:4] <= Code;
                S2: token[11:8] <= Code;
                S3: token[15:12] <= Code;
                default: ;
            endcase
            state <= next_state;
            if (Err == 1) begin
                if (cnt_failed == 2'b00) cnt_failed <= 2'b01;
                else if (cnt_failed == 2'b01) cnt_failed <= 2'b10;
                else if (cnt_failed == 2'b10) cnt_failed <= 2'b11;
                if (cnt_failed == 2'b11) begin
                    Warning <= 1; // password locked
                end
            end
            if (Unlock == 1) begin
                cnt_failed <= 2'b00;
                Warning <= 0;
            end
        end
    end

    always_comb begin
        case(state)
            S0:  next_state<=S1;
            S1:  next_state<=S2;
            S2:  next_state<=S3;
            S3:  next_state<=S4;
            S4:  next_state<=S4;
            default: next_state<=S0;
        endcase
    end

    always_comb begin
        // the following three statements eliminate latch
        Unlock = 0;
        Err = 0;
        real_password = 0;
        if (Mode == 1) begin // validate password
            if (state == S4 ) begin
                if (token == real_password || token == master_password) begin
                    Unlock = 1;
                    Err = 0;
                end
                else begin
                    Unlock = 0;
                    Err = 1;
                end
            end
        end else if (Mode == 0) begin // set password
            if (state == S4) begin
                Unlock = 1;
                Err = 1; // indicates that the password has been set
                real_password = token;
            end
        end
    end
```

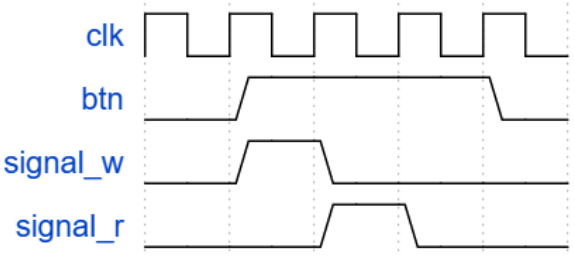
```
endmodule
```

思考：每次切换模式`Mode`输入的时候，需要手动按下`RST`来保证逻辑正确，是否有方法可以自动在改变`Mode`的时候进行`RST`呢?

答：采用同步时序电路设计

简化问题：生成一个信号，当按钮信号（对应`Mode`）从0变为1之后，保持高电平1个周期，其余时刻保持为低电平

```
input wire btn,

reg prev_btn;

always_ff @(posedge clk) begin
    prev_btn <= btn;
end

reg signal_w,  signal_r;

// 组合逻辑
always_comb begin
    if (prev_btn == 1'b0 && btn == 1'b1) begin
        signal_w = 1;
    end else begin
        signal_w = 0;
    end
end

// 时序逻辑
always_ff @(posedge clk) begin
    if (prev_btn == 1'b0 && btn == 1'b1) begin
        signal_r <= 1;
    end else begin
        signal_r <= 0;
    end
end
```



9 摩尔斯密码（模拟题）

> 题目具体见：模拟实验考试 - 数字逻辑实验
>
> - **步骤 1（5 分）：译码器**
>   本步骤请实现不带译码的数码管的译码模块。
>   输入：由 4 位拨码开关进行控制（0000 到 1111）。
>   输出：在数码管上对应显示 0 到 F。
> - **步骤 2（15 分）：简单摩斯电码翻译**
>   本步骤中我们将实现摩斯电码的翻译。
>   实现本步骤默认步骤 1 已经完成。
>   输入：两位拨码开关，00 表示短按，01 表示长按，10 表示结束。CLK 用于进行编码输入，每次 CLK 按下表示输入 1 位摩斯电码 / 结束输入。
>   输出：1 位 LED 灯，表示是否为错误的（表外的）摩斯电码。7 位不带译码的数码管，在完成输入时，显示对应的数字。
>   验收时将随机进行两个数字和 1 个字母的检查。
> - **步骤 3（10 分）：长按和短按**
>   本步骤中，我们将实现短按和长按的判定。
>   利用时钟和计数寄存器，我们可以在电路中完成按钮按下的时间的判断。
>   当按钮按下后（之前为 0，现在为 1），使用同步的方式（响应时钟而不是按钮）将计数寄存器清零。
>   当按钮持续按下时，每次时钟上升沿到来，将计数器加 1。
>   当计数器大于某个值时，激活长按灯，否则激活短按灯。
>   输入：1M 时钟，CLK 按钮。
>   输出：两个 LED 灯，分别表示长按和短按。
> - **步骤 4（5 分）：完整的摩斯电码译码**
>   实现本步骤默认步骤 1-3 已经完成。
>   本步骤中，我们将实现一个较为综合的摩斯电码译码。
>   通过按钮，按下一系列由长按和短按组成的信号序列，将译码结果输出在不带译码的数码管上。若不匹配，则输出错误信号。
>   按下 RST 时，重置状态机。
>   请仔细思考状态机进行转移的条件。
>   输入：1M 时钟，CLK 按钮，RST 按钮
>   输出：1 位 LED 灯，表示是否为错误的（表外的）摩斯电码。7 位不带译码的数码管，在完成输入时，显示对应的数字。
> - **步骤 5（5 分）：示波器的使用**
>   请将输入的 1M 时钟，通过 FPGA 进行 6 分频（即输出的频率为输入的频率的 1/6），将两个信号接到示波器的两个通道上进行显示。

直接给出完整版代码：

```
// counter
module counter(
    input wire clk, // input from 1M timer
    input wire press, // input from user
    output reg [1:0] led // display whether long press or short press
    );

    reg [31:0] count;
    reg prev_btn;
    reg signal; // when press down the btn
```

```
    initial begin
        count <= 32'b0;
        prev_btn <= 1'b0;
        signal <= 1'b0;
    end

    always_ff @(posedge clk) begin
        prev_btn <= press;
    end

    always_comb begin
        if (prev_btn == 1'b0 && press == 1'b1) signal <= 1'b1;
        else signal <= 1'b0;
    end

    always_ff @(posedge clk) begin
        if (signal) begin
            led <= 2'b00;
            count <= 32'b0;
        end
        else begin
            count <= count + 1;
            if (press && count >= 32'd1500000) led <= 2'b11; // long press
            else if (press) led <= 2'b01; // short press
            else led <= 2'b00; // not pressing
        end
    end
endmodule
```

```
`timescale 1ns / 1ps

module smart_morse(
    input wire clk, // input from 1M timer
    input wire rst,
    input wire press,
    output reg [1:0] led, // whether long press(11) or short press(01)
    output reg error, // indicating whether the input is legal
    output reg done, // indicating the end of the whole morse
    output reg [6:0] dmorse // displaying numbers
    );

    reg [3:0] number;
    reg signal; // indicating the end of pressing
    reg prev_btn;

    reg cur_input;
    reg [0:4] morse;
    reg [2:0] state; // counter of input
    reg [31:0] count; // counter of time

    decoder mydecoder (
        .sw(number),
        .seg(dmorse)
    );
```

```
    counter mycounter (
        .clk(clk),
        .press(press),
        .led(led)
    );

    initial begin
        number <= 0;
        error <= 0;
        prev_btn <= 0;
        state <= 0;
    end

    // these two blocks are use to derive signal
    always_ff @(posedge clk) begin
        prev_btn <= press;
    end
    always_comb begin
        if (prev_btn == 1'b1 && press == 1'b0) signal <= 1'b1;
        else signal <= 1'b0;
    end

    // don't use latch
    always_ff @(posedge clk) begin
        if (led != 0) begin
            case (led)
                2'b11: cur_input = 1;
                2'b01: cur_input = 0;
            endcase
        end
    end

    // judge input stage (including judge an end): state, done, count
    always_ff @(posedge clk) begin
        if (rst) begin
            state <= 0;
            done <= 0;
            count <= 0;
        end
        else if (signal) begin // if just finish inputing one word
            morse[state] <= cur_input;
            state <= state + 1;
        end
        else begin
            if (press) count <= 0; // 'count' here used for counting blank time
interval between input
            else count <= count + 1;
            if (!press && count >= 32'd4000000) done <= 1'b1; // doing nothing
for 4s => done
            else done <= 1'b0;
        end
    end

    // assign value to input: error, number
    always_comb begin
```

```
        error = 0;
        number = 0;
        if (done) begin
            if (state == 0 || state > 5) error = 1;
            else begin
                case (state)
                    1: begin
                        if (morse[0] == 0) number = 4'hE;
                        else error = 1;
                    end
                    2: begin
                        if (morse[0] == 0 && morse[1] == 1) number = 4'hA;
                        else error = 1;
                    end
                    3: begin
                        if (morse[0] == 1 && morse[1] == 0 && morse[2] == 0)
number = 4'hD;
                        else error = 1;
                    end
                    4: begin
                        if (morse[0] == 1 && morse[1] == 0 && morse[2] == 0 &&
morse[3] == 0) number = 4'hB;
                        else if (morse[0] == 1 && morse[1] == 0 && morse[2] == 1
&& morse[3] == 0) number = 4'hC;
                        else if (morse[0] == 0 && morse[1] == 0 && morse[2] == 1
&& morse[3] == 0) number = 4'hF;
                        else error = 1;
                    end
                    5: begin
                        case (morse)
                            5'b11111: number = 4'h0;
                            5'b01111: number = 4'h1;
                            5'b00111: number = 4'h2;
                            5'b00011: number = 4'h3;
                            5'b00001: number = 4'h4;
                            5'b00000: number = 4'h5;
                            5'b10000: number = 4'h6;
                            5'b11000: number = 4'h7;
                            5'b11100: number = 4'h8;
                            5'b11110: number = 4'h9;
                            default: error = 1;
                        endcase
                    end
                    default: error = 1;
                endcase
            end
        end
    end
endmodule
```

管脚约束文件：

```
# clk input
set_property -dict {PACKAGE_PIN J19 IOSTANDARD LVCMOS33} [get_ports clk];
#clk接插孔
```

```
# rst input
set_property -dict {PACKAGE_PIN K18 IOSTANDARD LVCMOS33} [get_ports rst];
#IO0接插孔

# press input
set_property -dict {PACKAGE_PIN R21 IOSTANDARD LVCMOS33} [get_ports press];
#IO8接插孔

# error output
set_property -dict {PACKAGE_PIN W21 IOSTANDARD LVCMOS33} [get_ports error];
#IO11接插孔

# done output
set_property -dict {PACKAGE_PIN M21 IOSTANDARD LVCMOS33} [get_ports done];
#IO12接插孔

# led output
set_property -dict {PACKAGE_PIN P21 IOSTANDARD LVCMOS33} [get_ports led[1]];
#IO4接插孔
set_property -dict {PACKAGE_PIN P22 IOSTANDARD LVCMOS33} [get_ports led[0]];
#IO5接插孔

# dmorse output
set_property -dict {PACKAGE_PIN Y21 IOSTANDARD LVCMOS33} [get_ports dmorse[6]];
  #IO14接插孔
set_property -dict {PACKAGE_PIN AB22 IOSTANDARD LVCMOS33} [get_ports dmorse[5]];
    #IO15接插孔
set_property -dict {PACKAGE_PIN AA18 IOSTANDARD LVCMOS33} [get_ports dmorse[4]];
    #IO16接插孔
set_property -dict {PACKAGE_PIN AB18 IOSTANDARD LVCMOS33} [get_ports dmorse[3]];
    #IO17接插孔
set_property -dict {PACKAGE_PIN AA20 IOSTANDARD LVCMOS33} [get_ports dmorse[2]];
    #IO18接插孔
set_property -dict {PACKAGE_PIN AB21 IOSTANDARD LVCMOS33} [get_ports dmorse[1]];
    #IO19接插孔
set_property -dict {PACKAGE_PIN AA21 IOSTANDARD LVCMOS33} [get_ports dmorse[0]];
    #IO20接插孔

# required if touch button used as manual clock source
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets CLK_IBUF]

set_property CFGBVS VCCO [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]
```

10 手动挡自动挡汽车（2022 模拟题）

```
`timescale 1ns / 1ps

module car(
    input wire mode, // 0: manual, 1: auto
```

```
    input wire clk,
    input wire rst,
    input wire manualgear,
    input wire brake, // 刹车
    output reg [3:0] gear, // 档位
    output reg [3:0] speed // 速度
    );

    reg [31:0] time_cnt;
    reg prev_btn;
    reg signal;

    always_ff @(posedge clk) begin
        prev_btn <= manualgear;
    end
    always_comb begin
        if (prev_btn == 1'b0 && manualgear == 1'b1) signal <= 1'b1;
        else signal <= 1'b0;
    end


    always_ff @(posedge clk) begin // speed up
        if (rst) begin
            speed <= 4'b0000;
            time_cnt <= 32'd0;
        end
        else begin
            if (time_cnt == 32'd1000000) begin
                time_cnt <= 32'd0;
                if (brake) begin
                    if (mode == 1 && speed != 4'd0) speed <= speed - 1;
                    else if (mode == 0 && speed > gear * 2 - 2) speed <= speed -
1;
                end
                else begin
                    if (mode == 1 && speed != 4'd9) speed <= speed + 1;
                    else if (mode == 0 && speed < gear * 2 - 1) speed <= speed +
1;
                end
            end
            else time_cnt <= time_cnt + 1;
        end
    end

    always_ff @(posedge clk) begin
        if (rst) begin
            gear <= 4'b0001;
        end
        else if (mode == 0 && signal) begin
            if (gear != 4'd5) gear <= gear + 1;
        end
        else if (mode == 1) begin
            case (speed)
                4'd0: gear <= 4'b0001;
                4'd1: gear <= 4'b0001;
                4'd2: gear <= 4'b0010;
```

```
                4'd3: gear <= 4'b0010;
                4'd4: gear <= 4'b0011;
                4'd5: gear <= 4'b0011;
                4'd6: gear <= 4'b0100;
                4'd7: gear <= 4'b0100;
                4'd8: gear <= 4'b0101;
                4'd9: gear <= 4'b0101;
            endcase
        end
    end
endmodule
```

```
# CLK input
set_property -dict {PACKAGE_PIN J19 IOSTANDARD LVCMOS33} [get_ports clk];
#CLK接插孔

# RST input
set_property -dict {PACKAGE_PIN K18 IOSTANDARD LVCMOS33} [get_ports rst];
#IO0接插孔

# mode input
set_property -dict {PACKAGE_PIN Y21 IOSTANDARD LVCMOS33} [get_ports mode];
#IO14接插孔

# manualgear input
set_property -dict {PACKAGE_PIN AA21 IOSTANDARD LVCMOS33} [get_ports manualgear];
    #IO20接插孔

# brake input
set_property -dict {PACKAGE_PIN AA20 IOSTANDARD LVCMOS33} [get_ports brake];
#IO18接插孔

# speed output
set_property -dict {PACKAGE_PIN M21 IOSTANDARD LVCMOS33} [get_ports speed[3]];
  #IO1接插孔
set_property -dict {PACKAGE_PIN N20 IOSTANDARD LVCMOS33} [get_ports speed[2]];
  #IO2接插孔
set_property -dict {PACKAGE_PIN N22 IOSTANDARD LVCMOS33} [get_ports speed[1]];
  #IO3接插孔
set_property -dict {PACKAGE_PIN P21 IOSTANDARD LVCMOS33} [get_ports speed[0]];
  #IO4接插孔

# gear output
set_property -dict {PACKAGE_PIN P22 IOSTANDARD LVCMOS33} [get_ports gear[3]];
#IO5接插孔
set_property -dict {PACKAGE_PIN T21 IOSTANDARD LVCMOS33} [get_ports gear[2]];
#IO6接插孔
set_property -dict {PACKAGE_PIN U21 IOSTANDARD LVCMOS33} [get_ports gear[1]];
#IO7接插孔
set_property -dict {PACKAGE_PIN R21 IOSTANDARD LVCMOS33} [get_ports gear[0]];
#IO8接插孔

# required if touch button used as manual clock source
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets CLK_IBUF]
```

```
set_property CFGBVS VCCO [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]
```