**序号：**

****

硬件课程设计报告 (2019-2020学年)

姓 名： 张宇辰

学 号： 18151010208

班 级： 计18-2

专 业： 计算机科学与技术

所在系： 计算机系

指导教师： 廖联军

2020年 9月 5日

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

目录

[实验一、在7段数码管上显示自己的最后三位学号数字 4](#_Toc50283209)

[1. 数码管上显示数字的原理 4](#_Toc50283210)

[2. Verilog程序的功能框图 5](#_Toc50283211)

[3. 源代码 5](#_Toc50283212)

[4. 实验结果 7](#_Toc50283213)

[5. 实验中遇到的问题及解决方法 8](#_Toc50283214)

[实验二、在16×16点阵上显示自己的名字 10](#_Toc50283215)

[1. 16×16点阵上显示汉字的原理 10](#_Toc50283216)

[2. Verilog程序功能框图 11](#_Toc50283217)

[3. 源代码及相关说明 11](#_Toc50283218)

[4. 影响每一列扫描快慢的关键程序部分 14](#_Toc50283219)

[5. 影响名字扫描快慢的关键程序部分 14](#_Toc50283220)

[6. 实验结果 15](#_Toc50283221)

[7. 实验中遇到的问题及解决方法 16](#_Toc50283222)

[实验三、调用两位数十进制加法器的IP核，生成多位数十进制的加法器 19](#_Toc50283223)

[1. 两位数十进制加法器 19](#_Toc50283224)

[2. 设计思路及程序功能框图 20](#_Toc50283225)

[3. 多位十进制加法电路图 21](#_Toc50283226)

[4. 源代码 21](#_Toc50283227)

[5. 实验结果 22](#_Toc50283228)

[6. 实验中遇到的问题及解决方法 23](#_Toc50283229)

[实验四、序列检测器 25](#_Toc50283230)

[1. 序列检测器原理 25](#_Toc50283231)

[2. 设计思路及程序功能框图 26](#_Toc50283232)

[3. 源代码 27](#_Toc50283233)

[4. 实验结果 29](#_Toc50283234)

[实验五、简单CPU设计 31](#_Toc50283235)

[1. CPU指令系统 31](#_Toc50283236)

[2. 每一类的一条CPU指令的实现 32](#_Toc50283237)

[3. 指令实现数据流图 34](#_Toc50283238)

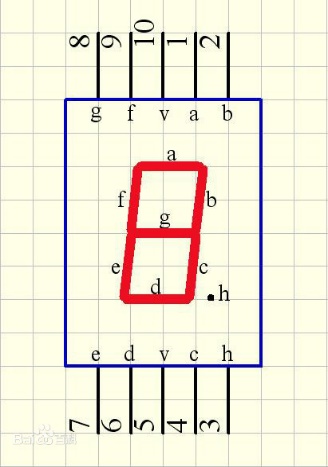
[4. CPU指令的测试与仿真 35](#_Toc50283239)

[5. 源代码 37](#_Toc50283240)

[实验设计心得 44](#_Toc50283241)

# 实验一、在7段数码管上显示自己的最后三位学号数字

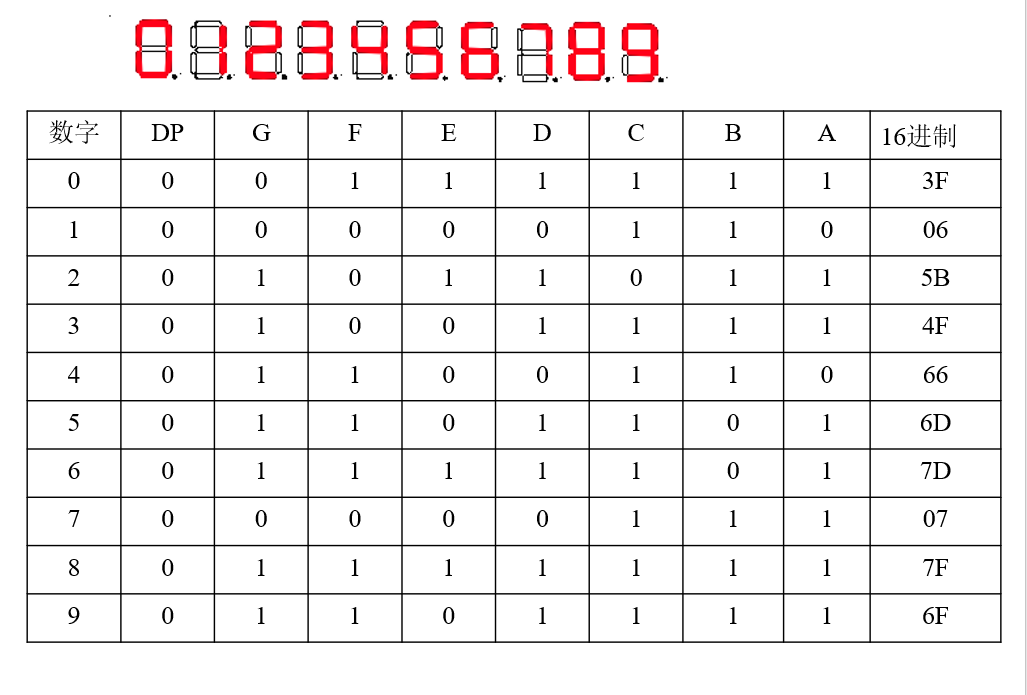
## 数码管上显示数字的原理

* 1. 是一种半导体发光器件，基本单元是发光二极管，按照LED灯管段数划分为七段数码管和八段数码管。八段比七段多一个小数点。
  2. 分为共阳极和共阴极两种类型：
     1. 共阴极就是将八个LED的阴极连接在一起，让其接地，给任何一个LED的另一端高电平，就可以点亮。
     2. 共阳极是将七个LED的阳极连在一起，给任何一个LED的另一端低电平，就可以点亮。
  3. 通过一个4\_7译码器可以将需要显示的数，转换为相应数码管信号，信号对应相应位置的灯管，通过不同的码，控制不同灯管的开启和关闭，最终拼出需要的数字。
  4. 采用共阴极，信号为1则对应的管亮，为零则不亮。例：想让管a亮则输入的信号的第一位为1
  5. 七段数码管的灯管顺序：

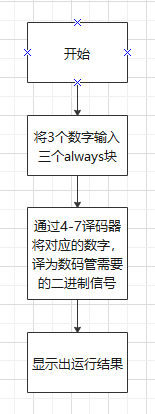
注：远程实验平台中的灯管顺序为

（a、g，f、e和b、c位置互换）

显示不同数字需要的信号：



## Verilog程序的功能框图



## 源代码

* 1. 代码1：通过21个管脚控制3个数码管的显示

module decode4\_7(

input [3:0] Incode1, //最大数为9，输入为4位

input [3:0] Incode2,

input [3:0] Incode3,

output [6:0] Outcode1, //七段数码管，需要七位控制信号，输出为7位

output [6:0] Outcode2,

output [6:0] Outcode3

);

reg[6:0]Outcode1;

reg[6:0]Outcode2;

reg[6:0]Outcode3;

always @(Incode1) //一号数码管，显示第一个数字2

begin

case(Incode1)

4'd0:Outcode1 = 7'b0111111; //译码

4'd1:Outcode1 = 7'b0000110; //例：输入1，在数码管上为最右侧两管亮对应为1，其余为0

4'd2:Outcode1 = 7'b1011011;

4'd3:Outcode1 = 7'b1001111;

4'd4:Outcode1 = 7'b1100110;

4'd5:Outcode1 = 7'b1101101;

4'd6:Outcode1 = 7'b1111101;

4'd7:Outcode1 = 7'b0000111;

4'd8:Outcode1 = 7'b1111111;

4'd9:Outcode1 = 7'b1101111;

default:Outcode1 = 7'bx;

endcase;

end;

always @(Incode2) //二号数码管，显示第二个数字0

begin

case(Incode2)

4'd0:Outcode2 = 7'b0111111; //译码

4'd1:Outcode2 = 7'b0000110;

4'd2:Outcode2 = 7'b1011011;

4'd3:Outcode2 = 7'b1001111;

4'd4:Outcode2 = 7'b1100110;

4'd5:Outcode2 = 7'b1101101;

4'd6:Outcode2 = 7'b1111101;

4'd7:Outcode2 = 7'b0000111;

4'd8:Outcode2 = 7'b1111111;

4'd9:Outcode2 = 7'b1101111;

default:Outcode2 = 7'bx;

endcase;

end;

always @(Incode3) //三号数码管，显示第三个数字8

begin

case(Incode3)

4'd0:Outcode3 = 7'b0111111; //译码

4'd1:Outcode3 = 7'b0000110;

4'd2:Outcode3 = 7'b1011011;

4'd3:Outcode3 = 7'b1001111;

4'd4:Outcode3 = 7'b1100110;

4'd5:Outcode3 = 7'b1101101;

4'd6:Outcode3 = 7'b1111101;

4'd7:Outcode3 = 7'b0000111;

4'd8:Outcode3 = 7'b1111111;

4'd9:Outcode3 = 7'b1101111;

default:Outcode3 = 7'bx;

endcase;

end;

endmodule

* 1. 代码2：通过8个输出管脚控制3个数码管的显示

module seven(

input clk,

output [2:0]choice,

output [6:0]Outcode

);

reg [6:0]Outcode;

reg [1:0]choice = 2'b01;

integer i;

always@(posedge clk)

begin

case(choice)

2'b01:Outcode = 7'b1011011;

2'b10:Outcode = 7'b0111111;

2'b11:Outcode = 7'b1111111;

endcase

if(choice == 2'b11)

choice <= 2'b01;

else

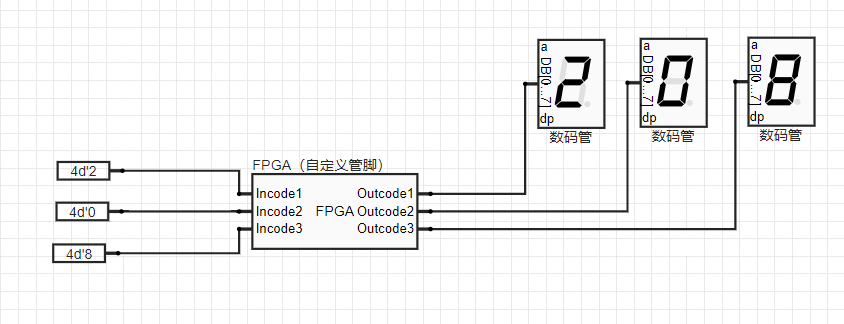
choice <= choice + 2'b01;

end

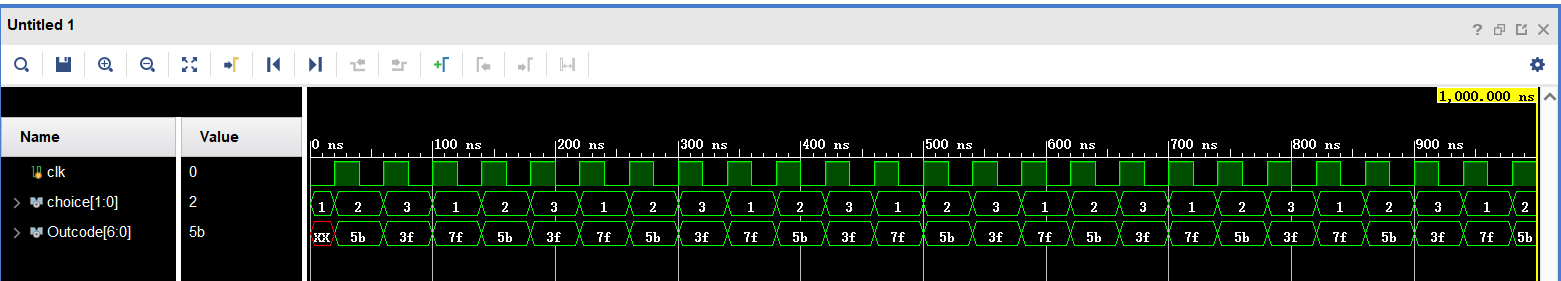
endmodule

## 实验结果

1. 代码1的实验结果



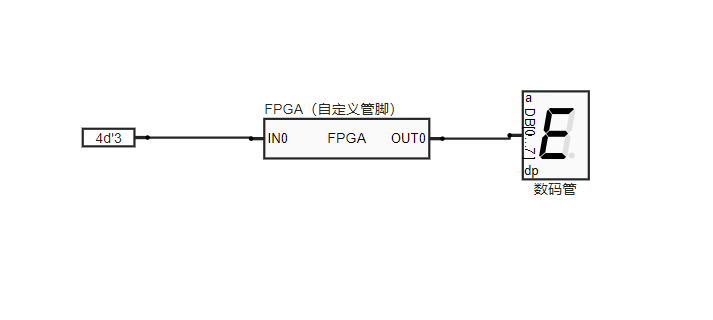
1. 代码2的实验结果



## 实验中遇到的问题及解决方法

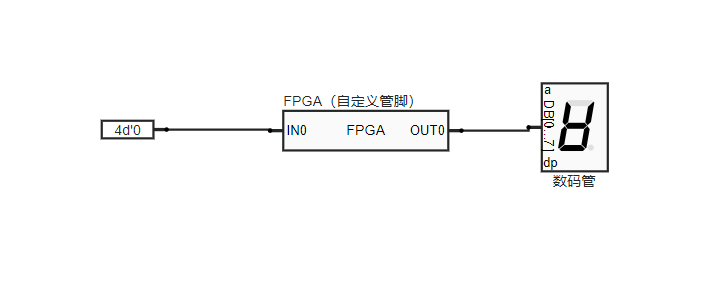
* 1. 问题1

数字是反的



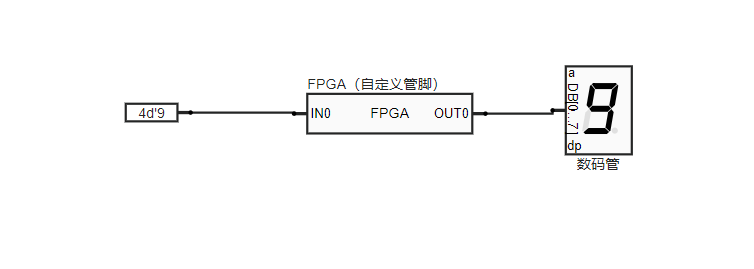
* 1. 问题2

数字不反了，但出现新问题，远程平台中的a和g的位置与，导致零出现中间横线没有上横线



* 1. 问题的原因与解决

问题原因是由于远程平台上，数码管中LED对应的信号位置，与所查资料不同，造成了没有在正确的位置显示的问题。正确的LED对应位置，在上述实验原理中进行了说明。



module decode4\_7(

input [3:0] Incode,

output [6:0] Outcode

);

reg[6:0]Outcode;

always @(Incode)

begin

case(Incode)

4'd0:Outcode = 7'b0111111; //译码

4'd1:Outcode = 7'b0000110;

4'd2:Outcode = 7'b1011011;

4'd3:Outcode = 7'b1001111;

4'd4:Outcode = 7'b1100110;

4'd5:Outcode = 7'b1101101;

4'd6:Outcode = 7'b1111101;

4'd7:Outcode = 7'b0000111;

4'd8:Outcode = 7'b1111111;

4'd9:Outcode = 7'b1101111;

default:Outcode = 7'bx;

endcase;

end;

endmodule

# 实验二、在16×16点阵上显示自己的名字

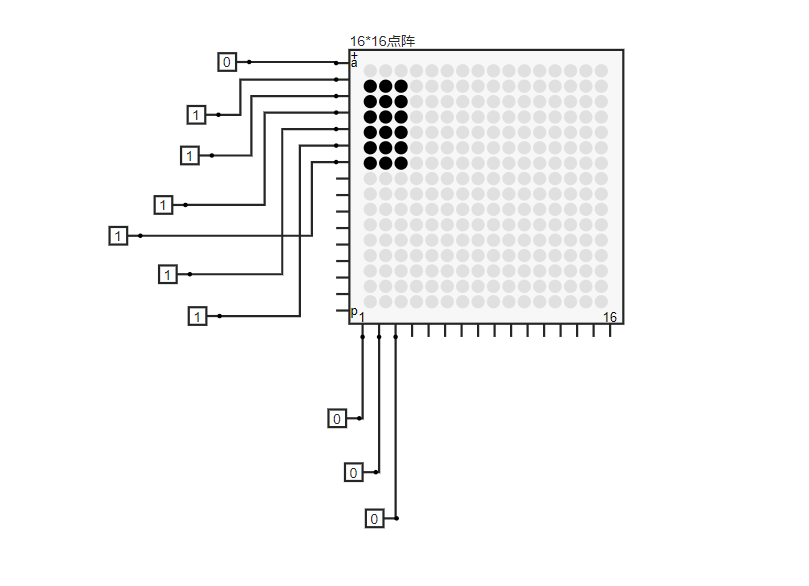
## 16×16点阵上显示汉字的原理

1. 使用256个发光二极管，组成16行16列的发光点阵。国际汉字库中的每一个字均有16行×16列的256点阵来表示。每个点为一个像素，每个字为一个图像。
2. 工作原理与8位数码管类似。通过16个共阴极输出端口，每个共阴极对应16个LED显示灯，所以其扫描译码地址需要4位地址线，汉字扫描码由16位段地址输入
3. 通过时钟对每列进行扫描，每一个时钟显示一列的文字信息，显示完整文字。
4. 扫描三种方式：

1. 点扫描：扫描频率要大于1024Hz

2. 行扫描：扫描频率大于128Hz

3. 列扫描：扫描频率大于128Hz

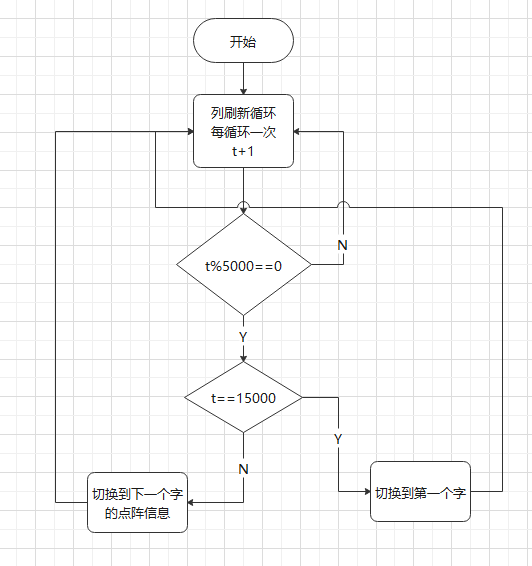


Y轴，信号为1，这行灯管亮，信号为0，这行不亮

X轴，信号为0，这列灯光亮，信号为1，这列不亮

需要足够的扫描频率才能符合视觉暂留，形成完整图像。

## Verilog程序功能框图



## 源代码及相关说明

* 1. Column控制哪一列显示，“0”为该列显示，“1”为该列不显示
  2. Row控制哪一行显示，“1”为该行显示，“0”为该行不显示
  3. Column和Row均是左下角为第一位点阵信息。
  4. 采用列刷新的方法
  5. 远程平台中，Column接16×16的底下的管脚，Row接左侧管脚
  6. 通过对于运行次数的计数，实现0.5秒的汉字切换，在代码中记为t。
  7. 每一个时钟运行一次always语句。
  8. 代码中使用10Hz的时钟频率，执行5000次always语句中的内容为0.5秒，always中添加条件语句，每5000次换一个字。

module display\_decode(

output [15:0] Column, //列

output [15:0]Row,

input clk

);

reg [15:0] Column; //点阵中哪一列亮

reg[15:0] Row; //一列中的哪一排亮

integer i= 0;//列的循环的标记

integer t = 1;//always块的运行次数标记

always@(posedge clk) //一个时钟刷新一行 时钟越快刷新速度越快

begin

case(i) //列刷新的方式，每次前进一列 不管是哪个字都需要进行列刷新

0:Column <= 16'b1111\_1111\_1111\_1110;

1:Column <= 16'b1111\_1111\_1111\_1101;

2:Column <= 16'b1111\_1111\_1111\_1011;

3:Column <= 16'b1111\_1111\_1111\_0111; //前4列的行显示情况

4:Column <= 16'b1111\_1111\_1110\_1111;

5:Column <= 16'b1111\_1111\_1101\_1111;

6:Column <= 16'b1111\_1111\_1011\_1111;

7:Column <= 16'b1111\_1111\_0111\_1111;//5-8显示情况

8:Column <= 16'b1111\_1110\_1111\_1111;

9:Column <= 16'b1111\_1101\_1111\_1111;

10:Column <= 16'b1111\_1011\_1111\_1111;

11:Column <= 16'b1111\_0111\_1111\_1111;//9-12

12:Column <= 16'b1110\_1111\_1111\_1111;

13:Column <= 16'b1101\_1111\_1111\_1111;

14:Column <= 16'b1011\_1111\_1111\_1111;

15:Column <= 16'b0111\_1111\_1111\_1111;//13-16

endcase

if(t <= 5000) //张字

begin

case(i) //每列显示的点阵信息

0:Row <= 16'b0010\_0000\_0000\_0010;

1:Row <= 16'b0100\_0011\_1110\_0010;

2:Row <= 16'b1000\_0010\_0010\_0010;

3:Row <= 16'b0100\_0010\_0010\_0010; //前4列的行显示情况

4:Row <= 16'b0011\_1110\_0011\_1111;

5:Row <= 16'b0000\_0000\_0100\_0010;

6:Row <= 16'b0000\_0000\_0100\_0000;

7:Row <= 16'b1111\_1111\_1111\_1111;//5-8显示情况

8:Row <= 16'b1000\_0000\_0100\_0000;

9:Row <= 16'b0100\_0001\_1101\_0000;

10:Row <= 16'b0000\_1100\_0100\_1000;

11:Row <= 16'b0001\_0010\_0100\_0100;//9-12

12:Row <= 16'b0010\_0000\_0100\_0110;

13:Row <= 16'b0110\_0000\_0110\_0000;

14:Row <= 16'b0010\_0000\_0100\_0000;

15:Row <= 16'b0000\_0000\_0000\_0000;//13-16

endcase

t = t+1;

end

else if(t>5000 && t<=10000) //宇字

begin

case(i)

0:Row <= 16'b0000\_0000\_0001\_0000;

1:Row <= 16'b0000\_0001\_0000\_1100;

2:Row <= 16'b0000\_0001\_0000\_0100;

3:Row <= 16'b0000\_0001\_0010\_0100;

4:Row <= 16'b0000\_0001\_0010\_0100;

5:Row <= 16'b0100\_0001\_0010\_0100;

6:Row <= 16'b1000\_0001\_0010\_0101;

7:Row <= 16'b0111\_1111\_1110\_0110;

8:Row <= 16'b0000\_0001\_0010\_0100;

9:Row <= 16'b0000\_0001\_0010\_0100;

10:Row <= 16'b0000\_0001\_0010\_0100;

11:Row <= 16'b0000\_0001\_0010\_0100;

12:Row <= 16'b0000\_0011\_0000\_0100;

13:Row <= 16'b0000\_0001\_0001\_0100;

14:Row <= 16'b0000\_0000\_0000\_1100;

15:Row <= 16'b0000\_0000\_0000\_0000;

endcase

t = t + 1;

end

else if(t>10000 && t<=15000) //辰

begin

case(i)

0:Row <= 16'b0100\_0000\_0000\_0000;

1:Row <= 16'b0011\_0000\_0000\_0000;

2:Row <= 16'b0000\_1111\_1111\_1110;

3:Row <= 16'b0000\_0000\_1001\_0010;

4:Row <= 16'b0000\_0000\_1001\_0010;

5:Row <= 16'b1000\_0000\_1001\_0010;

6:Row <= 16'b1111\_1111\_1001\_0010;

7:Row <= 16'b0010\_0011\_1001\_0010;

8:Row <= 16'b0000\_0100\_1001\_0010;

9:Row <= 16'b0000\_1000\_1001\_0010;

10:Row <= 16'b0001\_0100\_1001\_1010;

11:Row <= 16'b0001\_0010\_1001\_0011;

12:Row <= 16'b0010\_0000\_1100\_0010;

13:Row <= 16'b0110\_0000\_1000\_0000;

14:Row <= 16'b0010\_0000\_0000\_0000;

15:Row <= 16'b0000\_0000\_0000\_0000;

endcase

t = t + 1;

end

i = i + 1;

if(i==16) i = 0;

else if(t>15000) t=1;

end

endmodule

## 影响每一列扫描快慢的关键程序部分

通过时钟控制列刷新的速度，每一个时钟显示一行，通过控制时钟可以控制每一列的扫描的快慢，时钟越慢，扫描的越慢

module display\_decode(

output [15:0] Column, //列

output [15:0]Row,

input clk

);

integer i= 0;//列的循环的标记

integer t = 1;//always块的运行次数标记

always@(posedge clk) //一个时钟刷新一行 时钟越快列刷新速度越快

begin

case(i) //列刷新的方式，每次前进一列 不管是哪个字都需要进行列刷新

………（省略）

end

endmodule

## 影响名字扫描快慢的关键程序部分

通过设置一个变量t来控制每个字的显示时间，每循环5000次，也就是刷新5000行后换到下一个字，通过调整每个字循环的次数可以控制，每个字出现的时间。将刷新条件改为10000次，就可以让每个字的显示时间增长一倍。

integer t = 1;//always块的运行次数标记

always@(posedge clk) //一个时钟刷新一行 时钟越快刷新速度越快

begin

case(i) //列刷新的方式，每次前进一列 不管是哪个字都需要进行列刷新

if(t <= 5000) //张字

……

t = t+1;

end

else if(t>5000 && t<=10000) //宇字

……

t = t + 1;

end

else if(t>10000 && t<=15000) //辰

……（省略）

t = t + 1;

end

i = i + 1;

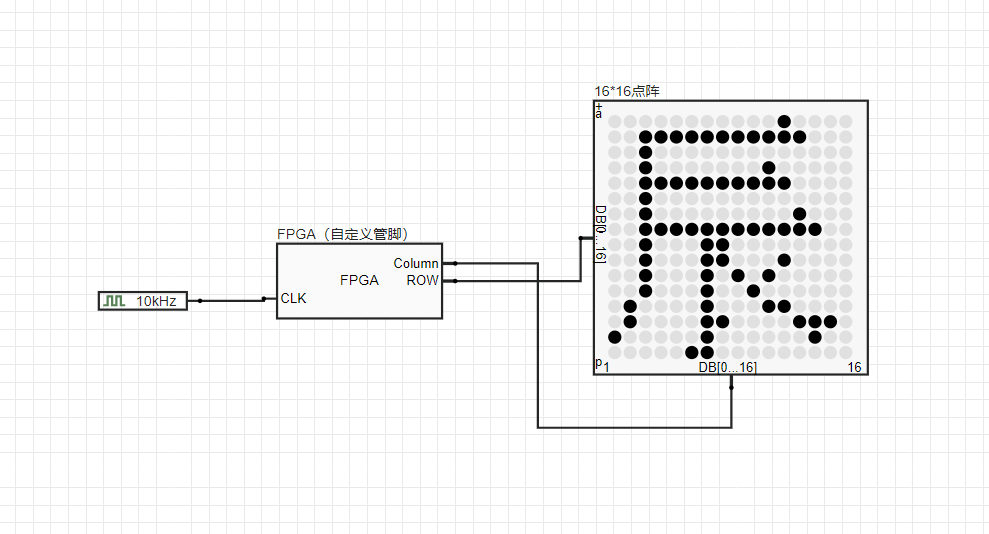
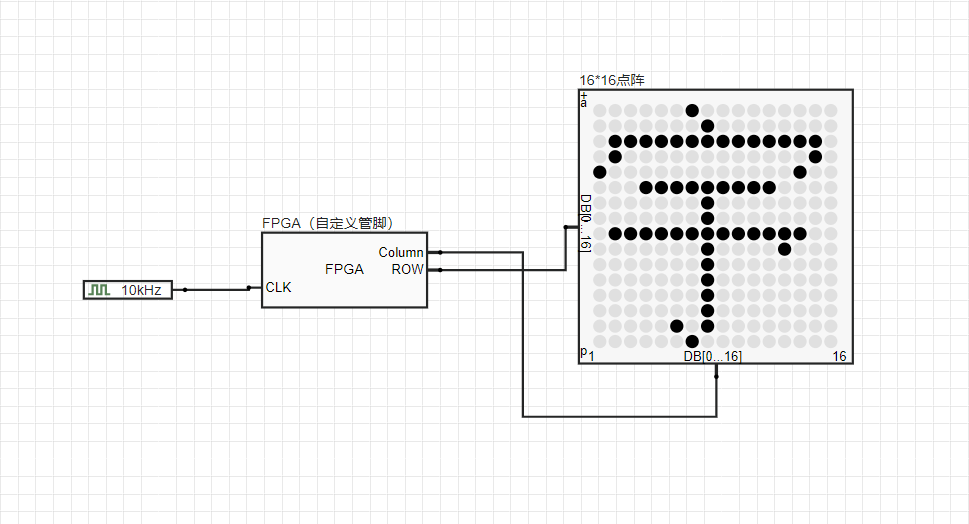
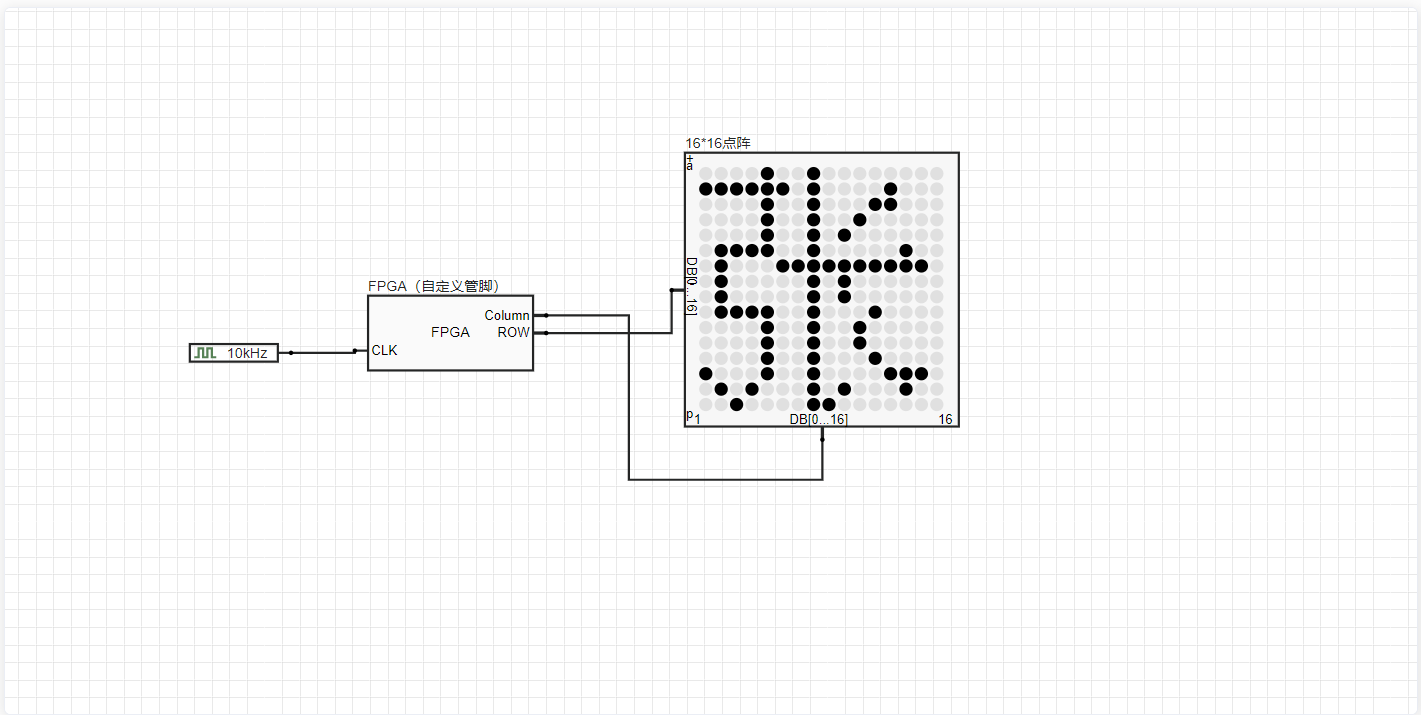
if(i==16) i = 0;

else if(t>15000) t=1;

end

endmodule

## 实验结果



## 实验中遇到的问题及解决方法

1. 问题1

输出的字是反的，说明Row信号给的是反的，点阵信息，最下面是第一个，最上面是一列中的最后一个。将给出的点阵信号进行对称，如下版代码所示字中有些点缺失需要调整，将i+1放在了begin的第一句话导致没有第一列显示

错误代码：

module display\_decode(

output [15:0] Column, //列

output [15:0]Row,

input clk

);

reg[15:0] Row;

reg [15:0] Column;

integer i= 1;

always@(posedge clk)

begin

i = i + 1;

case(i) //列为零，行为1显示

1:Row <= 16'b1000\_0111\_1000\_0000;

2:Row <= 16'b1000\_0100\_1000\_0000;

3:Row <= 16'b1000\_0100\_1000\_0000;

4:Row <= 16'b1000\_0100\_1000\_0000; //前4列的行显示情况

5:Row <= 16'b1000\_0100\_1000\_0100;

6:Row <= 16'b1000\_0100\_1000\_0010;

7:Row <= 16'b1111\_1100\_1111\_1111;

8:Row <= 16'b1000\_0010\_0000\_0000;//5-8显示情况

9:Row <= 16'b1111\_1111\_1111\_1111;

10:Row <= 16'b0000\_0111\_0000\_0010;

11:Row <= 16'b0000\_1010\_1000\_0100;

12:Row <= 16'b0001\_0010\_1000\_0000;//9-12

13:Row <= 16'b0010\_0010\_0010\_0000;

14:Row <= 16'b0100\_0010\_0001\_0000;

15:Row <= 16'b1000\_0010\_0000\_1000;

16:Row <= 16'b0000\_0101\_0000\_0100;//13-16

endcase

case(i)

1:Column <= 16'b1111\_1111\_1111\_1110;

2:Column <= 16'b1111\_1111\_1111\_1101;

3:Column <= 16'b1111\_1111\_1111\_1011;

4:Column <= 16'b1111\_1111\_1111\_0111; //前4列的行显示情况

5:Column <= 16'b1111\_1111\_1110\_1111;

6:Column <= 16'b1111\_1111\_1101\_1111;

7:Column <= 16'b1111\_1111\_1011\_1111;

8:Column <= 16'b1111\_1111\_0111\_1111;//5-8显示情况

9:Column <= 16'b1111\_1110\_1111\_1111;

10:Column <= 16'b1111\_1101\_1111\_1111;

11:Column <= 16'b1111\_1011\_1111\_1111;

12:Column <= 16'b1111\_0111\_1111\_1111;//9-12

13:Column <= 16'b1110\_1111\_1111\_1111;

14:Column <= 16'b1101\_1111\_1111\_1111;

15:Column <= 16'b1011\_1111\_1111\_1111;

16:Column <= 16'b0111\_1111\_1111\_1111;//13-16

endcase

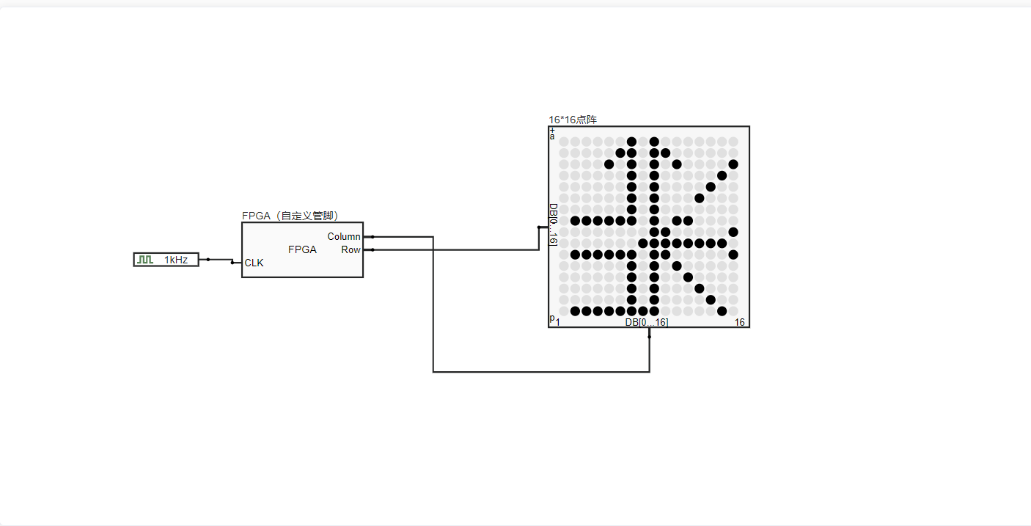
if(i==16)

i=1;

end

endmodule

错误效果：



解决方法：输入的信号是反的，更改输入的信号，让显示的字变成正常状态。

# 实验三、调用两位数十进制加法器的IP核，生成多位数十进制的加法器

## 两位数十进制加法器

module adder(

input [8:0] addend\_1,

input [8:0] addend\_2,

output [8:0] sum，

input cin, //向本器件进位

output cout //进位

);

reg [8:0]sum;

reg cout;

always@(addend\_1 or addend\_2 or cin)

begin

if( addend\_1 + addend\_2 + cin < 7'd100) //不用进位的情况

begin

sum = addend\_1 + addend\_2 + cin;

cout = 0;

end

else if( addend\_1 + addend\_2 + cin > 7'd99) //需要进位的情况

begin

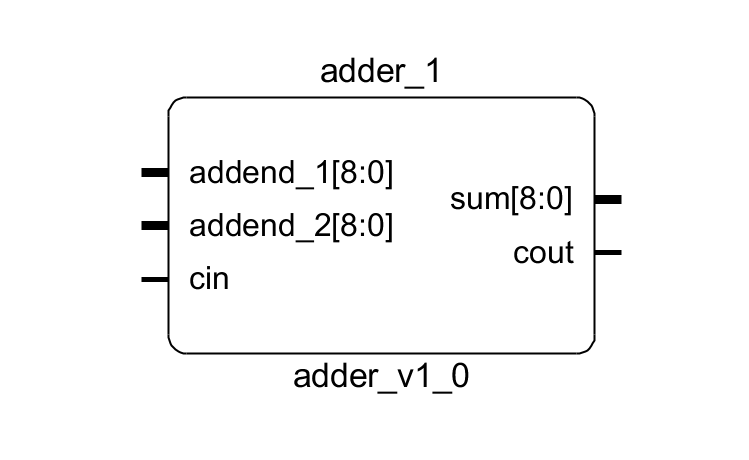
sum = addend\_1 + addend\_2 + cin - 7'd100;

cout = 1;

end

end

endmodule

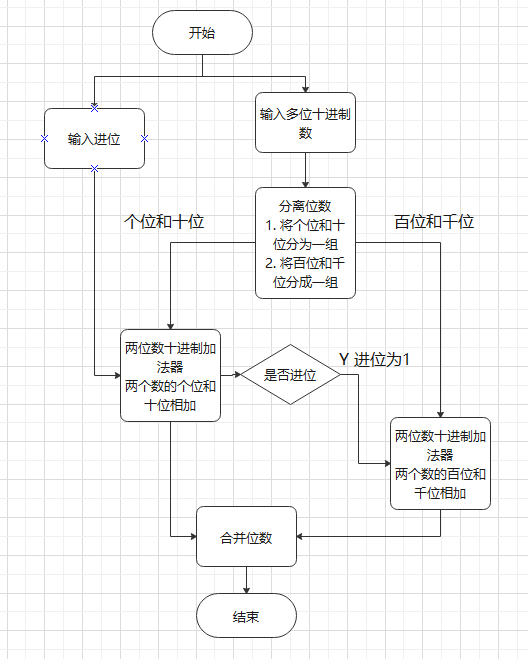


## 设计思路及程序功能框图

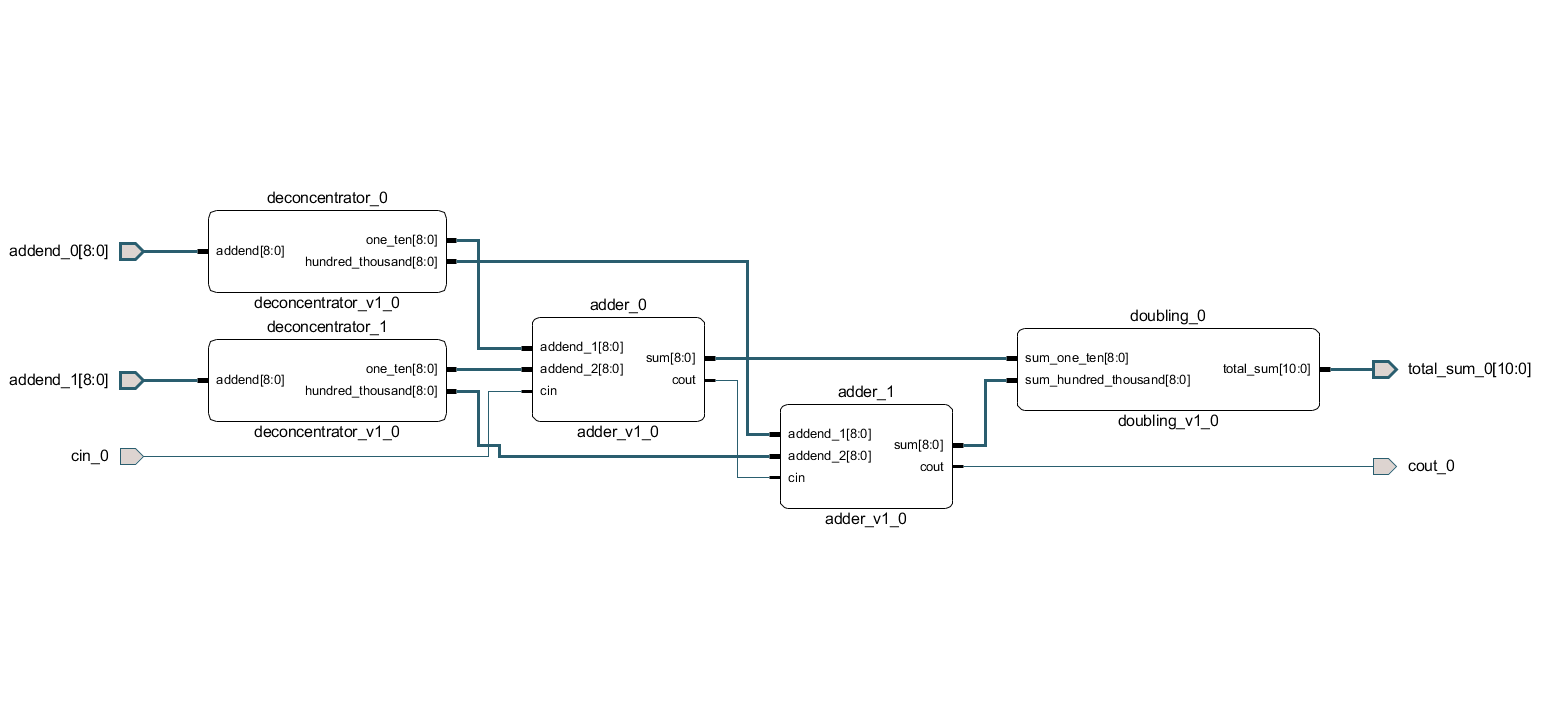
* 1. 设计思路

两位十进制加法器只能输入两位的十进制，想要输入多位的十进制数，需要通过一个分线器，分离位数。将个位和十位为一组，百位和千位为一组，分别经过两位十进制加法器计算。最后将计算完的数据，输入到位数合并器件，根据位数，乘以相应的位数，最终输出正确的答案。

* 1. 程序功能框图



## 多位十进制加法电路图



## 源代码

* 1. 位数分离器（deconcentrator）

module deconcentrator(

input [8:0] addend,

output [8:0] one\_ten, //将输入的分离为个位和十位

output [8:0] hundred\_thousand //百位，千位

);

assign one\_ten = addend % 100;

assign hundred\_thousand = (addend - one\_ten) / 100;

endmodule

* 1. 两位十进制加法器（adder）

module adder(

input [8:0] addend\_1,

input [8:0] addend\_2,

output [8:0] sum，

input cin, //向本器件进位

output cout //进位

);

reg [8:0]sum;

reg cout;

always@(addend\_1 or addend\_2 or cin)

begin

if( addend\_1 + addend\_2 + cin < 7'd100) //不用进位的情况

begin

sum = addend\_1 + addend\_2 + cin;

cout = 0;

end

else if( addend\_1 + addend\_2 + cin > 7'd99) //需要进位的情况

begin

sum = addend\_1 + addend\_2 + cin - 7'd100;

cout = 1;

end

end

endmodule

* 1. 位数合并器（doubling）

module doubling(

input [8:0] sum\_one\_ten,

input [8:0] sum\_hundred\_thousand,

output [10:0] total\_sum

);

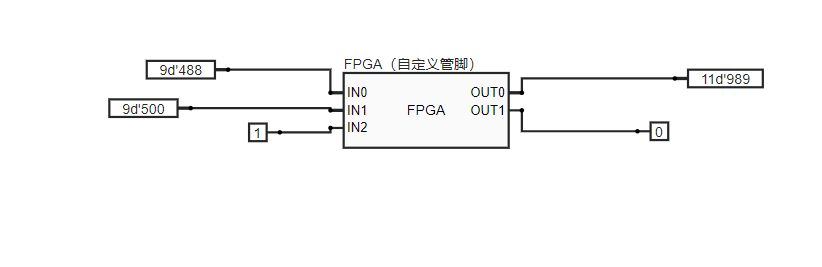
assign total\_sum = sum\_one\_ten + sum\_hundred\_thousand \* 100;

endmodule

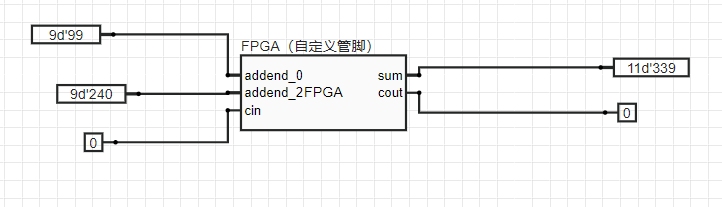
* 1. 相应参数的备注：
     1. addend\_1：加数1
     2. addend\_2：加数2
     3. deconcentrator\_addend1：分位器1 one\_ten:为个位和十位输出 hundred\_thousand :为百位千位输出
     4. deconcentrator\_addend2：分位器2
     5. doubling\_0：并位器 sum\_one\_ten：两个数个位和十位的和 sum\_hundred\_thousand:为两个数百位千位的和
     6. total\_sum：总和
     7. cin：上一加法器进位
     8. cout：向下一加法器进位
     9. one\_ten：计算个位和十位两位的加法器
     10. hundred\_thousand：计算百位和千位两位的加法器

## 实验结果

* 1. 有进位输入的情况

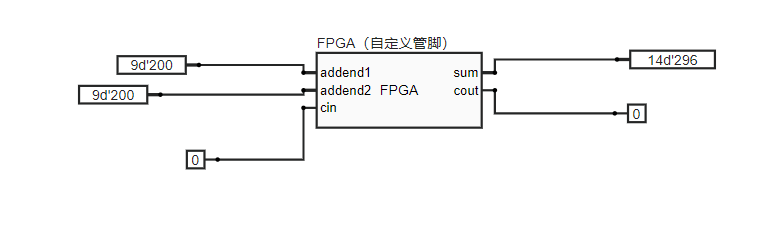
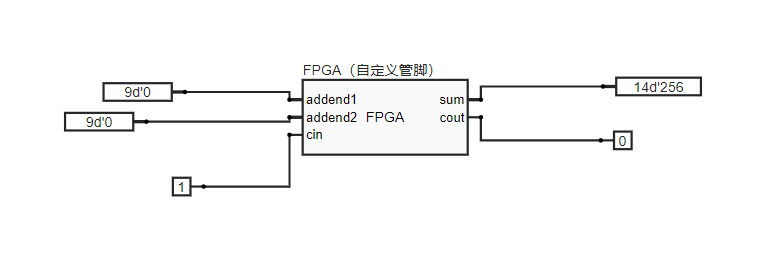


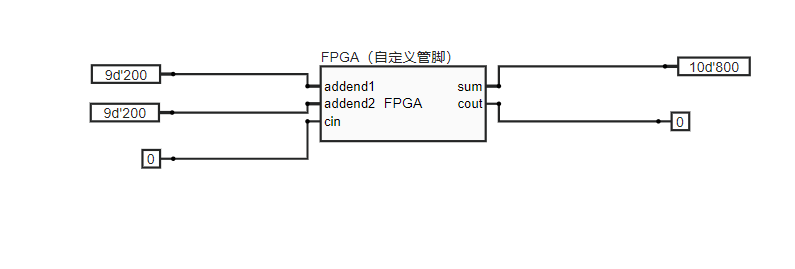
* 1. 无进位输入的情况



\* 因为输入的管脚限制，可以输入的最大的两个数相加，也不会超过可以输出的最大值，因此输出的进位一直为零。

## 实验中遇到的问题及解决方法

1. 问题1：两个加法相加没法得出正确的数值，在结果为6位二进制的表示范围内，答案是正确的，超出了就错误了
2. 问题2：cin为一结果增加了256
3. 问题3：结果是翻倍的



解决方法：增大输入的位数，将位数增大后问题解决。

# 实验四、序列检测器

## 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。
3. 逐个比特比较的序列检测分为两种情形：

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=101，并在输出Z的指定位置上显示。

X = 1 0 1 0 1 0 1 0

Z = 0 0 1 0 0 0 1 0

也就是说，在Z中包含了不重复的特定序列Y有2个。

②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=101，并在输出Z的指定位置上显示。

X = 1 0 1 0 1 0 1 0

Z = 0 0 1 0 1 0 1 0

也就是说，在Z中包含了重复的特定序列Y有3个。

## 设计思路及程序功能框图

* 1. 设计思路
     1. 需要检测的序列有八位，当连续输入“10”时，输出为1，其余情况输出为0。
     2. 根据检测要求，只需要记录一位的输入的序列就可以完成检测。
     3. 检测需要分为重复检测和不重复检测，但因为需要检测的特定序列为“10”，只有两位，且第一位与第二位不同，导致重复检测与不重复检测的结果相同，因此，只需要进行一种类型的检测即可。
     4. 本题中采用重复判断

通过拨挡开关K1～K8，输入一个八位二进制数据X=10101010。

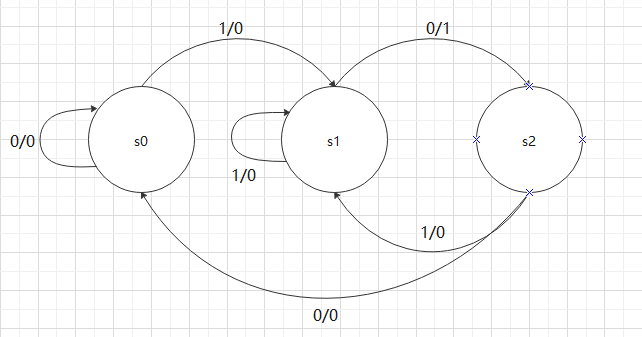
检测在X中包含了几个重复的特定序列Y=10，并在输出Z上显示。

X = 1 0 1 0 1 0 1 0

Z = 0 1 0 1 0 1 0 1

也就是说，在Z中包含了重复的特定序列Y有4个。

* + 1. 在LED灯上显示检测的结果。在七位数码管上显示特定序列的个数。
  1. 状态图



* 1. 状态迁移表

|  |  |  |  |
| --- | --- | --- | --- |
| 初态  次态/输出  输入 | S0 | S1 | S2 |
| 0 | S0/0 | S2/1 | S0/0 |
| 1 | S1/0 | S1/0 | S1/0 |

* 1. 对状态进行编码

设置两个状态 s1 = 0; s2 = 1;

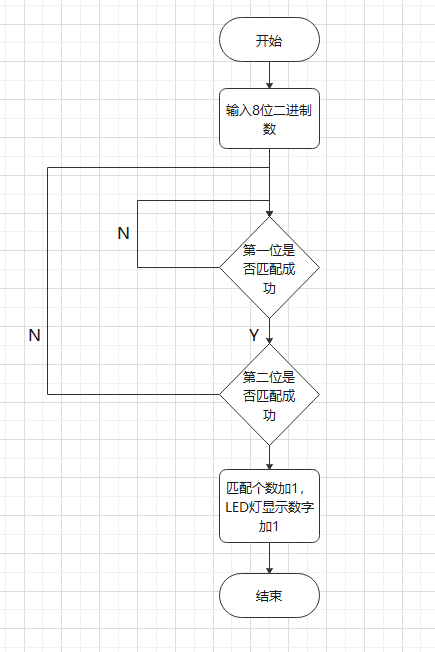
state 为当前位数值;

next 为下一位数值;

num 为需要输入到七位数码管的数字

cout 为输入到单个LED灯的信号

* 1. 程序功能框图



## 源代码

module desgin1(

input [7:0] kaiguan, //k1-k8 开关

input rst,clk, //复位，兼运行按钮

output [6:0]seven\_LED, //七位数码管

output [7:0]LED

);

reg n; //n为输入的每一位，

reg next;

reg cout=0;

reg state; //当前位置的值

integer i = 0; //判断循环使用

reg [4:0] num = 0; //匹配成功的次数

reg [6:0]seven\_LED;

reg [7:0]LED;

parameter s1 = 0;

parameter s2 = 1;

always@(posedge clk)//按下开关后触发

begin

if(rst) //如果重置

begin

num = 0;

state = kaiguan[0]; //第一位

LED= 2'b00000000;

end

else //如果不重置

begin

num = 0;

state = kaiguan[0];

LED[0]=0;

for(i = 1;i < 8;i = i+ 1 )

begin

n = kaiguan[i];

case(state)

2'b0:if(n==1)begin

next = s2;

cout = 0;

end

else //n为0，下一位的state也是0

begin

next = s1;

cout = 0;

end

2'b1: if(n==0)begin

next = s1; //下一位state为s1

cout = 1;

num = num + 1;

end

else

begin

next = s2;

cout = 0;

end

endcase

state=next;

LED[i]=cout; //对应LED灯的显示与否

end

end

case(num)

4'd0:seven\_LED = 7'b0111111; //将输入的数字，翻译成在数字管上显示正确的

4'd2:seven\_LED = 7'b1011011;

4'd3:seven\_LED = 7'b1001111;

4'd4:seven\_LED = 7'b1100110;

4'd5:seven\_LED = 7'b1101101;

4'd6:seven\_LED = 7'b1111101;

4'd7:seven\_LED = 7'b0000111;

4'd8:seven\_LED = 7'b1111111;

4'd9:seven\_LED = 7'b1101111;

default:seven\_LED = 7'bx;

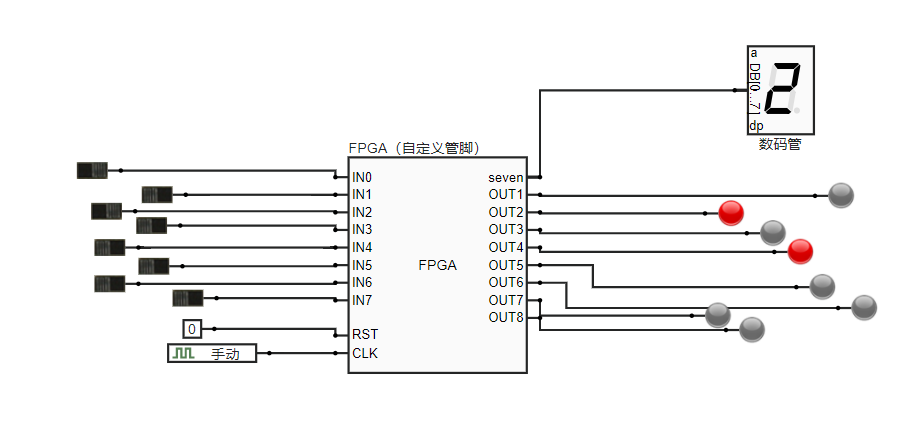
endcase;

end

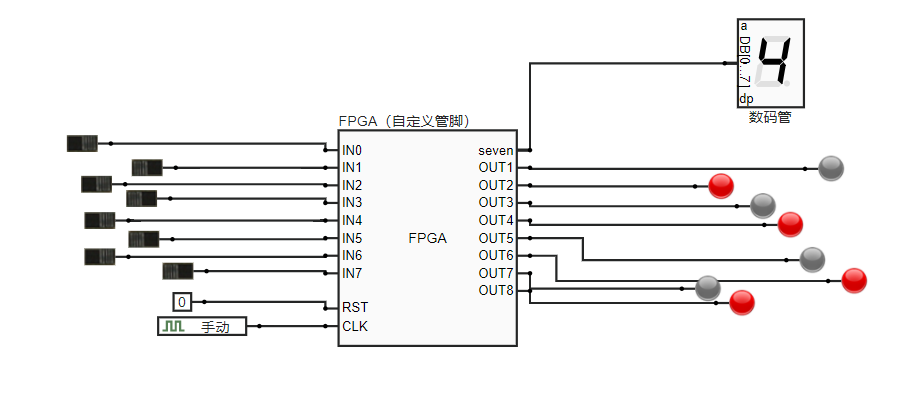
endmodule

## 实验结果

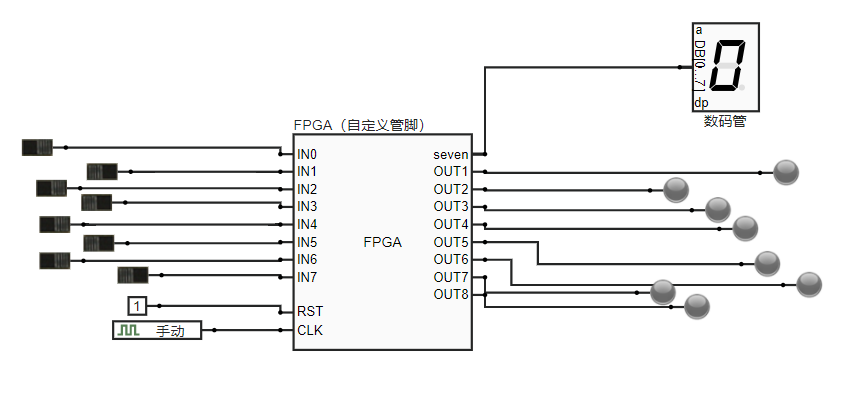
* 1. 输入的二进制码为10100000：输出为01010000，数码管显示为2



* 1. 输入的二进制码流为10101010：输出为01010101，数码管显示为4



* 1. 复位端置为一，器件复位



# 实验五、简单CPU设计

## CPU指令系统

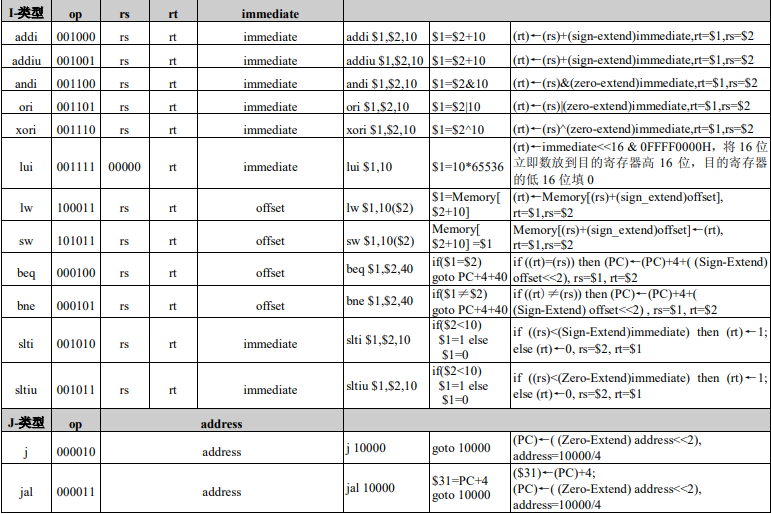
运算类指令包括算术运算、逻辑运算、移位运算等，转移类指令包括无条件转移和有条件转移两大类，其它类型指令包含停机指令等。

指令的长度为32位，分为三种类型的指令：

1. R型指令有6位操作码，6位功能码，rs, rt, rd三个不同寄存器的地址码，shamt移位量。
2. I型指令有操作码和立即数。
3. J型指令有操作码和地址码。

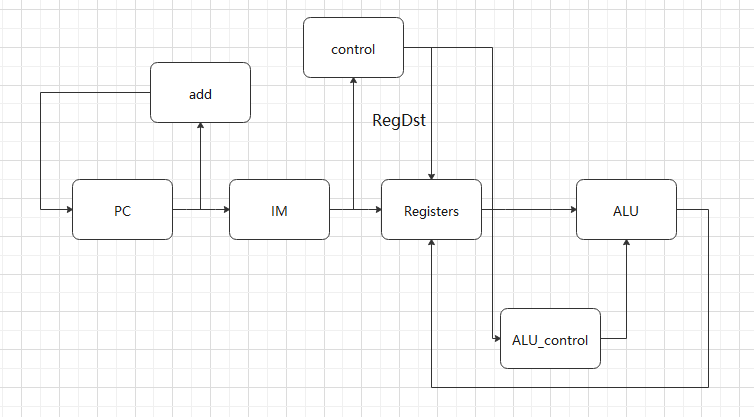
本次实验中实现了6条不同类型的指令。



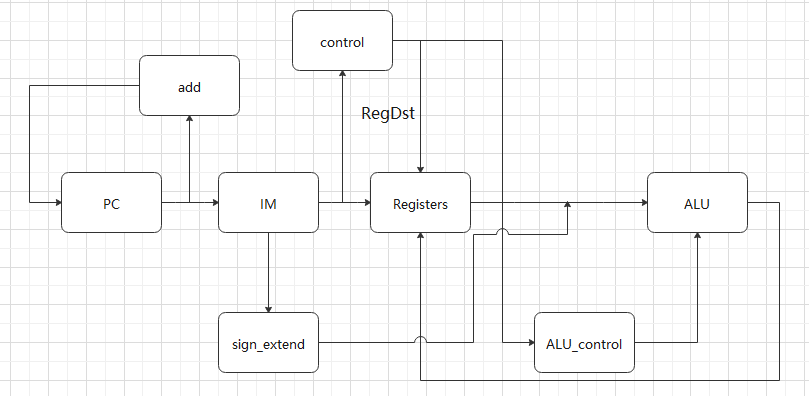


## 每一类的一条CPU指令的实现

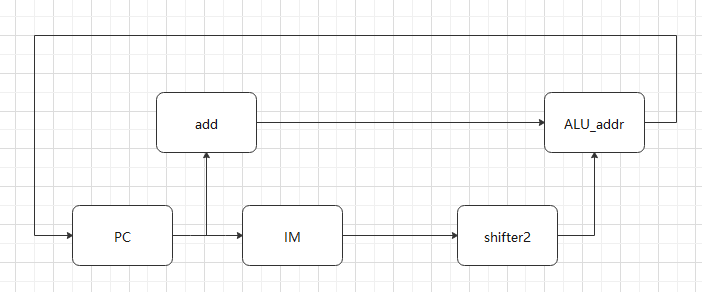
1. R型指令
   * 1. 指令名称：add
     2. 指令描述： (rd)←(rs)+(rt),rs=$2,rt=$3,rd=$1
     3. 指令功能：从寄存器中取出rs，rt地址对应的数据，运算后存入地址rd
     4. 指令格式：6位操作码，6位功能码，rs、rt、rd地址码，移位码
     5. 寻址方式：直接寻址
     6. 指令的执行流程



1. I型指令
   1. 指令名称：addi
   2. 指令描述：(rt)←(rs)+(sign-extend)immediate,rt=$1,rs=$2
   3. 指令功能： 将立即数进行符号扩展与rs的数据相加存入rt
   4. 指令格式：6位操作码，rs、rt地址码，15位立即数
   5. 寻址方式：直接寻址
   6. 指令的执行流程

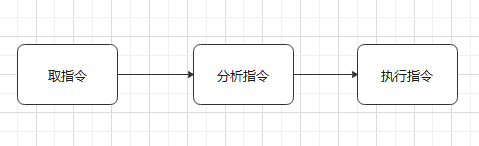


1. j型指令
   1. 指令名称：j
   2. 指令描述： (PC)←((Zero-Extend) address<<2), address=10000/4
   3. 指令功能：将地址码零扩展左移两位，与PC的高四位组合存入PC
   4. 指令格式：6位操作码，26位地址码
   5. 寻址方式：直接寻址
   6. 指令的执行流程

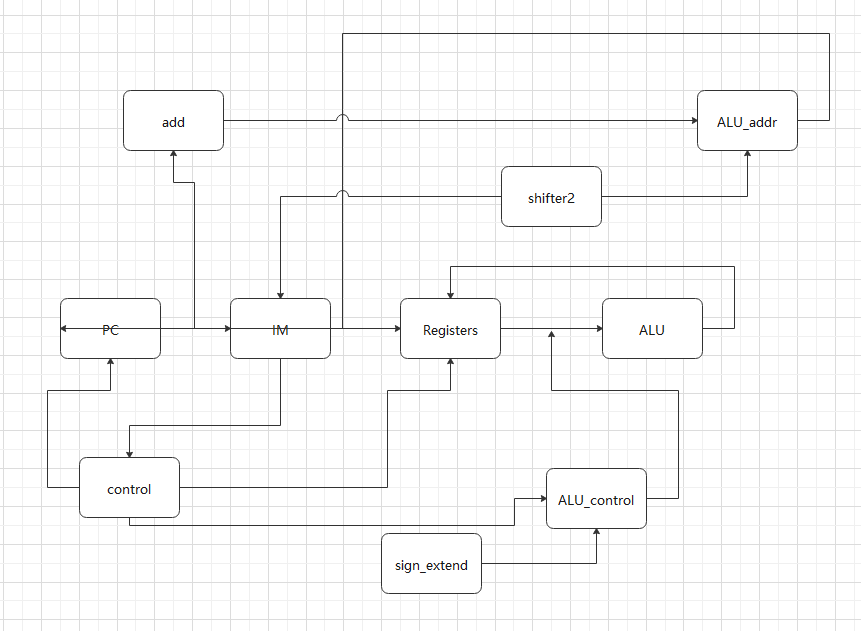


## 指令实现数据流图

指令执行流程：

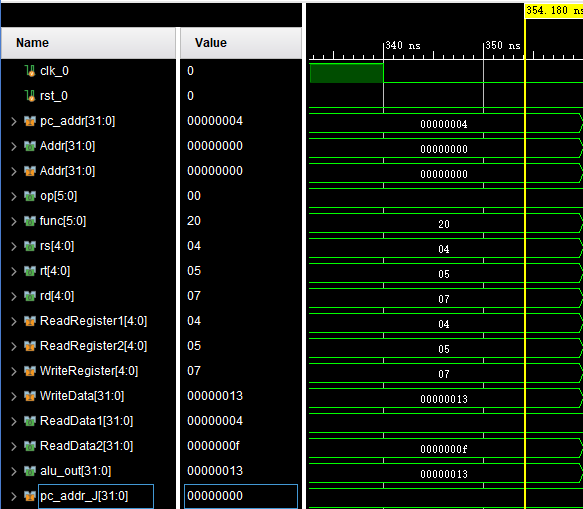


数据流图：

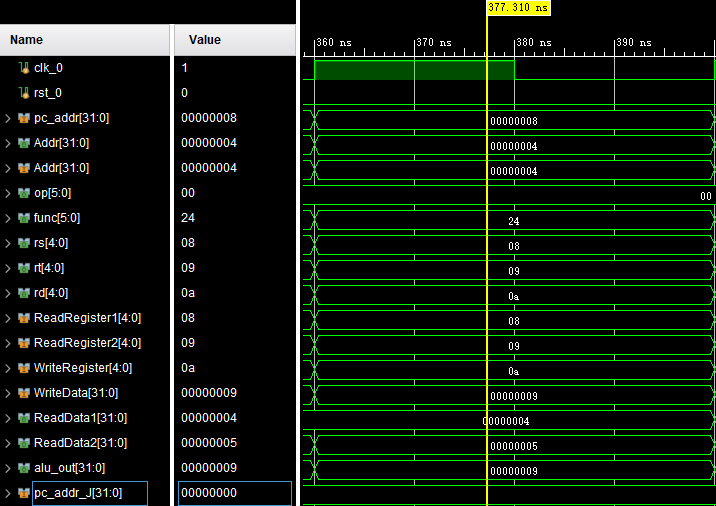


## CPU指令的测试与仿真

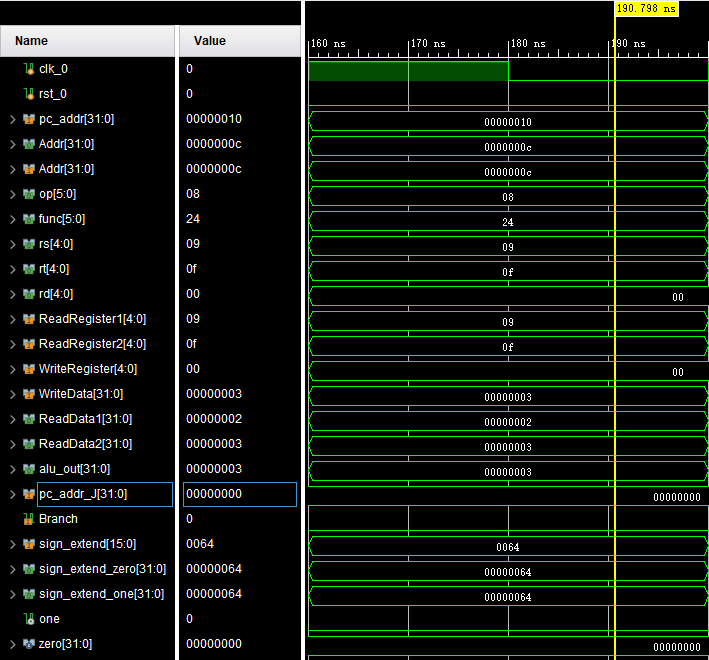
1. and



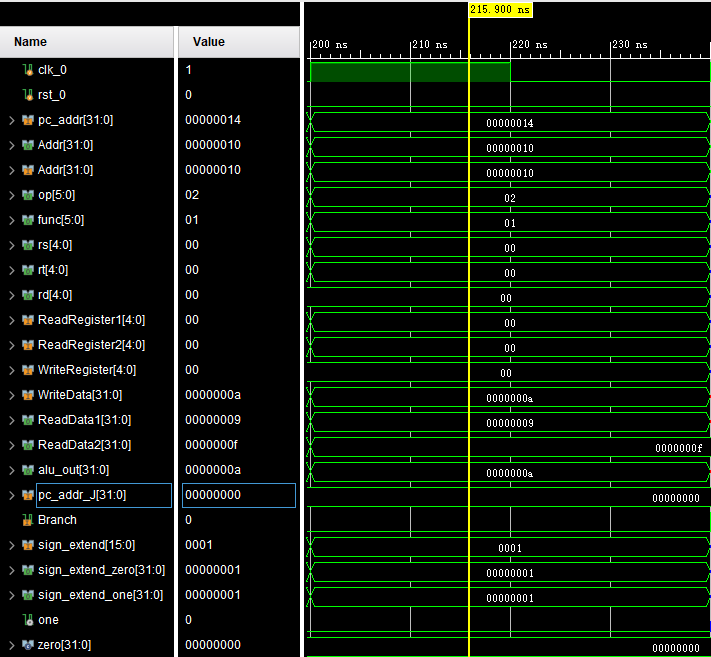
1. or



1. addi



1. j



## 源代码

* 1. 指令存储器（IM）

module IM(

input[31:0]Addr,

output[5:0]op,

output[5:0]func,

output [4:0]rs,

output [4:0]rt,

output [4:0]rd,

output [15:0]sign\_extend,

output [26:0]J\_addr);

wire[31:0]Inst;

wire[31:0]Rom[31:0];

wire[5:0]op;

wire[5:0]func;

wire [4:0]rs;

wire [4:0]rt;

wire [4:0]rd;

wire [15:0]sign\_extend;

assign Rom[5'h00]=32'b000000\_00100\_00101\_00111\_00000\_100000;//add

assign Rom[5'h01]=32'b000000\_01000\_01001\_01010\_00000\_100100;//and

assign Rom[5'h02]=32'b000000\_00010\_00011\_00001\_00000\_100101;//or

assign Rom[5'h03]=32'b001000\_01001\_01111\_00000\_00001\_100100;//addi

assign Rom[5'h04]=32'b000010\_00000\_00000\_00000\_00000\_000001;//J型

assign Inst =Rom[Addr[6:2]];

assign op = Inst[31:26];

assign func = Inst[5:0];

assign rs = Inst[25:21];

assign rt = Inst[20:16];

assign rd = Inst[15:11];

assign sign\_extend = Inst[15:0];

assign J\_addr = Inst[26:0];

endmodule

* 1. 寄存器堆（Registers）

module Registers(

input [4:0]ReadRegister1,

input [4:0]ReadRegister2,

input [4:0]WriteRegister,

input [31:0]WriteData,

input RegDst,

input rst,

input clk,

output [31:0]ReadData1,

output [31:0]ReadData2

);

reg [31:0]ReadData1;

reg [31:0]ReadData2;

reg [31:0]mem3[31:0];

reg [31:0]mem1[31:0];

reg [31:0]mem2[31:0];

reg [4:0]WR;

integer i;

always@(posedge clk)

begin

if(rst)

begin

for(i=1;i<33;i=i+1)

begin

mem1[i]=i;

mem2[i]=i;

end

end

else

begin

ReadData1 = mem1[ReadRegister1];

WR = WriteRegister;

if(RegDst == 1)

ReadData2 = mem1[ReadRegister2];

mem3[WR] = WriteData; //需要写入

if(RegDst ==0)

mem2[WR] = WriteData;

end

end

endmodule

* 1. 程序计数器（PC）

module PC(

input [31:0]pc\_addr,

input rst,

input clk,

output [31:0]Addr

);

reg [31:0]Addr;

initial

begin

Addr = 0;

end

always@(posedge clk)

begin

if(rst)

Addr = 32'b0;

else

begin

Addr = pc\_addr;

end

end

endmodule

* 1. 地址累加器（add）

module add(

input [31:0]Addr,

output [31:0]PC\_Addr

);

wire [31:0]PC\_Addr;

assign PC\_Addr = Addr+ 4;

endmodule

* 1. 移位器（shifter2）

module shifter2(

input [26:0] J\_addr,

input [31:0] pc\_addr,

output [31:0] pc\_addr\_J);

parameter zero=2'b00;

assign pc\_addr\_J = {pc\_addr[31:29],J\_addr[26:0],zero};

endmodule

* 1. 运算器（ALU）

module ALU(

output [31:0]alu\_out,

input [31:0]data1,

input [31:0]data2,

input [3:0]func //来自ALUcontrol

);

parameter ADD = 6'b0010,

AND = 6'b0000,

OR = 6'b0001;

reg [31:0]alu\_out;

always @(data1 or data2)

begin

casex(func)

ADD:alu\_out=data1 + data2;

AND:alu\_out = data1 & data2;

OR :alu\_out = data1 | data2;

default:alu\_out<=32'bxxxx\_xxxx;

endcase

end

endmodule

* 1. 运算器控制器（ALU\_control）

module ALU\_control(

input [5:0]op,

input [5:0]func,

input clk,

output [3:0]ALU\_func

);

reg [3:0]ALU\_func;

parameter ADD = 4'b0010,

AND = 4'b0000,

OR = 4'b0001;

always@(posedge clk)

begin

if(op==6'b000000)

begin

case(func)

6'b100000:ALU\_func = ADD;

6'b100100:ALU\_func = AND;

6'b100101:ALU\_func = OR;

endcase

end

else //i型指令，读op码，生成功能码

begin

case(op)

6'b001000:ALU\_func = ADD;

6'b001100:ALU\_func = AND;

endcase

end

end

endmodule

* 1. 控制器（control）

module control(

input [5:0]op,

input clk,

output ALUSrc, //失效时，ALU的第二个操作数来自寄存器堆输出，生效时来自经过符号扩展的指令的低16位

output RegDst, //失效时控制寄存器写的目的寄存器的序号为【20：16】，生效是写入字段是【15：11】

output Se, //置1为符号扩展，置0为零扩展

output Branch //置1为J指令移动地址

);

reg ALUSrc;

reg RegDst;

reg Se;

reg Branch;

always@(posedge clk)

begin

if(op == 6'b000000) //R型指令

begin

ALUSrc = 0;

RegDst = 1;

Se = 0;

Branch = 0;

end

else //i型指令

begin

ALUSrc = 1; //多位选择器

RegDst = 0;

if(op==6'b001000 | op==6'b001001)

begin

Se=1;

Branch = 0;

end

else if(op==6'b001100 | op==6'b001110)

begin

Se=0;

Branch = 0;

end

else if(op==6'b000010)

begin

Se = 0;

Branch = 1;

end

end

end

endmodule

* 1. 信号扩展器（sign\_extend）

module sign\_extend(

input [15:0]sign\_extend,

output [31:0]sign\_extend\_zero,

output [31:0]sign\_extend\_one

);

parameter zero = 16'b0; //16位0

wire one = {16{sign\_extend[15]}};

assign sign\_extend\_zero = {zero, sign\_extend};

assign sign\_extend\_one = {one, sign\_extend};

endmodule

* 1. 多位信号扩展选择器（MUX\_sign\_extend\_data）

module MUX\_sign\_extend\_data(

input Se,

input [31:0]sign\_extend\_zero,

input [31:0]sign\_extend\_one,

output [31:0]data

);

function [31:0] select;

input [31:0]sign\_extend\_zero;

input [31:0]sign\_extend\_one;

input Se;

case(Se)

0:select=sign\_extend\_zero;

1:select=sign\_extend\_one;

endcase

endfunction

assign data=select(sign\_extend\_zero,sign\_extend\_one,Se);

endmodule

* 1. 多位数据选择器（MUX\_data\_choice）

module MUX\_data\_choice(

input [31:0]data2,

input [31:0]data\_sign\_extend,

input ALUSrc,

output [31:0]data\_final

);

function [31:0] select;

input [31:0]data2;

input [31:0]data\_sign\_extend;

input ALUSrc;

case(ALUSrc)

0:select=data2;

1:select=data\_sign\_extend;

endcase

endfunction

assign data\_final = select(data2,data\_sign\_extend,ALUSrc);

endmodule

* 1. 多位地址选择器（MUX\_addr）

module MUX\_addr(

input [31:0]pc\_addr,

input [31:0]pc\_addr\_J,

input Branch,

output [31:0]pc\_addr\_final

);

function [31:0] select;

input [31:0]pc\_addr;

input [31:0]pc\_addr\_J;

input Branch;

case(Branch)

0:select=pc\_addr;

1:select=pc\_addr\_J;

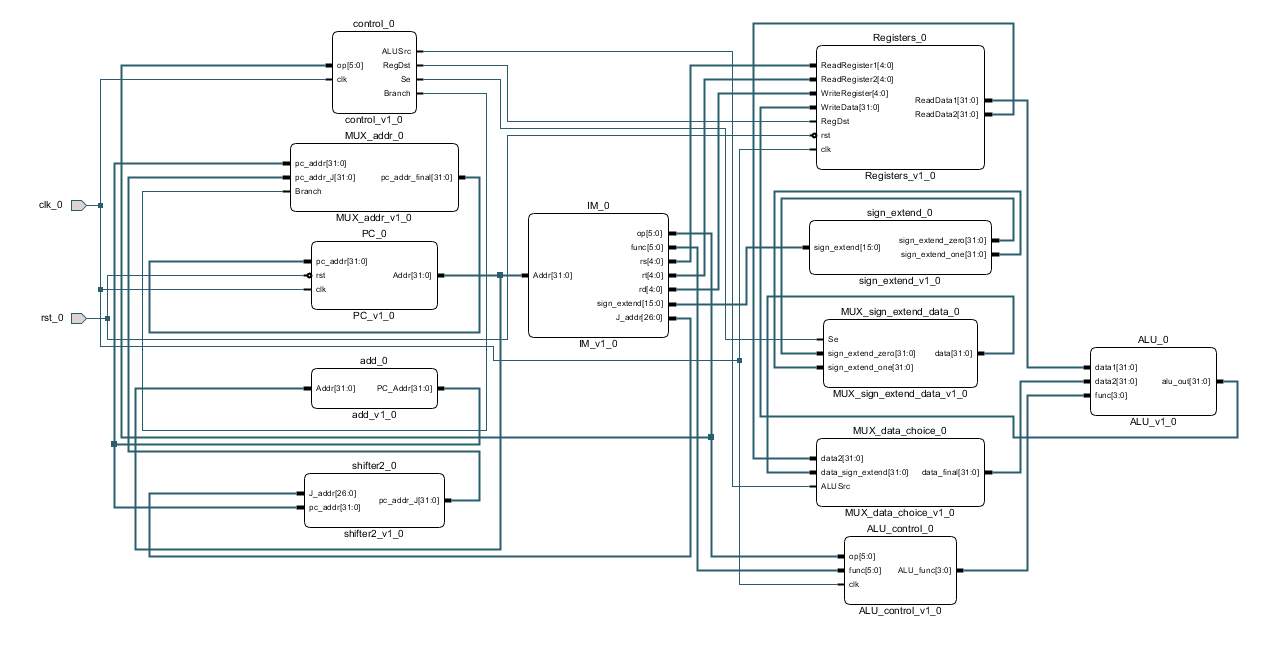
endcase

endfunction

assign pc\_addr\_final = select(pc\_addr,pc\_addr\_J,Branch);

endmodule

* 1. Block design



# 实验设计心得

通过这次实验，学习了Vivado的操作和Verilog语言的使用。掌握了Verilog分层次设计方法。能够将程序封装成IP，并调用自己的IP，能够用有限状态机描述硬件问题。通过前四个实验，让我了解了怎么控制基础的硬件器件，怎么让数码管显示对应的数字，通过时序控制16×16点阵显示名字，设计加法器等。在计算机组成原理课上学习了计算机的指令系统，通过CPU的设计，让我更加深入的了解了计算机的指令系统，了解三种不同类型指令的设计原理，学会了从上到下的CPU设计思路，自己规划并通过调用自封装的IP构建数据通路，实现了部分指令的功能。在实验中也遇到了很多的问题，对于Verilog语言的使用还不够熟练，语法不够严谨，算法不够优化。其中的七位数码管消耗的输出管脚非常多，经过改进，通过时序将管脚进行了缩减。CPU的设计中，没有在开始就对设计思路有一个全面的规划，导致后面在设计IP的时候，很多功能和管脚没有设计好，导致实验的过程很长。对于Verilog语言和Vivado的使用还有很大的提升空间。日后会继续强化编程和IP设计能力。