**序号：**

****

硬件课程设计报告 (2019-2020学年)

姓 名： 章磊

学 号： 19951060119

班 级： 计专起本19

专 业： 计算机科学与技术

所在系： 信息学院

指导教师： 肖蔼玲

2019年 9月 5 日

|  |
| --- |
| **总成绩：**  **评语：**  **指导教师签字： 日期：** |

目录

[一. 数码管显示学号 3](#_Toc50331497)

[**1.1. 实验任务** 3](#_Toc50331498)

[**1.2. 实验现象与结果** 3](#_Toc50331499)

[**1.3. 实验要求** 3](#_Toc50331500)

[二. 数码管显示学号 7](#_Toc50331501)

[**2.1. 实验任务** 7](#_Toc50331502)

[**2.2. 实验现象与结果** 7](#_Toc50331503)

[**2.3. 问题与思考** 7](#_Toc50331504)

[**2.4. 实验报告** 7](#_Toc50331505)

[三. 两位数十进制减法器的 IP 核，多位十进制减法器 18](#_Toc50331506)

[**3.1. 实验任务** 18](#_Toc50331507)

[**3.2. 实现步骤** 18](#_Toc50331508)

[**3.3．实验要求** 18](#_Toc50331509)

[**3.4 问题与思考** 18](#_Toc50331510)

[**3.5. 实验报告** 18](#_Toc50331511)

[四. 序列检测器 23](#_Toc50331512)

[**4.1.序列检测器原理** 23](#_Toc50331513)

[**4.2.状态机** 23](#_Toc50331514)

[**4.3.问题与思考** 23](#_Toc50331515)

[五、简单CPU的设计 35](#_Toc50331522)

[**5.1 实验报告要求** 35](#_Toc50331523)

[六、课设总结 63](#_Toc50331524)

# **一. 数码管显示学号**

## **1.1. 实验任务**

要求将自己的最后三位学号数字在数码管上显示。

## **1.2. 实验现象与结果**

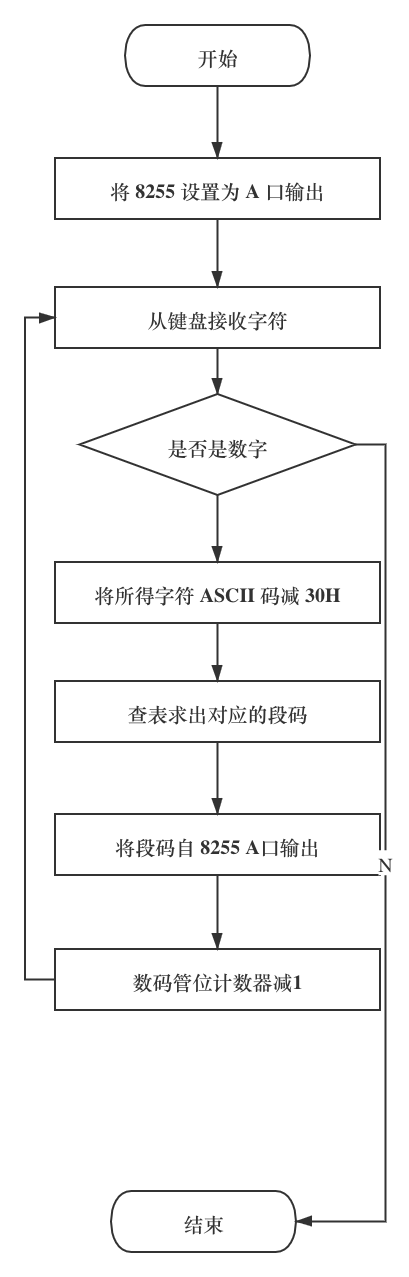
应能看见自己学号稳定的在数码管上显示，并用手机拍摄记录下来。

## **1.3. 实验要求**

1. 简单陈述在数码管上显示数字的原理。

LED显示器工作方式有两种：静态显示方式和动态显示方式。静态显示的特点是每个数码管的段选必须接一个8位数据线来保持显示的字形码。当送入一次字形码后，显示字形可一直保持，直到送入新字形码为止。这种方法的优点是占用CPU时间少，显示便于监测和控制。缺点是硬件电路比较复杂，成本较高。

1. 画出编写Verilog程序的功能框图。



1. 说明改变扫描频率的软件方法，及对显示的影响。

扫描频率大小不许合适才能有很好的效果。如果太小，而每个LED开启的时间大于人眼的视觉暂停时间，那么会产生闪烁现象。而扫描频率太大，则会造成LED的频繁开启和关断，大大增加LED功耗（开启和关断的时刻功耗很大）。一般来说，稍描频率选在50Hz比较合适。

1. 附上Verilog实验源程序，并对关键或重要的程序语句给出简要的注释和说明。

```Verilog

module prj\_1(decodeout1,decodeout2,decodeout3,indec1,indec2,indec3);

output[7:0] decodeout1,decodeout2,decodeout3;

// 定义接受输入的三个学号的变量

input[3:0] indec1,indec2,indec3;

reg[7:0] decodeout1,decodeout2,decodeout3;

always @(indec1)

begin

case(indec1)

4'd0:decodeout1=8'b11111101;

4'd1:decodeout1=8'b01100001;

4'd2:decodeout1=8'b11011011;

4'd3:decodeout1=8'b11110011;

4'd4:decodeout1=8'b01100111;

4'd5:decodeout1=8'b10110111;

4'd6:decodeout1=8'b10111111;

4'd7:decodeout1=8'b11100001;

4'd8:decodeout1=8'b11111111;

4'd9:decodeout1=8'b11110111;

default: decodeout1=8'bx;

endcase

end

always @(indec2)

begin

case(indec2)

4'd0:decodeout2=8'b11111101;

4'd1:decodeout2=8'b01100001;

4'd2:decodeout2=8'b11011011;

4'd3:decodeout2=8'b11110011;

4'd4:decodeout2=8'b01100111;

4'd5:decodeout2=8'b10110111;

4'd6:decodeout2=8'b10111111;

4'd7:decodeout2=8'b11100001;

4'd8:decodeout2=8'b11111111;

4'd9:decodeout2=8'b11110111;

default: decodeout2=8'bx;

endcase

end

always @(indec3)

begin

case(indec3)

4'd0:decodeout3=8'b11111101;

4'd1:decodeout3=8'b01100001;

4'd2:decodeout3=8'b11011011;

4'd3:decodeout3=8'b11110011;

4'd4:decodeout3=8'b01100111;

4'd5:decodeout3=8'b10110111;

4'd6:decodeout3=8'b10111111;

4'd7:decodeout3=8'b11100001;

4'd8:decodeout3=8'b11111111;

4'd9:decodeout3=8'b11110111;

default: decodeout3=8'bx;

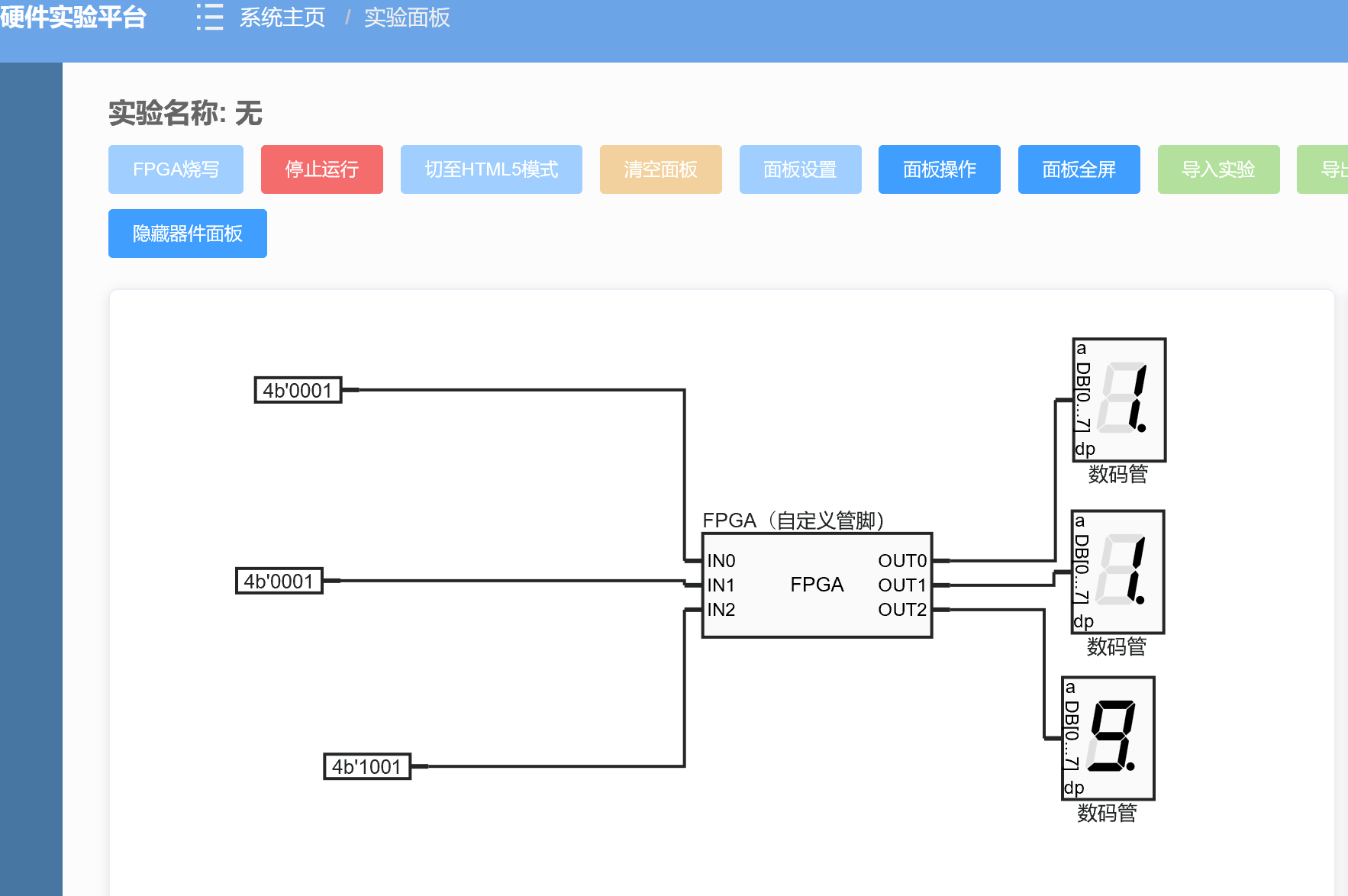
endcase

end

endmodule

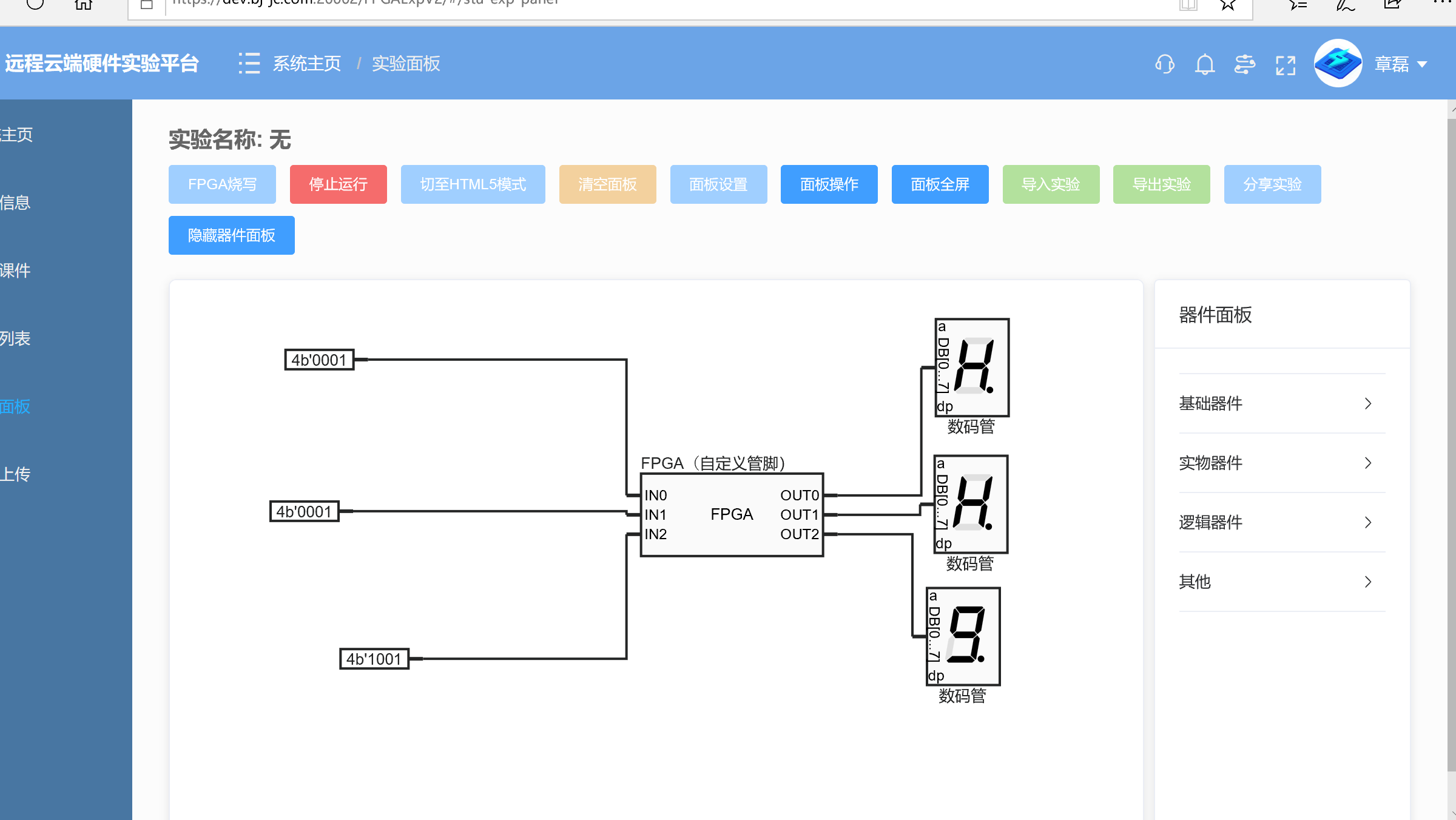
```

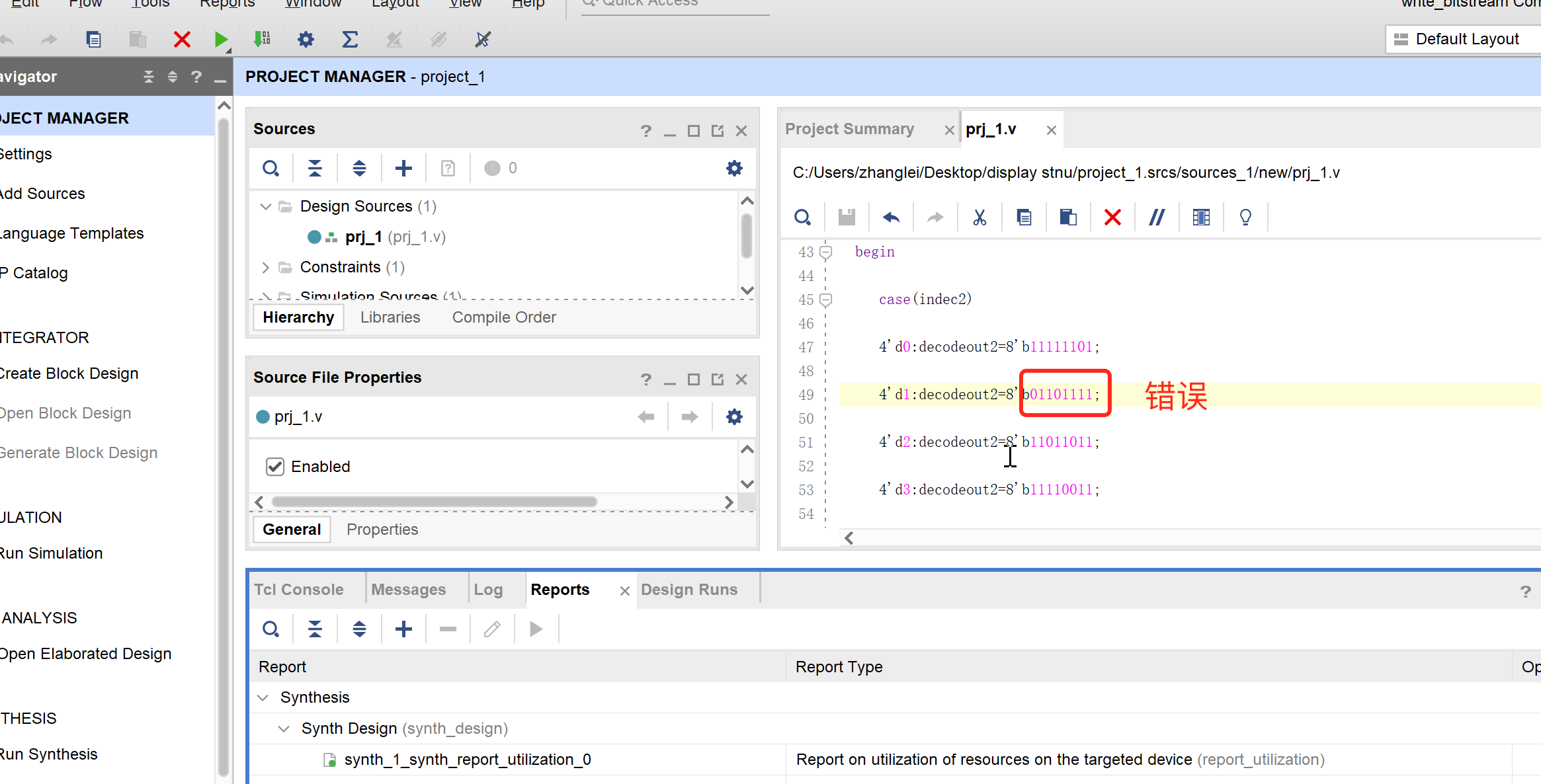
1. 将显示的学号截图附上。



1. 如果调试过程遇到了问题，应将问题截图，并给出解决问题的办法和对应的截图。

一开始段表书写错误，导致段码查询出现错误，在云平台进行项目烧写出现了后，运行试验时，数码管显示不全，后询问同学发现了解决该问题，并将以改正。





# **二. 数码管显示学号**

## **2.1. 实验任务**

在16×16点阵上，编写Verilog程序实现循环显示你自己的“名字”的汉字。

## **2.2. 实验现象与结果**

在点阵模块上循环依次显示你自己的“名字”，每个汉字显示的时间约为0.5秒。

## **2.3. 问题与思考**

* 怎样让汉字左旋转和右旋转。

使用16 \* 16 的矩阵旋转方式可以实现汉字左旋转和右旋转

* 怎样让汉字左移动和右移动。

左移动：将行的 16 位二进制左移 n 位，右边补0

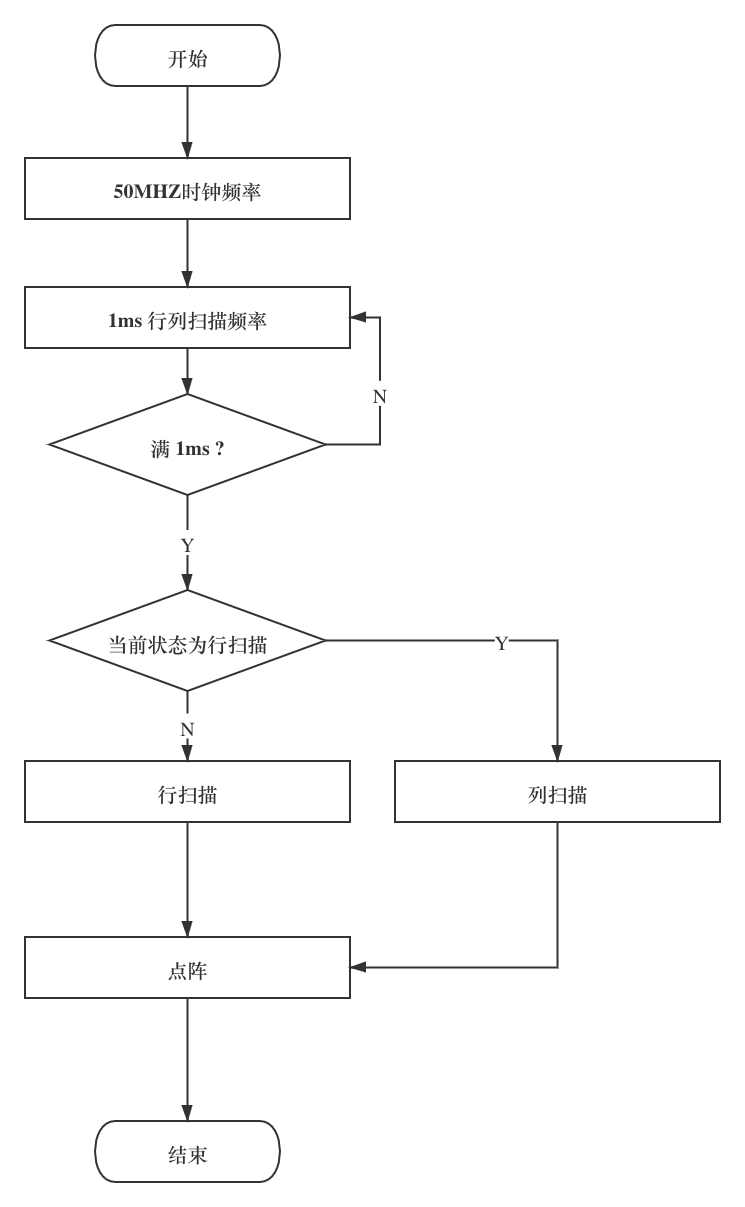
右移动：将行的 16 位二进制右移 n 位，左边补 0

## **2.4. 实验报告**

1. 16×16点阵上显示汉字的原理。

8X8点阵是由一排排的LED灯组成的，它们的阴端连接到行，阳端连接到列，当想要点阵显示什么数字或者字母或者图形的时候，只需要点亮相应的LED就可以了。比如说我们想点亮左上角第一个LED，而其它的都是灭的，我们只需要将行1位置置为1，其它行位置置为0，列1位置置为0，其他位置置为1则可以点亮左上角第一个LED，而其它的不亮。且每个发光二极管是放置在行线和列线的交叉点上，当对应的某一列置1电平，某一行置0电平，则相应的二极管就亮；要实现显示图形或字体，只需考虑其显示方式。通过编程控制各显示点对应LED阳极和阴极端的电平，就可以有效的控制各显示点的亮灭。当在LED点阵上显示中文时，可以用现成的汉字取模工具去生成，而不需要手动去计算。在16x16点阵中，点阵显示是通过4块8x8LED显示的，而最先输入的8x8LED块会被压入最后（右下角块），所以发送字节数据的时候是从最后一块往第一块发。

1. Verilog程序的功能框图。



1. Verilog程序语句。

```Verilog

module lattice(

clk,

rst\_n,

lin,

row

);

parameter TIME05 = 25\_000\_000;//0.5s，名字显示频率

parameter TIME1 = 50\_000;//1ms，行扫描频率

input clk;

input rst\_n;

output[15:0] lin;

output[15:0] row;

reg[15:0] lin;

reg[15:0] row;

reg[24:0] cnt\_05;

reg[24:0] cnt\_1;

reg[4:0] cnt;

reg[3:0] cnt\_reg;

wire add\_cnt\_05;

wire end\_cnt\_05;

wire add\_cnt\_1;

wire end\_cnt\_1;

wire add\_cnt;

wire end\_cnt;

always @(posedge clk or negedge rst\_n)begin

if(!rst\_n)begin

cnt\_05 <= 0;

end

else if(add\_cnt\_05)begin

if(end\_cnt\_05)

cnt\_05 <= 0;

else

cnt\_05 <= cnt\_05 + 1;

end

end

assign add\_cnt\_05 = 1;

assign end\_cnt\_05 = add\_cnt\_05 && cnt\_05== TIME05-1; //0.5s的计数器

always @(posedge clk or negedge rst\_n)begin

if(rst\_n==1'b0)begin

cnt\_reg <= 3'd0;

end

else if(end\_cnt\_05) begin

// 字数个数

if(cnt\_reg == 3'd1)

cnt\_reg <= 3'd0;

else

cnt\_reg <= cnt\_reg + 1'b1;

end

end //当cnt\_reg分别为0，1, 显示章、磊

always @(posedge clk or negedge rst\_n)begin

if(!rst\_n)begin

cnt\_1 <= 0;

end

else if(add\_cnt\_1)begin

if(end\_cnt\_1)

cnt\_1 <= 0;

else

cnt\_1 <= cnt\_1 + 1;

end

end

assign add\_cnt\_1 = 1;

assign end\_cnt\_1 = add\_cnt\_1 && cnt\_1== TIME1-1; //1ms计数器

always @(posedge clk or negedge rst\_n)begin

if(!rst\_n)begin

cnt <= 0;

end

else if(add\_cnt)begin

if(end\_cnt)

cnt <= 0;

else

cnt <= cnt + 1;

end

end

assign add\_cnt = end\_cnt\_1;

assign end\_cnt = add\_cnt && cnt== 16-1;//行扫描计数器

always @(\*)begin

case(cnt\_reg)

// 第一个字

0:begin

case(cnt)

0:begin

lin = 16'b0000\_0000\_0000\_0001;

row = 16'b1111\_1110\_1111\_1111;

;

end

1:begin

lin = 16'b0000\_0000\_0000\_0010;

row = 16'b1100\_0000\_0000\_0111;

end

2:begin

lin = 16'b0000\_0000\_0000\_0100;

row = 16'b1111\_0111\_1101\_1111;

end

3:begin

lin = 16'b0000\_0000\_0000\_1000;

row = 16'b1111\_1011\_1011\_1011;

end

4:begin

lin = 16'b0000\_0000\_0001\_0000;

row = 16'b0000\_0000\_0000\_0010;

end

5:begin

lin = 16'b0000\_0000\_0010\_0000;

row = 16'b1111\_1111\_1110\_1111;

end

6:begin

lin = 16'b0000\_0000\_0100\_0000;

row = 16'b1110\_0000\_0000\_0111;

end

7:begin

lin = 16'b0000\_0000\_1000\_0000;

row = 16'b1110\_1111\_1110\_1111;

end

8:begin

lin = 16'b0000\_0001\_0000\_0000;

row = 16'b1110\_0000\_0000\_1111;

end

9:begin

lin = 16'b0000\_0010\_0000\_0000;

row = 16'b1110\_1111\_1110\_1111;

end

10:begin

lin = 16'b0000\_0100\_0000\_0000;

row = 16'b110\_0000\_0000\_1111;

end

11:begin

lin = 16'b0000\_1000\_0000\_0000;

row = 16'b1111\_1110\_1111\_1011;

end

12:begin

lin = 16'b0001\_0000\_0000\_0000;

row = 16'b0000\_0000\_0000\_0010;

end

13:begin

lin = 16'b0010\_0000\_0000\_0000;

row = 16'b1111\_1110\_1111\_1111;

end

14:begin

lin = 16'b0100\_0000\_0000\_0000;

row = 16'b1111\_1110\_1111\_1111;

end

15:begin

lin = 16'b1000\_0000\_0000\_0000;

row = 16'b1111\_1110\_1111\_1111;

end

default:

row = 16'b1111\_1111\_1111\_1111;

endcase

end

// 第二个字

1:begin

case(cnt)

0:begin

lin = 16'b0000\_0000\_0000\_0001;

row = 16'b1110\_0000\_0000\_0111;

;

end

1:begin

lin = 16'b0000\_0000\_0000\_0010;

row = 16'b1111\_1011\_1111\_1111;

end

2:begin

lin = 16'b0000\_0000\_0000\_0100;

row = 16'b1111\_0111\_1111\_0111;

end

3:begin

lin = 16'b0000\_0000\_0000\_1000;

row = 16'b1110\_0000\_0000\_0111;

end

4:begin

lin = 16'b0000\_0000\_0001\_0000;

row = 16'b1101\_0111\_1110\_1111;

end

5:begin

lin = 16'b0000\_0000\_0010\_0000;

row = 16'b1011\_0111\_1110\_1111;

end

6:begin

lin = 16'b0000\_0000\_0100\_0000;

row = 16'b0111\_0000\_0000\_1111;

end

7:begin

lin = 16'b0000\_0000\_1000\_0000;

row = 16'b1111\_1011\_1111\_1101;

end

8:begin

lin = 16'b0000\_0001\_0000\_0000;

row = 16'b0000\_0010\_0000\_0001;

end

9:begin

lin = 16'b0000\_0010\_0000\_0000;

row = 16'b1110\_1111\_1101\_1111;

end

10:begin

lin = 16'b0000\_0100\_0000\_0000;

row = 16'b1101\_1111\_1011\_1011;

end

11:begin

lin = 16'b0000\_1000\_0000\_0000;

row = 16'b1000\_0010\_0000\_0001;

end

12:begin

lin = 16'b0001\_0000\_0000\_0000;

row = 16'b0101\_1100\_1011\_1011;

end

13:begin

lin = 16'b0010\_0000\_0000\_0000;

row = 16'b1101\_1101\_1101\_1011;

end

14:begin

lin = 16'b0100\_0000\_0000\_0000;

row = 16'b1100\_0001\_1000\_0011;

end

15:begin

lin = 16'b1000\_0000\_0000\_0000;

row = 16'b1101\_1111\_1111\_1011;

end

default:

row = 16'b1111\_1111\_1111\_1111;

endcase

end

endcase

end

endmodule

```

1. 对影响每一列扫描快慢的Verilog关键程序部分给予简要的解释和说明。

初值为 0 所有计数器从 0 开始计数，便于阅读，就不用阅读计数器的定义部分，可以计算出计数了多少次，结束值为 0 使得计数器循环重复使用计数器记完就清零，在最后一个加一条件的时候清零， end\_cnt 也是在说最后一个加一个条件，根据计数器产生信息的时候，要数 的下然后变化。

1. 对影响名字扫描快慢的Verilog关键程序部分给予简要的解释和说明。

时钟频率是影响名字扫描快慢的主要因数，名字显示时间而决定。如果设置名字的显示时间为 0.25，程序设置的时钟为 50M， 1 / 50,000,000 = 0.00000002，如果是 0.5 秒为显示时间则用 0.5除以 0.00000002，得出时间为 25,000,000；如果要设置慢，比如变为 0.4 S，则 0.4 / 0.00000002 = 20,000,000 或者改变时钟的频率，也好似影响名字扫描快慢的重要因素。

1. 附上显示的名字截图。

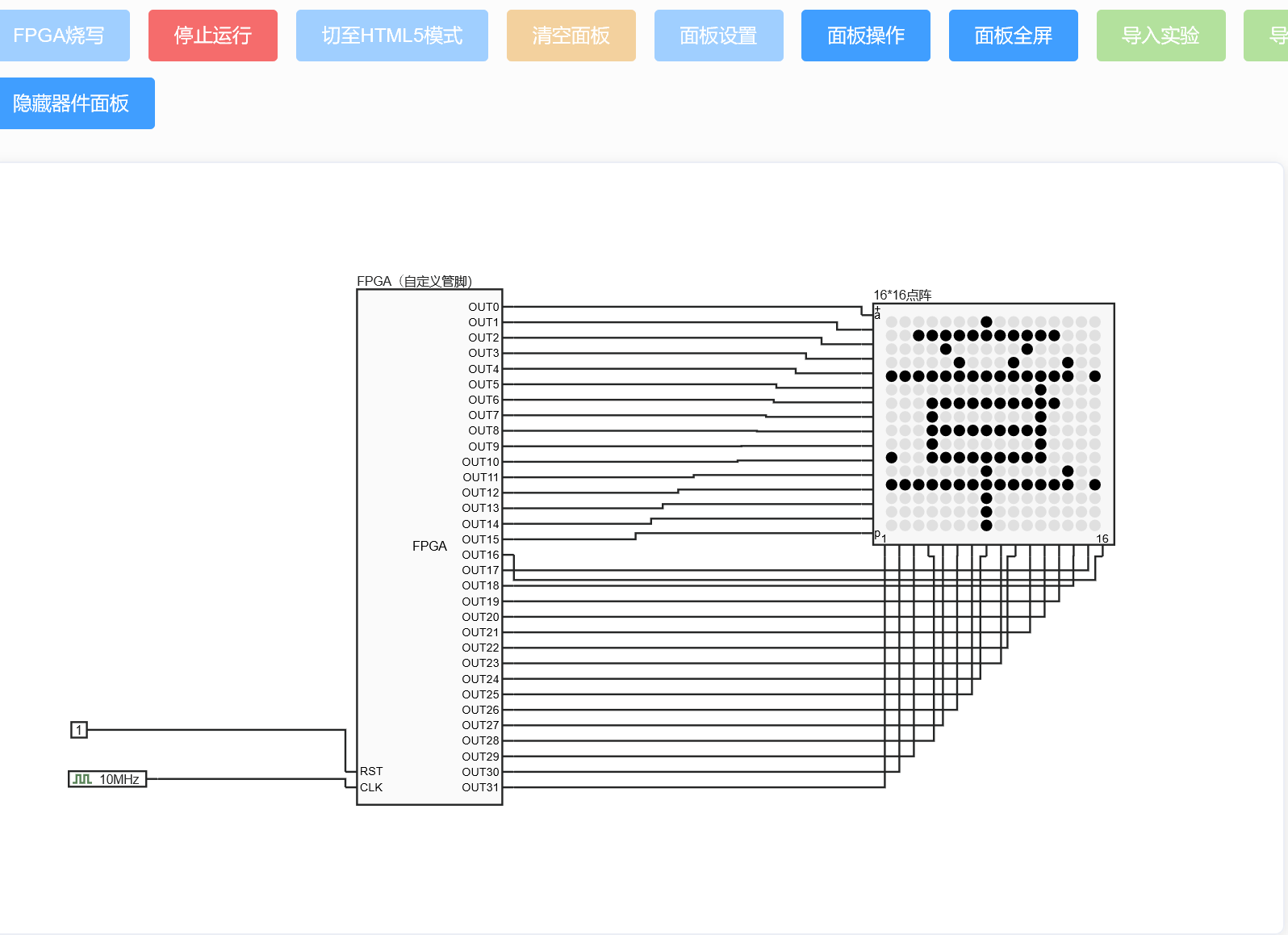
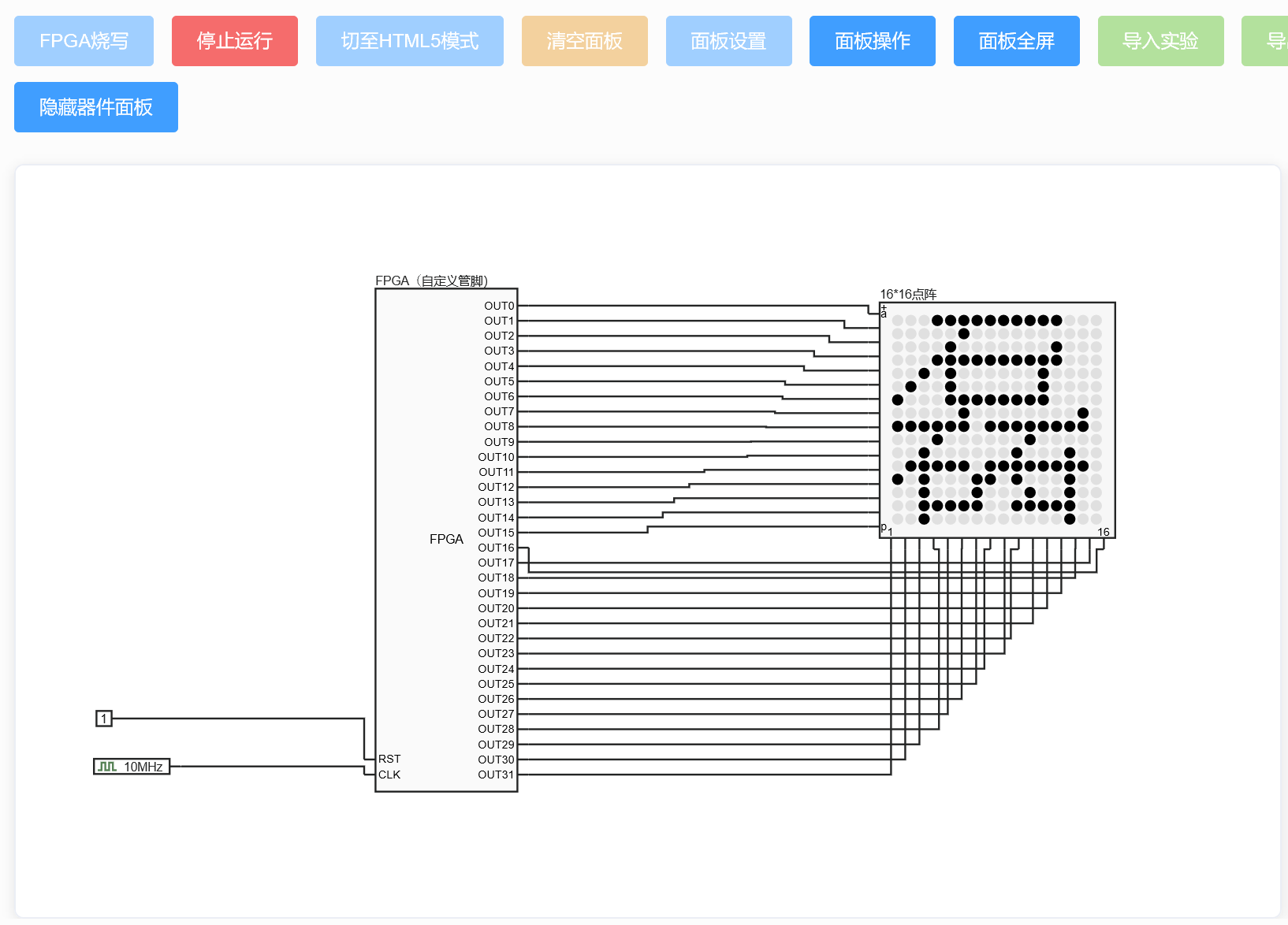
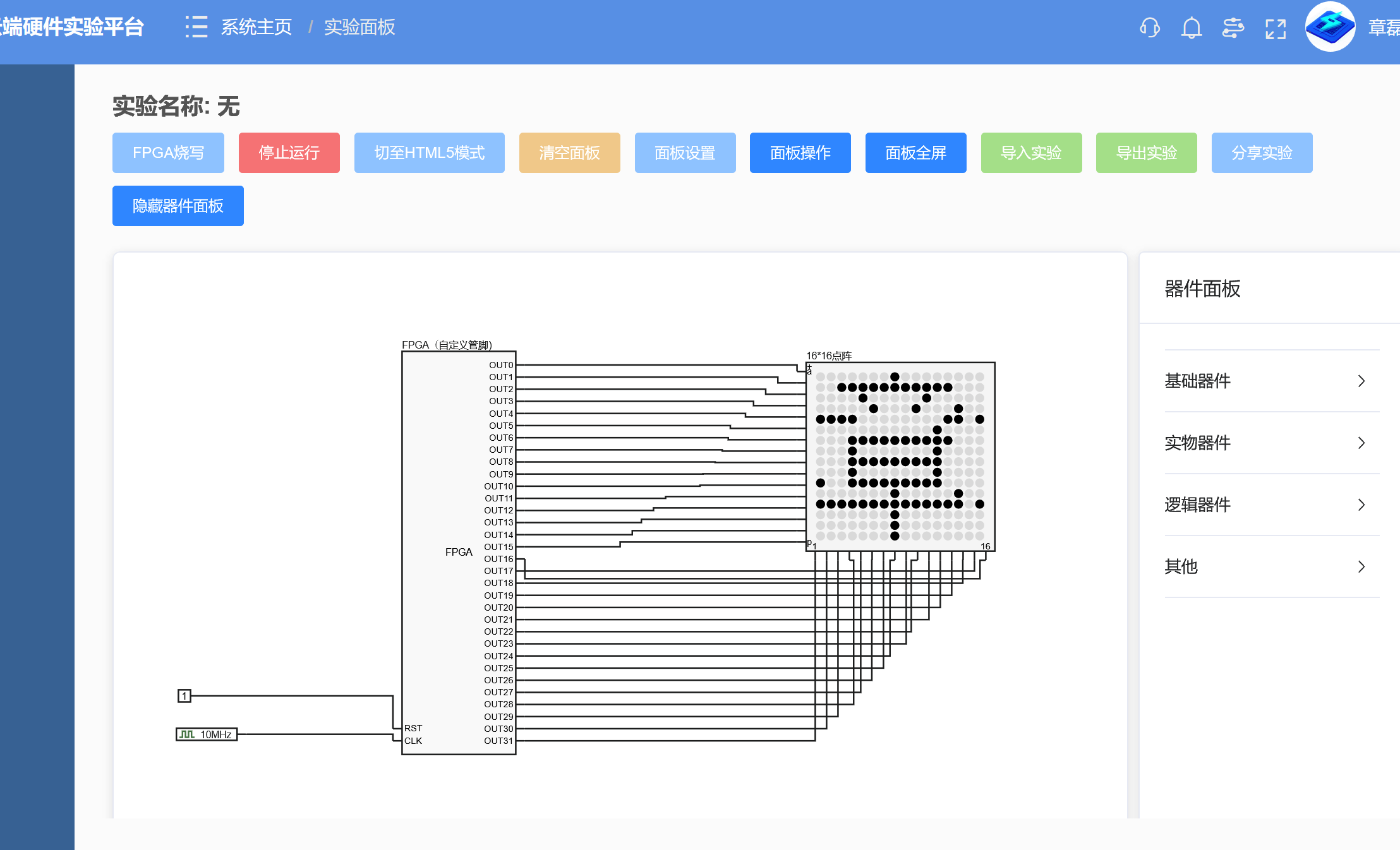


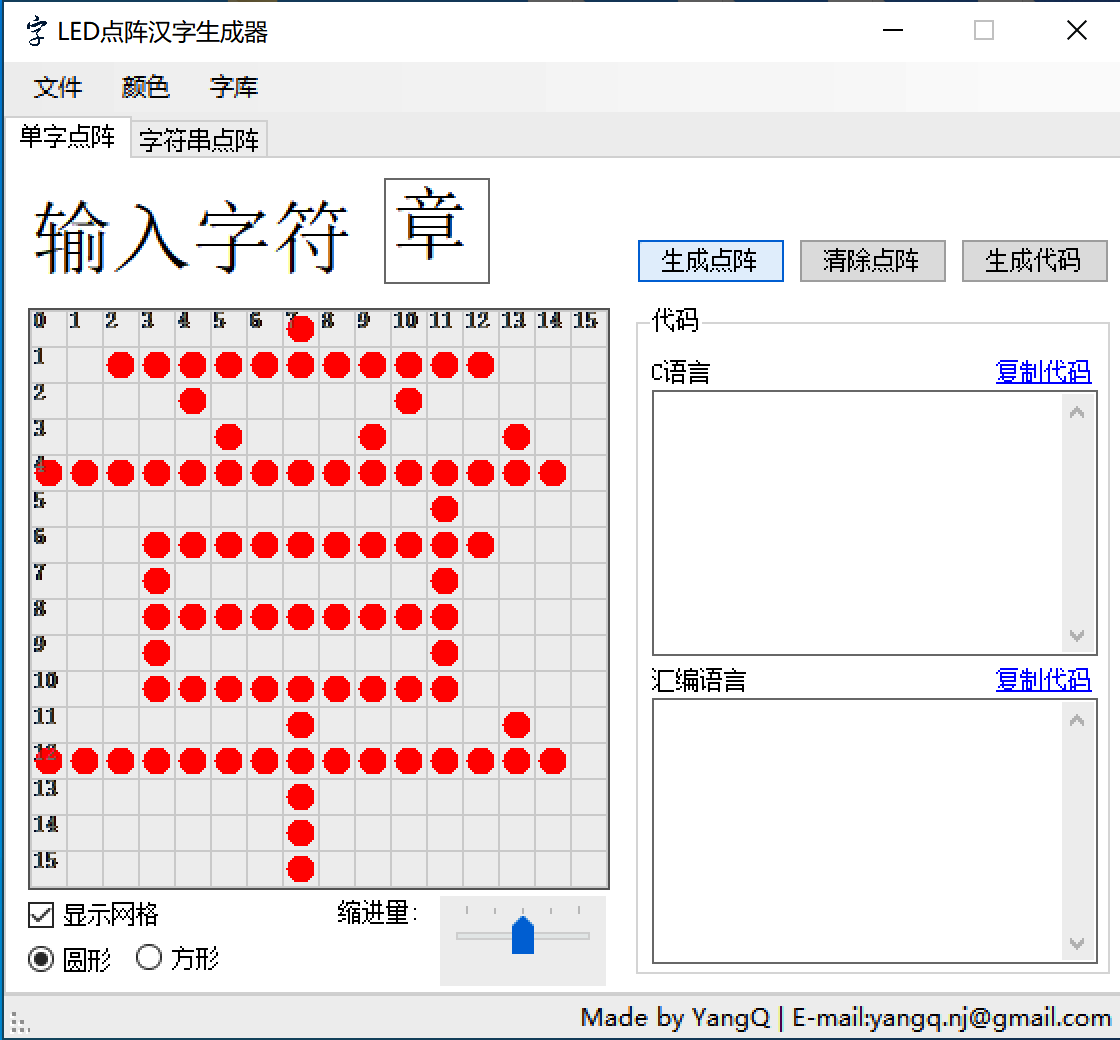
图 2‑‎0‑2



1. 如果调试过程遇到了问题，应将问题截图，并给出解决问题的办法和对应的截图。

点阵行列的 16位 二进制值错误，导致显示的名字不完全，通过使用 LED 点阵汉字生成器.exe 显示出正确的中文点阵。





# **三. 两位数十进制减法器的 IP 核，多位十进制减法器**

## **3.1. 实验任务**

调用自编写并封装的两位数十进制的减法器的IP核，生成多位数的十进制的减法器。

## **3.2. 实现步骤**

先用Verilog实现一个器件，并封装成IP。然后，调用刚才的IP核，生成新的器件。

## **3.3．实验要求**

对新生成的器件，

1. 进行仿真测试并截图；
2. 管脚绑定，下载到试验箱进行验证，并将验证结果截图保留。

## **3.4 问题与思考**

在Verilog程序中如何调用库中的IP核。

选择IP Sources，展开并选择 sub\_0.veo，可以打开实例化模板文件。在 pro\_3.v 文件中调用sub\_bcd mul(...)语句实例化了sub\_bcd类型的模块对象sub, 然后传入相应的参数即可。

## **3.5. 实验报告**

1. 根据题目要求，编写要生成器件的Verilog程序。

```Verilog

//两位数十进制的减法器的IP核

（1）源代码：

`timescale 1ns / 1ps

module sub\_bcd(

input [3:0] a0,

input [3:0] a1,

input [3:0] b0,

input [3:0] b1,

input c,

output [3:0] d0,

output [3:0] d1,

output ci

);

reg[3:0] d0;

reg[3:0] d1;

reg ci;

reg temp\_c;

always @(\*)

begin

if(a0>=b0+c) begin

temp\_c=1'b0;

d0=a0-b0-c;

if(a1>=b1+temp\_c) begin

ci=1'b0;

d1=a1-b1-temp\_c;

end

else if(a1<b1+temp\_c) begin

ci=1'b1;

d1=4'b1010-b1+a1-temp\_c;

end

end

else if(a0<b0+c) begin

temp\_c=1'b1;

d0=4'b1010-b0+a0-c;

if(a1>=b1+temp\_c) begin

ci=0;

d1=a1-b1-temp\_c;

end

else if(a1<b1+temp\_c) begin

ci=1'b1;

d1=4'b1010-b1+a1-temp\_c;

end

end

end

endmodule

//数码管显示模块的IP核

module display(clk,b\_0,b\_1,b\_2,b\_3,sel,seg);

input clk;//50M

input [3:0] b\_0,b\_1,b\_2,b\_3;

output [2:0]sel;//位选（控制哪个数码管亮）

output reg [7:0]seg;//段选（控制数码管显示什么数据）

//分频器的代码，这里为了完整，不做多个文件来写模块了

reg [2:0]sel\_r=3'b000;

reg [3:0]data\_temp;//待显示数据缓存

reg clock\_25m=0;

always@(posedge clk)

begin

clock\_25m <= ~clock\_25m;

end

//计数器，约为10ms扫描一次

reg [17:0] cnt=0;

always @(posedge clock\_25m)

begin

if(cnt == 18'b111111111111111111)

cnt <= 18'b0;

else

cnt <= cnt+1;

end

//位选移位寄存器

always@(posedge clock\_25m)

if(sel\_r==3'b111)

sel\_r<=3'b000;

else

sel\_r<=sel\_r+1'b1;

//设计一个多路器

always@(\*)

case(sel\_r)

3'b000:data\_temp=b\_3;

3'b001:data\_temp=b\_2;

3'b010:data\_temp=b\_1;

3'b011:data\_temp=b\_0;

3'b100:data\_temp=4'ha;

3'b101:data\_temp=4'ha;

3'b110:data\_temp=4'ha;

3'b111:data\_temp=4'ha;

//default

//data\_temp<=4'ha;

endcase

//译码器

always@(\*)

case (data\_temp)

4'h0:seg=8'b00111111;//这里按数码管码表来

4'h1:seg=8'b00000110;

4'h2:seg=8'b01011011;

4'h3:seg=8'b01001111;

4'h4:seg=8'b01100110;

4'h5:seg=8'b01101101;

4'h6:seg=8'b01111101;

4'h7:seg=8'b00000111;

4'h8:seg=8'b01111111;

4'h9:seg=8'b01101111;

4'ha:seg=8'b00000000;

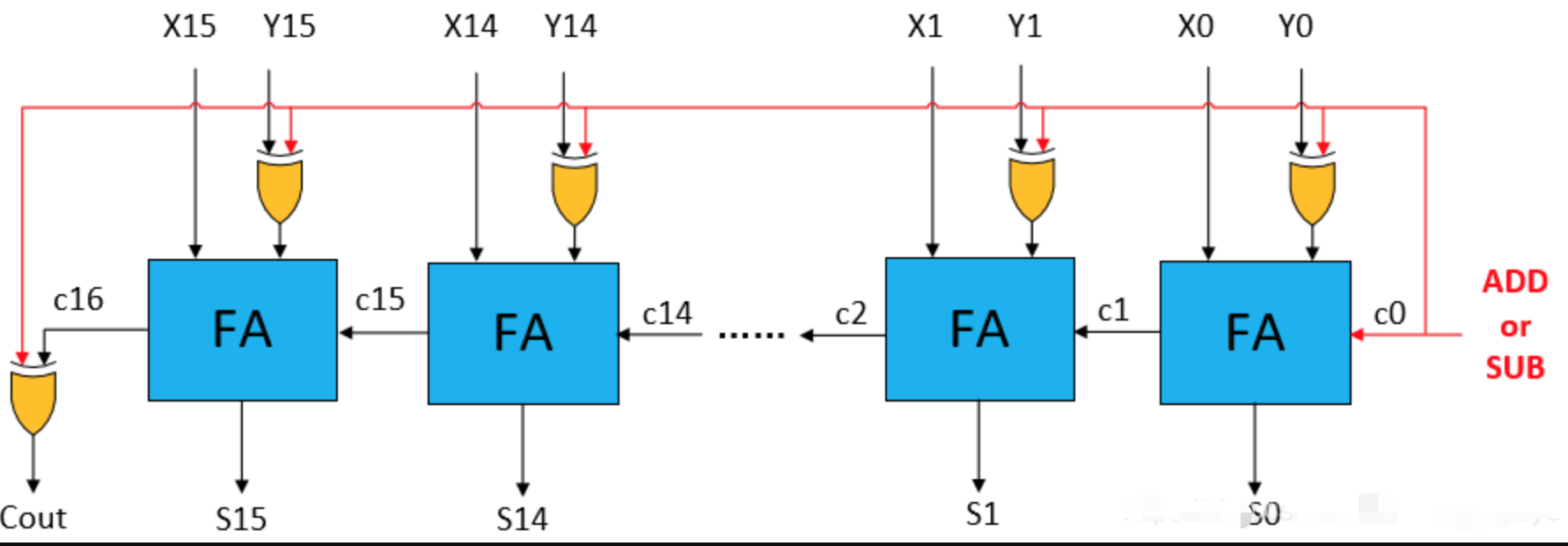
endcase

assign sel=sel\_r;

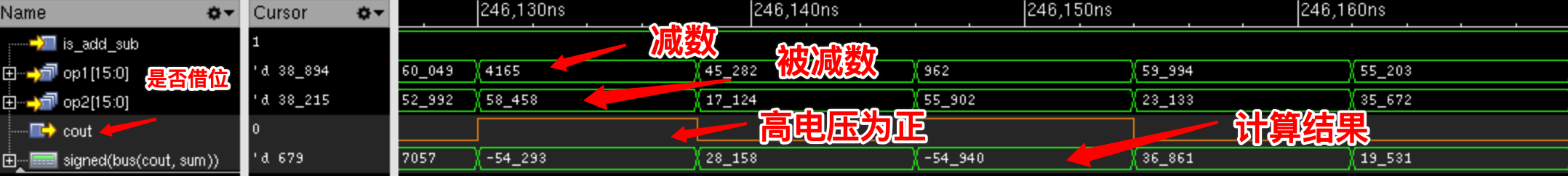
endmodule

```

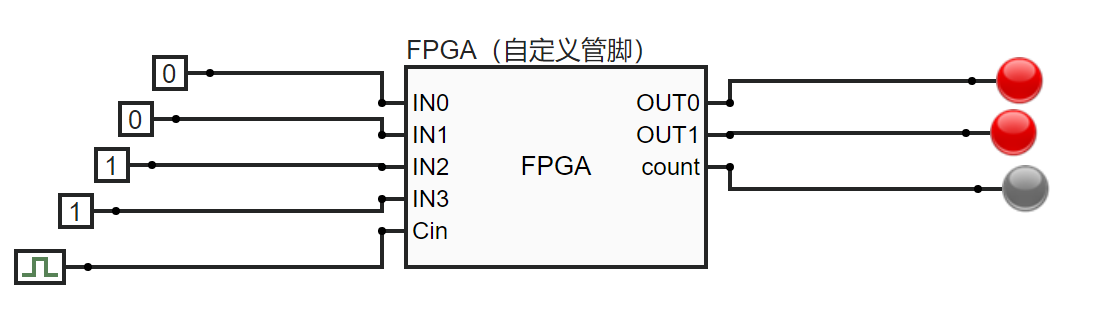
1. 附上Verilog程序对应的电路图。



1. 编写仿真程序，并对仿真波形用线段或带箭头的线段标注输出与各个输入之间的关系，并给予简单的解析和说明。【被减数和减数标记反了】



1. 调用刚才的IP核，生成新的器件。然后，管脚绑定，下载到试验箱进行验证，并将验证结果截图保留。



# **四. 序列检测器**

## **4.1.序列检测器原理**

1. 序列检测器在很多数字系统中都不可缺少，尤其是在通信系统当中。序列检测器的作用就是从一系列的码流中找出希望出现的序列，序列可长可短。比如在通信系统中，数据流帧头的检测就属于一个序列检测器。序列检测器的类型有很多种，有逐个比特比较的，有逐个字节比较的，也有其它的比较方式，实际应用中采用何种比较方式，主要是看序列的多少以及系统的延时要求。
2. 逐个比特比较的序列检测器，是将一个二进制码流与特定序列比较。首先将二进制码流与特定序列的第一个码比较，如果二进制码流与特定序列的第一个码相同，那么再和特定的序列的第二个码相比较，依次比较下去，直到二进制码流和特定序列相一致，就认为检测到一个特定序列。

## **4.2.状态机**

1. 为什么使用状态机
   * + 1. 有限状态机克服了纯硬件数字系统顺序方式控制不灵活的缺点。
       2. 设计方案相对固定，结构模式简单，可定义符号化枚举类型的状态。
       3. 状态机的Verilog描述层次分明，结构清晰，易读易懂。
       4. 状态机容易构成性能良好的同步时序逻辑模块。
       5. 基于有限状态机技术设计的控制器其工作速度大大优于CPU。
       6. 就可靠性而言，基于有限状态机技术设计的控制器其可靠性优于CPU。
2. 状态机是一类很重要的时序电路

时序电路是很多数字电路的核心部件，是大型电子设计的基础。

状态机相当于一个控制器，它将一项功能的完成分解为若干步，每一步对应于二进制的一个状态，通过预先设计的顺序在各状态之间进行转换，状态转换的过程就是实现逻辑功能的过程。

状态机图是指用图形的方式来表示一个设计实体的各种工作状态、内部各工作状态转换的条件以及各工作状态对应的输出信号序列。

## **4.3.问题与思考**

1. 如何保证状态机在初始时状态为S0？

通过判断 end 变量是否是0，如果是 0，返回 S0 状态，一开始 end 初始化为 0。

1. 在状态机跑飞，即脱离有效状态(S0—S3)时，如何使状态机能恢复工作。

设置 case 中，设置一个 default 分支处理状态机跑飞，一旦跑飞，跳转到 default 分支，将当前状态改为 S0 进行调正。

**4.4. 实验报告**

1. 带重复判断和不带重复判断检测特定二进制数字系列的原理。
   * 1. 不进行重复判断

假设，通过拨挡开关K1～K8，输入一个八位二进制数据X=10101010。

检测在X中包含了几个不重复的特定序列Y=11，并在输出Z的指定位置上显示。

X = 1 1 1 0 1 0 1 0

Z = 0 1 0 0 0 0 0 0

也就是说，在Z中包含了不重复的特定序列Y有1个。

* + 1. 进行重复判断

假设，通过拨挡开关K1～K8，输入一个八位二进制数据X=10101010。

检测在X中包含了几个重复的特定序列Y=11，并在输出Z的指定位置上显示。

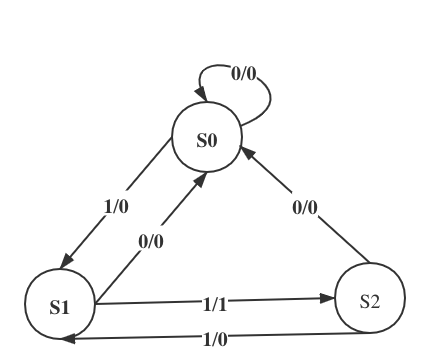
X = 1 1 1 0 1 0 1 0

Z = 0 1 1 0 0 0 0 0

也就是说，在Z中包含了重复的特定序列Y有2个。

1. 用有限状态机描述带重复判断和不带重复判断检测特定二进制数字的原理。和两个状态迁移图和两个状态迁移表。
   * 1. 进行重复判断

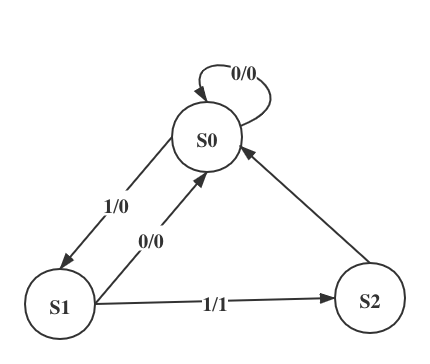
实验中我校验的序列是 Y=11，初始化在 S0 状态，如果此时接受到 1，跳转到 S1 状态，如果接受到 0，在 S0 进行闭环；在 S1 状态，如果接受到 1，跳转到 S2 状态，然后亮灯，如果接受到 0，返回 S0 状态；S2 状态，如果接受到 1 返回 S1 状态，开始进行 1 的重复检测，如果接受到 0，返回 S0 状态。



|  |  |  |  |
| --- | --- | --- | --- |
| 初态  次态/Z  X | S0 | S1 | S2 |
| 0 | 0/0 | 0/0 | 0/0 |
| 1 | 1/0 | 1/1 | 1/0 |

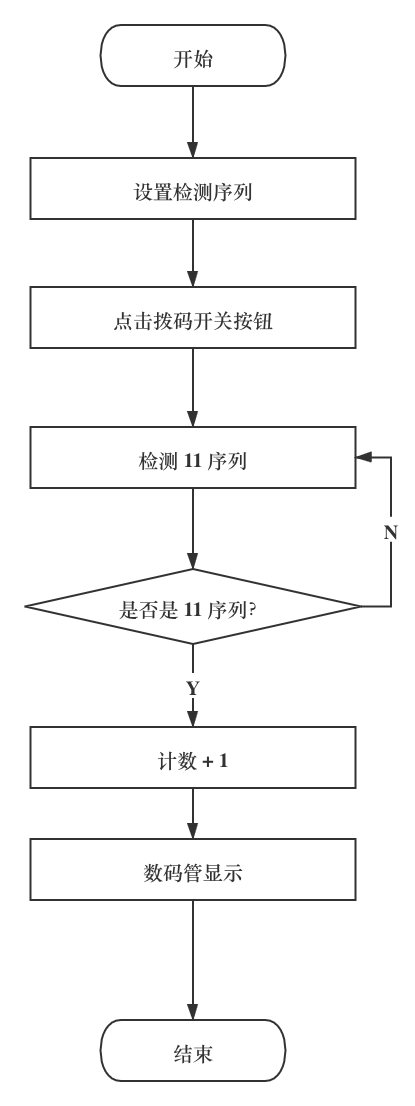
* + 1. 不进行重复判断

实验中我校验的序列是 Y=11，初始化在 S0 状态，如果此时接受到 1，跳转到 S1 状态，如果接受到 0，在 S0 进行闭环；在 S1 状态，如果接受到 1，跳转到 S2 状态，然后亮灯，开始新的序列校验，如果接受到 0，返回 S0 状态；S2 状态，直接返回 S0 状态进行新一轮的序列校验。



|  |  |  |  |
| --- | --- | --- | --- |
| 初态  次态/Z  X | S0 | S1 | S2 |
| 0 | 0/0 | 0/0 |  |
| 1 | 1/0 | 1/1 |  |

1. 程序功能框图。



1. Verilog程序

```Verilog

`timescale 1ns / 1ps

module jiance(

clk ,

rst\_n ,

key\_flag ,

switch ,

led ,

led\_cnt

);

parameter S0 = 0;

parameter S1 = 1;

parameter S2 = 2;

parameter S3 = 3;

//输入信号定义

input clk ;

input rst\_n ;

input key\_flag ;

input [7:0] switch ;

//输出信号定义

output reg [7:0] led ;

output reg [2:0] led\_cnt ;

//中间信号定义

reg en ;

reg [3:0] cnt ;

wire add\_cnt ;

wire end\_cnt ;

reg x ;

reg [2:0] state\_c ;

reg [2:0] state\_n ;

wire s02s1\_start ;

wire s12s2\_start ;

wire s22s1\_start ;

always @(posedge clk or negedge rst\_n)begin

if(rst\_n==1'b0)begin

en <= 0;

end

else if(key\_flag)begin

en <= 1;

end

else if(end\_cnt)begin

en <= 0;

end

end

always @(posedge clk or negedge rst\_n)begin

if(!rst\_n)begin

cnt <= 0;

end

else if(add\_cnt)begin

if(end\_cnt)

cnt <= 0;

else

cnt <= cnt + 1;

end

end

assign add\_cnt = en;

assign end\_cnt = add\_cnt && cnt==10-1 ;

always @(\*)begin

case(cnt)

0:x=0;

1:x=switch[0];

2:x=switch[1];

3:x=switch[2];

4:x=switch[3];

5:x=switch[4];

6:x=switch[5];

7:x=switch[6];

8:x=switch[7];

default:x=0;

endcase

end

always@(posedge clk or negedge rst\_n)begin

if(!rst\_n)begin

state\_c <= S0;

end

else begin

state\_c <= state\_n;

end

end

always@(\*)begin

if(en)begin

case(state\_c)

S0:begin

if(s02s1\_start)begin

state\_n = S1;

end

else begin

state\_n = S0;

end

end

S1:begin

if(s12s2\_start)begin

state\_n = S2;

end

else begin

state\_n = S0;

end

end

S2:begin

if(s22s1\_start)begin

state\_n = S2;

end

else begin

state\_n = S0;

end

end

default:begin

state\_n = S0;

end

endcase

end

else begin

state\_n <= S0;

end

end

assign s02s1\_start = state\_c==S0 && x==1;

assign s12s2\_start = state\_c==S1 && x==1;

assign s22s1\_start = state\_c==S2 && x==1;

always @(posedge clk or negedge rst\_n)begin

if(rst\_n==1'b0)begin

led <= 0;

end

else if(state\_c==S2)begin

led[cnt-2] <= 1 ;

end

else if(key\_flag)begin

led <= 0;

end

end

always @(posedge clk or negedge rst\_n)begin

if(rst\_n==1'b0)begin

led\_cnt <= 0;

end

else if(en)begin

if(state\_c==S2)

led\_cnt <= led\_cnt + 1;

else

led\_cnt <= led\_cnt;

end

else begin

led\_cnt <= 0;

end

end

endmodule

```

1. Verilog程序关键或重要的程序语句的注释和说明。

assign s02s1\_start = state\_c==S0 && x==1; // S0 to S1 的转换条件

assign s12s2\_start = state\_c==S1 && x==1; // S1 to S2 的转化条件

assign s22s1\_start = state\_c==S2 && x==1; // S2 to S1 的转化条件

// 状态转化跳转逻辑

always@(\*)begin

if(en)begin

case(state\_c)

// S0 跳转逻辑

S0:begin

if(s02s1\_start)begin // S0 to S1

state\_n = S1;

end

else begin // S0 to S0

state\_n = S0;

end

end

// S1 跳转逻辑

S1:begin

if(s12s2\_start)begin // S1 to S2

state\_n = S2;

end

else begin // S1 to S0

state\_n = S0;

end

end

// S2 跳转逻辑

S2:begin

if(s22s1\_start)begin

state\_n = S2;

end

else begin

state\_n = S0;

end

end

default:begin

state\_n = S0;

end

endcase

end

else begin

state\_n <= S0;

end

end

// 计算符合要求的校验序列

always @(posedge clk or negedge rst\_n)begin

if(rst\_n==1'b0)begin

led\_cnt <= 0;

end

else if(en)begin

if(state\_c==S2) // 当状态到达 S2 时，表示有一个符合，cnt + 1

led\_cnt <= led\_cnt + 1;

else

led\_cnt <= led\_cnt;

end

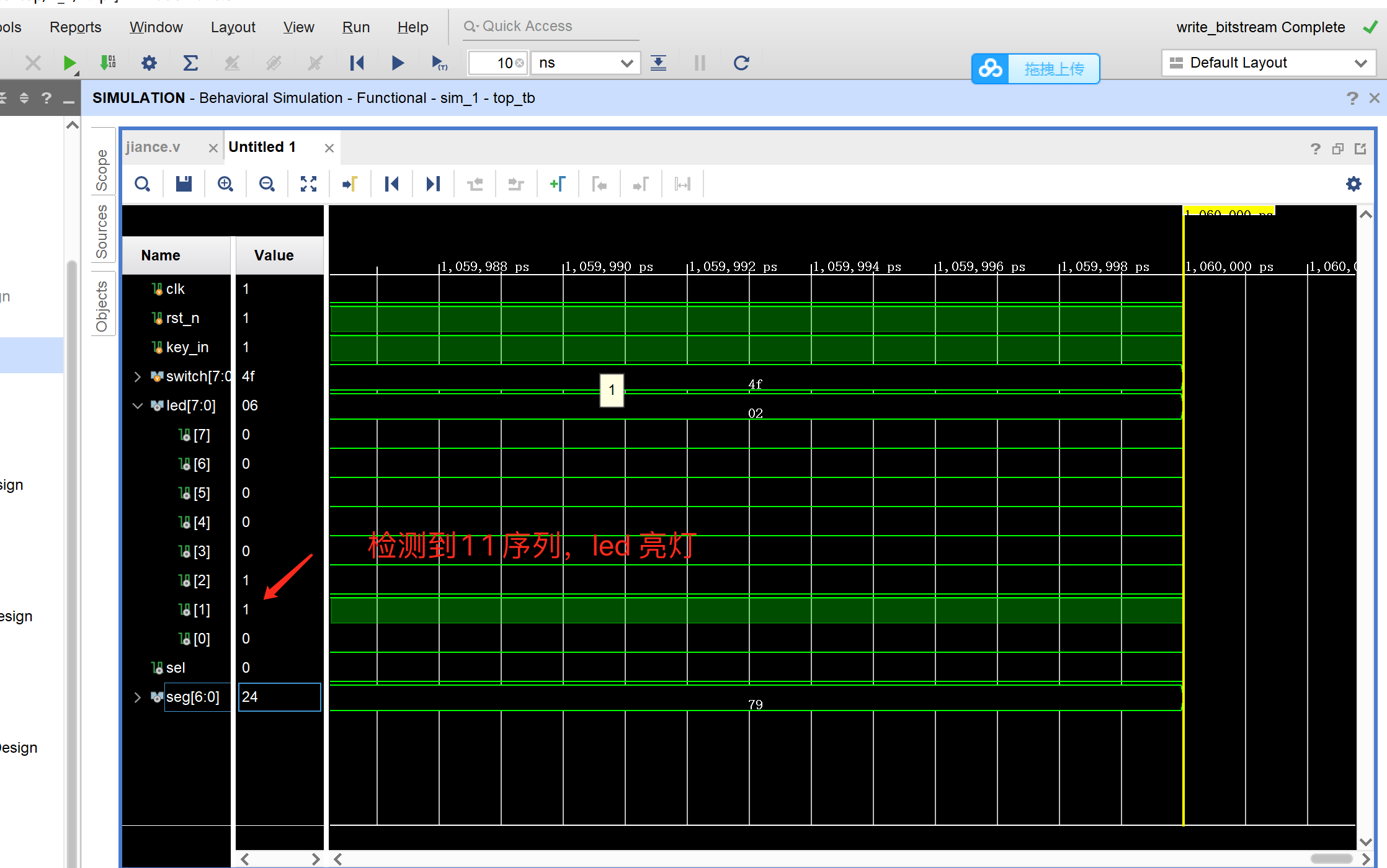
else begin

led\_cnt <= 0;

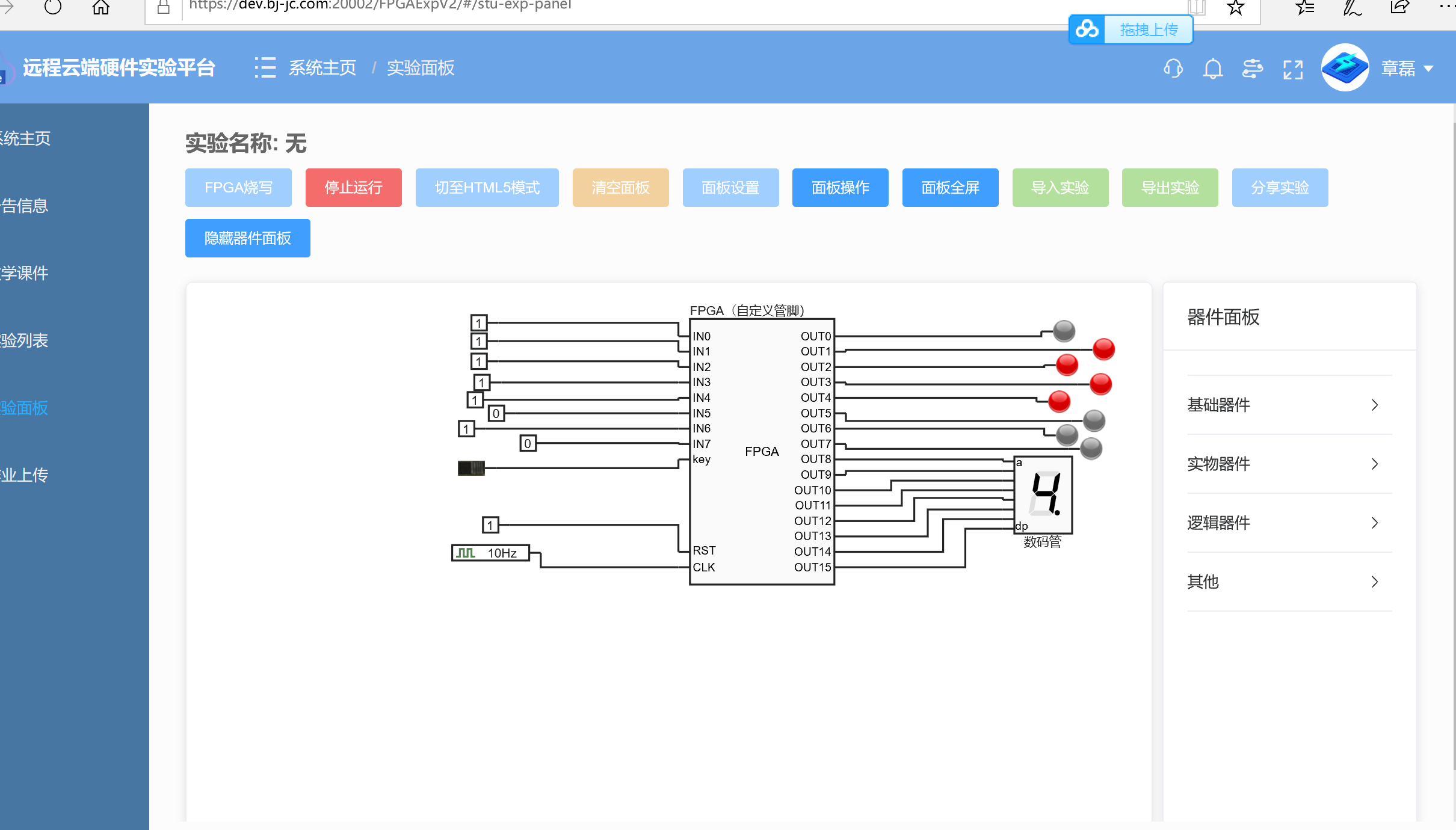
end

end

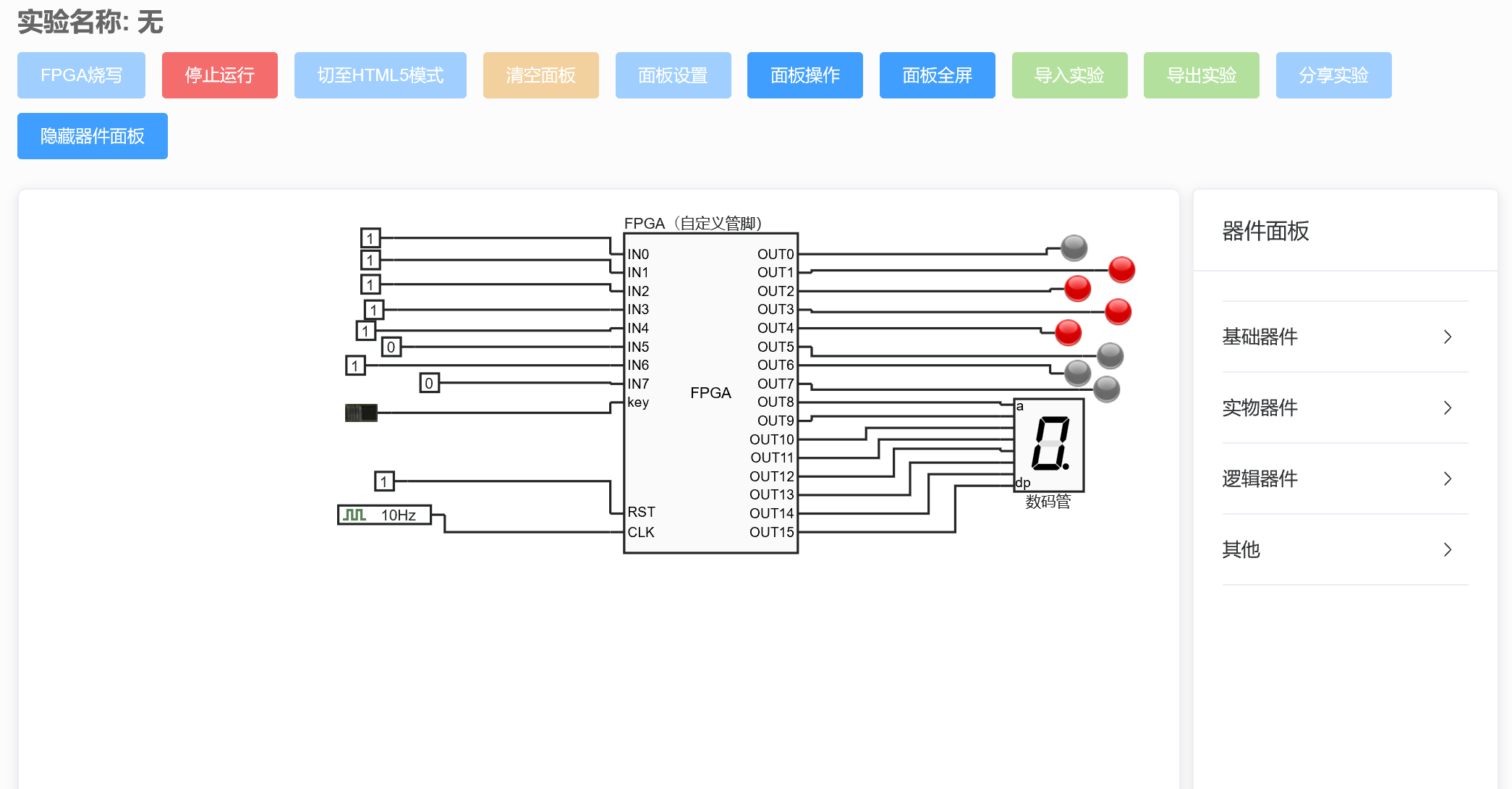
1. 重复判断和不带重复判断检测特定二进制数字系列仿真。



1. 实验结果



1. 问题

在云平台统计将匹配到的序列个数显示到数码管的时候，出现正确数字显示闪过的情况，主要是因为时钟频率调的太快了，通过调整时钟频率调可解决。

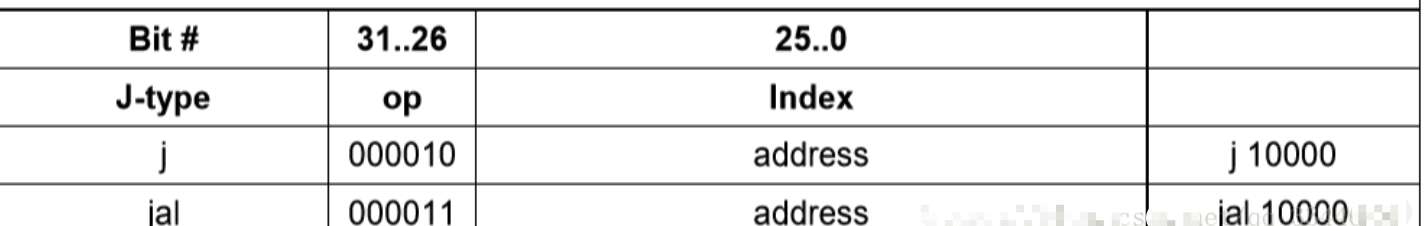
# **五、简单CPU的设计**

## **5.1 实验报告要求**

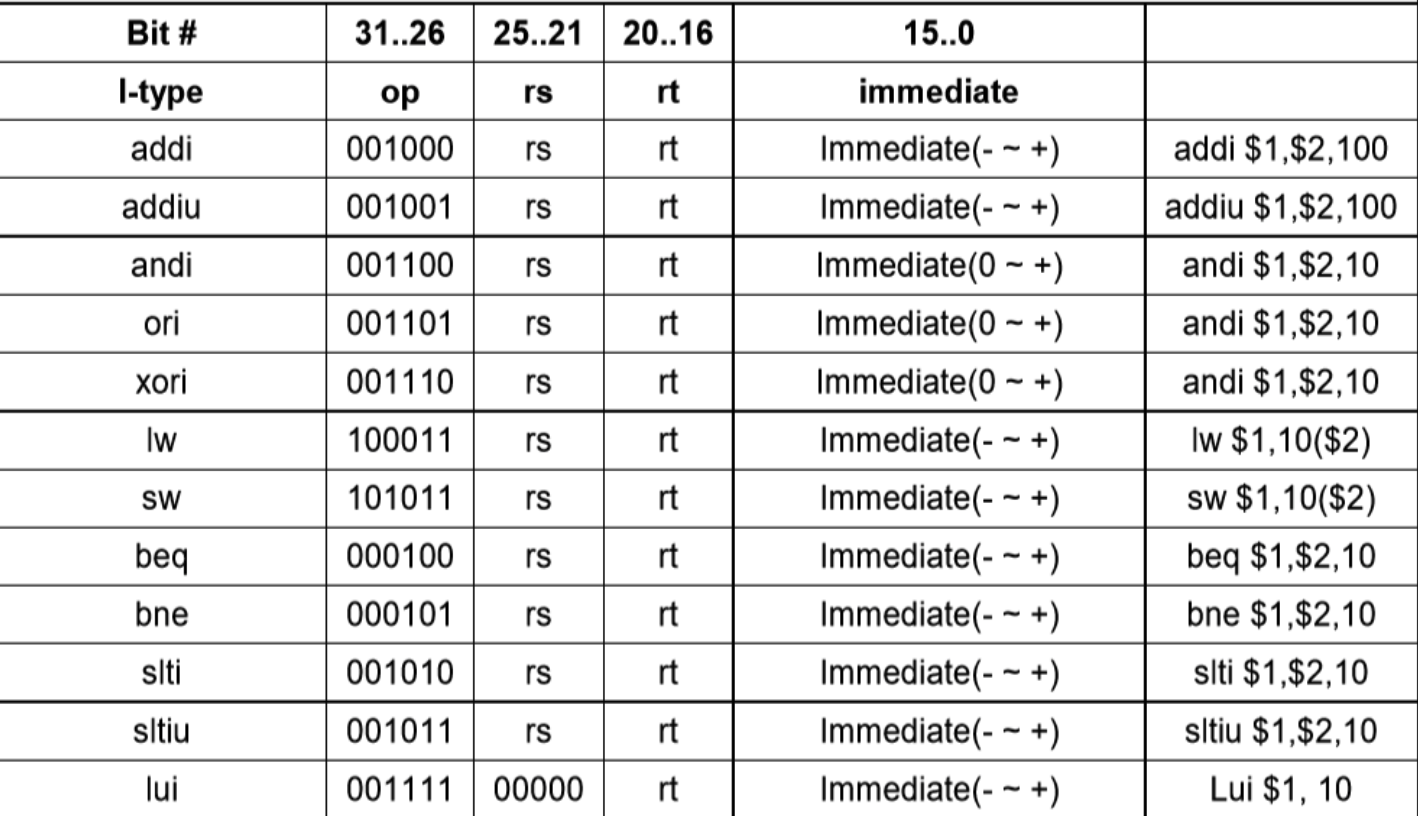
1. 简单陈述CPU指令系统

① 指令分类和指令分类表

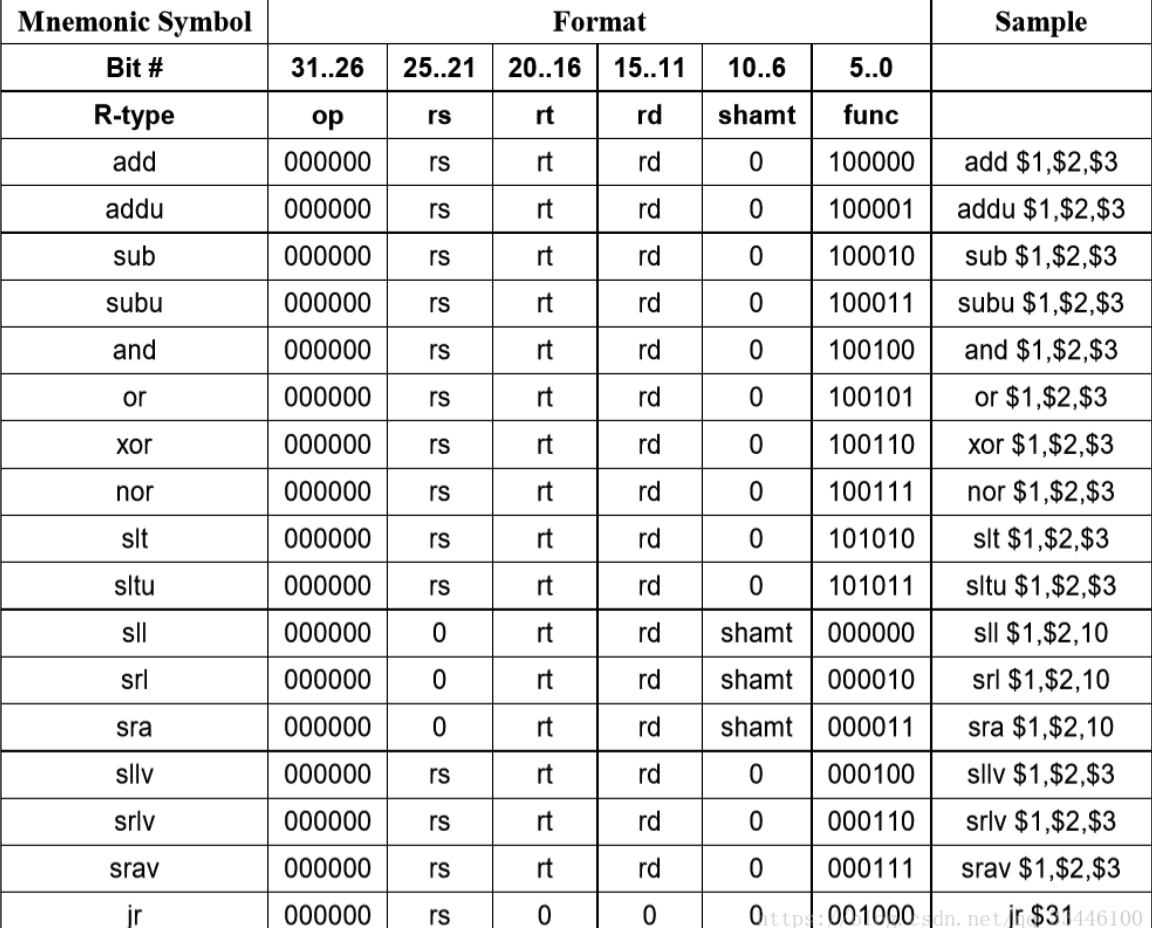
* J 型指令



* I 型指令



* R 型指令



② CPU的操作码描述

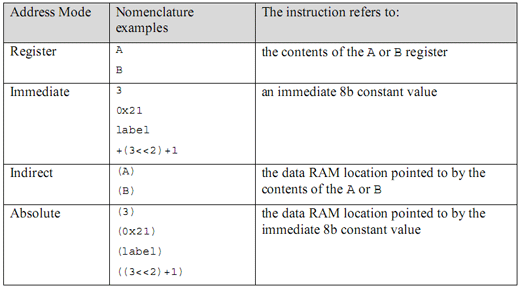
操作数（operand)，是计算机指令中的一个组成部分，它规定了指令中进行数字运算的量 。操作数指出指令执行的操作所需要数据的来源。操作数是汇编语言指令的一个字段。例如：mov a b 操作数（a 和 b）。在操作数这个字段中可以放操作数本身，也可以放操作地址，还可以放操作地址的计算方法。通常一条指令均包含操作符和操作数。例如：在比较指令中操作符指定计算机做比较操作，操作数则指定进行比较的两个数值。操作数是指令执行的参与者,也就是各种操作的对象.与之有关的是操作码,所谓操作码是说明计算机要执行哪种,如传送,运算,移位,跳转等操作,它是指令中不可缺少的组成部分。

③ 对CPU的功能码进行详细的描述

功能码用于标明一个Modbus信息帧的用途，如功能码01为读取[线圈](https://baike.baidu.com/item/%E7%BA%BF%E5%9C%88/6244981)状态，02为读取输入状态等。当主设备向从设备发送信息时，功能码将告诉从设备需要执行哪些行为。例如，去读取输入的开关状态、读一组寄存器的数据内容等。当从设备响应时，使用功能码用于指示是正常响应(无误)还是有某种错误发生(称作异议回应)。正常应答时，主机发送的功能码等于从机应答的功能码。

④ 对CPU的操作数进行分类并画出对应的操作数分类表

* 立即操作数：指令要操作的数据以常量的形式出现在指令中，称为立即数，它只能作为源操作数。
* 寄存器操作数：指令要操作的数据存放在CPU中的寄存器里，指令中给出寄存器名即可。
* 内存操作数：指令要操作的数据存放在内存某些单元中，指令中给出内存单元物理地址（实际上指令只给出了偏移地址，段地址采用隐含方式给出，也可以使用跨段方式指出当前段地址。



⑤ 对CPU的操作数涉及的寻址方式进行分类并画出对应的寻址方式分类表

* 存储器寻址

1. 直接寻址
2. 间接寻址
3. 基址寻址
4. 变址寻址
5. 基址加变址寻址

* 非存储器寻址方式

1. 寄存器寻址
2. 立即寻址
3. 隐含寻址
4. IO寻址

⑥ 对存储器进行存、取的过程进行简单的描述

* 取过程

1. CPU通过地址线将地址信息 A 发出。
2. CPU通过控制线发出内存读命令，选中存储器芯片。并通知它，将要从中读取数据。
3. 存储器将 A 号单元中的数据 D 通过数据线送入CPU。

* 存过程

1. CPU通过地址线将地址信息 A 发出。
2. CPU通过控制线发出内存写命令，选中存储器芯片。并通知它，要向其中写入数据。
3. CPU通过数据线将数据 D 送入到内存的A 号单元中。

2. 每一类的一条CPU指令的实现

1. 算术运算指令

指令名称：add

指令功能：rd ← rs + rt; PC ← NPC (PC + 4)

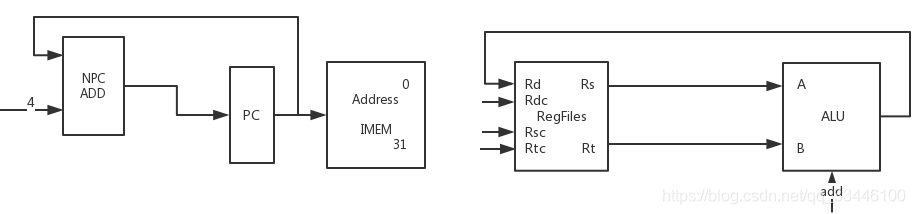
指令格式：

add rd , rs, rt

000000 rs(5位) rt(5位) rd(5位) reserved

寻址方式：直接寻址

执行流程图



1. 逻辑运算指令

指令名称：or

指令功能： rd←rs | rt；逻辑或运算。

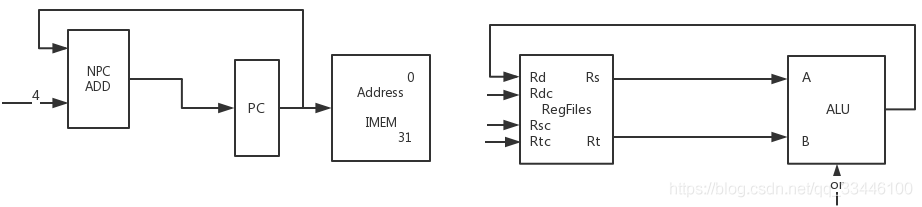
指令格式：

or rd , rs , rt

010010 rs(5位) rt(5位) rd(5位) reserved

寻址方式：直接寻址

执行流程图



1. 移位指令

指令名称：sll

指令功能：rd<－rt<<(zero-extend)sa，左移sa位 ，(zero-extend)sa

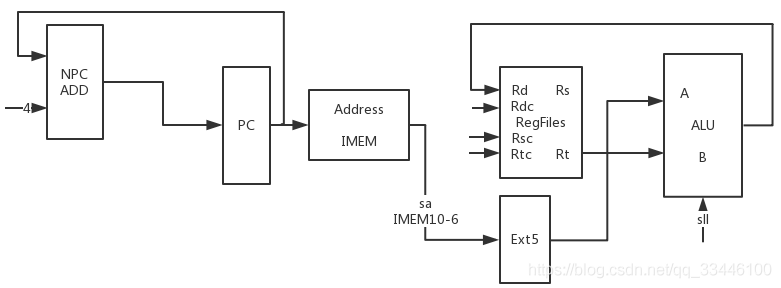
指令格式：

sll rd, rt, sa

011000 未用 rt(5位) rd(5位) sa reserved

寻址方式：寄存器寻址

执行流程图



1. 比较指令

指令名称：slti

指令功能：if (rs <(sign-extend)immediate) rt =1 else rt=0

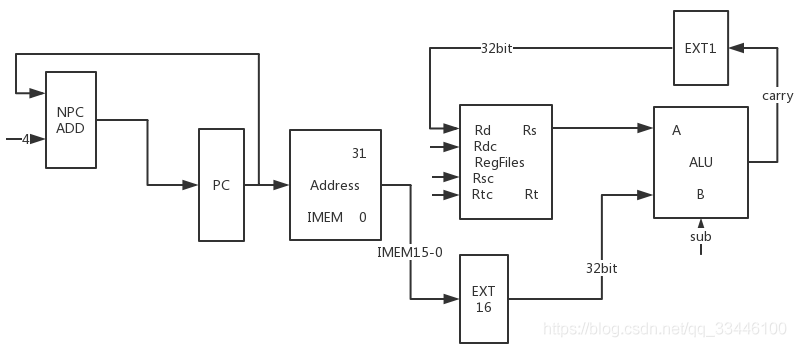
指令格式：

slti rt, rs,immediate 带符号

011011 rs(5位) rt(5位) immediate(16位)

寻址方式：直接寻址

执行流程图



1. 跳转指令

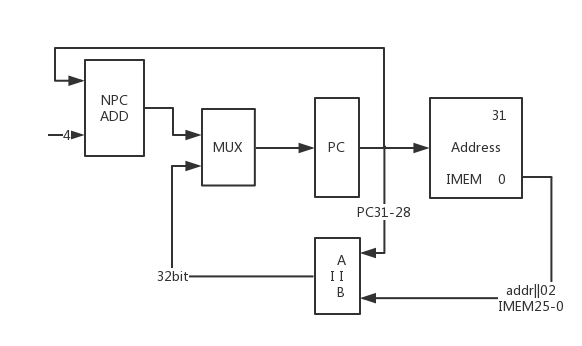
指令名称：j

指令功能：PC ← PC31-28 || instr\_index || 02; PC ← NPC(PC+4)

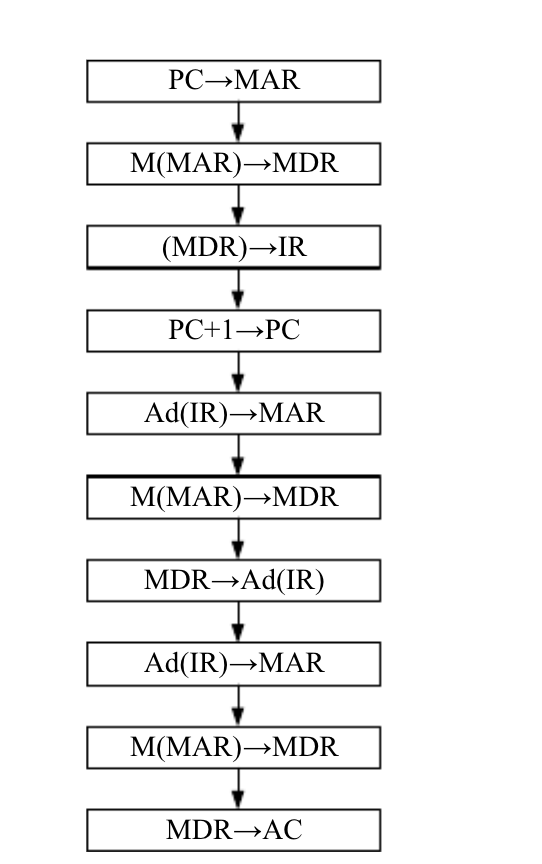
指令格式：j address

寻址方式：相对寻址

执行流程图



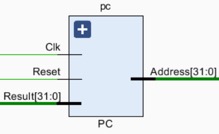
3. 画出指令实现的数据流图



4. 画出主要的功能部件，

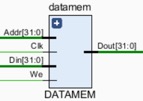
如PC、IR、MAR、MDR等功能部件的框图，需要对功能部件的输入/输出引脚一一标注清楚。

1. PC



**引脚及控制信号**  
Clk：时钟周期，输入信号  
Reset：控制信号，输入信号  
Result目标地址，可能是跳转地址或者是下一条指令的地址，输入信号  
Addr：指令地址，输出信号

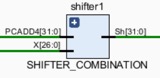
1. DATAMEN



**引脚及控制信号**

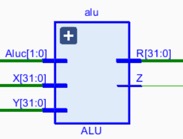
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 引脚 | 控制信号 | 作用 | 状态“0” | 状态“1” |
| Addr（输入） | R | 访存地址 |  |  |
| Din（输入） | Qb | 输入的值 |  |  |
| Clk（输入） | Clk | 时钟周期 |  |  |
| We（输入） | Wmem | 写使能信号 | 信号无效 | 信号有效 |
| Dout（输出） | Dout | 读取的值 |  |  |

1. SHIFTER\_COMBINATION



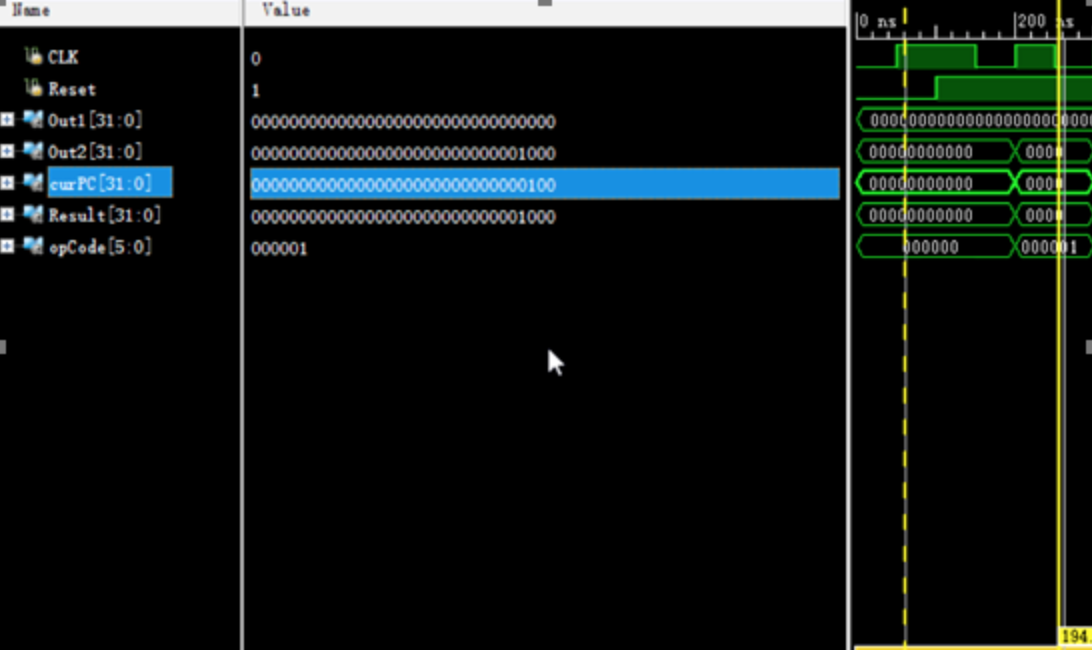
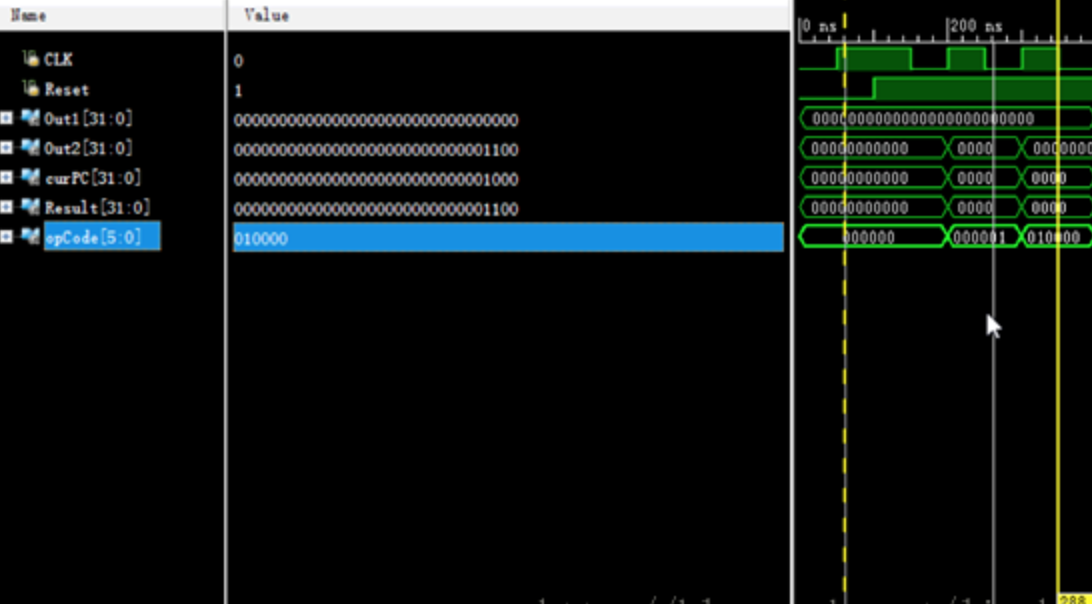
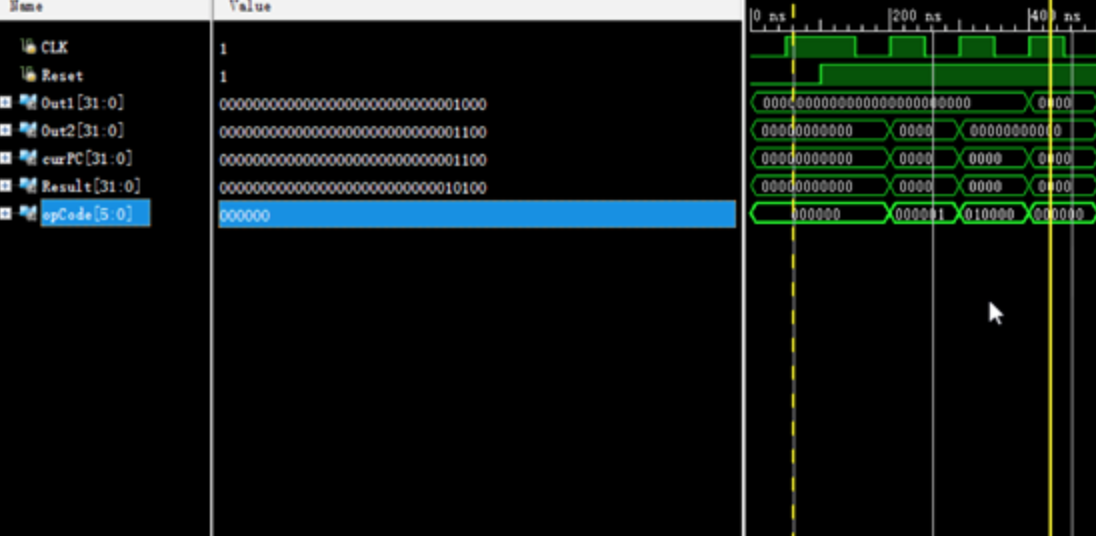
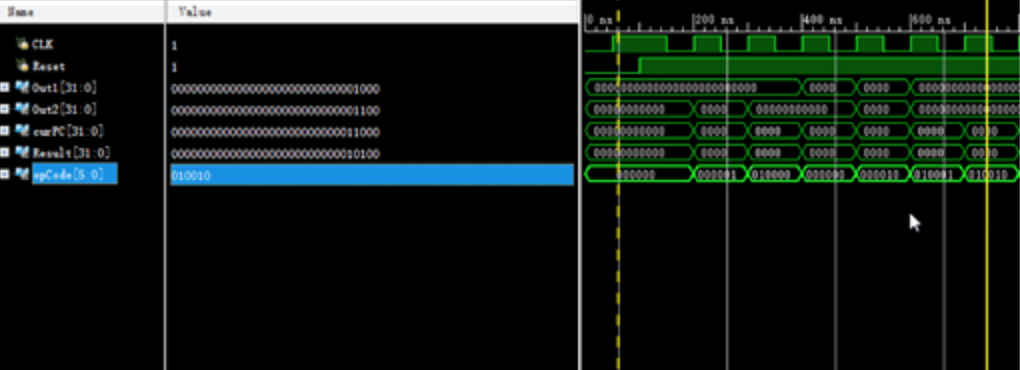
**引脚及控制信号**Inst[26:0]：指令编码的低26位字段，输入信号。  
PCadd4：PC+4的32位字段，输入信号。  
InstL2：32位转移目标地址，输出信号。

1. ALU



**引脚及控制信号**Qa：寄存器1的值。  
Y：寄存器2的值或立即数。  
Aluc：控制信号。  
R：输入寄存器端口D的计算结果，输出信号。  
Z：当值为1时代表两个输入信号值相等，当值为0时代表两个输入信号不等，输出信号。

5. 每一类的一条CPU指令的仿真

* Addi 操作
* Ori操纵
* Add 操作
* Or 操作

7.问题。

一直循环读取第一条，第二条指令  
解决方案：在cpu中需要一个指令存储器instmem，接口如下：

| **接口类型** | **名称** | **含义** |
| --- | --- | --- |
| input[31:0] | address | 下一条指令在指令存储器中的位置 |
| output[31:0] | inst | 读出来的指令 |

读出什么指令，取决于两方面

* 指令寄存器中的数据是否正确
* 传入的地址是否正确

通过观察modelsim看到，address一直在0,1之间徘徊，也没有报地址溢出的错误，所以问题关键，肯定在于address。  
对于address，也需要从两方面进行考虑

* 外部传入时，是否出错
* 在内部处理时，是否出错。

在debug过程中，在分析清楚出错可能的原因之后，一定是先从容易的方面入手。很幸运，在检查内部处理的部分中，找到了根源。在过程中使用了一个临时变量“接”address，实际应该是用一个32位的临时变量，但编写过程中，不小心使用了一个一位的变量，导致这个临时变量不是0就是1。

8. 源代码

```Verilog

module minisys(

clock,

reset,

instruction\_out,

PC\_plus\_4\_out,

PC\_out,

Jrn\_out,

RegDST\_out,

ALUSrc\_out,

MemtoReg\_out,

Lw\_out,

Lb\_out,

Lbu\_out,

Lh\_out,

Lhu\_out,

MemWrite\_out,

Sw\_out,

Sb\_out,

Sh\_out,

RegWrite\_out,

Branch\_out,

nBranch\_out,

bgez\_out,

bgtz\_out,

blez\_out,

bltz\_out,

bgezal\_out,

bltzal\_out,

Jmp\_out,

Jal\_out,

Jalr\_out,

Sftmd\_out,

MD\_out,

I\_format\_out,

ALUOp\_out,

Mfhi\_out,

Mflo\_out,

Mthi\_out,

Mtlo\_out,

break1\_out,

syscall\_out,

eret\_out,

Mfc0\_out,

Mtc0\_out,

Mult\_out,

Multu\_out,

Div\_out,

Divu\_out,

read\_data\_1\_out,

read\_data\_2\_out,

Sign\_extend\_out,

read\_data\_out,

Zero\_out,

EtZero\_out,

GtZero\_out,

LtZero\_out,

ALU\_result\_out,

ALU\_result\_HI\_out,

ALU\_result\_LO\_out,

write\_data\_out\_idcode,

write\_register\_address\_out,

opcplus4\_out,

MemorIOtoReg\_out,

IOWrite\_out,

IORead\_out,

int0,

int1,

ioread\_data\_out,

ioread\_data\_key\_out,

ioread\_data\_ctc\_out,

ioread\_data\_uart\_out,

led0,

led1,

led2,

led3,

col,

pulse0,

pulse1,

cout0,

cout1,

line,

xtal,

rxd,

txd,

rst,

pwm,

prst, //按钮复位信号

registerq\_out,

registerr\_out,

intPC\_out

);

//输入

input clock;

input reset;

input prst;

input int0,int1;

input[3:0] col;

output[3:0]line;

//输出

//取指

output[31:0] instruction\_out;

output[16:0] PC\_plus\_4\_out;

output[16:0] PC\_out;

output[16:0]opcplus4\_out;

//控制单元

output Jrn\_out;

output RegDST\_out;

output ALUSrc\_out;

output MemtoReg\_out;

output Lw\_out;

output Lb\_out;

output Lbu\_out;

output Lh\_out;

output Lhu\_out;

output MemWrite\_out;

output Sw\_out;

output Sb\_out;

output Sh\_out;

output RegWrite\_out;

output Branch\_out;

output nBranch\_out;

output bgez\_out;

output bgtz\_out;

output blez\_out;

output bltz\_out;

output bgezal\_out;

output bltzal\_out;

output Jmp\_out;

output Jal\_out;

output Jalr\_out;

output Sftmd\_out;

output MD\_out;

output I\_format\_out;

output[1:0] ALUOp\_out;

output Mfhi\_out;

output Mflo\_out;

output Mthi\_out;

output Mtlo\_out;

output break1\_out;

output syscall\_out;

output eret\_out;

output Mfc0\_out;

output Mtc0\_out;

output Mult\_out;

output Multu\_out;

output Div\_out;

output Divu\_out;

output MemorIOtoReg\_out;

output IOWrite\_out;

output IORead\_out;

//译码单元

output[31:0] read\_data\_1\_out;

output[31:0] read\_data\_2\_out;

output[31:0] Sign\_extend\_out;

output[31:0] read\_data\_out;

output[31:0] write\_data\_out\_idcode;

output[4:0] write\_register\_address\_out;

//执行单元

output[31:0] ALU\_result\_out;

output[31:0] ALU\_result\_HI\_out;

output[31:0] ALU\_result\_LO\_out;

output Zero\_out;

output EtZero\_out;

output GtZero\_out;

output LtZero\_out;

//中断

output[15:0]ioread\_data\_out;

output[15:0] ioread\_data\_key\_out;

output[15:0] ioread\_data\_ctc\_out;

output[15:0] ioread\_data\_uart\_out;

output[31:0] registerq\_out;

output[31:0] registerr\_out;

output[16:0] intPC\_out;

//LED16

output[6:0] led0;//LG

output[6:0] led1;

output[6:0] led2;

output[6:0] led3;

//ctc16

input pulse0;

input pulse1;

output cout0;

output cout1;

//pmw16

output pwm; //输出方波信号

//uart16

input xtal;

input rxd;

output txd;

//看门狗

output rst; //输出的复位信号

//中间线

//取指令

wire[31:0] instruction;

wire[16:0] pc\_plus\_4;

wire[16:0] pc;

wire[16:0] opcplus4; //JAL的跳转地址

//控制单元

wire Jrn;

wire RegDST;

wire ALUSrc;

wire MemtoReg;

wire Lw;

wire Lb;

wire Lbu;

wire Lh;

wire Lhu;

wire MemWrite;

wire Sw;

wire Sb;

wire Sh;

wire RegWrite;

wire Branch;

wire nBranch;

wire bgez;

wire bgtz;

wire blez;

wire bltz;

wire bgezal;

wire bltzal;

wire Jmp;

wire Jal;

wire Jalr;

wire Sftmd;

wire MD;

wire I\_format;

wire[1:0] ALUOp;

wire Mfhi;

wire Mflo;

wire Mthi;

wire Mtlo;

wire break1;

wire syscall;

wire eret;

wire Mfc0;

wire Mtc0;

wire Mult;

wire Multu;

wire Div;

wire Divu;

wire MemorIOtoReg;

wire IOWrite;

wire IORead;

//译码单元

wire[31:0] read\_data\_1;

wire[31:0] read\_data\_2;

wire[31:0] Sign\_extend;

wire[31:0] read\_data;

wire[31:0] write\_data\_out1;

wire[4:0] write\_register\_address;

//执行单元

wire[31:0] alu\_result;

wire[31:0] alu\_result\_hi;

wire[31:0] alu\_result\_lo;

wire[16:0] add\_result;

wire Zero;

wire EtZero;

wire GtZero;

wire LtZero;

//接口部分的wire

wire[15:0] ioread\_data;//IO来的数据。

wire[31:0] rdata;//来自存储器或者IO->寄存器

wire[31:0] write\_data;//写到IO或者存储器

wire[15:0] ioread\_data\_key;

wire[15:0] ioread\_data\_ctc;

wire[15:0] ioread\_data\_uart;

wire[16:0] address;

wire ledctrl;

wire keyctrl;

wire ctcctrl;

wire pwmctrl;

wire uartctrl;

wire wdtctrl;

wire swctrl;

wire dispctrl;

//中断

wire int0,int1;

wire[1:0] imask;

wire[16:0] intPC;

wire[4:0] read\_register\_1\_address;

//output 赋值

assign instruction\_out=instruction;

assign PC\_plus\_4\_out=pc\_plus\_4;

assign PC\_out=pc;

assign opcplus4\_out=opcplus4;

assign Jrn\_out=Jrn;

assign RegDST\_out=RegDST;

assign ALUSrc\_out=ALUSrc;

assign MemtoReg\_out=MemtoReg;

assign Lw\_out=Lw;

assign Lb\_out=Lb;

assign Lbu\_out=Lbu;

assign Lh\_out=Lh;

assign Lhu\_out=Lhu;

assign MemWrite\_out=MemWrite;

assign Sw\_out=Sw;

assign Sb\_out=Sb;

assign Sh\_out=Sh;

assign RegWrite\_out=RegWrite;

assign Branch\_out=Branch;

assign nBranch\_out=nBranch;

assign bgez\_out=bgez;

assign bgtz\_out=bgtz;

assign blez\_out=blez;

assign bltz\_out=bltz;

assign bgezal\_out=bgezal;

assign bltzal\_out=bltzal;

assign Jmp\_out=Jmp;

assign Jal\_out=Jal;

assign Jalr\_out=Jalr;

assign Sftmd\_out=Sftmd;

assign MD\_out=MD;

assign I\_format\_out=I\_format;

assign ALUOp\_out=ALUOp;

assign Mfhi\_out=Mfhi;

assign Mflo\_out=Mflo;

assign Mthi\_out=Mthi;

assign Mtlo\_out=Mtlo;

assign break1\_out=break1;

assign syscall\_out=syscall;

assign eret\_out=eret;

assign Mfc0\_out=Mfc0;

assign Mtc0\_out=Mtc0;

assign Mult\_out=Mult;

assign Multu\_out=Multu;

assign Div\_out=Div;

assign Divu\_out=Divu;

assign MemorIOtoReg\_out=MemorIOtoReg;

assign IOWrite\_out=IOWrite;

assign IORead\_out=IORead;

//译码单元

assign read\_data\_1\_out=read\_data\_1;

assign read\_data\_2\_out=read\_data\_2;

assign Sign\_extend\_out=Sign\_extend;

assign read\_data\_out=read\_data;

assign write\_data\_out\_idcode=write\_data\_out1;

assign write\_register\_address\_out=write\_register\_address;

//执行单元

assign ALU\_result\_out=alu\_result;

assign ALU\_result\_HI\_out=alu\_result\_hi;

assign ALU\_result\_LO\_out=alu\_result\_lo;

assign Zero\_out=Zero;

assign EtZero\_out=EtZero;

assign GtZero\_out=GtZero;

assign LtZero\_out=LtZero;

//RAM

assign write\_data\_out=(MemtoReg==1)?read\_data:alu\_result;

//中断接口

assign ioread\_data\_out=ioread\_data;

assign ioread\_data\_key\_out=ioread\_data\_key;

assign ioread\_data\_ctc\_out=ioread\_data\_ctc;

assign ioread\_data\_uart\_out=ioread\_data\_uart;

assign intPC\_out=intPC;

//取指单元

Ifect32 ifetch(

.Instrution(instruction),

.PC\_plus\_4\_out(pc\_plus\_4),

.Add\_result(add\_result), //来自执行单元

.read\_data\_1(read\_data\_1), //rs->pc

.nBranch(nBranch),

.Branch(Branch),

.bgez(bgez),

.bgtz(bgtz),

.blez(blez),

.bltz(bltz),

.bgezal(bgezal),

.bltzal(bltzal),

.Jmp(Jmp),

.Jal(Jal),

.Jrn(Jrn),

.Zero(Zero),

.EtZero(EtZero),

.GtZero(GtZero),

.LtZero(LtZero),

.PC\_out(pc),

.clock(clock),

.reset(reset),

.opcplus4(opcplus4),

.Jalr(Jalr),

.int0(int0),

.int1(int1),

.intPC(intPC),

.imask(imask),

.read\_register\_1\_address(read\_register\_1\_address)

);

//控制单元

control32 control(

.Opcode(instruction[31:26]),

.Funtion\_opcode(instruction[5:0]),

.rs(instruction[20:16]),

.Alu\_result(alu\_result),

.Jrn(Jrn),

.RegDST(RegDST),

.ALUSrc(ALUSrc),

.MemtoReg(MemtoReg),

.Lw(Lw),

.Lb(Lb),

.Lbu(Lbu),

.Lh(Lh),

.Lhu(Lhu),

.MemWrite(MemWrite),

.Sw(Sw),

.Sb(Sb),

.Sh(Sh),

.RegWrite(RegWrite),

.Branch(Branch),

.nBranch(nBranch),

.bgez(bgez),

.bgtz(bgtz),

.blez(blez),

.bltz(bltz),

.bgezal(bgezal),

.bltzal(bltzal),

.Jmp(Jmp),

.Jal(Jal),

.Jalr(Jalr),

.Sftmd(Sftmd),

.MD(MD),

.I\_format(I\_format),

.ALUOp(ALUOp),

.Mfhi(Mfhi),

.Mflo(Mflo),

.Mthi(Mthi),

.Mtlo(Mtlo),

.break1(break1),

.syscall(syscall),

.eret(eret),

.Mfc0(Mfc0),

.Mtc0(Mtc0),

.Mult(Mult),

.Multu(Multu),

.Div(Div),

.Divu(Divu),

.MemorIOtoReg(MemorIOtoReg),

.IOWrite(IOWrite),

.IORead(IORead)

);

//译码单元

Idecode32 idecode(

.read\_data\_1(read\_data\_1),

.read\_data\_2(read\_data\_2),

.Instruction(instruction),

.read\_data(read\_data),

.ALU\_result(alu\_result),

.ALU\_result\_HI(alu\_result\_hi),

.ALU\_result\_LO(alu\_result\_lo),

.Jal(Jal),

.Jalr(Jalr),

.bgezal(bgezal),

.bltzal(bltzal),

.mfhi(Mfhi),

.mflo(Mflo),

.mthi(Mthi),

.mtlo(Mtlo),

.Lw(Lw),

.Lb(Lb),

.Lbu(Lbu),

.Lh(Lh),

.Lhu(Lhu),

.Sw(Sw),

.Sb(Sb),

.Sh(Sh),

.MD(MD),

.RegWrite(RegWrite),

.MemtoReg(MemtoReg),

.RegDst(RegDST),

.Sign\_extend(Sign\_extend),

.clock(clock),

.reset(reset),

.opcplus4(opcplus4),

.write\_data\_out(write\_data\_out1),

.write\_register\_address\_out(write\_register\_address),

.int0(int0),

.int1(int1),

.intPC(intPC),

.imask(imask),

.read\_register\_1\_address(read\_register\_1\_address),

.registerq\_out(registerq\_out),

.registerr\_out(registerr\_out)

);

//执行单元

Execute32 execute(

.read\_data\_1(read\_data\_1),

.read\_data\_2(read\_data\_2),

.Sign\_extend(Sign\_extend),

.Function\_opcode(instruction[5:0]),

.Exe\_opcode(instruction[31:26]),

.ALUOp(ALUOp),

.Shamt(instruction[10:6]),

.ALUSrc(ALUSrc),

.I\_format(I\_format),

.Sftmd(Sftmd),

.MD(MD),

.Mult(Mult),

.Multu(Multu),

.Div(Div),

.Divu(Divu),

.Zero(Zero),

.EtZero(EtZero),

.GtZero(GtZero),

.LtZero(LtZero),

.ALU\_Result(alu\_result),

.Add\_Result(add\_result),

.ALU\_Result\_HI(alu\_result\_hi),

.ALU\_Result\_LO(alu\_result\_lo),

.PC\_plus\_4(pc\_plus\_4)

);

/\*RAm(添加接口之前的RAMyy|zyl)

dmemory32 dmemory(

.read\_data(read\_data),

.address(alu\_result[16:0]),

.write\_data(read\_data\_2),

.MemWrite(MemWrite),

.clock(clock)

);\*/

//RAM

dmemory32 dmemory(

.read\_data(read\_data),

.address(address),

.write\_data(read\_data\_2),

.MemWrite(MemWrite),

.clock(clock)

);

memorio memio(.caddress(alu\_result[16:0]),

.address(address),

.memread(MemtoReg),

.memwrite(MemWrite),

.ioread(IORead),

.iowrite(IOWrite),

.mread\_data(read\_data),

.ioread\_data(ioread\_data),

.wdata(read\_data\_2),

.rdata(rdata),

.write\_data(write\_data),

.LEDCtrl(ledctrl),

.KEYCtrl(keyctrl),

.CTCCtrl(ctcctrl),

.PWMCtrl(pwmctrl),

.UARTCtrl(uartctrl),

.WDTCtrl(wdtctrl),

.DISPCtrl(dispctrl),

.SWCtrl(swctrl));

ioread multiioread(.reset(reset),

.clk(clock),

.ior(ioread),

.keyctrl(keyctrl),

.ctcctrl(ctcctrl),

.uartctrl(uartctrl),

.ioread\_data(ioread\_data),

.ioread\_data\_key(ioread\_data\_key),

.ioread\_data\_ctc(ioread\_data\_ctc),

.ioread\_data\_uart(ioread\_data\_uart));

/\*

disp16 disp(.reset(reset),

.data(write\_data[15:0]),

.cs(dispctrl),

.iow(iowrite),

.led0(led0),

.led1(led1),

.led2(led2),

.led3(led3));

\*/

/\*

key16 key(.reset(reset),

.cs(keyctrl),

.clk(clock),

.ior(ioread),

.address(address[1:0]),

.col(col),

.line(line),

.ioread\_data(ioread\_data\_key));\*/

/\*

ctc16 ctc( .reset(reset),

.clk(clock),

.cs(ctcctrl),

.iow(iowrite),

.ior(ioread),

.pulse0(pulse0),

.pulse1(pulse1),

.address(address[3:0]),

.iowrite\_data(write\_data[15:0]),

.ioread\_data(ioread\_data\_ctc),

.cout0(cout0),

.cout1(cout1));

pwm16 pwmcontrol(.reset(reset),

.address(address[2:0]),

.data(write\_data[15:0]),

.cs(pwmctrl),

.clk(clock),

.iow(iowrite),

.pwm(pwm));

uart16 uart(.reset(reset),

.clk(clock),

.cs(uartctrl),

.iow(iowrite),

.ior(ioread),

.xtal(xtal),

.address(address[1:0]),

.iowrite\_data(write\_data[7:0]),

.ioread\_data(ioread\_data\_uart),

.txd(txd),

.rxd(rxd));

wtd16 wtd(.reset(reset),

.clk(clock),

.cs(wdtctrl),

.iow(iowrite),

.data(write\_data[15:0]),

.rst(rst));

init32 init(.prst(prst),

.rst(rst),

.clk(clock),

.reset(reset)); \*/

endmodule

```

# **六、课设总结**

这次硬件课程设计共计 3 周时间。在实验开始时，由于之前提前学习过编译原理，我以为这次实验的状态机部分不会很复杂，只是把以前做过的几个课程实验拼凑下就可以了，但实际操作过程中我们还是遇到了很多麻烦。在写微指令时，我们对于某些指令流程不了解，尤其是移位、相对寻址的微指令比较复杂，于是我们先分析了书本上的微指令，写出相对简单的机器指令的微指令，然后查阅更多的资料，经过反复调试才弄清楚复杂指令的微指令流程。但在最终运行时，我们将第一条指令的地址书写错误了，一直循环读取第一条，第二条指令，经由同学的帮助指出我们才发现这个错误。

总的来说，这次实验是至今为止我们所做的一个比较复杂的实验，花费的时间也最多，当然收获也很多。首先，通过这次实验，我们巩固了组成原理微程序的相关知识；其次，我们进一步了解了各部件之间的分工与配合，对机器指令与微指令之间的关系有了更加直接清晰的认识；再者，这次实验提高了我们的硬件设计及动手能力，这对于计算机专业的学生是十分重要的。最后，谢谢老师在实验过程中的耐心讲解和指导。